Escola Tècnica Superior d'Enginyeria ETSE-UV

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL

VNIVERSITAT DE VALÈNCIA

TRABAJO DE FIN DE GRADO

RECONOCIMIENTO DE SEÑALES DE TRÁFICO MEDIANTE VISIÓN ARTIFICIAL Y REDES NEURONALES

AUTORA:
LAURA GARRIDO REY

TUTORES:
VICENT GIRBÉS JUAN
VALERO LAPARRA PÉREZ-MUELAS

JUNIO, 2021

ÍNDICE

TABLA DE ILUSTRACIONES

ÍNDICE DE TABLAS

1. RESUMEN

En la actualidad, se puede observar cómo la Inteligencia Artificial se ha ido implantando de forma exponencial en nuestras vidas, ya que se encuentra presente en las cosas más mundanas, tales como los smartphones, los asistentes de voz, el contenido en las redes sociales, etc. Sin embargo, cuando se piensa en Inteligencia Artificial se restringe, mayoritariamente, a los robots humanoides y a los coches autónomos, tal vez porque es lo que se ha ido mostrando a través de la ciencia ficción.

La conducción autónoma será un hito en este campo el día que llegue a implantarse de forma completa, pero mientras tanto, se seguirá investigando y mejorando hasta que se consiga alcanzar la perfección, ya que este será un gran avance para la salud vial. Es por ello por lo que en este trabajo se pretenderá implementar una de las detecciones más importantes en la conducción autónoma como es el reconocimiento de señales de tráfico. Para ello se aplicará el Deep Learning junto con redes neuronales artificiales ya entrenadas con el objetivo de poder realizar dicha detección con éxito.

Así pues, de manera más específica, en el presente trabajo se realizará un estudio sobre cuál es el mejor marco de trabajo, de entre los existentes, para llevar a cabo la detección de objetos, teniendo en cuenta su exactitud y su tiempo de ejecución. Además, se determinará de entre todos los datasets presentes, cuál es el que mejor reconocimiento y detección de señales de tráfico lleva a cabo, permitiendo así no tener que realizar el entrenamiento de la red neuronal convolucional.

Palabras clave:

- Inteligencia Artificial
- Aprendizaje Automático
- Aprendizaje Profundo
- Redes neuronales convolucionales
- Procesamiento de imágenes

2. RESUM

En l'actualitat, es pot observar com la Intel·ligència Artificial s'ha estat establint de forma exponencial a les nostres vides, ja que es troba present en les coses més mundanes com ara els smartphones, els assitents de veu, el contingut a les xarxes socials, etc. No obstant això, quan es pensa en Intel·ligència Artificial es restringeix, majoritàriament, als robots humanoides i als cotxes autònoms, tal vegada perquè és el que s'ha estat mostrant a través de la ciència ficció.

La conducció autònoma serà una fita en este camp el dia que arribe a implantar-se de forma completa, però mentres tant, es seguirà investigant i millorant fins que es puga aconseguir la perfecció, ja que este serà un gran avanç per a la salut vial. És per això pel que en aquest treball es pretendrà implementar una de les deteccions més importants en la conducció autònoma com és el reconeixement dels senyals de tràfic. Per a això s'aplicarà el Deep Learning junt amb xarxes neuronals artificials ja entrenades amb l'objectiu de poder realizar aquesta detecció amb èxit.

Així doncs, de manera més específica, en el present treball es realitzarà un estudi sobre quin és el millor marc de treball, d'entre els existents, per dur a terme la detecció d'objectes, tenint en compte la seua exactitut i el seu temps d'ejecució. A més, es determinarà d'entre tots els datasets presents, quin és el que millor reconeixement i detecció de senayls de tràfic du a terme, permitint així no haver de realitzar l'entrenament de la xarxa neuronal convolucional.

Paraules clau:

- Intel·ligència Artificial
- Aprenentatge Automàtic
- Aprenentatge Profund
- Xarxes neuronals convolucionals
- Processament d'imatges

3. ABSTRACT

Nowadays, we can observe how the Artificial Intelligence has been introduced in our lives in an exponential way, considering that we can find it in the most mundane things such as the smartphones, the voice assistants, the contents implied in the social network, etc. However, when we think about the Artificial Intelligence, our thoughts are mostly restricted to humanoid robots and autonomous vehicles, perhaps because this part of the Artificial Intelligence has been shown to us through the science fiction since it has been instituted.

The autonomous driving will be a landmark in this field when it reaches completely its settlement, but in the meantime, people will continue investigating and improving until the perfection has been reached, because it will be a huge progress in the traffic health. Therefore, this project will intend to implement one of the most important detections in the autonomous driving field which is the traffic signs recognition. For that reason, there will be an implementation of the Deep Learning method with the artificial neural networks which have already been trained, with the aim of conducting that detection successfully.

So, in a more specific way, in this project there will be performed a study about which framework is the best, among the current ones, for accomplishing the object detection, keeping in mind its accuracy and the time that it needs for running. Furthermore, it will be determined, among all the current datasets, which one accomplishes the best traffic signs recognition and detection, allowing in that way not having to train the convolutional neural network.

Key words:

- Artificial Intelligence
- Machine Learning
- Deep Learning
- Convolutional neural networks
- Image processing

4. INTRODUCCIÓN

4.1. OBJETIVOS

El principal objetivo de este trabajo es poder realizar el reconocimiento y la detección de las señales de tráfico mediante redes neuronales convolucionales ya entrenadas, permitiendo así su fácil implementación a través de cualquier framework para el Deep Learning.

Es por ello, por lo que se llevará a cabo una búsqueda de marcos de trabajo que estén enfocados a la detección de objetos junto con datasets que contengan archivos relacionados con la detección de señales de tráfico, permitiendo así no tener que entrenar la red neuronal, ya que implicaría realizar la recolección de un gran conjunto de imágenes, junto con la información necesaria, y su posterior entrenamiento.

Además, se pretende encontrar un marco de trabajo que realice las detecciones con gran exactitud junto con tiempos de respuesta mínimos, ya que la conducción autónoma, debido a la rapidez con la que los eventos suceden, presenta especial necesidad en cuanto a la velocidad de actuación para poder asegurar una conducción segura.

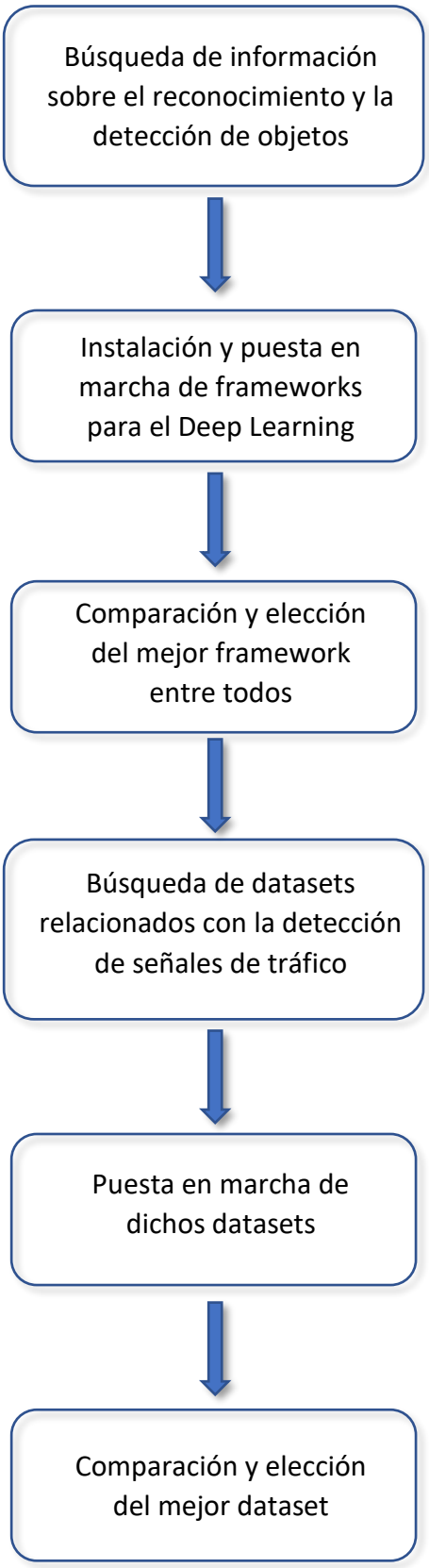
4.2. PLANIFICACIÓN DEL TRABAJO

Tal y como se ha podido observar en los objetivos de este trabajo, la planificación constará de dos partes: por un lado, la búsqueda de marcos de trabajo para llevar a cabo la detección de objetos y, por otro, la indagación de datasets para realizar la detección de señales de tráfico.

Por ello, primeramente, se investigará sobre el reconocimiento y la detección de objetos para, posteriormente, poder buscar marcos de trabajo relacionados con ello. Una vez llevado esto a cabo, se instalarán los frameworks para el Deep Learning que sean considerados adecuados para el trabajo que se quiere realizar y se pondrán en funcionamiento, realizando la posterior comparación y elección del mejor marco de trabajo para llevar a cabo la detección de objetos.

Tras haber seleccionado el framework que mejor se adapte a lo que se requiere, se procederá a la búsqueda de datasets que estén basados en la detección de señales de tráfico para luego ponerlos en funcionamiento y poder comparar cuál realiza el reconocimiento de estas señales con mayor exactitud y en el menor tiempo posible.

A continuación, se incorpora un diagrama para poder visualizar mejor la planificación de este trabajo:


Búsqueda de información sobre el reconocimiento y la detección de objetos
Instalación y puesta en marcha de frameworks para el Deep Learning
Comparación y elección del mejor framework entre todos
Búsqueda de datasets relacionados con la detección de señales de tráfico
Puesta en marcha de dichos datasets
Comparación y elección del mejor dataset

4.3. ESTADO DEL ARTE

Desde la aparición de la Inteligencia Artificial, se han ido implantando nuevas tecnologías a nuestras vidas que nos han cambiado la forma de trabajar e incluso de relacionarnos; junto a estas tecnologías se encuentra la conducción autónoma, cuyo objetivo ha sido delegar las funciones que realiza un conductor humano al propio vehículo, siendo este capaz de circular como es debido para llegar el destino requerido.

Esta conducción se ha ido desarrollando de forma más acelerada en los últimos años, pero el concepto de vehículo autónomo es más antiguo de lo que nos han hecho creer. En 1925, Francis Houdina, un ingeniero eléctrico de Nueva York, creó un automóvil dirigido a distancia y para ello, fundó su propia empresa, la Houdina Radio Control [1,2].

Su primer prototipo tan solo recorrió 19 kilómetros entre Broadway y la Quinta Avenida, finalizando su trayecto debido a un choque que tuvo contra otro vehículo. Aun así, construyó el vehículo que se encuentra en la ilustración 1, y le puso el nombre de Chandler, entre 1926 y 1930. El control remoto que se encargaba de manejar el vehículo empleaba radiofrecuencia y podía llevar a cabo el encendido de su motor, ponerlo en marcha, circular correctamente esquivando todos los obstáculos de la carretera y hacer sonar el claxon. Estos automóviles fueron vendidos por el distribuidor Achen Motor, de Milwaukee, y bajo el nombre de Phantom Auto.

502 *Radio News for November, 1925*

Radio-Controlled Automobile

By HERNDON GREEN

Radio is to control a car in transcontinental tour. The system, which is extremely simple and effective, is fully described here.

THE DIAGRAM

Ilustración 1: Vehículo Chandler (cortesía de Hope Reese)

Se podría decir que esta fue la semilla del coche autónomo, tal y como lo conocemos en la actualidad, pero realmente, cuando se habla de este, el nombre que suele aparecer es el de Norman Bel Geddes, un diseñador industrial estadounidense que destacó por sus diseños extravagantes considerados demasiado futuristas para la época. Su trabajo más reconocido fue el de un vehículo autónomo que funcionaba mediante electricidad y

dirigido por radiocontrol, y que fue presentado en 1939 en la Exposición Universal de Nueva York [3].

Este automóvil, que se muestra en la ilustración 2, se movía por un circuito eléctrico integrado en el pavimento, creando así la idea de coches eléctricos conducidos de forma autónoma guiados por autopistas inteligentes, concepto que hoy en día se sigue investigando. Las ideas de Norman Bel Geddes se encuentran en su libro Magic Motorways, publicado en 1940.

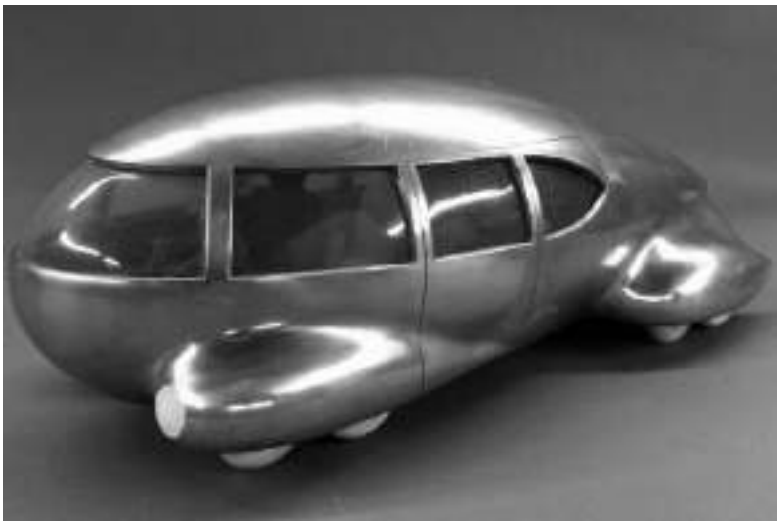


Ilustración 2: Primer vehículo autónomo de Norman Bel Geddes (cortesía de José María López)

Más adelante, entre 1940 y 1980, se siguió investigando a cerca del vehículo autónomo definitivo, pero no fue hasta los años 80 donde apareció Ernst Dickmanns, un alemán y profesor de la Bundeswehr Universitiy de Múnich, quien fue considerado un experto de la Inteligencia Artificial y lideró la construcción del primer vehículo autónomo moderno, convirtiéndolo así en el padre de este. En 1987 diseñó una furgoneta Mercedes-Benz (ilustración 3) guiada por visión y una computadora integrada, y consiguió que condujera sin conductor por una autopista alcanzando velocidades de 100km/h sin tráfico.

Ilustración 3: Furgoneta Mercedes-Benz guiada por visión y computadora (cortesía de Cristina Sánchez)

Años más tarde, en 1994, hizo algo parecido con un Mercedes 500 SEL (ilustración 4), renombrado como "VAmP", que recorrió más de 1000 kilómetros en la carretera de circunvalación que rodea París, siendo incluso capaz de adelantar a otros vehículos y alcanzar velocidades de hasta 130km/h.

Tras el éxito de estos experimentos, la Comisión Europea mostró interés y realizó una generosa inversión de 800 millones de euros que irían destinados al proyecto EUREKA Prometheus, concretamente al desarrollo del coche autónomo. Tras esto, en 1995, Ernst Dickmanns modificó un Mercedes-Benz Clase S para que llevara a cabo un trayecto entre Múnich y Copenhague, alcanzando velocidades de hasta 175km/h en las autopistas alemanas y cumpliendo con un 95% de conducción autónoma.

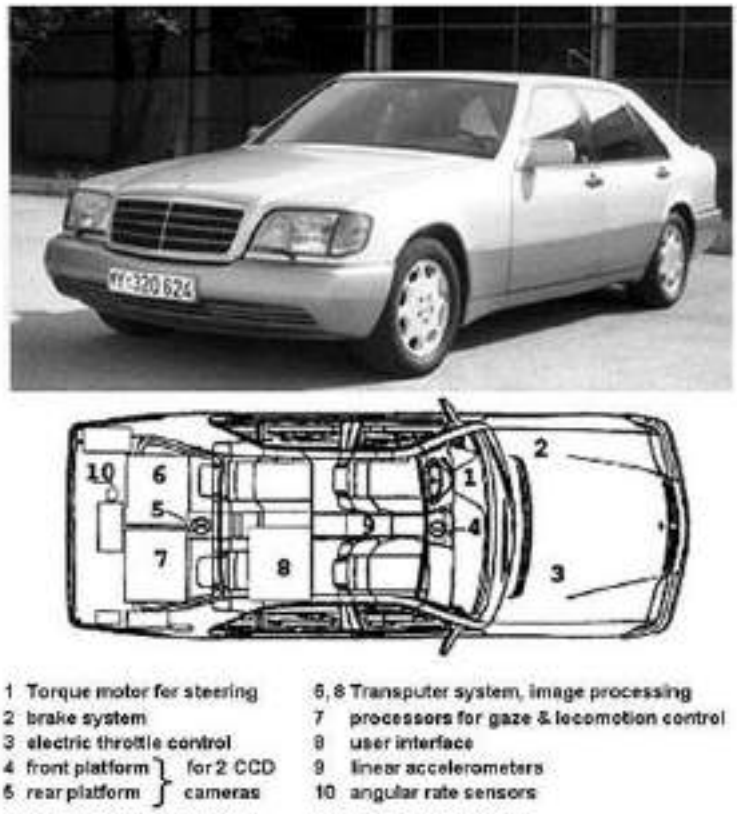


Ilustración 4: Mercedes 500 SEL (cortesía de David Rostcheck)

De los posteriores precursores del coche autónomo podríamos destacar a la agencia estadounidense ARPA, quien creó el DARPA Grand Challenge, un campeonato anual de vehículos autónomos (ilustración 6), en 2004; también estaría la empresa Google con el Google Car (ilustración 5) y Audi con un SQ5 (ilustración 7) que recorrió 5400 kilómetros en modo autónomo sin apenas intervenciones. En la actualidad podríamos destacar a la compañía Tesla como la principal precursora gracias al "Autopilot" (ilustración 8).

Ilustración 5: Google Car (cortesía de Ignacio López)

Ilustración 6: Coches del DARPA Grand Challenge (cortesía de Alex Davies)

Ilustración 7: Audi SQ5 (cortesía de Javier Álvarez)

Ilustración 8: Coche Tesla (cortesía de Nathan Bomey)

Tal y como se había destacado al principio, un coche autónomo debe ser capaz de realizar las acciones que en su caso llevaría a cabo un conductor. Para que este pueda efectuarlo, se necesita la cooperación de dos grandes partes involucradas como son la parte mecánica y la parte autónoma. Esta primera hace referencia a los propios actuadores del vehículo que se implementan fácilmente gracias a la existencia de ordenadores de abordo que asisten a diferentes tareas como pueden ser la dirección asistida o los sistemas de frenado inteligente como el ABS.

En la parte autónoma, que es la que más concierne, se encuentra un sistema que, mediante procesadores, toma las decisiones del vehículo y se encarga de comunicárselas a la parte mecánica para que las realice. Así pues, para poder llevar esto a cabo, se necesita también tener conocimiento del entorno para obtener la solución más eficaz en cada momento dependiendo de la situación en la que se encuentre el vehículo. Además, para que el coche pueda ser considerado completamente autónomo, el sistema de control debe tener un flujo constante de información que provenga tanto del interior como del entorno que lo rodea.

Por ello, los vehículos autónomos cuentan con una gran variedad de sensores que permiten al sistema de control obtener todo tipo de información sobre el entorno y poder así tomar las mejores decisiones y actuar de la forma más precisa posible. Según [4,5] algunos de estos sensores son:

- **Sensores de ultrasonidos:** los ultrasonidos son ondas sonoras de muy alta frecuencia, por lo que no llegan a ser perceptibles para nuestros oídos. Estos sensores se basan en la emisión y percepción de estas ondas permitiendo así que se puedan llegar a localizar objetos cercanos al vehículo y determinar la distancia a la que se encuentran. En la ilustración 9 se puede observar el funcionamiento de estos sensores.

  Gracias a la precisión que presentan a corto alcance, se utilizan durante los estacionamientos, ya que los movimientos lentos y las maniobras que se realizan favorecen su buen uso, y se colocan en los bajos del vehículo, tanto en la parte delantera como en la trasera, permitiéndoles detectar los objetos que se encuentran fuera del área de trabajo de los demás sistemas. En las ilustraciones 10 y 11 se puede observar dos tipos de sensores de ultrasonidos.

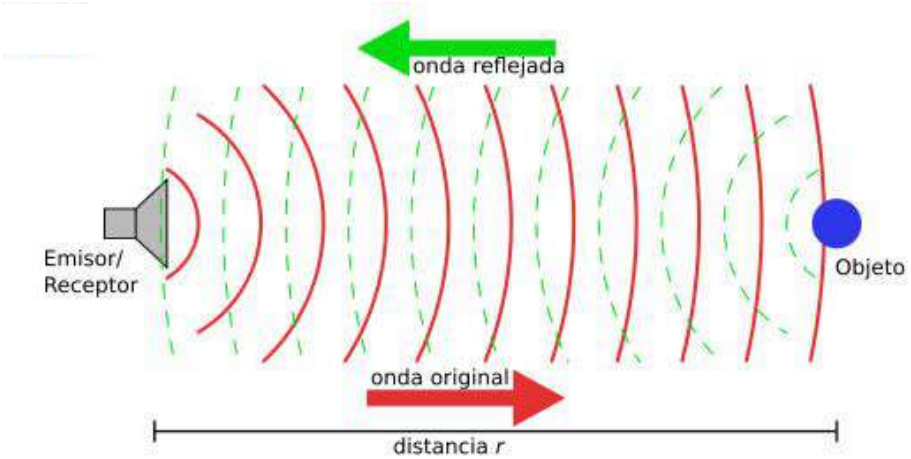


Ilustración 9: Muestra del funcionamiento del sensor de ultrasonidos (cortesía de Proyectos con Arduino)

Ilustración 10: Sensor de ultrasonidos HC-SR04 (cortesía de tiendatec)

Ilustración 11: Sensor de ultrasonidos microsonic mic+130/DD/TC (cortesía de Automation24)

- **Sensores RADAR:** al igual que los sensores de ultrasonidos, emiten y perciben ondas que les permiten conocer la localización de los obstáculos cercanos al vehículo y su velocidad. Estas ondas no son sonoras, sino que electromagnéticas, por lo que su rango de alcance es mucho mayor. Aun teniendo esta gran ventaja, poseen una parte negativa que los posicionan prioritariamente por debajo de los sensores de ultrasonidos, y es que no tienen percepción de la altura. Esta gran limitación no permite distinguir qué clase de objeto se está detectando aun pudiendo conocer que sí hay un objeto en la región del espacio.

  Es por ello por lo que este tipo de sensores se utilizan exclusivamente en situaciones de carretera donde los únicos objetos de interés a detectar sean los vehículos circulando en la misma dirección, ya que se requiere conocer su posición respecto al vehículo en cuestión.

  A continuación, en las ilustraciones 12 y 13 se muestra dos tipos de sensores de RADAR:

Ilustración 12: Sensor de RADAR T30R (cortesía de BANNER)

Ilustración 13: Sensor de RADAR Q130R con GUI (cortesía de BANNER)

- **Sensores de imagen:** estos sensores son lo que comúnmente se conocen como cámaras (tal y como se puede observar en las ilustraciones 14 y 15) y gracias a su gran versatilidad son los más utilizados en los vehículos autónomos. Proporcionan la capacidad de detectar de forma precisa tanto objetos en sí como su forma, convirtiéndolos así en unos sensores de gran utilidad en el mundo de la conducción.

  De esta forma, permiten distinguir formas en señales del mismo tamaño como en el caso de las de peligro y ceda el paso; gracias a su capacidad para captar colores, pueden distinguir entre las diferentes señalizaciones que se encuentran en las vías

pudiendo así reconocer sus distintos significados; además, pueden hacer distinciones entre señales con la misma forma, pero diferente serigrafía, como es el caso de las señales de limitación de velocidad o de prohibición de la circulación.

Aun teniendo todas estas ventajas, no permiten conocer la distancia entre los objetos y el vehículo, por lo que son sensores que apoyan y complementan al resto de ellos que se encuentran en el mismo.

Ilustración 14: Sensor de imagen VISOR Robotic (cortesía de Direct Industry)

Ilustración 15: Sensor de imagen O2V1xx series (cortesía de Direct Industry)

- **Sensores LIDAR:** tal y como indican sus siglas "Laser Imaging Detection and Ranging" se trata de unos sensores basados en láser. Al igual que los sensores de ultrasonidos y RADAR, estos emiten y perciben haces de rayos de luz láser infrarroja imperceptibles para nuestra vista ya que, al igual que los ultrasonidos, se encuentran a frecuencias de onda muy elevadas [6,7].

  Estos sensores son considerados de gran utilidad en los vehículos autónomos ya que permiten conocer a tiempo real la posición exacta de una gran cantidad de puntos y su distancia respecto al LIDAR. Esto es debido a que el sensor emite haces de rayos láser infrarrojos que rebotan sobre los objetos del entorno y son percibidos por una lente receptora infrarroja capaz de verlos, permitiendo así que el procesador del LIDAR sea capaz de obtener una nube de puntos del entorno que le proporcione a la computadora la posibilidad de procesar una imagen tridimensional en tiempo real. Gracias a la creación de esta nube de puntos se puede conocer la posición precisa del vehículo y su distancia respecto a los objetos del entorno; esto se puede observar en la ilustración 16.

  Estos sistemas suelen ser de 360 grados y se colocan en la parte superior del coche para que puedan obtener todo el rango de visión posible. Además, poseen mayor resolución de imágenes captadas que en el sensor de RADAR gracias a que su

longitud de onda es inferior, permitiendo así que puedan reflejarse en superficies más pequeñas. En la ilustración 17 se ofrecen tres tipos de sensores LIDAR.

