Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

IDENTIFICACIÓN DE ZONA CONDUCIBLE DE CARRETERA MEDIANTE VISIÓN Y "DEEP LEARNING"

Autor: Jonathan Rober Moncada Gavilanes

Tutor: Manuel Ocaña Miguel

2022

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

IDENTIFICACIÓN DE ZONA CONDUCIBLE DE CARRETERA MEDIANTE VISIÓN Y "DEEP LEARNING"

Autor: Jonathan Rober Moncada Gavilanes

Tutor: Manuel Ocaña Miguel

Tribunal:

Presidente: Pedro Alfonso Revenga de Toro

Vocal 1º: Ernesto Martín Gorostiza

Vocal 2º: Manuel Ocaña Miguel

Fecha de depósito: 8 de Diciembre 2022

A mi familia, mi pareja y mis compañeros de carrera que me han acompañado en todo este largo camino...

"Un genio se hace con un 1 % de talento y 99 % de trabajo"
Albert Einstein.

Agradecimientos

Este trabajo es el fruto de muchas horas de dedicación, sé que ha sido duro para los de mi alrededor tener que aguantarme durante todo este tiempo hablándoles de algo que me fascinaba pero que entenderlo se veía un poco complejo.

Primero de todo, necesito agradecer a mi familia, que me ha apoyado durante todo este tiempo mientras terminaba la carrera. Al final, todas esas noches que me quedaba hasta tardísimo delante del ordenador han dado sus frutos.

Además, creo que nadie puede conseguir una meta importante sin tener a compañeros a su lado. Por lo que merecen un agradecimiento enorme mis compañeros de carrera Álvaro García, Álvaro Monzón y Alejandro Pérez, que, gracias a sus explicaciones, ayuda y su ánimo en muchos momentos no podría haber terminado este camino de mi vida. Pero sin duda, también quiero agradecerles que hayamos podido compartir tan buenos momentos, sobre todo a mi compañero Álvaro Monzón, que me ha acompañado en mis peores y mejores momentos, y espero sinceramente poder compartir muchos más, tanto con él, como con mis otros compañeros.

Por último, solo me queda agradecer a una persona en la que siempre puedo confiar, en la que en cada momento malo está ahí para intentar levantarme, con la que celebro cada momento bueno y con la que quiero seguir compartiendo miles de logros y momentos más junto a ella. Nerea, gracias por apoyarme durante todo este largo camino y sé que una parte de esto te lo debo a ti y que me hiciste cambiar de mentalidad en el momento adecuado para bien.

Muchas gracias a cada uno de vosotros, ¡se os quiere!

Resumen

En este trabajo se plantea detectar las líneas de carretera y zonas conducibles de los carriles por los que circula un vehículo. Para ello se utilizará la librería YOLOP, un modelo de Deep Learning, o aprendizaje profundo, disponible en GitHub.

Durante la realización de este trabajo de fin de grado, se irá probando el funcionamiento del modelo en local, se creará un nuevo Dataset basado en imágenes del simulador CARLA para poder realizar un entrenamiento del modelo y por último se creará finalmente una interfaz web para poner a disposición de cualquier usuario el probar el modelo.

Palabras clave: Aprendizaje profundo, aprendizaje multitarea, detección de objetos con tráfico, segmentación del área de conducción, detección de carril.

Abstract

This paper aims to detect the different drivable lines and areas of the road where vehicle is moving. For this purpose, YOLOP library will be used, a model of Deep Learning, available on GitHub.

Along with this Degree's Final Thesis, the model is going to be tested locally, a new Dataset based on images of the CARLA simulator will be created to be able to carry out a training of the model and finally a Web interface will be created to make it available to any user for testing purposes.

Keywords: Deep learning, multitask learning, traffic object detection, drivable area segmentation, lane detection.

Resumen extendido

En este trabajo se plantea detectar las distintas líneas y zonas conducibles de los carriles por los que circula un vehículo. Para ello se utilizará la librería YOLOP, un modelo de Deep Learning, o aprendizaje profundo, desarrollado y disponible en GitHub.

El primer paso consistirá en estudiar e implementar dicha librería en una máquina local, para que realmente se pueda validar su funcionamiento.

A continuación, se creará nuestro propio Dataset, basado en imágenes sintéticas procedentes del simulador hiper-realista CARLA, para mejorarlo y poder comparar el modelo actual que se tiene, con uno que se entrenará con el nuevo Dataset de datos sintéticos. Gracias a este nuevo entrenamiento se puede observar que el modelo es capaz de adaptarse y mejorar gracias a las nuevas imágenes de simulación, con lo que se podrá llegar a utilizar con los diferentes simuladores de conducción autónoma que hay disponibles en el mercado.

Además, se creará una página Web, en la que se desplegará el modelo para que pueda ser testeado por cualquier persona de manera fácil e intuitiva.

Por último, una vez realizadas todas las pruebas del modelo, se pasará a probarlo en modo real empleando para ello imágenes procedentes de una cámara embarcada en el vehículo autónomo disponible en el grupo de investigación RobeSafe.

Con esto, se quiere probar el funcionamiento correcto del modelo en distintos entornos y cómo es capaz de realizar una inferencia en tiempo real, con lo que se puede llegar a obtener imágenes y datos que ayuden a tomar decisiones a la hora de la conducción autónoma.

Palabras clave: Aprendizaje profundo, aprendizaje multitarea, detección de objetos con tráfico, segmentación del área de conducción, detección de carril.

Extended Abstract

In this work, it is proposed to detect the different lines and drivable areas of the lanes through which a vehicle circulates. For this, the YOLOP library will be used, a Deep Learning model, developed and available on GitHub.

The first step will consist of studying and implementing YOLOP library on a local machine, so that its operation can really be validated.

Next, our own Dataset will be created, based on synthetic images from the CARLA hyper-realistic simulator, to improve it and be able to compare the current model with one that will be trained with the new synthetic data Dataset. Thanks to this new training, it can be seen that the model is capable of adapting and improving thanks to the new simulation images, with which it can be used with the different autonomous driving simulators that are available on the market.

In addition, a Web page will be developed, in which the model will be installed so that it can be tested by anyone in an easy and intuitive way.

Once all the tests of the model have been carried out, it will be tested in real mode, using images from a camera on board of an autonomous vehicle available from the RobeSafe research group.

With all this, we want to test the correct operation of the model in different environments and how it is capable of making an inference in real time, with which it is possible to obtain images and data that help make decisions when it comes to autonomous driving.

Keywords: Deep learning, multitask learning, traffic object detection, drivable area segmentation, lane detection.

Índice general

Índice de figuras

Índice de códigos

Capítulo 1

Introducción

1.1 Introducción

Actualmente el mundo de la conducción autónoma está en continua expansión y se compone de diferentes líneas de investigación: percepción, control autónomo, localización, mapeado, toma de decisiones.

Así mismo, se puede definir la conducción autónoma como una modalidad de conducción que consiste en que el vehículo llegue a conducirse por sí mismo, sin necesidad de un control activo del conductor.

Estos vehículos estarán compuestos por sensores, procesadores, actuadores y software que ayudarán en cada una de las líneas de investigación que componen a los Sistemas Automatizados de Conducción(ADS).

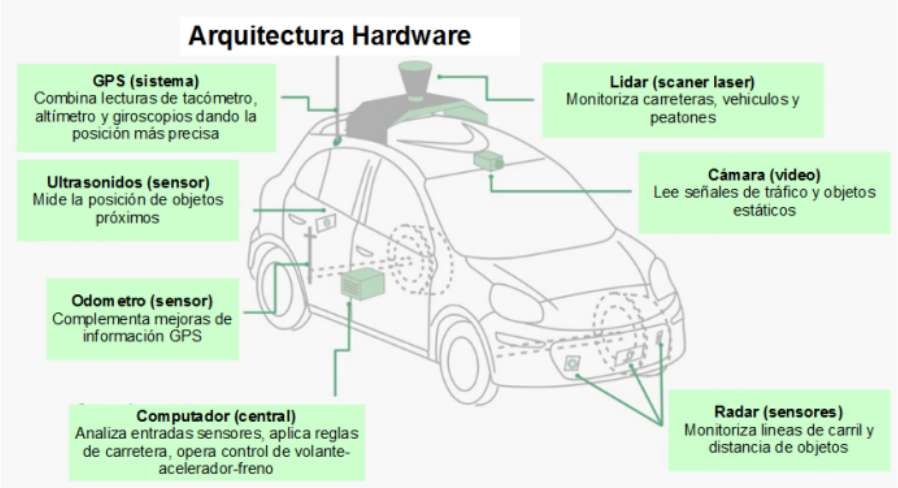


Figura 1.1: Sensores Sistemas Automatizados de Conducción(ADS).

La gran mayoría de los vehículos nuevos ya cuentan con tecnologías que ayudan a evitar que el vehículo se desvíe a los carriles adyacentes o hacer cambios de carril inseguros. También se nos advierte de otros vehículos o personas detrás de nosotros cuando damos marcha atrás, otros incluso cuentan con un sistema de frenado automático, entre muchas otras tecnologías.

Pero cuando se habla de los vehículos autónomos muchas veces se expresa un miedo en cuanto a los temas de seguridad. Para superar esta barrera se ha de ir avanzando a través de los de los 6 niveles de conducción autónoma que existen. Esto incluye todo, desde ninguna automatización, hasta la autonomía completa. Como se acaba de mencionar, todos los vehículos se catalogan dentro de un nivel concreto

de autonomía en función del mayor o menor grado con que pueden asistir al conductor. En total hay 6 niveles de conducción autónoma:

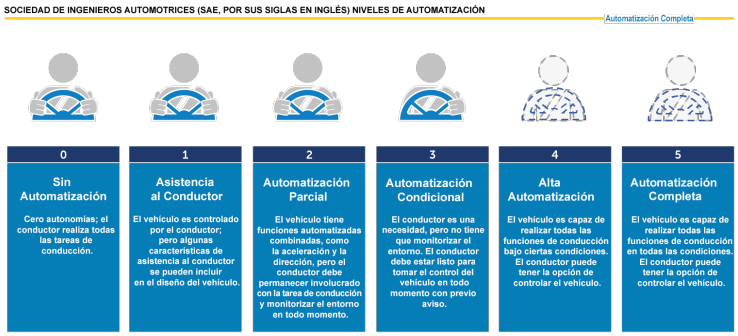


Figura 1.2: Niveles de Autonomía.

- Nivel 0: Sin automatización.
  - El conductor realiza todas las tareas que engloba la conducción. Se incluyen en este nivel los vehículos que tengan asistentes a la conducción cuyas funciones no implican el control lateral o longitudinal del vehículo.
- Nivel 1: Asistencia en la conducción.
  - Los vehículos que se incluyen en este nivel tienen asistentes a la conducción que pueden controlar el movimiento lateral o longitudinal, nunca los dos a la vez. Dispositivos de este tipo son, por ejemplo, el control de crucero, que ayuda a mantener la velocidad constante en el vehículo o un asistente de ayuda al aparcamiento en el que solo se pueda accionar el volante, pudiendo el conductor accionar los pedales.
- Nivel 2: Automatización parcial
  - En este nivel, los vehículos tienen asistentes a la conducción que pueden controlar el movimiento lateral y también el longitudinal. El conductor es el único responsable de la conducción porque estos asistentes tienen un ámbito de uso limitado y además el vehículo no está preparado para la detección de obstáculos imprevistos. Un sistema de mantenimiento en el centro del carril, trabajando junto a un programador de velocidad es un ejemplo de automatización de nivel 2. También lo es un asistente de conducción en atascos y algunos de ayuda al aparcamiento que operan el volante y los pedales. El primer coche con un nivel 2 fue el Mercedes-Benz Clase S 2013. Los modelos de Tesla, como el Model X, también son nivel 2, al igual que otros modelos de las marcas Audi, BMW, DS, Ford, Jaguar, Hyundai, Land Rover, Lexus, Nissan, Peugeot, SEAT y Volvo, entre otras. A partir del Nivel 3 hay un salto cualitativo grande porque el vehículo tiene un sistema de conducción automatizada que puede realizar la totalidad de las tareas que engloba la conducción.
- Nivel 3: Automatización condicionada.
  - A partir de este nivel, el conductor puede decidir que el sistema de conducción autónoma tome el control sobre el vehículo y realice todas las funciones de la conducción con unas ciertas limitaciones. En el Nivel 3, el conductor no necesita supervisar la conducción, pero sí ha de estar alerta e intervenir cuando el sistema lo requiera. El sistema se desactivará de inmediato cuando el conductor lo solicite y también demanda la intervención del conductor cuando éste detecte algún fallo en su funcionamiento o cuando los límites de su ámbito de actuación van

a ser sobrepasados. Además, cuando el sistema falle se genera un aviso y el sistema da un margen de tiempo, de varios segundos, para que el conductor retome el control (Handover).

- Nivel 4: Automatización elevada.
  - En este nivel, el sistema de conducción automatizada puede guiar el vehículo de forma sostenida en el tiempo sin la expectativa de que el conductor responda ante una demanda de intervención, salvo cuando se encuentre fuera de su ámbito de funcionamiento. El sistema está preparado para actuar ante una situación imprevista de peligro y para realizar sin ayuda el conjunto de acciones más segura posible (situación de mínimo riesgo). El sistema de conducción automatizada de los vehículos de nivel 4 también tiene, al igual que en el nivel 3, un ámbito de funcionamiento limitado. Mientras el vehículo se encuentre dentro de ese ámbito, el conductor puede elegir cuándo pone en funcionamiento el sistema y también puede solicitar la desconexión para retomar el control del vehículo, aunque en este caso, a diferencia del nivel 3, el sistema puede demorar su desconexión si lo considera necesario. Otra diferencia entre ambos niveles es que, en el tres, el tiempo que da el coche al conductor para que retome la conducción es de segundos y en el cuatro puede ser de minutos. Además, en el caso de que el conductor no lo haga, el vehículo será capaz de detenerse en una zona segura, ya que en ese caso no se espera que el conductor retome de vuelta el control.
- Nivel 5: Automatización completa.
  - En el nivel 5, el sistema de conducción automatizada (ADS; automated driving system) tiene un ámbito de funcionamiento que comprende todas las condiciones y lugares por los que podría circular un conductor humano. Esto quiere decir que para el ADS no se diseñan limitaciones geográficas o climatológicas y, por tanto, que el vehículo puede prescindir de un conductor y de elementos como el volante o los pedales. Aunque quizás parezca razonable que la industria vaya escalando de un nivel a otro hasta lograr la autonomía total, la complejidad técnica y la variabilidad estratégica que entraña el desafío del coche autónomo han propiciado un proceso de diversificación empresarial profunda que da respuesta a todas las nuevas oportunidades de negocio. Algunas empresas trabajan para satisfacer la demanda de movilidad compartida, cada vez mayor en ámbitos urbanos. En este sentido, uno de los objetivos principales de empresas como Waymo (la división de vehículos autónomos de Google) y Uber, por ejemplo, es desarrollar vehículos de nivel 5 geofenced (o lo que es lo mismo, con un ámbito geográfico de actuación limitado, pero sin necesidad de conductor) que puedan operar como taxis urbanos autónomos. A finales de 2018, Waymo puso en marcha un proyecto piloto mediante el que ofrece este servicio en una zona de la ciudad de Phoenix,Arizona. Algunas compañías están centrando sus esfuerzos en delimitar otros ámbitos de funcionamiento en modo autónomo para coches y camiones, como determinadas rutas por autopista. Otras, como por ejemplo Tesla o Audi, también trabajan para llevar las soluciones tecnológicas de la conducción autónoma al vehículo particular en un sentido más general, con el objetivo de evolucionar hacia el nivel 5, aunque no se espera que este sea una realidad a corto o medio plazo.

Y con respecto con lo que a las marcas se refiere, como se ha ido mencionando, ahora mismo las que tienen mayor impacto en el mercado de la conducción autónoma son marcas como **Tesla**,**Waymo (Google)**,**Lyft (Toyota)**,**Aurora**,**Cruise (General Motors)**,**Mobileye (Intel)**,**Argo AI (Ford)** y **Nvidia**, entre otras muchas.

En resumen, el mundo de la conducción autónoma es muy activo en investigación y está en continua expansión, y se espera que para principios de 2030 se consiga llegar al nivel 5 de automatización.

1.2 Motivación

La idea del proyecto surge en el seno del grupo de investigación RobeSafe [Robesafe(2022)], y más concretamente en el proyecto *Tech4Age* [Robotics und eSafety(2022)] de ayuda a la conducción a personas mayores, ya que existe la necesidad de mejorar la localización del vehículo autónomo dentro del carril.

En un primer momento, el grupo Robesafe estuvo trabajando en el proyecto *SmartElderlyCar* (2016-2018), desarrollando un primer prototipo de coche con algunas funciones autónomas. Basado en los resultados previos en ese proyecto, el principal objetivo de la propuesta de *Techs4AgeCar* es investigar técnicas para un nuevo concepto de automóvil eléctrico automático (*AgeCar*), capaz de ayudar a los conductores mayores con diferentes niveles de automatización, según las necesidades de estos y teniendo en cuenta la seguridad y la aceptación del usuario como puntos clave de desarrollo.

El 24 de julio 2020, el grupo de investigación RobeSafe realizó una demo del proyecto *Tech4AgeCar* en el Campus Externo de la Universidad de Alcalá. EL automóvil eléctrico autónomo, basado en una plataforma *TABBY EVO* de Open Motors®, demostró sus nuevas habilidades mejoradas en términos de control basado en el seguimiento de waypoints, localización basada en EKF usando dispositivos GNSS y Wheel Odometry y Drive-By-Wire sistema, todos ellos integrados mediante comunicaciones ROS. Además, se da un paso adelante en el paradigma V2U (Vehicle-To-User) probando los HMIs (Human Machine Interfaces), que ayudan a monitorizar los principales parámetros del vehículo (como el estado de la batería, cantidad de los satélites GLONASS y GPS o la velocidad del coche), así como la posibilidad de introducir una ruta mediante una pantalla táctil central de forma similar a algunas empresas de automoción como Tesla. En el siguiente enlace se puede ver una demostración de funcionamiento del proyecto *Tech4AgeCar*: https://youtu.be/ZDN6PU21WJ8

Así que con todo lo anteriormente visto, en este proyecto, me centraré en la línea de la percepción de la zona conducible y las líneas de carretera para ayudar a las etapas de mapeado local y localización del carril sobre el que circula el vehículo.

1.3 Objetivos

El objetivo principal de este trabajo es identificar la zona conducible y las líneas de la carretera a partir de la información de una cámara, que estará situada dentro del vehículo y, de esta manera, mejorar las características de conducción autónoma del mismo.

Para ello se hará uso de la librería YOLOP [YOLOP(2019)], con la que se genera el sistema en el que se basa el proyecto, del cual se cuenta con código desarrollado por otro grupo de investigación y a disposición de la comunidad científica a través de *GitHub*.

Una vez comprendido el funcionamiento del sistema, se pasará a instalarlo de forma local en un sistema cerrado, para poder realizar pruebas y verificar que realmente se pueden detectar las líneas, las zonas conducibles de los carriles y los vehículos en imágenes y vídeos.

Posteriormente se creará un Dataset de imágenes sintéticas gracias al simulador CARLA [Carla(2022)], para poder realizar pruebas e intentar mejorar el modelo en la medida de lo posible.

Además, se creará una página Web para que se pueda verificar el funcionamiento del modelo y que así quede a disposición de cualquier persona que quiera probarlo.

Si cumplimos todos estos objetivos, se podrá ver como cuando el vehículo autónomo realice alguna ruta irá detectando de una manera precisa los distintos carriles por los que circula, mejorando así la conducción y la localización de éste.

Capítulo 2

Estudio teórico

2.1 Introducción

En este capítulo se realiza una revisión de los conocimientos que son necesarios para llevar a cabo el desarrollo de la aplicación planteada en el trabajo, en concreto, se hará un repaso de las redes neuronales y los tipos de aprendizajes de que disponen, haciendo especial mención al Deep Learning o aprendizaje profundo. Como base de este trabajo, se realizará un estudio de la red neuronal con aprendizaje profundo YOLOP [YOLOP(2019)] y en que conceptos se basa su arquitectura. Además, se revisarán las herramientas y librerías de desarrollo para tratamiento de imágenes OpenCV y la librería PythonAPI del simulador hiper-realista CARLA.

2.2 Redes Neuronales

Se define el Machine Learning, o aprendizaje basado en máquinas, como una disciplina del campo de la Inteligencia Artificial que, a través de ciertos algoritmos, dota a los ordenadores de la capacidad de identificar patrones en datos masivos y elaborar predicciones (análisis predictivo). Este aprendizaje permite a los computadores realizar tareas específicas de forma autónoma, es decir, sin necesidad de ser programados.

Los algoritmos de Machine Learning se dividen en tres categorías, siendo las dos primeras las más comunes:

1. Aprendizaje supervisado: estos algoritmos cuentan con un aprendizaje previo basado en un sistema de etiquetas asociadas a unos datos que les permiten tomar decisiones o hacer predicciones. Un ejemplo es un detector de spam que etiqueta un e-mail como spam o no, dependiendo de los patrones que ha aprendido del histórico de correos (remitente, relación texto/imágenes, palabras clave en el asunto, etc.).

2. Aprendizaje no supervisado: estos algoritmos no cuentan con un conocimiento previo. Se enfrentan al caos de datos con el objetivo de encontrar patrones que permitan organizarlos de alguna manera. Por ejemplo, en el campo del marketing se utilizan para extraer patrones de datos masivos provenientes de las redes sociales y crear campañas de publicidad altamente segmentadas.

3. Aprendizaje por refuerzo: su objetivo es que un algoritmo aprenda a partir de la propia experiencia. Esto es, que sea capaz de tomar la mejor decisión ante diferentes situaciones de acuerdo con un

proceso de prueba y error en el que se recompensan las decisiones correctas. En la actualidad se está utilizando para posibilitar el reconocimiento facial, hacer diagnósticos médicos o clasificar secuencias de ADN.

