Universidad de Alcalá

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

**Trabajo Fin de Máster**

# "IMPLEMENTACIÓN DE ALGORITMOS DE DEEP LEARNING PARA DETECCIÓN DE OBJETOS TRIDIMENSIONALES EN VEHÍCULOS AUTÓNOMOS"



**Autor:** Sergio Rodríguez Manzanares

**Tutor/es:** Rafael Barea Navarro

2019

Universidad de Alcalá

Escuela Politécnica Superior

UNIVERSIDAD DE ÁLCALA

Escuela Politécnica Superior

**MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL**

Trabajo Fin de Máster

# "Implementación de algoritmos de Deep Learning para detección de objetos tridimensionales en vehículos autónomos"


Autor: Sergio Rodríguez Manzanares
Tutor/es: Rafael Barea Navarro


**TRIBUNAL:**

**Presidente:** Luis Miguel Bergasa Pascual

**Vocal 1:** Noelia Hernández Parra

**Vocal 2:** Rafael Barea Navarro

FECHA: ……

# *AGRADECIMIENTOS*

*En primer lugar, dar las gracias a mi tutor que me ha proporcionado la oportunidad de trabajar en este proyecto y me ha brindado la ayuda que he necesitado para terminarlo. También dar las gracias a Carlos por partida doble, primero por ser un gran compañero durante todos estos años y segundo por proporcionarme soporte durante las últimas fases del proyecto.*

*También me gustaría dar las gracias a mis compañeros durante esta etapa del máster. Siempre es de agradecer que exista compañerismo, pero en este caso, muchas veces se ha podido catalogar de amistad. En especial quiero dar las gracias a mí compañero y amigo Antonio, sin el que esta etapa hubiese sido mucho más complicada. También dar las gracias a Juan Carlos y a Rocío por ser unos grandes amigos y con los que espero seguir contando siempre. Finalmente me gustaría dar las gracias a Esther, que me ha dado la luz que necesitaba en los momentos que más lo necesitaba y que me ha ayudado a ver las cosas desde otro punto de vista, siempre formaremos un equipo perfecto.*

*Por último, dar las gracias a mis padres que han tenido que soportar muchas idas y venidas, historias de clase, conversaciones sobre mis problemas y un largo etcétera que jamás les podré compensar. También a mi hermano Dani y a Marta por hacerme salir cuando estaba saturado y darme consejo cuando era necesario. Todos vosotros habéis sido y siempre seréis un ejemplo a seguir para mí.*

# TABLA DE CONTENIDO

# TABLA DE FIGURAS

# ÍNDICE DE TABLAS

# RESUMEN

El trabajo descrito en este libro consiste en el estudio y la implementación de algunos algoritmos de Deep Learning y redes neuronales para detectar objetos tridimensionales en el ámbito de los vehículos autónomos. En concreto se implementará la arquitectura de Pointpillars [1]. Para ello se pretende utilizar únicamente los datos proporcionados por nubes de puntos obtenidas desde un LIDAR.

Para lograr este objetivo se estudiarán distintos algoritmos y arquitecturas de redes neuronales, se elegirá una para su implementación. La red seleccionada será entrenada con datos obtenidos de la base de datos de KITTI, la cual también será descrita en este trabajo.

**Palabras clave:** Detección, LIDAR, Pointcloud, KITTI, Deep Learning.

# ABSTRACT

The work described in this book consists of the study and implementation of some deep learning algorithms and neural networks to detect three-dimensional objects in the field of autonomous vehicles. Specifically, the architecture of` Pointpillars [1] will be used. The aim is to use only the data provided by point clouds obtained from a LIDAR.

In order to achieve this objective, different algorithms and neural network architectures will be studied and one of them will be chosen for its implementation. The selected network will be trained with data obtained from the KITTI database, which will also be described in this work.

**Palabras clave:** Detection, LIDAR, Pointcloud, KITTI, Deep Learning.

# RESUMEN EXTENDIDO

En este trabajo se desarrolla un análisis y la implementación de una de las principales redes neuronales destinadas a la detección de objetos 3D en el ámbito de la conducción autónoma. Además, se propone como elemento indispensable que la arquitectura se alimente únicamente de los datos proporcionados por un láser LIDAR.

Para realizar este trabajo se ha propuesto estudiar algunas de las arquitecturas existentes y seleccionar aquella que pueda tener un mejor desempeño. Después de realizar un análisis de las arquitecturas actuales se determina que la red Pointpillars puede resultar adecuada para el objetivo de este trabajo.

El trabajo explica el motivo por el cual esta arquitectura es válida para el propósito que se pretende utilizar y la forma en la que se implementa, basándose en un código proporcionado por los autores de la arquitectura. Dentro de la implementación de la red se encuentra el entrenamiento de esta. Se explica cómo se realiza el proceso de entrenamiento utilizando para ello una base de datos llamada KITTI [2].

La base de datos KITTI contiene información de distintos sensores conectados a un coche autónomo, organizados y etiquetados. Esta base de datos tiene como objetivo proporcionar una base de datos sobre vehículos autónomos y crear un marco de referencia con el que comparar distintos algoritmos ya que introduce un benchmark de código abierto.

Se ha comprobado con el benchmark de KITTI que la implementación y modificaciones de Pointpillars han sido válidas.

Durante el desarrollo de este trabajo se han modificado funciones de la implementación de Pointpillars y se ha generado una aplicación portable en Docker que permita su uso en cualquier ordenador. En este trabajo se incluye un manual de cómo utilizar dicho Docker.

# 1 INTRODUCCIÓN

## 1.1 OBJETIVO DEL PROYECTO

El objetivo principal es implementar y adaptar uno de los distintos algoritmos existentes de Deep Learning en el ámbito de los vehículos autónomos para la detección de objetos tridimensionales. Para lograr este objetivo, primero se estudiarán y compararán distintas arquitecturas de redes neuronales que se están utilizando ahora mismo. En segundo lugar, se deberá implementar el código necesario y se realizar el entrenamiento con la base de datos de KITTI. Finalmente, la red obtenida se validará con datos de KITTI.

## 1.2 ANTECEDENTES

Los vehículos autónomos existen desde hace un tiempo bastante pequeño, sin embargo, durante estos pocos años se ha avanzado enormemente en su desarrollo. El principal problema de los vehículos autónomos es la obtención de datos del entorno, un entorno cambiante, hostil y con amenazas completamente inesperadas.

Dentro de los distintos tipos de sensores en lo vehículos autónomos se encuentran las cámaras que informan de lo que se encuentra alrededor del vehículo, las IMU que indican los movimientos del vehículo, el GPS que informa de la posición absoluta del vehículo y finalmente las nubes de puntos tridimensionales que informan, al igual que las cámaras, del entorno en el que se encuentra el vehículo.

Las nubes de puntos tridimensionales se pueden obtener de distintas maneras, mediante mapas de disparidad utilizando técnicas de estéreo visión, mediante cámaras TOF o mediante el uso de la tecnología LIDAR, siendo esta última la más precisa de todas.

El principal problema de las nubes de puntos se encuentra en la detección de objetos dentro de ellas. Como se trata de una tecnología relativamente moderna, no existen muchos algoritmos capaces de detectar objetos sobre las nubes de puntos, al contrario que las imágenes 2D que gozan de todo un campo dedicado a ellas como es la visión artificial y un conjunto de algoritmos sumamente robustos y eficientes. Además, las nubes de puntos poseen las siguientes características que las convierten en elementos difíciles de procesar:

- **Datos desordenados**: Los datos de una nube de puntos se encuentran normalmente desorganizados. Al contario que las imágenes 2D, que poseen un criterio de array de pixeles establecido, las nubes de puntos se componen de una matriz de puntos sin ningún orden prestablecido.
- **Correlación de los puntos:** Los puntos pueden estar o no altamente correlacionados entre sí, teniendo como único elemento diferenciador la distancia entre ellos. Esto supone un problema debido a que el hecho de que un punto este muy cerca de otro no implica que pertenezcan al mismo conjunto.
- **Ausencia de información de color:** En principio, únicamente con tecnología LIDAR, no es posible obtener el color de los puntos que componen la nube de puntos. Esto hace que la segmentación e identificación de objetos sea más complicada.

Debido a esto la mayor parte de técnicas hasta ahora se han basado en la agrupación de conjuntos de puntos que comparten ciertas características geométricas (Coplanaridad, esfericidad, segmento). Estás técnicas permitían la identificación de objetos con forma conocida y complejidad baja.

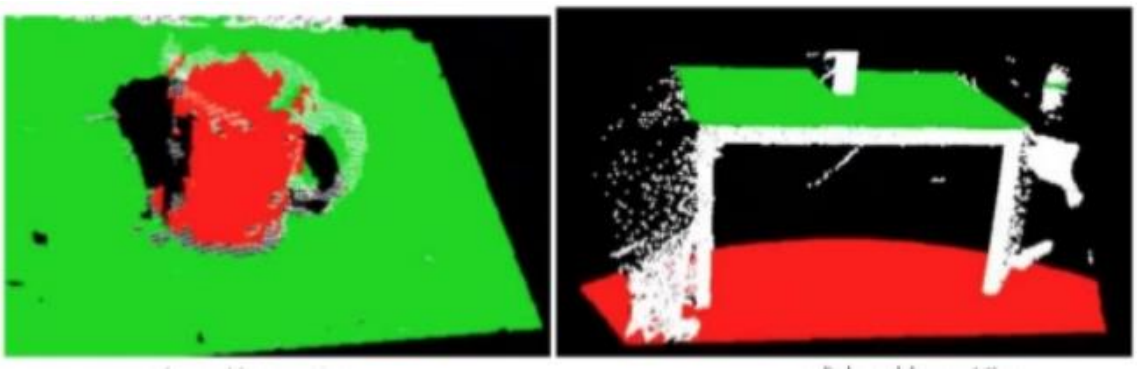


*Figura 1-1: Reducción de la nube de puntos para el entrenamiento*

Sin embargo, estas técnicas carecen de la potencia suficiente para hacer frente a retos modernos como la detección de vehículos, cuya forma depende del diseño y tipo de vehículo, la detección de personas, que es altamente dependiente de la postura del cuerpo, etc. Por ello se han desarrollado redes neuronales capaces de detectar elementos más complejos.

Durante la búsqueda de un algoritmo válido se han estudiado varias técnicas, cada una de ellas basándose en la utilización de redes neuronales, pero con visiones distintas desde el tratamiento de los datos.

## 1.3 ESTRUCTURA DE LA MEMORIA

La memoria se encuentra dividida en seis capítulos con la que se pretende explicar cómo ha sido el proceso de obtención de la red. La estructura de la memoria es la siguiente:

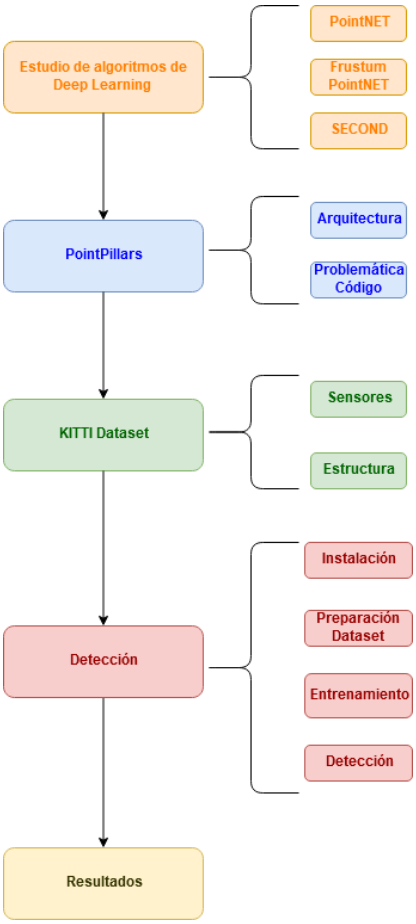



*Figura 1-2: Estructura de la memoria*

En el primer bloque se explican algunos de los algoritmos más interesantes en la detección sobre nube de puntos, indicando los motivos por los cuales no se ha desarrollado su implementación en este trabajo. En el segundo bloque se describe el algoritmo que se ha decidido utilizar, su arquitectura y los distintos problemas que trae su código. En el tercer bloque se define el dataset utilizado. En el cuarto bloque se describe el trabajo realizado, desde la instalación del framework utilizado, hasta las modificaciones realizadas en el código para obtener la detección deseada. Finalmente, en el último bloque se muestran los resultados obtenidos.

# 2 ALGORITMOS DE DEEP LEARNING

## 2.1 INTRODUCCIÓN

En este capítulo se comparan los algoritmos y las arquitecturas principales que se han estudiado para la realizar la implementación. También se indica porque no se han utilizado en el proyecto.

## 2.2 ESTADO DEL ARTE

Actualmente existen varias arquitecturas de redes neuronales destinadas a la detección de objetos mediante nube de puntos. Se dividen principalmente entre dos arquitecturas principales. La primera hace uso de imágenes 2D y nubes de puntos para realizar la clasificación y segmentación, la segunda únicamente utiliza la información de la nube de puntos. En la gráfica se puede observar una comparativa de la tasa de acierto de diversas redes en función de la frecuencia a la que puede funcionar las mismas.

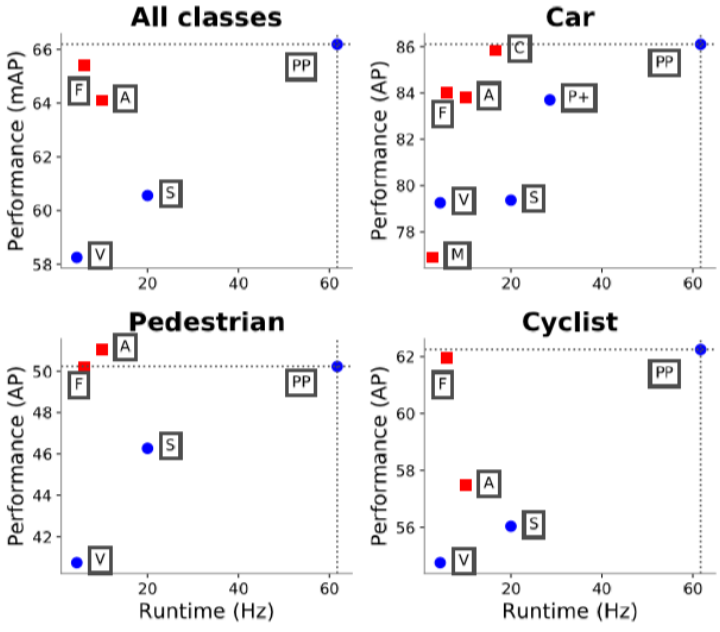



*Figura 2-1: Desempeño de las principales redes VS tiempo de procesado*

Las redes son las siguientes:

F = Frustum Pointnet

A = AVOD

V = VoxelNet

S = SECOND

PP = PointPillars

Como se puede observar, en la mayoría de las redes si el desempeño es elevado, se sacrifica la frecuencia de funcionamiento de la red y viceversa. En lo que resta de capítulo se describirán las arquitecturas de PointNet [3] que, aunque no se encuentra en esta gráfica, fue una de las primeras redes neuronales realmente potentes destinadas a la detección de objetos sobre nubes de puntos, Frustum Pointnet [4], por ser la que mejor desempeño general tiene únicamente por detrás de Pointpillars [1] y finalmente SECOND por ser la que mayor tasa de refresco tiene y el algoritmo más parecido al que se desea implementar.

## 2.3 PointNet

Una de las topologías de red más potentes para su uso con datos de nubes de puntos tridimensionales es la red PointNet la cual se describe en el artículo de Charles R Qi. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation" [3]. En su trabajo propone una red formadas por varias capas MLP. Se distinguen dos partes, la primera encargada de la clasificación de la escena global, la segunda se encarga de la segmentación semántica de cada uno de los puntos. Esto supone una mejora respecto a la mayoría de las redes anteriores ya que estas se centraban únicamente en la escena o únicamente en la segmentación local.

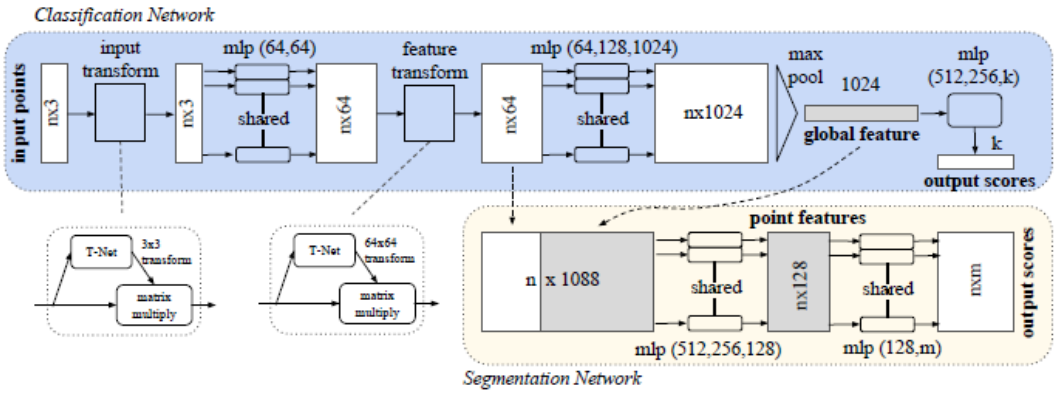



*Figura 2-2: Estructura de la red PointNET*

Los datos de entrada a la red son los datos de la nube de puntos, desorganizados y conteniendo únicamente las coordenadas “x, y, z”.

Primero se pasan los datos por una red denominada T-Net que realiza una transformación afín para convertir es espacio de la nube de puntos en un espacio canónico de forma que este espacio sea similar para cualquier tipo de entrada. A continuación, se realiza la primera detección de características en una red MLP de dos capas con 64 neuronas cada capa. Seguidamente se realiza de nuevo una transformación sobre el espacio de características para buscar un espacio canónico. Finalmente se pasa por una red de 3 capas de 64, 128 y 1024 neuronas respectivamente para obtener una clasificación global de la escena. La red de clasificación devuelve k scores correspondientes a las k clases que se hayan determinado.

La red de segmentación se nutre de las características globales para realizar la segmentación. La salida de esta red son los scores de los n puntos de cada una de las subclases semánticas definidas.

Esta red obtiene unos scores de acierto muy elevados y una precisión en las detecciones muy elevadas también.

El principal problema se encuentra en que esta red ha sido entrenada para un conjunto de objetos que se encuentran en un lugar interior y con un tiempo de procesamiento bastante elevado. El tiempo de procesamiento según su artículo se encuentra en 1 millón de puntos por segundo. Esto supone un problema ya que para realizar el entrenamiento de la red habría que hacer ciertos cambios sobre los datos de entrada o sobre como entiende los datos la red. Además, el tiempo de procesamiento elevado lo hace poco adecuado para una aplicación de tiempo real en la que se utiliza un LIDAR capaz de proporcionar 2,2 millones de puntos.