Ilustración 16: Muestra del funcionamiento del sensor LIDAR donde se puede apreciar una imagen 3D de 360 grados generada a tiempo real con la información capturada por los cuatro sensores que contiene el coche (cortesía de Xataka)

Ilustración 17: Tres tipos de sensor LIDAR de la marca Velodyne siendo, de izquierda a derecha: HDL-64E, HDL-32E y VLP-16 (cortesía de Ibáñez)

Tras haber visto cómo se ha ido introduciendo de manera paulatina el vehículo autónomo en nuestras vidas y habiendo destacado cuáles son los sensores más importantes para poder llevar a cabo esa autonomía, se profundizará en los diferentes grados de automatización en la conducción autónoma.

En 2013, el primer organismo que llevó a cabo esta clasificación fue la NHTSA (National Highway Traffic Safety Administration) diferenciando cinco niveles de automatización focalizados en las capacidades del vehículo para poder conducirse de forma autónoma. Sin embargo, en 2014, la SAE (Society of Automotive Engineers) realizó su propia clasificación, SAE J3016, diferenciando así seis niveles de automatización centrados en el nivel de atención e intervención de la persona en la conducción.

A raíz de esta nueva clasificación, la NHTSA en 2016 también se adaptó a ella y la convierte en el estándar más utilizado en la actualidad. Cabe destacar que este estándar no es una normativa que obligue a los fabricantes a llevarla a cabo, sino una especie de guía, no regulada por ningún organismo oficial, que sirve de ayuda para poder clasificar los diferentes vehículos que salen al mercado.

Según la SAE J3016 [8], y tal y como se puede observar en la ilustración 18, los seis niveles de automatización se clasifican en:

- **Nivel 0, sin automatización:** este nivel es el punto de partida, abarcando en él todos los vehículos cuya conducción se conoce desde hace décadas. Aun teniendo integrados estos vehículos los sistemas de seguridad activa como el ESP ("Elektronisches Stabilitätsprogramm", también llamado "Control Electrónico de Estabilidad") o el ABS ("Antiblockiersystem", también conocido como "Sistema Antibloqueo de Frenos"), o las tecnologías de alerta o advertencia, se considera que no tienen ninguna automatización porque el conductor está completamente al mando haciéndose cargo de todas las responsabilidades relacionadas con su uso.

- **Nivel 1, asistencia a la conducción:** en la actualidad, la mayoría de los vehículos que hay en el mercado se encuentran en este nivel debido a que incluyen los sistemas de frenada de emergencia automática (AEB "Autonomous Emergency Braking"), el control de velocidad de crucero o la alerta de cambio de carril involuntario.

  En este caso el vehículo no es capaz de actuar por él mismo, sino que ayuda al conductor a realizar dichas acciones, siendo así la persona quien todavía controla al vehículo. A diferencia del nivel 0, el conductor sí cuenta con sistemas de asistencia para el control de la dirección o la velocidad.

- **Nivel 2, automatización parcial:** en este nivel ya se pueden apreciar pequeños matices de la autonomía debido a la posible combinación del control de velocidad adaptativo y el asistente para el cambio del carril, junto a sistemas como el detector de ángulo muerto o el de frenada de emergencia.

  Además, aparece el sistema de aparcamiento automático donde el vehículo toma el control completo para llevar a cabo el estacionamiento. Así y todo, el conductor no puede despreocuparse y debe seguir manejando el vehículo y prestando atención a su entorno en todo momento. En este nivel podríamos decir que se encuentran los coches semiautónomos.

- **Nivel 3, automatización condicional:** en el nivel anterior el conductor todavía era necesario para realizar la conducción de forma completa, en cambio, en este nivel, el vehículo se puede encargar de realizar algunas tareas de forma autónoma tales como los cambios de carril, mantenerse dentro de las líneas establecidas en la calzada y frenar en situaciones donde se podría colisionar con otro vehículo.
  Sin embargo, el conductor debe retomar el control del vehículo cuando aparezcan otras situaciones, convirtiéndolo así en un elemento todavía necesario para una conducción segura.

- **Nivel 4, automatización alta:** a diferencia de los anteriores niveles, en este ya se habla de una conducción prácticamente autónoma. El conductor no llega a ser considerado completamente prescindible, ya que en eventualidades muy concretas podría llegar a ser necesario que intervenga, pero el vehículo cuenta con un sistema de detección y respuesta ante objetos y situaciones del entorno que lo convierten en prácticamente autónomo.

- **Nivel 5, automatización completa:** en este último nivel, como bien cabe suponer, ya no es necesaria la intervención del conductor. El vehículo es capaz de controlar todos los sistemas y puede adaptarse a la circulación gracias a su gran conocimiento del entorno y su capacidad de tomar decisiones.

  En ningún momento es necesario el control ni la presencia del conductor, pero este sí tendrá la posibilidad de retomar el control manual siempre que lo desee. Dentro de los vehículos prototipo con este nivel de automatización destacan aquellos que no presentan ni volante ni pedales.

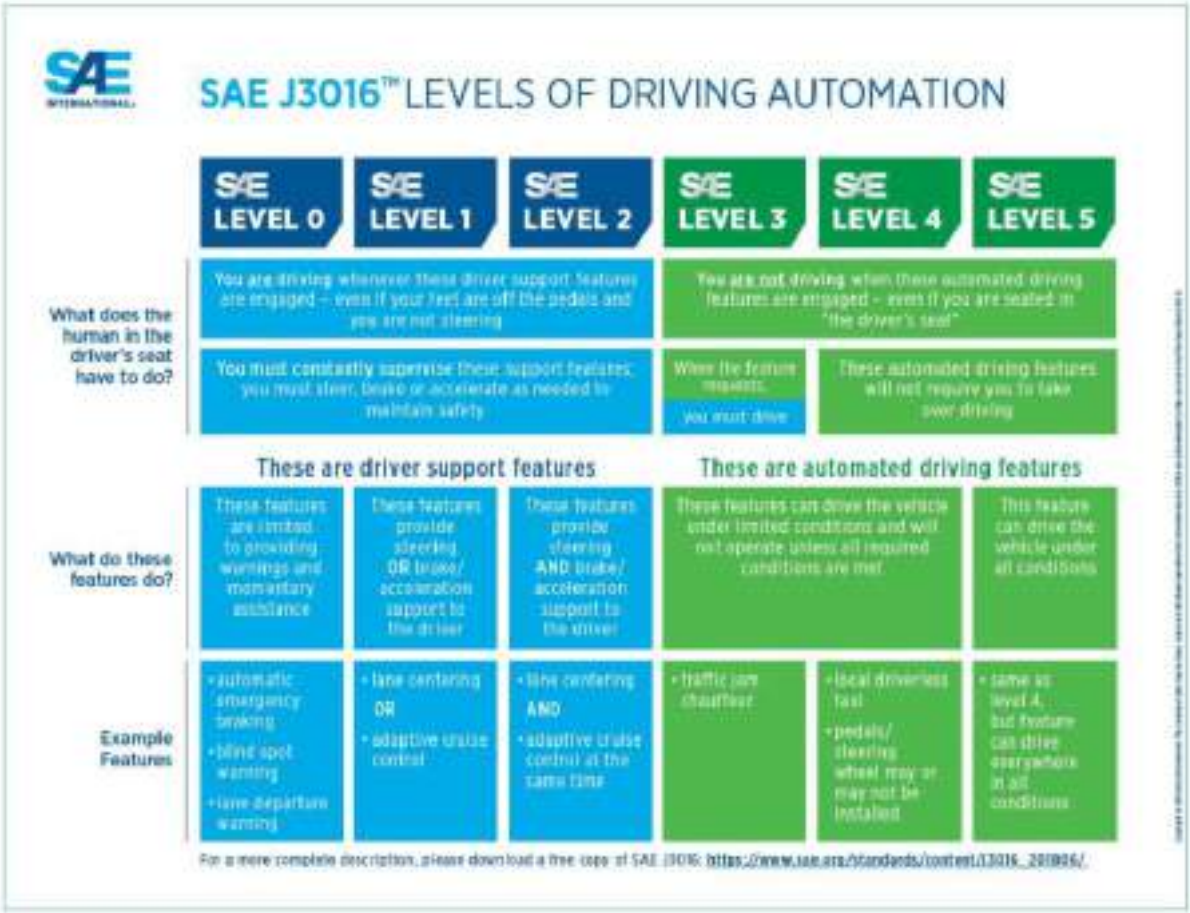


Ilustración 18: Niveles de la conducción automatizada (cortesía de SAE International)

En el futuro se espera haber alcanzado el nivel cinco, teniendo así una conducción completamente autónoma. De esta forma, se evitarían muchos de los accidentes de tráfico que suceden en la actualidad debido a factores humanos. También, conseguiría una sociedad más inclusiva porque permitiría que personas con limitaciones físicas pudieran ser aptos para la conducción; y, además, resultaría mucho más atractivo el hecho de realizar desplazamientos largos porque serían mucho más cómodos y seguros.

5. MARCO TEÓRICO

5.1. INTELIGENCIA ARTIFICIAL (IA)

El objetivo de la inteligencia artificial es conseguir que las máquinas adquieran unas capacidades que son exclusivas de la inteligencia humana, tales como el razonamiento, el aprendizaje, la creatividad y la capacidad de planear. El grupo de tecnologías que permiten que esto suceda son capaces de crear y aplicar algoritmos llevados a cabo en un entorno dinámico de computación [9].

La IA permite que estas máquinas, a través de sensores, sean capaces de recibir datos de su entorno para luego procesarlos y responder a ellos. Además, pueden analizar sus acciones previas junto con los efectos producidos y actuar como es debido, permitiendo así que trabajen de forma autónoma.

Por otro lado, dentro de la IA se pueden identificar dos ramas: la que tan solo implica software, donde están incluidos los asistentes virtuales, el software de análisis de imágenes, los motores de búsqueda y los sistemas de reconocimiento de voz y rostro; y, por otro lado, la que lleva la inteligencia artificial integrada, la cual abarca los robots, los drones, los vehículos autónomos y el internet de las cosas.

Además de esta diferenciación, Stuart Russell y Peter Norvig, en su libro "Inteligencia Artificial: Un Enfoque Moderno" [10], llegaron a la conclusión de que existían cuatro tipos de inteligencia artificial:

- Sistemas que piensan como humanos: estos tratan de emular el pensamiento humano como la toma de decisión, la resolución de problemas y el aprendizaje.
- Sistemas que actúan como humanos: tal y como indica la palabra, tratan de imitar el comportamiento humano, como sería el caso de la robótica.
- Sistemas que piensan racionalmente: los cuales se adentran un poco más e intentan emular el pensamiento lógico racional del ser humano, llevando a cabo el estudio de los cálculos que hacen posible percibir, razonar y actuar.
- Sistemas que actúan racionalmente: estos están relacionados con conductas inteligentes en artefactos, ya que tratan de emular de forma racional el comportamiento humano.

En la actualidad, según [11], la IA está presente en muchas aplicaciones técnicas destinadas a cualquier ámbito de trabajo, tal y como se puede ver en:

- Las industrias: reconociendo, clasificando y etiquetando imágenes estáticas.
- El sector financiero: ayudando a mejorar estrategias comerciales.
- El sector médico: procesando datos de pacientes.
- La industria del vehículo autónomo: realizando la detección y clasificación de objetos.
- El sector del marketing: implicado en la distribución de contenido en las redes sociales.

- El sector bancario: aportando protección contra las amenazas de ciberseguridad.

Por último, cabe destacar que cuando se habla de un software de inteligencia artificial, no se refiere a un programa informático. Esto es debido a que existe una clara diferencia entre estos dos conceptos y es que, mientras un programa informático se conforma por un conjunto de órdenes, dadas en todo momento por una persona, llegando así a cubrir todas las posibles opciones a las que se enfrenta un ordenador y permitiendo que la máquina no llegue a "pensar" por si sola, la IA utiliza una serie de algoritmos que permiten que la máquina en cuestión llegue a actuar por ella misma. Para poder llevar esto a cabo, se somete este software al aprendizaje, entrenamiento y puesta en práctica de una tarea que se le haya asignado, sin que tenga que recibir órdenes en todo momento de una persona.

Este tipo de aprendizaje se lleva a cabo mediante tareas mecánicas y repetitivas o que trabajan con el lenguaje humano. Además, en función del tipo de IA o de las tareas que se vayan a realizar, existen multitud de formas de aplicar esta teoría. Es por ello por lo que, para realizar todo lo descrito anteriormente, existen unas disciplinas que se verán a continuación.

5.2. MACHINE LEARNING

El Machine Learning, o también conocido como aprendizaje automático, es una rama de la Inteligencia Artificial que permite a los ordenadores hacer predicciones, gracias a la cantidad de datos que son capaces de manejar, para poder identificar patrones. Esto lo llevan a cabo mediante algoritmos y les permite realizar tareas específicas de forma autónoma, es decir, sin necesidad de ser programados.

Es por ello por lo que la máquina no aprende por sí misma, sino que lo hace un algoritmo de su programación, ya que este puede modificarse constantemente debido al gran flujo de datos que entran en su interfaz, permitiendo que pueda predecir escenarios futuros o que pueda realizar acciones de manera automática en ciertas situaciones. Debido a esto, el aprendizaje es automático, ya que no hay ningún tipo de intervención humana.

Este aprendizaje se divide en tres tipos principales [12]:

- **Aprendizaje supervisado:** se trata de algoritmos que se entrenan asociando etiquetas a unos datos definidos y que les permiten tomar decisiones o hacer predicciones. Una vez que se les haya proporcionado a estos algoritmos la suficiente cantidad de datos etiquetados para que puedan identificarlos, se eliminan estas etiquetas y la máquina es capaz de clasificar por ella misma los datos recibidos. Requiere de la intervención humana para retroalimentarlo.
- **Aprendizaje no supervisado:** estos algoritmos, a diferencia de los anteriores, no se entrenan, por lo que las máquinas no identifican patrones en base a datos etiquetados, sino que buscan similitudes entre los datos que están buscando y los que se les ha dado como ejemplo, llevando así a cabo una agrupación. Estos algoritmos no están programados para detectar un tipo específico de datos. A

diferencia del anterior, este tipo de aprendizaje no requiere de la intervención humana para retroalimentarlo.

- **Aprendizaje por refuerzo:** se basa en el típico ensayo de prueba y error, por lo que el objetivo es que el algoritmo aprenda a partir de la propia experiencia, al igual que los seres humanos. Su entrenamiento busca que modifique su conducta mediante "recompensas" por haber resuelto correctamente la tarea que se le había asignado, sin haberlo programado específicamente para que lo realizara de una forma determinada. Esto permite que, al igual que un ser humano, la máquina sea capaz de tomar la mejor decisión ante diferentes situaciones.

5.3. DEEP LEARNING

El Deep Learning, o también llamado aprendizaje profundo, es uno de los métodos de aprendizaje de la Inteligencia Artificial y pertenece a uno de los varios subcampos del Machine Learning. Se trata de un algoritmo automático estructurado que simula el aprendizaje humano con el fin de obtener ciertos conocimientos, lo cual consigue gracias a que está formado por unas redes neuronales artificiales entrelazadas que le permiten procesar la información obtenida [13].

Además, varios de los campos en los que se encuentra este aprendizaje son en el reconocimiento de voz, el procesamiento del lenguaje natural, la visión artificial y la identificación de vehículos en los sistemas de asistencia al conductor.

Respecto a las redes neuronales artificiales, tal y como se puede deducir por su propio nombre, se trata de un modelo inspirado en el funcionamiento del cerebro humano y están formadas por un conjunto de nodos que se encuentran conectados y transmiten señales entre sí. El objetivo de este modelo es aprender, modificándose a sí mismo, para poder llegar a realizar tareas complejas que no podrían ser llevadas a cabo mediante la programación convencional basada en reglas.

El entrenamiento de las redes neuronales se conoce como backpropagation (propagación hacia atrás) y se realiza modificando los pesos de sus neuronas para conseguir extraer los resultados deseados. Esto se alcanza introduciendo datos de entrenamiento a la red y, en función del resultado que se obtenga y de cuánto haya contribuido cada neurona a dicho resultado, se modifican sus pesos según el error obtenido.

Con este método se consigue que la red aprenda para poder obtener resultados muy acertados con datos muy diferentes a los que han sido utilizados durante su entrenamiento [14,15]. Es por ello por lo que los algoritmos del Deep Learning se encuentran en diferentes capas neuronales permitiendo así que el ordenador aprenda por él mismo reconociendo patrones mediante el uso de muchas capas de procesamiento.

El sistema llevado a cabo para entrenar los algoritmos se divide en tres capas principales, tal y como se puede observar en la ilustración 19:

- **Capa de entrada (Input Layer):** esta primera capa la conforman las neuronas que se encargan de analizar la imagen que entra en el sistema, asimilando su información, desmembrándola en píxeles y asegurándose de enviar cada trozo desglosado a las diferentes neuronas de la siguiente capa.
- **Capa oculta (Hidden Layer):** en esta zona se encuentra un conjunto de capas cuyo comportamiento se asimila al de una red. Así pues, se encargan de llevar a cabo el procesamiento de la información recibida por la capa de entrada y de la realización de los cálculos intermedios necesarios.

  Su funcionamiento se basa en el procesamiento de cada píxel recibido por la capa de entrada, realizando una delimitación dentro de cada uno; luego, se combinan los bordes de cada delimitación para diseñar las formas y constituir cada uno de los objetos de la imagen.

  Finalmente, se utilizan los filtros del sistema para poder discernir de entre todos los datos cuáles son los objetos que se requieren y cuáles no. Además, cabe destacar que cuantas más neuronas haya en esta capa, más complejos serán los cálculos que se realicen.

- **Capa de salida (Output Layer):** es la capa que recibe toda la información desmembrada y estudiada, y toma la decisión o realiza alguna conclusión aportando datos a la salida.

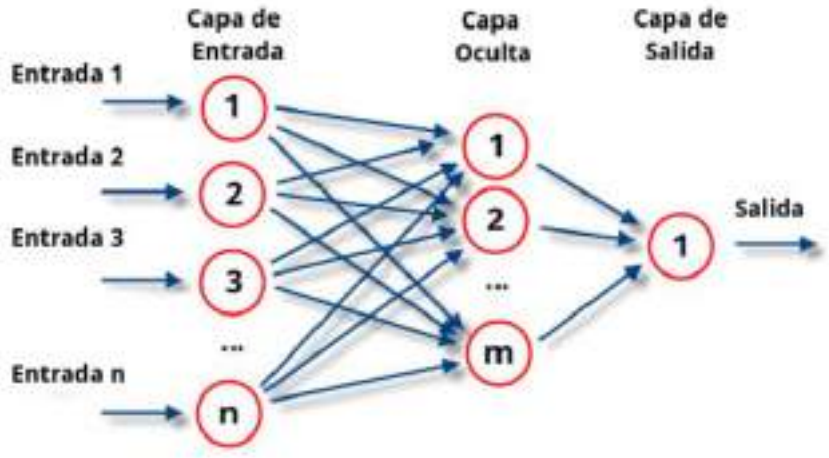


Ilustración 19: Diagrama del funcionamiento de la red neuronal (cortesía de Industria 4.0, Inteligencia Artificial)

Existen varios métodos de aprendizaje, pero todos ellos se pueden englobar en dos categorías [16]:

- **Aprendizaje Supervisado:** se conoce también como un entrenamiento "Con Profesor" y se basa en información global. En él se pueden apreciar dos vectores, el de entrada y el de salida deseada. Cuando se lleva a cabo el algoritmo de entrenamiento descrito anteriormente, se compara la salida computada por la red

con la salida deseada, y se modifican los pesos de la red con el objetivo de reducir el error cometido.

Este proceso se repite de forma iterada hasta que la diferencia entre la salida computada y la salida esperada sea ínfima. Así pues, de esta forma, con n parejas se conforma un "Conjunto de Entrenamiento".

Este aprendizaje, además, se divide en dos subcategorías:

- Aprendizaje Estructural: en él se busca la mejor afinidad posible entre los datos de entrada y de salida para cada pareja de patrones individuales. Además, la mayoría de los algoritmos dentro de este aprendizaje contienen este enfoque.
- Aprendizaje Temporal: se basa en la captura de una serie de patrones necesarios con el fin de conseguir un resultado final. En este aprendizaje, a diferencia del estructural, la respuesta de la red depende de las entradas y de las respuestas previas realizadas en esta.

- **Aprendizaje No Supervisado:** a diferencia del anterior aprendizaje, este se conoce como un aprendizaje "Sin Profesor" y tan solo utiliza información local. En él no se emplea el vector de salida esperada, sino que solo hay vectores de entrada, convirtiéndolo así en un modelo más cercano al sistema biológico.

  Su objetivo es que a entradas muy parecidas la red compute la misma salida, creando así un algoritmo cuya función sea modificar los pesos para que sus salidas sean consistentes. El proceso de entrenamiento extrae características y agrupa por clases de similitudes, permitiendo de esta forma que haya una asociación entre las entradas y las salidas dependiendo del proceso de entrenamiento que se haya llevado a cabo.

Una vez visto que gracias a la Inteligencia Artificial y sus diferentes métodos de aprendizaje los ordenadores son capaces de realizar tareas que solían ser exclusivas del ser humano, este apartado se centrará en la visión por ordenador y sus diferentes técnicas para reconocer objetos. En este campo se pueden distinguir tres tipos: la clasificación de imágenes (image classification), la detección de objetos (object detection) y la segmentación de imágenes (image segmentation).

Mientras que la clasificación de imágenes tan solo crea una etiqueta de clase por cada objeto identificado, la detección de objetos es capaz de localizar varios de ellos dentro de una imagen y la segmentación de imágenes permite comprender a nivel de píxel los elementos que se encuentran en la escena.

A continuación, se expondrán de forma más detallada las técnicas de detección de objetos y de segmentación de imágenes.

5.4. OBJECT DETECTION

La detección de objetos es una técnica de visión por ordenador cuyo objetivo es identificar y localizar diferentes objetos dentro de una imagen o vídeo. Se encarga de dibujar cajas delimitadoras (bounding box) alrededor de estos objetos permitiendo ubicar dónde se encuentran o cómo se mueven a través de la imagen. Esta técnica se puede aplicar en el conteo de personas, los vehículos autónomos, la videovigilancia y la detección de rostro y de anomalías [17,18].

Para poder detectar una imagen se pueden utilizar tanto algoritmos de Machine Learning como de Deep Learning pero, nos centraremos en estos últimos porque utilizan redes neuronales convolucionales (CNN), realizando así una detección de objetos no supervisada. Estas redes convolucionales permiten detectar una cantidad específica de objetos, además de que se deben entrenar mediante la indicación explícita de qué clase de objeto se requiere junto con su posición dentro de la imagen en cuestión, es decir, cuáles son sus medidas en el plano de dos dimensiones (X, Y).

Así pues, surgen la detección de objetos en dos pasos y en un paso. Esta primera utiliza algoritmos para identificar primeramente las "bounding box" que podrían contener objetos y luego las clasifica dependiendo de su tipo de límite; mientras que la detección de objetos en un paso surge debido a la necesidad de detectar objetos en tiempo real y por ello intenta combinar los dos pasos descritos previamente [19].

En la detección de objetos en dos pasos se encuentra:

- **R-CNN (Búsqueda selectiva):** tal y como indican sus siglas "Region Based Convolutional Neural Networks", estas redes neuronales determinan "regiones de interés" (RoI) dentro de la imagen en cuestión, es decir, regiones donde es probable que se encuentren los objetos que se desean identificar, otorgándole así el nombre de "selective search" o búsqueda selectiva.

  Una vez determinadas estas regiones, se realiza la clasificación de las imágenes sobre dichas áreas utilizando una red que ya ha sido entrenada. Varios ejemplos de estas áreas de interés podrían ser: áreas contiguas con un mismo tono de color, detección de líneas que delimiten áreas o cambios bruscos de contraste y brillo, etc. Para evitar que en diversas áreas se solape un mismo objeto, se utiliza el concepto de IoU (Intersection over Union), el cual ofrece un porcentaje de acierto del área de predicción frente a la "bounding box" real que se quería detectar donde se encuentra el objeto en cuestión (ilustración 20).

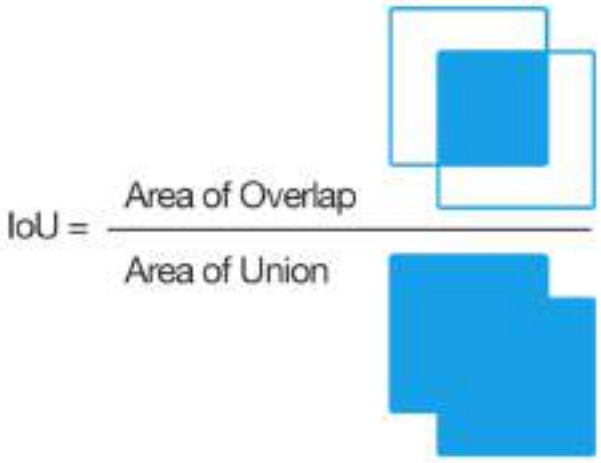


Ilustración 20: Ecuación IoU (cortesía de Adrian Rosebrock)

Además, se encuentra el "Non-Maximum Suppression (NMS)", fórmula que permite escoger la caja delimitadora que mejor se ajusta al resultado esperado, eligiendo y descartando todas las otras que se detectaron y se encuentran superpuestas al mismo objeto.

Aun así, el entrenamiento de la propia red es muy lento, por lo que, ante la necesidad de mayor rapidez, surgen dos mejoras:

- **Fast R-CNN:** este algoritmo mejora el inicial reutilizando algunos recursos como el de las características extraídas por la CNN, agilizando el entreno y la detección de imágenes. También ofrece mejoras en el IoU y en la función Loss, lo cual permite mejorar el posicionamiento de la caja delimitadora. Sin embargo, no se puede apreciar una mejoría significativa respecto a la velocidad en el entrenamiento y su posterior detección.
- **Faster R-CNN:** tal y como indica su nombre, en este algoritmo sí se puede apreciar una mejoría en cuanto a la velocidad de entrene y su respectiva detección al haber integrado un algoritmo de "region proposal" sobre la propia CNN. Además, aparece un concepto que utiliza ciertos tamaños calculados con anterioridad, llamados "anchor", que sirven para la detección de objetos específicos de la red.

