UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
PROGRAMA DE MAESTRÍA Y DOCTORADO EN INGENIERÍA
INGENIERÍA ELÉCTRICA – SISTEMAS ELECTRÓNICOS

DISEÑO DE TÉCNICAS DE OPTIMIZACIÓN PARA LA DESCRIPCIÓN DE REDES NEURONALES ARTIFICIALES EN FPGA

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRA EN INGENIERÍA

PRESENTA:
ING. REBECA MUÑOZ MELAMED

TUTORES:
DR. SAÚL DE LA ROSA NIEVES
DR. JESÚS SAVAGE CARMONA
FACULTAD DE INGENIERÍA, UNAM

CDMX, ENERO 2024

JURADO ASIGNADO:

Presidente: Dr. Peña Cabrera Juan Mario

Secretario: Dr. Rodríguez Cuevas Jorge

1er Vocal: Dr. De La Rosa Nieves Saúl

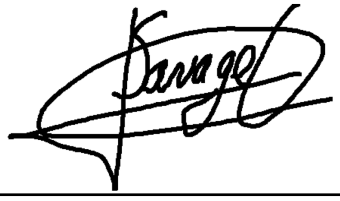
2do Vocal: Dr. Savage Carmona Jesús

3er Vocal: Dr. Negrete Villanueva Marco A.

LUGARES EN DONDE SE REALIZÓ LA TESIS:
Laboratorio de Instrumentación Electrónica de Sistemas Espaciales (LIESE) y Laboratorio de Bio-Robótica, Facultad de Ingeniería, UNAM, México

TUTORES DE TESIS:

DR. SAÚL
DE LA ROSA NIEVES



DR. JESÚS
SAVAGE CARMONA

Si quieres encontrar los secretos del universo,
piensa en términos de energía, frecuencia y vibración.

-Nikola Tesla

Agradecimientos

Primeramente quiero agradecer a mis padres, por todo su esfuerzo y dedicación para siempre brindarme lo mejor, que es lo que me permitió concretar con éxito esta etapa de mi vida. Gracias por su gran paciencia y amor incondicional.

A mi hermano Fausto, quien me vio crecer e influyó en la persona que soy el día de hoy. Gracias por cuidarme, apoyarme y siempre creer en mi.

A Aldair, quien me ha hecho crecer en todos los aspectos de mi vida, motivándome día a día a ser mi mejor versión. Gracias por tu compañía, comprensión, paciencia y apoyo, así como por el gran amor que me demuestras todos los días.

Al Dr. Saúl de la Rosa, por guiarme en cada paso, también por su gran paciencia, comprensión y apoyo para llegar a mi meta. Gracias por ser más que un tutor, por ser un amigo en el que puedo confiar y que se preocupa por mi futuro y bienestar.

Al Dr. Jesús Savage, por su gran guía y paciencia durante todo el proceso, así como por su compresión y apoyo para seguir adelante tanto académica como personalmente.

A todos mis amigos del LIESE, con quienes convivo todos los días y de quienes he aprendido mucho. Gracias por siempre demostrarme su apoyo y buenos deseos.

Índice general

Índice de figuras

Índice de tablas

Resumen

La implementación de redes neuronales artificiales (ANN) en FPGA, ha demostrado ser una alternativa tecnológica con grandes ventajas en cuanto a su procesamiento en paralelo, reconfigurabilidad, consumo energético y costos, comparándolo con otros dispositivos como las GPU o ASIC. A pesar del incremento en la cantidad de recursos disponibles en un solo dispositivo, los elementos lógicos y bloques de memoria limitados en los FPGA, continúan presentando grandes retos para la implementación de ANN. En esta tesis se propone el diseño y aplicación de técnicas de optimización en hardware, con el objetivo de instrumentar una red neuronal feed-forward (FFNN) reconfigurable, en un dispositivo FPGA de recursos limitados. El caso de estudio en este trabajo es comparar el desempeño de la FFNN descrita en FPGA, con una FFNN con la misma topología programada en Python. El desempeño de ambas redes se evaluará mediante la clasificación de imágenes de dígitos escritos a mano provenientes de la base de datos MNIST. El entrenamiento de la red se lleva a cabo empleando una interfaz gráfica de usuario en una CPU de propósito general, que permite al diseñador definir los parámetros de configuración de la red, entrenarla y transmitir los pesos sinápticos y umbrales al FPGA. Dentro de las técnicas de optimización empleadas para describir la FFNN se encuentran: la reutilización de hardware, describiendo únicamente las neuronas de la capa más grande; la implementación de un esquema serial-paralelo, en donde se reciben las entradas de forma serial y en paralelo se multiplican con los pesos de cada neurona; la descripción de un solo bloque de la función de activación en vez de uno por neurona, y la implementación del método de aproximación lineal por partes (PWL), para la descripción eficiente de la función de activación sigmoide. Además, se aprovecharon los recursos embebidos del FPGA, como la memoria y los multiplicadores, los cuales influyeron en el ahorro sustancial de recursos. La FFNN se implementó en el FPGA EP4CE6F17C8 de Altera perteneciente a la familia Cyclone IV, empleando una frecuencia de 50MHz y formatos numéricos de punto fijo de 8 bits Q7 y de 12 bits Q8. Se obtuvo el 89.92 % de precisión de reconocimiento tanto en Python como en el FPGA, siendo el tiempo de clasificación de una imagen en Python de $1.3s$ y en FPGA de $16\mu s$, consumiendo únicamente 36 % de los elementos lógicos.

Capítulo 1

Introducción

Las redes neuronales artificiales (ANN, por sus siglas en inglés) han destacado en incontables disciplinas por su capacidad de procesamiento y gran rendimiento en tareas de clasificación de sistemas tanto lineales como no lineales, lo que las convierte en herramientas ideales para agrupar, aproximar y modelar patrones complejos [1]. Estas características y su adaptabilidad, han colocado a las ANN como uno de los métodos de procesamiento más populares dentro del área científica y de la ingeniería.

A pesar de las abundantes ventajas de las ANN, es importante considerar que su estructura involucra grandes cantidades de cálculos y que corresponde a un modelo de procesamiento inherentemente paralelo, por lo que al ser implementada en un entorno secuencial, es decir, en procesadores de propósito general, se genera un gran consumo de recursos computacionales y tiempos de procesamiento elevados. Una de las propuestas más populares ha sido la implementación de ANN en unidades de procesamiento de gráficos (GPU, por sus siglas en inglés), sin embargo, esto implica una gran demanda en cuanto a dimensiones, costos y consumo energético, haciéndolo ineficiente para aplicaciones que requieren sistemas compactos, autónomos y de bajo costo [2]. Otra alternativa que se ha presentado son los circuitos integrados para aplicaciones específicas (ASIC, por sus siglas en inglés), conocidos por ser dispositivos diseñados y fabricados para un propósito en particular. Debido a que la optimización se enfoca en la aplicación destino, se obtiene mayor rendimiento y menor consumo de energía. Sin embargo, el hecho de ser diseñado de manera tan específica, limita las modificaciones que se deseen realizar en un futuro para adaptar o mejorar el diseño, además de que cuentan con un extenso y costoso ciclo de desarrollo [3], [4]. Por otro lado, la alternativa tecnológica más prometedora que ha surgido, es la implementación de ANN en arreglos de compuertas programables en campo (FPGA, por sus siglas en inglés), los cuales ofrecen un alto rendimiento, gran eficiencia energética y mayor flexibilidad que otros sistemas, lo que los convierte en un recurso extremadamente prometedor para aplicaciones embebidas. Estos dispositivos son circuitos integrados que se conforman por una estructura de hardware configurable, por lo que a diferencia de las GPU y ASIC, los cuales cuentan con una arquitectura fija, los FPGA permiten describir y modificar su arquitectura de acuerdo con la aplicación que se requiere y así obtener el mejor desempeño. Además, los FPGA son dispositivos cuya estructura permite el procesamiento en paralelo, lo que coincide con la naturaleza de un sistema neuronal y explota al máximo su desempeño.

1.1. Justificación: retos en la implementación de ANN en hardware

A pesar de las grandes ventajas que conlleva la instrumentación de redes neuronales en hardware, su implementación en FPGA continúa presentando grandes retos. Aunque la tecnología ha permitido incrementar sustancialmente la densidad de recursos disponibles en un solo FPGA, estos dispositivos cuentan con elementos lógicos finitos, por lo que la gran demanda de procesamiento de las ANN continúa limitando el tamaño de la red que puede ser implementada [5], [6]. De estas limitaciones surge la necesidad de priorizar la optimización de los diseños y definir cuidadosamente la instrumentación de cada elemento de la red, desde la correcta selección del formato y precisión numérica, tipo de multiplicación, hasta el método de descripción para la función de activación. Esto permite obtener el mejor rendimiento y reducir los requerimientos de recursos, sin comprometer demasiado la velocidad de procesamiento, dando paso a la generación de sistemas embebidos viables y de bajo costo.

1.2. Alcances

En este trabajo se propone el diseño y aplicación de técnicas de optimización, que permitan implementar una red neuronal feed-forward (FFNN, por sus siglas en inglés) en dispositivos FPGA de recursos limitados. El alcance de este trabajo es implementar una FFNN previamente programada y probada en Python, la cual clasifica imágenes provenientes de la base de datos MNIST [7], que consisten en dígitos del 0 al 9 escritos a mano. En la Figura 1.1 se muestran algunas de las imágenes empleadas para el entrenamiento y prueba de la FFNN.

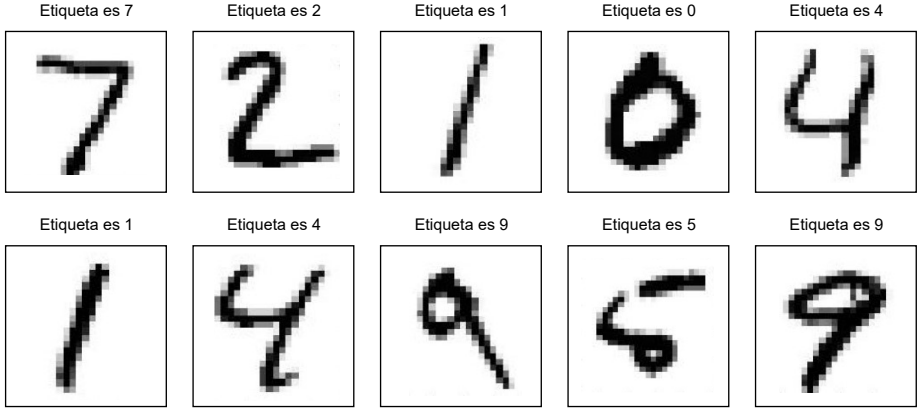


Figura 1.1: Ejemplo de imágenes de números escritos a mano de la base de datos MNIST [7]

1.3. Hipótesis

Mediante la aplicación de técnicas de optimización en hardware, es posible instrumentar una FFNN capaz de clasificar imágenes de dígitos escritos a mano, en dispositivos FPGA de recursos limitados.

1.4. Objetivos

Objetivo general

- Diseñar e implementar técnicas de optimización y reconfiguración para la descripción de una FFNN en FPGA.

Objetivos específicos

- Implementar técnicas de optimización para la instrumentación de FFNN en plataformas FPGA de recursos limitados.
- Realizar la arquitectura de una FFNN reconfigurable que permita acelerar los procesos de selección de parámetros.
- Realizar pruebas de desempeño de la FFNN con imágenes de la base de datos MNIST.
- Comparar el desempeño de la FFNN descrita en hardware con la FFNN realizada en Python.

Esta tesis está conformada por un total de 8 capítulos, en donde el capítulo 1 abarca la introducción, justificación, alcances, hipótesis y objetivos. En el capítulo 2 se estudian los conceptos teóricos fundamentales para el desarrollo del proyecto, mientras que en el capítulo 3 se presenta la información recopilada a partir de la investigación del Estado del Arte. El capítulo 4 y el capítulo 5 se enfocan en el diseño preliminar y el diseño de detalle, respectivamente. En el capítulo 6 se presenta la implementación del diseño en la plataforma FPGA seleccionada, mientras que en el capítulo 7 se muestran los resultados obtenidos. Por último, en el capítulo 8 se abordan las conclusiones y trabajo a futuro.

Capítulo 2

Marco teórico

La gran complejidad y capacidad de procesamiento del cerebro humano, ha despertado constantemente el interés de la comunidad científica por replicar el funcionamiento de los sistemas neuronales biológicos, lo cual inspiró la investigación y el desarrollo de las redes neuronales artificiales. Por ese motivo, es de gran interés para este trabajo conocer y estudiar los conceptos fundamentales de los modelos neuronales biológicos, lo cual se aborda en la primera sección de este capítulo. Más adelante se describe el modelo matemático de una neurona artificial, para posteriormente ahondar en la arquitectura y funcionamiento de las ANN. Debido a que el objetivo principal de este trabajo es probar el desempeño de una FFNN descrita en hardware, al final de este capítulo se aborda la estructura y elementos principales que conforman a los FPGA.

2.1. Neuronas biológicas

El cerebro contiene grandes cantidades de células nerviosas llamadas neuronas, las cuales operan masivamente en paralelo, con el objetivo de procesar y transmitir la información por medio de señales electroquímicas. La estructura general de una neurona consiste en tres secciones principales: una ramificación de entradas, conocidas como dendritas; un cuerpo, también llamado soma, y un árbol de salidas, conformado por el axón y sus terminales. Visualizando su funcionamiento de manera simplificada, es posible considerar a las neuronas como un interruptor, el cual es activado cuando el estímulo recibido por las dendritas alcanza un valor umbral, lo que genera la transmisión del impulso nervioso a través del axón y posteriormente a las neuronas adyacentes. Estas señales son transferidas a través de regiones denominadas sinapsis, que corresponden al espacio en donde las terminales de un axón se anclan, sin realizar contacto alguno, a las dendritas de otra neurona. En la neurona receptora, las múltiples entradas son procesadas y posteriormente compactadas en un solo pulso, si el estímulo de entrada alcanzó el valor umbral, la neurona emitirá la señal. En la Figura 2.1 se ilustra el modelo de una neurona biológica, sus partes fundamentales y conexiones sinápticas con otras neuronas.

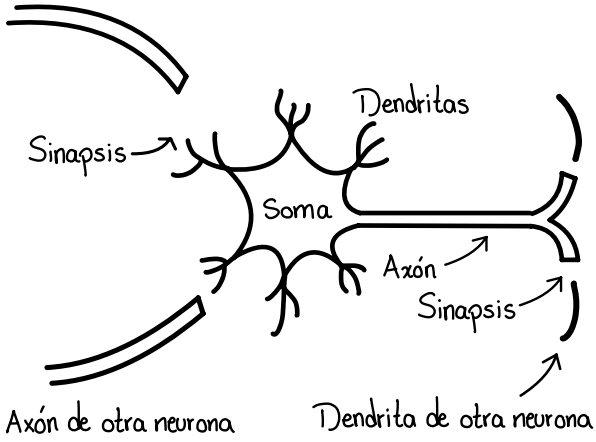


Figura 2.1: Modelo simplificado de una neurona biológica

2.2. Modelo neuronal artificial

A partir del conocimiento que se tenía sobre la estructura y funcionamiento del sistema neuronal biológico, comenzaron a llevarse a cabo intentos para simular su comportamiento en sistemas computacionales, que tuvieran la capacidad de resolver problemas y realizar tareas que son intuitivas para el ser humano. Sin embargo, debido a la gran complejidad de las estructuras biológicas, construir un modelo extremadamente similar conlleva el uso de grandes recursos computacionales. Partiendo de estas ideas, en 1943 los investigadores McCulloch y Pitts [8] presentaron el primer modelo neuronal y el más empleado en la actualidad, en donde introdujeron paradigmas que recrean el comportamiento de las neuronas como interruptores, demostrando que con redes sencillas es posible calcular una gran cantidad de funciones lógicas y aritméticas. En la Figura 2.2 se ilustra el diagrama de bloques de dicho modelo, el cual se explica con detalle a continuación.

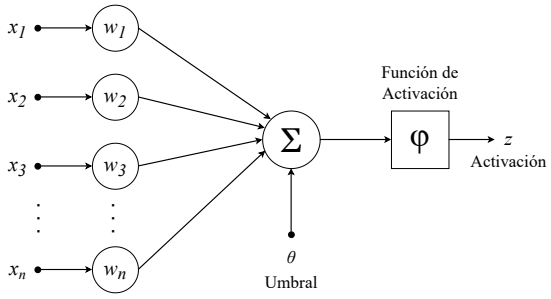


Figura 2.2: Diagrama de bloques de una neurona artificial [9]

El modelo consiste en realizar una combinación lineal de los elementos x_i del vector de entradas, representado como $\mathbf{x}$. La combinación lineal se define como u en la ecuación (2.1) y posteriormente en (2.2) se le aplica a u una transformación no lineal, mediante una función denominada *función de activación*.

$$u = \sum_{i=1}^{n} \omega_i x_i - \theta \tag{2.1}$$

$$z = f(u) \tag{2.2}$$

Los elementos del vector de entrada $\mathbf{x}$ provienen, ya sea de las entradas del sistema o de las salidas de otras neuronas, lo que equivale a los estímulos incidentes en las dendritas de las neuronas biológicas. Los parámetros ω_i son denominados pesos sinápticos, debido a que representan la fortaleza de los enlaces entre las neuronas. Así como la fortaleza de las conexiones biológicas se encuentra directamente relacionada con el aprendizaje y la experiencia, los pesos ω_i representan el conocimiento obtenido durante el entrenamiento de la red, de lo cual se hablará más adelante. Otro parámetro de gran importancia en el modelo neuronal es θ, el cual es conocido como el *umbral* del modelo. Durante la primera fase, mostrada en la expresión (2.1), los pesos sinápticos multiplican a cada uno de los elementos del vector de entrada, lo que genera la activación de la neurona siempre y cuando la suma de productos haya sido mayor que el umbral θ. Usualmente el umbral se incorpora a la suma ponderada como un peso con valor $\omega_0 = -\theta$, multiplicado por una entrada $x_0 = 1$, lo cual se representa en la ecuación (2.3).

$$u = \sum_{i=0}^{n} \omega_i x_i \tag{2.3}$$

Dentro de los primeros problemas de clasificación resueltos mediante neuronas artificiales, se encuentra la generación de funciones booleanas, tal como las compuertas lógicas *or* y *and*. Para ello, es necesario seleccionar los pesos sinápticos y el umbral adecuados, con el objetivo de obtener una neurona que cumpla con el comportamiento de estas operaciones. En la Figura 2.3 se presenta la tabla de verdad de una función lógica *or* que cuenta con dos entradas x_1 y x_2, en donde aquellos valores que provoquen S=1 son asignados a la región A y en caso contrario son asignados a la región B.

X_2	X_1	S	Región
0	0	0	B
0	1	1	A
1	0	1	A
1	1	1	A

Figura 2.3: Tabla de verdad de una función lógica *or*

Si se representan los valores de las entradas como coordenadas, siendo x_1 el eje de las abscisas y x_2 el eje de las ordenadas, se obtiene el plano cartesiano de la Figura 2.4, de tal forma que en la región A se encuentran las coordenadas (0,1),(1,0) y (1,1) y en la región B únicamente la coordenada (0,0), lo cual cumple con el planteamiento de la tabla de verdad

de la Figura 2.3. Por lo tanto, es necesario definir los pesos y el umbral adecuados, tal que cuando las entradas tomen los valores de la región A, la salida de la neurona sea igual a uno.

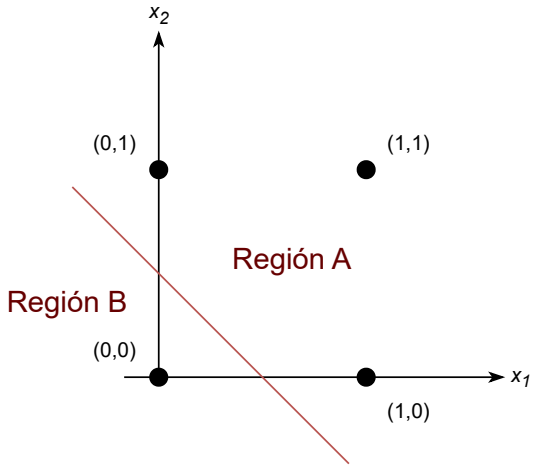


Figura 2.4: Clasificación de coordenadas correspondientes a la función *or* [9]

En la Figura 2.5 se ilustra una neurona artificial en donde se definen sus pesos sinápticos como $\omega_1 = \omega_2 = 1$ y $\theta = -\frac{1}{2}$, además de que cuenta con una función de activación de tipo escalón (2.4), la cual valdrá uno cuando la suma ponderada u sea mayor a cero. Esta selección de parámetros permite clasificar las coordenadas tal como se indica en la tabla de verdad y en el plano cartesiano, generando así el comportamiento de una función lógica *or*.

$$f(u) = \begin{cases} 1 \ u > 0 \\ 0 \ u \leq 0 \end{cases} \tag{2.4}$$

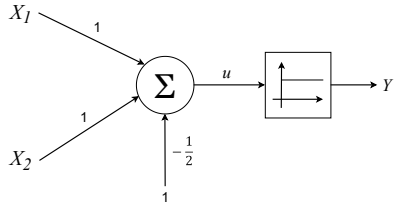


Figura 2.5: Neurona artificial de la función lógica *or* [9]

Para comprender cómo la neurona es capaz de clasificar las coordenadas en sus respectivas regiones, es ilustrativo analizar el resultado de la suma ponderada sustituyendo en cada caso los valores de los pesos, el umbral y las entradas en la ecuación (2.5).

$$u = x_1 \cdot \omega_i + x_2 \cdot \omega_2 - \theta \tag{2.5}$$

Los cuatro casos que pueden presentarse como entradas de la neurona son los siguientes:

Primer caso: $x_1 = 0, x_2 = 0$

$$u = (0)(1) + (0)(1) - \frac{1}{2} = -\frac{1}{2}$$

$$f(u) = f(-\frac{1}{2}) = 0$$

Segundo caso: $x_1 = 1, x_2 = 0$

$$u = (1)(1) + (0)(1) - \frac{1}{2} = \frac{1}{2}$$

$$f(u) = f(\frac{1}{2}) = 1$$

Tercer caso: $x_1 = 0, x_2 = 1$

$$u = (0)(1) + (1)(1) - \frac{1}{2} = \frac{1}{2}$$

$$f(u) = f(\frac{1}{2}) = 1$$

Cuarto caso: $x_1 = 1, x_2 = 1$

$$u = (1)(1) + (1)(1) - \frac{1}{2} = \frac{3}{2}$$

Se observa que en el segundo, tercer y cuarto caso u es mayor que cero, por lo que la función de activación será igual a uno para estas entradas, cumpliendo así con el comportamiento de una función lógica *or*. De este ejemplo es posible deducir que la selección de pesos y umbrales es variada, ya que redefinir los pesos como $\omega_1 = \omega_2 = 0.6$ o cambiar el umbral por $\frac{1}{3}$, también generaría el resultado deseado, por lo que existen múltiples valores de estos parámetros los cuales al seleccionarse cumplen con el comportamiento planteado. En este caso los valores fueron seleccionados de forma arbitraria, sin embargo, cuando se cuenta con funciones de mayor complejidad, lo cual involucra grandes cantidades de pesos y umbrales, la asignación de estos parámetros es realizada mediante métodos y algoritmos definidos, lo cual se explica más adelante en la sección 2.3.2. Esta fase de diseño es denominada *entrenamiento*, en donde el objetivo es encontrar los pesos y umbrales que permitan obtener la relación entre las entradas y salidas deseada. Para la función *or* se llevó a cabo una clasificación lineal, es decir, se empleó una sola neurona que realiza una combinación lineal. Sin embargo, existen datos de entrada que no presentan un comportamiento lineal, por lo que es pertinente la adición de neuronas. De esta necesidad surge el concepto de red neuronal artificial, en donde a partir de la organización de neuronas en capas, es posible modelar patrones de naturaleza no lineal.

