Universidad de Oviedo

Grado en Física

Implementación de algoritmos de aprendizaje automático para la optimización de procesos industriales

Autor: Olai Dizy Aranda

Tutores: Santiago Folgueras Gómez e Isidro González Caballero

15 de junio de 2026

Resumen

Este trabajo aborda la implementación de un modelo de aprendizaje automático en una FPGA (*Field Programmable Gate Array*). El modelo, un perceptrón multicapa (MLP, *Multilayer Perceptron*) entrenado en PyTorch para predecir el grosor del galvanizado en un proceso industrial de ArcelorMittal™, se utiliza como base de un algoritmo de optimización basado en el descenso del gradiente sobre el MLP, y en unos BDTs (*Boosted Decision Trees*) que calculan los parámetros óptimos de las cuchillas de aire que controlan la deposición de zinc en el proceso del galvanizado del acero.

El objetivo principal es implementar dicho algoritmo de forma funcional en una placa Kria KV260 de AMD Xilinx™, recorriendo el ciclo completo de desarrollo: exportación de pesos desde PyTorch usando la traducción a código HLS (*High-Level Synthesis*) mediante la librería *hls4ml* (paso común pero no siempre empleado), síntesis e implementación con Vitis y Vivado, y despliegue con control remoto mediante PYNQ. Este flujo es el mismo que se emplea en física experimental para el despliegue de redes neuronales en sistemas de adquisición de datos y *trigger* en experimentos como CMS o DUNE, lo que enmarca el trabajo en el ámbito de la instrumentación científica moderna.

La implementación en FPGA requiere adaptar el modelo a aritmética de punto fijo y configurar los factores de reutilización de cada capa para equilibrar latencia y uso de recursos. La retropropagación y el descenso de gradiente, no soportados directamente por *hls4ml*, se implementan manualmente en HLS. Una vez constatado que la FPGA superaba ampliamente en velocidad al modelo de PyTorch, se desarrolló a posteriori un modelo equivalente en C++ con los pesos originales en punto flotante. Este se ejecuta de forma nativa en la CPU de la Kria, como referencia de software optimizado, teniendo así tres modelos para comparar entre sí.

La comparación entre las tres implementaciones muestra que el kernel de FPGA alcanza tiempos de ejecución inferiores a $8\,ms$, frente a los $\sim$14 s del modelo de PyTorch. Sin embargo, la sobrecarga de inicialización de PYNQ añade $\sim$1 s por ejecución, y el modelo de C++ resulta ser más rápido, más preciso y portable que la implementación en FPGA en todas las métricas evaluadas. La principal fuente de degradación de la FPGA es la pérdida de precisión debida a la aritmética de punto fijo y la omisión de los modelos BDT que actualizan las variables de estado, los cuales no pudieron incluirse por limitaciones de recursos.

Se concluye que, para este caso de uso concreto, la aceleración en FPGA no aporta una ventaja suficiente frente a una solución software optimizada, dado que los requisitos temporales de la aplicación industrial (recolocación de cuchillas en $\sim$1 min) son compatibles con cualquiera de las tres implementaciones. No obstante el proceso me ha permitido familiarizarme con el proceso de optimización de un algoritmo de ML para su ejecución en FPGAs lo cual resulta vital en numerosas aplicaciones.

Índice general

Capítulo 1

Introducción

La física experimental moderna requiere adquirir, filtrar y procesar volúmenes ingentes de datos en tiempos extremadamente cortos. Este reto es común a detectores de partículas, telescopios de ondas gravitacionales o sistemas de calorimetría de alta granularidad, y constituye el campo de la *instrumentación científica*.

Un desafío central en este ámbito es la inteligencia en el detector (*intelligence on detector*), un paradigma de toma de decisiones en tiempo real necesario para gestionar los masivos flujos de datos en física de partículas. Esto se evidencia en el experimento CMS del LHC, donde el sistema de disparo de nivel 1 (L1T) debe procesar colisiones a 40 MHz y decidir si descarta o almacena un evento en menos de 4 μs [1]; una restricción extrema que hace inviable el software convencional y posiciona a las FPGAs como la tecnología de elección [2]. Este escenario se repite en experimentos de nueva generación como DUNE (*Deep Underground Neutrino Experiment*), el cual generaría 145 EB de datos brutos al año sin un filtrado previo [3], impulsando la investigación de redes neuronales en FPGAs para viabilizar su adquisición de datos. En ambos casos, el objetivo crítico es aproximar la inteligencia computacional al detector para minimizar la latencia y el consumo energético, maximizando el paralelismo y la fiabilidad.

La creciente complejidad de los algoritmos de aprendizaje automático (*machine learning*, ML) empleados en física ha impulsado el desarrollo de herramientas específicas para traducir modelos entrenados en lenguajes/plataformas de alto nivel, como Python o PyTorch [4], a código sintetizable para FPGA. La librería *hls4ml* [5], por ejemplo, automatiza la conversión de redes neuronales a HLS (*High-Level Synthesis*), una extensión de C++ orientada a la descripción de hardware. La aparición de estas herramientas ha democratizado el acceso al despliegue de ML en hardware reconfigurable, trasladando parte del flujo de trabajo al ámbito de la física experimental.

Este Trabajo de Fin de Grado se enmarca precisamente en este contexto. Desarrollado como continuación de las prácticas de empresa realizadas en TheNextPangea SL™. El trabajo afronta un problema de optimización de un proceso industrial de galvanizado mediante un modelo de aprendizaje automático, y tiene como objetivo principal la implementación funcional de dicho modelo en una FPGA. El trabajo abarca el ciclo completo de desarrollo de FPGAs: desde el modelo original entrenado en PyTorch hasta su traducción a HLS con *hls4ml*, su síntesis e implementación en una placa Kria KV260 [6] de AMD Xilinx™, y la comparación cuantitativa de su rendimiento frente a implementaciones

alternativas en software.

Los conocimientos adquiridos durante el Grado en Física han sido determinantes en el desarrollo de este trabajo. La asignatura de *Introducción a la Física Computacional* (IFC) proporcionó la base de programación en Python y la familiaridad con entornos de cálculo científico necesaria para el análisis de resultados. La asignatura de *Análisis de Datos en Física Moderna* aportó el conocimiento directo de PyTorch y de las técnicas de aprendizaje automático empleadas en el trabajo, incluyendo la arquitectura de redes neuronales y los algoritmos de optimización basados en descenso de gradiente. La asignatura de *Métodos Numéricos* proporcionó el fundamento matemático para comprender el comportamiento numérico del descenso de gradiente ante distintas funciones de pérdida, esencial para interpretar las diferencias de convergencia entre las distintas implementaciones comparadas. La asignatura de *Electrónica* aportó el contexto para comprender la arquitectura de sistemas embebidos.

El objetivo principal de este trabajo es la implementación funcional de un modelo de aprendizaje automático en hardware reconfigurable de tipo FPGA, recorriendo el ciclo completo desde el modelo original hasta su despliegue y evaluación. Este objetivo se enmarca en la línea de trabajo de despliegue de ML en hardware de baja latencia que se desarrolla activamente en física experimental, y se descompone en los siguientes objetivos específicos:

- Traducir el modelo original de PyTorch a código HLS mediante la librería *hls4ml*, adaptando la representación numérica a punto fijo para su síntesis en FPGA.
- Sintetizar e implementar el diseño en una placa Kria KV260, verificando que el resultado cumple los requisitos de recursos y temporización.
- Comparar cuantitativamente el rendimiento en precisión y tiempo de ejecución de tres implementaciones: el modelo original en PyTorch, la implementación en FPGA y un modelo equivalente en C++.

El modelo original es un MLP (Multi Layer Perceptron) [7] que está entrenado para predecir el grosor del galvanizado de un proceso industrial de ArcelorMittal™, el modelo fue creado por la empresa TheNextPangea SL™en colaboración con aquella, con el objetivo de asistir a los operarios de máquina a la hora de decidir los parámetros de actuación de las cuchillas de aire.

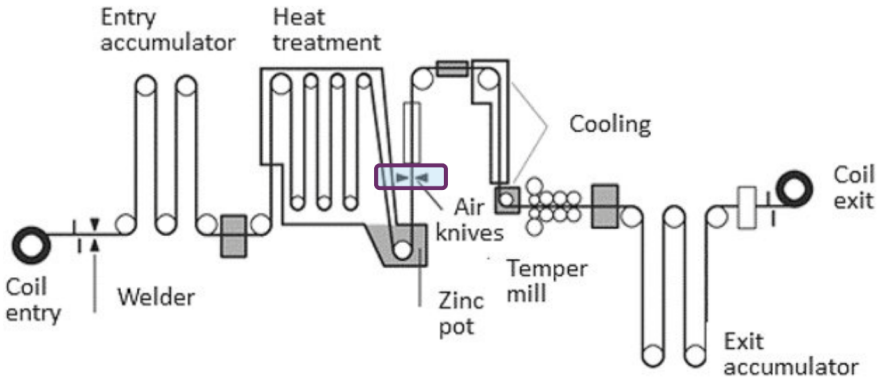

Figura 1.1: Esquema del proceso de galvanizado del acero en Arcelor™. Imagen proporcionada por TheNextPangea SL™.

En concreto, lo que hacen las cuchillas de aire es reducir la capa de deposición de zinc al grosor óptimo. Esto se hace soplando aire a presión que reduce la capa depositada después del baño de zinc que se le aplica a la lámina de acero. De ahí el interés por el modelo, ya que reducir los tiempos de toma de decisión puede, a la larga, traducirse en importantes ahorros de costes para la empresa, así como un beneficio medioambiental al reducirse el malgasto de material.

El presente trabajo se organiza de la siguiente manera. El Capítulo 2 presenta los fundamentos matemáticos de los algoritmos de aprendizaje automático empleados: una introducción básica a los conceptos generales, redes neuronales de tipo MLP, árboles de decisión potenciados, funciones de activación y el algoritmo de descenso de gradiente. El Capítulo 3 describe las herramientas utilizadas, con especial atención a las FPGAs y a HLS como capa de abstracción para su programación. El Capítulo 4 detalla el proceso completo de implementación del modelo en la FPGA, desde la exportación de pesos hasta el despliegue y la interfaz de control. Finalmente, el Capítulo 5 presenta los resultados experimentales y la comparativa cuantitativa entre las tres implementaciones desarrolladas. En el Capítulo 6 se discutirán las conclusiones del trabajo realizado desde dos puntos de vista. Primero el académico, enfocándose más en los aspectos formales y en lo que se aprendió del proyecto; segundo, el industrial, enfocándose más en las ventajas e inconvenientes de la implementación final. También se discuten mejoras y trabajo a futuro. Además, al final, se incluye un glosario con la terminología técnica que aparece en este trabajo, cada término que aparezca tiene un enlace en su primera mención relevante.

Capítulo 2

Fundamentos del Aprendizaje Automático

Este capítulo presenta los fundamentos matemáticos de las herramientas de aprendizaje automático utilizadas en el trabajo. Los conceptos aquí expuestos guardan una relación directa con los contenidos de la asignatura *Análisis de Datos en Física Moderna*, en la que se introducen técnicas de aprendizaje automático aplicadas al análisis de datos experimentales en física.

2.1. Introducción al Aprendizaje Automático

El aprendizaje automático (*Machine Learning*, ML) es una rama de la inteligencia artificial dedicada al desarrollo de modelos capaces de aprender patrones a partir de datos. En lugar de programar explícitamente todas las reglas que describen un sistema, estos modelos infieren automáticamente una relación entre variables observadas y magnitudes de interés a partir de ejemplos previamente conocidos.

En aprendizaje supervisado, utilizado en este trabajo, el objetivo es aproximar una función desconocida

$$y = f(\mathbf{x}), \tag{2.1}$$

donde $\mathbf{x}$ representa las variables de entrada e y la variable que se desea predecir. Una vez entrenado, el modelo proporciona una aproximación de esta función que puede utilizarse para realizar predicciones sobre nuevos datos.

Cada observación del conjunto de datos se denomina muestra y puede representarse como el par $(\mathbf{x}_i, y_i)$. Las componentes del vector $\mathbf{x}_i$ reciben el nombre de variables de entrada o *features*, mientras que y_i constituye la variable de salida o *target*. En el contexto de este trabajo, las variables de entrada corresponden a las magnitudes medidas durante el proceso de galvanizado, mientras que la salida es el grosor final del recubrimiento.

El conjunto de muestras disponible forma el conjunto de datos (*dataset*), que habitualmente se divide en un conjunto de entrenamiento, utilizado para ajustar los parámetros del modelo, y un conjunto de prueba, reservado para evaluar su capacidad de generalización. Durante el entrenamiento, las predicciones del modelo se comparan con los valores reales mediante una función de pérdida, cuyo valor se minimiza iterativamente para mejorar el rendimiento del sistema.

Los modelos empleados en este trabajo, los árboles de decisión potenciados y las redes neuronales artificiales, pertenecen a esta categoría de aprendizaje supervisado.

2.2. Redes neuronales: el Perceptrón Multicapa

Una red neuronal artificial es un modelo de aprendizaje automático inspirado en la estructura del sistema nervioso biológico. Está compuesta por unidades de cómputo elementales denominadas neuronas, organizadas en capas y conectadas mediante pesos ajustables. La familia más sencilla y general es el MLP (*Multilayer Perceptron*, perceptrón multicapa), en la que la información fluye en una única dirección desde la capa de entrada hasta la capa de salida, pasando por una o más capas ocultas. Estos conceptos se desarrollan en profundidad en la asignatura de *Análisis de Datos en Física Moderna*.

Para el problema de predicción del grosor del galvanizado se eligió un MLP por tres razones principales. En primer lugar, es una arquitectura sencilla y bien estudiada, cuyo comportamiento matemático es fácil de interpretar y depurar. En segundo lugar, el problema a modelizar no presenta una complejidad estructural que justifique arquitecturas más elaboradas como redes convolucionales o recurrentes. En tercer lugar, los MLP son modelos portables y compatibles con herramientas de traducción a hardware como *hls4ml*, lo que facilita su despliegue en FPGA.

2.2.1. Arquitectura y operación de cada capa

La operación fundamental de cada neurona consiste en calcular una combinación lineal de sus entradas y aplicarle una función no lineal denominada función de activación. Para una capa lineal con n entradas y p salidas, la transformación se escribe matricialmente como:

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}, \tag{2.2}$$

donde $\mathbf{x} \in \mathbb{R}^n$ es el vector de entrada, $W \in \mathbb{R}^{p\times n}$ es la matriz de pesos, $\mathbf{b} \in \mathbb{R}^p$ es el vector de sesgos (*bias*) y $\mathbf{z} \in \mathbb{R}^p$ es la salida antes de la activación. La aplicación sucesiva de estas transformaciones a través de las capas constituye la pasada hacia adelante (*forward pass*) de la red.

El modelo utilizado en este trabajo tiene la arquitectura representada en la Figura 2.1: una capa de entrada de dimensión 21 (una por cada variable del proceso), dos capas ocultas de dimensiones 128 y 64 respectivamente, y una capa de salida de dimensión 1 que predice el grosor del galvanizado. Entre cada capa lineal se intercalan una capa de normalización por lotes o *BatchNorm1d* [8] y una función de activación *SiLU*, cuyo papel se describe a continuación.

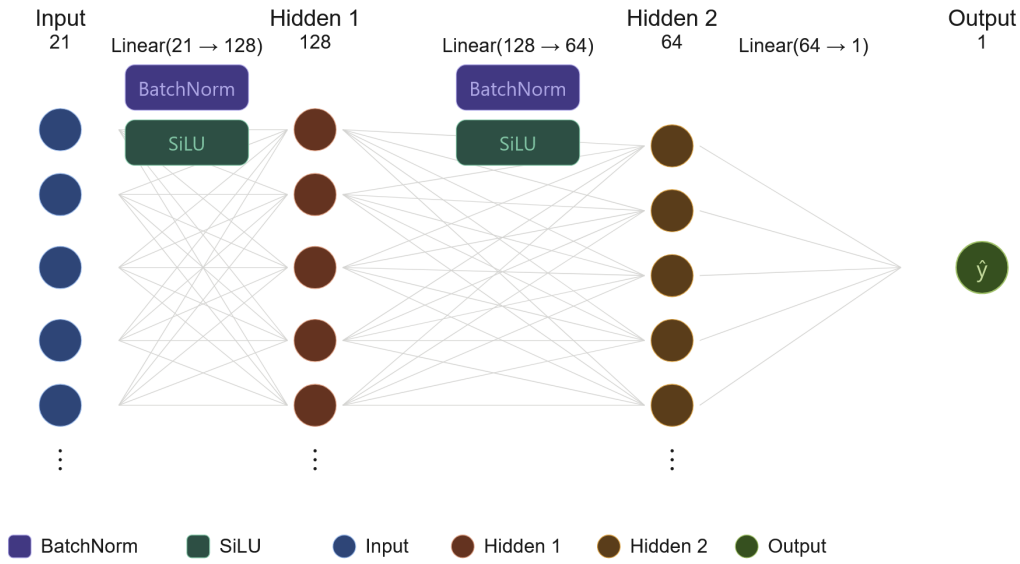

Figura 2.1: Arquitectura del modelo MLP original: capa de entrada de dimensión 21, dos capas ocultas de dimensiones 128 y 64 con normalización y activación SiLU, y capa de salida escalar.

