Escuela
Politécnica
Superior

Detección basada en datos 3D de objetos urbanos

Grado en Ingeniería Robótica

Trabajo Fin de Grado

Autor:
Julio Castaño Amorós
Tutor/es:
Miguel Ángel Cazorla Quevedo
Francisco Gómez Donoso

Julio 2020

Universitat d'Alacant
Universidad de Alicante

Detección basada en datos 3D de objetos urbanos

Autor
Julio Castaño Amorós

Tutor/es
Miguel Ángel Cazorla Quevedo
CIENCIA DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL
Francisco Gómez Donoso
CIENCIA DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

Grado en Ingeniería Robótica

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2020

Preámbulo

La elección de desarrollar mi Trabajo de Fin de Grado sobre Deep Learning viene dado, sobre todo, al gran interés que despierta en mí este campo. A lo largo del grado se nos ha introducido en machine learning/deep learning en varias asignaturas, este fue el hecho que me motivó a ampliar conocimientos de forma autónoma.

Mi Trabajo de Fin de Grado consiste en aplicar Deep Learning en conducción autónoma, un campo en el que todavía queda mucho por recorrer hasta que se considere superado. Concretamente, este proyecto tiene como finalidad corregir los errores de los sistemas de detección 2D a partir de datos 3D.

Agradecimientos

En primer lugar, me gustaría dar las gracias a mis tutores, Miguel y Fran, por darme la oportunidad de realizar este trabajo con ellos, sobre todo teniendo en cuenta las fechas en las que empezamos este proyecto y la situación tan complicada en la que nos encontramos por el COVID-19. Ambos han estado al pie del cañón, tanto con la búsqueda de un proyecto innovador y que diera tiempo a entregar, como con la resolución de cualquier problema que ha surgido durante el desarrollo del proyecto. Trabajar con ellos ha sido un placer.

No puedo olvidar mi agradecimiento hacia Jorge Pomares. Aunque el proyecto que teníamos previsto no pudo realizarse como esperábamos, siempre me ha ayudado en lo que ha podido, ha estado pendiente de mí durante muchos meses de trabajo y he aprendido mucho trabajando con él.

Dar las gracias también a todos mis compañeros de clase por hacer de estos cuatro años una experiencia increíble que no olvidaré jamás. Más que compañeros de clase, son una familia.

Por último, agradecer a mi familia y a mi novia por todo el apoyo que me han dado durante la carrera y, sobre todo, durante este último año. Sólo ellos saben lo difícil que ha sido para mí realizar este Trabajo de Fin de Grado. Mamá, papá, Erick, Gloria, gracias a vosotros por enseñarme a no rendirme nunca.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

La conducción autonóma es un tema cada vez más frecuente en el sector automovilístco. Desde hace unos años, es un problema que ha dejado de ser una mera imaginación gracias a la unión de diferentes tecnologías como son el control, fusión sensorial, inteligencia artificial...

Como su propio nombre indica, la conducción autónoma consiste en un vehículo que sea capaz de navegar por sí mismo gracias a los sensores que dispone. Estos sensores tienen una alta importancia debido a que son la percepción del vehículo autónomo.

Teniendo en cuenta que para los humanos conducir puede ser una tarea compleja en un principio, es difícil entender cómo un coche va a poder hacerlo de forma autónoma, cómo va a poder respetar las leyes de tráfico, por ejemplo, que sepa cuándo tiene que parar en un semáforo o dejar pasar a un peatón.

Para resolver este problema se utiliza la detección de objetos mediante Deep Learning. Desde que, en 2012, Alex Krizhevsky et al [16] superaran con AlexNet los resultados de las técnicas tradicionales para la clasificación de objetos en imágenes en ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [3], se empezó a invertir e investigar en Deep Learning hasta el punto de que se considera resuelto el problema de la clasificación de objetos en ese desafío.

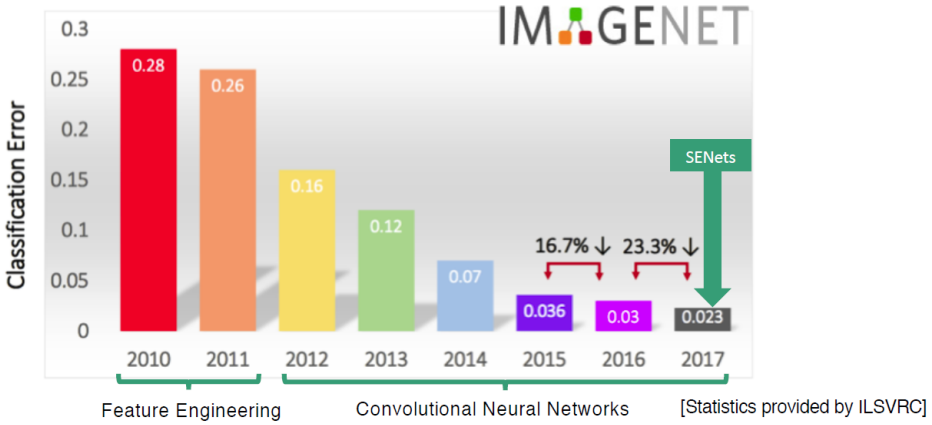

Figura 1.1: Ganadores del challenge de Imagenet hasta 2017. Fuente: [27]

Por lo tanto, la aplicación de Deep Learning en conducción autónoma está clara, se va a utilizar para detectar cualquier obstáculo que sea necesario para realizar una conducción adecuada, ya sea peatones, coches, señales, semáforos...

Las personas tienen la capacidad de intuir a qué distancia puede estar un objeto o si un objeto está más cerca de otro. Esto se utiliza a la hora de conducir para tener en cuenta las señales de tráfico, los peatones, la distancia de seguridad... Entonces, un coche autónomo también va a necesitar conocer la distancia a los obstáculos que tiene alrededor. Por esta razón, solo con reconocimiento de objetos en imágenes 2D no es suficiente.

Este es el hecho por el cual los coches autónomos disponen de sensores 3D que recogen nubes de puntos tridimensionales. Utilizando técnicas de Deep Learning similares a las que se utilizan en imágenes 2D, se pueden reconocer objetos tridimensionales. Por lo tanto, ya se dispone de un sistema que es capaz de detectar un objeto y tener la información sobre la profundidad para tomar decisiones.

Se podría pensar que el problema de la conducción autónoma está resuelto, el vehículo autónomo recoge información 2D y 3D de la cual reconoce los obstáculos y actúa en consecuencia. Pues no es que aún no esté resuelto, sino que aún queda bastante tiempo hasta que pueda considerarse resuelto. Problemas como el precio de los sensores, la ética y los fallos de seguridad, fallos en la detección..., frenan la evolución de la conducción autónoma.

Este proyecto se centra en mejorar un problema con la detección. Anteriormente, se ha comentado que la clasificación de objetos en imágenes 2D se considera resuelta, pero cuando se utilizan estos sistemas para detectar en un vehículo autónomo, se producen falsos positivos en la detección. Los errores más comunes que se producen son, por ejemplo, cuando el sistema predice como persona o vehículo cuando lo detecta en un cartel publicitario. El sistema reconoce que es una persona/vehículo, pero en realidad no lo es. Otro ejemplo es cuando el objeto se ve reflejado: si un coche está reflejado en alguna parte el sistema va a predecirlo como positivo cuando en realidad no lo es.

En las figuras, Figura 1.2 y Figura 1.3, se puede observar los errores comentados.

Figura 1.2: Detección errónea de Yolo. Fuente: [8]

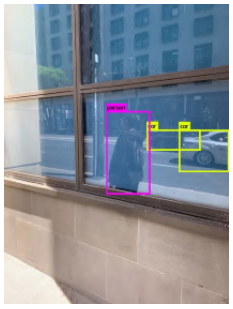

Figura 1.3: Detección errónea de Yolo. Fuente: [8]

Por tanto, se propone un pipeline en el que una red neuronal convolucional 3D corrige estos falsos positivos detectados por el sistema 2D.

2 Marco Teórico

El marco teórico sobre el problema de detección 3D de objetos se engloba en dos tipos de entornos en los que se aplica esta detección, *indoor* y *outdoor*.

2.1 Entornos indoor

En un entorno indoor, los objetos a detectar son aquellos que se encontrarían en entornos cerrados, como por ejemplo: mesas, sillas, puertas, sofás... Se mencionan trabajos que obtuvieron buenos resultados en los challenges de los datasets ModelNet [1] y SUNRGB-D [24].

ModelNet es un dataset de gran escala formado por los modelos CAD 3D de diferentes objetos. Dentro de ModelNet hay dos subdatasets, ModelNet10 y ModelNet40, donde tienen 10 y 40 clases, respectivamente. SUNRGB-D es un dataset con imágenes RGB-D cuyo objetivo era el avance en las tareas de entendimiento de la escena como reconocimiento de objetos, segmentación semántica...

En cuanto a ModelNet, se consideran estado del arte proyectos como **VRN Ensemble** [1], **PointNet** [9] o **Par3DNet** [11] debido a los resultados obtenidos en el challenge. Estas aproximaciones trabajan con una representación voxelizada de las nubes de puntos y predicen las categorías a partir de los voxels mediante una red neuronal convolucional profunda 3D. Aunque, por ejemplo, VRN Ensemble se encuentra en el primer puesto del challenge, las operaciones de voxelizado que utiliza son demasiado complejas computacionalmente para la ejecución en tiempo real.

Por otro lado, **LonchaNet** obtiene tres secciones del modelo, una por cada eje, y proyecta cada una sobre una imagen 2D, después pasará cada proyección por una GoogleLeNet y se juntan las tres en una capa totalmente conexa. LonchaNet no se ve afectada por el tiempo de ejecución, pero tiene problemas cuando las categorías representan objetos similares como una mesa normal y una mesa de escritorio o una mesita de noche y una mesa vestidor. Este problema se produce porque las secciones de los modelos son muy similares.

Pasando a SUNRGB-D, se podría destacar **Frustum PointNet** [18], **3D SS** [25], entre otros. Ambos trabajan con datos RGB-D ya que establecen sus resultados en torno al dataset SUNRGB-D como se ha comentado anteriormente.

En primer lugar, **Frustum PointNet** obtiene un set de bounding boxes 2D candidatos mediante una red neuronal convolucional 2D. De cada región 2D candidata se obtiene una nube de puntos que está formada por todos los puntos que entran dentro del campo de visión

[1] http://modelnet.cs.princeton.edu/

en el que se encuentra el objeto. Estas nubes de puntos entran a una red PointNet[2] que va a predecir la clase del objeto. Cabe destacar que este modelo tiene problemas cuando hay más de un objeto de la misma categoría dentro del campo de visión, ya que el bounding box 3D que devuelve de cada objeto está mezclado con el otro. Además, depende de la detección 2D porque si el detector 2D no funciona correctamente (poca luz, oclusiones...), la detección 3D obtendrá malos resultados.

En segundo lugar, **3D SS** utiliza la misma técnica de candidatos pero en tres dimensiones, obtiene los bounding boxes tridimensionales mediante una Region Proposal Network (RPN). A partir de los bounding boxes, obtiene sus proyecciones en 2D y entrena una red neuronal convolucional híbrida de manera que los bounding boxes son entrada a una Red Neuronal Convolucional 3D (3DCNN) y las proyecciones a una Red Neuronal Convolucional 2D (2DCNN). La RPN tiene problemas con objetos planos como televisores, monitores..., por lo tanto, afecta a la detección porque, sino se obtiene un buen bounding box 3D, no funcionará bien la detección. También falla con objetos cuya categoría tiene mucha varianza, por ejemplo la categoría caja.

2.2 Entornos outdoor

Para el entorno outdoor, se toma como referencia el dataset **KITTI** [10]. KITTI es un dataset estado del arte de objetos urbanos. Las clases principales que se suelen utilizar para testear los modelos son peatones, ciclistas y coches.

Como este dataset es el más utilizado para detección de objetos urbanos, podemos encontrar muchísimas propuestas, como por ejemplo:

- **MV3D**: Multi-View 3D Object Detection Network for Autonomous Driving [2].
- **Vote3Deep**: Fast Object Detection in 3D Point Clouds Using Efficient Convolutional Neural Networks [5].
- **F-PointNet**: Frustum PointNets for 3D Object Detection from RGB-D Data [18].

MV3D es un pipeline que tiene como entrada datos 3D desde diferentes perspectivas (vista de pájaro y vista frontal) y la imagen RGB. Está compuesto de dos fases, una primera fase que consiste en una 3D Proposal Network que va a utilizar la vista de pájaro para generar bounding boxes 3D candidatos. La segunda fase es una Region-based Fusion Network. Se proyectan los bounding boxes tanto en las dos vistas como en la imagen y se obtienen las regiones correspondientes a los bounding boxes. Por último, se extraen características de las tres vistas, se fusionan, y se envían como entrada a un clasificador. Aunque la vista de pájaro tiene alta precisión para generar los bounding boxes, tiene problemas con objetos pequeños como ciclistas y peatones, también tiene dificultades con escenas en las que hay muchos objetos moviéndose en dirección vertical.

[2]La arquitectura PointNet se explicará en detalle en el apartado Metodología.

Vote3Deep, en cambio, sólo trabaja con las nubes de puntos representadas como un grid 3D. Este modelo utiliza una 3DCNN junto con un algoritmo de votación y regularización L1. El algoritmo de votación consiste en hacer las convoluciones, pero solo en los elementos que son distintos de cero y obteniendo el mismo resultado que una convolución normal, pero más eficiente. Es rápido, aunque se podría aumentar la velocidad de predicción si se desarrollara una implementación del algoritmo de voto en GPU. Además, presentan dos modelos, uno para la detección de coches y otro para la detección de peatones y ciclistas, porque necesitan cambiar el tamaño del kernel en la última capa debido al tamaño de los coches comparado con el de los peatones o ciclistas. Por lo tanto, si presentaran un único modelo para las 3 clases, los resultados se verían afectados.

F-PointNet ya ha aparecido como ejemplo en un entorno indoor pero cabe destacar que también obtiene buenos resultados en un entorno outdoor.

En resumen, las diferentes aproximaciones explicadas anteriormente poseen desventajas en comparación con la propuesta de este proyecto tales como requisitos en cuanto al hardware para la ejecución en tiempo real, algunas aproximaciones están basadas en proyecciones 2D que no son posibles con nubes de puntos o precisan de una disposición específica de los sensores ya que requieren de diferentes vistas de un mismo objeto para la detección.

3 Objetivos

Con este Trabajo Final de Grado (TFG) se alcanzan los siguientes objetivos:

- **Entender la metodología del Deep Learning**: conocimiento del framework, implementación de redes, nociones teóricas de los diferentes aspectos que engloban las redes neuronales convolucionales en general.
- **Conocer las principales herramientas y modelos en detección de objetos 3D**: introducción al mundo de la investigación, entender las técnicas utilizadas por los modelos que están considerados estado del arte.
- **Entender los fundamentos de calibración de sensores**: sincronización cámara-LIDAR para obtención de nubes de puntos a partir de detección 2D.
- **Manejar los conceptos de ROS para incorporar el trabajo en un pipeline en ROS**: manejar publicadores, suscriptores, servicios, tipos de mensajes...
- **Introducción a la detección 3D en conducción autónoma**: comparación de diferentes técnicas según los recursos necesarios para la ejecución en un coche autónomo.

