



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

 etsinf

Escola Tècnica
Superior d'Enginyeria
Informàtica

Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Detección de defectos en objetos en movimiento mediante Redes Neuronales Convolucionales con optimizaciones específicas para hardware NVIDIA

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Haro Armero, Abel

Tutor: Flich Cardo, José
López Rodríguez, Pedro Juan

Curso 2024-2025

Resum

Aquest Treball de Fi de Grau aborda el repte de la detecció automàtica de defectes en objectes que es desplacen en entorns industrials, com ara les línies de producció. Es presenta el disseny i la implementació d'un sistema complet de visió artificial basat en tècniques d'aprenentatge profund.

El nucli del sistema són els models de Xarxes Neuronals Convolucionals (CNN), específicament unes variants de l'arquitectura YOLO, entrenades per identificar i localitzar amb precisió diversos tipus de defectes en temps real. Per fer front a les restriccions computacionals i energètiques pròpies dels sistemes embebuts, els models s'han implementat sobre acceleradors hardware de baix consum de la sèrie NVIDIA Jetson. S'han aplicat tècniques avançades d'optimització, incloent la conversió a TensorRT i l'exploració de diferents precisions numèriques, per maximitzar la velocitat d'inferència (FPS) i minimitzar el consum energètic (Watts) sense comprometre significativament la precisió de la detecció (mAP).

El treball inclou una anàlisi exhaustiva del rendiment del sistema sota diverses configuracions de hardware i software, demostrant la viabilitat d'aplicar solucions d'intel·ligència artificial d'alt rendiment en escenaris industrials amb recursos limitats.

Paraules clau: Xarxes Neuronals Convolucionals, Detecció de defectes, Visió per computador, NVIDIA Jetson, Temps real, Optimització energètica

Resumen

Este trabajo de fin de grado aborda el desafío de la detección automática de defectos en objetos que se desplazan en entornos industriales, como líneas de producción. Se presenta el diseño y la implementación de un sistema completo de visión artificial basado en técnicas de aprendizaje profundo.

El núcleo del sistema son los modelos de Redes Neuronales Convolucionales (CNN), específicamente unas variantes de la arquitectura YOLO, entrenadas para identificar y localizar con precisión diversos tipos de defectos en tiempo real. Para hacer frente a las restricciones computacionales y energéticas propias de los sistemas embebidos, los modelos se han implementado sobre aceleradores hardware de bajo consumo de la serie NVIDIA Jetson. Se han aplicado técnicas avanzadas de optimización, incluyendo la conversión a TensorRT y la exploración de diferentes precisiones numéricas, para maximizar la velocidad de inferencia (FPS) y minimizar el consumo energético (Watts) sin comprometer significativamente la precisión de la detección (mAP).

El trabajo incluye un análisis exhaustivo del rendimiento del sistema bajo diversas configuraciones de hardware y software, demostrando la viabilidad de aplicar soluciones de inteligencia artificial de alto rendimiento en escenarios industriales con recursos limitados.

Palabras clave: Redes Neuronales Convolucionales, Detección de defectos, Visión por computador, NVIDIA Jetson, Tiempo real, Optimización energética

Abstract

This Bachelor's thesis addresses the challenge of automatic defect detection in moving objects within industrial environments, such as production lines. It presents the design and implementation of a complete computer vision system based on deep learning techniques.

The core of the system consists of Convolutional Neural Network (CNN) models, specifically variants of the YOLO architecture, trained to accurately identify and locate various types of defects in real-time. To address the computational and energy constraints typical of embedded systems, the models have been implemented on low-power hardware accelerators from the NVIDIA Jetson series. Advanced optimization techniques, including conversion to TensorRT and exploration of different numerical precisions, have been applied to maximize inference speed (FPS) and minimize energy consumption (Watts) without significantly compromising detection accuracy (mAP).

The thesis includes a comprehensive performance analysis of the system under various hardware and software configurations, demonstrating the feasibility of applying high-performance artificial intelligence solutions in industrial scenarios with limited resources.

Key words: Convolutional Neural Networks, Defect Detection, Computer Vision, NVIDIA Jetson, Real-Time, Energy Optimization

Índice general

Índice general	v
Índice de figuras	vii
Índice de tablas	viii
1 Introducción	1
1.1 Motivación	3
1.2 Objetivos	4
1.3 Estructura de la memoria	4
1.4 Colaboraciones	5
2 Conceptos previos	7
2.1 Fundamentos y avances en redes neuronales para visión artificial	7
2.1.1 Fundamentos de la inteligencia artificial	7
2.1.2 Tareas fundamentales en visión por computador	8
2.1.3 Arquitectura y funcionamiento de las CNN	9
2.1.4 Detectores de dos etapas	14
2.1.5 Detectores de una etapa	15
2.1.6 Métricas de evaluación de modelos de detección de objetos	17
2.2 Aceleradores de procesamiento gráfico	18
2.2.1 Limitaciones del hardware tradicional	19
2.2.2 Arquitectura y funcionamiento de las GPUs	20
2.2.3 Serie Jetson: dispositivos para IA de bajo consumo	21
2.2.4 TensorRT	23
2.3 Seguimiento de objetos en tiempo real	24
2.3.1 Introducción al seguimiento de objetos	24
2.3.2 BYTETrack	26
2.3.3 Métricas de evaluación en seguimiento de objetos múltiples	28
3 Diseño e implementación de la solución	31
3.1 Análisis del problema	31
3.2 Entrenamiento de los modelos	33
3.3 Descripción del sistema	36
3.4 Diseño de las etapas del sistema	37
3.4.1 Captura de imágenes	37
3.4.2 Inferencia	38
3.4.3 Seguimiento	38
3.4.4 Escritura de resultados	39
3.5 Segmentación de las etapas del sistema	40
3.5.1 Secuencial	40
3.5.2 Segmentación en hilos	41
3.5.3 Segmentación en procesos	42
3.5.4 Segmentación heterogénea	44
3.5.5 Segmentación basada en procesos con memoria compartida	45
4 Evaluación de la solución	49

4.1	Metodología de evaluación y métricas de rendimiento	49
4.2	Variación de la configuración del sistema	51
4.2.1	Cantidad de objetos	51
4.2.2	Tipo de segmentación	57
4.2.3	Modelo y talla	59
4.2.4	Precisión numérica y acelerador de inferencia	61
4.2.5	Modo de energía y cores de la CPU	64
4.2.6	Dispositivos Jetson	65
4.3	Evaluación del seguimiento de objetos	66
5	Prueba de concepto	69
5.1	Construcción del entorno	69
6	Conclusiones	71
6.1	Objetivos alcanzados y dificultades	71
6.2	Aprendizaje	72
6.3	Relación con los estudios cursados	73
6.4	Trabajo futuro	73
Bibliografía		75
<hr/>		
Apéndices		
A	Objetivos de desarrollo sostenible	77
B	Código fuente	79

Índice de figuras

1.1	Evolución del interés público en inteligencia artificial según datos de Google Trends (2020-2025)	1
1.2	Proyección del consumo eléctrico de los centros de datos en el mundo	2
1.3	Esquema del sistema de visión artificial propuesto	3
2.1	Estructura de un perceptrón multicapa (MLP)	8
2.2	Ejemplo de HOG aplicado a una imagen	8
2.3	Tareas fundamentales en visión por computador	9
2.4	Relación entre Machine Learning, Deep Learning, CNN, Computer Vision y Human Vision	10
2.5	Operación de convolución sobre los pixeles de una imagen	10
2.6	Proceso de convolución aplicado a una imagen de un autobús	11
2.7	Ejemplo de operación de max-pooling con una ventana de 2×2 y un <i>stride</i> de 2	12
2.8	Ejemplo de una CNN simple	13
2.9	Proceso de búsqueda selectiva aplicado a una imagen	14
2.10	Arquitectura de R-CNN	15
2.11	Ejemplo de detección de objetos utilizando YOLO	16
2.12	Arquitectura de YOLO	17
2.13	Evolución histórica de las características de los microprocesadores (1970–2020)	19
2.14	Comparativa de arquitecturas CPU y GPU	20
2.15	Módulos Jetson de NVIDIA	22
2.16	Ejemplo de flujo de trabajo de optimización con TensorRT	24
2.17	Diagrama de flujo del filtro de Kalman	25
2.18	Comparativa de rendimiento de BYTETrack con otros algoritmos de seguimiento	27
2.19	Ejemplo de detección y seguimiento de objetos utilizando BYTETrack	27
3.1	Ejemplo de entorno simulado con canicas	32
3.2	Ejemplo de anotación de imágenes utilizando CVAT	34
3.3	Curvas de pérdida de entrenamiento y validación para las distintas tallas de modelos YOLOv5, YOLOv8 y YOLO11	36
3.4	Figura del sistema propuesto	37
3.5	Ejemplo de salida del sistema	40
3.6	Diagrama de flujo del sistema sin segmentar	41
3.7	Diagrama de flujo del sistema segmentado en hilos	42
3.8	Diagrama de flujo del sistema segmentado en procesos	43
3.9	Diagrama de flujo del sistema segmentado	43
3.10	Diagrama de flujo del sistema segmentado en diferentes unidades de procesamiento	45
3.11	Diagrama de flujo del sistema segmentado en procesos con memoria compartida	46
3.12	Ejemplo de buffer circular	46

4.1	Cantidad de objetos en los vídeos de prueba	52
4.2	FPS por fotograma en función de la cantidad de objetos para los cuatro vídeos de prueba	52
4.3	Ejecución temporal de las etapas del sistema durante el vídeo de prueba con carga variable	53
4.4	Ejecución temporal de las etapas del sistema durante el vídeo de prueba 2 (carga media y constante)	54
4.5	FPS y cantidad de objetos en función del tiempo para el vídeo de carga variable	55
4.6	Tiempo de ejecución de la etapa de seguimiento en función de la cantidad de objetos	55
4.7	Tiempo de ejecución de la etapa de escritura en función de la cantidad de objetos	56
4.8	Tiempos de ejecución de la etapa de inferencia para los diferentes modelos y tallas	60
4.9	Exportación del modelo YOLO11n con TensorRT a FP16 para su ejecución en la DLA	62
4.10	Tiempos de ejecución de la etapa de inferencia para las diferentes precisiones en GPU con TensorRT	63
5.1	Planos de la cinta transportadora	69
5.2	Cinta transportadora construida para la prueba de concepto	70
B.1	Diagrama de clases del objeto DetectionTrackingPipeline	79
B.2	Diagrama de clases del programa principal	94
B.3	Diagrama de clases del objeto TrackerWrapper	104
B.4	Diagrama de clases del objeto SharedCircularBuffer	106

Índice de tablas

2.1	Comparativa técnica entre diferentes modelos NVIDIA Jetson	22
3.1	Análisis comparativo de variantes de YOLO (v5, v8, v11) indicando tamaño, parámetros, latencias CPU/GPU y la GPU específica utilizada para la medición de latencia GPU	34
3.2	Comparativa del rendimiento de los modelos YOLOv5, YOLOv8 y YOLO11 en términos de tiempo de entrenamiento, precisión, recall y mAP . .	35
4.1	Resultados del experimento con distintas cantidades de objetos	54
4.2	Resultados del experimento con distintos tipos de segmentación a máxima capacidad	58
4.3	Resultados del experimento con distintos tipos de segmentación a 30 fps .	59
4.4	Resultados del experimento con distintos modelos y tallas a máxima capacidad con un vídeo de carga alta y constante	60
4.5	Resultados del experimento con distintos modelos y tallas a 30 fps	61
4.6	Resultados del experimento con distintos modelos y tallas a máxima capacidad con un vídeo de carga alta y constante	62

4.7 Resultados del experimento con distintos modelos y tallas a 30 fps con un vídeo de carga alta y constante	63
4.8 Resultados del experimento con distintos modelos y tallas a máxima capacidad con un vídeo de carga alta y constante	64
4.9 Resultados del experimento con distintos modelos y tallas a 30 fps con un vídeo de carga alta y constante	65
4.10 Resultados del experimento con distintos dispositivos Jetson a máxima capacidad con un vídeo de carga alta y constante	65
4.11 Resultados del experimento con distintos dispositivos Jetson a 30 fps con un vídeo de carga alta y constante	66

CAPÍTULO 1

Introducción

Durante los últimos años, la Inteligencia Artificial (IA) ha experimentado un crecimiento en popularidad sin precedentes, transformando nuestra capacidad tecnológica con herramientas revolucionarias. Este avance ha sido impulsado por la disponibilidad de grandes volúmenes de datos, el desarrollo de algoritmos avanzados y las mejoras significativas en el hardware de procesamiento, que han permitido a las máquinas aprender y adaptarse a situaciones complejas. Algunos campos destacados de aplicación incluyen el procesamiento del lenguaje natural, la visión por computador y la robótica. En particular, la visión por computador ha visto un auge significativo, con aplicaciones en áreas como la seguridad, la medicina y la automoción. Esta creciente popularidad por el mundo de la IA se refleja en la evolución del interés público en ella, como muestra la Figura 1.1.



Figura 1.1: Evolución del interés público en inteligencia artificial según datos de Google Trends (2020-2025).

El progreso en visión por computador ha sido posible gracias a los avances en Redes Neuronales Convolucionales (CNNs), que han revolucionado la capacidad de los sistemas para detectar y clasificar objetos en imágenes y vídeos con una gran precisión y velocidad.

Estos algoritmos de visión artificial requieren una potencia computacional significativa tanto para su entrenamiento como para su ejecución. Las Unidades Centrales de

Procesamiento (CPUs) tradicionales resultan insuficientes para estas tareas, por lo que la industria ha desarrollado arquitecturas específicas como las Unidades de Procesamiento Gráfico (GPUs)[10, cap. 3, pp. 2-7], Unidades de Procesamiento Tensorial (TPUs)[2] y Aceleradores de Aprendizaje Profundo (DLAs)[25]. Estos componentes están optimizados para ejecutar operaciones de entrenamiento e inferencia de manera eficiente, permitiendo implementar sistemas de visión artificial capaces de procesar información visual en tiempo real. Sin embargo, estos aceleradores suelen presentar un consumo energético elevado, lo que plantea importantes retos de eficiencia y sostenibilidad.

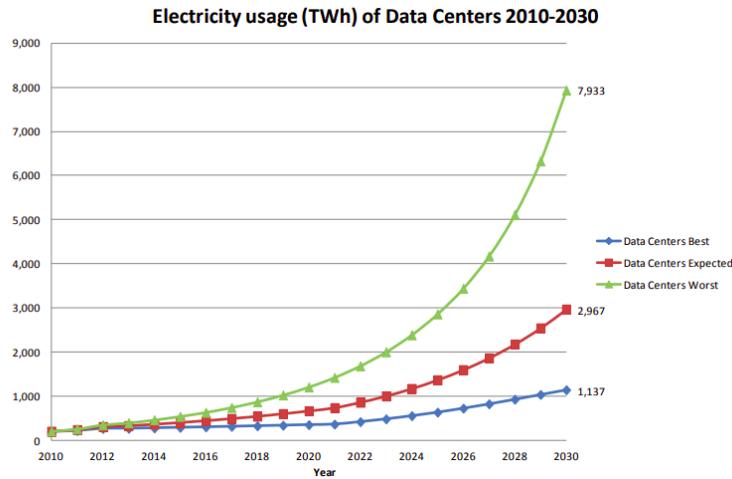


Figura 1.2: Proyección del consumo eléctrico de los centros de datos en el mundo. Extraído de [1, fig. 4, p. 17].

Como se observa en la Figura 1.2, el consumo eléctrico de los centros de datos en el mundo ha ido aumentando de forma exponencial, lo que plantea un desafío significativo para la sostenibilidad del crecimiento tecnológico [1]. En el peor escenario, esta tendencia podría llevar a un incremento insostenible en la huella de carbono del sector tecnológico, mientras que en el mejor de los casos, la adopción de tecnologías eficientes podría moderar el crecimiento. Este aumento del consumo energético no solo afecta a los centros de datos, sino también a los dispositivos embebidos y móviles, donde la eficiencia energética es crucial para prolongar la vida útil de las baterías y reducir el impacto ambiental.

Para enfrentar estos desafíos, se han desarrollado diversas técnicas de optimización y compresión que reducen el tamaño y la complejidad de los modelos neuronales manteniendo su rendimiento. Paralelamente, han surgido arquitecturas hardware específicamente diseñadas para la inferencia de modelos de aprendizaje profundo en entornos con restricciones energéticas. En este contexto, los dispositivos de la serie Jetson de NVIDIA[26] destacan por ofrecer un equilibrio entre alto rendimiento en tareas de IA y un consumo energético contenido, ideal para aplicaciones embebidas de visión artificial.

La combinación de CNNs y aceleradores hardware ha permitido la creación de sistemas de visión artificial que pueden detectar y clasificar objetos en movimiento, lo que es esencial en aplicaciones como la vigilancia, la conducción autónoma y la robótica.

Con todo ello en mente, este trabajo se centra en el desarrollo de un sistema de visión artificial capaz de detectar y clasificar objetos con posibles defectos en movimiento, utilizando CNNs y aceleradores hardware de bajo consumo.



Figura 1.3: Esquema del sistema de visión artificial propuesto.

La Figura 1.3 ilustra el esquema del sistema de visión artificial propuesto. Este sistema se basa en la captura de imágenes de objetos en movimiento, que son procesadas por un modelo de CNN para detectar y clasificar posibles defectos. El modelo se ejecuta en un dispositivo NVIDIA Jetson, optimizado para ofrecer un rendimiento eficiente en términos de velocidad y consumo energético. La implementación del sistema incluye la captura y etiquetado de imágenes, el diseño y entrenamiento del modelo CNN, así como la integración con el hardware NVIDIA para su despliegue en entornos industriales.

1.1 Motivación

Los humanos somos capaces de ver y entender el mundo que nos rodea. Dada una imagen, podemos identificar objetos, reconocer patrones y tomar decisiones basadas en la información visual. Sin embargo, esta capacidad no es innata en las máquinas. La visión por computador es la ciencia que busca dotar a las máquinas de la capacidad de interpretar y comprender imágenes y videos, emulando la forma en que los humanos percibimos el entorno.

Como se mencionó anteriormente, la IA ha revolucionado la forma en que interactuamos con la tecnología. Se ha convertido en una herramienta esencial para aplicar soluciones innovadoras en una amplia gama de campos. En particular, la visión por computador ha demostrado ser un área de gran potencial. También la existencia de dispositivos de bajo consumo, como los de la serie Jetson de NVIDIA, ha permitido llevar la IA a entornos de *edge computing* (cómputo en el borde), donde se acerca el procesamiento de datos a la fuente de información. Esto reduce la latencia y el consumo energético. Con todo esto, se abre un abanico de posibilidades para la implementación de sistemas de visión artificial en aplicaciones industriales.

Centrándose en el ámbito industrial, la detección y clasificación de objetos en movimiento es crucial para optimizar procesos, mejorar la seguridad y aumentar la eficiencia. En la mayoría de los entornos productivos, la detección de defectos se realiza de forma manual, lo que puede ser ineficiente y propenso a errores. La automatización de este proceso mediante sistemas de visión artificial puede reducir costes, aumentar la precisión y mejorar la calidad del producto final.

La motivación de este trabajo radica en la necesidad de desarrollar un sistema de visión artificial capaz de detectar y clasificar objetos en movimiento en un entorno industrial, específicamente en una cinta transportadora.

1.2 Objetivos

El objetivo principal de este trabajo es desarrollar un sistema de visión artificial capaz de detectar y clasificar objetos en movimiento en una cinta transportadora utilizando CNNs y aceleradores hardware de bajo consumo. Para lograr este objetivo, se plantean los siguientes objetivos específicos:

- Realizar un estudio del estado del arte en CNNs, aceleradores hardware de bajo consumo y técnicas avanzadas de optimización para visión artificial.
- Desarrollar un conjunto de datos para el entrenamiento y evaluación del sistema, mediante la captura y etiquetado de imágenes de objetos en movimiento.
- Diseñar, entrenar y validar un modelo de red neuronal convolucional optimizado para la detección y clasificación en tiempo real de defectos en objetos en movimiento.
- Implementar un sistema completo de visión artificial que integre el modelo entrenado con los aceleradores hardware NVIDIA, enfocado en maximizar la eficiencia y minimizar la latencia.
- Analizar los cuellos de botella del sistema, y aplicar técnicas específicas de optimización para mejorar el rendimiento y la eficiencia energética.
- Cuantificar de manera exhaustiva el rendimiento del sistema mediante métricas precisas de exactitud (mean Average Precision (mAP), precisión, recall), latencia (Frames Por Segundo (FPS)) y consumo energético (W, J/inferencia).
- Realizar un análisis comparativo sistemático entre diferentes configuraciones de hardware, software y parámetros de optimización para identificar la combinación que ofrezca el mejor equilibrio entre precisión, velocidad y eficiencia energética.

1.3 Estructura de la memoria

La memoria se estructura en seis capítulos, cada uno dedicado a un aspecto fundamental del trabajo desarrollado:

El **Capítulo 1** introduce el proyecto, detallando la motivación subyacente, los objetivos específicos que se persiguen, la organización general de esta memoria y las colaboraciones para el desarrollo del trabajo.

El **Capítulo 2** sienta las bases teóricas del trabajo. Comienza explorando los fundamentos de la IA y avanza hacia los desarrollos más recientes en CNNs aplicadas a la visión por computador. A continuación, se justifica el uso de aceleradores hardware para tareas de IA, presentando los dispositivos de bajo consumo NVIDIA Jetson. El capítulo concluye con una descripción de los algoritmos de seguimiento de objetos en tiempo real, detallando el funcionamiento de ByteTrack y las métricas de evaluación pertinentes.

El **Capítulo 3** detalla el proceso de diseño e implementación del sistema de visión artificial. Se inicia con un análisis del problema, seguido de la descripción de la recolección de imágenes, su etiquetado y el entrenamiento de los modelos de detección. Concluye con el diseño modular del sistema, especificando sus etapas y las estrategias de segmentación consideradas para su implementación.

El **Capítulo 4** presenta la metodología empleada para evaluar el rendimiento del sistema. Define las métricas clave, describe la configuración de los experimentos y la recolección de datos. Posteriormente, se analizan exhaustivamente los resultados obtenidos, comparando el comportamiento del sistema bajo diversas configuraciones de hardware y software, y evaluando su precisión, velocidad y eficiencia energética. Por último, se analizan las métricas de seguimiento de objetos para evaluar la efectividad del sistema en la detección y seguimiento de objetos en movimiento.

El **Capítulo 5** describe la construcción de un prototipo de cinta transportadora, diseñado para validar el sistema de visión artificial en un entorno que simula condiciones de producción. Se presentan los resultados obtenidos durante esta prueba de concepto.

Finalmente, el **Capítulo 6** resume las principales conclusiones derivadas del trabajo, resaltando los logros alcanzados, las limitaciones identificadas y las posibles líneas de investigación y desarrollo futuras.

1.4 Colaboraciones

El presente Trabajo de Fin de Grado se ha desarrollado en el marco de una beca de colaboración y un periodo de prácticas de empresa, ambos focalizados en la investigación y desarrollo que sustenta este proyecto. Estas actividades se han llevado a cabo en el Grupo de Arquitecturas Paralelas (GAP) de la Universidad Politécnica de Valencia (UPV). El GAP es un grupo de investigación dentro de la universidad, que se dedica al diseño y evaluación de redes de interconexión para sistemas de computación paralela de altas prestaciones, tales como supercomputadores, clústeres y centros de datos.

El Grupo de Arquitecturas Paralelas (GAP) de la Universidad Politécnica de Valencia cuenta con una amplia experiencia en el trabajo sobre redes de interconexión para computadoras paralelas, abarcando desde grandes supercomputadoras, pasando por clústeres de ordenadores personales, usualmente utilizados como servidores, hasta las redes en chip en procesadores multinúcleo. El GAP también ha desarrollado investigación en temas relacionados, como la microarquitectura de procesadores y los protocolos de coherencia de caché. Su investigación abarca también áreas como la optimización de arquitecturas para aplicaciones específicas y la mejora de la eficiencia energética en sistemas computacionales.

Esta colaboración se encuentra estrechamente vinculada con los contenidos y objetivos de la asignatura Sistemas Basados en Deep Learning para la Industria (SDL), una asignatura que forma parte de la mención de ingeniería dentro del Grado en Ingeniería Informática. La oportunidad de colaborar con el GAP ha permitido aplicar de manera práctica y en un contexto de investigación real los conocimientos teóricos y técnicos adquiridos durante la asignatura. Específicamente, ha facilitado la profundización en el diseño, implementación y optimización de sistemas de visión por computador basados en IA, enfrentando desafíos reales relacionados con el rendimiento, la eficiencia y el despliegue en hardware especializado. Esta sinergia entre la formación académica y la investigación aplicada ha Enriquecido significativamente la experiencia de aprendizaje, fomentando el desarrollo de habilidades prácticas avanzadas y una comprensión más profunda de las complejidades inherentes al campo de la visión artificial y el aprendizaje profundo en entornos industriales.

CAPÍTULO 2

Conceptos previos

En este capítulo se describirán los conceptos previos que constituyen la base teórica y técnica de este trabajo. Primero, se examinarán los fundamentos en IA centrándose en las CNNs, desde sus bases hasta los modelos más recientes en detección de objetos. A continuación, se analizarán los aceleradores hardware de bajo consumo, con especial énfasis en la arquitectura y capacidades de los dispositivos NVIDIA Jetson. Posteriormente, se estudiarán los algoritmos de seguimiento de objetos en tiempo real, fundamentales para aplicaciones con elementos en movimiento. Finalmente, se explorará la técnica de Slicing Aided Hyper Inference (SAHI), una metodología avanzada para mejorar la detección de objetos pequeños o densamente agrupados. Este marco teórico permitirá contextualizar adecuadamente la solución propuesta para la detección de defectos en objetos en movimiento.

2.1 Fundamentos y avances en redes neuronales para visión artificial

En esta sección se realizará un estudio de las redes neuronales profundas hasta las CNNs, desde sus fundamentos hasta los modelos más recientes en detección de objetos. Se explicarán los conceptos básicos de las redes neuronales y la evolución de las arquitecturas.

2.1.1. Fundamentos de la inteligencia artificial

La *Inteligencia Artificial* es un campo de estudio que busca desarrollar sistemas capaces de realizar tareas que normalmente requieren inteligencia humana, como el reconocimiento de voz, la toma de decisiones y la comprensión del lenguaje natural. Dentro de este campo, existen diversas subdisciplinas, entre las cuales destacan el *Machine Learning* y el *Deep Learning*.

El *Machine Learning* o Machine Learning (ML) es una rama de la IA que se centra en el desarrollo de algoritmos y modelos que permiten a las máquinas aprender de los datos y realizar predicciones o tomar decisiones sin ser programadas explícitamente. Este enfoque se basa en la idea de que las máquinas pueden identificar patrones y relaciones en grandes conjuntos de datos, lo que les permite generalizar y adaptarse a nuevas situaciones.

El *Deep Learning* o aprendizaje profundo es una rama del ML que utiliza redes neuronales artificiales con múltiples capas para modelar y resolver problemas complejos. Este enfoque permite aprender representaciones jerárquicas de los datos, donde cada capa ex-

trae características cada vez más abstractas. Una de las arquitecturas fundamentales es el Multilayer Perceptron (MLP) o perceptrón multicapa, que consiste en una red de neuronas artificiales organizadas en al menos tres capas: una de entrada, una o más capas ocultas y una capa de salida, como se muestra en la Figura 2.1. En un MLP, cada neurona recibe un conjunto de entradas ponderadas por pesos, aplica una función de activación no lineal a la suma de estas entradas ponderadas, y produce una salida que se transmite a la siguiente capa. Esta estructura permite al Deep Learning abordar tareas complejas en visión por computador, procesamiento del lenguaje natural y otros dominios con un alto grado de precisión.

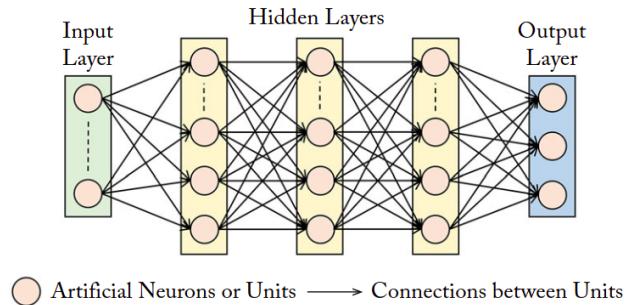


Figura 2.1: Estructura de un perceptrón multicapa (MLP). Extraído de [16, fig. 3.1, p. 32].

2.1.2. Tareas fundamentales en visión por computador

La visión por computador aborda el desafío de permitir que las máquinas interpreten y comprendan el contenido visual de imágenes y videos. Antes del auge del aprendizaje profundo, se empleaban descriptores de características diseñados manualmente, como el Histogram of Oriented Gradients (HOG). HOG captura la forma local de los objetos analizando la distribución de las orientaciones de los gradientes en pequeñas regiones de la imagen (celdas y bloques). La Figura 2.2 muestra una visualización de las características HOG extraídas. Aunque útiles en su momento, estos algoritmos “hechos a mano” presentan limitaciones significativas. En particular, no facilitan el aprendizaje por transferencia, es decir, la reutilización de conocimiento aprendido en tareas previas. Además, la complejidad de estas características está intrínsecamente limitada por la capacidad humana para diseñarlas explícitamente. Estos inconvenientes son superados por los algoritmos de ML de características, como las CNN, que aprenden representaciones relevantes directamente de los datos. Por ello, las CNN han demostrado ser más efectivas en tareas complejas, logrando avances significativos en precisión y eficiencia al aprender automáticamente características a partir de grandes conjuntos de datos.

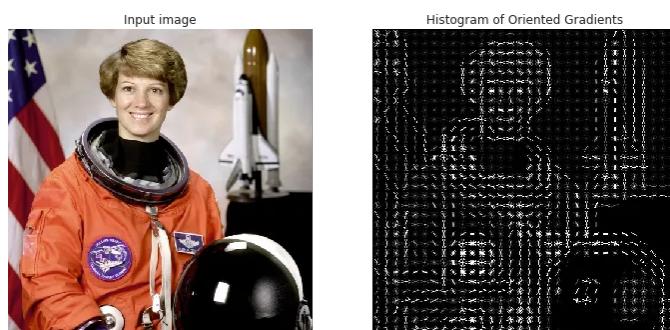


Figura 2.2: Ejemplo de HOG aplicado a una imagen. Extraído de [23].

En el ámbito del procesamiento de imágenes mediante técnicas de deep learning, existen diversas tareas con diferentes niveles de complejidad:

1. **Clasificación de imágenes:** Es la tarea más básica, donde la red neuronal asigna una etiqueta a toda la imagen. Por ejemplo, determinar si una imagen contiene un perro, gato o coche. El modelo genera un vector de probabilidades para cada clase posible.
2. **Clasificación con localización:** Además de clasificar el objeto principal, la red también proporciona un cuadro delimitador (bounding box) que indica dónde se encuentra ese objeto en la imagen. Es útil cuando existe un único objeto de interés.
3. **Detección de objetos:** Extiende la tarea anterior para identificar y localizar múltiples objetos en una imagen. Los algoritmos de detección se dividen principalmente en:
 - *Detectores de dos etapas:* Como R-CNN, Fast R-CNN y Faster R-CNN, primero generan propuestas de regiones que podrían contener objetos, y luego clasifican estas regiones. Son más precisos pero computacionalmente más costosos.
 - *Detectores de una etapa:* Como You Only Look Once (YOLO) y Single Shot MultiBox Detector (SSD) que predicen las cajas delimitadoras y las clases directamente en una sola pasada. Son más rápidos aunque tradicionalmente menos precisos.

Ambos enfoques proporcionan para cada objeto detectado su clasificación y cuadro delimitador.

4. **Segmentación:** Es la tarea más compleja, donde la red no solo identifica y localiza objetos, sino que también asigna una etiqueta a cada píxel de la imagen. Esto permite distinguir entre diferentes objetos y sus contornos, facilitando una comprensión más detallada de la escena.

La Figura 2.3 ilustra estas tareas fundamentales en visión por computador. Para este trabajo, nos centraremos en la tarea de detección de objetos, que es esencial para identificar y clasificar varios objetos en movimiento en un vídeo o imagen.

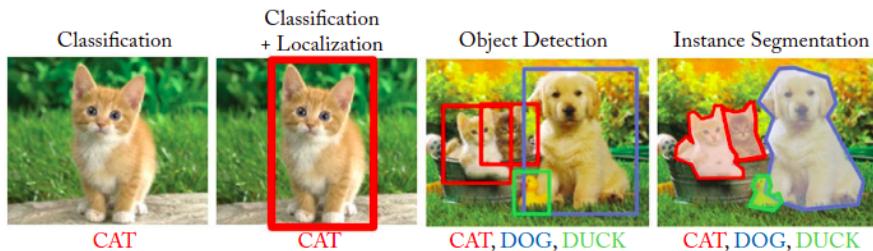


Figura 2.3: Tareas fundamentales en visión por computador. Extraído de [16, fig. 1.1, p. 2].

2.1.3. Arquitectura y funcionamiento de las CNN

Las CNN son un tipo específico de red neuronal profunda. Estas redes están diseñadas para procesar imágenes y extraer características relevantes de manera eficiente, lo que las hace especialmente adecuadas para tareas de visión por computador.

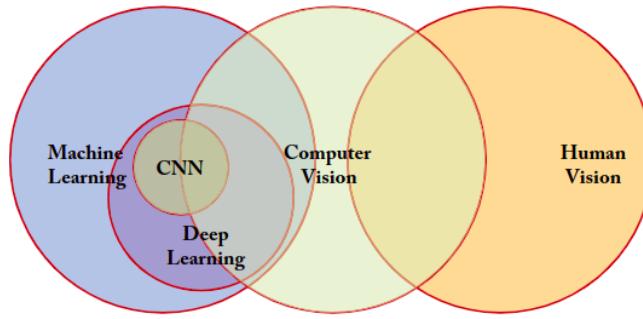


Figura 2.4: Relación entre Machine Learning, Deep Learning, CNN, Computer Vision y Human Vision. Extraído de [16, fig. 1.3, p. 7].

La Figura 2.4 ilustra la relación entre estos conceptos. Las CNN son una subcategoría del Deep Learning, que a su vez es una subcategoría del Machine Learning. Además, las CNN están estrechamente relacionadas con la visión por computador, que busca emular la capacidad de los humanos para interpretar imágenes y videos.

Las CNN se inspiran en la forma en que los humanos percibimos el mundo visual. Al igual que nuestro sistema visual, que procesa la información de manera jerárquica, las CNN utilizan capas convolucionales para extraer características de bajo nivel (como bordes y texturas) y capas más profundas para identificar patrones y objetos más complejos. Esta jerarquía de características permite a las CNN aprender representaciones ricas y abstractas de los datos visuales.

Estas redes se componen de varias capas, cada una de las cuales realiza operaciones específicas en los datos de entrada. Las capas más comunes y técnicas asociadas en una CNN, que se describen a continuación, son las capas convolucionales, las capas de activación, las capas de pooling, la normalización por lotes y las capas completamente conectadas.

Las **capas convolucionales** utilizan la operación de convolución para extraer características. Esta operación es fundamental en las CNN y consiste en aplicar un filtro (o kernel) a una imagen para extraer características locales. El filtro se desliza sobre la imagen, multiplicando sus valores por los valores de la imagen en cada posición y sumando los resultados. Este proceso genera un mapa de activación que resalta las características relevantes de la imagen.

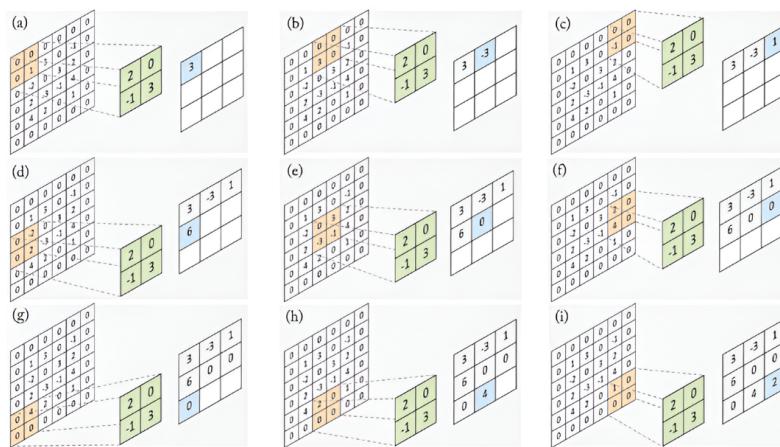


Figura 2.5: Operación de convolución sobre los pixeles de una imagen. Extraído de [16, fig. 2.5, p. 48].

La Figura 2.5 ilustra la operación de una capa de convolución. En este ejemplo, se aplica un filtro de 2×2 (mostrado en verde) a un mapa de características de entrada de 6×6 (incluyendo un relleno de ceros de 1) con un paso (stride) de 2. El filtro se desliza sobre la entrada, y en cada paso, se realiza una multiplicación elemento a elemento entre el filtro y la región correspondiente de la entrada. La suma de estos productos genera un valor en el mapa de características de salida (mostrado en azul).

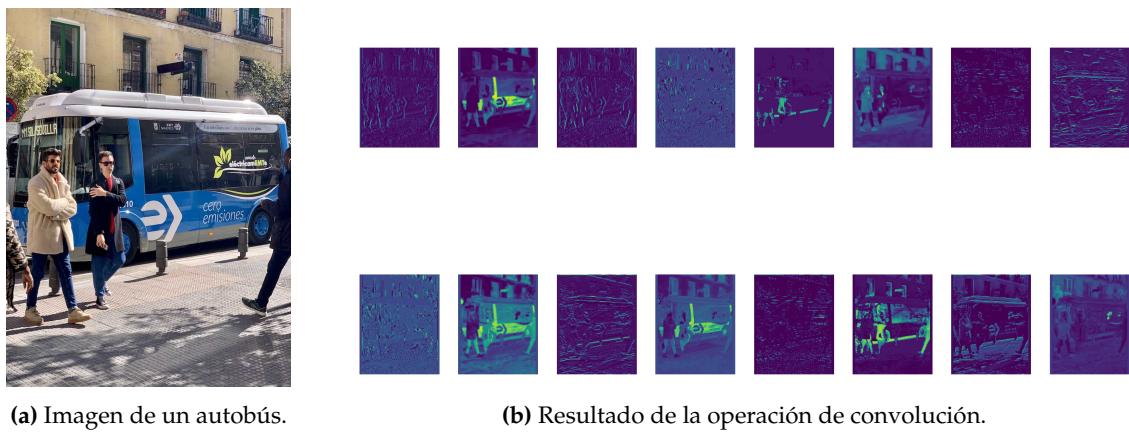


Figura 2.6: Proceso de convolución aplicado a una imagen de un autobús.

La Figura 2.6 ilustra el proceso de la primera convolución del modelo YOLO11n [13]. En la parte izquierda se muestra la imagen original de un autobús, mientras que en la parte derecha se presenta el resultado de aplicar la operación de convolución. En este caso, los 16 filtros de la primera capa convolucional han detectado diferentes características de la imagen, como bordes y texturas. Este proceso se repite en múltiples capas, lo que permite a la red aprender representaciones cada vez más complejas de la imagen.

Las **capas de activación**, especialmente la ReLU (Rectified Linear Unit), son fundamentales para introducir no-linealidad en el modelo. La función ReLU transforma cada valor negativo en cero mientras mantiene los valores positivos sin cambios, lo que ayuda a mitigar el problema del desvanecimiento del gradiente y acelera el proceso de entrenamiento. La función ReLU se define como:

$$f(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases} \quad (2.1)$$

Estas capas se aplican típicamente después de cada capa convolucional y completamente conectada.

Las CNN incorporan **capas de pooling** (o submuestreo). Estas capas desempeñan un papel crucial en la reducción progresiva de la dimensión espacial (ancho y alto) de los mapas de características, lo que conlleva varios beneficios importantes: disminuyen la cantidad de parámetros y la carga computacional, ayudan a controlar el sobreajuste (*overfitting*) al reducir la complejidad del modelo, y proporcionan un cierto grado de invarianza a pequeñas traslaciones y distorsiones. El funcionamiento del pooling implica deslizar una ventana sobre el mapa de características de entrada y aplicar una operación de agregación. Las operaciones más comunes son Max-Pooling, que selecciona el valor máximo dentro de la ventana (eficaz para capturar las características más prominentes, como se ilustra en la Figura 2.7), y Average-Pooling, que calcula el valor promedio (tiende a suavizar las características). El resultado es un mapa de características de menor tamaño pero que conserva la información esencial.



Figura 2.7: Ejemplo de operación de max-pooling con una ventana de 2×2 y un *stride* de 2. Se selecciona el valor máximo de cada región de color. Extraído de [3].

La **Normalización por Lotes** (*Batch Normalization*, BN) es una técnica fundamental para mitigar el problema del *cambio interno de covariables* (*internal covariate shift*), que es la alteración de la distribución de las activaciones intermedias durante el entrenamiento. BN opera a nivel de mini-lotes $\mathcal{B} = \{x_1, \dots, x_m\}$, calculando la media $\mu_{\mathcal{B}}$ y la varianza $\sigma_{\mathcal{B}}^2$:

$$\mu_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^m x_i \quad (2.2)$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad (2.3)$$

Luego, normaliza cada activación x_i :

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad (2.4)$$

donde ϵ es una constante pequeña para estabilidad numérica. Para preservar la capacidad expresiva, BN introduce parámetros aprendibles γ (escala) y β (desplazamiento) para realizar una transformación afín:

$$y_i = \gamma \hat{x}_i + \beta \quad (2.5)$$

La salida y_i se propaga a la siguiente operación. BN estabiliza y acelera el entrenamiento, permite tasas de aprendizaje más altas y tiene un efecto regularizador que ayuda a prevenir el sobreajuste. Se inserta típicamente después de capas convolucionales o totalmente conectadas, antes de la activación.

En el final de la red, se utilizan **capas completamente conectadas** (*fully connected*). Estas capas toman las características de alto nivel extraídas por las capas anteriores y las combinan para realizar la tarea final, como la clasificación de objetos. Cada neurona en una capa completamente conectada está conectada a todas las neuronas de la capa anterior. En la salida final, se utiliza una función de activación como Softmax para convertir las salidas en probabilidades de clase. La función Softmax se define como:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad (2.6)$$

donde $z = (z_1, \dots, z_K)$ es el vector de salidas de la capa anterior para K clases. La función Softmax transforma este vector en un vector de probabilidades $\sigma(z) = (\sigma(z_1), \dots, \sigma(z_K))$, donde cada componente $\sigma(z_i)$ se calcula como se muestra. El resultado es un vector de tamaño K donde cada elemento está en el rango $(0, 1)$ y la suma de todos los elementos

es igual a 1 ($\sum_{j=1}^K \sigma(z_j) = 1$). Esto permite interpretar la salida como una distribución de probabilidad sobre las K posibles clases.

En la Figura 2.8 se muestra un ejemplo de una CNN simple. Esta red incluye capas convolucionales para la extracción de características, capas de activación para introducir no linealidad, capas de pooling para reducir la dimensionalidad y capas completamente conectadas para realizar la clasificación final. La combinación de estas capas permite a las CNN aprender representaciones jerárquicas y complejas de los datos visuales, lo que las convierte en una herramienta poderosa para tareas de visión por computador.



Figura 2.8: Ejemplo de una CNN simple. Extraído de [15].

Tras comprender la arquitectura interna de las CNN, el siguiente paso es el entrenamiento. Este proceso se basa fundamentalmente en el uso de un conjunto de datos etiquetado, que actúa como la verdad fundamental (*ground truth*). Este conjunto contiene una colección de imágenes junto con sus correspondientes etiquetas o anotaciones, que la red utilizará para aprender.

El objetivo principal del entrenamiento es ajustar los parámetros internos de la CNN (pesos y sesgos) para minimizar una función de pérdida predefinida. Esta función mide la discrepancia o el error entre las predicciones generadas por la red y las etiquetas reales proporcionadas en los datos de entrenamiento.

El proceso de ajuste se realiza de forma iterativa mediante un algoritmo de optimización. El Descenso de Gradiente Estocástico (SGD) y sus variantes, como Adam o RMSprop, son opciones comunes. Estos algoritmos utilizan el cálculo de gradientes para determinar cómo modificar los pesos y sesgos para reducir el error.

El mecanismo central de este ajuste implica dos fases: la propagación hacia adelante (*forward propagation*) y la retropropagación (*backpropagation*). Durante la propagación hacia adelante, los datos de entrada atraviesan las capas de la red para generar una salida. Esta salida se compara con la etiqueta real mediante la función de pérdida. En la retropropagación, el algoritmo calcula el gradiente del error con respecto a los pesos y sesgos, utilizando la regla de la cadena. Estos gradientes guían la actualización de los parámetros.

Típicamente, todo el conjunto de datos de entrenamiento se procesa varias veces en ciclos conocidos como épocas. Este refinamiento iterativo continúa hasta que el rendimiento de la red, evaluado en un conjunto de datos de validación separado (que no se utiliza para el entrenamiento directo), alcanza un nivel satisfactorio. La validación es crucial para asegurar que el modelo generaliza bien a datos no vistos previamente.

Además, para prevenir el sobreajuste (*overfitting*) —donde el modelo aprende demasiado bien los datos de entrenamiento pero falla en generalizar— se emplean técnicas de regularización. El *dropout* es una técnica común que desactiva aleatoriamente un porcentaje de neuronas durante cada iteración de entrenamiento, forzando a la red a aprender representaciones más robustas. La normalización por lotes (*batch normalization*), explicada previamente, también actúa como regularizador y ayuda a estabilizar el entrenamiento.

En escenarios prácticos, desarrollar y entrenar una CNN desde cero puede ser computacionalmente costoso y requerir grandes cantidades de datos etiquetados.

