ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

Evaluación, optimización e implementación de modelos convolucionales de visión artificial en dispositivos con recursos limitados

---

(Evaluation, optimization and deployment of convolutional computer vision models on resource-constrained devices)

Para acceder al Título de Grado en

Graduado en

Ingeniería de Tecnologías de Telecomunicación

Autor: Álvaro Revuelta Burgos

Director: Pablo Pedro Sánchez Espeso

Septiembre – 2025

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: Álvaro Revuelta Burgos

Director del TFG: Pablo Pedro Sánchez Espeso

Título: "Evaluación, optimización e implementación de modelos convolucionales de visión artificial en dispositivos con recursos limitados"

Title: "Evaluation, Optimization and Deployment of Convolutional Computer Vision Models on Resource-Constrained Devices"

Presentado a examen el día: 17 de Septiembre de 2025

Para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del Tribunal:

Presidente (Apellidos, Nombre): Fernández Solorzano, Víctor Manuel

Secretario (Apellidos, Nombre): García Gutiérrez, Alberto Eloy

Vocal (Apellidos, Nombre): Herrera Guardado, Amparo

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del TFG
(sólo si es distinto del Secretario)

Vº Bº del Subdirector

Trabajo Fin de Grado N.º
(a asignar por Secretaría)

Agradecimientos

Quisiera destacar que haber terminado esta etapa universitaria no es únicamente fruto del esfuerzo personal. Nada de esto habría sido posible sin el incondicional apoyo de mis padres, quienes siempre me animaron a perseguir aquello que me apasionaba, ni sin los ánimos constantes de mi pareja, que me ayudaron a creer en mí y a seguir adelante, ni sin la ayuda de los profesores y compañeros de la carrera con los que supere codo con codo los retos de la carrera.

Agradecido y orgulloso del hogar que me vio crecer, donde siempre se me impulsó a luchar por mis metas, sin importar que tan difícil fuera el reto. Nunca sentí que se dudara de mis capacidades, y eso me motivó a dar lo mejor de mí y a superar mis propios límites.

Y en los peores momentos, siempre estuvo mi pareja a mi lado, dándome ánimos y alegrándome los días más sombríos. Con su apoyo los malos momentos son más llevaderos y compartir alegrías se vuelve más reconfortante.

Gracias a mi madre, por su dedicación y cariño; Gracias a mi padre, por mantenernos firmes cuando todo parecía desmoronarse; Gracias a mi pareja por estar siempre ahí; Y gracias a toda mi familia por su unidad y fortaleza.

Al final de todo, parece que ha sido un trabajo en equipo.

Resumen

Este Trabajo de Fin de Grado se centra en la evaluación comparativa de dos modelos de detección de objetos de la familia YOLO, YOLOX y YOLOv8, aplicados a tareas de detección de objetos mediante visión artificial. El objetivo principal es analizar el rendimiento y la eficiencia de dichas redes convolucionales en entornos de cómputo con recursos limitados.

El estudio compara la implementación de los modelos en dos populares frameworks o entornos de desarrollo: PyTorch y TensorFlow. En dicha comparación se incluye un análisis del rendimiento de los modelos en cada entorno. Además, se analiza cómo el número de parámetros, tamaño de las imágenes y uso de cuantización INT8 afectan a la precisión de detección y a los tiempos de inferencia.

Como parte del estudio, se presenta una metodología de conversión de modelos de Pytorch a TensorFlow, resolviendo incompatibilidades estructurales mediante transformaciones específicas en los grafos de cómputo, al tiempo que se incluyen optimizaciones de operadores que facilitan la compatibilidad del modelo resultante con TensorFlow Lite.

Por último, se ha desarrollado una implementación funcional que permite verificar los modelos optimizados en una Raspberry Pi 4, equipada con una CPU ARMv7l de 32 bits, lo que ha permitido obtener resultados no solo de tiempo de ejecución, sino también métricas de precisión, sensibilidad y latencia. Este enfoque demuestra la viabilidad del uso de modelos avanzados de detección en escenarios de baja capacidad de computo, abriendo camino a futuras aplicaciones embebidas de visión artificial en tiempo real en el borde (“edge computing”).

Abstract

This proyect focuses on the comparative evaluation of two object detection models from the YOLO family: YOLOX and YOLOv8, applied to computer vision tasks for object detection. The main objective is to analyze the performance and efficiency of both architectures in computing environments with limited resources.

The study includes a comparison between the models using two popular development frameworks: PyTorch and TensorFlow, incorporating an analysis of performance differences within each environment. It also explores how the number of parameters, the input image size, and the use of INT8 quantization impact detection accuracy and inference time.

As part of the process, a model conversion methodology between frameworks is presented, addressing structural incompatibilities through custom graph transformations and operator optimization for compatibility with TensorFlow Lite.

The project concludes with the practical implementation and testing of the optimized models on a Raspberry Pi 4 equipped with a 32-bit ARMv7l CPU, evaluating accuracy, sensitivity, and latency metrics. This approach demonstrates the feasibility of deploying advanced detection models in low-power scenarios, paving the way for future real-time embedded computer vision applications.

Índice general

Agradecimientos **5**

Resumen **6**

Abstract **7**

Índice general **8**

Índice de figuras **10**

Índice de tablas **12**

Lista de acrónimos **13**

Capítulo 1. Introducción **14**

1.1. Motivación y contexto del proyecto 14
1.2. Objetivos generales 15
1.3. Estructura del documento 15

Capítulo 2. Estado del arte **17**

2.1. Fundamentos de la detección de objetos 17
2.1.1. Arquitectura general 18
2.1.2. Tipos de detectores 21
2.1.3. Métricas de evaluación 22
2.2. YOLOX: Arquitectura convolucional optimizada para velocidad 25
2.3. YOLOv8: Evolución de YOLOX con mejoras estructurales 27
2.4. Ultralytics: Framework comercial sobre PyTorch 29
2.5. Diferencias entre los frameworks PyTorch y TensorFlow-Lite 30
2.5.1. TensorFlow-lite y su ecosistema para dispositivos embebidos 31
2.5.2. Limitaciones de TorchScript y ONNX en sistemas al borde 31
2.6. Importancia y ventajas de la cuantización de modelos 33

Capítulo 3. Desarrollo y entrenamiento de los modelos **36**

3.1. Adaptación y preprocesado del conjunto de datos 38
3.1.1. Unificación de los conjuntos de datos 38
3.1.2. Reorganización por clases 39
3.1.3. Reparto equilibrado de instancias y formatos requeridos 40
3.1.3. Traslación de la división de imágenes entre formatos 42

3.2. Entrenamiento con YOLOv8 mediante Ultralytics 43
3.3. Entrenamiento con YOLOX desde repositorio oficial 47
3.4. Evaluación y comparación con el conjunto de prueba en PyTorch 50
Capítulo 4. Conversión y optimización del modelo a TensorFlow-Lite 54
4.1. Exportación de modelos a ONNX 54
4.2. Conversión a formato TensorFlow 54
4.3. Conversión a TensorFlow-Lite y cuantización INT8 56
4.3.1. Proceso de conversión directa para YOLOX 57
4.3.2. Proceso de conversión para YOLOv8 60
4.3.3. Conversión a sistema uf2 63
Capítulo 5. Despliegue en dispositivo: Raspberry Pi 4 66
5.1. Justificación del entorno y perfil hardware 66
5.2. Preparación del entorno de ejecución 66
5.3. Implementación del pipeline de inferencia en TensorFlow-Lite 67
5.3.1. Algoritmo de supresión de máximos (NMS) 68
5.3.2. Decodificación de YOLOX 68
5.3.3. Decodificación de YOLOv8 70
5.4. Evaluación del rendimiento en Raspberry Pi 4 70
5.4.1. Precisión en test 71
5.4.2. Tiempos de inferencia 72
Capítulo 6. Conclusiones y líneas futuras 76
6.1. Conclusiones generales 76
6.1.1. Efectos observados debidos a la conversión de Frameworks 76
6.1.2. Efectos observados debidos al número de parámetros 77
6.1.3. Efectos observados debidos al tamaño de entrada 77
6.2. Desafíos superados 78
6.3. Limitaciones encontradas 78
6.4. Líneas de investigación futuras 79
Bibliografía 81

Índice de figuras

Figura 1. Principales hitos en la evolución de los modelos de detección de objetos. 18
Figura 2. Componentes de un modelo general de detección de objetos. 19
Figura 3. Representación de anclas en un ejemplo de juguete. 21
Figura 4. Cantidad de detectores de objetos de la escena del arte, por categoría, publicados en revistas de alto impacto y evaluados sobre el conjunto de datos MS-COCO. 22
Figura 5. Representación y fórmula de la precisión y sensibilidad. 24
Figura 6. Ilustración sobre el concepto de IoU. 25
Figura 7. Esquema de cabeza desacoplada respecto a la cabeza acoplada en YOLOv3. 27
Figura 8. Comparación de modelos con el dataset COCO: mAP@[0.5:0.95] vs Latencia. 28
Figura 9. Distribución original del conjunto de datos. 39
Figura 10. Ilustración de las divisiones por subconjuntos de los datos de entrenamiento. 41
Figura 11. Segunda distribución del conjunto de datos. 41
Figura 12. Distribución final empleada del conjunto de datos. 42
Figura 13. Ejemplo de interpolación por vecino más cercano. 43
Figura 14. Resultados de precisión y sensibilidad durante el entrenamiento de YOLOv8. 44
Figura 15. Evolución de los valores de pérdida durante el entrenamiento del modelo mediano. 45
Figura 16. Gráficas de los resultados durante el entrenamiento del modelo mediano en 50 épocas. 46
Figura 17. Comparación de las métricas finales en el entrenamiento entre modelos de YOLOv8. 47
Figura 18. Resultados del entrenamiento del modelo mediano de YOLOX. 48
Figura 19. Valores de precisión y sensibilidad obtenidos en el final del entrenamiento de los modelos de YOLOX. 49
Figura 20. Comparación de las métricas finales en el entrenamiento 49

Figura 21. Comparación de los resultados de los entrenamientos de YOLOv8 y YOLOX. 50
Figura 22. Valores de precisión y sensibilidad obtenidos en el conjunto de prueba por los modelos de YOLOv8. 51
Figura 23. Resultados con el conjunto de prueba con YOLOv8. 51
Figura 24. Valores de precisión y sensibilidad obtenidos en el conjunto de prueba por los modelos de YOLOX. 52
Figura 25. Resultados con el conjunto de prueba con YOLOX. 52
Figura 26. Esquema secuencial traslación entre de frameworks. 54
Figura 27. Distribución del dataset representativo. 57
Figura 28. Comparación del factor de compresión entre archivos de YOLOX. 58
Figura 29. Comparación entre los tamaños de archivos de ONNX y TensorFlow para YOLOv8. 59
Figura 30. Comparación del factor de compresión entre archivos ONNX y TF de YOLOX. 59
Figura 31. Comparación del factor de compresión entre archivos PyTorch y TensorFlow de YOLOv8. 63
Figura 32. Comparación del incremento de tamaño del archivo uf2 respecto a los archivos en tflite en YOLOv8. 65
Figura 33. Comparación del incremento de tamaño del archivo uf2 respecto a los archivos en tflite en YOLOX. 65
Figura 34. Resultados obtenidos en TFLite 72
Figura 35. Comparación entre predicciones de los modelos medianos de YOLOv8 y YOLOX. Cuadro azul: Caja real | Cuadro verde: Caja predicha. 73
Figura 36. Evolución de los fotogramas por segundo en función del tamaño de entrada en los modelos más ligeros. 74
Figura 37. Comparación de la estimación de tiempos de inferencia por Edge-Impulse. 74

Índice de tablas

Tabla 1. Comparación de modelos por su número de parámetros. 36
Tabla 2. Valores de precisión y sensibilidad obtenidos en el final del entrenamiento de los modelos de YOLOv8. 46
Tabla 3. Comparación entre los tamaños de archivos de PyTorch y TensorFlow para YOLOX. 58
Tabla 4. Comparación entre los tamaños de archivos de PyTorch y TensorFlow para YOLOv8. 62
Tabla 5. Comparación del tamaño en kilobytes del archivo uf2 para cada modelo. 64
Tabla 6. Relación entre stride, tamaño de celda y número de celdas en la decodificación de YOLOX. 69
Tabla 7. Resultados con el conjunto de prueba en la RaspberryPi 4 con YOLOv8. 71
Tabla 8. Resultados con el conjunto de prueba en la RaspberryPi 4 con YOLOX. 71
Tabla 9. Comparación de los tiempos de inferencia por imagen para cada modelo..... 73

Lista de acrónimos

AI	Artificial Intelligence
AP	Average Precision
CNN	Convolutional Neuronal Network
COCO	Common Objects in Context
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FPN	Feature Pyramid Network
FPU	Floating Point Unit
GPU	Graphics Processing Unit
IoU	Intersection Over Union
mAP	Mean Average Precision
NMS	Non Maximum Supressor
NPU	Neural Processing Unit
R-CNN	Region-based Convolutional Neural Network
SDK	Software Development Kit
TF	TensorFlow
TFG	Trabajo de Fin de Grado
TFLite	TensorFlow-Lite
TPU	Tensor Processing Unit
YOLO	You Only Look Once

Capítulo 1. Introducción

1.1. Motivación y contexto del proyecto

Durante los últimos años, la influencia de la inteligencia artificial (AI) se ha hecho cada vez mayor, no solo en el ámbito industrial, académico y de investigación, sino también en la vida cotidiana de los consumidores [1], [2]. Este aumento de interés ha generado avances significativos en diversas ramas, siendo la visión por ordenador una de las más activas y con mayor potencial futuro, gracias a sus múltiples aplicaciones, que van desde sectores como la ganadería y el control de tráfico o aforos hasta la videovigilancia o incluso la medicina [3].

Este proyecto nace por la escasez de modelos de visión por computador que, siendo precisos, sean accesibles y fácilmente desplegables en dispositivos con recursos limitados. Por ello, la motivación principal de este Trabajo de Fin de Grado es identificar, implementar y evaluar modelos de inteligencia artificial que proporcionen la mejor relación entre tamaño y precisión.

Además, se explora el nexo entre los dos entornos de desarrollo más utilizados en el ámbito del aprendizaje profundo: PyTorch y TensorFlow. Mientras que el primero es más empleado en el entorno académico por su sintaxis intuitiva y flexibilidad, el segundo está más implantado en la industria, principalmente por su estabilidad, escalabilidad y capacidad de optimización para entornos con restricciones de hardware [4].

En definitiva, en este proyecto, se abordan técnicas de entrenamiento, cuantización y despliegue de modelos para visión con computador con una metodología directa y replicable. El objetivo principal es generar modelos más ligeros y eficientes, facilitando así su uso en una amplia gama de dispositivos con bajos recursos hardware y coste, lo cual tiene interés tanto para la empresa como para la investigación académica.

1.2. Objetivos generales

El objetivo principal de este proyecto es desarrollar y evaluar un sistema de visión por ordenador para detección de objetos, optimizado la relación entre el tamaño del modelo y su precisión, con el fin de permitir su implementación en dispositivos con recursos computacionales limitados. Este estudio puede permitir despliegues menos costosos sin sacrificar la precisión del modelo, en contextos donde los bajos costes de hardware pueden ser un requisito esencial , como es normalmente el caso de explotaciones ganaderas o sistemas de videovigilancia domésticos.

Para alcanzar dicho objetivo principal, se han llevado a cabo las siguientes actividades:

- Analizar y comparar modelos de detección de objetos accesibles como código abierto.
- Implementar la conversión de los modelos seleccionados a formatos con implementación hardware más eficiente, como TFLite.
- Evaluar el rendimiento de los modelos generados en el punto anterior.
- Establecer una metodología replicable que permita a otros usuarios implementar soluciones similares con conocimientos técnicos moderados y equipos de bajo coste.

1.3. Estructura del documento

Este Trabajo de Fin de Grado se divide en seis capítulos, siendo esta introducción el primero de ellos.

El segundo capítulo explora el estado del arte relacionado con el proyecto, sirviendo como base para justificar las decisiones tomadas.

El tercer capítulo detalla el proceso de entrenamiento de modelos de aprendizaje profundo empleado para obtener una primera versión funcional.

El cuarto capítulo se centra en la conversión y cuantización de los modelos previamente entrenados.

El quinto capítulo aborda la implementación en un dispositivo de prototipado de sistemas embebidos, representativo de un entorno hardware con recursos limitados.

El sexto y último capítulo presenta las conclusiones del proyecto, así como las limitaciones encontradas y posibles líneas de mejora futura.

Capítulo 2. Estado del arte

La visión por ordenador es una de las ramas con más aplicaciones de la inteligencia artificial [1], cuyo objetivo principal es dotar a las máquinas de la capacidad de interpretar imágenes o secuencias de vídeo del mundo real [3].

Esta disciplina ha experimentado un desarrollo acelerado en la última década, impulsada principalmente por el avance de las redes neuronales convolucionales (CNN) y la disponibilidad de grandes conjuntos de datos anotados (como COCO). Gracias a estos progresos, hoy en día existen sistemas capaces de realizar tareas complejas como la detección de objetos o el reconocimiento facial, con niveles de precisión cercanos al rendimiento humano.

2.1. Fundamentos de la detección de objetos

A grandes rasgos, la detección de objetos mediante visión por computador es un problema que se basa en la extracción de características visuales de una imagen para resolver una tarea de regresión y/o clasificación, con el fin de generar una caja delimitadora (bounding box) que encuadre la instancia del objetivo dentro de la imagen [5].

Desde sus inicios, la detección de objetos ha representado un reto significativo para la comunidad científica, debido a la complejidad que supone localizar múltiples objetos de diferentes clases, tamaños y posiciones bajo condiciones variables de iluminación, ángulos de captura o fondo. A lo largo de los años, se han propuesto numerosos enfoques para superar estas limitaciones, marcando hitos clave en la evolución del campo [2].

Una de estos hitos es la arquitectura de detección en una sola etapa (como por ejemplo, YOLO iniciado en 2015 [2], [6]) que constituye un avance frente a los detectores de dos etapas, como la familia R-CNN iniciada en 2014 [2].

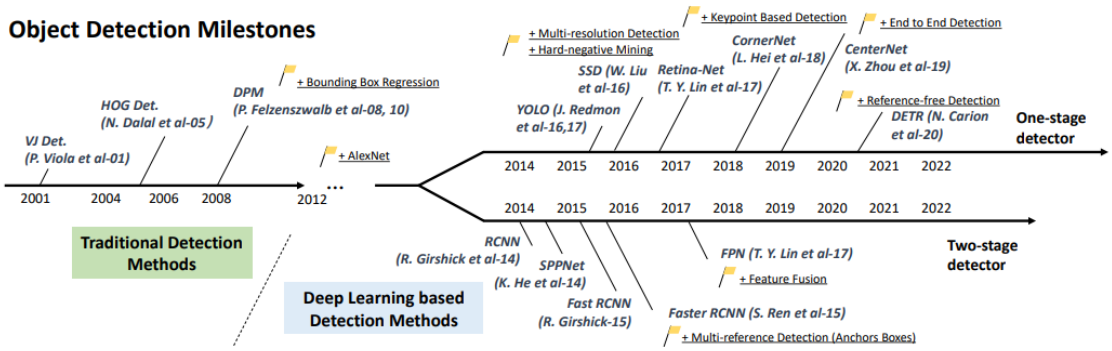


Figura 1. *Principales hitos en la evolución de los modelos de detección de objetos.*

Fuente: [2]

Mientras que los primeros ofrecen una mayor velocidad de inferencia, al combinar localización y clasificación en una sola red, los segundos suelen alcanzar una mayor precisión, al separar ambos procesos [2].

