UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE ELECTRÓNICA

# EXPLORACIÓN DE METODOLOGÍAS PARA LA OPTIMIZACIÓN DE INFERENCIA DE REDES NEURONALES EN GPU UTILIZANDO TENSORRT

Tesis de Grado presentada por

**Juan Carlos Aguilera Castillo**

como requisito parcial para optar al título de

**Ingeniero Civil Electrónico**

y al grado de

**Magíster en Ciencias de la Ingeniería Electrónica**

Director de Tesis
Dr. Gonzalo Carvajal

Valparaíso, 2025.

# CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

## 1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

**Tipo de monografía (marcar una opción):** ☐ Memoria o trabajo de título; ☒ Tesis de Postgrado;

**Título del trabajo:** Exploración de metodologías para la optimización de inferencia de redes neuronales en GPU utilizando TensorRT

**Nombre del candidato(a):** Juan Carlos Aguilera Castillo

**Carrera / Grado:** Magíster en Ciencias de la Ingeniería Electrónica

**Campus:** Campus Casa Central Valparaíso **; Departamento**: Departamento de Electrónica

## 2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Gonzalo Carvajal, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución

## 3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL

☒ El trabajo **NO contiene información que amerite confidencialidad** y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (embargo) por:

☐ 6 meses; ☐ 12 meses; ☐ 2 años; ☐ 3 años; ☐ 5 años; ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

## 4.- FIRMAS

**Profesor(a) guía o director(a) de memoria o tesis:**

**Fecha: 15/09/2025 ; Firma:**

**Estudiante o Candidato(a):**

**Fecha: 15/09/2025 ; Firma:**

*Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.*

TíTULO DE LA TESIS:

**EXPLORACIÓN DE METODOLOGÍAS PARA LA OPTIMIZACIÓN DE INFERENCIA DE REDES NEURONALES EN GPU UTILIZANDO TENSORRT**

AUTOR:

**Juan Carlos Aguilera Castillo**

TRABAJO DE TESIS, presentado en cumplimiento parcial de los requisitos para el título de **Ingeniero Civil Electrónico** y el grado de **Magíster en Ciencias de la Ingeniería Electrónica** de la Universidad Técnica Federico Santa María.

**Director de Tesis**
Dr. Gonzalo Carvajal ______________________

**Examinador Interno**
Dr. Marcos Zúñiga ______________________

**Examinador Externo**
Dr. Daniel Yunge ______________________

Valparaíso, 2025.

# AGRADECIMIENTOS

En primer lugar, deseo expresar mi más profundo agradecimiento a mi familia, y en especial a mis padres, por su apoyo incondicional.

También quiero agradecer a los amigos que conocí durante estos años de universidad. Con su compañía y apoyo, logré superar muchas de las barreras que se presentaron en esta etapa de mi vida.

Expreso, además, mi sincera gratitud a mi director de tesis, Gonzalo Carvajal, por su constante orientación y valiosa retroalimentación a lo largo del desarrollo de este trabajo.

Durante el trabajo desarrollado en esta tesis se recibió apoyo financiero del proyecto interno de investigación multidisciplinaria PI_M_23_05 de la Universidad Técnica Federico Santa María.

# CONTENIDO

**AGRADECIMIENTOS** II

**ÍNDICE DE FIGURAS** V

**ÍNDICE DE TABLAS** VII

**RESUMEN** IX

**ABSTRACT** X

**1. INTRODUCCIÓN** 1
- 1.1. Motivación y Contexto 1
- 1.2. Planteamiento del problema 5
- 1.3. Alcances y contribuciones 6
- 1.4. Organización de la Tesis 6

**2. ANTECEDENTES** 8
- 2.1. Overview y terminologia 8
- 2.2. CUDA Programming Model 12
- 2.3. Descripción de TensorRT 14
  - 2.3.1. Building phase 15
  - 2.3.2. Running Phase 20
- 2.4. Trabajos relacionados 20

**3. EVALUACIÓN DE CONFIGURACIONES REPORTADAS EN EL ESTADO DEL ARTE** 23
- 3.1. Configuración del entorno de operaciones 23
  - 3.1.1. Aplicación 24
  - 3.1.2. Workflow 25
  - 3.1.3. Hardware 26
  - 3.1.4. Métricas 27
- 3.2. Reporte de resultados 29
  - 3.2.1. Precisión de la inferencia 29

3.2.2. Caracterización de las propiedades del modelo 29
3.2.3. Caracterización de la latencia 31
3.2.4. Caracterización del throughput 32
3.3. Discusión de resultados 33

**4. EVALUACIÓN CON NUEVAS CONFIGURACIONES Y MÉTRICAS 36**
4.1. Configuración del entorno de operación 36
4.1.1. Aplicación 37
4.1.2. Workflow 37
4.1.3. Hardware 38
4.1.4. Métricas 39
4.2. Reporte de resultados 40
4.2.1. Caracterización de classification closeness 40
4.2.2. Batch size de configuración estático y dinámico 42
4.2.3. Modo de consumo 45
4.2.4. Nivel de optimización 48
4.2.5. Discusión de resultados 51

**5. EVALUACIÓN DE APLICACIONES 53**
5.1. Segmentación de salmones 53
5.1.1. Entorno de operación 54
5.1.2. Metodología de evaluación y configuración 56
5.1.3. Resultados y análisis 57
5.1.4. Discusión de resultados 64
5.2. Pirometría de hollín 65
5.2.1. Entorno de operación 66
5.2.2. Metodología de evaluación y configuración 69
5.2.3. Resultados y análisis 70
5.2.4. Discusión de resultados 76

**6. CONCLUSIONES Y TRABAJO FUTURO 77**

**REFERENCIAS 79**

**A. TABLAS DE RESULTADOS GENERALES 84**

**B. TABLAS DE RESULTADOS COMPLETOS DEL THROUGHPUT 90**

**C. CONFIGURACIÓN QUE PUEDE AUMENTAR EL USO DE MEMORIA DURANTE LA INFERENCIA 94**

# Índice de figuras


1.1. Diagrama de una ANN y una neurona. 2

2.1. Representación de un grafo computacional. 10
2.2. Flujo de trabajo para la optimización de la inferencia usando TensorRT. 11
2.3. Diagrama de comunicación entre *host* y *device*. 12
2.4. Flujo de una TensorRT Application. 15
2.5. Red neuronal de múltiples capas con respectivas fusiones de capas. 19

3.1. Diagrama del workflow Torch-ONNX-TensorRT. 25
3.2. Comparación de latencias promedio y máxima. 32
3.3. Comparación del throughput en función del batch size de inferencia. 33

4.1. Uso de memoria para distintas configuraciones de batch size. 43
4.2. Throughput para distintas configuraciones de batch size. 44
4.3. Uso de memoria durante la inferencia para distintos modos de consumo. 46
4.4. Latencia para distintos modos de consumo. 47
4.5. Throughput para distintos modos de consumo. 48
4.6. Uso de memoria durante la inferencia para distintos niveles de optimización. 49
4.7. Latencia para distintos niveles de optimización. 50
4.8. Throughput para distintos niveles de optimización. 50

5.1. Imágenes representativas extraídas del conjunto de datos de validación: *Img1*, *Img2*, *Img3* e *Img4*. 57
5.2. Segmentación de un ejemplo representativo. 59
5.3. Uso de memoria en segmentación de salmones. 62
5.4. Segmentación de un ejemplo representativo modelo final. 64
5.5. Esquema del arreglo óptico BEMI para capturar la radiación emitida por las partículas de hollín. 66
5.6. Soluciones de referencia para los campos de $T_s$ en una llama CLAD de Yale utilizando CoFlame. 68

5.7. Comparación de la predicción de $T_s$ entre el modelo base y los modelos optimizados bajo la condición de llama Yale-32. Se incluye el mapa de error $\Delta T$ respecto al modelo base. 72
5.8. Comparación de la predicción de $T_s$ entre el modelo base y los modelos optimizados bajo la condición de llama Yale-40. Se incluye el mapa de error $\Delta T$ respecto al modelo base. 73
5.9. Comparación de la predicción de $T_s$ entre el modelo base y los modelos optimizados bajo la condición de llama Yale-60. Se incluye el mapa de error $\Delta T$ respecto al modelo base. 74

C.1. Uso de memoria excesivo en segmentación de salmones. 95

# Índice de tablas

2.1. Niveles populares de software y hardware. 9
2.2. Resumen del estado del arte. 21

3.1. Entorno de operaciones. 24
3.2. Características de los dispositivos utilizados. 26
3.3. Versiones de software. 26
3.4. Modos de consumo energético predeterminados según la plataforma. 27
3.5. Resultados de precisión de la inferencia. 30
3.6. Variación en las propiedades del modelo respecto al modelo base. 31

4.1. Entorno de operación. 37
4.2. Modos de consumo evaluados según la plataforma. 39
4.3. Resultados de classification closeness. 41

5.1. Propiedades del modelo: modelo base, TRT fp32, TRT fp16 y TRT int8 generados con batch size de configuración dinámico. 58
5.2. Resultados de accuracy en porcentaje (%). 59
5.3. Resultados de segmentation closeness en porcentaje (%). 60
5.4. Resultados de latencia y throughput en la caracterización de la configuración del batch size en un problema de segmentación. 61
5.5. Resultados obtenidos al evaluar distintos niveles de optimización en un problema de regresión. 63
5.6. Resultados obtenidos al evaluar distintos modos de consumo en un problema de regresión. 63
5.7. Propiedades de los modelos U-Net y Attention (**Att.**) U-Net en sus distintas versiones: modelo base, TRT `fp32`, TRT `fp16` y TRT `int8`. 70
5.8. Resultados de **regression accuracy** en Kelvin (**K**) bajo diferentes condiciones de llama. 71
5.9. Resultados de closeness para U-Net y Attention U-Net. 73
5.10. Resultados de latencia y throughput en la caracterización del batch size en un problema de regresión. 74
5.11. Resultados obtenidos al evaluar distintos niveles de optimización en un problema de regresión. 75

5.12. Resultados obtenidos al evaluar distintos modos de consumo en un problema de regresión. 75

A.1. Resumen de resultados en la plataforma Jetson Orin AGX. 85
A.2. Resumen de resultados en la plataforma Jetson Orin Nano. 86
A.3. Resumen de resultados en la plataforma Jetson Xavier AGX. 87
A.4. Resumen de resultados en la plataforma RTX 2060MaxQ. 88
A.5. Resumen de resultados en la plataforma RTX 3060. 89

B.1. Resultados de throughput para el modelo de red MobileNetV2. 91
B.2. Resultados de throughput para el modelo de red ResNet18. 91
B.3. Resultados de throughput para el modelo de red ResNet34. 92
B.4. Resultados de throughput para el modelo de red ResNet50. 92
B.5. Resultados de throughput para el modelo de red ResNet101. 93
B.6. Resultados de throughput para el modelo de red ResNet152. 93

# RESUMEN

Las redes de deep learning han alcanzado un éxito considerable en tareas complejas, como el reconocimiento de patrones y la clasificación de datos. Sin embargo, realizar inferencias con estas redes demanda muchos recursos computacionales debido al número de operaciones y los requisitos de memoria, lo cual limita su efectividad en aplicaciones con garantías estrictas de rendimiento en términos de latencia y throughput, como en la robótica y las tecnologías de conducción asistida.

El procesamiento necesario para inferencias con deep learning muestra un alto grado de paralelismo en sus operaciones subyacentes, lo que puede ser aprovechado con las Unidades de Procesamiento Gráfico (GPUs) modernas. Sin embargo, mapear redes descritas en frameworks de alto nivel —que priorizan la productividad sobre el rendimiento— en GPUs es una tarea compleja. Para abordar este desafío, Nvidia introdujo TensorRT, una herramienta de software diseñada para optimizar algoritmos de redes neuronales en GPUs, mejorando el rendimiento en la inferencia mediante el uso eficiente de la computación paralela.

Los detalles internos de TensorRT son propietarios y cerrados, por lo que solo es posible evaluar su efectividad a través de estudios empíricos. Aunque estos estudios sugieren que TensorRT mejora el rendimiento de la inferencia en tareas como la clasificación de imágenes, la efectividad depende en gran medida de la configuración de la herramienta para el hardware objetivo. Los estudios recientes evalúan TensorRT utilizando varios modelos de redes neuronales profundas (DNN), configuraciones de hardware/software y métricas de rendimiento, pero suelen carecer de detalles concretos sobre configuraciones y códigos fuente, lo que limita la validación y extensión de los resultados. Además, la rápida evolución de los algoritmos de aprendizaje automático y de las tecnologías de soporte requiere evaluaciones periódicas para asegurar la validez de los hallazgos y derivar pautas para nuevos modelos y aplicaciones.

En esta tesis, se realiza una exploración experimental sistemática de las capacidades de optimización de TensorRT para tareas de inferencia, utilizando modelos de redes neuronales profundas ejecutados en GPUs de diversos rangos, con un enfoque en la familia Jetson de plataformas embebidas de Nvidia. Este estudio aborda las brechas identificadas en la literatura al ampliar los benchmarks para incluir las plataformas Jetson Orin más recientes e incorporar nuevas configuraciones de herramientas y métricas de evaluación. Al probar múltiples plataformas en un entorno uniforme, se establecieron pautas que luego fueron validadas en una variedad de aplicaciones utilizando modelos de DNN y conjuntos de datos personalizados que difieren de los ejemplos típicos utilizados en los benchmarks. En general, este estudio proporciona datos cuantitativos y verificables sobre las fortalezas y limitaciones de TensorRT en la optimización de inferencias con algoritmos de deep learning en plataformas de última generación. Anticipamos que estos hallazgos ayudarán a los profesionales y usuarios finales a aprovechar eficazmente las tecnologías de última generación para optimizar tareas de inferencia considerando requisitos y restricciones específicos.

# ABSTRACT

Deep learning networks have achieved considerable success in complex tasks such as pattern recognition and data classification. However, performing inference with these networks is computationally demanding due to the number of operations and memory requirements, which limits their effectiveness in applications that require strict performance guarantees in terms of latency and throughput, such as robotics and assisted driving technologies, among others.

The processing required to perform inferences with deep learning networks tends to exhibit a high degree of parallelism in the underlying operations, which can be leveraged by modern Graphics Processing Units (GPUs). However, mapping networks described in high-level frameworks—which prioritize productivity and functionality over computational performance—onto GPUs is a complex task. To address this challenge, Nvidia introduced TensorRT, a software tool designed to facilitate the mapping of neural network-based algorithms onto GPUs manufactured by the same company, with the goal of optimizing the utilization of parallel computing resources to improve inference performance.

The internal details of TensorRT are proprietary and closed, making it possible to evaluate the tool's effectiveness only through empirical studies. Although studies suggest that TensorRT is effective in improving inference performance for tasks such as image classification, the optimization's effectiveness depends heavily on configuring the tool for specific target hardware. Recent studies evaluate TensorRT using various DNN models, datasets, software/hardware configurations, and target metrics. However, existing studies generally lack concrete details about settings and generation scripts, which limits the ability to validate results and extend them to other datasets and applications. Furthermore, the rapid evolution of machine learning algorithms and supporting software/hardware technologies necessitates periodic evaluations to ensure previous findings remain valid and to derive guidelines for extending the techniques to new models and applications with custom datasets and target metrics.

In this thesis, we conducted a systematic experimental exploration of TensorRT's optimization capabilities for inference tasks using DNN models executed on GPUs across various ranges, with a focus on Nvidia's Jetson family of embedded platforms. Our study addresses gaps identified in the literature by extending benchmarks to include the latest Jetson Orin platforms and incorporating new tool settings and evaluation metrics. By testing multiple platforms under a uniform environment, we established guidelines that were further validated across a range of applications using DNN models and custom datasets that differ from typical benchmarking examples. Overall, this study provides quantitative and verifiable insights into TensorRT's strengths and limitations in optimizing deep learning inference on next-generation platforms. We anticipate that our findings will assist practitioners and end-users in effectively leveraging state-of-the-art technologies to optimize inference tasks while considering specific requirements and constraints.

# Capítulo 1

# INTRODUCCIÓN

Este documento describe el trabajo realizado para implementar un análisis sistemático del uso y desempeño de TensorRT, una herramienta orientada a optimizar el uso de recursos durante la inferencia en redes neuronales profundas (*Deep Neural Networks* o DNNs) en sistemas computacionales basados en GPUs de Nvidia. El objetivo general del trabajo es identificar y proponer pautas y directrices generales para el uso adecuado de esta herramienta en diversos campos de aplicación, mediante evaluaciones sistemáticas con casos de estudio específicos. Estas aplicaciones pueden presentar restricciones variables en términos de latencia de inferencia, throughput o la capacidad del hardware disponible.

Este capítulo contextualiza el trabajo de tesis. Comienza con una explicación de la motivación y el contexto del estudio, seguida del planteamiento del problema. Posteriormente, se detallan los alcances y las contribuciones del trabajo, concluyendo con una descripción de los capítulos que conforman el documento.

## 1.1. Motivación y Contexto

La disciplina de *machine learning* se centra en el desarrollo de técnicas y algoritmos que permiten a los sistemas computacionales aprender características de conjuntos de datos y mejorar su desempeño en tareas específicas a partir de ejemplos, sin requerir la programación explícita de instrucciones. En los últimos años, las técnicas de machine learning han revolucionado la industria tecnológica, impactando significativamente tareas como la clasificación, detección y segmentación de imágenes, así como la aproximación de funciones. Una subdisciplina clave dentro del machine learning es el *deep learning*, que ha sido un factor habilitador en el desarrollo acelerado de aplicaciones como la conducción autónoma [1], el reconocimiento de lenguaje natural [2], la predicción de enfermedades [3], el reconocimiento facial, entre muchas otras.

El deep learning se caracteriza por el uso de redes neuronales artificiales (*Artificial Neural Networks* o ANNs) [4]. La Figura 1.1 muestra un esquema de alto nivel de una ANN del tipo *feedforward*, las cuales están compuestas por múltiples capas formadas por nodos, comúnmente llamados neuronas. En una red feedforward, la información se propaga en una sola dirección desde la capa de entrada hasta la capa de salida, pasando por una o más capas ocultas, sin conexiones cíclicas que retroalimenten capas anteriores. Cada neurona genera una salida a partir de una combinación lineal de sus entradas con un conjunto de parámetros ajustables o pesos, seguida de una función de activación no lineal. Este tipo de red es ampliamente utilizado para tareas de clasificación y regresión, donde aprende a mapear entradas a salidas mediante un proceso de entrenamiento supervisado, ajustando sus pesos en función de los errores cometidos. Además, dentro del espacio de diseño de

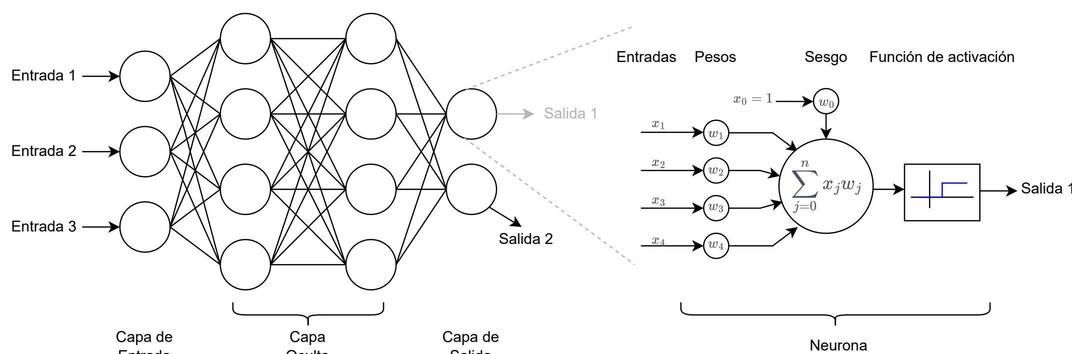



Fig. 1.1: Diagrama de una ANN y una neurona. A la izquierda se muestra el diagrama de una ANN con sus capas de entrada, intermedias y de salida. A la derecha se muestra el diagrama de una neurona, con sus entradas, pesos, sesgo, función de activación y salida.

una red neuronal, es importante tomar en cuenta los hiperparámetros, que corresponden a variables de configuración de la estructura que no se modifican durante el proceso de entrenamiento, lo cual incluye el número de capas, la cantidad de neuronas por capa, la función de activación utilizada, la tasa de aprendizaje, entre otros.

Las operaciones fundamentales de las ANN corresponden a operaciones vectoriales y matriciales que se enmarcan dentro de las operaciones típicas de álgebra lineal. Si bien las operaciones fundamentales son estructuralmente simples, las redes neuronales modernas suelen contener un gran número de neuronas, capas y parámetros, lo cual resulta en una gran cantidad de operaciones. Por ejemplo, el modelo `yolov8l-seg` [5], diseñado para la segmentación de imágenes, consta de 295 capas con entre 64 y 1024 neuronas por capa, resultando en un total de más de 46 millones de parámetros. Asumiendo que cada parámetro contribuye al menos dos operaciones (una multiplicación y una suma), esto da como resultado más de 92 millones de operaciones para procesar un solo dato de entrada. El elevado número de operaciones impone importantes desafíos técnicos para implementar sistemas que requieran alta tasa de procesamiento (o *throughput*) y/o bajo tiempo de respuesta (o *latencia*). Por ejemplo, en aplicaciones de la industria automotriz, la detección de obstáculos debe procesarse en menos de cien milisegundos para asegurar la correcta reacción del conductor o del sistema de conducción autónoma [6].

La implementación de una red feedforward para una aplicación específica comprende los procesos de entrenamiento e inferencia, los cuales se describen brevemente a continuación:

**Entrenamiento** El entrenamiento de una ANN feedforward con aprendizaje supervisado implica ajustar iterativamente los pesos de cada neurona para minimizar el error en sus predicciones sobre un conjunto de datos etiquetados. En cada iteración, una muestra de datos de entrada se propaga a través de la red para generar una predicción (*forward propagation*). A continuación, se calcula el error entre la predicción y el valor esperado mediante una función de pérdida. Este error se utiliza para ajustar los pesos de la red aplicando *backpropagation* junto con un algoritmo de optimización que busca minimizar el error. Este proceso se repite durante múltiples épocas, entendidas como recorridos completos a través del conjunto de datos de entrenamiento, hasta que el error converge. Durante el diseño de la red, es común entrenar varias instancias del modelo con diferentes combinaciones de hiperparámetros y evaluar su desempeño en conjuntos de datos de validación, eligiendo el modelo que mejor generalice a datos nuevos. En la práctica, iterar sobre diferentes estructuras de red y ajustar

sus pesos asociados es computacionalmente demandante, pudiendo demorar desde horas hasta días, dependiendo de la complejidad del modelo [7]. Sin embargo, el entrenamiento ocurre en tiempo de diseño y se realiza una sola vez o con poca frecuencia. Además, es posible acelerar el proceso utilizando sistemas de alto rendimiento, ya sea alojados localmente o distribuidos en la nube, cuando estos recursos están disponibles.

**Inferencia** Una vez determinada la mejor configuración de parámetros e hiperparámetros, la red puede ser utilizada para realizar inferencias, empleando los parámetros ya ajustados para generar predicciones sobre nuevos datos no etiquetados. Durante el proceso de inferencia, tanto la estructura como los parámetros de la red se mantienen fijos, y únicamente se lleva a cabo el proceso de forward propagation, sin necesidad de iteraciones. Aunque el proceso de inferencia tiene una demanda computacional menor que el entrenamiento, se realiza en tiempo de ejecución y suele estar sujeto a requisitos estrictos de tiempo de respuesta y tasa de procesamiento. Además, en escenarios prácticos, la inferencia frecuentemente se lleva a cabo en dispositivos embebidos con recursos limitados y restricciones de consumo energético.

Con respecto a la implementación práctica de modelos de deep learning, existen diversas bibliotecas y herramientas de software, también conocidas como *frameworks*, que proporcionan abstracciones de alto nivel para facilitar la descripción, ajuste y evaluación de redes neuronales. Algunos de los frameworks más populares en la actualidad son PyTorch [8], TensorFlow [9] y Keras [10], los cuales ofrecen diferentes niveles de abstracción para describir los procesos de entrenamiento e inferencia, permitiendo una rápida exploración del espacio de diseño para nuevos modelos de redes y el desarrollo de nuevas aplicaciones. Estas herramientas de alto nivel han fomentado la proliferación de arquitecturas de redes cada vez más complejas, con un creciente número de capas y neuronas, las cuales requieren procesar volúmenes de datos cada vez mayores.

La creciente demanda por aplicaciones más sofisticadas ha incrementado, a su vez, los requerimientos computacionales necesarios para la ejecución de modelos, fomentando la innovación en términos de optimizaciones algorítmicas que permitan una ejecución eficiente en sistemas con capacidad de cómputo limitada o que exploten el paralelismo para reducir los tiempos de procesamiento. En la actualidad, el machine learning es uno de los principales motores que impulsan el desarrollo de nuevas tecnologías computacionales y promueven la innovación en arquitecturas de hardware, especialmente en sistemas de cómputo paralelo. Un claro ejemplo de esta tendencia es el papel de las *Graphics Processing Units* (GPUs), cuyo desarrollo reciente ha sido significativamente impulsado por las necesidades del deep learning. En este contexto, Nvidia, uno de los principales fabricantes de GPUs, ha experimentado un aumento masivo en la demanda de sus productos debido a su capacidad para gestionar las exigentes cargas de trabajo asociadas con el deep learning. De hecho, en junio de 2024, Nvidia se convirtió en la empresa con mayor capitalización bursátil en los Estados Unidos, como resultado de su éxito en este sector [11, 12].

Las GPUs han sido fundamentales para el avance de los modelos de deep learning debido a su capacidad para realizar operaciones de forma simultánea. Esto se debe a que las GPUs están equipadas con cientos de núcleos especializados capaces de trabajar en paralelo, lo que las hace particularmente adecuadas para las operaciones de álgebra lineal entre matrices y vectores de alta dimensionalidad requeridas por las redes neuronales [4]. Una característica distintiva de las GPUs es su capacidad para realizar *context switching*, que les permite alternar rápidamente entre la ejecución de diferentes tareas. Si una tarea se bloquea porque necesita esperar datos de la memoria, la GPU puede cambiar a otra tarea que esté lista para ejecutarse, maximizando el uso de sus núcleos y aumentando la tasa de operaciones por segundo. Esta funcionalidad es especialmente útil para procesar grandes volúmenes de datos y reducir el tiempo necesario para completar tareas asociadas al deep learning [13]. No obstante, el context switching puede derivar en la ejecución de instrucciones fuera de orden, lo

que introduce variabilidad en los tiempos de procesamiento de cada tarea. Esto puede afectar el rendimiento en aplicaciones que requieren tiempos de respuesta acotados y predecibles [14].

En la práctica, el uso eficiente de GPUs en algoritmos de deep learning requiere comprender su modelo de programación y emplear lenguajes como CUDA u OpenCL para aprovechar las características del hardware subyacente. Aunque frameworks de alto nivel como TensorFlow y PyTorch ofrecen soporte para el uso de GPUs, suelen aplicar directrices genéricas que no están completamente optimizadas para explotar todas las propiedades específicas del hardware. Como consecuencia, estas herramientas pueden proporcionar cierta aceleración en comparación con la ejecución en CPU cuando se trabaja con grandes volúmenes de datos; sin embargo, la utilización de los recursos tiende a ser subóptima [15]. Además, una configuración inadecuada puede incrementar los tiempos de inferencia debido a factores como la transferencia de datos entre la memoria principal y la memoria de la GPU, así como la sincronización entre tareas.

Con el propósito de optimizar las operaciones de inferencia en modelos de deep learning en entornos de producción, la compañía Nvidia desarrolló TensorRT [16], una herramienta especializada para sus GPUs, que abarca hardware destinado a computadoras de escritorio, *datacenters* y sistemas embebidos. TensorRT recibe como entrada un modelo previamente entrenado en un framework de alto nivel y genera como salida un archivo ejecutable optimizado para el hardware objetivo. Entre las optimizaciones que realiza TensorRT se incluyen la fusión de capas, la reducción de la precisión numérica de los parámetros, el ajuste de operaciones según el hardware específico y otras configuraciones basadas en directrices definidas por el usuario. Sin embargo, el proceso de optimización y generación del ejecutable es cerrado y propietario, lo que limita al usuario a validar la funcionalidad y evaluar el desempeño únicamente en términos de entrada y salida. En cuanto a su usabilidad, existen numerosos tutoriales que destacan sus beneficios en tareas comunes como la clasificación de objetos en imágenes. No obstante, los resultados dependen significativamente de las directrices proporcionadas por el usuario durante el proceso de optimización, y la efectividad de estas técnicas no está garantizada para otros modelos o plataformas de hardware. Aunque la literatura reciente incluye benchmarks que evalúan las capacidades de aceleración y proponen directrices de uso para TensorRT, estos estudios suelen omitir detalles clave sobre la configuración de TensorRT y el entorno de operaciones, lo que dificulta la reproducibilidad de los resultados y su aplicabilidad a tareas diferentes de los típicos benchmarks de clasificación de imágenes [17–20]. Por otro lado, dada la evolución constante de las plataformas de hardware y el soporte de software, evaluaciones previas no han considerado herramientas de nueva generación, como la serie de GPUs embebidas Jetson Orin, lanzada en marzo de 2023 [21]. Esto deja pendiente la revisión de la compatibilidad y efectividad de las optimizaciones de TensorRT con estas tecnologías más recientes.

El presente trabajo de tesis apunta a realizar una evaluación experimental sistemática para validar los resultados de estudios previos y complementar el estado del arte relacionado con el uso de TensorRT para la optimización de tareas de inferencia, incorporando evaluaciones de nuevas técnicas y aplicaciones. Inicialmente, se llevará a cabo una evaluación basada en las metodologías y conjuntos de datos reportados en la literatura reciente, con el propósito de confirmar hallazgos y prácticas recomendadas derivadas de investigaciones anteriores. Posteriormente, a partir de las observaciones realizadas en el estudio preliminar, se ampliará el alcance de las evaluaciones para incluir nuevas métricas y combinaciones de hardware y software provistas por plataformas de última generación que no han sido previamente exploradas. Estos análisis permitirán derivar directrices generales para el uso eficiente de TensorRT, considerando distintos requerimientos de desempeño y restricciones en recursos computacionales. Adicionalmente, se busca validar la generalidad de los resultados obtenidos al aplicar las técnicas propuestas en entornos diferentes a los benchmarks genéricos típicamente utilizados en la literatura reciente.

## 1.2. Planteamiento del problema

Para optimizar el uso de los recursos disponibles en las GPUs durante el proceso de inferencia, TensorRT opera en capas de abstracción más bajas que aquellas utilizadas en los frameworks de alto nivel comúnmente empleados para describir modelos de aprendizaje profundo. El proceso de optimización y generación de código ejecutable es propietario y cerrado, lo que significa que el usuario solo puede proporcionar directivas generales para guiar la optimización, las cuales pueden ser consideradas u omitidas por la herramienta.

Además, la optimización para el hardware específico se basa en estadísticas recopiladas durante la generación del modelo optimizado, las cuales dependen de las condiciones de operación, como el voltaje de alimentación o la carga de trabajo en el sistema en ese momento. Por esta razón, los resultados pueden variar entre distintas ejecuciones, incluso en una misma plataforma.

Si bien el fabricante proporciona tutoriales y ejemplos que ocultan la complejidad de los procesos de optimización subyacentes y facilitan el uso de la herramienta, su utilización efectiva presenta ciertos desafíos, entre los cuales se incluyen los siguientes:

- La correcta especificación de directivas para el proceso de optimización requiere que el usuario tenga, al menos, un conocimiento intermedio sobre la estructura interna y las operaciones realizadas por la red neuronal, el modelo de programación de GPUs, la arquitectura del hardware objetivo y las propiedades específicas de TensorRT, a fin de explotar eficientemente las capacidades disponibles en el hardware.
- Los modelos descritos en alto nivel ofrecen abstracciones que pueden no estar completamente soportadas o ser incompatibles con TensorRT. Es fundamental comprender estas restricciones para determinar si un modelo puede ser optimizado directamente o si es necesario adaptar su código antes de la optimización.
- TensorRT ha demostrado ser altamente eficiente en benchmarks de clasificación y detección de objetos en imágenes y videos. Sin embargo, muchas de las evaluaciones reportadas en la literatura se basan en conjuntos de datos estandarizados cuyas propiedades están bien documentadas, mientras que existe poca evidencia sobre su efectividad en aplicaciones con bases de datos más diversas o en otros tipos de redes neuronales, como aquellas utilizadas en tareas de regresión y análisis de series de tiempo.
- La rápida evolución de los modelos de redes neuronales, el hardware de GPU y las versiones de software genera la necesidad de realizar verificaciones periódicas para asegurar la compatibilidad de las técnicas aplicadas en un modelo con las nuevas versiones.

En la práctica, los desafíos previamente planteados implican que el uso adecuado de TensorRT para la optimización de tareas de inferencia presenta una curva de aprendizaje empinada. Además, la correcta migración y optimización de modelos existentes puede convertirse en un proceso complejo y laborioso, con dificultades en cuanto a su reproducibilidad y extensibilidad a nuevos conjuntos de datos o aplicaciones.

En base a lo planteado previamente, se establece el problema a tratar en esta tesis como sigue:

*"Dado un modelo previamente entrenado de una red de deep learning, se requiere implementar y documentar un flujo de trabajo para su portabilidad a TensorRT, considerando distintas configuraciones de operación. Estas configuraciones incluyen la plataforma de hardware utilizada, el modo de consumo energético del sistema y los parámetros de optimización de TensorRT, tales como el batch size, la cuantización y el nivel de optimización. Los modelos optimizados serán evaluados en*

*estas distintas configuraciones, comparando métricas como latencia, throughput y propiedades estructurales del modelo. Además, se busca validar la versatilidad de TensorRT en la optimización de aplicaciones con arquitecturas de red y conjuntos de datos personalizados, distintos de las configuraciones estandarizadas comúnmente empleadas en benchmarking.”*.

## 1.3. Alcances y contribuciones

Este trabajo se enmarca dentro de los siguientes alcances:

- Se asume el uso de modelos de redes neuronales previamente entrenados y se espera que puedan reutilizarse sin necesidad de un reentrenamiento. En las evaluaciones realizadas, se verifica que las redes optimizadas mediante TensorRT conserven su funcionalidad con respecto al modelo base no optimizado.
- Se consideran exclusivamente redes neuronales feedforward con aprendizaje supervisado, ya que son las más utilizadas en estudios de benchmarking según la literatura. Este tipo de redes son ampliamente utilizadas en el contexto de procesamiento de imágenes y, además, cuentan con el mayor soporte documentado en TensorRT.
- Este trabajo se desarrolla con un enfoque en las plataformas embebidas de la familia Jetson Orin, ya que representan la generación más reciente y cuentan con el mayor soporte de software. Adicionalmente, se evalúan plataformas de generaciones anteriores, como la Jetson Xavier y computadores de escritorio, con el fin de asegurar la compatibilidad y validar el estado del arte.
- Para las evaluaciones experimentales, se utilizan las versiones más recientes del software de Nvidia (JetPack, TensorRT, CUDA, etc.) soportadas por el hardware correspondiente hasta marzo de 2024. La versión más reciente de JetPack en esa fecha es JetPack 6.0 para la familia Jetson Orin y JetPack 5.1 para las Jetson Xavier.

Las principales contribuciones de este trabajo se resumen a continuación:

- Validación y extensión del estado del arte en la optimización de la inferencia usando TensorRT, estableciendo directrices para el uso óptimo de estas herramientas en configuraciones de software y hardware que no han sido reportadas en la literatura previa.
- Directrices, metodologías y hallazgos para la optimización de la inferencia con TensorRT en aplicaciones que van más allá de los casos de uso tradicionales en benchmarking.
- Documentación detallada del proceso de implementación de TensorRT para benchmarking y evaluación de aplicaciones en distintos entornos, con un enfoque en la replicabilidad de los experimentos. Todo el código y la documentación serán publicados en repositorios públicos para garantizar la reproducibilidad y facilitar futuras extensiones del trabajo.

## 1.4. Organización de la Tesis

El contenido de esta tesis se organiza de la siguiente manera:

- El Capítulo 2 presenta los antecedentes, incluyendo una descripción general de TensorRT, el modelo de programación CUDA y trabajos relacionados.
- El Capítulo 3 analiza la evaluación de configuraciones reportadas en el estado del arte, describiendo métricas, entornos de operación y resultados.
- El Capítulo 4 explora la evaluación con nuevas configuraciones y métricas, incluyendo experimentos con diferentes configuraciones de batch size, niveles de optimización y modos de consumo.
- El Capítulo 5 aborda la evaluación de casos de estudio, como la segmentación de imágenes submarinas de salmones y la regresión para la pirometría del hollín.
- El Capítulo 6 presenta la conclusión de los resultados obtenidos y su impacto en la optimización de inferencia con TensorRT, así como enfoques a futuro para complementar el trabajo realizado.

Capítulo 2

# ANTECEDENTES

Este capítulo se centra en definir la terminología asociada al diseño e implementación de redes neuronales para deep learning. En primer lugar, se presentan términos generales relacionados con el entorno de trabajo, abarcando el software y hardware utilizados en la construcción e implementación de aplicaciones de machine learning. Asimismo, se incluye una breve explicación de los conceptos vinculados a CUDA, necesarios para comprender los temas que se abordan posteriormente. Seguidamente, se describe TensorRT y su funcionamiento. Por último, se ofrece una reseña de trabajos relacionados con el tema de esta tesis, con el objetivo de proporcionar una visión general de la literatura actual y resaltar los aspectos que no son abordados por esta.

## 2.1. Overview y terminologia

Como primer paso para proporcionar el contexto técnico y definir la terminología utilizada en este reporte, se presenta la Tabla 2.1. Esta tabla ofrece una guía propuesta en este trabajo para ilustrar las diferentes capas de abstracción involucradas en el diseño e implementación de redes neuronales. Además, incluye ejemplos de herramientas de software y hardware asociadas a cada capa, las cuales se definen a continuación.

**Aplicación** La aplicación objetivo define el problema que se busca resolver mediante el uso de deep learning, a partir del cual se derivan los requerimientos funcionales. Ejemplos de estos requerimientos incluyen detección de objetos, clasificación, segmentación y regresión. Además, en la capa de aplicación se identifican los requerimientos de desempeño o no funcionales, como el throughput y la latencia.

**Dataset** El dataset es un conjunto de datos representativos que contiene ejemplos de entrada y salida deseados, utilizado para entrenar modelos de redes en un entorno de aprendizaje supervisado, que es el enfoque de interés en este trabajo. Una vez establecidos los requisitos funcionales y no funcionales de la aplicación objetivo, es fundamental contar con un dataset estadísticamente representativo del espacio de entradas de la aplicación. Esto asegura que el modelo pueda aprender de manera efectiva y generalizar correctamente.

**Modelo** El modelo de machine learning es una representación matemática que define cómo los datos de entrada se transforman en las salidas deseadas. En el contexto del aprendizaje supervisado, el modelo aprende a partir de un dataset etiquetado, ajustando sus parámetros internos mediante un proceso iterativo de entrenamiento. Dependiendo de la naturaleza del problema y del tipo de datos, los modelos pueden incluir redes neuronales profundas, árboles de decisión, máquinas de soporte vectorial, entre otros. Para la aplicación objetivo, el modelo seleccionado debe capturar

Tabla 2.1: Niveles populares de software y hardware, adaptado de [18].

| Capa | Ejemplos |
|---|---|
| **Aplicación** | Visión por computadora<br>Control Automático<br>Reconocimiento de voz<br>Conducción autónoma |
| **Dataset** | ImageNet [7]<br>COCO<br>BERT<br>LibriSpeech<br>Berkeley DeepDrive |
| **Modelo** | ResNet<br>MobileNet<br>YOLO<br>U-Net |
| **Framework** | PyTorch [8]<br>TensorFlow [9]<br>Keras [10] |
| **Graph Format** | ONNX [22]<br>NNEF |
| **Inference Optimizer** | TensorRT [16]<br>ARM Compute Library<br>Intel OpenDNN<br>AMD ML Open<br>Xilinx Vitis AI |
| **Hardware** | GPU<br>CPU<br>FPGA<br>Aceleradores especializados (TPU, NPU, etc.) |

las características relevantes de los datos y generalizar de manera efectiva a nuevas muestras no vistas, optimizando tanto la precisión como la eficiencia de inferencia de acuerdo con los requisitos establecidos.

**Framework** El framework es un entorno de software que proporciona herramientas, librerías especializadas y estructuras necesarias para el desarrollo, entrenamiento y evaluación de modelos de machine learning. Estos frameworks ofrecen distintos niveles de abstracción, facilitando tanto la exploración de hiperparámetros como el entrenamiento y la inferencia de modelos, ya sean redes preentrenadas o diseñadas a medida. Dada una aplicación y dataset objetivo, los frameworks permiten implementar modelos adecuados para cumplir con la tarea requerida.

**Graph format** El graph format es una representación estandarizada de un modelo de machine learning que facilita su interoperabilidad y transferencia entre diferentes frameworks y herramientas. Esta representación se estructura como un grafo computacional (ver Figura 2.1), permitiendo visualizar y comprender cómo los datos son procesados a través del modelo. El grafo computacional consta de los siguientes elementos:

- **Nodos**: representan las capas, funciones u operaciones de la red. En la Figura 2.1, se identifican con un color y el nombre de la capa correspondiente. Por ejemplo, las capas convolucionales se muestran en azul.

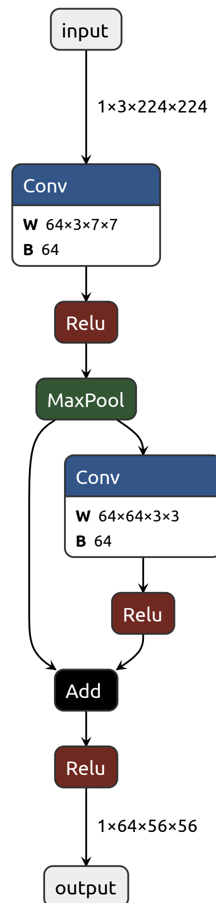



Fig. 2.1: Representación de un grafo computacional para un modelo de machine learning basado en una red residual simple, extraída utilizando Netron [23], ilustra el flujo de datos a través de operaciones como convoluciones (*Conv*), funciones de activación (*ReLU*) y *pooling* (*MaxPool*).