A su vez, una red neuronal es un método de la inteligencia artificial que enseña al ordenador a procesar datos de una manera que está inspirada en la forma en que lo hace el cerebro humano. Por lo que se trata de un tipo de Machine Learning llamado aprendizaje profundo(Deep Learning), que utiliza los nodos o las neuronas interconectados en una estructura de capas que se parece al cerebro humano. Creando un sistema adaptable que los ordenadores utilizan para aprender de sus errores y mejorar continuamente. De esta forma, las redes neuronales artificiales intentan resolver problemas complicados, como la realización de resúmenes de documentos o el reconocimiento de elementos de la carretera, con mayor precisión.

2.2.1 Conceptos básicos de una red

A continuación, se pasará a explicar los conceptos básicos que componen una red neuronal o se utilizan cuando hablamos de ellas. Estos términos son importantes para el desempeño de este proyecto, ya que se irá haciendo uso de ellos a lo largo del desarrollo del mismo.

- **Layer**

Existen tres tipos de capas (layers) en una red neuronal, todas contienen una o más neuronas:

1. Capa de entrada (Input Layer): Esta capa contiene neuronas que representan los datos que la red neuronal usara para entrenar. El número de neuronas de esta capa depende del número de características que tengan los datos.
2. Capa oculta (Hiden Layer): Una red neuronal puede tener varias capas de este tipo, cada una de estas capas contiene neuronas, en una red neuronal tradicional cada una de las neuronas de una capa están conectadas con todas las neuronas de la siguiente capa.
3. Capa de salida (Output Layer): Esta capa es la que se encarga de entregar los resultados, si estamos resolviendo un problema de clasificación esta capa tendrá un número de neuronas igual al número de clases que existan en los datos. El resultado es una lista de probabilidades para cada clase.

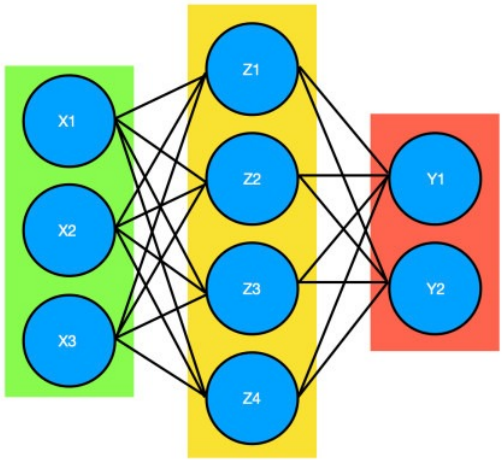


Figura 2.1: Layers.

- **Neuronas**

Una neurona, o nodo, es una parte importante de las redes neuronales y puede haber muchas de ellas repartidas entre varias capas (layers). Cada neurona contiene la función:

$$Z = WX + b \tag{2.1}$$

Esta función genera una línea que puede dividir los datos en dos grupos:

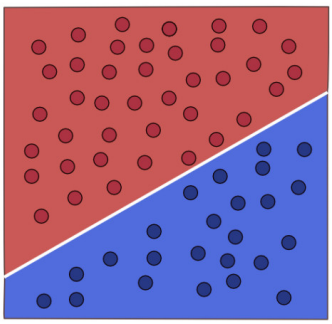


Figura 2.2: División de datos

El resultado de esta función es enviado a la función de activación:

$$A = sigmoid(Z) \tag{2.2}$$

$$A = relu(Z) \tag{2.3}$$

y el resultado final (A) es enviado a las neuronas de la siguiente capa (layer) o a la función de pérdida. Cada neurona tiene diferentes pesos (W) y un parámetro b que es igual entre las neuronas de una misma capa. Una neurona recibe un parámetro X que pueden ser los datos de entrada o los datos de salida de una neurona de la capa anterior.

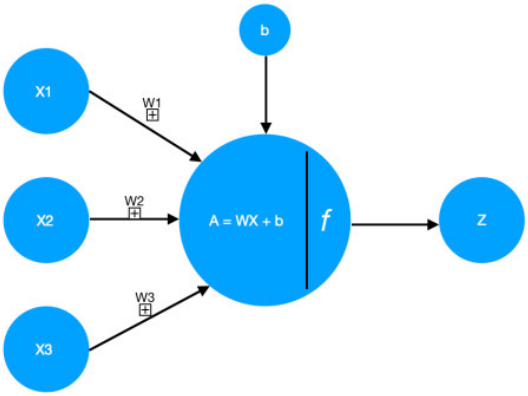


Figura 2.3: Esquema Red Neuronal

- **Weight**

Los pesos son parámetros importantes en una red neuronal, estos parámetros se multiplicarán con los datos de entrada X y con las salidas de las neuronas de la capa anterior A, cada parámetro W es diferente para cada neurona ya que en estos parámetros se guardan patrones que la neurona aprendió sobre los datos.

- **Overfitting**

Este concepto se aplica cuando la red neuronal aprende mucho sobre los datos de entrenamiento, pero tiene un desempeño pobre en los datos de validación o en datos que nunca ha visto. Este comportamiento suele ser indeseable excepto en casos muy controlados donde únicamente se encuentran casos del mismo tipo.

- **Underfitting**

Este concepto es lo contrario de overfitting. Sucede cuando la red neuronal no aprendió correctamente sobre los datos de entrenamiento y en general tiene un desempeño muy pobre en todas las predicciones.

- **Training set**

Cuando tenemos un set de datos, o conjunto de datos, es importante dividirlo en dos o tres sets diferentes, el primero de ellos es el set de entrenamiento, este se usa para que la red neuronal aprenda patrones sobre los datos.

- **Validation set**

Este es el segundo set de datos. Es necesario que este conjunto tenga datos diferentes de los otros dos, esto es importante ya que queremos ver como de buena es la red neuronal en datos que nunca ha visto.

Este conjunto se usa para buscar los mejores hiper-parámetros de una red neuronal, pueden ser: learning rate, epochs, optimizer, batch_size, número de capas, número de nodos, funciones de activación, etc. Queremos entrenar a la red neuronal con ciertos hiper-parámetros, ver el resultado en el set de validación, cambiar los valores de los hiper-parámetros e ir comparando resultados hasta encontrar los hiper-parámetros que sean buenos resolviendo el problema.

- **Test set**

Este es el último set de datos que se tiene en cuenta cuando se entrena una red neuronal. Este conjunto se usa al final para comprobar el rendimiento de la red neuronal después de entrenarla y encontrar los mejores hiper-parámetros.

Si no tenemos muchos datos, este conjunto se puede ignorar y solo quedarnos con el set de entrenamiento y el set de validación.

- **Forwardpropagation**

Es la manera en la cual las redes neuronales crean las predicciones. En un principio la red neuronal tiene valores de W y b aleatorios en cada neurona, los datos de entrenamiento pasan por estas neuronas hasta llegar a la capa de salida, en esta capa la red neuronal predice la clase a la cual pertenecen los datos de entrenamiento, estas predicciones las usa la función de pérdida para medir que tan buena es la red neuronal, este ciclo se repite varias veces según indiquemos y en cada ciclo se ejecuta el algoritmo de backpropagation para actualizar los valores de W y b.

- **Backpropagation**

Es importante conocer este concepto al detalle, generalmente se conoce el algoritmo de backpropagation como el encargado de optimizar la función de pérdida para mejorar las predicciones de una red neuronal. Este algoritmo se encarga de calcular las derivadas (o gradientes) de los parámetros W y b para saber cómo estos parámetros afectan al resultado de la función de pérdida, esta es una definición que puede ser usada para explicar el algoritmo pero, para ser más precisos, la optimización de una red neuronal se divide en dos partes: la primera es el algoritmo de backpropagation, este algoritmo se encarga de ver como los valores de W y b afectan al resultado de la función de pérdida y la segunda parte es el algoritmo de optimización, este se encarga de optimizar la red neuronal y cambiar los valores de W y de b conforme pasan los ciclos (o *epochs*).

Existen diferentes algoritmos de optimización, unos son mejores que otros, aunque depende del tipo de problema que se esté resolviendo. Este algoritmo es un parámetro de la red neuronal llamado *optimizer*. Los algoritmos tienen un comportamiento parecido que varía en algunos cálculos, pero la idea general es encontrar el *global minimum*, o el mínimo global, de la función de pérdida. Lo que hace el algoritmo es descender por la función hasta llegar a este punto donde la función se encuentra optimizada, esto se logra con ayuda de las derivadas que indican qué camino se tiene que seguir.

- **Optimizer**

Existen diferentes tipos de algoritmos de optimización el más famoso se llama *gradient descent* y el más usado se llama *Adam*. Este es considerado un hiper-parámetro.

- **Learning rate**

Este hiper-parámetro indica que tan largo será el camino que tome el algoritmo de optimización. Si el valor es muy pequeño la actualización se puede quedar atrapada en un mínimo local y los valores de W y b no cambiarían correctamente, asimismo la red neuronal tardara mucho más tiempo en optimizar. Por otro lado, si los valores son muy altos la actualización puede pasarse del punto perfecto que sería el mínimo global y nunca encontrarlo y aunque el aprendizaje sea más rápido nunca llegará al mínimo global.

- **Lost function**

La función de pérdida, también conocida como función de costo, es la función que nos dice cómo de buena es la red neuronal. Un resultado alto indica que la red neuronal tiene un desempeño pobre y un resultado bajo indica que la red neuronal está haciendo un buen trabajo.

- **Epoch**

Este es el número de veces que se ejecutarán los algoritmos de *forwardpropagation* y *backpropagation*. En cada ciclo (*epoch*) todos los datos de entrenamiento pasan por la red neuronal para que esta aprenda sobre ellos, si existen 10 ciclos y 1000 datos, cada ciclo los 1000 datos pasarán por la red neuronal. Si se especifica el parámetro batch size cada ciclo (*epoch*) tendrá más ejecuciones internas, estas ejecuciones se llaman iteraciones.

Si tenemos un *batch size* de 100, se tendrán 10 iteraciones para completar un ciclo, en cada iteración se ejecutan los algoritmos de *forwardpropagation* y *backpropagation*, de esta manera la red neuronal actualiza más veces los parámetros W y b.

- **Batch size**

Es el número de datos que tiene cada iteración de un ciclo (*epoch*), esto es útil porqué la red neuronal actualiza los parámetros W y b más veces, también cuando se tienen grandes cantidades de datos se necesitan computadoras con más memoria y la red neuronal tarda más en ejecutar cada ciclo.

Si dividimos los ciclos en iteraciones con un número de datos más pequeño ya no es necesario cargar todos los datos en la memoria al mismo tiempo y la red neuronal se entrena más rápido.

- **Hyperparameters**

Son los parámetros que modificamos manualmente en una red neuronal. Son muy importantes para lograr un buen desempeño y nuestro trabajo es encontrar los que mejor se adapten al problema que estamos resolviendo.

2.2.2 Conceptos básicos de Deep Learning

A continuación, se definirán términos más importantes que aparecen cuando trabajamos con Deep Learning y que podremos llegar a ver durante el desarrollo de este proyecto.

- **Datos**

El recurso principal de los modelos basados en Deep Learning son los datos y estos pueden ser de dos tipos: estructurados o no estructurados:

1. Datos estructurados: Se caracterizan por estar organizados de forma tal que siempre siguen un patrón predeterminado.
2. Datos no estructurados: Estos datos no están organizados de forma alguna, y se caracterizan porque no siguen un patrón predeterminado, o al menos este patrón no es evidente a simple vista.

- **Clasificación**

En este tipo de tarea el modelo se encarga de determinar si los datos de entrada corresponden a una de varias categorías (o clases) posibles. Los clasificadores se caracterizan porque a la salida del modelo se obtienen valores discretos 0 ó 1 para clasificación binaria, o bien 0, 1, 2, 3, etc... para clasificación multiclase.

- **Regresión**

En este tipo de tarea el modelo se encarga de establecer la relación existente entre dos variables para posteriormente predecir un valor numérico con base en los datos de entrada. Los sistemas de regresión se caracterizan porque a la salida generan valores continuos.

- **Aprendizaje**

El aprendizaje se puede definir como un procedimiento que permite calcular los parámetros del modelo que permiten encontrar la mejor representación que relacione los datos de salida con los de entrada.

- **Entrenamiento**

Durante el entrenamiento, los parámetros del modelo se calculan y refinan de forma progresiva, para así aprender de forma autónoma la mejor representación que relacione los datos de entrada y de salida.

2.2.3 Estudio de la red YOLOP

En este apartado, se realiza un resumen del artículo científico que supone la base de este trabajo fin de grado, la red YOLOP [YOLOP(2019)].

Un sistema panóptico es un componente esencial dentro del mundo de la conducción autónoma. Este sistema de alta precisión y que trabaja en tiempo real, ayuda a realizar la toma de decisiones más rápido y más eficientemente.

En conjunto la red YOLOP ayudará a detectar vehículos que se tengan alrededor, la segmentación del área conducible y la detección de carriles.

Además, para poder realizar el entrenamiento del modelo, en este trabajo fin de grado se utilizará el conjunto de datos original de YOLOP, el BDD100K [University(2019)], y también se creará un Dataset nuevo con imágenes sintéticas con el que se mejorará la red.

Los sistemas de percepción en los que se basa la conducción panóptica[1] desempeñan un gran papel en la conducción autónoma actual, ya que gracias a todas las imágenes que se tomarán durante el proceso, se podrá extraer información, y así, que se tenga la capacidad de ayudar a crear sistemas de conducción en los que las decisiones tomadas por el vehículo mejoren en gran medida.

La red YOLOP, como antes se ha comentado, será capaz de detectar vehículos[2], detectar la zona conducible del carril y las líneas del carril.

Hoy en día hay varios proyectos que manejan todas estas tareas por separado, véase por ejemplo R-CNN [R-CNN(2022)] y YOLOV [YOLOV(2022)], las cuales se dedican a la detección de objetos.

Para la segmentación semántica se utilizan mayormente redes como UNet [UNet(2022)] y PSPNet [PSPNet(2022)]. Además de las redes SCNN [SCNN(2022)] y SAD-ENet [SAD-ENet(2022)] para la detección de carriles.

Todos estos proyectos logran un gran rendimiento en cada uno de sus campos, pero procesar uno tras otro para intentar lograr un resultado que en su conjunto final sea bueno y completo, se convierte en un proceso lento de abordar. Por lo que la red YOLOP aborda estos tres campos a la vez, para así, facilitar la obtención de datos y mejorar en gran medida la rapidez de procesamiento.

(a) Calle sin procesar (b) Calle Procesada por YOLOP

Figura 2.4: Funcionamiento de la red YOLOP

Con todo esto lo que se consigue gracias a la red YOLOP es mejorar en gran medida el tiempo de inferencia dentro de un sistema, se restringen costos computacionales y se mejora el rendimiento global de cada tarea.

Para obtener una alta precisión y velocidad, se diseña una arquitectura que hace uso de una red simple y eficiente. Se utiliza un CNN [CNN(2022)] como codificador para extraer toda la información referente

[1] Cuando hablamos de conducción panóptica nos referimos a un sistema en el que se llega a captar todos los elementos del entorno.

[2] En este caso solo se detectarán vehículos para no sobrecargar mucho el procesamiento de datos, aunque podríamos añadir más elementos

a la imagen. Luego, tres decodificadores hacen uso de esta información para completar sus respectivas tareas.

El decodificador de detección de vehículos se basará en la red de una sola etapa de mejor rendimiento actual, por dos razones principales: la red de una sola etapa es más rápida que la de dos etapas y, además, el mecanismo de predicción basado en cuadrículas de detección de una sola etapa está más relacionado con las otras dos tareas de segmentación semántica, que es lo que se usará para realizar la transformación final del sistema.

La salida del codificador incorpora información semántica de diferentes niveles y escalas, y la rama de segmentación puede usar esta información para completar la predicción semántica por píxeles de una forma realmente eficaz.

La estrategia de entrenamiento que se seguirá será la de extremo a extremo, ya que se han probado otros paradigmas, pero con peores resultados. Gracias a este entrenamiento la red `YOLOP` consigue 41 FPS con una NVIDIA TITAN XP y 23 FPS en una Jetson TX2. Logrando este último resultado en las tres tareas del conjunto de datos BDD100K.

Las principales contribuciones de esta red son: se presenta una red multitarea eficiente que puede manejar conjuntamente las tres tareas cruciales en la conducción autónoma: detección de objetos, segmentación del área conducible y detección de carriles para ahorrar costos computacionales y reducir el tiempo de inferencia. Además, mejorar el desempeño de cada tarea. Este trabajo es el primero en alcanzar el tiempo real en dispositivos integrados mientras mantiene un gran nivel de rendimiento en el conjunto de datos BDD100K. Aparte, se diseñan los experimentos ablativos para verificar la efectividad del esquema multitarea.

Se demostrará que las tres tareas se pueden aprender de forma conjunta y sin la tediosa optimización alterna.

2.2.4 Metodología

Como se muestra en la 2.5, la red de percepción panóptica,`YOLOP`, contiene un codificador compartido y tres decodificadores posteriores para resolver las tareas específicas. No hay bloques compartidos complejos o redundantes entre los diferentes decodificadores, lo que reduce el consumo computacional y permite entrenar fácilmente la red de extremo a extremo.

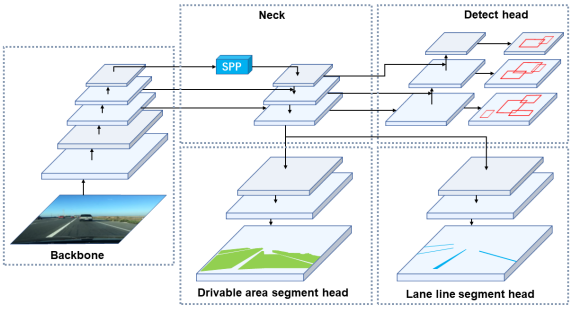


Figura 2.5: Arquitectura de la Red YOLOP

2.2.4.1 Codificador

La red comparte un codificador, que se compone de una red troncal(Blackbone) y de una red de cuello(Neck).

- Red troncal(Blackbone): la red troncal se utiliza para extraer las características de la imagen de entrada. El modelo utilizará la CSPDarknet como columna vertebral, lo que resuelve el problema de la duplicación de gradientes durante la optimización. Admite la propagación de la información y la reutilización de esta, lo que reduce la cantidad de parámetros y cálculos. Esto ayuda a garantizar el rendimiento en tiempo real de la red.

- Cuello(Neck): El cuello se utiliza para fusionar las características generadas por la columna vertebral(CSPDarknet). Está compuesto principalmente por el módulo Spatial Pyramid Pooling (SPP) y el módulo Feature Pyramid Network (FPN). SPP genera y fusiona características de diferentes escalas, y el FPN las fusiona en diferentes niveles semánticos, lo que hace que las características generadas contengan múltiples escalas e información de múltiples niveles. Se utilizará el método de concatenación para fusionar características en esta red.

2.2.4.2 Decodificadores

Los tres cabezales que componen la red YOLOP son decodificadores específicos para cada una de las tres tareas.

- Cabezal de detección(Detection Head): similar a YOLOv4, se adopta un esquema de detección de múltiples escalas basado en anclas. En primer lugar, se utiliza una estructura llamada Path Aggregation Network (PAN), una red piramidal de características de abajo hacia arriba (FPN). FPN transferirá las características semánticas de arriba hacia abajo y PAN transferirá las características de posicionamiento de abajo hacia arriba. Se combinan para obtener un mejor efecto de fusión de características y luego se usa directamente los mapas de características de fusión multiescalar en el PAN para la detección. Posteriormente, a cada cuadrícula se le asignarán tres anclas previas con diferentes relaciones de aspecto, y el cabezal de detección predecirá el desplazamiento de la posición y la escala de la altura y el ancho, así como la probabilidad correspondiente de cada categoría y la confianza de la predicción.

- Cabezal de segmento de área manejable (Drivable area segment head) y Cabezal de segmento de línea de carril (Lane line segment head): estos adoptan la misma estructura de red que el anterior cabezal. Se alimenta la capa inferior de FPN hacia la rama de segmentación, con el tamaño de (W/8, H/8, 256). Después de tres procesos de sobre-muestreo, se restaura el mapa de características de salida al tamaño de (W, H, 2), que representa la probabilidad de cada píxel en la imagen de entrada para el área de conducción/línea de carril y el fondo. Debido al SPP compartido en la red del cuello (Neck), no se agrega un módulo SPP adicional para segmentar las sucursales como lo hacen otros modelos, lo que por desgracia no mejora el rendimiento de la red. Además, se utiliza el método de interpolación más cercana en la capa de muestreo ascendente para reducir el costo de cálculo. Como resultado, los decodificadores de segmentos no solo obtienen resultados de alta precisión, sino que también son muy rápidos durante la inferencia.

2.2.4.3 Función de pérdida

Como ya se ha comentado anteriormente, la función de pérdida es aquella que consigue evaluar la desviación entre las predicciones realizadas por la red neuronal y los valores reales de las observaciones que se utilizarán durante el aprendizaje. Cuanto menos sea el valor de esta función, significará que más eficiente es la red del modelo. Por lo que siempre se intentará buscar reducir al mínimo dicha desviación, ajustando los distintos parámetros de los que está compuesta la red.

Debido a que la red YOLOP contiene tres decodificadores, la función de pérdida que se generará estará compuesta de tres partes. Para la pérdida en la detección (L_{det}) se trata de una suma de la pérdida de clasificación, pérdida de objetos y pérdida de los cuadros que delimitan los objetos, la siguiente fórmula muestra dicha suma:

$$L_{det} = \sigma_1 * L_{class} + \sigma_2 * L_{obj} + \sigma_3 * L_{box} \tag{2.4}$$

donde L_{class} y L_{obj} se utilizan para reducir la pérdida de ejemplos bien clasificados, lo que obliga a la red a centrarse en los difíciles. L_{class} se utiliza para penalizar la clasificación y L_{obj} para la confianza de una predicción. Mientras que L_{box} tendrá en cuanta la distancia, la superposición, la similitud en la escala y la relación de aspecto en la caja predicha con respecto a la parte real del terreno.

