UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Implementación de técnicas de optimización para la instrumentación en FPGA del diseño de funciones de activación para redes neuronales artificiales

TESIS

Que para obtener el título de

Ingeniera Eléctrica Electrónica

P R E S E N T A

Dennisse Alejandra Viveros González

DIRECTORA DE TESIS

M.I. Rebeca Muñoz Melamed

Ciudad Universitaria, Cd. Mx., 2025

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

En primer lugar, quiero agradecer profundamente a mis padres, por su amor incondicional, su apoyo constante, darme la fortaleza para superar cualquier obstáculo en mi camino y por enseñarme con su ejemplo el valor del esfuerzo y la perseverancia. Cada sacrificio que han hecho por mí se ha convertido en mi motivación para seguir adelante. Gracias por creer en mí incluso cuando yo dudaba.

A mis hermanas, por ser mi compañía, mis cómplices, por escucharme y por darme los mejores consejos, incluso cuando no los pedía. Su apoyo y risas han sido fundamentales en este proceso.

A Fernando, por estar a mi lado en los momentos de mayor presión y dificultad. Gracias por tu paciencia, tu ánimo inagotable y tu increíble amistad durante la carrera.

A la M.I. Rebeca, cuya guía y sabiduría han sido fundamentales en este proceso. Agradezco no solo tus grandes conocimientos y tu compromiso académico, sino también tu humanidad, tu empatía y tu generosidad. Encontrar un mentor que combine excelencia profesional con una gran calidad humana es un privilegio poco común, y me siento afortunada por haber cruzado caminos contigo.

Finalmente, a mis amigos, esos seres extraordinarios que han convertido los momentos difíciles en historias para reír y los días buenos en recuerdos inolvidables. Gracias por estar presentes, por sus palabras de aliento, por su comprensión cuando más lo necesitaba y por hacer que este camino, a veces arduo, fuera más llevadero con su compañía, sus locuras y su sinceridad. Cada uno de ustedes ha dejado una huella en mi corazón.

A todos ustedes, gracias de corazón.

Capítulo 1

1. Introducción

Durante los últimos años, las redes neuronales artificiales (ANN, por sus siglas en inglés) han sido un tema de gran interés entre la comunidad científica gracias a su capacidad de aprendizaje de patrones, que le permite predecir o clasificar información [1]. Las redes neuronales artificiales son un sistema de procesamiento inspirado en el funcionamiento real del cerebro humano, con el fin de obtener la resolución de problemas complejos, que sistemas de procesamiento convencionales no podrían [2]. Es por ello que las ANN se presentan como una herramienta capaz de agrupar, aproximar, modelar y asociar patrones complejos a partir de un análisis no lineal y paralelo [3], razón por la cual se ha vuelto muy atractiva.

A partir de su popularidad, se ha desarrollado una nueva área de investigación en torno a la optimización y el aumento de la capacidad de las ANN, centrándose en una de las partes más importantes de una neurona: la función de activación. Esta función permite a la ANN aprender patrones complejos, integrando no linealidad a su comportamiento [4]. Es por ello, que a lo largo de los años han surgido varias funciones de activación mediante la fusión de funciones matemáticas establecidas [5], brindando a cada una un comportamiento diferente, pero que permite simplificar la estructura de la red, así como acelerar el tiempo de convergencia y el proceso de aprendizaje [6].

En la actualidad, las ANN se han desarrollado especialmente por software, por medio de unidades de procesamiento de gráficos (GPU, por sus siglas en inglés), permitiendo desarrollar arquitecturas complejas, tal como las redes profundas. Esto ha permitido la resolución a un gran número de problemas, pero debido a la naturaleza paralela de las ANN, la ejecución de la inferencia en GPUs, al ser una máquina secuencial, conlleva un gran uso de recursos, convirtiéndolo en un proceso costoso [2]. Es por ello que surge como una alternativa viable el desarrollo de la clasificación de las ANN por medio de descripción de hardware, puesto que permite el desarrollo de aplicaciones en tiempo real, con bajo consumo de recursos y energía, permitiendo que se logre preservar el inherente paralelismo [6].

Considerando el beneficio que provee el desarrollo por hardware de las ANN, una alternativa que permite el paralelismo necesario es un Arreglo de Compuertas Programables en Campo (FPGA, por sus siglas en inglés), brindando un alto rendimiento, gran eficiencia energética y mayor flexibilidad que otros sistemas [7]. Los FPGA proporcionan la capacidad de reconfiguración permitiendo evaluar

diferentes opciones de diseño en muy poco tiempo, así como también proporcionan una ganancia de rendimiento superior a la implementación por software [2]. Aunque a pesar de los grandes beneficios que brinda el uso de FPGA, es importante considerar que la implementación de las ANN conlleva una limitación de los recursos lógicos disponibles, por lo que la función de activación implementada jugará un papel importante puesto que podrá reducir el tamaño de la estructura sin perder su eficacia.

1.1. Presentación del problema

Si bien la inteligencia artificial es un tema que surge en la época de los 50s [8], es hasta los 80s que se desarrolla formalmente la primera arquitectura que definiría una red neuronal artificial, comenzando con el desarrollo de las primeras aplicaciones y con el estudio profundo del *machine learning*. Conforme el paso de los años la tecnología avanzó y se comenzaron a construir cada vez computadoras más robustas, con un menor uso de recursos, por lo que se fueron desarrollando progresivamente redes neuronales más complejas y profundas.

Actualmente, las ANN han probado ser una herramienta capaz de resolver un gran número de problemas como la navegación autónoma, procesamiento del lenguaje natural, asistencia virtual, reconocimiento de imagen, detección de objetos, entre otros, realizando grandes contribuciones a la sociedad como la detección rápida de COVID, clasificación de electroencefalogramas, aplicaciones en la manipulación genética, detección temprana de cáncer [9–12], entre otras muchas aplicaciones.

A pesar de las grandes contribuciones y el gran número de aplicaciones que se le ha dado a las redes neuronales, esto es solo el comienzo de todo el potencial que puede proveer, puesto que el uso de las ANN sigue estando limitado a proyectos que cuenten con GPUs de gran potencia, lo cual implica altos costos, alto consumo de energía y la disponibilidad de espacio. Por ello, considerando esta problemática, con las nuevas investigaciones, se espera que las redes neuronales puedan aplicarse a un mayor número de proyectos los cuáles se caracterizan por ser compactos, de bajo consumo, etc.

1.2. Hipótesis

A partir de la implementación de técnicas de optimización y aproximación lineal por partes es posible instrumentar todas las funciones de activación del Estado del Arte, la función ReLU, Sigmoide, ELU, Tangente Hiperbólica, Leaky ReLU y Swish, manteniendo una buena precisión y un bajo consumo de recursos. Además, es posible instrumentar tales funciones del Estado del Arte en una misma red neuronal artificial eficiente, permitiendo la instrumentación del sistema en un FPGA de recursos limitados.

1.3. Objetivo

1.3.1. Objetivo general

- Implementación de técnicas de optimización para la instrumentación en FPGA de las funciones de activación Sigmoide, Tangente Hiperbólica, Swish, ELU, RELU y Leaky Relu, para una red neuronal artificial FeedFoward reconfigurable, siendo posible modificar el número de capas, el

número de neuronas por capas y la función de activación, por medio de una interfaz gráfica de usuario.

1.3.2. Objetivos específicos

- Realizar la arquitectura de una red neuronal artificial FeedFoward reconfigurable que permita modificar sus principales parámetros.
- Implementar técnicas de optimización para instrumentar las diferentes funciones de activación.
- Diseñar las diferentes funciones de activación en descripción de hardware.
- Comparar el consumo de recursos de las funciones de activación que ya cuentan con aproximaciones PWL del presente trabajo con el Estado del Arte.
- Realizar pruebas de desempeño de la red neuronal artificial con imágenes de la base de datos MNIST.

1.4. Alcance

Para este trabajo se propone el diseño de diferentes funciones de activación para su instrumentación en FPGA implementadas en una red neuronal Feed-Forward (FFNN, por sus siglas en inglés). El alcance del presente trabajo es implementar las funciones Sigmoide, Tangente Hiperbólica, Swish, ELU, RELU y Leaky Relu en una FFNN previamente programada y probada en Python, teniendo por entrada la base de datos MNIST [13], por lo que clasifica imágenes de números manuscritos en dígitos del 0 al 9.

1.5. Justificación

La investigación actual de las ANN apunta al beneficio de implementar redes en hardware de bajo consumo energético [2, 3, 7, 14], que posibilita su implementación en proyectos de recursos limitados. Aunque a pesar de las ventajas, la instrumentación de redes continua siendo un área poco explorada debido a los retos que conlleva tener elementos lógicos limitados, sin embargo, y en consecuencia se han desarrollado técnicas de optimización para la descripción de ANN en hardware [2, 7].

Si bien, la optimización de la descripción de redes ha permitido su instrumentación, siguen persistiendo limitaciones en cuestión a la profundidad de la red, por lo que la implementación eficiente de las funciones de activación en hardware se vuelve un área crítica, puesto que a partir de la función que se implemente se obtienen diferentes desempeños y diferente uso de recursos [1, 10, 15–20], permitiendo la creación de redes profundas, sin sacrificar la eficiencia de la misma.

Capítulo 2

2. Marco teórico

Las redes neuronales artificiales nacen bajo la intención de obtener una computadora que resuelva problemas complejos con la misma facilidad que un cerebro humano lo logra. La velocidad de respuesta, la capacidad de aprendizaje y la autonomía son las principales características que se han buscado imitar para implementar en sistemas computacionales. Es a partir de ese objetivo que se realizan los siguientes cuestionamientos ¿cómo funciona el cerebro humano? y ¿cómo se puede computarizar? Es por ello que, como primer sección de este capítulo, se abordará el funcionamiento de una neurona biológica, para posteriormente profundizar como se compone una neurona artificial, y siguiendo con el orden, se describirá el modelo que constituye la red neuronal artificial, para finalizar con la descripción de una FPGA.

2.1. Neurona biológica

A lo largo de los años el cerebro humano y su funcionamiento ha sido una de las grandes interrogantes para la comunidad científica, puesto que es extremadamente eficiente en la resolución de tareas complejas, como el reconocimiento de patrones, el aprendizaje a partir de la experiencia y la toma de decisiones en entornos inciertos. Aunque no sólo el humano esta previsto de estas capacidades, tan solo el pequeño cerebro de una paloma es más capaz que una computadora digital con un gran número de elementos lógicos, un enorme espacio de almacenamiento, y a frecuencias mayores que las de un cerebro [8].

De primera instancia, se definirá la neurona biológica como la unidad estructural y funcional del cerebro biológico; esta consta de un cuerpo celular o *soma* que contiene el núcleo, del cual salen prolongaciones arborescentes llamadas *dendritas*, como se observa en la Figura 2.1, que serán la entrada de información, mientras que la salida se conoce como *axón*, esta se encargará de la transmisión de información con las otras neuronas. Entre ellas existe una relación de contigüidad, pero no de continuidad y será por medio de la sinapsis que se vincularán por la transmisión de impulsos eléctricos, intercambiando información de una neurona a otra [21, 22].

El comportamiento fundamental de una neurona se basa en la excitación o estimulación de las entradas (dendritas), que cuando éstas alcanzan un cierto umbral, la neurona se dispara o activa, pasando una señal hacia al axón, que se transmitirá por las dendritas de otra neurona [23]. Por lo

tanto, se entiende la transmisión y procesamiento de información como una conexión masiva y paralela de neuronas, creando una red neuronal.

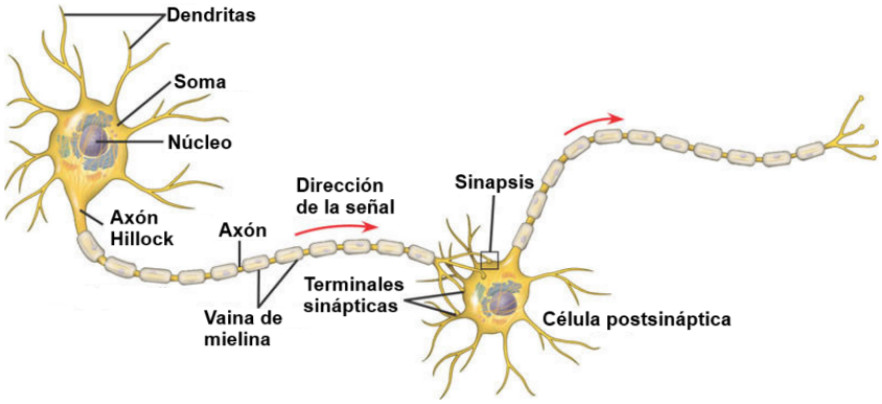


Figura 2.1: Estructura simplificada de una neurona biológica [22]

2.2. Neurona artificial

Desde el principio de la computación, uno de los mayores desafíos en la informática tradicional era programar sistemas que pudieran resolver tareas específicas sin intervención humana constante. Es por ello que a partir del entendimiento del funcionamiento de una neurona biológica, se plantea la meta de computarizar su comportamiento para obtener sistemas informáticos que puedan tener sus mismas capacidades.

En 1943, el neurofisiólogo Warren McCulloch y el matemático Walter Pitts, presentan la primer propuesta de como podría funcionar una red neuronal artificial, un modelo que se ha conservado hasta la actualidad gracias a su precisa similitud con la neurona biológica [7]. A partir de este primer modelo, que da pie a la posibilidad de imitar el cerebro humano, comienzan las investigaciones acerca del aprendizaje para poder simularlo, por ello, en 1949 Donald Hebb explica los procesos del aprendizaje, desarrollando una regla de cómo el aprendizaje ocurría, sentando las bases de la Teoría de las redes neuronales [23].

El modelo presentado por McCulloh y Pitts se basa en lo que se denominó como Lógica de Umbral, en el cual, por medio de una combinación lineal se activará o apagará una neurona en cierta medida, de acuerdo a sus entradas. Esto se puede percibir en la Figura 2.2, en la cual se observa un vector de entradas de tamaño n, representado como x, una variable θ que define el valor umbral, una función matemática denominada *Función de activación* y finalmente el resultado z que indica el nivel de encendido de la neurona.

Conjuntamente, gracias a las aportaciones de Donald Hebb, en la teoría de redes neuronales, se explica como a partir del aprendizaje y la experiencia, para la resolución de tareas, ciertas neuronas tendrán una conexión más fuerte con unas que con otras, por lo que ciertas entradas tendrán más importancia que otras, esto varía de acuerdo al problema a resolver. Es por ello, que en la Figura 2.2,

se observa además un vector de variables de entrada denominadas w, de tamaño n, que hace referencia al peso que tendrá cada entrada, o dicho de otra forma, que tanto impactará la entrada en la neurona.

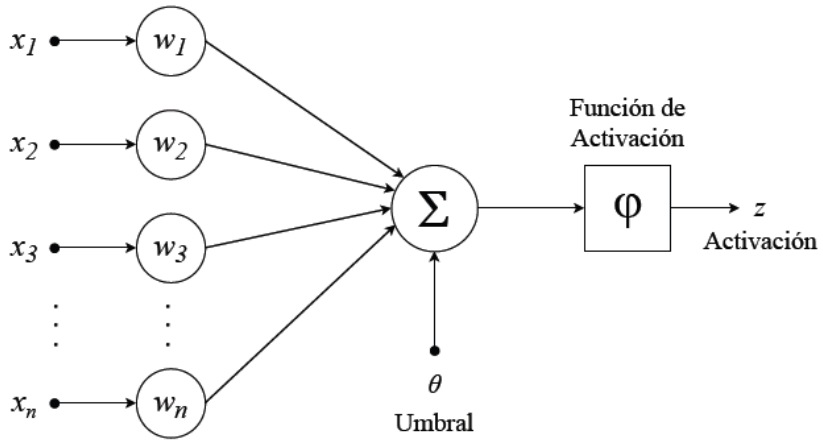


Figura 2.2: Modelo de una neurona artificial [7].

Este modelo de la neurona, se ve definida inicialmente por la ecuación 2.1, en la cual se hará la suma de las entradas ponderadas por los pesos y se les restará el valor umbral, esto con el objetivo de que únicamente pase por la entrada aquella suma ponderada que sobrepase el umbral de la neurona. Finalmente, la salida de la neurona (2.2) es el resultado de evaluar lo que se obtuvo en la ecuación 2.1, en la función de activación.

$$s = \sum_{i=1}^{n} w_i x_i - \theta \tag{2.1}$$

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

Con el fin de simplificar la expresión 2.1, el valor umbral se integrará como el peso inicial, asumiendo una entrada de $x_0 = 1$, por lo que se tendría que $w_0 = -\theta$. A partir de esta modificación se logra la expresión 2.3, con la ventaja de obtener todo en términos de las entradas y los pesos únicamente.

$$s = \sum_{i=0}^{n} w_i x_i \tag{2.3}$$

Considerando la definición matemática de la neurona es importante tomar en cuenta la fase de *entrenamiento*, que se explicará detalladamente en la sección 2.3.1, pero como objetivo principal se espera que en ella se determinen los valores de los pesos, puesto que para cada problema se obtendrán diferentes valores. Por otra parte, las entradas, a diferencia de los pesos, serán definidas de acuerdo a la naturaleza del problema, así como la función de activación.

Tomando en cuenta las limitaciones que tendría el uso de una sola neurona, es que surge la idea y la necesidad de replicar una red neuronal al igual que el cerebro biológico, esto con el fin de resolver una gama más amplia de problemas. Sin embargo, es hasta 1986, que gracias al desarrollo de diferentes teorías y el perfeccionamiento de la etapa del entrenamiento, es que se comienza a visualizar un

panorama alentador para las redes neuronales que puedan resolver problemas reales (no lineales).

2.2.1. Función de activación

Comprendiendo la estructura de una neurona artificial, se puede observar que un factor de gran relevancia en ella es la función de activación, siendo el corazón de la neurona, pues es determinante para el resultado. Las funciones de activación se definen como ecuaciones matemáticas bioinspiradas para representar el potencial de acción de disparo en una neurona o nodo [24]. Su objetivo principal es establecer una diferenciación entre ruido y entradas reales, analiza si la entrada es una señal lo suficientemente fuerte para sobrepasar el límite establecido [8].