Por lo tanto, esta red no es válida para su utilización en vehículos autónomos.

## 2.4 FRUSTUM POINTNET

La red de Frustum PointNet no es una red que trate únicamente nube de puntos, se basa en la detección de objetos sobre imágenes 2D para posteriormente realizar una proyección de la caja detectada en 2D sobre la nube de puntos. La siguiente imagen describe su estructura principal:

El procedimiento se divide en tres redes conectadas en cascada, cada una de ellas tiene una función distinta:

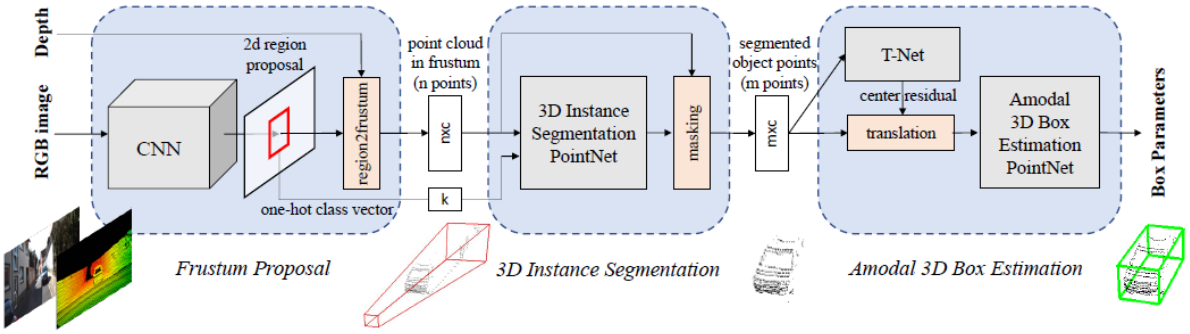



*Figura 2-3: Estructura de la red Frustum PointNet*

1. La primera red es una red convolucional 2D que se encarga de utilizar la información de la imagen 2D para obtener una detección de los distintos objetos dentro de la imagen.

2. Se estima un "frustum" (Sección de pirámide con base cuadrada) y acota la región dentro de la información de profundidad proporcionada por la cámara RGB-D.

3. Esta información se pasa a la segunda red que divide los puntos dentro del "frustum" es distintos clústeres de puntos.

4. Finalmente, los clústeres son pasados a la tercera red que determina las bounding boxes asociadas a las detecciones de las imágenes.

Esta red, como no obtiene los datos directamente de la nube de puntos y proporciona más información que las basadas únicamente en nubes de puntos, tiene porcentaje de acierto más elevado, sin embargo, esto también sacrifica su tiempo de ciclo. Además, esta red necesita información de una cámara RGB-D, mientras que el objetivo de este trabajo es utilizar únicamente los datos proporcionados por un LIDAR. Por lo tanto, tampoco se considera válida para su implementación.

## 2.5 SECOND

La red de SECOND se basa en el uso de redes convolucionales 3D mediante el uso de Voxels. Como elemento diferenciador utiliza un algoritmo llamado "Spatially Sparse Convolutional Networks" [5] que permiten obtener una optimización del proceso muy elevada. Esto es gracias a que no se realiza el proceso de convolución sobre todo el conjunto de datos, sino que únicamente se aplica en aquellas zonas que son de interés. Esto también proporciona una precisión mayor, ya que las zonas en las que no se aplica la operación de convolución permanecen sin alterar.

La estructura general de la red es la siguiente:

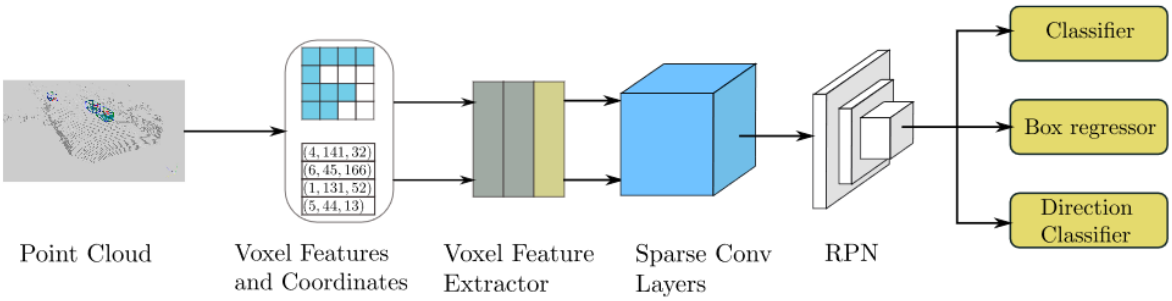



*Figura 2-4: Estructura de la red SECOND*

El algoritmo recibe como datos de entrada la nube de puntos que es convertida a un sistema ordenado de voxels. Estos Voxels son introducidos en una red "FCN" (Fully Connected Network) para obtener las características de cada uno de los voxels. Estas características son introducidas en la red convolucional 3D, la cual convierte el conjunto de características 3D obtenido anteriormente en un conjunto de características 2D de una imagen BEV. Finalmente, esta información se pasa a una red de detección 2D, en este caso es una red similar a la red Multibox SSD (Single-Shot Detector).

Esta red describe un algoritmo similar al que se pretende utilizar en este trabajo. Sin embargo, existe un algoritmo basado en esta red, que utiliza el mismo código fuente y obtiene unos resultados aparentemente mejores.

El algoritmo mencionado es Pointpillars [1], que será el que se aplicará en este trabajo y se describe en el capítulo siguiente.

# 3 POINTPILLARS

## 3.1 INTRODUCCIÓN

En este capítulo se describe la red PointPillars, la cual está basada en el framework de SECOND. También se describe en este capítulo la problemática que se ha encontrado en el código para el uso de esta red.

## 3.2 DESCRIPCIÓN DE LA ARQUITECTURA

Pointpillars se compone de tres etapas principalmente.

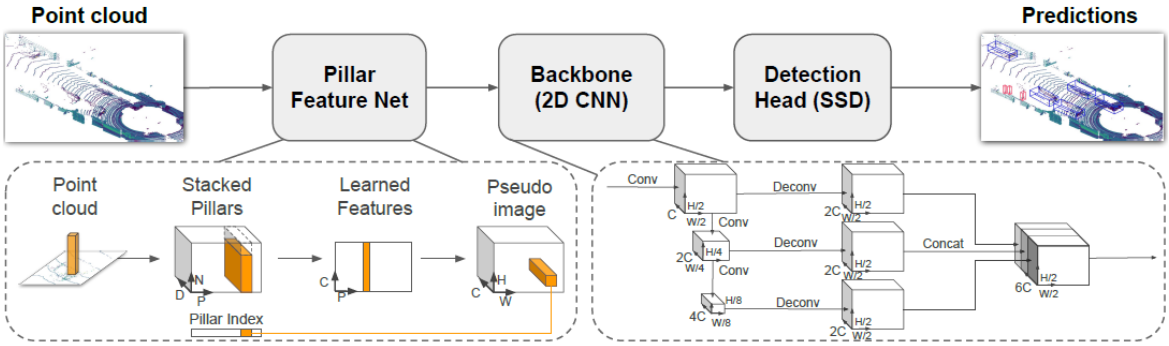



*Figura 3-1: Estructura de la red Pointpillars*

La primera etapa es la etapa en la que se traduce la nube de puntos, desorganizada, en una estructura de pilares organizada para posteriormente agrupar esos pilares en una "pseudo-imagen" sobre la que aplicar una red de convolución 2D. La segunda etapa es la propiamente dicha de la convolución 2D que prepara el mapa de características para la última etapa de detección. Le tercera capa es una red de clasificación para imágenes 2D llamada "Single Shot Detector" (SSD).

### 3.2.1 Primera etapa

Como se ha indicado anteriormente esta etapa es la encargada de traducir la nube de puntos en pilares y agruparlos en una pseudo-imagen. Los datos de entrada a esta primera etapa son los puntos de la nube de puntos agrupados como [x, y, z, r] donde "x, y, z" son las coordenadas del punto y 'r' la reflectancia de este. Para realizar la conversión a pseudo-imagen primero se hace una discretización del plano XY, dividiendo toda la nube de puntos en pilares discretizados. Posteriormente los puntos en el interior de cada pilar son aumentados con la siguiente información: [x, y, z, r, xc, yc, zc, xp, yp] donde ahora los "xc, yc, zc" son la distancia referida a la media de los

puntos en el interior del pilar y los puntos “xp” e “yp” indican la distancia del pilar hasta el pilar central. Este es el primer vector de características D.

Debido a la alta disparidad de las nubes de puntos, la mayor parte de los pilares se encontrarán vacíos y no será necesario su procesamiento. Por ello se seleccionan los pilares no vacíos y se crea un tensor con las dimensiones (D, P, N) donde D son las 9 características descritas antes, P el número de pilares no vacíos y N el número de puntos por pilar. Las dimensiones P y N son seleccionadas para tener unos pilares con el mismo número de puntos y evitar que haya muchos pilares con pocos puntos en su interior o pocos pilares con muchos puntos en su interior.

A continuación, se utiliza una versión reducida de “PointNet” y se genera un nuevo tensor de 2 dimensiones, P que sigue siendo el número de pilares y C que es el número de características de esta primera fase.

Finalmente, los pilares son de nuevo organizados cada uno en correspondiente posición X e Y, como tercera dimensión se encuentra el conjunto de características obtenidos en la “PointNet”, por lo tanto, el tensor de salida de la primera etapa será de dimensiones (C, H, W)

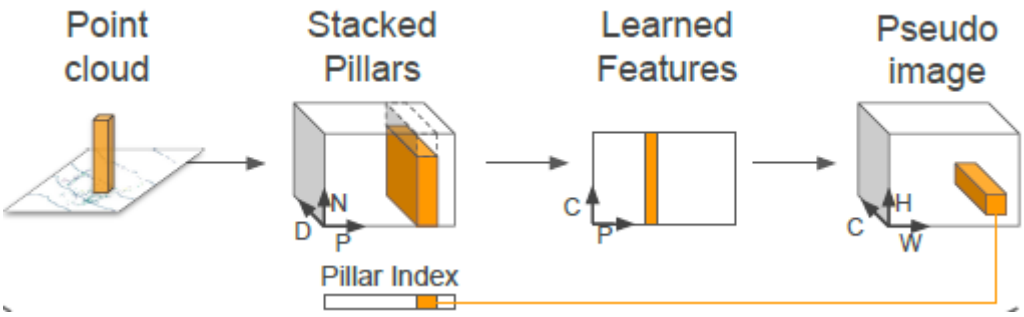



*Figura 3-2: Obtención de pilares en Pointpillars*

Durante esta etapa se utiliza la funcion ReLu como funcion de activación de las neuronas.

### 3.2.2 Segunda etapa

La segunda etapa consiste en la obtención de características a distintas resoluciones. Con esto se busca aumentar la precisión de las predicciones sin sacrificar el contexto general de la escena. Por ello la primera parte de esta etapa consiste en realizar convoluciones de forma que la dimensión de las características aumente para posteriormente realizar una convolución transpuesta y devolver a las mismas dimensiones de ancho y alto. Finalmente, estas características son concatenadas y se pasan a la siguiente etapa.

### 3.2.3 Tercera etapa

La tercera etapa es un detector ampliamente utilizado denominado SSD (Single Shot Detector). Esta arquitectura de detección es similar a la YOLO, la red combina predicciones de múltiples mapas de características con diferentes resoluciones para manejar objetos de varios tamaños. Este tipo de arquitecturas es realmente eficiente en cuanto a tiempo de procesamiento ya que elimina la necesidad de tomar varias imágenes para realizar la detección y con una única imagen es suficiente para realizar la detección.

## 3.3 FUNCIÓN DE ERROR

La función de error que determina la precisión de la red es una función que se tiene que definir en cada problema concreto. En este caso se utiliza la misma función de error que se utiliza en la red SECOND. Esta función en realidad está compuesta de tres funciones, una función de error para la localización, otra para la dirección y otra para la clasificación.

### 3.3.1 Función de error de la localización

La localización y el tamaño de las cajas están definidas por sus coordenadas (x, y, z), su tamaño (w, l, h) y su ángulo (θ). Por ello la función de error es una combinación de estos parámetros. Se define el error en cada uno de los parámetros como:

$$\Delta x = \frac{x^{gt} - x^a}{d^a}, \Delta y = \frac{y^{gt} - y^a}{d^a}, \Delta z = \frac{z^{gt} - z^a}{h^a}$$
$$\Delta w = \log \frac{w^{gt}}{w^a}, \Delta l = \log \frac{l^{gt}}{l^a}, \Delta h = \log \frac{h^{gt}}{h^a}$$
$$\Delta \theta = \sin\left(\theta^{gt} - \theta^a\right),$$

Donde los subíndices “gt” significa “Groundtruth” y “a” es el “anchor box” determinado por la red. La distancia “dª” se define como la diagonal del “anchor box”. Finalmente se aplica la función “SmoothL1” al sumatorio de todos los errores.

$$\mathcal{L}_{loc} = \sum_{b \in (x,y,z,w,l,h,\theta)} \text{SmoothL1}\left(\Delta b\right)$$

La funcion SmoothL1 se define como:

$$\begin{cases} 0.5x^2 & si\ |x| < 1 \\ |x| - 0.5 & si\ |x| \geq 1 \end{cases}$$

### 3.3.2 Función de error de la dirección

El principal problema de la detección de objetos con cubos es conocer la orientación del cubo. Un cubo por su forma geométrica no tiene un lado frontal o trasero, sin embargo, los vehículos y las personas si poseen un lado frontal. Para esto se utiliza una clasificación "Softmax" que determina si el cubo debe estar girado o no. La función de error en este caso, por ser una clasificación binaria, no necesita combinación alguna con el resto de los parámetros.

### 3.3.3 Función de error de la clasificación

La función de error de clasificación se utiliza la función de error focal que se define de la siguiente manera:

$$\mathcal{L}_{cls} = -\alpha_a \left(1 - p^a\right)^{\gamma} \log p^a$$

Donde "$p^a$" es la probabilidad de obtener un "anchor box" determinado. En referencia al artículo original la configuración de $\alpha = 0{,}25$ y $\Upsilon = 2$ obtiene buenos resultados.

### 3.3.4 Función de error total

La función de error total es:

$$\mathcal{L} = \frac{1}{N_{pos}} \left(\beta_{loc}\mathcal{L}_{loc} + \beta_{cls}\mathcal{L}_{cls} + \beta_{dir}\mathcal{L}_{dir}\right)$$

Donde "$\beta_{loc}$" es el valor de ponderación de la función de error de la localización y su valor es 2, "$\beta_{cls}$" es el valor de ponderación de la función de error de la clasificación y su valor es 1, "$\beta_{dir}$" es el valor de ponderación de la función de error de la dirección y su valor es 0,2. El parámetro "$N_{pos}$" es el número de detecciones válidas.

## 3.4 PROBLEMÁTICA DEL CÓDIGO

El código de evaluación de Pointpillars se puede encontrar en GitHub [6]. Este código se basa en el framework de SECOND, que fue desarrollado para el entrenamiento y la evaluación de redes destinadas a su uso en vehículos autónomos.

El hecho de que el código se encuentre en GitHub facilita mucho la tarea de implementación de la red, sin embargo, se han encontrado diversos problemas para realizar la implementación de esta.

El código que proporcionan en GitHub tiene los siguientes problemas:

- **Versiones:** Los creadores de Pointpillars proporcionan una página donde se puede descargar la versión con la que se ha escrito el artículo. Esta versión, sin embargo, ya no es compatible con algunos módulos que utilizan de Python en el framework de SECOND. La página de SECOND si llevaba un mantenimiento más activo y proporcionó una actualización del framework compatible con los nuevos módulos, sin embargo, durante esta migración algunas de las funciones que se habían desarrollado para Pointpillars dejaron de existir, llegando incluso a no poder utilizarse la red de Pointpillars.

- **Instalación:** Debido al problema con las versiones, la instalación del framework de SECOND no es sencilla y requiere de ciertos conocimientos a pesar de que disponen de guías para realizar la instalación.

- **Flexibilidad:** El código fuente de SECOND se encuentra disponible en GitHub también por si se desean realizar modificaciones. Sin embargo, este código no es sencillo de leer llamando a múltiples subrutinas cuyo código fuente está íntimamente ligado con la casuística del problema y que cuya modificación afectaría a su uso con otro tipo de problema. El framework en si no es flexible y requiere adaptar los módulos de Python para el problema que se quiere abordar.

# 4 KITTI DATASET

## 4.1 INTRODUCCIÓN

Para el entrenamiento y validación de la red se ha utilizado el dataset de KITTI. El dataset de KITTI es un dataset especializado de vehículos autónomos. El dataset se compone de datos de entrenamiento etiquetados y datos de test sin etiquetar de todos los sensores asociados al vehículo. A continuación, se describen todos los sensores y el formato de los datos.

## 4.2 DESCRIPCIÓN DE LOS SENSORES

El vehículo de KITTI se compone de los siguientes:

- 1 x GPS/IMU (OXTS RT 3003)
- 1 x LIDAR (Velodyne HDL-64E)
- 2 x cámaras en escala de grises (Point Grey Flea 2 (FL2-14S3M-C)
- 2 x cámaras a color (Point Grey Flea 2 (FL2-14S3C-C)
- 4 x lentes vari-focales

En este caso, el sensor del que necesitamos principalmente los datos es del sensor LIDAR. Sin embargo, es importante conocer la posición de cada uno de los sensores ya que, para cada uno de los sensores, el eje de coordenadas es diferente. En la siguiente imagen se puede observar la posición de cada uno de los sensores y el nombre de la matriz de transformación para el intercambio de coordenadas.

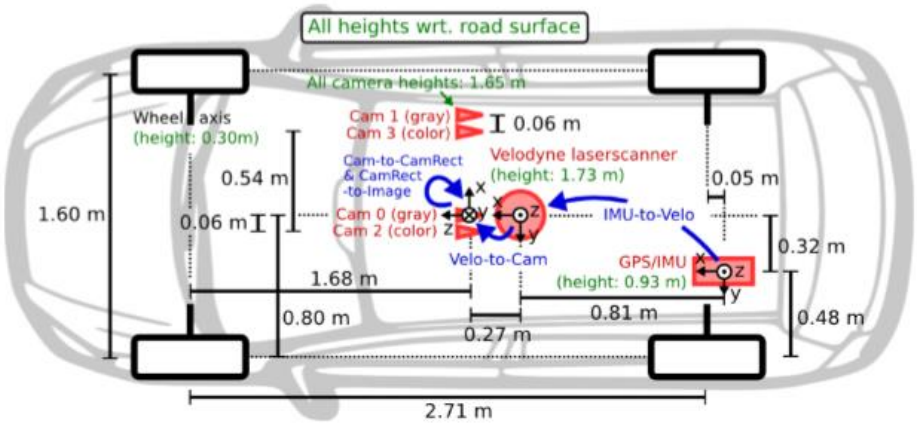



*Figura 4-1:Ubicación de los sensores en KITTI*

Los ejes de coordenadas se encuentran en la siguiente disposición:


*Figura 4-2: Ubicación de los ejes de coordenadas en KITTI*

El eje de coordenadas azul es el eje utilizado para los puntos obtenidos por el LIDAR. El eje coordenadas rojo es el utilizado por las cámaras en escala de grises y de color. Finalmente el eje de coordenadas verde es el utilizado por el GPS/IMU.

