Desarrollo de un Acelerador de Redes Neuronales en FPGA Para la identificación de imagenes

Alexis Horacio López Fragoso

Universidad Autónoma de México,
Facultad de Estudios Superiores Cuautitlán,
Cuautitlán, Estado de México, México
guados123@gmail.com

Abstract. Este trabajo aborda el diseño e implementación de un acelerador de redes neuronales convolucionales sobre la placa de desarrollo *Cora Z7*, empleando *SystemVerilog* para describir la lógica hardware y aprovechar las ventajas de paralelización propias del FPGA [1]. Para el entrenamiento de la red se utilizó la librería *Keras* para el entrenamiento de la red. Además haciendo uso del *Intel Image Classification Dataset* [2], el cual contiene imágenes de diversas clases, como *buildings*, *forest*, *glacier*, *mountains*, *sea* y *street*. Con esta dataset de imagenes se realizará el entrenamiento de la red.
El flujo de diseño incluyó la creación y validación de la red neuronal en el entorno de desarrollo *Python* con Visual Studio, y después la síntesis e implementación de la lógica correspondiente en el FPGA de la *Cora Z7*. Adicionalmente, se utilizó *Vitis* para configurar el *ARM* incorporado en la plataforma, con el propósito de recopilar y visualizar los datos de los resultados de la red neuronal convolucional.

Keywords: FPGA, Redes neuronales convolucionales, Clasificación de imágenes, Intel Image Classification Dataset, Cora Z7, SystemVerilog, Keras, Vitis

1 Introducción

En los últimos años, el uso de redes neuronales convolucionales (CNN, por sus siglas en inglés) ha demostrado ser altamente efectivo para la clasificación de imágenes, superando a muchos de los métodos tradicionales de visión por computadora [3, 4]. Esto se debe a la capacidad de las CNN de extraer características relevantes de manera automática, lo cual permite un mejor desempeño en tareas como la detección y clasificación de objetos [5].

Sin embargo, a pesar de sus grandes ventajas, el entrenamiento y la inferencia de redes neuronales suelen requerir un alto costo computacional. Este hecho motiva la búsqueda de soluciones de hardware especializadas, tales como las FPGAs (Field-Programmable Gate Arrays), que ofrecen una alta capacidad de paralelización y un menor consumo de energía en comparación con CPUs y GPUs

[1]. Por tanto, resulta de gran interés el diseño y desarrollo de aceleradores de redes neuronales en FPGA, con el fin de optimizar la ejecución y mejorar la eficiencia del sistema.

En este proyecto, se presenta el desarrollo de un acelerador de redes neuronales convolucionales en FPGA orientado a la clasificación de imágenes, enfocado en el procesamiento del *Intel Image Classification dataset* [2]. Dicho conjunto de datos contiene más de 25,000 imágenes etiquetadas en seis clases distintas: *buildings*, *forest*, *glacier*, *mountains*, *sea* y *street*. Al estar dividido en diversas categorías de entornos naturales y urbanos, dicho dataset es idóneo para evaluar la versatilidad y eficacia del acelerador propuesto, así como su capacidad de generalización ante escenarios con diferentes características visuales.

El presente trabajo aborda tanto el diseño lógico del acelerador en FPGA como la configuración de la red neuronal convolucional, describiendo la optimización de las arquitecturas utilizadas y los criterios para balancear el rendimiento y los recursos de hardware disponibles. Además, se exponen los resultados de las pruebas realizadas, comparando las métricas de exactitud, velocidad y consumo energético con enfoques alternativos de implementación en dispositivos de propósito general [6].

La solución propuesta tiene como principal aporte la demostración de que las plataformas FPGA permiten la implementación de redes neuronales de manera eficiente, facilitando la adaptación a entornos con restricciones de potencia y espacio, como pueden ser los sistemas empotrados y las aplicaciones de computación en el borde (*edge computing*).

2 Estado del Arte

En la actualidad, la clasificación de imágenes con redes neuronales convolucionales (CNN) se ha convertido en un eje central dentro de las tareas de visión por computadora y ha demostrado resultados sobresalientes en comparación con métodos tradicionales [3]. Sin embargo, el despliegue de las CNN en sistemas con recursos limitados o en aplicaciones que requieren un bajo consumo energético sigue representando un gran desafío. En este contexto, las FPGAs han surgido como una solución viable para llevar a cabo implementaciones eficientes de redes neuronales, combinando un buen rendimiento con flexibilidad y escalabilidad [1].

A continuación, se describen los principales avances relacionados con las redes neuronales convolucionales, su implementación en FPGA y las herramientas de software para la creación y despliegue de dichas redes.

