Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

# Acelerando la detección de códigos de barras mediante poda de redes neuronales

*Accelerating barcode detection through neural network pruning*

Mario Hernández García

La Laguna a 11 de Julio de 2024

D. **José Gil Marichal Hernández**, con N.I.F. 78.677.406-H, profesor titular adscrito al área Teoría de la Señal y Comunicaciones, del Departamento de Ingeniería Industrial de la Universidad de La Laguna, como tutor

**C E R T I F I C A**

Que la presente memoria titulada:

”*Acelerando la detección de códigos de barras mediante poda de redes neuronales*”

ha sido realizada bajo mi dirección por D. **Mario Hernández García**, con N.I.F. 43.841.312-T.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 11 de Julio de 2024.

# Agradecimientos

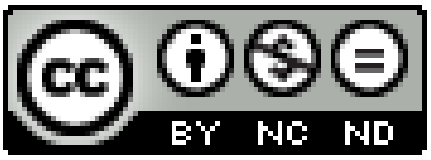
Gracias a la Universidad de La Laguna por la formación recibida y los recursos disponibles para los estudiantes durante su etapa académica.

Gracias a mi tutor, José Gil Marichal Hernández por mostrarme el mundo de la visión por computador.

Gracias a los profesores que han puesto todo su esfuerzo para ir mas allá y traer la teoría a la experiencia.

Gracias a mis compañeros, en especial a Jonay, por acompañarme desde el comienzo y haberme demostrado la verdadera perseverancia frente a las adversidades.

# Licencia



**Resumen**

*La codificación de la información usando códigos de barras es uno de los métodos mas adoptados para la identificación y seguimiento de procesos y productos etiquetados, con el énfasis en la rapidez y sencillez de su decodificación. Uno de los principales problemas a la hora de automatizar este sistema es la necesidad de un operario que localice, situé y oriente correctamente el lector sobre el código de barras. Este proceso aunque resulta sencillo para un operador humano, no lo es tanto para una máquina. Una de las técnicas actuales para solucionar este tipo de problemas es el uso de las redes neuronales convolucionales, CNN, especializadas en la detección, clasificación y segmentación de objetos en una imagen. El uso de redes neuronales convolucionales con resultados en tiempo real requiere requisitos computacionales que, en muchas situaciones, no permiten su despliegue en sistemas donde los recursos son limitados, como puede ser una plataforma móvil o dispositivos IoT. Para aliviar esta necesidad, en esta memoria se estudiarán y detallarán los procedimientos para entrenar y optimizar un modelo state-of-the-art, como YoloV5 de forma que se pueda implementar en una aplicación Android que consiga detectar y subrayar los códigos de barras visibles a través de la cámara del dispositivo móvil.*

***Palabras clave:*** *códigos de barras, aprendizaje automático, imágenes, redes neuronales convolucionales, aplicación móvil, modelos, entrenamiento*

**Abstract**

*The encoding of the information using barcodes is one of the most adopted methods of identification and tracking of processes and labelled products. With emphasis on the speed and simplicity of its decoding, one the major issues to automate this system is the need of requiring an operator to identify, place and orient the scanner properly on the barcode. This process although it's easy to do by an human, it's not so for a computer. One of the current techniques for approaching this problem is the use of a convolutional neuronal network, CNN, specially designed for the detection, classification and segmentation of an item on an image. The use of convolutional neural networks with results in real time presents computational requirements that, in many situations, doesn't allow its deployment in systems where the resources are constrained like a mobile platform or IoT devices. To ease this need, this report will study and detail the procedures for training and optimizing a state-of-the-art model like YoloV5 so that it can be implemented in an Android application that can detect and highlight the barcodes visible through the mobile device camera.*

**Keywords:** barcodes, machine learning, images, convolutional neuronal networks, mobile application, models, training

# Índice general

# Índice de Figuras

# Índice de Tablas

# Capítulo 1

# Introducción

## 1.1. Códigos de barras

Los códigos de barras son uno de los medios mas utilizados para la codificación de información relevante a un producto o proceso, asignándole una etiqueta inteligible por una máquina. Existen tres categorías según los elementos que se usen para codificar los caracteres alfanuméricos:

- Lineales (1D):
  - Su codificación consiste en el uso de lineas paralelas horizontales de distinto grosor y/o separación, generando patrones que identifican los caracteres alfanuméricos. Dentro de esta categoría existen una amplia cantidad de estándares pero no presentan grandes variaciones en su implementación [1].


(a) Código de barras UPC-A.


(b) Código de barras binario farmacéutico.

Figura 1.1: Códigos de barras 1D

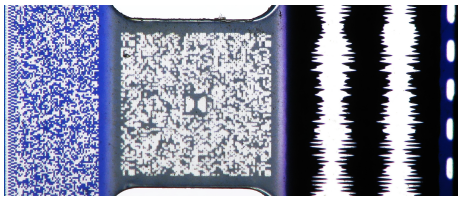
- Bidimensionales (2D):
  - Se siguen empleando las lineas pero se empieza a considerar la verticalidad del código. Dentro de esta categoría existen una amplia cantidad de estándares pero no presentan grandes variaciones en su implementación.

(a) Código de barras de un rollo de película DX.

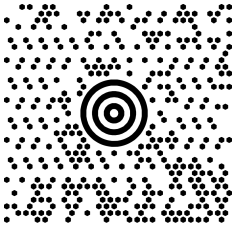
(b) Códigos de barras postal japonés.

Figura 1.2: Códigos de barras bidimensionales (2D).

- Matriciales (2D):
  - Se introducen una gran variedad de elementos para formar nuevos patrones que aumentan enormemente la capacidad de almacenamiento pero también lo hace su complejidad.



(a) Código 2D del audio de un rollo de película Dolby.



(b) Código 2D de reparto postal UPS.

Figura 1.3: Códigos de barras matriciales (2D).

Para recuperar la información de cualquiera de estos códigos de barras, es necesario utilizar un escáner que, independientemente del tipo de código de barras, requiere de un operador que identifique qué elemento es un código de barras y, una vez identificado, alinee y oriente correctamente el escáner sobre el código. Este proceso se convierte en trivial una vez el operador conoce los tipos de códigos de barras que existen y cómo operar el escáner. Sin embargo, al intentar automatizar este procedimiento, nos encontramos con que no se puede trasladar fácilmente a un ordenador.

## 1.2. Antecedentes

### 1.2.1. Orígenes, usos y obsolescencia de los códigos 1D

El código de barras fue inventado por Norman Joseph Woodland y Bernard Silver en 1949 [2], quienes patentaron un diseño de código de barras circular con líneas concéntricas de diferente grosor.

Los lectores de códigos de barras portátiles, basados en barrido láser, se introdujeron en los comercios en 1974, cuando se leyó el primer artículo con un código universal de producto (UPC) impreso en papel adhesivo pegado manualmente junto a las etiquetas de precio. Esto no solo permitía agilizar la lectura de artículos en cajas registradoras, sino sobre todo evitaba errores de digitación.[3]

Los lectores evolucionaron ganando portabilidad, resistencia y durabilidad, sumando aplicaciones de computación móvil y comunicaciones inalámbricas. La pervivencia por 5 décadas de códigos 1D, aparentemente rudimentarios, en parte es fruto de la posibilidad de realizar toda la lógica de negocio desde dispositivos móviles dotados de cámaras, accediendo a bases de datos que se consultan con el identificador impreso en las etiquetas. Para estos usos, los códigos de barra 2D son redundantes. La ventaja de los códigos 2D, permitiendo mayor cantidad de datos, en un mundo ultra conectado incluso juega en su contra, dado que no permiten estar actualizados continuamente, por lo que si solo se

emplean para recabar un código identificador de producto, no mejoran a los códigos 1D para esa tarea en la que resultan más económicos. A tenor de ello, los códigos 1D siguen siendo imbatibles en algunos sectores, como las compañías de correo, pero también se resisten a desaparecer de los lineales de comercios y supermercados.

Otra técnica que debía desbancarlos, los códigos de identificación por radio frecuencia (RFID), no terminan de asentarse por el costo unitario de las etiquetas, y sobre todo el coste de los lectores [4]. El mayor beneficio de los códigos RFID es de hecho, que al basarse en emisión de radiación electromagnética que se dispersa en todas direcciones, no precisa de una etapa de localización mediante operadores humanos que puede consumir mucho tiempo para ciertas tareas como la verificación de inventario.

Es por ello que en este TFG nos centraremos en el estudio de códigos de barras 1D.

Y concretamente, en estudiar técnicas para analizar imágenes tomadas con cámaras de móvil, y, computando en el propio móvil, determinar la presencia y localización precisa de los códigos de barras, como antesala de la decodificación. La decodificación, en la que no entraremos, es, de hecho, una tarea más sencilla que la localización de los códigos, dado que se diseñó deliberadamente para ser realizada por máquinas.

### 1.2.2. Localización automática de códigos de barras

Existen varias vertientes que intentar afrontar el problema: técnicas de visión por computador clásicas, la estandarización y el *deep learning*:

- **Técnicas de visión por computador clásicas:** Utilizando la transformada discreta de Radon[5], filtros *Sobel* [6], transformación *Top-hat* [7], análisis de texturas [8].
- **Estandarización:** Limitado a sistemas donde es posible establecer un estándar que asegure que el etiquetado, escaneo y procesamiento de los códigos de barras se realiza uniformemente.

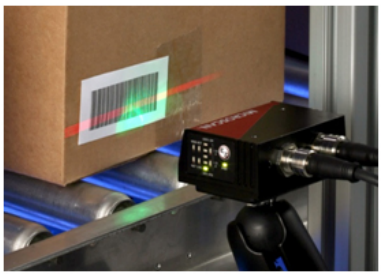


Figura 1.4: Estandarización en una cadena de paquetería.

- **Deep learning:** Mediante el uso de redes neuronales convolucionales que están especializadas, entre otras funciones, a aprender y reconocer patrones complejos en una imagen. Este método requiere de una gran cantidad de recursos iniciales para poder realizar el entrenamiento y, obtener un modelo que responda bien a las variaciones de las condiciones de las imágenes e identifique los códigos de barras correctamente. Este es el enfoque que se ha elegido estudiar en este TFG.

## 1.3. Deep learning y redes neuronales convolucionales

El *deep learning* es un tipo de proceso de *machine learning* de las ramas principales de la inteligencia artificial que se centra en establecer algoritmos que permitan que las maquinas aprendan y mejoren a través de la experiencia. Para ello, se utilizan grandes conjuntos de datos que son analizados en busca de patrones que consigan dar con la solución. En nuestro caso, se produce la extracción de características de una imagen, que en su conjunto, responden a una clasificación concreta. Nuestro sistema emplea solo dos clasificaciones: el código de barras y el fondo, que representa todo lo demás.

Dentro del *deep learning*, existen una amplia gama de tipos de redes neuronales, muchas de ellas especializadas en extraer características concretas de los datos y otras mas generalistas. En las tareas de visión por computador destacan las *CNNs* por su capacidad de procesar datos estructurados en una cuadricula como una imagen. Las tareas mas destacadas son la del reconocimiento de objetos, clasificación de imágenes, segmentación de imágenes, estimación de pose y, más recientemente, la detección de objetos orientados.


Figura 1.5: Ejemplos de aplicación de las CNNs.