## 4.3 ORGANIZACIÓN DEL DATSET

El dataset de KITTI se encuentra organizado siguiendo la siguiente estructura:

- KITTI DATSET
    - KITTI_dataset_object
        - Data_object_calib
        - Data_object_image2
        - Data_object_image3
        - Data_object_image_CNN
        - Data_object_label
        - Data_object_velodyne
        - Devkit_object
    - KITTI_dataset_tracking
        - Data_tracking_calib
        - Data_tracking_det_2_lsvm
        - Data_tracking_det_2_regionlets
        - Data_tracking_image2
        - Data_ tracking _image3
        - Data_ tracking _image_CNN
        - Data_ tracking _label
        - Data_tracking_oxts
        - Data_ tracking _velodyne
        - Devkit_tracking

En este caso se realiza el entrenamiento utilizando el KITTI_dataset_object. A continuación, se describe lo que contiene cada una de las carpetas de este dataset.

- **Data_object_calib:** Contiene ficheros TXT de cada una de las imágenes del dataset. En estos ficheros TXT se guarda la información de las matrices de calibración de las cámaras, y las matrices de cambio de coordenadas entre los distintos ejes de los sensores. La forma de este archivo de texto es la siguiente:

```
P0: 7.070493000000e+02 0.000000000000e+00 6.040814000000e+02
0.000000000000e+00 0.000000000000e+00 7.070493000000e+02
1.805066000000e+02 0.000000000000e+00 0.000000000000e+00
0.000000000000e+00 1.000000000000e+00 0.000000000000e+00
P1: 7.070493000000e+02 0.000000000000e+00 6.040814000000e+
02 -3.797842000000e+02 0.000000000000e+00 7.070493000000e+02
1.805066000000e+02 0.000000000000e+00 0.000000000000e+00
0.000000000000e+00 1.000000000000e+00 0.000000000000e+00
P2: 7.070493000000e+02 0.000000000000e+00 6.040814000000e+02
4.575831000000e+01 0.000000000000e+00 7.070493000000e+02
1.805066000000e+02 -3.454157000000e-01 0.000000000000e+00
0.000000000000e+00 1.000000000000e+00 4.981016000000e-03
P3: 7.070493000000e+02 0.000000000000e+00 6.040814000000e+
02 -3.341081000000e+02 0.000000000000e+00 7.070493000000e+02
1.805066000000e+02 2.330660000000e+00 0.000000000000e+00
0.000000000000e+00 1.000000000000e+00 3.201153000000e-03
R0_rect: 9.999128000000e-01
1.009263000000e-02 -8.511932000000e-03 -1.012729000000e-02
9.999406000000e-01 -4.037671000000e-03 8.470675000000e-03
4.123522000000e-03 9.999556000000e-01
Tr_velo_to_cam:
6.927964000000e-03 -9.999722000000e-01 -2.757829000000e-03 -2.457
729000000e-02 -1.162982000000e-03
2.749836000000e-03 -9.999955000000e-01 -6.127237000000e-02
9.999753000000e-01
6.931141000000e-03 -1.143899000000e-03 -3.321029000000e-01
Tr_imu_to_velo: 9.999976000000e-01
7.553071000000e-04 -2.035826000000e-03 -8.086759000000e-01 -7.854
027000000e-04 9.998898000000e-01 -1.482298000000e-02
3.195559000000e-01 2.024406000000e-03 1.482454000000e-02
9.998881000000e-01 -7.997231000000e-01
```

*Figura 4-3: Ejemplo de matrices de calib_data*

  - **P0 (3x4):** Matriz de proyección cámara izquierda de escala de grises
  - **P1 (3x4):** Matriz de proyección cámara derecha de escala de grises
  - **P2 (3x4):** Matriz de proyección cámara izquierda de color rectificada
  - **P3 (3x4):** Matriz de proyección cámara derecha de color rectificada
  - **R0_rect (3x3):** Matriz de rotación de rectificada a no rectificada
  - **Tr_velo_to_cam (3x4):** Matriz transformación Velodyne a coordenadas de cámaras.
  - **Tr_imu_to_velo (3x4):** Matriz transformación IMU a coordenadas de Velodyne.

- **Tr_cam_to_road (3x4):** Matriz transformación cámaras a coordenadas de carretera.

- **Data_object_image2:** Este fichero contiene las imágenes a color obtenidas por la cámara izquierda.
- **Data_object_image3:** Este fichero contiene las imágenes a color obtenidas por la cámara derecha.
- **Data_object_image_CNN:** Contiene las imágenes de segmentadas por colores en función del objeto.
- **Data_object_label:** Este fichero contiene las etiquetas de los objetos que se encuentran en las imágenes. Tiene la siguiente estructura:
  - **Type:** Indica el tipo de objeto detectado, distinguiendo entre "Car", "Truck", "Van", "Cyclist", "Pedestrian", "Person_sitting", "Tram", "Misc" o "DontCare".
  - **Truncated:** Indica cuanto de oculto se encuentra el objeto por encontrarse fuera de la imagen. Puede tomar valores entre 0 y 1 en función de su nivel de truncamiento.
  - **Occluded:** Indica cuanto de oculto se encuentra el objeto por haber otro objeto delante de él. Puede tomar valores entre 0 y 3 en función de su nivel de oclusión.
  - **Alpha:** Indica el ángulo de observación del objeto en la imagen. Es decir, define el ángulo de la trayectoria entre el vehículo y el objeto.
  - **Bbox:** Indica la bounding box del objeto definido como el píxel más arriba y más a la izquierda, y el pixel más abajo y más a la derecha.
  - **Dimensions:** Indica las dimensiones del objeto en 3D y en metros.
  - **Location:** Indica la posición del centroide según el eje de coordenadas de la cámara.
  - **Rotation_y:** Indica la rotación del objeto referenciado al eje Y del eje de coordenadas de la cámara.

    - **Score:** Este parámetro no se encuentra en los archivos de label proporcionados por KITTI, sin embargo, en los generados por la red si deben estar para poder ser evaluados. Indica la seguridad que tiene la predicción

- **Data_object_velodyne:** Contiene las nubes de puntos de cada una de las imágenes. La nube de puntos se encuentra codificada en forma de matriz binaria con valores en float. La matriz tiene dimensiones de 4 x N, donde N son los puntos y los cuatro parámetros son [x, y, z, r], donde los tres primeros hacen referencia a las coordenadas y el ultimo a la reflectividad de la medida. La reflectividad indica la intensidad de retorno del láser emitido. Este valor depende en gran medida de las condiciones externas.

- **Devkit:** En este fichero se encuentra el código oficial de KITTI que permite obtener una evaluación de las predicciones. El código devuelve unas gráficas que indican el desempeño de la detección.

El KITTI_dataset_tracking, a diferencia del KITTI_dataset_object, no son imágenes aisladas, sino que son imágenes consecutivas de una misma ruta. Los ficheros que se encuentran el KITTI_dataset_tracking son similares a las carpetas que se encuentran el KITTI_dataset_object, con la diferencia de la inclusión de unos ficheros con los datos de tracking que no se describen ya que no son objeto de este trabajo. Otra diferencia es que en el KITTI_dataset_object los labels se encuentran en archivos independientes para cada imagen mientras que en el KITTI_dataset_tracking se encuentran todas en un mismo archivo.

# 5 DETECCIÓN DE OBJETOS TRIDIMENSIONALES

## 5.1 INTRODUCCIÓN

En este capítulo se detalla la implementación realizada desde el inicio, abarcando el proceso de instalación de las herramientas necesarias, la preparación de los datos de entrenamiento y testing, el entrenamiento de la red, las modificaciones sobre el código para realizar detecciones y los procesos de evaluación.

La arquitectura del sistema obtenido es la siguiente:

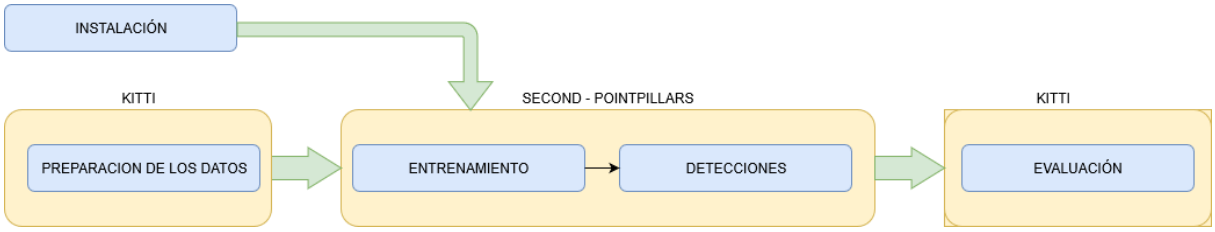



*Figura 5-1: Estructura del sistema realizado*

## 5.2 PROCESO DE INSTALACIÓN DEL FRAMEWORK SECOND

### 5.2.1 Requisitos del PC

El equipo debe contar con el siguiente software instalado:

- Ubuntu 16.04 o superior
- Python 3.6 o superior
- Pytorch 0.4.1 o superior e inferior a Pytorch 1.2.0

Además, es necesario disponer de una tarjeta gráfica que sea capaz de funcionar con CUDA. La versión que se va a utilizar en este caso será la versión de CUDA 10.1 con el driver de NVIDIA 430.40. La tarjeta gráfica utilizada es una GTX 1060 6GB.

### 5.2.2 Procedimiento de instalación

El proceso de instalación es similar al que se indica en la página de GitHub de Pointpillars y al que se indica en la página de GitHub de SECOND [7]. Sin embargo, hay que tener en cuenta que

el código de Pointpillars depende de distintos módulos que debido a actualizaciones es posible que hayan dejado de ser compatibles con el código.

El proceso es el siguiente:

1. Clonar el código desde la URL: https://github.com/nutonomy/second.pytorch

```
git clone https://github.com/nutonomy/second.pytorch.git
```

También es posible descargar el código desde la página de SECOND, sin embargo, la versión 1.6.0 de SECOND no es compatible con ciertos módulos destinados al uso de redes convolucionales.

Es recomendable utilizar los entornos de Anaconda para evitar conflictos entre los módulos de Python, especialmente si se utiliza junto con otro código desarrollado en Python.

2. Crear el entorno de Conda.

```
conda create -n pointpillars python=3.7 anaconda
source activate pointpillars
```

3. Instalar los paquetes necesarios dentro del entorno

```
pip install --upgrade pip
pip install fire tensorboardX
conda install shapely pybind11 protobuf scikit-image numba pillow
conda install pytorch=1.1.0 torchvision -c pytorch
conda install google-sparsehash -c bioconda
```

Es importante remarcar que el paquete de pytorch que se instale debe ser superior a la versión 0.4, sin embargo, no puede superar la versión 1.1.0, esto es debido a que en la versión 1.2.0 de pytorch se elimina el soporte para datos del tipo “boolean”.

4. Instalar SparseConvNet, una librería que permite a Pytorch trabajar con redes neuronales convolucionales en distintas dimensiones. La información proporcionada en el Github de Pointpillars no se encuentra actualizada, a continuación, se indica el procedimiento correcto.

```
git clone https://git hub.com/facebookresearch/SparseConvNet
cd SparseConvNet/
bash build.sh
```

5. Como paso no incluido en el proceso descrito por Pointpillars. Para la visualización de los datos será necesario instalar los siguientes paquetes:

```
conda install -c anaconda pyopengl
pip install pyqtgraph
```

6. Añadir las variables de entorno

```
export NUMBAPRO_CUDA_DRIVER=/usr/lib/x86_64-linux-gnu/libcuda.so
export NUMBAPRO_NVVM=/usr/local/cuda/nvvm/lib64/libnvvm.so
export NUMBAPRO_LIBDEVICE=/usr/local/cuda/nvvm/libdevic
```

7. Añadir el módulo de SECOND a PYTHONPATH

```
sudo gedit ./bashrc
export PYTHON PATH = /ruta/to/SECOND/
```

## 5.3 PREPARACIÓN DEL DATASET

Pointpillars requiere para su entrenamiento de una estructura determinada del Dataset. El dataset utilizado en este caso procede de la base de datos KITTI. La estructura de datos debe ser la siguiente:

```
└── KITTI_DATASET_ROOT
    ├── training    <-- 7481 train data
    |   ├── image_2 <-- for visualization
    |   ├── calib
    |   ├── label_2
    |   ├── velodyne
    |   └── velodyne_reduced <-- empty directory
    └── testing     <-- 7580 test data
        ├── image_2 <-- for visualization
        ├── calib
        ├── velodyne
        └── velodyne_reduced <-- empty directory
```

*Figura 5-2: Estructura del dataset de entrada a la red*

A continuación se indica lo que contiene cada una de las carpetas:

- **Training:** Contiene las imágenes de entrenamiento, con datos de calibración, validación y datos etiquetados.

- **Image_2:** Contiene imágenes 2D. Se utiliza únicamente para visualización y crear nubes de puntos reducidas, en ningún momento interviene en el entrenamiento.
- **Calib:** Contiene los datos de calibración de cada una de las imágenes. Entre ellos se encuentran las matrices de calibración de las cámaras, las matrices de traslación y rotación del velodyne y de la IMU.
- **Label_2:** Contiene los datos etiquetados de cada una de las imágenes.
- **Velodyne:** Contiene las nubes de puntos en formato de matriz binaria. Los datos dentro de estas matrices tienen un tamaño de N x 4 donde N es el número de puntos y se representan las coordenadas X, Y, Z y la reflectancia de la medida.
- **Velodyne_reduced:** Contiene las nubes de puntos reducidas generadas a partir de las nubes de puntos de velodyne y las imágenes 2D. Esto es debido a que KITTI únicamente tiene etiquetados los datos que se visualizan en la imagen 2D, por ello para realizar un entrenamiento correcto es necesario delimitar la zona sobre la que se tienen datos etiquetados.

- **Testing:** Contiene imágenes sin datos etiquetados, se utilizan para validar el sistema.
  - Las carpetas dentro de este directorio con el mismo nombre que en el caso de training contiene la misma clase de información.

A continuación, hay que leer los datos de calibración de cada una de las imágenes y transformarlos para poder pasarlos como parámetros de entrada a la red.

Para ello es necesario ejecutar el siguiente comando:

```
python create_data.py create_kitti_info_file --data_path=KITTI_DATASET_ROOT
```

Esta función, obtiene de cada una de las imágenes de KITTI que se han seleccionado para el entrenamiento las matrices de calibración de las cámaras, y las matrices de translación entre el velodyne, la cámara, la IMU y la carretera. También obtiene el path de las imágenes, el path de velodyne, el número de imagen y el número de características o ejes del archivo de la matriz binaria del velodyne. Todo esto lo guarda en un diccionario de Python por cada imagen. El archivo que crea contiene un diccionario de diccionarios de todas las imágenes.

| Índice | Tipo | Tamaño | Valor |
|---|---|---|---|
| 0 | dict | 13 | {'image_idx':0, 'pointcloud_num_features':4, 'velodyne_path':'training ... |
| 1 | dict | 13 | {'image_idx':3, 'pointcloud_num_features':4, 'velodyne_path':'training ... |
| 2 | dict | 13 | {'image_idx':7, 'pointcloud_num_features':4, 'velodyne_path':'training ... |
| 3 | dict | 13 | {'image_idx':9, 'pointcloud_num_features':4, 'velodyne_path':'training ... |
| 4 | dict | 13 | {'image_idx':10, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 5 | dict | 13 | {'image_idx':11, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 6 | dict | 13 | {'image_idx':12, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 7 | dict | 13 | {'image_idx':13, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 8 | dict | 13 | {'image_idx':14, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 9 | dict | 13 | {'image_idx':16, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 10 | dict | 13 | {'image_idx':17, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 11 | dict | 13 | {'image_idx':18, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 12 | dict | 13 | {'image_idx':22, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 13 | dict | 13 | {'image_idx':26, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 14 | dict | 13 | {'image_idx':29, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 15 | dict | 13 | {'image_idx':30, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 16 | dict | 13 | {'image_idx':32, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 17 | dict | 13 | {'image_idx':34, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |
| 18 | dict | 13 | {'image_idx':36, 'pointcloud_num_features':4, 'velodyne_path':'trainin ... |

*Figura 5-3:Información de cada una de las imágenes del KITTI Dataset*

| Clave/Tecla | Tipo | Tamaño | Valor |
|---|---|---|---|
| annos | dict | 13 | {'name':Numpy array, 'truncated':Numpy array, 'occluded':Numpy array, ... |
| calib/P0 | float64 | (4, 4) | [[707.0493 0. 604.0814 0. ]<br>[ 0. 707.0493 180.5066 ... |
| calib/P1 | float64 | (4, 4) | [[ 707.0493 0. 604.0814 -379.7842]<br>[ 0. 707.0493 180. ... |
| calib/P2 | float64 | (4, 4) | [[ 7.070493e+02 0.000000e+00 6.040814e+02 4.575831e+01]<br>[ 0.000000 ... |
| calib/P3 | float64 | (4, 4) | [[ 7.070493e+02 0.000000e+00 6.040814e+02 -3.341081e+02]<br>[ 0.000000 ... |
| calib/R0_rect | float64 | (4, 4) | [[ 0.9999128 0.01009263 -0.00851193 0. ]<br>[-0.01012729 0.99 ... |
| calib/Tr_imu_to_velo | float64 | (4, 4) | [[ 9.999976e-01 7.553071e-04 -2.035826e-03 -8.086759e-01]<br>[-7.854027 ... |
| calib/Tr_velo_to_cam | float64 | (4, 4) | [[ 0.00692796 -0.9999722 -0.00275783 -0.02457729]<br>[-0.00116298 0.00 ... |
| image_idx | int | 1 | 0 |
| img_path | str | 1 | training/image_2/000000.png |
| img_shape | int32 | (2,) | [ 370 1224] |
| pointcloud_num_features | int | 1 | 4 |
| velodyne_path | str | 1 | training/velodyne/000000.bin |

*Figura 5-4: Información de una imagen de KITTI Dataset*

En segundo lugar, hay que ejecutar el siguiente comando:

```
python create_data.py create_pointcloud_reduced --data_path=KITTI_DATASET_ROOT
```

Este comando sirve para obtener una nube de puntos reducida con la que entrenar la red neuronal y comprobar la eficacia de esta. La base de datos de KITTI únicamente etiqueta los datos de velodyne de lo que ven las cámaras delanteras, sin tener en cuenta los puntos obtenidos de las partes traseras y laterales del vehículo. Por lo tanto, para entrenar la red con esta base de datos es necesario obtener una nube de puntos únicamente de los datos que KITTI tiene disponible.