Tras haber conseguido una mejoría en cuanto a la velocidad de entrenamiento y detección de los objetos requeridos, aparece una nueva forma de detección que implica la reutilización de varias técnicas descritas anteriormente, pero vistas desde otra perspectiva. En ella encontramos arquitecturas de detección en un solo paso como son YOLO, SSD y RetinaNet, entre otros.

Este apartado tan solo se centrará en YOLO debido a que se utilizará en el presente trabajo.

- **YOLO (You Only Look Once):** se trata de una red neuronal cuyas siglas significan "solo miras una vez", lo cual describe a la perfección su forma de trabajar, ya que realiza una sola pasada a la red convolucional y detecta todos los objetos para los que ha sido entrenada [20].

  Al no tener la necesidad de iterar, a diferencia de otras redes neuronales, logra velocidades de detección muy rápidas (hasta 30 FPS) incluso en ordenadores que no necesitan ser muy potentes, permitiendo de esta forma la detección sobre un vídeo en tiempo real de cientos de objetos simultáneamente y hasta su ejecución en teléfonos móviles.

  Para llevar a cabo la detección, YOLO define una cuadrícula de tamaño fijo SxS sobre la imagen, lo que se conoce como alto x ancho, y en estas celdas predice N posibles "bounding boxes" o cajas delimitadoras, calculando así el nivel de probabilidad de cada una de ellas, es decir, calculando SxSxN diferentes cajas (Ilustración 21).

  La mayoría de ellas tienen un nivel muy bajo de certidumbre, por lo que, después de obtener estas predicciones, se eliminan las cajas que se encuentren por debajo de un límite. A las cajas restantes se les aplica el paso de "Non-Maximum Suppression" que permite eliminar los posibles objetos que se hayan detectado por duplicado, posibilitando así que se escoja la caja más exacta que contenga el objeto que se quiere detectar.

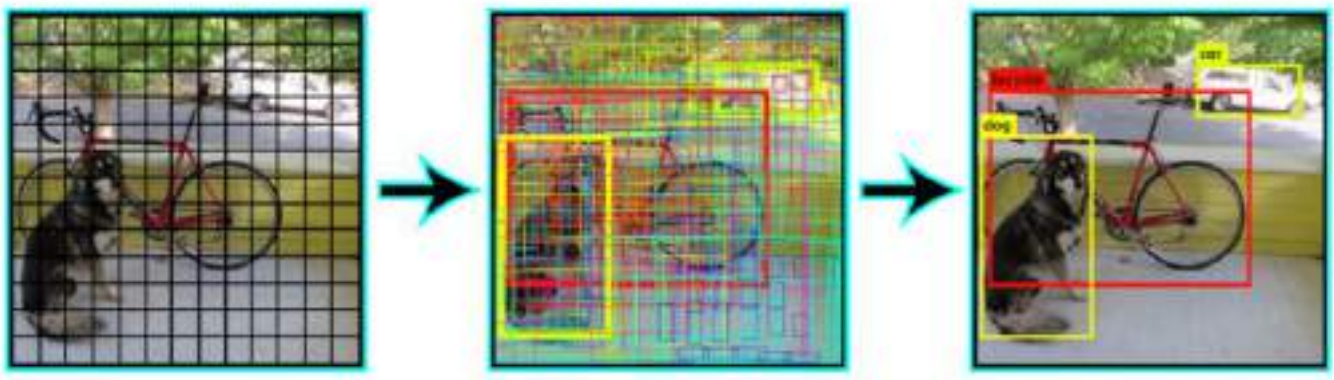


Ilustración 21: Funcionamiento YOLO (cortesía de enrique a.)

La implementación oficial de YOLO se lleva a cabo mediante Darknet, un marco de trabajo creado por las mismas personas que encuentran detrás del algoritmo de esta red neuronal, pero también se puede aplicar a través de Darkflow, un marco de trabajo basado en Tensorflow y que busca parecerse a Darknet.

A continuación, se expondrán otros marcos de trabajo de Deep Learning, y luego se hará más hincapié en los nombrados anteriormente.

5.4.1. COMPARACIÓN DE MARCOS DE TRABAJO DE DEEP LEARNING PARA LA DETECCIÓN DE OBJETOS

Existen infinidad de marcos de trabajo para la detección de objetos mediante el Deep Learning. En este apartado se expondrán varios frameworks, toolkits y APIs más destacados dentro de este amplio campo, además de un proyecto creado para facilitar el intercambio de marcos de trabajo.

1. **FRAMEWORKS**

Por lo que respecta a los frameworks, se van a detallar cinco de ellos indicando algunas de sus características más relevantes, además de que se llevará a cabo alguna comparación entre ellos para poder distinguirlos de forma más fácil.

- **TensorFlow [21]:** se trata de una gran plataforma de código abierto para construir y entrenar redes neuronales, desarrollada por investigadores e ingenieros de Google Brain Team. Al ser de código abierto proporciona facilidad de ejecución y escalabilidad. Además, este framework es el sucesor de DistBelief.

  En cuanto a sus características, está escrito mayoritariamente en C++ y en Python, aunque también lo está en Java, Go, Haskell, JavaScript, Swift y algunos más. Permite su implementación en una o más CPUs al igual que en GPUs y no hace distinciones entre equipos de escritorio, servidores o dispositivos móviles con solo una API. Para permitir la implementación en GPU, soporta el multiprocesador de kernel NVIDIA, junto con CUDA y cuDNN. Además, cabe destacar que puede ser ejecutado tanto en local como en la nube y las plataformas que lo soportan son Linux, macOS, Windows, iOS, Android y Raspberry Pi.

  Ayuda a los desarrolladores a acercarse al Machine Learning a través del código, lo que le lleva a convertirse en la herramienta más utilizada dentro del Deep Learning.

- **CAFFE [22]:** se trata de un framework con software de código abierto cuya comunidad es considerada lo suficientemente grande como para poder dar soporte en todos los sectores.

  Dentro de sus características se puede apreciar que el lenguaje de programación está basado en C++ y Python y que puede trabajar tanto con CPUs como con GPUs. También las plataformas que lo ejecutan son macOS, Windows, Linux y algunas más y puede soportar un número de multiprocesadores cuyas librerías son Intel MKL, WDNN, NVIDIA, CNN, LSTM, RCNN y modelos de redes neuronales. Además, aporta gran velocidad de procesamiento de imágenes gracias a "single nvidiak 47", el convertidor más rápido que hay en este dominio.

- **PyTorch [23]:** se trata de un framework de código abierto y es uno de los lenguajes de programación más populares. Lleva a cabo dos modos de operación: eager, utilizado para la investigación y el desarrollo, y gráfico, utilizado para la producción. Para poder realizar una transición fácil entre estos dos modos de operación utiliza

TorchScript. Además, lleva a cabo una depuración de datos sencilla y su soporte de gráfico es dinámico, es decir, el comportamiento de la red se puede ir cambiando mientras que la programación se está ejecutando.

En cuanto a sus características, proporciona un paquete de Python para funciones de alto nivel y una de sus bibliotecas más populares es NumPy. Puede llevar a cabo un paralelismo de datos gracias a la distribución de su trabajo entre múltiples CPUs o GPUs. Además, para dar soporte a la GPU utiliza CUDA, y se puede implementar en Windows, Linux y en la nube.

Presenta una mayor productividad del desarrollador, APIs simples pero potentes y tiene soporte nativo de ONNX, ya que puede exportar modelos en el formato estándar de intercambio de red neuronal abierta.

- **MXNet [24]:** se trata de un framework soportado por Python, R, C++ y Julia. Puede trabajar con GPUs gracias a que su backend está escrito en C++ y CUDA. Además, incluye la interfaz Gluton, permitiendo así a los desarrolladores de cualquier nivel de experiencia comenzar a utilizar el Deep Learning en la nube y en aplicaciones para móviles.

  Es por ello por lo que se considera un marco de interferencia y formación técnica escalable y ágil con una API concisa y fácil de usar para el Machine Learning.

- **WEKA [25]:** se considera un framework de software libre y destinado al Machine Learning y la minería de datos. Su uso es fácil gracias a su interfaz gráfica de usuario y se utiliza para la enseñanza, la investigación y aplicaciones industriales. Está escrita en Java y ofrece acceso a toolboxes como Scikit-learn, R y Deeplearning4j.

TensorFlow vs. CAFFE

TensorFlow	CAFFE
High-level API: reduce el trabajo de los desarrolladores	Utilizado para desarrolladores que trabajan en Deep Learning y en entrenamiento de modelos
Conocido como la librería más fiable para el Deep Learning	Aun siendo consistente con el procesamiento de imágenes, no llega a destacar con las redes neuronales
Utilizado por investigadores y lidia con ámbitos más amplios	Más apropiado para móviles Android y ordenadores
Soporta Python y C++	El entrenamiento de modelos debe ser hecho con C++ porque no soporta Python
Para multimedia utiliza TF dot device	Para multimedia utiliza la librería MP

Tabla 1: Comparación de características entre TensorFlow y CAFFE

TensorFlow vs. PyTorch

TensorFlow	PyTorch
Desarrollado por Google y basado en Theano (librería de Python)	Desarrollado por Facebook utilizando la librería Torch
Concepto de gráfico estático	Concepto de gráfico dinámico
Depuración: requiere la herramienta de depuración propia (TF debugger)	Depuración: proceso computacional dinámico (standard Python debugger)
Soporta a un nivel superior de funcionalidad y da un espectro amplio de posibilidades para trabajar	Tiene menos características, es más restringido en ese sentido
El gráfico entero puede ser guardado como un buffer de protocolo, incluyendo los parámetros y las operaciones	Sirve un simple API que guarda todos los pesos del modelo o alberga toda la clase
Da soporte a cualquier servidor	Da menos servicio a cualquier servidor en comparación con TensorFlow
No requiere que el usuario especifique nada porque hay valores predeterminados	Requiere que el usuario especifique lo que quiere si CUDA está disponible
Gran comunidad que da soporte	A comparación, una comunidad más pequeña

Tabla 2: Comparación de características entre TensorFlow y PyTorch

2. TOOLKITS

Dentro de esta categoría se describirán dos toolkits, uno destinado a Java y el otro a Windows y Linux.

- **Deeplearning4j [26]:** se trata del primer distribuidor de librerías de Deep Learning de código abierto escrito para Java, permitiendo así que sea compatible con cualquier lenguaje JVM (Scala, Clojure o Kotlin). Las computaciones más destacadas están escritas en C, C++ y CUDA, lo cual nos lleva a afirmar que se puede utilizar tanto en CPUs como en GPUs.

  Cabe destacar que en múltiples GPUs funciona igual que CAFFE. Además, está integrado con Hadoop y Apache Spark, destacando que las librerías son completamente de código abierto, Apache 2.0, y mantenidas por la comunidad del desarrollador y el grupo Konduit.

- **The Microsoft Cognitive Toolkit o CNTK [27]:** se trata de un toolkit de código abierto desarrollado por Microsoft Research cuya librería está basada en redes neuronales profundas (DNN). Además, es uno de los primero toolkits de Deep Learning que soporta el formato ONNX.

  En cuanto a sus características, admite C++ como soporte nativo, pero también Python y scripts de BrainScripts. Se puede implementar tanto en una única CPU o GPU como en múltiples, además de en varias máquinas o en Azure mediante

Docker. En cuanto a las plataformas que lo soportan se encuentran los sistemas operativos de Linux de 64-bit y de Windows de 64-bit.

3. API (Application Programming Interfaces)

Una API es una interfaz que permite la comunicación entre dos sistemas o plataformas distintas. En esta categoría se van a exponer dos APIs utilizadas para diferentes backends.

- **DeepDetect [28]:** se trata de una API y un servidor escrito en C++11, con una plataforma web pura que permite entrenar y manejar los modelos de Deep Learning.

  Contiene una librería remota de cliente de Python y se utiliza en CPU y en GPU. En cuanto al soporte de librerías de backend que contiene se encuentran CAFFE, CAFFE2, TensorFlow, XGBoost, Dlib y NCNN.

- **Keras [29]:** se trata de una librería de código abierto con licencia MIT que no funciona como un framework independiente sino como una interfaz de uso intuitivo (API). Contiene una interfaz diseñada expresamente para personas y secundariamente para máquinas, por lo que las acciones del usuario necesarias para los casos de uso más importantes se reducen al mínimo, permitiendo el fácil aprendizaje y una mayor productividad.

  En cuanto a sus características, esta API está escrita en Python y se puede utilizar tanto en CPU como en GPU, teniendo un excelente soporte para múltiples GPUs. Además, los modelos desarrollados son compatibles con iOS, Android, Google Cloud y Raspberry Pi, y permite acceder a varios frameworks de backend como Theano, CNTK y TensorFlow.

4. OTROS

En este apartado se va a presentar un proyecto creado para facilitar el intercambio de marcos de trabajo cuyos modelos son implementados por varios de los descritos anteriormente.

- **ONNX [30]:** se trata de un ecosistema de inteligencia artificial de código abierto cuyas siglas significan "Open Neural Network Exchange", por lo que, como se puede deducir por el acrónimo, se refiere al Intercambio Abierto de Redes Neuronales. Este intercambio lo lleva a cabo gracias a ONNX Runtime, el cual se va a desarrollar a continuación.
- **ONNX Runtime [31]:** se trata de un proyecto de código abierto diseñado para acelerar el Machine Learning a través de un amplio rango de frameworks, sistemas operativos y plataformas hardware. Como es lógico, implementa modelos de ONNX y admite todos los modelos de DNN y de Machine Learning tradicionales.

Está escrito en C++, pero también tiene las API de C, Python, C#, Java y JavaScript (Node.js). También ha mejorado el doble el rendimiento en la CPU y se integra en entornos de hardware diferentes como TensorRT en GPU de Nvidia, OpenVINO en procesadores de Intel, DirectML en Windows, etc. Funciona en plataformas como Linux, Windows y macOS, al igual que está optimizado tanto para la nube como para Edge. Además, utiliza servicios de Microsoft como Bing, Office y Azure Cognitive Services.

Además de los nombrados anteriormente, existen otros marcos de trabajo al igual que servicios disponibles para la detección de objetos, como se puede observar en estas tablas extraídas del ICCIDS 2018 (International Conference on Computational Intelligence and Data Science) [32].

- **SERVICIOS DISPONIBLES PARA LA DETECCIÓN DE OBJETOS**

Name	Service	Features	Access
Clarifai [15]	Image and Video Recognition Service	Image and video tagging, Model customization, visual similarity based image search, multi-language support, scalable processing of images and videos, Custom model (pre-trained model) for specific categories (like wedding, travel, food, face recognition, colour, Not Safe For Work model)	Client library to access API, HTTPS
Google Cloud Vision API [16]	Image Analysis Service and API	Label Detection, Explicit Content Detection, Facial Detection, Landmark Detection, Logo Detection, Image sentiment analysis, multi-language support	Integrated REST API
Microsoft Cognitive Service [17]	Computer Vision API	Recognition of face, speech and vision, detection of emotions and video, Motion tracking, speech and language understanding, image tagging and categorization, line drawing detection, Region dependent service availability, thumbnail generation from image and video	REST API
IBM Watson Vision Recognition Service [18]	Visual Recognition service for understanding image content	Image: Class description and class taxonomy, face detection, multi-language support, Image matching identification with confidence score	HTTP
Amazon Rekognition [19]	Deep learning based image recognition and analysis	Object and Scene Detection, Facial Analysis, Face Comparison, Facial Recognition, Integration with (AWS) Amazon Web Services, multi-language support	Command Line Interface, HTTP
CloudSight [20]	Image understanding API	Image description is sent as response when image is sent via REST API	REST API (REST library available in Ruby, Objective-C / Swift, Go, Python)

Tabla 3: Servicios para la detección de objetos (cortesía de ICCIDS 2018)

- **FRAMEWORKS PARA EL DEEP LEARNING**

Name	Features	Interface	Deep learning model			Multi-node parallel execution	Developer	License
			CNN	RNN	DBN/ RBM			
Caffe [4]	Speed, modular structure, plaintext schema for modelling and optimization, Data storage and communication using blob	C, C++, command line interface, Python, and MATLAB	√	√		Yes	Berkeley Vision & Learning Center	BSD License
Microsoft Cognitive Toolkit CNTK [5]	Multi-dimensional dense data handling, automatic hyperparameter tuning, batch normalization	Python, C++, C#, and command line interface	√	√		Yes	Microsoft Research	MIT License
TensorFlow [6]	Math computations using data flow graph, inception, image classification, auto-differentiation, portability	C++, Python, Java, Go	√	√	√	Yes	Google Brain team	Apache 2.0
Theano [7]	Compilation of math expressions using multi-dimensional arrays	Python	√	√	√	Yes	Université de Montréal	BSD License
Torch [8]	N-dimensional array support, automatic gradient differentiation, support for neural models and energy models	C, C++, Lua	√	√	√	Yes	R. Collobert, K. Kavukcuoglu, C. Farabet	BSD License
Chainer [9]	Define-by-Run Scheme for defining network, multi-GPU parallelization, Forward/Backward Computation, Per-batch architecture	Python	√	√		Yes	Preferred Networks Inc.	MIT License
Keras [10]	Fast prototyping, modular, minimalistic modules, extensible, arbitrary connection schemes	Python	√	√	√	Yes	F. Chollet	MIT License
Deeplearning4j [11]	Distributed deep learning framework, micro-service architecture	Python, Java, Scala	√	√	√	Yes	Skymind engineering team	Apache 2.0
Apache Singa [12]	Scalable distributed training platform	Python, Java, C++	√	√	√	Yes	Apache Incubator	Apache 2.0
MXnet [13]	Blend of symbolic programming and imperative programming, portability, auto-differentiation	Python, R, C++, Julia	√	√		Yes	Distributed (Deep) Machine Learning Community	Apache 2.0
Neon [14]	Fastest performance on various hardware & deep networks (GoogLeNet, VGG, AlexNet, Generative Adversarial Networks)	Python	√	√	√	Yes	Intel	Apache 2.0

Tabla 4: Frameworks para el Deep Learning (cortesía de ICCIDS 2018)

5.4.2. DARKNET VS. DARKFLOW

Aun habiendo una gran variedad de marcos de trabajo para poder aplicar el Deep Learning, tal y como se ha podido observar en el apartado anterior, este trabajo se centrará en dos marcos de trabajo con los que se puede implementar YOLO, que son Darknet y Darkflow.

Según [33], Darknet es un framework de red neuronal de código abierto escrito en C y CUDA, por lo que admite cálculos de CPU y GPU. El repositorio se encuentra en GitHub, es fácil de instalar, y solo se ha probado en Linux y macOS. Se puede instalar junto a dos dependencias: OpenCV y CUDA, las cuales ofrecen una variedad más amplia de tipos de imágenes compatibles y computación con GPU, respectivamente. Es opcional obtener estas dependencias, pero los resultados son mucho más satisfactorios si se cuenta con ellas. Además, Darknet muestra información al mismo tiempo que carga el archivo de configuración y va realizando todos los pesos, clasificando posteriormente la imagen que está analizando.

Tal y como se puede observar en su propio repositorio [34], Darkflow es una implementación de YOLO en TensorFlow, permitiendo que los modelos de Darknet se conviertan a este último. Además, se puede instalar tanto en Linux como en Windows y da la opción de implementar tanto la CPU como la GPU. Cabe destacar que Darkflow funciona bastante rápido tan solo con CPU, por lo que se suele escoger por ello, además de que es sencillo de instalar. Lo único que se debe tener en cuenta es que funciona con YOLOv2, mientras que Darknet funciona con cualquier versión de YOLO.

	DARKNET	DARKFLOW
Implementación tanto en CPU como en GPU	SÍ	SÍ
Dependencias necesarias	CUDA y OpenCV	NumPy, Cython y CUDA
Plataformas donde se puede instalar	Linux y macOS	Linux y Windows
Versiones de YOLO	Todas las versiones	YOLOv2
Velocidad de detección	Utilizando la GPU es mucho más rápido que tan solo con la CPU	No presenta una diferencia notable entre la CPU y la GPU, por lo que es un poco indiferente cuál de las dos utilizar

Tabla 5: Comparación de características entre Darknet y Darkflow

5.5. IMAGE SEGMENTATION

La segmentación de imágenes es una técnica de visión por ordenador que implica la conversión de una imagen en un conjunto de regiones de píxeles que se simbolizan por una máscara o una imagen etiquetada. Gracias a esta división de segmentos, puede discernir entre los más importantes, es decir, diferenciar los que ofrecen información del resto y procesarlos, evitando así el procesamiento de toda la imagen [35,36].

Tal y como se ha descrito, a grandes rasgos, este proceso consiste en dividir una imagen en múltiples segmentos, asociándole a cada píxel un tipo de objeto. A raíz de esto, se puede distinguir entre dos tipos de segmentación de imágenes:

- **Segmentación semántica (semantic segmentation):** en esta técnica todos los píxeles que agrupan un objeto del mismo tipo son marcados utilizando una sola etiqueta de clase. Como se puede observar en la ilustración 22, todos los píxeles que pertenecen a una misma clase están representados por el mismo color.

- **Segmentación de instancias (instance segmentation):** a diferencia de la otra, esta técnica ofrece sus propias etiquetas a objetos similares. Si observamos la ilustración 22, diferentes objetos que pertenecen a la misma clase son representados por colores diferentes, lo cual permite identificar la cantidad de diferentes objetos dentro de un mismo tipo.

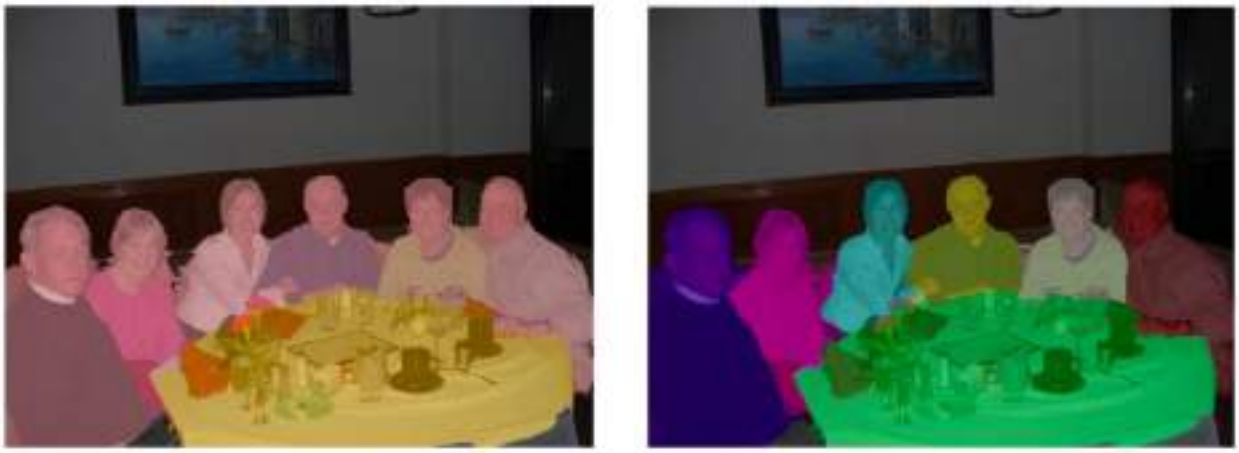


Ilustración 22: Ejemplo de segmentación semántica y segmentación de instancias (cortesía de Derrick Mwiti)

Además, la segmentación de imágenes se puede utilizar en diferentes ámbitos y para fines diferentes, tales como:

- Sistemas de reconocimiento facial.
- Realización de diagnósticos más eficientes y rápidos en la industria médica gracias a la detección de enfermedades y tumores, entre otros.
- Localización de objetos en imágenes de satélite.
- Videovigilancia.
- Conducción autónoma.

Para llevar a cabo el reconocimiento de objetos a través de esta técnica, aparecen dos tipos de enfoque basados en las propiedades de la imagen, retratados en la ilustración 23 [37]:

- **Detección de similitudes (enfoque regional):** este enfoque se basa en la detección de píxeles que presentan similitudes y se realiza teniendo en cuenta el umbral y el crecimiento, la expansión y la fusión de la región. Los algoritmos de Machine

Learning tales como la agrupación en clústeres o la clasificación utilizan este tipo de enfoque.

- **Detección de discontinuidad (enfoque de límites):** a diferencia del enfoque previo, este busca píxeles que presentan discontinuidades. Los algoritmos que se basan en este tipo son la detección de bordes, de puntos y de líneas.

Ilustración 23: Ejemplo de enfoque de límites y enfoque regional (cortesía de scikit-image)

Además, en base a estos dos enfoques y el tipo de procesamiento que se requiere, se pueden distinguir varias técnicas para la realización de segmentación de imágenes:

1. **Método de umbral**

   El objetivo de esta técnica es encontrar píxeles similares en una imagen basándose en los valores máximos que aparecen en su histograma. Esto se lleva a cabo comparando la intensidad del píxel con un valor de umbral, permitiendo así realizar una clasificación de los diferentes píxeles que contiene la imagen. La ventaja que presenta es que no requiere de un preprocesamiento complicado, pero, por el contrario, pueden aparecer los errores de umbral comunes que omiten demasiados detalles.

2. **Método basado en bordes**

   Este método se basa en la detección de discontinuidades de la imagen para poder apreciar los bordes y, por lo tanto, definir los límites del objeto. Además, reduce de forma significante el tamaño de la imagen y filtra la información menos relevante, centrándose así tan solo en sus propiedades estructurales más importantes.

   Los algoritmos que se emplean para detectar los bordes de un objeto se basan en discontinuidades en el nivel de gris, de color, de textura, de brillo, de saturación, de contraste, etc. Por ello, este método es ventajoso si se trata de imágenes con gran contraste entre objetos, pero no se recomienda si hay demasiado ruido en ellas o si hay una gran presencia de bordes.