4 Metodología

En este apartado se van a explicar todas las herramientas utilizadas en el proyecto.

4.1 Setup

En cuanto al setup, se han utilizado las siguientes herramientas:

- Sistema operativo Ubuntu 16.
- ROS Kinetic.
- Lenguajes de programación Python 2.7.12 y C++ (gcc5).
- Tensorflow 1.4.0.
- OpenCV 3.3.1-dev.
- Cuda v8.0.61 y cuDNN v6.
- Tarjeta gráfica Nvidia GeForce GTX 1080 Ti.

4.2 ROS

Es importante explicar qué es ROS y cómo funciona para entender mejor el funcionamiento del pipeline que engloba todo el proyecto.

4.2.1 ¿Qué es ROS?

Robot Operating System (ROS) [7] es un meta-sistema operativo open-source que fue diseñado para su uso con robots. Actualmente solo funciona sobre plataformas basadas en el sistema operativo Unix.

ROS proporciona los servicios básicos de un sistema operativo como pueden ser la abstracción del hardware, el control de dispositivos de bajo nivel, la implementación de funcionalidad de uso común, el paso de mensajes entre procesos y el manejo de paquetes.

4.2.2 Objetivos

Ros tiene como objetivo principal apoyar la reutilización de código en investigación y desarrollo en robótica. Además, también tiene objetivos secundarios que buscan complementar el objetivo principal:

- ROS está diseñado para poder reutilizar código en otros frameworks de robótica software.
- ROS tiene como objetivo la independencia de lenguaje, ya hay librerías implementadas en Python, C++ y Lisp.
- ROS es escalable.
- ROS permite un testeado más rápido al facilitar el aislamiento de los diferentes componentes del sistema.

4.2.3 Conceptos

ROS dispone de niveles conceptuales que se pueden dividir en:

- Sistema de archivos.
- Grafo de computación.

4.2.3.1 Sistema de archivos

Los conceptos sobre el sistema de archivos de ROS hacen referencia a aquellos recursos que se encuentran en el disco, tales como:

- **Paquetes:** un paquete es la unidad principal para la organización de software en ROS. Los paquetes pueden contener procesos de ejecución ROS, conocidos como nodos, librerías dependientes de ROS, por ejemplo roscpp o rospy, datasets, archivos de configuración...
- **Metapaquetes:** los metapaquetes son paquetes que están especializados para representar un grupo de paquetes relacionados.
- **Manifiestos del paquete:** los manifiestos hacen referencia a los ficheros xml. Estos proporcionan metadatos sobre un paquete, incluyen su nombre, versión, descripción, información sobre licencia, dependencias, etc.
- **Tipos de mensajes:** definen la estructura de los datos enviados en los mensajes de ROS.
- **Tipos de servicios:** definen la estructura de los datos tanto de solicitud como de respuesta utilizados en los servicios de ROS.

4.2.3.2 Grafo de computación

El grafo de computación de ROS es una red peer-to-peer, es decir, una red descentralizada en la que se permite la comunicación sin la necesidad de un intermediario. Los conceptos básicos del grafo de computación son los siguientes:

- **Nodos:** Un nodo es un proceso que realiza una tarea. En un sistema robótico, cada nodo se encargaría de una función, por ejemplo, un nodo controlaría el láser, otro nodo los motores de las ruedas, etc.

- **Maestro:** el maestro de ROS no actúa como intermediario, simplemente proporciona el registro de nombres para que los nodos puedan encontrarse y comunicarse entre ellos.
- **Servidor de parámetros:** se utiliza para almacenar datos, forma parte del maestro.
- **Mensajes:** un mensaje hace referencia a la información que dos nodos comparten al comunicarse. Los mensajes contienen tipos de datos estándar tales como enteros, booleanos...
- **Tópicos:** un tópico es un canal en el que se publica la información que contiene cada mensaje.
- **Servicios:** un servicio es otra forma de comunicación de ROS que consiste en un cliente-servidor. Un nodo envía una petición a otro nodo y se queda bloqueado hasta que le responde.
- **Bags:** la herramienta bag de ROS se utiliza para almacenar datos en forma de mensajes con el objetivo de reproducirlos en tiempo real posteriormente.

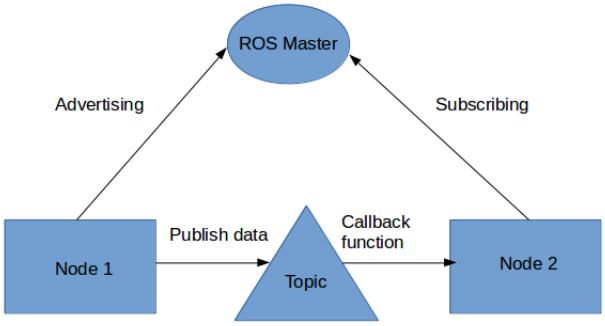

Figura 4.1: Esquema del funcionamiento básico de ROS.

A continuación se explica cómo se relacionan estos conceptos para entender la funcionalidad que tiene ROS.

El maestro es el que permite que haya comunicación entre los diferentes nodos, sin el maestro los nodos no serían capaces de comunicarse entre sí ya que los nodos se comunican con el maestro para indicar sus registros a los tópicos.

Los nodos envían datos en forma de mensajes mediante los tópicos. Un nodo puede publicar o suscribirse a un tópico, el nodo publicador es el que envía los datos a un tópico mientras que el nodo suscriptor es el que los recoge. Un nodo no puede publicar y suscribirse a un mismo tópico, pero sí que puede publicar en un tópico y suscribirse a otro distinto.

Cuando un nodo se suscribe a un tópico, su proceso se mantiene bloqueado hasta que otro nodo publica en ese tópico, entonces, se ejecuta la función callback que está asociada al nodo suscriptor. Por ejemplo, un nodo que está suscrito a la cámara del robot está esperando a

que se publique la imagen que está viendo la cámara. Una vez se ha publicado la imagen, la función callback recoge la imagen y ejecuta el código que haya dentro de esta función.

Este tipo de comunicación mediante tópicos es asíncrona, para una comunicación síncrona se utilizan los servicios. Este otro tipo de comunicación se suele utilizar cuando la tarea que está realizando el robot requiere de algún tipo de respuesta antes de seguir con la ejecución del programa.

Por ejemplo, a modo de comparación, se utilizaría la comunicación mediante tópicos en un nodo que se suscribe al tópico de la cámara del robot, hace ciertos cálculos con umbrales para detectar cierto color y publica en otro nodo el color que detecta. En este ejemplo, una vez el nodo ha publicado el color detectado, no necesita esperar ningún tipo de respuesta para realizar el proceso otra vez. En cambio, si el nodo en vez de calcular cierto color presente en la imagen, tiene como tarea detectar una cara y confirmar que es cierta persona, en este caso es mejor utilizar la comunicación mediante servicios ya que la ejecución se quedará bloqueada hasta que el nodo devuelva la confirmación de la detección.

4.3 Dataset USYD CAMPUS DATASET

El entrenamiento de una red neuronal requiere los datos suficientes para un correcto aprendizaje y generalización. Los datos provienen del dataset **USYD CAMPUS DATASET** [17], este dataset está formado por 52 secuencias que han sido grabadas cada semana durante un año, por lo que se ha grabado en diferentes condiciones ambientales. Los datos han sido recogidos por un coche autónomo por los alrededores de la Universidad de Sydney. Este dataset incluye datos de diferentes sensores, para cada secuencia se obtienen 6 imágenes en color, datos de la nube de puntos del LIDAR Velodyne (16 haces), también dispone de datos IMU y GPS. Además, dispone de un dataset para segmentación.

Figura 4.2: Ejemplo 1 de imagen del dataset. Fuente:[17]

Figura 4.3: Ejemplo 2 de imagen del dataset. Fuente:[17]

4.4 YOLOv3

El detector utilizado para obtener los bounding box 2D a partir de la imagen RGB es YOLOv3 [23]. El modelo utilizado en este proyecto es la tercera versión de YOLO, a continuación se muestra un breve resumen de las dos primeras versiones para poner en contexto la descripción de esta tercera versión.

4.4.1 YOLO-You Only Look Once

YOLO es un algoritmo que recibe como entrada una imagen y devuelve como salida los bounding boxes de los objetos que ha clasificado. YOLOv1 [22] divide la imagen de entrada en una malla S x S, si el punto medio de un objeto cae dentro de una celda, esa celda se encargará de detectar ese objeto. Cada celda predice un número B de bounding boxes y lo que se conoce como confianza, que representa la probabilidad de que ese objeto sea el que se ha predicho.

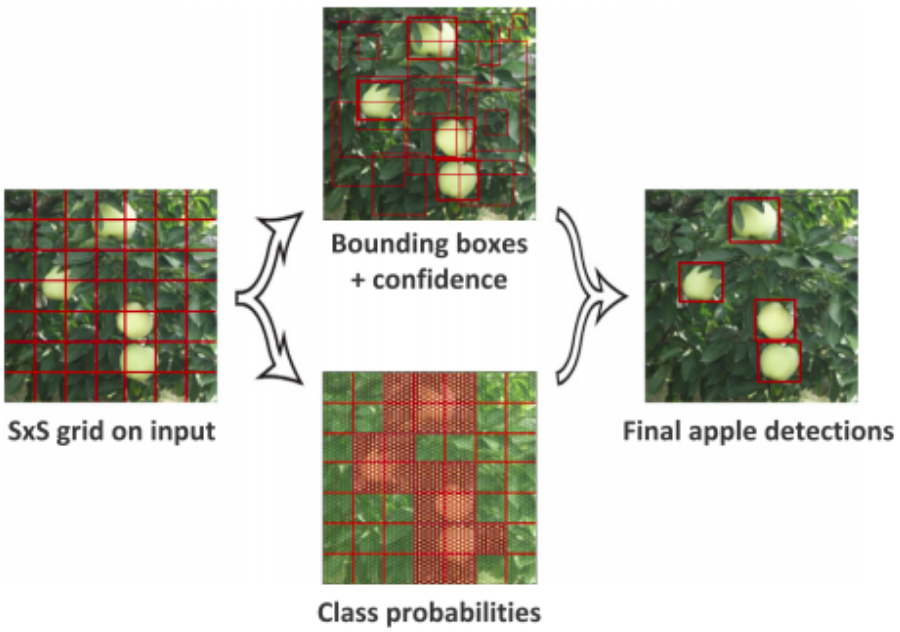

Figura 4.4: Descripción funcionamiento de YOLOv1. Fuente: [26]

En cuanto a la arquitectura, se recoge en la Figura 4.5.

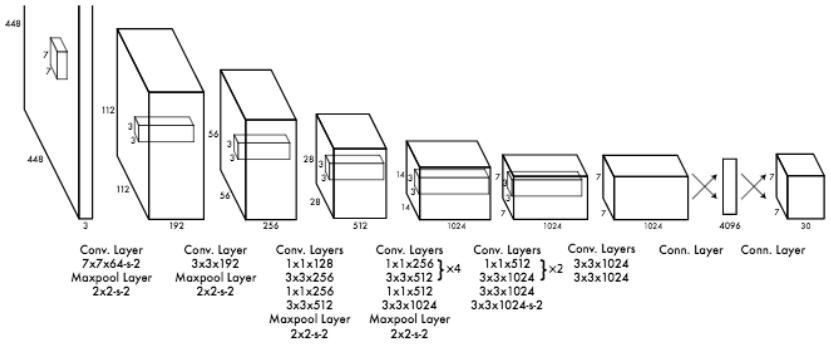

Figura 4.5: Aquitectura YOLOv1. Fuente: [22]

Esta primera versión supuso un avance en el estado del arte ya que es más rápida, tiene menos sensibilidad al fondo de las imágenes, lo que disminuye el número de falsos positivos, y tiene una alta capacidad de generalización. Al ejecutar detección y localización en un mismo pipeline, YOLOv1 predice con rapidez pero afecta a la precisión de localización. Otro problema de YOLOv1 es que, si hay más de un objeto dentro de una misma celda, no va a ser capaz de detectarlos todos, este es el problema que tiene esta versión con objetos muy cercanos. Por último, esta versión fue entrenada para detectar solo 20 clases.

4.4.2 YOLO9000-Better, Faster, Stronger

YOLOv2 [21] surge con el objetivo de mejorar el Recall [1] y la precisión de localización manteniendo la precisión de clasificación.

Se presentaron una serie de mejoras respecto a la primera versión que se pueden resumir en:

- **Batch Normalization**: se añade en todas las capas convolucionales, supone una mejora de un 2% en mean Average Precision (mAP).
- **High Resolution Classifier**: se aumenta la resolución de las imágenes de 224x224 a 448x448, mejora en un 4% en mAP.
- **Convolutions with Anchor Boxes**: en YOLOv2 se eliminan las capas totalmente conexas y se utiliza la técnica **anchor boxes** para predecir los bounding boxes. Esto hace que el mAP disminuya un 0.03% pero aumenta el recall un 7%.
- **Dimension clusters**: se eligen los k mejores anchor boxes. Estos k mejores se obtienen mediante el algoritmo K-Means.
- **Direct Location Prediction**: se calcula el centro del anchor box final con una función sigmoide y se utilizan las dimensiones de los anchor boxes obtenidos a partir de la clusterización para calcular las dimensiones de los anchor boxes finales.
- **Fine-Grained Features**: se modifica la dimensión del mapa de características para mejorar la localización de objetos pequeños.
- **Multi-Scale Training**: cada 10 batches, se modifican las dimensiones de la imagen de forma aleatoria y se hace un reshape de la red para seguir entrenando.

	YOLO							YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓	✓
anchor boxes?				✓	✓				
new network?					✓	✓	✓	✓	✓
dimension priors?						✓	✓	✓	✓
location prediction?						✓	✓	✓	✓
passthrough?							✓	✓	✓
multi-scale?								✓	✓
hi-res detector?									✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

Figura 4.6: Mejoras YOLOv2 versión final. Fuente: [21]

En la Figura 4.6 se resumen los avances añadidos en esta segunda versión de la red. Se puede resumir el funcionamiento en que se divide una imagen en una malla, en cada celda se van a

[1] División entre True Positive y True Positive + False Negative

colocar los anchor boxes con la posición calculada como se ha comentado en los puntos 4,5 y, por cada anchor box, se ejecuta la predicción de clasificación y localización. Por lo tanto, se obtendrá un bounding box por cada anchor box en el caso de haber detectado objeto en ese anchor box. Para elegir qué bounding box se ajusta mejor, se calcula el Intersection over Union (IoU) del bounding box con su anchor correspondiente y se coge el que mayor tenga. Estos bounding boxes son los que la red aprende.

Para finalizar con YOLOv2, es una versión más rápida que la anterior ya que cambia la arquitectura GoogLeNet por Darknet-19. Esta arquitectura utiliza convoluciones 1x1 para reducir el número de parámetros a aprender y obtiene un balance positivo entre precisión y complejidad. YOLOv2 es más robusta porque se entrena con 9418 clases, además de que soluciona el problema de devolver un único bounding box cuando hay más de un objeto en una misma celda.

4.4.3 YOLOv3: An incremental improvement

Una vez se ha entrado en contexto, la descripción de YOLOv3 será más sencilla de entender.