Además, en la actualidad pueden distinguirse tres tipos de detectores: los basados en anclas (anchor-based), como YOLOv4 o RetinaNet; los sin anclas (anchor-free), como YOLOv8 o YOLOX; y los basados en transformers, como la familia DETR [5].

Cada uno de estos enfoques presenta ventajas y desafíos específicos en términos de precisión, eficiencia computacional y complejidad de entrenamiento.

2.1.1. Arquitectura general

La arquitectura moderna de los modelos de visión por ordenador suele organizarse en cuatro etapas clave: la entrada (input), la columna vertebral (backbone), el cuello (neck) y la cabeza (head).

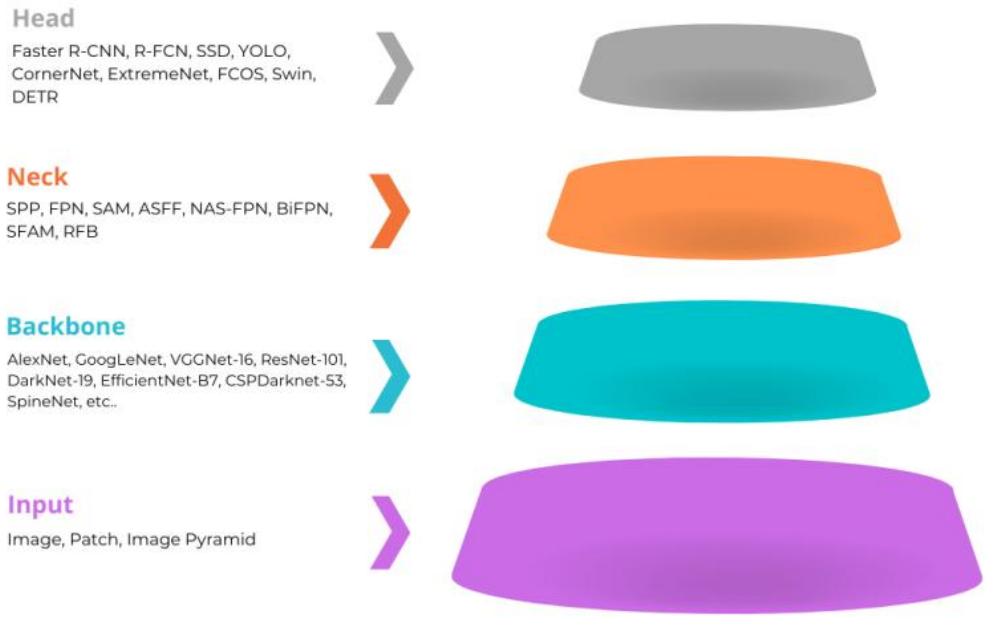


Figura 2. *Componentes de un modelo general de detección de objetos.*

Fuente: [5]

Entrada (input): La entrada del modelo está formada generalmente por una imagen única, aunque en algunos casos puede tratarse de una pequeña región de una imagen más grande, llamada parche o incluso una pirámide de imágenes a diferentes escalas, lo cual permite mejorar la detección de objetos de distinto tamaño.

La imagen de entrada se normaliza y ajusta a un tamaño predefinido antes de ser procesada por las capas siguientes. [5]

- Columna vertebral (Backbone):

El backbone es una red neuronal convolucional cuya función principal es extraer las características visuales más relevantes de la imagen. Actúa como base del modelo, generando representaciones intermedias que capturan la información estructural, de textura y de forma del contenido visual. La elección del backbone afecta sustancialmente al equilibrio deseado entre rendimiento y coste computacional [5].

YOLOX emplea como backbone una versión mejorada de CSPDarknet, concretamente CSPDarknet53 v5 [7], que mejora la eficiencia computacional. Por su parte, YOLOv8 introduce un backbone propio, optimizado por Ultralytics, que también deriva de CSPDarknet53, pero ha sido modificado y simplificado para adaptarse mejor a tareas de inferencia con baja latencia [8], no obstante, esta versión modificada no cuenta con un nombre propio.

- Cuello (Neck):

El cuello se sitúa sobre el backbone y está compuesto por rutas de procesamiento hacia adelante (downstream) y hacia atrás (upstream), que permiten fusionar características a diferentes niveles de resolución. Su objetivo es mejorar la capacidad del modelo para detectar objetos de diferentes tamaños y contextos [7].

Una estructura de neck común es la FPN (Feature Pyramid Network) [6], que resulta especialmente relevante en este caso, ya que fue uno de los primeros cambios sustanciales introducidos por YOLOX respecto a su predecesor, YOLOv3 [7]. La FPN combina características a distintas escalas utilizando una arquitectura de tipo piramidal con conexiones de tipo top-down y lateral, lo que permite mejorar la detección de objetos de diferentes tamaños [6]. YOLOX reemplaza esta estructura por un PAFPN (Path Aggregation FPN), inspirada en PANet, empleada por YOLOv4 y YOLOv5, que introduce operaciones puramente convolucionales y mejora la fusión de características intermedias, optimizando así tanto la precisión como la velocidad de inferencia [7].

- Cabeza (Head):

Por último, la cabeza es la parte encargada de realizar las predicciones finales. En ella se generan las cajas delimitadoras, las clases de los objetos detectados y sus puntuaciones de confianza. Estas cabezas se pueden clasificar en dos tipos: las de predicción densa, que realizan detecciones en múltiples puntos del espacio, y las de predicción dispersa que se centran en un número más reducido de regiones candidatas como Faster R-CNN [5].

2.1.2. Tipos de detectores

En función del tipo de predicción que realiza la etapa final (la cabeza del modelo), los detectores actuales pueden clasificarse principalmente en tres enfoques: anchor-based, anchor-free y basados en Transformers [5]:

- Basado en anclas (anchor-based):

Este enfoque fue dominante durante muchos años y consiste en predefinir un conjunto de cajas ancla (*anchor boxes*) de diferentes tamaños y proporciones que se superponen a la imagen para que durante el entrenamiento el modelo aprenda a ajustar estas anclas para encajar con los objetos reales [5], sirviendo como propuestas de detección. Este método, presente en detectores como YOLOv3 o Faster R-CNN, requiere una cuidadosa selección de anclas y puede ser computacionalmente costoso, especialmente en escenarios con objetos muy variados.

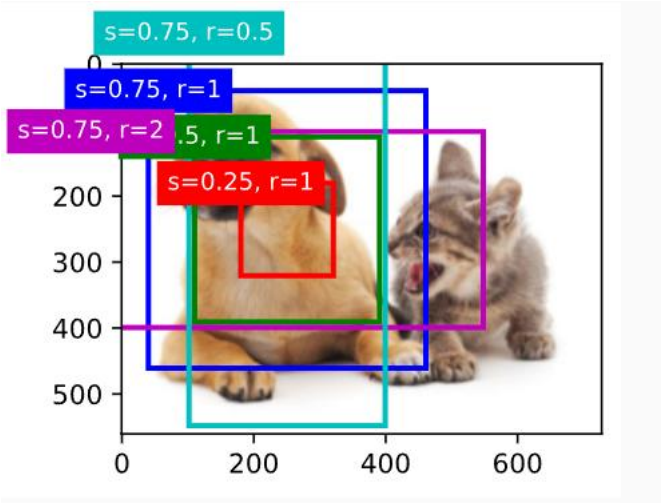


Figura 3. *Representación de anclas en un ejemplo de juguete.*

Fuente: [9]

- Libre de anclas (anchor-free):

Este nuevo enfoque elimina el uso de anclas predefinidas. En lugar de partir de cajas base, el modelo predice directamente la posición y el tamaño de los objetos desde puntos clave del mapa de características, obtenido en el cuello de la red.

Esto simplifica el diseño del modelo, reduce el número de hiperparámetros y mejora la generalización en conjuntos de datos complejos [5].

-Basado en Transformers:

Algunos modelos recientes, como DETR, utilizan arquitecturas basadas en Transformers para modelar las relaciones espaciales entre objetos [5]. Aunque prometedores, estos enfoques requieren gran capacidad computacional, gran número de datos y aún no superan en eficiencia a los métodos convencionales en tareas de detección en tiempo real. No obstante, los modelos basados en Transformers representan una nueva tendencia dentro de la investigación en visión artificial [5].

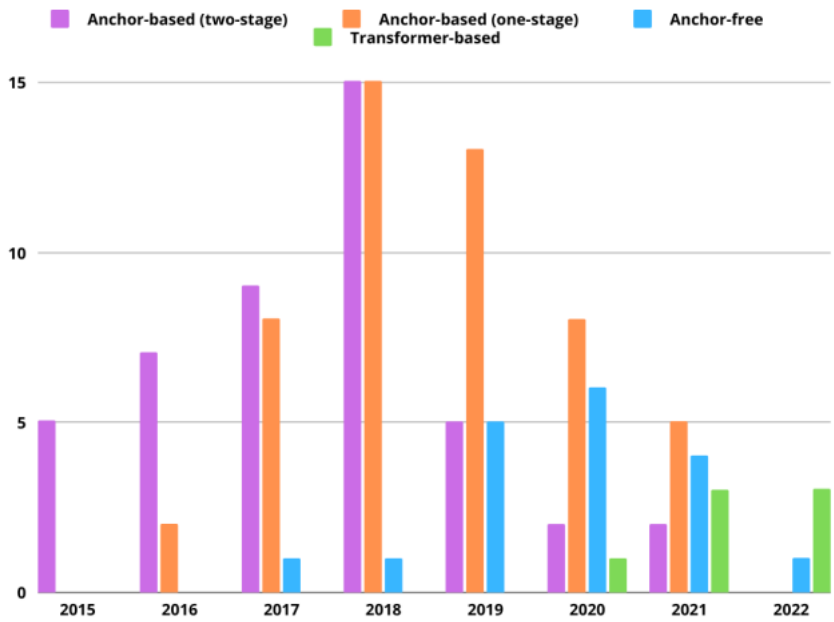


Figura 4. *Cantidad de detectores de objetos de la escena del arte, por categoría, publicados en revistas de alto impacto y evaluados sobre el conjunto de datos MS-COCO.*

Fuente: [5]

2.1.3. Métricas de evaluación

Para evaluar la calidad y precisión de las predicciones de este tipo de modelos, se emplean métricas estandarizadas, que permiten comparar resultados de forma objetiva. Este es el caso de la evaluación mediante las métricas definidas

en el conjunto de datos COCO, que se ha convertido en un estándar ampliamente empleado en el campo.

Estas métricas se basan principalmente en dos factores:

1. La precisión en la identificación de la clase del objeto.
2. El grado de solapamiento entre la caja delimitadora predicha y la anotación real.

El valor más representativo al evaluar modelos de detección de objetos es la precisión media (mean Average Precision, mAP) que cuantifica tanto la precisión como la exhaustividad del modelo. Esta métrica se calcula como el promedio de los valores de precisión promedio (Average Precision, AP) obtenidos para cada clase, considerando la coincidencia entre las cajas delimitadoras predichas y las cajas anotadas en el conjunto de prueba. Una mayor mAP indica una mejor capacidad del modelo para detectar correctamente los objetos y localizarlos con exactitud. Calculada según la ecuación (1), dónde N es el número de clases.

$$mAP = \frac{1}{N}\sum_{i=1}^{N} AP_i$$

Ecuación 1. Fórmula general de la mAP. Fuente: [10]

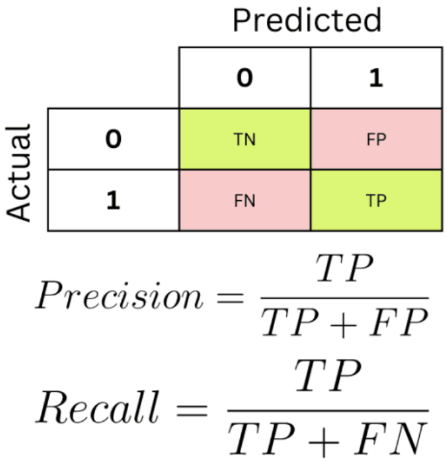
El AP se calcula como el área bajo la curva de precisión (P) frente a la sensibilidad (recall, R). En su forma discreta más común, se calcula según la ecuación (2).

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

Ecuación 2. Fórmula general de la AP. Fuente: [10]

La curva de precisión frente a la sensibilidad para cada clase representa la relación entre la capacidad del modelo para detectar todos los objetos (sensibilidad) y la capacidad para evitar falsos positivos (precisión). El área bajo esta curva proporciona el valor de AP para esa clase. Finalmente, la mAP se

obtiene como el promedio de los valores AP de todas las clases, ofreciendo una medida global del desempeño del modelo en la tarea de detección de objetos.



$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Figura 5. *Representación y fórmula de la precisión y sensibilidad.*

Fuente: [11]

Para determinar si una predicción se considera un verdadero positivo (TP) o un falso positivo (FP), se compara la caja predicha con la caja real, utilizando el Índice de Solapamiento (IoU, Intersection Over Union). Este índice cuantifica el grado de coincidencia entre ambas cajas: si el IoU supera un umbral predefinido (por ejemplo, 0.45) y la confianza de la predicción es superior a un valor mínimo, entonces la predicción se clasifica como un TP; en caso contrario, como FP.

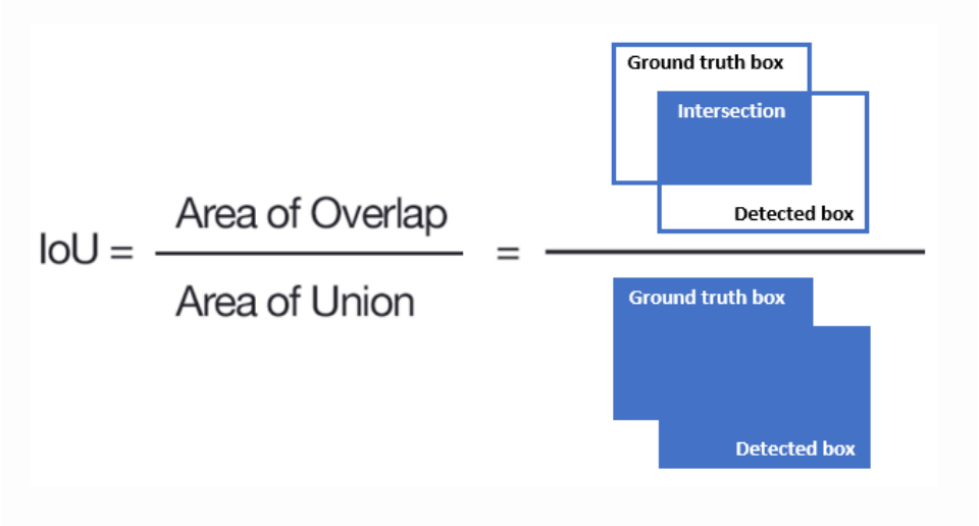


Figura 6. *Ilustración sobre el concepto de IoU.*

Fuente: [12]

Esta metodología es especialmente útil para evaluar modelos de detección en escenarios con múltiples clases, variedad de tamaños y niveles de dificultad, permitiendo así un análisis detallado y riguroso del comportamiento del sistema.

2.2. YOLOX: Arquitectura convolucional optimizada para velocidad

La familia de modelos YOLO surgió como una alternativa a los enfoques de detección de objetos basados en la aplicación reiterativa de algoritmos sobre la imagen, donde se evaluaban múltiples regiones o propuestas hasta generar la predicción final [5].

Además de ser pionera en fusionar en una única red las etapas de localización y clasificación, esta familia de modelos destaca por estar enfocada a la reducción de la latencia, lo que la convierte en una opción importante en aplicaciones de tiempo real.

YOLOX surge como una evolución dentro de esta familia [7], proponiendo una arquitectura aún más optimizada para reducir el tiempo de computo. Esto se consigue mediante la sustitución del sistema de detección basado en anclas por

un enfoque anchor-free más eficiente a la vez quese reemplaza el algoritmo piramidal para la extracción de características, por uno convolucional [7].

Las predicciones basadas en anclas, aunque efectivas, implican un elevado número de operaciones y requiere un cuidadoso ajuste de hiperparámetros, como el tamaño y número de anclas por escala. Por ello, al eliminar el uso de anclas mediante una estrategia anchor-free, YOLOX logra un aumento considerable en la velocidad de inferencia sin comprometer la precisión [7].

El modelo, de código abierto y propuesto por la empresa Megvii, introduce además una modificación clave en la arquitectura de la cabeza (head), apostando por el desacoplamiento de las tareas de clasificación y regresión [7].

YOLOX introduce el concepto de cabeza desacoplada, una innovación respecto a versiones anteriores como YOLOv5 [7]. En las versiones previas, una única rama de la red se encargaba simultáneamente de predecir las cajas, las clases y la puntuación de confianza. En cambio, la cabeza desacoplada separa explícitamente estas tareas en distintas ramas, permitiendo que la red aprenda mejor cada una por separado y con menos interferencia entre ellas [7]. A diferencia de los detectores de dos etapas, como Faster R-CNN, donde la separación ocurre entre dos fases completamente distintas (una para generar propuestas de regiones y otra para clasificarlas), en el enfoque de YOLOX esta separación ocurre dentro de una única etapa y de forma paralela, manteniendo así la rapidez de los detectores monofásicos (one-stage detectors) pero con una mayor especialización en las salidas [7].

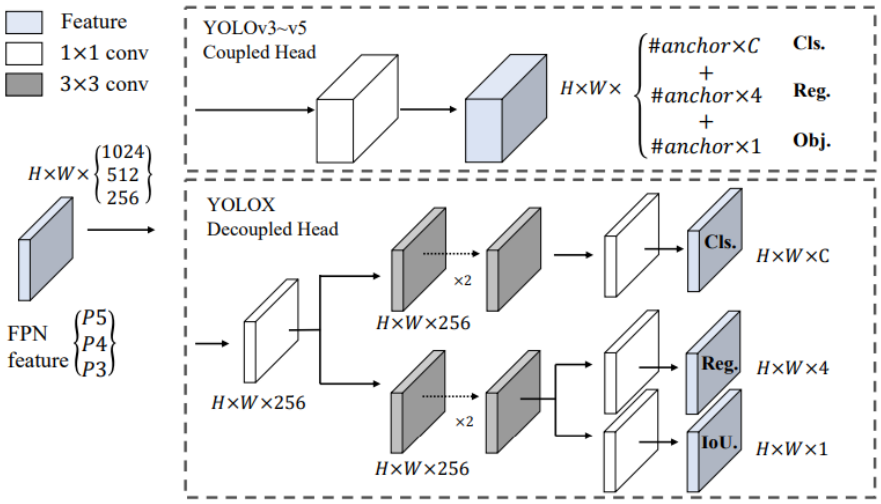


Figura 7. *Esquema de cabeza desacoplada respecto a la cabeza acoplada en YOLOv3.*

Fuente: [7]

Actualmente, plataformas como EdgeImpulse incluyen YOLOX entre los modelos soportados para despliegue en sistemas embebidos, gracias a su buen equilibrio entre precisión, tamaño y velocidad. EdgeImpulse es un entorno en la nube orientado al desarrollo de aplicaciones de AI en el borde (edge computing), utilizando plataformas hardware con recursos limitados.

