ULPGC Universidad de Las Palmas de Gran Canaria | eite ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO DE FIN DE GRADO

Desarrollo e Implementación en FPGA de Redes Neuronales Convolucionales para Clasificación de Imágenes Satelitales

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación
Mención: Telemática
Autor: Álvaro Peñalver Valverde
Tutores: Dr. Roberto Sarmiento Rodríguez
D. Samuel Torres Fau

Fecha: julio de 2025

ULPGC Universidad de Las Palmas de Gran Canaria | eite ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO DE FIN DE GRADO

Desarrollo e Implementación en FPGA de Redes Neuronales Convolucionales para Clasificación de Imágenes Satelitales

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo:

Vocal

Fdo:

Secretario

Fdo:

Fecha: julio de 2025

Abstract

This thesis explores the deployment of a Convolutional Neural Network (CNN) on an Field-Programmable Gate Array (FPGA) platform for efficient, real-time processing of satellite imagery. The primary motivation is to reduce dependency on ground-based computation by enabling on-board inference, thereby minimizing data transfer and improving energy efficiency. However, this approach presents significant conceptual and hardware implementation challenges.

The development process began with the implementation of a test CNN model to validate the workflow. This involved training and quantizing the network, exporting it to the Quantized Open Neural Network Exchange format (QONNX) format, and generating a hardware-synthesizable IP core using High-Level Synthesis for Machine Learning (hls4ml), which was finally implemented onto the FPGA using Vitis HLS. Subsequently, a comprehensive preprocessing study was conducted to evaluate the impact of various input transformations on model performance and hardware efficiency. Based on these insights, an optimized final CNN architecture was selected and trained.

The resulting system demonstrates the feasibility of deploying deep learning models on reconfigurable hardware. The model was implemented on an FPGA, specifically the PYNQ-Z1 development board, highlighting its practical viability and potential for enabling efficient, on-board inference in space environments.

Resumen

Esta tesis explora el despliegue de una Red Neuronal Convolucional (CNN) en una plataforma FPGA para el procesamiento eficiente y en tiempo real de imágenes satelitales. La motivación principal es reducir la dependencia del procesamiento en tierra al habilitar inferencia a bordo, minimizando así la transferencia de datos y mejorando la eficiencia energética. Sin embargo, este enfoque presenta desafíos conceptuales y de implementación en hardware significativos.

El proceso de desarrollo comenzó con la implementación de un modelo de prueba CNN para validar el flujo de trabajo. Esto implicó entrenar y cuantizar la red, exportarla al estándar Formato Quantizado de ONNX (QONNX), y generar un núcleo IP sintetizable en hardware utilizando hls4ml, que finalmente fue implementado en la FPGA mediante Vitis HLS. Posteriormente, se llevó a cabo un estudio exhaustivo de preprocesamiento para evaluar el impacto de diversas transformaciones de entrada en el desempeño del modelo y la eficiencia del hardware. Basándose en estos resultados, se seleccionó y entrenó una arquitectura final optimizada de CNN.

El sistema resultante demuestra la viabilidad de desplegar modelos de aprendizaje profundo en hardware reconfigurable. El modelo fue implementado en una FPGA, específicamente en la placa de desarrollo PYNQ-Z1, destacando su viabilidad práctica y su potencial para permitir inferencia eficiente a bordo en entornos espaciales.

Índice

Índice de figuras

Índice de tablas

Lista de acrónimos

BOULPGC Boletín Oficial de la Universidad de las Palmas de Gran Canaria (ULPGC). 108

CNN Red Neuronal Convolucional. iii, 2, 5, 6, 9, 12, 14–16, 21, 59, 60, 62, 101, 116

COITT Colegio Oficial de Ingenieros Técnicos de Telecomunicaciones. 107, 111, 112

CPU Central Processing Unit. 2, 6

DL Deep Learning (Aprendizaje Profundo). 2–4, 8, 12, 21

FINN Framework for fast, quantized neural network inference on FPGAs. vi, 8, 23, 24

FPGA Field-Programmable Gate Array. i, iii, vi, 3, 4, 6, 8, 9, 11, 16–22, 24–26, 35, 59, 60, 71, 83, 84, 87, 93–95, 98, 101, 102, 105, 116, 117

GPU Graphics Processing Unit. 6

hls4ml High-Level Synthesis for Machine Learning. i, iii, vi, ix, 6, 8, 9, 21, 23, 24, 26, 39, 59, 90, 93, 101, 109

IGIC Impuesto General Indirecto Canario. 112

ONNX Open Neural Network Exchange. 39, 90, 93

QONNX Formato Quantizado de ONNX. iii, vii, ix, 9, 30, 39, 40, 44, 59, 90, 93, 96, 101

TFG Trabajo de Fin de Grado. 9, 107, 112

Capítulo 1

Introducción

En este capítulo se exploran los motivos subyacentes a la realización de este proyecto. Asimismo se incluyen las necesidades y objetivos que pretende cubrir, concluyendo con una breve descripción de la organización de este documento.

1.1. Contexto del problema

Las imágenes satelitales constituyen una fuente de datos esencial en un amplio abanico de disciplinas científicas y aplicaciones industriales. Son utilizadas de forma intensiva en ámbitos como la agricultura de precisión, la geología, la vigilancia medioambiental, el monitoreo de la degradación del suelo, los estudios oceánicos y atmosféricos, y la predicción meteorológica, entre muchos otros. Gracias a su elevada resolución espacial y cobertura global, permiten la observación continua de fenómenos naturales y antrópicos a diferentes escalas temporales y geográficas.

En los últimos años, los avances en el campo del Deep Learning (Aprendizaje Profundo) (DL), y en particular de las Redes Neuronales Convolucionales (CNNs), han revolucionado el análisis de este tipo de datos, mejorando significativamente el rendimiento en tareas como la clasificación, segmentación y detección de objetos en imágenes de teledetección. Estas redes han sido aplicadas con éxito en contextos como la detección automática de cultivos, la identificación de infraestructuras, la vigilancia marítima y la gestión del tráfico aéreo o terrestre.

Sin embargo, a pesar del éxito demostrado por las CNNs en el procesamiento de imágenes satelitales, su adopción directa en plataformas embebidas, como los sistemas a bordo de satélites, ha resultado compleja. Esto se debe, en gran parte, a la elevada demanda computacional que implican estos modelos. Las redes neuronales convolucionales modernas suelen estar compuestas por millones de parámetros y requieren una cantidad significativa de operaciones matemáticas, como convoluciones, activaciones no lineales y normalizaciones, que son intensivas en cómputo y en consumo de memoria.

Estas exigencias dificultan su despliegue eficiente en plataformas con recursos limitados, como Central Processing Units (CPUs) de baja potencia, especialmente en condiciones donde la energía disponible es reducida y el espacio físico es restringido. A esto se suman las restricciones del entorno espacial, donde la fiabilidad, la tolerancia a fallos por radiación, y la necesidad de funcionamiento autónomo durante largos periodos de tiempo sin intervención externa imponen requisitos adicionales.

En este contexto, las Field-Programmable Gate Array (FPGA) se presentan como una alternativa prometedora para realizar inferencia directamente a bordo, gracias a su capacidad de paralelización, bajo consumo energético y adaptabilidad. No obstante, desplegar modelos de DL sobre este tipo de hardware requiere superar importantes retos técnicos, como la creación de diseños eficientes tanto energéticamente como en los recursos hardware empleados, así como propiedades de tolerancia ante condiciones adversas propias del entorno espacial, como la radiación o las variaciones extremas de temperatura.

1.2. Motivación del proyecto

El presente proyecto se enmarca en la búsqueda de soluciones que permitan ejecutar modelos de DL directamente a bordo de plataformas espaciales, eliminando la necesidad de transmitir grandes volúmenes de datos a estaciones en tierra. Esta autonomía operativa no solo reduce significativamente el uso del ancho de banda, sino que también permite una respuesta más rápida ante eventos relevantes y una disminución del consumo energético global del sistema.

En el contexto espacial, los enlaces de comunicación entre el satélite y la estación terrestre suelen ser un recurso compartido entre distintos subsistemas a bordo (telemetría, control de actitud, comunicaciones, etc.), lo que limita la disponibilidad de ancho de banda exclusivo para la transmisión de imágenes. A esto se suma que las tasas de transferencia en entornos espaciales son considerablemente más bajas que las que se encuentran en redes terrestres, y están sujetas a restricciones temporales debido a las ventanas de visibilidad entre el satélite y las estaciones receptoras.

Además, las imágenes adquiridas por los sensores remotos suelen tener alta resolución y ocupar una cantidad considerable de memoria. En plataformas embarcadas, los recursos de almacenamiento y procesamiento son muy limitados, por lo que mantener dichas imágenes en memoria hasta su transmisión puede comprometer otros procesos críticos o incluso ser inviable. Por ello, resulta preferible realizar el procesamiento de datos directamente a bordo, procesando las imágenes de forma incremental o "on-the-fly", reduciendo su tamaño o extrayendo información relevante antes de enviarla a tierra.

Este enfoque de inferencia a bordo permite reducir drásticamente el volumen de datos que deben ser transmitidos, priorizar únicamente aquellos resultados de interés (por ejemplo, la detección de un barco o evento anómalo), y habilita una mayor autonomía del satélite, haciendo posible que actúe de forma más inteligente e independiente en misiones críticas.

Para alcanzar este objetivo, se plantea el uso de FPGAs como plataforma de implementación, aprovechando su naturaleza reconfigurable y su capacidad de ejecutar

operaciones en paralelo con un consumo reducido. La motivación central de este trabajo radica, por tanto, en explorar y demostrar la viabilidad de integrar arquitecturas optimizadas de CNNs en dispositivos hardware limitados, haciendo frente a las restricciones inherentes al entorno espacial y proponiendo una alternativa realista y eficiente al procesamiento tradicional en tierra.

1.3. Objetivos generales y específicos

El objetivo general de este proyecto es implementar una arquitectura optimizada de CNN para su ejecución eficiente en una FPGA, orientada al procesamiento en tiempo real de imágenes satelitales.

Para alcanzar este objetivo, se plantean los siguientes objetivos específicos:

- **O1.** Analizar el estado del arte de las arquitecturas CNN aplicadas a imágenes satelitales y su viabilidad en entornos con restricciones computacionales.

- **O2.** Diseñar una arquitectura CNN considerando las limitaciones de hardware y buscando un balance óptimo entre precisión y complejidad del modelo.

- **O3.** Optimizar y cuantizar el modelo para reducir su complejidad y tamaño sin comprometer el desempeño.

- **O4.** Generar un núcleo IP acelerador para FPGA mediante herramientas de síntesis de alto nivel, como High-Level Synthesis for Machine Learning (hls4ml).

- **O5.** Desplegar y validar el sistema en una plataforma FPGA, evaluando desempeño, consumo y precisión, y comparando con soluciones basadas en CPUs y Graphics Processing Units (GPUs).

Además, es importante señalar que el desarrollo del modelo se ha llevado a cabo bajo ciertas restricciones, impuestas por la complejidad del problema y la limitada duración del proyecto.

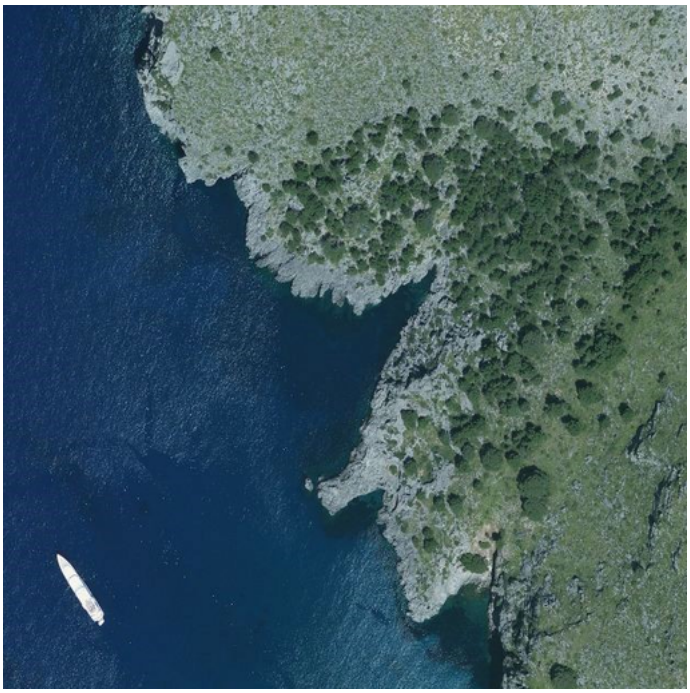


Figura 1.1: Ejemplo de imagen utilizada.

1.4. Antecedentes

La implementación de modelos de DL en FPGAs ha sido objeto de creciente interés en la comunidad científica y tecnológica, especialmente debido a la necesidad de ejecutar inferencia en tiempo real bajo restricciones de recursos. En este contexto, se han desarrollado múltiples herramientas que permiten traducir modelos entrenados en entornos *software* como `PyTorch` a arquitecturas optimizadas para *hardware*.

Entre las herramientas más destacadas se encuentran `hls4ml` y `Framework for fast, quantized neural network inference on FPGAs (FINN)`. hls4ml, desarrollado originalmente por el CERN en colaboración con diversas universidades, permite convertir modelos en redes neuronales a código `C++` y `VHDL`, facilitando su síntesis mediante herramientas como Vitis hls. FINN, por su parte, es una plataforma desarrollada por Xilinx enfocada en la implementación de redes neuronales cuantizadas, optimizadas para ejecutarse con gran eficiencia energética y mínimo uso de recursos lógicos.

Además de estas herramientas, existen otros frameworks industriales como `Vitis AI` (de AMD/Xilinx) o `Intel OpenVINO`, que proporcionan flujos de trabajo integrados para la aceleración de inferencia en dispositivos heterogéneos. Sin embargo, estas soluciones suelen requerir arquitecturas y formatos de modelos más rígidos, o bien están más orientadas a entornos comerciales cerrados, lo que limita su adaptabilidad en proyectos de investigación o entornos académicos.

En este trabajo se optó por el uso de hls4ml por varias razones fundamentales: su naturaleza *open-source*, su integración directa con flujos de trabajo basados en `PyTorch` y `ONNX`, su capacidad para manipular directamente la precisión de cada capa y, especialmente, su enfoque explícito en la reducción de latencia. Esto permite generar aceleradores altamente personalizables y eficientes para aplicaciones embebidas como las que se desarrollan en el contexto espacial.

Finalmente, una parte significativa de los conocimientos aplicados en este proyecto tiene su origen en el trabajo desarrollado durante el curso académico anterior, en el marco de las prácticas externas realizadas en la empresa Thales Alenia Space España,

S.A., en la cual también se enmarca el presente Trabajo de Fin de Grado (TFG). Durante dicho periodo se abordó el diseño y evaluación de una arquitectura basada en redes neuronales convolucionales (CNN) para la clasificación de imágenes satelitales. Aunque en aquel contexto no se implementó la inferencia sobre FPGA ni se utilizaron herramientas específicas como hls4ml, el trabajo permitió adquirir una base sólida en diseño de modelos, flujo de entrenamiento y técnicas de optimización, sirviendo como punto de partida comparativo durante las fases iniciales del proyecto actual.

No obstante, el origen conceptual de ambos trabajos puede trazarse aún más atrás, concretamente al proyecto de investigación publicado en *"FPGA-Based Implementation of a CNN Architecture for the On-Board Processing of Very High-Resolution Remote Sensing Images"* [1]. En dicho trabajo se exploraba la posibilidad de implementar modelos de aprendizaje profundo sobre plataformas reconfigurables para procesar imágenes de teledetección directamente a bordo del satélite. A pesar de tratarse de un estudio novedoso, presentaba ciertas limitaciones en cuanto a escalabilidad, eficiencia de los modelos utilizados y adaptabilidad del flujo de diseño.

Tanto las prácticas externas como el presente TFG pueden entenderse como una evolución directa de dicha investigación, proponiendo mejoras en múltiples aspectos; y empleando un flujo de diseño más moderno y automatizado, integrando herramientas como Brevitas, Formato Quantizado de ONNX (QONNX) y hls4ml, que permiten adaptar los modelos a plataformas embebidas de forma más eficiente y flexible.

Capítulo 2

Análisis del estado del arte

En este capítulo se exponen los conceptos clave que sustentan el desarrollo del presente trabajo, organizados en tres grandes bloques: redes neuronales convolucionales y su aplicación a visión por computador, la implementación de dichas arquitecturas sobre plataformas hardware reconfigurables como las FPGAs, y los principales frameworks que permiten su síntesis a nivel hardware mediante técnicas de alto nivel. Esta revisión proporciona el contexto técnico necesario para comprender tanto las decisiones de diseño como las herramientas utilizadas en este proyecto.

2.1. Redes neuronales convolucionales (CNN) en visión por computador

Las **redes neuronales convolucionales** (CNNs, por sus siglas en inglés) constituyen una clase especializada de modelos de DL diseñados para procesar datos con estructura espacial, como imágenes, vídeo o señales multicanal. Inspiradas en el funcionamiento del sistema visual humano, estas redes son capaces de aprender automáticamente patrones complejos como bordes, formas o texturas sin necesidad de que estos se definan explícitamente. Lo que las ha consolidado como uno de los tipos de arquitecturas más utilizadas en tareas de visión por computador, como clasificación de imágenes, detección de objetos, segmentación semántica o reconocimiento facial [2], [3].

2.1.1. Estructura básica de una CNN

Una CNN está compuesta por una serie de capas organizadas en secuencia, cada una de las cuales transforma los datos de entrada (por ejemplo, una imagen) en representaciones progresivamente más abstractas. A continuación se describen las capas más comunes que conforman una CNN típica:

- **Capa convolucional** (`Conv2D`): Es el núcleo de una CNN. Su función es aplicar una serie de filtros (también llamados *kernels*) sobre la imagen de entrada. Cada filtro es una pequeña "ventana" que se desplaza por la imagen, realizando una operación matemática llamada *convolución*. Estos filtros detectan patrones locales como bordes, líneas o texturas. Lo más importante es que los valores de estos filtros no se definen manualmente, sino que se aprenden automáticamente durante el proceso de entrenamiento.
- **Función de activación no lineal** (`ReLU`, siglas de *Rectified Linear Unit*): Después de cada convolución se suele aplicar una función no lineal para introducir complejidad en el modelo. La más común es `ReLU`, que simplemente reemplaza los valores negativos por cero. Esto ayuda al modelo a aprender representaciones más útiles.
- **Operaciones de reducción espacial (pooling)**: Estas capas se usan para

reducir la resolución de los mapas de activación generados por las capas anteriores. Los dos tipos más comunes son:

- **MaxPooling**: toma el valor máximo dentro de una región local.
- **AveragePooling**: calcula el promedio de los valores en esa región.

Al reducir la cantidad de datos, estas capas ayudan a disminuir el coste computacional del modelo y aumentan su robustez frente a pequeñas variaciones o desplazamientos en las imágenes (por ejemplo, si el objeto se mueve ligeramente en la imagen).

- **Capas completamente conectadas (Fully Connected / Dense)**: Al final de la red, los datos se transforman en un vector unidimensional y se introducen en una o varias capas totalmente conectadas. Estas capas actúan como "clasificadores", combinando toda la información extraída por las capas anteriores para generar la salida final, como por ejemplo la clase de un objeto en una imagen.
- **Capa de salida (Output)**: En tareas de clasificación multiclase, se emplea habitualmente una función de activación `Softmax`, que convierte los valores de salida en probabilidades asociadas a cada clase. En cambio, cuando se trata de clasificación binaria, se utiliza normalmente una `Sigmoide`, que transforma la salida en un valor entre 0 y 1, interpretado como probabilidad de pertenencia a la clase positiva.

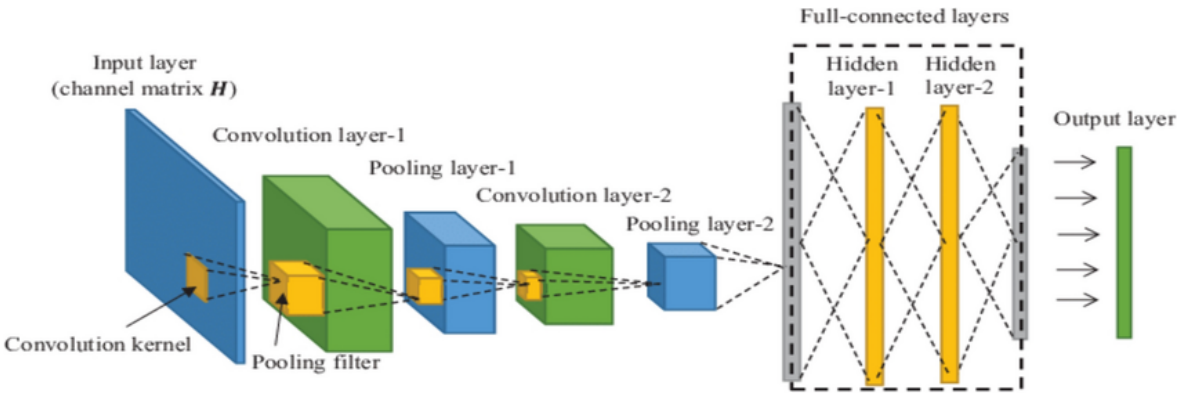


Figura 2.1: Esquema general del flujo de datos en una CNN.

En conjunto, las capas anteriores permiten que una CNN tome como entrada una imagen y, a través de múltiples transformaciones intermedias, produzca una predicción

precisa sobre su contenido. A medida que los datos fluyen a través de la red, las representaciones internas se hacen cada vez más abstractas: de bordes a formas, y de formas a objetos complejos.

Importancia del aprendizaje

Todas las operaciones descritas en una CNN dependen de un proceso previo conocido como entrenamiento, el cual pertenece al paradigma del aprendizaje supervisado. En este enfoque, se proporciona al modelo una gran cantidad de ejemplos (imágenes de entrada) junto con sus correspondientes etiquetas (la clase a la que pertenece cada imagen). El objetivo del entrenamiento es que la red aprenda, a través de múltiples iteraciones, los patrones que relacionan los datos de entrada con su salida deseada.

Durante este proceso, el modelo ajusta automáticamente parámetros internos (como los valores de los filtros en las capas convolucionales) con el fin de minimizar el error entre su predicción y la etiqueta real. Este ajuste se realiza mediante algoritmos de optimización como `Adam` o `SGD`, aplicando la técnica de retropropagación (*backpropagation*).

Sin embargo, el rendimiento final del modelo no solo depende de su arquitectura o del algoritmo de entrenamiento, sino especialmente de la calidad del conjunto de datos. Es fundamental contar con:

- **Un volumen suficientemente grande** de ejemplos, para que el modelo generalice correctamente y no simplemente memorice los datos vistos.
- **Etiquetas precisas y coherentes**, ya que el modelo aprende directamente a partir de ellas. Errores en el etiquetado pueden inducir al modelo a aprender relaciones incorrectas.
- **Diversidad en las muestras**, que incluya diferentes condiciones (iluminación, ángulos, ruido, variaciones contextuales) para que el modelo sea robusto y aplicable en escenarios reales.

2.1.2. Aplicaciones en imágenes satelitales

El uso de redes neuronales convolucionales (CNNs) ha transformado significativamente el campo de la **teledetección**, permitiendo extraer de manera automática información de alto nivel a partir de imágenes satelitales. A diferencia de los enfoques clásicos, que requerían el diseño manual de características específicas (*feature engineering*), las CNNs permiten aprender directamente representaciones útiles de los datos mediante entrenamiento supervisado.

Las aplicaciones más comunes de las CNNs en imágenes satelitales incluyen:

- **Clasificación de uso del suelo**: asignar a cada píxel o región de la imagen una categoría semántica (urbano, agrícola, bosque, agua, etc.). Modelos como U-Net o DeepLab han sido ampliamente utilizados para segmentación semántica multiclase en imágenes multiespectrales [4], [5].
- **Detección de objetos**: localizar y clasificar objetos específicos en las imágenes, como embarcaciones, vehículos, edificios o aeronaves. Trabajos como [6] y [7] han utilizado arquitecturas como Faster R-CNN, YOLO o RetinaNet adaptadas a resolución satelital para este tipo de tareas.
- **Monitorización ambiental**: detectar cambios en la cobertura del terreno, deforestación, evolución de masas de agua, crecimiento urbano o propagación de incendios forestales. En [8], se destaca el uso de redes neuronales para cambio temporal utilizando series multitemporales de imágenes.
- **Mapeo de desastres**: detección rápida de daños por inundaciones, terremotos, incendios o tormentas. Modelos entrenados con datos de satélites como Sentinel-1 y Sentinel-2 han sido utilizados para analizar áreas afectadas en tiempo casi real [9].
- **Estimación de cultivos y productividad agrícola**: utilizando modelos CNN y datos satelitales multiespectrales, se ha podido estimar el tipo de cultivo, su salud y su rendimiento potencial. En [10], se combina Sentinel-2 con modelos CNN para clasificar variedades de vegetación.

Además, las CNNs también se han integrado en sistemas híbridos con redes recurrentes (RNNs) o transformers para análisis multitemporal, especialmente útil en agricultura o seguimiento de cambios estacionales [11].

2.2. Implementación de redes neuronales en FPGAs

Pese a su efectividad, los modelos basados en redes neuronales convolucionales (CNNs) requieren una considerable cantidad de operaciones aritméticas y memoria, lo que puede ser prohibitivo en dispositivos embebidos con recursos limitados, como los sistemas que se utilizan en el espacio o en dispositivos autónomos. Ejecutar redes neuronales directamente en estos sistemas demanda arquitecturas especializadas, que combinen eficiencia energética, paralelismo y flexibilidad. Es en este contexto donde entran en juego las FPGAs.

2.2.1. ¿Qué es una FPGA?

Una Field-Programmable Gate Array (FPGA) es un tipo de circuito integrado reconfigurable que permite al usuario programar su funcionamiento después de la fabricación. A diferencia de un procesador tradicional (CPU) o de una tarjeta gráfica (GPU), que siguen un conjunto fijo de instrucciones, una FPGA puede ser configurada para implementar cualquier circuito lógico deseado, utilizando bloques básicos de hardware como compuertas lógicas, registros, memorias, multiplexores, etc.

En términos simples, una FPGA es una "hoja en blanco"de hardware digital que puede adoptar cualquier funcionalidad deseada: desde un simple sumador hasta un sistema complejo de procesamiento de señales o una red neuronal entera.

Características principales de una FPGA:

- **Reconfigurable**: A diferencia de los circuitos integrados tradicionales (ASICs), cuya funcionalidad queda fijada durante la fabricación, una FPGA puede ser reprogramada tras su despliegue. Esto permite modificar su comportamiento tantas veces como sea necesario, lo que resulta especialmente útil en entornos de prototipado, evolución de producto, o incluso en misiones espaciales donde el software de control de hardware puede actualizarse en vuelo.
- **Altamente paralela**: Una FPGA está compuesta por miles de bloques lógicos configurables, que pueden operar de forma simultánea. Esto permite diseñar arquitecturas que ejecuten muchas operaciones en paralelo, lo cual es ideal para tareas que requieren gran velocidad de procesamiento, como el procesamiento de vídeo en tiempo real, comunicaciones de alta velocidad, o inferencia en redes neuronales convolucionales.
- **Determinista**: En una FPGA, el diseñador tiene control total sobre la temporización del sistema. A diferencia de los procesadores tradicionales que dependen de sistemas operativos, interrupciones o planificación dinámica, una FPGA puede garantizar tiempos de respuesta estrictamente definidos, lo que es crucial en sistemas embebidos críticos como control industrial, aeroespacial o automoción.
- **Flexible**: El mismo dispositivo puede ser reutilizado para múltiples tareas simplemente cambiando su configuración. Esta flexibilidad permite que una misma FPGA sirva como controlador de comunicaciones, procesador de señales, acelerador de inteligencia artificial o interfaz multimedia, según las necesidades del proyecto. Además, se pueden integrar en ella tanto funciones personalizadas como interfaces estándar (por ejemplo, AXI, SPI, UART).

Usos comunes de las FPGAs

Históricamente, las FPGAs se han utilizado en sectores donde la velocidad, la precisión y la personalización del hardware son críticos, como:

- **Electrónica aeroespacial y defensa**: Las FPGAs se utilizan en sistemas de navegación, comunicaciones seguras y procesamiento en tiempo real en entornos extremos, donde la fiabilidad y la tolerancia a fallos son esenciales.
- **Telecomunicaciones y redes de alta velocidad**: Gracias a su capacidad para manejar protocolos personalizados y su baja latencia, las FPGAs son fundamentales en conmutadores, routers y estaciones base 5G.
- **Procesamiento de señales digitales (audio, imagen, radar)**: Las FPGAs permiten implementar filtros digitales, compresores de vídeo o analizadores espectrales con una latencia mínima, en aplicaciones como cámaras industriales o radares de automoción.
- **Control industrial y sistemas embebidos de automoción**: La posibilidad de diseñar lógica específica permite implementar sistemas de control en tiempo real, detección de fallos y controladores de motor de forma optimizada.
- **Aceleración de inteligencia artificial e inferencia de modelos**: Gracias a su capacidad de paralelismo y control sobre la aritmética, las FPGAs se utilizan para acelerar redes neuronales, especialmente modelos compactos y cuantizados.
- **High-Frequency Trading (HFT)**: En los mercados financieros, donde cada microsegundo cuenta, las FPGAs permiten ejecutar estrategias de compra-venta directamente en hardware, minimizando la latencia frente a soluciones basadas en CPU o GPU.

En los últimos años, gracias al aumento de capacidad y la mejora de herramientas de desarrollo, las FPGAs también se han posicionado como una solución atractiva para el despliegue de modelos de `machine learning`, en aplicaciones donde se requiere

una inferencia rápida y eficiente sin depender de la nube. O bien se quiere introducir arquitecturas personalizadas con propósitos específicos (I.E. ultra-low power, delays mínimos, etc). Los aceleradores típicos, como GPUs o TPUs, a menudo no pueden ser empleados en estos contextos.

2.2.2. Ventajas de las FPGAs en entornos embebidos

Las FPGAs permiten crear arquitecturas hardware personalizadas, ajustadas a las necesidades exactas de un modelo de red neuronal. Esto permite optimizar la ejecución de las redes en sistemas con recursos limitados, como satélites, drones o sensores inteligentes. Algunas de sus ventajas más destacadas incluyen:

- **Paralelismo masivo**: las operaciones aritméticas de una red neuronal (por ejemplo, multiplicaciones y sumas en una convolución) pueden implementarse en paralelo a nivel de hardware, permitiendo una ejecución mucho más rápida que en una CPU.
- **Bajo consumo energético**: en comparación con GPUs, las FPGAs consumen menos energía, lo cual es esencial en sistemas alimentados por batería o con restricciones térmicas.
- **Flexibilidad arquitectónica**: al ser reconfigurables, las FPGAs permiten modificar la arquitectura hardware sin necesidad de cambiar el dispositivo físico, facilitando la actualización del modelo o la adaptación a diferentes tareas.
- **Soporte para precisión reducida**: muchas FPGAs están optimizadas para trabajar con números en punto fijo o enteros, que son más eficientes en hardware que los números en coma flotante. Esto hace que la cuantización de modelos (por ejemplo, a 8 o 4 bits) sea especialmente adecuada para este tipo de dispositivos.
- **Tolerancia a fallos y radiación**: en entornos hostiles como el espacio, las FPGAs pueden configurarse para soportar errores provocados por radiación cósmica mediante técnicas como **TMR** (*Triple Modular Redundancy*), replicando bloques lógicos críticos y votando su salida. Además, existen FPGAs que están

preparadas para soportar radiación por diseño. Esto permite construir sistemas resilientes sin componentes adicionales.

