Desarrollo de aplicación basada en CNN para algoritmos de visión en coches autónomos

Developing CNN-based applications for vision algorithms in autonomous cars

Trabajo de fin de grado en Ingeniería de Computadores

Estudiantes:

Marco Expósito Pérez

Ioan Marian Tulai

Director:

Carlos García Sánchez

Universidad Complutense de Madrid
Facultad de Informática

Curso académico 2021-22

Resumen

La conducción autónoma es casi un medio en la actualidad gracias a la mejora de las tecnologías actuales y a la potencia de computación. Entre estas tecnologías se encuentran las redes neuronales, (CNN en inglés), especializadas en la detección y clasificación de objetos en imágenes y sobre las que trata este proyecto.

Este trabajo analiza el comportamiento de redes neuronales convolucionales a la hora de detectar los principales objetos en la vía pública, como coches, peatones o señales de tráfico. Se analizarán aspectos como la **velocidad de inferencia** y la **precisión** en **distinto hardware** con diferentes características computacionales.

Para ello se reentranará una CNN generalista con distintos datasets especializados en conducción autónoma. Esta red además se va a optimizar para buscar los mejores resultados posibles.

Palabras clave

Conducción autónoma, Detección y clasificación de objetos, CNN (Red neuronal convolucional), YOLOv5, BDD100k, NUSCENES, mAP, OpenVINO.

Abstract

Autonomous conduction is close to a medium thanks to the improvement of current technologies and computing potency. CNN (Neural networks) are some of these technologies, specialized in the detection and classification of objects in images and the ones related with this project.

This project analyze the behaviour of CNN when trying to detect the main objects of a public road, like cars, pedestrians o traffic signals. **Inference speed** and **precision** will be analyzed for **different hardware** with distinct computational characteristics.

In order to do it, we will re-train a CNN with different datasets focused on autonomous driving. Additionally, the CNN will be optimized trying to get the best possible results.

Key words

Autonomous driving, Detection and classification of objects, CNN(convolutional neural networks), YOLOv5, BDD100k, NUSCENES, mAP, OpenVINO.

Contents

1 Introducción

La conducción autónoma es una idea que poco a poco se está convirtiendo en realidad. En el proceso, se han implementado distintas tecnologías para dirigir un vehículo por una vía sin necesidad de supervisión o control humano. Una de estas tecnologías implica integrar una cámara y su procesador en el vehículo para poder reconocer obstáculos, caminos o elementos que alteren el comportamiento del vehículo. Otra alternativa es enviar la información de la cámara a la nube y realizar ahí su procesado, pero los tiempos de latencia, problemas de conectividad y debido a la necesidad de resultados en tiempo real lo descartamos.

Por ejemplo, debe ser capaz de reconocer una carretera para conducir a través, de identificar peatones para evitar atropellos y reconocer señales de tráfico para poder seguir las normas de circulación en tiempo real.

Aunque la conducción autónoma puede parecer una idea del siglo actual, el primer diseño de un coche autónomo [1] data de 1939. Norman Del Geddes ideó un coche con sensores capaces de detectar un circuito eléctrico embebido en la carretera, el cual era el responsable de dirigir al vehículo.

El primer diseño más relacionado con el contenido de este trabajo es el diseño de Erns Dickmans [2], quien junto a Mercedes-benz en 1987, crearon la primera furgoneta capaz de conducir por una vía vacía sin salirse de ella. Este diseño utilizaba cámaras capaces de identificar las líneas de la carretera y dirigir el vehículo basándose en la posición de la furgoneta respecto a las líneas.

1.1 Motivación y propósito

El objetivo de este proyecto es evaluar los resultados de distintos modelos de YOLOv5 [3]. La decisión de usar la YOLOv5 está motivada por ser una de las redes neuronales para detección de objetos más usada en el mundo y que hemos escogido al ser la última versión del algoritmo YOLO. Una vez obtenidos los resultados, los compararemos entre ellos e intentaremos buscar el modelo ideal

para la detección de objetos durante la conducción autónoma.

Los puntos principales que compararemos serán:

- **Precisión**. Durante la conducción autónoma la precisión es especialmente importante, ya que hay vidas humanas en juego, tanto de peatones como de los pasajeros del vehículo.
- **Velocidad de inferencia**. Un vehículo se mueve a una gran velocidad, así que es necesario que sea rápido para poder reaccionar a los estímulos a tiempo.

Los pasos que se van a seguir en el trabajo son los siguientes:

1. Obtener varios datasets dedicados a la detección de objetos durante la conducción autónoma y transformarlos a un formato que entienda la YOLOv5. Queremos utilizar distintos datasets para evitar posibles sesgos en uno de los datasets.
2. Reentrenar modelos de la YOLOv5 con estos datasets. La YOLOv5 ya está entrenada de base con el dataset COCO, un dataset generalista para detección de objetos. Mediante el reentrenamiento buscamos convertir YOLOv5 en una red neuronal más centrada en la detección de objetos para la conducción autónoma.
3. Optimizar todos los modelos obtenidos y obtener los resultados finales para distinto hardware objetivo.
4. Comparar los resultados y concluir que modelo y hardware es el más adecuado para integrar en un coche autónomo.

Para llevar a cabo estos pasos, se han utilizado distintas herramientas y frameworks:

- **Pytorch** [4]. Empleado para el reentrenamiento de los modelos.
- **Netron** [5]. Utilizado para visualizar los distintos nodos de la red neuronal.

- **OpenVINO** [6]. Utilizado para optimizar la velocidad de inferencia y obtener las medidas resultadas en el distinto hardware.
- **Wandb** [7]. Utilizado para visualizar datos del entrenamiento de los modelos durante el reentrenamiento.
- **Docker** [29]. Utilizado para ejecutar el workbench de OpenVINO en local.

Las redes neuronales dedicadas a la visión artificial están en pleno auge, cada año surgen nuevos modelos. Un ejemplo destacado de esto es la propia YOLOv5 que se publicó un mes después de que se publicase YOLOv4. El problema de este desarrollo acelerado es que las tecnologías muchas veces emplean distintos formatos, distintas versiones de una misma librería e incluso frameworks completamente diferentes al no existir un estándar claro.

2 Introduction

Self driving is an idea that step by step is becoming reality. During the process, a lot of different technologies have been implemented to lead a vehicle through a road without direct human control. One of this technologies imply to integrate a camera and its processor in the vehicle to recognize obstacles, roads and elements that modify the behaviour of the vehicle. Another alternative is to send the information of the camera to the cloud and process its data there but because of the high latency, connectivity problems and the necessity of results in real time we disregard this option.

For example, it must be capable of recognizing a road to drive through it, identify pedestrian to avoid run overs and see traffic signals to be able to follow the traffic rules in real time.

Despite of the autonomous driving may look like a new idea from current century, the first design of a self driving car [1] dates from 1939. Norman Del Geddes devised a car with sensors capable of detecting an electric circuit embedded in the road. This circuit was the responsible of directing the car.

The first design more related to the content of this project is Erns Dickmans's [2] design, who with Mercedes-benz help in 1987, created the first van capable of driving through an empty road without going out, This design used cameras capable of identifying the road's limiting lines and lead the vehicle between them.

2.1 Motivation and purpouse

The objective of this project is to evaluate the results of different YOLOv5's models [3]. The decision of using YOLOv5 is motivated because its one of the CNN for object detection most used in the world and because the version 5 is the last release of the YOLO algorithm. One we have obtained the results, we will compare them and we will try to find the ideal YOLOv5 model for object detection for autonomous driving.

The main points we will compare are:

- **Precision**. Precision is a measurement specially important in autonomous driving because there are human lives in danger, both pedestrian and the passengers of the vehicle.
- **Inference speed**. Vehicles move at a high speed, so its necessary that the inference speed is high enough to be able to react to the stimulus on time.

The steps we will follow in the project are the following:

1. Obtain different datasets dedicated to object detection and autonomous driving and translate them to YOLOv5's format. We want to use different datasets to avoid possible biases in one of the datasets.
2. Retrain different YOLOv5's models with these datasets. YOLOv5 is already trained with COCO dataset, a generic dataset for generic object detection. By the means of retraining it we are looking for transforming YOLOv5 into a CNN more focused on detecting objects during autonomous driving.
3. Optimize all trained models and get the final results doing inferences in different objective hardware.
4. Compare the final results and conclude what model and hardware is the most appropriate to integrate in a self driving vehicle.

In order to follow these steps we have used different tools and frameworks:

- **Pytorch** [4]. Used to retrain the models.
- **Netron** [5]. Used to visualize the different nodes of each CNN.
- **OpenVINO** [6]. Used to optimize the inference speed and obtain the final measurements in different hardware.
- **Wandb** [7]. Used to visualize training data while the YOLOv5 model is being trained.

- **Docker** [29]. Used to run OpenVINO's workbench in a local machine.

Neural networks dedicated to computer visions currently are in a boom, ever year new models are implemented. A clear example of this is YOLOv5 that was published one month after the YOLOv4 release. The problem of this accelerated progress is that each technology use different formats, different versions of the same library or even completely different frameworks because there is not a clear standard.

3 Bases teoricas del proyecto

3.1 ¿Qué es una red neuronal?

Una red neuronal [8] es un conjunto de nodos llamados neuronas, agrupados en capas que están interconectadas (figura 1). La capa inicial recibe datos del exterior y la capa de salida devuelve los valores de salida o resultado. Además, puede existir un número indeterminado de capas intermedias llamadas capas ocultas sin ninguna conexión al exterior de la red.

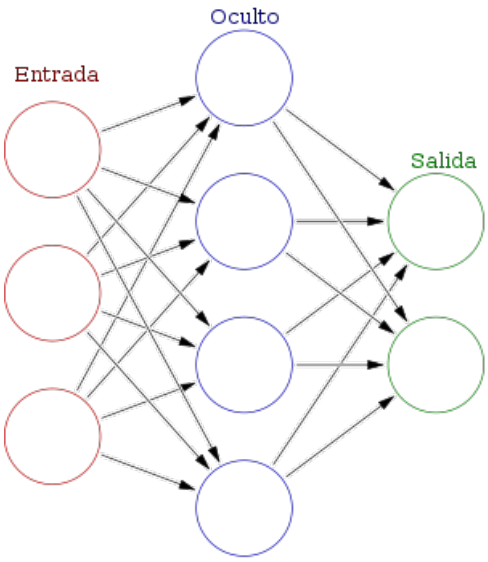

Figure 1: Abstracción de una red neuronal. Extraído de [9]

Cada neurona tiene un conjunto de valores de entrada cuyo valor se modifica en función del peso asignado a su entrada. Posteriormente, se aplica una función de activación para calcular el valor de salida de la neurona. Véase en la figura 2.

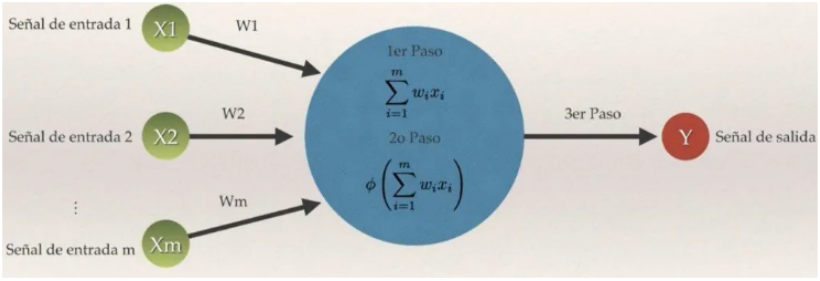

Figure 2: Neurona con varias entradas. Extraído de [8]

La función de activación es una función matemática que interpreta los valores de entrada y los utiliza para obtener el valor de salida del nodo. Un ejemplo sencillo de función de entrada es la función escalón (figura 3), que emite un 1 si la entrada es positiva, o un 0 si es negativa.

La función de activación no tiene porque siempre ser la misma para todos los nodos. Por ejemplo, en la YOLOv5 se utiliza la función Leaky ReLU (figura 4) en las capas ocultas iniciales y la función sigmoide (figura 5) en las finales. La función ReLU (figura 6) es otra función bastante utilizada de donde proviene la Leaky ReLU.

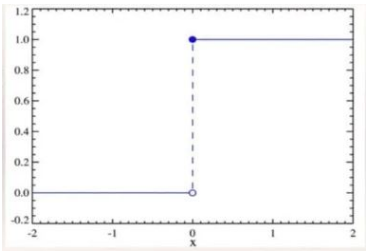

Figure 3: Función escalón. Extraído de [8]

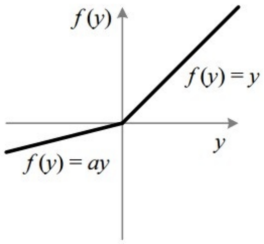

Figure 4: Función Leaky ReLU. Extraído de [32]

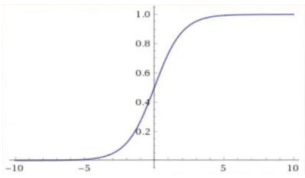

Figure 5: Función sigmoide. Extraído de [8]

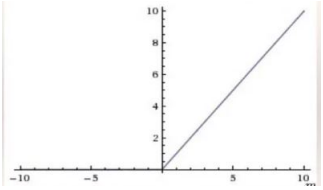

Figure 6: Función ReLu. Extraído de [8]

3.2 Entrenar una red neuronal

El entrenamiento de una red neuronal es una búsqueda de los mejores pesos para cada entrada de cada neurona. Los mejores pesos son aquellos que producen resultados suficientemente buenos para resolver un problema específico.

La búsqueda de los pesos adecuados depende del algoritmo concreto que utilice la red neuronal, pero todos siguen un proceso similar:

1. Se introduce un objeto de entrada. En el caso de un detector de objetos, la entrada sería una imagen y cada neurona recibiría un píxel de la imagen.
2. Las neuronas de la red neuronal reciben la entrada, aplican su función de activación y dan una salida. Esto se repite hasta que la capa de salida tiene los resultados de la inferencia.
3. Compara los resultados obtenidos con los resultados reales. En el caso del detector de imágenes, compara donde ha detectado un objeto la red neuronal y donde está el objeto en realidad.
4. En función de los resultados y el algoritmo de aprendizaje, todos los pesos de la red neuronal se ajustan para mejorar el resultado obtenido.
5. Se repiten todos los pasos con otro objeto de entrada hasta haber aprendido de todos los objetos.

Una vez se ha aprendido de todos los objetos, se dice que ha ocurrido o que se ha recorrido una época. El entrenamiento se termina cuando se cumplan las épocas indicadas en la configuración inicial.

Durante el entrenamiento puede ocurrir que la red acierte el 100% de los objetos de entrada. En estos casos se dice que la red está sobre-aprendiendo y puede provocar problemas a la hora de detectar objetos que no estén en el conjunto de objetos de entrada. Existe otro conjunto de técnicas y algoritmos paralelos al entrenamiento para evitar estos sucesos, como eliminar aleatoriamente algunas de las neuronas de la red o cambiar aleatoriamente algunos pesos.

3.3 Redes CNN

Las redes neuronales convolucionales [10], son conocidas por su desempeño en la clasificación de objetos, sacan las características de los objetos y los identifican con mayor rendimiento que otros modelos.

Las redes CNN procesan imágenes en 3 fases diferenciadas:

1. Se leen los píxeles de la imagen. Si la imagen es a color, cada píxel se divide en los diferentes colores y se normalizan los valores entre 0 y 1.

2. Se aplican múltiples convoluciones.

3. Los píxeles atraviesan la red neuronal multicapa donde se extraen las características extraídas de las convoluciones y se detecta el objeto.

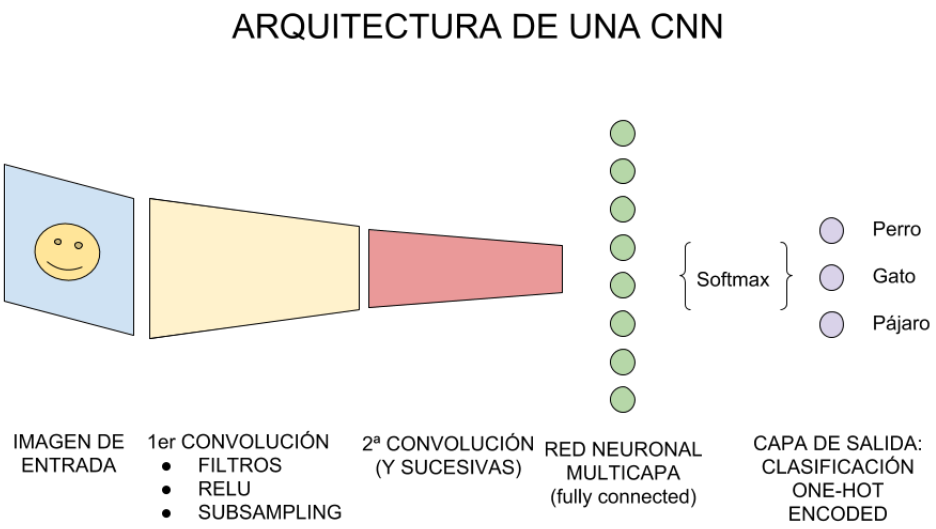

Figure 7: Arquitectura de una red neuronal convolucional. Extraído de [10]

3.3.1 Convoluciones

Las convoluciones [11] son el proceso que da nombre a las redes neuronales convolucionales y funcionan de la siguiente forma:

Primero se crea o se define un Kernel. El Kernel es una matriz cuyo contenido y tamaño va a determinar la transformación producida en el proceso de convolución.