Las *CNNs* están compuestas por múltiples capas que procesan paso a paso la estructura de datos de entrada. Las capas principales y las intermedias extraen características realizando convoluciones y agrupaciones, mientras que la capa final, denominada totalmente conectada, toma como entrada los resultados de las capas anteriores y, según el peso de cada característica, genera una clasificación probabilística final. Las implementaciones de estas *CNNs* se denominan modelos.

- Convoluciones:
  - Consiste en aplicar un kernel (filtro) a lo largo de una estructura de datos en cuadricula. Las características extraídas dependerán del kernel que apliquemos.

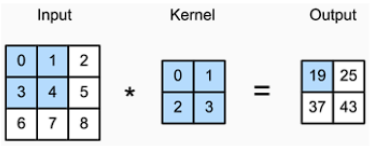



Figura 1.6: Aplicando el filtro

- Agrupaciones:
  - La capa realiza una simplificación de la dimensión de los datos de entrada, normalmente de los resultados de una capa de convolución, que tiene como efectos secundarios reducir la variación y la retención de las características mas relevantes. Las agregaciones que se suelen aplicar son las de máximo, mínimo o media.

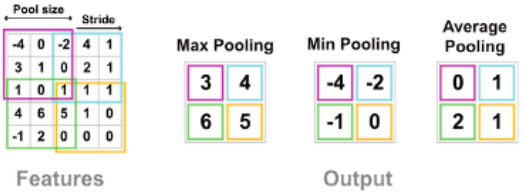



Figura 1.7: Aplicando la agrupación

- Capa totalmente conectada:
  - Es la capa que presenta la red neuronal donde se reciben los datos de las anteriores capas y se trasmiten a las neuronas que, tras calcular el resultado según los pesos de la entrada, generan un valor al que se le aplica una función de activación que evita la linealidad que puede generar el modelo y que se puede traducir a una probabilidad de clasificación de un objeto.

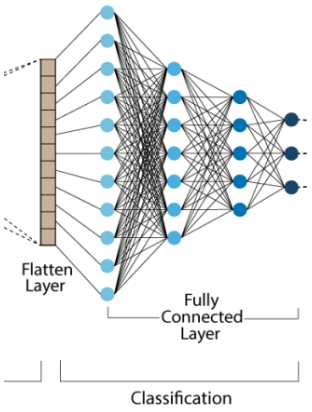



Figura 1.8: Capa totalmente conectada

Es vital comprender el funcionamiento de las *CNNs* para entender los resultados obtenidos, comprender las capas resultantes del entrenamiento y valorar que técnicas y o modificaciones aplicar para mejorar su rendimiento y su adaptabilidad.

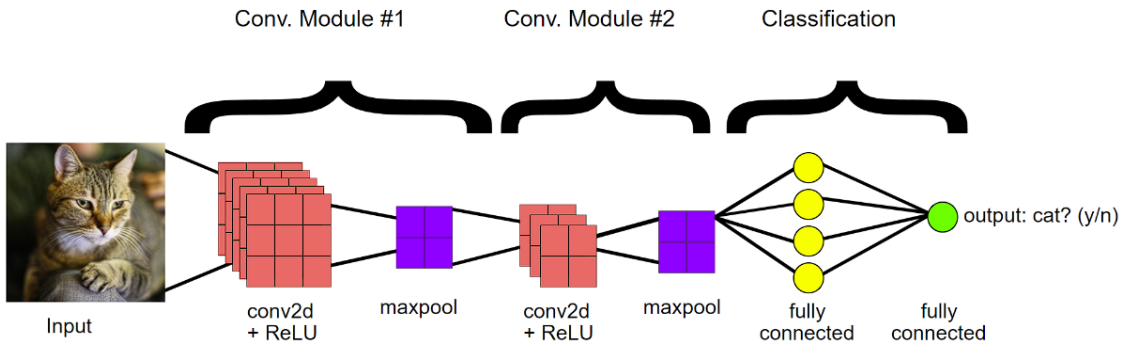



Figura 1.9: CNN completa

## 1.4. Retos de la aplicación de *CNNs* en móviles

Los principales problemas que se presentan cuando intentamos afrontar el problema de la detección de códigos de barras mediante el uso de *CNNs* son: la capacidad computacional que requieren la ejecución de los modelos, la escasa y difícil implementación en otras plataformas que no sean de escritorio y la poca flexibilidad en sistemas con recursos limitados, ya sean energéticos como de almacenamiento [9].

En un TFG previo [10], donde se entrenó una red ResNet50, el rendimiento obtenido cuando se ejecuta el modelo en un dispositivo móvil (>120ms) no permite su aplicación en tiempo real (<33ms). Otro de los problemas encontrados era el de convertir el modelo para su ejecución en un dispositivo Android ya que los conjuntos de herramientas disponibles en Android son diferentes y no es posible ejecutarlos directamente, provocando que se añadan capas de compatibilidad al modelo y por lo tanto aumentando su complejidad. Además, aunque no exclusivo al caso de uso o a las *CNNs*, también resulta problemático obtener un *dataset* que presente la mayor cantidad de situaciones en las que se pueda encontrar los objetos o elementos a identificar.

Por ello, en este TFG se va a tratar de dar solución a estos problemas partiendo de un modelo ya entrenado como es *You Only Look Once* (YOLO)[11], modelo por excelencia para visión artificial, entrenado con el dataset COCO128 que presenta una amplia cantidad de objetos. Aprovecharemos el conocimiento adquirido de analizar todos estos objetos para realizar una transferencia de conocimiento haciendo que pase a identificar únicamente códigos de barras y el fondo. Para este procedimiento se hará uso de un repositorio público de códigos de barras de uso libre, distribuidos en distintos artículos pero recopilados en [12].Durante el proceso de transferencia de conocimiento se ajustaran los súper parámetros de entrenamiento para reducir la linealidad del modelo y el sobre-ajuste. Se aplicaran técnicas de podado y de cuantización para simplificar el modelo y mejorar su rendimiento.

## 1.5. Estado del arte

Tras analizar la situación actual y compararla con las conclusiones obtenidas en el TFG previo [10], se concluye que el estado del arte no ha variado significativamente. Todas las aplicaciones tratadas a continuación han sido ejecutadas en un dispositivo móvil “One plus 11” que implementa un Snapdragon Gen 2, procesador ARM de ultima generación en 2023 [13].

Google dispone del *machine learning kit* [14] en el que, entre otras funcionalidades provee un detector de códigos de barras, el cual se ejecuta correctamente pero, habiéndolo configurado con un tamaño de imagen de 640x360, no es capaz de alcanzar el rendimiento ni la precisión necesarios para utilizarlo de forma efectiva en tiempo real.

Podemos encontrar otras librerías para Android pero éstas hacen uso internamente del kit ML de Google [15], o de su antecesor, implementado sin IA, la librería ZXing [16]. Multitud de librerías focalizadas en la localización, como es nuestro caso, una vez tienen la línea de barrido que cruza el código recurren a ZXing para la tarea de decodificación del mismo.

La empresa Scandit oferta su *Software Development Kit* (SDK) con una amplia cantidad de opciones y funcionalidades adicionales destinadas a la gestión de paquetería y inventario pero cuando ejecutamos su demo [17], nos encontramos con un rendimiento similar al del kit ML de Google y en algunas ocasiones incluso peor, cuando trata de maquillar, por estimación de desplazamiento, la posición de un código de barras ya identificado, pero que deja de serlo debido al desenfoque de movimiento. El uso del SDK requiere de una licencia comercial.

De forma similar, la empresa Dynamsoft ofrece un SDK de pago en el que, tras ejecutar su demo [18], se obtienen los mismos resultados que con el resto de librerías, no cumple con los resultados esperados. Las características de su servicio son similares a las que ofrece Scandit.

Por el lado de la investigación, podemos encontrar una gran cantidad de estudios con el mismo objetivo pero la mayoría se quedan en una implementación de escritorio o servidor. [19]

## 1.6. Motivaciones personales

Los principales motivos para la realización de este proyecto van desde lo personal hasta lo profesional. Al autor de este TFG siempre la ha interesado la automatización de procesos y las posibilidades que ofrece la visión artificial, desde el escalado de imágenes, reconocimiento de objetos hasta mas recientemente, la generación de imágenes artificiales. La inteligencia artificial es un campo en el que el autor no dispone de experiencia y del cual considera que es vital conocer las capacidades y los límites actuales, sobretodo con el rápido avance de los últimos años.

# Capítulo 2

# Elección del dataset

Existen varios tipos de aprendizaje que se pueden llevar a cabo sobre las *CNNs*. Los requerimientos definirán cual método es necesario utilizar. El *dataset* necesitará cumplir distintos requisitos según el aprendizaje escogido:

- **Aprendizaje no supervisado:** Se emplean conjuntos de datos sin etiquetar con el fin de encontrar patrones o estructuras desconocidos en los datos. Requieren de conjuntos de datos mucho mas grandes que los supervisados.
- **Aprendizaje por refuerzo:** Se apoya en la prueba y error dentro de un entorno donde los resultados positivos se recompensan y el objetivo es maximizar la recompensa. No es necesario disponer de un *dataset* inicial si bien, será necesario definir el entorno de forma precisa.
- **Aprendizaje supervisado:** Este aprendizaje hace uso de conjuntos de datos etiquetados de forma que, para cada entrada (dato) se conoce la solución exacta (etiqueta). Con esto se busca obtener la "función" que determina la respuesta de cualquier entrada. Este es el aprendizaje mas comúnmente empleado con las *CNNs* y es el que se va a emplear en este TFG.

En caso de necesitar un *dataset* etiquetado, también se debe comprobar que las etiquetas sigan el formato esperado por la herramienta de aprendizaje que vayamos a usar, ya que en caso de que no sea válido, se deberán transformar los datos. En el caso de YOLO, se espera que el etiquetado siga el formato:

(clase | centro_x | centro_y | ancho | alto)


Figura 2.1: Formato YOLO etiquetado del *dataset*.

## 2.1. Datos de entrenamiento

Un enfoque habitual consiste en la creación de un *dataset* utilizando imágenes tomadas o extraídas de internet, siempre teniendo en cuenta los derechos de autor. Además de obtener las imágenes, también es necesario etiquetar cada una, lo cual es un trabajo laborioso. Sin embargo, existen herramientas de etiquetado [20] y la posibilidad de automatizar este proceso. Estas últimas suelen ser soluciones a medida, como la elaborada en el TFG previo [10] por lo que puede que no siempre se puedan emplear por terceros.

Por otro lado, también existen plataformas donde se puede encontrar una gran cantidad de *datasets* elaborados por usuarios o empresas. Muchos de estos conjuntos de datos son recopilaciones de otros públicos que ya existían previamente, como Artelab y Muenster. El principal problema al intentar usar estos recursos es la dudosa calidad del etiquetado realizado, lo que puede afectar al modelo resultante. En caso de optar por esta opción, se recomienda realizar una supervisión manual y excluir los datos incorrectamente etiquetados.

También existe la posibilidad de crear *datasets* utilizando imágenes generadas artificialmente; no obstante, estos datos solo deberían emplearse como complemento de *datasets* reales. De lo contrario, el modelo podría no responder bien en situaciones del mundo real y presentar un rendimiento deficiente en condiciones variadas y no controladas.