- **Aristas**: simbolizan los tensores de la red y 'transportan' los datos desde la salida de un nodo hasta la entrada del siguiente. En la Figura 2.1, se representan como flechas que indican la dirección del flujo de datos y, junto a ellas, se muestra el tamaño del tensor.
- **Parámetros**: corresponden a los pesos y sesgos de la red, ajustados durante el proceso de entrenamiento. En la Figura 2.1, se denotan como **W** (pesos) y **B** (sesgos), indicando su dimensionalidad a un costado.

Una vez entrenado un modelo para una aplicación y dataset específicos, puede ser necesario exportarlo a un formato estándar mediante herramientas de la capa de graph format. Esto permite su interoperabilidad con otros frameworks y software de optimización, como la integración de PyTorch con TensorRT.

**Inference optimizer** Un inference optimizer es una herramienta o conjunto de técnicas utilizadas para mejorar la eficiencia y el rendimiento de un modelo de machine learning durante la fase de

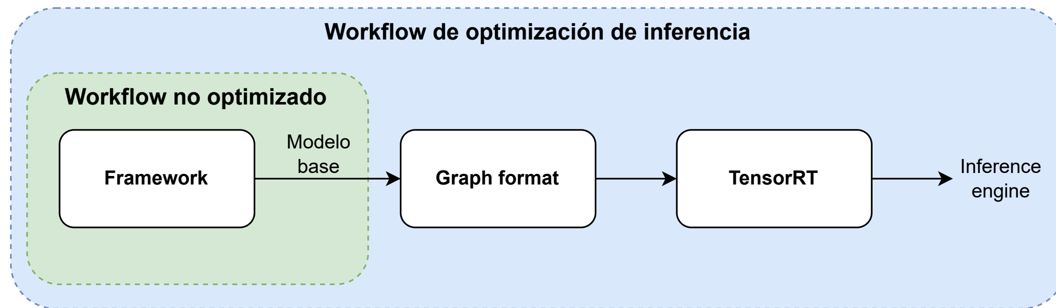



Fig. 2.2: Flujo de trabajo para la optimización de la inferencia usando TensorRT.

inferencia. Una vez que el modelo ha sido entrenado, puede ser procesado por un inference optimizer, que aplica diversas técnicas de optimización, como la compresión del modelo y el mapeo de operaciones a primitivas específicas del hardware disponible. Tras este proceso, el inference optimizer genera un modelo optimizado representado en lo que se conoce como **inference engine**, que incluye información sobre cómo ejecutar las operaciones de la red en el hardware objetivo.

**Hardware** El hardware objetivo se refiere al tipo específico de dispositivo en el que se ejecutará el modelo de machine learning durante la fase de inferencia. Este puede variar ampliamente según las necesidades de la aplicación, el entorno de despliegue y los requisitos de rendimiento y eficiencia. Los tipos de hardware especializado incluyen GPUs, CPUs, FPGAs y aceleradores específicos, como TPUs y NPUs. Entre estas opciones, las *Graphics Processing Units* (GPUs) destacan como las más accesibles y populares debido a sus capacidades de cómputo paralelo, esenciales para las aplicaciones de machine learning [14]. En este trabajo, se hace énfasis en el uso de GPUs de Nvidia, las cuales se analizan en detalle en la Sección 2.2.

El proceso de implementación de una red neuronal utilizando un framework tradicional (por ejemplo, PyTorch, TensorFlow, Caffe o TensorFlow Lite) sin emplear herramientas de optimización se conoce como **workflow no optimizado** y produce un **modelo base**. Posteriormente, considerando las capas de abstracción descritas en la Tabla 2.1, se establece un **workflow de optimización de inferencia**. Este workflow consiste en una secuencia de pasos para optimizar la inferencia de un modelo base mediante la integración de un framework, un graph format y un inference optimizer. El resultado final de este proceso es la creación de un **inference engine** (ver Figura 2.2).

A continuación, se presentan algunos ejemplos de workflows de optimización de inferencia reportados en la literatura reciente, los cuales se utilizan para aumentar la eficiencia computacional durante el proceso de inferencia:

- **PyTorch-TensorRT**: Describe el proceso proporcionado por PyTorch para optimizar modelos utilizando TensorRT. En este caso, el *graph format* está integrado en las funcionalidades de PyTorch y no se menciona explícitamente [17, 24].

- **TensorFlow-TensorRT**: Explica el proceso ofrecido por TensorFlow para optimizar modelos mediante TensorRT. Aquí, el *graph format* también está integrado dentro de TensorFlow, aunque no se detallan sus funcionalidades, y el nivel de configurabilidad disponible para el usuario es menor [19].

- **PyTorch-ONNX-TensorRT**: Detalla la transición de un modelo previamente entrenado en PyTorch al formato ONNX (*graph format*), que luego se optimiza utilizando TensorRT [17,20].

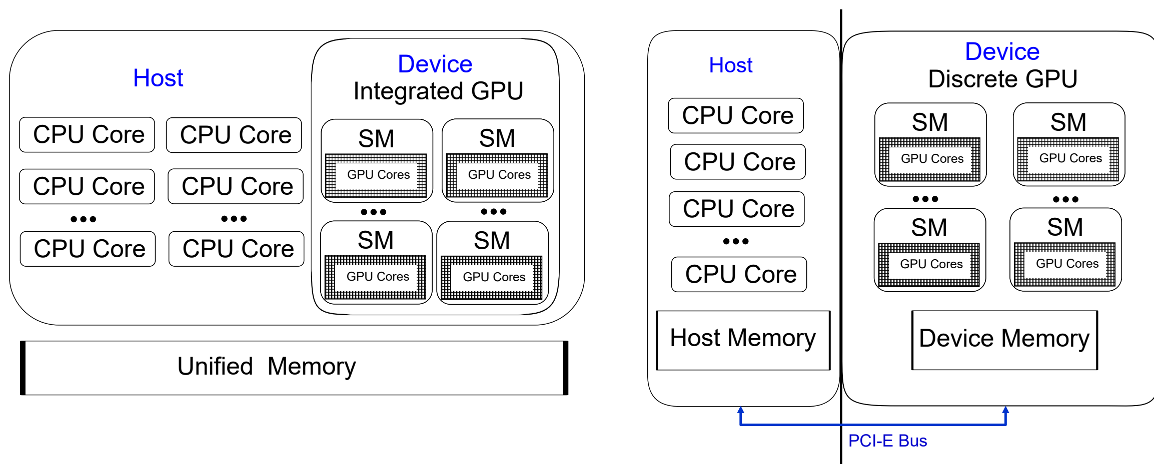



Fig. 2.3: Diagrama de comunicación entre *host* y *device*: GPU integrada (izquierda) y GPU discreta (derecha). Adaptado de [14].

- **ONNX-Runtime**: Describe el proceso mediante el cual ONNX recibe un modelo base de un framework compatible (por ejemplo, PyTorch o TensorFlow). Utilizando funcionalidades que integran TensorRT, ONNX-Runtime optimiza la inferencia de dicho modelo [17].

Existen múltiples combinaciones de plataformas de hardware y software ofrecidas por diversos proveedores en el mercado actual, así como una amplia variedad de inference optimizers. Este trabajo se enfoca específicamente en evaluar un workflow asociado a TensorRT para la generación de inference engines, optimizados para su ejecución en GPUs de Nvidia.

## 2.2. CUDA Programming Model

Como se mencionó en la sección anterior, las GPUs de Nvidia serán el hardware objetivo en las evaluaciones realizadas en este trabajo. Las aplicaciones que dependen del deep learning suelen ser tareas altamente paralelizables, lo que les permite aprovechar la arquitectura de las GPUs para ejecutarse de manera más eficiente que en plataformas como las CPUs [15]. Para habilitar el desarrollo de aplicaciones de propósito general en sus GPUs, Nvidia creó CUDA, una plataforma computacional y modelo de programación que permite a los desarrolladores explotar el poder del procesamiento paralelo de las GPUs de Nvidia, en un enfoque conocido como *General-Purpose Computing on GPUs* (GPGPU). Nvidia proporciona acceso a CUDA mediante librerías y APIs disponibles en lenguajes de programación estándar como C/C++, Fortran y Python.

Dadas las capacidades de CUDA y para comprender algunas de las etapas de optimización descritas en la Sección 2.3, es fundamental entender el funcionamiento y ciertos conceptos clave de este modelo de programación. Una descripción exhaustiva o un tutorial detallado sobre los aspectos técnicos de CUDA está fuera del alcance de este trabajo. En su lugar, se ofrece una visión general de los conceptos necesarios para entender las optimizaciones y análisis presentados. Para una explicación más completa, se recomienda consultar la documentación oficial de CUDA [13] o referencias especializadas como el libro de Kirk y Hwu sobre programación paralela en GPUs [25].

El modelo de programación CUDA es un modelo heterogéneo que aprovecha tanto la CPU como la GPU. En CUDA, el término *host* se refiere a la CPU y su memoria, mientras que *device* hace

referencia a la GPU y su memoria. La Figura 2.3 ilustra los diagramas de comunicación entre host y device en GPUs discretas e integradas. Las GPUs discretas son aquellas que están en un chip separado de la CPU y se conectan al sistema mediante buses como PCIe, como es común en las PCs de escritorio. En contraste, las GPUs integradas se encuentran dentro del mismo chip que la CPU, compartiendo memoria y otros recursos del sistema. Un ejemplo representativo de estas últimas son las plataformas Jetson de Nvidia, diseñadas para dispositivos embebidos y sistemas de bajo consumo energético.

A continuación, se describen algunos de los elementos involucrados en el procesamiento paralelo en GPUs:

- **CPU Cores**: Son las unidades de procesamiento de una CPU responsables de ejecutar instrucciones. Los CPU Cores están especializados para ejecutar tareas secuenciales, siendo ideales para realizar una operación individual de la manera más rápida posible.
- **GPU Cores**: Son las unidades de procesamiento dentro de una GPU. A diferencia de los CPU Cores, los GPU Cores están optimizados para realizar operaciones de cálculo en paralelo, lo que los hace especialmente eficientes en tareas que requieren un procesamiento masivo de datos, como el renderizado de gráficos o algoritmos de machine learning. Para explotar el paralelismo, los datos procesados deben ser independientes, es decir, las operaciones no deben depender unas de otras, lo que permite que los GPU Cores los procesen simultáneamente. Cabe destacar que los GPU Cores son fundamentalmente diferentes de los CPU Cores: las GPUs no pueden, por ejemplo, cargar sistemas operativos ni gestionar tareas autónomamente. Funcionan como coprocesadores, ejecutando las instrucciones enviadas desde la CPU.
- ***Streaming Multiprocessor* (SM)**: Es una unidad dentro de las GPUs de Nvidia que agrupa varios GPU Cores junto con otras unidades funcionales. Los SMs tienen la capacidad de ejecutar múltiples *threads* de procesamiento simultáneamente, coordinando la ejecución paralela de un gran número de operaciones.

En el desarrollo de este trabajo se emplean tanto GPUs discretas como integradas. Una GPU integrada, ilustrada en el diagrama de la izquierda de la Figura 2.3, forma parte de una implementación *system-on-chip* (SoC). Esta integra la GPU, los *CPU Cores*, la memoria (*Unified Memory*) y los conectores externos en una pequeña computadora de una sola placa. La GPU integrada comparte recursos de hardware, como la memoria, con los *CPU Cores*. Por otro lado, las GPUs discretas, representadas en el diagrama de la derecha de la Figura 2.3, consisten únicamente en los *Streaming Multiprocessors* (SMs) y la memoria local del dispositivo (*device memory*), típicamente empaquetados en una tarjeta para su instalación en una ranura de expansión PCIe de la placa base de una computadora. A diferencia de las GPUs integradas, las GPUs discretas no comparten memoria con el host; en su lugar, utilizan el bus PCIe para transferir datos entre la memoria del host (*host memory*) y la memoria del dispositivo [14].

Un programa CUDA comienza ejecutándose como una tarea o proceso en el host y depende del device para llevar a cabo las tareas de cálculo. La estructura general de un programa CUDA que interactúa con la GPU incluye los siguientes pasos: (i) asignar memoria para el uso de la GPU, (ii) transferir los datos de entrada desde la memoria de la CPU a la memoria de la GPU, (iii) iniciar la ejecución de un programa en la GPU para procesar los datos, (iv) copiar los resultados de la memoria de la GPU de regreso a la memoria de la CPU, y (v) liberar la memoria no necesaria. Las funciones o programas que se ejecutan en la GPU se denominan **kernels**, y el proceso mediante el cual el host ordena al device ejecutar un kernel se denomina **kernel launch**. Cada kernel launch introduce retardos debido a la configuración y la comunicación entre el host y el device. Por esta

razón, reducir la cantidad de kernel launches y seleccionar los kernels más adecuados para el hardware objetivo son optimizaciones que típicamente realizan los inference optimizers, como se analizará en la Sección 2.3.1.

En la mayoría de los programas CUDA, surge la necesidad de sincronizar las operaciones entre la GPU y la CPU. Por ejemplo, una operación en la CPU puede requerir esperar los resultados de una operación en la GPU. Esta sincronización puede provocar un bloqueo temporal en la ejecución de otras tareas hasta que los recursos vuelvan a estar disponibles. Como se mencionó en la Sección 1.1 al discutir las nociones sobre GPUs, una de las ventajas del uso de GPUs frente a otras herramientas de hardware es su capacidad para realizar *context switching* de manera eficiente entre diferentes tareas. Sin embargo, esta ventaja también implica la necesidad de sincronización y el riesgo de *stalling* o bloqueo de operaciones. El bloqueo de una operación que utiliza la GPU puede ocasionar pérdidas en la utilización de los recursos computacionales. Prever el stalling de tareas que emplean la GPU no es una tarea sencilla y puede alterar los tiempos de respuesta, afectando potencialmente la ejecución de aplicaciones que requieren tiempos de respuesta acotados.

Según estudios realizados sobre las GPUs de Nvidia y CUDA [14, 26, 27], se ha identificado que aspectos como la planificación, sincronización, ejecución y bloqueo de tareas en la GPU no están completamente documentados ni son accesibles al público. Esto obliga a los usuarios y desarrolladores a basarse en evidencia empírica para inferir el comportamiento de los programas desarrollados con CUDA. En este contexto, los desarrolladores carecen de un modelo confiable que describa el comportamiento de las GPUs, debido a la naturaleza de 'caja negra' que caracteriza a CUDA [14, 26]. Esto limita su capacidad para desarrollar aplicaciones que cumplan con requisitos de tiempos de respuesta.

Para optimizar de manera efectiva la inferencia de modelos previamente entrenados en plataformas de Nvidia, es fundamental comprender el funcionamiento de CUDA y el hardware subyacente, lo cual se ha demostrado como una tarea compleja. Para simplificar este proceso, Nvidia desarrolló TensorRT, un inference optimizer diseñado específicamente para esta finalidad. Al haber sido desarrollado por Nvidia, TensorRT aprovecha el conocimiento interno de la compañía sobre CUDA y sus plataformas de hardware para optimizar la inferencia. Esto, sumado a la escasa documentación disponible sobre CUDA mencionada anteriormente, justifica que no sea necesario profundizar en su funcionamiento para las evaluaciones específicas realizadas en este trabajo. TensorRT promete mejores resultados en la ejecución de la inferencia de modelos previamente entrenados en comparación con los frameworks tradicionales, sin la necesidad de programar directamente en CUDA. Los detalles de TensorRT se explicarán en la siguiente sección.

## 2.3. Descripción de TensorRT

TensorRT es un *Software Development Kit* (SDK), clasificado como uno de los inference optimizers descritos en la Sección 2.1, diseñado específicamente para generar inference engines a partir de modelos previamente entrenados, optimizados para una ejecución rápida y eficiente en las GPUs más recientes de Nvidia (véase Sección 1.9, *Hardware Support Lifetime* [28]). Este SDK realiza optimizaciones basadas en propiedades específicas del hardware objetivo, el cual es tecnología propietaria de Nvidia. Por tanto, TensorRT también es un software cerrado, y los detalles de su funcionamiento interno no están disponibles públicamente. No obstante, la documentación oficial de Nvidia, junto con estudios académicos basados en evaluaciones empíricas, ofrece descripciones generales de las optimizaciones que lleva a cabo. Además, TensorRT se integra con los frameworks de machine learning más populares, proporcionando documentación extensa y ejemplos prácticos que facilitan su adopción. Existe evidencia empírica sustancial que indica que los inference engines generados por

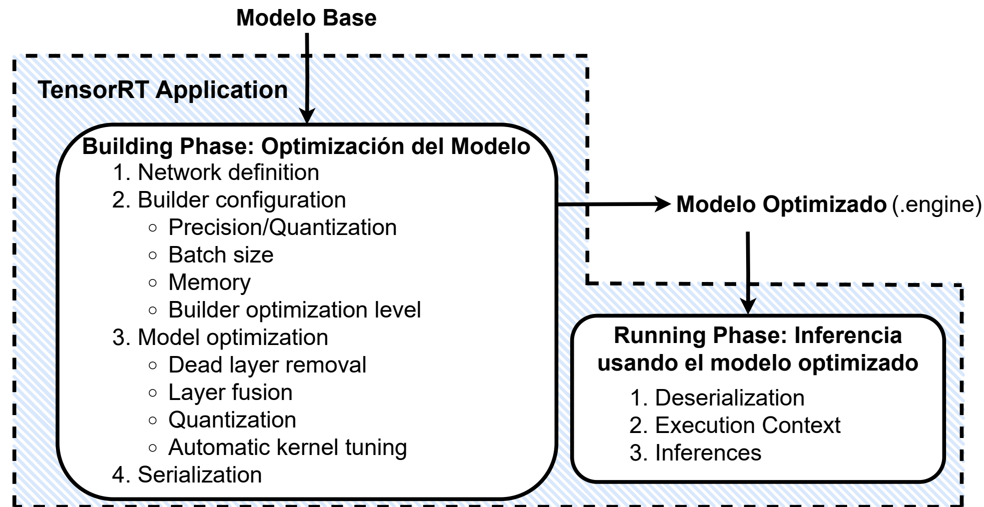



Fig. 2.4: Flujo de una TensorRT Application.

TensorRT representan una de las soluciones más eficientes para ejecutar modelos de redes neuronales en GPUs de Nvidia [17–20]. Finalmente, debido a la posición dominante de Nvidia en el mercado de GPUs, TensorRT se ha consolidado como la herramienta más popular para la optimización de tareas de inferencia.

TensorRT proporciona APIs tanto para C++ como para Python, que permiten programar una **TensorRT Application**, término que se utilizará en este trabajo para referirse a la implementación de un inference optimizer mediante las funciones proporcionadas por TensorRT. Una TensorRT Application tiene la capacidad de optimizar modelos y realizar inferencias (véase Figura 2.4). Los pasos para implementar una TensorRT Application se describen en la documentación oficial de TensorRT [28]; sin embargo, no se presentan de manera exhaustiva. Esto se debe a que los pasos específicos pueden variar en función de factores como la aplicación específica, el nivel de optimización requerido, el framework utilizado, entre otros [29]. Asimismo, dependiendo del workflow empleado para implementar la optimización, los resultados pueden diferir incluso para un mismo modelo base previamente entrenado.

La Figura 2.4 ilustra el flujo de una TensorRT Application. Este proceso comienza con un **modelo base**, que da inicio a la **building phase**, donde se optimiza la red ingresada para una GPU específica. Luego, en una segunda etapa conocida como **running phase**, se utiliza el modelo optimizado para ejecutar inferencias. Las siguientes subsecciones describen estas etapas con mayor detalle.

El contenido que se presenta a continuación no tiene como objetivo ser un tutorial exhaustivo ni una guía paso a paso. Su propósito principal es proporcionar un contexto que facilite el uso de la herramienta, permita comprender las decisiones de diseño tomadas y asegure la reproducibilidad de los resultados que se discutirán en los capítulos subsecuentes.

### 2.3.1. Building phase

En la building phase, TensorRT desempeña el papel de **builder**, encargado de optimizar la red y producir un *inference engine*. La Figura 2.4 muestra los pasos principales de esta etapa, los cuales

se describen brevemente a continuación.

### Network definition

El proceso de la building phase comienza con la generación de una network definition, una representación interna exclusiva de TensorRT que incluye información detallada sobre los tipos de capas (como convolucionales, totalmente conectadas, de activación, entre otras), los parámetros de cada capa, y la definición de las entradas y salidas de la red junto con sus dimensiones. Es importante señalar que los tensores no marcados como entradas o salidas se consideran valores transitorios que el builder puede optimizar.

La network definition puede obtenerse automáticamente mediante el *parser* integrado en TensorRT, compatible con el graph format ONNX. Alternativamente, TensorRT permite generar una network definition de forma manual utilizando funciones como *layer* y *tensor*, que permiten construir la red capa por capa. Sin embargo, este proceso manual está escasamente documentado y puede ser tedioso, por lo que el método más común para obtener la network definition es a través del uso de ONNX [28].

Luego de obtener la network definition, es necesario describir la builder configuration, etapa que se detalla a continuación.

### Builder configuration

Previo a la optimización del modelo, es necesario crear una configuración del builder. Esta configuración se puede modificar para personalizar la optimización del modelo. El ajuste de la configuración puede afectar el uso de recursos, la calidad de los resultados y la viabilidad del proceso de optimización. Los parámetros configurables están listados en la documentación online [30]. Los parámetros más importantes incluyen los siguientes:

- **Precision/quantization**: Este parámetro controla cómo se representan los valores numéricos de los parámetros del modelo. Una mayor precisión, como `fp32` (punto flotante de 32 bits), garantiza cálculos más exactos, pero aumenta el consumo de recursos computacionales. TensorRT permite reducir la precisión de los parámetros del modelo a `fp16` (punto flotante de 16 bits) o cuantizarlos a `int8` (enteros de 8 bits), lo que puede mejorar la eficiencia a costa de una posible pérdida de precisión en la inferencia. En el resto del documento, este parámetro se denominará **cuantización**.
- **Batch size**: Este parámetro define la cantidad de muestras que se procesan juntas en una sola iteración. Un batch size mayor puede incrementar la cantidad de inferencias realizadas por unidad de tiempo, pero también requiere más memoria. Es posible configurar un batch size estático, lo que implica que la optimización del modelo se realizará exclusivamente para ese tamaño fijo, y el inference engine resultante solo podrá operar con dicho tamaño al ejecutar inferencias. Alternativamente, se puede optar por un batch size dinámico, que permite al inference engine realizar inferencias con una cantidad variable de muestras de entrada dentro de un rango definido en la configuración.

  En este documento, se utilizarán los siguientes términos para mayor claridad:

  - **Batch size de inferencia**: La cantidad de muestras de entrada que el modelo procesa durante una inferencia.
  - **Batch size de configuración**: Indica si el inference engine fue configurado con un batch size **estático** o **dinámico**.

Para optimizar un modelo con un **batch size de configuración dinámico**, es necesario establecer parámetros adicionales: **batch size mínimo**, **óptimo** y **máximo**. Estos valores, definidos por el usuario, establecen los límites del rango de tamaños de batch con los que el inference engine podrá operar. Es importante señalar que el valor definido como batch size óptimo no garantiza que sea el tamaño más eficiente o ideal en todos los casos; este término es simplemente la denominación utilizada por Nvidia para indicar el valor en el que el builder centrará sus esfuerzos de optimización.

- **Memory**: Este parámetro define el límite de memoria que cada capa del modelo puede utilizar durante el proceso de optimización necesario para generar el engine. Durante el proceso de generación del engine se crean y prueban múltiples instancias del modelo, lo que incrementa los requerimientos de memoria en función del tamaño del modelo y las opciones de optimización seleccionadas. Si el uso de memoria supera el límite configurado, el proceso falla y no se genera el engine. Por defecto, el parámetro de memoria se establece como la memoria máxima disponible en el dispositivo. Sin embargo, puede reducirse si el usuario necesita, por ejemplo, realizar múltiples optimizaciones simultáneamente en un mismo dispositivo. Cabe destacar que este parámetro aplica únicamente durante el diseño y la generación del engine, y no está relacionado con la memoria requerida para la inferencia, donde se utiliza el engine generado.

- **Builder optimization level**: Este parámetro define el nivel de optimización que el builder empleará para buscar y aplicar mejoras al modelo. El valor del parámetro es un número entero en el intervalo de 0 a 5, con un valor predeterminado de 3. A continuación, se detalla el comportamiento del builder según el nivel seleccionado:

  - **Nivel 0:** Habilita la compilación más rápida al deshabilitar la generación dinámica de kernels y seleccionar la primera táctica que se ejecute con éxito. No utiliza caché de tiempos.
  - **Nivel 1:** Ordena las tácticas disponibles mediante heurísticas y prueba únicamente las más prometedoras para seleccionar la óptima. Si se genera un kernel dinámico, su nivel de optimización es bajo.
  - **Nivel 2:** Ordena las tácticas disponibles por heurísticas, pero prueba únicamente las tácticas más rápidas para determinar la mejor.
  - **Nivel 3:** Aplica heurísticas para decidir si se utiliza un kernel precompilado estático o si es necesario compilar uno dinámicamente.
  - **Nivel 4:** Siempre compila un kernel dinámico.
  - **Nivel 5:** Compila un kernel dinámico y lo compara con kernels estáticos para determinar el mejor.

  Cabe destacar que estas definiciones han sido extraídas de la documentación [30], la cual no proporciona mayores detalles sobre los kernels mencionados.

  El nivel de optimización configurado en este parámetro tiene un impacto significativo en la etapa de automatic kernel tuning, descrita en la subsección siguiente. Elegir un nivel más alto permite al builder probar una mayor variedad de combinaciones y tipos de kernels en el hardware objetivo, a cambio de un mayor tiempo de optimización.

- **Iterations**: Este parámetro define la cantidad de iteraciones que se realizan durante la etapa de automatic kernel tuning, descrita en la subsección siguiente. En esta etapa, el builder evalúa el tiempo que toman diferentes configuraciones de kernels y selecciona aquella que minimiza el tiempo de ejecución para cada capa evaluada. De manera predeterminada, el

builder mide el tiempo de cada configuración una sola vez. Sin embargo, al ajustar el parámetro Iterations, es posible hacer que el builder base su decisión en el tiempo promedio de ejecución obtenido tras múltiples iteraciones por cada opción. Un mayor número de iteraciones puede ayudar a la convergencia del inference engine generado, ya que ayuda a mitigar el efecto del no determinismo inherente a la etapa de automatic kernel tuning.

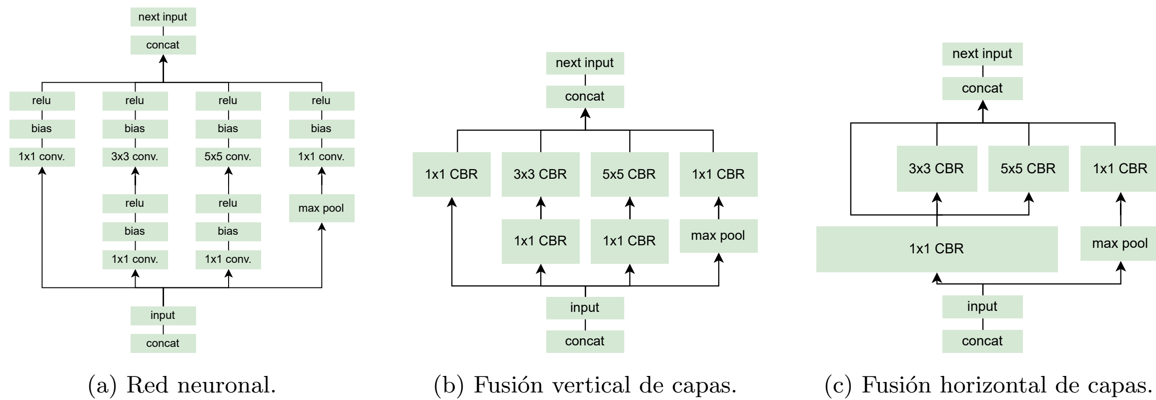
- **Dimensión Dinámica**: De manera predeterminada, TensorRT configura dimensiones estáticas para las entradas del modelo, las cuales se definen por el número de canales, el ancho y el alto. Una vez optimizado el modelo con dimensiones estáticas, este no aceptará entradas con dimensiones fuera de las especificadas. Sin embargo, para permitir la optimización de un modelo con **dimensiones dinámicas** en la altura y el ancho de las entradas, es necesario definir tres parámetros adicionales: **dimensión mínima**, **dimensión óptima** y **dimensión máxima**. Estos valores, establecidos por el usuario, determinan el rango de tamaños de entrada con los que el inference engine podrá operar. Cabe destacar que la **dimensión óptima** no necesariamente corresponde al tamaño más eficiente o ideal en todos los casos. Este término es simplemente la denominación utilizada por Nvidia para indicar el valor en torno al cual el builder enfocará sus esfuerzos de optimización.

Una vez especificadas las configuraciones, se puede optimizar el modelo.

### Model optimization

Luego de tener la configuración del builder, así como la network definition, se pueden aplicar las optimizaciones al modelo. Para ello, el builder elimina cálculos redundantes, combina y reordena operaciones. De manera opcional, puede reducir la precisión de los parámetros, transformándolos de `fp32` a `fp16` o cuantizando los valores de `fp32` a `int8`. Además, evalúa el tiempo de ejecución de múltiples implementaciones para cada capa y calcula una planificación óptima que minimice el costo combinado de las ejecuciones de los kernels asociados a cada capa. Estas optimizaciones se detallan a continuación.

- **Dead layer removal**: En esta etapa, se eliminan las capas cuyas salidas no son utilizadas. Esto puede ocurrir cuando algunos pesos convergen a valores cercanos a cero, haciendo que dichas capas no contribuyan significativamente al valor de salida.
- **Vertical fusion**: Generalmente, la ejecución de un kernel es más rápida que el *overhead* asociado a su lanzamiento, incluyendo la configuración y la comunicación entre el host y el device, como se mencionó en la Sección 2.2. TensorRT identifica capas con datos de entrada comunes y tamaños de filtro idénticos, pero con diferentes pesos, lo que permite ejecutar operaciones secuenciales en un único lanzamiento de kernel [19]. De este modo, se busca realizar dichas operaciones en un solo lanzamiento, reduciendo el costo asociado a múltiples lanzamientos. Un ejemplo de esta optimización es la fusión de capas de convolución, *bias* y ReLU en una única capa. Esta fusión se ilustra en la Figura 2.5b, basada en la red original mostrada en la Figura 2.5a, donde las capas fusionadas están etiquetadas como 'CBR'.
- **Horizontal fusion**: Esta técnica combina capas que toman el mismo tensor de origen y aplican las mismas operaciones con parámetros similares, fusionándolas en una única capa [19]. Al igual que en la etapa anterior, el objetivo es reducir la cantidad de lanzamientos de kernel y, con ello, su costo asociado. El ejemplo en la Figura 2.5c muestra la fusión de tres capas CBR de 1×1 de la Figura 2.5b, las cuales toman la misma entrada, en una única capa CBR de 1×1. Es importante mencionar que la salida de esta nueva capa debe desagregarse para alimentar las distintas capas subsiguientes del grafo original.




(a) Red neuronal. (b) Fusión vertical de capas. (c) Fusión horizontal de capas.

Fig. 2.5: Red neuronal de múltiples capas con respectivas fusiones de capas.

- **Quantization**: TensorRT proporciona herramientas para reducir la precisión de los parámetros (de `fp32` a `fp16`) o para cuantizar y calibrar el modelo (de `fp32` a `int8`), disminuyendo así el uso de recursos computacionales.

  La cuantización a `int8` permite representar únicamente 256 valores (de -128 a 127). Para minimizar la pérdida de precisión durante la inferencia, TensorRT realiza una calibración que ajusta los pesos y las activaciones para representarlos como enteros de 8 bits. Este proceso de calibración es totalmente automatizado y no requiere parámetros adicionales; utiliza una muestra representativa de los datos de entrada empleados en el entrenamiento para convertir de `fp32` a `int8` [19].

- **Automatic kernel tuning**: En esta etapa, el builder selecciona los kernels que se utilizarán para ejecutar el inference engine, de acuerdo con el nivel de optimización especificado en el parámetro `builder optimization level`, mencionado en la subsección anterior. TensorRT dispone de una biblioteca de kernels que implementan distintas capas empleadas en la construcción de diversos modelos de redes neuronales. El builder selecciona los kernels más adecuados para optimizar el modelo, considerando factores como la configuración específica del hardware objetivo, el batch size y el tamaño del filtro, entre otros.

  El proceso de *Automatic kernel tuning* consiste en probar iterativamente cuál de los kernels disponibles ofrece los mejores tiempos de ejecución en el hardware objetivo. Este tiempo se calcula con base en el parámetro `iterations`, también descrito en la subsección anterior. Sin embargo, este procedimiento está sujeto a ruidos de medición, como la velocidad variable de la GPU o los procesos en segundo plano del sistema operativo, lo que lo hace no determinístico (Sección 5.5, *Determinism* [28]). Debido a esto, optimizar el mismo modelo en múltiples ocasiones puede generar diferentes inference engines.

### Serialization

El builder produce un inference engine, que es una representación optimizada del modelo. Este inference engine puede ser utilizado directamente en la fase de ejecución (running phase) o, alternativamente, el builder puede serializarlo, creando un inference engine serializado, conocido como **plan**, que se puede almacenar en disco para su ejecución posterior.

### 2.3.2. Running Phase

En la running phase, el procedimiento más común consiste en tomar el plan generado en la etapa anterior y deserializarlo, con el objetivo de optimizar el modelo de manera offline. Esto evita realizar la optimización en tiempo de ejecución y mantener el modelo en memoria para llevar a cabo las inferencias. Al deserializar el plan, se obtiene el inference engine, a partir del cual es posible generar el contexto de ejecución. En este contexto de ejecución, se deben especificar las entradas y salidas necesarias para realizar las inferencias. En el caso de la programación de GPUs Nvidia en plataformas CUDA, este contexto corresponde a un entorno donde se ejecutan las tareas de procesamiento y contiene toda la información requerida para que el dispositivo específico pueda ejecutar kernels y gestionar su memoria. Las inferencias se efectúan dentro de este contexto de ejecución.

## 2.4. Trabajos relacionados

Este trabajo toma como punto de partida los resultados reportados en la tesis de magíster de Alexis Diomedi [31], la cual presenta una evaluación experimental sobre el efecto de diversas técnicas para optimizar los procesos de inferencia en estructuras típicas de redes neuronales artificiales. En particular, el trabajo previo explora los efectos de optimizar redes ya entrenadas mediante técnicas como *pruning* [32] y descomposición tensorial [33, 34], con el objetivo de simplificar la estructura de la red y reducir el tiempo de ejecución de las inferencias en plataformas embebidas basadas en CPUs y GPUs de distintos fabricantes. Las plataformas utilizadas incluyeron Jetson Xavier AGX, Jetson TX2, Odroid N2 y un computador de escritorio. Cabe destacar que las evaluaciones realizadas en este trabajo son agnósticas al hardware, ya que los métodos aplicados emplean técnicas genéricas. En el momento en que se desarrolló dicho trabajo, la ejecución de inferencia en dispositivos embebidos era aún incipiente, y las herramientas de diseño asistido o automatizado eran limitadas y poco documentadas. Por esta razón, las optimizaciones se implementaron mediante la manipulación directa del código fuente, utilizando bibliotecas como Keras o TensorFlow Lite. Las evaluaciones también consideraron la explotación de propiedades específicas de cada plataforma de hardware, lo que implicaba un esfuerzo significativo en términos de codificación y no siempre garantizaba una optimización eficiente en el uso de recursos. A pesar de estas limitaciones, los resultados mostraron mejoras en comparación con el modelo sin optimizaciones (modelo base). Las técnicas basadas en descomposición tensorial lograron los mejores desempeños, destacándose en métricas como la velocidad de inferencia, con una aceleración de hasta un factor de 1.6, y el ahorro de energía, con reducciones de hasta un factor de 3. Tanto el *pruning* como la descomposición tensorial lograron reducciones similares en el uso de memoria RAM, con mejoras de entre un 5 % y un 30 % respecto al modelo base.

Como aspecto complementario, el trabajo en [31] reporta comparaciones de las técnicas propuestas, las cuales eran agnósticas respecto al fabricante del hardware subyacente, con la versión de TensorRT disponible en ese momento. Los resultados muestran que, en las plataformas soportadas (específicamente la Jetson Xavier AGX y la Jetson TX2), TensorRT ofrecía los mejores desempeños en términos de consumo energético y velocidad de inferencia, alcanzando entre 1.5 y 4 veces la velocidad del modelo base. Sin embargo, fue superado por TensorFlow Lite en la optimización de los requerimientos de memoria RAM, logrando reducciones de entre 4 y 7 veces respecto al modelo base. Es importante señalar que estos resultados se consideran preliminares, ya que en ese entonces TensorRT estaba en etapas tempranas de desarrollo, con prestaciones y documentación limitadas, además de estar disponible únicamente para un número reducido de plataformas embebidas de la familia Jetson, que recién comenzaban a salir al mercado.

Tabla 2.2: Resumen del estado del arte.

| | **Hardware** | **Workflow** | **Quantization** | **Métricas** |
|---|---|---|---|---|
| **Zhou, 2022 [17]** | Nvidia Quadro (Discreta) | Pytorch-TensorRT<br>PyTorch-ONNX-TensorRT<br>ONNX-Runtime<br>PyTorch | fp32 | Precisión<br>Latencia<br>Throughput<br>Memoria |
| **Shafi, 2021 [18]** | Jetson Xavier NX<br>Jetson Xavier AGX | TensorRT<br>PyTorch<br>TensorFlow<br>Caffe | fp32 | Precisión<br>Latencia<br>Throughput |
| **Verma, 2021 [19]** | RTX 2080 (Discreta)<br>Tesla T4 (Discreta) | TensorFlow-TensorRT<br>TensorFlow Lite | fp32<br>fp16<br>int8 | Latencia<br>Throughput<br>Consumo energético<br>Tamaño del modelo |
| **Ulker, 2020 [20]** | Jetson Xavier AGX<br>Jetson TX2<br>Jetson Nano<br>Edge TPU<br>Nvidia 1050 Ti (Discreta) | PyTorch-ONNX-TensorRT<br>PyTorch<br>TensorFlow<br>Caffe | fp32<br>fp16 | Latencia<br>Throughput |

Desde la realización del trabajo citado en [31] hasta la actualidad, TensorRT ha experimentado un notable proceso de madurez, convirtiéndose en una herramienta versátil, con amplia documentación y soporte para múltiples plataformas, incluyendo GPUs discretas e integradas.

Estudios recientes evidencian un creciente interés en evaluar y caracterizar la optimización de la inferencia utilizando GPUs Nvidia junto con TensorRT. En este contexto, los trabajos resumidos en la Tabla 2.2 analizan distintas configuraciones que combinan hardware objetivo, workflows, cuantización y métricas. A continuación, se describen en detalle los aspectos representados en cada columna.

- **Hardware**: Plataforma de hardware objetivo utilizada para realizar las evaluaciones.
- **Workflow**: Flujos de trabajo evaluados en cada investigación, los cuales se describieron en la Sección 2.1.
- **Quantization**: Indica el nivel de cuantización aplicado por TensorRT a los modelos evaluados.
- **Métricas**: Las métricas reportadas incluyen: (i) la precisión de la inferencia, que corresponde a la métrica usada para comparar la exactitud con la que el modelo es capaz de hacer una inferencia; (ii) la latencia, que es el tiempo que el modelo tarda en completar una inferencia; (iii) el throughput, referente al número de inferencias que el modelo ejecuta por unidad de tiempo; (iv) la memoria, relacionada con la cantidad de memoria RAM/GPU empleada; (v) el consumo energético, que indica la energía que utiliza el sistema; y (vi) el tamaño del modelo, correspondiente al espacio en disco requerido para almacenarlo en megabyte (MB).

Los factores comunes en las evaluaciones del estado del arte sobre la optimización del rendimiento de la inferencia con TensorRT incluyen varios aspectos clave. Entre ellos, la latencia y el throughput destacan como las métricas principales consideradas en todos los estudios revisados. Además, la mayoría de los trabajos analizan la precisión de la inferencia para garantizar que las optimizaciones no comprometan la exactitud de los resultados. Otro aspecto crucial señalado por los autores es la flexibilidad, evaluada en términos del desempeño en diferentes arquitecturas de redes y tamaños

de batch. Los modelos de redes neuronales más utilizados en estas evaluaciones incluyen ResNet, MobileNet, SqueezeNet y AlexNet. En cuanto a los datasets, ImageNet-1k es el más empleado para medir la precisión de la inferencia y el rendimiento de los modelos optimizados.

Las evaluaciones revisadas varían en su enfoque específico y en los detalles técnicos. En particular, el estudio realizado por Ulker et al. [20] presenta una comparativa extensa entre diferentes frameworks y TensorRT, considerando diversos hardwares embebidos y dedicados, modelos, batch sizes y niveles de cuantización, ofreciendo un panorama amplio del rendimiento entre diferentes combinaciones de herramientas y hardware. Este estudio concluye que el mejor rendimiento general en la inferencia se obtiene utilizando TensorRT en comparación con los frameworks tradicionales. En contraste, el trabajo de Zhou et al. [17] profundiza en la integración de TensorRT con otros frameworks, proporcionando un análisis detallado sobre cómo estas integraciones afectan la precisión y el rendimiento de la inferencia, así como el uso de memoria GPU. Según Zhou et al., el workflow con el mejor rendimiento general es el de PyTorch-ONNX-TensorRT. Por su parte, Verma et al. [19] evalúan un conjunto de benchmarks para comparar TensorFlow Lite con respecto a TensorFlow-TensorRT, destacándose del resto de los estudios por evaluar la cuantización a `int8`, el consumo energético y el tamaño del modelo. Finalmente, Shafi et al. [18] se centran en caracterizar diferentes frameworks y TensorRT para la inferencia en dispositivos embebidos, revelando hallazgos específicos sobre TensorRT, como mejoras en la precisión de la inferencia tras la optimización, comportamientos inesperados e indeterminados (por ejemplo, variaciones en las salidas ante las mismas entradas), o incrementos en la latencia de ejecución para un mismo modelo al ser ejecutado en un hardware objetivo con más recursos computacionales.