Por lo tanto, una estrategia común y muy eficaz consiste en aprovechar modelos preentrenados. Arquitecturas como VGG16, ResNet50 y MobileNetV2, que han sido entrenadas previamente en conjuntos de datos de referencia masivos como ImageNet[6] (que contiene millones de imágenes en miles de categorías) o COCO[20], sirven como puntos de partida potentes. Estos modelos ya han aprendido características jerárquicas ricas a partir de datos visuales diversos.

Mediante una técnica conocida como *transfer learning* o aprendizaje por transferencia, estos modelos preentrenados pueden adaptarse eficientemente a tareas nuevas y específicas, incluso con conjuntos de datos personalizados más pequeños. El aprendizaje por transferencia generalmente implica tomar las capas de extracción de características del modelo preentrenado (la base convolucional) y ajustarlas (*fine-tuning*) o añadir nuevas capas de clasificación adaptadas a la tarea objetivo.

Este enfoque acelera significativamente el proceso de entrenamiento para la nueva tarea y, a menudo, conduce a un mejor rendimiento en comparación con el entrenamiento desde cero, ya que transfiere eficazmente el conocimiento visual general adquirido durante el entrenamiento inicial a gran escala.

2.1.4. Detectores de dos etapas

Los detectores de dos etapas, funcionan mediante un proceso secuencial: primero generan propuestas de regiones (region proposals) que podrían contener objetos y posteriormente clasifican estas regiones. Este enfoque favorece la precisión, aunque generalmente a costa de un mayor tiempo de procesamiento.

La primera arquitectura exitosa de detección de objetos basada en deep learning fue R-CNN (Regions with CNN features) [9]. Este modelo introdujo un enfoque de dos etapas que revolucionó el campo. En su primera fase, R-CNN utiliza un algoritmo de búsqueda selectiva (Selective Search) para generar aproximadamente 2,000 propuestas de regiones que podrían contener objetos. Este algoritmo de búsqueda selectiva divide la imagen en nodos y aristas, e iterativamente agrupa estas regiones en función del color, textura, tamaño y forma hasta que se obtienen las propuestas finales. En la figura 2.9 se muestra un ejemplo del resultado del algoritmo de búsqueda selectiva.



(a) Imagen original.



(b) Resultado de búsqueda selectiva.

Figura 2.9: Proceso de búsqueda selectiva aplicado a una imagen. Extraído de [19].

En la segunda fase, cada región propuesta es redimensionada y procesada individualmente por una CNN para extraer características de alto nivel. Estas características alimentan posteriormente a un clasificador SVM (Support Vector Machine) para determinar la categoría del objeto y a un regresor lineal para mejorar la localización del cuadro delimitador. Como se ilustra en la Figura 2.10, este enfoque fue innovador pero computacionalmente costoso, ya que requiere procesar cada propuesta de región de manera independiente.

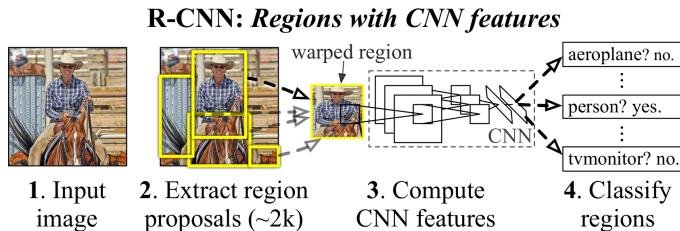


Figura 2.10: Arquitectura de R-CNN. Extraído de [9, fig. 1, p. 1].

2.1.5. Detectores de una etapa

En contraste con los detectores de dos etapas, los detectores de una etapa (one-stage detectors) adoptan un enfoque más directo y eficiente. Estos detectores realizan la localización y clasificación de objetos simultáneamente en una sola pasada a través de la red, sin necesidad de un paso intermedio de generación de propuestas.

La arquitectura de los detectores de una etapa procesa la imagen completa una única vez, típicamente mediante una red troncal o *backbone* (generalmente una CNN) para la extracción de características. Estas características son posteriormente procesadas por componentes intermedios (*neck*) y alimentadas a una cabeza de detección (*detection head*) que predice simultáneamente las coordenadas de los cuadros delimitadores (*bounding boxes*) y las probabilidades de clase. Esta arquitectura de una etapa prioriza la velocidad de inferencia, resultando idónea para aplicaciones en tiempo real donde la latencia es un factor crítico, aunque pueda suponer una ligera concesión en la precisión máxima. Modelos representativos de este enfoque incluyen SSD (Single Shot MultiBox Detector)[21] y YOLO (You Only Look Once)[27]. Estos han demostrado un equilibrio eficaz entre rapidez y exactitud, permitiendo la detección en tiempo real incluso en dispositivos con recursos computacionales limitados.

YOLO se destaca como una de las arquitecturas de detección de objetos más populares y efectivas. Concebida específicamente para la detección en tiempo real, YOLO introdujo un enfoque unificado que procesa la imagen completa en una sola pasada, realizando la localización y clasificación de forma simultánea. Esta metodología ha sido fundamental para su adopción en aplicaciones que requieren alta velocidad de procesamiento.

YOLO en su primera versión procesa la imagen completa de una vez. Divide la imagen en una cuadrícula de $S \times S$ celdas. Cada celda es responsable de detectar los objetos cuyo centro se encuentre dentro de ella. Para cada celda, YOLO predice B cuadros delimitadores (*bounding boxes*) y puntuaciones de confianza para esos cuadros. La puntuación de confianza indica la probabilidad de que haya un objeto en el cuadro y la precisión de la predicción del cuadro. Al mismo tiempo, predice las probabilidades de clase para cada objeto detectado en la celda.

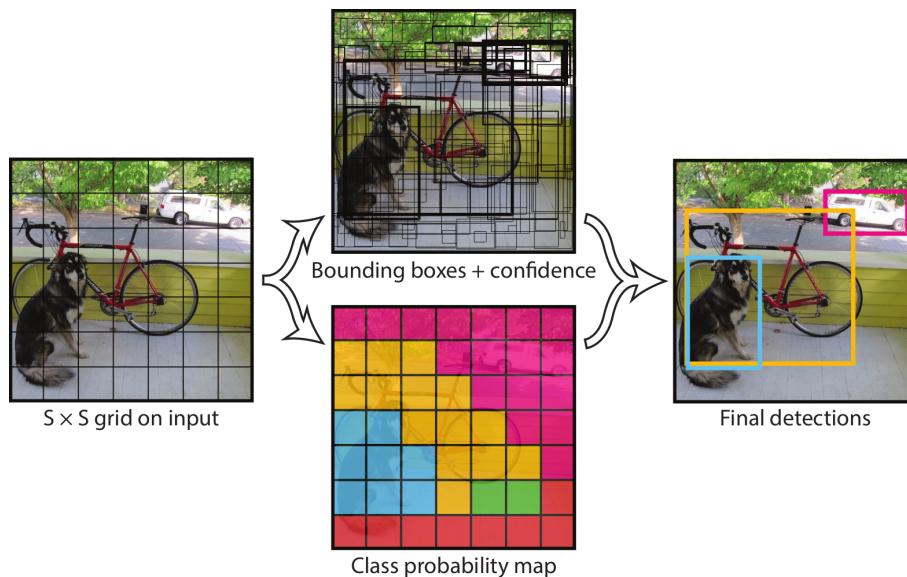


Figura 2.11: Ejemplo de detección de objetos utilizando YOLO. Extraído de [27, fig. 2, p. 2].

Como se ilustra en la Figura 2.11, YOLO modela la detección como un problema de regresión. Las predicciones del modelo se codifican como una matriz de 3 dimensiones de $S \times S \times (B * 5 + C)$, donde el factor 5 corresponde a las coordenadas x, y , ancho, alto y la puntuación de confianza para cada cuadro delimitador, mientras que C representa el número de clases posibles. Por ejemplo, si se utilizan $B = 2$ cuadros delimitadores (por cada celda, es decir, cada celda realiza 2 predicciones) y $C = 20$ clases, el tensor de salida tendrá dimensiones $S \times S \times (2 * 5 + 20)$.

Una vez generadas las predicciones, YOLO implementa un post-procesamiento mediante Non-Maximum Suppression (NMS) para eliminar detecciones redundantes y conservar únicamente las más precisas. El algoritmo NMS funciona de la siguiente manera:

1. **Ordenamiento por Confianza:** Las cajas delimitadoras detectadas se ordenan por su puntuación de confianza, de mayor a menor.
2. **Selección de la Detección Más Confiable:** Se selecciona la caja delimitadora con la puntuación de confianza más alta y se añade a la lista de detecciones finales.
3. **Cálculo de la Intersección sobre Unión (IoU):** Se calcula la Intersection over Union (IoU) entre la caja delimitadora seleccionada y todas las demás cajas delimitadoras restantes.
4. **Eliminación de Detecciones Redundantes:** Se eliminan todas las cajas delimitadoras con una IoU superior a un umbral predefinido con la caja delimitadora seleccionada.
5. **Iteración:** Se repiten los pasos 2-4 hasta que no queden más cajas delimitadoras por procesar.

En resumen, NMS evalúa la superposición entre las cajas y elimina aquellas que exceden un cierto umbral con la caja de mayor confianza, asegurando que solo las detecciones más precisas y no redundantes se conserven, mejorando la calidad general de las detecciones.

Este enfoque unificado permite que YOLO procese imágenes a velocidades significativamente mayores que los detectores de dos etapas, mientras mantiene una precisión

competitiva, lo que lo hace ideal para aplicaciones en tiempo real como las que se abordan en este trabajo.

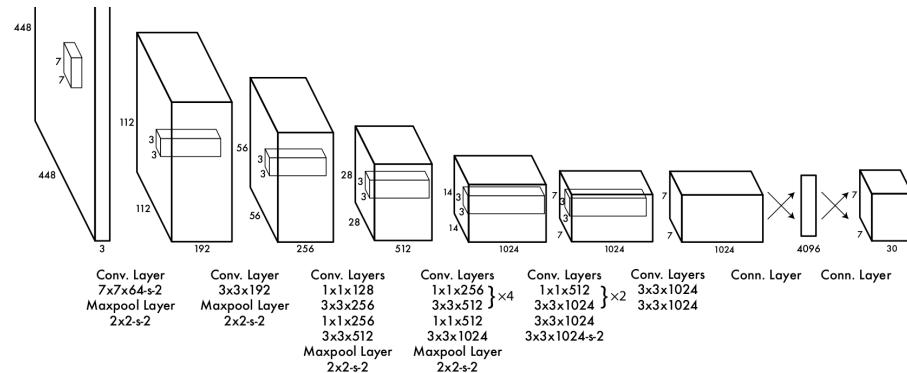


Figura 2.12: Arquitectura de YOLO. Extraído de [27, fig. 3, p. 3].

En la Figura 2.12 se presenta la arquitectura del modelo primigenio de YOLO [27]. Esta arquitectura se basa en una red neuronal convolucional que extrae características de la imagen de entrada y las procesa a través de varias capas para generar las predicciones finales. A lo largo de los años, se han desarrollado múltiples versiones y mejoras de YOLO, cada una optimizando aspectos como la precisión, la velocidad y la capacidad de detección de objetos pequeños o densamente agrupados.

En este proyecto, se han seleccionado diversas versiones de YOLOV5 [11], YOLOV8 [12] y YOLO11 [13]. Estas representan mejoras y optimizaciones sobre arquitecturas YOLO anteriores. Los modelos mencionados forman parte de la librería Ultralytics [14], que proporciona una implementación eficaz y sencilla de varias variantes de YOLO. Las aportaciones de Ultralytics a la comunidad de visión artificial son notables, ya que sus herramientas y recursos facilitan el ciclo de vida completo (entrenamiento, evaluación, implementación) de los modelos YOLO.

2.1.6. Métricas de evaluación de modelos de detección de objetos

Para evaluar el rendimiento de los modelos de detección de objetos, se utilizan métricas específicas que permiten cuantificar la precisión y efectividad del sistema. Entre las métricas más comunes se encuentran:

- **Precisión:** Mide la proporción de verdaderos positivos (TP) respecto al total de predicciones positivas realizadas por el modelo. Se calcula como:

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

donde FP representa los falsos positivos. Una alta precisión indica que el modelo realiza pocas predicciones incorrectas.

- **Recall:** Mide la proporción de verdaderos positivos respecto al total de objetos relevantes en la imagen. Se calcula como:

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

donde FN representa los falsos negativos. Un alto recall indica que el modelo es capaz de detectar la mayoría de los objetos relevantes, aunque pueda incluir algunas predicciones incorrectas.

- **IoU (Intersection over Union):** Mide la superposición entre el cuadro delimitador predicho y el cuadro delimitador real. Se calcula como:

$$\text{IoU} = \frac{\text{Área de intersección}}{\text{Área de unión}}$$

Una IoU alta indica que el modelo ha localizado correctamente el objeto. Generalmente, se considera que una IoU superior a 0.5 indica una detección correcta.

El cálculo del mAP se realiza en varios pasos: primero, para cada clase se calcula la curva PR (Precision-Recall) y se obtiene el Average Precision (AP), que representa el área bajo esta curva. Luego, el mAP se obtiene como la media de todos los AP de las diferentes clases. En detección de objetos, el mAP suele incorporar diferentes umbrales de IoU (Intersection over Union), expresándose como mAP@IoU=0.5 o mAP@IoU=0.5:0.95. Esta métrica es especialmente útil cuando las clases están desequilibradas, ya que da igual importancia a clases minoritarias y mayoritarias. Un valor de mAP cercano a 1 indica un modelo con alta precisión y recall en todas las clases.

- **Latencia:** Mide el tiempo que tarda el modelo en procesar una imagen y generar una predicción se mide en milisegundos (ms) y un valor bajo indica que el modelo es capaz de realizar inferencias rápidamente.
- **FPS (Frames Per Second):** Mide la velocidad de procesamiento del modelo, indicando cuántas imágenes puede procesar por segundo; un alto valor de FPS indica que el modelo es capaz de realizar inferencias rápidamente.
- **F1 Score:** Es la media armónica entre precisión y recall, proporcionando un balance entre ambos. Se utiliza para evaluar el rendimiento del modelo en situaciones donde hay un desbalance entre clases. Se calcula como:

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Un F1 Score alto indica que el modelo tiene un buen equilibrio entre precisión y recall, lo que es especialmente importante en aplicaciones donde tanto la detección correcta como la minimización de falsos positivos son críticas.

- **Loss (Pérdida):** Es una medida de la discrepancia entre las predicciones del modelo y las etiquetas reales. Durante el entrenamiento, el objetivo es minimizar esta pérdida. Existen diferentes tipos de funciones de pérdida, como la pérdida de clasificación (Cross-Entropy Loss) y la pérdida de localización (Smooth L1 Loss), que se combinan para evaluar el rendimiento del modelo.

2.2 Aceleradores de procesamiento gráfico

Un aspecto fundamental para el despliegue eficiente de modelos de IA es el hardware utilizado para su ejecución. A continuación, se analizan los principales aspectos de los aceleradores de procesamiento gráfico y su importancia en aplicaciones de visión artificial.

2.2.1. Limitaciones del hardware tradicional

La Dennard Scaling[7], formulada por Robert Dennard en 1974, establecía que a medida que los transistores se reducían de tamaño, su consumo de energía por unidad de área se mantenía constante. Esto significaba que, al reducir el tamaño de los transistores a la mitad, su área se reducía a un cuarto, pero su consumo de energía por unidad de área permanecía igual. Como resultado, el consumo total de energía se reducía a la mitad, permitiendo aumentar la frecuencia de reloj y el número de transistores sin incrementar significativamente el consumo total de energía. Este principio fue fundamental para el avance exponencial en el rendimiento de los procesadores durante décadas. Sin embargo, a partir de 2005, la Dennard Scaling dejó de cumplirse debido a varios factores físicos fundamentales. A escalas nanométricas, los efectos cuánticos y las fugas de corriente se volvieron significativos, impidiendo que el consumo de energía por unidad de área se mantuviera constante.

Por otro lado, la ley de Moore[24], formulada por Gordon Moore en 1965, predecía que el número de transistores en un chip se duplicaría aproximadamente cada año, predicción que posteriormente se ajustó a cada dos años. Durante décadas, esta ley se cumplió con notable precisión, permitiendo un crecimiento exponencial en la capacidad de procesamiento. Sin embargo, a medida que los transistores se acercan a escalas atómicas (actualmente en torno a los 3-5 nanómetros), los límites físicos y los desafíos de fabricación han ralentizado significativamente este ritmo de avance. Los costes de investigación y desarrollo para mantener esta tendencia se han disparado, y los beneficios en términos de rendimiento por transistor se han reducido.

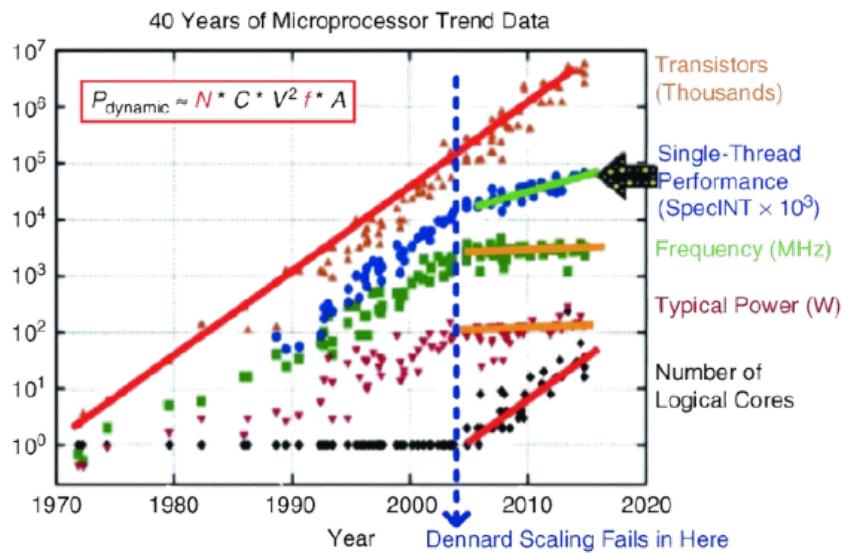


Figura 2.13: Evolución histórica de las características de los microprocesadores (1970–2020). Extraído de [4, p. 8].

En la Figura 2.13 se ilustra claramente el impacto combinado del fin de la Dennard Scaling y el estancamiento de la ley de Moore. La gráfica muestra cinco métricas fundamentales en escala logarítmica: el número de transistores (triángulos naranja) que sigue la ley de Moore, el rendimiento de un solo hilo (círculos azules), la frecuencia de reloj (cuadrados verdes), el consumo de potencia (triángulos invertidos morados) y el número de núcleos lógicos (rombos negros). La ecuación de potencia dinámica ($P_{dynamic} \approx N * C * V^2 f * A$)

$N * C * V^2 * f * A$) explica la relación entre el número de transistores (N), la capacitancia (C), el voltaje (V), la frecuencia (f) y el factor de actividad (A).

El punto de inflexión en 2005, marcado como *Dennard Scaling Fails in Here*, marca el momento en que la industria tuvo que cambiar radicalmente su estrategia. La frecuencia de reloj se estancó en torno a los 3-4 GHz, el rendimiento por núcleo comenzó a crecer más lentamente, y como respuesta, se adoptaron dos estrategias principales: el aumento del número de núcleos y la estabilización del consumo de potencia alrededor de los 100W. Este fenómeno ha llevado a la industria a buscar alternativas como las arquitecturas de dominio específico para continuar mejorando el rendimiento de los sistemas computacionales.

2.2.2. Arquitectura y funcionamiento de las GPUs

Para superar las limitaciones de las CPUs en tareas específicas, se emplean arquitecturas hardware especializadas. Entre ellas, las Field Programmable Gate Arrays (FPGAs) ofrecen flexibilidad al ser reconfigurables post-fabricación, aunque su programación es compleja. En el extremo de la especialización se encuentran los Application-Specific Integrated Circuits (ASICs), diseñados a medida para una única función, logrando máxima eficiencia a costa de un alto coste de diseño y nula reprogramabilidad; un ejemplo son las TPUs de Google, optimizadas para ML.

Las GPUs, por su parte, son arquitecturas orientadas al paralelismo masivo. Su diseño *many-core*, con miles de núcleos de procesamiento más simples, las hace idóneas para cargas de trabajo intensivas y paralelizables, como las operaciones matriciales del aprendizaje profundo. Dada esta capacidad, las GPUs son fundamentales en este proyecto.

El modelo de programación de las GPUs está basado en la ejecución masiva de hilos, organizados en bloques y rejillas (*blocks* y *grids*), según la terminología de CUDA, el modelo de programación desarrollado por NVIDIA. Cada hilo ejecuta la misma función, conocida como *kernel*, pero opera sobre diferentes fragmentos de datos. Este enfoque, conocido como SIMT (Single Instruction, Multiple Threads), permite aprovechar al máximo el paralelismo inherente a muchas aplicaciones de IA, como el entrenamiento y la inferencia de redes neuronales.

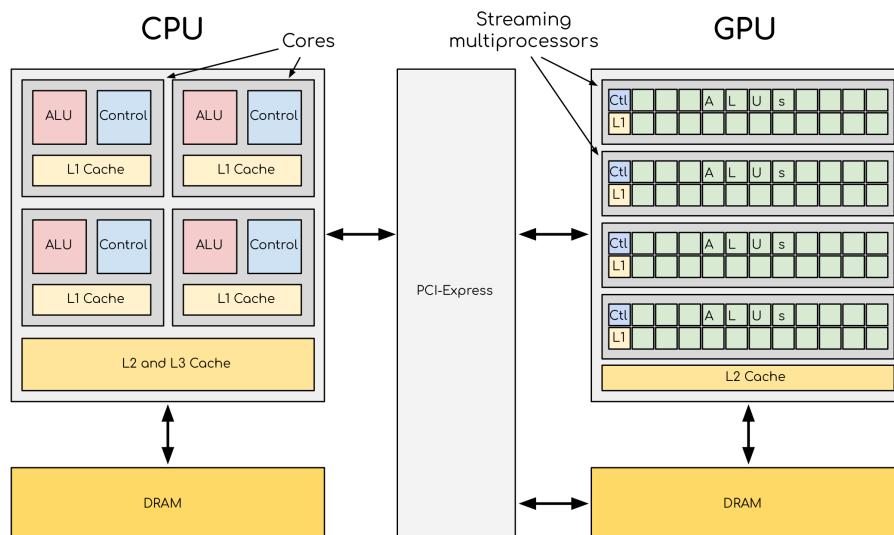


Figura 2.14: Comparativa de arquitecturas CPU y GPU. Extraído de [8].

La figura 2.14 ilustra las diferencias arquitectónicas entre CPU y GPU. Las CPUs constan de un número reducido de núcleos (generalmente entre 4 y 64) optimizados para un alto rendimiento secuencial, donde cada núcleo dispone típicamente de caché y unidad de control privadas. En cambio, las GPUs se basan en una arquitectura *many-core*, integrando miles de núcleos más simples diseñados para el paralelismo masivo, aunque con menor rendimiento individual por núcleo. Estos núcleos de GPUs suelen compartir recursos como la caché y las unidades de control, lo que posibilita empaquetar una mayor cantidad de unidades de procesamiento en el mismo chip y, consecuentemente, alcanzar una mayor densidad de cómputo.

NVIDIA ha sido una figura central en la evolución de la aceleración de IA, realizando contribuciones clave que han modelado el campo. Fue pionera en la computación de propósito general en GPU con la introducción de CUDA en 2006. Esta plataforma permitió utilizar la masiva capacidad de procesamiento paralelo de las GPU para tareas computacionales generales, extendiendo su uso más allá de los gráficos tradicionales. La programación de GPU se realiza utilizando lenguajes y APIs especializadas como CUDA y OpenCL, que otorgan al desarrollador un control explícito sobre la distribución de datos y tareas entre los miles de núcleos disponibles.

El desarrollo eficaz en este paradigma requiere identificar y explotar el paralelismo inherente a los algoritmos, así como gestionar eficientemente la compleja jerarquía de memoria de la GPU. Es crucial considerar la latencia significativamente mayor del acceso a la memoria global en comparación con memorias más rápidas pero de menor capacidad, como la memoria compartida local a un bloque de hilos.

Dada la complejidad de esta programación a bajo nivel, NVIDIA ha desarrollado un ecosistema de software robusto y optimizado para facilitar el desarrollo en IA. Este incluye bibliotecas fundamentales como cuDNN, específica para acelerar primitivas de redes neuronales profundas, y cuBLAS, para operaciones de álgebra lineal básica, ambas esenciales para el rendimiento. Además, es común recurrir a frameworks de alto nivel como PyTorch y TensorFlow. Estas librerías abstraen muchos detalles de la implementación en GPU, utilizando las bibliotecas de NVIDIA subyacentes y simplificando enormemente el desarrollo de aplicaciones de IA.

En el frente del hardware, la compañía ha impulsado continuamente la innovación con el desarrollo de componentes especializados como los Tensor Cores, introducidos en la arquitectura Volta. Estos núcleos están diseñados específicamente para acelerar las operaciones matriciales intensivas (como multiplicaciones de matrices mixtas) que son omnipresentes en el entrenamiento e inferencia de modelos de IA. Como resultado de estas continuas innovaciones en hardware y la creación de un ecosistema de software integral, NVIDIA ha logrado establecer estándares de facto para la aceleración de IA, consolidándose como la plataforma preferida en una amplia gama de entornos, desde grandes centros de datos hasta sistemas embebidos con recursos limitados.

2.2.3. Serie Jetson: dispositivos para IA de bajo consumo

La serie Jetson de NVIDIA constituye una familia de módulos computacionales diseñados específicamente para habilitar la IA y el aprendizaje profundo en dispositivos de borde (edge devices). Estos sistemas compactos y de bajo consumo son fundamentales para aplicaciones que requieren procesamiento local de datos con alta capacidad de cómputo.

El enfoque principal de la serie Jetson es la computación en el borde (edge computing), un paradigma que acerca el procesamiento de datos y la IA a la fuente donde se generan. Esto resulta crucial en aplicaciones donde la latencia, el ancho de banda limitado

o la privacidad son factores críticos, ya que evita la necesidad de enviar grandes volúmenes de datos a la nube para su análisis. Los dispositivos Jetson están optimizados para operar bajo restricciones significativas de energía, tamaño y coste, características típicas de los entornos embebidos y de borde.

Cada módulo Jetson se basa en una arquitectura System-on-Chip (SoC), que integra múltiples componentes de procesamiento en un único circuito integrado. Esto incluye núcleos de CPU basados en la arquitectura ARM y potentes núcleos de GPU NVIDIA con arquitecturas modernas. Además, algunos modelos de gama alta, como los Jetson AGX Orin y AGX Xavier utilizados en este trabajo, incorporan aceleradores de hardware dedicados conocidos como DLAs. Estas unidades especializadas están diseñadas para ejecutar operaciones de inferencia de redes neuronales de manera altamente eficiente en términos de rendimiento y consumo energético, liberando así la GPU y la CPU para otras tareas. Esta integración heterogénea permite una alta eficiencia computacional y energética, reduce la huella física del sistema y simplifica el diseño de la placa portadora, resultando en soluciones más compactas y eficientes.

Un pilar fundamental del diseño de la serie Jetson es la optimización de la eficiencia energética. Estos dispositivos están diseñados para ofrecer un alto rendimiento computacional por cada vatio de energía consumido (TOPS/W), esencial para aplicaciones embebidas con fuentes de alimentación limitadas o restricciones térmicas. La capacidad de configurar diferentes perfiles de energía permite ajustar dinámicamente el equilibrio entre rendimiento y consumo.

Modelo	AI Performance	GPU	GPU Max Freq.	CPU	Memoria	DLAs	Precio (€)
Jetson AGX Orin	275 TOPS	2048-core Ampere, 64 Tensor Cores	1.3 GHz	12-core Cortex-A78AE, 3MB L2 + 6MB L3, 2.2 GHz	64GB LPDDR5, 204.8GB/s	2x NVDLA v2	2400
Jetson Orin Nano	67 TOPS	1024-core Ampere, 32 Tensor Cores	1020 MHz	6-core Cortex-A78AE, 1.5MB L2 + 4MB L3, 1.7 GHz	8GB LPDDR5, 102 GB/s	-	300
Jetson AGX Xavier	32 TOPS	512-core Volta, 64 Tensor Cores	1377 MHz	8-core Carmel v8.2, 8MB L2 + 4MB L3, 2.2 GHz	32GB LPDDR4x, 136.5GB/s	2x NVDLA v1	1000

Tabla 2.1: Comparativa técnica entre diferentes modelos NVIDIA Jetson.

La tabla 2.1[26] presenta una comparativa técnica entre diferentes modelos de la serie Jetson disponibles para la realización de este trabajo, incluyendo la presencia y tipo de DLAs. Cada modelo está diseñado para satisfacer diferentes necesidades y requisitos de rendimiento, lo que permite a los desarrolladores seleccionar el módulo más adecuado para su aplicación específica. Para este trabajo, se han utilizado los tres modelos de la tabla y se han comparado sus resultados.

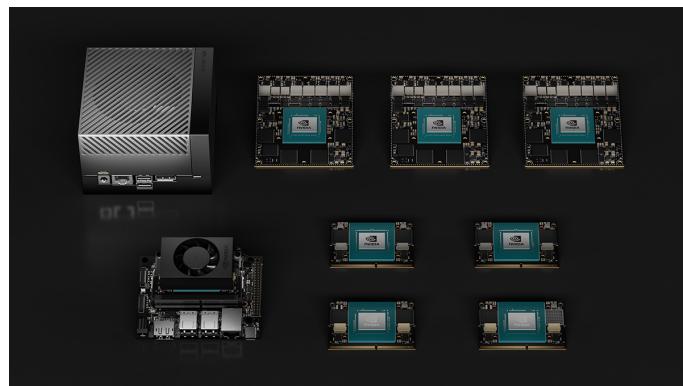


Figura 2.15: Módulos Jetson de NVIDIA

Más allá del hardware, la fortaleza de la plataforma Jetson reside en su completo ecosistema de software, proporcionado a través del SDK NVIDIA JetPack. Esto incluye un sistema operativo Linux optimizado (L4T), controladores, CUDA, cuDNN, TensorRT, herramientas de desarrollo y documentación. Esta plataforma unificada agiliza el desarrollo, desde la creación del modelo hasta la implementación optimizada.

Para este trabajo, todos los programas se ejecutaron utilizando imágenes de contenedores Docker proporcionadas por Ultralytics [14]. Construidas sobre las imágenes base de NVIDIA y optimizadas para la serie Jetson, estas imágenes preconfiguradas simplifican la implementación al incluir todas las dependencias necesarias como CUDA, cuDNN, TensorRT y las bibliotecas de Python requeridas, asegurando la reproducibilidad del entorno.

2.2.4. TensorRT

NVIDIA TensorRT es un kit de desarrollo de software (SDK) integral diseñado específicamente para la optimización de modelos de aprendizaje profundo y la consecución de una inferencia de muy alto rendimiento en la amplia gama de hardware de NVIDIA, desde centros de datos hasta sistemas embebidos como la serie Jetson. Actúa como un potente compilador y motor de ejecución en tiempo real que transforma modelos previamente entrenados en versiones altamente eficientes, optimizadas para el despliegue en producción.

El objetivo principal de TensorRT es cerrar la brecha entre los frameworks de entrenamiento (como TensorFlow o PyTorch), que priorizan la flexibilidad y la facilidad de desarrollo, y los requisitos estrictos de las aplicaciones de inferencia en el mundo real, que demandan baja latencia, alto rendimiento (throughput) y eficiencia energética. Para lograr esto, TensorRT aplica una serie de optimizaciones sofisticadas durante una fase de compilación offline:

- **Optimización del Grafo Computacional:** TensorRT analiza la estructura del modelo y realiza transformaciones significativas para mejorar la eficiencia. Esto incluye:
 - *Fusión de Capas (Layer Fusion):* Combina múltiples capas secuenciales (fusión vertical) o paralelas (fusión horizontal) en un único kernel optimizado. Por ejemplo, una secuencia de convolución, sesgo (bias) y activación (ReLU) puede fusionarse en una sola operación, reduciendo la sobrecarga de lanzamiento de kernels y el movimiento de datos en memoria.
 - *Eliminación de Capas:* Identifica y elimina capas que no son necesarias para la inferencia, como las capas de dropout.
 - *Fusión de Tensores (Tensor Fusion):* Combina operaciones que acceden a los mismos tensores para mejorar la localidad de los datos.
- **Calibración y Cuantización de Precisión:** TensorRT ofrece un soporte robusto para reducir la precisión numérica de los pesos y activaciones del modelo. Puede convertir modelos de precisión completa (FP32) a precisiones más bajas como FP16 (media precisión), INT8 (enteros de 8 bits), o incluso formatos más recientes como FP8 o FP4 en hardware compatible. Esta reducción disminuye drásticamente el tamaño del modelo, el ancho de banda de memoria requerido y acelera el cómputo (especialmente en hardware con soporte nativo como los Tensor Cores), a menudo con una pérdida mínima o nula de precisión.
- **Selección Automática de Kernels (Kernel Auto-Tuning):** Durante la fase de construcción, TensorRT evalúa múltiples implementaciones (kernels) para cada opera-

ción en el hardware de destino específico y selecciona la más rápida disponible, considerando factores como el tamaño de los tensores y la precisión utilizada.

- **Gestión Dinámica de Memoria (Dynamic Tensor Memory):** Optimiza la asignación de memoria para los tensores intermedios, reutilizando la memoria siempre que sea posible para minimizar la huella de memoria global.
- **Ejecución Multi-Stream:** Facilita la ejecución concurrente de múltiples flujos de inferencia en la misma GPU, mejorando el rendimiento general del sistema.

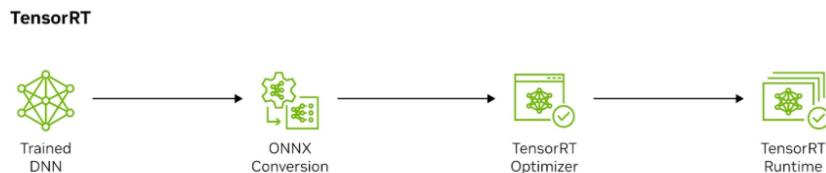


Figura 2.16: Ejemplo de flujo de trabajo de optimización con TensorRT

El proceso típico de uso de TensorRT, como se ilustra esquemáticamente en la Figura 2.16, implica tomar un modelo entrenado (generalmente exportado a un formato intermedio como Open Neural Network Exchange (ONNX)), utilizar el “constructor” (builder) de TensorRT para aplicar las optimizaciones y generar un “motor” (engine) de inferencia serializado y optimizado. Este motor es específico para el modelo, la precisión deseada y la plataforma hardware de destino (p. ej., un Jetson Orin Nano específico). Finalmente, el motor se carga en el “tiempo de ejecución” (runtime) de TensorRT para realizar inferencias rápidas y eficientes.

TensorRT se integra de forma nativa con los principales frameworks de aprendizaje profundo (TensorFlow, PyTorch) y soporta el formato de intercambio ONNX, lo que facilita la importación de modelos desde prácticamente cualquier framework de entrenamiento. Su capacidad para reducir significativamente la latencia y aumentar el rendimiento lo convierte en una herramienta esencial para desplegar modelos de IA en aplicaciones sensibles al tiempo real y en dispositivos con recursos limitados como los de la serie Jetson.

2.3 Seguimiento de objetos en tiempo real

En esta sección se presentarán los conceptos básicos del seguimiento de objetos en tiempo real, se explicará el problema del Multiple Object Tracking (MOT) y se presentarán los algoritmos más relevantes en este campo.

2.3.1. Introducción al seguimiento de objetos

El seguimiento de objetos es un proceso que complementa la salida de los modelos de detección. Mientras que un modelo de detección opera sobre cada fotograma de forma independiente, identificando y localizando objetos como si fueran imágenes estáticas, el MOT actúa sobre estas detecciones para establecer una correspondencia temporal. La función esencial del MOT es asignar un identificador único a cada objeto detectado en un fotograma y mantener dicho identificador de manera consistente a lo largo de la secuencia de vídeo. Esto permite reconstruir las trayectorias de los objetos y analizar su comportamiento dinámico en la escena.

Para realizar este seguimiento, el MOT se basa en la información temporal y espacial de las detecciones. Utiliza técnicas de predicción y asociación para determinar la continuidad de los objetos a lo largo del tiempo, teniendo en cuenta factores como la posición, velocidad y apariencia de los objetos. El algoritmo utilizado para el seguimiento puede variar en complejidad, desde enfoques simples que utilizan filtros de Kalman para predecir la posición futura de un objeto, hasta métodos más avanzados que incorporan redes neuronales profundas para aprender características de apariencia y mejorar la robustez del seguimiento.

El filtro de Kalman es un algoritmo de estimación recursivo fundamental en el seguimiento de objetos. Funciona como un estimador óptimo para sistemas dinámicos lineales, permitiendo predecir el estado futuro de un objeto (como su posición y velocidad) a partir de una serie de mediciones ruidosas o incompletas a lo largo del tiempo. Su proceso se basa en dos etapas cíclicas:

- **Predicción:** Utiliza un modelo dinámico del movimiento esperado del objeto para estimar su estado en el siguiente instante de tiempo.
- **Actualización:** Incorpora la nueva medición (detección) obtenida en ese instante para corregir la predicción inicial, ponderando la información del modelo y la medición según su incertidumbre asociada.

Este ciclo permite al filtro refinarse continuamente la estimación del estado del objeto, suavizar las trayectorias y manejar eficazmente el ruido inherente a las mediciones del detector. Es una herramienta clave para mantener la identidad de los objetos entre fotogramas, especialmente cuando las detecciones son intermitentes o imprecisas.

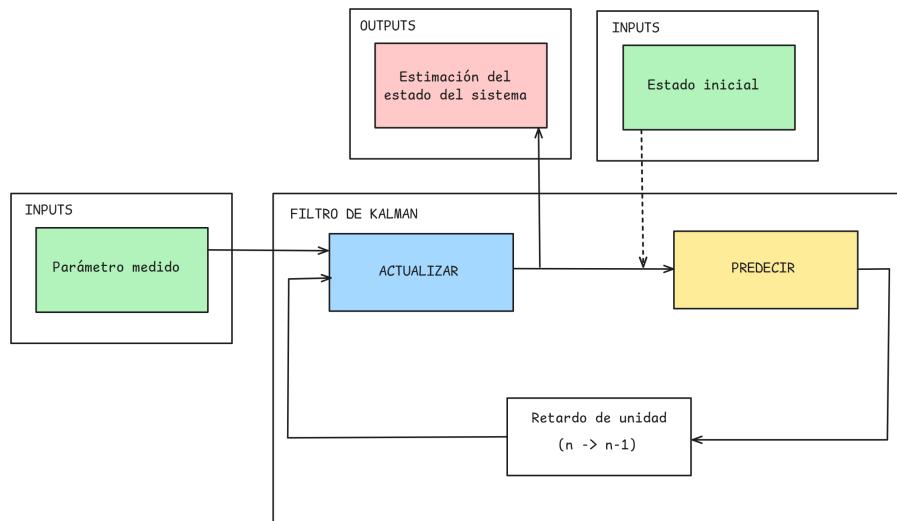


Figura 2.17: Diagrama de flujo del filtro de Kalman.

La Figura 2.17 representa el funcionamiento de un filtro de Kalman, un algoritmo muy utilizado para estimar el estado de un sistema dinámico en presencia de ruido e incertidumbre. A continuación se explica cada bloque:

- **INPUTS: Parámetro medido** Este bloque representa las mediciones que se obtienen del sistema. Estas mediciones contienen errores (ruido), por lo que no se usan directamente, sino que se pasan al filtro para su procesamiento.

- **INPUTS: Estado inicial** Proporciona una estimación inicial del estado del sistema y su incertidumbre asociada. Esta información se utiliza para arrancar el filtro de Kalman.
- **ACTUALIZAR** Este es uno de los dos pasos principales del filtro de Kalman. Combina la predicción previa con la medición actual para actualizar la estimación del estado del sistema y ajusta la incertidumbre de esa estimación.
- **PREDECIR** El otro paso clave del filtro utiliza el modelo del sistema para predecir el siguiente estado y su incertidumbre, basándose en la estimación anterior y considerando el retardo de una unidad.
- **Retardo de unidad ($n \rightarrow n-1$)** Este bloque representa el almacenamiento del estado estimado en el instante anterior para ser utilizado en la siguiente predicción.

Este es el resultado final del filtro: una estimación refinada del estado actual del sistema, que es más precisa que la simple medición directa.

En conjunto, el filtro de Kalman realiza un ciclo continuo de predicción y corrección, usando tanto el modelo del sistema como las mediciones reales, para obtener una estimación óptima del estado.

2.3.2. BYTETrack

BYTETrack [28] es un algoritmo avanzado de MOT que se inscribe dentro del paradigma de seguimiento por detección (*tracking-by-detection*). Este paradigma consiste en detectar objetos en cada fotograma de forma independiente y luego asociar estas detecciones a lo largo del tiempo para construir trayectorias coherentes. La innovación fundamental de BYTETrack reside en su novedoso método de asociación de datos, denominado BYTE, que aborda explícitamente un problema común en MOT: el manejo de detecciones con baja puntuación de confianza.

Mientras que la mayoría de los algoritmos de seguimiento descartan las detecciones por debajo de un cierto umbral de confianza para evitar la introducción de falsos positivos, BYTETrack reconoce que estas detecciones de baja confianza a menudo corresponden a objetos reales que están parcialmente ocluidos o cuya apariencia ha cambiado temporalmente. Descartarlas puede llevar a la pérdida de trayectorias y a una menor precisión general del seguimiento. Esta estrategia de asociación en dos pasos permite a BYTETrack recuperar objetos reales incluso cuando la confianza del detector disminuye debido a occlusiones o desenfoque, manteniendo la continuidad de las trayectorias. Al separar claramente las detecciones de alta y baja confianza y utilizarlas de manera diferenciada en el proceso de asociación, BYTETrack logra una notable mejora en la robustez del seguimiento, reduce significativamente la fragmentación de las trayectorias (medida por métricas como IDF1) y maneja eficazmente las variaciones en la calidad de las detecciones, todo ello manteniendo una alta eficiencia computacional.

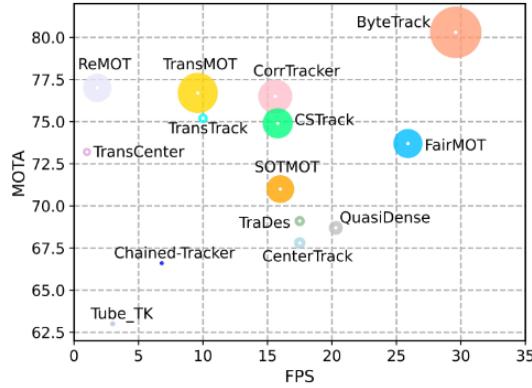


Figura 2.18: Comparativa de rendimiento de BYTETrack con otros algoritmos de seguimiento.
Extraído de [28, fig. 1, p. 1].

La Figura 2.18 presenta una comparativa de rendimiento que evidencia la superioridad de BYTETrack frente a otros algoritmos de seguimiento, según los resultados publicados en [28]. Como se observa, BYTETrack no solo alcanza una mayor precisión, medida por la métrica MOTA (Multiple Object Tracking Accuracy), sino que también demuestra una velocidad de procesamiento superior. Estas características lo posicionan como una solución particularmente eficaz y atractiva para aplicaciones que demandan seguimiento de objetos en tiempo real.

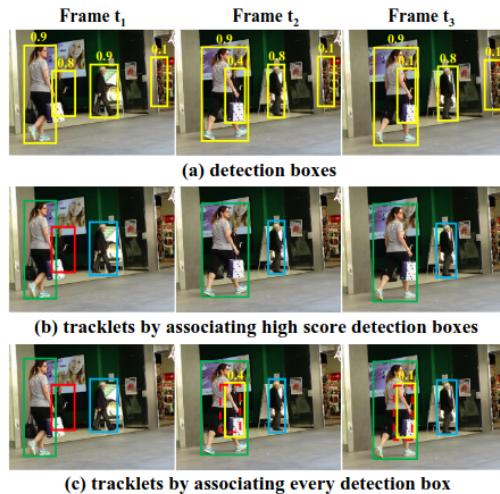


Figura 2.19: Ejemplo de detección y seguimiento de objetos utilizando BYTETrack. Extraído de [28, fig. 2, p. 2].

La Figura 2.19 ilustra el proceso de BYTETrack a través de tres fotogramas consecutivos (τ_1, τ_2, τ_3) de una secuencia de vídeo. En (a), se observan las detecciones iniciales que superan un umbral de confianza (p. ej., 0.5). La sección (b) muestra las trayectorias generadas al asociar exclusivamente las detecciones de alta confianza. Por el contrario, (c) presenta el resultado final de BYTETTrack, que integra también las detecciones de baja confianza en el proceso de asociación. Esta comparación evidencia cómo la estrategia de BYTETTrack permite mantener la continuidad de las trayectorias frente a desafíos como occlusiones parciales y variaciones en la confianza de las detecciones, logrando así una representación más precisa y robusta del movimiento de los objetos en el tiempo.

El funcionamiento del algoritmo BYTETTrack es el siguiente:

Input: Una secuencia de vídeo V , un detector de objetos Det , y un umbral de confianza de detección τ .

Output: Las trayectorias \mathcal{T} de los objetos detectados en el vídeo.

