UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Sistema de guiado automático de vehículos mediante técnicas de inteligencia artificial. Aplicación a un PiRacer

Trabajo Fin de Grado

Grado en Ingeniería Electrónica Industrial y Automática

AUTOR/A: Olivares Pérez, Miguel

Tutor/a: Crespo Lorente, Alfons

CURSO ACADÉMICO: 2022/2023

Agradecimientos

En primer lugar, me gustaría agradecer a mi familia, en especial a mis padres y mi hermana, por todo el sacrificio, apoyo y cariño recibidos en estos años de carrera y en el desarrollo de este trabajo. Gracias de corazón.

También me gustaría agradecer a mis compañeros de grado por estos cuatro maravillosos años en los que hemos vivido y aprendido tanto los unos de los otros, en especial a Jorge Párraga, Alejandro Olcina y Álvaro Castellanos, tres personas excepcionales que espero sigan a mi lado durante mucho tiempo.

Además, quisiera agradecer a Georgina Tejera por su apoyo incondicional, sin el cual este proyecto no habría salido adelante. Gracias por hacerme reír y aprender, pero sobre todo disfrutar a tu lado esta etapa de mi vida.

Por último, quisiera dar las gracias a mi tutor, Alfons Crespo Lorente, por proponerme este emocionante trabajo y asesorarme en todo momento para que este saliera adelante.

Documents del Proyecto

1. Memoria
2. Pliego de Condiciones
3. Presupuesto

Sistema de guiado automático de vehículos mediante técnicas de inteligencia artificial. Aplicación a un PiRacer

DOCUMENTO 1. MEMORIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Trabajo Final de Grado en Ingeniería Electrónica Industrial y Automática

Alumno: Miguel Olivares Pérez

Tutor: Alfons Crespo Lorente

Curso 2022-2023

Resumen

En el presente Trabajo de Fin de Grado, se ha entrenado una red neuronal para la detección de señales de tráfico en tiempo real, haciendo uso de conjuntos de datos personalizados. A su vez, se ha diseñado e implementado software para la conducción autónoma de un vehículo con función de radio control. Para ello, se han analizado y probado distintas opciones, las cuales se desarrollarán en más detalle en el proyecto. Finalmente, se ha decidido optar por trabajar utilizando un kit PiRacer Pro con RaspBerry Pi 4B para la conducción autónoma, y el algoritmo YOLOv4 junto a Darknet para la detección de objetos. Haciendo uso de estos componentes se desarrollarán ambas partes con éxito y se darán alternativas a su uso para un mejor desempeño en futuras actualizaciones del proyecto.

Abstract

In this Final Degree Thesis, a neural network has been trained for real-time traffic signs detection, using custom datasets. At the same time, software has been designed and implemented for its use in a radio control vehicle's autonomous driving. To do that, different options, which will be later explained, have been analysed and tried. Finally, it has been decided to work with a PiRacer Pro kit which uses RaspBerry Pi 4B for autonomous driving, and with YOLOv4 algorythm together with Darknet for object detection. Using these components, both parts will be successfully developed and alternative solutions will be given in order to improve the future performance of the project.

Documento 1. Memoria

1. Introducción 8

2. Objetivos 8

3. Estado del Arte 9

3.1. Detección y clasificación de objetos . . . 9
3.1.1. ¿Qué es una red neuronal? . . . 9
3.1.2. Función de error y coste . . . 11
3.1.3. Propagación hacia adelante y hacia atrás . . . 12
3.1.4. Redes Neuronales Profundas . . . 13
3.1.5. Redes Neuronales Convolucionales . . . 14
3.2. Herramientas para el uso de redes neuronales . . . 16
3.2.1. Tensorflow . . . 16
3.2.2. Keras . . . 16
3.2.3. Torch/Pytorch . . . 17
3.2.4. CUDA . . . 17
3.2.5. OpenCV . . . 17
3.2.6. Herramienta YOLO . . . 18
3.3. Sistemas empotrados . . . 20
3.3.1. Donkeycar . . . 20
3.3.2. Placas de desarrollo disponibles . . . 21
3.3.3. Vehículos disponibles . . . 23
3.3.4. Solución adoptada . . . 25

4. Diseño de la solución 26

4.1. Modelo de reconocimiento de señales de tráfico . . . 26
4.1.1. Conjunto de datos . . . 26
4.1.2. Entrenamiento de la red . . . 28
4.1.3. Extracción y análisis de los resultados . . . 32
4.2. Configuración del vehículo . . . 35
4.2.1. Configuración de RaspBerry Pi . . . 35
4.2.2. Calibrado de dirección y aceleración . . . 36
4.3. Prueba de funcionamiento . . . 37
4.3.1. Conducción autónoma por circuito . . . 37
4.3.2. Integración del modelo de detección de señales . . . 42

5. Conclusiones 44

6. Acrónimos 45

Índice de figuras

1. Neurona biológica frente a neurona artificial. 10
2. Función de activación sigmoide y no linealidad en la salida. . . 10
3. Red neuronal simple. 11
4. Red neuronal profunda con 3 capas ocultas. 13
5. Operación de convolución 14
6. Red neuronal convolucional 15
7. Comparación de rendimiento de YOLOv4 frente a su versión anterior 19
8. Detalle de RaspBerry Pi 4 22
9. Detalle de Jetson Nano 23
10. Kit DonkeyCar 24
11. Detalle de Piracer 24
12. Detalle de PiRacer Pro 25
13. Señales de tráfico elegidas para su posterior reconocimiento . . 26
14. Contenido de los archivos .name y .data del modelo 29
15. Entrenamiento de la red en Darknet 32
16. Gráfico del entrenamiento de la red 33
17. Imagen compuesta 34
18. Imagen real 34
19. Ejemplo de calibrado de dirección 36
20. Calibrado incluído en myconfig.py 37
21. Transformación de una imagen BGR a formato HSV 38
22. Búsqueda de los valores ideales HSV mediante trackbars . . . 39
23. Implementación del filtro HSV 39
24. Centro y centroide de los contornos, obtenido a partir de la imagen inicial recortada 41
25. Contornos y ángulo obtenido a partir de los mismos 41
26. Detección de señales de 30 y 120 Km/h en circuito 43
27. Detección de señales de Stop y Ceda el Paso en circuito 43

1. Introducción

La detección de objetos en tiempo real ha sido una total revolución durante los últimos años. Tanto, que cualquier vehículo moderno es capaz de detectar las señales de tráfico con las que se cruza en la calzada. Además, dicha detección de objetos es cada vez más importante en el campo de la medicina, haciendo posible el diagnóstico de enfermedades antes de que las detecte el imperfecto ojo humano.

Es por ello que se decidió empezar este proyecto. La motivación principal de este es hacer algo distinto a lo usual y a su vez aprender de las técnicas tan innovadoras que se usan hoy en día para proyectos de este estilo, pero que no son del todo visibles para la mayoría de las personas. Este proyecto intenta plasmar los conocimientos obtenidos del grado, y además conocer nuevos alcances de la tecnología estudiada en el proceso.

A lo largo de la lectura del proyecto, se podrá observar el diseño e implementación del software necesario para que un vehículo conduzca de forma autónoma alrededor de un circuito, detectando a su paso las distintas señales de tráfico que aparezcan.

2. Objetivos

El objetivo principal del proyecto se basa en crear y realizar pruebas experimentales a un prototipo, compuesto por 2 funcionalidades complementarias: la conducción autónoma y la detección de señales de tráfico. Dichas funciones pueden ser integradas al mismo tiempo en futuros desarrollos, ya que la elección del prototipo de pruebas puede condicionar en gran medida la capacidad de integración de estas.

El éxito del proyecto reside en la capacidad del vehículo de ejecutar ambas partes por separado correctamente, siendo posible su integración, si el hardware lo permite.

3. Estado del Arte

Durante el transcurso de los últimos años, la población mundial ha sido testigo de numerosos avances científicos, pero sin ninguna duda el que más ha llamado la atención al público es el de la Inteligencia Artificial. Y es que nos resulta tan fascinante que una máquina pueda aprender por sí misma, que no nos damos cuenta de que hace mucho tiempo que convivimos con esta tecnología.

Hoy en día, la mayor parte del trabajo automático que vemos en nuestra vida cotidiana está desarrollado mediante técnicas de Inteligencia Artificial, como la detección automática de objetos. Ejemplo de ello es la cámara que lee la matrícula de un coche para dejarle salir de un aparcamiento, o la lectura de los famosos códigos QR.

En este proyecto se ha desarrollado un modelo de detección y clasificación de señales de tráfico, para posteriormente incluirlo en un vehículo PiRacer Pro, y conseguir una conducción autónoma del mismo a lo largo de un circuito predefinido.

3.1. Detección y clasificación de objetos

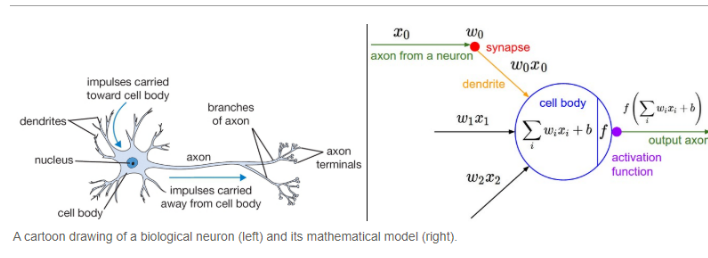
Para poder comprender el funcionamiento y el proceso que ha llevado al completo desarrollo del presente proyecto, es imperativo el entendimiento de los métodos de detección y clasificación de objetos que se han seguido. En los siguientes apartados se desarrollarán desde conceptos básicos como qué es una red neuronal, hasta el funcionamiento de las redes neuronales convolucionales y su importancia en entornos de detección de objetos.

Durante las explicaciones de dichos conceptos, se abordarán las redes neuronales dedicadas al reconocimiento de imágenes y se analizarán sus principales características y alternativas[Ber18]. Explicar redes con otros propósitos podría resultar en conclusiones erróneas, ya que sus métodos ,aunque muy similares, no son totalmente idénticos.

3.1.1. ¿Qué es una red neuronal?

Como su propio nombre indica, las redes neuronales intentan imitar el procesamiento del cerebro, usando ”neuronas” conectadas entre sí, que tras un entrenamiento complejo, consiguen formar un modelo para resolver un problema concreto. Estas ”neuronas” calculan valores de salida a partir de valores de entrada, mediante el uso de pesos (w), sesgos (b) y funciones de

activación. En la siguiente figura se pueden apreciar las similitudes entre las neuronas biológicas y artificiales.

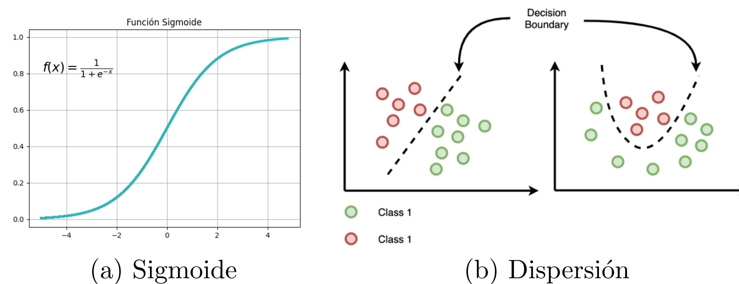


A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Figura 1: Neurona biológica frente a neurona artificial.

La función por defecto que contiene una neurona es una función lineal, la cual toma como parámetros los datos de entrada "X", los pesos "w", y los sesgos "b" ($Y = w \cdot X + b$). Si se construyera una red utilizando estas neuronas interconectadas entre sí, no se conseguiría más que un comportamiento linear, sin importar que la red sea más o menos densa. Dado que la mayor parte de problemas a resolver en el mundo real no pueden ser resueltos utilizando regresión linear, se desarrollaron las llamadas funciones de activación.

Las funciones de activación son funciones matemáticas de la forma $f(x)$, que se agregan a las redes neuronales para ayudar a la red a aprender y saber actuar frente a patrones más complejos. Unas de las más famosas o más utilizadas son las funciones Sigmoide, Tangente Hiperbólica (TanH) o ReLU. Una vez que la red utilice estas funciones, será capaz de realizar predicciones mucho más precisas que la linear estándar. Cabe recalcar que, para que el aprendizaje de la red sea lo más preciso posible, el arquitecto de la red deberá conocer las funciones de activación que se ajusten mejor matemáticamente al problema que se quiere afrontar.



(a) Sigmoide (b) Dispersión

Figura 2: Función de activación sigmoide y no linealidad en la salida.

Tras detallar el funcionamiento de una neurona dentro de la red, el siguiente paso es ahondar en las redes neuronales simples, o aquellas que no se componen de muchas capas. Estas redes, debido a su sencillez no podrán hacer frente a problemas de gran magnitud, pero sí hacer un gran trabajo en la resolución de cuestiones menos exigentes.