2.2.3. Limitaciones y desafíos

No obstante, la utilización de FPGAs en tareas de inteligencia artificial también plantea ciertos desafíos:

- **Mayor complejidad de desarrollo**: a diferencia del desarrollo de software tradicional, programar FPGAs requiere conocimientos de diseño digital y herramientas de síntesis de alto nivel (HLS) o lenguajes de descripción de hardware como VHDL o Verilog.
- **Tiempo de síntesis**: cualquier cambio en el diseño requiere un proceso de compilación (síntesis) que puede llevar desde varios minutos hasta horas, dependiendo de la complejidad del sistema.
- **Limitaciones de recursos físicos**: a pesar de su flexibilidad, las FPGAs tienen una cantidad finita de bloques lógicos, memorias y multiplicadores (DSPs), lo que puede restringir el tamaño y la complejidad de los modelos a implementar.
- **Conversión no trivial de modelos**: la conversión de un modelo de red neuronal entrenado (por ejemplo, en PyTorch) a una descripción hardware requiere procesos adicionales como cuantización, plegado de capas y adaptación de formato, que no siempre son directos.

En resumen, las FPGAs representan una excelente plataforma para ejecutar modelos de inferencia de forma eficiente en términos de consumo y latencia, especialmente cuando se requiere flexibilidad y personalización a nivel hardware. Esta combinación de características las convierte en una opción muy adecuada para aplicaciones espaciales y embarcadas, como la que se plantea en el presente proyecto.

2.3. Herramientas de desarrollo y conversión de modelos hacia hardware

Para facilitar el desarrollo de aceleradores de CNNs en hardware, se han desarrollado herramientas que permiten la conversión de modelos de alto nivel (entrenados en software) a implementaciones optimizadas en hardware. Estas herramientas automatizan gran parte del proceso de traducción, optimización y síntesis, reduciendo la necesidad de diseñar arquitecturas hardware desde cero.

En este proyecto se emplearon múltiples herramientas clave:

- **PyTorch**: para definir y entrenar modelos de redes neuronales.
- **Brevitas**: para cuantizar dichos modelos y prepararlos para hardware.
- **QONNX**: para exportar los modelos cuantizados a un formato intermedio.
- **hls4ml**: para convertir el modelo QONNX en una arquitectura hardware optimizada.
- **Vitis HLS**: como backend para la síntesis y evaluación del diseño en FPGA.

A continuación se describen estas herramientas en mayor detalle, junto con otras alternativas existentes.

2.3.1. PyTorch: definición y entrenamiento de modelos

`PyTorch` es una de las librerías más populares para el desarrollo de modelos de DL. Desarrollada inicialmente por Meta AI (anteriormente Facebook AI Research), ofrece una interfaz basada en Python altamente expresiva y flexible [12]. Entre sus características más relevantes se encuentran:

- **Definición dinámica de grafos computacionales** (*define-by-run*): permite construir el grafo de operaciones de forma dinámica durante la ejecución, lo que

facilita la depuración, el diseño flexible de modelos y la experimentación con arquitecturas no convencionales.

- **Sistema de diferenciación automática (`autograd`)**: permite calcular gradientes de forma automática a través de todo el grafo computacional, sin necesidad de derivar manualmente cada operación. Esta característica es clave para entrenar modelos mediante retropropagación.
- **Aceleración por GPU** mediante CUDA: permite que los modelos se ejecuten eficientemente sobre tarjetas gráficas, acelerando el entrenamiento y la inferencia.
- **Amplio ecosistema**: PyTorch cuenta con extensiones oficiales como `TorchVision` (visión por computador), `TorchAudio` (procesamiento de audio), `TorchText`, y utilidades para exportación como `TorchScript`.

2.3.2. Brevitas: cuantización de modelos en PyTorch

`Brevitas` es una librería open-source desarrollada por Xilinx Research Labs para facilitar la cuantización de redes neuronales dentro del ecosistema PyTorch [13]. Su objetivo es habilitar flujos de trabajo para la inferencia eficiente en hardware (FPGAs, ASICs), permitiendo definir redes con pesos y activaciones en baja precisión desde el inicio del entrenamiento.

Entre sus funcionalidades:

- **Amplio soporte de tipos de cuantización**, incluyendo configuraciones de punto fijo con distintas anchuras de palabra y niveles de precisión.
- **Capas cuantizadas equivalentes a PyTorch**: `QuantConv` es funcionalmente equivalente a una capa `Conv2D`, pero añade soporte para la cuantización de pesos, activaciones y bias. Del mismo modo, se definen versiones cuantizadas de otras capas comunes como `QuantLinear`, `QuantReLU`, entre otras.
- **Soporte completo para Quantization-Aware Training (QAT)**, introduciendo los efectos de la cuantización durante el propio entrenamiento. Esto permite

minimizar la pérdida de precisión al trasladar el modelo desde software a hardware, a diferencia de la cuantización post-entrenamiento.

- **Conversión nativa a QONNX**, lo que permite continuar el flujo de conversión hacia herramientas como `HLS4ML` o `FINN`.

2.3.3. QONNX: formato intermedio con soporte de cuantización

`ONNX` (*Open Neural Network Exchange*) es un formato estándar abierto diseñado para facilitar la interoperabilidad entre distintos frameworks de aprendizaje automático como PyTorch, TensorFlow, Keras o scikit-learn [14]. Permite exportar un modelo entrenado en un framework determinado y reutilizarlo en otro sin necesidad de redefinirlo o reentrenarlo desde cero. La representación se realiza mediante un grafo computacional, donde cada nodo corresponde a una operación (como convolución, activación o multiplicación), y los tensores fluyen entre dichos nodos.

`QONNX` es una extensión del formato ONNX desarrollada por Xilinx en el marco del proyecto `FINN` [15], orientada a incorporar soporte explícito para primitivas de cuantización. Mientras que ONNX permite describir modelos en punto flotante o representar la cuantización de forma indirecta mediante metadatos, QONNX introduce operadores específicos (como `QuantizeLinear` o `Trunc`) dentro del propio grafo, lo que permite conservar toda la lógica de cuantización de forma explícita.

Gracias a esta representación estructurada, herramientas como `hls4ml` o `FINN` pueden interpretar de forma precisa las configuraciones de cada capa: ancho de bit, escala, signo, y otros parámetros críticos para una implementación eficiente en hardware.

2.3.4. hls4ml

`hls4ml` (High-Level Synthesis for Machine Learning) es un framework de código abierto impulsado por el CERN, Fermilab, el MIT y otras instituciones académicas. Está diseñado para convertir modelos entrenados en redes neuronales en código en C++

sintetizable a nivel hardware, facilitando su implementación sobre plataformas como FPGAs [16], [17].

Una de sus principales ventajas es su capacidad para importar modelos directamente en formato `QONNX`, conservando la cuantización aplicada en herramientas como Brevitas. Además, permite generar núcleos IP sintetizables mediante herramientas como `Vitis HLS` o `Vivado`, ofreciendo compatibilidad total con el ecosistema de desarrollo de AMD/Xilinx.

2.3.5. FINN

`FINN` es otro framework de Xilinx orientado a la implementación de modelos cuantizados extremos (hasta 1 bit) sobre FPGAs. Su enfoque se basa en la construcción de arquitecturas mediante bloques funcionales IP, altamente optimizados para baja latencia y eficiencia energética [15].

A diferencia de hls4ml, FINN está más enfocado en aplicaciones comerciales y rendimiento extremo, pero presenta una curva de aprendizaje más pronunciada. Por esa razón, en este trabajo se optó por hls4ml.

2.3.6. Vitis HLS y Vivado

`Vitis HLS` es la herramienta de síntesis de alto nivel (HLS, por sus siglas en inglés) ofrecida por AMD/Xilinx para traducir código en lenguajes como C o C++ a descripciones hardware en RTL (Register Transfer Level), que pueden implementarse en FPGAs. A diferencia del enfoque tradicional basado en lenguajes como VHDL o Verilog, HLS permite a ingenieros y científicos diseñar aceleradores hardware utilizando lenguajes de programación de más alto nivel, facilitando la adopción de FPGAs por parte de comunidades no especializadas en diseño electrónico.

Las funcionalidades principales de `Vitis HLS` incluyen:

- **Traducción automática de C/C++ a HDL (VHDL o Verilog)**.
- **Análisis y optimización de rendimiento**: permite aplicar directivas para ajustar el paralelismo (`unroll`), canalización (`pipeline`) o partición de arrays.
- **Generación de informes de síntesis**: incluyendo latencia, uso de recursos (LUTs, BRAMs, DSPs), y frecuencia estimada.
- **Exportación como bloque IP**: se genera un núcleo que puede ser fácilmente integrado en el sistema completo.

Por su parte, `Vivado Design Suite` es el entorno integral de desarrollo de hardware para dispositivos de AMD/Xilinx. Incluye herramientas como:

- **Diseño de lógica programable** mediante esquemáticos o RTL.
- **Integración de bloques IP**, incluyendo IP personalizados generados con `Vitis HLS`.
- **Diseño de sistemas embebidos** completos, combinando lógica programable (PL) con procesadores embebidos (PS).
- **Generación del bitstream** necesario para programar la FPGA.

Vivado permite, por tanto, integrar el núcleo de inferencia generado con HLS4ML en un sistema completo de procesamiento, conectándolo mediante buses `AXI` a módulos como el `AXI-DMA`, controladores de memoria o el procesador embebido. Este enfoque permitió una implementación eficiente, reproducible y controlada del modelo en hardware, sin necesidad de escribir manualmente ningún código en HDL.

2.3.7. Otros frameworks: Vitis AI y OpenVINO

- **Vitis AI** [18] es la solución de AMD/Xilinx para ejecutar modelos de aprendizaje profundo en dispositivos con DPU (Deep Learning Processing Unit), aceleradores específicos presentes en FPGAs como las series Zynq UltraScale+ MPSoC.

- **OpenVINO** [19] es el framework equivalente de Intel, optimizado para CPUs, VPUs (Myriad) y FPGAs. Soporta modelos de ONNX, TensorFlow o PyTorch, ofreciendo herramientas de optimización post-entrenamiento y ejecución heterogénea.

Ambos frameworks son muy potentes, pero están más orientados a soluciones industriales cerradas.

2.3.8. Justificación del flujo de trabajo

El flujo de trabajo adoptado en este proyecto se basa en la combinación de `PyTorch + Brevitas + QONNX + hls4ml + Vitis HLS`. Esta elección se debe a los siguientes motivos:

- **Flujo totalmente open-source**, sin dependencias comerciales.
- **Compatibilidad total con cuantización y exportación desde Brevitas**.
- **Control preciso de recursos y paralelismo**, ideal para plataformas embebidas.
- **Documentación clara y soporte académico**, ampliamente adoptado en entornos de investigación.
- **Integración sencilla con herramientas de síntesis y diseño como Vivado**.

2.3.9. Placa de desarrollo empleada: PYNQ-Z1

La plataforma hardware seleccionada para la implementación final del modelo fue la `PYNQ-Z1`, una placa de desarrollo basada en el chip `Zynq-7020` de Xilinx. Esta placa combina una lógica programable (PL) tipo FPGA con un procesador ARM Cortex-A9 de doble núcleo (PS), permitiendo construir sistemas heterogéneos en los que conviven hardware reconfigurable y software embebido.

- **Procesador (PS)**: ARM Cortex-A9 dual-core a 650 MHz, capaz de ejecutar scripts Python, Linux embebido y aplicaciones de control.
- **Lógica programable (PL)**: FPGA Artix-7, con 53 200 LUTs, 106 400 Flip-Flops, 220 DSPs y 140 BRAMs.
- **Interfaces** integradas: HDMI, USB, Ethernet, GPIO, UART y dos conectores Pmod, ideales para prototipado rápido.
- **Entorno PYNQ**: distribuciones Linux adaptadas para la placa, que facilitan el desarrollo en Python y el acceso a hardware mediante APIs de alto nivel.

La `PYNQ-Z1` resulta una plataforma ideal para proyectos de investigación y desarrollo de bajo coste, ofreciendo un entorno accesible pero potente para la experimentación con aceleradores hardware, inferencia en tiempo real y sistemas embebidos. Su integración con herramientas como `Vivado`, `Vitis HLS` y `HLS4ML` la convierte en una opción muy adecuada para desplegar modelos cuantizados en entornos de computación heterogénea.

Capítulo 3

Diseño e Implementación del Modelo de Prueba

En este capítulo se describen en detalle las características de la arquitectura seleccionada para el modelo de prueba, el proceso de entrenamiento y evaluación, y los pasos seguidos hasta su despliegue en la FPGA. Los resultados obtenidos servirán como referencia inicial para posteriores fases de diseño y optimización del modelo final.

3.1. Objetivo del modelo de prueba

El objetivo principal de este modelo de prueba es validar la viabilidad del flujo de trabajo completo para la implementación de redes neuronales convolucionales cuantizadas en dispositivos FPGA utilizando las herramientas `Brevitas`, `Formato Quantizado de ONNX (QONNX)` y `HLS4ML`.

Para ello, se utiliza un modelo sencillo para la detección de dígitos del dataset MNIST, lo que permite centrarse en los aspectos técnicos del flujo sin la complejidad de una red profunda. Esta estrategia permite identificar posibles problemas en las etapas de cuantización, exportación, transformación y síntesis del modelo, sin que estos se vean enmascarados por la complejidad del diseño.

3.2. Dataset utilizado

Se ha utilizado el conjunto de datos MNIST, un estándar en problemas de clasificación de imágenes, compuesto por dígitos manuscritos del 0 al 9. Cada imagen tiene una resolución de 28x28 píxeles en escala de grises. El dataset está dividido en:

- Entrenamiento: 60,000 imágenes.
- Test: 10,000 imágenes.

Las imágenes son procesadas mediante los siguientes pasos:

- Redimensionado a 28x28.
- Conversión a escala de grises.

En la Figura 3.1 incluye algunas muestras típicas del dataset.

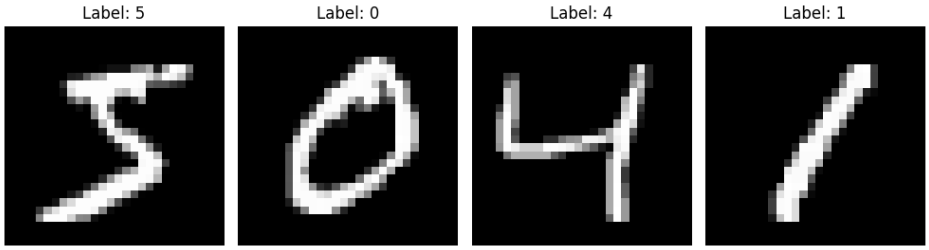


Figura 3.1: Muestras del dataset MNIST.

3.3. Arquitectura de red propuesta

La red neuronal propuesta se basa en una arquitectura clásica frecuentemente utilizada como ejemplo en la documentación de `PyTorch` para tareas de clasificación en el dataset `MNIST`. Esta arquitectura típica, compuesta por dos capas convolucionales seguidas de capas lineales con funciones de activación `ReLU` y `Dropout`, alcanza una precisión de hasta **96 %** cuando se entrena sin cuantización utilizando punto flotante de 32 bits.

La arquitectura aquí definida replica dicha estructura pero utilizando las capas equivalentes de la librería `Brevitas`, con todas las operaciones cuantizadas a `4 bits`, tanto en pesos como en activaciones.

Capa	**Descripción**
`QuantIdentity`	Cuantización de entrada a 4 bits
`QuantConv2d (1→10)`	Convolución con 10 filtros 5×5, pesos cuantizados a 4 bits
`QuantReLU`	Activación ReLU cuantizada (4 bits)
`MaxPool2d`	Submuestreo (pooling) 2×2
`QuantConv2d (10→20)`	Segunda convolución, también cuantizada a 4 bits
`QuantReLU`	Activación ReLU cuantizada (4 bits)
`MaxPool2d`	Submuestreo 2×2
`QuantLinear (320→50)`	Capa densa con pesos cuantizados a 4 bits
`QuantReLU`	Activación ReLU cuantizada (4 bits)
`Dropout`	Regularización (solo durante el entrenamiento)
`QuantLinear (50→10)`	Capa de salida con pesos cuantizados a 4 bits
`Softmax`	Normalización para clasificación multiclase

Tabla 3.1: Resumen de la arquitectura de red cuantizada basada en un modelo clásico de PyTorch

La Figura 3.2 muestra una visualización del modelo convertido a QONNX utilizando la herramienta `Netron`.

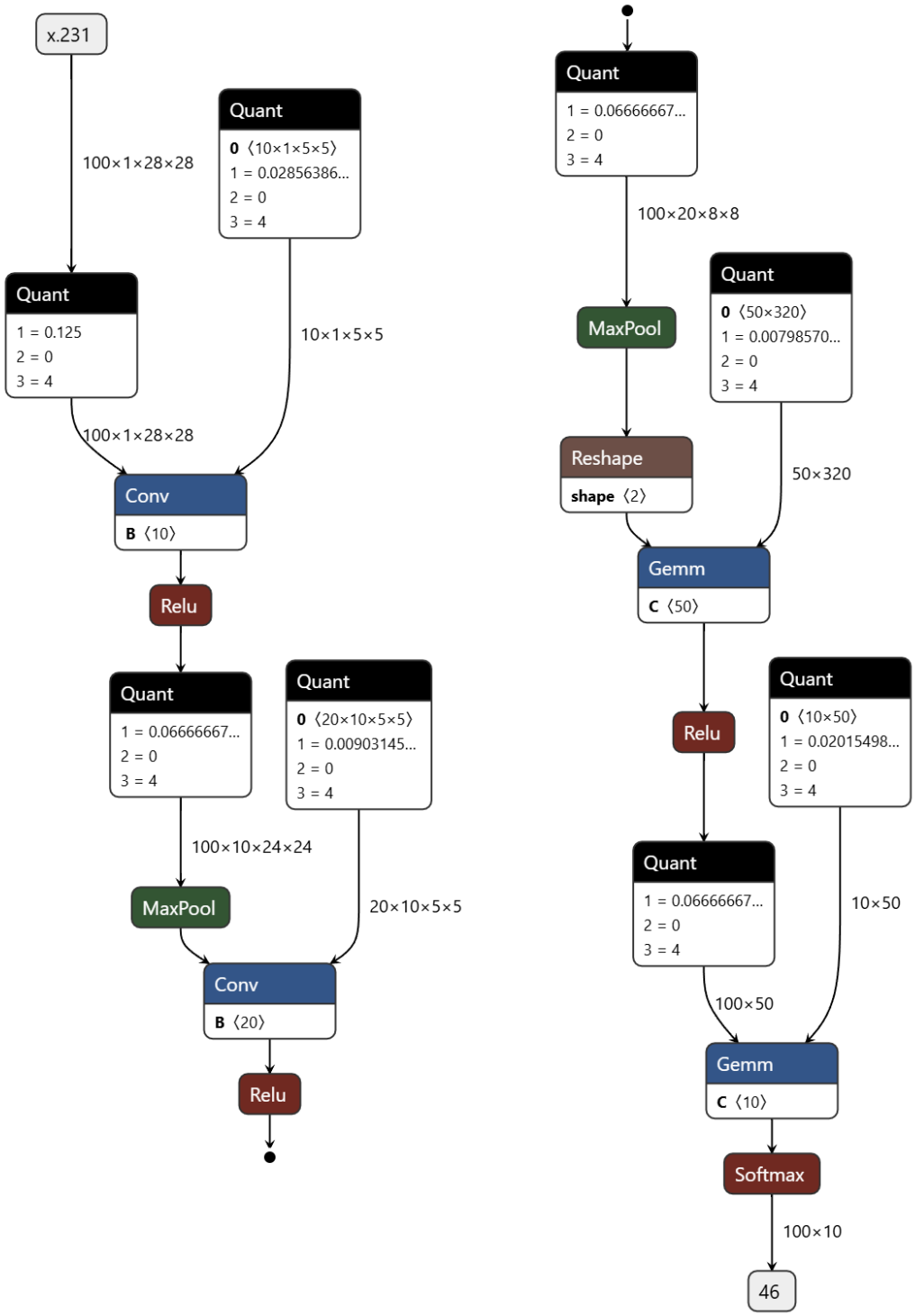


Figura 3.2: Visualización del modelo cuantizado en Netron

Capa Brevitas	**Operador QONNX**	**Descripción**
`QuantConv2d`	`Conv`	Convolución cuantizada, con pesos e input representados como tensores de 4 bits
`QuantReLU`	`Relu`	Aproximación a ReLU para tensores cuantizados, limitada por el rango definido
`MaxPool2d`	`MaxPool`	Operación estándar de submuestreo espacial
`QuantLinear`	`Gemm`	Multiplicación de matrices cuantizada, equivalente a una capa lineal
`Softmax`	`Softmax`	Capa final de clasificación

Tabla 3.2: Correspondencia entre capas Brevitas y operadores resultantes en el modelo QONNX

3.3.1. Motivación y beneficios de la cuantización a 4 bits

La cuantización reduce la representación numérica de pesos y activaciones de 32 bits en punto flotante a 4 bits en formato entero fijo. Esta reducción tiene múltiples ventajas, especialmente sobre sistemas embebidos o hardware especializado:

- **Reducción significativa del tamaño del modelo:** menor requerimiento de memoria para almacenamiento y transferencia. Este punto es crucial en modelos que pueden llegar a contener millones de parámetros.
- **Aceleración del procesamiento:** Los cálculos pueden ser paralelizados y acelerados de forma más efectiva debido a su menor complejidad lógica y eliminando la necesidad de unidades de coma flotante. En particular, las Field-Programmable Gate Arrays (FPGAs) están especialmente optimizadas para operaciones binarias y aritmética entera o de coma fija; la mayoría no dispone de bloques dedicados para operaciones en punto flotante, y su implementación en lógica programable suele implicar un elevado consumo de recursos. Además, muchos aceleradores modernos incluyen soporte específico para operaciones con 4 o 8 bits, lo que permite optimizar aún más el uso de recursos como memoria caché y buses internos, mejorando el rendimiento y la eficiencia energética del sistema.
- **Menor latencia:** El procesamiento con enteros de baja precisión suele ser más rápido, lo que reduce la latencia en tareas de inferencia, mejorando la capacidad de respuesta de aplicaciones en tiempo real.

Aunque la cuantización a INT4 puede introducir una ligera pérdida de precisión respecto a modelos en punto flotante, técnicas avanzadas de *quantization-aware training* y calibración permiten mantener un rendimiento competitivo.

3.4. Proceso de entrenamiento y evaluación

El modelo fue entrenado utilizando la librería `PyTorch` durante 10 épocas, haciendo uso del entorno de ejecución proporcionado por `Google Colab`, que permite el uso gratuito de GPU. El conjunto de datos utilizado fue `MNIST`, dividido en tres subconjuntos: entrenamiento, validación y prueba.

3.4.1. Configuración del entrenamiento

- **Optimizador: Adam**. Algoritmo de optimización adaptativa que ajusta dinámicamente la tasa de aprendizaje para cada parámetro, acelerando la convergencia del modelo.
- **Función de pérdida: CrossEntropyLoss**. Función común en tareas de clasificación multiclase, mide la discrepancia entre la distribución de salida del modelo y las etiquetas verdaderas.
- **Épocas: 15**. Número de veces que el modelo recorre completamente el conjunto de entrenamiento. Suficiente para lograr un rendimiento razonable sin sobreentrenar, dado el propósito demostrativo.
- **Batch size: 100**. Tamaño de los mini-lotes con los que se actualizan los pesos. Un valor moderado que equilibra velocidad de entrenamiento y estabilidad.
- **Dispositivo: GPU (T4)**. Se utilizó aceleración por GPU a través de Google Colab para reducir significativamente los tiempos de entrenamiento.

3.4.2. División del dataset

Para entrenar y evaluar de forma rigurosa un modelo de aprendizaje automático, es esencial dividir el conjunto de datos en tres subconjuntos: `entrenamiento`, `prueba` y `validación`. Esta separación permite asegurar que el modelo no solo aprende los patrones del conjunto de entrenamiento, sino que también generaliza correctamente a datos no vistos. Además, permite detectar fenómenos como el sobreajuste (*overfitting*).

- **Conjunto de entrenamiento: 50.000 imágenes**. Utilizado para ajustar los pesos del modelo durante las épocas de entrenamiento.
- **Conjunto de prueba: 10.000 imágenes**. Extraído del conjunto original de entrenamiento (60.000 imágenes totales). Se reserva para evaluar el rendimiento del modelo al final de cada época, sin influir en el ajuste de pesos. Esto permite detectar sobreajuste y ajustar hiperparámetros si es necesario.
- **Conjunto de validación: 10.000 imágenes**. Conjunto estándar de MNIST no visto durante el entrenamiento, utilizado únicamente al final para medir el rendimiento final del modelo cuantizado.

3.4.3. Flujo del entrenamiento

1. Se inicializa el modelo cuantizado (`QNN`) y se traslada al dispositivo correspondiente.
2. Se cargaron los datos mediante `DataLoader` para los tres subconjuntos.
3. Para cada época:
   a) Se entrena el modelo usando los datos del conjunto de entrenamiento.
   b) Tras finalizar la época, se evalua temporalmente el rendimiento en el conjunto de `prueba` para controlar posibles signos de sobreajuste.
4. Al final del proceso completo de entrenamiento, se evalua el modelo en el conjunto de `validación`, desactivando `dropout`, para obtener la exactitud final.

3.4.4. Resultados

Durante el proceso de entrenamiento se registra la precisión obtenida en el conjunto de validación al final de cada época. En la Figura 3.3 se observa cómo la precisión incrementa progresivamente hasta estabilizarse.

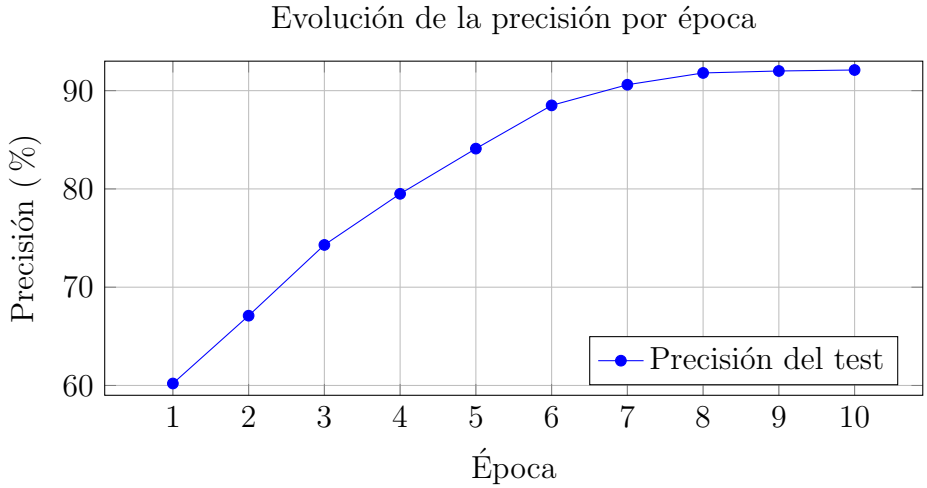


Figura 3.3: Precisión del modelo cuantizado sobre el conjunto de prueba a lo largo de 10 épocas

Finalmente el modelo alcanzó una exactitud del **92.1 %** sobre el conjunto de prueba. Esta cifra, aunque ligeramente inferior al rendimiento de la arquitectura original en punto flotante (alrededor del 96 %), representa una buena compensación entre exactitud y eficiencia computacional, gracias al uso de cuantización a 4 bits en todas las capas.

3.5. Exportación del modelo a formato QONNX

Tras completar el proceso de entrenamiento y validación, el siguiente paso consiste en preparar el modelo para su implementación en hardware utilizando `HLS4ML`. Sin embargo, esta herramienta no soporta directamente modelos definidos con la librería `Brevitas`, ya que sus operadores y capas cuantizadas no son compatibles de forma nativa con el flujo de trabajo de High-Level Synthesis for Machine Learning (hls4ml).

Para solventar esta incompatibilidad, se utiliza `QONNX`, una extensión del formato estándar `Open Neural Network Exchange (ONNX)` que permite conservar la información de cuantización (como el número de bits, escalas y signos) y otras configuraciones específicas de modelos entrenados con Brevitas.

El flujo de conversión seguido es el siguiente:

- **Conversión a QONNX**: Utilizando la función `brevitas.export.export_qonnx()`, se genera un grafo ONNX que reemplaza capas cuantizadas por primitivas estándar de cuantización, el formato QONNX representa explícitamente la cuantización mediante nodos dedicados como `Quant`, `Dequant`, `Trunc`... Este archivo incluye tanto pesos como activaciones cuantizadas a 4 bits, así como información estructural.
- **Inferencia de formas** (`InferShapes`): Se infieren las dimensiones de entrada y salida en cada nodo del grafo, asegurando una estructura completamente definida.
- **Plegado de constantes** (`FoldConstants`): Se eliminan nodos con valores constantes que no influyen en la computación final, reduciendo la complejidad del grafo.
- **Conversión a formato** `channels_last`: Se reorganizan los tensores al formato NHWC (batch, height, width, channels), requerido por algunas herramientas de backend como HLS4ML.
- **Sustitución de** `GEMM` **por** `MatMul`: Las operaciones de multiplicación de matrices (`GEMM`) se simplifican a `MatMul` para garantizar compatibilidad y facilitar la traducción a operaciones de hardware.

3.6. Generación del IP y bitstream mediante HLS4ML y Vitis HLS

La herramienta `HLS4ML` se utiliza para convertir el modelo `QONNX` en un diseño sintetizable. El proceso sigue los siguientes pasos:

1. **Asignación de nombres a nodos**: Algunos nodos del modelo QONNX pueden carecer de un identificador explícito (es decir, un atributo `name`). Estos se conocen como *nodos anónimos*. Aunque no afectan la ejecución del modelo, pueden provocar ambigüedades durante procesos posteriores como la conversión a hardware o la configuración capa por capa en `HLS4ML`. Para evitar este problema, se asignaron nombres únicos y descriptivos a todos los nodos del grafo.

2. **Generación del archivo de configuración con** `config_from_onnx_model()`:

   Este paso genera automáticamente una configuración detallada para cada capa del modelo. Se utilizaron los siguientes parámetros:

   - `granularity='name'`: genera una configuración específica por nombre de capa, permitiendo ajustar parámetros como precisión y paralelismo de forma independiente.
   - `backend='Vitis'`: indica que se usará la herramienta `Vitis HLS` como compilador de alto nivel.
   - `default_precision='fixed<16,6>'`: establece como tipo de dato predeterminado un número en punto fijo de 16 bits, con 6 bits para la parte entera. Aunque todas las capas fueron previamente cuantizadas a 4 bits con **Brevitas**, esta configuración actúa como medida de seguridad para cubrir componentes residuales del modelo.

3. **Conversión del modelo con** `convert_from_onnx_model()`:

   Aquí se transforma el modelo en un objeto intermedio propio de `HLS4ML` que puede ser sintetizado. Los argumentos clave fueron:

   - `output_dir=hls_path`: directorio donde se almacenará el proyecto generado.

- `io_type='io_stream'`: especifica que las interfaces de entrada/salida serán tipo *stream*, adecuadas para procesamiento secuencial de datos en FPGA.
- `backend='Vitis'`: reafirma el uso de Vitis HLS como entorno de síntesis.
- `hls_config=cnnCleanNet_cfg`: archivo de configuración detallado generado en el punto previo.
- `board='pynq-z1'`: especifica la FPGA de destino, en este caso la `PYNQ-Z1`, lo cual permite a HLS4ML ajustar automáticamente restricciones y capacidades de síntesis.

4. **Creación del proyecto HLS con** `.build(export=True, bitfile=True)`: Este paso convierte de forma efectiva el modelo cuantizado en una implementación física lista para ser desplegada en hardware, sin necesidad de intervención manual adicional.