Por todos estos motivos, YOLOX fue escogido como modelo base de referencia para este proyecto. Su bajo tamaño, velocidad, código abierto, respaldo académico y amplia adopción en entornos con recursos limitados lo convierten en un candidato ideal para realizar una comparación técnica con modelos más recientes, con menor madurez investigadora pero gran potencial como su contraparte en este estudio, YOLOv8.

2.3. YOLOv8: Evolución de YOLOX con mejoras estructurales

La empresa Ultralytics desarrolló YOLOv8 como modelo basado en detecciones sin ancla orientado a plataformas con pocos recursos hardware.

No obstante, a diferencia de YOLOX, la compañía no ha publicado ningún artículo detallando la arquitectura del modelo, limitándose a compartir sus ventajas, pesos preentrenados y los resultados obtenidos en el conjunto de datos COCO [13].

A pesar de esta falta de documentación técnica formal sobre el modelo, Ultralytics ofrece una API sencilla y accesible a nivel de usuario, que facilita tanto el entrenamiento como la evaluación e implementación del modelo en distintos entornos. Esta facilidad de uso, en la que se basa la empresa su propuesta, ha contribuido a su adopción dentro de la comunidad, especialmente en entornos de prototipado rápido y desarrollo orientado a plataformas embebidas.

La elección de este modelo como alternativa a YOLOX se basa en su similitud estructural y filosofía de funcionamiento. Sin embargo, mientras que muchos competidores de YOLOX (como es el caso de NanoDet) centran su propuesta en competir en latencia (tiempo de ejecución) a igualdad de precisión y tamaño, YOLOv8 apuesta por mejorar el rendimiento manteniendo un tamaño reducido. Esta diferencia en enfoque resulta de especial interés para este estudio, centrado en optimizar la relación entre precisión y tamaño.

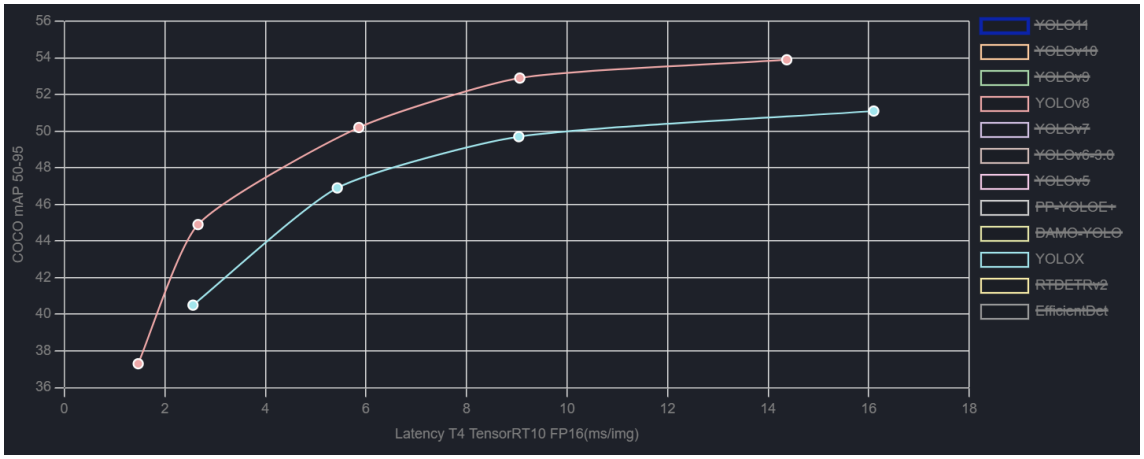


***Figura 8**. Comparación de modelos con el dataset COCO: mAP@[0.5:0.95] vs Latencia.*

Curva roja: YOLOv8 | Curva azul: YOLOX.

Fuente: [14]

En la figura 8 se muestra, en rojo, el rendimiento de YOLOv8 y en azul claro YOLOX. La gráfica compara ambos modelos en términos de precisión media frente a la latencia obtenida en una GPU NVIDIA T4, usando TensorRT con una precisión de flotantes de 16 bits. Se observa que YOLOv8 alcanza, en la mayoría de sus versiones, un mayor mAP con una latencia inferior, posicionándose como una alternativa más eficiente tanto en precisión como en velocidad. En cambio, YOLOX muestra un rendimiento algo inferior en términos de precisión, aunque sigue siendo competitivo respecto a la latencia, en donde no se aprecia una gran diferencia en este relevo generacional.

2.4. Ultralytics: Framework comercial sobre PyTorch

El término "*framework*" hace referencia a una estructura o conjunto de herramientas y componentes predefinidos que proporcionan una base para el desarrollo de aplicaciones de software [15] .En este contexto, Ultralytics ofrece un entorno accesible para el desarrollo de modelos de visión por computador tanto como para la tarea de detección de objetos, como para la tarea de clasificación o incluso de identificación de pose.

Sin embargo, pese a contar con un repositorio público y una documentación accesible, Ultralytics no opera como un proyecto de código abierto en sentido estricto, sino como una empresa privada que gestiona sus modelos y herramientas bajo distintas licencias. Su licencia AGPL-3.0 permite el uso gratuito únicamente en contextos personales, educativos o de investigación, con la condición de que cualquier proyecto derivado también deba ser abierto. Para usos comerciales o distribución privada, se requiere una licencia específica, como la Enterprise o la Academic [16].

En este trabajo se evalúa la viabilidad de los modelos de Ultralytics, recurriendo exclusivamente a los recursos públicos disponibles, sin modificar el código fuente ni realizar ningún tipo de distribución con fines comerciales.

Por estos motivos, resulta especialmente relevante distinguir entre los frameworks públicos y de código abierto, como PyTorch, ONNX o TensorFlow, y plataformas comerciales como Ultralytics que, aunque utilicen herramientas de acceso libre como base, funcionan como entornos de desarrollo propios en la nube o empaquetados con una licencia de uso restringido.

2.5. Diferencias entre los frameworks PyTorch y TensorFlow-Lite

En el estado del arte de la investigación en modelos de inteligencia artificial, los frameworks PyTorch y TensorFlow están presentes en la gran mayoría de artículos científicos y proyectos técnicos [4], debido a la potencia y versatilidad que ofrecen para tareas de entrenamiento, evaluación e implementación de redes neuronales profundas.

TensorFlow fue desarrollado por Google y lanzado en 2015 como sucesor del entorno DistBelief de 2011 [17]. Por otro lado, PyTorch fue publicado por Facebook AI Research (FAIR) en 2016 [18]. Actualmente, PyTorch es el framework más utilizado en entornos de investigación y prototipado rápido, gracias a su sintaxis más intuitiva, su uso de gráficos computacionales dinámicos y su facilidad para depurar modelos paso a paso [4], lo que lo hace especialmente accesible para experimentar con arquitecturas novedosas.

Por otro lado, TensorFlow es más empleado en aplicaciones industriales y comerciales gracias a su capacidad para integrarse directamente con hardware de bajo nivel [4], como procesadores ARM, GPUs móviles o aceleradores especializados (TPU).

Por ello existe un gran interés en encontrar metodologías que permitan la conversión de modelos entre frameworks, combinando así la accesibilidad de desarrollo de PyTorch con la implementación de TensorFlow Lite.

De esta necesidad surge ONNX (Open Neural Network Exchange), un proyecto comunitario lanzado en 2017 por Facebook y Microsoft [19]. ONNX define un formato abierto para representar modelos de redes neuronales, actuando como un descriptor de la estructura de la red [20] para exportar modelos desde un entorno y adaptarlos a otro, para ser interpretados por motores de inferencia compatibles, como TensorFlow Lite. Sin embargo, no siempre es posible la conversión a ONNX, habitualmente debido a la existencia en el modelo de operaciones específicas que no tienen traducción directa a ONNX.

2.5.1. TensorFlow-lite y su ecosistema para dispositivos embebidos

Dentro de TensorFlow existe la versión Lite, que como su nombre indica, es una variante ligera del entorno principal.

Lanzada en 2019, su principal objetivo es convertir los modelos de TensorFlow a modelos compatibles con dispositivos de capacidades reducidas, como móviles o hasta sistemas embebidos [21].

Esta versión está diseñada para reducir significativamente tanto el tamaño del modelo como su latencia durante la inferencia, manteniendo una precisión aceptable.

Para ello, TensorFlow Lite ofrece herramientas específicas como una herramienta propia convertidora y un proceso de cuantización dinámico lineal por rango de activaciones en función de los resultados de exponer un conjunto de datos (dataset) representativo al modelo [22].

2.5.2. Limitaciones de TorchScript y ONNX en sistemas al borde

La principal ventaja de TensorFlow Lite es la posibilidad de generar modelos en lenguajes de bajo nivel, como C++, lo que permite una inferencia más eficiente y adaptable a dispositivos con pocos recursos [23]. Esta integración directa

facilita también la compatibilidad con microcontroladores, sistemas embebidos, y la aceleración mediante dispositivos específicos (GPU, TPU, etc.).

En cambio, PyTorch, a pesar de su popularidad, presenta limitaciones en el proceso de despliegue. Para realizar inferencias fuera del entorno Python, se requiere convertir los modelos a TorchScript, una representación intermedia de PyTorch que puede ejecutarse en C++ [24]. No obstante, TorchScript suele generar modelos pesados y menos optimizados para ser ejecutados en plataformas con recursos limitados.

Además, su ecosistema carece de una infraestructura madura para el despliegue en dispositivos móviles o microcontroladores, lo que complica su integración en proyectos industriales o embebidos.

Por otro lado, aunque el formato ONNX permite la exportación desde PyTorch (e incluso desde TensorFlow) para ejecutar modelos en entornos multiplataforma [25], su soporte nativo no incluye, por defecto, una implementación optimizada orientada a hardware embebido.
Por ello, ONNX Runtime (su entorno de ejecución) no está específicamente adaptado para dispositivos como microcontroladores o NPUs de bajo consumo y, en muchos casos, necesita de compiladores o toolkits adicionales (como ONNX Runtime Mobile), que añaden complejidad técnica y no siempre alcanzan la eficiencia de TFLite.

En resumen, aunque TorchScript y ONNX son opciones válidas para el despliegue de los modelos, TFLite destaca por su enfoque específico hacia la eficiencia en sistemas en el borde (Edge), ofreciendo un flujo de trabajo más directo y sencillo, además de proporcionar mejor soporte a nivel de hardware.

2.6. Importancia y ventajas de la cuantización de modelos

El uso de microcontroladores y sistemas en el borde en aplicaciones de AI requiere de modelos que proporcionen las mejores prestaciones con la menor cantidad de memoria posible. Esto se traduce en productos con menor coste económico pero que mantienen un buen rendimiento.

Al cuantizar los pesos de las redes neuronales, pasando de valores reales a enteros, se alcanzan dos objetivos principales:

1. Reducción de tamaño: Al pasar de números flotantes de 32 bits a enteros de 8, se logra una reducción en un factor de 4. Es decir, cada nuevo peso cuantizado ocupa la cuarta parte del espacio de memoria que el peso sin cuantizar. En la práctica no se obtiene una reducción a un cuarto del espacio en memoria puesto que para el proceso de descuantización es necesario añadir un overhead (cabecera) con datos de la cuantización, como el punto del cero y el factor de escala. No obstante, la reducción de tamaño sigue siendo muy importante.

2. Se elimina la necesidad de procesamiento en punto flotante, que requiere de hardware (FPUs) o software específico, durante el proceso de inferencia del modelo. El formato de enteros de 8 bits es típicamente soportado hasta por el hardware más simple, por lo que es posible trasladar el proceso de inferencia prácticamente cualquier plataforma, siendo muy probable que esta incluya aceleradores hardware para este formato, lo que implica aún más eficiencia.

TensorFlow Lite realiza una cuantización post-entrenamiento dinámica [22]. Con esta técnica convierte los parámetros del modelo a valores enteros de 8 bits. Este proceso reduce el tamaño del modelo, disminuye el uso de memoria y acelera la inferencia al evitar operaciones en punto flotante, lo cual es crucial en

sistemas sin coprocesador de coma flotante, a cambio de una pérdida moderada de precisión por cuantización.

El procedimiento de cuantización se basa en aplicar una transformación lineal, que relaciona los valores reales “r” con sus equivalentes cuantizados “q” mediante la ecuación (3) [26].

$$r = s(q - z)$$

Ecuación 3. Ecuación de cuantización lineal. Fuente: [26]

Donde “s” es la escala de cuantización en punto flotante, y “Z” es el punto cero (un valor entero que representa el cero real en el rango cuantizado). Estos valores de escala y punto cero se obtienen mediante un análisis de rangos de activación de la red neuronal, cuando se utiliza un conjunto de datos representativo. De esta manera se consigue una representación que permite aproximar los pesos y activaciones originales con solo tres elementos: una escala “s”, un punto cero “Z” y los valores “q” codificados en 8 bits.

Como el valor de la escala y del punto cero son números reales que se utilizan para realizar la descuantización, y con el objetivo de eliminar completamente las operaciones en punto flotante, TensorFlow Lite representa el valor de s como un producto entre un multiplicador entero “M_0” y un desplazamiento de bits (2^{-n}).

$$s = M_0 2^{-n}$$
$$log_2(s) = log_2(M_0) + log_2(2^{-n})$$
$$M_0 = int(2^{log_2(s)+n})$$

Ecuación 4. Cómputo de M0. Fuente: [26]

Así, en lugar de multiplicar por un número real, la operación a realizar durante la inferencia se reduce a una multiplicación entera seguida de un desplazamiento de bits, opción mucho más eficiente [26]. Este método mantiene la precisión dentro de márgenes aceptables, con una drástica reducción del uso de memoria

y sin necesidad de soporte de punto flotante en el hardware de la plataforma de implementación.

Capítulo 3. Desarrollo y entrenamiento de los modelos

En este trabajo se entrenaron las tres versiones con menores parámetros de YOLOv8 (modelos mediano, pequeño y nano) y las tres versiones equivalentes de YOLOX (modelos mediano, pequeño y diminuto o “tiny”). Se puede observar que el modelo “nano” de YOLOv8 equivale al “Tiny” de YOLOX.

Número de parámetros (10^6)	Mediano	Pequeño	Tiny/nano
YOLOX	25.3	9.0	5.06
YOLOV8	25.9	11.2	3.2

Tabla 1. *Comparación de modelos por su número de parámetros.*

Fuentes: [13], [27]

En el trabajo realizado se ha profundizado en el análisis de los modelos más reducidos, mediante un entrenamiento con imágenes de diferentes resoluciones, para observar cómo afecta el tamaño de entrada al rendimiento, tanto en latencia (velocidad) como en precisión.

Los tamaños de entrada seleccionados son múltiplos de 32, en coherencia con el kernel interno de YOLOX de 32x32 [7]. Para ello se utilizaron tamaños cuadrados de 640, 416 y 192 píxeles. La elección se justifica de la siguiente manera:

- El tamaño de 640x640 pixeles es un estándar habitual en visión artificial.
- La imagen de 192x192 pixeles coincide con la resolución de una cámara del laboratorio de Ingeniería Microelectrónica, orientada a inteligencia artificial.
- El formato de 416x416 pixeles es el punto medio entre ambas resoluciones.

Por ello, el estudio se organiza en cuatro niveles. En primer lugar, las diferencias entre modelos exportados con PyTorch y TFLite. En segundo lugar, la comparación entre YOLOX y el más reciente YOLOv8. En tercer lugar, las diferencias entre variantes de un mismo modelo en función del número de parámetros y, por último, el impacto del tamaño de entrada (cuarto nivel).

Una vez seleccionadas las familias de modelos a analizar, se optó por no utilizar únicamente los pesos preentrenados con el dataset COCO, sino realizar un "transfer-learning" a un conjunto de datos personalizado, orientado a la detección de animales propios de zonas montañosas. El objetivo es aplicar los modelos a un caso de uso realista, que permita evaluar la capacidad de generalización y eficacia práctica de los mismos.
"Transfer-learning" o aprendizaje por transferencia es una técnica de AI que permite reutilizar el conocimiento adquirido en una tarea previa para mejorar el rendimiento en una tarea relacionada. En esta técnica se entrena el modelo a partir de una versión pre-entrenada con datos diferentes, en lugar de entrenar la red desde cero. "Transfer-learning" es una práctica habitual en el entrenamiento de modelos de inteligencia artificial, por la reducción del tiempo de entrenamiento y calidad de resultados que proporciona [28].

Con este objetivo, se inició una búsqueda preliminar de imágenes públicas de animales en repositorios abiertos, para entrenar el modelo. Finalmente, se seleccionaron tres conjuntos de datos anotados en la plataforma pública Roboflow [29], los cuales comparten temática y especies animales representadas.

Los dos primeros conjuntos contienen aproximadamente siete mil y diecisiete mil imágenes. Dichos conjuntos incluyen los siguientes animales: Oso, Ciervo, Zorro, Cabra, Jabalí, Lobo, Lince, Correcaminos, Coyote y Cara humana.

El tercer conjunto [30] ,contiene ocho mil imágenes con las siguientes especies: Oso, Ciervo, Zorro, Cabra, Jabalí, Lobo, Lince, Jabalí salvaje, Cría de jabalí, Tejón, Correcaminos, Coyote y Cara humana.

Por otro lado, YOLOX y YOLOv8 no comparten el mismo formato de datos ni el mismo flujo de entrenamiento. Por un lado, YOLOX utiliza anotaciones en formato COCO, donde toda la información de las imágenes y sus etiquetas se encuentra centralizada en un archivo con formato json. Por otro lado, YOLOv8 emplea un formato en el que cada imagen cuenta con su propio archivo de texto asociado. En este formato, el nombre del archivo de texto coincide con el de la imagen que describe, y cada línea del archivo contiene la clase y los parámetros de la caja delimitadora por instancia del objeto en formato YOLO.

La plataforma Roboflow permite la descarga de diferentes tipos de anotaciones para el mismo conjunto de datos. No obstante, al ser tres datasets distintos con anotaciones en dos formatos diferentes, para la conformación del conjunto de datos final fue necesario someter los conjuntos descargados a un preprocesamiento dividido en 4 etapas:

1. Fusión de los conjuntos: Unificar los 3 conjuntos en 1 solo.
2. Eliminación de animales no representativos para el caso de uso (animales de montaña) o con bajo ratio de aparición.
3. División equitativa de instancias de animales por subconjuntos para entrenamiento, validación y prueba.
4. Conversión y unificación de formatos.

3.1. Adaptación y preprocesado del conjunto de datos

Se consideró que el sistema de anotaciones en formato de archivos de texto era el más sencillo a la hora de procesarlo y convertirlo. Por ello, se definió un flujo de procesado que permite capturar la información en dicho formato para su posterior traslación al formato json.

3.1.1. Unificación de los conjuntos de datos

En primer lugar, se eliminaron mediante un programa propio en Python los datos de los animales no comunes entre los 3 conjuntos, dejando únicamente los más relevantes: Oso, ciervo, zorro, cabra, jabalí, lobo, lince, coyote y rostro humano.

En segundo lugar, se añadieron todas las imágenes del mismo tipo en una carpeta general. Por lo tanto, las imágenes y etiquetas de entrenamiento de cada conjunto original se añadieron a una carpeta general unificada, correspondiente al entrenamiento, y se actuó de igual forma para los conjuntos originales de validación y prueba.