La siguiente figura muestra una red neuronal completamente conectada, en la cual, mediante las técnicas de aprendizaje que se explicarán en los siguientes apartados de este proyecto, se podrá entrenar un modelo simple, de clasificación binaria por ejemplo. Véase que cada neurona está conectada a todas las de la siguiente capa, haciendo que cada salida de una neurona sirva como una de las entradas de todas las siguientes, hasta llegar a las neuronas finales, o, si es conveniente para el problema a solucionar, una última neurona que nos dé la predicción deseada.

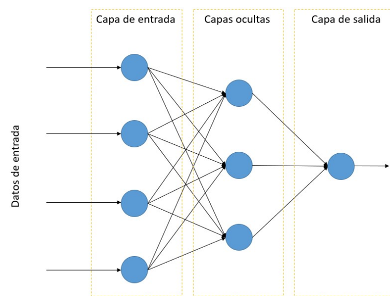


Figura 3: Red neuronal simple.

3.1.2. Función de error y coste

En el anterior apartado se ha explicado el funcionamiento de una neurona artificial, y cómo podemos hacer que su comportamiento cambie y se ajuste a la resolución de problemas más complejos mediante funciones de activación y una red simple. Conociendo el diseño de esta red, cabe preguntarse: ¿Cómo es capaz dicha red de aprender?

Para definir este aprendizaje, se utilizan las funciones de error y coste, contenidas una dentro de otra, respectivamente. La función de error para una regresión logística , siendo "Y" la predicción de nuestra red, e "y" la respuesta real al problema, se define como:

$$L(Y, y) = -(y \cdot \log(Y) + (1 - y) \cdot \log(1 - Y)) \tag{1}$$

Esta ecuación muestra el error entre la predicción hecha por la red para un ejemplo, y la respuesta real. Para el cálculo medio del error en todo el

conjunto de datos se utiliza la función de coste, la cual guarda este error medio, en función de los parámetros de la red: los pesos y los sesgos.

$$J(w,b) = \frac{1}{m} \cdot \sum_{i=1}^{m} L(Y_i, y_i) \tag{2}$$

Siendo "m" el número de ejemplos del conjunto de datos, e "i" el ejemplo que se está calculando en ese momento, la ecuación de coste devuelve el error medio de nuestro sistema, y por lo tanto da la clave para modificar los parámetros de la red, e intentar reducir al máximo este error.

Estas funciones serán la clave para que la red neuronal sea capaz de resolver un problema sin haberlo visto antes, mediante su entrenamiento con ejemplos similares.

3.1.3. Propagación hacia adelante y hacia atrás

La propagación hacia delante y hacia atrás se pueden definir como los procesos mediante los cuales se obtiene el error medio de una iteración, para después modificar los parámetros pertinentes y volver a empezar. En el caso de la propagación hacia adelante, se utiliza la función de coste, así como la entrada "X" y los parámetros a modificar, para conseguir el error medio. El cálculo que sigue cada neurona es el siguiente:

$$Z = W_1 \cdot X_1 + W_2 \cdot X_2 + b \tag{3}$$

$$Y = \sigma(Z) \tag{4}$$

En la variable "Z" se almacena el resultado de la función lineal propia de la neurona. Después, mediante el uso de la función de activación necesaria (en este caso la sigmoide), se calcula el resultado predicho con forma no lineal. Por último, haciendo esto con toda la red, tanto la predicción como la solución real del ejemplo son utilizadas como parámetros de la función de error, y a su vez por la de coste, para calcular el coste del modelo en la iteración actual.

El cálculo del coste marca el fin de la propagación hacia adelante, y da una buena visión del comportamiento de la red en cada iteración, pero, para que la red neuronal pueda desarrollar respuestas correctas, necesita corregir los pesos y los sesgos y así volver a realizar otra iteración con ellos. En este momento, se produce la propagación hacia atrás, la cual mediante el uso de derivadas, consigue obtener estos nuevos parámetros.

$$W_i = W_i - \alpha \cdot \frac{dJ(W,b)}{dW_i} \tag{5}$$

$$b_i = b_i - \alpha \cdot \frac{dJ(W,b)}{db_i} \tag{6}$$

En ambas ecuaciones, la constante alfa es usada. Esta constante se llama ratio de aprendizaje y la definirá el arquitecto de la red para indicar cuánto se modifican los parámetros en cada iteración.

Una vez finalizada la propagación hacia atrás, la red debe realizar muchas iteraciones con un número muy alto de ejemplos, para poder llevar sus pesos y sesgos al valor más óptimo posible, y por lo tanto a una precisión más alta de sus predicciones.

3.1.4. Redes Neuronales Profundas

Las redes neuronales profundas, como su nombre indica, son redes con un alto número de capas de neuronas ocultas, las cuales permiten a estas redes llegar a un nivel de precisión muy alto con respecto a sus predecesoras.

Mediante el uso de estas redes, el modelo podrá reconocer patrones cada vez más complicados, aunque no son las mejores para las tareas que competen a este proyecto, como se verá en los siguientes apartados. El arquitecto de la red será el encargado de decidir el número de capas ocultas que tendrá, su ratio de aprendizaje, las funciones de activación en cada capa, y otras características. Las mencionadas características de la red tendrán un impacto muy grande en el comportamiento de esta, modificando desde el número de iteraciones a completar, hasta el número de conexiones neuronales a eliminar, ya que podría reducir un posible sobreajuste, también denominado "overfitting".

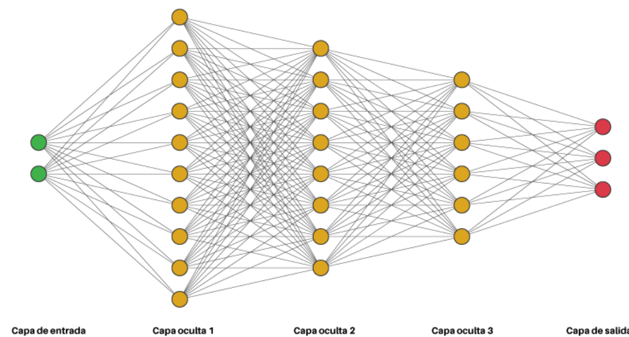


Figura 4: Red neuronal profunda con 3 capas ocultas.

En la figura 4 podemos apreciar las diferencias de una red neuronal profunda, frente a la red simple mostrada en la figura 3.

3.1.5. Redes Neuronales Convolucionales

Si bien las redes neuronales profundas consiguen obtener un mayor nivel de precisión en los modelos que entrenan, existe un tipo de redes neuronales diseñadas específicamente para procesar datos de tipo y forma similar al que podría tener una imagen. Las redes neuronales convolucionales (CNN por sus siglas en inglés), utilizan una operación matemática llamada convolución para extraer determinadas características a lo largo del entrenamiento.

En esta operación matemática, se utilizan uno o varios filtros compuestos por parámetros del entrenamiento. Estos filtros se desplazan a través de la matriz de datos de entrada, multiplicando cada uno de sus valores por el valor de sus parámetros correspondientes. Estos valores se suman para obtener un solo parámetro que se almacena en el correspondiente lugar de la matriz de salida.

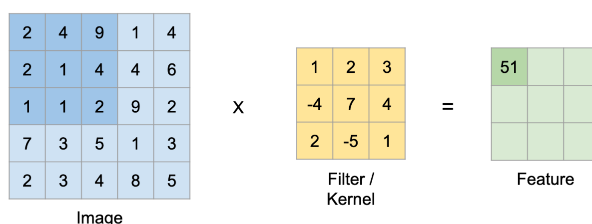


Figura 5: Operación de convolución

En la figura se puede observar el valor de la primera operación de convolución, la cual más tarde se desplazará de derecha a izquierda y de arriba a abajo, formando una matriz de salida de la convolución, de tamaño 3x3 en este caso. La siguiente expresión matemática muestra el cálculo realizado en la figura:

$$2 \cdot 1 + 4 \cdot 2 + 9 \cdot 3 + 2 \cdot (-4) + 1 \cdot 7 + 4 \cdot 4 + 1 \cdot 2 + 1 \cdot (-5) + 1 \cdot 2 = 51 \quad (7)$$

En el caso anterior se explica un tipo de convolución con filtro bidimensional. Este caso se puede aplicar también a filtros tridimensionales, como puede ser el caso de las imágenes en color. Estas operaciones se combinan a lo largo del modelo, para así formar una red neuronal profunda convolucional.

Estas redes profundas, combinan las operaciones de convolución para obtener distintas características de los datos de entrada. Por ejemplo, las primeras

capas se podrían dedicar a la obtención de contornos y formas, mientras que las siguientes podrían aprender de las texturas y relieves de las imágenes. El tamaño de los filtros, la cantidad de ellos y la forma en la que se desplaza el filtro en la matriz de entrada, son parámetros que se definen antes del entrenamiento, los cuales harán que se obtengan parámetros más o menos acertados. Es por ello que es necesario un gran conocimiento de arquitectura de las redes para conseguir un modelo óptimo para el uso que se le quiera dar.

Para que las últimas capas de la red se puedan clasificar con facilidad, existen técnicas de reducción del tamaño de las matrices, sin que estas pierdan las características o datos importantes que guardan para el modelo. Son las llamadas en inglés "Pooling Layers". Estas capas que se añaden entre capas de convolución, pueden ser de dos tipos: max pool y average pool. Ambos nombres se refieren a las operaciones que se realizan en estas capas para hacer más pequeñas las matrices.

Estas operaciones se podrían describir brevemente como la división de la matriz en varias partes, las cuales o bien devuelven el mayor de los valores de esa parte y lo incluyen en una nueva matriz, o bien se hace la media de esos valores y se incluye en la nueva matriz.

El tamaño de las divisiones de la matriz inicial, así como el tipo de pool se definen también en la configuración de la red.

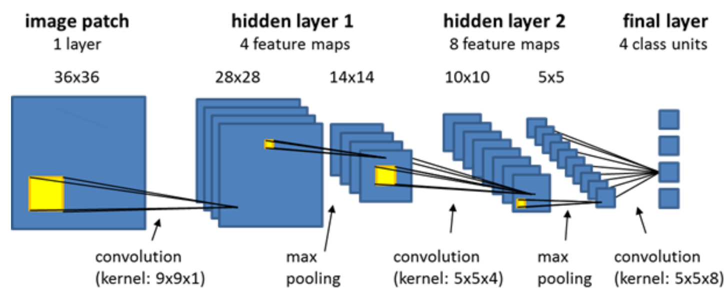


Figura 6: Red neuronal convolucional

En la figura, se puede apreciar como estas nuevas capas hacen más asequibles las matrices para su clasificación al final de la red, teniendo solo una fila de valores a comparar.

3.2. Herramientas para el uso de redes neuronales

Para abordar la programación de las redes neuronales es necesario disponer de una infraestructura base con el fin de poderlas aplicar con un nivel de calidad en entornos físicos. El integrar herramientas y librerías para el tratamiento de redes neuronales a nivel de programación permiten acelerar las operaciones básicas obteniendo tiempos de respuesta razonable. El uso de hardware dedicado para ello es otro elemento fundamental para conseguir buenos tiempos de cálculo. Se tienen disponibles en la red una serie de herramientas de libre disposición para su integración y uso en aplicaciones que permiten la rápida integración y desarrollo de aplicaciones. Python o R son los lenguajes de programación más usuales en estas herramientas[Cho18]. A continuación se presentan algunas herramientas que permiten el uso de las redes neuronales en las aplicaciones.

3.2.1. Tensorflow

Tensorflow es una librería de software de código abierto diseñada para realizar el cálculo de vectores (tensores) numéricos de forma eficiente. Desarrollado por el equipo de Google, Tensorflow permite a los desarrolladores el uso de librerías para el desarrollo de aplicaciones de machine learning sobre distintos procesadores, ya sean de propósito general (CPU), o gráficos (GPU). Para hacer el desarrollo de aplicaciones con Tensorflow lo más optimizado posible, Google ha desarrollado una unidad de hardware para procesar tensores directamente, llamada Tensor Processing Unit (TPU), la cual consigue una eficiencia muy alta en el procesamiento de vectores.

La interfaz de Tensorflow proporciona las funciones necesarias para el proceso de aprendizaje automático y profundo. Se ha desarrollado en C/C++ y ofrece interfaces (APIs) con lenguajes como Python, JavaScipt, Java, R y otros.

Además, Tensorflow es compatible con plataformas de computación en la nube, como por ejemplo Google Cloud y Amazon Web Services. Esto lo hace una herramienta a tener en cuenta si se desea desplegar aplicaciones o modelos en servidores en la nube.