1. **Inicialización:** Se inicializa el conjunto de trayectorias \mathcal{T} como vacío.
2. **Procesamiento por fotograma:** Para cada fotograma f_k en la secuencia de vídeo V :
 - 2.1. **Detección:** Se utiliza el detector Det para obtener las cajas delimitadoras y sus puntuaciones de confianza para el fotograma f_k , resultando en un conjunto de detecciones \mathcal{D}_k .
 - 2.2. **Separación de detecciones:** Se inicializan dos conjuntos vacíos: \mathcal{D}_{high} para detecciones de alta confianza y \mathcal{D}_{low} para detecciones de baja confianza. Se itera sobre cada detección d en \mathcal{D}_k :
 - Si la puntuación $d.score$ es mayor que el umbral τ , la detección d se añade a \mathcal{D}_{high} .
 - En caso contrario, la detección d se añade a \mathcal{D}_{low} .
 - 2.3. **Predicción de trayectorias:** Para cada trayectoria existente t en \mathcal{T} , se predice su nueva ubicación utilizando un Filtro de Kalman.
 - 2.4. **Primera asociación:** Se asocian las trayectorias \mathcal{T} con las detecciones de alta confianza \mathcal{D}_{high} utilizando el *Hungarian algorithm*[18] con la métrica de similitud IoU (Intersection over Union). Las detecciones no asociadas se guardan en \mathcal{D}_{remain} y las trayectorias no asociadas en \mathcal{T}_{remain} .
 - 2.5. **Segunda asociación:** Se asocian las trayectorias restantes \mathcal{T}_{remain} con las detecciones de baja confianza \mathcal{D}_{low} utilizando otra métrica de similitud (Similarity#2, usualmente IoU). Las trayectorias que siguen sin asociarse se guardan en $\mathcal{T}_{re-remain}$. Solo se asocian detecciones de baja confianza a trayectorias que no pudieron ser asociadas con detecciones de alta confianza.
 - 2.6. **Eliminación de trayectorias no asociadas:** Se eliminan de \mathcal{T} las trayectorias que quedaron en $\mathcal{T}_{re-remain}$ (aquellas que no se pudieron asociar ni en la primera ni en la segunda etapa) si han permanecido sin asociar durante un número determinado de fotogramas (definido por el parámetro `track_buffer`).
 - 2.7. **Inicialización de nuevas trayectorias:** Se itera sobre las detecciones de alta confianza que no fueron asociadas (\mathcal{D}_{remain}). Cada una de estas detecciones se considera el inicio de una nueva trayectoria y se añade al conjunto \mathcal{T} .
3. **Retorno:** Una vez procesados todos los fotogramas, se devuelve el conjunto final de trayectorias \mathcal{T} .

Con todo esto, BYTETrack logra un seguimiento robusto y preciso de múltiples objetos en movimiento, incluso en condiciones desafiantes como occlusiones parciales y cambios de apariencia. Su enfoque innovador para manejar detecciones de baja confianza lo distingue de otros algoritmos de seguimiento y lo convierte en una opción atractiva para aplicaciones en tiempo real.

2.3.3. Métricas de evaluación en seguimiento de objetos múltiples

Las métricas de evaluación son fundamentales para medir el rendimiento de los algoritmos de MOT. Estas métricas permiten cuantificar la precisión y la robustez del seguimiento, facilitando la comparación entre diferentes enfoques y configuraciones.

En el contexto de MOT, los errores pueden clasificarse en tres categorías principales:

- **Errores de detección:** Ocurren cuando el sistema predice detecciones que no existen en la verdad de referencia, o cuando falla en predecir detecciones que sí están presentes en la verdad fundamental. Estos corresponden a los Falsos Positivos (FP) y Falsos Negativos (FN) respectivamente.
- **Errores de asociación:** Se producen cuando el sistema asigna el mismo ID de predicción (prID) a dos detecciones que tienen diferentes IDs en la verdad de referencia (gtID), o cuando asigna diferentes prIDs a dos detecciones que deberían tener el mismo gtID. El caso más común es el Cambio de Identidad (IDS - IDentity SWitch).
- **Errores de localización:** Ocurren cuando las detecciones predichas (prDets) no están perfectamente alineadas espacialmente con las detecciones de la verdad de referencia (gtDets). La calidad de esta alineación se mide típicamente con la métrica IoU (Intersection over Union).

A continuación se presentan las métricas más relevantes en este campo, que cuantifican estos errores de diversas maneras:

- **Multiple Object Tracking Accuracy (MOTA):** Es una de las métricas más consolidadas y ampliamente utilizadas para evaluar el rendimiento general de un algoritmo MOT. MOTA agrega tres tipos de errores principales que pueden ocurrir durante el seguimiento:
 - Falsos Negativos (FN): Objetos reales presentes en la escena que el algoritmo de seguimiento no detecta o no sigue.
 - Falsos Positivos (FP): Detecciones o trayectorias generadas por el algoritmo de seguimiento que no corresponden a ningún objeto real.
 - Cambios de Identidad (IDS - IDentity SWitches): Ocurren cuando un objeto que ya está siendo seguido se le asigna incorrectamente un nuevo identificador, o cuando se intercambian los identificadores entre dos objetos seguidos.

La fórmula es:

$$MOTA = 1 - \frac{\sum_t (FN_t + FP_t + IDS_{t,i})}{\sum_t GT_t} \quad (2.7)$$

donde FN_t , FP_t , e $IDS_{t,i}$ son el número de falsos negativos, falsos positivos y cambios de identidad en el fotograma t , respectivamente. GT_t es el número total de objetos reales (verdad de referencia) en el fotograma t . El sumatorio se realiza sobre todos los fotogramas de la secuencia. Un valor de MOTA más alto indica un mejor rendimiento, con un máximo teórico de 1 (o 100%). Sin embargo, MOTA puede ser negativo si el número de errores supera el número de objetos reales. Aunque es una métrica integral, tiende a dar más peso a la precisión de la detección que a la consistencia de la identidad a largo plazo.

- **Multiple Object Tracking Precision (MOTP):** Esta métrica evalúa la precisión de la localización de los objetos a lo largo del tiempo. Se calcula como la media de las distancias entre las cajas delimitadoras predichas y las cajas delimitadoras reales (verdad de referencia) para todas las asociaciones correctas. Matemáticamente, se expresa como:

$$MOTP = \frac{\sum_{t,i} d_{t,i}}{\sum_t c_t} \quad (2.8)$$

donde $d_{t,i}$ es la distancia (generalmente IoU) entre la predicción y la verdad de referencia para el objeto i en el fotograma t , y c_t es el número total de asociaciones

correctas en el fotograma t . Un valor de MOTP más alto indica una mayor precisión en la localización de los objetos. A diferencia de MOTA, MOTP no penaliza los errores de identidad, sino que se centra exclusivamente en qué tan precisas son las localizaciones de los objetos cuando se asocian correctamente.

- **ID F1 Score (IDF1):** Esta métrica se centra específicamente en la capacidad del algoritmo de seguimiento para mantener correctamente la identidad de los objetos a lo largo del tiempo. Es la media armónica de la Precisión de ID (IDP) y el Recall de ID (IDR).

- IDP (ID Precision): Proporción de detecciones correctamente asignadas a una trayectoria (ID) respecto al total de detecciones asignadas.
- IDR (ID Recall): Proporción de objetos reales correctamente identificados y seguidos a lo largo de su trayectoria respecto al total de objetos reales.

La fórmula es:

$$IDF1 = \frac{2 \cdot IDTP}{2 \cdot IDTP + IDFP + IDFN} \quad (2.9)$$

donde $IDTP$ (ID True Positives) son los verdaderos positivos en términos de asignación de identidad correcta a lo largo de las trayectorias, $IDFP$ (ID False Positives) son las asignaciones de identidad incorrectas, y $IDFN$ (ID False Negatives) son las identidades de objetos reales que no fueron correctamente mantenidas. Un valor de IDF1 más alto (hasta 1 o 100 %) indica una mejor consistencia en el seguimiento de la identidad. Es particularmente útil para evaluar el rendimiento en escenarios con occlusiones prolongadas o interacciones complejas entre objetos.

- **Higher Order Tracking Accuracy (HOTA):** Es una métrica más reciente diseñada para proporcionar una evaluación más equilibrada y completa del rendimiento del MOT. HOTA descompone explícitamente el rendimiento en precisión de detección, precisión de asociación y precisión de localización. Se calcula como la media geométrica de la Precisión de Detección (DetA) y la Precisión de Asociación (AssA), donde cada una de estas componentes considera la precisión de localización (IoU).

$$HOTA = \sqrt{DetA \cdot AssA} \quad (2.10)$$

- DetA (Detection Accuracy): Mide qué tan bien se detectan los objetos, promediado sobre diferentes umbrales de IoU.
- AssA (Association Accuracy): Mide qué tan bien se asocian las detecciones correctas para formar trayectorias consistentes, también promediado sobre umbrales de IoU.

HOTA varía entre 0 y 1 (o 0 % y 100 %), donde valores más altos indican un mejor rendimiento. Se considera que HOTA ofrece una visión más matizada que MOTA, ya que penaliza de forma más equilibrada los diferentes tipos de errores y es sensible a la calidad de la localización.

CAPÍTULO 3

Diseño e implementación de la solución

En este capítulo se abordará en profundidad el diseño y la implementación del sistema de visión artificial propuesto. Se iniciará con un análisis detallado del problema a resolver, definiendo los desafíos inherentes a la detección y seguimiento de objetos en movimiento y los requisitos específicos del sistema, como la operación en tiempo real. A continuación, se describirá el proceso de entrenamiento de los modelos de detección de objetos, la metodología para la creación y anotación del conjunto de datos de canicas, y los parámetros de entrenamiento utilizados. Posteriormente, se presentará una descripción global del sistema, detallando su arquitectura modular y el flujo de datos entre sus componentes. Se profundizará en el diseño específico de cada una de las etapas que conforman el sistema: la captura de imágenes, la inferencia del modelo de detección, el seguimiento de los objetos mediante el algoritmo BYTETrack y la escritura de los resultados. Finalmente, se explorarán y justificarán las diversas estrategias de segmentación del sistema (secuencial, basada en hilos, basada en procesos, y optimizaciones con memoria compartida o aceleración por hardware específico) que se han considerado e implementado con el objetivo de optimizar el rendimiento y la eficiencia del procesamiento en la plataforma NVIDIA Jetson.

3.1 Análisis del problema

El desafío central abordado en este trabajo consiste en el desarrollo y la implementación de un sistema de visión artificial capaz de realizar el seguimiento de múltiples objetos en movimiento en tiempo real y poder detectar sus posibles defectos. Este sistema se fundamenta en la utilización de la plataforma de hardware NVIDIA Jetson, reconocida por su capacidad para ejecutar tareas de IA de manera eficiente en términos energéticos y computacionales. La tarea principal del sistema es procesar una secuencia de vídeo, identificar los objetos presentes en cada fotograma mediante un modelo de detección de objetos basado en redes neuronales profundas, y posteriormente, aplicar un algoritmo de seguimiento para mantener la identidad de cada objeto a lo largo del tiempo, reconstruyendo así sus trayectorias.

Un requisito fundamental y crítico para la viabilidad del sistema es su capacidad para operar en tiempo real. Esto impone una restricción estricta sobre la velocidad de procesamiento: el tiempo total necesario para analizar un fotograma individual, incluyendo tanto la detección como el seguimiento de los objetos, debe ser inferior al intervalo de tiempo que transcurre entre fotogramas consecutivos en la secuencia de vídeo. Por ejemplo, para un vídeo a 30 fotogramas por segundo, el procesamiento completo de cada fotograma

debe completarse en menos de 33.3 milisegundos. Cumplir con esta exigencia es particularmente desafiante dadas las limitaciones inherentes de los dispositivos embebidos como los de la serie Jetson, que, aunque potentes, disponen de recursos computacionales y memoria significativamente menores en comparación con sistemas de escritorio o servidores.

Dada la dificultad de acceder a un entorno industrial real para llevar a cabo las pruebas experimentales —como podría ser una línea de producción activa en una fábrica de conservas, una planta de ensamblaje de componentes electrónicos o una instalación de procesamiento de alimentos—, se optó por utilizar un entorno simulado controlado. Este entorno sustituye los objetos industriales por elementos más manejables y disponibles, específicamente, canicas de diversos colores. Estas canicas, al moverse sobre una superficie o cinta transportadora improvisada, simulan el flujo de objetos que se encontraría en una línea de producción. La utilización de este entorno simulado ofrece ventajas significativas para la fase de desarrollo y evaluación: permite realizar pruebas de manera sistemática y repetible, facilita la variación controlada de parámetros (como la velocidad de los objetos, la iluminación o la densidad de objetos) y posibilita la obtención de datos cuantitativos precisos sobre el rendimiento del sistema en diferentes condiciones. Aunque este entorno simplifica la complejidad del mundo real, los resultados y las conclusiones obtenidas proporcionan una base sólida y pueden ser extrapolados, con las debidas consideraciones, para predecir el comportamiento y la eficacia del sistema en escenarios industriales auténticos.



Figura 3.1: Ejemplo de entorno simulado con canicas.

Para las pruebas experimentales, se configuró un entorno simulado como el ilustrado en la Figura 3.1. Este entorno utiliza canicas de cuatro colores distintos (blanco, negro, azul y verde). Con el objetivo de simular la detección de anomalías, se consideraron tanto canicas sin defectos como canicas con defectos para cada uno de los colores. Como defecto, se añadió una mancha de un color diferente al de la canica, presentando diversas formas. Esta distinción duplica el número total de clases que el sistema debe identificar, alcanzando un total de ocho categorías (cuatro colores sin defecto y cuatro colores con defecto).

3.2 Entrenamiento de los modelos

La selección y el entrenamiento de los modelos de detección de objetos constituyen una fase crítica en el desarrollo del sistema propuesto, ya que de ello dependen directamente la precisión y la robustez del sistema final. Tras un análisis comparativo, se optó por los detectores de una etapa frente a los de dos etapas, dada su superior eficiencia para aplicaciones que exigen procesamiento en tiempo real. Dentro de esta categoría, se seleccionó la familia de modelos YOLO. Esta elección se fundamenta en su reconocido equilibrio entre velocidad de inferencia y precisión, así como en la disponibilidad de un robusto ecosistema de herramientas que facilitan tanto el entrenamiento como el despliegue. Alternativas como SSD, si bien competentes, no ofrecían el mismo conjunto de ventajas en términos de comunidad de soporte y facilidad de integración para los fines de este proyecto.

Como modelos pertenecientes a la familia de arquitecturas YOLO, se han seleccionado diversas variantes para una evaluación exhaustiva: YOLOv5 (específicamente las versiones 'n' y 'm'), YOLOv8 (también en sus variantes 'n' y 's') y YOLOv11 (en sus versiones 'n', 's', 'm' y 'l'). Esta elección se basa en varios factores clave. En primer lugar, estas familias de modelos YOLO son conocidas por su excelente equilibrio entre velocidad de inferencia y precisión de detección, lo que las hace particularmente adecuadas para aplicaciones que requieren procesamiento en tiempo real, como es el caso de este proyecto. En segundo lugar, estos modelos, especialmente a través de la implementación proporcionada por la biblioteca Ultralytics, ofrecen una gran flexibilidad en términos de escalabilidad (con las múltiples variantes mencionadas que difieren en tamaño y complejidad) y facilidad de uso para el entrenamiento, la validación y el despliegue.

Una vez seleccionado los modelos, el siguiente paso crítico es la creación y preparación del conjunto de datos (dataset). Este conjunto de datos es la base sobre la cual los modelos aprenderán a identificar y localizar los objetos de interés (en este caso, las canicas de diferentes colores y con/sin defectos). La calidad y representatividad del dataset tienen un impacto directo y significativo en el rendimiento final de los modelos. Un dataset bien construido debe incluir una variedad suficiente de ejemplos que cubran las diferentes condiciones que el sistema podría encontrar en el entorno real, como variaciones en la iluminación, ángulos de visión, occlusiones parciales y la diversidad intrínseca de los objetos mismos. El proceso de creación del dataset implica la captura de imágenes o vídeos del entorno simulado, seguido de una meticulosa fase de etiquetado (anotación), donde se marcan manualmente las cajas delimitadoras (bounding boxes) alrededor de cada objeto de interés y se les asigna la etiqueta de clase correspondiente (p. ej., 'canica_azul_defecto'). Este proceso, aunque laborioso, es indispensable para proporcionar a los modelos la "verdad fundamental" (ground truth) necesaria para su aprendizaje supervisado.

Para el etiquetado de las imágenes, se utilizó la herramienta Computer Vision Annotation Tool (CVAT)^[5] de código abierto, que permite realizar anotaciones precisas y eficientes en imágenes, como se muestra en la Figura 3.2. Permite la creación de diferentes tipos de anotaciones, como cajas delimitadoras, polígonos y puntos clave ademas de la exportación de los datos anotados en varios formatos compatibles con diferentes frameworks de aprendizaje profundo. En este caso, se optó por el formato YOLO, que es ampliamente utilizado y compatible con la implementación de Ultralytics.

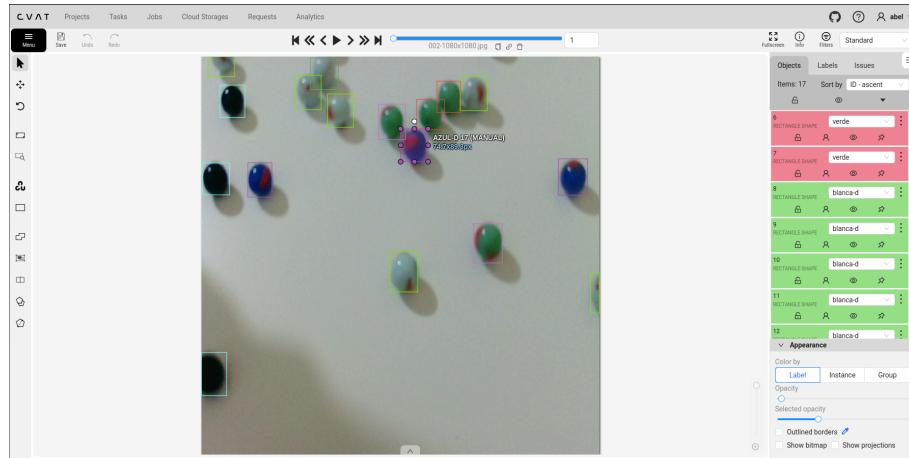


Figura 3.2: Ejemplo de anotación de imágenes utilizando CVAT.

Este formato sigue una estructura de texto simple, donde cada línea representa una anotación para un objeto en la imagen. Cada línea contiene cinco valores: el índice de la clase (0 para canica blanca, 1 para canica negra, 2 para canica azul, 3 para canica verde, 4 para canica blanca con defecto, 5 para canica negra con defecto, 6 para canica azul con defecto y 7 para canica verde con defecto), seguido de las coordenadas normalizadas del centro de la caja delimitadora (x, y) y su ancho y alto (w, h), todos ellos en relación a las dimensiones de la imagen.

Volviendo al dataset, se capturaron un total de 600 imágenes, de las cuales el 80% se utilizaron para el entrenamiento, el 10% para la validación y el 10% restante para el testeo del modelo. Esta división es crucial para garantizar que el modelo no solo aprenda a detectar los objetos en las imágenes de entrenamiento, sino que también generalice bien a nuevas imágenes que no ha visto antes. La validación se utiliza para ajustar los hiperparámetros del modelo y evitar el sobreajuste (overfitting), mientras que el conjunto de testeo proporciona una evaluación final del rendimiento del modelo.

Como se ha mencionado en la sección 2.1.5 y se detalla en la Tabla 3.1, los modelos YOLO presentan diversas variantes que difieren en tamaño, complejidad, latencia y precisión. Esta tabla comparativa es fundamental para seleccionar el modelo que mejor equilibre la velocidad de procesamiento y la precisión requerida.

Para este trabajo, se han entrenado y evaluado experimentalmente las siguientes variantes: YOLOv5n, YOLOv5m, YOLOv8n, YOLOv8s, YOLO11n, YOLO11s, YOLO11m y YOLO11l.

Familia	Variante	Tamaño (px)	Parámetros (M)	Latencia CPU ONNX (ms)	Latencia GPU (ms)	GPU (para Latencia)
YOLOv5	nu	640	2.6	73.6	1.06	A100 TensorRT
	mu	640	25.1	233.9	1.86	
YOLOv8	n	640	3.2	80.4	0.99	A100 TensorRT
	s	640	11.2	128.4	1.20	
YOLO11	n	640	2.6	56.1 ± 0.8	1.5 ± 0.0	
	s	640	9.4	90.0 ± 1.2	2.5 ± 0.0	
	m	640	20.1	183.2 ± 2.0	4.7 ± 0.1	T4 TRT10
	l	640	25.3	238.6 ± 1.4	6.2 ± 0.1	
	x	640	56.9	462.8 ± 6.7	11.3 ± 0.2	

Tabla 3.1: Análisis comparativo de variantes de YOLO (v5, v8, v11) indicando tamaño, parámetros, latencias CPU/GPU y la GPU específica utilizada para la medición de latencia GPU.

Para el entrenamiento de los modelos se han seleccionado los siguientes hiperparámetros:

- **Tasa de aprendizaje (learning rate):** Se ha utilizado un valor inicial de 0.01, con un ajuste posterior basado en la tasa de convergencia observada durante el entrenamiento.
- **Número de épocas (epochs):** Se han realizado 30 épocas, para permitir que el modelo aprenda de manera efectiva sin caer en el sobreajuste.
- **Tamaño del lote (batch size):** Se ha establecido en 16, lo que permite un equilibrio entre la velocidad de entrenamiento y la utilización de memoria.
- **Tamaño de imagen (image size):** Se ha utilizado una resolución de 640x640 píxeles, que es un tamaño estándar para los modelos YOLO y proporciona un buen compromiso entre precisión y velocidad.
- **Optimizador:** Se ha utilizado el optimizador AdamW, que es conocido por su eficacia en el entrenamiento de modelos de aprendizaje profundo.
- **Tasa de aumento de datos (data augmentation):** Se han aplicado técnicas de aumento de datos como rotación, cambio de brillo y contraste, y recortes aleatorios para mejorar la generalización del modelo.
- **Device:** Se ha utilizado una Jetson AGX Xavier, que proporciona un entorno de hardware optimizado para el entrenamiento.

Los resultados del entrenamiento se han evaluado utilizando el conjunto de validación, y se han registrado métricas como la precisión (precision), la recuperación (recall) y el mAP (mean Average Precision). Estas métricas son fundamentales para entender el rendimiento del modelo y su capacidad para generalizar a nuevos datos.

Modelo	Tiempo (h) ↓	Precisión ↑	Recall ↑	mAP50 ↑	mAP50-95 ↑
YOLOv5nu	0.128	0.898	0.922	0.939	0.759
YOLOv5mu	0.333	0.952	0.928	0.955	0.790
YOLOv8n	0.137	0.934	0.905	0.950	0.770
YOLOv8s	0.202	0.943	0.928	0.955	0.786
YOLO11n	0.140	0.950	0.882	0.939	0.761
YOLO11s	0.192	0.941	0.936	0.963	0.796
YOLO11m	0.377	0.941	0.936	0.963	0.796
YOLO11l	0.485	0.954	0.947	0.967	0.801

Tabla 3.2: Comparativa del rendimiento de los modelos YOLOv5, YOLOv8 y YOLO11 en términos de tiempo de entrenamiento, precisión, recall y mAP.

La Tabla 3.2 resume los resultados del entrenamiento para las variantes de YOLOv5, YOLOv8 y YOLO11. Se observa una tendencia general: a medida que aumenta la complejidad del modelo (de las versiones más pequeñas a las más grandes dentro de cada familia), tanto la precisión como el recall experimentan una mejora constante. Esto sugiere que los modelos de mayor tamaño poseen una capacidad superior para aprender representaciones de características más ricas y discriminativas, resultando en una detección de objetos más efectiva. No obstante, esta mejora en el rendimiento se acompaña de un incremento en el tiempo de entrenamiento y la demanda de recursos computacionales, lo que refleja el inherente compromiso entre la capacidad del modelo y la eficiencia del proceso de aprendizaje.

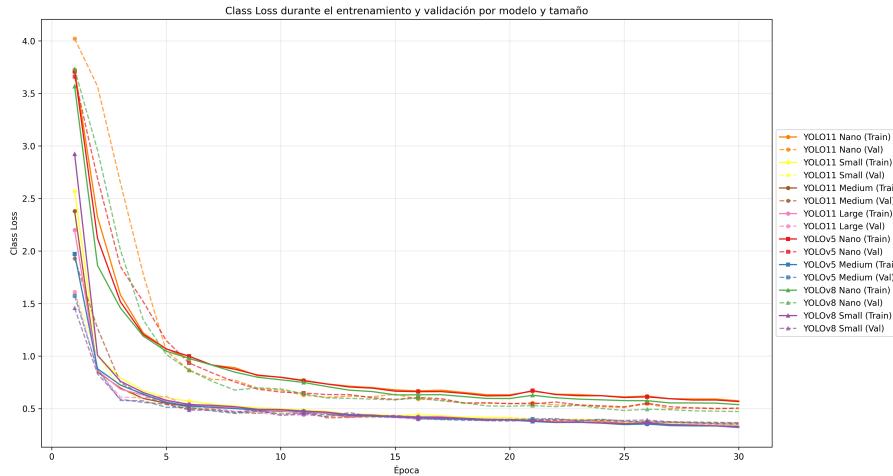


Figura 3.3: Curvas de pérdida de entrenamiento y validación para las distintas tallas de modelos YOLOv5, YOLOv8 y YOLOv11.

La Figura 3.3 presenta las curvas de pérdida (loss) durante el entrenamiento y la validación para las diferentes variantes de YOLOv5, YOLOv8 y YOLOv11. Estas gráficas evidencian una progresiva disminución de la función de pérdida a lo largo de las épocas, lo cual es un indicador clave de que cada modelo está aprendiendo a generalizar a partir de los datos para la tarea de detección de objetos. Es notable que, en todas las familias, la pérdida de validación sigue una tendencia descendente similar a la de entrenamiento, sugiriendo una buena capacidad de generalización y la ausencia de un sobreajuste significativo. Se aprecia un descenso pronunciado de la pérdida en las épocas iniciales, indicativo de una rápida asimilación de patrones, y conforme avanza el entrenamiento, la tasa de reducción disminuye gradualmente hasta alcanzar una meseta. Este comportamiento es característico en el entrenamiento de modelos de aprendizaje profundo y señala la convergencia hacia un mínimo local en la función de pérdida, donde las mejoras adicionales resultan marginales.

3.3 Descripción del sistema

El sistema está organizado como una serie de etapas de procesamiento secuenciales que trabajan de forma coordinada para lograr la detección y seguimiento de objetos en tiempo real. Como se muestra en la Figura 3.4, el sistema recibe como entrada imágenes de una cámara y consta de cuatro componentes principales: un módulo de captura de imágenes que obtiene los fotogramas del vídeo, un módulo de inferencia que ejecuta el modelo de detección de objetos, un módulo de seguimiento que implementa el algoritmo BYTETrack para mantener la identidad de los objetos detectados, y un módulo de escritura que gestiona la salida del sistema.

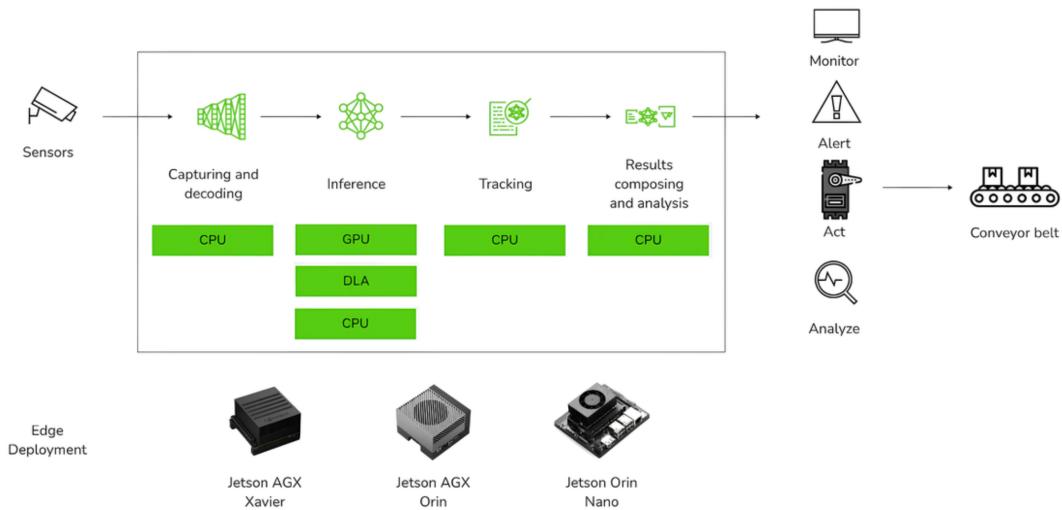


Figura 3.4: Figura del sistema propuesto.

Los resultados del sistema pueden utilizarse de diversas formas, monitorización en tiempo real a través de visualización de las detecciones, generación de alertas basadas en reglas predefinidas, interacción con sistemas de control automatizado para realizar acciones específicas, y almacenamiento de datos para su posterior análisis y extracción de métricas que permitan optimizar procesos industriales.

La arquitectura del sistema se ha diseñado estratégicamente para maximizar el rendimiento computacional y la eficiencia en el uso de recursos, asegurando una operación coordinada entre sus componentes modulares. Los datos fluyen secuencialmente a través de las distintas etapas, permitiendo un procesamiento continuo y eficaz de la información visual. Esta estructura modular intrínseca no solo simplifica el mantenimiento y las futuras actualizaciones, sino que también dota al sistema de una gran flexibilidad para adaptarse a diferentes requisitos operativos y escenarios de despliegue. Aunque optimizado específicamente para la plataforma NVIDIA Jetson, su diseño modular facilita su potencial portabilidad a otras plataformas hardware de NVIDIA, como GPUs de escritorio o servidores de alto rendimiento, siempre que se cumplan los requisitos de hardware y software.

3.4 Diseño de las etapas del sistema

El sistema propuesto se compone de cuatro etapas principales, cada una de las cuales desempeña un papel crucial en el procesamiento y análisis de las imágenes capturadas. A continuación, se describen en detalle cada una de estas etapas.

3.4.1. Captura de imágenes

La etapa de captura de imágenes es la responsable de adquirir los fotogramas del flujo de vídeo en tiempo real. Sus funciones abarcan la configuración y el control de la cámara, la adquisición de las imágenes y su posible preprocesamiento inicial antes de ser transferidas al módulo de inferencia. Para la implementación de este módulo se ha empleado la biblioteca OpenCV, la cual ofrece una interfaz robusta y eficiente para la interacción con dispositivos de captura y el procesamiento básico de imágenes. La cámara se configura para operar a una resolución y tasa de fotogramas adecuadas a los requisitos de la aplicación. El preprocesamiento puede incluir diversas operaciones, tales como la

conversión a escala de grises, la normalización de píxeles o el redimensionamiento, adaptándose a las especificaciones del modelo de detección de objetos utilizado. En el sistema desarrollado, se empleó una cámara de alta definición capaz de alcanzar una resolución máxima de 1920x1080 píxeles y una tasa de 30 fps. Si bien se experimentó con diversas configuraciones de resolución y tasa de fotogramas según las necesidades de cada prueba, la configuración estándar para la mayoría de los experimentos fue de 640x640 píxeles a 30 fps. Esta etapa se ejecuta íntegramente en la CPU.

3.4.2. Inferencia

La etapa de inferencia constituye el núcleo computacional del sistema, siendo responsable de ejecutar el modelo de detección de objetos preentrenado sobre cada fotograma adquirido por la etapa de captura. Para esta tarea crucial, se ha seleccionado el modelo YOLO11, una variante optimizada dentro de la familia YOLO, reconocida por su equilibrio entre velocidad y precisión en tareas de detección en tiempo real. La implementación se apoya en el framework de Ultralytics [14], una biblioteca de alto nivel que simplifica significativamente el ciclo de vida de los modelos YOLO, desde el entrenamiento hasta el despliegue y la inferencia, ofreciendo una interfaz robusta y eficiente.

El proceso de inferencia comienza con la adquisición de un fotograma de vídeo, que se convierte en un tensor adecuado para la entrada del modelo. Este tensor es una representación numérica de la imagen, donde cada píxel se traduce en un valor que el modelo puede procesar. La transformación del fotograma a tensor incluye operaciones como la normalización y el redimensionamiento, asegurando que los datos estén en el formato correcto para el modelo YOLO11. La ejecución de la inferencia propiamente dicha implica cargar el modelo YOLO11 (potencialmente optimizado mediante NVIDIA TensorRT para maximizar el rendimiento en la plataforma Jetson) y pasarle el tensor de entrada. Aprovechando la aceleración por hardware (GPU y/o DLAs disponibles en los módulos Jetson), el modelo procesa la imagen y genera un conjunto de predicciones. Estas predicciones iniciales suelen ser numerosas y requieren un postprocesamiento para refinar los resultados. Este postprocesamiento, a menudo gestionado internamente por el framework Ultralytics o aplicado explícitamente, incluye la aplicación de un umbral de confianza para descartar detecciones poco fiables y la ejecución del algoritmo de NMS para eliminar cajas delimitadoras redundantes que correspondan al mismo objeto.

El resultado final de la etapa de inferencia, que se transfiere a la etapa de seguimiento, es una lista estructurada de las detecciones finales para el fotograma actual. Cada detección en esta lista contiene información esencial: las coordenadas de la caja delimitadora que localiza al objeto (comúnmente en formato (x, y, w, h) , donde (x, y) representan las coordenadas del centro de la caja, y (w, h) su anchura y altura), una puntuación de confianza que cuantifica la fiabilidad de la detección, y la etiqueta de la clase predicha para el objeto (p. ej., 'canica_azul', 'canica_verde_defecto'). La eficiencia y rapidez de esta etapa son críticas para mantener la capacidad de procesamiento en tiempo real del sistema global.

3.4.3. Seguimiento

La etapa de seguimiento es la encargada de mantener la identidad de los objetos detectados a lo largo del tiempo, asegurando que cada objeto en el flujo de vídeo conserve su etiqueta y trayectoria a pesar de las variaciones en su posición, apariencia o posibles occlusiones. Para lograr esto, se ha utilizado la implementación del algoritmo BYTETrack en el framework de Ultralytics[14], que se basa en un enfoque de seguimiento por detección explicado en la subsección 2.3.2. Este algoritmo se encarga de asociar las detecciones

generadas por la etapa de inferencia con las trayectorias existentes, utilizando tanto las detecciones de alta confianza como las de baja confianza para mejorar la robustez del seguimiento.

Para la configuración del algoritmo BYTETrack existen varios parámetros ajustables que permiten optimizar su rendimiento según las características específicas del entorno y los objetos a seguir. Estos parámetros son:

- **track_high_thresh**: Umbral de confianza para considerar una detección como de alta confianza (valor típico: 0.6). Las detecciones por encima de este umbral se utilizan en la primera etapa de asociación.
- **track_low_thresh**: Umbral de confianza para considerar una detección como de baja confianza (valor típico: 0.15). Las detecciones que se encuentren entre este umbral y **track_high_thresh** se utilizan en la segunda etapa de asociación para recuperar objetos ocluidos.
- **new_track_thresh**: Umbral de confianza mínimo para iniciar una nueva trayectoria a partir de una detección de alta confianza no asociada (valor típico: 0.6). Debe ser al menos tan alto como **track_high_thresh**.
- **track_buffer**: Número máximo de fotogramas que una trayectoria puede permanecer sin asociar antes de ser eliminada. Define la “edad” máxima de una pista perdida. Un valor típico es 30 fotogramas.
- **match_thresh**: Umbral de IoU (Intersection over Union) para la asociación entre las predicciones del Filtro de Kalman y las detecciones (valor típico: 0.8). Si la IoU es mayor que este umbral, se considera una coincidencia potencial.

Este algoritmo se basa en un ciclo continuo de predicción y corrección, donde el Filtro de Kalman se utiliza para predecir la posición futura de los objetos y suavizar las trayectorias, manejando la incertidumbre en las mediciones. La asociación de detecciones y trayectorias se realiza mediante un algoritmo de asignación, como el algoritmo Húngaro, que busca minimizar el costo total de la asociación entre las detecciones y las trayectorias existentes. Se ejecuta íntegramente en la CPU debido a la implementación del algoritmo en la biblioteca de Ultralytics.

Tras aplicar la asociación de detecciones y trayectorias, el algoritmo ofrece como salida una lista con el identificador único de cada objeto, su clase, la puntuación de confianza y las coordenadas de la caja delimitadora. Esta información es fundamental para la siguiente etapa del sistema.

3.4.4. Escritura de resultados

La etapa de escritura de resultados es responsable de gestionar la salida del sistema, que puede adoptar diversas formas según los requisitos específicos de la aplicación. Esta etapa se encarga de presentar los resultados de manera comprensible y útil, permitiendo su interpretación y análisis posterior. Las principales funciones de esta etapa incluyen la visualización de los resultados en tiempo real, la generación de alertas basadas en reglas predefinidas y el almacenamiento de datos para su posterior análisis. También se encarga de los posibles conexiones a sistemas de control automatizado, permitiendo la interacción con otros sistemas o dispositivos.



Figura 3.5: Ejemplo de salida del sistema.

La Figura 3.5 muestra un ejemplo de la salida del sistema, donde se visualizan las detecciones y trayectorias de los objetos en el flujo de vídeo. Cada objeto detectado está representado por una caja delimitadora que incluye su etiqueta de clase y un identificador único. Esta representación gráfica permite una rápida identificación y seguimiento de los objetos en movimiento, facilitando la monitorización en tiempo real del sistema.

3.5 Segmentación de las etapas del sistema

Tras la implementación de las etapas del sistema, se ha considerado la posibilidad de segmentar el sistema mediante estas etapas para mejorar el rendimiento y la eficiencia del procesamiento. La segmentación permite distribuir la carga de trabajo entre diferentes unidades de procesamiento, optimizando así el uso de los recursos disponibles en la plataforma.

La segmentación de las etapas del sistema se puede realizar de varias maneras, dependiendo de los requisitos específicos de la aplicación y de los recursos disponibles. A continuación, se describen las diferentes opciones de segmentación que se han considerado, implementado y evaluado en el sistema propuesto.

3.5.1. Secuencial

La primera y más trivial opción es la ejecución secuencial de las etapas del sistema. En este enfoque, cada etapa se ejecuta de forma consecutiva, donde la salida de una etapa se convierte en la entrada de la siguiente. Este método es el más sencillo de implementar y no requiere una configuración adicional para la comunicación entre etapas. Sin embargo, presenta limitaciones significativas en términos de rendimiento y eficiencia, ya que no aprovecha al máximo los recursos disponibles. Si se hace la analogía con un procesador, este enfoque se asemeja a un procesador no segmentado.

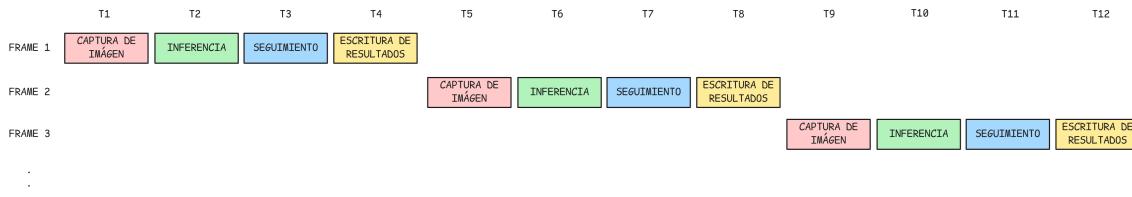


Figura 3.6: Diagrama de flujo del sistema sin segmentar.

La Figura 3.6 ilustra el flujo de datos en un sistema secuencial. En este diagrama, cada etapa del sistema se ejecuta de forma lineal, donde la salida de una etapa se convierte en la entrada de la siguiente. Este enfoque es fácil de entender y de implementar, pero no aprovecha al máximo los recursos disponibles.

3.5.2. Segmentación en hilos

La segunda opción es la segmentación del sistema en diferentes hilos. En este enfoque, cada etapa principal del sistema (captura, inferencia, seguimiento, escritura) se ejecuta en un hilo (*thread*) independiente. A diferencia del enfoque secuencial donde cada etapa debe esperar a que la anterior finalice, la segmentación por hilos permite que las etapas operen de forma concurrente, solapando sus ejecuciones. Esto puede mejorar significativamente el rendimiento (throughput) y reducir la latencia, ya que mientras una etapa espera por una operación (p. ej., E/S de la cámara), otra etapa puede estar procesando datos (p. ej., inferencia en GPU o seguimiento en CPU). Los hilos operan dentro del mismo proceso, compartiendo el mismo espacio de memoria. La comunicación y transferencia de datos (fotogramas, detecciones) entre estas etapas/hilos se gestiona mediante colas de mensajes (*Queue*) de la librería estándar de Python, que aseguran una transferencia eficiente y segura entre hilos (*thread-safe*).

En Python, un hilo es una unidad básica de ejecución. Sin embargo, al trabajar con hilos en Python, es crucial entender el impacto del Global Interpreter Lock (GIL) (Global Interpreter Lock). El GIL es un mecanismo de bloqueo mutuo (mutex) que protege el acceso al intérprete de Python. Su función principal es permitir que solo un hilo ejecute código de bytes Python (*Python bytecode*) a la vez dentro de un único proceso, incluso si el sistema dispone de múltiples núcleos de CPU. Es decir, en Python, debido al GIL, dos hilos de un mismo proceso no pueden ejecutar código Python de manera simultánea en núcleos de CPU diferentes. Esta restricción se implementó originalmente para simplificar la gestión de memoria (específicamente, el conteo de referencias) y prevenir condiciones de carrera en el acceso a objetos Python.

La principal consecuencia del GIL es que impide el verdadero paralelismo para tareas que son intensivas en CPU (*CPU-bound*) y están escritas puramente en Python. Aunque se creen múltiples hilos, solo uno podrá ejecutar código Python en un instante dado. En aplicaciones con muchos hilos compitiendo por la CPU, el GIL puede incluso introducir sobrecarga y contención, llevando a un rendimiento inferior al de una ejecución secuencial.

No obstante, el GIL no bloquea la ejecución en todas las circunstancias. Se libera automáticamente durante operaciones que no ejecutan código Python directamente, como:

- Operaciones de entrada/salida (E/S): Lectura/escritura de archivos, operaciones de red, interacción con dispositivos como cámaras.

- Llamadas a código nativo compilado: Cuando se utilizan bibliotecas como NumPy, SciPy, o las bibliotecas específicas para la ejecución en GPU (como las de CUDA/-TensorRT), que realizan el cómputo fuera del intérprete Python.
- Llamadas explícitas de espera: Como ‘time.sleep()’.

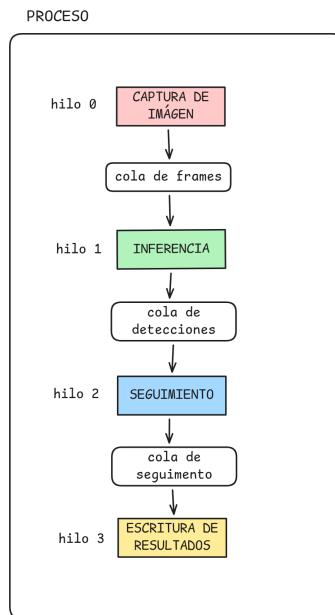


Figura 3.7: Diagrama de flujo del sistema segmentado en hilos.

La Figura 3.7 ilustra este enfoque, donde cada etapa opera en su propio hilo y se comunica mediante colas.

Considerando estas características, la segmentación basada en hilos puede ser beneficiosa para nuestro sistema. La etapa de captura de imágenes es intensiva en E/S. La etapa de inferencia, especialmente cuando se ejecuta en la GPU o DLA utilizando bibliotecas optimizadas como TensorRT, realiza la mayor parte de su trabajo en código nativo, liberando el GIL. La etapa de escritura también implica operaciones de E/S. Durante los momentos en que estas etapas liberan el GIL, otros hilos pueden progresar.

Aunque más complejo de implementar que el enfoque secuencial debido a la necesidad de sincronización y comunicación entre hilos, este modelo permite una mejor utilización de los recursos y mejora la capacidad de respuesta y el rendimiento general del sistema al solapar operaciones de E/S y cómputo intensivo (en GPU/DLA) con otras tareas.

3.5.3. Segmentación en procesos

La tercera opción es la segmentación del sistema en diferentes procesos. En este enfoque, cada etapa principal del sistema (captura, inferencia, seguimiento, escritura) se ejecuta en un proceso independiente. La comunicación y transferencia de datos entre estas etapas/procesos se gestiona mediante colas de mensajes (*Queue*). Estas colas provienen del módulo ‘multiprocessing’ de la librería estándar de Python y comparten la misma interfaz que las colas estándar del módulo ‘queue’, lo que asegura una transferencia eficiente y segura entre procesos (*process-safe*).

La principal ventaja de este enfoque es que cada etapa del sistema se ejecuta en su propio proceso independiente, lo que permite un mejor aprovechamiento de los recur-

sos disponibles. Además, al estar cada etapa en su propio proceso, se evita el problema del GIL (Global Interpreter Lock) presente en Python. Esto se debe a que cada proceso tiene su propio intérprete de Python y, por lo tanto, su propio GIL independiente. Como resultado, múltiples procesos pueden ejecutar código Python simultáneamente en diferentes núcleos de CPU, logrando un paralelismo real, a diferencia de los hilos dentro de un mismo proceso. Esto es especialmente beneficioso en sistemas con múltiples núcleos de CPU, donde cada proceso puede ejecutarse en un núcleo diferente, maximizando así el rendimiento del sistema.