Los artículos que evalúan TensorRT [17–20] se enfocan principalmente en los resultados de inferencia. Sin embargo, la mayoría de estos estudios no explican en detalle el funcionamiento interno del builder ni cómo se realizó la implementación específica que permite obtener los inference engines evaluados. Este enfoque genera una brecha significativa en la literatura actual, ya que no proporciona información suficiente sobre los procesos y métodos empleados en la construcción de los inference engines, ni sobre las optimizaciones y ajustes necesarios para alcanzar los resultados de inferencia reportados. Además, la falta de detalles sobre la implementación de la TensorRT Application dificulta la replicación precisa de los estudios y limita una comprensión completa de las capacidades y limitaciones de la herramienta evaluada. Por lo tanto, resulta necesario un análisis más profundo y detallado de la TensorRT Application que ofrezca una visión holística y comprensiva sobre el rendimiento y la eficiencia de esta herramienta.

Existen diversas áreas que no han sido abordadas adecuadamente en las evaluaciones actuales sobre la optimización del rendimiento de la inferencia utilizando TensorRT, lo que justifica la necesidad de una nueva evaluación. En primer lugar, no se ha analizado el desempeño en plataformas de la serie Jetson Orin [21], lo que limita la aplicabilidad de los resultados a los avances tecnológicos más recientes en hardware. En segundo lugar, las evaluaciones existentes no consideran todas las configuraciones disponibles, como los distintos modos de consumo energético configurables en las plataformas Jetson o las múltiples opciones de optimización descritas en la Sección 2.3.1 de TensorRT. Esta omisión impide una comprensión completa del rendimiento potencial bajo diversas condiciones operativas. Además, la falta de uniformidad en los métodos y la ausencia de una metodología clara dificultan la reproducibilidad de los resultados, un aspecto fundamental para validar hallazgos científicos. Por otro lado, las evaluaciones se han basado predominantemente en un único benchmark basado en la clasificación de imágenes, sin considerar otros escenarios que podrían revelar limitaciones prácticas, como el uso de TensorRT en tareas de segmentación o regresión. Finalmente, se requiere el desarrollo de nuevas métricas más relevantes a nivel de aplicación, que reflejen con mayor precisión el impacto de las optimizaciones en situaciones prácticas. Estos vacíos evidencian la necesidad de llevar a cabo estudios más exhaustivos y actualizados que aborden estas limitaciones.

Capítulo 3

# EVALUACIÓN DE CONFIGURACIONES REPORTADAS EN EL ESTADO DEL ARTE

Este capítulo presenta la metodología y los resultados de la evaluación del rendimiento de modelos optimizados con TensorRT, siguiendo prácticas recomendadas y procedimientos documentados en el estado del arte. El objetivo de la evaluación es cuantificar el impacto de las optimizaciones de TensorRT en tareas de inferencia, considerando diversas configuraciones de software y hardware, incluida la familia Jetson Orin de Nvidia, que aún cuenta con poca exploración en la literatura. La incorporación de estas arquitecturas permite ampliar el análisis hacia plataformas emergentes y validar las tendencias observadas en entornos convencionales, ofreciendo una perspectiva integral sobre el desempeño en escenarios típicos de benchmarking.

El capítulo describe en detalle el entorno de operaciones, especificando las plataformas de hardware, el software, los datasets y las configuraciones de TensorRT utilizadas para optimizar los modelos. Las métricas de evaluación incluyen precisión en la inferencia, propiedades de los modelos optimizados, latencia por inferencia y throughput. Los resultados se presentan con un análisis crítico que identifica vacíos en la literatura previa y destaca oportunidades para explorar nuevos puntos de optimización en aplicaciones que no han sido abordadas en estudios anteriores.

Para facilitar la reproducibilidad de los resultados, se ha habilitado un repositorio público en GitHub [35], que incluye los códigos y configuraciones utilizados en cada evaluación, junto con instrucciones detalladas para replicar la generación de aplicaciones con TensorRT en las distintas configuraciones documentadas en este trabajo. Esta estructura asegura el escrutinio y fomenta la aplicación práctica de los hallazgos en futuros desarrollos y estudios.

## 3.1. Configuración del entorno de operaciones

El entorno de operaciones utilizado en los experimentos de este capítulo está compuesto por la aplicación, el flujo de trabajo, el hardware y las métricas a evaluar. Las configuraciones consideradas incluyen la selección del modelo de red, el tipo de cuantización, el batch size de configuración, el batch size de inferencia, el nivel de optimización y la plataforma de hardware utilizada. Además, algunas de las plataformas de hardware evaluadas permiten operar en distintos modos de consumo. Por ejemplo, una posible configuración de operación podría estar compuesta por el modelo ResNet50, cuantización `fp16`, batch size de configuración estático, batch size de inferencia igual a uno, nivel de optimización

Tabla 3.1: Entorno de operaciones.

| | | | |
|---|---|---|---|
| **Aplicación** | **Tarea** | Clasificación de imágenes | |
| | **Dataset** | ImageNet-1k | |
| | **Modelo** | MobileNetV2<br>ResNet18<br>ResNet34<br>ResNet50<br>ResNet101<br>ResNet152 | |
| **Workflow** | **Framework** | PyTorch | |
| | **Graph Format** | ONNX | |
| | **Inference Optimizer** | TensorRT | |
| | **Configuración** | **Cuantización** | fp32, fp16, int8 |
| | | **Nivel de optimización** | 3 (predeterminado) |
| | | **Batch size de configuración** | Dinámico<br>Estático (permite evaluar latencia) |
| | | **Batch Size de inferencia** | 1, 32, 64, 128, 256 |
| **Hardware** | **Plataformas** | PC/RTX 3060<br>Laptop/RTX 2060<br>Jetson Xavier NX/AGX<br>Jetson Orin Nano/AGX | |
| | **Modo de consumo** | Predeterminado | |
| **Métricas** | | Precisión de la inferencia<br>Propiedades del modelo<br>Latencia<br>Throughput | |

predeterminado y la plataforma Jetson Orin Nano con un modo de consumo predefinido.

La Tabla 3.1 presenta un resumen del entorno de operaciones utilizado en las evaluaciones reportadas en este capítulo. Cada elemento del entorno de operaciones se detalla en las subsecciones siguientes.

### 3.1.1. Aplicación

La tarea desarrollada en las evaluaciones de este capítulo corresponde a la **clasificación de imágenes** utilizando el conjunto de datos **ImageNet-1k** (ILSVRC 2012) [7]. Este dataset contiene más de un millón de imágenes etiquetadas en mil categorías y es ampliamente utilizado como un benchmark para la evaluación de algoritmos de clasificación de imágenes. Además, ha sido empleado en todos los benchmarks analizados en la Sección 2.4, lo que proporciona resultados de referencia que sirven como punto de validación y comparación.

El dataset completo de ImageNet-1k tiene un tamaño aproximado de 138 GB e incluye conjuntos de entrenamiento y validación. En esta evaluación, se utiliza únicamente el conjunto de validación. Un enlace para su descarga, junto con el de los demás datasets mencionados en este capítulo, está disponible en un repositorio online [35]. Este conjunto contiene cincuenta muestras de cada una de las mil clases del dataset y se empleó para evaluar los modelos preentrenados de PyTorch en el conjunto de entrenamiento de ImageNet. Por otro lado, la creación de un inference engine de cuantización `int8` requiere un conjunto de datos de calibración [36], el cual se generó a partir de un

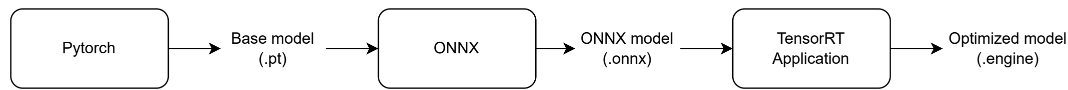



Fig. 3.1: Diagrama del workflow Torch-ONNX-TensorRT.

conjunto de muestras representativas de la aplicación.

Para evaluar los efectos del uso de TensorRT en la optimización de la inferencia, se seleccionaron los modelos de red **MobileNetV2** [37] y variantes de la arquitectura **ResNet** [38]. Estos modelos son ampliamente utilizados en aplicaciones de clasificación de imágenes, y sus propiedades y desempeño han sido documentados en numerosos benchmarks [17–20]. Además, debido a su popularidad, cuentan con estructuras preentrenadas en PyTorch, lo que simplifica la obtención de los modelos base.

MobileNetV2 es un modelo interesante en el contexto del experimento, ya que fue diseñado para ser computacionalmente eficiente, reduciendo la cantidad de parámetros mientras mantiene altos niveles de precisión en la inferencia. Estas características lo hacen ideal para su ejecución en plataformas embebidas con recursos computacionales limitados. Además, el uso de un modelo compacto permite evaluar la capacidad de TensorRT para optimizar aspectos de la implementación que no fueron considerados en el diseño inicial del modelo.

La familia de redes ResNet incluye variantes como ResNet18, ResNet50 y ResNet152. En estas denominaciones, el número indica la cantidad de capas, reflejando una relación directa con la profundidad de la red. Las versiones más avanzadas, como ResNet50 y ResNet152, incorporan bloques *bottleneck*, lo que mejora su capacidad de aprendizaje sin un incremento significativo en los parámetros.

### 3.1.2. Workflow

Para las pruebas realizadas en este capítulo, se emplea un workflow de optimización de la inferencia basado en la nomenclatura y terminología presentadas en la Sección 2.1. Este workflow integra un **framework** (PyTorch), un **graph format** (ONNX) y un **inference optimizer** (TensorRT). La Figura 3.1 ilustra el proceso: PyTorch se utiliza para definir y entrenar la red, ONNX actúa como formato intermedio que facilita la interoperabilidad, y TensorRT se encarga de la optimización del modelo. La literatura revisada respalda la elección de este workflow, destacando su superior desempeño frente a otras alternativas, según los resultados reportados para sus inference engines [39]. Además, las herramientas empleadas en este flujo están en desarrollo activo, reciben actualizaciones periódicas, cuentan con documentación accesible y son recomendadas por Nvidia [16].

#### Configuración de la TensorRT Application

Para crear los inference engines en cada configuración de operación, es necesario generar una TensorRT Application, como se describió en la Sección 2.3. Para ello, se define una configuración en el builder que ajusta los parámetros de cuantización y el batch size.

En los experimentos de este capítulo, se consideraron **cuantizaciones** en punto flotante de 32 bits, punto flotante de 16 bits y enteros de 8 bits, generando los engines denominados `TRT fp32`, `TRT fp16` y `TRT int8`, respectivamente.

Con respecto al **batch size**, se analizaron dos configuraciones: (i) batch size de configuración estático, que emplea un batch size de inferencia de tamaño uno, utilizado para caracterizar la latencia

Tabla 3.2: Características de los dispositivos utilizados [40].

| **Plataforma** | **Núcleos GPU** | **Memoria (GB)** | **CPU** |
|---|---|---|---|
| Nvidia GeForce RTX 3060 | — | 12 (GDDR6) + 36 RAM | Intel i3-12100F |
| Nvidia GeForce RTX 2060 | — | 6 (GDDR6) + 12 RAM | Intel i7-10750H |
| Jetson Xavier NX | 384 | 8 RAM (compartida) | ARMv8.2, 6 núcleos, 1.9 GHz |
| Jetson Xavier AGX | 512 | 16 RAM (compartida) | ARMv8.2, 8 núcleos, 2.3 GHz |
| Jetson Orin Nano | 1024 | 8 RAM (compartida) | ARMv8.2, 6 núcleos, 1.7 GHz |
| Jetson Orin AGX | 2048 | 64 RAM (compartida) | ARMv8.2, 12 núcleos, 2.2 GHz |

Tabla 3.3: Versiones de software.

| Plataforma | JetPack | CUDA | Python | PyTorch | ONNX | TensorRT |
|---|---|---|---|---|---|---|
| RTX 2060 | - | 12.6 | 3.10 | 2.2.2 | 1.16.0 | 8.6.1.post1 |
| RTX 3060 | - | 12.6 | 3.10 | 2.2.2 | 1.16.0 | 8.6.1.post1 |
| Jetson Xavier NX/AGX | 5.1.2-b104 | 11.4 | 3.8.10 | 2.1.0a0+41361538.nv23.06 | 1.15.0 | 8.5.2.2 |
| Jetson Orin Nano/AGX | 6.0-b52 | 12.2 | 3.10.12 | 2.2.0a0+6a974be | 1.16.0 | 8.6.2 |

de cada inferencia; y (ii) batch size de configuración dinámico, con un valor mínimo de uno, óptimo de 128 y máximo de 256. Esta última configuración proporciona una solución versátil para evaluar el throughput obtenido con distintos tamaños de batch size de inferencia, sin necesidad de generar un engine específico para cada tamaño. Cabe destacar que, en general, los detalles sobre la configuración del batch size para generar los engines suelen omitirse en los reportes de la literatura, por lo que el uso de la configuración dinámica se considera adecuado para esta evaluación.

### 3.1.3. Hardware

#### Plataformas

Las plataformas de hardware seleccionadas abarcan una gama diversa en términos de rendimiento, costo y consumo energético, lo que permite realizar una evaluación exhaustiva en distintos contextos. Las GPUs discretas utilizadas en computadoras de escritorio proporcionan un punto de referencia, mientras que las plataformas embebidas, como las de la familia Jetson, están diseñadas para ofrecer eficiencia y alto rendimiento en entornos donde el espacio y el consumo energético son factores críticos. Estas últimas constituyen el principal foco de las evaluaciones realizadas. Las plataformas analizadas se detallan en la Tabla 3.2.

Es importante mencionar que las plataformas de hardware descritas no cuentan todas con el mismo soporte de software, debido al rápido avance de las tecnologías. Por ejemplo, las versiones de TensorRT para las plataformas de la serie Jetson se distribuyen en los llamados *Jetpacks*, que incluyen versiones específicas de Linux, Python, CUDA y TensorRT adaptadas a cada hardware. En este contexto, las plataformas de la familia Xavier no son compatibles con las versiones más recientes de Jetpacks, las cuales sí están disponibles para la familia Orin, de gama más reciente. La versión utilizada en cada plataforma depende exclusivamente del soporte disponible y se detalla en la Tabla 3.3. Para las computadoras de escritorio, se empleó el sistema operativo Ubuntu 22.04 LTS. Aunque las limitaciones mencionadas impiden el uso de un flujo de trabajo uniforme para todas las plataformas, cada una fue configurada con la versión más reciente de las herramientas compatibles al momento de los experimentos, lo que, según la documentación, asegura el mejor desempeño posible en cada caso.

Tabla 3.4: Modos de consumo energético predeterminados según la plataforma.

| **Plataforma** | **Modo de Consumo** | **Power budget** [W] | **Online CPU** |
|---|---|---|---|
| Jetson Xavier NX | Modo PM5 | 10 | 4 |
| Jetson Xavier AGX | Modo PM7 | 15 | 4 |
| Jetson Orin Nano | Modo PM0 | 15 | 6 |
| Jetson Orin AGX | Modo PM2 | 30 | 8 |

#### Modo de consumo

En las plataformas Jetson utilizadas, es posible configurar el parámetro global del modo de consumo energético, independientemente de las configuraciones de la TensorRT Application. Este modo define la cantidad de núcleos de procesamiento activos y establece un límite en el consumo energético permitido para el hardware objetivo, un aspecto que no ha sido ampliamente reportado en la literatura. Para mantener la uniformidad en las pruebas presentadas en este capítulo, se emplearon las configuraciones de energía activas al momento de inicializar los dispositivos según su configuración de fábrica, como se indica en la Tabla 3.4. Cabe destacar que estas configuraciones no coincidían con las consideradas predeterminadas según la documentación de Nvidia. Los detalles sobre los modos de consumo energético se encuentran disponibles en [41].

### 3.1.4. Métricas

Los trabajos revisados en la Sección 2.4 que evalúan modelos optimizados mediante TensorRT se centran principalmente en obtener métricas de desempeño computacional de dichos modelos, como el promedio de latencia por inferencia y el throughput de la aplicación. Algunos estudios también reportan el tamaño en bytes de los modelos optimizados, lo cual es relevante para su almacenamiento en plataformas embebidas con restricciones de espacio. Además, el tamaño del modelo proporciona un indicio sobre el uso de memoria en tiempo de ejecución, siendo esperable que un modelo más compacto reduzca el consumo de memoria durante las inferencias.

Como se concluyó al final de la Sección 2.4, una de las principales falencias de la literatura previa es la falta de definiciones concretas de las métricas reportadas, así como la omisión de detalles de implementación que permitan reproducir los experimentos y validar los resultados. Esta limitación implica que los resultados reportados no sean directamente comparables, permitiendo únicamente identificar ciertas tendencias generales.

En este contexto, para las evaluaciones realizadas en este capítulo se utilizarán métricas similares a las reportadas en la literatura. Las métricas de interés incluyen precisión de inferencia, propiedades del modelo optimizado, latencia y throughput, las cuales se describen a continuación:

**Precisión de la inferencia** Para que los resultados de la optimización realizada por un inference optimizer sean relevantes en la aplicación final, la salida de los inference engines evaluados debería ser, idealmente, idéntica a la de los modelos base con los que se comparan. En tareas de clasificación, como la considerada en esta evaluación, la salida de una red consiste típicamente en un conjunto de puntajes asociados a cada posible clase. Estos puntajes reflejan la confianza del modelo en que la entrada pertenece a una determinada clase. Las métricas finales de classification accuracy [1] incluyen:

- **Classification accuracy Top-1**: indica el porcentaje de veces que la clase predicha con mayor confianza (es decir, el máximo puntaje según la salida de la red) corresponde a la clase correcta,

[1] La métrica accuracy se traduce como 'precisión' en este contexto.

siendo esta la métrica estándar de precisión en tareas de clasificación.

- **Classification accuracy Top-5**: Mide el porcentaje de veces que la clase verdadera se encuentra entre las cinco principales predicciones del modelo, ordenadas según su nivel de confianza. Esta métrica ofrece una visión más amplia del rendimiento del modelo en comparación con Top-1, especialmente en problemas con muchas clases posibles. Es útil saber si la clase correcta estuvo entre las opciones mejor puntuadas, incluso si no fue la más alta. Esto permite comprender si el modelo tiene un nivel aceptable de discriminación. Por ejemplo, si la clase correcta es 'buque' pero quedó detrás de clases como 'velero' y 'crucero' en puntuación, esto indica que el modelo es capaz de identificar aspectos comunes a todas estas clases.

**Propiedades del modelo** Para obtener un mejor entendimiento de las optimizaciones que realiza TensorRT en la configuración de los hiperparámetros del modelo, se caracterizan las siguientes propiedades:

- **Layers**: Corresponde al número de capas que conforman la red neuronal. Esta información se extrae usando la herramienta `Polygraphy` [42].
- **Parameters**: Corresponde al número de pesos y sesgos que conforman la red neuronal. Este valor se calcula utilizando las funcionalidades internas de TensorRT, como '`EngineInspector`' [43], para los inference engines. Para el modelo base, se emplea la librería *ONNX Operations Counter* (`ONNX-opcounter`) [44].
- **Model Size**: Se refiere al tamaño total ocupado por los parámetros y la estructura del modelo, medido en términos de la cantidad de memoria necesaria para almacenarlo en megabytes. Se mide utilizando la función '`getsize`' [45] del módulo '`os`' de Python.

**Latencia** La latencia se define como el tiempo necesario para procesar una sola entrada a través de la red (es decir, con un batch size de inferencia igual a uno), excluyendo el preprocesamiento de la entrada [46]. En nuestros experimentos, se mide utilizando rondas de warm-up equivalentes al 10 % del total de datos del conjunto de validación. El uso del warm-up permite que el sistema alcance un estado de operación estable antes de medir la latencia, evitando *outliers* que pueden surgir durante las primeras inferencias debido a la carga inicial del modelo [17]. En este trabajo, la latencia se mide calculando el tiempo total de ejecución de cada inferencia, desde el inicio hasta el final, incluyendo las transferencias de datos entre la CPU y la GPU, y viceversa mediante la función '`time`' [47] en el módulo '`time`' de Python. Se registran los valores de latencia de cada inferencia para una única ejecución en todo el conjunto de validación, con los cuales se calcula una latencia promedio y una latencia máxima.

**Throughput** El throughput se utiliza para indicar la cantidad de datos procesados por unidad de tiempo. Para calcularlo, se divide el número de entradas del modelo (batch size de inferencia) por el tiempo que le toma al modelo procesar dichas entradas, de forma similar a lo mostrado en [17]. Este tiempo de procesamiento de las entradas se calcula de la misma manera que la latencia, utilizando un warm-up del 10 % e incluyendo los tiempos de transferencia de datos entre la CPU y la GPU mediante la función '`time`' de Python. Se registran los valores de tiempo de procesamiento de cada lote de entradas con el tamaño del batch size de inferencia para una única ejecución en todo el conjunto de validación, a partir de los cuales se calcula un tiempo de procesamiento promedio que se utiliza para determinar el throughput promedio para dicho batch size de inferencia. Como se describe en [17, 48], incrementar el batch size de inferencia puede incrementar el throughput, por lo que se mide el throughput de inferencia evaluando distintos tamaños de batch size. Comenzamos

con un batch size de inferencia de uno, seguido de 32 y lo duplicamos sucesivamente hasta llegar a 256.

Los detalles de la implementación utilizada para extraer las métricas de este trabajo están disponibles en el código del repositorio [35].

## 3.2. Reporte de resultados

A continuación, y para evitar el exceso de datos y facilitar el análisis, se presentan solo algunos de los resultados relevantes para algunas configuraciones representativas de hardware y software. Para obtener más detalles sobre los experimentos, se recomienda consultar el Apéndice A, que incluye tablas adicionales con las métricas de todas las configuraciones evaluadas, y el Apéndice B, que contiene las tablas específicas con los resultados de throughput. Todos los datos generados durante el experimento están disponibles en el repositorio de GitHub [35], específicamente en el directorio `outputs/table_outputs/revision_practica`.

### 3.2.1. Precisión de la inferencia

La Tabla 3.5 presenta los resultados de precisión de la inferencia Top-1 y Top-5, evaluadas en la plataforma Jetson Orin Nano. Los valores obtenidos indican que, a pesar de la cuantización, la variación en la precisión es despreciable, con un decremento máximo de aproximadamente 1.8 % (peor caso al evaluar Top-1 en ResNet101 con `TRT int8`). Este leve decremento en la precisión se explica por la naturaleza discreta de la tarea de clasificación de imágenes, donde el resultado final no depende directamente de los puntajes individuales, sino de su relación relativa durante el postprocesamiento, como se revisó en la Sección 3.1.4. En este contexto, mientras los puntajes no se crucen, es decir, las clases permanezcan suficientemente separables, los puntajes pueden variar dentro de ciertos márgenes sin afectar el resultado de la clasificación. Esto permite que el desempeño del modelo se mantenga incluso cuando se reduce la precisión numérica de sus parámetros mediante técnicas como la cuantización con `TRT int8`.

Los resultados obtenidos en el resto de las configuraciones evaluadas son similares a los mostrados en la Tabla 3.5, destacándose que TensorRT preserva la precisión de la inferencia en distintos niveles de cuantización. Este comportamiento se observa independientemente del tamaño de batch evaluado en el mismo modelo, siempre que la arquitectura de la red permanezca inalterada. Del mismo modo, los resultados se mantienen consistentes al ser evaluados en distintas plataformas.

### 3.2.2. Caracterización de las propiedades del modelo

La Tabla 3.6 resume los resultados obtenidos para la plataforma Jetson Orin Nano usando un batch size de configuración dinámico. Se analiza esta plataforma como ejemplo considerando que esta orientada a entornos embebidos que suelen tener mayores restricciones de almacenamiento y memoria. Los resultados muestran las propiedades de los modelos en términos de número de capas, número de parámetros y tamaño, así como su variación relativa respecto al modelo base.

Los resultados de la Tabla 3.6 confirman que TensorRT optimiza la estructura del modelo, reduciendo en todos los casos el número de capas en comparación con el modelo base. Para las configuraciones evaluadas, se observa una disminución del número de capas de aproximadamente un 30 % en las optimizaciones `TRT fp32` y `TRT fp16`, y de alrededor de un 55 % en `TRT int8`. Este resultado se atribuye al proceso de optimización mediante la fusión de capas, descrito en la Sección 2.3.1.

Tabla 3.5: Precisión de la inferencia respecto al modelo base en la plataforma Jetson Orin Nano con un batch size de inferencia de uno.

| | | Classification accuracy [%] | |
|---|---|---|---|
| | | Top 1 | Top 5 |
| MobileNet | Modelo base | 72.02 | 90.62 |
| | TRT fp32 | 72.01 | 90.62 |
| | TRT fp16 | 72.00 | 90.64 |
| | TRT int8 | 71.45 | 90.29 |
| ResNet18 | Modelo base | 69.75 | 89.07 |
| | TRT fp32 | 69.76 | 89.08 |
| | TRT fp16 | 69.75 | 89.08 |
| | TRT int8 | 69.57 | 88.95 |
| ResNet34 | Modelo base | 73.31 | 91.42 |
| | TRT fp32 | 73.31 | 91.43 |
| | TRT fp16 | 73.32 | 91.45 |
| | TRT int8 | 73.18 | 91.41 |
| ResNet50 | Modelo base | 80.34 | 95.13 |
| | TRT fp32 | 80.35 | 95.14 |
| | TRT fp16 | 80.35 | 95.13 |
| | TRT int8 | 79.75 | 95.07 |
| ResNet101 | Modelo base | 81.68 | 95.66 |
| | TRT fp32 | 81.67 | 95.67 |
| | TRT fp16 | 81.67 | 95.65 |
| | TRT int8 | 79.95 | 95.55 |
| ResNet152 | Modelo base | 82.32 | 95.92 |
| | TRT fp32 | 82.32 | 95.92 |
| | TRT fp16 | 82.33 | 95.91 |
| | TRT int8 | 82.04 | 95.84 |

Por otro lado, los datos muestran que TensorRT no redujo significativamente la cantidad total de parámetros en comparación con el modelo base, lo que sugiere que no se llevó a cabo una poda de la red.

La configuración `TRT fp32` muestra un aumento en el tamaño del modelo con respecto al modelo base en todos los casos evaluados. Por ejemplo, en MobileNetV2, el tamaño del modelo optimizado en MB es aproximadamente un 12 % mayor que el modelo base. Este comportamiento puede explicarse considerando que, en modelos de aprendizaje profundo, los mayores requerimientos de almacenamiento están asociados con la necesidad de guardar millones de parámetros. Dado que el número de parámetros no se reduce significativamente y que `TRT fp32` utiliza la misma cantidad de bits por parámetro que el modelo base, se deduce que TensorRT introduce un overhead en la representación binaria del modelo que supera el ahorro en bits generado por la leve reducción del número de parámetros.

Por otro lado, en las configuraciones `TRT fp16` y `TRT int8`, el tamaño en MB requerido para almacenar el modelo disminuye en comparación con el modelo base, alcanzando una reducción de aproximadamente un 72 % en los modelos ResNet con cuantización `TRT int8`.

Las propiedades de los modelos para el resto de las configuraciones evaluadas se presentan en el Apéndice A y muestran tendencias similares.

Tabla 3.6: Variación en las propiedades del modelo respecto al modelo base en la plataforma Jetson Orin Nano, mostrada como razón relativa (Tamaño del modelo optimizado / Tamaño del modelo base).

| | | Propiedades del modelo | | | Variación (Razón Relativa) | | |
|---|---|---|---|---|---|---|---|
| | | # Layers | # Parameters | Size [MB] | # Layers | # Parameters | Size [MB] |
| MobileNet | Modelo base | 109 | 3487816 | 13.6 | – | – | – |
| | TRT fp32 | 75 | 3469760 | 15.3 | 0.69 | 0.99 | 1.12 |
| | TRT fp16 | 70 | 3469760 | 8.5 | 0.64 | 0.99 | 0.63 |
| | TRT int8 | 57 | 3469760 | 4.8 | 0.52 | 0.99 | 0.35 |
| ResNet18 | Modelo base | 58 | 11684712 | 44.7 | – | – | – |
| | TRT fp32 | 38 | 11678912 | 45.6 | 0.66 | 1.00 | 1.02 |
| | TRT fp16 | 39 | 11678912 | 23.4 | 0.67 | 1.00 | 0.52 |
| | TRT int8 | 25 | 11669504 | 12.1 | 0.43 | 1.00 | 0.27 |
| ResNet34 | Modelo base | 98 | 21789160 | 83.3 | – | – | – |
| | TRT fp32 | 84 | 21779648 | 84.2 | 0.86 | 1.00 | 1.01 |
| | TRT fp16 | 76 | 21779648 | 42.5 | 0.78 | 1.00 | 0.51 |
| | TRT int8 | 41 | 21770240 | 21.8 | 0.42 | 1.00 | 0.26 |
| ResNet50 | Modelo base | 131 | 25530472 | 97.8 | – | – | – |
| | TRT fp32 | 95 | 25502912 | 99.5 | 0.73 | 1.00 | 1.02 |
| | TRT fp16 | 99 | 25502912 | 50.4 | 0.76 | 1.00 | 0.52 |
| | TRT int8 | 58 | 25493504 | 26.6 | 0.44 | 1.00 | 0.27 |
| ResNet101 | Modelo base | 250 | 44496488 | 170.5 | – | – | – |
| | TRT fp32 | 203 | 44442816 | 172.3 | 0.81 | 1.00 | 1.01 |
| | TRT fp16 | 174 | 44442816 | 86.7 | 0.70 | 1.00 | 0.51 |
| | TRT int8 | 109 | 44433408 | 45.1 | 0.44 | 1.00 | 0.26 |
| ResNet152 | Modelo base | 369 | 60117096 | 230.5 | – | – | – |
| | TRT fp32 | 296 | 60040384 | 231.8 | 0.80 | 1.00 | 1.01 |
| | TRT fp16 | 297 | 60040384 | 116.9 | 0.81 | 1.00 | 0.51 |
| | TRT int8 | 160 | 60030976 | 61.0 | 0.43 | 1.00 | 0.26 |

### 3.2.3. Caracterización de la latencia

La Figura 3.2 presenta los resultados obtenidos para las plataformas Jetson Orin Nano, Jetson Orin AGX, Jetson Xavier AGX y RTX 2060. Estos resultados se basan en configuraciones con un batch size de configuración estático y un batch size de inferencia de uno, considerando las cuantizaciones `fp32`, `fp16` e `int8`. En dicha figura, se muestra la latencia promedio (representada por la altura de las barras) y la latencia máxima (indicada por la extensión superior de cada barra), comparando las diferentes configuraciones con los modelos base.

La Figura 3.2 muestra los resultados de latencia, los que indican que las optimizaciones mediante TensorRT reducen significativamente la latencia del modelo optimizado en comparación con el modelo base. Las configuraciones que usan cuantización `TRT int8` reportaron las latencias más bajas, con aceleraciones de aproximadamente cinco veces respecto al modelo base en la red ResNet152 en plataformas como la Jetson Orin Nano. Además, aunque la red MobileNet está diseñada desde su origen para eficiencia en dispositivos embebidos, TensorRT sigue mostrando un impacto positivo en la reducción de latencia, con una disminución de alrededor de seis veces en plataformas como la Jetson Orin Nano. Finalmente, se observa que las tecnologías más recientes, como la Jetson Orin, tienden a obtener menores valores de latencia para `TRT fp32` en comparación con otras plataformas de generación previa.

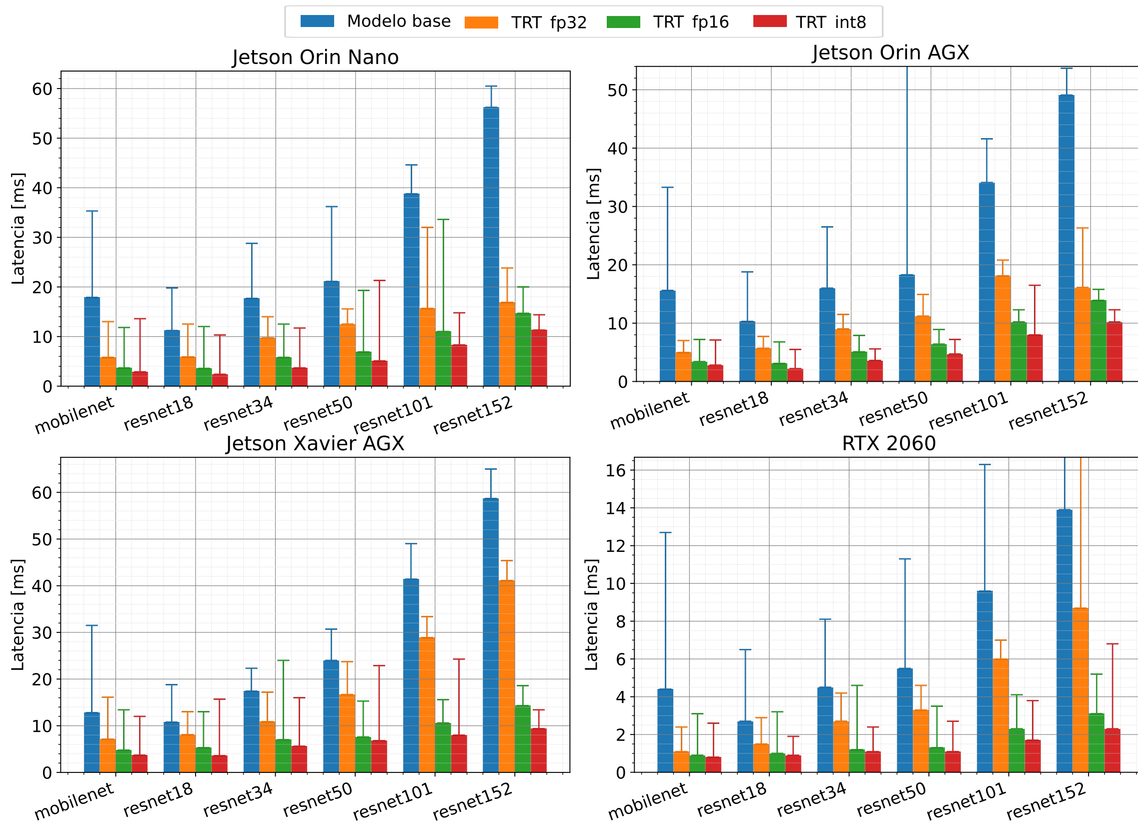



Fig. 3.2: Comparación de las latencias promedio y máximas de los modelos MobileNet y ResNet en distintas plataformas, utilizando un batch size de configuración estático y un batch size de inferencia de uno.

Los resultados de latencia para todas las configuraciones evaluadas se presentan en el Apéndice A y muestran tendencias similares.

### 3.2.4. Caracterización del throughput

La Figura 3.3 muestra los resultados del throughput promedio para las plataformas Orin Nano y RTX 2060 al ejecutar las redes MobileNet y ResNet152. Estas configuraciones incluyen la red más pequeña y la más compleja entre los modelos evaluados, ejecutadas en una plataforma embebida y una GPU discreta. La evaluación considera tanto el batch size de configuración dinámica como los batch sizes de inferencia indicados en la Tabla 3.1, y utiliza configuraciones de cuantización `fp32`, `fp16` e `int8`.

La Figura 3.3 muestra que todas las implementaciones generadas mediante TensorRT logran un mayor throughput de inferencia comparado con el modelo base. Las configuraciones con `TRT int8`, ResNet152 y batch sizes de inferencia superiores a 32 destacan por alcanzar mejoras de hasta 10 veces en comparación con el modelo base. Aunque el modelo base no muestra incrementos en throughput con batch sizes mayores a 32, TensorRT aprovecha más eficientemente la memoria del hardware para optimizar el throughput. No obstante, el impacto del batch size en el throughput varía según

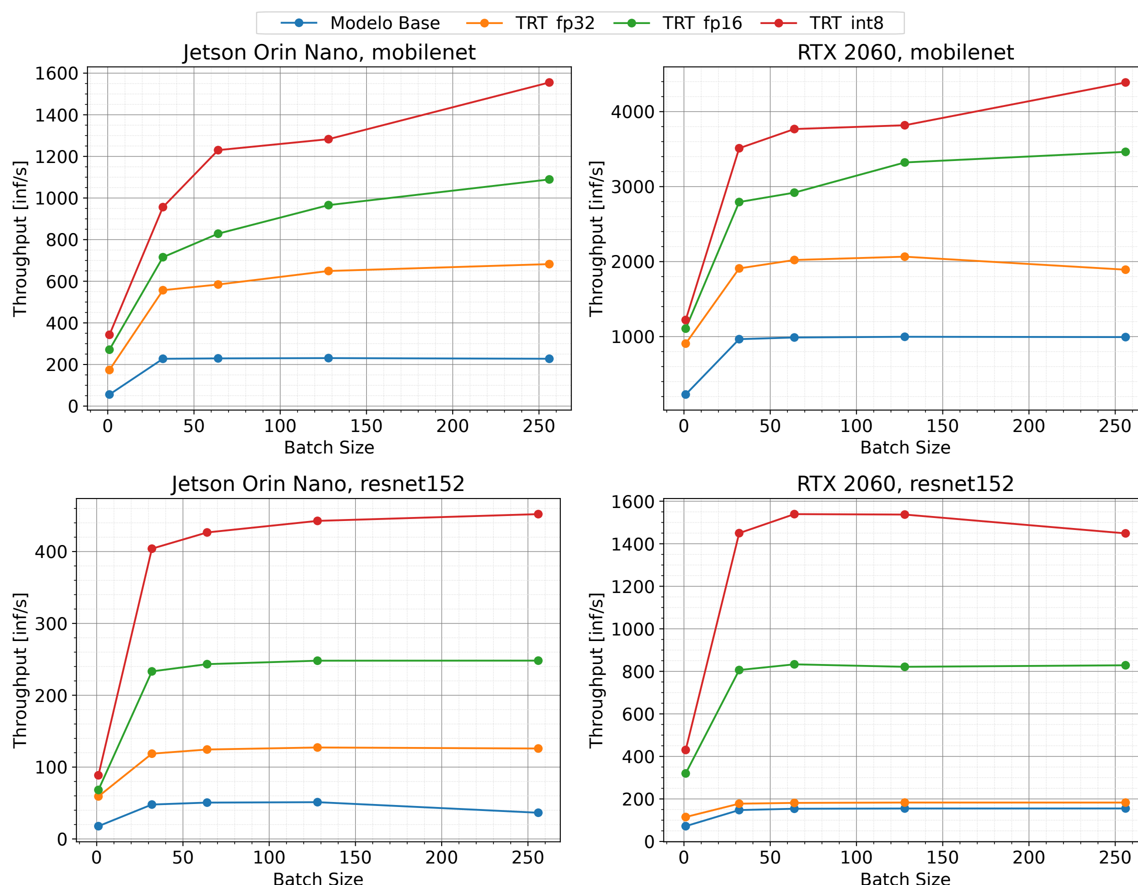



Fig. 3.3: Comparación del throughput en función del batch size de inferencia para los modelos MobileNet y ResNet152 en las plataformas Jetson Orin Nano y RTX 2060.

el hardware. Por ejemplo, en la plataforma RTX 2060, un batch size de inferencia superior a 64 en una red grande como ResNet152 no solo no mejora el throughput, sino que puede reducirlo debido a las limitaciones de memoria del dispositivo.

Los resultados detallados de la evaluación de throughput para toda la gama de configuraciones evaluadas se presentan en el Apéndice B y muestran tendencias similares a las reportadas en esta sección.

## 3.3. Discusión de resultados

Los resultados analizados en las secciones previas son consistentes con el marco en el que las representaciones más compactas mejoran el desempeño computacional. Esto concuerda con la literatura previa; sin embargo, como se mencionó al inicio de la Sección 3.1.4, no fue posible realizar comparaciones cuantitativas con los trabajos reportados en la Sección 2.4. Por lo tanto, este estudio se centró en validar tendencias en métricas típicas de desempeño, tales como la precisión de la inferencia, la latencia y el throughput, evaluadas en diversas plataformas de hardware y configuraciones

de TensorRT. Se observó que las tendencias relativas eran consistentes, lo que respalda la validez de los resultados obtenidos.

Se observó que la **precisión de la inferencia** mostró un comportamiento consistente. Aunque la metodología de evaluación difiere de estudios previos [17, 18], la tendencia general se mantiene: las optimizaciones de TensorRT, incluida la cuantización a `int8`, no afectan significativamente la precisión de la inferencia en tareas de clasificación de imágenes. No obstante, surge un vacío importante al considerar el impacto de estas optimizaciones en tareas distintas a la clasificación, como la regresión, donde las salidas son continuas en lugar de discretas. En estos casos, la modificación de puntajes basada en una cantidad fija de clases para determinar la salida de la red, como se describió en la Sección 3.1.4, no resulta aplicable. Esto resalta la necesidad de explorar otras métricas de evaluación, como *closeness*, que permitan reflejar de manera más completa el impacto de TensorRT en las salidas de los modelos.

Respecto a las **propiedades del modelo**, al analizar el número de capas y parámetros, es evidente que TensorRT no comprime la representación del modelo. El número total de parámetros permanece prácticamente intacto. Sin embargo, se observa que TensorRT modifica la estructura del modelo al reducir la cantidad de capas, reorganizando todos los parámetros dentro de estas nuevas configuraciones. En cuanto al tamaño del modelo, se encontró que, al usar los modelos optimizados con `TRT fp32`, el modelo final resulta más pesado que el modelo base. No obstante, el tamaño se reduce de manera significativa al emplear `TRT fp16` y `TRT int8`. Cabe destacar que, aunque la evaluación de [19] identifica el overhead asociado a la optimización y la reducción de tamaño al usar cuantización a `fp16`, no reporta una disminución mayor al emplear cuantización a `int8`, como sí ocurre en los resultados obtenidos en este trabajo.

En cuanto a la **latencia y el throughput**, los resultados confirman que las optimizaciones de TensorRT reducen consistentemente los tiempos de inferencia mientras aumentan el throughput, en línea con las tendencias reportadas en estudios previos [17–20]. En el análisis específico del throughput, se identificaron diferencias al planificar los experimentos entre el uso de batch size de configuración dinámica y estática, configuraciones que pueden impactar tanto el desempeño computacional como el **uso de memoria durante la inferencia**. Como se mencionó en la Sección 3.1.2, este trabajo empleó un batch size de configuración dinámico para la evaluación del throughput. No obstante, en los estudios previos no se especifica qué tipo de configuración de batch size se utilizó, lo que resalta la importancia de realizar análisis más detallados.