Esta función únicamente sirve para realizar el entrenamiento de la red y comprobar su correcto funcionamiento.

A continuación, se muestran las diferencias entre las nubes de puntos completas y reducidas.

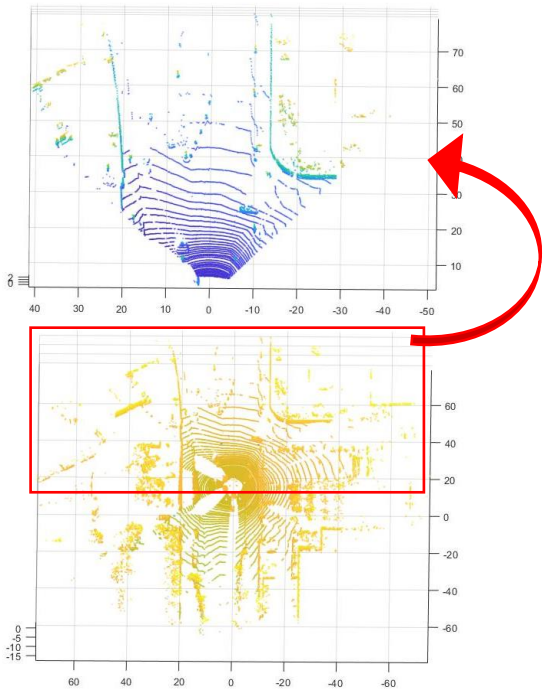


*Figura 5-5: Reducción de la nube de puntos para el entrenamiento*

Finalmente se deben obtener los datos de groundtruth en un formato compatible con la red. Para ello se debe ejecutar el siguiente comando:

```
Python créate_data.py create_groundtruth_database --
data_path=KITTI_DATASET_ROOT
```

Este comando se encarga de leer los archivos de texto con las detecciones de KITTI y convertirlas en un diccionario de Python que puede ser leído por el código de SECOND.

## 5.4 ENTRENAMIENTO DE LA RED

En la fase de entrenamiento es importante conocer los hiper-parámetros que definen el entrenamiento, la función que se utiliza para calcular el error y el tipo de dato con el que se está entrenando.

### 5.4.1 Modificación de los hiper-parámetros

Los parámetros que definen el entrenamiento y la validación de la red se pueden cambiar en el archivo de configuración. Este archivo de configuración tiene la siguiente estructura:

- Model → Define la forma del modelo utilizado.
- Train_input_reader → Define la lectura de los datos de entrenamiento.
- Train_config → Define el proceso de entrenamiento y el optimizador.
- Eval_input_reader → Define la lectura de los datos de validación.

Dentro de este archivo de configuración es necesario ajustar algunos valores.

**Parámetro anchor_box**

Los parámetros que definen a los "anchor boxes" son las dimensiones y ubicaciones aproximadas de los elementos que se desea detectar. Se pueden encontrar en la siguiente tabla:

| *Detección* | *Ancho* | *Largo* | *Alto* | *Centro en Z* |
|---|---|---|---|---|
| ***Coche*** | 1,6 m | 3,5 m | 1,5 m | -1 m |
| ***Peatón*** | 0,6 m | 0,8 m | 1,73 m | -0,6 m |
| ***Ciclista*** | 0,6 m | 1,76 m | 1,73 m | -0,6 m |

*Tabla 1: Anchor box predefinidos*

Estos valores son los recomendados por los autores del artículo de Pointpillars. Cabe mencionar que las coordenadas del centro en Z hacen referencia al centro del cubo de detección desde el eje de coordenadas del LIDAR.

Según el artículo de Pointpillars, se hicieron dos redes, la primera únicamente detectaba coches y la segunda detectaba peatones y ciclistas. Esto es adecuado para el entrenamiento pues los "anchors boxes" de los ciclistas y peatones son similares entre sí, pero muy distintos a los de los coches. Sin embargo, desde un punto de vista más práctico no es adecuado, pues requiere más procesos para realizar la detección. Por ello se han modificado los archivos de configuración para obtener la detección en una única red.

Se han realizado dos pruebas, la primera utilizando los "anchor boxes" anteriormente citados, sin embargo, el desempeño no era muy adecuado en el caso de los peatones. La segunda prueba utilizaba únicamente los "anchor boxes" de los coches, y el desempeño en este caso ha sido superior. Por lo tanto, se ha tomado la decisión de utilizar la segunda configuración.

**Parámetro xy_resolution**

También es importante la modificación del parámetro "xy_resolution" que determina cuales van a ser las dimensiones de los pilares. En este caso se ha utilizado un valor de $0.16m^2$ por pilar. En términos de precisión, cuanto menor sea este valor mayor será la precisión obtenida en las detecciones. Sin embargo, desde un punto de vista del desempeño en tiempo real, el tiempo de procesamiento que conlleva computar un mayor número de pilares es mayor, y por tanto, su uso en un sistema de tiempo real se reduce. Hay que buscar un valor intermedio que permita obtener un buen desempeño en tiempo real y su precisión sea elevada.

Se ha comprobado que con un valor de 0.16 $m^2$ se obtiene un tiempo de procesado de la red de 100 ms con la configuración hardware actual. Teniendo en cuenta que la tasa de refresco del láser de Velodyne va de 5Hz a 20 Hz, un valor de 10 Hz para el procesado es adecuado. Además, con una configuración de PC más potente se podrían conseguir tasas de refresco para el procesado de 60 Hz.

**Parámetro DATASET_ROOT**

Este parámetro no afecta directamente a la forma de entrenamiento de la red, sin embargo, es indispensable para realizar correctamente el entrenamiento.

El código de Pointpillars está preparado para realizar el entrenamiento y la validación de la red con una estructura de datos como la que se ha mencionado en el capítulo Preparación del DataSet. Esta estructura de datos es obtenida por el código y analizada dentro del mismo. Por lo tanto, como parámetro de entrada hay que introducir la ruta donde se encuentra el "KITTI_DATASET_ROOT". Este parámetro hay que modificarlo tanto en la lectura de datos de evaluación como en la lectura de datos de entrenamiento. A continuación, se indica donde modificar los parámetros de KITTI_DATASET_ROOT.

```
train_input_reader: {
  ...
  database_sampler {
    database_info_path: "/path/to/kitti_dbinfos_train.pkl"
    ...
  }
  kitti_info_path: "/path/to/kitti_infos_train.pkl"
  kitti_root_path: "KITTI_DATASET_ROOT"
}
...
eval_input_reader: {
  ...
  kitti_info_path: "/path/to/kitti_infos_val.pkl"
  kitti_root_path: "KITTI_DATASET_ROOT"
}
```

*Figura 5-6: Configuración del path*

### 5.4.2 Selección de los datos

En cualquier entrenamiento de redes neuronales es importante marcar las diferencias entre el dataset de entrenamiento, el dataset de validación y el dataset de test. En este caso, el dataset de KITTI de entrenamiento es dividido en dos, de forma que las imágenes que se utilizan en el entrenamiento no se utilizan en la validación de la red. La separación se realiza mediante un fichero en el que se indica cuáles son los datos utilizados para el entrenamiento.

Este fichero se llama "train.txt", se puede encontrar en la ruta "*second.pytorch/second/data/ImageSets/train.txt"* y en él se puede observar que los datos utilizados en el entrenamiento son distintos de los utilizados en la validación que se pueden encontrar en el fichero "val.txt".

| | |
|---|---|
| 000001 | 000000 |
| 000002 | 000003 |
| 000004 | 000007 |
| 000005 | 000009 |
| 000006 | 000010 |
| 000008 | 000011 |
| 000015 | 000012 |
| 000019 | 000013 |
| 000020 | 000014 |
| 000021 | 000016 |

*Figura 5-7: Índices de datos de entrenamiento (Izquierda) e índices de datos de validación (Derecha)*

Los datos se han separado en 3712 nubes de puntos para realizar el entrenamiento, 3769 nubes de puntos para realizar la validación y 7518 nubes de puntos para realizar el test.

### 5.4.3 Entrenamiento

El entrenamiento propiamente de la red se realiza con el siguiente comando:

```
python ./pytorch/train.py train --config_path= /path-to-config --
model_dir=/model-dir
```

Durante el proceso de entrenamiento se mostrarán las distintas mediciones que se hacen del proceso, entre ellas se encuentran las tres funciones de perdida descritas en Función de error, el número de voxels, el número de cubos detectados y el identificador de la imagen.

El entrenamiento realiza guardados cada media hora del estado actual del entrenamiento. Esto es una medida de seguridad para que, en caso de que el ordenador se apague, no se pierdan los progresos conseguidos durante horas. Además, permite realizar el entrenamiento de una forma escalonada.

El tiempo que ha tomado entrenar la red para tener un desempeño deseable ha sido de 47 horas.

### 5.4.4 Función de error

Se ha utilizado la misma función de error que la descrita en el artículo original, incluyendo los mismos valores de parámetros para las funciones de error.

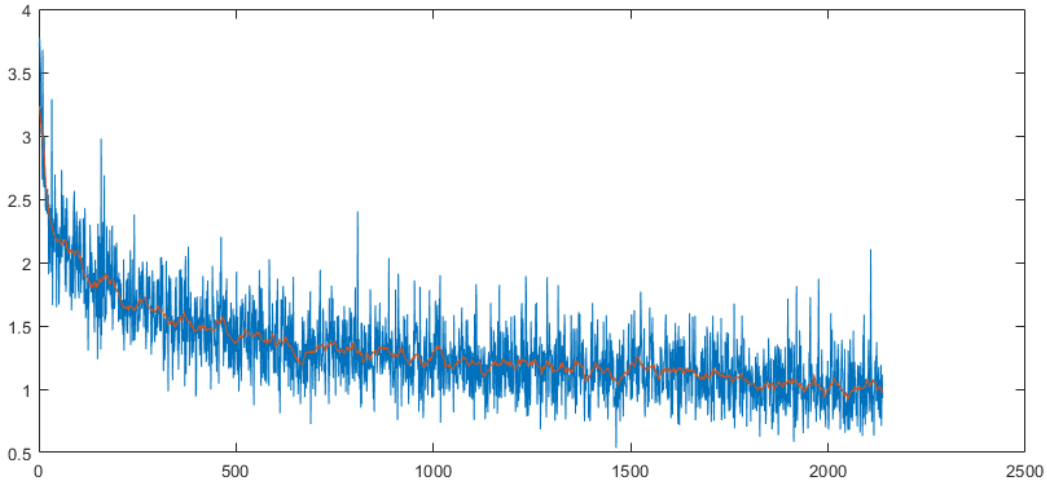



*Figura 5-8: Función de error obtenida durante el entrenamiento.*

## 5.5 DETECCIÓN

El código proporcionado por los creadores de Pointpillars no dispone de una función para obtener detecciones de una única nube de puntos. El código está preparado para obtener detecciones sobre todo el conjunto de la base de datos de KITTI y NuScenes. Sin embargo, este código es útil si se desea efectuar la validación del sistema con la base de datos de KITTI.

### 5.5.1 Evaluación

Como se ha indicado, los creadores de Pointpillars disponen de un código que sirve para realizar la evaluación de la red. Esta evaluación permite guardar los archivos en el formato de KITTI en archivos de texto independientes o los permite guardar en un archivo de tipo "Pickle" que contiene un diccionario de Python con la detección de todas y cada una de las imágenes.

### 5.5.2 Predicción aislada

Por este motivo se ha decidido crear una función que sirva para obtener detecciones de una única nube de puntos. El código se llama "Predict_kitti_single.py" y se describe a continuación:

```
1.  def predict(config_path,
2.          path_reduced_pointcloud,
3.          model_dir,
4.          result_path=None,
5.          ckpt_path=None):
```

La función necesita los siguientes parámetros

- **Config_path:** Hay que indicar la ruta donde se guarda la configuración del modelo.
- **Path_pointcloud:** Es la ruta donde se encuentra la nube de puntos que se quiere evaluar.
- **model_dir***:* Es la ruta de directorio donde se guardan los archivos que tienen los pesos de la red neuronal ya entrenada.
- **result_path:** La ruta donde se guarda el archivo de texto que contiene la salida de la red.

- **ckpt_path:** Similar a “model_dir”, sin embargo, en vez de buscar un directorio, busca un archivo concreto. En caso de dejarlo en “None”, la función busca el último modelo valido en la carpeta indicada por “model_dir”.

```
6.  model_dir = pathlib.Path(model_dir)
7.
8.  if result_path is None:
9.      result_path = model_dir / result_name
10. else:
11.     result_path = pathlib.Path(result_path)
12. config = pipeline_pb2.TrainEvalPipelineConfig()
13. with open(config_path, "r") as f:
14.     proto_str = f.read()
15.     text_format.Merge(proto_str, config)
16.
17. input_cfg = config.eval_input_reader
18. model_cfg = config.model.second
19. train_cfg = config.train_config
20. class_names = list(input_cfg.class_names)
21. center_limit_range = model_cfg.post_center_limit_range
```

En primer lugar, se realiza una lectura de los archivos de configuración. Estas configuraciones serán los parámetros de entrada para los constructores de la red y otros elementos.

```
25. voxel_generator = voxel_builder.build(model_cfg.voxel_generator)
26. bv_range = voxel_generator.point_cloud_range[[0, 1, 3, 4]]
27. box_coder = box_coder_builder.build(model_cfg.box_coder)
28. target_assigner_cfg = model_cfg.target_assigner
29. target_assigner = target_assigner_builder.build(target_assigner_cfg,
30.                                                 bv_range, box_coder)
31.
32. net = second_builder.build(model_cfg, voxel_generator, target_assigner)
33. net.cuda()
34. if train_cfg.enable_mixed_precision:
35.     net.half()
36.     net.metrics_to_float()
37.     net.convert_norm_to_float(net)
38.
39. if ckpt_path is None:
40.     torchplus.train.try_restore_latest_checkpoints(model_dir, [net])
41. else:
```

```
42.      torchplus.train.restore(ckpt_path, net)
43.
44.    if train_cfg.enable_mixed_precision:
45.      float_dtype = torch.float16
46.    else:
47.      float_dtype = torch.float32
```

Las primeras funciones hacen referencia al constructor de generación de vóxeles. Un vóxel es un elemento 3D el cual contiene información, su símil en el mundo 2D sería un pixel. Los vóxeles pueden contener información sobre reflectividad, color u otros datos. El vóxel no contiene información sobre su tamaño en el mundo real, de forma similar un píxel no contiene información sobre la distancia que representa.

La función "net= second_builder.build(model_cfg, voxel_generator, target_assigner)" construye la red con los parámetros indicados en model_cfg.

Las funciones desde la línea 39 hasta la línea 42 obtienen los pesos obtenidos durante el entrenamiento.

```
48.    net = net.eval()
49.    result_path_step = result_path / f"step_{net.get_global_step()}"
50.    result_path_step.mkdir(parents=True, exist_ok=True)
51.    t = time.time()
```

Estas funciones preparan la red para detección y configuran la ruta donde se guardará el texto con los datos de detección.

```
52.  With open(path_calib_data, 'rb') as data_info_val:
53.      Data_info = pickle.load(data_info_val),
54.    Rect = data_info['calib/R0_rect'],
55.    Trv2c = data_info['calib/Tr_velo_to_cam'],
56.    P2 = data_info['calib/P2'],
```

Las anteriores asignaciones son correspondientes a los datos de entrada, se indican las matrices de calibración descritas al inicio.

```
57.       input_dict = {
58.              'points': points,
59.              'rect': rect,
60.              'Trv2c': Trv2c,
```

```
            'P2': P2,
            'image_shape': np.array([0, 0], dtype=np.float32),
            'image_idx': None,
            'image_path': None,
            # 'pointcloud_num_features': num_point_features,
        }
        example=prep_pointcloud(input_dict,
                        root_path=None,
                        voxel_generator=voxel_generator,
                        target_assigner=target_assigner,
                        db_sampler=None,
                        max_voxels=20000,
                        class_names=['Car'],
                        remove_outside_points=False,
                        training=False,
                        create_targets=True,
                        shuffle_points=False,
                        reduce_valid_area=False,
                        remove_unknown=False,
                        gt_rotation_noise=[-np.pi / 3, np.pi / 3],
                        gt_loc_noise_std=[1.0, 1.0, 1.0],
                        global_rotation_noise=[-np.pi / 4, np.pi /
  4],
                        global_scaling_noise=[0.95, 1.05],
                        global_loc_noise_std=(0.2, 0.2, 0.2),
                        global_random_rot_range=[0.78, 2.35],
                        generate_bev=False,
                        without_reflectivity=False,
                        num_point_features=4,
                        anchor_area_threshold=1,
                        gt_points_drop=0.0,
                        gt_drop_max_keep=10,
                        remove_points_after_sample=True,
                        anchor_cache=None,
                        remove_environment=False,
                        random_crop=False,
                        reference_detections=None,
                        add_rgb_to_points=False,
                        lidar_input=False,
```

```
                        unlabeled_db_sampler=None,
                        out_size_factor=2,
                        min_gt_point_dict=None,
                        bev_only=False,
                        use_group_id=False,
                        out_dtype=np.float32)
        example["image_idx"] = image_idx
        example["image_shape"] = input_dict["image_shape"]
        print('-------------------------------------------------
  /n')

        example = merge_second_batch([example])
    example = example_convert_to_torch(example, float_dtype)
```

Posteriormente los datos se cargan en un diccionario de Python y se le pasa a la función "prep_pointcloud" junto con otros datos que definen la nube de puntos de entrada. Una vez se obtiene la nube de puntos procesada se convierte a tensores que Pytorch pueda leer.

```
        annos_result=predict_kitti_to_anno(
            net, example, class_names, center_limit_range,
            model_cfg.lidar_input, global_set)

          print(annos_result)
          _predict_kitti_to_file(net,
                        example,
                        result_path,
                        class_names,
                        center_limit_range=center_limit_range,
                        lidar_input=model_cfg.lidar_input)
          print(f"avg forward time per example: {net.avg_forward_time:.3f}")
          print(f"avg postprocess time per example: {net.avg_postprocess_time:.3f}")
```

Finalmente se evalúa la red con el tensor creado anteriormente y se obtienen las detecciones y algunas métricas como el score.

### 5.5.3 Detección utilizando ROS

Con el objetivo de realizar una implementación en un sistema real, se ha creado un código capaz de funcionar con el sistema ROS. El código genera un nodo subscriptor de ROS que se subscribe a un Topic cuyo mensaje de publicación es del tipo "*sensor_msgs/PointCloud2*".