   Se llama al método `.build()` sobre el modelo convertido, indicando los argumentos `export=True` y `bitfile=True`. Esta instrucción automatiza la ejecución de todo el flujo de Vitis HLS y produce:

   - El código fuente sintetizable en `C++`, adaptado a la estructura del modelo.
   - Un proyecto completo de `Vitis HLS`, que incluye un *Block Design* con el núcleo IP de la red neuronal, así como IPs auxiliares necesarias como el controlador `AXI-DMA` para la comunicación entre la lógica programable (PL) y el procesador (PS).
   - El archivo `bitstream (.bit)` completamente generado, listo para ser cargado en la FPGA.

3.7. Validación funcional del IP

Una vez generado el modelo sintetizado y empaquetado, se procedió a validar su funcionamiento directamente sobre la placa `PYNQ-Z1`. Para ello se tomaron los siguientes pasos:

1. **Preparación de archivos**:

   Se recopilaron y empaquetaron todos los archivos necesarios para la ejecución en la `PYNQ-Z1`:

   - Bitstream del diseño: `hls4ml_nn.bit`
   - Descripción hardware del diseño (HWH): `hls4ml_nn.hwh`
   - Driver AXI para comunicación con el IP: `axi_stream_driver.py`
   - Datos de prueba: `X_test.npy`, `y_test.npy`
   - Script Python para ejecución: `deploy.py`

   Estos archivos fueron empaquetados en un archivo `package.tar.gz` y transferidos a la placa mediante `scp`, usando la configuración de red estática por defecto, (`192.168.2.99`) y los credenciales por defecto (`xilinx:xilinx`).

2. **Despliegue y ejecución del modelo en la PYNQ-Z1**:

   En la `PYNQ-Z1`, se descomprimió el paquete y se ejecutó el script `deploy.py`. Este script realiza los siguientes pasos:

   - Carga del bitstream en la lógica programable (PL) mediante el controlador `Overlay` de PYNQ.
   - Inicialización del driver AXI-Stream para establecer comunicación entre el procesador (PS) y el IP de la red neuronal.
   - Carga de los datos de entrada (`X_test`) en memoria y transferencia hacia el IP.
   - Ejecución de la inferencia en hardware y medición de latencia y rendimiento (inferences/s).

- Almacenamiento del resultado de salida en un archivo `y_hw.npy`.

En este caso se observo:

```
Classified 10000 samples in 3.367483 seconds (2969 inferences/s)
```

3. **Transferencia de resultados y validación final**:

   El archivo de resultados `y_hw.npy`, generado en la `PYNQ-Z1` tras la inferencia en hardware, fue transferido de vuelta al host mediante `scp`. Posteriormente, se comparó con las salidas de referencia obtenidas a partir del modelo cuantizado base.

   La exactitud final del modelo desplegado en la FPGA el 90.9 %, prácticamente idéntica a la obtenida con el modelo original en Brevitas.

   Esta ligera diferencia puede deberse a diversos factores inherentes al flujo de síntesis y despliegue:

   - **Redondeos y saturaciones en tiempo de inferencia**: Los bloques generados por `Vitis HLS` pueden aplicar reglas de redondeo, truncamiento o saturación distintas a las utilizadas durante el entrenamiento con `Brevitas`, lo que provoca pequeñas diferencias acumuladas en el cómputo de activaciones.
   - **Implementación de operaciones no idéntica**: Algunos operadores del modelo (como `ReLU`, `MaxPool` o funciones de activación) pueden estar implementados de forma ligeramente diferente en hardware que en las bibliotecas de `PyTorch`/`Brevitas`.

3.8. Limitaciones del modelo de prueba

Aunque el modelo propuesto cumple satisfactoriamente su objetivo como prueba de concepto y ha obtenido buenos resultados, presenta aún ciertas limitaciones que conviene tener en cuenta:

- **Simplicidad del problema:** El dataset MNIST, aunque útil para validar arquitecturas y flujos de trabajo, no representa la complejidad de aplicaciones reales como clasificación de imágenes de alta resolución, donde se requieren modelos más robustos.
- **Arquitectura sencilla:** La red CNN utilizada tiene una profundidad limitada, sin capas residuales. Esto facilita la síntesis pero no explora todos los desafíos de implementar redes más complejas en hardware.
- **Optimización de recursos limitada:** No se exploraron técnicas avanzadas de optimización como poda, ni búsqueda de hiperparámetros adaptados al hardware, lo cual deja margen de mejora en cuanto a eficiencia de área.

Estas limitaciones son deliberadas, dado que el objetivo principal de este modelo fue validar la viabilidad del flujo completo desde un modelo cuantizado en Brevitas hasta su ejecución en FPGA mediante QONNX y HLS4ML.

Capítulo 4

Análisis del problema y reducción

En este capítulo se analiza la naturaleza del conjunto de datos utilizado en el proyecto, así como las dificultades que presenta para la detección de objetos pequeños en entornos marítimos. Se detallan las distintas técnicas de preprocesado exploradas para mitigar estas dificultades, con el objetivo de resaltar las regiones de interés y reducir el ruido visual en las imágenes. Finalmente, se justifica la selección de la estrategia adoptada y se analiza su impacto sobre el rendimiento del modelo.

4.1. Análisis del conjunto de datos objetivo

Para el entrenamiento y evaluación del modelo final se utilizó el conjunto de datos `Masati-v2`, una versión extendida del dataset MASATI [20], diseñado para tareas de detección y clasificación en entornos marítimos. El dataset está compuesto por miles de imágenes RGB de resolución 512×512 píxeles, capturadas desde una vista aérea o satelital. Las escenas representan zonas *costeras*, *océano abierto* y *tierra firme*, tanto con como sin presencia de embarcaciones.

Esta variedad de escenarios aporta riqueza visual al dataset, pero introduce también una alta variabilidad de fondo, dificultando la detección de objetos pequeños como barcos, especialmente cerca de la costa. Por ello, se analizan técnicas de preprocesado orientadas a resaltar las estructuras relevantes (embarcaciones) y reducir el ruido visual (como la tierra, mar o la costa), con el objetivo adicional de reducir el número de canales de entrada a la red, pasando de tres canales RGB a una única banda informativa; con la consecuente reducción del tamaño de entrada del modelo, así como la complejidad computacional y memoria requerida.

Figura 4.1: Ejemplos de imágenes del conjunto de datos Masati-v2: con embarcaciones (izquierda) y sin embarcaciones (derecha).

4.2. Técnicas de preprocesado evaluadas

En la fase inicial de exploración se considera como método base una simple conversión de las imágenes RGB a escala de grises, aprovechando su simplicidad y bajo coste computacional. Sin embargo, se estima que esta transformación no explotaba adecuadamente la información contenida en los tres canales de color. Esto motivó la búsqueda de técnicas más expresivas que pudieran realzar mejor las diferencias espectrales entre el agua y otros elementos en la escena.

Durante esta búsqueda se exploraran también métodos basados en kernels de tamaño 3×3 o 5×5, como filtros de detección de bordes (edge detection) y otros filtros espaciales para realce o suavizado. Aunque estos métodos pueden mejorar ciertos detalles locales en las imágenes, añaden una carga computacional adicional y complejidad al proceso de preprocesado.

Dado que el objetivo principal de esta etapa es reducir la complejidad del sistema y no aumentarla, se opta por centrarse en métodos que funcionen píxel a píxel (de manera similar a la conversión a escala de grises), evitando operaciones convolucionales complejas. En esta línea se descubre el *Normalized Difference Water Index* (NDWI)[21], un índice espectral diseñado para resaltar cuerpos de agua en imágenes satelitales.

El NDWI tiene una gran relevancia en aplicaciones de teledetección por su simplicidad y eficacia para distinguir áreas acuáticas de tierra firme basándose únicamente en el valor de cada píxel, sin necesidad de procesamiento espacial, convirtiéndolo en un candidato ideal. En su formulación clásica, el NDWI se define como:

$$NDWI = \frac{G - NIR}{G + NIR} \tag{4.1}$$

Donde G representa la intensidad del canal verde (*Green*) y NIR la del canal infrarrojo cercano (*Near-Infrared*). Un valor de $NDWI$ más cercano a 1 indica una mayor probabilidad de presencia de agua, mientras que valores bajos o negativos suelen corresponder a terreno, suelo u otras superficies no acuáticas.

4.2.1. NDWI Modificado

En nuestro caso no se dispone del canal infrarrojo cercano (NIR), por lo que se desarrolló una variante adaptada a imágenes RGB mediante un proceso iterativo. Este proceso consiste en modificar progresivamente la fórmula y evaluar los resultados utilizando el modelo ya entrenado desarrollado durante las prácticas de empresa, que permitía comparar objetivamente el impacto de cada versión en la detección de embarcaciones. Gracias a esta metodología se llegó a la siguiene fórmula:

$$NDWI^*(x, y) = 1 - \left(\frac{B - R^2}{B + R^2}\right)^2 \tag{4.2}$$

Donde R y B representan los valores de los canales rojo y azul del píxel, y se ha invertido la función, donde un valor de $NDWI^*$ más cercano a 1 indica en este caso una mayor probabilidad de embarcaciones, suelo u otras superficies no acuáticas.

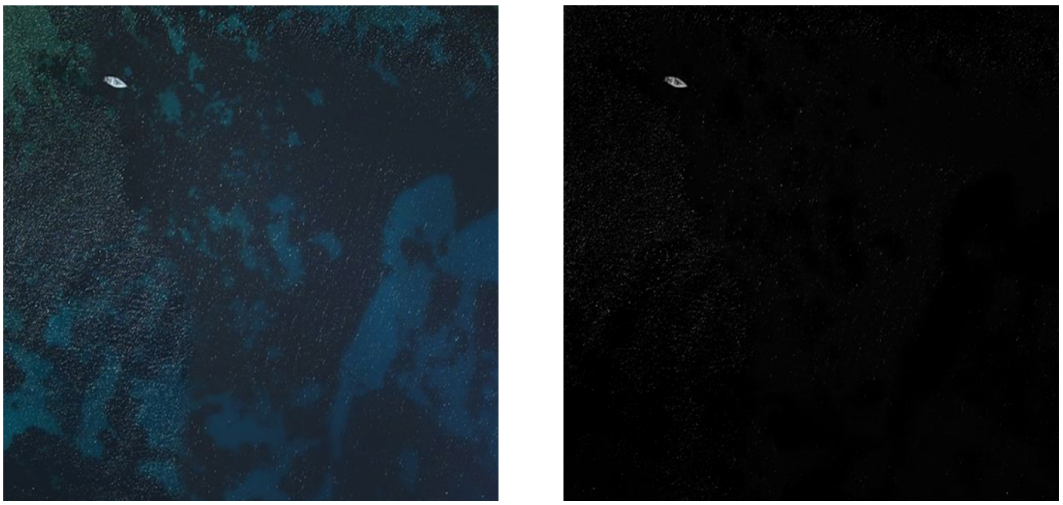


Figura 4.2: Ejemplo de imagen maritima original (izquierda) y su correspondiente imagen procesada con el NDWI modificado (derecha).

Desafortunadamente, este método tiene una desventaja, si bien el NDWI modificado resalta eficazmente objetos contra cuerpos de agua también tiende a resaltar las regiones costeras y, en general, cualquier objeto o área con colores significativamente distintos al agua, lo que puede incluir estructuras terrestres o elementos flotantes; lo que generó sospechas de posibles falsos positivos. Motivando la exploración de otro método capaz de superar dichos problemas.

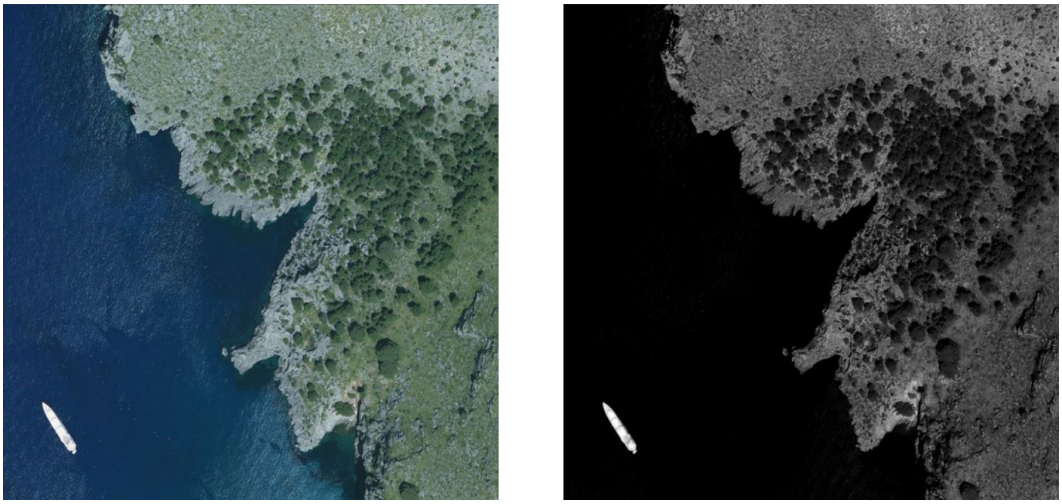


Figura 4.3: Ejemplo de imagen costera original (izquierda) y su correspondiente imagen procesada con el NDWI modificado (derecha).

4.2.2. NDWI Modificado + Detección de costas

Este método es una extensión del método NDWI modificado que introduce una etapa adicional para eliminar regiones costeras que pueden inducir falsos positivos en la detección de embarcaciones. El objetivo es generar una máscara binaria $M_{\text{ocean}}(x, y)$ que anule las zonas cercanas a la costa o estructuras terrestres complejas.

La imagen final se obtiene aplicando dicha máscara al índice NDWI modificado:

$$NDWI_{\text{final}}(x, y) = NDWI^*(x, y) \cdot M_{\text{ocean}}(x, y) \tag{4.3}$$

El proceso de generación de esta máscara consta de varias etapas secuenciales, las cuales se describen a continuación:

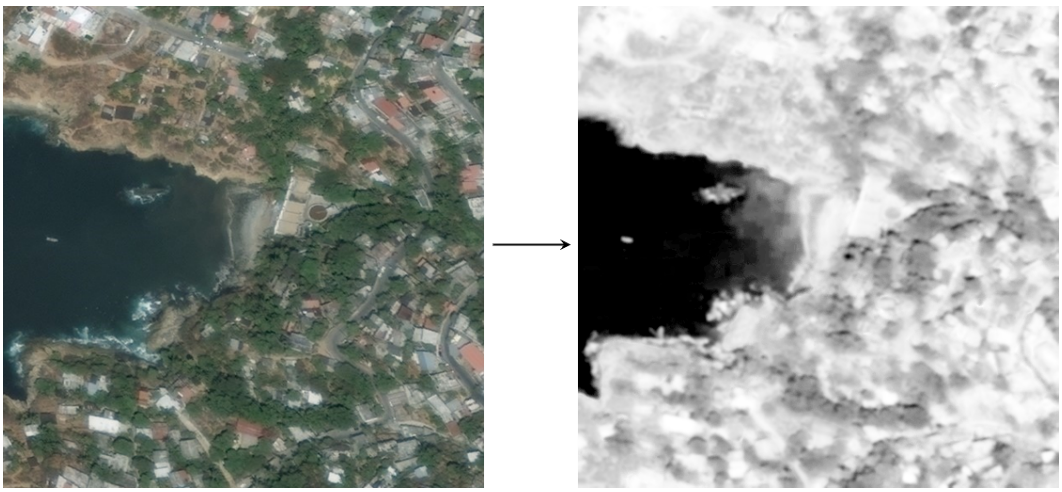


Figura 4.4: Imagen costera original (izquierda) y su correspondiente imagen procesada con el NDWI* (derecha).

1. **Binarización inicial del NDWI modificado:** se invierte el mapa $NDWI^*$ y se convierte en una máscara binaria $M_0(x, y)$ aplicando un umbral fijo τ_{mask} seleccionado por el usuario, típicamente entre 0.4 y 0.6. Píxeles con alta probabilidad de ser agua se marcan como 1, y el resto como 0.

$$M_0(x, y) = \begin{cases} 1 & \text{si } 1 - NDWI^*(x, y) > \tau_{\text{mask}} \\ 0 & \text{en otro caso} \end{cases} \tag{4.4}$$

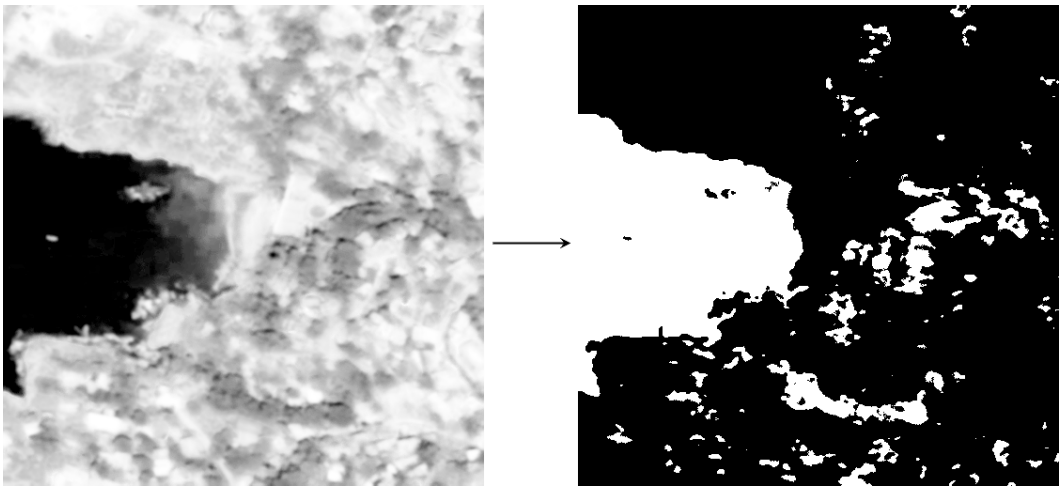


Figura 4.5: Máscara binaria obtenida por umbralización del NDWI modificado.

2. **Reducción de ruido mediante pooling:** Para eliminar pequeños artefactos o falsos positivos derivados de la binarización inicial, se aplica una operación de *average pooling* sobre la máscara binaria $M_0(x, y)$ utilizando bloques de 32×32 píxeles.

   A continuación, se aplica un segundo umbral τ_{pool} sobre el resultado del pooling. Este umbral define el porcentaje mínimo de píxeles activos (con valor 1) necesario dentro de cada bloque para que la región se mantenga marcada como un posible cuerpo de agua. Si el promedio es menor que el umbral, se considera que el bloque contiene solo ruido o pequeñas imperfecciones y se descarta.

   El valor seleccionado de $\tau_{\text{pool}} = 0{,}4$ implica que al menos el 40 % de los píxeles del bloque deben estar activos para que la región sea retenida.

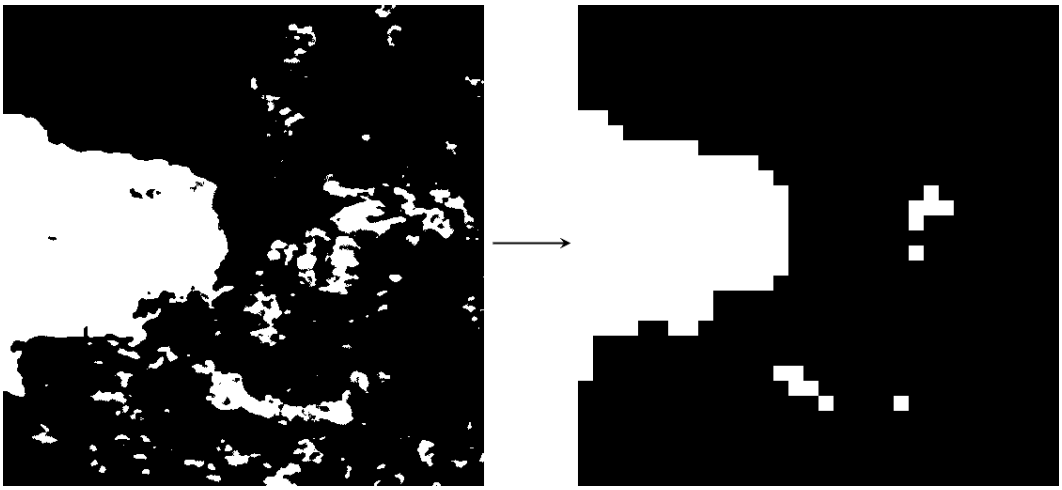


Figura 4.6: Resultado del average pooling sobre la máscara binaria.

3. **Aplicación de flood fill:** partiendo de los bordes de la imagen, se realiza una operación de *flood fill* sobre las regiones que aún estén marcadas como agua. Esta técnica permite identificar el océano abierto conectando únicamente las regiones contiguas más exteriores, y descartando masas de agua interiores o ambiguas.

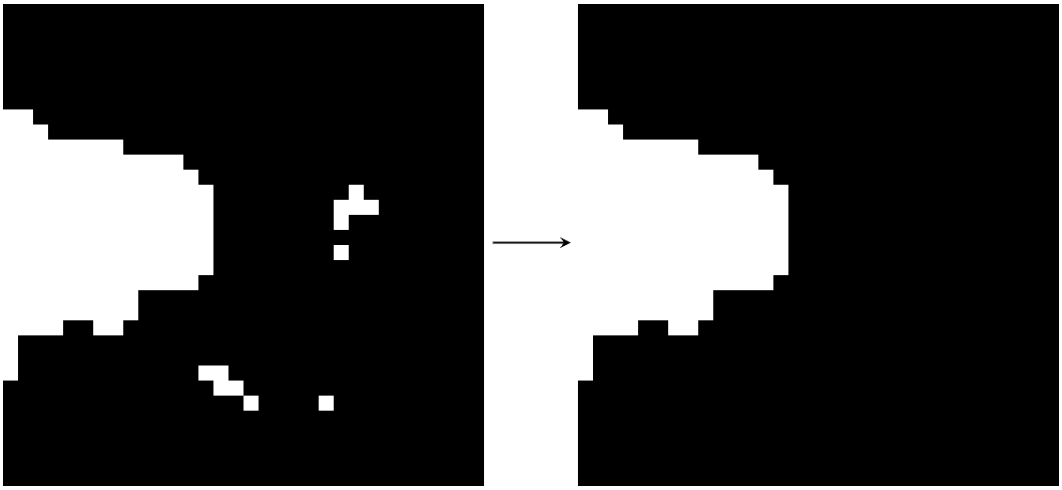


Figura 4.7: Máscara resultante tras la operación de flood fill desde los bordes.

4. **(Opcional) Morfología matemática (dilatación):** opcionalmente se puede aplicar una dilatación a la máscara final para cubrir bordes fragmentados o suavizar el contorno entre mar y costa, aunque esta operacion es muy costosa computacionalmente y no se utilizó en el método final.

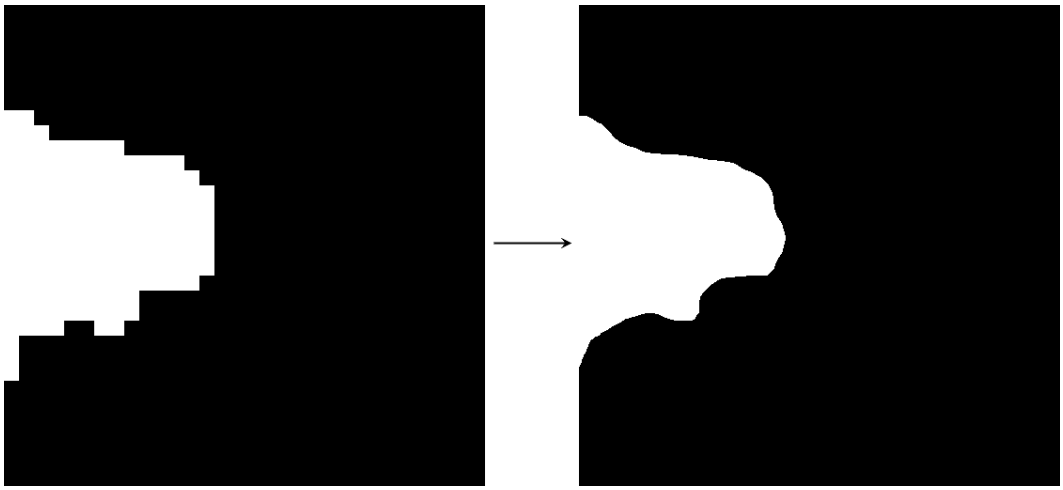


Figura 4.8: Máscara oceánica tras aplicar morfología matemática.

El resultado es una máscara robusta $M_{\text{ocean}}(x, y)$ que identifica con relativa precisión las regiones oceánicas abiertas, eliminando visualmente las zonas costeras, terreno flotante o bordes que puedan inducir falsos positivos. Esta máscara se multiplica finalmente por el mapa de NDWI modificado para obtener una representación más limpia y centrada exclusivamente en el mar.

Cabe destacar que, dado que el proceso conserva regiones compactas y aisladas no conectadas con la costa, los objetos pequeños presentes en el mar, como embarcaciones, pero también rocas o formaciones naturales persisten en la imagen resultante, lo cual es un comportamiento deseado en el contexto de la detección. Será entonces responsabilidad del modelo de detección interpretar y clasificar correctamente estos elementos.

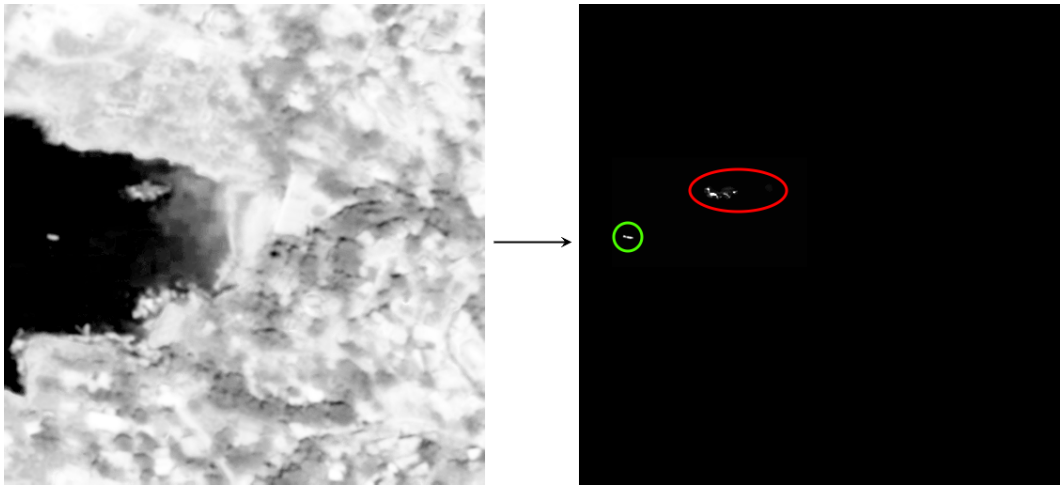
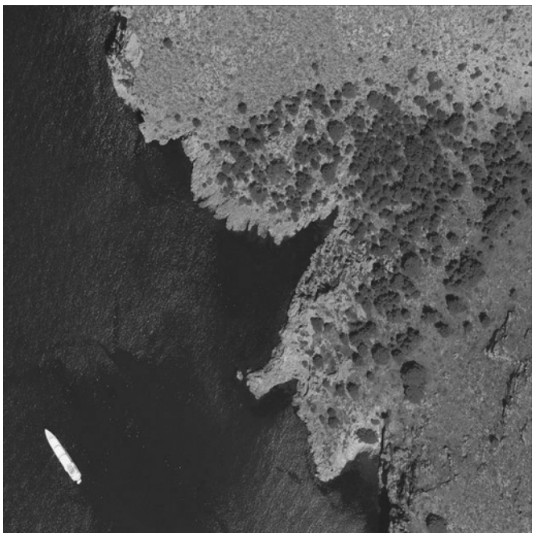


Figura 4.9: Imagen procesada con el NDWI modificado (izquierda) y su correspondiente imagen procesada tras haber aplicado la máscara (derecha). Se observa la embarcación señalada en verde, y una formación rocosa en rojo,

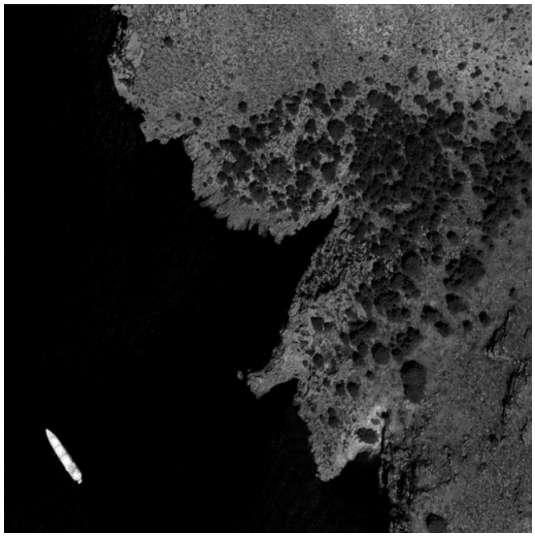
4.3. Comparación visual de las técnicas evaluadas

A continuación se muestra una comparación visual entre las tres técnicas de preprocesado analizadas: la conversión directa a escala de grises, el NDWI modificado, y la versión con eliminación de costa. Estas imágenes ilustran claramente cómo cada método transforma la representación original, destacando diferentes características relevantes para la detección de embarcaciones.

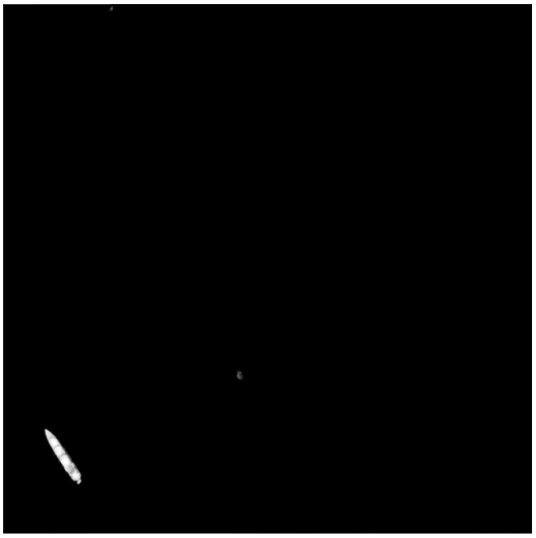
(a) Imagen RGB original



(b) Escala de grises



(c) NDWI modificado



(d) NDWI con eliminación de costa

Figura 4.10: Comparación visual de las distintas técnicas de preprocesado aplicadas sobre una imagen RGB original.

4.4. Resumen del preprocesado y decisión metodológica

Tras una fase de exploración experimental, se identificó el NDWI modificado como el método de preprocesado más prometedor, gracias a su capacidad para resaltar eficazmente las zonas oceánicas y mejorar el contraste de las embarcaciones frente al mar. Frente a la escala de grises utilizada inicialmente como referencia base, el NDWI ofrecía una representación más informativa, basada en una mejor explotación de las relaciones entre los canales RGB.

Por otro lado, la variante con eliminación de costa, aunque más precisa en la supresión de bordes y zonas terrestres, introducía una complejidad computacional adicional que no resultó compensada por mejoras significativas durante las primeras evaluaciones. No obstante, algunos casos límite aún motivaron su análisis.

Dado que el objetivo de este capítulo es analizar las estrategias de reducción y preprocesado, se omite aquí el análisis del rendimiento específico sobre las arquitecturas. Dicho análisis se desarrolla en el siguiente capítulo, donde se estudia la influencia de cada técnica de preprocesado en el comportamiento de las distintas arquitecturas.

Capítulo 5

Diseño e Implementación del Modelo Final

Este capítulo describe el proceso seguido para diseñar, entrenar y desplegar el modelo de detección de embarcaciones, optimizado para su implementación en hardware FPGA. Se definen los requisitos funcionales y técnicos clave del sistema, como exactitud, latencia, uso de recursos y compatibilidad con herramientas de síntesis de alto nivel.