Parte de la importancia de la función de activación es que provee la necesaria no linealidad al modelo, para ser capaz de aprender de datos complejos y de orden superior, con el fin de proporcionar predicciones y clasificaciones precisas. Sin embargo, para incorporar la no linealidad al modelo es importante considerar que la función matemática que se integre deberá ser no lineal. Es a partir de esta premisa que se han implementando diferentes funciones matemáticas como funciones de activación obteniéndose un impacto diferente por cada función, análisis que se realizará de manera más detallada en la sección 3.3.

En la Tabla 2.1 se muestran algunas de las funciones más utilizadas en el desarrollo de redes neuronales artificiales, así como algunas de las más nuevas que se han propuesto. Como se puede observar, la complejidad de las funciones varía, algunas contendrán parámetros, hiperparámetros, fracciones y/o multiplicaciones.

De primera instancia se encuentra la ecuación lineal, que a pesar de ser una función que resultaría fácil de integrar en una neurona, tiene un comportamiento lineal que no beneficia al aprendizaje, por lo que fue descartada como una buena opción. Es importante tomar en cuenta que la no linealidad en las funciones no mejora sólo el aprendizaje, jugando un papel importante en su velocidad de convergencia, así como en la eficiencia computacional. La convergencia de una red neuronal artificial hace referencia a aquel punto en el entrenamiento, en el cual la red neuronal ya no está mejorando su resultado significativamente, aunque se explicará con mayor profundidad en la Sección 2.3.1.

En contra parte, la ecuación escalón fue la función que se comenzó a utilizar gracias a tener un comportamiento excluyente, al producir una respuesta nula ante ciertas entradas, lo cual es lo esperado de la función de activación. Sin embargo, a pesar de la facilidad y el comportamiento deseado que presenta la ecuación escalón, su gráfico, que se muestra en la Figura 2.3, se define por bordes duros y cambios instantáneos, un comportamiento anti natural y poco realista en comparación a una neurona biológica real.

Función de activación	**Fórmula**
Lineal	$f(s) = as + b$
Escalón	$h(s) = \begin{cases} 1, & \text{si } s > 0 \\ 0, & \text{si } s \leq 0 \end{cases}$
Sigmoide	$\sigma(s) = \frac{1}{1+e^{-s}}$
Tangente hiperbólica	$tanh(s) = \frac{e^{s}-e^{-s}}{e^{s}+e^{-s}}$
Swish	$f(s) = s \cdot \sigma(s)$
Rectifier Linear Unit (ReLU)	$ReLU(s) = \begin{cases} s, & \text{si } s > 0 \\ 0, & \text{si } s \leq 0 \end{cases}$
Leaky ReLU	$LReLU(s) = \begin{cases} s, & \text{si } s > 0 \\ \alpha s, & \text{si } s \leq 0 \end{cases}$
Exponential Linear Unit (ELU)	$ELU(s) = \begin{cases} s, & \text{si } s > 0 \\ \alpha(e^{s} - 1), & \text{si } s \leq 0 \end{cases}$

Tabla 2.1: Funciones de activación.

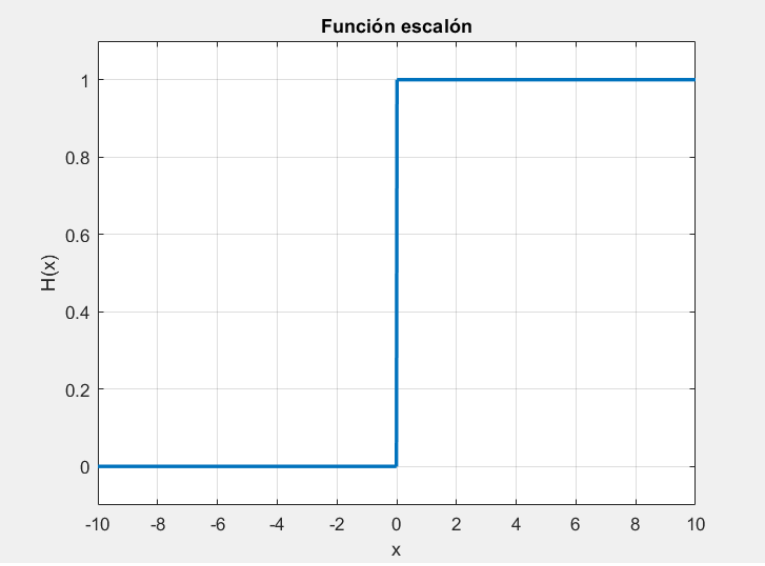


Figura 2.3: Gráfica de la función Escalón.

Debido a la inconformidad de la comunidad científica sobre la función escalón, es que comienzan a surgir funciones que tienen por objetivo asemejar un comportamiento más natural y real, y conforme se avanza en la investigación de las ANN, ya no sólo se busca tal comportamiento, también se empiezan a implementar ecuaciones que mejoren el rendimiento computacional, ya sea que eficienten

el entrenamiento o que su implementación conlleve una menor cantidad de recursos. En la Sección 2.3.2 se describirá con mayor detalle las diferentes funciones de activación que se han implementado en redes neuronales artificiales, y su impacto en la red.

2.3. Red Neuronal Artificial (ANN)

Con el objetivo de resolver tareas complejas, y bajo la inspiración de un cerebro humano, se plantea la conexión de múltiples neuronas, las cuales se encuentran estructuradas en capas y conectadas entre sí, para obtener una red de procesamiento de información. Por consecuente, se obtiene una Red Neuronal Artificial, un sistema de análisis que se constituye por nodos computacionales (neuronas) que se caracterizan por un procesamiento extraordinario de información; se definen como un paradigma matemático y computacional enfocado al reconocimiento de patrones y clasificación de datos [7].

Es en 1957, que a partir del modelo de la neurona artificial, al cual se nombra como *perceptrón*, Frank Rosenblatt comienza el desarrollo de la primera red neurona artificial, a la cual nombró *Perceptrón Multicapa* (MLP, por sus siglas en ingles). El MLP resolvía una serie de patrones a partir de un entrenamiento dado, pero su mayor defecto fue que sólo podía resolver problemas lineales; el primer problema que no pudo resolver fue simular el comportamiento de una compuerta XOR [23]. Sin embargo, las limitantes de la red que se propuso no se encontraban en la estructura, sino en el entrenamiento realizado.

La estructura del Perceptrón Multicapa se convierte en la disposición más básica de una ANN, arquitectura que se nombró como *Feed-Forward*, que como su nombre lo indica, la información es transferida en una única dirección. Como se observa en la Figura 2.4, las unidades neuronales o perceptrones, están acomodadas en capas, en las cuales cada nodo está conectado con todos los nodos de la siguiente capa y así sucesivamente hasta llegar a la salida, tomando en cuenta que su conexión jamas deberá ser cíclica; a esta topografía se le conoce como *totalmente conectada* (FC, por sus siglas en ingles).

Considerando la estructura del MLP, y de acuerdo con lo mencionado en la sección 2.2, se describió la ecuación que define a una neurona artificial (2.3) y (2.2), por lo que ahora en una red, la neurona seguirá descrita por la misma función, aunque esta vez considerando que esta se encuentra en una capa L, obteniéndose la ecuación 2.4 y 2.5, que definen a una neurona j.

$$s_j^L = \sum_{i=0}^{n} w_{ij}^L z_i^{L-1} \tag{2.4}$$

$$z_j^L = f(s_j^L) \tag{2.5}$$

La arquitectura se construye de tres capas principales, la *capa de entrada*, que serán las entradas del problema a resolver; la *capa oculta*, en la cual podrán existir varias capas y varios nodos, esto dependerá del problema a resolver y los recursos disponibles; finalmente, se tendrá la *capa de salida*,

que contendrá las salidas del problema. Cada conexión será la representación de un enlace sináptico, y estará definida por un peso, y una única dirección, hacia adelante, procesándose la información de forma paralela.

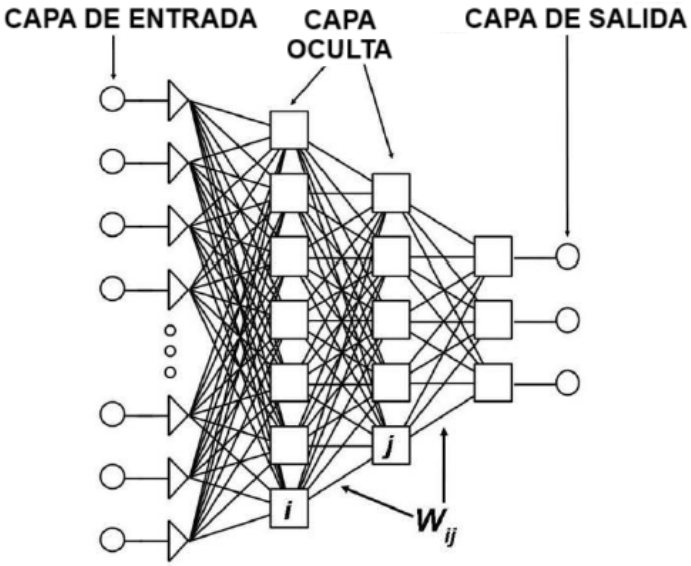


Figura 2.4: Estructura de una Red Neuronal Artificial Feed-Forward con dos capas ocultas [3].

En la actualidad, las redes neuronales Feed-Forward FC (FFNN, por sus siglas en ingles) es una de las redes más utilizadas por su facilidad de implementación, su estructura básica no supone un gran consumo de recursos, aunque esto varía de acuerdo a cuantas capas ocultas se agreguen, siendo que cuando se integran mínimo tres capas ocultas, comienza a considerarse una red neuronal profunda (DNN, por sus siglas en ingles), esto hace referencia a que tiene un 'aprendizaje profundo', o dicho de otra forma, el aprendizaje es más robusto logrando resolver problemas cada vez más complejos.

2.3.1. Entrenamiento

Como se ha mencionado anteriormente, el entrenamiento será aquella fase que tiene por objetivo asignarle a los pesos valores adecuados para que se obtenga la salida deseada, resolviendo una tarea en específico. Esta etapa se logra gracias a procesos iterativos, en los cuales se probará la ANN con un conjunto de datos, los cuales se tienen previamente etiquetados con su respuesta correcta, se introducirán y se repetirá el proceso hasta que los resultados, en su mayoría, concuerden con las etiquetas de los datos y se obtenga un error mínimo.

Para lograr este proceso se hace uso de algún método de aprendizaje, existiendo múltiples criterios que permiten llevar a cabo el entrenamiento de una red neuronal, los cuales involucran la modificación sucesiva de sus parámetros [7]. En el presente trabajo se abordará únicamente el método de **Propagación Inversa**, el cual ha demostrado ser eficiente e idóneo para redes con arquitectura MLP [3], siendo uno de los métodos de aprendizaje más populares.

A *grosso modo*, el método obtendrá el error en la salida, comparando el valor esperado con lo que se obtuvo, para que, como su nombre lo indica, propague el error en dirección opuesta, de la salida hacia la entrada, para afectar el valor de los pesos en proporción a su aportación a la salida; esto provocará que los pesos mejoren sus valores en correspondencia al error que generan. Por lo que, por cada iteración se medirá el error y se detendrá cuando este error sea aceptable. La propagación del error en dirección inversa se realizará por medio del método del *descenso de gradiente*.

El descenso de gradiente es un algoritmo de optimización iterativo de primer orden que permite encontrar mínimos locales en una función diferenciable; para minimizar la función se sigue el negativo del gradiente, obteniéndose varias aproximaciones en las cuales se va minimizando el error, deteniéndose el proceso cuando el error sea aceptable [8]. El método es útil para resolver fácilmente funciones matemáticas complejas o con muchos parámetros, ya que de una función continua, crea una discontinua.

La principal ventaja del método es que es prácticamente a prueba de fallas si se comete un error en el proceso o los datos contienen errores, puesto que con cada iteración se corrige el error. Para este método se considera el error cuadrático, que estará definido por la ecuación 2.6, aunque a pesar de poder definirse de distintas formas, el error cuadrático presenta mayores ventajas al tener una escala más pequeña, permitiendo que el gradiente no pierda valores importantes, además de que su función es lisa y continua.

$$Error = (ValorEsperado - ValorReal)^2 \tag{2.6}$$

La propagación del error se realizará en la actualización de los pesos, en cada valor nuevo se considera el valor pasado menos la derivada del error escalado por una variable α, como se muestra en la ecuación 2.7. Esta variable se le conoce como *Tasa de aprendizaje*, es un factor de moderación, puesto que modera la fuerza de los cambios del error para garantizar que no se exceda el impacto del error o ruido.

$$nuevoW_{ij} = pasadoW_{ij} - \alpha \frac{\delta E}{\delta W_{ij}} \tag{2.7}$$

La derivada del error estará definida por la ecuación 2.8, conocida como la llave del entrenamiento; se definirá por:

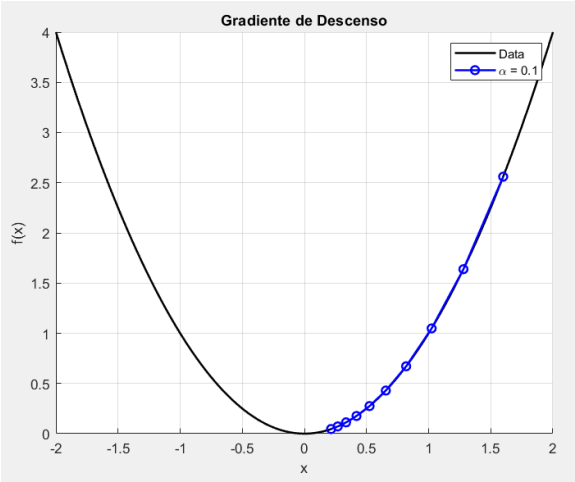
$$\frac{\delta E}{\delta W_{ij}} = \frac{\delta}{\delta W_{ij}}(t_j - O_j)^2 = -(t_j - O_j) \cdot \frac{\delta O_i}{\delta W_{ij}} \cdot O_j \tag{2.8}$$

- El error de la salida actual
- La salida de la señal del nodo anterior antes de que se aplique la función de activación, que matemáticamente se define como la derivada de la función de activación aplicada en el nodo anterior
- La salida del nodo actual

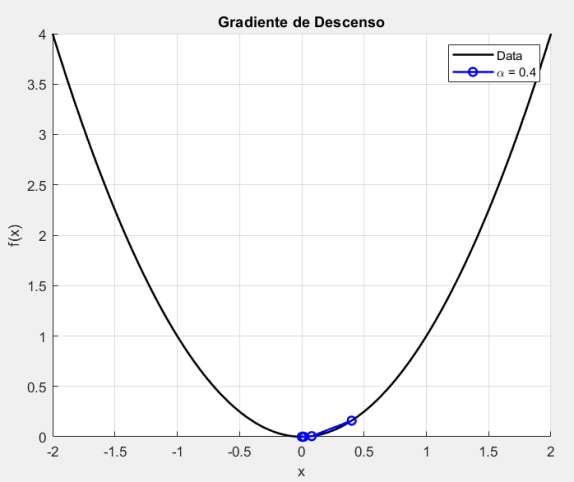
Donde

- t_j es la salida esperada del nodo actual
- O_j es la salida real del nodo actual
- O_i es la salida real del nodo anterior

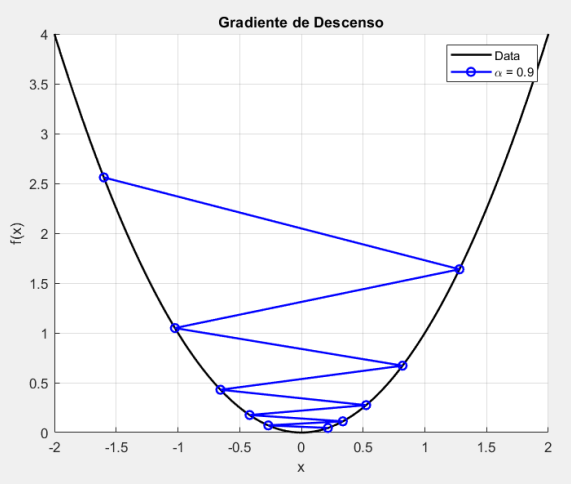
Gráficamente, el descenso del gradiente comenzará en un punto aleatorio y a partir de la derivada, se avanzará a aquel punto en el que el descenso sea más pronunciado; dependiendo del valor de la tasa de aprendizaje, cada salto entre punto y punto será más grande o pequeño, siendo que con un número fijo de iteraciones, este parámetro hará que se llegue o no al mínimo local del error. Como se puede observar en la Figura 2.5, una tasa de aprendizaje muy grande puede perder los mínimos locales fácilmente, mientras que uno muy pequeño puede hacer el proceso muy tardado; siendo que cada punto que se observa representa una iteración.



(a) Tasa de aprendizaje del 0.1



(b) Tasa de aprendizaje del 0.4



(c) Tasa de aprendizaje del 0.9

Figura 2.5: Gradiente de descenso con diferentes tasas de aprendizaje.

Es importante considerar que es de acuerdo a la forma y tamaño de la red neuronal que se establecerán los límites del aprendizaje, puesto que la capacidad de aprendizaje no depende solo del

entrenamiento, sino que también de las características de la red y sus diferentes parámetros, siendo que el comportamiento de los parámetros no es lineal, como el número de capas ocultas, de nodos, de iteraciones y la tasa de aprendizaje.

Uno de los factores de diseño que más afectan el entrenamiento es la función de activación que se implemente, ya que como se observa en la ecuación 2.8, la derivada de la función es un factor importante, esto puede ayudar a que se tenga una convergencia del gradiente más rápida o al contrario, que exista un desvanecimiento del gradiente.

La convergencia del gradiente es cuando el gradiente alcanza el mínimo local. Un ejemplo es la Figura 5b, como se observa en el proceso se alcanza con pocas iteraciones el mínimo de la función; mientras que en las Figuras 5a y 5c se realizan 10 iteraciones, pero en ambos procesos no converge el gradiente al no llegar al mínimo local. En contra parte, el desvanecimiento del gradiente es lo que se puede observar en la Figura 5a, el gradiente comienza a progresar cada vez en pasos más pequeños, por lo que las actualizaciones comienzan a ser insignificantes y el aprendizaje se vuelve lento; este problema se vuelve crítico en redes neuronales profundas, debido a que la cantidad de capas provoca que los valores en el momento de la propagación inversa se vuelvan prácticamente nulos.