Según [23], YOLOv3 se basa en:

- **Bounding Box Prediction**: se mantiene el mismo procedimiento para el cálculo de los anchor boxes finales que utiliza YOLOv2. YOLOv3 añade una condición que consiste en que si hay un bounding box que no tiene el mayor IoU pero es mayor que 0.5, se obvia esta predicción. Si un bounding box no está asignado a ningún anchor box, solo se va a tener en cuenta si detecta objeto o no[2] objeto para calcular el error.

- **Class Prediction**: Se utilizan clasificadores logísticos independientes para detectar las múltiples clases que puede contener un bounding box en vez de utilizar softmax. La razón es porque si se aplica esta versión en otros datasets como Open Images Dataset [15], hay clases que se superponen como por ejemplo las clases mujer y persona, y con softmax solo podría devolver una clase por bounding box. Entonces, requiere que se pueda detectar más de una clase en cada bounding box, pero siempre teniendo en cuenta que es un único objeto. El error utilizado en el entrenamiento es binary cross-entropy.

- **Prediction Across Scales**: Como se comentará en el siguiente apartado, se utiliza la arquitectura Darknet-53. Se añaden capas convolucionales en diferentes zonas de la arquitectura, la última capa de estas devuelve un vector con el bounding box, el objectness score y las predicciones de las clases. El proceso es el siguiente: se hace la primera predicción, se recogen características y se le aplica upsample x2, básicamente es duplicar el tamaño de las características, y se concatenan con características de capas anteriores. Se realiza la segunda predicción, se repite el proceso de las características y se predice por última vez en la escala final. Este proceso permite obtener información semántica de mayor calidad.

[2] Objectness score.

- **Feature Extractor**: la arquitectura utilizada añade conexiones hacia delante [3] a Darknet-19. Además, aumenta el número de capas hasta 53, por eso el nombre de Darknet-53. Es más potente que Darknet-19 y más rápida que otras arquitecturas estado del arte como ResNet-101 [13] o ResNet-152 [13] con las que comparan en el paper.

Backbone	Top-1	Top-5	Bn Ops	BFLOP/s	FPS
Darknet-19 [15]	74.1	91.8	7.29	1246	**171**
ResNet-101[5]	77.1	93.7	19.7	1039	53
ResNet-152 [5]	**77.6**	**93.8**	29.4	1090	37
Darknet-53	77.2	**93.8**	18.7	**1457**	78

Figura 4.7: Comparación de YOLOv3 con otras arquitecuras en el challenge Imagenet. Fuente: [23]

En la Figura 4.7 aparece la comparación de YOLOv3 en cuanto a precisión, Floating Point Operations Per Second (FLOPS) y Frames Per Second (FPS). Para concluir con YOLO, se incorpora esta versión en el proyecto por su alta precisión y bajo tiempo de predicción.

4.5 PointRgbNet

PointRgbNet [8] es una red neuronal híbrida cuyo propósito es detectar objetos urbanos evitando los errores de los detectores 2D que se han comentado en la introducción del proyecto.

En primer lugar, una red híbrida es aquella que presenta una 3DCNN y una 2DCNN. Esta red neuronal tiene como entradas una imagen RGB y una nube de puntos voxelizada que serán las entradas de las 2DCNN y 3DCNN, respectivamente. Ambas redes se ejecutan en paralelo y se juntan sus salidas en una capa totalmente conexa para la predicción.

En cuanto a las arquitecturas, se define primero la del detector 2D y luego la del detector 3D.

4.5.1 Arquitectura subred 2D

- Capa de entrada 200x150 píxeles.
- Capa convolucional 2D con 96 filtros 11x11 y función de activación ReLu.
- MaxPooling 2D 2x2 con stride de 2.
- Capa convolucional 2D con 256 filtros 11x11 y función de activación ReLu.
- MaxPooling 2D 2x2 con stride de 2.
- Capa convolucional 2D con 384 filtros 3x3 y función de activación ReLu.
- Capa de dropout con un valor de 0.5.

[3] Shorcut connections

4.5.2 Arquitectura subred 3D

- Capa de entrada 20x20x20 voxels.
- Capa convolucional 3D con 100 filtros 33x33 y función de activación ReLu.
- Capa de dropout con un valor de 0.5.
- Capa convolucional 3D con 100 filtros 2x2 y función de activación ReLu.
- Capa convolucional 3D con 64 filtros 2x2 y función de activación ReLu.
- Capa de dropout con un valor de 0.5.

Se aplastan (flatten) ambas salidas para poder pasarlas como entrada a una última capa densa con 3 salidas y función de activación softmax. Las 3 salidas representan las clases peatón, vehículo o fondo.

4.6 PointNet

PointNet [19] es una red neuronal 3D que, a diferencia de otras propuestas, trabaja directamente con la nube de puntos. Esta red se utiliza para diferentes tareas: clasificación, segmentación de objetos independientes y segmentación de la escena completa.

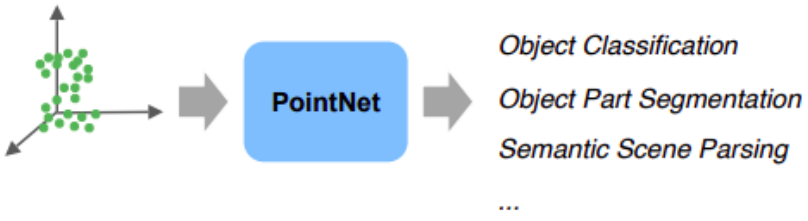

Figura 4.8: PointNet Fuente: [19]

En cuanto a la arquitectura, se va a explicar solo la correspondiente a la clasificación.

La entrada a la red es una capa de tamaño nx3 donde n representa el número de nubes de puntos. A estas nubes de puntos se les aplica una transformación, esta consiste en multiplicar la matriz de entrada por una matriz de transformación afín predicha por una red **T-net**. Además, a esta matriz predicha se le aplica una transformación para que sea ortogonal.

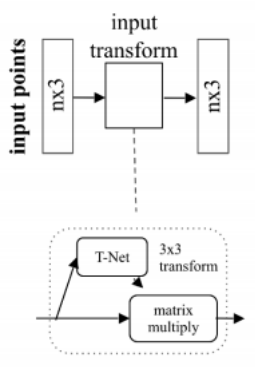

Figura 4.9: Arquitectura PointNet parte 1 Fuente: [20]

Este primer proceso se aplica para que la red sea invariante a cualquier transformación, por ejemplo que no influya en la predicción el hecho de que la nube esté girada.

En el siguiente bloque se pasa como entrada las nubes de puntos transformadas a un bloque de multi layer perceptrons (2 capas de 64 neuronas cada una) obteniendo así 64 características por cada nube de puntos.

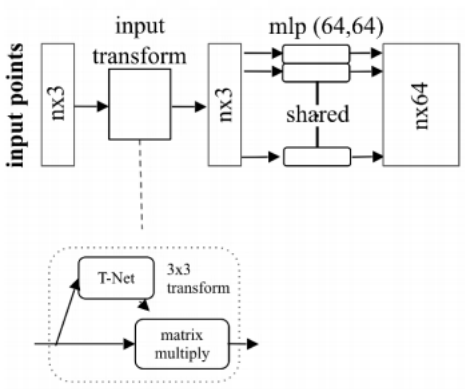

Figura 4.10: Arquitectura PointNet parte 2 Fuente: [20]

A la salida nx64 obtenida se le aplica una transformación como la de la primera parte, lo único que cambia es el tamaño de la matriz predicha, que era 3x3 y aquí es 64x64. Recordamos que esta operación hace que las nubes sean invariantes a cualquier transformación rígida.

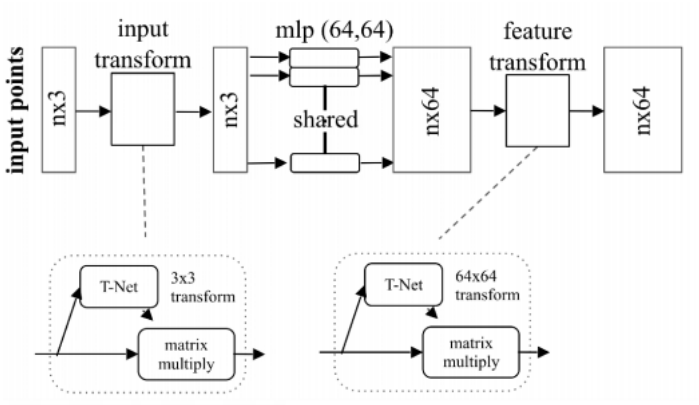

Figura 4.11: Arquitectura PointNet parte 3 Fuente: [20]

Se vuelve a repetir el proceso, las nubes transformadas se pasan a un bloque de multi layer perceptrons pero esta vez tiene 3 capas, 64,128 y 1024 neuronas, teniendo en la salida 1024 características por cada nube.

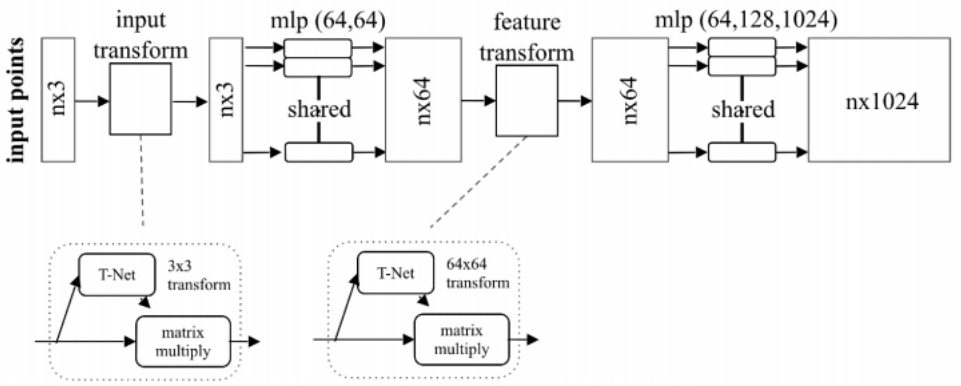

Figura 4.12: Arquitectura PointNet parte 4 Fuente: [20]

Por último, se aplica una capa de max pooling la cual es una función simétrica que permite que los datos sean invariantes a permutaciones. La salida de este max pooling se pasa por otro bloque mlp con tres capas (512,256,k) donde k son los resultados que se obtienen para la predicción de los k objetos detectados.

La arquitectura de PointNet completa se puede observar en la siguiente Figura 4.13:

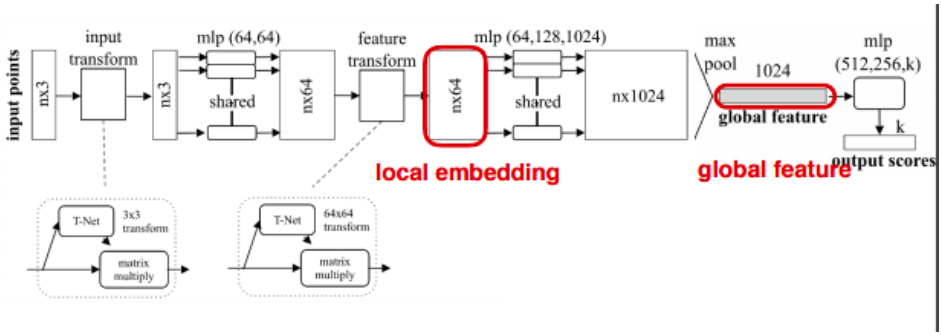

Figura 4.13: Arquitectura PointNet Fuente: [19]

5 Desarrollo

En este apartado se va a describir el funcionamiento del pipeline de detección explicando cada una de las partes que lo componen, también se describirá cómo se ha creado el dataset a partir de los datos del dataset explicado en la Sección 4.3.

5.1 Descripción del pipeline

En primer lugar, este pipeline está creado en ROS por lo que se hace uso de las diferentes herramientas que nos proporciona como son los nodos publicadores y suscriptores, los tópicos, los servicios, etc. Todo el pipeline se encuentra dentro de un catkin workspace.

Como se ha comentado en apartados anteriores, hay dos detectores, uno 2D y otro 3D. El detector 2D es YOLOv3 y el detector 3D será PointNet. El funcionamiento es el siguiente:

1. Roscore.

2. Se cargan los pesos de YOLOv3 y se espera a que le envíen las imágenes.

3. Reproduce los datos del dataset.

4. Pasa una imagen a YOLOv3, detecta los diferentes objetos y devuelve las clases predichas junto con los bounding boxes correspondientes.

5. Aplicando una serie de transformaciones se obtienen los puntos 3D correspondientes al bounding box de cada predicción.

6. Preprocesamiento de las nubes de puntos: aquellas nubes que tengan menos de 50 puntos se obviarán, las que tengan más de 50, se aumenta o reduce el número de puntos hasta 1024.

7. Envía las nubes de puntos 3D a PointNet, predice y devuelve la clase predicha.

8. Guarda los resultados de las predicciones.

9. Vuelve al paso 4 y pasa una nueva imagen.

Una vez explicado el funcionamiento general, se entra en detalle en cada una de las partes que lo componen.

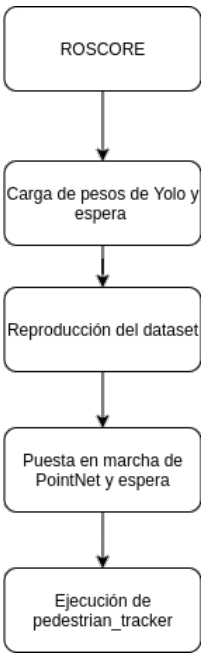

Figura 5.1: Diagrama funcionamiento general

5.1.1 Instanciar YOLOv3 preentrenada

Este primer paso consiste en cargar los pesos de YOLOv3 y crear un servicio de ROS, **yolo_server/detect_persons**. Este servicio recibe una imagen como entrada y devuelve un string con las coordenadas de cada bounding box, las etiquetas y el score de la predicción.

Esta parte se encuentra en el script **yolo_server.py** dentro del paquete **camera_projection_mod** en el que, mediante un rospy.spin(), se mantiene bloqueada la ejecución hasta que se reciba una llamada al servicio.

Después de lanzar roscore en una terminal, ejecuta **rosrun camera_projection_mod yolo_server.py** en otra terminal para ejecutar esta parte del pipeline.

5.1.2 Cargar los datos

Los datos se encuentran almacenados en un formato .bag, por lo que se utiliza la herramienta rosbag de ROS para poder cargarlos. ROS proporciona una API para desarrollar el código necesario para leer los datos, este código se encuentra dentro del paquete **h264_bag_tools**.

En realidad, la herramienta rosbag sirve para poder reproducir los datos que se habían guardado, mandando los mensajes a los tópicos en tiempo real. Esta información se recoge en el script principal, **pedestrian_tracker.cpp**, el cual enviará los datos al resto de scripts.

Para ejecutar este paso, lanza en una terminal el siguiente comando: **roslaunch h264_bag_tools h264_playback.launch bag_file_name:="/path/to/any/week/of/the/usyd**

/dataset.bag".

5.1.3 Pasar imágenes a YOLOv3 y predicción

Después de cargar los pesos de YOLOv3 y ejecutar la reproducción del dataset, hay que ejecutar en dos terminales distintas los siguientes comandos: **rosrun camera_projection _mod pointnet_multiclass_predict.py** y **rosrun camera_projection_mod pedestrian_tracker 0**. El primer comando lanza el script en el que se encuentra PointNet, mientras que el segundo lanza el script principal en el modo detección normal. Con estos últimos dos comandos ya estaría en marcha todo el pipeline.