Se presenta un estudio comparativo de distintas arquitecturas de Red Neuronal Convolucional (CNN), evaluando su rendimiento y el impacto de los métodos de preprocesado descritos en el capítulo anterior. A partir de estos resultados, se selecciona una arquitectura final que equilibra exactitud y eficiencia.

También se detallan las etapas de entrenamiento, ajuste fino y evaluación del modelo. A continuación, se exploran técnicas de cuantización orientadas a la conversión al formato QONNX, necesario para su integración con hls4ml y la generación del acelerador hardware.

Finalmente, se describe la síntesis del IP, su validación funcional y la implementación en FPGA, acompañadas de un análisis de los resultados obtenidos.

5.1. Requisitos funcionales y técnicos

El sistema debía cumplir con los siguientes requisitos:

- Procesar imágenes del dataset MASATI-v2 en tiempo cercano al real.
- Alcanzar una exactitud superior al 90 % en la tarea de clasificación binaria (barco/no barco).
- Tener un número de parámetros suficientemente reducido para ser implementado en una FPGA, en este caso una PYNQ-Z1.
- Mantener una arquitectura completamente cuantizada, idealmente a 8 bits, para optimizar recursos y consumo.

5.2. Estudio de distintas arquitecturas CNN

El proceso de diseño del modelo final comenzó con la exploración de distintas arquitecturas CNN adaptadas a las restricciones impuestas por la implementación en FPGAs. Se realizaron múltiples iteraciones, con el objetivo de lograr un equilibrio entre exactitud, eficiencia computacional y tamaño del modelo.

5.2.1. Criterios de evaluación del rendimiento

Antes de detallar las arquitecturas evaluadas, es importante aclarar las métricas utilizadas para medir su rendimiento. En problemas de clasificación binaria, como el abordado en este proyecto, existen diversas métricas relevantes:

	Predicción: Barco	**Predicción: No Barco**
Real: Barco	Verdadero Positivo (TP)	Falso Negativo (FN)
Real: No Barco	Falso Positivo (FP)	Verdadero Negativo (TN)

Tabla 5.1: Tabla de confusión: relaciones entre clases reales y predichas

- **Accuracy** (exactitud): proporción de predicciones correctas sobre el total de predicciones.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision** (precisión): proporción de verdaderos positivos sobre todas las predicciones positivas.

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

- **Recall** (exhaustividad): proporción de verdaderos positivos sobre el total de elementos positivos reales.

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

- **F1-Score**: media armónica entre precisión y *recall*, especialmente útil cuando hay desbalance entre clases.

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

En el presente trabajo, el conjunto de datos empleado ha sido cuidadosamente balanceado para contener cantidades identicas de ejemplos de ambas clases (*barco* y *no-barco*). En este contexto, la métrica más representativa y directa para evaluar el rendimiento general del modelo es la `accuracy`, ya que no se ve sesgada por distribuciones desbalanceadas. Por tanto, todas las referencias a `exactitud` en este capítulo se entienden como `accuracy`, salvo que se indique lo contrario.

5.2.2. Punto de partida: ResNet18 y CNN512

Durante las prácticas externas en la empresa **Thales Alenia Space España**, se desarrolló un sistema de clasificación de imágenes satelitales basado en la red `ResNet18`. Esta arquitectura, ampliamente utilizada en tareas de clasificación, proporcionó buenos resultados en ese contexto, alcanzando una exactitud del 93 %. Sin embargo, con alrededor de 11.7 millones de parámetros, su implementación en hardware era inviable.

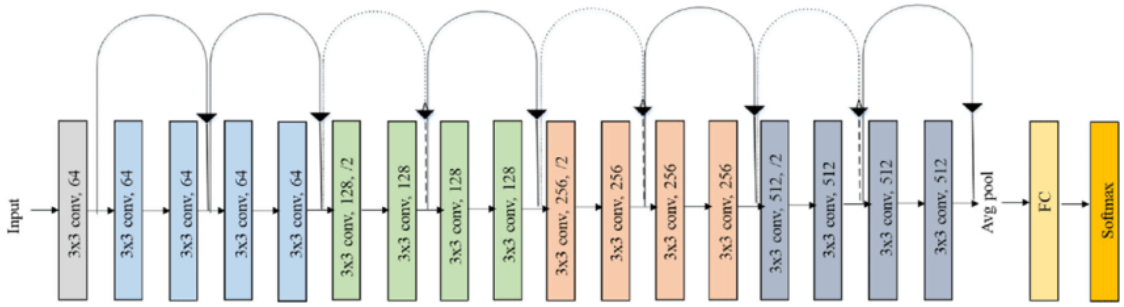


Figura 5.1: Arquitectura original de la Resnet18

Debido a su éxito en el proyecto anterior, el desarrollo inicial del presente trabajo adoptó un enfoque similar, basado en analizar directamente imágenes completas de 512×512 píxeles. Para ello se diseñó la arquitectura `CNN512`, una red CNN más sencilla y ligera que ResNet18, con el objetivo de reducir drásticamente la complejidad y el número de parámetros, manteniendo un rendimiento razonable.

`CNN512` cuenta con tan solo 24 547 parámetros y una arquitectura (Tabla 5.2) inspirada en modelos tradicionales de clasificación, combinando capas convolucionales cuantizadas con funciones de activación y reducción de dimensionalidad. Sin embargo, a pesar de los esfuerzos de optimización, el modelo solo alcanzó una exactitud del 75 %,

lo que se consideró insuficiente para los objetivos del proyecto.

La causa principal de este bajo rendimiento se identificó en el diseño de la red: debido a la necesidad de limitar el número total de parámetros, cada capa convolucional fue significativamente reducida en tamaño y profundidad. Esta decisión provocó una pérdida acelerada de resolución espacial en las primeras etapas del modelo, lo que dificultó la detección de objetos pequeños, como las embarcaciones presentes en las imágenes satelitales, que requieren una mayor preservación del detalle visual.

Este resultado motivó un cambio conceptual importante: abandonar el enfoque de clasificación sobre imágenes completas y adoptar una estrategia basada en subimágenes o patches, con el fin de mejorar la capacidad de detección sobre regiones más localizadas de la imagen.

Capa	**Descripción**
`QuantIdentity`	Cuantización de entrada a 8 bits
`QuantConv2d (1→16)`	Kernel 3×3, stride 1, padding 1
`QuantReLU`	Activación cuantizada
`QuantConv2d (16→32)`	Kernel 3×3, stride 2, padding 1
`QuantReLU`	Activación cuantizada
`QuantConv2d (32→32)`	Kernel 3×3, stride 2, padding 1
`QuantReLU`	Activación cuantizada
`AdaptiveAvgPool2d (1×1)`	Promediado global de características
`QuantLinear (32→2)`	Clasificador lineal final

Tabla 5.2: Capas de la arquitectura `CNN512` (24 547 parámetros)

5.2.3. Cambio de paradigma: Patches de 128×128

Dada la limitada exactitud obtenida al clasificar imágenes completas de 512×512 píxeles, se adoptó un nuevo enfoque basado en la subdivisión de cada imagen en patches de 128×128 píxeles, con una superposición (*stride*) de 64 píxeles entre regiones adyacentes. Esta técnica permite reducir la complejidad de cada entrada, focalizar la atención del modelo en detalles locales, y facilitar el uso de arquitecturas más pequeñas y eficientes sin comprometer el rendimiento.

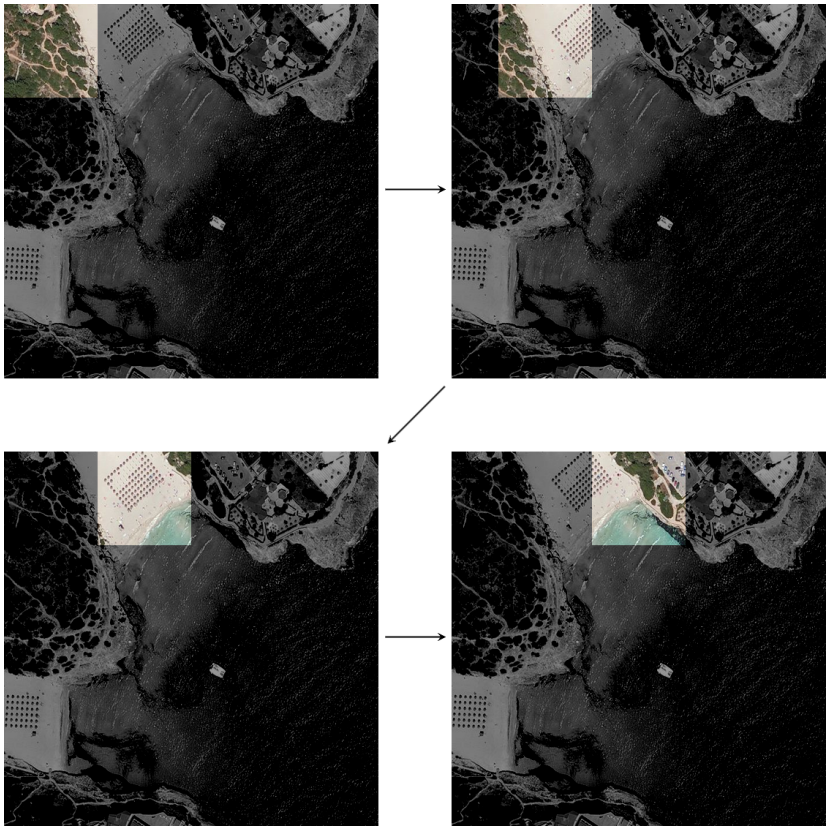


Figura 5.2: División de una imagen satelital de 512×512 en 49 patches de 128×128 con stride 64.

Además de mejorar la exactitud, este cambio de paradigma ofrece una ventaja operativa clave: permite estimar la posición aproximada de una embarcación dentro de la imagen original. Si un determinado patch es clasificado como "barco", puede inferirse que la embarcación se encuentra en la región cubierta por ese patch, permitiendo una forma básica de detección espacial sin recurrir a arquitecturas más complejas como *segmentación semántica* o *bounding boxes*.

Gracias al uso de stride 64 en la generación de patches, cada píxel de la imagen está presente en múltiples patches superpuestos (hasta 4, en el caso del centro de la imagen). Esto permite una estimación más robusta: si al menos tres patches adyacentes (que se solapan parcialmente) son clasificados como positivos, se puede asumir con alta probabilidad que la embarcación se encuentra en la región común a esos tres. Como esta intersección tiene un tamaño máximo de 64×64 píxeles, se puede determinar una zona de presencia del barco con esa resolución espacial.

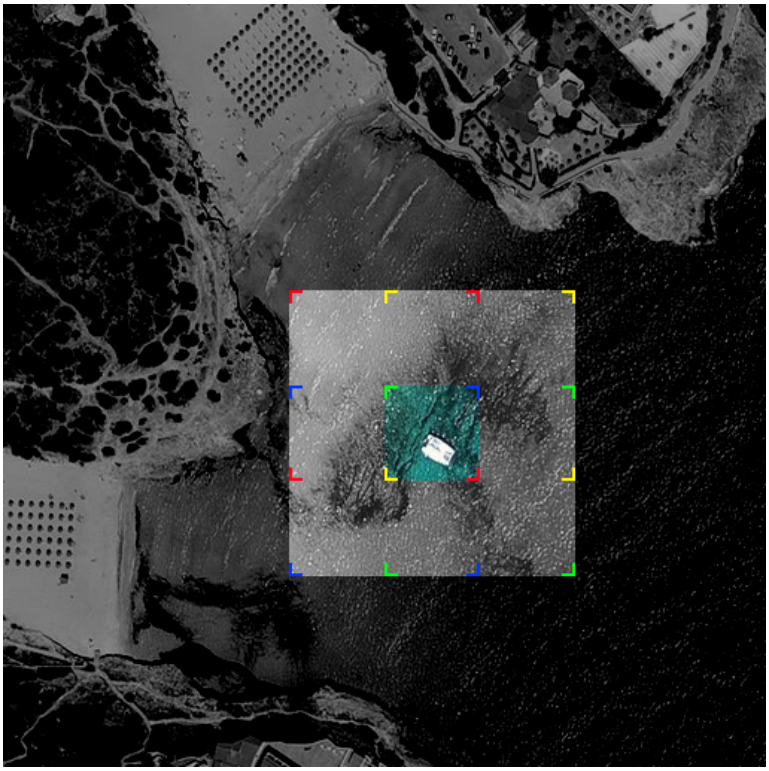


Figura 5.3: Estimación de la posición de la embarcación a partir de la clasificación positiva de varios patches.

Este enfoque híbrido, entre clasificación y localización, resulta especialmente valioso en contextos embebidos, como satélites, donde se desea realizar un filtrado o priorización de imágenes en tiempo real, sin incurrir en el coste computacional de una detección exhaustiva.

A continuación, se describen las distintas arquitecturas desarrolladas bajo esta estrategia de clasificación por patches, explicando su motivación, estructura y resultados.

5.2.4. CNN128_Shallow

Motivación: Validar la viabilidad del enfoque por patches con una arquitectura mínima (Tabla 5.3) y de bajo coste.

Resultados: Muy baja exactitud (máx. 72 %). Resultó insuficiente, pero útil como línea base para comparaciones posteriores.

exactitud por preprocesado:

- Grayscale: 67 %
- NDWI Modificado: 66 %
- NDWI Modificado + Detección de costa: 72 %

Capa	**Descripción**
`QuantIdentity`	Entrada cuantizada a 8 bits
`QuantConv2d (1→8)`	Kernel 3×3, stride 1
`QuantReLU`	Activación cuantizada
`QuantConv2d (8→8)`	Kernel 3×3, stride 2
`QuantReLU`	Activación cuantizada
`AdaptiveAvgPool2D`	Promedio global espacial
`QuantLinear (8→2)`	Capa de clasificación

Tabla 5.3: Capas de la arquitectura `CNN128_Shallow` (4 887 parámetros)

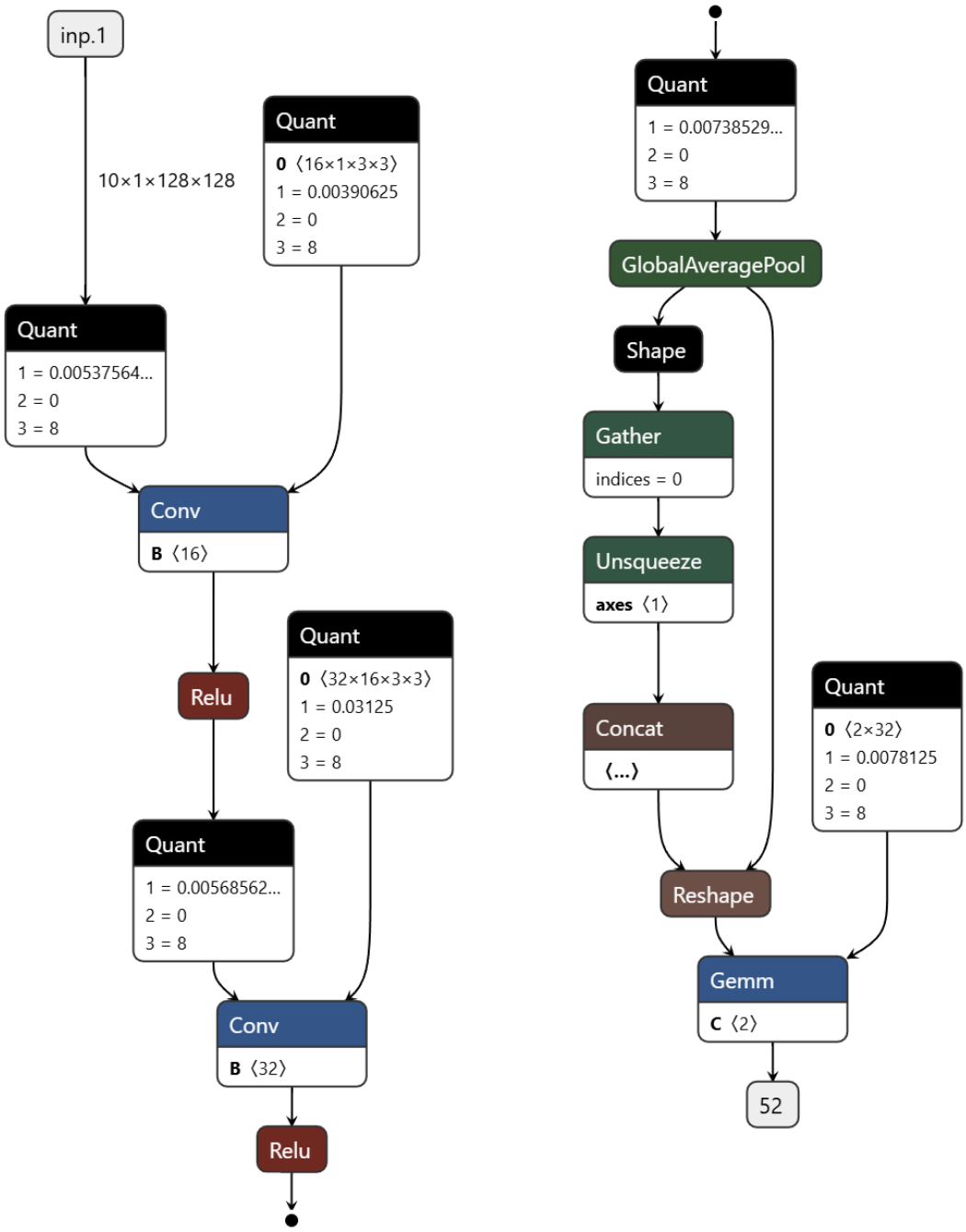


Figura 5.4: Visualización del modelo convertido a QONNX CNN128_Shallow en Netron

5.2.5. CNN128_DeepQuant_k5

Motivación: Maximizar la exactitud con una red profunda (Tabla 5.4) y kernels grandes (5×5).

Resultados: Muy buena exactitud (hasta 97 %), pero con 254 345 parámetros, resultó demasiado grande para una implementación eficiente en hardware.

Exactitud por preprocesado:

- Grayscale: 94 %
- NDWI Modificado: 97 %
- NDWI Modificado + Detección de costa: 92 %

Capa	**Descripción**
ConvBlock1	Conv(1→32), 5×5 + ReLU + Conv(32→32), stride 2
ConvBlock2	Conv(32→64), 5×5 + ReLU + Conv(64→64), stride 2
ConvBlock3	Conv(64→128), 3×3, stride 2 + ReLU
AdaptiveAvgPool2D	
QuantLinear (128→2)	

Tabla 5.4: Capas de la arquitectura `CNN128_DeepQuant_k5` (254 345 parámetros)

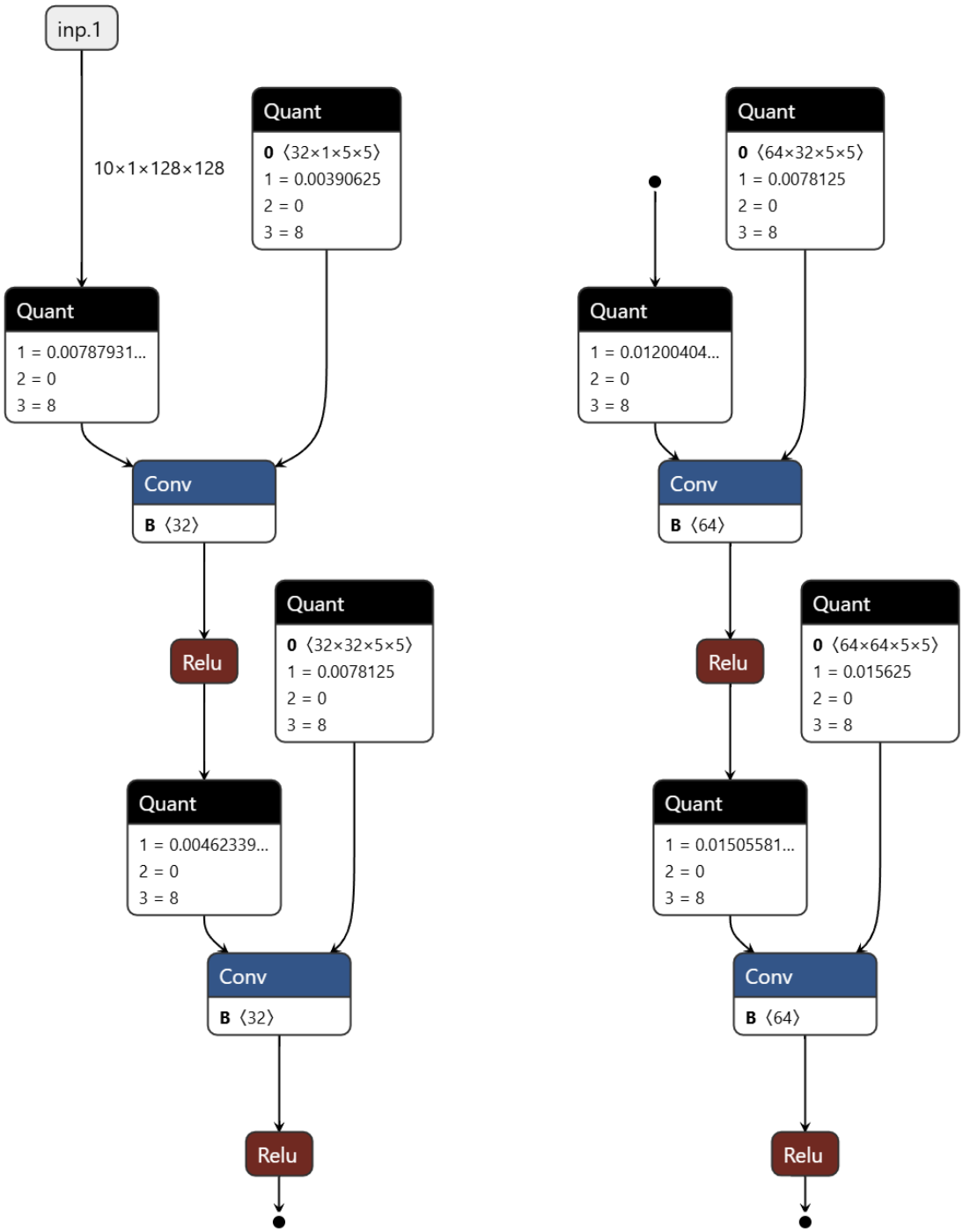


Figura 5.5: Visualización del modelo convertido a QONNX CNN128_DeepQuant_k5 en Netron (Parte 1)

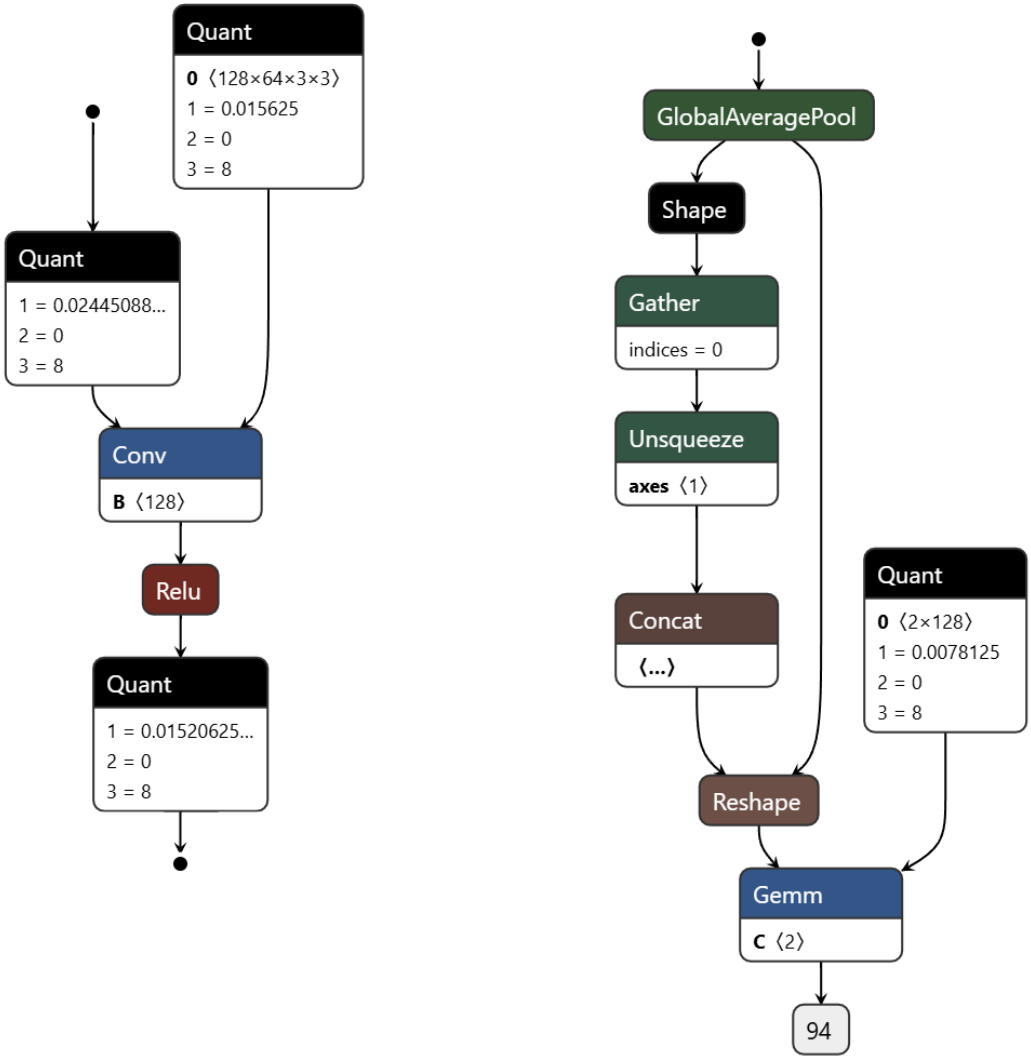


Figura 5.6: Visualización del modelo convertido a QONNX CNN128_DeepQuant_k5 en Netron (Parte 2)

5.2.6. CNN128_DeepQuant_k3

Motivación: Probar si una reducción del tamaño de los kernel (3×3) en las capas de la arquitectura (Tabla 5.5) podía conservar la exactitud mientras se disminuían los parámetros.

Resultados: Buena exactitud (hasta 93 %) y complejidad reducida (139 145 parámetros), aunque aún algo elevada para una FPGA.

Exactitud por preprocesado:

- Grayscale: 90 %
- NDWI Modificado: 91 %
- NDWI Modificado + Detección de costa: 89 %

Capa	**Descripción**
ConvBlock1	Conv(1→32), 3×3 + ReLU + Conv(32→32), stride 2
ConvBlock2	Conv(32→64), 3×3 + ReLU + Conv(64→64), stride 2
ConvBlock3	Conv(64→128), 3×3, stride 2 + ReLU
AdaptiveAvgPool2D	
QuantLinear (128→2)	

Tabla 5.5: Capas de la arquitectura `CNN128_DeepQuant_k3` (139 145 parámetros)

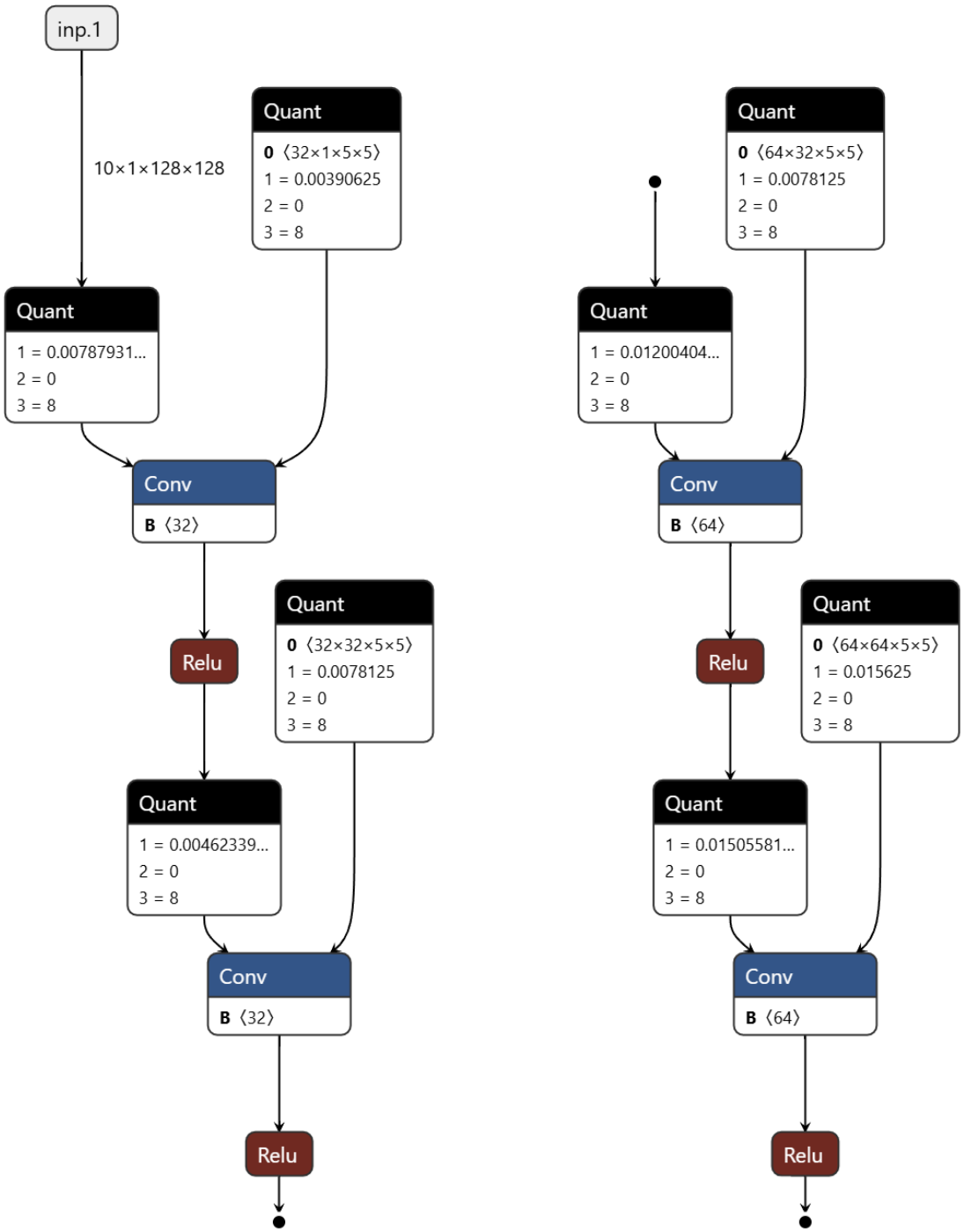


Figura 5.7: Visualización del modelo convertido a QONNX CNN128_DeepQuant_k3 en Netron (Parte 1)

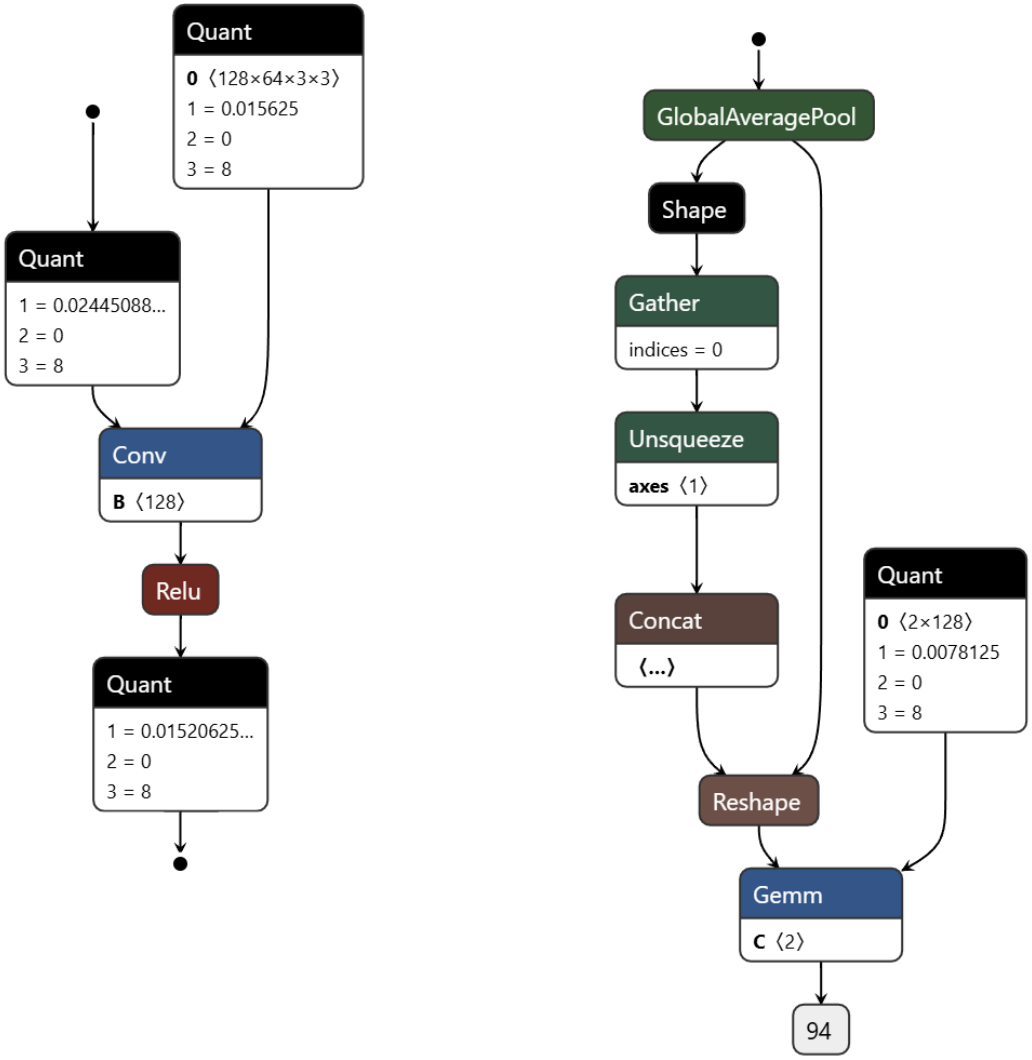


Figura 5.8: Visualización del modelo convertido a QONNX CNN128_DeepQuant_k3 en Netron (Parte 2)

5.2.7. CNN128_Compact_k3

Motivación: Reducir aún más el número de canales y parámetros en la arquitectura (Tabla 5.6) manteniendo la estructura general de la iteración previa.