2.2.2. Normalización por lotes: BatchNorm

Antes de aplicar la función de activación, cada capa oculta normaliza su salida mediante *BatchNorm1d*. Dado un mini-lote de m muestras, la normalización transforma cada característica x_i restando la media del lote $\mu_{\mathcal{B}}$ y dividiendo por su desviación típica $\sigma_{\mathcal{B}}$, para luego reescalar con parámetros aprendibles γ y β:

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}} \cdot \gamma + \beta, \qquad \mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i, \qquad \sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2, \tag{2.3}$$

donde $\varepsilon \ll 1$ es una constante de estabilidad numérica. El efecto práctico es doble: mantiene las activaciones en rangos acotados a lo largo del entrenamiento, evitando que los gradientes exploten o se anulen, y acelera la convergencia al reducir la dependencia entre capas.

2.2.3. Funciones de activación: SiLU frente a ReLU

Sin una función de activación no lineal entre capas, la composición de transformaciones lineales de la Ec. (2.2) seguiría siendo lineal, independientemente del número de capas. Las funciones de activación introducen la no linealidad necesaria para que la red pueda aproximar funciones arbitrariamente complejas.

La función más utilizada históricamente es ReLU (*Rectified Linear Unit*) [9]:

$$\mathrm{ReLU}(x) = \max(0, x). \tag{2.4}$$

ReLU es computacionalmente eficiente y mitiga el problema del gradiente desvaneciente para valores positivos. Sin embargo, presenta una discontinuidad en la derivada en $x = 0$:

$$\frac{d}{dx}\mathrm{ReLU}(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}, \tag{2.5}$$

lo que puede provocar inestabilidades numéricas en el cálculo del gradiente, especialmente relevante en este trabajo, donde el descenso de gradiente se aplica sobre las variables de entrada (no solo sobre los pesos) con precisión reducida de punto fijo.

En su lugar se utiliza SiLU (*Sigmoid Linear Unit*) [10], definida como el producto de la entrada por la función sigmoide $\sigma(x) = (1 + e^{-x})^{-1}$:

$$\mathrm{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}. \tag{2.6}$$

Su derivada es continua y suave en todo $\mathbb{R}$:

$$\frac{d}{dx}\mathrm{SiLU}(x) = \sigma(x) + x\,\sigma(x)\,(1 - \sigma(x)) = \sigma(x)\big(1 + x\,(1 - \sigma(x))\big), \tag{2.7}$$

lo que produce gradientes más estables durante la optimización. A diferencia de ReLU, SiLU no acota la salida por abajo en cero, permitiendo valores negativos pequeños que enriquecen la capacidad expresiva de la red. Estas propiedades la hacen preferible para el cálculo del gradiente con respecto a las entradas que se realiza en este trabajo.

2.3. Boosted Decision Trees

Un árbol de decisión es un modelo de aprendizaje automático que divide el espacio de variables de entrada mediante una serie de reglas binarias encadenadas. Partiendo de un nodo raíz, el árbol evalúa sucesivamente condiciones del tipo $x_i \leq \theta$ sobre las variables de entrada y ramifica hacia nodos hijos hasta alcanzar un nodo hoja, donde se emite una predicción.

Un árbol de decisión individual es, en general, un estimador de alta varianza. El método de potenciación (*boosting*) mitiga este problema combinando un conjunto de árboles débiles en un estimador fuerte: cada árbol se entrena sobre el conjunto de datos completo y sus resultados se agregan mediante votación por mayoría (en clasificación) o promediado (en regresión), tal y como se ilustra en la Figura 2.2 [11]. En el esquema de potenciación del gradiente (*gradient boosting*) [12], cada nuevo árbol se entrena específicamente para corregir los errores residuales del conjunto (*ensemble*) acumulado. El modelo final adopta la forma:

$$F_M(\mathbf{x}) = \sum_{m=1}^{M} \eta\, h_m(\mathbf{x}), \tag{2.8}$$

donde M es el número de árboles, $h_m(\mathbf{x})$ es la predicción del m-ésimo árbol y $\eta \in (0, 1]$ es una tasa de aprendizaje que controla la contribución de cada árbol. Al conjunto resultante se le denomina BDT (*Boosted Decision Tree*, árbol de decisión potenciado).

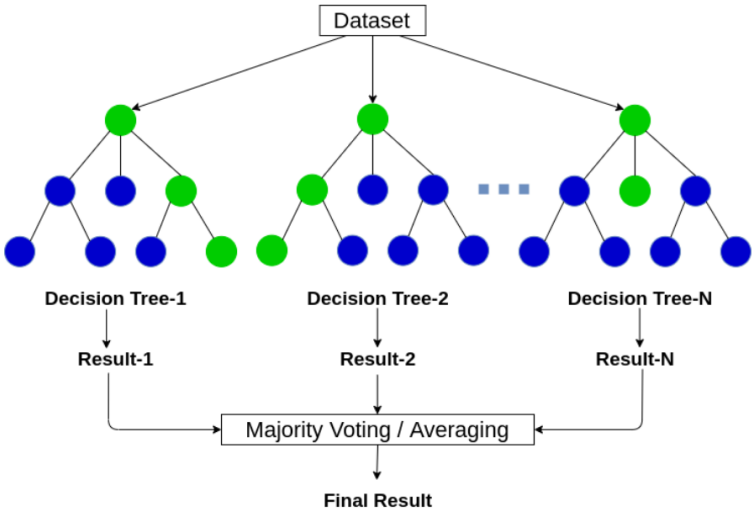

Figura 2.2: Esquema de un BDT: cada árbol produce un resultado independiente que se combina mediante promediado o votación para obtener la predicción final. Los puntos verdes son los nodos recorridos en cada árbol.

En el modelo empleado en este trabajo, los BDTs predicen la evolución de las variables de estado en respuesta a los cambios en las variables de control a lo largo de las iteraciones del descenso de gradiente. Por limitaciones de recursos de la FPGA, los BDTs no pudieron incluirse en la implementación en hardware; las implicaciones de esta simplificación se discuten en los Capítulos 4 y 5.

2.4. Función de pérdida y descenso de gradiente

Para cuantificar el error de un modelo de aprendizaje automático se define una función de pérdida (*loss function*) $\mathcal{L}$, que mide la discrepancia entre la salida predicha por la red y el valor objetivo. En este trabajo se utiliza el error cuadrático medio (MSE, *Mean Squared Error*), ampliamente estudiado en la asignatura de *Métodos Numéricos* en el contexto de ajuste de curvas y mínimos cuadrados. Para una única muestra:

$$\mathcal{L}(\hat{y}, y_{\text{target}}) = (\hat{y} - y_{\text{target}})^2, \tag{2.9}$$

donde $\hat{y}$ es el grosor predicho por la red e y_{target} el grosor objetivo. La elección de MSE como función de pérdida es natural para problemas de regresión escalar: su derivada es continua y proporcional al error residual, lo que produce gradientes bien condicionados.

El algoritmo de optimización consta de tres pasos que se repiten hasta alcanzar la tolerancia o el número máximo de iteraciones:

1. Calcular la salida de la red con los parámetros actuales (pasada hacia adelante), Ec. (2.2).
2. Calcular el gradiente de $\mathcal{L}$ respecto a las variables de control mediante retropropagación, Ec. (2.10).
3. Aplicar un paso de descenso de gradiente sobre las variables de control, Ec. (2.13).

2.4.1. Cálculo del gradiente mediante la regla de la cadena

El gradiente $\nabla_{x_{\text{ctrl}}}\mathcal{L}$ se obtiene aplicando la regla de la cadena a lo largo de todas las capas de la red. Denotando la salida de la capa l-ésima como $\mathbf{a}^{(l)}$, con $\mathbf{a}^{(0)} = \mathbf{x}$ y $\mathbf{a}^{(L)} = \hat{y}$, la derivada de la pérdida respecto a la entrada de la red se propaga hacia atrás como:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{a}^{(L-1)}} \cdots \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{x}}. \tag{2.10}$$

Para la función de pérdida MSE, el término inicial es simplemente:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - y_{\text{target}}). \tag{2.11}$$

La contribución de cada capa lineal $\mathbf{z}^{(l)} = W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$ al gradiente es:

$$\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{a}^{(l-1)}} = W^{(l)}, \tag{2.12}$$

es decir, la matriz de pesos transpuesta en el paso hacia atrás. La contribución de la función de activación SiLU es su derivada puntual, Ec. (2.7), aplicada elemento a elemento. El proceso completo de propagación del gradiente hacia atrás se denomina retropropagación, y es implementado de forma automática por PyTorch mediante diferenciación automática. En este trabajo, sin embargo, la retropropagación se implementa explícitamente en HLS para poder ejecutarse en la FPGA sin depender de ningún *framework* externo.

2.4.2. Descenso de gradiente con saturación

Una vez calculado el gradiente, se aplica un paso de descenso de gradiente exclusivamente sobre las variables de control $\mathbf{x}_{\text{ctrl}}$, manteniendo fijas las variables de estado:

$$\mathbf{x}_{\text{ctrl}}^{k+1} = \text{clip}\big(\mathbf{x}_{\text{ctrl}}^{k} - \eta\, \nabla_{\mathbf{x}_{\text{ctrl}}}\mathcal{L},\ \mathbf{x}_{\text{mín}},\ \mathbf{x}_{\text{máx}}\big)\,, \tag{2.13}$$

donde η es la tasa de aprendizaje y la función $\text{clip}(\cdot, \mathbf{x}_{\text{mín}}, \mathbf{x}_{\text{máx}})$ satura cada componente a su rango físico admisible (véase tabla 2.1), garantizando que los parámetros optimizados sean siempre realizables por la maquinaria.

La calidad de la optimización no depende únicamente de que la red prediga con exactitud el grosor del galvanizado, sino también de la suavidad del gradiente con respecto a las entradas. Un gradiente irregular dificulta la exploración del espacio de parámetros y puede hacer que el algoritmo quede atrapado en mínimos locales de baja calidad o que diverja. Esta es la razón por la que se eligieron SiLU y BatchNorm en lugar de funciones de activación más simples como ReLU: el gradiente suave de SiLU, Ec. (2.7), y la estabilización de activaciones que proporciona BatchNorm, Ec. (2.3), contribuyen conjuntamente a un descenso de gradiente más estable, especialmente importante al trabajar con aritmética de punto fijo en la FPGA.

2.5. Variables del modelo: control y estado

En el modelo original se seleccionaron 21 variables de las que depende más el grosor del galvanizado. Estas están normalizadas con un escalador o *scaler* para mejorar el análisis

de la red, ya que muchas varían en cuanto a cuál es su rango de valores. Esto acelera el entrenamiento del modelo.

También hay que distinguir entre dos tipos de variables; de control y de estado. Las de control se pueden cambiar directamente por el operario, mientras que las de estado no. Igualmente, estas últimas pueden depender indirectamente de las de control; por lo que, en el modelo, el descenso del gradiente se aplica solo a las variables de control. Las variables de estado no se actualizan directamente, sino que se usan unos modelos BDT para predecir su evolución mientras se actualizan las variables de control con el descenso del gradiente.

Las 21 variables de entrada del modelo se dividen en dos categorías. Las **variables de estado** describen condiciones del proceso que el operario no puede modificar directamente durante la optimización: el ancho de la banda (*ANCHO ACTUAL*), el espesor de la chapa (*ESPESOR ACTUAL*), la velocidad de línea (*CUCHILLA / AV VELOCIDAD DE LINEA*) y la posición central de las cuchillas (*CUCHILLA / AV. POS. CENTRAL CUCHILLAS*). Las **variables de control** son los 17 parámetros restantes, accionables directamente por el operario, y sobre los cuales se aplica el descenso de gradiente. La tabla 2.1 resume todas las variables con sus rangos físicos admisibles.

Variable	**Tipo**	**Mín.**	**Máx.**
ANCHO ACTUAL (mm)	Estado	600.0	1276.0
ESPESOR ACTUAL (mm^2)	Estado	0.1	3.5
CUCHILLA / AV VELOCIDAD DE LINEA (m/min)	Estado	0.1	105.0
CUCHILLA / AV. POS. CENTRAL CUCHILLAS (mm)	Estado	0.0	244.25
CUCHILLA / BASCULAMIENTO VERTICAL (mm)	Control	−298.2	172.5
CUCHILLA / DELANTERA / SP POS. OBLICUO (mm)	Control	−63.0	70.0
CUCHILLAS DUMA / AV. PRESION DELANTERA (mbar)	Control	0.0	430.6
CUCHILLAS DUMA / AJUSTE RODILLO CORRECTOR L.M. (mm)	Control	−65.1	101.4
CUCHILLA / TRASERA NORTE / SP. POS ABSOLUTA (mm)	Control	−51.9	104.6
CUCHILLA / DELANTERA NORTE / SP. POS ABSOLUTA (mm)	Control	19.5	174.9
CUCHILLA / TRASERA SUR / SP. POS ABSOLUTA (mm)	Control	−40.0	104.6
CUCHILLA / TRASERA / SP POS. OBLICUO (mm)	Control	−40.0	55.0
CUCHILLA DELANTERA / REGULACIÓN SOPLADO / AV. PRESION (gr/cm^2)	Control	−7.8	600.0
CUCHILLA / TRASERA NORTE / NIVEL 2 / SP. POS ABSOLUTA (mm)	Control	−40.0	100.0
CUCHILLA / SOPLANTE 1 / SP. VELOCIDAD (R.P.M.)	Control	0.0	3000.0
CUCHILLAS DUMA / AJUSTE HORIZONTAL TRASERA L.M. (mm)	Control	−14.7	100.0
CUCHILLA / DELANTERA NORTE / NIVEL 2 / SP. POS ABSOLUTA (mm)	Control	−15.0	115.0
CUCHILLAS DUMA / AJUSTE VERTICAL L.M. (mm)	Control	80.4	799.5
CUCHILLA / DELANTERA SUR / SP. POS ABSOLUTA (mm)	Control	11.9	167.2
CUCHILLA / NIVEL 2 / PRESET ALTURA (mm)	Control	100.0	500.0
CUCHILLA / VERTICAL NORTE / SP. POSICION (mm)	Control	90.0	550.0

Tabla 2.1: Variables de entrada del modelo con su clasificación y rangos físicos admisibles. Las variables de estado permanecen fijas durante la optimización en FPGA y C++; las de control son actualizadas en cada paso del descenso de gradiente.

2.6. Descripción detallada del modelo original

El modelo original usa las 21 variables de entrada en el MLP para hacer una predicción del grosor actual del galvanizado. Después se usa la propagación hacia atrás del gradiente sobre las variables de entrada. Estas se actualizan con un paso de descenso de gradiente saturado, en el que se limita el valor de las entradas acorde a los rangos permitidos. También se hace que las entradas correspondientes a las variables de estado sean 0, ya

que estas no pueden ser modificadas por el operario como las de control. Ya que las 4 variables de estado dependen indirectamente de las de control, se aplican los BDTs sobre estas para predecir su cambio. El proceso se repite hasta llegar a una tolerancia deseada. Aquí un esquema del proceso:

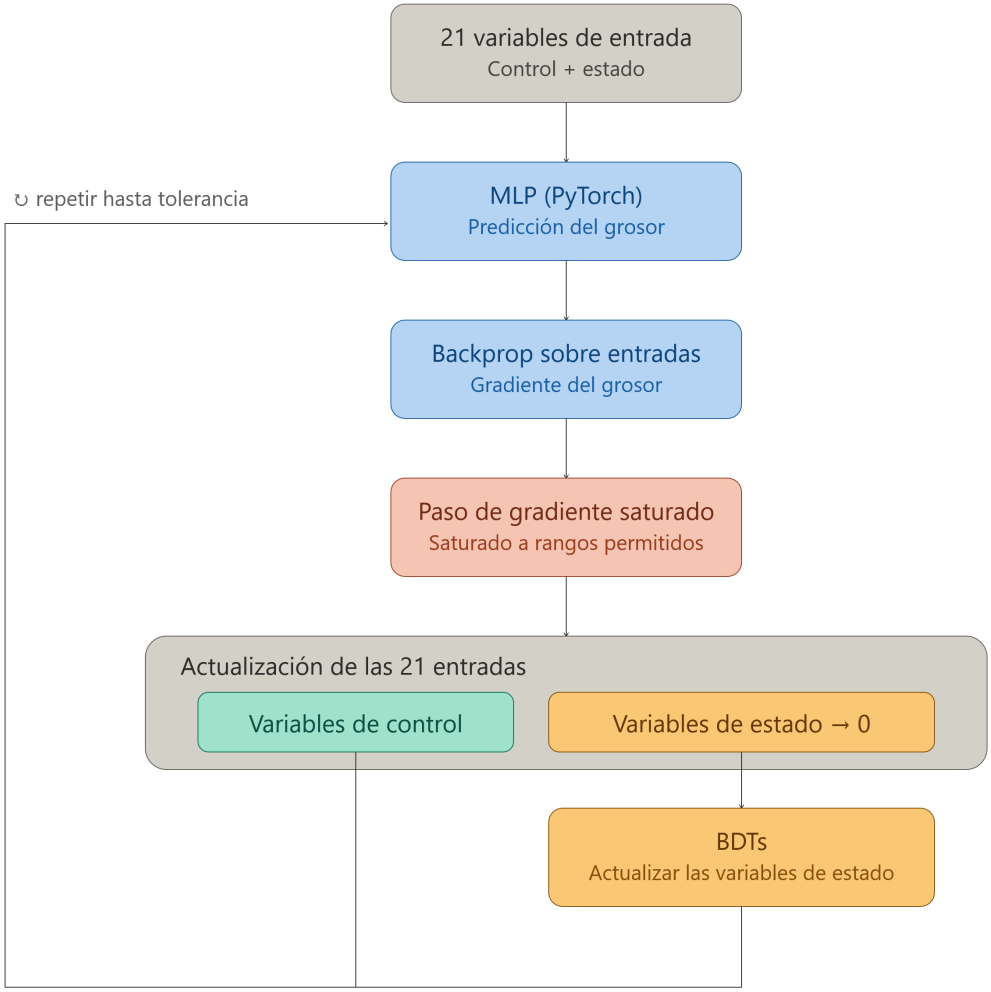

Figura 2.3: Esquema del modelo original.