Para evitar esta revisión manual, se ha decidido utilizar los *datasets* públicos probados en otro estudio [21], como Artelab, Muenster, ParcelBar, InventBar, y 1D Barcode Extended, también recopilados en [12]. Cada uno de ellos presenta características diferentes:

| Nombre | Resoluciones | Nº Imágenes | Dispositivos |
|---|---|---|---|
| Arte-Lab | 640x480, 1152x864, 1600x1200, 2976x2232 | 795 | Nokia 5800 Xpres, CEC DV Camera, Nokia 7610 |
| Muenster | 640x480, 800x600, 1024x768, 1600x1200 | 1055 | Nokia N95 |
| ParcelBar | 640x480, 800x600, 1024x768, 1600x1200 | 844 | ? |
| InventBar | 4032x3024 | 527 | Samsung Galaxy S10 Plus |
| 1D Barcode Extended | 648x488 | 155 | ? |

Dependiendo de la herramienta de aprendizaje, puede que la discordancia en el tamaño de las imágenes impida realizar el entrenamiento. En este caso, será necesario modificar la carga de las imágenes para añadirles relleno y rectificar las etiquetas acorde a un nuevo tamaño común válido.

## 2.2. Preparación del *dataset*

Una vez formado el *dataset*, las herramientas de entrenamiento normalmente requieren que el conjunto de datos se distribuya para cumplir tres necesidades: entrenamiento, validación y testeo, siendo esta última opcional. Para YoloV5, se espera que el *dataset* siga la siguiente estructura:

Para realizar la distribución de los datos, habitualmente se siguen las proporciones de 80 % para entrenamiento y el 20 % restante para validación y testeo. En este trabajo se ha escogido una proporción de 70 %-30 %, con 2363 imágenes para entrenamiento y 1013 para validación y testeo, distribuyendo esta ultima equitativamente.

Además, junto al *dataset*, es necesario crear un fichero de configuración que detalle la ruta al *dataset*, las rutas a las carpetas de `testing`, `training` y `validation`, la lista de clases con su identificador único indicado en el etiquetado de las imágenes en el que, el primer elemento corresponde al numero 0 y el siguiente al 1, consecutivamente. Por último, se debe indicar el número de clases etiquetadas, indicado por la variable *nc*.

```
1 path: ../datasets/barcodes
2 train: training/images
3 val: validation/images
4 test: testing/images
5
6 nc: 1
7 names: ['barcode']
```

Código Fuente 2.1: Fichero de configuración del *dataset* `barcodes.yaml`.

# Capítulo 3

# Entrenamiento del modelo

## 3.1. Elección del modelo

La correcta elección del modelo es crucial para obtener un buen rendimiento en dispositivos con recursos limitados. Un modelo complejo extrae un mayor numero de características que pueden lograr una mejor detección de los elementos. Sin embargo, su ejecución va a requerir una mayor capacidad de procesamiento, memoria, y espacio por lo que, dependiendo de los requisitos y las limitaciones, puede que sea mas adecuado sacrificar capacidad de detección por obtener un mejor rendimiento y reducir el consumo de recursos.

En el caso de YoloV5, los modelos disponibles presentan las siguientes características:

Cuadro 3.1: Comparación principales modelos YoloV5 sobre el dataset *COCO val20217*

| Modelo | Tamaño (px) | $mAP_{val}$ 50-95 | $mAP_{val}$ 50 | Vel CPU (ms) | Parámetros (M) |
|---|---|---|---|---|---|
| YOLOv5n | 640 | 28.0 | 45.7 | 45 | 4.5 |
| YOLOv5s | 640 | 37.4 | 56.8 | 98 | 16.5 |
| YOLOv5m | 640 | 45.4 | 64.1 | 224 | 49.0 |
| YOLOv5l | 640 | 49.0 | 67.3 | 430 | 109.1 |
| YOLOv5x | 640 | 50.7 | 68.9 | 766 | 205.7 |

[1] Los modelos se han ejecutado en una instancia AWS p3.2xlarge[22] sobre CPU (Intel Xeon E5-2686 v4).
[2] Los tiempos de inferencia son en ejecución dato a dato (no ejecución en lote).
[3] Se han realizado 300 *epochs* (iteraciones) sobre el dataset de entrenamiento.

Teniendo en cuenta las especificaciones y el equipo donde se han ejecutado los modelos, podemos descartar automáticamente los modelos *YOLOv5m* (mediano) *YOLOv5l* (grande) y *YOLOv5x* (extra grande) ya que, incluyendo las posibles mejoras de rendimiento tras simplificar el modelo, sera poco probable lograr el objetivo de <33ms de inferencia. Por ello, se va a realizar el entrenamiento en los modelos *YOLOv5s* (pequeño) *YOLOv5n* (nano).

## 3.2. Preparación

Podemos obtener los modelos YoloV5 y las herramientas para trabajarlos a través del repositorio de Github de Ultralytics [23], empresa detrás de las versiones iniciales, así como las más recientes, de YOLO. Sin embargo, para este proyecto, se ha decidido utilizar un repositorio [24] que implementa el motor de inferencia *DeepSparse* de Neuralmagic [25] sobre CPU, el cual, aunque no es posible utilizarlo en dispositivos móviles, optimiza el modelo y su ejecución sobre CPU, obteniendo mejoras significativas que pueden poner en contraste la capacidad del modelo cuando se ejecuta en un entorno especializado.

### 3.2.1. Hiper parámetros

Los hiper parámetros son variables de configuración externas del algoritmo de entrenamiento. Son establecidos previamente al realizar el entrenamiento y definen cual va a ser el comportamiento de las herramientas que intervienen en el aprendizaje. Entre ellas, los parámetros del algoritmo de optimización, configuración del aprendizaje y funciones de perdida y por último, las variables de aumento de datos.

Figura 3.1: Influencia de los hiper parámetros de aumento de datos durante el entrenamiento.

En la figura 3.1 se muestran dos casos de aumento de datos, extraídos de un lote de pruebas resultante del entrenamiento. Se observa alteración de la saturación, traslación, reducción de escala, agrupación en mosaicos y volteo de la imagen, todo estos parámetros establecidos como una probabilidad en el fichero de configuración. En la figura 3.1 se ha utilizado el fichero `hyp.scratch-low.yaml`

Normalmente se emplea una configuración de partida procedente de otros entrenamientos sobre *datasets* y en modelos de tamaños similares, aunque también existe la opción de realizar un proceso denominado *evolución de hiper parámetros*. En este proceso, se utiliza un algoritmo de optimización heurística como es el algoritmo genético con el que definimos el *fitness* (aptitud), que este caso son las métricas de evaluación del modelo: *precisión*, *recall*, *AP*, *mAP* que veremos en el apartado 3.4. A estas métricas se le asignara un peso de 1 a repartir, que sera el objetivo a optimizar por el algoritmo. El procedimiento necesitara el *dataset*, el modelo y el numero de iteraciones a realizar,

donde en cada una de estas iteraciones se ejecutara el algoritmo genético múltiples veces, normalmente 300 generaciones por lo que se trata de un procedimiento lento y costoso.

## 3.3. Entrenamiento

Habiendo preparado el entorno de Python y cargado el *dataset* con su fichero de configuración, podemos proceder al entrenamiento.

Debemos tener en cuenta múltiples parámetros a indicar a la hora de comenzar el entrenamiento. En el caso de entrenar el modelo para ejecutarlo bajo *DeepSparse* será necesario indicarle a la herramienta que use una de las recetas que provee Neuralmagic en su repositorio. Esta receta contiene la definición de los parámetros de cuantización, poda, hiper-parámetros y configuración de las capas para poder ejecutarse con su motor de inferencia, estas operaciones las trataremos más adelante. Un ejemplo de la ejecución del entrenamiento para *DeepSparse*:

```
python train.py --cfg ./models_v5.0/yolov5n.yaml \
  --data data/barcodes.yaml \
  --recipe ../recipes/yolov5.transfer_learn_pruned_quantized.md \
  --hyp data/hyps/hyp.scratch.yaml \
  --weights yolov5n.pt \
  --img 640 \
  --batch-size 16 \
  --optimizer SGD \
  --single-cls \
  --epochs 240
```

Código Fuente 3.1: Comando para comenzar el entrenamiento de un modelo YoloV5n.

- **cfg**: Especificación del modelo YoloV5.
- **data**: Configuración del dataset.
- **recipe**: Receta de Neuralmagic con la especificación de la cuantización, poda, hiper-parámetros y algunos parámetros del entrenamiento.
- **hyp**: Configuración de los hiper-parámetros, sustituidos por la receta.
- **weights**: Modelo en el que, en este caso, se va a realizar el aprendizaje transferido.
- **img**: Tamaño de entrada de imagen que va a recibir el modelo.
- **batch-size**: Número de muestras a pasar antes de actualizar los pesos del modelo, dependerá de la capacidad de la GPU.
- **optimizer**: Algoritmo de optimización a usar, por defecto descenso de gradiente estocástico (SGD).
- **single-cls**: Indica que el modelo solo va a constar de una clase, el código de barras.

- **epochs**: Número de iteraciones del entrenamiento sobre el dataset.

Una ejecución satisfactoria generará una suite de imágenes y datos estadísticos sobre los resultados obtenidos y dos modelos finales, uno con la iteración con mejores resultados `best.pt` y otro con el resultado de la última iteración `last.pt`.

## 3.4. Evaluación del modelo resultante

Tenemos que tener en cuenta las principales métricas que nos permiten evaluar los modelos que procesan imágenes:

- **precisión**: Mide la proporción de verdaderos positivos entre todos los ejemplos predichos como positivos.

$$\text{precisión} = \frac{\text{verdaderos positivos}}{\text{verdaderos positivos} + \text{falsos positivos}}$$

- **recall**: El recall mide la proporción de verdaderos positivos entre todos los ejemplos que son realmente positivos.

$$\text{recall} = \frac{\text{verdaderos positivos}}{\text{verdaderos positivos} + \text{falsos negativos}}$$

- **AP**: Área de la curva P-R, un valor alto indica la detección y clasificación correcta de los casos.

$$\text{AP} = \sum_{n} (R_n - R_{n-1}) P_n$$

- **mAP**: Media del AP, se presenta teniendo en cuenta las precisiones de todas las clases del modelo.

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^{N} \text{AP}_i$$

- **box_loss**: Mide la diferencia entre los cuadros delimitadores inferidos y los cuadros delimitadores reales.
- **obj_loss**: Mide la probabilidad de que el elemento exista en el cuadro delimitador inferido.
- **cls_loss**: Mide la diferencia entre la clase inferida y la real.

Resultados tras realizar *transfer learning* con 240 *epochs*:

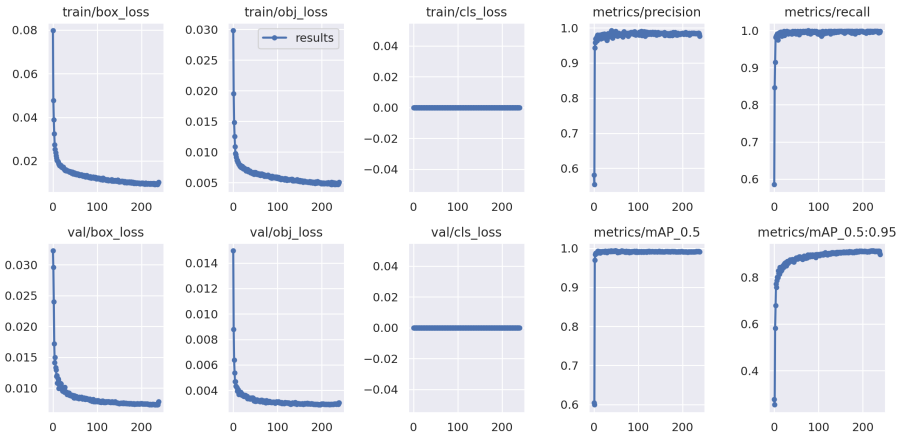



Figura 3.2: Métricas YoloV5n.