3.2.2. Keras

Keras es una biblioteca de aprendizaje profundo escrita en Python, capaz de ejecutarse tanto sobre Tensorflow, como otras herramientas similares. Presenta una interfaz de fácil uso que oculta las complejidades internas de dichas herramientas, lo que lo hace uno de los entornos más utilizados tanto en desarrollos a nivel personal como empresarial o de producción[Tor19].

Keras permite la gestión y configuración de los bloques de redes neuronales mas usados, como pueden ser el número y tipo de capas, funciones de activación u optimizadores. Keras también es capaz de gestionar modelos convolucionales y redes neuronales recurrentes.

Keras es una buena opción para desarrolladores que deseen aumentar las funcionalidades de sus modelos de aprendizaje profundo, a la vez que hacer su manejo y visualización más sencillos. Esto hará que sean capaces de desarrollar aplicaciones a un ritmo mucho más rápido.

3.2.3. Torch/Pytorch

Torch es una librería de código abierto para aprendizaje automático. Desarrollado inicialmente por Facebook, Torch ha sido usado a su vez por Twitter y Google, entre otros. Una característica que hace a esta librería útil para el desarrollo de proyectos de aprendizaje automático, es el uso de CUDA o Compute Unified Device Architecture.

Pytorch[lab16] es un entorno de programación cuyo uso principal reside en el aprendizaje profundo. Este entorno se basa en el lenguaje Python, y utiliza la librería Torch mencionada anteriormente. Pytorch ha sido bien recibida por la comunidad, ya que permite utilizar modelos de gran eficiencia directamente desde Python.

3.2.4. CUDA

Mencionada anteriormente, CUDA (Compute Unified Device Architecture)[VLR12] no es un entorno de modelado de redes neuronales, pero puede ser utilizado para acelerar las operaciones de dichas redes. Se basa en una plataforma de ejecución para la computación paralela, integrado mediante una librería, y un hardware específico, normalmente GPU. Creado por NVIDIA, permite acelerar los gráficos en el ordenador y extenderse como librería para el uso de otras aplicaciones. Se puede hacer uso de CUDA desde lenguajes como C, C++, Python, Java, etc.

3.2.5. OpenCV

OpenCV es una librería bajo licencia BSD, normalmente utilizada para procesamiento de imágenes. Desarrollada por Intel, se ha convertido en una de las librerías mas usada para el proceso de imágenes por su gran versatilidad.[Int16]

La librería dispone de un gran abanico de funciones, las cuales cubren la mayor parte de necesidades en el tratamiento de imágenes, así como en de-

tección de objetos o visión en tiempo real. Sus usos van desde el personal hasta su uso en campos de investigación e industriales diversos: robótica, guiado automático, seguimiento de objetos, etc.

Desarrollada inicialmente en C, se puede utilizar utilizando distintos lenguajes como C, C++, Java y Python.

3.2.6. Herramienta YOLO

Existen incontables usos de las redes neuronales y de los modelos de aprendizaje profundo, pero uno de los más visuales e importantes de todos es la detección y clasificación de objetos mediante el uso de una cámara. La detección de objetos se puede llevar a cabo mediante el uso de distintos frameworks dedicados a la creación y ejecución de modelos de aprendizaje profundo, como pueden ser Tensorflow o Keras. Estos frameworks no suelen ser intuitivos en su uso y la ejecución de los modelos no saca el máximo partido en tiempo real.

Por ello principalmente, se desarrollaron otras plataformas y algoritmos que hicieran el entrenamiento de una red neuronal y su posterior ejecución más sencillos. Entre estos algoritmos destaca uno que ha demostrado una gran precisión y velocidad de detección en modelos de tiempo real: el algoritmo YOLO (You Only Look Once).

La primera implementación de YOLO[Red15] se dio en 2015, de la mano de Joseph Redmon en un artículo llamado "You Only Look Once: Unified, Real-Time Object Detection". Desde esta primera implementación, han ido surgiendo distintas versiones del algoritmo soportados en diferentes plataformas, siendo la más famosa y usada de ellas YOLOv3[RF18]. En este momento, la versión más nueva es la YOLOv7, aunque por las necesidades del proyecto y del hardware utilizado, se usará YOLOv4, el cual ha demostrado un gran rendimiento a altos FPS, como muestra la siguiente figura:

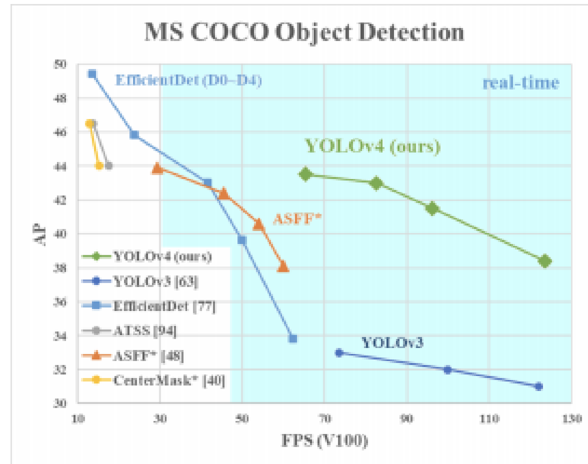


Figura 7: Comparación de rendimiento de YOLOv4 frente a su versión anterior

En posteriores versiones, se demostró que se conseguían más FPS, pero a partir de YOLOv5, los modelos debían ser entrenados en la plataforma PyTorch. La dificultad de entrenamiento y su instalación hicieron que para el proyecto se decidiera que valía la pena optar por la versión anterior, la cual funciona en Darknet.

Darknet[Red16] es un framework desarrollado por un usuario de GitHub, compatible con los modelos de tipo YOLO. Su instalación es muy sencilla y en la página del autor se describen los pasos necesarios para poder configurar un modelo a antojo del usuario. Los pasos a seguir para la configuración completa de un modelo a entrenar, serán explicados con detenimiento en el diseño de la solución, pero se podrían definir brevemente como:

Primero, la creación de un conjunto de imágenes de entrenamiento y de test, las cuales tengan a su vez una etiqueta por imagen que muestre el lugar en la imagen donde se encuentra el objeto a identificar y la clase que representa. Estos datos se incluirán en la carpeta pertinente de Darknet.

Después, se modificará el archivo de configuración para hacer el entrenamiento lo más óptimo posible, y se crearán dos archivos que indican las clases del modelo, así como los lugares donde se encuentran los archivos importantes para la configuración.

Por último, se hacen las comprobaciones necesarias para que el entrenamiento funcione, como que la tarjeta gráfica funcione, o que existan todos los

archivos.

Con estas pautas, el modelo podrá ser entrenado y validado dentro de Darknet, y así obtener los parámetros o pesos finales.

3.3. Sistemas empotrados

Una vez explicado el funcionamiento de una red neuronal, es imperativo hablar de los sistemas empotrados o embebidos, los cuales serán los utilizados en el proyecto.

Los sistemas empotrados se definen como sistemas de computación basados en un microprocesador o un microcontrolador diseñados para realizar una o algunas funciones dedicadas. Estos sistemas, por norma general intentan ser lo más pequeños posible, para así poder incluirse dentro de estructuras complejas como coches, satélites, aviones, etc.

Existen muchos tipos de sistemas empotrados, los más famosos los que desempeñan tareas mencionadas anteriormente y que se centran en realizar operaciones en tiempo real. Sin embargo, desde hace años, se ha disparado el uso de otro tipo de ellos, los sistemas empotrados de uso prototípico y desarrollo de aplicaciones. Estos sistemas son de tamaño reducido, e intentan ofrecer las máximas prestaciones posibles dentro de sus posibilidades para así crear aplicaciones de todo tipo. Los más famosos en este tipo de sistemas son Arduino y Raspberri Pi.

Tanto Arduino como Raspberry han sido un antes y un después en el desarrollo de aplicaciones de usuarios sin experiencia, e incluso una gran ayuda para las personas que pretenden aprender a programar. Ambos tienen características similares, si bien es cierto que Raspberry ofrece más funcionalidades, ya que contiene un sistema operativo propio basado en Linux.

3.3.1. Donkeycar

El objetivo principal de este proyecto se basa en la conducción autónoma atendiendo a las señales de tráfico que se detecten, por lo tanto se ha de escoger un vehículo en el cual poder desarrollar el proyecto. Entre todas las opciones de coches a radiocontrol (RC) disponibles, el proyecto DonkeyCar[Don] sin duda llama la atención debido a sus infinitas prestaciones.

Este proyecto se basa en el desarrollo de una librería a alto nivel escrita en python, mediante la cual se puedan controlar vehículos, normalmente dedicados a la conducción autónoma. Es por ello que DonkeyCar ofrece módulos

dedicados al entrenamiento de modelos de inteligencia artificial para que el vehículo pueda recorrer el circuito sin ayuda del ser humano.

Entre sus muchas ventajas, el código del proyecto es abierto al público, lo que permite una gran personalización. Esto hace que cada usuario que utilice esta librería y sus distintos tipos de hardware (de los cuales se hablará en siguientes apartados), cree proyectos totalmente diferentes y con todo tipo de fines. Es por todas estas características por las cuales se ha decidido usar DonkeyCar para el presente proyecto.

Su funcionamiento, aunque complicado a más bajo nivel, es fácil de entender si se atiende a los módulos que contiene. Está formado por distintas ubicaciones de archivos, dos de las cuales son de especial interés: /templates y /parts.

En la carpeta /templates se almacenan todos los programas completos que se ejecutan, cada uno haciendo referencia a distintos usos. Unos ejemplos pueden ser los templates que se dedican a controlar el coche con mando, y otros que permiten todos los controles a la vez

Por otro lado, en la carpeta /parts, se incluyen los módulos individuales dedicados a tareas específicas que se incluyen dentro de los templates. Ejemplos de estos módulos podrían ser unos dedicados al uso de cámara, o a la aceleración y giro del vehículo.

Mediante comandos propios de Donkey, se selecciona la plantilla deseada para luego crear un modelo con las características ideales para el proyecto. Tanto las plantillas como los módulos son personalizables y pueden ser creados desde cero para ajustarse al uso deseado por el programador.

Durante el proyecto se crearan estos dos tipos de programa para conseguir una conducción autónoma por el circuito.

3.3.2. Placas de desarrollo disponibles

Para este proyecto, se han valorado opciones de placas de desarrollo que se ajusten a las necesidades propias de la conducción autónoma y de la detección de objetos. Tras analizar los dispositivos disponibles en el mercado, se ha llegado a la conclusión de que los siguientes podrían ser los indicados:

RaspBerry Pi 4[Jol21]. Esta placa de desarrollo es una de las más famosas del mundo en lo que respecta a proyectos de baja envergadura, con capacidad de cómputo limitada. Aunque es una placa muy polivalente con usos casi limitados a la mente del usuario, no llega a tener una velocidad competente

en el mercado de alta gama. Sin embargo, podría ser perfectamente factible su uso en este proyecto, ya que salvo el entrenamiento de la red neuronal, el cual se hará en ordenador, no hay otros usos que requieran una capacidad de cómputo muy alta. El único punto crítico en el proyecto será la detección de los objetos en tiempo real.

Si se habla de características técnicas, la RaspBerry Pi 4 contiene un microprocesador ARM Cortex-A72, de 1 a 4 GB de RAM dependiendo del modelo, Bluetooth 5.0, 4 puertos USB, 2 puertos micro-HDMI y hasta 40 pines GPIO, entre otras cosas. Mediante el uso de tarjeta micro-sd, será posible incluir el sistema operativo Raspbian que el usuario desee. En la siguiente figura se aprecian las características mencionadas anteriormente:

Figura 8: Detalle de RaspBerry Pi 4

Por otro lado, también podría utilizarse la placa de desarrollo NVIDIA Jetson Nano. Este kit de desarrollo ha sido creado por la compañía NVIDIA, específicamente para su uso en sistemas de inteligencia artificial. A diferencia de RaspBerry, cuenta con tarjeta gráfica o GPU, la cual permite una velocidad y capacidad de cómputo mucho mayores. Al igual que RaspBerry, también es considerada un ordenador en miniatura, y contiene un sistema operativo basado en Linux para mucha más facilidad de desarrollo.

En cuanto a especificaciones técnicas, este kit contiene una GPU 128-core Maxwell, un microprocesador Quad-core ARM A57, 4GB de RAM, puerto micro-HDMI, 4 puertos USB, pines GPIO y puerto Ethernet, entre otras. A su vez cuenta con un disipador en forma de radiador para evitar el sobrecalentamiento de la placa y así conseguir aún más potencia.

Este kit de desarrollo sería óptimo para este proyecto debido a sus carac-

terísticas técnicas y su reducido tamaño, siendo útil tanto para el desarrollo de código como para la detección de objetos en tiempo real. En la siguiente figura se pueden apreciar los detalles del kit:

Figura 9: Detalle de Jetson Nano

3.3.3. Vehículos disponibles

Debido a que el proyecto DonkeyCar es de código abierto, existen infinidad de ejemplos de coches construidos por usuarios. Sin embargo existen algunos kits ya preparados para sacar el máximo partido y compatibilidad.