El Kernel recorre todos los píxeles de la imagen aplicando la función del kernel al píxel principal y a sus adyacentes. Normalmente, se evita centrarse en un píxel cuyos píxeles adyacentes no formen una matriz del mismo tamaño que el kernel, ya que puede dar resultados conflictivos.

En la figura 8 el kernel está centrado en el 1 del recuadro rojo mientras también opera con todos los de su alrededor. En este caso, nunca se centraría en la primera fila de ceros, porque no sería posible formar una matriz de 3x3 mientras se centra en ellos.

Al aplicar el kernel a un grupo de píxeles, se aplica una función a todos ellos para obtener un único valor para cada pixel de toda la imagen. En el caso de la figura 8, vemos como el resultado de la matriz azul en ”source pixel” y el kernel ”convolution” es -8.

En el caso de las imágenes a color, se aplica un kernel distinto para cada canal de color (rojo, verde y azul) y posteriormente se unen durante el procesamiento.

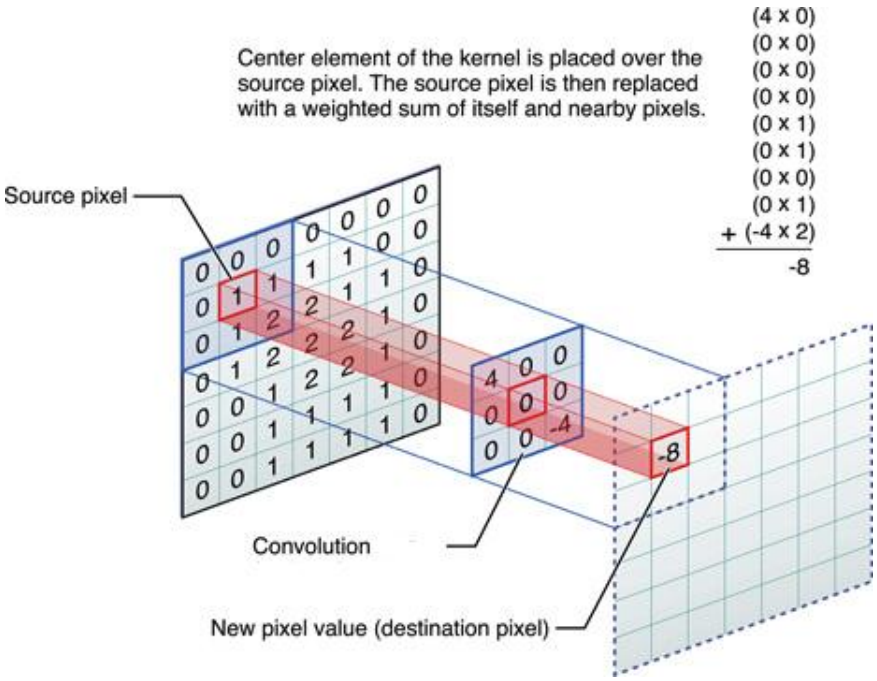

Figure 8: Ejemplo de convolución. Extraído de [11]

3.3.2 Aplicación de función de activación

Una vez aplicado el kernel a toda la imagen, el resultado se manda distintas neuronas y se aplica la función de activación para obtener los resultados.

En la figura 9 podemos observar el proceso completo desde la imagen con los valores de los píxeles normalizados, la convolución del kernel y la aplicación de la función de activación.

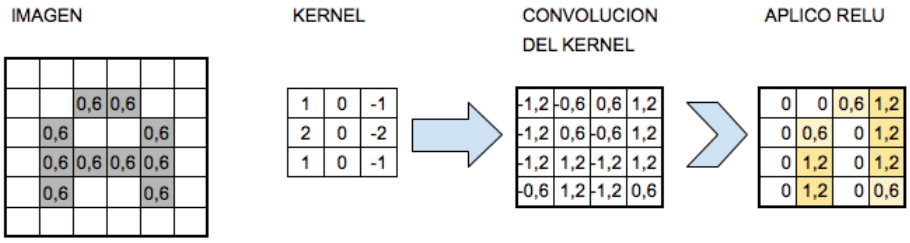

Figure 9: Proceso completo de convolución. Extraído de [10]

3.4 Algoritmos de detección de objetos

Una vez hemos entrenado una red neuronal CNN, le podemos pasar una imagen con un objeto que debería ser capaz de identificar sin mucho problema. ¿Y si la imagen tiene 2 o más objetos, iteramos más veces buscando más objetos?

Esta situación provoca que se itere sobre la imagen con un pequeño recuadro que en cada iteración detecta distintos objetos en distintas áreas de la imagen [13]. Esto nos genera más preguntas:

¿Qué tamaño debe tener el recuadro? ¿Cuánto se debe desplazar en cada iteración? ¿Se ha encontrado el objeto completo o solo una parte del mismo?

Estas cuestiones pueden provocar problemas como la detección del mismo objeto sucesivas veces si el desplazamiento es pequeño, o saltarse objetos si es muy grande. Además, el tamaño y la distancia de desplazamiento afecta directamente al tiempo de inferencia de la CNN.

Estos problemas se resuelven de distinta forma en función del algoritmo de detección de objetos, una de las características que diferencia unas CNN de otras.

3.4.1 R-CNN

El algoritmo R-CNN surge en 2014 [14] y se basa en 3 módulos con diferentes responsabilidades.

1. El primero módulo selecciona en torno a 2000 regiones de interés de diversos tamaños de la imagen. De esta forma se preseleccionan zonas del mismo color o delimitadas por líneas bruscas o cambios de brillo.
2. El segundo módulo es una CNN que extrae una cantidad fija de características de cada región.
3. El tercer y último módulo se compone de una serie de SVM que se encarga de decidir si las regiones pertenecen a un objeto.

El principal problema de este algoritmo es que tarda mucho en entrenarse y en realizar la inferencia debido a las 2000 regiones de interés.

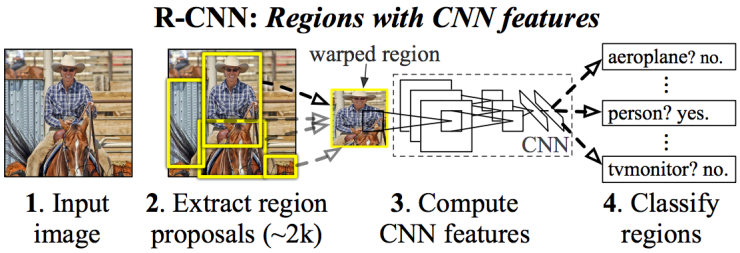

Figure 10: Ejemplo del funcionamiento de R-CNN. Extraído de [13]

Buscando resolver este problema, se crean las Fast R-CNN [16]. Este algoritmo, en vez de preseleccionar 2000 regiones, la CNN directamente recibe la imagen y la divide en regiones. Un nuevo módulo llamado ROI poolling extraerá las características en vectores de tamaño fijo de las regiones obtenidas por la red neuronal.

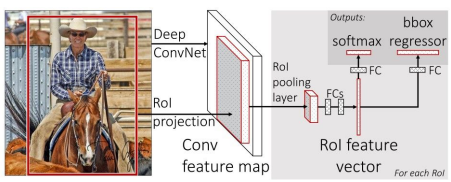

Figure 11: Ejemplo del funcionamiento de fast R-CNN. Extraído de [15]

Posteriormente, se desarrolla la Faster R-CNN [16]. Funcionan de la misma forma que la Fast R-CNN, pero se utiliza una segunda red neuronal para proponer regiones de interés antes de inferir sobre ellas.

Todos los algoritmos de la familia R-CNN se consideran algoritmos de dos etapas, ya que se utiliza un modelo para clasificar los objetos y otro modelo para su detección.

4 YOLOv5 y sus modelos

4.1 YOLO

YOLO (You only look once) es un algoritmo de detección de objetos de una única etapa. Cada imagen se divide en una malla y cada celda de la malla se encarga de detectar y clasificar los objetos dentro de su área. De esta forma solo se recorre la imagen una única vez, de ahí el nombre del algoritmo. Su buen rendimiento y licencia open-source han hecho de este algoritmo uno de los más populares en el ámbito de las redes CNN y la detección de objetos. [18]

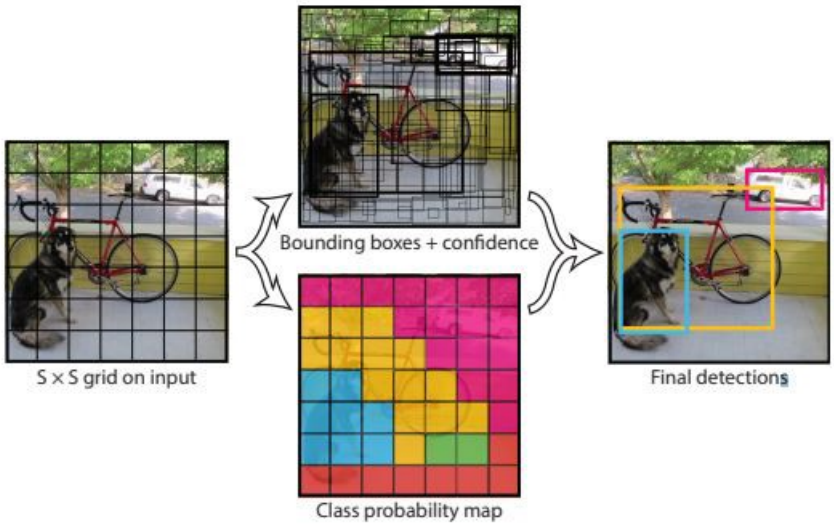

Figure 12: Algoritmo YOLO. Extraído de [15]

YOLO es sustancialmente más rápido que los algoritmos de la familia R-CNN sin perder una precisión notable. Para poner en perspectiva la diferencia de velocidad, en la siguiente gráfica (figura 13) se pueden observar los resultados de faster R-CNN y de YOLO. SSD (Single Shot Detector) es otro algoritmo de una única etapa. En este ejemplo se han entrenado las redes con el dataset de PASCAL VOC 2007 con imágenes de alta y baja resolución. [17]

Teniendo en cuenta las necesidades de velocidad que requiere la conducción autónoma, un algoritmo de dos etapas es demasiado lento. Por tanto, escogimos utilizar uno de los mejores y más populares algoritmos de una etapa, YOLO.

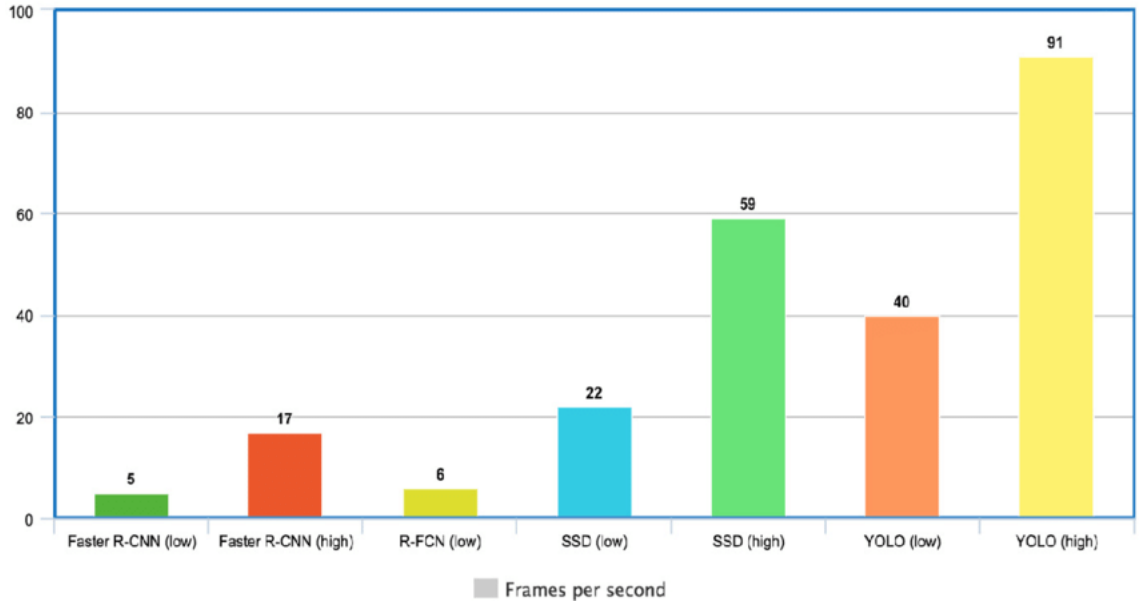

Figure 13: Algoritmos de una etapa vs de dos etapas. Extraído de [17]

4.2 YOLOv5

Como el nombre YOLOv5 sugiere, YOLO es una familia de algoritmos de detección de objetos con varias versiones. La primera versión fue presentada en 2015 y desde entonces se ha seguido actualizando y creando nuevas versiones. Hasta la versión 5, todas las implementaciones estaban escritas en C++, sin embargo, YOLOv5 está escrita en phyton facilitando el manejo de la red neuronal y permitiendo que más personas la utilicen. [3]

YOLOv5 tiene distintos modelos que aunque todos comparten la misma arquitectura y algoritmo, difieren en número de capas y número de neuronas por capa. Esto influye directamente en el peso de la red neuronal, en su velocidad y en su precisión. Todos estos modelos además están pre-entrenados con el dataset COCO que se comentará en la sección 5.2.

El eje vertical de la gráfica (figura 14) indica la precisión del modelo y el horizontal la velocidad. Podemos observar que cuanto más rápido es un modelo, peor precisión tiene. Teniendo en cuenta las diferencias entre los modelos, decidimos utilizar el modelo más pequeño, uno mediano y el más grande, YOLOv5**n**, YOLOv5**m** y YOLOv5**x** respectivamente para tener un abanico de modelos variado a la hora de obtener resultados.

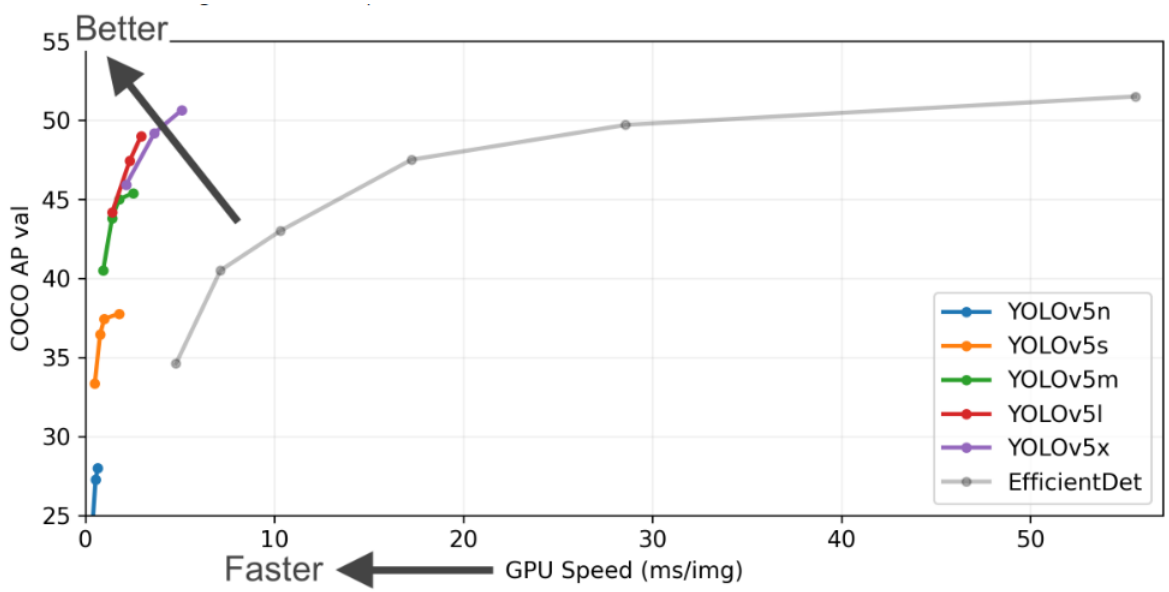

Figure 14: Modelos YOLOv5. Extraído de [3]

Lamentablemente, el tamaño del modelo también afecta muy negativamente a la duración del entrenamiento, el modelo YOLOv5**x** tardaba más de dos meses en completar el entrenamiento con solo la mitad de las imágenes que teníamos preparadas, así que tuvimos que abandonar el modelo. Los modelos que finalmente se han utilizado en este trabajo son YOLOv5**n** y YOLOv5**m**.

5 Datasets

5.1 ¿Qué es un dataset? ¿Cómo deben ser?

Un dataset no es más que un conjunto de datos agrupados con el objetivo de facilitar el análisis de todo el conjunto o el entrenamiento de un modelo. En el caso de redes neuronales para el reconocimiento de imágenes, un dataset es un conjunto de imágenes que, siguiendo un formato, indica para cada imagen donde se encuentra cada objeto que vamos a intentar detectar. A este formato se le llama etiqueta y es lo que se utiliza para comprobar si la red neuronal ha acertado con su predicción. El proceso de añadir la o las etiquetas a una imagen se le llama etiquetado.