3.1.2. Reorganización por clases

Posteriormente, se desarrolló otro código en Python para realizar el recuento de instancias de los animales por conjunto, para visualizar su distribución. El resultado de este análisis se muestra en la figura adjunta.

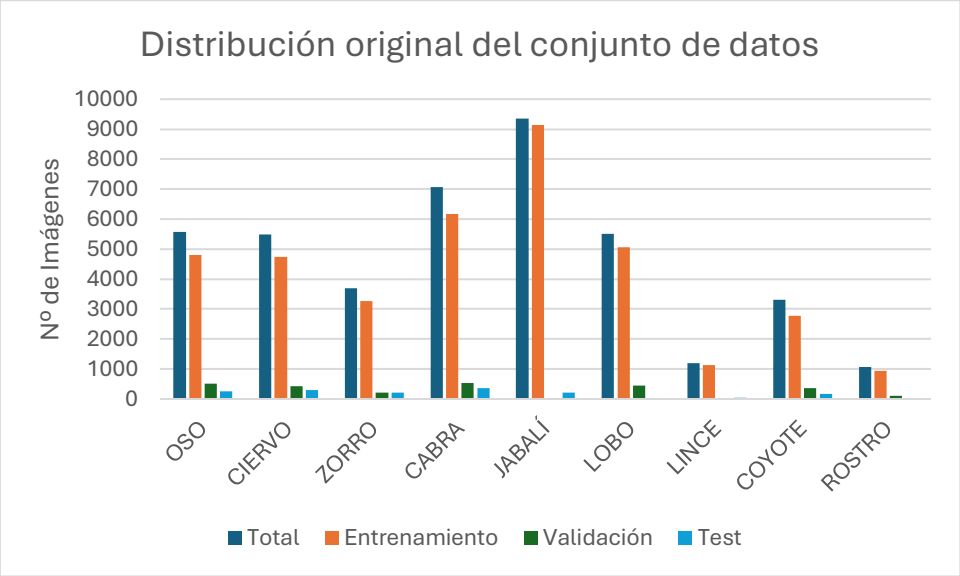


Figura 9. *Distribución original del conjunto de datos.*

Fuente: Elaboración propia

Debido a la gran disparidad entre apariciones de animales, se eliminaron las categorías con menos de 3000 instancias (Lince y Rostro humano). Para ello se desarrolló un código en Python que eliminaba las imágenes y anotaciones de las categorías Lince y "Rostro Humano" a la vez que ajustaba los nuevos índices de las clases.

3.1.3. Reparto equilibrado de instancias y formatos requeridos

Como se ha observado en las gráficas anteriores, los conjuntos de datos originales dividían las imágenes en 3 subconjuntos: entrenamiento, validación y test. Sin embargo, la distribución de imágenes en estos subconjuntos era poco homogénea. Por ello, se unieron todas las imágenes y anotaciones en 1 única carpeta y se realizó una división 70-15-15% respecto al recuento total de instancias. Lo que se traduce en que el 70% de todos los datos conforman el conjunto de entrenamiento, el 15% el conjunto de validación y el último 15% el conjunto de prueba.

La división de datos cumple un papel crucial en el desarrollo de modelos de inteligencia artificial dónde el conjunto de entrenamiento contiene las imágenes utilizadas para ajustar los pesos del modelo. Por otro lado, el conjunto de validación se evalúa al final de cada época para ajustar hiperparámetros y prevenir sobreajuste, fenómeno que se produce cuando el modelo Aprende no solo los patrones generales, sino también el ruido o las particularidades específicas del conjunto de entrenamiento, reduciendo su capacidad de generalización. Por último, el conjunto de prueba se mantiene completamente al margen del entrenamiento, utilizándose exclusivamente para calcular métricas objetivas del rendimiento final del modelo. Por este motivo, la gran mayoría de datos se emplean en el conjunto de entrenamiento.

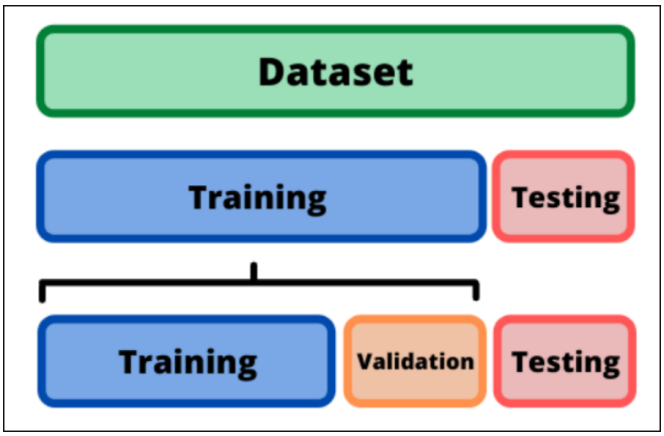


Figura 10. *Ilustración de las divisiones por subconjuntos de los datos de entrenamiento.*

Fuente: [31]

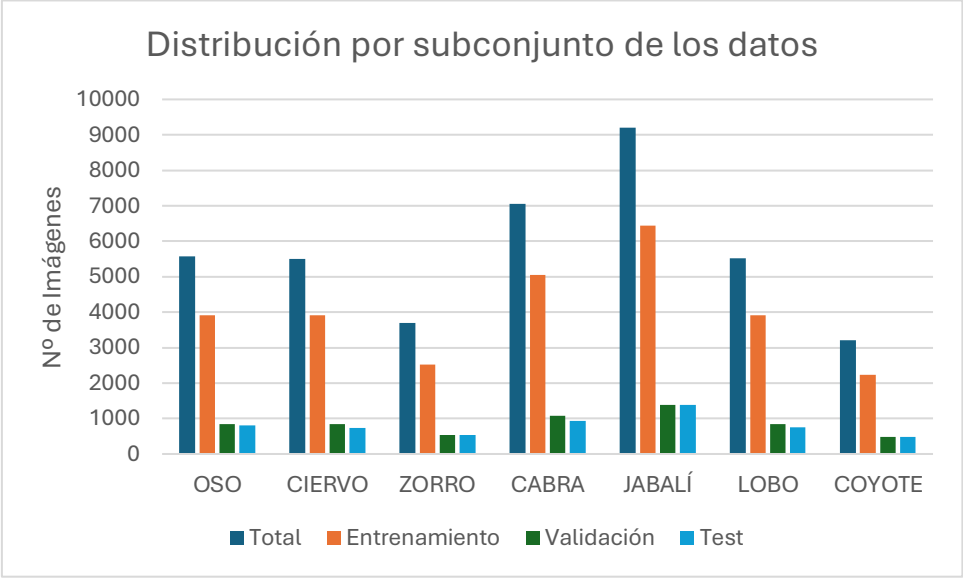


Figura 11. *Segunda distribución del conjunto de datos.*

Fuente: Elaboración propia

Una vez equilibrados los subconjuntos de datos (como se observa en la Figura 11), se procede a la eliminación de imágenes de las clases con más instancias, para evitar sesgos y reducir la brecha entre clases. Para ello se desarrolla un código que identifica las anotaciones de los jabalís y las elimina junto a su imagen asociada, para reducir el peso de este animal respecto al resto.

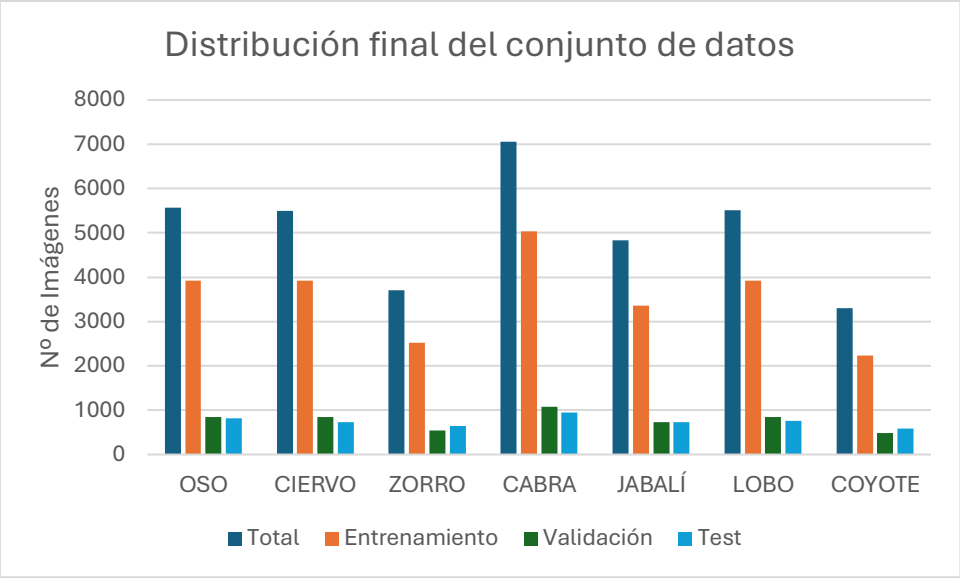


Figura 12. *Distribución final empleada del conjunto de datos.*

Fuente: Elaboración propia.

Debido a las líneas de investigación relacionadas con la industria ganadera del grupo GIM del departamento TEISA en donde se desarrolla este proyecto, se decidió mantener la ligera superioridad numérica de imágenes correspondientes a las cabras.

3.1.3. Traslación de la división de imágenes entre formatos

Una vez obtenida una distribución de datos satisfactoria con el formato de anotación en texto, se procede a su traslación al formato COCO.

Partiendo de la división realizada y de varios archivos json con las anotaciones de los datos, en lugar de repetir el procedimiento anterior, se optó por reunir todas las imágenes en una sola carpeta general, unir todos los archivos de anotaciones en uno y aplicar un nuevo flujo de datos. Con un programa propio de Python se buscaba el nombre de las imágenes del conjunto para YOLOv8 y se borraban de cada copia de las anotaciones totales correspondiente. Por ejemplo, si “imagen.jpg” se halla en la carpeta de entrenamiento, se borra su información de los json correspondientes para validación y test. De esta forma obtenemos las anotaciones en el formato correspondiente para el entrenamiento de YOLOX para cada subconjunto de datos.

Como paso adicional, es necesario especificar como en las anotaciones descargadas de Roboflow para YOLOv8 los datos de la caja limitadora se hallan normalizados, mientras que para YOLOX se encuentran en valores de píxeles absolutos relativos al tamaño de 640x640, por lo que para el barrido de tamaños de entrada es necesario transponer las coordenadas a los tamaños de 416 y 192. Acción realizada por un programa propio de Python que recorre los valores de los campos de las cajas limitadores y los multiplica por el cociente entre el tamaño objetivo y 640.

Por último, Ultralytics en su entrenamiento realiza un redimensionamiento de la imagen de entrada respecto al tamaño de entrada introducido como parámetro. Sin embargo, para YOLOX es necesario que el tamaño de la imagen coincida con el tamaño especificado, por ello se hizo uso de un código propio para redimensionar las imágenes mediante la librería “PIL” por interpolación del vecino más cercano (Nearest Neighbor) basado en copiar el valor del píxel más cercano sin suavizado, generando 2 nuevos datasets de diferentes tamaños de menor calidad visual.

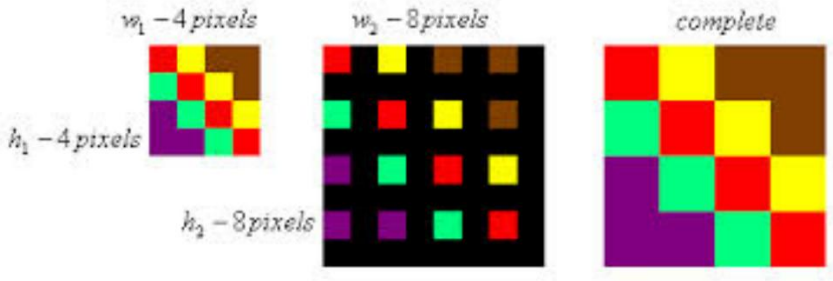


Figura 13. *Ejemplo de interpolación por vecino más cercano.*

Fuente: [32]

3.2. Entrenamiento con YOLOv8 mediante Ultralytics

El entrenamiento se llevó a cabo en el grupo GIM de la Universidad de Cantabria, utilizando un ordenador equipado con una unidad GPU RTX 4090 y herramientas de CUDA.

Gracias a la librería de Ultralytics, fue posible entrenar el modelo con solo cargar los pesos iniciales y la arquitectura, así como configurar los parámetros básicos.

Como hiperparámetros se utilizaron 100 épocas (número de veces que se recorre por completo el conjunto de datos) y un tamaño de lote de 32 imágenes (número de muestras procesadas por iteración).

El entorno de Ultralytics proporciona automáticamente gráficos que reflejan la evolución del entrenamiento y métricas clave sobre el rendimiento final del modelo, como se observa en la Figura 14.

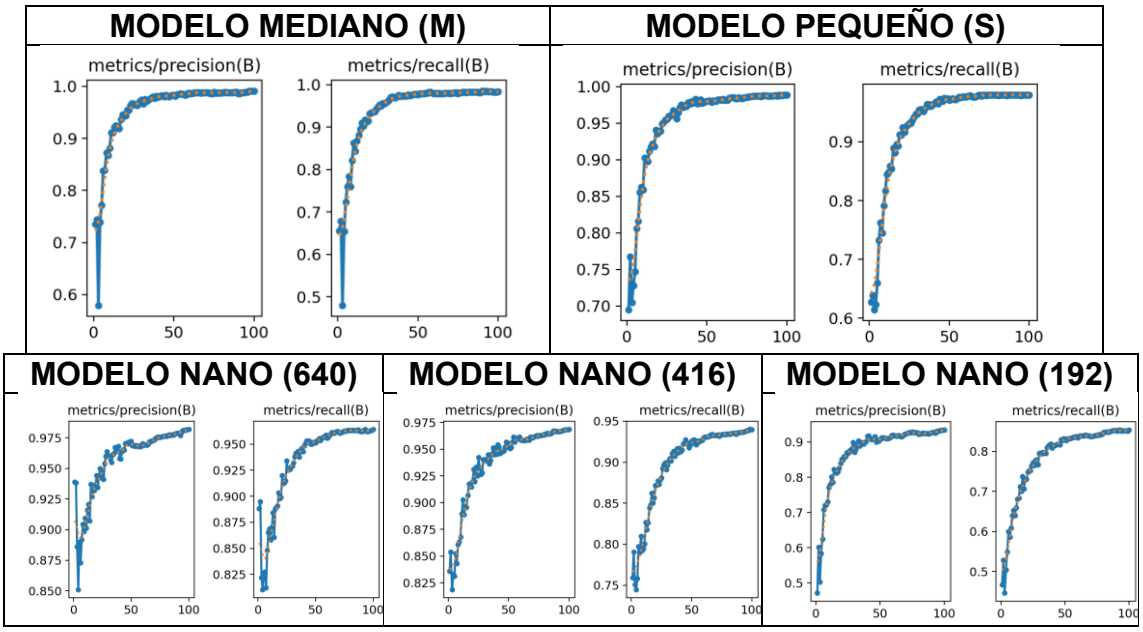


Figura 14. *Resultados de precisión y sensibilidad durante el entrenamiento de YOLOv8.*

Fuente: Elaboración propia

Se observa cómo el proceso de entrenamiento en Ultralytics aplica de forma automática una estrategia de ajuste dinámico del “learning rate”, comenzando con valores relativamente altos en las primeras épocas. Esto permite que el modelo explore rápidamente el espacio de soluciones y se adapte con agilidad a los datos, lo que provoca las variaciones bruscas en las métricas de las épocas iniciales.

A medida que el entrenamiento avanza, la tasa de aprendizaje se reduce de forma progresiva, favoreciendo una optimización más estable. En paralelo, durante las últimas 15 épocas, se desactiva la técnica de “mosaic augmentation”, una estrategia que combina varias imágenes en una sola para exponer al modelo a objetos en distintas escalas y contextos. Al eliminar esta técnica al final, se utilizan imágenes más realistas y parecidas a las del set de validación, generando un descenso repentino en las funciones de pérdida.

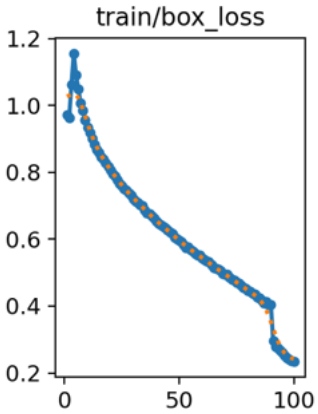


Figura 15. *Evolución de los valores de pérdida durante el entrenamiento del modelo mediano.*

Fuente: Elaboración propia

Otra conclusión obtenida tras el entrenamiento es como converge el valor de las métricas, a valores casi ideales, a partir de las 50 épocas. Por ello se realizó un entrenamiento con solo 50 épocas, en lugar de 100, para observar si este comportamiento es debido a la arquitectura de la red o al mecanismo de entrenamiento de Ultralytics. Se observó (Figura 16) que la trayectoria descrita es idéntica incluso con la mitad de las épocas.

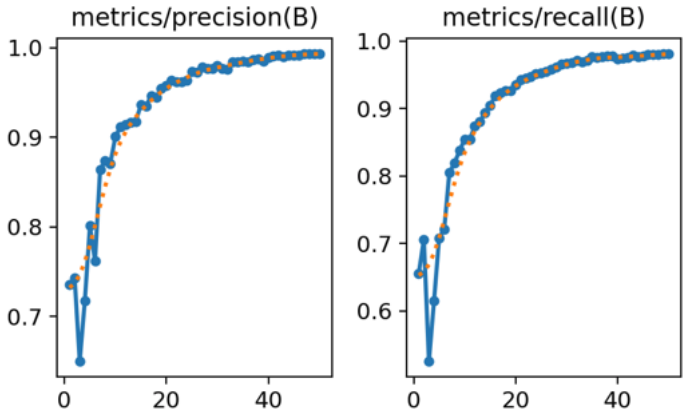


Figura 16. *Gráficas de los resultados durante el entrenamiento del modelo mediano en 50 épocas.*

Fuente: Elaboración propia.

Por ello se concluye que la convergencia a valores cercanos al 100% es debida al sistema de entrenamiento de Ultralytics.

Para ilustrar de forma comparativa los resultados respecto a YOLOX, se ha extraído únicamente los valores de las métricas COCO más representativas: la precisión media (relativa a la exactitud de la caja) y sensibilidad media (relativa a la identificación de objetos). Estas métricas son:

- El valor de Average Precision (AP) en todo el rango de IoU de 0.50 a 0.95 y de media de todas las áreas.
- El Average Recall (AR) medio en todos los tipos de área con hasta 100 detecciones por imagen en el rango de IoU de 0.50 a 0.95.