2.2.1. Función de activación

Una vez realizada la combinación lineal de las entradas, la función de activación determina si se transmitirá el impulso, lo cual impacta en el comportamiento de las neuronas y la forma en la que cada una responde a las señales de entrada. En la Tabla 2.1 se muestran algunas de las funciones de activación más empleadas dentro del modelo neuronal.

Tabla 2.1: Tipos de funciones de activación

Función de activación	Fórmula
Sigmoide	$\sigma(u) = \frac{1}{1+e^{-u}}$
Tangente hiperbólica	$tanh(u) = \frac{e^{u}-e^{-u}}{e^{u}+e^{-u}}$
Escalón	$h(u) = \begin{cases} 1 \ u > 0 \\ 0 \ u \leq 0 \end{cases}$
Lineal	$f(u) = au + b$

Una de las funciones de activación que ha sido mayormente empleada es la *función sigmoide*, ilustrada en la Figura 2.6, debido a que cuenta con múltiples características que resultan ser útiles para el entrenamiento y prueba de las ANN. Una de las principales virtudes de la función sigmoide, es que su salida puede ser interpretada como una probabilidad que se le asigna a cada región, lo que fundamenta la pertenencia de un conjunto de datos, a una región en particular. Además, el cálculo de derivadas es esencial en el entrenamiento de redes neuronales, sobre todo cuando se emplean algoritmos basados en gradientes, siendo la función sigmoide ideal para estas ocasiones. Esto es debido a que la derivada de la función sigmoide se puede obtener fácilmente usando su propia salida, tal como se observa en la expresión (2.6).

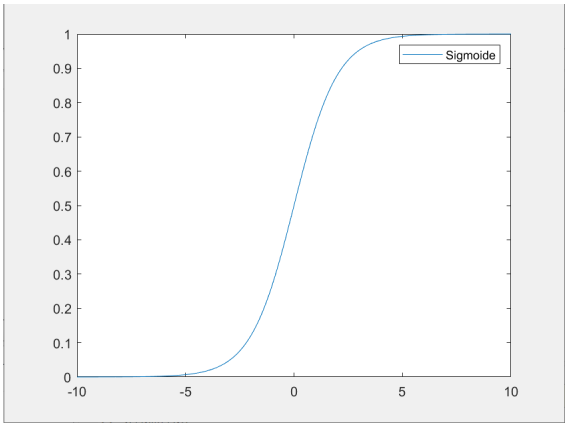


Figura 2.6: Función de activación sigmoide

$$\sigma'(u) = \frac{e^{-u}}{(1+e^{-u})^2} = \sigma(u)(1-\sigma(u)) \tag{2.6}$$

Con el propósito de resumir la analogía entre neuronas artificiales y biológicas, en la Tabla 2.2 se presenta una comparación de los elementos esenciales de ambas neuronas.

Tabla 2.2: Comparación entre una neurona biológica y una neurona artificial

Elemento	Neurona	
	Biológica	Artificial
Entradas	Señales de dendritas	Vector de entrada
Valores ajustables de entrada	Fortaleza de conexiones	Pesos sinápticos
Acumulación de entradas	Acumulación en el cono axónico	Suma ponderada
Modalidad de activación	Valor umbral	Función de activación
Salida	Un impulso nervioso	Valor escalar

2.3. Redes neuronales artificiales (ANN)

Formalmente, una ANN se define como un paradigma matemático y computacional enfocado al reconocimiento de patrones y clasificación de datos, que se conforma por unidades básicas denominadas neuronas, las cuales se encuentran estructuradas en capas y conectadas entre sí, permitiendo gran diversidad de topologías con propósitos y cualidades distintas. Para comprender de mejor manera cómo el estructurar a las neuronas en capas, nos permite reconocer patrones más complejos, en esta sección se continuará analizando el comportamiento de las neuronas para el cálculo de operaciones lógicas.

En el ejemplo de la operación lógica *or* se observó que una neurona por sí sola, permite clasificar un conjunto de datos en dos regiones mediante una recta, sin embargo, existen funciones de mayor complejidad que requieren de neuronas adicionales para agrupar los datos en sus respectivas regiones. Como ejemplo de esto se encuentra la función lógica *xor*, que a diferencia de las operaciones *and* y *or*, presenta un comportamiento que es imposible de generar con solo una neurona, por lo tanto no existe una recta que por sí sola agrupe los datos en la región que les corresponde. Similar a la sección pasada, en la Figura 2.7 se presenta la tabla de verdad de una función *xor*, en donde el resultado será igual a 1 siempre y cuando las entradas tengan valores distintos, es decir, cuando x_1 sea diferente de x_2.

X_2	X_1	S	Región
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B

Figura 2.7: Tabla de verdad de una función lógica *xor*

Aplicando el mismo criterio del ejemplo anterior, las combinaciones de entradas en donde S=1 son asignadas a la región A y en caso contrario pertenecen a la región B. Si representamos los valores de las entradas en un plano cartesiano y las asignamos a su región correspondiente, se obtiene la gráfica de la Figura 2.8.

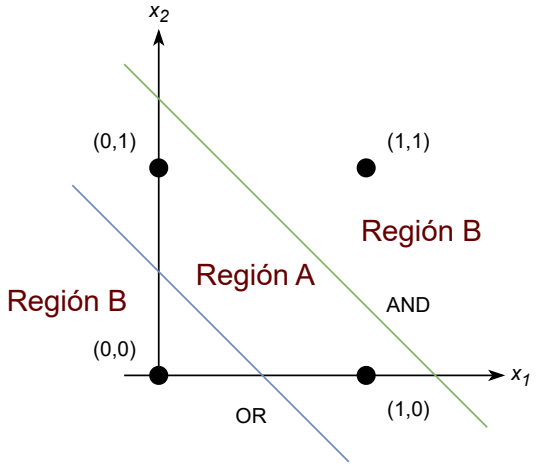


Figura 2.8: Clasificación de coordenadas correspondientes a la función *xor* [9]

Como puede observarse, no es posible separar los datos en las regiones A y B con una sola recta, en su lugar, es necesario emplear dos de ellas para conseguir la clasificación que cumpla con el criterio de la tabla de verdad. Las rectas azul y verde corresponden a las neuronas *or* y *and*, respectivamente. En conjunto, estas neuronas forman una red sencilla capaz de generar el comportamiento de la función lógica *xor*, tal como se muestra en la Figura 2.9.

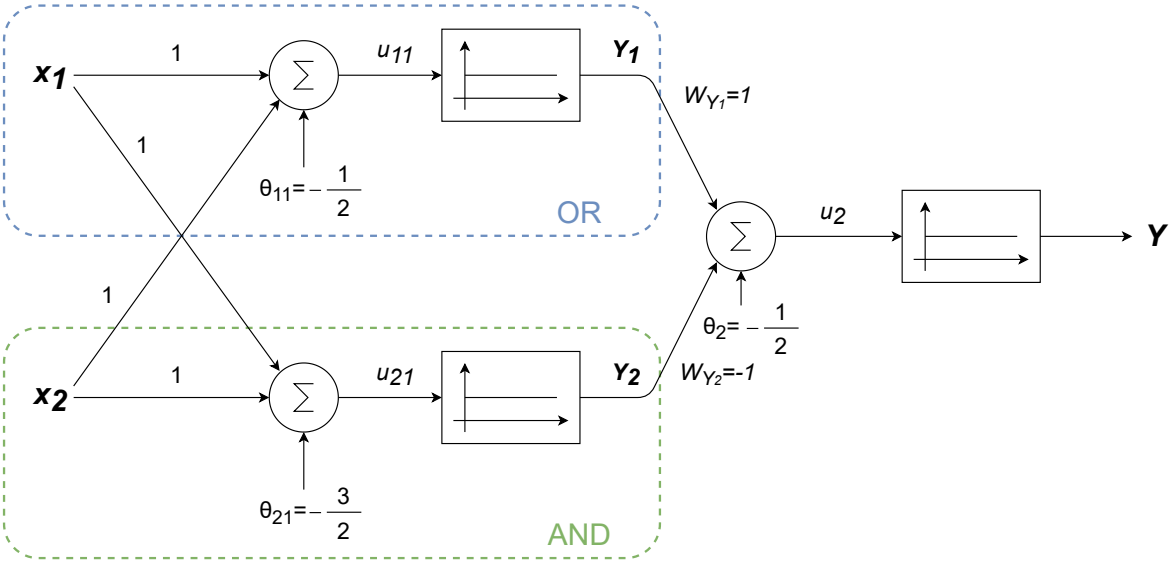


Figura 2.9: Red neuronal artificial para la función lógica *xor* [9]

En este ejemplo se introduce la arquitectura de una ANN sencilla, la cual cuenta con neuronas organizadas en tres capas: la capa de entrada, que corresponde a los datos de entrada x_1 y x_2; la capa intermedia, que cuenta con las neuronas *or* y *and*, y la capa de salida, la cual genera el resultado final Y de la red. El parámetro Y_1 es la respuesta de la neurona *or* y Y_2 la de la neurona *and*, que a su vez son entradas de la última neurona, la cual cuenta con los pesos $\omega_{Y1} = 1$ y $\omega_{Y2} = -1$, mientras que su umbral es $\theta_2 = -\frac{1}{2}$. La suma ponderada de la última neurona, expresada en la ecuación (2.7), toma en cuenta dichos parámetros y se encuentra en función de Y_1 y Y_2.

$$u_2 = Y_1 \cdot \omega_{Y1} + Y_2 \cdot \omega_{Y2} - \theta_2 \tag{2.7}$$

Al igual que se realizó anteriormente, es ilustrativo analizar los cuatro casos que pueden presentarse a la entrada de la red. De forma anticipada se saben los valores de Y_1 y Y_2 para cada caso, ya que corresponden a las respuestas de las funciones *or* y *and*, respectivamente.

Primer caso: $x_1 = 0, x_2 = 0$

$$Y_1 = 0, Y_2 = 0$$

$$u = (0)(1) + (0)(-1) - \frac{1}{2} = -\frac{1}{2}$$

$$f(u) = f(-\frac{1}{2}) = 0$$

Segundo caso: $x_1 = 1, x_2 = 0$

$$Y_1 = 1, Y_2 = 0$$

$$u = (1)(1) + (0)(-1) - \frac{1}{2} = \frac{1}{2}$$

$$f(u) = f(\frac{1}{2}) = 1$$

Tercer caso: $x_1 = 0, x_2 = 1$

$$Y_1 = 1, Y_2 = 0$$

$$u = (1)(1) + (0)(-1) - \frac{1}{2} = \frac{1}{2}$$

$$f(u) = f(\frac{1}{2}) = 1$$

Cuarto caso: $x_1 = 1, x_2 = 1$

$$Y_1 = 1, Y_2 = 1$$

$$u = (1)(1) + (1)(-1) - \frac{1}{2} = -\frac{1}{2}$$

$$f(u) = f(-\frac{1}{2}) = 0$$

Únicamente en el segundo y tercer caso la función de activación vale 1, lo cual cumple con el comportamiento planteado de la función lógica *xor*. Ampliando este concepto es posible concluir que conforme nos enfrentamos a funciones cada vez más complejas, surge la necesidad de adicionar y organizar neuronas capaces de generalizar y asociar la información proporcionada.

2.3.1. Perceptrón multicapa (MLP)

El modelo neuronal presentado en [8] permitió a Frank Rosenblatt desarrollar la primera generación de redes neuronales: el perceptrón multicapa (MLP, por sus siglas en inglés). El MLP está conformado por las neuronas de McCulloch y Pitts, las cuales se interconectan, se organizan en capas y cada una cuenta con una función de activación no lineal, usualmente la función sigmoide o tangente hiperbólica. La estructura básica de un MLP, mostrada en la Figura 2.10, está conformada por lo siguiente: una capa de entrada, la cual representa los datos que entran a la red; un número variable de capas ocultas, y una capa de salida, que constituye la respuesta final de la ANN.

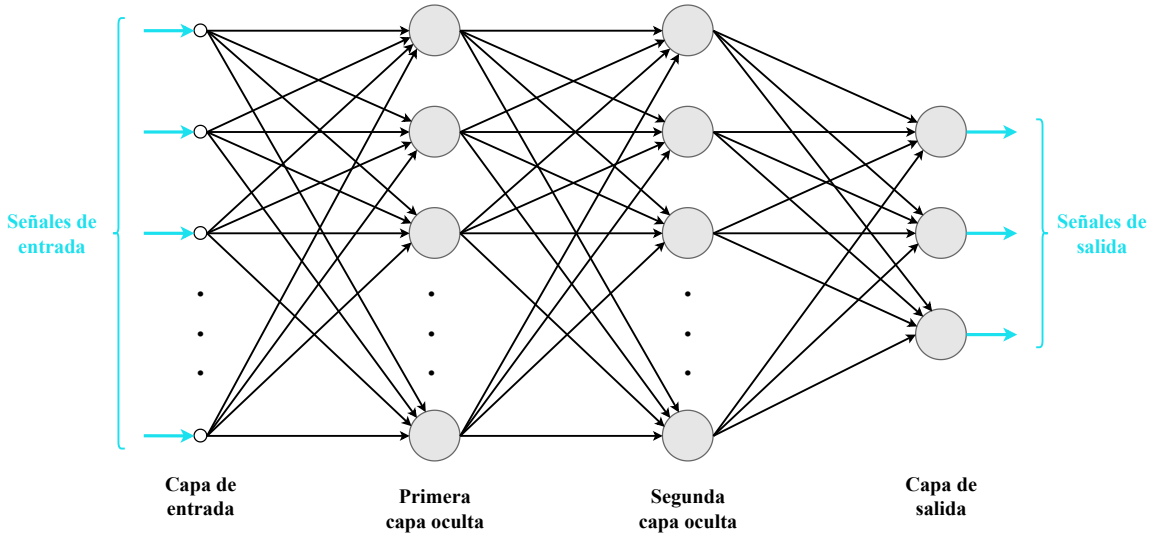


Figura 2.10: Ejemplo de un MLP con dos capas ocultas

Las flechas que se dirigen o salen de cada neurona son conocidas como enlaces sinápticos, cada una con un peso asociado e indican el flujo de la señal a través de la red. Este tipo de arquitecturas, en donde todas las neuronas de una capa se conectan con cada una de las neuronas de la siguiente capa, se conoce también como red neuronal feed-forward (FFNN). Anteriormente se vio que en una neurona se realiza la combinación lineal de las variables de entrada, la cual es posteriormente transformada por una función de activación no lineal. Para una FFNN, la suma de productos de una neurona j en la capa L se expresa en la ecuación (2.8).

$$u_j^L = \sum_{i=0}^{n} \omega_{ij}^L z_i^{L-1} \tag{2.8}$$

En donde w_{ij}^L representa el peso de la conexión entre la neurona i de la capa L-1 y la neurona j de la capa L. La salida de la neurona j, expresada como z_j^L en la ecuación (2.9), se obtiene transformando u_j^L mediante una función de activación.

$$z_j^L = f(u_j^L) \tag{2.9}$$

2.3.2. Métodos de aprendizaje

Tal como se explicó en la sección anterior, el objetivo del entrenamiento es encontrar los pesos y umbrales, que permitan obtener la relación entre los datos de entrada y salida que se desea. El aprendizaje de la ANN se basa en procesos iterativos que emplean muestras de entrenamiento, lo que permite a la red ajustar sus pesos y umbrales progresivamente. Esto brinda a las redes neuronales la capacidad de aproximar funciones desconocidas que dependen de una gran cantidad de entradas. Esta característica es aprovechada en tareas en donde no es posible derivar restricciones lógicas de forma explícita.

Existen múltiples criterios que permiten llevar a cabo el entrenamiento de una red neuronal, los cuales involucran la modificación sucesiva de sus parámetros. En este trabajo únicamente se estudiará el método del descenso del gradiente, el cual se ha empleado con mayor frecuencia para el entrenamiento de redes neuronales.

2.3.3. Método del descenso del gradiente

Este método es por mucho el algoritmo más popular dirigido al entrenamiento de ANN y se basa en el ajuste del vector de pesos sinápticos y umbrales, mediante la minimización de una función de error. Cabe recordar que se entiende por entrenamiento a la etapa en donde el algoritmo aprende de una serie de vectores de entrada, denominado conjunto de entrenamiento, lo que le permite al sistema ajustarse para obtener los resultados esperados. Para una ANN con M entradas, se requiere que la red aprenda a generar una salida de acuerdo con las muestras de entrenamiento siguientes:

$$\left\{(\vec{x_k}, \vec{d_k}); 1 \leq k \leq K\right\}$$

$$\vec{x_k} = [1 \quad x_{1k} \quad x_{2k} \quad \cdots \quad x_{Mk}]$$

En donde el vector $\vec{x_k}$ contiene las M entradas de la muestra de entrenamiento k, el vector $\vec{d_k}$ representa las salidas deseadas y K es el número total de muestras de entrenamiento [9]. La salida de una neurona está en función tanto del vector de pesos sinápticos $\vec{\omega}$, como de las entradas $\vec{x_k}$, por lo que para una muestra de entrenamiento k, la salida de una neurona se representa por la ecuación (2.10).

$$z_k = f(u_k) = f(\vec{\omega} \cdot \vec{x_k}) \tag{2.10}$$

Como se mencionó anteriormente, durante el aprendizaje de la ANN es necesario partir de una función de error, la cual corresponde al error medio cuadrático. El error generado por la neurona j en la capa L, cuando se ingresa la muestra de entrenamiento k, se expresa en la ecuación (2.11).

$$E_j^L = \sum_{k=1}^{K} (d_{jk}^L - z_{jk}^L)^2 \tag{2.11}$$

El objetivo es encontrar todos los pesos sinápticos w_{ij}^L, los cuales minimicen el error cuadrático medio total. Dicha minimización es realizada mediante algoritmos iterativos, en donde comúnmente se comienza en un punto aleatorio y gradualmente se alcanza el mínimo local o global, lo cual se ilustra en la Figura 2.11.

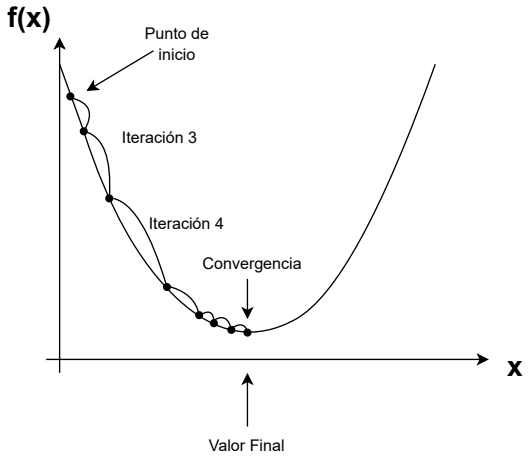


Figura 2.11: Ilustración del descenso del gradiente

El movimiento se realiza en la dirección de descenso más pronunciada, es decir, el gradiente negativo. Se considera que se ha logrado la convergencia, cuando no se producen más cambios significativos en el vector de pesos, lo cual sucede cuando la función de error es cercana o igual a cero.

2.4. Estructura de los FPGA

Los arreglos de compuertas programables en campo (FPGA), son circuitos integrados digitales conformados por bloques lógicos programables, los cuales se enlazan mediante interconexiones configurables. El término *programable en campo*, hace alusión a que incluso después de ser fabricados y distribuidos, el diseñador puede configurar el dispositivo para la aplicación requerida [10].

Los elementos principales de un FPGA son:

- Una matriz de bloques lógicos configurables (CLB, por sus siglas en inglés).
- Una red de interconexiones programables.
- Bloques de entrada/salida que rodean a los CLB.

La distribución de estos elementos se ilustra en la Figura 2.12.

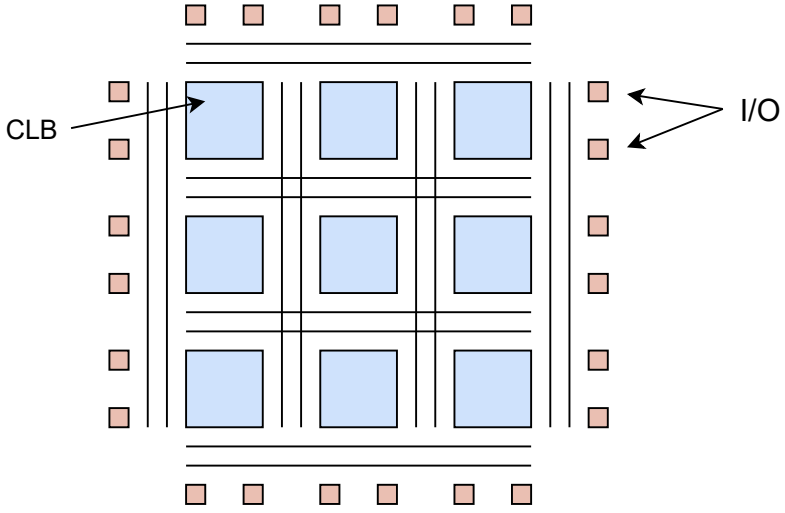


Figura 2.12: Arquitectura general de un FPGA

Los elementos fundamentales que conforman una celda lógica típica de FPGA son los siguientes:

- Una tabla de búsqueda (LUT, por sus siglas en inglés) de 4 entradas.
- Un *flip-flop* D.
- Un multiplexor 2 a 1.

La estructura y conexiones entre estos elementos se muestra en la Figura 2.13, en donde la configuración del multiplexor 2 a 1 permite la selección de dos tipos de funciones: *combinacional o secuencial*. Las LUT están conformadas por un conjunto de celdas de memoria

programables, cuya salida se define mediante un multiplexor, lo que permite elegir una celda de memoria específica.

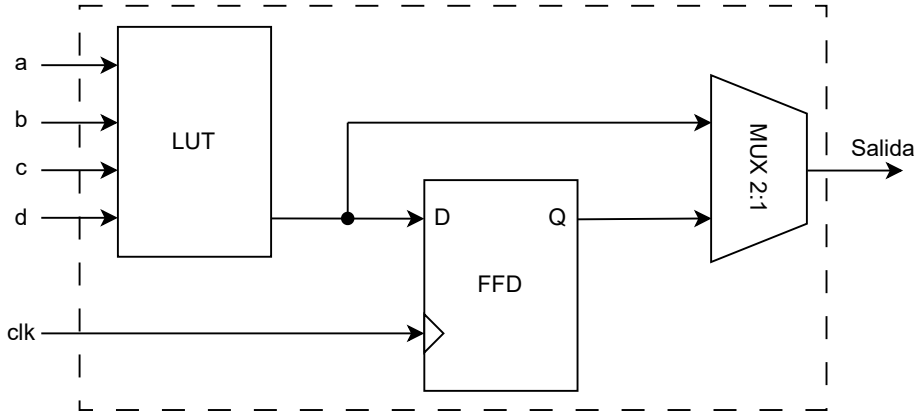


Figura 2.13: Estructura típica de un CLB

Capítulo 3

Estado del Arte

Previo al desarrollo de un proyecto, es prudente realizar una investigación preliminar que permita definir el punto de partida, así como determinar los objetivos y alcances del trabajo. Para ello, se realiza un análisis del nivel de desarrollo actual de un determinado campo tecnológico, las técnicas que han brindado los mejores resultados y los desafíos que aún están presentes en el área. Al resultado de esta investigación se le conoce como el Estado del Arte.

