APLICACIÓN DE INTELIGENCIA AMBIENTAL PARA LA PLATAFORMA DE CÁMARAS INTELIGENTES INTEOX

AMBIENT INTELLIGENCE APPLICATION FOR INTEOX SMART CAMERA PLATFORM

TRABAJO FIN DE GRADO
CURSO 2021-2022

AUTORES

IKER GONZALEZ GONZALEZ

LUIS SÁNCHEZ CAMACHO

DIRECTORES

JUAN ANTONIO RECIO GARCÍA

CARLOS GARCÍA SÁNCHEZ

COLABORADOR

CARLOS GROSSOCORDON PEREZ

GRADO EN INGENIERÍA DE COMPUTADORES
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

APLICACIÓN DE INTELIGENCIA AMBIENTAL PARA LA PLATAFORMA DE CÁMARAS INTELIGENTES INTEOX

AMBIENT INTELLIGENCE APPLICATION FOR INTEOX SMART CAMERA PLATFORM

TRABAJO DE FIN DE GRADO EN INGENIERÍA DE COMPUTADORES
DEPARTAMENTO DE COMPUTACIÓN Y AUTOMÁTICA

AUTORES

IKER GONZALEZ GONZALEZ

LUIS SÁNCHEZ CAMACHO

DIRECTORES

JUAN ANTONIO RECIO GARCÍA

CARLOS GARCÍA SÁNCHEZ

COLABORADOR

CARLOS GROSSOCORDON PEREZ

CONVOCATORIA: JUNIO 2022

GRADO EN INGENIERÍA DE COMPUTADORES
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

24 DE MAYO DE 2022

Resumen

El objeto de este trabajo es el estudio y evaluación de la plataforma Inteox 7000i de Bosch para su aplicación en inteligencia ambiental dedicada a la detección de los distintivos medioambientales de la DGT, que ayude a controlar la contaminación en calles de *Madrid Central*. Para ello se enseña cómo se configura este dispositivo IoT y el ecosistema de Azena, conocido también como *Security and Safety Things,* para el desarrollo aplicaciones móviles que ejecuta la cámara.

Durante el proceso de evaluación se explicará en qué consiste la *detección de objetos* y su aplicación en el ámbito de la inteligencia artificial, como se ha generado el dataset utilizado, las diferentes versiones de redes neuronales utilizadas y como han sido entrenadas. Finalizando con una comparativa del rendimiento de los modelos de deep learning utilizados en la cámara y diferentes propuestas para mejorar los resultados.

Palabras clave

Inteligencia artificial, Internet de las cosas, deep learning, detección de objetos, TensorFlow, YOLOV5, Bosch, Azena, Object detection Api, Python.

Abstract

The object of this work is the study and evaluation of the Bosch Inteox 7000i platform for its application in environmental intelligence dedicated to the detection of the environmental hallmarks of the DGT, which help control pollution in the streets of Madrid Central. To do this, it is taught how to configure this IoT device and the Azena ecosystem, also known as *Security and Safety Things*, for the development of mobile applications that run the camera.

During the evaluation process, it will be explained what object detection consists of and its application in the field of artificial intelligence, how the dataset used has been generated, the different versions of neural networks used and how they have been trained. Ending with a comparison of the performance of the deep learning models used in the camera and different proposals to improve the results.

Keywords

Artificial Intelligence, Internet of Things, Deep Learning, Object Detection, TensorFlow, YOLOV5, Bosch, Azena, Object Detection APIs, Python.

Índice de contenidos

Capítulo 1 - Introducción

1.1 Contexto

Con la premisa de tener un control más detallado de la generación de contaminación en las calles de Madrid, se propone la idea de desarrollar una aplicación que pueda detectar en tiempo real los distintivos medioambientales distribuidos por la dirección general de tráfico, DGT, o "pegatinas" comúnmente llamadas.

Según la DGT, "El distintivo ambiental es una manera de clasificar los vehículos en función de su eficiencia energética, teniendo en cuenta el impacto medioambiental de los mismos. La clasificación del parque tiene como objetivo discriminar positivamente a los vehículos más respetuosos con el medio ambiente y ser un instrumento eficaz al servicio de las políticas municipales, tanto restrictivas de tráfico en episodios de alta contaminación, como de promoción de nuevas tecnologías a través de beneficios fiscales o relativos a la movilidad y el medio ambiente." [1]

1.2 Objetivos

El objetivo de este trabajo de fin de grado es el de la creación de una aplicación capaz de detectar estos distintivos, mejorando el sistema actual el cual trata de identificar los vehículos mediante la matrícula, y a posteriori consultar en el sistema de la DGT el distintivo correspondiente, de esta manera se consigue agilizar el control de la contaminación con el fin de discretizar que calles son las que contienen el tráfico más contaminante, y aplicar las medidas correspondientes.

Junto con la creación de la aplicación y el modelo de IA, se hará una investigación con una cámara IoT cedida por Bosch, en colaboración a la Cátedra de inteligencia artificial aplicada al internet de las cosas de la Universidad Complutense de Madrid, con el fin de analizar la integración de la aplicación con el dispositivo.

Para ello se definen los siguientes objetivos del proyecto:

- Configuración de la cámara, AUTODOME Inteox 7000i, proporcionada por Bosch.
- Creación de un dataset de imágenes, para el entrenamiento del modelo.
- Selección de redes neuronales que mejor se adapten al funcionamiento de la cámara.
- Entrenamiento de un modelo de deep learning para detección de objetos.
- Integración del modelo en una aplicación de Android.
- Pruebas de rendimiento con diferentes modelos y resoluciones.

Capítulo 2 - Introduction

2.1 Context

The project starts from the idea of having a more detailed control of the pollution generation in the streets of Madrid, developing an application that can detect in real time the environmental badges distributed by the *dirección general de trfico*, DGT.

According to the DGT, "The environmental badge is a way of classifying vehicles according to their energy efficiency, considering their environmental impact. The classification of aims to positively discriminate against the most environmentally friendly vehicles and to be an effective instrument at the service of municipal policies, both restrictive of traffic in episodes of high pollution, and the promotion of new technologies through tax benefits or related to mobility and the environment." [1]

2.2 Objectives

The objective of this final degree project is the creation of an application capable of detecting these badges, improving the current system which tries to identify vehicles through the license plate, and a posteriori consult in the DGT system the corresponding badge. With this method is possible to streamline the control of pollution in order to discretize which streets are those that contain the most polluting traffic and implement appropriate measures.

Along with the creation of the application and the AI model, an investigation will be done with an IoT camera provided by Bosch, in collaboration with the Chair of artificial intelligence applied to the Internet of Things of the Complutense University of Madrid, to analyze the integration of the application with the device.

To this end, the following objectives of the project are defined:

- Camera configuration, AUTODOME Inteox 7000i, provided by Bosch.
- Creation of an image dataset, for model training.
- Selection of neural networks that best suit the operation of the camera.
- Training of a deep learning model for object detection.
- Integration of the model in an Android application.
- Performance testing with different models and resolutions.

Capítulo 3 - Marco teórico

La inteligencia artificial o IA, explicada de manera simple, es la simulación del razonamiento humano a la hora de realizar tareas y tomar decisiones, y a la vez aprender para mejorar esa dicha tarea.

Esta rama de la informática tuvo un gran desarrollo a partir de los años 50, donde personajes importantes como Alan Turing, quien se preguntaba ya en su artículo, *Maquinaria computacional e inteligencia*, por qué una máquina no iba a poder pensar y describe el problema como una especie de juego de imitación, o John McCarthy, quien acuñó el término de inteligencia artificial en 1956. [1]

En la actualidad, las aplicaciones de la inteligencia artificial están presentes en nuestro día a día, desde la optimización de búsquedas en internet o la detección de objetos en tiempo real, hasta la conducción autónoma de coches. Y esta materia se divide en diferentes subgrupos, y uno de los más comunes es el machine learning, en el cual vamos a profundizar.

El término machine learning fue acuñado en el año 1959, por Arthur Samuel, y se define como el proceso en el cual una máquina, sin necesariamente estar expresamente programada para ello y con la ayuda de modelos de datos y algoritmos, aprende a identificar patrones con el objetivo de generar predicciones y tomar decisiones en base a lo aprendido.

Hasta ahora los principales, y más comunes, tipos de modelos de machine learning eran cuatro: aprendizaje supervisado, aprendizaje no supervisado, aprendizaje por refuerzo y aprendizaje semi-supervisado. Pero recientemente se ha añadido un tipo nuevo denominado deep learning.

El deep learning, como hemos comentado anteriormente, se fundamenta en una red neuronal con tres o más capas. Si bien una sola capa es capaz de ofrecer predicciones, las demás capas ocultas ofrecen una optimización de estas, que deriva en una precisión mayor. Y es debido a su estructura que ha tomado relevancia últimamente, dado que necesitan grandes cantidades de datos sobre los que aprender, estos necesitan una potencia de cálculo significativa que no era accesible a un público más general, hasta recientemente, por su coste.

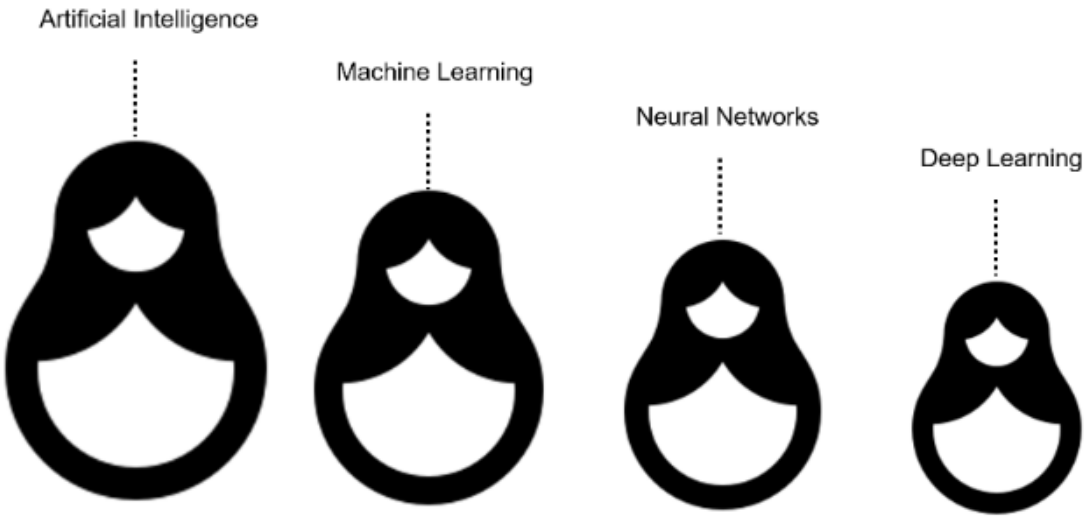


Imagen 1: Matrioskas represntado que contiene la inteligencia artificial

3.1 Redes Neuronales

Una red neuronal, en términos de la informática, es un modelo de computación simplificado que trata de emular el procesamiento de información que realiza el cerebro humano, tratando de computar un número elevado de unidades de procesamiento, de manera simultánea, interconectadas entre sí denominadas neuronas.

La neurona es el componente más básico del algoritmo, la cual se puede interpretar como un autómata o función matemática que trata de modelar un comportamiento en concreto y transmitir/propagar el resultado al resto de neuronas adyacentes. [2]

La red neuronal recibe como dato de entrada un vector donde cada uno de los elementos llegan a un nodo o neurona. Cada nodo realiza su función modificando los datos de entrada y emitiendo el resultado a la siguiente capa de neuronas, de manera que este proceso se repite hasta que llega a una capa final donde converge el resultado de todas ellas y se emite una predicción en relación con los cálculos realizados en las capas anteriores.

La razón por la que las redes neuronales son realmente útiles es debido a su capacidad para resolver situaciones complejas. Ya que, están diseñadas para aprender de las relaciones entre los datos de entrada y salida, encontrando en ellos relaciones que podrían pasar inadvertidas y patrones. Con esto se consigue generar predicciones precisas a posteriores datos de entrada. [3]

3.2 Clasificación de las redes neuronales

Según su topología las redes neuronales se clasifican principalmente en redes artificiales, convolucionales y recurrentes.

3.2.1 Redes neuronales artificiales (ANN)

Las redes neuronales artificiales, o también conocidas como Artificial Neural Networks (ANN) en inglés, fueron diseñadas para afrontar situaciones que hasta el momento solo las podía resolver un cerebro humano. Este tipo de redes se empezaron a desarrollar en 1943 por el fisiólogo, Warren McCulloch, y el matemático, Walter Pitts, quienes modelaron una red neuronal simple usando circuitos eléctricos para describir cómo trabajan las neuronas. [4]

Sin embargo, no es hasta que Rosenblatt, entre 1957 y 1958, desarrolla el primer tipo de modelo llamado Perceptron, o Perceptrón en castellano. Y no es hasta 1969 que no se crea el primer modelo del segundo tipo, el perceptrón multicapa o multilayer perceptron (MLP), por Marvin Minsky y Seymour Papert.

Estos dos tipos de redes eran unidireccionales hasta que, en 1986, Geoffery Hinton presenta un nuevo procedimiento de aprendizaje bajo el nombre de back-propagation, o retro propagación, que permitía el ajuste de los pesos de las conexiones en la red para minimizar la diferencia entre el vector actual de salida y el vector de salida deseado. [5]

Dentro de esta categoría se distinguen dos tipos, comentados anteriormente:

- *Perceptrón simple* (red neuronal monocapa), corresponde al modelo más básico compuesto por una capa de neuronas que propagan los datos de entrada a una capa de salida donde se obtiene la predicción final.

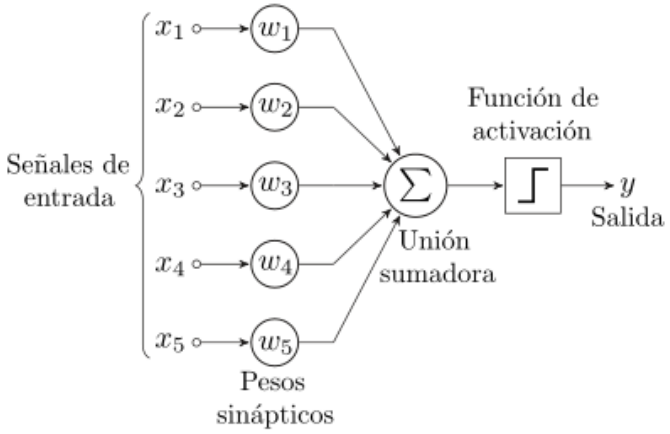


Imagen 2: Ejemplo de perceptrón simple

- *Perceptrón multicapa* (red neuronal multicapa), mantiene una topología similar a su versión monocapa, con la diferencia que entre la entrada y la salida dispone de una serie de capas de neuronas intermedias denominadas capas ocultas, donde se realizan una serie de modificaciones a los datos de entrada y estos son enviados a la capa de salida que obtiene la predicción final.

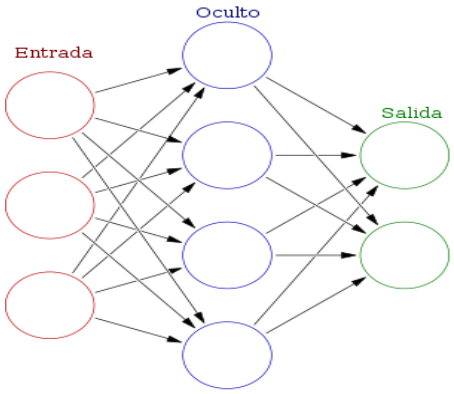


Imagen 3: Ejemplo de perceptrón multicapa

3.2.2 Redes recurrentes (RNN)

Este tipo de redes no tienen una estructura de capas definida, sino que puede tener una conexión arbitraria entre neuronas permitiendo realimentar la entrada de otras neuronas de la misma o diferente capa. Este tipo de tipología permite que la red tenga memoria, permitiendo crear ciclos o transmisión de estados. [6]

Gracias a esta arquitectura son principalmente utilizadas para el análisis de secuencias en textos, videos o sonidos.

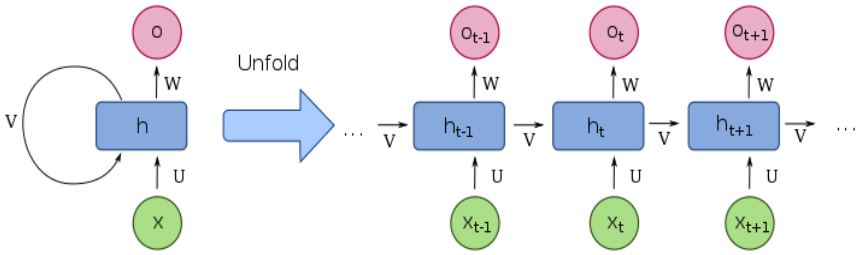


Imagen 4: Ejemplo de red recurrente (RNN)

3.2.3 Redes convolucionales

Este tipo de redes son una variación del perceptrón multicapa, ya que su aplicación se realiza en matrices multidimensionales (imágenes que tienen tres canales, RGB) cuya finalidad es tratar de emular la percepción visual del cerebro humano. El propósito de estas redes es tratar de extraer toda la información posible de una imagen y luego utilizar estas características para detectar patrones que permitan clasificar un objeto en la imagen. Su principal aplicación es la visión artificial para clasificación de imágenes y detección de objetos en tiempo real. [7]

La principal ventaja de este tipo de redes es que a cada parte o capa de la red se le asigna una tarea específica lo que reduce el número de capas ocultas y hace que su entrenamiento sea más rápido.

Su arquitectura se basa en combinar de forma alternada capas convolucionales y de reducción y finalmente, al igual que en el perceptrón multicapa, tiene una conexión a una capa de salida que hace la clasificación y proporciona la predicción final.

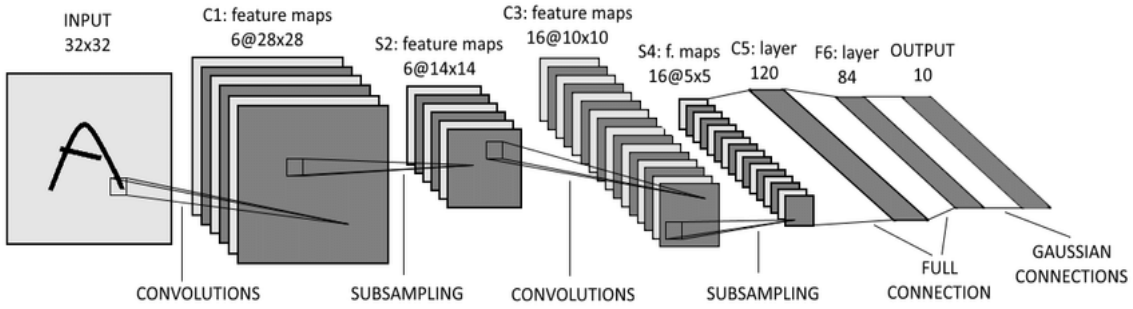


Imagen 5: Arquitectura de la red LeNet5

3.2.3.1 Convoluciones

En esta capa se extrae toda la información posible de la imagen de entrada. A esta información extraída se le aplica un filtro denominado *K* o núcleo de convolución, el cual consiste en una matriz bidimensional, y al resultado se le denomina mapa de características.

Para poder aplicar este filtro es necesario definir el número de mapas de características a utilizar ya que por cada capa de color se aplicará un filtrado, y las dimensiones del propio filtro serán inferiores a las del dato de entrada.

El proceso de filtrado se realiza colocando el filtro en la parte superior izquierda del primer canal de color de la imagen y se va desplazando hasta posiciones finales de la matriz de la imagen. En el desplazamiento se realiza el producto de cada valor del núcleo de convolución por su correspondiente valor en la imagen, lo que se resume en la multiplicación matricial entre el núcleo y una sección de la imagen. Esta acción se repite con el resto de los canales de entrada de la imagen.

El objetivo de aplicar estos filtros es intensificar las características presentes en la imagen, como pueden ser bordes horizontales o verticales, colores, formas, etc. que faciliten la detección de patrones del objeto a identificar.

3.2.3.2 Reducción

Estas capas denominadas también *pooling o subsampling* toman la información proporcionada por la convolución previa, y reducen la información que reciben. Para ello recorre la matriz del mapa de información final en pequeñas regiones o submatrices donde selecciona el máximo valor o la media de los píxeles de la región y genera un mapa más pequeño con esta información.

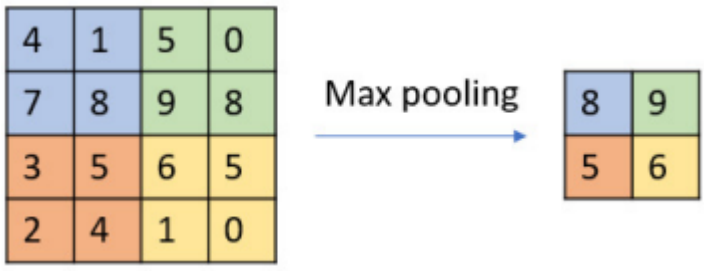


Imagen 6: Ejemplo de max pooling

3.2.3.3 Capas de salida

También conocidas como completamente conectadas (fully connected), enlazan todas las neuronas de entrada de la red con las de la salida. En las redes convolucionales se colocan al final para que una vez se hayan sustraído las características de la imagen, se puedan relacionar entre ellas y faciliten la predicción final.