2.3.2. Funciones de activación en ANN

A continuación se describirá cada una de las funciones restantes que aparecen en la Tabla 2.1, conforme al orden cronológico de aparición. Se analizará su comportamiento a partir de su gráfica y su paisaje de salida, el cual es una superficie multidimensional, que en el contexto de ANN, se refiere a la representación geométrica de cómo varían las salidas de la red en función de sus parámetros de entrada y los pesos de la red [25]. Es un gráfico que ayuda a visualizar la complejidad y estructura del espacio de salida, prediciendo su comportamiento.

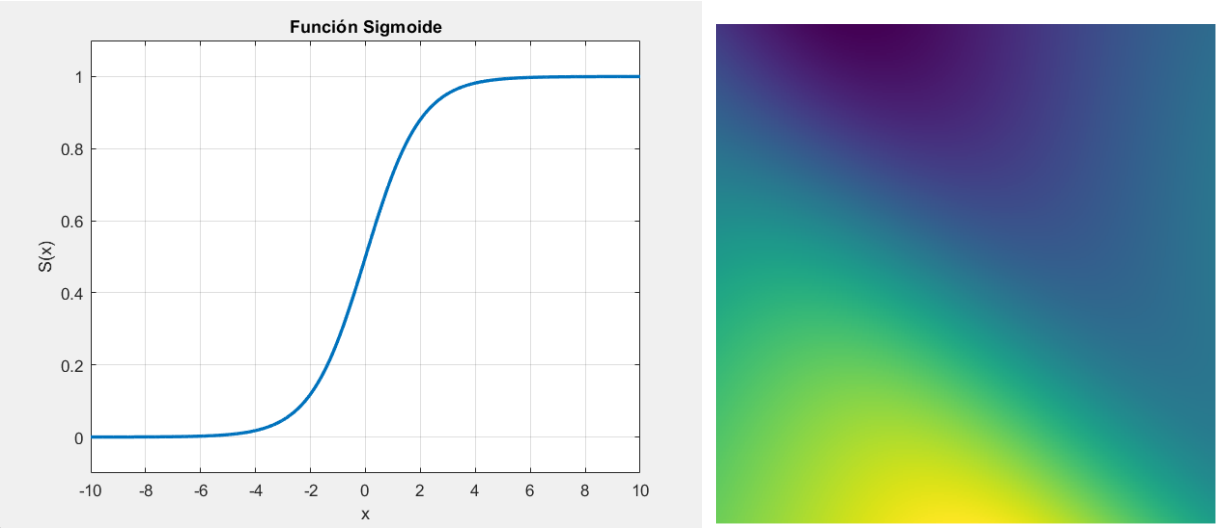
El paisaje de salida indica características propias de la función como forma, suavidad, saturación, linealidad, valles y/o picos, y será a partir de estos elementos que se podrá interpretar comportamientos como la propagación del gradiente, robustez, eficiencia e impacto en la optimización. La representación de las características mencionadas se observan a través del mapeo de colores; las zonas de amarillo a verde indica aquella región de salida en el que las neuronas se activan ($f(x) \neq 0$), siendo que la zona amarilla son los valores de salida más altos aceptables y la región verde son las salidas óptimas que permite un correcto aprendizaje. En contra parte,la zona de azul claro a fuerte indica aquella región en el que las neuronas comienzan a morir ($f(x) = 0$), por lo que mientras más fuerte sea el azul, indica un mayor número de neuronas muertas.

Es a partir de la interacción entre los colores que se puede extraer el comportamiento de la función: se puede observar la transición de colores, que cuando sucede de manera paulatina indica suavidad de la función; la presencia de azul indica una progresiva saturación; la concentración de verde indican valles, región en donde el error es mínimo, mientras que los azules son picos, que representa errores grandes. Posteriormente, el análisis de tales comportamientos nos indica que funciones suaves son

más fáciles de optimizar, debido a que tiende menos al desvanecimiento del gradiente, puesto que si hubiera regiones con cortes bruscos entre colores, podría indicar zonas de alta sensibilidad al gradiente o problemas de convergencia; la presencia de mucho azul indica una fácil saturación lo cual indica que es una función poco robusta y que además, no funcionaría con aprendizaje profundo, mientras que una mayor presencia de verde o amarillo indica mayor eficiencia.

- **Sigmoide**

  La función Sigmoide surge como una solución a los defectos de la función escalón, debido a que logra suavizar el comportamiento de disparo, dando origen al propio nombre de la función, debido a que 'Sigmoide' se refiere a 'con forma de S' [4]. La comparación se puede observar fácilmente entre la Figura 2.3 y 2.7.



(a) Gráfica de la función Sigmoide. (b) Paisaje de salida de la función [18].

Figura 2.6: Función Sigmoide.

La ecuación traduce una entrada de rango $(-\infty, \infty)$ a una salida con rango de [0,1], siendo esto uno de sus mayores beneficios, pues su salida puede ser interpretada como una probabilidad que se le asigna a cada región, es por ello, que es una de las ecuaciones más utilizadas para la clasificación binaria, así como en el desarrollo de redes neuronales FeedFoward [19]. Además, tiene como ventaja que su derivada usa la propia función, como se observa en la ecuación 2.9, ahorrando recursos en su implementación.

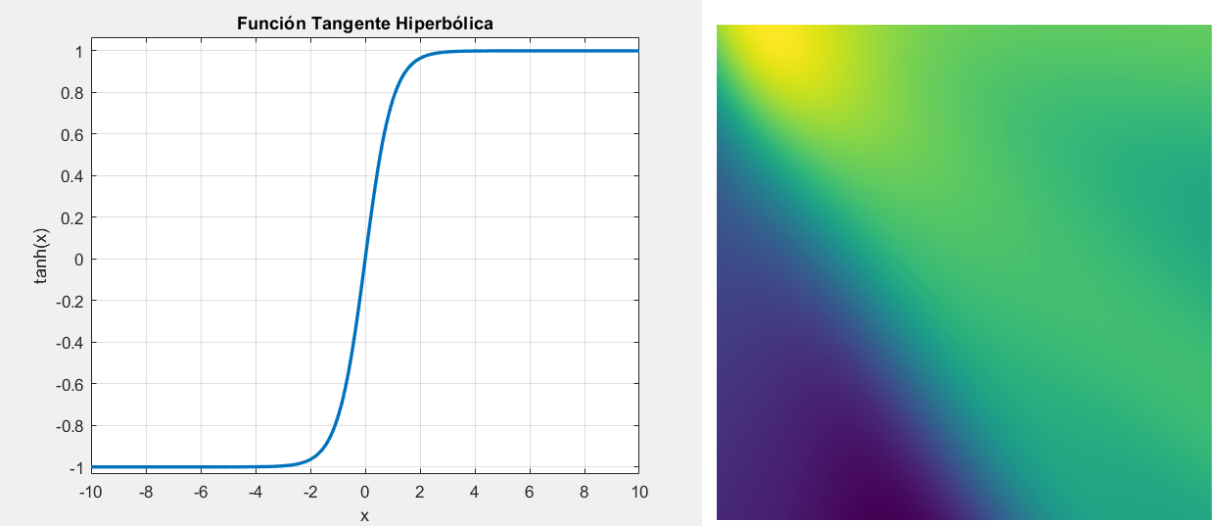
$$\sigma'(s) = \frac{e^{-s}}{(1+e^{-s})^2} = \sigma(s)(1-\sigma(s)) \tag{2.9}$$

Sin embargo, a pesar de las ventajas que presenta, la función tiene importantes inconvenientes. Como se puede observar en la Figura 6b, indica una saturación de gradiente, convergencia lenta y una salida centrada distinta de cero, lo que hace que las actualizaciones de gradiente se propaguen en diferentes direcciones, provocando el desvanecimiento del gradiente. Esto provoca que

la función no sea la mejor opción para aprendizaje profundo, volviéndose difícil de optimizar después de cierto punto.

- **Tangente Hiperbólica**

  Esta función se define como la relación entre las funciones hiperbólica del seno y el coseno. Como se muestra en la Figura 7a, tiene una estructura similar a la función Sigmoide, por lo que se caracteriza por ser una función suave, pero se diferenciará por su salida que va de un rango de [-1, 1], centrada en cero, siendo esta su mayor ventaja, ya que ayuda al proceso de entrenamiento. Además, presenta un gradiente más fuerte que la Sigmoide, con un rango útil y valores negativos como salida beneficiando a la diferenciación entre ruido y una entrada real. Gracias a estos beneficios se comenzó a utilizar como una función de activación, esperando que resuelva los problemas que causaban la función Sigmoide.



(a) Gráfica de la función Tangente hiperbólica. (b) Paisaje de salida de la función [18].

Figura 2.7: Función Tangente hiperbólica.

La derivada de la Tangente hiperbólica (2.10) otorga la misma ventaja que la función Sigmoide, que se define con la propia función, ahorrando recursos. Sin embargo, como se puede observar en la Figura 7b, la derivada tiende al desvanecimiento del gradiente perjudicando el aprendizaje, esto debido a que la función produce que algunas neuronas mueran (que tengan por salida cero) durante el cálculo. A pesar de sus desventajas, es una función comúnmente usada para redes neuronales multi capas, debido a que mejora el resultado de entrenamiento, así como en redes neuronales recurrentes [19], un tipo de arquitectura que la presente investigación no abarca.

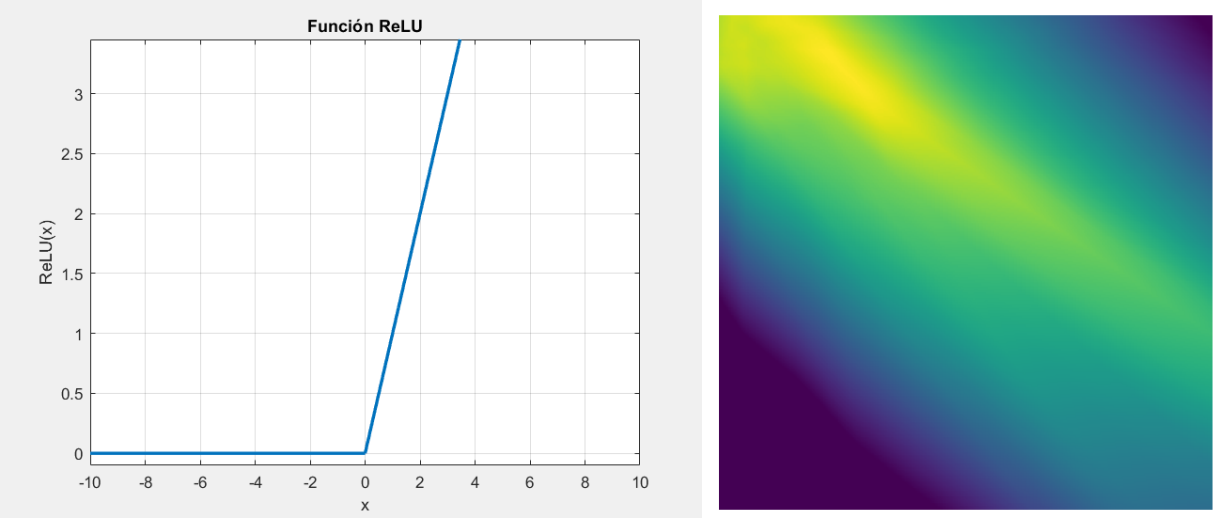
$$\sigma'(s) = \frac{1}{cosh^2(s)} = 1 - tanh^2(s) \tag{2.10}$$

- **Rectifier Linear Unit (ReLU)**

La ecuación ReLU es una de las funciones más utilizadas en todo el campo de redes neuronales. Se propone en 2010, por Nair y Hinton, como una función que pueda implementarse en redes neuronales profundas, que hasta ese momento ninguna de las funciones anteriores servían. La ecuación representa una función casi lineal y, por lo tanto, conserva las propiedades de los modelos lineales que la hacen fácil de optimizar, pero mantiene la propiedad de no linealidad que le permite aprender patrones complejos. Su implementación resulta bastante eficiente puesto que consume lo mínimo de recursos gracias a su simplicidad matemática.

Su principal ventaja es que logra reducir el problema del desvanecimiento del gradiente, gracias a que su derivada resulta ser la función escalón, dando como resultado que el gradiente sólo pueda tomar dos valores, 0 para los valores negativos y 1 para los valores positivos. Este comportamiento, asimismo, garantiza un procesamiento computacional mucho más rápido que las funciones anteriores, esto provoca que la convergencia del gradiente sea muy rápida.

A pesar de su fiabilidad, el mayor defecto de la función es lo que se conoce como "*dying ReLU*", un problema que provoca la muerte de muchas neuronas, debido a que toda aquella entrada negativa genera como salida un cero, con gradiente cero, como se puede ver en la Figura 9b, con una gran presencia de azul fuerte. Este problema se puede volver crítico o no dependiendo de la tarea que se desee resolver. Adicionalmente, la función no es una buena opción para resolver problemas con entradas muy grandes, debido a que el resultado escala de gran manera que puede generar que provoque resultados falsos. Igualmente, su salida carece de una interpretación probabilística, ya que la salida no está ligada entre 0 y 1, lo que dificulta su uso en modelos que requieren salidas probabilísticas.



(a) Gráfica de la función ReLU. (b) Paisaje de salida de la función [18].

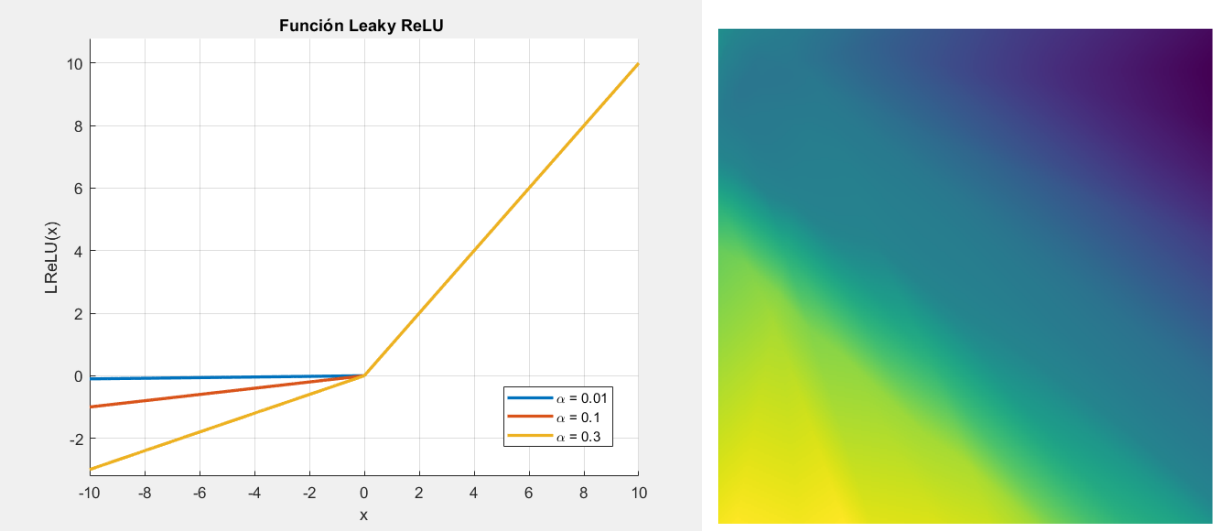
Figura 2.8: Función ReLU.

- **Leaky Rectifier Linear Unit (LReLU)**

  Debido al problema que provoca la salida igual a cero con valores negativos en la función ReLU, se propone cambiar esta salida por un valor pequeño, pero diferente de cero. Es en 2013 que se propone la ecuación Leaky ReLU, que plantea introducir un parámetro externo que escale la entrada a un valor más pequeño para valores negativos, usualmente este parámetro es igual a 0.01 [17, 19, 26, 27]. Gracias a esta modificación en la ecuación, se logra reducir el problema de "*dying ReLU*", debido a que su derivada (2.11) tampoco produce resultados nulos.

$$LReLU'(s) = \begin{cases} 1, & \text{si } s > 0 \\ \alpha, & \text{si } s \leq 0 \end{cases} \tag{2.11}$$

  La función mantienen casi todos los beneficios que presenta la ecuación ReLU; para el desarrollo de redes neuronales profundas se prefiere el uso de la función Leaky ReLU gracias a que sus ventajas en cuestión al entrenamiento y el aprendizaje, logrando mejores resultados con redes sin tantas capas ocultas. Su principal desventaja radica en el parámetro que se integra debido a que implementa una multiplicación que resulta en un mayor consumo de recursos, el cual es propuesto por el usuario y se obtiene a partir de la experimentación, dependiendo totalmente de la tarea a resolver y las características de la red.



(a) Gráfica de la función Leaky ReLU. (b) Paisaje de salida de la función [18].

Figura 2.9: Función Leaky ReLU.