Resultados: Muy buena exactitud (91 %) con sólo 35 033 parámetros. Se posicionó como una alternativa sólida por su balance rendimiento/eficiencia.

Exactitud por preprocesado:

- Grayscale: 89 %
- NDWI Modificado: 91 %
- NDWI Modificado + Detección de costa: 87 %

Capa	**Descripción**
ConvBlock1	Conv(1→16), 3×3 + ReLU + Conv(16→16), stride 2
ConvBlock2	Conv(16→32), 3×3 + ReLU + Conv(32→32), stride 2
ConvBlock3	Conv(32→64), 3×3, stride 2 + ReLU
AdaptiveAvgPool2D	
QuantLinear (64→2)	

Tabla 5.6: Capas de la arquitectura `CNN128_Compact_k3` (35 033 parámetros)

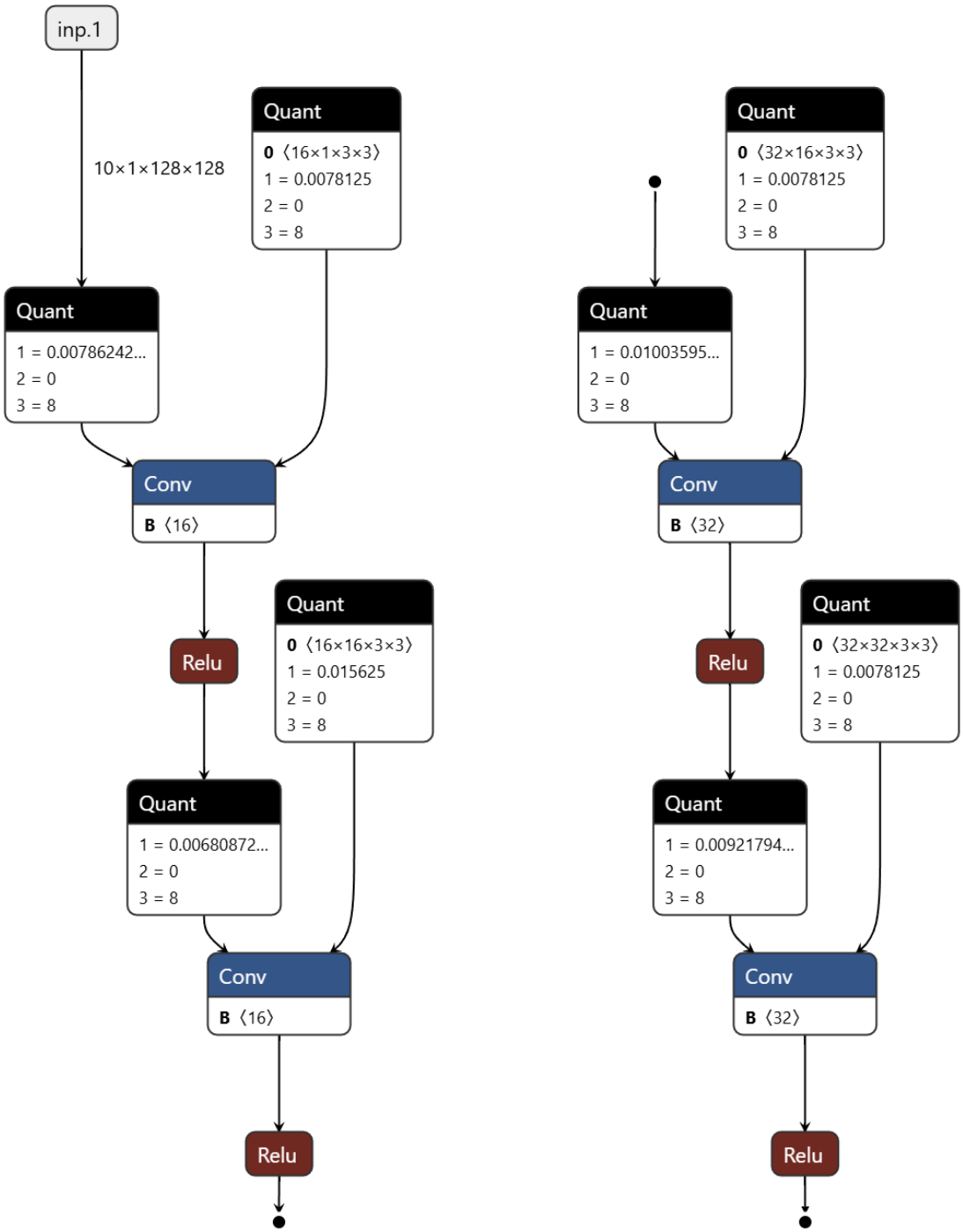


Figura 5.9: Visualización del modelo convertido a QONNX CNN128_Compact_k3 en Netron (Parte 1)

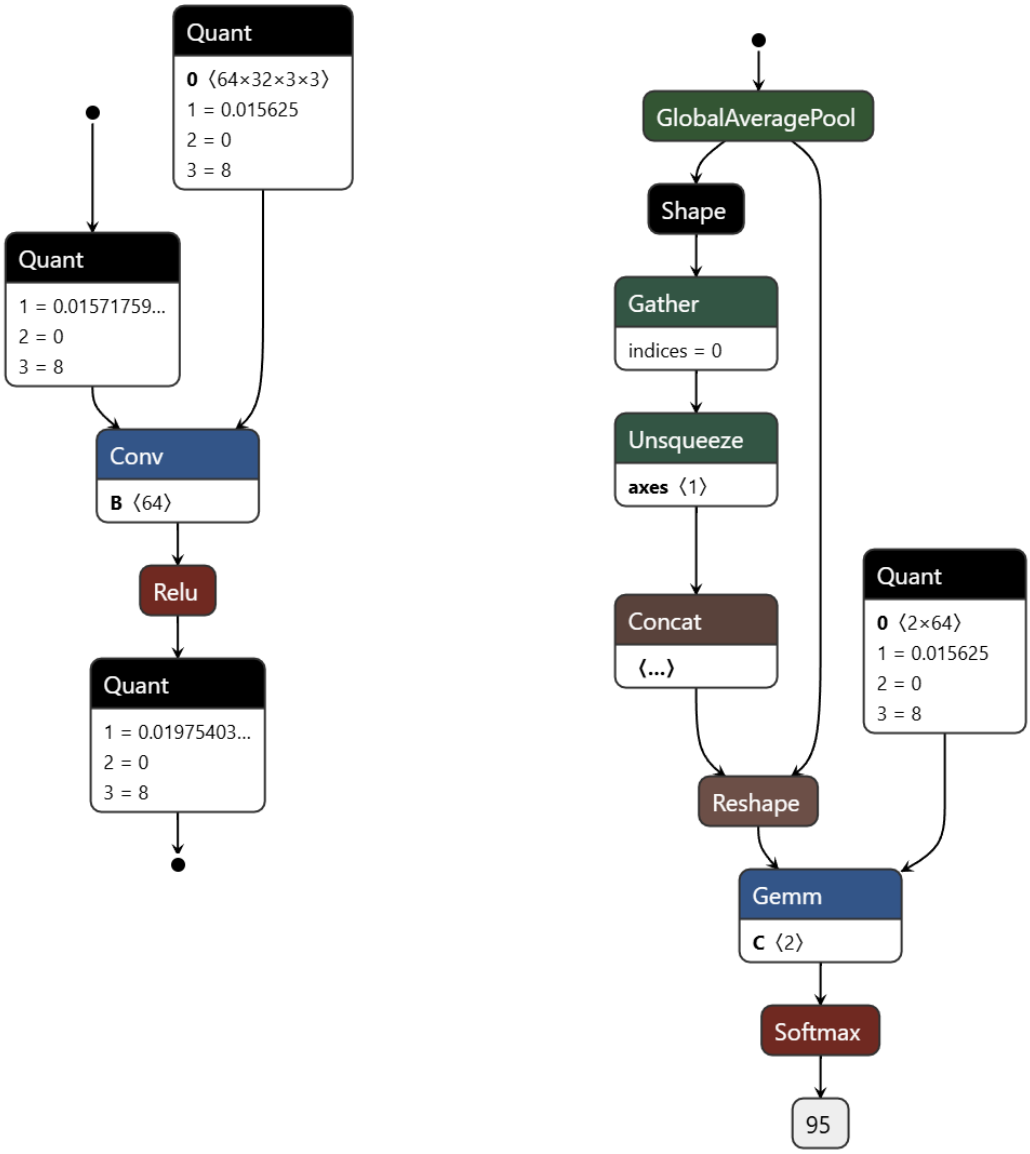


Figura 5.10: Visualización del modelo convertido a QONNX CNN128_Compact_k3 en Netron (Parte 2)

5.2.8. CNN128_Compact_k3_Pooling

Motivación: Reintroducir pooling para reducir la dimensionalidad, observando que en esta arquitectura (Tabla 5.7) al contrario que en la CNN512, el `MaxPooling` moderado no deterioraba el rendimiento.

Resultados: Exactitud incrementada hasta 94 %, manteniendo el número de parámetros (35 033). El `MaxPooling` en capas tempranas permitió reducir resolución intermedia sin pérdida de información crítica.

Exactitud por preprocesado:

- Grayscale: 93 %
- NDWI Modificado: 94 %
- NDWI Modificado + Detección de costa: 92 %

Capa	**Descripción**
ConvBlock1	Conv(1→16), 3×3 + ReLU + Conv(16→16), stride 2 + MaxPool
ConvBlock2	Conv(16→32), 3×3 + ReLU + Conv(32→32), stride 2 + MaxPool
ConvBlock3	Conv(32→64), 3×3, stride 2 + ReLU + MaxPool
AdaptiveAvgPool2D	
QuantLinear (64→2)	

Tabla 5.7: Capas de la arquitectura `CNN128_Compact_k3_Pooling` (35 033 parámetros)

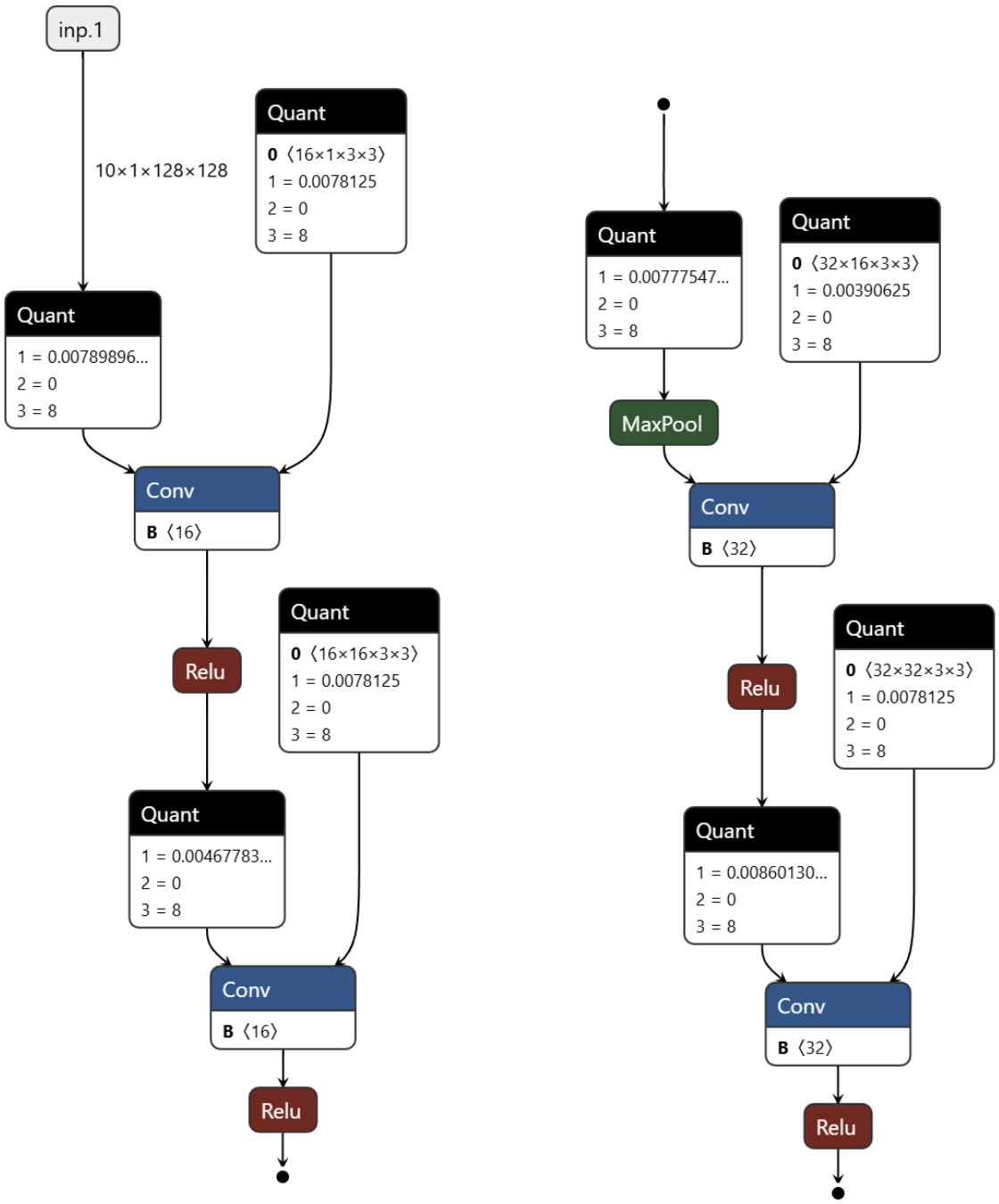


Figura 5.11: Visualización del modelo convertido a QONNX CNN128_Compact_k3_Pooling en Netron (Parte 1)

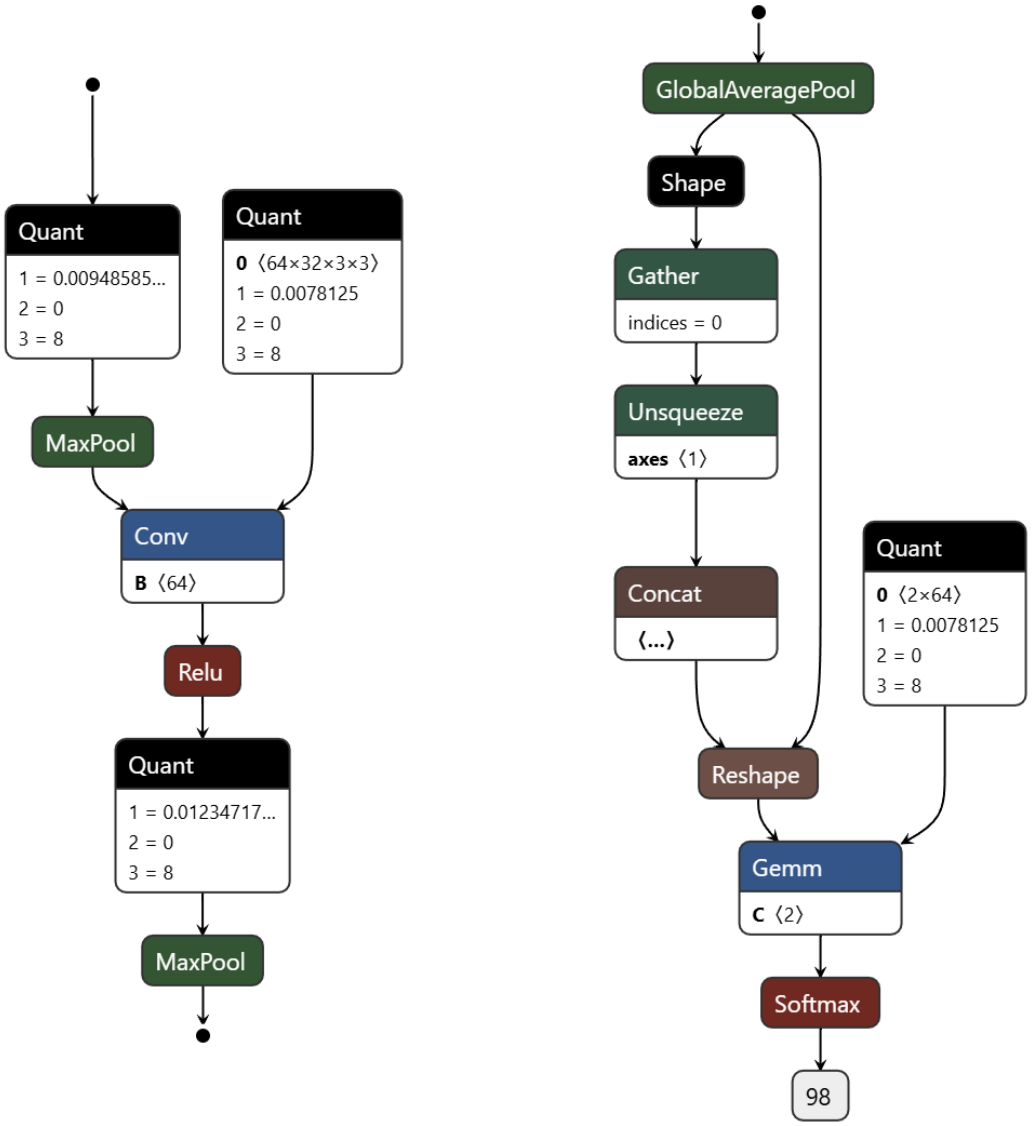


Figura 5.12: Visualización del modelo convertido a QONNX CNN128_Compact_k3_Pooling en Netron (Parte 2)

5.2.9. CNN128_UltraCompact_k3_Pooling

Motivación: Buscar una arquitectura (Tabla 5.8) aún más ligera, basada en el éxito del modelo anterior, pero reduciendo el número de canales.

Resultados: Con tan solo 8 897 parámetros, la reducción de la resolución de las capas convolucionales no afectó la exactitud esta arquitectura manteniendo un 94 %.

Exactitud por preprocesado:

- Grayscale: 93 %
- NDWI Modificado: 94 %
- NDWI Modificado + Detección de costa: 92 %

Capa	**Descripción**
ConvBlock1	Conv(1→8) + ReLU + Conv(8→8) + ReLU + MaxPool
ConvBlock2	Conv(8→16) + ReLU + Conv(16→16) + ReLU + MaxPool
ConvBlock3	Conv(16→32) + ReLU + MaxPool
AdaptiveAvgPool2D	
QuantLinear (32→2)	

Tabla 5.8: Capas del modelo `CNN128_UltraCompact_k3_Pooling` (8 897 parámetros)

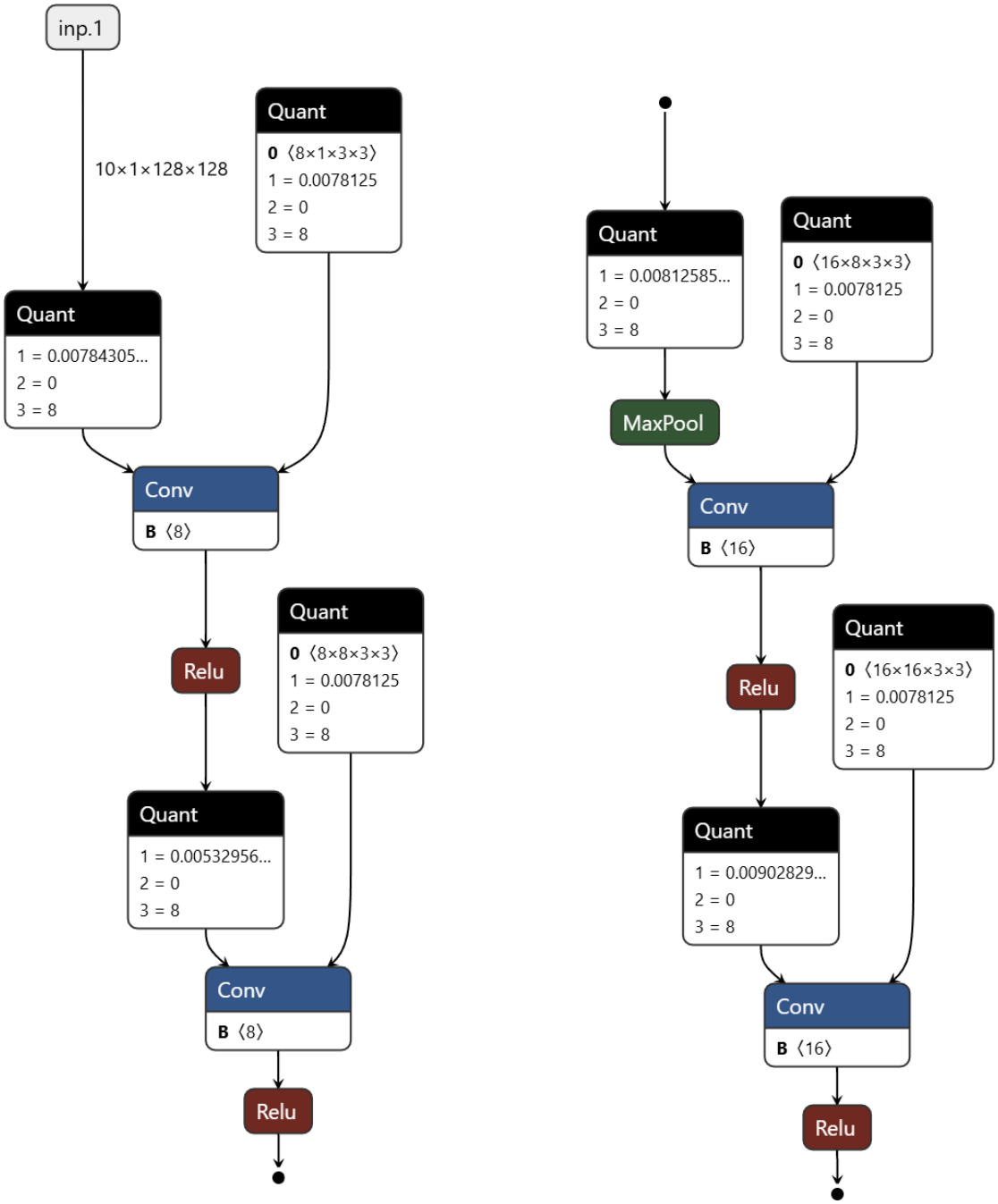


Figura 5.13: Visualización del modelo convertido a QONNX CNN128_UltraCompact_k3_Pooling en Netron (Parte 1)

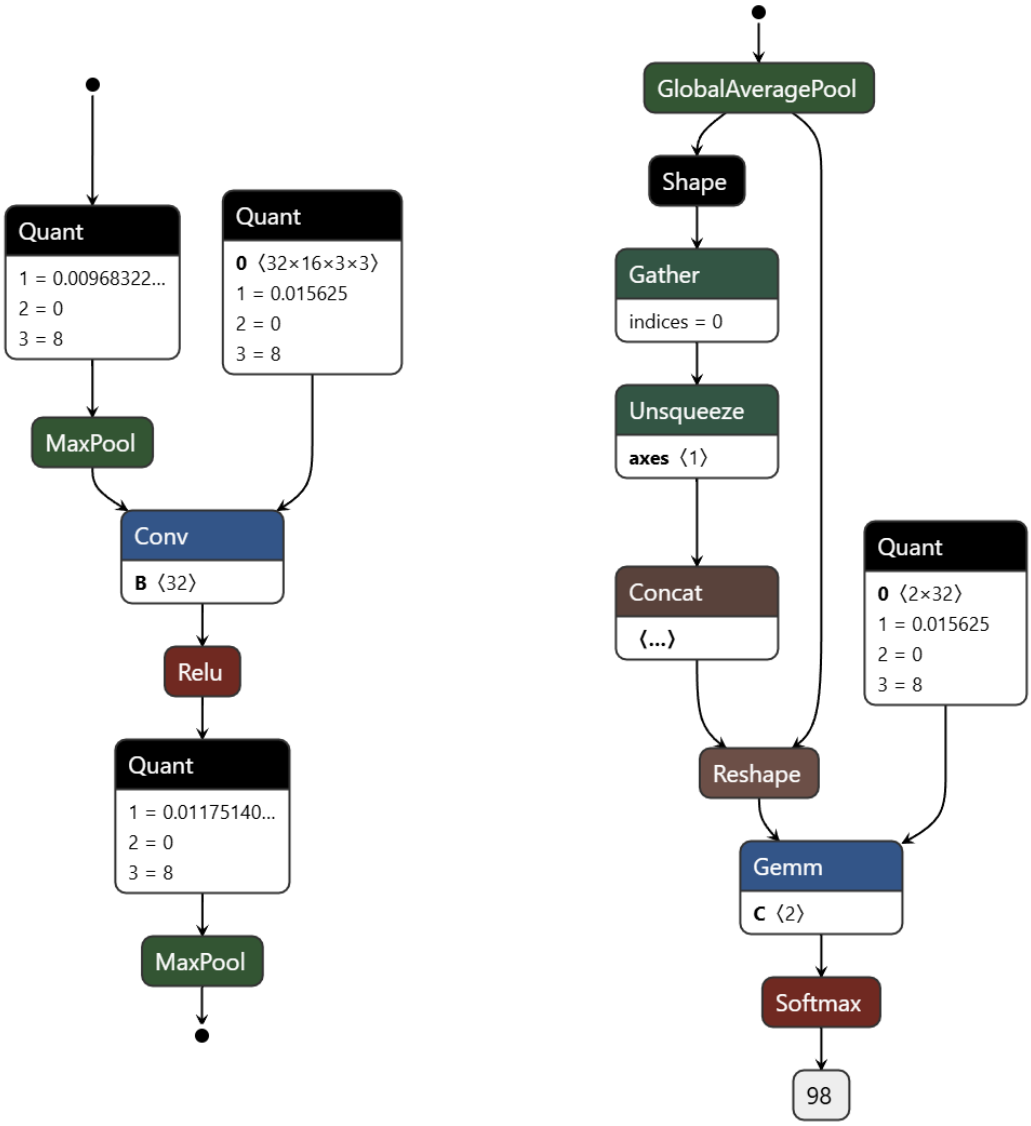


Figura 5.14: Visualización del modelo convertido a QONNX CNN128_UltraCompact_k3_Pooling en Netron (Parte 2)

5.3. Evaluación comparativa y selección de la arquitectura final

Con el objetivo de seleccionar la mejor arquitectura para su implementación final en una FPGA, se evaluaron todas las variantes desarrolladas atendiendo a dos criterios principales: exactitud obtenida y número total de parámetros.

La Figura 5.15 muestra la comparación entre los distintos modelos considerando su exactitud máxima alcanzada (entre los tres métodos de preprocesado evaluados) frente al número de parámetros de cada red.

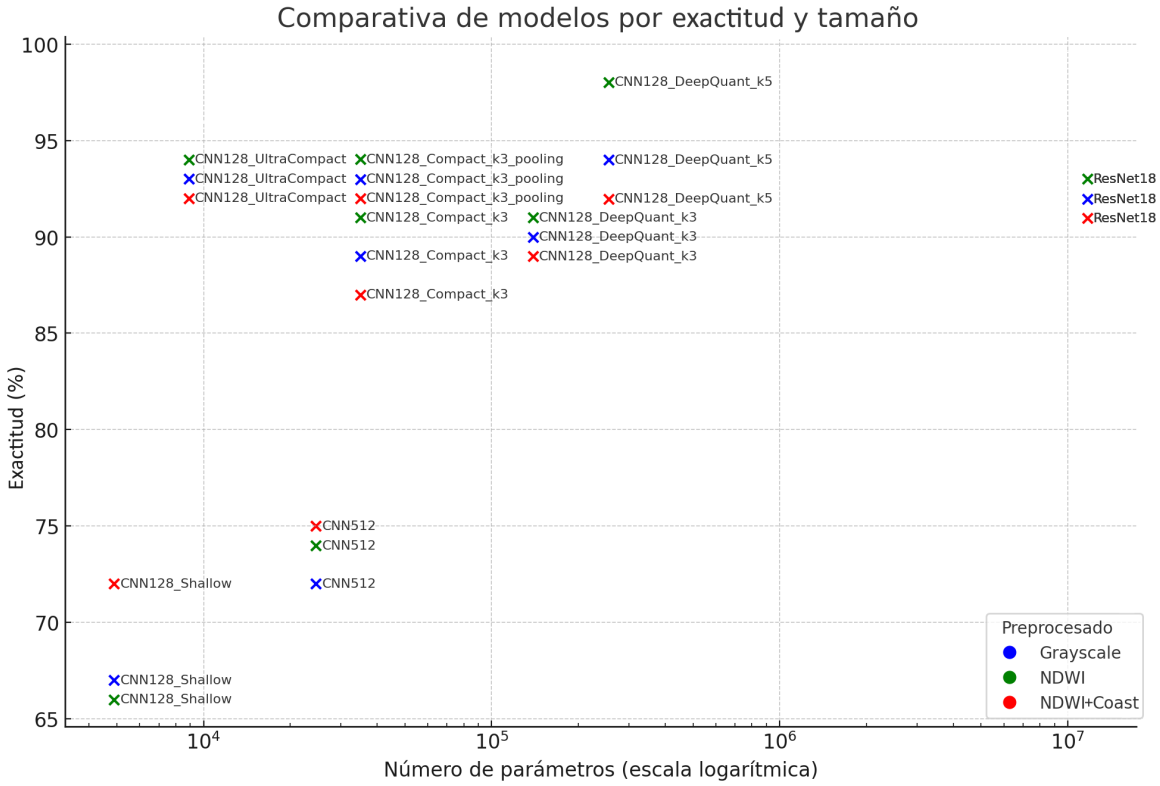


Figura 5.15: Relación entre número de parámetros y exactitud máxima alcanzada por cada arquitectura

Como puede observarse, aunque algunas arquitecturas como `CNN128_DeepQuant_k5` alcanzan una exactitud elevada (hasta 97 %), lo hacen a costa de un número de parámetros excesivo para una implementación eficiente en hardware. Por otro lado, modelos más ligeros como `CNN128_Shallow` o `CNN512` no alcanzan niveles de exactitud aceptables para el sistema objetivo.