3. Método basado en la región

Tal y como dice la palabra, este método se basa en la partición de la imagen en varias regiones homogéneas, es decir, el algoritmo que se utiliza crea segmentos que permiten dividir la imagen en diversos conjuntos de píxeles que presentan características similares. Dentro de este método se diferencian dos técnicas:

- Región de crecimiento: se trata de un método donde se escoge un píxel base arbitrario y se compara con los píxeles adyacentes; si aparece una similitud, estos píxeles se agregan al píxel base, aumentando de esta forma el tamaño de la región. Cuando se llega al punto de que esa región ya no puede aumentar más, se encuentra está saturada, se escoge otro píxel base y se realiza la misma técnica.

  Para evitar sesgar la segmentación, varias regiones van creciendo de forma simultánea llevando a cabo la técnica descrita. Además, el crecimiento de las regiones se utiliza para imágenes con mucho ruido donde detectar los bordes de los objetos es una tarea difícil.

- División y fusión de regiones: estas técnicas trabajan juntas y sus funciones son, tal y como se puede deducir, por una parte, dividir de forma iterativa una imagen en regiones con características similares y, por otra, combinar regiones adyacentes que presentan similitudes entre sí.

Este método presenta unos cálculos sencillos, una velocidad de operación rápida y una gran capacidad para detectar objetos en imágenes con alto grado de contraste. Por el contrario, consume tiempo y memoria, y no es muy preciso en imágenes donde la escala de grises no presenta una diferencia significativa.

4. Método basado en clústeres

Se trata de algoritmos no supervisados donde no se presenta un conjunto predefinido de características, clases o grupos, ayudando así a obtener información que se encuentra oculta de forma subyacente en los datos, tales como las estructuras y las agrupaciones.

Esta técnica segmenta la imagen en grupos de píxeles con características muy parecidas, permitiendo así que los datos que se juntan sean mucho más similares entre sí con respecto a los demás que se encuentran en otros grupos. Algunos de estos algoritmos más eficientes son:

- K-means y K-means mejorado: se trata de uno de los algoritmos de aprendizaje no supervisado más simple y eficiente computacionalmente. El proceso que lleva a cabo es la clasificación de una imagen a través de un cierto número de grupos que son fijos (K). El K-means mejorado puede llegar a minimizar el número de iteraciones necesarias.
- Fuzzy C-Mean (FCM) e Improved Fuzzy-C Mean (IFCM): el algoritmo FCM permite que los píxeles pertenezcan a varias clases diferenciadas con diversos

grados de pertenencia. Mientras que, tal y como indica su nombre, el algoritmo IFCM es tan solo una mejora del primero.

La ventaja de utilizar este método es que es útil para aplicaciones en tiempo real y ha sido probado y reforzado con lógica difusa. Por el contrario, puede llegar a resultar complicado determinar la función de costo para la minimización.

5. Método basado en redes neuronales artificiales

Tal y como ya se ha ido introduciendo, este método se basa en algoritmos de Deep Learning, en concreto, en algoritmos de redes neuronales artificiales. Se trata de un método donde se utiliza la Inteligencia Artificial para poder procesar e identificar componentes tan característicos como objetos, caras, texto, etc., además de que estos algoritmos se utilizan en una amplia variedad de industrias y aplicaciones.

Presentan una gran ventaja a la hora de implementarlos, ya que no tienen necesidad de seguir algoritmos complicados, además de que disponen de bibliotecas en Python y de aplicaciones prácticas. Por otro lado, el entrenamiento del modelo requiere de mucho tiempo y es bastante costoso en cuanto a recursos.

Según [38], en este método se encuentran varias estructuras para poder llevarlo a cabo:

- U-Net: se trata de una red convolucional desarrollada para segmentar imágenes biomédicas y se puede apreciar su estructura en la ilustración 24.

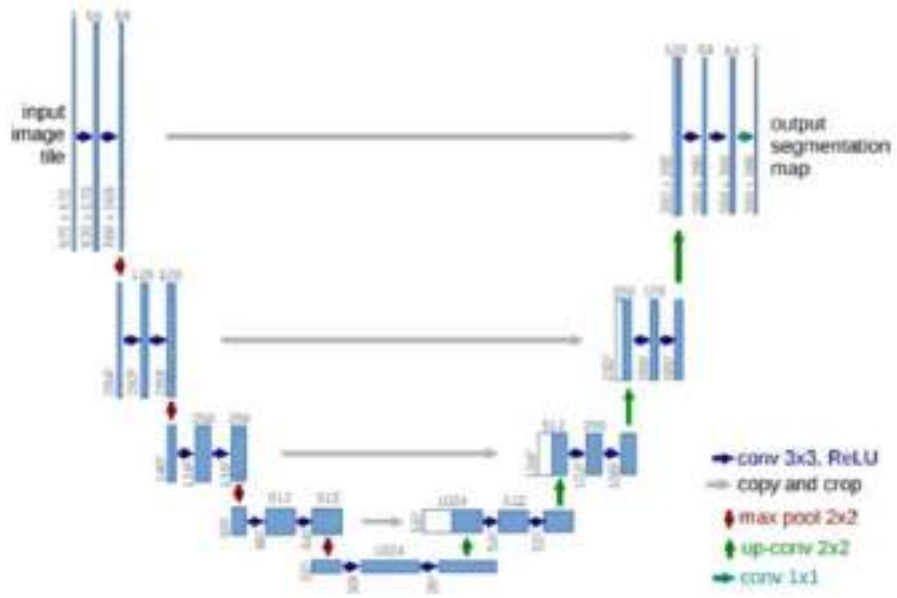


Ilustración 24: Estructura U-Net (cortesía de Olaf Ronneberger)

- FastFCN (Fast Fully-Connected Network): en este tipo se utiliza una red conectada a su núcleo mientras aplica JPU (Joint Pyramid Upsampling) para el muestreo superior. Se puede observar su estructura en la ilustración 25.

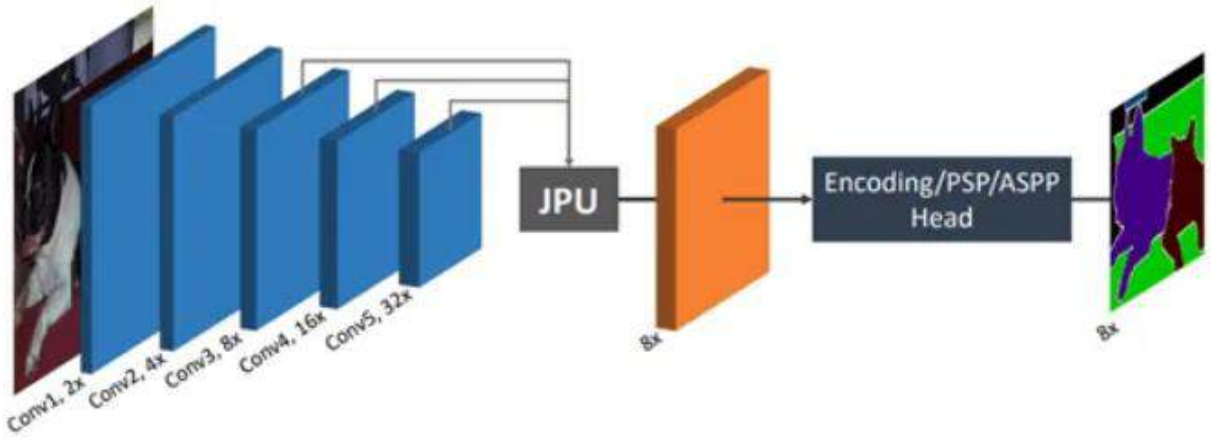


Ilustración 25: Estructura FastFCN (cortesía de Huikai Wu)

- Gated-SCNN (Gated Shape CNN): esta red tiene una arquitectura CNN de dos flujos y su estructura se puede observar en la ilustración 26.

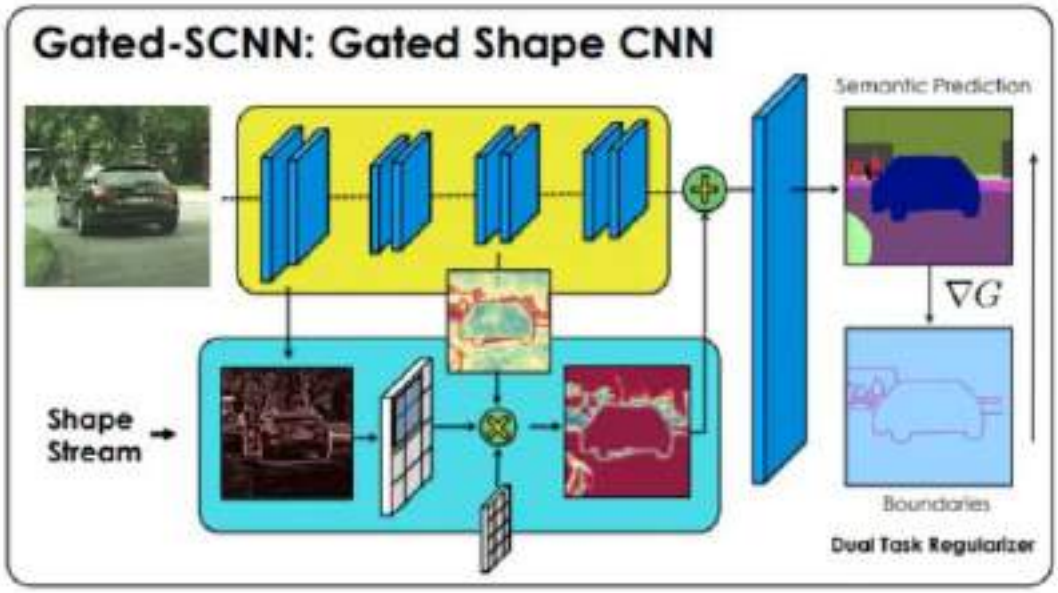


Ilustración 26: Estructura Gated-SCNN (cortesía de Towaki Takikawa)

- DeepLab: se utiliza esta red para tareas que implican una predicción densa. Se puede observar un claro ejemplo en la ilustración 27.

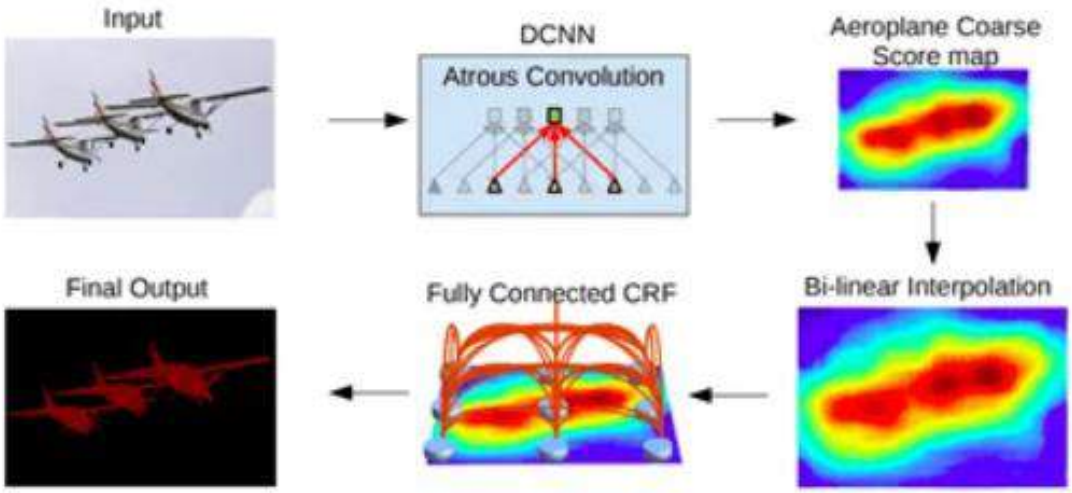


Ilustración 27: Estructura DeepLab (cortesía de Liang-Chieh Chen)

- Mask R-CNN: en esta red se clasifican y localizan los objetos mediante un "bounding box" y una segmentación semántica que clasifica los píxeles en diferentes conjuntos de categorías. Su estructura se puede observar en la ilustración 28.

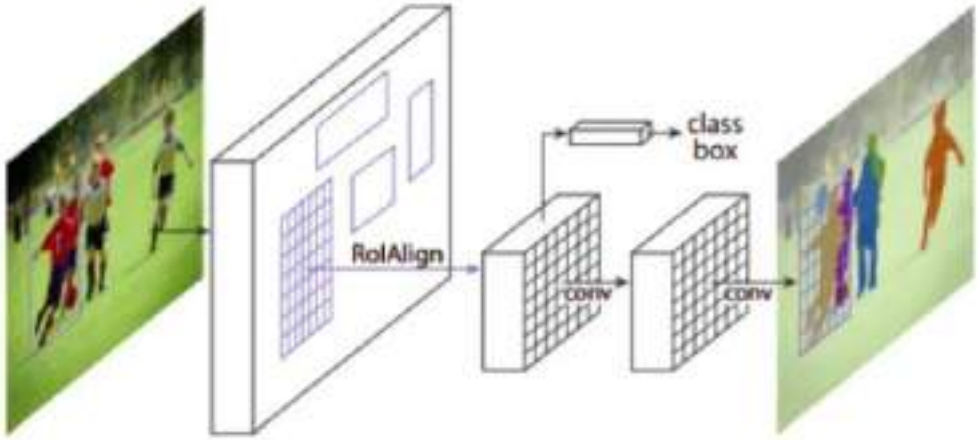


Ilustración 28: Estructura Mask R-CNN (cortesía de Kaiming He)

5.5.1. MASK R-CNN

Al igual que en el apartado de object detection se han descrito las R-CNN, en este se verá qué es Mask R-CNN y en qué se diferencia de ellas.

Mask R-CNN [39] es simplemente una extensión de Faster R-CNN, cuya diferencia reside en que dada una imagen devolverá la etiqueta de clase y las coordenadas del "bounding box", al igual que las R-CNN, además de la máscara del objeto. Para predecir la etiqueta de clase y la caja delimitadora lleva a cabo la misma tarea que la Faster R-CNN; mientras que esta última utiliza la arquitectura ConvNet para extraer características de la imagen en cuestión, Mask R-CNN usa la arquitectura ResNet 101, permitiéndole así extraer estas características y utilizarlas como entrada para la siguiente capa.

Una vez hechos estos mapas de características, se aplica una red de propuesta de región (RPN), la cual predice si un objeto se encuentra presente en dicho lugar y permite obtener aquellas regiones donde el modelo supone que se encuentra algún objeto. Tras haber discernido entre regiones que presentan objetos de los que no, se aplica una capa de agrupación que convierte todas estas regiones a la misma forma, y se pasan a través de una red conectada para predecir la clase de etiqueta y los "bounding box". Además, también generará la máscara de segmentación.

Finalmente, para poder realizar estas tres tareas, calcula la región de interés (RoI), que reduce el tiempo el cálculo, y luego, para todas las regiones predichas, calcula la intersección sobre la unión (IoU). Una vez extraídas todas estas RoI, agrega una rama de máscara a la arquitectura, devolviendo así una máscara de segmentación para cada región que contiene un objeto.

Esta red neuronal se puede utilizar con cualquier framework, pero en este trabajo se implementará en Detectron2, el cual se detalla a continuación.

5.5.2. DETECTRON2

Detectron2 [40] se trata de una mejora de Detectron, el cual era una plataforma de detección de objetos creada por investigadores de Facebook y basada en Caffe2, que empezó siendo implementada con maskrcnn-benchmark y ahora lo es con PyTorch, siendo así flexible y permitiendo el entrenamiento en una o en múltiples GPUs.

Incluye gran variedad de implementaciones para poder llegar a detectar objetos gracias a algoritmos como DensePose, Faster R-CNN, RetinaNet y Mask R-CNN, los cuales ya se encontraban en Detectron, además de Panoptic Feature Pyramid Network (Panoptic FPN), Cascade R-CNN y TensorMask, los cuales son nuevos modelos añadidos en esta versión.

Gracias a la cantidad de algoritmos que implementa, Detectron2 puede realizar una gran variedad de tareas basadas en detección de objetos, ya sea a través de cajas delimitadoras como con máscaras para segmentación de instancias. Además, ofrece soporte para segmentación semántica y panóptica, la cual es una tarea que combina la segmentación semántica junto con la de instancias.

Dentro de esta librería se pueden encontrar implementaciones a varios datasets tales como CityScapes, COCO o PASCAL VOC, los cuales se detallan en el siguiente apartado, y se puede descargar desde su repositorio en GitHub.

5.5.3. OTROS DATASETS Y MARCOS DE TRABAJO PARA EL IMAGE SEGMENTATION

Al igual que en object detection, en image segmentation se encuentran diferentes marcos de trabajo que permiten realizar el propio entrenamiento de la red neuronal; además, también ofrecen datasets donde aparecen tanto imágenes como vídeos que se pueden descargar desde sus páginas oficiales.

A. DATASETS

De entre todos los datasets que existen, se destacan seis de ellos, debido a su idoneidad para el trabajo en cuestión, junto con los archivos que contienen.

- Common Objects in COntext – COCO Dataset [41]
  - Contiene 91 clases
  - Tiene 250.000 personas con puntos clave
  - Su tamaño de descarga es de 37,57 GB
  - Contiene 81 categorías de objetos
  - Disponible bajo licencia Apache 2.0
- PASCAL Visual Object Classes (PASCAL VOC) [42]
  - Contiene 9.963 imágenes
  - Contiene 20 clases diferentes
  - El archivo de descarga es de 2 GB
- The Cityscapes Dataset [43]
  - Contiene 30 clases
  - El dataset está sacado de 50 ciudades diferentes
  - Contiene 25.000 imágenes
  - Contiene vídeos
- The Cambridge-driving Labeled Video Database – CamVid [44]
  - Contiene 32 clases semánticas
  - Contiene más de 700 imágenes
  - Los vídeos están formato AVI y MPEG
  - El archivo de vídeos AVI es de 30 MB
  - El archivo de vídeos MPEG es de 11 MB
- The Waymo Open Dataset [45]
  - Se trata de un dataset de licencia libre
  - Contiene 1.950 segmentos de 20 segundos cada uno
  - Se le pueden acoplar sensores LIDAR, de calibración y cámaras
  - Contiene etiquetas para 4 clases de objetos
  - Se encuentra bajo la licencia de Apache 2.0
- BDD100K [46]
  - El toolkit depende de Python 3.7+

- Contiene 19 clases para segmentación semántica
- Contiene etiquetas para la calzada
- Las etiquetas están en forma JSON

B. MARCOS DE TRABAJO

Al igual que en los datasets, en este apartado se destacan siete frameworks para el Deep Learning, en cuanto al image segmentation, junto con algunas de sus características más destacables.

- FastAI Library [47]
  - Esta librería es capaz de crear una máscara en los objetos de la imagen dada
  - No necesita ninguna instalación, se puede implementar en Google Colab
  - Se puede utilizar la GPU para que vaya más rápido
  - Se puede utilizar tanto en Linux como en Windows
  - Se trata de una librería de PyTorch
  - El lenguaje de programación que utiliza es Python
- Sefexa Image Segmentation Tool [48]
  - Se trata de una herramienta de licencia libre que se puede utilizar para segmentación de imágenes semiautomática, análisis de imágenes y creación de datos reales
  - Requiere Windows 7 o superior
  - Contiene un manual donde se explican detalles sobre el software
- Deepmask [49]
  - Es una implementación de Torch de DeepMask y SharpMask
  - Sus creadores son Facebook Research
  - Se puede utilizar tanto en Linux como en MAC OS X
  - Se puede utilizar la GPU de NVIDIA con capacidad informática 3.5+
  - Contiene los paquetes: COCO API, image, tds, cjson, nnx, optim, inn, cutorch, cunn, cudnn
- MultiPath [50]
  - Se trata de otra implementación de Torch de la red de detección de objetos "A MultiPath Network for Object Detection"
  - Se puede utilizar en Linux
  - Se puede utilizar la GPU de NVIDIA con capacidad informática 3.5+
  - El código depende de Torch-7
- OpenCV [51]
  - Se trata de una librería de código abierto con más de 2.500 algoritmos optimizados
  - Contiene más de 47 mil personas en su comunidad y el número de descargas estimadas es de 18 millones
  - Contiene interfaces para C++, Python, Java y MATLAB
  - Soporta Windows, Android, Linux y Mac OS

- Se están desarrollando interfaces para CUDA y OpenCL

❖ MIScnn [52]
- Se trata de una librería de código abierto para la segmentación de imágenes en el campo de la medicina
- Trabaja con Python
- Basado en Keras con TensorFlow como backend

❖ Fritz [53]
- Se trata de una plataforma que ofrece varias herramientas de visión por ordenador incluyendo herramientas de segmentación de imágenes para teléfonos móviles

6. DESCRIPCIÓN DE LA TAREA REALIZADA

En el presente trabajo se pretende comparar y decidir qué marco de trabajo realiza la detección y el reconocimiento de los objetos para la conducción autónoma de forma más exacta y en el menor tiempo posible, implementando así tanto la detección de objetos como la segmentación de imágenes. Por ello, tras haber extraído varias imágenes encontradas en internet, se procede a comprobar cómo trabajan tanto DarkFlow y Darknet con YOLO, como Detectron2 con Faster R-CNN y Mask R-CNN.

6.1. OBJECT DETECTION PARA LA DETECCIÓN DE OBJETOS EN LA CALZADA

Tal y como se ha explicado en el marco teórico, el "object detection" utiliza redes neuronales convolucionales para poder localizar varios objetos dentro de una imagen o de un vídeo. Además, existen dos tipos de redes neuronales que se diferencian en la cantidad de pasos que deben llevar a cabo para realizar la detección.

En este caso, por un lado, debido a la necesidad de obtener una detección en tiempos ínfimos, ya que la conducción autónoma requiere de toma de decisiones inmediata, se utilizará YOLO, mientras que, por otra parte, debido a que Detectron2 tan solo realiza detecciones de objetos a través de Faster R-CNN se podrá comprobar la diferencia de velocidad entre esta red neuronal que lleva a cabo dos pasos junto con la otra que tan solo realiza uno.

6.1.1. IMÁGENES UTILIZADAS PARA LA DETECCIÓN DE OBJETOS

Como se puede observar en la descripción de la tarea realizada, las imágenes utilizadas para llevar a cabo esta detección son extraídas de internet, por lo que seguidamente se expondrán las originales para que se pueda apreciar la exactitud de las detecciones realizadas a continuación, ya que a veces las cajas delimitadoras no dejan percibir qué clase de objetos se están detectando.

- **Imagen 1**

  La primera imagen del dataset se muestra en la ilustración 26 y se puede observar en ella una autovía con varios coches y algunas señales de tráfico.

Ilustración 29: Imagen 1 original (cortesía de Blanca Silvestre Pedraza)

- **Imagen 2**

La segunda imagen del dataset se ve retratada en la ilustración 30 y se pueden apreciar varias señales junto a dos paradas de autobús en una calle de Madrid.

Ilustración 30: Imagen 2 original (cortesía del periódico ABC)

- **Imagen 3**

En la tercera imagen del dataset, tal y como se ve en la ilustración 31, se pueden apreciar dos personas cruzando un paso de peatones, además de varios coches estacionados y algunas señales de tráfico.

Ilustración 31: Imagen 3 original (cortesía de José Ignacio Arminio)

- **Imagen 4**

En la ilustración 32 se puede apreciar la cuarta imagen de este dataset, en la cual se observa a una persona en una calle donde aparecen varias señales de tráfico y algunos vehículos.

Ilustración 32: Imagen 4 original (cortesía del Ayuntamiento de Las Rozas

6.1.2. YOLO

La ejecución de YOLO se puede realizar mediante Darknet, el cual se considera su implementación oficial, o a través de Darkflow, un marco de trabajo que imita a Darknet pero que está basado en TensorFlow. En este trabajo se ha llevado a cabo la implementación de YOLO en los dos marcos con el fin de poder escoger el que realiza detecciones más exactas en tiempos mínimos.

Además, esta red neuronal convolucional está basada en el dataset de COCO (Common Objects in COntext), el cual ofrece una detección para 80 diferentes clases de objetos; de entre todos ellos se puede observar que puede distinguir entre semáforos, señales de STOP, coches y personas, lo cual nos concierne debido a que el objetivo es poder implementarlo en el vehículo autónomo.

Para poder realizar las debidas detecciones necesita un archivo "configuración", el cual indica a la red neuronal las medidas de los "bounding box", aspectos relacionados con la imagen como pueden ser la saturación o la exposición, cuántos filtros debe utilizar y cuántas clases de objetos debe detectar, además de los "anchor" que puede encontrar; por otro lado, necesita unos pesos (weights) ya entrenados que son los encargados de detectar y realizar las clasificaciones de cada objeto.

A diferencia de Darkflow, Darknet también necesita un archivo "data" donde se especifica el número de clases de objetos a detectar, la ubicación de los documentos donde se encuentran el entrenamiento y la validación de los objetos, los cuales contienen las medidas y el tipo de clase de cada una de las imágenes entrenadas, y la lista de nombres que clasifica cada tipo de objeto.

Por lo que se puede observar, tanto Darknet como Darkflow comparten los mismos archivos "configuración", "data" y "names", además de los mismos pesos; pero, los comandos que utilizan para realizar las detecciones son diferentes y se podrán observar a continuación.

6.1.3. PUESTA EN MARCHA DE YOLO EN DARKFLOW

Como ya se ha expuesto anteriormente, Darkflow es un marco de trabajo que permite la implementación de TensorFlow en YOLO e imita a Darknet. Se puede descargar desde su repositorio en GitHub [54] (Ver anexo I) y se debe tener presente que es bastante restrictivo en cuanto a las versiones de las dependencias y del propio YOLO en sí, aspecto a contemplar a la hora de su instalación.