En pedestrian_tracker se crean 3 nodos suscriptores para recoger los datos que están reproduciéndose, concretamente, recogen la nube de puntos, los parámetros extrínsecos e intrínsecos de la cámara y la imagen[1]. Los nodos están sincronizados de manera que, hasta que los tres nodos no reciban su correspondiente mensaje, no se ejecutará el callback ya que los 3 llevan asociados la misma función como callback.

Al recibir cada nodo su mensaje, empieza a ejecutarse la función de callback. Este apartado se centra en la obtención de la imagen, su envío a YOLOv3 y la predicción.

La primera parte de esta función recoge la imagen y la convierte a un formato **bgr8** mediante la herramienta **cv_bridge** de **OPENCV**. Una vez tiene la imagen, lanza un hilo que ejecuta la función **callToYoloServer**. Esta función envía la imagen al servicio creado en el apartado 5.1.1.

Al enviar la imagen, la ejecución del pipeline vuelve al script **yolo_server.py** que estaba en espera. Entonces, se ejecuta la función **run_yolo**. Esta función obtiene la imagen enviada, comprueba si existe, y si es así, convierte la imagen de bgr a rgb. Después, ejecuta la detección de YOLOv3 y separa las etiquetas con sus correspondientes boxes y scores en las categorías person, car, motorbike y bicycle. Por último, envía las etiquetas, scores y boxes como respuesta del servicio al script pedestrian_tracker.

5.1.4 Obtención de los puntos 3D

La obtención de las nubes de puntos correspondientes a los bounding boxes de cada objeto es sencilla porque la cámara y el láser están calibrados. Por lo tanto, para conocer los puntos 3D respectivos al box, simplemente se aplica una transformación que pasa del sistema de referencia del láser al de la cámara proyectando los puntos. Una vez hecha esta proyección, es como si el láser y la cámara estuvieran en el mismo punto del espacio.

El siguiente paso es calcular el punto mediano de todos los puntos tridimensionales y crear un bounding box 3D alrededor de este punto. El objetivo del cálculo de la mediana es eliminar puntos que se encuentren dentro del bounding box de YOLOv3 pero que sean del fondo o de algún objeto que se encuentre delante del objeto de interés.

[1]Recordatorio: sólo se obtiene la imagen de la cámara central.

5.1.5 Preprocesado de las nubes de puntos

La predicción de PointNet es menos precisa cuando el número de puntos de las nubes es pequeño. Por lo tanto, revisando el número de puntos que tienen las nubes de los boxes 3D, se llega a la conclusión de que un buen umbral puede ser 50 puntos. De esta manera, todas las nubes que tengan menos de 50 puntos son obviadas mientras que el resto de nubes son preprocesadas.

La red PointNet necesita que el número de puntos de entrada sea como mínimo 1024, por lo que hay que aumentar/reducir el número de puntos de las nubes. El proceso para aumentar el número de puntos consiste en repetir los puntos que ya tiene la nube hasta que se llegue a 1024.

Para aquellas nubes que tienen más de 1024 puntos, primero hay que eliminar los puntos que sobran hasta 1024, pero no es buena idea eliminar estos puntos del principio o del final. Es una mejor idea eliminar un punto cada x puntos dejando que x dependa del número de puntos sobrantes. En resumen, el cálculo sería el siguiente:

- Se define m como la resta entre 1024 y el número de puntos de la nube.
- Se define x como la división entera entre el número de puntos y m. X se calcula haciendo la división entera para asegurar que el número de puntos después de borrar quede siempre por debajo de 1024 y en ningún caso quede por encima.
- Se borran los puntos sobrantes de modo que se borra un punto cada x puntos de la nube.
- Después de borrar, la nube tendrá algunos puntos menos de 1024 por la división entera, entonces se vuelve a rellenar hasta 1024.

5.1.6 Envío de datos a PointNet y predicción

Por sencillez, las nubes de puntos se envían como un vector de floats en el que cada punto tridimensional corresponde con tres elementos de ese vector, es decir, cada nube representará 1024x3 elementos del vector.

La forma de enviar los datos es igual que con las imágenes, se ha creado un servicio de ROS que tiene como entrada el vector de floats y como salida un vector de enteros.

Cuando los datos están preparados para ser enviados, en el script principal se genera un nuevo hilo que llama a la función **callToPointNetBulk** que llama al servicio creado y le pasa las nubes de puntos.

El código de la predicción de PointNet está dentro del paquete **camera_projection_mod** y se encuentra en el script de python **pointnet_multiclass_predict.py**. Este script, primero carga el modelo de la red entrenada, hace un reshape del vector de las nubes de puntos porque el formato de la capa de entrada de PointNet es (nº nubes, nº puntos, 3), predice y devuelve el vector de enteros con las predicciones[2].

[2] 0-vehículo,1-no objecto,2-persona.

5.1.7 Guarda los resultados

Una vez pedestrian_tracker recoge las predicciones, las compara con las que ha hecho YOLOv3 y guarda en una carpeta todas las predicciones junto con las imágenes y, en otra carpeta, las predicciones e imágenes en las que PointNet no coindice con YOLOv3. Se realiza de esta manera con el objetivo de ver más rápido si PointNet es capaz de detectar los ejemplos en los que YOLOv3 falla y son de interés.[3] Con la carpeta de las predicciones generales se calcula la precisión con la que detecta PointNet, mientras que con la carpeta de las predicciones contradictorias se calcula la precisión que tiene PointNet para cumplir el objetivo del proyecto.

Por último, en cuanto al pipeline, en el caso de ejecutar el pipeline en tiempo real, lo único que cambia es el paso en el que se reproducen los datos del dataset, este se cambia por la obtención en tiempo real de las imágenes y la información del láser.

El siguiente diagrama de la Figura 5.2 entra más en detalle en el funcionamiento del script pedestrian_tracker y su interacción con el resto de elementos del pipeline.

[3]Recordatorio: los ejemplos de interés son aquellos en los que hay un objeto real en un anuncio, cartel publicitario...

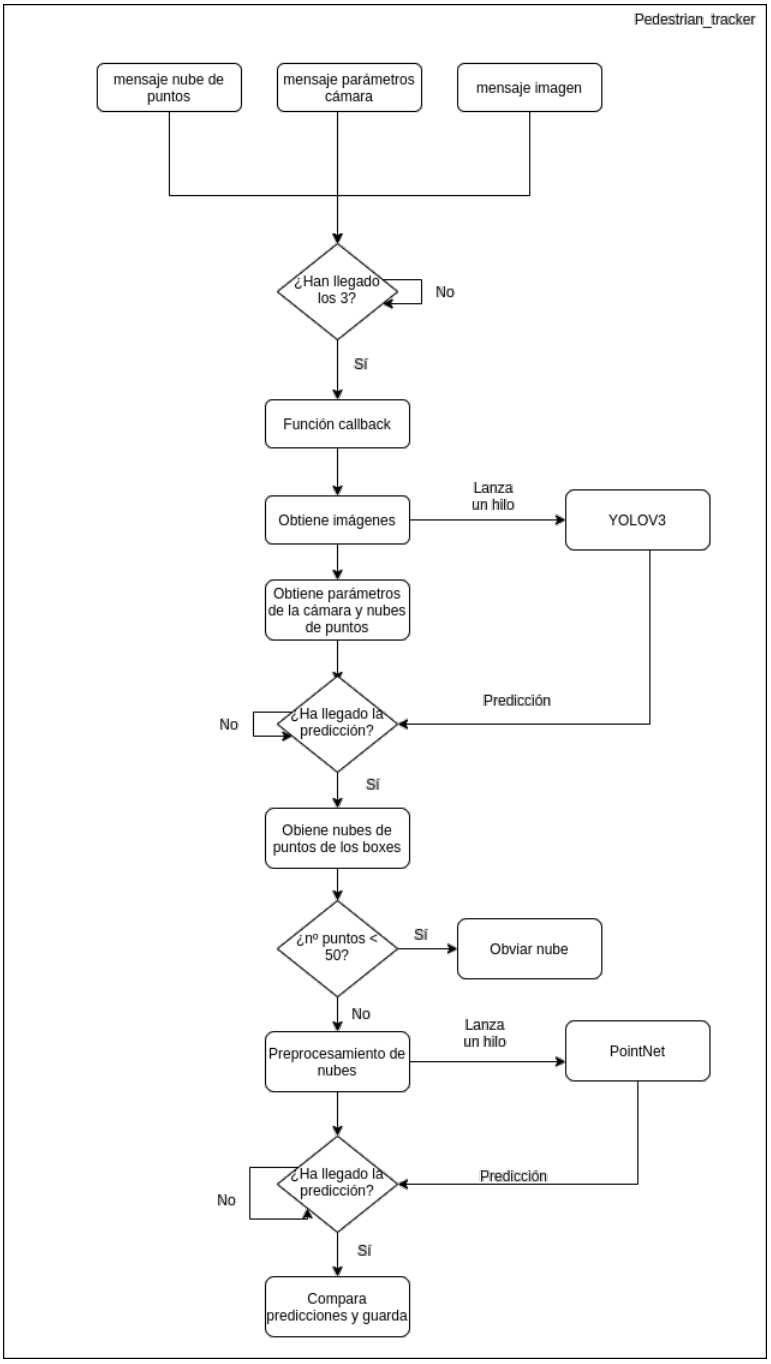

Figura 5.2: Diagrama pedestrian_tracker

5.2 Creación del dataset

Es importante recordar que no se utiliza PointNet con sus pesos por defecto, sino que se realiza el entrenamiento de la red con los datos específicos para el problema que pretende resolver este proyecto. Por lo tanto, se debe crear la base de datos para entrenar la red.

El primer paso para crear el dataset es utilizar el pipeline del mismo modo que se ha comentado anteriormente, pero modificando el argumento que se le pasa a pedestrian_tracker. Si el objetivo es la detección en tiempo real, el argumento es un 0, pero, en este caso, el objetivo es almacenar las imágenes junto con sus respectivas nubes de puntos, entonces, el argumento es el nombre que identifica esas predicciones.

Es importante hacer hincapié en que en el dataset visto en la Sección 4.3 se reproducen primero las semanas impares y luego las semanas pares para crear el dataset propio. Esto es importante para que las nubes de puntos no sean consecutivas en cuanto a las semanas del dataset.

Después, se desarrolla un script de python, **create_dataset.py**, que lee las nubes de puntos de las carpetas en las que están guardadas y almacena las nubes en un vector para cada categoría. Estos vectores se barajan aleatoriamente para que las nubes de puntos estén almacenadas sin ningún tipo de relación entre ellas. Tanto reproducir las semanas de esta forma como hacer aleatorios los vectores de nubes de puntos es importante para el entrenamiento de PointNet.

Una vez las nubes de puntos están distribuidas aleatoriamente, el siguiente paso es separar los datos en dos sets, entrenamiento y validación. A modo de recordatorio, el set de entrenamiento es el que se utiliza para que la red aprenda durante el entrenamiento, mientras que el set de validación se utiliza para que, después de cada época[4], la red prediga sobre este set. Por lo tanto, el set de validación es útil para comprobar si el entrenamiento está yendo bien o mal.

Volviendo al dataset, una buena práctica es dividir los datos en 70% para entrenamiento y un 30% para validación. Este script lleva a cabo esta división de los datos, pero también tiene en cuenta que el set de validación es solo un 30% de los datos, por lo que se llenará antes que el set de entrenamiento y cogerá los datos que estén en la primera mitad del vector. Aunque ya se han realizado dos pasos para evitar que las nubes de puntos sean de la misma semana o de semanas consecutivas, se vuelve a añadir aleatoriedad al establecer que un dato entrará en el set de validación con una probabilidad mayor del 60-70%. De esta manera, se garantiza que el set de validación estará formado por datos distribuidos por todo el vector.

Un detalle que no se ha comentado en la descripción del pipeline pero que es importante a la hora de crear el dataset, es que YOLOv3 no va a detectar objetos de la categoría no objeto. Por lo tanto, esa categoría se crea cogiendo bounding boxes de la imagen, con un tamaño aleatorio dentro de unos límites y respetando siempre los bounding boxes del resto de objetos detectados. A partir de estas imágenes, se obtienen las nubes de puntos y se guardan

[4]Una época es cuando la red predice todas las muestras del set de entrenamiento.

con el resto de predicciones.

Antes de crear el dataset, se investiga el formato que utilizan los creadores de PointNet para sus datasets y se decide adoptar el mismo formato para que sea más sencillo el uso de PointNet, este formato es **h5**. Por último, se crean los dos datasets, el de entrenamiento y el de validación siguiendo la estructura que exige la capa de entrada de PointNet, (nº nubes, nº puntos, 3).

Como se explicará en el apartado 6, hace falta crear una base datos de nubes de puntos de los ejemplos que estamos buscando. El siguiente apartado explica cómo se crea esta base de datos.

5.3 Base de datos de ejemplos de interés

Con el propósito de tener un dataset en el que probar si la red cumple o no los objetivos del proyecto, se utiliza el simulador Gazebo para crear este dataset, ya que los ejemplos de interés aparecen con menor frecuencia y se necesitan los suficientes datos sintéticos por un lado para los conjuntos de train y test, y para validar si el objetivo se cumple correctamente o no.

En primer lugar, para obtener nubes de puntos de las mismas características que las del dataset, se necesita el mismo modelo del láser que lleva el coche autónomo con el que se grabó el dataset. Este láser es el **Velodyne VLP-16** y tiene una implementación de su modelo en Gazebo. Por lo tanto, se añade el modelo del láser junto con una cámara colocada encima. Es importante conocer la transformación entre la cámara y el láser para cambiar de un sistema de referencia a otro a la hora de obtener las nubes de puntos de los boxes.

En la siguientes figuras, Figura 5.3 y Figura 5.4, se puede observar el láser y la cámara en Gazebo y en RVIZ. RVIZ, un simulador 2D, muestra también la nube de puntos detectada por el láser.

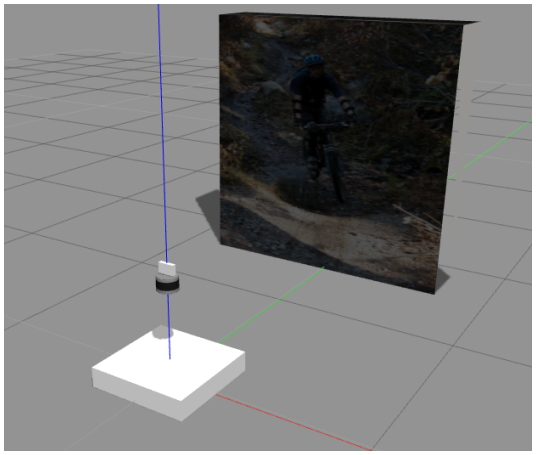

Figura 5.3: Láser y cámara en Gazebo.

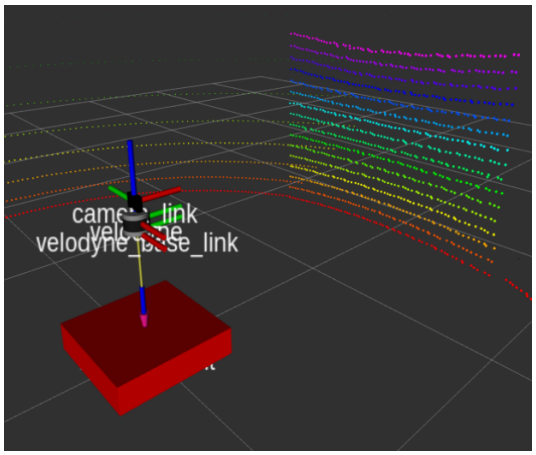

Figura 5.4: Láser y cámara en RVIZ.