La arquitectura que mejor equilibra ambos factores es `CNN128_UltraCompact_k3_Pooling`, que alcanza una exactitud del 94 % utilizando tan solo 8 897 parámetros. Esto representa una reducción de varios órdenes de magnitud respecto a la red de partida, `ResNet18`, manteniendo al mismo tiempo e incluso superando ligeramente su rendimiento. Por su eficiencia y exactitud, esta arquitectura fue seleccionada como modelo final para su implementación en la plataforma FPGA.

5.3.1. Elección del método de preprocesado

Durante la fase de pruebas se evaluaron tres métodos de preprocesado: escala de grises, `NDWI Modificado` y una versión con `NDWI Modificado + Detección de costa` de costa. La Figura 5.15 resume los resultados obtenidos para cada arquitectura y método.

Se observó que el preprocesado basado en `NDWI Modificado` generó de forma consistente mejoras de exactitud sobre la versión en escala de grises, especialmente en arquitecturas de mayor capacidad. Este comportamiento se puede atribuir a que la información espectral enfatizada ayuda a distinguir mejor masas de agua, incluso con imágenes de baja resolución espacial.

Por el contrario, el método `NDWI Modificado + Detección de costa` sólo fue beneficioso en redes menos profundas o con menor capacidad, como `CNN128_Shallow`. En arquitecturas más potentes, esta eliminación de información adicional no aportó ventajas significativas y en algunos casos incluso redujo la exactitud. Esto puede atribuirse a la limitada capacidad de representación de las redes más pequeñas, que se benefician de una reducción del ruido y del enfoque en las regiones más relevantes gracias al preprocesado. Sin embargo, en modelos de mayor capacidad, dicha simplificación puede resultar contraproducente; eliminar información espacial o contextual limita su potencial discriminativo.

Por tanto, el preprocesado seleccionado para el modelo final fue `NDWI Modificado`, por su robustez, simplicidad y mejores resultados en arquitecturas de alta exactitud.

Arquitectura	**Grayscale**	**NDWI Mod.**	**NDWI Mod. + Costa**
CNN128_Shallow	67 %	66 %	72 %
CNN128_Compact_k3	89 %	91 %	87 %
CNN128_Compact_k3_Pooling	93 %	94 %	92 %
CNN128_UltraCompact_k3_Pooling	93 %	**94 %**	92 %
CNN128_DeepQuant_k3	90 %	91 %	89 %
CNN128_DeepQuant_k5	94 %	97 %	92 %
CNN512	72 %	74 %	75 %

Tabla 5.9: Comparativa de exactitud según arquitectura y preprocesado

5.4. Proceso de entrenamiento y evaluación

Una vez seleccionada la arquitectura `CNN128_UltraCompact_k3_Pooling` como candidata final por su excelente equilibrio entre exactitud y eficiencia, se procedió al entrenamiento completo del modelo y a la evaluación de diferentes niveles de cuantización para maximizar la compatibilidad con dispositivos FPGA y reducir el coste computacional.

5.4.1. Configuración del entrenamiento

- **Optimizador: Adam**. Algoritmo adaptativo de primer orden que ajusta automáticamente la tasa de aprendizaje, adecuado para redes convolucionales pequeñas y medias.
- **Función de pérdida: CrossEntropyLoss**. Calcula la entropía cruzada entre la predicción del modelo y la etiqueta real, estándar para tareas de clasificación binaria o multiclase.
- **Épocas: 15**. Cada modelo fue entrenado durante 15 épocas completas.
- **Batch size: 64**. Tamaño del lote moderado, apropiado para mejorar la estabilidad del gradiente.
- **Dispositivo: GPU (Tesla k40)**. Todos los experimentos fueron realizados utilizando aceleración por GPU sobre el servidor de cómputo disponible.

5.4.2. División del dataset

Como ya se mencionó, el dataset se generó dividiendo las imágenes satelitales originales (512×512 píxeles) en *patches* de 128×128 con *stride* de 64, aplicando el método de preprocesado `NDWI Modificado`. Cada *patch* fue etiquetado como "barco" o "no-barco", formando un conjunto balanceado para entrenamiento, prueba y validación.

- **Conjunto de entrenamiento: 50.000 imágenes** utilizado para actualizar los pesos del modelo en cada época.
- **Conjunto de prueba: 10.000 imágenes** reservado para la evaluación parcial del modelo tras cada época, permitiendo monitorizar la exactitud durante el entrenamiento.
- **Conjunto de validación: 10.000 imágenes** usado únicamente tras finalizar el entrenamiento para medir la exactitud definitiva con cada configuración.

5.4.3. Flujo del entrenamiento

1. Se define la arquitectura `CNN128_UltraCompact_k3_Pooling` utilizando la librería `Brevitas`, configurando explícitamente el nivel de cuantización en cada experimento. Para ello, se entrenan versiones independientes del modelo con pesos y activaciones cuantizados a 4, 8 y 16 bits en formato de punto fijo, así como una versión sin cuantizar en punto flotante de 32 bits.
2. Se cargan los `patches` preprocesados con `NDWI Modificado` y se organizan mediante `DataLoader`.
3. Para cada época:
   a) Se entrena el modelo en modo `train()` sobre el conjunto de entrenamiento.
   b) Al finalizar, se mide el rendimiento en el conjunto de prueba.
4. Una vez completado el entrenamiento, se evalua el modelo en el conjunto de validación, desactivando `dropout` y usando modo `eval()`.

5.4.4. Comparativa de exactitud según cuantización

Cada configuración fue entrenada y evaluada de manera independiente. Los resultados de exactitud obtenidos en el conjunto de validación se muestran a continuación:

Formato	Tipo de exactitud	Exactitud alcanzada (NDWI Modificado)
`float32`	Punto flotante 32 bits	94.3 %
`fixed<16,6>`	Punto fijo 16 bits	94.4 %
`fixed<8,2>`	**Punto fijo 8 bits (seleccionado)**	**94.1 %**
`fixed<4,1>`	Punto fijo 4 bits	89.7 %

Tabla 5.10: Exactitud del modelo final entrenado desde cero para cada nivel de cuantización

5.4.5. Selección del nivel de cuantización

El nivel `fixed<8,2>` fue seleccionado como el más adecuado. Presenta la misma exactitud que las configuraciones en punto flotante y 16 bits, pero con una complejidad numérica significativamente menor. La alternativa en 4 bits, aunque eficiente, suponía una pérdida de exactitud del 5 porciento.

La figura 5.16 muestra el progreso de exactitud por época para la versión seleccionada en 8 bits, evidenciando una rápida convergencia y buena estabilidad durante el entrenamiento.

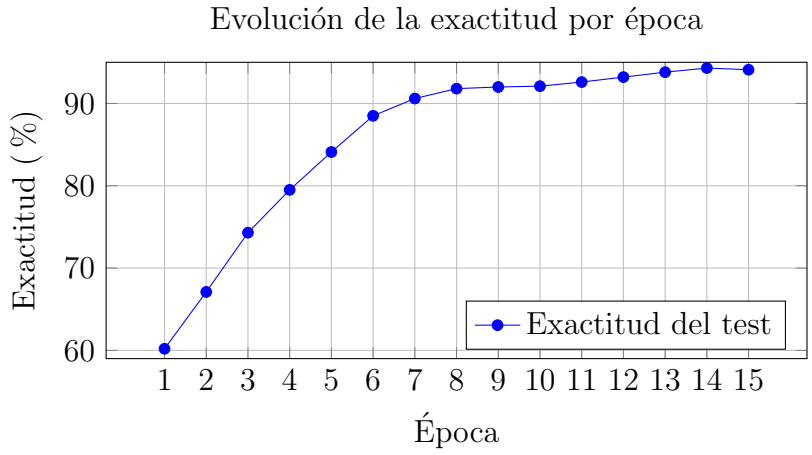


Figura 5.16: Exactitud del modelo cuantizado sobre el conjunto de prueba a lo largo de 14 épocas con extrapolación

5.5. Exportación del modelo a formato QONNX

Una vez completado el entrenamiento del modelo final (`CNN128_UltraCompact_k3_Pooling`) con cuantización a 8 bits, se procedió a su exportación al formato QONNX. Esto es necesario para permitir la compatibilidad con hls4ml, ya que dicha herramienta no soporta directamente los modelos cuantizados con `Brevitas`.

Flujo de exportación seguido:

1. **Exportación con Brevitas:** mediante la función `brevitas.export.export_qonnx()`, se generó el modelo ONNX incluyendo la configuración de cuantización (pesos y activaciones a 8 bits).

2. **Inferencia de formas (InferShapes):** se aplicó una transformación para deducir y fijar las dimensiones de todos los tensores del grafo, facilitando las etapas posteriores.

3. **Plegado de constantes (FoldConstants):** se eliminaron nodos constantes y redundantes, reduciendo la complejidad del modelo y mejorando la legibilidad.

4. **Conversión a formato `channels_last`:** se reorganizaron los tensores al formato `NHWC`, requerido por hls4ml para optimizar el paralelismo en hardware.

5. **Reemplazo de GEMM por MatMul:** la operación `GEMM` se reemplazó por `MatMul` para garantizar compatibilidad y facilitar la traducción en hardware.

El modelo limpio y optimizado fue guardado como `clean_quant_model.onnx` y verificado visualmente en `Netron` (Figura 5.17 y Figura 5.18), confirmando la estructura de capas y la presencia de la cuantización.

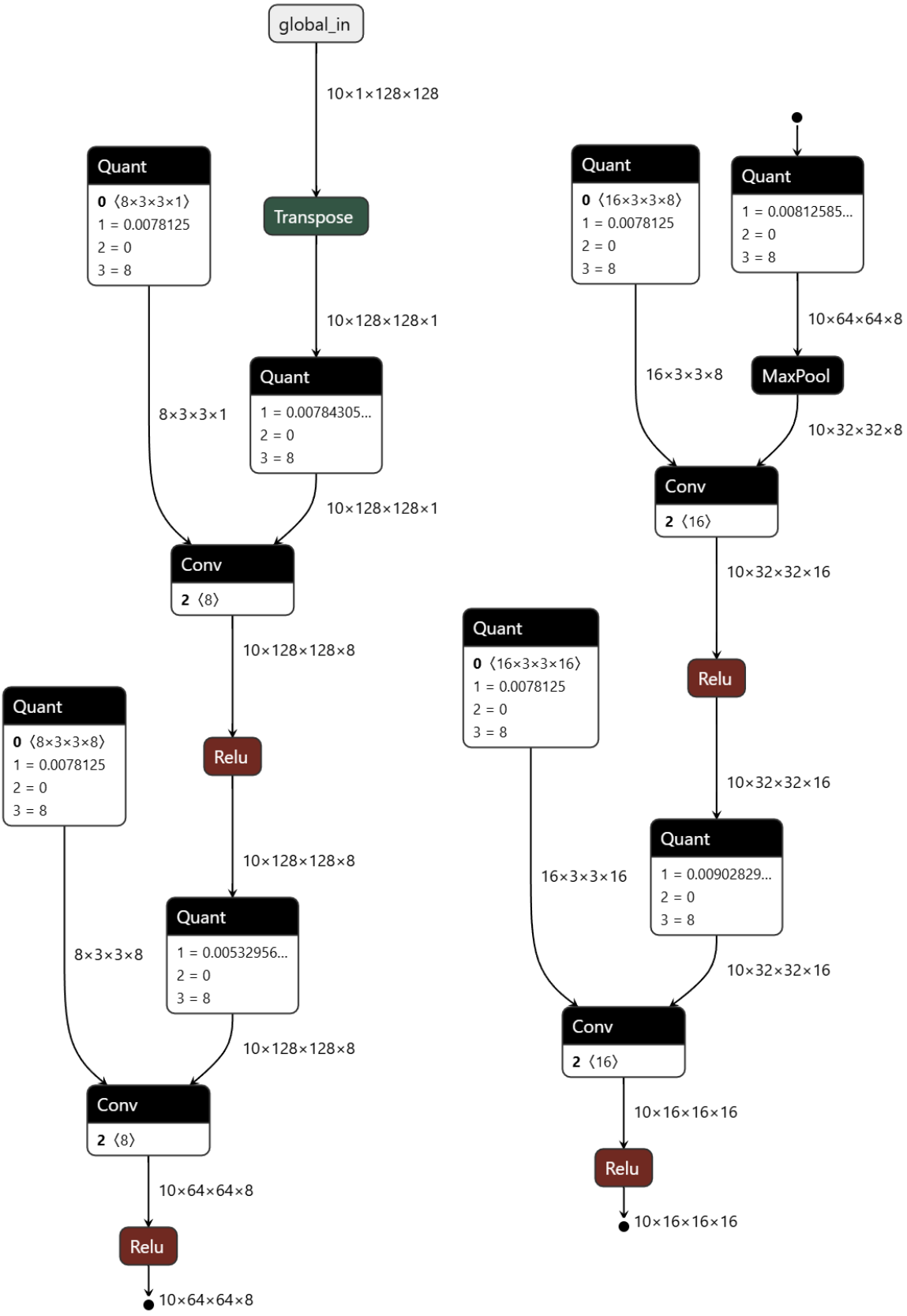


Figura 5.17: Visualización del modelo limpio y optimizado QONNX `CNN128_UltraCompact_k3_Pooling` en Netron (Parte 1)

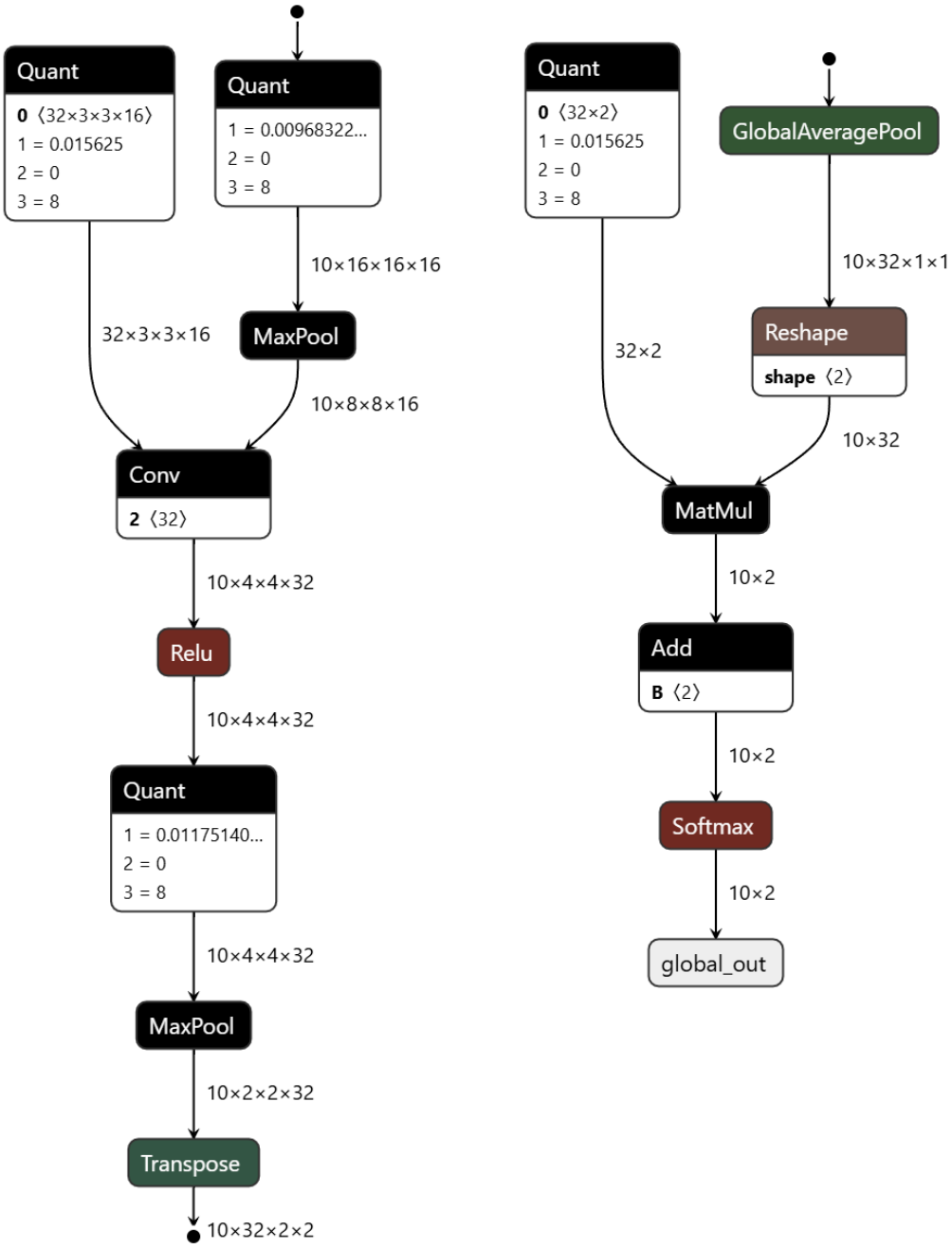


Figura 5.18: Visualización del modelo limpio y optimizado QONNX `CNN128_UltraCompact_k3_Pooling` en Netron (Parte 2)

5.6. Generación del IP y bitstream mediante hls4ml y Vitis HLS

Con el modelo QONNX exportado, se procedió a su conversión en un núcleo IP sintetizable mediante hls4ml, dirigido a Vitis HLS como entorno de síntesis de alto nivel.

Pasos del flujo de síntesis:

1. **Renombrado de nodos:** se asignaron nombres a los nodos anónimos del modelo ONNX para asegurar trazabilidad y evitar errores durante la generación del archivo de configuración.

2. **Generación del archivo de configuración:**

   Se utilizó la función `hls4ml.utils.config_from_onnx_model()` con los siguientes parámetros:

   - `granularity='name'`: configuración individual por capa.
   - `backend='Vitis'`: uso de Vitis HLS como motor de síntesis.
   - `default_precision='fixed<8,2>'`: exactitud predeterminada segura, aunque las capas ya están cuantizadas a 8 bits.

3. **Conversión a objeto hls4ml:** el modelo se transformó con `convert_from_onnx_model()`, configurado con:

   - `output_dir='hls_model/'`: carpeta de salida del proyecto.
   - `io_type='io_stream'`: entradas/salidas en flujo continuo.
   - `backend='Vitis'`: entorno de compilación.
   - `board='pynq-z1'`: FPGA de destino.
   - `hls_config=config`: archivo de configuración generado.

4. **Construcción con Vitis HLS:** se ejecutó el comando `hls_model.build(export=True, bitfile=True)`, que genera:

- Código fuente en C++ para cada capa del modelo.
- Proyecto completo de `Vitis HLS` listo para ser integrado con Vivado.
- Bitstream final (`.bit`) y archivos `.hwh` de descripción hardware.

5.7. Validación funcional del IP

Con el IP generado, se procedió a su validación funcional desplegando el modelo en la placa `PYNQ-Z1`. Para ello, se siguió el siguiente flujo:

Preparación de archivos:

- `hls4ml_nn.bit`: bitstream del diseño completo.
- `hls4ml_nn.hwh`: descripción hardware del diseño (metadatos del diseño generado).
- `deploy.py`: script de inferencia sobre la placa.
- `axi_stream_driver.py`: controlador para comunicación entre PS y PL vía AXI-Stream.
- `X_test.npy`, `y_test.npy`: entradas y etiquetas reales para la prueba.

Estos archivos fueron empaquetados y transferidos vía `scp` a la IP `192.168.2.99`, accediendo con las credenciales por defecto `xilinx:xilinx`.

Despliegue en FPGA:

1. Se carga el `bitstream` en la lógica programable usando el controlador `Overlay` de `PYNQ`.
2. Se instancia el controlador AXI-Stream y se configura el canal de comunicación entre el procesador y el IP.

3. Se cargan los datos de entrada y se ejecuta la inferencia completamente en hardware.

4. Se mide el rendimiento total en tiempo y número de inferencias por segundo.

Resultados de ejecución en hardware:

```
Classified 10000 samples in 68.963265 seconds (145 inferences/s)
```

Validación final:

Los resultados almacenados en `y_hw.npy` fueron comparados con las etiquetas de prueba. Se obtuvo una exactitud del **93.2 %** en hardware, valor muy cercano a la exactitud obtenida con el modelo cuantizado en entorno software (94.1 %). Las causas que justifican esta ligera diferencia han sido proporcionadas en el apartado 3.7 del capítulo 3.

Este resultado demuestra que el flujo completo es viable y funcional para implementar redes convolucionales ligeras y cuantizadas en dispositivos embebidos como FPGA.

5.8. Análisis de los resultados

El modelo final, `CNN128_UltraCompact_k3_Pooling`, alcanzó una exactitud del 94 % utilizando únicamente 8 897 parámetros, lo que representa una mejora de más del 99.95 % en reducción de parámetros respecto al modelo inicial `ResNet18`, sin una pérdida significativa de exactitud. Esta arquitectura fue seleccionada por ofrecer un equilibrio ideal entre rendimiento y eficiencia computacional, especialmente considerando las restricciones impuestas por la implementación en FPGAs.

Durante el proceso de evaluación, se observó que el modelo mantenía una exactitud del 94 % tanto en punto flotante como en formatos cuantizados de 16 y 8 bits. Sin embargo, al reducir la precisión a 4 bits, la exactitud descendió hasta el 89 %, lo cual

motivó la elección del formato `fixed<8,2>` como el compromiso óptimo entre eficiencia de recursos y fidelidad en la inferencia.

5.8.1. Rendimiento del acelerador en FPGA

Tras la exportación del modelo a QONNX y su posterior conversión con `HLS4ML`, se procedió a su síntesis utilizando `Vitis HLS` sobre la FPGA PYNQ-Z1.en para evaluar la escalabilidad del diseño.

El sistema fue sintetizado con una frecuencia objetivo de `200 MHz` (`5ns` de período de reloj), lo que se tradujo en un rendimiento de 145 inferencias por segundo en la plataforma final.

Ocupación de recursos

Recurso FPGA	**Utilización**	**Disponible (PYNQ-Z1)**
LUTs (Look-Up Tables)	8 416	53 200
FFs (Flip-Flops)	13 227	106 400
BRAMs (Block RAM)	15	140
DSPs (Multiplicadores)	15	220

Tabla 5.11: Utilización de recursos tras la síntesis con `HLS4ML` y `Vitis HLS`

Los resultados de utilización de recursos son especialmente destacables considerando las limitaciones de la `PYNQ-Z1`. La arquitectura final emplea menos del 16 % de las LUTs, en torno al 12 % de los Flip-Flops disponibles y únicamente un 6.8 % de los bloques de memoria BRAM. Además, sólo se utilizan 15 DSPs de los 220 disponibles. Esta baja ocupación confirma la viabilidad del diseño incluso en FPGAs de gama baja, y sugiere un amplio margen para posibles extensiones futuras.

Frecuencia de reloj y latencia total

- **Frecuencia de reloj objetivo:** 200 MHz (`5 ns`).
- **Tasa de inferencias:** 145 inferencias/s.
- **Tiempo total de inferencia sobre 10 000 imágenes:** 68.96 s.

5.8.2. Integración del IP en el sistema embebido

El núcleo generado con `HLS4ML` fue integrado dentro del diseño hardware que se compone de los siguientes elementos:

- **ZYNQ Processing System (PS):** El procesador ARM Cortex-A9 de la `PYNQ-Z1` actúa como maestro del sistema. Está conectado a la memoria principal (DDR) y gestiona la comunicación con la lógica programable mediante buses AXI. Se habilitan los puertos `M_AXI_GP0` y `S_AXI_HP0` para la interfaz general de control y transferencia de datos de alto rendimiento, respectivamente.
- **Núcleo IP de inferencia (`myproject_axi_0`):** Es el bloque generado por `HLS4ML` a partir del modelo QONNX cuantizado. Se conecta al bus AXI-Lite para control (`S_AXI_LITE`) y a la interfaz `AXI-Stream` para recibir y enviar datos (`in_r`, `out_r`).
- **AXI DMA (`axi_dma_0`):** Módulo de transferencia directa de memoria (DMA) que gestiona el flujo de datos entre el procesador (PS) y el núcleo de inferencia (PL) sin intervención del procesador. Utiliza las interfaces:
  - `S_AXIS_S2MM` para escribir los resultados generados por el IP en memoria.
  - `M_AXIS_MM2S` para leer datos de entrada desde memoria y enviarlos al IP.
  - `S_AXI_LITE` para su configuración desde el PS.
- **Interconectores AXI:** Dos bloques `AXI Interconnect` (`ps7_axi_periph` y `axi_mem_intercon`) permiten gestionar múltiples conexiones entre el PS y la PL, separando el tráfico de control (AXI-Lite) y el de datos (AXI-Full).

- **Módulo de Reset:** El bloque `Processor System Reset` gestiona las señales de reinicio sincronizadas para garantizar la correcta inicialización del sistema.

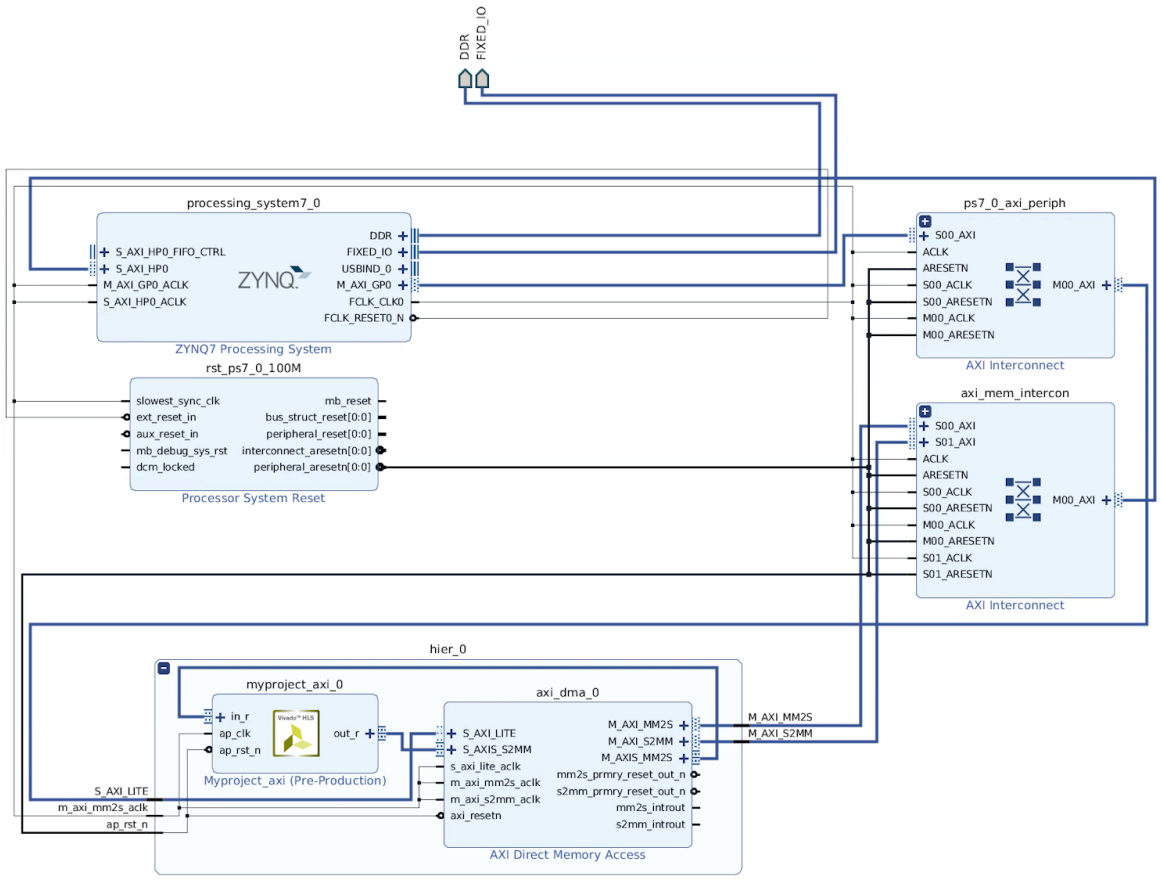


Figura 5.19: Diagrama de integración del núcleo HLS4ML con el sistema embebido

El procesador de la `PYNQ-Z1` ejecuta un script en Python que se comunica con el núcleo y gestiona el flujo de datos usando el driver `axi_stream_driver.py`. Gracias a esta arquitectura, el sistema puede recibir imágenes, ejecutar inferencia en tiempo real y devolver la predicción con una latencia mínima.

En conjunto, los resultados obtenidos confirman la viabilidad de aplicar técnicas de aprendizaje profundo sobre FPGA para tareas de detección de objetos en imágenes satelitales, cumpliendo con los requisitos de eficiencia, exactitud y ejecución en tiempo real que imponen los sistemas espaciales embarcados.

Capítulo 6

Discusión y Conclusiones

En este capítulo se presentan las conclusiones obtenidas tras el desarrollo completo del proyecto, así como una discusión crítica de los resultados alcanzados y de las decisiones adoptadas a lo largo del proceso. Se identifican también las principales limitaciones del enfoque seguido, tanto a nivel técnico como metodológico, y se proponen posibles líneas de mejora futura que podrían ampliar la aplicabilidad y rendimiento del sistema en contextos reales.

El objetivo es ofrecer una visión global del trabajo realizado, evaluando el impacto de cada una de las etapas desde el diseño del modelo hasta su despliegue en hardware y valorando su relevancia dentro del marco de sistemas espaciales inteligentes y autónomos.

6.1. Conclusiones

Este trabajo ha demostrado la viabilidad de implementar modelos de redes neuronales convolucionales (CNN) cuantizadas en plataformas FPGA, orientadas al procesamiento en tiempo real de imágenes satelitales. La motivación principal fue reducir la dependencia de los sistemas espaciales en la comunicación con estaciones en tierra, permitiendo una mayor autonomía a bordo para tareas de detección. Esto supone una ventaja significativa en escenarios con recursos limitados, tanto a nivel computacional como de ancho de banda y consumo energético.

Se ha desarrollado un flujo de trabajo completo, desde la definición del modelo hasta su implementación física en la FPGA PYNQ-Z1. A través del uso de Brevitas, QONNX y hls4ml, se ha logrado convertir una arquitectura optimizada en un diseño sintetizable, validada tanto por simulación como por pruebas reales en hardware.

6.1.1. Discusión de resultados

A lo largo del desarrollo se exploraron múltiples arquitecturas CNN aplicadas a imágenes satelitales, comenzando con una red `ResNet18` como punto de partida. Aunque este modelo alcanzó una exactitud del 93 %, su elevado número de parámetros (más de 18 millones) lo hacía inviable para su implementación directa en una FPGA de recursos limitados.