Al presentar estas restricciones, los pesos ya entrenados que se necesitan para poder realizar las detecciones no pueden ser descargados desde la página oficial de Darknet, ya que estos van siendo actualizados con el paso del tiempo. Por ello, en su repositorio, dan acceso a una carpeta que se encuentra en Google Drive donde se hallan guardados los pesos originales sin ninguna actualización.

Además, las únicas dependencias cuyas versiones se deben tener en cuenta son las de Python y TensorFlow, ya que existe una librería llamada "tensorflow.contrib" que las

restringe. Así pues, Python y TensorFlow deben estar en las versiones 3.7.x y 1.15, respectivamente. También se debe considerar que la versión de YOLO que se utiliza con Darkflow es la 2, es decir, YOLOv2.

Una vez se ha instalado correctamente es muy fácil de utilizar gracias a la simplicidad de los comandos y a su estructura intuitiva. Tiene la posibilidad de realizar detecciones en imágenes, vídeos y cámaras web, además de poder utilizar las demos de muestra que facilita. También presenta una guía donde explica cómo hacer tu propio dataset y cómo realizar su entrenamiento.

El objetivo de esta tarea es comprobar qué marco de trabajo lleva a cabo una detección más exacta en el menor tiempo posible y, como se pretende implementar en un vehículo autónomo, los objetos a detectar estarán relacionados con aquellos que se puedan encontrar en la calzada. Para ello, se presentan las cuatro imágenes anteriores que permitirán comprobar qué objetos detecta y cuánto tiempo necesita para realizar dicha detección.

Como se quiere observar qué opción es la idónea, se realizará la comparación utilizando tan solo la CPU o la GPU. Además, la detección que se llevará a cabo es la misma, lo único que cambiará será la velocidad a la que lo realice, por lo que tan solo se presentará una detección por cada imagen.

- El comando para llevar a cabo la detección de las cuatro imágenes con Darkflow sin utilizar la GPU es:

```
$ flow --imgdir my_photos/ --model cfg/yolo.cfg --load bin/yolo.weights
```

- El comando para llevar a cabo la detección de las cuatro imágenes con Darkflow utilizando la GPU es:

```
$ flow --imgdir my_photos/ --model cfg/yolo.cfg --load bin/yolo.weights --gpu 1.0
```

donde “flow” es el comando que da las órdenes, “imgdir” es donde se encuentran todas las imágenes, “model” es la ubicación donde está el archivo configuración y “load”, tal y como dice la palabra, carga los pesos ya entrenados.

Los resultados de la detección realizada, tanto en GPU como en CPU, son:

- **Imagen 1:** en la ilustración 33 se observa la detección de varios coches junto a personas dentro de ellos, además de una señal de tráfico, aunque no está correctamente reconocida ya que indica que es de STOP mientras se trata de una señal de prohibición.

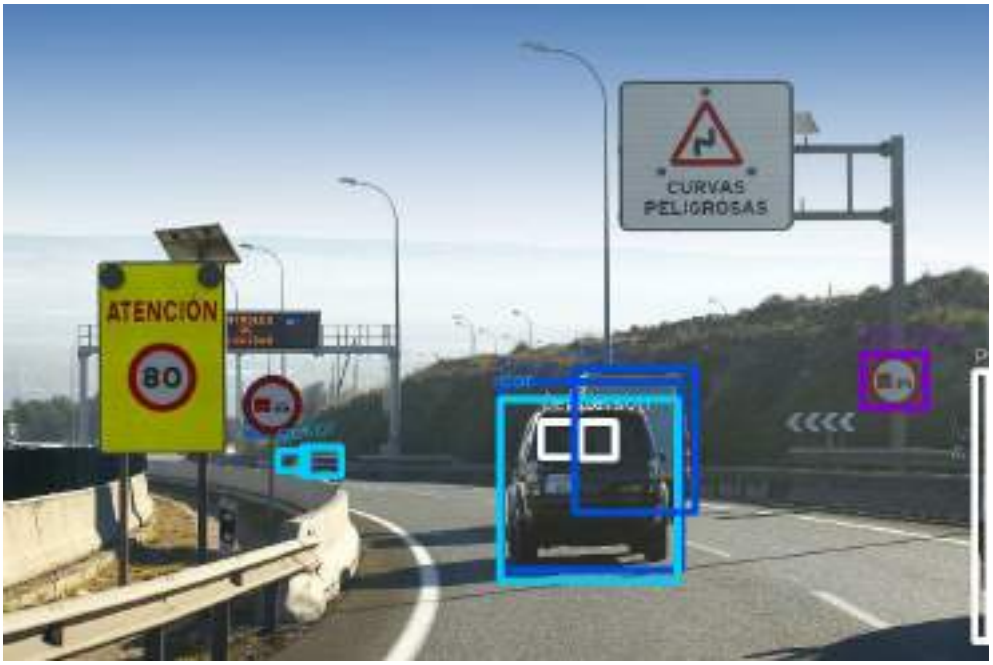


Ilustración 33: Detección imagen 1 en Darkflow

- **Imagen 2:** en la ilustración 34 se puede apreciar la detección de varios objetos que no se relacionan con los que se encuentran en la imagen, ya que relaciona una jirafa con varias estatuas, un frisbee con una señal de prohibición y varios coches y personas con objetos de la calle. Por otro lado, sí que reconoce y detecta de forma correcta el semáforo.

Ilustración 34: Detección imagen 2 en Darkflow

- **Imagen 3:** en la ilustración 35 se observa la detección de una gran cantidad de objetos que, al igual que en la imagen anterior, no se relacionan con los que se encuentran en ella, ya que reconoce un semáforo en un edificio y un coche y un parquímetro en un contenedor de basura y sus alrededores. Por el contrario, sí reconoce la señal de tráfico, aunque no la clasifica correctamente, y las personas junto con las bolsas y mochilas que llevan con ellas.

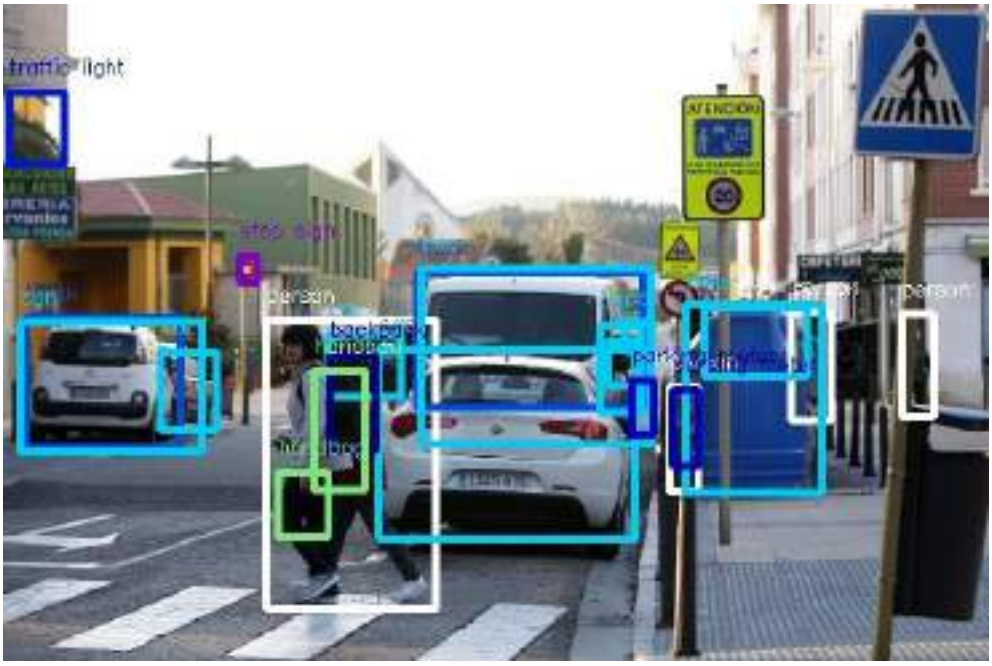


Ilustración 35: Detección imagen 3 en Darkflow

- **Imagen 4:** en la ilustración 36 se llega a realizar de forma correcta la detección de los coches y la persona, además de la señal de STOP, mientras que se detecta de forma errónea un semáforo, una señal de STOP en un edificio y un parquímetro.

Ilustración 36: Detección imagen 4 en Darkflow

Los tiempos de ejecución empleados para realizar la detección de las cuatro imágenes son:

- Utilizando la CPU: 4,550001859554917s
- Utilizando la GPU: 3,0867843627929688s

Se puede observar una clara diferencia en cuanto a la velocidad de ejecución, ya que utilizando la GPU se necesita menos tiempo para llevar a cabo las detecciones. Más adelante se hará una comparación más detallada.

6.1.4. PUESTA EN MARCHA DE YOLO EN DARKNET

Este marco de trabajo se puede obtener por medio de su repositorio en GitHub [55] (Ver anexo II). Además, requiere de la instalación de OpenCV, para el procesamiento de las imágenes, y de CUDA, para la mejora de la velocidad de detección.

A diferencia de Darkflow, este no presenta ninguna restricción de versiones entre sus dependencias, por lo que se pueden utilizar tanto los diferentes pesos actualizados como las diversas versiones de YOLO tales como YOLOv2, YOLOv3 y YOLOv4, entre otras.

Tal vez su instalación sea un poco más pesada en comparación a la de Darkflow, debido a que instalar OpenCV y CUDA es bastante engorroso; pero, una vez realizada, sus comandos también son fáciles e intuitivos y permite realizar detecciones en imágenes, vídeos o cámaras web.

En Darknet la detección de imágenes sí que se lleva a cabo de forma individual, por lo que se mostrarán las imágenes previas con sus detecciones realizadas y el tiempo requerido. Además, la intención de utilizar GPU o CPU no se expresa en el comando, sino que se debe cambiar la configuración del archivo "MakeFile", sustituyendo GPU=O por GPU=1 si se quiere utilizar la GPU, junto con el cuDNN=1, y poniendo OPENCV=1 para poder usarlo.

El comando global para llevar a cabo la detección de las imágenes es:

```
$ ./darknet detector test cfg/coco.data cfg/yolo.cfg yolo.weights
data/imagen.jpg
```

donde "./darknet" es el comando que realiza las órdenes, "detector test" indica que se va a realizar la detección de una imagen, "cfg/coco.data" indica dónde se encuentra el archivo data, al igual que "cfg/yolo.cfg" indica dónde se encuentra el archivo configuración, "yolo.weights" muestra qué pesos se van a utilizar, y "data/imagen.jpg" indica dónde se encuentra y cómo se llama la imagen requerida.

A continuación, se van a mostrar las detecciones realizadas utilizando tanto la CPU como la GPU:

- **Imagen 1:** en la ilustración 37 se puede observar el mismo reconocimiento que utilizando Darkflow diferenciándose en que en Darknet no se detectan las personas dentro del coche.

El comando utilizado para realizar la detección con Darknet en la imagen 1 es:

```
$ ./darknet detector test cfg/coco.data cfg/yolo.cfg yolo.weights
data/imagen1.jpg
```

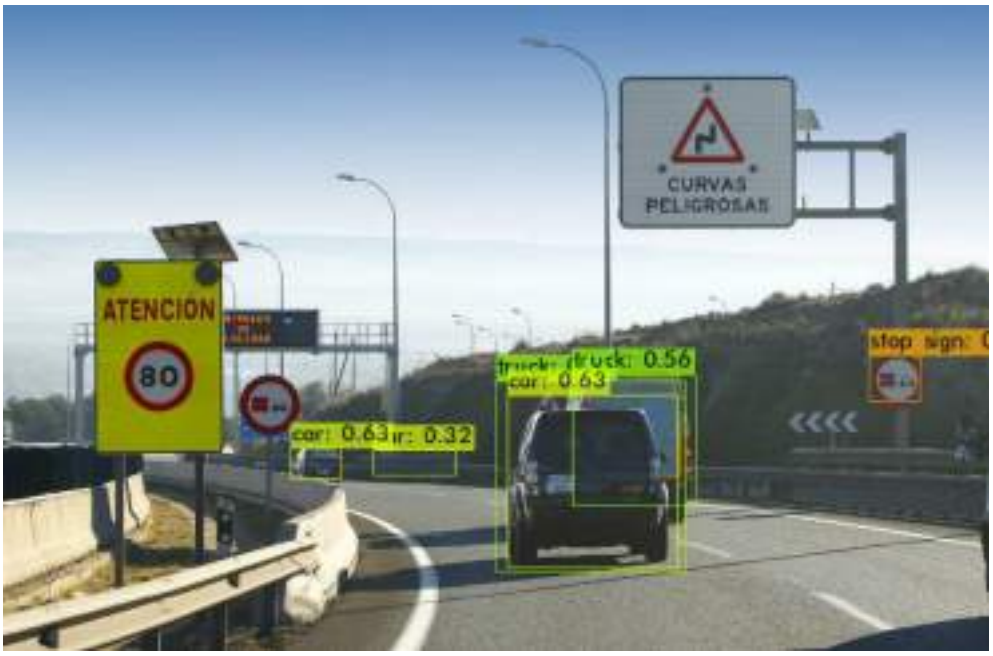


Ilustración 37: Detección imagen 1 en Darknet

Los tiempos de ejecución empleados para realizar la detección en la imagen 1 son:

- Utilizando tan solo la CPU: 3,18823s
- Utilizando tan solo la GPU: 0,731841s

❖ **Imagen 2:** en la ilustración 38 se puede apreciar que el marco de trabajo no es capaz de detectar ningún objeto de los presentes en ella.

El comando utilizado para realizar la detección con Darknet en la imagen 2 es:

```
$ ./darknet detector test cfg/coco.data cfg/yolo.cfg yolo.weights
data/imagen2.jpg
```

Ilustración 38: Detección imagen 2 utilizando Darknet

Los tiempos de ejecución empleados para realizar la detección en la imagen 2 son:

- Utilizando tan solo la CPU: 3,174868s
- Utilizando tan solo la GPU: 0,374633s

❖ **Imagen 3:** en la ilustración 39 se puede observar que realiza de forma correcta la detección de los coches y las personas, mientras que asocia la etiqueta de coche a un contenedor de la basura.

El comando utilizado para realizar la detección con Darknet en la imagen 3 es:

```
$ ./darknet detector test cfg/coco.data cfg/yolo.cfg yolo.weights
data/imagen3.jpg
```

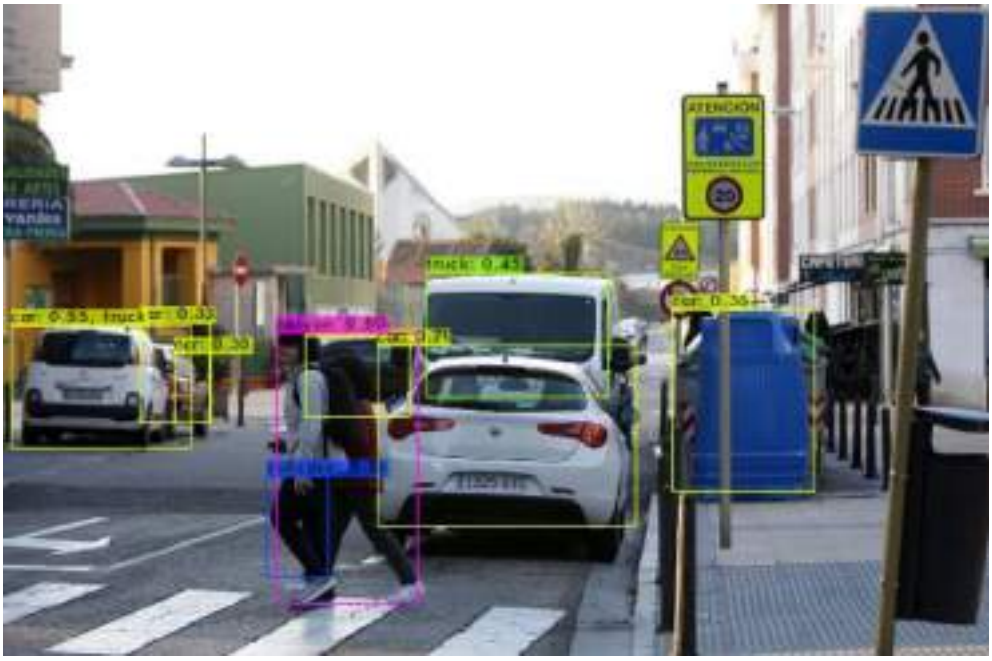


Ilustración 39: Detección imagen 3 utilizando Darknet

Los tiempos de ejecución empleados para realizar la detección en la imagen 3 son:

- Utilizando tan solo la CPU: 3,13617s
- Utilizando tan solo la GPU: 0,372333s

❖ **Imagen 4:** en la ilustración 40 se observa que se realiza de forma correcta tanto la detección de los coches como de la persona y de la señal de STOP.

El comando utilizado para realizar la detección con Darknet en la imagen 4 es:

```
$ ./darknet detector test cfg/coco.data cfg/yolo.cfg yolo.weights
data/imagen4.jpg
```

Ilustración 40: Detección imagen 4 utilizando Darknet

Los tiempos de ejecución empleados para realizar la detección en la imagen 4 son:

- Utilizando tan solo la CPU: 3,263616s
- Utilizando tan solo la GPU: 0,373693s

A la vista de los resultados, se puede afirmar que existe una gran diferencia entre la utilización de la CPU y la GPU, ya que esta última realiza las ejecuciones a una velocidad mayor, empleando así tiempos ínfimos. De todas formas, a continuación, se realiza una comparación más exacta.

6.1.5. COMPARACIÓN DE RESULTADOS ENTRE DARKFLOW Y DARKNET

Tras haber sido utilizado el mismo dataset para realizar la detección con YOLO tanto a través de Darkflow como de Darknet, se procede a observar detenidamente cada imagen para constatar las detecciones llevadas a cabo, y así poder comprobar la exactitud de cada marco de trabajo, y el tiempo que necesitan para ejecutarlo.

Por ello, se van a realizar unas tablas comparativas que reflejen los tiempos de ejecución de cada una de las detecciones y, posteriormente, se observará su grado de exactitud.

❖ Detección de objetos utilizando CPU

	Darkflow (tiempo estimado)	**Darknet**
Imagen 1	1,13750047 s	3,18823 s
Imagen 2	1,13750047 s	3,174868 s
Imagen 3	1,13750047 s	3,13617 s
Imagen 4	1,13750047 s	3,263616 s
TOTAL	**4,550001859 s**	**12,762884 s**

Tabla 6: Comparación del tiempo empleado entre Darkflow y Darknet utilizando la CPU

❖ Detección de objetos utilizando GPU

	Darkflow (tiempo estimado)	**Darknet**
Imagen 1	0,77169609 s	0,731841 s
Imagen 2	0,77169609 s	0,374633 s
Imagen 3	0,77169609 s	0,372333 s
Imagen 4	0,77169609 s	0,373693 s
TOTAL	**3,086784362 s**	**1,8525 s**

Tabla 7: Comparación del tiempo empleado entre Darkflow y Darknet utilizando la GPU

En cuanto a la detección de objetos realizada, se puede observar que Darkflow llega a reconocer gran variedad de ellos mientras que Darknet incluso deja una imagen sin detectar ningún objeto. Aun así, se puede advertir que no todas las etiquetas asignadas a los objetos reconocidos por Darkflow se asociar de forma correcta, ya que asigna nombres que no pertecen a esos objetos. Por ello, aun no llevando a cabo una detección con gran

exactitud, en cuanto a este ítem se refiere, Darknet sería la mejor opción ya que la mayoría de los objetos que reconoce sí que se pueden relacionar con sus etiquetas.

Por otro lado, si se observa el tiempo de ejecución, cuando tan solo se utiliza la CPU se puede afirmar que Darkflow emplea menos tiempo que Darknet para realizar la detección de las cuatro imágenes. Por el contrario, si se utiliza la GPU, indudablemente Darknet es la mejor opción ya que, aun realizando las detecciones de las cuatro imágenes en menor tiempo que Darkflow, emplea tan solo dos segundos, de forma estimada, para ello, convirtiéndolo así en el marco de trabajo idóneo para su utilización en la conducción autónoma.

A continuación, una vez concordado que el marco de trabajo con el que se implementará YOLO será Darknet trabajando con la GPU, se procederá a comprobar la exactitud y el tiempo de ejecución necesario para llevar a cabo la misma labor en Detectron2 para, posteriormente, seleccionar el mejor marco de trabajo relacionado con la detección de objetos.

6.1.6. PUESTA EN MARCHA DE DETECTRON2 JUNTO CON FASTER R-CNN

Detectron2, al igual que los anteriores marcos de trabajo, se puede obtener a través de su repositorio en GitHub [56] (Ver anexo III). Contiene un tutorial en Google Colab y varios archivos que ofrecen información sobre qué dependencias instalar y cómo empezar a utilizarlo. Además, como está basado en PyTorch, es necesario instalarlo teniendo en cuenta su número de versión, ya que esta depende de la versión de CUDA que se tenga instalada.

Por otro lado, aun ofreciendo una gran variedad de redes neuronales ya entrenadas, se utilizará Faster R-CNN porque es la que puede realizar la detección de objetos. Asimismo, se utilizará el dataset de COCO para poder llevar a cabo una comparación adecuada.

En Detectron2, al igual que en Darknet, se realiza las detecciones de las imágenes de forma individual, difiriendo en la posibilidad de utilizar CPU o GPU, ya que de forma predeterminada trabaja siempre junto con la GPU.

El comando general para llevar a cabo la detección de los objetos es:

```
$ python3 demo.py --config-file ../configs/COCO-
Detection/faster_rcnn_R_50_DC5_3x.yaml --input ../fotos/imagen.jpg
--opts MODEL.WEIGHTS detectron2://COCO-
Detection/faster_rcnn_R_50_DC5_3x/137849425/model_final_68d202.pkl
```

donde “python3” es el comando que abre el archivo “demo.py” donde se encuentra el código escrito en Python que se encarga de dar las órdenes, “config-file” determina dónde se encuentra el archivo de configuración necesario y “opts” de dónde se puede obtener los pesos relacionados con ese archivo de configuración, mientras que “input” indica dónde se encuentra la imagen a examinar.

Tanto el archivo de configuración como la ruta a sus pesos se encuentran en un archivo llamado MODEL_ZOO.md donde aparece un gran listado con varias opciones dependiendo del dataset que se vaya a utilizar y del tipo de detección que se quiera realizar.

A continuación, se muestran las detecciones realizadas en cada una de las imágenes:

- **Imagen 1:** en la ilustración 41 se observa tan solo la detección de dos coches y una furgoneta.

  El comando utilizado para realizar la detección con Detectron2 en la imagen 1 es:

```
$ python3 demo.py --config-file ../configs/COCO-Detection/faster_rcnn_R_50_DC5_3x.yaml --input ../fotos/imagen1.jpg --opts MODEL.WEIGHTS detectron2://COCO-Detection/faster_rcnn_R_50_DC5_3x/137849425/model_final_68d202.pkl
```

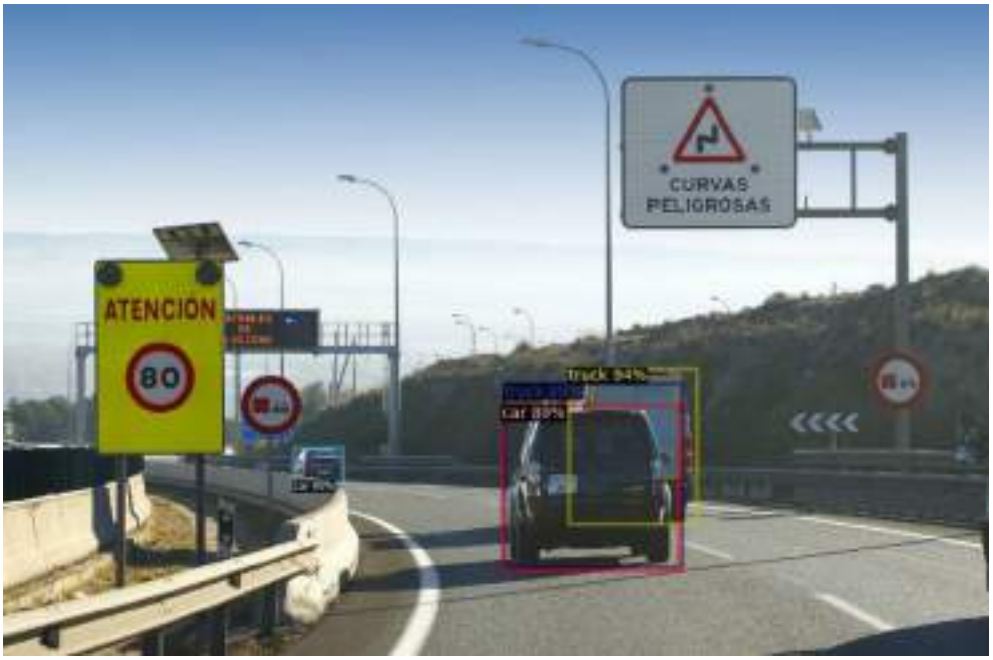


Ilustración 41: Detección imagen 1 utilizando Detectron2

El tiempo de ejecución empleado para realizar la detección en la imagen 1 es:

- Utilizando tan solo la GPU: 0,96s

- **Imagen 2:** en la ilustración 42 se observa la detección de varias personas donde sí se corresponden con las que se encuentran en la imagen mientras que otras se refieren a otros objetos. También se detecta una planta y una señal de tráfico, aunque la clasifica como señal de STOP mientras que se trata de una señal de prohibición. Aun así, sí reconoce el semáforo presente en la imagen.