Podemos observar en la figura 3.2 como, a partir de las 110-120 iteraciones, las métricas de *precision*, *recall* y *mAP* comienzan a estabilizarse mientras que *box_loss* y *obj_loss* continúan mejorando, sin embargo en las ultimas iteraciones se observa un empeoramiento en las métricas de la validación, se esta produciendo el efecto de sobre ajuste. Aparece cuando el modelo comienza a aprender el ruido de las imágenes de entrenamiento, haciendo que tenga un rendimiento excelente cuando procesa imágenes del *dataset* pero sea peor en situaciones nuevas, al producirse una perdida de generalización. En estas situaciones es necesario aumentar los datos de entrenamiento o parar el entrenamiento cuando se detecta el empeoramiento de las métricas de validación.

Tras repetir el entrenamiento sin usar la receta, es decir, sin cuantizar, podar y la configuración especifica de hiper-parámetros los modelos presentan las siguientes características:

Cuadro 3.2: Comparación características modelos entrenados YoloV5n y YoloV5s

| Modelo | Tamaño (px) | $mAP_{val}$ 50-95 | $mAP_{val}$ 50 | Vel CPU (ms) | Vel GPU (ms) |
|---|---|---|---|---|---|
| YOLOv5n | 320 | 0.992 | 0.868 | 14.22ms | 5.84ms |
| YOLOv5s | 320 | 0.993 | 0.876 | 25.89ms | 6.91ms |
| YOLOv5n | 640 | 0.993 | 0.890 | 30.11ms | 6.07ms |
| YOLOv5s | 640 | 0.989 | 0.894 | 78.98ms | 6.96ms |
| YOLOv5n DS | 640 | 0.993 | 0.890 | 25.07ms | X |
| YOLOv5s DS | 640 | 0.989 | 0.894 | 44.61ms | X |

[1] Los modelos se han ejecutado en una CPU I7-11800h (AVX-512 con *DeepSparse*) 8 núcleos y una GPU RTX3070 Max-q.
[2] Los tiempos de inferencia son en ejecución dato a dato (no ejecución en lote).
[3] Se han realizado 240 *epochs* (iteraciones) sobre el dataset de entrenamiento.
[4] Para la obtención de las métricas de precisión se ha hecho uso del*val.py* y para los cálculos de inferencia *benchmark.py* ambos del repositorio de DeepSparse, ejecutado 5 veces y obteniendo la media.

Para continuar con el despliegue del modelo en el dispositivo móvil, tenemos que descartar el modelo optimizado y proseguir con el simple, ya que, como se comentará

en el apartado de la transformación a *TFlite*, hay capas del modelo optimizado que no se pueden convertir ni ejecutar con el motor de *Tensorflow Lite* que se utilizará en la aplicación móvil.

## 3.5. Evaluación de los resultados

En situaciones de buena luminosidad el modelo responde correctamente pero, si se introducen efectos como el desenfoque de movimiento, destellos, oclusión, orientación casi perpendicular, el modelo deja de responder correctamente perdiendo por completo en muchas instancias la detección. Aun con la aplicación de aumentación de los datos durante el entrenamiento, no es suficiente para preparar el modelo ante estas situaciones. Los *datasets* que se han utilizado no contienen datos que sufran de estos efectos por lo que es entendible que el modelo no responda adecuadamente ante ese tipo de entradas.

# Capítulo 4

# Mejorando el rendimiento: cuantización, poda y aceleradores

## 4.1. Cuantización

La cuantización es una de las técnicas de adelgazamiento de redes neuronales que mejores resultados produce. Se entiende por adelgazamiento la reducción del tamaño de la red, o de la complejidad de las operaciones orientadas a reducir el tiempo de inferencia.

Existen dos categorías de cuantización:

- **Cuantización post entrenamiento:** Se reduce la precisión de los números de los pesos y las activaciones de las redes neuronales una vez el modelo ya se ha entrenado. Habitualmente, se produce una reducción de coma flotante de 32 bits a 16 bits o incluso a enteros con signo de 8 bits. Al aplicar este procedimiento sobre el modelo ya entrenado, se incurre en una reducción de las métricas de precisión del modelo, aunque habitualmente, para mitigar esta reducción, se calibra la cuantización en base a una pequeña porción del *dataset*. Es el método más sencillo de aplicar la cuantización.
- **Cuantización durante el entrenamiento:** Se añaden las modificaciones de la precisión de los pesos y las activaciones durante el proceso de entrenamiento, reduciendo el empeoramiento de la métrica de precisión del modelo. Para esto, se introducen falsas etapas de cuantización de los pesos y las activaciones, de forma que el modelo experimenta los efectos de la reducción de precisión. Los ajustes que realiza el modelo en esta simulación son registrados y empleados posteriormente para realizar la simplificación final del modelo. Es la cuantización más difícil de aplicar ya que se debe realizar un buen ajuste continuo del optimizador.

Matemáticamente el proceso de cuantización se define como:

$$x_q = redondear\left(\frac{1}{e}x + z\right)$$

Donde:

- $x_q$ es el valor cuantizado.
- $x$ valor inicial.
- $e$ es el factor de escala.
- $z$ es el punto cero.

Este factor de escala se almacena en los modelos más recientes como metadato para posteriormente, durante la ejecución del modelo poder convertir el valor de entrada a la precisión reducida aceptada por el modelo y también transformar la salida cuantizada al valor con precisión en coma flotante en 32 bits real.

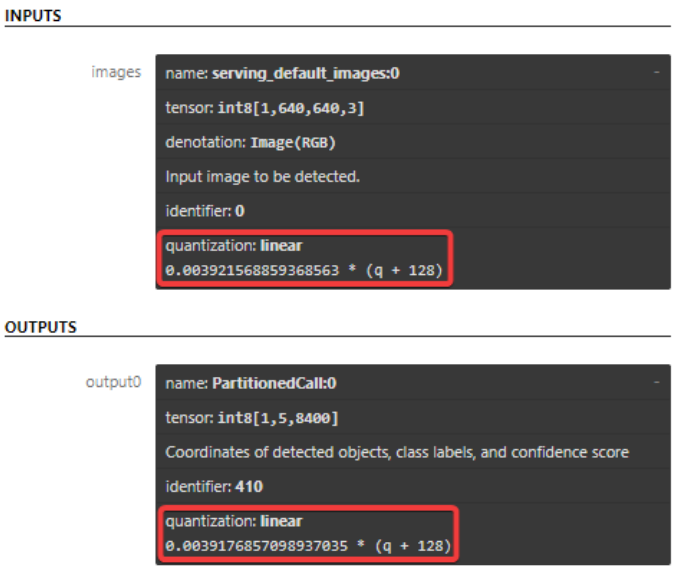



Figura 4.1: Datos de la capa de entrada y salida de un modelo YoloV8n cuantizado post entrenamiento.

Para cuantizar nuestro modelo YoloV5 podemos utilizar la recetas y herramientas del repositorio de Nerualmagic, simplemente nos basta con repetir el entrenamiento con una de las recetas que contenga la definición de la cuantización. Ejemplos: `yolov5.pruned_quantized.md` o `yolov5.transfer_learn_pruned_quantized.md`. Este método generará el modelo cuantizado pero no sera compatible con el resto de motores de inferencia.

Otra opción es transformar el modelo al formato *ONNX* (intercambio de redes neuronales abiertas) que sirve de nexo con otras APIs de entrenamiento de redes neuronales y que nos permite utilizar las herramientas de cuantización de otras APIs que sean compatibles con el motor que empleemos. En este caso se utilizará el optimizador de modelos de OpenVino [26] junto a la cuantización de Tensorflow que finalmente se transformará

a Tensorflow Lite. Existen librerías que ya integran todas estas operaciones como la denominada *openvino2tensorflow*[27] o la más reciente *onnx2tf* [28] que ya se encuentra integrada en el ecosistema *Yolo* en su versión V8 [29]. Todo este proceso se detallará en el apartado 5.1.

Resultados de emplear la receta de Neuralmagic:

Cuadro 4.1: Comparación caracteristicas YoloV5n y YoloV5n cuantizado

| Modelo | Tamaño (px) | $mAP_{val}$ 50-95 | $mAP_{val}$ 50 | Vel CPU (ms) |
|---|---|---|---|---|
| YOLOv5n DS | 640 | 0.993 | 0.890 | 25.07ms |
| YOLOv5n DS cuantizado | 640 | 0.992 | 0.879 | 21.23ms |

[1] Los modelos se han ejecutado en una CPU I7-11800h y una GPU RTX3070 Max-q.
[2] Los tiempos de inferencia son en ejecución dato a dato (no ejecución en lote).
[3] Se han realizado 50 *epochs* (iteraciones) sobre el dataset de entrenamiento.
[4] Para la obtención de las métricas de precisión se ha hecho uso de *val.py* y para los cálculos de inferencia de *benchmark.py*, ambos del repositorio de DeepSparse, ejecutado 5 veces y obteniendo la media.

Tras la cuantización el modelo ha sufrido una pérdida inferior al 1 % en la calidad de la detección, y a cambio se ha obtenido una reducción del 16 % en el tiempo de inferencia.

## 4.2. Poda

La poda es el proceso por el cual se eliminan elementos del modelo que se consideran superfluos, dado que su efecto en el resultado final es despreciable o porque se considere que se puede eliminar para mejorar el rendimiento a cambio de una penalización en las métricas del modelo.

Hay dos tipos de podas:

- **Estructurada**: Reduce del tiempo de inferencia al eliminar bloques completos de la red neuronal:
  - **Filtros o canales**: Se eliminan filtros o canales de las capas de la red neuronal, reduciendo su complejidad y su tamaño.
  - **Neuronas**: Se eliminan neuronas y sus conexiones reduciendo significativamente el tamaño del modelo y su complejidad pero afectando notablemente a las métricas del modelo.
- **No estructurada**: Reduce el tamaño del modelo al convertir los parámetros o pesos con valores de poca magnitud a 0, haciendo que el modelo se “disperse”.

De forma similar a la anterior técnica, podemos repetir el entrenamiento utilizando la receta de Neuralmagic que realiza poda no estructurada durante el entrenamiento pero no será compatible con Tensorflow Lite.

No obstante, podemos hacer uso de las herramientas de Pytorch para cargar el modelo y realizar una poda estructurada o no estructurada e incluso ir eliminando o manipulando bloque a bloque el modelo. Este es el método que se ha empleado en el modelo final, con una poda del 10 %, es decir, eliminar el 10 % de los parámetros o conexiones menos importantes del modelo. Valores mas altos incurren en una gran pérdida de precisión cuando lo aplicamos sobre un modelo ligero como *YoloV5n*.