La idea es crear modelos 3D con fotos de personas, coches, bicis y motos colocadas en estos modelos para simular la superficie 3D de objetos detectados como personas y vehículos en vallas publicitarias, motos en marquesinas de autobús... Después, leer la imagen del tópico de la cámara y guardarla junto con la nube de puntos.

Este sería el proceso para un modelo, cuando se acaba con uno, se envía un comando desde terminal que permite mover el bloque en el que se encuentra el láser y la cámara hasta el siguiente modelo. Como este comando consiste en publicar en un tópico de ROS que representa el modelo de Gazebo, se puede incorporar en el código de manera que, cuando el bloque haya terminado de recoger los datos y los haya guardado, pase a la siguiente posición y vuelva a detectar. Así, la detección está totalmente automatizada.

Al inicio de crear el dataset, se empezó creando los modelos 3D con Skectchup 3D en Windows y luego se enviaban a Ubuntu donde, a mano, se creaban las carpetas necesarias para añadir los modelos en Gazebo. El proceso era muy repetitivo y lento. Entonces, se plantea automatizar el proceso de creación de modelos 3D, de creación/adición de las carpetas en Gazebo y de creación del mundo de Gazebo.

Se descarga el dataset [6] que contiene imágenes de 20 clases diferentes entre las cuales incluye personas, motos, bicis y coches. Por lo tanto, se almacenan las imágenes correspondientes a estas categorías eliminando las imágenes que se repiten. Por ejemplo, una imagen de una persona en una bicicleta aparece tanto en la categoría de persona como en la de bicicleta, por lo que se elimina de una de las categorías para que no se repitan los modelos 3D, teniendo en cuenta que YOLOV3 seguirá detectando ambos ejemplos.

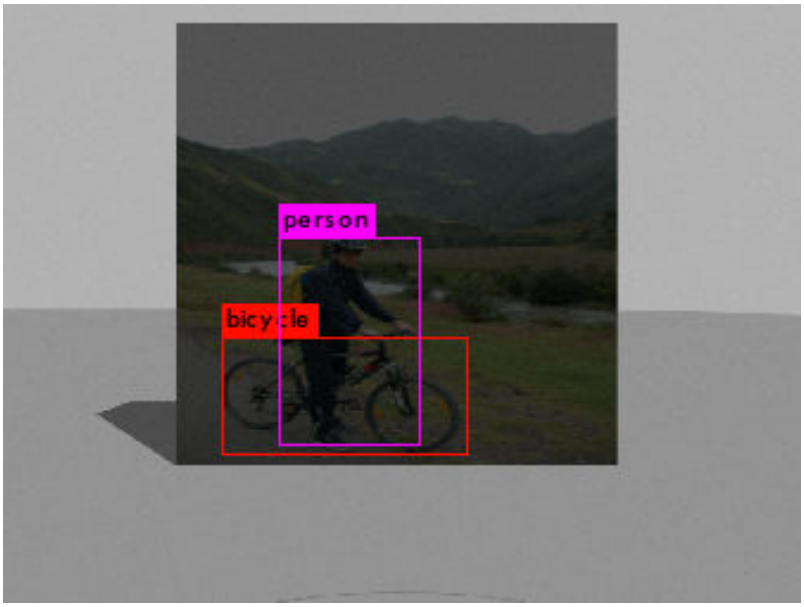

Figura 5.5: Detección en modelo 3D.

Después de las imágenes, el siguiente paso es crear los modelos 3D de forma automática, es decir, ejecutar un script y generar todos los modelos.

Para ello, se utiliza la librería de python **pycollada** que permite crear modelos **COLLADA**. Estos modelos tienen formato **.dae** y son los que se añaden en Gazebo. Entonces, se realiza un script que genera un modelo 3D para cada imagen.

Por lo tanto, quedan dos pasos para tener todo el proceso automatizado, añadir cada imagen junto con su modelo 3D en Gazebo y crear el mundo de Gazebo.

La estructura que debe seguir la carpeta con la imagen y el modelo es la mostrada en la Figura 5.6.

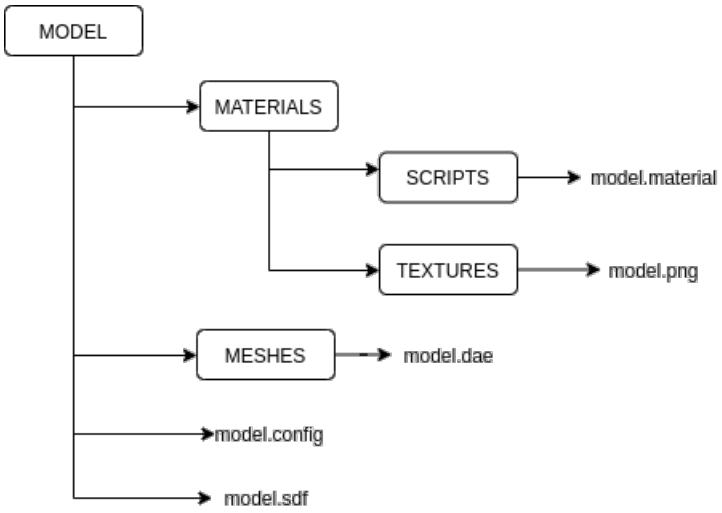

Figura 5.6: Estructura de la carpeta del modelo 3D.

Donde MODEL es la carpeta principal, dentro de esta se encuentra la carpeta MATERIALS, MESHES y los archivos model.config y model.sdf. Dentro de MATERIALS hay otras dos carpetas, SCRIPTS y TEXTURES, en SCRIPTS está el archivo model.material donde se crea el material del modelo 3D a partir de la imagen model.png que se encuentra en TEXTURES. En la carpeta MESHES, se encuentra el modelo 3D creado con pycollada. Por último, quedan los archivos model.config y model.sdf, el primero llama al segundo y, el segundo, define el modelo 3D llamando al resto de archivos: model.material, model.png, model.dae.

El último paso es crear un mundo de Gazebo formado por todos los modelos 3D. Por la situación actual, se trabaja en remoto, en remoto no se puede ejecutar el simulador Gazebo en el servidor de la universidad y el ordenador local no soporta un mundo con tantos modelos. Entonces, hay que crear mundos con menos modelos y repetir el proceso con cada mundo. La Figura 5.7 es un ejemplo de un mundo.

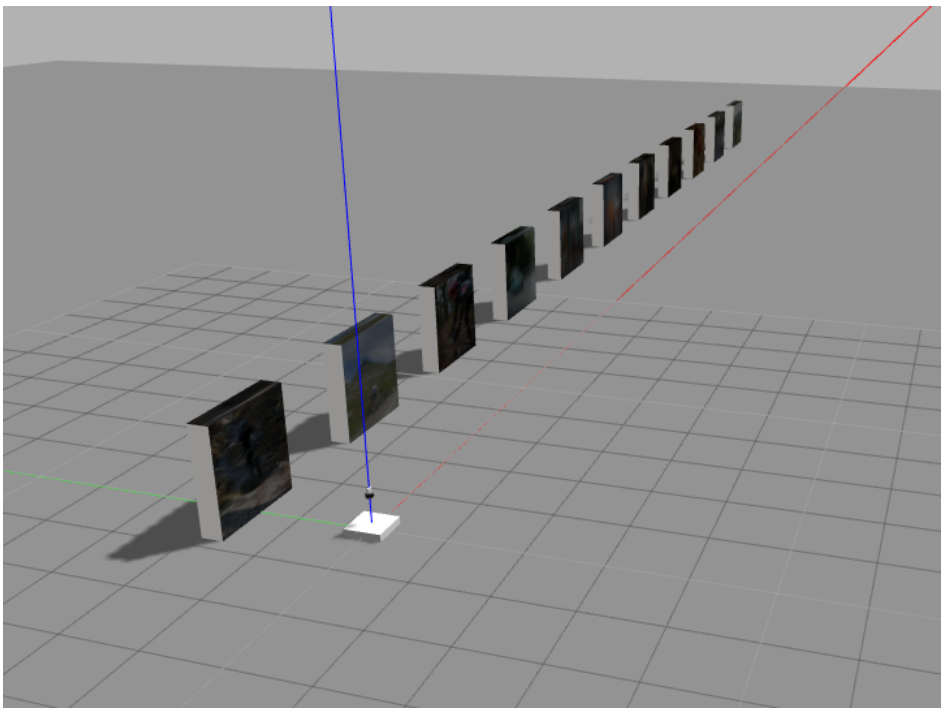

Figura 5.7: Mundo en gazebo.

Se probaron diferentes números de modelos hasta saber el límite que podía cargar mi ordenador local, este límite son 250 modelos por cada mundo. Hay un total de 1387 modelos, por lo que se crean 6 mundos, los 5 primeros con 250 modelos cada uno y, el último, con los 137 restantes.

Además, es una buena práctica obtener las nubes de puntos de los modelos a diferentes distancias. De esta manera, revisando el rango de distancias a las que puede detectar PointNet (2-15m), se crean estos 6 mundos para distancias de 3, 5, 7, 9, 11 y 13 metros. Por lo que hay 36 mundos en total y 8322 imágenes y nubes.

Uno de los problemas que tienen los modelos que se añaden a Gazebo es la iluminación. Teniendo en cuenta que YOLOv3 deberá detectar objetos a una distancia de hasta 13 metros, se decide añadir una fuente de luz encima de cada modelo para facilitar la detección de YOLOv3 y, así, tener más datos ya que detectará más objetos.

5.4 Adaptación del pipeline

Una vez se obtienen las imágenes con sus respectivas nubes de puntos, hay que utilizar el pipeline para detectar los bounding boxes en las imágenes y obtener las nubes de puntos correspondientes a los objetos.

A modo de recordatorio, el pipeline utiliza la herramienta rosbag para reproducir los da-

tos y los envía al script principal. En este caso, se utiliza una modificación del pipeline que recibe las imágenes, las nubes de puntos y las proyecciones de los puntos 3D en las imágenes.

A partir de las nubes de puntos y de los parámetros intrínsecos de la cámara, se obtienen las proyecciones de los puntos en la imagen utilizando la fórmula del modelo de cámara de Pinhole: $x = f * X/Z + cx$ e $y = f * Y/Z + cy$, donde (X,Y,Z) son las coordenadas del punto 3D, (x,y) son las coordenadas del punto proyectado en la imagen, f la distancia focal de la cámara y (cx,cy) las coordenadas del centro óptico de la cámara. En la Figura 5.8 se muestra un ejemplo de esta proyección.

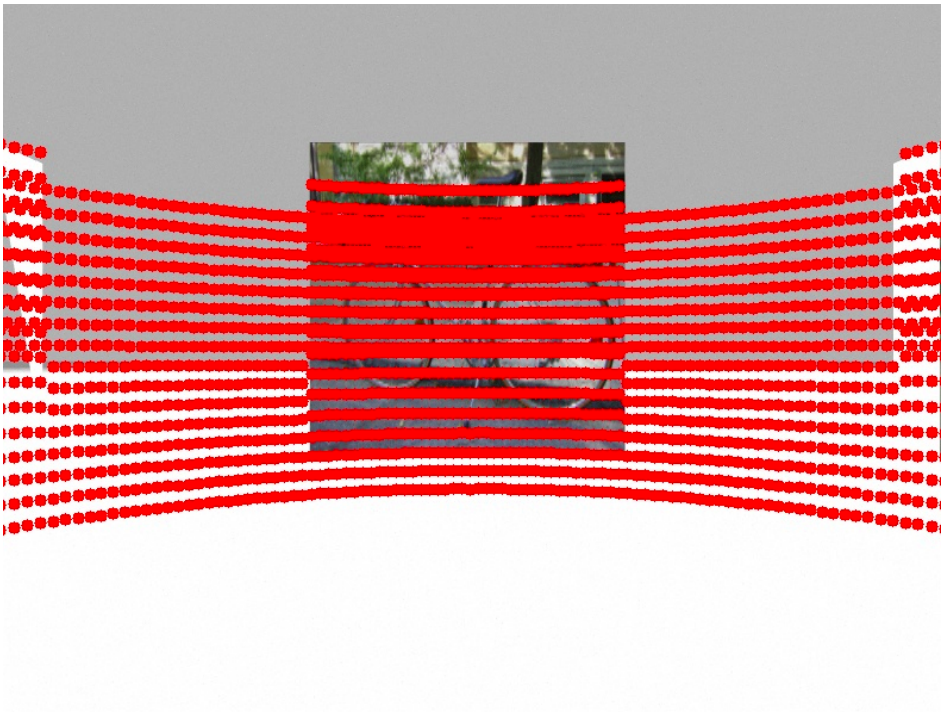

Figura 5.8: Puntos 3D proyectados en imagen 2D.

El funcionamiento del pipeline es similar, primero se detecta con YOLOv3 y se obtienen las etiquetas y boxes, después, se obtienen aquellos puntos 3D cuyas proyecciones estén dentro de los bounding boxes y, a partir de estos puntos 3D, se obtiene el bounding box 3D. Depende del modo de trabajo, se podrá detectar estos bounding boxes 3D con PointNet o guardar el conjunto imagen/nube para crear el dataset.

Los datos se envían al pipeline mediante un servicio de ros, en cada iteración el servicio envía una imagen, su correspondiente nube de puntos y la proyección. Una vez el pipeline ha finalizado con esos datos, el bucle itera y envía el siguiente conjunto imagen/nube/proyección.

Finalmente, el dataset general está formado por 30000 datos sintéticos, de los cuales 4650 son vehículos y 4350 personas mientras el resto corresponden con la clase no objeto, y 7678

datos reales formados por nubes de puntos que representan 1756 vehículos, 4591 no objetos y 1331 personas. A partir de estos datos, se crean datasets con tamaños distintos según cada prueba. Además, para algunas pruebas es necesario volver a recopilar estos datos debido a ciertas modificaciones en el pipeline.

6 Resultados

En este capítulo se van a explicar las diferentes pruebas que se han llevado a cabo para resolver el problema que plantea este proyecto.

6.1 Dataset sin balancear

La primera prueba que se realiza en el proyecto consiste en entrenar el primer modelo de PointNet. En este primer entrenamiento, el dataset creado contiene nubes de puntos con cualquier número de puntos. Además, el dataset está desbalanceado, es decir, las diferentes clases no tienen el mismo número de ejemplos. En las siguientes imágenes, Figura 6.1 y Figura 6.2, se observan las distribuciones del número de ejemplos por clase y dentro de la clase vehículo.

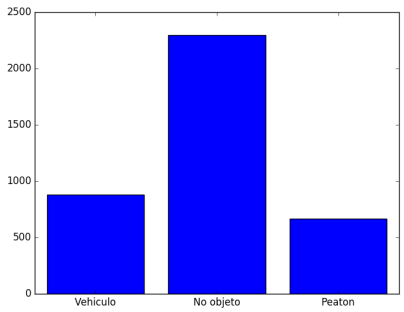

Figura 6.1: Número de ejemplos por clase.