Por otra parte, tanto para la pérdida en segmentación del área conducible (L_{da-seg}), como para la de las líneas del carril (L_{ll-seg}), contienen una pérdida de entropía cruzada (L_{ce}), que intentará minimizar los errores de clasificación de cada píxel procesado por la red. Además, utilizaremos la pérdida de IoU que se añadirá a L_{ll-seg} ya que es especialmente eficiente en el sentido de detección de líneas de carril. L_{da-seg} y L_{ll-seg} se definirán mediante las siguientes ecuaciones:

$$L_{da-seg} = L_{ce} \tag{2.5}$$

$$L_{ll-seg} = L_{ce} + L_{IoU} \tag{2.6}$$

$$L_{Iou} = TN/(TN + FP + FN) \tag{2.7}$$

Como conclusión, la pérdida final de la red será una suma ponderada de cada una de las tres partes que la conforman:

$$L_{all} = \sigma_1 * L_{det} + \sigma_2 * L_{da-seg} + \sigma_3 * L_{ll-seg} \tag{2.8}$$

donde se podrán ir cambiando los valores de sigma para poder ir balanceando todas las partes del total de la pérdida.

2.2.4.4 Paradigma de Entrenamiento

Para la YOLOP se utilizará el entrenamiento de extremo a extremo, ya que con esto se consigue que finalmente las tres partes que conforman el modelo puedan aprender de forma conjunta. Se utiliza este paradigma ya que todas las partes del modelo están relacionadas entre sí. Por otra parte, se han probado también algoritmos que entrenan el modelo paso a paso, con lo que se consigue que en cada momento del entrenamiento se pueda llegar a entrenar una o varias tareas que se relacionen. Para este último paradigma diseñaron un algoritmo para poder generar dicho entrenamiento:

Algoritmo 1: Primero, sólo se entrena el codificador y cabezal de detección. Luego se detienen estos dos primeros, y se pasa a entrenar los dos cabezales de segmentación. Finalmente, toda la red se entrenará conjuntamente para las tres tareas.

2.2.5 Experimentos

2.2.5.1 Ajustes

Ajustes del conjunto de datos: Con la BDD100K se consigue respaldar la investigación del aprendizaje multitarea en el campo de la conducción autónoma. Ya que cuenta con 100k imágenes y anotaciones de

10 tareas. Además, dado que este conjunto de datos está compuesto por la geografía, entorno y clima, se puede considerar que es lo suficientemente robusto para poder utilizarlo con otros entornos. Gracias a todo esto, es por lo que se elige este conjunto para poder entrenar la `YOLOP`.

La base de datos, desglosándola estará compuesta de tres partes, 70k de imágenes que corresponden con el entrenamiento, 10k de imágenes que servirán para la validación y 20k de imágenes que se utilizarán para realizar pruebas (test).

Detalles de implementación: para mejorar el rendimiento del modelo, se adoptan algunas técnicas y métodos prácticos de aumento de datos. Ya que así se consigue aumentar la variabilidad de las imágenes y que el modelo consiga ser cada vez más robusto. Además, se usará Adams como optimizador para entrenar el modelo.

Configuración experimental: Se muestra un gráfico con la comparativa de este modelo con respecto a otras redes que han utilizado BDD100K pero se enfocan en una sola tarea a la vez.

Network	Recall(%)	mAP50(%)	Speed(fps)
MultiNet	81.3	60.2	8.6
DLT-Net	89.4	68.4	9.3
Faster R-CNN	77.2	55.6	5.3
YOLOv5s	86.8	77.2	82
YOLOP (ours)	89.2	76.5	41

Figura 2.6: Comparativa YOLOP con respecto a otras redes actuales.

2.2.5.2 Estudios de Ablación

Como punto de partida diremos que la ablación es la eliminación de un componente de un sistema de inteligencia artificial (IA). Un estudio de ablación investiga el rendimiento de un sistema de IA mediante la eliminación de ciertos componentes para comprender la contribución del componente al sistema general. Para la red `YOLOP` se han diseñado dos experimentos de ablación para así demostrar aún más la efectividad de la red.

End-to-end v.s. Step-by-step: En la figura 2.7 se podrá ver una comparativa del rendimiento del modelo entero con respecto a entrenarlo utilizando solo distintas partes. Como era de esperar, es mejor el End-to-End ya que el modelo está preparado para ello, pero sin embargo los métodos de entrenamiento paso a paso dan resultados muy similares:

Training method	Recall(%)	AP(%)	mIoU(%)	Accuracy(%)	IoU(%)
ES-W	87.0	75.3	90.4	66.8	26.2
ED-W	87.3	76.0	91.6	71,2	26.1
ES-D-W	87.0	75.1	91.7	68.6	27.0
ED-S-W	87.5	76.1	91.6	68.0	26.8
End-to-end	89.2	76.5	91.5	70.5	26.2

Figura 2.7: End to End vs Step by Step.

Multi-task v.s. Single task: Se comparará en este caso el rendimiento del modelo al ejecutar todas las tareas a la vez, en comparación a cuando lo hace tarea por tarea. Así, en un primer lugar se entrena el modelo para realizar las tres tareas de las que está compuesto a la vez, y por otro lado se entrenará la detección de objetos, la segmentación del área de conducción y las líneas del carril por separado. En la figura 2.5 podremos ver los resultados que se han obtenido:

Training method	Recall(%)	AP(%)	mIoU(%)	Accuracy(%)	IoU(%)	Speed(ms/frame)
Det(only)	88.2	76.9	-	-	-	15.7
Da-Seg(only)	-	-	92.0	-	-	14.8
Ll-Seg(only)	-	-	-	79.6	27.9	14.8
Multitask	89.2	76.5	91.5	70.5	26.2	24.4

Figura 2.8: Multi-task vs Single task

2.3 Simulador hiper-realista CARLA

2.3.1 Conceptos básicos

CARLA [Carla(2022)] es uno de los simuladores de conducción autónoma más completos que existen en la actualidad, sus siglas vienen del inglés 'Car Learning to Act'(Coche que aprende a actuar) y fue creado por la Universidad Politécnica de Barcelona. Es un simulador de código abierto enfocado en la investigación de la conducción autónoma.

CARLA se ha desarrollado desde cero para apoyar el desarrollo, la formación y la validación de los sistemas de conducción autónoma. A parte del código y los protocolos, CARLA proporciona diversos elementos (trazados urbanos, edificios, vehículos), que se crearon para la simulación. Además, la plataforma admite la especificación flexible de conjuntos de sensores y condiciones ambientales.

Asimismo, se trata de un sistema multiplataforma, por lo que se puede instalar tanto en Windows como en Linux sin problemas, aparte de poder contar con la posibilidad de utilizar la tecnología Docker para desplegarlo mediante contenedores.

El simulador estará compuesto de los siguientes elementos:

- Fundamentos: descripción general de los componentes básicos de CARLA. https://carla.readthedocs.io/en/latest/core_concepts/
- Actores: aprenda sobre los actores y cómo manejarlos. https://carla.readthedocs.io/en/latest/core_actors/
- Mapas: descubre los diferentes mapas y cómo se mueven los vehículos. https://carla.readthedocs.io/en/latest/core_map/
- Sensores y datos: recupere datos de simulación mediante sensores. https://carla.readthedocs.io/en/latest/core_sensors/
- Tráfico: una descripción general de las diferentes opciones disponibles para llenar sus escenas con tráfico. https://carla.readthedocs.io/en/latest/ts_traffic_simulation_overview/
- Integraciones de terceros: integraciones con aplicaciones y bibliotecas de terceros. https://carla.readthedocs.io/en/latest/3rd_party_integrations/
- Desarrollo: información sobre cómo desarrollar funciones personalizadas para CARLA. https://carla.readthedocs.io/en/latest/development_tutorials/
- Custom assets: información sobre cómo desarrollar activos personalizados. https://carla.readthedocs.io/en/latest/custom_assets_tutorials/

Aparte se pone a disposición de los usuarios unos recursos para poder interactuar con el simulador:

- Blueprint library: recurso para generar actores.
- Python API: clases y métodos en la API de Python. Estos recursos nos ayudarán a poder interactuar de una manera sencilla con el simulador y será esta herramienta la que utilizaremos durante el proyecto.
- C++ reference: Clases y métodos en CARLA C++..

2.3.2 Librería PythonAPI de CARLA

La librería PythonAPI es el conjunto de clases y métodos desarrollados en Python y que nos ayudará a interactuar de una manera sencilla con el simulador. Gracias a esta biblioteca se ha podido generar dentro del simulador tráfico continuo y además un vehículo autónomo que vaya realizando un recorrido aleatorio dentro del simulador para que desde este se puedan ir capturando imágenes y así poder generar el Dataset que se explicará en el apartado 3.3.

Las clases que se han considerado más importantes durante el desarrollo de este trabajo son:

- *carla.World*: Los objetos del mundo son creados por el cliente para tener un lugar para que suceda la simulación. El mundo contiene el mapa que podemos ver, es decir, el activo, no el mapa de navegación. Los mapas de navegación son parte de la clase *carla.Map*. También gestiona el clima y los actores presentes en él. Solo puede haber un mundo por simulación, pero se puede cambiar en cualquier momento.
- *carla.Actor*: CARLA define a los actores como cualquier cosa que juega un papel en la simulación o que se puede mover. Eso incluye: peatones, vehículos, sensores y señales de tránsito (considerando los semáforos como parte de estos). Los actores son generados en la simulación por *carla.World* y necesitan que se cree un *carla.ActorBlueprint*.
- *carla.BlueprintLibrary*: Una clase que contiene los planos proporcionados para la generación de actores. Su aplicación principal es devolver los objetos *carla.ActorBlueprint* necesarios para generar actores. Cada *blueprint* tiene un identificador y atributos que pueden o no ser modificables. El servidor crea automáticamente la biblioteca y se puede acceder a ella a través de *carla.World*.
- *carla.ColorConverter*: Clase que define patrones de conversión que se pueden aplicar a una *carla.Image* para mostrar información proporcionada por carla.Sensor. Las conversiones de profundidad provocan una pérdida de precisión, ya que los sensores detectan la profundidad como flotante que luego se convierte a un valor de escala de grises entre 0 y 255.
- *carla.Imagen*: es una clase heredada de *carla.SensorData*. Clase que define una imagen de colores BGRA de 32 bits que se utilizará como datos iniciales recuperados por los sensores de la cámara. Hay diferentes sensores de cámara (actualmente tres, RGB, profundidad y segmentación semántica) y cada uno de ellos hace un uso diferente de las imágenes.
- carla.Sensor: Clase heredado de *carla.Actor*. Los sensores componen una familia específica de actores bastante diversa y única. Normalmente se generan como adjuntos/hijos de un vehículo (eche un vistazo a *carla.World* para obtener información sobre la generación de actores). Los sensores están completamente diseñados para recuperar diferentes tipos de datos que están escuchando. Los datos que reciben tienen la forma de diferentes subclases heredadas de *carla.SensorData* (según el sensor).La mayoría de los sensores se pueden dividir en dos grupos: los que reciben datos en cada

tick (cámaras, nubes de puntos y algunos sensores específicos) y los que solo reciben en determinadas circunstancias (detectores de disparo). CARLA proporciona un conjunto específico de sensores y su modelo se puede encontrar en *carla.BlueprintLibrary*. Recibir datos en cada tick. (- Cámara de profundidad. -Sensor gnss. -Sensor IMU. - Transmisión de rayos Lidar. - SemanticLidar raycast. - Radar. - Cámara RGB. - Sensor RSS. - Cámara de Segmentación Semántica). Sólo recibe datos cuando se activa.(- Detector de colisión. - Detector de invasión de carril. - Detector de obstáculos).

- *carla.Vehicle*: Clase heredada de carla.Actor. Uno de los grupos de actores más importantes de CARLA. Estos incluyen cualquier tipo de vehículo desde automóviles hasta camiones, motos, furgonetas, bicicletas y también vehículos oficiales como los coches de policía. Se proporciona un amplio conjunto de estos actores en *carla.BlueprintLibrary* para facilitar diferentes requisitos. Los vehículos pueden controlarse manualmente o configurarse en un modo de piloto automático que el administrador de tráfico conducirá del lado del cliente.

Y también se tienen un par de métodos que he considerado importante durante el desarrollo del trabajo:

- *command.SetAutopilot*(): Adaptación del comando set_autopilot() en *carla.Vehicle*. Enciende/apaga el modo de piloto automático del vehículo.
- *command.SpawnActor*(): Adaptación de comandos de spawn_actor() en *carla.World*. Genera un actor en el mundo basado en el plano proporcionado y la transformación. Si se proporciona un padre, el actor se adjunta a él.

En el caso de que se quiera ver el contenido completo de la PythonAPI, se puede consultar el siguiente enlace: Python API - https://carla.readthedocs.io/en/latest/python_api/

2.4 Librería de tratamiento de imágenes OpenCV

OpenCV es una biblioteca libre de visión artificial originalmente desarrollada por Intel. OpenCV significa Open Computer Vision (Visión Artificial Abierta). Desde que apareció su primera versión alfa en el mes de enero de 1999, se ha utilizado en una gran cantidad de aplicaciones, y hasta 2022 se la sigue mencionando como una de las bibliotecas más popular de visión artificial. Detección de movimiento, reconocimiento de objetos, reconstrucción 3D a partir de imágenes, son sólo algunos ejemplos de aplicaciones de OpenCV.

Su popularidad se debe a que es:

- Libre: publicada bajo licencia BSD, que permite que sea usada libremente para propósitos comerciales y de investigación
- Multiplataforma: para los sistemas operativos GNU/Linux, Mac OS X, Windows y Android, y para diversas arquitecturas de hardware como x86, x64 (PC), ARM (celulares y Raspberry Pi)
- Documentada y explicada: la organización tiene una preocupación activa de mantener la documentación de referencia para desarrolladores lo más completa y actualizada posible, ejemplos de uso de sus funciones y tutoriales accesibles al público no iniciado en visión artificial, además de difundir y fomentar libros y sitios de formación.

El proyecto pretende proporcionar un entorno de desarrollo fácil de utilizar y altamente eficiente. Esto se ha logrado realizando su programación en código C y C++ optimizados, aprovechando, además

las capacidades que proveen los procesadores multinúcleo. OpenCV puede además utilizar el sistema de primitivas de rendimiento integradas de Intel, un conjunto de rutinas de bajo nivel específicas para procesadores Intel (IPP).

Esta biblioteca será realmente útil para el tratamiento de imágenes, y se usará para generar máscaras o para adaptar el tamaño de las imágenes. Se usará en varios puntos durante este trabajo, suponiendo un elemento bastante crucial ya que es una biblioteca muy potente y que nos aporta muchísimas utilidades

2.5 Conclusiones del capítulo

La red `YOLOP` es una red simple y eficiente, la cual consigue manejar simultáneamente tres tareas de percepción de conducción, como son la detección de objetos, la segmentación de la zona conducible y la detección de las líneas de los carriles. Además, puede ser entrenada End-to-End, para agilizar en gran medida este paso y no tener que hacerlo tarea a tarea. Con esto último consigue superar a otras redes que se utilizan en la actualidad. Y por último cabe recalcar, que consigue realizar inferencias en tiempo real en dispositivos embebidos, con lo que se demuestra que se puede utilizar la red en aplicaciones reales.

Además, gracias a CARLA se podrá crear un escenario real del que poder sacar mucha información, ya sean imágenes de la circulación o información de los sensores. Con el uso de la PythonAPI, se logrará conectar con el simulador y extraer datos de una manera bastante sencilla.

Como se ha visto, OpenCV es una biblioteca de tratamiento de imágenes realmente potente y, gracias a ella, podremos realizar el procesado de las imágenes sintéticas que extraigamos de CARLA para así poder generar un Dataset en el apartado 3.3.

Con todo esto podremos empezar a desarrollar todos los objetivos que se han planteado para este trabajo fin de grado.

Capítulo 3

Desarrollo

3.1 Introducción

En este capítulo se empezará probando la implementación del modelo en local, después se pasará a crear un Dataset para poder entrenar el modelo y, por último, se desarrollará una interfaz web para que se pueda probar la red de una manera simple e intuitiva.

Se estructurará el capítulo en los siguientes apartados:

- Implementación local del modelo `YOLOP`.3.2
- Creación de un nuevo Dataset sintético para entrenamiento del modelo.3.3
- Desarrollo de interfaz web para prueba de funcionamiento del modelo. 3.4

3.2 Implementación local del modelo **YOLOP**

3.2.1 Introducción e instalación

En este apartado se estudiará el funcionamiento del modelo y se implementará en un entorno local.
Se empezará descargando el modelo desde su repositorio oficial de Github:

- **YOLOP**: https://github.com/hustvl/YOLOP

La estructura de carpetas que sigue el modelo será la siguiente:

```
|--inference
| |--images   # inference images
| |--output   # inference result
|--lib
| |--config/default   # configuration of training and validation
| |--core
| | |--activations.py   # activation function
| | |--evaluate.py   # calculation of metric
| | |--function.py   # training and validation of model
| | |--general.py   #calculation of metric
| | |--loss.py   # loss function
| | |--postprocess.py   # postprocess(refine da-seg and ll-seg, unrelated to paper)
| |--dataset
```

```
| | |--AutoDriveDataset.py   # Superclass dataset, general function
| | |--bdd.py   # Subclass dataset, specific function
| | |--hust.py   # Subclass dataset(Campus scene,unrelated to paper)
| | |--convect.py
| | |--DemoDataset.py   # demo dataset(image, video and stream)
| |--models
| | |--YOLOP.py    # Setup and Configuration of model
| | |--commom.py   # calculation module
| |--utils
| | |--augmentations.py    # data augumentation
| | |--autoanchor.py   # auto anchor(k-means)
| | |--split_dataset.py  # (Campus scene, unrelated to paper)
| | |--utils.py  # Logging
| |--run
| | |--dataset/training time  # Visualization, logging and model_save
|--tools
| | |--demo.py    # demo(folder,camera)
| | |--test.py
| | |--train.py
|--toolkits
| | |--deploy    # Deployment of model
| | |--datapre    # Generation of gt(mask) for drivable area segmentation task
|-- weights    # Pretraining model
```

Código 3.1: Estructuras de carpetas.

Para empezar a trabajar con el modelo primero deberemos instalar Python 3.7 o superior, pip y, además, todos los módulos necesarios para su correcto funcionamiento. Como se menciona en el anexo de requisitos software A.1, se tendrán que instalar todas las librerías que el modelo necesita y están especificadas en **requirements.txt** [YOLOP(2022)].

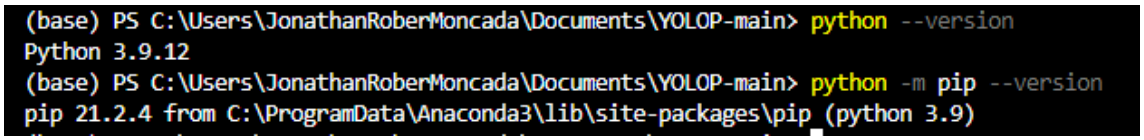


Figura 3.1: Versión de Python

Con el siguiente comando se podrá revisar qué librerías se tienen instaladas en el entorno actualmente:

```
pip list
```

Si al ejecutar alguno de los pasos, que veremos a posteriori, nos sale un error con algún módulo que no tengamos instalado, se soluciona rápidamente utilizando:

```
pip install <nombre_modulo>
```

3.2.2 Inferencia de imágenes

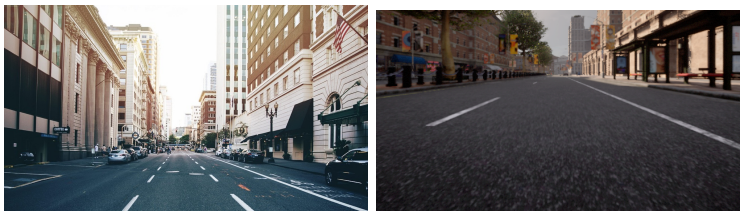
Con esto ya se podrá empezar a probar el funcionamiento del modelo YOLOP. Como punto de partida se puede empezar enseñando como el modelo nos genera una salida con las tres capas juntas, poniendo de color verde la zona conducible de la carretera, en rojo las líneas de los carriles y nos señalará con unos rectángulos de borde morado los vehículos que detecta, como se ve en la figura 2.5. Para ello se utilizará el archivo 'demo.py' que se sitúa en la carpeta **tools**.

A continuación, se muestran todas las opciones que ofrece este programa:

- *weights*: la ruta donde consultará los pesos elegidos para realizar la inferencia (/weigths/End-to-end.pth por defecto).
- *source*: se le indica la ruta de donde sacarán las imágenes o vídeos con los que realizará la inferencia, por defecto está en /inference/videos.
- *img-size*: tamaño que se le quiere dar a la inferencia.
- *conf-thres*: umbral de confianza del objeto, por defecto 0.25.
- *iou-thres*: Umbral *IOU* para *NMS*, por defecto 0.45.
- *device*: dispositivo con el que generar la inferencia, por defecto la CPU, aunque se puede elegir con un número entero el puerto de la gráfica que se tiene conectada.
- *save-dir*: ruta del directorio donde se almacenarán las imágenes finales.

Para el procesamiento de imágenes, se le puede pasar tanto una carpeta con múltiples elementos o una imagen aislada, esto mismo pasará con el procesamiento de vídeos. En este caso se va a realizar la inferencia de unas imágenes que se tienen guardadas en la carpeta inferences/images:

```
python tools/demo.py --source inference/images
```



(a) Imagen de Prueba 1. (b) Imagen de Prueba 2.

