Universidad Miguel Hernández de Elche

MASTER UNIVERSITARIO EN ROBÓTICA

Implementación y estudio de funcionamiento de un detector de objetos en nubes de puntos tridimensionales

Trabajo de Fin de Máster

Curso 2021/2022

Autor: Álvaro Martínez Ballester
Tutor/es: Arturo Gil Aparicio
Luis Payá Castelló

AGRADECIMIENTOS

En primer lugar, me gustaría agradecer toda la labor de los docentes e investigadores del Máster en Robótica que me han proporcionado conocimientos necesarios para poder llevar a cabo este trabajo de fin de máster. Por su puesto, en especial al departamento de Sistemas y Automática de la UMH, a Arturo Gil Aparicio y a Luis Payá Castelló, por darme la oportunidad de desarrollar este proyecto.

Agradecer también a mi familia por el apoyo incondicional durante toda mi vida académica, por haber estado ahí para mí siempre que lo he necesitado, pendientes siempre de mi progreso y celebrando mis triunfos como suyos.

RESUMEN

El objetivo principal de este proyecto es el de la implementación de un detector de objetos a partir de las nubes de puntos 3D obtenidos mediante LIDAR, para estudiar su funcionamiento, tanto con el dataset de KITTI como con nubes de puntos 3D obtenidas durante el desarrollo del proyecto.

La detección de los llamados objetos (normalmente elementos dinámicos del entorno como son los coches, las bicicletas, los peatones, etc...) es útil para la realización de un mapeado robusto sin elementos dinámicos, a la vez que puede servir en la planificación y modificación automática de rutas en robótica móvil.

ÍNDICE

Capítulo 1

Introducción

1.1. Motivación

El desarrollo y la innovación tecnológica son uno de los pilares más importantes en la sociedad actual, ganando cada vez más y más peso. La aparición de nuevas líneas de investigación y la implementación de las tecnologías más desarrolladas en el día a día son unos de los motivos por lo que cada vez nuestras vidas cambian más y más rápido.

Uno de los campos tecnológicos que más cambios está provocando y que, con total seguridad, provocará en un futuro cercano, es el de la robótica móvil. Multitud de líneas de investigación distintas, compartidas con otros muchos campos científicos o no, convergen en el campo de la robótica móvil. Desde la conducción autónoma, pasando por robots domésticos hasta aplicaciones de mayor infraestructura y coste como pueden ser la industria espacial o la militar son unos de los pocos ejemplos de aplicaciones que se han abierto paso en nuestra realidad tecnológica de la mano de la robótica móvil.

Uno de los campos con más líneas de investigación abiertas es el de la conducción autónoma. En concreto, las que conllevan la comprensión del entorno del vehículo, les queda aún mucho por recorrer. Esto es debido a que hacer entender el entorno y lo que se encuentra en él a

una máquina como lo haría un humano es un reto para nada trivial. Es ahí donde entran en juego las inteligencias artificiales, en gran auge esta última década y presente en multitud de campos científicos.

1.2. Objetivos

El objetivo de este proyecto es el de la implementación de un detector de objetos (elementos dinámicos dentro de un entorno como lo son los vehículos y las personas) en nubes de puntos tridimensionales obtenidas mediante un LIDAR, para estudiar con detalle su funcionamiento. Más en concreto, del detector de objetos PointPillars, basado en un conjunto de redes neuronales profundas, para detectar coches, ciclistas, peatones y otros vehículos.

El funcionamiento de este detector se realizará mediante el uso de las nubes de puntos del dataset de KITTI [1], y de nubes de puntos obtenidas mediante un LIDAR durante el desarrollo del proyecto.

1.3. Estructura del proyecto

El proyecto está estructurado de la siguiente forma:

Capítulo 1: Introducción. Se describe de manera introductoria el trabajo mediante la motivación del mismo y los objetivos a lograr, a su vez que la estructura contenida en este.

Capítulo 2: Estado del arte. Contiene un desarrollo extendido de los conceptos sobre los que se basa el proyecto, como son el diseño y funcionamiento de PointPillars [2] de manera general, otros detectores de objetos a partir de nubes de puntos tridimensionales y sus particularidades.

Capítulo 3: Redes Neuronales Profundas. En este capítulo se describen los fundamentos de las redes neuronales profundas, que se usan en el popularmente conocido *Deep Learning*. Nos adentraremos en su arquitectura, en la arquitectura usada en PointPillars [2], el estudio de su funcionamiento y cómo y por qué se usan para detectar y clasificar objetos.

Capítulo 4: Experimentos. Se describirá el diseño de los experimentos llevados a cabo para el estudio del funcionamiento del detector de objetos. También se hablará sobre las limitaciones y problemas encontrados.

Capítulo 5: Conclusiones. Resume los aspectos más importantes del trabajo realizado, valorando los resultados y comentando las posibles futuras líneas de investigación.

Capítulo 2

Estado del Arte

El propósito de este apartado es el de conocer las investigaciones más importantes en las que se basa este proyecto para así poder documentarlo. Empezando por lo desarrollado en [2] y continuando por otras investigaciones llevadas a cabo con objetivos similares.

2.1. PointPillars

PointPillars usa las nubes de puntos como entrada y estima cajas 3D orientadas para coches, personas y ciclistas, además de otros vehículos [2]. La estructura de PointPillars se divide en tres etapas, la primera consiste en un codificador de características que convierte la nube de puntos a una pseudoimagen bidimensional; la segunda es una estructura convolucional 2D que procesa la pseudoimagen en una representación de alto nivel; y la tercera es una proceso de detección que detecta y realiza la regresión de cajas tridimensionales.

2.1.1. Diferencias con otros métodos

Una de las características más importantes de PointPillars respecto a otros métodos de detección de objetos a partir de nubes de puntos es su velocidad, dominando el estado del arte, siendo capaz de funcionar a 62 Hz con una CPU Intel i7 y una GPU 1080ti. Esto es debido a la ausencia de redes convolucionales 3D.

El uso de redes convolucionales tridimensionales está presente en otros métodos [3, 4]. Uno de ellos es el de VoxelNet [5], el cual divide el espacio en voxels, aplicando en ellos PointNet [6], seguido de una red convolucional 3D para consolidar el eje vertical, aplicando finalmente una red neuronal bidimensional para la detección como podemos observar en la figura 2.1. En el caso de VoxelNet, el tiempo de inferencia es de tan solo 4.4 Hz. Más adelante, SECOND [7] mejoró el rendimiento, pero la red neuronal tridimensional continuaba siendo el mayor escollo para la mejora del método.

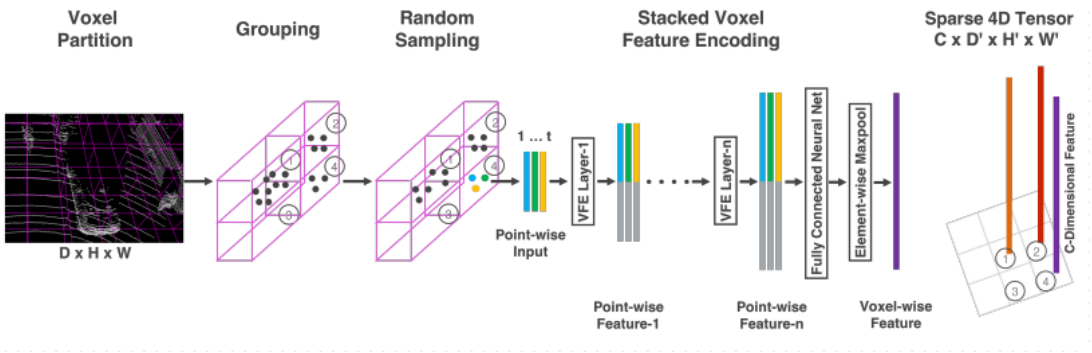

Figura 2.1. Arquitectura de la red de extracción de características a partir de una nube de puntos en VoxelNet [5], obteniendo un Tensor de cuatro dimensiones.

Otros métodos propuestos previamente [8, 9, 10, 11] contemplan el uso de la vista aérea de la nube de puntos, los cuales ofrecían ciertas ventajas como la ausencia de ambigüedad de escala y la casi completa ausencia de oclusión. Sin embargo la desventaja se encuentra en que la

dispersión de los puntos en una vista aérea hace que las redes convolucionales sean poco prácticas e ineficientes.

Para eliminar este problema se propone la partición del plano en rejillas de cierto tamaño fijo, para las que se aplica un método de extracción de características a cada punto. Sin embargo, estos métodos pueden ser subóptimos debido a que quizás se requiera un gran esfuerzo para que el método de extracción de características sea capaz de generalizar.

Para mejorar los métodos comentados anteriormente, PointPillars basa su funcionamiento en la transformación de una nube de puntos en una pseudoimagen. La nube de puntos, al igual que en [5, 8, 9, 10, 11], es divida en rejillas en el plano x-y, creando así los pilares.

Los puntos de la nube contienen cuatro coordenadas (x, y, z, *reflectancia*), a los que se les añade la distancia a la media aritmética del pilar (x_c, y_c, z_c) y la distancia al centro del pilar en coordenadas x-y (x_p, y_p). De esta manera se crea el vector D de nueve dimensiones. La mayoría de pilares estarán vacíos debido a la dispersión de la nube de puntos. Para explotar la dispersión de la nube de puntos se impone un límite tanto en el número de pilares no vacíos por muestra (P) como al número de puntos por pilar (N), creando un tensor denso de tamaño (D, P, N) como podemos observar en la figura 2.2. Si la muestra o el pilar contiene demasiados datos como para caber en el tensor fijado, los datos se muestrean de manera aleatoria. De la misma manera, si la muestra o el pilar no tienen los datos suficientes para poblar el tensor, se rellena de ceros [2].

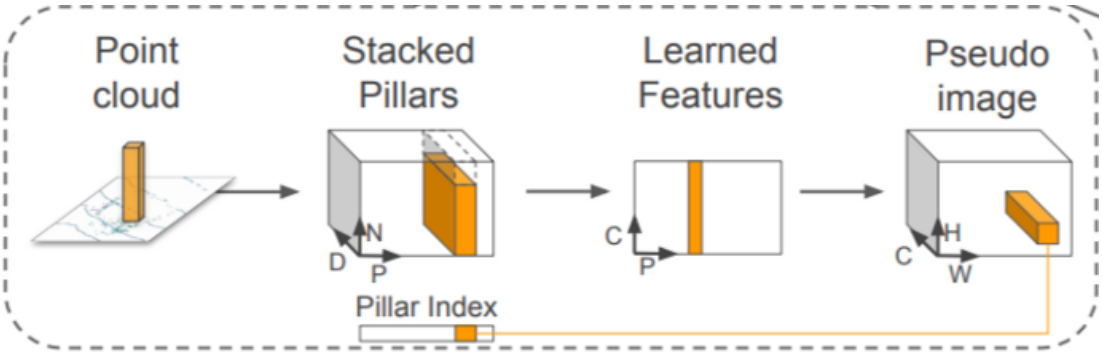

Figura 2.2. Arquitectura de la red de extracción de la pseudoimagen a partir de una nube de puntos en PointPillars [2].

Una vez realizado este proceso, se aplica una capa neuronal lineal, seguida de una normalización de lotes (batch-norm) [12] y una ReLu [13]. Así generamos un tensor de (C, P, N) dimensiones. A esto le sigue una capa de máximos sobre los canales para obtener un tensor de tamaño (C, P). La pseudoimagen final está formada por (C, H, W) donde H y W indican la altura y el ancho del plano respectivamente.

Capítulo 3

Redes Neuronales Profundas (Deep Learning networks)

El aprendizaje profundo (deep learning) es una particularidad del aprendizaje automático (machine learning). La filosofía del aprendizaje profundo consiste en trabajar con un gran volumen de datos sin tratamiento previo. Es decir, a diferencia del aprendizaje automático, donde se extraerían las características más importantes, como por ejemplo, en una imagen; en el aprendizaje profundo las características son inferidas y extraídas de forma automática directamente en el proceso.

Una de las particularidades del aprendizaje profundo es que, por regla general, no necesita un procesamiento previo de los datos, lo que supone una ventaja respecto a otros métodos de inteligencia artificial. Además, al no estar el aprendizaje sujeto al tratamiento previo de datos (extracción de características) la precisión y capacidad en el aprendizaje automático no queda limitado más allá de los propios datos. Sin embargo, esto lleva a la necesidad del uso de una gran cantidad de ellos para realizar el aprendizaje, en comparación con otros métodos de aprendizaje automático, además de una gran necesidad de computación para el procesamiento de estos.

El uso del aprendizaje profundo está cada vez más extendido en la actualidad debido a las ventajas comentadas previamente, tanto en el ámbito científico e investigador como en el empresarial. Se puede aplicar en prácticamente en cualquier proceso de aprendizaje automático que requiera una gran cantidad de datos.

De manera general y menos específica, los tipos de datos usados en el aprendizaje profundo pueden ser económicos, de mercados, médicos o sociales en general (procesamiento del lenguaje p. e.), aunque más adelante nos centraremos concretamente en el uso de imágenes y nubes de puntos.

3.1. Arquitectura y funcionamiento

En este apartado nos centraremos en la arquitectura de las redes neuronales profundas aplicadas a imágenes y nubes de puntos. A la hora de tratar este tipo de datos, hemos de tener en cuenta que la particularidad que tienen respecto a otros es la relación espacial en estos.

Las imágenes y nubes de puntos están formadas por distintos elementos (objetos, espacios, etc...) que queremos identificar y entender de manera automática. Estos elementos están formados por píxeles o puntos, guardando generalmente una relación espacial entre ellos. Esto sugiere la posibilidad del uso de operaciones que contemplen tal relación, es decir, añadir cierto conocimiento previo de la naturaleza de los datos, no en su procesamiento o extracción de características, sino en el propio diseño de la red neuronal profunda. Estas operaciones aplicadas en las capas de las redes neuronales profundas son conocidas como convolucionales, y basan su funcionamiento a, como su propio nombre indica, operaciones convolucionales.