3.2.4 Arquitectura SSD MobileNet

Para la evaluación del proyecto haremos uso de las redes neuronales SSD (single shot detection). Están formadas por una única red convolucional con el propósito de aprender a predecir las ubicaciones de los recuadros de la entidad y clasificar el resultado en un único paso. Su arquitectura está basada en la red MobileNet a la que le añade un par de capas convolucionales. [8]

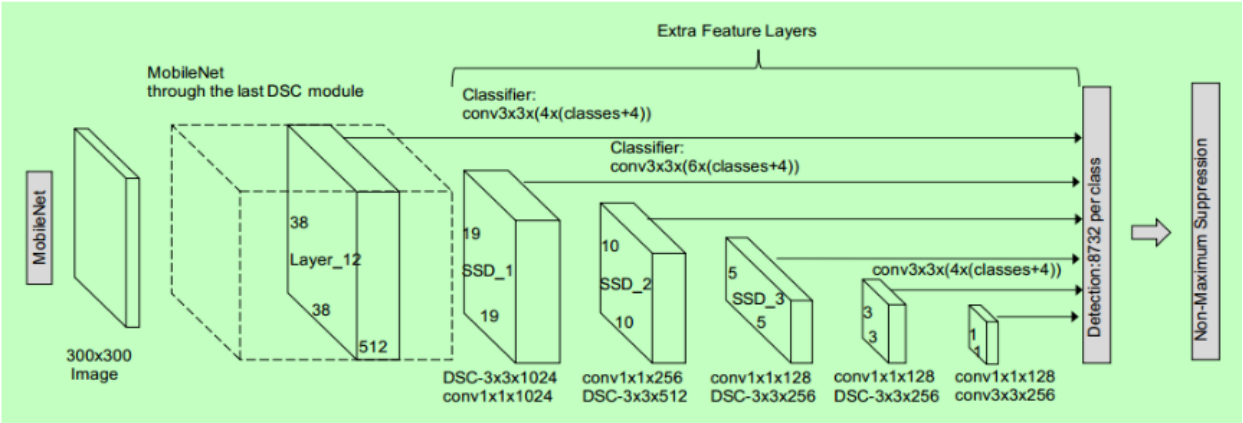


Imagen 7: ejemplo de red SSD modificada basada en MobileNet

La característica principal de este tipo de modelos es la capa final que incorporan, conocida como NMS, *non max suppression.* Con esta capa lo que se busca es que cuando el algoritmo detecta varias veces la misma entidad y su output sean múltiples recuadros sobre la misma región de la imagen, se eliminen las repeticiones y de una única respuesta. [12]

Imagen 8: Ejemplo de la característica Non-Max Suppression

El algoritmo funciona de la siguiente manera:

- Como **entrada** recibe una lista de recuadros propuestos con sus respectivas valoraciones de predicción y un umbral de pertenencia del contenido del recuadro al objeto detectado.
- Como **salida** devuelve una lista filtrando las proposiciones

Especificación:

1. Selecciona la proposición con la valoración más alta, la elimina de la lista de entrada y la incorpora a la lista de salida.
2. Comparar la proposición con el resto calculado el IOU, intersección sobre la unión. Si el valor de la intersección supera el umbral se elimina la propuesta de la lista de entrada.
3. Nuevamente se vuelve a elegir la propuesta de mayor valoración sobre las restantes en la lista de entrada, eliminando y agregando a la lista de salida para su posterior comparación.
4. Este proceso de selección y comparación se repite hasta que la lista de entradas se quede sin proposiciones.

Algorithm 1 Non-Max Suppression

```
procedure NMS(B,c)
    B_nms ← ∅                                   Initialize empty set
    for b_i ∈ B do                              Iterate over all the boxes
        discard ← False                         Take boolean variable and set it as false. This variable indicates whether b(i) should be kept or discarded
        for b_j ∈ B do                          Start another loop to compare with b(i)
            if same(b_i, b_j) > λ_nms then      If both boxes having same IOU
                if score(c, b_j) > score(c, b_i) then
                    discard ← True              Compare the scores. If score of b(i) is less than that of b(j), b(i) should be discarded, so set the flag to True.
        if not discard then                     Once b(i) is compared with all other boxes and still the discarded flag is False, then b(i) should be considered. So add it to the final list.
            B_nms ← B_nms ∪ b_i
    return B_nms                                Do the same procedure for remaining boxes and return the final list
```

Imagen 9: Algoritmo de la característica Non-Max Suppression

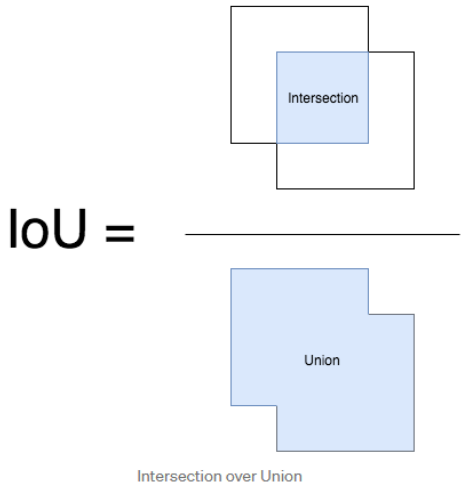


Imagen 10: Intersección sobre unión

Capítulo 4 - Desarrollo de modelos Deep Learning para la detección de objetos

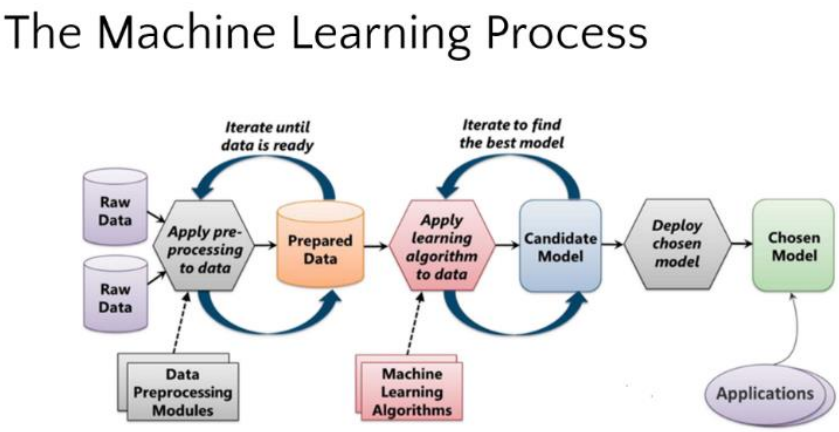


Imagen 11: Ciclo de vida de un proyecto de Machine Learning

Si bien ya hemos explicado la estructura del deep learning, ahora necesitamos poner a trabajar dicha estructura. Para ello hay que seguir una serie de pasos:

- *Preparación de los datos*: esta parte consiste en la adquisición de todos los datos necesarios para el entrenamiento. Por ejemplo, para este proyecto se han utilizado fotografías de vehículos tomadas desde el frente donde el distintivo ambiental era reconocible. En esta etapa también pueden utilizarse métodos para aumentar el conjunto de datos recopilados, como por ejemplo en el caso de las fotografías, se pueden crear nuevas imágenes a partir de las originales modificando su ángulo o cambiando su color.

- *Entrenamiento del modelo*: en este punto se va a definir cuál es la predicción que pretendes conseguir con tu modelo. Debes seleccionar una gran parte de los datos preparados previamente y utilizarlos para hacer aprender a tu algoritmo los diferentes patrones existentes, por eso la preparación de los datos es un punto muy importante, ya que, de esta depende el conseguir generar predicciones acertadas. Aunque no todo son los datos, también se pueden crear modelos más o menos parecidos dependiendo del framework utilizado y su configuración.

- *Validación del modelo*: esta es la prueba que sirve para confirmar si el modelo entrenado es aceptable o no para el usuario. En esta parte se utilizan datos de los ya preparados previamente, donde ya conoces cuál debería ser el resultado de la predicción, y comprueba cuán preciso es tu modelo

- *Despliegue del modelo*: por último, una vez obtenido el modelo deseado ya está preparado para su implementación. En el caso de este proyecto el despliegue se realiza en forma de una aplicación de Android instalada en la cámara de Bosch.

A continuación, se explicará más a fondo el procedimiento que realizan las CNN, en el filtrado de imágenes para la detección de entidades, en relación con la tarea que concierne a este proyecto, detectar en tiempo real los distintivos medioambientales de la DGT.

4.1 Introducción

La detección de objetos es una técnica muy utilizada en el ámbito de la visión artificial, especialmente en el campo del coche autónomo donde una gran parte de los vehículos de la actualidad incorporan un sistema capaz de detectar elementos presentes en la carretera como pueden ser señales de velocidad, peatones, etc.

Esta tecnología consiste en agrupar elementos comunes y no comunes bajo una serie de entidades o etiquetas. Es decir, en el ejemplo que corresponde al proyecto todas las pegatinas medioambientales, se agrupan en 4 categorías: B, C, ECO y OE.

Imagen 12: Distintivos ambientales de la DGT

El objetivo es que la red convolucional sea capaz de clasificar en tiempo real la imagen que recibe en una de esas 4 etiquetas y localizar la posición del objeto dentro de la imagen.

Imagen 13: Ejemplo de una imagen etiquetada por el modelo desarrollado

4.2 Redes pre entrenadas

En la fase de entrenamiento se buscan los mejores valores de los pesos para las conexiones internas de las neuronas, de cara a la clasificación en la salida. Tras entrenar la red es posible guardar el estado del modelo para poder utilizarlo en un futuro sin la necesidad de volver a entrenarlo.

Debido a que la primera fase de entrenamiento de una red neuronal es larga y tediosa, ya que los pesos de las neuronas están sin ajustar y provoca un coste algorítmico elevado, es muy común utilizar redes neuronales ya entrenadas con colecciones de datos lo suficientemente amplios que hagan que realizar un nuevo ajuste de los pesos sea más liviano. Este tipo de modelos son entrenados por grandes empresas tecnológicas como Google, Nvidia, Microsoft, etc. que tienen a su disposición una infraestructura de hardware del que un usuario medio no puede disponer.

Sobre estas redes entrenadas para casos de usos genéricos, es posible realizar un ajuste sobre los pesos para ajustar la clasificación al problema que nos concierne. Este proceso se conoce como *fine tuning*, consiste en recuperar el estado de una red entrenada con otros datos y ajustar los parámetros necesarios para adaptarlo a nuestro caso de uso, y volver a entrenar la misma red neuronal con los nuevos datos de nuestro problema.

Este proceso de entrenamiento es menos costoso ya que al usar un modelo pre entrenado del mismo dominio de nuestro problema, visión artificial, muchos de estos pesos serán similares a los que necesite la nueva versión de la red neuronal, y hará que el ajuste se realice con mayor rapidez.

4.3 Metodología

Para efectuar esta tarea el modelo o red neuronal trata de aplicar una serie de filtros a la imagen de entrada, como se ha explicado anteriormente en la convolución, con el objetivo de facilitar la detección de características.

Un preprocesado de imagen muy comúnmente utilizado es la transformación de la imagen a una escala de grises, que facilita ver los contornos de los elementos y reduce el coste de cómputo al eliminar los tres canales de color. [8]

Imagen 14: Ejemplo de preprocesado de datos con señales[1]

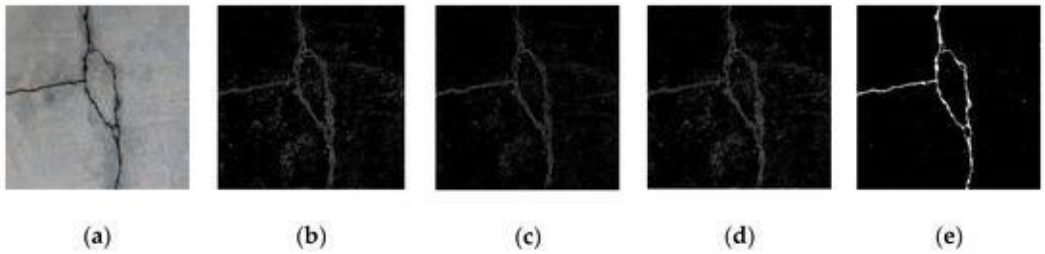


Imagen 15: Ejemplo de preprocesado de datos con una grieta

Como se puede observar en las imágenes anteriores, tanto el número como el contorno de la señal se aprecia con mayor facilidad y lo mismo ocurre con la imagen de la grieta.

La finalidad está en que en el resto de las capas convolucionales se apliquen una serie de filtros y transformaciones que permitan extraer toda la información posible. Estas características son propagadas a las neuronas de las capas de salida, donde habrá una neurona por cada clase que se desea identificar, en la que la neurona indicada será activada y dará la predicción correspondiente.

[1] *https://www.mdpi.com/1424-8220/20/7/2021*

4.3.1 Funciones de activación

Este tipo de funciones se encarga de devolver una salida en un rango de valores entre (0, 1) o (-1, 1), este valor se conoce como *score* o *confianza* que hace referencia a la probabilidad de que los datos de entrada pertenezcan a un conjunto o clase.

Su funcionalidad en la capa final de predicciones consiste en que cada neurona de la salida ejecuta su propia función de activación con unos determinados parámetros y en relación a esto devolverá un resultado. Tras obtener las *predicciones* de todas las neuronas la capa final elige la que tiene mayor peso y devuelve la clase asociada a ella.

Existen varios tipos de funciones:

- *Tangente hiperbólica*, transforma los valores introducidos a una escala (-1, 1), donde los valores altos se aproximan a 1 y los valores bajos a -1.
- *ReLu (Rectified Lineal Unit),* anula los valores negativos persistiendo solo los positivos.
- *Leaky ReLu,* corrige los valores negativos en positivos, multiplicándolos por un coeficiente rectificativo.
- *Softmax*, transforma las salidas a una lista de probabilidades de forma que todas ellas suman 1.

4.4 Frameworks de Deep Learning

4.4.1 TensorFlow

TensoFflow [2]es una plataforma de código abierto orientada al desarrollo de aplicaciones de aprendizaje automático. Desarrollada por Google con el objetivo de crear y entrenar redes neuronales para detectar correlaciones y patrones, análogos al aprendizaje y razonamiento del ser humano.

Puede ejecutarse en múltiples dispositivos haciendo uso de la CPU y en GPUs compatibles con CUDA[3], librería propia de aceleradores o GPUs de NVIDIA para operaciones vectoriales como las que realizan internamente las capas de una red neuronal. Está disponible principalmente para sistemas basados en Unix de 64 bits y dispositivos Android y iOS. Existen versiones

[2] https://www.tensorflow.org/?hl=es-419

[3] https://developer.nvidia.com/cuda-zone

adaptadas para Windows, haciendo uso de directx12 y otro tipo de hardware como el de Intel mediante la librería OpenVino[4].

4.4.2 Tensorflow lite

Uno de los paradigmas más relevantes de la IA en relación con la computación, es la posibilidad de ejecutar estos modelos o redes neuronales sobre dispositivos que no tengan una capacidad de cómputo equiparable a la que puede tener un ordenador convencional con una GPU.

TensorFlow lite[5], es el framework de TensorFlow que permite adaptar modelos de deep learning a un formato más liviano para dispositivos de bajo consumo denominados *edge devices*, como pueden ser las Raspberry pi, Arduino, dispositivos móviles u otro tipo de sistemas empotrados. Existen otros frameworks para este propósito como ONNX [6]o el citado anteriormente, OpenVino.

En *TensorFlow lite*, a este proceso de optimización se le denomina *cuantización*[7]. Se realiza tras finalizar el entrenamiento, cuya finalidad es transformar el tipo de datos que usa internamente la red neuronal para reducir el peso del modelo y agilizar la inferencia, a costa de una pequeña degradación en la precisión.

Existen varias técnicas de optimización:

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU

Imagen 16: Alguna de las tecticas de optimización de TensoFflow Lite

[4] https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html

[5] https://www.tensorflow.org/lite?hl=es-419

[6] https://onnx.ai/

[7] https://www.tensorflow.org/lite/performance/post_training_quantization

4.4.2.1 Cuantización de rango dinámico

Es la forma más simple de optimización, consiste en transformar los pesos internos de valores de punto flotante (float 32) a enteros de 8-bits. En la inferencia estos valores son convertidos dinámicamente en floats para hacer las operaciones y se almacena de forma temporal la conversión para reducir la latencia.

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]

tflite_quant_model = converter.convert()
```

4.4.2.2 Cuantización completa de enteros

En este tipo de cuantización se trata de reducir al máximo el coste computacional y de memoria para mejorar la compatibilidad con el hardware. Para ello es necesario transformar todos los tipos de datos, tanto entrada, salida, como los internos del propio modelo a enteros de 8-bits. Esto conlleva que los datos que se van a utilizar para la inferencia también tienen que ser preprocesados y convertidos al mismo formato, es por ello que para poder cambiar calibrar las capas internas a enteros de 8-bits es necesario indicar, a modo de ejemplo, una representación de los datos de entrada que va a recibir a enteros de 8-bits.

Utilizando entradas de nuestro dataset:

```
def representative_dataset():
  for data in f.data.Dataset.from_tensor_slices((images)).batch(1).take(100):

    yield [tf.dtypes.cast(data, tf.float32)]
```

Con fines de prueba:

```
def representative_dataset():
    for _ in range(100):
      data = np.random.rand(1, 244, 244, 3)

      yield [data.astype(np.float32)]

import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8  # or tf.uint8
converter.inference_output_type = tf.int8  # or tf.uint8

tflite_quant_model = converter.convert()
```

4.4.2.3 Cuantización float16

Esta conversión funciona igual que la de rango dinámico, con la diferencia de que los pesos se mantienen en valores de punto flotante de longitud 16. Esto hace que la pérdida de precisión sea menor y se reduce el tamaño del modelo a la mitad.

Es de las más utilizadas ya que existe una gran cantidad de GPUs con la capacidad de operar de forma nativa con este tipo de datos sin tener que hacer la conversión desde float32.

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]
tflite_quant_model = converter.convert()
```

4.4.3 Object detection api

Para el entrenamiento de las redes neuronales haremos uso de una Api denominada Object Detection[8] de Tensorflow, que nos ofrece una serie de scripts y funcionalidades en Python que facilitan la labor de entrenamiento. Es un framework de código abierto desarrollado por la comunidad y colaboración de Google, que facilita el entrenamiento de redes neuronales orientadas a la visión artificial.

Desarrollado en Python 3.6, presenta dos versiones basadas en TensorFlow:

- TensorFlow 1.15[9], para modelos de TF1
- TensorFlow 2.2[10], para modelos TF2

Para su instalación y uso es necesario que nuestro entorno de desarrollo tenga una GPU compatible con CUDA, como ocurre con la mayoría de frameworks de inteligencia artificial orientados a las redes neuronales.

Clonamos el proyecto de GitHub:

```
git clone https://github.com/tensorflow/models.git
```

---

[8] *https://github.com/tensorflow/models/tree/master/research/object_detection*

[9] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1.md

[10] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2.md

Instalamos el paquete de Python y las dependencias correspondientes:

```
python object_detection/builders/model_builder_tf1_test.py

python object_detection/builders/model_builder_tf2_test.py
```

Es recomendable separar las dos versiones de TensorFlow en proyectos y entornos de Python diferentes para evitar conflictos de librerías.

Para el uso de esta API, tenemos a disposición un repositorio de modelos pre entrenados con el dataset mscoco[11]. Este consiste en una colección de imágenes que recogen entidades cotidianas como sillas, bicicletas, animales, algunos vehículos, etc.

[11] https://cocodataset.org/#home

Capítulo 5 - Arquitectura y Tecnologías

En este capítulo se presenta una introducción a la cámara y una guía más detallada de como configurarla por primera vez. Para esta tarea se utilizará la aplicación Configuration Manager[12] de Bosch y la interfaz web que ofrece la cámara.

Asimismo, se explicará cómo poner a punto el entorno de desarrollo de las aplicaciones, con las herramientas que nos proporciona Azena.

5.1 Cámara Inteox 7000i

La cámara sobre la cual se va a realizar la instalación de las aplicaciones ha sido proporcionada por Bosch, y corresponde al modelo AUTODOME Inteox 7000i. Este tipo de dispositivos se alimentan usando el puerto ethernet mediante un adaptador PoE (Power over Ethernet) que transmite corriente y a la vez que transfiere datos.

Citando expresamente la información recogida del datasheet, "La cámara AUTODOME inteox 7000i es una cámara de vigilancia PTZ avanzada 30x basada en un sistema operativo abierto compatible con OSSA de Security and Safety Things. Gracias a la tecnología de captura de imagen starlight, que ofrece una excelente sensibilidad con poca luz, la versión de Video Analytics más completa del mercado y flujos de vídeo, la cámara proporciona una calidad de imagen inigualable. Incluso en las peores condiciones de iluminación, la cámara proporciona vídeo de alta definición (HD) de 1080p." [10]

Antes de comenzar con la configuración es necesario tener conectada la cámara a un router, mediante el PoE, para poder conseguir una conexión a internet. Este paso es imprescindible debido a la necesidad de conectar la cámara a diferentes servicios que haremos uso posteriormente.

La configuración se realiza en Windows mediante la aplicación de Bosch, Configuration Manager, donde en la pestaña device allocator se mostrarán todas las cámaras que son accesibles a través de la red. Para poder acceder a la cámara, primero deberemos autenticarnos, haciendo clic en el botón derecho del ratón, sobre el dispositivo donde deseamos realizar la autenticación.

[12] https://downloadstore.boschsecurity.com/index.php

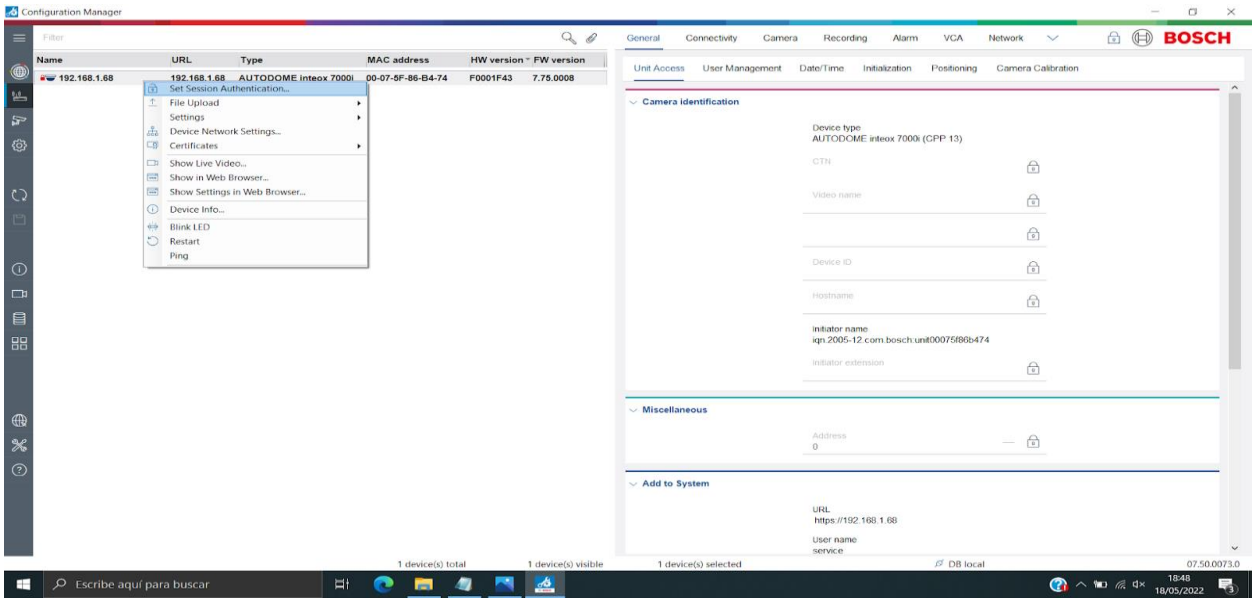


Imagen 17: Creación de una clave de autenticación en Configuration Manager

Ahora podremos acceder a los diferentes paneles que hay en la aplicación para configurar el dispositivo en diferentes ámbitos, como, por ejemplo, la dirección de la lente o la dirección IP, entre otras muchas cosas. También podremos acceder a la interfaz web de la cámara a través de un navegador con la IP asignada a la cámara.

Para poder instalar aplicaciones en la cámara debemos registrar la misma en el portal de Azena, que es donde podremos encontrar diferentes aplicaciones, y en Bosch Remote Portal. Para ello accedemos en el apartado Connectivity desde la aplicación, Configurator Manager.

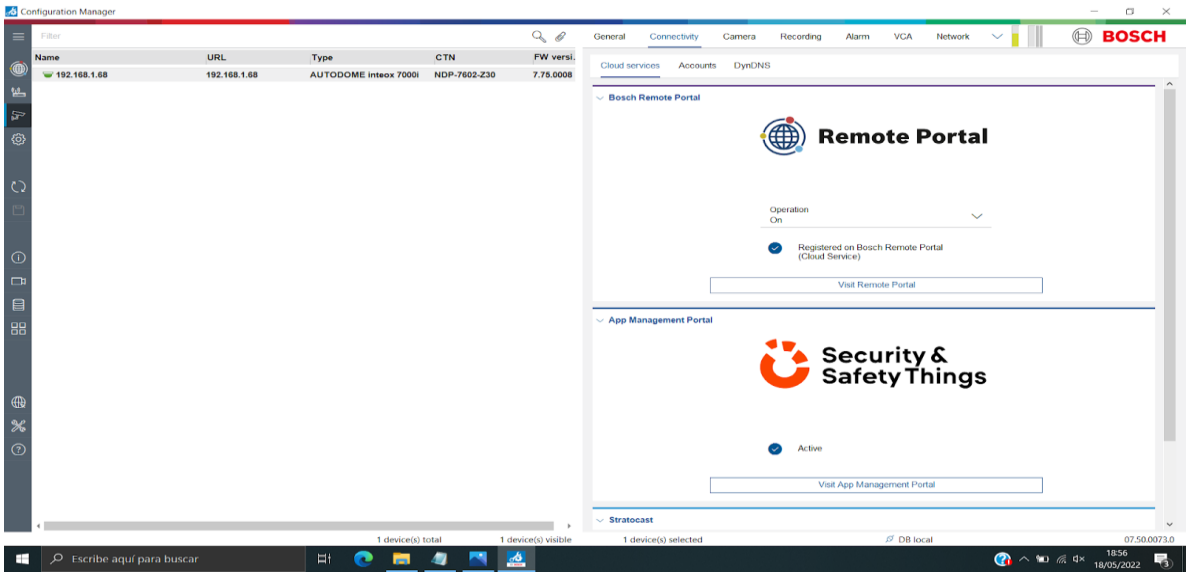


Imagen 18: Visualización del apartado conectivity de Configuration Manager

Sin embargo, si se desea crear aplicaciones para la cámara, y por consiguiente probarlas sobre la misma, deberemos activar el modo desarrollador. Este modo nos permite realizar instalaciones sobre la cámara, bajo nuestra responsabilidad, que no han sido verificadas por Bosch ni Azena.

Para poder activar este modo primero tendremos que contactar con el soporte técnico de Bosch, el cual nos enviará un archivo que deberemos reenviar relleno con los datos que nos piden. Una vez validen los datos nos enviarán un código que deberemos introducir en la interfaz web de la cámara, en la pestaña de Configuration, apartado Service y seleccionar Licenses e introducir el código recibido.A partir de ahora podremos instalar cualquier aplicación, siguiendo los pasos como se muestra en el apartado de compilación e instalación de la aplicación.

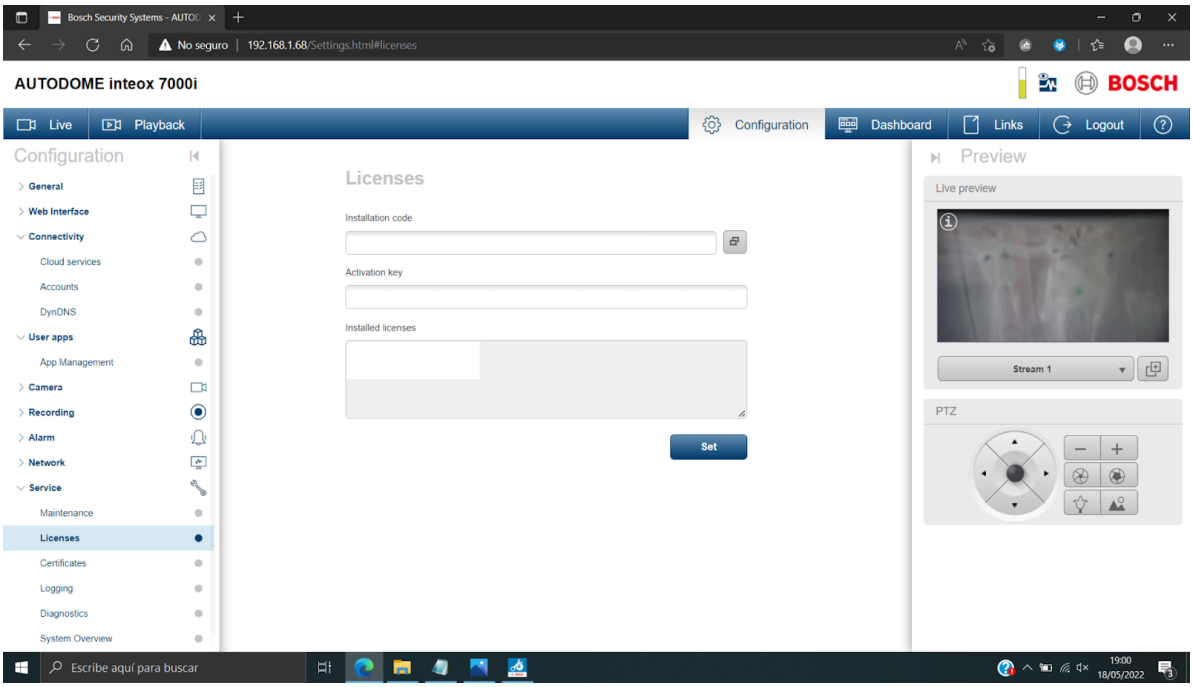


Imagen 19: Visión del apartado licenses de la interfaz web de la cámara AUTODOME Inteox 7000i

Una vez realizado activado el modo desarrollador deberemos acceder a la pestaña de User Apps y seleccionar App Management. Ahora tendremos que pinchar en Switch to local App Management Console y nos direccionará a la página donde podremos gestionar las diferentes aplicaciones instaladas en la cámara.

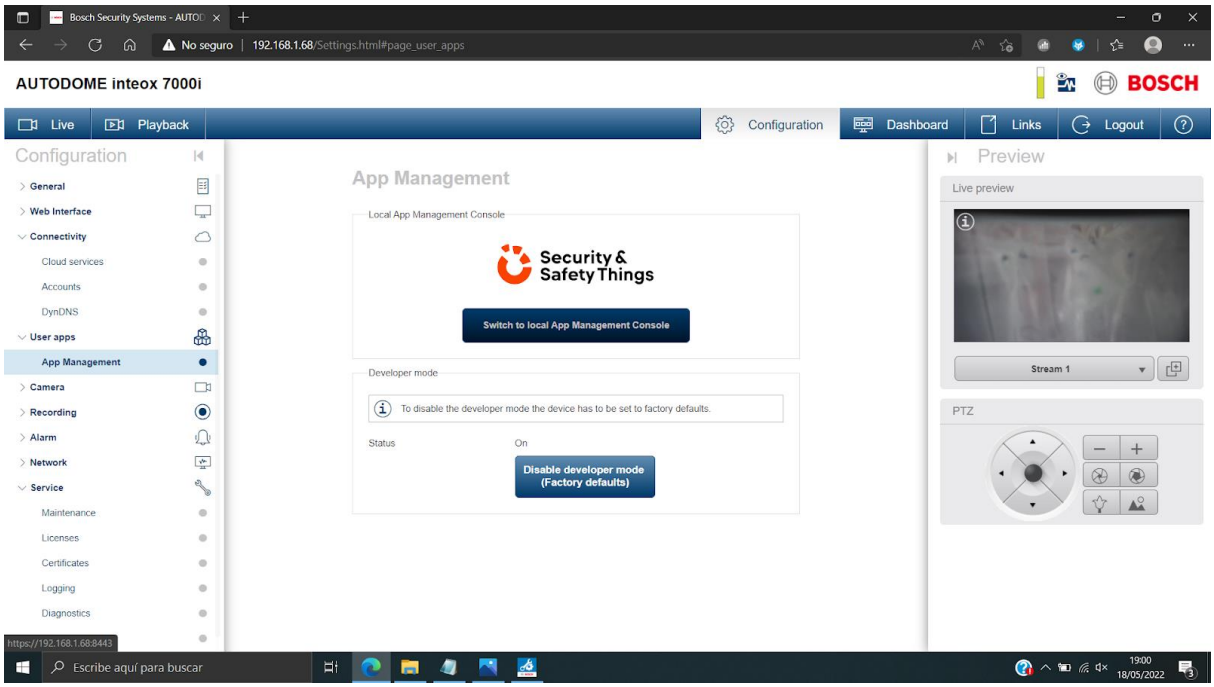


Imagen 20: Visión del apartado App Management de la interfaz web del dispositivo

Una vez en la página aparecerán todas las aplicaciones que tenemos instaladas y datos relativos a las mismas, como la versión, su estado o la cantidad de recursos de la cámara que están utilizando. También, si pinchamos sobre los 3 puntos verticales que aparecen a la derecha de cada aplicación se desplegará un menú que nos permitirá comenzar o parar la ejecución de dicha aplicación.

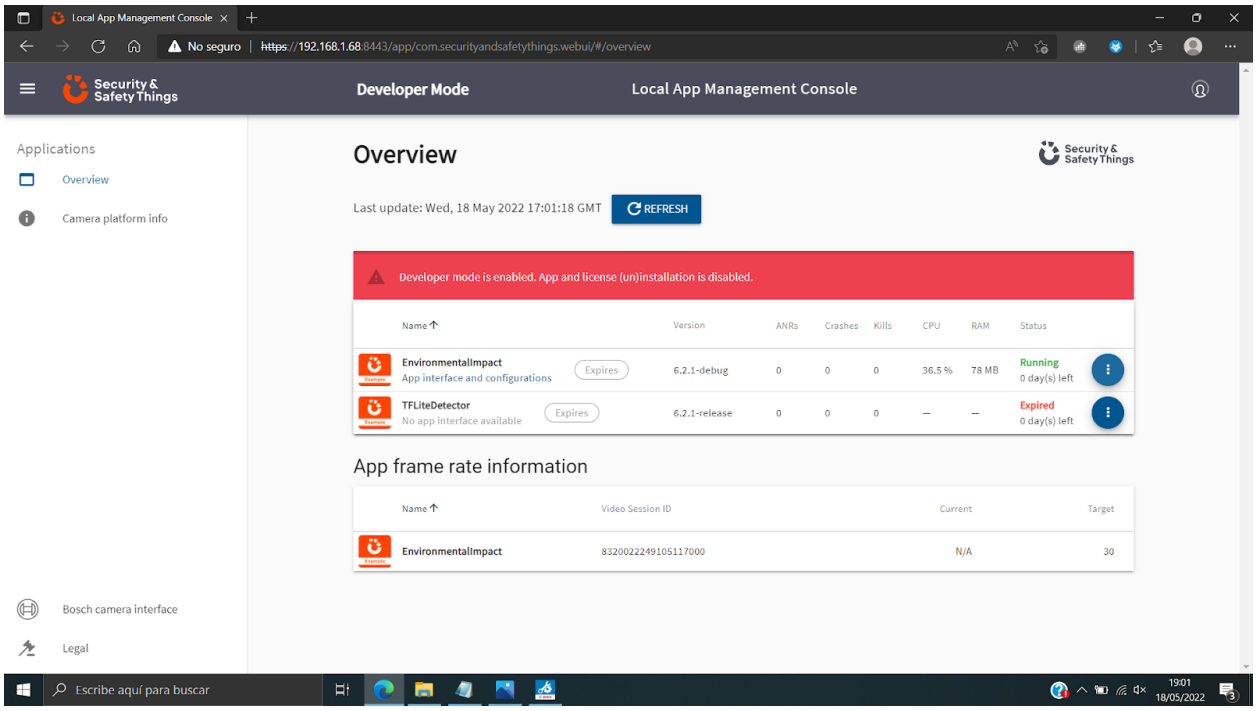


Imagen 21: Visión de la interfaz Local App Management Console del dispositivo

Por último, podremos seleccionar App interface and configuration, que aparece debajo de la aplicación, para abrir otra interfaz que nos muestra tanto una imagen en tiempo real de la cámara como datos sobre la aplicación, estos datos son el tipo de aceleración utilizado, los fotogramas por segundo y el tiempo de inferencia.

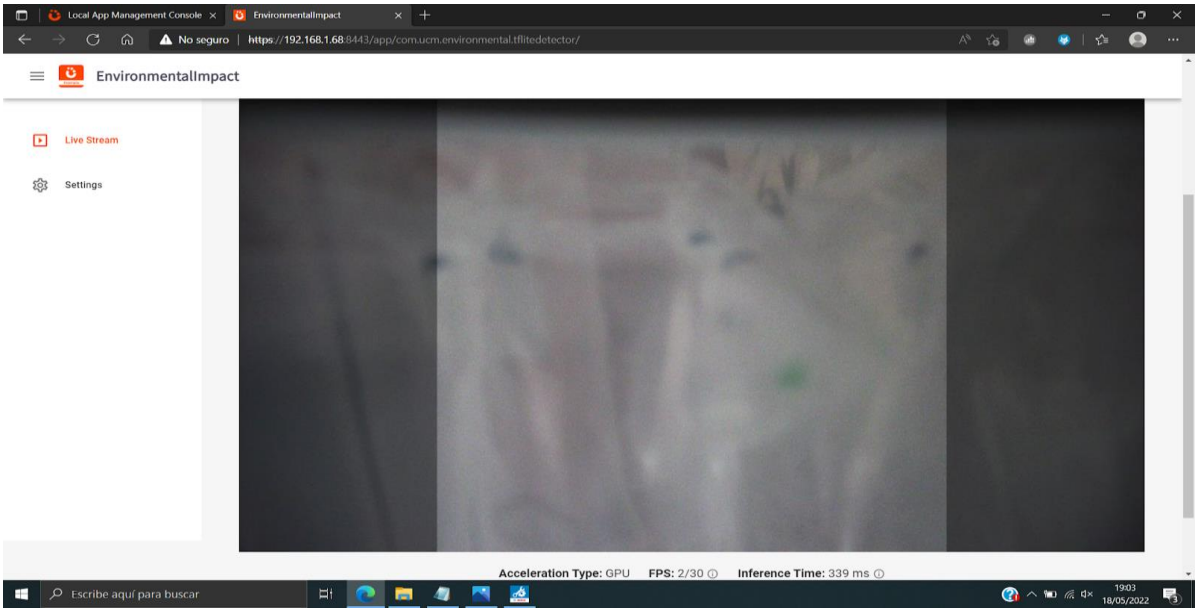


Imagen 22: Visión de la configuración de una aplicación instalada en el dispositivo

En el apartado de Settings, podremos modificar ciertos aspectos de la aplicación como el threshold y el tipo de aceleración que se quiere aplicar, si es que se desea aplicar.

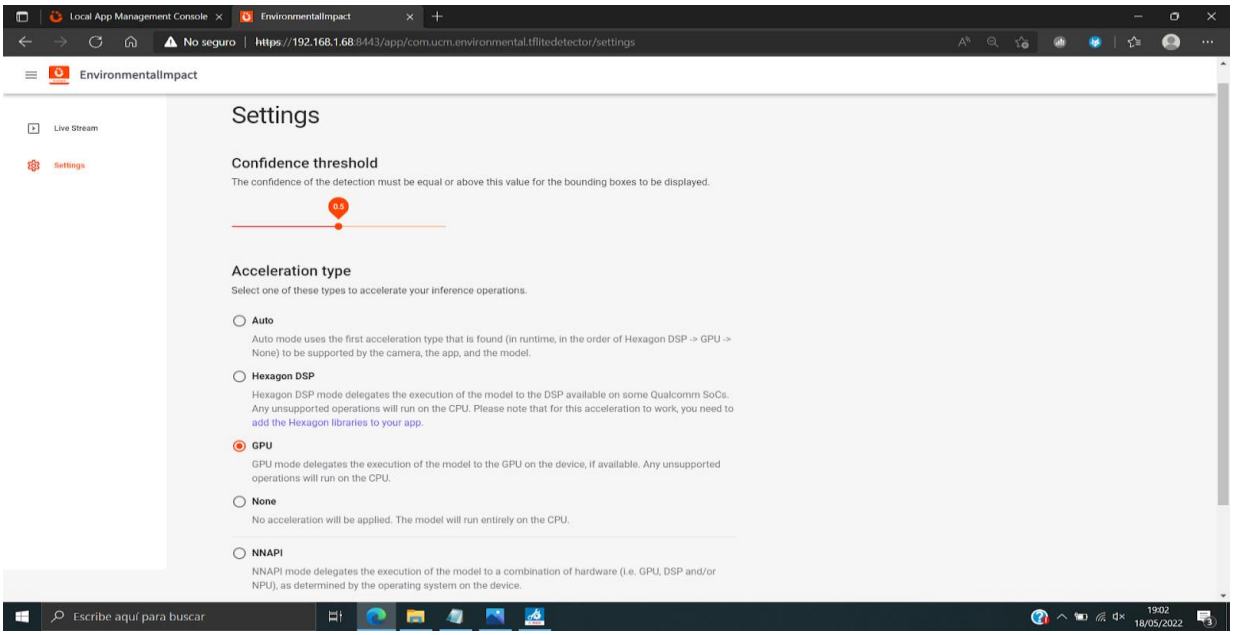


Imagen 23: Apartado Settings de una aplicación instalada en el dispositivo

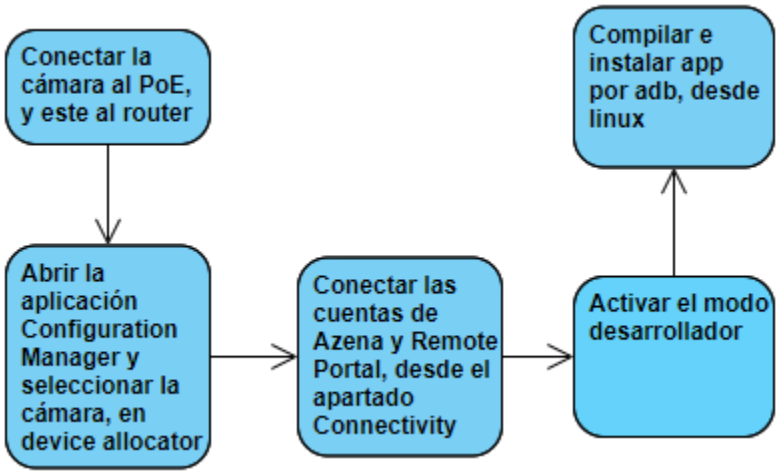


Imagen 24: Flujo de la configuración de la cámara

5.2 Ecosistema de desarrollo

Para el desarrollo de la aplicación hemos hecho uso de las herramientas que nos proporciona Azena. Ya que las cámaras de Bosch implementan una versión modificada de Android con el paquete de *security and safety things*, propiedad de *Azena*, que contiene las librerías necesarias para desarrollar aplicaciones de deep learning. Este software está disponible en los dispositivos IoT orientados a la visión artificial de los siguientes fabricantes:

Imagen 25: Fabricantes que desarrollan software para los dispositivos IoT

Para la creación de estas aplicaciones, Azena ofrece un ecosistema basado en:

- **Sistema operativo**, estandarizado para los dispositivos compatibles IoT *Security and Safety Things*, el cual ha sido modificado para su uso en dispositivos empotrados.
- **Consola de desarrollo y tienda de aplicaciones**, permite a los desarrolladores publicar sus aplicaciones y ofrecerlas al resto de desarrolladores y usuarios para poder integrarlas en sus dispositivos.
- **Device management portal**, es la herramienta que permite controlar los dispositivos. En el caso de Bosch existe la alternativa del *Configuration Manager*, que integra la herramienta de Azena, además de otra serie de opciones propias de la cámara.

5.2.1 Azena Toolchain.

Para la creación de aplicaciones disponemos de *Azena Toolchain*, consta de una serie de librerías y herramientas para el desarrollo de aplicaciones Android en Java/Kotlin y C/C++.

Para ello es necesario disponer de un sistema basado en *Linux/Unix*, debido a que hace uso de OpenJDK 8[13], la versión open source del paquete de desarrollo de Java que no está disponible en Windows hasta la versión 9.

[13] https://openjdk.java.net/install/

Continuando con la configuración, necesitamos descargar y extraer dos paquetes en la misma carpeta, /azena/sdk/:

- SDK, para el desarrollo en Java/Kotlin.
- NDK, para aplicaciones nativas en C/C++.

De forma opcional podemos agregar al PATH del sistema dos herramientas principales:

- ADB, para interactuar con nuestro dispositivo android, azena/sdk/*platform-tools/adb*
- SDK Manager, para actualizar librerías e instalar los componentes necesarios para el desarrollo. azena/sdk/*tools/bin/sdkmanager*

Tras instalar y configurar las herramientas del *Azena toolchain*[14], tenemos que adquirir una serie de componentes necesarios para el desarrollo y prueba de la aplicación, haciendo uso de *sdkmanager*.

Listamos todos los componentes disponibles:

```
sdkmanager --list
```

En la lista que nos aparece por consola instalamos los componentes deseados, en nuestro caso elegimos la versión v5 o v6 que son las compatibles con el firmware de la cámara y la aplicación:

```
sdkmanager --install "system-images;android-29;securityandsafetythings_os_v5;x86_64"
```

Una vez realizados los pasos anteriores, estamos en disposición de desarrollar y configurar nuestra aplicación. Disponemos de tres entornos de programación para esta finalidad, IntelliJ IDEA, Android Studio y Visual Studio Code. Este último es el más recomendable al tener una configuración más sencilla, donde solo necesitamos instalar una extensión e indicar la ruta donde fueron extraídos los paquetes *SDK* y *NDK*.

```
code --install-extension securityandsafetythings.vsix
```

[14] https://docs.azena.com/developer/introduction/downloads#developer-toolchain

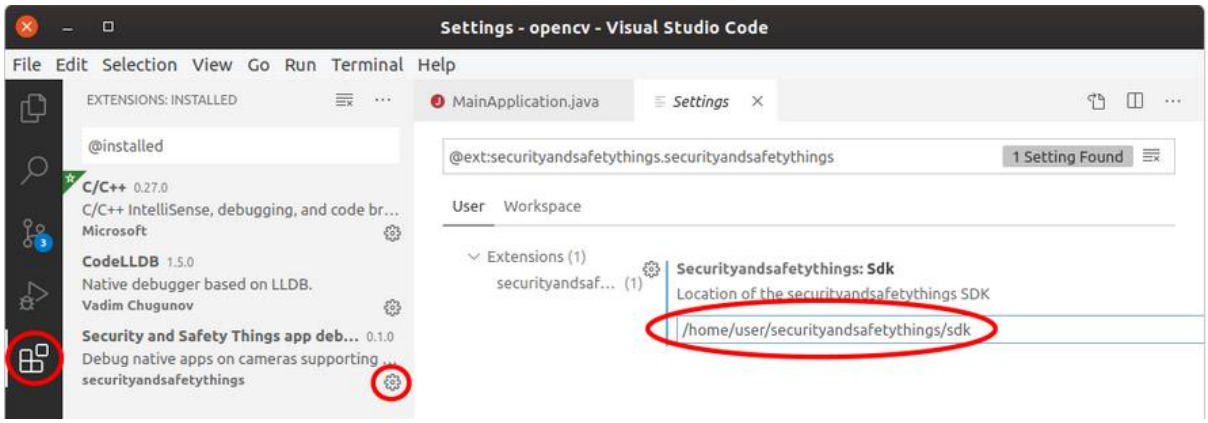


Imagen 26: Ejemplo de donde indicar la ruta de los paquetes SDK y NDK

De manera común al resto de entornos hay que realizar unos pasos extras para poder compilar las aplicaciones. Indicar en las variables del sistema el path de las librerías:

```
export ANDROID_SDK_ROOT=~/azena/sdk

export ANDROID_NDK_HOME=~/azena/sdk/ndk-bundle
```

Configurar las credenciales de Azena para descargar las dependencias necesarias de los repositorios:

- Creamos un archivo *gradle.properties* dentro de la carpeta *.gradle* en el directorio raíz de nuestro usuario. En el agregamos nuestras credenciales:

```
sstUsername=John.Doe@example.com

sstPassword=MySecurePassword
```

- Continuamos creando en el directorio raíz el fichero *.npmrc*:

```
# npm registry for Azena
@azena:registry=https://artifacts.azena.com/repository/npm/
artifacts.azena.com/repository/npm/:username=John.Doe@example.com
artifacts.azena.com/repository/npm/:email=John.Doe@example.com
# Password "MySecurePassword" must be base64-encoded before adding to
this file.
artifacts.azena.com/repository/npm/:_password=TXlTZWN1cmVQYXNzd29yZA==
```

En este caso es necesario que la contraseña esté codificada en *base64*. Para ello necesitamos hacer uso del siguiente comando de consola y copiar el resultado en *_password*, en el fichero citado anteriormente.

```
echo -n "MySecurePassword" | openssl base64
```

Tras realizar estos pasos tendremos configurado el entorno necesario para la creación de aplicaciones.

Capítulo 6 - Metodología del proyecto

El objeto de investigación del proyecto consiste en evaluar el rendimiento de la cámara proporcionada por BOSCH, y seleccionar la red neuronal que mejor se adapte para su utilización en la detección de distintivos medioambientales.

En esta sección se explicará de manera práctica cómo se genera un dataset para entrenamiento y validación, descarga de un modelo pre entrenado para su posterior *fine tuning*, y su conversión a tflite para su integración *edge devices*. Para realizar esta tarea, se ha seguido el siguiente flujo.

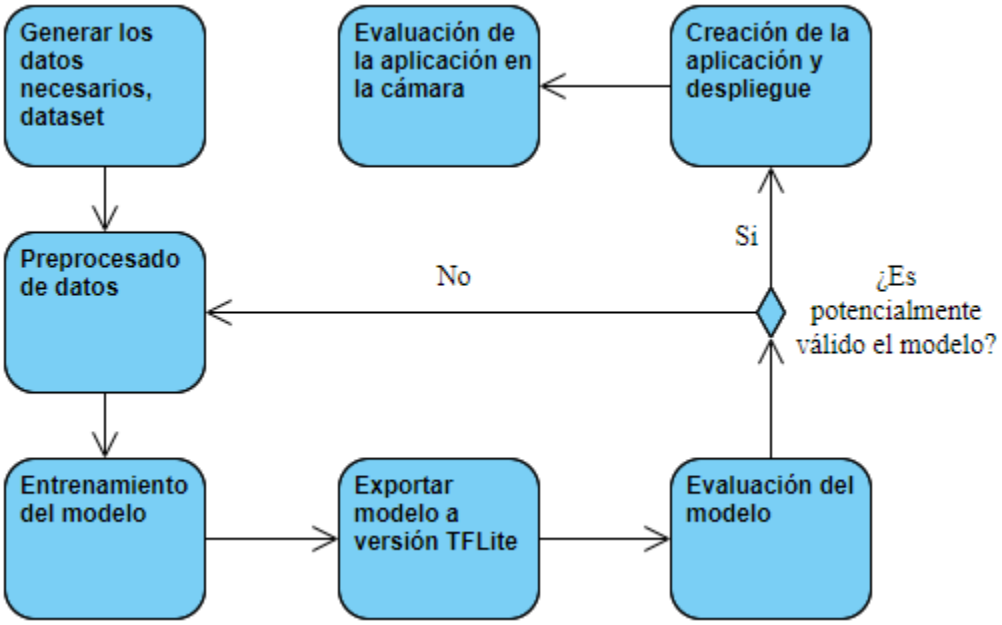


Imagen 27: Flujo de investigación del proyecto

6.1 Dataset

Como el objetivo de la red neuronal es detectar los distintivos medioambientales, nuestros datos de entrenamiento serán imágenes de vehículos desde una perspectiva frontal.

Imagen 28: Imágenes de ejemplo del dataset creado

Para la creación del dataset es necesario que los datos de entrenamiento se ajusten al caso de uso real, es decir, que a la hora de utilizar el modelo para hacer predicciones la perspectiva de la imagen sea la misma. En nuestro caso de uso implica que la cámara va a recibir imágenes de vehículos que vienen de frente, por lo que las imágenes de entrenamiento tienen que ser iguales.

Las imágenes son realizadas en las proximidades del campus de la Universidad Complutense de Madrid, ya que al ser una zona transitada existe una gran variedad de vehículos, y fué posible capturar todo tipo de distintivos. El repositorio original de imágenes consta inicialmente de 45 fotos, debido a que recoger más fotografías no aportaba más información al caso de uso al ser siempre el mismo escenario.

Tras analizar las fotografías encontramos que en algunos casos el distintivo no estaba colocada correctamente, apareciendo volteada 90 grados a derecha o izquierda, revertida, etc. Para cubrir este caso de uso y aprovechar a realizar un dataset que contemple todas las opciones, tratamos de replicar este suceso de forma genérica sobre todo el conjunto de fotografías iniciales, generando varias versiones de las mismas. A este proceso se le conoce como *data augmentation,* se basa en aumentar la colección de datos inicial de entrenamiento generando varias versiones de estos, de esta forma se reduce el coste humano de generar casos de uso.

Para este proceso de *data augmentation* haremos uso de una librería Python de código abierto llamada Pillow, que es un fork de la versión original PIL la cual solo tenía soporte hasta Python 2.7 y está desarrollada por Alex Clark, en colaboración con otros programadores.

```
from PIL import Image, ImageOps

img_files = glob.glob(str(Path(".") / "imgs" / "*.jpg"))

for img in img_files:
    im = Image.open(img)
    name = img.strip('.jpg')
    im_flip = ImageOps.flip(im)
    im_flip.save(str(name + '_flip.jpg'), quality=100)
    im_mirror = ImageOps.mirror(im)
    im_mirror.save(str(name + '_mirror.jpg'), quality=100)
```

Con esta sección de código iteramos sobre todas las imágenes del repositorio y aplicamos la transformación de girar 90 grados y efecto espejo para ver la imagen revertida. Tras finalizar este proceso, tendríamos un dataset más informado.

6.1.1 Etiquetado

Llegados a este punto ya tendríamos una colección de imágenes inicial, pero aún no estaría lista para el entrenamiento de una red neuronal.

Como se ha citado previamente, una red neuronal clasifica un dato de entrada en una clase de salida. Para que esto sea posible antes de la fase de entrenamiento es necesario que los datos a utilizar estén previamente clasificados o etiquetados, para que las neuronas de las capas internas *aprendan* y sean capaces de asociar una entidad a su correspondiente etiqueta.

Para esta tarea nos ayudaremos de LabelImg[15], una herramienta con versiones en Linux y Windows que permite de forma sencilla realizar el etiquetado de imágenes.

[15] https://tzutalin.github.io/labelImg/

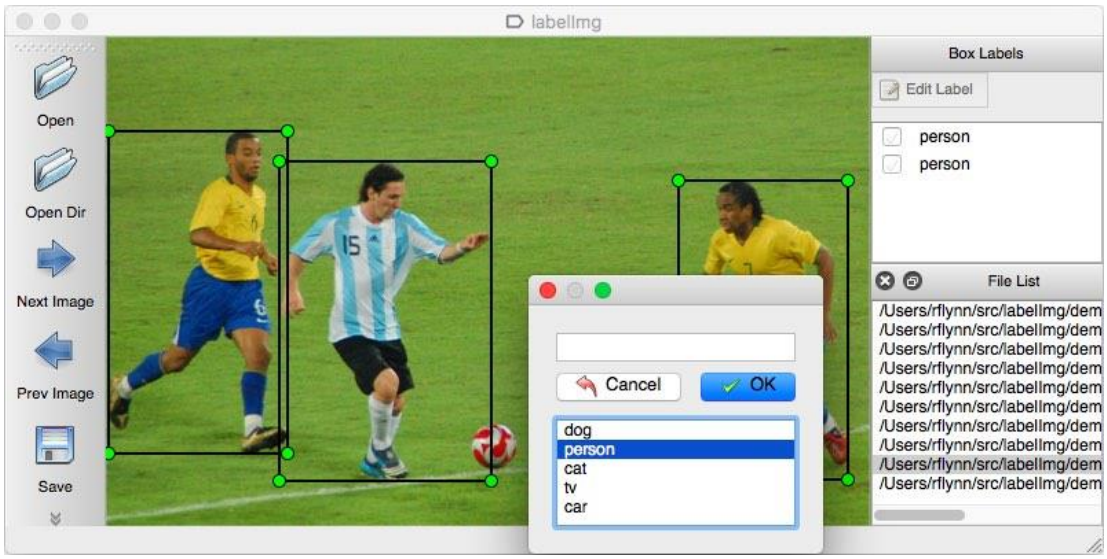


Imagen 29: Ejemplo del programa de etiquetado LabelImg

Para el caso de la detección de objetos, el proceso consiste en marcar la entidad con un recuadro y asignarle a mano el nombre de la clase correspondiente, de esta manera en la fase de entrenamiento la red neuronal sabe dónde localizar la entidad. En el caso de la clasificación de imágenes solo es necesario crear un fichero excel o csv, que contenga una tabla con el nombre de todas las imágenes y una columna adyacente indicando la etiqueta asociada a la imagen. Al finalizar la tarea, el programa creará una serie de ficheros que contendrán toda la información asociada al etiquetado: nombre de la imagen, coordenadas del recuadro y la etiqueta correspondiente.

Información que se almacena en los ficheros:

```
class BoundingBox(NamedTuple):
    xmin: int
    ymin: int
    xmax: int
    ymax: int
    label: int
```

Etiquetas correspondientes a nuestro proyecto:

```
LABELS_STR = {
    "0E": 1,
    "B": 2,
    "C": 3,
    "ECO": 4
}
```

6.1.1.1 TFRecords

En Tensorflow es necesario realizar un paso extra antes de utilizar los datos para entrenamiento, que consiste en generar una serie de archivos binarios, *tfrecords*, que comprime toda la información extraída en el etiquetado. La finalidad de este formato es agrupar toda la información de dataset en un único archivo evitando tener que manejar el repositorio de carpetas en las tareas de entrenamiento y evaluación.

6.1.1.2 Roboflow

Para la creación de estos archivos haremos uso de *Roboflow*, una herramienta online gratuita, muy común en el ámbito de la visión artificial. Esta página nos permite subir nuestro repositorio de imágenes etiquetado, indicando el formato utilizado. Además, también integra la opción de etiquetar nuevas fotos que se agreguen a la colección y ofrecerte información acerca de las imágenes y el número de clases utilizadas.

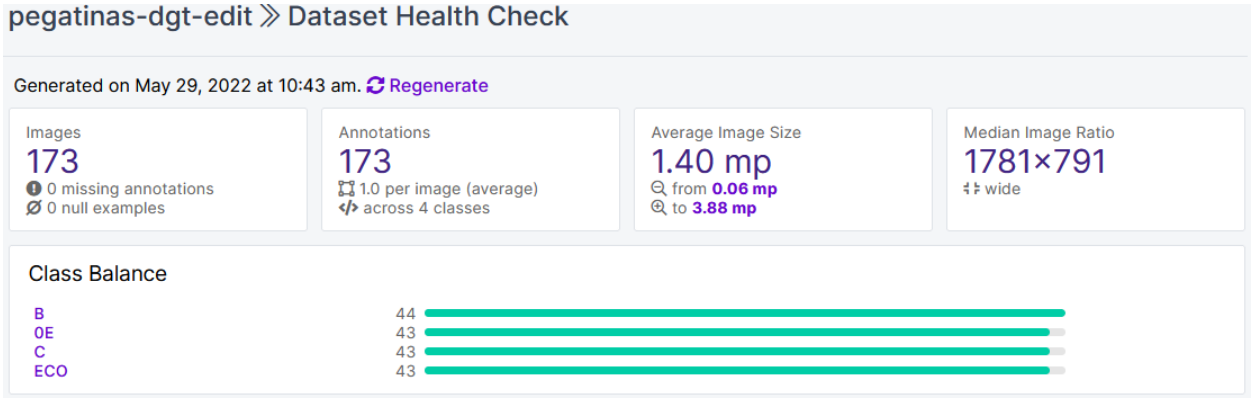


Imagen 30: Información relativa al dataset

Entre todas las opciones que ofrece y el principal motivo de su uso, es la generación de diferentes versiones, es decir, permite crear modificaciones independientes del dataset en función de la resolución, filtros de imágenes, etc. y exportar estas versiones a un formato específico para uso en el entrenamiento.

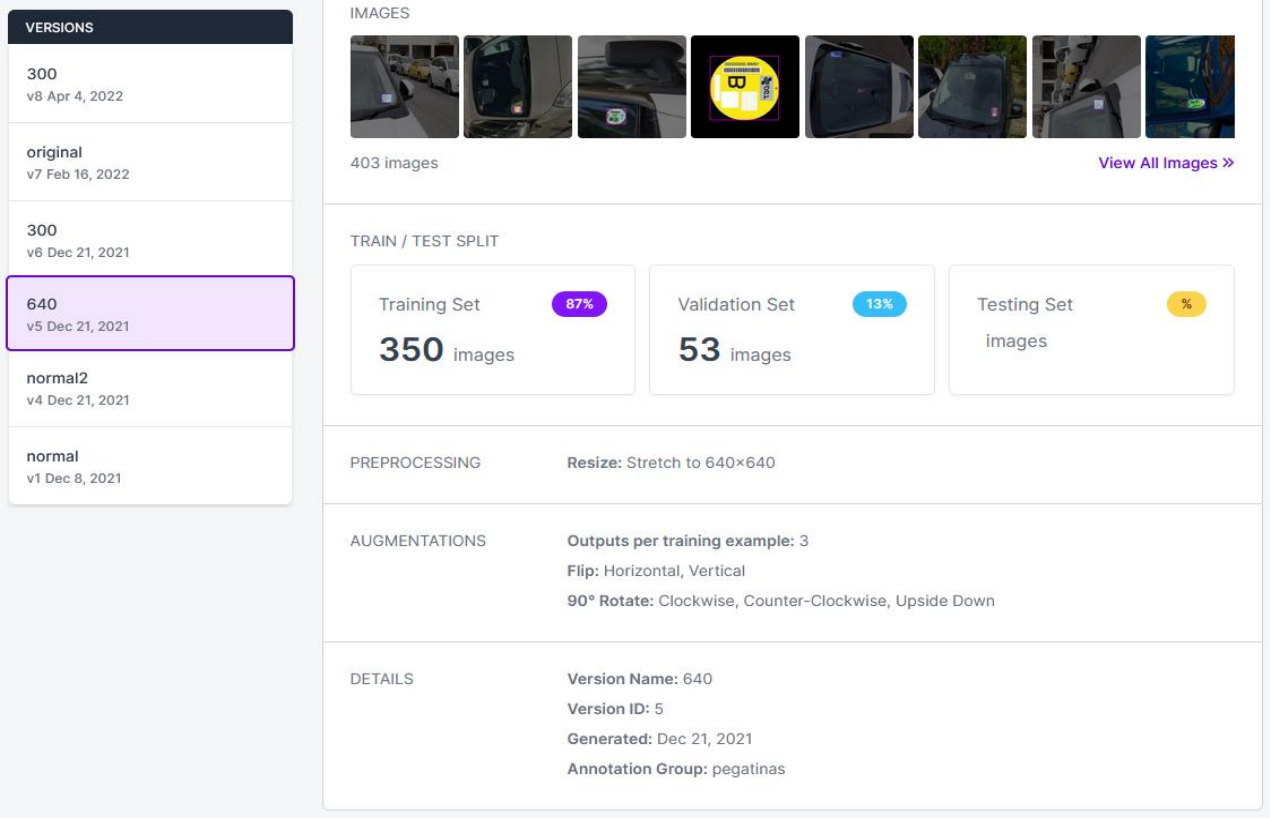


Imagen 31: Diferentes versiones del dataset en función de su resolución

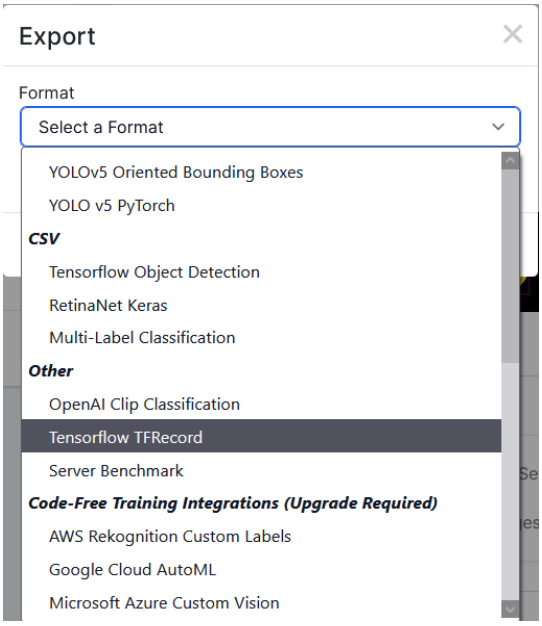


Imagen 32: Ejemplo de cómo exportar el dataset a TFRecord

6.2 Entrenamiento

Para la fase de entrenamiento y detección de distintivos medioambientales haremos uso de las versiones de Tensorflow 1 y Tensorflow 2 de la red neuronal preentrenada ssd_*mobilenet_v2* cuyas resoluciones son 300x300 y 640x640 respectivamente. Este modelo es de los más populares en el propósito de la detección de objetos en dispositivos de bajo consumo y cómputo como es nuestro caso.

6.2.1 Entrenamiento en Python

Una vez explicado cómo se genera el dataset, y todo lo relevante al funcionamiento del modelo utilizado, procedemos a exponer el entrenamiento del modelo de manera práctica en un entorno de python, haciendo uso de la api Object Detection de Tensorflow, tanto para la versión TF1 como TF2.

Antes de realizar el entrenamiento primero procedemos a descargar la versión preentrenada de ambos modelos del repositorio de *Tensorflow Detection Model Zoo.*

COCO-trained models TF1[16]

Model name	Speed(ms)	COCO mAP	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_mobilenet_v1_quantized_coco ☆	29	18	Boxes
ssd_mobilenet_v1_0.75_depth_quantized_coco ☆	29	16	Boxes
ssd_mobilenet_v1_ppn_coco ☆	26	20	Boxes
ssd_mobilenet_v1_fpn_coco ☆	56	32	Boxes
ssd_resnet_50_fpn_coco ☆	76	35	Boxes
ssd_mobilenet_v2_coco	31	22	Boxes
ssd_mobilenet_v2_quantized_coco	29	22	Boxes
ssdlite_mobilenet_v2_coco	27	22	Boxes
ssd_inception_v2_coco	42	24	Boxes

[16]*https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md*

Tabla 1: Redes SSD mobilnet para la version TensorFlow 1

COCO-trained models TF2[17]

Model name	Speed(ms)	COCO mAP	Outputs
SSD MobileNet V2 FPNLite 640x640	39	28.2	Boxes
SSD ResNet50 V1 FPN 640x640 (RetinaNet50)	46	34.3	Boxes
SSD ResNet50 V1 FPN 1024x1024 (RetinaNet50)	87	38.3	Boxes
SSD ResNet101 V1 FPN 640x640 (RetinaNet101)	57	35.6	Boxes
SSD ResNet101 V1 FPN 1024x1024 (RetinaNet101)	104	39.5	Boxes
SSD ResNet152 V1 FPN 640x640 (RetinaNet152)	80	35.4	Boxes
SSD ResNet152 V1 FPN 1024x1024 (RetinaNet152)	111	39.6	Boxes
Faster R-CNN ResNet50 V1 640x640	53	29.3	Boxes
Faster R-CNN ResNet50 V1 1024x1024	65	31.0	Boxes
Faster R-CNN ResNet50 V1 800x1333	65	31.6	Boxes

Tabla 2: Redes SSD mobilnet para la version TensorFlow 2

Con los siguientes comandos de consola descargamos el modelo y los descomprimimos dentro de la ruta *training/runs*, junto con el resto de los modelos.

```
!wget -N
http://download.tensorflow.org/models/object_detection/{MODEL_NAME}.tar.gz -P
{MODELS_FOLDER.absolute()}
!tar -xvf {MODELS_FOLDER.joinpath(MODEL_NAME).with_suffix(".tar.gz")} -C
{MODELS_FOLDER.absolute()}
```

[17] *https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md*

6.2.2 Tensorflow 1 Training

Para el caso de tf1 lo haremos mediante secciones de código usando a mano las librerías de la API.

```
with tf.Session():
    tf.set_random_seed(RANDOM_SEED)
    config = tf.estimator.RunConfig(str(CURRENT_RUN_FOLDER))

      train_and_eval_dict =
object_detection.model_lib.create_estimator_and_inputs(
      run_config=config,
      hparams=object_detection.model_hparams.create_hparams(),
      pipeline_config_path=str(CONFIG_FILE.absolute()),
      sample_1_of_n_eval_examples=1
    )

    estimator = train_and_eval_dict['estimator']
    train_input_fn = train_and_eval_dict['train_input_fn']
    eval_input_fns = train_and_eval_dict['eval_input_fns']
    eval_on_train_input_fn = train_and_eval_dict['eval_on_train_input_fn']
    predict_input_fn = train_and_eval_dict['predict_input_fn']
    train_steps = train_and_eval_dict['train_steps']

    train_spec, eval_specs =
object_detection.model_lib.create_train_and_eval_specs(
        train_input_fn,
        eval_input_fns,
        eval_on_train_input_fn,
        predict_input_fn,
        train_steps,
        eval_on_train_data=False)

    tf.estimator.train_and_evaluate(estimator, train_spec, eval_specs[0])
```

6.2.3 Tensorflow 2 training

En esta versión haremos usos de las scripts definidas en la api, que incorporan el código que aparece anteriormente adaptado a las librerías de Tensorflow 2.

```
python models/research/object_detection/model_main_tf2.py \
--model_dir=training/runs/ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8\
--sample_1_of_n_eval_examples=1 \
--pipeline_config_path=pipeline.config \
--alsologtostderr
```

6.2.4 Configuración de entrenamiento

En ambos casos es necesario especificar un archivo, *pipeline.config*[18], que contiene la configuración de la red neuronal, y donde se indica la ruta de los ficheros necesarios para el entrenamiento. Este fichero viene por defecto en la carpeta del modelo pre entrenado, donde es necesario adaptar una serie de parámetros en función de nuestro problema.

```
model {
  ssd {
      num_classes: 4
      box_coder {
        faster_rcnn_box_coder {
          y_scale: 10.0
          x_scale: 10.0
          height_scale: 5.0
          width_scale: 5.0
        }
      }
}
```

De este apartado es relevante cambiar la variable *num_classes*, que hace referencia al número de etiquetas utilizadas, en nuestro caso 4 que indica el tipo de distintivos que existen.

```
train_config: {
  batch_size: 64
  optimizer {
      adam_optimizer: {
      learning_rate: {
      cosine_decay_learning_rate {
            learning_rate_base: 0.001
            warmup_learning_rate: 0.00005
            warmup_steps: 3200
            total_steps: 25000
      }
      }
      }
  }
```

Aquí se indican los parámetros a utilizar en el entrenamiento:

- *batch_size*, es la cantidad de datos que la red neuronal utiliza para entrenar en una iteración. Este parámetro hay que ajustarlo a mano en función de la GPU que estemos utilizando ya que dependemos de la memoria que tenga.

- *learning_rate* [10], indica el grado de aprendizaje que va a seguir nuestro modelo en cada iteración. Este parámetro hará que nuestra red neuronal necesite más o menos iteraciones para el entrenamiento. Por defecto este parámetro se suele dejar en 0.001, ya que tras

[18] https://github.com/tensorflow/models/blob/master/research/object_detection/protos/optimizer.proto

varios estudios de la comunidad como *fast.ai*[19]*y kaggle*[20], se ha encontrado que es el mejor valor debido a que mantiene un aprendizaje constante. Este concepto es relevante puesto que si se ajusta mal puede hacer que nuestro modelo no "aprenda" de forma correcta siendo insuficiente si se ajusta a la baja o genere *overfitting* si se ajusta al alza, haciendo que el modelo en la fase de entrenamiento *memorice* el dataset y no sea capaz de encontrar las características que le hagan detectar nuevos datos.

- *total_steps*, número de iteraciones que se van a emplear para el entrenamiento. Es importante ser coherente con el número, ya que, podrías generar overfitting.

```
train_input_reader: {
  tf_record_input_reader {
      input_path: "pegatinas-dgt-6/train/pegatinas.tfrecord"
  }
  label_map_path: "pegatinas-dgt-6/train/pegatinas_label_map.pbtxt"
}

eval_input_reader: {
  tf_record_input_reader {
      input_path: "pegatinas-dgt-6/valid/pegatinas.tfrecord"
  }
  label_map_path: "pegatinas-dgt-6/valid/pegatinas_label_map.pbtxt"
}
```

Indicamos la ruta de los tfrecord y el fichero txt con el listado de las etiquetas.

6.2.5 Información del entrenamiento

Durante la fase de entrenamiento podemos hacer uso de la herramienta *TensorBoard*[21] para monitorizar y analizar el proceso de entrenamiento en tiempo real. Para lanzar esta herramienta basta con ejecutar un comando que nos desplegará un servicio en local, al cual podemos acceder desde nuestro navegador.

tensorboard --logdir='training_folder'

http://localhost:6006/

---

[19] https://www.fast.ai/

[20] https://www.kaggle.com/code/residentmario/tuning-your-learning-rate/notebook

[21] https://www.tensorflow.org/tensorboard?hl=es-419

Imagen 33: Interfaz web de TensorBoard

La interfaz web nos muestra una serie de gráficas con la evaluación y el entrenamiento del modelo, de las cuales son relevantes las dos siguientes:

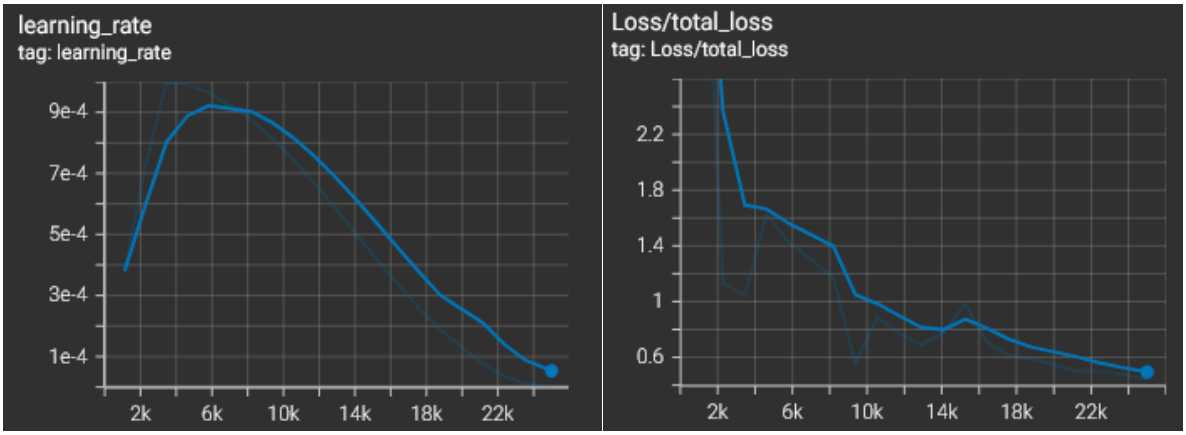


Imagen 34: Grafícas sobre el learning rate y el loss del modelo 300x300

En la gráfica de *learning rate*, podemos observar la evolución del entrenamiento durante el paso del tiempo y cómo a medida que avanzan el número de *steps* el índice de aprendizaje se reduce. Este resultado, junto con el loss, es importante a tener en cuenta ya que nos indica a partir de qué etapa de entrenamiento el modelo deja de adquirir más información válida y empieza a entrar en una fase de overfitting.

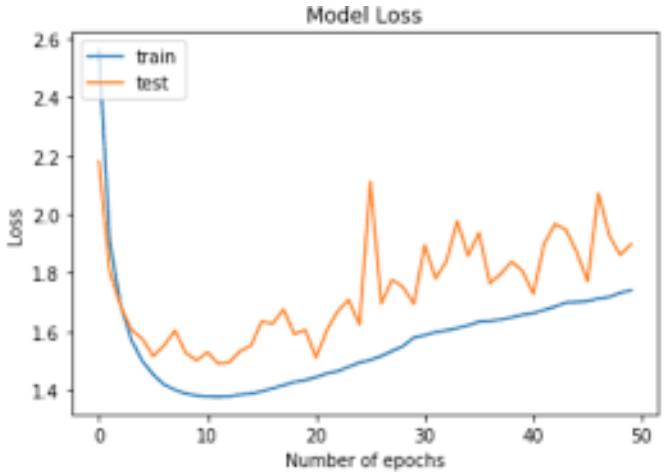


Imagen 35: Gráfica donde se observa el avance del entrenamiento en función de las épocas

En esta imagen de ejemplo, se puede apreciar un caso en el que el modelo sobre las 10 épocas de entrenamiento alcanza su índice más bajo de aprendizaje válido, y a partir de ese valor el loss vuelve a incrementar.

6.2.6 Post entrenamiento

Una vez finalizado el entrenamiento, en la carpeta donde hicimos el proceso tendremos un listado de *checkpoints*, que no son más que archivos que guardan el último estado de entrenamiento de la red neuronal y el cual se puede volver a utilizar para realizar un posterior entrenamiento, de esta forma tendríamos nuestro modelo *pre entrenado*.

Este formato del modelo aun no es útil para hacer inferencia, por lo que hay que *exportarlo* a un formato conocido como *frozen graph*. El cual es un binario que recoge la información de los pesos del último checkpoint alcanzado. En el caso de Tensorflow suele tener la extensión *model_name.pb*.

Para exportar el checkpoint a un *frozen graph*, seguiremos haciendo uso de las scripts que nos proporciona la api object_detection.

```
python models/research/object_detection/export_tflite_graph_tf2.py \
--trained_checkpoint_dir
{'training/runs/ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8'} \
--output_directory
{'training/runs/ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8/export'} \
--pipeline_config_path {'pipeline.config'}
```

Tras su ejecución se creará una carpeta *saved_model* en el directorio indicado, nuestro caso *export*, que contendrá el archivo binario.

Como se ha comentado en apartado anteriores, nuestro objetivo es integrar la red neuronal en un *edge device*, por lo que la versión del *frozen graph* no es práctica debido a su elevado peso y falta de optimización, lo que nos conlleva a transformarlo en una versión *tflite*, para su posterior evaluación.

```
python models/research/object_detection/export_tflite_graph_tf2.py \
--trained_checkpoint_dir
{'training/runs/ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8'} \
--output_directory
{'training/runs/ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8/export'} \
--pipeline_config_path {'pipeline.config'}
```

Para que el binario de *tflite* sea reconocido por android es necesario agregar *metadatos*[22] al modelo. Es la forma estándar de especificar la información del modelo, y que pueda ser interpretado para inferencia.

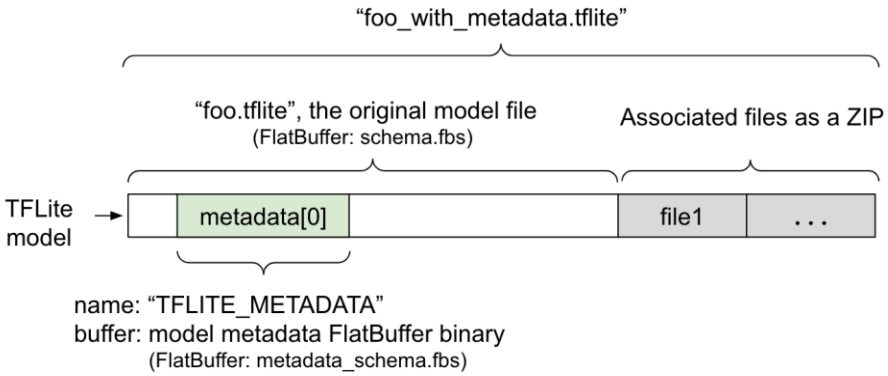


Imagen 36: Modelo TFLite con metadatos y archivos asociados

Este propósito se consigue haciendo uso de la función *metadata:writer* de Tensorflow, la cual se encarga de extraer la información del modelo y añadirla como metadato, además hay que agregar un fichero con el listado de las etiquetas

```
from tflite_support.metadata_writers import object_detector
from tflite_support.metadata_writers import writer_utils

writer = object_detector.MetadataWriter.create_for_inference(
      writer_utils.load_file(_TFLITE_MODEL_PATH), input_norm_mean=[127.5],
      input_norm_std=[127.5],
      label_file_paths=[_TFLITE_LABEL_PATH])
writer_utils.save_file(writer.populate(), _TFLITE_MODEL_WITH_METADATA_PATH)
```

[22] https://www.tensorflow.org/lite/models/convert/metadata

6.3 Inferencia

En la evaluación de un modelo hay que realizar una acción se conoce como *inferencia*, que se resume en pasar unos datos de entrada preprocesados y obtener un resultado.

Antes de realizar esta labor es necesario comprender que el modelo necesita recibir los datos de la misma forma que en la fase de entrenamiento, es decir, si utilizamos imágenes con una resolución de 320x320 en el entrenamiento, para la inferencia en tiempo real es necesario que las imágenes de entrada sean preprocesadas y se reescalen a la resolución deseada. Para ello haremos uso de una librería llamada *opencv*[23], muy común en el ámbito de la visión artificial, que proporciona diversas herramientas para el tratamiento de imágenes. Está desarrollada en el lenguaje de programación C, lo que la hace portable a cualquier sistema operativo y lenguaje, como python en nuestro caso.

Abarcado el concepto anterior procedemos a la primera fase, la carga del modelo. Esta se realiza nuevamente haciendo uso de las librerías de Tensorflow:

```
# Load the TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path=TFLITE_QUANT_MODEL)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

En input_details podemos observar la información de la primera capa de entrada:

```
[{'name': 'serving_default_input:0', 'index': 0, 'shape': array([  1, 640,
640,   3], dtype=int32), 'shape_signature': array([  1, 640, 640,   3],
dtype=int32), 'dtype': <class 'numpy.float32'>, 'quantization': (0.0, 0),
'quantization_parameters': {'scales': array([], dtype=float32),
'zero_points': array([], dtype=int32), 'quantized_dimension': 0},
'sparsity_parameters': {}}]
```

[23] https://opencv.org/

Lo correspondiente a la salida en outputs_details

```
[{'name': 'StatefulPartitionedCall:1', 'index': 386, 'shape': array([ 1, 10],
dtype=int32), 'shape_signature': array([ 1, 10], dtype=int32), 'dtype': <class
'numpy.float32'>, 'quantization': (0.0, 0), 'quantization_parameters':
{'scales': array([], dtype=float32), 'zero_points': array([], dtype=int32),
'quantized_dimension': 0}, 'sparsity_parameters': {}}, {'name':
'StatefulPartitionedCall:3', 'index': 382, 'shape': array([ 1, 10, 4],
dtype=int32), 'shape_signature': array([ 1, 10, 4], dtype=int32), 'dtype':
<class 'numpy.float32'>, 'quantization': (0.0, 0), 'quantization_parameters':
{'scales': array([], dtype=float32), 'zero_points': array([], dtype=int32),
'quantized_dimension': 0}, 'sparsity_parameters': {}}, {'name':
'StatefulPartitionedCall:0', 'index': 388, 'shape': array([1], dtype=int32),
'shape_signature': array([1], dtype=int32), 'dtype': <class 'numpy.float32'>,
'quantization': (0.0, 0), 'quantization_parameters': {'scales': array([],
dtype=float32), 'zero_points': array([], dtype=int32), 'quantized_dimension':
0}, 'sparsity_parameters': {}}, {'name': 'StatefulPartitionedCall:2', 'index':
384, 'shape': array([ 1, 10], dtype=int32), 'shape_signature': array([ 1, 10],
dtype=int32), 'dtype': <class 'numpy.float32'>, 'quantization': (0.0, 0),
'quantization_parameters': {'scales': array([], dtype=float32), 'zero_points':
array([], dtype=int32), 'quantized_dimension': 0}, 'sparsity_parameters': {}}]
```

La información relevante es conocer las dimensiones de la capa de entrada y el tipo de dato utilizado, 640 y float32, y la estructura de la capa de salida. Como se ha comentado anteriormente nuestra red neuronal incorpora la capa *NMS* a su salida y además de eliminar las repeticiones en la detección también tiene un formato de salida que facilita recoger la información de las predicciones.

Esta información se divide en 4 arrays:

- StatefulPartitionedCall:3, [1, 10, 4] # bounding box, recuadros
- StatefulPartitionedCall:2, [1, 10] # classes
- StatefulPartitionedCall:1, [1, 10] # scores
- StatefulPartitionedCall:0, [10] # n detections

La posición de los arrays puede variar en función del modelo, en algunos casos están ordenados y en ocasiones hay que observar la información en output_details.

Una vez cargado el modelo, procedemos a enviarle datos con el objetivo de clasificarlos. Para ello haremos uso del siguiente código:

```
import cv2

LABELS = {
    0: "0E",
    1: "B",
    2: "C",
    3: "ECO",
}

def model_inference(image_path: str):
    # Read the image and decode to a tensor
    img = cv2.imread(image_path)
    img = cv2.resize(img,(input_details[0]['shape'][1],
input_details[0]['shape'][2])) # Preprocess the image to required size and
cast
    img = np.asarray(img)/255
    img = np.reshape(img,(1,input_details[0]['shape'][1],
input_details[0]['shape'][2],3))
    input_data = np.array(img, dtype=np.float32)

    #set the tensor to point to the input data to be inferred
    interpreter.set_tensor(input_details[0]['index'], input_data)

    # Run inference
    interpreter.invoke()
    # Extract data from detection array, StatefulPartitionedCall:0
    output_data = int(interpreter.get_tensor(output_details[2]['index'])
    # SSD Mobilenet tflite model returns 10 boxes by default.
    # boxes =
interpreter.get_tensor(output_details[0]['index'])[0][:output_data]
    classes =
interpreter.get_tensor(output_details[3]['index'])[0][:output_data]
    scores =
interpreter.get_tensor(output_details[0]['index'])[0][:output_data]
    top_k = np.argmax(scores)
    label = classes[top_k]

    return scores[top_k], LABELS[label]
```

Mediante esta función recibimos una imagen, se reescala a la dimensión de la primera capa y se envía a la entrada. Extraemos la información de la salida, y del listado de scores buscamos el que tenga una puntuación más elevada. Una vez conseguido devolvemos el valor correspondiente a esa posición en el resto de las listas, ya que boxes[i], classes[i] y scores[i] hacen referencia al mismo elemento.

Un ejemplo de ejecución del Código anterior, mostrando el contenido de las listas sería el siguiente:

```
score, label = model_inference('pegatinas-dgt-
7/valid/eco_flip_jpg.rf.5a6ac123d1d0c20f44912dd53dba1fbf.jpg')
print(score, label)
```

```
10
[3. 3. 1. 3. 1. 2. 0. 2. 0. 1.]
[0.9999672  0.13977176 0.07906932 0.02971154 0.02800533 0.02631041
 0.02521428 0.02509114 0.0190798  0.01715925]

0.9999672 ECO
```

El primer valor hace referencia al número de detecciones, que por defecto en los modelos SSD es 10. La segunda lista es el identificador numérico de la etiqueta correspondiente y por último el listado con las probabilidades de cada etiqueta. Como se puede apreciar, la puntuación más alta es 0.99 asociada a la posición 0 de la lista de etiquetas, que cuyo valor es el 3 y la etiqueta *ECO*, y esta sería nuestra predicción final.

6.4 Evaluación del entrenamiento y validación

En la fase de entrenamiento hacemos una separación en el dataset, seleccionando una parte para entrenamiento y otra colección dedicada a validar los resultados haciendo inferencia sobre el mismo. A este proceso se le conoce como *validación cruzada*, consiste en iterar sobre todos los elementos del conjunto de pruebas para obtener las predicciones del modelo y comparar el resultado de la clasificación con la etiqueta que le corresponde.

Ayudándonos de la función definida anteriormente, la única tarea restante es la de iterar sobre la colección de imágenes de validación y guardar los resultados de inferencia.

```
folder = Path(".") / "pegatinas-dgt-7" / "valid"
data = df = pd.read_csv(Path(folder) / "_annotations.csv")
predictions = list()

for img in data.iterrows():
    score, label = model_inference(str(Path(folder) / img[1]['filename']))
    valid = 0
    if label == img[1]['class']:
        valid = 1
    dict_a = {"Image": img[1]['filename'], "Score": score, "Label": label,
"Valid": valid}
    predictions.append(dict_a)

pred = pd.DataFrame(predictions, columns=["Image", "Score", "Label",
"Valid"], index=False)
avg = pred['Valid'].mean()
```

```
pred.to_csv('predictions_50k.csv')
print("Accuracy:", avg)
```

El proceso consiste en enviar a inferencia cada imagen de la colección, recoger la predicción obtenida y comprobar si coincide con la que tenía asociada realmente. Para ello se añade una nueva columna *Valid* a la tabla donde marcamos con 1 ó 0 el acierto y al terminar calculamos la media de esa columna para obtener la tasa de acierto del modelo.

6.4.1 Precisión de los modelos

Nombre	Tiempo de entrenamiento	Número de etapas (steps)	Acierto
ssd_mobilenet_300	5h	25k	51%
ssd_mobilenet_300	7h	40k	47%
ssd_mobilenet_640	8h	25k	64%
ssd_mobilenet_640	12h	30k	79%
ssd_mobilenet_640	15h	50k	88.67%

Tabla 3: Resultados del entrenamiento de los modelos de 300x300 y 640x640

En los resultados anteriores se puede apreciar la importancia de la resolución en los modelos, haciendo que la versión de 640x640 sea capaz de recopilar información con más detalle. En el caso de la versión de 300x300 al aumentar el número de *steps* en el entrenamiento se produce un decremento de la precisión debido al overfitting ya que al utilizar una resolución menor la calidad de la información no es la suficiente y es contraproducente superar los 30k steps. En la versión TF2 de 640, encontramos los mejores valores próximos a los 50k.

Para analizar más a fondo cómo se comporta el modelo, es necesario obtener otra métrica, *matriz de confusión*. Gracias a ella podemos analizar con qué porcentaje de éxito predice cada una de las etiquetas, ya que tener un accuracy elevado no implica que nuestro modelo se comporte correctamente. Para ello obtenemos la matriz de las dos versiones del modelo en su mejor caso.

6.4.2 Métricas y evaluación

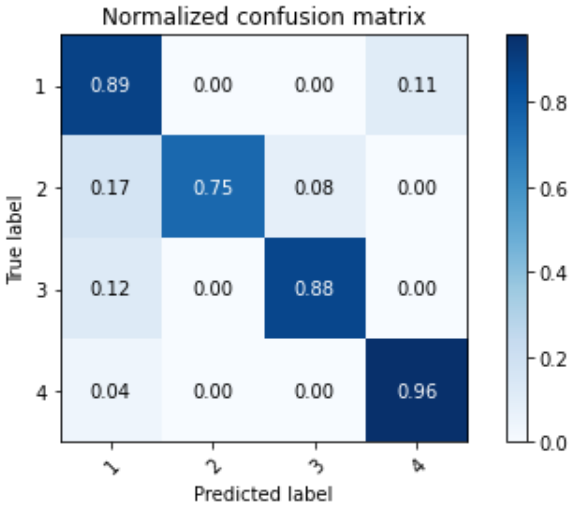


Imagen 37: Matriz de confusión del modelo de 640x640 tras 50000 steps

Esta matriz es la correspondiente a la versión de 640x640, como se ha comentado previamente, al ser de mayor resolución los resultados de la clasificación son correctos, sin un grado de confusión entre clases elevado. Esta matriz se asemeja al caso ideal donde todos los elementos se agrupan en la diagonal principal y se acercan al total, en este caso 1, ya que los resultados están normalizados entre 0 y 1.

A continuación, compararemos el resultado de las matrices del modelo de 300x300, donde a la izquierda se sitúa la versión entrenada con 25k steps y a la derecha con 40k steps.

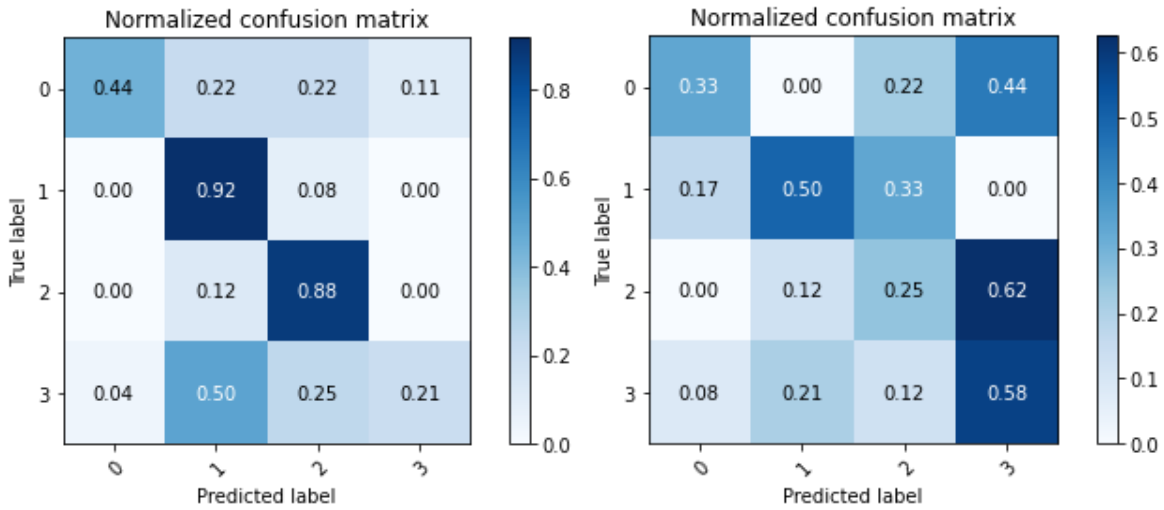


Imagen 38: Matrices de confusión del modelo de 300x300 tras 25000 y 40000 steps respectivamente

En ellas se puede apreciar que la clasificación no es tan correcta, como en la versión anterior, y tienden a confundir en mayor grado las etiquetas, en especial la 1 y 4 que corresponden con las 0E y ECO respectivamente que son entidades de cierta similitud. En la versión de mayor tiempo de entrenamiento, se puede apreciar un decremento del rendimiento debido a un posible overfitting.

De las matrices anteriores se pueden extraer 4 valores principales, que nos ayudan a analizar en profundidad el rendimiento del modelo.

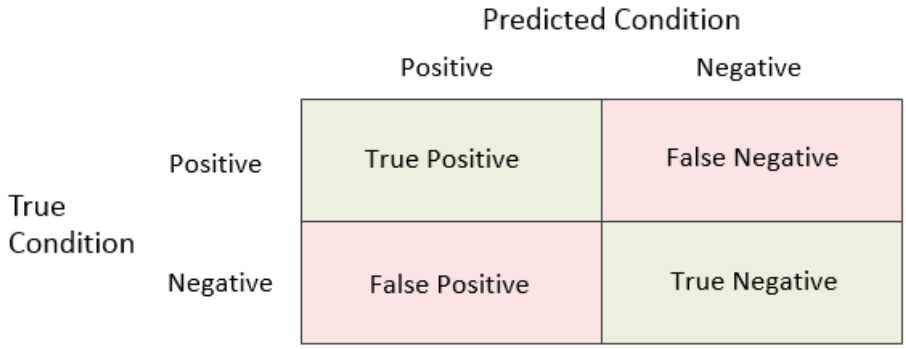


Imagen 39: Explicación grafica de los valores que componen una matriz de confusión

TN	FP
FN	TP

$$\text{Recall} = \frac{TP}{(TP + FN)}, \quad \text{Precision} = \frac{TP}{(TP + FP)}$$

$$\text{F-Score} = \frac{2 \cdot \text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}}, \quad \text{FP-rate} = \frac{FP}{(FP + TN)}$$

$$\text{Correlation Coefficient (CC)} = \frac{TP \cdot TN - FN \cdot FP}{\sqrt{(TP + FN)(TN + FP)(TP + FP)(TN + FN)}}$$

Imagen 40: Diferentes métricas que podemos extraer de los resultados de una matriz de confusión

- *Precisión*: De todos los valores predichos, cuantas veces se ha acertado.
- *Recall*: De todos los valores positivos, cuantas veces se ha acertado.
- *F1-score*: Combina los resultados de las medias de precisión y recall.

Métricas red 640 (50k steps)

%	0E	B	C	ECO
Precisión	66.67	1	87.5	95.83
Recall	88.88	75	87.5	95.83
F1	76.19	85.71	87.5	95.83

Tabla 4: Métricas de la red de 640x640 de 50000 steps

Media global: 88.67%

Media de la precisión: 87.5%

Media del recall: 85.71%

Media F1: 86.30%

Métricas red 320 (25k steps)

%	0E	B	C	ECO
Precisión	80	42.31	43.75	83.33
Recall	44.45	91.67	87.5	20.83
F1	57.14	57.89	58.34	33.33

Tabla 5: Métricas de la red de 300x300 de 25000 steps

Media global: 50.94%

Media de la precisión: 62.34%

Media del recall: 61.12%

Media F1: 51.67%

Métricas red 320 (40k steps)

%	0E	B	C	ECO
Precisión	42.85	50	18.18	60.86
Recall	33.33	50	25	58.33
F1	37.5	50	21.05	59.57

Tabla 6: Métricas de la red de 300x300 de 40000 steps

Media global: 47.16%

Media de la precisión: 42.97%

Media del recall: 41.67%

Media F1: 42.03%

6.4.3 Conclusiones sobre la matriz de confusión

En la red neuronal de mayor resolución se puede apreciar una tasa de aciertos alta, observando que los distintivos ambientales que más confunde son las etiquetas 0E, con el resto de las etiquetas, y ECO, solo con el distintivo 0E.

Sin embargo, en la red neuronal de menor resolución podemos ver como el distintivo 0E tiene un valor de falsos negativos muy alto, por lo que podemos decir que es te es confundido con facilidad. Mientras que el valor que menos precisión tiene es la etiqueta C, ya que, es el valor seleccionado mayormente por las demás etiquetas en los falsos positivos. Y, por último, comentar que a la etiqueta ECO se le atribuye en su mayoría una predicción del valor B.

Capítulo 7 - Desarrollo de la aplicación

En este capítulo se documenta tanto la creación de la aplicación, como la instalación de la misma en la cámara. Además del despliegue de la aplicación sobre la cámara, se realiza una evaluación de la aplicación en un entorno real.

7.1 Creación del proyecto

En la documentación de Azena, disponemos de un listado de aplicaciones de ejemplo que sirven de partida para nuestro propósito, detección de objetos con *tflite*.

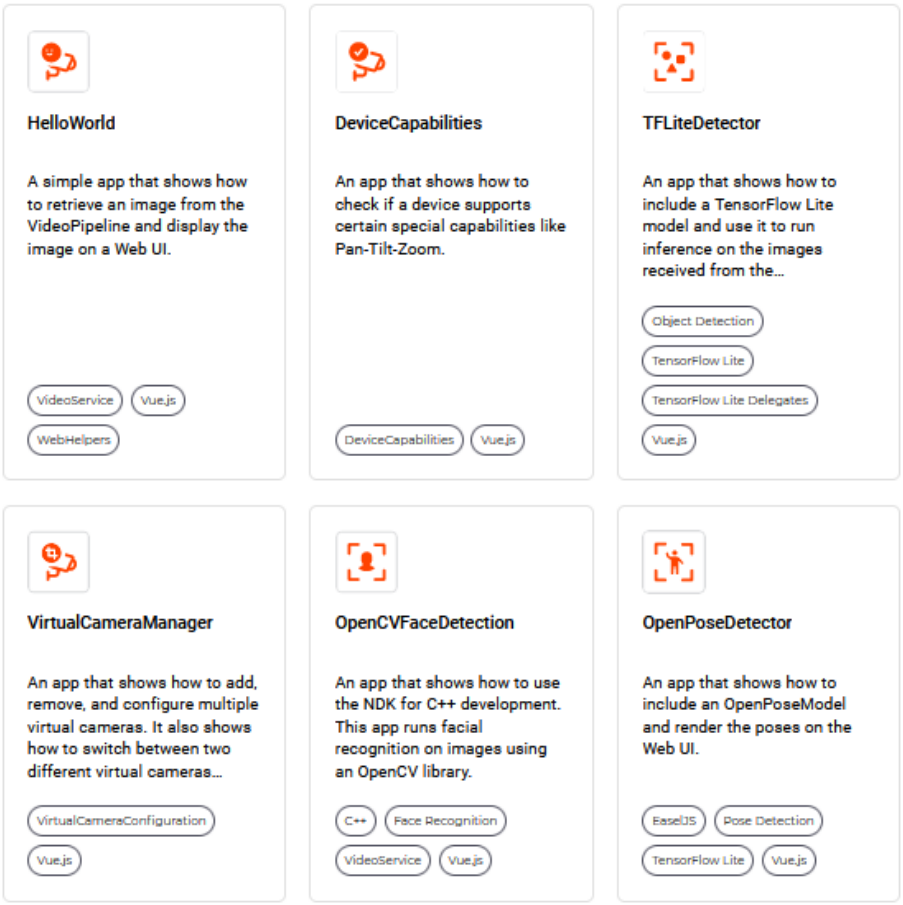


Imagen 41: Aplicaciones de ejemplo de Azena[24]

[24] https://docs.azena.com/developer/example_apps/overview

La estructura del proyecto en la aplicación se divide en dos partes, una sección correspondiente al código Java destinado a la integración del modelo e inferencia, y por otro lado encontramos el código referente al servicio web para visualizar la aplicación y el cual no es necesario modificar.

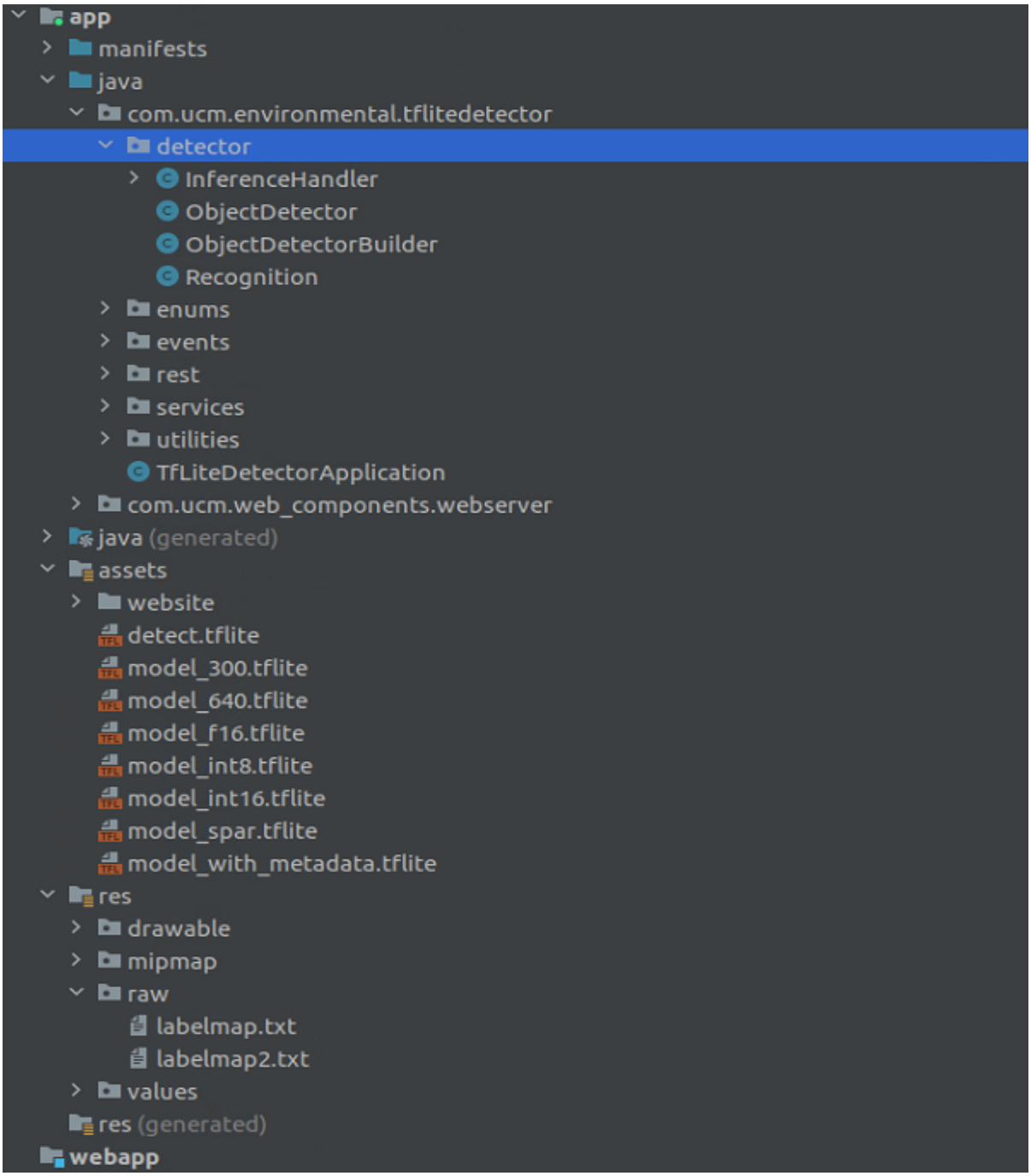


Imagen 42: Directorios de la aplicación, donde aparece resaltado detector

En la estructura del proyecto destacan tres directorios principales, *detector* donde se localiza el código de instancia del modelo y la inferencia del mismo, en *assets* encontramos el modelo, y por último la carpeta *res/raw* donde colocaremos el fichero txt con las etiquetas, las cuales tienen que tener el orden correcto para evitar resultados erróneos.

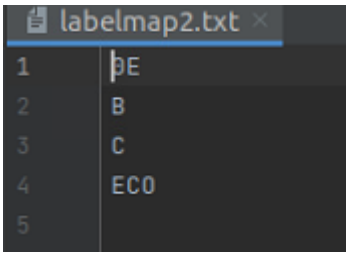


Imagen 43: Fichero .txt donde aparecen nombradas en orden los distintivos de la DGT

Dentro del directorio *detector* tiene principal relevancia *InferenceHandler* donde crearemos la instancia de nuestro modelo mediante la clase ObjectDetectorBuilder, que define los atributos del modelo.