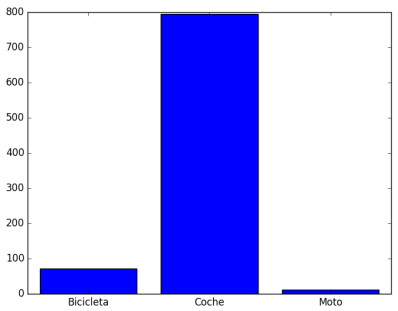

Figura 6.2: Ejemplos en la clase vehículo.

En cuanto a los hiper-parámetros utilizados para el entrenamiento, se han dejado los que vienen por defecto en PointNet. Estos son los siguientes:

- **Número de épocas** = 3000.
- **Batch size** = 32.
- **Learning rate** = 0.001.
- **Optimizador** = Adam.
- **Decay step** = 200000.
- **Decay rate** = 0.8.

A continuación, se describen brevemente los parámetros que aparecen arriba. El número de épocas, como su propio nombre indica, es el número de iteraciones que va a durar el entrenamiento.

El batch size es un parámetro más difícil de explicar pero una vez explicado es más sencillo de entender. Ya se ha explicado que una época consiste en aplicar el forward pass[1] a todos los ejemplos del set. Cada época se divide en un número determinado de iteraciones. En cada iteración la red predice un número determinado de ejemplos, este número es el batch size. Normalmente, se calcula el número de iteraciones de una época en función del número de ejemplos y el batch size para que, al final de una época, la red haya predicho sobre todos los ejemplos.

El optimizador hace referencia al algoritmo de aprendizaje que se aplica en el entrenamiento. Este algoritmo se conoce como descenso del gradiente, donde Adam es un tipo de descenso del gradiente.

El learning rate es un parámetro que indica cuánto se están ajustando los pesos de la red al error. Cuanto mayor sea, el gradiente se va a mover dando saltos más grandes y seguramente no converja. Cuanto más pequeño, la red aprenderá más lento y cabe la posibilidad de que se quede estancada en mínimos locales, mesetas... Por último, los dos parámetros decay se utilizan para disminuir el valor del learning rate a medida que pasan las épocas. Cómo afecta el learning rate al entrenamiento se puede entender mejor en la Figura 6.3.

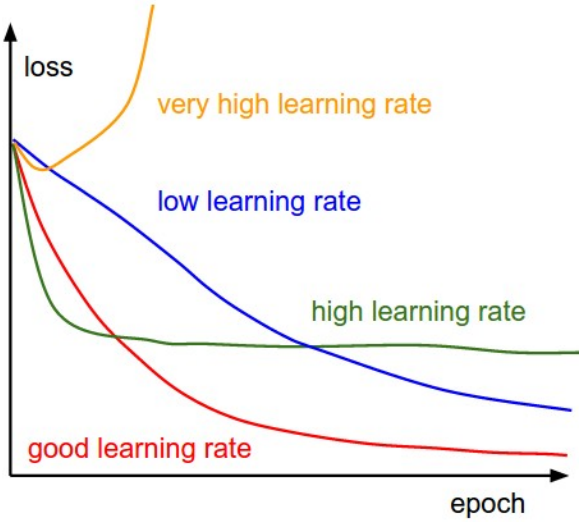

Figura 6.3: Explicación de la variación del parámetro learning rate. Fuente: [29]

[1] Una única predicción de la red.

6.1.1 Resultados del entrenamiento

Aunque el número de épocas establecido era 3000, el entrenamiento se para antes porque, como se verá en la gráfica, se queda estancado durante más de 1000 épocas y no tiene sentido seguir con ese entrenamiento.

En primer lugar, se muestran las gráficas correspondientes al error y la precisión. Estas gráficas muestran la razón por la cual el entrenamiento no ha tenido éxito.

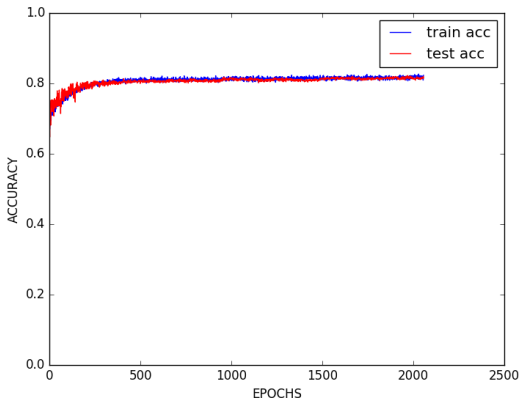

Figura 6.4: Precisión en entrenamiento vs Precisión en validación.

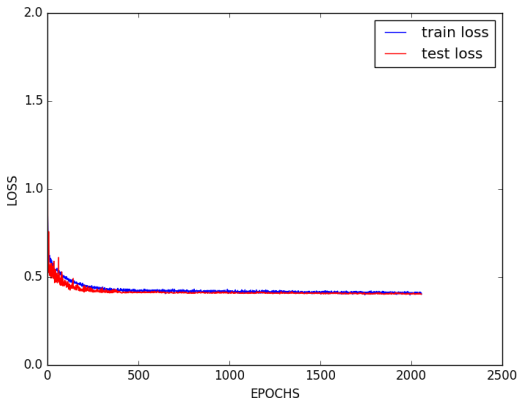

Figura 6.5: Error de entrenamiento vs Error de validación.

Ambas gráficas, Figura 6.4 y Figura 6.5, muestran que en las primeras 200-300 épocas, el entrenamiento se queda estancado en un mínimo local. Después, durante más de 1000 épocas, tanto la precisión como el error solo mejoran en un 2 ó 3 %.

El comportamiento del error se puede identificar con uno de los comportamientos de la Figura 6.3. Este corresponde con el comportamiento en el que el learning rate es demasiado alto. Como ya se ha comentado, si el learning rate es demasiado alto, la red aprenderá rápido al principio del entrenamiento pero luego no convergerá.

Además de las gráficas de precisión y error, se han obtenido las gráficas de precisión de cada clase. Como el dataset está desbalanceado, la red ajusta sus pesos a la categoría que más ejemplos tiene, en este caso es la clase no objeto. Además, también cabe destacar que los ejemplos que representan la clase no objeto son más sencillos de aprender que el resto de clases. Por lo tanto, los resultados obtenidos en cuanto a la precisión en cada clase tienen sentido, la precisión en la clase no objeto ronda el 90-95% durante todo el entrenamiento, mientras que en el resto no supera el 70%.

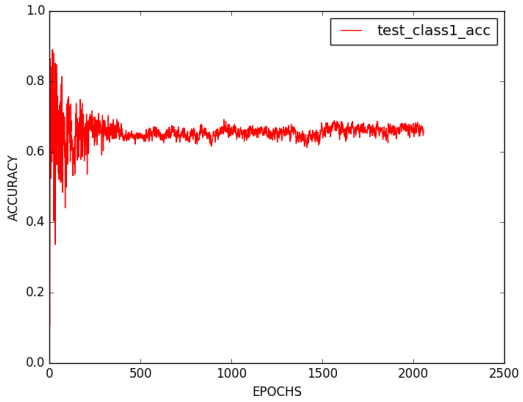

Figura 6.6: Precisión en la clase vehículo.

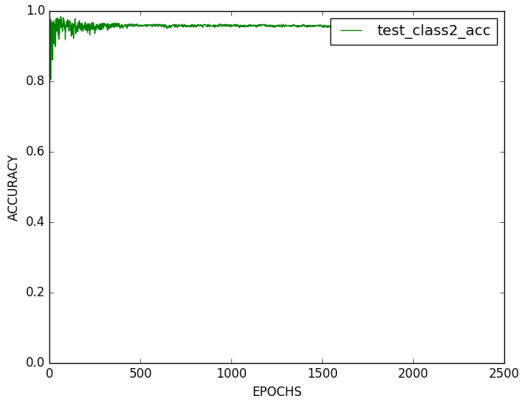

Figura 6.7: Precisión en la clase no objeto.

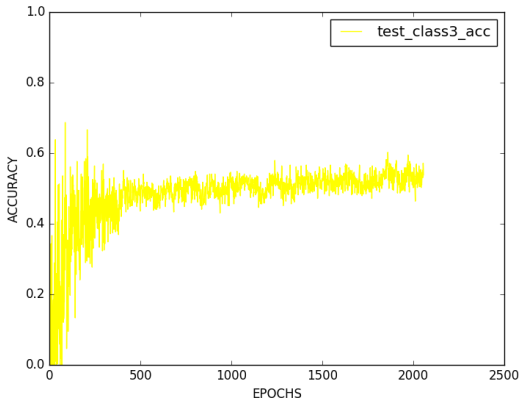

Figura 6.8: Precisión en la clase persona.

Como conclusión de la primera prueba, aunque la precisión obtenida es de un 80%, en realidad no es una medida real de como funciona el modelo porque básicamente ha aprendido a diferenciar lo que es un no objeto del resto, ya que tiene baja precisión en las otras dos clases.

6.2 Dataset balanceado

La segunda prueba que se propone consiste en balancear el dataset y volver a entrenar el modelo para mejorar la precisión de las clases vehículo y persona. Para balancear el dataset, se ejecuta el pipeline durante más semanas que para crear el anterior dataset ya que se intenta elevar el número de vehículos y personas al de no objetos. Para el dataset desbalanceado se reprodujeron 13 semanas, mientras que para este se reproducen 28.

La distribución del número de ejemplos por clase y en la clase vehículo son las mostradas en las Figuras 6.9 y 6.10.

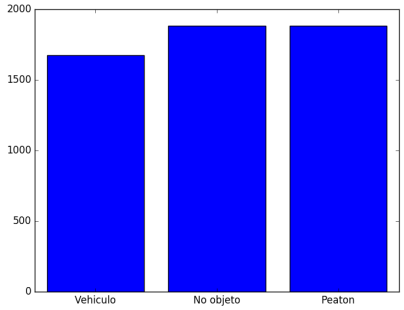

Figura 6.9: Número de ejemplos por clase.

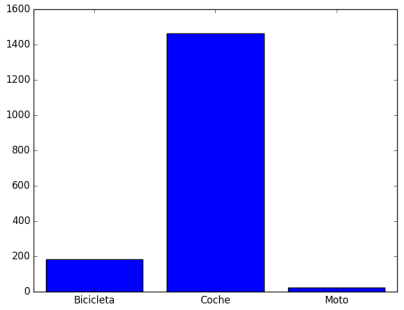

Figura 6.10: Ejemplos en la clase vehículo.

El número de ejemplos por clase ya es prácticamente el mismo en las tres clases, la clase vehículo tiene 1673 ejemplos mientras las clases no objeto y persona tienen 1884. La distribución dentro de la clase vehículo sigue estando desbalanceada, pero tiene sentido ya que es más fácil encontrar coches por la carretera que bicicletas o motos.

6.2.1 Resultados del entrenamiento

Los resultados obtenidos se muestran siguiendo el mismo patrón que en la Sección 6.1.1.

La red entrena durante 3000 épocas, este entrenamiento se mantiene todas las épocas para comprobar si mejora el rendimiento, pero se podría haber parado antes. Al no modificar el learning rate, se produce el mismo comportamiento, la red aprende rápido al principio y luego se mantiene estancada sin mejoras notables durante muchas épocas.

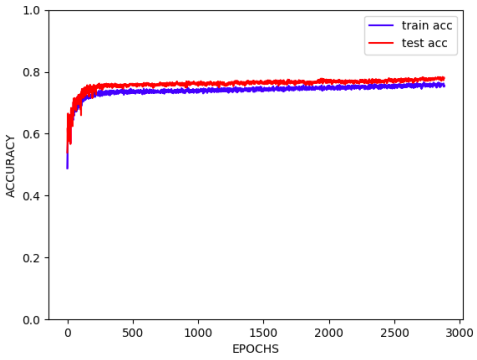

Figura 6.11: Precisión.

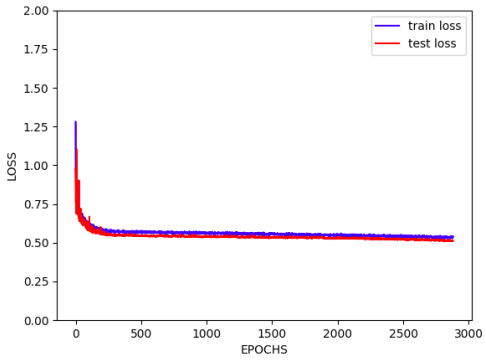

Figura 6.12: Error.

El avance que supone esta prueba se encuentra en el aumento de precisión de las clases vehículo y persona. Esto se consigue gracias al balanceo del dataset.

Mientras que la precisión en el modelo entrenado con el dataset sin balancear es de 60% y 65% para las clases persona y vehículo, respectivamente, en el modelo entrenado con el dataset balanceado las precisiones suben hasta 78% y 70%.

Modelo	Vehículo	Persona
Sin balancear	65%	60%
Balanceado	78%	70%

Tabla 6.1: Comparación de precisiones.

La prueba anterior no se llegó a evaluar detectando semanas nuevas pero este modelo sí se ha incorporado al pipeline para tener una primera toma de contacto y comprobar si cumple el objetivo del proyecto.

El modelo se ejecuta con dos semanas nuevas reuniendo hasta 200 ejemplos, obtiene una precisión de un 74.13%. No es un mal resultado si se compara con los resultados del entrenamiento.

Las gráficas que muestran la precisión en cada clase son las mostradas en las Figuras 6.13 y 6.14.

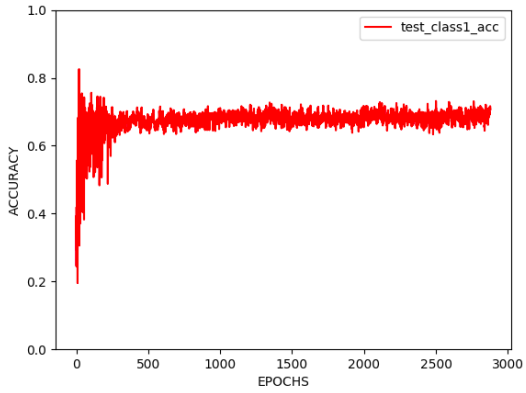

Figura 6.13: Precisión en la clase vehículo.

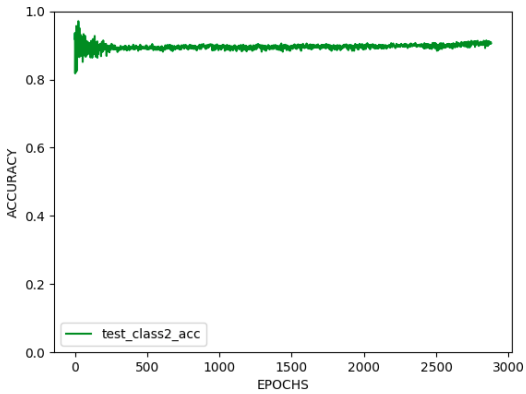

Figura 6.14: Precisión en la clase no objeto.

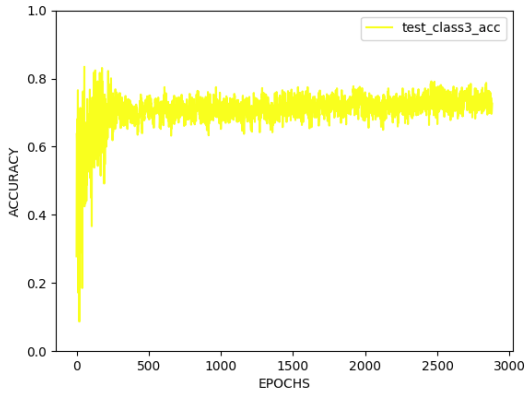

Figura 6.15: Precisión en la clase persona.