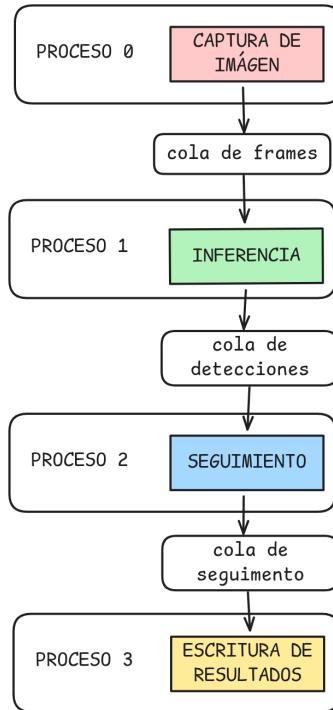


Figura 3.8: Diagrama de flujo del sistema segmentado en procesos.

La Figura 3.8 ilustra este enfoque, donde cada etapa opera en su propio proceso y se comunica mediante colas.

Sin embargo, este enfoque también presenta desventajas. La comunicación entre procesos es más costosa en términos de tiempo y recursos que la comunicación entre hilos dentro de un mismo proceso. Además, la gestión de memoria y el intercambio de datos entre procesos pueden ser más complejos, lo que puede aumentar la dificultad de implementación y mantenimiento del sistema.

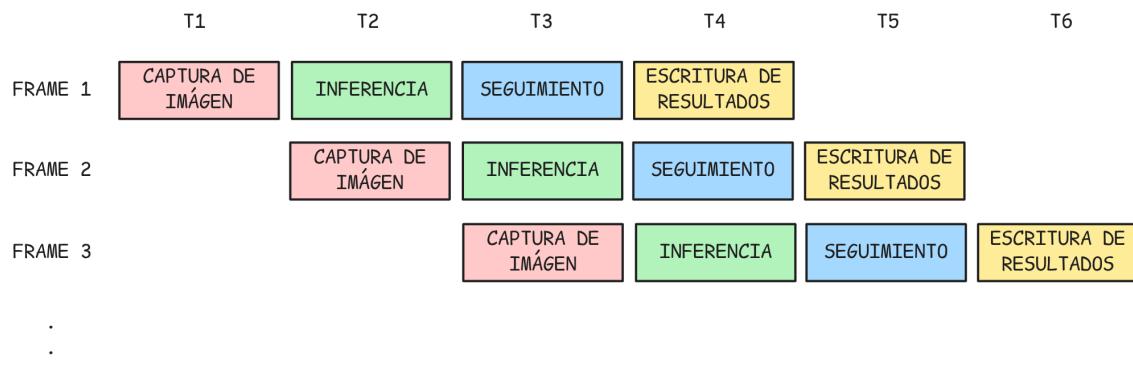


Figura 3.9: Diagrama de flujo del sistema segmentado.

La Figura 3.9 ilustra el flujo de datos en un sistema segmentado mediante procesos independientes, contrastando con el enfoque secuencial mostrado en la Figura 3.6. En este diseño segmentado, cada etapa principal (captura, inferencia, seguimiento, escritura) opera en su propio proceso, permitiendo la ejecución concurrente en diferentes núcleos de CPU si están disponibles.

Siguiendo la analogía con la arquitectura de un procesador, este enfoque se asemeja a un procesador segmentado (pipelined), donde diferentes instrucciones se encuentran en distintas fases de ejecución simultáneamente. Sin embargo, existe una diferencia fundamental: mientras que en un procesador segmentado todas las etapas avanzan sincronizadas por un ciclo de reloj común, determinado por la duración de la etapa más lenta, en nuestro sistema las etapas operan de forma asíncrona.

Cada etapa del sistema (captura, inferencia, seguimiento, escritura) tiene una duración variable y no necesariamente igual a las demás. Por ejemplo, la inferencia en la GPU puede ser mucho más rápida o lenta que la captura de imágenes o el seguimiento en la CPU. Las colas de mensajes actúan como buffers intermedios que desacoplan las etapas, permitiendo que cada una procese datos a su propio ritmo. Una etapa más rápida puede producir resultados que se acumulan en la cola de salida, mientras que una etapa más lenta consumirá datos de su cola de entrada cuando estén disponibles, esperando si la cola está vacía.

Esta asincronía, gestionada mediante colas, permite un mayor rendimiento (throughput) en comparación con el modelo estrictamente secuencial (Figura 3.6), donde cada etapa debe esperar a que la anterior finalice completamente. No obstante, si una etapa es significativamente más lenta que las demás, puede convertirse en un cuello de botella, haciendo que las colas anteriores se llenen y las posteriores permanezcan vacías, limitando el rendimiento general del sistema al ritmo de la etapa más lenta.

3.5.4. Segmentación heterogénea

La cuarta opción es la segmentación basada en computación heterogénea. Aprovechando la arquitectura heterogénea de la plataforma NVIDIA Jetson, esta opción de segmentación distribuye las tareas entre las diferentes unidades de procesamiento disponibles. La etapa de inferencia, supuestamente la más exigente computacionalmente, se descarga específicamente a los aceleradores de hardware: la GPU o uno de los DLAs (DLA0, DLA1) si están presentes en el módulo Jetson. Las demás etapas (captura, seguimiento y escritura) permanecen asignadas a la CPU.

Este enfoque permite una ejecución paralela real, donde la CPU gestiona el flujo de datos y la lógica de seguimiento mientras la GPU y/o las DLAs procesan simultáneamente los fotogramas para la detección de objetos. Para manejar los resultados que llegan de forma asíncrona desde estos aceleradores, a cada fotograma capturado se le asigna un identificador único. Esto garantiza que las detecciones se asocien correctamente con el fotograma original antes de pasar a la etapa de seguimiento, preservando así el orden temporal de la secuencia. La comunicación entre los procesos que se ejecutan en la CPU y aquellos que gestionan la inferencia en los aceleradores se realiza mediante las colas inter-proceso seguras (*process-safe queues*) descritas anteriormente.

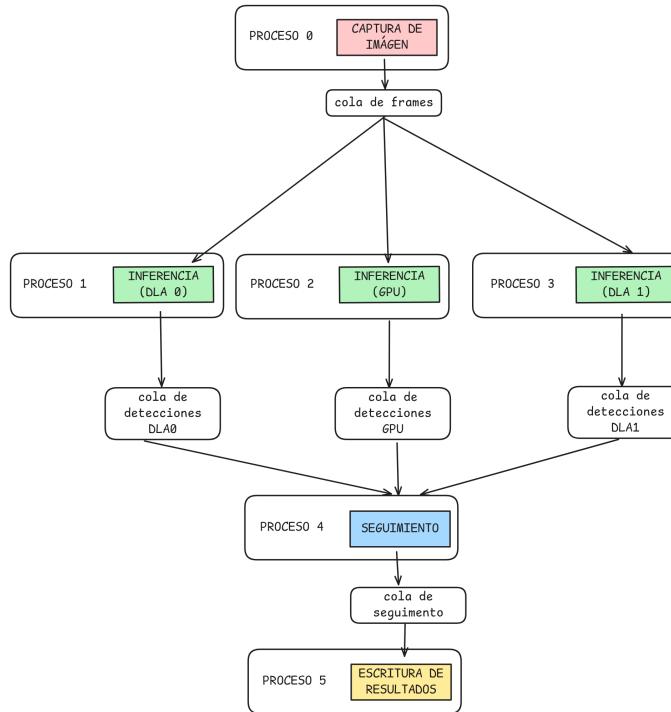


Figura 3.10: Diagrama de flujo del sistema segmentado en diferentes unidades de procesamiento.

La Figura 3.10 ilustra cómo la etapa de inferencia puede ejecutarse en paralelo en la GPU o DLA, comunicándose con la etapa de seguimiento (en CPU) a través de colas. Esta distribución optimiza el uso de los recursos especializados, acelerando significativamente el rendimiento general del sistema.

3.5.5. Segmentación basada en procesos con memoria compartida

La quinta opción de segmentación emplea procesos independientes como en la sección 3.5.3, pero busca optimizar la comunicación entre ellos utilizando memoria compartida como alternativa a las colas estándar del módulo `multiprocessing`. La transferencia de grandes volúmenes de datos, como los fotogramas de vídeo, puede volverse ineficiente con `multiprocessing` debido a la sobrecarga asociada a la serialización (*pickling*) y deserialización de objetos, así como a la posible copia de datos entre los espacios de memoria de los procesos a través de mecanismos subyacentes como pipes.

Para superar estas limitaciones, la biblioteca `multiprocessing.shared_memory` de Python permite a múltiples procesos acceder directamente a la misma región de memoria. Un proceso crea un bloque de memoria compartida, y otros procesos pueden adjuntarse a él usando su nombre único. Ambos pueden leer y escribir directamente en el *buffer* de memoria (`shm.buf`). Este acceso directo elimina los pasos de serialización/deserialización y las copias intermedias, resultando en una latencia mucho menor y un mayor ancho de banda, lo cual es especialmente beneficioso para datos grandes y estructurados como imágenes o *arrays* NumPy.

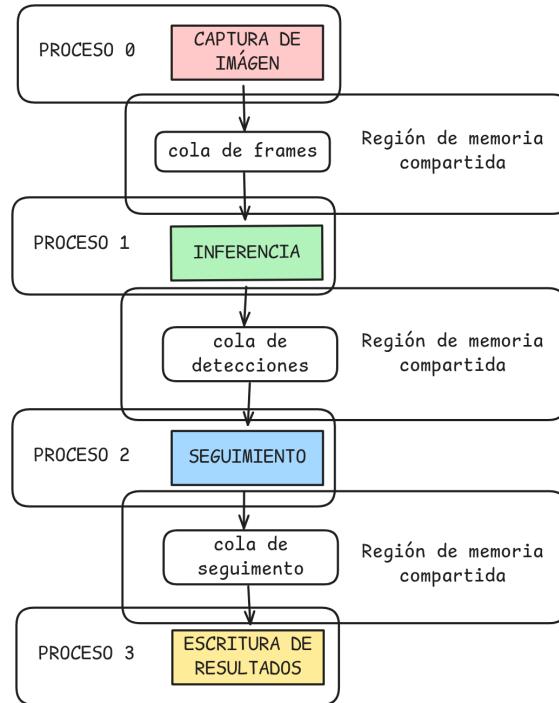


Figura 3.11: Diagrama de flujo del sistema segmentado en procesos con memoria compartida.

La Figura 3.11 ilustra este enfoque, donde cada etapa opera en su propio proceso y se comunica mediante memoria compartida. Este método permite una transferencia de datos más eficiente entre procesos, eliminando la necesidad de serialización y deserialización, lo que resulta en una latencia reducida y un mayor rendimiento general del sistema.

Sin embargo, la gestión directa de la memoria compartida requiere una implementación cuidadosa. Para facilitar su uso y gestionar el flujo de datos de manera estructurada, se ha implementado una capa de abstracción: un buffer circular. Este buffer opera sobre un bloque de memoria compartida preasignado y funciona como una cola de capacidad fija. Los datos se escriben en una posición (*tail*) y se leen desde otra (*head*). Cuando los índices alcanzan el final del *buffer*, vuelven al principio, permitiendo un uso continuo del espacio de memoria. La Figura 3.12 ilustra esta estructura conceptual.

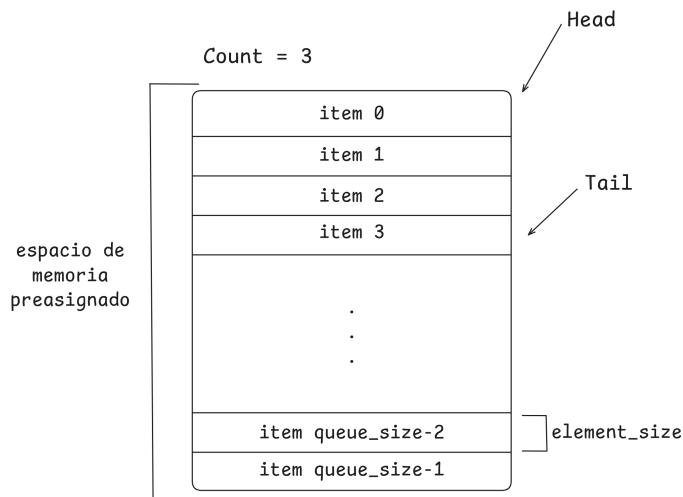


Figura 3.12: Ejemplo de buffer circular.

Dado que múltiples procesos acceden concurrentemente a la misma memoria a través del *buffer* circular, es crucial garantizar la coherencia de los datos y prevenir condiciones de carrera. A diferencia de `multiprocessing.Queue`, que gestiona la sincronización internamente, `shared_memory` no la proporciona automáticamente. Por lo tanto, el *buffer* circular implementado integra mecanismos de sincronización explícitos, como bloqueos (*Lock*) y variables de condición (*Semaphore*) del módulo `multiprocessing`. Estos controlan el acceso: un proceso productor que intente añadir datos a un buffer lleno se bloqueará hasta que un consumidor libere espacio, y viceversa, un consumidor que intente leer de un buffer vacío esperará. Este comportamiento asegura que no se pierdan datos y que las operaciones se realicen de forma segura.

La configuración de cada buffer circular requiere definir estáticamente su capacidad, el tamaño de memoria para cada elemento y un nombre único para la región de memoria compartida. Una limitación clave es la necesidad de preasignar la memoria. Una asignación incorrecta (demasiado grande o demasiado pequeña) puede llevar a desperdicio de recursos o a bloqueos frecuentes que limiten el rendimiento. Por ello, se requiere una calibración experimental para determinar los tamaños óptimos de buffer para cada enlace entre etapas, buscando el equilibrio entre uso eficiente de memoria y fluidez en el procesamiento. Además, el programador debe gestionar manualmente el ciclo de vida del bloque de memoria compartida (creación, cierre con `close()` y liberación final con `unlink()`).

En resumen, la segmentación basada en procesos con memoria compartida y un buffer circular busca ofrecer un rendimiento superior para la transferencia de grandes bloques de datos como fotogramas de vídeo, aunque esto implicaría una mayor complejidad en la implementación y gestión de la memoria.

CAPÍTULO 4

Evaluación de la solución

En este capítulo se presenta la evaluación del sistema propuesto, analizando su rendimiento y eficiencia en diferentes configuraciones. Se describen las metodologías de evaluación empleadas, las métricas de rendimiento consideradas y los resultados obtenidos en cada caso. El objetivo es proporcionar una visión clara de cómo el sistema se comporta bajo diversas condiciones y configuraciones, permitiendo identificar sus fortalezas y áreas de mejora.

4.1 Metodología de evaluación y métricas de rendimiento

Tras el desarrollo de la solución propuesta, se procede a analizar su rendimiento. La evaluación se basa en diferentes vídeos de prueba de 80 segundos (2400 fotogramas) que muestran un flujo de canicas, grabados a 640x640 píxeles y 30 fotogramas por segundo. Estos vídeos sirven como entrada estándar para el sistema, permitiendo medir diversas métricas de rendimiento según la configuración.

Para evaluar el rendimiento del sistema, se han implementado dos metodologías de ejecución:

1. **Ejecución a máxima capacidad:** Los fotogramas se entregan al sistema consecutivamente, tan pronto como el procesamiento del fotograma anterior ha concluido, utilizando colas intermedias que no descartan fotogramas. En este escenario, se mide el tiempo total que el sistema tarda en procesar la secuencia completa, lo que permite evaluar su rendimiento máximo (throughput) y compararlo con la duración real del vídeo.
2. **Simulación de entrada en tiempo real:** Los fotogramas del vídeo se suministran al sistema a una tasa fija de 30 fps (un fotograma cada 33.3 ms), emulando la entrada de una cámara. Si el sistema no logra procesar un fotograma dentro de este intervalo, dicho fotograma se descarta. Esta metodología permite cuantificar la capacidad del sistema para operar en tiempo real, midiendo el número de fotogramas procesados y perdidos.

Como métricas de rendimiento, se han considerado las siguientes:

- **Tasa de fotogramas por segundo (FPS):** Número de fotogramas completamente procesados por segundo. Esta métrica es fundamental para evaluar la capacidad del sistema para operar en tiempo real, ya que cuantifica directamente su velocidad de procesamiento. Un valor mayor o igual a 30 FPS generalmente indica que el sistema puede funcionar en tiempo real para vídeos estándar.

- **Tasa de fotogramas perdidos (LFPS):** Número de fotogramas descartados por segundo debido a la incapacidad del sistema para procesarlos dentro del intervalo temporal requerido. Esta métrica refleja la robustez del sistema bajo restricciones de tiempo real y su eficacia para mantener sincronización con la fuente de entrada. Un valor cercano a cero indica un rendimiento óptimo.
- **Potencia media consumida (W):** Potencia eléctrica media requerida por el sistema durante la ejecución, medida en vatios. Se obtiene promediando las lecturas de potencia instantánea registradas por las herramientas de monitorización hardware. Esta métrica es crucial para evaluar la viabilidad del sistema en entornos con restricciones energéticas, especialmente en aplicaciones *embedded* o en el *edge*.
- **Energía consumida (J):** Cantidad total de energía eléctrica utilizada por el sistema para procesar la secuencia completa, medida en julios. Se calcula integrando la potencia instantánea a lo largo del tiempo de ejecución. Esta métrica proporciona una perspectiva integral de la eficiencia energética y es fundamental para estimar costes operativos y requisitos de refrigeración en implementaciones a largo plazo.
- **Fotogramas por vatio (Frames/W):** Métrica compuesta que indica la eficiencia energética del procesamiento, cuantificando cuántos fotogramas se procesan por cada vatio de potencia consumida. Permite comparar directamente configuraciones con diferentes compromisos entre velocidad y consumo. Se calcula mediante la siguiente relación:

$$\text{FPS}/\text{W} = \frac{\text{Frames procesados}}{\text{Potencia (W)} \cdot \text{Tiempo (s)}} \quad (4.1)$$

donde

$$\text{Energía consumida (J)} = \text{Potencia (W)} \cdot \text{Tiempo (s)} \quad (4.2)$$

por lo tanto

$$\text{FPS}/\text{W} = \frac{\text{Frames procesados}}{\text{Energía consumida (J)}} \quad (4.3)$$

- **Speedup:** Factor de aceleración que cuantifica la mejora relativa en velocidad de procesamiento. Se calcula como el cociente entre el tiempo de ejecución de la configuración de referencia (generalmente la más lenta) y el tiempo de la configuración evaluada. Un valor de 2.0 indicaría que la configuración actual es exactamente dos veces más rápida que la referencia. Esta métrica permite evaluar objetivamente los beneficios de optimizaciones específicas.
- **GPU Average Utilization:** Porcentaje medio de utilización de los recursos de la GPU durante el tiempo de ejecución. Este valor, obtenido mediante las herramientas de monitorización de NVIDIA, representa qué fracción de la capacidad computacional total de la GPU se aprovecha efectivamente. Una utilización cercana al 100% sugiere un aprovechamiento óptimo del hardware, mientras que valores bajos podrían indicar cuellos de botella en otras partes del sistema o ineficiencias en la implementación.
- **CPU Average Utilization:** Porcentaje medio de utilización de los núcleos de la CPU durante el tiempo de ejecución. Esta métrica es la media aritmética de todos los núcleos disponibles (entre 1 y 12, dependiendo de la configuración específica del dispositivo Jetson) y lo promedia. Permite identificar si las etapas del sistema que se ejecutan en CPU están equilibradas respecto a las que utilizan aceleradores hardware, y detectar posibles desbalances en la carga de trabajo entre las diferentes unidades de procesamiento.

4.2 Variación de la configuración del sistema

Para evaluar el rendimiento del sistema propuesto, se han realizado pruebas variando la configuración de las etapas del sistema. Estas pruebas se han llevado a cabo utilizando los vídeos de prueba descritos en la sección 4.2.1, con el objetivo de analizar cómo diferentes configuraciones afectan al rendimiento y a las métricas de eficiencia energética.

Para obtener estas métricas, se han utilizado herramientas de medición de potencia y energía, como el comando `tegrastats` de NVIDIA, que proporciona información sobre el consumo de energía y la carga de la CPU y GPU. Esta herramienta permite monitorizar el rendimiento del sistema en tiempo real, proporcionando datos precisos sobre el uso de recursos y el consumo energético.

4.2.1. Cantidad de objetos

La cantidad de objetos presentes en el flujo de vídeo es un factor determinante en la evaluación del rendimiento del sistema, ya que influye directamente en la carga computacional de la etapa de seguimiento y escritura. Para analizar esta influencia de manera sistemática, se han realizado pruebas utilizando tres vídeos de prueba distintos, cada uno diseñado para representar un escenario de carga diferente:

1. **Vídeo 1: Carga baja y constante:** Un vídeo que mantiene una cantidad constante baja de 17 objetos en cada fotograma. Este escenario permite evaluar el rendimiento base del sistema bajo una carga predecible y reducida.
2. **Vídeo 2: Carga media y constante:** Un vídeo que presenta una cantidad constante media de 43 objetos por fotograma. Este escenario permite observar el rendimiento del sistema bajo una carga moderada.
3. **Vídeo 3: Carga alta y constante:** Un vídeo que presenta una cantidad constante media de 84 objetos por fotograma. Este escenario somete al sistema a una carga significativamente mayor, permitiendo identificar posibles cuellos de botella bajo condiciones de alta densidad de objetos.
4. **Vídeo 4: Carga variable:** Un vídeo donde la cantidad de objetos fluctúa dinámicamente a lo largo de su duración, variando entre un mínimo de 0 y un máximo de 180 objetos. Este escenario simula condiciones más realistas y complejas, donde el sistema debe adaptarse a cambios abruptos en la carga de trabajo.

Todos los vídeos de prueba se grabaron con una resolución de 640x640 píxeles y a una tasa de 30 fotogramas por segundo, sirviendo como entrada estándar para el sistema, lo que asegura la consistencia en las condiciones de captura.

Para el resto de la configuración del sistema durante estas pruebas específicas sobre la cantidad de objetos, se ha empleado la segmentación por procesos con memoria compartida. El modelo de detección de objetos utilizado fue YOLO11n, optimizado mediante NVIDIA TensorRT para su ejecución en la GPU.

Se configuró para operar con precisión FP16 y con el perfil de energía del dispositivo Jetson ajustado al modo de máxima potencia (MAXN). Esta configuración se seleccionó con el objetivo de maximizar el rendimiento del sistema y evaluar su capacidad de respuesta y robustez bajo condiciones exigentes impuestas por la variación en la densidad de objetos.

Las pruebas que se presentan a continuación se realizaron siguiendo la metodología de ejecución a máxima capacidad descrita en la sección 4.1, donde los fotogramas se

entregan al sistema consecutivamente, tan pronto como el procesamiento del fotograma anterior ha concluido, utilizando colas intermedias que no descartan fotogramas. En este escenario, se mide el tiempo total que el sistema tarda en procesar la secuencia completa, lo que permite evaluar su rendimiento máximo (throughput) y compararlo con la duración real del vídeo.

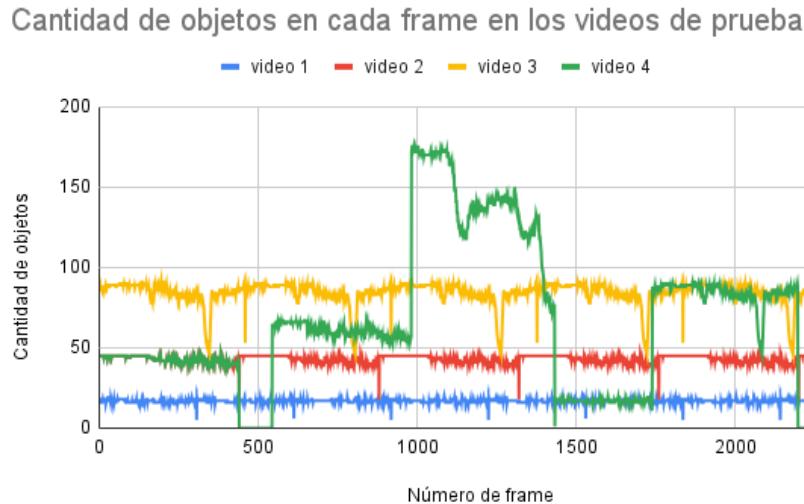


Figura 4.1: Cantidad de objetos en los vídeos de prueba.

La Figura 4.1 ilustra la cantidad de objetos presentes en cada fotograma para los cuatro vídeos de prueba utilizados: carga baja constante (17 objetos), carga media constante (43 objetos), carga alta constante (84 objetos) y carga variable (entre 0 y 180 objetos).

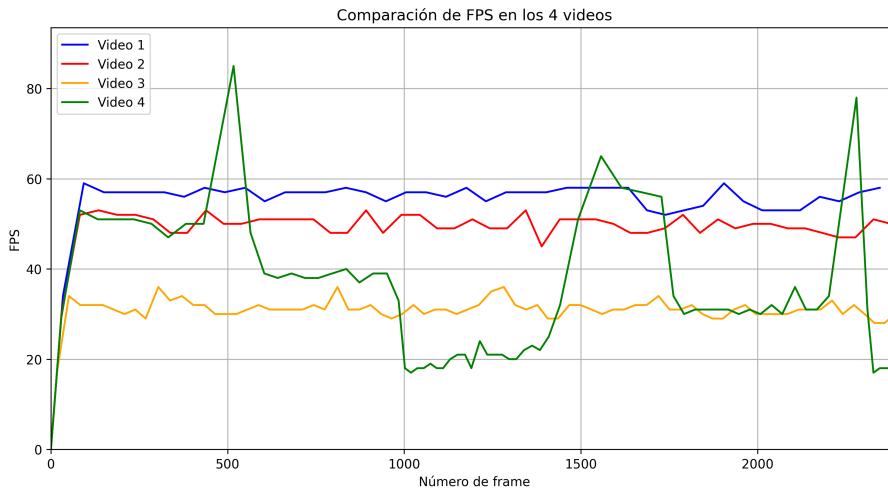


Figura 4.2: FPS por fotograma en función de la cantidad de objetos para los cuatro vídeos de prueba.

Al examinar la Figura 4.2, se evidencia una clara correlación entre el rendimiento del sistema, cuantificado en fotogramas por segundo (FPS), y la densidad de objetos presentes en el flujo de vídeo. En el escenario de carga baja y constante (línea azul), el sistema mantiene un rendimiento consistente y elevado, oscilando entre 50-60 FPS. Para el vídeo de carga media y constante (línea roja), se observa una ligera degradación del rendimiento, que se estabiliza en el rango de 45-50 FPS. En condiciones de carga alta y constante

(línea amarilla), el rendimiento experimenta una reducción más significativa, estableciéndose en un promedio de 30-40 FPS. Esta progresiva disminución del rendimiento se alinea con las expectativas teóricas, dado que el incremento en la cantidad de objetos aumenta proporcionalmente la carga computacional en las etapas de seguimiento y escritura.

El escenario de carga variable (línea verde) revela un comportamiento particularmente informativo: el sistema alcanza picos superiores a 60 FPS durante intervalos con menos de 20 objetos, pero experimenta una pronunciada caída hasta aproximadamente 20 FPS cuando la densidad supera los 100 objetos. Esta marcada fluctuación en el rendimiento confirma la sensibilidad de determinadas etapas del sistema al volumen de objetos procesados simultáneamente, lo que resulta crucial para comprender sus limitaciones operativas en entornos dinámicos.

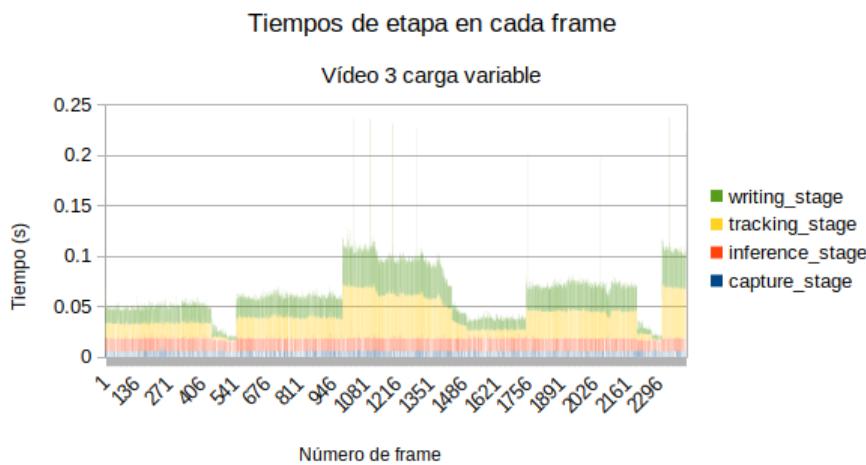


Figura 4.3: Ejecución temporal de las etapas del sistema durante el vídeo de prueba con carga variable.

Para comprender a fondo el impacto de esta variación en la cantidad de objetos, es esencial analizar cómo afecta a cada etapa dentro de la arquitectura segmentada del sistema. La Figura 4.3 presenta el tiempo de ejecución de cada etapa del sistema a lo largo del vídeo de prueba con carga variable.

Se observa que la etapa de captura (línea azul) se mantiene constante, ya que sus operaciones (adquisición de fotogramas) no dependen del contenido de la imagen. De manera similar, la etapa de inferencia (línea roja), ejecutada en la GPU, muestra un tiempo de procesamiento relativamente estable. Aunque podría esperarse una ligera variación, el modelo YOLO procesa la imagen completa y su carga principal no escala linealmente con el número de objetos detectados una vez que la imagen está en la GPU.

Por el contrario, la etapa de seguimiento (línea amarilla), que se ejecuta en la CPU, muestra una clara correlación entre su tiempo de ejecución y la cantidad de objetos. A medida que aumenta el número de objetos (como se ve en la curva de carga variable de la Figura 4.1), el tiempo que tarda la etapa de seguimiento también aumenta significativamente. Esto se debe a que el algoritmo BYTETrack debe gestionar más trayectorias, realizar más comparaciones para la asociación de datos y actualizar más filtros de Kalman. Este comportamiento indica que la etapa de seguimiento puede convertirse en un cuello de botella cuando la densidad de objetos es alta.

Finalmente, la etapa de escritura (línea verde), también dependiente de la CPU, muestra un ligero incremento en su tiempo de ejecución a medida que aumenta el número de

objetos. Esto es esperable, ya que debe procesar y registrar la información de un mayor número de detecciones y trayectorias.

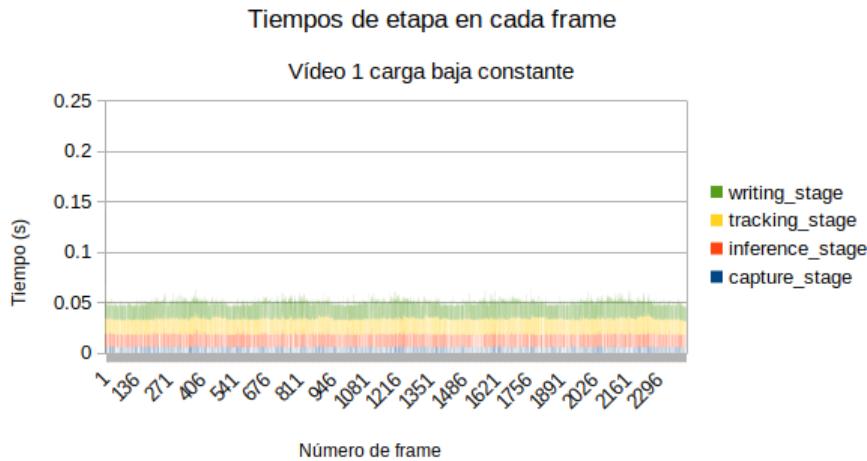


Figura 4.4: Ejecución temporal de las etapas del sistema durante el vídeo de prueba 2 (carga media y constante).

Por otro lado, la Figura 4.4 muestra el tiempo de ejecución de cada etapa del sistema a lo largo del vídeo de prueba 2 (carga media y constante).

En este caso, la etapa de seguimiento (línea amarilla) presenta un tiempo de ejecución bajo y estable, reflejando la menor carga de trabajo al procesar un número constante de objetos. Esto indica que el sistema gestiona eficientemente escenarios de carga baja y media, sin generar cuellos de botella significativos en las etapas de seguimiento o escritura.

De forma similar, la etapa de escritura (línea verde) muestra tiempos de ejecución bajos y uniformes, lo que confirma la capacidad del sistema para mantener un rendimiento constante en condiciones de carga moderada.

La etapa de captura (línea azul) y la etapa de inferencia (línea roja) mantienen un rendimiento estable, similar al observado en el vídeo de carga variable, lo que indica que su rendimiento no se ve afectado por la cantidad de objetos presentes.

Ahora vamos a analizar el rendimiento del sistema con una simulación de entrada en tiempo real. En este caso, el sistema debe procesar cada fotograma en un tiempo máximo de 33.3 ms (30 fps). Si no logra hacerlo, el fotograma se descarta. Esta metodología permite evaluar la capacidad del sistema para operar en tiempo real y medir la tasa de fotogramas perdidos (LFPS).

Cant. Objetos	Frames Procesados	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑	GPU Avg. (%) ↓	CPU Avg. (%) ↓
17	2,397	80,43	29,8	0,04	14,05	1,129	2,12	16,63	22,32
43	2,398	80,46	29,8	0,02	14,66	1,179	2,03	16,61	27,92
84	2,383	80,52	29,6	0,21	15,97	1,285	1,85	16,73	37,61
variable	2,208	80,64	27,38	2,38	14,8	1,192	2	15,05	30,7

Tabla 4.1: Resultados del experimento con distintas cantidades de objetos.

Los datos de la Tabla 4.1 revelan que el sistema logra procesar una cantidad considerable de fotogramas en tiempo real bajo distintas condiciones de carga. Sin embargo, la Figura 4.5 muestra una relación más compleja: cuando la cantidad de objetos excede el umbral de 100, el rendimiento del sistema disminuye notablemente, llegando a generar

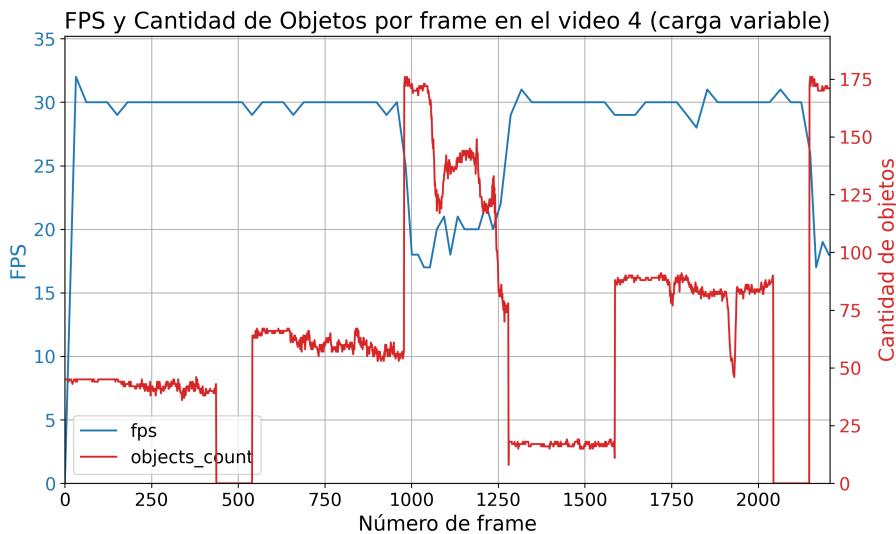


Figura 4.5: FPS y cantidad de objetos en función del tiempo para el vídeo de carga variable.

tasas de fotogramas perdidos (LFPS) de hasta 10 frames por segundo en estos segmentos de alta densidad. Esta observación pone de manifiesto que las métricas promedio pueden resultar engañosas, ya que tienden a ocultar intervalos críticos de bajo rendimiento durante los picos de carga, precisamente cuando el sistema sería más necesario en aplicaciones industriales reales.

El análisis de la Tabla 4.1 también refleja una clara correlación inversa entre la cantidad de objetos y la eficiencia energética del sistema, expresada en fotogramas por vatío (Frames/W). A mayor densidad de objetos, menor es esta eficiencia, lo que indica que el sistema requiere proporcionalmente más energía para procesar escenas complejas. Este comportamiento se alinea con la expectativa de que una mayor carga de trabajo implica un incremento en el consumo de recursos computacionales y energéticos. Esta tendencia se confirma al examinar los niveles de utilización de hardware: el aumento en la cantidad de objetos se corresponde directamente con una mayor utilización media de CPU evidenciando el incremento gradual en la demanda de recursos del sistema conforme crece la complejidad de la escena analizada.

Tiempo de la etapa de tracking en función del número de objetos detectados

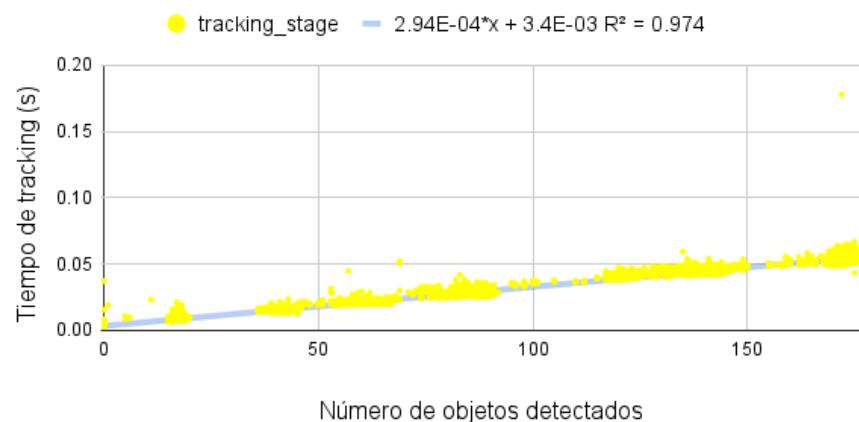


Figura 4.6: Tiempo de ejecución de la etapa de seguimiento en función de la cantidad de objetos.

Con los datos obtenidos, se ha creado una gráfica que muestra el tiempo de ejecución de la etapa de seguimiento en función de la cantidad de objetos presentes en el flujo de vídeo. La Figura 4.6 ilustra la relación empírica que describe esta dependencia. Mediante un análisis de regresión lineal, se ha obtenido la siguiente ecuación:

$$t_{seguimiento} = 2,94 \times 10^{-4} \times n_{objetos} + 3,4 \times 10^{-3} \quad (4.4)$$

donde $t_{seguimiento}$ es el tiempo de ejecución de la etapa de seguimiento en segundos y $n_{objetos}$ es la cantidad de objetos. Esta fórmula presenta un coeficiente de determinación (R^2) de 0.974, lo que indica un ajuste excelente al comportamiento observado.

Para determinar el umbral teórico donde el sistema dejaría de cumplir los requisitos de tiempo real (33.3 ms por fotograma), podemos despejar $n_{objetos}$ de la ecuación anterior:

$$2,94 \times 10^{-4} \times n_{objetos} + 3,4 \times 10^{-3} \leq 33,3 \times 10^{-3} \quad (4.5)$$

$$2,94 \times 10^{-4} \times n_{objetos} \leq 33,3 \times 10^{-3} - 3,4 \times 10^{-3} \quad (4.6)$$

$$2,94 \times 10^{-4} \times n_{objetos} \leq 29,9 \times 10^{-3} \quad (4.7)$$

$$n_{objetos} \leq \frac{29,9 \times 10^{-3}}{2,94 \times 10^{-4}} \quad (4.8)$$

$$n_{objetos} \leq 101,70 \quad (4.9)$$

Este cálculo teórico revela que el sistema alcanzaría su límite de procesamiento en tiempo real aproximadamente con 102 objetos, lo que coincide notablemente con los resultados experimentales mostrados en la Figura 4.5, donde se observaba una degradación significativa del rendimiento alrededor de los 100 objetos.

Tiempo de la etapa de tracking en función del número de objetos detectados

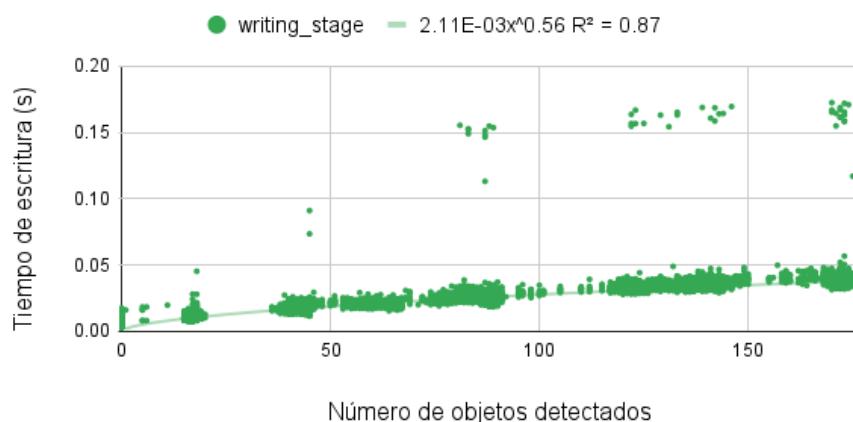


Figura 4.7: Tiempo de ejecución de la etapa de escritura en función de la cantidad de objetos.

Con los datos obtenidos, se ha creado una gráfica que muestra el tiempo de ejecución de la etapa de escritura en función de la cantidad de objetos presentes en el flujo de vídeo. La Figura 4.7 ilustra la relación empírica que describe esta dependencia. Mediante un análisis de regresión polinómica, se ha obtenido la siguiente ecuación:

$$t_{escritura} = 2,11 \times 10^{-3} \times n_{objetos}^{0,56} \quad (4.10)$$

donde $t_{escritura}$ es el tiempo de ejecución de la etapa de escritura en segundos y $n_{objetos}$ es la cantidad de objetos. Esta fórmula presenta un coeficiente de determinación (R^2) de 0.874, lo que indica un ajuste aceptable al comportamiento observado. Para determinar el umbral teórico donde el sistema dejaría de cumplir los requisitos de tiempo real (33.3 ms por fotograma), podemos despejar $n_{objetos}$ de la ecuación anterior:

$$2,11 \times 10^{-3} \times n_{objetos}^{0,56} \leq 33,3 \times 10^{-3} \quad (4.11)$$

$$n_{objetos}^{0,56} \leq \frac{33,3 \times 10^{-3}}{2,11 \times 10^{-3}} \quad (4.12)$$

$$n_{objetos} \leq \left(\frac{33,3}{2,11} \right)^{\frac{1}{0,56}} \quad (4.13)$$

$$n_{objetos} \leq 137,90 \quad (4.14)$$

Este cálculo teórico revela que el sistema alcanzaría su límite de procesamiento en tiempo real para la etapa de escritura aproximadamente con 138 objetos. No obstante, y aunque puedan existir algunos valores atípicos (*outliers*), la etapa de seguimiento (*tracking*) es la que marcará el cuello de botella del sistema frente a la de escritura.

Como conclusiones, se puede afirmar que el rendimiento del sistema es altamente sensible a la cantidad de objetos presentes en el flujo de vídeo. El umbral de aproximadamente 100 objetos constituye un límite crítico, donde la etapa de seguimiento comienza a experimentar un aumento significativo en su tiempo de ejecución, haciendo que el sistema no pueda cumplir con los requisitos de tiempo real.

4.2.2. Tipo de segmentación

Como se ha comentado en la sección 3.5, el sistema se puede segmentar de diferentes maneras. En esta sección se analizará el rendimiento de la solución propuesta variando el tipo de segmentación.

Repasando la sección 3.5, se han implementado cinco tipos de modos de segmentación:

1. **Ejecución secuencial** (subsección 3.5.1): Cada etapa del sistema se ejecuta de forma consecutiva. Este es el enfoque base y no se considera una segmentación propiamente dicha, sino el punto de partida para la comparación.
2. **Segmentación por hilos** (subsección 3.5.2): Cada etapa del sistema se ejecuta en un hilo independiente.
3. **Segmentación por procesos** (subsección 3.5.3): Cada etapa del sistema se ejecuta en un proceso independiente.
4. **Segmentación heterogénea** (subsección 3.5.4): La etapa de inferencia se descarga a los aceleradores de hardware (GPU o DLA).
5. **Segmentación por procesos con memoria compartida** (subsección 3.5.5): Cada etapa del sistema se ejecuta en un proceso independiente y se comunica mediante memoria compartida.

Para evaluar el rendimiento de cada tipo de segmentación, se han realizado pruebas utilizando el vídeo de 84 objetos (carga alta y constante) como entrada estándar. Este vídeo presenta una carga computacional significativa, lo que permite observar las diferencias de rendimiento entre los distintos tipos de segmentación. Para todas las pruebas

se ha utilizado el modelo de detección de objetos YOLO11n, optimizado mediante NVIDIA TensorRT para su ejecución en la GPU. Se configuró para operar con precisión FP16 y con el perfil de energía del dispositivo Jetson ajustado al modo de máxima potencia (MAXN).

Primero se va analizar el rendimiento de cada tipo de segmentación utilizando la metodología de ejecución a máxima capacidad, donde los fotogramas se entregan al sistema consecutivamente, tan pronto como el procesamiento del fotograma anterior ha concluido, utilizando colas intermedias que no descartan fotogramas. En este escenario, se mide el tiempo total que el sistema tarda en procesar la secuencia completa, lo que permite evaluar su rendimiento máximo (throughput) y compararlo con la duración real del vídeo.

Modo de Segmentación	Frames Procesados ↑	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑	Speedup ↑	GPU Avg. ¹↓	CPU Avg. ↓
secuencial	2,400	198,57	12,09	0	9,49	1,884,03	1,27	1	24,62	15,46
hilos	2,400	131,4	18,26	0	12,99	1,703,12	1,41	1,51	10,35	25,53
multiprocesos	2,400	105,16	22,82	0	14,27	1,500,58	1,6	1,89	12,84	33,65
multihardware	2,400	84,36	28,45	0	17,812	1,502,39	1,6	2,35	10	44,25
multiprocesos memoria compartida	2,400	80,27	29,9	0	16,55	1,320,25	1,82	2,47	17,12	40,1

Tabla 4.2: Resultados del experimento con distintos tipos de segmentación a máxima capacidad.