Este obstáculo motivó un cambio de paradigma hacia un enfoque basado en **división por *patches***, donde cada imagen de entrada se fracciona en regiones locales de 128×128 píxeles con *stride* 64. Este nuevo enfoque aportó varias ventajas clave:

- **Reducción significativa del tamaño de entrada**, permitiendo el uso de modelos mucho más compactos y eficientes.
- **Mejora en la capacidad de detección local**, al centrarse cada patch en una región reducida, favoreciendo la identificación de objetos pequeños como barcos.

- **Estimación espacial implícita**, ya que la detección en múltiples patches contiguos permite inferir la posición aproximada de los objetos detectados en la imagen original, sin necesidad de arquitecturas específicas de segmentación ni bounding boxes.

No obstante, este enfoque también implica una contraprestación importante: debido a la superposición entre patches (causada por el *stride* reducido), cada imagen genera 49 regiones a procesar. Esto incrementa notablemente el número total de inferencias requeridas por imagen completa, lo cual puede afectar negativamente la latencia total del sistema si no se dispone de suficiente paralelismo o aceleración.

En cuanto al diseño del modelo, se evaluaron múltiples variantes arquitectónicas en función del número de parámetros y la exactitud alcanzada (*accuracy*). La Figura 5.15 mostró cómo el modelo `CNN128_UltraCompact_k3_Pooling`, con sólo 8 897 parámetros, alcanzó un 94 % de exactitud sobre el conjunto de validación, lo que representa una reducción superior al **99.95 %** en número de parámetros respecto a `ResNet18`, manteniendo un rendimiento competitivo.

Respecto al preprocesado, se compararon tres variantes: `Grayscale`, `NDWI modificado` y `NDWI modificado + detección de costa`. El método `NDWI` se consolidó como el más robusto en redes de alta capacidad, mejorando la discriminación entre mar y objetos flotantes. En cambio, la eliminación de costa solo fue efectiva en arquitecturas menos profundas, donde el modelo requería asistencia externa para eliminar falsos positivos.

Asimismo, se evaluaron distintos niveles de cuantización: punto flotante (`float32`), y punto fijo a 16, 8 y 4 bits. Se observó que tanto la representación en `fixed<16,6>` como `fixed<8,2>` conservaron una exactitud del 94 %, mientras que el modelo cuantizado a 4 bits descendió al 89 %. Por tanto, el formato de 8 bits fue seleccionado como configuración final por representar el mejor compromiso entre eficiencia de recursos y fidelidad en la inferencia.

Finalmente, el sistema fue sintetizado con `HLS4ML` y desplegado físicamente sobre una FPGA `PYNQ-Z1`, alcanzando una tasa de procesamiento de 145 inferencias por segundo

y una latencia total de 68.96 segundos para procesar 10 000 patches. La exactitud obtenida en la FPGA fue del **93.2 %**, prácticamente idéntica a la del modelo cuantizado base, lo que valida tanto la robustez del modelo como la fidelidad del flujo de síntesis utilizado.

6.1.2. Limitaciones del enfoque

A pesar de los resultados positivos, el enfoque seguido presenta algunas limitaciones:

- **Simplificación del problema**: El sistema está diseñado para una tarea de clasificación binaria (barco/no-barco) sobre imágenes satelitales presegmentadas. No se consideraron múltiples clases.
- **Preprocesado fuera de la FPGA**: Aunque la red neuronal se ejecuta a bordo, el preprocesamiento (como el cálculo de NDWI) se realiza externamente, lo que limita la autonomía completa del sistema.
- **Tamaño fijo de patches**: El enfoque por ventanas deslizantes con tamaño fijo podría resultar ineficiente en imágenes con estructuras más complejas o embarcaciones de gran tamaño.
- **Escalabilidad**: Aunque la arquitectura está optimizada, su integración en un sistema espacial real requeriría mecanismos adicionales como tolerancia a fallos, gestión térmica y adaptación a radiación.

6.2. Posibles mejoras futuras

A partir de los resultados y limitaciones detectadas, se proponen las siguientes líneas de trabajo para futuras mejoras:

- **Implementación del preprocesado a bordo**: Integrar el cálculo del NDWI directamente en el sistema embebido permitiría una inferencia completamente autónoma. Dado que las operaciones necesarias son computacionalmente simples, este preprocesado podría implementarse:
  - En el procesador ARM del sistema (`PS`), si se dispone de suficiente capacidad de cómputo.
  - O como un módulo hardware adicional en la FPGA (`PL`), si se desea minimizar latencia o consumo.
- **Extensión a detección multiclase o segmentación**: Ampliar el modelo para incluir detección multiclase o técnicas de segmentación permitiría una clasificación más detallada, así como una estimación más precisa de la forma y contorno de los objetos detectados (por ejemplo, mediante *semantic segmentation* o *bounding boxes*).
- **Uso de técnicas de *pruning***: La aplicación de técnicas de poda estructural podría reducir aún más el número de parámetros del modelo sin pérdida significativa de rendimiento, optimizando aún más la ocupación de recursos hardware.
- **Síntesis en FPGAs espaciales**: Evaluar la implementación sobre plataformas de grado espacial, como las FPGAs `Xilinx Kintex UltraScale` o `Microsemi RTG4`, permitiría validar su uso en entornos reales de vuelo. Esto implicaría además adaptar el diseño a los flujos de desarrollo y restricciones propias de estas plataformas.
- **Tolerancia a fallos mediante TMR**: Considerando el entorno espacial hostil, sería recomendable implementar mecanismos de tolerancia a fallos como Triple Modular Redundancy (TMR), replicando componentes críticos del acelerador e incorporando lógica de votación.

- **Integración con comunicaciones satélite-tierra**: Una futura línea de trabajo consistiría en integrar el sistema de inferencia con el gestor de comunicaciones del satélite, priorizando la transmisión de imágenes en las que se haya detectado algún objeto de interés (como embarcaciones). Esto permitiría optimizar el uso del limitado ancho de banda disponible en el enlace descendente, reduciendo la carga de datos y acelerando la respuesta operativa.

En resumen, el trabajo ha demostrado con éxito que es posible implementar una red neuronal eficiente, precisa y funcional en una FPGA, abriendo el camino hacia sistemas de percepción embarcados más autónomos y eficientes en plataformas espaciales.

Anexo A

Presupuesto

Habiendo completado satisfactoriamente el diseño propuesto, ahora se realiza un análisis de los costes económicos del presente Trabajo de Fin de Grado (TFG). Dichos costes se desglosan en los siguientes apartados:

- Recursos humanos.
- Recursos materiales.
  - Recursos *hardware*.
  - Recursos *software*.
- Redacción del documento.
- Derechos de visado del Colegio Oficial de Ingenieros Técnicos de Telecomunicaciones (COITT).
- Gastos de tramitación y envío.
- Material fungible.
- Presupuesto final (con impuestos).

A.1. Recursos humanos

El coste de los recursos humanos se corresponde con los honorarios que recibe el ingeniero a cargo del desarrollo del proyecto considerando el salario correspondiente a un graduado en ingeniería en tecnologías de la telecomunicación. Contemplando que el proyecto propuesto se ha realizado en la Universidad de las Palmas de Gran Canaria (ULPGC), la tarifa aplicada es la asociada al personal técnico con titulación mínima exigida de graduado o equivalente, de acuerdo con Boletín Oficial de la ULPGC (ULPGC). El resultado del coste de los recursos humanos se halla en la Tabla A.1.

Personal	Coste total mensual (€)	Tiempo	Total (€)
Ingeniero técnico	760,05	4 meses	3040,20

Tabla A.1: Trabajo tarifado por tiempo empleado.

El coste final del trabajo tarifado por tiempo empleado es de TRES MIL CUARENTA EUROS CON VEINTE CÉNTIMOS.

A.2. Recursos materiales

Para el cálculo del coste relativo a los recursos materiales empleados se tiene en cuenta un periodo de amortización de tres años de carácter lineal. El cálculo de la cuota de amortización anual se realiza mediante la siguiente expresión siendo C la cuota de amortización anual, V_{ad} el valor de adquisición, V_{res} el valor residual y N el número de años considerados para la amortización de cada producto. Nótese que el valor residual se supone 0 € para todos los casos.

$$C = \frac{V_{ad} - V_{res}}{N}$$

A.2.1. Recursos *software*

Dado que todos los recursos *software* empleados en el desarrollo del trabajo (listados más abajo) son directamente gratuitos, ofrecen versiones gratuitas para estudiantes o

se encuentran disponibles para su uso en los laboratorios de la ULPGC, el coste total asociado a los recursos *software* es de CERO EUROS.

- **Google Colab**: entorno en la nube gratuito que permite ejecutar código Python con acceso a GPU.
- **Python + bibliotecas auxiliares** (`numpy`, `matplotlib`, `torchvision`, `onnx`, etc.).
- **PyTorch**: biblioteca de aprendizaje profundo de código abierto.
- **Brevitas**: librería para el diseño de redes cuantizadas sobre PyTorch.
- **QONNX**: herramienta para la exportación de modelos cuantizados a formato ONNX compatible con High-Level Synthesis for Machine Learning (hls4ml).
- **HLS4ML**: framework open-source para la conversión de modelos de redes neuronales en diseños sintetizables mediante HLS.
- **Vitis HLS**: herramienta de síntesis de alto nivel de Xilinx (licencia gratuita para estudiantes).
- **Netron**: visualizador de modelos `ONNX` y arquitecturas de redes neuronales.
- **Entorno LaTeX** para la redacción del documento.
- **Zotero**: herramienta de gestión de referencias bibliográficas.

A.2.2. Recursos *hardware*

Como el trabajo se ha elaborado en un periodo inferior al estipulado para la amortización, se realiza una amortización equiparable al periodo de duración del mismo. En la Tabla A.2 aparece el coste total de los recursos *hardware*.

El coste total asociado a los recursos *hardware* es de TRESCIENTOS CUARENTA Y CUATRO EUROS CON CUARENTA Y CINCO CÉNTIMOS.

Descripción	Uds.	Tiempo de uso	V_{ad} (€)	Coste anual (€)	Coste final (€)
PC personal	1	4 meses	1.700,00	566,67	188,89
Estación de trabajo del laboratorio	1	4 meses	1.400,00	466,67	155,56
Total					**344,45**

Tabla A.2: Amortización de los recursos *hardware*.

A.3. Redacción del documento

El coste de la redacción del documento se calcula según la siguiente expresión donde R representa dichos costes, P es el presupuesto y C_n es el coeficiente de ponderación del presupuesto. Como el coste total del proyecto no supera los 30.050,00 €, C_n se supone de valor unitario.

$$R = 0,07 \cdot P \cdot C_n$$

Por otro lado, el presupuesto se calcula como la suma del coste de las amortizaciones y los costes del trabajo tarifado por tiempo empleado. En la Tabla A.3 se halla el resultado de esta operación.

Descripción	Coste (€)
Coste tarifado por tiempo empleado	3.040,20
Amortización de los recursos *software*	0,00
Amortización de los recursos *hardware*	344,45
Total	**3384,65**

Tabla A.3: Total de las amortizaciones y trabajo tarifado por tiempo empleado.

Una vez hallado el presupuesto, finalmente pueden determinarse los costes asociados a la redacción del documento aplicando la expresión previamente expuesta:

$$R = 0,07 \cdot P \cdot C_n = 0,07 \cdot 3384,65 \cdot 1 \approx 236,92 \text{ €}$$

El coste derivado de la redacción del documento es de DOSCIENTOS TREINTA Y SEIS EUROS CON NOVENTA Y DOS CÉNTIMOS.

A.4. Derechos de visado del COITT

En cuanto a los derechos de visado del COITT, estos pueden calcularse aplicando la siguiente expresión según **coitt**:

$$V = 0,007 \cdot P \cdot C_r$$

Nótese que V hace alusión a los costes de los derechos de visado, P al presupuesto de ejecución sin impuestos[1] y C_r al coeficiente reductor. Por ser $P < 6000$ €, el valor de C_r es 1. Asimismo, en **coitt** se menciona que el mínimo importe para los derechos de visado de un documento de estas características es de 40 €. Finalmente, los costes de los derechos de visado son:

$$V = 0,007 \cdot P \cdot C_r = 0,007 \cdot (3384,65 + 236,92) \cdot 1 = 25,35 \text{ €} \rightarrow V = 40 \text{ €}$$

El coste de los derechos de visado es de CUARENTA EUROS.

A.5. Gastos de administración

De acuerdo con **coitt**, los gastos de administración por documento son de nueve o doce euros dependiendo de si el visado es digital o manual, respectivamente. Por tanto, suponiendo visado digital, los gastos de administración para este documento ascienden a NUEVE EUROS.

A.6. Material fungible

Además de los costes expuestos previamente, en esta sección también se incluyen los asociados al material de papelería empleado durante el transcurso del proyecto, así como los derivados de la impresión de la memoria y su encuadernación. En la Tabla A.4 se muestran estos gastos.

Descripción	Coste (€)
Material de papelería	5,00
Impresión de la memoria	30,00
Encuadernación	5,00
Total	**40,00**

Tabla A.4: Total del material fungible.

[1] En el presupuesto de ejecución sin impuestos deben incluirse los costes de redacción del documento.

El coste total del material fungible es de CUARENTA EUROS.

A.7. Presupuesto final del proyecto

Al desarrollo del presente TFG se le aplica el Impuesto General Indirecto Canario (IGIC), el cual representa un siete por ciento del valor del presupuesto. En la Tabla A.5 se recoge el presupuesto final.

Descripción	**Coste (€)**
Coste tarifado por tiempo empleado	3.040,20
Amortización de los recursos *software*	0,00
Amortización de los recursos *hardware*	344,45
Redacción del documento	236,92
Derechos de visado del COITT	40,00
Gastos de administración	9,00
Material fungible	40,00
Subtotal	3710,57
Total (+7 % IGIC)	**3970,31**

Tabla A.5: Presupuesto final.

El valor del presupuesto final para el presente TFG es de TRES MIL NOVECIENTOS SETENTA EUROS CON TREINTA Y UN CÉNTIMOS.

Anexo B

Grado de relación con los Objetivos de Desarrollo Sostenible

Grado de relación con los Objetivos de Desarrollo Sostenible (ODS)

ODS	No procede **0**	Bajo **1**	Medio **2**	Alto **3**
ODS 1 Fin de la Pobreza	**X**			
ODS 2 Hambre cero	**X**			
ODS 3 Salud y Bienestar	**X**			
ODS 4 Educación de calidad	**X**			
ODS 5 Igualdad de género	**X**			
ODS 6 Agua limpia y saneamiento	**X**			
ODS 7 Energía asequible y no contaminante	**X**			
ODS 8 Trabajo decente y crecimiento económico	**X**			
ODS 9 Industria, Innovación e Infraestructuras				**X**
ODS 10 Reducción de las desigualdades	**X**			
ODS 11 Ciudades y comunidades sostenibles	**X**			
ODS 12 Producción y consumo sostenibles	**X**			
ODS 13 Acción por el clima	**X**			
ODS 14 Vida submarina	**X**			
ODS 15 Vida de ecosistemas terrestres	**X**			
ODS 16 Paz, justicia e instituciones sólidas	**X**			
ODS 17 Alianzas para lograr objetivos	**X**			

Tabla B.1: Grado de relación del Trabajo Fin de Título con los ODS

Justificación del alineamiento:

La presente propuesta introduce innovación en el ámbito de la inteligencia artificial embebida, al abordar la implementación eficiente de modelos de *deep learning*, en particular redes neuronales convolucionales (Redes Neuronales Convolucionales (CNNs)), sobre arquitecturas de hardware reconfigurable como las Field-Programmable Gate Arrays (FPGAs) (*Field-Programmable Gate Arrays*).

Tradicionalmente, los modelos de aprendizaje profundo se han ejecutado en servidores con unidades de procesamiento gráfico (GPUs) debido a su elevado requerimiento computacional. No obstante, este enfoque no es viable en sistemas embebidos con restricciones de consumo energético, espacio, latencia y tolerancia a fallos, como ocurre en entornos aeroespaciales o aplicaciones remotas en tiempo real. En este contexto, las FPGAs emergen como una alternativa atractiva, al permitir la creación de aceleradores hardware específicos para cada modelo, optimizados a nivel de precisión, paralelismo y consumo.

Este trabajo destaca por integrar múltiples técnicas innovadoras en dicho contexto:

- **Cuantización extrema de modelos CNN** para reducir drásticamente el uso de recursos hardware, manteniendo una alta precisión de inferencia.
- **Exportación del modelo cuantizado mediante QONNX**, facilitando su interoperabilidad con herramientas de síntesis de alto nivel.
- **Conversión del modelo a una arquitectura hardware mediante HLS4ML**, herramienta que traduce redes neuronales en código C++ sintetizable con herramientas como Vitis HLS.
- **Implementación y validación funcional en una plataforma real FPGA (PYNQ-Z1)**, demostrando viabilidad práctica, baja latencia y precisión comparable al modelo software.
- **Uso de preprocesado satelital especializado (NDWI)** para mejorar la capacidad de detección de barcos en imágenes multiespectrales.

La combinación de estos elementos permite abordar un problema real —la detección de objetos en imágenes satelitales— desde una perspectiva integradora que abarca

desde el diseño del modelo hasta su ejecución optimizada en hardware. Este enfoque no solo aporta una solución técnica eficiente, sino que también constituye un avance en el desarrollo de sistemas autónomos e inteligentes capaces de operar directamente a bordo, con aplicaciones potenciales en observación terrestre, defensa, vigilancia marítima o respuesta ante emergencias ambientales.

En conjunto, este trabajo se alinea con las tendencias emergentes en inteligencia artificial embebida y representa un ejemplo práctico de cómo las FPGAs pueden ser utilizadas como plataforma para desplegar modelos de *deep learning* en entornos con recursos limitados, contribuyendo al desarrollo de sistemas más sostenibles, autónomos y eficientes.