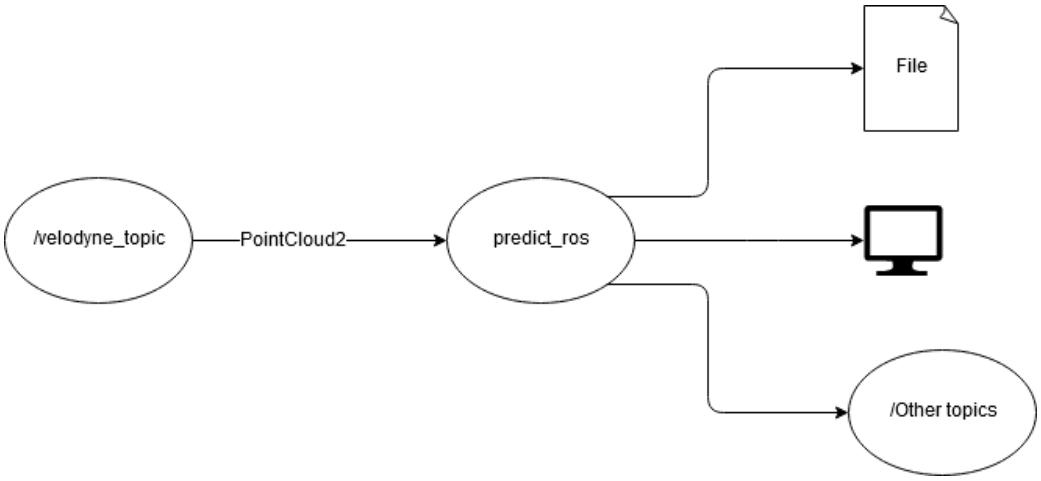



*Figura 5-9: Diagrama ROS*

En este caso, el código es prácticamente igual al mostrado en la función de predicción aislada, aunque se introducen algunas funciones para el tratamiento de la nube de puntos de entrada y la creación del nodo. Se ha utilizado la librería "ros_numpy" para extraer las coordenadas del formato de PointCloud2 en un array de Numpy. Las detecciones pueden ser obtenidas en ficheros de texto, Pickle o ser mostrados por pantalla.

# 6 RESULTADOS

## 6.1 INTRODUCCIÓN

En este capítulo se presentan los resultados obtenidos, comparándolos con el KITTI Benchmark. Los resultados obtenidos en el benchmark de KITTI son similares a los publicados en el artículo de Pointpillars. Se presentan también unos datos temporales de ejecución que muestras las bondades de esta arquitectura y sus limitaciones.

## 6.2 RESULTADOS CUALITATIVOS EN KITTI

A continuación, se muestran algunos ejemplos cualitativos de los resultados obtenidos con la red entrenada para detectar vehículos, ciclistas y peatones. Los resultados (verde) son comparados con los groundtruth de KITTI (Rojo y rosa). Se muestra también el score o fiabilidad de la detección y el IoU con el groundtruth.

Para obtener estos resultados se ha utilizado la función de evaluación proporcionada en el código de SECOND/Pointpillars que utiliza los datos del dataset destinados a la evaluación, es decir, los resultados aquí mostrados no son los mismos con los que se ha entrenado la red. Una vez obtenido todos los datos de evaluación se han visualizado utilizando el visualizador "*kittiviewer*" [6] y se han seleccionado algunos de los casos más representativos.

Se mostrarán siete imágenes de ejemplo donde se puede observar la capacidad para detectar vehículos, peatones y ciclistas bajo distintas circunstancias. Las imágenes del dataset de KITTI_dataset_object de la carpeta de training, son las siguientes:

1. Archivo 000108 → Escenario con ciclista y coche sencilla.
2. Archivo 000005 → Escenario con peatón sencillo.
3. Archivo 000076 → Escenario con peatones complicado.
4. Archivo 000006 → Escenario con coches moderado.
5. Archivo 000037 → Escenario con coches, peatones y ciclistas moderado.
6. Archivo 000062 → Escenario con coches y ciclistas moderado.
7. Archivo 000134 → Escenario con peatones, coches y ciclistas complicado.

### 6.2.1 Escenario con ciclista y coche sencillo

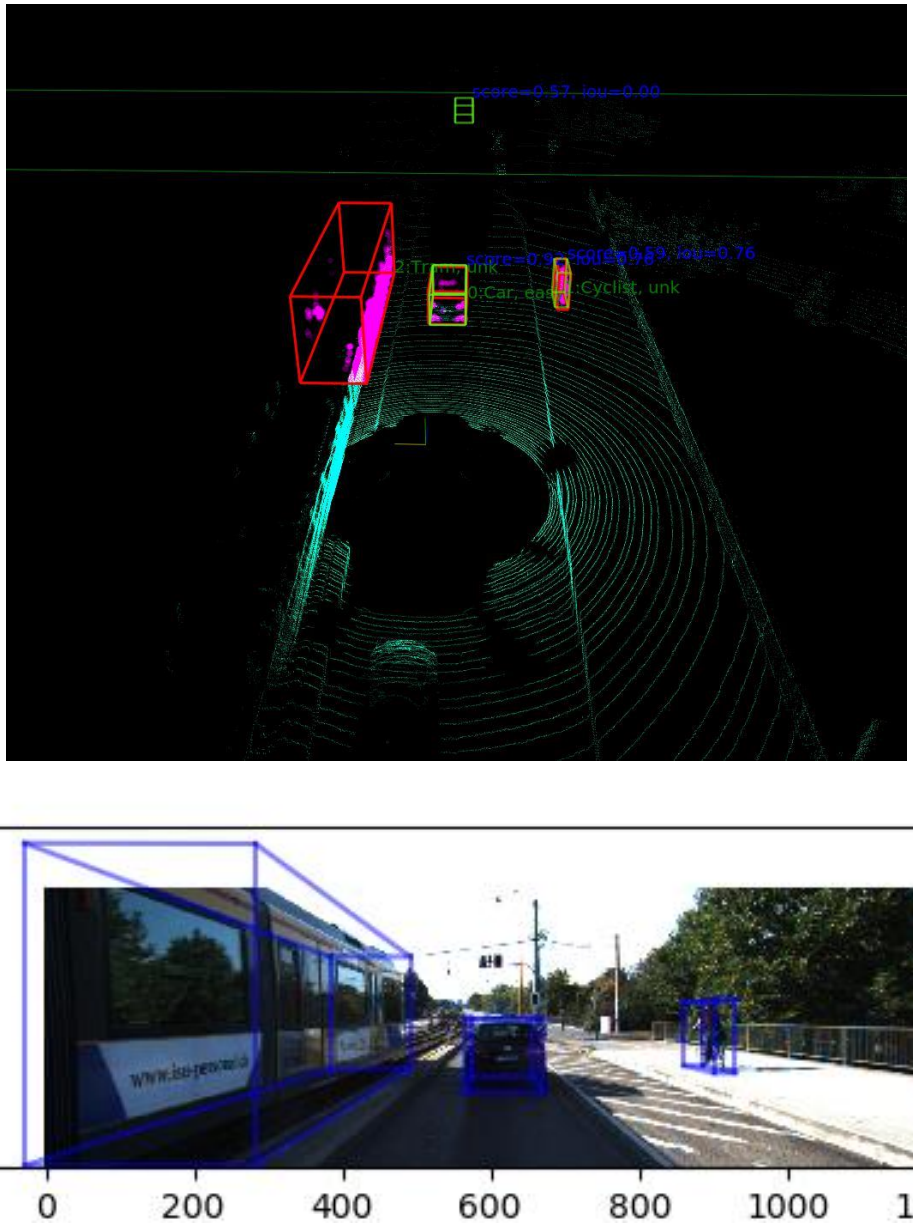



*Figura 6-1: Resultados de KITTI. Escenario con ciclista y coche sencilla.*

En esta imagen se puede observar como la red detecta un vehículo delante y un ciclista en la misma escena. En cuanto al tren indicado por KITTI, la red no ha sido entrenada para detectar este tipo de elementos, por lo tanto, es normal que no lo haya detectado. También se detecta al fondo otro vehículo, sin embargo, el score de este no es muy elevado dado el número de puntos tan reducida que se tiene.

Se puede apreciar como el IoU de la detección sobre el groundtruth proporcionado por KITTI es aproximadamente 0,78 en ambos casos. Con este valor, el proceso de evaluación de KITTI da por válida la detección tanto en vehículos como en ciclistas.

### 6.2.2 Escenario con peatón sencillo

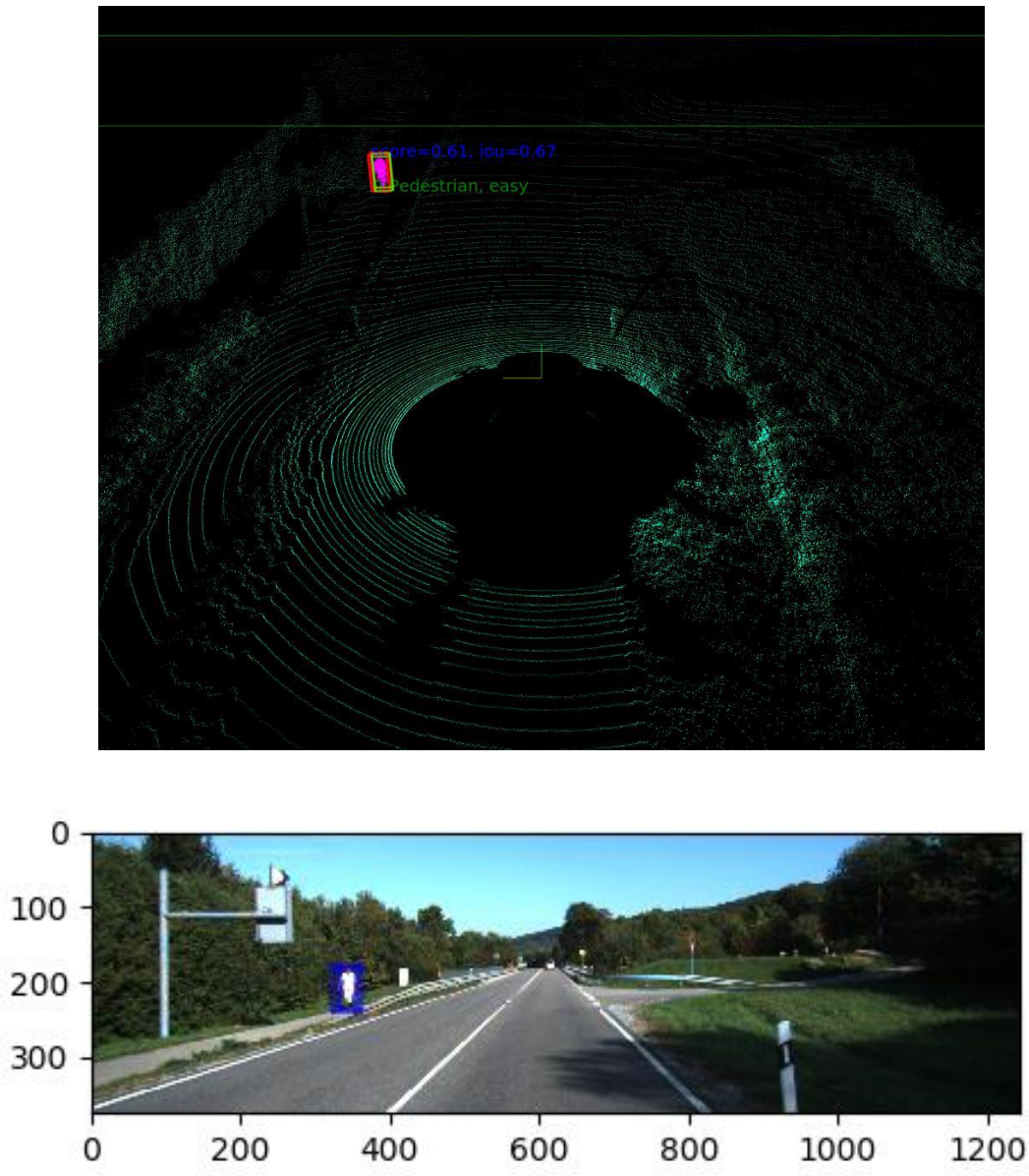
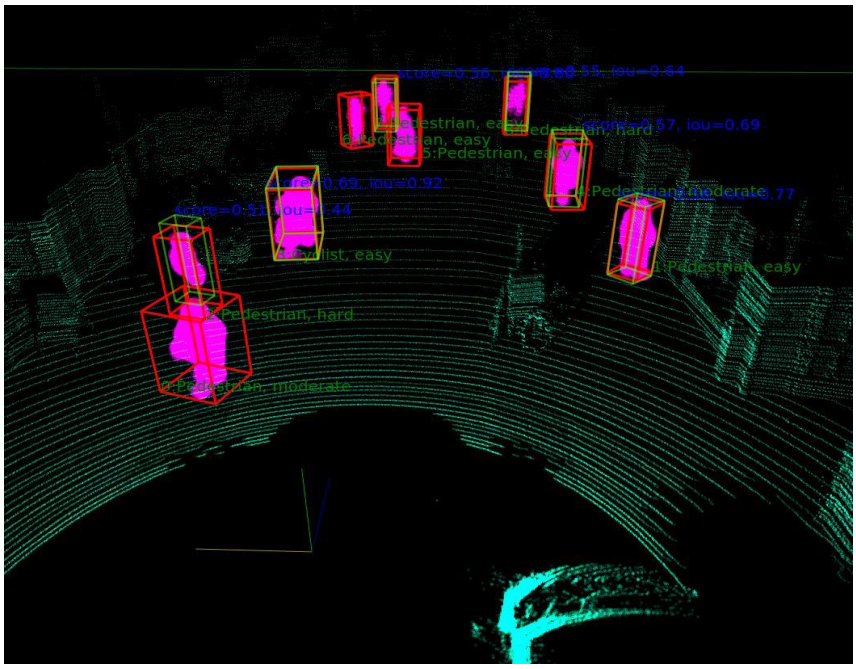



*Figura 6-2: Resultados de KITTI. Escenario con peatón sencillo.*

En esta imagen se puede observar como la red es capaz de detectar peatones. El score obtenido no es muy elevado. Esto puede ser debido a que el entrenamiento de la red se ha realizado junto con el de vehículos y ciclistas.

El IoU es de 0.61, de nuevo este es un valor que según el procedimiento de evaluación de KITTI es suficiente para dar la detección por válida.

### 6.2.3 Escenario con peatones complicado



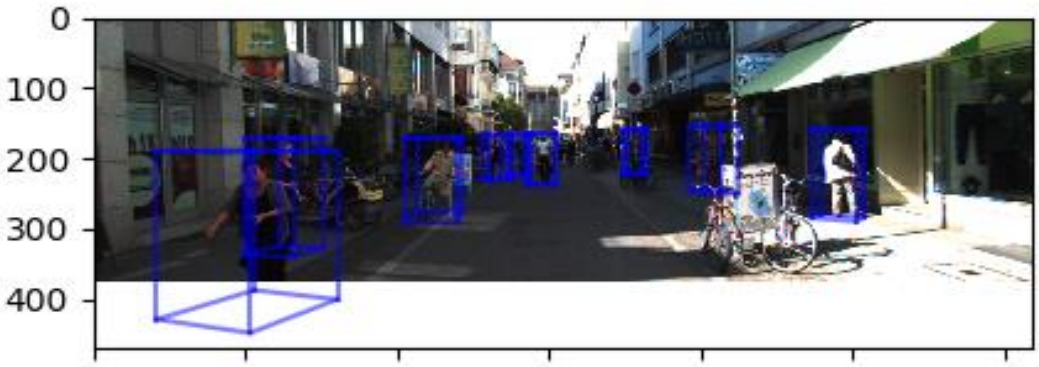



*Figura 6-3: Resultados de KITTI. Escenario con peatones complicado.*

De nuevo se muestra en la imagen como la red es capaz de detectar varios peatones juntos. Sin embargo, en este caso se puede apreciar como algunos peatones no son detectados correctamente. Concretamente no se detecta el primer peatón a la izquierda y alguno del final. Como se ha indicado anteriormente, la detección de peatones es sin duda la más complicada de todas. Esto es debido a la alta precisión que hace falta en la medida, las distintas anatomías de los cuerpos y las poses que pueden adoptar.