Como se mencionó en la Sección 3.1.3, ninguno de los trabajos que utilizan las plataformas Jetson reporta el impacto de los diferentes **modos de consumo** disponibles en estas plataformas ni especifica cuál es el modo empleado en las evaluaciones. Es razonable suponer que un cambio en el modo de consumo podría tener impactos significativos en el desempeño de las optimizaciones realizadas por TensorRT. Por ello, se considera necesario realizar evaluaciones que incluyan el modo de consumo de las plataformas Jetson como un factor en el análisis.

Finalmente, no se ha reportado el uso de otras configuraciones de TensorRT, como la evaluación de los distintos **niveles de optimización** disponibles al configurar la TensorRT Application, según lo señalado en la Sección 2.3.1. Por ello, es crucial evaluar cómo estas configuraciones afectan métricas como el throughput y la latencia, además de analizar el impacto en el uso de memoria del hardware objetivo en tiempo de ejecución.

En el siguiente capítulo se abordarán los vacíos identificados en esta discusión. Para ello, se evaluará la métrica **closeness** con el fin de analizar el impacto de TensorRT en las salidas de los modelos optimizados, junto con el **uso de memoria durante la inferencia**. Además, se comparará el rendimiento entre batch size de configuración estático y dinámico, y se analizarán los efectos de los distintos modos de consumo disponibles en las plataformas Jetson sobre las optimizaciones

realizadas. Finalmente, se explorará la configuración de diferentes niveles de optimización dentro de la TensorRT Application.

Capítulo 4

# EVALUACIÓN CON NUEVAS CONFIGURACIONES Y MÉTRICAS

Este capítulo presenta la metodología y los resultados de la evaluación del rendimiento de modelos optimizados con TensorRT, expandiendo la exploración presentada en el capítulo anterior al incorporar configuraciones de hardware/software y métricas que no han sido reportadas en la literatura previa.

En el contexto de la configuración de la TensorRT Application, se incorporan nuevos parámetros, incluyendo el uso de batch size de configuración estático y dinámico, así como el nivel de optimización empleado en la construcción del engine. Además, las plataformas de hardware consideradas corresponden a las versiones más recientes de la línea Jetson disponibles al momento de este trabajo, las cuales cuentan con soporte extendido por parte del proveedor y permiten la configuración de distintos modos de consumo energético. Para evaluar el impacto de los nuevos parámetros, se analizan métricas de latencia y throughput, definidas en el Capítulo 3, junto con nuevas métricas, como el error de salida del modelo (*closeness*) y el uso de memoria durante la inferencia. El objetivo de estas evaluaciones es cuantificar el impacto de las optimizaciones de TensorRT en tareas de inferencia bajo configuraciones y métricas que permiten extender el análisis y los resultados reportados en la literatura reciente.

Para facilitar la reproducibilidad de los resultados, se ha habilitado un repositorio público en GitHub [35], que incluye los códigos y configuraciones utilizadas en cada evaluación, junto con instrucciones detalladas para replicar la generación de aplicaciones con TensorRT en las distintas configuraciones documentadas en este trabajo. Esta estructura asegura el escrutinio y fomenta la aplicación práctica de los hallazgos en futuros desarrollos y estudios.

## 4.1. Configuración del entorno de operación

El entorno de operaciones utilizado en las evaluaciones de este capítulo se define de manera similar a lo expuesto en la Sección 3.1. La Tabla 4.1 resume las configuraciones empleadas, mientras que a continuación se detallan los elementos que complementan y amplían la información presentada en el capítulo anterior.

Tabla 4.1: Entorno de operación.

| **Aplicación** | **Tarea** | Clasificación de imágenes | |
|---|---|---|---|
| | **Dataset** | ImageNet-1k | |
| | **Modelo** | ResNet50<br>ResNet152 | |
| **Workflow** | **Framework** | PyTorch | |
| | **Graph Format** | ONNX | |
| | **Inference Optimizer** | TensorRT | |
| | **Configuración** | **Cuantización** | fp32, fp16, int8 |
| | | **Batch size de configuración** | Dinámico<br>Estático |
| | | **Batch size de inferencia** | 1, 32, 64, 128, 256 |
| | | **Nivel de optimización** | 0, 3, 5 |
| **Hardware** | **Plataformas** | Jetson Orin Nano/AGX | |
| | **Modo de consumo** | PM0, PM1, PM2 | |
| **Métricas** | | Classification closeness<br>Memoria<br>Latencia<br>Throughput | |

### 4.1.1. Aplicación

La tarea realizada en las evaluaciones de este capítulo corresponde a la **clasificación de imágenes** mediante un subconjunto extraído del dataset de validación de ImageNet-1k utilizado en el capítulo anterior, cuyo enlace de descarga está disponible en el repositorio online [35]. Este subconjunto contiene cinco muestras de cada una de las mil clases del dataset original, con el objetivo de reducir los tiempos de evaluación en comparación con el uso del dataset de validación completo.

Para reducir el espacio de exploración, en las evaluaciones de este capítulo se considera solo un subconjunto de los modelos de clasificación de imágenes analizados en el Capítulo 3. En particular, se emplean las redes **ResNet50** y **ResNet152**. La primera es una de las más utilizadas en la literatura reciente, mientras que la segunda es la más compleja dentro del conjunto de modelos evaluados previamente. Debido a su mayor profundidad, **ResNet152** impone un entorno computacional más exigente, lo que permite dimensionar con mayor precisión los beneficios de utilizar TensorRT.

### 4.1.2. Workflow

El flujo de trabajo para obtener los modelos optimizados en este capítulo sigue el esquema **PyTorch-ONNX-TensorRT** utilizado en el capítulo anterior. No obstante, en esta ocasión se incorporan configuraciones adicionales en la **TensorRT Application** que permiten evaluar el impacto de parámetros como el batch size de configuración y el nivel de optimización. A continuación, se detallan estas configuraciones.

#### Configuración de la TensorRT Application

Para la construcción de los inference engines en cada entorno de operación, es necesario generar una **TensorRT Application**, tal como se describió en la Sección 2.3. En este proceso, se define una configuración en el builder, que permite ajustar parámetros clave como la cuantización, el batch size de configuración y el nivel de optimización.

**Cuantización** En los experimentos de este capítulo, se consideran tres esquemas de cuantización: punto flotante de 32 bits, punto flotante de 16 bits y enteros de 8 bits. Estos esquemas se utilizan para generar los inference engines denominados `TRT fp32`, `TRT fp16` y `TRT int8`, respectivamente.

**Batch size de configuración** El batch size puede configurarse de dos maneras: estático o dinámico.

- **Batch size estático:** Se construye un inference engine optimizado para operar con un batch size de inferencia fijo. Un engine configurado de esta forma no podrá ejecutar inferencias con un batch size distinto al especificado en su construcción.
- **Batch size dinámico:** Se construye un inference engine capaz de operar con un batch size de inferencia dentro de un rango de valores, lo que permite su ajuste durante la ejecución de las inferencias. En este capítulo, la configuración utilizada define un valor mínimo de 1, un valor óptimo de 128 y un valor máximo de 256.

**Nivel de optimización** La TensorRT Application permite configurar distintos niveles de optimización, que varían entre 0 y 5. El nivel 0 aplica el menor esfuerzo de optimización, priorizando la reducción del tiempo de construcción del inference engine, mientras que el nivel 5 aplica el mayor esfuerzo de optimización, a costa de un mayor tiempo de construcción. Los detalles específicos de cada nivel se encuentran en la documentación [30] y se describen brevemente en la Sección 2.3. En la literatura, este parámetro suele omitirse en los reportes experimentales. No obstante, por consistencia con la documentación, en las evaluaciones del Capítulo 3 se utilizó el nivel 3, considerado el valor predeterminado en la construcción de un inference engine.

### 4.1.3. Hardware

#### Plataformas de hardware

Las evaluaciones reportadas en este capítulo se realizan exclusivamente en plataformas de la familia Jetson Orin, las cuales, al momento de este estudio, representan la generación más reciente de sistemas embebidos. La decisión de acotar la evaluación a esta familia responde a varios factores prácticos observados en el análisis del Capítulo 3:

- La capacidad de modificar el modo de consumo energético es una característica exclusiva de las plataformas Jetson y no está disponible en las GPUs de escritorio.
- Las herramientas para el monitoreo del rendimiento varían entre GPUs de escritorio y plataformas embebidas, ya que cada una está diseñada para analizar métricas específicas. A partir de evaluaciones preliminares, se determinó el uso de la herramienta `tegrastats` [49], la cual solo es compatible con plataformas Jetson.
- Como se menciona en la Sección 3.1.3, las plataformas anteriores a la familia Orin han dejado de recibir soporte oficial para las nuevas versiones de librerías de desarrollo. Esto implica que,

Tabla 4.2: Modos de consumo evaluados según la plataforma.

| Plataforma | Modo de Consumo | Power budget [W] | Online CPU |
|---|---|---|---|
| Jetson Orin Nano | Modo PM0 | 15 | 6 |
| | Modo PM1 | 7 | 4 |
| Jetson Orin AGX | Modo PM2 | 30 | 8 |
| | Modo PM1 | 15 | 4 |
| | Modo PM0 | n/a | 12 |

en la práctica, quedarán obsoletas a mediano plazo, reduciendo su relevancia. En contraste, la familia Orin es compatible con entornos de desarrollo más recientes y garantiza soporte de actualizaciones a largo plazo [50].

Por las razones mencionadas, se han seleccionado las plataformas Jetson Orin Nano y Jetson Orin AGX, cuyas principales características se presentan en la Tabla 3.2.

#### Modo de consumo

En las plataformas utilizadas en los experimentos de este capítulo, el parámetro global del modo de consumo energético puede configurarse de forma independiente a las opciones de configuración de la TensorRT Application, como se mencionó anteriormente en la Sección 3.1.3.

Los modos de consumo disponibles dependen directamente de la plataforma de hardware utilizada. En el caso de la Jetson Orin AGX, estos modos van desde `PM0` hasta `PM3`. El modo `PM1` impone las mayores restricciones en términos de la cantidad de CPUs activas, la frecuencia de operación y el *power budget*, mientras que el modo `PM3` aplica las menores restricciones en estos aspectos. Por otro lado, el modo `PM0` representa un estado de consumo energético *sin restricciones*. Los detalles específicos de estos modos se encuentran en la documentación [41] y se describen brevemente en la Sección 3.1.3.

Cabe destacar que este parámetro suele omitirse en los reportes disponibles en la literatura. No obstante, con el objetivo de mantener coherencia con el modo de consumo configurado por defecto al inicializar las plataformas, en las evaluaciones del Capítulo 3 se utilizó el modo `PM0` en la Jetson Orin Nano y el modo `PM2` en la Jetson Orin AGX como configuraciones predeterminadas.

En esta evaluación, se analiza el impacto del modo de consumo en el desempeño de la inferencia en las plataformas Jetson Orin. Para ello, es necesario generar un nuevo inference engine con la plataforma configurada en el modo de consumo correspondiente, ya que el proceso de optimización de TensorRT depende del estado actual del sistema, incluyendo la frecuencia de operación de las CPU disponibles, entre otros factores. De este modo, TensorRT puede ajustar el modelo a las condiciones específicas del modo de consumo seleccionado. Finalmente, los inference engines generados se evalúan en la misma plataforma y bajo el mismo modo de consumo con el que fueron construidos.

La Tabla 4.2 presenta las características de los modos de consumo considerados.

### 4.1.4. Métricas

Para las evaluaciones presentadas en este capítulo, se emplean como métricas de interés la latencia y el throughput, tal como se definieron en el Capítulo 3. Estas métricas son ampliamente utilizadas en evaluaciones relacionadas con el desempeño computacional. Adicionalmente, a partir de las observaciones derivadas del análisis en la Sección 3.3, se incorporan dos nuevas métricas:

**classification closeness** y **uso de memoria durante la inferencia**, las cuales se describen a continuación.

**Classification closeness** Esta métrica evalúa en qué medida las optimizaciones implementadas por TensorRT afectan el valor numérico de la salida de las redes neuronales. Basándose en lo expuesto en [39], se utiliza la función `isclose` de PyTorch [51] para determinar si la desviación entre la salida del modelo base y la del modelo optimizado se mantiene dentro de un rango predefinido. Esta relación se expresa matemáticamente como:

$$|\mathrm{a} - \mathrm{b}| \leq \mathrm{atol} + \mathrm{rtol} \times |\mathrm{b}| \tag{4.1.1}$$

donde $a$ representa la salida del modelo base, $b$ la salida del modelo optimizado, `atol` es la tolerancia absoluta y `rtol` la tolerancia relativa.

En este estudio, se fija `rtol` $= 0$ y se experimenta con distintos valores de `atol`. Para definir estos valores, primero se registran las salidas del modelo base y se determina el valor máximo dentro del **percentil 90** de los datos observados. Posteriormente, se calculan distintos valores de `atol` como un porcentaje de este máximo.

**Uso de memoria durante la inferencia** Esta métrica evalúa el consumo de memoria RAM durante la inferencia y se calcula midiendo el uso de memoria a lo largo de la ejecución de inferencias sobre el dataset de validación. El proceso incluye la carga de imágenes, el preprocesamiento, la inferencia y el postprocesamiento. Durante la ejecución, el consumo de memoria instantáneo se monitorea con la herramienta `tegrastat` [49], configurada con un intervalo de muestreo de 1 milisegundo. A partir de los datos recopilados, se calcula el consumo promedio y máximo de memoria para cada configuración de operación. Finalmente, la medición se repite para distintos valores del batch size de inferencia, obteniendo el uso de memoria en función de este parámetro.

La evaluación de estas métricas se lleva a cabo en distintas configuraciones de operación, las cuales incluyen parámetros previamente considerados, como la cuantización y el batch size de inferencia, junto con nuevas configuraciones, tales como el **batch size de configuración** (estático y dinámico), el **modo de consumo** y el **nivel de optimización**.

## 4.2. Reporte de resultados

A continuación, se presentan los resultados obtenidos al evaluar las diferentes configuraciones. Todos los datos generados durante el experimento están disponibles en el repositorio de GitHub [35].

### 4.2.1. Caracterización de classification closeness

La Tabla 4.3 presenta los resultados de classification closeness obtenidos para las plataformas Jetson Orin Nano y Jetson Orin AGX al utilizar el modelo ResNet152. En dicha tabla, la columna `Accuracy (Top-1)` muestra la precisión de la inferencia Top-1 para las tareas de clasificación revisadas previamente en la Sección 3.1.4, las cuales se emplearon como referencia en este experimento. Las columnas restantes están etiquetadas como `atol`, seguidas de un factor que representa la relación entre el valor máximo reportado por el modelo base en el percentil 90 % y el valor de `atol` utilizado para calcular el closeness reportado. Por ejemplo, la columna `atol 1` muestra los resultados de closeness para un `atol` equivalente al 100 % del valor máximo reportado por el modelo base en el percentil 90 %. Es importante destacar que el valor de classification accuracy (Top-1) para cada modelo evaluado permanece invariable respecto al `atol` considerado.

Tabla 4.3: Classification closeness (%) según diferentes razones de tolerancia absoluta, como se describe en la Sección 4.1.4, en las plataformas Jetson Orin AGX y Jetson Orin Nano con el modelo ResNet152.

| | | Accuracy (Top-1) | atol 0.005 | atol 0.01 | atol 0.1 | atol 0.2 | atol 0.5 | atol 1 |
|---|---|---|---|---|---|---|---|---|
| AGX | TRT fp32 | 82.32 % | 94.32 % | 98.73 % | 100.00 % | 100.00 % | 100.00 % | 100.00 % |
| | TRT fp16 | 82.32 % | 79.19 % | 93.62 % | 99.99 % | 100.00 % | 100.00 % | 100.00 % |
| | TRT int8 | 80.49 % | 3.11 % | 6.24 % | 53.71 % | 79.97 % | 96.66 % | 99.32 % |
| Nano | TRT fp32 | 82.32 % | 94.32 % | 98.71 % | 100.00 % | 100.00 % | 100.00 % | 100.00 % |
| | TRT fp16 | 82.32 % | 79.26 % | 93.58 % | 99.99 % | 100.00 % | 100.00 % | 100.00 % |
| | TRT int8 | 82.04 % | 3.23 % | 6.43 % | 54.86 % | 80.76 % | 96.71 % | 99.32 % |

Los resultados muestran que, al igual que en las evaluaciones del Capítulo 3, los tres modelos optimizados con distintos niveles de cuantización alcanzan una Accuracy Top-1 superior al 80 %, incluyendo el modelo `TRT int8`. Sin embargo, al analizar el classification closeness, se evidencian diferencias significativas en función del nivel de cuantización empleado. Con la representación `TRT fp32`, el error relativo en las salidas es bajo con respecto al modelo base, observándose que más del 94 % de los valores de salida presentan un error inferior al 0.5 %. No obstante, a medida que se reduce la precisión numérica de los parámetros del modelo (pasando de `fp32` a `fp16` y luego a `int8`), el porcentaje de valores que permanecen dentro de este umbral disminuye, alcanzando un 79 % para `TRT fp16` y un 3 % para `TRT int8`. Al considerar distintos rangos de tolerancia, se observa que para lograr una coincidencia del 90 % con el modelo `TRT fp16` es necesario utilizar un `atol` de 0.01. Por otro lado, para alcanzar una coincidencia cercana al 90 % con el modelo `TRT int8`, se requiere un `atol` de 0.5, lo que equivale a aceptar una diferencia del 50 % respecto al valor máximo en el percentil 90.

Los resultados de la Tabla 4.3 indican que, tal como se puede esperar intuitivamente, las representaciones cuantizadas en TensorRT alteran significativamente los valores de salida de la red neuronal. Sin embargo, en las tareas de clasificación, las salidas de la red corresponden a puntajes asociados a cada clase, los cuales son postprocesados para determinar la clase resultante, generalmente seleccionando aquella con la mayor puntuación. En el caso de la métrica classification accuracy (Top-1), lo fundamental es que la clase correcta conserve la mayor puntuación relativa, independientemente del error absoluto introducido por la cuantización con respecto al modelo base. Esta propiedad hace que las tareas de clasificación sean robustas al error en la puntuación de cada clase, lo que explica por qué suelen utilizarse como ejemplo para ilustrar las ventajas de TensorRT.

El error en los valores de salida puede afectar de distintas formas la fidelidad de los resultados, dependiendo del tipo de tarea abordada. En el contexto del procesamiento de imágenes mediante redes neuronales, además de la clasificación, otras tareas comunes son la segmentación y regresión. En el caso de la segmentación, la salida de la red corresponde a una máscara en la que cada píxel de la imagen es clasificado dentro de una categoría específica (por ejemplo, separar objetos de interés del *background*). Aunque los valores de salida de la red para cada píxel son discretos y representan clases específicas, un pequeño error causado por la cuantización de los parámetros de la red puede provocar que un píxel cambie de etiqueta. Este error puede repetirse a lo largo de la imagen, generando regiones mal segmentadas y bordes imperfectos, lo que afecta directamente métricas de accuracy como la IoU. Por otro lado, en tareas de regresión, donde la salida final corresponde a valores continuos, las alteraciones introducidas por la cuantización pueden impactar directamente la precisión de las predicciones, lo que puede generar errores significativos en los resultados.

Como conclusión, si bien la métrica `Top-1 Accuracy` se mantiene estable incluso con cuantización a `int8`, debido a que lo fundamental es la relación relativa entre los puntajes de las clases, el análisis de classification closeness evidencia un aumento significativo en las diferencias de los valores de salida tras la cuantización. Este fenómeno sugiere que otras tareas, como la segmentación y regresión, podrían ser más sensibles a estos efectos. Por lo tanto, es fundamental analizar con mayor detalle el impacto de estas optimizaciones en distintos tipos de tareas, lo que motiva el estudio presentado en el siguiente capítulo.

### 4.2.2. Batch size de configuración estático y dinámico

En esta sección se presentan los resultados de evaluación de la inferencia en modelos optimizados con TensorRT, utilizando las métricas de throughput y uso de memoria durante la inferencia, según lo descrito en la Sección 4.1.4.

Las pruebas se realizaron empleando distintos inference engines. Por un lado, se generaron inference engines con un batch size estático, lo que implica la construcción de un engine específico para cada valor de batch size de inferencia evaluado. Por otro lado, se construyó un inference engine con un batch size dinámico, capaz de operar dentro de un rango de valores de batch size de inferencia.

Los resultados de estas evaluaciones se presentan utilizando el modelo ResNet152, el cual es el de mayor complejidad entre los considerados, permitiendo así una mayor exigencia sobre los recursos de la plataforma.

#### Caracterización del uso de memoria durante la inferencia

La Figura 4.1 muestra los resultados del uso de memoria durante la inferencia en la plataforma Jetson Orin AGX. En esta figura, los marcadores cuadrados representan el uso promedio de memoria para cada batch size de inferencia evaluado, mientras que los marcadores triangulares indican el uso máximo registrado. Este esquema de representación se mantendrá en los gráficos subsecuentes de uso de memoria.

El primer gráfico de la Figura 4.1 muestra el uso de memoria del modelo base. Dado que este modelo no contiene ninguna optimización ni configuración específica respecto al batch size de configuración, su consumo de memoria solo se utiliza como referencia para evaluar el impacto de las optimizaciones aplicadas al resto de configuraciones.

El gráfico `TRT fp32` de la Figura 4.1 muestra el uso de memoria en una configuración con cuantización en `fp32`, comparando el comportamiento entre un batch size estático y uno dinámico. Se observa que, en el caso del batch size estático, el uso de memoria aumenta a medida que se incrementa el batch size de inferencia. En cambio, para el batch size dinámico, el uso de memoria permanece constante, independientemente del batch size de inferencia evaluado.

Como se explicó en la Sección 4.1.2, un inference engine generado con un batch size de configuración dinámico está diseñado para soportar cualquier batch size de inferencia dentro de un rango determinado. Dada esta configuración, se puede deducir que el uso de memoria se satura para acomodar el peor caso, es decir, el batch size de mayor tamaño dentro del rango permitido. En contraste, cuando se emplea un batch size de configuración estático, se construye un inference engine específico para cada batch size de inferencia, optimizado únicamente para dicho tamaño. Esto implica que el uso de memoria es proporcional al batch size para el cual el modelo fue optimizado.

Los gráficos `TRT fp16` y `TRT int8` de la Figura 4.1 presentan un comportamiento similar al de `TRT fp32` al comparar los usos de memoria resultantes entre batch size estático y dinámico. Por otro

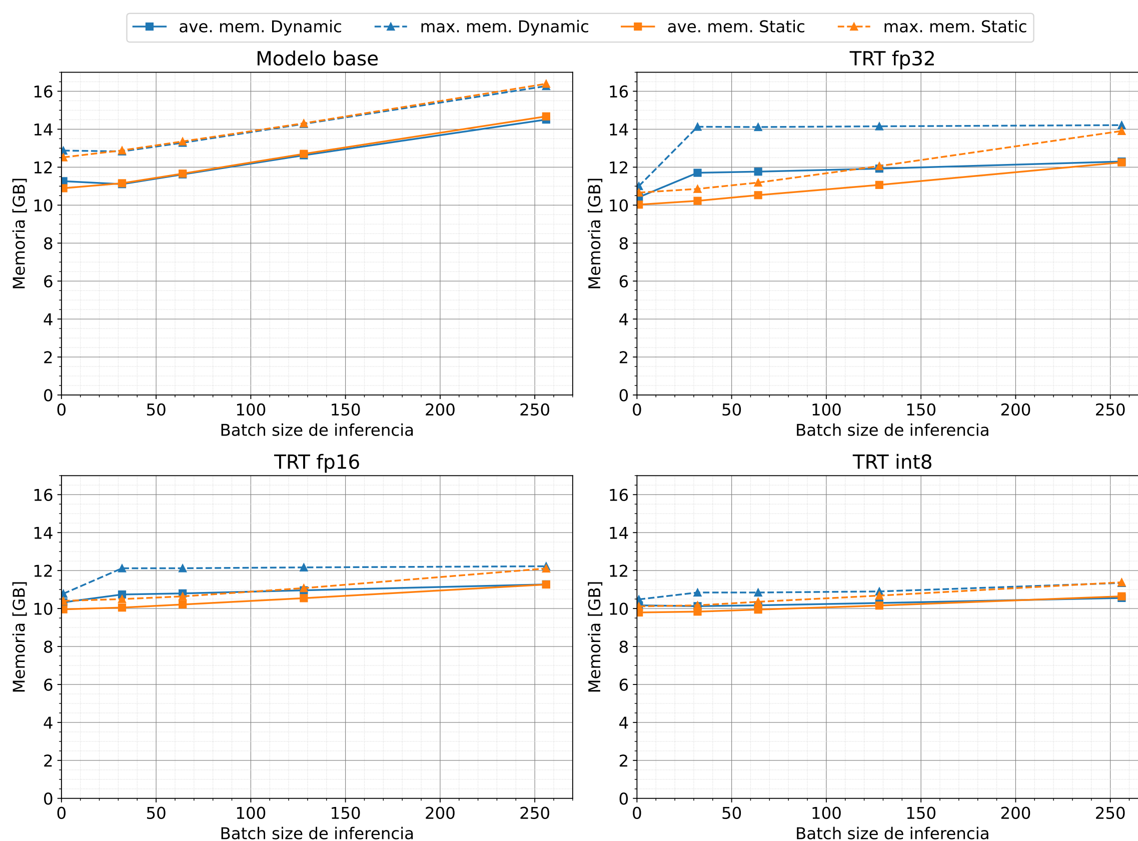
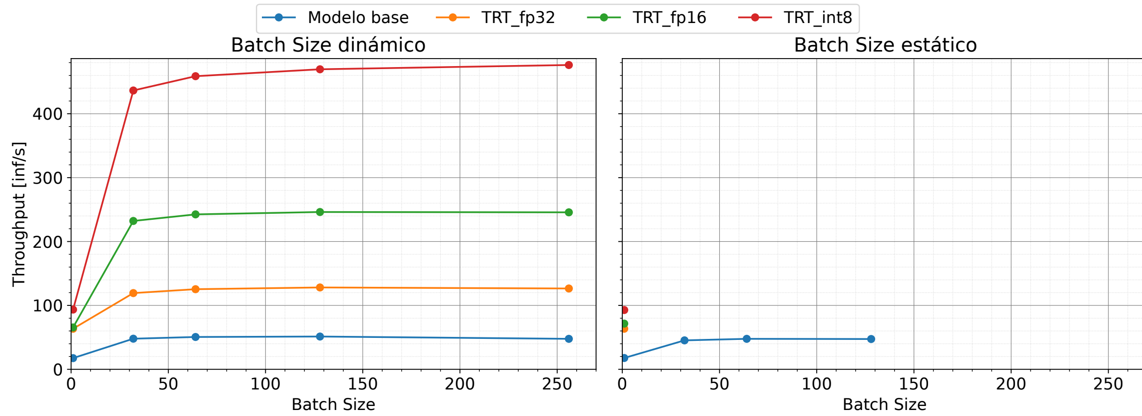



Fig. 4.1: Uso de memoria durante la inferencia en función del batch size de inferencia, comparando batch size de configuración estático y dinámico en el modelo ResNet152 sobre la plataforma Jetson Orin AGX. Se incluyen el modelo base y las optimizaciones de TensorRT: TRT fp32, TRT fp16 y TRT int8.

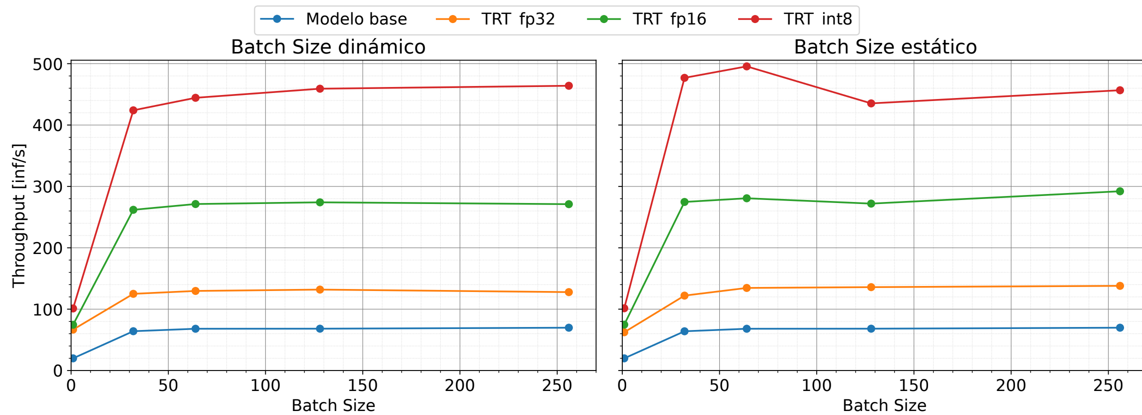
lado, se observa que, a medida que se reduce la precisión numérica de los parámetros del modelo (pasando de `fp32` a `fp16` y luego a `int8`), el uso de memoria disminuye en comparación con el modelo base. Asimismo, la diferencia en el uso de memoria entre las configuraciones estática y dinámica se reduce a medida que se aplican esquemas de cuantización más agresivos, alcanzando una diferencia máxima de un gigabyte al utilizar `TRT int8`.

Cabe mencionar que, aunque se realizaron pruebas en la plataforma Jetson Orin Nano, no fue posible generar inference engines con un batch size de configuración estático y un batch size de inferencia mayor a uno. Por esta razón, no se obtuvieron resultados de uso de memoria para dicha configuración.

Al realizar este experimento, se observó que el proceso de construcción de los inference engines requiere más memoria que el proceso de inferencia. Además, durante el desarrollo del experimento, se constató que la configuración estática demanda más memoria que la configuración dinámica durante la etapa de construcción. Por otro lado, la plataforma Jetson Orin Nano dispone de solo 8 GB de memoria RAM, en contraste con los 64 GB disponibles en la Jetson Orin AGX. Esta limitación de memoria impidió la generación de los inference engines en la Jetson Orin Nano durante la etapa de




(a) Jetson Orin Nano.




(b) Jetson Orin AGX.

Fig. 4.2: Evaluación del throughput en función del batch size de inferencia, comparando configuraciones estática y dinámica en modelos de red ResNet152 sobre las plataformas Jetson Orin Nano y Jetson Orin AGX.

construcción.

Los resultados de esta evaluación indican que, en la práctica, el uso de un batch size de configuración dinámico es más versátil y conveniente, ya que proporciona mayor flexibilidad sin requerir una estimación precisa del batch size de inferencia. Además, esta configuración reduce el consumo de memoria durante el proceso de construcción del engine. Por lo tanto, se recomienda optar por un batch size estático únicamente cuando se conoce con certeza el batch size de inferencia que se empleará y se tiene la seguridad de que la plataforma objetivo dispone de suficiente memoria para la construcción del inference engine.

### Caracterización del throughput

La Figura 4.2 muestra los resultados de throughput en función del batch size de inferencia para las configuraciones de batch size estático y dinámico en las plataformas Jetson Orin Nano y Jetson

Orin AGX.

La Figura 4.2a presenta los resultados obtenidos en la plataforma Jetson Orin Nano. Se observa que, en la configuración de batch size dinámico, el throughput aumenta conforme se incrementa el batch size de inferencia evaluado, siendo el modelo `TRT int8` el que alcanza el mayor throughput. Por otro lado, en la configuración de batch size estático, el gráfico no pudo completarse debido a que TensorRT no logró generar inference engines bajo esta configuración. Esto se debe a la limitada cantidad de memoria disponible en la plataforma Nano. Como se mencionó previamente, esta restricción de memoria impacta directamente la generación de engines en la plataforma.

La Figura 4.2b muestra los resultados obtenidos en la plataforma Jetson Orin AGX, la cual dispone de suficiente memoria para generar todos los engines con batch size de configuración estático y ejecutar inferencias con ellos. En la figura se observa que, en la configuración dinámica, el throughput aumenta a medida que incrementa el batch size de inferencia, siguiendo una tendencia similar a la observada en la Jetson Orin Nano. Por otro lado, en la configuración estática, el throughput no presenta un aumento uniforme, alcanzando sus valores máximos para batch sizes de 32 y 64 al utilizar `TRT int8`. Finalmente, las configuraciones con otros niveles de cuantización exhiben resultados similares entre ambas configuraciones de batch size.

El aumento irregular en el throughput de la configuración estática se atribuye a que, para evaluar cada batch size de inferencia, fue necesario generar un nuevo inference engine. Debido a la naturaleza estocástica de este proceso, los engines generados presentan variaciones en su rendimiento. En contraste, en la configuración dinámica, solo se requiere un único inference engine para evaluar distintos batch sizes de inferencia. El uso del mismo engine en todas las evaluaciones resultó en un rendimiento más uniforme para todos los batch sizes analizados.

#### Selección del batch size de configuración para los experimentos siguientes

Los resultados obtenidos en esta sección indican que el uso de un batch size de configuración dinámico ofrece una mayor flexibilidad en comparación con el batch size estático. Esto se debe a que dicha configuración elimina la necesidad de generar múltiples inference engines para diferentes tamaños de batch de inferencia, lo que simplifica el proceso de experimentación y reduce los requerimientos de memoria durante la construcción de los engines.

Por estas razones, en las secciones siguientes se empleará un batch size de configuración dinámico, estableciendo un batch size mínimo de 1, un valor óptimo de 128 y un máximo de 256.

Si bien esta elección es adecuada para la mayoría de las métricas evaluadas, como el throughput y el uso de memoria con batch sizes de inferencia mayores a uno, las evaluaciones para configuraciones con batch size de inferencia igual a uno se realizaron utilizando un batch size de configuración estático. Esto sigue la metodología establecida para la medición de latencia en la Sección 3.1.4, con el uso específico de un batch size de inferencia igual a uno.

### 4.2.3. Modo de consumo

En esta sección se presentan los resultados de la evaluación del comportamiento de la inferencia en modelos optimizados con TensorRT, configurados según los distintos modos de consumo disponibles, como se indica en la Tabla 4.2. Los resultados se enfocan específicamente en la tarjeta Jetson Orin Nano, lo que permite una presentación concisa de los datos.

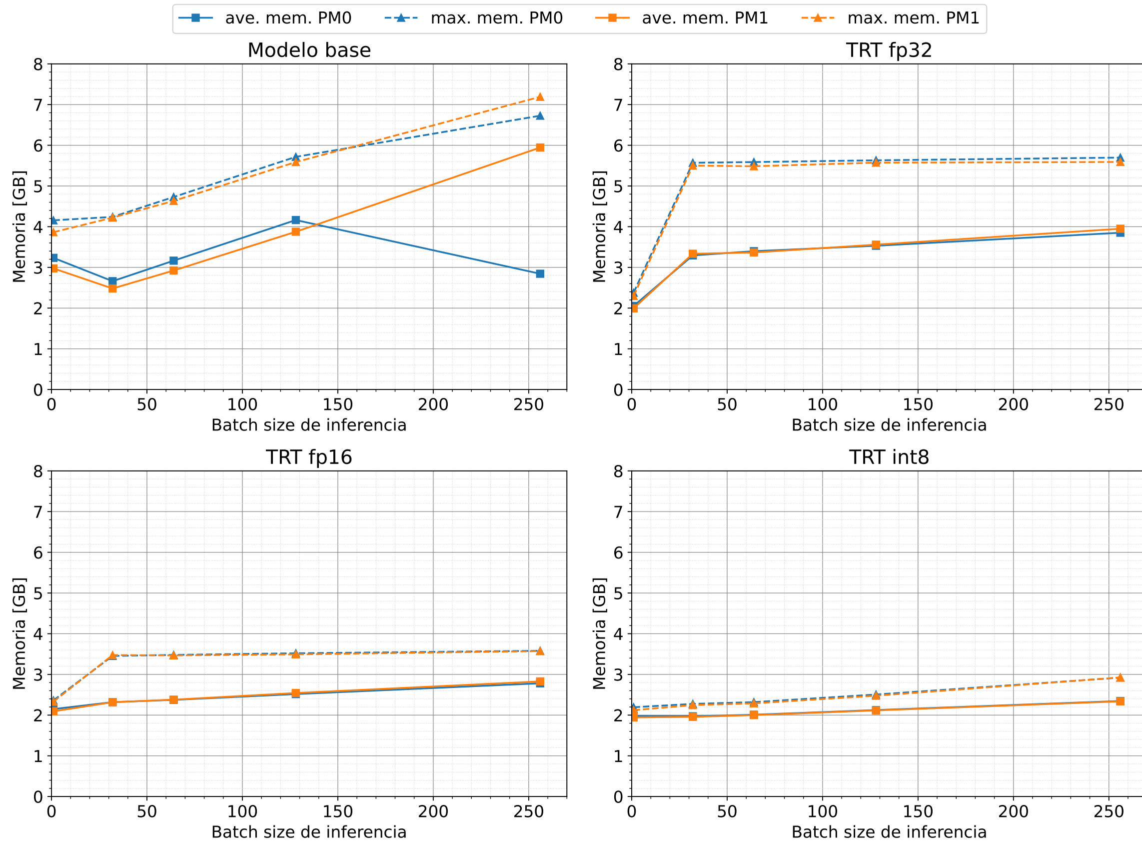



Fig. 4.3: Evaluación del uso de memoria en función del batch size de inferencia, considerando los modos de consumo `PM0` y `PM1` en la plataforma Jetson Orin Nano. Se incluyen los resultados para el modelo ResNet152 en su versión base, junto con las optimizaciones de TensorRT: `TRT fp32`, `TRT fp16` y `TRT int8`.

### Caracterización del uso de memoria durante la inferencia

La Figura 4.3 muestra el uso de memoria durante la inferencia de la red ResNet152. El primer gráfico de la figura presenta los resultados obtenidos con el modelo base. Generalmente, el rendimiento del modelo base no se ve afectado por configuraciones de optimización, ya que estas suelen aplicarse únicamente a modelos transformados mediante TensorRT. Sin embargo, en este caso, la elección del modo de consumo es una configuración aplicada directamente en la plataforma de hardware, por lo que también influye en la ejecución de inferencias con el modelo base. A pesar de esto, no se observan diferencias significativas en el uso de memoria, salvo por un posible *outlier* en la memoria promedio para `PM0`.

El gráfico correspondiente a `TRT fp32` muestra que el modo de consumo no influye en el uso de memoria para ninguna de las configuraciones evaluadas. De manera similar, los resultados para `TRT fp16` y `TRT int8` reflejan el mismo comportamiento observado en `TRT fp32`.

A partir de estos resultados, se concluye que el modo de consumo **no tiene impacto** en el uso de memoria durante la inferencia. Esto puede explicarse por el hecho de que la configuración del

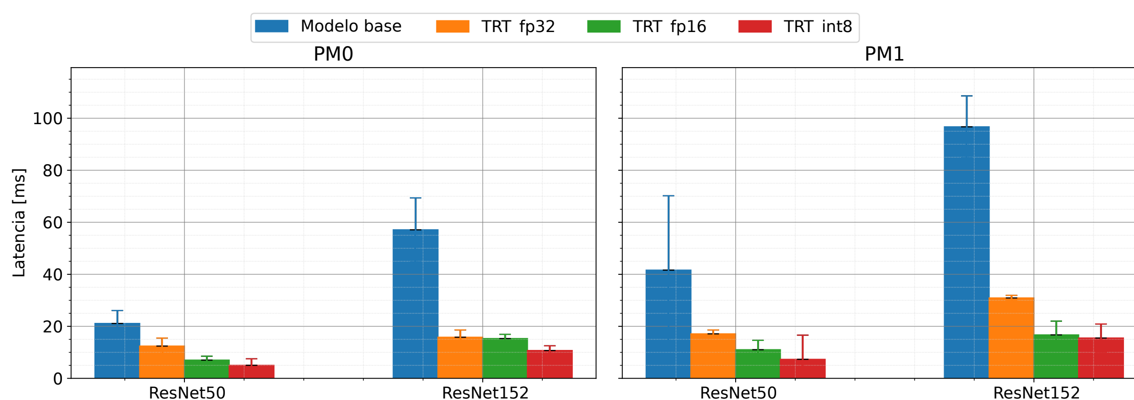



Fig. 4.4: Evaluación de la latencia en función del batch size de inferencia, considerando los modos de consumo `PM0` y `PM1` en la plataforma Jetson Orin Nano. Se presentan los resultados para los modelos ResNet50 y ResNet152.

modo de consumo afecta principalmente el límite energético, el número de núcleos CPU activos y sus frecuencias de operación en el hardware evaluado, sin modificar directamente el uso de memoria del dispositivo.

### Caracterización de la latencia

La Figura 4.4 presenta dos gráficos que muestran los resultados de latencia según el modo de consumo evaluado. Cada gráfico representa la latencia promedio mediante la altura de las barras y la latencia máxima a través de la extensión superior de cada una.

El primer gráfico de la Figura 4.4 corresponde a `PM0`, el modo predeterminado en la Jetson Orin Nano, e incluye los resultados para las redes ResNet50 y ResNet152. El segundo gráfico presenta los resultados para `PM1`, un modo con restricciones adicionales en consumo energético y núcleos activos, considerando las mismas redes.

Los resultados indican que `PM0` permite reducir la latencia en comparación con `PM1` en ambas redes evaluadas. En el modelo base, la latencia promedio de la ResNet50 con `PM0` es un 50 % menor que con `PM1`, mientras que en la ResNet152 la reducción alcanza un 40 %. De manera similar, en las configuraciones optimizadas con TensorRT, se observa una disminución en la latencia al utilizar `PM0` en lugar de `PM1`. Por ejemplo, en la ResNet50 con `TRT fp32`, la latencia promedio se reduce en un 30 %, mientras que en la ResNet152 la reducción es del 50 %. Con `TRT fp16`, la ResNet50 experimenta una disminución del 33 % al usar `PM0`, aunque en la ResNet152 no se observa una reducción significativa. Finalmente, con `TRT int8`, la latencia promedio de la ResNet50 disminuye un 35 %, y en la ResNet152 la reducción es de aproximadamente un 33 %. Además, al analizar las latencias máximas mostradas en la figura, se observa un comportamiento similar al comparar ambos modos de consumo en ambos modelos.