A partir de los datos de la Tabla 4.2, se pueden extraer las siguientes conclusiones sobre el rendimiento de los diferentes enfoques de segmentación:

La ejecución secuencial, como era de esperar, ofrece el rendimiento más bajo, sirviendo como línea base para las comparaciones de speedup. Al no solapar ninguna operación, su capacidad de procesamiento es la más limitada.

La segmentación por hilos es la que ofrece el peor rendimiento después de la secuencial. Esto se debe fundamentalmente al Global Interpreter Lock (GIL) de Python, que impide el paralelismo real entre hilos dentro de un mismo proceso, limitando la capacidad de aprovechar múltiples núcleos de CPU.

La segmentación por procesos mejora esta situación al introducir paralelismo real, ya que cada proceso tiene su propio intérprete Python y, por ende, su propio GIL. Sin embargo, la comunicación estándar entre procesos mediante colas introduce una sobrecarga que impide extraer el máximo rendimiento.

La segmentación heterogénea, que asigna la inferencia a la GPU o DLA, ofrece un rendimiento notable y es capaz de procesar el vídeo en tiempo real. Si bien su velocidad podría ser comparable a la segmentación por procesos con memoria compartida, su consumo energético es superior debido al uso simultáneo de la GPU y, en su caso, la DLA. El rendimiento se ve limitado porque el modelo no se ejecuta en su totalidad en la DLA, lo que resulta en que una porción de la carga de trabajo recaiga sobre la GPU. Esto impide de que las unidades de procesamiento operen de forma completamente independiente, generando competencia por los recursos de la GPU.

Finalmente, la segmentación por procesos con memoria compartida es la que ofrece el mejor rendimiento global. Este enfoque combina el paralelismo real de los procesos con una comunicación entre ellos mucho más eficiente gracias al uso de memoria compartida, lo que minimiza la sobrecarga de la transferencia de datos. Esta combinación la posiciona como la opción más rápida, siendo también capaz de procesar el vídeo en tiempo real (30 fps).

¹En la configuración de segmentación heterogénea, se utilizan la GPU y los dos núcleos DLA; sin embargo, solo se muestra la utilización de la GPU, ya que la herramienta de monitorización de NVIDIA no reporta la utilización media de la DLA.

Ahora se va a analizar el rendimiento de cada tipo de segmentación utilizando la metodología de ejecución en tiempo real, donde el sistema debe procesar cada fotograma en un tiempo máximo de 33.3 ms (30 fps). Si no logra hacerlo, el fotograma se descarta. Esta metodología permite evaluar la capacidad del sistema para operar en tiempo real y medir la tasa de fotogramas perdidos (LFPS).

Modo de Segmentación	Frames Procesados ↑	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑
secuencial	800	80,13	9,98	19,97	9,95	796,97	1
hilos	1,468	80,98	18,13	11,51	13,27	1,106,05	1,33
multiprocesos	1,813	80,57	22,5	7,29	14,35	1,156,09	1,57
multihardware	2,353	80,81	29,12	0,58	17,36	1,402,1	1,68
multiprocesos memoria compartida	2,327	80,63	28,86	0,91	16,33	1,316,14	1,77

Tabla 4.3: Resultados del experimento con distintos tipos de segmentación a 30 fps.

Como se puede observar en la Tabla 4.3, el tipo de segmentación influye significativamente en el rendimiento del sistema. La ejecución secuencial es inviable para tiempo real, presentando la tasa más alta de fotogramas perdidos (LFPS) con 19.87 fps.

La segmentación por hilos, aunque mejora ligeramente, sigue siendo inviable para tiempo real, con una LFPS de 11.51, lo que confirma los resultados de la ejecución a máxima capacidad. Aunque la segmentación por procesos mejora este valor a 7.29 LFPS, tampoco alcanza los requisitos de tiempo real.

Por su parte, la segmentación heterogénea logra operar en tiempo real, pero a costa de un mayor consumo energético. Finalmente, la segmentación por procesos con memoria compartida destaca como la opción más eficiente, logrando el mejor rendimiento en tiempo real (LFPS de 0.91) con el menor consumo energético.

Como conclusión, la segmentación por procesos con memoria compartida es la más adecuada para aplicaciones en tiempo real, ya que combina un rendimiento óptimo con un consumo energético eficiente. Este enfoque permite al sistema operar de manera efectiva bajo condiciones de alta carga computacional, cumpliendo con los requisitos de tiempo real sin comprometer la eficiencia energética. La segmentación heterogénea, no resulta tan eficiente debido a la falta de independencia total entre la GPU y la DLA, lo que limita su capacidad de procesamiento en paralelo pero resulta una aproximación viable en un futuro si NVIDIA mejora la implementación de la DLA para que pueda ejecutar todo el modelo de detección de objetos.

4.2.3. Modelo y talla

En esta sección se analizará el rendimiento de la solución propuesta variando el modelo de detección de objetos y sus diferentes tallas.

Repasando la tabla 3.1, se han empleado tres modelos de detección de objetos: YOLOv5, YOLOv8 y YOLOv11. Cada uno de estos modelos tiene diferentes tallas, que son versiones optimizadas para diferentes capacidades computacionales.

Para la evaluación del rendimiento de cada modelo y sus diferentes tallas, se estableció una configuración base común: los modelos de detección de objetos se optimizaron con NVIDIA TensorRT para su ejecución en la GPU, utilizando precisión FP16 y el perfil de energía MAXN del dispositivo Jetson AGX Xavier. Adicionalmente, se empleó la segmentación por procesos con memoria compartida, identificada en análisis previos como la más eficiente en rendimiento y consumo energético. Sobre esta configuración, las

pruebas se realizaron utilizando como entrada estándar el vídeo de 84 objetos (carga alta y constante), lo que permite someter al sistema a una carga computacional elevada y así comparar objetivamente las distintas variantes de modelos y tallas.

Modelo	Frames Procesados ↑	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑	GPU Avg. (%) ↓	CPU Avg. (%) ↓	T. Prepro. (ms) ↓	T. Infer. (ms) ↓	T. Postpro. (ms) ↓
YOLOv5nu	2,400	84,44	28,42	0	14,847	1,253,41	1,91	42,73	43,73	2,8	13,84	4,11
YOLOv5mu	2,400	84	28,57	0	25,428	2,135,61	1,12	52,45	37,83	2	18,88	3,92
YOLOv8n	2,400	83,74	28,66	0	13,925	1,165,72	2,06	43,12	37,35	2,59	13,46	3,83
YOLOv8s	2,400	84,46	28,42	0	16,404	1,385,11	1,73	49,12	37,5	2,44	17,01	3,92
YOLO11n	2,400	82,28	29,17	0	14,112	1,160,85	2,07	44,4	37,9	2,55	13,94	3,81
YOLO11s	2,400	86,34	27,8	0	15,75	1,359,56	1,77	49,09	36,09	2,45	16,98	3,85
YOLO11m	2,400	84,66	28,35	0	26,125	2,211,51	1,09	55,3	37,61	1,99	19,33	3,8
YOLO11l	2,400	85,56	28,05	0	30,396	2,600,28	0,92	67,09	37,4	2,11	24,51	3,72

Tabla 4.4: Resultados del experimento con distintos modelos y tallas a máxima capacidad con un vídeo de carga alta y constante.

Al analizar los resultados presentados en la Tabla 4.4, se observa una notable similitud en el rendimiento de fotogramas por segundo (FPS) entre todos los modelos y sus respectivas tallas, manteniéndose en un rango aproximado de 28 ± 1 FPS. Esta homogeneidad en la velocidad de procesamiento se explica por la naturaleza del vídeo de prueba utilizado, que presenta una carga alta y constante de objetos. En este escenario, el rendimiento global del sistema se ve limitado predominantemente por la capacidad de las etapas posteriores a la inferencia (como el seguimiento y la escritura) para procesar la gran cantidad de objetos detectados, más que por la complejidad intrínseca o la velocidad de la etapa de inferencia en sí misma. Esto emascara las diferencias potenciales en la velocidad de inferencia pura que podrían observarse bajo condiciones de menor carga de objetos.

No obstante, al considerar la eficiencia energética, medida en fotogramas por vatio (Frames/W), emergen diferencias significativas. Los modelos YOLO11n y YOLOv8n se destacan como las opciones más eficientes, logrando un mayor número de fotogramas procesados por cada vatio de energía consumida. En contraste, el modelo YOLO11l, a pesar de ofrecer una tasa de FPS comparable, presenta un consumo energético superior, lo que resulta en una menor eficiencia. Esta diferencia se atribuye al mayor número de parámetros del modelo YOLO11l, que inherentemente demanda más recursos computacionales y, por ende, energéticos para su ejecución.

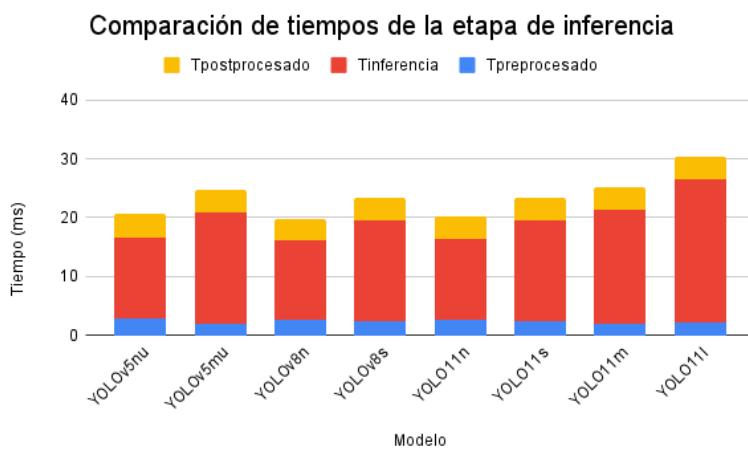


Figura 4.8: Tiempos de ejecución de la etapa de inferencia para los diferentes modelos y tallas.

Un análisis detallado de los tiempos de la etapa de inferencia, presentado en la Figura 4.8, revela que los modelos YOLOv8n y YOLO11n destacan por su menor tiempo

de ejecución. En contraste, YOLOv5mu y YOLO11l exhiben tiempos más elevados, una tendencia que se alinea con el mayor consumo energético reportado en la Tabla 4.4.

Ahora se va a analizar el rendimiento de cada modelo y sus diferentes tallas utilizando la metodología de ejecución en tiempo real, donde el sistema debe procesar cada fotograma en un tiempo máximo de 33.3 ms (30 fps). Si no logra hacerlo, el fotograma se descarta. Esta metodología permite evaluar la capacidad del sistema para operar en tiempo real y medir la tasa de fotogramas perdidos (LFPS).

Modelo	Frames Procesados ↑	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑	mAP _{50–95} ↑
YOLOv5nu	2,303	80,58	28,58	1,2	14,018	1,129,33	2,04	0,76
YOLOv5mu	2,277	80,62	28,24	1,53	24,464	1,971,89	1,15	0,79
YOLOv8n	2,273	80,59	28,2	1,58	13,941	1,123,15	2,02	0,77
YOLOv8s	2,207	80,62	27,38	2,39	16,03	1,292,14	1,71	0,79
YOLO11n	2,297	80,63	28,49	1,28	12,883	1,118,99	2,05	0,76
YOLO11s	2,284	80,62	28,33	1,44	15,823	1,275,37	1,79	0,8
YOLO11m	2,296	80,6	28,49	1,29	26,085	2,102,2	1,09	0,8
YOLO11l	2,247	80,57	27,89	1,9	30,216	2,434,21	0,92	0,8

Tabla 4.5: Resultados del experimento con distintos modelos y tallas a 30 fps.

Los resultados de la Tabla 4.5 muestran una tendencia general similar a la observada en la Tabla 4.4 en términos de rendimiento. Sin embargo, la Tabla 4.5 permite un análisis más detallado de la tasa de fotogramas perdidos (LFPS) para cada combinación de modelo y talla. Se observa que el modelo YOLOv8s presenta la tasa de LFPS más alta, alcanzando 2.39 LFPS. A pesar de esto, todas las variantes de modelos y tallas evaluadas mantienen una tasa de LFPS que podría considerarse aceptable para un sistema de visión artificial operando en tiempo real a 30 fps, ya que el sistema podría continuar funcionando de manera efectiva. El modelo YOLOv8n registra la tasa de LFPS más baja (1.2 LFPS), aunque las diferencias entre los modelos no son drásticas en este aspecto.

Para seleccionar el modelo y la talla más adecuados, es crucial considerar un equilibrio entre el consumo energético y la precisión del modelo. En el contexto de la tarea actual, que consiste en clasificar canicas de diferentes colores y con/sin defectos, la complejidad es relativamente baja. Como resultado, todos los modelos y tallas evaluados alcanzan niveles de precisión (mAP50-95) similares, tal como se evidencia en la última columna de la Tabla 4.5. Dada esta similitud en precisión, para esta aplicación específica, se podría priorizar la elección del modelo y talla que ofrezca el menor consumo energético. No obstante, en aplicaciones donde la precisión de detección sea un factor crítico, la selección debería inclinarse hacia el modelo y talla que demuestre el mayor rendimiento en dicha métrica, incluso si esto implica un mayor consumo energético.

4.2.4. Precisión numérica y acelerador de inferencia

En esta sección se analizará el rendimiento de la solución propuesta variando la precisión numérica (FP32, FP16, INT8) del modelo de detección de objetos y el acelerador de inferencia (CPU, GPU, DLA).

Para la realización de estas pruebas, se ha utilizado el modelo YOLO11n. Para la ejecución en GPU y DLA, el modelo fue optimizado con NVIDIA TensorRT, evaluándose las precisiones FP32, FP16 e INT8. En el caso de la ejecución en CPU, se empleó el modelo YOLO11n sin la optimización de TensorRT, ya que esta no es aplicable a dicho dispositivo. La exportación del modelo de PyTorch a TensorRT con precisión INT8 requiere un conjunto de datos de calibración; para este propósito, se utilizó un subconjunto de 100 imágenes extraídas del conjunto de entrenamiento original, con el fin de calibrar el

modelo y optimizar su rendimiento para la inferencia en INT8. Por último, la precisión de validación de todos los modelos en la Tabla 4.6 es casi idéntica, con un mAP₅₀₋₉₅ de 0.77 para todas las variantes, lo que indica que la precisión del modelo no se ve afectada significativamente por la reducción de la precisión numérica posiblemente debido a la sencillez de la tarea de detección de objetos en este caso específico.

Acelerador	Precisión	Frames Procesados ↑	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑	GPU Avg. (%) ↓	CPU Avg. (%) ↓	mAP ₅₀₋₉₅ ↑
CPU	FP32	2,400	2,569,87	0,93	0	11,694	30,042,76	0,08	0	33,9	0,7757
GPU	FP32(sin exportar a TensorRT)	2,400	183,29	13,09	0	15,731	2,882,77	0,83	23,74	23,74	0,7757
GPU	FP32(exportado TensorRT)	2,400	84,44	28,42	0	19,371	1,635,31	1,47	29,08	37,81	0,7836
GPU	FP16	2,400	83,55	28,73	0	16,01	1,337,34	1,79	16,28	37,79	0,7831
GPU	INT8	2,400	84,4	28,44	0	14,469	1,222,37	1,96	12,86	37,75	0,7792
DLA0	FP16	2,400	86,77	27,66	0	15,597	1,439,73	1,67	6,2	35,93	0,7827
DLA0	INT8	2,400	89,11	26,93	0	14,346	1,278,04	1,88	10,71	37,46	0,7727

Tabla 4.6: Resultados del experimento con distintos modelos y tallas a máxima capacidad con un vídeo de carga alta y constante.

Basándose en los resultados de la Tabla 4.6, se pueden extraer varias conclusiones sobre el impacto de la precisión numérica y el acelerador de inferencia.

En primer lugar, la ejecución del modelo en CPU resulta completamente inviable para aplicaciones en tiempo real. Como se aprecia en la tabla, la tasa de procesamiento es de tan solo 0.93 FPS, lo que demuestra la necesidad de aceleradores hardware para esta tarea.

Al comparar el rendimiento de la GPU y los DLAs, se observa que la DLA, en teoría diseñada para manejar cargas de trabajo específicas de inferencia de manera eficiente (como se repasó en la sección 2.2.3), ofrece un rendimiento similar al de la GPU en este caso. Sin embargo, es crucial señalar que el modelo YOLO11n, tanto en precisión FP16 como en INT8, no se ejecuta completamente en la DLA, ya que algunas de sus operaciones no son compatibles con este dispositivo. Esto implica que la GPU también participa en el procesamiento. Este fenómeno se hace particularmente evidente en la precisión INT8, donde ninguna capa del modelo se ejecuta en la DLA, recayendo toda la carga en la GPU.

```
[TRT] [I] ----- Layers Running on DLA -----
[TRT] [I] [DlaLayer] (ForeignNode[/model_0/conv/Conv.../model.10/m/m.0/attn/qkv/conv/Conv])
[TRT] [I] [DlaLayer] (ForeignNode[/model_0/m/m.0/attn/Split.../model.10/m/m.0/attn/Transpose])
[TRT] [I] [DlaLayer] (ForeignNode[/model_0/m/m.0/attn/Constant_1 output_0 + (Unnamed Layer* 146) [Shuffle] + /model.10/m/m.0/attn/Mul])
[TRT] [I] [DlaLayer] (ForeignNode[/model_0/m/m.0/attn/Split_20.../Unnamed Layer* 149] [Shuffle] + /model.10/m/m.0/attn/Transpose_1])
[TRT] [I] [DlaLayer] (ForeignNode[/model_0/m/m.0/attn/Pe])
[TRT] [I] [DlaLayer] (ForeignNode[/model_0/m/m.0/pe/conv/Conv.../model.23/Concat_2])
[TRT] [I] ----- Layers Running on GPU -----
[TRT] [I] [GpuLayer] SHUFFLE: /model.10/m/m.0/attn/Reshape
[TRT] [I] [GpuLayer] MATRIX MULTIPLY: /model.10/m/m.0/attn/MatMul
[TRT] [I] [GpuLayer] SOFTMAX: /model.10/m/m.0/attn/Softmax
[TRT] [I] [GpuLayer] MATRIX MULTIPLY: /model.10/m/m.0/attn/MatMul_1
[TRT] [I] [GpuLayer] SHUFFLE: /model.10/m/m.0/attn/Reshape_2
[TRT] [I] [GpuLayer] SHUFFLE: /model.23/Reshape
[TRT] [I] [GpuLayer] COPY: /model.23/Reshape copy output
[TRT] [I] [GpuLayer] SHUFFLE: /model.23/Reshape_1
[TRT] [I] [GpuLayer] COPY: /model.23/Reshape_1 copy output
[TRT] [I] [GpuLayer] SHUFFLE: /model.23/Reshape_2
[TRT] [I] [GpuLayer] COPY: /model.23/Reshape_2 copy output
[TRT] [I] [GpuLayer] SHUFFLE: /model.23/dfl/Reshape + /model.23/dfl/Transpose
[TRT] [I] [GpuLayer] SOFTMAX: /model.23/dfl/Softmax
[TRT] [I] [GpuLayer] CONVOLUTION: /model.23/conv/Conv
[TRT] [I] [GpuLayer] CONSTANT: scale constant of /model.23/Sub_1
[TRT] [I] [GpuLayer] CONSTANT: scale constant of /model.23/Sub
[TRT] [I] [GpuLayer] SHUFFLE: /model.23/dfl/Reshape_1
[TRT] [I] [GpuLayer] CONSTANT: /model.23/Constant_9 output_0
[TRT] [I] [GpuLayer] CONSTANT: /model.23/Constant_10 output_0
[TRT] [I] [GpuLayer] POINTEWISE: PwN(scale eltwise of /model.23/Sub, /model.23/Sub)
[TRT] [I] [GpuLayer] ELEMENTWISE: /model.23/Add_1
[TRT] [I] [GpuLayer] POINTEWISE: PwN(scale eltwise of /model.23/Sub_1, /model.23/Sub_1)
[TRT] [I] [GpuLayer] POINTEWISE: PwN(/model.23/Constant_11 output_0 + (Unnamed Layer* 386) [Shuffle], PwN(/model.23/Add_2, /model.23/Div_1))
[TRT] [I] [GpuLayer] COPY: /model.23/Div_1 output_0 copy
[TRT] [I] [GpuLayer] CONSTANT: /model.23/Constant_12 output_0 + (Unnamed Layer* 390) [Shuffle]
[TRT] [I] [GpuLayer] POINTEWISE: PwN(/model.23/Sigmoid)
[TRT] [I] [GpuLayer] ELEMENTWISE: /model.23/Mul_2
[TRT] [I] [GpuLayer] COPY: /model.23/Mul_2 output_0 copy
```

Figura 4.9: Exportación del modelo YOLO11n con TensorRT a FP16 para su ejecución en la DLA.

En la precisión FP16, como se ilustra en la Figura 4.9, la DLA se encarga de 6 de las 34 operaciones/capas del modelo, mientras que la GPU procesa las 28 restantes. En esencia, la DLA no logra descargar completamente a la GPU, lo que limita su capacidad para operar de forma independiente y optimizar el rendimiento general del sistema. Si bien en

modelos más simples la DLA podría asumir una mayor proporción de las operaciones, la complejidad del modelo YOLO11n (y de forma similar para YOLOv5 y YOLOv8) impide que la DLA asuma una carga de trabajo más significativa.

Considerando la ejecución en GPU, la optimización con TensorRT, incluso manteniendo la precisión FP32, produce una notable aceleración en comparación con la ejecución del modelo sin optimizar en GPU. El modelo optimizado para GPU con TensorRT en FP32 alcanza un speedup de $\frac{183,29}{84,44} = 2,17 \times$ frente a la GPU. Esto demuestra que TensorRT mejora drásticamente el rendimiento mediante optimizaciones como la fusión de capas y la eliminación de operaciones redundantes, resultando en una ejecución más eficiente.

En cuanto a las diferentes precisiones numéricas en la GPU (todas optimizadas con TensorRT), los resultados muestran un rendimiento en FPS similar entre FP32, FP16 e INT8.

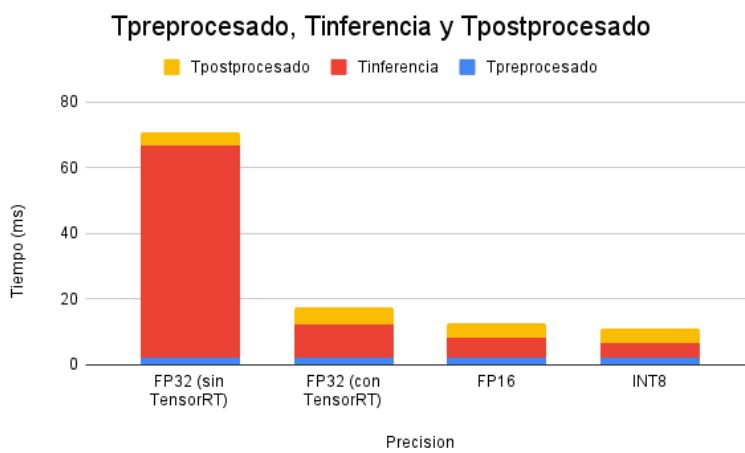


Figura 4.10: Tiempos de ejecución de la etapa de inferencia para las diferentes precisiones en GPU con TensorRT.

Un análisis más detallado de los tiempos de la etapa de inferencia, como se observa en la Figura 4.10, revela una reducción en el tiempo de inferencia al utilizar precisiones más bajas. Aunque el impacto en los FPS totales del sistema es limitado por otros factores del *pipeline* en este escenario de alta carga de objetos, es importante destacar que las precisiones FP16 e INT8 logran un menor consumo energético. Esto se debe a que estas precisiones reducidas requieren menos recursos computacionales, lo que se traduce en una mayor eficiencia energética. En particular, la precisión INT8 se presenta como la opción más eficiente en términos de consumo energético, alcanzando un rendimiento de 1.96 Frames/W.

Acelerador	Precisión	Frames Procesados ↑	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑	GPU Avg. (%) ↓	CPU Avg. (%) ↓	mAP ₅₀₋₉₅ ↑
CPU	FP32	65	81,97	0,79	28,49	12,988	1,064,25	0,06	0	49,37	0,7757
GPU	FP32(sin exportar a TensorRT)	1,006	80,59	12,48	17,3	15,829	1,275,39	0,79	22,56	30,97	0,7757
GPU	FP32(exportado TensorRT)	2,292	80,62	28,43	1,34	19,273	1,553,41	1,48	28,98	37,25	0,7836
GPU	FP16	2,302	80,58	28,57	1,22	15,769	1,270,4	1,81	16,2	36,98	0,7831
GPU	INT8	2,279	80,06	28,47	1,51	14,32	1,153,87	1,98	12,09	36,72	0,7792
DLA0	FP16	2,229	80,56	27,67	2,12	16,616	1,338,17	1,67	6,09	36,03	0,7827
DLA0	INT8	2,137	80,62	26,51	3,26	14,445	1,164,98	1,83	10,94	37,75	0,7727

Tabla 4.7: Resultados del experimento con distintos modelos y tallas a 30 fps con un vídeo de carga alta y constante.

En la Tabla 4.7, se observan las ejecuciones en tiempo real de los modelos con diferentes precisiones y dispositivos. Observando los resultados, como se ha mencionado

anteriormente, la ejecución del modelo en CPU es completamente inviable para tiempo real, con una tasa de 0.79 FPS y 28.49 LFPS.

En cuanto a la GPU, todas las precisiones (FP32, FP16 e INT8 con TensorRT) logran operar en tiempo real, con tasas de FPS de 28.42, 28.73 y 28.44 respectivamente, lo que indica que el sistema puede procesar los fotogramas a la velocidad requerida sin perder ninguno.

La combinación de la GPU con la precisión INT8 es la más eficiente en términos de consumo energético, alcanzando 1.96 Frames/W.

4.2.5. Modo de energía y cores de la CPU

En esta sección, se analiza el impacto de la configuración energética del dispositivo NVIDIA Jetson AGX Xavier en el rendimiento del sistema. Se exploran distintos perfiles de energía predefinidos (10W, 15W, 30W y MAXN), los cuales ajustan tanto el número de núcleos de CPU activos como su frecuencia máxima de operación. El objetivo es evaluar cómo estas configuraciones influyen en la velocidad de procesamiento (FPS), el consumo energético y, por ende, la eficiencia energética global del sistema.

Para la realización de estas pruebas, se ha utilizado el modelo YOLO11n con la precisión FP16 optimizado con TensorRT para su ejecución en la GPU, un video de 84 objetos (carga alta y constante) y la segmentación por procesos con memoria compartida.

Modo de Energía	Núcleos de CPU	Frecuencia (GHz)	Frames Procesados ↑	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑	GPU Avg. (%) ↓	CPU Avg. (%) ↓
10W	2	1,2	2,400	343,03	7	0	5,518	1,903,73	1,26	22,29	23,73
15W	4	1,2	2,400	174,44	13,76	0	6,897	1,199,61	2	12,64	39,22
30W	2	2,11	2,400	179,66	13,36	0	8,621	1,548,38	1,55	10,34	23,18
30W	4	1,8	2,400	119,21	20,13	0	9,535	1,136,27	2,11	17,94	36,89
30W	6	1,41	2,400	135,58	17,7	0	8,398	1,139,58	2,11	38,51	38,81
30W	8	1,2	2,400	163,04	14,72	0	7,95	1,295,48	1,85	16,2	38,35
MAXN	8	2,23	2,400	89,92	26,69	0	15,676	1,409,25	1,7	15,74	37,63

Tabla 4.8: Resultados del experimento con distintos modelos y tallas a máxima capacidad con un vídeo de carga alta y constante.

Observando los resultados de la Tabla 4.8, se aprecia una clara influencia del perfil de energía y del número de núcleos de CPU activos en el rendimiento del sistema. En general, al aumentar el perfil de energía y la cantidad de núcleos habilitados, se observa una mejora en la tasa de fotogramas por segundo (FPS). Sin embargo, el perfil de energía de 30W presenta un comportamiento particular: la configuración óptima se alcanza con 4 núcleos de CPU. Esto se debe a que esta combinación ofrece la frecuencia máxima del procesador más alta (1.8 GHz) en comparación con las configuraciones de 6 y 8 núcleos bajo el mismo perfil (1.41 GHz y 1.2 GHz, respectivamente). Aunque la configuración de 30W con 4 núcleos exhibe el mayor consumo de potencia medio (9.535 W), resulta en el menor consumo de energía total (1136.27 J) debido a la reducción en el tiempo de ejecución en comparación con las configuraciones de 6 y 8 núcleos bajo el mismo perfil. Esto subraya la importancia de considerar tanto el consumo de potencia instantáneo como la duración total de la ejecución al evaluar la eficiencia energética del sistema.

Para evaluar el rendimiento en condiciones de tiempo real, se utilizó el mismo vídeo de 84 objetos (carga alta y constante) y la segmentación por procesos con memoria compartida. Se estableció un límite de 33.3 ms por fotograma (30 FPS) para evaluar la capacidad del sistema para operar en tiempo real y medir la tasa de fotogramas perdidos (LFPS). Es importante señalar que, en los perfiles de energía más bajos, no se logró simular el tiempo real, lo que resultó en tiempos de ejecución superiores a los 80 segundos del vídeo a 30 FPS.

Modo de Energía	Núcleos de CPU	Frecuencia (GHz)	Frames Procesados ↑	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑	GPU Avg. (%) ↓	CPU Avg. (%) ↓
10W	2	1,2	537	118,74	4,52	15,69	5,219	619,43	0,87	11,35	24,44
15W	4	1,2	959	83,1	11,54	17,34	6,806	565,46	1,7	11,5	41,48
30W	2	2,11	1,017	88,48	11,49	15,63	8,515	752,92	1,35	10,98	23,65
30W	4	1,8	1,546	81,76	18,91	10,45	9,488	775,38	1,99	15,37	37,87
30W	6	1,41	1,309	81,51	16,06	13,38	8,21	668,79	1,96	36,97	29,62
30W	8	1,2	1,102	82,27	13,39	15,78	7,855	646,02	1,71	10,4	40,47
MAXN	8	2,23	2,141	80,73	26,52	3,21	15,653	1,263,31	1,69	14,92	37,79

Tabla 4.9: Resultados del experimento con distintos modelos y tallas a 30 fps con un vídeo de carga alta y constante.

La Tabla 4.9 presenta los resultados de la simulación en tiempo real. Como se mencionó anteriormente, los perfiles de energía más bajos no lograron simular el tiempo real, evidenciado por tiempos de ejecución superiores a la duración del vídeo de prueba. De todos los perfiles de energía evaluados, únicamente la configuración de máximas prestaciones (MAXN) con 8 núcleos de CPU activos logró operar a una tasa de 26.69 FPS, lo que indica que el sistema puede procesar los fotogramas a una velocidad cercana a la requerida, aunque con cierta pérdida de fotogramas.

4.2.6. Dispositivos Jetson

En esta sección se analizará el rendimiento de la solución propuesta en diferentes dispositivos Jetson, específicamente en el Jetson Orin Nano, Jetson AGX Xavier y Jetson AGX Orin. Se utilizará el modelo YOLO11n con la precisión FP16 optimizado con TensorRT para su ejecución en la GPU, un video de 84 objetos (carga alta y constante) y la segmentación por procesos con memoria compartida. Repasando lo visto en la sección 2.2.3, los dispositivos Jetson evaluados pertenecen a dos generaciones distintas: la Jetson AGX Xavier, que forma parte de una generación anterior, y los modelos Jetson Orin Nano y Jetson AGX Orin, que pertenecen a la generación más reciente de dispositivos Jetson.

La Jetson AGX Xavier es un dispositivo de la generación anterior que cuenta con 8 procesadores ARM Carmel v8.2 a 2.2GHz, una GPU NVIDIA con 512 núcleos Volta y 64 Tensor Cores, junto con 32GB de memoria LPDDR4x. Además, incluye 2 NVIDIA DLAs v1.

Por otro lado, el Jetson Orin Nano, perteneciente a la generación más reciente, está equipado con 6 procesadores ARM Cortex-A78AE a 1.7GHz, una GPU NVIDIA con 1024 núcleos Ampere y 32 Tensor Cores, y 8GB de memoria LPDDR5. Aunque es el modelo más compacto de la serie Orin, ofrece un rendimiento notable para aplicaciones de IA en el *edge*.

Finalmente, la Jetson AGX Orin, también de la generación más reciente, es el modelo más avanzado de los tres. Cuenta con 12 procesadores ARM Cortex-A78AE a 2.2GHz, una GPU NVIDIA con 2048 núcleos Ampere y 64 Tensor Cores, y 64GB de memoria LPDDR5. Además, incluye 2 NVIDIA DLAs v2, que representan una mejora respecto a los DLA v1 de la Jetson AGX Xavier, proporcionando mayor eficiencia y capacidad para tareas de inferencia en redes neuronales profundas.

Dispositivo	Frames Procesados ↑	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑	GPU Avg. (%) ↓	CPU Avg. (%) ↓	T. Prepro. (ms) ↓	T. Infer. (ms) ↓	T. Postpro. (ms) ↓
Jetson AGX Xavier	2,400	78,06	30,75	0	16,47	1,285,27	1,87	17,72	40,08	2,14	5,77	3,8
Jetson Orin Nano	2,400	54,53	44,01	0	7,568	412,49	5,82	19,97	53,38	1,76	4,44	3,96
Jetson AGX Orin	2,400	42,53	56,43	0	13,029	553,97	4,33	21,32	26,93	1,32	2,51	2,89

Tabla 4.10: Resultados del experimento con distintos dispositivos Jetson a máxima capacidad con un vídeo de carga alta y constante.

Analizando los resultados de la Tabla 4.10, se observa que todos los dispositivos Jetson logran procesar el vídeo de 84 objetos (carga alta y constante) a una tasa de FPS superior a 30, lo que indica que posiblemente pueden operar en tiempo real. Sin embargo, el rendimiento varía significativamente entre los dispositivos. El que mayor rendimiento ofrece es el Jetson AGX Orin, con una tasa de FPS de 56.43, seguido por el Jetson Orin Nano con 44.01 FPS, y finalmente el Jetson AGX Xavier con 30.75 FPS. Esta diferencia en el rendimiento se debe a las especificaciones de hardware de cada dispositivo, como la cantidad de núcleos de CPU, la potencia de la GPU.

De manera sorprendente, el Jetson Orin Nano, a pesar de ser un dispositivo más compacto y de menor potencia que el Jetson AGX Orin, logra un rendimiento notablemente alto, superando al Jetson AGX Xavier y siendo el dispositivo más eficiente en términos de consumo energético, alcanzando 5.82 Frames/W frente a los 4.33 Frames/W del Jetson AGX Orin.

Dispositivo	Frames Procesados ↑	Tiempo Ejec. (s) ↓	FPS ↑	LFPS ↓	Potencia media (W) ↓	Energía (J) ↓	Frames/W ↑	GPU Avg. (%) ↓	CPU Avg. (%) ↓	T. Prepro. (ms) ↓	T. Infer. (ms) ↓	T. Postpro. (ms) ↓
Jetson AGX Xavier	2,383	80,52	29,6	0	15,965	1,285,37	1,85	16,73	37,61	2,09	5,73	3,78
Jetson Orin Nano	2,400	80,33	29,88	0	6,324	507,68	4,73	13,04	34,68	1,66	4,41	3,77
Jetson AGX Orin	2,399	80,28	29,88	0	9,581	768,75	3,12	12,69	14	1,34	2,48	2,85

Tabla 4.11: Resultados del experimento con distintos dispositivos Jetson a 30 fps con un vídeo de carga alta y constante.

Los resultados de la Tabla 4.11 reflejan una tendencia similar a la observada en la Tabla 4.10. Todos los dispositivos evaluados tienen la capacidad de operar en tiempo real. Sin embargo, para determinar cuál es el más adecuado para la solución propuesta, es necesario considerar no solo el rendimiento en FPS, sino también el consumo energético y la eficiencia.

En este contexto, el Jetson Orin Nano destaca como el dispositivo más eficiente, alcanzando una tasa de 4.73 Frames/W en condiciones de tiempo real, superando a los otros dos dispositivos. Además, su precio es significativamente más bajo, siendo de 300€, en comparación con los 2400€ del Jetson AGX Orin. Por lo tanto, para este caso específico, donde se utiliza un modelo de detección de objetos relativamente sencillo y se requiere un rendimiento en tiempo real, el Jetson Orin Nano se presenta como la opción más adecuada. Su principal limitación radica en la memoria, que es de 8GB LPDDR5, en comparación con los 64GB LPDDR5 del Jetson AGX Orin y los 32GB LPDDR4x del Jetson AGX Xavier.

Sin embargo, si la aplicación requiere un mayor rendimiento para modelos más complejos o tareas más exigentes, el Jetson AGX Orin sería la elección ideal debido a su superior capacidad de procesamiento y su habilidad para manejar cargas de trabajo más intensivas.

4.3 Evaluación del seguimiento de objetos

Tras implementar la solución propuesta, se evaluó su capacidad para realizar un seguimiento preciso de los objetos detectados. Este análisis se centró en determinar cómo el sistema mantiene la coherencia en la identificación de los objetos a lo largo del tiempo, un aspecto esencial para aplicaciones como la inspección de calidad y el conteo automatizado, donde la trayectoria y la identidad de cada elemento son cruciales.

Para medir el rendimiento de manera exhaustiva, se recurrió a un conjunto de métricas estándar en el campo del seguimiento de objetos múltiples (MOT), tal como se describió en la subsección 2.3.3. Estas métricas, incluyendo MOTA para la precisión glo-

bal del seguimiento, MOTP para la exactitud en la localización, IDF1 para la consistencia en la asignación de identidades, y HOTA para una visión equilibrada de detección y asociación, ofrecen una perspectiva multidimensional del comportamiento del sistema.

El experimento se llevó a cabo utilizando un vídeo de prueba compuesto por 600 fotogramas. Este vídeo fue cuidadosamente etiquetado de forma manual para generar una referencia precisa (*ground truth*), contra la cual se compararon las salidas del sistema. En esta evaluación, el modelo YOLO11n, optimizado con NVIDIA TensorRT y configurado para operar con precisión FP16, se encargó de la detección inicial de objetos en cada fotograma. Posteriormente, el algoritmo BYTETrack[28] gestionó la tarea de seguimiento, asociando las detecciones a lo largo de la secuencia para construir y mantener las trayectorias de los objetos.

Los resultados obtenidos fueron notablemente positivos y validan la robustez del enfoque implementado. El sistema alcanzó un MOTA del 81.8 %, lo que indica una alta precisión general en el seguimiento, minimizando errores como falsos positivos, detecciones omitidas o cambios incorrectos de identidad.

En términos de localización, el MOTP fue del 80.4 %, reflejando que las cajas delimitadoras predichas por el sistema se alinearon con gran exactitud con las posiciones reales de los objetos, un factor importante para cualquier análisis espacial posterior.

De particular relevancia para la aplicación objetivo, el sistema demostró una excelente consistencia en la asignación de identidades, logrando un IDF1 del 90.2 %. Este alto valor de IDF1 es un fuerte indicador de que el sistema puede mantener la identidad correcta de cada objeto a lo largo del tiempo, incluso en presencia de occlusiones o interacciones complejas, lo cual es fundamental para un seguimiento fiable a nivel individual.

Finalmente, la métrica HOTA, que ofrece una evaluación más holística, alcanzó un valor del 71.7 %, confirmando un equilibrio sólido y competente entre la calidad de la detección inicial y la efectividad de la asociación de trayectorias. Colectivamente, estos resultados cuantitativos no solo demuestran la eficacia del sistema para el seguimiento de objetos múltiples, sino que también justifican las elecciones de diseño y la combinación de tecnologías empleadas, subrayando su preparación para aplicaciones que requieren un seguimiento robusto y preciso en entornos dinámicos.

CAPÍTULO 5

Prueba de concepto

Aquí se explicará la implementación de la solución propuesta en el entorno de producción con la cinta transportadora.

5.1 Construcción del entorno

Tras la fase de desarrollo y análisis de la solución propuesta, que se llevó a cabo utilizando vídeos pregrabados en un entorno de laboratorio, se procedió a la implementación de una prueba de concepto. Esta prueba simula un entorno de producción real mediante la construcción de una cinta transportadora sencilla. El principal objetivo de esta etapa es validar la viabilidad y el rendimiento de la solución en un escenario práctico, evaluando su capacidad para detectar y clasificar objetos (canicas) en movimiento sobre dicha cinta. Para la construcción de la cinta transportadora, se tomaron como referencia los planos y la lista de materiales detallados en un proyecto de acceso público [17].

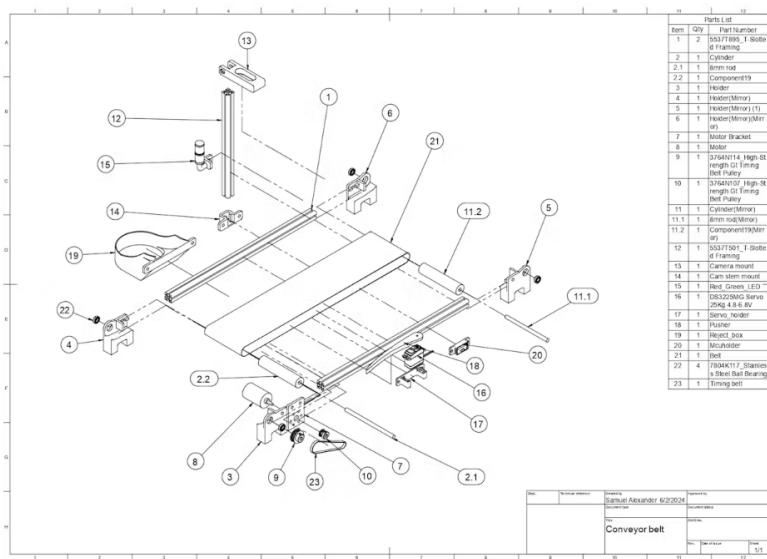


Figura 5.1: Planos de la cinta transportadora. Extraído de [17].

Los materiales empleados para la construcción de la cinta transportadora son los siguientes:

- Un motor de corriente continua (DC) de 12V para el accionamiento de la cinta.

- Filamento PLA (ácido poliláctico) para la impresión 3D de las piezas estructurales de la cinta.
- Cuatro rodamientos de tipo 608ZZ para facilitar el movimiento suave de la cinta.
- Un servomotor de 9g, encargado de accionar el mecanismo de rechazo de canicas defectuosas.
- Una banda transportadora de 1.5 cm de ancho.
- Una placa microcontroladora Raspberry Pi Pico WH para el control del servomotor.
- Una cámara para la captura de imágenes de los objetos en la cinta.
- Cableado diverso y conectores para las interconexiones eléctricas.
- Una fuente de alimentación para suministrar energía a los componentes.

El montaje de la cinta transportadora se realizó siguiendo los planos ilustrados en la Figura 5.1. La Raspberry Pi Pico WH se programó para controlar el servomotor que desvía las canicas identificadas como defectuosas. Para la programación de este microcontrolador se utilizó MicroPython[22]. La cámara se ubicó estratégicamente en la parte superior de la cinta para obtener una vista clara de las canicas durante su tránsito. El motor DC, alimentado por una fuente de 12V, es el responsable del movimiento continuo de la banda transportadora.

El sistema completo se configuró de la siguiente manera: El procesamiento principal, incluyendo la ejecución del modelo de detección de objetos, se realiza en un dispositivo NVIDIA Jetson AGX Xavier. Este dispositivo es el encargado de analizar las imágenes capturadas por la cámara. Para la comunicación entre el Jetson AGX Xavier y la Raspberry Pi Pico WH, se establece una conexión TCP/IP. Al iniciar el sistema, el Jetson AGX Xavier actúa como servidor, abriendo un socket en el puerto 5000 y esperando la conexión de la Raspberry Pi Pico WH (cliente). Una vez que la Raspberry Pi Pico WH se conecta, envía un mensaje de inicio al Jetson. A partir de este momento, el Jetson comienza a procesar el flujo de vídeo de la cámara. Cuando el modelo de detección en el Jetson identifica una canica defectuosa, envía un mensaje específico a la Raspberry Pi Pico WH. Al recibir esta señal, la Raspberry Pi Pico WH activa el servomotor durante un breve instante. Este accionamiento mueve un mecanismo que desvía la canica defectuosa fuera de la trayectoria principal de la cinta transportadora.

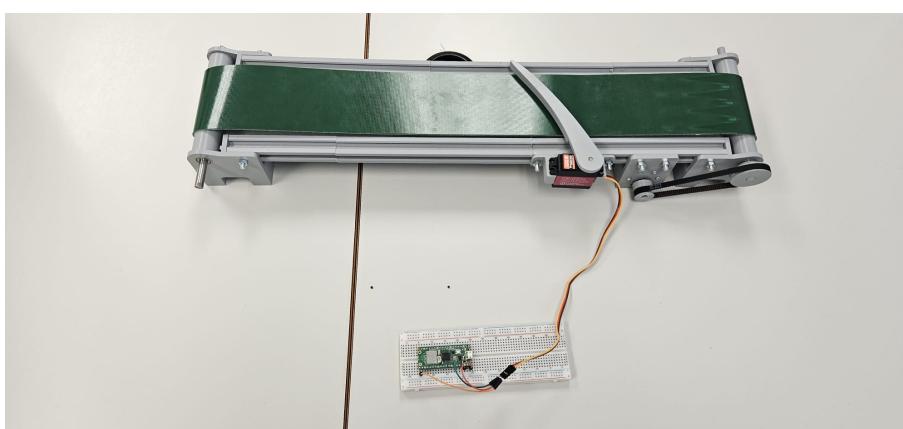


Figura 5.2: Cinta transportadora construida para la prueba de concepto.

En la Figura 5.2 se muestra la cinta transportadora construida para la prueba de concepto.