- **Exponential Linear Unit (ELU)**

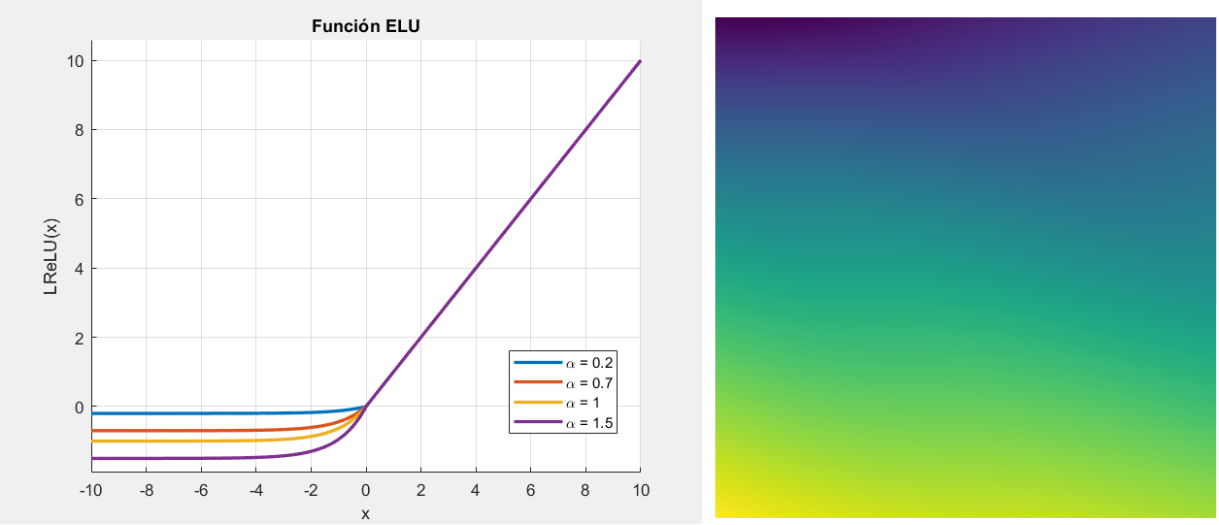
  Gracias a la popularidad de la función ReLU y sus buenos resultados, comenzaron a surgir más funciones que la tomaron como base. En 2015 se plantea la función ELU, que más allá de buscar reducir las desventajas de la función ReLU (como lo hace la Leaky ReLU), espera aumentar los beneficios de cualquier otra función en el entrenamiento. La principal ventaja de la función es que logra alcanzar el aprendizaje más rápido en comparación a las demás funciones, pues con muy

pocas iteraciones se obtienen resultados competitivos, reduciendo la complejidad computacional.

Adicionalmente, la función logra casi eliminar el problema del desvanecimiento del gradiente, gracias a su comportamiento exponencial para las entradas negativas, así como eliminar el problema de "*dying ReLU*". Sin embargo, a pesar de las ventajas importantes que provee la función, su principal defecto es la complejidad matemática de la ecuación, puesto que su implementación conlleva un gran consumo de recursos dificultando la optimización de la misma; aunque su derivada (2.12) hace uso de la propia función.

$$ELU'(s) = \begin{cases} 1, & \text{si } s > 0 \\ ELU(s) + \alpha, & \text{si } s < 0 \end{cases} \tag{2.12}$$

Como se puede observar en la función, contiene un parámetro externo que, de igual forma que en la Leaky ReLU, será definido por el usuario, aunque usualmente lo igualan a 1 [19, 26–28], pero únicamente a través de la experimentación se podrá conocer el valor que dé los mejores resultados. Por lo que su principal desventaja resulta en su gran consumo de recursos, sin embargo, en la actualidad, gracias al crecimiento de la tecnología, comienza a ser una función más atractiva, ya que los recursos mínimos requeridos comienzan a ser más viables.



(a) Gráfica de la función ELU. (b) Paisaje de salida de la función [18].

Figura 2.10: Función ELU.

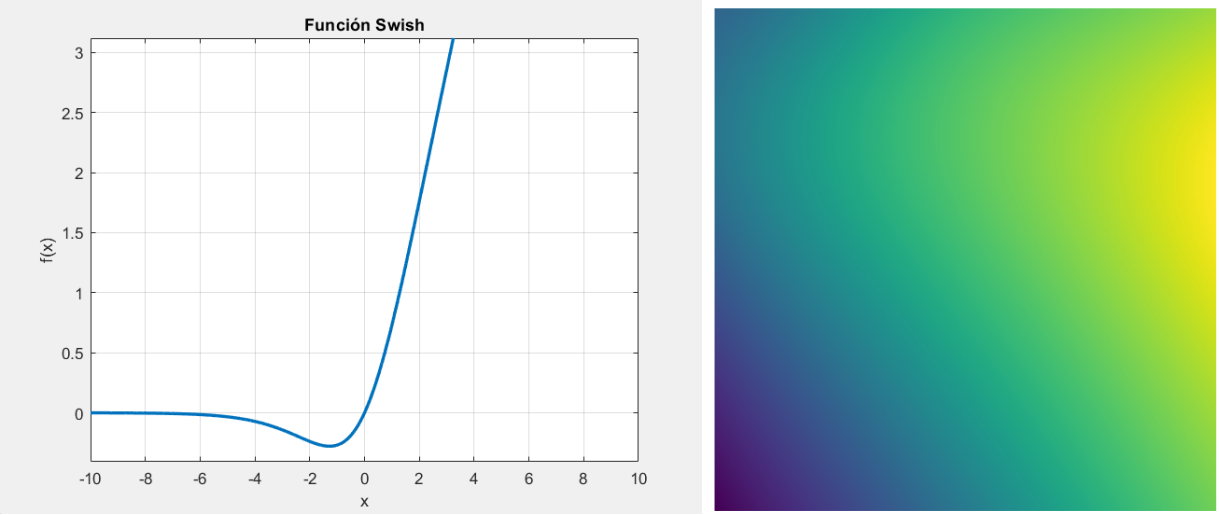
- **Swish**

La ecuación Swish es la primera función híbrida propuesta, siendo la combinación entre la función Sigmoide y la propia entrada. Se plantea en 2017 con el fin de superar a la función ReLU de acuerdo a su rendimiento en el aprendizaje profundo, caracterizándose por utilizar la técnica de búsqueda automática basada en el aprendizaje por refuerzo para calcular la función. La suavidad de la ecuación produce una mejor optimización y resultados más precisos, debido a que

se reduce la sensibilidad a los valores iniciales aleatorios.

Al igual que la función ReLU, se encuentra acotada por debajo y no acotada en los límites superiores, pero se diferencia en su suavidad y su no monotonía, como se observa en la Figura 11b. Su derivada (2.13) tiene el beneficio de contener la propia función, así como la función Sigmoide, por lo que, el problema del desvanecimiento del gradiente se reduce.

$$f'(s) = f(s) + \sigma(s)(1 - f(s)) \tag{2.13}$$



(a) Gráfica de la función Swish. (b) Paisaje de salida de la función [18].

Figura 2.11: Función Swish.

2.4. Métodos de aproximación

Los métodos de aproximación lineal es un conjunto de técnicas matemáticas que permiten asemejar una función compleja o no lineal a una función lineal, esto se realiza debido a que provoca que se simplifique la función inicial y pueda ser analizada de manera más rápida, así como una implementación más sencilla, específicamente en software o hardware, reduce el consumo de recursos, así como también se vuelve más fácil de optimizar, brindando resultados más rápido.

Existen diversos métodos de aproximación lineal muy diferentes entre sí, puesto que cada método hace uso de recursos matemáticos diferentes. A continuación se describirán brevemente los métodos más utilizados con sus ventajas y desventajas, considerando su implementación en hardware.

- **Serie de Taylor**

  La serie de Taylor es una expansión de alguna función en una suma infinita de términos calculados a partir de las derivadas de esa función en un punto específico. Es un método muy eficiente que permite simplificar un gran número de funciones, siendo que la aproximación a la función original es más precisa mientras más términos de la suma se tengan.

$$f(x) = \sum_{n=0}^{\infty} \frac{f^n(a)}{n!}(x-a)^n \tag{2.14}$$

Como se puede observar en la ecuación 2.14, se integra hasta la n derivada de la función, así como fracciones y multiplicaciones, por lo que, a pesar de ser un método bastante robusto, su implementación en Hardware resulta en un gran consumo de recursos, en consecuencia, el beneficio de aproximar la función resulta ineficiente, perdiendo los beneficios de simplificar la función.

- **Segunda curva**

  El método de la segunda curva consiste en modelar el comportamiento de alguna función en dos fases distintas, mediante dos aproximaciones lineales o dos funciones no lineales más sencillas, que juntas capturan un comportamiento general más complejo. Este método permite simplificar fácilmente una función compleja, puesto que es un método flexible que no se encasilla en alguna otra función matemática, esto permite que sea adaptable con un gran número de funciones complejas.

  El principal inconveniente de este método es que mientras más complejo sea el comportamiento de la función, más difícil será simplificar la ecuación. Adicionalmente, el método carece de una precisión, puesto que al aproximar el comportamiento, se pueden perder fácilmente puntos o tramos de la función, o en contraparte, agregarlos.

- **Por partes**

  El método de aproximación lineal por partes (PWL, por sus siglas en ingles) consiste en dividir el comportamiento de una función en segmentos, seleccionando específicos puntos, para posteriormente realizar una interpolación lineal entre tales puntos, obteniendo un conjunto de rectas que componen la función inicial. Este método es uno de los métodos más eficientes, puesto que matemáticamente hablando hace uso del mínimo número de recursos, provocando que su implementación sea óptima.

  El método permite abarcar prácticamente cualquier función, siempre que el número de segmentos no este limitado. La limitación de este método es que se pueden generar discontinuidades en la derivada de la función, afectando el análisis matemático. Además, realizar la selección de puntos es fundamental para obtener una buena aproximación, sin embargo la elección de tales puntos puede resultar subjetivo.

Adicionalmente a los métodos de aproximación lineal matemáticos, se han desarrollado diversos métodos de aproximación computacionales, los cuales también tienen como objetivo simplificar una función compleja o no lineal. La diferencia entre estos métodos son las técnicas que son utilizadas, que se mencionan en el propio nombre. Algunos de los métodos de aproximación computacionales más utilizados son las Tablas de búsqueda (LUT, por sus siglas en ingles), Newton-Raphson, Bisección, etc.

A pesar de que los métodos computacionales brindan buenas aproximaciones, siendo capaces de simplificar cualquier función compleja, las técnicas que se implementan están pensadas para ser óptimas únicamente en software. Por lo que, fuera de ese entorno, difícilmente los métodos son viables ya que se basan en métodos iterativos o la disponibilidad de memoria, que representan un gran consumo de recursos.

2.5. Field Programmable Gate Array (FPGA)

Un Arreglo de Compuertas Programables en Campo (FPGA) es una plataforma de circuitos integrados digitales re configurables eficiente, está constituido por bloques lógicos programables, los cuales se enlazan mediante interconexiones configurables. Los FPGAs son dispositivos de silicio prefabricado que pueden ser eléctricamente programados para implementar cualquier tipo de circuito o sistema digital [29].

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 [30]. Es un dispositivo pensado para que el usuario lo personalice y lo programe de acuerdo a al tarea que busque realizar y sea reprogramado repetidamente. En comparación a otro tipo de dispositivos programables como los Dispositivos de Lógica Programable Compleja (CPLDs, por sus siglas en ingles), un FPGA contiene un mayor número de componentes lógicos que permite implementar sistemas digitales más grandes y complejos.

Gracias al rápido crecimiento de la tecnología de semiconductores, se ha desarrollado una grande gama de FPGAs con una gran diversidad de capacidades, a pesar de ello, los FPGAs funcionan a partir los mismos elementos fundamentales: una tabla de búsqueda (LUT, por sus siglas en ingles), un Flip-Flop D y un multiplexor 2 a 1. Estos elementos conforman una celda lógica típica, la cual a su vez, un conjunto de celdas lógicas constituye un bloque lógico configurable [7].

La arquitectura general de un FPGA se compone de una matriz de bloques lógicos configurables (CLB, por sus siglas en ingles), una red de interconexiones programables, bloques de entrada/salida que rodean los CLB, bloques de memoria, bloques de procesamiento digital de señales (DSP, por sus siglas en ingles), encargados de la aritmética, bloques de transacción de información, un reloj de procesamiento y finalmente, un núcleo de procesamiento embebido, como se puede observar en la Figura 2.12. Una de las ventajas del FPGA es que permite la selección de dos tipos de funciones, *combinacional* o *secuencial*, permitiendo una mayor gama de aplicaciones, así como un eficiente uso de recursos.

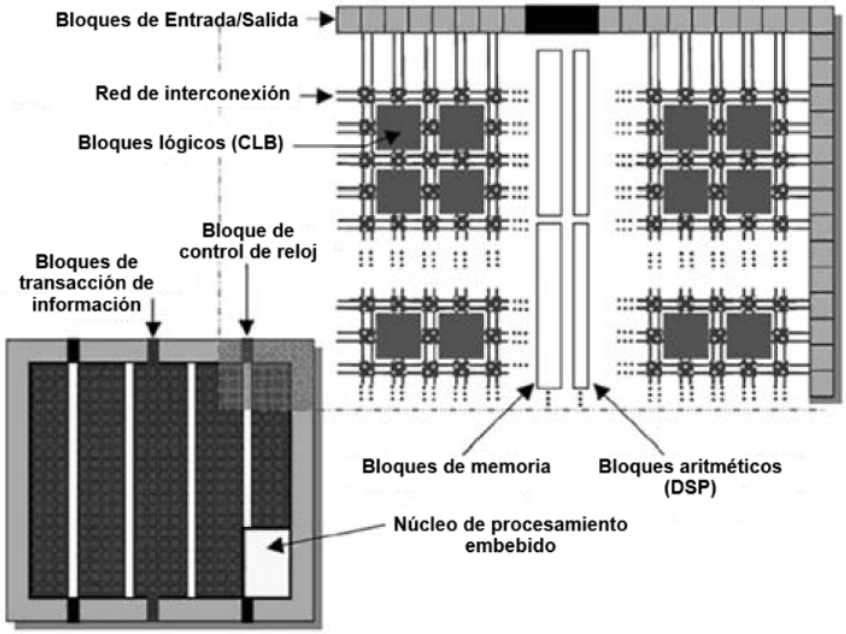


Figura 2.12: Arquitectura general de un FPGA [29].

Capítulo 3

3. Estado del arte

Partiendo de las bases teóricas del presente trabajo, se procede a contextualizar la investigación con el objetivo de comprender los avances teóricos y/o tecnológicos que le conciernen y mostrar cómo se ha abordado el problema hasta el momento. Se espera identificar los avances y tendencias actuales desarrolladas, con el fin de evitar redundancias y reafirmar la justificación del trabajo, al demostrar las áreas de oportunidad, brindando referencias para elegir herramientas, técnicas o estrategias que puedan ser implementadas.

Específicamente hablando de la presente investigación, a pesar del crecimiento exponencial de los circuitos integrados, es importante conocer las limitaciones actuales y las alternativas empleadas para el uso de ANN en una gama de proyectos más amplia. Por ello, de la investigación reciente, se pretende analizar aquellas arquitecturas y modelos innovadores que han permitido realizar una ANN en hardware de recursos lógicos limitados, así como su proceso de entrenamiento. Posteriormente en el capítulo, se analizará y comparará el impacto que provoca la implementación de diferentes ecuaciones matemáticas como funciones de activación, considerándose como una vía de optimización de la propia red neuronal. Finalmente se conocerá el impacto de implementar diferentes funciones en una misma red.

3.1. Entrenamiento

De acuerdo con el Estado del Arte, el entrenamiento realizado para redes neuronales que se implementan en FPGA se desarrolla en CPU de propósito general. Esto se debe porque el proceso de entrenamiento es exigente en cuanto a recursos y consumo energético, representa un alto grado de complejidad a nivel de descripción de hardware y existe una gran limitación de precisión numérica. Es por ello, que se desarrolla un enlace entre CPU y FPGA, en el cual se comunicarán los datos de configuración, de entrada, así como pesos y umbrales ya entrenados [1, 7, 14, 31].

A este tipo de arquitectura se le conoce como un acelerador típico [14], un sistema que consiste en un CPU anfitrión y un FPGA. Tanto el anfitrión como el FPGA pueden trabajar con su propia memoria externa y acceder a los datos de memoria del otro a través de la conexión. La mayoría de los diseños de redes neuronales implementan un acelerador en la parte del FPGA y controlan el acelerador con el software del anfitrión (CPU o GPU).

Con respecto a la Figura 3.1, la parte lógica de todo el sistema se encuentra de color azul. La CPU anfitrión emite una carga de trabajo o comandos a la parte lógica del FPGA y supervisa su estado de funcionamiento, siendo en esta última sección donde normalmente se implementa un controlador para comunicarse con el anfitrión y generar señales de control para todos los demás módulos del FPGA. El controlador puede ser una máquina de estado o un decodificador de instrucciones, dependiendo de la estructura del envió de datos que se desee implementar.

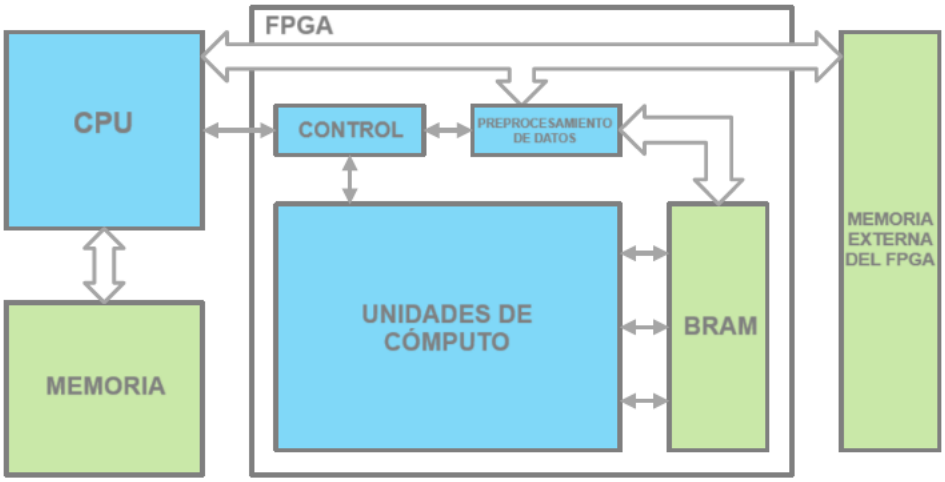


Figura 3.1: Arquitectura típica de un acelerador basado en FPGA [14].

Considerando el diseño específico para redes neuronales artificiales, el CPU llevará a cabo el entrenamiento de la red, y será en su memoria correspondiente que se almacenarán los parámetros requeridos, tal como los pesos y umbrales entrenados, que posteriormente serán transferidos al FPGA. La segunda sección del acelerador consiste en los bloques integrados del FPGA, los cuales conforman la red neuronal que realizará la clasificación. Dependiendo del tamaño de la red, se usará la memoria embebida del FPGA o una memoria externa, si así se requiere. Las unidades de cómputo corresponden a las neuronas y los cálculos matemáticos como la obtención de las entradas ponderadas, mientras que el bloque de control coordina y sincroniza a los elementos de la unidad de cómputo, así como la comunicación entre CPU y FPGA [7].