A partir de los resultados obtenidos, se infiere que seleccionar un modo de consumo que permita a la tarjeta aprovechar su máximo potencial, tanto en consumo energético como en el uso de núcleos, contribuye a reducir la latencia en la inferencia. Sin embargo, el impacto del modo de consumo varía según el nivel de cuantización aplicado durante la optimización. Las diferencias más pronunciadas se observaron en el modelo base, donde el cambio de `PM1` a `PM0` redujo la latencia de manera considerable. En contraste, las reducciones de latencia en las configuraciones optimizadas con TensorRT fueron

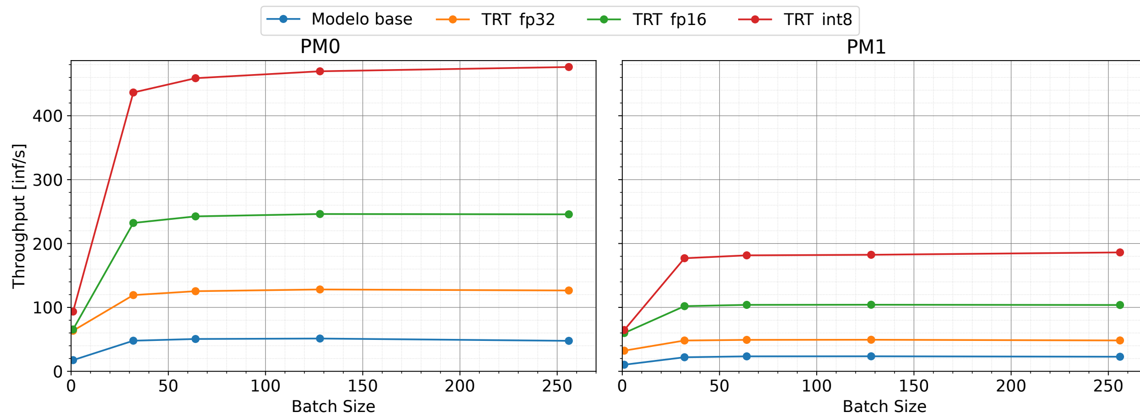



Fig. 4.5: Evaluación del throughput en función del batch size de inferencia, considerando los modos de consumo PM0 y PM1 en la plataforma Jetson Orin Nano. Se presentan los resultados para el modelo ResNet152.

menores y dependieron del nivel de cuantización, siendo más notorias en fp32. Esto sugiere que las mejoras de rendimiento al modificar el modo de consumo están influenciadas por el grado de optimización aplicado al modelo.

#### Caracterización del throughput

La Figura 4.5 presenta los resultados de throughput para la red ResNet152. En el primer gráfico de la figura se muestran los resultados de throughput con la configuración de modo de consumo PM0 de la Jetson Nano, mientras que en el gráfico de la derecha se presentan los resultados con el modo de consumo PM1.

Al comparar ambos gráficos, se observa que la elección del modo de consumo afecta significativamente el throughput. En particular, seleccionar el modo PM0 en lugar de PM1 incrementa el throughput a más del doble para la mayoría de los valores de batch size evaluados. Por lo tanto, optar por un modo de consumo que permita a la plataforma aprovechar todo su potencial, tanto en consumo energético como en el uso de núcleos de procesamiento, resulta en un aumento consistente del throughput.

Estos resultados muestran tendencias similares en otros modelos, como ResNet50, así como en la plataforma Jetson Orin AGX.

### 4.2.4. Nivel de optimización

En esta sección se presentan los resultados de la evaluación del comportamiento de la inferencia en modelos optimizados con TensorRT con los niveles de optimización descritos en la Sección 4.1.2. Los resultados se reportan específicamente en la plataforma Jetson Orin AGX, lo que permite exponerlos de manera concisa y consistente con el comportamiento observado en otras configuraciones.

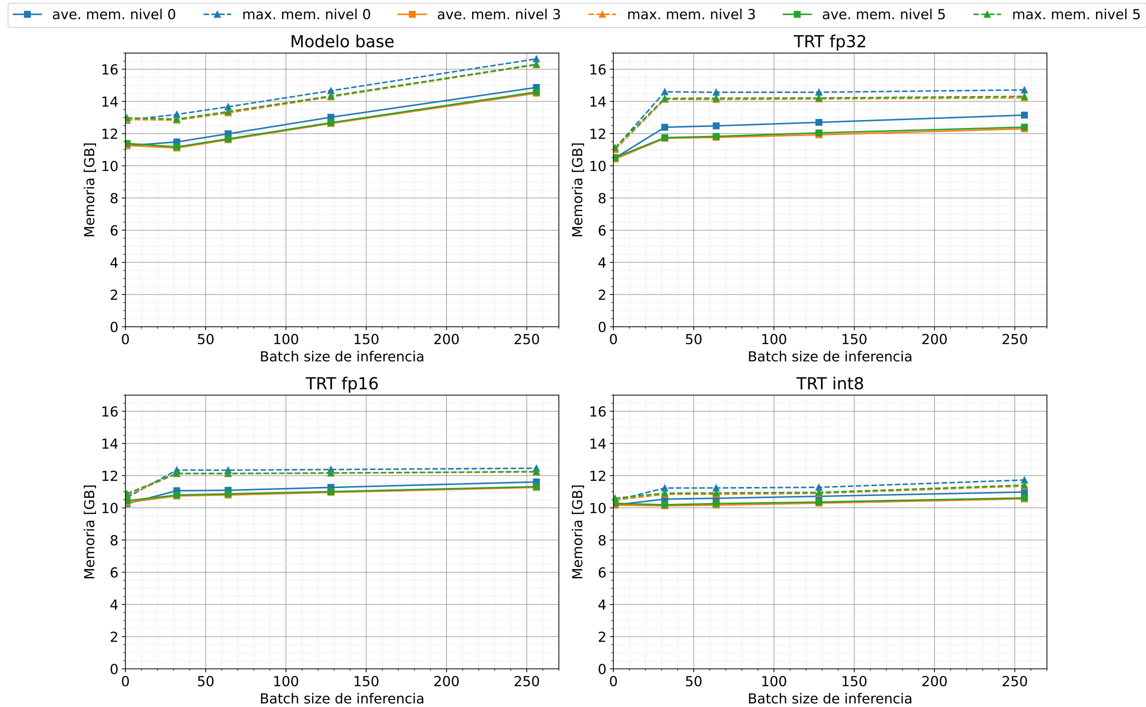



Fig. 4.6: Evaluación del uso de memoria durante la inferencia en función del batch size, considerando los niveles de optimización 0, 3 y 5 en la plataforma Jetson Orin AGX. Se analiza la red ResNet152 en su modelo base y con las optimizaciones de TensorRT: `TRT fp32`, `TRT fp16` y `TRT int8`.

### Caracterización del uso de memoria durante la inferencia

La Figura 4.6 muestra los resultados del uso de memoria durante la inferencia al evaluar los distintos niveles de optimización. El primer gráfico de la figura presenta el uso de memoria del modelo base, el cual se emplea como referencia para los modelos optimizados. Es importante recordar que la configuración del nivel de optimización no afecta al modelo base, ya que esta solo influye en los modelos optimizados mediante TensorRT.

El segundo gráfico de la Figura 4.6 muestra el uso de memoria en `TRT fp32`, donde no se observan cambios significativos al comparar los distintos niveles de optimización, salvo un leve aumento en los valores promedio y máximo para el `nivel 0`. Los gráficos correspondientes a `TRT fp16` y `TRT int8` exhiben un comportamiento similar al de `TRT fp32`, donde únicamente el `nivel 0` presenta un ligero incremento en el consumo de memoria.

A partir de estos resultados, se concluye que el nivel de optimización no impacta de manera significativa en el uso de memoria. Sin embargo, el `nivel 0` muestra un mayor consumo de memoria en comparación con los niveles 3 y 5 que se mantienen prácticamente idénticos. Este comportamiento podría explicarse por el hecho de que el nivel de optimización no incide directamente en la aplicación de técnicas de TensorRT para reducir el tamaño del modelo en disco. En su lugar, este nivel influye en la selección de los kernels utilizados en la ejecución del modelo optimizado. Dichos kernels no parecen afectar de manera significativa el uso de memoria, con la excepción del nivel de optimización 0.

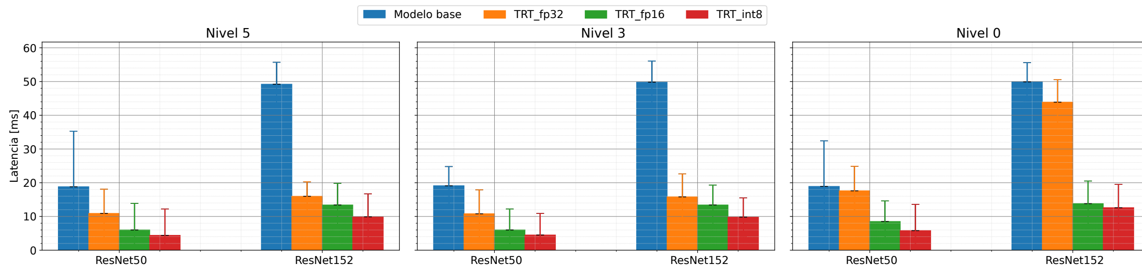



Fig. 4.7: Latencia de los modelos ResNet50 y ResNet152 en la plataforma Jetson Orin AGX para los niveles de optimización 0, 3 y 5.

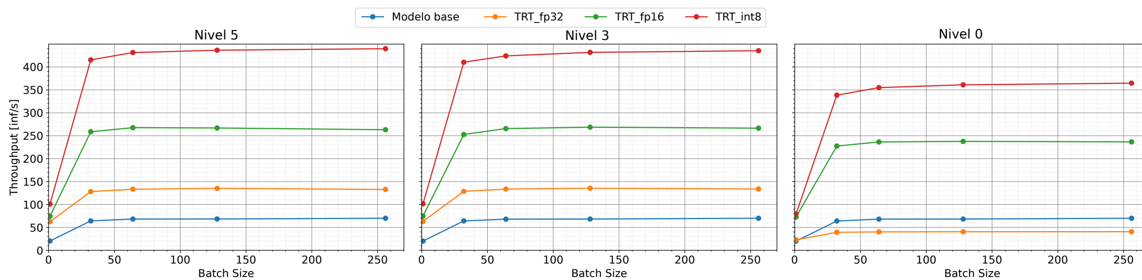



Fig. 4.8: Comparación del Throughput en función del batch size de inferencia en red ResNet152 en plataforma Jetson Orin AGX en niveles de optimización 0, 3 y 5.

### Caracterización de la latencia

La Figura 4.7 presenta los resultados de latencia obtenidos considerando tres niveles de optimización: `nivel 5`, `nivel 3` y `nivel 0`. Cada uno de los gráficos de la figura muestra las latencias promedio y máximas, representadas por la altura de las barras y la extensión superior de cada una.

La Figura 4.7 presenta los resultados obtenidos para las redes ResNet50 y ResNet152 en tres niveles de optimización. El primer gráfico corresponde al `nivel 5`, el segundo al `nivel 3` y el último al `nivel 0`.

En primer lugar, la configuración del nivel de optimización no afecta los resultados del modelo base, ya que esta se aplica únicamente a los modelos optimizados por TensorRT. Por lo tanto, los resultados del modelo base se presentan únicamente como referencia en cada gráfico.

En cuanto a `TRT fp32`, se observa que la latencia promedio no varía al elegir el `nivel 5` en lugar del `nivel 3` para ninguna de las redes. Por otro lado, al utilizar el `nivel 0`, se registra un aumento significativo en la latencia promedio, alcanzando aproximadamente un 66 % más en comparación con el nivel predeterminado. Para las optimizaciones `TRT fp16` y `TRT int8`, se observa un comportamiento similar, con incrementos en la latencia promedio al emplear el `nivel 0` en lugar del `nivel 3`.

En cuanto a la latencia máxima, esta no se ve afectada de forma significativa por los niveles de optimización evaluados.

En general, se observa que no existe un impacto significativo en la latencia entre los niveles de optimización 5 y 3. Sin embargo, seleccionar el `nivel 0` puede aumentar significativamente la latencia, especialmente al usar optimizaciones como `TRT fp32`.

#### Caracterización del throughput

La Figura 4.8 presenta los resultados de throughput obtenidos considerando tres niveles de optimización: `nivel 5`, `nivel 3` y `nivel 0`. En la figura, el primer gráfico muestra el throughput correspondiente al `nivel 5`, seguido por el `nivel 3` y, finalmente, el `nivel 0`.

Al comparar los resultados obtenidos para los niveles 5 y 3, se observa que la elección del `nivel 5` sobre el `nivel 3` no produce un aumento significativo en el throughput. Esto sugiere que las optimizaciones adicionales implementadas en el `nivel 5` no generan una mejora considerable en el rendimiento computacional con respecto al `nivel 3`. Una posible explicación es que el modelo ya ha alcanzado un punto de eficiencia en el cual las optimizaciones adicionales no logran aportar un beneficio sustancial en términos de desempeño.

Por otro lado, la construcción del inference engine con un `nivel 0` de optimización impacta negativamente en el desempeño computacional. En este caso, se observa una reducción significativa en el throughput en comparación con los otros niveles de optimización. En particular, el inference engine generado con un `nivel 0` de optimización y cuantización `fp32` muestra un rendimiento inferior incluso al del modelo base. Esto sugiere que la ausencia de optimización no solo afecta la eficiencia del modelo, sino que también puede introducir cuellos de botella computacionales que limitan su desempeño.

Los resultados de este experimento destacan la importancia de elegir un nivel de optimización adecuado para maximizar el throughput. El análisis de resultados mostró que el `nivel 5` no ofrecía mejoras sustanciales sobre el `nivel 3`; sin embargo, la selección del nivel de optimización podría depender de la aplicación específica, dado que en ciertos escenarios el `nivel 3` podría no alcanzar una eficiencia cercana a la del `nivel 5`. Por otro lado, aunque el `nivel 0` presenta un throughput significativamente inferior, en este experimento no se evaluó el tiempo de construcción del engine, un aspecto relevante en etapas tempranas del desarrollo. En particular, su uso puede ser ventajoso para la evaluación y prueba de modelos durante la fase de construcción, donde la rapidez en la generación del inference engine facilita el ajuste de parámetros como el batch size y la cuantización antes de seleccionar el nivel de optimización más adecuado. Estos aspectos se analizarán en mayor profundidad en el capítulo siguiente.

### 4.2.5. Discusión de resultados

Los resultados analizados en las secciones previas constituyen un primer acercamiento hacia un análisis completo sobre la optimización de la inferencia usando TensorRT, al abordar las limitaciones y vacíos identificados en la literatura, aspectos discutidos en la Sección 3.3 del capítulo anterior.

Al evaluar la métrica de **classification closeness**, se observó que las optimizaciones de TensorRT que reducen la precisión de los parámetros del modelo pueden tener un impacto significativo en las salidas del mismo, un efecto que no se refleja en las métricas típicas de tareas de clasificación. La degradación de las salidas de los modelos optimizados encontrada al analizar closeness en modelos cuantizados a `int8` podría afectar significativamente aplicaciones que emplean tareas como la regresión o la segmentación, las cuales aún están pendientes de evaluación.

En cuanto al **batch size de configuración**, se observó que la construcción de un inference engine con un batch size estático puede requerir un mayor uso de memoria en comparación con batch size dinámico. Además, la configuración dinámica proporciona una mayor versatilidad al permitir el uso de distintos batch sizes de inferencia con un único inference engine, lo que la convierte en la opción más recomendable para la mayoría de los casos. No obstante, un batch size estático resulta preferible cuando se busca reducir la latencia, dado que permite priorizar inferencias con un batch size de uno.

Asimismo, es una opción adecuada cuando se tiene la certeza de que la aplicación siempre operará con un batch size de inferencia fijo.

En relación con el **modo de consumo**, para la mayoría de las aplicaciones centradas en la latencia con restricciones en el consumo energético, se recomienda seleccionar un modo que limite la potencia al nivel necesario para la aplicación. El uso de modos más permisivos en este aspecto no demuestra un impacto consistente en la latencia. Sin embargo, si se busca aumentar el throughput y no existen restricciones de consumo energético, un modo de consumo que permita a la tarjeta utilizar toda su potencia y activar todos los núcleos puede aumentar significativamente las inferencias por segundo para la mayoría de los tamaños de batch de inferencia.

Por último, respecto al **nivel de optimización**, se observó que el `nivel 5` no ofrece mejoras significativas en ninguna de las métricas analizadas en comparación con los resultados obtenidos con el `nivel 3`, que es el predeterminado. Sin embargo, la elección del nivel de optimización puede depender de la aplicación específica, ya que en ciertos escenarios el `nivel 3` podría no alcanzar la eficiencia del `nivel 5`, lo que justificaría evaluaciones adicionales en otros casos de estudio. Por otro lado, al utilizar el `nivel 0` en lugar del `nivel 3`, se evidenció un aumento en la latencia, una reducción en el throughput y un mayor consumo de memoria. Estos resultados indican que, en la mayoría de los casos, el `nivel 3` es la opción más recomendable. Sin embargo, cabe destacar que este análisis no consideró el tiempo requerido para generar los inference engines. En situaciones donde se necesite evaluar rápidamente distintos parámetros del modelo antes de seleccionar el nivel de optimización definitivo, podría ser conveniente emplear niveles de optimización inferiores para acelerar el proceso de experimentación.

El capítulo siguiente abordará principalmente la evaluación de otras aplicaciones más allá del benchmark, poniendo a prueba tareas como la segmentación y regresión.

Capítulo 5

# EVALUACIÓN DE APLICACIONES

En este capítulo se presentan los resultados de la evaluación experimental del uso de TensorRT en dos aplicaciones prácticas distintas a la típica clasificación de imágenes: la segmentación de salmones en imágenes submarinas y la regresión para la implementación de pirometría de hollín en llamas laminares. El objetivo de esta evaluación es validar los hallazgos y las directrices de optimización derivadas en los capítulos anteriores, aplicándolos a contextos diferentes de los comúnmente reportados en la literatura.

Con el objetivo de facilitar la reproducibilidad de los resultados, se han habilitado repositorios públicos en GitHub [52,53]. Estos repositorios incluyen el código fuente y las configuraciones utilizadas en cada evaluación, junto con instrucciones detalladas para replicar la generación de aplicaciones con TensorRT en las distintas configuraciones documentadas en este trabajo. Esta estructura garantiza la posibilidad de escrutinio y promueve la aplicación práctica de los hallazgos en futuros desarrollos y estudios.

## 5.1. Segmentación de salmones

El monitoreo del comportamiento de los peces mediante el análisis de video capturado en jaulas de cultivo representa una alternativa no invasiva y eficiente frente a los métodos manuales tradicionales utilizados para la estimación de biomasa y la detección de enfermedades [54, 55]. La integración de tecnologías como visión artificial y aprendizaje profundo en el procesamiento de imágenes submarinas tiene el potencial de transformar la industria acuícola, al proporcionar datos más precisos y en mayor cantidad, lo que habilita el desarrollo de sistemas inteligentes de monitoreo y estimación de biomasa, facilitando así una gestión más efectiva de la producción.

Una tarea fundamental en el monitoreo basado en visión artificial es la segmentación de instancias de peces, la cual permite identificar y aislar los píxeles correspondientes a cada individuo en una imagen. Una segmentación precisa es un paso fundamental para la extracción de características relevantes, las cuales son utilizadas en distintos procesos, como por ejemplo, la estimación de biomasa y en la evaluación del estado de salud de los peces. Sin embargo, la segmentación de peces en entornos submarinos presenta múltiples desafíos, como la turbidez del agua, la iluminación dinámica, la propagación espectral desigual, el bajo contraste y la presencia de elementos en suspensión, entre los que se incluyen vegetación flotante, residuos de alimento y burbujas [56, 57].

En este contexto, trabajos previos realizados en la Universidad han evaluado distintos algoritmos de segmentación de peces, concluyendo que los métodos basados en Redes Neuronales Convolucionales (CNN) presentan un mejor desempeño al operar sobre imágenes capturadas en entornos sub-

marinos característicos de jaulas de cultivo. En particular, la memoria de título presentada en [58] describe un sistema basado en el modelo `yolov8l-seg`, el cual alcanza una precisión superior al 60 % en la segmentación de salmones. No obstante, dicha evaluación se enfocó principalmente en la validación del sistema a nivel algorítmico y funcional, y las pruebas fueron realizadas en computadores de escritorio sin un mayor análisis sobre el desempeño computacional. Para avanzar hacia una implementación en sistemas de monitoreo autónomos, es necesario optimizar el modelo para su ejecución eficiente en hardware embebido, de modo que pueda integrarse en vehículos operados de forma remota (ROV) submarinos.

### 5.1.1. Entorno de operación

Aquí se describirá el entorno de operaciones utilizado para las evaluaciones de esta sección, el cual incluye la aplicación, el workflow, el hardware y las métricas empleadas. Los elementos que conforman este entorno se detallan en las subsecciones siguientes.

#### Contexto de aplicación

Para la evaluación experimental en este contexto de aplicación, se busca implementar una red para la segmentación de salmones, siguiendo los procedimientos descritos en [58]. Las evaluaciones presentadas en este trabajo forman parte de un proyecto de desarrollo tecnológico en curso. Dado que los métodos y conjuntos de datos asociados al diseño y entrenamiento del modelo de red neuronal están protegidos por propiedad intelectual, no es posible su publicación ni distribución. Por lo tanto, se proporciona únicamente una descripción general del entorno de operación, junto con algunos ejemplos de resultados de inferencia que permitan ilustrar aspectos relevantes para el análisis desarrollado en torno a la optimización de los modelos para su ejecución en GPUs.

Para las evaluaciones, se utiliza el modelo `yolov8l-seg`, especializado en **segmentación de objetos** y proporcionado por Ultralytics [5]. Este modelo fue reentrenado específicamente para la segmentación de salmones, empleando un conjunto de datos privado compuesto por imágenes submarinas capturadas en un entorno de acuicultura. Dichas imágenes presentan variaciones en sus dimensiones, tanto en altura como en anchura, e incluyen tanto cuadros con uno o más salmones como cuadros que contienen únicamente elementos del entorno, tales como las mallas de las jaulas y otros componentes característicos del hábitat.

El conjunto de datos se encuentra dividido en dos particiones principales: un conjunto de entrenamiento, compuesto por 715 imágenes, y un conjunto de validación, con 86 imágenes. Cada imagen está acompañada de sus respectivas etiquetas de segmentación, organizadas en directorios separados para las fases de entrenamiento y validación. La estructura del conjunto de datos se define mediante un archivo de configuración en formato YAML, el cual especifica las rutas hacia las imágenes y sus etiquetas correspondientes. Además, en este archivo se indica que la única clase objetivo es `salmon`.

El contexto de aplicación considera que el modelo de segmentación debe ser optimizado para su ejecución en un ROV equipado con una cámara y una plataforma Jetson. Se requiere que el proceso de segmentación se realice directamente en el ROV, mientras que los resultados obtenidos se envíen a un computador externo ubicado en la superficie para su posterior análisis. Debido a estas condiciones, es necesario encontrar un equilibrio entre la calidad de la segmentación, un alto throughput y un uso eficiente de la memoria.

### Workflow

El flujo de trabajo empleado para obtener los inference engines derivados del modelo base `yolov8l-seg` sigue el esquema **PyTorch-ONNX-TensorRT** descrito en el Capítulo 3. Sin embargo, con el objetivo de aprovechar funcionalidades como la medición de accuracy y la visualización de las máscaras de segmentación generadas durante el proceso de inferencia, es necesario que el modelo optimizado mantenga compatibilidad con el framework de Ultralytics. Para lograrlo, se debe prestar especial atención a la configuración específica tanto de ONNX como de la TensorRT Application.

En primer lugar, se configuran los nombres de las entradas y salidas esperados por Ultralytics en el graph format ONNX. Esta configuración se realiza especificando el siguiente formato:

- **Entradas:** `images: batch0, height, width`
- **Salidas:**
  - `output0 = batch, anchors`
  - `output1 = batch, mask height, mask width`

Estas configuraciones se definen en el *script* encargado de transformar el modelo base a ONNX.

Por otro lado, el conjunto de datos utilizado para la evaluación de los resultados contiene imágenes con dimensiones variables tanto en ancho como en alto. Para manejar esta variabilidad, se configuró la aplicación TensorRT para aceptar dimensiones de entrada dinámicas. Esto se logró especificando, en la configuración de TensorRT, los valores correspondientes a las dimensiones mínima, óptima y máxima para el alto y el ancho de las imágenes de entrada. En este caso, se establecieron valores mínimos de 320 píxeles, óptimos de 640 píxeles y máximos de 960 píxeles, dado que Ultralytics utiliza 640 píxeles por defecto.

Es importante destacar que la configuración de las dimensiones dinámicas debe realizarse con cuidado, considerando las características del conjunto de datos, ya que una configuración inadecuada puede afectar el desempeño del inference engine. Por ejemplo, valores mal definidos pueden provocar un consumo excesivo de memoria en tiempo de ejecución. Un caso particular de este tipo de anomalías se describe en el Apéndice C, donde se analiza el impacto de asignar un valor mínimo de dimensión igual a cero para la altura y la anchura.

Para crear los inference engines en cada configuración de operación, es necesario generar una TensorRT Application, siguiendo el procedimiento descrito anteriormente y complementado con lo expuesto en la Sección 2.3. Para ello, se define una configuración en el builder, que permite ajustar parámetros como la cuantización, el batch size, el nivel de optimización y las dimensiones de las imágenes de entrada. Para más detalles sobre esta implementación, refiérase al repositorio [52].

### Hardware

La Tabla 3.2 presenta las características de las plataformas de hardware seleccionadas, las cuales corresponden a una plataforma embebida Jetson Orin AGX y un computador de escritorio equipado con una tarjeta gráfica RTX 3060. La Jetson Orin AGX es de principal interés para las evaluaciones realizadas, debido a su capacidad para ejecutar el modelo directamente en el ROV, mientras que el computador de escritorio se utiliza como referencia para la comparación de rendimiento.

#### Métricas

A continuación, se describe la metodología empleada para obtener las métricas evaluadas en este capítulo. Cabe destacar que las métricas utilizadas se basan en las definidas en capítulos anteriores, pero han sido adaptadas al contexto específico de la aplicación objetivo.

Las métricas relevantes para el contexto de operación consideran las **propiedades del modelo**, el **uso de memoria durante la inferencia**, el **throughput** y la **latencia**. La inclusión de la latencia permite evaluar el tiempo de respuesta del modelo bajo escenarios de inferencia con un batch size igual a uno. Estas métricas se definen y miden de manera idéntica a lo descrito en la Sección 3.1.4.

Por otro lado, las métricas asociadas al desempeño funcional, como accuracy y closeness, descritas en los capítulos previos, fueron definidas para el contexto de clasificación de imágenes. Por lo tanto, deben ser redefinidas para el contexto de segmentación. Para esta evaluación, se introducen las siguientes métricas:

**Segmentation accuracy** La precisión de la inferencia en la tarea de segmentación de objetos se mide utilizando la métrica de *mean average precision* (mAP). En particular, la métrica **mAP50** evalúa la capacidad del modelo para segmentar objetos correctamente, considerando un solapamiento mínimo del 50 % entre la predicción y el ground truth, según la métrica de intersección sobre unión (IoU). Por otro lado, **mAP50-95** calcula el promedio de la precisión media en diferentes umbrales de IoU, desde el 50 % hasta el 95 %, con incrementos de 5 %. Esto permite evaluar la precisión del modelo en la segmentación de objetos bajo distintos niveles de solapamiento. Esta métrica se calcula utilizando las funcionalidades integradas de Ultralytics [59] sobre el conjunto de validación descrito en la Subsección 5.1.1. Los valores de mAP se reportan tanto para las máscaras de segmentación como para las *bounding boxes* (cajas delimitadoras de los objetos). Sin embargo, en este experimento se presentan exclusivamente los valores de mAP correspondientes a las bounding boxes.

**Segmentation closeness** En este experimento, la salida de los modelos consiste en arreglos que representan, entre otros elementos, máscaras de segmentación y bounding boxes. El postprocesamiento de esta información permite obtener los píxeles correspondientes a los objetos de interés. Para esta aplicación, el segmentation closeness se define como la proporción de píxeles coincidentes dentro de la región segmentada en las imágenes postprocesadas a partir del modelo base y de un modelo optimizado en evaluación. La comparación se realiza exclusivamente sobre los píxeles dentro de la región delimitada por la máscara de segmentación generada por el modelo base.

**Tiempo de construcción** El tiempo de construcción se define como el tiempo que toma el proceso de optimización para convertir un modelo base en un inference engine. El tiempo de construcción se mide utilizando el comando `time` en la terminal al ejecutar el script encargado de construir los engines, reportándose el *real time* proporcionado por dicho comando, el cual corresponde al tiempo total utilizado por el proceso, en este caso reportado en minutos.

### 5.1.2. Metodología de evaluación y configuración

Para evaluar los efectos de distintas configuraciones de parámetros en las métricas definidas previamente, se generó un conjunto de inference engines, considerando combinaciones específicas de plataforma de hardware, modelo de red, cuantización, batch size de configuración y batch size de inferencia. Además, se tuvieron en cuenta aspectos como el modo de consumo energético y el nivel de optimización.

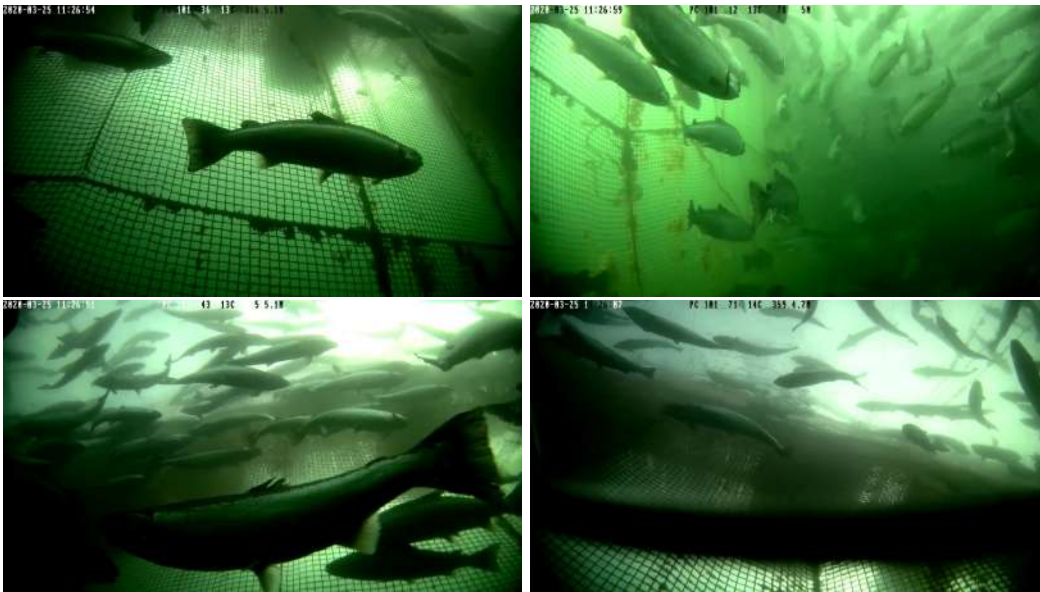


Fig. 5.1: Imágenes representativas extraídas del conjunto de datos de validación: *Img1*, *Img2*, *Img3* e *Img4*.

El proceso de evaluación desarrollado en este capítulo contempla la optimización incremental del modelo de segmentación, mediante la aplicación secuencial de parámetros en la generación del engine. Para ello, se ajusta un parámetro mientras los demás se mantienen fijos. Este proceso comienza con la determinación del nivel de **cuantización** adecuado, de manera que el desempeño de la red se mantenga comparable al del modelo base en términos de segmentation accuracy. Una vez identificada la cuantización que satisface los requisitos de accuracy, y tras evaluar la segmentation closeness con el objetivo de analizar la robustez del modelo ante la cuantización, se procede a optimizar la configuración del **batch size** que permita maximizar el **throughput**. Finalmente, se exploran configuraciones adicionales con el fin de analizar los trade-offs en función del **nivel de optimización** durante la generación del engine y el **modo de consumo**, disponible solo en la tarjeta Jetson.

### 5.1.3. Resultados y análisis

A continuación, se presentan los resultados obtenidos al evaluar las diferentes configuraciones. Todos los datos generados durante el experimento están disponibles en el repositorio de GitHub [52].

En la Figura 5.1 se muestran imágenes de muestra extraídas del dataset de validación. Estas imágenes son representativas del entorno de uso en una aplicación real y permiten visualizar ejemplos concretos de las condiciones bajo las cuales se realizó la evaluación del sistema. La selección de estas muestras busca destacar tanto la variedad como la complejidad de los escenarios analizados.

#### Caracterización de las propiedades del modelo

Como instancia previa a la evaluación del nivel de cuantización, es necesario caracterizar las propiedades de los modelos generados para las evaluaciones siguientes, en las cuales se compara el

Tabla 5.1: Propiedades del modelo: modelo base, TRT fp32, TRT fp16 y TRT int8 generados con batch size de configuración dinámico.

| **Hardware** | **Modelo** | **Propiedades** | | |
|---|---|---|---|---|
| | | Tamaño [MB] | # Capas | # Parámetros |
| RTX 3060 | Modelo Base | 88.0 | 402 | 45936819 |
| | TRT fp32 | 195.9 | 367 | 45887952 |
| | TRT fp16 | 90.8 | 380 | 45887952 |
| | TRT int8 | 48.8 | 278 | 45887952 |
| Jetson Orin AGX | Modelo Base | 88.0 | 402 | 45936819 |
| | TRT fp32 | 181.6 | 401 | 45887952 |
| | TRT fp16 | 92.0 | 286 | 45887952 |
| | TRT int8 | 48.6 | 281 | 45887952 |

modelo base con sus versiones cuantizadas en `fp32`, `fp16` e `int8`.

La Tabla 5.1 presenta las propiedades de los modelos evaluados en ambas plataformas. En ella se compara el tamaño de los modelos optimizados con el del modelo base, destacando un hecho inesperado: el modelo en formato `fp16` resulta más pesado que el modelo base, a pesar de que, intuitivamente, se esperaría lo contrario. Este incremento en el tamaño podría deberse tanto a la configuración dinámica del tamaño de las entradas de la red, descrita en la Sección 5.1.1, como a propiedades intrínsecas del modelo base `yolov8l-seg` y la forma en que es gestionado internamente por TensorRT. No obstante, se observa que el modelo optimizado con TensorRT en `fp16` es más compacto que su versión en `fp32`, y que el modelo en `int8` presenta una reducción adicional respecto a `fp16`. Este comportamiento es consistente con las observaciones de la Sección 3.2.2. Además, se mantiene la tendencia esperada y previamente reportada de reducción en el número de capas con respecto al modelo base, mientras que el número de parámetros permanece constante.

### Caracterización de la configuración de la cuantización

Esta evaluación tiene como objetivo identificar un nivel de cuantización adecuado que permita mantener el desempeño de la red en un nivel comparable al modelo base, considerando las métricas de segmentation accuracy y segmentation closeness, al aplicar cuantización en `fp32`, `fp16` e `int8`. Para todas las pruebas, se emplea un batch size de inferencia igual a uno. Esta elección se justifica debido a que el desempeño se mantiene consistente bajo dicha configuración, tal como se evidenció en los resultados presentados en la Sección 3.2.1.

La Tabla 5.2 presenta los resultados de la precisión de la inferencia, evaluada mediante **mAP50** y **mAP50-95**, según lo descrito en la Subsección 5.1.1, sobre el conjunto de validación. Los resultados indican que la cuantización a `fp32` y `fp16` no produce cambios significativos en la precisión de la inferencia. Sin embargo, al cuantizar los parámetros a `int8`, se observa una disminución considerable en la precisión. En particular, la métrica **mAP50** experimenta una reducción de más de 10 puntos porcentuales en ambas plataformas evaluadas.

En la Figura 5.2 se ilustra la comparación de resultados de segmentación a partir de un ejemplo representativo de entrada. La primera imagen de la figura muestra la segmentación obtenida por el modelo base, utilizada como referencia. A su derecha, se observa la segmentación generada por el modelo `TRT fp32`. En la esquina inferior izquierda se presenta el resultado del modelo `TRT fp16`, mientras que en la esquina inferior derecha se muestra el desempeño del modelo `TRT int8`. En la

Tabla 5.2: Resultados de accuracy en porcentaje (%).

| Hardware | Accuracy | Modelo base | TRT fp32 | TRT fp16 | TRT int8 |
|---|---|---|---|---|---|
| RTX 3060 | **mAP50** | 62.0 | 61.4 | 61.4 | 48.0 |
| | **mAP50-95** | 40.0 | 39.7 | 39.6 | 29.8 |
| Jetson Orin AGX | **mAP50** | 62.0 | 61.2 | 61.5 | 45.8 |
| | **mAP50-95** | 40.0 | 39.7 | 39.6 | 29.0 |

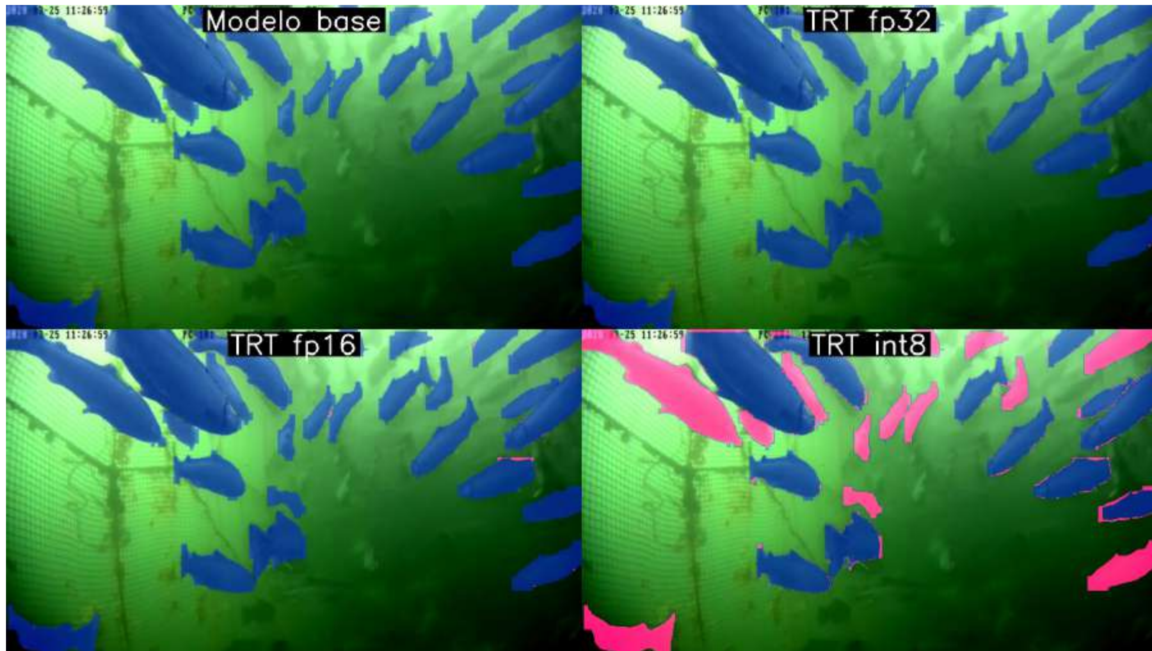



Fig. 5.2: Segmentación en Jetson Orin AGX del ejemplo de entrada *Img2*. Se presentan los resultados de los modelos: modelo base, TRT fp32, TRT fp16 y TRT int8, destacando en rojo las diferencias con respecto al modelo base.

figura, el área azul corresponde a la segmentación del modelo evaluado, y el área roja indica las diferencias frente a la segmentación del modelo base.

A partir de la Figura 5.2 se observa que el modelo `TRT fp32` no presenta diferencias perceptibles con respecto al modelo base. En contraste, el modelo `TRT fp16` exhibe leves discrepancias, destacadas mediante áreas rojas que indican errores de segmentación. Por último, el modelo `TRT int8` muestra una degradación significativa en la calidad de la segmentación, evidenciada por un aumento considerable de zonas rojas que reflejan mayores divergencias frente al modelo base.

Por su parte, la Tabla 5.3 presenta los resultados de **segmentation closeness** obtenidos al evaluar ejemplos representativos de entrada, los cuales se muestran en la Figura 5.1. Este análisis permite cuantificar numéricamente las discrepancias entre las salidas de los modelos evaluados y el modelo base. Se observa que el modelo `TRT fp32` alcanza más del 98 % de segmentation closeness en todos los ejemplos evaluados, mientras que `TRT fp16` supera el 95 %. En contraste, `TRT int8` muestra resultados mixtos: en el mejor caso, se aproxima al 50 %, mientras que en el peor cae por

Tabla 5.3: Resultados de segmentation closeness en porcentaje ( %).

| **Hardware** | **Sample** | **TRT fp32** | **TRT fp16** | **TRT int8** |
|---|---|---|---|---|
| RTX 3060 | ***Img1*** | 99.9 | 97.3 | 51.8 |
| | ***Img2*** | 99.3 | 96.1 | 36.7 |
| | ***Img3*** | 99.0 | 97.7 | 4.4 |
| | ***Img4*** | 99.4 | 94.9 | 9.4 |
| Jetson Orin AGX | ***Img1*** | 99.9 | 99.7 | 50.9 |
| | ***Img2*** | 99.2 | 96.3 | 33.7 |
| | ***Img3*** | 99.1 | 97.3 | 0.3 |
| | ***Img4*** | 99.0 | 95.1 | 6.1 |

debajo del 1 %. Es importante tener en cuenta que un segmentation closeness del 1 % implica que la segmentación producida por el modelo evaluado presenta una discrepancia de 99 puntos porcentuales con respecto a la segmentación obtenida por el modelo base.

Las tendencias observadas en el segmentation closeness son, en general, consistentes con los resultados de mAP. En particular, al comparar los mAP50 y mAP50-95 de los modelos optimizados `TRT fp32` y `TRT fp16` con los del modelo base, se observa que la diferencia no supera un punto porcentual en ningún caso. Esto coincide con los resultados de segmentation closeness, que indican que la salida de las imágenes evaluadas varía en menos de un punto porcentual para `TRT fp32` respecto al modelo base y en no más de cinco puntos porcentuales para `TRT fp16`. Por otro lado, la diferencia de mAP entre `TRT int8` y el modelo base alcanza hasta 14 puntos porcentuales, mientras que los valores de **segmentation closeness** para `TRT int8` revela diferencias de hasta 99 puntos porcentuales, evidenciando una mayor discrepancia con el modelo base.