  El comando utilizado para realizar la detección con Detectron2 en la imagen 2 es:

```
$ python3 demo.py --config-file ../configs/COCO-
Detection/faster_rcnn_R_50_DC5_3x.yaml --input ../fotos/imagen2.jpg
--opts MODEL.WEIGHTS detectron2://COCO-
Detection/faster_rcnn_R_50_DC5_3x/137849425/model_final_68d202.pkl
```

Ilustración 42: Detección imagen 2 utilizando Detectron2

El tiempo de ejecución empleado para realizar la detección en la imagen 2 es:

- Utilizando tan solo la GPU: 0,96s

❖ **Imagen 3:** en la ilustración 43 se observa una correcta detección de los vehículos y las personas que se encuentran en la imagen, mientras que reconoce la señal de tráfico, pero la clasifica como señal de STOP cuando se trata de una señal de prohibición.

El comando utilizado para realizar la detección con Detectron2 en la imagen 3 es:

```
$ python3 demo.py --config-file ../configs/COCO-
Detection/faster_rcnn_R_50_DC5_3x.yaml --input ../fotos/imagen3.jpg
--opts MODEL.WEIGHTS detectron2://COCO-
Detection/faster_rcnn_R_50_DC5_3x/137849425/model_final_68d202.pkl
```

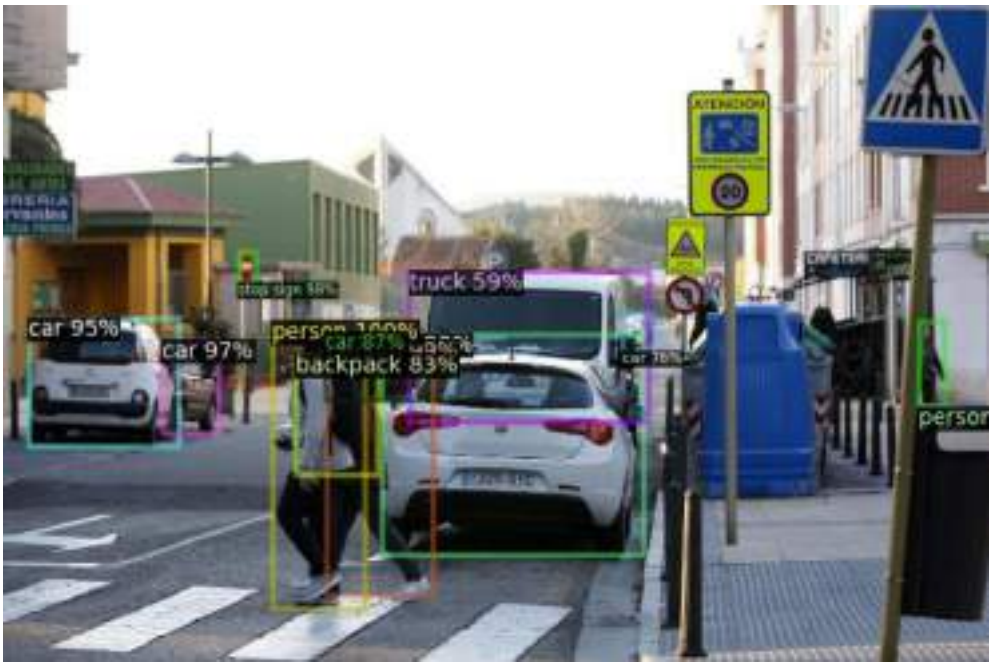


Ilustración 43: Detección imagen 3 utilizando Detectron2

El tiempo de ejecución empleado para realizar la detección en la imagen 3 es:

- Utilizando tan solo la GPU: 0,99s

❖ **Imagen 4:** en la ilustración 44 se observa una clara detección de los coches y la persona que se encuentra en ella además de la señal de STOP. Por otro lado, clasifica como parquímetro a un objeto que no se distingue bien qué es exactamente, pero se podría considerar como parquímetro.

El comando utilizado para realizar la detección con Detectron2 en la imagen 4 es:

```
$ python3 demo.py --config-file ../configs/COCO-
Detection/faster_rcnn_R_50_DC5_3x.yaml --input ../fotos/imagen4.jpg
--opts MODEL.WEIGHTS detectron2://COCO-
Detection/faster_rcnn_R_50_DC5_3x/137849425/model_final_68d202.pkl
```

Ilustración 44: Detección imagen 4 utilizando Detectron2

El tiempo de ejecución empleado para realizar la detección en la imagen 4 es:

- Utilizando tan solo la GPU: 0,99s

Tras realizar las cuatro detecciones se puede afirmar que los tiempos de ejecución empleados son bastante pequeños, además de que se podría considerar que realiza una buena detección.

6.1.7. COMPARACIÓN DE RESULTADOS ENTRE DARKNET Y DETECTRON2

Al igual que en la comparación realizada anteriormente, se procederá a efectuar una tabla comparativa sobre los tiempos de ejecución de las detecciones llevadas a cabo por Darknet y Detectron2. Además, se compararán el número de objetos reconocidos junto con su exactitud.

	Darknet	**Detectron2**
Imagen 1	0,731841 s	0,96 s
Imagen 2	0,374633 s	0,96 s
Imagen 3	0,372333 s	0,99 s
Imagen 4	0,373693 s	0,99 s
TOTAL	**1,8525 s**	**3,9 s**

Tabla 8: Comparación del tiempo empleado entre Darknet y Detectron2

Si se tiene en cuenta el tiempo de ejecución, se puede observar que Darknet trabajando con GPU sigue teniendo tiempos de respuestas menores a Detectron2, ya que emplea cerca de los dos segundos para realizar la detección de las cuatro imágenes, mientras que Detectron2 necesita casi cuatro segundos para llevarlo a cabo.

Sin embargo, se comprueba que Detectron2 detecta muchos más objetos, aunque menos que Darkflow, dentro de la imagen y de los cuales Darknet no es capaz de reconocer. Por ello, se van a realizar unas tablas comparativas escogiendo aquellos objetos, de izquierda a derecha, que llegan a distinguir los dos marcos de trabajo y se verificará su exactitud de detección.

- Imagen 1

	Darknet	**Detectron2**
Coche	63%	89%
Coche	63%	89%
Camioneta	56%	94%
TOTAL	**60,7%**	**90,7%**

Tabla 9: Comparación de la exactitud de detección de la imagen 1 entre Darknet y Detectron2

- Imagen 3

	Darknet	**Detectron2**
Coche	55%	95%
Coche	33%	97%
Persona	80%	100%
Coche	71%	76%
Camioneta	45%	59%
TOTAL	**56,8%**	**85,4%**

Tabla 10: Comparación de la exactitud de detección de la imagen 3 entre Darknet y Detectron2

- Imagen 4

	Darknet	**Detectron2**
Señal de STOP	50%	100%
Persona	72%	100%
Coche	69%	100%
Coche	78%	100%
TOTAL	**67,25%**	**100%**

Tabla 11: Comparación de la exactitud de detección de la imagen 4 entre Darknet y Detectron2

Como se puede observar, sin tener en cuenta que Detectron2 llega a detectar más objetos que Darknet, ya que este último en la imagen 2 no distingue ninguno, Detectron2 realiza detecciones mucho más exactas e incluso se podría decir que idóneas porque en las tres imágenes alcanza una exactitud mayor del 70% mientras que Darknet se encuentra en todas por debajo de este porcentaje.

Así pues, se podría afirmar que Detectron2, aun siendo un poco más lento en su velocidad de detección, reconoce los objetos con gran exactitud, otorgándole así la idoneidad para la conducción autónoma.

Sin embargo, este trabajo busca el mejor framework capaz de detectar las señales de tráfico y, como se ha podido comprobar, el dataset de COCO no realiza esta función con buenos resultados. Además, en los datasets que contiene Detectron2, tan solo hay uno que está relacionado con estas señales y es el de CityScapes. El problema reside en que este tan

solo se puede aplicar para la segmentación de instancias, por lo que, a continuación, se expondrá cómo realizar la segmentación de imágenes para la detección de objetos.

6.2. IMAGE SEGMENTATION PARA LA DETECCIÓN DE OBJETOS EN LA CALZADA

Como se ha explicado en la parte del marco teórico, la segmentación de imágenes funciona igual que la detección de objetos, simplemente se diferencian en que en esta se facilita un tipo de información adicional gracias a la máscara que aplica, la cual permite obtener información sobre la forma del objeto.

Además, existen dos tipos de segmentación, la segmentación semántica y la segmentación de instancias. En este apartado se probará esta última ya que, en vez de englobar los objetos de un mismo grupo en una sola capa, diferencia entre estos dentro de su grupo.

Por otro lado, al tratarse de un nuevo tipo de reconocimiento de objetos, primeramente, se volverá a utilizar el dataset de COCO para poder comparar su tiempo de ejecución y su exactitud de detección y luego, si no difieren mucho sus resultados de los obtenidos anteriormente, se pondrá a prueba el dataset de CityScapes.

El comando general para llevar a cabo la segmentación de instancias es:

```
$ python3 demo.py --config-file ../configs/COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml --input ../fotos/imagen.jpg --opts MODEL.WEIGHTS detectron2://COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x/137849600/model_final_f10217.pkl
```

donde lo único que se diferencia respecto al comando que se utilizaba en la detección de objetos es el tipo de archivo de configuración y la ruta donde se encuentran sus pesos.

Tras haber realizado la segmentación en las cuatro imágenes, los resultados son:

- **Imagen 1:** se puede observar en la ilustración 45 que realiza el reconocimiento de dos coches y una furgoneta, al igual que en la detección de objetos con Detectron2 en la dicha imagen.

  El comando utilizado para realizar la segmentación con Detectron2 en la imagen 1 es:

```
$ python3 demo.py --config-file ../configs/COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml --input ../fotos/imagen1.jpg --opts MODEL.WEIGHTS detectron2://COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x/137849600/model_final_f10217.pkl
```

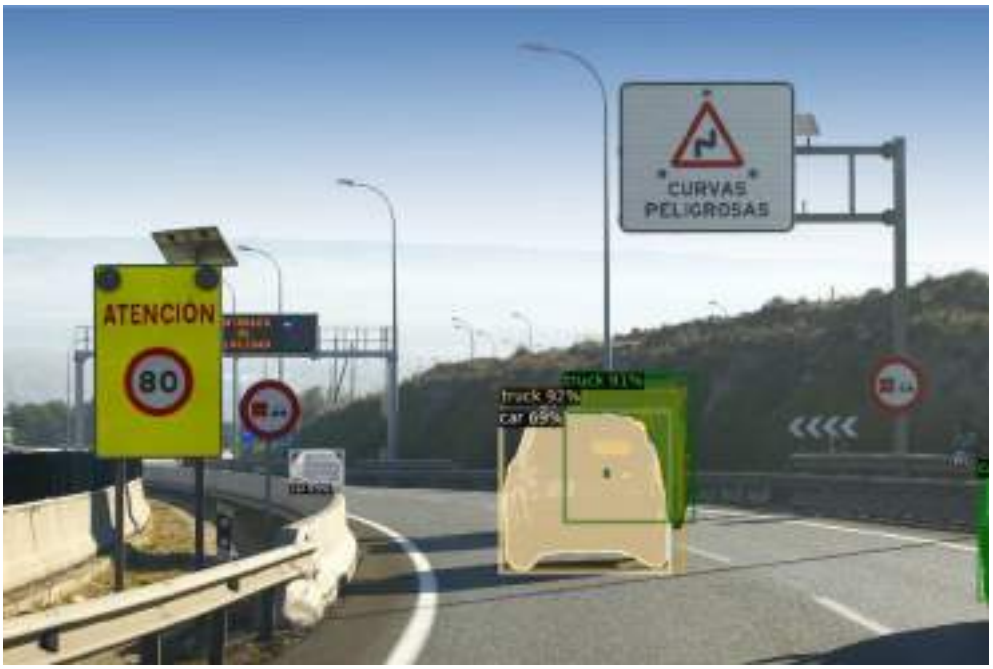


Ilustración 45: Segmentación imagen 1 utilizando Detectron2

El tiempo de ejecución empleado para realizar la segmentación en la imagen 1 es:

- Utilizando tan solo la GPU: 0,62s

❖ **Imagen 2:** en la ilustración 46 se puede apreciar la detección de personas, un semáforo y una señal de STOP, aun siendo esta una señal de prohibición.

El comando utilizado para realizar la segmentación con Detectron2 en la imagen 2 es:

```
$ python3 demo.py --config-file ../configs/COCO-
InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml --input
../fotos/imagen2.jpg --opts MODEL.WEIGHTS detectron2://COCO-
InstanceSegmentation/mask_rcnn_R_50_FPN_3x/137849600/model_final_f
10217.pkl
```

Ilustración 46: Segmentación imagen 2 utilizando Detectron2

El tiempo de ejecución empleado para realizar la segmentación en la imagen 2 es:

- Utilizando tan solo la GPU: 0.6s

❖ **Imagen 3:** en la ilustración 47 se puede observar que detecta las personas junto con las mochilas que llevan, además de los coches y una señal de tráfico.

El comando utilizado para realizar la segmentación con Detectron2 de la imagen 3 es:

```
$ python3 demo.py --config-file ../configs/COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml --input ../fotos/imagen3.jpg --opts MODEL.WEIGHTS detectron2://COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x/137849600/model_final_f10217.pkl
```

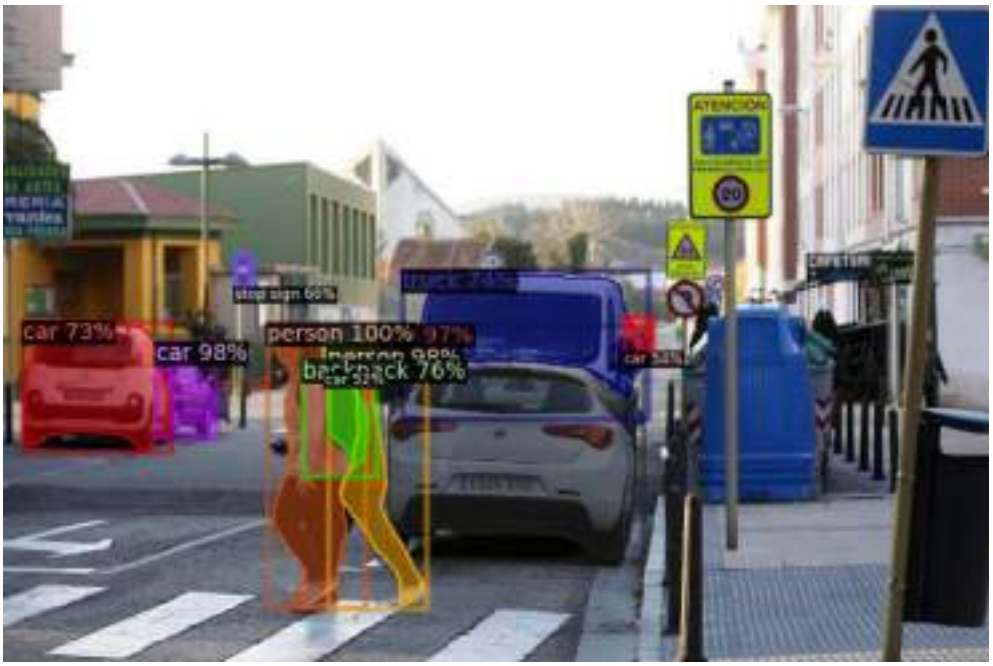


Ilustración 47: Segmentación imagen 3 utilizando Detectron2

El tiempo de ejecución empleado para realizar la segmentación en la imagen 3 es:

- Utilizando tan solo la GPU: 0.6s

❖ **Imagen 4:** en la ilustración 48, al igual que en el apartado anterior, se puede observar que detecta la persona, varios coches y la señal de STOP.

El comando utilizado para realizar la segmentación con Detectron2 en la imagen 4 es:

```
$ python3 demo.py --config-file ../configs/COCO-
InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml --input
../fotos/imagen4.jpg --opts MODEL.WEIGHTS detectron2://COCO-
InstanceSegmentation/mask_rcnn_R_50_FPN_3x/137849600/model_final_f
10217.pkl
```

Ilustración 48: Segmentación imagen 4 utilizando Detectron2

El tiempo de ejecución empleado para realizar la segmentación en la imagen 4 es:

- Utilizando tan solo la GPU: 0,63s

A diferencia de Detectron2 utilizando la red neuronal Faster R-CNN, cuando utiliza la red neuronal Mask R-CNN para llevar a cabo la segmentación de instancias se puede comprobar que lo realiza en menos tiempo, empleando así una media de 0,61 segundos o, lo que es lo mismo, 2,45 segundos para ejecutar el reconocimiento de las cuatro imágenes.

En cuanto a la exactitud de la detección de los objetos, se podría decir que es menos preciso, pero no presenta una gran diferencia respecto a la detección llevada a cabo anteriormente, por lo que se procederá a utilizar el dataset de CityScapes junto a la red neuronal Mask R-CNN para poder comprobar si esta es la idónea para la conducción autónoma.

6.2.1. UTILIZACIÓN DEL DATASET DE CITYSCAPES JUNTO CON DETECTRON2 PARA LA DETECCIÓN DE SEÑALES DE TRÁFICO

Tal y como se ha expuesto en el apartado de datasets dentro de la segmentación de imágenes del marco teórico, el dataset de CityScapes contiene treinta clases diferentes de objetos, entre ellos vehículos, humanos, señales de tráfico y semáforos. Para poder comprobar cómo de exactas realiza las detecciones y cuánto tiempo necesita para su ejecución, se escribirá el siguiente comando:

```
$ python3 demo.py --config-file
../configs/Cityscapes/mask_rcnn_r_50_FPN.yaml --input
../fotos/imagen.jpg --opts MODEL.WEIGHTS ../model_final_af9cf5.pkl
```

donde lo único que difiere de los comandos utilizados en segmentación anteriormente es el archivo de configuración y el tipo de pesos.

Tras haber realizado la detección en las cuatro imágenes anteriores los resultados son:

- **Imagen 1:** en la ilustración 49 se puede apreciar la detección de los dos coches y la furgoneta.

  El comando utilizado para realizar la segmentación con el dataset de CityScapes en la imagen 1 es:

```
$ python3 demo.py --config-file
../configs/Cityscapes/mask_rcnn_r_50_FPN.yaml --input
../fotos/imagen1.jpg --opts MODEL.WEIGHTS
../model_final_af9cf5.pkl
```

Ilustración 49: Segmentación de la imagen 1 utilizando dataset de CityScapes en Detectron2

El tiempo de ejecución empleado para realizar la segmentación en la imagen 1 es:

- Utilizando tan solo la GPU: 1,11s

❖ **Imagen 2:** en la ilustración 50 se puede observar que no llega a detectar ningún objeto de los presentes.

El comando utilizado para realizar la segmentación con el dataset de CityScapes en la imagen 2 es:

```
$ python3 demo.py --config-file
../configs/Cityscapes/mask_rcnn_r_50_FPN.yaml --input
../fotos/imagen2.jpg --opts MODEL.WEIGHTS
../model_final_af9cf5.pkl
```

Ilustración 50: Segmentación de la imagen 2 utilizando dataset de CityScapes en Detectron2

El tiempo de ejecución empleado para realizar la segmentación en la imagen 2 es:

- Utilizando tan solo la GPU: 0,77s

❖ **Imagen 3:** en la ilustración 51 se puede observar tan solo la detección de las personas y de algunos coches.

El comando utilizado para realizar la segmentación con el dataset de CityScapes en la imagen 3 es:

```
$ python3 demo.py --config-file
../configs/Cityscapes/mask_rcnn_r_50_FPN.yaml --input
../fotos/imagen3.jpg --opts MODEL.WEIGHTS
../model_final_af9cf5.pkl
```

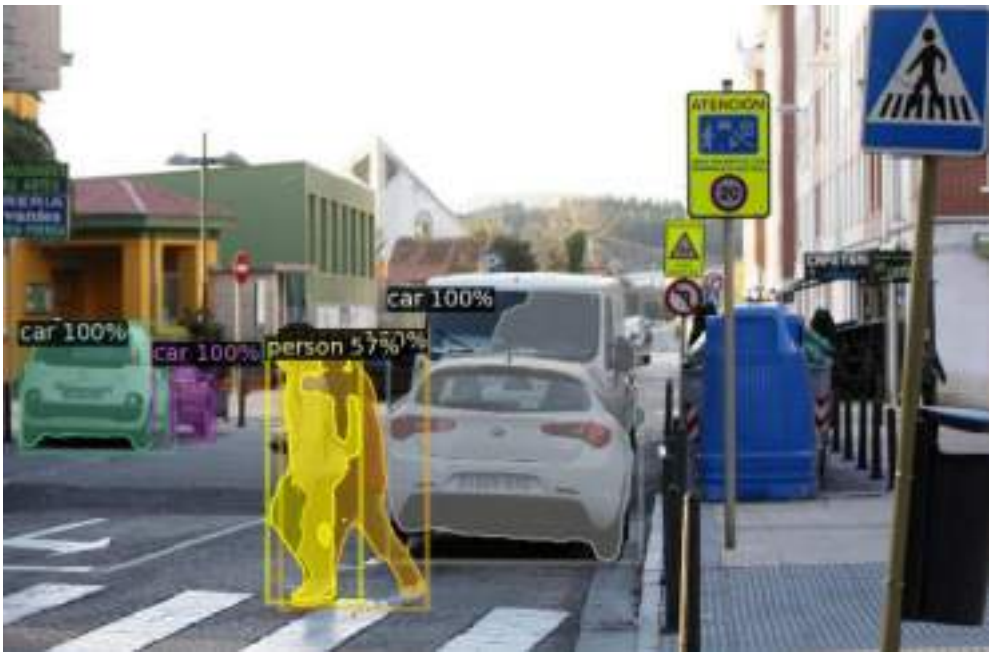


Ilustración 51: Segmentación de la imagen 3 utilizando dataset de CityScapes en Detectron2

El tiempo de ejecución empleado para realizar la segmentación en la imagen 3 es:

- Utilizando tan solo la GPU: 0,73s

❖ **Imagen 4:** en la ilustración 52 se observa que tan solo se realiza la detección de dos coches estacionados.

El comando utilizado para realizar la segmentación con el dataset de CityScapes en la imagen 4 es:

```
$ python3 demo.py --config-file
../configs/Cityscapes/mask_rcnn_r_50_FPN.yaml --input
../fotos/imagen4.jpg --opts MODEL.WEIGHTS
../model_final_af9cf5.pkl
```

Ilustración 52: Segmentación de la imagen 4 utilizando dataset de CityScapes en Detectron2

El tiempo de ejecución empleado para realizar la segmentación en la imagen 4 es:

- Utilizando tan solo la GPU: 0,8s

Se puede comprobar claramente que, con este dataset y sus respectivos archivos de configuración y pesos, se detecta menos objetos que con el dataset de COCO y, además, de que necesita más tiempo para ejecutarse. Por otro lado, lo que se busca es que se reconozcan las señales de tráfico y con este dataset se puede observar que en ninguna de las imágenes se ha detectado alguna señal.

Es por ello por lo que, aun siendo más eficaz el framework de Detectron2 debido a la gran exactitud de detección en el corto tiempo que lo realiza, YOLO permite utilizar varios datasets que se encuentran en repositorios de GitHub y que ya están especializados en la detección de señales de tráfico.

Así pues, se proseguirá con la búsqueda del mejor dataset junto al framework que permita detectar las señales de tráfico con la mayor exactitud posible, indagando en los repositorios que se encuentran en internet.

6.3. BÚSQUEDA DE DATASETS REFERIDOS A TRAFFIC SIGN DETECTION PARA SU IMPLEMENTACIÓN EN YOLO

Si no se restringe la búsqueda de datasets, existen infinidad de repositorios destinados a la detección de señales de tráfico a través de redes neuronales convolucionales. Por un lado, se pueden encontrar algunos basados en RetinaNet o MobileNet-SSD e implementados en Google Colab o en Raspberry Pi, como son:

- ptran1203/ traffic_sign_detection [57]

  El objetivo de este repositorio es realizar la detección de señales de tráfico utilizando RetinaNet. Por ello, ofrece todo un código que puede ser implementado en Google Colab y está destinado a llevar a cabo el entrenamiento de la red neuronal.

  Dentro del repositorio se ofrece un ejemplo visual de cómo el autor ha realizado el entrenamiento. Este alega que el dataset contiene imágenes muy pequeñas, por lo que, en primer lugar, procede a dividir la imagen en diferentes partes, tal y como se observa en la ilustración 53.

Ilustración 53: Muestra de la imagen separada en segmentos del dataset ptran1203/ traffic_sign_detection

A continuación, realiza la detección de cada una de las partes, como se puede observar en la ilustración 54, y después la detección en toda la imagen, tal y como se aprecia en la ilustración 55.

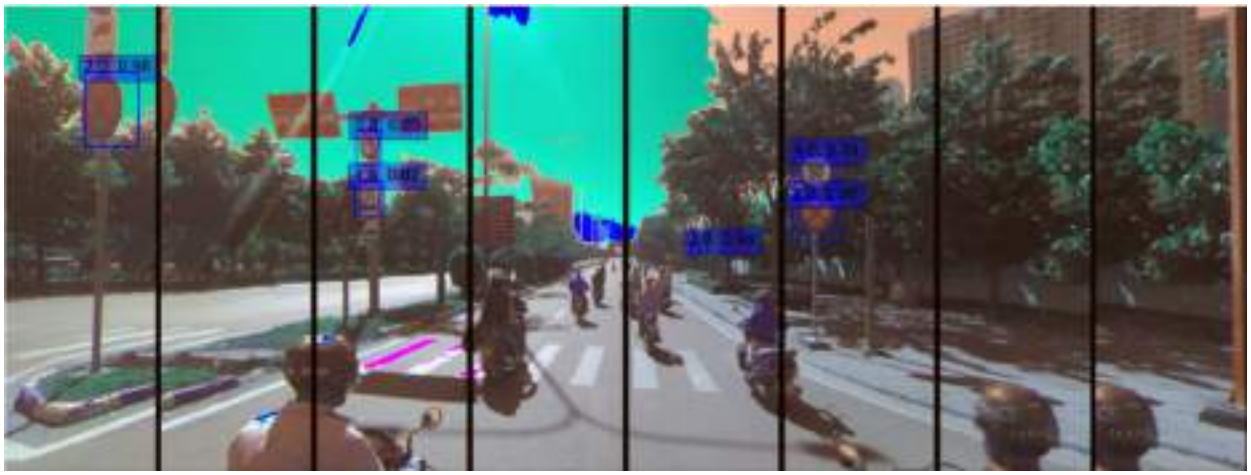


Ilustración 54: Muestra de la detección realizada por partes en el dataset ptran1203/ traffic_sign_detection

Ilustración 55: Muestra de la detección realizada en toda la imagen en el dataset ptran1203/ traffic_sign_detection

Finalmente, combina los resultados obtenidos utilizando la técnica de Non-Max Suppression, descrita anteriormente. Se puede observar la detección final en la ilustración 56.