Como el dataset no dispone prácticamente de ejemplos de interés, solo se puede evaluar el modelo con 4 ejemplos, de los cuales 2 son personas en un cartel publicitario y los otros 2 son momias egipcias que el sistema ha detectado como personas.

En resumen, este modelo supone la primera prueba real en el pipeline y el resultado es que no cumple el objetivo que busca el proyecto.

6.3 Cambio en learning rate

Antes de tomar una decisión en cuanto a los datos, se realiza una tercera prueba en la que se modifica el learning rate de 0.001 a 0.00001 para ver si mejora el entrenamiento. Los demás parámetros siguen siendo los mismos y el dataset está balanceado.

Las Figuras 6.16 y 6.17 muestran cómo la red ya no aprende tan rápido al principio y mantiene una pendiente más inclinada.

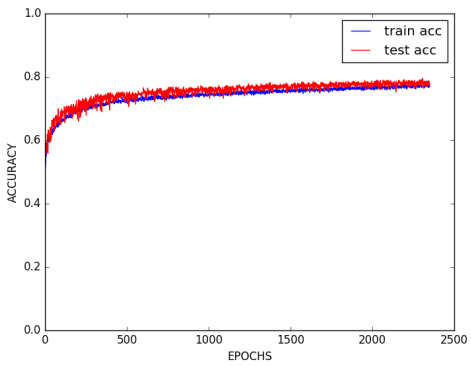

Figura 6.16: Precisión.

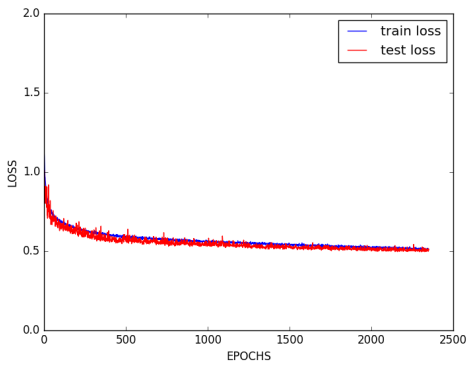

Figura 6.17: Error.

La red se entrena durante 2500 épocas aproximadamente, aunque podría dejarse más mientras la curva tenga una pendiente que a simple vista se vea que está mejorando. La precisión obtenida en la última época ha sido 78%.

En cuanto a los resultados de cada clase, son similares a los de la prueba anterior, mejoran en un pequeño porcentaje. Aunque mejora el entrenamiento, sigue sin cumplirse el objetivo del proyecto.

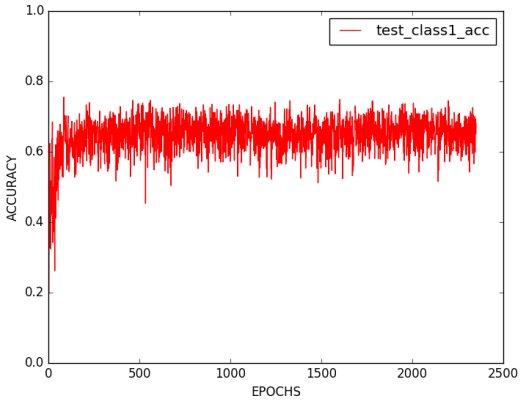

Figura 6.18: Precisión en la clase vehículo.

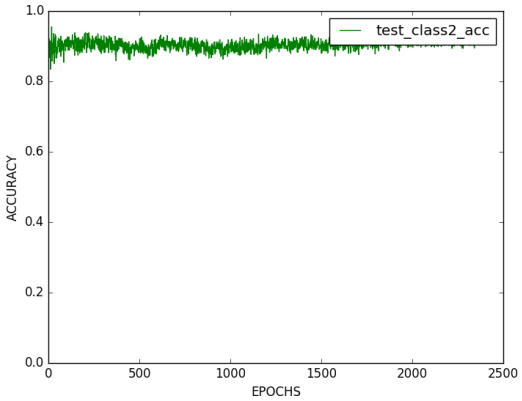

Figura 6.19: Precisión en la clase no objeto.

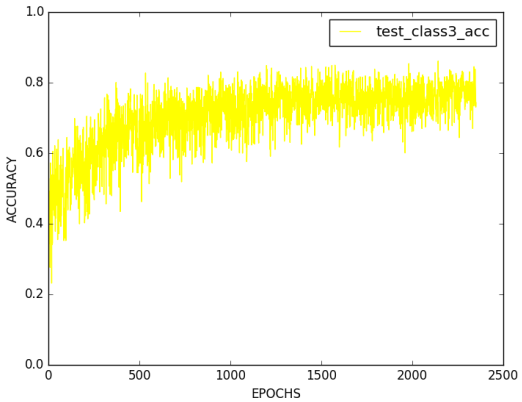

Figura 6.20: Precisión en la clase persona.

6.4 Dataset sintético

Después de probar a modificar hiper-parámetros de la red neuronal, la siguiente prueba se basa en utilizar los datos sintéticos en la clase no objeto. Esta clase va a contener tanto cajas aleatorias, que es lo que contiene en las pruebas anteriores, como datos sintéticos en un 50-50%.

El dataset general está formado por estos datos para la clase no objeto y por datos reales en las otras dos clases, vehículo y peatón. Para esta primera prueba con datos sintéticos, se utilizan 1615 ejemplos para cada clase, como se puede observar en la Figura 6.21. Además, el dataset sigue estando balanceado ya que, como se vio en el Sección 6.2, el entrenamiento obtiene mejores resultados.

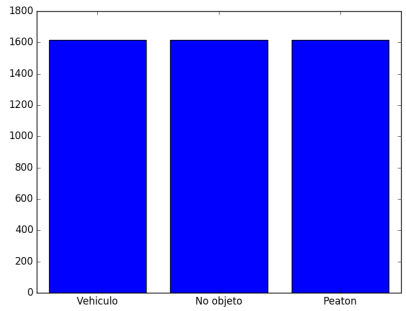

Figura 6.21: Número de ejemplos por clase.

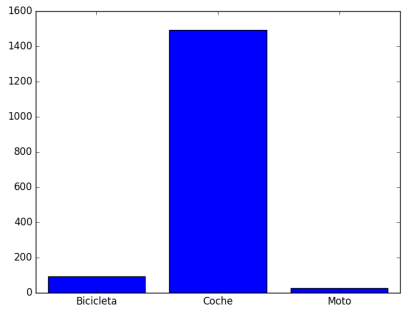

Figura 6.22: Ejemplos en la clase vehículo.

En cuanto a los hiper-parámetros, el único que se modifica es el learning rate, utilizando el mismo que en la anterior prueba, 0.00001, el resto vienen por defecto. El modelo se entrena durante 1000 épocas, las Figuras 6.23 y 6.24 reflejan el entrenamiento de este modelo. Como en el resto de pruebas, también es de gran ayuda disponer de las gráficas que indican la precisión de cada clase.

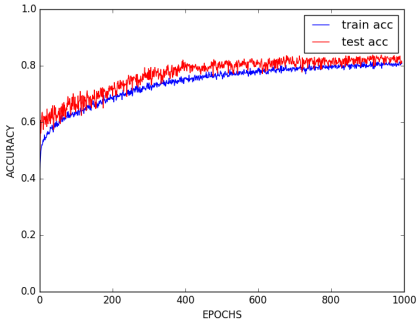

Figura 6.23: Precisión.

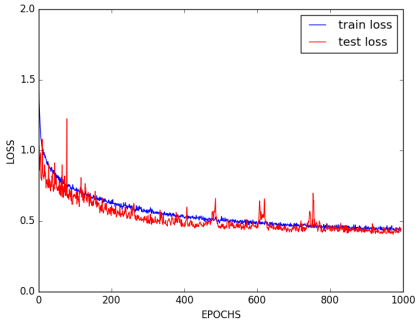

Figura 6.24: Error.

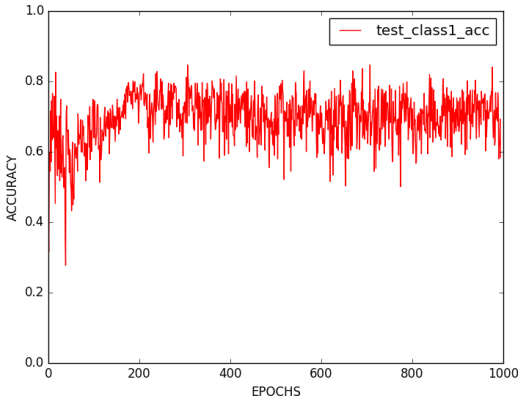

Figura 6.25: Precisión en la clase vehículo.

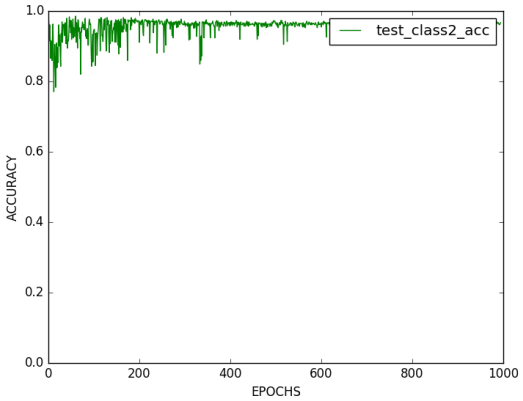

Figura 6.26: Precisión en la clase no objeto.

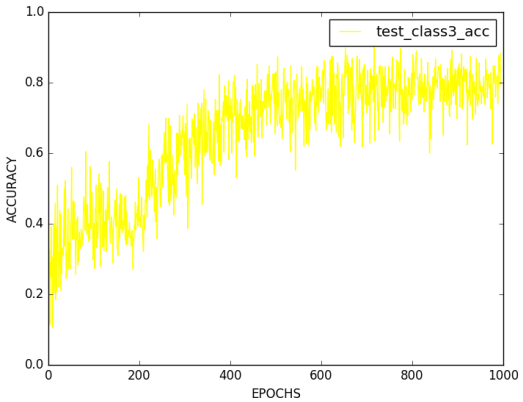

Figura 6.27: Precisión en la clase persona.

Antes de evaluar el modelo con datos reales, se realiza la predicción del conjunto de test de datos sintéticos, obteniendo un 99% de acierto.

Esta prueba se evalúa de una manera más intensa que las anteriores. La evaluación consiste en elegir 3 modelos siguiendo 3 criterios distintos. Estos criterios están basados en los porcentajes de precisión obtenidos por cada clase en la validación durante el entrenamiento.

6.4.1 Primer modelo

El primer modelo elegido es un modelo cuyo porcentaje en la clase vehículo y peatón son lo más alto posible y similares. Como ya se ha visto, el porcentaje de acierto de la clase no objeto es siempre superior al resto, por lo que no se tiene en cuenta para elegir el modelo aunque, obviamente, cuanto más alto sea, mejor.

Concretamente, el modelo corresponde a la época 938, cuyos porcentajes son 77%, 96% y 79% para las clases vehículo, no objeto y peatón, respectivamente. Hasta ahora no se le ha dado mucha importancia a la precisión de cada clase, solamente que fuera lo mayor posible, pero no se ha llegado a indagar como influye cada clase en la precisión total del modelo, por lo que es buen momento para ello.

El modelo se ha probado con 435 muestras de dos semanas diferentes del dataset. La precisión obtenida por este modelo es de 78%, aunque no es fiable del todo porque no se incluyen ejemplos de la clase no objeto para la detección. Para entender mejor los resultados de este modelo, se rellena la matriz de confusión para clase.

	Vehículo	No objeto	Persona
Vehículo	278	0	43
No objeto	0	0	0
Persona	50	1	63

Tabla 6.2: Matriz de confusión modelo 1.

A primera vista lo que más llama la atención es la fila de ceros pero ya se ha comentado que no hay ejemplos de la clase no objeto en esta predicción, se añadirán para el siguiente modelo.

En cuanto a las otras dos clases, es importante comentar que el número de ejemplos de la clase vehículo es mayor ya que aparecen más durante el dataset. Esto influye en los cálculos que se realizan para indicar el rendimiento del modelo, concretamente influye en la precisión de la clase persona.

Se calcula el recall y la precisión de cada clase. El recall indica las predicciones que han sido correctas respecto del número de ejemplos de esa clase, mientras que la precisión indica las predicciones correctas respecto a todas las predicciones de esa clase.

Por ejemplo, para la primera clase sería : recall = 278 * 100 / (278 + 0 + 50) = 84.75% y precisión = 278 * 100 / (278 + 0 + 43) = 83.60%.

	Vehículo	No objeto	Persona
Recall	84.75 %	0	59.43 %
Precisión	86.60 %	0	55.26 %

Tabla 6.3: Recall y precisión.

Sin tener en cuenta la clase no objeto, es obvio que la clase persona no se está detectando bien, ya que tanto el recall como la precisión son bajos. Para el siguiente modelo, se añaden ejemplos de la clase no objeto en la evaluación.

6.4.2 Segundo modelo

El criterio utilizado para este modelo es simplemente coger el que mayor precisión total tiene en la validación, sin tener en cuenta si las precisiones de las clases están descompensadas. El modelo corresponde a la época 908, la precisión general es de 77%, y los porcentajes de cada clase son 68%, 77% y 89 %.

En cuanto a la matriz de confusión, la clase no objeto obtiene buenos resultados como en el entrenamiento. La clase persona detecta bastante mejor que en la prueba anterior ya que el porcentaje es más elevado, mientras que la detección de la clase vehículo empeora, lo cual es lógico debido a su bajo porcentaje.

	Vehículo	No objeto	Persona
Vehículo	149	12	5
No objeto	1	114	0
Persona	69	3	41

Tabla 6.4: Matriz de confusión modelo 2.

	Vehículo	No objeto	Persona
Recall	68.03 %	88.37 %	89.13 %
Precisión	89.76 %	99.13 %	36.28 %

Tabla 6.5: Recall y precisión.

En resumen, se llega a la conclusión de que un modelo que tenga una relación 70-80% en las clases vehículo y persona, puede mejorar la detección de la clase vehículo manteniendo la de

la clase persona.

6.4.3 Tercer modelo

El último modelo de esta prueba sigue el criterio comentado en el que las precisiones de las clases vehículo y persona son 74% y 81%, respectivamente. Para el primer modelo se utilizan dos semanas distintas que para el segundo y tercer modelo. En estos dos últimos, se utilizan las mismas semanas pero diferente número de ejemplos.

	Vehículo	No objeto	Persona
Vehículo	133	22	6
No objeto	1	125	1
Persona	38	3	37

Tabla 6.6: Matriz de confusión modelo 3.

	Vehículo	No objeto	Persona
Recall	77.32 %	83.33 %	84.09 %
Precisión	82.61 %	98.42 %	47.44 %

Tabla 6.7: Recall y precisión.

Los resultados obtenidos para este último modelo son mejores que los dos anteriores, obteniendo una precisión total de 81%. Este modelo consigue mejorar la detección en la clase vehículo y mantener la detección en la clase persona.