Este análisis resalta la utilidad de la métrica **segmentation closeness** como complemento a las métricas tradicionales de **mAP**, ya que permite evaluar el impacto de la cuantización no solo desde una perspectiva numérica, sino también mediante una interpretación visual más intuitiva del desempeño del modelo.

A partir de los resultados obtenidos, se opta por emplear la cuantización a `fp16` en las siguientes evaluaciones. Esta decisión se fundamenta en el equilibrio entre la fidelidad de los resultados y las propiedades del modelo. Si bien `TRT fp16` presenta ligeras discrepancias en la segmentación, estas son despreciables en comparación con la degradación observada en `TRT int8`. Además, como se evidenció en la subsección anterior, la cuantización a `fp16` reduce el tamaño del modelo a menos de la mitad en comparación con `TRT fp32`, lo que sugiere una mejora en la gestión de la memoria sin comprometer significativamente la accuracy.

#### Caracterización de la configuración del batch size

En esta evaluación se busca determinar el batch size de inferencia óptimo para maximizar el throughput al construir un inference engine con cuantización `fp16` y batch size dinámico. El análisis considera un rango de batch sizes: mínimo de 1, óptimo de 8 y máximo de 16. Para ello, se evaluará la latencia en inferencia con batch size de inferencia uno y el throughput para los batch sizes restantes dentro del rango definido. Además, se medirá la accuracy y el uso de memoria para el batch size de inferencia encontrado con el objetivo de asegurar un correcto funcionamiento en el modelo seleccionado.

La Tabla 5.4 presenta los resultados de latencia y throughput para distintos batch sizes de infe-

Tabla 5.4: Resultados de latencia para batch size de inferencia igual a uno y de throughput para batch sizes de inferencia desde 2 hasta 16.

| **Hardware** | **Modelo** | **Latencia [ms]** | **Throughput [inf/s]** | | | |
|---|---|---|---|---|---|---|
| | | | 2 | 4 | 8 | 16 |
| RTX 3060 | Modelo Base | 34.5 | 31.7 | 34.1 | 36.0 | 27.6 |
| | TRT fp32 | 30.4 | 38.3 | 41.4 | 42.7 | - |
| | TRT fp16 | 15.7 | 85.9 | 101.6 | 113.4 | - |
| | TRT int8 | 9.7 | 131.3 | 151.9 | 167.4 | - |
| Jetson Orin AGX | Modelo Base | 125.7 | 8.3 | 9.3 | 9.9 | 7.7 |
| | TRT fp32 | 93.5 | 11.7 | 12.5 | 13.0 | 5.0 |
| | TRT fp16 | 52.3 | 20.8 | 22.9 | 24.0 | 7.3 |
| | TRT int8 | 36.2 | 33.4 | 35.6 | 37.2 | 21.5 |

rencia. Como se observó en capítulos anteriores, la latencia se reduce significativamente al comparar el modelo base con la versión optimizada mediante TensorRT. Por otro lado, el throughput aumenta con el tamaño del batch, alcanzando su valor máximo con un batch size de inferencia de ocho, antes de disminuir al seguir incrementando el tamaño del batch. En ambas plataformas evaluadas, el mayor throughput se obtuvo con un **batch size de inferencia de ocho**. Este resultado es especialmente relevante al emplear el modelo optimizado con TensorRT y cuantización `fp16`, logrando 24 y 113.4 inferencias por segundo en las plataformas Jetson Orin AGX y RTX 3060, respectivamente.

Por otro lado, la accuracy de estos modelos permanece constante al evaluarlos con un batch size de inferencia de ocho, tal como se ha comprobado en la Sección 3.2.1 y como se muestra en la Tabla 5.2.

La Figura 5.3 ilustra el uso de memoria al evaluar diferentes batch size de inferencia en la plataforma Jetson Orin AGX durante el proceso de inferencia en el dataset de validación. Se observa un incremento de alrededor de 200 MB en el uso de memoria promedio y de 400 MB en el uso de memoria máxima al utilizar un **batch size de inferencia de ocho**. Es importante destacar que, si bien se registra un aumento en el consumo de memoria al emplear modelos optimizados mediante TensorRT, las diferencias entre los distintos batch size de inferencia evaluados son despreciables, y la diferencia respecto al modelo base se reduce en las inferencias con batch size ocho. Esto sugiere que se puede optar por el batch size que ofrezca el mejor throughput, sin que el uso de memoria represente una limitante significativa.

A partir del análisis realizado, se selecciona un **batch size de inferencia de ocho** para las siguientes evaluaciones, debido a que proporciona el throughput más alto en ambas plataformas evaluadas. Si bien se observa un incremento en el uso de memoria con respecto al modelo base, este aumento se considera despreciable para el entorno de operación.

### Caracterización de la configuración del nivel de optimización

En esta evaluación se analiza el impacto de utilizar distintos **niveles de optimización** en la construcción de un inference engine, tanto en la ejecución del modelo como en el tiempo de construcción del mismo. En concordancia con las evaluaciones previas, el modelo optimizado emplea cuantización `fp16`, un batch size de configuración dinámico y un batch size de inferencia de ocho.

La Tabla 5.5 muestra los resultados obtenidos al utilizar los niveles de optimización 0, 3 y 5 para la generación del engine. Las métricas evaluadas incluyen el tiempo de construcción del inference

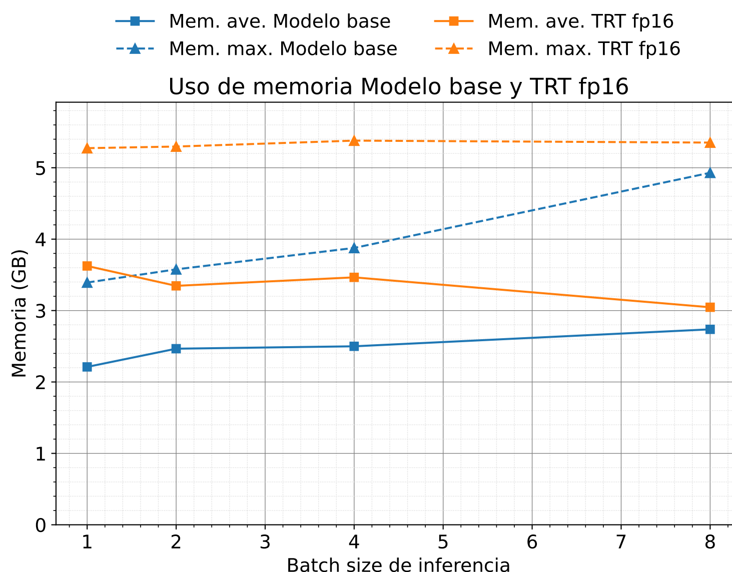



Fig. 5.3: Uso de memoria promedio (ave.) y máxima (max.) en plataforma Jetson Orin AGX durante inferencia en el dataset de validación para distintos batch size de inferencia.

engine (**Tconst**), tamaño del modelo resultante (**Tamaño**), accuracy, memoria promedio (**ave.**), memoria máxima (**max.**) y throughput. Los resultados muestran que un aumento en el nivel de optimización permite **incrementar el throughput** sin afectar significativamente la fidelidad de los resultados del modelo, dado que la métrica mAP permanece prácticamente constante en comparación con el modelo base. Se observa también que el tamaño del modelo y el uso de memoria no varían significativamente con el nivel de optimización. Además, se observa que al aumentar el nivel de optimización aumenta significativamente el tiempo de generación del engine.

Es importante señalar que, según las recomendaciones de Nvidia [28], el engine debe generarse en la misma plataforma en la que se llevarán a cabo las inferencias, como se puede extender del funcionamiento de TensorRT descrito en la Sección 2.3.1. Debido a esto, es esperable que el tiempo de construcción sea significativamente mayor en la plataforma embebida que en la de escritorio. En particular, en una Jetson Orin AGX, el tiempo de generación puede extenderse hasta varias horas. De hecho, en experimentos no reportados se observaron tiempos superiores a 18 horas. Sin embargo, este proceso se realiza únicamente durante la etapa de diseño. En la práctica, este costo computacional puede considerarse aceptable si permite maximizar el throughput durante la inferencia.

Con base en los resultados obtenidos, se selecciona el `nivel 5` para la evaluación siguiente debido a que proporciona el mayor **throughput** sin impactar la **accuracy** del modelo.

### Caracterización de la configuración del modo de consumo

Finalmente, se busca analizar los trade-offs al emplear distintos **modos de consumo** en la construcción de un inference engine con cuantización `fp16`, un batch size de configuración dinámico, un batch size de inferencia de ocho y un nivel de optimización de 5. El análisis se centrará en los modos de consumo PM2 y PM0 para la plataforma Jetson Orin AGX. Para ello, se evaluarán métricas de tiempo de construcción del inference engine, el tamaño del modelo, la accuracy, el uso

Tabla 5.5: Resultados de las métricas evaluadas para el modelo base y TRT fp16 construido con diferentes niveles de optimización (**oplvl**), considerando un batch size de inferencia igual a ocho.

| | **Modelo** | **oplvl** | **Tconst** [min] | **Tamaño** [MB] | **Accuracy [%]** mAP50 | mAP50-95 | **Memoria [GB]** ave. | max. | **Throughput** [inf/s] |
|---|---|---|---|---|---|---|---|---|---|
| RTX3060 | Modelo Base | - | - | 88.0 | 62.0 | 40.0 | - | - | 36.0 |
| | TRT fp16 | 0 | 1 | 91.8 | 61.5 | 39.7 | - | - | 95.4 |
| | TRT fp16 | 3 | 8 | 90.8 | 61.4 | 39.6 | - | - | 113.4 |
| | TRT fp16 | 5 | 11 | 91.0 | 61.5 | 39.7 | - | - | 114.3 |
| †AGX | Modelo Base | - | - | 88.0 | 62.0 | 40.0 | 2.74 | 4.93 | 9.9 |
| | TRT fp16 | 0 | 7 | 91.4 | 61.5 | 39.7 | 3.59 | 5.41 | 20.0 |
| | TRT fp16 | 3 | 150 | 92.0 | 61.5 | 39.6 | 3.05 | 5.35 | 24.0 |
| | TRT fp16 | 5 | 360 | 91.5 | 61.5 | 39.7 | 3.46 | 5.36 | 24.8 |

†**AGX:** Jetson Orin AGX.

Tabla 5.6: Resultados de las métricas evaluadas para el modelo base y TRT fp16 construido con diferentes modos de consumo (PM) en la plataforma Jetson Orin AGX, considerando un nivel de optimización de cinco y un batch size de inferencia igual a ocho.

| **Modelo** | **PM** | **Tconst** [min] | **Tamaño** [MB] | **Accuracy [%]** mAP50 | mAP50-95 | **Memoria [GB]** Promedio | Máxima | **Throughput** [inf/s] |
|---|---|---|---|---|---|---|---|---|
| Modelo Base | PM2 | - | 88.0 | 62.0 | 40.0 | 2.74 | 4.93 | 9.9 |
| Modelo Base | PM0 | - | 88.0 | 62.0 | 40.0 | 2.37 | 4.47 | 25.7 |
| TRT fp16 | PM2 | 360 | 91.5 | 61.5 | 39.7 | 3.46 | 5.36 | 24.8 |
| TRT fp16 | PM0 | 299 | 91.0 | 61.4 | 39.7 | 2.95 | 5.49 | 49.9 |

de memoria durante la inferencia y el throughput.

La Tabla 5.6 presenta los resultados de las métricas de tiempo de construcción (**Tconst**), tamaño del modelo (**Tamaño**), accuracy, memoria y throughput. Los resultados muestran que optar por un modo de consumo más permisivo, en términos de energía, frecuencia de operación y núcleos de CPU activos, permite **incrementar el throughput** sin afectar la fidelidad de los resultados del modelo, ya que la métrica mAP se mantiene prácticamente constante en comparación con el modelo base. Este aumento en el throughput no solo se observa en el modelo `TRT fp16`, sino que también permite duplicar el rendimiento del modelo base.

En cuanto al tamaño del modelo y al uso de memoria, estas métricas se mantienen en niveles similares al comparar los resultados de `PM0` con respecto al modo de consumo por defecto (`PM2`). Además, el tiempo de construcción del modelo se reduce aproximadamente un 17 % en comparación con el inference engine generado bajo las mismas configuraciones utilizando el modo de consumo por defecto.

Dadas las condiciones específicas de la aplicación, en las que la plataforma no está sujeta a restricciones de bajo consumo energético, es posible tolerar un aumento en el consumo de energía a cambio de una mejora en el throughput. Esto se debe a que la plataforma se encuentra alimentada por el cable de alimentación integrado en el ROV, eliminando la necesidad de optimizar el consumo energético. Por otro lado, si la aplicación requiriera un consumo energético restringido, sería recomendable optar por el modo de consumo `PM2`, ya que este permite optimizar la duración de la batería a costa de una reducción en el rendimiento de las métricas evaluadas.

Con base en los resultados obtenidos y en las especificaciones de la aplicación, se concluye que el modo de consumo `PM0` es la opción más adecuada para la construcción del modelo final en este caso particular. Este modo de operación permite duplicar el throughput en comparación con el

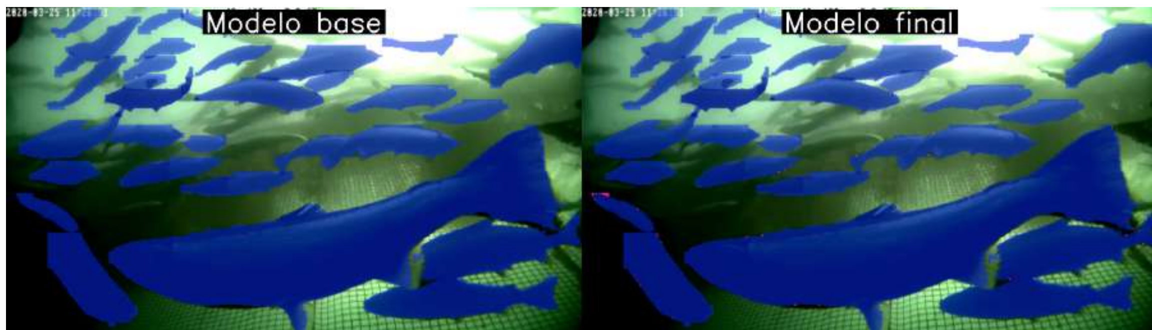



Fig. 5.4: Segmentación en Jetson Orin AGX del ejemplo de entrada *Img3*. Se presentan los resultados de los modelos: modelo base y modelo final, destacando en rojo las diferencias con respecto al modelo base.

modelo base, sin afectar negativamente la accuracy. Además, se logra una reducción en el tiempo de construcción del inference engine en comparación con el modo de consumo por defecto. Estas mejoras se obtienen sin un incremento significativo en el uso de memoria ni en el tamaño del modelo.

Finalmente, la construcción del modelo final se realiza con la siguiente configuración de operación: plataforma Jetson Orin AGX, modelo `yolov8l-seg`, cuantización `fp16`, batch size de configuración dinámico, batch size de inferencia igual a ocho, nivel de optimización 5 y modo de consumo `PM0`.

La Figura 5.4 muestra un ejemplo de segmentación realizado por el modelo final en comparación con el modelo base. Además, en el directorio `outputs` del repositorio online [52], se encuentra un video que ilustra el comportamiento del modelo final en un entorno real.

### 5.1.4. Discusión de resultados

Los resultados analizados en las secciones previas representan un estudio sobre la optimización de la inferencia mediante TensorRT en un entorno distinto al de los benchmarks reportados anteriormente. Este análisis aborda las limitaciones y vacíos identificados en la literatura, aspectos que fueron discutidos en la Sección 4.2.5 del capítulo anterior.

Al evaluar la métrica de **segmentation closeness** para identificar la cuantización adecuada, se observó que las optimizaciones de TensorRT, que cuantizan los parámetros del modelo a int8, provocan una degradación significativa en la salida del mismo. Como se sugirió en la Sección 4.2.5, la cuantización a int8 afecta negativamente la tarea de segmentación de objetos en la aplicación específica evaluada en este capítulo. Por lo tanto, se recomienda el uso de las optimizaciones de TensorRT con cuantización a fp16 en aplicaciones que requieran la tarea de segmentación de objetos.

En cuanto a la elección del **batch size de inferencia**, se evidenció que esta configuración no influye en los resultados de accuracy, pero tiene un papel relevante en el throughput resultante. Se demostró que es vital evaluar un rango completo de batch size de inferencia para encontrar aquel que permita **maximizar el throughput** en la aplicación específica; en este caso particular, el tamaño óptimo fue de ocho. Sin embargo, esta elección depende del hardware objetivo, el modelo de red, la tarea y las configuraciones específicas de cada aplicación.

En relación con el **nivel de optimización**, el objetivo principal de este experimento fue maximizar el throughput. Finalmente, se optó por el nivel cinco, ya que permite un aumento moderado en el throughput en comparación con el nivel predeterminado, aunque a costa de un incremento

significativo en el tiempo de construcción del inference engine. A pesar de esta elección, el uso de un inference engine con nivel de optimización cero sigue siendo relevante para pruebas de concepto, ya que facilita la evaluación del flujo de trabajo y la configuración de la TensorRT Application en un tiempo reducido. Para la generación del engine final, es recomendable utilizar un nivel de optimización mayor una vez que el resto de las configuraciones hayan sido validadas con éxito.

Por último, en cuanto al **modo de consumo**, dado que la aplicación específica se definió sin restricciones de consumo energético, fue posible seleccionar un modo de consumo más permisivo en términos de frecuencia de operación, límite energético y cantidad de CPUs activas. Como se discutió en la Sección 4.2.5, esta elección permitió aumentar significativamente el throughput sin afectar de manera considerable el resto de las métricas evaluadas. Además, en este capítulo se observó que un modo de consumo menos restrictivo también contribuye a reducir ligeramente el tiempo de construcción del inference engine. Por lo tanto, en escenarios como el de este experimento, donde el objetivo principal es maximizar el throughput sin restricciones de consumo energético, la opción más adecuada es utilizar el modo de consumo con menos limitaciones (`PM0` en la Jetson Orin AGX).

En la sección siguiente, se llevará a cabo la evaluación de una aplicación que emplea la tarea de regresión.

## 5.2. Pirometría de hollín

La estimación de la temperatura del hollín en llamas laminares es fundamental para comprender la formación de material particulado en sistemas de combustión y su impacto en la eficiencia energética [60]. La pirometría de hollín permite estimar la temperatura de las partículas dentro de la llama a partir de la radiación que emiten, utilizando técnicas ópticas no invasivas. Entre los métodos tradicionales se encuentran los enfoques basados en mediciones de absorción/emisión modulada (*Modulated Absorption Emission*, MAE) y de emisión de banda ancha (*Broadband Emission*, BEMI). Este último ha ganado interés debido a su capacidad de implementación con cámaras RGB de bajo costo [61].

Tanto MAE como BEMI presentan limitaciones que afectan la confiabilidad de las estimaciones de temperatura del hollín. En el caso de MAE, la dependencia de mediciones de absorción y emisión requiere una alineación precisa de los componentes ópticos y una sincronización meticulosa en la captura de datos, lo que introduce incertidumbre y dificulta la repetibilidad de los resultados incluso bajo condiciones experimentales controladas. Además, la correcta aplicación de esta técnica demanda un nivel avanzado de conocimiento en óptica, lo que la hace poco viable para implementaciones prácticas fuera de entornos especializados. En contraste, BEMI simplifica considerablemente la instrumentación al basarse únicamente en señales de emisión capturadas con cámaras RGB, lo que facilita su implementación práctica. No obstante, la falta de mediciones de absorción implica que la técnica asume la emisión integrada a lo largo de la línea de visión sin corregir la autoabsorción dentro de la llama, lo que introduce desviaciones en la estimación de temperatura de hasta 50 K.

La Figura 5.5 muestra un esquema del arreglo experimental típico utilizado en la técnica BEMI. En este montaje, la llama se modela en coordenadas cilíndricas $(r, z)$ y emite radiación en múltiples longitudes de onda. Esta radiación es capturada por una cámara RGB ubicada en el extremo opuesto del camino óptico. Las señales proyectadas en el plano de la cámara, $P_{\{R,G,B\}}$, se describen en coordenadas cartesianas $(y, z)$ y posteriormente se someten a un proceso de deconvolución mediante la resolución de un problema inverso mal condicionado. Finalmente, a partir de estas señales y mediante modelos matemáticos, se obtiene el campo 2D de temperatura del hollín $(T_s)$.

Los avances recientes en visión por computador han impulsado el uso de ANNs para la estimación

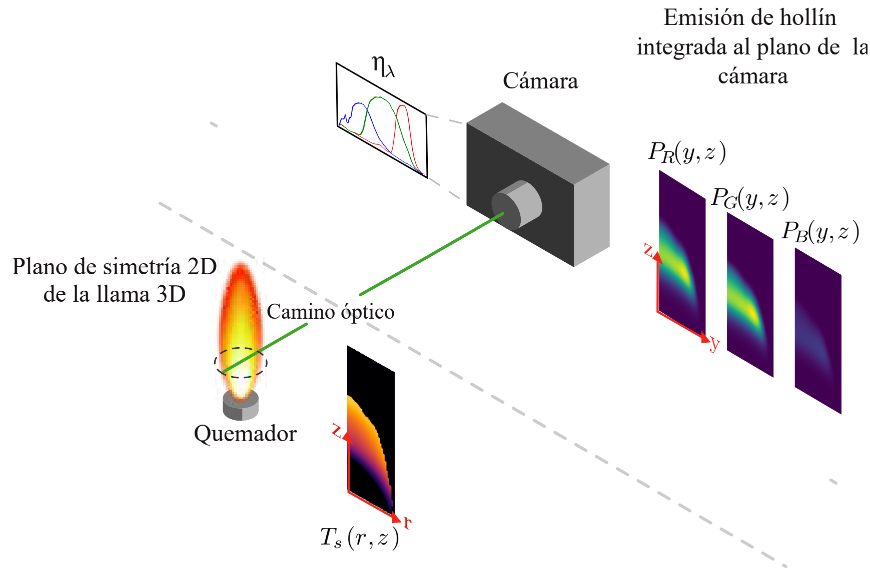



Fig. 5.5: Esquema del arreglo óptico BEMI para capturar la radiación emitida por las partículas de hollín. Adaptado de [62].

de la temperatura del hollín, superando las limitaciones inherentes a los enfoques clásicos [63, 64]. Estudios previos han demostrado que modelos basados en ANNs, como `U-Net` [65] y `Attention U-Net` [66], son capaces de inferir relaciones no lineales entre las señales de emisión y la temperatura real del hollín. Estos modelos han mostrado una notable capacidad de generalización, permitiendo estimaciones precisas bajo distintas condiciones de combustión. Además, las ANNs pueden capturar efectos complejos, como la autoabsorción del hollín, lo que contribuye a reducir los errores característicos de los métodos basados en BEMI.

En este contexto, la Universidad Técnica Federico Santa María ha desarrollado metodologías para la pirometría de hollín mediante visión por computador y redes neuronales [62,67,68]. En particular, una colaboración entre el Departamento de Electrónica y el *Energy Conversion and Combustion Group* (EC2G) ha explorado la integración de modelos como `U-Net` y `Attention U-Net` para la estimación de temperatura a partir de imágenes de llamas capturadas con cámaras RGB.

## 5.2.1. Entorno de operación

Aquí se describe el entorno de operación utilizado para las evaluaciones de esta sección, el cual incluye la aplicación, el flujo de trabajo, el hardware y las métricas empleadas. Los elementos que conforman este entorno se detallan en las subsecciones siguientes.

#### Contexto de aplicación

En esta evaluación experimental, se implementan dos modelos basados en ANNs para la estimación de la temperatura del hollín en llamas laminares, siguiendo los procedimientos descritos en estudios previos sobre pirometría de hollín [62,67,68]. Dado que el diseño, entrenamiento y validación de estos modelos han sido abordados en detalle en dichas referencias, en este trabajo se presenta únicamente una descripción general del entorno de operación y ejemplos de inferencia relevantes

para el análisis desarrollado.

En el desarrollo de esta evaluación se emplean los modelos basados en `U-Net` y `Attention U-Net`, especializados en la estimación de temperatura a partir de imágenes RGB de llamas laminares, abordando el problema como una tarea de regresión. Estos modelos han sido entrenados con un conjunto de datos generado mediante simulaciones en el software CoFlame [69], basado en llamas de difusión laminares axi-simétricas canónicas (CLAD) generadas con el quemador Yale [70], caracterizadas por su estabilidad y reproducibilidad, lo que las hace ideales para el estudio de temperatura y formación de hollín.

En este trabajo se consideran **tres condiciones de llama**, definidas por distintas configuraciones de flujo de combustible y oxígeno, denominadas Yale-32, Yale-40 y Yale-60. Estas variaciones influyen directamente en la distribución espacial tanto de la temperatura como de la concentración de hollín. La Figura 5.6 presenta ejemplos de pares de temperatura del hollín ($T_s$) y sus correspondientes proyecciones en el plano de la cámara ($P_{\{R,G,B\}}$), generados mediante simulaciones numéricas. A lo largo de este trabajo, $P_{\{R,G,B\}}$ se considera la entrada del modelo y $T_s$, su salida.

Para la validación de los modelos se utiliza un conjunto de datos experimental compuesto por imágenes obtenidas en entornos controlados. Este dataset se divide en dos subconjuntos:

- El primero se emplea para validar la precisión de la regresión. Está compuesto por una imagen promediada por cada condición de llama, obtenida a partir del promedio de 100 frames correspondientes a una misma llama, con el objetivo de mejorar la relación señal/ruido. Cada una de estas imágenes cuenta con su par asociado ($P_{\{R,G,B\}}, T_s$), el cual se utiliza como *ground truth*.

- El segundo subconjunto se utiliza para la evaluación de métricas de rendimiento e incluye 300 imágenes individuales correspondientes a una misma condición de llama, específicamente Yale-32. Cada imagen está asociada únicamente a su valor de entrada $P_{\{R,G,B\}}$.

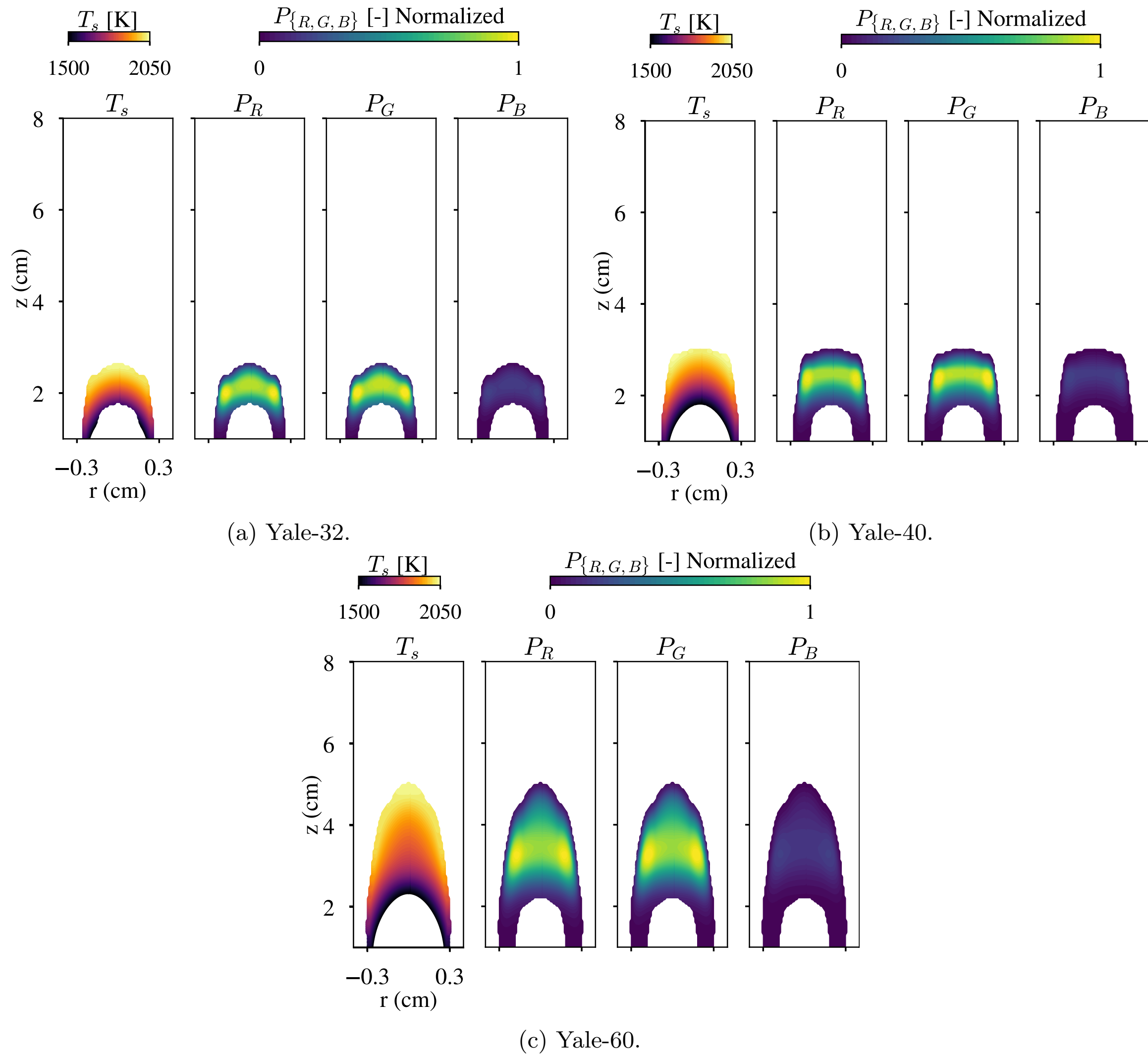
El acceso a estos datos está disponible a través de un enlace en el repositorio en línea [53]. Para más detalles sobre la aplicación específica y el entorno de operación, se recomienda consultar [62, 67, 68].

El contexto de aplicación considerado para la evaluación de desempeño asume que el modelo de inferencia debe estar optimizado para su ejecución en hardware embebido, específicamente en una plataforma Jetson equipada con una cámara RGB estándar. Se requiere que el proceso de estimación de temperatura se realice en tiempo real, y que los resultados obtenidos sean transmitidos a un computador externo para su posterior análisis. Bajo estas condiciones, es fundamental alcanzar un equilibrio entre la precisión de la estimación de temperatura y las métricas de desempeño computacional, considerando el uso de plataformas embebidas con un enfoque en bajo costo y accesibilidad.

#### Workflow

El flujo de trabajo utilizado para obtener los inference engines derivados de los modelos base `U-Net` y `Attention U-Net` sigue el esquema PyTorch-ONNX-TensorRT descrito en el Capítulo 3.

Para crear los inference engines en cada configuración de operación, es necesario generar una TensorRT Application, siguiendo el procedimiento descrito anteriormente y complementado con lo expuesto en la Sección 2.3. Para ello, se define una configuración en el builder, que permite ajustar parámetros como la cuantización, el batch size, el nivel de optimización y las dimensiones de las imágenes de entrada. Para más detalles sobre esta implementación, refiérase al repositorio [53].




(a) Yale-32.

(b) Yale-40.

(c) Yale-60.

Fig. 5.6: Soluciones de referencia para los campos de $T_s$ en una llama CLAD de Yale utilizando CoFlame. Adaptado de [62].

#### Hardware

La Tabla 3.2 presenta las características de las plataformas de hardware seleccionadas: una plataforma embebida Jetson Orin AGX y un computador de escritorio equipado con una tarjeta gráfica RTX 3060. La Jetson Orin AGX es de principal interés para las evaluaciones realizadas, ya que permite ejecutar el modelo directamente en entornos de laboratorio. Por otro lado, el computador de escritorio se emplea como referencia para la comparación de rendimiento.

#### Métricas

A continuación, se describe la metodología empleada para obtener las métricas evaluadas en esta sección. Cabe destacar que las métricas utilizadas se basan en las definidas en capítulos anteriores, pero han sido adaptadas al contexto específico de la aplicación objetivo.

Las métricas relevantes en el contexto de operación incluyen las **propiedades del modelo**, el **uso de memoria durante la inferencia**, el **throughput** y la **latencia**. La latencia es particularmente importante, ya que permite evaluar el tiempo de respuesta del modelo en escenarios de inferencia con un batch size igual a uno. Estas métricas se definen y miden de manera idéntica a lo descrito en la Sección 3.1.4, junto con el **tiempo de construcción del modelo**, definido en la Sección 5.1.1.

Por otro lado, las métricas asociadas al desempeño funcional, como accuracy y closeness, descritas en los capítulos previos, fueron definidas para el contexto de clasificación y segmentación de imágenes. Por lo tanto, deben ser redefinidas para el contexto de problemas de regresión. Para esta evaluación, se introducen las siguientes métricas:

**Regression accuracy** La precisión de la regresión en este trabajo se evalúa según la definición de error global propuesta en [67]. En particular, se compara el campo de temperatura estimado por la red neuronal con el ground truth, definido en este caso como el campo de temperatura del hollín ($T_s$) obtenido a partir de soluciones numéricas generadas con el modelo CoFlame [69]. Para cuantificar la diferencia entre la estimación y la referencia, se emplea el error cuadrático medio (RMSE), calculado como:

$$\text{RMSE} = \sqrt{\frac{1}{m \cdot p} \sum_{i=1}^{m} \sum_{j=1}^{p} (\hat{Y}_{i,j} - Y_{i,j})^2}, \tag{5.2.1}$$

donde $Y$ representa la imagen del campo de temperatura de referencia, $\hat{Y}$ representa la imagen del campo de temperatura estimado, y $m$ y $p$ corresponden al número de filas y columnas de píxeles en las imágenes. La evaluación se realiza a nivel de píxel en el campo bidimensional.

Es importante destacar que, debido a la presencia de outliers inherentes en $T_s$, el cálculo del RMSE se realiza excluyendo aquellas diferencias que superen los $\pm 100$ K entre la estimación y el valor de referencia.

**Regression closeness** Se define de manera análoga a classification closeness, descrito en la Sección 4.1.4, pero aplicado a la salida de las redes utilizadas en este experimento.

### 5.2.2. Metodología de evaluación y configuración

El proceso de evaluación desarrollado en esta sección es equivalente al introducido en la Sección 5.1.2, donde se aplica una optimización incremental del modelo de regresión mediante la modificación secuencial de parámetros en la generación del inference engine. Para ello, se ajusta un parámetro a la vez, manteniendo los demás fijos. El procedimiento comienza con la selección del

Tabla 5.7: Propiedades de los modelos U-Net y Attention (**Att.**) U-Net en sus distintas versiones: modelo base, TRT `fp32`, TRT `fp16` y TRT `int8`.

| **Hardware** | **Modelo** | **Propiedades** | | |
|---|---|---|---|---|
| | | Tamaño [MB] | # Capas | # Parámetros |
| RTX 3060 | U-Net Base | 7.25 | 91 | 1884631 |
| | U-Net TRT fp32 | 10.75 | 23 | 1880040 |
| | U-Net TRT fp16 | 4.45 | 25 | 1880040 |
| | U-Net TRT int8 | 2.84 | 31 | 1880040 |
| | Att. U-Net Base | 33.75 | 114 | 8828070 |
| | Att. U-Net TRT fp32 | 35.13 | 70 | 8823221 |
| | Att. U-Net TRT fp16 | 18.57 | 46 | 8823221 |
| | Att. U-Net TRT int8 | 11.06 | 48 | 8823221 |
| Jetson Orin AGX | U-Net Base | 7.25 | 91 | 1884631 |
| | U-Net TRT fp32 | 7.88 | 24 | 1880040 |
| | U-Net TRT fp16 | 4.66 | 25 | 1880040 |
| | U-Net TRT int8 | 2.64 | 30 | 1880040 |
| | Att. U-Net Base | 33.75 | 114 | 8828070 |
| | Att. U-Net TRT fp32 | 35.33 | 61 | 8823221 |
| | Att. U-Net TRT fp16 | 18.39 | 46 | 8823221 |
| | Att. U-Net TRT int8 | 9.75 | 30 | 8823221 |

nivel de **cuantización** adecuado, asegurando que el desempeño de la red sea comparable al del modelo base en términos de regression accuracy. Una vez identificada la cuantización que satisface los requisitos de precisión, se evalúa la regression closeness para analizar la robustez del modelo ante la cuantización. Posteriormente, se optimiza la configuración del **batch size**, con el objetivo de maximizar el **throughput**. Finalmente, se exploran configuraciones adicionales para analizar los trade-offs entre el **nivel de optimización** aplicado durante la generación del motor de inferencia y el **modo de consumo energético**, disponible exclusivamente en la tarjeta Jetson.

### 5.2.3. Resultados y análisis

A continuación, se presentan los resultados obtenidos al evaluar las diferentes configuraciones. Todos los datos generados durante el experimento están disponibles en el repositorio de GitHub [53].

#### Caracterización de las propiedades del modelo

Como etapa previa a la evaluación del nivel de cuantización, es necesario caracterizar las propiedades de los modelos generados para las pruebas posteriores, en las cuales se compara el modelo base con sus versiones cuantizadas en `fp32`, `fp16` e `int8`.

La Tabla 5.7 presenta las propiedades de los modelos evaluados en ambas plataformas. Estos resultados son consistentes con las observaciones de la Sección 3.2.2. En particular, se demuestra que TensorRT efectivamente aplica optimizaciones en los modelos evaluados en esta sección, lo que se refleja en la reducción del tamaño del modelo en las versiones TRT `fp16` y TRT `int8`, así como en la disminución general del número de capas, manteniendo casi el mismo número de parámetros.

Tabla 5.8: Resultados de **regression accuracy** en Kelvin (**K**) bajo diferentes condiciones de llama.

| | Modelo | Cond. de llama | Modelo base | TRT fp32 | TRT fp16 | TRT int8 |
|---|---|---|---|---|---|---|
| **RTX 3060** | **U-Net** | Yale-32 | 15.3 | 19.4 | 19.5 | 21.3 |
| | | Yale-40 | 40.8 | 39.8 | 39.6 | 36.7 |
| | | Yale-60 | 31.7 | 35.4 | 35.4 | 35.2 |
| | **Att. U-Net** | Yale-32 | 31.0 | 31.0 | 31.0 | 29.9 |
| | | Yale-40 | 38.4 | 38.4 | 38.4 | 56.6 |
| | | Yale-60 | 28.4 | 28.4 | 28.4 | 41.6 |
| **J. Orin AGX** | **U-Net** | Yale-32 | 15.3 | 19.4 | 19.5 | 19.6 |
| | | Yale-40 | 40.8 | 39.8 | 39.6 | 37.9 |
| | | Yale-60 | 31.7 | 35.4 | 35.4 | 33.9 |
| | **Att. U-Net** | Yale-32 | 31.0 | 31.0 | 31.0 | 29.4 |
| | | Yale-40 | 38.4 | 38.4 | 38.4 | 49.5 |
| | | Yale-60 | 28.4 | 28.4 | 28.4 | 43.2 |

### Caracterización de la configuración de la cuantización

El objetivo de esta evaluación es determinar un nivel de cuantización adecuado que permita mantener el desempeño de la red en un nivel comparable al modelo base. Para ello, se consideran las métricas regression accuracy y regression closeness, evaluando la cuantización en `fp32`, `fp16` e `int8`. En todas las pruebas, se utiliza un batch size de inferencia igual a uno, dado que esta configuración garantiza un desempeño consistente, como se evidenció en los resultados de la Sección 3.2.1.

La Tabla 5.8 presenta los resultados de regression accuracy, siguiendo la metodología descrita en la Sección 5.2.1, sobre el conjunto de datos experimentales definido en la Sección 5.2.1. Los resultados indican que la cuantización a `fp32` y `fp16` no introduce cambios significativos en la precisión respecto al modelo base, con variaciones inferiores a 3 K en todas las configuraciones evaluadas. Sin embargo, al cuantizar los parámetros a `int8`, se observa un incremento del error de aproximadamente 10 K en ambas plataformas evaluadas, lo que implica una mayor degradación en la precisión en comparación con el modelo base.

Los resultados presentados en la Tabla 5.8, particularmente para la configuración `TRT int8`, tienden a estar subestimados, ya que la métrica de regression accuracy ignora los outliers que superan los $\pm 100$ K. Esto hace que los resultados obtenidos para `int8` parezcan más favorables de lo que realmente son, dado que varios píxeles en la salida superan este umbral y, al ser descartados, terminan afectando la evaluación de la métrica.

Por otro lado, las Figuras 5.7, 5.8 y 5.9 muestran los resultados del modelo base, así como la diferencia de temperatura entre el modelo optimizado y el modelo base en la plataforma Jetson Orin AGX. Cabe destacar que los resultados en la plataforma RTX 3060 son equivalentes.