CAPÍTULO 6

Conclusiones

En este capítulo se presentan las conclusiones obtenidas a partir del desarrollo y la implementación de la solución propuesta para la detección y clasificación de objetos en movimiento sobre una cinta transportadora. Se reflexiona sobre los logros alcanzados, las lecciones aprendidas durante el proceso y las posibles direcciones futuras para mejorar y ampliar el sistema.

6.1 Objetivos alcanzados y dificultades

A lo largo de este Trabajo de Fin de Grado, se han abordado diversos desafíos y se han alcanzado los objetivos propuestos, aunque no sin encontrar ciertas dificultades inherentes al desarrollo de un sistema complejo de visión artificial. A continuación, se detallan los principales logros y los obstáculos superados.

Los objetivos fundamentales del proyecto se han alcanzado con éxito. Se llevó a cabo un estudio exhaustivo del estado del arte, abarcando tanto las CNNs, con experimentación en diversos modelos de detección, como los aceleradores hardware, con especial atención a las GPUs y la plataforma NVIDIA Jetson. Durante esta fase, se identificaron y superaron desafíos significativos, principalmente relacionados con la gestión de la compatibilidad entre las múltiples versiones de frameworks de aprendizaje profundo y el software de NVIDIA. La arquitectura ARM64 de los dispositivos Jetson también presentó particularidades que complejizaron la instalación y configuración de dependencias críticas, como ciertas bibliotecas de Python y componentes específicos del ecosistema de NVIDIA. A pesar de estos obstáculos técnicos, se logró una configuración estable y funcional del entorno de desarrollo y experimentación.

Se generó un conjunto de datos y vídeos de entrenamiento, fundamental para optimizar la precisión del modelo de detección. Estos recursos, obtenidos grabando canicas en movimiento sobre la cinta transportadora, proporcionaron un entorno de prueba realista. El principal desafío fue seleccionar un objeto de prueba fácilmente identificable y con variabilidad controlada en forma y color. Aunque se hubiera preferido un objeto industrial más complejo, no se dispuso de medios para su adquisición.

Se entrenaron múltiples modelos de detección de las familias YOLOv5, YOLOv8 y YOLO11, en diversas precisiones y tamaños. Esto permitió evaluar su evolución y rendimiento en precisión y velocidad. La principal dificultad fue el ajuste fino de hiperparámetros para optimizar el rendimiento, proceso que requirió numerosas iteraciones y pruebas.

Se implementó un sistema de seguimiento que integra los modelos de detección entrenados con el algoritmo BYTETrack. Esta combinación permitió mantener la identidad

de los objetos a lo largo del tiempo, mejorando significativamente la precisión y robustez del sistema. El principal obstáculo fue la integración del algoritmo de seguimiento con la salida de detección, debido a incompatibilidades entre versiones de frameworks y dependencias. No obstante, se logró una integración exitosa y eficaz en la práctica.

Para analizar los cuellos de botella, se exploraron diversas estrategias de segmentación para identificar componentes críticos y optimizar su rendimiento. La segmentación por procesos con memoria compartida fue la más efectiva, facilitando la comunicación eficiente entre detección y seguimiento, y mejorando notablemente el rendimiento global. La mayor complejidad residió en la programación de la comunicación interproceso (transmisión de datos y sincronización), que exigió un conocimiento profundo de programación concurrente y gestión de memoria compartida.

Para cuantificar las métricas de rendimiento bajo diferentes configuraciones de hardware, se realizaron múltiples experimentos. Estos incluyeron la evaluación de distintos modelos y tamaños, operando a diversas tasas de fotogramas por segundo (FPS) y utilizando diferentes dispositivos Jetson. Estos experimentos permitieron evaluar el rendimiento en condiciones realistas y ajustar la configuración para un equilibrio óptimo entre precisión y velocidad. La principal dificultad fue la ejecución de numerosas pruebas y ajustes para identificar la configuración óptima, lo que demandó una considerable inversión de tiempo y recursos.

6.2 Aprendizaje

Durante el desarrollo de este Trabajo de Fin de Grado, se adquirieron conocimientos y habilidades valiosas en diversas áreas. En primer lugar, se profundizó en el ciclo de vida completo de las redes neuronales, desde la creación y anotación de conjuntos de datos, pasando por el entrenamiento y ajuste de hiperparámetros, hasta la optimización de los modelos para una inferencia eficiente. Este proceso implicó un aprendizaje significativo sobre la arquitectura interna de las redes, las métricas de evaluación y las técnicas para mejorar su rendimiento.

Paralelamente, se exploró en detalle el uso de aceleradores hardware, con un enfoque particular en las GPUs y la plataforma NVIDIA Jetson. Esta exploración permitió comprender cómo estas tecnologías pueden mejorar drásticamente el rendimiento de los sistemas de visión artificial en tiempo real. La experiencia práctica obtenida en la configuración y optimización del entorno de desarrollo, así como en la resolución de problemas complejos relacionados con la compatibilidad de software y las particularidades de la arquitectura ARM64, constituyó un aspecto crucial del aprendizaje.

Asimismo, se desarrollaron habilidades prácticas en la implementación de sistemas de seguimiento de objetos. La integración de los modelos de detección entrenados con algoritmos avanzados como BYTETrack supuso un aprendizaje profundo sobre la gestión de identidades y la reconstrucción de trayectorias de objetos en movimiento, aspectos esenciales para aplicaciones industriales y comerciales.

Además, se adquirieron conocimientos sólidos sobre técnicas de programación concurrente, específicamente la segmentación por procesos y la comunicación interproceso mediante memoria compartida. Esta área permitió optimizar significativamente el rendimiento del sistema al identificar y abordar cuellos de botella críticos en el flujo de datos.

Finalmente, la planificación y ejecución de una batería exhaustiva de experimentos para cuantificar métricas de rendimiento bajo diversas configuraciones de hardware y software proporcionó una comprensión práctica y metodológica sobre cómo evaluar, comparar y mejorar el rendimiento en sistemas complejos de visión artificial.

6.3 Relación con los estudios cursados

Este Trabajo de Fin de Grado se relaciona estrechamente con los estudios cursados en el Grado en Ingeniería Informática, especialmente en las siguientes asignaturas han sido las que han proporcionado la base teórica y práctica necesaria para llevar a cabo este proyecto:

- **SDL (Sistemas basados en Deep Learning para la Industria):** La asignatura se encuentra en el segundo cuatrimestre del tercer curso del grado en Ingeniería Informática en la mención de Ingeniería de Computadores. La asignatura tiene como objetivos «Conocer, configurar y utilizar los sistemas actuales específicos para inteligencia artificial (en concreto aprendizaje profundo) que existen para la industria. Especial énfasis en sistemas reales con capacidad de procesamiento de imágenes (clasificación/detección de objetos)» y «Conocer y profundizar en sistemas basados en GPU para el procesamiento de modelos de redes neuronales». Esta asignatura es la base teórica y práctica de este Trabajo de Fin de Grado, ya que se ha utilizado el conocimiento adquirido en ella para implementar la solución propuesta, incluyendo la selección y entrenamiento de modelos de detección de objetos, así como la optimización para su ejecución en dispositivos Jetson.
- **CPA (Computación Paralela):** La asignatura se encuentra en el primer cuatrimestre del tercer curso del grado en Ingeniería Informática. La asignatura tiene como objetivo «Conocer la computación paralela y los modelos de programación paralela a través de los modelos más extendidos: memoria compartida y memoria distribuida». Los conocimientos adquiridos en esta asignatura han sido de especial relevancia para la implementación de técnicas de paralelización para distribuir la carga computacional del procesado de vídeos entre hilos y/o procesos para reducir el tiempo de procesado.

6.4 Trabajo futuro

Como trabajo futuro para mejorar la solución propuesta, se plantean varias líneas de investigación y desarrollo que podrían ampliar las capacidades del sistema y optimizar su rendimiento.

Una primera línea de mejora consistiría en **evaluar el sistema con objetos de mayor complejidad**. Aunque la prueba de concepto se realizó con canicas, un objeto relativamente simple, sería beneficioso probar el sistema con una gama más amplia de objetos que presenten mayor variabilidad en forma, tamaño, textura y color. Esto permitiría validar la robustez y la capacidad de generalización del sistema en escenarios más desafiantes, acercándose a las condiciones reales de un entorno industrial. La utilización de objetos industriales auténticos, con sus irregularidades y posibles defectos específicos, proporcionaría una evaluación más fidedigna del rendimiento del sistema en aplicaciones prácticas.

Otra dirección prometedora es la **incorporación de múltiples cámaras** para obtener una visión más completa de la escena. Un sistema multicámara permitiría capturar imágenes desde diferentes ángulos y perspectivas, lo que podría mejorar significativamente la detección y el seguimiento de objetos, especialmente en situaciones de oclusión parcial o cuando los objetos se mueven rápidamente y cambian de orientación. La fusión de información de múltiples vistas podría conducir a una reconstrucción 3D más precisa de los objetos y sus trayectorias, enriqueciendo el análisis y permitiendo una inspección más detallada.

Además de la visión, se podría **integrar información de otros tipos de sensores** para complementar los datos visuales. Sensores de proximidad, sensores de peso, cámaras térmicas o incluso sensores hiperespectrales podrían proporcionar información adicional valiosa. Por ejemplo, los sensores de peso podrían ayudar a verificar la cantidad de producto, mientras que las cámaras térmicas podrían detectar anomalías de temperatura no visibles. Esta fusión multisensorial podría mejorar la capacidad del sistema para identificar defectos sutiles o características que no son fácilmente discernibles solo con cámaras RGB.

Para completar el ciclo de inspección, se podría **desarrollar un mecanismo de rechazo de objetos defectuosos más sofisticado y automatizado**. Esto podría implicar el diseño e integración de actuadores neumáticos, brazos robóticos pequeños o desviadores mecánicos controlados por el sistema. Al detectar un objeto no conforme, el sistema enviaría una señal para activar el mecanismo de rechazo, retirando el producto defectuoso de la línea de producción de manera eficiente y sin intervención manual, lo que mejoraría la productividad y reduciría errores.

Finalmente, para escalar la capacidad de procesamiento y manejar flujos de vídeo de mayor resolución o múltiples líneas de producción, se podría explorar la **implementación de un sistema distribuido utilizando varios dispositivos Jetson**. Cada dispositivo podría encargarse de una porción del flujo de vídeo o de una tarea específica (p. ej., un dispositivo para detección y otro para seguimiento avanzado). Esta arquitectura distribuida permitiría un procesamiento paralelo más efectivo, mejorando la velocidad general del sistema y su capacidad para manejar cargas de trabajo más intensivas, haciéndolo adecuado para entornos industriales de mayor escala.

Bibliografía

- [1] Anders S. G. Andrae and Tomas Edler. On global electricity usage of communication technology: Trends to 2030. *Challenges*, 6(1):117–157, 2015.
- [2] Google Cloud. Tensor processing units (tpus). <https://cloud.google.com/tpu>, 2025. Accedido el 21 de mayo de 2025.
- [3] Computer Science Wiki. File:maxpoolsample2.png. <https://computersciencewiki.org/index.php/File:MaxpoolSample2.png>, 2018. Accedido el 20 de mayo de 2025.
- [4] T. Conte. IEEE rebooting computing initiative & international roadmap of devices and systems. In *Proc. IEEE Rebooting Computer Architecture 2030 Workshop*.
- [5] CVAT.ai Corporation. Computer Vision Annotation Tool (CVAT), November 2023.
- [6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.
- [7] Robert H. Dennard, Fritz H. Gaenslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bas-sous, and Andre R. Leblanc. Design of ion-implanted mosfets with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, SC-9(5):256–268, October 1974.
- [8] ENCCS. The gpu hardware and software ecosystem. <https://enccs.github.io/gpu-programming/2-gpu-ecosystem/>, 2025. Parte del curso "GPU programming: why, when and how?".
- [9] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierar-chies for accurate object detection and semantic segmentation, 2014.
- [10] Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, 4th edition, 2022.
- [11] Glenn Jocher. Ultralytics yolov5, 2020.
- [12] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics yolov8, 2023.
- [13] Glenn Jocher and Jing Qiu. Ultralytics yolo11, 2024.
- [14] Glenn Jocher, Jing Qiu, and Ayush Chaurasia. Ultralytics YOLO, January 2023.
- [15] Swapna Kategaru. Convolution neural network in deep learning. <https://developersbreach.com/convolution-neural-network-deep-learning/>, 2025. Ac-cedido el 26 de abril de 2025.

- [16] Salman Khan, Hossein Rahmani, Syed Afaq Ali Shah, and Mohammed Bennamoun. *A Guide to Convolutional Neural Networks for Computer Vision*. Synthesis Lectures on Computer Vision. Springer Cham, 1 edition, 2018.
- [17] Kinetica. Counting for inspection and quality control with tensorrt. <https://www.hackster.io/kinetika/counting-for-inspection-and-quality-control-with-tensorrt-550b91>, 2024. Accedido el 26 de mayo de 2025.
- [18] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.
- [19] Tushar Kumar. R-cnn explained. <https://youtu.be/5DvljLV4S1E?si=oeQUo9Cmv4NInTns>, 2024. Video. Accedido: 14 de abril de 2025.
- [20] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015.
- [21] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. *SSD: Single Shot MultiBox Detector*, page 21–37. Springer International Publishing, 2016.
- [22] MicroPython. Micropython - python for microcontrollers. <https://micropython.org/>, 2025. Accedido el 2 de junio de 2025.
- [23] Karthik Mittal. Object detection with histogram of oriented gradients (hog). <https://iq.opengenus.org/object-detection-with-histogram-of-oriented-gradients-hog/>, 2020. Accedido el 20 de mayo de 2025.
- [24] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.
- [25] NVIDIA Corporation. Nvidia deep learning accelerator (nvdla). <https://developer.nvidia.com/deep-learning-accelerator>, 2024. Accedido el 21 de mayo de 2025.
- [26] NVIDIA Corporation. Jetson modules, support, ecosystem, and lineup. <https://developer.nvidia.com/embedded/jetson-modules>, 2025. Accedido el 24 de abril de 2025.
- [27] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.
- [28] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. Bytetrack: Multi-object tracking by associating every detection box, 2022.

APÉNDICE A

Objetivos de desarrollo sostenible

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

Justificación de los Objetivos de Desarrollo Sostenible

ODS-8. Trabajo decente y crecimiento económico: Este proyecto contribuye directamente al crecimiento económico sostenible mediante la automatización inteligente de procesos de control de calidad. La implementación de sistemas de detección de defectos basados en IA permite reducir costes operativos, minimizar desperdicios y optimizar la cadena de producción, lo que se traduce en mayor productividad y competitividad empresarial. Se alinea específicamente con la meta 8.2 de la ONU: «Lograr niveles más elevados de productividad económica mediante la diversificación, la modernización tecnológica y la innovación», al incorporar tecnologías avanzadas de procesamiento de imágenes y ML en entornos industriales tradicionales.

ODS-9. Industria, innovación e infraestructura: El desarrollo de sistemas inteligentes para la detección de defectos representa una clara apuesta por la innovación industrial. Este proyecto no solo implementa tecnologías emergentes como la IA en procesos productivos, sino que además optimiza su rendimiento mediante el uso eficiente de hardware especializado como GPUs, algoritmos de seguimiento multi-objeto y técnicas de pa-

ralelización. Esto responde directamente a la meta 9.4 de la ONU: «Modernizar la infraestructura y reconvertir las industrias para que sean sostenibles, utilizando los recursos con mayor eficacia y promoviendo la adopción de tecnologías y procesos industriales limpios y ambientalmente racionales», al permitir mejoras significativas en eficiencia energética y uso de recursos mediante sistemas de inspección automatizados.

ODS-12. Producción y consumo responsables: La implementación de sistemas de detección temprana de defectos contribuye sustancialmente a la producción responsable mediante: 1) la reducción del descarte de productos y materias primas al identificar problemas en etapas iniciales del proceso productivo, 2) la optimización del consumo energético al evitar el procesamiento completo de productos defectuosos, y 3) la mejora de la calidad final que aumenta la vida útil de los productos. Estas aportaciones se vinculan directamente con la meta 12.5 de la ONU: «De aquí a 2030, reducir considerablemente la generación de desechos mediante actividades de prevención, reducción, reciclado y reutilización», ya que el sistema desarrollado actúa preventivamente evitando la generación de residuos industriales y facilitando la reutilización de materiales recuperados.

APÉNDICE B

Código fuente

En este capítulo se presenta el código fuente más relevante de la solución propuesta eliminando las partes que no aportan valor al lector. El código completo se encuentra disponible en el repositorio de GitHub del proyecto¹

El primero de los archivos que se muestra a continuación es el objeto DetectionTrackingPipeline, que implementa las funciones de las cuatro fases del pipeline de detección y seguimiento de objetos. Este objeto se encarga de inicializar el modelo de detección, el algoritmo de seguimiento, la cámara y el socket para la comunicación con la Raspberry Pi Pico WH. Además, implementa las funciones para procesar los fotogramas de vídeo, detectar los objetos, realizar el seguimiento y pintarlos en el fotograma.

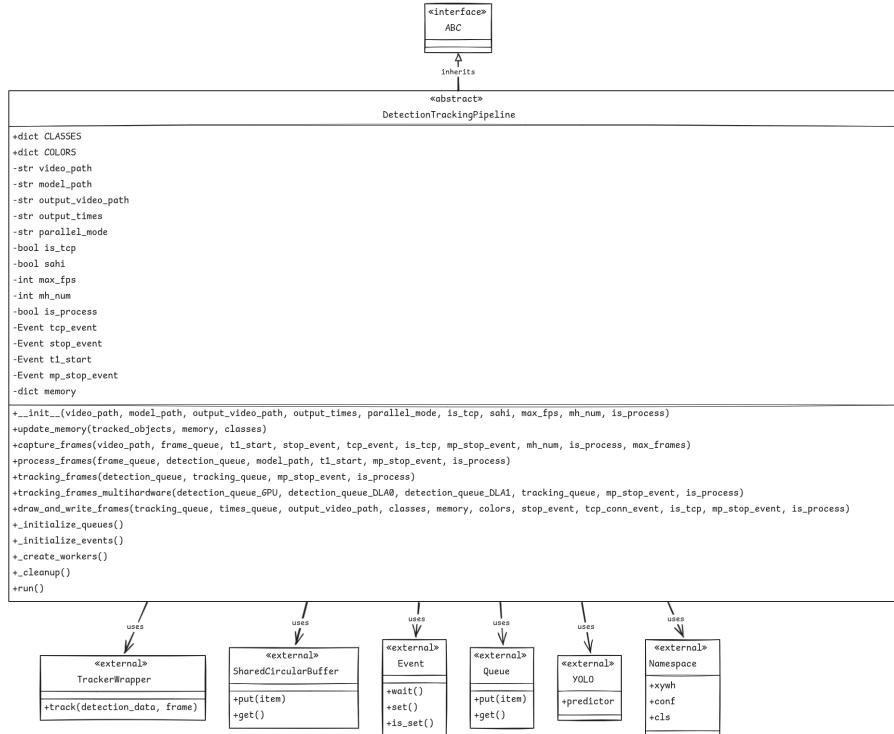


Figura B.1: Diagrama de clases del objeto DetectionTrackingPipeline.

La figura B.1 muestra el diagrama de clases del objeto DetectionTrackingPipeline. Durante la ejecución del programa, las funciones *capture_frames*, *process_frames*, *tracking_frames* y *draw_and_write_frames* se ejecutarán en paralelo, cada una en un hilo o proceso diferente.

¹<https://github.com/AbelHaro/TFG>

rente, según la configuración elegida. El objeto DetectionTrackingPipeline se encarga de gestionar la comunicación entre estos hilos o procesos, asegurando que los datos se transmitan correctamente entre ellos y que el sistema funcione de manera eficiente.

Listing B.1: detection_tracking_pipeline.py

```

1  from abc import ABC, abstractmethod
2  import cv2
3  import os
4  from argparse import Namespace
5  from classes.tracker_wrapper import TrackerWrapper
6  from lib.tcp import handle_send, tcp_server
7  import logging
8  from typing import Union, Optional
9  import torch.multiprocessing as mp
10 from classes.shared_circular_buffer import SharedCircularBuffer
11 import time
12
13
14 class DetectionTrackingPipeline(ABC):
15     """Abstract base class for object detection and tracking pipelines.
16
17     This class defines the structure and common functionality for various
18         pipelines
19     that integrate object detection and their subsequent tracking through
20         video
21     sequences. It allows the implementation of different parallelization
22         strategies
23     and hardware management.
24     """
25
26     CLASSES = {
27         0: "negra",
28         1: "blanca",
29         2: "verde",
30         3: "azul",
31         4: "negra-d",
32         5: "blanca-d",
33         6: "verde-d",
34         7: "azul-d",
35     }
36
37     COLORS = {
38         "negra": (0, 0, 255),
39         "blanca": (0, 255, 0),
40         "verde": (255, 0, 0),
41         "azul": (255, 255, 0),
42         "negra-d": (0, 165, 255),
43         "blanca-d": (255, 165, 0),
44         "verde-d": (255, 105, 180),
45         "azul-d": (255, 0, 255),
46     }
47
48     def update_memory(self, tracked_objects, memory, classes) -> None:
49         """Updates tracking memory with detected and tracked objects.
50
51         Maintains a record of objects, their state (defective or not),
52         and their visibility throughout frames. An object is considered
53         permanently defective if detected as defective during a
54         consecutive number of frames defined by 'PERMANENT_DEFECT_THRESHOLD'
55         .
56
57         Objects that have not been seen for 'FRAME_AGE' frames are removed
58         from memory.
59
60         Args:
61             tracked_objects (List[dict]): A list of dictionaries containing
62                 information about tracked objects, including their bounding
63                 boxes and states.
64             memory (SharedCircularBuffer): A shared circular buffer used
65                 to store tracked objects and their history.
66             classes (List[str]): A list of class names corresponding to the
67                 objects in the tracked_objects list.
68
69         Returns:
70             None
71     """
72
73     def __init__(self, frame_rate, frame_size, frame_age, permanent_defect_threshold):
74         self.frame_rate = frame_rate
75         self.frame_size = frame_size
76         self.frame_age = frame_age
77         self.permanent_defect_threshold = permanent_defect_threshold
78
79         self._start_time = time.time()
80         self._frame_count = 0
81
82         self._memory = SharedCircularBuffer(
83             max_size=frame_size,
84             max_items=frame_age * frame_size
85         )
86
87         self._trackers = {}
88
89         self._classes = classes
90
91         self._logger = logging.getLogger(__name__)
92
93         self._logger.info("DetectionTrackingPipeline initialized")
94
95     def _process_frame(self, frame):
96         """Processes a single frame and updates the tracking pipeline.
97
98         Args:
99             frame (np.ndarray): The current frame to process.
100
101        Returns:
102            None
103        """
104
105         # Implement frame processing logic here
106
107         tracked_objects = self._detect_and_track(frame)
108
109         self._update_memory(tracked_objects, self._memory, self._classes)
110
111         return tracked_objects
112
113     def _detect_and_track(self, frame):
114         """Performs object detection and tracking on the current frame.
115
116         Args:
117             frame (np.ndarray): The current frame to process.
118
119         Returns:
120             List[dict]: A list of dictionaries containing information about
121                 tracked objects, including their bounding boxes and states.
122         """
123
124         # Implement detection and tracking logic here
125
126         tracked_objects = []
127
128         for class_id, class_name in enumerate(self._classes):
129             tracked_objects.append({
130                 "id": class_id,
131                 "bbox": [0, 0, 0, 0],
132                 "state": "normal"
133             })
134
135         return tracked_objects
136
137     def _update_memory(self, tracked_objects, memory, classes):
138         """Updates tracking memory with detected and tracked objects.
139
140         Args:
141             tracked_objects (List[dict]): A list of dictionaries containing
142                 information about tracked objects, including their bounding
143                 boxes and states.
144             memory (SharedCircularBuffer): A shared circular buffer used
145                 to store tracked objects and their history.
146             classes (List[str]): A list of class names corresponding to the
147                 objects in the tracked_objects list.
148
149         Returns:
150             None
151     """
152
153     def _track(self, tracked_objects):
154         """Tracks objects in the current frame.
155
156         Args:
157             tracked_objects (List[dict]): A list of dictionaries containing
158                 information about tracked objects, including their bounding
159                 boxes and states.
160
161         Returns:
162             None
163     """
164
165     def _detect(self, frame):
166         """Performs object detection on the current frame.
167
168         Args:
169             frame (np.ndarray): The current frame to process.
170
171         Returns:
172             List[dict]: A list of dictionaries containing information about
173                 detected objects, including their bounding boxes and states.
174         """
175
176         # Implement detection logic here
177
178         detected_objects = []
179
180         for class_id, class_name in enumerate(self._classes):
181             detected_objects.append({
182                 "id": class_id,
183                 "bbox": [0, 0, 0, 0],
184                 "state": "normal"
185             })
186
187         return detected_objects
188
189     def _process(self):
190         """Processes the pipeline.
191
192         Returns:
193             None
194     """
195
196     def _detect_and_track(self, frame):
197         """Performs object detection and tracking on the current frame.
198
199         Args:
200             frame (np.ndarray): The current frame to process.
201
202         Returns:
203             List[dict]: A list of dictionaries containing information about
204                 tracked objects, including their bounding boxes and states.
205         """
206
207         # Implement detection and tracking logic here
208
209         tracked_objects = []
210
211         for class_id, class_name in enumerate(self._classes):
212             tracked_objects.append({
213                 "id": class_id,
214                 "bbox": [0, 0, 0, 0],
215                 "state": "normal"
216             })
217
218         return tracked_objects
219
220     def _update_memory(self, tracked_objects, memory, classes):
221         """Updates tracking memory with detected and tracked objects.
222
223         Args:
224             tracked_objects (List[dict]): A list of dictionaries containing
225                 information about tracked objects, including their bounding
226                 boxes and states.
227             memory (SharedCircularBuffer): A shared circular buffer used
228                 to store tracked objects and their history.
229             classes (List[str]): A list of class names corresponding to the
230                 objects in the tracked_objects list.
231
232         Returns:
233             None
234     """
235
236     def _track(self, tracked_objects):
237         """Tracks objects in the current frame.
238
239         Args:
240             tracked_objects (List[dict]): A list of dictionaries containing
241                 information about tracked objects, including their bounding
242                 boxes and states.
243
244         Returns:
245             None
246     """
247
248     def _detect(self, frame):
249         """Performs object detection on the current frame.
250
251         Args:
252             frame (np.ndarray): The current frame to process.
253
254         Returns:
255             List[dict]: A list of dictionaries containing information about
256                 detected objects, including their bounding boxes and states.
257         """
258
259         # Implement detection logic here
260
261         detected_objects = []
262
263         for class_id, class_name in enumerate(self._classes):
264             detected_objects.append({
265                 "id": class_id,
266                 "bbox": [0, 0, 0, 0],
267                 "state": "normal"
268             })
269
270         return detected_objects
271
272     def _process(self):
273         """Processes the pipeline.
274
275         Returns:
276             None
277     """
278
279     def _detect_and_track(self, frame):
280         """Performs object detection and tracking on the current frame.
281
282         Args:
283             frame (np.ndarray): The current frame to process.
284
285         Returns:
286             List[dict]: A list of dictionaries containing information about
287                 tracked objects, including their bounding boxes and states.
288         """
289
290         # Implement detection and tracking logic here
291
292         tracked_objects = []
293
294         for class_id, class_name in enumerate(self._classes):
295             tracked_objects.append({
296                 "id": class_id,
297                 "bbox": [0, 0, 0, 0],
298                 "state": "normal"
299             })
300
301         return tracked_objects
302
303     def _update_memory(self, tracked_objects, memory, classes):
304         """Updates tracking memory with detected and tracked objects.
305
306         Args:
307             tracked_objects (List[dict]): A list of dictionaries containing
308                 information about tracked objects, including their bounding
309                 boxes and states.
310             memory (SharedCircularBuffer): A shared circular buffer used
311                 to store tracked objects and their history.
312             classes (List[str]): A list of class names corresponding to the
313                 objects in the tracked_objects list.
314
315         Returns:
316             None
317     """
318
319     def _track(self, tracked_objects):
320         """Tracks objects in the current frame.
321
322         Args:
323             tracked_objects (List[dict]): A list of dictionaries containing
324                 information about tracked objects, including their bounding
325                 boxes and states.
326
327         Returns:
328             None
329     """
330
331     def _detect(self, frame):
332         """Performs object detection on the current frame.
333
334         Args:
335             frame (np.ndarray): The current frame to process.
336
337         Returns:
338             List[dict]: A list of dictionaries containing information about
339                 detected objects, including their bounding boxes and states.
340         """
341
342         # Implement detection logic here
343
344         detected_objects = []
345
346         for class_id, class_name in enumerate(self._classes):
347             detected_objects.append({
348                 "id": class_id,
349                 "bbox": [0, 0, 0, 0],
350                 "state": "normal"
351             })
352
353         return detected_objects
354
355     def _process(self):
356         """Processes the pipeline.
357
358         Returns:
359             None
360     """
361
362     def _detect_and_track(self, frame):
363         """Performs object detection and tracking on the current frame.
364
365         Args:
366             frame (np.ndarray): The current frame to process.
367
368         Returns:
369             List[dict]: A list of dictionaries containing information about
370                 tracked objects, including their bounding boxes and states.
371         """
372
373         # Implement detection and tracking logic here
374
375         tracked_objects = []
376
377         for class_id, class_name in enumerate(self._classes):
378             tracked_objects.append({
379                 "id": class_id,
380                 "bbox": [0, 0, 0, 0],
381                 "state": "normal"
382             })
383
384         return tracked_objects
385
386     def _update_memory(self, tracked_objects, memory, classes):
387         """Updates tracking memory with detected and tracked objects.
388
389         Args:
390             tracked_objects (List[dict]): A list of dictionaries containing
391                 information about tracked objects, including their bounding
392                 boxes and states.
393             memory (SharedCircularBuffer): A shared circular buffer used
394                 to store tracked objects and their history.
395             classes (List[str]): A list of class names corresponding to the
396                 objects in the tracked_objects list.
397
398         Returns:
399             None
400     """
401
402     def _track(self, tracked_objects):
403         """Tracks objects in the current frame.
404
405         Args:
406             tracked_objects (List[dict]): A list of dictionaries containing
407                 information about tracked objects, including their bounding
408                 boxes and states.
409
410         Returns:
411             None
412     """
413
414     def _detect(self, frame):
415         """Performs object detection on the current frame.
416
417         Args:
418             frame (np.ndarray): The current frame to process.
419
420         Returns:
421             List[dict]: A list of dictionaries containing information about
422                 detected objects, including their bounding boxes and states.
423         """
424
425         # Implement detection logic here
426
427         detected_objects = []
428
429         for class_id, class_name in enumerate(self._classes):
430             detected_objects.append({
431                 "id": class_id,
432                 "bbox": [0, 0, 0, 0],
433                 "state": "normal"
434             })
435
436         return detected_objects
437
438     def _process(self):
439         """Processes the pipeline.
440
441         Returns:
442             None
443     """
444
445     def _detect_and_track(self, frame):
446         """Performs object detection and tracking on the current frame.
447
448         Args:
449             frame (np.ndarray): The current frame to process.
450
451         Returns:
452             List[dict]: A list of dictionaries containing information about
453                 tracked objects, including their bounding boxes and states.
454         """
455
456         # Implement detection and tracking logic here
457
458         tracked_objects = []
459
460         for class_id, class_name in enumerate(self._classes):
461             tracked_objects.append({
462                 "id": class_id,
463                 "bbox": [0, 0, 0, 0],
464                 "state": "normal"
465             })
466
467         return tracked_objects
468
469     def _update_memory(self, tracked_objects, memory, classes):
470         """Updates tracking memory with detected and tracked objects.
471
472         Args:
473             tracked_objects (List[dict]): A list of dictionaries containing
474                 information about tracked objects, including their bounding
475                 boxes and states.
476             memory (SharedCircularBuffer): A shared circular buffer used
477                 to store tracked objects and their history.
478             classes (List[str]): A list of class names corresponding to the
479                 objects in the tracked_objects list.
480
481         Returns:
482             None
483     """
484
485     def _track(self, tracked_objects):
486         """Tracks objects in the current frame.
487
488         Args:
489             tracked_objects (List[dict]): A list of dictionaries containing
490                 information about tracked objects, including their bounding
491                 boxes and states.
492
493         Returns:
494             None
495     """
496
497     def _detect(self, frame):
498         """Performs object detection on the current frame.
499
500         Args:
501             frame (np.ndarray): The current frame to process.
502
503         Returns:
504             List[dict]: A list of dictionaries containing information about
505                 detected objects, including their bounding boxes and states.
506         """
507
508         # Implement detection logic here
509
510         detected_objects = []
511
512         for class_id, class_name in enumerate(self._classes):
513             detected_objects.append({
514                 "id": class_id,
515                 "bbox": [0, 0, 0, 0],
516                 "state": "normal"
517             })
518
519         return detected_objects
520
521     def _process(self):
522         """Processes the pipeline.
523
524         Returns:
525             None
526     """
527
528     def _detect_and_track(self, frame):
529         """Performs object detection and tracking on the current frame.
530
531         Args:
532             frame (np.ndarray): The current frame to process.
533
534         Returns:
535             List[dict]: A list of dictionaries containing information about
536                 tracked objects, including their bounding boxes and states.
537         """
538
539         # Implement detection and tracking logic here
540
541         tracked_objects = []
542
543         for class_id, class_name in enumerate(self._classes):
544             tracked_objects.append({
545                 "id": class_id,
546                 "bbox": [0, 0, 0, 0],
547                 "state": "normal"
548             })
549
550         return tracked_objects
551
552     def _update_memory(self, tracked_objects, memory, classes):
553         """Updates tracking memory with detected and tracked objects.
554
555         Args:
556             tracked_objects (List[dict]): A list of dictionaries containing
557                 information about tracked objects, including their bounding
558                 boxes and states.
559             memory (SharedCircularBuffer): A shared circular buffer used
560                 to store tracked objects and their history.
561             classes (List[str]): A list of class names corresponding to the
562                 objects in the tracked_objects list.
563
564         Returns:
565             None
566     """
567
568     def _track(self, tracked_objects):
569         """Tracks objects in the current frame.
570
571         Args:
572             tracked_objects (List[dict]): A list of dictionaries containing
573                 information about tracked objects, including their bounding
574                 boxes and states.
575
576         Returns:
577             None
578     """
579
580     def _detect(self, frame):
581         """Performs object detection on the current frame.
582
583         Args:
584             frame (np.ndarray): The current frame to process.
585
586         Returns:
587             List[dict]: A list of dictionaries containing information about
588                 detected objects, including their bounding boxes and states.
589         """
590
591         # Implement detection logic here
592
593         detected_objects = []
594
595         for class_id, class_name in enumerate(self._classes):
596             detected_objects.append({
597                 "id": class_id,
598                 "bbox": [0, 0, 0, 0],
599                 "state": "normal"
600             })
601
602         return detected_objects
603
604     def _process(self):
605         """Processes the pipeline.
606
607         Returns:
608             None
609     """
610
611     def _detect_and_track(self, frame):
612         """Performs object detection and tracking on the current frame.
613
614         Args:
615             frame (np.ndarray): The current frame to process.
616
617         Returns:
618             List[dict]: A list of dictionaries containing information about
619                 tracked objects, including their bounding boxes and states.
620         """
621
622         # Implement detection and tracking logic here
623
624         tracked_objects = []
625
626         for class_id, class_name in enumerate(self._classes):
627             tracked_objects.append({
628                 "id": class_id,
629                 "bbox": [0, 0, 0, 0],
630                 "state": "normal"
631             })
632
633         return tracked_objects
634
635     def _update_memory(self, tracked_objects, memory, classes):
636         """Updates tracking memory with detected and tracked objects.
637
638         Args:
639             tracked_objects (List[dict]): A list of dictionaries containing
640                 information about tracked objects, including their bounding
641                 boxes and states.
642             memory (SharedCircularBuffer): A shared circular buffer used
643                 to store tracked objects and their history.
644             classes (List[str]): A list of class names corresponding to the
645                 objects in the tracked_objects list.
646
647         Returns:
648             None
649     """
650
651     def _track(self, tracked_objects):
652         """Tracks objects in the current frame.
653
654         Args:
655             tracked_objects (List[dict]): A list of dictionaries containing
656                 information about tracked objects, including their bounding
657                 boxes and states.
658
659         Returns:
660             None
661     """
662
663     def _detect(self, frame):
664         """Performs object detection on the current frame.
665
666         Args:
667             frame (np.ndarray): The current frame to process.
668
669         Returns:
670             List[dict]: A list of dictionaries containing information about
671                 detected objects, including their bounding boxes and states.
672         """
673
674         # Implement detection logic here
675
676         detected_objects = []
677
678         for class_id, class_name in enumerate(self._classes):
679             detected_objects.append({
680                 "id": class_id,
681                 "bbox": [0, 0, 0, 0],
682                 "state": "normal"
683             })
684
685         return detected_objects
686
687     def _process(self):
688         """Processes the pipeline.
689
690         Returns:
691             None
692     """
693
694     def _detect_and_track(self, frame):
695         """Performs object detection and tracking on the current frame.
696
697         Args:
698             frame (np.ndarray): The current frame to process.
699
700         Returns:
701             List[dict]: A list of dictionaries containing information about
702                 tracked objects, including their bounding boxes and states.
703         """
704
705         # Implement detection and tracking logic here
706
707         tracked_objects = []
708
709         for class_id, class_name in enumerate(self._classes):
710             tracked_objects.append({
711                 "id": class_id,
712                 "bbox": [0, 0, 0, 0],
713                 "state": "normal"
714             })
715
716         return tracked_objects
717
718     def _update_memory(self, tracked_objects, memory, classes):
719         """Updates tracking memory with detected and tracked objects.
720
721         Args:
722             tracked_objects (List[dict]): A list of dictionaries containing
723                 information about tracked objects, including their bounding
724                 boxes and states.
725             memory (SharedCircularBuffer): A shared circular buffer used
726                 to store tracked objects and their history.
727             classes (List[str]): A list of class names corresponding to the
728                 objects in the tracked_objects list.
729
730         Returns:
731             None
732     """
733
734     def _track(self, tracked_objects):
735         """Tracks objects in the current frame.
736
737         Args:
738             tracked_objects (List[dict]): A list of dictionaries containing
739                 information about tracked objects, including their bounding
740                 boxes and states.
741
742         Returns:
743             None
744     """
745
746     def _detect(self, frame):
747         """Performs object detection on the current frame.
748
749         Args:
750             frame (np.ndarray): The current frame to process.
751
752         Returns:
753             List[dict]: A list of dictionaries containing information about
754                 detected objects, including their bounding boxes and states.
755         """
756
757         # Implement detection logic here
758
759         detected_objects = []
760
761         for class_id, class_name in enumerate(self._classes):
762             detected_objects.append({
763                 "id": class_id,
764                 "bbox": [0, 0, 0, 0],
765                 "state": "normal"
766             })
767
768         return detected_objects
769
770     def _process(self):
771         """Processes the pipeline.
772
773         Returns:
774             None
775     """
776
777     def _detect_and_track(self, frame):
778         """Performs object detection and tracking on the current frame.
779
780         Args:
781             frame (np.ndarray): The current frame to process.
782
783         Returns:
784             List[dict]: A list of dictionaries containing information about
785                 tracked objects, including their bounding boxes and states.
786         """
787
788         # Implement detection and tracking logic here
789
790         tracked_objects = []
791
792         for class_id, class_name in enumerate(self._classes):
793             tracked_objects.append({
794                 "id": class_id,
795                 "bbox": [0, 0, 0, 0],
796                 "state": "normal"
797             })
798
799         return tracked_objects
800
801     def _update_memory(self, tracked_objects, memory, classes):
802         """Updates tracking memory with detected and tracked objects.
803
804         Args:
805             tracked_objects (List[dict]): A list of dictionaries containing
806                 information about tracked objects, including their bounding
807                 boxes and states.
808             memory (SharedCircularBuffer): A shared circular buffer used
809                 to store tracked objects and their history.
810             classes (List[str]): A list of class names corresponding to the
811                 objects in the tracked_objects list.
812
813         Returns:
814             None
815     """
816
817     def _track(self, tracked_objects):
818         """Tracks objects in the current frame.
819
820         Args:
821             tracked_objects (List[dict]): A list of dictionaries containing
822                 information about tracked objects, including their bounding
823                 boxes and states.
824
825         Returns:
826             None
827     """
828
829     def _detect(self, frame):
830         """Performs object detection on the current frame.
831
832         Args:
833             frame (np.ndarray): The current frame to process.
834
835         Returns:
836             List[dict]: A list of dictionaries containing information about
837                 detected objects, including their bounding boxes and states.
838         """
839
840         # Implement detection logic here
841
842         detected_objects = []
843
844         for class_id, class_name in enumerate(self._classes):
845             detected_objects.append({
846                 "id": class_id,
847                 "bbox": [0, 0, 0, 0],
848                 "state": "normal"
849             })
850
851         return detected_objects
852
853     def _process(self):
854         """Processes the pipeline.
855
856         Returns:
857             None
858     """
859
860     def _detect_and_track(self, frame):
861         """Performs object detection and tracking on the current frame.
862
863         Args:
864             frame (np.ndarray): The current frame to process.
865
866         Returns:
867             List[dict]: A list of dictionaries containing information about
868                 tracked objects, including their bounding boxes and states.
869         """
870
871         # Implement detection and tracking logic here
872
873         tracked_objects = []
874
875         for class_id, class_name in enumerate(self._classes):
876             tracked_objects.append({
877                 "id": class_id,
878                 "bbox": [0, 0, 0, 0],
879                 "state": "normal"
880             })
881
882         return tracked_objects
883
884     def _update_memory(self, tracked_objects, memory, classes):
885         """Updates tracking memory with detected and tracked objects.
886
887         Args:
888             tracked_objects (List[dict]): A list of dictionaries containing
889                 information about tracked objects, including their bounding
890                 boxes and states.
891             memory (SharedCircularBuffer): A shared circular buffer used
892                 to store tracked objects and their history.
893             classes (List[str]): A list of class names corresponding to the
894                 objects in the tracked_objects list.
895
896         Returns:
897             None
898     """
899
900     def _track(self, tracked_objects):
901         """Tracks objects in the current frame.
902
903         Args:
904             tracked_objects (List[dict]): A list of dictionaries containing
905                 information about tracked objects, including their bounding
906                 boxes and states.
907
908         Returns:
909             None
910     """
911
912     def _detect(self, frame):
913         """Performs object detection on the current frame.
914
915         Args:
916             frame (np.ndarray): The current frame to process.
917
918         Returns:
919             List[dict]: A list of dictionaries containing information about
920                 detected objects, including their bounding boxes and states.
921         """
922
923         # Implement detection logic here
924
925         detected_objects = []
926
927         for class_id, class_name in enumerate(self._classes):
928             detected_objects.append({
929                 "id": class_id,
930                 "bbox": [0, 0, 0, 0],
931                 "state": "normal"
932             })
933
934         return detected_objects
935
936     def _process(self):
937         """Processes the pipeline.
938
939         Returns:
940             None
941     """
942
943     def _detect_and_track(self, frame):
944         """Performs object detection and tracking on the current frame.
945
946         Args:
947             frame (np.ndarray): The current frame to process.
948
949         Returns:
950             List[dict]: A list of dictionaries containing information about
951                 tracked objects, including their bounding boxes and states.
952         """
953
954         # Implement detection and tracking logic here
955
956         tracked_objects = []
957
958         for class_id, class_name in enumerate(self._classes):
959             tracked_objects.append({
960                 "id": class_id,
961                 "bbox": [0, 0, 0, 0],
962                 "state": "normal"
963             })
964
965         return tracked_objects
966
967     def _update_memory(self, tracked_objects, memory, classes):
968         """Updates tracking memory with detected and tracked objects.
969
970         Args:
971             tracked_objects (List[dict]): A list of dictionaries containing
972                 information about tracked objects, including their bounding
973                 boxes and states.
974             memory (SharedCircularBuffer): A shared circular buffer used
975                 to store tracked objects and their history.
976             classes (List[str]): A list of class names corresponding to the
977                 objects in the tracked_objects list.
978
979         Returns:
980             None
981     """
982
983     def _track(self, tracked_objects):
984         """Tracks objects in the current frame.
985
986         Args:
987             tracked_objects (List[dict]): A list of dictionaries containing
988                 information about tracked objects, including their bounding
989                 boxes and states.
990
991         Returns:
992             None
993     """
994
995     def _detect(self, frame):
996         """Performs object detection on the current frame.
997
998         Args:
999             frame (np.ndarray): The current frame to process.
1000
1001         Returns:
1002             List[dict]: A list of dictionaries containing information about
1003                 detected objects, including their bounding boxes and states.
1004         """
1005
1006         # Implement detection logic here
1007
1008         detected_objects = []
1009
1010         for class_id, class_name in enumerate(self._classes):
1011             detected_objects.append({
1012                 "id": class_id,
1013                 "bbox": [0, 0, 0, 0],
1014                 "state": "normal"
1015             })
1016
1017         return detected_objects
1018
1019     def _process(self):
1020         """Processes the pipeline.
1021
1022         Returns:
1023             None
1024     """
1025
1026     def _detect_and_track(self, frame):
1027         """Performs object detection and tracking on the current frame.
1028
1029         Args:
1030             frame (np.ndarray): The current frame to process.
1031
1032         Returns:
1033             List[dict]: A list of dictionaries containing information about
1034                 tracked objects, including their bounding boxes and states.
1035         """
1036
1037         # Implement detection and tracking logic here
1038
1039         tracked_objects = []
1040
1041         for class_id, class_name in enumerate(self._classes):
1042             tracked_objects.append({
1043                 "id": class_id,
1044                 "bbox": [0, 0, 0, 0],
1045                 "state": "normal"
1046             })
1047
1048         return tracked_objects
1049
1050     def _update_memory(self, tracked_objects, memory, classes):
1051         """Updates tracking memory with detected and tracked objects.
1052
1053         Args:
1054             tracked_objects (List[dict]): A list of dictionaries containing
1055                 information about tracked objects, including their bounding
1056                 boxes and states.
1057             memory (SharedCircularBuffer): A shared circular buffer used
1058                 to store tracked objects and their history.
1059             classes (List[str]): A list of class names corresponding to the
1060                 objects in the tracked_objects list.
1061
1062         Returns:
1063             None
1064     """
1065
1066     def _track(self, tracked_objects):
1067         """Tracks objects in the current frame.
1068
1069         Args:
1070             tracked_objects (List[dict]): A list of dictionaries containing
1071                 information about tracked objects, including their bounding
1072                 boxes and states.
1073
1074         Returns:
1075             None
1076     """
1077
1078     def _detect(self, frame):
1079         """Performs object detection on the current frame.
1080
1081         Args:
1082             frame (np.ndarray): The current frame to process.
1083
1084         Returns:
1085             List[dict]: A list of dictionaries containing information about
1086                 detected objects, including their bounding boxes and states.
1087         """
1088
1089         # Implement detection logic here
1090
1091         detected_objects = []
1092
1093         for class_id, class_name in enumerate(self._classes):
1094             detected_objects.append({
1095                 "id": class_id,
1096                 "bbox": [0, 0, 0, 0],
1097                 "state": "normal"
1098             })
1099
1100         return detected_objects
1101
1102     def _process(self):
1103         """Processes the pipeline.
1104
1105         Returns:
1106             None
1107     """
1108
1109     def _detect_and_track(self, frame):
1110         """Performs object detection and tracking on the current frame.
1111
1112         Args:
1113             frame (np.ndarray): The current frame to process.
1114
1115         Returns:
1116             List[dict]: A list of dictionaries containing information about
1117                 tracked objects, including their bounding boxes and states.
1118         """
1119
1120         # Implement detection and tracking logic here
1121
1122         tracked_objects = []
1123
1124         for class_id, class_name in enumerate(self._classes):
1125             tracked_objects.append({
1126                 "id": class_id,
1127                 "bbox": [0, 0, 0, 0],
1128                 "state": "normal"
1129             })
1130
1131         return tracked_objects
1132
1133     def _update_memory(self, tracked_objects, memory, classes):
1134         """Updates tracking memory with detected and tracked objects.
1135
1136         Args:
1137             tracked_objects (List[dict]): A list of dictionaries containing
1138                 information about tracked objects, including their bounding
1139                 boxes and states.
1140             memory (SharedCircularBuffer): A shared circular buffer used
1141                 to store tracked objects and their history.
1142             classes (List[str]): A list of class names corresponding to the
1143                 objects in the tracked_objects list.
1144
1145         Returns:
1146             None
1147     """
1148
1149     def _track(self, tracked_objects):
1150         """Tracks objects in the current frame.
1151
1152         Args:
1153             tracked_objects (List[dict]): A list of dictionaries containing
1154                 information about tracked objects, including their bounding
1155                 boxes and states.
1156
1157         Returns:
1158             None
1159     """
1160
1161     def _detect(self, frame):
1162         """Performs object detection on the current frame.
1163
1164         Args:
1165             frame (np.ndarray): The current frame to process.
1166
1167         Returns:
1168             List[dict]: A list of dictionaries containing information about
1169                 detected objects, including their bounding boxes and states.
1170         """
1171
1172         # Implement detection logic here
1173
1174         detected_objects = []
1175
1176         for class_id, class_name in enumerate(self._classes):
1177             detected_objects.append({
1178                 "id": class_id,
1179                 "bbox": [0, 0, 0, 0],
1180                 "state": "normal"
1181             })
1182
1183         return detected_objects
1184
1185     def _process(self):
1186         """Processes the pipeline.
1187
1188         Returns:
1189             None
1190     """
1191
1192     def _detect_and_track(self, frame):
1193         """Performs object detection and tracking on the current frame.
1194
1195         Args:
1196             frame (np.ndarray): The current frame to process.
1197
1198         Returns:
1199             List[dict]: A list of dictionaries containing information about
1200                 tracked objects, including their bounding boxes and states.
1201         """
1202
1203         # Implement detection and tracking logic here
1204
1205         tracked_objects = []
1206
1207         for class_id, class_name in enumerate(self._classes):
1208             tracked_objects.append({
1209                 "id": class_id,
1210                 "bbox": [0, 0, 0, 0],
1211                 "state": "normal"
1212             })
1213
1214         return tracked_objects
1215
1216     def _update_memory(self, tracked_objects, memory, classes):
1217         """Updates tracking memory with detected and tracked objects.
1218
1219         Args:
1220             tracked_objects (List[dict]): A
```

```

56     tracked_objects: List of tracked objects in the current frame.
57         Each object is a tuple or list with
58             information like
59                 tracking ID, detected class, etc.
60             memory: Dictionary that stores the state of tracked objects
61                 between frames. Keys are tracking IDs.
62             classes: Dictionary that maps class IDs to their names.
63             """
64
65             FRAME_AGE = 60 # Number of frames to keep an object in memory if
66             not visible
67             PERMANENT_DEFECT_THRESHOLD = 3 # Consecutive frames to mark as "
68             permanent defect"
69
70
71             for obj in tracked_objects:
72                 track_id = int(obj[4])
73                 detected_class = classes[int(obj[6])]
74                 is_defective = detected_class.endswith("-d")
75
76                 if track_id in memory:
77                     entry = memory[track_id]
78
79                     # If already marked as permanent defect, only update its
80                     visibility
81                     if entry.get("permanent_defect", False):
82                         entry["visible_frames"] = FRAME_AGE
83                         continue
84
85                     # Updates the consecutive defect counter
86                     if is_defective:
87                         entry["defect_counter"] = entry.get("defect_counter",
88                             0) + 1
89                     else:
90                         entry["defect_counter"] = 0 # Reset if not defective
91                         in this frame
92
93                     # Mark as permanent defect if it reaches the threshold
94                     if entry["defect_counter"] >= PERMANENT_DEFECT_THRESHOLD:
95                         entry["permanent_defect"] = True
96                         # The class already includes '-d', no need to reassign
97                         'detected_class' here
98                         # entry["defective"] will be updated below
99
100
101                     # Update defective status and visibility
102                     entry["defective"] = entry.get("permanent_defect", False)
103                     or is_defective
104                     entry["visible_frames"] = FRAME_AGE
105                     entry["class"] = detected_class
106
107                     else:
108                         # New detected object
109                         memory[track_id] = {
110                             "defective": is_defective,
111                             "visible_frames": FRAME_AGE,
112                             "class": detected_class,
113                             "defect_counter": 1 if is_defective else 0,
114                             "permanent_defect": False, # Initialize as not
115                                         permanent
116                         }
117
118
119                     # Decrement visibility and remove old objects
120                     for track_id in list(memory): # Iterate over a copy of the keys
121                         memory[track_id]["visible_frames"] -= 1
122                         if memory[track_id]["visible_frames"] <= 0:
123                             del memory[track_id]