Figura 3.2: Imágenes usadas para la inferencia.

Con esto se puede conseguir que el modelo genere y almacene en la carpeta /inference/output las imágenes procesadas:

(a) Imagen procesada 1. (b) Imagen procesada 2.

Figura 3.3: Imágenes procesadas.

3.2.3 Inferencia de vídeo

Para el procesado de vídeo será idéntico, se ejecuta el archivo demo con el cual se consigue que se procese el vídeo y que la salida de la inferencia se almacene en la carpeta /inference/output. Para esta parte se ha grabado un pequeño recorrido en coche y se ha pasado al modelo para que lo procese. Además, se ha subido el vídeo a Youtube para que se vea su funcionamiento:

```
python tools/demo.py --source inference/video/calle.mp4
```

Vídeo procesado del recorrido grabado desde el coche. - https://youtu.be/SCZKZsmJICs

3.2.4 Inferencia en tiempo real

Además de los puntos anteriores podemos ver que el modelo también realiza la inferencia de vídeo en tiempo real, ya que es capaz de coger la cámara que se tenga conectada al ordenador y utilizarla para generar una ventana de salida, donde mostrará la imagen procesada en tiempo real. En este caso cogerá por defecto el 'device 0' que concuerda con la cámara que se tenga conectada al ordenador y para la lanzar la inferencia se tiene el siguiente comando:

```
python tools/demo.py --source 0
```

3.2.5 Segmentación de la imagen

Y como punto final se muestra como también el modelo es capaz de generar por separado todas las capas de las que está compuesto. Como se puede ver en la imagen 2.5, el modelo lo forman tres partes, la detección de la zona conducible, la detección de las líneas de carril y la detección de vehículos. Y gracias al archivo test-onnx.py se puede generar la segmentación de una imagen o vídeo que se le pasa como parámetro:

```
python ./test_onnx.py --weight yolop-640-640.onnx --img test.jpg
```

Indagando más en este archivo, en el apartado 3.4, ayuda a crear una aplicación web donde probar el modelo. Y como se puede ver, se dispone de otro parámetro importante, como es el '–weight', con el cual se le pasa al archivo el modelo exportado para que resulte más fácil trabajar con él. De momento se muestra a continuación un ejemplo del funcionamiento de este archivo y las imágenes que genera:

Figura 3.4: Imagen a segmentar

(a) Área conducible (b) Líneas de carril (c) Detección de objetos

Figura 3.5: Segmentación de la imagen

Con esto se termina esta sección y se ha podido comprobar el funcionamiento del modelo en sus diferentes variantes.

3.3 Creación de un nuevo Dataset sintético para entrenamiento del modelo.

3.3.1 Introducción

En esta sección se describe la creación de un nuevo Dataset con imágenes sintéticas para poder entrenar la `YOLOP`, con esto se quiere hacer ver como va variando el rendimiento del modelo dependiendo del Dataset que se utilice para el entrenamiento.

Como se verá más adelante, el modelo funciona en situaciones en las que las condiciones son favorables y los distintos elementos son fácilmente diferenciables, en cambio cuando se tienen malas condiciones la inferencia realizada puede ser un poco mala.

Teniendo esto en cuenta, se puede llegar a incluso mejorar el rendimiento del modelo realizando más entrenamientos y que se pueda llegar a realizar la inferencia en cualquier situación, u orientarlo hacia el ámbito sintético como se irá viendo a lo largo de esta sección, para que también funcione con simuladores y no solo con imágenes del mundo real y así poder utilizarlo de forma local para poder realizar pruebas o ver como trabajaría el modelo.

Se utilizará para la generación de este Dataset el formato estándar definido por los creadores de la YOLOP, para así ajustarnos lo máximo posible al modelo y que no se nos presenten errores de tipado a la hora del entrenamiento, estos formatos son por ejemplo la utilización de imágenes en 1280x720, utilizar máscaras en blanco y negro y para las anotaciones archivos json.

Se pueden encontrar los scripts de Python [Python(2022)] que se han generado para este apartado en:

- generar-máscaras-anotaciones.py [Gavilanes(2022a)].
- extraer-imagenes-finales.py [Gavilanes(2022b)].

Por lo tanto, se empezará presentando el simulador CARLA [Carla(2022)],el cual servirá para recopilar datos en esta sección.

3.3.2 Instalación de CARLA

Desde los siguientes enlaces podremos acceder a la documentación de instalación:

- Instalación de los binarios de CARLA. https://carla.readthedocs.io/en/latest/build_carla/
- Instalación rápida. https://github.com/carla-simulator/carla/blob/master/Docs/download.md

Una vez se tiene CARLA instalado siguiendo las guías indicadas, el método para iniciar un servidor depende del método de instalación que hayamos utilizado y del sistema operativo sobre el que se ejecuta:

Instalación en Debian:

```
cd /opt/carla-simulator/bin/

./CarlaUE4.sh
```

Instalación en Linux:

```
cd ruta/a/carla/raíz

./CarlaUE4.sh
```

Instalación en Windows:

```
cd ruta/a/carla/raíz

CarlaUE4.exe
```

Aparecerá una ventana con una vista de la ciudad. Esta es la vista del espectador. Para volar por la ciudad se utiliza el ratón y las teclas 'WASD'. Además, manteniendo pulsado el botón derecho del ratón se puede controlar la dirección de la cámara.

Figura 3.6: Carla Simulator

Éste es el simulador del servidor que ahora se está ejecutando y esperando a que se realice una conexión como cliente e interactuemos con él. Se pueden probar algunos de los scripts de ejemplo para generar movimiento en la ciudad:

```
#Terminal A
cd PythonAPI\examples

python -m pip install -r requisitos.txt

python generate_traffic.py  #Se genera tráfico en la ciudad

#Terminal B
cd PythonAPI\examples

python manual_control.py # Se genera un coche autónomo
```

Con esto se habrá terminado de instalar y probar el correcto funcionamiento de CARLA, este paso es importante ya que este simulador es con el que se trabajará más adelante.

3.3.3 ¿Qué partes componen un Dataset?

Un Dataset, podría definirse como una de las partes que conforman el Big Data. Esto se construye alrededor de su propio concepto, siendo que la traducción de Dataset es un conjunto de datos.

Ahora bien, estos datos representan un conjunto particular de información, representada en una especie de tabla o matriz de análisis. La tabla se conforma por columnas, y cada columna representa una variable de datos, y las filas, representan un grupo de datos específicos.

Es decir, las filas podrían considerarse como las categorías de los datos, y las columnas, las variables particulares que la conforman. Esta combinación entre columnas y filas, es lo se conoce y define pues, como un Dataset.

Para nuestro caso particular, nuestro Dataset estará compuesto de 4 categorías, las imágenes, las máscaras de las líneas de la carretera, la máscara del área de la zona conducible y las anotaciones de cada objeto que contiene la imagen, este último será almacenado en archivos .json como se mostrará más adelante.

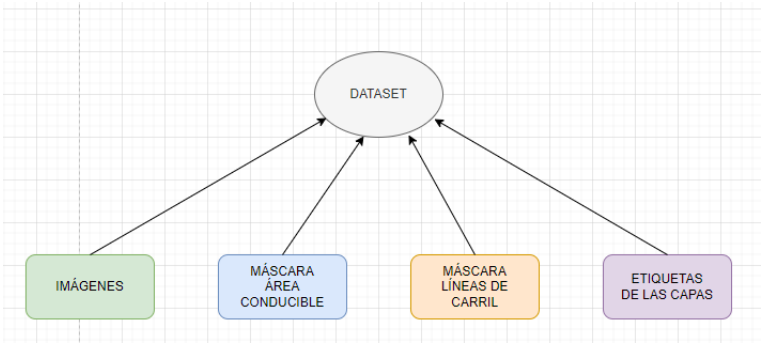


Figura 3.7: Arquitectura del Dataset

Con todos estos datos reunidos, ya se podría generar un Dataset con el que entrenar el modelo `YOLOP`.

A continuación, se podrá ir viendo como se irá generando o extrayendo cada una de las categorías de las que se compone el nuevo Dataset.

3.3.4 Extracción de imágenes de CARLA

Se empieza con la extracción de imágenes desde el simulador CARLA, para ello, se ha creado un script en Python [Python(2022)] para conectarse al simulador y así poder hacer capturas creando un vehículo virtual.

Para ello se consultará la documentación sobre sensores y señales que proporciona CARLA que se puede consultar en la siguiente página: https://carla.readthedocs.io/en/latest/core_sensors/ [Simulator(2022)]

Ahora es el momento de construir un script que será explicado paso a paso:

```
import glob
import os
import sys
try:
```

```
        sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
            sys.version_info.major,
            sys.version_info.minor,
            'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
    except IndexError:
        pass
    import carla
    import random
    import keyboard as kb
```

Código 3.2: Extracto 1 de extraerImagenesFinales.py

Como se puede ver, se empezará importando todas las librerías de Python [Python(2022)] que se necesitan para poder extraer las imágenes, además de añadir la ruta donde se encuentra el ".egg"de CARLA, que sirve para poder identificar la distribución con la que se está trabajando y que se puede utilizar a posteriori para conectarse al servidor.

```
    #Definimos el tamanno de la imagen que se creara.
    IM_WIDTH = 1280
    IM_HEIGHT = 720
    # Nos conectaremos al servidor de carla y generaremos los puntos de spawn
    client = carla.Client('127.0.0.1', 2000)
    world = client.get_world()
    bp_lib = world.get_blueprint_library()
    spawn_point = random.choice(world.get_map().get_spawn_points())
    # Annadimos el vehiculo
    vehicle_bp = bp_lib.find('vehicle.lincoln.mkz_2020')
    vehicle = world.try_spawn_actor(vehicle_bp, spawn_point)
```

Código 3.3: Extracto 2 de extraerImagenesFinales.py

Se define el tamaño que tendrán las imágenes de salida, que para el caso concreto del proyecto serán de 1280x720. Además, se iniciará la conexión con el simulador pasándole en este caso la IP local y el puerto por defecto, el 2000. Se cogerán los datos del mundo del simulador y de los elementos que lo componen (Blueprints), y se definirá un punto aleatorio de aparición (spawn) para el vehículo que se está generando.

A continuación, se crea el vehículo donde se colocarán los sensores, que para este trabajo es el modelo *lincoln mkz*, y se añadirá al mundo, colocándolo aleatoriamente en el punto de aparición (spawn) que se ha definido anteriormente.

```
    # Annadimos tr fico
    for i in range(50):
        vehicle_bp = random.choice(bp_lib.filter('vehicle'))
        npc = world.try_spawn_actor(vehicle_bp, random.choice(world.get_map().get_spawn_points
    ()))

    # Hacemos que el tr fico se mueva solo
    for v in world.get_actors().filter('*vehicle*'):
        v.set_autopilot(True)

    # Colocamos la orientacion a la camara
    camera_init_trans = carla.Transform(carla.Location(x=2.5, z=0.7))
```

Código 3.4: Extracto 3 de extraerImagenesFinales.py

Para conseguir que las imágenes extraídas sean lo más reales posibles, lo que se hará será añadir vehículos por todo el mapa, y que estos se muevan automáticamente por las carreteras. Con esto lo que

se consigue es que haya tráfico continuo y así poder obtener imágenes más realistas y más situaciones distintas, mejorando así el rango de imágenes de pruebas que se generarán.

Además, se colocará la orientación de la cámara con la que capturar las imágenes. Se le darán estos valores para que apunte mejor hacia la carretera.

```
# Annadimos los sensores de las camaras
camera_bp = bp_lib.find('sensor.camera.rgb')
camera_bp.set_attribute('image_size_x', f'{IM_WIDTH}')
camera_bp.set_attribute('image_size_y', f'{IM_HEIGHT}')
camera = world.spawn_actor(camera_bp, camera_init_trans, attach_to=vehicle)

sem_camera_bp = bp_lib.find('sensor.camera.semantic_segmentation')
sem_camera_bp.set_attribute('image_size_x', f'{IM_WIDTH}')
sem_camera_bp.set_attribute('image_size_y', f'{IM_HEIGHT}')
sem_camera = world.spawn_actor(sem_camera_bp, camera_init_trans, attach_to=vehicle)
```

Código 3.5: Extracto 4 de extraerImagenesFinales.py

Se pasa ahora a la parte en la que se definen los sensores, como se ve simplemente se busca en los elementos que tiene definidos el mapa con el método 'find', eligiendo así la cámara RGB y la de segmentación semántica. Además, gracias a las variables definidas al principio del script se le asignará el tamaño con el que se generarán las imágenes de cada sensor. Con esto se consigue obtener imágenes del entorno real, y además otra máscara con el código de colores - https://github.com/carla-simulator/carla/blob/master/LibCarla/source/carla/image/CityScapesPalette.h [Palette(2022)] definido por CARLA para la segmentación de elementos del entorno.

Por último, para esta parte, se añadirán los sensores dentro del mapa con el método "spawn Actor", al cual se le pasarán las cámaras que se acaban de crear, la orientación que deben tener, y que se definió anteriormente, y se anexionará al vehículo que se creó al principio para que mientras circule éste, se vayan capturando imágenes en cada frame o imagen.

```
# Definimos funciones que almacenar n las im genes
def rgb_callback(image):
    image.save_to_disk('./images/%06d.jpg' % image.frame)

def sem_callback(image):
    image.save_to_disk('./output_seg/%06d.png' % image.frame, carla.ColorConverter.
CityScapesPalette)

# Llamada a las funciones cada vez que se genera un frame nuevo
camera.listen(lambda image: rgb_callback(image))
sem_camera.listen(lambda image: sem_callback(image))
```

Código 3.6: Extracto 5 de extraerImagenesFinales.py

Gracias al método "save to disk", se conseguirá guardar cada una de las imágenes que generen los sensores. Para las imágenes normales, se guardarán directamente y le se le asignará el nombre del número del frame en el que se hayan generado. Pero en cambio, para las imágenes semánticas se les ha de pasar primero por un conversor de color para que se le asignen los colores [Palette(2022)] que tiene predeterminado CARLA para cada uno de los elementos.

Las funciones de almacenamiento recibirán como parámetro la imagen en crudo. Y éstas a su vez, serán llamadas cada vez que cada uno de los sensores capture datos, todo esto gracias al método "listen", que ayuda a ejecutar código o una acción cada vez que el sensor se active.

```
print("1- Extrayendo imagenes, pulsa q para parar.")
while True:
```

```
        if kb.is_pressed("q"):
            break

    # Paramos los sensores
    camera.stop()
    sem_camera.stop()
```

Código 3.7: Extracto 6 de extraerImagenesFinales.py

En este último paso de la captura de imágenes, se tiene un bucle que ejecutará el script hasta que se decida pararlo con la letra 'q'. Con esto ya se estará guardando tantas imágenes como sea capaz el simulador de capturar mientras se esté ejecutando.

Además, cuando se detiene el programa, se procederá a detener los sensores de las cámaras también, para que así se libere memoria del ordenador.

```
print("2- Criba de imagenes correctas.")
#Descartamos los fotogramas que no nos interesan
archivos1 = os.listdir("C:/CARLA_0.9.12/WindowsNoEditor/PythonAPI/examples/images/")
archivos2 = os.listdir("C:/CARLA_0.9.12/WindowsNoEditor/PythonAPI/examples/output_seg/")
ruta1 =("C:/CARLA_0.9.12/WindowsNoEditor/PythonAPI/examples/images/")
ruta2 =("C:/CARLA_0.9.12/WindowsNoEditor/PythonAPI/examples/output_seg/")
for i in archivos1:
    if(not (os.path.exists(ruta2+i))):
        os.remove(ruta1+i)

for j in archivos2:
    if(not (os.path.exists(ruta1+j))):
        os.remove(ruta2+j)

print("3- La extraccion de imagenes a terminado correctamente!.")
print("Rutas de guardado de las imagenes extraidas")
print("=========================================")
print("Imagenes originales: C:/CARLA_0.9.12/WindowsNoEditor/PythonAPI/examples/images/")
print("m scara segmentacion sem ntica: C:/CARLA_0.9.12/WindowsNoEditor/PythonAPI/
examples/output_seg/")
```

Código 3.8: Extracto 7 de extraerImagenesFinales.py

En esta última parte lo que se pretende es revisar los directorios donde se han almacenado las imágenes capturadas, y se almacenarán únicamente las que coincidan tanto en la parte de imágenes reales como en las imágenes segmentadas. Con esto se consigue simplificar la creación de las máscaras de las imágenes y las anotaciones que se necesitan para el Dataset que se pretende crear.

Con esto ya tendríamos el script preparado para poder generar imágenes de forma totalmente automática para el nuevo Dataset de imágenes sintéticas.

3.3.4.1 Ejecución del Script

Para esta parte se empezará ejecutando el simulador CARLA como se comentó en la imagen 3.6, para que se pueda establecer conexión con éste y extraer imágenes. Hecho esto, se pasará a ejecutar el script siguiente:

```
cd <Path del directorio de Carla>/PythonAPI/examples/

python extraer_Imagenes_Finales.py
```

Con lo que se consigue por una parte las imágenes y por otra la máscara de la segmentación semántica:

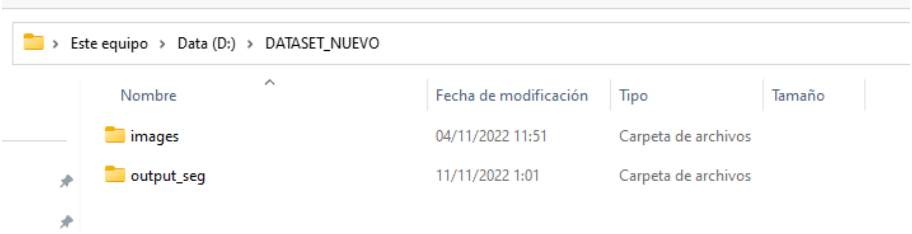


Figura 3.8: Estructura de directorios para las imágenes extraídas.

Dentro de la carpeta ”images”, se hará una diferenciación entre las que se utilizarán para el entrenamiento (”train”) y entre las que se utilizarán para validación (”val”). Por lo que la estructura que queda dentro de esta carpeta es la siguiente:

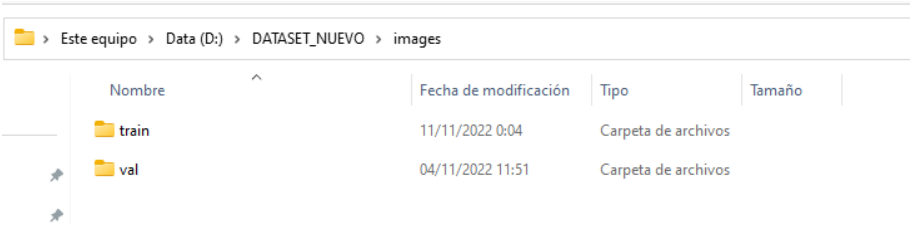


Figura 3.9: Estructura del directorio imágenes.

Y dentro de cada una se habrán almacenado las imágenes que gracias al script que anteriormente se ha explicado, se ha podido crear:

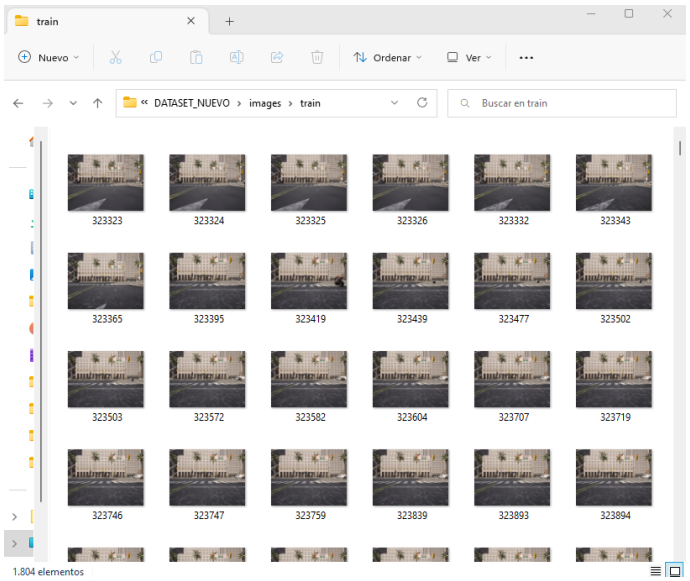


Figura 3.10: Imágenes generadas.

Por otro lado, se tiene que en la raíz de la carpeta de segmentación semántica (”output seg”), se guardarán todas las imágenes con la máscara de colores de la segmentación, para que a partir de éstas ya se puedan generar más adelante las máscaras, tanto para las carreteras como para las líneas de carril, en las carpetas correspondientes como se mostrará a continuación:

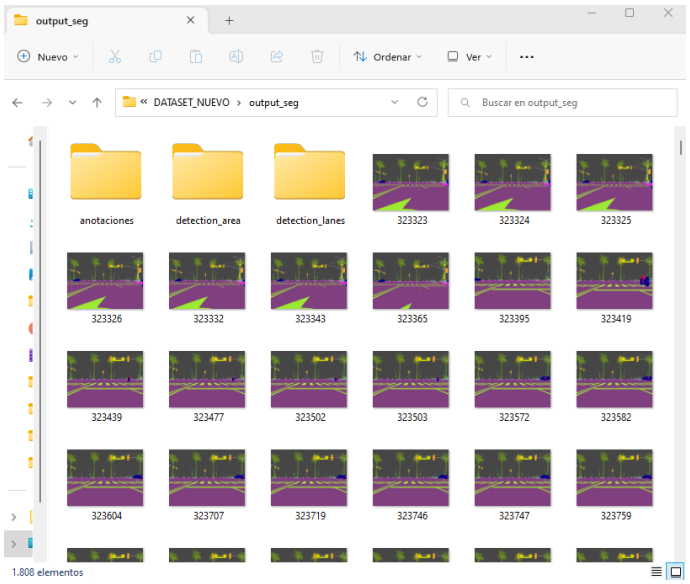


Figura 3.11: Imágenes de la segmentación semántica generadas.