### 6.2.4 Escenario con coches moderado

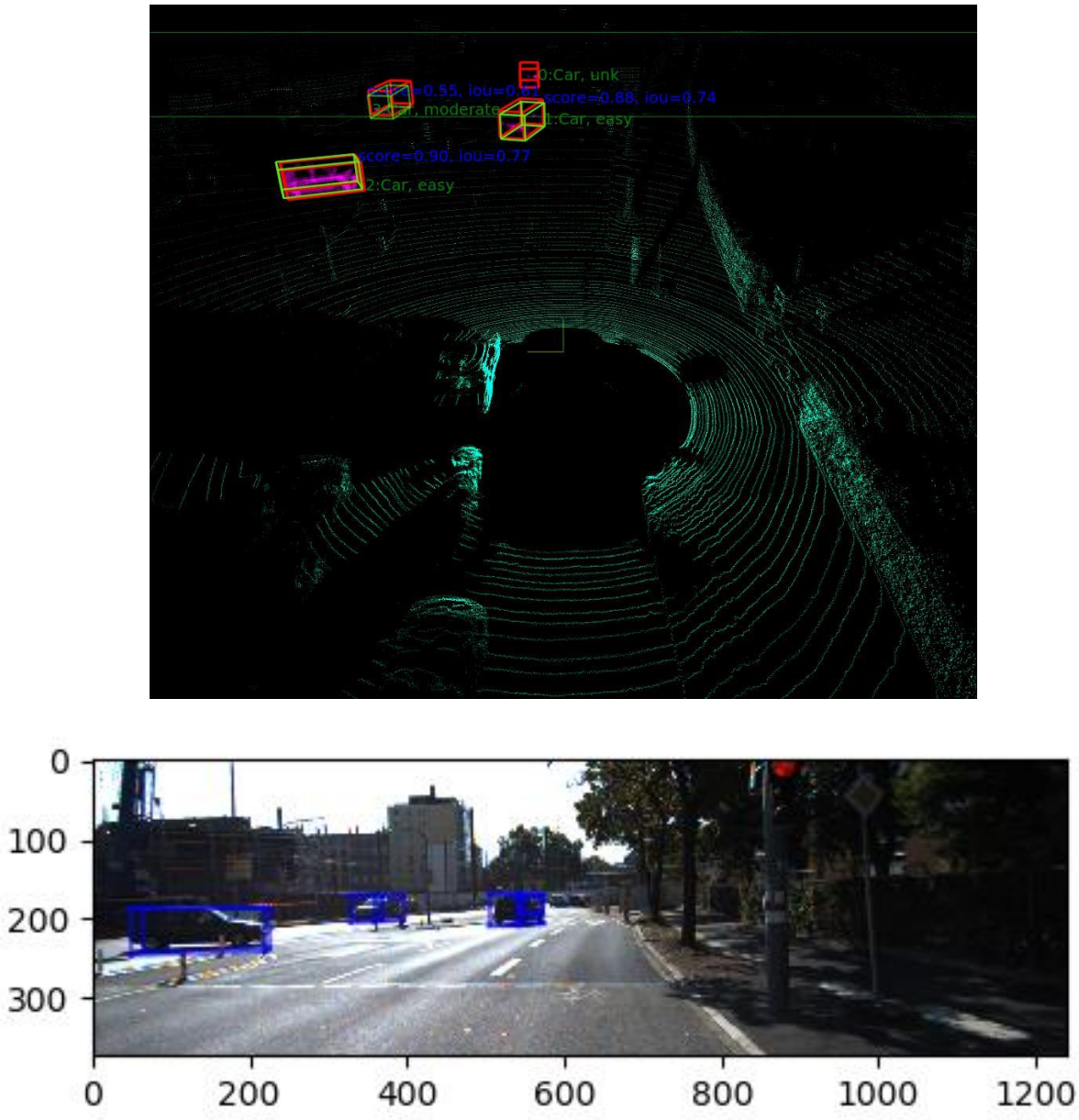
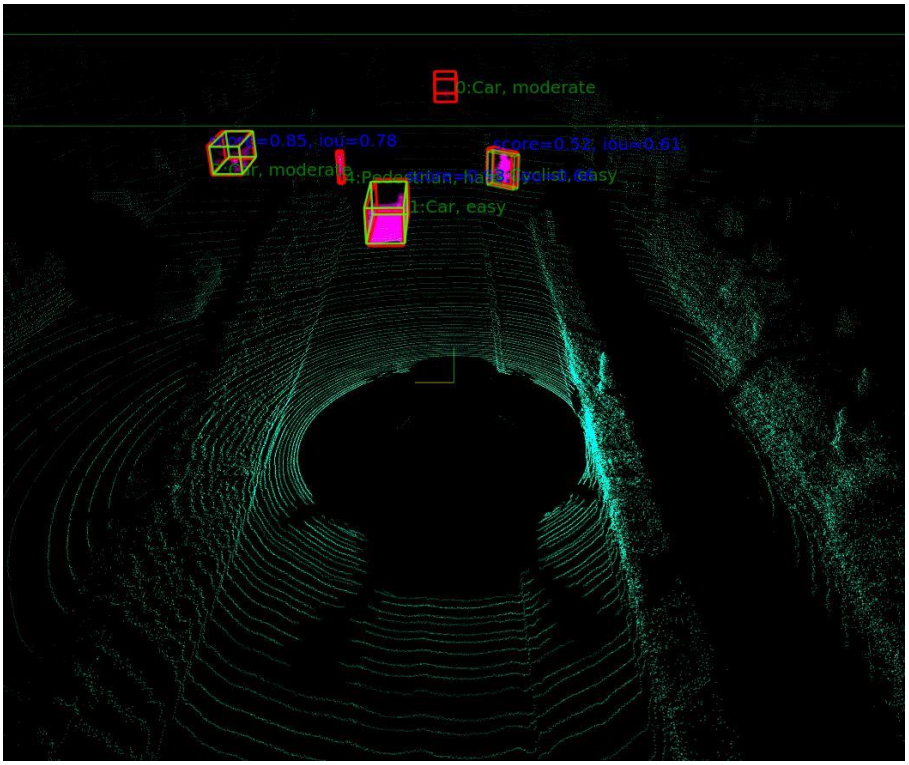



*Figura 6-4: Resultados de KITTI. Escenario con coches moderado.*

En este caso únicamente se muestra la detección de vehículos. Se observa que los scores en estos casos son bastante elevados. Esto se debe a que la mayor parte de los datos de KITTI son vehículos, y por lo tanto, el entrenamiento y la detección de este tipo de dato es notablemente mejor.

### 6.2.5 Escenario con coches, peatones y ciclistas moderado



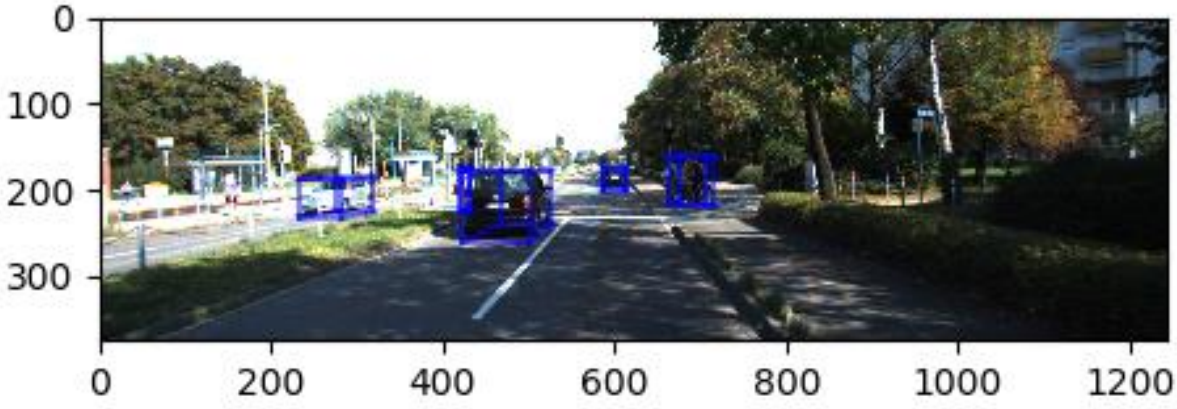




*Figura 6-5: Resultados de KITTI. Escenario con coches, peatones y ciclistas moderado.*

En este caso se muestra como la red detecta los vehículos de la calzada y al ciclista que se encuentra parado en la acera. No es capaz de detectar el vehículo del fondo debido a la falta de información, ni al peatón que se encuentra parado en el semáforo.

### 6.2.6 Escenario con coches y ciclistas moderado

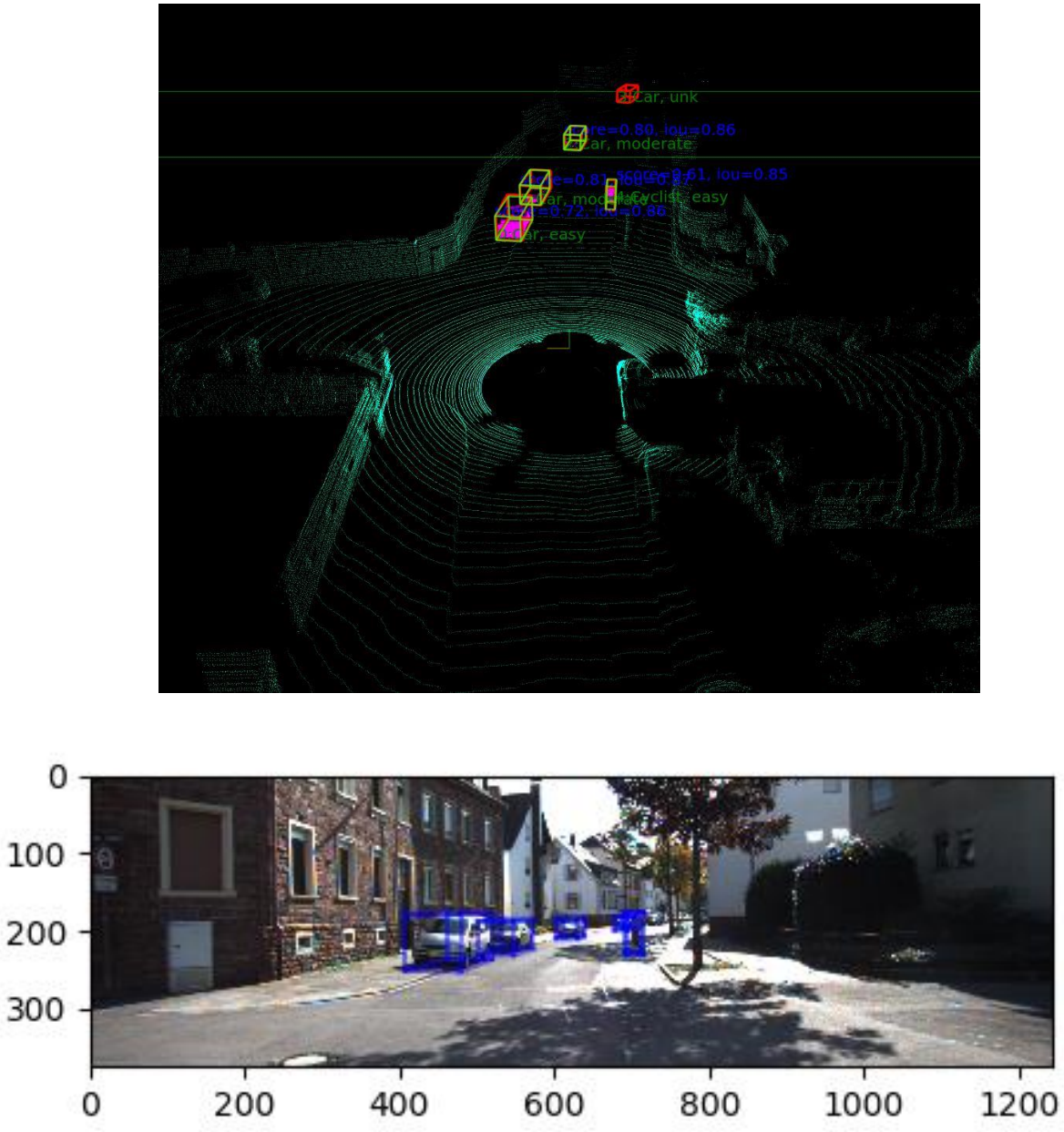



*Figura 6-6: Resultados de KITTI. Escenario con coches y ciclistas moderado.*

En esta imagen se vuelve a mostrar detección de vehículos, esta vez estacionados fuera de la calzada, y la detección de un ciclista de espaldas. Se puede apreciar cómo se detectan vehículos con scores elevados incluso con dificultad de detección moderada.

### 6.2.7 Escenario con peatones, coches y ciclistas complicado

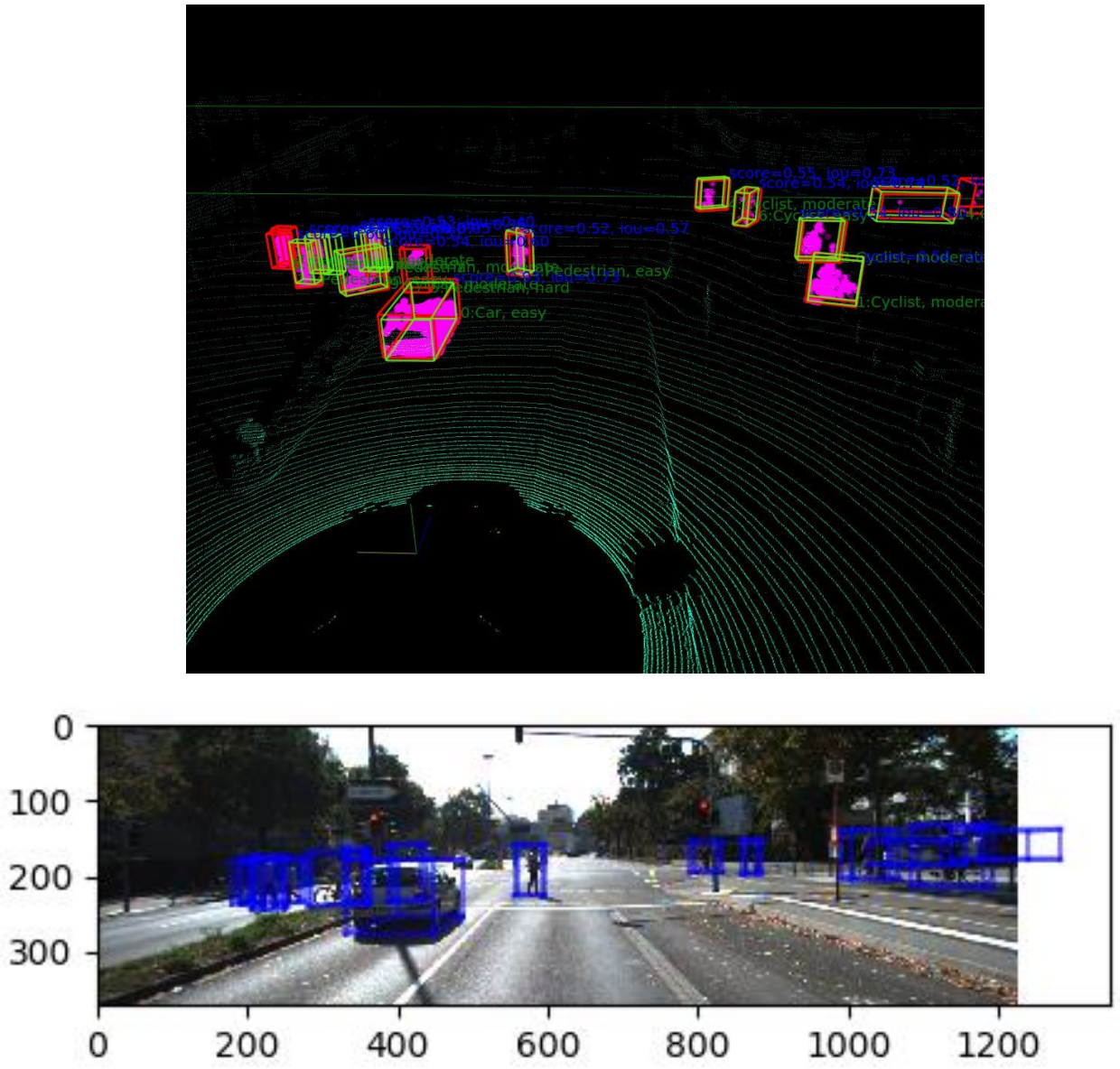



*Figura 6-7: Resultados de KITTI. Escenario con peatones, coches y ciclistas complicado.*

En este caso se muestra una escena sumamente compleja donde intervienen vehículos, peatones y ciclistas. Se puede observar como el único problema de detección se encuentra en la zona donde se aglomeran algunos peatones y ciclistas. Esto es debido, como anteriormente se ha dicho, a la falta de datos de entrenamiento para estas clases. Sin embargo, en este caso donde se encuentran muchos peatones juntos también influye la precisión con la que se procesa la red. Cabe mencionar que actualmente los pilares evaluados son de 0.16 $m^2$. Si este valor fuera inferior, esta imagen tendría mejores detecciones, sin embargo, el tiempo de procesamiento subiría.

## 6.3 RESULTADOS DE KITTI

En esta sección se mostrarán los resultados obtenidos en el proceso de evaluación proporcionado por KITTI. Este procedimiento de evaluación, desarrollado por KITTI, proporciona un conjunto de gráficas donde se muestra la capacidad de detección de las clases "car", "pedestrian" y "cyclist" sobre la base de datos de KITTI.

En el procedimiento de evaluación se diferencian tres niveles de dificultad easy, moderate y hard. Estos niveles tienen relación con la dificultad de detección según el siguiente criterio:

- **Easy**: Altura mínima de la caja 40 px, sin oclusión y truncamiento máxima del 15%
- **Moderate**: Altura mínima de la caja 25 px, parcialmente ocluido y truncamiento máximo 30%.
- **Hard**: Altura mínima de la caja 25 px, complicado de ver por oclusión y truncamiento máximo 50%.

Se mostrarán las tres gráficas para cada una de las clases entrenadas, vehículos, peatones y ciclistas. Las gráficas muestran el parámetro denominado "Recall" y el parámetro "precisión". Estas gráficas son generadas por un código proporcionado por KITTI, donde se evalúa la detección de las clases "pedestrian", "cyclist" y "car" debido a que son las únicas que se considera que tienen suficientes datos como para realizar una validación.

Para que una detección cuente como detección positiva se deben cumplir que las bounding box se ajusten lo máximo posible a las proporcionadas por el groundtruth. Para ello se utiliza el algoritmo IoU que indica en tanto por uno la similitud entre dos bounding box.

Las tres gráficas de cada clase serán las siguientes:

- **Detection:** Indicará la precisión obtenida en la detección de la clase sobre la imagen 2D. Esto quiere decir que se proyectará la bounding box obtenida sobre la imagen 2D y se calculará su IoU.
- **3D detection:** Indicará la precisión obtenida en la detección 3D. Es decir, se calculará el IoU sobre las bounding box 3D.
- **BEV:** Se proyecta la bounding box para mostrar una visualización desde arriba y calcular el IoU sobre esa proyección.

### 6.3.1 Resultados “Detection”

Las siguientes gráficas han sido obtenidas con todas las imágenes del dataset de entrenamiento de KITTI (KITTI_dataset_object), en total 7481 imágenes, entre las cuales se encuentran 3712 imágenes con las que se ha realizado el entrenamiento y 3769 imágenes que se han

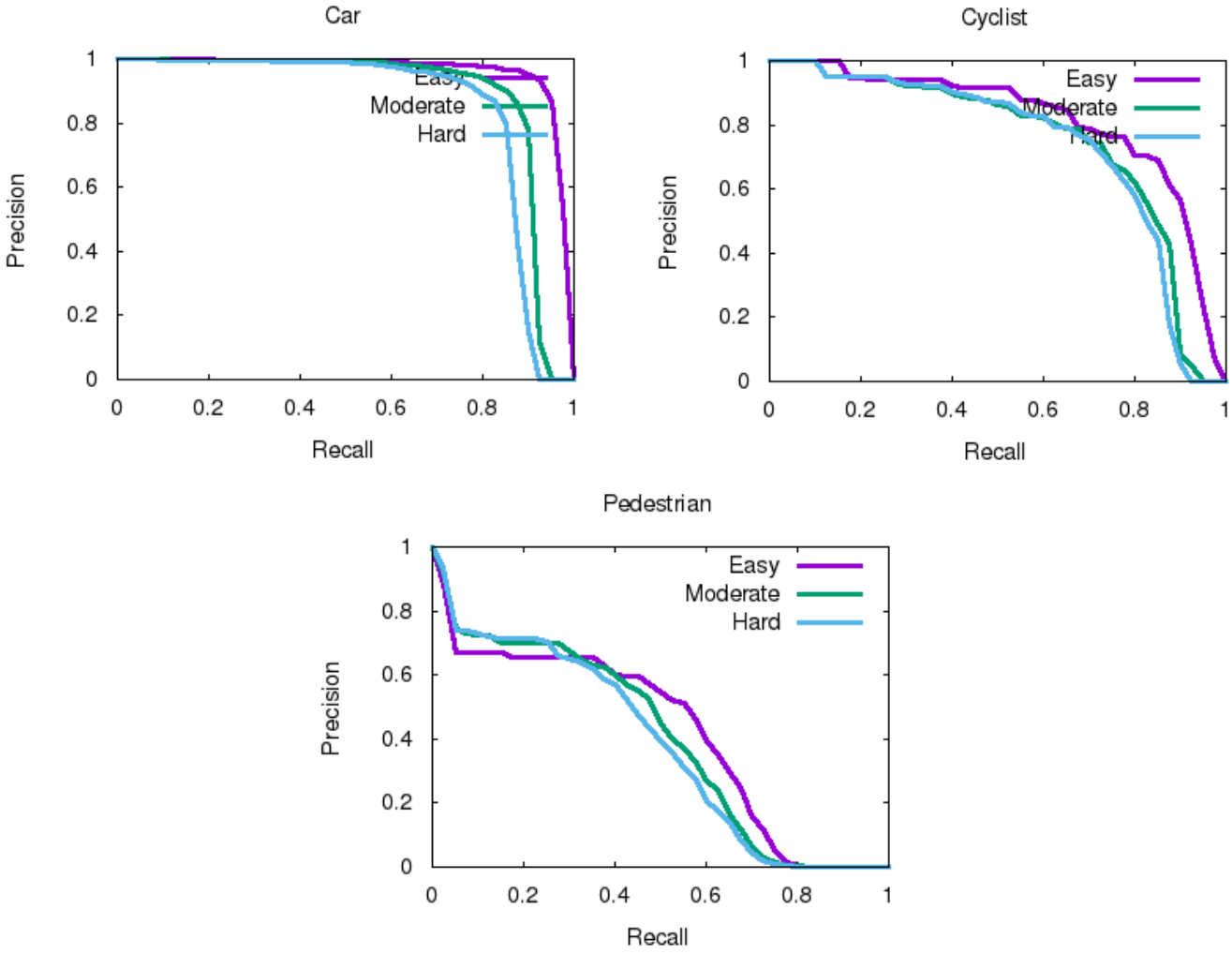



*Figura 6-8: Resultados de detection*

utilizado en evaluación.