Resultados de emplear la receta de Neuralmagic:

Cuadro 4.2: Comparación características YoloV5n y YoloV5n con poda no estructurada realizada

| Modelo | Tamaño (px) | $mAP_{val}$ 50-95 | $mAP_{val}$ 50 | Vel CPU (ms) |
|---|---|---|---|---|
| YOLOv5n DS | 640 | 0.993 | 0.890 | 25.07ms |
| YOLOv5n DS podado (10 %) | 640 | 0.948 | 0.741 | 24.27ms |

[1] Los modelos se han ejecutado en una CPU I7-11800h y una GPU RTX3070 Max-q.
[2] Los tiempos de inferencia son en ejecución dato a dato (no ejecución en lote).
[3] Se han realizado 50 *epochs* (iteraciones) sobre el dataset de entrenamiento.
[4] Para la obtención de las métricas de precisión se ha hecho uso del*val.py* y para los cálculos de inferencia *benchmark.py*, ambos del repositorio de DeepSparse, ejecutado 5 veces y obteniendo la media.

Tras realizar la poda no estructurada del 10 % el modelo resultante se ha visto afectado considerablemente. Esto es esperado ya que el modelo que se está podando es el *nano* que presenta poca cantidad de parámetros y capas con respecto al *small*, 1 millón de parámetros y 200 capas frente a 7 millones y 283 capas respectivamente. Una poda del 10 % afecta a muchos pesos que sí influyen en el resultado final, y no compensa por la escasa reducción del tiempo de inferencia.

## 4.3. Aceleradores

### 4.3.1. GPU

La unidad de procesamiento gráfico (GPU) mejora el rendimiento del modelo gracias a la gran capacidad de paralización de las operaciones de las redes neuronales. Es el acelerador por excelencia para tareas de entrenamiento como de inferencia de los modelos de redes neuronales. Es preferible sobre la ejecución en CPU cuando sea posible siempre que no existan recursos energéticos limitados.

### 4.3.2. DSP

Es posible hacer uso del procesador de señales digitales (DSP), incluido en la gran mayoría de dispositivos móviles *Snapdragon* [30] actuales de prestaciones medias/altas para acelerar parte de las operaciones matemáticas de la red neuronal sin incurrir en los

costes energéticos que acarrearía utilizar la GPU. Las librerías y SDK que permiten su uso requieren que el modelo se encuentre cuantizado a enteros de 8 bits o coma flotante de 16 bits.

### 4.3.3. NPU

La unidad de procesamiento neuronal (NPU) esta diseñada específicamente para acelerar las tareas de inteligencia artificial y *deep learning*. Están optimizadas para realizar operaciones de redes neuronales de forma eficiente y con un bajo consumo de energía. Se trata de un acelerador comercialmente reciente por lo que existe muy poca documentación y librerías que integren su uso.

### 4.3.4. TPU

La unidad de procesamiento tensorial (TPU) es propietario de Google para la aceleración de redes neuronales que se especializa en ejecutar modelos de baja precisión. Solo se incluyen en la gama de dispositivos móviles *Pixel* a partir de su versión 6. Es posible adquirir estos aceleradores como accesorio a un sistema existente o utilizar la plataforma *Codelab* para ejecutar las redes en estos aceleradores. Hay un estudio [31] de su uso en centros de datos donde se obtienen resultados muy superiores frente al uso de GPUs.

# Capítulo 5

# Aplicación Android

## 5.1. Transformaciones del modelo Pytorch a Tensorflow Lite

Para poder ejecutar el modelo YoloV5n bajo el motor de Tensorflow Lite se ha necesitado realizar varias transformaciones y modificaciones al modelo. Entre ellas se deben realizar las conversiones:


Figura 5.1: Transformaciones a realizar para obtener el modelo en formato *tflite*.

Durante este proceso se han aplicado las técnicas y herramientas recogidas en las secciones 4.1 y 4.2.

Una vez se obtuvo el modelo entrenado, se ha procedido a realizar una poda no estructurada del 10 %. Para ello se creo un script en Python *prune.py* incluido como anexo A.1 que hace uso de las librerias de Pytorch y Yolo para cargar y realizar la modificación.

Luego, el modelo resultante es necesario convertirlo a ONNX. Para ello se ha utilizado el script *export.py* disponible en el repositorio de YoloV5:

```
python export.py --weights ./yolov5-deepsparse/yolov5s-sgd5/weights/best.pt \
    --include onnx \
    --imgsz 640 \
    --simplify
```

- **weights**: Ruta al modelo que se ha podado.

- **include**: Formato a generar.
- **imgsz**: Tamaño de la entrada del modelo.
- **simplify**: Aplica optimizaciones de ONNX.

Una vez creado el fichero ONNX, se debe descargar e instalar el toolkit de OpenVino para poder utilizar el conversor *openvino2tensorflow*. El *toolkit* se instala como aplicación del sistema, y entre los sistemas oficialmente soportados no se encuentra Arch Linux, sistema usado para la realización del proyecto. Debido a esto y para evitar conflictos con el resto de herramientas, se ha empleado Docker para virtualizar una imagen de Ubuntu (*nvidia/cuda:10.2-cudnn8-devel-ubuntu18.04*) con soporte *CUDA* en la que se instaló el *toolkit* y se le asignó la gráfica del sistema para realizar la transformación. Para la creación de la imagen del contenedor se empleado el siguiente *Dockerfile* A.2.

```
docker build ./ -f ./docker/Dockerfile  -t openvino_suite
```

Una vez creada la imagen, se ha ejecutado el contenedor y se ha accedido a él a través de la terminal bash enlazando la carpeta actual con el espacio de trabajo del contenedor.

```
docker run -it --gpus all -v `pwd`:/workspace openvino_suite bash
```

Antes de haberse realizado la conversión, es necesario identificar el nombre único de las últimas capas del modelo antes de pasar a las capa de detección. Estas capas implementan operaciones no soportadas por la API de inferencia neuronal de Android *NNAPI* que permite el uso de la GPU y NPU del dispositivo móvil. Esto hace que se tengan que implementar estas últimas capas en código externo y ejecutarlos en CPU por lo que la ejecución con aceleradores no va a realizarse en su totalidad.

Para visualizar las capas de los modelos se ha utilizado la herramienta *Netron*, accesible por internet o alojándola temporalmente en local, como se muestra en la figura 5.2. Para identificar las capas, se debe buscar cuáles son las últimas operaciones de convolución '`Conv`' y anotar la propiedad '`name`'. Estos nombres provienen de la definición del modelo, por lo que si se mantiene el mismo modelo inicial, también lo harán los nombres.

Habiendo anotado los nombres de las ultimas capas, se ha procedido a realizar la optimización y conversión al formato OpenVino cuyo resultado se compone de tres ficheros: '`.bin`', '`.mapping`' y '`.xml`'.

```
python3 /opt/intel/openvino_2021.3.394/deployment_tools/model_optimizer/mo.py
 ↪  \
 --input_model bestv5n640.onnx \
 --input_shape [1,3,640,640] \
 --output_dir ./openvino \
 --data_type FP32 \
 --output Conv_245,Conv_344,Conv_443
```


Figura 5.2: Visualizador Netron con el modelo YoloV5n a transformar.

Una vez hecha la conversión a *OpenVino* la siguiente transformación sería a *TensorFlow* y *TensorFlow Lite*. En esta transformación se ha hecho uso de *openvino2tensorflow* que permite generar tanto el fichero para ejecutar en *TensorFlow* (‘`.pb`’) como el de TensorFlow Lite (‘`.tflite`’) aunque este último se puede obtener teniendo el modelo en formato *TensorFlow* y usando su propia herramienta.

```
source /opt/intel/openvino_2021/bin/setupvars.sh
export PYTHONPATH=/opt/intel/openvino_2021/python/python3.6/:$PYTHONPATH
openvino2tensorflow \
--model_path ./openvino/bestv5n640.xml \
--model_output_path tflite \
--output_pb \
--output_saved_model \
--output_no_quant_float32_tflite
```

Por último, para realizar la cuantización del modelo se debe configurar un dataset de calibración. Para ello se ha hecho uso del conjunto de datos de validación que se han usado durante la fase de entrenamiento del modelo. Luego, se ha modificado el script ‘`quantize.py`’ que contiene el procedimiento de cuantización y exportación a Tensorflow Lite para indicar la ruta al conjunto de datos de calibración. Una vez configurado se ha procedido a su ejecución.

```
python3 quantize.py --input_size 640 --pb_path
  ↪  /workspace/yolov5/tflite/model_float32.pb \
--output_path /workspace/yolov5/tflite/bestv5n640int8.tflite \
--calib_num 100
```

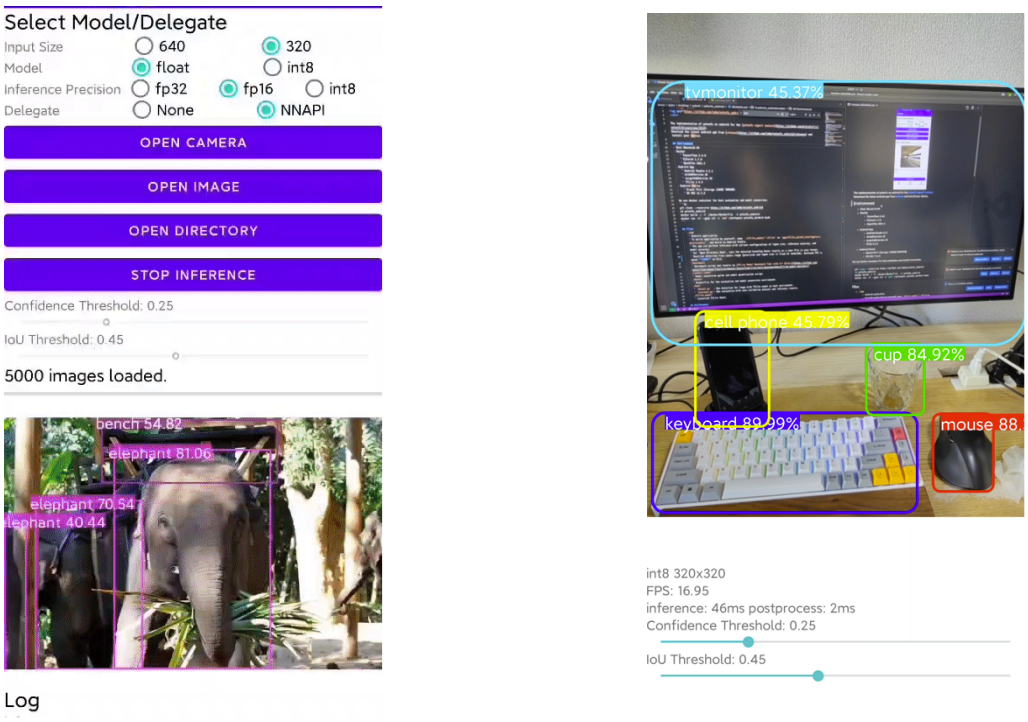
Si se observa el modelo resultante en *Netron*, se puede ver como se han introducido en la entrada una nueva capa de cuantización y en las capas de salida un capa de des-cuantización.

(a) Capa de entrada. (b) Capa de salida.