2.1 Redes Neuronales Convolucionales (CNN)

Las redes neuronales convolucionales surgieron como una evolución de los modelos de redes neuronales tradicionales, introduciendo la operación de convolución como mecanismo para extraer características locales de las imágenes [4]. Gracias a esta operación, las CNN pueden identificar patrones en diferentes escalas y profundidades, ofreciendo un alto poder de generalización. Durante la última

década, la aplicación de CNN en clasificación de imágenes ha sido objeto de un intenso desarrollo, destacando además su uso en tareas como detección de objetos y segmentación semántica [5].

2.2 Implementaciones de CNN en FPGA

Las FPGAs ofrecen una arquitectura reconfigurable que permite explotar el paralelismo inherente de las operaciones de convolución, max-pooling y activación que se encuentran en la estructura de las CNN [1]. Esto posibilita la ejecución en paralelo de múltiples filtros y cápas, reduciendo los tiempos de inferencia y, al mismo tiempo, manteniendo un consumo energético bajo en comparación con otras plataformas, como las GPUs [6].

Sin embargo, una de las principales barreras para su uso masivo reside en la complejidad asociada al diseño de la lógica en HDL (*Hardware Description Language*). Para solventar este inconveniente, han aparecido diversas herramientas de diseño de alto nivel (*High-Level Synthesis*, HLS) y flujos de trabajo integrados que buscan automatizar gran parte del proceso de generación del código y optimización de recursos en FPGA.

2.3 Herramientas de Software para el Desarrollo de CNN (Extensión)

El desarrollo de redes neuronales convolucionales se ha visto ampliamente impulsado por la aparición de librerías de software especializadas que permiten a los investigadores y desarrolladores enfocarse en la creación de modelos complejos, sin necesidad de profundizar en los detalles de bajo nivel de las operaciones matemáticas o de la gestión de recursos computacionales. De esta manera, librerías como *TensorFlow*, *PyTorch*, *Caffe* y *MXNet* facilitan la implementación de arquitecturas de CNN, el entrenamiento de modelos en unidades de procesamiento de gráficos (GPU) y la evaluación rápida de su desempeño [3]. Estas herramientas permiten, además, la integración de diferentes optimizaciones, como la reducción de la precisión (p. ej., de 32 bits a 16 o 8 bits), sin que el usuario deba intervenir manualmente en la mayoría de los casos.

La librería *Keras*, que forma parte de *TensorFlow*, ha adquirido gran popularidad debido a su sencillo paradigma de construcción de modelos. Mediante un enfoque basado en objetos y llamadas a funciones de alto nivel, es posible definir una CNN con pocas líneas de código, facilitando la experimentación y el ajuste de hiperparámetros (número de filtros, tamaño de kernel, función de activación, etc.). Asimismo, Keras proporciona herramientas para la carga y el manejo de datasets, así como métodos de entrenamiento que integran la paralelización de la computación en CPU y GPU, según la configuración hardware disponible. En el contexto de este proyecto, el uso de Keras resulta ventajoso para la rápida prototipación y evaluación del modelo sobre el *Intel Image Classification Dataset* [2].

Para la implementación en FPGA, cada vez más fabricantes y proyectos de código abierto ofrecen flujos de trabajo que permiten traducir modelos entrenados con librerías de software de alto nivel a implementaciones optimizadas para hardware reconfigurable. Por ejemplo, se han propuesto enfoques basados en *High-Level Synthesis* (HLS) que toman como entrada las representaciones de la red (en lenguajes como *C/C++* o a través de frameworks como *Vitis AI*), con el fin de generar el código *HDL* correspondiente para la FPGA [1]. Estas herramientas realizan optimizaciones automáticas, como la unrolling de bucles, la paralelización de operaciones y el uso de memorias locales, reduciendo el tiempo de desarrollo y la complejidad de diseño.

Otro aspecto fundamental en la transición desde el entrenamiento hasta la inferencia en FPGA radica en la calibración y cuantización de los parámetros de la red, con el fin de ajustarse a las restricciones de presición y ancho de palabra del hardware [6]. Por ejemplo, pasar de cálculos en punto flotante de 32 bits a 8 bits puede reducir significativamente el consumo de recursos, facilitando la distribución de la computación en múltiples áreas lógicas dentro del dispositivo. Este proceso de reducción de precisión es soportado por diversas librerías y entornos de desarrollo, lo que contribuye a una mejor adaptación del modelo entrenado a las características físicas de la FPGA.

2.4 Conjuntos de Datos para la Clasificación de Imágenes

En el proceso de investigación y validación de modelos de clasificación de imágenes, es esencial contar con conjuntos de datos amplios y diversos. El *Intel Image Classification Dataset* [2] se ha convertido en una alternativa popular debido a su versatilidad y variedad de clases, las cuales abarcan escenas de *buildings*, *forest*, *glacier*, *mountains*, *sea* y *street*. Dicho conjunto de datos resulta particularmente valioso para medir la capacidad de los modelos de CNN de reconocer elementos tanto en entornos urbanos como naturales, permitiendo así una evaluación más completa de la generalización de la red.