La siguiente foto es un ejemplo del dataset BDD100k [20] donde se pueden apreciar los distintos objetos y la posición de ellos marcada con una caja. Tambien nos indica la naturaleza de cada objeto, para poder diferenciar coches, personas o señales de tráfico.

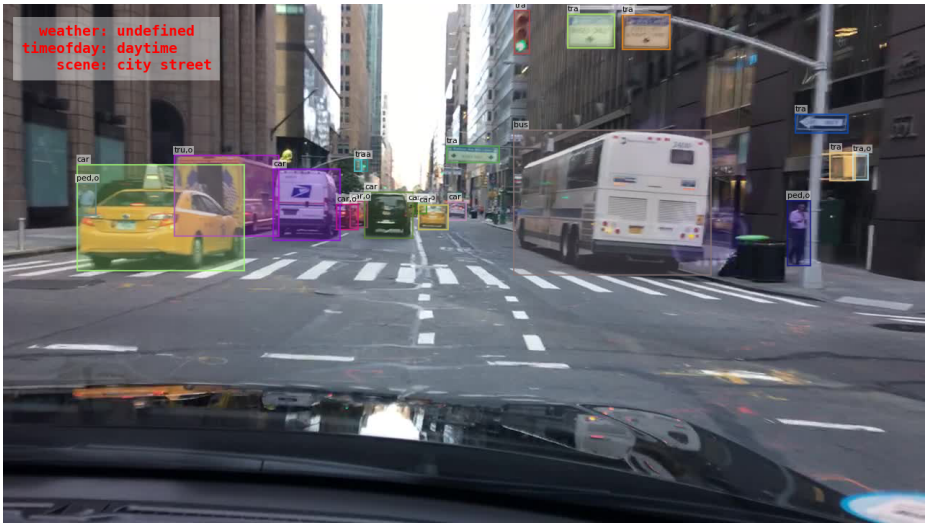

Figure 15: Ejemplo del dataset bdd100k

En la conducción autónoma nos interesan principalmente los datasets que incluyan imágenes de carreteras o caminos donde el punto de vista sea el propio vehículo, ya que es lo que principalmente va a encontrarse una vez implementado el sistema de detección en un coche real. Los datasets deben incluir, a su vez, vehículos, personas y señales de tráfico como objetos a clasificar, porque

estos tienen mucha importancia a la hora de conducir y tomar decisiones. Por ejemplo, frenar en un semáforo en rojo o evitar golpear a otro vehículo. El clima y el tiempo también deben estar reflejados en las imágenes si queremos que el vehículo autónomo sea capaz de moverse adecuadamente en condiciones temporales adversas.

Por último, destacar la importancia de que las imágenes sean diversas y diferentes entre sí. Explicándolo con un ejemplo específico, no nos interesa que la red neuronal aprenda a identificar un modelo de coche concreto o que solo sea capaz de identificar a personas que se vistan de azul.

Lo que buscamos es que sea capaz de identificar cualquier tipo de vehículo, desde motos hasta camiones. Que identifique a cualquier persona independientemente de su ropa o de si es un ciclista o patinador. Y que identifique cualquier señal de tráfico independientemente del clima.

5.2 COCO

COCO [19] es un dataset orientado a la detección de objetos bastante popular y de buena calidad, pero no está dedicado a la conducción autónoma.

- No tiene muchas imágenes comparado con otros datasets.
- Contiene muchas clases identificables (80) de las cuales solo 6 son interesantes para la conducción autónoma.
- No se centra especialmente en la diversidad climática y geográfica.

Aunque YOLOv5 ya este pre-entrenada con el dataset COCO, no nos parece interesante utilizarlo en el proyecto porque no cumple con ninguno de los requisitos necesarios.

5.3 BDD100k

BDD100k [20] es uno de los datasets más grandes dedicados a la conducción autónoma. Contiene más de cien mil vídeos grabados desde un coche mientras este circula por la carretera. Las imágenes del dataset se obtienen de distintos frames de estos vídeos que posteriormente se etiquetan.

- BDD100k posee datos geográficos, medioambientales y climáticos distintos, facilitando que un modelo entrenado con el BDD100k no sea sorprendido por condiciones adversas.
- Este dataset incluye 10 clases identificables. Peatón, coche, camión, autobús, tren, motocicleta, bicicleta, semáforo, luz de tráfico y rider (motoristas y ciclistas están incluidos en una única clase).

Decidimos utilizarlo en el proyecto teniendo en cuenta la diversidad del dataset y que tiene las clases identificables necesarias para la conducción autónoma.

5.4 NUSCENES

NUSCENES [21] es un dataset dedicado a la conducción autónoma más actual que BDD100k. No solo tiene un gran número de imágenes y vídeos, también cuenta con datos de LIDAR y de radar. Esto influye directamente en el dataset, ya que ha permitido que los objetos están etiquetados en 3 dimensiones. Las imágenes han sido tomadas con 6 cámaras apuntando a distintos ángulos del vehículo, permitiendo ángulos e imágenes que BDD100k no incluye que podrían ser beneficiosas.

- Contiene una gran cantidad de imágenes provenientes de Boston y Singapore y alrededores. Dos zonas de distinto clima y geográficamente diferentes.
- Este dataset incluye 23 clases identificables. Aunque son bastante más que las que utiliza BDD100k, no son mucho más diversas. La diferencia principal es que en NUSCENES especifica más que es cada objeto. Por ejemplo,

Figure 16: Ejemplo del dataset NUSCENES

en vez de simplemente etiquetar personas, las diferencias entre niños, adultos, policías...

Hemos decidido utilizarlo porque cumpla con los requisitos que nos hemos propuesto.

Aunque puede ser interesante que las clases etiquetadas sean más específicas, hemos decidimos acotar el número de clases para facilitar la comparación de sus datos con los obtenidos al entrenar con BDD100k.

Las clases que finalmente incluye son 14: Animal, peatón, barrera, escombros, objeto empujable, cono de tráfico, aparca-bicicletas, bicicletas, autobuses, coches, vehículo de construcción, furgoneta, motocicleta y camión.

6 Métricas

6.1 Medir la predicción

Con medir la predicción nos referimos a cuantificar matemáticamente como de correcta ha sido una predicción del detector de objetos. Es decir, si se ha detectado correctamente la localización del objeto y la clase a la que pertenece.

Para ello es necesario explicar distintas medidas que combinadas nos llevan la medida final utilizada en el proyecto, el mAP, que se explica en la sección 6.1.3.

6.1.1 Matriz de confusión

Para entender las medidas básicas es importante definir que es la matriz de confusión [22] (figura 17), donde se representan los datos reales frente a los datos predichos por la tecnología que esté realizando la predicción.

		predicción	
		0	1
realidad	0	70	10
	1	15	5

		predicción	
		0	1
realidad	0	TN	FP
	1	FN	TP

Figure 17: Matriz de confusión. Extraído de [25]

En una matriz de confusión falso(F)/verdadero(T) nos indica si el predictor ha acertado, positivo(P)/negativo(N) nos indica que se ha predicho:

- **TP**. True positive o verdadero positivo. Se da cuando la predicción es positiva y el resultado real es verdadero. Por ejemplo, La imagen es un coche y se predice que sí es un coche.
- **FP**. False positive o falso positivo. Se da cuando la predicción es positiva, pero el resultado real es falso. Por ejemplo, la imagen es un camión y se predice que sí es un coche.

- **TN**. True negative o verdadero negativo. Se da cuando la predicción es negativa y el resultado real es falso. Por ejemplo, la imagen es un camión y se predice que no es un coche.

- **FN**. False negative o falso negativo. Se da cuando la predicción es negativa, pero el resultado real es verdadero. Por ejemplo, la imagen es un coche y se predice que no es un coche.

La matriz de confusión nos sirve para representar resultados binarios, predicción acertada o predicción fallida. La detección de múltiples objetos no es tan sencilla. Tenemos que tener en cuenta dos cosas para determinar el resultado, si se ha detectado correctamente la clase del objeto a detectar y si la localización del objeto es correcta.

La existencia de más de una clase supone un ligero problema. El ejemplo ya no es ”es un coche o no lo es”, ahora nos encontramos con ”es un coche, o es un camión, o es un peatón, o etc.”. Existen distintos resultados acertados para cada clase. Una solución a este problema es hacer una matriz de confusión para cada clase.

Respecto a determinar si la localización del objeto es correcta, si revisamos el auténtico resultado de una detección de objetos (figura 18) podemos observar como el recuadro verde (predicción) no es igual al azul (realidad). ¿Deberíamos contar como que ha acertado correctamente la posición del coche que se encuentra en el centro de la imagen? ¿Solo contamos los casos donde los recuadros estén completamente solapados?

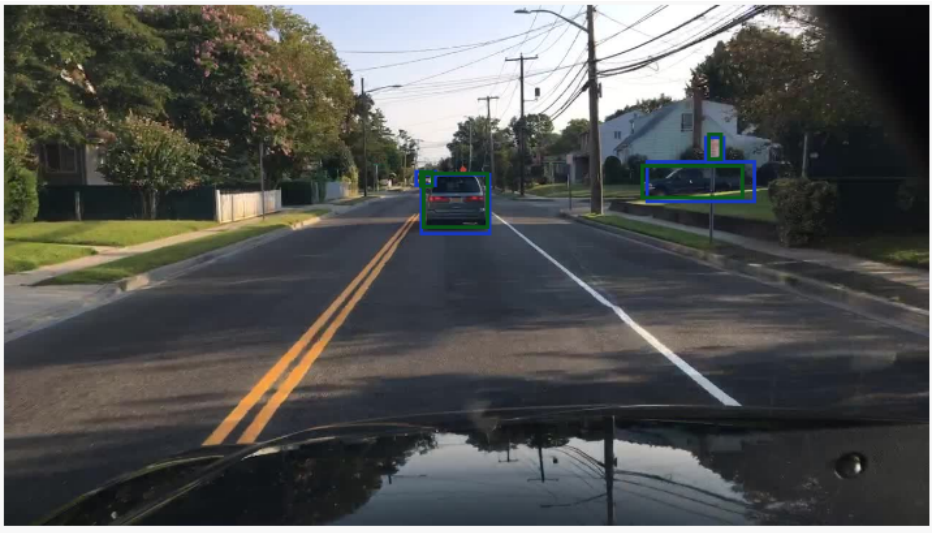

Figure 18: Ejemplo de detección. Verde = Predicción, azul = realidad

IoU Para solucionar este segundo problema, en la detección de objetos se utiliza el IoU [23] cuya fórmula es la siguiente (figura 19). De esta forma podemos determinar matemáticamente cuan precisa ha sido la detección de la localización del objeto.

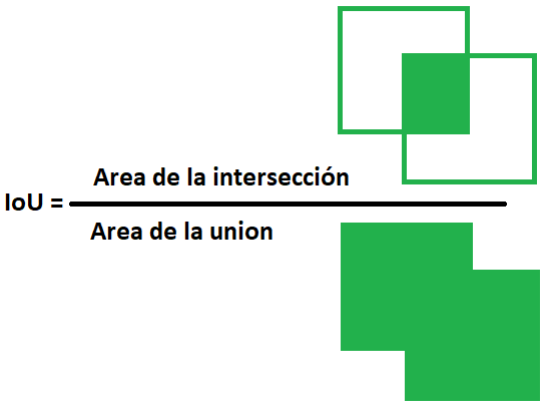

Figure 19: Formula de IoU

Ahora podemos definir un mínimo de IoU necesario para considerar si una predicción ha acertado la posición del objeto, y si además se ha acertado la clase correctamente, podemos entenderlo como un verdadero positivo.

Por ejemplo, si suponemos que el coche de la figura tiene un IoU de 0.85 y consideramos que es lo suficientemente buena, podemos contar como acierto

todas las predicciones con un IoU mayor o igual a 0.85.

6.1.2 Medidas basadas en la matriz de confusión

Una vez claros los conceptos de la matriz de confusión y como reconocer un acierto, podemos utilizarlos para obtener las métricas de precisión que nos van a ayudar a entender como de buena es una predicción.

Precision La fórmula de la precision [22] (precisión en castellano) es la siguiente (figura 20). Esta medida indica la seguridad del predictor de que cuando predice positivo, es realmente un verdadero positivo.

Generalmente, cuanto más alto sea este valor, mejor, siempre está bien que el predictor este seguro de haber predecido correctamente. Pero no se debe tener en cuenta como una métrica general para evaluar predicciones, ya que puede dar lugar a casos que no son los que habitualmente alguien se esperaría de un 100% de precisión.

Ejemplo 1. Imaginemos un caso donde hay 50 coches y 50 camiones, se predice un coche como positivo y los 99 vehículos restantes como negativos. La precisión del sistema al identificar coches es de 1. 100% de precisión a pesar de haber fallado en la predicción de los otros 49 coches.

$$precision = \frac{TP}{TP + FP}$$

Figure 20: Formula de precisión. Extraído de [25]

Recall La fórmula del recall [22] (exhaustividad en castellano) es la siguiente (figura 21). Esta medida indica cuantos de los casos reales se han detectado como positivos.

Igual que con la precisión, cuanto más alto mejor, puesto que queremos detectar todos los casos verdaderos. También, de forma similar a la precisión, un alto recall no implica que el predictor esté funcionando muy bien.

Ejemplo 2. Imaginemos un caso como el del ejemplo 1, 50 coches y 50 camiones. Si todo se predice como coche, efectivamente todos los coches se han predicho correctamente. 100% de recall, pero se ha fallado la predicción de todos los camiones.

$$recall = \frac{TP}{TP + FN}$$

Figure 21: Formula de recall. Extraído de [25]

Accuracy La fórmula del accuracy [22] (exactitud en castellano) es la siguiente (figura 22). Esta medida indíca cuantos casos ha predicho correctamente respecto al total de casos.

Puede parecer una medida mejor que las anteriores para evaluar como de bien predice un sistema, pero sigue sin ser completa para entender como está funcionando el predictor. En los dos ejemplos anteriores se obtiene una exactitud del 50% a pesar de que los resultados de la matriz de confusión son bastante distintos.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Figure 22: Formula de accuracy. Extraído de [25]

Precision y Recall Calculando las tres medidas de los dos ejemplos obtenemos:

- **Ejemplo 1**. Precision 1, Recall 0.02, Accuaracy 0,51.
- **Ejemplo 2**. Precision 0.5, Recall 1, Accuaracy 0,50.

Podemos observar como mientras que la Accuaracy no nos da ninguna información sobre el predictor, la precisión y el recall combinados si nos permiten entender mejor su funcionamiento:

- **Ejemplo 1**. Aunque la precisión sea del 100%. Lo único que nos garantiza es que si dice que algo es positivo, seguramente sea verdadero. Podría ser útil a la hora de probar un fármaco muy potente, solo queremos usarlo en pacientes que estemos completamente seguros de que sufren la enfermedad, aunque no se vayan a detectar muchos de los casos verdaderos.

- **Ejemplo 2**. Este ejemplo nos garantiza que se van a detectar la mayoría de los casos verdaderos, aunque su precisión nos indica que se va a catalogar como verdadero muchos casos falsos. Podría ser útil si queremos tratar a contagiados de una enfermedad contagiosa con una medicina que no tiene ningún efecto dañino para el paciente. Queremos tener un recall alto, ya que aunque nos confundamos con algunos pacientes falsos, vamos a asegurarnos de tratar a todo el que lo necesite sin perjudicar a los que no.

6.1.3 AP y mAP

Finalmente, el AP (Average precision) [23] es la medida que vamos a utilizar para comprobar como de certeza es la predicción para una clase concreta. Esta medida combina la relación entre la precisión y el recall que hemos explicado y que además tiene en cuenta distintos valores de IoU.

Primero se calculan los distintos valores de precision y de recall para distintos valores de IoU. Después se representan estos valores en una tabla similar a la figura 23. AP es el área bajo la curva.

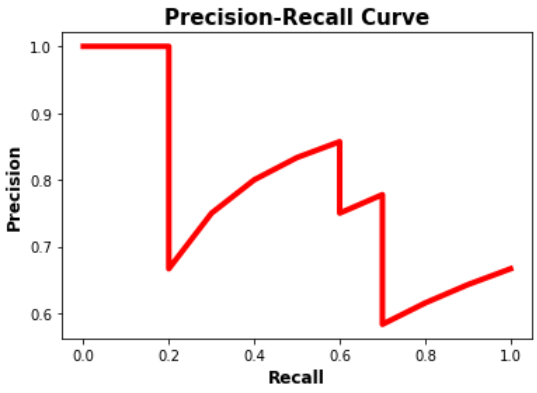

Figure 23: Curva de precision-recall

De esta manera hemos logrado calcular de una forma adecuada un único valor que es capaz de representar como de bien se está prediciendo una clase. Al tener en cuenta todas las clases, la medida generalmente se llama mAP (medium Average Precision), que no es más que la media de los valores de AP para cada clase.