La página oficial de DonkeyCar ofrece un kit para construir un coche compatible, o modificar otro para hacerlo compatible. Este kit consiste en un chasis impreso en 3D, adaptadores para las placas, cámara, drivers del servomotor y todos los tornillos y cables necesarios. El vehículo resultante tras adquirir los componentes restantes de cualquier tienda online tendría el siguiente aspecto:

Figura 10: Kit DonkeyCar

Las siguientes opciones de vehículos son parte de la página web de WaveShare, en la cual se presentan dos kits dedicados a las carreras de coches a radio control, pero modificados para su uso recreativo en conducción autónoma.

El primero de ellos es PiRacer[Rac20]. Este coche, con una velocidad media, cuenta con todo lo necesario para su uso en una conducción autónoma: placa de desarrollo, cámara, batería duradera, etc. Compatible con RaspberryPi y Jetson Nano, este vehículo podría ser perfectamente capaz de desarrollar las tareas propias del proyecto.

Figura 11: Detalle de Piracer

El siguiente vehículo podría describirse como una actualización más avanzada de PiRacer, el llamado PiRacer Pro. Este vehículo cuenta con todas las especificaciones técnicas de PiRacer, y además ofrece más calidad. PiRacer Pro cuenta con suspensión mediante muelles, motor de corriente continua más potente y más capacidad de batería. Estas características lo hacen un modelo a tener en cuenta para el presente proyecto, debido a que cumple de sobra las espectativas necesarias para su desarrollo.

Figura 12: Detalle de PiRacer Pro

3.3.4. Solución adoptada

Atendiendo a las características del proyecto, sin ninguna duda la solución a elegir entre vehículo y placa de desarrollo es clara: el kit de desarrollo de NVIDIA Jetson Nano y el vehículo PiRacer Pro son los más indicados para un correcto rendimiento. Sin embargo, debido a la falta de medios, el proyecto finalmente se desarrollará haciendo uso de la placa de desarrollo RaspBerryPi 4, junto con el vehículo PiRacer Pro.

A su vez, se planteó la idea de crear un vehículo propio para este proyecto, pero el coste de tiempo necesario, a la vez que el coste económico de ello, produjo que se tomara la decisión de optar por otras alternativas.

4. Diseño de la solución

4.1. Modelo de reconocimiento de señales de tráfico

En este apartado del proyecto, se explicará la manera mediante la cual se diseñó, entrenó y analizó el modelo de reconocimiento de señales de trafico. Para ello, distinguimos dos grandes objetivos a conseguir:

Por un lado es imprescindible contar con el conjunto de datos adecuado, el cual permitirá que la red sea entrenada de acuerdo con los objetos que más tarde tendrá que detectar, consiguiendo así mejorar la precisión.

Por otro lado, una correcta configuración de la red y de Darknet, evitará fallos durante el proceso e incluso hará que este sea más rápido y fluido.

4.1.1. Conjunto de datos

Existen muchas formas de conseguir imágenes que sirvan como base para el entrenamiento de una red neuronal, pero tanto la elección de la cantidad de estas, las diferentes clases a tratar o incluso el tamaño de las imágenes, serán de gran importancia para el correcto funcionamiento de la red.

Teniendo esto en cuenta, se llegó a la conclusión de que las señales de 30 km/h, 120 km/h, Stop y Ceda el Paso, mostradas a continuación, serían las indicadas para el modelo de detección de objetos debido principalmente a las diferencias en forma y color entre ellas, a excepción de las señales de 30 y 120 km/h, las cuales sí son parecidas en color y forma. Sin embargo el distinto número de dígitos entre ambas debería ser suficiente para que la red haga predicciones bastante exactas.

Figura 13: Señales de tráfico elegidas para su posterior reconocimiento

Una vez definidas las clases a detectar en el proyecto, el siguiente paso es recolectar el mayor número de imágenes posible para el correcto funcionamiento del entrenamiento. En primera instancia, se optó por el uso de la base de datos de GTSRB[Han13], la cual cuenta con un gran número de imágenes

separadas en 42 clases distintas, entre las cuales se encuentran las clases previamente mencionadas. Aunque tal conjunto de datos desempeñó una gran función en el reconocimiento de las señales, estas solo eran reconocidas cuando ocupaban gran parte de la imagen, debido principalmente a que todas las imágenes de GTSRB con las que entrenó a la red también ocupaban todo el perímetro de la imagen. Si se buscaba la detección de las señales cuando el objeto aún es de tamaño reducido con respecto al cuadro de la cámara, se debía incluir imágenes de entrenamiento que cumpliesen este requisito.

En la página web [Rob], se dispone de herramientas que facilitan en gran medida el trabajo de recolección de datos, o en este caso, imágenes, a partir de un número limitado de estas.

Tras una búsqueda exhaustiva de imágenes relativas a las cuatro clases mencionadas anteriormente, se procedió a recopilar las 3220 imágenes que forman el conjunto de datos de entrenamiento, 120 de ellas dedicadas al conjunto de datos de comprobación o test. Todas las imágenes son de tamaño 416x416, y a todas ellas se le han aplicado técnicas de aumento de datos, tales como giros, ruido o recortes. De esta manera se consigue que dichos datos sean lo más óptimos posibles para el entrenamiento de la red, el cual se desarrollará en el siguiente apartado.

Sin embargo, aunque ya se han descargado todas las imágenes del entrenamiento de la red, esta no será precisa, o directamente no será capaz de funcionar si antes no se le indica dónde se encuentra el objeto a detectar en cada imagen, y de que clase se trata. De esta manera el algoritmo de entrenamiento podrá saber para cada iteración si su predicción ha sido acertada o no, para así cambiar su decisión en la siguiente iteración.

En este caso nuestras anotaciones o etiquetas deberán ser redactadas en formato YOLO, mencionado anteriormente en documento. Por suerte, la previamente mencionada página web también ofrece herramientas de etiquetado en diferentes formatos, uno de ellos el necesario para el conjunto de imágenes del proyecto. Esto facilita el trabajo y hace que no se tengan que etiquetar una a una las imágenes de forma manual. Aún así, para estar completamente seguros de que tales etiquetas no son erróneas, se utiliza la herramienta LabelImg, la cual, ejecutada mediante Python, permite introducir la carpeta deseada y comprobar la validez de las etiquetas.

Las imágenes y sus respectivas etiquetas, que a su vez reciben el mismo nombre que la imagen a la que representan, son clasificadas en dos carpetas comprimidas, llamadas "obj.zip" y "test.zip". Como se puede apreciar, ambas representan el conjunto de datos de entrenamiento y test respectivamente.

Ambos nombres de carpetas son importantes ya que desde el algoritmo de entrenamiento se referirán a ellas por sendos nombres, aunque también cabría la posibilidad de cambiar manualmente tal configuración.

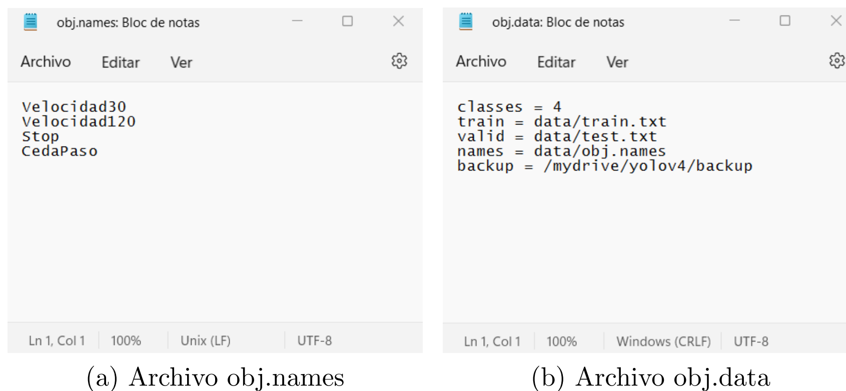
4.1.2. Entrenamiento de la red

Tras haber configurado el conjunto de datos, la red neuronal diseñada en el algoritmo YOLO está lista para ser entrenada.

El primer paso seguido para dicho entrenamiento es la búsqueda de un sistema o framework dentro del cual poder realizar este entrenamiento. Como se mencionó en apartados anteriores, los frameworks más famosos para entrenar una red basada en YOLOv4 son PyTorch y Darknet. Este proyecto se ha desarrollado utilizando Darknet debido a la facilidad de su uso y búsqueda de información en la red. Este framework se obtuvo en este caso mediante GitHub.

Entrenar una red neuronal puede llevar horas e incluso días, dependiendo de la complejidad de los cálculos que realice la máquina, la cantidad de imágenes entre las que se itere y la potencia gráfica de dicha máquina. Es por esto que se decidió usar una máquina externa, la cual tuviera mayor cantidad de potencia de GPU que un ordenador portátil común. Se ha elegido Google Colab como el lugar idóneo para entrenar la red, lo que permite un acceso a una memoria y potencia de cómputo mucho más grandes, así como el fácil acceso a los archivos guardados en carpetas de Google Drive.

Una vez definido el entorno en el que va a trabajar, se procede a incluir dos archivos esenciales para el entrenamiento, denominados "obj.names" y "obj.data" respectivamente, en una misma carpeta, dentro de la cual se desarrollará el entrenamiento del modelo. El primero contiene el nombre de una de las clases a utilizar escrito en cada línea, en este caso cuatro líneas de texto con las palabras: Velocidad30, Velocidad120, Stop y CedaPaso. El segundo, contiene información esencial para el algoritmo YOLO, específicamente el número de clases del modelo, la ruta a los archivos "train.txt" y "test.txt" (los cuales contienen la ruta de cada imagen del conjunto de datos), la ruta del archivo con extensión .names antes mencionado, y la ruta de la carpeta de backup, donde se volcarán los archivos resultantes del entrenamiento.



(a) Archivo obj.names (b) Archivo obj.data

Figura 14: Contenido de los archivos .name y .data del modelo

En la misma carpeta, se incluirán las carpetas .zip las cuales contienen el set de imágenes y sus respectivas etiquetas, así como dos scripts escritos en python, los cuales serán los encargados de obtener y guardar los archivos "train.txt" y "test.txt" mencionados anteriormente.

Una vez preparada la carpeta de Drive en la cual se entrenará a la red, se escribió el código en una libreta de Google Colab. De esta manera será mucho más sencillo analizar cada una de las ejecuciones que se llevan a cabo en el script, con la finalidad de que todas funcionen correctamente.

Lo primero que habremos de hacer será montar Google Drive en Google Colab, para que, de esta manera, podamos acceder a los archivos que previamente han sido guardados en una carpeta de Drive. Esto viene dado mediante la importación de la librería "google.colab.drive" en la libreta, para después, mediante la función "mount()", poder montar satisfactoriamente Drive.

El siguiente paso es clonar Darknet desde su repositorio en GitHub, por lo que "git clone" permitirá crear la carpeta llamada darknet, dentro de la carpeta del proyecto. Una vez dentro de la carpeta de darknet, se modifica su Makefile para habilitar el uso de GPU. Después se verifica que tenemos la última versión de CUDA instalada, la cual nos permitirá obtener un rendimiento de la GPU aún mayor gracias a su forma de computación paralela. Con las verificaciones completadas, se ejecuta el comando "!make", para así completar la creación del ejecutable de Darknet, el cual servirá para dar la orden de entrenamiento de la red más adelante.

Para facilitar la visualización de imágenes, el siguiente paso consiste en crear una función de ayuda llamada "imShow()", basada en la función "imshow()"

de OpenCV, que, combinada al módulo pyplot de matplotlib, generará la imagen directamente en el formato y tamaño deseados.

En este momento, ya se puede contar con un framework sólido para ejecutar el entrenamiento de la red, pero para que este surja efecto, hay algunos pasos más que dar con respecto a la configuración del propio algoritmo YOLO, así como de los archivos y carpetas del conjunto de datos.

Primero, Darknet necesita el conjunto de datos en la carpeta "darknet/data/", por lo que copiaremos las carpetas comprimidas de entrenamiento y test a la mencionada carpeta, para posteriormente descomprimirlas, lo que dejaría la carpeta "data" con dos subcarpetas: "obj" y "test".