Métricas / versión	Precisión (AP)	Sensibilidad (AR)
Mediano (640x640)	94.67%	98.35%
Pequeño (640x640)	92.11%	99.14%
Nano (640x640)	87.43%	96.44%
Nano (416x416)	84.38%	94.03%
Nano (192x192)	73.03%	85.33%

Tabla 2. *Valores de precisión y sensibilidad obtenidos en el final del entrenamiento de los modelos de YOLOv8.*

Fuente: Elaboración propia

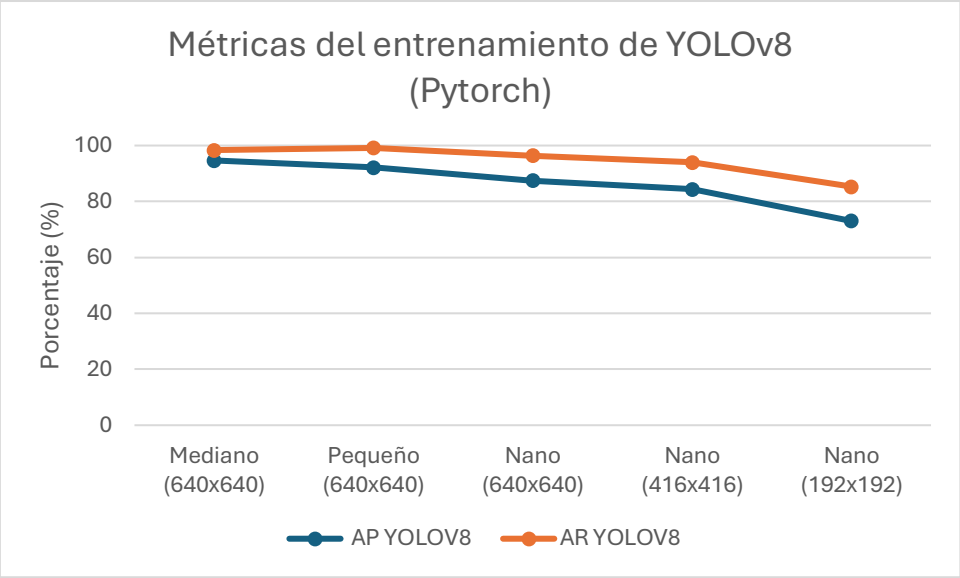


***Figura 17**. Comparación de las métricas finales en el entrenamiento entre modelos de YOLOv8.*

Fuente: Elaboración propia

3.3. Entrenamiento con YOLOX desde repositorio oficial

En el caso de YOLOX, es necesario clonar su repositorio oficial [33] y configurar los denominados "archivos de experimento". Estos archivos, incluidos en el propio repositorio, contienen el código que define los hiperparámetros del modelo y deben modificarse para adaptar el entrenamiento a cada caso concreto. Entre los parámetros a ajustar se encuentran las rutas a las imágenes y anotaciones, el tamaño de entrada, el número de clases, el número de épocas y el tamaño del lote.

A su vez, se debe modificar el código fuente para ajustar la llamada a las rutas para la carga de datos. También se debe añadir la ruta del repositorio al path de Python, de modo que las funciones del repositorio se utilicen correctamente en lugar de intentar importarlas desde paquetes instalados globalmente.

Una vez realizados estos ajustes y descargados los pesos preentrenados en COCO, el entrenamiento se realiza mediante el script "train.py", incluido en el repositorio [34].

Al finalizar, a diferencia del entorno de Ultralytics, no se facilitan resultados intermedios por época que permiten ver la evolución del modelo de manera gráfica, sino que se genera un archivo de texto con todos los valores obtenidos durante el entrenamiento, y un resumen de las métricas finales de pérdida, precisión y otras estadísticas relevantes del modelo final.

```
Average forward time: 2.30 ms, Average NMS time: 0.61 ms, Average inference time: 2.92 ms
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.871
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.959
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.932
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.607
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.794
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.891
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.682
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.897
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.899
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.651
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.828
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.918
per class AP:
| class   | AP     | class   | AP     | class   | AP     |
|:--------|:-------|:--------|:-------|:--------|:-------|
| Bear    | 94.217 | Deer    | 83.196 | Fox     | 93.635 |
| Goat    | 86.184 | Hog     | 80.259 | Wolf    | 82.651 |
| Coyote  | 89.480 |         |        |         |        |
per class AR:
| class   | AR     | class   | AR     | class   | AR     |
|:--------|:-------|:--------|:-------|:--------|:-------|
| Bear    | 96.473 | Deer    | 85.912 | Fox     | 94.461 |
| Goat    | 89.081 | Hog     | 82.937 | Wolf    | 86.559 |
| Coyote  | 93.784 |         |        |         |        |
```

Figura 18. *Resultados del entrenamiento del modelo mediano de YOLOX.*

Fuente: Elaboración propia

Debido al gran detalle de las métricas, y para facilitar su comparación con las métricas de Ultralytics, se han extraído solo las dos más relevantes por modelo como representantes de su rendimiento.

Métricas / versión	Precisión (AP)	Sensibilidad (AR)
Mediano (640x640)	87.1%	89.9%
Pequeño (640x640)	83.5%	87.1%
Tiny (640x640)	77.3%	81.6%
Tiny (416x416)	79.8%	83.3%
Tiny (192x192)	61.4%	66.6%

Figura 19. *Valores de precisión y sensibilidad obtenidos en el final del entrenamiento de los modelos de YOLOX.*

Fuente: Elaboración propia

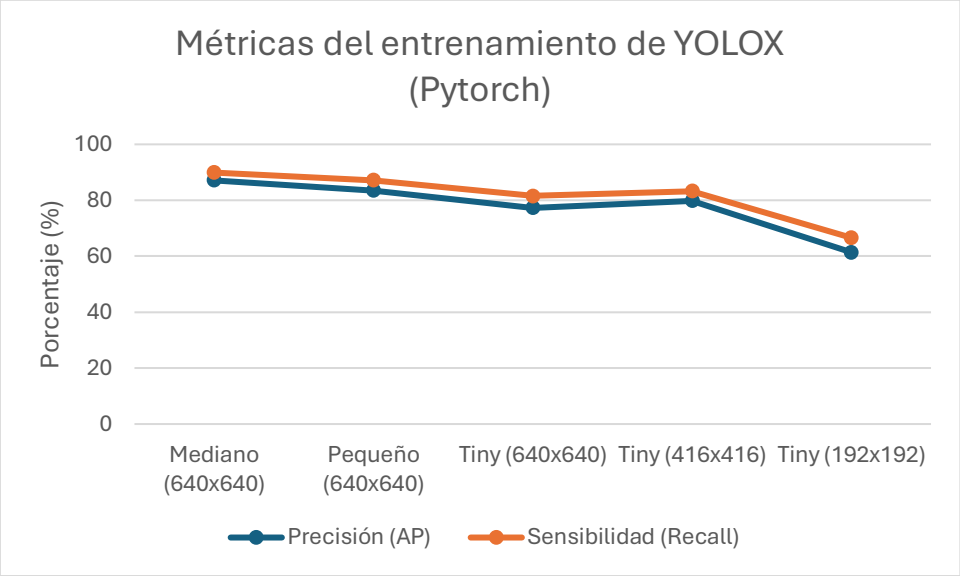


Figura 20. *Comparación de las métricas finales en el entrenamiento entre modelos de YOLOX.*

Fuente: Elaboración propia.

La comparación entre resultados demuestra como YOLOv8 ofrece mejores prestaciones, tanto en precisión como en sensibilidad.

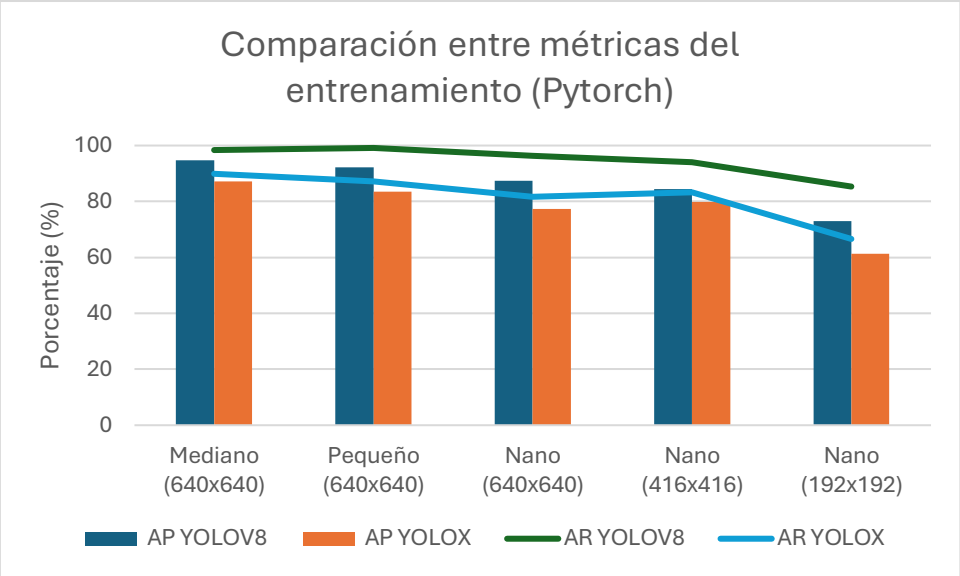


***Figura 21**. Comparación de los resultados de los entrenamientos de YOLOv8 y YOLOX.*

Fuente: Elaboración propia

3.4. Evaluación y comparación con el conjunto de prueba en PyTorch

Los resultados del entrenamiento no se deben tomar como representantes del rendimiento del modelo directamente, pues pueden estar sesgados o haber sufrido fenómenos de sobreajuste, como se explicó anteriormente. Por ello, resulta esencial evaluar el rendimiento utilizando un conjunto de prueba independiente a los datos de entrenamiento y validación, datos nuevos que el modelo no debe conocer, permitiendo obtener una medida más realista de cómo se comportará el modelo.

Por este motivo, se procede a evaluar las métricas con el conjunto de prueba (test). Al contrario que en el caso del entrenamiento, Ultralytics no genera directamente las métricas COCO del modelo YOLOv8 con el conjunto de prueba, aunque sí las representa en formato gráfico. Para su comparación numérica frente a los valores ofrecidos por YOLOX, se ha desarrollado un código que modifica el formato de las predicciones generadas por YOLOv8 respecto al

conjunto de prueba. Ello permite obtener los resultados que se muestran en la Figura 22.

YOLOv8

Métricas / versión	Precisión (AP)	Sensibilidad (AR)
Mediano (640x640)	90.50%	93.24%
Pequeño (640x640)	88.04%	91.21%
Nano (640x640)	83.75%	87.84%
Nano (416x416)	80.80%	85.37%
Nano (192x192)	70.25%	77.24%

Figura 22. *Valores de precisión y sensibilidad obtenidos en el conjunto de prueba por los modelos de YOLOv8.*

Fuente: Elaboración propia

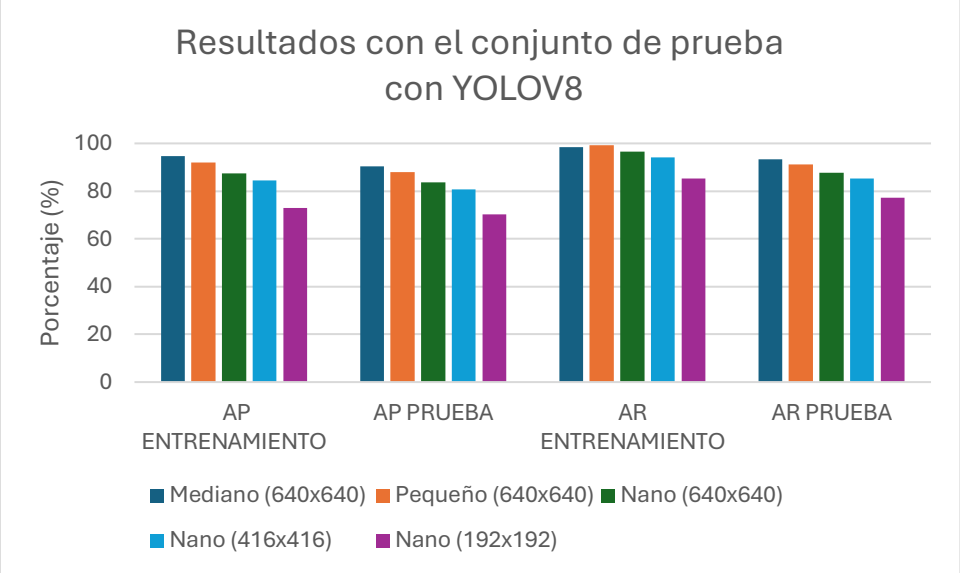


Figura 23*. Resultados con el conjunto de prueba con YOLOv8.*

Fuente: Elaboración propia

YOLOX

Métricas / versión	Precisión (AP)	Sensibilidad (AR)
Mediano (640x640)	86.7%	89.6%
Pequeño (640x640)	83.2%	86.6%
Tiny (640x640)	77.1%	81.3%
Tiny (416x416)	79.5%	83.1%
Tiny (192x192)	59.8%	64.6%

Figura 24. *Valores de precisión y sensibilidad obtenidos en el conjunto de prueba por los modelos de YOLOX.*

Fuente: Elaboración propia

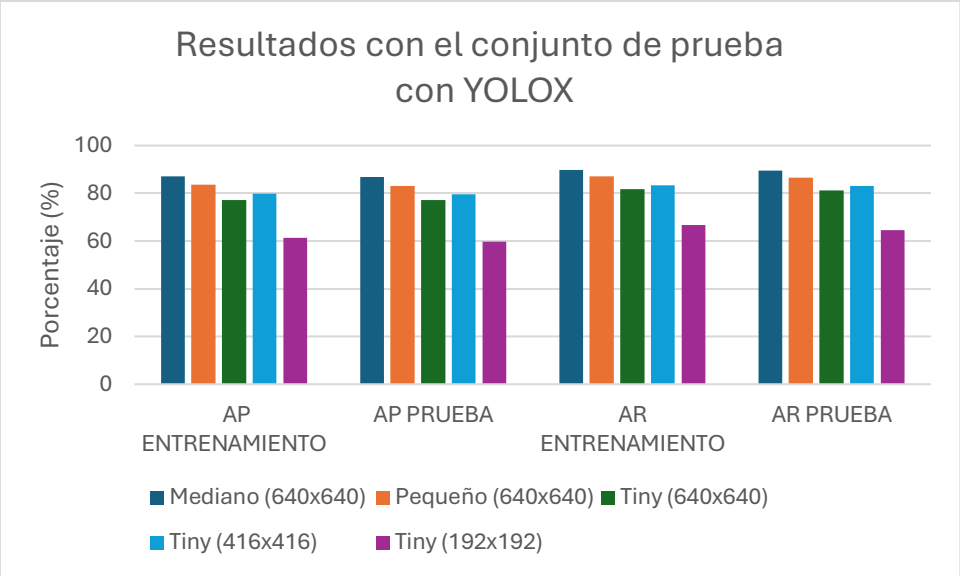


Figura 25. *Resultados con el conjunto de prueba con YOLOX.*

Fuente: Elaboración propia

Tras obtener las métricas relevantes en el entorno de PyTorch de los modelos YOLOX y YOLOv8, se extraen las siguientes conclusiones:

1. Los modelos no han sufrido sobreajuste. Los valores respecto al conjunto de prueba son similares a los obtenidos durante el entrenamiento.
2. YOLOv8 genera mejores predicciones que YOLOX, tanto en precisión como en sensibilidad.
3. El rendimiento en función del número de parámetros actúa de acorde a lo esperado: Mayor número de parámetros produce mejores resultados.

Respecto a la influencia de la resolución de las imágenes, en YOLOv8 se observa el fenómeno esperado: a menor resolución, el rendimiento disminuye debido a la pérdida de información.

En cambio, en YOLOX ocurre un comportamiento no esperado, ya que es la resolución intermedia la que ofrece los mejores resultados. Esto puede deberse

a varios motivos, siendo el más probable que esta arquitectura estaba pre-entrenada con formato 416x416 pixeles, lo que favorece las predicciones de las imágenes de este formato.

Capítulo 4. Conversión y optimización del modelo a TensorFlow-Lite

Para realizar la conversión de modelos entre frameworks se sigue, en primera instancia, el esquema de la Figura 26, que será comentado en los siguientes apartados.

Figura 26. *Esquema secuencial traslación entre de frameworks.*

Fuente: Elaboración propia

4.1. Exportación de modelos a ONNX

Para la exportación al formato ONNX se utilizan las herramientas propias de cada entorno. En el caso de YOLOv8, se emplea el sistema de exportación integrado en Ultralytics, mientras que en YOLOX, se recurre a los scripts proporcionados en su repositorio oficial.

Un aspecto clave en este proceso es la elección del "opset", que hace referencia al conjunto de operaciones (operators) disponibles en la versión de ONNX empleada. Un "opset" más reciente permite implementar nodos con operaciones más avanzadas. Sin embargo, un "opset" más conservador y antiguo suele proporcionar una mayor una mayor portabilidad [20].

En este trabajo se optó por el "opset" 13, al tratarse de una versión ampliamente soportada y estable en múltiples entornos de ejecución.

4.2. Conversión a formato TensorFlow

Tras obtener el archivo descriptor del modelo en formato ONNX desde PyTorch, se procede con la siguiente etapa de la Figura 26: su conversión al formato

“SavedModel” de TensorFlow. Este formato almacena tanto la arquitectura del grafo como los pesos del modelo en archivos “.pb” y metadatos complementarios [35]. Además, el formato reseñado es necesario como paso intermedio para la posterior conversión del modelo a TensorFlow Lite, pues las posibles modificaciones del grafo se llevan a cabo con su información.

La conversión se realiza mediante la librería onnx-TensorFlow, que permite capturar el grafo ONNX y traducirlo a una representación en TensorFlow, a través del módulo “onnx_tf.backend.prepare”.

Sin embargo, un aspecto crítico de esta fase es la compatibilidad entre versiones. Dado que tanto ONNX como TensorFlow y sus librerías puente están en constante evolución, es frecuente encontrar errores si las versiones no están alineadas.

Por ello, resulta fundamental identificar y fijar un conjunto de versiones específicas que sean compatibles entre sí, para que la conversión se realice de forma exitosa. Por ejemplo, TensorFlow, en sus versiones 2.x, incorpora cambios en la gestión del grafo respecto a su versión 1.x, especialmente en lo relativo a la ejecución diferida y el sistema de firmas para modelos exportados. Estos cambios afectan directamente a cómo se representan y almacenan los modelos, lo que puede provocar errores de incompatibilidad si se usan versiones no alineadas.

En este proyecto se empleó el siguiente conjunto de librerías para garantizar una conversión estable:

- TensorFlow: versión 2.15.0
- keras: versión 2.15.0
- keras_applications: versión 1.0.8
- TensorFlow-probability: versión 0.22.1
- onnx-tf: versión 1.14.0
- numpy: 1.25.1

Estas versiones fueron seleccionadas tras pruebas iterativas, para evitar conflictos comunes relacionados con operadores no soportados, cambios en estructuras internas de TensorFlow o errores durante la exportación del grafo.

4.3. Conversión a TensorFlow-Lite y cuantización INT8

Con el grafo en formato “SavedModel”, ya es posible iniciar la etapa de conversión a TensorFlow Lite mediante la herramienta oficial de conversión del framework.

Para realizar la cuantización a enteros de 8 bits (INT8), es necesario implementar una función generadora que proporcione imágenes de un dataset representativo. Esta función toma las rutas de las imágenes, las carga, las convierte de enteros sin signo (formato .jpg) a punto flotante (formato esperado por el modelo original), y emplea el operador “yield” de Python para entregar cada imagen de manera secuencial al convertidor. De este modo, el proceso puede recoger las estadísticas de activación necesarias para la cuantización, y calcular los parámetros necesarios presentados en el capítulo 2, apartado 6.