A pesar de que los avances tecnológicos han permitido disponer de una mayor cantidad de recursos en un solo circuito integrado, los retos continúan presentes durante la implementación de modelos computacionales de gran complejidad, tal como lo son las ANN. Por este motivo se ha buscado diseñar arquitecturas optimizadas, las cuales aprovechen las características y recursos de los FPGA. Es importante mencionar que existen múltiples tipos de ANN, las cuales cuentan con propósitos y topologías diferentes, por lo que los métodos y técnicas de descripción varían considerablemente entre algunas de ellas. Es por dicho motivo que la investigación realizada se acota al tipo de red que será implementada en este trabajo, la cual corresponde a una FFNN. Esta investigación permitirá identificar con claridad las técnicas de descripción que han presentado el mejor desempeño, así como las desventajas de cada una, dándonos la oportunidad de encontrar el o los métodos con mayor potencial para desarrollos futuros. Los múltiples retos del diseño en hardware involucra la multiplicación entre el vector de entradas y la matriz de pesos, la descripción de la función de activación, el almacenamiento de los pesos, la selección del formato numérico, entre otros [11], por lo que en este capítulo se presentan los trabajos y propuestas para cada uno de estos aspectos.

3.1. Entrenamiento

Una de las características que comparten los trabajos encontrados en el Estado del Arte, es que el entrenamiento de la red neuronal se realiza en las CPU de propósito general, mientras que la etapa de clasificación se lleva a cabo en el FPGA. Esto se debe principalmente a que el entrenamiento es un proceso exigente en cuanto a recursos y consumo energético, que puede resultar contraproducente cuando lo que principalmente se busca es mejorar el desempeño del sistema de clasificación. Una vez entrenada la ANN, se transfieren los parámetros necesarios al FPGA, en donde se realizan las tareas de clasificación con nuevos

datos [12], [13], [14]. A este tipo de arquitecturas se les denomina comúnmente como *aceleradores* [5], [15]. En la Figura 3.1 se ilustra la arquitectura e interconexiones de un acelerador, encontrado típicamente en estos trabajos.

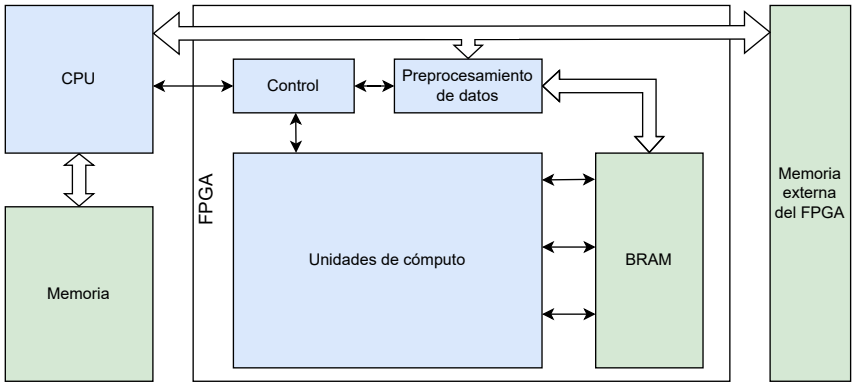


Figura 3.1: Diagrama de bloques de un acelerador de ANN típico, basado en FPGA [5]

En el esquema se muestra que la etapa de entrenamiento se lleva a cabo en la CPU y su memoria correspondiente, en donde se almacenan los parámetros necesarios, tal como los pesos y umbrales que serán transferidos posteriormente al FPGA. La segunda sección de un acelerador consiste en los bloques integrados en el FPGA, los cuales conforman la red neuronal que realizará la clasificación. Para el almacenamiento de los pesos y umbrales suele emplearse, dependiendo de la cantidad de datos, la memoria embebida del FPGA o una memoria externa. Las unidades de cómputo corresponden a las neuronas y otros elementos que realizan los cálculos como el producto de entradas y pesos, mientras que el bloque de control coordina y sincroniza a los elementos de la unidad de cómputo. El bloque de preprocesamiento se emplea cuando los datos de entrada requieren un tratamiento previo.

3.2. Optimización de recursos

En el año 2007, en [16] se presentó por primera vez la idea de implementar una FFNN describiendo únicamente su capa más grande, es decir, en lugar de describir la red completa, se limitaron a implementar la capa con más neuronas y así reutilizarla. Lo anterior se realiza mediante un bloque de control, el cual asigna las entradas, pesos y umbrales dependiendo de la capa que se esté ejecutando. Este concepto resultó ideal, ya que el paralelismo de una FFNN aplica únicamente para las neuronas de una misma capa, sin embargo, la ejecución entre una capa y otra se lleva a cabo de forma secuencial, debido a que las neuronas de una capa toman como entradas los resultados de la capa anterior. Por ello, la propuesta de [16] demostró que es posible ahorrar recursos ejecutando una capa a la vez, sin afectar la velocidad de procesamiento de la FFNN, lo cual marcó la pauta para los desarrollos futuros.

3.3. Almacenamiento de parámetros de entrenamiento

Otro de los aspectos que influyen en la eficiencia del diseño, es el almacenamiento de los pesos en el FPGA y el manejo de memoria que implica. En las Figuras 3.2 y 3.3 se ilustra la información obtenida de la investigación realizada en [2], en donde se destaca el gran consumo energético que representa la lectura de una memoria de acceso aleatorio estática (SRAM, por sus siglas en inglés) y de una memoria de acceso aleatorio dinámica (DRAM, por sus siglas en inglés), comparándolo con el consumo generado al realizar operaciones como la adición y la multiplicación.

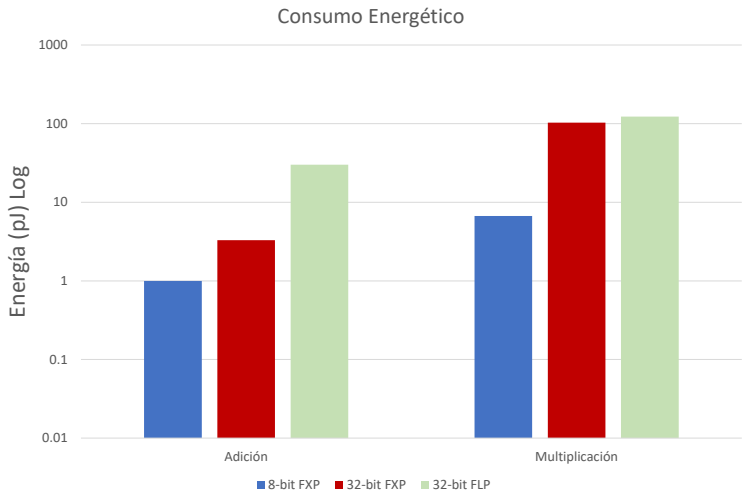


Figura 3.2: Comparación del consumo energético entre operaciones [2]

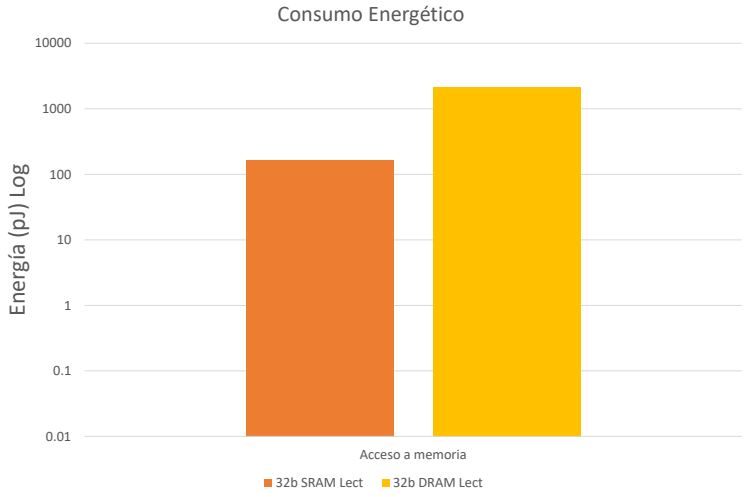


Figura 3.3: Comparación del consumo energético entre lectura de SRAM y DRAM [2]

Para sobrellevar este inconveniente, en diversos trabajos aprovechan la memoria embebida

del FPGA, en la cual almacenan los pesos y umbrales, permitiendo el ahorro de energía e incrementar la velocidad de cómputo [14].

3.4. Herramientas de software

Un paradigma que ha surgido, dirigido a facilitar la generación de hardware específico, son los trabajos que proponen herramientas de diseño utilizando interfaces gráficas, en las cuales es posible generar *automáticamente* archivos de configuración de hardware de acuerdo con las opciones seleccionadas por el usuario [15]. Sin embargo, un mapeo directo de software a hardware mediante el empleo de estas herramientas, comúnmente resulta en un uso de recursos ineficiente, alta latencia y requerimientos de memoria elevados, además de mayor consumo energético. Por lo tanto, no se ahondará en la investigación de trabajos con este enfoque.

3.5. Multiplicación de pesos y entradas

Además del entrenamiento y el almacenamiento de los parámetros, otra variable que hay que considerar durante el diseño, es cómo se realiza la multiplicación entre la matriz de pesos y el vector de entradas. Se han propuesto estructuras como la mostrada en la Figura 3.4, la cual consiste en implementar un multiplicador por cada entrada y su respectivo peso, para posteriormente sumarlos [17]. Sin embargo, se ha comprobado que dicha arquitectura carece de practicidad y eficiencia debido a la gran exigencia de recursos de hardware. Esto además de limitar el tamaño de la FFNN implementada en el FPGA, disminuye el número de aplicaciones en las que podría ser empleada. Debido a ello, se han propuesto otros enfoques, como el caso de [14], en donde implementan unidades neuronales de procesamiento paralelo-serial, que consiste en adquirir de manera serial los datos de entrada, sin desaprovechar el paralelismo que ofrecen los FPGA. Este enfoque, ilustrado en la Figura 3.5, permite describir solo un bloque de la función de activación en vez de uno por neurona, lo que disminuye considerablemente los recursos consumidos.

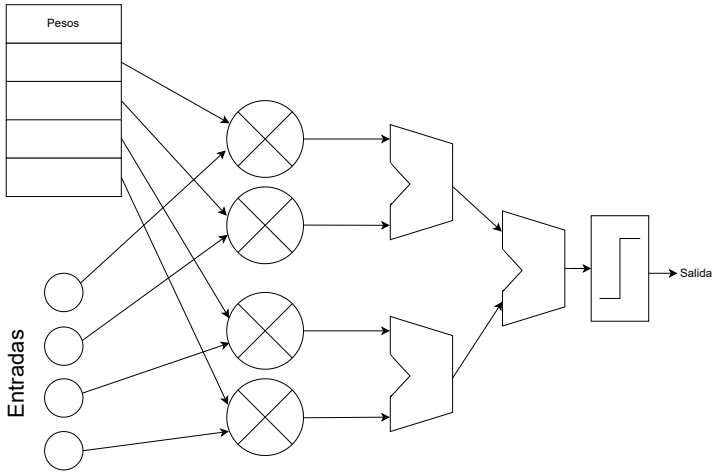


Figura 3.4: Estructura de una neurona de procesamiento parcialmente paralelo [17]

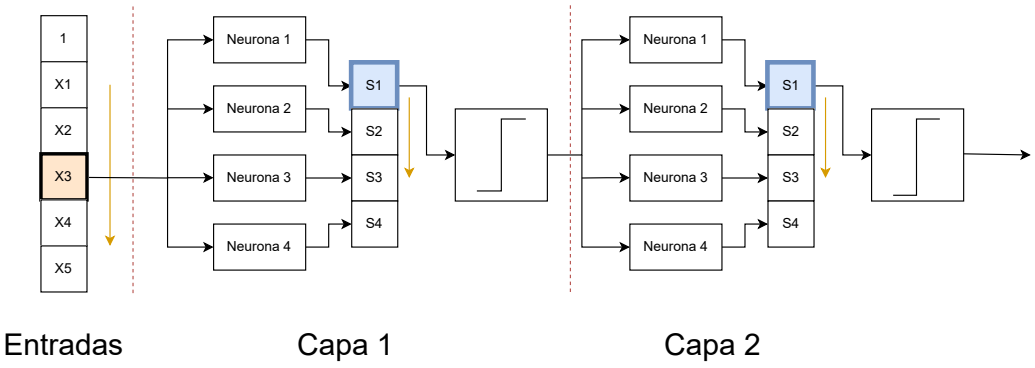


Figura 3.5: FFNN con procesamiento paralelo-serial [14]

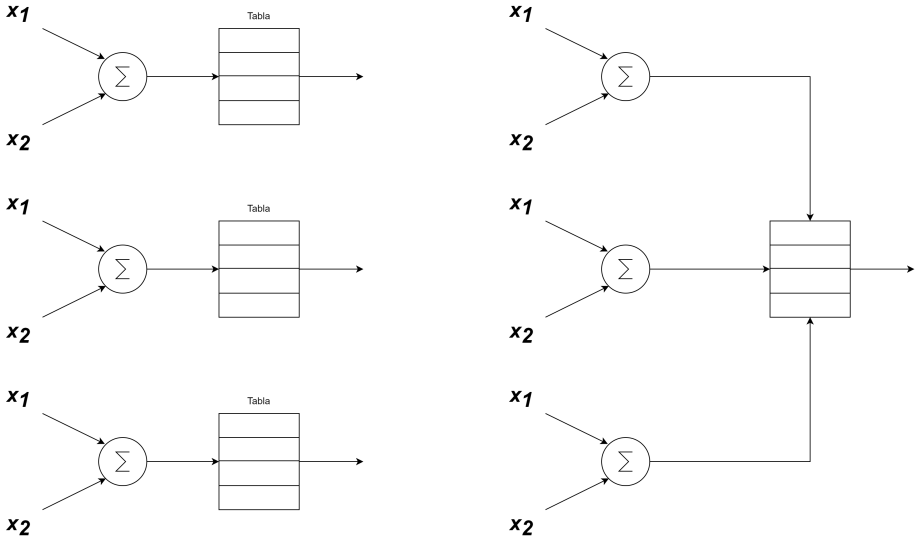
3.6. Función sigmoide y la precisión numérica

La función de activación sigmoide es por mucho la más empleada dentro del diseño de redes neuronales, por lo que en este apartado únicamente se presentarán los métodos aplicados para dicha función, sin embargo, se ha demostrado que estos mismos métodos pueden ser empleados para otro tipo de funciones de activación.

La eficiencia, precisión y velocidad de ejecución de una ANN implementada en hardware, depende en gran parte de cómo fue descrita su función de activación, por lo que es imprescindible elegir el método que cumpla, en mayor medida, con los requerimientos planteados. Cuando se está describiendo en hardware, uno de los aspectos más importantes a considerar es el formato numérico que será empleado, es decir, seleccionar si la representación de los datos será en punto fijo o punto flotante. Se ha demostrado que al emplear el formato de punto flotante se obtiene una mayor precisión numérica, sin embargo, con la representación de punto fijo se logra mayor eficiencia de recursos, por lo que su aplicación es de gran utilidad para sistemas embebidos con recursos limitados [18]. Las principales dificultades de la descripción en punto fijo, es la presencia de la exponencial y la operación de división en la función sigmoide. Los dos aspectos que más se consideran al aplicar una determinada técnica son: la cantidad de recursos en hardware utilizados y la precisión numérica. Es importante resaltar que cuando se persigue una alta eficiencia en cuanto a recursos lógicos, la precisión numérica es sacrificada, mientras que, cuando se desea obtener la mayor precisión posible, invariablemente es exigida una mayor cantidad de recursos. Debido a ello, en los trabajos más recientes se han enfocado en la búsqueda de métodos numéricos para la descripción de la función sigmoide, en donde se optimicen ambos aspectos y con ello, la implementación de ANN en hardware sea cada vez más viable.

Dentro de las principales técnicas de aproximación se encuentra la implementación de tablas de búsqueda (LUT), expansión de las series de Taylor, aproximación lineal por partes (PWL, por sus siglas en inglés), entre otras. Cada una de estas técnicas presentan variaciones que pueden mejorar o mermar el desempeño del sistema neuronal, por lo que es importante elegir aquella que mejores resultados genere. En cuanto al desarrollo de series de Taylor, el problema principal se centra en la necesidad de aplicar una gran cantidad de multiplicaciones,

por lo que se ha descartado esta técnica en diversas ocasiones [18]. Los métodos que emplean tablas de búsqueda consisten en calcular previamente un intervalo de valores de la función y posteriormente almacenarlos en una LUT. Una forma de implementar este método es utilizar una tabla de búsqueda por cada neurona, lo cual se ilustra en la Figura 3.6(a). Sin embargo, esto usualmente se traduce en un gran consumo de recursos de hardware. Otra opción para implementar este método, ilustrado en la Figura 3.6(b), es asignar una LUT a un grupo de neuronas. Si los grupos se conforman por k neuronas, esta variación será k veces más lenta que el primer método, debido a que el acceso a la LUT se realiza con una neurona a la vez. En general, la implementación con LUT no es popular actualmente, debido a la cantidad de recursos de almacenamiento requeridos para obtener una alta precisión.



(a) Primer método (b) Segundo método

Figura 3.6: Variaciones de la implementación de las tablas de búsqueda

Por otro lado, el método PWL ha sido ampliamente usado y actualmente es la técnica predominante en cuanto a descripción de funciones en hardware. Este método consiste en dividir en segmentos el intervalo de entrada y emplear una función lineal que aproxime cada segmento. Esto ha presentado resultados satisfactorios, ya que brinda flexibilidad en cuanto al intercambio de recursos y precisión, según el número de segmentos en los que se divida la función. En [19] presentan un diseño conformado por tres bloques, en donde el primer módulo utiliza PWL, el segundo emplea aproximación de series de Taylor y el tercero integra los resultados de los módulos anteriores, con la aproximación de Newton-Raphson (NRA, por sus siglas en inglés). En este trabajo realizan una comparación de los resultados de su propuesta con los resultados obtenidos en otros trabajos, los cuales se muestran en la Tabla 3.1 y la Tabla 3.2. Para mostrar sus resultados emplearon un PWL de 4 segmentos, también llamado aproximación lineal por partes de una función no lineal (PLAN, por sus siglas en inglés).

Tabla 3.1: Comparación de la precisión de aproximación, empleando E_{avg} y E_{max} [19]

Referencia	E_{max}	E_{avg}	Entrada	Salida	Tech.
Hajduk (McLaurin) [20]	1.192E-07	1.453E-08	32b FP	32b FP	FPGA
Hajduk (Pade) [20]	1.192E-07	3.268E-09	32b FP	32b FP	FPGA
Zaki et al. [21]	1.0E-02	N/A	32b FP	32b FP	FPGA
Tiwari et al. [22]	4.77E-05	N/A	32b FXP	32b FXP	FPGA
Tiwari et al. [22]	9.97E-11	N/A	64b FXP	64b FXP	FPGA
Tsmots et al. (PWL) [18]	1.85E-02	5.87E-03	16b FXP	16b FXP	FPGA
Wei et al. [23]	1.25E-02	4.2E-03	16b FXP	12b FXP	FPGA
Sun et al. [24]	1.0E-03	5.3E-04	10b FXP	32b FXP	ASIC
Qin et al. [25]	7.6E-03	1.6E-03	12b FXP	12b FXP	ASIC
PLAN (16b/16b) [19]	1.89E-02	5.87E-03	16b FXP	16b FXP	FPGA
PLAN (16b/8b) [19]	2.35E-02	6.64E-03	16b FXP	8b FXP	FPGA
PLAN (8b/8b) [19]	2.54E-02	7.19E-03	8b FXP	8b FXP	FPGA
NRA (16b/16b) [19]	5.72E-04	8.60E-05	16b FXP	16b FXP	FPGA
NRA (16b/8b) [19]	8.21E-03	4.64E-03	16b FXP	8b FXP	FPGA
NRA (8b/8b) [19]	1.83E-02	4.72E-03	8b FXP	8b FXP	FPGA

Tabla 3.2: Comparación de recursos utilizados y desempeño en FPGA [19]

Referencia	LUT	FF	DSP	Frec (MHz)	Retardo (ns)	Consumo (mW)
Hajduk (McLaurin) [20]	1916	792	4	89.3	940.8	N/A
Hajduk (Pade) [20]	2624	1059	8	97.1	494.4	N/A
Zaki et al. [21]	363	566	2	358.166	N/A	N/A
Tiwari et al. (32b) [22]	1388	597	22	N/A	2130 (Pos) 1590 (Neg)	N/A
Tiwari et al. (64b) [22]	1388	597	22	N/A	5830 (Pos) 4250 (Neg)	N/A
Wei et al. [23]	140	23	0	N/A	9.856	6
PLAN (16b/16b) [19]	235	153	0	200	30	9
PLAN (16b/8b) [19]	230	148	0	200	30	9
PLAN (8b/8b) [19]	99	114	0	200	30	5
NRA (16b/16b) [19]	351	325	6	200	85	26
NRA (16b/8b) [19]	340	309	6	200	85	24
NRA (8b/8b) [19]	142	159	5	200	55	19

La Tabla 3.1 compara los resultados tomando en cuenta el error absoluto máximo (E_{max}) y el error absoluto promedio (E_{avg}), los cuales son empleados como métricas para obtener la precisión de aproximación de cada implementación. En esta misma tabla se especifica el formato numérico de las entradas y salidas, además del dispositivo en el que fue implementado. Por otro lado, la Tabla 3.2 compara la cantidad de recursos de hardware consumidos y el desempeño de cada FPGA, en donde toman en cuenta elementos como las tablas de búsqueda (LUT), *flip-flops* (FF) y bloques de procesamiento digital de señales (DSP, por sus siglas en inglés), así como la frecuencia de operación, tiempos de ejecución y consumo energético en cada FPGA. A partir de las tablas se destaca que las soluciones que emplean formatos numéricos de punto flotante, son las que obtuvieron la mayor precisión de aproximación, sin embargo, son las que más cantidad de recursos utilizaron, además de que usualmente no es necesaria tal precisión en la práctica [19]. Otra observación es que el método de NRA presenta mayor precisión conforme se incrementa el tamaño de los datos, mientras que incrementar el número de bits en el método PWL no mejora la precisión, ya que eso depende principalmente de la selección del número de segmentos. A partir de estas observaciones se concluye que un modelo que cuente con diferentes métodos de aproximación, permite tener una alta flexibilidad de acuerdo a los requerimientos de la aplicación, ya que es posible ajustar parámetros como el número de segmentos en PWL o incluir más términos en la expansión de series de Taylor.

Capítulo 4

Diseño preliminar

Una vez realizada la investigación preliminar mediante la revisión del Estado del Arte y estudiados los fundamentos clave en el Marco Teórico, se cuenta con las herramientas para comenzar con el proceso de diseño.

Un proceso o metodología de diseño se basa en una serie de pasos sistematizados que permiten llevar un control y visualizar con claridad las limitaciones, alcances y porcentaje del progreso, por lo que es importante dedicar tiempo a su planeación y seguimiento. La metodología seleccionada para este trabajo, inspirada en la lectura [26], se estructura en las siete etapas ilustradas en el esquema de bloques de la Figura 4.1.