```
public class ObjectDetectorBuilder {
      private String mModelFileName = "detect.tflite";
      private int mLabelFileResId = R.raw.labelmap;
      @SuppressWarnings("MagicNumber")
      private int mMaxDetectionsPerImage = 10;
      @SuppressWarnings("MagicNumber")
      private int mInputSize = 300;
      @SuppressWarnings("MagicNumber")
      private int mNumThreads = 4;
      private boolean mAllowFp16PrecisionForFp32 = false;
      private boolean mIsQuantized = false;
      private AccelerationType mAccelerationType = AccelerationType.AUTO;
```

En ObjectDetector se define el comportamiento de la inferencia del modelo y cómo procesa las salidas del mismo. En esta clase destaca la función *List<Recognition> recognizeImage(final Bitmap bitmap)*, la cual preprocesa los frames del pipeline de video de la aplicación adaptándolos a la entrada del modelo y los manda al modelo para la inferencia.

```
final Object[] inputArray = {mImgData};

// Allocate buffers
mOutputLocations = new float[inputArray.length][mMaxDetectionsPerImage][4];
mOutputClasses = new float[inputArray.length][mMaxDetectionsPerImage];
mOutputScores = new float[inputArray.length][mMaxDetectionsPerImage];
mDetectionCount = new float[inputArray.length];
final Map<Integer, Object> outputMap = new HashMap<>();

/* ADAPTAR AL MODELO */
//    outputMap.put(0, mOutputLocations);
//    outputMap.put(1, mOutputClasses);
//    outputMap.put(2, mOutputScores);
//    outputMap.put(3, mDetectionCount);

outputMap.put(1, mOutputLocations);
outputMap.put(3, mOutputClasses);
outputMap.put(0, mOutputScores);
outputMap.put(2, mDetectionCount);

mModel.runForMultipleInputsOutputs(inputArray, outputMap);
```

Esta última sección de código es de principal relevancia, ya que es donde tenemos que indicar las posiciones de los arrays de salida. Como se comentó anteriormente, estas pueden estar desordenadas en función de la red neuronal utilizada, en nuestro caso tenemos la versión adaptada para el modelo de 300x300 y 640x640 respectivamente.

Una vez preprocesada la imagen y definidas las salidas a utilizar, se manda la información a inferencia y se rellena el *HasMap outputMap* con los resultados obtenidos. Estos son agregados a una lista, la cual incluye las detecciones obtenidas con sus respectivos *bounding boxes, classes y scores*.

```
for (int i = 0; i < mMaxDetectionsPerImage; ++i) {
    // Return coordinates relative to the detection area
    final RectF detection =
       new RectF(
       mOutputLocations[0][i][1],
       mOutputLocations[0][i][0],
       mOutputLocations[0][i][3],
       mOutputLocations[0][i][2]);

    recognitions.add(
       new Recognition(
       String.valueOf(i),
       mLabels.get((int)mOutputClasses[0][i]),
       mOutputScores[0][i],
       detection));
}
```

7.2 Compilación e instalación de la aplicación

Para compilar la aplicación disponemos de un ejecutable que nos permite crear las dos versiones de la aplicación, *debug* o *release.*

```
./gradlew assembleDebug
./gradlew assembleRelease
```

Para instalar y probar la aplicación existen dos alternativas de comandos haciendo uso de la herramienta adb, proporcionada en el paquete SDK de Azena:

Conectándonos directamente a nuestro dispositivo físico mediante la ip que se le haya asignado en la configuración. Utilizando el siguiente comando:

```
adb connect <ip_address_of_device>:5555
```

Y hacer uso de un dispositivo virtual, con:

```
adb forward tcp:8443 tcp:8443
```

Una vez conectados a nuestro dispositivo, procedemos a instalar la aplicación por consola.