```

```

111     def capture_frames(
112         self,
113         video_path: str,
114         frame_queue: Union[mp.Queue, SharedCircularBuffer],
115         t1_start: mp.Event,
116         stop_event: mp.Event,
117         tcp_event: mp.Event,
118         is_tcp: bool,
119         mp_stop_event: Optional[mp.Event] = None,
120         mh_num: int = 1,
121         is_process: bool = False,
122         max_frames: Optional[int] = None,
123     ):
124         """Captures frames from a video file and queues them for processing
125
126         Reads frames from a video specified by 'video_path'. If 'is_tcp' is
127         True,
128         waits for 'tcp_event' to be set before starting capture.
129         Frames are put into 'frame_queue'. If 'max_frames' is defined,
130         attempts to maintain that FPS rate by limiting capture speed.
131         When finished or if 'stop_event' is set, sends 'None' to the queue
132         (as many times as 'mh_num') to signal the end of capture.
133
134         Args:
135             video_path: Path to the video file.
136             frame_queue: Queue (multiprocessing or shared circular buffer)
137                 to send frames.
138             t1_start: Multiprocessing event to synchronize start.
139             stop_event: Event to stop capture.
140             tcp_event: Event for synchronization in TCP mode.
141             is_tcp: Boolean indicating if operating in TCP mode.
142             mp_stop_event: Optional event to wait before terminating
143                 process/thread.
144             mh_num: Number of queue consumers (to send multiple 'None' at
145                 the end).
146             is_process: Boolean, True if running as a separate process.
147             max_frames: Maximum desired FPS for capture.
148
149         Raises:
150             FileNotFoundError: If 'video_path' does not exist.
151             IOError: If the video cannot be opened.
152
153
154     if not os.path.exists(video_path):
155         logging.error(f"Video file does not exist: {video_path}")
156         for _ in range(mh_num): # Notify all consumers
157             frame_queue.put(None)
158         raise FileNotFoundError(f"Video file does not exist: {video_path}")
159
160     cap = cv2.VideoCapture(video_path)
161
162     if not cap.isOpened():
163         logging.error(f"Error opening video file: {video_path}")
164         for _ in range(mh_num): # Notify all consumers
165             frame_queue.put(None)
166         raise IOError(f"Error opening video file: {video_path}")
167
168     # Wait for TCP signal if enabled
169     if is_tcp:
170         tcp_event.wait()
171
172     frame_count = 0

```

```

169     first_time = True # To record the time of the first processed
170         frame
171
172     # Wait for t1_start signal
173     t1_start.wait()
174
175     # Calculate time per frame if max_fps is specified
176     frame_time_target = 1 / max_frames if max_frames else None
177
178     logging.info("Starting frame capture...")
179     while cap.isOpened() and not stop_event.is_set():
180         loop_start_time = time.time()
181
182         if first_time: # This t1 seems to be for a benchmark, not for
183             FPS logic
184             t1 = cv2.getTickCount()
185             first_time = False
186
187         ret, frame = cap.read()
188
189         if not ret:
190             logging.info("End of video or read error.")
191             break
192
193         try:
194             # Try to put the frame in the queue, without waiting if
195             # there's an FPS limit
196             # (to discard frames if the queue is full and maintain the
197             # pace)
198             if max_frames:
199                 frame_queue.put_nowait((frame, frame_count))
200             else:
201                 frame_queue.put((frame, frame_count))
202
203             except Exception as e: # It would be better to catch a more
204                 specific exception if known
205                 logging.warning(f"Could not queue frame {frame_count}: {e}")
206
207             # Decide whether to continue or not, here the frame is
208             # simply skipped
209             pass
210
211
212             # If there's a frame_time_target, sleep to not exceed
213             # max_frames
214             if frame_time_target:
215                 elapsed_time = time.time() - loop_start_time
216                 if elapsed_time < frame_time_target:
217                     time.sleep(frame_time_target - elapsed_time)
218
219             frame_count += 1
220
221             cap.release()
222             logging.info(f"Capture finished. Total frames read: {frame_count}")
223
224             # Signal the end to consumers
225             for _ in range(mh_num):
226                 frame_queue.put(None)
227
228             # Wait for the main process/thread stop signal if necessary
229             if mp_stop_event:
230                 mp_stop_event.wait()
231
232             # Terminate the process if running as such
233             if is_process:

```

```

225     logging.info("Terminating capture process.")
226     os._exit(0)
227
228     def process_frames(
229         self,
230         frame_queue: Union[mp.Queue, SharedCircularBuffer],
231         detection_queue: Union[mp.Queue, SharedCircularBuffer],
232         model_path: str,
233         t1_start: mp.Event,
234         mp_stop_event: Optional[mp.Event] = None,
235         is_process: bool = False,
236     ):
237         """Processes frames from a queue, performs object detection, and
238         queues the results.
239
240         Consumes frames from 'frame_queue', uses a YOLO model (loaded from
241         'model_path') to detect objects, and then queues the original frame
242         along with detection results in 'detection_queue'.
243         Signals 't1_start' after initializing the model.
244
245         Args:
246             frame_queue: Input queue with frames to process.
247             detection_queue: Output queue for frames with detections.
248             model_path: Path to the YOLO model file.
249             t1_start: Event to signal that model initialization has
250                     finished.
251             mp_stop_event: Optional event to wait before terminating the
252                         process/thread.
253             is_process: Boolean, True if running as a separate process.
254
255         """
256
257     from ultralytics import YOLO
258
259     logging.info(f"Loading model from: {model_path}")
260     model = YOLO(model_path, task="detect")
261
262     # Model warm-up
263     # This can improve the speed of the first real inferences.
264     logging.info("Performing model warm-up...")
265     # It's assumed that the model has these default parameters or they
266     # are configurable.
267     # It's good practice to do warm-up with data similar to the input.
268     # Here a generic configuration is used.
269     try:
270         model(conf=0.5, half=True, imgsz=(640, 640), augment=True,
271               verbose=False)
272     except Exception as e:
273         logging.warning(f"Error during model warm-up: {e}")
274
275     logging.info("Model loaded and ready. Signaling t1_start.")
276     t1_start.set() # Signal that the model is ready
277
278     while True:
279         item = frame_queue.get()
280         if item is None: # End signal
281             detection_queue.put(None) # Propagate the signal
282             logging.info("End signal received in process_frames.")
283             break
284
285         frame, frame_count = item
286
287         # Perform preprocessing (assuming model.predictor exists and
288         # has these methods)
289         # It's important to verify the API of the ultralytics version
290         # being used.

```



```

339     # Perform tracking
340     # 'result_detections' must be compatible with what 'tracker_wrapper.track' expects
341     tracked_outputs = tracker_wrapper.track(result_detections,
342                                              frame)
343
344     tracking_queue.put((frame, tracked_outputs))
345
346     if mp_stop_event:
347         mp_stop_event.wait()
348
349     if is_process:
350         logging.info("Terminating tracking process.")
351         os._exit(0)
352
353 def tracking_frames_multihardware(
354     self,
355     detection_queue_GPU: Union[mp.Queue, SharedCircularBuffer],
356     detection_queue_DLAO: Union[mp.Queue, SharedCircularBuffer],
357     detection_queue_DLAI: Union[mp.Queue, SharedCircularBuffer],
358     tracking_queue: Union[mp.Queue, SharedCircularBuffer],
359     mp_stop_event: Optional[mp.Event] = None,
360     is_process: bool = False,
361 ):
362     """Performs object tracking from multiple detection queues (multi-hardware).
363
364     Consumes frames and detections from three different queues ('detection_queue_GPU',
365     'detection_queue_DLAO', 'detection_queue_DLAI'), which are assumed to come from
366     different hardware accelerators. Orders frames by their frame number
367     before processing them with 'TrackerWrapper' to maintain temporal coherence.
368     Tracking results are queued in 'tracking_queue'.
369
370     Args:
371         detection_queue_GPU: Detection queue for GPU.
372         detection_queue_DLAO: Detection queue for DLAO.
373         detection_queue_DLAI: Detection queue for DLAI.
374         tracking_queue: Output queue for frames with tracked objects.
375         mp_stop_event: Optional event to wait before terminating the process/thread.
376         is_process: Boolean, True if running as a separate process.
377     """
378     tracker_wrapper = TrackerWrapper(frame_rate=30) # Adjust frame_rate if necessary
379     logging.info("Multi-hardware tracker initialized.")
380
381     # Flags to control if each input queue has finished
382     stop_gpu, stop_dla0, stop_dla1 = False, False, False
383     # Buffers to store the last item read from each queue
384     item_gpu, item_dla0, item_dla1 = None, None, None
385
386     while True:
387         # Try to get a new item from each queue if it's not stopped and the buffer is empty
388         if not stop_gpu and item_gpu is None:
389             item_gpu = detection_queue_GPU.get()
390             if item_gpu is None:
391                 stop_gpu = True
392                 logging.info("GPU queue finished.")

```

```

393     if not stop_dla0 and item_dla0 is None:
394         item_dla0 = detection_queue_DLA0.get()
395         if item_dla0 is None:
396             stop_dla0 = True
397             logging.info("DLA0 queue finished.")
398
399     if not stop_dla1 and item_dla1 is None:
400         item_dla1 = detection_queue_DLA1.get()
401         if item_dla1 is None:
402             stop_dla1 = True
403             logging.info("DLA1 queue finished.")
404
405     # If all input queues have finished, terminate this process/
406     # thread
407     if stop_gpu and stop_dla0 and stop_dla1:
408         tracking_queue.put(None) # Signal the end to the next in
409         the chain
410         logging.info("All detection queues finished. Terminating multi-hardware tracking.")
411         if mp_stop_event:
412             mp_stop_event.wait()
413         if is_process:
414             os._exit(0)
415         break # Exit the while loop
416
417     # Extract frame numbers from current items (if they exist)
418     # The expected item format is (frame, result, times,
419     # frame_number)
420     # A very high value (float('inf')) is used if the item is None
421     # or doesn't have frame_number,
422     # so that valid items have priority.
423     frame_number_gpu = item_gpu[3] if item_gpu else float("inf")
424     frame_number_dla0 = item_dla0[3] if item_dla0 else float("inf")
425     frame_number_dla1 = item_dla1[3] if item_dla1 else float("inf")
426
427     # Select the item with the lowest frame number to process
428     # This ensures frames are processed in chronological order
429     selected_item = None
430     if (
431         frame_number_gpu <= frame_number_dla0
432         and frame_number_gpu <= frame_number_dla1
433         and item_gpu is not None
434     ):
435         selected_item = item_gpu
436         item_gpu = None # Empty the buffer so the next item from
437         this queue is read
438     elif (
439         frame_number_dla0 <= frame_number_gpu
440         and frame_number_dla0 <= frame_number_dla1
441         and item_dla0 is not None
442     ):
443         selected_item = item_dla0
444         item_dla0 = None
445     elif (
446         frame_number_dla1 <= frame_number_gpu
447         and frame_number_dla1 <= frame_number_dla0
448         and item_dla1 is not None
449     ):
449         selected_item = item_dla1
450         item_dla1 = None
451     else:
452         # If there are no valid items or all buffers are empty (and
453         # some queue hasn't finished)

```

```

450         # wait a bit to not consume CPU unnecessarily.
451         # This can happen if one queue is much faster than the
452         # others and the others are waiting for data.
453         if (
454             item_gpu is None
455             and item_dla0 is None
456             and item_dla1 is None
457             and not (stop_gpu and stop_dla0 and stop_dla1)
458         ):
459             time.sleep(0.001) # Small pause
460             continue # Return to the beginning of the loop to re-
461             evaluate or read new inputs
462
463             frame, result_detections, _, _ = (
464                 selected_item # times and frame_number are not used
465                 directly here
466             )
467
468             # Perform tracking
469             tracked_outputs = tracker_wrapper.track(result_detections,
470                 frame)
471             tracking_queue.put((frame, tracked_outputs))
472
473     def draw_and_write_frames(
474         self,
475         tracking_queue: Union[mp.Queue, SharedCircularBuffer],
476         times_queue: Union[
477             mp.Queue, SharedCircularBuffer
478         ], # Assume this queue is for benchmark times
479         output_video_path: str,
480         classes: dict, # Mapping of class ID to name
481         memory: dict, # Shared/updated tracking memory
482         colors: dict, # Mapping of class name to color for drawing
483         stop_event: mp.Event, # Event to stop this process/thread
484         tcp_conn_event: mp.Event, # Event to signal TCP connection
485             establishment (renamed from tcp_conn)
486         is_tcp: bool,
487         mp_stop_event: Optional[mp.Event] = None,
488         is_process: bool = False,
489     ):
490         """Draws tracked objects on frames, writes output video and handles
491             TCP communication.
492
493             Consumes frames with tracked objects from 'tracking_queue'. Draws
494             rectangles
495             and labels for each object using information from 'memory' and 'colors'.
496             Writes processed frames to a video file at 'output_video_path'.
497             If 'is_tcp' is True, establishes a TCP server and sends "
498                 DETECTED_DEFECT"
499             messages when a defective object is detected for the first time (
500                 according to 'sended_id').
501             Signals 'tcp_conn_event' after starting the TCP server.
502             Upon completion, puts 'None' in 'times_queue' and activates 'stop_event'.
503
504             Args:
505                 tracking_queue: Input queue with frames and tracked objects.
506                 times_queue: Queue to send a completion signal (or times).
507                 output_video_path: Path to save the output video.
508                 classes: Dictionary mapping class ID to name. (Used indirectly
509                     via 'update_memory')
510                 memory: Tracking memory dictionary.
511                 colors: Dictionary mapping class name to color.

```

```

502     stop_event: Global event to stop all processes/threads in the
503         pipeline.
504     tcp_conn_event: Event to signal that TCP connection is ready.
505     is_tcp: Boolean indicating if operating in TCP mode.
506     mp_stop_event: Optional event to wait before terminating the
507         process/thread.
508     is_process: Boolean, True if running as a separate process.
509     """
510
511     from concurrent.futures import ThreadPoolExecutor
512
513     # ThreadPoolExecutor to handle background tasks (e.g. TCP sending)
514     # max_workers could be adjusted according to expected load.
515     thread_pool = ThreadPoolExecutor(max_workers=8)
516     video_writer = None # Initialize VideoWriter to None
517     frame_number_counter = 0 # Counter for written frames
518
519     # Dictionary to track defect IDs already sent via TCP
520     # to avoid sending multiple messages for the same defect.
521     sended_defect_ids = {}
522
523     client_socket, server_socket = None, None # Initialize sockets
524
525     if is_tcp:
526         try:
527             logging.info("Starting TCP server on 0.0.0.0:8765...")
528             client_socket, server_socket = tcp_server("0.0.0.0", 8765)
529             # Send "READY" in a separate thread to not block.
530             thread_pool.submit(handle_send, client_socket, "READY")
531             tcp_conn_event.set() # Signal that TCP server is ready
532         except Exception as e:
533             raise RuntimeError(f"Error starting TCP server: {e}")
534
535     while True:
536         item = tracking_queue.get()
537         if item is None: # End signal
538             logging.info("End signal received in draw_and_write_frames.")
539             break
540
541         frame, tracked_objects = item
542
543         # Initialize VideoWriter with the first frame to get dimensions
544         if video_writer is None:
545             try:
546                 frame_height, frame_width = frame.shape[:2]
547                 fourcc = cv2.VideoWriter_fourcc(*"mp4v") # Codec for .mp4
548                 video_writer = cv2.VideoWriter(
549                     output_video_path, fourcc, 30, (frame_width,
550                         frame_height))
551                 logging.info(f"VideoWriter initialized for: {output_video_path}")
552             except Exception as e:
553                 logging.error(f"Error initializing VideoWriter: {e}")
554                 break
555
556         # Update memory with current tracked objects
557         self.update_memory(tracked_objects, memory, classes)
558
559         tcp_message_sent_this_frame = False
560
561         # Internal function to draw a single object on the frame

```

```
560     def draw_single_object(obj_data):
561         nonlocal frame, memory, colors, tcp_message_sent_this_frame
562         , is_tcp, client_socket, sended_defect_ids
563
564         # Expected format: (xmin, ymin, xmax, ymax, obj_id, conf,
565         # ...
566         xmin, ymin, xmax, ymax, obj_id = map(int, obj_data[:5])
567         confidence = float(obj_data[5])
568
569         # Confidence threshold to draw the object
570         if confidence < 0.4:
571             return
572
573         # Get updated class and memory state
574         obj_id_in_memory = memory.get(obj_id)
575         if not obj_id_in_memory:
576             return
577
578         current_class_name = obj_id_in_memory["class"]
579         is_currently_defective = current_class_name.endswith("-d")
580
581         # TCP sending logic for defects
582         if (
583             is_tcp
584             and is_currently_defective
585             and not tcp_message_sent_this_frame
586             and not sended_defect_ids.get(obj_id)
587         ):
588             sended_defect_ids[obj_id] = True
589
590             # Send TCP message in a pool thread to not block
591             # drawing
592             thread_pool.submit(handle_send, client_socket, "
593                 DETECTED_DEFECT")
594             tcp_message_sent_this_frame =
595                 True # Mark that a message was already sent in
596                 this frame
597             )
598             logging.debug(f"[TCP] Sending 'DETECTED_DEFECT' for ID {obj_id}")
599
600             # Draw rectangle and text
601             object_color = colors.get(
602                 current_class_name, (255, 255, 255)
603             ) # Default color: white
604             cv2.rectangle(frame, (xmin, ymin), (xmax, ymax),
605             object_color, 2)
606             text_label = f"ID:{obj_id} {current_class_name} {confidence
607             :.2f}"
608             cv2.putText(
609                 frame,
610                 text_label,
611                 (xmin, ymin - 10), # Text position above the rectangle
612                 cv2.FONT_HERSHEY_SIMPLEX,
613                 0.5, # Font size
614                 (255, 255, 255), # Text color (white)
615                 2, # Line thickness
616             )
617
618             # Draw all tracked objects using the thread pool
619             draw_tasks = [thread_pool.submit(draw_single_object, obj) for
620             obj in tracked_objects]
621             for task in draw_tasks: # Wait for all drawing tasks to
622             complete
```

```

        task.result()

616     # Draw frame number on the video
617     cv2.putText(
618         frame,
619         f"Frame:{frame_number_counter}",
620         (10, 30), # Position
621         cv2.FONT_HERSHEY_SIMPLEX,
622         1, # Size
623         (0, 255, 0), # Color (green)
624         2, # Thickness
625     )

626

627     if video_writer:
628         video_writer.write(frame)
629         frame_number_counter += 1
630

631     # Finalization and cleanup
632     if video_writer:
633         video_writer.release()
634

635     thread_pool.shutdown(wait=True) # Close thread pool waiting for
636     pending tasks to finish
637

638     stop_event.set() # Activate global stop event for other processes/
639     threads
640

641     if mp_stop_event:
642         mp_stop_event.wait()
643

644     if is_tcp and client_socket:
645         try:
646             client_socket.close()
647         except Exception as e:
648             print(f"Error closing TCP client socket: {e}")
649     if is_tcp and server_socket:
650         try:
651             server_socket.close()
652         except Exception as e:
653             print(f"Error closing TCP server socket: {e}")
654

655     if is_process:
656         os._exit(0)

657 def __init__(
658     self,
659     video_path: str,
660     model_path: str,
661     output_video_path: str,
662     output_times: str,
663     parallel_mode: str,
664     is_tcp: bool = False,
665     sahi: bool = False,
666     max_fps: int = None,
667     mh_num: int = 1,
668     is_process: bool = True,
669 ):
670     """Initializes the pipeline with common configuration and control
671     events.
672

673     Args:
674         video_path: Path to the input video file.
675         model_path: Path to the detection model file.
676         output_video_path: Path to save the processed video.

```

```
675     output_times: Path to save timing/benchmark information.
676     parallel_mode: Parallelization mode (e.g. 'sequential', ,
677                     processes', 'threads').
678     is_tcp: Enables TCP communication for defect notifications.
679     sahi: Enables the use of SAHI (Slice-Aided Hyper Inference) for
680           detection. (Not implemented in this fragment)
681     max_fps: Limits the FPS of video capture.
682     mh_num: Number of handlers/consumers for certain queues (multi-
683               hardware/processing).
684     is_process: Indicates if pipeline components run as separate
685                  processes.
686
687     """
688     self.video_path = video_path
689     self.model_path = model_path
690     self.output_video_path = output_video_path
691     self.output_times = output_times
692     self.parallel_mode = parallel_mode
693     self.is_tcp = is_tcp
694     self.sahi = sahi
695     self.max_fps = max_fps
696     self.mh_num = mh_num
697     self.is_process = is_process
698
699     # Common control events
700     self.tcp_event = mp.Event()
701     self.stop_event = mp.Event()
702     self.t1_start = mp.Event()
703     self.mp_stop_event = mp.Event() if is_process else None
704
705     # Shared memory
706     self.memory = {}
707
708     @abstractmethod
709     def _initialize_queues(self):
710         """Abstract method to initialize communication queues between
711             stages.
712
713             Must be implemented by derived classes to configure queues
714             (e.g. 'mp.Queue', 'SharedCircularBuffer') according to the
715                 parallelization
716                 strategy and pipeline type.
717             """
718
719         pass
720
721     @abstractmethod
722     def _initialize_events(self):
723         """Abstract method to initialize pipeline control events.
724
725             Must be implemented by derived classes to configure events
726             necessary for synchronization and pipeline flow control.
727             """
728
729         pass
730
731     @abstractmethod
732     def _create_workers(self):
733         """Abstract method to create pipeline workers.
734             Must be implemented by derived classes to start processes,
735             threads, or any other parallel execution mechanism needed
736             for pipeline stages.
737             """
738
739         pass
740
741     @abstractmethod
742     def _cleanup(self):
```

```
733     """Abstract method for resource cleanup when finishing the pipeline
734
735     Must be implemented by derived classes to free resources such as
736     processes, threads, queues, events, or any other handlers opened
737     during pipeline execution.
738     """
739
740     pass
741
742     @abstractmethod
743     def run(self):
744         """Executes the complete pipeline.
745
746         This is the main method that orchestrates the startup, execution
747         and
748         orderly finalization of all pipeline stages.
749         Must be implemented by derived classes.
750         """
751
752     pass
```

El segundo archivo es el programa que configura las llamadas a las funciones del objeto `DetectionTrackingPipeline`. Este programa se encarga de inicializar el objeto según el tipo de segmentación elegida, ya sea por hilos, procesos, multiproceso o memoria compartida.

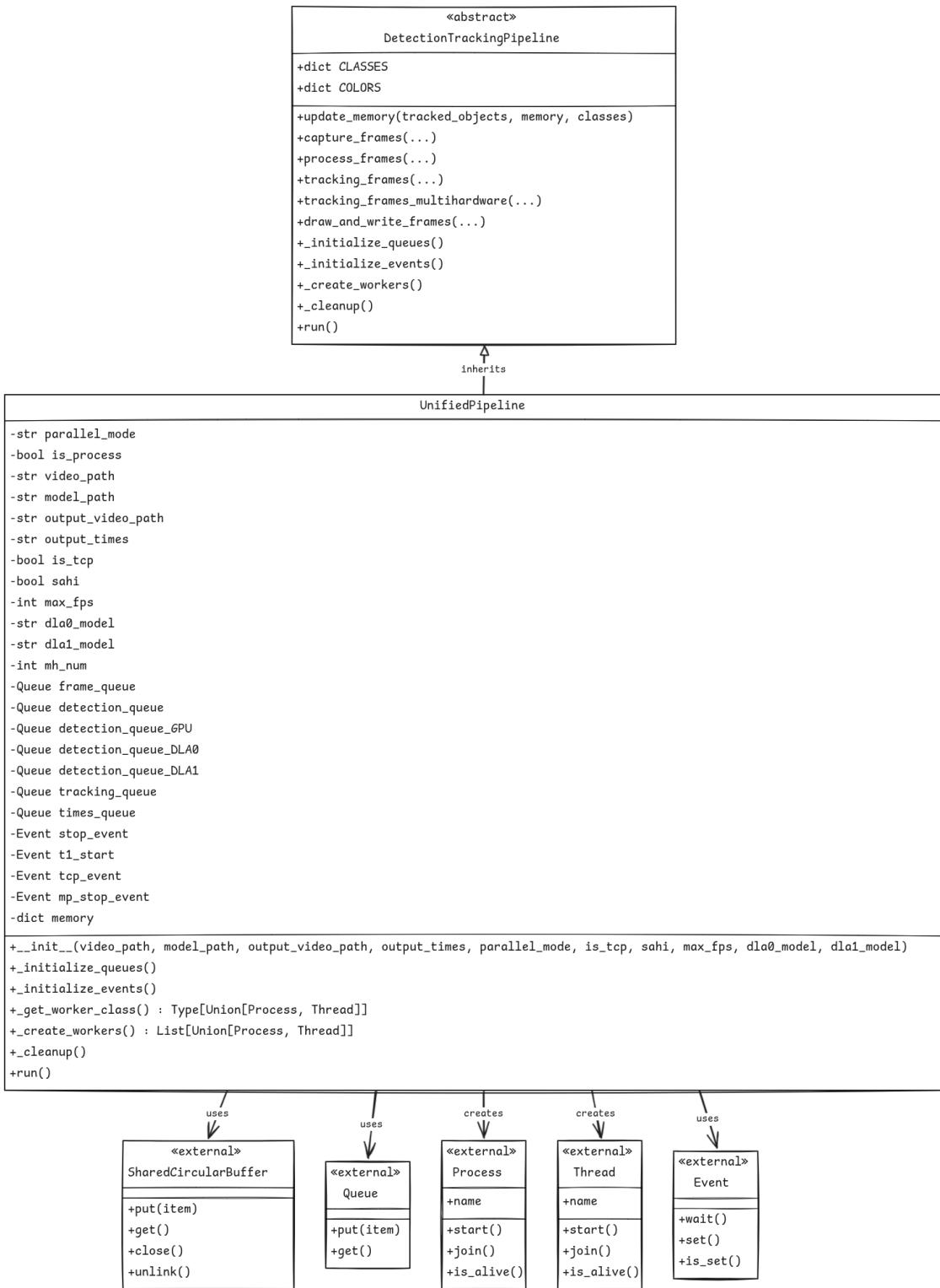


Figura B.2: Diagrama de clases del programa principal.

La figura B.2 muestra el diagrama de clases de objeto `UnifiedPipeline`. Esta hereda del objeto `DetectionTrackingPipeline` y añade la funcionalidad de inicializar el objeto según el tipo de segmentación elegida. La función `run` se encarga de iniciar el proceso de captura, procesamiento, seguimiento y escritura de fotogramas. Dependiendo del tipo de segmentación elegida, se ejecutará en un hilo o proceso diferente.

Listing B.2: unified_pipeline.py

```
1 import cv2
2 import torch.multiprocessing as mp
3 from queue import Queue
4 from classes.shared_circular_buffer import SharedCircularBuffer
5 from detection_tracking_pipeline import DetectionTrackingPipeline
6 import threading
7 import logging
8 from typing import Union, List, Type
9
10
11 class UnifiedPipeline(DetectionTrackingPipeline):
12     """Unified pipeline that supports different parallelization
13     strategies.
14
15     This class inherits from 'DetectionTrackingPipeline' and
16     provides a concrete
17     implementation that allows configuring the pipeline to run in
18     different modes:
19     - 'mp.hardware': Uses multiple processes and shared memory
20     queues,
21             optimized for scenarios with multiple hardware
22             accelerators
23             (e.g. GPU, DLA0, DLA1).
24     - 'threads': Uses threads for concurrency within the same
25     process.
26     - 'mp.shared_memory': Uses multiple processes with shared
27     memory queues
28             (SharedCircularBuffer).
29     - Default (any other string): Uses multiple processes with
30     standard
31             'torch.multiprocessing' queues.
32
33     """
34
35     def __init__(
36         self,
37         video_path: str,
38         model_path: str,
39         output_video_path: str,
40         output_times: str,
41         parallel_mode: str,
42         is_tcp: bool = False,
43         sahi: bool = False,
44         max_fps: int = None,
45         dla0_model: str = None,
46         dla1_model: str = None,
47     ):
48         """Initializes the unified pipeline.
49
50         Args:
51             video_path: Path to the input video file.
52             model_path: Path to the main detection model (e.g. for
53                         GPU).
54             output_video_path: Path to save the processed video
55                         with detections.
56             output_times: Path to the CSV file to save processing
57                         times.
58             parallel_mode: Parallelization strategy to use.
59             is_tcp: Boolean to enable TCP communication (e.g. to
60                         notify defects).
```

```

48         sahi: Boolean to enable SAHI (Slice-Aided Hyper
49             Inference).
50         max_fps: Optional. Limits the frames per second of
51             processing.
52         dla0_model: Optional. Path to the model for DLA0
53             accelerator (if 'parallel_mode' is 'mp_hardware').
54         dla1_model: Optional. Path to the model for DLA1
55             accelerator (if 'parallel_mode' is 'mp_hardware').
56         """
57     self.parallel_mode = parallel_mode
58     # Determines if workers will be processes or threads based
59     # on the parallelization mode.
60     self.is_process = parallel_mode != "threads"
61     self.video_path = video_path
62     self.model_path = model_path
63     self.output_video_path = output_video_path
64     self.output_times = output_times
65     self.is_tcp = is_tcp
66     self.sahi = sahi
67     self.max_fps = max_fps
68     self.dla0_model = dla0_model
69     self.dla1_model = dla1_model
70     # mh_num is used to indicate to capture_frames how many ,
71     # None' signals to send
72     # when finishing, so that all frame_queue consumers
73     # terminate.
74     self.mh_num = 1 # Default: one frame consumer (
75         process_frames), if using mp_hardware, it would be 3 (
76         GPU + DLA0 + DLA1)
77
78     self._initialize_queues()
79     self._initialize_events()
80     # Initializes shared memory for object tracking.
81     # This memory is used by 'draw_and_write_frames' and
82     # updated by 'update_memory'.
83     self.memory = {}
84
85     def _initialize_queues(self):
86         """Initializes communication queues between pipeline stages
87         .
88
89         The choice of queue type (standard Queue, mp.Queue,
90             SharedCircularBuffer)
91         and its size is based on 'parallel_mode' and whether 'max_fps'
92         is defined.
93         'SharedCircularBuffer' is used for modes with explicit
94         shared memory.
95         """
96
97         # Queue size: 1 if max_fps is limited (to avoid
98             accumulation), otherwise 10.
99         queue_size = 1 if self.max_fps else 10
100
101         if self.parallel_mode == "mp_hardware":
102             # Multiple detection queues for different hardware (GPU
103                 , DLA0, DLA1)
104             # and a common frame queue. All use
105                 SharedCircularBuffer.
106             logging.info("mp_hardware\u2014mode:\u2014Using\u2014
107                 SharedCircularBuffer\u2014for\u2014all\u2014queues.\")
```

```

89         self.frame_queue = SharedCircularBuffer(
90             queue_size=queue_size, max_item_size=16
91         ) # Arbitrary item size
92         self.detection_queue_GPU = SharedCircularBuffer(
93             queue_size=queue_size, max_item_size=16)
94         self.detection_queue_DLao = SharedCircularBuffer(
95             queue_size=queue_size, max_item_size=16
96         )
97         self.detection_queue_DLao1 = SharedCircularBuffer(
98             queue_size=queue_size, max_item_size=16
99         )
100        self.tracking_queue = SharedCircularBuffer(queue_size=
101            queue_size, max_item_size=16)
102        self.times_queue = SharedCircularBuffer(queue_size=
103            queue_size, max_item_size=16)
104        self.mh_num = 3 # One frame capturer feeds 3 frame
105            processors (GPU, DLao, DLao1)

106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
self.parallel_mode == "threads":
    # Standard 'queue.Queue' queues for communication
    # between threads.
    logging.info("threads mode: Using queue.Queue.")
    self.frame_queue = Queue(maxsize=queue_size)
    self.detection_queue = Queue(maxsize=queue_size)
    self.tracking_queue = Queue(maxsize=queue_size)
    self.times_queue = Queue(maxsize=queue_size)

elif self.parallel_mode == "mp_shared_memory":
    # 'SharedCircularBuffer' queues for multiprocessing
    # with shared memory.
    logging.info("mp_shared_memory mode: Using"
                 " SharedCircularBuffer.")
    self.frame_queue = SharedCircularBuffer(queue_size=
        queue_size, max_item_size=16)
    self.detection_queue = SharedCircularBuffer(queue_size=
        queue_size, max_item_size=16)
    self.tracking_queue = SharedCircularBuffer(queue_size=
        queue_size, max_item_size=16)
    self.times_queue = SharedCircularBuffer(queue_size=
        queue_size, max_item_size=16)

else: # Default mode: standard multiprocessing
    # 'mp.Queue' queues from 'torch.multiprocessing'.
    # If max_fps is defined, the frame queue has size 1 to
    # process frame by frame.
    logging.info("Standard multiprocessing mode: Using mp."
                 " Queue.")
    self.frame_queue = mp.Queue(maxsize=1) if self.max_fps
        else mp.Queue(maxsize=queue_size)
    self.detection_queue = mp.Queue(maxsize=queue_size)
    self.tracking_queue = mp.Queue(maxsize=queue_size)
    self.times_queue = mp.Queue(maxsize=queue_size)

def _initialize_events(self):
    """Initializes synchronization events.

    All events are from 'torch.multiprocessing.Event'
    regardless of mode,
    since they can be shared between processes if necessary,

```

```
133     and also work correctly with threads.
134     - 'stop_event': Signal to stop all pipeline workers.
135     - 't1_start': Signal to synchronize the start of timing and
136                   certain operations after model initialization
137
138     - 'tcp_event': Signal to synchronize TCP operations.
139     - 'mp_stop_event': Signal for workers to wait before
140                       exiting,
141                           allowing an orderly shutdown.
142
143     # Multiprocessing events are used for all modes for
144     # consistency
145     # and because they work for both processes and threads.
146     self.stop_event = mp.Event()
147     self.t1_start = mp.Event()
148     self.tcp_event = mp.Event()
149     self.mp_stop_event = (
150         mp.Event()
151     ) # Used for workers to wait before os._exit if they are
152       processes
153
154     def _get_worker_class(self) -> Type[Union[mp.Process, threading.
155     Thread]]:
156         """Determines the base class for workers (process or thread
157         )."""
158
159         Returns:
160             The 'threading.Thread' class if 'parallel_mode' is 'threads',
161             otherwise, 'torch.multiprocessing.Process'.
162
163         if self.parallel_mode == "threads":
164             return threading.Thread
165             return mp.Process
166
167         def _create_workers(self) -> List[Union[mp.Process, threading.
168     Thread]]:
169             """Creates and initializes the list of workers (processes
170             or threads) for the pipeline.
171
172             Each worker is an instance of the class returned by
173             '_get_worker_class()'.
174             The configuration of workers (target functions and
175             arguments) depends
176             on the 'parallel_mode' and whether SAHI or multiple
177             hardware is used.
178
179             Returns:
180                 A list of Worker objects (Process or Thread) ready to
181                 be started.
182
183             Worker = self._get_worker_class()
184             workers: List[Union[mp.Process, threading.Thread]] = []
185
186             # 1. Frame Capture Worker (common to all modes)
187             # This worker reads frames from the video and puts them in
188             # 'frame_queue'.
189             workers.append(
190                 Worker(
191
```

```

178         name="CaptureWorker",
179         target=self.capture_frames,
180         args=(
181             self.video_path,
182             self.frame_queue,
183             self.t1_start, # Event to start capture after
184             others are ready
185             self.stop_event, # Event to stop capture
186             self.tcp_event, # Event for TCP
187             synchronization (if is_tcp)
188             self.is_tcp,
189             self.mp_stop_event, # Event to wait before
190             exiting (if process)
191             self.mh_num, # Number of frame_queue consumers
192             self.is_process, # True if the worker is a
193             process
194             self.max_fps, # FPS limit for capture
195         ),
196     )
197 )
198
199 # 2. Frame Processing Workers (Detection)
200 if self.parallel_mode == "mp_hardware":
201     # Multiple detection workers, one for each specified
202     # model/hardware.
203     # Each consumes from 'frame_queue' and produces to its
204     # 'detection_queue_*'.
205     hardware_setups = [
206         (self.model_path, self.detection_queue_GPU, "GPU"),
207         (self.dla0_model, self.detection_queue_DLA0, "DLA0"
208             ),
209         (self.dla1_model, self.detection_queue_DLA1, "DLA1"
210             ),
211     ]
212     for model_p, detection_q, hw_name in hardware_setups:
213         if model_p: # Only create the worker if a model
214             path was provided
215             workers.append(
216                 Worker(
217                     name=f"ProcessWorker_{hw_name}",
218                     target=self.process_frames_sahi if self
219                     .sahi else self.process_frames,
220                     args=(
221                         self.frame_queue,
222                         detection_q,
223                         model_p,
224                         self.t1_start, # Signals when the
225                         model is ready
226                         self.mp_stop_event,
227                         self.is_process,
228                     ),
229                 )
230             )
231
232             # Multi-Hardware specific Tracking Worker
233             # Consumes from all 'detection_queue_*' and produces to
234             # 'tracking_queue'.
235             workers.append(
236                 Worker(

```

```
225         name="TrackingWorker_MH",
226         target=self.tracking_frames_multihardware,
227         args=(
228             self.detection_queue_GPU,
229             self.detection_queue_DL40,
230             self.detection_queue_DL41,
231             self.tracking_queue,
232             self.mp_stop_event,
233             self.is_process,
234         ),
235     )
236 )
237 else:
238     # Single Frame Processing Worker (Detection)
239     # Consumes from 'frame_queue' and produces to 'detection_queue'.
240     workers.append(
241         Worker(
242             name="ProcessWorker",
243             target=self.process_frames_sahi if self.sahi
244             else self.process_frames,
245             args=(
246                 self.frame_queue,
247                 self.detection_queue,
248                 self.model_path,
249                 self.t1_start,
250                 self.mp_stop_event,
251                 self.is_process,
252             ),
253         )
254     )
255
256     # Standard Tracking Worker
257     # Consumes from 'detection_queue' and produces to 'tracking_queue'.
258     workers.append(
259         Worker(
260             name="TrackingWorker",
261             target=self.tracking_frames,
262             args=(
263                 self.detection_queue,
264                 self.tracking_queue,
265                 self.mp_stop_event,
266                 self.is_process,
267             ),
268         )
269     )
270
271     # 3. Workers common to all modes (Drawing, CSV Writing,
272     # Hardware Usage)
273
274     # Worker to draw detections/tracking on frames and write
275     # the output video.
276     # Consumes from 'tracking_queue' and can interact with TCP.
277     workers.append(
278         Worker(
279             name="DrawWriteWorker",
280             target=self.draw_and_write_frames,
281             args=
```



```

326             queue_buffer.unlink()
327             logging.debug(f"Buffer {i} cleaned.")
328         else:
329             logging.warning(
330                 f"Expected SharedCircularBuffer in "
331                 "cleanup, got {type(queue_buffer)}"
332             )
333     except Exception as e:
334         logging.error(f"Error cleaning buffer {i}: {e}")
335     else:
336         logging.info(
337             f"{self.parallel_mode} mode: No manual cleanup "
338             "required for standard queues."
339         )
340     logging.info("Resource cleanup finished.")

341 def run(self):
342     """Runs the unified pipeline.
343
344     This method orchestrates the creation, startup and
345     termination of workers.
346     Measures the total processing time from when 't1_start' is
347     activated
348     (usually after models are ready) until 'stop_event'
349     is activated (usually by 'draw_and_write_frames' when
350     finishing processing).
351     Finally, performs resource cleanup and waits for thread
352     termination
353     if that is the parallelization mode.
354     """
355     logging.info(f"Starting pipeline in mode: {self.
356                  parallel_mode}")

357     # Create and start all workers (processes or threads)
358     workers = self._create_workers()
359     for worker in workers:
360         logging.info(f"Starting worker: {worker.name}")
361         worker.start()

362     # Wait for the initialization stage (e.g. model loading in
363     # process_frames)
364     # to activate the t1_start event. This marks the actual
365     # start of measurable processing.
366     logging.info("Waiting for t1_start signal (models ready/
367                  measurement start)...")
368     self.t1_start.wait()
369     logging.info("t1_start signal received.")

370     t_start_processing = cv2.getTickCount()

371     # Wait for the pipeline to complete its main task.
372     # 'stop_event' is usually activated by the last worker in
373     # the chain
374     # (draw_and_write_frames) when there are no more frames to
375     # process.
376     logging.info("Pipeline running. Waiting for stop event/
377                  signal (end of processing)...")
378     self.stop_event.wait()

```

```

371     logging.info("stop_event signal received.")
372
373     t_end_processing = cv2.getTickCount()
374
375     # Clean up resources (e.g. shared memory queues)
376     # It's important to do this before processes terminate
377     # completely.
378     self._cleanup()
379
380     # Calculate and display performance statistics
381     tiempo_total_segundos = (t_end_processing -
382                               t_start_processing) / cv2.getTickFrequency()
383
384     # Get the total number of frames from the video to
385     # calculate average FPS.
386     # I assume that get_total_frames is a method (possibly
387     # static or instance)
388     # that reads video metadata.
389
390     print(f"Total processing time: {tiempo_total_segundos:.2f} seconds")
391
392     # Wait for all workers to finish, especially important for
393     # threads.
394     # For processes, mp_stop_event and os._exit() handle their
395     # termination.
396     if self.parallel_mode == "threads":
397         logging.info("Waiting for thread termination...")
398         for worker in workers:
399             if worker.is_alive(): # Only join if the thread is
400                 still alive
401                 logging.info(f"Waiting for {worker.name}...")
402                 worker.join(timeout=10) # Add a timeout to
403                 # avoid indefinite blocking
404                 if worker.is_alive():
405                     logging.warning(f"Thread {worker.name} did not finish after timeout.")
406                 else:
407                     logging.info(f"Thread {worker.name} finished.")
408
409     # Signal processes that they can terminate (if they are
410     # waiting for mp_stop_event)
411     if self.is_process:
412         self.mp_stop_event.set()
413
414     print("Pipeline finished.")