El número de vehículos detectados como personas es notable porque se produce en casos determinados como en camiones, furgonetas altas, coches en parkings con una columna delante... Esto es debido a que la red aprende que una persona tiene una forma vertical, entonces, en estas situaciones, detecta que es una persona porque los camiones y furgonetas tienen una forma más vertical que los coches estándar, porque confunde la forma vertical de la columna de un parking con una persona... La causa de estos fallos reside en que la red sólo utiliza las nubes de puntos para detectar por lo que es difícil diferenciar en estos casos.

Para concluir con esta prueba, es importante destacar que con este modelo se mejora la detección respecto al entrenamiento anterior ya que mejora de un 74% a un 81% la precisión total. Sin embargo, ninguno de los tres modelos es capaz de detectar como no objeto los pocos ejemplos de personas en anuncios o de los carteles de las momias, por lo que aún no se ha llegado al objetivo del proyecto.

6.5 Ampliación del tamaño de la caja 3D

Después de la prueba anterior, la cual demuestra que el uso de datos sintéticos mejora los resultados, se siguen utilizando estos datos para la clase no objeto.

Las novedades de esta prueba están en el aumento de la caja 3D que recoge los puntos de dentro del bounding box que devuelve YOLOv3 y en el intercambio manual de los ejemplos de interés de la clase persona a la clase no objeto.

Al modificar el tamaño de la caja 3D, hay que ejecutar el pipeline de nuevo para todas la semanas necesarias para el entrenamiento ya que las nubes son distintas. El número de semanas en las que se ha ejecutado el pipeline es 23, en las cuales se han obtenido 2389 ejemplos para cada clase. El número de ejemplos es mayor que en la Prueba 6.2 debido a que se disminuye el tiempo de espera en la visualización de los datos del dataset, por lo tanto, se utilizan más frames.

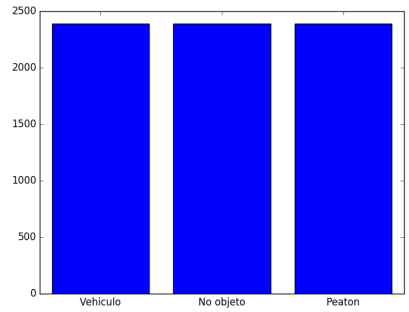

Figura 6.28: Número de ejemplos por clase.

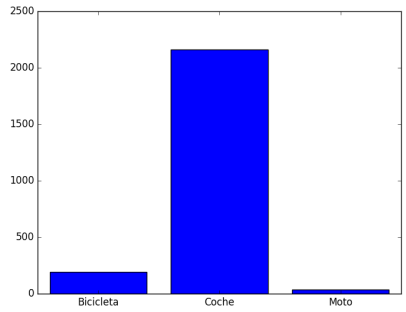

Figura 6.29: Ejemplos en la clase vehículo.

El entrenamiento de este modelo es igual que en las pruebas anteriores, es decir, mantiene el learning rate en 0.00001 y tiene una duración de 1300 épocas.

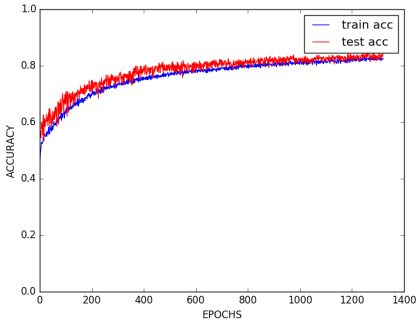

Figura 6.30: Precisión.

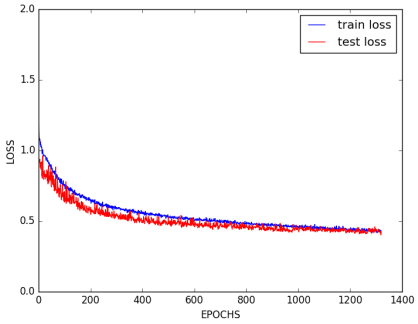

Figura 6.31: Error.

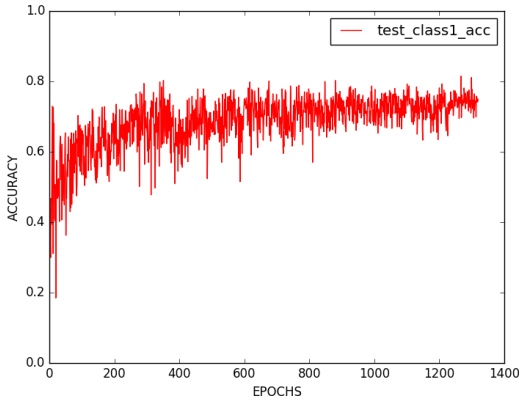

Figura 6.32: Precisión en la clase vehículo.

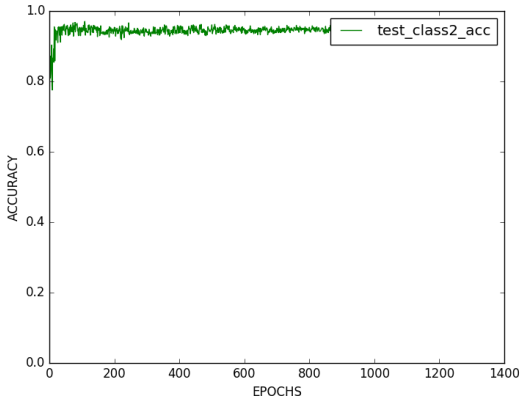

Figura 6.33: Precisión en la clase no objeto.

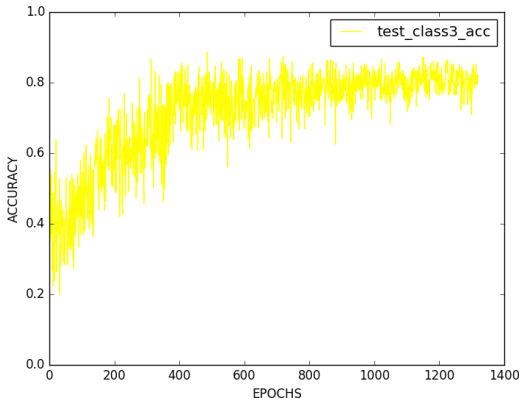

Figura 6.34: Precisión en la clase persona.

El entrenamiento obtiene resultados similares a la prueba anterior, el modelo elegido corresponde al de la época 1252 con una precisión general de 82% y las siguientes precisiones para cada clase: 77%, 95% y 84%.

Para evaluar este modelo se sigue la misma dinámica que en las pruebas anteriores, se cogen 400 ejemplos de dos semanas distintas y, de forma manual, se comprueba si la predicción corresponde con la clase real.

	Vehículo	No objeto	Persona
Vehículo	153	13	6
No objeto	5	178	2
Persona	34	7	50

Tabla 6.8: Matriz de confusión modelo 3.

	Vehículo	No objeto	Persona
Recall	79.69 %	89.89 %	86.21 %
Precisión	88.95 %	96.22 %	54.95 %

Tabla 6.9: Recall y precisión.

Comparando los resultados de este modelo con el mejor modelo de la prueba anterior, se puede observar una ligera mejora en la detección. Mientras que el anterior obtuvo un 81% de precisión general, este modelo obtiene un 85% con los ejemplos utilizados.

Este modelo es el primero que logra detectar algunos ejemplos de interés como no objeto, por lo que supone un gran avance para conseguir el objetivo del proyecto. El hecho de haber añadido los pocos ejemplos para que el modelo los aprenda ha resultado tener algo de éxito. Por lo tanto, si el dataset contuviera más ejemplos reales de interés, aumentaría el número de ejemplos detectados. La Figura 6.35 muestra uno de estos ejemplos, cabe recordar que las etiquetas de cada clase son: 0-vehículo, 1-no objeto y 2-persona/peatón.

Figura 6.35: Ejemplo de interés.

6.6 Pruebas adicionales

Una vez se ha demostrado que el objetivo del proyecto empieza a cumplirse, vamos a realizar dos pruebas para intentar mejorar la detección de los ejemplos de interés, aún sin disponer de más ejemplos reales, y para intentar reducir el número de vehículos que se detectan como personas.

Para la primera prueba es importante recordar que los datos sintéticos se obtienen con el láser y los bloques posicionados uno enfrente del otro. En la mayoría de ocasiones, cuando el coche autónomo detecta estos ejemplos de interés, los va a detectar en los laterales ya que no se van a encontrar dentro de la carretera. Por lo tanto, se recogen los datos sintéticos de la misma forma pero colocados en diagonal.

Estos nuevos datos sintéticos se añaden en la clase no objeto, que está formada por datos sintéticos, tanto rectos como en diagonal, y por las cajas aleatorias que se han utilizado desde la primera prueba.

Después del entrenamiento y de la evaluación de diferentes modelos, esta prueba no consigue mejores resultados que en la anterior, sigue detectando algunos ejemplos de interés pero no consigue detectarlos todos. Además, disminuye la precisión de la detección en general.

Para la segunda prueba la idea es la siguiente, hasta ahora el tamaño de la caja 3D utilizada para recoger la nube de puntos a partir del bounding box de YOLOv3 tenía una forma cuadrada, más enfocada a la detección de personas. En esta prueba, la idea es ampliar el tamaño de la caja y darle una forma más rectangular con el propósito de disminuir el número

de falsos positivos de la clase persona con la clase vehículo.

Una vez modificado el tamaño y forma de la caja 3D, se ejecuta el pipeline para obtener los ejemplos para el dataset. Después de entrenar y evaluar diferentes modelos, esta prueba tampoco mejora los resultados obtenidos por la prueba anterior.

7 Conclusiones

Este proyecto debe cerrarse resumiendo, en primer lugar, la idea principal por la cual se realiza: la detección de ejemplos de personas y vehículos falsos para conducción autónoma.

A lo largo del proyecto se han propuesto diferentes ideas con el fin de alcanzar este objetivo, desde entrenar PointNet para ver si era capaz de detectarlos sin ningún tipo de modificación hasta ejecutar el pipeline entero y buscar estos ejemplos para añadirlos al dataset, pasando por la creación de un dataset de ejemplos sintéticos.

Después de todas estas pruebas, la única que alcanza este objetivo es, quizás, la más simple y obvia: añadir los ejemplos en la clase no objeto. El problema es que el número de ejemplos de interés es demasiado pequeño y era bastante difícil pensar que con 5 ó 6 ejemplos de 2000-3000 pudiera llegar a detectar algún ejemplo bien.

Por lo tanto, teniendo en cuenta este handicap en el dataset, los resultados obtenidos son buenos y muestran que este objetivo podría alcanzarse plenamente con más ejemplos reales. La causa por la cual no hay más ejemplos reales es porque no se dispone del coche autónomo con el que se grabaron los datos. Además, no se ha contemplado la idea de utilizar otro dataset para obtener más ejemplos reales porque el proyecto se ha realizado con el tiempo justo debido a que se empezó en Abril.

El objetivo secundario de este proyecto es comparar los resultados de PointNet con los resultados obtenidos con PointRgbNet ya que este proyecto consiste en reemplazar PointRgbNet por PointNet.

En cuanto a la precisión, PointNet obtiene un 85% al igual que la precisión máxima que obtiene PointRgbNet. Cabe destacar que PointNet solo se ha probado con 400 ejemplos por lo que el porcentaje oscila dependiendo de estos ejemplos. Se ha probado con otros ejemplos del dataset y, aunque obtienen menor precisión, sigue estando entre 80-85%.

En cuanto al tiempo de ejecución, PointNet es más lenta que PointRgbNet. Mientras que PointRgbNet realiza un feedforward en una media de 11 milisegundos, PointNet tarda 1.2 segundos.

Aun así, PointNet es más eficiente que PointRgbNet debido a que tiene una única arquitectura y solo trabaja con nubes de puntos, mientras que PointRgbNet tiene una arquitectura híbrida y utiliza información 2D.

Bibliografía

[1] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Generative and discriminative voxel modeling with convolutional neural networks. *arXiv preprint arXiv:1608.04236*, 2016.

[2] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3d object detection network for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1907–1915, 2017.

[3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[4] D@RKLORD. Exploring the yolo evolution!!, 2019.

[5] Martin Engelcke, Dushyant Rao, Dominic Zeng Wang, Chi Hay Tong, and Ingmar Posner. Vote3deep: Fast object detection in 3d point clouds using efficient convolutional neural networks. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1355–1361. IEEE, 2017.

[6] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[7] Open Source Robotics Foundation. *Documentation - ROS Wiki*, 2020 (accessed July 9, 2020).

[8] Stewart Worrall Miguel Cazorla Francisco Gomez-Donoso, Edmanuel Cruz and Eduardo Nebot. Pointrgbnet: A hybrid 2d/3d cnn for urban object recognition.

[9] A. Garcia-Garcia, F. Gomez-Donoso, J. Garcia-Rodriguez, S. Orts-Escolano, M. Cazorla, and J. Azorin-Lopez. Pointnet: A 3d convolutional neural network for real-time object class recognition. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 1578–1584, 2016.

[10] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[11] Francisco Gomez-Donoso, Felix Escalona, and Miguel Cazorla. Par3dnet: Using 3dcnns for object recognition on tridimensional partial views. *Applied Sciences*, 10(10):3409, May 2020.

[12] Francisco Gomez-Donoso, Alberto Garcia-Garcia, J Garcia-Rodriguez, Sergio Orts-Escolano, and Miguel Cazorla. Lonchanet: A sliced-based cnn architecture for real-time 3d object recognition. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 412–418. IEEE, 2017.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] Divakar Kapil. Yolo v1: Part 1, 2018.

[15] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Andreas Veit, et al. Openimages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://github. com/openimages*, 2(3):18, 2017.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[17] Wei Zhou; Julie Stephany Berrio Perez; Charika De Alvis; Mao Shan; Stewart Worrall; James Ward; Eduardo Nebot. The usyd campus dataset, 2019.

[18] Charles R Qi, Wei Liu, Chenxia Wu, Hao Su, and Leonidas J Guibas. Frustum pointnets for 3d object detection from rgb-d data. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 918–927, 2018.

[19] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *arXiv preprint arXiv:1612.00593*, 2016.

[20] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation, 2017.

[21] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6517–6525, 2017.

[22] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[23] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[24] Shuran Song, Samuel P. Lichtenberg, and Jianxiong Xiao. Sun rgb-d: A rgb-d scene understanding benchmark suite. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[25] Shuran Song and Jianxiong Xiao. Deep sliding shapes for amodal 3d object detection in rgb-d images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 808–816, 2016.

[26] Yunong Tian, Guodong Yang, Zhe Wang, Hao Wang, En Li, and Zize Liang. Apple detection during different growth stages in orchards using the improved yolo-v3 model. *Computers and electronics in agriculture*, 157:417–426, 2019.

[27] Sik-Ho Tsang.

[28] Sik-Ho Tsang. Review: Yolov2 & yolo9000 — you only look once (object detection), 2018.

[29] Hafidz Zulkifli. Understanding learning rates and how it improves performance in deep learning, 2018.

Lista de Acrónimos y Abreviaturas

2DCNN	Red Neuronal Convolucional 2D.
3DCNN	Red Neuronal Convolucional 3D.
FLOPS	Floating Point Operations Per Second.
FPS	Frames Per Second.
ILSVRC	ImageNet Large Scale Visual Recognition Challenge.
IoU	Intersection over Union.
mAP	mean Average Precision.
ROS	Robot Operating System.
RPN	Region Proposal Network.
TFG	Trabajo Final de Grado.