Se puede observar que en la clase “car”, donde es mayor el número de datos de entrenamiento, se ha obtenido una precisión en la detección muy elevada. El desempeño de la red disminuye en el caso de los ciclistas, aunque sigue estando en valores próximos al estado del arte. Donde realmente no ha tenido un buen desempeño la red ha sido en la detección de peatones. Esto se puede deber a utilizar la misma red para detectar las tres clases, a la falta de datos de entrenamiento y a la precisión de los pilares que se ha establecido en 0,16$m^2$.

### 6.3.2 Resultados “3D Detection”

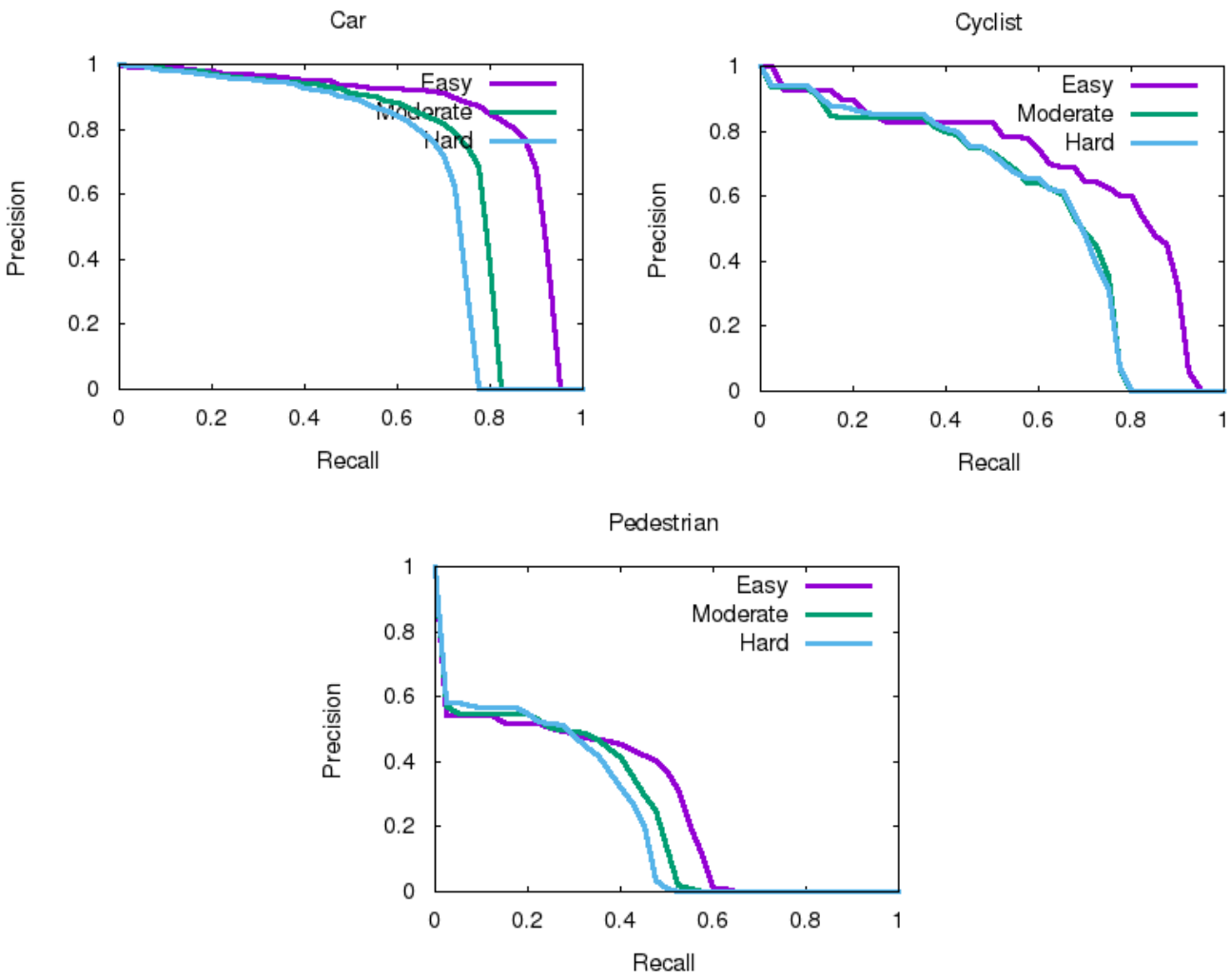



*Figura 6-9: Resultados 3D detection*

De nuevo, la clase “car” obtiene unos resultados elevados, la clase “cyclist” se mantiene próximo al estado del arte y la clase pedestrian obtiene unos resultados poco adecuados. En este caso el procedimiento de evaluación es más agresivo ya que se trata de solapar un cubo tridimensional, al contrario que en el caso anterior que el objetivo era obtener un solapamiento de cuadrados.

### 6.3.3 Resultados “BEV”

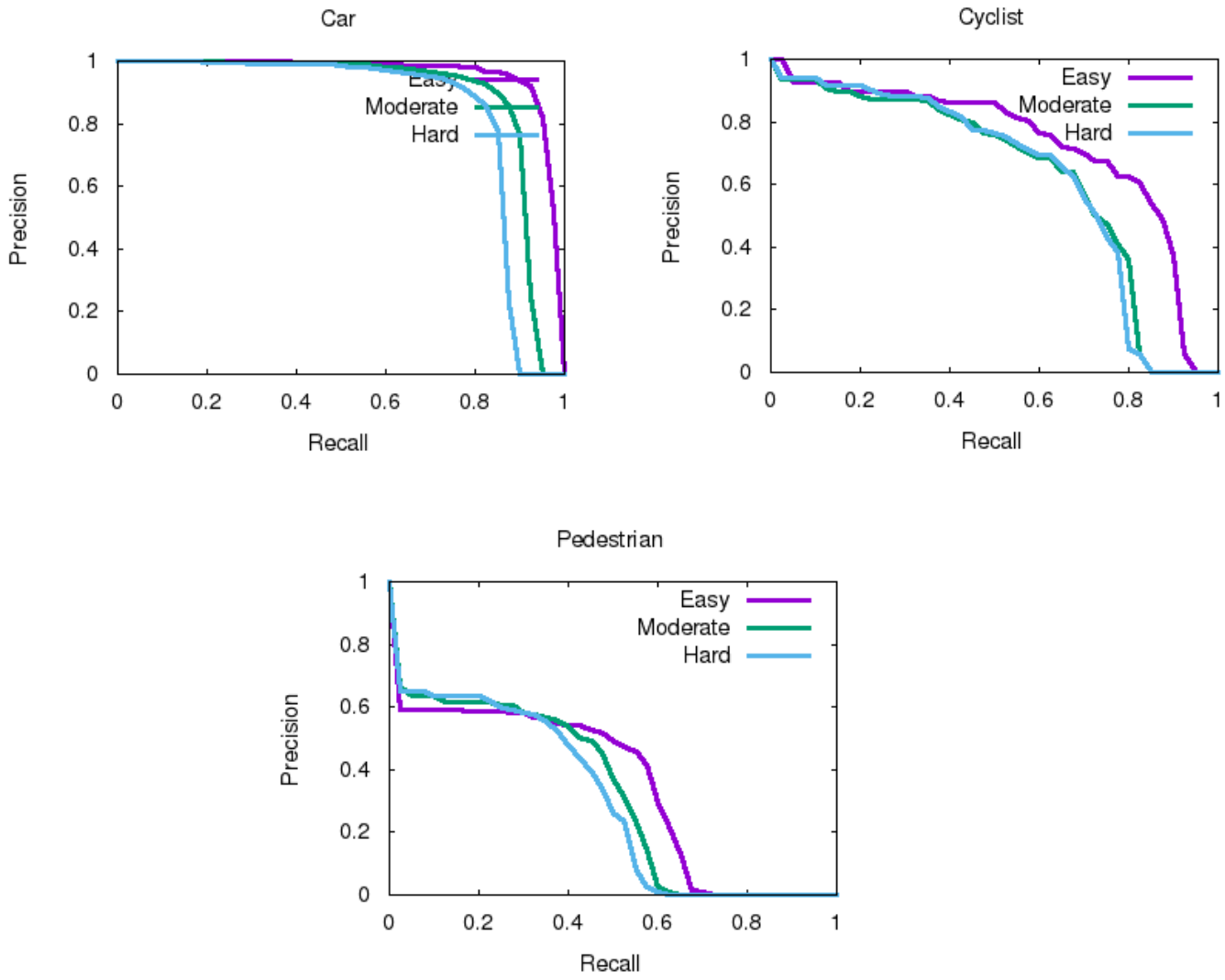



*Figura 6-10: Resultados BEV*

Los resultados de BEV en los casos de “car” y “cyclist” se encuentran, al igual que en los casos anteriores, en el estado del arte. En el caso de la clase “pedestrian”, se encuentra por debajo del estado del arte actual.

# 7 MANUAL DE USUARIO

## 7.1 INTRODUCCIÓN

En este capítulo se detalla cómo se utiliza la aplicación desarrollada. Se describirá el procedimiento para ejecutar el entrenamiento de la red, la detección y como mostrar los resultados. Toda la aplicación ha sido empaquetada un contenedor de Docker para facilitar su portabilidad.

## 7.2 MANUAL DE USO

### 7.2.1 Comprobaciones previas

Lo primero es asegurarse de que el ordenador tiene instalado CUDA TOOLKIT, los drivers de NVIDIA, Docker y Nvidia-docker. Para realizar estas comprobaciones se deben ejecutar los siguientes comandos:

```
Nvidia-smi
```

Deberá mostrar lo siguiente:

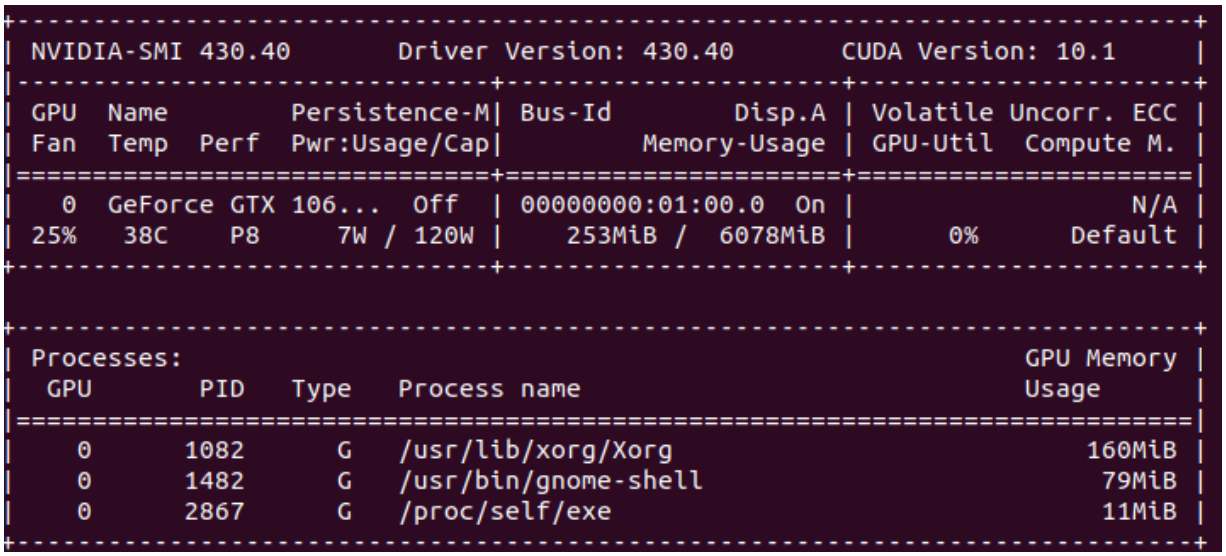



*Figura 7-1: Comprobación Driver NVIDIA*

Aquí se muestra la versión del Driver de NVIDIA y la versión de CUDA utilizada.

Para comprobar la instalación de Docker se puede ejecutar el siguiente comando:

```
Docker images
```

```
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
pointpillars            final               4a016da0e1c9        22 hours ago        22.4GB
pointpillars            version1            cd07d1a02dcf        42 hours ago        8.6GB
nvidia/cuda             10.1-base           a5f5d3b655ca        2 weeks ago         106MB
scrin/second-pytorch    latest              6187b5d21625        9 months ago        5.52GB
```

*Figura 7-2: Comprobación Docker*

Si todo está correcto debería mostrar las imágenes de Docker importadas o una lista vacía en caso de no tener ninguna imagen importada.

### 7.2.2 Estructura y ejecución del Docker

Una vez comprobado que se tienen los paquetes instalados necesarios se debe importar la imagen de Docker. Para ello se debe ejecutar lo siguiente:

```
cat /home/pointpillars.tar | docker import – pointpillars
```

Una vez importada la imagen se debe obtener una terminal del Docker. Para ello hay que ejecutar el siguiente comando:

```
Nvidia-docker run -it –rm -v /:/host -v /[location
KITTI_DATASET_ROOT]:/root/data –ipc=host pointpillars:final
```

La consola arrancará automáticamente con la consola “Fish”, para salir de ella basta con escribir el comando:

```
Bash
```

La estructura del fichero “root”, que es donde se encuentra todo el material necesario para Pointpillars, tiene la siguiente estructura principal:

- Second.pytorch → Se encuentra todo el código de SECOND.
    - Second → Contiene las funciones de creación de datos necesarios para entrenamiento y validación.
        - Pytorch → Contiene las funciones creadas para entrenamiento, validación y predicción.
- SparseConvnet → Programa de Facebook Research necesario para que funcione correctamente.

- Data → Ubicación de los datos necesarios para realizar el entrenamiento y la validación.
- Results → Carpeta destinada al guardado de resultados.

### 7.2.3 Entrenamiento de la red

Para realizar cualquier operación con la red es necesario activar el entorno de Conda de pointpillars. Para ello se debe ejecutar el siguiente comando:

```
Conda actívate pointpillars
```

Lo siguiente sería comprobar que el dataset que se va a utilizar tiene la misma estructura que el mencionado en el capítulo *5.3Preparación del DataSet*.

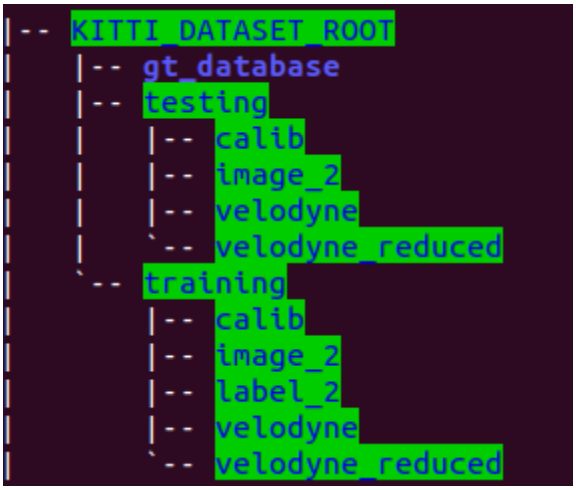



*Figura 7-3: Árbol del Dataset*

Lo siguiente sería obtener los datos de las imágenes de KITTI. Para ello se debe ejecutar el siguiente comando:

```
Python créate_data.py create_kitti_info_file --
data_path=/root/data/KITTI_DATASET_ROOT
```

Deberá mostrar el siguiente mensaje "Kitti info train file is saved to /root/data/KITTI_DATASET_ROOT/kitti_infos_train.pkl"

Lo siguiente será crear la nube de puntos reducida. Para ello habrá que ejecutar el siguiente comando:

```
Python créate_data.py create_reduced_point_cloud --
data_path=/root/data/KITTI_DATASET_ROOT
```

Esto tardará varios minutos. Durante el proceso deberá mostrar varias barras de progreso como las siguientes:

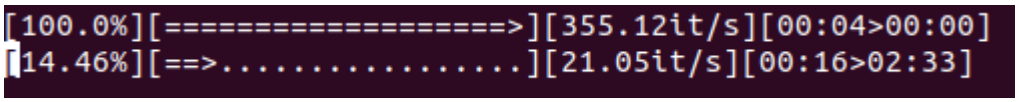



*Figura 7-4: Terminal Create reduced pointcloud*

A continuación, se deben generar los archivos de groundtruth del entrenamiento. Para ello hay que ejecutar el siguiente comando:

```
Python créate_data.py create_groundtruth_database --
data_path=/root/data/KITTI_DATASET_ROOT
```

Cuando termine deberá mostrar lo siguiente:

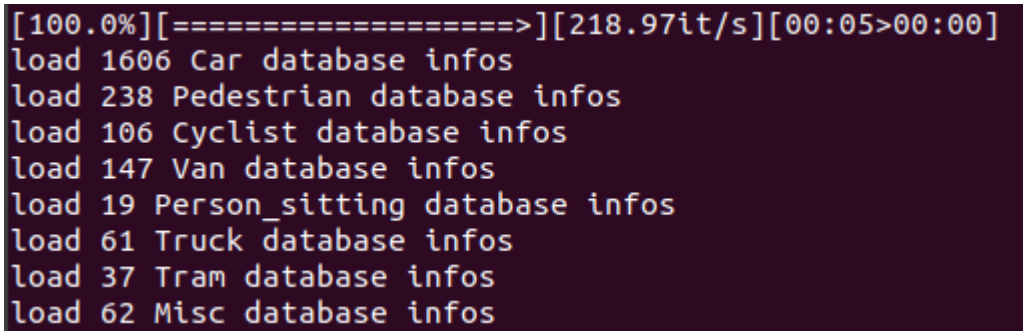



*Figura 7-5: Terminal create KITTI Info*

A continuación, hay que modificar el archivo de configuración de la red para indicar donde se encuentran las rutas de los archivos generados. Para ello hay que modificar los siguientes parámetros del archivo de configuración ubicado en */root/second.pytorch/second/configs/pointpillars/complete.proto*, habrá que modificar los siguientes campos:

```
train_input_reader: {
  ...
  database_sampler {
    database_info_path: "/path/to/kitti_dbinfos_train.pkl"
    ...
  }
  kitti_info_path: "/path/to/kitti_infos_train.pkl"
  kitti_root_path: "KITTI_DATASET_ROOT"
}
...
eval_input_reader: {
  ...
  kitti_info_path: "/path/to/kitti_infos_val.pkl"
  kitti_root_path: "KITTI_DATASET_ROOT"
}
```

*Figura 7-6: Parámetros de evaluación y entrenamiento*

También se pueden modificar otros parámetros si se desea como el número de epochs o los anchor boxes.