Este modelo fue cedido por la empresa y no fue modificado. Esto hace que, al no estar optimizado para su uso en una FPGA, genere una implementación no óptima. Esto se discute más en detalle en los capítulos 4 y 5. En el capítulo 6 se sugieren mejoras para trabajos futuros.

Capítulo 3

FPGAs para Instrumentación Científica

Como se mencionó en la introducción al trabajo, la instrumentación científica moderna depende de sistemas capaces de procesar flujos de datos a velocidades que ningún procesador de propósito general puede alcanzar (obviamente, esto depende de la aplicación). Esto ha hecho que las FPGAs se hayan consolidado como una de las tecnologías de referencia para este tipo de procesamiento determinista de baja latencia. Este capítulo describe las herramientas empleadas en el presente trabajo para llevar a cabo la implementación de un modelo de aprendizaje automático en este tipo de hardware: la arquitectura de la FPGA utilizada, el flujo de diseño y las capas de abstracción que permiten traducir un modelo entrenado en PyTorch a firmware sintetizable.

3.1. FPGAs en la instrumentación científica

Los circuitos integrados pueden clasificarse según su grado de especialización: las CPUs son procesadores de propósito general flexibles pero poco eficientes para tareas específicas; las GPUs disponen de miles de núcleos paralelos optimizados para operaciones matriciales; los ASICs ofrecen el máximo rendimiento pero no pueden modificarse tras su fabricación [13]; y las FPGAs ocupan un espacio intermedio, combinando la reconfigurabilidad de una CPU con una latencia y eficiencia próximas a las de un ASIC.

Una FPGA (*Field Programmable Gate Array*) es, en esencia, una matriz de nodos de computación con conexiones programables, lo que permite replicar a nivel de hardware un algoritmo concreto. Sus bloques funcionales se agrupan en tres categorías, ilustradas en la Figura 3.1:

- **Módulos DSP** (*Digital Signal Processing*): realizan las operaciones matemáticas, principalmente multiplicaciones y acumulaciones.
- **Bloques lógicos**: implementan operaciones lógicas y combinacionales mediante LUTs (*Look-Up Tables*). También existen otros tipos como los FIFOs, las puertas lógicas, etc, que son menos relevantes en este trabajo.
- **Bloques de memoria**: almacenan datos intermedios y constantes, implementados como BRAM (*Block RAM*). También existen otros tipos como los registros.

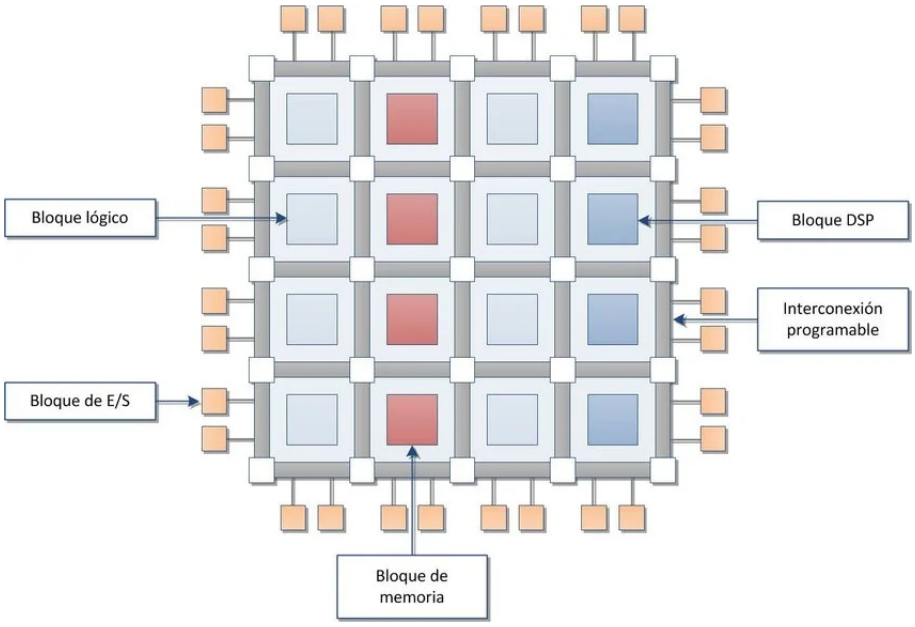

Figura 3.1: Diagrama de arquitectura de una FPGA. [14, Figura 2.15]

En el diseño de este trabajo, estos bloques se reparten de la siguiente forma: los DSPs se emplean para las multiplicaciones de las capas lineales del MLP, los bloques lógicos (LUTs) para la lógica de control y para aproximar las funciones SiLU y sigmoide mediante tablas de valores precalculados, y la BRAM para almacenar los pesos de la red y los resultados intermedios. La comunicación entre la FPGA y la CPU se gestiona mediante puertos AXI (*Advanced eXtensible Interface*). En cuanto a sus aplicaciones, las FPGAs son especialmente adecuadas para cualquier problema que requiera baja latencia, procesamiento en flujo o control determinista:

- **Instrumentación científica:** adquisición y procesamiento de datos en tiempo real en experimentos de física de partículas.
- **Telecomunicaciones:** procesamiento de señal y redes 4G/5G.
- **Automatización industrial:** control determinista de maquinaria, que es precisamente el contexto de este trabajo.

3.2. Razones para el uso de una FPGA

La ventaja principal de una FPGA frente a una CPU reside en la posibilidad de implementar un algoritmo directamente como un circuito hardware, explotando el paralelismo de forma adaptada al problema concreto. Para ilustrarlo, consideremos la operación central de cada capa del MLP: el producto escalar entre el vector de entrada $\mathbf{x}$ y los pesos $\mathbf{w}$ de la neurona. En una CPU, este cálculo se ejecuta secuencialmente, instrucción a instrucción:

```
y = 0
for i in range(N):
    y += w[i] * x[i]    # acumulacion secuencial
```

Listing 3.1: Producto escalar en Python: el bucle se ejecuta instrucción a instrucción en la CPU.

El código VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) equivalente describe el mismo cálculo, pero como un circuito, cada componente existe físicamente en la FPGA y las operaciones del bucle pueden ejecutarse en paralelo si se dispone de suficientes multiplicadores:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Tipo auxiliar: array de enteros de longitud variable
package tipos is
    type int_array is array (natural range <>) of integer;
end package;
-- ------------------------------------------------------
entity dot_product is
    generic (N : integer := 21);        -- entradas del MLP
    port (
        clk : in  std_logic;            -- senal de reloj
        w   : in  int_array(0 to N-1); -- pesos de la neurona
        x   : in  int_array(0 to N-1); -- vector de entrada
        y   : out integer               -- resultado: y = sum w[i]*x
           [i]
    );
end entity dot_product;

architecture rtl of dot_product is
begin
    process(clk)
        variable acc : integer := 0;
    begin
        if rising_edge(clk) then        -- ejecutar en cada pulso de
            reloj
            acc := 0;
            for i in 0 to N-1 loop
                acc := acc + w(i) * x(i);
            end loop;
            y <= acc;                   -- escribir resultado en la
               salida
        end if;
    end process;
end architecture rtl;
```

Listing 3.2: Producto escalar en VHDL: descripción hardware equivalente al código Python anterior. $N = 21$ corresponde al número de entradas de la primera capa del MLP.

La diferencia clave respecto al código Python es que en VHDL se *describen* los elementos del circuito (puertos, señales, registros) antes de describir su comportamiento. La sentencia `if rising_edge(clk)` sincroniza el cálculo con el reloj de la FPGA, garantizando que el resultado esté disponible de forma determinista en cada ciclo.

Con el factor de reutilización apropiado, el bucle `for` puede desplegarse (*unroll*) para instanciar varios multiplicadores hardware en paralelo, reduciendo la latencia a costa de mayor uso de área.

En este proyecto, uno de los principales atractivos de utilizar una FPGA es la posibilidad de ejecutar el modelo en el extremo de forma local y próxima al proceso industrial, es decir, integrada en el propio entorno de producción. Esta aproximación ofrece varias ventajas. En primer lugar, reduce la dependencia de infraestructuras de red externas, mejorando la seguridad y la robustez del sistema frente a posibles incidencias de comunicación. En segundo lugar, facilita las tareas de mantenimiento e integración al permitir que el sistema opere de manera autónoma dentro de la planta industrial. Finalmente, una FPGA puede conectarse directamente a sensores y actuadores mediante interfaces hardware específicas, lo que reduce la latencia y simplifica la adquisición y el procesamiento de datos en tiempo real.

Las FPGAs también ofrecen ventajas en consumo energético y latencia. Estas ventajas no probarán ser relevantes en la implementación del proyecto, debido a dificultades en la optimización del modelo que serán discutidas en los capítulos 4 y 5.

3.3. Punto fijo frente a punto flotante

Casi todo el software utiliza representación de punto flotante en la que el número se representa como $m \cdot 2^e$, donde m es la mantisa y e el exponente. En general esto nos permite representar un amplio rango de números con la misma precisión relativa a cada orden de magnitud. Por otro lado, esto es más costoso a nivel de área en el diseño y es más lento, por lo que en FPGA y ASIC suele preferirse otra representación.

Por otro lado tenemos la representación de punto fijo, en la que tenemos un número dado de bits para la parte entera del número y otro para la fraccionaria. Esto provoca que tengamos menos rango numérico y menos precisión. Pero podemos ajustarlo lo máximo posible a las necesidades del diseño. Esta representación tiene la ventaja de ocupar menos área y ser más rápida y determinista.

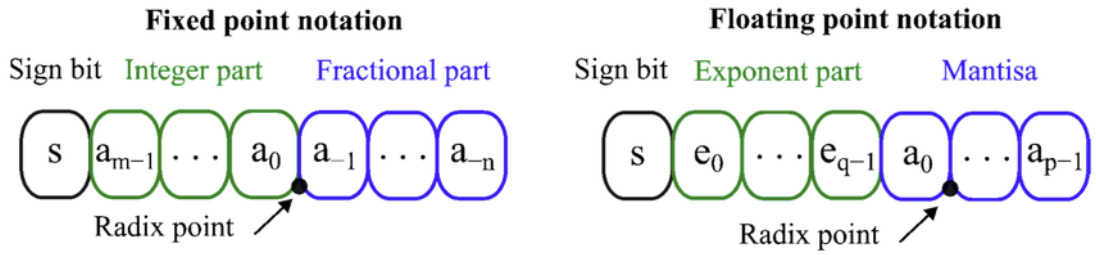

Figura 3.2: Comparación entre notación en punto fijo y punto flotante. Fuente: [15], Figura 1.

	Punto flotante	**Punto fijo**
Rango	Muy amplio	Limitado
Precisión	Alta y adaptativa	Fija en diseño
Coste hardware	Alto	Bajo
Velocidad	Menor	Mayor
Uso típico	CPU, GPU	FPGA, ASIC

Tabla 3.1: Comparativa entre representación en punto flotante y punto fijo.

3.4. Dispositivo experimental empleado

En esta sección se muestran diagramas del hardware utilizado, la Kria KV260[1], la cual, aparte de la FPGA posee una CPU para facilitar el sistema de control así como múltiples puertos de conexión. La tarjeta fue diseñada en principio para aplicaciones de visión por ordenador. Por eso posee varios puertos dedicados a la transmisión de video (véase Figura 3.3). A pesar de ello, la versatilidad de las FPGAs permite usar esta tarjeta para abordar el problema planteado en este trabajo.

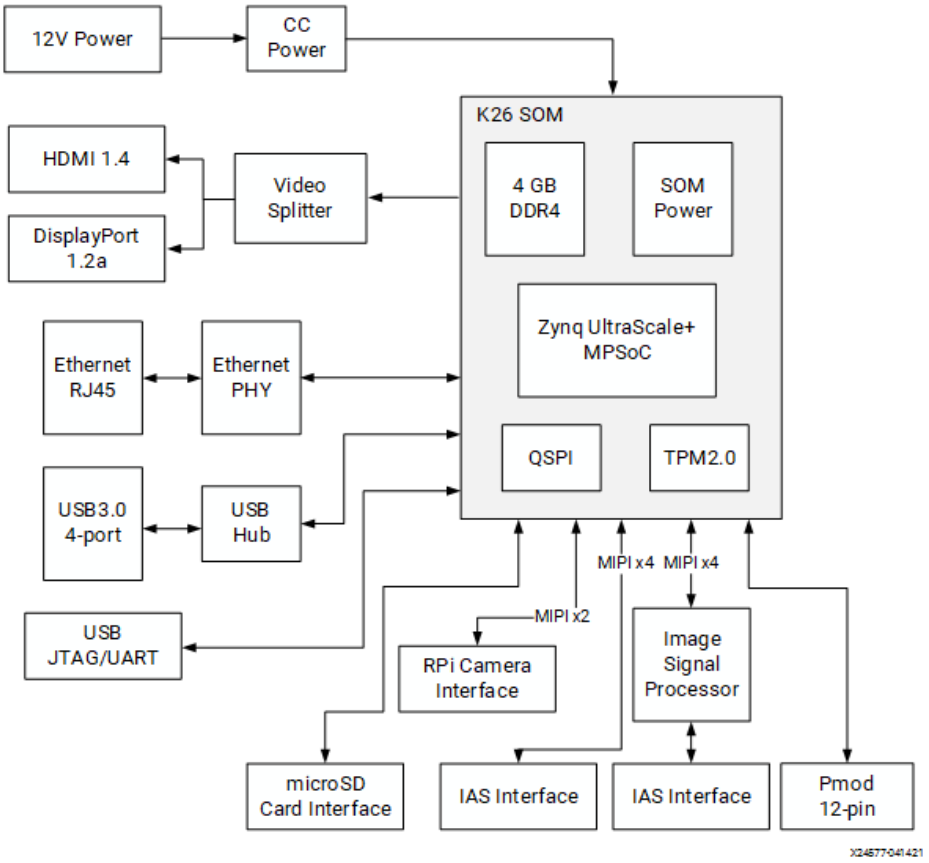

Figura 3.3: Diagrama de bloques del hardware usado.[16]

En este trabajo se han usado fundamentalmente el puerto de carga (12V Power), para suministrar energía, el puerto ethernet RJ45 para la conexión de red con el exterior, y la interfaz para tarjetas microSD (microSD Card Interface) para proveer el sistema operativo. Los múltiples puertos USB podrían servir en un futuro para conectarla a distintos sensores de máquinas.

[1]https://www.amd.com/es/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html

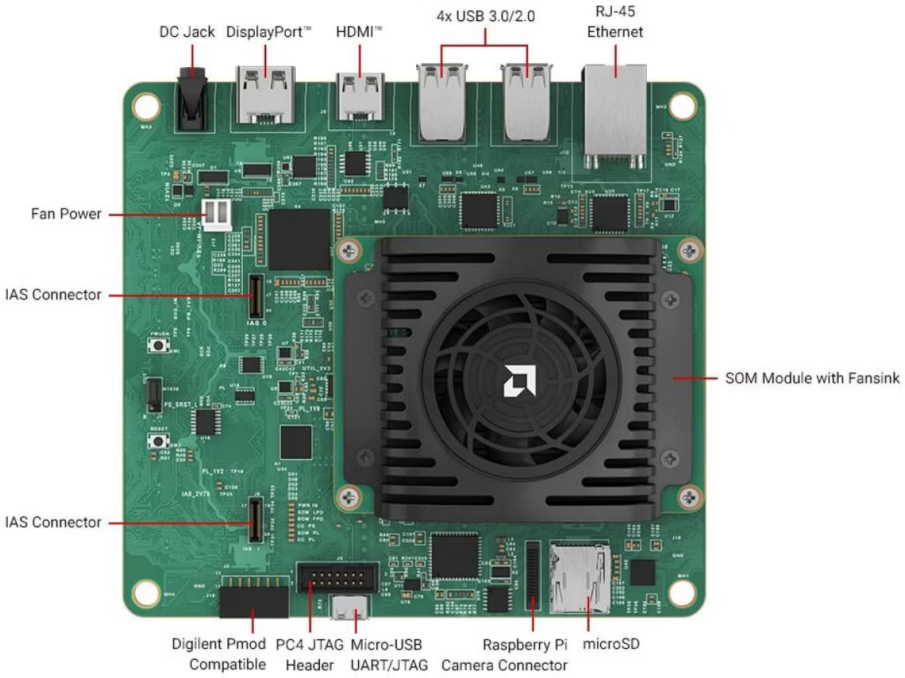

Figura 3.4: Imagen de la Kria.[17]

La tarjeta (ilustrada en la Figura 3.4) mide $140 \times 119\,mm$, lo que la hace adecuada para usos en local cerca de la maquinaria, ya que no ocupa mucho espacio. Su gran cantidad de puertos permite conectar sensores directamente a ella. Lo que la hace viable para el proyecto industrial. También se pueden crear soportes y cajas para esta mediante, por ejemplo, impresoras 3D, para protegerla del polvo y golpes.

Capítulo 4

Proceso de implementación en la FPGA

Para este trabajo se utilizó una Kria KV260 de AMD Xilinx™. En este capítulo se explica el proceso completo que se llevó a cabo para la traducción del modelo.

El software requerido para este trabajo es Vitis[18] y Vivado[19] de AMD, las librerías originales de Python y hls4ml.

4.1. Descripción del problema y flujo de trabajo

El objetivo de este capítulo es explicar cómo se implementó en una FPGA el algoritmo de optimización descrito en el Cap. 2. El flujo de trabajo completo se resume en la Figura 4.1.

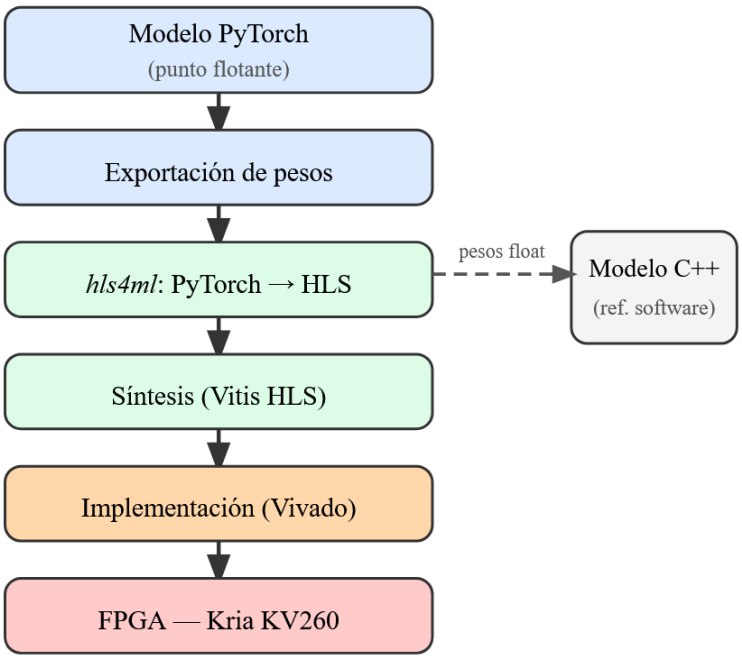

Figura 4.1: Flujo de trabajo completo del proyecto. La línea discontinua indica que el modelo C++ se desarrolló a posteriori, partiendo de los pesos exportados tras la traducción HLS, como referencia de software optimizado.

El punto de partida es el modelo MLP entrenado en PyTorch. Sus pesos se exportan y se traducen a código HLS mediante *hls4ml*, que genera automáticamente la implemen-

tación en punto fijo de la pasada hacia adelante. La retropropagación y el descenso de gradiente, al no estar soportados directamente por *hls4ml*, se implementaron manualmente en HLS. El esquema completo del algoritmo se muestra en la Figura 4.2: primero se calcula la pasada hacia adelante, luego se propaga el gradiente hacia atrás aplicando la regla de la cadena, y finalmente se aplica un paso de descenso de gradiente con saturación (*clamp*) a los límites físicos de cada variable de control (véase tabla 2.1). Este ciclo se repite hasta alcanzar las 1000 iteraciones o la tolerancia de 0,1 mg. Esta tolerancia es distinta a la tolerancia de medida. Como se pierde precisión al pasar de un modelo a otro se es más exigente con la condición de parada interna para mejorar la precisión.

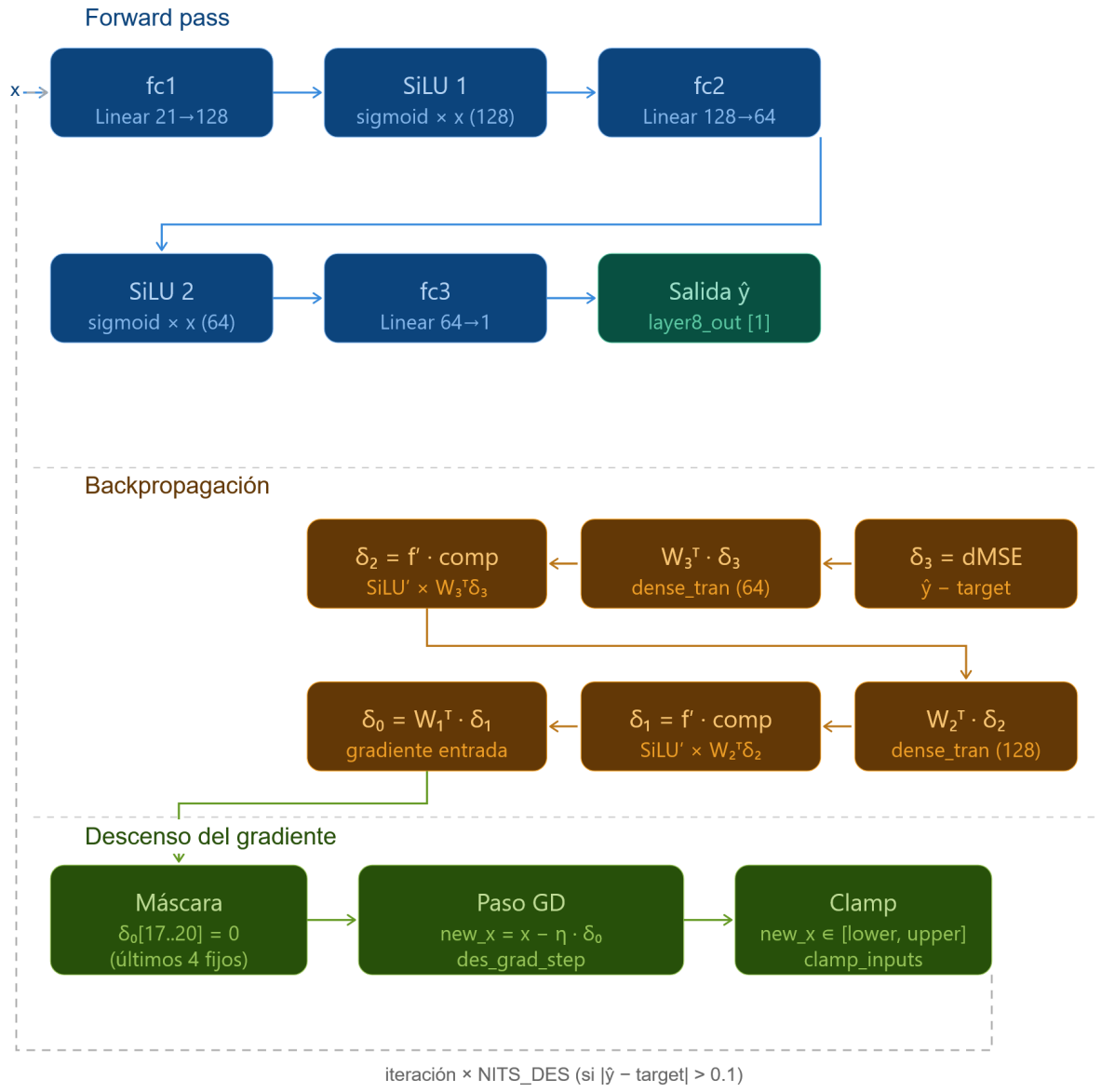

Figura 4.2: Esquema del algoritmo completo implementado en HLS: pasada hacia adelante, retropropagación y descenso de gradiente con saturación.

Una decisión de implementación relevante es el uso de *arrays* unidimensionales en lugar de bidimensionales para representar las matrices de pesos, ya que la gestión de memoria en la FPGA resulta más eficiente en ese formato. El código HLS resultante se

sintetiza con Vitis HLS y se implementa con Vivado, generando el fichero `.bit` que se carga en la placa Kria KV260 y el fichero `.hwh` que describe a la CPU la interfaz de comunicación (véase Sec. 4.3). Es importante tener en cuenta que las estimaciones de recursos obtenidas en la fase de síntesis de Vitis son optimistas, especialmente en el caso de las LUTs; los valores reales se obtienen únicamente tras la implementación completa en Vivado, y pueden diferir de forma notable (véase Cap. 5).

Una vez constatado que la FPGA superaba ampliamente en velocidad al modelo de PyTorch, se desarrolló a posteriori un modelo equivalente en C++ [20] usando los pesos originales en punto flotante de 32 bits, compilado con optimización máxima y ejecutado de forma nativa en la CPU de la Kria. Su propósito es proporcionar una referencia de software optimizado con la que comparar la FPGA en igualdad de condiciones de hardware. El sistema de control sigue la cadena PC remoto $\xrightarrow{\text{SSH}}$ CPU Kria $\rightarrow$ FPGA, gestionada mediante PYNQ [21].

4.2. Representación numérica y optimización de recursos

La primera decisión de diseño es la elección de los formatos de punto fijo para cada tipo de dato. En HLS estos se especifican como `ap_fixed<W,I>`, donde W es el número total de bits e I los dedicados a la parte entera, dejando $W - I$ bits para la parte fraccionaria. Para hacer un uso eficiente de la memoria, los tamaños totales se eligen como potencia de dos o múltiplo de 8. Los formatos utilizados en este trabajo son:

- **Pesos** (`ap_fixed<16,5>`): 16 bits con 5 para la parte entera. Rango suficiente para los pesos normalizados del modelo tras el entrenamiento.
- **Activaciones y vector de entrada** (`ap_fixed<24,13>`): 24 bits con 13 para la parte entera, necesarios para cubrir el rango de salida del modelo (0–400 mg) sin desbordamiento. Estos nos dan un error relativo de mediana $\sim 0{,}6\,\%$ para el MLP.
- **Learning rate** (`ap_fixed<24,1>`): 24 bits con solo 1 para la parte entera, maximizando la precisión fraccionaria para un valor que idealmente es un número decimal pequeño.
- **LUTs de la función sigmoide** (`ap_fixed<18,8>`): 18 bits con 8 para la parte entera, dimensionadas para cubrir el rango de las activaciones intermedias de la red con precisión suficiente.

La segunda decisión es la configuración de los factores de reutilización para cada capa, que determinan el equilibrio entre latencia y uso de recursos. La función sigmoide, demasiado costosa para implementarse con DSPs, se aproxima mediante tablas de consulta (LUTs) de 1024 entradas; las mismas tablas se reutilizan en la retropropagación para calcular la derivada de SiLU, Ec. (2.7). La tabla 4.1 resume los valores elegidos para cada operación de la pasada hacia adelante y la retropropagación.

Además de los factores de reutilización, se aplica segmentación *pipelining* a los bucles principales (véase Figura 4.3), permitiendo que cada iteración comience tan pronto como los datos de la anterior estén disponibles, reduciendo la latencia total sin incrementar el uso de recursos.

Capa	Operación	Reuse Factor	Multiplicaciones/ciclo
Pasada hacia adelante			
Capa 1	Linear $21 \to 128$	168	$\lceil 2688/168 \rceil = 16$
Activación 1	SiLU_1 (sigmoid)	128	— (tabla, 1024 entradas)
Capa 2	Linear $128 \to 64$	256	$\lceil 8192/256 \rceil = 32$
Activación 2	SiLU_2 (sigmoid)	64	— (tabla, 1024 entradas)
Capa 3	Linear $64 \to 1$	64	$\lceil 64/64 \rceil = 1$
Retropropagación			
Capa traspuesta 3	$W_3^T \cdot \delta_3$	1	$\lceil 64/1 \rceil = 64$
Capa traspuesta 2	$W_2^T \cdot \delta_2$	256	$\lceil 8192/256 \rceil = 32$
Capa traspuesta 1	$W_1^T \cdot \delta_1$	384	$\lceil 2688/384 \rceil = 7$

Tabla 4.1: *Reuse factors* configurados en el proyecto. Las activaciones SiLU usan una LUT de sigmoide de 1024 entradas en lugar de DSPs; las mismas tablas se emplean en la retropropagación para la derivada de la función de activación.

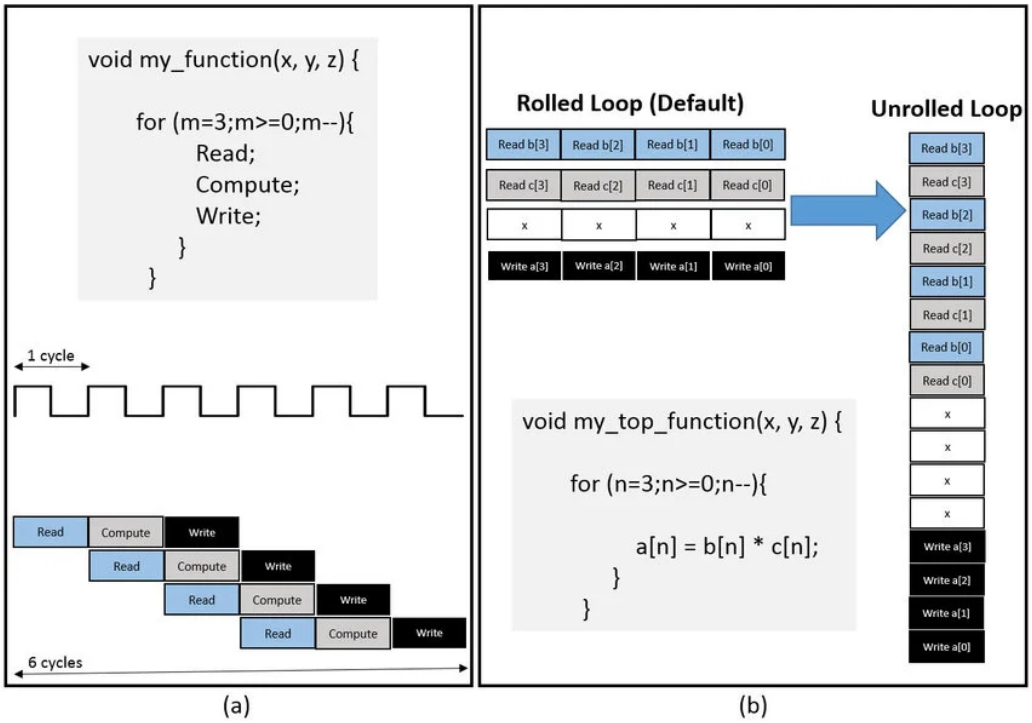

Figura 4.3: Comparación entre segmentación de bucles y desenrollado de bucles. Fuente: [22].

Tanto el desenrollado de bucles como la segmentación, tienen como objetivo reducir la latencia del diseño lo máximo posible.

4.3. Generación de la interfaz de comunicación con Vivado

Para que el kernel HLS pueda comunicarse con la CPU es necesario integrarlo en un diseño de bloques (*block design*, BD) de Vivado, que define las conexiones hardware

entre los distintos bloques del sistema. El BD utilizado en este trabajo, mostrado en la Figura 4.4, consta de cuatro elementos principales:

- **Zynq UltraScale+ MPSoC:** el procesador ARM de la Kria, que ejecuta el sistema operativo y el script PYNQ. Expone dos tipos de puertos AXI: un puerto de altas prestaciones (`S_AXI_HP0_FPD`) para transferencias de datos en masa hacia DDR, y un puerto de propósito general para control (`M_AXI_HPM0_FPD`).
- **IP HLS** (`mlp_galva_desgrad_co_0`): el kernel sintetizado que implementa el algoritmo completo. Expone un puerto `s_axi_control` para recibir escalares y punteros de memoria, y un puerto `m_axi_gmem` para acceder directamente a los buffers DMA en DDR.
- **AXI SmartConnect:** interconexiones que enrutan el tráfico AXI entre el procesador y la IP, que el hardware creado a medida para resolver el algoritmo. Un SmartConnect conecta el CPU al IP para transferencia de datos; otro gestiona el canal de control.
- **Processor System Reset:** módulo que gestiona las señales de reset sincronizadas para todos los bloques del diseño.

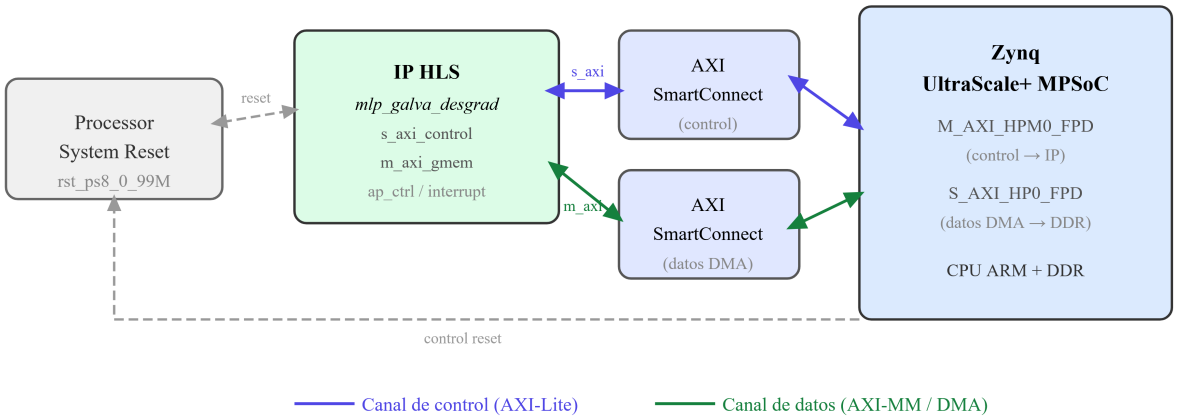

Figura 4.4: Esquema simplificado del BD. Los bloques son los módulos de hardware y las flechas indican las conexiones entre ellos.

4.4. Interfaz de control usando PYNQ

Para comunicar la CPU de la Kria con el kernel HLS se utilizó PYNQ, una librería de Python que abstrae el protocolo AXI-Lite y la gestión de memoria DMA. El flujo de ejecución de cada optimización consta de cuatro pasos:

1. **Carga de la capa de configuración (*overlay*).** Se carga el fichero `.bit` en la FPGA y se obtiene un manejador (*handle*) al IP `mlp_galva_desgrad_complete`. Dado que el protocolo de control empleado (`ap_ctrl_chain`) requiere un reset entre ejecuciones, la capa de configuración se recarga antes de cada llamada; este es el principal responsable de la sobrecarga (*overhead*) de tiempo observado en la Sección 5.3.

2. **Codificación y escritura de entradas.** El vector de entrada de 21 variables se convierte de punto flotante a punto fijo (`ap_fixed<24,13>`, escala 2^{11}) y se escribe en un buffer de memoria físicamente contigua reservado con `pynq.allocate()`. La tasa de aprendizaje se codifica con formato `ap_fixed<24,1>` (escala 2^{23}, rango $(-1, 1)$). Las direcciones físicas de los buffers y los valores escalares (`target`, `lr`) se escriben en los registros AXI-Lite del IP.

3. **Ejecución y espera.** Se activa la señal `AP_START` y se espera la señal `AP_DONE` o `AP_IDLE` mediante un bucle de sondeo (*polling*) con tiempo de espera límite (*timeout*) de seguridad. La latencia esperada del kernel, según el informe de síntesis, es de ∼145 ms para las 1000 iteraciones internas.

4. **Lectura y decodificación de resultados.** Se invalidan las líneas de caché de la CPU para forzar la lectura desde DDR y se decodifican los buffers de salida: `new_x` (vector de variables optimizadas, 21 elementos) y `layer_8_out` (grosor predicho al final de la optimización), ambos en formato `ap_fixed<24,13>`.

La conversión explícita entre punto flotante y punto fijo es necesaria porque el bus AXI transfiere los valores como enteros de 32 bits; sin la codificación correcta, los valores numéricos que recibe el kernel serían erróneos. Igualmente, las llamadas a `flush()` e `invalidate()` sobre los buffers son imprescindibles para mantener la coherencia de caché entre la CPU ARM y el IP hardware.

4.5. Resumen del proceso global de implementación y dificultades encontradas.

El proceso fue largo y de unos 5 meses de duración de trabajo continuo casi diario, esto se debió a que el proceso no tiene un camino fijo. Las optimizaciones se afectan entre sí, modificar una para reducir el uso de un recurso suele aumentar el uso de otro y/o reducir significativamente la latencia del diseño.

Para el proceso se utilizó un repositorio (sistema de almacenaje de versiones de código) en el que se generaron dos ramas principales aplicando dos paradigmas de optimización, la división era usar listas de pesos unidimensionales o bidimensionales[1]. En cada una se iban aplicando las optimizaciones en paralelo. Para validar una optimización se tiene que hacer la síntesis de Vitis, que lleva, en este caso una hora cada vez, así como la implementación que dura aproximadamente 6 horas y la cosimulación (que simula los resultados numéricos del hardware generado para validar la implementación) que son 4 horas. Como se mencionará en el capítulo 6, al Vitis requerir de mucha RAM forzó a correr el código en la infraestructura de computación científica del grupo de investigación ICTEA en el C^3.

Otra dificultad encontrada fue la falta de documentación. Al ser una tecnología poco popular, la documentación y tutoriales referentes a ella es muy escasa y, en general, de poca calidad. Con lo que se requirió de la ayuda de los ingenieros del grupo de investigación y de un proceso de prueba y error.

[1]La implementación final, como se mencionó anteriormente, usa listas unidimensionales, que es lo que probó más eficiente.

4.6. Metodología de comparación

Para evaluar el programa de FPGA se hace una comparación completa de la viabilidad del software, la precisión y las alternativas. La comparación se hace usando 3 versiones del programa:

- **Modelo original (PyTorch/Python):** se toma como referencia de precisión. Se asume que sus predicciones son el valor de referencia frente al que se comparan las demás implementaciones.
- **Modelo FPGA:** el modelo desarrollado con HLS usando pesos en punto fijo.
- **Modelo C++:** implementado después del modelo FPGA como comparación con software optimizado, usando los pesos originales en punto flotante de 32 bits y compilado con el nivel máximo de optimización. Corre nativamente en la CPU de la Kria.

Los vectores optimizados por cada modelo se evalúan con la MLP original de PyTorch, que se toma como salida de referencia alcanzada por la optimización. Conviene aclarar que el modelo de PyTorch no representa la realidad física del proceso, sino la referencia software; el valor real correspondería a una medida industrial del grosor.

Capítulo 5

Resultados

Este capítulo presenta los resultados de la comparación entre las tres implementaciones del algoritmo de optimización desarrolladas en el Cap. 4: el modelo original en PyTorch, la implementación en FPGA mediante HLS, y el modelo de referencia en C++. Salvo que se indique lo contrario, todas las comparativas se realizan bajo las mismas condiciones experimentales: $lr = 5{\cdot}10^{-4}$, tolerancia $tol = 1{,}5\,mg$, 10 muestras como puntos iniciales de optimización y 7 objetivos en el rango de operación más habitual. Los vectores optimizados por cada modelo se evalúan siempre con el MLP original de PyTorch como referencia de salida. Conviene recordar que durante la optimización en FPGA y C++ únicamente se actualizan las variables de control, mientras que las variables de estado permanecen fijas; la clasificación completa de variables se recoge en la tabla 2.1.

5.1. Uso estimado de recursos en síntesis e implementación

La tabla 5.1 recoge las estimaciones de latencia y recursos obtenidas durante la síntesis en Vitis HLS, y la tabla 5.2 los valores reales medidos tras la implementación en Vivado. Ambas tablas no son directamente comparables entre sí, ya que Vitis y Vivado operan en etapas distintas del flujo de diseño y utilizan modelos de recursos diferentes.

Módulo / Bucle	Latencia (ciclos)	Latencia (ms)	Intervalo	BRAM	DSP	FF	LUT
`mlp_galva_desgrad_complete`	14 536 001	145,4	14 536 002	150 (52 %)	563 (45 %)	76 242 (32 %)	81 166 (69 %)
↪ *Iter_Loop* ×1000	14 536 000	145,4	—	—	—	—	—
Capa 1 — Linear 21 → 128	241	$2{,}41 \times 10^{-3}$	168	—	16 (1 %)	6 531 (2 %)	9 440 (8 %)
↪ *ReuseLoop* ×168	240	$2{,}40 \times 10^{-3}$	1	—	—	—	—
Activación 1 — SiLU 128	0	0	1	—	256 (20 %)	—	5 120 (4 %)
Capa 2 — Linear 128 → 64	257	$2{,}57 \times 10^{-3}$	256	—	32 (2 %)	6 173 (2 %)	13 428 (11 %)
↪ *ReuseLoop* ×256	256	$2{,}56 \times 10^{-3}$	1	—	—	—	—
Activación 2 — SiLU 64	0	0	1	—	128 (10 %)	—	2 560 (2 %)
Capa 3 — Linear 64 → 1	134	$1{,}34 \times 10^{-3}$	64	1 (∼0 %)	1 (∼0 %)	1 798 (∼0 %)	2 343 (2 %)
Derivada MSE (δ_3)	0	0	1	—	—	—	31 (∼0 %)
Capa traspuesta 3 — $W_3^T \cdot \delta_3$	0	0	0	—	64 (5 %)	—	2 560 (2 %)
Capa traspuesta 2 — $W_2^T \cdot \delta_2$	258	$2{,}58 \times 10^{-3}$	256	—	32 (2 %)	7 720 (3 %)	12 584 (10 %)
↪ *ReuseLoop* ×256	257	$2{,}57 \times 10^{-3}$	1	—	—	—	—
Capa traspuesta 1 — $W_1^T \cdot \delta_1$*	4 226	$4{,}226 \times 10^{-2}$	4 224	—	7 (∼0 %)	7 270 (3 %)	6 906 (5 %)
↪ *ReuseLoop* ×384	4 225	$4{,}225 \times 10^{-2}$	11	—	—	—	—
Máscara — *Mask_Loop* ×21	23	$2{,}3 \times 10^{-4}$	—	—	—	7 (∼0 %)	64 (∼0 %)
Descenso del gradiente ×21	3 002	$3{,}002 \times 10^{-2}$	21	—	21 (1 %)	6 469 (2 %)	4 610 (3 %)

* Presenta *II Violation* por limitación de recursos (II = 11, objetivo = 1).

Tabla 5.1: Estimación de latencia y recursos del módulo `mlp_galva_desgrad_complete` obtenida con la síntesis de Vitis HLS (valores estimados, no implementados). La latencia se expresa en ciclos y en milisegundos.

La estimación de síntesis indica que el diseño cabe en el dispositivo y cumple los requisitos. El recurso más consumido son las LUTs (∼69 %), empleadas tanto para la lógica de control como para las tablas de la función sigmoide; los flip-flops no superan el 32 %, y la BRAM y los DSPs se sitúan en torno al 50 % y 45 % respectivamente. La latencia total del kernel es de ∼145 ms para las 1000 iteraciones internas a 100 MHz; los tiempos *end-to-end* medidos desde PYNQ se discuten en la Sección 5.3. Cabe destacar que se decidió **no incluir los BDTs** en la implementación: su tamaño habría obligado a aumentar el factor de reutilización hasta el punto de eliminar la ventaja de latencia frente a una solución software[1]. Con este informe se concluyó que era viable continuar con la implementación.

Categoría	Parámetro	Valor
Utilización de recursos	CLB LUTs (total)	47 693 / 117 120 (40,72 %)
	LUT como lógica	44 063 / 117 120 (37,62 %)
	LUT como memoria	3 630 / 57 600 (6,30 %)
	CLB Registers (FF)	63 320 / 234 240 (27,03 %)
	CLB Slices	10 338 / 14 640 (70,61 %)
	CARRY8	1 305 / 14 640 (8,91 %)
	Block RAM Tile	74 / 144 (51,39 %)
	URAM	0 / 64 (0,00 %)
	DSPs	565 / 1 248 (45,27 %)
Timing	Frecuencia de reloj	100 MHz
	Periodo de reloj	10,000 ns
	Worst Negative Slack (WNS)	+0,223 ns
	Total Negative Slack (TNS)	0,000 ns
	Worst Hold Slack (WHS)	+0,033 ns
	Worst Pulse Width Slack (WPWS)	+4,238 ns
	Constraints cumplidos	Sí (0 endpoints fallando)
Potencia (estimada post-implementación)	Relojes	0,096 W
	Lógica CLB	0,116 W
	Señales	0,211 W
	Block RAM	0,029 W
	DSPs	0,077 W
	Potencia dinámica	**0,530 W**
	Potencia estática	0,291 W
	Potencia total estimada	**0,821 W**
	Temperatura de juntura	26,9 °C (máx. ambiente: 83,1 °C)

Tabla 5.2: Resumen de la implementación *post-route* del módulo `mlp_galva_desgrad_complete` en la Kria KV260 (xck26-sfvc784-2LV-c), obtenido con Vivado. Las cifras de potencia son una estimación post-implementación del diseño, no el consumo total medido de la placa. Nótese que las LUTs reales (∼40 %) son notablemente inferiores a la estimación de síntesis (∼69 %), lo que ilustra el carácter optimista de las estimaciones de Vitis.

Los reports que se generan a partir de la implementación no se pueden correlacionar 1:1 con el de la síntesis; por eso la tabla es distinta. De esta tabla podemos destacar:

- Las LUTs reales están en torno al 40 %, un 30 % inferior a lo predicho por la síntesis.
- Las predicciones de BRAM y DSPs fueron precisas en la síntesis.
- El programa alcanza los 100 MHz, es decir, 10^8 ciclos de reloj por segundo.

[1]Si se comprime el modelo y se aumenta el factor de reutilización se pierde la ventaja de latencia frente a una implementación por software.

- Los *slices* se sitúan en torno al 70 %, por lo que, aunque el diseño cabe y cumple *timing*, el margen no es holgado.
- La potencia estimada del diseño implementado es de 0,821 W. Esto no tiene en cuenta el software que corra en la CPU.

Después de esto se creó el programa completo con Vivado y se pasó a la FPGA, la cual se controla remotamente con PYNQ.

5.2. FPGA vs PyTorch

Ahora se muestran los resultados de los distintos modelos y las comparativas. **A partir de esta sección, todas las figuras son de generación propia.** Salvo que se indique lo contrario, las optimizaciones de esta sección se ejecutan con $lr = 5 \cdot 10^{-4}$ y una tolerancia de 1,5 mg sobre el modelo de FPGA, y el modelo de PyTorch se toma como referencia.

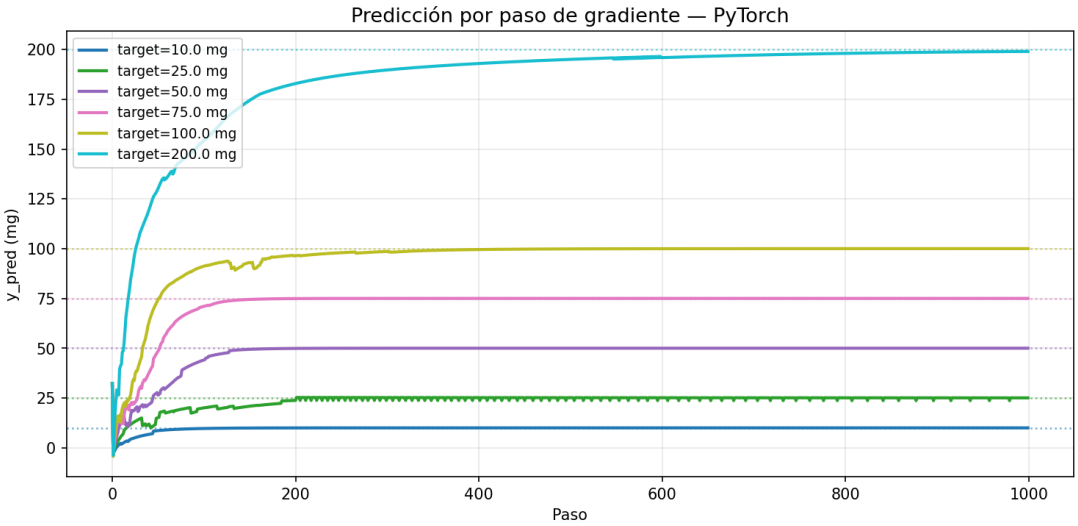

Figura 5.1: Convergencia del modelo de PyTorch a lo largo de las iteraciones de optimización ($lr = 10^{-5}$, partida fija común a todo el trabajo). Cada línea representa un *target* distinto. El eje y muestra el error absoluto respecto al objetivo en cada iteración.

Esta convergencia es usando un $lr = 10^{-5}$, que es menor que el usado en la FPGA ($lr = 5 \cdot 10^{-4}$); esto es para una mejor ilustración del proceso. Aumentar el lr hace mucho más errático el proceso, pero para la FPGA es necesario por falta de precisión: si se reduce, aumentan los casos en los que el gradiente se va a 0 antes de llegar.

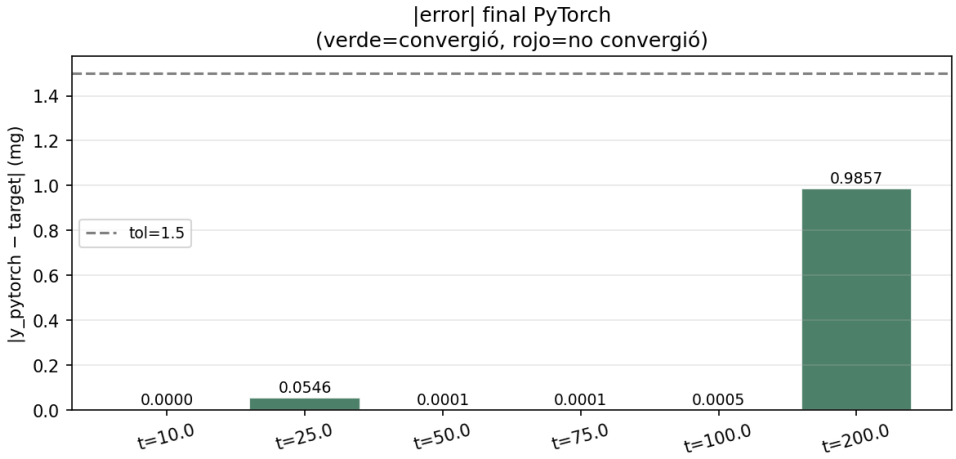

Figura 5.2: Errores del modelo original al final de la optimización (1000 pasos, $lr = 10^{-5}$, tolerancia $= 1{,}5\,mg$).

Aquí se ilustran los errores obtenidos al final de los 1000 pasos de optimización por el modelo original. El modelo original es preciso pero tarda una media de $15 \pm 1\,s$ en optimizar[2]. Si se aumentan los pasos o ligeramente el lr se llega a $0\,mg$ de error en todos los casos con la precisión manejada.

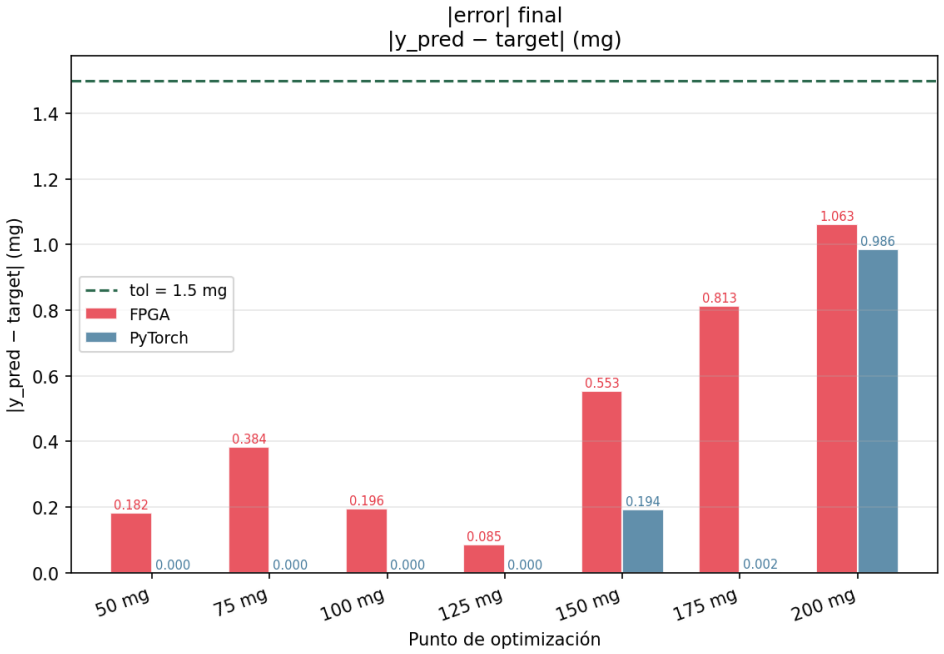

Figura 5.3: Comparación del error final de optimización entre el modelo de PyTorch y el de la FPGA para un mismo *target* (partida fija común; FPGA con $lr = 5 \cdot 10^{-4}$, PyTorch con $lr = 10^{-5}$). La línea horizontal discontinua indica la tolerancia de $1{,}5\,mg$.

En la figura 5.3 se puede observar que tanto la FPGA como el modelo original llegan

[2]La kria no disponía de las versiones correctas de las librerías para correr el modelo original. Debido a esto, los tiempos del modelo de PyTorch se midieron en una CPU de portátil de gama media. En concreto un Ryzen AI 7 350, con lo que no son 100 % comparables pero en un dispositivo local de bajo consumo se esperan tiempos similares.

al *target* de manera satisfactoria. Se muestra siempre el mismo punto de partida de la optimización a lo largo del trabajo, con el fin de eliminar fuentes de incertidumbre en las comparaciones.

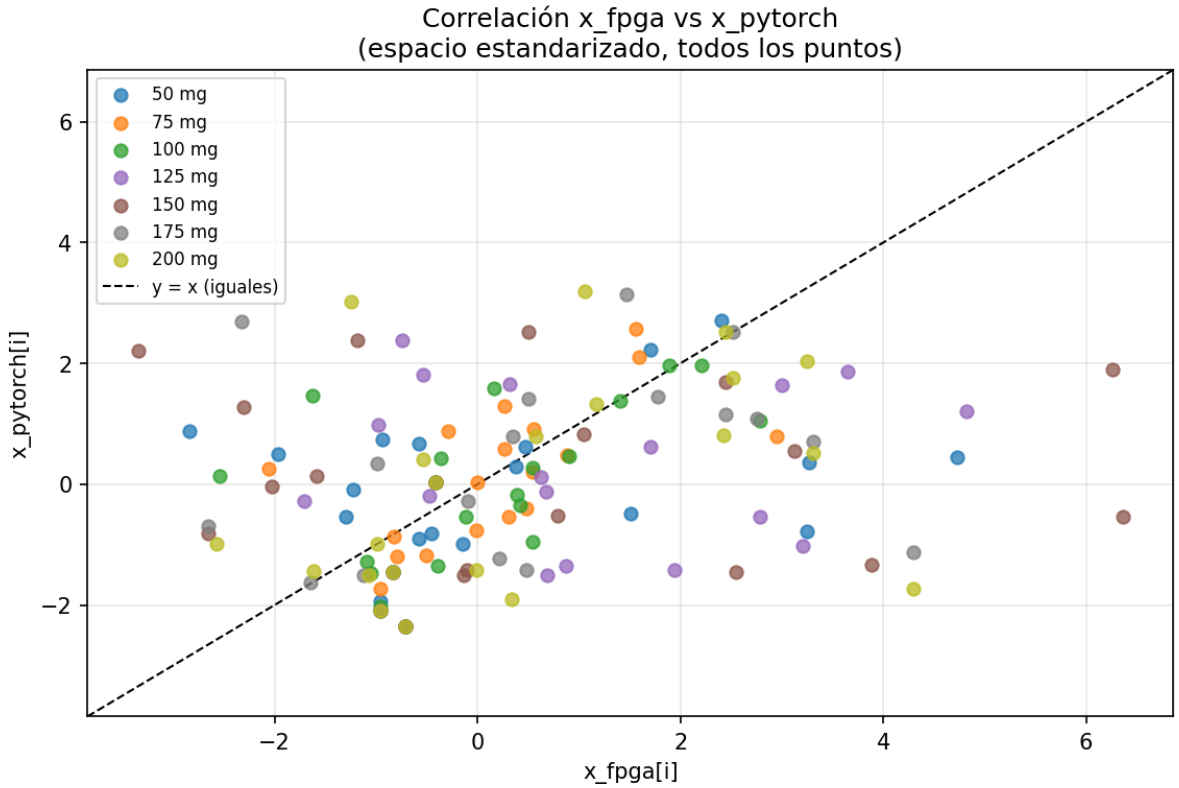

Figura 5.4: Diagrama de correlation de las *features* optimizadas, FPGA (eje x) frente a PyTorch (eje y). Una coincidencia perfecta daría la recta $x = y$. Cada punto representa el valor final de una *feature* tras la optimización; la dispersión respecto a la diagonal indica que ambos modelos convergen a mínimos locales distintos.

La Figura 5.4 muestra que los vectores optimizados por la FPGA y por PyTorch difieren considerablemente: los puntos se alejan de la diagonal $x = y$, indicando que ambos modelos convergen a mínimos locales distintos. Esto es esperable por dos razones. En primer lugar, el descenso de gradiente no garantiza convergencia al mismo mínimo cuando se parte de condiciones numéricas distintas: la aritmética de punto fijo de la FPGA introduce pequeñas diferencias en cada paso que acumuladas a lo largo de las iteraciones pueden desviar la trayectoria hacia una región diferente del espacio de parámetros. En segundo lugar, y más determinante, la FPGA no actualiza las variables de estado durante la optimización al no disponer de los BDTs: estas cuatro variables permanecen fijas en sus valores iniciales, lo que restringe el espacio de búsqueda y fuerza un camino de gradiente diferente al de PyTorch. Este hecho implica que la comparación no es estrictamente 1:1. Sin embargo, la existencia de mínimos distintos no invalida el resultado siempre que el error final sea pequeño: el espacio de parámetros del proceso de galvanizado tiene múltiples mínimos equivalentes, y lo relevante industrialmente es que el grosor predicho se aproxime al objetivo, no que el vector de parámetros coincida. La relevancia con respecto al error cometido de recorrer un camino de optimización distinto no se pudo medir, ya que requiere de medidas reales en el proceso industrial.

5.3. FPGA vs C++

En esta parte se hace una comparación entre la implementación de C++ y la de la FPGA, evaluadas en una lista de 10 muestras seleccionadas de una base de datos proporcionada por TheNextPangea SL™, con 7 objetivos que se encuentran en el rango más común. La tolerancia para las optimizaciones de esta sección es de $tol = 1{,}5\,mg$, y el $lr = 5 \cdot 10^{-4}$. Los vectores optimizados se evalúan en el modelo MLP de PyTorch para obtener el grosor de referencia optimizado.

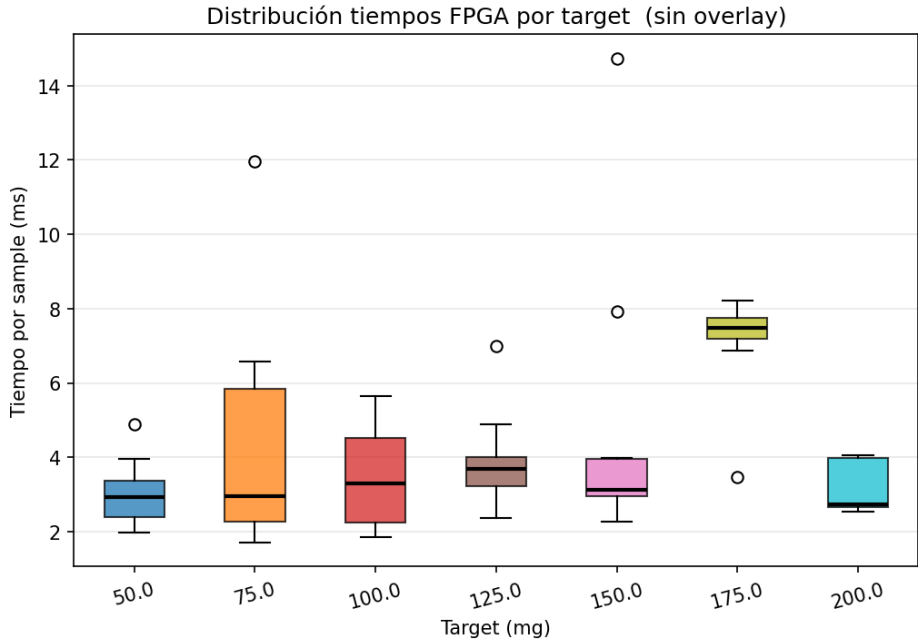

Figura 5.5: Tiempo de ejecución del kernel de FPGA por cada muestra y *target* (10 muestras, 7 objetivos, $lr = 5 \cdot 10^{-4}$, $tol = 1{,}5\,mg$, sin incluir la recarga de la capa de configuración). Se usa un diagrama de caja y bigotes; para su interpretación.

Como se puede observar, los tiempos de ejecución del kernel de la FPGA están en promedio por debajo de los $8\,ms$, es decir, 4 órdenes de magnitud por debajo del modelo de PyTorch. Estos tiempos corresponden a la salida del bucle tras pocas decenas de iteraciones debido a la tolerancia interna. En el sistema de FPGA es difícil incluir un sistema para cuantizar las iteraciones internas por lo que no se tienen datos exactos al respecto.

Cabe aclarar que el modelo pierde velocidad significativa con la capa de configuración, aumentando el tiempo de ejecución por cada optimización unos 3 órdenes de magnitud. También aumenta la incertidumbre del tiempo de ejecución, ya que vuelve el proceso más sensible a ralentizaciones por procesos externos del sistema. En general esto se traduce en una ralentización de $\sim 1\,s$ por optimización.

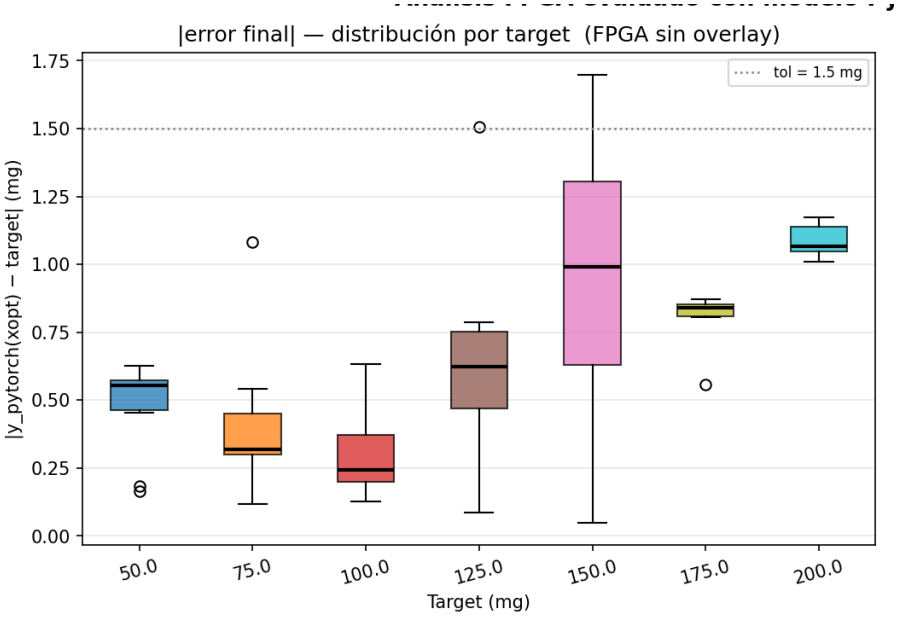

Figura 5.6: Error final del modelo de FPGA por cada muestra y *target* (10 muestras, 7 *objetivos*, $lr = 5 \cdot 10^{-4}$, $tol = 1{,}5\,mg$).

En la figura 5.6 se pueden observar los errores del modelo de FPGA. Si bien es cierto que no se llega a un 100 % de convergencia para todos los objetivos, ninguno es muy grande.

Conviene distinguir con cuidado qué mide cada tiempo. La latencia anterior corresponde al tiempo de cómputo del kernel en la FPGA. Sin embargo, por cómo está montado el sistema, se tiene que recargar la capa de configuración de PYNQ cada vez que se quiere volver a usar; este es un coste de inicialización/control, no del cálculo hardware en sí. Distinguimos por tanto tres medidas: (i) el tiempo del kernel FPGA, (ii) el tiempo *end-to-end* desde Python/PYNQ y (iii) el tiempo incluyendo la recarga de la capa de configuración.

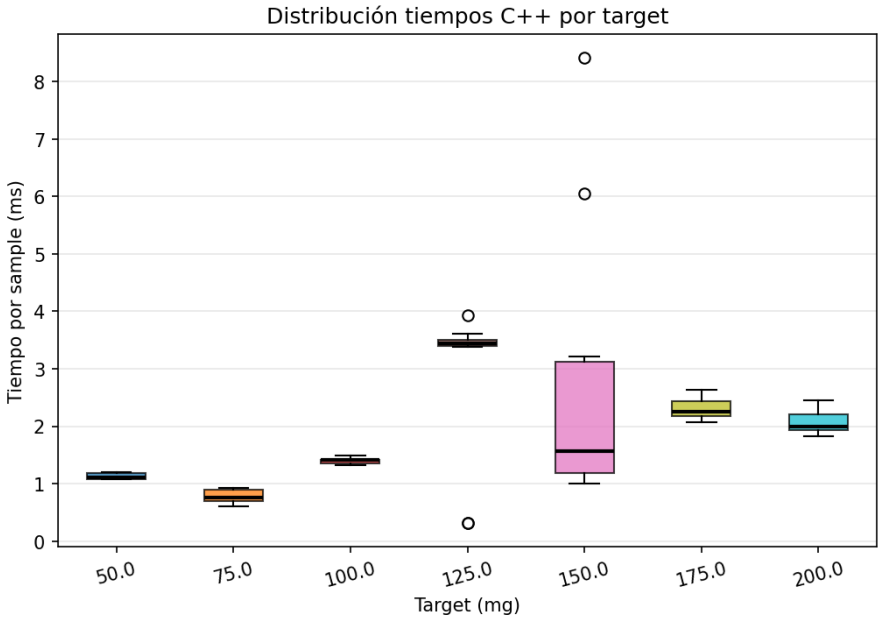

Figura 5.7: Tiempo de ejecución del modelo de C++ por cada muestra y *target* (10 muestras, 7 objetivos, $lr = 5 \cdot 10^{-4}$, $tol = 1{,}5\,mg$, ejecución nativa en la CPU de la Kria).

La figura 5.7 es la gráfica equivalente a la 5.5 para el modelo de C++.

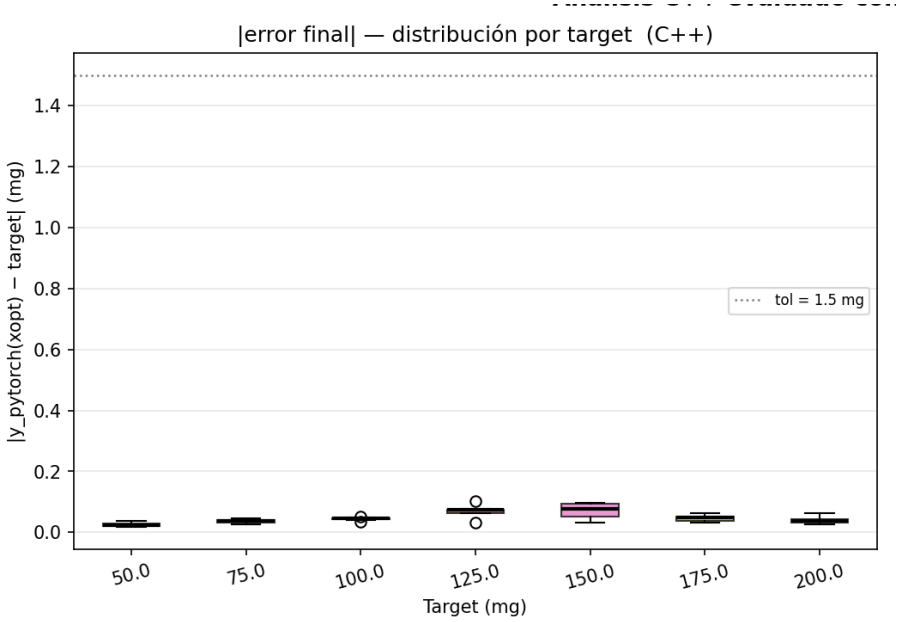

Figura 5.8: Error final del modelo de C++ por cada muestra y *target* (mismas condiciones que la Fig. 5.6).

Como se puede apreciar, la precisión es mejor, ya que utiliza los pesos originales en *float* de 32 bits. Además es en media ligeramente más rápido y con desviaciones mucho menores que el de FPGA, sobre todo si se tiene en cuenta el retraso (*delay*) que añaden los reseteos de la capa de configuración.

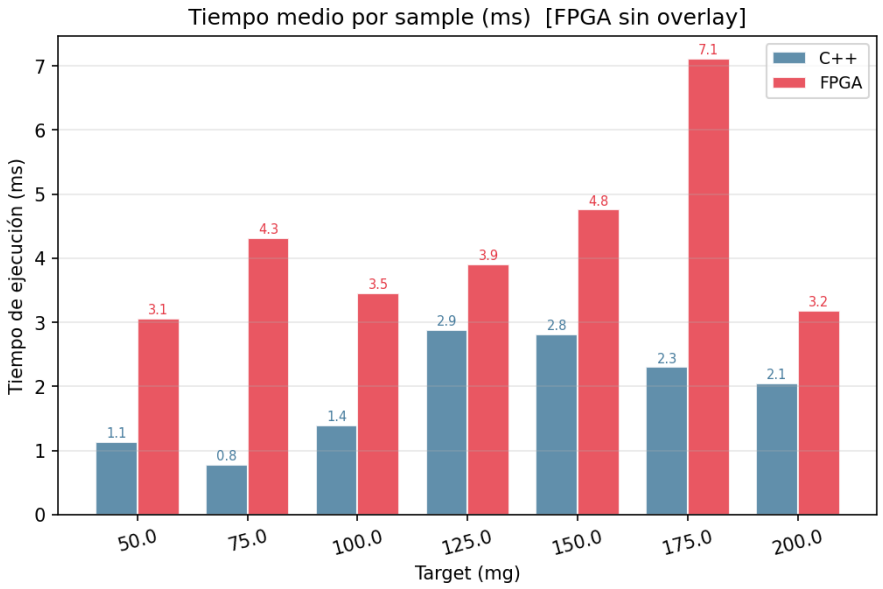

Figura 5.9: Comparación global de tiempos de ejecución de FPGA (kernel, sin capa de configuración) frente al modelo de C++.

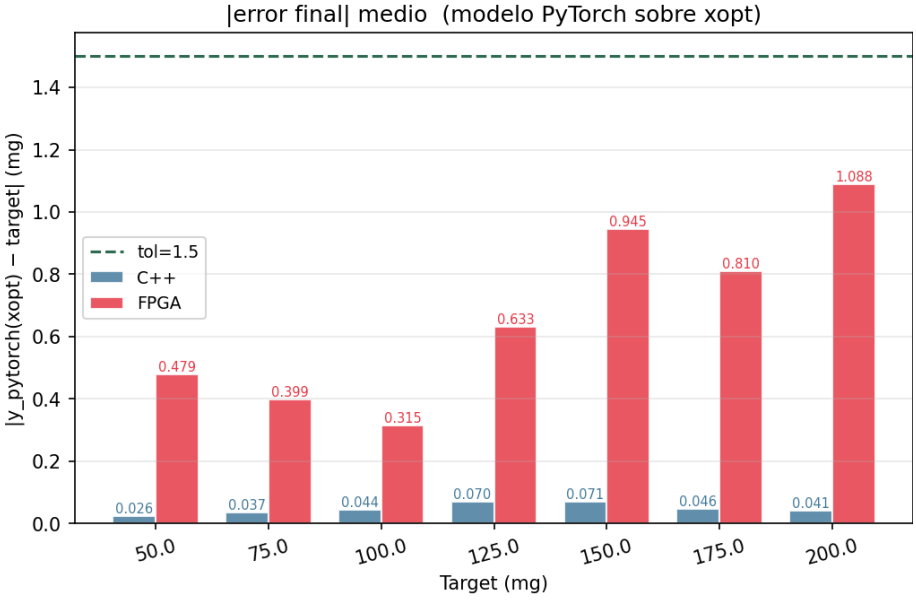

Figura 5.10: Comparación global de errores de FPGA frente al modelo de C++ (10 muestras, 7 objetivos, $lr = 5 \cdot 10^{-4}$, $tol = 1{,}5\,mg$).

Aquí se muestra una comparativa completa entre el modelo de C++ y FPGA. Se observa que el modelo de C++ es más rápido en media en todos los casos, y que es más preciso en todos ellos (salvo en un valor atípico (*outliers*) de $target = 150\,mg$ en el que el modelo de FPGA es más preciso).

En cuanto a los tiempos totales de ejecución, comparando las tres implementaciones bajo sus condiciones de uso reales: el modelo de C++ es el más rápido, con tiempos del orden de pocas decenas de milisegundos; el kernel de FPGA tiene una latencia comparable, pero la sobrecarga de PYNQ añade en torno a $1\,s$ adicional por ejecución al tener que recargar la capa de configuración; y el modelo de PyTorch es el más lento, con tiempos en

torno a 14 s por optimización. Esta diferencia de varios órdenes de magnitud entre PyTorch y las implementaciones en hardware se discute con más detalle en las conclusiones.

5.4. FPGA vs C++ vs PyTorch

Aquí se muestra una comparación final entre las tres implementaciones, agrupando error relativo y tasa de convergencia bajo las mismas condiciones experimentales (10 muestras, 7 objetivos, $lr = 5 \cdot 10^{-4}$, $tol = 1{,}5\,mg$).

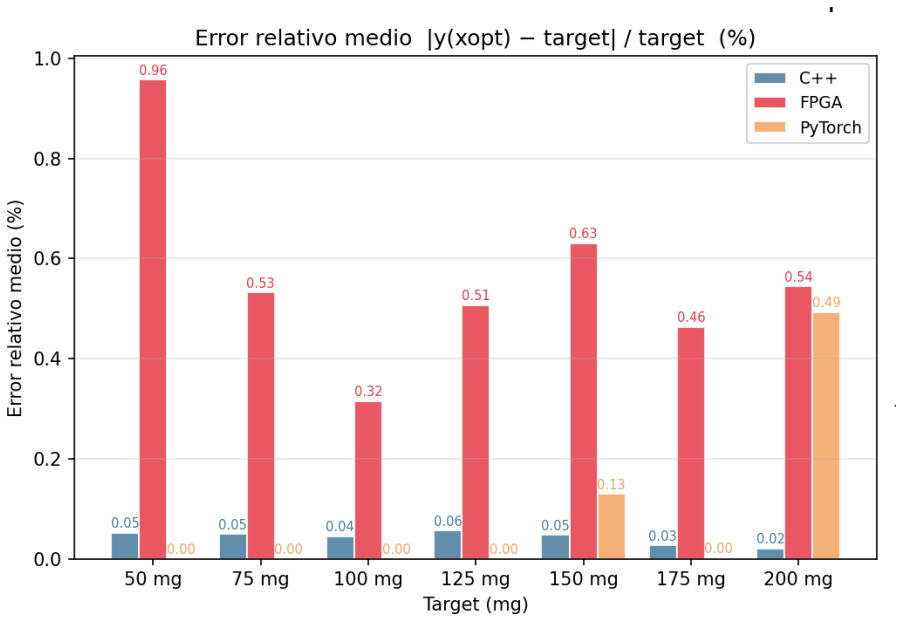

Figura 5.11: Comparativa global de errores relativos entre PyTorch, C++ y FPGA.

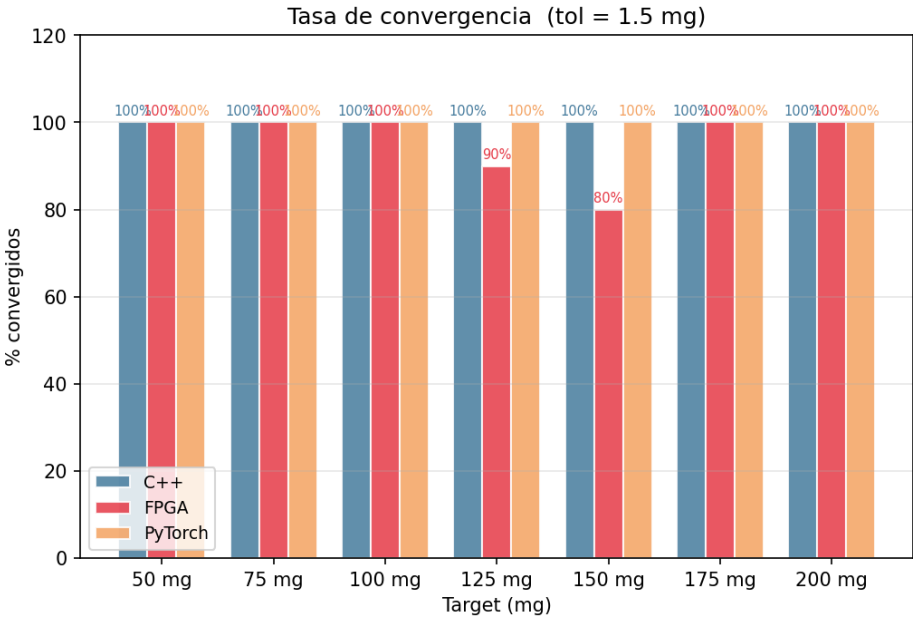

Figura 5.12: Comparativa global de tasas de convergencia entre PyTorch, C++ y FPGA.

Como se puede observar rápidamente, el modelo de C++ es el más competitivo, al ser el más rápido sin perder precisión. Además, C++ es un lenguaje muy portable, con soporte en todo tipo de dispositivos, lo que resulta útil por si se quiere reciclar otro hardware ya adquirido previamente para ejecutar el modelo.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones académicas

Este trabajo ha permitido comprender en profundidad el proceso de traducción de un modelo de aprendizaje automático a una plataforma de hardware acelerado, recorriendo el ciclo completo desde el entrenamiento en PyTorch hasta el despliegue en FPGA. El flujo de desarrollo empleado (basado en *hls4ml* como capa de traducción y Vitis HLS y Vivado para la síntesis e implementación) es el mismo que se utiliza actualmente en física experimental para desplegar redes neuronales en sistemas DAQ y de disparo [2, 23].

El proceso no está exento de dificultades. La traducción requiere conocimientos simultáneos de arquitectura hardware, representación numérica en punto fijo, protocolos AXI y gestión de memoria física, varios de los cuales tuvieron que adquirirse de forma semi-autodidacta durante el desarrollo. Las principales limitaciones encontradas fueron las siguientes:

- La elección de los factores de reutilización es un problema de optimización multiobjetivo no trivial: reducirlos demasiado agota los recursos disponibles y aumentarlos en exceso destruye la ventaja de latencia frente al software.
- La representación en punto fijo [15] introduce errores no uniformes que afectan más al cálculo del gradiente que a la inferencia directa, lo que dificulta la elección de los formatos numéricos óptimos.
- Las herramientas de síntesis e implementación (Vitis y Vivado) requieren del orden de 64 GB de RAM o más para diseños de tamaño mediano, lo que impidió ejecutarlas en local en un ordenador portátil e hizo necesario utilizar la infraestructura de computación científica del grupo de investigación en Física Experimental de Altas Energías en el C^3.

El trabajo ilustra también una de las tensiones fundamentales del despliegue de ML en sistemas embebidos: los modelos más precisos son mucho más complejos de implementar y en general deben comprimirse agresivamente para caber en el hardware disponible [24, 25]. Esta misma tensión aparece en los sistemas de disparo de física de partículas, donde los modelos deben reducirse para operar dentro de los estrictos límites de recursos y latencia.

A pesar de las dificultades mencionadas, se ha logrado la traducción de forma exitosa de un modelo de ML a una FPGA. Se ha podido comparar su rendimiento con modelos de software y el error final medido está por debajo de la tolerancia fijada. Este proceso

sirvió de aprendizaje sobre hardware y sistemas embedidos lo cual es útil a la hora de optimizar código. Ya sea para simulaciones u otro tipo de trabajos en física que requieran de implementar y optimizar software.

6.2. Conclusiones del proyecto industrial

Desde el punto de vista de la aplicación industrial, hay que tener en cuenta las limitaciones físicas de la maquinaria: el proceso de recolocación de las cuchillas de soplado tarda en torno a un minuto. Con este requisito temporal, la conclusión principal es que la aceleración en FPGA no aporta una ventaja suficiente frente a una solución software optimizada: las optimizaciones en PyTorch tardan del orden de $10\,s$, muy por debajo del tiempo de recolocación, por lo que acelerar el cálculo no aporta valor práctico si además se pierde precisión. Las ventajas de ejecución en local (mayor seguridad y resiliencia frente a fallos en sistemas externos) pueden obtenerse igualmente ejecutando el modelo de Python o de C++ en un dispositivo compacto como una Raspberry Pi [26], sin necesidad de hardware reconfigurable.

Cabe recordar que el objetivo de la empresa era una prueba de concepto para evaluar la viabilidad del proceso de traducción. Se concluye que la implementación en FPGA no resulta ventajosa para este caso de uso concreto bajo las condiciones evaluadas, por las siguientes razones:

- Se pierde precisión al pasar a aritmética de punto fijo. Aunque la relevancia en este caso no es muy grande (el modelo pierde un $\sim 0{,}6\,\%$).
- Las ganancias de velocidad no son relevantes para el problema a resolver.
- El modelo equivalente en C++ ha demostrado ser superior en todos los aspectos evaluados.
- La omisión de los BDTs introduce una fuente de error no cuantificable en el modelo de optimización traducido. Esto es debido a que, como se explicó en el capítulo 5, no sigue un camino de optimización realista.

En cuanto al consumo energético, el diseño implementado consume $\sim$0,82 W según el informe de Vivado. Sin embargo, dado que el sistema requiere la CPU con un sistema operativo GNU/Linux [27, 28] activo para ejecutar PYNQ, el ahorro energético real respecto a una solución software es despreciable en este caso. Según la ficha técnica, la Kria puede consumir hasta unos 36W, mientras que una Raspberry Pi hasta unos 15W.

6.3. Trabajo futuro

Como mejoras para una implementación más eficiente, se recomienda cuantizar los pesos durante el entrenamiento (por ejemplo a 8 bits), técnica que ha demostrado producir modelos más robustos a la reducción de precisión. Otra línea de mejora sería sustituir los BDTs por modelos más compactos (BDTs optimizados más agresivamente o de menor profundidad, para que sean traducibles con *conifer* [29], o usar ajustes polinómicos) ya que las dependencias indirectas entre variables de estado y de control no deberían requerir modelos de alta complejidad. Si si quiere explotar la FPGA de forma que aporte una ventaja real, el requisito de latencia debería ser submilisegundo; en ese régimen, la ventaja de la FPGA frente a cualquier solución software es clara e indiscutible.

Glosario

Área de chip

Al implementar un algoritmo en un circuito digital se van instanciando bloques hardware (multiplicadores, registros, LUTs, etc.). Cuantos más bloques se utilicen, mayor espacio ocupará el diseño en el chip. En FPGAs el área se mide en unidades de recursos disponibles: LUTs, flip-flops (FF), DSPs y BRAM.

ASIC (*Application Specific Integrated Circuit*)

Circuito integrado diseñado y fabricado para realizar una única función concreta. Ofrece el máximo rendimiento y la mínima latencia entre los tipos de circuitos integrados, pero su diseño no puede modificarse tras la fabricación, lo que hace su desarrollo costoso y poco flexible.

BRAM (*Block RAM*)

Memoria de acceso aleatorio integrada en la FPGA, organizada en bloques físicos contiguos. Se usa para almacenar datos de alta capacidad (pesos de la red, resultados intermedios) con acceso rápido y determinista. A diferencia de los registros distribuidos, admite puertos de lectura y escritura separados, pero no permite dos accesos al mismo banco en el mismo ciclo; para aumentar el paralelismo es necesario duplicar los bancos, lo que incrementa el uso de área.

Buffer DMA (*Direct Memory Access*)

Región de memoria físicamente contigua en DDR reservada para transferencias de datos directas entre un periférico hardware (en este caso el IP HLS de la FPGA) y la memoria principal, sin intervención de la CPU.

Ciclo de reloj

Los circuitos digitales se sincronizan mediante una señal de reloj que emite pulsos periódicos. Cada pulso delimita un *ciclo de reloj*, la unidad mínima de tiempo del sistema. En cada ciclo se pueden completar un número determinado de operaciones en paralelo, dependiendo de la arquitectura. A 100 MHz el periodo de un ciclo es de 10 ns.

Compilación

Proceso de traducción de código fuente escrito en un lenguaje de alto nivel (C++, Python) a instrucciones máquina ejecutables por una CPU. El compilador analiza el código, lo optimiza y genera un binario que el procesador puede ejecutar directamente. En este trabajo se emplea compilación de C++ con nivel máximo de optimización para el modelo de referencia software que corre en la CPU de la Kria.

CPU (*Central Processing Unit*)

Procesador de propósito general que ejecuta secuencias arbitrarias de instrucciones software. Su flexibilidad lo hace universal, pero implica una eficiencia subóptima para tareas de cómputo intensivo y muy específicas frente a soluciones en hardware dedicado.

DAQ (*Data Acquisition*)

Sistema de adquisición de datos encargado de recoger, filtrar y almacenar las señales producidas por un detector. Por ejemplo, en experimentos de física de partículas, el DAQ debe operar en tiempo real y tomar decisiones de selección de eventos (*trigger*) en tiempos del orden de microsegundos.

DDR (*Double Data Rate*)

Tipo de memoria RAM de acceso aleatorio utilizada como memoria principal en sistemas embebidos como la Kria KV260. En la arquitectura Zynq UltraScale+, la DDR es compartida entre la CPU ARM y la lógica programable de la FPGA: la CPU accede a ella a través del controlador de memoria del procesador, mientras que el IP HLS accede mediante puertos AXI de alta prestaciones (`S_AXI_HP0_FPD`).

Diagrama de caja y bigotes (*boxplot*)

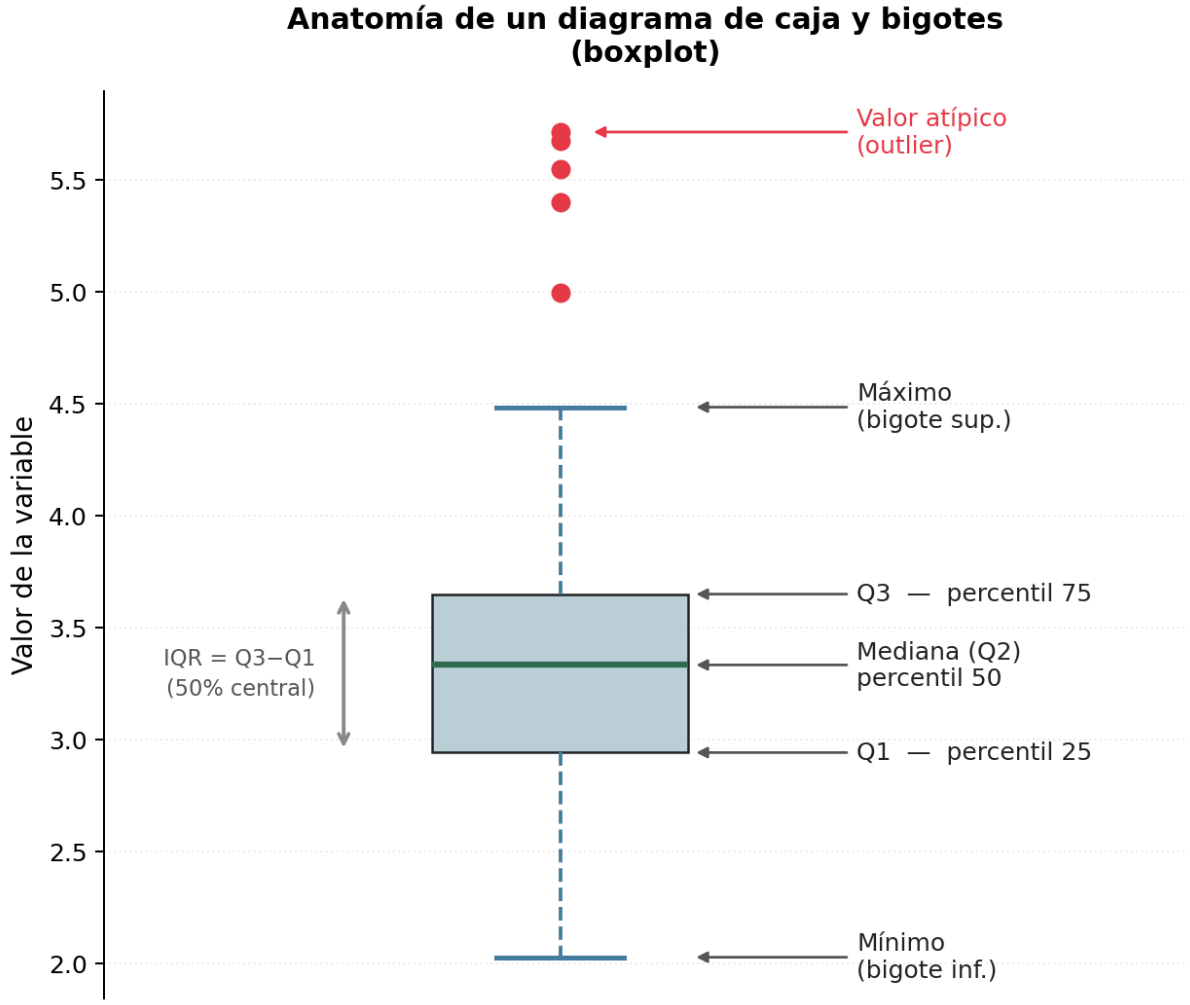

Figura 6.1: Anatomía de un diagrama de caja y bigotes.

El diagrama de caja y bigotes resume la distribución estadística de un conjunto de valores mediante cinco estadísticos robustos (Figura 6.1). La **caja** delimita el rango intercuartílico (IQR $= Q_3 - Q_1$), que contiene el 50 % central de los datos: su borde inferior es el primer cuartil Q_1 (percentil 25) y el superior el tercer cuartil Q_3 (percentil 75). La línea interior es la **mediana** Q_2 (percentil 50), robusta frente a valores extremos. Los **bigotes** llegan hasta el dato más alejado dentro del intervalo $[Q_1 - 1{,}5 \cdot \text{IQR},\ Q_3 + 1{,}5 \cdot \text{IQR}]$; cualquier valor fuera de ese umbral se considera **atípico** (*outlier*) y se representa individualmente. Esta representación permite comparar dispersión, simetría y anomalías entre grupos sin asumir ninguna distribución subyacente.

FPGA (*Field Programmable Gate Array*)

Circuito integrado reconfigurable compuesto por una matriz de bloques lógicos con conexiones programables. A diferencia de los ASICs, puede reprogramarse después de su fabricación para implementar distintos algoritmos en hardware. Combina una latencia

y eficiencia energética próximas a las de un ASIC con la flexibilidad de reutilización, lo que la convierte en la plataforma habitual para sistemas de *trigger* y DAQ en física experimental.

Función sigmoide

Función matemática definida por

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

con imagen en el intervalo $(0, 1)$. Es la base de la función de activación SiLU (véase Cap. 2) y se utiliza en este trabajo como tabla de consulta (*look-up table*) en la FPGA para aproximar la no linealidad de la red neuronal.

GPU (*Graphics Processing Unit*)

Procesador masivamente paralelo compuesto por miles de núcleos simples, originalmente diseñado para renderizado gráfico y actualmente empleado de forma generalizada para el entrenamiento de modelos de aprendizaje automático por su eficiencia en operaciones matriciales.

HLS (*High-Level Synthesis*)

Extensión del lenguaje C++ que permite describir hardware digital de forma más abstracta que con lenguajes nativos como VHDL o Verilog. Una herramienta de síntesis, como Vitis HLS, traduce el código HLS a una descripción hardware sintetizable e implementable en una FPGA.

hls4ml

Librería de código abierto que traduce automáticamente modelos de aprendizaje automático entrenados en *frameworks* como PyTorch o Keras a código HLS sintetizable para FPGA [5, 1]. Genera la implementación en punto fijo del pasada hacia adelante de la red, incluyendo capas lineales, normalización y funciones de activación, y permite configurar parámetros como los factores de reutilización para cada capa. En este trabajo se emplea para traducir el MLP entrenado; la retropropagación y el descenso de gradiente, no soportados por la herramienta, se implementan manualmente.

Implementación

Etapa posterior a la síntesis en la que Vivado traduce la netlist a una configuración física concreta de la FPGA: calcula las rutas exactas por las que circulará la electricidad (*place and route*) y verifica que se cumplen las restricciones temporales (*timing constraints*). El resultado es el fichero binario (`.bit`) que programa la FPGA y el fichero `.hwh` que describe la interfaz a la CPU. Los valores de recursos obtenidos en esta etapa son los definitivos y pueden diferir notablemente de las estimaciones de síntesis.

Loop unrolling (*desenrollado de bucles*)

Técnica de optimización hardware en la que las iteraciones de un bucle se instancian como operaciones hardware independientes en lugar de ejecutarse secuencialmente. Permite explotar el paralelismo a costa de mayor uso de área. En este trabajo controla cuántos multiplicadores se instancian en paralelo para cada capa del MLP; su efecto está parametrizado por el factor de reutilización. (véase Cap. 4).

LUT (*Look-Up Table*)

Tabla de verdad programable que implementa lógica combinacional arbitraria dentro de los bloques lógicos de una FPGA. Para cada combinación de entradas devuelve un valor de salida predefinido. Además de implementar lógica, las LUTs pueden emplearse como memoria distribuida o para aproximar funciones matemáticas complejas (exponenciales, funciones trigonométricas) almacenando valores precalculados, a costa de una pequeña pérdida de precisión.

Paralelismo

Capacidad de ejecutar varias operaciones simultáneamente, con la condición de que no existan dependencias de datos entre ellas. En FPGAs el paralelismo se explota instanciando múltiples unidades de cómputo (multiplicadores, sumadores) que operan en el mismo ciclo de reloj.

Puerto AXI (*Advanced eXtensible Interface*)

Protocolo de comunicación estándar para interconectar bloques hardware en sistemas basados en FPGA, parte del estándar ARM AMBA. En el diseño de este trabajo se emplean dos variantes: **AXI-Lite**, un canal de control sencillo de baja velocidad usado para escribir escalares y direcciones de memoria en los registros del IP (`s_axi_control`); y **AXI Memory-Mapped** (AXI-MM), un canal de datos de alta velocidad que permite al IP acceder directamente a la memoria DDR a través de buffers DMA (`m_axi_gmem`). Las interconexiones *AXI SmartConnect* enrutan estos canales entre la CPU y el IP HLS (véase Sec. 4.3).

PYNQ

Librería de Python desarrollada por AMD que permite controlar el hardware de una FPGA desde la CPU sin necesidad de escribir código de bajo nivel. PYNQ abstrae la carga del fichero `.bit`, la gestión de buffers DMA, y la lectura/escritura de los registros AXI-Lite del IP, permitiendo controlar el kernel HLS desde un script de Python como si fuera una función más. En este trabajo se utiliza para gestionar el ciclo completo de cada optimización: carga de la capa de configuración, escritura de entradas, ejecución y lectura de resultados (véase Sec. 4.4).

Reuse factor

Parámetro de *hls4ml* que controla el grado de paralelismo hardware instanciado para las multiplicaciones de cada capa del MLP. Para una capa con N multiplicaciones, un factor de reutilización r instancia $\lceil N/r \rceil$ multiplicadores hardware que se reutilizan r veces de forma secuencial para completar el cálculo. Los dos casos extremos son:

- $r = 1$: se instancian N multiplicadores en paralelo; máxima velocidad (un ciclo de reloj por capa), máximo consumo de recursos (DSPs, LUTs, BRAM).
- $r = N$: se instancia un único multiplicador que realiza las N operaciones secuencialmente; mínimo consumo de recursos, máxima latencia (N ciclos por capa).

En la práctica, el factor de reutilización se ajusta para equilibrar latencia y uso de recursos, de forma que el diseño quepa en la FPGA disponible sin exceder los límites de DSPs o BRAM.

Síntesis

Traducción de una descripción de hardware en HLS o VHDL a una netlist: una representación abstracta de los bloques lógicos (LUTs, DSPs, BRAM) y sus interconexiones que serán instanciados en la FPGA. La herramienta de síntesis (Vitis HLS en este trabajo) genera también un informe de recursos estimados (uso de LUTs, DSPs, BRAM y latencia). Estas estimaciones tienden a ser optimistas.

VHDL (*Very High Speed Integrated Circuit Hardware Description Language*)

Lenguaje estándar de descripción de hardware digital [30]. A diferencia de los lenguajes de programación convencionales, VHDL no describe una secuencia de instrucciones sino la estructura y el comportamiento de un circuito: los elementos declarados (puertos, señales, registros) existen físicamente en el chip. Es el lenguaje nativo de síntesis para FPGAs, aunque en la práctica suele generarse automáticamente a partir de código HLS.

Bibliografía

[1] Javier Duarte et al. "Fast inference of deep neural networks in FPGAs for particle physics". En: *JINST* 13.07 (2018), P07027. DOI: 10.1088/1748-0221/13/07/P07027. arXiv: 1804.06913 [physics.ins-det].

[2] Allison McCarn Deiana, Nhan Tran, Joshua Agar et al. "Applications and Techniques for Fast Machine Learning in Science". En: *Frontiers in Big Data* 5 (2022), pág. 787421. DOI: 10.3389/fdata.2022.787421.

[3] Manuel J. Rodriguez y DUNE Collaboration. "Fast inference using FPGAs for DUNE data reconstruction". En: *EPJ Web of Conferences*. Vol. 245. 2020, pág. 01030. DOI: 10.1051/epjconf/202024501030.

[4] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". En: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, págs. 8024-8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[5] FastML Team. *fastmachinelearning/hls4ml*. Ver. v1.2.0. 2025. DOI: 10.5281/zenodo.1201549. URL: https://github.com/fastmachinelearning/hls4ml.

[6] AMD Xilinx. *Kria KV260 Vision AI Starter Kit Data Sheet (DS986)*. Advanced Micro Devices, Inc. 2023. URL: https://docs.amd.com/r/en-US/ds986-kv260-starter-kit.

[7] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. "Learning Representations by Back-propagating Errors". En: *Nature*. Vol. 323. Nature Publishing Group, 1986, págs. 533-536.

[8] PyTorch Contributors. *BatchNorm1d — PyTorch documentation*. Accedido: mayo 2026. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html.

[9] PyTorch Contributors. *ReLU — PyTorch documentation*. Accedido: mayo 2026. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html.

[10] PyTorch Contributors. *SiLU — PyTorch documentation*. Accedido: mayo 2026. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.SiLU.html.

[11] Soumallya. *Tree Boosting Methods in Machine Learning*. Medium. Accedido: junio de 2026. 2023. URL: https://medium.com/@soumallya160/tree-boosting-methods-in-machine-learning-89db872db509.

[12] Yoav Freund y Robert E. Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". En: *Journal of Computer and System Sciences* 55.1 (1997), págs. 119-139.

[13] Ian Kuon y Jonathan Rose. "Measuring the Gap Between FPGAs and ASICs". En: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), págs. 203-215.

[14] Enzo Rucci. *Evaluación de rendimiento y eficiencia energética de sistemas heterogéneos para bioinformática*. 1.ª ed. Editorial de la Universidad de La Plata (EDULP), 2018. ISBN: 978-987-4127-54-9. URL: https://libros.unlp.edu.ar/index.php/unlp/catalog/book/871.

[15] A. Jiménez y A. Muñoz. "Very-Large-Scale Integration (VLSI) Implementation and Performance Comparison of Multiplier Topologies for Fixed- and Floating-Point Numbers". En: *Applied Sciences* 15.9 (2025), pág. 4621. DOI: 10.3390/app15094621.

[16] AMD. *Mechanical – Kria KV260 Vision AI Starter Kit Data Sheet (DS986)*. https://docs.amd.com/r/en-US/ds986-kv260-starter-kit/Mechanical. Accessed: 2026-06-13. 2025.

[17] AMD. *Kria KV260 Vision AI Starter Kit*. https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html. Accessed: 2026-06-13. 2026.

[18] AMD. *Vitis High-Level Synthesis User Guide (UG1399)*. Version 2025.2, consultado: 2026-04. AMD. 2025. URL: https://docs.amd.com/r/en-US/ug1399-vitis-hls.

[19] AMD. *Vivado Design Suite User Guide: Designing with IP (UG896)*. Version 2025.2, consultado: 2026-04. AMD. 2025. URL: https://docs.amd.com/r/en-US/ug896-vivado-ip.

[20] ISO/IEC. *Programming Languages — C++*. Inf. téc. 14882:2017. International Organization for Standardization, 2017.

[21] Advanced Micro Devices, Inc. *PYNQ*. Ver. 3.1.2. Repositorio: https://github.com/Xilinx/PYNQ. Sep. de 2025. URL: https://pynq.io.

[22] Murali Ravi et al. "FPGA as a Hardware Accelerator for Computation Intensive Maximum Likelihood Expectation Maximization Medical Image Reconstruction Algorithm". En: (ago. de 2019). Full-text available on ResearchGate. URL: https://www.researchgate.net/publication/334855119_FPGA_as_a_Hardware_Accelerator_for_Computation_Intensive_Maximum_Likelihood_Expectation_Maximization_Medical_Image_Reconstruction_Algorithm.

[23] Jan-Frederik Schulte et al. "hls4ml: A Flexible, Open-Source Platform for Deep Learning Acceleration on Reconfigurable Hardware". En: (dic. de 2025). arXiv: 2512.01463 [cs.AR].

[24] Jennifer Ngadiuba et al. "Compressing deep neural networks on FPGAs to binary and ternary precision with <tt>hls4ml</tt>". En: *Machine Learning: Science and Technology* 2.1 (dic. de 2020), pág. 015001. ISSN: 2632-2153. DOI: 10.1088/2632-2153/aba042. URL: http://dx.doi.org/10.1088/2632-2153/aba042.

[25] Dingyi Dai et al. *Trainable Fixed-Point Quantization for Deep Learning Acceleration on FPGAs*. 2024. arXiv: 2401.17544 [cs.LG]. URL: https://arxiv.org/abs/2401.17544.

[26] Raspberry Pi (Trading) Ltd. *Raspberry Pi 4 Model B Datasheet*. Accedido: mayo de 2026. Raspberry Pi (Trading) Ltd. 2024. URL: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf.

[27] Richard M. Stallman. *Initial GNU Announcement*. Accedido: mayo de 2026. 1983. URL: https://www.gnu.org/gnu/initial-announcement.html.

[28] Linus Torvalds. *What would you like to see most in minix?* Newsgroup comp.os.minix. Mensaje original de anuncio del kernel Linux. 1991. URL: https://www.cs.cmu.edu/~awb/linux.history.html.

[29] S. Summers et al. "Fast inference of Boosted Decision Trees in FPGAs for particle physics". En: *Journal of Instrumentation* 15.05 (mayo de 2020), P05026. DOI: 10.1088/1748-0221/15/05/P05026. URL: https://doi.org/10.1088/1748-0221/15/05/P05026.

[30] IEEE. *IEEE Standard VHDL Language Reference Manual*. Inf. téc. 1076-2008. Institute of Electrical y Electronics Engineers, 2009. DOI: 10.1109/IEEESTD.2009.4772740.