El siguiente paso tiene que ver directamente con la configuración del algoritmo YOLO, y la estructura de la red neuronal. Por ello, tras copiar el archivo llamado "yolov4-custom.cfg" desde la carpeta "darknet/cfg/" a la carpeta principal del proyecto, se cambiarán distintos aspectos de este archivo mediante cualquier editor de texto:

batch = 32: Se reduce el número de batch de 64 a 32 para así evitar un gran consumo de memoria, aumentando el tiempo de entrenamiento, cosa que no tiene real relevancia para este proyecto. El batch se define como el número de ejemplos en los que se ejecutará la propagación hacia adelante, antes de actualizar los parámetros.

subdivisions = 16: Las subdivisiones en una red neuronal se refieren a la técnica de dividir una red neuronal en partes más pequeñas, cada una de las cuales se entrena y optimiza de forma individual. Esto es importante en redes profundas como la que utiliza YOLO, por lo que permitirá mejorar la eficiencia y acelerar el proceso de entrenamiento, además de reducir el sobreajuste.

max_batches = 8000: El número máximo de batch en la red se define como el número de clases de esta, multiplicado por 2000, en este caso dando un valor de 8000 para un entrenamiento óptimo estándar.

steps = 6400, 7200: Al igual que el número máximo de batch, el número máximo de steps también viene dado por una fórmula estándar que nos permite tener una visión del valor óptimo para nuestra red, en este caso de el 80 % y 90 % del número máximo de batch. Estos steps se podrían definir como la cantidad de veces que se actualizan los pesos y sesgos de la red, mientras se procesan los datos de entrenamiento. Sus dos valores definen los valores máximo y mínimo entre los cuales se moverá, y su cambio dependerá del comportamiento de la red durante el entrenamiento.

filters = 27: De la misma manera que el número máximo de batch, así como de los steps, el número de filtros de las capas importantes de la red viene dado por otra fórmula, la cual indica que para un correcto funcionamiento de la red, el parámetro "filters" debe ser el resultado de multiplicar por 3 el número de clases, al que previamente se le suma 5. Para nuestra red, 27 será el número de filtros de las capas más importantes. Como ya se explicó previamente, estos filtros son los encargados del aprendizaje de la red, si esta es de tipo convolucional.

También es posible cambiar el ancho y alto de los datos de entrada mediante los parámetros "width" y "height", pero como estos coinciden con el ancho y alto de las imágenes del conjunto de datos, no se modificarán.

Como último cambio en el archivo de configuración, se editó el valor de "learning_rate" a 0.0001, debido a que con su anterior valor, 0.001, el aprendizaje era algo menos preciso. De esta manera conseguiremos más precisión, sacrificando algo de tiempo de entrenamiento, lo cual no es de gran importancia para el desarrollo del proyecto y se trata de un cambio asequible.

Tras realizar estos cambios, el archivo de configuración se vuelve a copiar a la carpeta de configuración de YOLO bajo el nombre de "yolov4-obj.cfg", para acto seguido copiar los archivos "obj.data" y "obj.names" a la carpeta /data.

Para que el algoritmo YOLO funcione, se deben especificar las rutas a todas las imágenes del conjunto de datos, tanto los de entrenamiento como los de test. Esto se lleva a cabo mediante el uso de dos documentos de texto: "train.txt" y "test.txt". Ya que la escritura manual de estos archivos es casi imposible teniendo en cuenta el número de imágenes del conjunto de datos del proyecto, se han redactado dos programas en Python para hacerlo automáticamente. Mediante la ejecución de "generate_test.py" y "generate_train.py" estos documentos de texto son creados y archivados en su carpeta correspondiente dentro de Darknet.

Al este ser el último paso de la configuración previa al entrenamiento de la red, se comprueba que en la carpeta "darknet/data/" existan todos los archivos creados en los pasos previos de la configuración, para así poder proceder con seguridad al siguiente paso. Estas comprobaciones se realizan mediante el comando "!ls" seguido de la ruta a la carpeta que se compruebe.

Realizados todas las comprobaciones pertinentes previas al entrenamiento, el siguiente paso es ejecutar el propio entrenamiento de la red. Para ello, primero se deben dar permisos para ejecutar el comando ./darknet mediante "!chmod 755 ./darknet". Una vez dados los permisos se ejecuta dicho comando, con

los argumentos que se ilustran en la siguiente figura:

```
[ ]  !./darknet detector train data/obj.data cfg/yolov4-obj.cfg yolov4.conv.137 -dont_show -map
```

Figura 15: Entrenamiento de la red en Darknet

Como se puede observar, los argumentos utilizados tienen que ver con las ubicaciones tanto del archivo de configuración, como del archivo que define la ubicación del conjunto de datos. También se incluyen dos flags, usados para evitar la impresión de todos los datos del entrenamiento, solo los necesarios.

Tras alrededor de 20 horas de entrenamiento, la red neuronal ya ha sido entrenada y Darknet ha creado y almacenado un gráfico que muestra la precisión obtenida a lo largo del tiempo. Este gráfico también puede ser visto desde Google Colab mediante el comando "detector map" dentro de Darknet, pero se almacena en la carpeta general del entorno.

Para poder entender cómo ha sido el proceso de entrenamiento, se analizará esta gráfica y se sacarán las conclusiones pertinentes, repitiendo así el entrenamiento con otra configuración si se estima necesario.

A su vez, el programa también ha almacenado los pesos derivados del entrenamiento en la carpeta backup. Estos pesos serán de gran importancia para ejecutar las predicciones en un futuro.

4.1.3. Extracción y análisis de los resultados

En el siguiente gráfico podemos observar con claridad el comportamiento del entrenamiento a lo largo del tiempo, atendiendo principalmente a dos parámetros: mAP y avg loss.

El primer parámetro, mAP o Mean Average Precision, hace referencia a la precisión media del modelo, la cual se ve reflejada a lo largo del tiempo en color rojo. Esta precisión media define en gran medida la calidad del entrenamiento, y aunque no quiere decir que sea un éxito, si que se puede observar que alcanza un valor del 97.5 %, lo cual es un claro indicio de que ha ido bien.

El segundo parámetro, avg loss, se refiere a la pérdida media en cada momento del entrenamiento, indicada en la gráfica con el color azul. En apartados anteriores se ha explicado el valor de esta pérdida media y lo importante que es para el entrenamiento que sea lo más baja posible al final. Sabiendo esto,

es otro claro indicio de que nuestro entrenamiento ha ido bien, que la función de pérdidas es decreciente y acaba en un valor muy bajo, de 0.38.

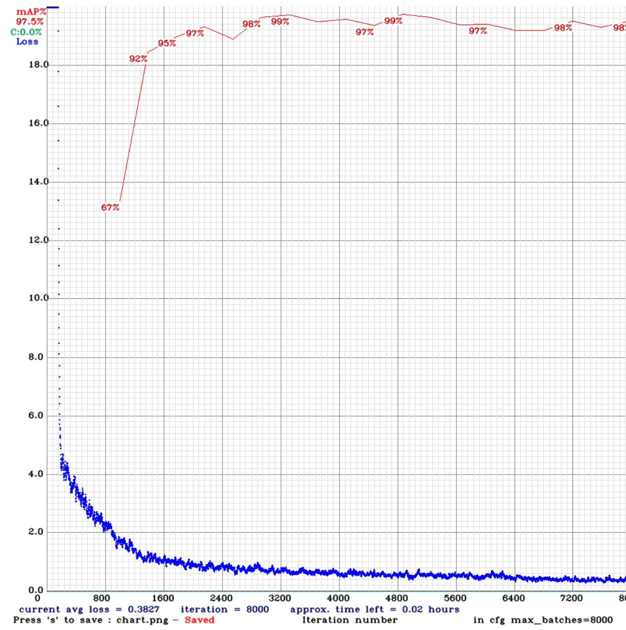


Figura 16: Gráfico del entrenamiento de la red

Si es cierto que ambos parámetros arrojan resultados prometedores para este proyecto, no se puede saber a ciencia cierta si ha ido bien hasta que no se pruebe con un caso real del conjunto de datos de validación.

Para ello, primero se deben cambiar los parámetros ”batch” y ”subdivisions” de sus valores anteriores a 1, para así poder realizar la validación. Una vez hecho esto, se utiliza el comando de darknet ”detector test” junto al archivo final de pesos generado en la carpeta backup y la imagen a predecir. A esto se le puede añadir un valor de threshold, para descartar así las predicciones que sean menores que un valor determinado.

Este test se probó usando dos imágenes distintas, una de ellas mostrando las cuatro señales a la vez claramente diferenciadas, y otra mostrando una imagen real de la calle. Las predicciones fueron las siguientes:

Figura 17: Imagen compuesta

En la imagen compuesta por las cuatro señales de tráfico, podemos observar que las predicciones son excelentes, todas por encima del 95 % de seguridad en la decisión. Esto muestra que el modelo funciona y los datos obtenidos en el gráfico tienen sentido en la validación.

Figura 18: Imagen real

En la imagen real se puede observar que la señal de stop tiene un 100 % de seguridad, mientras que la señal de ceda el paso tiene un 49 % aunque no se aprecie en la imagen. Esto se debe principalmente al tamaño de las señales en la imagen, así como a la calidad de esta. A su vez también podemos ver

que detecta la señal de prohibido girar a la derecha como una señal de 30, esto se debe a su parecido tanto en forma como en color y tamaño.

Tras analizar los resultados del entrenamiento, podemos afirmar que ha sido un éxito, ya que, pese a algunos fallos o imperfecciones en la validación en un caso real, ha sabido comportarse con mucha precisión tanto en la predicción como en el lugar en el que se encuentra la señal. Dichas imperfecciones son por otro lado normales si se tiene en cuenta el tamaño del conjunto de datos con el que se entrenó la red. Con una cantidad más grande de datos y con más variedad de ellos, el resultado sería mejor.

4.2. Configuración del vehículo

Los primeros pasos para el correcto uso de PiRacer Pro son, por un lado la configuración de RaspBerry, y por otro lado el calibrado y pruebas de funcionamiento de los motores y cámara del vehículo. Si estas comprobaciones son satisfactorias, se procederá a la implementación del código.

4.2.1. Configuración de RaspBerry Pi

La placa de desarrollo RaspBerry requiere de una configuración inicial básica para su uso correcto. Esta configuración inicial, la cual está disponible en la página oficial de RaspBerry comienza mediante la descarga y el posterior quemado del sistema operativo RaspBian en una tarjeta micro-SD. Dicha tarjeta es posteriormente introducida en su correspondiente ranura de la placa, y de esta forma, cuando se de corriente al sistema, RaspBerry utilizará el sistema operativo almacenado en la tarjeta para su inicio.

Tras el arranque del sistema operativo, se debe acceder a la interfaz de configuración de RaspBerry mediante el uso del comando ”sudo raspi-config”. En esta interfaz se activarán distintos tipos de configuración, como el uso de cámara, I2C, o el uso completo de la memoria, el cual viene desactivado por defecto. Una vez hecho esto, es recomendable el reinicio del sistema.

El siguiente paso de la configuración es la instalación de los paquetes necesarios para el proyecto. Dichos paquetes son enumerados por DonkeyCar en su página web. Entre ellos se encuentran Python3, Numpy, virtualenv, OpenCV, etc.

Por último, se clonará DonkeyCar desde GitHub, para así poder hacer uso de él, pero antes es indispesable crear un entorno virtual mediante el comando ”python3 -m virtualenv -p python3 env –system-site-packages”, para así poder trabajar en un sistema aislado y poder instalar correctamente cuantos

paquetes sean necesarios. Una vez dentro del entrorno virtual, se procede a clonar la página de DonkeyCar, e instalar los paquetes restantes necesarios.

Con esta configuración inicial, Raspberry está lista para su uso en el proyecto, sin embargo, es imperativo el calibrado del vehículo para su correcto uso más tarde.

4.2.2. Calibrado de dirección y aceleración

Como se explica en la web oficial del proyecto DonkeyCar, el calibrado es indispensable para que el coche circule de forma homogénea y correcta. El calibrado consistirá en la búsqueda de los valores máximos y mínimos de la dirección del coche, así como en la selección de los valores máximos y mínimos de la aceleración, a gusto del usuario.