6.2 Medir la velocidad de inferencia

La red neuronal requiere un tiempo de procesado para detectar los objetos que hay en la imagen. Este tiempo se ve influido tanto por el tipo de procesador donde se esté ejecutando como con la dificultad de la imagen a procesar y su resolución. Para medir este tiempo de procesado vamos a utilizar los FPS (frames per second) o imágenes por segundo en castellano. Que no es más que medir el tiempo total que ha tardado en realizar la inferencia entre las imágenes de un dataset, y aplicar la siguiente fórmula (figura 24).

$$FPS \frac{\text{nº imagenes}}{\text{tiempo total}}$$

Figure 24: Fórmula de FPS

6.3 Optimizar estas medidas

Con el ánimo de obtener los mejores resultados posibles y facilitar las pruebas en distinto hardware, se va a utilizar OpenVINO [24] como herramienta de optimización.

OpenVINO es una herramienta open source para optimizar y realizar inferencias sobre distintas inteligencias artificiales. Con OpenVINO es posible aumentar los FPS de una CNN sin reducir sustancialmente el mAP. Una vez optimizada, permite obtener estas dos medidas en distintas máquinas objetivo como distintos procesadores o GPU de intel.

En la figura 25 podemos observar todos los diferentes frameworks que acepta y es capaz de optimizar, también podemos ver los diferentes hardwares de intel disponibles como máquinas remotas para realizar inferencias y obtener resultados.

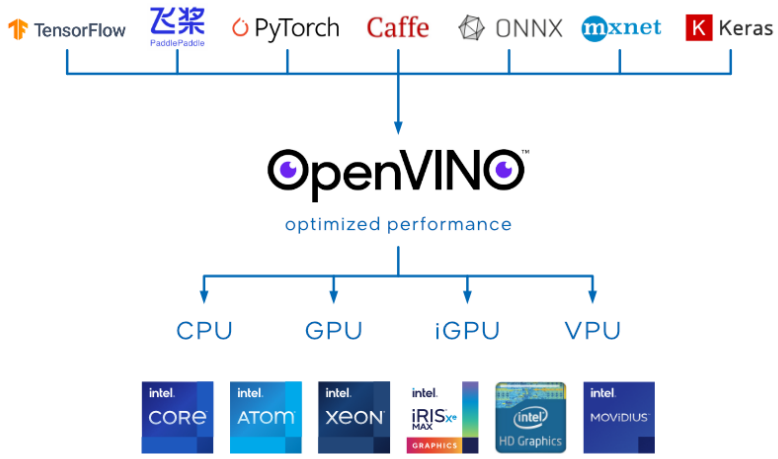

Figure 25: Frameworks soportados y hardware disponible en OpenVINO [24]

7 Metodología

El diagrama de la figura 26 es un resumen de los pasos realizados durante este proyecto. En verde podemos ver el tratamiento de datasets, en naranja los pasos a seguir con YOLOv5 y en azul los procedimientos ejecutados en el workbench de OpenVINO y en el cloud de Intel.

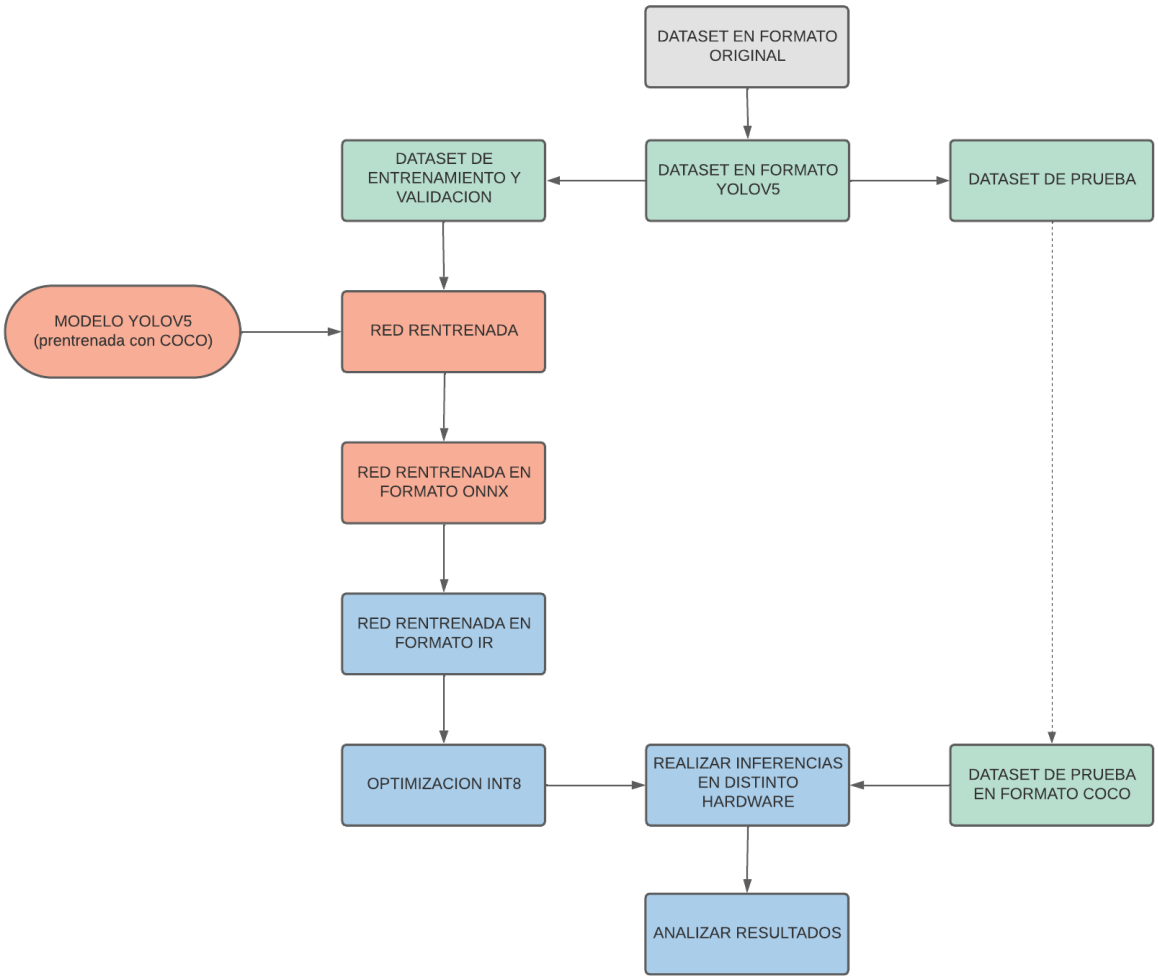

Figure 26: Paso a paso de todo el proyecto

Para realizar el proyecto se utiliza la red YOLOv5 con tres de sus modelos, YOLOv5**n**, YOLOv5**m** y YOLOv5**x**. Cada uno de estos modelos se entrena ,por un lado, con el dataset BDD100k y, por otro lado, con el NUSCENES para finalmente obtener 6 modelos entrenados.

En la figura 27 podemos observar todos los modelos disponibles y su tamaño

en MB, velocidad de inferencia en ms y su precisión en mAP para el dataset COCO.

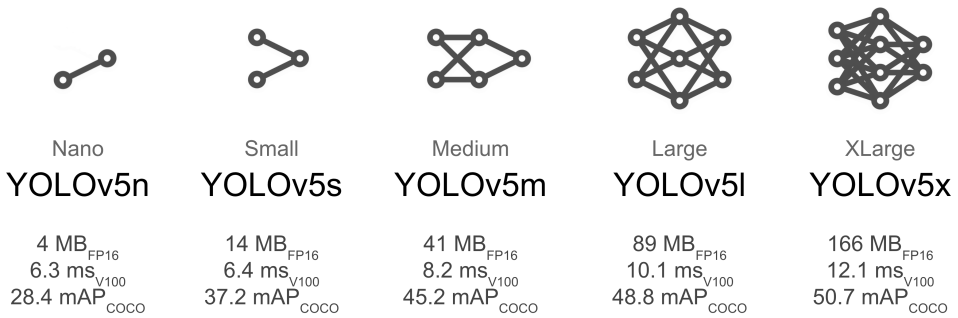

Figure 27: Modelos YOLOv5. Extraído de [3]

7.1 Conversión de los datasets a YOLOv5

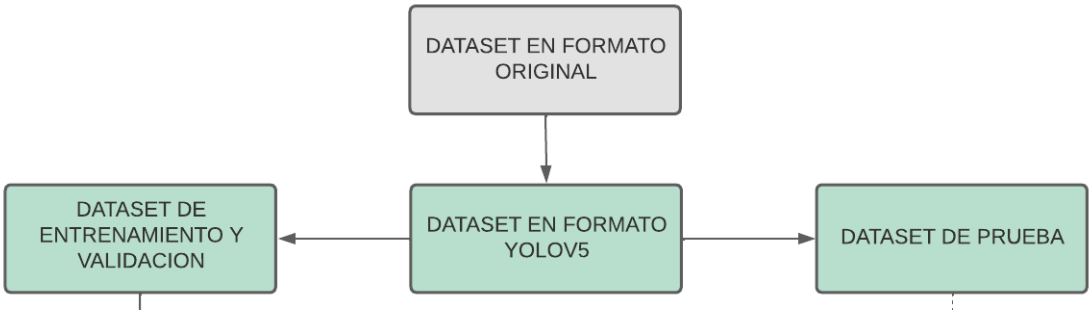

Figure 28: Pasos para tratar los datasets

El primer problema que se platea es debido al formato de los datasets, YOLOv5 solo acepta un formato propio muy concreto, este formato se basa en que las anotaciones se representan mediante un txt por cada foto del tipo nombrefoto.txt, y dentro contiene:

- Una línea por cada objeto en la foto.
- La línea está formada por el id de la clase, las coordenadas ”x” e ”y” del centro del recuadro etiquetado, y el ancho y alto del recuadro etiquetado, véase en la figura 29

Para realizar la conversión se descarga el dataset Bdd100k en su propio formato JSON y se crea un script en Python para realizar la conversión [26]. Su

funcionamiento es muy simple, se lee el JSON en formato Bdd100k, se reconocen los datos del objeto y se convierten al formato de YOLOv5 mediante una serie de operaciones simples. El único detalle importante a mencionar es que las posiciones en foto en el formato YOLOv5 van normalizadas.

Para la conversión del dataset NUSCENES utilizamos un script que ya existe, ya que el formato del dataset es complicado [28]. Los recuadros de las etiquetas en NUSCENES vienen dados en 3 dimensiones, pero este código las convierte a formato YOLOv5 de una forma similar a nuestro script para el dataset bdd100k.

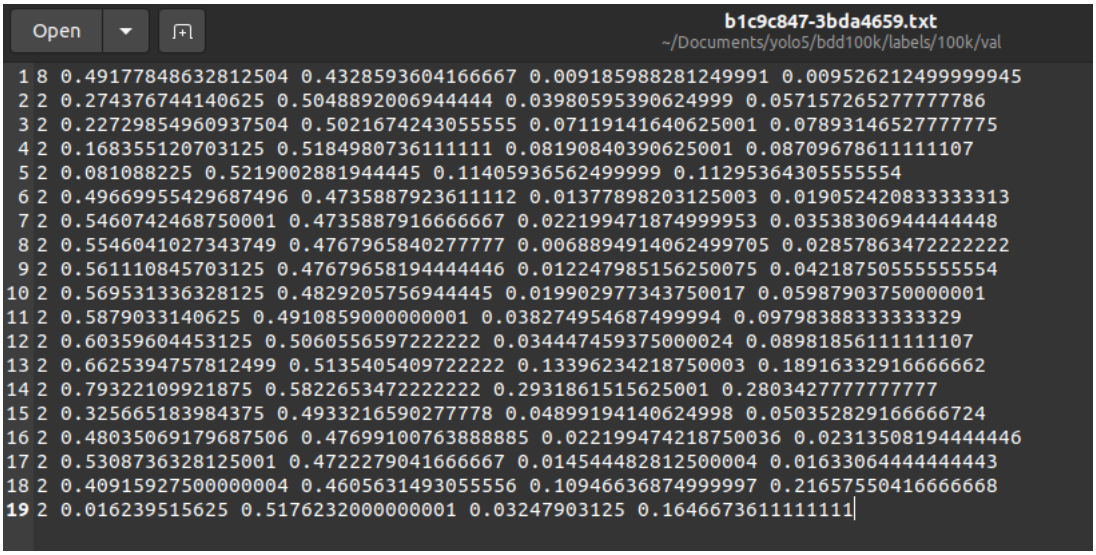

Figure 29: Ejemplo de label en formato YOLOv5

7.2 Entrenamiento

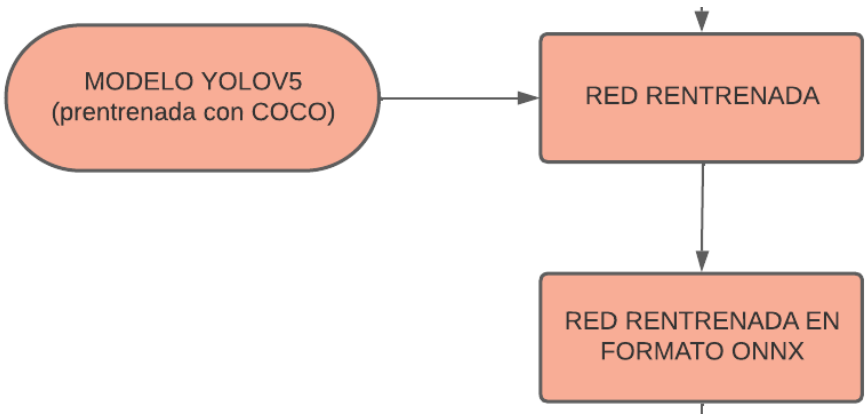

Figure 30: Modificaciones y entrenamiento de YOLOv5

Durante el proceso de entrenamiento se van a ejecutar/recorrer varias épocas repasando el dataset y mientras que la red neuronal aprende. En la documentación de la YOLOv5 se recomienda empezar el entrenamiento con 300 épocas

Lo primero de lo que nos damos cuenta al empezar a entrenar los modelos es que se tarda mucho tiempo. En nuestros ordenadores locales, se tardaba en torno a 1 semana en recorrer una época.

Para la realización del entreno se nos proporcionó acceso a un computador con hardware muy potente (Intel Xeon Gold 2.00GHz, NVIDIA GeForce RTX 3090 y una NVIDIA Tesla V100), ya que el entrenamiento requiere de una buena GPU para ejecutarse en un tiempo razonable.

Para realizar el entrenamiento es necesario conectarse mediante ssh al computador, transferir los modelos, los datasets y los archivos YAML de configuración. En estos archivos YAML se establecen las clases a identificar y la ruta a las fotos y las anotaciones, para esto se debe crear con el siguiente formato:

- Ruta relativa a los datasets.
- Numero de clases.
- Los nombres de las clases.

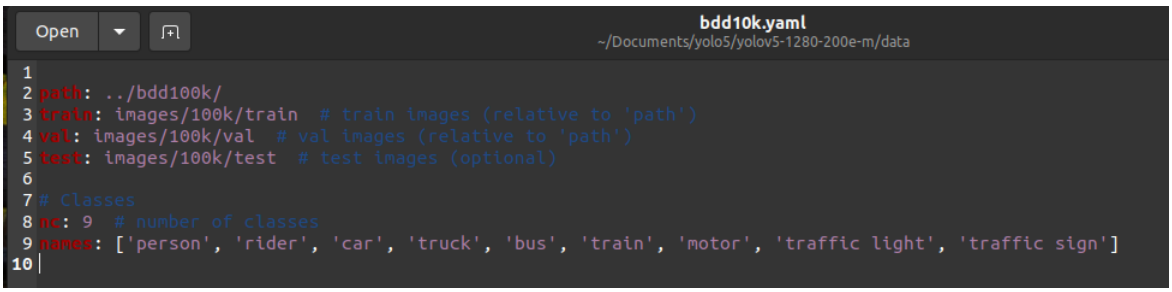

Figure 31: Ejemplo de fichero de configuración YAML

Para realizar el entrenamiento, una vez tenemos preparados los datasets ejecutamos el comando de entrenamiento:

```
python3 train.py --img 1280 --batch 16 --epochs 200 --device 1
--data ../bdd100k.yalm --weights yolov5n.pt
```

- **–img**. Índica el tamaño esperable de las imágenes del dataset.
- **–batch**. Índica él cuantas imágenes se van a estar observando a la vez durante el entrenamiento. Según nuestras pruebas, reduce la duración del entrenamiento cuanto más alto el valor, pero también consume más energía y uso de la GPU. Lo recomendable es utilizar cuantos más batches mejor, pero debido a que el ordenador no es de nuestra propiedad, utilizamos 16 en vez de los 32 máximos posibles.
- **–epochs**. Índica cuantas épocas se deben recorrer para terminar el entrenamiento. A pesar de que lo recomendado son al menos 300, durante el entrenamiento observamos que a partir de las 200 no se apreciaba una mejora notable. Aquí (figura 32) podemos observar como en torno a la época (step) 175, la precisión se mantiene estable.
- **–weights**. Indica los pesos del modelo que se va a entrenar. En este ejemplo yolo5n.pt indica utilizar el modelo nano de YOLOv5.