Con esto ya se da por terminado el primer elemento que se necesita para la generación del Dataset, que son las imágenes en las que se basará el modelo para generar el entrenamiento.

Además, como se explica durante este punto, se generarán a la par las imágenes con los colores de la segmentación semántica que nos da Carla, con las que se conseguirá construir las máscaras para las carreteras y las líneas de carril.

3.3.5 Generar máscaras

Para la generación de las dos máscaras que necesita el Dataset, como son la de la zona conducible de la carretera y de las líneas del carril, se partirá del conjunto de imágenes semánticas que se consiguió en el apartado anterior. Con estas imágenes como punto de partida, y que gracias a que Carla define un color para cada uno de los elementos que componen el mapa, se simplificará bastante la obtención de estas máscaras.

Además, un punto importante a tener en cuenta para este apartado es que OpenCV trabaja con el formato de color BGR [BGR(2022)] y no RGB [RGB(2022)]. Con esto ya se pueden generar las máscaras mediante un script que se ha creado para este apartado y que se irá desglosando en los siguientes apartados.

El siguiente script (generar-mascaras-anotaciones.py) será capaz de generar automáticamente las máscaras necesarias, así como las anotaciones para las mismas, todo desde un único sitio. Con esto se consigue poder generar de una manera muchísimo más eficaz y rápida todos los elementos que se necesitan, y a la vez colocarlos dónde les corresponde de una sola vez.

```
import cv2
import os
import numpy as np
import json
directorio_imagenes = './'
print("========================================================================")
print("Generando anotaciones en: ./anotaciones/train/ ")
print("Generando mascaras de zona conducible en: ./detection_area/train/ ")
print("Generando mascaras de lineas de carril en: ./detection_lanes/train/ ")
print("========================================================================")
with os.scandir(directorio_imagenes) as ficheros:
```

```
        for fichero in ficheros:
            if (fichero.name.endswith('.png')):
```

Código 3.9: Imports del script generador de máscaras.

Como se ve, en un primer lugar, se tienen los imports correspondientes a todas las librerías que se utilizarán durante el programa, tales como OpenCV (cv2), una que nos permite el control del SO (so), la librería numpy [NumPy(2022)] y por último la json, con la que se conseguirá generar las anotaciones en dicho formato.

Además, se ha definido el directorio donde se estará trabajando, y gracias al bucle principal 'for' se irá recorriendo cada uno de los archivos que encuentre en formato PNG. En este caso, estos archivos son los que corresponderán con las imágenes que se han obtenido anteriormente de la parte semántica de CARLA, así que se puede ir tratando cada una por separado y generando cada parte que interese.

Por último, se imprimirá por pantalla la ruta donde se almacenarán cada uno de los 3 elementos del nuevo Dataset, para que así el usuario pueda identificarlos de una manera más simple.

3.3.5.1 Generar máscaras de las líneas de carril

Para empezar en este apartado se define el color [Palette(2022)] que utiliza CARLA para las líneas del carril, que en este caso es el (157u, 234u, 50u)RGB. Con esta información se ha conseguido extraer/generar las máscaras para las líneas del carril.

A continuación, se pasará a desglosar la metodología que se ha seguido para este componente:

```
#MASCARAS
img = cv2.imread(fichero.name)
# m scara de las lineas de la carretera BGR
mask2 = cv2.inRange(img, (50,234,157), (50,234,157))
filename2 = "./detection_lanes/train/"+fichero.name
cv2.imwrite(filename2, mask2)
```

Código 3.10: Generando la máscara de la línea de carril.

Primero, se ve como se lee la imagen del directorio, gracias al método 'imread' de OpenCV. Con esto ya se podrá trabajar con cada archivo que se tenga dentro del directorio de trabajo.

Después, gracias al método 'inRange' y que la imagen segmentada está dividida en colores, se puede extraer y separar sólo el color que corresponde con las líneas de carril. Aunque en este caso, y como se ha mencionado antes, OpenCV espera recibir la codificación de colores en formato BGR y no en formato RGB. Con esto ya se tiene el objeto mask2, que correspondería con la máscara en blanco y negro de las líneas de carril correspondiente a la imagen tratada.

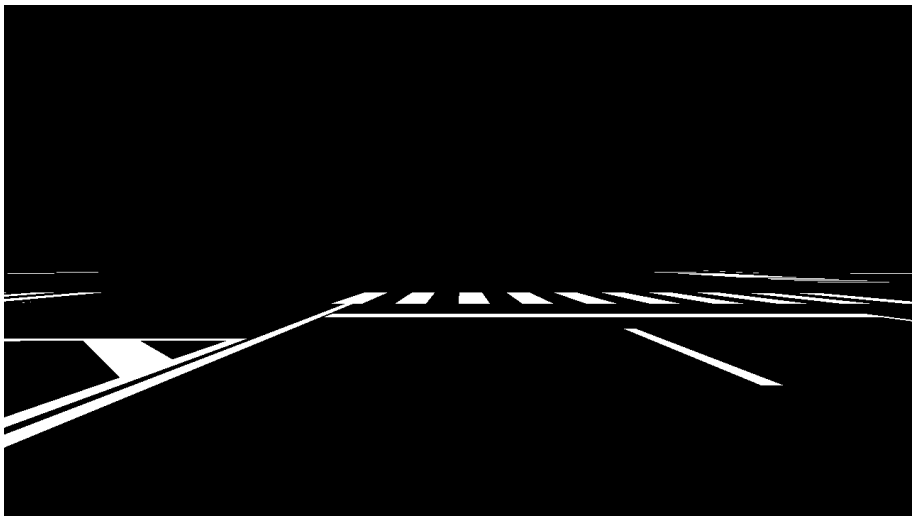


Figura 3.12: Máscara de la línea de carril.

Solo quedaría guardar dicha máscara asignándole un path o ruta y utilizando el método 'imWrite' para guardarlo donde se le ha indicado.

3.3.5.2 Generar máscaras de la zona conducible de la carretera

Para las máscaras de la zona conducible se debe tener en cuenta que CARLA utiliza el color (128u, 64u, 128u)RGB. Con esto ya se puede pasar a generar la máscara correspondiente.

```
#MASCARAS
img = cv2.imread(fichero.name)
# m scara de la carretera BGR
mask1 = cv2.inRange(img, (128, 64, 128), (128, 64, 128))
filename1 = "./detection_area/train/"+fichero.name
cv2.imwrite(filename1, mask1)
```

Código 3.11: Generando la máscara de la zona conducible.

Como se ve en el código anterior, se han seguido los mismos pasos que con la obtención de la máscara de las líneas de carril, por ende, simplemente se definirá el color que se quiere abstraer y con ello se puede conseguir la máscara para la zona conducible y almacenarla en el path que se le ha asignado. Un ejemplo de máscara de zona conducible que se genera es el siguiente:

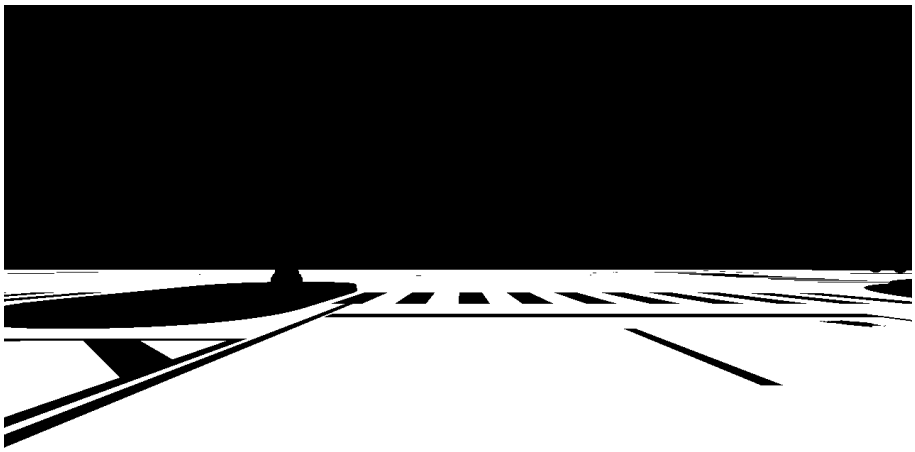


Figura 3.13: Máscara de la zona conducible.

Una vez se tiene está máscara generada, se tiene ya tres de los cuatro elementos del nuevo Dataset, como son las imágenes capturadas del simulador, y las máscaras para las líneas de carril y de la zona conducible de cada una de estas.

3.3.6 Extracción de anotaciones para las máscaras

Éste será el último elemento que se necesita para completar el nuevo Dataset, gracias a las anotaciones de las imágenes se conseguirá darles un contexto a las máscaras generadas.

Las anotaciones consisten en ficheros ".json" los cuales contendrán, como punto más importante, las coordenadas con las que se pueden definir tanto las máscaras, como los vehículos que se detectan durante la conducción.

Por lo tanto, es un elemento crucial en el Dataset, ya que con esto se consigue unir las imágenes planas con las máscaras que se han generado de éstas.

Como punto de partida, se mirará como vienen definidas las anotaciones de los ficheros del Dataset Original para tener un contexto y así definir una plantilla a la que poder pasarle toda la información que se genera mediante este script.

Para ello se empezará con lo siguiente:

```
data = {}
nombre= fichero.name.split('.')
data['name'] = nombre[0]
data['frames'] = []
data["attributes"]= {
"weather": "clear",
"scene": "city street",
"timeofday": "daytime"
}
```

Código 3.12: Definiendo la estructura del json.

Como se puede ver, se empezará definiendo un array de datos donde se almacenarán todos los atributos que debe contener el "json".

En primer lugar, se identifica que se le debe asignar un nombre, que se corresponderá con el archivo que durante el ciclo del bucle toque trabajar, pero obviando gracias al método 'split' la extensión de éste.

Después, también tiene que contener el elemento ”frames”, el cuál contendrá toda la información relevante a las coordenadas de las máscaras y de las Bounding Box, o cajas de contorno, de los vehículos detectados. Así que se definirá como un elemento vacío, hasta que más adelante cuando se obtenga la información de las coordenadas de cada elemento, se pueda rellenar.

Y, por último, como se ve en la imagen, se tiene el elemento 'attributes', el cual se puede dejar vacío ya que simplemente contiene información relevante al mapa que no es útil ahora mismo, pero por darle algo más de realismo, se le ha asignado esos valores por defecto.

Con esta estructura de datos generada se puede pasar a capturar las coordenadas para las máscaras:

```
#cogemos los datos para las zonas conducibles
img=cv2.imread(filename1,cv2.IMREAD_GRAYSCALE)
#degradamos la imagen para poder coger mejor las coordenadas de la zona conducible
kernel = np.ones((3,5), np.uint8)
#img_dilation = cv2.dilate(img, kernel, iterations=1)
img_erosion = cv2.erode(img, kernel, iterations=1)
_, threshold = cv2.threshold(img_erosion, 110, 255, cv2.THRESH_BINARY)
contours, _= cv2.findContours(threshold, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
area=""
for cnt in contours :
    approx = cv2.approxPolyDP(cnt, 0.009 * cv2.arcLength(cnt, True), True)
    n = approx.ravel()
    i = 0
    for j in n :
        if(i % 2 == 0):
            x = n[i]
            y = n[i + 1]
            if(i == 0):
                area+='['+str(x)+','+str(y)+', \"L\"],'
            else:
                area+='['+str(x)+','+str(y)+', \"L\"],'
        i = i + 1
#cogemos los datos para las lineas
img=cv2.imread(filename2,cv2.IMREAD_GRAYSCALE)
#degradamos la imagen para poder coger mejor las coordenadas de las lineas
kernel = np.ones((1,5), np.uint8)
#img_dilation = cv2.dilate(img, kernel, iterations=1)
img_erosion = cv2.erode(img, kernel, iterations=1)
_, threshold = cv2.threshold(img_erosion, 110, 255, cv2.THRESH_BINARY)
contours, _= cv2.findContours(threshold, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
lineas=""
for cnt in contours :
    approx = cv2.approxPolyDP(cnt, 0.009 * cv2.arcLength(cnt, True), True)
    n = approx.ravel()
    i = 0
    for j in n :
        if(i % 2 == 0):
            x = n[i]
            y = n[i + 1]
            if(i == 0):
                lineas+='['+str(x)+','+str(y)+', \"L\"],'
            else:
                lineas+='['+str(x)+','+str(y)+', \"L\"],'
        i = i + 1
```

Código 3.13: Captura de coordenadas para las dos máscaras.

Como se ve en el código anterior, lo primero que se hará es leer en cada caso la máscara del directorio que toque, ya que están separadas en su correspondiente carpeta. Además, se pasará estas máscaras a escalas de grises para facilitar el tratamiento de las mismas, ya que se utiliza en esta ocasión librerías como Numpy y OpenCV que trabajan muy bien con estos colores planos.

Con la imagen leída, se procede a degradar un poco la imagen. Esto se hace con la única finalidad de poder conseguir puntos más concretos con los que se definirán las coordenadas y con los que se generan las máscaras, ya que haciendo las línea más gruesas y expandiendo la imagen se evita coger pequeños detalles o puntos irrelevantes en las máscaras que lo único que harían sería errar el entrenamiento, ya que no aportan nada en sí mismos.

Ahora bien, una vez se tiene la imagen tratada, se puede empezar a definir las coordenadas gracias al método 'findCountours' de OpenCV, el cuál da las coordenadas de las esquinas del elemento que se le ha pasado, en este caso la imagen ya tratada.

Una vez se tienen estas coordenadas, se aproximan al formato de PolyLinea, ya que se trata del formato con el que trabaja la YOLOP para este tipo de elementos.

Por otro lado, ya que estos datos se tienen que almacenar dentro del "json", se crean unas variables área y línea, respectivamente, en las cuales se almacenará la información que se saca de cada máscara para después poder utilizarla.

De esta forma ya se tiene adquirida toda la información que se necesita de las máscaras de la zona conducible y de las línea de carril, con lo que se puede continuar con la creación de la estructura para las anotaciones.

3.3.7 Extracción de anotaciones BoundingBox vehículos

En el caso de las BoundingBox, que simplemente se tratan de cajas que se dibujarán alrededor de los vehículos para poder señalizarlos de forma correcta, se seguirá una metodología similar al de las máscaras anteriores.

```
#datos de los vehiculos en las imagenes
gray = np.all(img == (142,0,0), 2)  # Cogemos el color de los vehiculos que tiene Carla
# Convert logical matrix to uint8
gray = gray.astype(np.uint8)*255
# Find contours
cnts = cv2.findContours(gray, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)[-2]  # Use index
[-2] to be compatible to OpenCV 3 and 4
if cnts != ():
# Get contour with maximum area
    c = max(cnts, key=cv2.contourArea)
    x1, y1, w, h = cv2.boundingRect(c)
else:
    x1, y1, w, h = 0,0,0,0
```

Código 3.14: Captura de coordenadas para las Bounding Box de los vehículos.

En este caso, ya que no se tiene ninguna máscara de la que partir, se empezará separando los vehículos del resto de la imagen, esto se logra gracias a la segmentación de colores que ofrece CARLA y que anteriormente se adquirió la capa de segmentación a la hora de generar imágenes de prueba. Para ello, en este caso el color que se le asigna a los vehículos en CARLA es el (142,0,0)RGB.

En el caso de las máscaras se parte de una imagen determinada de la que extraer información, pero en el caso de las BoundingBox se extraerá la figura de los vehículos por colores y no se tiene una imagen

como tal de partida, así que se debe convertir esta matriz de datos de sólo los vehículos que se ha generado temporalmente en algo con lo que OpenCV pueda trabajar, por lo que se pasa al formato UNINT8, que es un formato estándar que sí entenderá la librería.

A partir de aquí, ya se puede trabajar con los datos de los vehículos como si de una imagen se tratara, por lo que se hace lo mismo que en el apartado anterior. Gracias al método 'findContours' se puede generar las coordenadas para las cajas que se dibujarán sobre puestas en las imágenes alrededor de los vehículos.

Antes de almacenar esta información en coordenadas, se debe pensar si las imágenes contienen o no coches en ellas, por lo que, si no cuentan con ningún coche dentro de ellas, se hará que éstas sean cero y no se dibuje ninguna caja. En caso contrario, se necesita pasar las coordenadas, al igual que se hizo con las máscaras anteriores, a un formato que la YOLOP entienda, en este caso BOX2D. Esto se logra gracias a los métodos 'max' y 'boundingRect', que generan 4 puntos que corresponderán a cada una de las esquinas del rectángulo que se sobrepone sobre el vehículo dentro de la imagen y que se almacenará en las variables x1, y1, w, h respectivamente.

Teniendo esto, finalmente ya se tendría la información necesaria para poder detectar los vehículos de forma automática dentro de cualquier imagen del Dataset, y se puede empezar a pasarle esta información a la estructura principal, para que se genere la anotación final de la imagen.

3.3.8 Generar Anotación Final

Gracias a los puntos anteriores se dispondrá de toda la información relevante de cada imagen pudiendo así rellenar la estructura principal con los datos que se han adquirido a lo largo del script.

```
#Creamos la estructura  final del json
data['frames'].append({"timestamp": 10000,
'objects':[{
        "category": "car",
        "id": 0,
        "attributes": {
            "occluded": "none",
            "truncated": "none",
            "trafficLightColor": "none"
        },
        "box2d": {
            "x1": x1,
            "y1": y1,
            "x2": x1+w,
            "y2": y1+h
        }},
        {
        "category": "area/drivable",
        "id": 1,
        "attributes": {},
        "poly2d": [
            area
        ]
    },
    {
        "category": "lane/single white",
        "id": 2,
        "attributes": {
            "direction": "parallel",
            "style": "solid"
```

```
            },
            "poly2d": [
                lineas
            ]
        }]})
```

Código 3.15: Se rellena el elemento frames con los datos obtenidos.

Con esto ya se cuenta con toda la información que hacía falta sobre cada imagen, pudiendo así generar el archivo ”json” correspondiente, con las anotaciones particulares de cada una de éstas como se ve a continuación:

```
    with open('./anotaciones/train/'+nombre[0]+'.json', 'w') as file:
                json.dump(data, file, indent=4)
```

Código 3.16: Se genera el archivo json con la estructura de datos.

Se puede ver como gracias a la librería json de Python, se le puede pasar una estructura de datos y ésta generará un fichero con el formato correcto de un archivo ”json” donde se le indique.

3.3.9 Dataset Final

Llegados a este punto se puede ver como gracias a estos dos archivos, extraerImagenesFinales.py y generar-MascarasAnotaciones.py, se ha podido generar de una forma totalmente automática todos los elementos que hacían falta para poder generar el Dataset de imágenes sintéticas del simulador CARLA, pudiendo así empezar a entrenar el modelo YOLOP con los nuevos datos.

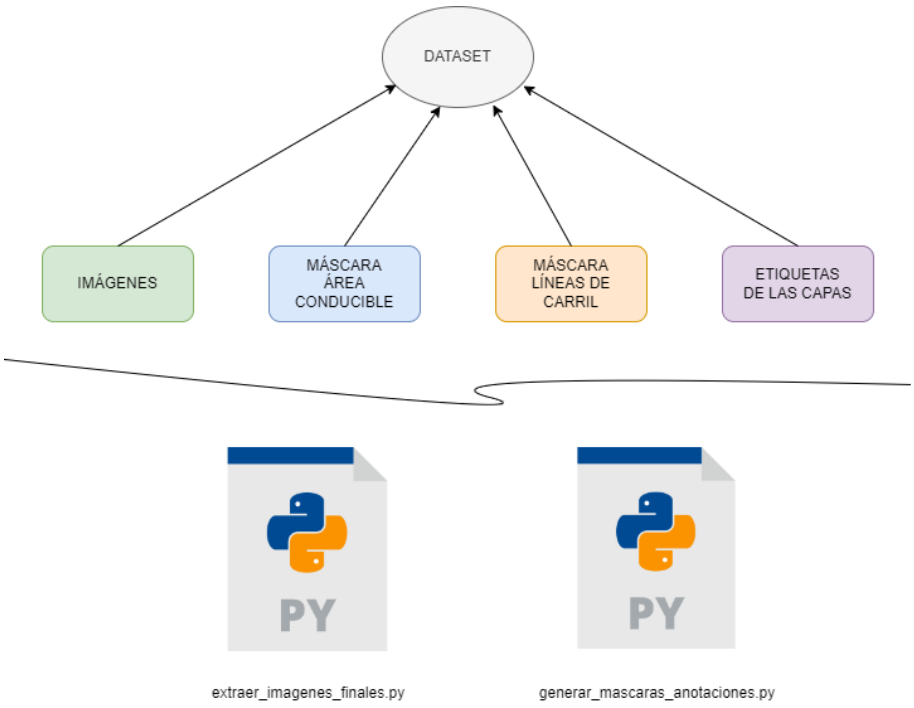


Figura 3.14: Estructura final del Dataset.

3.4 Desarrollo de interfaz web para prueba de funcionamiento del modelo.

3.4.1 Introducción

Para esta parte se hará uso de la segmentación de las distintas partes que genera el modelo `YOLOP`, la zona conducible de la carretera, la detección de carriles y la detección de vehículos para crear una página web en la que cualquiera pueda subir una imagen y consiga ver el funcionamiento del modelo en tiempo real, sin necesidad de hacer instalaciones en local.

Figura 3.15: Interfaz Web del Modelo YOLOP

3.4.2 Generación de pesos del modelo

Se empieza explicando como genera el modelo YOLOP la segmentación de las distintas partes.

Todo se centra en un único archivo testonnx.py - `https://github.com/hustvl/YOLOP/blob/main/test_onnx.py`, con el cual se le pasará una imagen de inferencia y la segmentará.