Las operaciones convolucionales aplicadas a matrices n-dimensionales se basan en la aplicación de una máscara de unas dimensiones en concreto a todos o a parte de los datos de la matriz (píxeles o puntos). La máscara es aplicada a un valor de la matriz, obteniendo como resultado un valor numérico combinación lineal de los datos de la matriz dentro de la máscara.

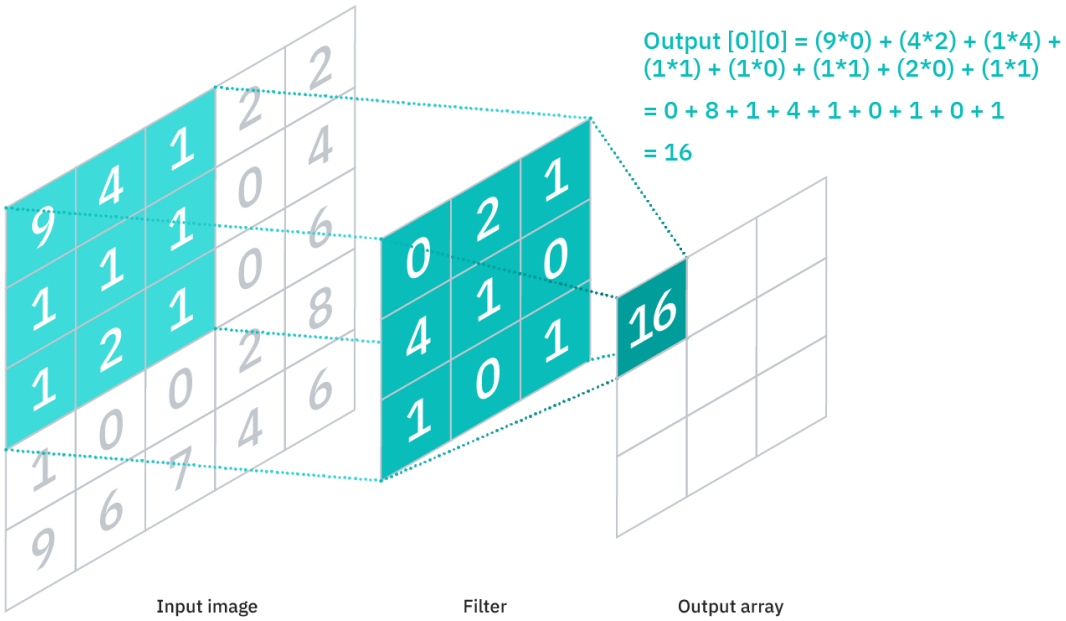

Figura 3.1. Ejemplo de aplicación de una máscara en una operación convolucional [14].

Por lo que en función del tamaño de la máscara y del número de datos de la matriz de inicio en los que se aplique, obtendremos una matriz de un tamaño menor con información de la matriz previa en función de la máscara empleada, como podemos contemplar en la figura 3.1.

Cuando hablamos de redes neuronales profundas, en las que la entrada es una matriz bidimensional (imágenes o pseudoimágenes), tridimensional (nubes de puntos), o n-dimensional; las operaciones convolucionales aplicadas a estos datos para obtener una nueva capa

neuronal equivalen a realizar una interconexión entre capas neuronales mucho más robusta (relaciona los datos con los de su entorno) y mucho menos pesada computacionalmente (reduce mucho el número de operaciones en comparación con las redes completamente interconectadas donde los datos muy alejados se afectan entre sí).

Es por estos motivos que las capas neuronales convolucionales son una de las partes principales de la arquitectura de redes en el aprendizaje profundo.

El diseño de las dimensiones de la máscara es algo prefijado, mientras que los pesos de la máscara (valor por el que se multiplican los pesos de la capa neuronal), cambiarán y se irán adaptando durante el aprendizaje.

3.2. Detección y clasificación de objetos en imágenes

Para la detección de objetos en imágenes, las arquitecturas de las redes neuronales desarrolladas son varias, pero generalmente están compuestas por cuatros tipos de capas. Estas son las convolucionales, las ReLu [13], las de pooling y las completamente conectadas.

Las capas de pooling son una particularidad de las capas convolucionales. Estas capas se aplican tras varias convoluciones para reducir el tamaño de los datos. Suelen aplicar una máscara de, por ejemplo, 2x2 en una capa bidimensional, a cada dos neuronas, reduciendo el tamaño de la capa resultante en un factor de dos. La operación convolucional realizada por la máscara puede ser cualquiera, pero generalmente se suele aplicar la operación que se quede con el máximo valor de los pesos dentro de la máscara (operación prefijada).

Las capas completamente conectadas calculan cada uno de los pesos de la capa de salida a partir de todos los pesos de la capa previa por lo que, si las capas contienen muchas neuronas, la computación de estas puede ser muy pesada. Es por esto que estas capas solo se suelen aplicar al final, cuando los datos ya son bastante reducidos.

Una de las primeras redes clasificadoras de imágenes con grandes resultados fue AlexNet [15]. Esta red intercala capas convolucionales con ReLu y Poolmax, finalizando con capas completamente conectadas para clasificar las imágenes como podemos observar en la figura 3.2.

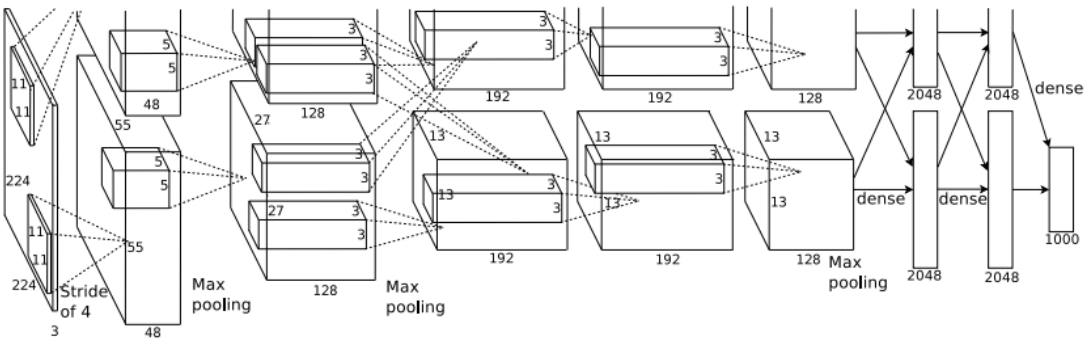

Figura 3.2. Arquitectura de la red neuronal AlexNet para la clasificación de imágenes.

Algunas arquitecturas desarrolladas posteriormente han mejorado el desempeño en la detección y clasificación de objetos en imágenes, desarrollando arquitecturas con cada vez más capas (redes cada vez más profundas), hasta llegar a aplicar cientos de estas para mejorar su precisión y rendimiento.

Uno de los ejemplos es la red ResNet [16] la cual utiliza un total de 152 capas para la clasificación de imágenes.

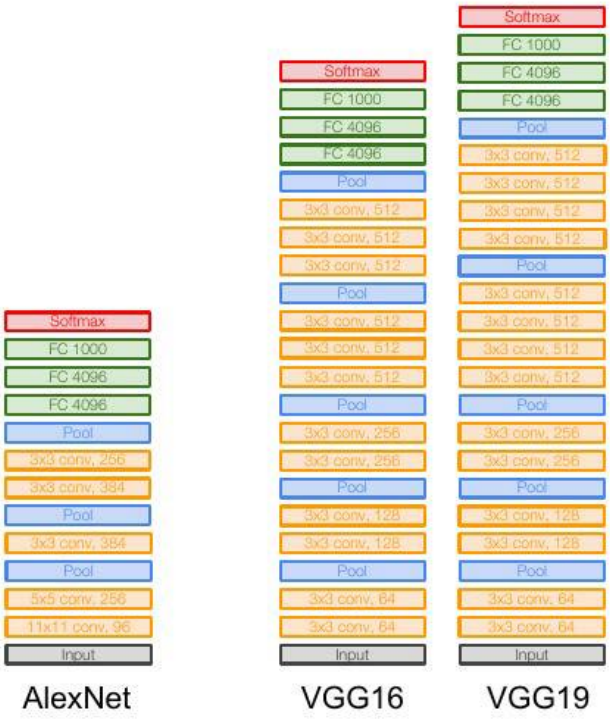

Figura 3.3. Capas utilizadas para diferentes arquitecturas. Se compara AlexNet con arquitecturas desarrolladas posteriormente.

3.3. Detección y clasificación de objetos en nubes de puntos

Como hemos desarrollado en el estado del arte (apartado 2.1), parte de las investigaciones llevadas a cabo para la detección de objetos a partir de nubes de puntos utilizaban redes convolucionales tridimensionales [3, 4, 5, 7]. Sin embargo, la inferencia de datos es demasiado lenta, siendo las redes convolucionales 3D un cuello de botella en el proceso.

Por otro lado, en [2] se propone la extracción de una pseudoimagen para poder eliminar el uso de una red convolucional 3D y utilizar una 2D, siendo esta segunda mucho menos costosa computacionalmente.

3.3.1. Redes convolucionales en PointPillars

Al proceso de obtención de la pseudoimagen le sigue el uso de una estructura convolucional (Backbone). La estructura de esta red se puede dividir en dos subredes. Una que produce características de arriba a abajo a una resolución espacial cada vez más pequeña y una segunda que aumenta la resolución y concatena las características de la primera.

La primera subred se caracteriza por una serie de bloques (S, L, F), en los que S es el paso (espacio entre neuronas) con el que se aplica cada máscara L bidimensional de 3x3 con F canales de salida, y cada uno es seguido de una normalización de lotes [12] y una ReLu [13].

La segunda subred, donde se realiza el aumento de resolución usando una red convolucional 2D traspuesta, aplica de nuevo una normalización de lotes [12] y una ReLu [13]. Finalmente se concatenan las características generadas con cada paso en los distintos bloques. Podemos ver esta sucesión de bloques de manera esquemática en la figura 3.4.

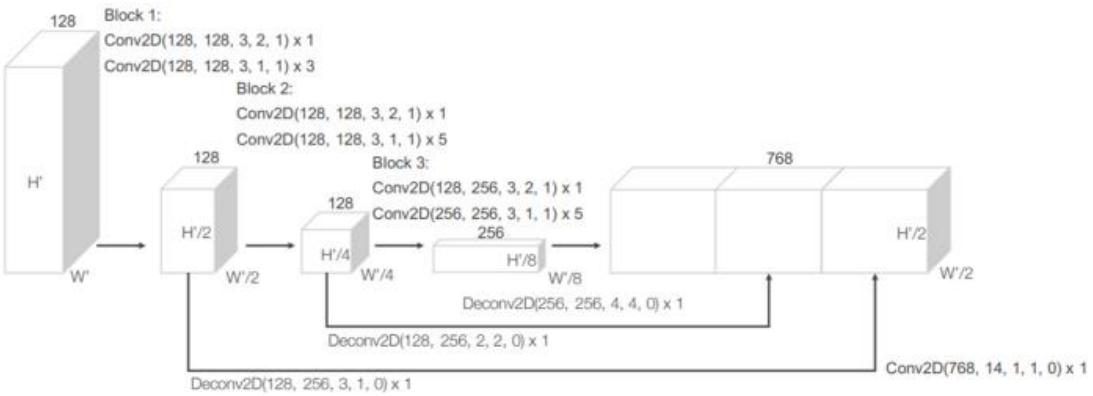

Figura 3.4. Arquitectura de la columna vertebral convolucional aplicada a la pseudoimagen en el método de PointPillars [2].

El paso posterior a la estructura convolucional es la proceso de detección. En este caso se aplica la configuración del *Single Shot Detector* (SSD) [17] (figura 3.5). Para emparejar las cajas predichas al inicio por la red solo se utiliza la intersección sobre unión (IoU) bidimensional, es decir, no se utilizan ni la elevación ni la altura.

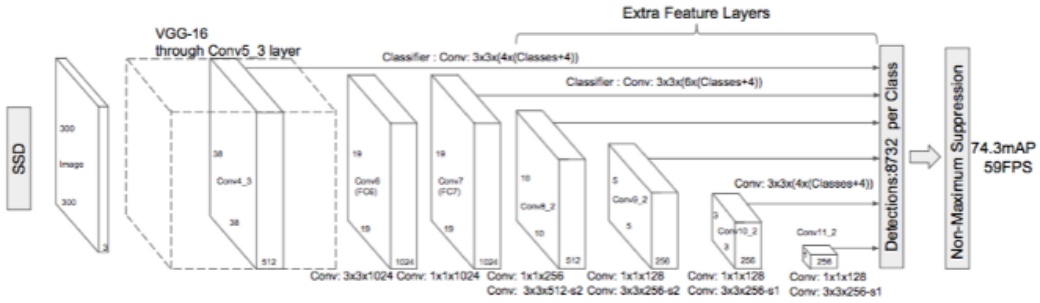

Figura 3.5. Arquitectura de la cabeza de detección en PointPillars [2] basada en la configuración del SSD [17].

3.4. Funcionamiento de clasificadores

Las redes neuronales, tanto profundas como no profundas, se diseñan para el aprendizaje y reproducción automática de distintas tareas. En este trabajo nos hemos centrado en el desarrollo de redes neuronales con la capacidad de clasificar y detectar objetos. Para estudiar su funcionamiento y los resultados obtenidos existen distintos parámetros, algunos de los cuales hemos empleado en este proyecto para el estudio de la implementación de PointPillars.

Para el estudio del funcionamiento de una red neuronal es necesaria una gran cantidad de datos de "ground truth". Estos datos son los que aportan el conocimiento durante el aprendizaje. En el caso de una clasificación de datos en un número de grupos concreto, necesitamos conocer a ciencia cierta que a qué clase pertenece cada dato, no sólo

para saber si la clasificación realizada es correcta, sino para corregir su funcionamiento durante el aprendizaje.

Una vez realizada la detección y clasificación de los datos y conociendo su realidad sobre el terreno, hay distintos modos de evaluar y validar los modelos y clasificadores diseñados.

3.4.1. Matriz de confusión

La matriz de confusión nos permite visualizar de manera rápida y precisa es la clasificación y, en caso de errores en la clasificación, cómo se cometen estos.