Para esta aplicación específica, se considera un rango visual de error entre $-100$ K y 100 K con respecto a la predicción de $T_s$ del modelo base, el cual se representa mediante la barra de $\Delta T$ en las Figuras 5.7, 5.8 y 5.9. Si el error excede este margen, su valor se satura en los límites de dicha barra. En todas las condiciones de llama evaluadas, los modelos optimizados de la red `U-Net` en formatos `fp32` y `fp16` presentan, en su mayoría, errores dentro de un rango aceptable inferior a $\pm 30$ K. En contraste, la cuantización a `int8` introduce una mayor degradación en la predicción de $T_s$, superando los $\pm 30$ K en varios píxeles. Por otro lado, el modelo `Attention U-Net` demuestra ser **altamente robusto** en sus versiones `fp32` y `fp16`, con errores prácticamente nulos en relación con

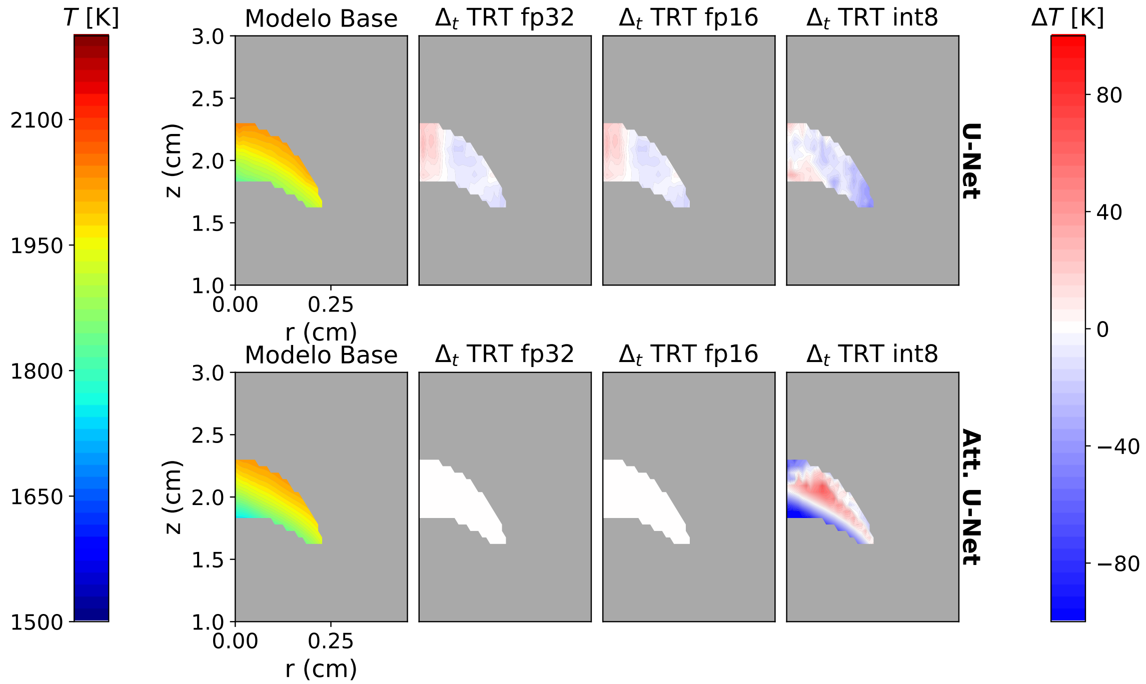



Fig. 5.7: Comparación de la predicción de $T_s$ entre el modelo base y los modelos optimizados bajo la condición de llama Yale-32. Se incluye el mapa de error $\Delta T$ respecto al modelo base.

el modelo base. No obstante, su versión cuantizada a `int8` presenta errores que exceden los $\pm 100$ K en múltiples regiones de las imágenes evaluadas.

La Tabla 5.9 presenta los resultados de regression closeness, obtenidos a partir de 300 imágenes del conjunto de datos experimental descrito en la Sección 5.2.1, considerando todas las configuraciones de modelos y plataformas de hardware evaluadas. En dicha tabla, las columnas etiquetadas como `atol` indican el porcentaje de coincidencia entre el modelo optimizado y el modelo base, para distintos valores de tolerancia absoluta. Estos porcentajes se calculan siguiendo la metodología descrita en la Sección 4.1.4, de forma análoga a lo expuesto en la Sección 4.2.1.

Los resultados de la Tabla 5.9 se mantienen consistentes entre las distintas plataformas, pero **varían según el modelo evaluado**. Para las configuraciones `fp32` y `fp16`, el modelo `Attention U-Net` alcanza un closeness del 100 % con una tolerancia del 0.5 %. En contraste, la red `U-Net`, bajo la misma condición de `atol`, solo alcanza un closeness del 2 %, lo que evidencia una mayor robustez de `Attention U-Net` frente a la optimización con cuantización `fp32` y `fp16`, como también se observa en las Figuras 5.7, 5.8 y 5.9. Por otro lado, se destaca que la versión cuantizada a `int8` de `Attention U-Net` presenta los mayores errores con respecto al modelo base, lo que se refleja en valores de closeness generalmente inferiores en comparación con el resto de las configuraciones.

En función de los resultados obtenidos, se concluye que la cuantización a `fp16` mantiene un desempeño comparable al del modelo base, sin introducir degradaciones en la predicción de $T_s$, en especial para el modelo `Attention U-Net`. A pesar de que las diferencias en la predicciones de $T_s$ respecto al modelo base son casi nulas, la optimización mediante TensorRT se hace evidente al observar la modificación en la estructura del modelo, como se reporta en la Tabla 5.7. Por lo tanto, en las siguientes secciones se adoptará la configuración `fp16` para evaluar el desempeño en términos

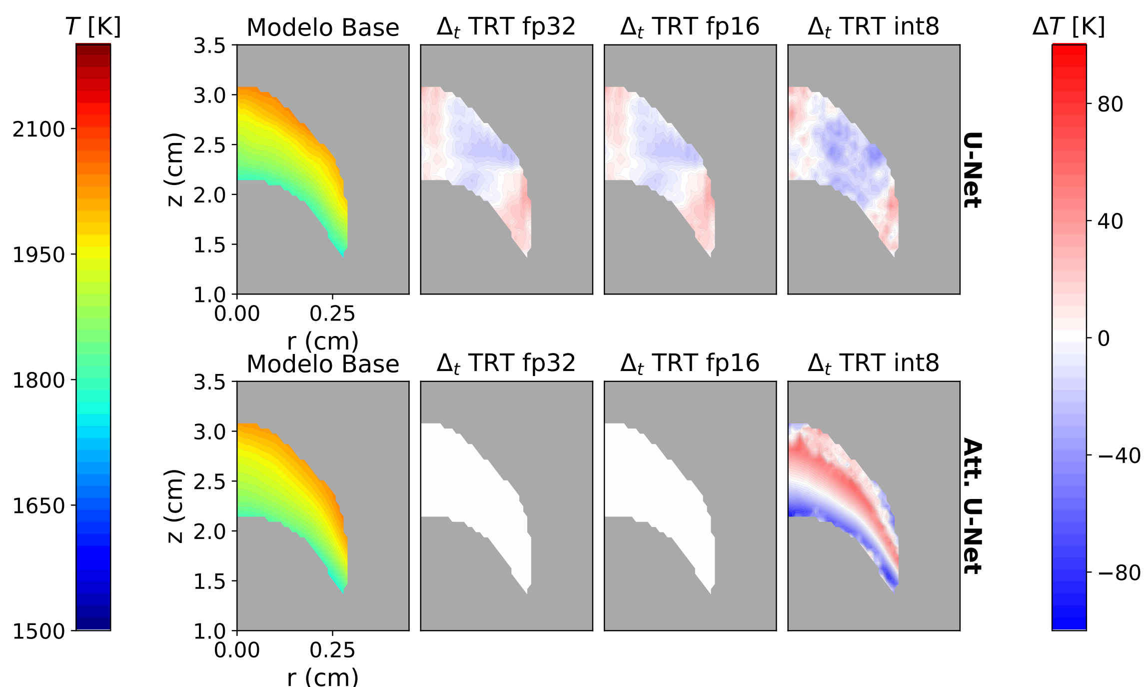



Fig. 5.8: Comparación de la predicción de $T_s$ entre el modelo base y los modelos optimizados bajo la condición de llama Yale-40. Se incluye el mapa de error $\Delta T$ respecto al modelo base.

Tabla 5.9: Resultados de closeness (%) según diferentes razones de tolerancia absoluta para los modelos U-Net y Attention U-Net.

| | | | atol 0.005 | atol 0.01 | atol 0.1 | atol 0.2 | atol 0.5 | atol 1 |
|---|---|---|---|---|---|---|---|---|
| RTX 3060 | U-Net | fp32 | 3.10 % | 6.51 % | 61.28 % | 85.52 % | 97.94 % | 99.99 % |
| | | fp16 | 3.10 % | 6.43 % | 61.36 % | 85.51 % | 97.94 % | 99.99 % |
| | | int8 | 3.46 % | 6.92 % | 62.53 % | 85.27 % | 97.08 % | 99.98 % |
| | Att. U-Net | fp32 | 100.00 % | 100.00 % | 100.00 % | 100.00 % | 100.00 % | 100.00 % |
| | | fp16 | 99.97 % | 100.00 % | 100.00 % | 100.00 % | 100.00 % | 100.00 % |
| | | int8 | 4.53 % | 8.94 % | 73.67 % | 93.21 % | 99.87 % | 100.00 % |
| J.O.AGX | U-Net | fp32 | 3.14 % | 6.47 % | 61.81 % | 85.63 % | 98.00 % | 99.99 % |
| | | fp16 | 3.16 % | 6.47 % | 61.82 % | 85.62 % | 98.00 % | 99.99 % |
| | | int8 | 3.11 % | 5.65 % | 51.87 % | 82.60 % | 98.62 % | 99.97 % |
| | Att. U-Net | fp32 | 100.00 % | 100.00 % | 100.00 % | 100.00 % | 100.00 % | 100.00 % |
| | | fp16 | 99.97 % | 100.00 % | 100.00 % | 100.00 % | 100.00 % | 100.00 % |
| | | int8 | 2.55 % | 4.99 % | 37.79 % | 59.58 % | 91.32 % | 99.96 % |

de latencia, throughput, uso de memoria durante la inferencia y tiempo de construcción del modelo.

### Caracterización de distintas configuraciones

En esta evaluación se busca determinar las configuraciones que maximicen el throughput al construir un inference engine con cuantización `fp16` y batch size dinámico. Para resumir la información presentada, los resultados se mostrarán únicamente para la plataforma Jetson Orin AGX, dado que

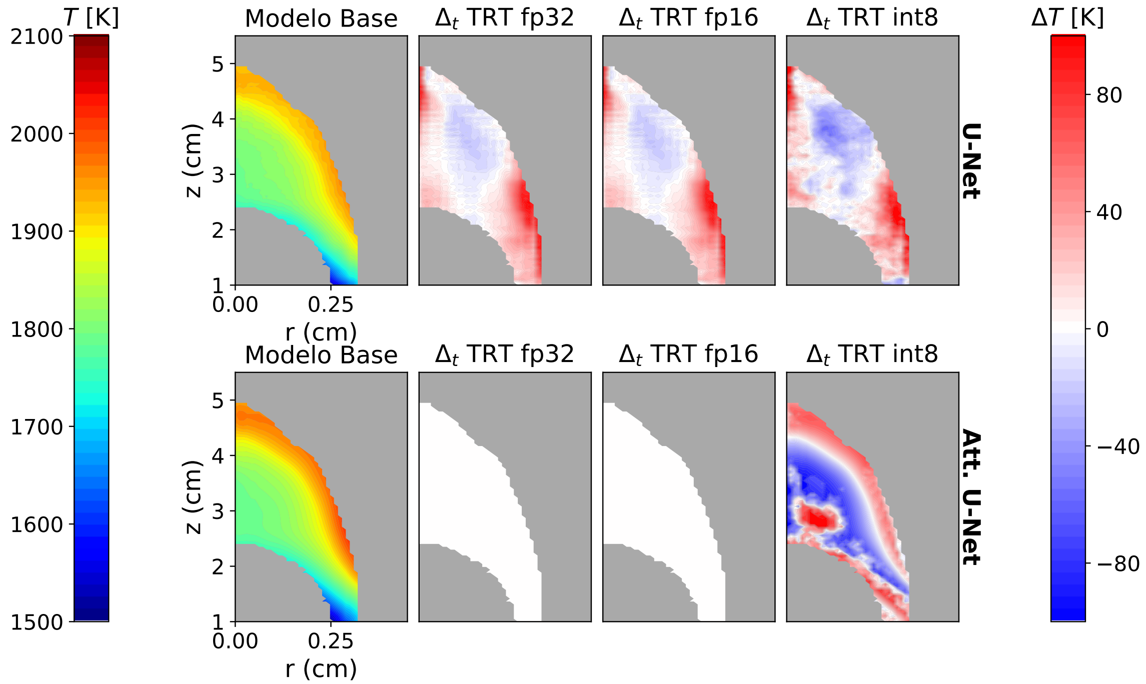



Fig. 5.9: Comparación de la predicción de $T_s$ entre el modelo base y los modelos optimizados bajo la condición de llama Yale-60. Se incluye el mapa de error $\Delta T$ respecto al modelo base.

Tabla 5.10: Resultados de latencia para un batch size de inferencia de 1 y de throughput para batch sizes de inferencia entre 2 y 32 en la plataforma Jetson Orin AGX.

| | **Modelo** | **Latencia [ms]** | | **Throughput [inf/s]** | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **ave.** | **max.** | 2 | 4 | 8 | 16 | 32 |
| U-Net | Modelo Base | 126.5 | 714.5 | 27.17 | 31.29 | 81.33 | 117.49 | 208.05 |
| | TRT fp16 | 112.2 | 139.7 | 35.18 | 52.53 | 125.72 | 135.41 | 268.36 |
| Att. U-Net | Modelo Base | 134.2 | 753.8 | 15.20 | 30.97 | 58.53 | 92.60 | 122.94 |
| | TRT fp16 | 115.4 | 312.8 | 19.74 | 42.10 | 65.11 | 123.34 | 220.30 |

los obtenidos en la RTX 3060 son equivalentes.

Para esta evaluación, se considera un rango de batch sizes que incluye un mínimo de 1, un valor óptimo de 16 y un máximo de 32. Con el inference engine configurado de esta manera, se evaluará la latencia en inferencia con un batch size de 1 y el throughput para los valores restantes dentro del rango definido.

La Tabla 5.10 presenta los resultados de latencia y throughput para distintos batch sizes de inferencia. Como se observó en capítulos anteriores, la latencia se reduce significativamente al comparar el modelo base con la versión optimizada mediante TensorRT. Por otro lado, el throughput aumenta con el tamaño del batch, alcanzando su valor máximo con un batch size de 32. En las siguientes evaluaciones de esta subsección, se considerará únicamente un batch size de inferencia de 32.

La Tabla 5.11 muestra los resultados obtenidos al utilizar los niveles de optimización 0, 3 y

Tabla 5.11: Resultados de las métricas evaluadas para el modelo base y TRT fp16 construido con diferentes niveles de optimización (oplvl) en la plataforma Jetson Orin AGX.

| | Modelo | oplvl | Tconst [min] | Tamaño [MB] | Accuracy [K] | Memoria [GB] ave. | Memoria [GB] max. | Throughput [inf/s] |
|---|---|---|---|---|---|---|---|---|
| U-Net | Modelo Base | - | - | 7.3 | 15.3 | 3.4 | 3.8 | 396.9 |
| U-Net | TRT fp16 | 0 | 0 | 4.3 | 19.5 | 3.1 | 3.4 | 1162.2 |
| U-Net | TRT fp16 | 3 | 2 | 4.4 | 19.5 | 3.1 | 3.4 | 1640.2 |
| U-Net | TRT fp16 | 5 | 6 | 4.6 | 19.5 | 3.3 | 3.7 | 1672.0 |
| A.U-Net | Modelo Base | - | - | 33.8 | 31.0 | 3.7 | 4.2 | 183.0 |
| A.U-Net | TRT fp16 | 0 | 0 | 18.7 | 31.1 | 3.3 | 3.7 | 366.0 |
| A.U-Net | TRT fp16 | 3 | 2 | 18.4 | 31.0 | 3.3 | 3.7 | 481.3 |
| A.U-Net | TRT fp16 | 5 | - | - | - | - | - | - |

Tabla 5.12: Resultados de las métricas evaluadas para el modelo base y TRT fp16 construido con diferentes modos de consumo (PM) en la plataforma Jetson Orin AGX, considerando un nivel de optimización de cinco.

| | Modelo | PM | Tconst [min] | Tamaño [MB] | Accuracy [K] | Memoria [GB] ave. | Memoria [GB] max. | Throughput [inf/s] |
|---|---|---|---|---|---|---|---|---|
| U-Net | Modelo Base | PM2 | - | 7.3 | 15.3 | 3.4 | 3.8 | 396.9 |
| U-Net | Modelo Base | PM0 | - | 7.3 | 15.3 | 3.7 | 4.0 | 589.7 |
| U-Net | TRT fp16 | PM2 | 6 | 4.6 | 19.5 | 3.3 | 3.7 | 1672.0 |
| U-Net | TRT fp16 | PM0 | 5 | 4.6 | 19.5 | 3.2 | 3.5 | 2442.7 |
| A.U-Net | Modelo Base | PM2 | - | 33.8 | 31.0 | 3.7 | 4.2 | 183.0 |
| A.U-Net | Modelo Base | PM0 | - | 33.8 | 31.0 | 3.8 | 4.2 | 285.4 |
| A.U-Net | TRT fp16 | PM2 | 2 | 18.4 | 31.0 | 3.3 | 3.7 | 481.3 |
| A.U-Net | TRT fp16 | PM0 | 2 | 18.9 | 31.0 | 3.0 | 3.3 | 667.7 |

5 para la generación del engine. Cabe destacar que, en el caso de la red `Attention U-Net`, no fue posible generar el modelo con un nivel de optimización 5. Las métricas evaluadas incluyen el tiempo de construcción del inference engine (**Tconst**), el tamaño del modelo resultante (**Tamaño**), la accuracy (para resumir la información, en la tabla se muestran únicamente los resultados de regression accuracy para la condición de llama Yale-32), la memoria promedio (**ave.**), la memoria máxima (**max.**) y el throughput.

Los resultados presentados en la Tabla 5.11 indican que un aumento en el nivel de optimización permite **incrementar el throughput** sin comprometer la fidelidad de los resultados del modelo, ya que la métrica RMSE se mantiene constante en comparación con el modelo base. Asimismo, se observa que ni el tamaño del modelo ni el uso de memoria presentan variaciones significativas a medida que aumenta el nivel de optimización, registrándose un incremento aproximado de 200 MB en el uso de memoria solo en el nivel 5. Por último, aunque el tiempo de generación del engine también crece con el nivel de optimización, dicho aumento no es considerable, lo que se atribuye al tamaño reducido de los modelos evaluados.

La Tabla 5.12 presenta los resultados de la evaluación de los modos de consumo PM0 y PM2, utilizando las mismas métricas que en la Tabla 5.11. En esta ocasión, se consideraron el modelo base, el modelo U-Net `TRT fp16` con nivel de optimización 5 y el modelo Attention U-Net `TRT fp16` con nivel de optimización 3, ya que estos modelos obtuvieron los mejores resultados de throughput en la evaluación anterior.

Los resultados de la Tabla 5.12 indican que optar por un modo de consumo más permisivo —en términos de energía, frecuencia de operación y número de núcleos de CPU activos— permite **incrementar el throughput** sin comprometer la fidelidad de los resultados del modelo, ya que

la métrica RMSE se mantiene constante en comparación con el modelo base. Este aumento en el throughput no solo se observa en los modelos `TRT fp16`, sino que también beneficia al rendimiento del modelo base. En cuanto al uso de memoria durante la inferencia, se aprecia un incremento en el consumo de memoria para el modelo base al utilizar PM0 en lugar de PM2. No obstante, al realizar la misma comparación en los modelos `TRT fp16`, se observa una reducción de hasta 300 MB en el uso de memoria. Finalmente, se observa que ni el tamaño del modelo ni los tiempos de construcción se ven alterados de manera significativa al utilizar distintos modos de consumo. Aunque el tiempo de construcción se reduce levemente al emplear PM0, esta diferencia no resulta relevante dado el tamaño de los modelos considerados.

### 5.2.4. Discusión de resultados

Los resultados analizados en las secciones previas representan un estudio sobre la optimización de la inferencia mediante TensorRT en un entorno distinto al de los benchmarks reportados anteriormente. Este análisis aborda las limitaciones y vacíos identificados en la literatura, aspectos que fueron discutidos en la Sección 4.2.5 del capítulo anterior.

Al evaluar las métricas de **regression accuracy** y **regression closeness** para determinar la cuantización más adecuada, se observó que los resultados obtenidos mediante las optimizaciones de TensorRT varían significativamente según el modelo de red utilizado como base. En particular, se evidenció que el modelo `Attention U-Net` presenta una alta robustez en sus versiones optimizadas `TRT fp32` y `TRT fp16`, cuyos resultados, en términos de accuracy y closeness, indican que la salida de estos modelos es prácticamente indistinguible de la generada por el modelo base. En contraste, el modelo `U-Net` no mostró el mismo nivel de robustez, presentando discrepancias más pronunciadas respecto a su versión original. Respecto a la cuantización a `int8`, tal como se señaló en la Sección 4.2.5, esta afecta negativamente el desempeño en tareas de regresión para ambos modelos evaluados. Por consiguiente, se recomienda preferir optimizaciones con TensorRT utilizando cuantización a `fp16` en aplicaciones orientadas a tareas de regresión.

En relación con el resto de las configuraciones, los resultados asociados al batch size de inferencia, el nivel de optimización y el modo de consumo concuerdan con las conclusiones presentadas en la Sección 5.1.4. No obstante, además de dichas conclusiones, se observó que, **dependiendo del modelo base**, TensorRT podría no ser capaz de generar un modelo optimizado para ciertos **niveles de optimización**, como se evidenció en el caso de `Attention U-Net` al utilizar un nivel de optimización 5. Este resultado resalta la importancia de evaluar múltiples niveles de optimización con el fin de verificar la factibilidad de aplicar mejoras sobre un modelo base determinado.

Capítulo 6

# CONCLUSIONES Y TRABAJO FUTURO

En esta tesis se ha explorado la optimización de modelos de deep learning para su implementación en TensorRT, considerando distintas configuraciones de hardware, modos de consumo y parámetros de optimización de TensorRT. Se han evaluado métricas clave como latencia, throughput y uso de memoria durante la inferencia, con el objetivo de cuantificar el impacto de estas optimizaciones en plataformas embebidas.

En primer lugar, se logró validar las tendencias reportadas en el estado del arte. En particular, se demostró que, en entornos típicamente utilizados en procesos de benchmarking por la literatura, TensorRT reduce significativamente la latencia y aumenta el throughput de los modelos optimizados. No obstante, el impacto varía según el tipo de cuantización empleada, siendo `int8` la opción que ofrece las mayores reducciones en el tiempo de inferencia, a costa de una leve disminución en la precisión.

En cuanto a las configuraciones adicionales exploradas más allá del estado del arte, se evidenció que la configuración del batch size tiene un impacto directo en el rendimiento del modelo, siendo la configuración dinámica más versátil para la mayoría de los casos. Asimismo, se identificó que el uso de un modo de consumo más permisivo en cuanto al uso de recursos permite mejorar considerablemente el rendimiento tanto del modelo base como de los modelos optimizados. Finalmente, al evaluar distintos niveles de optimización, se observó que aplicar un mayor nivel puede mejorar levemente el rendimiento de los modelos, aunque a costa de un mayor tiempo de construcción.

Por otro lado, la evaluación de aplicaciones que emplean tareas como segmentación y regresión permitió validar la versatilidad de TensorRT en escenarios distintos al benchmarking, destacando su aplicabilidad en contextos que van más allá de la clasificación de imágenes convencional.

Los resultados obtenidos respaldan el uso de TensorRT como una herramienta efectiva para la optimización de inferencias en modelos de deep learning, aunque su implementación requiere un análisis detallado de las configuraciones más adecuadas para cada caso de uso específico. En este contexto, y considerando los hallazgos reportados en este trabajo, se proponen una serie de directrices que pueden resultar útiles al momento de utilizar TensorRT en diferentes entornos de operación:

- En general, se recomienda el uso de un batch size de configuración dinámico, debido a su versatilidad para aceptar distintos tamaños de batch en tiempo de inferencia y al requerir un menor uso de memoria durante la construcción del modelo, en comparación con la configuración estática.
- Se recomienda revisar cuidadosamente los *warnings* generados por TensorRT al momento de

construir los modelos optimizados y prestar especial atención a las configuraciones seleccionadas, ya que una configuración inadecuada puede generar comportamientos indeseados. En particular:

- Al configurar dimensiones dinámicas y definir un tamaño mínimo igual a cero, se genera un *warning* que indica que esta configuración puede aumentar significativamente el uso de memoria durante la inferencia.
- Al configurar un nivel de optimización igual a 5, pueden generarse *warnings* durante el proceso de construcción e incluso impedir la generación del modelo optimizado en ciertas configuraciones.

- Se recomienda evaluar distintos modelos de redes al momento de definir la arquitectura base, ya que se observó que existen modelos más robustos frente a las optimizaciones aplicadas por TensorRT. Estos modelos optimizados presentan mínimas diferencias en la salida respecto al modelo original.

A partir de los resultados obtenidos, se identifican diversas líneas de trabajo futuro que pueden complementar y extender los hallazgos de esta investigación:

- El desarrollo de este trabajo se basó en las versiones de TensorRT disponibles al momento de realizar los experimentos en plataformas Jetson, en particular, restringidas por la versión de JetPack. Por lo tanto, se recomienda continuar con evaluaciones periódicas que permitan verificar el funcionamiento de TensorRT en sus versiones más actuales, a medida que estas se incorporen a las plataformas embebidas mediante actualizaciones de software. Asimismo, se sugiere explorar nuevas funcionalidades incorporadas en versiones recientes, como el soporte para `int4` introducido en TensorRT 10.
- Se tiene considerado para trabajos futuros la evaluación de redes de control, en particular aquellas utilizadas para control automático mediante Model Predictive Control (MPC).
- Se recomienda explorar la optimización de modelos más complejos, incluyendo arquitecturas de redes neuronales recurrentes (RNNs) y modelos basados en *transformers*, los cuales han adquirido una creciente relevancia en tareas de procesamiento de lenguaje natural y visión artificial.
- Un desafío identificado en esta tesis es la complejidad asociada a la definición de configuraciones óptimas en TensorRT. El desarrollo de herramientas automatizadas que seleccionen las mejores configuraciones en función del modelo y del hardware disponible podría simplificar significativamente este proceso y facilitar su adopción en entornos productivos.

# REFERENCIAS


[1] M. Bojarski, D. Testa, D. Dworakowski, B. Firner, B. Flepp *et al.*, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016.

[2] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[3] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. Swetter *et al.*, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, Febrero 2017.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[5] U. Inc., *YOLOv8 Pretrained Segment Models*, https://docs.ultralytics.com/tasks/segment/#models. Accedido: Octubre 2024.

[6] P. Fekri, V. Abedi, J. Dargahi, and M. Zadeh, "A forward collision warning system using deep reinforcement learning," *SAE Technical Paper*, no. 2020-01-0138, 2020.

[7] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[8] A. Paszke, S. Gross, and S. Chintala, "Automatic differentiation in pytorch," 2017, https://pytorch.org/.

[9] M. Abadi, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Google, 2015, https://www.tensorflow.org/.

[10] F. Chollet, "Keras," https://github.com/fchollet/keras, 2015.

[11] T. Mickle and J. Rennison. Nvidia becomes most valuable public company, topping microsoft. https://www.nytimes.com/2024/06/18/technology/nvidia-most-valuable-company.html#. Accedido: Octubre 2024.

[12] Nvidia. Financial reports. https://investor.nvidia.com/financial-info/financial-reports/default.aspx. Accedido: Octubre 2024.

[13] ——. CUDA programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#. Accedido: Octubre 2024.

[14] M. Yang, "Avoiding pitfalls when using Nvidia gpus for real-time tasks in autonomous systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, 2018, pp. 20:1–20:21.

[15] E. Buber and B. Diri, “Performance analysis and CPU vs GPU comparison for deep learning,” in *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*, 2018, pp. 1–6.

[16] Nvidia, *Nvidia TensorRT*, Nvidia Corporation, https://developer.nvidia.com/tensorrt. Accedido: Octubre 2024.

[17] Y. Zhou and K. Yang, “Exploring TensorRT to improve real-time inference for deep learning,” in *IEEE 24th International Conference on High Performance Computing and Communications*, 2022, pp. 2011–2018.

[18] O. Shafi, C. Rai, R. Sen, and G. Ananthanarayanan, “Demystifying TensorRT: Characterizing neural network inference engine on Nvidia edge devices,” in *IEEE International Symposium on Workload Characterization*, 2021, pp. 226–237.

[19] G. Verma, Y. Gupta, A. Malik, and B. Chapman, “Performance evaluation of deep learning compilers for edge inference,” in *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2021, pp. 858–865.

[20] B. Ulker, S. Stuijk, H. Corporaal, and R. Wijnhoven, “Reviewing inference performance of state-of-the-art deep learning frameworks,” in *Proceedings of the 23rd International Workshop on Software and Compilers for Embedded Systems*, 2020, pp. 48–53.

[21] Nvidia, *Nvidia Jetson Orin Series*, Nvidia Corporation, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/. Accedido: Octubre 2024.

[22] ONNX, *Open Neural Network Exchange*, https://github.com/onnx/onnx. Accedido: Octubre 2024.

[23] L. Roeder, *Netron Tool*, https://github.com/lutzroeder/netron. Accedido: Octubre 2024.

[24] M. AI, *Torch-TensorRT*, https://pytorch.org/TensorRT/. Accedido: Octubre 2024.

[25] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers, 2010.

[26] T. Amert, “Gpu scheduling on the Nvidia tx2: Hidden details revealed,” in *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 104–115.

[27] Y. Fujii, “Data transfer matters for gpu computing,” in *2013 International Conference on Parallel and Distributed Systems*, 2013, pp. 275–282.

[28] Nvidia, *TensorRT 8.6 Documentation*, Nvidia Corporation, https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-861/developer-guide/index.html. Accedido: Octubre 2024.

[29] ——, *Selecting the Correct Workflow*, Nvidia Corporation, https://docs.nvidia.com/deeplearning/tensorrt/quick-start-guide/index.html#select-workflow. Accedido: Octubre 2024.

[30] ——, *IBuilderConfig*, Nvidia Corporation, https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/infer/Core/BuilderConfig.html. Accedido: Octubre 2024.

[31] A. Diomedi, “Técnicas de aceleración de inferencia para redes neuronales en plataformas embebidas: Una evaluación experimental,” Master’s thesis, Universidad Técnica Federico Santa María, 2021.

[32] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2755–2763.

[33] C. Tai, T. Xiao, X. Wang, and W. E, "Convolutional neural networks with low-rank regularization," *arXiv preprint arXiv:1511.06067*, 2015.

[34] M. Astrid and S. Lee, "Cp-decomposition with tensor power method for convolutional neural networks compression," in *2017 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2017, pp. 115–118.

[35] J. Aguilera, "Repositorio ArtTRT," https://github.com/Juanx65/ArtTRT, accedido: Octubre 2024.

[36] Nvidia, "Post-Training Quantization Using Calibration," https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-861/developer-guide/index.html#enable_int8_c, accessed: Agosto 2024.

[37] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2019. [En línea]. Disponible en: https://arxiv.org/abs/1801.04381

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [En línea]. Disponible en: https://arxiv.org/abs/1512.03385

[39] C. Zhu, J. Qian, and B. Wang, "Yolox on embedded device with cctv & tensorrt for intelligent multicategories garbage identification and classification," *IEEE Sensors Journal*, vol. 22, no. 16, pp. 16 275–16 283, 2022.

[40] Nvidia, "Jetson Modules," https://developer.nvidia.com/embedded/jetson-modules, accessed: Feb 2025.

[41] ——, "Supported Modes and Power Efficiency," https://docs.nvidia.com/jetson/archives/r35.1/DeveloperGuide/text/SD/PlatformPowerAndPerformance/JetsonOrinNxSeriesAndJetsonAgxOrinSeries.html#supported-modes-and-power-efficiency, accessed: Octubre 2024.

[42] ——, "Polygraphy," https://docs.nvidia.com/deeplearning/tensorrt/polygraphy/docs/index.html, accessed: Agosto 2024.

[43] ——, "EngineInspector," https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/infer/Core/EngineInspector.html, accessed: Agosto 2024.

[44] gmalivenko, "onnx-opcounter," https://github.com/gmalivenko/onnx-opcounter, accessed: Agosto 2024.

[45] P. S. Foundation, "getsize," https://docs.python.org/3/library/os.path.html#os.path.getsize, accessed: Agosto 2024.

[46] F. Hamanaka, T. Odan, K. Kise, and T. Chu, "An exploration of state-of-the-art automation frameworks for fpga-based dnn acceleration," *IEEE Access*, vol. 11, pp. 5701–5713, 2023.

[47] P. S. Foundation, "time," https://docs.python.org/3/library/time.html, accessed: Agosto 2024.

[48] R. Xu, F. Han, and Q. Ta, "Deep learning at scale on nvidia v100 accelerators," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2018, pp. 23–32.

[49] Nvidia, "tegrastats Utility," https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html, accessed: Agosto 2024.

[50] ——, "Jetson Product Lifecycle," https://developer.nvidia.com/embedded/lifecycle, accessed: Feb 2025.

[51] P. Contributors, "torch.isclose," https://pytorch.org/docs/stable/generated/torch.isclose.html, accessed: Agosto 2024.

[52] J. Aguilera, "Repositorio SalmonsTRT," https://github.com/Juanx65/SalmonsTRT, accedido: Octubre 2024.

[53] J. Aguilera and J. Portilla, "UNetTRT," https://github.com/Juanx65/UNetTRT, accessed: Feb 2025.

[54] X. Yang, S. Zhang, J. Liu, Q. Gao, S. Dong, and C. Zhou, "Deep learning for smart fish farming: applications, opportunities and challenges," *Reviews in Aquaculture 13 (1)*, pp. 66–90, 2020.

[55] M. Sun, X. Yang, and Y. Xie, "Deep learning in aquaculture: A review," *Journal of Computers 31 (1)*, p. 294–319, 2020.

[56] A. Saleh, I. Laradji, D. Konovalov, M. Bradley, D. Vazquez, and M. Sheaves, "A realistic fish-habitat dataset to evaluate algorithms for underwater visual analysis," *Scientific Reports 1089 10 (1)*, 2020.

[57] W. Xu and S. Matzner, "Underwater fish detection using deep learning for water power applications," in *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2018, pp. 313–318.

[58] A. Guerrero, "Análisis del modelo yolo-v8 para la segmentación de salmones en jaulas de cultivo en proceso estimador de biomasa en tiempo real," 2024, memoria de título, Universidad Técnica Federico Santa María.

[59] ultralytics, "Model Validation with Ultralytics YOLO," https://docs.ultralytics.com/modes/val/, accessed: Feb 2025.

[60] C. Schulz, B. Kock, M. Hofmann, H. Michelsen, S. Will *et al.*, "Laser-induced incandescence: recent trends and current questions," *Applied Physics B*, vol. 83, no. 3, pp. 333–354, June 2006.

[61] A. Guibaud, J. Citerne, J. Orlac'h, O. Fujita, J.-L. Consalvi *et al.*, "Broadband modulated absorption/emission technique to probe sooting flames: Implementation, validation, and limitations," *Proceedings of the Combustion Institute*, vol. 37, no. 3, pp. 3959–3966, 2019.

[62] J. Portilla, "Desarrollo de instrumentaciÓn de bajo costo para pirometrÍa de hollÍn usando cÁmaras a color y redes neuronales artificiales," Ph.D. dissertation, Universidad Técnica Federico Santa María, 2024.

[63] T. Ren, M. F. Modest, A. Fateev, G. Sutton, W. Zhao, and F. Rusu, "Machine learning applied to retrieval of temperature and concentration distributions from infrared emission measurements," *Applied Energy*, vol. 252, p. 113448, 2019.

[64] A. Rodríguez, F. Escudero, J. Cruz, G. Carvajal, and A. Fuentes, "Retrieving soot volume fraction fields for laminar axisymmetric diffusion flames using convolutional neural networks," *Fuel*, vol. 285, p. 119011, 2021.

[65] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," 2015. [En línea]. Disponible en: https://arxiv.org/abs/1505.04597

[66] O. Oktay, J. Schlemper, L. L. Folgoc, M. Lee, M. Heinrich *et al.*, "Attention u-net: Learning where to look for the pancreas," 2018. [En línea]. Disponible en: https://arxiv.org/abs/1804.03999

[67] J. Portilla, J. Cruz, F. Escudero, R. Demarco, A. Fuentes, and G. Carvajal, "A generalized neural network for accurate estimation of soot temperature in laminar flames using a single rgb image," *Journal of the Energy Institute*, vol. 119, p. 102001, 2025.

[68] J. Portilla, J. Cruz, F. Escudero, A. Rodríguez, R. Demarco *et al.*, "Towards low-cost soot pyrometry in laminar flames using broadband emission measurements and artificial neural networks," *Journal of the Energy Institute*, vol. 109, p. 101258, 2023.

[69] N. Eaves, Q. Zhang, F. Liu, H. Guo, S. Dworkin, and M. Thomson, "Coflame: A refined and validated numerical algorithm for modeling sooting laminar coflow diffusion flames," *Computer Physics Communications*, vol. 207, pp. 464–477, 2016.

[70] J. Gau, D. Das, C. McEnally, D. Giassi, N. Kempema, and M. Long, "Yale coflow diffusion flames steady flame burner," *URL: guilford. eng. yale. edu/yalecoflowflames/steady_burner. html*, 2014.

Apéndice A

# TABLAS DE RESULTADOS GENERALES

En las siguientes tablas se presentan un resumen de los resultados de rendimiento de distintas redes neuronales ejecutadas en todas las plataformas evaluadas en el Capítulo 3. Cada tabla incluye métricas clave como latencia promedio (**ave.**) y máxima (**max.**) , throughput promedio, propiedades del modelo (tamaño, número de capas y número de pesos) y precisión en Top-1 y Top-5 .

La columna **Thr.** indica el mayor throughput promedio alcanzado por cada configuración de la red neuronal, considerando un batch size de inferencia de 256 en la mayoría de los casos. Sin embargo, si el mayor throughput promedio se obtuvo con un batch size diferente, este valor se especifica con un subíndice $_{bX}$, donde $X$ representa el tamaño de batch correspondiente.

Los resultados incluyen distintas configuraciones de precisión para TensorRT (`fp32`, `fp16` e `int8`).

Tabla A.1: Resumen de resultados en la plataforma Jetson Orin AGX.

| | | Latencia [ms] | | Thr. [inf/s] | Propiedades del Modelo | | | Accuracy [%] | |
|---|---|---|---|---|---|---|---|---|---|
| | | ave. | max. | | Size [MB] | # Layers | # Weights | Top 1 | Top 5 |
| MobileNet | Modelo base | 15.6 | 33.3 | 293.0 | 13.6 | 104 | 3487816 | 72.02 | 90.62 |
| | TRT fp32 | 5.0 | 7.0 | 748.3 | 14.4 | 57 | 3469760 | 72.02 | 90.62 |
| | TRT fp16 | 3.4 | 7.2 | 1144.2 | 9.5 | 58 | 3469760 | 71.97 | 90.63 |
| | TRT int8 | 2.8 | 7.1 | 1603.2 | 5.6 | 57 | 3469760 | 71.44 | 90.31 |
| ResNet18 | Modelo base | 10.3 | 18.8 | $409.5_{b64}$ | 44.7 | 53 | 11684712 | 69.75 | 89.07 |
| | TRT fp32 | 5.7 | 7.7 | 651.6 | 46.5 | 26 | 11678912 | 69.76 | 89.08 |
| | TRT fp16 | 3.1 | 6.8 | 1214.4 | 24.3 | 27 | 11678912 | 69.78 | 89.09 |
| | TRT int8 | 2.2 | 5.5 | 1668.7 | 12.8 | 25 | 11669504 | 69.60 | 88.94 |
| ResNet34 | Modelo base | 16.0 | 26.5 | 258.4 | 83.3 | 93 | 21789160 | 73.31 | 91.42 |
| | TRT fp32 | 9.0 | 11.5 | 408.0 | 85.1 | 42 | 21779648 | 73.31 | 91.43 |
| | TRT fp16 | 5.1 | 7.9 | 804.7 | 43.7 | 43 | 21779648 | 73.30 | 91.43 |
| | TRT int8 | 3.6 | 5.6 | 1167.8 | 22.6 | 41 | 21770240 | 73.16 | 91.37 |
| ResNet50 | Modelo base | 18.3 | 272.6 | 155.5 | 97.8 | 126 | 25530472 | 80.34 | 95.13 |
| | TRT fp32 | 11.2 | 14.9 | $312.4_{b64}$ | 99.7 | 59 | 25502912 | 80.35 | 95.14 |
| | TRT fp16 | 6.4 | 8.9 | 583.8 | 51.2 | 60 | 25502912 | 80.36 | 95.13 |
| | TRT int8 | 4.7 | 7.2 | 918.4 | 27.2 | 58 | 25493504 | 79.78 | 95.08 |
| ResNet101 | Modelo base | 34.1 | 41.6 | 98.4 | 170.5 | 245 | 44496488 | 81.68 | 95.66 |
| | TRT fp32 | 18.1 | 20.8 | $191.2_{b128}$ | 172.2 | 112 | 44442816 | 81.67 | 95.67 |
| | TRT fp16 | 10.2 | 12.3 | 382.1 | 87.7 | 111 | 44442816 | 81.67 | 95.66 |
| | TRT int8 | 8.0 | 16.5 | 608.5 | 46.0 | 109 | 44433408 | 80.09 | 95.56 |
| ResNet152 | Modelo base | 49.1 | 53.7 | 70.0 | 230.5 | 364 | 60117096 | 82.32 | 95.92 |
| | TRT fp32 | 16.1 | 26.3 | $137.2_{b128}$ | 232.7 | 161 | 60040384 | 82.32 | 95.92 |
| | TRT fp16 | 13.9 | 15.8 | 258.5 | 117.5 | 162 | 60040384 | 82.33 | 95.91 |
| | TRT int8 | 10.2 | 12.3 | 432.4 | 61.5 | 160 | 60030976 | 80.47 | 95.75 |

Tabla A.2: Resumen de resultados en la plataforma Jetson Orin Nano.

| | | Latencia [ms] | | Thr. [inf/s] | Propiedades del Modelo | | | Accuracy [%] | |
|---|---|---|---|---|---|---|---|---|---|
| | | ave. | max. | | Size [MB] | # Layers | # Weights | Top 1 | Top 5 |
| MobileNet | Modelo base | 17.9 | 35.3 | $230.7_{b128}$ | 13.6 | 104 | 3487816 | 72.02 | 90.62 |
| | TRT fp32 | 5.8 | 13.0 | 682.4 | 14.6 | 60 | 3469760 | 72.01 | 90.62 |
| | TRT fp16 | 3.7 | 11.8 | 1089.3 | 8.3 | 58 | 3469760 | 72.00 | 90.64 |
| | TRT int8 | 2.9 | 13.6 | 1555.3 | 5.5 | 57 | 3469760 | 71.45 | 90.29 |
| ResNet18 | Modelo base | 11.2 | 19.8 | 343.9 | 44.7 | 53 | 11684712 | 69.75 | 89.07 |
| | TRT fp32 | 5.9 | 12.5 | 644.8 | 46.5 | 26 | 11678912 | 69.76 | 89.08 |
| | TRT fp16 | 3.6 | 12.0 | 1172.5 | 24.3 | 27 | 11678912 | 69.75 | 89.08 |
| | TRT int8 | 2.4 | 10.3 | 1727.7 | 12.8 | 25 | 11669504 | 69.57 | 88.95 |
| ResNet34 | Modelo base | 17.7 | 28.8 | 220.2 | 83.3 | 93 | 21789160 | 73.31 | 91.42 |
| | TRT fp32 | 9.7 | 14.0 | 409.4 | 84.2 | 42 | 21779648 | 73.31 | 91.43 |
| | TRT fp16 | 5.8 | 12.5 | 794.0 | 43.6 | 43 | 21779648 | 73.32 | 91.45 |
| | TRT int8 | 3.7 | 11.7 | 1270.8 | 22.5 | 41 | 21770240 | 73.18 | 91.41 |
| ResNet50 | Modelo base | 21.1 | 36.2 | $112.7_{b128}$ | 97.8 | 126 | 25530472 | 80.34 | 95.13 |
| | TRT fp32 | 12.5 | 15.6 | $286.0_{b128}$ | 100.7 | 59 | 25502912 | 80.35 | 95.14 |
| | TRT fp16 | 6.9 | 19.3 | 545.3 | 51.5 | 60 | 25502912 | 80.35 | 95.13 |
| | TRT int8 | 5.1 | 21.3 | 890.2 | 27.1 | 58 | 25493504 | 79.75 | 95.07 |
| ResNet101 | Modelo base | 38.8 | 44.6 | $71.9_{b128}$ | 170.5 | 245 | 44496488 | 81.68 | 95.66 |
| | TRT fp32 | 15.7 | 32.0 | $180.2_{b128}$ | 173.2 | 110 | 44442816 | 81.67 | 95.67 |
| | TRT fp16 | 11.0 | 33.6 | 351.0 | 87.6 | 111 | 44442816 | 81.67 | 95.65 |
| | TRT int8 | 8.3 | 14.8 | 590.8 | 45.8 | 109 | 44433408 | 79.95 | 95.55 |
| ResNet152 | Modelo base | 56.2 | 60.5 | $51.2_{b128}$ | 230.5 | 364 | 60117096 | 82.32 | 95.92 |
| | TRT fp32 | 16.9 | 23.8 | $127.3_{b128}$ | 233.2 | 161 | 60040384 | 82.32 | 95.92 |
| | TRT fp16 | 14.7 | 20.0 | 248.2 | 117.5 | 161 | 60040384 | 82.33 | 95.91 |
| | TRT int8 | 11.3 | 14.4 | 452.1 | 61.4 | 160 | 60030976 | 82.04 | 95.84 |

Tabla A.3: Resumen de resultados en la plataforma Jetson Xavier AGX.

| | | Latencia [ms] | | Thr. [inf/s] | Propiedades del Modelo | | | Accuracy [%] | |
|---|---|---|---|---|---|---|---|---|---|
| | | ave. | max. | | Size [MB] | # Layers | # Weights | Top 1 | Top 5 |
| MobileNet | Modelo base | 12.8 | 31.5 | 242.8 | 13.6 | 104 | - | 72.01 | 90.62 |
| | TRT fp32 | 7.1 | 16.1 | 470.7 | 14.3 | 76 | 3469760 | 72.01 | 90.62 |
| | TRT fp16 | 4.8 | 13.4 | 864.7 | 7.5 | 57 | 3469760 | 72.00 | 90.63 |
| | TRT int8 | 3.7 | 12.0 | 1284.0 | 4.4 | 57 | 3469760 | 71.39 | 90.30 |
| ResNet18 | Modelo base | 10.8 | 18.8 | 231.3 | 44.7 | 53 | - | 69.76 | 89.08 |
| | TRT fp32 | 8.1 | 13.0 | 237.5 | 66.3 | 24 | 11678912 | 69.76 | 89.08 |
| | TRT fp16 | 5.3 | 13.0 | 755.4 | 23.1 | 26 | 11678912 | 69.75 | 89.08 |
| | TRT int8 | 3.6 | 15.7 | 1256.1 | 11.9 | 25 | 11669504 | 69.59 | 88.99 |
| ResNet34 | Modelo base | 17.4 | 22.3 | 142.4 | 83.3 | 93 | - | 73.30 | 91.42 |
| | TRT fp32 | 10.9 | 17.2 | 140.3 | 127.8 | 40 | 21779648 | 73.30 | 91.42 |
| | TRT fp16 | 7.0 | 24.0 | 468.9 | 42.5 | 43 | 21779648 | 73.29 | 91.42 |
| | TRT int8 | 5.6 | 16.0 | 840.0 | 21.6 | 41 | 21770240 | 73.25 | 91.41 |
| ResNet50 | Modelo base | 24.0 | 30.7 | 68.9 | 97.8 | 126 | - | 80.35 | 95.13 |
| | TRT fp32 | 16.7 | 23.7 | 92.4 | 108.2 | 57 | 25502912 | 80.35 | 95.13 |
| | TRT fp16 | 7.6 | 15.3 | 355.4 | 49.9 | 59 | 25502912 | 80.35 | 95.12 |
| | TRT int8 | 6.8 | 22.9 | 612.9 | 25.6 | 58 | 25493504 | 79.85 | 95.07 |
| ResNet101 | Modelo base | 41.4 | 49.0 | 41.6 | 170.5 | 245 | - | 81.67 | 95.66 |
| | TRT fp32 | 28.9 | 33.4 | 54.3 | 210.4 | 119 | 44442816 | 81.67 | 95.66 |
| | TRT fp16 | 10.6 | 15.6 | $217.2_{b128}$ | 86.2 | 110 | 44442816 | 81.66 | 95.67 |
| | TRT int8 | 8.0 | 24.3 | 386.6 | 44.4 | 109 | 44433408 | 80.83 | 95.57 |
| ResNet152 | Modelo base | 58.7 | 65.0 | 29.2 | 230.5 | 364 | - | 82.35 | 95.92 |
| | TRT fp32 | 41.1 | 45.4 | 38.3 | 294.7 | 159 | 60040384 | 82.35 | 95.92 |
| | TRT fp16 | 14.3 | 18.6 | $153.4_{b128}$ | 116.1 | 161 | 60040384 | 82.34 | 95.91 |
| | TRT int8 | 9.4 | 13.4 | 286.1 | 59.9 | 160 | 60030976 | 81.27 | 95.84 |

Tabla A.4: Resumen de resultados en la plataforma RTX 2060MaxQ.

| | | Latencia [ms] | | Thr. [inf/s] | Propiedades del Modelo | | | Accuracy [%] | |
|---|---|---|---|---|---|---|---|---|---|
| | | ave. | max. | | Size [MB] | # Layers | # Weights | Top 1 | Top 5 |
| MobileNet | Modelo base | 4.4 | 12.7 | $997.7_{b128}$ | 13.6 | 104 | 3487816 | 72.01 | 90.62 |
| | TRT fp32 | 1.1 | 2.4 | $2065.9_{b128}$ | 14.0 | 88 | 3469760 | 72.01 | 90.62 |
| | TRT fp16 | 0.9 | 3.1 | 3463.7 | 10.3 | 57 | 3469760 | 71.99 | 90.62 |
| | TRT int8 | 0.8 | 2.6 | 4389.1 | 5.7 | 57 | 3469760 | 71.44 | 90.34 |
| ResNet18 | Modelo base | 2.7 | 6.5 | 1117.1 | 44.7 | 53 | 11684712 | 69.76 | 89.08 |
| | TRT fp32 | 1.5 | 2.9 | $1096.0_{b64}$ | 65.8 | 35 | 11678912 | 69.76 | 89.08 |
| | TRT fp16 | 1.0 | 3.2 | 3322.4 | 24.9 | 27 | 11678912 | 69.76 | 89.09 |
| | TRT int8 | 0.9 | 1.9 | 4671.2 | 13.3 | 25 | 11669504 | 69.57 | 88.94 |
| ResNet34 | Modelo base | 4.5 | 8.1 | 685.9 | 83.3 | 93 | 21789160 | 73.30 | 91.42 |
| | TRT fp32 | 2.7 | 4.2 | $584.8_{b128}$ | 125.5 | 45 | 21779648 | 73.30 | 91.42 |
| | TRT fp16 | 1.2 | 4.6 | $2457.4_{b128}$ | 44.2 | 47 | 21779648 | 73.31 | 91.42 |
| | TRT int8 | 1.1 | 2.4 | 3631.3 | 23.0 | 41 | 21770240 | 73.31 | 91.33 |
| ResNet50 | Modelo base | 5.5 | 11.3 | 361.9 | 97.8 | 126 | 25530472 | 80.35 | 95.13 |
| | TRT fp32 | 3.3 | 4.6 | $461.0_{b128}$ | 107.1 | 97 | 25502912 | 80.35 | 95.13 |
| | TRT fp16 | 1.3 | 3.5 | $1811.5_{b128}$ | 50.1 | 64 | 25502912 | 80.37 | 95.13 |
| | TRT int8 | 1.1 | 2.7 | $2948.1_{b128}$ | 27.2 | 58 | 25493504 | 78.59 | 94.96 |
| ResNet101 | Modelo base | 9.6 | 16.3 | 220.5 | 170.5 | 245 | 44496488 | 81.67 | 95.66 |
| | TRT fp32 | 6.0 | 7.0 | $262.3_{b128}$ | 209.5 | 172 | 44442816 | 81.67 | 95.66 |
| | TRT fp16 | 2.3 | 4.1 | $1173.8_{b64}$ | 88.0 | 111 | 44442816 | 81.65 | 95.67 |
| | TRT int8 | 1.7 | 3.8 | $2118.8_{b128}$ | 46.0 | 109 | 44433408 | 79.91 | 95.58 |
| ResNet152 | Modelo base | 13.9 | 22.8 | 155.2 | 230.5 | 364 | 60117096 | 82.35 | 95.92 |
| | TRT fp32 | 8.7 | 30.6 | $183.0_{b128}$ | 291.9 | 258 | 60040384 | 82.35 | 95.92 |
| | TRT fp16 | 3.1 | 5.2 | $833.0_{b64}$ | 116.1 | 166 | 60040384 | 82.32 | 95.92 |
| | TRT int8 | 2.3 | 6.8 | $1539.3_{b64}$ | 61.5 | 160 | 60030976 | 79.96 | 95.74 |

Tabla A.5: Resumen de resultados en la plataforma RTX 3060.

| | | Latencia [ms] | | Thr. [inf/s] | Propiedades del Modelo | | | Accuracy [%] | |
|---|---|---|---|---|---|---|---|---|---|
| | | ave. | max. | | Size [MB] | # Layers | # Weights | Top 1 | Top 5 |
| MobileNet | Modelo base | 2.4 | 4.9 | 1305.6 | 13.6 | 104 | 3487816 | 72.02 | 90.63 |
| | TRT fp32 | 0.8 | 5.0 | 2670.0 | 14.2 | 57 | 3469760 | 72.02 | 90.62 |
| | TRT fp16 | 0.7 | 1.8 | 3529.2 | 8.5 | 58 | 3469760 | 71.98 | 90.63 |
| | TRT int8 | 0.6 | 1.7 | 4098.0 | 6.1 | 57 | 3469760 | 71.42 | 90.29 |
| ResNet18 | Modelo base | 1.7 | 4.0 | 1421.8 | 44.7 | 53 | 11684712 | 69.76 | 89.08 |
| | TRT fp32 | 1.3 | 1.8 | 1903.4 | 66.4 | 26 | 11678912 | 69.75 | 89.09 |
| | TRT fp16 | 0.7 | 1.3 | 3390.5 | 25.0 | 27 | 11678912 | 69.77 | 89.10 |
| | TRT int8 | 0.6 | 1.3 | 4251.9 | 13.3 | 25 | 11669504 | 69.58 | 88.94 |
| ResNet34 | Modelo base | 2.8 | 5.6 | 921.3 | 83.3 | 93 | 21789160 | 73.29 | 91.42 |
| | TRT fp32 | 2.0 | 2.5 | 1214.7 | 128.0 | 42 | 21779648 | 73.29 | 91.42 |
| | TRT fp16 | 0.9 | 1.5 | 2668.6 | 44.4 | 43 | 21779648 | 73.31 | 91.43 |
| | TRT int8 | 0.7 | 1.5 | 3665.7 | 23.1 | 41 | 21770240 | 73.28 | 91.35 |
| ResNet50 | Modelo base | 3.3 | 5.6 | 577.0 | 97.8 | 126 | 25530472 | 80.34 | 95.12 |
| | TRT fp32 | 2.2 | 2.7 | 1066.5 | 107.9 | 79 | 25502912 | 80.34 | 95.13 |
| | TRT fp16 | 1.1 | 1.7 | 2187.4 | 50.7 | 60 | 25502912 | 80.36 | 95.13 |
| | TRT int8 | 0.8 | 1.6 | 3178.2 | 28.0 | 58 | 25493504 | 78.55 | 94.97 |
| ResNet101 | Modelo base | 5.7 | 8.6 | 365.5 | 170.5 | 245 | 44496488 | 81.67 | 95.66 |
| | TRT fp32 | 3.8 | 4.3 | 677.3 | 210.3 | 164 | 44442816 | 81.66 | 95.67 |
| | TRT fp16 | 1.6 | 2.2 | 1586.7 | 87.0 | 111 | 44442816 | 81.66 | 95.66 |
| | TRT int8 | 1.2 | 2.1 | 2516.5 | 46.8 | 109 | 44433408 | 79.93 | 95.61 |
| ResNet152 | Modelo base | 8.0 | 11.5 | 259.8 | 230.5 | 364 | 60117096 | 82.34 | 95.92 |
| | TRT fp32 | 5.4 | 5.8 | 489.8 | 292.8 | 241 | 60040384 | 82.34 | 95.92 |
| | TRT fp16 | 2.2 | 2.6 | 1225.7 | 116.9 | 162 | 60040384 | 82.33 | 95.91 |
| | TRT int8 | 1.5 | 2.6 | 2053.4 | 62.4 | 160 | 60030976 | 79.96 | 95.76 |

Apéndice B

# TABLAS DE RESULTADOS COMPLETOS DEL THROUGHPUT

En las siguientes tablas se presentan los resultados de throughput para todos los modelos de red evaluados en distintas plataformas de hardware en el Capítulo 3. Se evalúan diferentes configuraciones de precisión (`fp32`, `fp16` e `int8`) utilizando TensorRT, y se comparan con el modelo base sin optimizaciones.

Para cada configuración, los resultados se presentan considerando distintos batch size de inferencia, lo que permite observar cómo varía el rendimiento en función de este parámetro. Adicionalmente, junto a cada medición de throughput, se muestra el intervalo de confianza del 95 %, representado por la columna `±(95 %)`. Este valor indica la variabilidad de las mediciones y proporciona una estimación del rango dentro del cual se espera que se encuentren el 95 % de los resultados si se repitieran las mediciones en condiciones similares. Un intervalo más estrecho sugiere una mayor estabilidad en los resultados, mientras que un intervalo más amplio indica una mayor variabilidad en las mediciones.

Tabla B.1: Resultados de throughput para el modelo de red MobileNetV2.

| | | **Modelo Base** | | `TRT fp32` | | `TRT fp16` | | `TRT int8` | |
|---|---|---|---|---|---|---|---|---|---|
| | Batch size | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) |
| RTX 3060 | ***1*** | **408.3** | 112.4 | **1193.4** | 136.2 | **1441.6** | 226.3 | **1555.9** | 291.4 |
| | ***32*** | **1240.5** | 11.9 | **2456.7** | 49.6 | **3218.8** | 75.9 | **3735.2** | 106.2 |
| | ***64*** | **1273.5** | 7.9 | **2570.2** | 30.7 | **3373.9** | 59.0 | **3918.9** | 84.8 |
| | ***128*** | **1294.4** | 6.9 | **2636.8** | 22.1 | **3477.2** | 41.1 | **4039.5** | 58.4 |
| | ***256*** | **1305.6** | 6.7 | **2670.0** | 23.1 | **3529.2** | 39.8 | **4098.0** | 48.4 |
| RTX 2060 | ***1*** | **228.1** | 27.4 | **906.8** | 112.6 | **1106.6** | 132.2 | **1221.5** | 155.2 |
| | ***32*** | **966.0** | 42.8 | **1909.4** | 254.5 | **2794.6** | 347.7 | **3511.9** | 539.0 |
| | ***64*** | **988.4** | 29.0 | **2021.7** | 182.7 | **2919.9** | 348.8 | **3768.0** | 534.1 |
| | ***128*** | **997.7** | 27.8 | **2065.9** | 153.1 | **3322.3** | 593.6 | **3818.4** | 663.8 |
| | ***256*** | **993.6** | 27.3 | **1892.5** | 113.1 | **3463.7** | 387.7 | **4389.1** | 672.3 |
| Xavier AGX | ***1*** | **78.1** | 15.8 | **141.2** | 30.6 | **210.4** | 39.5 | **273.7** | 87.6 |
| | ***32*** | **228.4** | 7.1 | **445.2** | 58.6 | **820.5** | 202.8 | **985.5** | 399.9 |
| | ***64*** | **238.5** | 7.0 | **439.0** | 46.3 | **789.7** | 174.0 | **1148.3** | 389.3 |
| | ***128*** | **242.6** | 4.8 | **460.5** | 39.2 | **863.5** | 142.4 | **1272.2** | 328.3 |
| | ***256*** | **242.8** | 5.9 | **470.7** | 40.1 | **864.7** | 182.6 | **1284.0** | 407.2 |
| Orin Nano | ***1*** | **55.8** | 3.7 | **173.7** | 42.3 | **271.1** | 69.2 | **342.7** | 79.3 |
| | ***32*** | **227.4** | 2.3 | **557.0** | 200.2 | **715.6** | 413.6 | **956.2** | 134.5 |
| | ***64*** | **229.3** | 2.2 | **584.3** | 88.9 | **828.6** | 251.5 | **1230.1** | 722.9 |
| | ***128*** | **230.7** | 2.1 | **649.3** | 61.3 | **965.9** | 115.5 | **1282.8** | 148.7 |
| | ***256*** | **227.5** | 36.0 | **682.4** | 37.2 | **1089.3** | 90.5 | **1555.3** | 136.9 |
| Orin AGX | ***1*** | **64.0** | 4.7 | **200.7** | 23.9 | **291.2** | 54.1 | **358.7** | 61.9 |
| | ***32*** | **281.1** | 11.9 | **646.7** | 144.3 | **813.0** | 434.2 | **1005.3** | 29.1 |
| | ***64*** | **288.1** | 5.4 | **673.0** | 125.7 | **914.0** | 294.9 | **1226.2** | 643.7 |
| | ***128*** | **290.6** | 5.5 | **725.7** | 95.2 | **1019.7** | 113.8 | **1322.0** | 184.0 |
| | ***256*** | **293.0** | 3.2 | **748.3** | 63.2 | **1144.2** | 84.4 | **1603.2** | 157.1 |

Tabla B.2: Resultados de throughput para el modelo de red ResNet18.

| | | **Modelo Base** | | `TRT fp32` | | `TRT fp16` | | `TRT int8` | |
|---|---|---|---|---|---|---|---|---|---|
| | Batch size | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) |
| RTX 3060 | ***1*** | **588.7** | 96.0 | **784.8** | 50.6 | **1430.7** | 209.2 | **1727.7** | 310.4 |
| | ***32*** | **1311.4** | 13.3 | **1761.3** | 23.6 | **3066.2** | 72.8 | **3874.8** | 113.1 |
| | ***64*** | **1378.5** | 8.7 | **1833.4** | 15.2 | **3221.3** | 65.4 | **4080.7** | 83.3 |
| | ***128*** | **1419.9** | 6.8 | **1890.8** | 14.8 | **3325.7** | 48.2 | **4187.1** | 71.1 |
| | ***256*** | **1421.8** | 5.6 | **1903.4** | 11.7 | **3390.5** | 45.0 | **4251.9** | 51.9 |
| RTX 2060 | ***1*** | **367.6** | 45.5 | **645.2** | 40.2 | **961.5** | 126.5 | **1136.1** | 157.6 |
| | ***32*** | **1066.0** | 64.6 | **1093.0** | 203.1 | **2473.7** | 247.4 | **3645.1** | 577.4 |
| | ***64*** | **1098.9** | 46.0 | **1096.0** | 162.5 | **2727.4** | 367.6 | **3889.1** | 585.0 |
| | ***128*** | **1096.2** | 32.7 | **1070.8** | 39.7 | **3135.7** | 728.8 | **4216.7** | 737.2 |
| | ***256*** | **1117.1** | 27.4 | **1045.2** | 32.8 | **3322.4** | 524.3 | **4671.2** | 783.1 |
| Xavier AGX | ***1*** | **92.5** | 6.5 | **122.8** | 9.7 | **190.2** | 27.8 | **274.2** | 83.8 |
| | ***32*** | **218.7** | 4.2 | **211.5** | 4.3 | **707.7** | 131.6 | **1147.0** | 358.8 |
| | ***64*** | **223.3** | 4.9 | **224.1** | 6.3 | **689.2** | 138.8 | **1078.2** | 337.4 |
| | ***128*** | **227.1** | 3.3 | **233.0** | 4.7 | **741.8** | 119.4 | **1210.1** | 303.7 |
| | ***256*** | **231.3** | 3.7 | **237.5** | 4.8 | **755.4** | 129.2 | **1256.1** | 372.8 |
| Orin Nano | ***1*** | **88.9** | 10.2 | **169.1** | 22.5 | **277.3** | 87.4 | **425.4** | 122.9 |
| | ***32*** | **304.3** | 9.8 | **511.0** | 141.1 | **772.1** | 452.4 | **1108.6** | 44.5 |
| | ***64*** | **330.6** | 10.8 | **559.9** | 97.1 | **889.5** | 240.2 | **1300.8** | 642.9 |
| | ***128*** | **334.2** | 10.5 | **615.5** | 62.1 | **1048.8** | 144.1 | **1408.8** | 205.2 |
| | ***256*** | **343.9** | 9.3 | **644.8** | 34.4 | **1172.5** | 92.2 | **1727.7** | 139.6 |
| Orin AGX | ***1*** | **96.7** | 14.8 | **175.0** | 18.6 | **317.8** | 74.1 | **456.5** | 54.8 |
| | ***32*** | **378.9** | 14.4 | **594.9** | 84.4 | **862.9** | 509.0 | **1074.9** | 31.1 |
| | ***64*** | **409.5** | 13.9 | **610.2** | 91.1 | **1000.3** | 433.0 | **1256.0** | 572.5 |
| | ***128*** | **404.9** | 10.5 | **636.2** | 65.1 | **1091.1** | 111.8 | **1388.7** | 216.9 |
| | ***256*** | **403.5** | 9.7 | **651.6** | 41.2 | **1214.4** | 94.2 | **1668.7** | 146.6 |

Tabla B.3: Resultados de throughput para el modelo de red ResNet34.

| | | **Modelo Base** | | `TRT fp32` | | `TRT fp16` | | `TRT int8` | |
|---|---|---|---|---|---|---|---|---|---|
| | Batch size | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) |
| RTX 3060 | ***1*** | **352.6** | 41.8 | **496.1** | 16.7 | **1103.3** | 105.0 | **1412.9** | 195.4 |
| | ***32*** | **835.3** | 3.3 | **1115.2** | 10.6 | **2382.8** | 44.0 | **3308.6** | 94.4 |
| | ***64*** | **883.8** | 3.8 | **1173.0** | 7.3 | **2506.1** | 28.9 | **3503.7** | 65.5 |
| | ***128*** | **914.0** | 14.3 | **1205.2** | 7.4 | **2611.7** | 26.8 | **3615.8** | 50.7 |
| | ***256*** | **921.3** | 15.3 | **1214.7** | 17.3 | **2668.6** | 26.2 | **3665.7** | 51.3 |
| RTX 2060 | ***1*** | **223.2** | 20.0 | **374.1** | 16.1 | **813.6** | 78.3 | **928.7** | 108.3 |
| | ***32*** | **647.1** | 15.5 | **578.6** | 10.9 | **1943.0** | 335.8 | **2741.0** | 293.0 |
| | ***64*** | **672.4** | 13.5 | **578.8** | 10.6 | **2377.5** | 307.7 | **2932.4** | 297.4 |
| | ***128*** | **680.0** | 11.8 | **584.8** | 12.9 | **2457.4** | 296.1 | **3449.5** | 941.1 |
| | ***256*** | **685.9** | 12.2 | **579.9** | 12.1 | **2274.2** | 159.4 | **3631.3** | 661.9 |
| Xavier AGX | ***1*** | **57.5** | 1.2 | **91.4** | 4.9 | **142.5** | 11.4 | **178.6** | 25.1 |
| | ***32*** | **131.3** | 0.7 | **116.8** | 0.9 | **423.9** | 46.3 | **783.5** | 165.4 |
| | ***64*** | **133.9** | 0.8 | **128.6** | 0.8 | **427.5** | 46.1 | **748.4** | 148.4 |
| | ***128*** | **138.4** | 0.5 | **136.4** | 0.5 | **461.2** | 45.1 | **811.0** | 125.6 |
| | ***256*** | **142.4** | 0.5 | **140.3** | 0.5 | **468.9** | 44.7 | **840.0** | 153.8 |
| Orin Nano | ***1*** | **56.5** | 2.5 | **102.8** | 2.7 | **172.0** | 39.3 | **269.1** | 69.2 |
| | ***32*** | **192.0** | 1.7 | **364.6** | 25.3 | **596.6** | 303.8 | **850.2** | 548.9 |
| | ***64*** | **212.9** | 1.8 | **386.9** | 24.0 | **619.3** | 81.9 | **1001.3** | 424.2 |
| | ***128*** | **214.3** | 1.5 | **400.7** | 23.7 | **740.3** | 86.7 | **1136.2** | 136.8 |
| | ***256*** | **220.2** | 1.5 | **409.4** | 15.0 | **794.0** | 57.4 | **1270.8** | 100.0 |
| Orin AGX | ***1*** | **62.5** | 2.4 | **110.7** | 7.6 | **197.4** | 30.3 | **278.6** | 43.9 |
| | ***32*** | **237.7** | 7.0 | **373.7** | 6.4 | **652.0** | 154.3 | **815.2** | 449.1 |
| | ***64*** | **258.2** | 4.5 | **393.6** | 10.9 | **693.2** | 128.2 | **934.8** | 344.4 |
| | ***128*** | **256.6** | 4.8 | **404.5** | 9.4 | **760.0** | 96.2 | **1037.8** | 112.5 |
| | ***256*** | **258.4** | 4.0 | **408.0** | 5.1 | **804.7** | 59.2 | **1167.8** | 94.6 |

Tabla B.4: Resultados de throughput para el modelo de red ResNet50.

| | | **Modelo Base** | | `TRT fp32` | | `TRT fp16` | | `TRT int8` | |
|---|---|---|---|---|---|---|---|---|---|
| | Batch size | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) |
| RTX 3060 | ***1*** | **298.7** | 30.3 | **447.1** | 15.3 | **945.7** | 74.2 | **1224.5** | 135.3 |
| | ***32*** | **535.7** | 2.3 | **982.9** | 8.2 | **1973.8** | 30.0 | **2887.8** | 62.9 |
| | ***64*** | **555.3** | 2.1 | **1028.3** | 5.2 | **2069.5** | 19.2 | **3042.0** | 96.6 |
| | ***128*** | **568.0** | 2.1 | **1055.0** | 6.9 | **2148.7** | 17.2 | **3128.5** | 45.0 |
| | ***256*** | **577.0** | 2.8 | **1066.5** | 16.8 | **2187.4** | 16.0 | **3178.2** | 32.4 |
| RTX 2060 | ***1*** | **181.1** | 17.5 | **303.8** | 11.0 | **759.0** | 56.5 | **924.4** | 97.7 |
| | ***32*** | **347.1** | 3.2 | **455.4** | 6.5 | **1761.4** | 154.2 | **2247.7** | 226.3 |
| | ***64*** | **357.5** | 2.9 | **457.1** | 9.3 | **1784.9** | 163.7 | **2640.6** | 441.1 |
| | ***128*** | **361.0** | 2.6 | **461.0** | 7.9 | **1811.5** | 120.9 | **2948.1** | 325.7 |
| | ***256*** | **361.9** | 2.2 | **459.6** | 7.7 | **1683.3** | 85.0 | **2911.8** | 319.7 |
| Xavier AGX | ***1*** | **41.7** | 0.8 | **59.8** | 1.4 | **130.9** | 8.5 | **146.0** | 13.0 |
| | ***32*** | **65.6** | 0.2 | **84.8** | 0.5 | **333.4** | 26.7 | **586.9** | 91.0 |
| | ***64*** | **67.3** | 0.1 | **89.1** | 0.3 | **341.4** | 28.8 | **573.0** | 82.3 |
| | ***128*** | **68.3** | 0.1 | **91.5** | 0.2 | **351.2** | 24.9 | **600.0** | 63.6 |
| | ***256*** | **68.9** | 0.1 | **92.4** | 0.2 | **355.4** | 23.7 | **612.9** | 78.5 |
| Orin Nano | ***1*** | **47.4** | 2.2 | **80.0** | 1.6 | **145.7** | 8.1 | **196.1** | 40.0 |
| | ***32*** | **104.5** | 0.7 | **269.3** | 8.2 | **466.3** | 110.7 | **632.1** | 356.9 |
| | ***64*** | **110.4** | 0.7 | **280.7** | 3.9 | **497.5** | 72.2 | **690.0** | 168.2 |
| | ***128*** | **112.7** | 0.7 | **286.0** | 4.4 | **533.3** | 48.4 | **812.3** | 84.1 |
| | ***256*** | **64.4** | 63.5 | **282.8** | 4.0 | **545.3** | 27.5 | **890.2** | 68.2 |
| Orin AGX | ***1*** | **54.5** | 3.7 | **89.6** | 6.7 | **156.7** | 20.7 | **213.2** | 29.4 |
| | ***32*** | **140.5** | 1.8 | **303.8** | 6.2 | **555.5** | 57.9 | **701.6** | 225.5 |
| | ***64*** | **150.7** | 2.2 | **312.4** | 5.9 | **575.5** | 64.2 | **760.8** | 182.7 |
| | ***128*** | **152.8** | 1.4 | **311.7** | 6.1 | **582.9** | 45.9 | **840.4** | 85.0 |
| | ***256*** | **155.5** | 0.8 | **310.6** | 4.2 | **583.8** | 36.4 | **918.4** | 77.2 |

Tabla B.5: Resultados de throughput para el modelo de red ResNet101.

| | | **Modelo Base** | | `TRT fp32` | | `TRT fp16` | | `TRT int8` | |
|---|---|---|---|---|---|---|---|---|---|
| | Batch size | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) |
| RTX 3060 | ***1*** | **175.1** | 16.2 | **260.9** | 4.4 | **617.2** | 28.3 | **867.5** | 60.5 |
| | ***32*** | **335.3** | 1.0 | **595.1** | 2.3 | **1395.0** | 17.3 | **2265.0** | 39.6 |
| | ***64*** | **348.8** | 1.2 | **660.5** | 11.9 | **1464.7** | 11.6 | **2397.7** | 28.3 |
| | ***128*** | **355.1** | 1.2 | **675.2** | 1.4 | **1555.6** | 8.8 | **2479.5** | 18.6 |
| | ***256*** | **365.5** | 1.2 | **677.3** | 4.4 | **1586.7** | 8.0 | **2516.5** | 20.4 |
| RTX 2060 | ***1*** | **104.0** | 9.4 | **166.0** | 4.4 | **431.4** | 19.3 | **590.8** | 36.6 |
| | ***32*** | **209.4** | 1.7 | **256.0** | 2.5 | **1133.1** | 140.7 | **1850.4** | 243.3 |
| | ***64*** | **218.1** | 1.1 | **260.2** | 2.2 | **1173.8** | 135.6 | **2020.4** | 221.6 |
| | ***128*** | **219.9** | 0.9 | **262.3** | 2.6 | **1146.8** | 37.0 | **2118.8** | 175.8 |
| | ***256*** | **220.5** | 1.0 | **262.0** | 2.8 | **1113.0** | 52.7 | **1917.9** | 116.2 |
| Xavier AGX | ***1*** | **24.2** | 0.3 | **34.6** | 0.4 | **94.8** | 5.6 | **125.6** | 9.3 |
| | ***32*** | **39.3** | 0.1 | **48.0** | 0.1 | **201.1** | 3.4 | **363.6** | 32.0 |
| | ***64*** | **40.2** | 0.1 | **51.4** | 0.1 | **211.4** | 3.4 | **367.3** | 35.2 |
| | ***128*** | **41.0** | 0.0 | **53.4** | 0.1 | **217.2** | 3.3 | **379.6** | 26.6 |
| | ***256*** | **41.6** | 0.0 | **54.3** | 0.0 | **217.1** | 2.7 | **386.6** | 30.9 |
| Orin Nano | ***1*** | **25.8** | 0.5 | **63.5** | 10.1 | **91.1** | 4.3 | **120.1** | 12.0 |
| | ***32*** | **66.8** | 0.3 | **167.1** | 1.0 | **327.5** | 14.8 | **485.0** | 119.4 |
| | ***64*** | **70.7** | 0.2 | **175.7** | 1.1 | **342.7** | 15.8 | **527.7** | 83.6 |
| | ***128*** | **71.9** | 0.2 | **180.2** | 1.1 | **349.2** | 14.1 | **569.3** | 52.6 |
| | ***256*** | **45.5** | 14.9 | **178.8** | 0.6 | **351.0** | 11.1 | **590.8** | 31.3 |
| Orin AGX | ***1*** | **29.3** | 0.9 | **55.2** | 12.4 | **98.3** | 8.1 | **124.2** | 53.4 |
| | ***32*** | **88.9** | 0.8 | **183.4** | 2.4 | **361.8** | 7.2 | **563.6** | 64.5 |
| | ***64*** | **95.3** | 1.1 | **188.8** | 3.8 | **375.2** | 10.2 | **582.6** | 68.4 |
| | ***128*** | **95.9** | 0.7 | **191.2** | 2.2 | **382.0** | 8.1 | **597.6** | 48.5 |
| | ***256*** | **98.4** | 0.3 | **186.2** | 1.4 | **382.1** | 5.6 | **608.5** | 35.4 |

Tabla B.6: Resultados de throughput para el modelo de red ResNet152.

| | | **Modelo Base** | | `TRT fp32` | | `TRT fp16` | | `TRT int8` | |
|---|---|---|---|---|---|---|---|---|---|
| | Batch size | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) | **inf/s** | ±(95 %) |
| RTX 3060 | ***1*** | **124.6** | 10.6 | **186.3** | 2.2 | **464.1** | 13.8 | **675.6** | 33.5 |
| | ***32*** | **238.5** | 0.3 | **443.8** | 1.0 | **1067.3** | 9.1 | **1842.0** | 25.1 |
| | ***64*** | **247.7** | 0.7 | **478.9** | 0.7 | **1124.0** | 7.2 | **1950.8** | 24.9 |
| | ***128*** | **251.7** | 0.5 | **487.5** | 0.7 | **1202.0** | 6.7 | **2019.9** | 14.6 |
| | ***256*** | **259.8** | 0.5 | **489.8** | 1.8 | **1225.7** | 10.2 | **2053.4** | 14.0 |
| RTX 2060 | ***1*** | **71.9** | 4.7 | **114.9** | 7.6 | **319.9** | 11.5 | **430.2** | 20.4 |
| | ***32*** | **147.6** | 0.9 | **177.7** | 1.4 | **806.1** | 71.4 | **1449.9** | 165.6 |
| | ***64*** | **153.8** | 0.6 | **181.3** | 1.2 | **833.0** | 43.3 | **1539.3** | 208.6 |
| | ***128*** | **155.0** | 0.4 | **183.0** | 0.9 | **821.3** | 22.5 | **1537.3** | 85.9 |
| | ***256*** | **155.2** | 0.5 | **182.9** | 1.0 | **828.5** | 20.5 | **1448.8** | 59.6 |
| Xavier AGX | ***1*** | **17.0** | 0.2 | **24.3** | 0.2 | **69.9** | 1.6 | **106.9** | 5.9 |
| | ***32*** | **27.7** | 0.0 | **33.4** | 0.1 | **140.2** | 1.0 | **264.6** | 8.8 |
| | ***64*** | **28.2** | 0.0 | **36.1** | 0.0 | **149.2** | 1.0 | **274.7** | 11.5 |
| | ***128*** | **28.8** | 0.0 | **37.6** | 0.0 | **153.4** | 0.8 | **283.3** | 8.8 |
| | ***256*** | **29.2** | 0.0 | **38.3** | 0.0 | **153.4** | 0.7 | **286.1** | 12.0 |
| Orin Nano | ***1*** | **17.8** | 0.3 | **59.2** | 4.1 | **68.2** | 12.7 | **88.6** | 4.8 |
| | ***32*** | **47.9** | 0.2 | **118.7** | 0.7 | **233.2** | 2.1 | **404.0** | 50.2 |
| | ***64*** | **50.6** | 0.1 | **124.5** | 0.8 | **243.3** | 1.6 | **426.6** | 40.5 |
| | ***128*** | **51.2** | 0.1 | **127.3** | 0.7 | **248.1** | 1.4 | **442.7** | 31.3 |
| | ***256*** | **36.5** | 12.2 | **125.9** | 0.4 | **248.2** | 1.0 | **452.1** | 19.5 |
| Orin AGX | ***1*** | **20.4** | 0.5 | **62.3** | 7.7 | **71.9** | 4.2 | **98.5** | 6.1 |
| | ***32*** | **64.0** | 0.5 | **130.3** | 1.8 | **245.4** | 5.9 | **405.8** | 11.1 |
| | ***64*** | **68.1** | 0.6 | **135.1** | 2.0 | **255.4** | 5.8 | **420.9** | 11.7 |
| | ***128*** | **68.3** | 0.4 | **137.2** | 1.5 | **258.2** | 4.4 | **429.2** | 11.0 |
| | ***256*** | **70.0** | 0.3 | **134.8** | 0.8 | **258.5** | 1.7 | **432.4** | 8.6 |

Apéndice C

# CONFIGURACIÓN QUE PUEDE AUMENTAR EL USO DE MEMORIA DURANTE LA INFERENCIA

Como se mencionó en el Capítulo 2, la TensorRT Application puede configurarse de diversas maneras según las necesidades del usuario. Una de las configuraciones que no se analizó en detalle, pero que se explora en el Capítulo 5, es la posibilidad de definir una dimensión dinámica para la altura y la anchura de las entradas del modelo. Esta opción permite establecer valores mínimos, óptimos y máximos para dichos parámetros.

Al generar un inference engine con la TensorRT Application configurada con una dimensión dinámica **mínima de cero**, TensorRT emite la siguiente advertencia:

> "IShuffle layer with zero is placeholder = True. May cause excessive memory consumption."

Esta advertencia indica que, cuando el inference engine se genera con una dimensión dinámica mínima de altura y anchura igual a cero, TensorRT lo interpreta como un parámetro provisional, es decir, un valor que puede variar dentro de un rango indefinido. Como consecuencia, el modelo resultante puede experimentar un consumo excesivo de memoria durante la inferencia.

Durante el proceso de evaluación descrito en la Sección 5.1.3, una de las pruebas consistió en medir el uso de memoria asociado a una configuración específica de operación: modelo `yolov8l-seg`, cuantización `fp16`, batch size de configuración dinámico, nivel de optimización por defecto y modo de consumo predeterminado en la plataforma Jetson Orin AGX. La Figura 5.3 muestra los resultados obtenidos para esta configuración utilizando una dimensión dinámica mínima de 360 píxeles.

Por otro lado, la Figura C.1 presenta los resultados del uso de memoria para la misma configuración, pero con un dimensión dinámica mínima igual a cero píxeles (**D.D. min. cero**). Estos resultados evidencian un incremento de más de 10 GB en el uso máximo y de 3 GB en el consumo promedio para un batch size de inferencia de 8, en comparación con la configuración que emplea una dimensión dinámica mínima de 360 píxeles (**D.D. min. 360**) en anchura y altura.

Es importante destacar que, a pesar del incremento en el uso de memoria, ninguna otra métrica de rendimiento evaluada en el Capítulo 5—latencia, throughput y propiedades del modelo—se vio afectada. El único impacto observado fue en el consumo de memoria durante la inferencia.

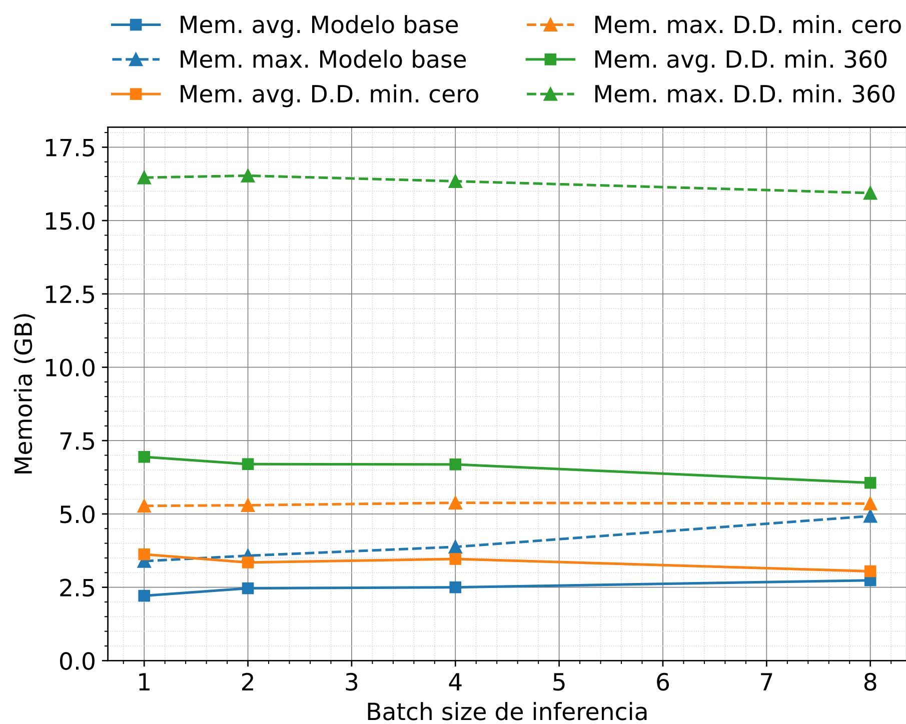



Fig. C.1: Uso de memoria promedio (ave.) y máxima (max.) en la plataforma Jetson Orin AGX durante la inferencia en el dataset de validación, para distintos tamaños de batch, comparando el uso de una dimensión dinámica mínima de 0 píxeles frente a una dimensión dinámica mínima de 360 píxeles y el modelo base.