Pero para ello antes se hará uso de la tecnología ONNX (Open Neural Network Exchange) [ONNX(2022)], la cual se puede definir como un ecosistema de código abierto que ayuda a representar modelos de Deep Learning. Esta tecnología es clave a la hora de representación de modelos de aprendizaje automático ya que consigue exportarlos y que los desarrolladores se muevan más fácilmente entre marcos, ya que una vez que se realiza la exportación se puede hacer uso de distintas herramientas software para desplegarlo (Pythorch, Tensor RX, etc.) y se puede ejecutar con independencia del hardware que se esté utilizando, ya que éste se adaptará. Con esto se consigue que, al exportar el modelo, su utilización sea más flexible y se adapte a cualquier entorno en el que se quiera implementar.

En consecuencia, se empieza exportando nuestro modelo al estándar ONNX para poder trabajar con él más fácilmente. Para ello se tiene el archivo ”exportonnx.py”, con el que se puede exportar el modelo

pasándole como parámetro la resolución en la que se hará el tratamiento de imágenes. Para esta prueba se ejecutará la exportación a 1280x1280 px, aunque también se puede hacer a resoluciones tales como 640p ó 320p:

```
python export_onnx.py --height 1280 --width 1280
```

Con esto se consigue generar el archivo yolop-1280-1280.onnx, el cual se utiliza como base para poder trabajar con el modelo de una manera más sencilla dentro de la carpeta "weights".

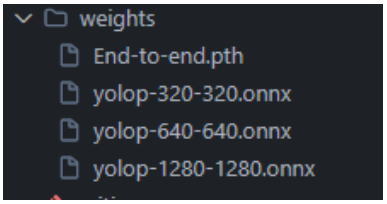


Figura 3.16: Pesos YOLOP

3.4.3 Segmentación de imágenes

A partir de aquí se probará como el modelo diseñado de forma local es capaz de realizar la segmentación al pasarle una imagen de prueba gracias al archivo testonnx.py, del cual se explicarán sus métodos más importantes a continuación:

- infer-yolop(weight,img-path): se puede definir como el método principal del archivo ya que lo que hace es primero convertir el color de la imagen a BGR, para así detectar cada parte del modelo gracias al archivo onxx, ya que éste le dará la información de como separar cada una de las partes pues contiene el ecosistema de nuestro modelo ya entrenado, y después creará máscaras de cada una de las partes gracias a la librería OpenCV de Python. Consiguiendo así generar imágenes que guardará en la carpeta "/pictures".
- resize-unscale(): sirve para poder reescalar y normalizar la imagen para poder así después guardarla con el tamaño original.

Ahora se probará con una imagen real, y veremos el funcionamiento de la segmentación que realiza el modelo:

```
python ./test_onnx.py --weight yolop-1280-1280.onnx --img test.jpg
```

Figura 3.17: Imagen a segmentar

(a) Área conducible (b) Líneas de carril (c) Detección de objetos

Figura 3.18: Segmentación de la imagen

3.4.4 Interfaz web

Ahora que se ha podido comprobar que la segmentación se realiza correctamente se pasa a generar la interfaz web de la aplicación.

Para ello se ha utilizado la ayuda de Gradio [Gradio(2022)], que es una biblioteca de Python de código abierto que permite crear rápidamente componentes de IU personalizables y fáciles de usar para modelos de machine learning en sólo unas pocas líneas de código.

Se empieza implementando la interfaz web en local, creando un archivo que se llamará app.py dentro del directorio raíz de la `YOLOP`.

```
import gradio as gr
import test_onnx

with gr.Blocks() as yolop:
    gr.Markdown(
    """
    <center>
    <u>

    # Escenario de prueba del modelo YOLOP
    </u>
    Escenario para probar el funcionamiento del modelo YOLOP, arrastra o sube una imagen
en la casilla     Original, y dandole al boton de "Segmentar Imagen" se veran las imagenes
 procesadas y segmentadas que     genera el modelo, cuando se quiera probar con otra
imagen se puede limpiar la que esta puesta y colocar     una nueva.
    </center>
    """)
    with gr.Row():
        with gr.Column():
            img = gr.Image(type="filepath",label="Original")
            segmentar = gr.Button(value="Segmentar Imagen")
        with gr.Column():
            img2 = gr.Image("C:/Users/JonathanRoberMoncada/Desktop/escritorio/yolop.png",
label="Arquitectura     YOLOP")
    with gr.Row():
        with gr.Column():
            s1 = gr.Image(label='Zona Conducible')
        with gr.Column():
            s2 = gr.Image(label='Lineas del Carril')
        with gr.Column():
            s3 = gr.Image(label='Vehiculos Detectados')
        with gr.Column():
            s4 = gr.Image(label='Imagen Final')
```

```
        segmentar.click(test_onnx.infer_yolop, inputs=img, outputs=[s1,s2,s3,s4])
        examples = gr.Examples(examples=[['./calle.png']],inputs=[img])
        gr.Markdown(
        """
        <center>
            <a href="https://github.com/hustvl/YOLOP"><b>GITHUB YOLOP</b></a>
        </center>
        """)

    yolop.launch()
```

Código 3.17: Código APP.

En esta ocasión se hará uso de la distribución por bloques que tiene Gradio, para así poder crear una interfaz bien distribuida e intuitiva. El primer paso que se da como se puede ver, es importar la librería de gradio además del archivo test-onnx que es en el que se basa esta aplicación para generar la segmentación y enseñarla.

Antes de nada, se hará hincapié en los cambios realizados en el archivo test-onnx.py:

```
    print("detect done.")
    img_merge_rgb = cv2.cvtColor(img_merge, cv2.COLOR_BGR2RGB)
    img_det_rgb = cv2.cvtColor(img_det, cv2.COLOR_BGR2RGB)
    return da_seg_mask,ll_seg_mask,img_det_rgb,img_merge_rgb
```

Código 3.18: Código añadido a test-onnx.py.

Como se ve en el código anterior se han añadido 3 líneas de código dentro de la función infer-yolop() que se encuentra del archivo test-onnx.py, con esto se consigue que la función devuelva como datos las imágenes segmentadas gracias a él:

```
return da_seg_mask,ll_seg_mask,img_det_rgb,img_merge_rgb
```

Pero como la función utiliza imágenes en BGR, se necesita convertir la imagen de salida de la detección de vehículos y la imagen final a RGB, por eso antes del método return se hará uso de OpenCV [OpenCV(2022)] para transformar el color de las imágenes a RGB. Para las máscaras de la detección de la zona conducible y la de detección de las líneas de carril no es necesario hacerlo, ya que son máscaras que van en blanco y negro. Con esto ya se tendría configurado el archivo para que retorne correctamente las imágenes que se necesitan para la interfaz web.

Volviendo al ”app.py” se empezará viendo que se utiliza la estructura de bloques de Gradio, por lo que se hace uso del método blocks(), el cual permite trabajar con este tipo de estructura y crear cuadrículas trabajando así con filas y columnas, creándolas con sus métodos correspondientes (Row() y Column()). Además, se utilizarán objetos dentro de cada bloque con los que se irán definiendo lo que aparecerá en la web.

Se empieza con la parte del título de la página y una pequeña explicación, que irán fuera de los bloques, para ello se utiliza la función Markdown(), que deja escribir texto plano o en formato HTML.

Figura 3.19: Sección del título y subtítulo.

Después se pasa a añadir contenido en la primera columna de la primera fila, añadiendo un recuadro donde poder subir una imagen, y de la que se obtiene el path de dónde se almacena, para después pasárselo como parte de los parámetros que necesita la función infer-yolop() para realizar la segmentación. Además, en esta primera parte se coloca un botón que llame a esta función para que así la interfaz llame a la función y se realice la segmentación, con lo que se consigue que se nos presenten en la fila de abajo las imágenes de salida.

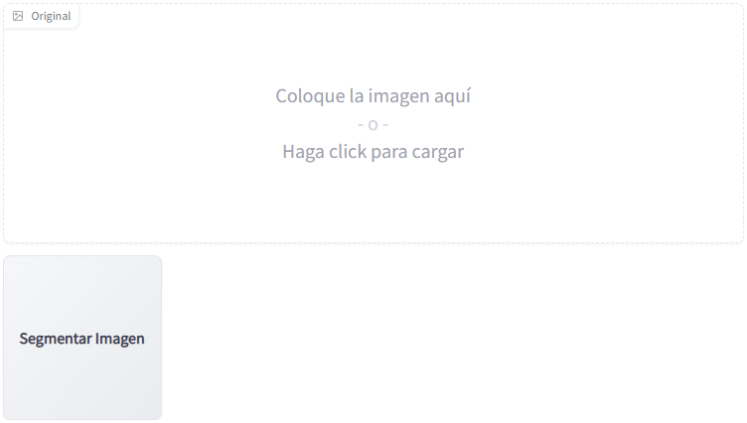


Figura 3.20: Contenedor de imagen original y botón de llamada a la función.

El siguiente bloque que se trata es en el que se coloca una imagen de la arquitectura del modelo `YOLOP` para mostrar las diferentes partes que se pueden conseguir gracias a éste, para ello simplemente se coloca un bloque de tipo imagen dentro de la segunda columna en la primera fila, quedando de las siguiente manera:

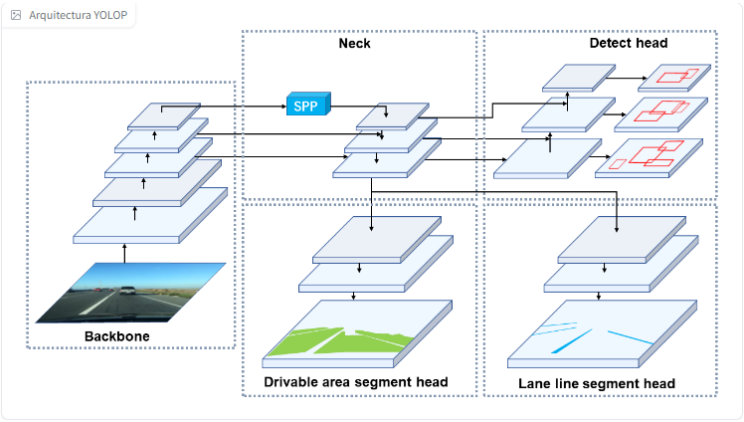


Figura 3.21: Imagen de la arquitectura del modelo.

Con esto se termina de configurar la primera fila, con lo que se pasa a la segunda. Aquí se encuentran las imágenes de salida de la segmentación. Para ello como se ve en la figura 3.15, se crean cuatro columnas en las que insertar cuatro secciones de tipo imagen. A cada una de ellas se le dará un nombre (s1,s2,s3,s4), con lo que se consigue asignarle a cada una de estas secciones las imágenes de salida de la función infer-yolop() respectivamente como se ve en la figura 3.25. Esta fila queda de la siguiente manera:

Figura 3.22: Contenedores de salida de la segmentación.

Con esto se habrá terminado de configurar los bloques con lo que se captura la entrada de la imagen a segmentar, además de poder mostrar todas las imágenes de salida que el modelo es capaz de generar. El último bloque que se muestra por pantalla es el de la imagen de ejemplo, aunque este es un bloque que se genera mediante un método reservado de Gradio, que viene ya pre-configurado, con lo que sólo se le pasará el path de la imagen a mostrar y la mostrará en un bloque dentro de la interfaz web:

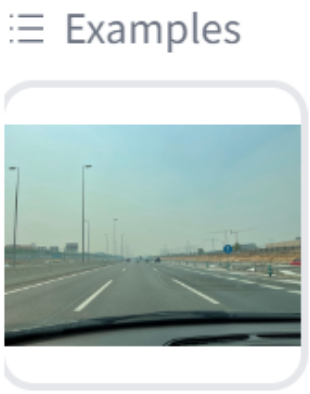


Figura 3.23: Imagen de ejemplo.

Sólo queda ver la parte final, en la que como al principio se hará uso de lenguaje Markdown para poner un enlace hacia el repositorio original en el que se ha basado:

Figura 3.24: Enlace hacia el repositorio YOLOP.

Con esto se termina de comentar todos los elementos de la parte visual de la interfaz.

Sólo queda explicar como funciona internamente la aplicación web, y como se ha visto anteriormente en el código 3.15, en la línea 30, se utiliza el método click() de la clase button().

A este método se le tienen que pasar mínimo tres parámetros, la función a la que llamar cuando se acciona el botón, los elementos de entrada y los elementos de salida.

Así que como se ve, se utiliza el nombre que se le ha asignado a cada bloque para poder hacer referencia a cada uno de ellos más fácilmente, con lo que al final el primer parámetro que se le pasará será el de la función encargada de segmentar las imágenes 'infer-yolop()', el segundo parámetro serán los datos de entrada a la función, para ello como se ha visto al principio se ha creado un bloque especial para poder subir imágenes al que se ha llamado 'img', además servirá como parámetro de entrada.

Como último punto del método click() se tienen las imágenes de salida que genera la función y las cuales se las asignaremos a los bloques de imágenes de la segunda fila respectivamente.

Con esto se consigue que cuando un usuario suba una imagen y pulse el botón, la web llame a la función internamente y que se presenten las imágenes de salida del modelo por pantalla.

```
segmentar.click(test_onnx.infer_yolop,inputs=img,outputs=[s1,s2,s3,s4])
```

Gracias a todo lo anteriormente realizado se muestra una prueba de funcionamiento de la aplicación web, con una imagen real que se ha capturado:

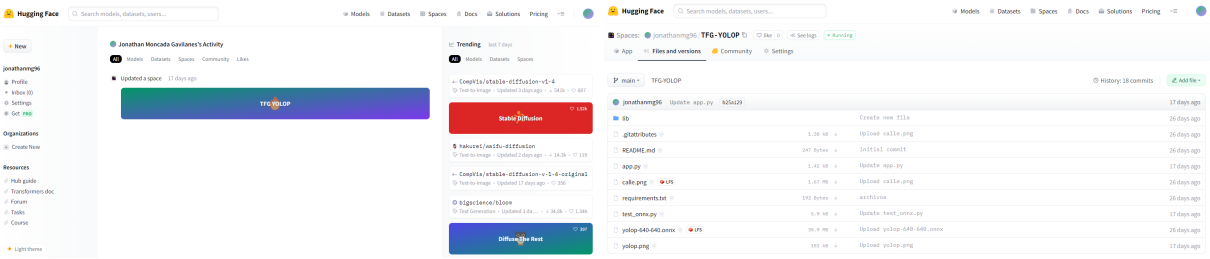
Figura 3.25: Prueba de funcionamiento web.

Como apunte final sobre la interfaz web, se tiene que saber que para cada bloque se ha usado el atributo label, con el que se puede dar un nombre a cada uno de ellos, que será mostrado cuando se lance la aplicación web en vez del nombre por defecto que tienen asignado.

3.4.5 Hosting web - HuggingFace

Una vez que se ha comprobado que la aplicación web funciona correctamente en local, el siguiente paso es poder subirlo a un hosting web para que pueda ser accesible desde cualquier ordenador y que sirva a cualquier persona que quiera probar el funcionamiento del modelo.

Para esta parte se ha elegido la web HuggingFace [Huggingface(2022)], la cual es una plataforma que sirve para poder crear un entorno donde poder instalar todo lo necesario para implementar la aplicación web. Ahora bien, lo primero que se debe hacer es iniciar sesión y crear un espacio de trabajo al cual se llamará TFG-YOLOP, donde subir todos los archivos necesarios para la aplicación:



(a) Login HuggingFace. (b) Archivos dentro del espacio de trabajo.

Figura 3.26: Creación de la aplicación web en HuggingFace

Como se ve en la figura 3.26b, los archivos que se han necesitado son:

- requirements.txt: aquí se tiene apuntado todas las librerías de Python que se necesitan tener en el espacio de trabajo.
- app.py: archivo donde escribir la aplicación web.
- test-onnx.py: archivo del modelo YOLOP con el que segmentar las imágenes.
- yolop-640-640.onnx: archivo que se genera tras la exportación del modelo en formato ONNX.
- *.png: distintas imágenes que se necesitan para la interfaz de la aplicación.

Con todo esto se generará en el espacio de trabajo la aplicación web que puede ser consultada en: https://huggingface.co/spaces/jonathanmg96/TFG-YOLOP:

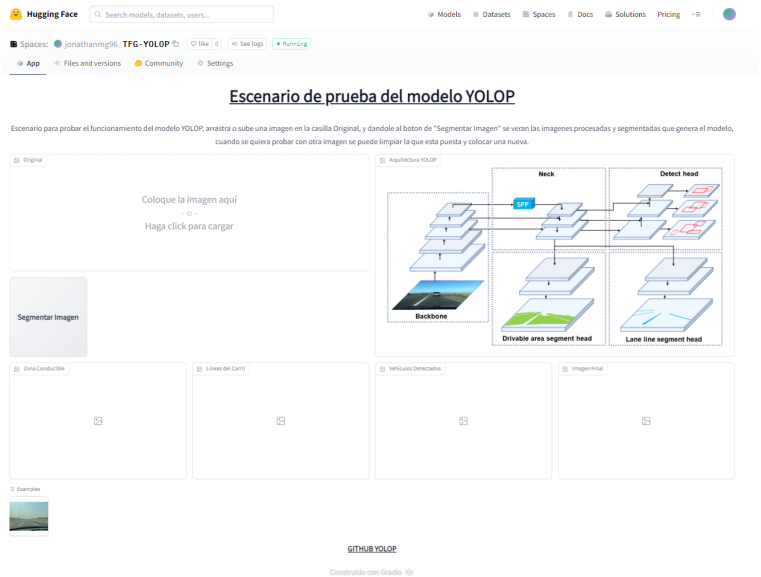


Figura 3.27: Aplicación web en HuggingFace.

Y como se puede ver a continuación sigue funcionando de la misma manera que en local:

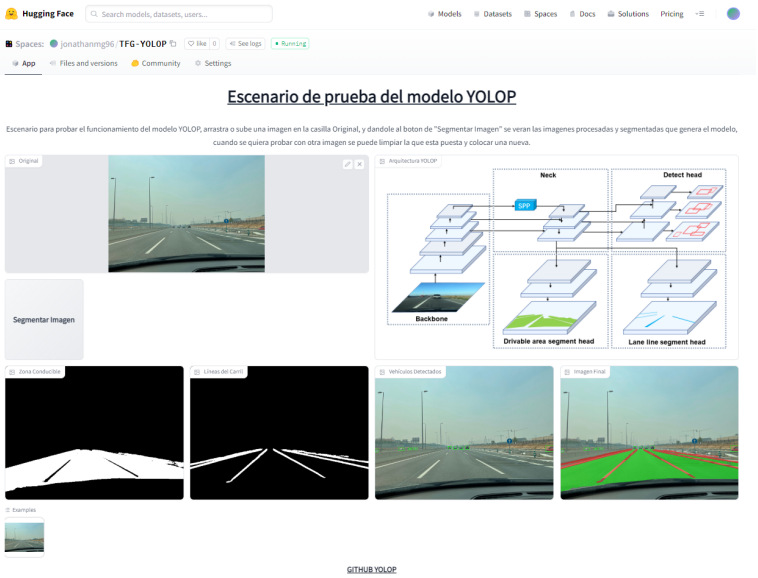


Figura 3.28: Funcionamiento en HuggingFace.

Con esto se termina este apartado del desarrollo, ya que se ha conseguido implementar en la web de HuggingFace el modelo `YOLOP`.

3.5 Conclusiones del capítulo

Como se ha visto a lo largo de esta sección, se ha conseguido demostrar el correcto funcionamiento del modelo YOLOP y que se puede implementar en cualquier entorno local, tanto para la inferencia en imágenes como para vídeos.

Además, gracias al simulador CARLA, se han podido extraer imágenes para que después se pueda hacer un tratamiento de estas y así crear un Dataset que la red YOLOP entienda. Para ello se han creado y explicado dos scripts en Python, los cuales realizan todas las tareas necesarias para la creación del Dataset de forma totalmente automática y eficaz, simplificando todo el largo proceso.

Por último, se puede ver como se ha desplegado el modelo mediante una interfaz web y se ha puesto a disposición de cualquier usuario que quiera probar el funcionamiento del modelo YOLOP. Esta web se ha diseñado de tal manera que sea lo más fácil e intuitiva posible para facilitar su uso para cualquier usuario.

Capítulo 4

Resultados

4.1 Introducción

En este capítulo, se mostrarán los resultados globales obtenidos en este trabajo. Se realiza el entrenamiento del modelo YOLOP en el que se basa este proyecto, y los resultados que se han conseguido como consecuencia de éste.

Además, poder contrastar los sistemas desarrollados, se ha hecho una comparación del Dataset que se ha conseguido generar con las imágenes sintéticas de CARLA frente al modelo y las imágenes originales que la YOLOP proporcionaba, y así con estos últimos datos conseguir unos resultados base con los que realizar la comparación final.

4.2 Entrenamiento del modelo YOLOP con el nuevo Dataset

Para empezar a realizar el entrenamiento, se comenzará instalando la tecnología NVIDIA CUDA Toolkit [CUDA(2022)], la cual ayuda a que algunas librerías de Python, que utiliza la YOLOP para realizar el entrenamiento, sean capaces de utilizar y comunicarse con la tarjeta gráfica que se tiene instalada. Para esta prueba se ha utilizadla la NVIDIA RTX 2070 super.

```
PS D:\YOLOP-main> nvidia-smi
Sun Nov 27 15:06:56 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 526.98       Driver Version: 526.98       CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ... WDDM  | 00000000:01:00.0  On |                  N/A |
|  0%   42C    P8    22W / 215W |    663MiB /  8192MiB |     10%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      4444    C+G   ...n1h2txyewy\SearchHost.exe    N/A      |
|    0   N/A  N/A      7944    C+G   C:\Windows\explorer.exe         N/A      |
|    0   N/A  N/A      8824    C+G   ...artMenuExperienceHost.exe    N/A      |
|    0   N/A  N/A      9856    C+G   ...v1g1gvanyjgm\WhatsApp.exe    N/A      |
|    0   N/A  N/A     10672    C+G   ...e\PhoneExperienceHost.exe    N/A      |
|    0   N/A  N/A     13540    C+G   ...2txyewy\TextInputHost.exe    N/A      |
|    0   N/A  N/A     14168    C+G   ...\Programs\Blitz\Blitz.exe    N/A      |
|    0   N/A  N/A     14828    C+G   ...obeNotificationClient.exe    N/A      |
|    0   N/A  N/A     15600    C+G   ...icrosoft VS Code\Code.exe    N/A      |
|    0   N/A  N/A     15980    C+G   ...(x86)\AnyDesk\AnyDesk.exe    N/A      |
|    0   N/A  N/A     16652    C+G   ...y\ShellExperienceHost.exe    N/A      |
+-----------------------------------------------------------------------------+
PS D:\YOLOP-main>
```

Figura 4.1: Drivers de NVIDIA instalados correctamente.