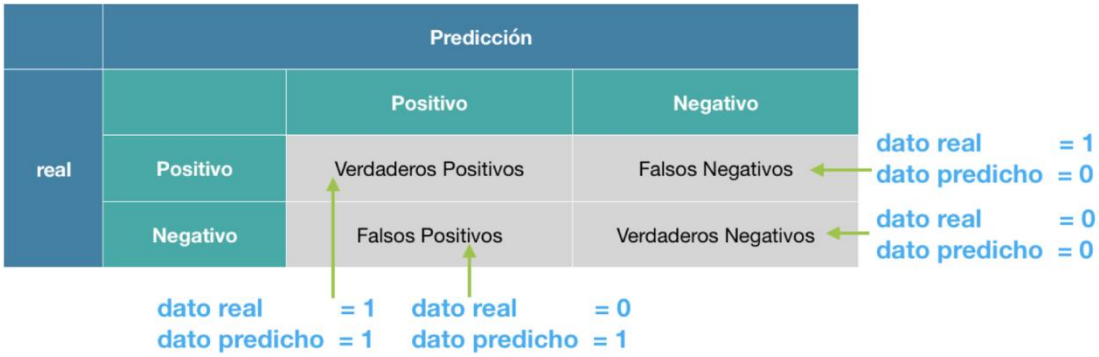

Figura 3.6. Ejemplo sencillo de una matriz de confusión de una clasificación en dos clases.

Como podemos observar en la figura 3.6, si las predicciones coinciden con la realidad, la matriz será diagonal. Para clasificaciones no binarias (con más de dos clases), obtenemos más información en cuanto a cómo falla la clasificación, es decir, si hay cierta clase real que se predice más incorrectamente como otra clase en concreto o no.

Teniendo los datos clasificados como verdaderos positivos, falsos positivos, falsos negativos y verdaderos negativos, podemos obtener distintos parámetros que dan un valor cualitativo sobre la clasificación, aunque nos centraremos en los dos más importantes

3.4.1.1. Precisión

El parámetro de precisión de un clasificador a partir de una matriz de confusión se obtiene de la siguiente forma:

$$Precisión = \frac{verdaderos\ positivos}{verdaderos\ positivos + falsos\ positivos}$$

En el caso de clasificar entre, por ejemplo, cuatro clases, tendríamos la precisión de la clasificación para cada una de las clases.

3.4.1.2. Sensibilidad

El parámetro de la sensibilidad, más conocido como 'recall' se calcula del siguiente modo:

$$Sensibilidad = \frac{verdaderos\ positivos}{verdaderos\ positivos + falsos\ negativos}$$

Del mismo modo, tendremos una sensibilidad distinta para cada clase.

3.4.1.3. Relación precisión-sensibilidad

En la mayoría de los clasificadores se estudia la relación que tienen la precisión y la sensibilidad en función de un umbral. Es por esto que se suelen graficar en dos ejes, donde la tendencia general es que el aumento de uno de los dos parámetros suponga la caída del otro.

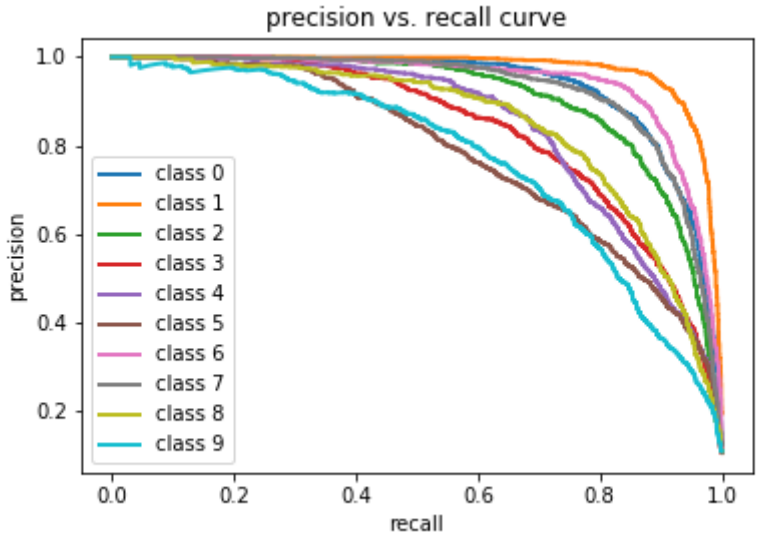

Figura 3.7. Ejemplo de gráfica de precisión-sensibilidad.

La clasificación es cualitativamente mejor para las gráficas con una mayor área bajo la curva de precisión-sensibilidad, lo que se conoce como 'mAP'. En estas gráficas observamos múltiples puntos de funcionamiento, algunos con mayor precisión y otros con mayor sensibilidad. En función del parámetro que más nos interese maximizar, se escoge uno de estos puntos para un valor del umbral concreto. Comúnmente se escoge el punto de la gráfica más cercano a las coordenadas (1, 1).

Capítulo 4

Experimentos

Para la implementación de PointPillars y el estudio de su funcionamiento nos basamos en el código desarrollado en [18], el cual fue modificado para realizar dicho estudio y experimentos. En [19] se explica de manera detallada el código desarrollado en [18], como funciona y como se implementa.

La base de datos utilizada es la de KITTI [1], la cual contiene hasta 7481 nubes de puntos etiquetadas para el entrenamiento de PointPillars.

4.1. Repositorio utilizado

En este apartado se describir con cierto grado de detalle como es el repositorio utilizado, alguno de los scripts más importantes a la hora de entender su funcionamiento y los cambios realizados para la realización del estudio de PointPillars.

4.1.1. config.py

Este fichero simplemente contiene los valores numéricos de los parámetros que se utilizan, tanto de la región de interés donde se realizan las predicciones como de la red neuronal diseñada. En este apartado añadimos un parámetro que indica el número de GPUs a utilizar durante el entrenamiento o la inferencia.

4.1.2. network.py

En este script se define toda la red neuronal de PointPillars (código en el anexo I). La funciones para crear los pilares y las cajas envolventes, sin embargo, se encuentran dentro de la carpeta src, en el único script que hay, point_pillars.cpp, programado en C++ para realizar este proceso muy repetido y repetitivo con una mayor velocidad de cómputo.

4.1.3. point_pillars_training_run.py

Este es el script que ejecuta el entrenamiento del modelo con los datos de Kitti. En este apartado simplemente modificamos lo necesario para poder realizar un entrenamiento utilizando múltiples GPUs de manera simultánea.

4.1.4. point_pillars_prediction.py

Este es el script ejecutable realiza la inferencia de los datos de Kitti y muestra por pantalla en forma de texto las etiquetas existentes y las predicciones realizadas. También lo modificamos para cambiar el número de GPUs a utilizar durante la inferencia.

4.1.5. inference_utils.py

Este script contiene la definición de la clase de la caja envolvente, la función para aplicar la supresión de no máximos explicada en el apartado 4.2.2, la clase para generar las cajas envolventes a partir de los elementos de regresión (posición, umbral de ocupación, clase, ocupación, etc...), el generador de datos de "ground truth" y un "focal_loss_checker" utilizado para mostrar que objetos son o no predichos.

Además nosotros añadimos una función utilizada más adelante para guardar los siguientes datos en un vector de la siguiente forma: objetos predichos que coinciden con los etiquetados, los etiquetados que han sido predichos, los predichos que no coinciden con ningún etiquetado y los etiquetados que no han sido predicho. Mediante esta separación de los datos podemos realizar el estudio del funcionamiento de la red de manera sencilla. Podemos observar esta parte del código en el anexo I.

4.1.6. readers.py

En este script únicamente encontramos clases para leer los ficheros a utilizar, teniendo ya incorporada la que lee los datos de Kitti, los cuales vienen en binario. Añadimos en el lector por defecto unas pocas líneas para poder leer datos de ficheros en formato .csv.

4.1.7. processors.py

Este script contiene clases para procesar los datos (crear los pilares llamando a funciones del fichero en C++) y para generar los datos para el entrenamiento o el testeo. En esta última clase añadimos la rotación de la nube de puntos.

4.1.8. Scripts creados

En este apartado comentaremos los scripts creados para realizar el estudio del funcionamiento de la red.

4.1.8.1. point_pillars_evaluation.py

Un script con una estructura muy similar que el del apartado 4.1.4, pero con el que guardamos los datos en el formato comentado al final del apartado 4.1.5. El código se encuentra disponible en el anexo I.

4.1.8.2. match_data.py

Es el script mediante el que extraemos los datos mostrados en el apartado 4.2.4. a partir de los datos guardados por el script del apartado 4.1.8.1. De nuevo, podemos encontrar el código en el anexo I.

4.2. Resultados obtenidos

La red neuronal de PointPillars localiza los objetos 3D, devolviendo los datos de las cajas delimitadoras que los definen (centroide, altura, anchura, largo, ángulo, etc...), así como su clasificación. La clasificación se realiza entre cuatro clases: 0 para únicamente coches, 1 para peatones y personas sentadas, 2 para ciclistas y 3 para camiones, furgones, caravanas, tranvías y el resto de objetos que no pertenecen a ninguna de las tres primeras clases.

4.2.1. Umbral de ocupación

Una de las características que describen las cajas delimitadoras es su índice de ocupación. Se trata de un valor de confianza de la predicción entre cero y uno. Las predicciones cuyo valor del índice de ocupación es

bajo pueden formar parte de los falsos positivos, por lo que utilizamos el umbral de ocupación, con el que nos deshacemos de las peores predicciones.

La variación del umbral de ocupación y el estudio de precisión y sensibilidad en función de éste es necesario para conocer con qué umbral obtenemos un mejor funcionamiento.

4.2.2. Supresión de No Máximos

La red neuronal realiza varias predicciones para un mismo objeto, por lo que es necesaria una supresión de no máximos. La supresión de no máximos clasifica las cajas delimitantes en base al índice de intersección sobre unión y valores de confianza y elige la que tenga mejor valor como la caja delimitante final. Podemos ver un ejemplo en la figura 4.1.

Si el valor del umbral utilizado para la supresión de no máximos es superior al de ocupación, la supresión de no máximos elimina las cajas debajo de ese umbral, por lo que es necesario que el valor umbral para la supresión de no máximos sea, como mínimo, igual o menor que el de ocupación. Sin embargo, si el umbral de supresión de no máximos es demasiado alto, la supresión es cada vez peor, por lo que pueden aparecer predicciones para el mismo objeto repetidas.

Para el estudio del funcionamiento de la red en función del umbral de ocupación, escogemos un umbral de supresión de no máximos igual al de ocupación, pero si el umbral de ocupación sobrepasa el 0.6, el umbral de supresión de no máximos se queda en 0.6.

Figura 4.1. Supresión de No Máximos para converger a una caja envolvente final cuando hay multiples cajas envolventes predichas por el algoritmo de detección de objetos [19].

4.2.3. Comparación Predicción-Realidad

Una vez realizada la detección de objetos por PointPillars, para saber si esta es correcta hemos de compararla con la nube de puntos con los objetos ya etiquetados.

Durante el entrenamiento, PointPillars sólo tiene en cuenta la intersección sobre unión (IoU) del área bidimensional que ocupa la caja (figura 4.2), sin tener en cuenta ni la elevación ni la altura. En nuestro caso, para determinar que un objeto ha sido predicho por PointPillars utilizamos el siguiente criterio:

$$\frac{distancia\ entre\ centroides}{media\ de\ los\ 3\ lados\ de\ la\ caja\ predicha} < 0.5$$

De esta forma podemos considerar cajas predichas como correctas si la distancia a la caja real no se aleja lo suficiente.

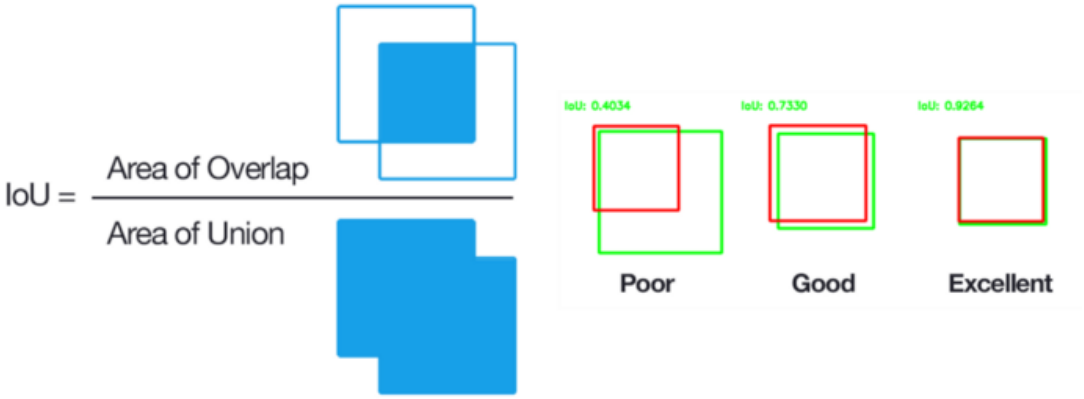

Figura 4.2. Cálculo del criterio de intersección sobre unión usado durante el entrenamiento de PointPillars para emparejar predicciones con los objetos etiquetados.

4.2.4. Precisión y Sensibilidad

Estudiamos el funcionamiento de PointPillars en función del umbral de ocupación para obtener los datos de precisión y sensibilidad.

Calculamos la precisión y la sensibilidad variando el umbral de ocupación y las representamos en la Figura 4.3, donde podemos comprobar como la PointPillars funciona mejor para la detección de objetos del tipo coche, ya que, como comentaremos en futuros apartados, son más numerosos en la base de datos de KITTI; mientras que empeora su funcionamiento para el resto de clases.

Si tenemos en cuenta que un funcionamiento óptimo de la red es el punto de cada una de las gráficas más cercano a un sensibilidad y precisión de 1, obtenemos que el umbral para la detección de las distintas clases de objetos que cumple este requisito es de 0.38 para la clase coche, 0.22 para la clase peatón, 0.24 para la clase ciclista y 0.26 para la clase de miscelánea, furgoneta, tranvía y camión.

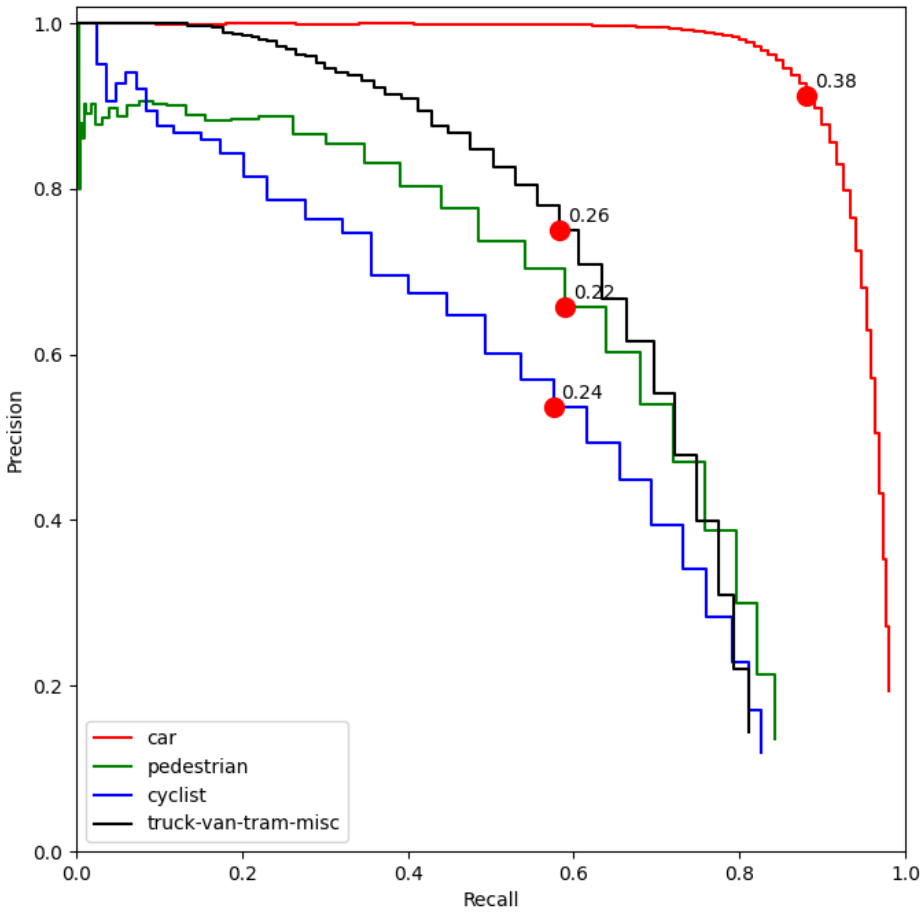

Figura 4.3. Precision contra Recall. Marcados en rojo los puntos con módulo precisión-sensibilidad más cercano a 1, etiquetado el valor del umbral de ocupación.

En la figura 4.4 observamos precisión y sensibilidad por separado respecto del umbral de ocupación que va desde 0.08 hasta 0.98. En el caso de la sensibilidad podemos observar como, si el umbral es cercano a uno (más exigente), la sensibilidad cae debido a que filtramos gran cantidad de predicciones por considerarlas de demasiada baja calidad.

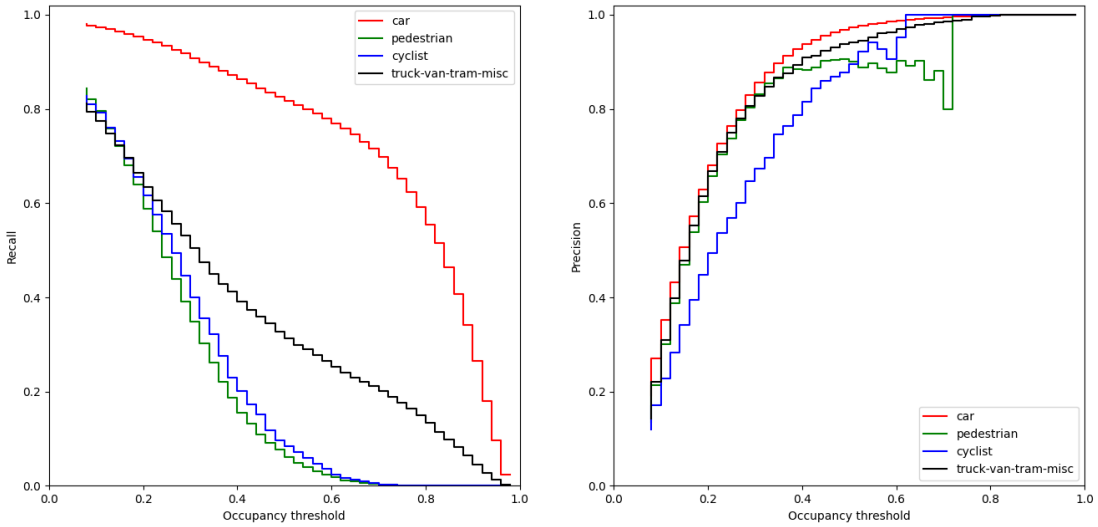

Figura 4.4. Precisión y sensibilidad en función del umbral de ocupación.

Por otro lado, en la precisión (figura 4.4), observamos como en cierto valor del umbral de ocupación, para las clases de peatones y ciclistas, la precisión que parecía decaer, aumenta a 1. Esto es debido a que a partir de un umbral tan alto, PointPillars es incapaz de detectar ningún objeto de estas clases para ninguna nube de puntos. La ligera caída de la curva antes del aumento de la precisión es debida al sesgo que se produce al tener un número de detecciones demasiado bajo.

También podemos observar las gráficas de verdaderos positivos, falsos positivos, falsos negativos y verdaderos negativos en función del umbral de ocupación en la figura 4.5. Vemos como la gráfica de verdaderos positivos es equivalente a la de la sensibilidad, la de falsos negativos es inversa a esta, a la vez que la de falsos positivos es inversa a la de precisión.

Para el caso de los verdaderos negativos, al no ser una clase a predecir y clasificar como tal, sino el espacio libre de objetos no clasificado, representamos la cantidad de nubes de puntos sin falsos negativos ni falsos positivos respecto del total.

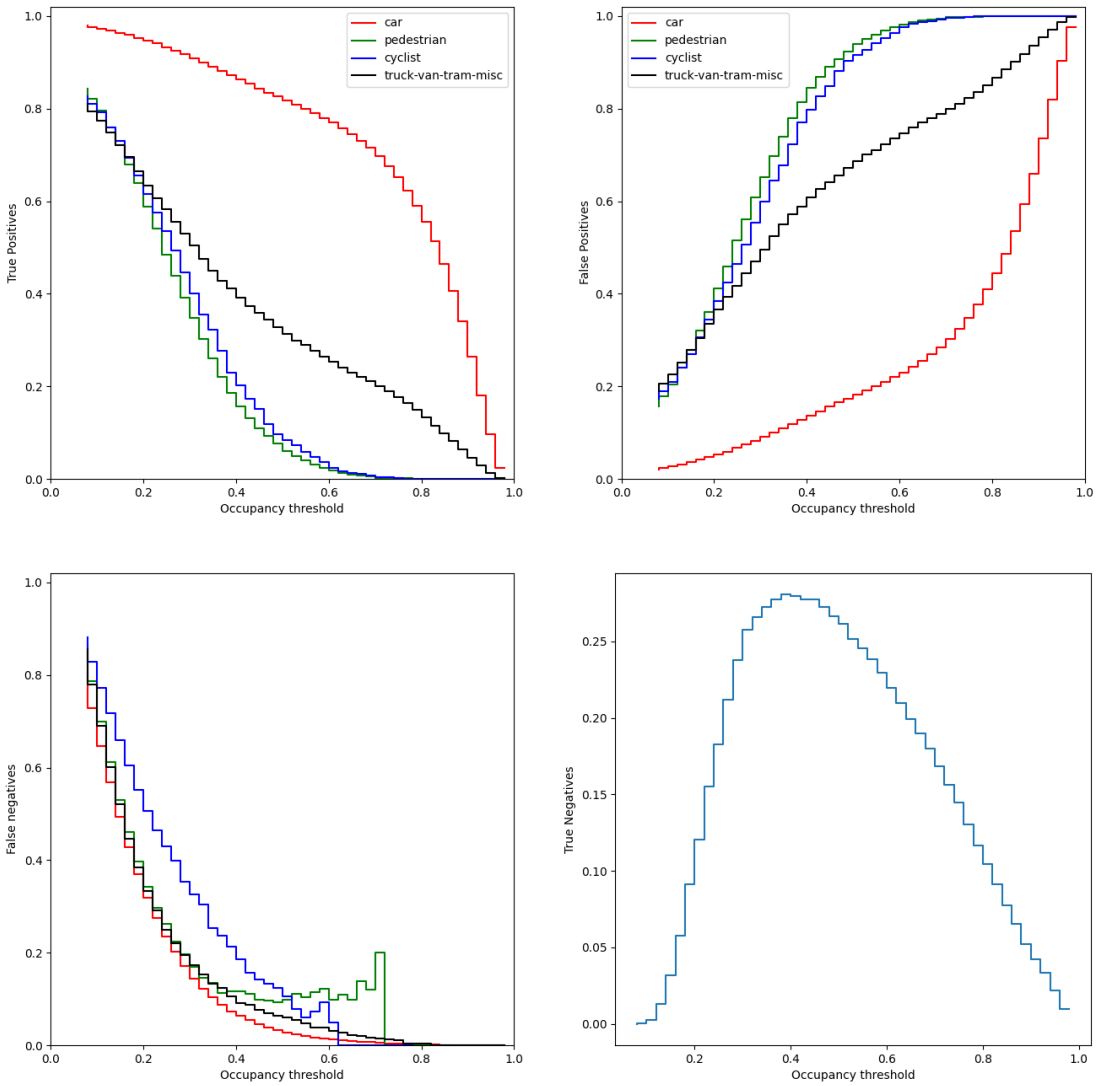

Figura 4.5. (De izquierda a derecha y de arriba abajo) Gráficas de Verdaderos Positivos, Falsos Positivos, Falsos Negativos y Verdaderos Negativos respecto al umbral de ocupación.

Los datos en una matriz de confusión para un punto de funcionamiento en concreto se pueden observar en la figura 4.6, donde se escoge un umbral de ocupación de 0.4, donde la precisión es mucho más alta que la sensibilidad para las clases peatón, bicicleta y miscelánea.

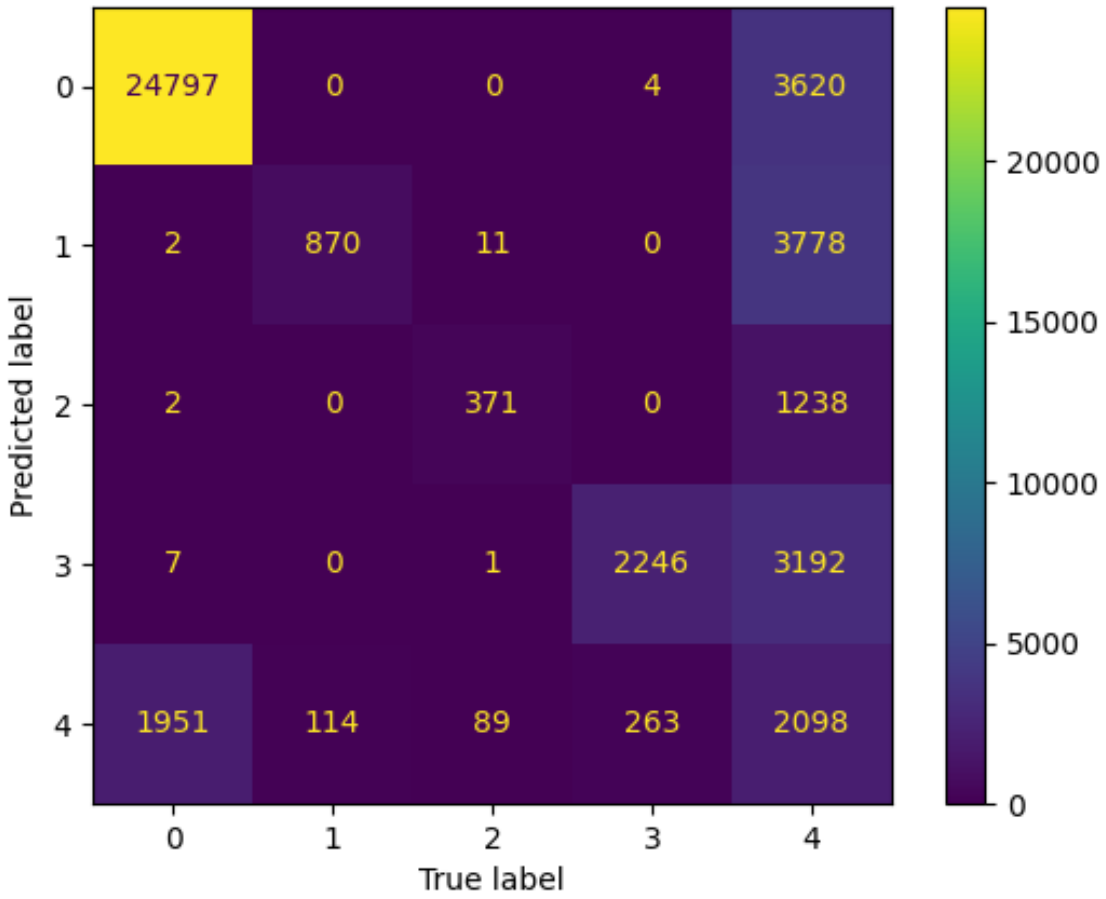

Figura 4.6. Matriz de Confusión para un umbral de ocupación de 0.4. Clases: 0 – coche, 1 – bicicleta, 2 – peatón, 3 – miscelánea / caravana / camión / tranvía, 4 – falsos positivos y falsos negativos.

Podemos comprobar en la figura 4.6, que una vez detectado el objeto, el error a la hora de clasificarlo es muy pequeño comparado con el error en la detección en sí. Para este valor del umbral de ocupación, vemos como para las clases 1, 2 y 3, se han detectado muchos más objetos de los que se deberían, lo que significa un número elevado de falsos positivos, como podemos observar en la gráfica de la figura 4.5, donde para estas tres clases tenemos un valor de falsos por encima de la mitad del número total de objetos.

4.3. Visualización

Durante el desarrollo del proyecto, se diseñaron visualizadores para observar las nubes de puntos acompañadas de las cajas envolventes de objetos, ya fuesen predichas o etiquetadas. De esta manera se puede comprobar de manera visual cuales son las predicciones correctas, dónde se encuentran y como de acertadas son de manera visual. Del mismo modo podemos observar predicciones en nubes de puntos no etiquetadas y de que clase es cada objeto predicho.

El software desarrollado hace uso tanto del repositorio de PointPillars [18] como de la librería de Open3D [20] disponible para su uso con Python.

Open3D es una librería de código abierto que ayuda al desarrollo de software para manejar datos 3D como son nubes de puntos obtenidas con LiDAR. En ella están implementados diferentes algoritmos y estructuras tanto en C++ como en Python.

4.3.1. Desarrollo de visualizadores

Para visualizar las nubes de puntos con sus cajas envolventes desarrollamos tres scripts distintos, uno para las nubes etiquetadas de KITTI (training), otro para las nubes no etiquetadas (testing) y finalmente otro para las nubes de puntos tomadas con un LiDAR propio más adelante.

4.3.1.1. Visualizador de training

Las cajas etiquetadas para el entrenamiento se visualizan en color negro, mientras que las a las predichas se les asignan dos colores distintos: verde si se considera que la predicción corresponde a una etiqueta, o rojo si no lo hace (código en el anexo II).

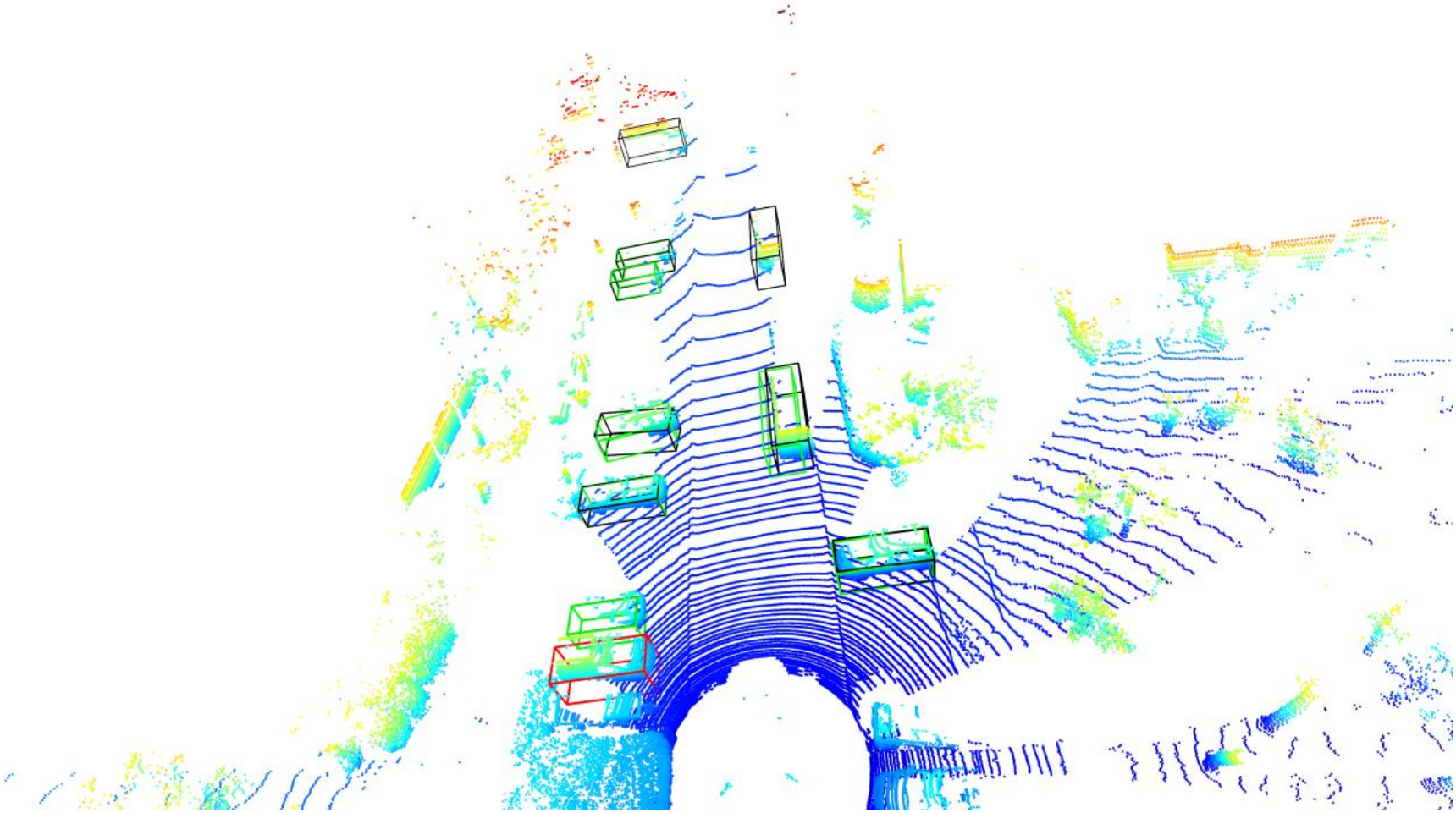

Figura 4.7. Ejemplo de nube de puntos de KITTI con etiquetas en negro y predicciones (en verde las correspondientes a las etiquetas y en rojo las que no corresponden a ninguna etiqueta).

4.3.1.2. Visualizador de testing y nubes de puntos no etiquetadas

Estos visualizadores, al no tener cajas ya etiquetadas a diferencia del visualizador de training, utilizan cuatro colores para visualizar cómo se clasifica cada objeto detectado. En este caso empleamos el negro para la clase coche, el verde para la clase ciclista, un tono de naranja para la clase peatón y el negro para la clase de miscelánea. Podemos consultar el código en el anexo II.

Podemos observar un ejemplo en la imagen de la figura 4.8. En este caso, como es normal, no podemos saber si la detección y la clasificación es correcta más allá que intuyendo nosotros mismos los objetos desde la visualización de la nube de puntos.

4.3.1.3. Funciones utilizadas

Para el desarrollo de los visualizadores desarrollamos un script con funciones que nos permitiesen la visualización de las cajas envolventes. La primera de ellas es para convertir el formato binario a formato pcd, que es el que utiliza Open3D para la representación de las nubes de puntos. Otra función es para extraer las esquinas de las cajas envolventes a partir de los datos extraídos en la inferencia, donde obtenemos el centroide; la orientación y la altura, anchura y longitud. La última función es para realizar una modificación que la librería de Open3D no ofrece, la cual es cambiar el ancho de las líneas, por lo que creamos una clase de falsas líneas, que en realidad son un conjunto de pequeños triángulos formando un cilindro de un radio determinado. Adjuntamos el código en el anexo II.

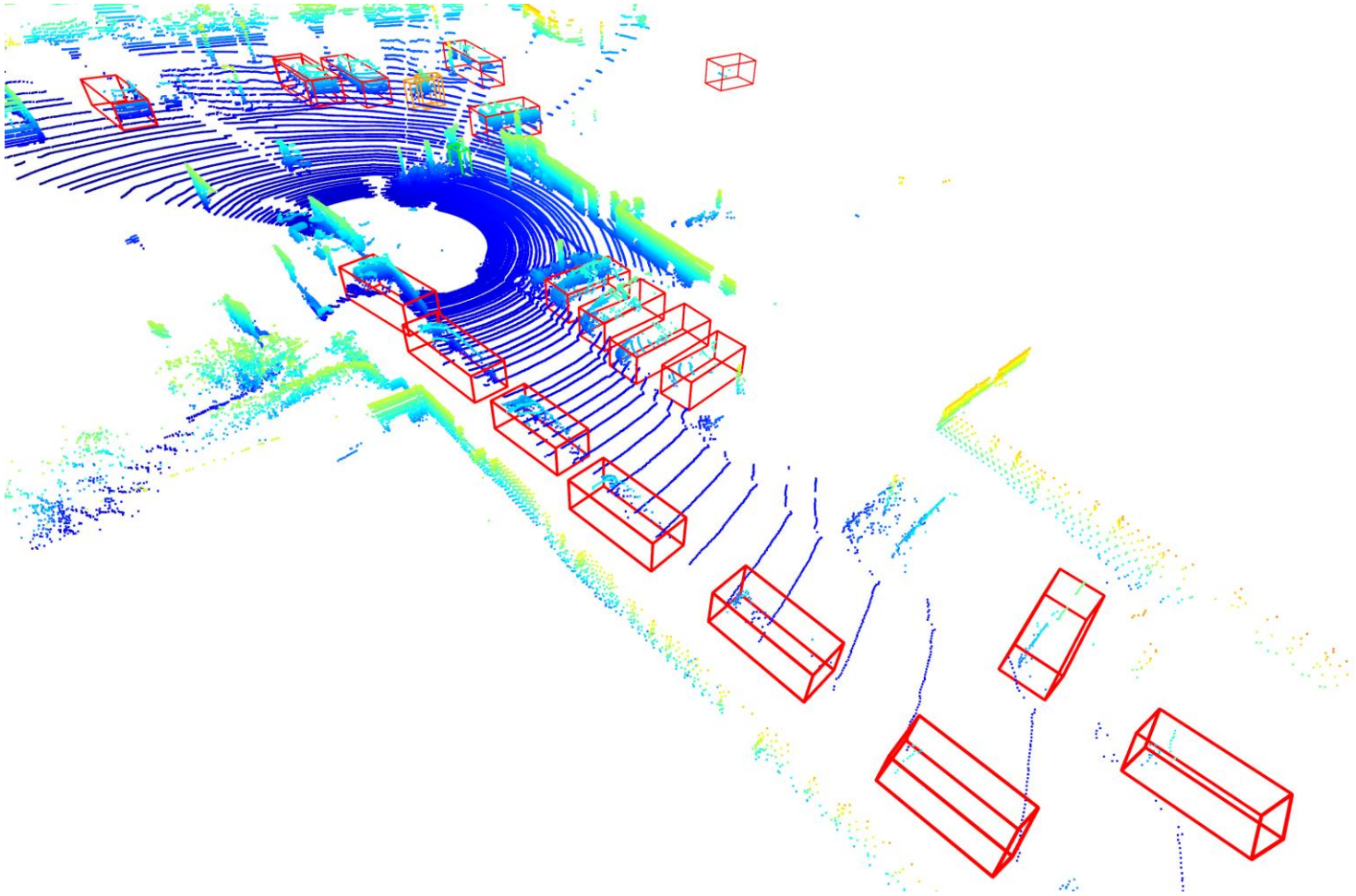

Figura 4.8. Visualización de la predicción en una nube de puntos de test. En rojo los elementos de la clase coche, en naranja los de clase ciclista y en verde los de clase peatón.

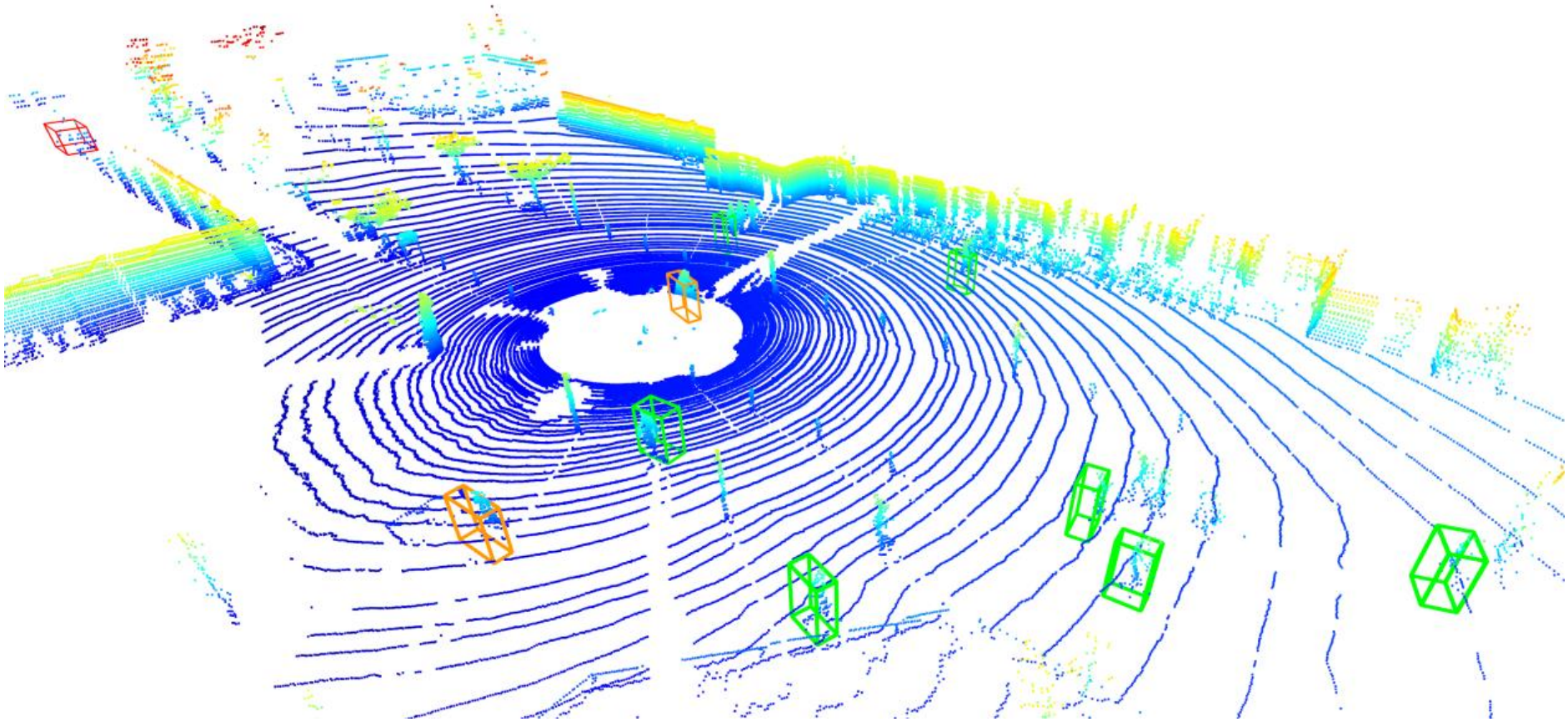

Figura 4.9. Visualización de la predicción en una nube de puntos de test. En rojo los elementos de la clase coche, en naranja los de clase ciclista y en verde los de clase peatón.