Ilustración 56: Resultado final de la detección realizada en el dataset ptran1203/ traffic_sign_detection

- BrandonHanx/ Traffic-sign-detection [58]

  Se trata de un proyecto basado en Raspberry Pi donde se aplica el modelo de detección de peatones llevado a cabo con MobileNet-SSD.

  Dentro del repositorio se aprecian dos vídeos aportados por el autor donde se muestra un claro ejemplo de cómo realiza las detecciones. En la ilustración 57 se ofrece una captura de esos vídeos donde se observa la detección de varias personas y de una señal de limitación de velocidad.

Ilustración 57: Muestra de la detección de una señal de limitación y de varias personas en el dataset BrandonHanx/ Traffic-sign-detection

También, se pueden encontrar otros datasets que utilizan YOLO implementándolo en el simulador CARLA (Car Learning to Act), el cual resulta bastante útil porque simula una gran variedad de condiciones dadas en la conducción y repite situaciones de peligro de forma repetitiva para afianzar el aprendizaje de la red neuronal; además, ofrece acceso a su propio dataset y es bastante realista:

- guilherme-mendes/ CARLA-Traffic-Sign-Detection [59]

  Se trata de un modelo desarrollado para la detección de objetos de tráfico utilizando el simulador CARLA, YOLOv3 y Python.

  El repositorio ofrece varios ejemplos de cómo actúa el simulador, de entre ellos un mapa de las calles junto con la detección de una señal de tráfico y un vehículo, como se puede observar en la ilustración 58, y una detección de una señal de STOP, tal y como aparece en la ilustración 59.

Ilustración 58: Detección de una señal de tráfico y un vehículo por el simulador CARLA en el dataset guilherme-mendes/ CARLA-Traffic-Sign-Detection

Ilustración 59: Detección de una señal de STOP por el simulador CARLA en el dataset guillherme-mendes/ CARLA-Traffic-sign-detection

- martisaju/ CARLA-Speed-Traffic_Sign-Detection-Using-Yolo [60]

  Es una tesis de máster basada en cómo detectar a través del simulador CARLA y de YOLOv3 las señales de tráfico que indican la velocidad.

Además, aunque el marco de trabajo que se vaya a utilizar sea Darknet, se pueden encontrar varios repositorios basados en Darkflow, tales como:

- aarcosg/ traffic-sign-detection [61]

  Ofrece el código del artículo "Evaluation of deep neural networks for traffic sign detection systems" donde se analiza el uso de diferentes redes neuronales para dicho aspecto, entre ellas YOLOv2, implementándolas en TensorFlow a través de Darkflow.

  Dentro del repositorio se puede apreciar una imagen de ejemplo mostrando cómo realiza la detección de las señales de tráfico, tal y como se observar en la ilustración 60, pero sin especificar con qué red neuronal se ha llevado a cabo.

Ilustración 60: Muestra de la detección de señales de tráfico realizada por el dataset aarcosg/ traffic-sign-detection

- AmeyaWagh/ Traffic-sign-detection-YOLO [62]

  Se trata de la implementación de Darkflow para entrenar la red con el objetivo de poder llevar a cabo la detección y clasificación de señales de tráfico.

- near77/ Tiny-YOLO-voc-traffic-sign-detection [63]

  Este repositorio ofrece la detección de hasta 25 tipos de señales de tráfico en PC, NVIDIA TX2 o Raspberry Pi, gracias a que el código está destinado al entrenamiento de la red neuronal a través de Darkflow.

  En él se aprecia tres imágenes de ejemplo donde se muestra cómo de exactas son las detecciones. Una de ellas es la que aparece en la ilustración 61.

Ilustración 61: Muestra de la detección llevada a cabo con el dataset del repositorio near77/ Tiny-YOLO-voc-traffic-sign-detection

- dark-archerx/ Traffic-Signs-and-Object-Detection [64]

  Se trata de un proyecto que fue utilizado en el Smart India Hackaton 2018 (SIH 2018) y lleva a cabo la detección de todo lo relacionado con el tráfico a través de Darkflow.

Tras haber comprobado que existe gran variedad de repositorios, la mayoría de ellos son proyectos basados en Darknet. A continuación, se expondrán varios de ellos y, posteriormente, se hará hincapié en otros, los cuales se pondrán en funcionamiento y se comprobará su precisión.

- wyfsh/ YOLO-LISA [65]

  Ofrece el código para realizar la detección de señales de tráfico en tiempo real con YOLOv2. Ha sido entrenado con el dataset LISA-TS el cual contiene señales de tráfico propias de Estados Unidos.

- yogeshgajjar/ bosch-traffic-sign-detection-YOLOv3 [66]

  Se trata de un código donde se puede entrenar o llevar a cabo la detección de señales de tráfico con el dataset de Bosch utilizando YOLOv3-tiny, el cual ha sido entrenado en Jetson TX2.

- rechardchen123/ YOLOv4_traffic_sign_detection [67]

  Es un repositorio destinado a la detección de señales de tráfico mediante YOLOv4. Contiene el dataset TT100K, el cual hace referencia a las señales de circulación chinas y se puede tanto entrenar como comprobar que funciona correctamente.

- sdbidon/ Traffic-Sign-Detection-using-YOLOv3 [68]

  Ofrece un enlace a una carpeta donde contiene varios códigos para probar el modelo que facilitan junto con los vídeos de prueba o con tu propia cámara web; o, por el contrario, para realizar el entrenamiento con tu propio dataset.

  En el repositorio se puede apreciar una imagen de ejemplo de la detección de señales de información, tal y como se puede observar en la ilustración 62.

Ilustración 62: Muestra de la detección realizada con el dataset del repositorio sdbidon/ Traffic-Sign-Detection-using-YOLOv3

- aakashjhawar/ traffic-sign-detection [69]

  Este último ofrece un entrenamiento para la detección de señales de tráfico en YOLOv3, tanto para Windows como para Linux.

En su repositorio se puede observar una imagen de ejemplo, retratada en la ilustración 63, donde se muestra cómo realiza la detección en un dataset propio ya entrenado.

Ilustración 63: Muestra de la detección realizada por el dataset en el repositorio aakashjhawar/ traffic-sign-detection

Tal y como se ha mencionado al principio de este apartado, existen infinidad de repositorios destinados a la detección de señales de tráfico; por ello, se han destacado varios con el fin de ofrecer una amplia variedad de códigos y datasets.

A continuación, con el objetivo de comprobar cómo de exacta es la detección utilizando YOLO a través de Darknet cuando se refiere al reconocimiento de señales de tráfico, se ha proseguido a la puesta en marcha de cuatro repositorios diferentes y se han utilizado varias capturas realizadas a un vídeo propio y a otro extraído de internet [70] para su posterior comprobación:

1. **AlexeyAB/ darknet [71]**

Este repositorio, tal y como indica su nombre, ofrece la versión de Darknet tanto para Linux como para Windows o macOS. La implementación oficial de este marco de trabajo se encuentra en "pjreddie/ darknet", pero al haberse desarrollado YOLOv4 presenta varios errores con OpenCV, por lo que desde esa misma página se ofrece la posibilidad de implementar Darknet a través del repositorio de AlexeyAB.

Dentro de este podemos observar que está destinado a la implementación de YOLOv4, YOLOv3 o YOLOv2 a través de Darknet. Además, ofrece información sobre:

- Los requerimientos necesarios y cómo instalar las dependencias
- Cómo instalar YOLOv4 en otros frameworks
- Ofrece enlace a datasets distintos a los que ya contiene en el repositorio
- Muestra las mejoras llevadas a cabo
- Explica cómo compilarlo en Linux, Windows o macOS
- Explica cómo entrenarlo con múltiples GPUs
- Explica cómo entrenar tu propio dataset
- Explica cómo mejorar la detección de objetos

- Explica cómo realizar las cajas delimitadoras de los objetos y cómo crear los archivos con anotaciones
- Muestra cómo utilizar YOLO como una librería de DLL y SO

Como se puede observar, ofrece todo tipo de información sobre YOLO y su funcionamiento a través de Darknet, además de que presenta gran cantidad de archivos "cfg", "data" y sus respectivos "weights" que serán los que permitirán realizar las detecciones sin necesidad de entrenar la red neuronal.

En el apartado de la implementación de YOLO en Darknet, se ha comprobado la velocidad de actuación de este a la hora de realizar la detección de objetos mediante el dataset de COCO ubicado en este repositorio; así pues, cabe destacar que todas las detecciones realizadas tanto en este como en los siguientes repositorios están basadas en la misma implementación de YOLO, la cual es a la que se está refiriendo durante todo el apartado.

Por lo tanto, el primer paso a realizar es observar qué clase de datasets ofrece el repositorio, percibiendo así que existen seis opciones diferentes, de las cuales dos contienen un archivo con extensión .py y otras dos un archivo con extensión .sh, los cuales se encuentran en el mismo repositorio y otorgan las etiquetas para las imágenes de los datasets. Los dos restantes tan solo ofrecen un enlace a sus propios datasets.

De entre los que se encuentran disponibles en este repositorio, llama la atención el dataset de OpenImages, ya que ofrece la detección de 601 clases de objetos diferentes y, entre ellos, se pueden destacar las señales de tráfico, la señal de STOP, los semáforos, los vehículos y el cuerpo humano. Además, contiene sus propios archivos "cfg" y "data", y da la posibilidad de descargar sus "weights".

Así pues, teniendo todo lo necesario para la realización de la detección de señales de tráfico, se prosigue a comprobar con qué exactitud realiza su reconocimiento.

El comando general utilizado para realizar la detección de las imágenes es:

```
$ ./ darknet detector test cfg/openimages.data cfg/yolov3-openimages.cfg yolov3-openimages.weights data/imagen.jpg
```

- **Imagen 1:** en la ilustración 64 se puede observar que realiza la detección de tres vehículos y una rueda, pero no detecta ninguna señal de tráfico.

  El comando utilizado para realizar la detección de la imagen 1 con el dataset de OpenImages es:

```
$ ./darknet detector test cfg/openimages.data cfg/yolov3-openimages.cfg yolov3-openimages.weights data/imagen1.jpg
```

Ilustración 64: Detección imagen 1 utilizando dataset OpenImages en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 1 es:

- Utilizando tan solo la GPU: 0,499971s

❖ **Imagen 2:** en la ilustración 65 se puede observar que se realiza la detección de un vehículo y de una planta, aun siendo un árbol, pero no se reconocen las dos señales de tráfico.

El comando utilizado para realizar la detección de la imagen 2 con el dataset de OpenImages es:

```
$ ./darknet detector test cfg/openimages.data cfg/yolov3-openimages.cfg yolov3-openimages.weights data/imagen2.jpg
```

Ilustración 65: Detección imagen 2 utilizando dataset OpenImages en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 2 es:

- Utilizando tan solo la GPU: 0,480042s

❖ **Imagen 3:** en la ilustración 66 se puede observar la detección de dos vehículos y una rueda, sin llegar a reconocer las señales de tráfico que aparecen.

El comando utilizado para realizar la detección de la imagen 3 con el dataset de OpenImages es:

```
$ . /darknet detector test cfg/openimages.data cfg/yolov3-openimages.cfg yolov3-openimages.weights data/imagen3.jpg
```

Ilustración 66: Detección imagen 3 utilizando dataset OpenImages en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 3 es:

- Utilizando tan solo la GPU: 0,45183s

Tal y como se puede observar, aun trabajando a tiempos ínfimos, no llega a detectar todos los objetos que aparecen en las imágenes y, en aquellas en las que sí reconoce algunos de ellos, se puede comprobar que lo hace con unos porcentajes demasiado bajos. Además, aunque ofrezca la posibilidad de reconocer las señales de tráfico y los semáforos, se puede comprobar que no lo realiza.

Por lo tanto, aun reconociendo varios objetos, que podrían ser de interés en trabajos futuros, con un tiempo de respuesta mínimo, no permite alcanzar el objetivo de este trabajo, el cual es la detección de las señales de tráfico. Es por ello por lo que se buscará otro dataset diferente.

2. fredotran/ traffic-sign-detector-yolov4 [72]

Tras los resultados observados en la detección anterior, se procede a la búsqueda de otro dataset cuyo repositorio contenga todos los archivos necesarios para tan solo realizar la detección de señales de tráfico. Por ello, se encuentra este, el cual ofrece el entrenamiento y la detección de 4 clases específicas de señales de tráfico, que son: señales de límite de velocidad, señales de STOP, semáforos y señales de señalización de paso de peatones. También, ofrece la posibilidad de implementarlo en Google Colab.

Se puede observar que contiene varios archivos con extensión .py que facilitan la obtención de los "weights" y las imágenes con sus propias etiquetas, entre otras acciones. Además, ofrece el archivo "cfg" mientras que el archivo "data" se debe crear, lo cual no es gran complicación debido a que este tan solo necesita contener el número de clase de objetos que se van a detectar y la localización del archivo de texto donde se encuentran los

nombres de cada objeto, dispuestos en el mismo orden en el que se realizó el entrenamiento.

Una vez obtenidos todos los archivos necesarios, se procede a la detección en cada una de las imágenes:

- **Imagen 1:** tal y como se puede apreciar en la ilustración 67, se realiza la detección de las dos señales de STOP que aparecen, ya que ha sido entrenado para ello, y una señal de límite de velocidad que asocia a un ventilador de aire acondicionado.

  El comando utilizado para realizar la detección de la imagen 1 con el dataset de fredotran es:

```
$ ./darknet detector test data/obj1.data cfg/yolov4-rds.cfg yolov4-rds_best_2000.weights data/imagen1.jpg
```

Ilustración 67: Detección imagen 1 utilizando repositorio fredotran en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 1 es:

- Utilizando tan solo la GPU: 0,392135s

- **Imagen 2:** en la ilustración 68 se puede observar que realiza la detección de la señal de límite de velocidad para la que ha sido entrenado, obviando así la señal de peligro.

  El comando utilizado para realizar la detección de la imagen 2 con el dataset de fredotran es:

```
$ ./darknet detector test data/obj1.data cfg/yolov4-rds.cfg yolov4-
rds_best_2000.weights data/imagen2.jpg
```

Ilustración 68: Detección imagen 2 utilizando repositorio fredotran en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 2 es:

- Utilizando tan solo la GPU: 0,38958s

❖ **Imagen 3:** en la ilustración 69 se puede observar que tan solo detecta la señal de información de paso de peatones, ya es la única para la que ha sido entrenada de entre las que se encuentran en la imagen.

El comando utilizado para realizar la detección de la imagen 3 con el dataset de fredotran es:

```
$ ./darknet detector test data/obj1.data cfg/yolov4-rds.cfg yolov4-
rds_best_2000.weights data/imagen3.jpg
```

Ilustración 69: Detección imagen 3 utilizando repositorio fredotran en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 3 es:

- Utilizando tan solo la GPU: 0,403037s

En el caso de este dataset se puede comprobar que sí detecta tres de los cuatro tipos de objetos para los que ha sido entrenado y con unos porcentajes bastante aceptables, además de que lo lleva a cabo en tiempos inferiores al anterior; pero, aun así, lo que se busca es realizar la detección de todos los tipos de señales de tráfico, por lo que se seguirá buscando otro dataset que sea más exacto.

3. sichkar-valentyn/ Traffic-Signs-for-training-YOLO-v3-Detector [73]

El presente repositorio contiene un enlace a la inscripción de un curso donde se explica cómo entrenar tu propio dataset a través de YOLOv3, pero si no se considera necesario para poder realizar el entrenamiento, también facilita la posibilidad de descargar, a través de Kaggle, los archivos utilizados en dicho curso.

Por ello, ofrece los archivos "cfg" tanto para entrenar la red como para comprobar su correcto funcionamiento, además de las imágenes junto con sus etiquetas, el archivo de texto junto con los nombres de cada tipo de objeto y el archivo "data". Es por eso, por lo que para poder comprobar cómo funciona este dataset se debe entrenar la red, ya que no ofrecen los "weights" si no te inscribes a dicho curso.

El objetivo de este repositorio es llegar a detectar las señales que se clasifican en 4 tipos: prohibición, obligatoriedad, peligro y otros. Por ello, el comando utilizado para realizar el entrenamiento es:

```
$ ./darknet detector train data/ts-data.data
cfg/yolov3_ts_train.cfg yolov4.conv.137
```

donde “detector train” indica que se va a realizar el entrenamiento, también aparecen los archivos “cfg” y “data”, y además, se facilita uno de los varios pesos que se utilizan para llevar a cabo los entrenamientos en Darknet.

Tras unas 10 horas de entrenamiento, se prosigue a comprobar cómo de exacta es la detección en las diferentes imágenes:

- **Imagen 1:** en la ilustración 70 se puede observar que tan solo detecta la señal de prohibición mientras que las señales de STOP ni las reconoce.

  El comando utilizado para realizar la detección de la imagen 1 con el dataset de sichkar-valentyn es:

```
$ ./darknet detector test data/ts-data.data cfg/yolov3_ts_test.cfg
yolov3_ts_train_best.weights data/imagen1.jpg
```

Ilustración 70: Detección imagen 1 utilizando repositorio sichkar-valentyn en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 1 es:

- Utilizando tan solo la GPU: 0,384535s

- **Imagen 2:** en la ilustración 71 se observa la detección de la señal de prohibición sin llegar a reconocer la de peligro.

  El comando utilizado para realizar la detección en la imagen 2 con el dataset de sichkar-valentyn es:

```
$ ./darknet detector test data/ts-data.data cfg/yolov3_ts_test.cfg
yolov3_ts_train_best.weights data/imagen2.jpg
```

Ilustración 71: Detección imagen 2 utilizando repositorio sichkar-valentyn en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 2 es:

- Utilizando tan solo la GPU: 0,413296s

- **Imagen 3:** en la ilustración 72 se observa la detección de las señales de Ceda el Paso y la señal de obligación, dejando sin reconocer la señal de información del paso de peatones y otras que se encuentran en el fondo.

  El comando utilizado para realizar la detección de la imagen 3 con el dataset de sichkar-valentyn es:

```
$ ./darknet detector test data/ts-data.data cfg/yolov3_ts_test.cfg
yolov3_ts_train_best.weights data/imagen3.jpg
```

Ilustración 72: Detección imagen 3 utilizando repositorio sichkar-valentyn en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 3 es:

- Utilizando tan solo la GPU: 0,388575s

Se puede comprobar que este realiza las detecciones de las señales de tráfico con unos porcentajes muy elevados, siendo casi ideales, y en unos tiempos de respuesta ínfimos. Aun así, no llega a reconocer todas las señales que aparecen en las imágenes, por lo que se procederá a buscar otro dataset que sea un poco más preciso.

4. angeligareta/ SaferAuto [74]

Este repositorio contiene una interfaz gráfica de usuario (GUI) elaborada para la detección y localización de elementos de la calzada a través de un vídeo en tiempo real utilizando YOLOv3. De entre todos ellos, el modelo está entrenado para detectar semáforos, vehículos y algunos tipos de señales de tráfico.

El objetivo de este repositorio es crear una interfaz que permita al usuario cargar los archivos que se usarían en Darknet (archivos de configuración, nombres de los objetos, pesos y archivos que contengan fotos o vídeos) permitiendo así evitar tener que escribir los comandos necesarios.

Por otro lado, si se quiere seguir utilizando Darknet, ofrece los archivos "data", "config" y los pesos para poder llevarlo a cabo independientemente. No indica qué tipo de señales son las que detecta, por lo que se deberá poner en funcionamiento y comprobar cuáles son, cómo de exacta es la detección que realiza y en cuál es su tiempo de ejecución.

Cabe tener en cuenta que, como se ha indicado al principio, los archivos están configurados para realizar detecciones en vídeos en tiempo real, por lo que a la hora de realizar las

detecciones en las imágenes muestra un error que restringe la aparición de estas con sus cajas delimitadoras, pero sí permite observar el tiempo de ejecución y la exactitud de las detecciones realizadas. Es por ello por lo que se adjuntará la información obtenida del comando y una captura realizada lo más exacta posible en el momento en el que se capturó la imagen previa.

El comando necesario para realizar las detecciones en los dos vídeos es:

```
$ ./darknet detector demo prueba/ere.data prueba/yolov3-tiny.cfg
yolov3-tiny_best.weights video2.MOV
```

```
$ ./darknet detector demo prueba/ere.data prueba/yolov3-tiny.cfg
yolov3-tiny_best.weights señales.mp4
```

donde “detector demo” indica que se va a realizar la detección en un vídeo.

Mientras que el comando necesario para realizar las detecciones en las imágenes es:

- **Imagen 1:** en la ilustración 73 se puede observar que tan solo se detectan las señales de STOP, obviando de esta forma la señal de prohibición.

  El comando utilizado para realizar la detección de la imagen 1 con el dataset de angeligareta es:

```
$ ./darknet detector test prueba/ere.data prueba/yolov3-tiny.cfg
yolov3-tiny_best.weights data/imagen1.jpg
```

Ilustración 73: Detección imagen 1 utilizando repositorio angeligareta en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 1 es:

- Utilizando tan solo la GPU: 0,356554s

❖ **Imagen 2:** en la ilustración 74 se puede observar que se realiza la detección tanto de la señal de prohibición como de la señal de peligro.

El comando utilizado para realizar la detección de la imagen 2 con el repositorio de angeligareta es:

```
$ ./darknet detector test prueba/ere.data prueba/yolov3-tiny.cfg
yolov3-tiny_best.weights data/imagen2.jpg
```

Ilustración 74: Detección imagen 2 utilizando repositorio angeligareta en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 2 es:

- Utilizando tan solo la GPU: 0,35842s

❖ **Imagen 3:** en la ilustración 75 se puede observar la detección de las señales de Ceda el Paso, de obligación y de información, obviando la primera señal de obligación que aparece.

El comando utilizado para realizar la detección de la imagen 3 con el dataset de angeligareta es:

```
$ ./darknet detector test prueba/ere.data prueba/yolov3-tiny.cfg
yolov3-tiny_best.weights data/imagen3.jpg
```

Ilustración 75: Detección imagen 3 utilizando repositorio angeligareta en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 3 es:

- Utilizando tan solo la GPU: 0,356554s

En este caso se puede observar que los tiempos de ejecución son menores, en comparación a los tres repositorios anteriores, y que realiza la detección de mayor número de señales de tráfico con porcentajes de exactitud bastante elevados.

En vista de los resultados, se han extraído dos capturas más para afirmar la idoneidad de este repositorio.

❖ **Imagen 4:** en la ilustración 76 se observa la detección de tres señales de peligro y una de obligación, dejando sin detectar una señal de prohibición.

El comando utilizado para realizar la detección de la imagen 4 con el dataset de angeligareta es:

```
$ ./darknet detector test prueba/ere.data prueba/yolov3-tiny.cfg
yolov3-tiny_best.weights data/imagen4.jpg
```

Ilustración 76: Detección imagen 4 utilizando repositorio angeligareta en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 4 es:

- Utilizando tan solo la GPU: 0,356085s

❖ **Imagen 5:** en la ilustración 77 se observa la detección de dos señales Ceda el Paso, una de obligación y otra de información de un paso de peatones, dejando sin detectar una señal de obligación que se encuentran más lejos.

El comando utilizado para realizar la detección de la imagen 5 con el dataset de angeligareta es:

```
$ ./darknet detector test prueba/ere.data prueba/yolov3-tiny.cfg
yolov3-tiny_best.weights data/imagen5.jpg
```

Ilustración 77: Detección imagen 5 utilizando repositorio angeligareta en Darknet

El tiempo de ejecución empleado para realizar la detección de la imagen 5 es:

- Utilizando tan solo la GPU: 0,348922s

Así pues, se podría concluir afirmando que los archivos de este repositorio serían los idóneos para realizar las detecciones de las señales de tráfico en la conducción autónoma, pero, aun así, a continuación, se efectuarán unas tablas que aporten todos los resultados obtenidos y permita de esta forma alcanzar una conclusión válida y respaldada.

6.3.1. MUESTRA DE RESULTADOS OBTENIDOS

Tras haber realizado las detecciones de las tres imágenes en los cuatro repositorios que contenían todos los archivos necesarios para llevar a cabo el reconocimiento y la detección de las señales de tráfico, se procede a clasificar los resultados en base a la exactitud y el tiempo de ejecución requerido en cada una de ellas.

- De la imagen 1 se obtiene la siguiente información:

SEÑALES QUE SE DEBERÍAN DETECTAR

	AlexeyAB	**fredotran**	**sichkar-valentyn**	**angeligareta**
STOP	0%	96%	0%	98%
STOP	0%	95%	0%	64%
Prohibición	0%	0%	97%	0%

Tabla 12: Comparación exactitud de detección en la imagen 1 entre los cuatro repositorios

TIEMPO DE EJECUCIÓN	
AlexeyAB	0,499 s
fredotran	0,392 s
sichkar-valentyn	0,384 s
angeligareta	0,358 s

Tabla 13: Comparación tiempo empleado por cada repositorio en la imagen 1

Se puede observar que ninguno de los cuatro repositorios llega a detectar las tres señales principales, destacando que el repositorio de AlexeyAB no reconoce ninguna de ellas y emplea mayor tiempo que los otros tres. Por otro lado, se advierte que dentro de los tipos de señales que son capaces de detectar cada uno de los otros repositorios, el de angeligareta obtiene un porcentaje bajo para la segunda señal de STOP aun siendo el que menor tiempo necesita para realizar la detección.