Una vez se tiene instalado el software de NVIDIA [Nvidia(2022)], se pasará a instalar la librería que ayudará a que se pueda establecer la conexión con la gráfica. En este caso, ya que se utiliza PyTorch [Pytorch(2022)], se instalarán todos los componentes necesarios directamente con el comando pip, como hemos visto en anteriores apartados:

```
pip3 install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu116
```

También está el caso en el que no se pueda contar con una gráfica para realizar el entrenamiento, en ese caso no haría falta instalar nada de lo anteriormente indicado, ya que se utilizaría la CPU. Pero hay que tener en cuenta que con esto se alargaría enormemente los tiempos de entrenamiento del modelo ya que los núcleos CUDA de la gráfica ayudan bastante en este proceso.

Con esto instalado, ya se puede proceder a realizar nuestro entrenamiento. Pero antes de eso se ha de poner en contexto, que elementos son necesarios para llevarlo a cabo. Como se ha mostrado cuando se creó el Dataset, se requieren 4 elementos fundamentales para poder entrenar el modelo YOLOP, como son:

- Imágenes.
- Máscaras de la zona conducible.
- Máscaras de las líneas de la carretera.
- Anotaciones en formato json.

Como ya se ha descrito en el apartado 3.3, se ha podido generar de una manera totalmente automática todo el conjunto de elementos gracias a los dos scripts que se han creado. Con ello sólo queda ajustar y tener en cuenta una serie de puntos para el entrenamiento.

Primero, y bastante importante, se ha de definir una arquitectura de directorios que el modelo sea capaz de entender, por lo que se ha seguido la siguiente estructura:

```
-DatasetNuevo
-- imagenes
    -- train
    -- val
-- output_seg
    -- Anotaciones
        -- train
        -- val
    -- Detection_area
        -- train
        -- val
    -- Detection_lanes
        -- train
        -- val
```

Código 4.1: Arquitectura de carpetas para el entrenamiento.

Como se puede ver, estará disponible para todos los directorios de los distintos elementos una carpeta denominada train y otro a la que se ha llamado val.

Se ha de tener en cuenta que la carpeta de val almacenará imágenes contra las que se irán realizando inferencias mientras se esté lanzando el entrenamiento, y así poder ver resultados. Si no se tuviera esta carpeta, el entrenamiento no podría ejecutarse ya que no se tendría contra que probar los valores que se van obteniendo del entrenamiento y por ende los valores finales siempre serían nulos.

Y por otra parte en la carpeta train se meterán los elementos que se han ido creando durante este trabajo, donde correspondan.

Para esta prueba de entrenamiento y para a posteriori poder comparar resultados los únicos valores que irán cambiando serán los de la carpeta 'train', ya que de primeras se utilizarán las imágenes originales que ofrece la YOLOP para entrenar su modelo, pero a posteriori se cambiarán con las que se han generado nuevas. Y así, las imágenes que se tienen en la carpeta VAL siempre serán las mismas para los dos tipos de entrenamiento, ya que servirá para que se pueda comparar como se han ido comportando cada uno de ellos, ya que si no se modifica la carpeta VAL se irán realizando inferencias sobre las mismas imágenes en los entrenamientos y así se verán los resultados particulares que se obtienen en cada caso.

Con esto dicho, se va a plantear el siguiente punto, que será ver que archivos del modelo se utilizarán para configurar y lanzar el entrenamiento del modelo.

Primero, para la configuración del entrenamiento se hará uso del archivo que se encuentra dentro de la YOLOP y que se denomina default.py y que se puede encontrar en la siguiente ruta:

```
~\YOLOP\lib\config\default.py
```

De este archivo se configurará la ruta de donde el modelo indexará las imágenes para el entrenamiento, modificando las siguientes líneas:

```
_C.DATASET.DATAROOT = 'D:/DATASET_NUEVO/images/'
_C.DATASET.LABELROOT = 'D:/DATASET_NUEVO/output_seg/anotaciones/'
_C.DATASET.MASKROOT = 'D:/DATASET_NUEVO/output_seg/detection_area/'
_C.DATASET.LANEROOT = 'D:/DATASET_NUEVO/output_seg/detection_lanes/'
```

Código 4.2: Modificación de las rutas para el entrenamiento.

Como se ve, simplemente lo primero que se hará es modificar la ruta de acceso de donde se ha almacenado el nuevo Dataset.

```
# if training 3 tasks end-to-end, set all parameters as True
# Alternating optimization
_C.TRAIN.SEG_ONLY = True           # Only train two segmentation branchs
_C.TRAIN.DET_ONLY = True           # Only train detection branch
_C.TRAIN.ENC_SEG_ONLY = True       # Only train encoder and two segmentation branchs
_C.TRAIN.ENC_DET_ONLY = True       # Only train encoder and detection branch

# Single task
_C.TRAIN.DRIVABLE_ONLY = True      # Only train da_segmentation task
_C.TRAIN.LANE_ONLY = True          # Only train ll_segmentation task
_C.TRAIN.DET_ONLY = True          # Only train detection task
```

Código 4.3: Modificar tipo de entrenamiento.

El siguiente punto será modificar las líneas en las que se define el tipo de entrenamiento que se va a realizar. La YOLOP, deja realizar entrenamientos por separado para cada una de las partes que la componen, pero en esta ocasión y ya que se quiere hacer un entrenamiento completo, se pondrán todos los parámetros a TRUE como se indica en los comentarios, ya que es así como se puede realizar un entrenamiento END-TO-END de todas las capas, significando esto que el entrenamiento será completo y es justamente lo que se pretende en este caso.

```
_C.TRAIN.BEGIN_EPOCH = 0
_C.TRAIN.END_EPOCH = 25
```

Código 4.4: Modificar Tiempo de entrenamiento.

Lo siguiente a tener en cuenta son las épocas o etapas que durará el entrenamiento, esto quiere decir, cuantas veces se realizará el entrenamiento END-TO-END. Por lo que se ha configurado en este entrenamiento para que haga 25 etapas, con esto se consigue ver como para cada etapa de este, el modelo irá arrojando datos que consigue mediante la inferencia de las fotos de la carpeta de validación.

```
_C.LOG_DIR = 'runs/'
_C.GPUS = (0,1)
_C.WORKERS = 6
```

Código 4.5: Modificar parámetros

Por último, para esta parte de configuración, se hablará de la ruta donde se almacenarán los datos del entrenamiento, en este caso es configurable y se ha decidido almacenar en la carpeta 'runs/'. Además, se indica a la YOLOP que haga uso de la GPU 0 en primer lugar, ya que por defecto al montar la gráfica en el ordenador ésta recibirá el nombre de 'device 0' con lo que así se puede configurar el uso de la misma. Además, se configurará el número de hilos a utilizar para este entrenamiento, en este caso y debido al hardware empleado, Pytorch advierte que lo más recomendable son 6 hilos o 'workers'.

Con todos estos parámetros configurados ya sólo queda lanzar el entrenamiento y generar resultados. Esto se logrará gracias al archivo 'train.py' que ofrece la YOLOP:

```
python /tools/train.py
```

4.3 Comparativa

Se empezará planteando el escenario que se va a comparar, primero de todo se ha obtenido 6379 imágenes sintéticas para el nuevo Dataset, y de ellas se han generado sus respectivas máscaras para la zona conducible y las líneas de la carretera. Por último, también se tienen las anotaciones para cada una de ellas. Todo esto gracias a que se procesa automáticamente cada una de ellas con los scripts que se explicaron en el apartado 3.3.

Por otro lado, se tienen también las imágenes de la base de datos BD100K con las que trabaja la YOLOP. De éstas se cogerán el mismo número de fotos, siendo así 6379 imágenes. Para que se puedan comparar los resultados de la manera más pareja posible. De estas, se contarán también con sus otros elementos que faltan para que se pueda hacer el entrenamiento, como son las distintas máscaras y las anotaciones de las imágenes, que se pueden encontrar dentro de los enlaces que tiene la YOLOP en su github - https://github.com/hustvl/YOLOP.

Con esto ya se tiene dos Dataset con los que realizar el entrenamiento y comparar resultados. Otro punto que se ha mencionado anteriormente y que es crucial para que los resultados que se obtengan de los dos entrenamientos tengan un punto en común, serán las imágenes contra las que se realiza la validación de los dos entrenamientos, por lo que en este caso estas serán las mismas para los dos entrenamientos. Con esto se consigue ver qué Dataset arroja mejores resultados al realizar la inferencia sobre estas imágenes de validación.

Con todo esto dicho, se empieza el entrenamiento de la YOLOP con las imágenes de la BDD100K, y posteriormente se procederá a realizar el entrenamiento con el nuevo Dataset sintético.

Una vez se realiza el primer entrenamiento se consiguen los siguientes datos, que como ya se ha mencionado en el apartado anterior, se guardarán en la carpeta 'runs/'. Estos resultados están disponibles para su consulta en el siguiente enlace:

Resultados con imágenes del Dataset BDD100k. https://universidaddealcala-my.sharepoint.com/:f:/g/personal/jonathan_moncada_edu_uah_es/EhazqK_WGn9GlzlLBKaeBWsBI51TeSifwloYfaqgr28Qmg?e=Z8FRZv

Ahora se procede al entrenamiento del modelo con el nuevo Dataset de imágenes sintéticas de CARLA, con lo que al finalizar éste se consiguen los siguientes resultados, que se pueden consultar en el siguiente enlace:

Resultados con imágenes del Dataset de imágenes sintéticas de CARLA. https://universidaddealcala-my.sharepoint.com/:f:/g/personal/jonathan_moncada_edu_uah_es/EueWHtbHwFJJltR3VeUepncBwEtLsLuV1NFaH3rJRxRYiQ?e=RnfnwW

Una vez se consiguen datos de los dos entrenamientos, se procede a comparar resultados y finalmente se realizará una inferencia sobre imágenes de prueba para ver si al realizar el entrenamiento con los distintos Dataset se consigue mejorar o empeorar el rendimiento del modelo.

Como parte común al principio de los ficheros logs de cada entrenamiento que se pueden encontrar en los enlaces que antes se han compartido, se puede ver que la YOLOP muestra toda la configuración que se está utilizando para el entrenamiento, además de que Pytorch avisa qué gráfica se ha utilizado durante el proceso:

```
2022-11-24 11:55:24,978 Namespace(conf_thres=0.001, dataDir='', iou_thres=0.6, local_rank=-1,
    logDir='runs/', modelDir='', prevModelDir='', sync_bn=False)
2022-11-24 11:55:24,978 AUTO_RESUME: False
CUDNN:
  BENCHMARK: True
  DETERMINISTIC: False
  ENABLED: True
DATASET:
  COLOR_RGB: False
  DATAROOT: D:/DATASET_NUEVO/images/
  DATASET: BddDataset
  DATA_FORMAT: jpg
  FLIP: True
  HSV_H: 0.015
  HSV_S: 0.7
  HSV_V: 0.4
  LABELROOT: D:/DATASET_NUEVO/output_seg/anotaciones/
  LANEROOT: D:/DATASET_NUEVO/output_seg/detection_lanes/
  MASKROOT: D:/DATASET_NUEVO/output_seg/detection_area/
  ORG_IMG_SIZE: [720, 1280]
  ROT_FACTOR: 10
  SCALE_FACTOR: 0.25
  SELECT_DATA: False
  SHEAR: 0.0
  TEST_SET: val
  TRAIN_SET: train
  TRANSLATE: 0.1
DEBUG: False
GPUS: (0, 1)
LOG_DIR: runs/
LOSS:
  BOX_GAIN: 0.05
  CLS_GAIN: 0.5
  CLS_POS_WEIGHT: 1.0
  DA_SEG_GAIN: 0.2
  FL_GAMMA: 0.0
  LL_IOU_GAIN: 0.2
```

```
  LL_SEG_GAIN: 0.2
  LOSS_NAME:
  MULTI_HEAD_LAMBDA: None
  OBJ_GAIN: 1.0
  OBJ_POS_WEIGHT: 1.0
  SEG_POS_WEIGHT: 1.0
MODEL:
  EXTRA:

  HEADS_NAME: ['']
  IMAGE_SIZE: [640, 640]
  NAME:
  PRETRAINED:
  PRETRAINED_DET:
  STRU_WITHSHARE: False
NEED_AUTOANCHOR: False
PIN_MEMORY: False
PRINT_FREQ: 20
TEST:
  BATCH_SIZE_PER_GPU: 24
  MODEL_FILE:
  NMS_CONF_THRESHOLD: 0.001
  NMS_IOU_THRESHOLD: 0.6
  PLOTS: True
  SAVE_JSON: False
  SAVE_TXT: False
TRAIN:
  ANCHOR_THRESHOLD: 4.0
  BATCH_SIZE_PER_GPU: 24
  BEGIN_EPOCH: 0
  DET_ONLY: True
  DRIVABLE_ONLY: True
  ENC_DET_ONLY: True
  ENC_SEG_ONLY: True
  END_EPOCH: 25
  GAMMA1: 0.99
  GAMMA2: 0.0
  IOU_THRESHOLD: 0.2
  LANE_ONLY: True
  LR0: 0.001
  LRF: 0.2
  MOMENTUM: 0.937
  NESTEROV: True
  OPTIMIZER: adam
  PLOT: True
  SEG_ONLY: True
  SHUFFLE: True
  VAL_FREQ: 1
  WARMUP_BIASE_LR: 0.1
  WARMUP_EPOCHS: 3.0
  WARMUP_MOMENTUM: 0.8
  WD: 0.0005
WORKERS: 6
num_seg_class: 2
2022-11-24 11:55:24,994 Using torch 1.11.0+cu113 CUDA:0 (NVIDIA GeForce RTX 2070 SUPER, 8191MB
    )
2022-11-24 11:55:24,994
2022-11-24 11:55:25,269 freeze encoder and Det head...
2022-11-24 11:55:25,294 freeze encoder and two Seg heads...
```

```
2022-11-24 11:55:25,316 freeze Det head...
2022-11-24 11:55:25,324 freeze two Seg heads...
2022-11-24 11:55:25,333 freeze encoder and Det head and Da_Seg heads...
2022-11-24 11:55:25,351 freeze encoder and Det head and Ll_Seg heads...
2022-11-24 11:56:00,025 anchors loaded successfully
2022-11-24 11:56:01,066 tensor([[[0.3750, 1.1250],
         [0.6250, 1.3750],
         [0.5000, 2.5000]],

        [[0.4375, 1.1250],
         [0.3750, 2.4375],
         [0.7500, 1.9375]],

        [[0.5938, 1.5625],
         [1.1875, 2.5312],
         [2.1250, 4.9062]]], device='cuda:0')
```

Código 4.6: Cabecera común de los logs de los entrenamientos.

Como se puede ver al principio de las líneas anteriores del fichero, la variable 'prevModelDir' está vacía, esto quiere decir que el entrenamiento que se está haciendo del modelo es completo y no es un simple perfilamiento del mismo.

A partir de aquí empezaran las etapas o épocas de entrenamiento del modelo:

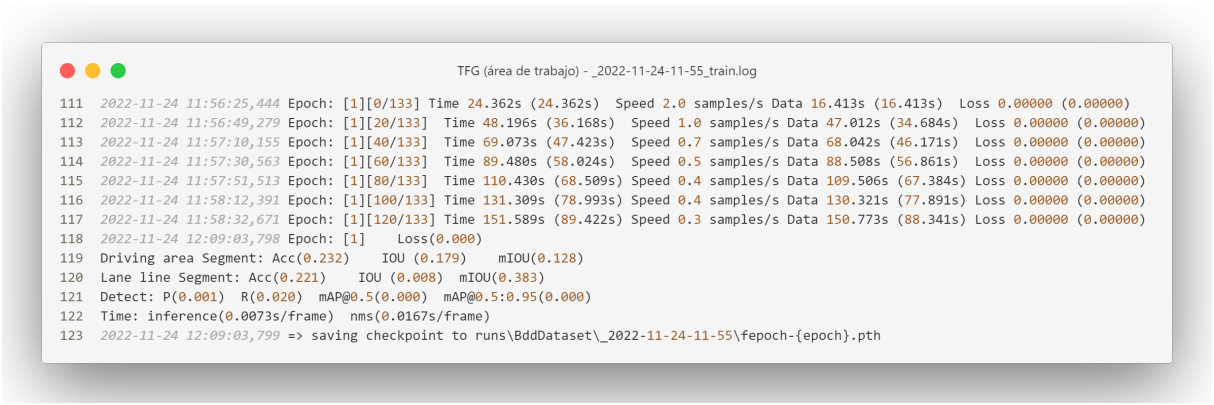


Figura 4.2: Época 1 - Dataset Original.

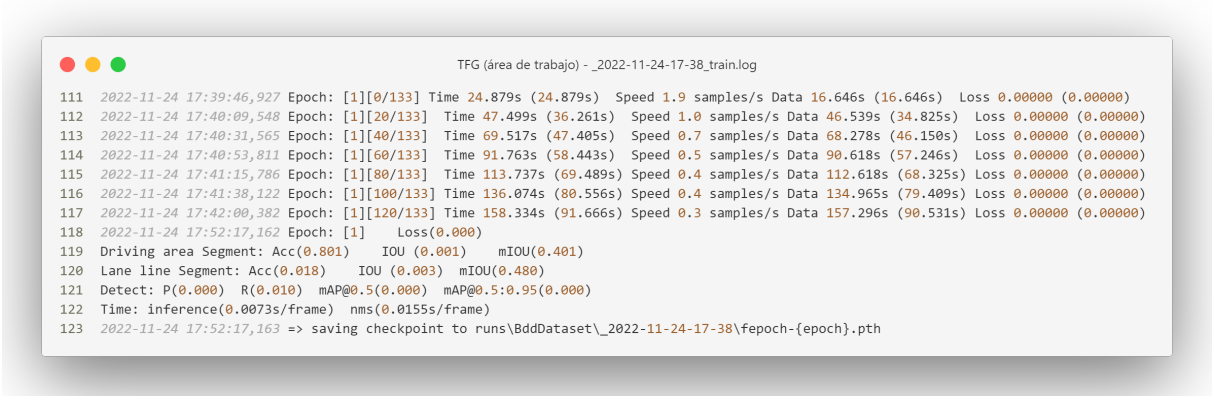


Figura 4.3: Época 1 - Dataset nuevo sintético.

Primero de todo se ha de tener en cuenta que el programa de entrenamiento muestra tanto la hora como la fecha en la cual se está realizando el entrenamiento, en este caso cada uno de los entrenamientos han tardado en torno a 5 horas y esto se podrá ver claramente con el log de la última época que se realiza.

Otro punto importante, son los tiempos que tarda en realizar cada parte de la que se compone una época, y la velocidad con la que hace el tratamiento de las muestras que componen el Dataset.

Cuando termina una época, se muestran los resultados que se han obtenido de ésta con respecto a la inferencia que hace con las imágenes de validación que se tiene. Así, los resultados más importantes que se obtienen para esta parte son los de la precisión (Acc), con los que se puede ver como se encuentra el modelo en esta etapa del entrenamiento. Hay que tener en cuenta que, si se está obteniendo un 'Acc' de 0, los resultados no serían válidos, ya estaría diciendo que el entrenamiento no tiene precisión alguna y no merecería seguir con el mismo.

Para la detección de vehículos es algo distinto, pero el parámetro que mide la precisión en este caso es el 'R'.

Con esto dicho, se puede ver que en la primera imagen corresponde con la primera época de entrenamiento con las imágenes del Dataset original y la segunda, con la primera época del entrenamiento del modelo con las imágenes sintéticas, con lo que examinando los resultados se puede observar que en un primer momento los resultados del Dataset original son superiores en la parte de detección de líneas de carretera y en la detección de vehículos, pero sin embargo ya desde un primer momento el nuevo Dataset supera la precisión del original a la hora de detectar la zona conducible de la carretera. Con respecto a los tiempos de procesado, se ve que no difieren mucho, pero esto tiene mucho sentido, ya que el número de imágenes con las que se están entrenando el modelo es el mismo.

Además, ya se puede ver otro dato interesante que muestra el entrenamiento, como es el de cuánto tiempo tarda en hacerse la inferencia de una imagen, el tiempo es muy bajo e igual en ambos casos así que este dato no resulta muy relevante por el momento, pero aun así se puede decir que al ser muy bajo el tiempo, los resultados que se están obteniendo para poder hacer una inferencia de una imagen en el futuro es muy óptimo.

Por último, cada vez que se termina una época el programa guardará un checkpoint de como va esta fase del entrenamiento, por si se produce algún error que se pueda dar la posibilidad de retomarlo desde dicho checkpoint.