4.4. Kitti Dataset: limitaciones y adaptaciones

El entrenamiento de PointPillars se realiza con el dataset de Kitti. Uno de los problemas que se ha identificado durante la realización de este proyecto es que el etiquetado de las nubes de puntos no es preciso en su totalidad, dejando a veces objetos sin etiquetar (predicción en rojo de la figura 4.7 y figura 4.11) y otros con una caja envolvente con una mala orientación, dejando parte del objeto fuera de la nube de puntos (figura 4.10).

Claramente, no tener un dataset perfectamente etiquetado supone una limitación a la hora de entrenar una red, ya que el funcionamiento de esta se va a ver sesgado. La red se va a ver limitada al dataset, y en ocasiones, cuando haga una detección correcta durante el entrenamiento, se intentará corregir de manera errónea.

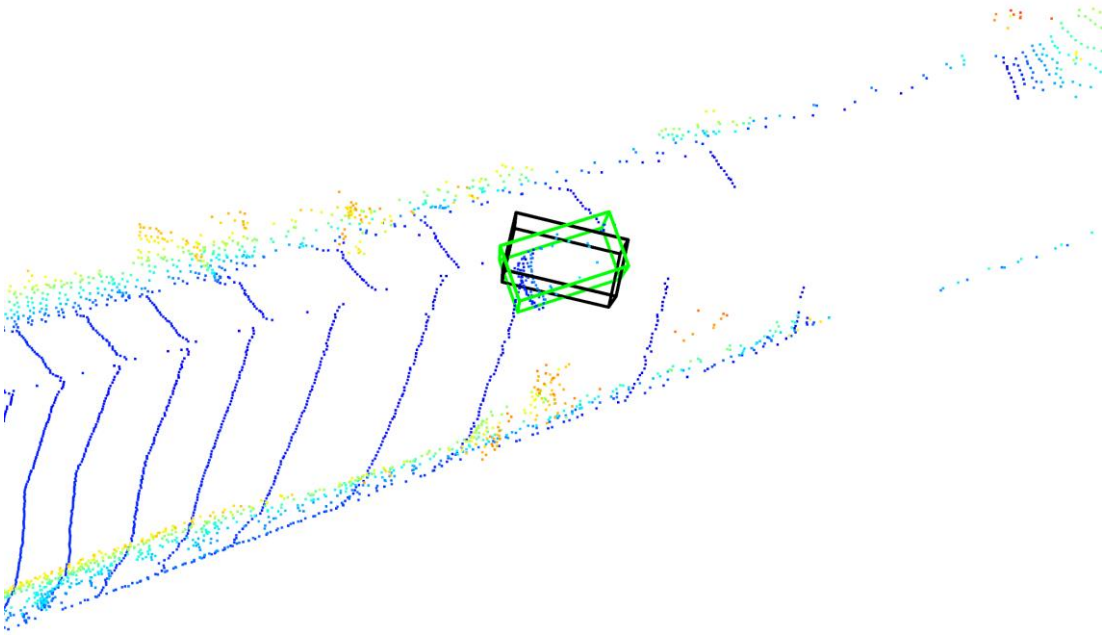

Figura 4.10. Nube de puntos etiquetada del dataset de Kitti. En negro la caja etiquetada y en verde la predicha.

Otra de las particularidades del dataset de Kitti es que el etiquetado se realiza únicamente en un rango de orientación limitado, es decir, solo están etiquetados los objetos de la parte delantera de la nube. Al entrenar la red con estos datos, las predicciones quedan limitadas a ese rango. Para solucionar este problema debemos de rotar la nube de puntos para realizar una detección de los objetos en la totalidad de la nube. Esto supone un inconveniente sobretodo en el coste computacional, teniendo que realizar varias predicciones para una misma nube de puntos.

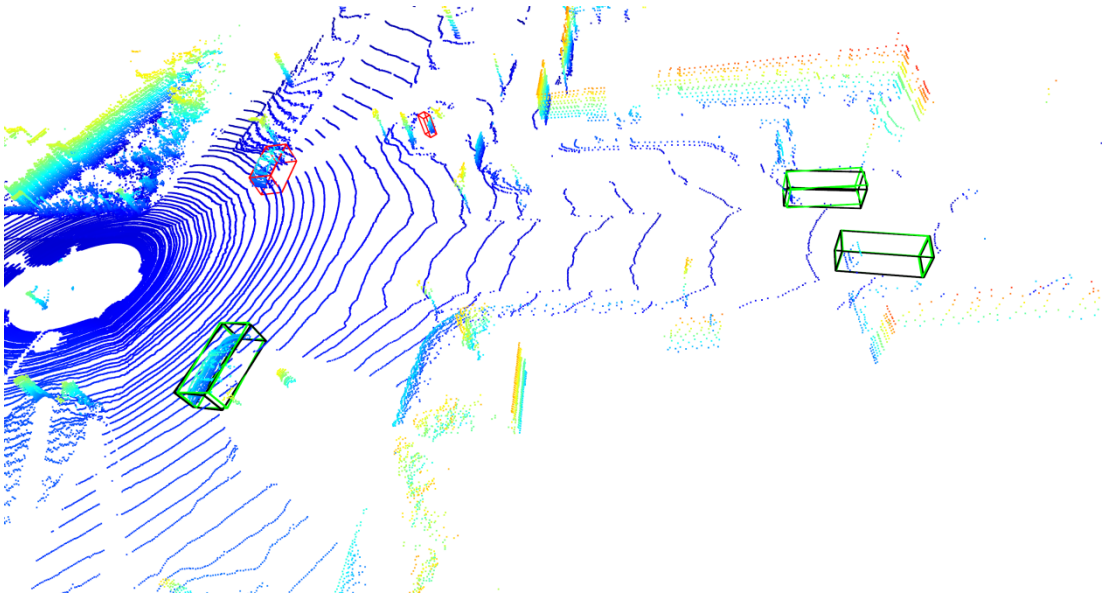

Figura 4.11. Nube de puntos etiquetada del dataset de Kitti con predicciones en verde y en rojo.

4.5. Experimentos con LiDAR

Una vez entrenada la red intentamos probar su funcionamiento para nubes de puntos detectadas con un LiDAR propio. En los experimentos realizados, se transportó el LiDAR con un robot móvil al exterior del edificio, llevándolo cerca de coches aparcados y con personas caminando

alrededor. Las zonas donde se captaron las nubes de puntos eran ricas en árboles y palmeras.

Los resultados obtenidos al realizar la detección de objetos de dichas nubes de puntos no fueron buenos. Apenas se detectaron objetos y de los pocos detectados, gran parte eran erróneos. Esto puede ser debido a distintos factores.

Uno de los factores a tener en cuenta es la resolución de los LiDAR empleados en cada caso. En el caso del LiDAR usado en con Kitti, las nubes de puntos tienen una resolución distinta respecto a las captadas con el LiDAR de nuestros experimentos. Mientras que las nubes de puntos de Kitti contenían alrededor de 110000 puntos, probamos a guardar nubes de puntos con diferentes resoluciones: 1024x64, 2048x64 o 2048x128; todas ellas sin resultados favorables.

Otro motivo que podemos encontrar y el cual puede ser muy influyente es el de la altura relativa del LiDAR. Mientras que en el LiDAR usado para el dataset de Kitti iba montado a una altura de 1.73 metros, el nuestro estaba montado en un robot a una altura mucho inferior (cercana al metro). Esta podría ser también una de las causas de porque la red no es capaz de detectar objetos para esas nuevas muestras.

Por último también se estudiaron los histogramas de reflectancia de algunas nubes de puntos. Al compararlas pudimos ver distinción de la distribución de los valores de reflectancia, aunque no podemos concluir que sea realmente uno de los problemas, ya que esto puede depender mucho de cada nube de puntos, por lo que no se le puede otorgar un gran peso a este factor.

Para una futura implementación del detector de PointPillars sería necesario el reentreno de la red, para lo que sería necesaria la creación de un dataset propio con nubes de puntos pertenecientes al entorno de uso.

Capítulo 5

Conclusiones

En este trabajo se ha desarrollado la implementación de un detector de objetos como es el de PointPillars, creando y diseñando software necesario para el estudio de su funcionamiento, a su vez que para la visualización de los datos y los resultados.

Del estudio realizado podemos extraer distintas conclusiones. Una de ellas es que el detector de objetos funciona mucho mejor para la detección de coches de las nubes de puntos del propio dataset de Kitti. Esto es debido a que es el elemento que aparece en mayor medida en las nubes de puntos de Kitti (figura 4.6 y figura 5.1), por lo que la red aprende mejor como detectar objetos de esa clase sobre las otras.

Otra de las conclusiones que hemos obtenido al probar el detector sobre nubes de puntos externas al dataset de Kitti es que no funciona bien para las nuevas nubes de puntos, por lo que sería interesante contemplar como opción a futuro la realización de un reentrenamiento de la red con nuevos datos, lo que supondría la creación de un dataset y la posible creación o implementación de una herramienta de etiquetado.

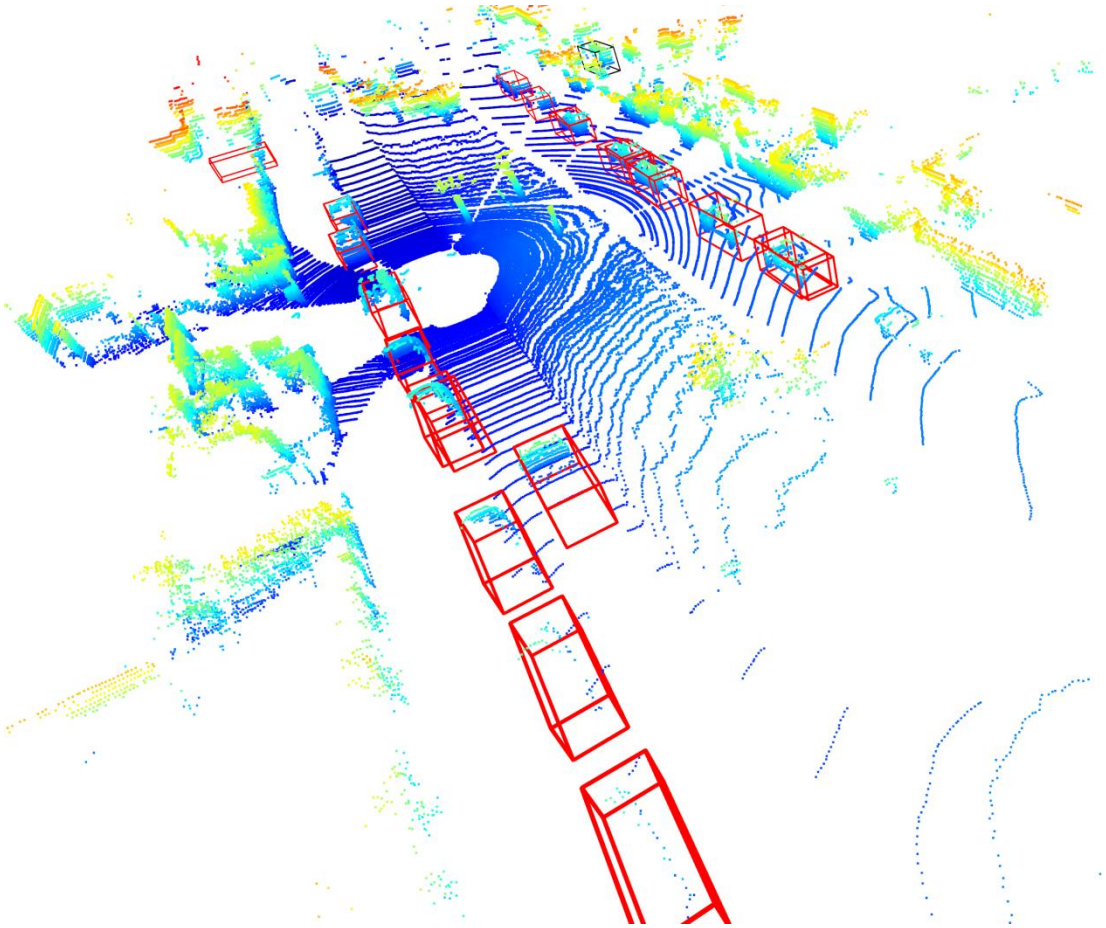

Figura 5.1. Visualización de la predicción en una nube de puntos de test. En rojo los elementos de la clase coche, en naranja los de clase ciclista y en verde los de clase peatón.

5.1. Futuras líneas de investigación

Como se ha comentado al principio del capítulo, la red de PointPillars rinde peor en la detección de clases ajenas a los coches. Para comprobar si hay capacidad de mejora en la detección de estos objetos, no solo para PointPillars, si no para otros detectores de objetos en nubes de puntos, sería interesante la creación de un dataset de nubes de puntos ricas en objetos de estas clases.

En caso de llegar a crear un dataset, la herramienta de etiquetado sería fundamental. Una de las posibles ideas a desarrollar sería la de la creación de una herramienta de etiquetado que midiese la distancia entre puntos de dos nubes de puntos tomadas desde la misma posición, una sin objetos (nube base) y la otra con objetos. Más tarde podrían obtenerse los objetos mediante una clasificación de k-means. Podría contemplarse la posibilidad de, por ejemplo, filtrar elementos dinámicos no clasificables como objetos. Los árboles o arbustos podrían ser un inconveniente al medir la distancia entre dos nubes de puntos.