```
adb install -r -g ~/work/helloworld/app/build/outputs/apk/debug/app-debug.apk
```

Para poder visualizarla accedemos a la url *https://<ip_address_of_device>:8443*, desde un navegador web. De esta manera tendríamos la aplicación en funcionamiento.

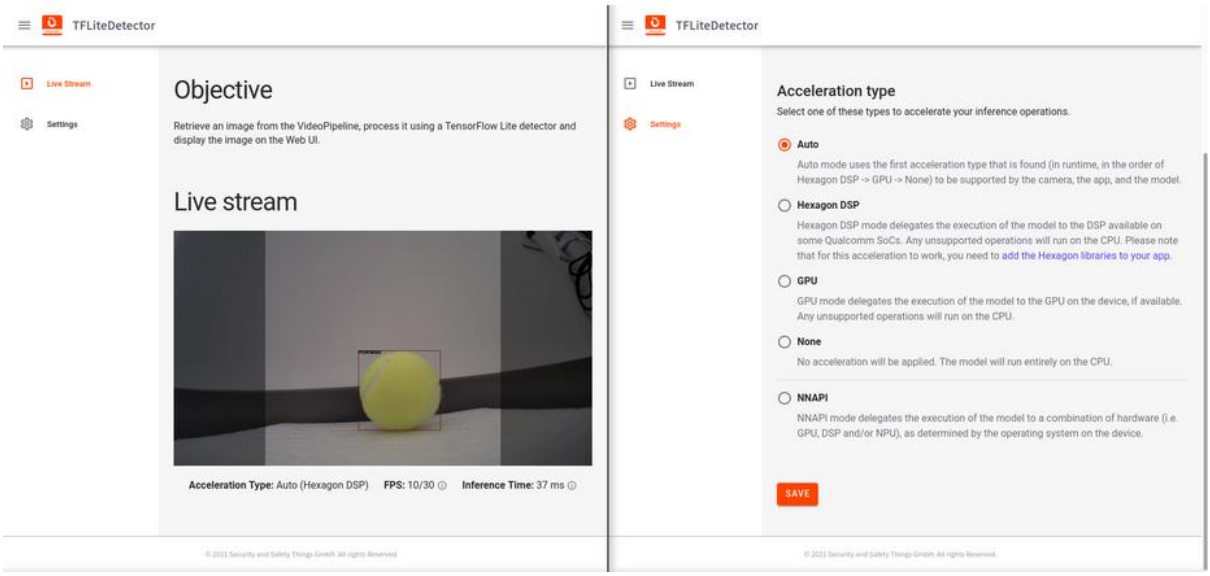


Imagen 44: Visualización de la aplicación instalada desde la interfaz web

7.3 Despliegue y evaluación de la cámara

Tras analizar las distintas versiones de los modelos disponibles en el repositorio de TensorFlow model zoo, nos decantamos por las versiones de la red ssd mobilenet ya que son las que menor tiempo de inferencia tienen y respetan la estructura final en la salida, de los cuatro arrays.

Nombre	**Resolución**	**Inferencia Volta-100**	**Inferencia Cámara**
ssd_mobilenet_v2	300x300	0.18s	45ms/16fps
ssd_mobilenet_v2	640x640	0.26s	245ms/6fps

Tabla 7: Tiempo de inferencia de los modelos en la GPU y la cámara

Las redes seleccionadas para el desarrollo del modelo son dos redes SSD, single shot detector. Se ha seleccionado este tipo de red neuronal ya que para los objetivos propuestos creemos es la más acertada, porque nos permite una detección de múltiples objetos en la imagen simultáneamente. Y, específicamente, se ha seleccionado esta red en concreto porque su tiempo de inferencia es el más bajo y respeta la estructura final en la salida, en concreto, contiene la capa final NMS.

Esta capa es necesaria a la hora de la creación de la aplicación, ya que nos devuelve la información de la forma requerida a la hora de añadir metadatos al modelo, en su versión de TensorFlowLite.

Otros modelos no pertenecientes a la rama de mobile_net, que no utilizaban esta capa, daban error al añadir los metadatos, lo cual, los incapacita a la hora de crear la aplicación.

Con el tipo de red elegido, sigue la selección de la resolución de las imágenes con las que el modelo va a trabajar. La opción elegida en un primer momento, solo habiendo tenido en cuenta el tiempo de inferencia y la precisión del modelo sobre la tarjeta gráfica Volta-100, era la utilización del modelo con mayor resolución. Aunque el tiempo de inferencia fuera mayor, no creímos que fuera lo suficientemente superior en relación a la precisión ganada.

Esta decisión cambia a la hora de probar la aplicación en la cámara, puesto que, el tiempo de inferencia es demasiado alto y provocaba una caída de los fotogramas detectados por segundo. Esto, extrapolado al caso real del proyecto, puede provocar la pérdida de información, por ejemplo, si un vehículo viaja a una velocidad demasiado alta podría no ser captado a tiempo.

Por este motivo consideramos que el modelo más óptimo, en un entorno real, es el de menor resolución ya que en pruebas sobre la cámara creemos que da un rendimiento aceptable a la hora de realizar la inferencia y capturar imágenes en tiempo real. Aunque, al tener una resolución tan baja se pierde precisión, lo que implica un mayor riesgo de error como se ha visto en la matriz de confusión.

Capítulo 8 - Propuestas post-entrenamiento

En este apartado se exponen dos alternativas al proyecto principal. Siendo la primera una agrupación de las dos etiquetas menos contaminantes y, la segunda, un recorte de la luna delantera con el fin de mejorar la precisión.

8.1 Primera propuesta: Agrupación de etiquetas

Tras analizar los resultados de entrenamiento en ambos modelos se plantea como propuesta la opción de reducir el número de etiquetas a utilizar, combinando las clases *ECO* y *0E* en una misma clase, ya que en casos prácticos los vehículos de ambas categorías son similares y de baja contaminación.

Para realizar este estudio haremos uso de otra API distinta conocida como *Yolov5*[25], debido a su gran facilidad en la creación de distintas versiones de redes neuronales.

Yolo, *you only look once*, es un conjunto de librerías y arquitecturas de redes neuronales, orientadas a la detección de objetos y basadas en Pytorch, framework similar a TensorFlow para la creación de modelos de deep learning haciendo uso de GPUs y CPUs con la diferencia de que es de código abierto. Inicialmente fue desarrollada por la empresa *Ultralytics en 2014* y hoy en día tiene un gran reconocimiento y participación dentro de las comunidades de inteligencia artificial y software libre.

Al igual que en TensorFlow, en YOLO disponemos de una serie de redes pre entrenadas, con distintas arquitecturas que van desde la versión más minimalista a nivel de capas *Yolov5n* a la más compleja *yolov5x6*.

Model	**size** (pixels)	**mAP^{val}** 0.5:0.95	**mAP^{val}** 0.5	**Speed** CPU b1 (ms)	**Speed** V100 b1 (ms)	**Speed** V100 b32 (ms)	**params** (M)	**FLOPs** @640 (B)
YOLOv5n	640	28.0	45.7	**45**	**6.3**	**0.6**	**1.9**	**4.5**
YOLOv5s	640	37.4	56.8	98	6.4	0.9	7.2	16.5
YOLOv5m	640	45.4	64.1	224	8.2	1.7	21.2	49.0
YOLOv5l	640	49.0	67.3	430	10.1	2.7	46.5	109.1

[25] *https://github.com/ultralytics/yolov5*

YOLOv5x	640	50.7	68.9	766	12.1	4.8	86.7	205.7
YOLOv5n6	1280	36.0	54.4	153	8.1	2.1	3.2	4.6
YOLOv5s6	1280	44.8	63.7	385	8.2	3.6	12.6	16.8
YOLOv5m6	1280	51.3	69.3	887	11.1	6.8	35.7	50.0
YOLOv5l6	1280	53.7	71.3	1784	15.8	10.5	76.8	111.4
YOLOv5x6 + TTA	1280 1536	55.0 **55.8**	72.7 **72.7**	3136 -	26.2 -	19.4 -	140.7 -	209.8 -

Tabla 8: Redes preentrenadas de YOLO

Para las pruebas utilizaremos la versión *yolov5s*, ya que es similar a la ssd_mobilenet utilizada previamente.

Para las dos versiones del modelo se realizarán dos tipos de pruebas: un entrenamiento aumentando el dataset mediante rotaciones y efecto espejo en las imágenes, y una segunda prueba donde se aplicarán filtros de saturación y brillo con el objetivo de intensificar los colores de las pegatinas, y analizar el posible efecto de usar efectos de rotación que hagan que el modelo se centre en *aprender* imágenes complejas que en adquirir las características del elemento a clasificar.

Resultados modelo 320x320, usando 4 etiquetas:

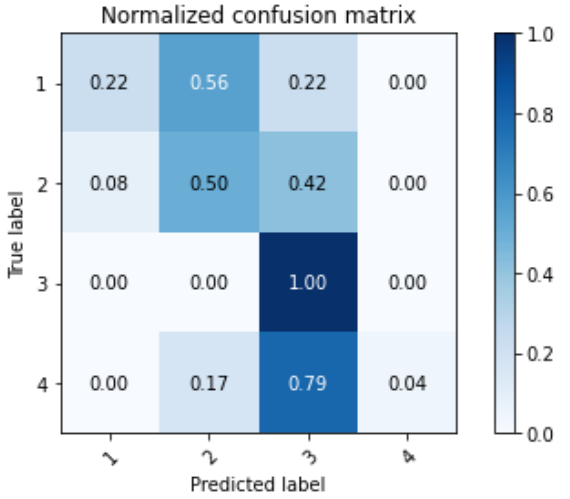


Imagen 45: Matriz de confusión tras entrenar la red con un dataset aumentado

Resultados de entrenamiento usando un dataset haciendo data augmentation de flip, rotate y mirror de las imágenes.

Media total: 32.1%
Media precisión: 57.54%
Media recall: 44.1%
Media F1: 30.9%

%	0E	B	C	ECO
Precisión	66.67	40	23.52	1
Recall	22.22	50	100	4
F1	33.33	44.44	38.1	8

Tabla 9: Métricas obtenidas de la matriz de confusión de la red entrenada con un dataset aumentado

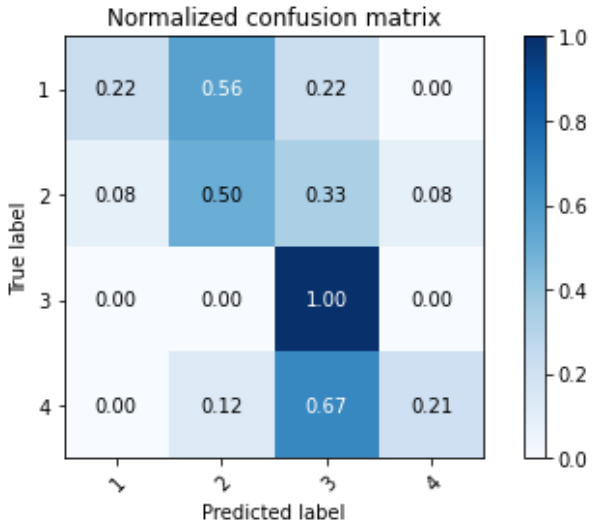


Imagen 46: Matriz de confusión tras entrenar la red con filtros de brillo y saturación

Resultados aplicando filtros de saturación y brillo a las imágenes de entrenamiento.

Media total: 39%
Media precisión: 54.88%
Media recall: 48.26%
Media F1: 38.73%

%	0E	B	C	ECO
Precisión	66.67	42.85	26.67	83.34
Recall	22.22	50	100	20.84
F1	33.33	46.15	42.1	33.34

Tabla 10: Métricas obtenidas de la matriz de confusión de la red entrenada con filtros de brillo y saturación

Como se puede observar se aprecia una leve mejora en los resultados de entrenamiento reduciendo el número de efectos de giro y transposición de las imágenes y agregando efectos de saturación y brillo, que faciliten la adquisición y detección de patrones sobre las imágenes.

Resultados modelo 320x320, usando 3 etiquetas:

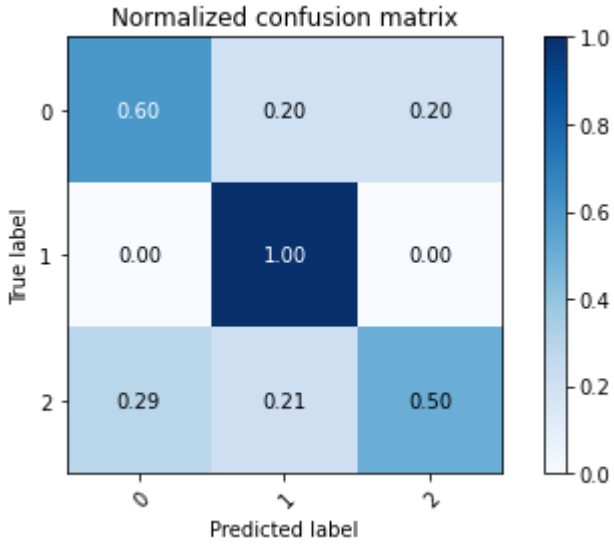


Imagen 47: Matriz de confusión del modelo entrenado con la etiqueta ECO y 0E juntas

Para el caso de utilizar 3 clases, hacemos las pruebas reduciendo el efecto de flip y mirror de las imágenes y aplicamos los filtros de imagen y saturación, tras analizar el ligero incremento en los resultados anteriores.

Media total: 65.62%
Media precisión: 66.43%
Media recall: 70 %
Media F1: 65.68%

%	B	C	ECO/0E
Precisión	60	61.53	77.78
Recall	60	100	50
F1	60	76.19	60.87

Tabla 11: Métricas obtenidas del modelo entrenado con la etiqueta ECO y 0E juntas

Tras examinar las valoraciones obtenidas se puede percibir una clara mejora en la clasificación agrupando las etiquetas *ECO* y *0E* en una única clase, y reduciendo los efectos de transposición de imágenes que dificulten el aprendizaje de la red neuronal.

8.2 Segunda propuesta: Recorte de imagen y clasificación

Como alternativa a la limitación a usar un modelo de baja resolución, se considera la opción de simular el encadenamiento de un modelo que sirva para para detectar el *cristal* frontal del vehículo y hacer un recorte de la imagen, y el resultado obtenido enviarlo al modelo entrenado para detectar los distintivos medioambientales. De esta forma la red neuronal final procesaría imágenes de menor resolución en las cuales las pegatinas se observarían con mayor tamaño facilitando la labor de entrenamiento.

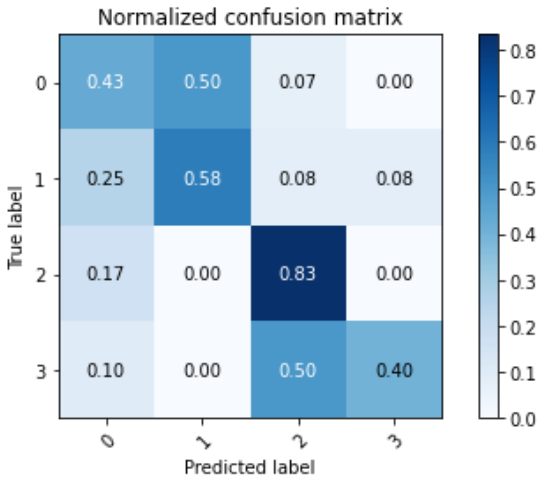
Para realizar esta simulación y analizar el comportamiento del modelo final, procedemos a recortar las imágenes del dataset, labor que haría el primer modelo.

Imagen 48: Ejemplo del recorte que realizaría el primer modelo

La imagen de la izquierda sería la que recibirá la cámara originalmente, y a su derecha tendríamos el resultado tras ser procesada por un modelo que haga el recorte. De esta forma se puede apreciar lo comentado anteriormente, donde la proporción de píxeles del distintivo frente al total de la imagen es mucho mayor tras el recorte.

Tras realizar el recorte y etiquetado del dataset, procedemos a ajustar el modelo nuevamente y obtener la matriz de confusión. Las pruebas se seguirán haciendo en YoloV5, para su posterior evaluación en el modelo final de TensorFlow que utiliza la aplicación en la cámara.

Resultados modelo 320x320:



Resultados nueva versión:
Media total: 56.25%
Media precisión: 59.70%
Media recall: 56.13%
Media F1: 55.57%

Resultados versión anterior:
Media total: 39%
Media precisión: 54.88%
Media recall: 48.26%
Media F1: 38.73%

Imagen 49: Matriz de confusión del nuevo modelo

%	0E	B	C	ECO
Precisión	50	50	58.82	80
Recall	42.85	58.33	83.33	40
F1	46.15	53.84	68.96	53.33

Tabla 12: Métricas obtenidas del nuevo modelo Yolov5

Como se observa, hay una clara mejora en rendimiento de la red neuronal al filtrar imágenes enfocadas únicamente a la parte del cristal frontal del vehículo, frente a la primera versión del modelo. Teniendo en cuenta estos resultados, procedemos al reajuste de la red neuronal ssd_mobilenet, la cual recibe una notable mejora respecto a la versión inicial.

%	0E	B	C	ECO
Precisión	1	61.53	54.54	1
Recall	57.14	66.67	1	50
F1	72.73	64	70.58	66.67

Tabla 133: Métricas obtenidas del nuevo modelo ssd_mobilenet

Precision: 79.02% *Media total*: 68.75%

Recall: 68.45%

F1-Score: 68.49%

Capítulo 9 - Conclusiones

En este apartado expondremos el veredicto en función a los datos obtenidos, en base al rendimiento global, además de una conclusión sobre los objetivos propuestos. Así como, una explicación de las limitaciones y el trabajo futuro.

9.1 Análisis global del rendimiento obtenido

Los primeros datos que vamos a analizar son los obtenidos a raíz del entrenamiento, donde se van a analizar los resultados obtenidos.

- Red neuronal de resolución 320x320

    1. El número de steps óptimo se encuentra alrededor de los 30.000 steps
    2. El porcentaje de acierto de esta red, entrenada con 30.000 steps, es del 51%.

- Red neuronal de resolución 640x640

    1. El número de steps óptimo se encuentra alrededor de los 50.000 steps
    2. El porcentaje de acierto de esta red, entrenada con 50.000 steps, es del 88,67%.

A raíz de estos datos podemos observar como el modelo entrenado de resolución 300x300 alcanza más rápidamente el pico de rendimiento con un porcentaje de acierto del 51%. Mientras que el modelo de mayor resolución logra obtener un porcentaje de aciertos del 88,67% al alcanzar los 50.000 steps. Con estos datos podemos observar como el mejor modelo es el de mayor resolución. Esto puede deberse al tamaño de los distintivos, que no es más que el puño de una persona adulta, y a su detalle, ya que, cambia el color del distintivo y la letra de su interior. Por lo cual, el nivel de detalle observable siempre va a ser más alto en una imagen cuya resolución sea mayor.

Estos datos se pueden contrastar con la matriz de confusión obtenida de cada modelo, donde se puede observar como el modelo de menor resolución tiende a confundir en mayor medida los distintivos entre ellos, mientras que el modelo de una resolución mayor genera unas predicciones más precisas.

El siguiente valor a tener en cuenta es el tiempo de inferencia, es decir, cuánto tiempo necesita nuestro modelo en obtener una predicción sobre la imagen recibida.

- Red neuronal de resolución 300x300:
    1. Tiempo de inferencia sobre Volta-100: 0.18 segundos
    2. Tiempo de inferencia sobre hardware de la cámara: 45ms/16fps

- Red neuronal de resolución 640x640:
    1. Tiempo de inferencia sobre Volta-100: 0.26 segundos
    2. Tiempo de inferencia sobre hardware de la cámara: 245ms/6fps

Examinando estos datos, podemos observar como el modelo de menor resolución es el más rápido a la hora de obtener una predicción, y qué más imágenes puede analizar por segundo. Esto se debe a que el hardware de la cámara debe grabar imágenes en la resolución deseada, o aplicar un reescalado de las mismas después de obtener dichas imágenes, lo cual conlleva un esfuerzo mayor cuanto mayor es el nivel de detalle de las imágenes. Y también, realizar el cálculo para obtener las predicciones sobre esas imágenes, el cual es más costoso cuanto mayor es el nivel de detalle.

Con los resultados obtenidos, haciendo uso de las redes neuronales de Yolov5, se estudian diferentes alternativas para mejorar el rendimiento de la red neuronal de 300x300, que parece ser la única viable en un entorno real en el que la cámara sea capaz de detectar los distintivos en tiempo real. Para ello se proponen principalmente dos ideas: reducir el número de etiquetas utilizadas agrupando las clases ECO y 0E en una, ya que ambas se pueden considerar del mismo tipo en cuanto a contaminación, y por otro lado enlazar dos modelos, uno orientado a detectar y recortar el cristal frontal del frame procesado por la cámara y enviar la imagen resultante al modelo final para que clasifique el distintivo de la DGT.

9.2 Conclusión

Basándonos en los resultados analizados en el punto anterior de las propuestas post-entrenamiento, se puede resumir que hay que realizar algún tipo de preprocesado para mejorar el rendimiento del modelo. Partiendo de tener que usar una red neuronal con una resolución no demasiado elevada, es necesario que la información que le llegue a esta sea lo más liviana posible para facilitar la clasificación de los distintivos.

Por ello, tras finalizar de evaluar las propuestas de preprocesado, se obtienen los siguientes resultados, en cuanto a acierto, en una red neuronal con una resolución de 300x300, de la arquitectura Yolov5:

- Versión original: 39%
- Usando 3 clases: 65.62%
- Recorte de imagen: 56.25%

Estos resultados extrapolados a la red neuronal *ssd_mobilenet_300* de TF1, que instancia la cámara y la cual nos ofrece un acierto del 51% en su versión inicial, recibe una notable mejora llegando al 68%.

Llegados a este punto se puede asumir que se han completado con éxito los objetivos a seguir para el desarrollo del proyecto, donde se evalúa la funcionalidad de la cámara y las distintas alternativas para obtener un modelo que funcione en caso de uso real.

Durante la investigación del proyecto, haciendo uso del conocimiento base que se ha obtenido durante el estudio del grado, se ha adquirido unas nociones más profundas sobre las redes neuronales y su aplicación a la visión artificial para la detección de objetos, la importancia que tiene el dataset utilizado y como puede afectar al entrenamiento, como se ha podido apreciar en las pruebas de simulación del recorte de imágenes, así como la integración de la inteligencia artificial en el ámbito de sistemas empotrados o *Edge devices*, tarea que guarda una fuerte relación con los estudios de Computadores.

9.3 Limitaciones y trabajo futuro

Debido a que el objetivo final del proyecto es controlar la contaminación de las calles de Madrid mediante la clasificación de los distintivos medioambientales de la DGT haciendo uso de la plataforma Inteox, existe la limitación de que este tipo de dispositivos están pensados para realizar tareas de cómputo rápido e incorporan hardware de bajo consumo. Esto hace que no podamos usar redes neuronales con resoluciones elevadas, ya que tendrían un considerable retardo de inferencia y no sería valido en una aplicación real.

Con esta premisa existe la necesidad de hacer un filtrado previo de las imágenes que envía la cámara, tratando de reducir la información a procesar por el modelo. Esto implica hacer un preprocesado previo mediante un modelo que sea capaz de detectar la luna frontal del vehículo o la pegatina en sí, hacer un recorte en la imagen del bounding box de la detección del modelo y procesar este contenido con un segundo modelo que se dedique únicamente a clasificar.

A este proceso se le conoce como clasificación en cascada donde un modelo se encarga de detectar el objeto deseado y otro de extraer las características que permitan clasificarlo. Para la clasificación de la pegatina se pueden utilizar dos alternativas: hacer uso de un modelo de clasificación de imágenes, que diferencia los distintivos por su color, símbolo y demás características; o hacer uso del OCR, mismo mecanismo que se utilizar para leer las matrículas de los vehículos, de esta forma podemos extraer los caracteres de las pegatinas (B, C, ECO, 0E) y clasificarlas por nombre. [14]

Capítulo 10 - Conclusions

In this section we will explain the final verdict based on the data obtained from the overall performance, limitations, and future work.

10.1 Global analysis of the performance obtained

The first data that we are going to analyze are those obtained as a result of the training, where the results collected in section 5.2.3.9 Result of the training and validation will be analyzed.

1. 320x320 resolution neural network

    1. The optimal number of steps is around 30,000 steps
    2. The percentage of success of this network, trained with 30,000 steps, is 51%.

1. 640x640 resolution neural network

    1. The optimal number of steps is around 50,000 steps
    2. The percentage of success of this network, trained with 50,000 steps, is 88.67%.

Because of these data, we can see how the trained model of 300x300 resolution reaches the peak of performance more quickly with a success rate of 51%. While the model of higher resolution manages to obtain a percentage of successes of 88.67% when reaching 50,000 steps. With this data we can see how the best model is the one with the highest resolution. This may be due to the size of the badges, which is nothing more than the fist of an adult person, and their detail since it changes the color of the badge and the letter of its interior. Therefore, the level of observable detail will always be higher in an image whose resolution is higher.

This data can be contrasted with the confusion matrix obtained from each model, where it can be observed how the lower resolution model tends to confuse the distinctives between them to a greater extent, while the model of a higher resolution generates more accurate predictions.

The next value to consider is the inference time, that is, how much time our model needs to obtain a prediction about the received image.

1. 300x300 resolution neural network:
    1. Inference time over Volta-100: 0.18 seconds
    2. Inference time on camera hardware: 45ms/16fps

1. 640x640 resolution neural network:
    1. Inference time over Volta-100: 0.26 seconds
    2. Inference time on camera hardware: 245ms/6fps

Examining this data, we can see how the lowest resolution model is the fastest to obtain a prediction, and what more images it can analyze per second. This is because the camera hardware must record images in the desired resolution, or rescale them after obtaining such images, which entails a greater effort the higher the level of detail of the images. Also, perform the calculation to get the predictions on those images, which is more expensive the higher the level of detail.

With the results obtained, making use of Yolov5 neural network, different alternatives are studied to improve the performance of the 300x300 neural network, which seems to be the only viable one in a real environment in which the camera can detect the badges in real time. To do this, two main ideas are proposed: reduce the number of labels used by grouping the ECO and 0E classes into one, since both can be considered of the same type in terms of contamination, and on the other hand link two models, one aimed at detecting and cropping the car front glass from the frame processed by the camera and send the resulting image to the final model to classify the DGT badge.

10.2 Conclusion

Based on the results analyzed in the previous point of the post-training proposals, it can be summarized that some type of pre-processing must be carried out to improve the performance of the model. Starting from having to use a neural network with a resolution not too high, because is necessary that the information that reaches the model it is light as possible to facilitate the classification of the badges.

Therefore, after completing the preprocessing proposals, the following results are obtained, in terms of success, in a neural network with a resolution of 300x300 from Yolov5:

1. Original version: 39%
2. Using 3 classes: 65.62%
3. Image Crop: 56.25%

These results extrapolated to the *ssd_mobilenet_300* neural network of TF1, which runs in the camera and offers an accuracy of 51% in its initial version, received a notable improvement reaching 68%.

At this point it can be assumed that the objectives to be followed for the development of the project have been successfully completed, where the functionality of the camera and the different alternatives are evaluated to obtain a model that works in case of real use.

During the research of the project, making use of the basic knowledge that has been obtained during the study of the degree, it has acquired deeper notions about neural networks and their application to artificial vision for the detection of objects, the importance of the dataset used and how it can affect training, as has been seen in the simulation tests of image trimming, as well as the integration of artificial intelligence in the field of embedded systems or *Edge devices*, a task that has a strong relationship with computer engineering studies.

10.3 Limitations and future work

Because the final objective of the project is to control the pollution of the streets of Madrid by classifying the environmental badges of the DGT making use of the Inteox platform, there is a limitation that this type of devices are designed to perform fast computing tasks and incorporate low consumption hardware. This means that we cannot use neural networks with high resolutions, since they would have a considerable inference delay and would not be valid in a real application.

With this premise there is a need to make a previous filtering of the images sent by the camera, trying to reduce the information to be processed by the model. This involves pre-processing using a model that can detect the front window of the vehicle or the badge itself, making a cut in the bounding box image of the model detection and processing this content with a second model that is dedicated solely to classifying. This process is known as cascade classification where one model is responsible for detecting the desired object and another for extracting the characteristics that allow it to be classified. For the classification of the badge, two alternatives can be used: using an image classification model, which differentiates the badges by their color, symbol and other characteristics; or make use of the OCR, the same mechanism that is used to read the license plates of the vehicles, in this way we can extract the characters from the stickers (B, C, ECO, 0E) and classify them by name. [14]

Apéndice A - Funcionamiento de los scripts de Python

En este apéndice expresamos con más detalle las scripts utilizadas en los Jupyter Notebooks[26] de entrenamiento, tanto en la API de Object Detection de Tensorflow, como, en YOLO.

Object Detection API

Las funciones de las versiones de TensorFlow TF1[27] y TF2[28] utilizadas en el proyecto se ejecutan de manera idéntica, es decir, con los mismos parámetros. Por lo cual, explicaremos las funciones como si de una sola versión se tratase.

Las funciones a detallar se encuentran en la siguiente ruta:

object_detection_api/models/research/object_detection/

- model_main.py: Script utilizada para realizar el entrenamiento de las redes neuronales. Se utilizan los siguientes parámetros:
    - model_dir: Directorio de salida donde se exportan los resultados del entrenamiento.
    - sample_1_of_n_eval_example: Número de muestras que se van a evaluar, por defecto 1.
    - pipeline_config_path: Dirección del fichero de pipeline de configuración. Este fichero esta explicado en el apartado 5.2.3.5 Configuración del entrenamiento.
    - alsologtostderr: Muestra por la salida estándar los errores durante la ejecución.

[26] https://jupyter.org/

[27] https://github.com/BOSCH-UCM/Inteox-EnvironmentalImpact/blob/tf1/ssd_mobilenet_training.ipynb

[28] https://github.com/BOSCH-UCM/Inteox-EnvironmentalImpact/blob/tf2/ssd_mobilenet_training.ipynb

- export_tflite_graph.py: Transforma el ultimo checkpoint obtenido al entrenar al formato frozen graph, el cual es necesario para obtener el modelo en formato TensorFlow Lite.
    - trained_checkpoint_dir: Dirección del checkpoint resultante del entrenamiento, misma ruta que model_dir del script anterior.
    - output_directory: Directorio de salida donde se exportará el fichero en formato frozen graph (model.pb)
    - pipeline_config_path: Dirección del fichero de pipeline de configuración.

- Para transformar el frozen graph obtenido al formato TensorFlow Lite no existe una script, por lo que hay que ejecutar las siguientes líneas de código.
    - tf.lite.TFLiteConverter.from_saved_model: ruta donde se encuentra el modelo en formato frozen graph.
    - _TFLITE_MODEL_PATH: variable que contiene la ruta en la cual se exporta el modelo ya en formato TensorFlow Lite.

```
_TFLITE_MODEL_PATH =
"training/runs/ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-
8_30k/export/model_fp16.tflite"

converter =
tf.lite.TFLiteConverter.from_saved_model('training/runs/ssd_mobilenet_v2_fpnl
ite_640x640_coco17_tpu-8_30k/export/saved_model')
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]
converter.allow_custom_ops = True

tflite_model = converter.convert()

with open(_TFLITE_MODEL_PATH, 'wb') as f:
    f.write(tflite_model)
```

YOLO

Al igual que la api de TensorFlow explicada previamente, en YOLO[29] disponemos de diferentes scripts para desarrollo de un modelo.

- train.py: Script utilizado para la fase de entrenamiento del modelo. Hace uso de los siguientes parámetros:
    - img: Resolución de las imágenes de entrenamiento y del modelo.
    - device: Dispositivo utilizado para el entrenamiento, CPU o Cuda:x (siendo x el numero de la GPU)
    - epochs: Número de épocas de entrenamiento
    - batch: Número de muestras que se transmite a la red neuronal en cada época de entrenamiento.
    - data: Ruta al fichero .yaml que contiene la configuración del dataset al ser exportado a formato YOLOv5
    - weights: Ruta de los pesos de la red a utilizar durante el entrenamiento, por ejemplo: yolov5m.pt
    - cfg: Ruta al fichero de configuración, en formato yaml, de la red neuronal
    - cache: Booleano para activar la opción de cargar las imágenes en de forma temporal en la memoria RAM durante la etapa de entrenamiento. Solo utilizar este flag en caso de tener el hardware necesario para ello.

- export.py: exporta el modelo al formato deseado, en este caso TensorFlow Lite.
    - img: Resolución del modelo y las imágenes.
    - data: Ruta al fichero .yaml que contiene la configuración del dataset al ser exportado a formato YOLOv5
    - weights: Ruta de los pesos de la red tras el entrenamiento. Los cuales se encuentran en runs/train/exp/weights/best.pt
    - include: Formato al cual se va a exportar. Soporta un gran número de formatos. Como, por ejemplo, frozen graph, TensorFlow Lite u ONNX, entre otros.
- val.py: Valida que el modelo ha sido exportado con éxito.

[29] https://github.com/BOSCH-UCM/Inteox-EnvironmentalImpact/blob/yolov5/training.ipynb

- img: Resolución de las imágenes y el modelo.
- weights: Ruta de los pesos de la red tras ser exportada. Se localiza en la siguiente ruta: runs/train/exp/weights/best-fp16.tflite

Bibliografía

[1] «Dirección General de Tráfico,» [En línea]. Available: https://sede.dgt.gob.es/es/vehiculos/distintivo-ambiental/.

[2] C. Smith, B. McGuire, T. Huang y G. Yang, «University of Washington,» [En línea]. Available: https://courses.cs.washington.edu/courses/csep590/06au/projects/history-ai.pdf.

[3] «IBM,» IBM Cloud Education, [En línea]. Available: https://www.ibm.com/cloud/learn/neural-networks.

[4] «SAS,» [En línea]. Available: https://www.sas.com/en_us/insights/analytics/neural-networks.html.

[5] «tutorials point,» [En línea]. Available: https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_basic_concepts.htm#.

[6] D. E. Rumelhart, G. E. Hinton y R. J. Williams, «Learning representations by back-propagating errors,» [En línea]. Available: https://www.iro.umontreal.ca/~pift6266/A06/refs/backprop_old.pdf.

[7] D. Calvo, «Diego Calvo,» [En línea]. Available: https://www.diegocalvo.es/red-neuronal-recurrente/.

[8] J. I. Bagnato, «JuanBarrios,» [En línea]. Available: https://www.juanbarrios.com/redes-neurales-convolucionales/.

[9] «MDPI,» [En línea]. Available: https://www.mdpi.com/1424-8220/20/7/2021/htm.

[10] B. S. Systems, «BOSCH,» [En línea]. Available: https://resources-boschsecurity-cdn.azureedge.net/public/documents/AUTODOME_inteox_7000_Data_sheet_esES_77603614859.pdf.

[11] M. Sighal, «Medium,» [En línea]. Available: https://medium.com/@techmayank2000/object-detection-using-ssd-mobilenetv2-using-tensorflow-api-can-detect-any-single-class-from-31a31bbd0691.

[12] S. K., «Towards data science,» [En línea]. Available: https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c.

[13] D. Mack, «FreeCodeCamp,» [En línea]. Available: https://www.freecodecamp.org/news/how-to-pick-the-best-learning-rate-for-your-machine-learning-project-9c28865039a8/.

[14] A. Rosenbrock, «pyimagesearch,» [En línea]. Available: https://pyimagesearch.com/2020/09/21/opencv-automatic-license-number-plate-recognition-anpr-with-python/.

Repositorios del proyecto

Código del proyecto

https://github.com/BOSCH-UCM/Inteox-EnvironmentalImpact

Dataset

https://universe.roboflow.com/luisan06-ucm-es/pegatinas-dgt

https://universe.roboflow.com/luisan06-ucm-es/pegatinas-dgt-3tags

https://universe.roboflow.com/luisan06-ucm-es/pegatinas-dgt-edit

https://drive.google.com/drive/folders/1dznCIr5Rqkke9F3Et6CgS6dZUiEtiqsX?usp=sharing