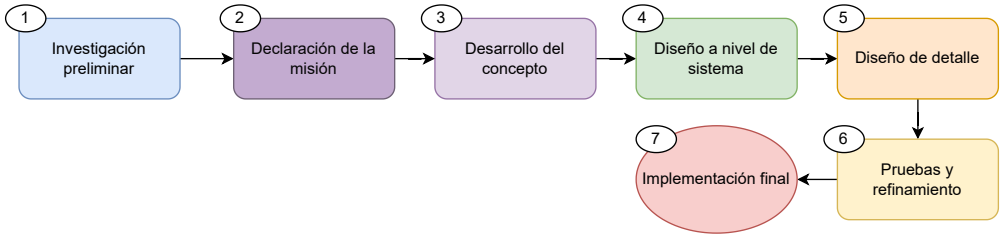


Figura 4.1: Etapas de la metodología de diseño propuesta

La primera etapa, correspondiente a la investigación preliminar, fue llevada a cabo en los capítulos previos, mientras que cada una de las etapas posteriores serán abordadas en lo que resta del trabajo.

4.1. Declaración de la misión

El primer paso para concretar las especificaciones y con ello el concepto de este proyecto, es presentar el propósito y metas del trabajo, lo cual se define formalmente como la *declaración de la misión*. La misión surge a partir de la etapa de *investigación preliminar*, la cual permite esclarecer los retos y oportunidades del área. La declaración de la misión se presenta en la Tabla 4.1, dividiéndose en 3 rubros: descripción del tema, propuesta de valor y metas clave.

Tabla 4.1: Declaración de la misión

Descripción del sistema	- FFNN reconfigurable en FPGA
Propuesta de valor	- Facilidad de reconfiguración - Implementación de bajo costo - Alta velocidad de procesamiento
Metas clave	- Implementar una FFNN en FPGA de recursos lógicos limitados - Probar el sistema con la base de datos de imágenes de dígitos escritos a mano - Comparar el desempeño de la FFNN instrumentada en FPGA con la red programada en Python

4.2. Desarrollo de concepto

El objetivo de esta etapa es definir las funcionalidades y características del sistema, las cuales es posible establecer a partir de una lista de necesidades y especificaciones objetivo, presentadas a lo largo de esta sección.

4.2.1. Identificación de necesidades

Una vez planteada la misión, es posible generar una lista de necesidades, que resulta parte imprescindible en la definición de las especificaciones objetivo y por lo tanto, en el proceso de generación del concepto. Las necesidades que busca satisfacer este trabajo se enlistan en la Tabla 4.2, en donde se evalúa numéricamente la relevancia de cada una, siendo el número 1 la mayor prioridad y 3 la menor prioridad.

Tabla 4.2: Lista de necesidades y su relevancia

Necesidades	Relevancia
Bajo consumo de energía	1
Reconfigurable	1
Bajo costo	1
Escalable	3
Alta velocidad de procesamiento	1
Robusto	2
Alta precisión de clasificación	3

4.2.2. Especificaciones objetivo

Cuando ya se han determinado las necesidades y sus prioridades, es pertinente traducir a términos técnicos las cualidades a las que se aspiran, lo que permitirá establecer los alcances y límites del trabajo. Dichas cualidades son conocidas como *especificaciones objetivo*, las cuales emplean métricas dirigidas al control del progreso y cumplimiento de los objetivos. Debido a que el alcance de este trabajo se enfoca en instrumentar la FFNN previamente implementada en Python, ya se cuenta con una serie de especificaciones que representan los parámetros de dicha red, en donde se emplearán las técnicas de optimización pertinentes, para cumplir con las especificaciones planteadas. Cada una de las métricas y su correspondiente valor se presenta en la Tabla 4.3.

Tabla 4.3: Especificaciones objetivo

Métrica	Unidad	Valor
Tamaño de la imagen	pixel	784
Tiempo de reconocimiento	segundos	≤ 1
Precisión de clasificación	%	89.92
Número de capas	Num	3
Neuronas en la capa más grande	Num	30
Funciones de activación	Num	1

4.2.3. Definición del concepto

Una vez planteada la misión, necesidades y especificaciones objetivo del trabajo, en este apartado se describirá detalladamente el concepto que se ha construido a partir de los criterios mencionados.

Se propone un sistema de reconocimiento de imágenes de dígitos escritos a mano, conformado por una FFNN instrumentada en un FPGA de recursos limitados, en donde se emplean técnicas de optimización en hardware, con el objetivo de cumplir con las especificaciones planteadas. El entrenamiento se realiza de manera externa, para posteriormente transmitir los parámetros de configuración y de entrenamiento al FPGA.

4.3. Diseño a nivel de sistema

En este trabajo se adoptó la técnica de diseño Top-Down [27], en donde se comienza por definir el sistema desde el más alto nivel de abstracción, para posteriormente ahondar en cada subsistema y sus características. Este método fue seleccionado debido a que permite la reutilización del diseño, además de facilitar la identificación de errores y minimizar los tiempos de desarrollo.

En general, la implementación de una FFNN puede dividirse en dos etapas, la de *entrenamiento* y la de *clasificación*. En la primera etapa se definen los parámetros, tal como el número de capas, entradas, salidas y neuronas por capa. Posteriormente, se lleva a cabo el entrenamiento de la red mediante el método del descenso del gradiente u otros algoritmos, con los cuales se calculan los pesos sinápticos y umbrales. La segunda etapa consiste en probar el desempeño de la red neuronal con los pesos y umbrales calculados en la etapa previa, realizando la clasificación de datos que no se hayan presentado antes a la red. Dicho lo anterior, la representación más abstracta del sistema propuesto se encuentra dividida en dos unidades, las cuales se ilustran con sus entradas y salidas en la Figura 4.2. La unidad de la izquierda (entrenamiento) representa la primera etapa, ya que es la responsable tanto de entrenar a la FFNN, es decir, calcular los pesos sinápticos y umbrales que serán usados en la segunda etapa, así como de transmitir los parámetros necesarios a la unidad de clasificación. Por otro lado, el bloque de la derecha (clasificación) contiene a la FFNN, que recibe la configuración y parámetros de entrenamiento, con los cuales realiza el reconocimiento de nuevas imágenes. En las siguientes secciones se ahondará en cada una de estas unidades, incrementando el nivel de detalle en la descripción de su funcionamiento y de los bloques que las conforman.

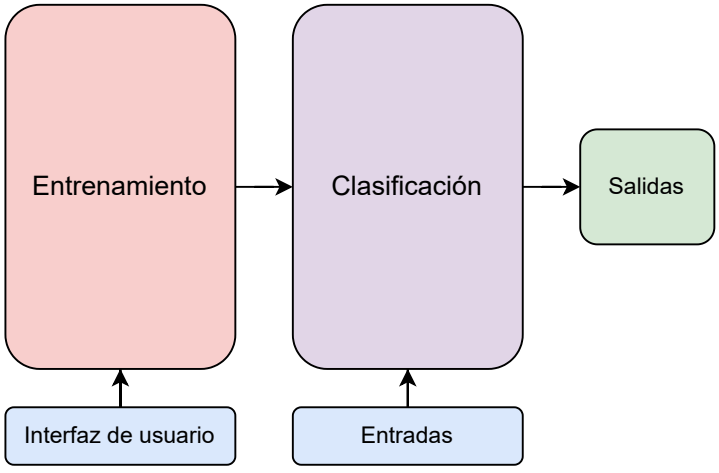


Figura 4.2: Representación del más alto nivel de abstracción del sistema

4.3.1. Unidad de entrenamiento

Esta unidad es responsable de generar los pesos sinápticos y umbrales a partir de los datos de entrenamiento, para posteriormente transmitirlos a la unidad de clasificación. El proceso que será llevado a cabo se resume en los 5 pasos mostrados en la Figura 4.3.

Figura 4.3: Proceso que lleva a cabo la unidad de entrenamiento

Para realizar este proceso, se propone la implementación de una interfaz gráfica de usuario (GUI, por sus siglas en inglés), ya que esta permite ingresar los datos de entrenamiento y parámetros de configuración de manera directa, además de que facilita la ejecución del entrenamiento y de la transmisión de datos, empleando elementos interactivos. Cada paso mostrado en la Figura 4.3 se describe con detalle a continuación:

1. ***Proponer los parámetros de configuración de la FFNN***: el primer paso de diseño es proponer los parámetros de la red neuronal, es decir, seleccionar el número de capas, el número de neuronas en cada capa y la cantidad de entradas y salidas. Es importante tomar en cuenta que la selección de los parámetros es un proceso empírico y será necesario repetir este paso hasta obtener los resultados deseados.

2. ***Ingresar los parámetros de configuración de la FFNN a la GUI***: la GUI diseñada cuenta con campos de escritura, en donde se colocan los parámetros definidos en el paso anterior.

3. ***Cargar datos de entrenamiento***: en la misma GUI el usuario debe de ingresar un archivo que contenga los datos con los cuales se entrenará a la FFNN.

4. ***Realizar entrenamiento***: ya que se ingresaron los datos de entrenamiento y los parámetros de configuración, en la GUI se habilitará la opción para entrenar la red, la cual debe iniciarse por el usuario. Si no se han ingresado los parámetros y datos correspondientes a los pasos anteriores, la GUI no activará esta opción de entrenamiento.

5. ***Transmitir parámetros a la unidad de clasificación***: la GUI indica mediante un cuadro de texto cuando ha finalizado el entrenamiento de la red, es decir, cuando ya se terminaron de calcular los pesos y umbrales, permitiendo al usuario iniciar la transmisión a la unidad de clasificación.

Las entradas y salidas de la unidad de entrenamiento se muestran en la Tabla 4.4.

Tabla 4.4: Entradas y salidas de la unidad de entrenamiento

Tipo	Datos	Descripción
Entradas	Parámetros de configuración	Número de capas, neuronas, entradas y salidas
	Datos de entrenamiento	Vectores con los datos de entrenamiento
Salidas	Parámetros de configuración	Número de capas, neuronas, entradas y salidas
	Parámetros de entrenamiento	Pesos y umbrales calculados durante el entrenamiento

4.3.2. Unidad de clasificación

La unidad de clasificación recibe directamente los parámetros calculados en la fase de entrenamiento. El proceso llevado a cabo en esta etapa se ilustra en la Figura 4.4.

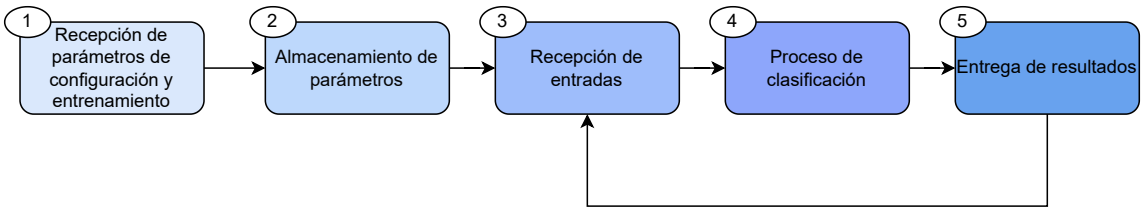


Figura 4.4: Proceso que lleva a cabo la unidad de clasificación

Cada uno de estos pasos se describe con detalle a continuación:

1. ***Recepción de parámetros de configuración y entrenamiento***: una vez calculados los pesos sinápticos y umbrales en la unidad de entrenamiento, son transmitidos y recibidos por la unidad de clasificación, al igual que el número de capas, entradas, salidas y neuronas por cada capa.

2. ***Almacenamiento de parámetros***: tanto los parámetros de configuración como los de entrenamiento, son almacenados en los registros y bloques de memoria correspondientes.

3. ***Recepción de entradas***: cuando todos los parámetros estén almacenados, la unidad se encontrará lista para recibir las entradas provenientes del exterior, para así comenzar con el proceso de clasificación.

4. ***Proceso de clasificación***: con los parámetros y datos de entrada recibidos, se realizan los cálculos de las neuronas, hasta llegar a los resultados de la última capa.

5. ***Entrega de resultados***: una vez que se ha concluido la clasificación es posible extraer los datos finales, para posteriormente repetir el proceso desde el paso 3, en donde se reciben nuevas entradas para comenzar con el siguiente proceso de clasificación.

Como parte de la metodología de diseño, es necesario analizar la estructura y secuencia de las operaciones desde una perspectiva de hardware, con el objetivo de plantear una arquitectura que sea eficiente para llevar a cabo la FFNN. Por lo tanto, a continuación se analizará paso a paso la suma de productos que realiza una neurona.

Las neuronas, siendo las unidades básicas que constituyen a las ANN, son responsables de realizar la multiplicación de la matriz de pesos sinápticos $\mathbf{B}$ por el vector de entradas $\vec{A}$, los cuales se representan matemáticamente con las ecuaciones (4.1) y (4.2), respectivamente. La matriz de pesos está conformada por N_0+1 columnas y M_0+1 renglones, mientras que el vector de entradas contiene N_0+1 elementos. En el Marco Teórico se demostró que es posible incorporar a la suma de productos el umbral como ω_0, añadiendo al vector de entradas el valor $X = 1$.

$$\vec{A} = \begin{pmatrix} 1 \\ X_1 \\ X_2 \\ \vdots \\ X_{N_0} \end{pmatrix} \tag{4.1}$$

$$\mathbf{B} = \begin{pmatrix} \omega_{00} & \omega_{10} & ... & \omega_{N_0 0} \\ \omega_{01} & \omega_{11} & ... & \omega_{N_0 1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{0M_0} & \omega_{1M_0} & ... & \omega_{N_0 M_0} \end{pmatrix} \tag{4.2}$$

En las ecuaciones (4.3) y (4.4) se lleva a cabo la multiplicación de $\mathbf{B}$ por $\vec{A}$, obteniendo así la suma de productos de las neuronas de una capa. Una vez calculadas se evalúa la función de activación, con lo que se obtiene el resultado final X_i de una neurona (4.5).

$$\begin{pmatrix} S_0 \\ S_1 \\ \vdots \\ S_{M_0} \end{pmatrix} = \mathbf{B}\vec{A} = \begin{pmatrix} \omega_{00} & \omega_{10} & ... & \omega_{N_0 0} \\ \omega_{01} & \omega_{11} & ... & \omega_{N_0 1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{0M_0} & \omega_{1M_0} & ... & \omega_{N_0 M_0} \end{pmatrix} \times \begin{pmatrix} 1 \\ X_1 \\ X_2 \\ \vdots \\ X_{N_0} \end{pmatrix} \tag{4.3}$$

$$\begin{pmatrix} S_0 \\ S_1 \\ \vdots \\ S_{M_0} \end{pmatrix} = \begin{pmatrix} \omega_{00} + \omega_{10} * X_1 + \omega_{20} * X_2 + & ... & +\omega_{N_0 0} * X_{N_0} \\ \omega_{01} + \omega_{11} * X_1 + \omega_{21} * X_2 + & ... & +\omega_{N_0 1} * X_{N_0} \\ \vdots & \ddots & \vdots \\ \omega_{0M_0} + \omega_{1M_0} * X_1 + \omega_{2M_0} * X_2 + & ... & +\omega_{N_0 M_0} * X_{N_0} \end{pmatrix} \tag{4.4}$$

$$X_i = f(S_i) \tag{4.5}$$

La multiplicación de una matriz $\mathbf{B}$ de tamaño MxN por el vector $\vec{A}$ de N elementos, puede realizarse de dos formas [14]:

1. La primera forma consiste en multiplicar en paralelo todos los elementos del vector $\vec{A}$, por todos los elementos del vector renglón $\vec{B}_i$, el cual es obtenido de la matriz de pesos sinápticos. Debido a que la matriz cuenta con N columnas, se requieren N multiplicadores para que la suma de productos S_i se calcule en un solo ciclo de reloj. Por lo tanto, S_0 se calcula en el primer ciclo, S_1 en el segundo ciclo y así sucesivamente. Esto significa que al contar con M neuronas en la capa, el proceso para obtener todos los resultados toma M ciclos de reloj, lo cual se ilustra en la Figura 4.5.

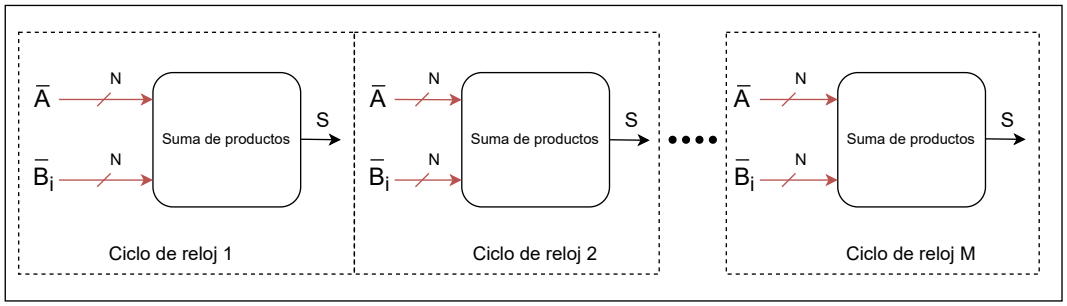


Figura 4.5: Primer método para multiplicar $\mathbf{B}\vec{A}$

2. La segunda forma de llevar a cabo la operación, es multiplicar en paralelo un solo elemento del vector de entradas $\vec{A}$ por cada elemento del vector columna $\vec{B}_j$, igualmente obtenido de la matriz de pesos sinápticos. En esta opción se calcula solo una suma parcial de cada una de las M neuronas de la capa, por lo que se requieren M multiplicadores. Además, si se cuenta con N datos de entrada, se tomarán N ciclos de reloj para completar las sumas parciales de todas las neuronas de la capa. En la Figura 4.6 se ilustra este segundo método, en el cual a diferencia del primero, en donde se tiene una sola neurona en cada ciclo de reloj, aquí se cuenta con las M neuronas. Una vez transcurridos los N ciclos de reloj, se obtienen las sumas parciales de todas las neuronas.

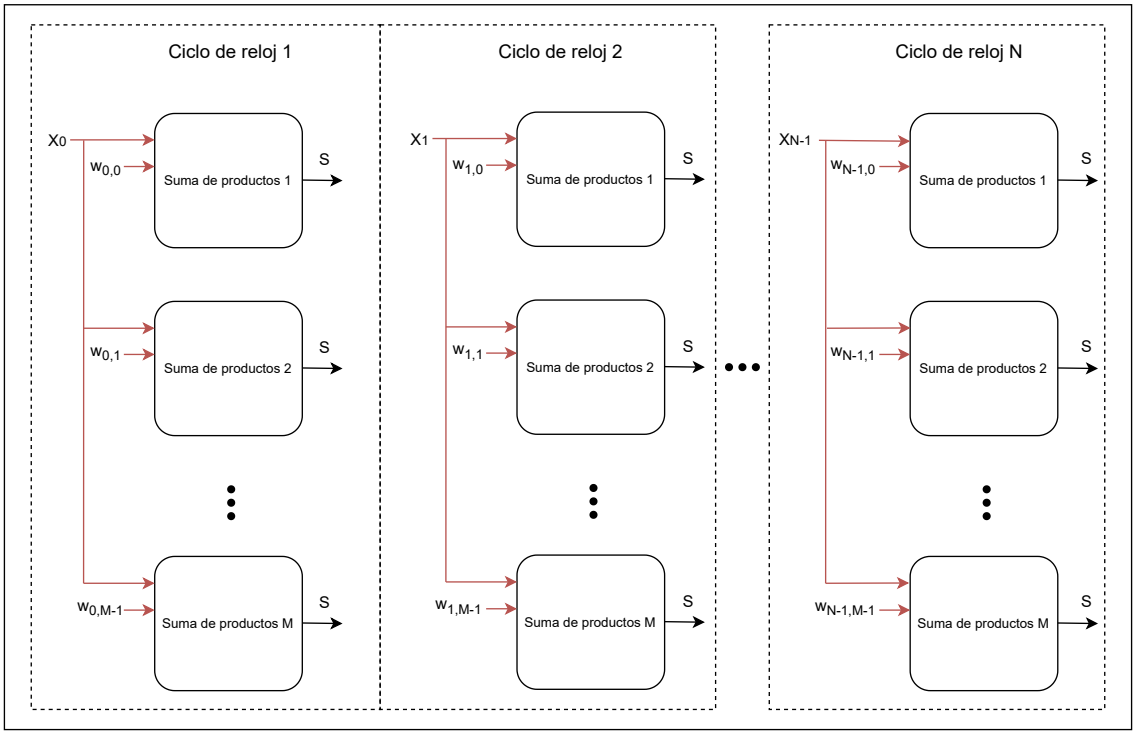


Figura 4.6: Segundo método para multiplicar $\mathbf{B}\vec{A}$

A partir del análisis realizado anteriormente, es posible tomar en cuenta los siguientes puntos sobre ambos métodos:

- **Primer método**: requiere la multiplicación simultánea de diferentes valores, tanto de $\vec{A}$ como de $\vec{B}_i$.
- **Segundo método**: requiere la multiplicación simultánea de un argumento fijo de $\vec{A}_i$, por los diferentes elementos de $\vec{B}_j$. Esto resulta ser una operación de tipo escalar por vector, siendo el escalar una de las entradas.

Desde la perspectiva matemática ambas operaciones brindan los mismos resultados, sin embargo, son diferentes desde la perspectiva de hardware. En el segundo método se consumen menos recursos, debido a que se multiplica un escalar (un elemento del vector de entradas) por un vector, mientras que en el primer método es necesario acceder a más de un elemento a la vez del vector de entradas, por lo que se ocupa mayor cantidad de líneas de datos. Además, esto permite que el vector de entradas $\vec{A}$ sea alimentado al sistema de manera serial, en vez de cargar todos los elementos al mismo tiempo. Por lo tanto, el segundo método es la opción seleccionada para llevar a cabo la suma de productos de cada neurona.

Debido a que la segunda opción permite recibir las entradas de forma serial, cada neurona debe de contar con un módulo de multiplicación y acumulación (MAC). Partiendo de esto, es posible definir un diagrama de bloques que represente la arquitectura de la FFNN descrita en el FPGA, la cual se ilustra en la Figura 4.7.

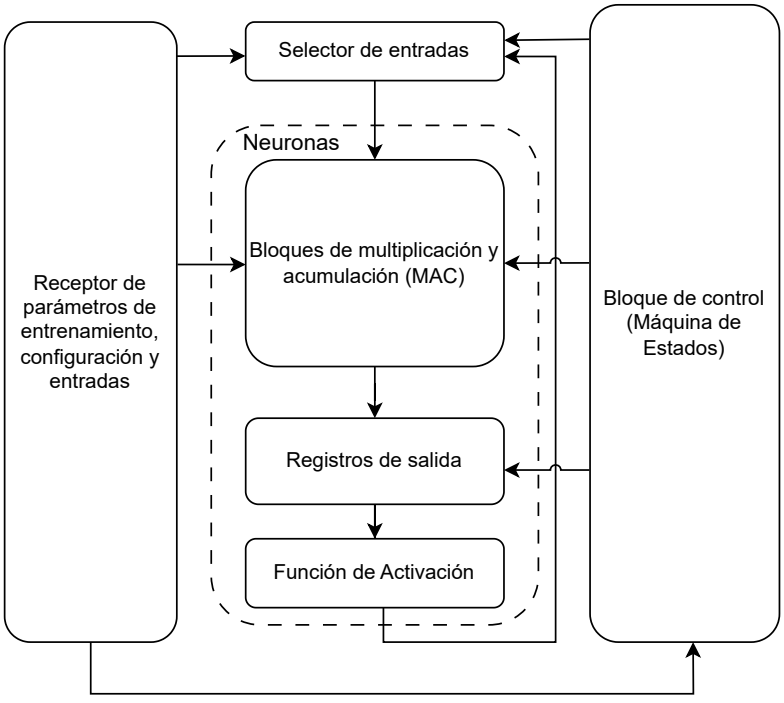


Figura 4.7: Diagrama de bloques de la unidad de clasificación

- ***Receptor de parámetros de entrenamiento, configuración y entradas***: este bloque se encarga de recibir los parámetros de entrenamiento y de configuración, provenientes de la unidad de entrenamiento, así como los datos de entrada que lleguen del exterior. A su vez, carga los parámetros de configuración y entrenamiento en los elementos de almacenamiento correspondientes.