Figura 5.3: Capas de entrada y salida del modelo cuantizado visualizado en Netron

## 5.2. Ampliando una aplicación ya existente

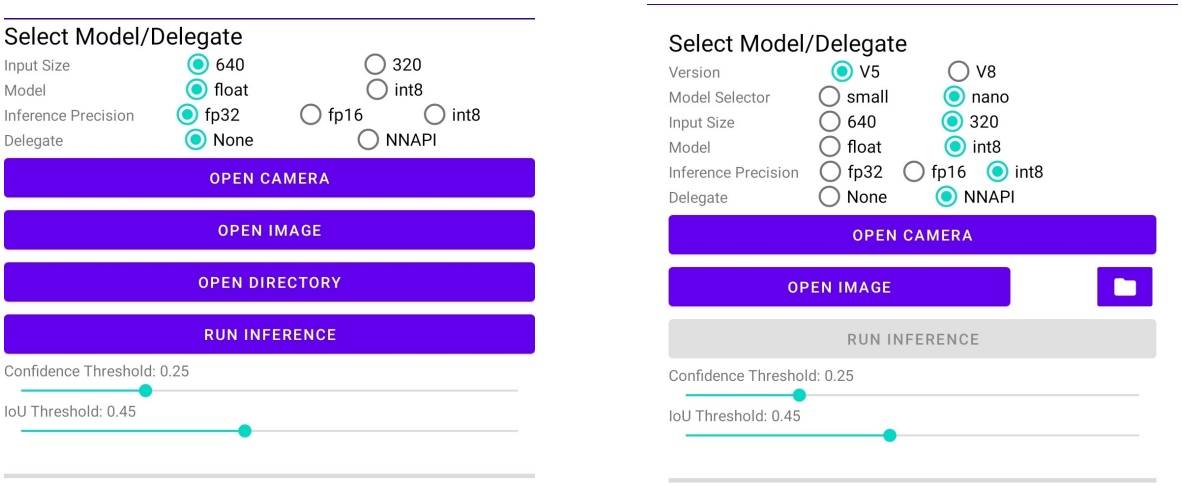
Existen varias aplicaciones oficiales tanto de Google [32] como de Tensorflow [33] que sirven de plantilla para probar los modelos entrenados, pero en un primer intento, no se consiguió hacer funcionar ninguna de estas aplicaciones con el modelo generado. Esto es debido a que utilizan versiones antiguas de las librerías de TensorFlow que no soportan algunas operaciones que introduce la red YoloV5 y migrarlas costaría actualizar mucho código deprecado. Se ha optado por usar una aplicación completa que implementa la red YoloV5 en su versión *small* [34] escrito en Java y en C++ para procesar las redes no soportadas por el delegado NNAPI que permite la ejecución del modelo en hardware dedicado como la GPU o el NPU/DSP (Unidad de procesamiento neuronal/Procesador de señales digitales).




(a) Menú inicial de la app. (b) Sección de la cámara.

Figura 5.4: App utilizada para ejecutar el modelo

Tras haber importado el proyecto en el Android Studio solo ha hecho falta cambiar los fragmentos de código relacionados con las etiquetas y cargar el modelo dentro de la carpeta de recursos siguiendo la denominación del fichero establecida para poder conseguir que el modelo se ejecutara. No obstante, en el proyecto se ha querido evaluar el rendimiento del modelo en su versión *small* y *nano* además de tener en cuenta otras métricas de rendimiento de la aplicación. De modo que se han realizado mejoras en la aplicación tanto en la interfaz como en el código de la inferencia. Como última instancia se ha añadido el código necesario de C++ y de Java, ver anexo A.4, para poder realizar la inferencia de los nuevos modelos YoloV8, aunque sin lograr la ejecución del modelo completamente cuantizado.




(a) Antigua interfaz. (b) Nueva interfaz.

Figura 5.5: Comparación de los cambios realizados al menú de inicio


(a) Antigua interfaz. (b) Nueva interfaz.

Figura 5.6: Comparación de los cambios realizados al menú de la cámara

## 5.3. Evaluación de los resultados para su aplicación en tiempo real

Para que sea viable la implementación de este sistema en tiempo real es necesario que la velocidad de inferencia sea inferior a 28-30ms, dejando de margen 5-7ms de post procesamiento y representación de los resultados, ya que, aunque no se va a proceder a la decodificación del código de barras, existe un procesamiento posterior que hay que tener en cuenta.

Cuadro 5.1: Comparación ejecución de los modelos *nano* y *small* en la aplicación Android frente a su contrapartida de escritorio usando DeepSparse

| Modelo | Tamaño (px) | CPU DS 8b (ms) | CPU 32b (ms) | GPU 16b (ms) | DSP 8b (ms) |
|---|---|---|---|---|---|
| YOLOv5n | 320 | 9.45ms | 37ms | 15ms | 8ms |
| YOLOv5s | 320 | 18.26ms | 125ms | 26ms | 20ms |
| YOLOv5n | 640 | 20.87ms | 145ms | 38ms | 28ms |
| YOLOv5s | 640 | 34.82ms | 515ms | 86ms | 68ms |

[1] 32b = 32 bits.
[2] Los modelos de escritorio se han ejecutado en una CPU I7-11800h (AVX-512 con *DeepSparse*) 8 núcleos y una GPU RTX3070 Max-q mientras que los de Tensorflow Lite se han ejecutado en un *Xiaomi Poco F2 Pro* con procesador Snapdragon 865.
[3] Los tiempos de inferencia son en ejecución dato a dato (no ejecución en lote).
[4] Se han realizado 240 *epochs* (iteraciones) sobre el dataset de entrenamiento.
[5] Para la obtención de los cálculos de inferencia se ha hecho uso del registro *inference avg* de la aplicación Android en el caso de los modelos de móvil mientras que en los de escritorio se ha empleado el *benchmark.py* del repositorio de DeepSparse.

Evaluando los resultados obtenidos en la tabla 5.1 podemos ver como hay tres modelos que consiguen cumplir con los tiempos de inferencia máximos. De estos tres, dos pertenecen a la versión *nano* con entradas de 320 y 640 pixeles y un modelo *small* con entradas de 320 píxeles.

Si tenemos en cuenta la cantidad de parámetros que implementa cada modelo, el *small* no aporta gran mejora sobre la detección como se ha visto en la tabla 3.2. Emplear un modelo con entrada de imágenes superiores mejora considerablemente los resultados y la detección cuando se enfrenta entornos complejos.

Sin embargo, hay situaciones que no puede salvar y comienza a no detectar correctamente, como se aprecia en las figuras 5.7, 5.9 y 5.8. Esto va de la mano del *dataset* que se ha empleado que carece de datos en los que se presentan estos problemas y del poco uso de la aumentación de los datos durante el entrenamiento.

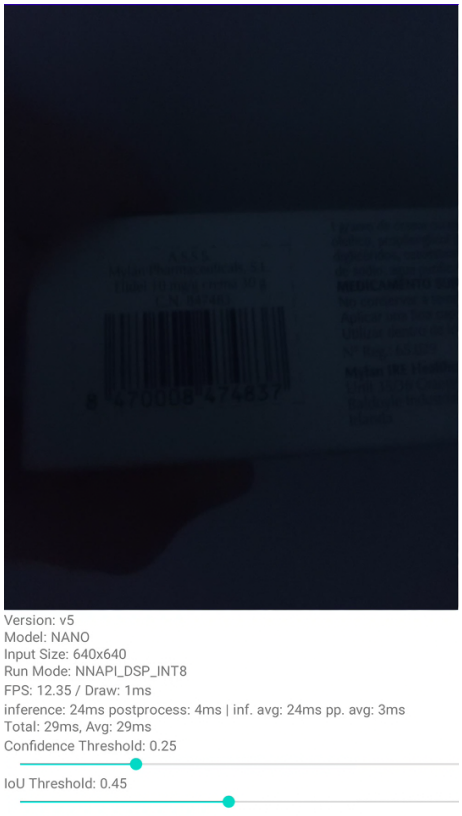
En situaciones de buena iluminación y sin desenfoque la detección se realiza correctamente, como ilustra la figura 5.10.


(a) Desenfoque


(b) Oclusión

Figura 5.7: Ejemplos de problemas: desenfoque y oclusión


(a) Destello



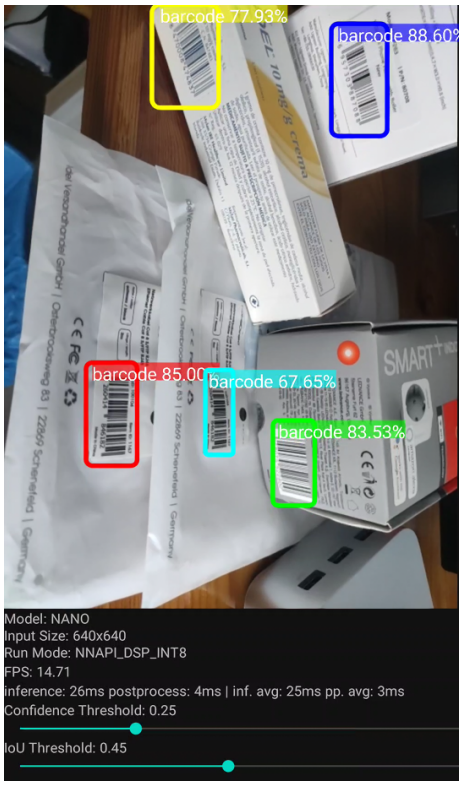

(b) Baja luminosidad

Figura 5.8: Ejemplos de problemas: destello y baja luminosidad

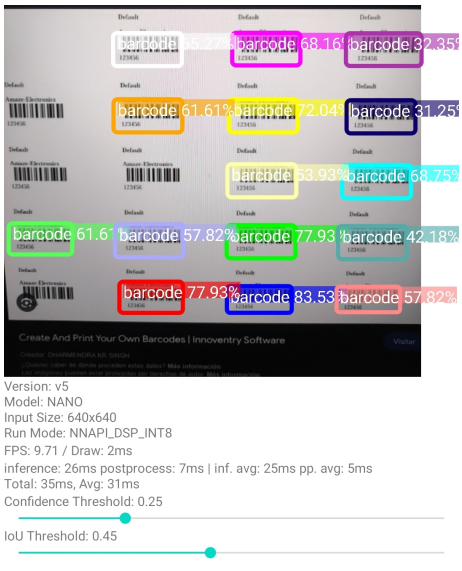

(a) Inclinación


(b) Distancia

Figura 5.9: Ejemplos de problemas: inclinación y distancia




(a) Productos con códigos de barras




(b) Lista de códigos de barras

Figura 5.10: Ejemplos de correcto funcionamiento

# Capítulo 6

# Conclusions and future lines

## 6.1. Conclusions

The ability to run a real-time barcode detection model on a mobile device with good results has been achieved, having in consideration the dataset used and the fact that the YOLO layers have not been manually modified. To reach this result, multiple conversions have had to be performed which, although optimised, require manual evaluation and pruning of unnecessary layers or filters. These conversions rely in many cases on third party tools, which can lead to incompatibilities in the resulting model. However, these processes are being integrated into the YOLO ecosystem, so many of the procedures are already largely simplified in the latest versions.

As evidence of this, the necessary code to add the compatibility with YOLOv8 has been implemented in the Android application. A nano model has been trained and quantization has been applied to it, generating a quantified model in a simple way. Running this YOLOv8 model does not show any performance improvements, which is expected, as the latest versions have focused on extending functionality and improving existing tools.