Anexo C

Código fuente: `shipDetection.py`

```
1
2
3  import inspect
4  import os
5  import numpy as np
6  import pathlib
7  import random
8  import matplotlib.pyplot as plt
9  import cv2
10
11 from typing import Set, Tuple, Dict, List, Union
12
13 from collections import defaultdict
14 from tqdm import tqdm
15 from collections import Counter
16 from PIL import Image
17 from torch.utils.data import Dataset
18 from torchvision import transforms
19 import torch
20 import torch.nn as nn
21 import torch.nn.functional as F
22 import torch.optim as optim
23 from torchvision import datasets, transforms
24 from torch.utils.data import DataLoader
25 import brevitas.nn as qnn
26 from brevitas.quant import Int8WeightPerTensorFixedPoint,
       Int8ActPerTensorFixedPoint
27 from torch.utils.data import WeightedRandomSampler
28 from torch.utils.data import random_split
29 from torch.utils.data import Subset
30 from collections import defaultdict
31
32 import hls4ml
33 import onnx
34 import onnxruntime
35 import onnxoptimizer
36 from qonnx.util.inference_cost import inference_cost
37 from qonnx.util.exec_qonnx import exec_qonnx
```

```
38 from brevitas.export import export_onnx_qcdq, export_qonnx
39 from qonnx.core.modelwrapper import ModelWrapper
40 import netron
41
42 from qonnx.util.cleanup import cleanup_model
43 from qonnx.transformation.infer_shapes   import InferShapes
44 from qonnx.transformation.fold_constants import FoldConstants
45 from qonnx.transformation.channels_last  import
       ConvertToChannelsLastAndClean
46 from qonnx.transformation.gemm_to_matmul import GemmToMatMul
47
48 import warnings
49 #warnings.filterwarnings("ignore", category=UserWarning)
50
51 WRK_DIR = '/home/tfga/ShipDetectionCNN/'
52 CHECKPOINT_PATH = WRK_DIR + '/Model/checkpoints/'
53 SAVE_PATH = WRK_DIR + '/Model/final/'
54
55
56
57 """
58 ##############################
59 Helper Functions
60 ##############################
61
62 """
63
64
65 def fileCount(folder):
66     "count the number of files in a directory"
67
68     count = 0
69
70     for filename in os.listdir(folder):
71         path = os.path.join(folder, filename)
72
73         if os.path.isfile(path):
74             count += 1
75         elif os.path.isfolder(path):
```

```
76              count += fileCount(path)
77
78      return count
79
80  def walk_through_dir(dir_path):
81    """
82    Walks through dir_path returning its contents.
83    Args:
84      dir_path (str or pathlib.Path): target directory
85
86    Returns:
87      A print out of:
88        number of subdiretories in dir_path
89        number of images (files) in each subdirectory
90        name of each subdirectory
91    """
92    for dirpath, dirnames, filenames in os.walk(dir_path):
93      print(f"There are {len(dirnames)} directories and {len(filenames)}
          files in '{dirpath}'.")
94
95  def showSrc(what):
96      print("".join(inspect.getsourcelines(what)[0]))
97
98  def showInNetron(model_filename: str, localhost_url: str = None, port:
      int = None):
99      """Shows a ONNX model file using Netron.
100
101     :param model_filename: The path to the ONNX model file.
102     :type model_filename: str
103
104     :param localhost_url: The IP address used by the Jupyter IFrame to
          show the model.
105      Defaults to localhost.
106     :type localhost_url: str, optional
107
108     :param port: The port number used by Netron and the Jupyter IFrame
          to show
109      the ONNX model.  Defaults to 8081.
110     :type port: int, optional
```

```
111
112     :return: The IFrame displaying the ONNX model.
113     :rtype: IPython.lib.display.IFrame
114     """
115     try:
116         port = port or int(os.getenv("NETRON_PORT", default="8081"))
117     except ValueError:
118         port = 8081
119     localhost_url = localhost_url or os.getenv("LOCALHOST_URL",
            default="localhost")
120     netron.start(model_filename, address=("0.0.0.0", port), browse=
            False)
121
122 def show_random_samples(dataset, class_names=None, n=6, cols=3):
123     """
124     Display a few random images from a dataset (Dataset or Subset).
125
126     Args:
127         dataset: PyTorch Dataset or Subset
128         class_names: Optional list of class names (indexed by label)
129         n: Number of samples to display
130         cols: Number of columns in the plot grid
131     """
132     # Handle Subset datasets
133     if isinstance(dataset, torch.utils.data.Subset):
134         indices = random.sample(dataset.indices, min(n, len(dataset)))
135         samples = [dataset.dataset[i] for i in indices]
136     else:
137         indices = random.sample(range(len(dataset)), min(n, len(
                dataset)))
138         samples = [dataset[i] for i in indices]
139
140     rows = (n + cols - 1) // cols
141     plt.figure(figsize=(cols * 3, rows * 3))
142
143     for i, (img, label) in enumerate(samples):
144         plt.subplot(rows, cols, i + 1)
145
146         # Convert tensor to numpy if needed
```

```
147         if isinstance(img, torch.Tensor):
148             img = img.permute(1, 2, 0).numpy()  # CHW -> HWC
149         plt.imshow(img)
150         plt.axis('off')
151 
152         label_str = class_names[label] if class_names else str(label)
153         plt.title(f"Label: {label_str}")
154 
155     plt.tight_layout()
156     plt.savefig(WRK_DIR + "samples.png")
157 
158 """
159 ################################
160 Preprocessing Methods
161 ################################
162 """
163 
164 
165 def normalize_array(arr):
166   norm_arr = np.minimum(1, (arr - np.min(arr)) / (np.percentile(arr,
       95) - np.min(arr)))
167   return norm_arr
168 
169 def ndwi(pil_image):
170   image = np.array(pil_image)
171 
172   # Convert RGB to BGR
173   image = image[:, :, ::-1].copy()
174 
175   # grab the image dimensions
176   h = image.shape[0]
177   w = image.shape[1]
178   output = np.zeros((h, w), np.float32)
179 
180   # loop over the image, pixel by pixel
181   for y in range(0, h):
182     for x in range(0, w):
183       b, g, r = image[y, x] / 255
184       minV = min(image[y, x])
```

```
185         maxV = max(image[y, x])
186         l = (b + g + r)/3
187 
188         if ((g == 0 and r == 0) or (b == 0 and r == 0)):
189           output[y, x] = 0
190         else:
191           output[y, x] = min(255, 255 * (1 - (float(b) - float(r)**2) /
                (float(b) + float(r)**2))**2) * (1 - (1 - l**2) * abs(b -
                r))**2
192 
193   return Image.fromarray(output)
194 
195 def ndwi_coast_removal(pil_image):
196   image = np.array(pil_image)
197 
198   # Convert RGB to BGR
199   image = image[:, :, ::-1].copy()
200 
201   image = cv2.GaussianBlur(image,(7,7),0)
202   # grab the image dimensions
203   h = image.shape[0]
204   w = image.shape[1]
205   layerN = np.zeros((h,w), np.float32)
206   ndwi_prediction = np.zeros((h,w), np.uint8)
207 
208   # loop over the image, pixel by pixel
209   for y in range(0, h):
210     for x in range(0, w):
211       b, g, r = image[y, x]/255.
212       minV = min(image[y, x])
213       maxV = max(image[y, x])
214       l = (b + g + r)/3
215 
216       if np.count_nonzero(image[y, x]) == 0:
217         layerN[y, x] = 0
218       else:
219         layerN[y, x] = max(0, ((float(b) - float(r)**2) / (float(b) +
              float(r)**2))) * max(0, ((float(g) - float(r)) / (float(g)
               + float(r)))) * (float(b)/(float(g) + float(b)))**5 * (1
```

```
            - l**2)**3

    layerN = normalize_array(layerN)
    ndwi_prediction = (np.minimum(255, layerN * 255)).astype(np.uint8)

    ############

    pooling = poolingOverlap(ndwi_prediction, (8, 8), None, 'mean', True
        ).astype(np.uint8)
    pooling = cv2.blur(pooling, (3,3))
    (_, maxVal, _, maxLoc) = cv2.minMaxLoc(pooling)

    _, pooling = cv2.threshold(pooling,32,255,cv2.THRESH_BINARY)
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(4,4))
    pooling = cv2.morphologyEx(pooling, cv2.MORPH_ERODE, kernel)

    ocean_percent = pooling.mean()/255.
    if ocean_percent > 0.95 or ocean_percent < 0.05:
      landmasses = pooling

    else:
      landmasses = cv2.bitwise_not(pooling.copy())
      cv2.floodFill(landmasses, None, maxLoc, 255)
      landmasses = cv2.bitwise_and(landmasses, pooling)

    landmasses = cv2.resize(landmasses, (512, 512), 0, 0, interpolation
        = cv2.INTER_NEAREST)
    landmasses = cv2.bitwise_not(landmasses.astype(np.uint8))

    #######

    _, ship_mask = cv2.threshold(ndwi_prediction,32,255,cv2.
        THRESH_BINARY)
    ship_mask = cv2.bitwise_not(cv2.bitwise_or(ship_mask, landmasses))
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
    ship_mask = cv2.morphologyEx(ship_mask, cv2.MORPH_DILATE, kernel)

    #######
    output = np.zeros((h, w), np.float32)
```

```
255
256   # loop over the image, pixel by pixel
257   for y in range(0, h):
258     for x in range(0, w):
259       b, g, r = image[y, x] / 255
260       minV = min(image[y, x])
261       maxV = max(image[y, x])
262       l = (b + g + r)/3
263
264       if ((g == 0 and r == 0) or (b == 0 and r == 0) or ship_mask[y, x
              ] == 0):
265         output[y, x] = 0
266       else:
267         output[y, x] = min(255, 255 * (1 - (float(b) - float(r)**2) /
              (float(b) + float(r)**2))**2) * (1 - (1 - l**2) * abs(b -
              r))**2
268
269   # return the processed image
270   return Image.fromarray(output)
271
272 def asStride(arr,sub_shape,stride):
273     '''Get a strided sub-matrices view of an ndarray.
274     See also skimage.util.shape.view_as_windows()
275     '''
276     s0,s1=arr.strides[:2]
277     m1,n1=arr.shape[:2]
278     m2,n2=sub_shape
279     view_shape=(1+(m1-m2)//stride[0],1+(n1-n2)//stride[1],m2,n2)+arr.
          shape[2:]
280     strides=(stride[0]*s0,stride[1]*s1,s0,s1)+arr.strides[2:]
281     subs=np.lib.stride_tricks.as_strided(arr,view_shape,strides=
          strides)
282     return subs
283
284 def poolingOverlap(mat,ksize,stride=None,method='max',pad=False):
285     '''Overlapping pooling on 2D or 3D data.
286
287     <mat>: ndarray, input array to pool.
288     <ksize>: tuple of 2, kernel size in (ky, kx).
```

```
289     <stride>: tuple of 2 or None, stride of pooling window.
290               If None, same as <ksize> (non-overlapping pooling).
291     <method>: str, 'max for max-pooling,
292                    'mean' for mean-pooling.
293     <pad>: bool, pad <mat> or not. If no pad, output has size
294            (n-f)//s+1, n being <mat> size, f being kernel size, s
                stride.
295            if pad, output has size ceil(n/s).
296
297     Return <result>: pooled matrix.
298     '''
299
300     m, n = mat.shape[:2]
301     ky,kx=ksize
302     if stride is None:
303         stride=(ky,kx)
304     sy,sx=stride
305
306     _ceil=lambda x,y: int(np.ceil(x/float(y)))
307
308     if pad:
309         ny=_ceil(m,sy)
310         nx=_ceil(n,sx)
311         size=((ny-1)*sy+ky, (nx-1)*sx+kx) + mat.shape[2:]
312         mat_pad=np.full(size,np.nan)
313         mat_pad[:m,:n,...]=mat
314     else:
315         mat_pad=mat[:(m-ky)//sy*sy+ky, :(n-kx)//sx*sx+kx, ...]
316
317     view=asStride(mat_pad,ksize,stride)
318
319     if method=='max':
320         result=np.nanmax(view,axis=(2,3))
321     else:
322         result=np.nanmean(view,axis=(2,3))
323
324     return result
325
326
```

```
"""
##############################
Custom Dataset
##############################
"""


def find_classes(directory: str) -> Tuple[List[str], Dict[str, int]]:
    """Finds the class folder names in a target directory.

    Assumes target directory is in standard image classification format.

    Args:
        directory (str): target directory to load classnames from.

    Returns:
        Tuple[List[str], Dict[str, int]]: (list_of_class_names, dict(class_name: idx...))

    Example:
        find_classes("food_images/train")
        >>> (["class_1", "class_2"], {"class_1": 0, ...})
    """
    # 1. Get the class names by scanning the target directory
    classes = sorted(entry.name for entry in os.scandir(directory) if entry.is_dir())

    # 2. Raise an error if class names not found
    if not classes:
        raise FileNotFoundError(f"Couldn't find any classes in {directory}.")

    # 3. Create a dictionary of index labels (computers prefer numerical rather than string labels)
    class_to_idx = {cls_name: i for i, cls_name in enumerate(classes)}
    return classes, class_to_idx

def extract_class_from_path(path: Union[str, pathlib.Path],
```

```
    known_classes: Set[str]) -> str:
361     """
362     Automatically extracts the class name from a path by matching
            known class folder names.
363 
364     Parameters:
365         path (str or Path): Full path to the image.
366         known_classes (set): A set of known class names (e.g., {'ship
                ', 'noship'}).
367 
368     Returns:
369         str: The class name found in the path.
370 
371     Raises:
372         ValueError: If no known class name is found in the path.
373     """
374     path = pathlib.Path(path)
375     for part in path.parts:
376         if part in known_classes:
377             return part
378     raise ValueError(f"Class name not found in path: {path}")
379 
380 class ShipDatasetWithPreprocessing(Dataset):
381 
382     def __init__(self, targ_dir: str, transform=None, preprocessing=
            None) -> None:
383         self.paths = list(pathlib.Path(targ_dir).rglob("*.png"))
384 
385         self.transform = transform
386         self.preprocessing = preprocessing
387         self.classes, self.class_to_idx = find_classes(targ_dir)
388 
389     def load_image(self, index: int) -> Image.Image:
390         image_path = self.paths[index]
391         return Image.open(image_path).convert('RGB')
392     def __len__(self) -> int:
393         return len(self.paths)
394 
395     def __getitem__(self, index: int) -> Tuple[torch.Tensor, int]:
```

```
396         img_path = self.paths[index]
397         img = self.load_image(index)
398 
399         # class name is the first folder after targ_dir
400         class_name = extract_class_from_path(img_path, self.classes)
401         class_idx = self.class_to_idx[class_name]
402 
403         # Preprocess if necessary
404         if self.preprocessing:
405           img = self.preprocessing(img)
406 
407         # Transform if necessary
408         if self.transform:
409             return self.transform(img), class_idx # return data, label
                    (X, y)
410         else:
411             return img, class_idx # return data, label (X, y)
412 
413 def stratified_limit_dataset(dataset, max_size):
414 
415     # Group indices by class
416     class_indices = defaultdict(list)
417     for idx, path in enumerate(dataset.paths):
418         class_name = extract_class_from_path(path, dataset.classes)
419         class_idx = dataset.class_to_idx[class_name]
420         class_indices[class_idx].append(idx)
421 
422     # Find the size of the smallest class
423     min_class_size = min(len(idxs) for idxs in class_indices.values())
424 
425     # Calculate total number of classes
426     num_classes = len(class_indices)
427 
428     # Determine how many samples per class to take:
429     # Oversample smaller classes by ensuring at least min_class_size
        samples each,
430     # or scale to max_size accordingly if needed
431     base_samples_per_class = min_class_size
432     total_samples_needed = base_samples_per_class * num_classes
```

```
433
434     # If max_size is smaller than total_samples_needed, scale down
            proportionally
435     if max_size < total_samples_needed:
436         base_samples_per_class = max_size // num_classes
437
438     selected_indices = []
439     for class_idx, indices in class_indices.items():
440         # Oversample smaller classes by repeating indices if needed
441         n_samples = base_samples_per_class
442
443         # If class is smaller, oversample by repeating indices
444         if len(indices) < n_samples:
445             reps = n_samples // len(indices)
446             rem = n_samples % len(indices)
447             oversampled = indices * reps + indices[:rem]
448         else:
449             random.shuffle(indices)
450             oversampled = indices[:n_samples]
451
452         selected_indices.extend(oversampled)
453
454     random.shuffle(selected_indices)
455
456     # Print class distribution in limited dataset
457     selected_class_counts = Counter()
458     for idx in selected_indices:
459         path = dataset.paths[idx]
460         class_name = extract_class_from_path(path, dataset.classes)
461         selected_class_counts[class_name] += 1
462
463     print("Samples per class in limited dataset:")
464     for cls_name, count in selected_class_counts.items():
465         print(f"  {cls_name}: {count}")
466
467     return Subset(dataset, selected_indices)
468
469 def unwrap_subset(dataset):
470     """
```

```
471     Recursively unwraps a torch.utils.data.Subset to get the original
            dataset
472     and the corresponding flattened indices.
473     """
474     indices = []
475 
476     while isinstance(dataset, Subset):
477         if not indices:
478             indices = dataset.indices
479         else:
480             indices = [dataset.indices[i] for i in indices]
481         dataset = dataset.dataset
482 
483     return dataset, indices
484 
485 def compute_class_and_sample_weights(dataset):
486     """
487     Compute class weights (for CrossEntropyLoss) and sample weights (
            for WeightedRandomSampler),
488     supporting both Dataset and Subset objects.
489     """
490     base_dataset, indices = unwrap_subset(dataset)
491 
492     if not indices:
493         indices = list(range(len(base_dataset)))
494 
495     # Attempt fast path: use dataset.targets if available on
            base_dataset
496     try:
497         targets = base_dataset.targets
498         # Select targets only for subset indices
499         targets = [targets[i] for i in indices]
500     except AttributeError:
501         # Fallback: extract labels from paths (or any other way) only
                for subset indices
502         targets = []
503         for i in indices:
504             path = base_dataset.paths[i]
505             class_name = extract_class_from_path(path, base_dataset.
```

```
                classes)
506             class_idx = base_dataset.class_to_idx[class_name]
507             targets.append(class_idx)
508
509     class_counts = Counter(targets)
510     total_samples = sum(class_counts.values())
511     num_classes = len(class_counts)
512
513     class_weights_dict = {
514         cls: total_samples / (num_classes * count)
515         for cls, count in class_counts.items()
516     }
517
518     # Map to full index space for loss function (some may be zero if
        unused)
519     class_weights = [
520         class_weights_dict.get(cls, 0.0)
521         for cls in range(len(base_dataset.class_to_idx))
522     ]
523     class_weights = torch.tensor(class_weights, dtype=torch.float)
524
525     # Sample weights using the subset class_weights
526     sample_weights = [class_weights_dict[cls] for cls in targets]
527
528     return class_weights, sample_weights
529
530
531 """
532 ##############################
533 Model training, testing, loading and saving
534 ##############################
535 """
536
537 def train(model, device, train_loader, optimizer, criterion, epoch):
538     model.train()
539     running_loss = 0.0
540     for batch_idx, (data, target) in enumerate(train_loader):
541         data, target = data.to(device), target.to(device)
542
```

```
543         optimizer.zero_grad()
544         output = model(data)
545         loss = criterion(output, target)
546         loss.backward()
547         optimizer.step()
548 
549         running_loss += loss.item()
550         if batch_idx % 10 == 0:
551             print(f"Train Epoch: {epoch} [{(batch_idx) * len(data)} /
                {len(train_loader.dataset)}] ({100 * (batch_idx) / len
                (train_loader):.2f}%)\t Loss: {loss.item():.6f}")
552 
553     avg_loss = running_loss / len(train_loader)
554     print(f"Train Epoch: {epoch}, Average Loss: {avg_loss:.4f}")
555     return avg_loss
556 
557 def test(model, device, test_loader, criterion):
558     model.eval()
559     test_loss = 0.0
560     correct = 0
561 
562     with torch.no_grad():
563         for data, target in test_loader:
564             data, target = data.to(device), target.to(device)
565             output = model(data)
566             test_loss += criterion(output, target).item()
567             pred = output.argmax(dim=1)
568             correct += pred.eq(target).sum().item()
569 
570     test_loss /= len(test_loader)
571     accuracy = 100. * correct / len(test_loader.dataset)
572     print(f"\nTest set: Average loss: {test_loss: 0.4f}, Accuracy {
        correct}/{len(test_loader.dataset)}  ({accuracy:.2f}%)\n")
573     return accuracy
574 
575 def save_checkpoint(model, optimizer, epoch, loss, path=
    CHECKPOINT_PATH):
576     torch.save({
577         'epoch': epoch,
```

```
578         'model_state_dict': model.state_dict(),
579         'optimizer_state_dict': optimizer.state_dict(),
580         'loss': loss,
581     }, path)
582     print(f"Checkpoint saved at epoch {epoch}.")
583 
584 def load_checkpoint(device, model, optimizer, path=CHECKPOINT_PATH):
585     print("Loading from checkpoint...")
586     if os.path.isfile(path):
587         checkpoint = torch.load(path, map_location=device)
588 
589         model.load_state_dict(checkpoint['model_state_dict'])
590         model.to(device)  # Move model to device after loading
591 
592         optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
593 
594         # Move optimizer tensors to the correct device
595         for state in optimizer.state.values():
596             for k, v in state.items():
597                 if isinstance(v, torch.Tensor):
598                     state[k] = v.to(device)
599 
600         start_epoch = checkpoint['epoch'] + 1
601         loss = checkpoint['loss']
602         print(f"Checkpoint loaded     resuming from epoch {start_epoch
            }")
603         return start_epoch, loss
604     else:
605         print("No checkpoint found, starting from scratch.")
606         return 1, None
607 
608 
609 """
610 ###############################
611 Models
612 ###############################
613 """
614 
615 
```

```
616 class CNN512(nn.Module):
617     def __init__(self, num_classes=2):
618         super(CNN512, self).__init__()
619         self.quant_inp = qnn.QuantIdentity(bit_width=8,
                return_quant_tensor=True)
620 
621         self.conv1 = qnn.QuantConv2d(1, 8, 3, stride=2, padding=1,
                weight_bit_width=8, weight_quant=
                Int8WeightPerTensorFixedPoint)
622         self.relu1 = qnn.QuantReLU(bit_width=8, return_quant_tensor=
                True)
623 
624         self.conv2 = qnn.QuantConv2d(8, 16, 3, stride=2, padding=1,
                weight_bit_width=8, weight_quant=
                Int8WeightPerTensorFixedPoint)
625         self.relu2 = qnn.QuantReLU(bit_width=8, return_quant_tensor=
                True)
626 
627         self.conv3 = qnn.QuantConv2d(16, 32, 3, stride=2, padding=1,
                weight_bit_width=8, weight_quant=
                Int8WeightPerTensorFixedPoint)
628         self.relu3 = qnn.QuantReLU(bit_width=8, return_quant_tensor=
                True)
629 
630         self.conv4 = qnn.QuantConv2d(32, 64, 3, stride=2, padding=1,
                weight_bit_width=8, weight_quant=
                Int8WeightPerTensorFixedPoint)
631         self.relu4 = qnn.QuantReLU(bit_width=8, return_quant_tensor=
                True)
632 
633         self.pool = nn.AdaptiveAvgPool2d((1, 1))
634         self.fc = qnn.QuantLinear(64, num_classes, weight_bit_width=8,
                 weight_quant=Int8WeightPerTensorFixedPoint)
635 
636     def forward(self, x):
637         x = self.quant_inp(x)
638         x = self.relu1(self.conv1(x))
639         x = self.relu2(self.conv2(x))
640         x = self.relu3(self.conv3(x))
```

```
641         x = self.relu4(self.conv4(x))
642         x = self.pool(x)
643         x = x.view(x.size(0), -1)
644         return self.fc(x)
645 
646 class CNN128_Shallow(nn.Module):
647     def __init__(self, num_classes=2):
648         super(CNN128_Shallow, self).__init__()
649 
650         self.quant_inp = qnn.QuantIdentity(bit_width=8,
                return_quant_tensor=True)
651 
652         self.conv_block1 = nn.Sequential(
653             qnn.QuantConv2d(1, 16, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
654             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
655             qnn.QuantConv2d(16, 32, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
656             qnn.QuantReLU(bit_width=8, return_quant_tensor=True)
657         )
658 
659         self.pool = nn.AdaptiveAvgPool2d((1, 1))
660 
661         self.fc = qnn.QuantLinear(32, num_classes, weight_bit_width=8,
                weight_quant=Int8WeightPerTensorFixedPoint)
662 
663     def forward(self, x):
664         x = self.quant_inp(x)
665         x = self.conv_block1(x)
666         x = self.pool(x)
667         x = x.view(x.size(0), -1)
668         return self.fc(x)
669 
670 class CNN128_Compact_k3(nn.Module):
671     def __init__(self, num_classes=2):
672         super(CNN128_Compact_k3, self).__init__()
673 
```

```
674         self.quant_inp = qnn.QuantIdentity(bit_width=8,
                return_quant_tensor=True)
675
676         self.conv_block1 = nn.Sequential(
677             qnn.QuantConv2d(1, 16, 3, stride=1, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
678             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
679             qnn.QuantConv2d(16, 16, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
680             qnn.QuantReLU(bit_width=8, return_quant_tensor=True)
681         )
682
683         self.conv_block2 = nn.Sequential(
684             qnn.QuantConv2d(16, 32, 3, stride=1, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
685             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
686             qnn.QuantConv2d(32, 32, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
687             qnn.QuantReLU(bit_width=8, return_quant_tensor=True)
688         )
689
690         self.conv_block3 = nn.Sequential(
691             qnn.QuantConv2d(32, 64, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
692             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
693         )
694
695         self.pool = nn.AdaptiveAvgPool2d((1, 1))
696
697         self.fc = qnn.QuantLinear(64, num_classes, weight_bit_width=8,
                weight_quant=Int8WeightPerTensorFixedPoint)
698
699     def forward(self, x):
700         x = self.quant_inp(x)
```

```
701         x = self.conv_block1(x)
702         x = self.conv_block2(x)
703         x = self.conv_block3(x)
704         x = self.pool(x)
705         x = x.view(x.size(0), -1)
706         logits = self.fc(x)
707         # Apply softmax only in eval/inference mode
708         if not self.training:
709             return torch.softmax(logits, dim=1)
710         return logits
711
712 class CNN128_Compact_k3_Pooling(nn.Module):
713     def __init__(self, num_classes=2):
714         super(CNN128_Compact_k3_Pooling, self).__init__()
715
716         self.quant_inp = qnn.QuantIdentity(bit_width=8,
                return_quant_tensor=True)
717
718         self.conv_block1 = nn.Sequential(
719             qnn.QuantConv2d(1, 16, 3, stride=1, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
720             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
721             qnn.QuantConv2d(16, 16, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
722             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
723             nn.MaxPool2d(kernel_size=2, stride=2)
724         )
725
726         self.conv_block2 = nn.Sequential(
727             qnn.QuantConv2d(16, 32, 3, stride=1, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
728             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
729             qnn.QuantConv2d(32, 32, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
730             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
```

```
731             nn.MaxPool2d(kernel_size=2, stride=2)
732         )
733 
734         self.conv_block3 = nn.Sequential(
735             qnn.QuantConv2d(32, 64, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
736             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
737             nn.MaxPool2d(kernel_size=2, stride=2)
738         )
739 
740         self.pool = nn.AdaptiveAvgPool2d((1, 1))
741 
742         self.fc = qnn.QuantLinear(64, num_classes, weight_bit_width=8,
                weight_quant=Int8WeightPerTensorFixedPoint)
743 
744     def forward(self, x):
745         x = self.quant_inp(x)
746         x = self.conv_block1(x)
747         x = self.conv_block2(x)
748         x = self.conv_block3(x)
749         x = self.pool(x)
750         x = x.view(x.size(0), -1)
751         logits = self.fc(x)
752         # Apply softmax only in eval/inference mode
753         if not self.training:
754             return torch.softmax(logits, dim=1)
755         return logits
756 
757 class CNN128_UltraCompact_k3_Pooling(nn.Module):
758     def __init__(self, num_classes=2):
759         super(CNN128_UltraCompact_k3_Pooling, self).__init__()
760 
761         self.quant_inp = qnn.QuantIdentity(bit_width=8,
               return_quant_tensor=True)
762 
763         self.conv_block1 = nn.Sequential(
764             qnn.QuantConv2d(1, 8, 3, stride=1, padding=1,
                    weight_bit_width=8, weight_quant=
```

```
                Int8WeightPerTensorFixedPoint),
765             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
766             qnn.QuantConv2d(8, 8, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
767             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
768             nn.MaxPool2d(kernel_size=2, stride=2)
769         )
770
771         self.conv_block2 = nn.Sequential(
772             qnn.QuantConv2d(8, 16, 3, stride=1, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
773             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
774             qnn.QuantConv2d(16, 16, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
775             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
776             nn.MaxPool2d(kernel_size=2, stride=2)
777         )
778
779         self.conv_block3 = nn.Sequential(
780             qnn.QuantConv2d(16, 32, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
781             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
782             nn.MaxPool2d(kernel_size=2, stride=2)
783         )
784
785         self.pool = nn.AdaptiveAvgPool2d((1, 1))
786
787         self.fc = qnn.QuantLinear(32, num_classes, weight_bit_width=8,
                weight_quant=Int8WeightPerTensorFixedPoint)
788
789     def forward(self, x):
790         x = self.quant_inp(x)
791         x = self.conv_block1(x)
792         x = self.conv_block2(x)
793         x = self.conv_block3(x)
```

```
794         x = self.pool(x)
795         x = x.view(x.size(0), -1)
796         logits = self.fc(x)
797         # Apply softmax only in eval/inference mode
798         if not self.training:
799             return torch.softmax(logits, dim=1)
800         return logits
801
802 class CNN128_DeepQuant_k3(nn.Module):
803     def __init__(self, num_classes=2):
804         super(CNN128_DeepQuant_k3, self).__init__()
805
806         self.quant_inp = qnn.QuantIdentity(bit_width=8,
               return_quant_tensor=True)
807
808         self.conv_block1 = nn.Sequential(
809             qnn.QuantConv2d(1, 32, 3, stride=1, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
810             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
811             qnn.QuantConv2d(32, 32, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
812             qnn.QuantReLU(bit_width=8, return_quant_tensor=True)
813         )
814
815         self.conv_block2 = nn.Sequential(
816             qnn.QuantConv2d(32, 64, 3, stride=1, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
817             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
818             qnn.QuantConv2d(64, 64, 3, stride=2, padding=1,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),
819             qnn.QuantReLU(bit_width=8, return_quant_tensor=True)
820         )
821
822         self.conv_block3 = nn.Sequential(
823             qnn.QuantConv2d(64, 128, 3, stride=2, padding=1,
```

```
                weight_bit_width=8, weight_quant=
                Int8WeightPerTensorFixedPoint),
824             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
825         )
826
827         self.pool = nn.AdaptiveAvgPool2d((1, 1))
828
829         self.fc = qnn.QuantLinear(128, num_classes, weight_bit_width
                =8, weight_quant=Int8WeightPerTensorFixedPoint)
830
831     def forward(self, x):
832         x = self.quant_inp(x)
833         x = self.conv_block1(x)
834         x = self.conv_block2(x)
835         x = self.conv_block3(x)
836         x = self.pool(x)
837         x = x.view(x.size(0), -1)
838         return self.fc(x)
839
840 class CNN128_DeepQuant_k5(nn.Module):
841     def __init__(self, num_classes=2):
842         super(CNN128_DeepQuant_k5, self).__init__()
843
844         self.quant_inp = qnn.QuantIdentity(bit_width=8,
                return_quant_tensor=True)
845
846         self.conv_block1 = nn.Sequential(
847             qnn.QuantConv2d(1, 32, 5, stride=1, padding=2,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint),  # 5x5 kernel
848             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
849             qnn.QuantConv2d(32, 32, 5, stride=2, padding=2,
                    weight_bit_width=8, weight_quant=
                    Int8WeightPerTensorFixedPoint), # 5x5 kernel
850             qnn.QuantReLU(bit_width=8, return_quant_tensor=True)
851         )
852
853         self.conv_block2 = nn.Sequential(
854             qnn.QuantConv2d(32, 64, 5, stride=1, padding=2,
```

```
                weight_bit_width=8, weight_quant=
                Int8WeightPerTensorFixedPoint),  # 5x5 kernel
855             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
856             qnn.QuantConv2d(64, 64, 5, stride=2, padding=2,
                weight_bit_width=8, weight_quant=
                Int8WeightPerTensorFixedPoint),  # 5x5 kernel
857             qnn.QuantReLU(bit_width=8, return_quant_tensor=True)
858         )
859
860         self.conv_block3 = nn.Sequential(
861             qnn.QuantConv2d(64, 128, 3, stride=2, padding=1,
                weight_bit_width=8, weight_quant=
                Int8WeightPerTensorFixedPoint),  # keep 3x3 here
862             qnn.QuantReLU(bit_width=8, return_quant_tensor=True),
863         )
864
865         self.pool = nn.AdaptiveAvgPool2d((1, 1))
866         self.fc = qnn.QuantLinear(128, num_classes, weight_bit_width
            =8, weight_quant=Int8WeightPerTensorFixedPoint)
867
868     def forward(self, x):
869         x = self.quant_inp(x)
870         x = self.conv_block1(x)
871         x = self.conv_block2(x)
872         x = self.conv_block3(x)
873         x = self.pool(x)
874         x = x.view(x.size(0), -1)
875         return self.fc(x)
876
877 class CNN128_TEST(nn.Module):
878     def __init__(self, num_classes=2):
879         super(CNN128_TEST, self).__init__()
880         self.conv1 = nn.Conv2d(1, 4, kernel_size=3, padding=1)
881         self.pool = nn.MaxPool2d(2, 2)
882         self.fc1 = nn.Linear(4 * 64 * 64, 2)  # assuming 10 classes
883
884     def forward(self, x):
885         x = self.pool(F.relu(self.conv1(x)))  # -> [B, 4, 64, 64]
886         x = x.view(x.size(0), -1)             # flatten
```

```
887         x = self.fc1(x)
888         return x
889
890 """
891 ################################
892 MAIN
893 ################################
894 """
895
896 architectures = {
897     "CNN512": CNN512,
898     "CNN128_Shallow": CNN128_Shallow,
899     "CNN128_DeepQuant_k3": CNN128_DeepQuant_k3,
900     "CNN128_DeepQuant_k5": CNN128_DeepQuant_k5,
901     "CNN128_Compact_k3": CNN128_Compact_k3,
902     "CNN128_Compact_k3_Pooling": CNN128_Compact_k3_Pooling,
903     "CNN128_UltraCompact_k3_Pooling": CNN128_UltraCompact_k3_Pooling,
904     "CNN128_TEST": CNN128_TEST
905 }
906
907 DATASETS = {
908    "MASATI-v2": WRK_DIR + '/Datasets/MASATI-v2/Data/images/',
909    "Patch": WRK_DIR + '/Datasets/PatchDataset/'
910 }
911
912 preprocessing_methods = {
913     "grayscale": None,
914     "ndwi": ndwi,
915     "ndwi_coast_rmv": ndwi_coast_removal
916 }
917
918
919 def main():
920
921     """
922     ================================
923     Training Loop
924     ================================
925     """
```

```
926     DATASET_DIR = DATASETS["Patch"]
927     architecture = 'CNN128_UltraCompact_k3_Pooling'
928     preprocessing_method = "ndwi"
929
930     use_cuda = True
931     load_model = True
932     batch_size = 64
933     epochs = 15
934     dataset_config = {"max_size": 20000, "train_test_split": 0.8}
935
936     export_onnx = True
937     export_hls4ml = True
938
939     device = torch.device("cuda" if torch.cuda.is_available() and
            use_cuda else "cpu")
940
941     print('
            ------------------------------------------------------------')
942     print('Working Directory: ' + WRK_DIR)
943     print('Dataset Source: ' + DATASET_DIR)
944     print('Using device: ' + device.type)
945     print('
            ------------------------------------------------------------')
946     print('Preprocessing Method: ' + preprocessing_method)
947
948     dataset_transforms = transforms.Compose([
949         transforms.Grayscale(),
950         transforms.ToTensor()
951     ])
952
953     full_dataset = ShipDatasetWithPreprocessing(targ_dir=DATASET_DIR,
            transform = dataset_transforms, preprocessing=
            preprocessing_methods[preprocessing_method])
954
955     limited_dataset = stratified_limit_dataset(full_dataset, max_size=
            dataset_config["max_size"])
956
957     train_size = int(dataset_config["train_test_split"] * len(
            limited_dataset))
```

```
958     test_size = len(limited_dataset) - train_size
959
960     train_dataset, test_dataset = random_split(limited_dataset, [
            train_size, test_size])
961
962     class_weights, sample_weights = compute_class_and_sample_weights(
            train_dataset)
963
964     class_names = full_dataset.classes
965     num_classes = len(class_names)
966
967     print('Class Weights: ' + str(class_weights))
968     print('Train Length: ' + str(len(train_dataset)))
969     print('Test Length: ' + str(len(test_dataset)))
970     print('
            ------------------------------------------------------------')
971
972
973     sampler = WeightedRandomSampler(weights=sample_weights,
            num_samples=len(sample_weights), replacement=True)
974
975     train_loader = DataLoader(train_dataset, batch_size=batch_size,
            sampler=sampler)
976     test_loader = DataLoader(test_dataset, batch_size=batch_size)
977
978     model = architectures[architecture](num_classes=num_classes).to(
            device)
979     optimizer = optim.Adam(model.parameters(), lr=0.001)
980
981     criterion = nn.CrossEntropyLoss(weight=class_weights.to(device))
982
983
984     print('Architecture: ' + architecture)
985
986     if(load_model):
987         start_epoch, _ = load_checkpoint(device, model, optimizer,
                path=CHECKPOINT_PATH + architecture + '_' +
                preprocessing_method + '_checkpoint.pth')
988     else:
```

```
989         start_epoch = 1
990
991     print('
            ------------------------------------------------------------')
992     print('Starting Training...\n')
993
994     #show_random_samples(train_dataset, class_names=class_names, n=8,
            cols=4)
995
996
997     for epoch in range(start_epoch, epochs + 1):
998         train_loss = train(model, device, train_loader, optimizer,
                criterion, epoch)
999         accuracy = test(model, device, test_loader, criterion)
1000        save_checkpoint(model, optimizer, epoch, train_loss, path =
                CHECKPOINT_PATH + architecture + '_' +
                preprocessing_method + "_checkpoint.pth")
1001
1002    # Save final model
1003    print("Saving PyTorch Model...")
1004    torch.save(model.state_dict(), SAVE_PATH + '/pytorch/' +
            architecture + '_' + preprocessing_method + '_FINAL.pt')
1005    print("Done.")
1006
1007    """
1008    ==============================
1009    ONNX Export
1010    ==============================
1011    """
1012
1013    if export_onnx:
1014        input_sample, _ = next(iter(train_loader))
1015        input_sample = input_sample.to(device)
1016
1017        model_path = SAVE_PATH + '/onnx/' + architecture + '_' +
                preprocessing_method + "_FINAL.onnx"
1018        print("Saving Onnx Model...")
1019        exported_model = export_qonnx(model, args=input_sample,
                export_path=model_path, opset_version=13)
```

```
1020        print("Done.")
1021
1022        cnnCleanNet = ModelWrapper(model_path)
1023        cnnCleanNet = cleanup_model(cnnCleanNet)
1024        cnnCleanNet = cnnCleanNet.transform(InferShapes())
1025        cnnCleanNet = cnnCleanNet.cleanup()
1026        cnnCleanNet = cnnCleanNet.transform(FoldConstants())
1027        cnnCleanNet = cnnCleanNet.cleanup()
1028        cnnCleanNet = cnnCleanNet.transform(
               ConvertToChannelsLastAndClean())
1029        cnnCleanNet = cnnCleanNet.cleanup()
1030        cnnCleanNet = cnnCleanNet.transform(GemmToMatMul())
1031        cnnCleanNet = cleanup_model(cnnCleanNet)
1032
1033        # Save the cleaned model to visualize once again with netron
1034        clean_path = SAVE_PATH + '/onnx/' + architecture + '_' +
               preprocessing_method + "_FINAL_CLEANED.onnx"
1035        print("Saving Cleaned Onnx Model...")
1036        cnnCleanNet.save(clean_path)  # quantized cleaned net
1037        print("Done.")
1038
1039        #showInNetron(model_path)
1040
1041    """
1042    ==============================
1043    Hls4Ml Export
1044    ==============================
1045    """
1046    if export_hls4ml and export_onnx:
1047            def assign_default_names(onnx_model):
1048                """ Assign default names to unnamed ONNX nodes"""
1049                for i, node in enumerate(onnx_model.graph.node):
1050                    if node.name == "":
1051                        node.name = f"{node.op_type}_layer_{i}"  #
                               Example: Gemm_layer_2
1052                        print(f"Assigned default name: {node.name}")
1053
1054            print('Starting hls4ml parsing...')
1055
```

```
1056            hls_path = SAVE_PATH + '/hls4ml/' + architecture + '_' +
                    preprocessing_method + "_HLS"
1057
1058            # First avoid no-named nodes by assigning default names
1059            assign_default_names(cnnCleanNet)
1060
1061            # Default
1062            cnnCleanNet_cfg = hls4ml.utils.config.
                    config_from_onnx_model(
1063                cnnCleanNet, granularity='name', backend='Vitis', #
                        default_precision='fixed<8,2>'
1064            )
1065
1066            # modify the config as desired
1067            hls_model = hls4ml.converters.convert_from_onnx_model(
1068                cnnCleanNet,
1069                output_dir = hls_path,
1070                io_type = 'io_stream',
1071                backend = 'Vitis',
1072                hls_config = cnnCleanNet_cfg,
1073                board='pynq-z1'
1074            )
1075            hls_model.compile()
1076            hls_model.build(csim=False, export=True)
1077
1078 if __name__ == '__main__':
1079     main()
```

Bibliografía

[1] R. Neris, A. Rodríguez, R. Guerra, S. López y R. Sarmiento, "FPGA-Based Implementation of a CNN Architecture for the On-Board Processing of Very High-Resolution Remote Sensing Images," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 15, págs. 3740-3750, 2022. DOI: 10.1109/JSTARS.2022.3169330.

[2] Y. LeCun, L. Bottou, Y. Bengio y P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, n.º 11, págs. 2278-2324, 1998.

[3] A. Krizhevsky, I. Sutskever y G. E. Hinton, "ImageNet classification with deep convolutional neural networks," en *Advances in Neural Information Processing Systems*, 2012.

[4] X. X. Zhu, D. Tuia, L. Mou et al., "Deep learning in remote sensing: A comprehensive review and list of resources," *IEEE Geoscience and Remote Sensing Magazine*, vol. 5, n.º 4, págs. 8-36, 2017.

[5] A. Shelestov, M. Lavreniuk, N. Kussul y A. Novikov, "Exploring deep learning techniques for land use and land cover classification of satellite imagery," *Cybernetics and Systems Analysis*, vol. 53, n.º 6, págs. 759-768, 2017.

[6] W. Li, Q. Wu e Y. Yan, "Ship detection using deep learning and GIS information," *Sensors*, vol. 18, n.º 10, pág. 3346, 2018.

[7] G.-S. Xia, X. Bai, J. Ding et al., "DOTA: A large-scale dataset for object detection in aerial images," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[8] G. Cheng, J. Han y X. Lu, "Remote sensing image scene classification: Benchmark and state of the art," *Proceedings of the IEEE*, vol. 105, n.º 10, págs. 1865-1883, 2020.

[9] T. Rudner, M. Rußwurm, F. Fil et al., "Multi3Net: Segmenting flooded buildings via fusion of multiresolution, multisensor, and multitemporal satellite imagery," *arXiv preprint arXiv:1812.01703*, 2018.

[10] J. Zhang, X. Feng, X. Wang, R. Yang y M. Lu, "A deep learning-based approach for automated classification of land cover using optical remote sensing data," *International Journal of Remote Sensing*, vol. 40, n.º 17, págs. 6713-6734, 2019.

[11] M. Rußwurm y M. Körner, "Self-attention for raw optical satellite time series classification," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 169, págs. 421-435, 2020.

[12] A. Paszke, S. Gross, F. Massa et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[13] X. R. Labs, *Brevitas: Quantization-aware training in PyTorch*, 2022. dirección: `https://github.com/Xilinx/brevitas`.

[14] *Open Neural Network Exchange (ONNX)*, 2022. dirección: `https://onnx.ai`.

[15] M. Blott, T. Preußer, N. Fraser et al., "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, n.º 3, págs. 1-23, 2018.

[16] J. Duarte, S.-C. Han et al., "Fast inference of deep neural networks in FPGAs for particle physics," en *Journal of Instrumentation*, vol. 13, 2018.

[17] V. Loncar et al., "hls4ml: An open-source framework for high-level synthesis of machine learning models," *Frontiers in Big Data*, vol. 4, pág. 709 636, 2021.

[18] *Vitis AI Development Environment*, 2023. dirección: `https://github.com/Xilinx/Vitis-AI`.

[19] I. Corporation, *OpenVINO Toolkit*, 2023. dirección: `https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html`.

[20] C. Benedek, V. Gál y Z. Kato, "MASATI: A Satellite Imagery Dataset for Maritime Applications," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 166, págs. 185-195, 2020. DOI: 10.1016/j.isprsjprs.2020.06.013.

[21] H. Xu, "Modification of normalised difference water index (NDWI) to enhance open water features in remotely sensed imagery," *International Journal of Remote Sensing*, vol. 27, n.º 14, págs. 3025-3033, 2006. DOI: 10.1080/01431160600589179.