- ***Selector de entradas***: este bloque define las entradas que serán procesadas por las neuronas. Cuando se trata de la primera capa, entonces selecciona las entradas que provienen del exterior, mientras que para las demás capas, las neuronas reciben los resultados de la capa anterior.

- ***Multiplicación y acumulación (MAC)***: este bloque realiza la multiplicación de la matriz de pesos por el vector de entradas, obteniendo así la suma de productos.

- ***Registros de salida***: son los registros que almacenan las sumas de productos de cada neurona y posteriormente los transfieren al bloque de la función de activación.

- ***Función de activación***: este bloque recibe la suma de productos de los bloques MAC por medio de los registros de salida, para posteriormente calcular el resultado de la función sigmoide. Este dato se retroalimenta al bloque selector de entradas para ser procesado por las neuronas de la siguiente capa.

- ***Bloque de control***: es el bloque encargado de generar las señales de control de la unidad, correspondientes a las líneas de selección de los multiplexores, escritura a registros y lectura de memorias.

- ***Neuronas***: las neuronas corresponden al conjunto de los bloques MAC, los registros de salida y la función de activación.

Las entradas y salidas de la unidad de clasificación se muestran en la Tabla 4.5.

Tabla 4.5: Entradas y salidas de la unidad de clasificación

Tipo	Datos	Descripción
Entradas	Parámetros de configuración	Número de capas, neuronas, entradas y salidas
	Parámetros de entrenamiento	Pesos y umbrales calculados en el entrenamiento
	Datos de entrada	Vector de datos provenientes del exterior
Salidas	Resultados de la FFNN	Resultados de la última capa

Capítulo 5

Diseño de detalle

En el capítulo anterior se presentó un diagrama de bloques con dos unidades, la de entrenamiento y la de clasificación, lo cual representa el más alto nivel de abstracción del sistema. Siguiendo la metodología Top-Down, en este capítulo es necesario detallar la tecnología y las técnicas que serán empleadas para llevar a la realidad lo propuesto.

5.1. Unidad de entrenamiento

Como se ha mencionado anteriormente, el entrenamiento de la red es un proceso exigente en cuanto a recursos, tiempo y consumo energético, además de que nuestro interés principal es optimizar el desempeño de la unidad de clasificación. Por lo tanto, se propone implementar la GUI en una CPU de propósito general, para llevar a cabo el entrenamiento de la red. La FFNN fue desarrollada mediante el lenguaje de programación Python, empleando un formato numérico de punto flotante de 32 bits. En la Figura 5.1 se presenta la GUI también generada con Python, en la cual se muestran cada uno de los elementos que permiten ingresar los parámetros de configuración, los datos de entrenamiento y los botones encargados de iniciar el entrenamiento y la transmisión de datos.

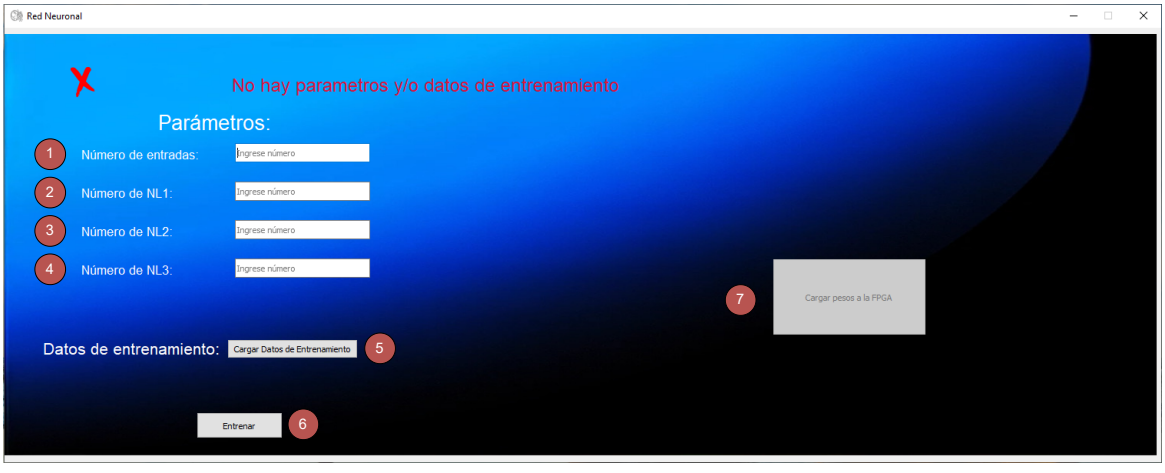


Figura 5.1: Interfaz gráfica de usuario implementada en Python

Cada uno de los elementos se describen a continuación:

1. Cuadro de texto para ingresar el número de entradas.
2. Cuadro de texto para ingresar el número de neuronas de la capa 1.
3. Cuadro de texto para ingresar el número de neuronas de la capa 2.
4. Cuadro de texto para ingresar el número de neuronas de la capa 3.
5. Cuadro de texto para ingresar los datos de entrenamiento.
6. Botón para iniciar el entrenamiento.
7. Botón para transmitir los parámetros de entrenamiento y configuración al FPGA.

5.2. Unidad de clasificación

En esta sección se continúa con el planteamiento del diseño, detallando el funcionamiento, construcción y conexiones de los múltiples bloques que conforman la FFNN mostrada en la Figura 4.7. El diseño se enfoca en llevar a cabo las tareas de clasificación de manera exitosa, priorizando el cumplimiento de las especificaciones y necesidades planteadas, mediante la implementación de técnicas de optimización en hardware, las cuales se describen a lo largo de esta sección.

El objetivo de este trabajo es implementar en FPGA una FFNN, con los mismos parámetros que la red programada y probada previamente en Python, es decir, debe de contar con el mismo número de capas, número de neuronas por capa, número de entradas, número de salidas y misma función de activación. En la Tabla 5.1 se resumen las características de la FFNN implementada en Python y también en el FPGA, mientras que en la Figura 5.2 se ilustra el esquema de la red.

Tabla 5.1: Parámetros de la FFNN

Parámetro	Valor
Número de neuronas de la capa de entrada (número de entradas)	784
Número de neuronas de la capa oculta	30
Número de neuronas de la capa de salida (número de salidas)	10
Función de activación	Sigmoide

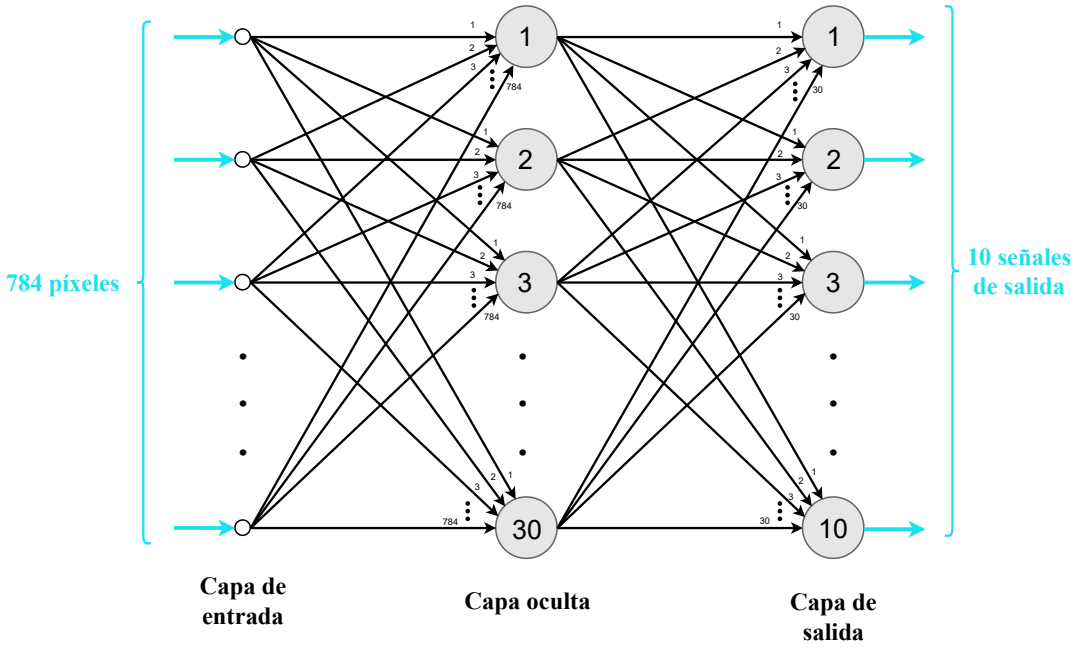


Figura 5.2: Arquitectura de la FFNN implementada

5.2.1. Selección del dispositivo FPGA

Para implementar la unidad de clasificación se propone la descripción de la FFNN en el FPGA EP4CE6F17C8 de Altera, perteneciente a la familia Cyclone IV. Este dispositivo fue seleccionado por diversos factores, dentro de los cuales se encuentra su amplia disponibilidad en el mercado, bajo costo y bajo consumo energético. Además, la razón principal por la cual fue seleccionado este dispositivo, es que es uno de los modelos de bajo consumo con menor cantidad de elementos lógicos, lo cual representa un reto para la descripción de redes neuronales. Debido a esto, la instrumentación exitosa de la FFNN permitirá asegurar que podrá ser implementada en una amplia gama de FPGA. En la Tabla 5.2 se resumen las especificaciones del FPGA seleccionado, en donde se observa que cuenta con la cantidad adecuada de multiplicadores, para implementar el número de neuronas en la capa más grande de la FFNN que se desea instrumentar.

Tabla 5.2: Especificaciones del FPGA EP4CE6F17C8

Parámetro	Valor
Elementos lógicos (LE)	6272
Memoria embebida (Kbits)	270
Multiplicadores de 9x9 embebidos	30
PLL	2
Voltaje de Kernel (V)	1.2
Temperatura de trabajo (°C)	0 - 85

5.2.2. Selección del formato numérico

Es pertinente resaltar la importancia sobre el formato numérico y su impacto en la descripción en hardware, no únicamente por la precisión numérica, sino también porque determina la complejidad del sistema y los recursos lógicos consumidos. En el Estado del Arte se demostró que con el formato de punto flotante se obtiene una mayor precisión numérica, sin embargo, con la representación de punto fijo se logra mayor eficiencia de recursos, por lo que su aplicación es de gran utilidad para sistemas embebidos con recursos limitados.

Debido a que la prioridad de este trabajo es el ahorro de los recursos disponibles, se decide emplear un formato numérico de punto fijo de 8 bits Q_7, cuya estructura se ilustra en la Figura 5.3.

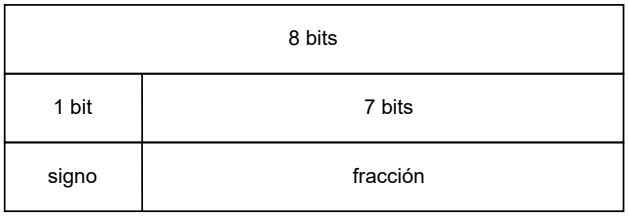


Figura 5.3: Formato numérico de 8 bits Q_7

Se decide trabajar con un sistema numérico signado, por lo que los números negativos son representados con su complemento a 2. Tal como se muestra en la Figura 5.3, se cuenta con 1 bit de signo y 7 bits que conforman la porción fraccionaria del dato. Los pesos sinápticos, umbrales y entradas emplean este formato, por lo que su rango es de $-1 < x < 1$.

Es importante considerar que durante la suma de productos, es posible obtener resultados que superen el rango delimitado por el formato de 8 bits Q7, por lo que es necesario emplear un formato numérico adicional. Los datos que ingresan al bloque de multiplicación (pesos, umbrales y entradas), serán de 8 bits Q_7, sin embargo, una vez que han sido multiplicados el resultado será de 12 bits Q_8. Este formato se representa mediante la Figura 5.4.

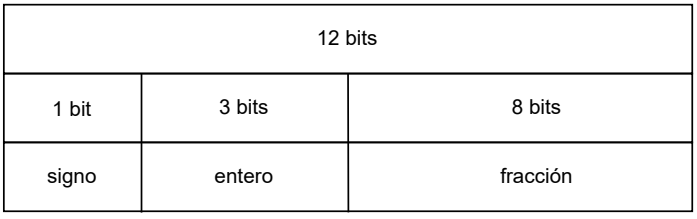


Figura 5.4: Formato numérico de 12 bits Q_8

Se ha decidido emplear este formato debido a que 3 bits son suficientes para representar los valores obtenidos después de la suma de productos. Como se verá más adelante, cuando el dato de entrada de la sigmoide es mayor o igual a 5, la salida de la función es 1. Por lo tanto, aunque se presente una condición de *overflow* durante la suma de productos, esto puede

aprovecharse para asignar directamente el número 1 como respuesta de la función sigmoide. Por este motivo 3 bits son suficientes para representar la parte entera de los datos, lo cual genera un ahorro de recursos.

5.2.3. Receptor de parámetros de entrenamiento, configuración y entradas

Este bloque permite la conexión entre la unidad de entrenamiento y la unidad de clasificación, ilustrada en la Figura 5.5, la cual se realiza a través del protocolo de comunicación UART. Por parte de la unidad de entrenamiento se emplea el puerto USB de la computadora, al cual se tiene acceso mediante el mismo código de Python. En la unidad de clasificación, dentro del FPGA, se implementó un módulo de interfaz UART que recibe los parámetros de entrenamiento y configuración, además de las entradas de la red cuando el sistema se encuentra listo para clasificar.

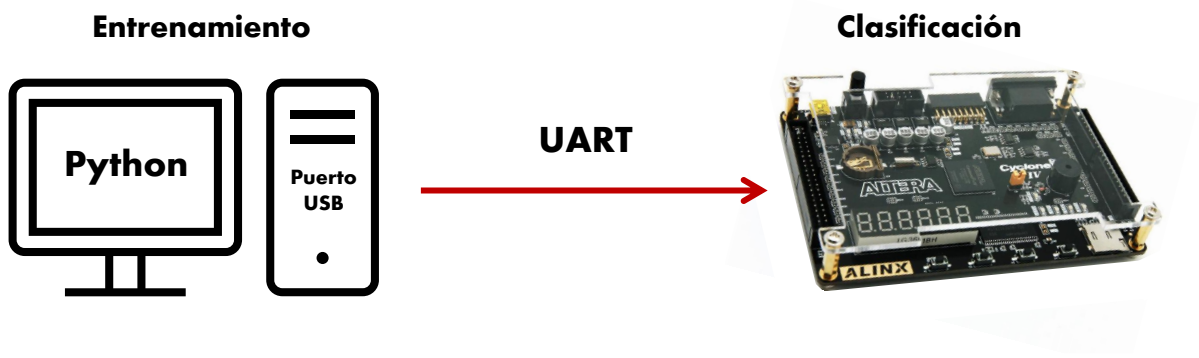


Figura 5.5: Conexión entre la unidad de entrenamiento y la unidad de clasificación

5.2.4. Selector de entradas

Este selector es el elemento encargado de alimentar a las neuronas con los datos de entrada correspondientes, los cuales dependen de la capa que esté por ejecutarse. Las entradas de las neuronas serán los datos provenientes del exterior, únicamente cuando se van a efectuar los cálculos de la primera capa. Por otro lado, cuando se ejecuten las capas posteriores, se seleccionarán como datos de entrada las respuestas de la capa anterior, los cuales provienen del bloque de la función de activación. Este selector se describe en hardware como un multiplexor, el cual cuenta con dos entradas, una salida y una línea de selección. En la Figura 5.6 se indica su conexión con los demás bloques de la red, mientras que en la Figura 5.7 se muestra el nombre y tamaño en bits de las señales.

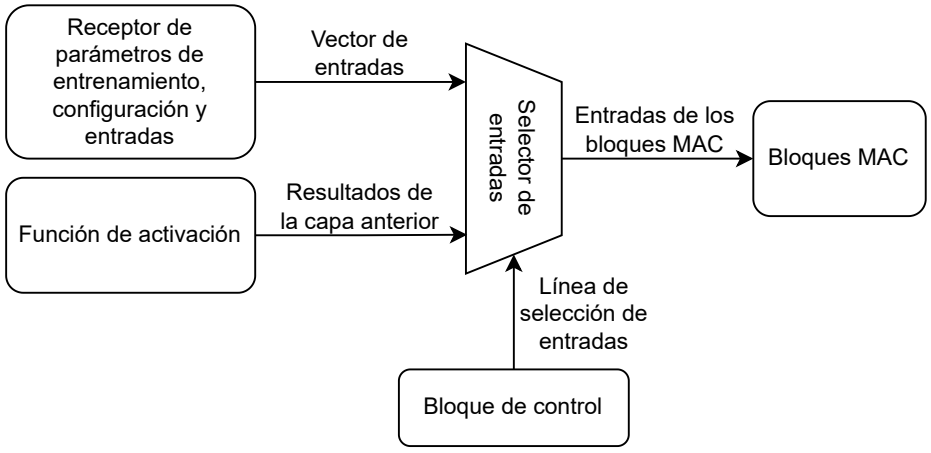


Figura 5.6: Diagrama de bloques del selector de entradas

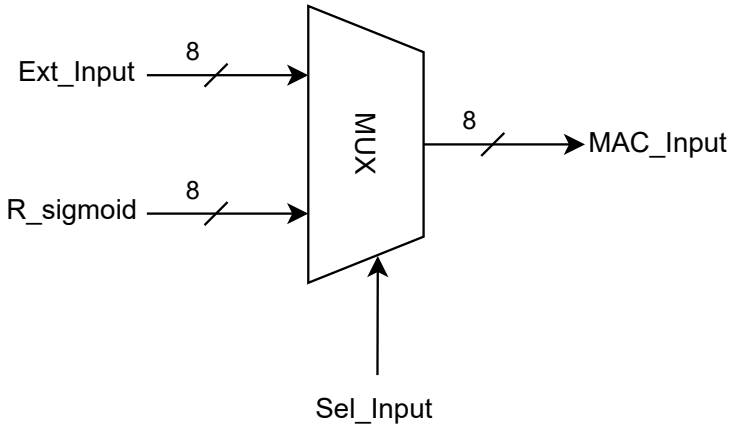


Figura 5.7: Diseño de detalle del selector de entradas

5.2.5. Multiplicación y acumulación (MAC)

Uno de los elementos que componen a las neuronas son los bloques MAC. Del análisis realizado en la sección 4.3.2, se definió que cada neurona de la capa realiza una multiplicación y una suma en un ciclo de reloj. Para llevar este método a cabo, se proponen los siguientes elementos: un bloque de memoria de acceso aleatorio (BRAM, por sus siglas en inglés), un multiplicador, un sumador y un registro acumulador. En la Figura 5.8 se ilustra el proceso de multiplicación y acumulación, además de su interacción con los demás bloques de la unidad.

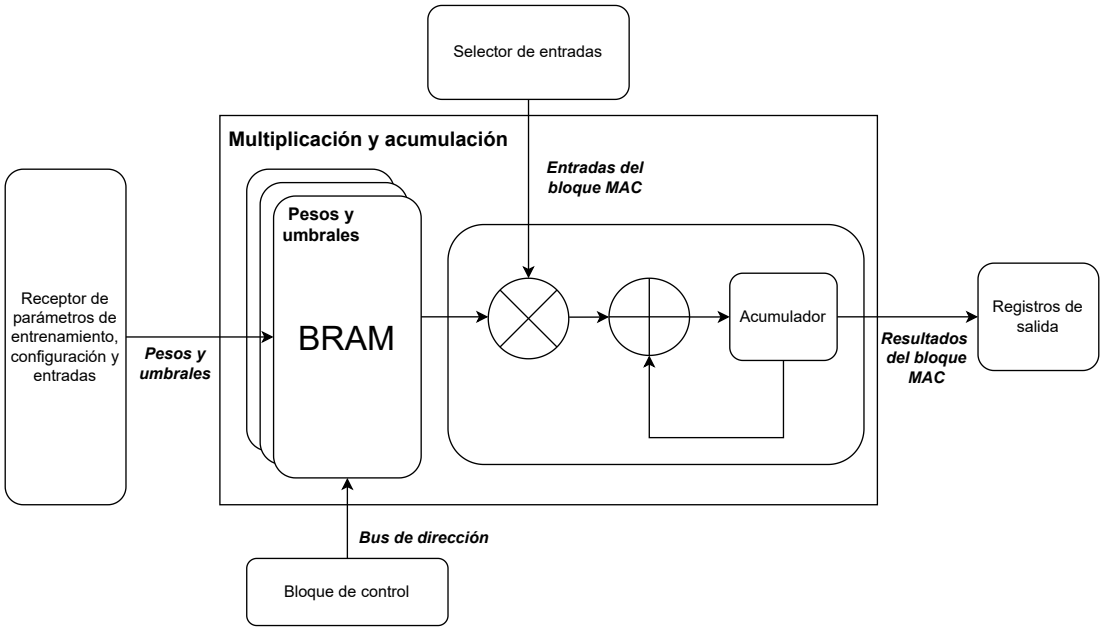


Figura 5.8: Diagrama de bloques del proceso de multiplicación y acumulación

BRAM

Tal como se observó en la Figura 3.2 y la Figura 3.3 del Estado del Arte, existe un gran consumo de energía durante el acceso a memorias como las SRAM o DRAM, además de que se requiere hardware adicional para establecer comunicación con ellas. Por lo tanto, en este diseño se decide aprovechar los recursos del FPGA, almacenando los pesos sinápticos y umbrales en su memoria embebida. Es importante recordar que uno de los métodos de optimización empleados, es reutilizar las neuronas descritas para cada capa, por lo que un mismo BRAM puede contener los pesos y umbrales de más de una capa.

Multiplicador, sumador y acumulador

El multiplicador realiza el producto de los pesos y las entradas, recibiéndolos del BRAM y del selector de entradas, respectivamente. Dicho producto se suma sucesivamente con el valor almacenado en el acumulador, hasta obtener el resultado final.

La multiplicación es una operación que ha demostrado ser demandante en cuanto a recursos y energía, lo cual puede limitar la capacidad de la red neuronal implementada. Sin embargo, en la tabla de especificaciones del FPGA seleccionado, observamos que el dispositivo cuenta con 30 multiplicadores embebidos de 9 bits, los cuales ofrecen la opción de configurar el tamaño de los datos que serán multiplicados. La ventaja de estos multiplicadores es que pueden utilizarse sin consumir de los 6272 elementos lógicos disponibles en el FPGA. Debido a que los pesos y entradas son de 8 bits, el número de multiplicadores disponibles nos permite generar 30 neuronas con estos elementos. Emplear los multiplicadores embebidos representa una gran opción para el ahorro de recursos en FPGA de elementos lógicos limitados.

En la Figura 5.9 se muestra con detalle la estructura de un bloque MAC, que cuenta con un BRAM, un multiplicador embebido configurado para recibir datos de 8 bits, un sumador de 12 bits y un acumulador de 12 bits, el cual es un registro conformado por 12 *flip-flop* D (Figura 5.10).