3.2. Implementación en Hardware

A pesar de tenerse un avance significativo de las redes neuronales artificiales, en cuestión a su robustez y complejidad, la instrumentación de las mismas se ha visto rezagada. Actualmente, el hardware en el que tradicionalmente se implementan es en CPU y GPU, tecnología con una alta capacidad de procesamiento de datos en un lapso de tiempo corto, pero con un alto consumo de energía, de espacio y costo. Es por ello que en los últimos años, los FPGA se han convertido en una solución prometedora para la aceleración de algoritmos. En comparación con las plataformas CPU, GPU y DSP, para las que el software y el hardware se diseñan de forma independiente, los FPGA permiten a los desarrolladores

implementar solo la lógica necesaria en el hardware según el algoritmo de destino [14].

Como parte de una arquitectura innovadora, en el 2012 con el artículo [32], se presenta por primera vez el uso de un diseño de una FFNN con el objetivo de hacer el mínimo uso de recursos lógicos en FPGA, así como un tiempo menor de ejecución. Establece una nueva arquitectura pensada en el aprovechamiento de la paralelidad del FPGA, haciendo uso de diferentes técnicas seriales, parcialmente paralelas y completamente paralelas. La implementación de tales técnicas es a partir de la distribución de datos de la capa anterior a cada neurona de la siguiente capa, se realiza simultáneamente mediante un multiplexor y el procesamiento de datos en todas las neuronas de la capa también se realiza en paralelo.

Reduce el uso de recursos lógicos empleando bloques aritméticos de activación rápida y una aproximación lineal por partes para la función de activación. Además, demuestra la importancia de la selección de formato numérico y su impacto en la red, seleccionando un formato de punto fijo. Como resultado, en el dispositivo Stratix III EP3S50F484C2, se desarrolló una red neuronal de dos entradas con dos neuronas en la capa oculta y una neurona de salida, haciendo uso del 14.8 % de los elementos lógicos.

Posteriormente, el Estado del Arte sitúa en 2021 el desarrollo de un nuevo diseño, con el artículo [2], que plantea una nueva estructura de la red que permita optimizar más el uso de recursos, declarando un sólo módulo de la función de activación por el que cada neurona procesará su información, a partir de un procesamiento serial-paralelo. Esta nueva arquitectura representa un ahorro significativo de recursos lógicos puesto que es la función de activación (teóricamente presente en cada neurona) el bloque que conlleva más consumo de recursos por los operandos matemáticos, permitiendo realizar redes neuronales más grandes, con respecto al número de neuronas y capas, puesto que, se implementa además la capa neuronal más grande únicamente. Como resultado, en el dispositivo Xilinx XC7z020clg484-1 Artix-7, se obtiene un 99 % de precisión en la clasificación de imágenes de números manuscritos, haciendo uso únicamente del 1.09 % de los elementos lógicos.

Es en 2024, con la tesis [7], se lleva a cabo este nuevo diseño en el que se implementa además más técnicas de optimización, obteniendo una arquitectura que explota en su mayor capacidad la paralelidad que ofrece la FPGA, promoviendo la reutilización de neuronas, la implementación de un esquema serial-paralelo, la descripción de un sólo bloque de la función de activación, la descripción de la ecuación por aproximación lineal por partes (PWL), la selección de un formato numérico de punto fijo y el aprovechamiento de los recursos embebidos del FPGA para el ahorro de recursos. Como resultado, en el dispositivo Cyclone IV EP4CE6F17C8, se logra una ANN que obtiene un 89.9 % de precisión en la clasificación de imágenes de números manuscritos, haciendo uso del 36 % de los elementos lógicos.

3.3. Funciones de activación

Conforme a la Sección 2.2.1, se comprende la importancia de las funciones de activación en las redes neuronales y el papel tan fundamental que conlleva su elección, debido a que afecta directamente el rendimiento del modelo, la eficiencia computacional y la estabilidad del entrenamiento. Es por ello, que en el Estado del Arte se han encontrado diversos artículos que realizan un análisis del impacto de la implementación de diferentes funciones de activación para resolver tareas específicas, obteniéndose una gran diversidad de resultados que varía de acuerdo con:

- Las funciones que se están comparando
- Las características específicas de la red neuronal y su profundidad
- Al problema que se esté intentando resolver
- El conjunto de datos que se utilice para resolverlo

Cabe resaltar lo mencionado en el artículo [15], el cual concluye que es importante tener en cuenta que la elección de la función de activación debe basarse en el problema específico y el conjunto de datos que se utilice. Además, considera pertinente experimentar con diferentes funciones de activación para ver cuál se desempeña mejor para un problema en particular. Como por ejemplo, para un problema de tipo regresión, las funciones de activación lineales funcionan bien, sin necesidad de agregar mayor complejidad. De acuerdo al artículo, menciona que para clasificación binaria las mejores funciones son la Sigmoide, así como para problemas de clasificación de múltiples secciones; mientras que, la función Softmax ofrece una buena precisión para problemas de múltiples clases.

En contra parte, el artículo [18] menciona que de acuerdo a los paisajes de salida de cada función, las mejores funciones de activación son la SELU, Swish y Mish debido a que tienen la salida más lisa, por lo que, teóricamente serían más fácil de optimizar. Sin embargo, en los artículos [1], [2], [5], [33], [34] y [20] los cuales realizan redes neuronales con diferentes características entre sí, y que además resuelven diferentes problemas, obtienen conclusiones diferentes de cuál es la mejor función de activación, concluyéndose que la mejor función varía de acuerdo al contexto.

Realizando una investigación profunda, buscando aquellos artículos que realicen una comparativa entre diferentes funciones de activación, se obtiene de manera sintetizada la Tabla 3.1, la cual, como se puede observar, contiene un gran número de funciones de activación, siendo que las presentadas ahí, son únicamente las nombradas en los últimos años, existiendo aún todavía más. Es por ello la importancia de seleccionar aquellas funciones que sean representativas del Estado del Arte, es decir, seleccionar las funciones más mencionadas, esto debido a que se espera seleccionar las ecuaciones que persisten a través de los años gracias a los beneficios que conlleva implementarlas o las funciones nuevas que logran competir o superar a aquellas ecuaciones que ya son conocidas por sus ventajas. En otras palabras, se esperan encontrar las mejores funciones de activación del Estado del Arte.

Función de activación	**Fórmula**	**Artículos del Estado del Arte**
Lineal	$f(s) = as + b$	[1], [16], [10], [35]
Escalón	$h(s) = \begin{cases} 1, & \text{si } s > 0 \\ 0, & \text{si } s \leq 0 \end{cases}$	[15], [24]
Sigmoide	$\sigma(s) = \frac{1}{1+e^{-s}}$	[1], [18], [15], [20], [16], [10], [35], [17], [26]
Tangente hiperbólica	$tanh(s) = \frac{e^{s}-e^{-s}}{e^{s}+e^{-s}}$	[1], [18], [15], [20], [10], [35], [17], [26]
Softmax	$softmax(s) = \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}}$	[15], [10], [17],
Softplus	$softplus(s) = In(1 + e^{s})$	[1], [20]
Softsign	$softsign(s) = \frac{s}{1+\|s\|}$	[15], [20]
Swish	$Swish(s) = s \cdot \sigma(s)$	[5], [18], [28]
Mish	$Mish(s) = s \cdot tanh(log(1 + e^{s}))$	[5], [18]
Serf	$Serf(s) = s(In(1 + e^{s}))$	[5]
Nipuna	$Nipuna(s) = max(\frac{s}{1+e^{-(\beta s)}}, s)$	[5]
Collapsing Linear Unit (CoLU)	$CoLU(s) = \frac{s}{1-se^{-(s+e^{s})}}$	[5]
Rectifier Linear Unit (ReLU)	$ReLU(s) = \begin{cases} s, & \text{si } s > 0 \\ 0, & \text{si } s \leq 0 \end{cases}$	[1], [5], [18], [15], [34], [20], [16], [10], [35], [24], [17], [26], [28]
Leaky ReLU	$LReLU(s) = \begin{cases} s, & \text{si } s > 0 \\ \alpha s, & \text{si } s \leq 0 \end{cases}$	[18], [15], [34], [35], [17], [26], [28]
Parametric Leaky ReLU	$PReLU(s) = \begin{cases} s, & \text{si } s > 0 \\ \lambda s, & \text{si } s \leq 0 \end{cases}$	[18], [15], [24]
Exponential Linear Unit (ELU)	$ELU(s) = \begin{cases} s, & \text{si } s > 0 \\ \alpha(e^{s} - 1), & \text{si } s \leq 0 \end{cases}$	[18], [15], [34], [20], [35], [24], [17], [26], [28]
Scaled Exponential Linear Unit (SELU)	$SELU(s) = \gamma \begin{cases} s, & \text{si } s > 0 \\ \alpha(e^{s} - 1), & \text{si } s \leq 0 \end{cases}$	[18], [34], [20], [24]
Exponential Linear Squashing (ELiSH)	$ELiSH(s) = \begin{cases} \frac{e^{s}-1}{e^{-s}+1}, & \text{si } \geq 0 \\ \frac{s}{e^{-s}+1}, & \text{si } s < 0 \end{cases}$	[24]

Tabla 3.1: Funciones de activación del Estado del Arte.

A partir de la Tabla 3.1, se analiza que de acuerdo al número de artículos que mencionan y hacen uso de una función se pueden obtener cinco ecuaciones destacables, las cuáles, ordenadas de la más nombrada a la menos nombrada, son:

1. Rectifier Linear Unit (ReLU)
2. Sigmoide
3. Exponential Linear Unit (ELU)
4. Tangente hiperbólica
5. Leaky ReLU

Es a partir de estas cinco funciones representativas del Estado del Arte que se decide realizar la implementación en hardware en el presente trabajo. Sin embargo, de acuerdo a los artículos mencionados en la Tabla 3.1, cabe resaltar la función *Swish*, la cual en cada artículo que es mencionada se posiciona entre las tres mejores funciones, además que representa la primera función híbrida, por lo que, se considera una función de importancia. Es por lo anteriormente mencionado que esta es la sexta función que se implementará en el trabajo.

3.4. Implementación de diferentes funciones de activación

Como se ha visto a lo largo del trabajo, las funciones de activación impactan de gran manera las redes neuronales proveyendo cada función diferentes beneficios y desventajas. Es a partir del análisis de cada función, que en el Estado del Arte, surgió una nueva linea de investigación que propone el uso de diferentes funciones de activación en una misma red neuronal, variando la función entre la capa oculta y la capa de salida. De primera impresión, se podría pensar que la unión de dos funciones de activación en una misma red puede proveer lo mejor de dos mundos, sin embargo, la experimentación demostró lo contrario.

De acuerdo con los artículos [1] y [16], se realizan diferentes combinaciones de distintas ecuaciones entre la capa oculta y la de salida, uniendo funciones lineales y no lineales o no lineales con no lineales, concluyendo que la red que arrojó los mejores resultados en cuanto a precisión fue en aquellas redes en las que se utilizó una misma función para todas las capas, por lo que se descarta la propuesta de combinar diferentes funciones en una misma ANN.

Continuando con la investigación, centrándonos en la implementación de las funciones, se procedió en su análisis ya no sólo en una ANN en software, sino en aquella instrumentación en hardware, específicamente hablando en dispositivos FPGA. Las redes neuronales son modelos que han existido desde hace algunas décadas, y es a partir del paso de los años que su complejidad y robustez ha crecido exponencialmente, por lo que la idea de instrumentalizar una red y su función se ha realizado desde 1989 con el artículo [36], realizándose la función Sigmoide en una Integración a muy gran escala (VLSI, por sus siglas en inglés), el cual es un proceso de creación de un circuito especializado, diferente de

gran manera al desarrollo en FPGA, pero establece el principio de la implementación de la función a partir una aproximación lineal por partes.

Posteriormente en 1997, con el artículo [37], se realiza tal aproximación en FPGA. Es a partir de la instrumentación de la función Sigmoide, que se comenzó a implementar otras funciones fueron en FPGA como la tangente hiperbólica [1], así como la función ReLU, la cual no conlleva ninguna aproximación debido a la simplicidad de su fórmula. Sin embargo, estas son las pocas funciones que se han implementado, la función ELU, Leaky ReLU y Swish, son ecuaciones que hasta el momento no se han instrumentado en FPGA, y por consecuente, no existe su aproximación lineal por partes, siendo este el método de aproximación que constantemente ha demostrado ser el más eficiente en cuestión a recursos y precisión.

Capítulo 4

4. Diseño preliminar

Tras haber definido los fundamentos teóricos y el contexto actual de la presente investigación, es posible presentar un diseño preliminar, contando ya con las herramientas necesarias. Se busca abordar las especificaciones generales de la arquitectura que se propone, considerando aspectos clave como el propósito central, los objetivos y la dirección del trabajo de investigación, estableciendo un panorama más específico del diseño de la red neuronal artificial, las funciones de activación, la selección de técnicas de optimización y su integración en hardware.

A lo largo de este capítulo, se proporciona una descripción detallada del flujo de diseño, incluyendo las especificaciones del problema, sus limitaciones y su alcance, así como la metodología a seguir, que se puede observar en la Figura 4.1, que se desarrollará en lo que resta del trabajo. Este análisis permitirá justificar las decisiones tomadas y sentar las bases para la validación del Diseño a detalle en el capítulo posterior.

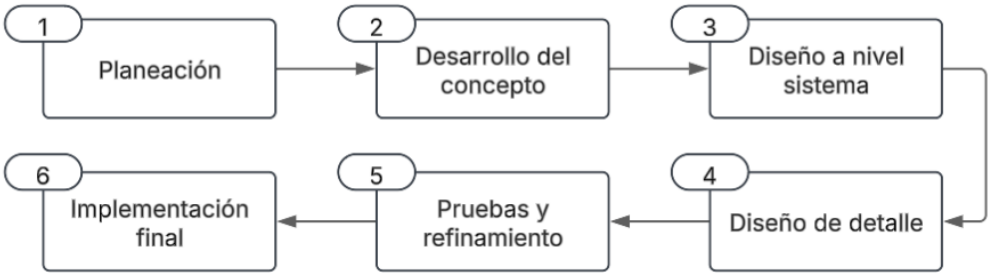


Figura 4.1: Etapas de la metodología de diseño.

4.1. Declaración de la misión

El presente trabajo de investigación tiene como propósito principal la instrumentación en FPGA de las primordiales funciones de activación del Estado del Arte, las cuales cumplan con un diseño que tenga por objetivo la eficiencia y reducción de consumo de recursos, asegurando que sean implementadas con un equilibrio adecuado entre precisión y velocidad. Para ello, de acuerdo al Estado del Arte, se realizará una aproximación lineal por partes para aquellas funciones de las cuales no existe tal aproximación, para posteriormente implementarlas en FPGA optimizando su instrumentación y realizando un análisis a su consumo de recursos.

Posteriormente, como una segunda misión, se establece el desarrollo de una FFNN re-configurable, a partir del diseño mencionado en el Estado del Arte, en el cual se describe una arquitectura innovadora. Por lo que se implementará una FFNN re-configurable que explote en su mayor capacidad la paralelidad que ofrece la FPGA, es decir, que presente un ahorro significativo de recursos, que sea fácil de reconfigurar y provea la misma precisión que se obtiene a nivel Software. Esto con el fin de probar las diferentes funciones en la red, comparando su desempeño instrumentadas en FPGA con la red programada en Python, analizando sus resultados de acuerdo con la clasificación correcta de números manuscritos.

4.2. Desarrollo de concepto

Partiendo de la declaración de la misión, se tiene claro lo que se desea realizar, por lo que el desarrollo del concepto especificará la propuesta a mayor detalle. De esta forma, se analizarán las necesidades y limitantes con el fin de establecer una definición exacta de lo que se realizará.

4.2.1. Identificación de necesidades

Como primer paso en el desarrollo de concepto es necesario plantear las necesidades que se requieren cumplir de acuerdo a los objetivos establecidos, así como su prioridad. Es a partir de estas necesidades, y su relevancia, que se podrá establecer un concepto que se alinee a la Tabla 4.1, en la cual se en listan tales requerimientos.

Prioridad	**Necesidades**
1	Bajo consumo de recursos Bajo consumo de energía Alta velocidad de respuesta
2	FFNN re-configurable Alta precisión de clasificación
3	Robustez Escalabilidad

Tabla 4.1: Necesidades y su relevancia.

4.2.2. Especificaciones objetivo

Prosiguiendo con el desarrollo de concepto, de acuerdo con las necesidades identificadas, es posible generar una lista de especificaciones de nivel técnico con el fin de cuantificar esas necesidades en el desarrollo del proyecto. Las cualidades a las que se aspiran se encuentran definidas por el trabajo previo a la instrumentación, la implementación en Python, que dictamina el error esperado por cada aproximación de cada función, así como los parámetros necesarios con los que debe de contar la FFNN.

A continuación se enlistan las métricas que se deberán contemplar en el diseño a nivel sistema, esto con el fin de obtener una comparación justa entre la implementación en Software y en Hardware.

- Las imágenes de prueba tienen un tamaño de 28x28 pixeles, un total de 784 pixeles.
- Un total de 3 capas.
- La implementación de 6 funciones de activación.
- Un error absoluto de aproximación menor al 5 %.
- Un tiempo de reconocimiento menor a 1 segundo.

4.2.3. Definición del concepto

A partir del entendimiento de la misión del presente trabajo, sus necesidades y especificaciones, se puede proceder finalmente con la definición detallada del concepto.

Se propone un sistema de reconocimiento de imágenes de números manuscritos instrumentado en FPGA, clasificación desarrollada a partir de una FFNN re configurable, implementando un diseño que optimice los recursos, así como contemple las especificaciones objetivo. Por consecuente la FPGA fungirá como un acelerador de la FFNN, mientras que el entrenamiento se desarrollará en CPU, transmitiéndole al FPGA los datos entrenados, así como de configuración.