En síntesis, el presente trabajo se enmarca en la línea de investigación enfocada en la implementación de redes neuronales convolucionales sobre dispositivos FPGA para lograr sistemas eficientes tanto en rendimiento como en consumo energético. La comparación con soluciones basadas en CPU o GPU, así como el aprovechamiento de herramientas de software y librerías, constituye una de las principales motivaciones para lograr la adopción de estos enfoques en aplicaciones del mundo real. El uso de un dataset variado, como el de Intel, permite demostrar la efectividad de la propuesta en diferentes categorías de clasificación de imágenes.

3 Conocimientos Previos

En esta sección se describen, con mayor detalle, los fundamentos teóricos y matemáticos necesarios para comprender la implementación y el funcionamiento

de las redes neuronales convolucionales (CNN) en el contexto de su despliegue sobre plataformas FPGA. Para ello, se abordan los conceptos clave de la operación de convolución y el mapeo de características, las funciones de activación y su importancia en la no linealidad del sistema, el proceso de entrenamiento a través de la retropropagación, las implicaciones de la cuantización de datos y, finalmente, el pipeline de diseño que unifica todos estos elementos para lograr la implementación en hardware reconfigurable.

3.1 Convolución y Mapeo Espacio-Canal

La operación de convolución es uno de los pilares fundamentales de las CNN, pues permite extraer características locales significativas de una imagen o volumen de características previo [4]. Formalmente, sea $\mathbf{X} \in R^{m \times n}$ una imagen de entrada (o el mapa de características proveniente de una capa anterior), y $\mathbf{W} \in R^{(2k+1)\times(2l+1)}$ el kernel (o filtro) con un sesgo asociado $b \in R$. El valor de la salida en el punto (i, j) se calcula mediante la siguiente expresión:

$$y(i,j) \;=\; \sum_{u=-k}^{k} \sum_{v=-l}^{l} X(i+u,\, j+v)\, W(u,\, v) \;+\; b.$$

En la práctica, se utilizan múltiples filtros para una misma capa convolucional, generando diferentes mapas de características que resaltan atributos diversos de la región local de la imagen. Si la entrada tiene D canales (p. ej. en una imagen RGB, $D = 3$), cada filtro se expande por los mismos D canales, y la salida final en cada posición es la suma de las contribuciones de los canales individuales [3].

Manejo de Bordes y Estride Generalmente, se introducen parámetros de configuración como:

- *Padding* (relleno): se añade un marco de ceros (u otro valor) alrededor de la imagen, a fin de controlar la reducción en el tamaño del mapa de características resultante.
- *Stride* (paso): define cuánto se desplaza el kernel después de cada cálculo de la convolución. Un stride mayor a 1 produce mapas de características más pequeños y acelera la ejecución, pero puede perder información espacial.

Estos parámetros permiten regular el compromiso entre el nivel de detalle espacial capturado y la eficiencia computacional de la red.

3.2 Funciones de Activación

Para aumentar la capacidad de representación de la red, las salidas de las capas convolucionales se someten a funciones de activación no lineales. Sin estas funciones, las CNN se comportarían como simples modelos lineales apilados, limitando su potencial de aprendizaje [3].

ReLU y Variantes Una de las funciones más difundidas es la *Rectified Linear Unit* (ReLU), definida como:

$$\mathrm{ReLU}(z) \;=\; \max(0,\, z).$$

Gracias a su sencillez computacional y a que mitiga el problema de desvanecimiento de gradientes, la ReLU se ha convertido en un estándar en redes profundas. Sin embargo, su principal desventaja es que puede producir neuronas *muertas* (salidas constantes en cero) si el valor de entrada es frecuentemente negativo.

Para contrarrestar lo anterior, se utilizan variantes como la *Leaky ReLU*:

$$\mathrm{LeakyReLU}(z) \;=\; \begin{cases} z & \text{si } z \geq 0, \\ \alpha\, z & \text{si } z < 0, \end{cases}$$

donde α (generalmente pequeño, p. ej. 0.01) evita que la función se anule por completo para valores negativos.

Funciones Sigmoide y Softmax En ciertos contextos, particularmente en la capa de salida para clasificación binaria o multiclase, se utilizan funciones como la sigmoide:

$$\sigma(z) \;=\; \frac{1}{1+e^{-z}}$$

o la *Softmax*:

$$\mathrm{softmax}(\mathbf{z})_i \;=\; \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}},$$

que permiten interpretar las salidas como probabilidades, siendo C el número de clases. Estas funciones aseguran que la suma de las probabilidades en un problema de clasificación sea igual a 1.

3.3 Función de Pérdida y Entrenamiento

El entrenamiento de una CNN consiste en la actualización iterativa de los pesos y sesgos para minimizar una función de pérdida que mida cuán bien predice la red la salida deseada. Entre las distintas funciones de pérdida disponibles, una de las más comúnes para clasificación multiclase es la *entropía cruzada* o *cross-entropy* [3, 5]:

$$\mathcal{L}(\theta) \;=\; -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} t_{i,c}\;\log\big(\hat{y}_{i,c}\big),$$

en la que:

- N es el número de muestras en el conjunto de entrenamiento,
- C es el número de clases,
- $\hat{y}_{i,c}$ es la probabilidad predicha por la red para que la muestra i pertenezca a la clase c,
- $t_{i,c}$ representa la etiqueta real (en un formato *one-hot*).

Retropropagación y Optimización La minimización de la función de pérdida se realiza a través de la retropropagación del gradiente (*backpropagation*), calculando las derivadas parciales con respecto a cada parámetro de la red y ajustándolo en dirección contraria al gradiente. Este proceso está gobernado por algoritmos de optimización como *Stochastic Gradient Descent* (SGD), *Adam* o *RMSProp*, entre otros. El algoritmo *Adam* es especialmente popular dada su capacidad de adaptar la tasa de aprendizaje (*learning rate*) para cada parámetro con base en estadísticas de primer y segundo momento.

Regularización y Tcnicas para Mejorar la Generalización En redes profundas se aplican técnicas como la *regularización L2* (sumatoria de cuadrados de los pesos), *Dropout* (apagado aleatorio de neuronas) o normalización por lotes (*Batch Normalization*) para prevenir el sobreajuste e incrementar la capacidad de generalización. El uso de *Batch Normalization* introduce la normalización de las activaciones intermedias y ayuda a estabilizar y acelerar el entrenamiento.

3.4 Cuantización y Precisión de Datos

En la implementación sobre FPGA, el empleo de representaciones de datos en punto flotante de 32 bits suele ser ineficiente, pues conlleva un alto consumo de recursos lógicos y energéticos. Por tanto, se hace necesaria la cuantización de los pesos y activaciones de la red hacia formatos de menor precisión (por ejemplo, 8 bits o incluso 4 bits) [6].

Proceso de Cuantización La cuantización puede describirse como:

$$\tilde{x} \;=\; \mathrm{round}\Big(\alpha\, x\Big) \times \frac{1}{\alpha},$$

donde α es el factor de escala que define cómo se mapean los valores de punto flotante originales a un rango más reducido. La elección de α y del número de bits disponibles es crítica: un rango muy estrecho puede ocasionar saturación o pérdida de información relevante, mientras que un rango excesivo contrarresta el ahorro de recursos que se pretende lograr.

Estratégias de Cuantización Existen varios métodos para determinar α y los rangos de cuantización:

- *Cuantización Estática:* Durante el entrenamiento, se estiman estadísticas (máximo, mínimo o varianza) para cada tensor de activaciones y para los pesos, estableciendo así un rango fijo previo a la inferencia.
- *Cuantización Dinámica:* Se ajusta el rango de cuantización al vuelo, según los valores que se presenten en cada lote de inferencia, lo cual mejora la precisión a costa de una mayor complejidad en tiempo de ejecución.

– *Cuantización Posterior al Entrenamiento (*Post-Training Quantization*):* Se realiza una vez concluido el entrenamiento, usando un subconjunto de datos para perfilar y ajustar los factores de escala.

La selección del método depende de la aplicación, los recursos de hardware y la complejidad de implementación deseada.

3.5 Uso de SystemVerilog en el Desarrollo de Aceleradores

La descripción de hardware (*Hardware Description Language*, HDL) constituye uno de los pilares para la implementación de sistemas digitales en FPGA. En este contexto, *SystemVerilog* se presenta como un lenguaje de alto nivel que extiende las funcionalidades de Verilog-2005, facilitando tanto la programación de módulos hardware como la verificación de diseños complejos [7]. A diferencia de Verilog, SystemVerilog añade características de programación orientada a objetos, interfaces más robustas para el manejo de buses y protocolos, así como bloques especializados (`always_comb`, `always_ff`) que ayudan a describir de manera más clara la intención lógica del diseño.

Relevancia de SystemVerilog en Aceleradores de CNN En el contexto de los aceleradores de redes neuronales convolucionales, SystemVerilog permite describir la lógica RTL (*Register-Transfer Level*) de una manera modular y escalable, lo que resulta esencial al trabajar con sistemas de gran complejidad y alto paralelismo [1]. Mediante la utilización de módulos, interfaces y paquetes, se pueden encapsular las distintas partes de la red (capas convolucionales, función de activación, buffers de entrada/salida, etc.) de forma que sea más sencillo integrarlas y mantenerlas.

Por otra parte, la compatibilidad de SystemVerilog con herramientas de *High-Level Synthesis* (HLS) y flujos de trabajo como *Vitis* favorece la generación automática de código optimizado a partir de modelos descritos en lenguajes de más alto nivel (*C/C++*) o incluso con librerías de aprendizaje profundo como *Keras*. Una vez sintetizado el diseño, es posible importar los módulos SystemVerilog resultantes a la plataforma *Cora Z7*, integrando además la parte software en el subsistema ARM para configurar y monitorear la inferencia de la red.

4 Metodología

En esta sección se describe el flujo de trabajo seguido para la implementación y validación de la red neuronal convolucional (CNN) tanto en software como en hardware, detallando las herramientas utilizadas y los procedimientos llevados a cabo. Asimismo, se incluyen espacios destinados a la inclusión de imágenes que ilustran los principales resultados y la configuración del entorno de diseño.

4.1 Desarrollo en Python y Entrenamiento de la Red

Para la fase de entrenamiento y validación inicial del modelo, se utilizó el lenguaje *Python* en un entorno de desarrollo común (p. ej. `Jupyter Notebook`). Se emplearon las siguientes librerías:

- `cv2` (*OpenCV*): para el preprocesamiento de imágenes (lectura, redimensionado, transformaciones).
- `os`: para administrar rutas y directorios en el sistema de archivos.
- `matplotlib` y `seaborn`: para la generación de gráficas (curvas de pérdida y precisión, matrices de confusión, etc.).
- `keras` y `sklearn`: para la definición, entrenamiento y evaluación de la red neuronal convolucional.

Carga y Preparación del Dataset Se organizó el *Intel Image Classification Dataset* en diferentes carpetas (una por categoría: *buildings, forest, glacier, mountains, sea, street*) y se aplicó una separación en conjuntos de entrenamiento y validación (por ejemplo, 80% para entrenamiento y 20% para validación). Adicionalmente, se redimensionaron las imágenes a 150×150 pixeles, de modo que coincidieran con la `input_shape` de la CNN. Durante la carga, se normalizaron los valores de los pixeles al rango $[0, 1]$.

Definición de la Arquitectura en Keras El modelo de *Convolutional Neural Network* se definió de la siguiente manera:

```
model = Sequential([
Conv2D(16, (3,3), activation='relu', input_shape=(150, 150, 3)),
MaxPooling2D(2,2),
Conv2D(32, (3,3), activation='relu'),
MaxPooling2D(2,2),
Conv2D(64, (3,3), activation='relu'),
MaxPooling2D(2,2),
Flatten(),
Dense(64, activation='relu'),
Dropout(0.5),
Dense(6, activation='softmax')  # 6 categoras
])
```

Este modelo consta de:

- **Tres capas convolucionales** consecutivas (`Conv2D`), cada una seguida de un `MaxPooling2D` para reducir la dimensionalidad espacial.
- `Flatten`: para convertir la salida tridimensional en un vector unidimensional.
- `Dense(64)` con activación `relu`: capa totalmente conectada que aprende combinaciones de características extraídas.
- `Dropout(0.5)`: que ayuda a regularizar el modelo y reducir el sobreajuste.

– `Dense(6)` con activación `softmax`: capa final para la clasificación en las 6 categorías objetivo.

Se empleó la función de pérdida *categorical crossentropy* y el optimizador *Adam* con sus hiperparámetros por defecto (tasa de aprendizaje inicial 10^{-3}). El *batch size* se estableció en 32 para equilibrar eficiencia de cálculo y estabilidad del entrenamiento.

Entrenamiento y Evaluación El entrenamiento se realizó durante 15 épocas, tras las cuales se obtuvo una precisión (*accuracy*) aproximada del 87% en el conjunto de validación. Una vez finalizado, se guardaron los pesos y sesgos de la red en formato `.mem`, con el fin de utilizarlos posteriormente en la implementación sobre *Vivado*.

4.2 Visualización de Resultados

En esta sección se presentan las métricas obtenidas tras el entrenamiento y validación de la red neuronal convolucional, así como su comparación con la fase de inferencia en hardware. Se incluyen la matriz de confusión y las gráficas de pérdida y precisión, elementos que permiten evaluar la calidad y el comportamiento del modelo durante el proceso de entrenamiento.

Matriz de Confusión La Figura 1 muestra la matriz de confusión resultante tras las 15 épocas de entrenamiento. En dicha matriz, se pueden observar las proporciones de aciertos y errores para cada una de las 6 categorías del *Intel Image Classification Dataset* (*buildings, forest, glacier, mountains, sea, street*). Este análisis es fundamental para identificar clases que pudieran resultar más difíciles de distinguir entre sí y para focalizar esfuerzos de mejora en redes futuras.

Gráfica de Pérdida (*Loss*) En la Figura 2, se presenta la evolución de la pérdida (*loss*) durante el entrenamiento y la validación a lo largo de las 15 épocas. Se observa la disminución progresiva de la función de pérdida, lo cual indica que el modelo se ha ido ajustando de manera adecuada a los datos de entrenamiento. Idealmente, se espera que también haya consistencia entre la curva de entrenamiento y la de validación, reflejando una buena capacidad de generalización.

Gráfica de Precisión (*Accuracy*) La Figura 3 ilustra la evolución de la precisión (*accuracy*) tanto en el conjunto de entrenamiento como en el de validación. Puede apreciarse cómo, con el transcurso de las épocas, el modelo incrementa su capacidad de clasificación, alcanzando finalmente un 87% de exactitud. Al igual que con la pérdida, la brecha entre las curvas de entrenamiento y validación es indicativa del grado de sobreajuste (*overfitting*) o subajuste (*underfitting*) del modelo.

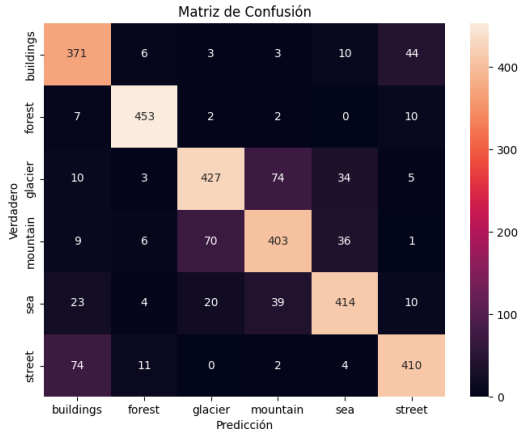

Fig. 1: Matriz de confusión de la CNN entrenada.

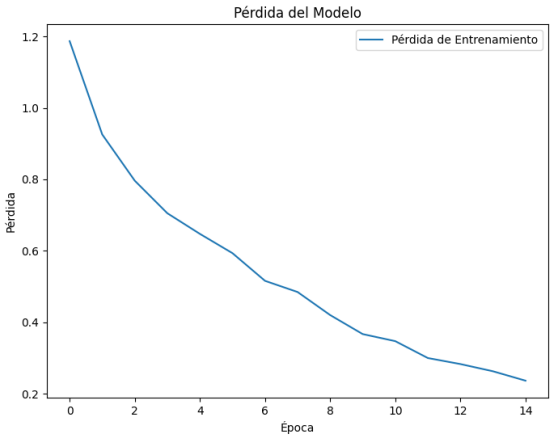

Fig. 2: Evolución de la función de pérdida durante las 15 épocas.

Como se ha observado, la CNN no solo adquiere la capacidad de discriminar entre las diferentes categorías del dataset, sino que también mantiene un rendimiento estable en la validación. Este desempeño se ha logrado respetar durante la implementación en FPGA, validando la coherencia entre los cálculos

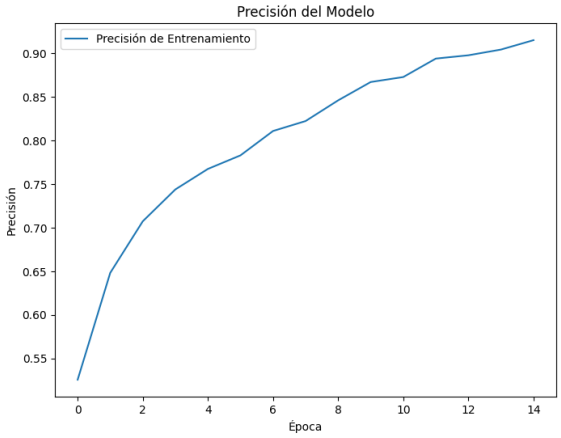

Fig. 3: Precisión en entrenamiento y validación a lo largo de las 15 épocas.

en punto flotante (Python) y la red cuantizada e integrada en la plataforma *Cora Z7*.

4.3 Implementación en Vivado

Descripción del Acelerador en SystemVerilog La etapa de hardware se desarrolló en *Vivado*, empleando *SystemVerilog* para describir los módulos lógicos correspondientes a cada bloque de la red (capas convolucionales, *pooling*, activación, etc.). Para ello, se definieron módulos que:

- **Reciben los pesos y sesgos** precargados desde los ficheros `.mem` para inicializar la lógica de cada capa.
- **Procesan los datos de entrada** (pixeles) mediante la operación de convolución y aplican la función de activación `ReLU`.
- **Gestionan la reducción** espacial a través de *MaxPooling* y la propagación de datos entre capas.

Los resultados parciales se almacenan en memorias internas (*BRAM*) para facilitar el acceso de la siguiente capa y reducir la latencia derivada de accesos a memorias externas.

Creación de la Arquitectura en Vivado La arquitectura final se basó en la familia *Zynq 7000*, optándose por la *Cora Z7* como plataforma FPGA. En *Vivado*, se creó el hardware con los siguientes elementos:

– **Zynq 7000** (SoC): incluye un procesador ARM para la ejecución de código en *C*.
– **Dos instancias de AXI GPIO**: se utilizó para el intercambio de datos entre la lógica de la red neuronal y el subsistema ARM, posibilitando la recepción/envío de pixeles de imágen y la lectura del vector de salida para la clasificación.
– **Módulos RTL en SystemVerilog**: cada capa de la CNN se integró como un bloque independiente, interconectado mediante buses y señales diseñadas para soportar el flujo de datos entre capas.

Tras la configuración de conexiones y la asignación de recursos, se generó el *bitstream* para el FPGA y se exportó el diseño a *Vitis* para la fase de programación del procesador ARM.

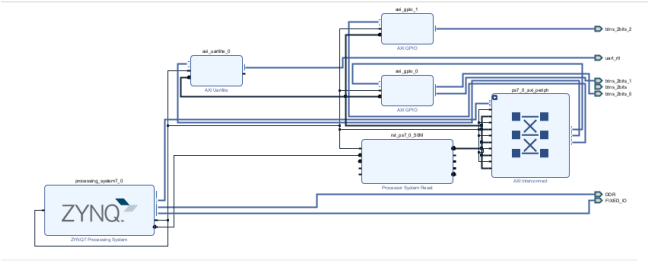

Fig. 4: Captura de la arquitectura final en *Vivado/Vitis*.

4.4 Programación en C y Validación del Sistema Completo

Aplicación en *Vitis* Dentro de *Vitis*, se realizó lo siguiente:

1. **Creación de la Aplicación C:** Se desarrolló el código en lenguaje *C* para inicializar el sistema, configurar la interacción con los bloques AXI GPIO y enviar/recibir datos de inferencia.
2. **Exportación del Hardware:** Se importó el *bitstream* y la información de conexión a *Vitis*, integrando la aplicación software con la plataforma hardware (Zynq + acelerador en SystemVerilog).
3. **Ejecución y Pruebas:** Se cargó la aplicación al procesador ARM del *Zynq 7000*, comprobando la funcionalidad de la CNN embebida y midiendo el rendimiento (*latencia* y *throughput* de clasificación).

Verificación de Resultados En esta fase, se validó que la salida de la red neuronal acelerada en FPGA coincidiera con la salida obtenida en la implementación

en *Python*, garantizando la coherencia de resultados para un subconjunto de imágenes de prueba. Además, se recopilaron datos de recursos utilizados (*LUTs*, *BRAMs*, *DSPs*) y del consumo energético, con el fin de evaluar las ventajas de la aceleración por hardware frente a la ejecución puramente software en CPU o GPU.

Conclusiones Parciales Se observó que la clasificación en FPGA mantiene la precisión alcanzada durante la fase de entrenamiento (cercana al 87%), evidenciando que el traspaso de los pesos y sesgos a la plataforma hardware se realizó satisfactoriamente. Asimismo, se confirmó una reducción de la latencia y un mejor aprovechamiento de recursos en escenarios de inferencia continua, en comparación con la ejecución en CPU de propósito general.

En síntesis, esta metodología combinó la flexibilidad de *Python* para el entrenamiento de la CNN con la eficiencia de la implementación en *SystemVerilog* para la creación del acelerador hardware, demostrando la viabilidad de llevar una red neuronal convolucional a un sistema embebido basado en FPGA y corroborando la coherencia de resultados entre la fase de desarrollo software y el despliegue final en la plataforma *Cora Z7*.

5 Comparación entre la Ejecución en Acelerador y en Software

Una vez finalizada la implementación de la red neuronal convolucional (CNN) tanto en *Python* (entorno de desarrollo y ejecución en CPU o GPU) como en el acelerador basado en *SystemVerilog* (en la *Cora Z7*), resulta fundamental comparar el desempeño y las características de ambos enfoques. A continuación, se destacan los principales criterios de comparación:

5.1 Rendimiento (*Throughput* y Latencia)

- **Ejecución en Python (CPU/GPU)**:
  - *CPU:* ofrece mayor flexibilidad y facilidad de depuración, pero suele experimentar un incremento en la latencia al procesar un número elevado de imágenes.
  - *GPU:* acelera considerablemente la inferencia, en especial si la arquitectura es compatible con paralelización masiva de los kernels. Sin embargo, el consumo energético es más elevado y el hardware suele ser más costoso.
- **Acelerador en FPGA (SystemVerilog en Cora Z7)**:
  - Proporciona baja latencia en la ejecución de operaciones convolucionales, ya que cada capa puede estar muy paralelizada y optimizada.
  - El *throughput* puede superar a la CPU en escenarios de inferencia continua, al no requerir la carga constante de librerías y a la dedicación del hardware al cálculo de CNN.

5.2 Consumo Energético y Uso de Recursos

- **Ejecución en Python (CPU/GPU)**:
  - Las CPU suelen presentar un consumo moderado, pero el tiempo de procesamiento puede ser alto.
  - Las GPU están diseñadas para computación de alta densidad, lo cual se refleja en un consumo energético superior, justificado por la velocidad de inferencia.
- **Acelerador en FPGA:**
  - Las FPGAs permiten un excelente balance rendimiento/consumo, al adaptar la lógica hardware de forma específica para las operaciones involucradas en la CNN.
  - El aprovechamiento de *BRAM*, *DSPs* y *LUTs* puede configurarse minuciosamente para ajustarse a los requisitos de cada capa convolucional y a la precisión (p. ej. 8 bits).

5.3 Facilidad de Desarrollo y Flexibilidad

- **Ejecución en Python (CPU/GPU)**:
  - Desarrollar un modelo de CNN en *Keras*, *TensorFlow* o *PyTorch* es altamente intuitivo, ofreciendo útiles funciones para depuración, análisis y visualización de datos.
  - Cambiar la arquitectura de la red, la función de activación o el número de parámetros es rápido y no requiere re-sintetizar hardware.
- **Acelerador en FPGA (SystemVerilog)**:
  - La implementación RTL exige conocimientos específicos de diseño digital y descripción de hardware, así como un proceso de síntesis y lugar/roteo (*place & route*) que puede consumir más tiempo.
  - La flexibilidad de cambiar parámetros (p.ej. el tamaño del kernel) es menor que en Python, pues requiere regenerar la lógica y volver a programar el FPGA.

5.4 Precisión y Robusteza

- **Ejecución en Python (CPU/GPU)**:
  - Usar punto flotante de 32 bits es el estándar, lo que evita los errores de cuantización e incrementa la precisión.
  - La robustez del entrenamiento y la facilidad para probar variantes de hipeparámetros permite optimizar rápidamente el modelo.
- **Acelerador en FPGA (SystemVerilog)**:
  - A menudo se recurre a formatos de precisión reducida (p. ej. 8 bits), introduciendo errores de cuantización. Sin embargo, si esta cuantización está bien diseñada, la pérdida de exactitud puede ser mínima respecto al modelo original.
  - Resulta clave verificar que la implementación de las funciones de activación y la normalización de datos respete la semántica de la CNN entrenada en punto flotante.

6 Conclusión

A lo largo de este proyecto, se llevó a cabo el diseño e implementación de un acelerador de redes neuronales convolucionales (CNN) sobre una FPGA *Cora Z7*, utilizando *SystemVerilog* para la descripción hardware de las capas y *Python* (con `Keras`, `OpenCV` y `sklearn`) para el preprocesamiento de datos y el entrenamiento del modelo. El *Intel Image Classification Dataset*, con sus seis clases (*buildings*, *forest*, *glacier*, *mountains*, *sea*, *street*), sirvió como base para validar la capacidad de la red en la clasificación de escenas variadas.

En la fase de desarrollo software, la creación y entrenamiento de la CNN demostraron ser rápidos y flexibles, permitiendo un ajuste de hiperparámetros y la visualización de métricas (pérdida y precisión) a lo largo de 15 épocas, logrando alcanzar un 87% de exactitud. Posteriormente, se exportaron los pesos y sesgos en formato `.mem` para integrarlos en la estructura RTL de *SystemVerilog*.

En la fase de implementación hardware, el uso del microprocesador *Zynq 7000* y la plataforma *Cora Z7* posibilitó la incorporación de un subsistema ARM para controlar el flujo de datos hacia el acelerador de CNN, descrito mediante módulos SystemVerilog. El diseño permitió explotar la paralelización de las operaciones convolucionales y la gestión de memorias internas para optimizar la latencia de inferencia. Además, la integración con *Vitis* facilitó la generación del *bitstream*, la configuración del SoC y la programación en *C* para la validación final del sistema.

La comparación entre la solución FPGA y la versión puramente software (Python sobre CPU/GPU) revela un balance interesante: si bien la ejecución en Python otorga gran flexibilidad y simplicidad en la experimentación, la implementación en FPGA ofrece menores latencias y un consumo energético potencialmente más bajo en entornos de inferencia continua, factores cruciales en aplicaciones empotradas y de *edge computing*.

Este trabajo mostró cómo la combinación de librerías de *Deep Learning* en Python para entrenamiento, junto con la descripción de hardware en *SystemVerilog* para la aceleración, habilita la creación de un sistema embebido robusto y eficiente para la clasificación de imágenes. La arquitectura resultante mantiene la precisión obtenida en la fase de entrenamiento, a la vez que ofrece ventajas significativas en términos de latencia y consumo energético, abriendo la puerta a futuras soluciones en diversos entornos de investigación.

References

1. C. Zhang, P. Li, G. Zhang, Y. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.
2. "Intel Image Classification Dataset," https://www.kaggle.com/datasets/puneet6060/intel-image-classification, accedido en: enero de 2025.
3. Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

4. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Communications of the ACM*, vol. 60, no. 6, 2012, pp. 84–90, versión extendida publicada en 2017.
5. R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 580–587.
6. V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
7. "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800-2017)," IEEE Xplore, 2017, https://ieeexplore.ieee.org/document/8299595 (Accedido en: enero de 2025).