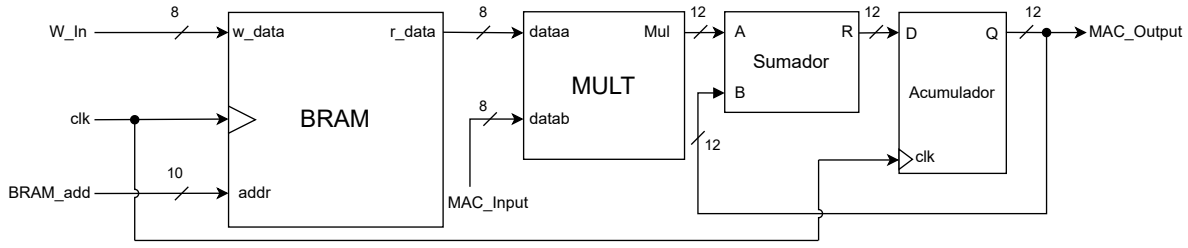


Figura 5.9: Diseño de detalle de los bloques MAC

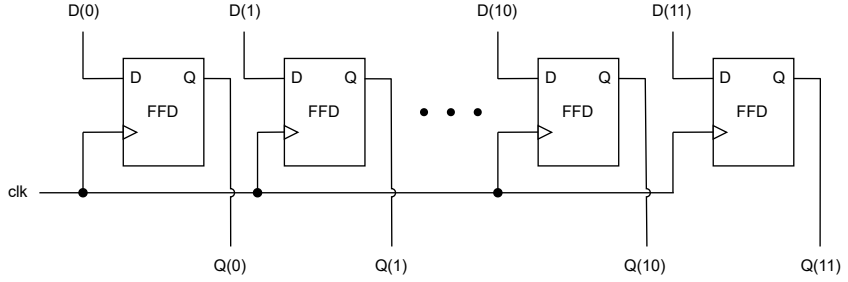


Figura 5.10: Estructura de un registro de 12 bits

5.2.6. Registros de salida

La estructura de los registros que almacenan las salidas de los bloques MAC, juegan un papel sumamente importante en cuanto a la arquitectura de la FFNN. Como se demostró anteriormente, la metodología seleccionada para la multiplicación de la matriz de pesos y el vector de entradas, implica que las neuronas de una capa reciben los datos de entrada de forma serial. Esto aplica para los datos provenientes del exterior, así como para los resultados de una capa anterior. Por lo tanto, los registros de salida deben de contar con una opción de corrimiento, la cual permita ingresar serialmente los datos a la función sigmoide y posteriormente a la siguiente capa. En la Figura 5.11 se muestran los registros de salida y sus conexiones con los demás bloques, mientras que en la Figura 5.12 se ilustra su arquitectura interna, además del nombre y tamaño de las señales.

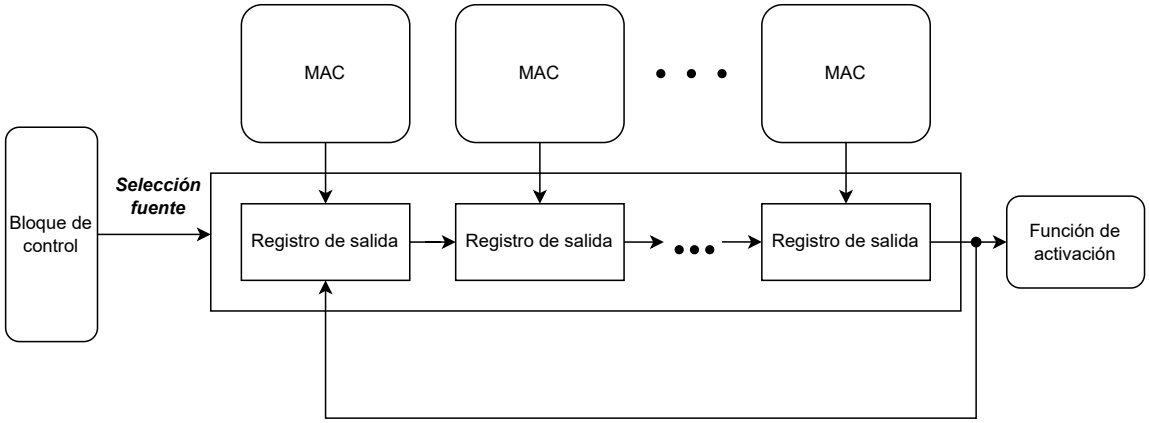


Figura 5.11: Diagrama de bloques de los registros de salida

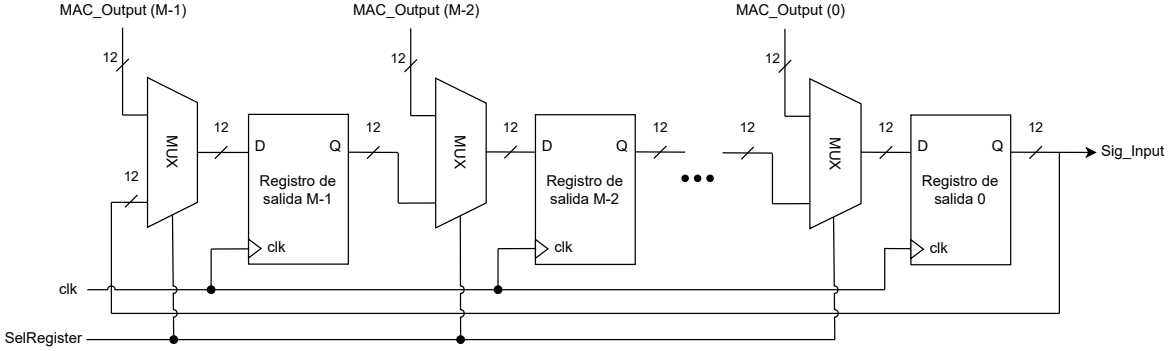


Figura 5.12: Diseño de detalle de los registros de salida

Los registros de salida reciben en paralelo los resultados de los bloques MAC, para posteriormente recorrer los datos de izquierda a derecha. Cuando comienza el desplazamiento, el dato almacenado en el *registro de salida 0*, será el primero en ingresar al bloque de la función sigmoide por medio de la señal *Sig_Input*. El desplazamiento continúa, hasta que se hayan procesado las sumas ponderadas de todas las neuronas. Para llevar esto a cabo, se conecta un multiplexor en la entrada de cada registro y el dato que será cargado, depende del valor de la línea de selección *SelRegister*, la cual proviene del bloque de control.

En la Tabla 5.3 se muestra el valor de *SelRegister* y el dato que será almacenado en los registros de salida. La tabla nos indica que cuando la línea de selección sea 0, se les cargará el dato del registro que se encuentra a su izquierda, funcionando como un corrimiento a la derecha. En el caso en donde la línea de selección es 1, se les cargará el valor del bloque MAC correspondiente.

5.2.7. Función de activación

Debido a que desde un inicio se definió el procesamiento serial de las entradas, los resultados de las neuronas de una capa pueden procesarse una a una, a través del bloque de la

Tabla 5.3: Entrada de los registros de salida según *SelRegister*

SelRegister	Entrada del registro	Nombre de la señal
0	Valor del registro anterior	Q
1	Valor del acumulador	*MAC_Output*

función de activación. El implementar un solo bloque de la función de activación, en vez de uno por neurona, representa un ahorro de recursos considerable. La función de activación seleccionada para este trabajo es la *función sigmoide*, debido a las diversas ventajas que ofrece para el entrenamiento y prueba de las ANN, las cuales se explicaron en el Marco Teórico.

De los métodos estudiados en el Estado del Arte, se demostró que la aproximación lineal por partes es una de las opciones que mejores resultados ha brindado en cuanto a flexibilidad, ya que ajustando los segmentos es como se puede realizar un intercambio entre precisión y uso de recursos. El número de segmentos implementados depende de las prioridades planteadas para la aplicación objetivo, ya que existen ocasiones en donde se requiere una alta precisión y se cuenta con dispositivos de gran capacidad, lo que permite implementar mayor cantidad de segmentos. Por otro lado, existen casos en donde se busca el mayor ahorro de recursos y la misma aplicación permite la disminución de precisión sin afectar el rendimiento del sistema. En este trabajo se plantea el segundo caso, por lo que se selecciona la aproximación de 4 segmentos debido a que su implementación ha demostrado un ahorro sustancial de recursos y una precisión suficiente para obtener los resultados deseados. En la ecuación (5.1) se define la función sigmoide mediante la aproximación de 4 segmentos.

$$\sigma(x) = \begin{cases} 1 & 5.0 \leq x \\ 0.03125x + 0.84375 & 2.375 \leq x < 5.0 \\ 0.125x + 0.625 & 1.0 \leq x < 2.375 \\ 0.25x + 0.5 & 0 \leq x < 1.0 \end{cases} \tag{5.1}$$

Seleccionando cuidadosamente los cortes entre cada uno de los segmentos, es posible obtener expresiones que sean eficientes en hardware. En (5.2) se muestra que las multiplicaciones en la ecuación (5.1) se pueden reemplazar por operaciones de corrimiento de n bits, dependiendo del valor de x.

$$\sigma(x) = \begin{cases} 1 & 5.0 \leq x \\ (x \gg 5) + 0.84375 & 2.375 \leq x < 5.0 \\ (x \gg 3) + 0.625 & 1.0 \leq x < 2.375 \\ (x \gg 2) + 0.5 & 0 \leq x < 1.0 \end{cases} \tag{5.2}$$

Mediante las ecuaciones previas, es posible observar que los segmentos aplican únicamente para valores positivos de x. Esto es debido a que la función sigmoide es simétrica cruzando el origen, por lo que el valor de la función para $x < 0$ se obtiene mediante la ecuación (5.3).

$$\sigma(x) = 1 - \sigma(-x) \tag{5.3}$$

En la Figura 5.13 se ilustra el diagrama de bloques con las entradas, salidas y conexiones de la función de activación con los demás bloques.

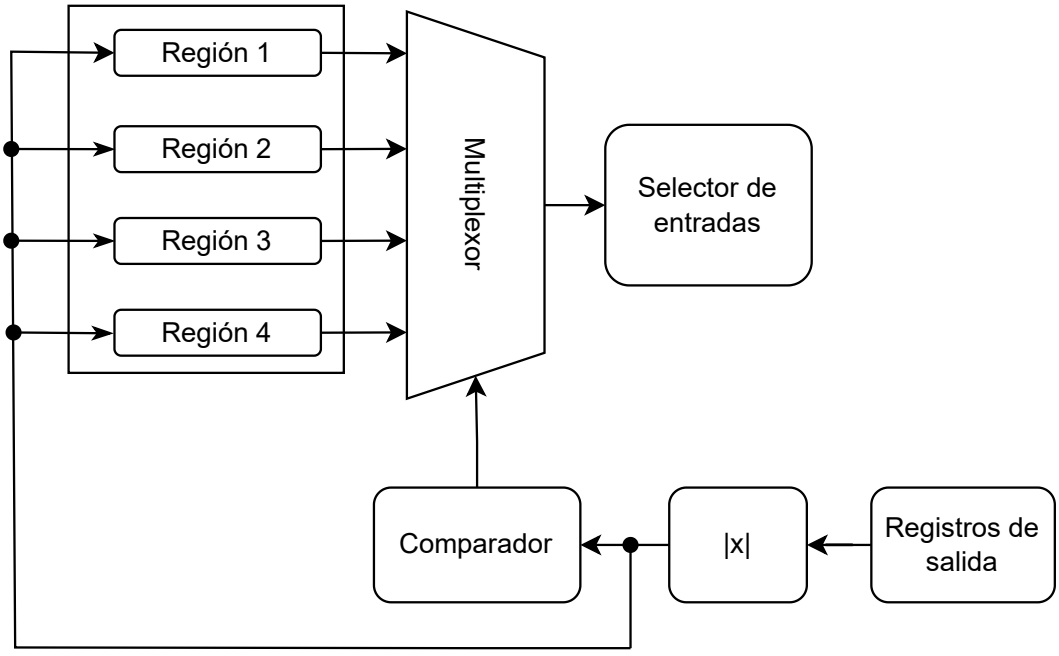


Figura 5.13: Diagrama de bloques de la función de activación

El dato que procesa el bloque de la función de activación, es la suma de productos que contiene el registro de salida menos significativo. A continuación, se describen cada uno de los bloques necesarios para llevar a cabo la implementación de la función de activación sigmoide.

Valor absoluto

Como se explicó con anterioridad, la función de activación sigmoide es simétrica cruzando el origen, por lo que independientemente de si el valor de x es positivo o negativo, se emplea la aproximación mostrada en la ecuación (5.2). Por lo tanto, el primer paso es obtener el valor absoluto de x mediante el bloque mostrado en la Figura 5.14.

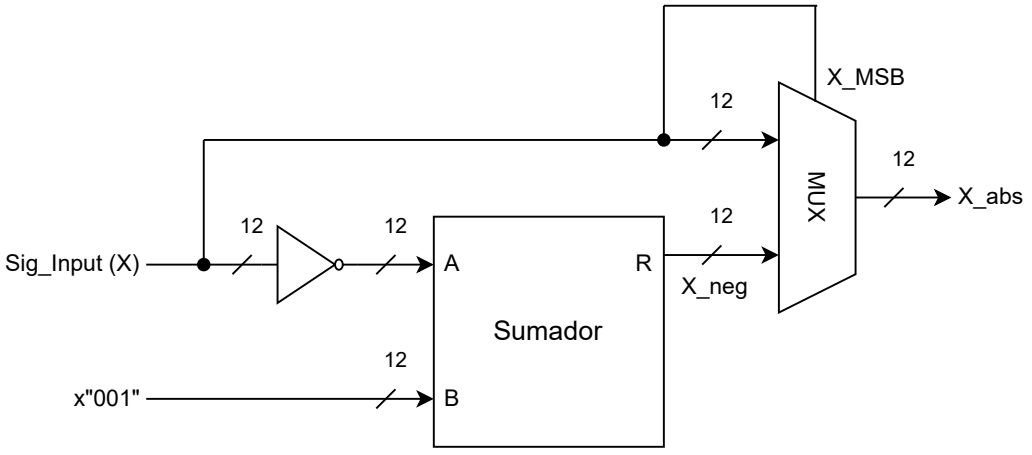


Figura 5.14: Diseño de detalle del bloque de valor absoluto

Este bloque calcula el complemento a 2 de x invirtiendo sus bits y posteriormente incrementándole uno, mediante un sumador de 12 bits. Cuando el bit más significativo (MSB, por sus siglas en inglés) de x es igual a 1, indica que el dato es negativo. Esto se aprovecha como línea de selección para el multiplexor de salida, ya que si el MSB equivale a 1, entonces la salida del bloque es el complemento a 2 (X_neg). Por otro lado, cuando el MSB es igual a 0, indica que x es positivo y pasa directamente a la salida. En la Tabla 5.4 se muestra la salida del multiplexor por cada valor de la línea de selección.

Tabla 5.4: Salida del multiplexor según X_MSB

X_MSB	Salida del multiplexor	Nombre de la señal
0	Directamente x	x
1	El complemento a 2 de x	X_neg

Comparador

Este bloque se encarga de generar las líneas de selección del multiplexor que determina la región a la que pertenece x. La comparación se realiza mediante 3 sumadores, en donde se resta x con los límites de las regiones. En la Tabla 5.5 se muestran 3 representaciones distintas de estas constantes en 12 bits Q8, las cuales son el valor en decimal, en hexadecimal y el complemento a 2 en hexadecimal, siendo esta última representación la que se utilizará para realizar las restas. Cada uno de estos valores en complemento a 2 se encuentran almacenados como constantes en el FPGA. El bloque diseñado como comparador se muestra en la Figura 5.15.

Tabla 5.5: Límites de las regiones

Decimal	Hexa Q8	C2 Hexa Q8
5.0	500	B00
2.375	260	DA0
1.0	100	F00

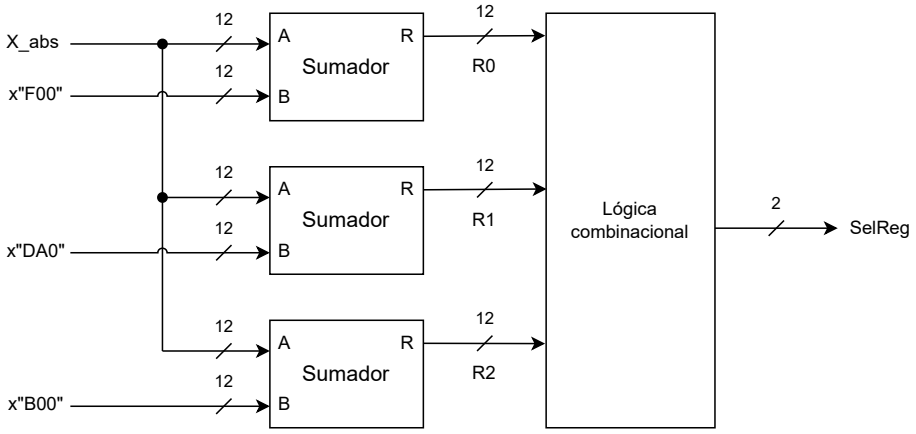


Figura 5.15: Diseño de detalle del comparador

El resultado de las restas puede ser positivo, negativo o cero, dependiendo de si x es mayor, menor o igual a la constante, respectivamente. Por lo tanto, el bit más significativo de cada resultado $R0$, $R1$ *y* $R2$, sirve para determinar la región a la que pertenece x. Se realizó la tabla de verdad mostrada en la Figura 5.16, la cual recibe como datos de entrada los bits más significativos de cada resultado y las salidas son $SelReg(0)$ y $SelReg(1)$. En este caso se emplean 2 bits como líneas de selección, ya que se debe de elegir entre cuatro regiones.

R2	R1	R0	SelReg(1)	SelReg(0)
0	0	0	0	0
0	0	1	*	*
0	1	0	*	*
0	1	1	*	*
1	0	0	0	1
1	0	1	*	*
1	1	0	1	0
1	1	1	1	1

Figura 5.16: Tabla de verdad de las líneas de selección *SelReg*

Resolviendo la tabla de verdad se obtiene la lógica combinacional, mostrada en la Figura 5.17, que genera las señales *SelReg*(1) y *SelReg*(0) del multiplexor.

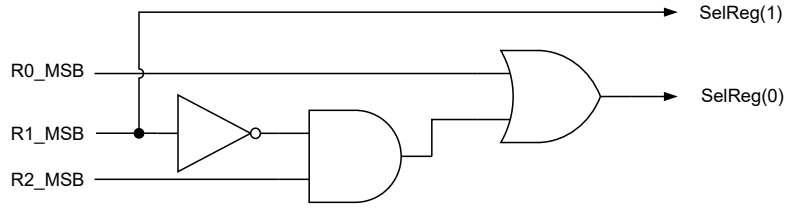


Figura 5.17: Circuito combinacional de las señales *SelReg*(1) y *SelReg*(0)

En la Tabla 5.6 se muestra la región seleccionada y la operación realizada, dependiendo del valor que adquiera la señal *SelReg*.

Tabla 5.6: Región seleccionada según *SelReg*

SelReg	Región seleccionada	
0 0	Región 1	1
0 1	Región 2	$(x \gg 5) + 0.84375$
1 0	Región 3	$(x \gg 3) + 0.625$
1 1	Región 4	$(x \gg 2) + 0.5$

Cálculo de regiones

En las regiones 2, 3 y 4 se realiza un procedimiento similar, diferenciándose únicamente por el número de bits recorridos y la constante que se suma. Para llevar a cabo estas operaciones, primeramente se cuenta con un registro de corrimiento implementado con multiplexores, el cual se muestra en la Figura 5.18. Para los multiplexores de este registro también se emplean las señales de selección *SelReg*(1) y *SelReg*(0). En la Tabla 5.7 se muestra el valor resultante en el registro, dependiendo del valor de estas dos líneas de selección.

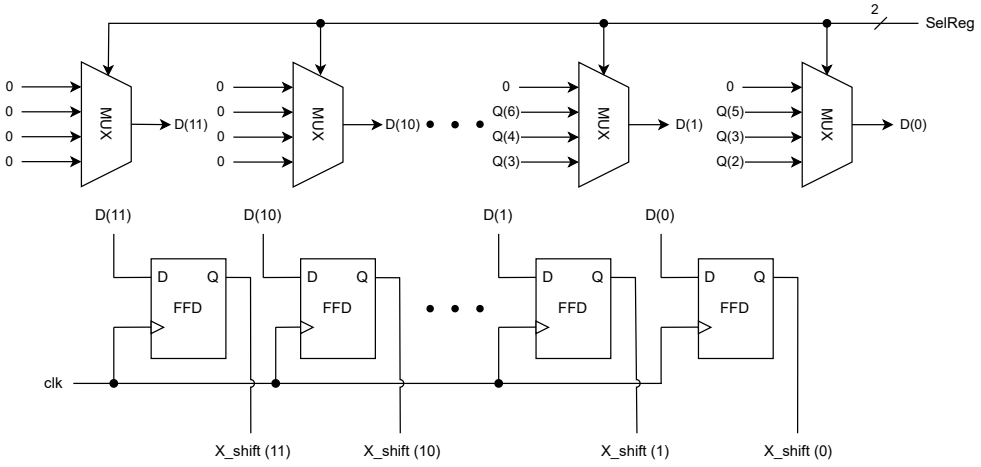


Figura 5.18: Diseño de detalle del registro de corrimiento

Tabla 5.7: Valor de corrimiento según *SelReg*

SelReg	Valor que se carga al registro
0 0	1
0 1	$x \gg 5$
1 0	$x \gg 3$
1 1	$x \gg 2$

La estructura planteada con multiplexores permite que el corrimiento de x se realice en un solo ciclo de reloj, independientemente de la región. Esto es debido a que el multiplexor carga directamente el bit correspondiente, dependiendo de la región identificada. Es importante notar que para la primera región el valor de la sigmoide es directamente 1, por lo que el multiplexor cargará este valor con el formato numérico de 12 bits en Q8, el cual se representa en hexadecimal como $0x100$.

Una vez que se ha realizado el corrimiento, se suma la constante correspondiente a cada región, las cuales se muestran en la Tabla 5.8. La selección de esta constante se lleva a cabo mediante un multiplexor que tiene como líneas de selección a $SelReg(1)$ y $SelReg(0)$. Al igual que sucede con los límites de cada región, estos valores se almacenan como constantes en el FPGA.

Tabla 5.8: Constantes sumadas según *SelReg*

SelReg	Valor sumado
0 0	0
0 1	0.84375
1 0	0.625
1 1	0.5

En la Figura 5.19 se muestra el bloque que lleva a cabo la operación de adición con la constante seleccionada. Para el caso de las regiones 2, 3 y 4 se suman sus constantes correspondientes, sin embargo, para la región 1 se suma con cero para pasar directamente el $0x100$ a la salida.

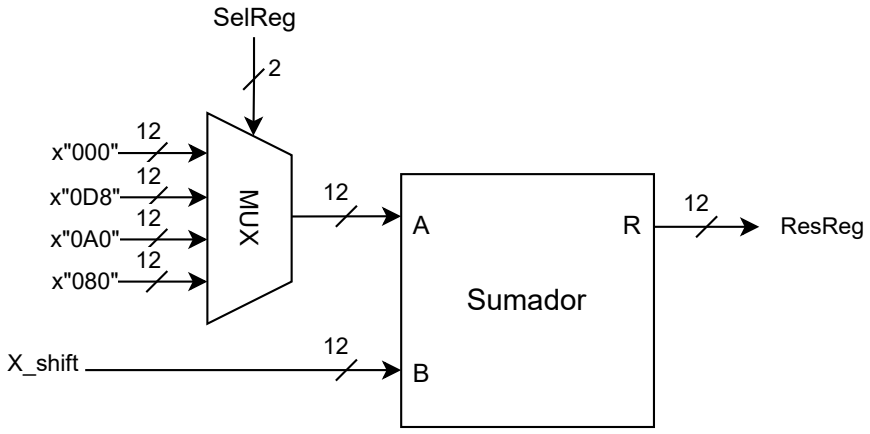


Figura 5.19: Diseño de detalle del sumador de constantes

Selección de la salida

Por último, se mencionó que cuando el valor de x es positivo, la salida es directamente el resultado de la región. Por otro lado, si x resultó ser negativo se lleva a cabo la operación mostrada en la ecuación (5.3). Para realizar esta resta es necesario calcular el complemento a 2 de *ResReg*, para posteriormente restarle a 1 dicho valor. Esto puede realizarse en un solo paso mediante la máquina mostrada en la Figura 5.20.