```

El tercer archivo define ‘TrackerWrapper’, una clase que encapsula la funcionalidad del algoritmo de seguimiento BYTETrack. Su propósito es simplificar la integración del algoritmo de seguimiento en el pipeline principal.

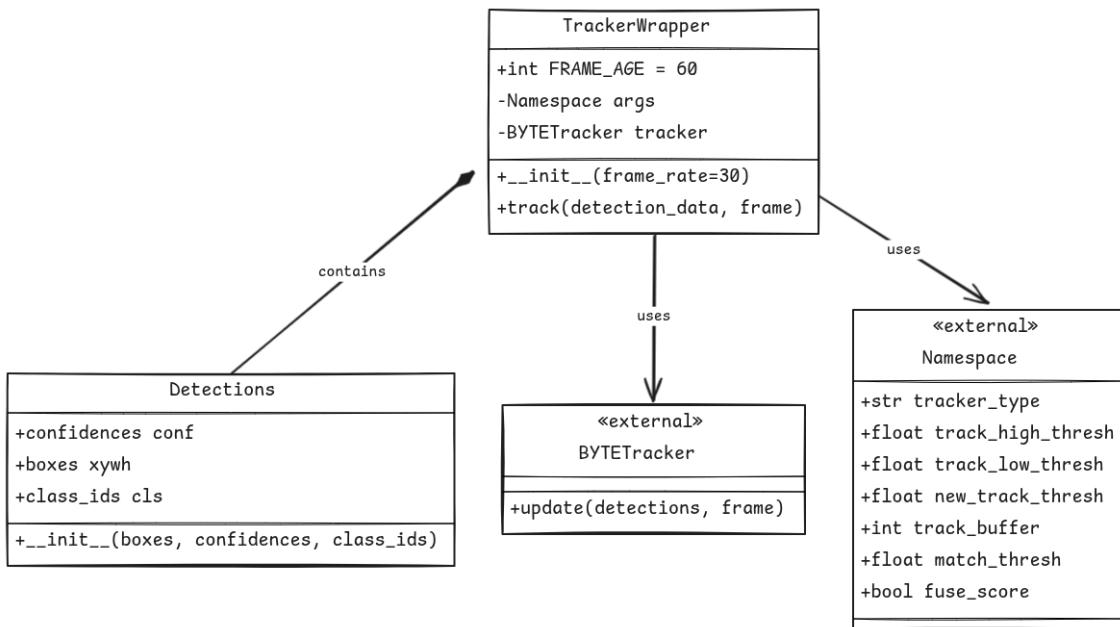


Figura B.3: Diagrama de clases del objeto TrackerWrapper.

La Figura B.3 presenta el diagrama de clases de ‘TrackerWrapper’. A partir de ‘BaseTracker’, esta clase se encarga de inicializar el algoritmo BYTETrack y provee la función ‘track’ para procesar cada fotograma y retornar las trayectorias de los objetos detectados. En esencia, ‘TrackerWrapper’ abstrae la complejidad de la interacción directa con BYTETrack, ofreciendo una interfaz limpia y consistente al resto del sistema.

Listing B.3: tracker_wrapper.py

```

1  from argparse import Namespace
2  from ultralytics.trackers.byte_tracker import BYTETracker # type:
3      ignore
4
5
6
7  class TrackerWrapper:
8      FRAME_AGE = 60
9
10     def __init__(self, frame_rate=30):
11         self.args = Namespace(
12             tracker_type="bytetrack",
13             track_high_thresh=0.25,
14             track_low_thresh=0.1,
15             new_track_thresh=0.25,
16             track_buffer=self.FRAME_AGE,
17             match_thresh=0.8,
18             fuse_score=True,
19         )
20         self.tracker = BYTETracker(self.args, frame_rate=frame_rate
21             )
22
23     class Detections:
24         def __init__(self, boxes, confidences, class_ids):
25             self.conf = confidences
26             self.xywh = boxes
27             self.cls = class_ids
  
```

```
27
28     def track(self, detection_data, frame):
29         detections = self.Detections(
30             detection_data.xywh.numpy(),
31             detection_data.conf.numpy(),
32             detection_data.cls.numpy().astype(int),
33         )
34         return self.tracker.update(detections, frame)
```

El cuarto archivo es el objeto ‘SharedCircularBuffer’, que implementa un buffer circular con memoria compartida. Este buffer permite crear la abstracción de una cola de mensajes que puede ser utilizada por diferentes procesos que comparten memoria.

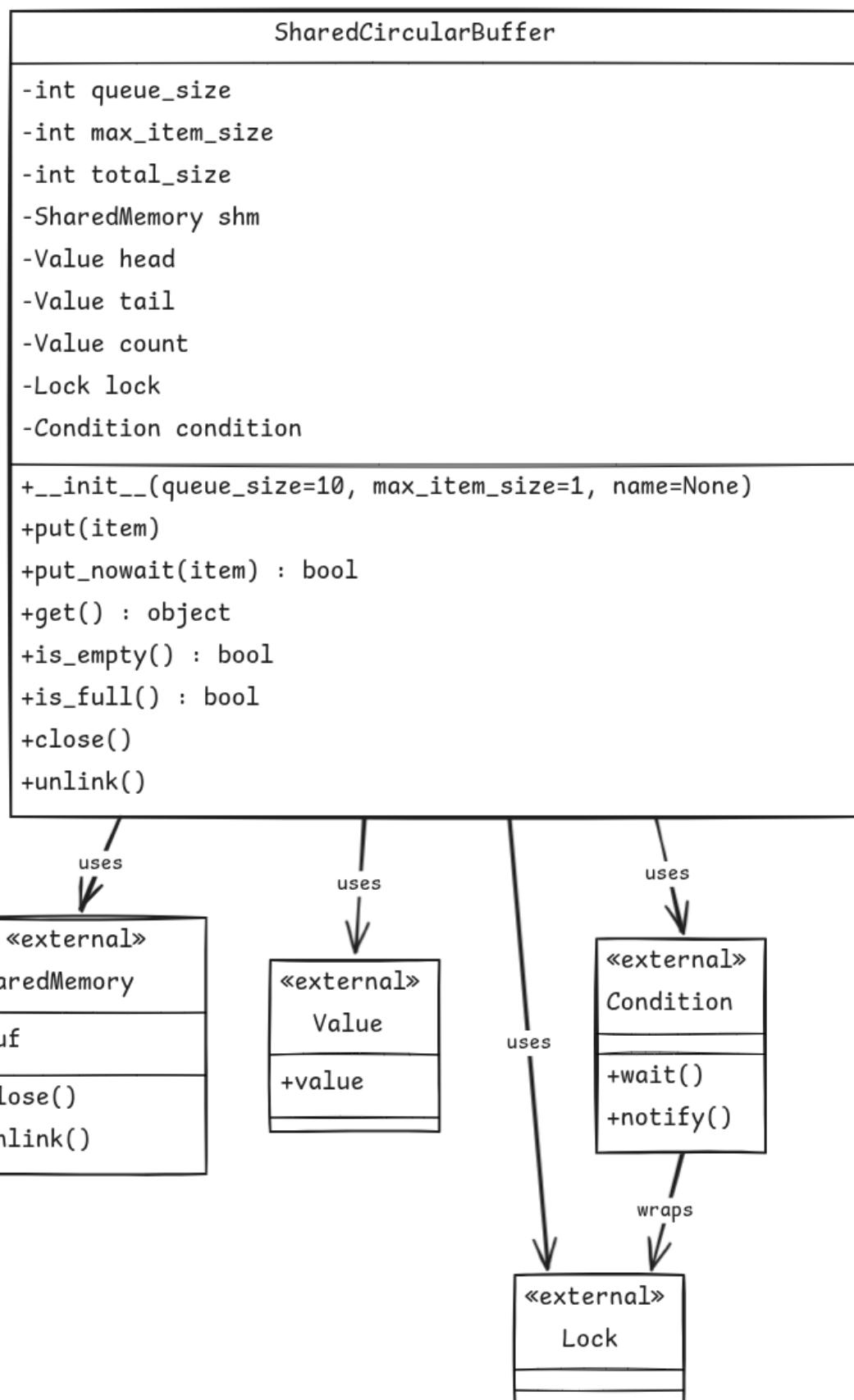


Figura B.4: Diagrama de clases del objeto SharedCircularBuffer.

La Figura B.4 muestra el diagrama de clases del objeto 'SharedCircularBuffer'. Este objeto implementa las funciones para inicializar el buffer, escribir y leer mensajes, y gestionar la memoria compartida. El buffer circular permite que los procesos que comparten memoria puedan comunicarse de manera eficiente, evitando bloqueos y garantizando un acceso seguro a los datos.

Listing B.4: shared_circular_buffer.py

```
1 import pickle
2 import numpy as np
3 from multiprocessing import shared_memory, Value, Lock, Condition
4
5
6 class SharedCircularBuffer:
7     def __init__(self, queue_size=10, max_item_size=1, name=None):
8         """
9             Initializes a circular buffer in shared memory.
10
11             :param queue_size: Maximum number of elements in the queue.
12             :param max_item_size: Maximum size in MB of each element.
13             :param name: Name of the shared memory (None to create a
14                         new one).
15         """
16
17         self.queue_size = queue_size
18
19         if max_item_size not in (1, 2, 4, 8, 16, 32, 64, 128, 256,
20             512):
21             raise ValueError(
22                 "The maximum item size must be 1, 2, 4, 8, 16, 32, 64, 128, 256 or 512 MB."
23             )
24
25         self.max_item_size = max_item_size * 1024 * 1024
26         self.total_size = queue_size * self.max_item_size
27
28         if name:
29             self.shm = shared_memory.SharedMemory(name=name)
30         else:
31             self.shm = shared_memory.SharedMemory(create=True, size
32                                         =self.total_size)
33
34         self.head = Value("i", 0) # Read index
35         self.tail = Value("i", 0) # Write index
36         self.count = Value("i", 0) # Number of elements in the
37                                     queue
38         self.lock = Lock()
39         self.condition = Condition(self.lock) # For
40                                     synchronization
41
42     def put(self, item):
43         """Adds an item to the queue in shared memory without
44             overwriting unread elements."""
45         if isinstance(item, np.ndarray):
46             reshaped_item = item.reshape(-1)
47             item_data = {"data": reshaped_item, "shape": item.shape
48                         }
49         else:
50             item_data = {"data": item, "shape": None}
51
52         with self.lock:
53             while self.count.value == self.queue_size:
54                 self.condition.wait()
55
56             self.tail.value += 1
57             self.shm[slice(self.tail.value, self.head.value)] = item_data
58
59             if self.tail.value == self.queue_size:
60                 self.tail.value = 0
61
62             self.count.value += 1
63
64             self.condition.notify()
65
66     def get(self):
67         """Gets an item from the queue in shared memory without
68             overwriting unread elements.
69
70             Returns:
71                 item: The item at the head of the queue.
72                 shape: The original shape of the item.
73
74             Raises:
75                 IndexError: If the queue is empty.
76         """
77         with self.lock:
78             while self.count.value == 0:
79                 self.condition.wait()
80
81             self.head.value += 1
82             item_data = self.shm[slice(self.head.value, self.tail.value)]
83
84             if self.head.value == self.queue_size:
85                 self.head.value = 0
86
87             self.count.value -= 1
88
89             self.condition.notify()
90
91         return item_data["data"], item_data["shape"]
```

```
44     data_bytes = pickle.dumps(item_data)
45
46     if len(data_bytes) > self.max_item_size:
47         raise ValueError("The item is too large for the queue.")
48
49     with self.condition:
50         while self.count.value == self.queue_size:
51             # The queue is full, wait until there is space
52             self.condition.wait()
53
54         pos = (self.tail.value % self.queue_size) * self.
55             max_item_size
56         self.shm.buf[pos : pos + len(data_bytes)] = memoryview(
57             data_bytes)
58
59         self.tail.value = (self.tail.value + 1) % self.
60             queue_size
61         self.count.value += 1
62
63         self.condition.notify() # Notify 'get' that a new
64             element is available
65
66     def put_nowait(self, item):
67         """Adds an item to the queue in shared memory. If the queue
68             is full, returns False without adding the item."""
69
70         if isinstance(item, np.ndarray):
71             reshaped_item = item.reshape(-1)
72             item_data = {"data": reshaped_item, "shape": item.shape
73                         }
74         else:
75             item_data = {"data": item, "shape": None}
76
77         data_bytes = pickle.dumps(item_data)
78
79         if len(data_bytes) > self.max_item_size:
80             raise ValueError("The item is too large for the queue.")
81
82         with self.condition:
83             if self.count.value >= self.queue_size:
84                 return False # Queue is full, cannot add the item
85
86             pos = (self.tail.value % self.queue_size) * self.
87                 max_item_size
88             self.shm.buf[pos : pos + len(data_bytes)] = memoryview(
89                 data_bytes)
90
91             self.tail.value = (self.tail.value + 1) % self.
92                 queue_size
93             self.count.value += 1
94
95             self.condition.notify() # Notify 'get' that a new
96                 element is available
97             return True
98
99
100
```

```

91     def get(self):
92         """Extracts an item from the queue in shared memory,
93             waiting if it is empty."""
94         with self.condition:
95             while self.count.value == 0: # Wait if the queue is
96                 empty
97                 self.condition.wait()
98
99             pos = (self.head.value % self.queue_size) * self.
100                max_item_size
101            data_bytes = bytes(self.shm.buf[pos : pos + self.
102                max_item_size])
103
104            self.head.value = (self.head.value + 1) % self.
105                queue_size
106            self.count.value -= 1
107
108            self.condition.notify() # Notify 'put' that space is
109                available
110
111            item_data = pickle.loads(data_bytes)
112
113        if item_data["shape"] is not None:
114            return np.array(item_data["data"]).reshape(item_data["
115                shape"])
116
117        return item_data["data"]
118
119    def is_empty(self):
120        """Returns True if the queue is empty."""
121        with self.lock:
122            return self.count.value == 0
123
124    def is_full(self):
125        """Returns True if the queue is full."""
126        with self.lock:
127            return self.count.value == self.queue_size
128
129    def close(self):
130        """Closes the shared memory."""
131        self.shm.close()
132
133    def unlink(self):
134        """Releases the shared memory (should only be called once).
135            """
136        self.shm.unlink()

```

Por último, el quinto archivo es el programa que se encarga de proporcionar los argumentos necesarios para ejecutar el programa principal. Este programa permite al usuario elegir todas las opciones de configuración del pipeline, como el tipo de segmentación, el modelo de detección, la precisión, el tamaño del modelo, el dispositivo a utilizar y la tasa de fotogramas por segundo.

Listing B.5: inference.py

```

1  """
2  Object Detection and Tracking Inference Pipeline
3

```

```
4 This module provides a comprehensive inference pipeline for object
5     detection and tracking
6 using YOLO models with support for different hardware configurations,
7     parallelization
8 modes.
9
10 Features:
11 - Multiple YOLO model variants (YOLOv8, YOLOv11, etc.)
12 - Hardware acceleration (GPU, DLA0, DLA1, CPU)
13 - Various precision modes (FP32, FP16, INT8)
14 - Parallelization strategies (threads, multiprocessing, shared memory)
15 - Configurable FPS limits
16 """
17
18 import argparse
19 import os
20 import torch.multiprocessing as mp
21 from unified_pipeline import UnifiedPipeline
22
23 def parse_arguments():
24     """
25         Parse command line arguments for the inference pipeline.
26
27     Returns:
28         argparse.Namespace: Parsed command line arguments containing:
29             - num_objects: Number of objects to count in video
30             - model: YOLO model variant to use
31             - precision: Model precision (FP32, FP16, INT8)
32             - hardware: Target hardware (GPU, DLA0, DLA1, CPU)
33             - mode: Power mode configuration
34             - tcp: Whether to use TCP communication
35             - version: Dataset version
36             - parallel: Parallelization strategy
37             - max_fps: Maximum FPS limit
38
39     parser = argparse.ArgumentParser(description="Object-detection-and-
40                                     tracking-inference-pipeline")
41
42     # Object counting configuration
43     parser.add_argument(
44         "--num_objects",
45         default="free",
46         type=str,
47         choices=["free", "variable", "0", "18", "40", "48", "60", "70",
48             "88", "176"],
49         help="Number of objects to count in the video, default=free",
50     )
51
52     # Model selection
53     parser.add_argument(
54         "--model",
55         default="yolo11n",
56         type=str,
57         choices=[
58             "yolo11n",
59             "yolo11s",
60             "yolo11m",
61             "yolo11l",
62             "yolo11x",
63             "yolov5nu",
64             "yolov5mu",
65             "yolov8n",
66             "yolov8s",
67         ],
68     )
69
70     # Video input configuration
71     parser.add_argument(
72         "--video",
73         default=None,
74         type=str,
75         help="Path to the video file to process, default=None"
76     )
77
78     # Output configuration
79     parser.add_argument(
80         "--output",
81         default=None,
82         type=str,
83         help="Path to the output file for results, default=None"
84     )
85
86     # Logging and metrics
87     parser.add_argument(
88         "--log_level",
89         default="INFO",
90         type=str,
91         choices=[],
92         help="Log level (INFO, DEBUG, etc.)"
93     )
94
95     # Performance monitoring
96     parser.add_argument(
97         "--max_fps",
98         default=60,
99         type=int,
100        help="Maximum FPS limit, default=60"
101    )
102
103    # Parallelization
104    parser.add_argument(
105        "--parallel",
106        default="multiprocessing",
107        type=str,
108        choices=[],
109        help="Parallelization strategy (multiprocessing, threads, shared
110             memory), default=multiprocessing"
111    )
112
113    # Hardware configuration
114    parser.add_argument(
115        "--hardware",
116        default="auto",
117        type=str,
118        choices=[],
119        help="Target hardware (GPU, DLA0, DLA1, CPU), default=auto"
120    )
121
122    # Precision configuration
123    parser.add_argument(
124        "--precision",
125        default="fp32",
126        type=str,
127        choices=[],
128        help="Model precision (FP32, FP16, INT8), default=fp32"
129    )
130
131    # Mode configuration
132    parser.add_argument(
133        "--mode",
134        default="normal",
135        type=str,
136        choices=[],
137        help="Power mode configuration (normal, power, performance),
138             default=normal"
139    )
140
141    # TCP configuration
142    parser.add_argument(
143        "--tcp",
144        default=False,
145        type=bool,
146        help="Whether to use TCP communication, default=False"
147    )
148
149    # Version configuration
150    parser.add_argument(
151        "--version",
152        default="latest",
153        type=str,
154        choices=[],
155        help="Dataset version (latest, v1, v2, v3, v4), default=latest"
156    )
157
158    # Parallelization strategy configuration
159    parser.add_argument(
160        "--parallel",
161        default="multiprocessing",
162        type=str,
163        choices=[],
164        help="Parallelization strategy (multiprocessing, threads, shared
165             memory), default=multiprocessing"
166    )
167
168    # Max FPS limit configuration
169    parser.add_argument(
170        "--max_fps",
171        default=60,
172        type=int,
173        help="Maximum FPS limit, default=60"
174    )
175
176    # Log level configuration
177    parser.add_argument(
178        "--log_level",
179        default="INFO",
180        type=str,
181        choices=[],
182        help="Log level (INFO, DEBUG, etc.)"
183    )
184
185    # Shared memory configuration
186    parser.add_argument(
187        "--shared_memory",
188        default=False,
189        type=bool,
190        help="Use shared memory for parallel processing, default=False"
191    )
192
193    # Power mode configuration
194    parser.add_argument(
195        "--mode",
196        default="normal",
197        type=str,
198        choices=[],
199        help="Power mode configuration (normal, power, performance),
200             default=normal"
201    )
202
203    # GPU configuration
204    parser.add_argument(
205        "--gpu",
206        default=None,
207        type=int,
208        help="GPU index to use, default=None"
209    )
210
211    # DLA configuration
212    parser.add_argument(
213        "--dla",
214        default=None,
215        type=int,
216        help="DLA index to use, default=None"
217    )
218
219    # CPU configuration
220    parser.add_argument(
221        "--cpu",
222        default=False,
223        type=bool,
224        help="Force CPU usage, default=False"
225    )
226
227    # Model variant configuration
228    parser.add_argument(
229        "--model",
230        default="yolo11n",
231        type=str,
232        choices=[],
233        help="YOLO model variant to use, default=yolo11n"
234    )
235
236    # Precision configuration
237    parser.add_argument(
238        "--precision",
239        default="fp32",
240        type=str,
241        choices=[],
242        help="Model precision (FP32, FP16, INT8), default=fp32"
243    )
244
245    # Mode configuration
246    parser.add_argument(
247        "--mode",
248        default="normal",
249        type=str,
250        choices=[],
251        help="Power mode configuration (normal, power, performance),
252             default=normal"
253    )
254
255    # TCP configuration
256    parser.add_argument(
257        "--tcp",
258        default=False,
259        type=bool,
260        help="Whether to use TCP communication, default=False"
261    )
262
263    # Version configuration
264    parser.add_argument(
265        "--version",
266        default="latest",
267        type=str,
268        choices=[],
269        help="Dataset version (latest, v1, v2, v3, v4), default=latest"
270    )
271
272    # Shared memory configuration
273    parser.add_argument(
274        "--shared_memory",
275        default=False,
276        type=bool,
277        help="Use shared memory for parallel processing, default=False"
278    )
279
280    # Power mode configuration
281    parser.add_argument(
282        "--mode",
283        default="normal",
284        type=str,
285        choices=[],
286        help="Power mode configuration (normal, power, performance),
287             default=normal"
288    )
289
290    # GPU configuration
291    parser.add_argument(
292        "--gpu",
293        default=None,
294        type=int,
295        help="GPU index to use, default=None"
296    )
297
298    # DLA configuration
299    parser.add_argument(
300        "--dla",
301        default=None,
302        type=int,
303        help="DLA index to use, default=None"
304    )
305
306    # CPU configuration
307    parser.add_argument(
308        "--cpu",
309        default=False,
310        type=bool,
311        help="Force CPU usage, default=False"
312    )
313
314    # Model variant configuration
315    parser.add_argument(
316        "--model",
317        default="yolo11n",
318        type=str,
319        choices=[],
320        help="YOLO model variant to use, default=yolo11n"
321    )
322
323    # Precision configuration
324    parser.add_argument(
325        "--precision",
326        default="fp32",
327        type=str,
328        choices=[],
329        help="Model precision (FP32, FP16, INT8), default=fp32"
330    )
331
332    # Mode configuration
333    parser.add_argument(
334        "--mode",
335        default="normal",
336        type=str,
337        choices=[],
338        help="Power mode configuration (normal, power, performance),
339             default=normal"
340    )
341
342    # TCP configuration
343    parser.add_argument(
344        "--tcp",
345        default=False,
346        type=bool,
347        help="Whether to use TCP communication, default=False"
348    )
349
350    # Version configuration
351    parser.add_argument(
352        "--version",
353        default="latest",
354        type=str,
355        choices=[],
356        help="Dataset version (latest, v1, v2, v3, v4), default=latest"
357    )
358
359    # Shared memory configuration
360    parser.add_argument(
361        "--shared_memory",
362        default=False,
363        type=bool,
364        help="Use shared memory for parallel processing, default=False"
365    )
366
367    # Power mode configuration
368    parser.add_argument(
369        "--mode",
370        default="normal",
371        type=str,
372        choices=[],
373        help="Power mode configuration (normal, power, performance),
374             default=normal"
375    )
376
377    # GPU configuration
378    parser.add_argument(
379        "--gpu",
380        default=None,
381        type=int,
382        help="GPU index to use, default=None"
383    )
384
385    # DLA configuration
386    parser.add_argument(
387        "--dla",
388        default=None,
389        type=int,
390        help="DLA index to use, default=None"
391    )
392
393    # CPU configuration
394    parser.add_argument(
395        "--cpu",
396        default=False,
397        type=bool,
398        help="Force CPU usage, default=False"
399    )
400
401    # Model variant configuration
402    parser.add_argument(
403        "--model",
404        default="yolo11n",
405        type=str,
406        choices=[],
407        help="YOLO model variant to use, default=yolo11n"
408    )
409
410    # Precision configuration
411    parser.add_argument(
412        "--precision",
413        default="fp32",
414        type=str,
415        choices=[],
416        help="Model precision (FP32, FP16, INT8), default=fp32"
417    )
418
419    # Mode configuration
420    parser.add_argument(
421        "--mode",
422        default="normal",
423        type=str,
424        choices=[],
425        help="Power mode configuration (normal, power, performance),
426             default=normal"
426    )
427
428    # TCP configuration
429    parser.add_argument(
430        "--tcp",
431        default=False,
432        type=bool,
433        help="Whether to use TCP communication, default=False"
434    )
435
436    # Version configuration
437    parser.add_argument(
438        "--version",
439        default="latest",
440        type=str,
441        choices=[],
442        help="Dataset version (latest, v1, v2, v3, v4), default=latest"
443    )
444
445    # Shared memory configuration
446    parser.add_argument(
447        "--shared_memory",
448        default=False,
449        type=bool,
450        help="Use shared memory for parallel processing, default=False"
451    )
452
453    # Power mode configuration
454    parser.add_argument(
455        "--mode",
456        default="normal",
457        type=str,
458        choices=[],
459        help="Power mode configuration (normal, power, performance),
460             default=normal"
460    )
461
462    # GPU configuration
463    parser.add_argument(
464        "--gpu",
465        default=None,
466        type=int,
467        help="GPU index to use, default=None"
468    )
469
470    # DLA configuration
471    parser.add_argument(
472        "--dla",
473        default=None,
474        type=int,
475        help="DLA index to use, default=None"
476    )
477
478    # CPU configuration
479    parser.add_argument(
480        "--cpu",
481        default=False,
482        type=bool,
483        help="Force CPU usage, default=False"
484    )
485
486    # Model variant configuration
487    parser.add_argument(
488        "--model",
489        default="yolo11n",
490        type=str,
491        choices=[],
492        help="YOLO model variant to use, default=yolo11n"
493    )
494
495    # Precision configuration
496    parser.add_argument(
497        "--precision",
498        default="fp32",
499        type=str,
500        choices=[],
501        help="Model precision (FP32, FP16, INT8), default=fp32"
502    )
503
504    # Mode configuration
505    parser.add_argument(
506        "--mode",
507        default="normal",
508        type=str,
509        choices=[],
510        help="Power mode configuration (normal, power, performance),
511             default=normal"
511    )
512
513    # TCP configuration
514    parser.add_argument(
515        "--tcp",
516        default=False,
517        type=bool,
518        help="Whether to use TCP communication, default=False"
519    )
520
521    # Version configuration
522    parser.add_argument(
523        "--version",
524        default="latest",
525        type=str,
526        choices=[],
527        help="Dataset version (latest, v1, v2, v3, v4), default=latest"
528    )
529
530    # Shared memory configuration
531    parser.add_argument(
532        "--shared_memory",
533        default=False,
534        type=bool,
535        help="Use shared memory for parallel processing, default=False"
536    )
537
538    # Power mode configuration
539    parser.add_argument(
540        "--mode",
541        default="normal",
542        type=str,
543        choices=[],
544        help="Power mode configuration (normal, power, performance),
545             default=normal"
545    )
546
547    # GPU configuration
548    parser.add_argument(
549        "--gpu",
550        default=None,
551        type=int,
552        help="GPU index to use, default=None"
553    )
554
555    # DLA configuration
556    parser.add_argument(
557        "--dla",
558        default=None,
559        type=int,
560        help="DLA index to use, default=None"
561    )
562
563    # CPU configuration
564    parser.add_argument(
565        "--cpu",
566        default=False,
567        type=bool,
568        help="Force CPU usage, default=False"
569    )
570
571    # Model variant configuration
572    parser.add_argument(
573        "--model",
574        default="yolo11n",
575        type=str,
576        choices=[],
577        help="YOLO model variant to use, default=yolo11n"
578    )
579
580    # Precision configuration
581    parser.add_argument(
582        "--precision",
583        default="fp32",
584        type=str,
585        choices=[],
586        help="Model precision (FP32, FP16, INT8), default=fp32"
587    )
588
589    # Mode configuration
590    parser.add_argument(
591        "--mode",
592        default="normal",
593        type=str,
594        choices=[],
595        help="Power mode configuration (normal, power, performance),
596             default=normal"
596    )
597
598    # TCP configuration
599    parser.add_argument(
600        "--tcp",
601        default=False,
602        type=bool,
603        help="Whether to use TCP communication, default=False"
604    )
605
606    # Version configuration
607    parser.add_argument(
608        "--version",
609        default="latest",
610        type=str,
611        choices=[],
612        help="Dataset version (latest, v1, v2, v3, v4), default=latest"
613    )
614
615    # Shared memory configuration
616    parser.add_argument(
617        "--shared_memory",
618        default=False,
619        type=bool,
620        help="Use shared memory for parallel processing, default=False"
621    )
622
623    # Power mode configuration
624    parser.add_argument(
625        "--mode",
626        default="normal",
627        type=str,
628        choices=[],
629        help="Power mode configuration (normal, power, performance),
630             default=normal"
630    )
631
632    # GPU configuration
633    parser.add_argument(
634        "--gpu",
635        default=None,
636        type=int,
637        help="GPU index to use, default=None"
638    )
639
640    # DLA configuration
641    parser.add_argument(
642        "--dla",
643        default=None,
644        type=int,
645        help="DLA index to use, default=None"
646    )
647
648    # CPU configuration
649    parser.add_argument(
650        "--cpu",
651        default=False,
652        type=bool,
653        help="Force CPU usage, default=False"
654    )
655
656    # Model variant configuration
657    parser.add_argument(
658        "--model",
659        default="yolo11n",
660        type=str,
661        choices=[],
662        help="YOLO model variant to use, default=yolo11n"
663    )
664
665    # Precision configuration
666    parser.add_argument(
667        "--precision",
668        default="fp32",
669        type=str,
670        choices=[],
671        help="Model precision (FP32, FP16, INT8), default=fp32"
672    )
673
674    # Mode configuration
675    parser.add_argument(
676        "--mode",
677        default="normal",
678        type=str,
679        choices=[],
680        help="Power mode configuration (normal, power, performance),
681             default=normal"
681    )
682
683    # TCP configuration
684    parser.add_argument(
685        "--tcp",
686        default=False,
687        type=bool,
688        help="Whether to use TCP communication, default=False"
689    )
690
691    # Version configuration
692    parser.add_argument(
693        "--version",
694        default="latest",
695        type=str,
696        choices=[],
697        help="Dataset version (latest, v1, v2, v3, v4), default=latest"
698    )
699
700    # Shared memory configuration
701    parser.add_argument(
702        "--shared_memory",
703        default=False,
704        type=bool,
705        help="Use shared memory for parallel processing, default=False"
706    )
707
708    # Power mode configuration
709    parser.add_argument(
710        "--mode",
711        default="normal",
712        type=str,
713        choices=[],
714        help="Power mode configuration (normal, power, performance),
715             default=normal"
715    )
716
717    # GPU configuration
718    parser.add_argument(
719        "--gpu",
720        default=None,
721        type=int,
722        help="GPU index to use, default=None"
723    )
724
725    # DLA configuration
726    parser.add_argument(
727        "--dla",
728        default=None,
729        type=int,
730        help="DLA index to use, default=None"
731    )
732
733    # CPU configuration
734    parser.add_argument(
735        "--cpu",
736        default=False,
737        type=bool,
738        help="Force CPU usage, default=False"
739    )
740
741    # Model variant configuration
742    parser.add_argument(
743        "--model",
744        default="yolo11n",
745        type=str,
746        choices=[],
747        help="YOLO model variant to use, default=yolo11n"
748    )
749
750    # Precision configuration
751    parser.add_argument(
752        "--precision",
753        default="fp32",
754        type=str,
755        choices=[],
756        help="Model precision (FP32, FP16, INT8), default=fp32"
757    )
758
759    # Mode configuration
760    parser.add_argument(
761        "--mode",
762        default="normal",
763        type=str,
764        choices=[],
765        help="Power mode configuration (normal, power, performance),
766             default=normal"
766    )
767
768    # TCP configuration
769    parser.add_argument(
770        "--tcp",
771        default=False,
772        type=bool,
773        help="Whether to use TCP communication, default=False"
774    )
775
776    # Version configuration
777    parser.add_argument(
778        "--version",
779        default="latest",
780        type=str,
781        choices=[],
782        help="Dataset version (latest, v1, v2, v3, v4), default=latest"
783    )
784
785    # Shared memory configuration
786    parser.add_argument(
787        "--shared_memory",
788        default=False,
789        type=bool,
790        help="Use shared memory for parallel processing, default=False"
791    )
792
793    # Power mode configuration
794    parser.add_argument(
795        "--mode",
796        default="normal",
797        type=str,
798        choices=[],
799        help="Power mode configuration (normal, power, performance),
800             default=normal"
800    )
801
802    # GPU configuration
803    parser.add_argument(
804        "--gpu",
805        default=None,
806        type=int,
807        help="GPU index to use, default=None"
808    )
809
810    # DLA configuration
811    parser.add_argument(
812        "--dla",
813        default=None,
814        type=int,
815        help="DLA index to use, default=None"
816    )
817
818    # CPU configuration
819    parser.add_argument(
820        "--cpu",
821        default=False,
822        type=bool,
823        help="Force CPU usage, default=False"
824    )
825
826    # Model variant configuration
827    parser.add_argument(
828        "--model",
829        default="yolo11n",
830        type=str,
831        choices=[],
832        help="YOLO model variant to use, default=yolo11n"
833    )
834
835    # Precision configuration
836    parser.add_argument(
837        "--precision",
838        default="fp32",
839        type=str,
840        choices=[],
841        help="Model precision (FP32, FP16, INT8), default=fp32"
842    )
843
844    # Mode configuration
845    parser.add_argument(
846        "--mode",
847        default="normal",
848        type=str,
849        choices=[],
850        help="Power mode configuration (normal, power, performance),
851             default=normal"
851    )
852
853    # TCP configuration
854    parser.add_argument(
855        "--tcp",
856        default=False,
857        type=bool,
858        help="Whether to use TCP communication, default=False"
859    )
860
861    # Version configuration
862    parser.add_argument(
863        "--version",
864        default="latest",
865        type=str,
866        choices=[],
867        help="Dataset version (latest, v1, v2, v3, v4), default=latest"
868    )
869
870    # Shared memory configuration
871    parser.add_argument(
872        "--shared_memory",
873        default=False,
874        type=bool,
875        help="Use shared memory for parallel processing, default=False"
876    )
877
878    # Power mode configuration
879    parser.add_argument(
880        "--mode",
881        default="normal",
882        type=str,
883        choices=[],
884        help="Power mode configuration (normal, power, performance),
885             default=normal"
885    )
886
887    # GPU configuration
888    parser.add_argument(
889        "--gpu",
890        default=None,
891        type=int,
892        help="GPU index to use, default=None"
893    )
894
895    # DLA configuration
896    parser.add_argument(
897        "--dla",
898        default=None,
899        type=int,
900        help="DLA index to use, default=None"
901    )
902
903    # CPU configuration
904    parser.add_argument(
905        "--cpu",
906        default=False,
907        type=bool,
908        help="Force CPU usage, default=False"
909    )
910
911    # Model variant configuration
912    parser.add_argument(
913        "--model",
914        default="yolo11n",
915        type=str,
916        choices=[],
917        help="YOLO model variant to use, default=yolo11n"
918    )
919
920    # Precision configuration
921    parser.add_argument(
922        "--precision",
923        default="fp32",
924        type=str,
925        choices=[],
926        help="Model precision (FP32, FP16, INT8), default=fp32"
927    )
928
929    # Mode configuration
930    parser.add_argument(
931        "--mode",
932        default="normal",
933        type=str,
934        choices=[],
935        help="Power mode configuration (normal, power, performance),
936             default=normal"
936    )
937
938    # TCP configuration
939    parser.add_argument(
940        "--tcp",
941        default=False,
942        type=bool,
943        help="Whether to use TCP communication, default=False"
944    )
945
946    # Version configuration
947    parser.add_argument(
948        "--version",
949        default="latest",
950        type=str,
951        choices=[],
952        help="Dataset version (latest, v1, v2, v3, v4), default=latest"
953    )
954
955    # Shared memory configuration
956    parser.add_argument(
957        "--shared_memory",
958        default=False,
959        type=bool,
960        help="Use shared memory for parallel processing, default=False"
961    )
962
963    # Power mode configuration
964    parser.add_argument(
965        "--mode",
966        default="normal",
967        type=str,
968        choices=[],
969        help="Power mode configuration (normal, power, performance),
970             default=normal"
970    )
971
972    # GPU configuration
973    parser.add_argument(
974        "--gpu",
975        default=None,
976        type=int,
977        help="GPU index to use, default=None"
978    )
979
980    # DLA configuration
981    parser.add_argument(
982        "--dla",
983        default=None,
984        type=int,
985        help="DLA index to use, default=None"
986    )
987
988    # CPU configuration
989    parser.add_argument(
990        "--cpu",
991        default=False,
992        type=bool,
993        help="Force CPU usage, default=False"
994    )
995
996    # Model variant configuration
997    parser.add_argument(
998        "--model",
999        default="yolo11n",
1000       type=str,
1001      choices=[],
1002      help="YOLO model variant to use, default=yolo11n"
1003 )
```

```

64     ],
65     help="YOLO model variant to use, default=yolo1in",
66 )
67
68 # Model precision configuration
69 parser.add_argument(
70     "--precision",
71     default="FP16",
72     type=str,
73     choices=["FP32", "FP16", "INT8"],
74     help="Model precision format, default=FP16",
75 )
76
77 # Hardware acceleration target
78 parser.add_argument(
79     "--hardware",
80     default="GPU",
81     type=str,
82     choices=["GPU", "DLA0", "DLA1", "ALL", "CPU"],
83     help="Target hardware for inference, default=GPU",
84 )
85
86 # Power mode configuration
87 parser.add_argument(
88     "--mode",
89     required=True,
90     type=str,
91     choices=["MAXN", "30W", "15W", "10W"],
92     help="Power mode configuration (required)",
93 )
94
95 # Network communication
96 parser.add_argument(
97     "--tcp", default=False, type=bool, help="Enable TCP communication",
98     default=False
99 )
100
101 # Dataset version
102 parser.add_argument(
103     "--version",
104     default="2025_02_24",
105     type=str,
106     choices=["2025_02_24", "2024_11_28"],
107     help="Dataset version to use, default=2025_02_24",
108 )
109
110 # Parallelization strategy
111 parser.add_argument(
112     "--parallel",
113     default="mp_shared_memory",
114     type=str,
115     choices=["threads", "mp", "mp_shared_memory", "mp_hardware"],
116     help="Parallelization strategy, default=mp_shared_memory",
117 )
118
119 # FPS limiting
120 parser.add_argument(
121     "--max_fps",
122     default=None,
123     type=int,
124     help="Maximum FPS limit for processing, default=None (unlimited)",
125 )
126
127     return parser.parse_args()

```

```
127
128
129 def initialize_pipeline(args):
130     """Initialize the detection and tracking pipeline according to
131         parallelization mode."""
132     mode = f"{args.mode}_{mp.cpu_count()}CORE"
133     model_name = args.model
134
135     batch_size = 1
136     batch_suffix = f"_batch{batch_size}" if batch_size > 1 else ""
137
138     # Define model paths for different hardware configurations
139     base_path = (
140         f"../../models/canicas/{args.version}/\u2022\
141         {args.version}_canicas_{model_name}_{args.precision}"
142     )
143
144     GPU_model_path = f"{base_path}_GPU{batch_suffix}.engine"
145     DLA0_model_path = f"{base_path}_DLA0{batch_suffix}.engine"
146     DLA1_model_path = f"{base_path}_DLA1{batch_suffix}.engine"
147     CPU_model_path = f"../../models/canicas/\u2022\
148         {args.version}/{args.version}_canicas_{model_name}.pt"
149
150     model_path = (
151         GPU_model_path
152         if args.hardware == "GPU"
153         else (
154             DLA0_model_path
155             if args.hardware == "DLA0"
156             else DLA1_model_path if args.hardware == "DLA1" else
157                 CPU_model_path
158         )
159     )
160
161     # Select video path based on number of objects
162     if args.num_objects == "free":
163         video_path = "../../datasets_labeled/videos/\u2022\
164         contar_objetos_variable_2min.mp4"
165     else:
166         video_path = f"../../datasets_labeled/videos/\u2022\
167         contar_objetos_{args.num_objects}_2min.mp4"
168
169     output_dir = "../../inference_predictions/custom_tracker"
170
171     os.makedirs(output_dir, exist_ok=True)
172
173     output_video_path = os.path.join(
174         output_dir,
175         f"{args.parallel}_{model_name}_{args.precision}_{args.hardware}_\
176         f'{args.num_objects}_objects_{mode}.mp4",
177     )
178     fps_prefix = f"_-{args.max_fps}fps" if args.max_fps else "maxfps"
179     output_times = (
180         f"{model_name}_{args.precision}_{args.hardware}_\
181         f'{args.num_objects}_objects_{mode}{fps_prefix}'"
182     )
183
184     print("\n\n[PROGRAM] \u2022Selected\u2022options:", args, "\n\n")
185
186     if not args.parallel in ["threads", "mp", "mp_shared_memory",
187         "mp.hardware"]:
188         raise ValueError(
189             "Invalid\u2022parallelization\u2022mode.\u2022Must\u2022be\u2022'threads', '\u2022'mp', '\u2022'\
190             '\u2022'mp_shared_memory'\u2022or\u2022'mp.hardware'.\u2022"
191         )
```

```
188     )
189
190     # Create a unified pipeline instance
191     if args.parallel == "mp.hardware":
192         pipeline = UnifiedPipeline(
193             video_path,
194             GPU_model_path,
195             output_video_path,
196             output_times,
197             args.parallel,
198             is_tcp=args.tcp,
199             max_fps=args.max_fps,
200             dla0_model=DLA0_model_path,
201             dla1_model=DLA1_model_path,
202         )
203     else:
204         pipeline = UnifiedPipeline(
205             video_path,
206             model_path,
207             output_video_path,
208             output_times,
209             args.parallel,
210             is_tcp=args.tcp,
211             max_fps=args.max_fps,
212         )
213
214     return pipeline
215
216
217 def main():
218     args = parse_arguments()
219     detection_tracking_pipeline = initialize_pipeline(args)
220     detection_tracking_pipeline.run()
221
222
223 if __name__ == "__main__":
224     mp.set_start_method("spawn")
225     print(f"[PROGRAM] Number of CPUs: {mp.cpu_count()}")
226     main()
```