Ambos calibrados se realizan de la misma forma, mediante el uso del comando mostrado en la siguiente figura:

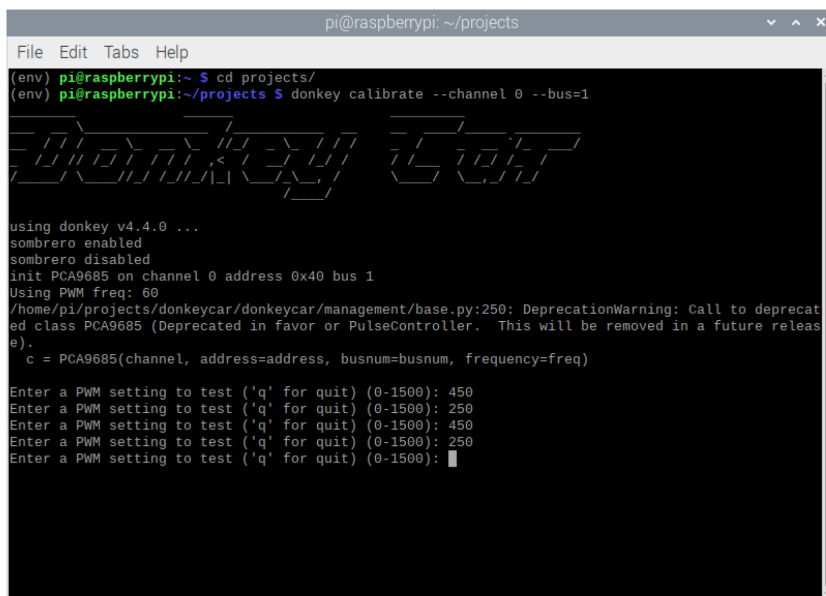


Figura 19: Ejemplo de calibrado de dirección

En el caso de este proyecto, los canales de calibrado de dirección y aceleración son 0 y 1 respectivamente, y el bus I2C es el número 1. Este comando abrirá un apartado en el terminal, dentro del cual se introducen valores PWM del servomotor, buscando los valores máximos y mínimos. Es importante para la salud del servomotor, no introducir valores en la configuración final, los

cuales superen el rango de movimiento del motor, ya que podría suponer un desgaste innecesario para el mismo.

Los valores obtenidos tras la calibración son de 450 y 250 para la dirección, y de 420, 340 y 260 para los valores máximos de marcha adelante, parado y marcha atrás respectivamente. Dichos valores son posteriormente incluidos en el archivo de configuración "myconfig.py", en el cual se descomentan las líneas de código que el usuario quiera cambiar, mientras que las comentadas toman los valores por defecto. En la siguiente figura se pueden apreciar dichos cambios:

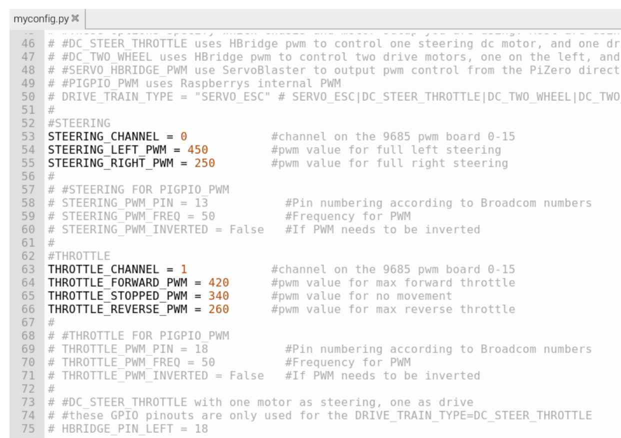


Figura 20: Calibrado incluído en myconfig.py

4.3. Prueba de funcionamiento

La prueba de funcionamiento de el presente proyecto consta de tres partes principales y diferenciadas entre sí. Por un lado, se pondrá a prueba la conducción autónoma del vehículo por el circuito cerrado. Por otro lado, se mostrará el reconocimiento de señales de tráfico mediante el modelo entrenado, ejecutado en el vehículo. Por último, se integrarán ambas partes en el vehículo y se procederá a su ejecución simultánea.

4.3.1. Conducción autónoma por circuito

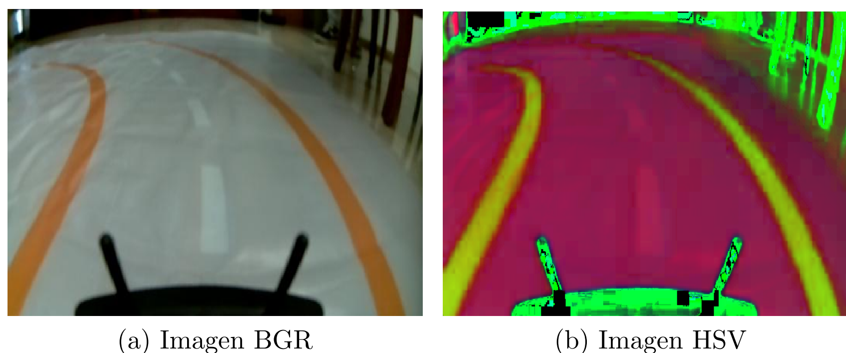
La conducción autónoma por circuito cerrado se realizará haciendo uso del circuito ovalado que se incluye en el kit de PiRacer Pro. El diseño de dicho circuito es de fondo gris, con línea discontinua blanca en el centro del carril, y líneas continuas naranjas a los lados.

Como primera opción para dicha conducción autónoma, el vehículo sería

entrenado mediante redes neuronales para que, mediante el uso de imágenes del trazado, pudiera seguir el camino idóneo. Esta opción requiere de una capacidad de cómputo que por desgracia RaspBerry Pi no puede ofrecer, ya que sería imposible poder realizar la ejecución del modelo de conducción autónoma junto con el de reconocimiento de señales de tráfico, entrenado anteriormente. Es por ello que finalmente se ha decidido el uso de filtros HSV e identificación de contornos para reducir el coste computacional del proceso.

Los filtros HSV son muy utilizados en visión artificial. Su funcionamiento se basa en la transformación de una imagen BGR (Azul, Verde, Rojo, por sus siglas en inglés) al formato HSV, siglas de "Hue" (matiz), "Saturation" (saturación) y "Value" (valor), que son tres parámetros que describen el color de un píxel en una imagen. Mediante el uso de una máscara compuesta por los valores máximos y mínimos de cada uno de los componentes del formato HSV, se aíslan los objetos de colores deseados en una imagen binaria resultante en blanco y negro. Los objetos del color deseado aparecerán en blanco, mientras que los no deseados aparecerán en negro.

Para la selección de los valores HSV ideales del contorno del circuito, se ha escrito un breve programa de nombre "filtro.py", el cual, mediante el uso de barras de seguimiento o track bars, permite ajustar de forma manual dichos valores, para así ajustarlos al color específico del circuito. En la siguiente imagen se puede apreciar el cambio de aspecto que se da al cambiar una imagen de formato BGR a HSV:



(a) Imagen BGR (b) Imagen HSV

Figura 21: Transformación de una imagen BGR a formato HSV

Como se puede observar en las imágenes, el contorno del circuito se diferencia perfectamente en formato HSV. El siguiente paso es, como ya se ha mencionado anteriormente, buscar los valores ideales HSV. En la siguiente figura

podemos observar los valores finales que han logrado diferenciar el circuito en la imagen final binaria:

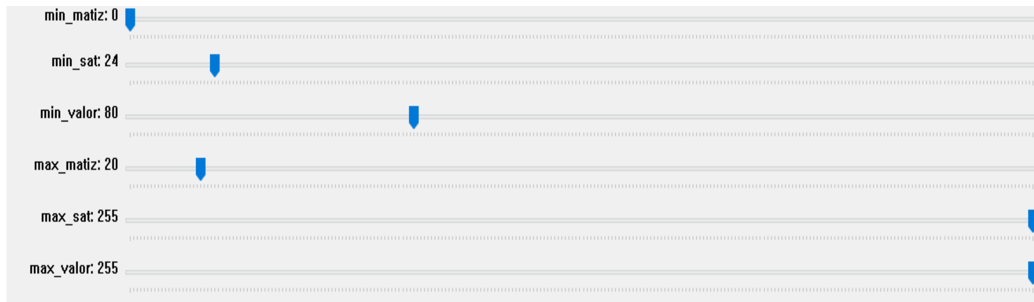
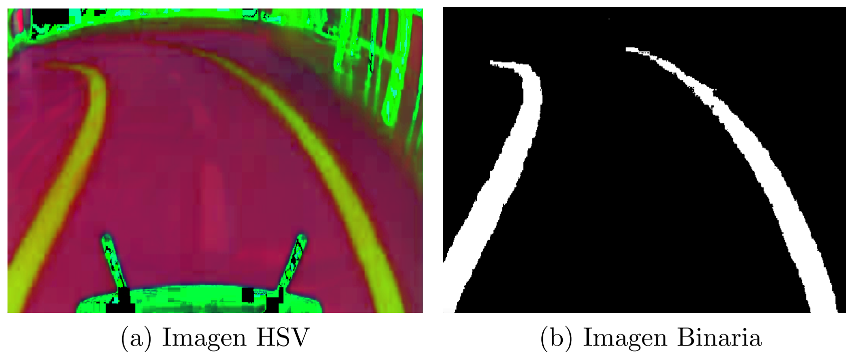


Figura 22: Búsqueda de los valores ideales HSV mediante trackbars

Los valores elegidos han sido, en mínimos y máximos, (0, 20) para el matiz, (24, 255) para la saturación, y (80, 255) para el valor. Utilizando estos valores como máscara en la imagen HSV, se obtiene la imagen binaria resultante, comparada frente a la imagen HSV en la siguiente figura:



(a) Imagen HSV (b) Imagen Binaria

Figura 23: Implementación del filtro HSV

Una vez obtenida la imagen binaria, en la cual se destacan perfectamente los límites del circuito, el siguiente paso es calcular el ángulo de desviación del vehículo en cada instante, usándolo como input del controlador de dirección.

Para ello, y para que DonkeyCar pueda funcionar de la mejor forma posible, lo más adecuado es crear una parte o módulo dentro de dicho framework, para así más tarde cargar dicho módulo en un template de DonkeyCar.

El módulo a crear, llamado "steering_angle.py", estará formado por tres clases principales, las cuales se encargarán de obtener la imagen de la cámara,

calcular el ángulo de giro a partir de dicha imagen, y si es necesario mostrar las imágenes en pantalla.

La primera clase, PiCamera(), es una clase obtenida directamente de otro módulo propio de Donkeycar, la cual se encarga de gestionar la obtención de imágenes en tiempo real de la cámara de RaspBerry, llamada PiCam. Dicha clase devuelve a la salida cada frame obtenido.

La segunda clase, SteeringAngle(), toma como entrada una imagen, dada por la anterior clase, y reproduce los pasos vistos anteriormente en este apartado, aunque a diferencia de hacerlo con una imagen completa, se realiza la transformación a una imagen recortada de la original, de tamaño 518x40. Dicho tamaño ha sido definido tras la experimentación del comportamiento del vehículo con otros tamaños, y su importancia reside en la facilidad de la identificación de contornos en un espacio reducido, seleccionando únicamente una porción de la mitad de la imagen.

Con la imagen binaria recortada, se realizan operaciones de identificación de contornos, haciendo uso de las herramientas de OpenCV, librería la cual contiene funciones tales como "cv2.findContours()", que seleccionan y devuelven el borde de los objetos en blanco de la imagen, filtrada anteriormente. Estos contornos son filtrados por área mediante la función "cv2.contourArea()", para así evitar la confusión puntual de otros contornos más grandes que los del circuito. Después, cada contorno es suavizado para evitar ruido excesivo, y se obtienen los momentos de dicho contorno. Los momentos de una imagen, sirven para calcular muchas de sus características, como su centro, área, perímetro, etc. Haciendo uso de estos momentos, se obtiene el centroide de cada uno de los contornos, representados por sus cordenadas x e y.

De los centroides obtenidos, se calcula el centro entre dichos centroides en el frame original, sin recortar. La diferencia entre el centro de la imagen y el centro entre los contornos, será la distancia que tendrá que corregir el vehículo. Podemos apreciar estos cálculos en la figura a continuación:

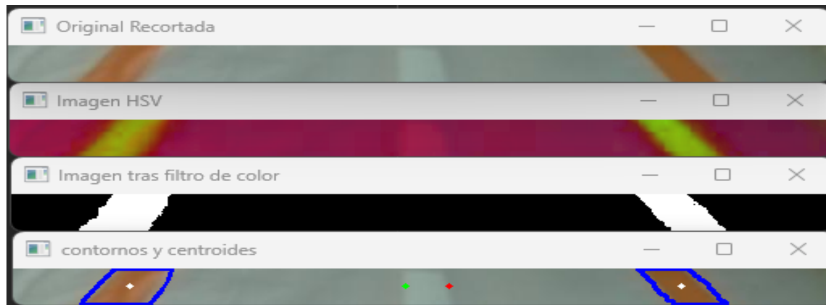


Figura 24: Centro y centroide de los contornos, obtenido a partir de la imagen inicial recortada

Para obtener el ángulo final, se ha de definir un punto de referencia desde el cual calcularlo, el cual ha sido de una distancia de 230 píxeles desde la parte superior de la imagen, y de la mitad del ancho de la misma. El cálculo de dicho ángulo se puede apreciar en la siguiente figura:

Figura 25: Contornos y ángulo obtenido a partir de los mismos

La tercera y última clase utilizada en el módulo, se encarga de mostrar las imágenes que se han plasmado en este documento. Dicha clase está basada en OpenCV, y utiliza sus funciones "cv2.imshow()" y "cv2.destroyAllWindows()", para mostrar y destruir los frames de la cámara.

En el template "demo.py", creado para albergar el módulo anteriormente explicado, se utilizará también el módulo controlador de la dirección y aceleración, para que, a partir de un ángulo dado, el servomotor gire correspondientemente. De esta manera, el modelo de conducción autónoma es puesto en práctica en el circuito, en el cual se obtienen resultados satisfactorios.

4.3.2. Integración del modelo de detección de señales

La ejecución del modelo de detección de señales de tráfico en PiRacer Pro, ha sido implementada mediante la librería OpenCV, ya utilizada anteriormente en el proyecto. Esta librería ofrece muchas utilidades para la carga de modelos ya entrenados de redes neuronales en distintos formatos, siendo uno de ellos YOLO, el cual se utilizó en el entrenamiento de la red del proyecto.

En un ordenador convencional, haciendo uso de tarjeta gráfica y procesador, el comportamiento de OpenCV para el reconocimiento de objetos es aceptable, obteniendo una velocidad de alrededor de 3 FPS, llegando a 5 en algunas ocasiones concretas. Sin embargo, teniendo en cuenta las limitaciones de la placa de desarrollo utilizada en el proyecto, no cabe duda de que el rendimiento será mucho más bajo que en un ordenador al uso.

Esta hipótesis fue puesta en práctica, ejecutando un script de python el cual, mediante el uso de OpenCV, carga los pesos del modelo entrenado, para más tarde analizar cada frame obtenido directamente de la cámara del vehículo, en busca de objetos que se identifiquen con las 4 clases de señales de tráfico a detectar. Desgraciadamente, la hipótesis planteada era correcta, y el rendimiento fue de 0.037 FPS, lo que se podría traducir en 27 segundos por cada frame analizado.

Para buscar una solución al problema, se plantearon diferentes propuestas, entre las cuales se destacan las siguientes:

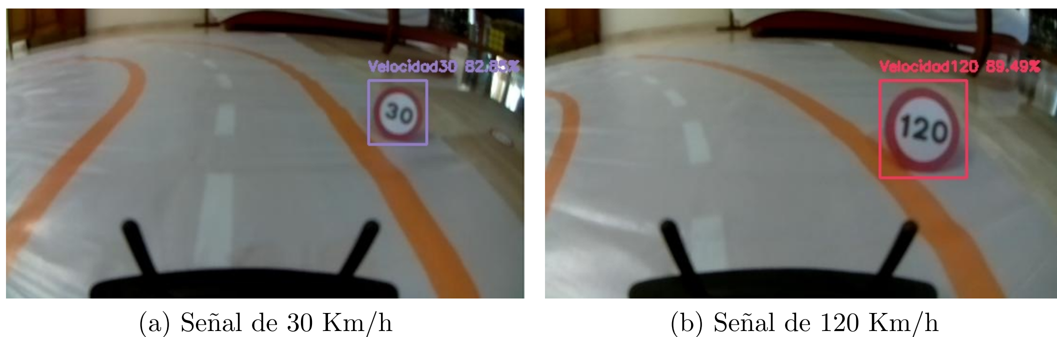
Como primera opción, buscar la manera de hacer que OpenCV analice las imágenes más rápido, mediante algún tipo de optimización de código, que permita hacer menos cálculos por segundo.

Como segunda opción, buscar otra librería o forma de ejecutar el modelo, la cual sea compatible con los pesos y el formato del modelo entrenado, obteniendo así un mejor rendimiento del mismo.

Ninguna de las opciones antes mencionadas ha dado el resultado que se esperaba por distintos factores. Se ha probado una librería derivada de OpenCV, desarrollada por el MIT, llamada Cvlib, la cual contiene más facilidades para su uso con YOLO. Desgraciadamente su rendimiento no cambia en gran medida en RaspBerry, aportando unos pocos segundos menos por frame. A su vez, se buscó la forma de cambiar los pesos entrenados en formato YOLO, a otros formatos que hicieran más fácil su ejecución, como podrían ser Keras o Tensorflow. Estos cambios de formato no son desarrollados por páginas oficiales y su rendimiento o incluso funcionamiento no son correctos para el modelo de este proyecto. Los cambios de formato en las páginas oficiales solo

son compatibles con Tensorflow 1.x, el cual ya está en desuso y no es posible trabajar con él, al menos no de forma correcta, haciendo uso de todas sus funciones.

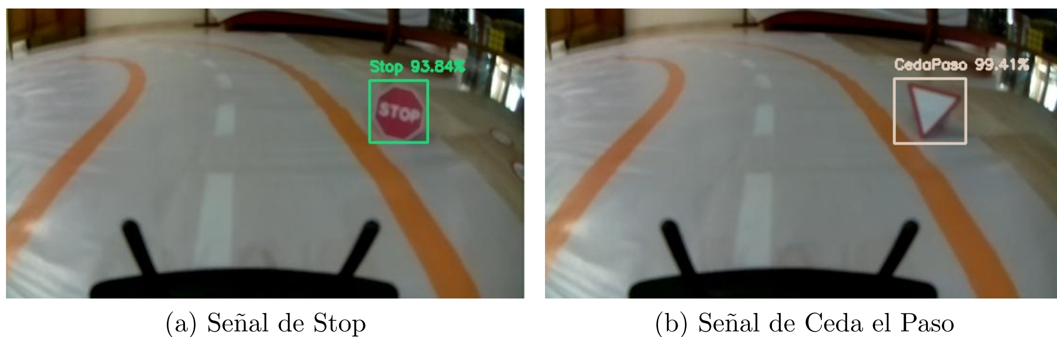
Pese a no haber dado con la solución a este problema de rendimiento, se ha buscado la forma de poder demostrar el buen funcionamiento de la red neuronal en el vehículo. Para ello se han tomado capturas de vídeo, partidos en frames, del circuito en el cual se han ido posicionando las señales de tráfico una a una. Se han extraído dichos frames y han sido analizados por el modelo a modo de vídeo. En las siguientes imágenes se puede apreciar la excelente precisión del modelo con las cuatro clases de señales de tráfico:



(a) Señal de 30 Km/h

(b) Señal de 120 Km/h

Figura 26: Detección de señales de 30 y 120 Km/h en circuito



(a) Señal de Stop

(b) Señal de Ceda el Paso

Figura 27: Detección de señales de Stop y Ceda el Paso en circuito

Así como en la explicación dada con anterioridad en este documento, se observa una gran precisión, de más del 90 %, en las señales de Stop y Ceda el paso, y de alrededor del 85 % en las de 30 y 120 Km/h. Esto es debido a

las similitudes en forma y color entre las señales de velocidad, frente a las grandes diferencias con las de Stop y Ceda el Paso.

5. Conclusiones

Tras la finalización del proyecto, se puede afirmar que los principales objetivos han sido cumplidos. El vehículo ha dado resultados satisfactorios en las pruebas de conducción autónoma, y la detección de objetos es totalmente funcional.

Pese a ello, la integración de ambas partes en una misma plataforma no ha sido posible por limitaciones de hardware. Estas limitaciones son fácilmente resolubles en un prototipo futuro, mediante el uso de una placa de desarrollo más potente, con capacidad gráfica.

De esta manera, en un futuro este diseño de detección de objetos podría ser utilizado en casos reales de distintos ámbitos, como pueden ser el deportivo o doméstico.

6. Acrónimos

Los acrónimos utilizados en este TFG se listan a continuación junto a su significado.

Acrónimo	Significado
API	Interfaz de Programación de Aplicaciones
BSD	Berkeley Software Distribution
CNN	Convolutional neural networks.
CPU	Unidad Central de Proceso
CUDA	Compute Unified Device Architecture
FPS	Frames Per Second
FPU	Floating Point Unit
GPIO	General Purpose Input Output
GPU	Graphic Processing Unit
GTSRB	German Traffic Sign Recognition Benchmark
IoT	Internet of Things
LCD	Pantalla de cristal líquido
RAM	Random Access Memory
ReLu	Rectifier Linear Unit
RGB	Red, green and blue
RN	Redes neuronales
RNN	Recurrent neural networks
SSD	Unidad de estado sólido
TanH	Tangente Hiperbólica
TFG	Trabajo Fin de Grado
TPU	Tensor Processing Unit
YOLO	You Only Look Once

Cuadro 1: Acrónimos

Referencias

[Ber18] Fernando Berzal. *Redes Neuronales & Deep Learning*. Edición independiente, 2018.

[Cho18] François Chollet. *Deep Learning with Python*. Manning Publications Co, 2018.

[Don] DonkeyCar. Donkeycar documentation page. https://docs.donkeycar.com/.

[Han13] INI Klinik Hannover. German traffic sign benchmarks, 2013.

[Int16] Intel. Opencv library page. https://opencv.org/, 2016.

[Jol21] Jolle W. Jolles. Broad-scale applications of the raspberry pi: A review and guide for biologists. *Methods in Ecology and Evolution*, 2021.

[lab16] Facebook's AI Research lab. Pytorch main page, 2016.

[Rac20] Pi Racer. Piracer ai kit wiki, 2020.

[Red15] Joseph Redmon. Yolo: Real time object detection, 2015.

[Red16] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[RF18] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[Rob] Roboflow. Roboflow main page. https://universe.roboflow.com/.

[Tor19] Jordi Torres. *Deep Learning, introducción práctica con Keras*. What this Space, 2019.

[VLR12] Miguel Vargas-Lombardo and Iván Rivera. Principios y campos de aplicación en cuda programación paralela y sus potencialidades. *Nexo Revista Científica, pags. 39-46*, 25, 2012.

Anexo 1. Bucle principal ”demo.py”

```
import donkeycar as dk
from donkeycar.vehicle import Vehicle
from donkeycar.parts.steering_angle import PiCamera,
    CvImageDisplay, SteeringAngle
from donkeycar.parts.actuator import PCA9685, PWMSteering,
    PWMThrottle
import time


V = Vehicle()

# Clase sencilla creada para obtener una aceleración
    constante
class MiAcelerador:

  def run(self):
    throttle = 0.43

    return throttle

V.add(MiAcelerador(), outputs = ["aceleracion"])


# Se añade la clase PiCamera, derivada del módulo
    steering_angle
cam = PiCamera()

V.add(cam, outputs = ["camera/image"])


# Se añade la clase SteeringAngle, derivada del módulo
    steering_angle
angle = SteeringAngle()

V.add(angle, inputs = ["camera/image"], outputs = ["camera/
    angle", "angulo"])


# Comentado para evitar la visualización y conseguir más
    velocidad
# disp = CvImageDisplay()
# V.add(disp, inputs = ["camera/angle"])


# Se añaden las clases dedicadas al control de la aceleración
     y dirección
```

```
steering_controller = PCA9685(channel = 0)
steering = PWMSteering(controller=steering_controller)

throttle_controller = PCA9685(channel = 1)
throttle = PWMThrottle(controller = throttle_controller,
   max_pulse = 420, zero_pulse = 340, min_pulse = 260)

V.add(steering, inputs = ["angulo"])
V.add(throttle, inputs = ["aceleracion"])

# Se inicia el bucle de DonkeyCar
V.start()
```

Anexo 2. Módulo de conducción autónoma ”steering_angle.py”