4.3. Diseño a nivel sistema

Una vez establecida la definición del concepto, se puede iniciar el desarrollo de su arquitectura, la cual incluye la identificación y descripción de sus elementos principales y subsistemas clave. Para una mejor planificación de la propuesta, se seleccionará un diseño *Top-Down*, el cual es un proceso en el cual se captura un idea en un alto nivel de abstracción, para posteriormente partir de tal descripción a un nivel de detalle más específico, o visto de otra forma, ramifica el problema en sub problemas hasta que sean sencillos de tratar [38]. La ventaja de este tipo de método es que incrementa la productividad del diseño para proyectos pensados en FPGA, incrementa su reutilización y genera una rápida detección de errores.

Como se ha visto a lo largo de la planificación, la misión del trabajo se divide en dos grandes tareas muy diferentes entre sí, pero que se complementan: la creación de un diseño para algunas funciones de activación y su total instrumentación, así como el desarrollo de una red neuronal *Feed Foward*. De modo que se analizará la primera entidad de abstracción del concepto, comprendiendo su relación, para posteriormente analizar cada tarea de acuerdo a sus entidades de abstracción.

Con forme a la Figura 4.2, se puede observar el primer nivel de abstracción de la propuesta, que establece el FPGA como el acelerador de una red neuronal, el cual realizará únicamente la clasificación de la FFNN. Además, se encontrará comunicado con un CPU, que dispone del entrenamiento y la

configuración de la propia red. Es a partir de este nivel de abstracción que se puede dividir el problema en dos etapas, el entrenamiento y la clasificación, los cuales conllevan diferentes análisis y procesos.

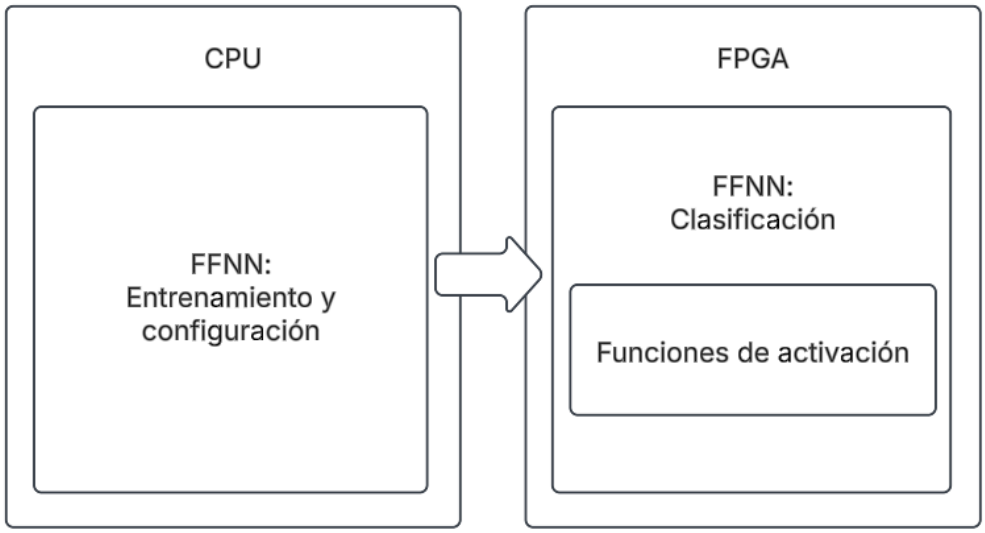


Figura 4.2: Primer nivel de abstracción de la propuesta.

La primera etapa del trabajo consta de realizar una FFNN en Software que permita otorgar a la red en FPGA los parámetros entrenados, como los pesos y umbrales, así como una interfaz de usuario que permita manipular la configuración de la red. El principal objetivo de esta etapa es desarrollar aquellos procesos que no se pueden llevar a cabo en una FPGA por su complejidad, como lo es el entrenamiento que conlleva métodos matemáticos basados en varias iteraciones. Por otro lado, se encuentra la red instrumentada en Hardware, la segunda etapa del trabajo, la cual recibirá los parámetros entrenados y de configuración, con la finalidad de obtener una FFNN lista para ser probada, que clasifique las imágenes que recibe por entrada, por lo que se podrá obtener el desempeño de la red.

4.3.1. FFNN en Software

Esta etapa, como se menciono anteriormente, tiene por finalidad realizar el entrenamiento y establecer la configuración de la red. Por lo que, teniendo claro los objetivos, se procede a plantear el segundo nivel de abstracción de este subproblema. De acuerdo a la Figura 4.3, esta unidad de entrenamiento consiste en una interfaz gráfica de usuario (GUI, por sus siglas en inglés), y en la FFNN desarrollada en software. Como se puede observar en la imagen, la GUI estará diseñada para que el usuario ingrese los datos de configuración que definen la red, que bien estos parámetros dependen del problema a resolver y es a partir de la prueba y error que se obtiene la mejor configuración.

Posteriormente, estos datos de configuración se ingresaran a la FFNN, para que realice el entrenamiento y se logre obtener un archivo que contenga los pesos y umbrales de cada neurona por capa, archivo que después se ingresará a la GUI. De modo que, una vez cargados los datos entrenados en la interfaz, se habilitará la opción que permita la transmisión de datos, finalizando la etapa de entrenamiento con la recepción de tales datos en la FPGA.

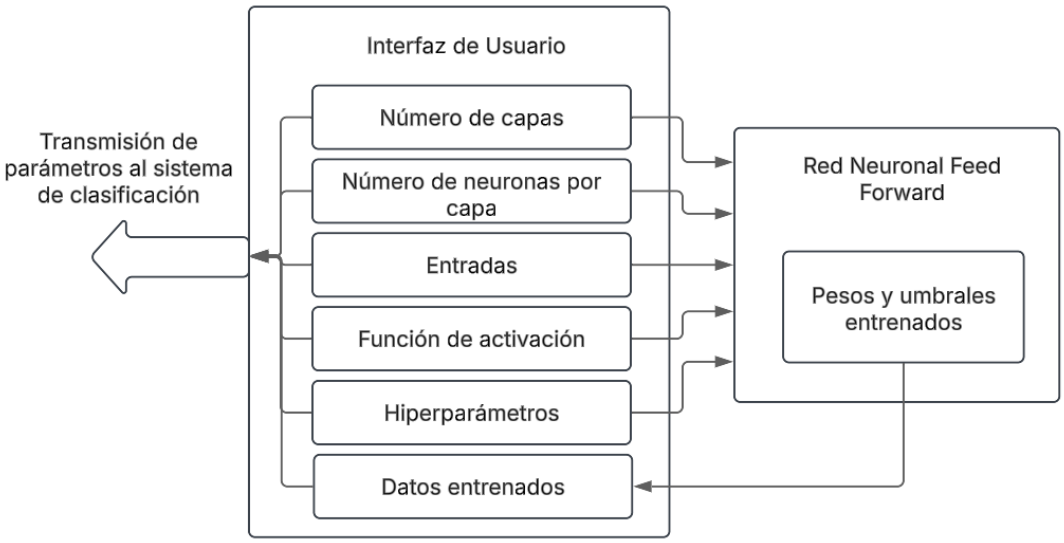


Figura 4.3: Segundo nivel de abstracción de la etapa en CPU.

4.3.2. FFNN en Hardware

La segunda etapa de la propuesta consiste en la evaluación de la FFNN en FPGA, puesto que a partir de la recepción de los datos entrenados, así como de configuración, la red ya será capaz de clasificar cualquier imagen que se le ponga como entrada, repitiendo este proceso el número de veces que el usuario desee. Por ello, como se ha mencionado con anterioridad, el diseño que se llevara a cabo para la construcción de la FFNN se basa en el modelo encontrado en el Estado del Arte, el cual establece una arquitectura innovadora que tiene por objetivo optimizar recursos.

Comprendiendo así que el proceso más complejo se encuentra en esta etapa, que abarca la instrumentación de la FFNN, se analizará de acuerdo a la metodología planteada de un inicio, por su nivel mas alto de abstracción. De primera instancia será importante retomar la Figura 2.2, en la cual se muestra el modelo más básico de una neurona, se observan los componentes que la conforman: las entradas ponderadas que ingresarán a la función de activación y se obtiene una salida. Por lo que, trasladando la idea a su instrumentación, se puede observar la Figura 4.4, se contemplan las entradas, los pesos y umbrales, un bloque de multiplicación y acumulación (MAC), que se encargará de obtener la suma ponderada y la función de activación que producirá la salida, teniendo finalmente una neurona instrumentada.

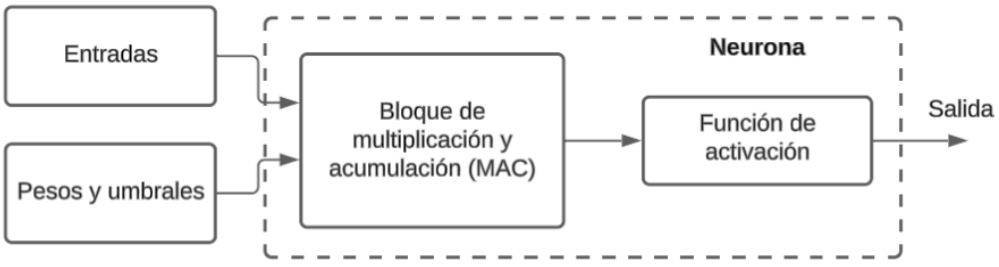


Figura 4.4: Modelo de una neurona instrumentada en Hardware.

Trasladando la neurona instrumentada a una red completa, se deberán tener las consideraciones que se mencionan en el Estado del Arte, el uso de un solo bloque de la función de activación, la reutilización de neuronas y la implementación de un esquema serial-paralelo. A partir de estos puntos, como se puede ver en la Figura 4.5, se construye una FFNN instrumentada, la cual contiene bloques adicionales a una sola neurona para un control exacto.

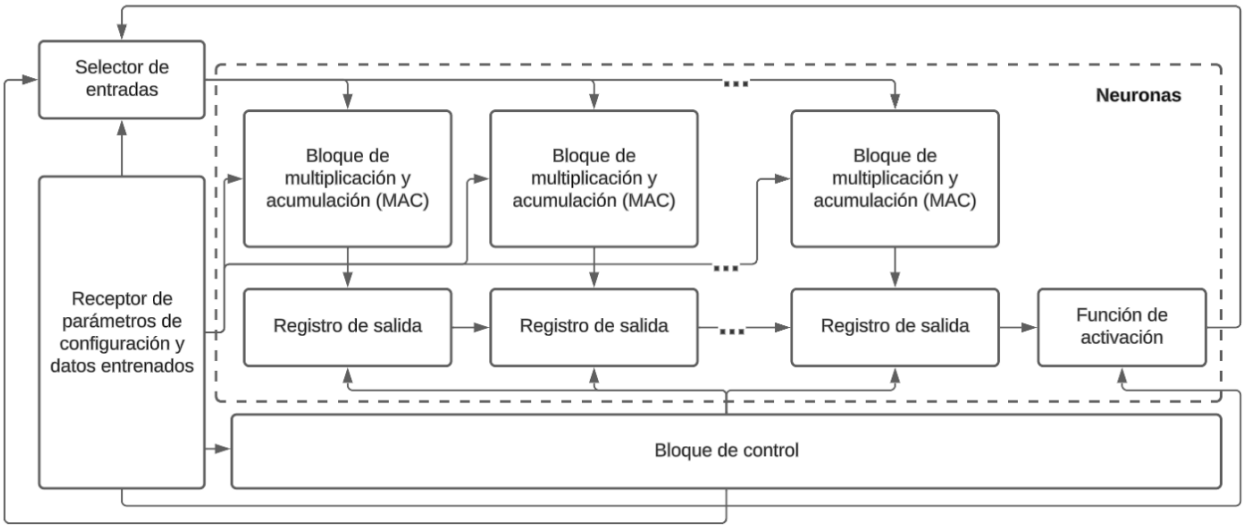


Figura 4.5: Segundo nivel de abstracción de la etapa en FPGA.

A continuación se explicara la función de cada bloque que conforma la red neuronal instrumentada en FPGA.

- **Receptor de parámetros de configuración y datos entrenados**

  Este bloque será el encargado de, como su nombre lo indica, recibir la información del CPU, que contiene los parámetros de configuración y los datos entrenados. En consecuente, el bloque deberá estar conectado a los bloques MAC, para cargar los datos entrenados en ellas, así como con el bloque de control que guardará los datos de configuración

- **Selector de entradas**

  Este bloque se encargará de seleccionar con que entrada las neuronas trabajaran, puesto que puede ser la primer entrada dada por el usuario o la salida de la función de activación, diferenciando a las neuronas de la entrada con las de otras capas, que reciben el resultado de la capa anterior.

- **Bloque de multiplicación y acumulación (MAC)**

  Este bloque obtiene la suma ponderada, multiplica la entrada con su peso, para posteriormente sumarlas todas, junto con el umbral.

- **Registro de salida**

Los registros de salida se integran al diseño en un esquema serial-paralelo que permite guardar la suma ponderada, para recorrerse entre todos los registros y así permitir que a la función sólo entre una suma a la vez, sin perder datos en el proceso.

- **Función de activación**

  En este bloque se encontrarán las funciones de activación que se implementarán, recibiendo la suma ponderada, y obteniendo la salida activada. Se analizará a mas detalle este bloque en la Sección 4.3.3.

- **Bloque de control**

  Este bloque controlará el flujo de la red, generando las señales necesarias para una correcta selección de entrada, para un correcto avance de los registros, así como también establecerá que neuronas se activaran de acuerdo a los parámetros de configuración.

De acuerdo a cada bloque se puede entender de mejor forma el funcionamiento de esta arquitectura da una FFNN instrumentada. *A grosso modo*, se tendrán n número de neuronas establecidas, que se reciclarán en cada capa, de modo que se logra el ahorro de recursos y se aprovecha en su máxima capacidad la paralelidad del FPGA, al permitir ejecutar varias neuronas al mismo tiempo sin establecer una función de activación para cada neurona.

4.3.3. Funciones de activación

Una vez establecida la arquitectura de la FFNN en su mayor nivel de abstracción, se logra comprender el lugar del bloque de la función de activación dentro de la red. De acuerdo con la misión del presente trabajo, será en este bloque en dónde se encontrarán las seis funciones de activación del Estado del Arte, aunque será sólo una función la que se utilizará. Por ello, como se observa en la Figura 4.6, se establece su nivel de abstracción, bloque en el que se encontrarán las seis funciones y un selector de la función. El dato que indica la función seleccionada lo proporcionará el usuario por medio de la GUI, así como el valor del hiperparámetro si es que se requiere.

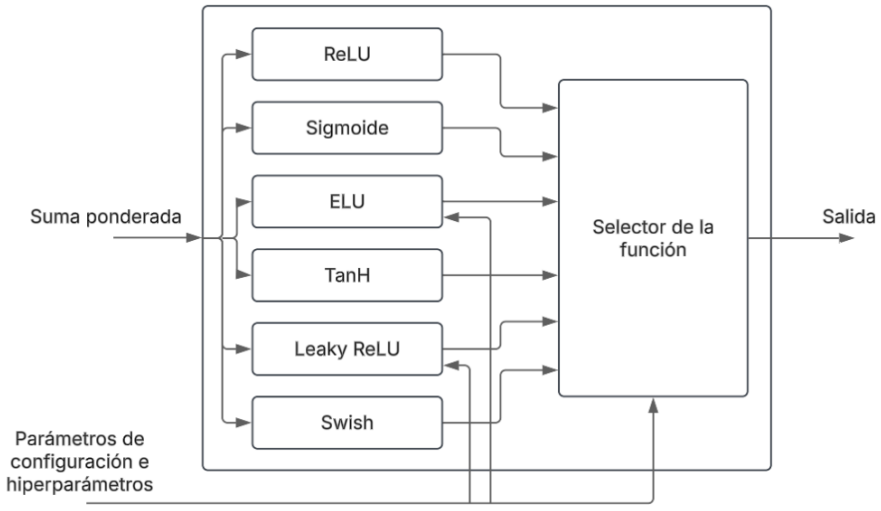


Figura 4.6: Tercer nivel de abstracción de las funciones de activación.

Capítulo 5

5. Diseño de detalle

Continuando con la metodología, el diseño de detalle describirá de manera específica y técnica cómo se desarrollará el proyecto propuesto, abarcando el diseño a nivel sistema, donde se contemplará ya la selección de los componentes. El propósito principal es proporcionar una guía clara sobre cómo se realizará el trabajo y qué métodos, herramientas y procesos se usarán, considerando lo establecido en el diseño preliminar.

5.1. FFNN en Software

Acorde con el diseño preliminar y el Estado del Arte, se realizó una red neuronal *Feed Foward* en Software con el fin de llevar a cabo el entrenamiento de la red que posteriormente se encontrará en Hardware. El desarrollo de la FFNN fue realizado en Python, lenguaje de programación comúnmente ocupado en la investigación relacionada con redes neuronales artificiales, siendo en este lenguaje en el que se desarrolla todo el estudio de las funciones de activación. Se realizó con un formato numérico de punto flotante de 32 bits.

Realizada la FFNN en Python es pertinente realizar un análisis comparativo acerca de los mejores parámetros de configuración para la instrumentación de la red, puesto que, a pesar de que se realizará una red re-configurable, se seleccionará de antemano aquellos parámetros que brinden los mejores resultados. Por ello, como se puede observar en la Tabla 5.2, se prueba con un diferente número de neuronas en la capa oculta, para encontrar un balance entre la menor cantidad de neuronas y la mejor precisión de clasificación. Estas pruebas se realizaron considerando lo siguiente:

Configuración	**Valor**	**Origen**
Número de entradas	784	Número de pixeles (28x28)
Número de neuronas en la capa de salida	10	Número de clases (0-9)
Épocas	5-20	Valor empírico
Hiperparámetro de ELU	$\alpha = 0.75$	Valor empírico
Hiperparámetro de Leaky ReLU	$\alpha = 0.015625 = 2^{-6}$	Valor empírico

Tabla 5.1: Parámetros de configuración

Función de activación	**Tasa de aprendizaje**	**Neuronas en la capa oculta**	**Precisión de clasificación**
ReLU	0.001	10	89.27 %
		20	93.30 %
		30	94.56 %
		40	95.14 %
		50	95.31 %
Sigmoide	0.1	10	90.97 %
		20	93.57 %
		30	94.80 %
		40	94.94 %
		50	95.28 %
ELU	0.001	10	80.75 %
		20	90.51 %
		30	91.02 %
		40	95.20 %
		50	95.02 %
TanH	0.01	10	91.26 %
		20	92.60 %
		30	93.57 %
		40	93.83 %
		50	94.62 %
Leaky ReLU	0.001	10	89.36 %
		20	93.50 %
		30	94.56 %
		40	94.89 %
		50	94.06 %
Swish	0.001	10	85.03 %
		20	90.11 %
		30	90.57 %
		40	92.97 %
		50	93.04 %

Tabla 5.2: Rendimiento de la FFNN en software con diferentes parámetros de configuración.