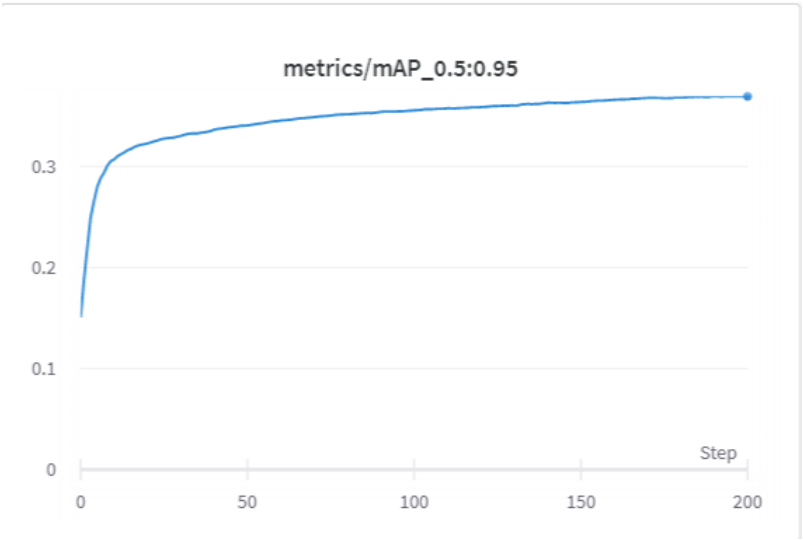

Figure 32: Map durante el entrenamiento de YOLOv5n con nuscenes

–device y –data no son parámetros que afecten al entrenamiento, simplemente escogen la GPU 1 del ordenador y la ruta relativa al archivo .yalm de configuración respectivamente,

Es durante esta fase que nos dimos cuenta de que incluso con una GPU potente, los tiempos de entrenamiento seguían siendo demasiado largos. El entrenamiento del modelo X con el dataset BDD100k iba a tardar aproximadamente 2 meses en completarse y con NUSCENES se preveía 3 meses o más. Por tanto, decidimos eliminar las pruebas con los modelos YOLOv5x, ya que la duración del entrenamiento era extremadamente larga.

El modelo YOLOv5n en comparación solo tardaba una semana en entrenarse y YOLOv5m tardaba un mes.

7.3 Conversion a IR

Para exportar este modelo y poder trabajar con él en OpenVino, se requiere primero exportar la red YOLOv5 a formato IR. Utilizamos un script que proporciona Ultralitycs para exportarla al formato ONNX, un formato que OpenVINO si soporta.

```
python3 export.py --weights runs/train/exp/weights/best.pt
--include onnx
```

Este comando exporta la red neuronal cuyo modelo se encuentra en la ruta –weights y lo exporta al formato que indica –include, ONNX en este caso.

Ahora, con la red en formato ONNX, OpenVINO es capaz de interpretarla y convertirla al formato IR que necesita.

```
python3.8 mo.py
--input_model
./Documents/yolo5/yolov5-1280-200e-n/runs/train/exp/weights/best.onnx
--output_dir /Documents/yolo5/yolov5-1280-200e-n/runs/train/exp/weights
--s 255 --reverse_input_channels --output Conv_198,Conv_233,Conv_268
```

Este comando incluye opciones que son importantes y que afectan al rendimiento de la red neuronal.

- **–s**. Indica el escalado de los valores del modelo. Como estábamos trabajando en RGB, cada canal de color tiene un tamaño de 255.

- **–reverse_input_channels**. Invierte el sentido de los canales de entrada de RGB a BGR o viceversa. Tras experimentar con la opción, comprobamos que se obtienen mejores resultados activando la opción e invirtiendo el sentido.

- **–output**. En output es necesario incluir el nombre de las neuronas que sirven de output a la red neuronal. En el caso de YOLOv5, 3 neuronas son las encargadas del output. Se utilizó Netron [5] para visualizar la red y poder encontrar estas neuronas.

Cabe destacar que aunque inicialmente no existía esta opción, a finales del desarrollo de este proyecto, se añadió la opción de exportar directamente al formato IR desde el propio script de Ultralitycs.

7.4 Inferencia y optimización

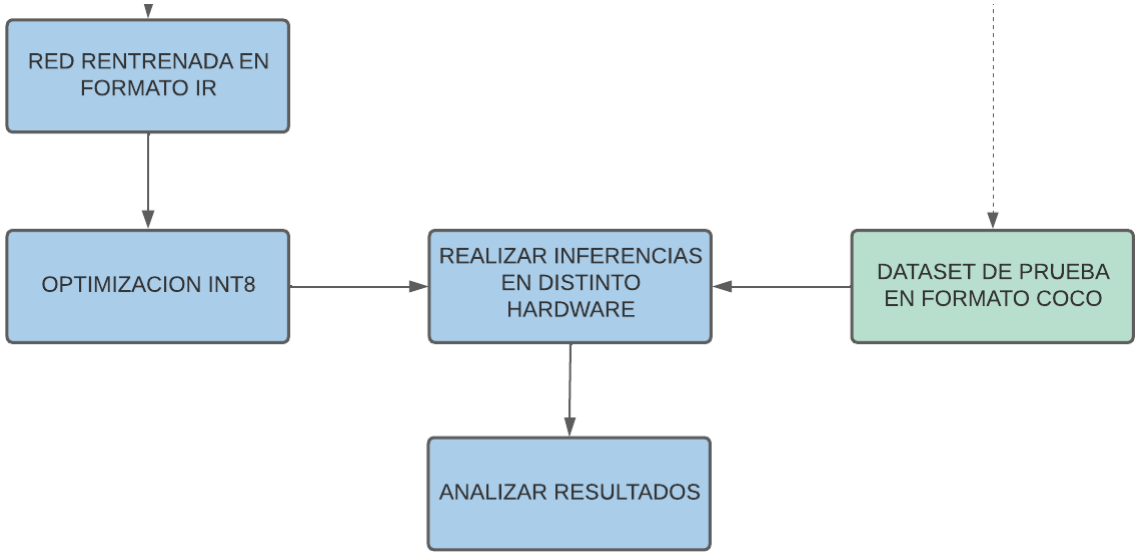

Figure 33: Pasos ejecutados en OpenVINO

Una vez tenemos el modelo en IR podemos pasar a realizar las pruebas, inferencia y precisión del modelo, para esto podemos hacerlo directamente con la API de OpenVINO llamando a sus scripts manualmente o utilizando una interfaz conocida como Workbench que además facilita el acceso al cloud de intel, llamado Edge.

Es muy sencillo acceder al workbench, se debe autorizar el acceso para un proyecto en concreto, inicializar la carga del workbench desde un cuaderno de Jupyter y tras unos minutos de espera, la aplicación del workbench se lanza automáticamente.

Para inicializar el workbench en local debemos tener instalado previamente el programa Docker[29] donde se cargara como un contenedor la aplicación, a partir de una imagen que se descarga de la web de OpenVINO. [24]

Utilizar el cloud proporciona la principal ventaja de poder utilizar distinto hardware con el que no contamos, aunque por lo general funciona mucho mejor en local.

Una vez en el Workbench procedemos a importar el modelo IR y le aplicamos la configuración elegida en la figura 34.

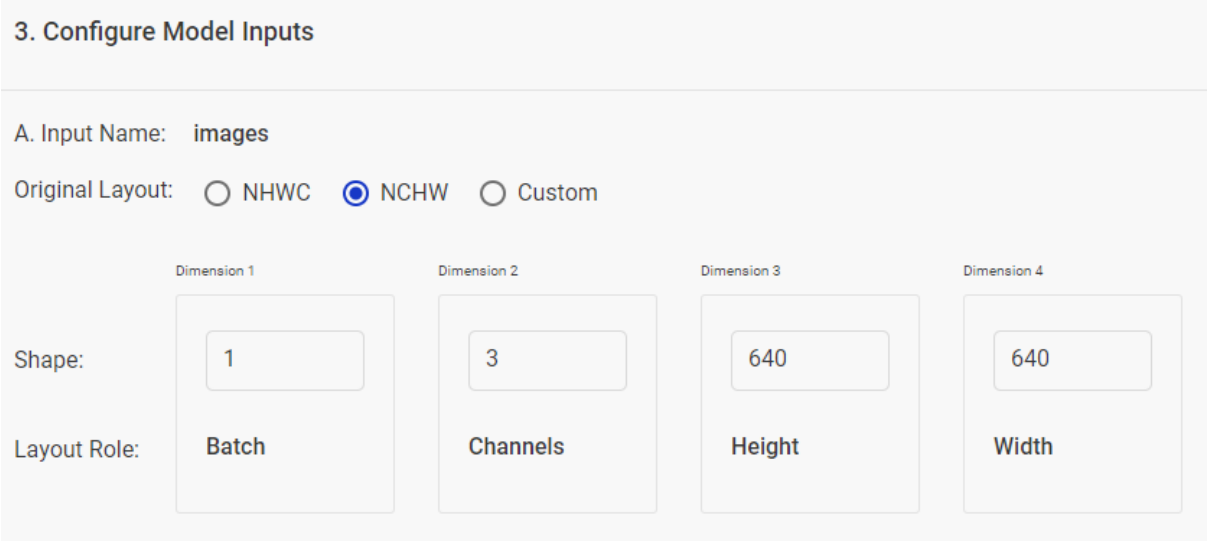

Figure 34: Configuración modelo IR

En esta configuración definimos:

- El número batches, es decir, el número de imágenes que se carga en cada batch. Por defecto al exportarlo con los comandos anteriores es 1 y es estático, aunque existen formas que no hemos explorado de hacerlo dinámico.
- El número de canales en nuestro caso siempre es 3 porque son a color, en el caso de ser a escala de grises sería 1.

- El ancho y el alto de las imágenes, por comprobaciones experimentales para un correcto funcionamiento en la detección de objetos, los modelos deben configurares en una resolución cuadrada, es decir 640x640 o 720x720 o 1280x1280, en este caso nos decantamos por 640 para obtener un mayor velocidad de inferencia aunque esto afecta a la precisión.

7.4.1 Conversión de modelos a coco

Una vez tenemos el modelo ya importado, elegimos el hardware en el que se realizará la inferencia y seleccionamos un dataset. Este dataset deber ser importado en formato del dataset COCO o sin formato, aunque para poder realizar los test de precisión de detección de objetos es necesario que esté en el formato de COCO.

Para realizar esto se proceden a realizar 2 códigos, uno por cada dataset que son modificaciones ligeras del los códigos que convertían al formato YOLOv5, además estos códigos se basan en el código presentado por otro compañero en un trabajo anterior[27]. Los códigos generan los archivos JSON en formato COCO que junto a las imágenes se almacenan en un ZIP. Este ZIP debe contener una carpeta llamada val con las imágenes y otra llamada annotations con un fichero instances_val.json con las anotaciones.

7.4.2 Inferencia

Una vez el proyecto está creado y completo, automáticamente se realizan pruebas de inferencia de donde obtener los FPS y la latencia para los distintos dispositivos usados. También podemos realizar pruebas con diferente número de streams (figura 35) para mejorar alguno de estos valores. Al final, si se aumentan los FPS, la latencia también aumenta, así que hay que buscar un equilibrio entre ambos valores.

El otro valor a modificar son los batchs, pero como comentaba anteriormente en nuestro modelo, son estáticos.

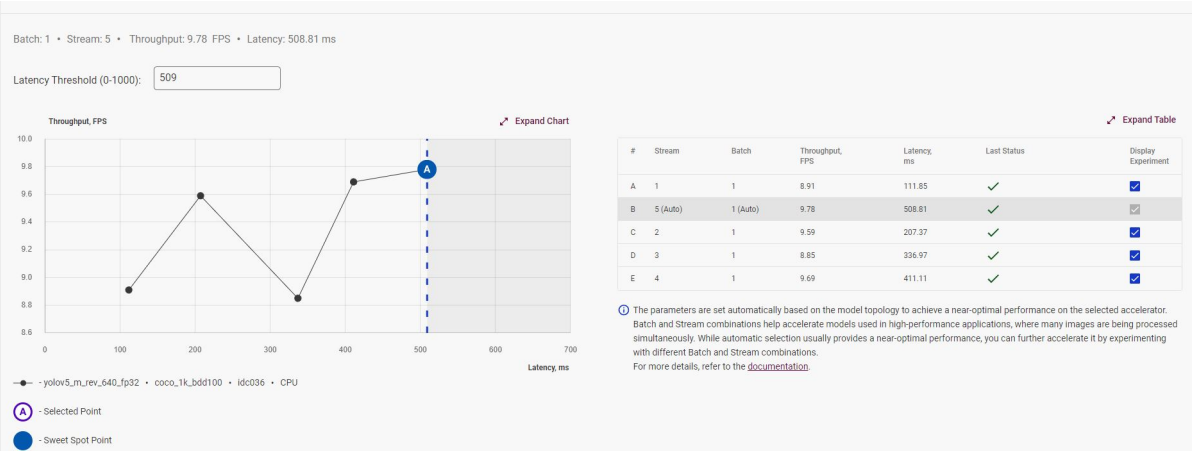

Figure 35: Gráfica fps/latencia

7.4.3 Prueba de precisión

Para medir la precisión se debe proveer de una configuración como la de la figura 36. OpenVINO no soporta de forma nativa a la YOLOv5, pero si tiene soporte para YOLOv3 y YOLOv4. Se escoge el modelo del tipo YOLOv3, ya que experimentalmente se obtiene mejor precisión que escogiendo YOLOv4. A continuación se añade el número de clases de nuestro modelo y posteriormente el valor de NMS overlap en nuestro caso lo bajamos a 0.25 y la métrica la dejamos por defecto mAP 0,5.

Cabe destacar que los 2 datasets de validación que se subieron inicialmente, con 10 mil imágenes cada uno, daban errores a la hora de obtener la precisión, los cual nos llevó a realizar las pruebas únicamente con mil imágenes de validación, en el caso del dataset BDD100k y 500 en el caso de NUSCENES y que al ser tan pocas imágenes, los resultados puede que sean extraños.

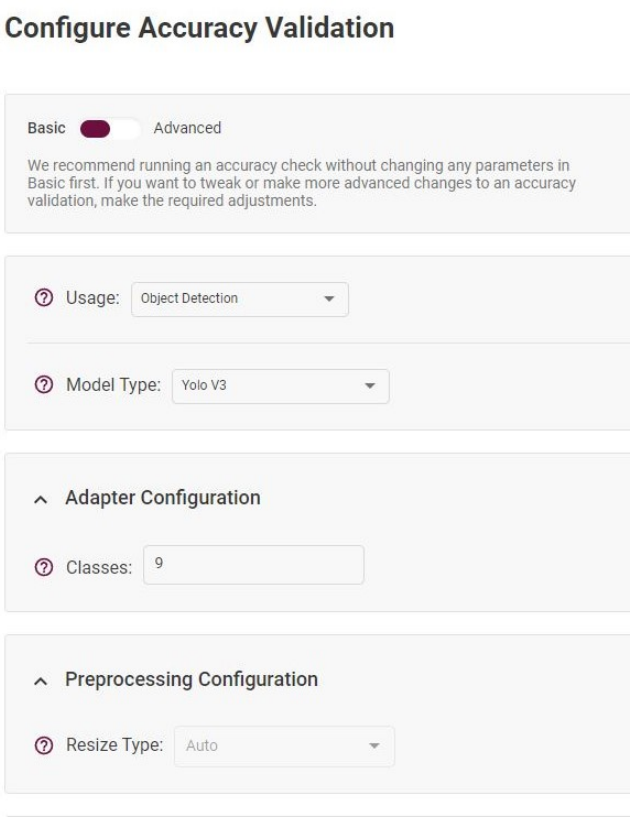

Figure 36: Configuración acurracy

7.5 Optimización a int8

OpenVINO permite mejorar los FPS del modelo, bajando la precisión del modelo de fp32 o fp16 a int8 sin reducir notablemente el mAP, aunque esto depende de la configuración escogida a la hora de realizar la calibración a int8.

Para las pruebas se realizaron 2 configuraciones, una en local y otra en el cloud:

- Para mejorar rendimiento se escogió el método por defecto y la mejora máxima de FPS. Como se puede ver en la figura 37 produce una mayor perdida de precisión en el modelo.
- La otra configuración realizada en el workbench en local se realiza con AccuaracyAware Method y Mixed Presed. Como se puede ver en la figura 38, esta configuración hace que la perdida de precisión sea mínima, pero el tiempo de cómputo es extremadamente alto, produciendo que en el cloud

muchas veces de fallo al terminarse el tiempo disponible por sesión, por tanto, nos vemos obligados a utilizar la primera de las configuraciones.