Una vez modificado el archivo de configuración, se deberá ejecutar el siguiente comando para comenzar el entrenamiento:

```
python ./pytorch/train.py train --
config_path=./configs/pointpillars/complete.proto --
model_dir=/root/model
```

Es importante saber que para que el modelo entrene desde cero la carpeta /root/model debe estar completamente vacía.

Durante el entrenamiento deberá aparecer lo siguiente:

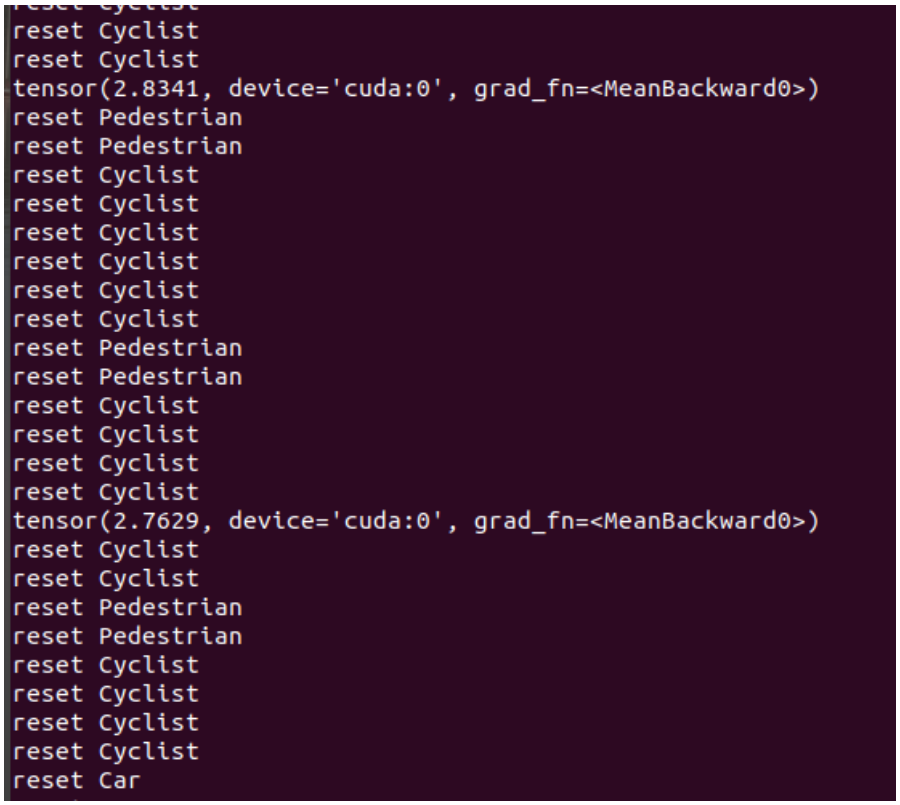



*Figura 7-7: Terminal entrenamiento KITTI*

Se muestra el tensor de la función de error. También se muestra el reinicio de las métricas para cada una de las clases. La clase que más se reinicia es la de “Cyclist” debido a que el número de datos para entrenamiento de esta clase es el más reducido de los tres. Por el contrario, la clase coche es la que menos se reinicia.

Durante la fase de entrenamiento también se realizan validaciones para comprobar como progresa el entrenamiento. Las evaluaciones tienen el siguiente aspecto:

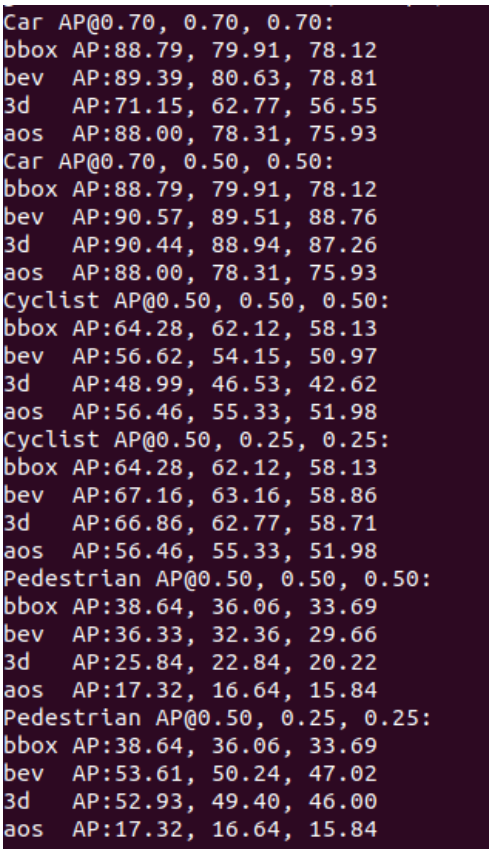



*Figura 7-8: Evaluación KITTI durante entrenamiento*

Aquí se muestran los valores de desempeño conseguido en función de los groundtruth proporcionados por KITTI. El valor obtenido es en porcentaje sobre el groundtruth de KITTI. Para que una predicción se considere válida debe tener por lo menos la siguiente correspondencia de IoU.

| ***Elemento/Dificultad*** | **Easy** | **Moderate** | **Hard** |
|---|---|---|---|
| *Coche* | 0.7 | 0.7 | 0.7 |
| *Ciclista* | 0.5 | 0.5 | 0.5 |
| *Peatón* | 0.5 | 0.25 | 0.25 |

*Tabla 2: IoU mínimo*

Los vehículos dado su tamaño y forma similar son más fáciles de detectar, por esto tienen un mayor IoU. Los peatones son los elementos más complicados de detectar dado que existen tamaños muy distintos y poses muy distintas también

### 7.2.4 Uso de la red para evaluación

Para realizar una evaluación con el dataset de KITTI creado anteriormente hay que ejecutar el siguiente comando:

```
python ./pytorch/train.py evaluate --
config_path=./configs/pointpillars/complete.proto --
model_dir=/root/model –pickle_result=true
```

Esta función generará en pantalla una tabla similar a la mostrada durante el entrenamiento. Además, generará también un archivo denominado “results.pkl”. Este archivo es un diccionario de Python que contiene todas las detecciones realizadas en cada una de las imágenes.

Este archivo es útil si se desea utilizar el visualizador ”Kittiviewer” que permitirá cargar las imágenes, los groundtruth de las mismas y las detecciones realizadas.

### 7.2.5 Uso de la red para ejemplo único

En caso de que se ejecute la red para la evaluación de una única nube de puntos, será necesario disponer de la información de calibración de la nube de puntos obtenida. Estas matrices son las matrices de R0 que es la matriz de rotación de imagen rectificada a no rectificada, P2 que es la matriz de calibración de la cámara izquierda y Tr_velo2cam que es la matriz de transformación de coordenadas de velodyne a cámara. Estos parámetros no son críticos para el desempeño de la red dado que las detecciones no dependen de imágenes 2D. Sin embargo, si se introducen, devolverá las detecciones referenciadas al eje de coordenadas de la cámara. Si se desea se puede dejar la matriz unidad para obtener detecciones sobre el eje de coordenadas del velodyne. En caso de que se quiera evaluar una imagen de KITTI es necesario introducir estas matrices ya que las predicciones de KITTI se realizan sobre el eje de coordenadas de las cámaras.

Para modificar estos valores se puede acceder al archivo que genera los datos de calibración de una única imagen. La función se ejecuta con el siguiente comando:

```
Vim /root/second.pytorch/second/create_calib_single.py
```

Se deberá mostrar lo siguiente:

```
import pickle
import numpy as np


R0 = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
Trv2c = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
P2 = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])

image_info={'calib/R0_rect': R0, 'calib/Tr_velo_to_cam': Trv2c, 'calib/P2': P2}
filename = 'CALIB_INFO.pkl'
with open(filename, 'wb') as f:
    pickle.dump(image_info, f)
```

*Figura 7-9: Publicación de matrices de calibración*

Después de modificar las matrices de calibración, se debe ejecutar el siguiente comando para obtener las matrices de calibración en un formato de Python.

```
Python créate_calib_single.py
```

Esto generará un archivo denominado “CALIB_INFO.PKL”.

Una vez generado el archivo se puede llamar a la función de detección de una única nube de puntos. El comando es el siguiente:

```
Python pytorch/predict_kitti_single.py
--path_calib_data=/root/second.pytorch/second/CALIB_INFO.pkl
--model_dir=[path/to/model]
--config_path=/root/second.pytorch/second/configs/pointpillars/complete.proto
--path_pointcloud=[/path/to/pointcloud]
--result_path=/root/results
```

El programa primero debería mostrar las matrices de calibración introducidas:

```
Restoring parameters from /host/home/sergio/TFM/model_car_pedestrian_cyclist/voxelnet-260707.tckpt
rect =
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
Trv2c =
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
P2 =
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

*Figura 7-10: Terminal, matrices de calibración*

A continuación, debería mostrar los tensores de detección con información sobre las clases detectadas, sus scores, dimensiones y localizaciones. Además, generará los archivos “000001.txt” y “result.pkl” que contiene la información de la predicción, el primero en un archivo de texto compatible con KITTI y el segundo en un archivo de Python.

### 7.2.6 Uso de la red con ROS

Para ejecutar la red como un nodo subscriptor de ROS es necesario modificar el código "*predict_ros.py*" para cambiar la ubicación del modelo, los datos de calibración y donde se van a guardar los resultados.

```
def predict(pointcloud):

    model_dir = '/host/home/sergio/TFM/model_car_pedestrian_cyclist'
    config_path = '/host/home/sergio/TFM/PointPillars/second.pytorch/second/configs/pointpillars/complete.proto'
    path_calib_data = '/root/second.pytorch/second/CALIB_INFO.pkl'
    result_path = '/root/results'
```

*Figura 7-11:Configuración del nodo de ROS - Parámetros de directorios*

También será necesario cambiar el Topic al que se debe subscribir.

```
rospy.init_node('listener', anonymous=True)
rospy.Subscriber("/kitti_player/hdl64e",PointCloud2,predict)
```

*Figura 7-12:Configuración del nodo de ROS - Nombre del Topic*

Finalmente también se deberá cambiar la forma en la que se desean obtener las detecciones. Se puede elegir entre:

- Predict_kitti_to_anno: Obtiene un diccionario de Python con las detecciones, únicamente es accesible desde el propio programa.
- Predict_kitti_to_file: Obtiene archivos de texto con el formato de KITTI de cada una de las imágenes.
- Pickle.dump: Esta línea se puede comentar o dejar sin comentar. Si se deja sin comentar se guardarán todas las predicciones en un archivo de diccionario de Python.

```
    annos_result = predict_kitti_to_anno(
        net, example, class_names, center_limit_range,
        model_cfg.lidar_input, global_set)

    print(annos_result)

    _predict_kitti_to_file(net,
                           example,
                           result_path,
                           class_names,
                           center_limit_range=center_limit_range,
                           lidar_input=model_cfg.lidar_input)
    with open(result_path / "result.pkl", 'wb') as f:
            pickle.dump(annos_result, f)
```

*Figura 7-13: Configuración del nodo de ROS - Obtención de los resultados.*

Para arrancar el nodo basta con ejecutar el siguiente comando:

```
python predict_ros.py
```

La información de calibración la sacará, al igual que el caso anterior, del fichero "CALIB_INFO.PKL".

# 8 CONCLUSIONES Y TRABAJOS FUTUROS

En este trabajo se ha realizado la implementación de una red neuronal desde su entrenamiento hasta su aplicación funcionando en un sistema con ROS. Durante el proceso se ha utilizado solamente los datos proporcionados por el láser LiDAR 3D de la base de datos de KITTI, utilizando las imágenes únicamente para realizar la visualización. Se ha desarrollado una herramienta portable que contiene el código necesario para ejecutar la red, realizar el entrenamiento y que permite su utilización con el sistema ROS.

Se ha podido observar que la utilización de estos algoritmos proporciona una fiabilidad bastante alta y un tiempo de procesado no muy elevado, llegando a tasas de refresco de 5 Hz en un equipo de características de gama media.

El principal problema que se ha encontrado durante el desarrollo del trabajo ha sido la instalación de los distintos paquetes que son necesarios para el funcionamiento de la red. El proceso de configuración de los distintos componentes ha sido bastante tedioso, llegando en ocasiones a tener que renunciar a las características que ofrecen las últimas versiones de los módulos para que pudiese ser compatible con el código existente.

En cuanto a la red, los resultados han sido bastante satisfactorios, a excepción de los encontrados en los peatones. Como se ha indicado a lo largo del trabajo, estos fallos en la detección de peatones se pueden deber a la precisión empleada en la creación de los pilares. Sin embargo, el aumento de precisión de los pilares aumentaría el tiempo de procesamiento enormemente. En vista de los resultados obtenidos se puede concluir que la arquitectura de Pointpillars es una arquitectura válida para su uso en detecciones en vehículos autónomos.

Se ha determinado que se puede emplear esta arquitectura de red en un sistema real donde se tenga información de LiDAR. Uno de los trabajos posteriores sería realizar la implementación de esta red en el sistema real, donde las detecciones tengan una razón de ser. También sería interesante realizar el entrenamiento con datos obtenidos directamente desde el vehículo que se vaya a usar, en vez de utilizar una base de datos. Otra vía que abordar sería la de realizar la implementación de esta red en la plataforma TensorRT, lo cual le proporcionaría un desempeño temporal aún mayor.

# 9 PRESUPUESTO

En este capítulo se detalla el coste total del proyecto. Se estimará el coste material de los equipos utilizados y el coste de la mano de obra. Cabe mencionar que se ha utilizado en gran medida herramientas de software libre y por lo tanto no existe ningún pago de licencias.

## 9.1 COSTES MATERIALES

| *Concepto* | *Unidades* | *Coste unitario* | *Coste total* |
|---|---|---|---|
| *Ordenador* | 1 | 690,00€ | 690,00€ |
| *Ubuntu 18.04* | 1 | 0,00€ | 0,00€ |
| *Microsoft Officce 365* | 1 | 0,00€ | 0,00€ |
| *Programas de evaluación KITTI* | 1 | 0,00€ | 0,00€ |
| *Subtotal* | | | *690,00€* |

*Tabla 3: Presupuesto costes materiales*

## 9.2 COSTES MANO DE OBRA

| *Concepto* | *Unidades* | *Coste unitario* | *Coste total* |
|---|---|---|---|
| *Horas ingeniería* | 320 | 11,50 € | 3.860,00€€ |
| *Horas escritura* | 30 | 9,00€ | 270,00€ |
| *Subtotal* | | | *4.130,00€* |

*Tabla 4: Presupuesto costes mano de obra*

## 9.3 COSTES TOTALES

| *Concepto* | *Unidades* |
|---|---|
| *Costes materiales* | 690,00€ |
| *Costes mano de obra* | 4.130,00€ |
| *Subtotal* | 4.820,00 € |
| *IVA (21%)* | 1.012,20 € |
| *TOTAL* | *5.832,00 €* |

*Tabla 5: Presupuesto total*

# BIBLIOGRAFÍA


[1] S. V. H. C. L. Z. J. Y. O. B. Alex H.Lang, «PointPillars: Fast Encoders for Object Detection from Point Clouds,» p. 9, 2018.

[2] A. G. a. P. L. a. R. Urtasun, «Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite,» de *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[3] C. R. Qi, «PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation,» 2017.

[4] C. R. Qi, «Frustum PointNets for 3D Objetc Detection from RGB-D Data,» 2018.

[5] Facebook research, «GitHub SparseConvNet,» [En línea]. Available: https://github.com/facebookresearch/SparseConvNet. [Último acceso: 25 08 2019].

[6] Nutonomy, «PointPillars for KITTI object detection,» [En línea]. Available: https://github.com/nutonomy/second.pytorch. [Último acceso: 07 02 2019].

[7] Y. Yan, «SECOND fot KITTI/NuScenes object detection,» [En línea]. Available: https://github.com/traveller59/second.pytorch. [Último acceso: 12 02 2019].

[8] Y. M. B. L. Yan Yan, «SECOND: Sparsely EmbeddedConvolutional Detection,» *Sensors — Open Access Journal*, p. 17, 2018.

[9] Python, «Manual de Python,» [En línea]. Available: https://docs.python.org/3/tutorial/. [Último acceso: 23 07 2019].

[10] PyTorch, «Manual PyTorch,» [En línea]. Available: https://pytorch.org/docs/stable/torch.html.

[11] ROS, «Wiki de ROS,» [En línea]. Available: http://wiki.ros.org/es.

[12] NVIDIA, «Manual de instalación y programación CUDA,» [En línea]. Available: https://docs.nvidia.com/cuda/.

[13] D. A. D. E. C. S. R. C.-Y. F. A. C. B. Wei Liu, «SSD: Single Shot Multibox Detector,» p. 17, 2016.

[14] A. W. X. Y. K. K. Bichen Wu, «SqueezeSeg: Convolutional Neural Nets with Recurrent CRF for Real-Time Road-Object Segmentation from 3D LiDAR Point Cloud,» 2017.

[15] Docker, «Manual de Docker,» [En línea]. Available: https://docs.docker.com/.

Universidad de Alcalá

Escuela Politécnica Superior

Universidad
de Alcalá