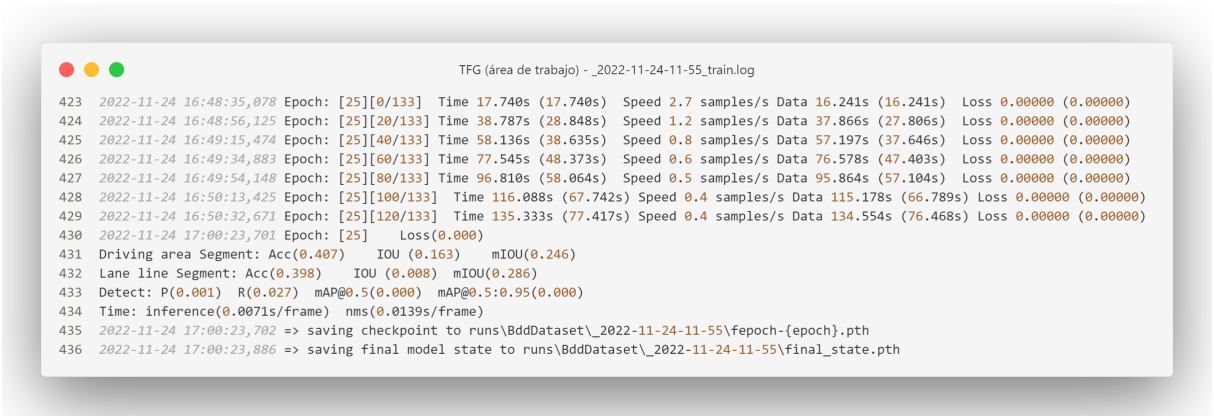


Figura 4.4: Época 25 - Dataset Original.

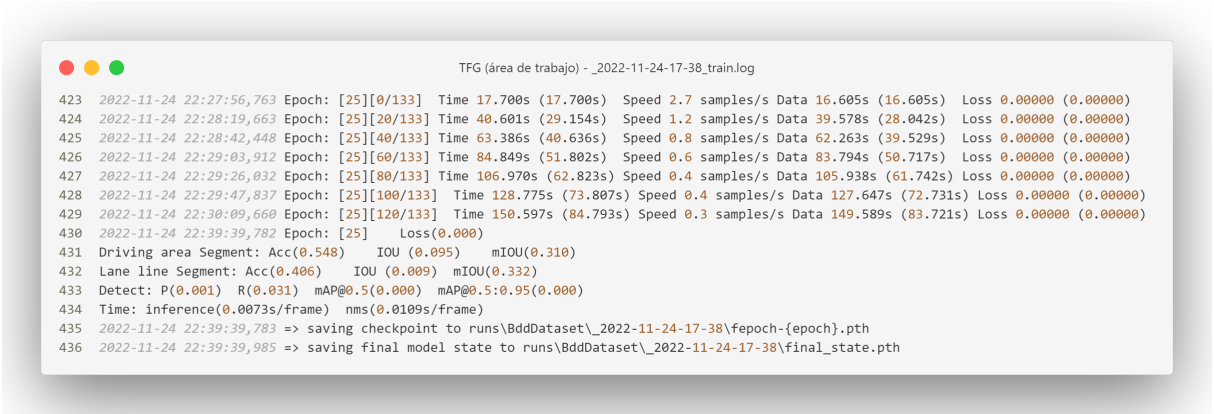


Figura 4.5: Época 25 - Dataset nuevo sintético.

Ahora se pasará a ver como se ha terminado de comportar el entrenamiento con los dos Dataset, revisando la etapa final que aparece en los logs de éstos. Como se ha dicho en anteriores apartados, se ha configurado estos entrenamientos para que duren los dos 25 épocas, por eso la última época que se ve en los logs es ésta.

Como primer punto, se puede ver que en este caso estas últimas épocas han tardado menos en procesarse con el Dataset de imágenes originales que con el de imágenes sintéticas, esto no es muy relevante para los resultados, simplemente se resaltan estos valores porque al final el tiempo de procesamiento de cada época ayuda a ver como se han ido indexando y tratando los datos a la hora del entrenamiento y si las imágenes y datos que se le pasan han sido más o menos óptimas. Pero al final, al no tener valor para los resultados finales y como no distan muchos los tiempos, estos valores se pueden pasar por alto.

Además, en este punto final se puede ver la hora a la que ha terminado cada entrenamiento, y se comprobará que cada uno de los entrenamientos al estar compuestos de las mismas imágenes tardaran prácticamente lo mismo, siendo el tiempo global que tarda en hacerse cada uno de unas 5 horas.

Ahora como punto más importante, se verá la precisión final que se ha conseguido en cada entrenamiento. Para el primer el primer valor de precisión, que corresponde con la detección de zona conducible, se ve que el Dataset original consigue una puntación de **0.407** y por contra parte se tiene que con el nuevo Dataset se ha conseguido incrementar este valor hasta la puntuación de **0.548**, siendo este segundo resultado un **15 %** mejor que el primero, porque lo que se ha conseguido mejorar el modelo en gran medida para este punto.

Con respecto a la detección de las líneas de la carretera se ha obtenido un valor con el Dataset original de **0.398**, frente al valor de **0.406** que consigue el nuevo Dataset. Aquí se puede ver que la diferencia es pequeña, pero aún así, se ha conseguido mejorar la precisión con este nuevo conjunto de imágenes sintéticas, por lo que aunque sea poca la diferencia en 25 épocas si se siguiese entrenando el modelo durante más tiempo o se utilizaran más imágenes en el nuevo Dataset, se conseguiría refinar aun más los resultados y la diferencia en este apartado crecería.

Por último, se tiene el parámetro de la detección de vehículos, se tiene en este caso que para el entrenamiento del modelo con el Dataset original se ha conseguido un resultado final de precisión de **0.027**, frente a los **0.031** puntos de precisión que ha conseguido el nuevo Dataset de imágenes sintéticas. Como se ve, al igual que pasa con la detección de las líneas de la carretera, la diferencia entre estos dos valores no es muy grande, pero aun así, con el nuevo Dataset se ha conseguido mejorar en los resultados finales. Esta diferencia que se ve se podría incrementar haciendo un entrenamiento más largo o utilizando

un Dataset de imágenes más grande, ya que con más tiempo de entrenamiento el modelo iría mejorando, aunque fuese muy poco a poco su precisión final en cada apartado.

Otro punto también importante para revisar es el tiempo final que genera cada entrenamiento para la inferencia de imágenes, en este último caso como se ve que el tiempo de inferencia para el entrenamiento con el Dataset original es de **0.0071 s/frame** frente a **0.0073 s/frame** que se consigue con el nuevo Dataset. Viendo estos valores, se tiene que el tiempo es prácticamente el mismo, ya que sólo se diferencian en **2 diezmilésimas** de segundo, por lo que, aunque el primer entrenamiento haya conseguido generar inferencias más rápidas esta diferencia de tiempo es totalmente irrelevante. Damos como conclusión que ambos entrenamientos consiguen tiempos de inferencias realmente rápidos y eficaces.

Por último, el entrenamiento muestra la información de dónde se almacenará, además que guardará el estado final que se ha conseguido del entrenamiento y que se utilizará para realizar la inferencia de imágenes y vídeos en el futuro si se configura el uso de éste.

4.4 Inferencia recorrido real

Como prueba de concepto final, se mostrará a continuación una inferencia de recorridos del vehículo del que dispone Robesafe, y con el que se trabaja a diario para temas de conducción autónoma, y con esto se podrá ver como funciona el modelo en un entorno real y comprobar que es totalmente funcional.

Se ha realizado la inferencia sobre dos recorridos completos, a continuación, se muestran algunas secuencias de los mismos:

Figura 4.6: Secuencia de Imágenes Representativa.

A continuación, se adjuntan enlaces a los dos vídeos de inferencias completas que se han realizado:

- Recorrido 1: https://youtu.be/Ar0cPIgQP40

- Recorrido 2: https://youtu.be/XW4sYgEgKTk

Como se puede ver en los vídeos, la zona conducible se detecta en el mayor número de los casos, aunque es cierto que, en algunos casos, también se detecta como zona de carretera conducible el pavimento de la acera, ya que resulta de un color y textura parecido al de la carretera y que las líneas de la carretera, por estar tan desgastadas, apenas son detectadas.

4.5 Conclusiones del capítulo

Como se ha visto a lo largo de esta sección se ha estado comparando los resultados de un entrenamiento que se ha realizado con las imágenes con las que trabaja la YOLOP por defecto, frente a las nuevas imágenes sintéticas que se ha conseguido generar gracias al simulador CARLA, y a los scripts que se han creado para este proyecto. Teniendo esto en cuenta, se ve como gracias al nuevo Dataset, se ha conseguido mejorar el rendimiento en la precisión de la detección de la zona conducible, las líneas de la carretera y también a la hora de detectar vehículos durante la circulación.

Por la tanto, se puede ver una mejora global en el modelo y se puede afirmar que con un entrenamiento de este estilo se puede ir mejorando el modelo inicial poco a poco. Además, si se consiguiese una cantidad mayor de imágenes sintéticas y se entrenara el modelo durante mucho más tiempo se conseguiría continuar mejorando el modelo cada vez más.

Además, se ha realizado una inferencia con el modelo entrenado sobre recorridos reales del vehículo automático disponible en el grupo de investigación RobeSafe.

Capítulo 5

Conclusiones y Líneas Futuras

5.1 Conclusiones

Como se ha estado viendo a lo largo de todo este trabajo, se ha conseguido demostrar que el modelo YOLOP funciona correctamente y si se quiere, se puede utilizar para realizar inferencias en tiempo real, que es de las cosas más importante que se necesitan a día de hoy en el mundo de la conducción autónoma.

Además, se ha conseguido generar un Dataset para mejorar la precisión del modelo, y así que pueda llegar a ser funcional cuando se trabaje con simuladores. Gracias a esto, el modelo puede ser utilizado para realizar pruebas o simular escenarios con mayor precisión sin necesidad de que se tenga que hacer pruebas en entornos reales.

Por último, se ha creado una interfaz web, para que cualquier persona que esté interesada en el modelo pruebe su funcionamiento sin estar preocupándose de librerías y/o software de terceros, además la Web cuenta con una interfaz sencilla, lo que facilita su usabilidad y es muy práctica ya que muestra los resultados obtenidos de una sola vez.

Con esto ya sólo quedaría por ver como se puede seguir trabajando con el modelo, y qué líneas futuras se pueden tomar con éste. Para ello, se planteará poder utilizar el modelo instanciándolo ya sea dentro de una Raspberry Pi o de una Jetson TX2, para conseguir que este sea portable y que se pueda montar en cualquier vehículo del que se disponga.

5.2 Líneas futuras

Como se ha terminado de mencionar en el apartado anterior, una idea de línea futura en la que se puede trabajar es montar el modelo en algún dispositivo hardware, para conseguir que sea portable y que se pueda utilizar en cualquier momento. Para ello, se ha pensado en una Raspberry Pi o una Jetson TX2, de los cuales se dispone en el grupo de investigación Robesafe. Para esta idea se podría basar en los archivos que ofrece la propia YOLOP, ya que existe un apartado dedicado al despliegue del modelo en un sistema real.

Pero antes, hay que mencionar que existe una tecnología de NVIDIA que ayuda con este tipo de despliegues, la cual se denomina TensorRT (https://developer.nvidia.com/tensorrt) [TensorRT(2022)], la cual es un SDK para la inferencia de aprendizaje profundo de alto rendimiento, que incluye un optimizador de inferencia de aprendizaje profundo y tiempo de ejecución que ofrece baja latencia y alto rendimiento

para aplicaciones de inferencia. Por lo que este software ayudará a la hora de implantar el modelo a que funcione de manera mucho más óptima.

Para poder hacer uso de TensorRT, se le ha de pasar unos archivos binarios que se deben generar del modelo, que de otra forma se pueden definir como hacer una exportación del modelo, generando así un archivo binario con extensión '.wts' que contendrá toda la información de este y con el que TensorRT puede trabajar.

Para generar este archivo binario, se utilizará el archivo 'toolkits/deploy/gen_wts.py', el cual de por sí ya viene integrado con el modelo.

Con todo esto ya se podría empezar a instalar el modelo dentro de la Jetson TX2 o de la Raspberry Pi, ya que estas cuentan con una arquitectura compatible para poder hacer la instalación en ellas.

Por último, se podría trabajar en ampliar la base de datos de entrenamiento con imágenes reales del vehículo autónomo de RobeSafe, para que se aprendiera el modelo el tipo de carreteras del campus por donde suele circular, eso sí, evitando en todo momento que el porcentaje de imágenes del campus fuera tan representativo que se pudiera llegar a producir un problema de overfitting.

Bibliografía

[BGR(2022)] BGR 2022 BGR: ***BGR***. 2022. – URL https://noemioocc.github.io/posts/Cambio-de-espacio-de-color-openCV-python/. – Zugriffsdatum: 2022-04-15

[Carla(2022)] Carla 2022 CARLA: ***Carla***. 2022. – URL https://carla.org/. – Zugriffsdatum: 2022-04-15

[CNN(2022)] CNN 2022 CNN: ***CNN***. 2022. – URL https://doi.org/10.48550/arXiv.1511.08458. – Zugriffsdatum: 2022-04-15

[CUDA(2022)] CUDA 2022 CUDA: ***CUDA***. 2022. – URL https://developer.nvidia.com/cuda-downloads. – Zugriffsdatum: 2022-04-15

[Docker(2022)] Docker 2022 DOCKER: ***Docker***. 2022. – URL https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04-es. – Zugriffsdatum: 2022-04-15

[Dong Wu(2021)] Dong Wu 2021 DONG WU, Weitian Zhang Xinggang Wang Xiang Bai Wenqing Cheng Wenyu L.: **YOLOP: You Only Look Once for Panoptic Driving Perception**. In: *ARXIV* (2021), Agust, Nr. 1, S. 826–836

[Gavilanes(2022a)] Gavilanes 2022a GAVILANES, Jonathan M.: ***Script para generar los componentes del Dataset.*** 2022. – URL https://universidaddealcala-my.sharepoint.com/:u:/g/personal/jonathan_moncada_edu_uah_es/Eb4RZQfMnCJGq4HDveO-4ScBpQQxtG5TYzA8bmYmLBdKLg?e=r0JlWm. – Zugriffsdatum: 2022-04-15

[Gavilanes(2022b)] Gavilanes 2022b GAVILANES, Jonathan Rober M.: ***Script para extraer imagenes de CARLA.*** 2022. – URL https://universidaddealcala-my.sharepoint.com/:u:/g/personal/jonathan_moncada_edu_uah_es/Efbx0BBhstRFm9Me826ZGcQBbHBXUNLVrE85ib_tGAj_zw?e=lXD5oG. – Zugriffsdatum: 2022-04-15

[Gradio(2022)] Gradio 2022 GRADIO: ***Gradio***. 2022. – URL https://gradio.app/blocks_and_event_listeners/. – Zugriffsdatum: 2022-04-15

[Huggingface(2022)] Huggingface 2022 HUGGINGFACE: ***Huggingface***. 2022. – URL https://huggingface.co/. – Zugriffsdatum: 2022-04-15

[NumPy(2022)] NumPy 2022 NUMPY: ***NumPy***. 2022. – URL https://numpy.org/. – Zugriffsdatum: 2022-04-15

[Nvidia(2022)] Nvidia 2022 NVIDIA: ***Nvidia***. 2022. – URL https://la.nvidia.com/Download/index.aspx?lang=la. – Zugriffsdatum: 2022-04-15

[ONNX(2022)] ONNX 2022 ONNX: ***ONNX***. 2022. – URL https://docs.microsoft.com/es-es/azure/machine-learning/concept-onnx. – Zugriffsdatum: 2022-04-15

[OpenCV(2022)] OpenCV 2022 OpenCV: ***OpenCV***. 2022. – URL https://opencv.org/. – Zugriffsdatum: 2022-04-15

[Palette(2022)] Palette 2022 Palette, Carla City S.: ***Docuemtacion Colores de la Sementacion***. 2022. – URL https://github.com/carla-simulator/carla/blob/master/LibCarla/source/carla/image/CityScapesPalette.h/. – Zugriffsdatum: 2022-04-15

[PSPNet(2022)] PSPNet 2022 PSPNet: ***PSPNet***. 2022. – URL https://paperswithcode.com/method/pspnet. – Zugriffsdatum: 2022-04-15

[Python(2022)] Python 2022 Python: ***python***. 2022. – URL https://www.python.org/doc/. – Zugriffsdatum: 2022-04-15

[Pytorch(2022)] Pytorch 2022 Pytorch: ***Pytorch***. 2022. – URL https://pytorch.org/get-started/locally/. – Zugriffsdatum: 2022-04-15

[R-CNN(2022)] R-CNN 2022 R-CNN: ***R-CNN***. 2022. – URL https://www.mathworks.com/help/vision/ug/getting-started-with-r-cnn-fast-r-cnn-and-faster-r-cnn.html. – Zugriffsdatum: 2022-04-15

[RGB(2022)] RGB 2022 RGB: ***RGB***. 2022. – URL https://es.wikipedia.org/wiki/RGB. – Zugriffsdatum: 2022-04-15

[Robesafe(2022)] Robesafe 2022 Robesafe: ***RobeSafe***. 2022. – URL http://www.robesafe.es/index.php/en/. – Zugriffsdatum: 2022-04-15

[Robotics und eSafety(2022)] Robotics und eSafety 2022 Robotics ; eSafety: ***Tech4Age***. 2022. – URL https://www.robesafe.uah.es/index.php/en/proyects?view=project&task=show&id=130. – Zugriffsdatum: 2022-04-15

[SAD-ENet(2022)] SAD-ENet 2022 SAD-ENet: ***SAD-ENet***. 2022. – URL https://github.com/InhwanBae/ENet-SAD_Pytorch. – Zugriffsdatum: 2022-04-15

[SCNN(2022)] SCNN 2022 SCNN: ***SCNN***. 2022. – URL https://programmerclick.com/article/5441712208/. – Zugriffsdatum: 2022-04-15

[Simulator(2022)] Simulator 2022 Simulator, Carla: ***Docuemtacion Sensores***. 2022. – URL https://carla.readthedocs.io/en/latest/core_sensors/. – Zugriffsdatum: 2022-04-15

[TensorRT(2022)] TensorRT 2022 TensorRT: ***TensorRT***. 2022. – URL https://developer.nvidia.com/tensorrt. – Zugriffsdatum: 2022-04-15

[UNet(2022)] UNet 2022 UNet: ***UNet***. 2022. – URL https://towardsdatascience.com/unet-line-by-line-explanation-9b191c76baf5. – Zugriffsdatum: 2022-04-15

[University(2019)] University 2019 University, Masachusets: ***BDD100K***. 2019. – URL https://bdd-data.berkeley.edu/. – Zugriffsdatum: 2022-04-15

[YOLOP(2019)] YOLOP 2019 YOLOP: ***YOLOP***. 2019. – URL https://github.com/hustvl/YOLOP. – Zugriffsdatum: 2022-04-15

[YOLOP(2022)] YOLOP 2022 YOLOP: ***requirementst***. 2022. – URL https://huggingface.co/spaces/jonathanmg96/TFG-YOLOP/blob/main/requirements.txt. – Zugriffsdatum: 2022-04-15

[YOLOV(2022)] YOLOV 2022 YOLOV: ***YOLOV***. 2022. – URL https://docs.ultralytics.com/. – Zugriffsdatum: 2022-04-15

Apéndice A

Requerimientos

A.1 Software

En esta sección comentaremos todos los requisitos softwares que se necesitan, tanto para instalar como para ejecutar nuestro modelo YOLOP.

Como punto de inicio del trabajo, el modelo YOLOP, se puede conseguir a través del siguiente enlace:

- https://github.com/hustvl/YOLOP [YOLOP(2019)]

Además, para poder tener en cuenta como funciona el modelo YOLOP y todas las utilidades que se han podido encontrar dentro de él, se puede revisar el siguiente paper(el cual explicaremos más en detalle en la sección estudio del estado del arte 2.2.3):

- https://arxiv.org/abs/2108.11250[Dong Wu(2021)]

Igualmente, será necesario tener instaladas una serie de utilidades y aplicaciones:

- Windows/Linux: se utiliza como base estos sistemas operativos ya que es donde se desarrollo la librería, y además se trata de sistemas livianos y que nos facilitan poder instalar de una manera rápida todas las librerías y dependencias que se necesitarán para este proyecto.
- Librerías:
  - El código de la YOLOP[YOLOP(2019)] depende en gran medida de python versión 3.7, PyTorch 1.7+ and torchvision 0.8+
  - Además de una serie de librerías extra:
    - scipy
    - tqdm
    - yacs
    - Cython
    - matplotlib>=3.2.2
    - numpy>=1.18.5
    - opencv-python>=4.1.2
    - Pillow

    - PyYAML>=5.3
    - tensorboardX
    - seaborn
    - prefetch_generator
    - imageio
    - scikit-learn
    - torch
    - onnxruntime
    - torchvision

- Además se ha de tener instalados los drivers de nuestra tarjeta **Nvidia** [Nvidia(2022)].
- Tener el sistema CUDA [CUDA(2022)] instalado y habilitado para la librería Pytorch.
- Y se utiliza también **Docker** [Docker(2022)], para poder trabajar con algunas herramientas más fácilmente.

A.2 Hardware

En cuanto a los requisitos Hardware no se tienen unos requisitos exactos, ya que el modelo no necesita de muchos recursos para poder funcionar. Aun así, se mostrará a continuación los elementos con los que contamos para la realización del proyecto, pero siempre que se pueda se debería utilizar el mejor hardware disponible ya que así la YOLOP [YOLOP(2019)] funcionará de manera más rápida:

- Procesador: Intel(R) Core(TM) i5-9600KF CPU @ 3.70GHz
- Memoria RAM: 16GB
- Tarjeta Gráfica: NVIDIA GeForce RTX 2070 SUPER
- Placa Base: MSI MAG B560M Mortar
- Fuente de Alimentación: EVGA 700 GD, 80+ GOLD 700W
- Almacenamiento: 2 x SSD Crucial P1 1 TB 3D NAND NVMe PCIe M.2

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá