



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 desafiament de la detecció automàtica de defectes en objectes que es desplacen en entorns industrials, com ara 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 és un model de xarxa neuronal convolucional (CNN), específicament una variant optimitzada de l'arquitectura YOLO, entrenat 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 encastats, el model s'ha 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 es un modelo de red neuronal convolucional (CNN), específicamente una variante optimizada de la arquitectura YOLO, entrenado 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, el modelo se ha 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 is a convolutional neural network (CNN) model, specifically an optimized variant of the YOLO architecture, trained to accurately identify and locate various types of defects in real-time. To address the computational and energy constraints inherent in embedded systems, the model has been implemented on low-power hardware accelerators from the NVIDIA Jetson series. Advanced optimization techniques have been applied, including conversion to TensorRT and exploration of different numerical precisions, to maximize inference speed (FPS) and minimize energy consumption (Watts) without significantly compromising detection accuracy (mAP).

The work includes a comprehensive analysis of the system's performance 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
<hr/>	
1 Introducción	1
1.1 Motivación	3
1.2 Objetivos	3
1.3 Estructura de la memoria	4
2 Conceptos Previos	5
2.1 Fundamentos y avances en redes neuronales para visión artificial	5
2.1.1 Fundamentos de la inteligencia artificial	5
2.1.2 Tareas fundamentales en visión por computador	6
2.1.3 Arquitectura y funcionamiento de las CNN	7
2.1.4 Detectores de dos etapas	12
2.1.5 Detectores de una etapa	13
2.1.6 Métricas de evaluación	15
2.2 Aceleradores de procesamiento gráfico	17
2.2.1 Limitaciones del hardware tradicional	17
2.2.2 Arquitectura y funcionamiento de las GPUs	18
2.2.3 Serie Jetson: Dispositivos de IA de bajo consumo	20
2.2.4 TensorRT	21
2.3 Seguimiento de objetos en tiempo real	22
2.3.1 Introducción al seguimiento de objetos	23
2.3.2 BYTETrack	24
3 Diseño e implementación de la solución	27
3.1 Análisis del problema	27
3.2 Entrenamiento del modelo	28
3.3 Descripción del sistema	30
3.4 Diseño de las etapas del sistema	30
3.4.1 Captura de imágenes	30
3.4.2 Inferencia	31
3.4.3 Seguimiento	32
3.4.4 Escritura de resultados	32
3.5 Segmentación de las etapas del sistema	33
3.5.1 Secuencial	33
3.5.2 Segmentación basada en hilos	34
3.5.3 Segmentación basada en procesos	35
3.5.4 Segmentación basada en hardware	37
3.5.5 Segmentación basada en procesos con memoria compartida	38
4 Análisis de la solución	41
4.1 Variación de los parámetros	41
4.2 Tipo de segmentación	41

4.3 Talla del modelo	41
4.4 Precisión del modelo	41
4.5 Modo de energía y cores de la CPU	42
4.6 Tamaño de la imagen	42
5 Prueba de concepto	43
5.1 Construcción del entorno	43
5.2 Instalación del entorno	43
6 Conclusiones	45
Bibliografía	47

Apéndices

A Configuración del sistema	49
A.1 Fase de inicialización	49
A.2 Identificación de dispositivos	49
B Objetivos de Desarrollo Sostenible	51

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
2.1	Estructura de un perceptrón multicapa (MLP).	6
2.2	Ejemplo de HOG aplicado a una imagen.	6
2.3	Tareas fundamentales en visión por computador.	7
2.4	Relación entre Machine Learning, Deep Learning, CNN, Computer Vision y Human Vision.	8
2.5	Operación de convolución sobre los pixeles de una imagen.	8
2.6	Proceso de convolución aplicado a una imagen de un autobús.	9
2.7	Ejemplo de operación de max-pooling con una ventana de 2×2 y un paso de 2. Se selecciona el valor máximo de cada región de color.	10
2.8	Ejemplo de una CNN simple.	11
2.9	Proceso de búsqueda selectiva aplicado a una imagen.	13
2.10	Arquitectura de R-CNN.	13
2.11	Ejemplo de detección de objetos utilizando YOLO.	14
2.12	Arquitectura de YOLO	15
2.13	Evolución histórica de las características de los microprocesadores (1970-2020).	17
2.14	Comparativa de arquitecturas CPU y GPU.	19
2.15	Módulos Jetson de NVIDIA.	21
2.16	Ejemplo de flujo de trabajo de optimización con TensorRT.	22
2.17	Diagrama de flujo del filtro de Kalman.	23
2.18	Comparativa de rendimiento de BYTETrack con otros algoritmos de seguimiento.	25
2.19	Ejemplo de detección y seguimiento de objetos utilizando BYTETrack. .	25
3.1	Ejemplo de entorno simulado con canicas.	28
3.2	Ejemplo de anotación de imágenes utilizando CVAT.	29
3.3	Figura del sistema propuesto	30
3.4	Ejemplo de salida del sistema.	33
3.5	Diagrama de flujo del sistema sin segmentar.	34
3.6	Diagrama de flujo del sistema segmentado en hilos.	35
3.7	Diagrama de flujo del sistema segmentado en procesos.	36
3.8	Diagrama de flujo del sistema segmentado.	36
3.9	Diagrama de flujo del sistema segmentado en diferentes unidades de procesamiento.	38
3.10	Ejemplo de buffer circular.	39

Índice de tablas

2.1	Análisis comparativo de las variantes de YOLO11 considerando precisión, velocidad y complejidad computacional.	15
2.2	Comparativa técnica entre diferentes modelos NVIDIA Jetson.	20
4.1	Comparación de modelos en términos de inferencia, consumo de energía y potencia.	41

CAPÍTULO 1

Introducción

Durante los últimos años, la inteligencia artificial 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 inteligencia artificial se refleja en la evolución del interés público en ella, como muestra la Figura 1.1.

Esto es un texto de prueba Artificial Intelligence (AI) y luego se vuelve a repetir AI.

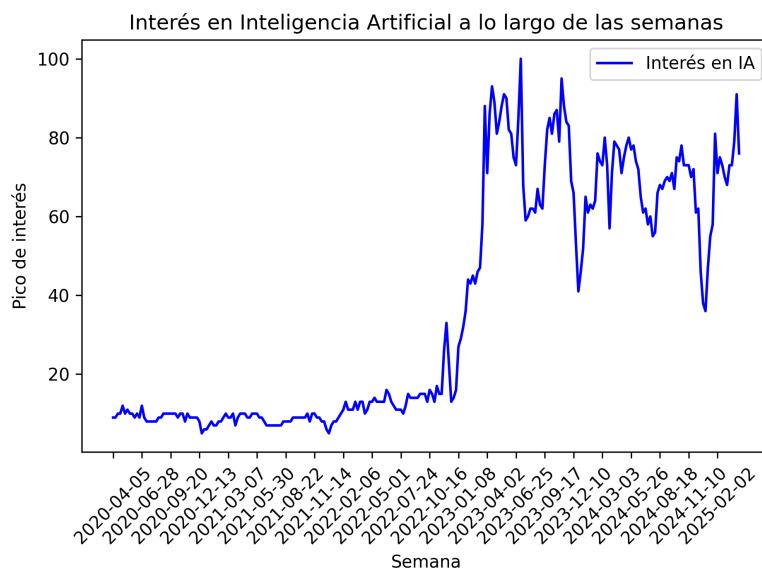


Figura 1.1: Evolución del interés público en inteligencia artificial según datos de Google Trends (2020-2025)

El progreso en visión por computador ha sido posible gracias a los avances en redes neuronales convolucionales, 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 CPUs (Unidades Centrales de Procesamiento) tradicionales resultan insuficientes para estas tareas, por lo que la industria ha desarrollado arquitecturas específicas como las GPUs (Unidades de Procesamiento Gráfico), TPUs (Unidades de Procesamiento Tensorial) y DLAs (Aceleradores de Aprendizaje Profundo). 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.

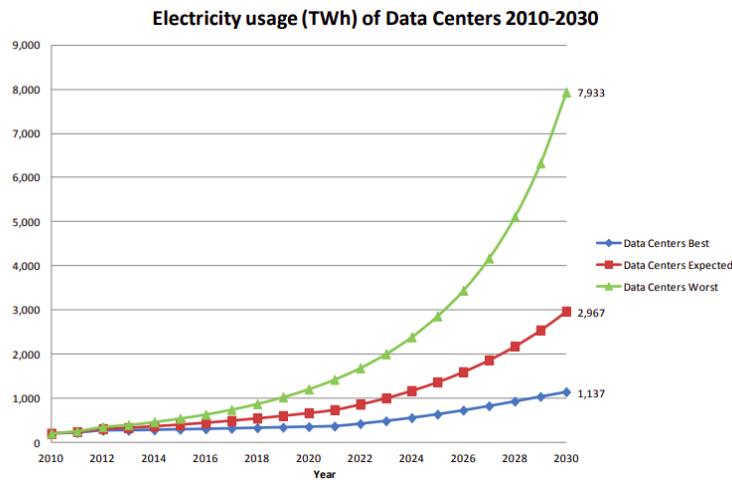


Figura 1.2: Proyección del consumo eléctrico de los centros de datos en el mundo

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 destacan por ofrecer un equilibrio entre alto rendimiento en tareas de inteligencia artificial y un consumo energético contenido, ideal para aplicaciones embebidas de visión artificial.

La combinación de redes neuronales convolucionales 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.

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 inteligencia artificial 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 inteligencia artificial 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 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 costos, 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 redes neuronales convolucionales y aceleradores hardware de bajo consumo. Para lograr este objetivo, se plantean los siguientes objetivos específicos:

- Realizar un estudio del estado del arte en redes neuronales convolucionales, 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 (mAP, precisión, recall), latencia (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

????? ?????????????? ?????????????? ?????????????? ?????????????? ??????????????

CAPÍTULO 2

Conceptos Previos

En este capítulo se explicará los conceptos previos que constituyen la base teórica y técnica de este trabajo. Primero, se examinarán los fundamentos en Inteligencia Artificial centrándose en las redes neuronales convolucionales, 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 redes neuronales convolucionales, 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 aprendizaje automático es una rama de la inteligencia artificial 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 aprendizaje automático 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 extrae 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 [8]. 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.

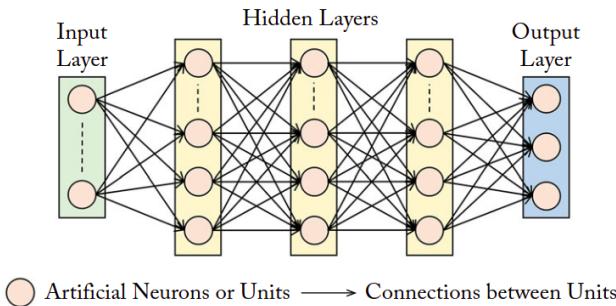


Figura 2.1: Estructura de un perceptrón multicapa (MLP).

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 Histograma de Gradientes Orientados (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 aprendizaje automático 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.

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 YOLO (You Only Look Once) y SSD (Single Shot MultiBox Detector), 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.

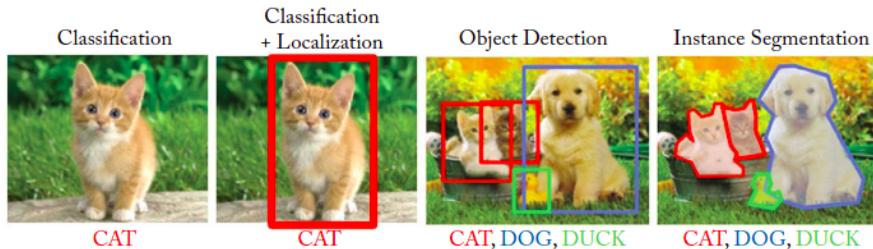


Figura 2.3: Tareas fundamentales en visión por computador.

2.1.3. Arquitectura y funcionamiento de las CNN

Las *Convolutional Neural Networks* (CNN) o redes neuronales convolucionales 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.

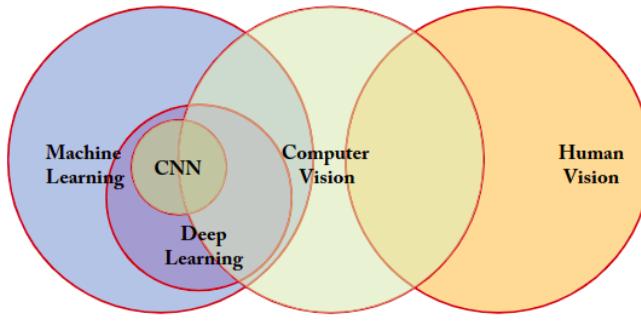


Figura 2.4: Relación entre Machine Learning, Deep Learning, CNN, Computer Vision y Human Vision.

La Figura 2.4 ilustra la relación entre estos conceptos [8]. 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 vídeos.

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.

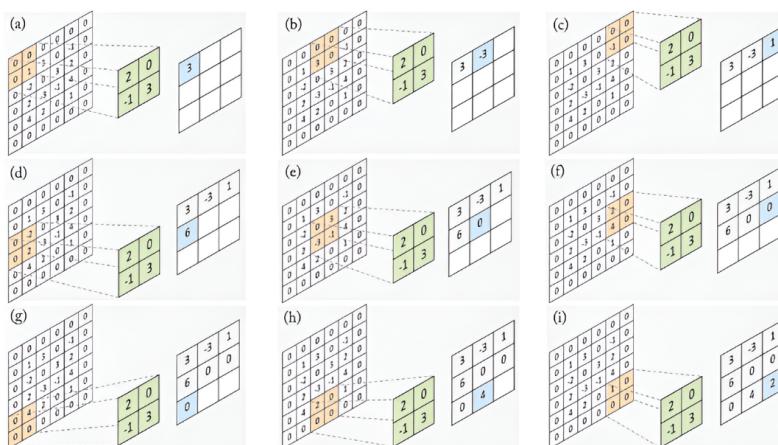


Figura 2.5: Operación de convolución sobre los píxeles de una imagen.

La Figura 2.5[8] 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 [5]. 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 paso de 2. Se selecciona el valor máximo de cada región de color.

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[7] se muestra un ejemplo de una red neuronal convolucional (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.

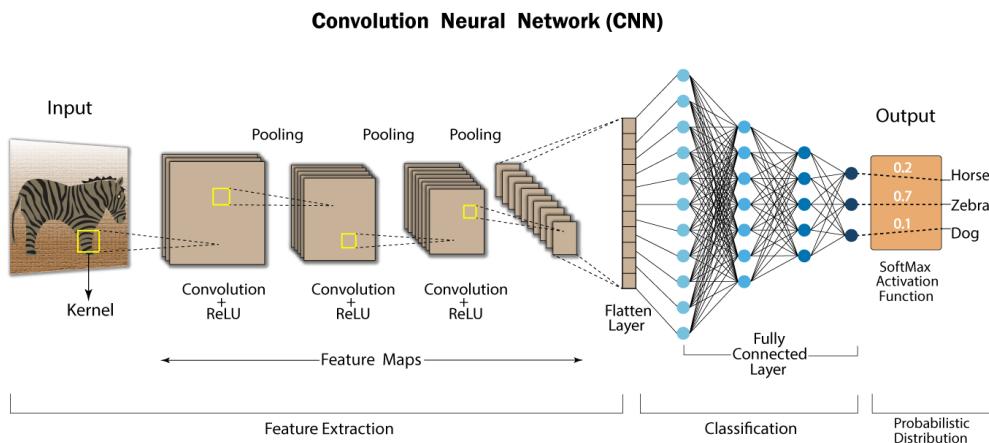


Figura 2.8: Ejemplo de una CNN simple.

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 (que contiene millones de imágenes en miles de categorías) o COCO (centrado en la detección y segmentación de objetos), 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) [4]. 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[10] se muestra un ejemplo del resultado del algoritmo de búsqueda selectiva.

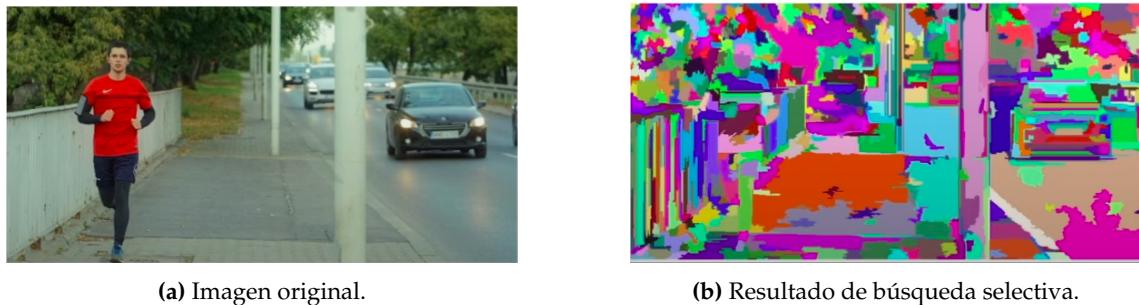


Figura 2.9: Proceso de búsqueda selectiva aplicado a una imagen.

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.

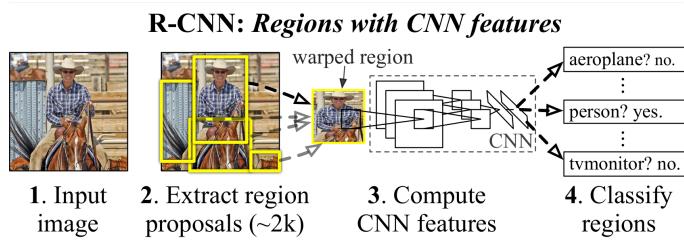


Figura 2.10: Arquitectura de R-CNN.

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 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)[12] y YOLO (You Only Look Once)[14]. Estos han demostrado un equilibrio eficaz entre rapidez y exactitud, permitiendo la detección en tiempo real incluso en dispositivos con recursos computacionales limitados.

YOLO (You Only Look Once) 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 me-

todología 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.

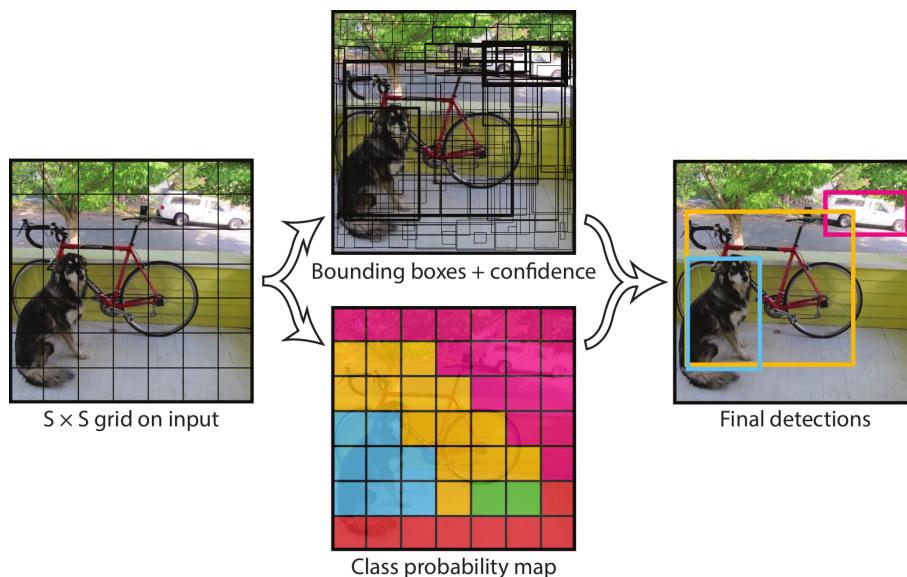


Figura 2.11: Ejemplo de detección de objetos utilizando YOLO.

Como se ilustra en la Figura 2.11[14], 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 todas las predicciones, YOLO implementa un post-procesamiento mediante supresión de no máximos (NMS) para eliminar detecciones redundantes y conservar únicamente las más precisas. 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.

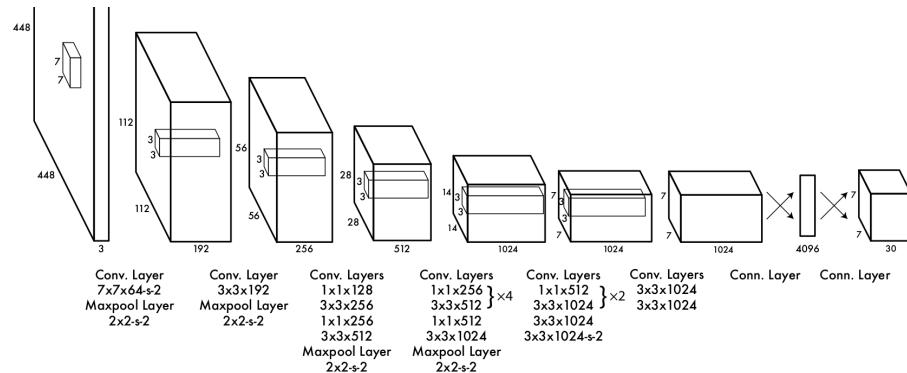


Figura 2.12: Arquitectura de YOLO

En la Figura 2.12 se presenta la arquitectura del modelo primigenio de YOLO [14]. 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 ha seleccionado YOLO11 [5], que representa una mejora y optimización sobre las arquitecturas YOLO precedentes. Dicho modelo forma parte de la librería Ultralytics [6], la cual ofrece una implementación eficaz y sencilla de diversas variantes de YOLO. Las aportaciones de Ultralytics a la comunidad de visión artificial son notables, ya que sus herramientas y recursos agilizan el ciclo de vida (entrenamiento, evaluación, implementación) de los modelos YOLO en múltiples contextos. YOLO11 se ofrece en diversas configuraciones (11n, 11s, 11m, 11l y 11x).

Modelo	Tamaño (px)	Latencia CPU ONNX (ms)	Latencia T4 TRT10 (ms)	mAPval (50-95)	Parámetros (M)
YOLO11n	640	56.1 ± 0.8	1.5 ± 0.0	39.5	2.6
YOLO11s	640	90.0 ± 1.2	2.5 ± 0.0	47.0	9.4
YOLO11m	640	183.2 ± 2.0	4.7 ± 0.1	51.5	20.1
YOLO11l	640	238.6 ± 1.4	6.2 ± 0.1	53.4	25.3
YOLO11x	640	462.8 ± 6.7	11.3 ± 0.2	54.7	56.9

Tabla 2.1: Análisis comparativo de las variantes de YOLO11 considerando precisión, velocidad y complejidad computacional.

La Tabla 2.1 presenta un análisis comparativo de las distintas variantes de YOLO11 evaluadas sobre el dataset COCO [11]. Los datos revelan una clara correlación: los modelos más grandes ofrecen mayor precisión (mAP) a costa de mayor complejidad computacional y menor velocidad de procesamiento. Esta escalabilidad permite adaptar la implementación según los requisitos específicos de cada aplicación, balanceando efectivamente el rendimiento con las restricciones computacionales del sistema objetivo.

2.1.6. Métricas de evaluación

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 (Precision):** 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.

- **mAP (mean Average Precision):** Es una métrica que combina precisión y recall, promediando la precisión a diferentes niveles de recall. Se utiliza para evaluar el rendimiento general del modelo en múltiples clases.

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.

2.2 Aceleradores de procesamiento gráfico

Un aspecto fundamental para el despliegue eficiente de modelos de inteligencia artificial 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, 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, 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.

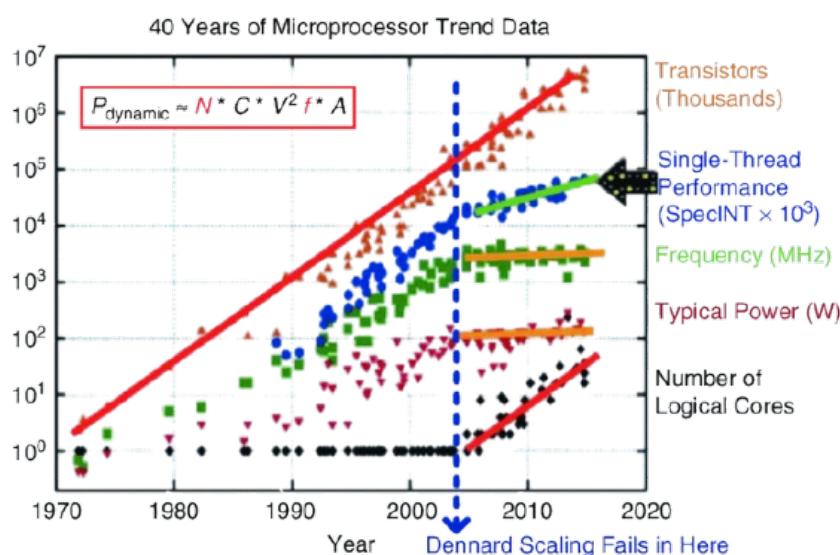


Figura 2.13: Evolución histórica de las características de los microprocesadores (1970-2020).

En la Figura 2.13[15] 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 ($P_{dynamic} \approx N * C * V^2 * f * A$) 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

Frente a las limitaciones de las CPUs, han surgido arquitecturas de dominio específico diseñadas para optimizar el rendimiento en tareas concretas. Entre ellas destacan las FPGAs (Field Programmable Gate Arrays), los ASICs (Application-Specific Integrated Circuits) y las GPUs (Graphics Processing Units). A diferencia de las CPUs de propósito general, estas arquitecturas sacrifican versatilidad para lograr una eficiencia superior en operaciones específicas como el procesamiento gráfico, la inteligencia artificial o el procesamiento de señales.

Las FPGAs son circuitos reconfigurables que pueden programarse después de su fabricación, ofreciendo flexibilidad para adaptarse a distintas aplicaciones, aunque su programación requiere conocimientos especializados. Los ASICs, por otro lado, se diseñan a medida para una única tarea, alcanzando la máxima eficiencia y rendimiento, pero a costa de un diseño costoso, largo y la imposibilidad de reprogramación. Un ejemplo notable de ASIC son las TPUs (Tensor Processing Units) desarrolladas por Google, optimizadas específicamente para acelerar las cargas de trabajo de aprendizaje automático neuronal.

Las GPUs (Unidades de Procesamiento Gráfico) son un ejemplo paradigmático de arquitectura de dominio específico orientada al paralelismo masivo. Su diseño *many-core* integra miles de núcleos de procesamiento más simples en un único chip. Esta estructura les permite ejecutar una gran cantidad de hilos de forma concurrente, lo que las hace ideales para cargas de trabajo intensivas y fácilmente parallelizables, como las operaciones matriciales típicas del aprendizaje profundo. Dada su capacidad para acelerar significativamente las tareas de inteligencia artificial, las GPUs constituyen la arquitectura hardware central utilizada 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 inteligencia artificial, como el entrenamiento y la inferencia de redes neuronales.

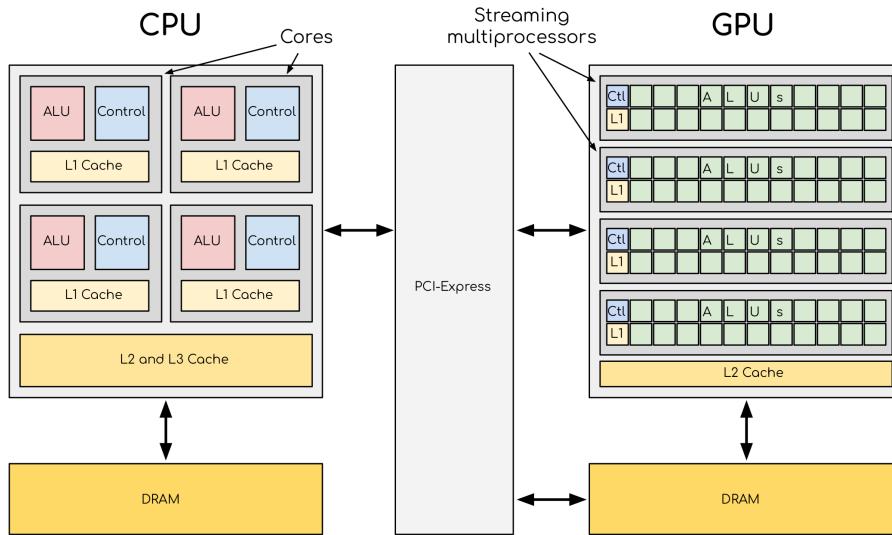


Figura 2.14: Comparativa de arquitecturas CPU y GPU.

La figura 2.14[3] 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 GPU 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 GPUs (GPGPU) con la introducción de CUDA en 2006. Esta plataforma permitió utilizar la masiva capacidad de procesamiento paralelo de las GPUs para tareas computacionales generales, extendiendo su uso más allá de los gráficos tradicionales. La programación de GPUs 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 inteligencia artificial.

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 de IA de bajo consumo

La serie Jetson de NVIDIA constituye una familia de módulos computacionales diseñados específicamente para habilitar la inteligencia artificial 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 inteligencia artificial 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 Deep Learning Accelerators (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.

Más allá del hardware, el valor de la plataforma Jetson reside en su ecosistema de software integral, proporcionado a través del NVIDIA JetPack SDK. Este paquete incluye el sistema operativo Linux optimizado (L4T), controladores, bibliotecas aceleradas por GPU como CUDA, cuDNN y TensorRT, además de herramientas de desarrollo y documentación. Esta plataforma unificada simplifica el ciclo de desarrollo, desde la creación y entrenamiento de modelos hasta su optimización y despliegue en los dispositivos Jetson, acelerando la innovación.

Modelo	AI Performance	GPU	GPU Max Freq.	CPU	Memoria	DLAs
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
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	-
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

Tabla 2.2: Comparativa técnica entre diferentes modelos NVIDIA Jetson.

La tabla 2.2[13] 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.

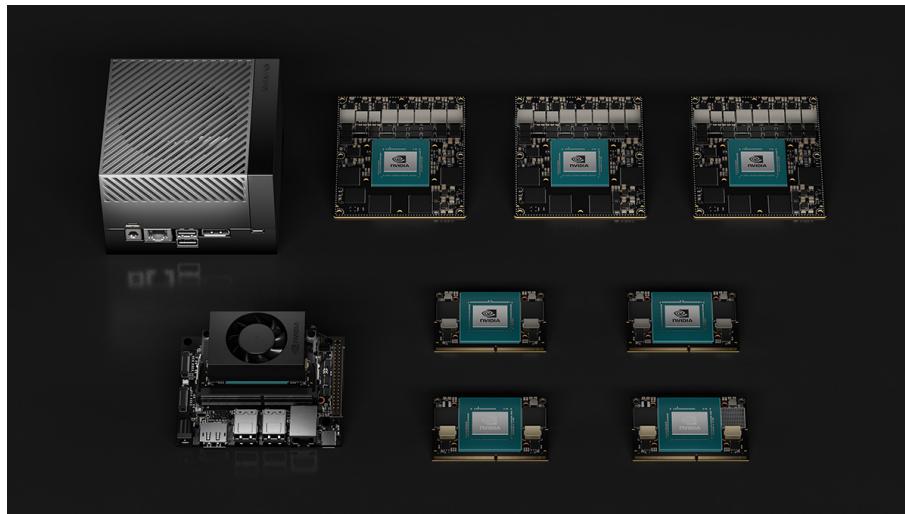


Figura 2.15: Módulos Jetson de NVIDIA.

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 operación 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.

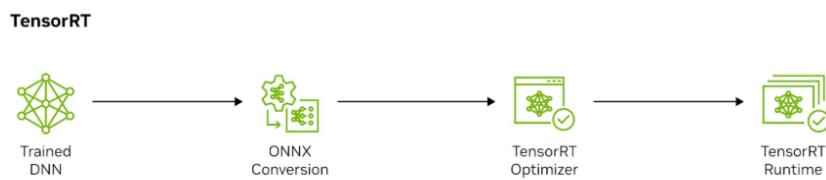


Figura 2.16: Ejemplo de flujo de trabajo de optimización con TensorRT.

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 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 (Open Neural Network Exchange), 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 presentará los conceptos básicos del seguimiento de objetos en tiempo real, se explicará el problema del seguimiento de objetos múltiples (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 Seguimiento de Objetos Múltiples (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.
- **PREDDECIR** 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.
- **OUTPUTS: Estimación del estado del sistema** 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 [16] es un algoritmo avanzado de Seguimiento de Objetos Múltiples (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.

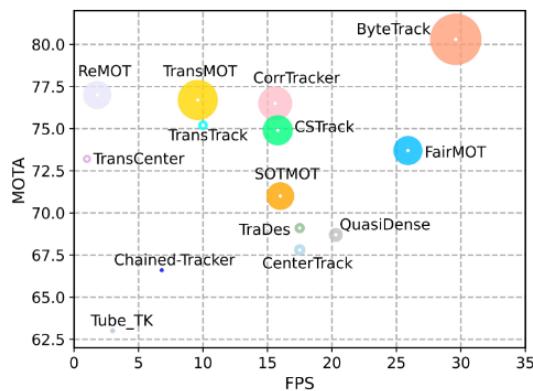


Figura 2.18: Comparativa de rendimiento de BYTETrack con otros algoritmos de seguimiento.

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 [16]. 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.

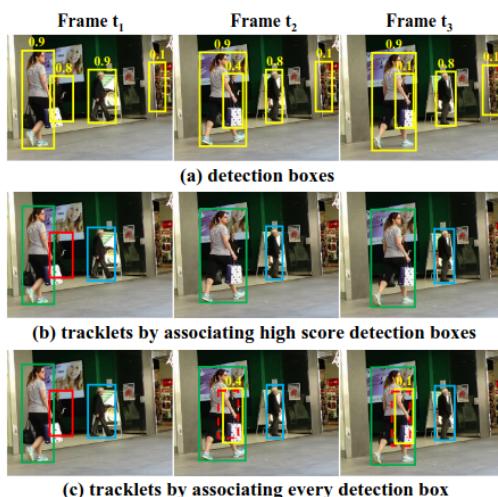


Figura 2.19: Ejemplo de detección y seguimiento de objetos utilizando BYTETrack.

Fuente: [16]

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 BYTETrack, que integra también las detecciones de baja confianza en el proceso de asociación. Esta comparación evidencia cómo la estrategia de BYTETrack 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 BYTETrack 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*[9] 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.

CAPÍTULO 3

Diseño e implementación de la solución

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. Este sistema se fundamenta en la utilización de la plataforma de hardware NVIDIA Jetson, reconocida por su capacidad para ejecutar tareas de inteligencia artificial 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. Aun-

que 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. 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 del modelo

El proceso de entrenamiento del modelo de detección de objetos es una fase fundamental en el desarrollo del sistema propuesto. La elección del modelo adecuado y la preparación de un conjunto de datos representativo son cruciales para asegurar la precisión y robustez del sistema final.

Para la tarea de detección de objetos, se ha seleccionado el modelo YOLO11, perteneciente a la familia de arquitecturas YOLO (You Only Look Once). Esta elección se basa en varios factores clave. En primer lugar, YOLO11 es conocido por su excelente equilibrio entre velocidad de inferencia y precisión de detección, lo que lo hace particularmente adecuado para aplicaciones que requieren procesamiento en tiempo real, como es el caso de este proyecto. En segundo lugar, YOLO11, especialmente a través de la implementación proporcionada por la biblioteca Ultralytics, ofrece una gran flexibilidad en términos de escalabilidad (con variantes como 11n, 11s, 11m, etc.) y facilidad de uso para el entrenamiento, la validación y el despliegue.

Una vez seleccionado el modelo, el siguiente paso crítico es la creación y preparación del conjunto de datos (dataset). Este conjunto de datos es la base sobre la cual el modelo aprenderá 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 del modelo. 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, oclusiones 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 al modelo la "verdad fundamental"(ground truth) necesaria para su aprendizaje supervisado.

Para el etiquetado de las imágenes, se utilizó la herramienta CVAT[2] (Computer Vision Annotation Tool) 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.

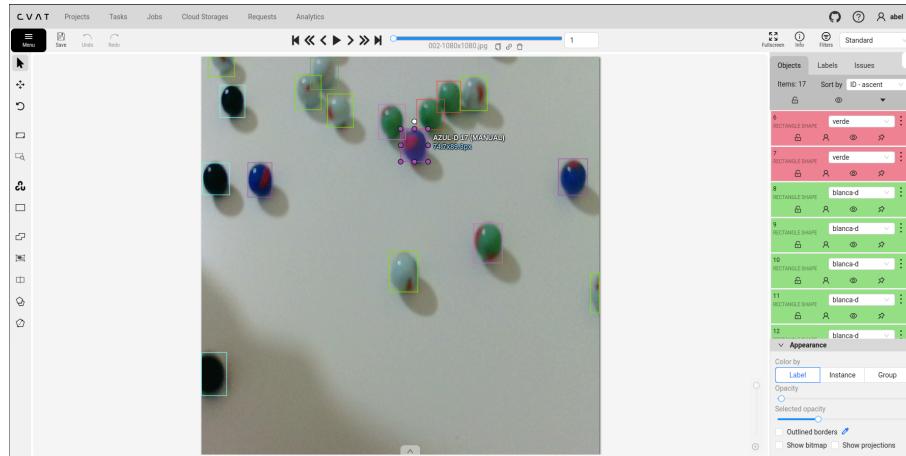


Figura 3.2: Ejemplo de anotación de imágenes utilizando CVAT.

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.

// PONER AQUI LOS RESULTADOS DEL ENTRENAMIENTO DEL MODELO Y LAS DIFERENTES PRUEBAS REALIZADAS

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 3.3, 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 YOLO11, un módulo de seguimiento que implementa el algoritmo BYTETrack para mantener la identidad de los objetos detectados, y un módulo de escritura que gestiona los diferentes tipos de salida del sistema.

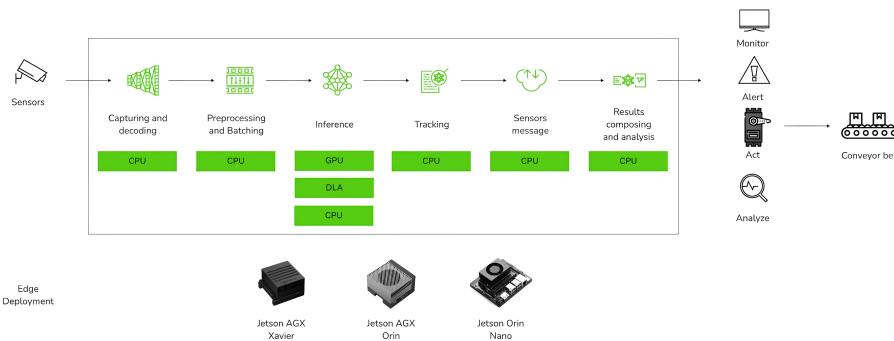


Figura 3.3: Figura del sistema propuesto

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 preprocessamiento 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 preprocessamiento 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 [6], 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 Supresión de No Máximos (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[6], 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 3.4: Ejemplo de salida del sistema.

La Figura 3.4 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 en diferentes 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 NVIDIA Jetson.

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.

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.

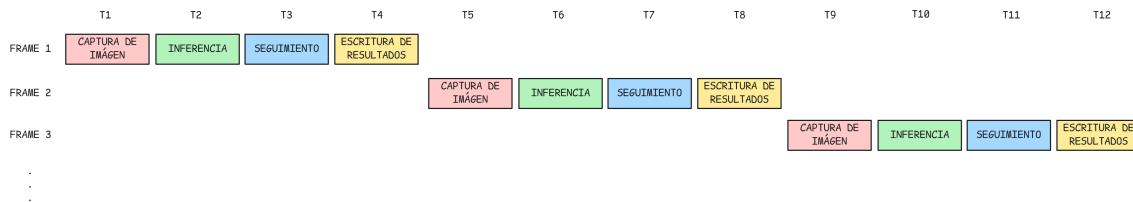


Figura 3.5: Diagrama de flujo del sistema sin segmentar.

La Figura 3.5 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 basada 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 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()’.

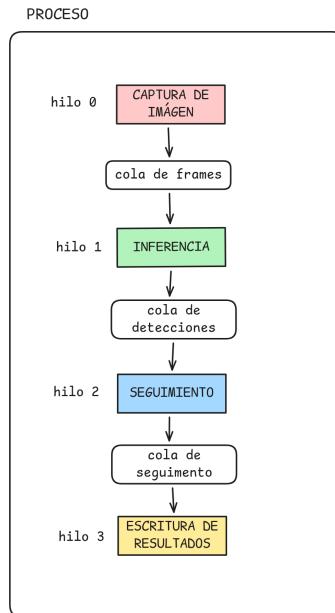


Figura 3.6: Diagrama de flujo del sistema segmentado en hilos.

La Figura 3.6 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 basada 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.

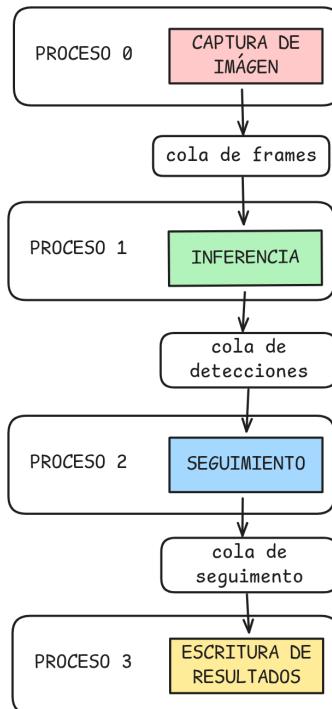


Figura 3.7: Diagrama de flujo del sistema segmentado en procesos.

La Figura 3.7 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.

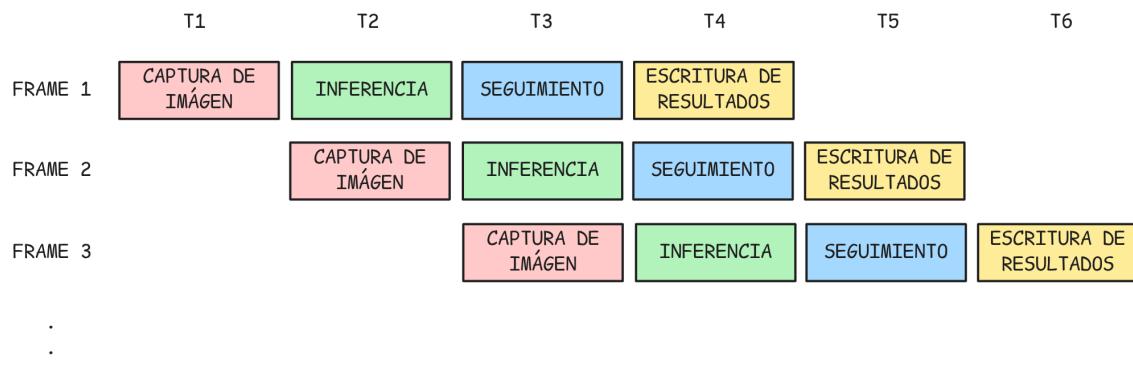


Figura 3.8: Diagrama de flujo del sistema segmentado.

La Figura 3.8 ilustra el flujo de datos en un sistema segmentado mediante procesos independientes, contrastando con el enfoque secuencial mostrado en la Figura 3.5. 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 3.5), 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 basada en hardware

La cuarta opción es la segmentación basada en hardware. 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 Deep Learning Accelerators (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.

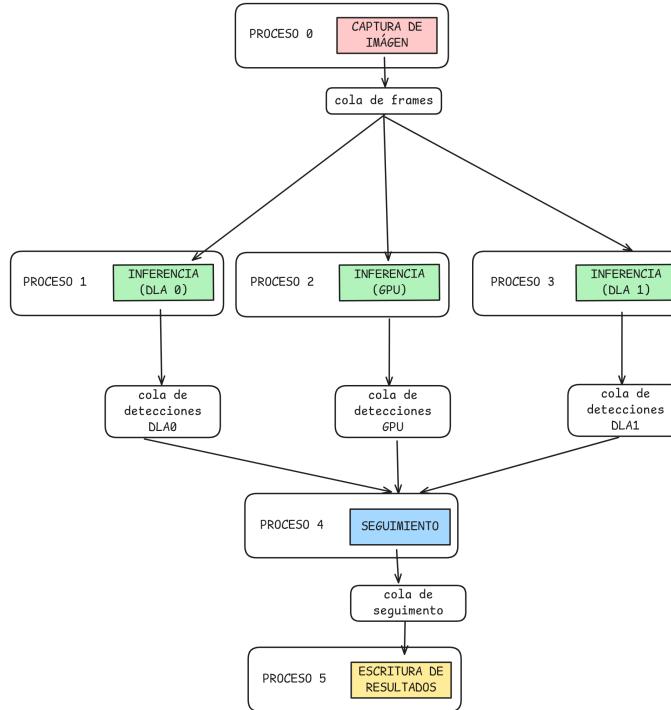


Figura 3.9: Diagrama de flujo del sistema segmentado en diferentes unidades de procesamiento.

La Figura 3.9 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 también emplea procesos independientes, 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.

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 3.10 ilustra esta estructura conceptual.

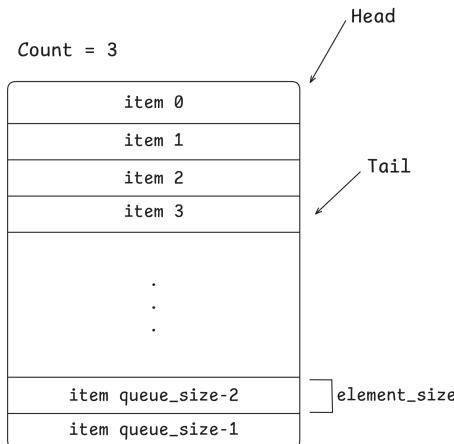


Figura 3.10: Ejemplo de buffer circular.

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 (*thread-safe* y *process-safe*).

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 ofrece un rendimiento potencialmente superior al de `multiprocessing.Queue` para la transferencia de grandes bloques de datos como fotogramas de vídeo, al evitar la sobrecarga de serialización y copia. Sin embargo, esta eficiencia conlleva una mayor complejidad de implementación, requiriendo una gestión manual cuidadosa de la memoria y la sincronización explícita.

CAPÍTULO 4

Análisis de la solución

En este capítulo se analizará la solución propuesta variando los parámetros posibles

4.1 Variación de los parámetros

Explicación de los parámetros que se pueden variar en la solución propuesta y su efecto en el rendimiento del sistema.

—PRUEBA—

Model	IoU	CPU_Inference	GPU_Inference	DLA_Inference	CPU_Power	GPU_Power	DLA_Power	CPU_Energy	GPU_Energy	DLA_Energy
YOLOv11-N	0,85	45,2	12,3	15,8	8,2	12,5	6,8	369,64	153,75	107,44
YOLOv11-S	0,87	52,1	14,8	18,2	8,5	13,2	7,1	442,85	195,36	129,22
YOLOv11-M	0,89	68,4	18,2	22,5	9,1	14,8	7,8	622,44	269,36	175,5
YOLOv11-L	0,91	85,6	24,6	28,9	9,8	16,2	8,4	838,88	398,52	242,76

Tabla 4.1: Comparación de modelos en términos de inferencia, consumo de energía y potencia.

—PRUEBA—

4.2 Tipo de segmentación

En esta sección se analizará el rendimiento de la solución propuesta variando el tipo de segmentación de las etapas del sistema con gráficas y tablas.

4.3 Talla del modelo

En esta sección se analizará el rendimiento de la solución propuesta variando la talla del modelo de detección de objetos con gráficas y tablas.

4.4 Precisión del modelo

En esta sección se analizará el rendimiento de la solución propuesta variando la precisión del modelo de detección de objetos con gráficas y tablas.

4.5 Modo de energía y cores de la CPU

En esta sección se analizará el rendimiento de la solución propuesta variando el modo de energía del dispositivo y el número de cores de la CPU con gráficas y tablas.

4.6 Tamaño de la imagen

En esta sección se analizará el rendimiento de la solución propuesta variando el tamaño de la imagen de entrada del modelo con la técnica de Slicing Aided Hyper Inference (SAHI) con gráficas y tablas.

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

5.2 Instalación del entorno

????? ?????????????? ?????????????? ?????????????? ?????????????? ??????????????

CAPÍTULO 6

Conclusiones

????? ?????????????? ?????????????? ?????????????? ?????????????? ??????????????

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] CVAT.ai Corporation. Computer Vision Annotation Tool (CVAT), November 2023.
- [3] 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?".
- [4] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2014.
- [5] Glenn Jocher and Jing Qiu. Ultralytics yolo11, 2024.
- [6] Glenn Jocher, Jing Qiu, and Ayush Chaurasia. Ultralytics YOLO, January 2023.
- [7] Swapna Kategaru. Convolution neural network in deep learning. <https://developersbreach.com/convolution-neural-network-deep-learning/>, 2025. Accedido el 26 de abril de 2025.
- [8] 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.
- [9] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.
- [10] Tushar Kumar. R-cnn explained, 2024. Accedido: 14 de abril de 2025.
- [11] 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.
- [12] 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.
- [13] NVIDIA Corporation. Jetson modules, support, ecosystem, and lineup. <https://developer.nvidia.com/embedded/jetson-modules>, 2025. Accedido el 24 de abril de 2025.
- [14] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.
- [15] Liming Xiu. Time moore: Exploiting moore's law from the perspective of time. *IEEE Solid-State Circuits Magazine*, 11:39–55, 01 2019.

- [16] 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

Configuración del sistema

????? ?????????????? ?????????????? ?????????????? ?????????????? ??????????????

A.1 Fase de inicialización

????? ?????????????? ?????????????? ?????????????? ?????????????? ??????????????

A.2 Identificación de dispositivos

????? ?????????????? ?????????????? ?????????????? ?????????????? ??????????????

APÉNDICE B

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 aprendizaje automático 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 hard-

ware especializado como GPUs, algoritmos de seguimiento multi-objeto y técnicas de paralelización. 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.