Como se observa en la Tabla 5.2, cada función brinda una diferente precisión con respecto a los cambios de los parámetros de configuración de la ANN. De inicio, es importante tomar en cuenta que el aumento del número de neuronas en la capa oculta no está modificando la profundidad de la red, por lo que no se tiene un aprendizaje profundo, de modo que las ventajas o desventajas de cada función sobre este tipo de aprendizaje no son observables en esta aplicación.

De acuerdo con la Tabla, se comprende que mientras menos neuronas haya, la precisión será menor, pero no es un crecimiento lineal, puesto que con 50 neuronas, se comienza a ver una reducción de precisión, esto se observa debido a que comienza a existir un sobre entrenamiento, que se puede evitar o reducir integrando menos épocas del entrenamiento, aunque la precisión de clasificación empieza a mejorar o empeorar a pasos muy pequeños, estancándose en un porcentaje.

Finalmente, considerando las prioridades establecidas en el presente trabajo, se da preferencia al ahorro de recursos que a la mejor precisión de clasificación, es por ello que, con el exclusivo objetivo de demostrar el funcionamiento de la red, se probará y comparará el resultado obtenido con 30 neuronas en la capa oculta en software, con su implementación en Hardware. Sin embargo, es importante recordar que la FFNN en FPGA es re-configurable, por lo que, si se deseara probar con más neuronas es posible.

5.1.1. Transmisión de parámetros de entrenamiento, configuración y entradas

La transmisión de datos se realizó a partir de una interfaz gráfica de usuario, la cual se puede observar en la Figura 5.1, en la que se tendrá una serie de campos a rellenar para que el usuario pueda ingresar los parámetros deseados, así como los archivos de texto con los pesos y umbrales entrenados. La GUI utilizada fue desarrollada por [7], aunque posteriormente modificada para que satisfaga las necesidades del presente trabajo, en ella se contempla el envío de datos en el orden en el que aparecen en la interfaz, desarrollando la comunicación por UART, a partir de Python.

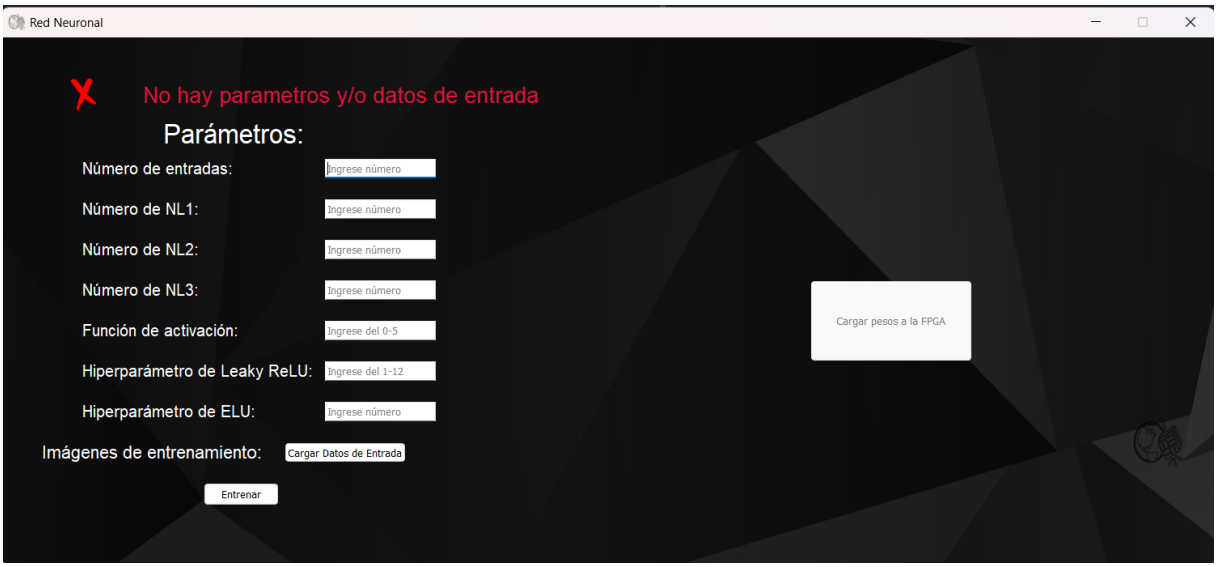


Figura 5.1: Interfaz gráfica de usuario [7].

5.2. FFNN en Hardware

Continuando con la propuesta, se procede con la descripción detallada de la construcción de una FFNN en Hardware, analizándola conforme al subconjunto de problemas establecidos en el diseño preliminar, esto con el fin de continuar el análisis *Top-Down*. Es importante tomar en cuenta que el diseño de detalle atiende los objetivos, la misión y las necesidades de la propuesta, por lo que la prioridad en esta sección es el diseño de las funciones pensadas para su instrumentación, optimizando su implementación.

Es a partir de este enfoque que se elabora la propuesta, por lo que, se espera implementar técnicas de optimización a la instrumentación de las funciones de activación, analizando a detalle el consumo de recursos que cada diseño produce. Sin embargo, en el caso de la instrumentación de la FFNN, se pretende desarrollar el modelo encontrado en el Estado del Arte, esto debido a que no se está aportando alguna técnica de optimización diferente a las ya mencionadas, considerando además que el mejoramiento del diseño de la FFNN no se encuentra como uno de los objetivos de la presente investigación.

Por consiguiente, tomando en cuenta la red diseñada en Software, así como el diseño preliminar, se tendrá que la FFNN que se instrumentará será la que se encuentra en la Figura 5.2, que como se observa, se establece la arquitectura específica, así como su tamaño y profundidad, siendo la meta final y que a continuación se describirá con mayor detalle.

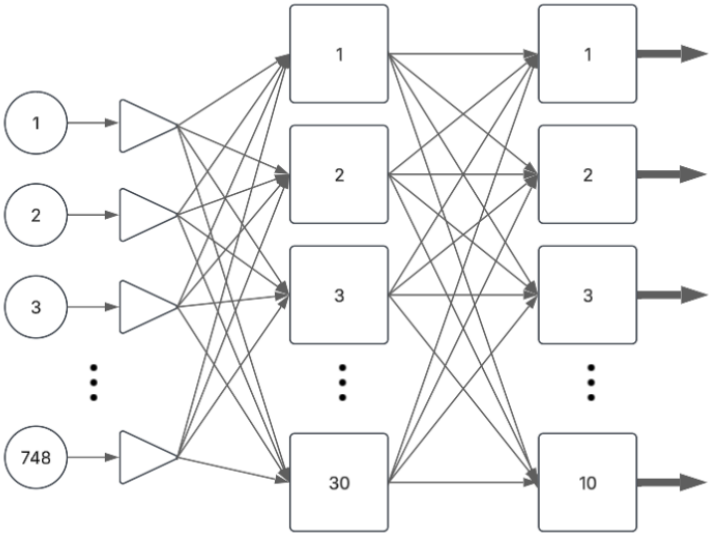


Figura 5.2: Modelo de la FFNN que se implementará.

5.2.1. Selección del formato numérico

Uno de los aspectos más importantes en la construcción de la FFNN es su formato numérico, puesto que no tiene un impacto únicamente en la precisión numérica, sino que además afecta de gran manera la complejidad del sistema y el consumo de recursos. En consecuencia, el formato que se seleccione definirá el dispositivo que se utilice. Por lo que, tomando como referencia lo encontrado en el

Estado del Arte, se hará uso del punto fijo, esto debido a que aunque en comparación al punto flotante no da tanta precisión numérica, presenta una mayor eficiencia de recursos, ideal para proyectos de recursos limitados.

Partiendo de la elección del formato punto fijo, se debe tomar en cuenta que se implementan diferentes funciones de activación que obtienen resultados muy diferentes entre sí, en los cuales la parte decimal es relevante para evitar la muerte de neuronas ($f(x) = 0$)). Por ello, se deberá dar una mayor prioridad a la parte fraccionaria, considerando que se tendrá el signo y una parte entera. Es así que, retomando la Figura 2.2, en la que se muestra el modelo más básico de una neurona instrumentada, se considerarán dos tamaños diferentes de punto fijo, se tendrá un primer tamaño para las entradas, los pesos y los umbrales, para que posteriormente, al ser multiplicados estos entre si, se tendrá un segundo tamaño que sea la suma ponderada y el resultado de la función de activación.

Ya que las entradas se encuentran normalizadas y los pesos y umbrales son números con un valor entero pequeño y una gran parte fraccionar, se establecerá un formato de 8 bits Q5, cuya estructura se muestra en la Tabla 5.3.

8 bits		
1 bit	2 bits	5 bits
Signo	Entero	Fracción

Tabla 5.3: Formato numérico 8 bits Q5.

Considerando que se multiplican dos números de 8 bits Q5, el resultado será un número de 16 bits Q10, estando en este formato la suma ponderada y el valor final de la función de activación, como se observa en la Tabla 5.4. Es una estructura conveniente ya que se sigue teniendo prioridad en la parte fraccionaria, considerando valores muy pequeños, sin embargo, al tener una pequeña parte entera existe el riesgo de que exista un *overflow*, especialmente para aquellas funciones en las que cuando $x > 0 \rightarrow x = x$, por lo que para aquellos casos, se asignará automáticamente el valor saturado.

16 bits		
1 bit	5 bits	10 bits
Signo	Entero	Fracción

Tabla 5.4: Formato numérico 16 bits Q10.

5.2.2. Selección del dispositivo FPGA

Considerando todo lo anteriormente establecido, se requiere un dispositivo FPGA que logre soportar principalmente el formato numérico seleccionado, porque si bien la red no deberá consumir muchos recursos, el formato de 16 bits Q10 conlleva un considerable uso de memoria RAM debido al gran volumen de datos. En consecuencia, se deberá descartar aquellos dispositivos con memoria RAM

limitada.

Buscando un dispositivo de recursos limitados y bajo consumo de energía, que cumplan con la necesidad de la memoria RAM, se selecciona entonces la FPGA EP4CE22F17C6 de la familia Cyclone IV E. La principal ventaja de este dispositivo es que también es conocida como Cyclone IV Nano debido a su pequeño tamaño, por lo que es una opción ideal para proyectos en los cuales el espacio es limitado. En la Tabla 5.6 se puede ver un resumen de las principales características del FPGA.

Características	**Valor**
Voltaje de Kernel	$1.1[V]$
Elementos lógicos	$22,320[EL]$
Memoria RAM	$608,256[bits]$
Multiplicadores de 9-bits embebidos	154
PLL	4
Temperatura de operación	$0[C] - 85[C]$
Velocidad	$50[MHz]$

Tabla 5.5: Especificaciones del FPGA EP4CE22F17C6 [39].

5.2.3. Arquitectura de la FNN

Retomando el diseño de la Figura 4.5, basado en los modelos encontrados en el Estado del Arte, se describirá a continuación el diseño específico que se implemento en FPGA.

Receptor de parámetros de entrenamiento, configuración y entradas

De acuerdo con lo establecido en la transmisión de parámetros, el receptor deberá concordar en el formato, por lo que se desarrollará la conexión por medio del protocolo UART. El bloque de control de la recepción de datos, como la propia comunicación es un área fuera del interés del presente trabajo, por ende se hizo uso de una IP previamente desarrollada en el trabajo [7], insertando únicamente tal modulo hecho en el proyecto.

Bloque de Control

Conforme a lo establecido en el presente trabajo, a pesar de no proponer un diseño nuevo, se desarrollará todos los elementos siguientes de la red neuronal desde cero. De primera instancia, el bloque de control deberá generar las señales necesarias para coordinar el flujo de datos en los registros de los bloques MAC, deberá indicar al selector de entrada cual entrada elegir y además brindará los contadores necesarios para recorrer las memorias de los bloques MAC. Esto nos indica la necesidad

de dos señales y n contadores, donde n es el número de neuronas que se utilizará por capa.

Se deberá considerar la presencia de una memoria RAM, en la cual se tengan los datos de configuración, el número de entradas y de neuronas por capa. Se tendrá entonces una memoria que constará de hasta 32 datos de configuración, o dicho de otra forma, se podrá realizar una FFNN con hasta 32 capas en total, con un tamaño de hasta 10 bits por dato. Consecuentemente se requerirá un contador que recorra las 32 localidades, el cual avanzará una posición cada vez que se termine el procesamiento de las entradas o de las neuronas de una capa.

Existirá paralelamente un contador de 10 bits, al ser este el tamaño máximo de cada dato, y un comparador que este equiparando constantemente el valor de la localidad con el contador, para que cuando estos sean iguales se levante una señal indicando el fin de una capa, que permitirá que el contador de la memoria avance al siguiente dato; reiniciando, además, el contador de 10 bits, para que se repita el mismo proceso con la siguiente localidad. Esta señal generada es aquella que permitirá controlar el flujo de datos en los registros de salida de los bloques MAC.

La señal faltante es aquella necesaria para el selector de entradas, un cero para que seleccione la entrada del usuario y un uno para que seleccione la respuesta de la función de activación. La manera más sencilla de cumplir con este requerimiento es con una compuerta OR, la cual se aplicará al contador de la memoria, entregando un cero únicamente cuando el valor del contador sea igual a cero, siendo el único momento en que se encuentra en la capa de entrada. Finalmente los contadores serán módulos que se crearán de acuerdo a cuantas neuronas se utilizarán por capa. Teniéndose finalmente, en la Figura 5.3, el diseño de detalle del bloque de control, en el que se sintetiza todo lo anteriormente descrito.

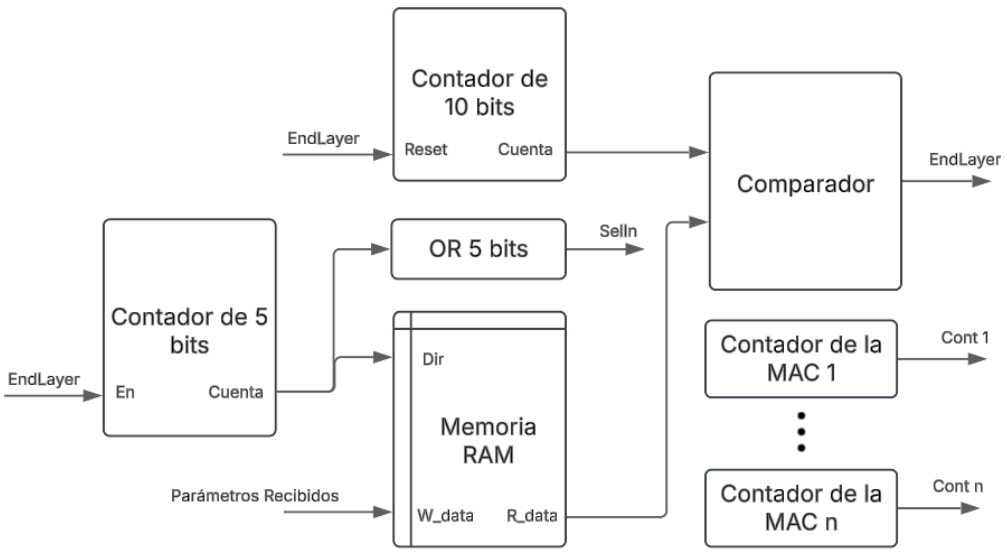


Figura 5.3: Diseño de detalle del bloque de control.

Selector de entradas

El selector de entradas, como su nombre lo indica, elige qué entrada se introducirá a los bloques MAC.

El bloque tendrá por entradas la imagen o el resultado de la función de activación, seleccionando la primer opción mencionada cuando sea la primera capa, mientras que la segunda opción será cuando ya se esté procesando los valores de la capa oculta o de salida. Por lo tanto, se comprender que este bloque se describe en hardware como un multiplexor 2 a 1 con una línea de selección, tal como se muestra en la Figura 5.4.

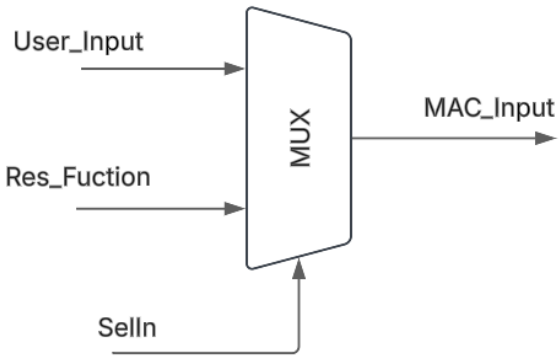


Figura 5.4: Diseño de detalle del selector de entradas.

Bloque de multiplicación y acumulación (MAC) y su registro de salida
Como un elemento fundamental de la red se creará el bloque MAC que tiene como objetivo la obtención de la suma ponderada, mientras que su registro de salida controla el flujo de esta suma. Por lo tanto, se deberá establecer primero las entradas del bloque, las cuales serán la salida del selector de entrada, así como los pesos y umbrales, los cuales el usuario transmitió desde el CPU al FPGA, y serán cargados en una memoria RAM integrada al bloque.

Los datos se encontrarán acomodados de la siguiente forma, primero con el umbral de la primera neurona de la primer capa y después todos los pesos asociados a esa neurona; posteriormente estará de nuevo el umbral de la primera neurona pero ahora de la capa siguiente y después sus pesos asociados, así consecutivamente hasta abarcar todas las neuronas de todas las capas que se establezcan, como se muestra en la Figura 5.5. Es este acomodo que permite la creación únicamente de n bloques MAC, donde n es el número de neuronas de la capa más grande.

Retomando el marco teórico, se sabe que el umbral no se encuentra multiplicado con alguna entrada, por ello, será un valor que se cargará en un inicio al acumulador. Para proceder con la multiplicación de las entradas con los pesos, se hará uso de los multiplicadores embebidos en la FPGA, que ahorran de gran manera los recursos utilizados. Una vez obtenida la suma ponderada, este valor ingresará al registro de salida, que cuando se dé la señal del fin de una capa, el registro entregará su suma ponderada, de otro modo, el registro estará realizando un corrimiento de sumas.