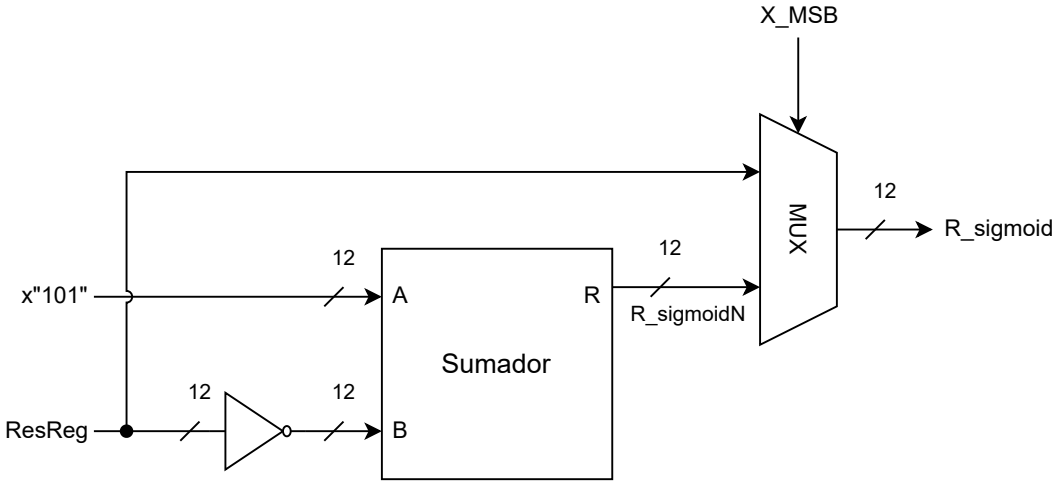


Figura 5.20: Diseño de detalle del selector de salida

El valor $0x101$ es para que se realice el complemento a 2 y se reste con 1, en una sola operación del sumador. La línea de selección del multiplexor de salida es el bit más significativo de x, ya que si es igual a 1 significa que es negativo y el resultado de la sigmoide debe ser $1 - \sigma(x)$, lo cual corresponde a *R_SigmoidN*.

5.2.8. Bloque de control

Una vez detallados cada uno de los elementos que conforman la FFNN, es posible diseñar el bloque de control, el cual determina el funcionamiento de los demás bloques, la interacción entre ellos y la sincronización de los procesos. El diseño de detalle del bloque de control se ilustra en la Figura 5.21, en donde se muestran las conexiones de cada uno de los elementos que lo conforman, así como las señales de control generadas.

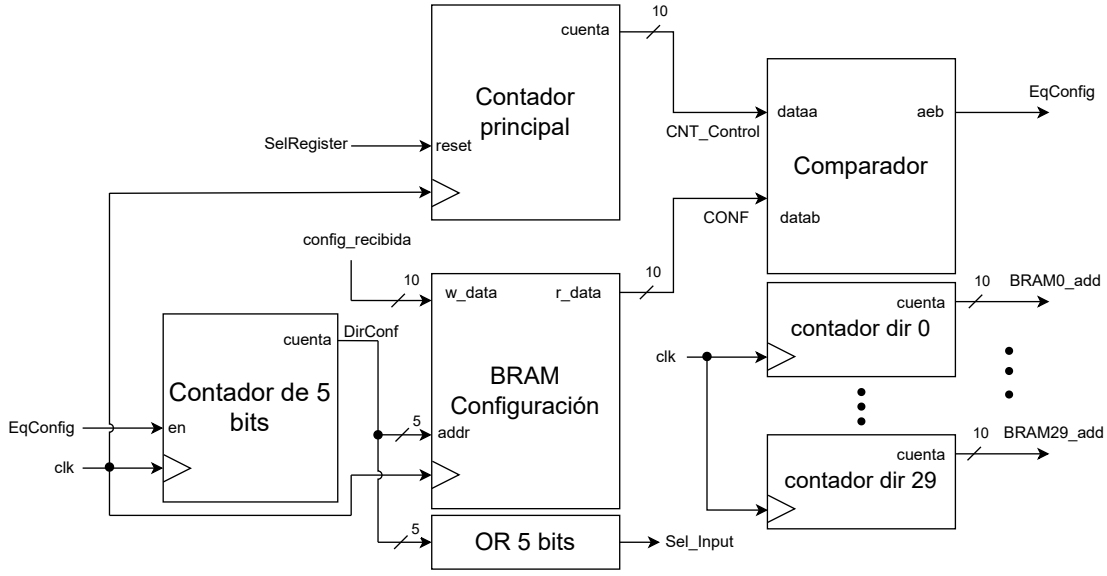


Figura 5.21: Diseño de detalle del bloque de control

La función de cada uno de los elementos y señales de control, se explica con detalle a continuación.

Contador principal

El elemento base del bloque de control es el contador principal de 10 bits mostrado en la Figura 5.21, el cual indica el estado actual del sistema. Dicho contador se reinicia en cada capa con ayuda de la señal de control *SelRegister*, por lo que es necesaria la implementación de un comparador. En este trabajo se pretende probar el desempeño de la FFNN con imágenes conformadas por un total de 784 píxeles, siendo esta la razón por la que se generó un contador de 10 bits.

Bus de dirección de los BRAM

En la Figura 5.21 se observa que en total se cuenta con 30 contadores, los cuales se enumeran desde **contador dir 0**, hasta **contador dir 29**. Estos corresponden al bus de dirección de los BRAM, que contienen los pesos y umbrales de cada uno de los bloques MAC.

Memoria de configuración

La memoria de configuración almacena el número de entradas, número de neuronas en cada capa y el número de salidas. Debido a que en este caso la red neuronal que será implementada tiene 784 datos de entrada y este valor debe almacenarse en la memoria, cada localidad contiene datos de 10 bits. El número de localidades en la memoria de configuración depende del número de capas máximas que se desearía implementar. En este caso se empleó un bus de dirección de 5 bits para tener hasta 32 datos de configuración, lo que permitirá probar diversas FFNN con diferente número de capas y neuronas. El bus de dirección de la memoria de configuración es un contador de 5 bits, el cual debe de aumentar en uno su valor, cada vez que se pasa a otra capa. Esto nos indica que el contador requiere de una línea de habilitación, correspondiente a *EqConfig*, la cual solo debe permitir que el contador aumente en uno cuando se pase a la siguiente capa.

Comparador

Es importante recordar que cada entrada se procesa en 1 ciclo de reloj, obteniendo así los resultados de cada capa en N ciclos, siendo N el número de entradas. Debido a que el número de entradas y neuronas por capa se encuentran almacenados en la memoria de configuración, se debe comparar constantemente el valor de la memoria de configuración, con el valor del contador principal. Cuando el valor del contador sea igual al número de entradas de esa capa, entonces se activan todas las señales que preparan los registros para la siguiente capa. La línea de control generada por el comparador es *EqConfig*.

Línea de control del selector de entradas

El multiplexor de entradas igualmente necesita una señal de control que permita seleccionar entre las entradas externas y las respuestas de la capa anterior. Para ello se añade una operación *or* con los 5 bits del contador, ilustrado en la Figura 5.21, el cual funciona como bus de dirección de la memoria de configuración. Cuando el valor del contador es cero, significa que nos encontramos en la primera capa y los datos de entrada provienen del exterior, por lo que la salida de la operación *or* será *Sel_Input=0* y el multiplexor ingresará los datos externos. Para todas las demás capas el valor del contador es diferente de cero, por lo que la operación *or* generará la señal *Sel_Input=1* y así las entradas serán los resultados de la capa anterior.

Línea de control de los registros de salida

Recordando el funcionamiento de los registros de salida, estos reciben a través de un multiplexor el valor que se les cargará, ya sea el dato del acumulador de su bloque MAC correspondiente o del registro previo para realizar el corrimiento de los datos. Como se vio en la Tabla 5.3, se debe de cargar el valor del acumulador cuando *SelRegister* valga 1, por lo que la línea de selección puede ser un ciclo de reloj después de que se ha cumplido *EqConfig*. Para ello se decidió emplear un *flip-flop* D, como el que se muestra en la Figura 5.22.

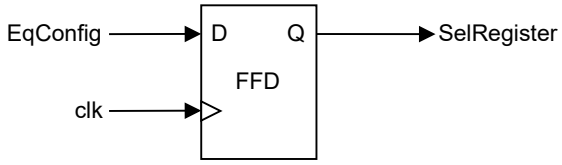


Figura 5.22: Generación de la señal *SelRegister*

Capítulo 6

Implementación

6.1. Implementación en Quartus y RTL viewer

Una vez detallada la funcionalidad, elementos y conexiones de cada bloque del diseño planteado, es posible presentar cómo fue llevada a cabo la FFNN en el FPGA seleccionado. La implementación del sistema fue por medio del entorno de desarrollo integrado (IDE, por sus siglas en inglés) de Intel, *Quartus Prime* y empleando el lenguaje de descripción de hardware VHSIC (*VHDL*, por sus siglas en inglés). El entorno de Quartus cuenta con diversas herramientas, dentro de las cuales se encuentra un visualizador del hardware sintetizado a nivel de transferencia de registro (RTL, por sus siglas en inglés). Esta herramienta, nombrada *RTL viewer*, permite observar con diversos niveles de abstracción los bloques del sistema. En la Figura 6.1 se muestra el bloque de control, con sus entradas y salidas.

Figura 6.1: Bloque de control en RTL viewer

Dentro de la misma herramienta es posible expandir el bloque y mostrar cada uno de los elementos y conexiones internas del bloque de control, tal como se observa en la Figura 6.2.

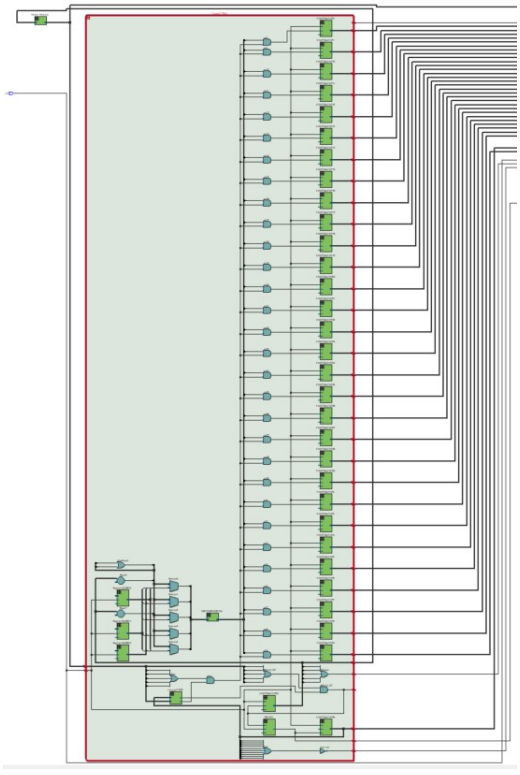


Figura 6.2: Estructura interna del bloque de control en RTL viewer

De igual forma, es posible visualizar en la Figura 6.3 los BRAM que almacenan los pesos y umbrales, los bloques que contienen a los multiplicadores y sumadores, así como sus registros de salida.

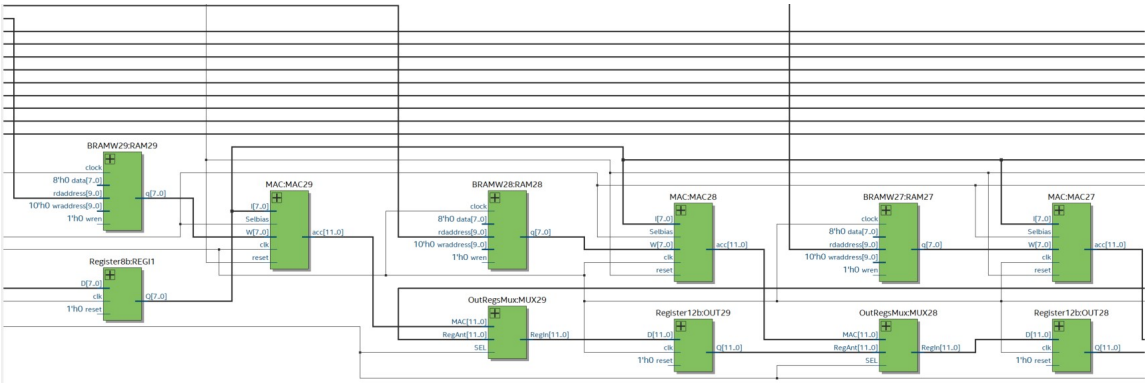


Figura 6.3: Bloques MAC y registros de salida en RTL viewer

A su vez, la estructura interna de los bloques MAC se observa en la Figura 6.4, en donde se incluyen los elementos principales como el multiplicador, el sumador y el registro acumulador. También es importante notar la lógica combinacional generada, encargada de cargar al acumulador el umbral de la neurona.

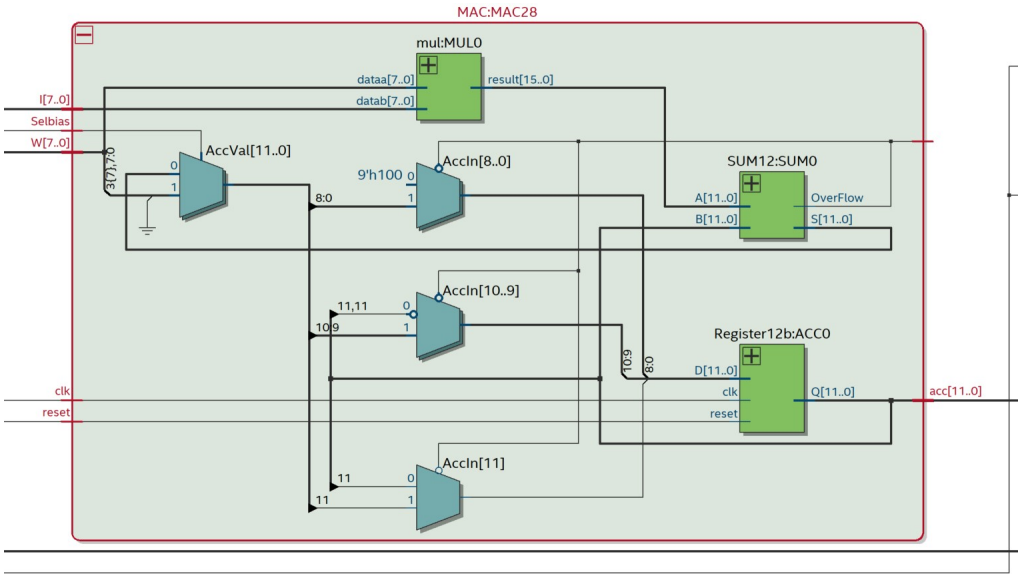


Figura 6.4: Estructura interna del bloque MAC en RTL viewer

Estos bloques se repiten para cada una de las 30 neuronas descritas, por lo que se considera suficiente mostrar la estructura de una de ellas. En la Figura 6.5 se encuentra la estructura de la función sigmoide implementada, la cual también incluye cada uno de los elementos descritos en el diseño de detalle.

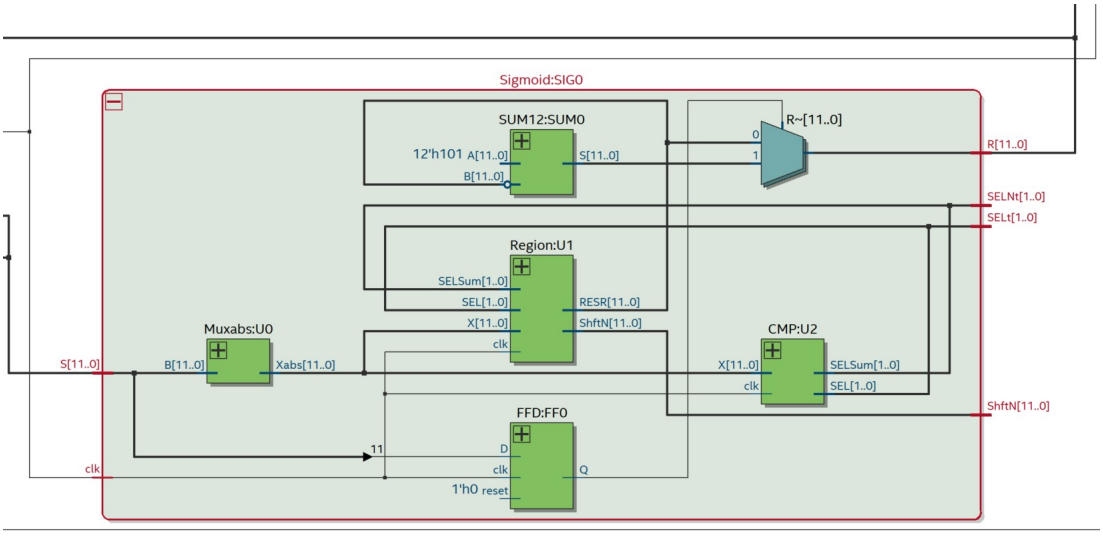


Figura 6.5: Estructura interna de la función sigmoide en RTL viewer

Otro de los elementos, mostrado en la Figura 6.6, es el bloque encargado de identificar

el valor de salida de mayor magnitud, lo cual define la clase a la que pertenece el vector de entrada.

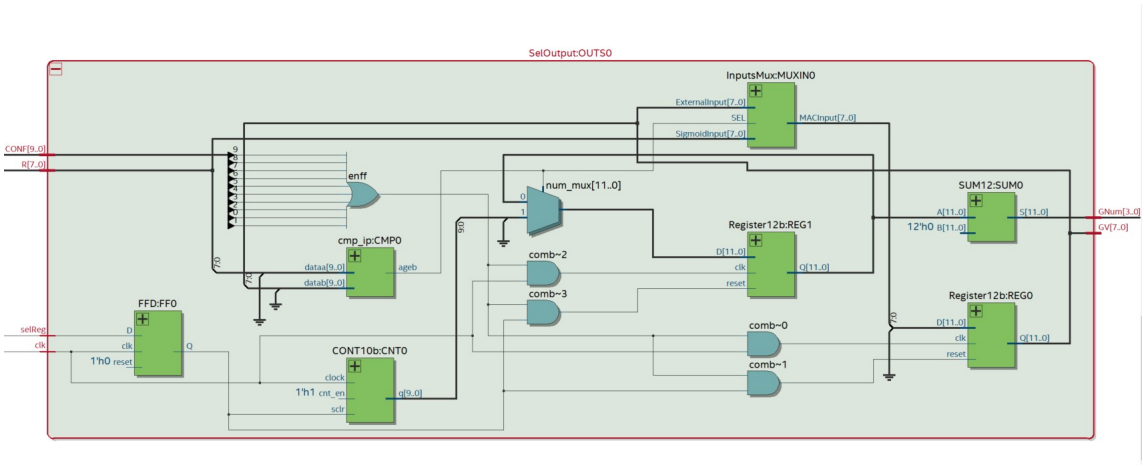


Figura 6.6: Estructura interna del bloque selector en la salida en RTL viewer

Por cuestiones de visualización en la tarjeta, se añadió un bloque, mostrado en la Figura 6.7, el cual permite observar directamente el resultado que arrojó la FFNN. Este bloque consiste en un decodificador BCD a 7 segmentos, por lo que el número clasificado se mostrará en uno de los módulos de 7 segmentos que posee la tarjeta de desarrollo.

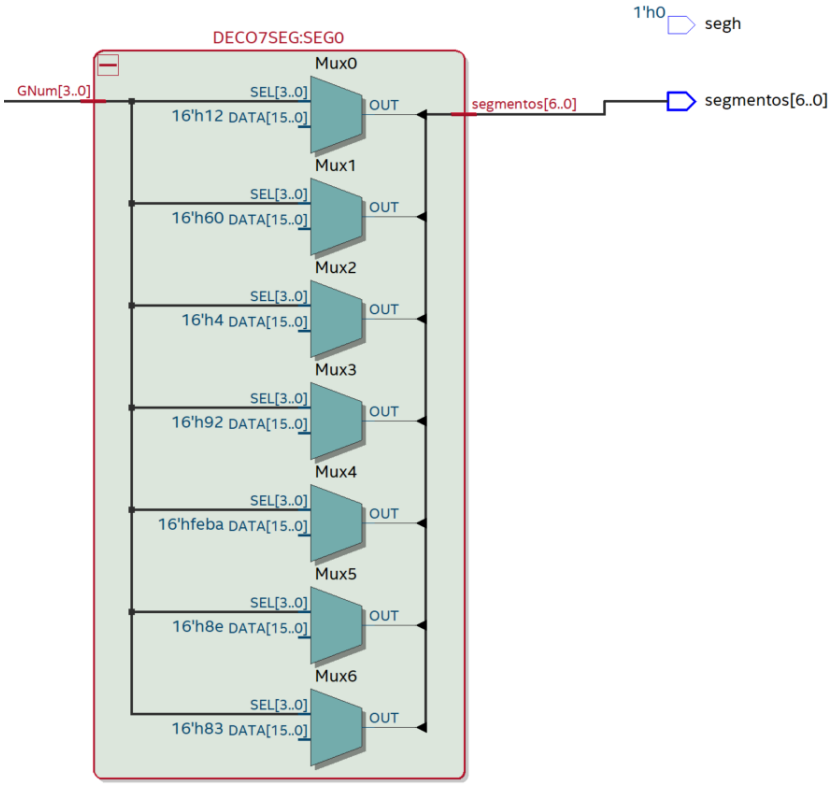


Figura 6.7: Decodificador BCD a 7 segmentos en RTL viewer

6.2. Implementación en el FPGA

La tarjeta de desarrollo en la cual se probó el desempeño de la FFNN y la asignación de pines correspondiente, se muestran en la Figura 6.8 y la Figura 6.9, respectivamente.