Para generar este dataset representativo, se ha desarrollado un código en Python que organiza las imágenes por clase utilizando un diccionario, donde cada clave corresponde al identificador de clase y su valor es una lista con los nombres de los archivos de anotaciones (formato en archivos de texto) en los que aparece dicha clase. Una vez agrupadas las instancias, se calcula el número objetivo de imágenes a seleccionar por clase como el cociente entre el número total de instancias deseadas y el número de clases presentes.

A continuación, para cada clase, se realiza una selección aleatoria de dicho número de archivos, asegurando una distribución equilibrada en el conjunto final con el objetivo de que en la cuantización se preserve adecuadamente la variabilidad de entradas reales que el modelo encontrará durante la inferencia.

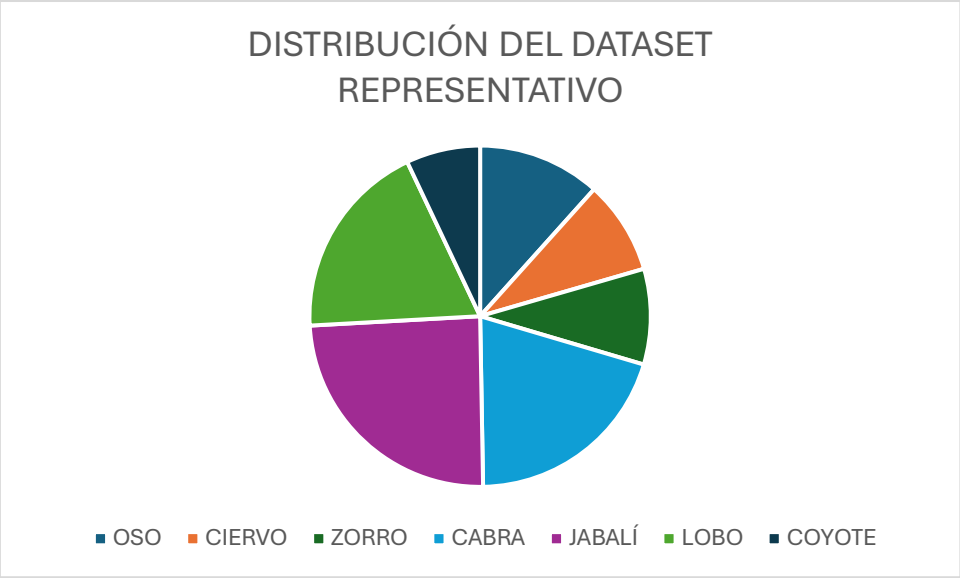


Figura 27. *Distribución del dataset representativo.*

Fuente: Elaboración propia

Aunque se han seleccionado 500 imágenes de forma equilibrada por clase, la distribución final de instancias por clase no es perfectamente uniforme. Esto se debe a que una misma imagen puede contener múltiples objetos, ya sean varias instancias de la misma clase o combinaciones de diferentes animales. Por ejemplo, en muchas imágenes donde aparece el coyote, este suele ser el único animal presente, lo que implica un menor número de instancias por imagen en comparación con otras clases que comparten escena con varias especies.

4.3.1. Proceso de conversión directa para YOLOX

Como ya se ha mencionado previamente, YOLOX es un modelo con respaldo académico y un gran número de despliegues exitosos. Esto se ve reflejado en su integración con la herramienta de conversión de TensorFlow, sin necesidad de acciones intermedias, dando lugar a los archivos nativos en TFlite cuantizados a enteros de 8 bits.

En la Tabla 3 se muestra la comparación de tamaños respecto a la versión en PyTorch con pesos reales.

YOLOX

Modelo / Tamaño (KB)	PyTorch (.pth)	TFLite (.tflite)
Mediano (640x640)	198.019	25.562
Pequeño (640x640)	70.176	9.233
Tiny (640x640)	39.641	5.323
Tiny (416x416)	39.641	5.323
Tiny (192x192)	39.641	5.323

Tabla 3. *Comparación entre los tamaños de archivos de PyTorch y TensorFlow para YOLOX.*

Fuente: Elaboración propia

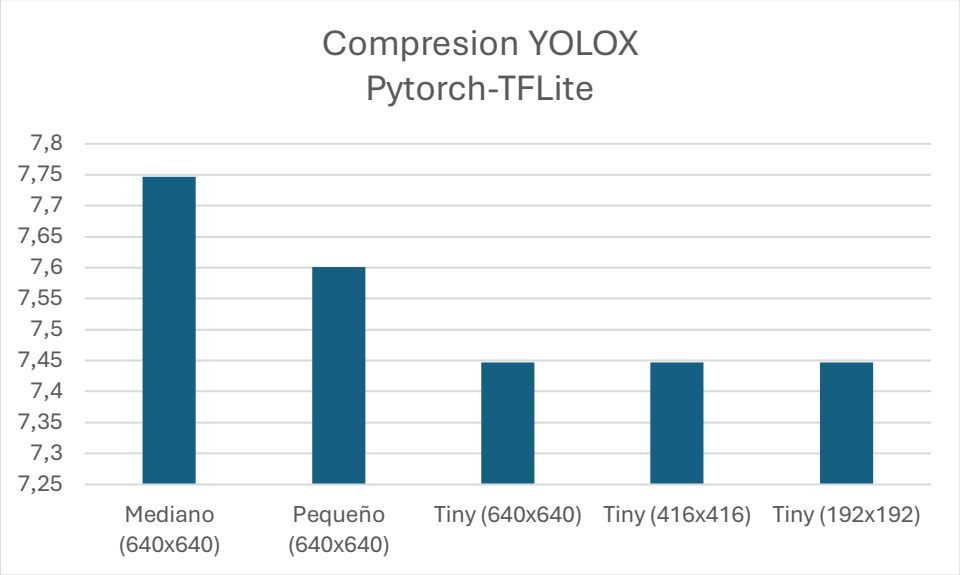


Figura 28. *Comparación del factor de compresión entre archivos de YOLOX.*

Fuente: Elaboración propia

Como se puede observar, la media de reducción de tamaños alcanza un cociente medio de 7.5. Es importante destacar que la información del modelo YOLOX requiere su repositorio, debido a que el archivo de extensión ".pth" guarda únicamente los pesos y se necesita el código del modelo del repositorio para volver a usarlo. Esto es debido a que YOLOX está diseñado en PyTorch para emplearse junto a su repositorio. La ventaja de la conversión a TFLite es que el archivo listo para realizar inferencias con el intérprete del framework, pesa 7 veces menos que solamente el archivo de pesos en PyTorch.

A continuación, se compara el tamaño del formato TFLite con el descriptor ONNX, que representa la arquitectura encapsulada en un solo fichero, de forma parecida al formato ".pt" (asociado al modelo ya serializado como objeto completo en PyTorch) en el que se halla la versión original usada en este proyecto de YOLOv8.

YOLOX

Modelo / Tamaño (KB)	ONNX (.onnx)	TFLite (.tflite)
Mediano (640x640)	98.794	25.562
Pequeño (640x640)	34.962	9.233
Tiny (640x640)	19.717	5.323
Tiny (416x416)	19.717	5.323
Tiny (192x192)	19.717	5.323

Figura 29. *Comparación entre los tamaños de archivos de ONNX y TensorFlow para YOLOv8.*

Fuente: Elaboración propia

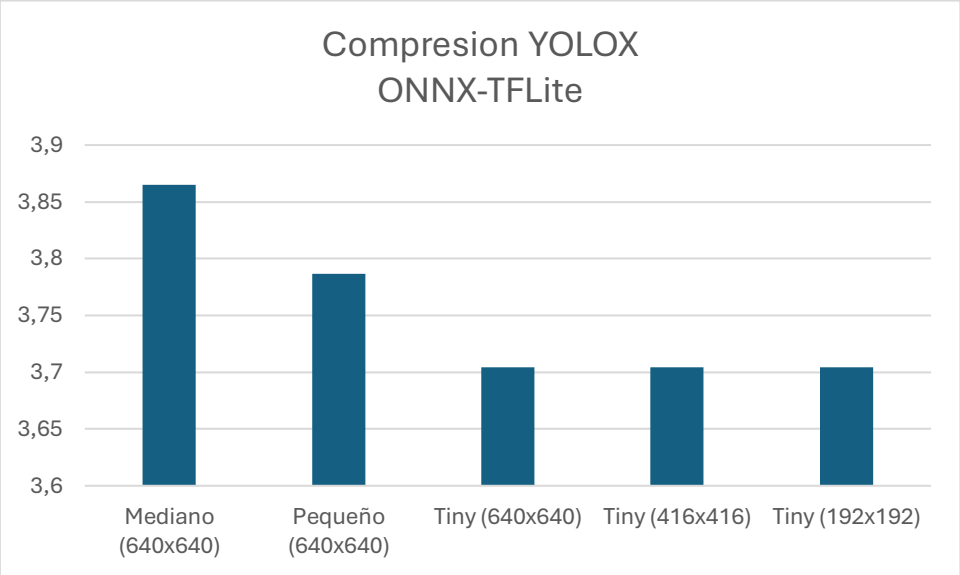


Figura 30. *Comparación del factor de compresión entre archivos ONNX y TF de YOLOX.*

Fuente: Elaboración propia

Frente al modelo de YOLOX encapsulado en el formato descriptor ONNX, el cociente de compresión toma un valor medio de 3.75.

4.3.2. Proceso de conversión para YOLOv8

En el caso del modelo YOLOv8, se siguió el mismo procedimiento utilizado para YOLOX y descrito en el apartado anterior. Sin embargo, se produjo en error en el proceso. Este error era debido a la incompatibilidad entre algunas de las operaciones que efectuaban algunos nodos de la red. Aunque esta incompatibilidad no imposibilitaba la conversión a TFLite, requiere añadir una capa donde el modelo utilice TensorFlow para las operaciones incompatibles, generando un modelo híbrido.

Como esta aproximación no pareció adecuada, se optó por la sustitución de las operaciones conflictivas. Para ello se visualizó la arquitectura de la red en profundidad, gracias a la aplicación web de netron [36], y se , localizaron los nombres e identificadores de los nodos conflictivos. Dichos nodos conflictivos se reemplazaron por operaciones nativas de TensorFlow-Lite eliminando los problemas de incompatibilidad.

En el caso del modelo YOLOv8, las operaciones conflictivas eran “SliceV” y “Div”. La primera operación (“SliceV”) recibe un tensor de entrada e información sobre el tipo de división, y se emplea para la división dinámica del tensor, incompatible por naturaleza con TFLite.

Gracias al análisis de la red se observó cómo, salvo en un caso, este nodo solo realizaba divisiones simétricas, es decir, dividir un tensor por un eje en dos tensores del mismo tamaño. Este procedimiento ya esta implementado en una función nativa de TFLite, llamada “Split” [37], por lo que dichos nodos se sustituyeron por nodos compatibles. En el caso de la excepción, la división era asimétrica a 64 y 7, partiendo de un eje de valor 71. Por ello, se duplicó el tensor y se implementó el nodo “Slice” [38], de forma que de un nodo divida de forma asimétrica y se instanciaron 2 nodos, donde el primero toma los primeros 64 valores y el segundo los 7 últimos respectivos al eje de división.

El nodo “DIV”, realiza la división aritmética de los valores de un tensor por un valor definido. Otra vez gracias a la investigación en netron, se detectó como

este valor constante era siempre el número dos. Por ello se cambió el nodo de división por un nodo de multiplicación por 0.5.

Estos cambios fueron llevados a cabo mediante la generación del grafo en el formato “SavedModel” de TensorFlow. A partir del grado original, se obtuvo la lista de nodos que se iban añadiendo aun nuevo grafo, salvo aquellos cuyo nombre coincidiera con los conflictivos. De esta forma se iba formando el grafo nuevo al recorrer el anterior. En el proceso se modifican los nodos conflictivos como se ha indicado y se guarda el modelo en el formato “SavedModel”.

Es en este paso donde además se hallaba la oportunidad de añadir nuevos nodos a voluntad, por lo que se añadió al final de la red un nodo de transposición para obtener el mismo formato tensorial para ambas familias de modelos, de la forma: 1 x NºCampos x NºPredicciones

De esta forma se obtuvo el archivo convertido, sin dependencia de Ultralytics, de YOLOv8 en formato TFLite desde ONNX.

Al revisar el rango de valores descuantizados del modelo convertido se comprobó que, aunque el procedimiento aplicado era correcto y coherente con el modelo original en PyTorch, los resultados no se obtenían en un formato decodificable.

Se recurrió a delegar la conversión al sistema automatizado que ofrece la propia librería Ultralytics, la cual recurre internamente al framework experimental de NVIDIA denominado “ai-edge-liteRT” para realizar la exportación del modelo a formato TensorFlow Lite. A diferencia del procedimiento propuesto anteriormente, este nuevo proceso es más complejo y opaco y los modelos que genera, aunque logra generar modelos funcionales, los resultados obtenidos son menos transparentes y más difíciles de interpretar (como introducir nodos redundantes en el grafo, como multiplicaciones consecutivas por -1 sin una función aparente)

Además, de entre las versiones exportadas por la herramienta de NVIDIA, solo una versión ofrecía un tensor de salida decodificable, las demás versiones, así como el modelo exportado con anterioridad, producen valores no interpretables. Esto evidencia que la conversión no se limita únicamente a la cuantización de pesos, sino que también modifica la estructura de la cabeza de detección y el proceso de decodificación. Lo que demuestra que la arquitectura original no está diseñada para producir directamente cajas de predicción listas para visualización.

A pesar de todo esto, el modelo final sí ofrece los 11 valores de salida con valores normalizados y una descuantización útil, por lo que se ha dejado el refinamiento del método original como línea de investigación futura.

YOLOv8

Modelo / Tamaño (KB)	PyTorch (.pt)	TFLite (.tflite) (Propio)	TFLite (.tflite) (Ultralytics)
Mediano (640x640)	50.824	25.993	25.479
Pequeño (640x640)	22.002	11.358	11.057
Nano (640x640)	6.112	3.262	3.112
Nano (416x416)	6.082	3.248	3.084
Nano (192x192)	6.063	3.240	3.067

Tabla 4. *Comparación entre los tamaños de archivos de PyTorch y TensorFlow para YOLOv8.*

Fuente: Elaboración propia

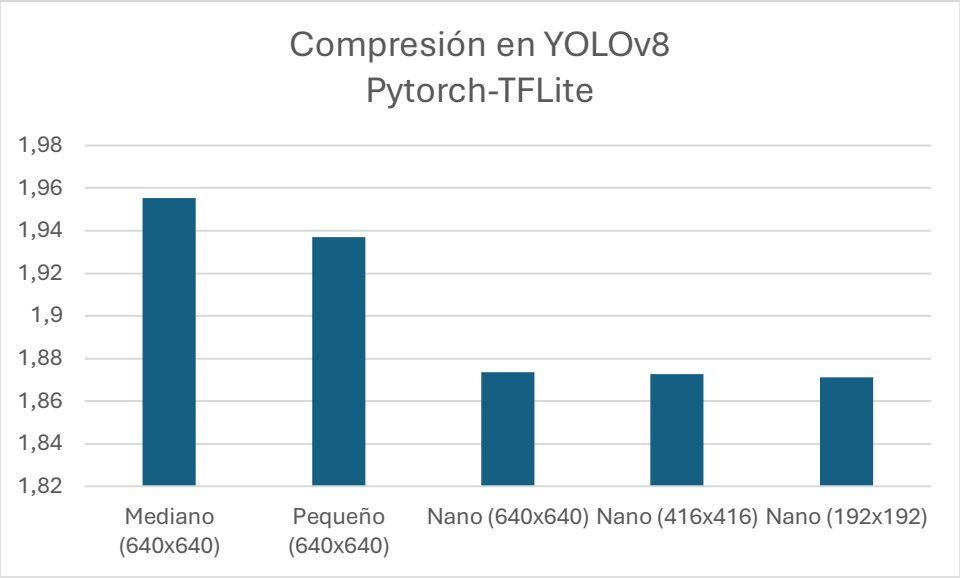


Figura 31. *Comparación del factor de compresión entre archivos PyTorch y TensorFlow de YOLOv8.*

Fuente: Elaboración propia

El cociente entre el archivo en PyTorch y su análogo en TensorFlow Lite toma un valor medio de 1.9, por lo que se reduce aproximadamente a la mitad del tamaño original.

También se observa como el tamaño de los valores de entrada para los que esta preparado el modelo, no influyeron en los modelos de YOLOX, pero sí en los de YOLOv8. Esta pequeña fluctuación puede ser debida al uso de capas específicas en la parte inicial, que dependen en mayor medida de las dimensiones de entrada.

4.3.3. Conversión a sistema uf2

El formato UF2 de Microsoft [39] nace de la necesidad de simplificar la programación de dispositivos embebidos. Su principal función es encapsular imágenes de firmware o modelos de machine learning en un formato que facilite su carga directa mediante mecanismos estándar, como el modo USB Mass Storage, de forma que el modelo junto con el firmware pueda ser introducido fácilmente en sistemas embebidos con recursos limitados y sin necesidad de herramientas complejas.

Este formato permite actualizar el dispositivo simplemente copiando un archivo en el almacenamiento simulado en el mismo, como ocurre en los dispositivos de la familia SenseCap de SeedStudio.

Del repositorio de SeedStudio se extrajo el código de conversión de un modelo en formato TFLite a un archivo encapsulado en UF2 [40], con la finalidad de realizar una estimación del tamaño del archivo a almacenar en el dispositivo.
El problema de los sistemas embebidos es que el tamaño del fichero UF2 está limitado por el tamaño de la memoria flash del dispositivo. En el caso de los dispositivos utilizados originalmente, el tamaño de la memoria flash está limitado a 2MB, en los cuales no solo debe incluirse el modelo TFLite, sino también todo el encapsulado. Esto hace que solo modelos de medio megabyte puedan ser inducidos en este tipo de sistemas.

No obstante, es posible una expansión futura que permita almacenar hasta 6 MB de modelo flasheado. Este límite, reconocido en plataformas como Edge Impulse, responde a un equilibrio entre la capacidad de memoria disponible en microcontroladores y la necesidad de ejecutar modelos cada vez más complejos para tareas avanzadas de inferencia local.