This has not been the case for the TensorFlow Lite and Google application libraries and templates for neural network acceleration as they are outdated in many aspects and also have incompatibilities with some of the operations of the new neural network models. In part, these being out of date is understandable due to the great push of the NPUs of the different ARM processor manufacturers, where it is necessary to use their SDK and documentation, as in the case of Qualcomm through its developer portal, with prior registration.

On the other hand, the training of the model has been used to test Neuralmagic's specialised DeepSparse engine, which has given very good results. Although it cannot be implemented on all architectures, it is an engine that only uses CPU for processing, with the possibility of making use of several CPU cores simultaneously.

Finally, following the addition of code to support YOLOv8 models, an application has been implemented in which we can easily add and run YOLO models of various versions, as long as they are compatible with the TensorFlow Lite engine and libraries.

This is why it is considered that the objectives that were established in the draft of the TFG have been accomplished:

- Carrying out transfer learning using the YOLO network.
- Thinning of the trained model.
- Analysis and comparison of the models.
- Development of the Android application.
- Use of alternative methods.

## 6.2. Future lines

There are some future lines we can follow:

- Making an in-house dataset where the issues found when testing the model detection on the barcodes are addressed.
- Do manual evaluation and pruning of the model's layers.
- Improve the cyclic code of the Android app as the cumulated times doesn't match the percieved framerate.
- Streamline the load of the new trained models and its labels without needing to rebuild the application, collaterally this should also free some currently wasted space used by models.
- To try changing the Darknet backbone of the model with other optimized for mobile deployments like MobilenetV2.
- Add the Qualcomm Hexagon delegate or NPU SDK to the android application, this way newer ARM processors DSPs and NPUs are supported and also boost the speed of the inference, as NNAPI serves as the common translation layer between accelerators, being non optimal by definition.

# Capítulo 7

# Presupuesto

Los costes de este tipo de proyectos se pueden reducir significativamente si utilizamos sistemas en la nube como Google Cloud para entrenar nuestros modelos rápidamente. Además, usar *datasets* ya preparados nos puede ahorrar una carga de trabajo significativa y costes en software. En este TFG se ha tomado un enfoque mas tradicional utilizando hardware adquirido personalmente. Por ultimo, el resto del presupuesto iría destinado a cubrir el coste humano con un programador junior con experiencia en python y en java o kotlin.

## 7.1. Licencias de software

| Nombre del software | Coste | Anotaciones |
|---|---|---|
| Python | 0.00€ | Código libre y gratuito |
| Java | 0.00€ | Código libre y gratuito |
| Pytorch | 0.00€ | Código libre y gratuito |
| Docker | 0.00€ | Código libre y gratuito |
| OpenVino Toolkit | 0.00€ | Código libre y gratuito |
| ONNX | 0.00€ | Código libre y gratuito |
| Tensorflow | 0.00€ | Código libre y gratuito |
| Tensorflow Lite | 0.00€ | Código libre y gratuito |
| Git | 0.00€ | Código libre y gratuito |
| VSCode | 0.00€ | Código libre y gratuito |
| Android Studio | 0.00€ | Código libre y gratuito |
| Jetbrain Youtrack | 0.00€ | Licencia gratuita hasta 10 usuarios |
| Overleaf | 0.00€ | Edición comunidad alojada en servidor propio |
| Android | 0.00€ | Código libre y gratuito |
| EndeavourOS | 0.00€ | Código libre y gratuito |
| | | |
| Total: | 0€ | |

Cuadro 7.1: Coste de las licencias de software utilizadas

## 7.2. Materiales

| Material | Valor | Factor de amortización | Coste |
|---|---|---|---|
| Ordenador sobremesa | 836.95€ | 20 % | 167.39€ |
| Tarjeta gráfica RTX 4070 | 539.99€ | 20 % | 108€ |
| Móvil con Snapdragon 865 | 249.00€ | 5 % | 12.45€ |
| Móvil con Snapdragon Gen 2 | 479.00€ | 5 % | 23.95€ |
| | | | |
| Total: | | | 311.79€ |

Cuadro 7.2: Coste de los materiales utilizados

## 7.3. Costes de personal

Asumiendo un coste del programador de 20€ por hora.

| Tarea | Horas | Coste |
|---|---|---|
| Búsqueda y preparación del *dataset* | 21 | 420.00€ |
| Análisis y elección del modelo de la red neuronal convolucional | 15 | 300.00€ |
| Preparación y configuración de la plataforma de entrenamiento | 23 | 460.00€ |
| Entrenamiento, evaluación y afinamiento del entrenamiento | 74 | 1,480.00€ |
| Aplicar técnicas de adelgazamiento y transformaciones | 50 | 1,000.00€ |
| Desarrollo de la aplicación Android | 90 | 1,800.00€ |
| Validación de la aplicación en los distintos SOCs | 20 | 400.00€ |
| | | |
| Total | 293 | 5,860.00€ |

Cuadro 7.3: Costes de personal.

## 7.4. Presupuesto final del proyecto

| Motivo | Coste |
|---|---|
| Licencias de software | 0€ |
| Materiales | 311.79€ |
| Mano de obra | 5,860.00€ |
| | |
| Coste | 6,171.79€ |
| Beneficio (6 %) | 370.31€ |
| | |
| Presupuesto total del proyecto | 6,542.10€ |

Cuadro 7.4: Presupuesto final del proyecto.

# Apéndice A

# Códigos

## A.1. Script `prune.py` para realizar la poda del modelo

```python
from utils.torch_utils import prune
from utils.torch_utils import select_device
from models.experimental import attempt_load
from utils.torch_utils import model_info
import torch

device = select_device("0") # Cargamos el primer dispositivo disponible, habitualmente la GPU
model = attempt_load("./yolov5-deepsparse/yolov5n-sgd-ds/weights/best5n640.pt", map_location=device)  # Cargamos el modelo entrenado
model_info(model) # Informacion general del modelo
torch.save({'model': model}, 'best5n640.pt')  # Cargamos una copia del modelo entrenado

prune(model, 0.1) # Poda no estructurada del 10%
model_info(model)
torch.save({'model': model}, 'best5n640_pruned.pt') # Guardamos el modelo podado
```

## A.2. Dockerfile para el contenedor de OpenVino con las dependencias para *onnx2tensorflow*

```
FROM nvidia/cuda:10.2-cudnn8-devel-ubuntu18.04

# se va a acceder desde la terminal por lo que no es necesario tener activo el gestor de ventanas.
ENV DEBIAN_FRONTEND noninteractive

RUN apt-get update --fix-missing
RUN apt-get install -y python3 python3-pip
# instalamos los paquetes necesario para poder realizar la cuantización del modelo Tensorflow
RUN pip install --upgrade pip
RUN pip install tensorflow-gpu==2.4.0
```

```
RUN pip install torch==1.7.0
RUN pip install tensorflow-datasets

# instalamos openvino en modo headless y sus dependencias de python
ENV http_proxy $HTTP_PROXY
ENV https_proxy $HTTP_PROXY
ARG DOWNLOAD_LINK=https://registrationcenter-download.intel.com/akdlm/irc_nas/17662/l_open
ARG INSTALL_DIR=/opt/intel/computer_vision_sdk
ARG TEMP_DIR=/tmp/openvino_installer
RUN apt-get update && apt-get install -y --no-install-recommends \
    wget \
    cpio \
    sudo \
    lsb-release && \
    rm -rf /var/lib/apt/lists/*
RUN mkdir -p $TEMP_DIR && cd $TEMP_DIR && \
    wget -c $DOWNLOAD_LINK && \
    tar xf l_openvino_toolkit*.tgz && \
    cd l_openvino_toolkit* && \
    sed -i 's/decline/accept/g' silent.cfg && \
    ./install.sh -s silent.cfg && \
    rm -rf $TEMP_DIR
RUN pip install networkx
RUN pip install defusedxml

# instalamos los paquetes de pythob necesarios para la conversion con openvino2tensorflow
RUN pip install netron
RUN pip install onnx
RUN pip install onnx-simplifier
RUN pip install tensorflow-datasets
RUN pip install openvino2tensorflow==1.17.2
RUN pip install gdown

# configuracion de usuario y ruta del contenedor
ARG USERNAME=developer
ARG GROUPNAME=developer
ARG UID=1000
ARG GID=1000
ARG PASSWORD=developer
RUN groupadd -g $GID $GROUPNAME && \
    useradd -m -s /bin/bash -u $UID -g $GID -G sudo $USERNAME && \
    echo $USERNAME:$PASSWORD | chpasswd && \
    echo "$USERNAME   ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers
USER $USERNAME
ENV HOME /home/developer
```

## A.3. Script `quantize.py` para realizar la cuantización del modelo en formato TensorFlow

```
import argparse
import sys
import os
```

```
import glob
import cv2

import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds

def quantize_model(INPUT_SIZE, pb_path, output_path, calib_num, image_dir):
    input_shapes = [(3, INPUT_SIZE, INPUT_SIZE)]
    def representative_dataset_gen():
        files = list(glob.glob(image_dir + "/*.jpg"))
        for i, imgpath in enumerate(files[:calib_num]):
            print('calibrating...', i)
            image = cv2.imread(imgpath)
            images = []
            for shape in input_shapes:
                data = tf.image.resize(image, (shape[1], shape[2]))
                tmp_image = data / 255.
                tmp_image = tmp_image[np.newaxis,:,:,:]
                images.append(tmp_image)
            yield images

    input_arrays = ['inputs']
    output_arrays = ['Identity', 'Identity_1', 'Identity_2']
    converter = tf.compat.v1.lite.TFLiteConverter.from_frozen_graph(pb_path,
    ↪  input_arrays, output_arrays)
    converter.experimental_new_quantizer = False
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
    converter.allow_custom_ops = False
    converter.inference_input_type = tf.uint8
    converter.inference_output_type = tf.float32
    converter.representative_dataset = representative_dataset_gen
    tflite_model = converter.convert()
    with open(output_path, 'wb') as w:
        w.write(tflite_model)
    print('Quantization Completed!', output_path)

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--input_size', type=int, default=640)
    parser.add_argument('--pb_path',
    ↪  default="/workspace/yolov5/tflite/model_float32.pb")
    parser.add_argument('--output_path',
    ↪  default='/workspace/yolov5/tflite/model_quantized.tflite')
    parser.add_argument('--calib_num', type=int, default=100, help='number of
    ↪  images for calibration.')
    # ruta al dataset de calibracion
    parser.add_argument('--image_dir', type=str,
    ↪  default="/home/mario/repositorios/yolov5s_android/dataset/barcodes/validation")
    args = parser.parse_args()
    quantize_model(args.input_size, args.pb_path, args.output_path,
    ↪  args.calib_num, args.image_dir)
```