Bibliografía

[1] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In Conference on Computer Vision and Pattern Recognition (CVPR), 2012.

[2] Lang, A., Vora, S., Caesar, H., Zhou, L., Yang, J., and Beijbom, O. PointPillars: Fast Encoders for Object Detection From Point Clouds. (pp. 12689-12697). arXiv:1812.05784, 2019.

[3] M. Engelcke, D. Rao, D. Z.Wang, C. H. Tong, and I. Posner. Vote3deep: Fast object detection in 3d point clouds using efficient convolutional neural networks. In ICRA, 2017.

[4] J. Ku, M. Mozifian, J. Lee, A. Harakeh, and S. Waslander. Joint 3d proposal generation and object detection from view aggregation. In IROS, 2018.

[5] Y. Zhou and O. Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In CVPR, 2018.

[6] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In CVPR, 2017.

[7] Y. Yan, Y. Mao, and B. Li. SECOND: Sparsely embedded convolutional detection. Sensors, 18(10), 2018.

[8] J. Ku, M. Mozifian, J. Lee, A. Harakeh, and S. Waslander. Joint 3d proposal generation and object detection from view aggregation. In IROS, 2018.

[9] M. Simon, S. Milz, K. Amende, and H.-M. Gross. Complex-YOLO: Real-time 3d object detection on point clouds. arXiv:1803.06199, 2018.

[10] B. Yang, W. Luo, and R. Urtasun. PIXOR: Real-time 3d object detection from point clouds. In CVPR, 2018.

[11] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia. Multi-view 3d object detection network for autonomous driving. In CVPR, 2017.

[12] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. CoRR, abs/1502.03167, 2015

[13] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In ICML, 2010.

[14] IBM Cloud Education (2020), Convolutional Neural Networks. URL: https://www.ibm.com/cloud/learn/convolutional-neural-networks

[15] Krizhevsky, A., Sutskever, I., and Hinton, G. ImageNet Classification with Deep Convolutional Neural Networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (pp. 1097–1105). Curran Associates Inc. 2012.

[16] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), p.770-778. 2016.

[17] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. SSD: Single shot multibox detector. In ECCV, 2016.

[18] Tyagi A. (2020). PointPillars. URL: https://github.com/tyagi-iiitv/PointPillars

[19] Tyagi A. (2020), Implementing Point Pillars in TensorFlow. URL: https://towardsdatascience.com/implementing-point-pillars-in-tensorflow-c38d10e9286

[20] Open3D (2022), A Modern Library for 3D Data Processing. URL: http://www.open3d.org

ANEXO I

network.py

```
import tensorflow as tf
import numpy as np
from config import Parameters


def build_point_pillar_graph(params: Parameters):

    # extract required parameters
    max_pillars = int(params.max_pillars)
    max_points  = int(params.max_points_per_pillar)
    nb_features = int(params.nb_features)
    nb_channels = int(params.nb_channels)
    batch_size  = int(params.batch_size)
    n_gpus      = int(params.n_gpus)
    image_size  = tuple([params.Xn, params.Yn])
    nb_classes  = int(params.nb_classes)
    nb_anchors  = len(params.anchor_dims)

    if tf.keras.backend.image_data_format() == "channels_first":
        raise NotImplementedError
    else:
        input_shape = (max_pillars, max_points, nb_features)

    input_pillars = tf.keras.layers.Input(input_shape, batch_size=batch_size
, name="pillars/input")
    input_indices = tf.keras.layers.Input((max_pillars, 3), batch_size=batch
_size, name="pillars/indices",
                                          dtype=tf.int32)

    def correct_batch_indices(tensor, batch_size):
        array = np.zeros((batch_size, max_pillars, 3), dtype=np.float32)
        for i in range(batch_size):
            array[i, :, 0] = i
        return tensor + tf.constant(array, dtype=tf.int32)

    if batch_size > 1:
        corrected_indices = tf.keras.layers.Lambda(lambda t: correct_batch_i
ndices(t, int(batch_size/n_gpus)))(input_indices)
    else:
        corrected_indices = input_indices

    # pillars
    x = tf.keras.layers.Conv2D(nb_channels, (1, 1), activation='linear', use
_bias=False, name="pillars/conv2d")(input_pillars)
    x = tf.keras.layers.BatchNormalization(name="pillars/batchnorm", fused=T
```

```
rue, epsilon=1e-3, momentum=0.99)(x)
    x = tf.keras.layers.Activation("relu", name="pillars/relu")(x)
    x = tf.keras.layers.MaxPool2D((1, max_points), name="pillars/maxpooling2
d")(x)

    if tf.keras.backend.image_data_format() == "channels_first":
        reshape_shape = (nb_channels, max_pillars)
    else:
        reshape_shape = (max_pillars, nb_channels)

    x = tf.keras.layers.Reshape(reshape_shape, name="pillars/reshape")(x)
    pillars = tf.keras.layers.Lambda(lambda inp: tf.scatter_nd(inp[0], inp[1
],
                                                               (batch_size,)
 + image_size + (nb_channels,)),
                                     name="pillars/scatter_nd")([corrected_i
ndices, x])

    # 2d cnn backbone

    # Block1(S, 4, C)
    x = pillars
    for n in range(4):
        S = (2, 2) if n == 0 else (1, 1)
        x = tf.keras.layers.Conv2D(nb_channels, (3, 3), strides=S, padding="
same", activation="relu",
                                   name="cnn/block1/conv2d%i" % n)(x)
        x = tf.keras.layers.BatchNormalization(name="cnn/block1/bn%i" % n, f
used=True)(x)
    x1 = x

    # Block2(2S, 6, 2C)
    for n in range(6):
        S = (2, 2) if n == 0 else (1, 1)
        x = tf.keras.layers.Conv2D(2 * nb_channels, (3, 3), strides=S, paddi
ng="same", activation="relu",
                                   name="cnn/block2/conv2d%i" % n)(x)
        x = tf.keras.layers.BatchNormalization(name="cnn/block2/bn%i" % n, f
used=True)(x)
    x2 = x

    # Block3(4S, 6, 4C)
    for n in range(6):
        S = (2, 2) if n == 0 else (1, 1)
        x = tf.keras.layers.Conv2D(2 * nb_channels, (3, 3), strides=S, paddi
ng="same", activation="relu",
                                   name="cnn/block3/conv2d%i" % n)(x)
        x = tf.keras.layers.BatchNormalization(name="cnn/block3/bn%i" % n, f
used=True)(x)
    x3 = x

    # Up1 (S, S, 2C)
    up1 = tf.keras.layers.Conv2DTranspose(2 * nb_channels, (3, 3), strides=(
1, 1), padding="same", activation="relu",
```

```
                                            name="cnn/up1/conv2dt")(x1)
    up1 = tf.keras.layers.BatchNormalization(name="cnn/up1/bn", fused=True)(
up1)

    # Up2 (2S, S, 2C)
    up2 = tf.keras.layers.Conv2DTranspose(2 * nb_channels, (3, 3), strides=(
2, 2), padding="same", activation="relu",
                                            name="cnn/up2/conv2dt")(x2)
    up2 = tf.keras.layers.BatchNormalization(name="cnn/up2/bn", fused=True)(
up2)

    # Up3 (4S, S, 2C)
    up3 = tf.keras.layers.Conv2DTranspose(2 * nb_channels, (3, 3), strides=(
4, 4), padding="same", activation="relu",
                                            name="cnn/up3/conv2dt")(x3)
    up3 = tf.keras.layers.BatchNormalization(name="cnn/up3/bn", fused=True)(
up3)

    # Concat
    concat = tf.keras.layers.Concatenate(name="cnn/concatenate")([up1, up2,
up3])

    # Detection head
    occ = tf.keras.layers.Conv2D(nb_anchors, (1, 1), name="occupancy/conv2d"
, activation="sigmoid")(concat)

    loc = tf.keras.layers.Conv2D(nb_anchors * 3, (1, 1), name="loc/conv2d",
kernel_initializer=tf.keras.initializers.TruncatedNormal(0, 0.001))(concat)
    loc = tf.keras.layers.Reshape(tuple(i//2 for i in image_size) + (nb_anch
ors, 3), name="loc/reshape")(loc)

    size = tf.keras.layers.Conv2D(nb_anchors * 3, (1, 1), name="size/conv2d"
, kernel_initializer=tf.keras.initializers.TruncatedNormal(0, 0.001))(concat
)
    size = tf.keras.layers.Reshape(tuple(i//2 for i in image_size) + (nb_anc
hors, 3), name="size/reshape")(size)

    angle = tf.keras.layers.Conv2D(nb_anchors, (1, 1), name="angle/conv2d")(
concat)

    heading = tf.keras.layers.Conv2D(nb_anchors, (1, 1), name="heading/conv2
d", activation="sigmoid")(concat)

    clf = tf.keras.layers.Conv2D(nb_anchors * nb_classes, (1, 1), name="clf/
conv2d")(concat)
    clf = tf.keras.layers.Reshape(tuple(i // 2 for i in image_size) + (nb_an
chors, nb_classes), name="clf/reshape")(clf)

    pillar_net = tf.keras.models.Model([input_pillars, input_indices], [occ,
 loc, size, angle, heading, clf])
#     print(pillar_net.summary())

    return pillar_net
```

inference_utils.py

```
import numpy as np
import cv2 as cv
from typing import List
from config import Parameters
from readers import DataReader
from processors import DataProcessor


                                        .
                                        .
                                        .


def matching_boxes(pred_boxes, gt_boxes, min_dist_param=0.5):
    data = [[], [], [], []]  # prediction matches, ground truth matches, unm
atched prediction, unmatched ground truth
    for pred in pred_boxes:
        bbcentroid = [pred.x, pred.y, pred.z]
        bbsize = [pred.height, pred.length, pred.width]
        match_flag = 0
        for gt in gt_boxes:
            # distances between centroids from ground truth and predicted bo
xes
            centroid_dist = np.linalg.norm(np.array(bbcentroid)-
np.array(gt.centroid))
            size_mean = sum(bbsize)/len(bbsize)
            if centroid_dist/size_mean < min_dist_param:
                match_flag = 1
                data[0].append(pred)
                data[1].append(gt)
                gt_boxes.remove(gt)
                break
        if match_flag == 0:
            data[2].append(pred)
    if len(gt_boxes) > 0:
        for gt in gt_boxes:
            data[3].append(gt)
    return data
```

point_pillars_evaluation.py

```
import os
import time
import numpy as np
import pickle
import tensorflow as tf
from glob import glob

from config import Parameters
from inference_utils import generate_bboxes_from_pred, GroundTruthGenerator,
 rotational_nms, matching_boxes
from network import build_point_pillar_graph
from processors import SimpleDataGenerator
from readers import KittiDataReader

DATA_ROOT = "/home/arvc/3D-PC/KITTI/training"  # TODO make main arg
MODEL_ROOT = "./logs"
EVAL_ROOT = MODEL_ROOT+"/eval"


def matching_extraction(n_pcl, occ_threshold):

    os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
    os.environ["CUDA_VISIBLE_DEVICES"] = "0, 1, 2, 3"
    os.environ["TF_GPU_ALLOCATOR"] = "cuda_malloc_async"

    devices = tf.config.experimental.list_physical_devices('GPU')
    for d in range(len(devices)):
        tf.config.experimental.set_memory_growth(devices[d], True)

    params = Parameters()

    n_gpus = int(params.n_gpus)
    devices = tf.config.experimental.list_physical_devices('GPU')
    devices_names = [d.name.split("e:")[1] for d in devices]
    strategy = tf.distribute.MirroredStrategy(
        devices=devices_names[:n_gpus])  # cross_device_ops=tf.distribute.Hi
erarchicalCopyAllReduce()))

    with strategy.scope():
        pillar_net = build_point_pillar_graph(params)
        pillar_net.load_weights(os.path.join(MODEL_ROOT, "model.h5"))

    data_reader = KittiDataReader()

    lidar_files = sorted(glob(os.path.join(DATA_ROOT, "velodyne", n_pcl + "*
.bin")))
    label_files = sorted(glob(os.path.join(DATA_ROOT, "label_2", n_pcl + "*.
txt")))
    calibration_files = sorted(glob(os.path.join(DATA_ROOT, "calib", n_pcl +
 "*.txt")))
    assert len(lidar_files) == len(label_files) == len(calibration_files), "
```

```
Input dirs require equal number of files."
    eval_gen = SimpleDataGenerator(data_reader, params.batch_size, lidar_fil
es, label_files, calibration_files)
    occupancy, position, size, angle, heading, classification = pillar_net.p
redict(eval_gen,

       batch_size=params.batch_size)

    set_boxes, confidences = [], []
    loop_range = occupancy.shape[0] if len(occupancy.shape) == 4 else 1
    for i in range(loop_range):
        set_boxes.append(generate_bboxes_from_pred(occupancy[i], position[i]
, size[i], angle[i], heading[i],
                                                   classification[i], params
.anchor_dims, occ_threshold))
        confidences.append([float(boxes.conf) for boxes in set_boxes[-1]])

    # NMS
    min_threshold = np.amin(occ_threshold)
    if min_threshold > 0.6:
        min_threshold = 0.6
    nms_boxes = rotational_nms(set_boxes, confidences, occ_threshold=min_thr
eshold, nms_iou_thr=min_threshold)

    gt_gen = GroundTruthGenerator(data_reader, label_files, calibration_file
s, network_format=False)

    matching_data = []
    for seq_boxes, gt_label in zip(nms_boxes, gt_gen):
        matching_data.append(matching_boxes(seq_boxes, gt_label, min_dist_pa
ram=0.5))

    return matching_data


if __name__ == "__main__":
    step = 0.02
    occupancy_thresholds = np.arange(0.08, 1, step)
    for occ in occupancy_thresholds:
        evaluation_data = []
        for n in range(75):
            pcl = str(n).zfill(4)
            print("Current pcl:", n*100)
            evaluation_data.extend(matching_extraction(pcl, [occ, occ, occ,
occ]))
        with open(EVAL_ROOT+"/matchings_ot"+str(round(occ*100)).zfill(2), "w
b") as fp:
            pickle.dump(evaluation_data, fp)
```