Otro detalle importante es escoger el porcentaje de las fotos del dataset para realizar la calibración a int8, se recomienda escoger entre 300 y 1000 fotos, en nuestro caso nos quedamos con 500 fotos

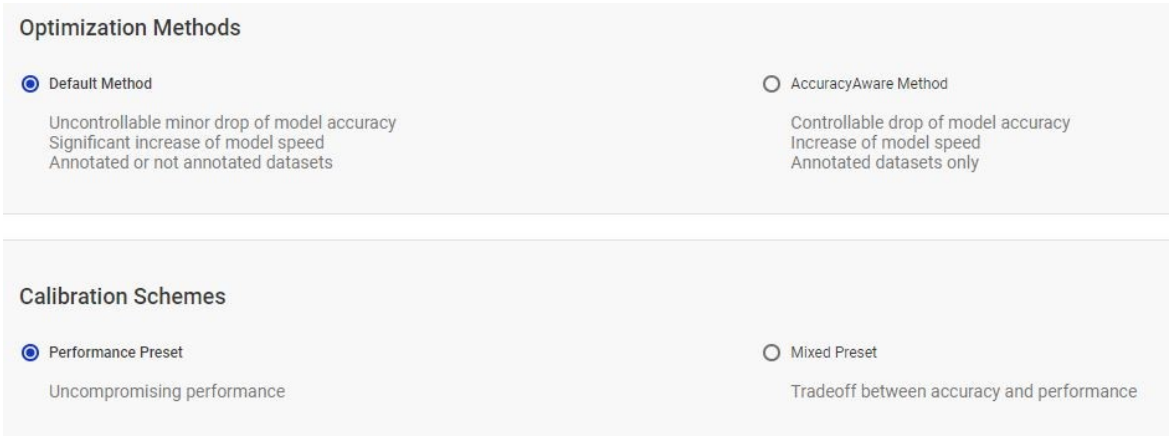

Figure 37: Configuración optimizacion int8

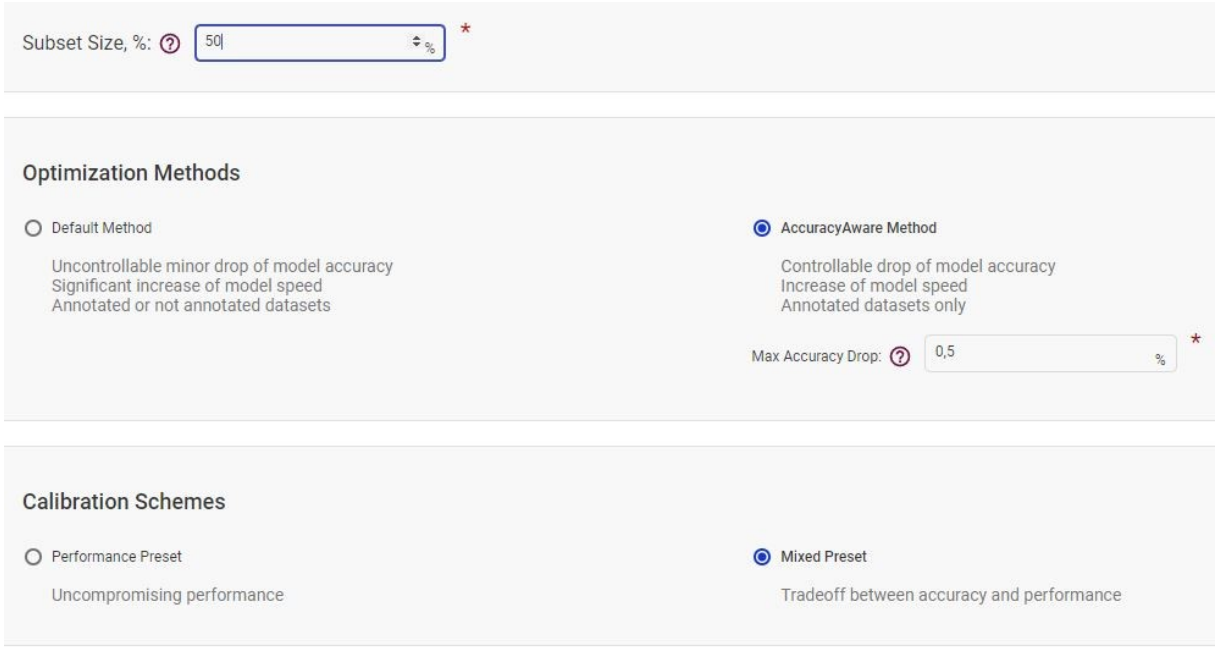

Figure 38: Configuración optimizacion int8 maximo acurracy

8 Análisis de resultados

Inicialmente, vamos a comparar la precisión obtenida al terminar él reentreno con los datasets dedicados a la automoción. Posteriormente, observaremos todos los resultados de los diferentes modelos una vez los optimizamos con OpenVINO y hacemos pruebas con distinto hardware.

Cabe destacar que al concluir con el análisis de los resultados, nos pareció interesante reentrenar el modelo small de YOLOv5 para compararlo con los otros modelos. Según los resultados, puede que el modelo small sea el óptimo para la conducción autónoma, No nos ha parecido interesante incluir este modelo en OpenVINO por los problemas que se comentan a lo largo de esta sección.

8.1 Resultados del reentreno

Estos son los resultados del reentrenamiento antes de optimizarlo. En las figuras 39 y 40 podemos observar como el modelo medium tiene mejor precisión que el modelo nano aunque alcanza menos cantidad de fotogramas por segundo. Como esperábamos, el modelo small obtiene una precisión bastante mejor que el modelo nano mientras que funciona más rápido que el modelo medium.

Los resultados de velocidad se han obtenido utilizando una RTX 2060.

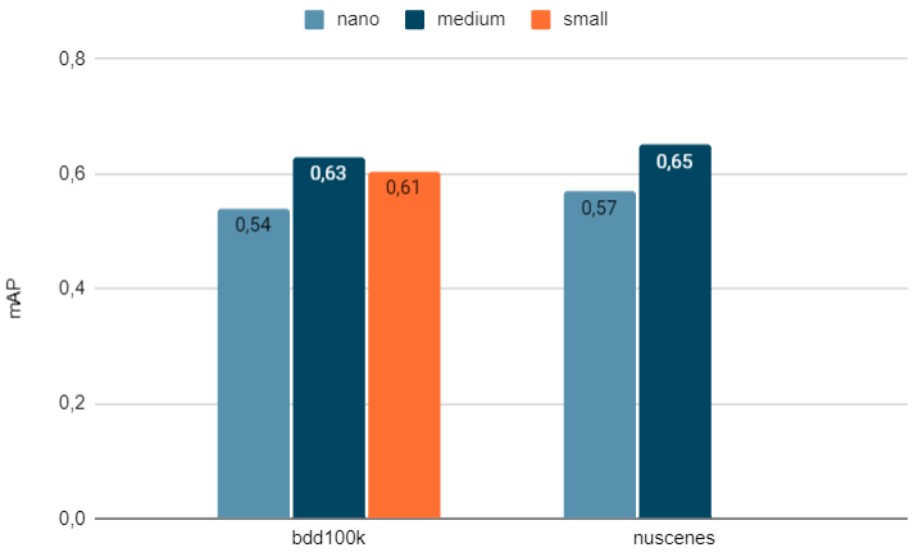

Figure 39: Precisión pre optimización

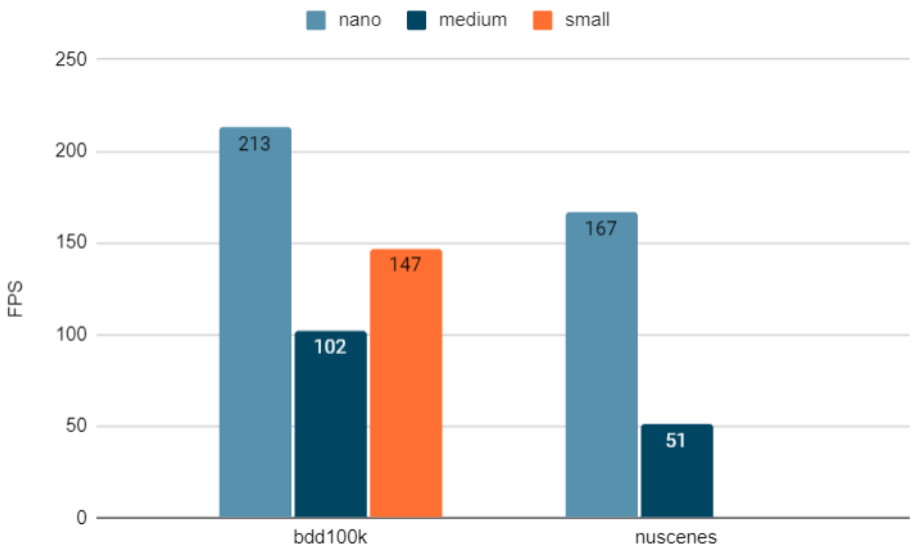

Figure 40: Velocidad pre optimización

En la figura 41 podemos observar visualmente un resultado real de una imagen de BDD100K para el modelo medium. En la imagen podemos observar como ha detectado correctamente a la mayoría de los objetos que están en el plano. Aun así, tiene problemas para detectar los que están al fondo de la imagen.

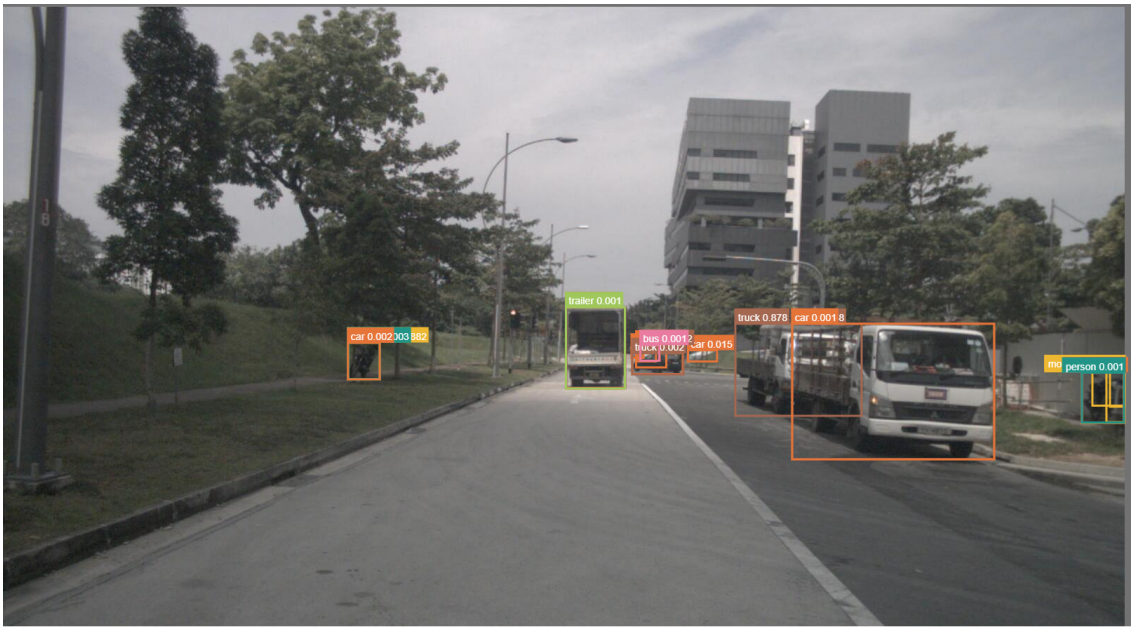

Figure 41: Ejemplo preoptimizacion YOLOv5m BDD100k

8.2 Resultados de la inferencia y optimización

Analizaremos la precisión obtenida y la velocidad de inferencia para los distintos modelos antes y después de aplicar la calibración int8.

En local se utiliza una configuración que permite mantener la precisión lo más alta posible al calibrar a int8 además de utilizar una resolución de 1280x1280 en el caso de la BDD100K y de 1600x1600 en NUSCENES. Esto mejora la precisión, aunque también aumenta el tiempo de ejecución y reduce los FPS considerablemente.

En el cloud utilizaremos la configuración por defecto que busca la máxima mejora de fps sin tener en cuenta la perdida de precisión, con una resolución 640x640 en el caso de BDD100K y 1280x1280 en NUSCENES.

8.3 Resultados locales

El hardware usado para las pruebas locales es un i7 6700k (4.3GHz), pero en las pruebas la potencia no es usado al 100 por cien, ya que se le proporciona solo el 50 por ciento de los recursos del computador al contenedor de docker donde se ejecutan las pruebas.

8.3.1 Análisis de precisión

En la figura 42 se puede observar la precisión obtenida en los diferentes modelos. Esperamos una precisión inferior en los modelos entrenados con NUSCENES, ya que tiene que predecir más clases, cosa que no ocurre. Creemos que la causa es la mayor resolución de las imágenes de NUSCENES, 1600 en contra de los 1280 de bdd100k. Sorprende que los datos calibrados con int8 tengan mejor precisión que los que no.

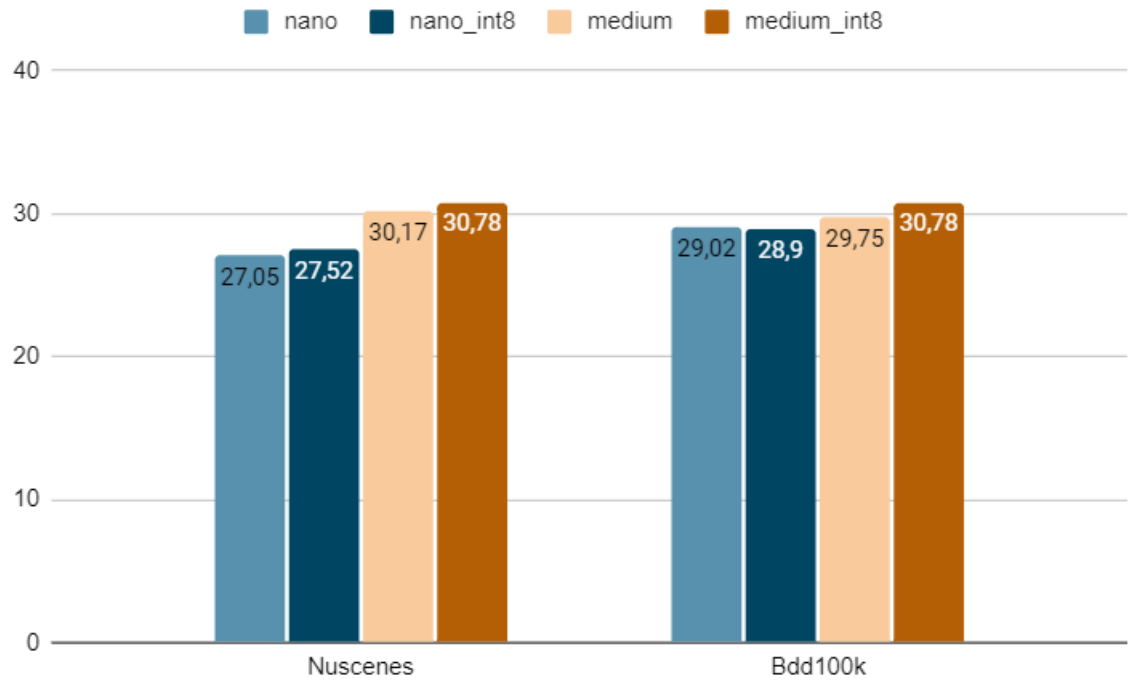

Figure 42: Resultados de precisión en local

8.3.2 Análisis de velocidad

En la figura 43 se muestran los resultados de la velocidad de inferencia. No es interesante ver su velocidad pura, ya que se está ejecutando en un contenedor de docker con potencia limitada, aun así, se puede comparar la velocidad entre ambos modelos.

También se puede observar una tendencia clara, son mejores los FPS obtenidos en los modelos BDD100k. Creemos que se debe a la inferior resolución, en este caso además es interesante la mejora de FPS al realizar la calibración a int8 que se mantiene constante en las diferentes pruebas mejorando la velocidad entre un 50% y 100% aproximadamente.

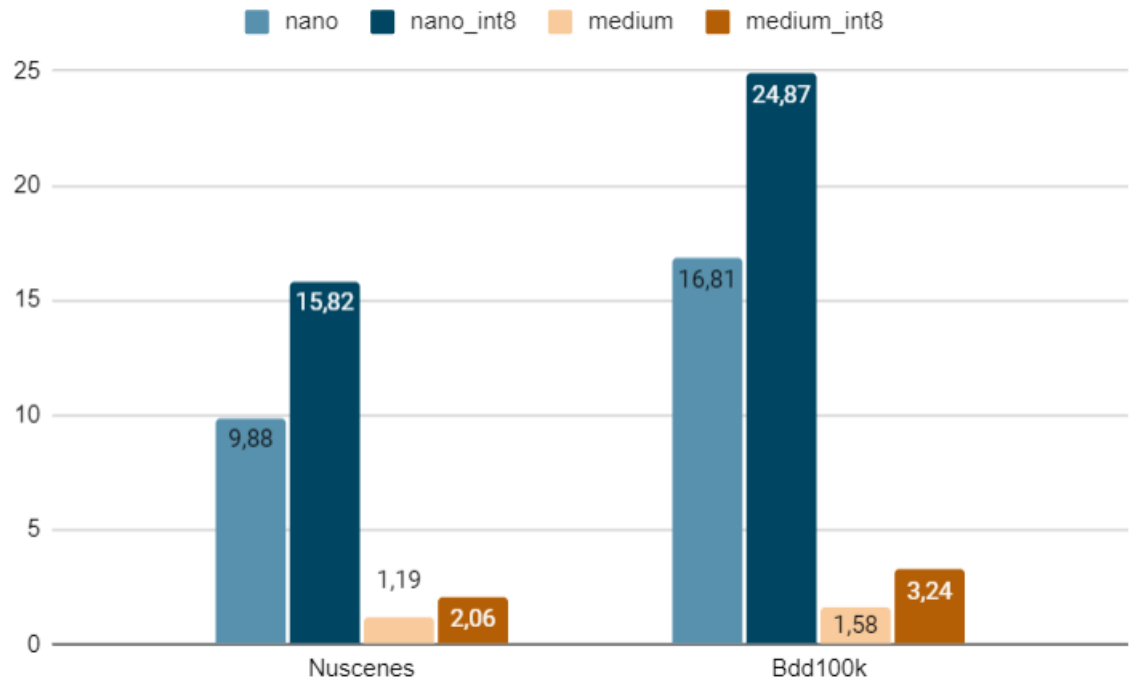

Figure 43: Resultados de velocidad en local

8.4 Resultados Cloud

8.4.1 Análisis del hardware

Para la inferencia de los modelos de la YOLOv5 utilizamos 3 dispositivos hardware diferentes en el cloud.