## A.4. Función JNI en C++ adaptada para procesar los resultados del modelo YoloV8

```
extern "C"
JNIEXPORT jobjectArray JNICALL
    ↪ Java_com_example_tflite_1yolo_1test_TfliteRunner_postprocessV8 (JNIEnv *env,
    jobject thiz,
    jobjectArray recognitions,
    jint w, jint h,
    jfloat confidence_threshold,
    jfloat iou_threshold,
    jint num_items_threshold,
    jint num_classes, jint input_size) {

    std::vector<DetectedObject> proposals;
    std::vector<DetectedObject> objects;

    // Initialize the C++ vector
    std::vector<std::vector<float>> vec(h, std::vector<float>(w, 0.0f));
    for (int i = 0; i < h; ++i) {
        jfloatArray row = (jfloatArray) env->GetObjectArrayElement(recognitions,
        ↪ i);
        jfloat *rowData = env->GetFloatArrayElements(row, JNI_FALSE);

        for (int j = 0; j < w; ++j) {
            vec[i][j] = rowData[j];
        }

        // Release the local references
        env->ReleaseFloatArrayElements(row, rowData, JNI_ABORT);
        env->DeleteLocalRef(row);
    }

    // find boxes with score > threshold and class > threshold
    for (int i = 0; i < w; ++i) {
        // find class index with max class score
        int class_index = 0;
        float class_score = -FLT_MAX;

        // get scores for all class indexes of current box
        std::vector<float> classes(num_classes);
        for (int c = 0; c < num_classes; c++) {
            classes[c] = vec[c + 4][i];
        }

        // find class index with max class score
        for (int c = 0; c < num_classes; ++c) {
            if (classes[c] > class_score) {
                class_index = c;
                class_score = classes[c];
            }
        }

        // if class score is less than threshold, move to next box
        if (class_score > confidence_threshold) {
            float dx = vec[0][i];
```

```
52              float dy = vec[1][i];
53              float dw = vec[2][i];
54              float dh = vec[3][i];
55
56              DetectedObject obj;
57              obj.rect.x = dx;
58              obj.rect.y = dy;
59              obj.rect.width = dw;
60              obj.rect.height = dh;
61              obj.index = class_index;
62              obj.confidence = class_score;
63
64              proposals.push_back(obj);
65          }
66      }
67      // sort all proposals by score from highest to lowest
68      qsort_descent_inplace(proposals);
69
70      // apply nms with nms_threshold
71      std::vector<int> picked;
72      nms_sorted_bboxes(proposals, picked, iou_threshold);
73
74      int count = (int) std::min((float) picked.size(), (float)
        ↪  num_items_threshold);
75
76      objects.resize(count);
77      vector<bbox> bbox_candidates;
78      for (int i = 0; i < count; i++) {
79          objects[i] = proposals[picked[i]];
80          float x0 = std::max(0.f, objects[i].rect.x - objects[i].rect.width / 2);
81          float y0 = std::max(0.f, objects[i].rect.y - objects[i].rect.height /
            ↪  2);
82          float x1 = std::min(1.f, objects[i].rect.x + objects[i].rect.width / 2);
83          float y1 = std::min(1.f, objects[i].rect.y + objects[i].rect.height /
            ↪  2);
84
85          objects[i].rect.x = x0;
86          objects[i].rect.y = y0;
87          objects[i].rect.width = (x1 - x0);
88          objects[i].rect.height = (y1 - y0);
89          bbox box = bbox(x0 * input_size, y0* input_size, x1* input_size, y1*
            ↪  input_size, objects[i].confidence, objects[i].index);
90          bbox_candidates.push_back(box);
91      }
92
93      //return 2-dimension array [detected_box][6(x, y, width, height, conf,
        ↪  class)]
94      jobjectArray objArray;
95      jclass floatArray = env->FindClass("[F");
96      if (floatArray == NULL)
97          return NULL;
98      int size = bbox_candidates.size();
99      objArray = env->NewObjectArray(size, floatArray, NULL);
100     if (objArray == NULL)
101         return NULL;
102     for (int i = 0; i < bbox_candidates.size(); i++) {
103         int index = bbox_candidates[i].class_idx;
104         float x1 = bbox_candidates[i].x1;
```

```
105             float y1 = bbox_candidates[i].y1;
106             float x2 = bbox_candidates[i].x2;
107             float y2 = bbox_candidates[i].y2;
108             float confidence = bbox_candidates[i].conf;
109 
110             float boxres[6] = {x1, y1, x2, y2, confidence, (float) index};
111             jfloatArray iarr = env->NewFloatArray((jsize) 6);
112             if (iarr == NULL)
113                 return NULL;
114             env->SetFloatArrayRegion(iarr, 0, 6, boxres);
115             env->SetObjectArrayElement(objArray, i, iarr);
116             env->DeleteLocalRef(iarr);
117         }
118         return objArray;
119     }
```

# Glosario

**CNN** red neuronal convolucional. 4, 6, 8

**DSP** procesador de señales digitales. 21

**GPU** unidad de procesamiento gráfico. 21, 22, 24

**NPU** unidad de procesamiento neuronal. 22, 24

**RFID** identificación por radio frecuencia. 3

**SDK** *Software Development Kit*. 7, 22

**SGD** descenso de gradiente estocástico. 14

**SOC** system on a chip. 34

**TPU** unidad de procesamiento tensorial. 22

**UPC** código universal de producto. 1, 2

**YOLO** *You Only Look Once*. 6, 8, 9, 13

# Bibliografía


[1] ISO, "ISO/IEC 15426-1:2006.." https://www.iso.org/standard/43643.html, 2006.

[2] N. J. Woodland and B. Silver, "US2612994A - classifying apparatus and method." https://patents.google.com/patent/US2612994A/en, Oct 1949.

[3] IBM, "The UPC." https://www.ibm.com/history/upc, 2024.

[4] A. Hensel, "After years of hype, RFID is still struggling to catch on in retail." https://www.modernretail.co/retailers/after-years-of-hype-rfid-is-still-slowly-catching-on-in-retail/, 2020.

[5] Óscar Gómez-Cárdenez, J. G. Marichal-Hernández, J.-Y. Son, R. Pérez-Jiménez, and J. M. Rodríguez-Ramos, "An encoder–decoder architecture within a classical signal-processing framework for real-time barcode segmentation," *Sensors*, 2023. https://www.mdpi.com/1424-8220/23/13/6109.

[6] T. R. Tuinstra, *Reading barcodes from digital imagery*. PhD thesis, 2006. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=f0fe6ba5589d01d8c1968b2a5e21373030a9b721.

[7] S. Kaur and R. Maini, "Implementation of barcode localization technique using morphological operations," *International Journal of Computer Applications*, vol. 97, pp. 42–47, July 2014. https://ijcaonline.org/archives/volume97/number13/17068-7488/.

[8] P. Bodnár and L. G. Nyúl, "Efficient barcode detection with texture analysis," in *Signal Processing, Pattern Recognition, and Applications, Proceedings of the Ninth IASTED International Conference on*, pp. 51–57, 2012.

[9] T. Zhao, Y. Xie, Y. Wang, J. Cheng, X. Guo, B. Hu, and Y. Chen, "A survey of deep learning on mobile devices: Applications, optimizations, challenges, and research opportunities," *Proceedings of the IEEE*, vol. 110, no. 3, pp. 334–354, 2022.

[10] C. N. Yanes Mesa, "Localizador de códigos de barras mediante inteligencia artificial," 2022. https://riull.ull.es/xmlui/handle/915/29409.

[11] Ultralytics, "Yolov5." https://docs.ultralytics.com/es/yolov5/, 2020.

[12] B. D. Ben Souchet, "Github - bensouchet/barcode-datasets: A list of available barcode & qr code datasets — github.com." https://github.com/BenSouchet/barcode-datasets, 2022.

[13] Qualcomm, "Snapdragon 8 Gen 2 Mobile Platform." https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-8-gen-2-mobile-platform/, 2023.

[14] Google, "Aprendizaje automático destinado a desarrolladores de apps para dispositivos móviles." https://developers.google.com/ml-kit, 2024.

[15] Steenbakker, "A universal scanner for Flutter based on MLKit. Uses CameraX on Android and AVFoundation on iOS.." https://pub.dev/packages/mobile_scanner, 2024.

[16] ZXing community, "ZXing (Zebra Crossing) barcode scanning library for Java, Android." https://github.com/zxing/zxing, 2023.

[17] Scandit, "Scandit SDK Showcase." https://play.google.com/store/apps/details?id=com.scandit.demoapp, 2024.

[18] Dynamsoft, "Dynamsoft Barcode Scanner demo." https://play.google.com/store/apps/details?id=com.dynamsoft.demo.dynamsoftbarcodereaderdemo, 2024.

[19] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, and M. Grundmann, "On-device neural net inference with mobile gpus," *CoRR*, vol. abs/1907.01989, 2019. http://arxiv.org/abs/1907.01989.

[20] M. Tkachenko, M. Malyuk, A. Holmanyuk, and N. Liubimov, "Label Studio: Data labeling software," 2020-2022. Open source software available from https://github.com/heartexlabs/label-studio.

[21] T. Kamnardsiri, P. Charoenkwan, C. Malang, and R. Wudhikarn, "1D Barcode Detection: Novel Benchmark Datasets and Comprehensive Comparison of Deep Convolutional Neural Network Approaches," *Sensors*, vol. 22, p. 8788, Jan. 2022. https://www.mdpi.com/1424-8220/22/22/8788.

[22] Vantages, "p3.2xlarge pricing and specs - vantage." https://instances.vantage.sh/aws/ec2/p3.2xlarge, 2024.

[23] Ultralytics, "GitHub - ultralytics/yolov5: YOLOv5 in PyTorch >ONNX >CoreML >TFLite." https://github.com/ultralytics/yolov5, 2020.

[24] D. Neoh, "GitHub - dnth/yolov5-deepsparse-blogpost: Supercharging YOLOv5: How I Got 182.4 FPS Inference Without a GPU." https://github.com/dnth/yolov5-deepsparse-blogpost, 2022.

[25] M. K. Neuralmagic, "YOLOv5 on CPUs: Sparsifying to achieve GPU-level performance and a smaller footprint." https://neuralmagic.com/blog/benchmark-yolov5-on-cpus-with-deepsparse/, 2022.

[26] OpenVino, "Openvino - Model Optimizer Usage." https://docs.openvino.ai/2022.3/openvino_docs_MO_DG_Deep_Learning_Model_Optimizer_DevGuide.html, 2022.

[27] K. Hyodo, “GitHub - PINTO0309/openvino2tensorflow: PyTorch (NCHW) ->ONNX (NCHW) ->OpenVINO (NCHW) ->openvino2tensorflow ->TensorFlow/Keras (NHWC/NCHW) ->TFLite (NHWC/NCHW).” https://github.com/PINTO0309/openvino2tensorflow, 2022.

[28] K. Hyodo, “GitHub - PINTO0309/onnx2tf: Self-Created Tools to convert ONNX files (NCHW) to TensorFlow/TFLite/Keras format (NHWC).” https://github.com/PINTO0309/onnx2tf, 2024.

[29] Ultralytics, “GitHub - ultralytics/ultralytics: YOLOv8 in PyTorch >ONNX >OpenVINO >CoreML >TFLite.” https://github.com/ultralytics/ultralytics, 2023.

[30] Qualcomm, “Hexagon DSP Processor.” https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor, 2022.

[31] Qualcomm, “In-Datacenter Performance Analysis of a Tensor Processing Unit.” https://arxiv.org/pdf/1704.04760, 2017.

[32] Google, “MediaPipe Tasks Object Detection Android demo.” https://github.com/google-ai-edge/mediapipe-samples/tree/main/examples/object_detection/android, 2022.

[33] TensorFlow, “TensorFlow Lite Object Detection Android demo.” https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection/android, 2022.

[34] Y. Nitta, “Github - lp6m/yolov5s_android: Run yolov5s on Android device!.” https://github.com/lp6m/yolov5s_android, 2022.