match_data.py

```
import numpy as np
import os
import pickle
import matplotlib.pyplot as plt
import sklearn.metrics as skl
from config import Parameters

dirname = "eval_backup"
savedir = "/home/arvc/3D-PC/TFM/Figures/"+dirname
EVAL_ROOT = "./logs/"+dirname


if __name__ == "__main__":
    if not os.path.exists(savedir):
        os.mkdir(savedir)
    step = 0.02
    occupancy_thresholds = np.arange(0.08, 1.0, step)
    recall = []
    precis = []
    occ_mean_TP = []
    occ_mean_FP = []
    occ_mean = []
    conf_mat = []
    for occ_threshold in occupancy_thresholds:
        print(round(occ_threshold*100))
        with open(EVAL_ROOT+"/matchings_ot"+str(round(occ_threshold*100)).zf
ill(2), "rb") as fp:
            evaluation_data = pickle.load(fp)
        match_conf = [[], [], [], []]
        match_dist = [[], [], [], []]
        unmatch_conf = [[], [], [], []]
        unmatch_dist = [[], [], [], []]
        match_pred = []
        match_true = []
        true_neg = 0
        for pcl in evaluation_data:
            for box in pcl[0]:
                match_conf[box.cls].append(box.conf)
                match_pred.append(box.cls)
            for box in pcl[1]:
                dist = np.linalg.norm([box.centroid[0], box.centroid[1]])
                match_dist[Parameters.classes[box.classification]].append(di
st)
                match_true.append(Parameters.classes[box.classification])
            for box in pcl[2]:
                unmatch_conf[box.cls].append(box.conf)
            for box in pcl[3]:
                dist = np.linalg.norm([box.centroid[0], box.centroid[1]])
                unmatch_dist[Parameters.classes[box.classification]].append(
dist)
            if len(pcl[2]) == 0 and len(pcl[3]) == 0:
```

```
                true_neg += 1

        # Confusion Matrix
        cm = skl.confusion_matrix(match_true, match_pred, labels=[0, 1, 2, 3
])
        cm_list = cm.tolist()
        for j in range(4):
            cm_list[j].append(len(unmatch_dist[j]))
        cm_list.append([len(unmatch_conf[0]), len(unmatch_conf[1]), len(unma
tch_conf[2]), len(unmatch_conf[3]),
                        true_neg])
        conf_mat.append(np.array(cm_list))

        # Recall and precision
        newR = []
        newP = []
        for i in range(4):
            if sum(cm_list[i]) != 0:
                newP.append(cm_list[i][i]/sum(cm_list[i]))
            else:
                newP.append(0)
            if sum(row[i] for row in cm_list) != 0:
                newR.append(cm_list[i][i]/sum(row[i] for row in cm_list))
            else:
                newR.append(1)
        recall.append(newR)
        precis.append(newP)
        occ_mean_TP.append([np.mean(match_conf[0]), np.mean(match_conf[1]),
                            np.mean(match_conf[2]), np.mean(match_conf[3])])
        occ_mean_FP.append([np.mean(unmatch_conf[0]), np.mean(unmatch_conf[1
]),
                            np.mean(unmatch_conf[2]), np.mean(unmatch_conf[3
])])
        occ_mean.append([np.mean(match_conf[0] + unmatch_conf[0]),
                         np.mean(match_conf[1] + unmatch_conf[1]),
                         np.mean(match_conf[2] + unmatch_conf[2]),
                         np.mean(match_conf[3] + unmatch_conf[3])])

    col = ["red", "green", "blue", "black"]
    lab = ["car", "pedestrian", "cyclist", "truck-van-tram-misc"]
    print_precision_recall(recall, precis, occ_mean, col, lab, savedir)
    print_precision_occ_recall_occ(recall, precis, occupancy_thresholds, col
, lab, savedir)
    print_confmatrix_occ(conf_mat, occupancy_thresholds, col, lab, savedir)
    occupancy_threshold = 0.4
    conf_mat_display = skl.ConfusionMatrixDisplay(conf_mat[round((occupancy_
threshold*100-8)/2)]).plot()
    plt.xlabel("True label")
    plt.ylabel("Predicted label")
    plt.savefig(savedir+'/confMatrix_'+str(occupancy_threshold)+'.png', form
at='png')
    plt.show()
```

ANEXO II

Visualizador de training

```
import open3d as o3d
from visualize_utils import convert_bin_to_pcd, bbcenter2corner, bbox_lines
from point_pillars_evaluation import matching_extraction

KITTI_PATH = "/home/arvc/3D-PC/KITTI/training/velodyne/"


def vis_kitti_train(pcl):
    # pcl = 220
    n = str(pcl).zfill(6)
    kitti_plc = KITTI_PATH + str(n) + ".bin"
    pcd = [convert_bin_to_pcd(kitti_plc)]  # Point Cloud with Kitti data

    occ_threshold = [0.38, 0.22, 0.24, 0.26]
    bounding_boxes = matching_extraction(n, occ_threshold)
    bounding_boxes = bounding_boxes[0]

    colors = [[0, 0, 0], [0, 1, 0], [1, 0, 0]]
    lines = []  # Lines from bounding boxes

    # Ground Truth
    for gt in bounding_boxes[1] + bounding_boxes[3]:
        gt_box = [list(gt.centroid) + list(gt.dimension) + [gt.yaw]]
        corner_gt = bbcenter2corner(gt_box[0])
        lines.extend(bbox_lines(corner_gt, colors[0]))

    # True positives
    for pred in bounding_boxes[0]:
        pred_box = [pred.x, pred.y, pred.z, pred.length, pred.width, pred.he
ight, pred.yaw]
        corner_pred = bbcenter2corner(pred_box)
        lines.extend(bbox_lines(corner_pred, colors[1]))

    # False Positives
    for pred in bounding_boxes[2]:
        false_pred_box = [pred.x, pred.y, pred.z, pred.length, pred.width, p
red.height, pred.yaw]
        corner_false_pred = bbcenter2corner(false_pred_box)
        lines.extend(bbox_lines(corner_false_pred, colors[2]))

    # vis = o3d.visualization.Visualizer()
    o3d.visualization.draw_geometries(pcd+lines)
    # vis.run()
```

```
if __name__ == "__main__":
    for i in range(4090, 4100):
        vis_kitti_train(i)
        print(i)
```

Visualizador de testing

```
import open3d as o3d
import numpy as np
from test_prediction import test_pred
from visualize_utils import convert_bin_to_pcd, bbcenter2corner, bbox_lines

KITTI_PATH = "/home/arvc/3D-PC/KITTI/testing/velodyne/"


def vis_kitti_test(pcl):
    # pcl = 1124
    n = str(pcl).zfill(6)
    kitti_plc = KITTI_PATH + str(n) + ".bin"
    pcd = [convert_bin_to_pcd(kitti_plc)]  # Point Cloud with Kitti data

    occ_threshold = [0.38, 0.22, 0.24, 0.26]
    bounding_boxes = test_pred(n, occ_threshold)

    colors = [[1, 0, 0], [0, 1, 0], [1, 0.6, 0], [0, 0, 0]]
    lines = []  # Lines from bounding boxes

    # Predictions
    for pred in bounding_boxes:
        pred_box = [pred.x, pred.y, pred.z, pred.length, pred.width, pred.he
ight, pred.yaw]
        corner_pred = bbcenter2corner(pred_box)
        lines.extend(bbox_lines(corner_pred, colors[pred.cls]))

    # vis = o3d.visualization.Visualizer()
    o3d.visualization.draw_geometries(pcd+lines)
    # vis.run()


if __name__ == "__main__":
    for i in range(500, 510):
        vis_kitti_test(i)
```

visualize_utils.py

```
import open3d as o3d
import numpy as np
import struct


def labeled_pcd_points(pointcloud):
    label_pcd = []
    for point in pointcloud:
        # angle = np.arctan2(point[1], point[0])
        if point[2] >= -
2.2:  # (abs(angle) <= np.pi/4 or (5*np.pi/4 >= abs(angle) >= 3*np.pi/4)) an
d point[2] >= -2.2:
            label_pcd.append(point)
    return label_pcd


def convert_bin_to_pcd(bin_file_path):
    size_float = 4
    list_pcd = []
    with open(bin_file_path, "rb") as f:
        byte = f.read(size_float * 4)
        while byte:
            x, y, z, intensity = struct.unpack("ffff", byte)
            list_pcd.append([x, y, z])
            byte = f.read(size_float * 4)
    labeled_pcd = labeled_pcd_points(list_pcd)
    np_pcd = np.asarray(labeled_pcd)
    pc = o3d.geometry.PointCloud()
    pc.points = o3d.utility.Vector3dVector(np_pcd)
    return pc


def bbcenter2corner(box):
    translation = box[0:3]
    l, w, h = box[3], box[4], box[5]
    rotation = box[6]

    # Create a bounding box outline
    bounding_box = np.array([
        [-l/2, -l/2, l/2, l/2, -l/2, -l/2, l/2, l/2],
        [w/2, -w/2, -w/2, w/2, w/2, -w/2, -w/2, w/2],
        [-h/2, -h/2, -h/2, -h/2, h/2, h/2, h/2, h/2]])

    # Standard 3x3 rotation matrix around the Z axis
    rotation_matrix = np.array([
        [np.cos(rotation), -np.sin(rotation), 0.0],
        [np.sin(rotation), np.cos(rotation), 0.0],
        [0.0, 0.0, 1.0]])

    # Repeat the [x, y, z] eight times
    eight_points = np.tile(translation, (8, 1))
```

```
    # Translate the rotated bounding box by the original center position to
obtain the final box
    corner_box = np.dot(
        rotation_matrix, bounding_box) + eight_points.transpose()

    return corner_box.transpose()


def bbox_lines(box, color):
    # Our lines span from points 0 to 1, 1 to 2, 2 to 3, etc...
    corners_conection = [[0, 1], [1, 2], [2, 3], [0, 3],
                         [4, 5], [5, 6], [6, 7], [4, 7],
                         [0, 4], [1, 5], [2, 6], [3, 7]]

    # Use the same color for all lines
    colors = [color for _ in range(len(corners_conection))]

    # line_set = o3d.geometry.LineSet()
    # line_set.points = o3d.utility.Vector3dVector(box)
    # line_set.lines = o3d.utility.Vector2iVector(corners_conection)
    # line_set.colors = o3d.utility.Vector3dVector(colors)
    line_mesh = LineMesh(box, corners_conection, colors, radius=0.05)
    line_mesh_geoms = line_mesh.cylinder_segments

    return line_mesh_geoms


def align_vector_to_another(a=np.array([0, 0, 1]), b=np.array([1, 0, 0])):
    """Aligns vector a to vector b with axis angle rotation"""
    if np.array_equal(a, b):
        return None, None
    axis_ = np.cross(a, b)
    axis_ = axis_ / np.linalg.norm(axis_)
    angle = np.arccos(np.dot(a, b))

    return axis_, angle


def normalized(a, axis=-1, order=2):
    """Normalizes a numpy array of points"""
    l2 = np.atleast_1d(np.linalg.norm(a, order, axis))
    l2[l2 == 0] = 1
    return a / np.expand_dims(l2, axis), l2


class LineMesh(object):
    def __init__(self, points, lines=None, colors=[0, 1, 0], radius=0.15):
        """Creates a line represented as sequence of cylinder triangular mes
hes

        Arguments:
            points {ndarray} -- Numpy array of ponts Nx3.

        Keyword Arguments:
```

```
            lines {list[list] or None} --
 List of point index pairs denoting line segments.
            If None, implicit lines from ordered pairwise points. (default:
{None})
            colors {list} --
 list of colors, or single color of the line (default: {[0, 1, 0]})
            radius {float} -- radius of cylinder (default: {0.15})
        """
        self.points = np.array(points)
        self.lines = np.array(
            lines) if lines is not None else self.lines_from_ordered_points(
self.points)
        self.colors = np.array(colors)
        self.radius = radius
        self.cylinder_segments = []

        self.create_line_mesh()

    @staticmethod
    def lines_from_ordered_points(points):
        lines = [[i, i + 1] for i in range(0, points.shape[0] - 1, 1)]
        return np.array(lines)

    def create_line_mesh(self):
        first_points = self.points[self.lines[:, 0], :]
        second_points = self.points[self.lines[:, 1], :]
        line_segments = second_points - first_points
        line_segments_unit, line_lengths = normalized(line_segments)

        z_axis = np.array([0, 0, 1])
        # Create triangular mesh cylinder segments of line
        for i in range(line_segments_unit.shape[0]):
            line_segment = line_segments_unit[i, :]
            line_length = line_lengths[i]
            # get axis angle rotation to allign cylinder with line segment
            axis, angle = align_vector_to_another(z_axis, line_segment)
            # Get translation vector
            translation = first_points[i, :] + line_segment * line_length *
0.5
            # create cylinder and apply transformations
            cylinder_segment = o3d.geometry.TriangleMesh.create_cylinder(sel
f.radius, line_length)
            cylinder_segment = cylinder_segment.translate(translation, relat
ive=False)
            if axis is not None:
                axis_a = axis * angle
                cylinder_segment = cylinder_segment.rotate(R=o3d.geometry.ge
t_rotation_matrix_from_axis_angle(axis_a),
                                                           center=cylinder_s
egment.get_center())
                # cylinder_segment = cylinder_segment.rotate(axis_a, center=
True,
                #                                            type=o3d.geomet
ry.RotationType.AxisAngle)
```

```
            # color cylinder
            color = self.colors if self.colors.ndim == 1 else self.colors[i,
:]
            cylinder_segment.paint_uniform_color(color)

            self.cylinder_segments.append(cylinder_segment)

    def add_line(self, vis):
        """Adds this line to the visualizer"""
        for cylinder in self.cylinder_segments:
            vis.add_geometry(cylinder)

    def remove_line(self, vis):
        """Removes this line from the visualizer"""
        for cylinder in self.cylinder_segments:
            vis.remove_geometry(cylinder)
```