Un Intel Core i9-10900T [31] a 1.9GHz, un procesador muy potente de alta gama, pero de uso en computadores comunes, este procesador es interesante para ver los resultados en un procesador de alta gama, aunque el rendimiento/-coste no sea bueno. En cambio, con el Intel Atom x6425Re [30] a 1.90 GHz tiene un rendimiento inferior, pero también es interesante, ya que la relación rendimiento/coste es mejor.

Finalmente, se añade una tarjeta gráfica de 9ª generación que acompaña al i9 para comparar el rendimiento obtenido en aceleradores de propósito general con los procesadores comentados anteriormente.

8.4.2 Análisis de precisión

En la figura 44 se muestran los resultados obtenidos de la precisión de los 4 modelos en los diferentes hardwares.

Un detalle interesante a revisar en estos resultados es la perdida de precisión al realizar la calibración a int8. En algunos casos, como en el modelo nano entrenado con NUSCENES, la precisión decae bastante, en el i9 un 72,94% y un 70,45% en el Atom, en el resto de modelos la perdida de precisión no supera el 18,69%. Finalmente, destacar que la perdida de precisión de la GPU en formato fp16 es prácticamente nula.

		Atom x6425 10.9GHz		i9-10900T 1.9GHz		Gen9 HD graphics	
		-	int8	-	int8	-	fp16
BDD100K	nano	25,25	20,53	25,25	20,62	25,25	25,19
	medium	28,30	27,53	28,30	27,97	28,30	28,30
NUSCENES	nano	28,16	8,32	28,16	7,62	28,15	28,25
	medium	32,88	28,74	32,88	29,04	32,88	33,02

Figure 44: Resultados de precisión en mAP

8.4.3 Análisis de velocidad

En la figura 45 se muestran los resultados de velocidad de inferencia de los 4 modelos en diferente hardware.

Como era de esperar, los resultados del i9 son buenos, mientras que los de Atom son malos, incluso usando el modelo nano, La tarjeta gráfica obtiene un rendimiento similar aunque un poco inferior al i9. Cabe destacar que el rendimiento obtenido al calibrar a int8 es muy bueno, por lo general se llega a duplicar los FPS.

Es interesante observar como la GPU en formato fp16 ha mejorado sustancialmente su velocidad de inferencia, mientras que, como hemos visto antes, no se ha perdido precisión.

		Atom x6425 10.9GHz		i9-10900T 1.9GHz		Gen9 HD graphics	
		-	int8	-	int8	-	fp16
BDD100K	nano	7,15	12,15	86,80	141,06	63,01	90,60
	medium	0,76	1,62	9,78	17,90	7,37	13,13
NUSCENES	nano	1,76	2,97	17,66	35,24	17,15	28,46
	medium	0,19	0,40	2,38	4,30	1,70	3,35

Figure 45: Resultados de velocidad en FPS

8.5 Resultados en fotos

En la figura 46 se puede observar el resultado para una imagen obtenida en YOLOv5m con Bdd100k, tras exportarlo al workbench en una resolución de 640x640. Aunque alguna caja está desplazada un poco o repetida, consigue detectar los objetos de la imagen con bastante acierto.

En la figura 47 observamos al mismo modelo en otra foto más compleja con el sol de fondo, esto causa que solo se detecten 6 de los 8 objetos de la imagen.

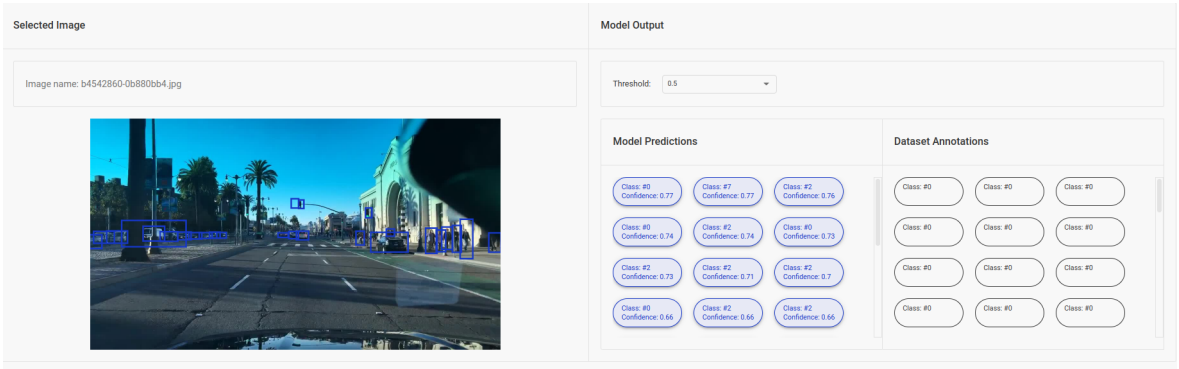

Figure 46: Resultado de YOLOv5m entrenado con Bdd100k en 640x640

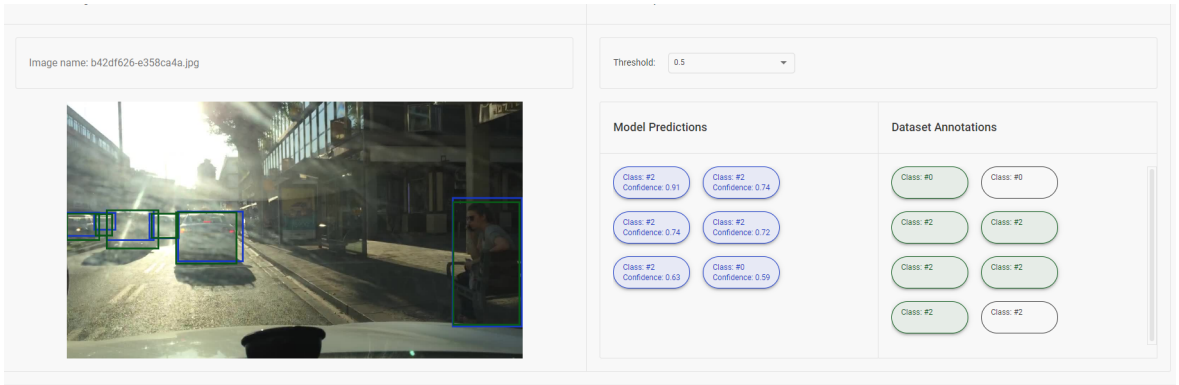

Figure 47: Resultado de YOLOv5m entrenado con Bdd100k en 640x640

La figura 50 es un resultado de YOLOv5m, con NUSCENES. En este caso reconoce bastante bien los 4 objetos que son 4 coches, aunque el mAP es bajo, ya que las cajas generadas no coinciden con la presentada en las anotaciones del dataset de validación.

Aunque en estos ejemplos funciona bien, habría que puntualizar que son los modelos medios, en los modelos nano la precisión baja mucho. La mejora en los modelos medium respecto a los nano es causada principalmente porque hemos observado que se detectan más objetos en los modelos medium, sin embargo esto no significa automáticamente que coincidan con la caja del objeto a detectar, está la causa principal por la que suponemos que el mAP es tan bajo por lo general el objeto detectado no está suficientemente bien ajustado.

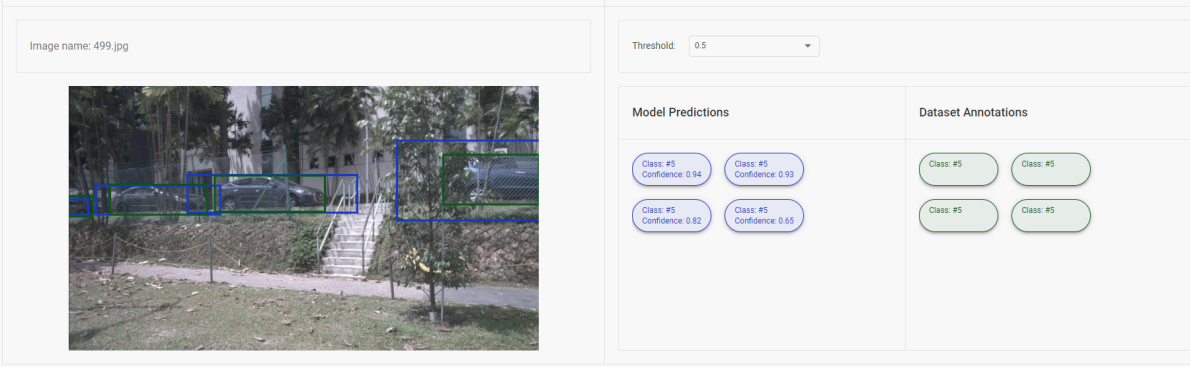

Figure 48: Resultado de YOLOv5m entrenado con NUSCENES en 1280x1280

En las siguientes 2 figuras se presentan los resultados obtenidos en una foto con buena visibilidad, en la cual el comportamiento es bastante bueno, aunque como sucede en la mayoría de las fotos faltan por detectar objetos de la vía.

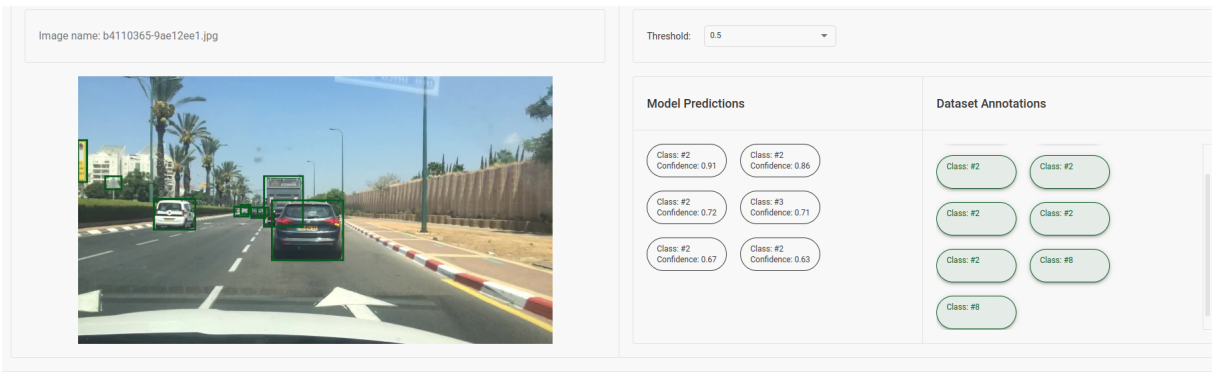

Figure 49: Objetos a detectar

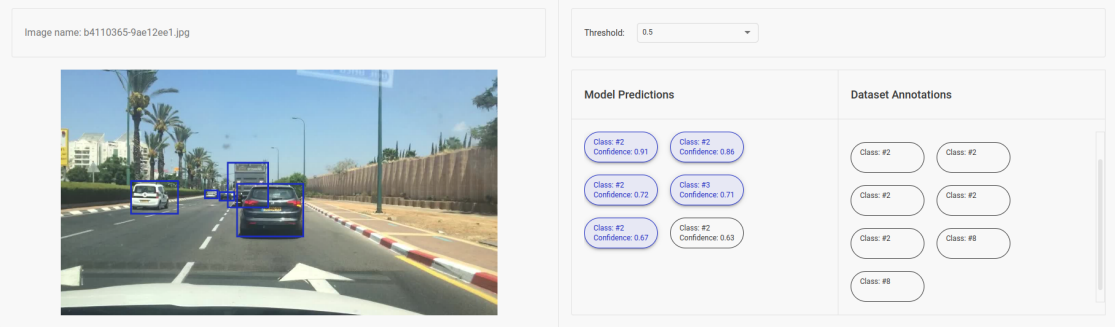

Figure 50: Resultado de YOLOv5m entrenado con Bdd100k en 640x640

En las siguientes figuras 51, 52 se presenta el modelo medio de la misma foto en 2 resoluciones diferentes y aunque como comentaba anteriormente el comportamiento del modelo medio es mejor porque detecta más objetos, en este caso en concreto el comportamiento es un poco errático y se pueden ver como no se ajustan muy bien las cajas a algunos objetos detectados.

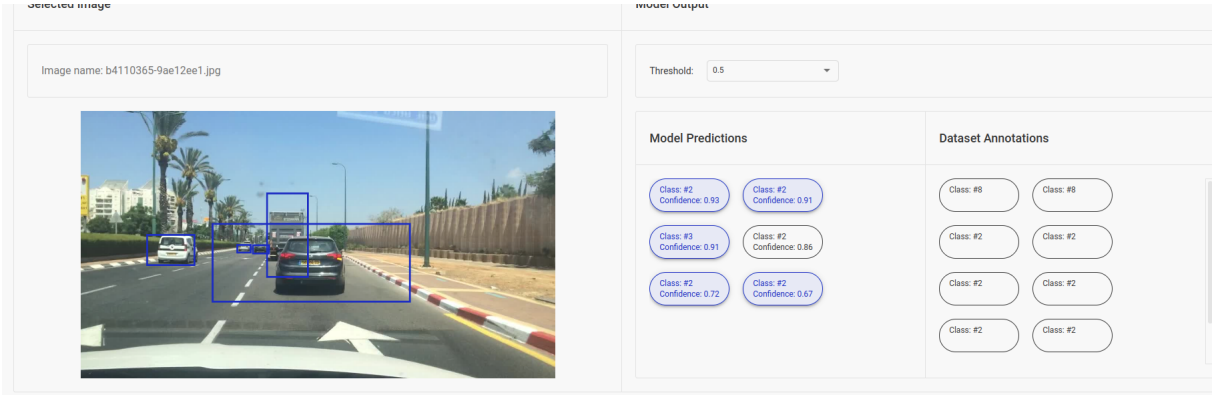

Figure 51: Resultado de YOLOv5m entrenado con Bdd100k en 640x640

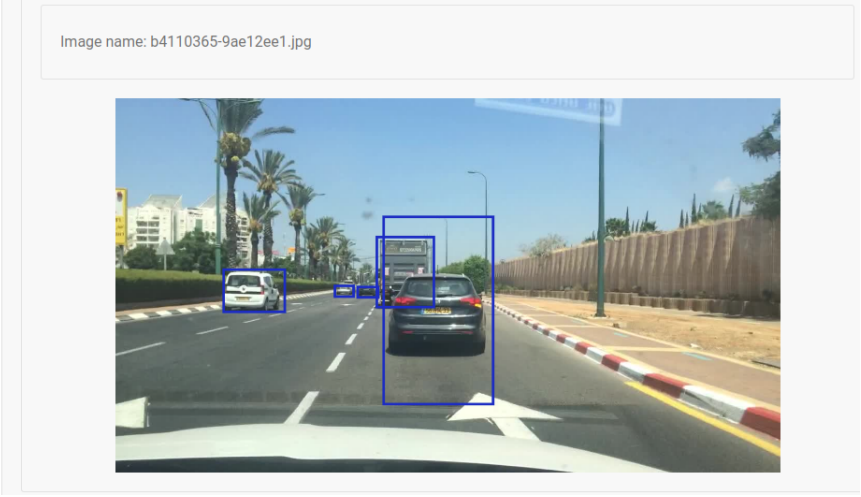

Figure 52: Resultado de YOLOv5m entrenado con Bdd100k en 1280x1280

Y aunque en fotos con buena visibilidad el comportamiento no es óptimo, se esperaría que en fotos con baja visibilidad como el ejemplo que se presenta en la figura 54 el funcionamiento fuese muy malo, en cambio en este caso se detectan bastante bien los 4 objetos que a simple vista ya es complicado identificarlos.