Las líneas de código comentadas deben descomentarse en el caso de necesitar la imagen en tiempo real por pantalla, si no se necesita hacer tal cosa, se dejarán comentadas.

```
import cv2
import time
import math
import numpy

class PiCamera():
    def __init__(self, image_w=544, image_h=304, image_d=3,
   framerate=20, vflip=True, hflip=True):
        from picamera.array import PiRGBArray
        from picamera import PiCamera

        resolution = (image_w, image_h)
        self.camera = PiCamera()
        self.camera.resolution = resolution
        self.camera.framerate = framerate
        self.camera.vflip = vflip
        self.camera.hflip = hflip
        self.rawCapture = PiRGBArray(self.camera, size=
   resolution)
        self.stream = self.camera.capture_continuous(self.
   rawCapture,
            format="bgr", use_video_port=True)

        self.frame = None
        self.on = True
        self.image_d = image_d

        print('PiCamera loaded.. .warming camera')
        time.sleep(2)

    def run(self):
        f = next(self.stream)
        frame = f.array
        self.rawCapture.truncate(0)
        if self.image_d == 1:
            frame = rgb2gray(frame)
        return frame

    def update(self):
```

```
        for f in self.stream:

            self.frame = f.array
            self.rawCapture.truncate(0)

            if self.image_d == 1:
                self.frame = rgb2gray(self.frame)

            if not self.on:
                break

    def shutdown(self):
        self.on = False
        print('Stopping PiCamera')
        time.sleep(.5)
        self.stream.close()
        self.rawCapture.close()
        self.camera.close()

class SteeringAngle():

    # Las líneas comentadas se descomentarán en caso de
   querer obtener una visualización en pantalla
    def run(self, frame):
        cropped_image = frame[130:170, 0:518]
        cropped_HSV = cv2.cvtColor(cropped_image, cv2.
   COLOR_BGR2HSV)
        mask = cv2.inRange(cropped_HSV, (0, 24, 80), (20,
   255, 255))
        contornos,ret = cv2.findContours(mask, cv2.
   RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

        n = 0
        cx0=0
        cx1=0
        cy0=0
        cy1=0

        for c in contornos:

             area = cv2.contourArea(c)

             if area > 600:

                 if n == 0:

                     contorno_suave0 = cv2.convexHull(c)
                     # ~ cv2.drawContours(cropped_image, [c],
    0, (255,0,0), 2)
```

```
                    M = cv2.moments(c)

                    cx0 = int(M["m10"]/ M["m00"])
                    cy0 = int(M["m01"]/ M["m00"])

                    # ~ cv2.circle(cropped_image, (cx0,cy0),
 2, (255,255,255), -1)

                if n == 1:

                    contorno_suave1 = cv2.convexHull(c)
                    # ~ cv2.drawContours(cropped_image, [
contorno_suave1], 0, (255,0,0), 2)

                    M = cv2.moments(c)

                    cx1 = int(M["m10"]/ M["m00"])
                    cy1 = int(M["m01"]/ M["m00"])

                    # ~ cv2.circle(cropped_image, (cx1,cy1),
 2, (255,255,255), -1)

                n = n+1
     angulo_rad = 1
     if cx0 == 0 or cx1 == 0 or cy0 == 0 or cy1 == 0:
         pass

     else:
         Centroide_cropped = (int(((cx0 - cx1)/2)+cx1),
cy0)
         # ~ cv2.circle(cropped_image, Centroide_cropped,
2, (255,255,0), -1)

         # ~ Centro_cropped = (272, cy0)
         # ~ cv2.circle(cropped_image, Centro_cropped, 2,
(0,0,255), -1)

         Centro_completo = (259, (cy0 + 130))
         # ~ cv2.circle(frame, Centro_completo, 2,
(0,0,255), -1)
         Centroide_completo = (Centroide_cropped[0], cy0 +
 130)
         # ~ cv2.circle(cropped_image, Centroide_completo,
 2, (0,255,0), -1)
         Pto_referencia = (259, 230)

         # ~ cv2.line(frame, Pto_referencia,
Centro_completo, (0,0,255), 2)
```

```
            # ~ cv2.line(frame, Pto_referencia,
    Centroide_completo, (0,255,0), 2)

            hipotenusa = math.sqrt((Centro_completo[0] -
    Centroide_completo[0])**2 + (304-Centro_completo[1])**2)
            angulo_rad = math.asin((Centro_completo[0] -
    Centroide_completo[0])/hipotenusa)
            # ~ angulo = float(angulo_rad * 180/math.pi)
            # ~ print('Centro de contornos = ',
    Centro_contornos)
            # ~ print('Centro de imagen = ', Centro_imagen)
            # ~ print('Distancia entre ambos: ', (
    Centro_imagen[0] - Centro_contornos[0]))

            # ~ print(angulo)
        if angulo_rad != None or (frame != None).any():
            return frame, angulo_rad

class CvImageDisplay():

    def run(self, image):
        if (image != None).any():
            cv2.namedWindow('frame', cv2.WINDOW_NORMAL)
            cv2.imshow('frame', image)
            cv2.waitKey(1)

    def shutdown(self):
        cv2.destroyAllWindows()
```

Anexo 4. Búsqueda de filtro mediante trackbars "filtro.py"

```
import cv2


def do_nothing(x):
    pass


# Creación de la pestaña de Trackbars
cv2.namedWindow('Track Bars', cv2.WINDOW_NORMAL)

cv2.createTrackbar('min_matiz', 'Track Bars', 0, 255,
   do_nothing)
cv2.createTrackbar('min_sat', 'Track Bars', 0, 255,
   do_nothing)
cv2.createTrackbar('min_valor', 'Track Bars', 0, 255,
   do_nothing)


cv2.createTrackbar('max_matiz', 'Track Bars', 0, 255,
   do_nothing)
cv2.createTrackbar('max_sat', 'Track Bars', 0, 255,
   do_nothing)
cv2.createTrackbar('max_valor', 'Track Bars', 0, 255,
   do_nothing)

# Creación de la pestaña de Imagen Original
image_BGR = cv2.imread('foto1.jpg')

cv2.namedWindow('Imagen Original', cv2.WINDOW_NORMAL)
cv2.imshow('Imagen Original', image_BGR)

# Pasamos la imagen original a formato HSV
image_HSV = cv2.cvtColor(image_BGR, cv2.COLOR_BGR2HSV)

cv2.namedWindow('Imagen HSV', cv2.WINDOW_NORMAL)
cv2.imshow('Imagen HSV', image_HSV)

while True:

    # Se utilizan las trackbars para conseguir el filtro
   correcto
    min_matiz = cv2.getTrackbarPos('min_matiz', 'Track Bars')
    min_sat = cv2.getTrackbarPos('min_sat', 'Track Bars')
    min_valor = cv2.getTrackbarPos('min_valor', 'Track Bars')
```

```
    max_matiz = cv2.getTrackbarPos('max_matiz', 'Track Bars')
    max_sat = cv2.getTrackbarPos('max_sat', 'Track Bars')
    max_valor = cv2.getTrackbarPos('max_valor', 'Track Bars')

    mask = cv2.inRange(image_HSV, (min_matiz, min_sat,
   min_valor), (max_matiz, max_sat, max_valor))

    cv2.namedWindow('Imagen Binaria con mascara', cv2.
   WINDOW_NORMAL)
    cv2.imshow('Imagen Binaria con mascara', mask)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break


cv2.destroyAllWindows()

# Imprime en pantalla los valores finales tras el ajuste
print('min_matiz, min_sat, min_valor = {0}, {1}, {2}'.format(
   min_matiz, min_sat, min_valor))
print('max_matiz, max_sat, max_valor = {0}, {1}, {2}'.format(
   max_matiz, max_sat, max_valor))
```

Anexo 5. Entrenamiento de la red en Google Colab

"Yolo_training_TFG"

Empezamos montando drive en Google Colab

```
%cd ..
from google.colab import drive
drive.mount('/content/gdrive')

!ln -s /content/gdrive/My\ Drive/ /mydrive

%cd /mydrive/yolov4/darknet/
```

Se clona darknet desde github y se cambia el makefile para habilitar el uso de GPU

```
!git clone https://github.com/AlexeyAB/darknet
```

```
%cd darknet
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile
```

Se verifica que tenemos la última versión de CUDA, y mediante make, se instala darknet para entrenar más tarde nuestro modelo

```
!/usr/local/cuda/bin/nvcc --version
```

```
!make
```

Se define una función de ayuda al display de imágenes

```
def imShow(path):
  import cv2
  import matplotlib.pyplot as plt
  %matplotlib inline

  image = cv2.imread(path)
  height, width = image.shape[:2]
  resized_image = cv2.resize(image,(3*width, 3*height), interpolation = cv2.INTER_CUBIC)

  fig = plt.gcf()
  fig.set_size_inches(18, 10)
  plt.axis("off")
  plt.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
  plt.show()
```

```
!ls /mydrive/yolov4
```

Aqui empieza la configuración de los archivos YOLO

```
!cp /mydrive/yolov4/obj.zip ../
!cp /mydrive/yolov4/test.zip ../
```

Se extraen las imágenes de los archivos comprimidos obj.zip y test.zip en la carpeta /data

```
!unzip ../test.zip -d data/
```

```
!unzip ../obj.zip -d data/
```

Tras copiar a nuestra carpeta y cambiar la configuración del .cfg, se devuelve a la carpeta de configuracion:

```
!cp cfg/yolov4-custom.cfg /mydrive/yolov4/yolov4-obj.cfg
```

```
!cp /mydrive/yolov4/yolov4-obj.cfg ./cfg
```

Se copian los archivos restantes a su lugar correspondiente, y se crean train.txt y test.txt

```
!cp /mydrive/yolov4/obj.names ./data
```

```
!cp /mydrive/yolov4/obj.data ./data
```

```
!cp /mydrive/yolov4/generate_train.py ./
```

```
!cp /mydrive/yolov4/generate_test.py ./
```

```
!python generate_train.py
```

```
!python generate_test.py
```

```
# verificación
!ls data/
```

Descargamos de github los pesos preentrenados de las capas convolucionales de la red YOLOV4 para hacer más fácil el entrenamiento

```
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.conv.137
```

Se ejecuta el entrenamiento a partir de los pesos descargados de YOLO y nuestros datos

```
#Damos permisos para ejecutar el entrenamiento
!chmod 755 ./darknet
```

```
!./darknet detector train data/obj.data cfg/yolov4-obj.cfg yolov4.conv.137 -dont_show -map
```

```
#En el caso de que se quiera seguir desde un punto ejecutar esta línea con el path a los últimos pesos (weights)
!./darknet detector train data/obj.data cfg/yolov4-obj.cfg /mydrive/yolov4/backup/yolov4-obj_last.weights -dont_show -map
```

```
#El entrenamiento genera una gráfica del entrenamiento, también podemos observarla mediante el siguiente comando
!./darknet detector map data/obj.data cfg/yolov4-obj.cfg /mydrive/yolov4/backup/yolov4-obj_final.weights
```

Se cambia a modo de test, y se hace la comprobación con una imagen aleatoria

```
%cd cfg
!sed -i 's/batch=64/batch=1/' yolov4-obj.cfg
!sed -i 's/subdivisions=16/subdivisions=1/' yolov4-obj.cfg
%cd ..
```

```
!./darknet detector test data/obj.data cfg/yolov4-obj.cfg /mydrive/yolov4/backup/yolov4-obj_best.weights /mydrive/yolov4/Imagenes_test/2881558.jpg
imShow('predictions.jpg')
```

Anexo 6. Script provisto por CVlib para pruebas de velocidad en tiempo real: ”yolo_custom_weights_inference_webcam.py”

```
# author: Arun Ponnusamy

# object detection with yolo custom trained weights
# usage: python3 yolo_custom_weights_inference.py <yolov3.
   weights> <yolov3.config> <labels.names>

# import necessary packages
import cvlib as cv
from cvlib.object_detection import YOLO
import cv2
import sys
import time

weights = sys.argv[1]
config = sys.argv[2]
labels = sys.argv[3]

# open webcam
webcam = cv2.VideoCapture(0)

if not webcam.isOpened():
    print("Could not open webcam")
    exit()

yolo = YOLO(weights, config, labels)

# loop through frames
while webcam.isOpened():
    start = time.time()
    # read frame from webcam
    status, frame = webcam.read()

    if not status:
        print("Could not read frame")
        exit()

    # apply object detection
    bbox, label, conf = yolo.detect_objects(frame)

    print(bbox, label, conf)

    # draw bounding box over detected objects
    yolo.draw_bbox(frame, bbox, label, conf, write_conf=True)
```

```
    # display output
    cv2.imshow("Real-time object detection", frame)

    end = time.time()

    # press "Q" to stop
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

    print(end - start)

# release resources
webcam.release()
cv2.destroyAllWindows()
```

ANEXO 7. RELACIÓN DEL TRABAJO CON LOS OBJETIVOS DE DESARROLLO SOSTENIBLE DE LA AGENDA 2030

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**			X	
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.		X		
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Descripción de la alineación del TFG/TFM con los ODS con un grado de relación más alto:

Tras un análisis de los procesos llevados a cabo durante el desarrollo del presente Trabajo de Fin de Grado, se ha decidido incluir los siguientes Objetivos de Desarrollo Sostenible como los más altos en los cuales el trabajo pretende influir:

ODS 9. **Industria, innovación e infraestructuras:**

Durante el desarrollo del proyecto, se ha incorporado tecnología muy utilizada en la sociedad hoy en día, siendo la detección de señales de tráfico una técnica clave para las próximas generaciones de vehículos autónomos. A su vez, se ha trabajado con un prototipo de dicha conducción autónoma en circuito cerrado.

De esta manera se ha contribuido al desarrollo de tecnología innovadora, la cual tendrá mucha repercusión en el futuro cercano.

ODS 11. **Ciudades y comunidades sostenibles**:

Mediante el uso de tecnología como la utilizada en el proyecto, se podrá desarrollar una red de vehículos autónomos eléctricos de uso público, mediante los cuales se reduciría la contaminación y los accidentes humanos en las ciudades.

Todo el proyecto se ha realizado mediante el uso de vehículos eléctricos, los cuales funcionan haciendo uso de baterías recargables. Se ha evitado el uso de vehículos a gasolina o cualquier otro tipo de combustible fósil.