- De la imagen 2 se obtiene la siguiente información:

SEÑALES QUE SE DEBERÍAN DETECTAR				
	AlexeyAB	**fredotran**	**sichkar-valentyn**	**angeligareta**
Peligro	0%	0%	0%	84%
Prohibición	0%	88%	99%	99%

Tabla 14: Comparación exactitud de detección en la imagen 2 entre los cuatro repositorios

TIEMPO DE EJECUCIÓN	
AlexeyAB	0,48 s
fredotran	0,389 s
sichkar-valentyn	0,413 s
angeligareta	0,358 s

Tabla 15: Comparación tiempo empleado por cada repositorio en la imagen 2

En este caso se puede observar que el repositorio de AlexeyAB sigue sin detectar ninguna de las dos señales principales, mientras que el de angeligareta sí las detecta y con porcentajes de exactitud muy altos. Además, aunque los otros dos repositorios lleguen a reconocer una de las dos señales con porcentajes más que aceptables, sigue siendo el repositorio de angeligareta el que emplea menos tiempo en su ejecución.

- De la imagen 3 se obtiene la siguiente información:

SEÑALES QUE SE DEBERÍAN DETECTAR

	AlexeyAB	**fredotran**	**sichkar-valentyn**	**angeligareta**
Información	0%	77%	0%	98%
Ceda el paso	0%	0%	99%	97%
Obligación	0%	0%	99%	77%

Tabla 16: Comparación exactitud de detección en la imagen 3 entre los cuatro repositorios

TIEMPO DE EJECUCIÓN

AlexeyAB	0,452 s
fredotran	0,403 s
sichkar-valentyn	0,388 s
angeligareta	0,356 s

Tabla 17: Comparación tiempo empleado por cada repositorio en la imagen 3

Como se puede observar, el repositorio de AlexeyAB tampoco reconoce ninguna de las señales principales que aparecen en la imagen, además de que emplea el mayor tiempo para su ejecución. Respecto a los otros tres repositorios, se puede advertir que el de fredotran reconoce una de las señales para las que ha sido entrenado con un valor de exactitud aceptable, pero emplea demasiado tiempo en ello. Por otro lado, el repositorio de sichkar-valentyn reconoce dos de las señales para las que ha sido entrenado con una exactitud prácticamente ideal y emplea un tiempo aceptable en ello. Finalmente, se puede observar que el repositorio de angeligareta detecta las tres señales principales y con unos valores de precisión bastante elevados, además de que es el que menor tiempo emplea para su ejecución.

Así pues, luego de haber comparado todos los resultados, se puede afirmar que el repositorio "angeligareta/ SaferAuto" contiene una red neuronal entrenada para la detección de señales de tráfico bastante precisa. Además, su aplicación en YOLO a través de Darknet ofrece unos tiempos de ejecución ínfimos, lo cual permite otorgarle la idoneidad para su implementación en la conducción autónoma.

7. DISCUSIÓN Y CONCLUSIÓN DEL TRABAJO REALIZADO

A la luz de los resultados, se puede determinar que la mejor opción para llevar a cabo el reconocimiento y la detección de las señales de tráfico mediante la Inteligencia Artificial es utilizando el repositorio "angeligareta/ SaferAuto" a través de Darknet. Aun así, se resalta que existen infinidad de posibilidades para realizar cualquier detección de objetos, ya que se ha podido comprobar que tanto Darkflow como Detectron2 ofrecían buenos resultados. Además, se debe destacar que este último otorgaba gran precisión a la hora de detectar los objetos que ofrecía el dataset de COCO, convirtiéndolo así en la mejor opción de entre los tres marcos de trabajo expuestos en este proyecto.

También, se debe recordar que existen otros tipos de redes neuronales convolucionales diferentes a YOLO, Faster R-CNN y Mask R-CNN que realizan la misma función y, tal vez, con los mismos o incluso mejores resultados. Aun así, las utilizadas en este trabajo son las más conocidas por su eficacia y su facilidad de ejecución y es por ello por lo que fueron seleccionadas. Siguiendo en la misma línea, también se pueden encontrar otros tipos de frameworks, tal y como se ha expuesto en el marco teórico, que realicen la misma función que se ha llevado a cabo en este trabajo y que, tal vez, se ajusten mejor a otro tipo de detecciones.

En cuanto a las limitaciones de búsqueda, cabe resaltar que no se ha apreciado gran dificultad a la hora de realizar la indagación de información debido a que se encuentran infinidad de repositorios y datasets en internet; además, aparecen muchos blogs y tutoriales que explican cómo instalar las dependencias y los frameworks necesarios.

De todas formas, se podría destacar de forma negativa que, aun habiendo gran variedad de repositorios, estos no siempre ofrecen todos los archivos necesarios para realizar la detección sin tener que entrenar la red neuronal, por lo que, en este aspecto, se puede apreciar un pequeño conflicto. Aun así, gracias a la gran comunidad que existe en este campo de investigación, se puede resolver y llegar a alcanzar de forma factible el objetivo que se desee.

Tal vez, aun habiendo podido realizar de forma bastante precisa la detección de las señales de tráfico utilizando una red neuronal artificial ya entrenada, se debería en un futuro llevar a cabo el entrenamiento de esta red neuronal junto con un dataset propio para conseguir unos resultados prácticamente ideales y así llegar a asegurar una excelente conducción autónoma.

De este modo, se da respuesta a la cuestión que se plantea en este Trabajo de Fin de Grado ya que tras haber realizado una búsqueda exhaustiva de frameworks y habiendo escogido el mejor de ellos, se ha podido poner en funcionamiento, junto con uno de los repositorios que fue considerado el mejor para esta cuestión, y se ha conseguido comprobar que realiza de forma precisa el reconocimiento y la detección de las señales de tráfico, otorgándole así la idoneidad para su aplicación en un vehículo autónomo.

8. BIBLIOGRAFÍA

[1] López JM. La historia y origen del coche autónomo en el siglo XX [Internet]. Hipertextual.com. 2020 [citado 2021 Jun 16]. Disponible en: https://hipertextual.com/2020/08/origen-historia-vehiculo-autonomo

[2] Esta es la historia del coche autónomo, y ojo porque no es tan nuevo como lo pintan… [Internet]. Autonocion.com. 2018 [citado 2021 Jun 16]. Disponible en: https://www.autonocion.com/historia-coche-autonomo/

[3] Broggi A, Zelinsky A, Parent M, Thorpe CE. Intelligent Vehicles. In: Springer Handbook of Robotics. Berlin, Heidelberg: Springer Berlin Heidelberg; 2008. p. 1175–98.

[4] Rivera N. ¿Qué sensores tiene un coche autónomo y cómo funcionan? [Internet]. Hipertextual.com. 2017 [citado 2021 Jun 16]. Disponible en: https://hipertextual.com/2017/06/sensores-coches-autonomos

[5] Chai Z, Nie T, Becker J. Technologies for Autonomous Driving. In: Autonomous Driving Changes the Future. Singapore: Springer Singapore; 2021. p. 17–61.

[6] Fisher RB, Konolige K. Range Sensors. In: Springer Handbook of Robotics. Berlin, Heidelberg: Springer Berlin Heidelberg; 2008. p. 521–42.

[7] Ibáñez. Qué es un LIDAR, y cómo funciona el sensor más caro de los coches autónomos [Internet]. Motorpasion.com. Motorpasión; 2017 [citado 2021 Jun 16]. Disponible en: https://www.motorpasion.com/tecnologia/que-es-un-lidar-y-como-funciona-el-sistema-de-medicion-y-deteccion-de-objetos-mediante-laser

[8] Chai Z, Nie T, Becker J. Applications of autonomous driving that you should know. In: Autonomous Driving Changes the Future. Singapore: Springer Singapore; 2021. p. 63–80.

[9] Joshi AV. Introduction to AI and ML. In: Machine Learning and Artificial Intelligence. Cham: Springer International Publishing; 2020. p. 3–7.

[10] Maravilloso “un Logro, Profundidad y una amplitud increíbles CU. Un Enfoque Moderno [Internet]. Wordpress.com. [citado 2021 Jun 19]. Disponible en: https://luismejias21.files.wordpress.com/2017/09/inteligencia-artificial-un-enfoque-moderno-stuart-j-russell.pdf

[11] Rouhiainen L. Una guía para estar en el bando ganador cuando la tecnología transforme el mundo por completo: anticípate a lo que viene [Internet]. Planetadelibros.com. [citado 2021 Jun 19]. Disponible en: https://www.planetadelibros.com/libros_contenido_extra/40/39307_Inteligencia_artificial.pdf

[12] Joshi AV. Essential concepts in artificial intelligence and machine learning. In: Machine Learning and Artificial Intelligence. Cham: Springer International Publishing; 2020. p. 9–20.

[13] Joshi AV. Deep Learning. In: Machine Learning and Artificial Intelligence. Cham: Springer International Publishing; 2020. p. 117–26.

[14] Kim K, Aminanto ME, Tanuwidjaja HC. Deep learning-based IDSs. In: SpringerBriefs on Cyber Security Systems and Networks. Singapore: Springer Singapore; 2018. p. 35–45.

[15] Daniilidis K, Eklundh J-O. 3-D Vision and Recognition. In: Springer Handbook of Robotics. Berlin, Heidelberg: Springer Berlin Heidelberg; 2008. p. 543–62.

[16] Quesada FJG, Graciani MAF, Bonal MTL, Díaz-Mata MA. Aprendizaje con redes neuronales artificiales. Ensayos. 1994;(9):169–80.

[17] Object Detection Guide [Internet]. Fritz.ai. [citado 2021 Jun 16]. Disponible en: https://www.fritz.ai/object-detection/

[18] Ganesh P. Object Detection: Simplified [Internet]. Towards Data Science. 2019 [citado 2021 Jun 16]. Disponible en: https://towardsdatascience.com/object-detection-simplified-e07aa3830954

[19] Na. Modelos de detección de objetos [Internet]. Aprendemachinelearning.com. 2020 [citado 2021 Jun 16]. Disponible en: https://www.aprendemachinelearning.com/modelos-de-deteccion-de-objetos/

[20] a. E. Detección de objetos con YOLO: implementaciones y como usarlas [Internet]. Medium. 2018 [citado 2021 Jun 16]. Disponible en: https://medium.com/@enriqueav/detecci%C3%B3n-de-objetos-con-yolo-implementaciones-y-como-usarlas-c73ca2489246

[21] TensorFlow [Internet]. Tensorflow.org. [citado 2021 Jun 16]. Disponible en: https://www.tensorflow.org/?hl=es-419

[22] Caffe [Internet]. Berkeleyvision.org. [citado 2021 Jun 16]. Available from: https://caffe.berkeleyvision.org/

[23] PyTorch [Internet]. Pytorch.org. [citado 2021 Jun 16]. Disponible en: https://pytorch.org/

[24] Apache MXNet [Internet]. Apache.org. [citado 2021 Jun 16]. Disponible en: https://mxnet.apache.org/versions/1.8.0/

[25] Machine learning project at the university of waikato in New Zealand [Internet]. Waikato.ac.nz. [citado 2021 Jun 16]. Disponible en: http://old-www.cms.waikato.ac.nz/~ml/

[26] Deeplearning4j [Internet]. Deeplearning4j.org. [citado 2021 Jun 16]. Disponible en: https://deeplearning4j.org/

[27] chrisbasoglu. The Microsoft cognitive toolkit [Internet]. Microsoft.com. [citado 2021 Jun 16]. Disponible en: https://docs.microsoft.com/en-us/cognitive-toolkit/

[28] DeepDetect [Internet]. Deepdetect.com. [citado 2021 Jun 16]. Disponible en: https://www.deepdetect.com/

[29] Keras Team. Keras: the Python deep learning API [Internet]. Keras.io. [citado 2021 Jun 16]. Disponible en: https://keras.io/

[30] ONNX [Internet]. Onnx.ai. [citado 2021 Jun 16]. Disponible en: https://onnx.ai/

[31] ONNX Runtime [Internet]. Onnxruntime.ai. [citado 2021 Jun 19]. Disponible en: https://www.onnxruntime.ai/

[32] COMPUTATIONAL INTELLIGENCE AND DATA SCIENCE. INTERNATIONAL CONFERENCE. 2018. (ICCIDS 2018) (3 PARTS) [Internet]. Proceedings.com. [citado 2021 Jun 16]. Disponible en: http://www.proceedings.com/39803.html

[33] Redmon J. Darknet: Open Source Neural Networks in C [Internet]. Pjreddie.com. [citado 2021 Jun 16]. Disponible en: https://pjreddie.com/darknet/

[34] Trieu. darkflow. Disponible en: https://github.com/thtrieu/darkflow

[35] Sharma P. Image segmentation [Internet]. Analyticsvidhya.com. 2019 [citado 2021 Jun 16]. Disponible en: https://www.analyticsvidhya.com/blog/2019/04/introduction-image-segmentation-techniques-python/?utm_source=blog&utm_medium=computer-vision-implementing-mask-r-cnn-image-segmentation

[36] Analyticsvidhya.com. [citado 2021 Jun 16]. Disponible en: https://www.analyticsvidhya.com/blog/2019/07/computer-vision-implementing-mask-r-cnn-image-segmentation/

[37] Prasad S. What is image segmentation or segmentation in image processing? [Internet]. Analytixlabs.co.in. 2020 [citado 2021 Jun 16]. Disponible en: https://www.analytixlabs.co.in/blog/what-is-image-segmentation/

[38] Mwiti D. Image segmentation in 2021: Architectures, losses, datasets, and frameworks [Internet]. Neptune.ai. 2020 [citado 2021 Jun 16]. Disponible en: https://neptune.ai/blog/image-segmentation-in-2020

[39] Khandelwal R. Computer vision: Instance segmentation with Mask R-CNN [Internet]. Towards Data Science. 2019 [citado 2021 Jun 16]. Disponible en: https://towardsdatascience.com/computer-vision-instance-segmentation-with-mask-r-cnn-7983502fcad1

[40] Detectron2: A PyTorch-based modular object detection library [Internet]. Facebook.com. [citado 2021 Jun 16]. Disponible en: https://ai.facebook.com/blog/-detectron2-a-pytorch-based-modular-object-detection-library-/

[41] COCO - common objects in context [Internet]. Cocodataset.org. [citado 2021 Jun 16]. Disponible en: https://cocodataset.org/

[42] Papers with Code - PASCAL VOC Dataset [Internet]. Paperswithcode.com. [citado 2021 Jun 16]. Disponible en: https://paperswithcode.com/dataset/pascal-voc

[43] Cityscapes dataset – semantic understanding of urban street scenes [Internet]. Cityscapes-dataset.com. [citado 2021 Jun 16]. Disponible en: https://www.cityscapes-dataset.com/

[44] Brostow GJ. Object Recognition in Video Dataset [Internet]. Cam.ac.uk. [citado 2021 Jun 16]. Disponible en: http://mi.eng.cam.ac.uk/research/projects/VideoRec/CamVid/

[45] Open Dataset – Waymo [Internet]. Waymo.com. [citado 2021 Jun 16]. Disponible en: https://waymo.com/open/

[46] Bdd100k.com. [citado 2021 Jun 16]. Disponible en: https://www.bdd100k.com/

[47] Welcome to fastai [Internet]. Fast.ai. [citado 2021 Jun 16]. Disponible en: https://docs.fast.ai/

[48] Fexa A. Sefexa Image Segmentation Tool [Internet]. Fexovi.com. [citado 2021 Jun 16]. Disponible en: http://www.fexovi.com/sefexa.html

[49] DeepMask [Internet]. Paperswithcode.com. [citado 2021 Jun 16]. Disponible en: https://paperswithcode.com/method/deepmask

[50] Zagoruyko S, Lerer A, Lin T-Y, Pinheiro PO, Gross S, Chintala S, et al. A MultiPath network for object detection [Internet]. arXiv [cs.CV]. 2016. Disponible en: http://arxiv.org/abs/1604.02135

[51] Home - OpenCV [Internet]. Opencv.org. 2021 [citado 2021 Jun 16]. Disponible en: https://opencv.org/

[52] Müller D, Kramer F. MIScnn: A framework for medical image segmentation with convolutional neural networks and deep learning [Internet]. arXiv [eess.IV]. 2019. Disponible en: http://arxiv.org/abs/1910.09308

[53] Fritz AI [Internet]. Fritz.ai. [citado 2021 Jun 16]. Disponible en: https://www.fritz.ai/

[54] Trieu. darkflow. Disponible en: https://github.com/thtrieu/darkflow

[55] Redmon J. darknet. Disponible en: https://github.com/pjreddie/darknet

[56] detectron2. Disponible en: https://github.com/facebookresearch/detectron2

[57] Tran P. Traffic_sign_detection. Disponible en: https://github.com/ptran1203/traffic_sign_detection

[58] Han X. Traffic-sign-detection. Disponible en: https://github.com/BrandonHanx/Traffic-sign-detection

[59] Mendes G. CARLA-traffic-Sign-detection. Disponible en: https://github.com/guilherme-mendes/CARLA-Traffic-Sign-Detection

[60] Juanola MS. CARLA-Speed-Traffic-Sign-detection-using-Yolo. Disponible en: https://github.com/martisaju/CARLA-Speed-Traffic-Sign-Detection-Using-Yolo

[61] Arcos Á. Traffic-sign-detection. Disponible en: https://github.com/aarcosg/traffic-sign-detection

[62] Wagh A. Traffic_sign_detection_YOLO. Disponible en: https://github.com/AmeyaWagh/Traffic_sign_detection_YOLO

[63] Near. Tiny-YOLO-voc-traffic-sign-detection. Disponible en: https://github.com/near77/Tiny-YOLO-voc-traffic-sign-detection

[64] Shukla A. Traffic-signs-and-object-detection. Disponible en: https://github.com/dark-archerx/Traffic-Signs-and-Object-Detection

[65] Wang Y. YOLO-LISA. Disponible en: https://github.com/wyfsh/YOLO-LISA

[66] Gajjar Y. Bosch-traffic-sign-detection-YOLOv3. Disponible en: https://github.com/yogeshgajjar/bosch-traffic-sign-detection-YOLOv3

[67] CHEN(Richard) X. YOLOv4_traffic_sign_detection. Disponible en: https://github.com/rechardchen123/YOLOv4_traffic_sign_detection

[68] Das S. Traffic-Sign-Detection-using-YOLOv3. Disponible en: https://github.com/sdbibon/Traffic-Sign-Detection-using-YOLOv3

[69] Jhawar A. Traffic-sign-detection. Disponible en: https://github.com/aakashjhawar/traffic-sign-detection

[70] Vídeo curso autoescuela: Señales de reglamentacion. Prioridad [Internet]. Youtube.com. Youtube; [citado 2021 Jun 19]. Disponible en: https://www.youtube.com/watch?v=KsZ6CGAePlI&t=18s

[71] Alexey. darknet. Disponible en: https://github.com/AlexeyAB/darknet

[72] Fredotran. traffic-sign-detector-yolov4. Disponible en: https://github.com/fredotran/traffic-sign-detector-yolov4

[73] Sichkar VN. Traffic-signs-for-training-YOLO-v3-detector. Disponible en: https://github.com/sichkar-valentyn/Traffic-Signs-for-training-YOLO-v3-Detector

[74] Igareta Á. SaferAuto. Disponible en: https://github.com/angeligareta/SaferAuto

ANEXOS

I. ANEXO I

En este anexo se explicará cómo instalar el marco de trabajo Darkflow a través del terminal de Linux.

Primeramente, se debe crear una carpeta para saber dónde se encuentra la posterior instalación. En este caso se creará la carpeta "darkflow" dentro del directorio "lib":

```
$ cd ~/lib
```

Una vez dentro, se clonará el repositorio de Darkflow en GitHub y se posicionará el comando en esa ruta:

```
$ git clone https://github.com/thtrieu/darkflow
$ cd darkflow
```

Se procederá a crear un ambiente virtual para realizar la instalación dentro de ella, ya que Darkflow utiliza el comando "flow" que puede dar problemas en algunas ocasiones.

El ambiente virtual debe reflejar la versión de Python que se vaya a utilizar; como previamente se ha dicho que Darkflow solo funciona con versiones de Python anteriores a la 3.8, se procederá a utilizar la última versión de Python antes de esta.

```
$ virtualenv --python=python3.7.10 .venv
$ source .venv/bin/actívate
```

Una vez instalado y activado el ambiente virtual, se procederá a instalar las dependencias necesarias. Como se puede observar, a la dependencia "tensorflow "ya se le indica qué versión debe instalar porque, como previamente se ha comentado, Darkflow solo funciona con la versión 15 de este.

```
$ pip3 install Cython
$ pip3 install numpy
$ pip3 install tensorflow==1.15
$ pip3 install opencv-python
$ pip3 install .
$ python setup.py build_ext --inplace
```

Luego de haberlo instalado todo sin ningún error, se debe escribir el siguiente comando.

```
$ flow --help
```

Si aparece la lista de comandos es que todo se ha instalado correctamente.

Si luego de hacer “pip3 install .” el terminal da errores con Python se deben instalar las siguientes dependencias:

```
$ sudo apt install build-essential
$ sudo apt install libffi-dev
$ sudo apt install python3.7-dev
```

Si en el sistema ya existe una instalación de Python cuya versión es mayor o igual a la 3.8, se puede instalar Python 3.7 de la siguiente forma:

```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install software-properties-common
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt update
$ sudo apt install python3.7
```

Para comprobar la versión instalada de Python:

```
$ python3.7 --version
```

II. ANEXO II

En el siguiente anexo se procede a explicar cómo instalar Darknet y OpenCV.

Primeramente, se posicionará el comando en el directorio "lib", donde posteriormente se instalará Darknet:

```
$ cd ~/lib
```

Luego se instalará la carpeta de Darknet que se encuentra en el repositorio de GitHub y se hará "make" para que se realice la instalación.

```
$ git clone https://github.com/pjreddie/darknet.git
$ cd darknet
$ make
```

Si sale algún error se debe solucionar, si no se teclea:

```
$ ./darknet
```

Debe salir la siguiente línea:

```
usage: ./darknet <function>
```

Para la instalación de OpenCV desde el terminal se deben instalar primeramente todas las dependencias necesarias

```
$ sudo apt install build-essential cmake git pkg-config libgtk-3-dev
$ sudo apt install libavcodec-dev libavformat-dev libswscale-dev libv4l-dev
$ sudo apt install libxvidcore-dev libx264-dev libjpeg-dev libpng-dev libtiff-dev
$ sudo apt install gfortran openexr libatlas-base-dev python3-dev python3-numpy
$ sudo apt install libtbb2 libtbb-dev libdc1394-22-dev libopenexr-dev
$ sudo apt install libgstreamer-plugins-base1.0-dev libgstreamer1.0-dev
```

Luego se procede a crear la carpeta donde se encontrará OpenCV y a clonar sus repositorios:

```
$ mkdir ~/opencv_build && cd ~/opencv_build
$ git clone https://github.com/opencv/opencv.git
$ git clone https://github.com/opencv/opencv_contrib.git
```

Una vez realizado esto se debe hacer “CMake” para realizar su correcta instalación

```
$ cd ~/opencv_build/opencv
$ mkdir -p build && cd build
$ cmake -D CMAKE_BUILD_TYPE=RELEASE \
-D CMAKE_INSTALL_PREFIX=/usr/local \
-D INSTALL_C_EXAMPLES=ON \
-D INSTALL_PYTHON_EXAMPLES=ON \
-D OPENCV_GENERATE_PKGCONFIG=ON \
-D  OPENCV_EXTRA_MODULES_PATH=~/opencv_build/opencv_contrib/modules
\
-D BUILD_EXAMPLES=ON ..
```

En este punto se debe llevar a cabo la compilación utilizando el siguiente comando:

```
$ make -j12
```

#La j depende del procesador que se tenga, en este caso es de 12 núcleos

#Para saber qué procesador se tiene se puede teclear "nproc"

Tras todo lo hecho anteriormente, se instala OpenCV tecleando:

```
$ sudo make install
```

Para comprobar que se ha instalado correctamente y para saber cuál es su versión:

```
$ python3 -c "import cv2; print(cv2.__version__)"
```

III. ANEXO III

En este anexo se explicará cómo instalar Detectron2 desde el terminal de Linux.

Primeramente, se clona el repositorio en la carpeta "lib" y se instala tecleando los siguientes comandos:

```
$ git clone https://github.com/facebookresearch/detectron2.git
$ python3 -m pip install -e detectron2
```

Luego, se lleva a cabo una compilación previa (solo Linux) que depende de la versión de CUDA; en este caso se utiliza CUDA 11.1, por lo que la versión de torch será 1.8

```
$ python3 -m pip install detectron2 -f \
https://dl.fbaipublicfiles.com/detectron2/wheels/cu111/torch1.8/index.html
```

Tras haber realizado esto, se debe instalar PyTorch. En este caso se instalará para Linux con versión de CUDA 11.1

```
$ pip3 install torch==1.9.0+cu111 torchvision==0.10.0+cu111
torchaudio==0.9.0 -f
https://download.pytorch.org/whl/torch_stable.html
```

Una vez instalado todo, se debe dirigir el comando al repositorio donde se encuentra Detectron2 y se pone en marcha el archivo .py que contiene donde realiza la instalación del marco de trabajo.

En este caso se dirigirá a la carpeta de Detectron2 que se encuentra dentro de "lib":

```
$ cd ~/lib/Detectron2
$ python3 setup.py
```