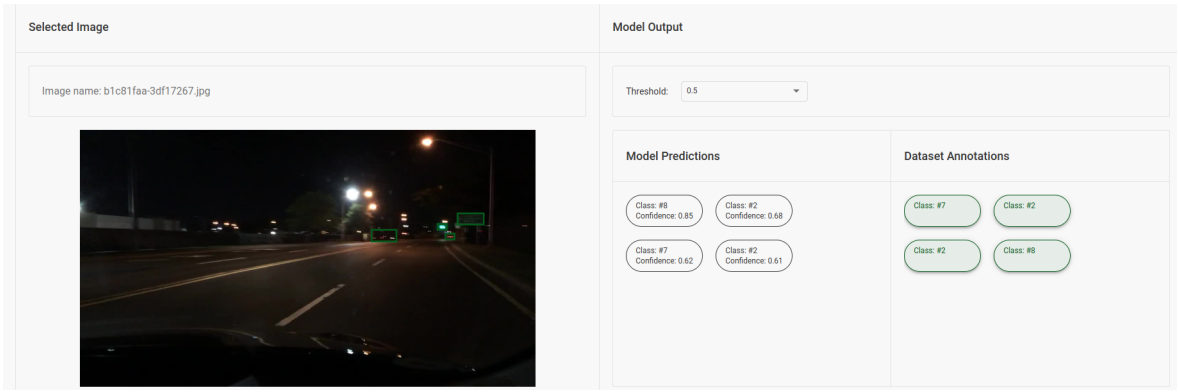

Figure 53: Objetos a detectar

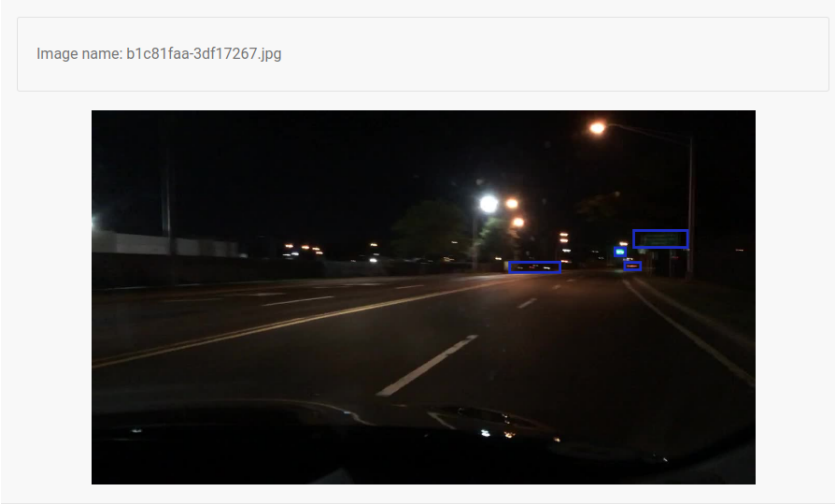

Figure 54: Resultado de YOLOv5m entrenado con Bdd100k en 1280x1280

9 Conclusiones

Una vez llegados a este punto podemos evaluar si se cumplieron los objetivos planteados para este trabajo. Si los resultados de las pruebas de inferencia y precisión son aceptables, si el rendimiento de la YOLOv5 para detección de objetos en el ámbito de la automoción es bueno y si la optimización de la red con OpenVINO es satisfactoria.

Para determinar si la velocidad de inferencia es suficientemente buena para la conducción autónoma, deberíamos obtener velocidades de entorno a 30 FPS, 20 FPS como poco para que sea capaz de detectar los objetos a tiempo cuando el vehículo se mueve a altas velocidades.

Cabe destacar que como las pruebas antes de OpenVINO y tras la optimización se ejecutan en distinto hardware, no es adecuado comparar los FPS para saber si la optimización aumentó los FPS o los disminuyó.

- Si observamos los resultados antes de introducir la red en OpenVINO, todos los modelos obtienen la velocidad requerida sobradamente.
- Los resultados tras la optimización indican que el modelo nano es suficientemente rápido y que el modelo medium necesitaría de hardware más potente para alcanzar el mínimo.

La precisión antes de introducir la red neuronal en OpenVINO es buena con entorno a 60% mAP. Con un entrenamiento más extenso seguramente se podría aumentar este valor.

Aun así, al introducir la red en OpenVINO nos encontramos con una perdida de precisión de cerca del 50%. Creemos que la causa es que OpenVINO no tiene soporte real para la YOLOv5 lo que ha causado problemas a la hora de obtener resultados. Hemos tenido que hacerla pasar como YOLOv4 y YOLOv3 y hemos tenido que acotar los datasets de validación a 500 imágenes.

Teniendo en cuenta el rendimiento antes de la optimización, creemos que el rendimiento de YOLOv5 es adecuado para la conducción autónoma. Además,

los distintos modelos son interesantes, ya que ayudan a calibrar la velocidad/-precisión al nivel que se necesite. Sería interesante comprobar los resultados del entrenamiento de los modelos large y extraLarge para observar si obtienen la velocidad suficiente, porque es de esperar que su precisión sea aún mayor.

Entre los modelos estudiados, el modelo small es el más interesante, ya que obtiene una precisión muy similar al modelo medium (63% y 61% mAP) mientras que funciona aproximadamente un 50% más rápido (104 y 147 FPS).

Finalmente, la optimización en OpenVINO no ha sido satisfactoria, ya que simplemente la transformación a formato IR ha empeorado en gran medida la precisión. Aun así, podemos considerar la calibración int8 como un acierto, porque aumenta los FPS en gran medida (en torno a un 50%) mientras que apenas se reduce la precisión del modelo si lo comparamos con el mismo modelo en OpenVINO.

9.1 Continuación del proyecto

El proyecto puede continuar de varias formas:

Repetir las pruebas realizadas con las versiones YOLOv6 y YOLOv7, nuevos modelos de YOLO publicados recientemente que prometen mejor rendimiento y más velocidad. No están soportadas por OpenVINO, pero se pueden buscar alternativas para evaluar su rendimiento, aunque esto supone no poder inferir en distinto hardware en el cloud de Intel, ni poder utilizar las opciones de optimización de OpenVINO.

También se puede enfocar este mismo trabajo a más pruebas en hardware local, como obtener los resultados en una Jetson Nano de NVIDIA y observar el rendimiento en dispositivos de baja potencia.

Por último, otra posible continuación del proyecto puede ser enfocarlo a redes neuronales soportadas en OpenVINO oficialmente. A pesar de los problemas de precisión en OpenVINO, la calibración int8 ha aumentado la velocidad en gran medida y quizás con modelos que OpenVINO soporte se pueden obtener mejores

resultados de precisión y experimentar más con sus opciones.

10 Conclusions

At this stage of the project we can evaluate if the objectives have been fulfilled. We will conclude if the results from the inference speed and precision are acceptable, if YOLOv5 performance for object detection in a self driving environment is good and if OpenVINO and int8 optimization is satisfactory.

To consider the inference speed good enough for self-driving We should get around 30 FPS, 20 FPS as a minimum, The CNN must be capable of detect objects on time when the vehicle is moving at high speeds.

We have to highlight that the testing before and after OpenVINO optimization are executed in different harware, so it's not appropiate to compare FPS to know if the optimization increased or decrese the FPS.

- If we analyze the results before introduce the modelo in OpenVINO, all models get more than required speed.
- The results after optimization indicate that nano model is fast enought and that medium model needs powerful harware to reach the minimum.

Acurracy before introducing the CNN in OpenVINO is good with around 60% mAP. With more extensive training this result could be improved.

Even so, when introducing the neural networks in OpenVINO's workbench, we found an accuaracy lose of about 50%. We think the cause is that OpenVINO does not have real support for YOLOv5 and has caused problems when we run inferences. We have had to run it as YOLOv4 and YOLOv3 and we have had to limit the validation datasets to 500 images.

Considering the performance before optimization, we belive that the preformance of YOLOv5 is suitable for autonomous driving. In addition, the different models are intereseting as they help to calibrate the speed/acurracy to the level needed. It would be interesting to check the training result of the large and extraLarge models to see if they obtain enough speed because their accuaracy is expected to be even higher.

If we look at the studied models pre OpenVINO, the small model is the most interesting one because it has a similar precision to the medium model (63% y 61% mAP) while working moreless 50% faster (104 y 147 FPS).

Finally, the optimization in OpenVINO has not been satisfactory because the transformation to IR format has worsened the accuracy to a great extent. Even so, we can consider the int8 calibration as a success since it increases the FPS greatly (around 50%) while hardly reducing the mAP of the model if we compere it with the same model in OpenVINO.

10.1 Future work

The project can continue in several ways:

Repeat the tests performed with YOLOv6 and YOLOv7, new YOLO models recently released that promise better performance and more speed.They are not supported by OpendVINO but alternatives can be looked upon. Although this means not being able to infer on different hardware in the Intel cloud, and not being able to use OpenVINO optimization options

It is also possible to focus this same work on more local harware tests, such as getting the results on an NVIDIA Jetson Nano and observing the performance on low-power devices.

Finally, another possible continuation of the project may be to focus on officialy OpenVINO's supported neural networks. Despite the acurracy problems in OpenVINO, int8 calibration has greatly increased the speed and perhaps with OpenVINO supported models better accuracy results can be obtained.

11 Contribuciones

11.1 Marco Expósito Pérez

Marco Expósito Pérez ha contribuido en los siguientes puntos:

- Buscar los datasets que se van a utilizar durante el proyecto. Acabamos utilizando BDD100k y NUSCENES de entre los que encontramos.
- Buscar información sobre los distintos modelos de YOLOv5 y las cualidades de cada uno.
- Experimentar con el entrenamiento de distintos modelos de YOLOv5 en local. Finalmente, vimos que en local tarda demasiado en entrenarse.
- Utilizar el entorno de OpenVINO en local y para optimizar redes neuronales con un formato compatible con OpenVINO.
- Investigar cuáles son los parámetros óptimos de entrenamiento para los modelos de YOLOv5. De aquí obtuvimos que 300 épocas es lo recomendable aunque demasiado largo y que 200 es más adecuado.
- Entrenar una red neuronal con un dataset sencillo en las máquinas virtuales de Intel. Los resultados no fueron satisfactorios, ya que el entrenamiento tardaba demasiado, aunque fuese en una máquina remota.
- Pedir acceso a una máquina remota potente de la facultad y experimentar entrenando la red en ella. Asi descubrimos que el entrenamiento del modelo más grande, YOLOv5X era demasiado largo incluso para esta máquina, asi que desechamos el modelo.
- Analizar el rendimiento de los modelos entrenados en wandb durante el entrenamiento y al final del mismo.
- Intentar optimizar los modelos exportados utilizando el codigo de OpenVINO en local. No dio resultado este proceso debido a los problemas de compatibilidad.

- Escribir las secciones de este documento correspondiente a la 1-introducción, 4-YOLOv5, 5-Datasets, 6-Metricas, 11-Contribuciones.
- Contribuir a la sección 12-bibliografia y repasar las secciones escritas por Ioan Marian Tulai.
- Traducir al inglés los apartados que se requieren.

11.2 Ioan Marian Tulai

Ioan Marian Tulai ha contribuido en los siguientes puntos:

- Buscar los datasets que se van a utilizar durante el proyecto. Acabamos utilizando BDD100k y NUSCENES de entre los que encontramos.
- Experimentar con el entrenamiento de distintos modelos de YOLOv5 en local. Finalmente, vimos que en local tarda demasiado en entrenarse.
- Utilizar el entorno de OpenVINO en local y para optimizar redes neuronales con un formato compatible con OpenVINO. Era bastante sencillo, ya que todo era compatible
- Crear un script en python para traducir el formato del dataset BDD100k a formato YOLOv5.
- Buscar una forma de traducir el formato del dataset NUSCENES a formato YOLOv5 ya que NUSCENES es más complejo que BDD100k. El resultado es que encontramos un script en python para hacerlo.
- Exportar los modelos de la YOLOv5 ya entrenados a ONNX y a formato IR.
- Optimizar los modelos exportados utilizando el work-bench de OpenVINO en local. Funcionó al hacer pasar la YOLOv5 como si fuera la YOLOv4.
- Experimentar con las formas de optimizar, distinto hardware y obtener los resultados de precisión y velocidad para cada combinación. Aquí por ejemplo, se descubrió que invertir los canales de color mejoraban el rendimiento de la red neuronal.
- Ordenar los resultados y representarlos en gráficas para visualizarse de forma más sencilla.
- Crear la estructura inicial de este documento de este documento.
- Escribir las secciones de este documento correspondiente a las: 3-Bases teóricas, 7-Metodología, 8-Análisis de resultados y 9-Conclusiones.

- Contribuir a la sección 12-bibliografia y repasar las secciones escritas por Marco Expósito Pérez.

Algunas de las contribuciones son similares o están repetidas en los dos alumnos. Esto ocurre porque al inicio del proyecto era necesario que ambos nos familiarizásemos con las herramientas a utilizar. Posteriormente, optimizar los modelos tras exportarlos creaba muchos problemas de compatibilidad, así que ambos, por distintos caminos, intentamos resolverlos.

12 Bibliografia

[1] Titlemax, *History of the Autonomous Car*, https://www.titlemax.com/resources/history-of-the-autonomous-car/, s.f..

[2] Wikipedia, *Ernt Dickmans*, https://en.wikipedia.org/wiki/Ernst_Dickmanns, s.f..

[3] Ultralytics, *YOLOv5*, https://github.com/ultralytics/yolov5

[4] Ultralytics, *YOLOv5 Documentation*, https://docs.ultralytics.com

[5] Lutz Roeder, *Netron*, https://github.com/lutzroeder/netron/

[6] OpenVINO Toolkit, *OpenVINO*, https://github.com/openvinotoolkit/openvino

[7] Wanbd.Ia, *The developer-first MLOps plataform*, https://wandb.ai

[8] Jose David Villanueva Garcia, *Redes neuronales desde cero*, https://www.iartificial.net/redes-neuronales-desde-cero-i-introduccion/, 2020.

[9] Wikipedia, *Red neuronal artificial*, https://es.wikipedia.org/wiki/Red_neuronal_artificial, s.f..

[10] Na8, *¿Cómo funcionan las Convolutional Neural Networks? Visión por Ordenador*, https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/, 2018.

[11] Madhushree Basavarajaiah, *6 basic things to know about Convolution*, https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411, 2019.

[12] Richmond Alake, *(You Should) Understanding Sub-Sampling Layers Within Deep Learning*https://towardsdatascience.com/you-should-understand-sub-sampling-layers-within-deep-learning-b51016acd551, 2020.

[13] Na8, *Modelos de Detección de Objetos*, https://www.aprendemachinelearning.com/modelos-de-deteccion-de-objetos, 2020.

[14] Ross Girshick, Jeff Donahue, Trevor Darrel, Jitendra Malik, UC Berklely, *Rich feature hierarchies for accurate object detection and semantic segmentation*, https://arxiv.org/pdf/1311.2524.pdf

[15] Rohith Gandhi, *R-CNN,Fast R-CNN,Faster R-CNN,YOLO-Object detection Algorithms*, https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e, 2018.

[16] Ahmed Frawzy Gad, *Faster R-CNN Explained for Object Detection Tasks*, https://blog.paperspace.com/faster-r-cnn-explained-object-detection/, 2020.

[17] S.A.Sanchez, H.J.Romero y A.D.Morales, *A review: Comparasion of performance metrics of pretrained models for object detection using the TensorFlow framework*,

https://www.researchgate.net/publication/342570032_A_review_Comparison_of

_performance_metrics_of_pretrained_models_for_object_detection_using_the_TensorFlow_framework

[18] Joseph Redmon,Santosh Divvala, Ross Girshick,Ali Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, https://arxiv.org/pdf/1506.02640v5.pdf

[19] Coco, *Coco Common Objects in Context*, https://cocodataset.org/

[20] Fisher Yu,Haofeng Chen,Xin Wang, Wenqi Xian, Yingying Chen, Fangchen Liu, Vashisht Madhavan, Trevor Darrell, UC Berkeley, Cornell University, UC San Diego, Element,Inc., *BDD100k:A Diverse Driving Dataset for Heterogeneous Miltitask Learning*, https://arxiv.org/abs/1805.04687

[21] Holger Caesar,Varun Bankiti,Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, Oscar Beijbom, nuTonomy:an APTIV company, *nuScenes: A mulyimodal dataset for autonomous driving*, https://arxiv.org/abs/1903.11027

[22] Sarang Narkhede, *Understanding Confusion Matrix*, https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62, 2018.

[23] Deval Shah, *Mean Average Precision(mAP) Explained: Everything You Nedd to Know*, https://www.v7labs.com/blog/mean-average-precision, 2022

[24] Intel, *OpenVINO* https://docs.openvino.ai/latest/index.html, 2021.

[25] Jose Martinez Heras, *Precision,Recall,F1,Accuracy en clasificación*, https://www.iartificial.net/precision-recall-f1-accuracy-en-clasificacion/, 2020

[26] Ioan Marian Tulai, *YoloV5_conversion_scripts*, https://github.com/marian1010/YoloV5_conversion_scripts, 2022.

[27] Sergio Gil Gavela, *annotations-and-mAP-scripts*, https://github.com/sgavela/annotations-and-mAP-sripts

[28] WelkinU, *yolov5*, https://github.com/WelkinU/yolov5/tree/master/nuscenes

[29] Docker Inc, *Docker docs*, https://docs.docker.com/

[30] Intel, *Intel Atom x6425RE Processor* https://www.intel.com/content/www/us/en/products/sku/207899/intel-atom-x6425re-processor-1-5m-cache-1-90-ghz/specifications.html

[31] Intel, *Intel Core i9-10900T Processor* ,https://www.intel.com/content/www/us/en/products/sku/199324/intel-core-i910900t-processor-20m-cache-up-to-4-60-ghz/specifications.html

[32] Paperswithcode, *Leaky ReLU*, https://paperswithcode.com/method/leaky-relu, s.f..