Tamaño del archivo en formato uf2

Modelo/Tamaño (KB)	YOLOV8	YOLOX
Mediano (640x640)	51.986	51.124
Pequeño (640x640)	22.715	18.465
Nano/Tiny (640x640)	6.524	10.646
Nano/Tiny (416x416)	6.496	10.645
Nano/Tiny (192x192)	6.479	10.645

Tabla 5. *Comparación del tamaño en kilobytes del archivo uf2 para cada modelo.*

Fuente: Elaboración propia

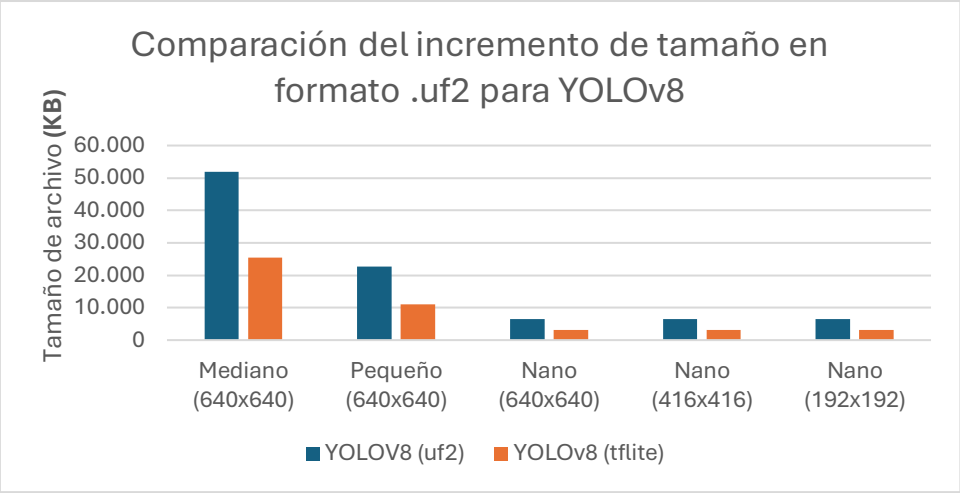


Figura 32. *Comparación del incremento de tamaño del archivo uf2 respecto a los archivos en tflite en YOLOv8.*

Fuente: Elaboración propia

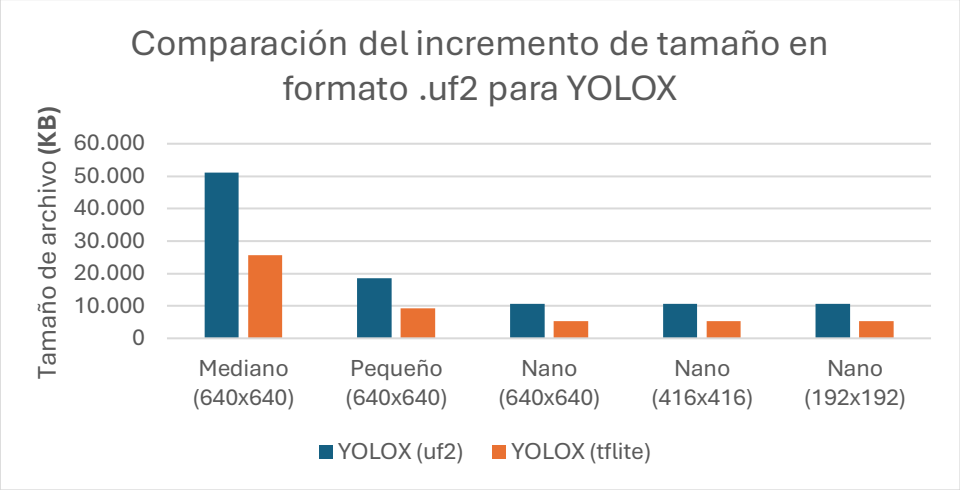


Figura 33. *Comparación del incremento de tamaño del archivo uf2 respecto a los archivos en tflite en YOLOX.*

Fuente: Elaboración propia

Gracias a las gráficas de las Figuras 33 y 34 se observa como el encapsulado UF2 prácticamente duplica el tamaño del archivo. Por lo tanto, para sistemas con 2MB de flash es necesario condensar el modelo en aproximadamente medio megabyte.

Capítulo 5. Despliegue en dispositivo: Raspberry Pi 4

Debido a los problemas comentados anteriormente con dispositivos de la familia SeedStudio, se decidió utilizar como plataforma de prototipado una Raspberry Pi 4. Aunque no se trata de un sistema embebido para AI en sentido estricto, ni de un microcontrolador tradicional (como los basados en arquitecturas ARM Cortex-M) sí ofrece un entorno intermedio ideal para validar modelos y algoritmos antes de su implementación final en hardware más limitado.

5.1. Justificación del entorno y perfil hardware

El dispositivo seleccionado para las pruebas de inferencia ejecuta un sistema operativo basado en arquitectura ARMv7 de 32 bits (armv7l), a pesar de que su CPU, un ARM Cortex-A72, es compatible con la arquitectura ARMv8-A de 64 bits [41]. Esto lo posiciona como un entorno de menor capacidad computacional que los procesadores x86-64, habitualmente presentes en ordenadores personales, especialmente en tareas que requieren altas prestaciones como la inferencia con redes neuronales.

Además, es una primera aproximación hacia un dispositivo más limitado pero eficiente energéticamente, como es la Raspberry Pi Zero, con arquitectura ARMv6 y sin soporte para operaciones de punto flotante avanzadas [42].

Además, el realizar la inferencia sobre este dispositivo reafirma la ventaja de la conversión a TensorFlow-Lite pues no existe una versión oficial de PyTorch para arquitecturas ARM.

5.2. Preparación del entorno de ejecución

En este proyecto, la Raspberry Pi 4 ejecuta un sistema operativo basado en Linux (Raspberry Pi OS), por lo que haciendo uso de su interfaz gráfica se instaló

Python con un entorno virtual, y las librerías necesarias para realizar la inferencia, evaluar las métricas y visualizar los resultados.

En una primera aproximación se intentó forzar la instalación de PyTorch en el entorno para hacer una comparación, con el mismo hardware, de los distintos modelos. No obstante, pese a los varios intentos utilizando versiones precompiladas no oficiales [43] o compilando la librería manualmente, no fue posible la instalación de este entorno, lo que demuestra la superior portabilidad de TF en entornos limitados.

Para el procesado de las imágenes, como extraerlas de una cámara o emplearlas en el código, se hace uso del framework OpenCV, así como de su función de supresión de no-máximos.

5.3. Implementación del pipeline de inferencia en TensorFlow-Lite

Se desarrolló en Python un simulador del flujo de trabajo del proceso de inferencia. Dicho flujo se inicia con la carga de la imagen a inferir y continua con su preprocesado al formato esperado por el modelo, la inferencia con el modelo, la extracción del el tensor resultante y la realización de un post-procesado para obtener las coordenadas en pixeles absolutos de las cajas limitadoras del objeto identificado y la predicción final, mediante el algoritmo de supresión de máximos (NMS).

Una vez definido el flujo de procesado, se desarrollaron 3 programas diferentes: implementación de la inferencia sobre imágenes de una carpeta, inferencia con imágenes provenientes de una webcam y un último código para depuración, obteniendo rangos de valores en los modelos o la visualización de los valores de los tensores post-inferencia.

5.3.1. Algoritmo de supresión de máximos (NMS)

El algoritmo de supresión de no-máximos consiste en eliminar aquellas detecciones redundantes que se solapan significativamente con otras de mayor confianza, manteniendo únicamente la predicción más relevante para cada objeto detectado [44]. Para ello, se ordenan las predicciones por su puntuación de confianza y, a medida que se recorren, se descartan aquellas cuyo valor de intersección sobre unión (IoU) con otra predicción supere un cierto umbral. Este procedimiento permite reducir los falsos positivos y clarificar la salida del modelo, especialmente en escenarios donde se producen múltiples detecciones sobre un mismo objeto.

5.3.2. Decodificación de YOLOX

El modelo de YOLOX devuelve un tensor de forma 1x8400x12, lo que implica 8400 detecciones y 12 campos por detección. Los 12 campos son:

- Centro de la caja en X (relativo al ”grid”).
- Centro de la caja en Y (relativo al “grid”).
- Ancho de la caja (logarítmico, relativo al “stride”).
- Alto de la caja (logarítmico, relativo al “stride”).
- Confianza de que hay un objeto en esta caja (“objectness score”).
- Probabilidades de pertenencia a cada una de las 7 clases

Los términos “grid” (malla) y “stride” (paso) son fundamentales para comprender el funcionamiento interno de los detectores de objetos modernos, como YOLOX. La imagen de entrada es procesada a través de una red convolucional que genera mapas de características a distintas escalas. Cada uno de estos mapas se divide en una malla o “grid”, donde cada celda es responsable de predecir posibles objetos en una región concreta de la imagen original. El “stride” indica cuánto se ha reducido la resolución de la imagen en cada una de esas escalas. Por ejemplo, un “stride” de 8 implica que cada celda del “grid” representa un bloque de 8x8 píxeles de la imagen de entrada.

Paso (Stride)	Tamaño de la malla	Número de celdas (grid)
8	80x80	6400
16	40x40	1600
32	20x20	400

Tabla 6. *Relación entre stride, tamaño de celda y número de celdas en la decodificación de YOLOX.*

Fuente: [33]

Como se observa en la Tabla 6, en total se forman 8400 celdas, dando lugar a 8400 posibles detecciones distribuidas en tres escalas del mapa de características, correspondientes a los pasos 8, 16 y 32. Cada celda del "grid" predice una posible caja mediante coordenadas relativas. Estas se convierten en coordenadas absolutas multiplicando por el paso y desplazando por la posición del "grid". Además, los valores de anchura y altura están codificadas de forma logarítmica, lo cual permite representar con más estabilidad una gran gama de tamaños posibles.

Por este motivo, su decodificación no es directa, pues es necesario reconstruir la malla en función del índice de la predicción, para posteriormente situar el centro de la caja limitadora en su cuadrícula correspondiente.

Para ello se filtran las predicciones en función de si el mayor resultado de la multiplicación entre la confianza de objeto y la confianza de clase sobrepasa el umbral de confianza especificado. Y las predicciones restantes se procesan por el algoritmo de NMS. El resultado del algoritmo se toma como predicción final.

Mediante las pruebas realizadas en la Raspberry, se ha podido observar como en imágenes donde el objeto ocupa una gran proporción de esta, la caja es casi idéntica a la anotada. No obstante, en las predicciones de áreas menores se observa un desplazamiento del centro de la predicción, a pesar de generar la caja con el área adecuada al objeto. Esto puede ser debido a la pérdida de precisión por la cuantización o a la falta de un factor de corrección en las predicciones de áreas pequeñas, por lo que se espera una muy baja puntación en las métricas de precisión respecto a las predicciones decodificas del modelo.

Por este motivo, para evaluar su rendimiento como modelo en TFLite se dará más importancia a la métrica de sensibilidad, pues su variación respecto a los valores obtenidos en TFLite, serán debidos a la propia conversión más que al algoritmo decodificador.

5.3.3. Decodificación de YOLOv8

El tensor de salida de YOLOv8 tiene la forma 1x11x8400, aunque se transponen los ejes 1 y 2, dando como resultando la forma 1x8400x11, que es más fácil de manejar.

A diferencia de YOLOX, YOLOv8 no genera un valor basado en la confianza de objeto en la imagen, sino que de sus 11 parámetros son:

- Centro de la caja en X (normalizado respecto al tamaño de entrada).
- Centro de la caja en Y (normalizado respecto al tamaño de entrada).
- Ancho de la caja (normalizado respecto al tamaño de entrada).
- Alto de la caja (normalizado respecto al tamaño de entrada).
- Probabilidades de pertenencia a cada una de las 7 clases

Su decodificación se vuelve más sencilla, pues basta con convertir el centro en coordenadas mínimas y máximas, y multiplicar por el tamaño real de entrada, sin necesidad de información adicional como “grids” o “strides”. Y al igual que con YOLOX, se escogen las predicciones cuyos valores de confianza por clase superen el valor del umbral y se someten al algoritmo de NMS.

5.4. Evaluación del rendimiento en Raspberry Pi 4

Tras ser capaz de extraer predicciones sobre las imágenes del conjunto de prueba, se procede a evaluar el rendimiento de los modelos mediante la librería “pycoco”. Para ello se establece un umbral de confianza de 0.01, para permitir el paso de la gran mayoría de predicciones. Después es necesario procesar las aproximadamente 3700 imágenes del subconjunto de prueba, a las cuales no se les aplica ningún redimensionamiento, pues se hace uso de los conjuntos ya

dimensionados para el entrenamiento de YOLOX. Y por último se envían las predicciones al evaluador COCO de la librería.

5.4.1. Precisión en test

De los resultados producidos por el evaluador de COCO, se extraen los mismos campos empleados en el análisis de rendimiento de los modelos en PyTorch. De esta forma se calcula la precisión media en un intervalo IoU de 0.5 a 0.95 para todos los tipos de áreas y un máximo de 100 detecciones por objeto, y la sensibilidad media bajo las mismas condiciones.

YOLOv8

Versión / Métricas	Precisión (AP)	Sensibilidad (AR)
Mediano (640x640)	31.97%	44.39%
Pequeño (640x640)	27.71%	43.51%
Tiny (640x640)	23.02%	35.85%
Tiny (416x416)	24.83%	38.86%
Tiny (192x192)	21.47%	34.61%

Tabla 7. *Resultados con el conjunto de prueba en la RaspberryPi 4 con YOLOv8.*

Fuente: Elaboración propia

YOLOX

Versión / Métricas	Precisión (AP)	Sensibilidad (AR)
Mediano (640x640)	7.60%	30.46%
Pequeño (640x640)	8.43%	31.20%
Nano (640x640)	7.75%	32.63%
Nano (416x416)	9.98%	39.41%
Nano (192x192)	11.33%	42.81%

Tabla 8. *Resultados con el conjunto de prueba en la RaspberryPi 4 con YOLOX.*

Fuente: Elaboración propia

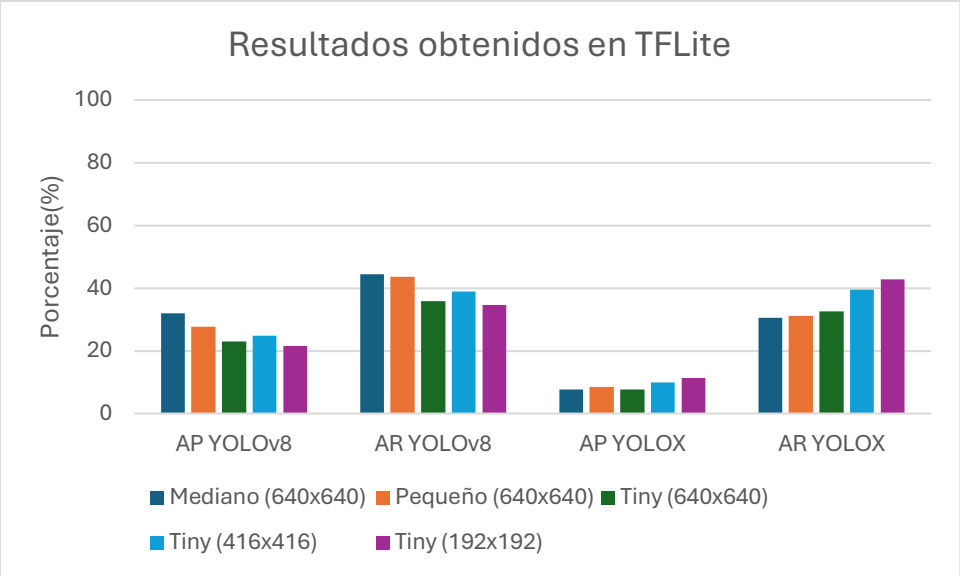


Figura 34. *Resultados obtenidos en TFLite*

Fuente: Elaboración propia

Las diferencias de rendimiento se deben en gran medida al algoritmo utilizado para decodificar los valores de salida de la red. En el caso de YOLOX, la decodificación implementada afecta principalmente a la métrica de precisión; sin embargo, los resultados en términos de sensibilidad muestran que el origen del error reside en el propio algoritmo de decodificación y no en el modelo en sí. Por lo tanto, para obtener mejores resultados es necesario refinar dicho proceso de decodificación adaptando el algoritmo estándar para ajustar las predicciones.

5.4.2. Tiempos de inferencia

Haciendo uso de los códigos de visualización, se calcula el tiempo de inferencia gracias a la librería time de Python. Mediante la función “time()”, se registra una marca temporal antes de la inferencia y otra inmediatamente después. La diferencia entre ambas permite calcular el tiempo total empleado en generar las predicciones. Al ser desarrollado para aplicaciones de video, se presenta el inverso del tiempo de ejecución como imágenes por segundo, para estimar su viabilidad en aplicaciones en tiempo real.

Ejemplo de los resultados de los modelos medianos de ambas familias:

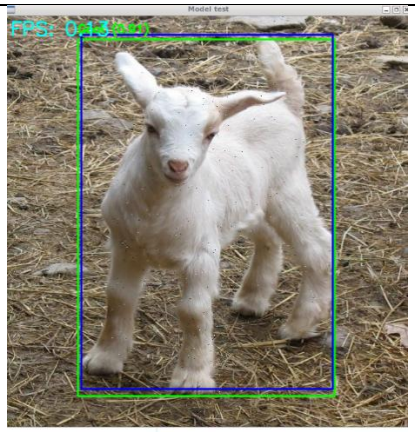
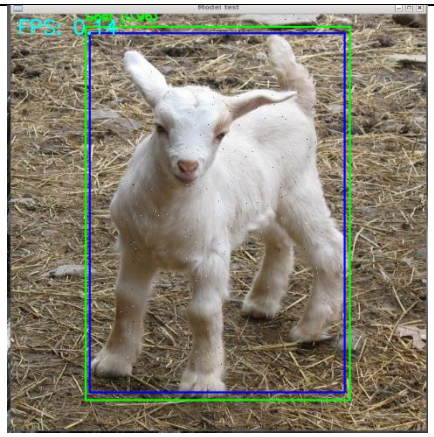
YOLOV8	YOLOX
	

Figura 35. *Comparación entre predicciones de los modelos medianos de YOLOv8 y YOLOX. Cuadro azul: Caja real | Cuadro verde: Caja predicha.*

Fuente: Elaboración propia

Con este método se estiman los tiempos de inferencia de los modelos, que se muestran en la Tabla 9.

Tiempos de inferencia	YOLOv8	YOLOX
Mediano (640x640)	7,69 s	7,14 s
Pequeño (640x640)	3,45 s	3,22 s
Nano/Tiny (640x640)	1,61 s	2,22 s
Nano/Tiny (416x416)	770 ms	970 ms
Nano/Tiny (192x192)	340 ms	356 ms

Tabla 9*. Comparación de los tiempos de inferencia por imagen para cada modelo.*

Fuente: Elaboración propia

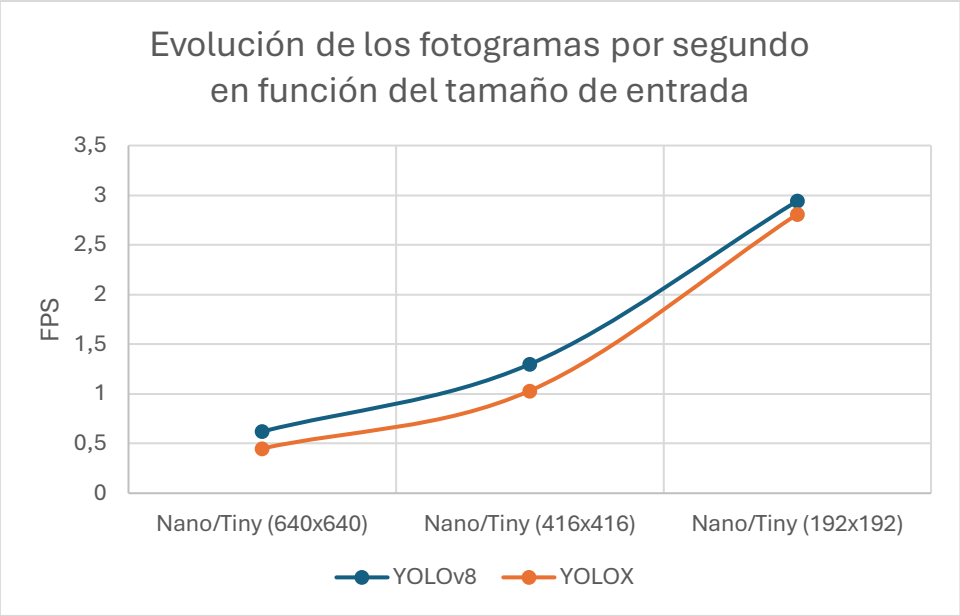


Figura 36. *Evolución de los fotogramas por segundo en función del tamaño de entrada en los modelos más ligeros.*

Fuente: Elaboración propia

De los resultados se concluye que YOLOv8 es, además de más preciso y con mayor sensibilidad, más veloz.