Figura 6.8: Tarjeta de desarrollo empleada

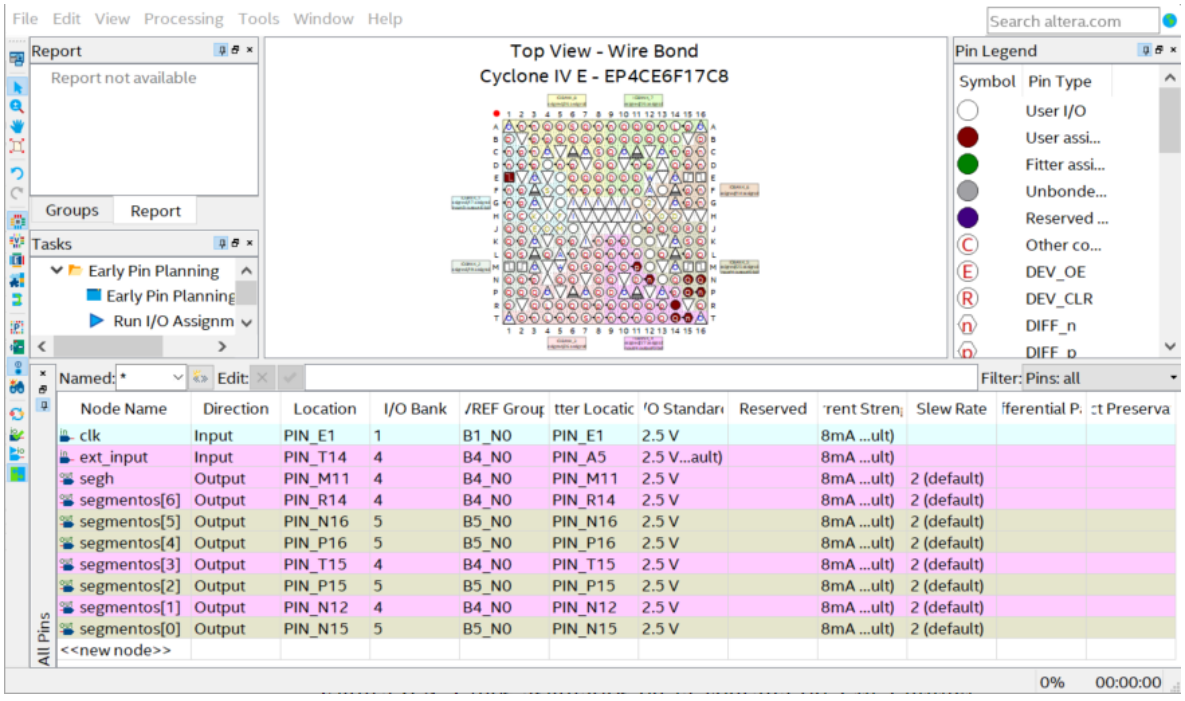


Figura 6.9: Pines asignados en la ventana de *Pin Planner*

En la Tabla 6.1 se muestran los pines de entrada y de salida, con su respectiva función.

Tabla 6.1: Asignación de pines

Tipo	PIN	Descripción
Entradas	PIN_E1	reloj de 50MHz
	PIN_T14	recibe entradas, pesos, umbrales y configuración
Salidas	PIN_M11	habilitador del 7 segmentos
	PIN_N15	enciende segmento a
	PIN_N12	enciende segmento b
	PIN_P15	enciende segmento c
	PIN_T15	enciende segmento d
	PIN_P16	enciende segmento e
	PIN_N16	enciende segmento f
	PIN_R14	enciende segmento g

Capítulo 7

Pruebas y resultados

La FFNN desarrollada fue sometida a pruebas de desempeño, tanto en simulación como implementada directamente en el FPGA. Como se mencionó desde la introducción, las pruebas de desempeño se realizan con un conjunto de imágenes de la base de datos MNIST, que representan dígitos escritos a mano. Se tienen 10 clases distintas, es decir, una por cada dígito. La FFNN recibe como entradas los 784 píxeles de una imagen y la capa de salida se conforma por 10 neuronas, en donde cada una representa un dígito.

En la Figura 7.1 se muestra una captura de la simulación generada mediante la herramienta *University Program VWF*, en donde se seleccionó como ejemplo la imagen de un 5 escrito a mano, mostrado en la Figura 7.2.

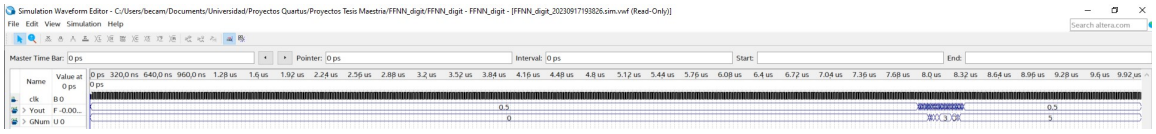


Figura 7.1: Tiempo completo de simulación

Figura 7.2: Imagen de prueba del dígito 5

La simulación cuenta con 3 señales incluyendo al reloj maestro, las cuales permiten visualizar los resultados tanto intermedios como finales de la FFNN. La señal *Yout* representa los resultados de cada neurona que arroja el bloque de la función sigmoide, mientras que la señal *GNum* es el número reconocido por la red. En la primera porción de la simulación no se observa cambio alguno en las señales, ya que es cuando se realiza el procesamiento de los 784 datos de entrada. En la Figura 7.3 se muestra una captura de la porción de la simulación en donde ya es posible observar los resultados intermedios y finales.

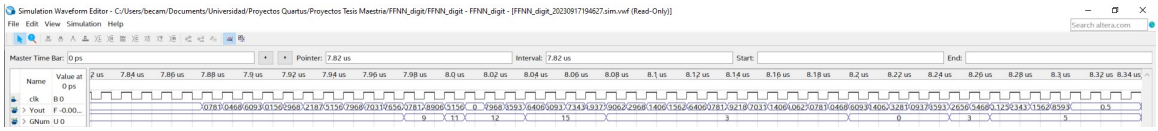


Figura 7.3: Resultados de la simulación

Gracias a la herramienta de simulación se logra determinar que se consumen en total 831 ciclos de reloj, para obtener la clasificación final. Por lo que conociendo que se cuenta con un reloj de *50MHz*, es decir, un periodo de *20ns*, el diseño realizado toma 16μs en realizar una clasificación.

Otro aspecto importante es el consumo energético del sistema, el cual mediante el *Power Analyzer Tool* que brinda *Quartus*, es posible estimar en el FPGA seleccionado. Este resultado se muestra en la Figura 7.4.

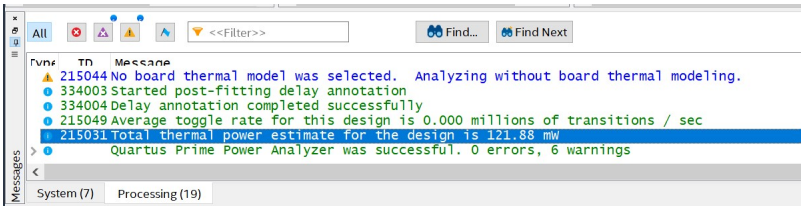


Figura 7.4: Consumo energético estimado en el FPGA

Para poder visualizar rápidamente el resultado en el FPGA, se muestra en el display de 7 segmentos el dígito que clasificó la FFNN. En la Figura 7.5 se muestra un ejemplo de esta visualización en la tarjeta de desarrollo, empleando como entrada la imagen del dígito 5. Esto mismo sucede con las demás imágenes que contienen otros dígitos.

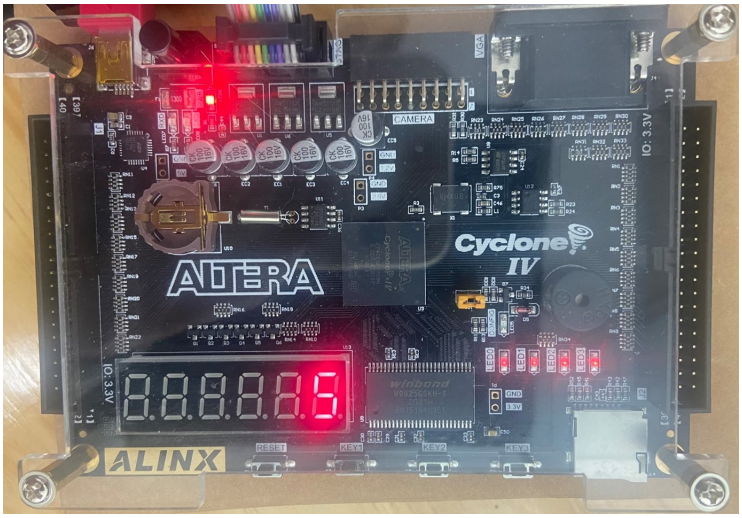


Figura 7.5: Número 5 clasificado por el FPGA

En la Figura 7.6 se muestra la matriz de confusión obtenida a partir de los resultados, tanto de la FFNN en Python como la implementada en FPGA. En este caso se emplearon 500 imágenes de prueba por cada dígito escrito a mano.

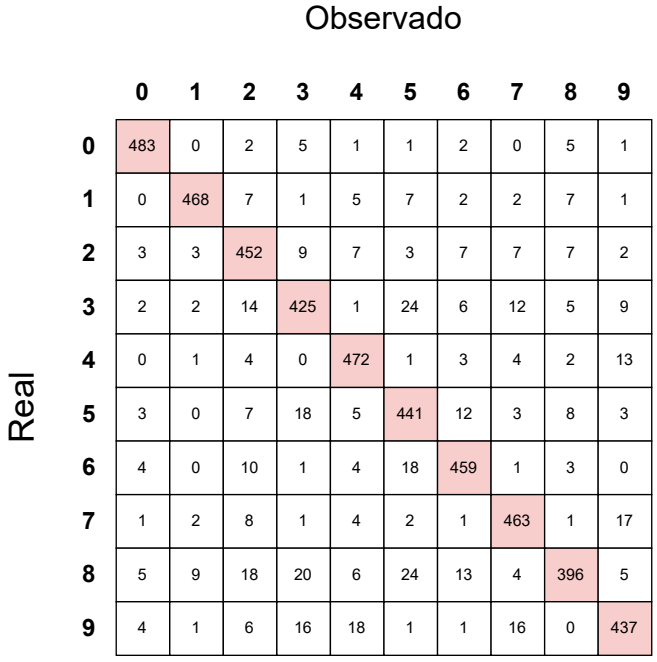


Figura 7.6: Matriz de confusión de las imágenes clasificadas

En la Tabla 7.1 se realiza la comparación de los parámetros más relevantes de la FFNN implementada en Python y la descrita en FPGA, en donde es posible observar que el desempeño del FPGA superó el rendimiento del CPU, tomando en cuenta la velocidad de procesamiento y el consumo energético, además de que conservó la precisión de clasificación.

Tabla 7.1: Comparación de FFNN en Python y FPGA

Parámetro	Python (CPU)	FPGA
Frecuencia del reloj	$1.80GHz$	$50MHz$
Tiempo de ejecución	$1.3s$	$16\mu s$
Elementos lógicos usados	-	2244 (36 %)
Precisión de clasificación	89.92 %	89.92 %
Consumo	$4.2W$	$121.88mW$

Capítulo 8

Conclusiones

Se probó el desempeño tanto de la FFNN implementada en Python, como de la descrita en FPGA, empleando imágenes de dígitos escritos a mano provenientes de la base de datos MNIST. Esta comparación se ilustra concretamente en la Tabla 7.1, en la cual es posible visualizar que la velocidad de procesamiento de la FFNN es superior en FPGA, empleando únicamente el 36 % de los elementos lógicos, además de que la precisión de clasificación fue del 89.92 % en ambas plataformas. Por lo anterior, se concluye que la hipótesis, objetivos y especificaciones planteadas al comienzo de este trabajo se cumplieron.

Se aplicaron diversas técnicas de optimización y se diseñaron sistemas eficientes, los cuales en conjunto permitieron generar una arquitectura que logró implementarse en uno de los FPGA de bajo consumo energético con menor cantidad de recursos lógicos. Dentro de las técnicas de optimización empleadas para describir la FFNN se encuentran: la reutilización de hardware, describiendo únicamente las neuronas de la capa más grande; la implementación de un esquema serial-paralelo, en donde se reciben las entradas de forma serial y en paralelo se multiplican con los pesos de cada neurona; la descripción de un solo bloque de función de activación en vez de uno por neurona, y la implementación del método de aproximación lineal por partes (PWL), para la descripción eficiente de la función de activación sigmoide. Además, la selección del formato numérico de punto fijo y el aprovechamiento de los recursos embebidos en el FPGA, tal como los BRAM y los multiplicadores, influyeron en el ahorro sustancial de recursos. Otro aspecto que se solucionó fue uno de los inconvenientes más grandes en el proceso de definición de parámetros de una FFNN, el cual es el hecho de que no existe una fórmula que indique el número de capas y neuronas que tendrán los mejores resultados, por lo que es necesario realizar múltiples pruebas, hasta encontrar la mejor combinación de parámetros. Esto se solventó gracias a la conexión que existe entre la unidad de entrenamiento y la unidad de clasificación, ya que es posible reconfigurar la FFNN sin la necesidad de modificar el hardware ya descrito en el FPGA.

Los resultados de este trabajo demuestran que es posible implementar una FFNN dentro de un FPGA de recursos limitados, capaz de clasificar imágenes de dígitos escritos a mano. También demostró tener mejor desempeño respecto a la misma red implementada en Python, tanto en el tiempo de ejecución, consumo energético y costo del sistema en el cual se implementa. Esto permite resaltar el gran potencial que representan los FPGA para la

implementación de ANN, las cuales con los años han incrementado su complejidad como respuesta a los problemas emergentes, que requieren sistemas cada vez más compactos y eficientes.

8.1. Trabajo a futuro

- Realizar pruebas de desempeño de la FFNN empleando conjuntos de imágenes diferentes a los dígitos escritos a mano.
- Complementar la FFNN añadiendo funciones de activación diferentes a la sigmoide.
- Diseñar un bloque de control que también permita reutilizar las neuronas para una misma capa.
- Hacer uso de las memorias disponibles en la tarjeta de desarrollo, para contar con mayor espacio de almacenamiento para pesos y umbrales.
- Implementar un pipeline, el cual aproveche las neuronas que no son empleadas en una capa.
- Adaptar la FFNN para implementarse como sistema de visión en un robot móvil.

Referencias

[1] A. Rana, A. Singh Rawat, A. Bijalwan y H. Bahuguna, "Application of Multi Layer (Perceptron) Artificial Neural Network in the Diagnosis System: A Systematic Review," en *2018 International Conference on Research in Intelligent and Computing in Engineering (RICE)*, San Salvador: IEEE, ago. de 2018, págs. 1-6, ISBN: 978-1-5386-2599-6. DOI: 10.1109/RICE.2018.8509069. dirección: https://ieeexplore.ieee.org/document/8509069/ (visitado 18-09-2023).

[2] K. S. Zaman, M. B. I. Reaz, S. H. Md Ali, A. A. A. Bakar y M. E. H. Chowdhury, "Custom Hardware Architectures for Deep Learning on Portable Devices: A Review," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, n.º 11, págs. 6068-6088, nov. de 2022, ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2021.3082304. dirección: https://ieeexplore.ieee.org/document/9447019/ (visitado 17-09-2023).

[3] N. Zheng y P. Mazumder, *Learning in Energy-Efficient Neuromorphic Computing: Algorithm and Architecture Co-Design*, 1.ª ed. Wiley, 9 de dic. de 2019, ISBN: 978-1-119-50738-3 978-1-119-50736-9. DOI: 10.1002/9781119507369. dirección: https://onlinelibrary.wiley.com/doi/book/10.1002/9781119507369 (visitado 18-09-2023).

[4] A. Shawahna, S. M. Sait y A. El-Maleh, "FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review," *IEEE Access*, vol. 7, págs. 7823-7859, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2890150. dirección: https://ieeexplore.ieee.org/document/8594633/ (visitado 17-09-2023).

[5] K. Guo, S. Zeng, J. Yu, Y. Wang y H. Yang, "[DL] a survey of FPGA-based neural network inference accelerators," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, n.º 1, págs. 1-26, 31 de mar. de 2019, ISSN: 1936-7406, 1936-7414. DOI: 10.1145/3289185. dirección: https://dl.acm.org/doi/10.1145/3289185 (visitado 18-09-2023).

[6] R. Novickis, D. J. Justs, K. Ozols y M. Greitāns, "An approach of feed-forward neural network throughput-optimized implementation in FPGA," *Electronics*, vol. 9, n.º 12, pág. 2193, 18 de dic. de 2020, ISSN: 2079-9292. DOI: 10.3390/electronics9122193. dirección: https://www.mdpi.com/2079-9292/9/12/2193 (visitado 18-09-2023).

[7] Y. LeCun. "MNIST handwritten digit database." (), dirección: https://yann.lecun.com/exdb/mnist/.

[8] W. S. McCulloch y W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, n.º 4, págs. 115-133, dic. de 1943, ISSN: 0007-4985, 1522-9602. DOI: 10.1007/BF02478259. dirección: http://link.springer.com/10.1007/BF02478259 (visitado 18-09-2023).

[9] J. Savage, "Lección 12: Redes Neuronales Artificiales," CDMX, 21 de abr. de 2023. dirección: https://biorobotics.fi-p.unam.mx/reconocimiento-de-patrones/.

[10] C. Maxfield, *The design warrior's guide to FPGAs: devices, tools and flows*. Boston: Newnes Elsevier, 2004, ISBN: 978-0-7506-7604-5.

[11] J. Misra e I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," *Neurocomputing*, vol. 74, n.º 1, págs. 239-255, dic. de 2010, ISSN: 09252312. DOI: 10.1016/j.neucom.2010.03.021. dirección: https://linkinghub.elsevier.com/retrieve/pii/S092523121000216X (visitado 17-09-2023).

[12] P. Dondon, J. Carvalho, R. Gardere, P. Lahalle, G. Tsenov y V. Mladenov, "Implementation of a feed-forward Artificial Neural Network in VHDL on FPGA," en *12th Symposium on Neural Network Applications in Electrical Engineering (NEUREL)*, Belgrade, Serbia: IEEE, nov. de 2014, págs. 37-40, ISBN: 978-1-4799-5888-7 978-1-4799-5887-0 978-1-4799-5886-3. DOI: 10.1109/NEUREL.2014.7011454. dirección: http://ieeexplore.ieee.org/document/7011454/ (visitado 17-09-2023).

[13] D. Parra y C. Camargo, "A Systematic Literature Review of Hardware Neural Networks," en *2018 IEEE 1st Colombian Conference on Applications in Computational Intelligence (ColCACI)*, Medellin: IEEE, mayo de 2018, págs. 1-6, ISBN: 978-1-5386-6740-8. DOI: 10.1109/ColCACI.2018.8484858. dirección: https://ieeexplore.ieee.org/document/8484858/ (visitado 17-09-2023).

[14] L. D. Medus, T. Iakymchuk, J. V. Frances-Villora, M. Bataller-Mompean y A. Rosado-Munoz, "A Novel Systolic Parallel Hardware Architecture for the FPGA Acceleration of Feedforward Neural Networks," *IEEE Access*, vol. 7, págs. 76 084-76 103, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2920885. dirección: https://ieeexplore.ieee.org/document/8731886/ (visitado 17-09-2023).

[15] A. Dhavlle y S. M. Pudukotai Dinakarrao, "A Comprehensive Review of ML-based Time-Series and Signal Processing Techniques and their Hardware Implementations," en *2020 11th International Green and Sustainable Computing Workshops (IGSC)*, Pullman, WA, USA: IEEE, 19 de oct. de 2020, págs. 1-8, ISBN: 978-1-66541-552-1. DOI: 10.1109/IGSC51522.2020.9291173. dirección: https://ieeexplore.ieee.org/document/9291173/ (visitado 17-09-2023).

[16] S. Himavathi, D. Anitha y A. Muthuramalingam, "Feedforward Neural Network Implementation in FPGA Using Layer Multiplexing for Effective Resource Utilization," *IEEE Transactions on Neural Networks*, vol. 18, n.º 3, págs. 880-888, mayo de 2007, ISSN: 1045-9227, 1941-0093. DOI: 10.1109/TNN.2007.891626. dirección: http://ieeexplore.ieee.org/document/4182384/ (visitado 17-09-2023).

[17] A. W. Savich, M. Moussa y S. Areibi, "The Impact of Arithmetic Representation on Implementing MLP-BP on FPGAs: A Study," *IEEE Transactions on Neural Networks*, vol. 18, n.º 1, págs. 240-252, ene. de 2007, ISSN: 1045-9227. DOI: 10.1109/TNN.2006.883002. dirección: http://ieeexplore.ieee.org/document/4049835/ (visitado 17-09-2023).

[18] I. Tsmots, O. Skorokhoda y V. Rabyk, "Hardware Implementation of Sigmoid Activation Functions using FPGA," en *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*, Polyana, Ukraine: IEEE, feb. de 2019, págs. 34-38, ISBN: 978-1-72810-053-1. DOI: 10.1109/CADSM.2019.8779253. dirección: https://ieeexplore.ieee.org/document/8779253/ (visitado 18-09-2023).

[19] Z. Pan, Z. Gu, X. Jiang, G. Zhu y D. Ma, "A Modular Approximation Methodology for Efficient Fixed-Point Hardware Implementation of the Sigmoid Function," *IEEE Transactions on Industrial Electronics*, vol. 69, n.º 10, págs. 10 694-10 703, oct. de 2022, ISSN: 0278-0046, 1557-9948. DOI: 10.1109/TIE.2022.3146573. dirección: https://ieeexplore.ieee.org/document/9700744/ (visitado 17-09-2023).

[20] Z. Hajduk, "Hardware implementation of hyperbolic tangent and sigmoid activation functions," *Bulletin of the Polish Academy of Sciences: Technical Sciences*, vol. 66, n.º 5, págs. 563-577, oct. de 2018, ISSN: 0239-7528. DOI: 10.24425/124272. dirección: https://www-engineeringvillage-com.pbidi.unam.mx:2443/app/doc/?docid=cpx_7b5db44516775e6dd65M75d21017816339&pageSize=25&index=1&searchId=3174264bd2ef43a8abaaede2bea81026&resultsCount=54&usageZone=resultslist&usageOrigin=searchresults&searchType=Quick.

[21] P. W. Zaki, A. M. Hashem, E. A. Fahim et al., "A Novel Sigmoid Function Approximation Suitable for Neural Networks on FPGA," en *2019 15th International Computer Engineering Conference (ICENCO)*, Cairo, Egypt: IEEE, dic. de 2019, págs. 95-99, ISBN: 978-1-72815-146-5. DOI: 10.1109/ICENCO48310.2019.9027479. dirección: https://ieeexplore.ieee.org/document/9027479/ (visitado 18-09-2023).

[22] V. Tiwari y N. Khare, "Hardware implementation of neural network with sigmoidal activation functions using CORDIC," *Microprocessors and Microsystems*, vol. 39, n.º 6, págs. 373-381, ago. de 2015, ISSN: 01419331. DOI: 10.1016/j.micpro.2015.05.012. dirección: https://linkinghub.elsevier.com/retrieve/pii/S0141933115000642 (visitado 18-09-2023).

[23] L. Wei, J. Cai, V. Nguyen, J. Chu y K. Wen, "P-SFA: Probability based sigmoid function approximation for low-complexity hardware implementation," *Microprocessors and Microsystems*, vol. 76, pág. 103 105, jul. de 2020, ISSN: 01419331. DOI: 10.1016/j.micpro.2020.103105. dirección: https://linkinghub.elsevier.com/retrieve/pii/S0141933120302726 (visitado 18-09-2023).

[24] H. Sun, Y. Luo, Y. Ha et al., "A Universal Method of Linear Approximation With Controllable Error for the Efficient Implementation of Transcendental Functions," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, n.º 1, págs. 177-188, ene. de 2020, ISSN: 1549-8328, 1558-0806. DOI: 10.1109/TCSI.2019.2939563. dirección: https://ieeexplore.ieee.org/document/8844990/ (visitado 18-09-2023).

[25] Z. Qin, Y. Qiu, H. Sun et al., "A Novel Approximation Methodology and Its Efficient VLSI Implementation for the Sigmoid Function," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, n.º 12, págs. 3422-3426, dic. de 2020, ISSN: 1549-7747, 1558-3791. DOI: 10.1109/TCSII.2020.2999458. dirección: https://ieeexplore.ieee.org/document/9106311/ (visitado 18-09-2023).

[26] K. T. Ulrich, S. D. Eppinger y M. C. Yang, *Product design and development*, Seventh edition. New York, NY: McGraw-Hill Education, 2020, 432 págs., ISBN: 978-1-260-04365-5.

[27] F. Pardo Carpio y J. A. Boluda Grau, *VHDL: lenguaje para síntesis y modelado de circuitos*, 3a ed. México, D.F., Madrid: Alfaomega ; RA-MA, 2011, OCLC: 859352781, ISBN: 978-607-707-174-7.