La plataforma de Edge-Impulse, permite realizar una estimación de los recursos hardware que consumirá el modelo. Para ello se ha empleado el modelo “nano” de la familia de YOLOv8, por ser el de menor tamaño entre todos los modelos estudiados (en su versión con imágenes de 640x640 pixeles) para la implementación en hardware.

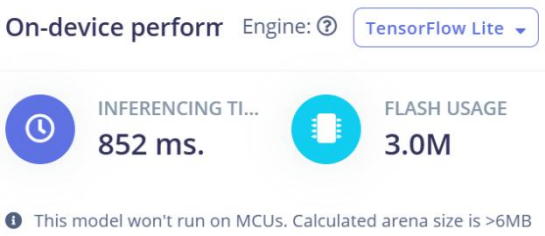
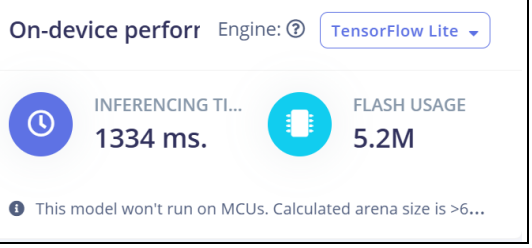
YOLOv8	YOLOX
	

Figura 37. *Comparación de la estimación de tiempos de inferencia por Edge-Impulse.*

Fuente: Elaboración propia

A pesar de que el modelo ocupa menos de 6 MB en su forma pura, es indispensable que todo su encapsulado (como el producido en formato uf2) también se mantenga por debajo de ese umbral. Tal como se ha comprobado, este límite puede superarse fácilmente, lo que genera advertencias de la plataforma relacionadas con el tamaño de la "arena", es decir, la región de memoria reservada para la ejecución del modelo en dispositivos embebidos.

Durante la obtención de métricas en su framework de origen, en el caso de YOLOX se detallaba el tiempo requerido para la inferencia. Para el modelo nano con resolución de entrada de 640x640 píxeles, este tiempo fue de aproximadamente 13.8 ms, incluyendo tanto el tiempo de procesado de la imagen por la red (forward time) como la supresión de no-máximos (NMS). No obstante, este valor fue obtenido utilizando PyTorch en un procesador Intel Xeon Gold con frecuencia máxima de 3.7 GHz, un entorno muy superior al de despliegue real.

En contraste, en el entorno de inferencia final, una Raspberry Pi 4 cuyo sistema operativo soporta una arquitectura ARMv7 de 32 bits y frecuencia máxima de 1.5 GHz, el tiempo medido por imagen se elevó hasta los 356 ms. Esta diferencia resalta las limitaciones impuestas por el hardware, pero también pone en valor el proceso de optimización realizado: a pesar de la gran diferencia en capacidad de cómputo, el modelo ha podido ser ejecutado en una plataforma de recursos limitados gracias a su conversión a TensorFlow Lite y cuantización INT8, sacrificando parte de la precisión de las predicciones, pero manteniendo la funcionalidad completa del sistema.

Capítulo 6. Conclusiones y líneas futuras

6.1. Conclusiones generales

Como resultado de la comparación entre modelos, se puede concluir que el modelo YOLOv8 es capaz de superar las prestaciones de YOLOX, siendo más ligero y rápido, por su enfoque e implementado con operaciones más recientes.

La mayor limitación de YOLOv8 es la falta de soporte, como la ausencia de documentación técnica, y su restrictiva licencia de uso. No obstante, se ha comprobado en este trabajo como su desarrollo puede dar lugar a una nueva generación de modelos de detección de objetos ligera, rápida y fiable.

6.1.1. Efectos observados debidos a la conversión de Frameworks

Los modelos en TensorFlow-Lite son mucho más ligeros que en su versión para PyTorch, la conversión de formatos disminuye a la mitad el tamaño de archivo y son capaces de generar predicciones en un tiempo similar.

La pérdida significativa de rendimiento del modelo TFLite refleja el coste de aplicar técnicas de compresión extrema en modelos que originalmente no fueron diseñados para su ejecución en entornos tan restringidos. No obstante, su implementación demuestra su potencial y futura viabilidad.

Es importante añadir, como una vez obtenidas las predicciones en Tflite, el algoritmo de decodificación implementado afecta directamente a la calidad de los resultados, siendo necesaria una mayor comprensión del proceso de inferencia por red para ajustar la decodificación corrigiendo las descentralizaciones provocadas por la pérdida de precisión.

6.1.2. Efectos observados debidos al número de parámetros

Dentro de cada familia de modelos, se observa que, como era de esperar, los modelos más pesados ofrecen mejores resultados en cuanto a precisión y sensibilidad (recall).

En la versión de PyTorch, el modelo más pesado (ocupando 50 MB) alcanza un recall del 93,24 %, mientras que el modelo más ligero (de 6 MB) obtiene un 87,84 %. De forma análoga, en la versión cuantizada para TensorFlow Lite, el modelo más pesado (de 25 MB) logra un recall del 44,39 %, frente al 35,85 % del modelo ligero (de 3 MB). La diferencia entre valores es constante por lo que se puede afirmar como ocupando 8 veces más tamaño, solo se logra mejorar las métricas en aproximadamente un 5%. Por ello se consideró la pérdida relativa de rendimiento entre modelos asumible, sobre todo si se prioriza la reducción del tamaño para su implementación en dispositivos con recursos limitados.

6.1.3. Efectos observados debidos al tamaño de entrada

La diferencia en rendimiento entre los modelos YOLOX-tiny y YOLOv8-nano frente a distintas resoluciones de entrada es prácticamente inapreciable, salvo en el caso de las imágenes de 192x192 píxeles, donde la pérdida de información afecta visiblemente a la capacidad de los modelos para extraer patrones relevantes. Esta resolución, si bien mejora los tiempos de inferencia, lo hace a costa de la calidad de las predicciones.

Asimismo, se ha demostrado que el tamaño de entrada óptimo para esta aplicación es de 416×416. Esto se debe a que las redes estaban originalmente entrenadas con este tamaño, por lo que parten de una base más adecuada para aprender nuevas características, lo que se traduce en predicciones más precisas. Este hallazgo resulta interesante, ya que muestra que un mayor tamaño de entrada o más información visual no garantiza necesariamente mejores resultados.

Por otro lado, la variación del tiempo de inferencia es significativamente más sensible al tamaño de entrada. A modo ilustrativo, con imágenes de 192x192 píxeles se alcanzan casi 3 FPS, mientras que con 416x416 se reduce ligeramente por encima de 1 FPS, y con 640x640 cae hasta 0.5 FPS.

Cabe destacar que, para aplicaciones de tiempo real no críticas, como la monitorización de fauna en entornos naturales mediante cámaras, no se requiere alcanzar los habituales 30 FPS de vídeo continuo. Basta con realizar inferencias en una ventana temporal razonable, lo cual permita detectar animales que atraviesen el campo de visión del sistema con una latencia aceptable.

En este sentido, los resultados obtenidos con los modelos YOLOX-tiny y YOLOv8-nano demuestran que es técnicamente viable su implementación en sistemas embebidos de bajo consumo sin necesidad de recursos de hardware avanzados.

6.2. Desafíos superados

A lo largo de este Trabajo de Fin de Grado, se ha investigado el estado del arte de modelos de visión artificial ligeros. Además, se ha trabajado con recursos (como modelos, repositorios y documentación) de empresas como SeedStudio, realizando ingeniería inversa de los modelos proporcionados, pero no publicados, para analizar las características de la tecnología comercial actual. Por otro lado, se han adaptado conjuntos de datos de diferentes formatos con los que se han entrenado 10 modelos. También se ha definido una metodología para la conversión de modelos entre frameworks, y se ha presentado el proceso de modificación de grafos para maximizar compatibilidad. Por último, se ha prototipado un modelo en un sistema embebido de arquitectura ARM de 32 bits, decodificando las salidas de los modelos y obteniendo las métricas estándar.

6.3. Limitaciones encontradas

La principal limitación de este proyecto ha sido la falta de documentación y herramientas públicamente disponible en diferentes etapas del proyecto.

En el caso de la empresa SeedStudio, las hojas de características de sus productos carecían de datos importantes, como los valores de memoria disponibles. En placas de interés para este proyecto, como la VisionAi Grove, sus herramientas de desarrollo solo permiten a la captura de datos hacia la memoria SD, no existiendo herramientas disponibles que permitan la modificación de su firmware para introducir modelos diferentes a los de su galería. Además, no existe información adecuada sobre la extensión de las capacidades de las herramientas SDK.

Por otro lado, existe una muy limitada disposición pública de modelos de TFLite. La gran mayoría de modelos públicos se encuentran en formato PyTorch, siendo la cantidad relativa de modelos de TensorFlow pequeña y aún menor en el caso de TFLite.

Las limitaciones de ONNX, sumado a la falta de modelos en TFLite, hace que no siempre sea posible la exportación de modelos de PyTorch a onnx (formato descriptor de redes neuronales). Este fue el caso de la red nanodet, competidora de YOLO, que se evaluó como opción en este proyecto.

Además, la cuantización y operaciones de TensorFlow no es un proceso transparente y bien documentado. Las explicaciones más detalladas disponibles provienen principalmente de blogs técnicos escritos por desarrolladores, más que de documentación oficial del framework.

6.4. Líneas de investigación futuras

Tras este proyecto, la línea de investigación futura es la referida a la modificación del grafo resultante de YOLOv8 para no solo ser capaz de implementarse en TFLite (como se ha conseguido en este trabajo) sino para añadir los nodos necesarios para su decodificación.

Además de estudiar los efectos de un proceso de destilación o de reducción de número de parámetros para obtener un modelo más ligero que la versión “tiny”, cuyo tamaño sea inferior a los 2 MB para así poder implementarlo en dispositivos como los de la familia SeedStudio.

Bibliografía

[1] N. Maslej, «Artificial Intelligence Index Report 2025», *Artif. Intell.*, 2025.

[2] Z. Zou, K. Chen, Z. Shi, Y. Guo, y J. Ye, «Object Detection in 20 Years: A Survey», 18 de enero de 2023, *arXiv*: arXiv:1905.05055. doi: 10.48550/arXiv.1905.05055.

[3] I. Santos, V. Santos, y T. Infinita, «La visión artificial y los campos de aplicación artificial», *Research Gate*, vol. 1, pp. 98-108, 2015, doi: 10.32645/26028131.76.

[4] L. Marketing, «PyTorch vs. TensorFlow: Full Overview 2025 Guide», Lazy Programmer. Accedido: 20 de julio de 2025. [En línea]. Disponible en: https://lazyprogrammer.me/PyTorch-vs-TensorFlow/

[5] A. B. Amjoud y M. Amrouch, «Object Detection Using Deep Learning, CNNs and Vision Transformers: A Review», *IEEE Access*, vol. 11, pp. 35479-35516, 2023, doi: 10.1109/access.2023.3266093.

[6] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, y S. Belongie, «Feature Pyramid Networks for Object Detection», 19 de abril de 2017, *arXiv*: arXiv:1612.03144. doi: 10.48550/arXiv.1612.03144.

[7] Z. Ge, S. Liu, F. Wang, Z. Li, y J. Sun, «YOLOX: Exceeding YOLO Series in 2021», 6 de agosto de 2021, *arXiv*: arXiv:2107.08430. doi: 10.48550/arXiv.2107.08430.

[8] «A Review on YOLOv8 and Its Advancements», en *Algorithms for Intelligent Systems*, Singapore: Springer Nature Singapore, 2024, pp. 529-545. doi: 10.1007/978-981-99-7962-2_39.

[9] «14.4. Anchor Boxes — Dive into Deep Learning 1.0.3 documentation». Accedido: Julio de 2025. [En línea]. Disponible en: https://d2l.ai/chapter_computer-vision/anchor.html

[10] «Cálculo de la Precisión Promedio y mAP», HackMD. Accedido: Julio de 2025. [En línea]. Disponible en: https://hackmd.io/@api-conabio-ml/SyuWOKESU

[11] D. Wei, «Essential Math for Machine Learning: Confusion Matrix, Accuracy, Precision, Recall, F1-Score», Medium. Accedido: Julio de 2025. [En línea]. Disponible en: https://medium.com/@weidagang/demystifying-precision-and-recall-in-machine-learning-6f756a4c54ac

[12] «COCO Evaluation metrics explained — Picsellia». Accedido: 12 de julio de 2025. [En línea]. Disponible en: https://www.picsellia.com/post/coco-evaluation-metrics-explained

[13] Ultralytics, «YOLOv8». Accedido: Julio de 2025. [En línea]. Disponible en: https://docs.ultralytics.com/es/models/yolov8

[14] R. Varghese y S. M., «YOLOv8: A Novel Object Detection Algorithm with Enhanced Performance and Robustness», en *2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)*, Chennai, India: IEEE, abr. 2024, pp. 1-6. doi: 10.1109/adics58448.2024.10533619.

[15] «(PDF) What is a framework? Understanding their purpose, value, development and use», *ResearchGate*, doi: 10.1007/s13412-023-00833-w.

[16] «Ultralytics License». Accedido: Julio de 2025. [En línea]. Disponible en: https://www.ultralytics.com/license

[17] «(PDF) TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems», ResearchGate. Accedido: Julio de 2025. [En línea]. Disponible en: https://www.researchgate.net/publication/301839500_TensorFlow_Large-Scale_Machine_Learning_on_Heterogeneous_Distributed_Systems

[18] «(PDF) PyTorch: An Imperative Style, High-Performance Deep Learning Library», ResearchGate. Accedido: Julio de 2025. [En línea]. Disponible en: https://www.researchgate.net/publication/337756689_PyTorch_An_Imperative_Style_High-Performance_Deep_Learning_Library

[19] «Facebook and Microsoft introduce new open ecosystem for interchangeable AI frameworks - Meta Research», Meta Research. Accedido: Julio de 2025. [En línea]. Disponible en: https://research.facebook.com/blog/2017/9/facebook-and-microsoft-introduce-new-open-ecosystem-for-interchangeable-ai-frameworks/

[20] Y. Zhao *et al.*, «ONNXExplainer: an ONNX Based Generic Framework to Explain Neural Networks Using Shapley Values», 3 de octubre de 2023, *arXiv*: arXiv:2309.16916. doi: 10.48550/arXiv.2309.16916.

[21] «TensorFlow Lite», TensorFlow. Accedido: Julio de 2025. [En línea]. Disponible en: https://www.TensorFlow.org/lite/guide?hl=es-419

[22] «tf.lite.TFLiteConverter | TensorFlow v2.16.1», TensorFlow. Accedido: Julio de 2025. [En línea]. Disponible en: https://www.TensorFlow.org/api_docs/python/tf/lite/TFLiteConverter

[23] «TensorFlow Lite C++ API Reference | Google AI Edge», Google AI for Developers. Accedido: Julio de 2025. [En línea]. Disponible en: https://ai.google.dev/edge/api/tflite/cc

[24] «TorchScript — PyTorch 2.7 documentation». Accedido: Julio de 2025. [En línea]. Disponible en: https://docs.PyTorch.org/docs/stable/jit.html

[25] «ONNX Runtime», onnxruntime. Accedido: Julio de 2025. [En línea]. Disponible en: https://onnxruntime.ai/docs/

[26] «Understanding the TFLite Quantization Approach (Part I)», CLIKA Compressed Blog. Accedido: Julio de 2025. [En línea]. Disponible en: https://blog.clika.io/understanding-the-TensorFlow-lite-quantization-approach-part-i/

[27] «Model Zoo — YOLOX 0.2.0 documentation». Accedido: Julio de 2025. [En línea]. Disponible en: https://yolox.readthedocs.io/en/latest/model_zoo.html

[28] F. Zhuang *et al.*, «A Comprehensive Survey on Transfer Learning», 23 de junio de 2020, *arXiv*: arXiv:1911.02685. doi: 10.48550/arXiv.1911.02685.

[29] Kocaeli University, *ProjectB Dataset*. [RoboFlow]. Disponible en: https://universe.roboflow.com/kocaeli-university-hgoxr/projectb-5ne4k

[30] project-qnjnj, *Proje_B Dataset*. (abril de 2023). [Roboflow]. Disponible en: https://universe.roboflow.com/project-qnjnj/proje_b

[31] «Train Validation Test Dataset and K-Fold Validation». Accedido: Julio de 2025. [En línea]. Disponible en: https://velog.io/@iguv/Train-Validation-Test-Dataset-and-K-Fold-Validation

[32] «Interpolación de Imágenes», n.º 2.

[33] «GitHub - Megvii-BaseDetection/YOLOX: YOLOX is a high-performance anchor-free YOLO, exceeding yolov3~v5 with MegEngine, ONNX, TensorRT, ncnn, and OpenVINO supported. Documentation: https://yolox.readthedocs.io/». Accedido: Julio de 2025. [En línea]. Disponible en: https://github.com/Megvii-BaseDetection/YOLOX

[34] ultralytics, «ultralytics/ultralytics/engine/trainer.py at main · ultralytics/ultralytics», GitHub. Accedido: Julio de 2025. [En línea]. Disponible en: https://github.com/ultralytics/ultralytics/blob/main/ultralytics/engine/trainer.py

[35] «Usando el formato SavedModel | TensorFlow Core», TensorFlow. Accedido: Julio de 2025. [En línea]. Disponible en: https://www.TensorFlow.org/guide/saved_model?hl=es-419

[36] «Netron». Accedido: Julio de 2025. [En línea]. Disponible en: https://netron.app/

[37] «tf.split | TensorFlow v2.16.1». Accedido: Julio de 2025. [En línea]. Disponible en: https://www.TensorFlow.org/api_docs/python/tf/split

[38] «tf.slice | TensorFlow v2.16.1», TensorFlow. Accedido: Julio de 2025. [En línea]. Disponible en: https://www.TensorFlow.org/api_docs/python/tf/slice

[39] *microsoft/uf2*. (11 de julio de 2025). JavaScript. Microsoft. Accedido: Julio de 2025. [En línea]. Disponible en: https://github.com/microsoft/uf2

[40] «yolov5-swift/uf2conv.py at master · Seeed-Studio/yolov5-swift · GitHub». Accedido: Julio de 2025. [En línea]. Disponible en: https://github.com/Seeed-Studio/yolov5-swift/blob/master/uf2conv.py

[41] R. P. Ltd, «Raspberry Pi 4 Model B specifications», Raspberry Pi. Accedido: Julio de 2025. [En línea]. Disponible en: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[42] «raspberry-pi-zero-2-w-product-brief.pdf». [En línea]. Disponible en: https://datasheets.raspberrypi.com/rpizero2/raspberry-pi-zero-2-w-product-brief.pdf

[43] K. Yamazaki, *Kashu7100/PyTorch-armv7l*. (26 de marzo de 2025). Accedido: Julio de 2025. [En línea]. Disponible en: https://github.com/Kashu7100/PyTorch-armv7l

[44] J. Hosang, R. Benenson, y B. Schiele, «Learning non-maximum suppression», 9 de mayo de 2017, *arXiv*: arXiv:1705.02950. doi: 10.48550/arXiv.1705.02950.