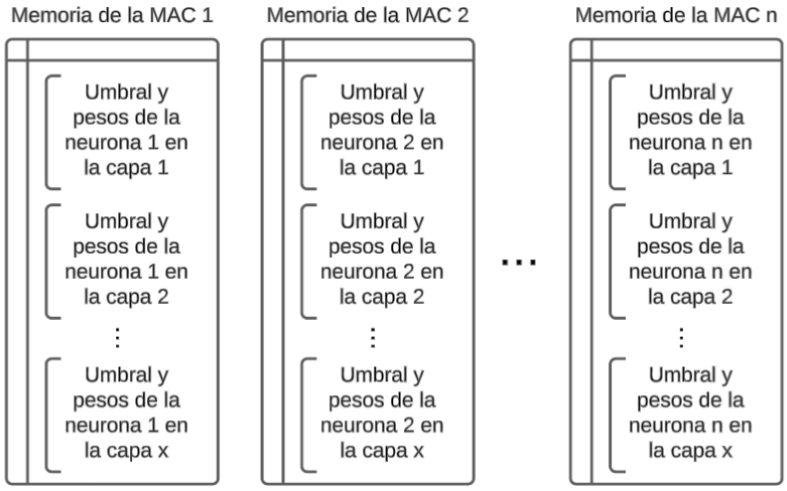


Figura 5.5: Acomodo de las memorias MAC

El objetivo de este funcionamiento es que cada suma ponderada ingrese una por una a la función de activación sin perderse valores y aprovechando la paralelidad del FPGA, permitiendo la eficiencia del uso de recursos. En la Figura 5.6 se puede observar el diseño de detalle del bloque MAC, en el que se sintetiza todo lo anteriormente descrito.

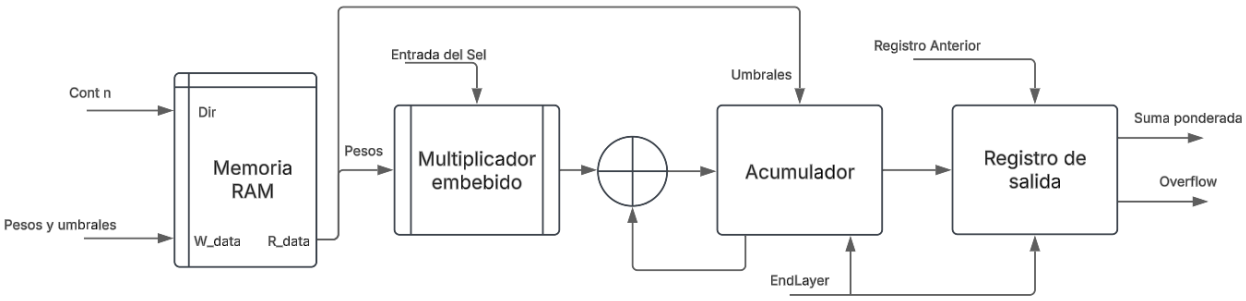


Figura 5.6: Diseño de detalle del bloque MAC con su registro de salida.

5.2.4. Funciones de activación

En principio, es importante comprender que la red realizada en Python hace uso de la fórmula exacta de las diferentes funciones, puesto que hacer uso de las aproximaciones resultaría en errores, debido que para el entrenamiento se necesita las derivadas de las ecuaciones. Por ello, centrándonos en el diseño de las funciones para su instrumentación en FPGA, que como se estableció, será una aproximación lineal por partes (PWL) debido a la optimización de recursos que permite, así como resultados con un muy bajo error.

Entonces se deberá tomar en cuenta las necesidades establecidas de la Tabla 4.1. La prioridad es generar un diseño de bajo consumo de recursos, entendiéndose que el mayor consumo se encuentra en las multiplicaciones de la función, que aunque sea una PWL sigue estando presente este factor. Por lo que, tomando como ejemplo la aproximación ya existente de la función Sigmoide, se realizará la

aproximación de tal modo que las multiplicaciones no sean un producto real, sino corrimientos debido a que se multiplica por algún valor que sea potencia dos (2^x).

A continuación se describirá el diseño llevado a cabo para cada función, considerando el principio anteriormente descrito.

ReLU

La función ReLU, encontrada en la Tabla 2.1, como una de sus principales ventajas, conlleva una gran facilidad de implementación, ya sea en Software o Hardware, consumiendo el mínimo de recursos. Esto se debe gracias a que en los dos casos considerados ($x \geq 0$ o $x < 0$) no se encuentra ningún operando matemático, únicamente una igualdad ($x = x$) o una constante ($x = 0$), respectivamente, por lo que su implementación en Hardware, siendo la excepción con respecto a las demás funciones, será idéntica a la propia definición de la función.

A nivel de descripción de Hardware, la instrumentación de la función sería un multiplexor, que tiene por entradas un cero y la suma ponderada, siendo el selector el signo de la entrada de la función, que recordando el formato numérico (Tabla 5.4), se encuentra en el bit más significativo. La función implementada se observa en la Figura 5.7, el cual es el diseño implementado en Hardware.

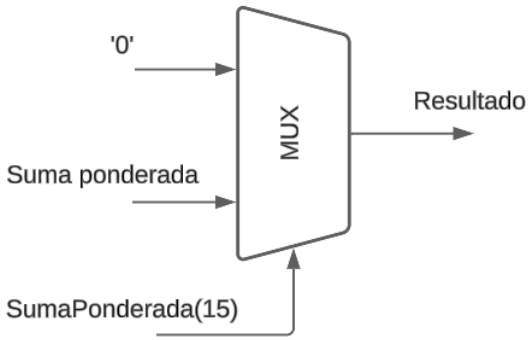


Figura 5.7: Diseño de detalle de la función ReLU.

Signo	Valor	Selección
+	0	Suma ponderada
-	1	"0"

Tabla 5.6: Configuración del multiplexor.

Sigmoide

De acuerdo a la investigación previamente hecha, se presentó en 1989 el primer diseño de la función Sigmoide realizado a partir de aproximación lineal por partes, el cual es utilizado hoy en día. Este modelo representa un parteaguas del presente trabajo, puesto que es su simplicidad, eficiencia y ahorro de recursos que lo convierte en un diseño muy atractivo y útil, siendo esta aproximación lo que ha

permitido que, además de las propias ventajas de la función, sea la ecuación Sigmoide, encontrada en la Tabla 2.1, una de las funciones más utilizadas, principalmente en aquellas redes instrumentadas en FPGA.

La aproximación se encuentra descrita en la ecuación (5.1), que como se puede observar, se encuentra fraccionada en cuatro líneas rectas, que además, como se mencionaba anteriormente, las multiplicaciones son sustituidas por corrimientos, gracias a que las pendientes son potencia de 2, por lo que considerando el formato numérico establecido, se obtiene la ecuación (5.2).

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

$$\sigma(x) = \begin{cases} 1024, & |x| > 5120 \\ 2^{-5} * |x| + 864, & 2432 \leq |x| < 5120 \\ 2^{-3} * |x| + 640, & 1024 \leq |x| < 2432 \\ 2^{-2} * |x| + 512, & 0 \leq |x| < 1024 \end{cases} \tag{5.2}$$

Como se puede observar en la Figura 5.8, la aproximación realizada es bastante precisa arrojando un error promedio de 0.00587 [40]. De acuerdo a su ecuación (5.2), se observa que su aproximación sólo se encuentra definida para los valores positivos de la función, esto es debido a que es una función impar, por ende, para valores negativos se definirá como se muestra en la ecuación (5.3).

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

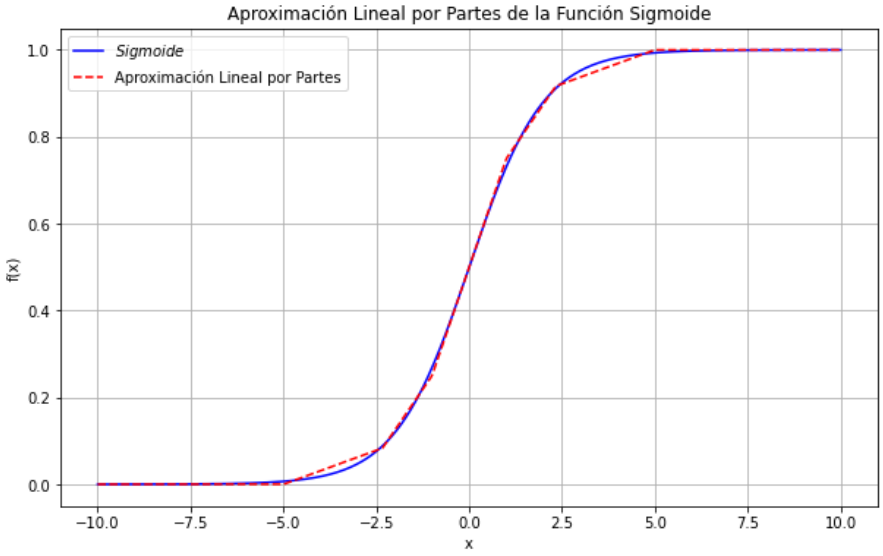


Figura 5.8: Aproximación lineal por partes de la función Sigmoide.

ELU

La función ELU, encontrada en la Tabla 2.1, a pesar de sus múltiples ventajas, es una ecuación nunca antes implementada a nivel de descripción de hardware debido a su complejidad matemática. Por lo que, con el objetivo de obtener una función de bajo consumo de recursos, que permita implementarla en dispositivos de elementos lógicos limitados, se realizó una aproximación lineal por partes de esta ecuación, sustituyendo además las multiplicaciones por corrimientos.

El método de aproximación lineal por partes se comprende como un método gráfico, en el cual, a partir del análisis del comportamiento de la función, se construye la aproximación seccionando la gráfica en puntos críticos y linealizando cada sección. En consideración a esto, se presenta la mayor dificultad de obtener una aproximación de la ecuación ELU: el comportamiento de la gráfica depende del hiperparámetro de la fórmula. Es ante esta problemática que se decide limitar los valores de tal parámetro a un pequeño rango útil, puesto que, considerando que la gran mayoría de estudios establecen este parámetro igual a 1, se establecerá un rango de (0, 2).

Se tomó esta decisión debido a que es aquella solución en la que se hace un mayor ahorro de recursos, sin perder precisión. Una alternativa es definir el hiperparámetro con valores de base 2 para sustituir la multiplicación por corrimientos, pero esto provocaría una gran perdida de valores útiles posibles. Otra opción es realizar una aproximación lineal por partes de la sección para números negativos, e integrar el hiperparámetro por medio de un multiplicador embebido, pero se espera que las funciones de activación no consuman más que elementos lógicos, esto con el fin de que pueda ser implementada en prácticamente cualquier dispositivo y sea la red neuronal la única que consuma aquellos multiplicadores embebidos.

Tomando en cuenta tales consideraciones, se procede con el análisis del comportamiento de la gráfica de la función considerando el cambio que sufre con respecto al rango establecido del hiperparámetro. Como se puede observar en la Figura 10a, para valores positivos de x, la gráfica ya es una línea recta ($f(x) = x$), mientras que para valores negativos, se tendrá un gran cambio entre la región de (-3, 0), y posteriormente en la región siguiente $(-\infty$, -3] el comportamiento de la función se estabiliza y se obtiene una recta definida como $f(x) = \alpha$, como se muestra en la Figura 5.9.

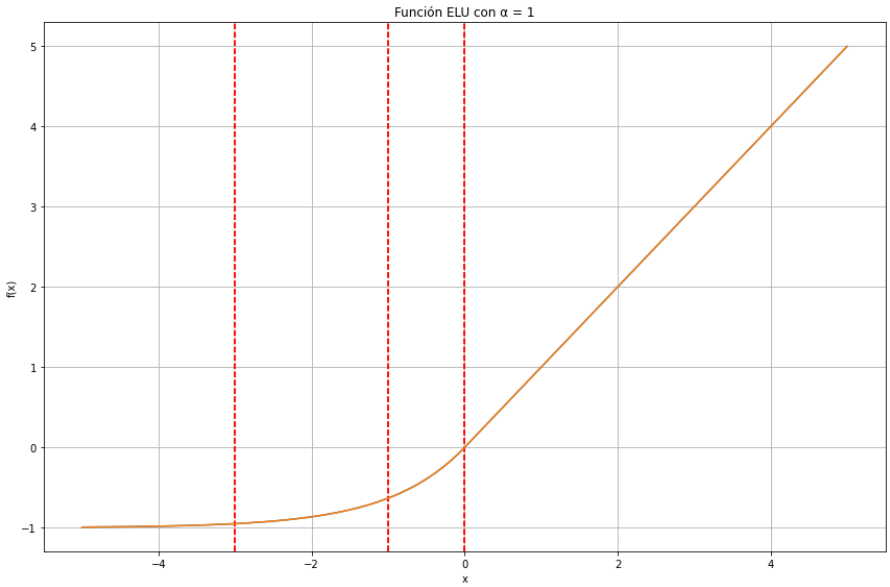


Figura 5.9: Secciones de la función ELU.

A partir de este primer análisis, se puede ya visualizar dos regiones claras de la función, una primera, en la cual se tiene una curva cambiante con respecto al parámetro y una segunda sección que se compone de una línea recta horizontal. Por lo tanto, esta última región ya cumple con el objetivo en cuanto al óptimo consumo de recursos que conlleva implementarla, así como una gran precisión.

Continuando con la aproximación, la primer región de los valores negativos no es tan sencilla como la segunda, puesto que vemos una curvatura característica de un crecimiento logarítmico. Para linealizar aquella sección no se puede lograr una buena precisión con una sola recta, por lo que, se dividirá en dos regiones para abarcar la parte de la curva con mayor pendiente y una segunda recta para la menor pendiente. Se tendrá entonces una primer región que abarca de (-1, 0) con una pendiente más pronunciada y una segunda que va de (-3, -1], con una pendiente más plana.

Establecidas las regiones críticas en las que se implementará la linealización, se deberá seccionar el rango del hiperparámetro en pequeñas regiones para que se tengan diferentes rectas de acuerdo al valor del parámetro. Esto se realiza debido a que los cambios de pendiente que tienen las rectas en aquellas regiones no son constantes o proporcionales a algún valor de base 2, que permita evitar las multiplicaciones. Cada región del hiperparámetro fue establecida a partir del cambio de la curva de la gráfica, obteniéndose cinco secciones en total:

1. $\alpha \in (0, 0.4]$
2. $\alpha \in (0.4, 0.9)$
3. $\alpha \in [0.9, 1.2)$
4. $\alpha \in [1.2, 1.5)$
5. $\alpha \in [1.5, 2)$

Con respecto a tales regiones, se crean entonces cinco casos diferentes de α para las regiones de $x \in (-1, 0)$ y $x \in (-3, -1]$. Partiendo del seccionamiento de toda la gráfica en nuestras áreas de interés, se puede proceder con la linealización de cada sección, que se realizará cada recta con la fórmula general de una:

$$f(x) = ax + b \tag{5.4}$$

Donde a y b son constantes de base dos que facilitarán y ahorrará recursos en su implementación en hardware. Finalmente se obtiene la aproximación que se observa en la ecuación (5.5) y considerando la base numérica establecida, se obtiene la ecuación (5.6).

$$ELU(x) = \begin{cases} x, & x \geq 0, \quad -\infty < \alpha < \infty \\ 0.125 * x - 0.125 * \alpha, & -1 < x < 0, \quad 0 < \alpha \leq 0.4 \\ 0.5 * x, & -1 < x < 0, \quad 0.4 < \alpha < 0.9 \\ 0.5 * x - 0.125 * \alpha, & -1 < x < 0, \quad 0.9 \leq \alpha < 1.2 \\ 0.5 * x + 0.125 * x + 0.125 * x - 0.125 * \alpha, & -1 < x < 0, \quad 1.2 \leq \alpha < 1.5 \\ 0.5 * x + 0.25 * x + 0.0625 * x - 0.125 * \alpha, & -1 < x < 0, \quad 1.5 \leq \alpha < 2 \\ 0.0625 * x - 0.25 * \alpha, & -3 < x \leq -1, \quad 0 < \alpha \leq 0.4 \\ 0.125 * x - 0.5 * \alpha, & -3 < x \leq -1, \quad 0.4 < \alpha < 0.9 \\ 0.25 * x - 0.25 * \alpha - 0.125 * \alpha, & -3 < x \leq -1, \quad 0.9 \leq \alpha < 1.2 \\ 0.125 * x - 0.5 * \alpha - 0.125 * \alpha - 0.0625, & -3 < x \leq -1, \quad 1.2 \leq \alpha < 1.5 \\ 0.25 * x - 0.5 * \alpha - 0.0625 * \alpha, & -3 < x \leq -1, \quad 1.5 \leq \alpha < 2 \\ -\alpha, & x \leq -3, \quad -\infty < \alpha < \infty \end{cases} \tag{5.5}$$

$$
ELU(x) = \begin{cases}
x, & x \geq 0, \quad -\infty < \alpha < \infty \\
2^{-3} * x - 2^{-3} * \alpha, & -1024 < x < 0, \quad 0 < \alpha \leq 409 \\
2^{-1} * x, & -1024 < x < 0, \quad 409 < \alpha < 921 \\
2^{-1} * x - 2^{-3} * \alpha, & -1024 < x < 0, \quad 921 \leq \alpha < 1280 \\
2^{-1} * x + 2^{-3} * x + 2^{-3} * x - 2^{-3} * \alpha, & -1024 < x < 0, \quad 1280 \leq \alpha < 1536 \\
2^{-1} * x + 2^{-2} * x + 2^{-4} * x - 2^{-3} * \alpha, & -1024 < x < 0, \quad 1536 \leq \alpha < 2048 \\
2^{-4} * x - 2^{-2} * \alpha, & -3072 < x \leq -1024, \quad 0 < \alpha \leq 409 \\
2^{-3} * x - 2^{-1} * \alpha, & -3072 < x \leq -1024, \quad 409 < \alpha < 921 \\
2^{-2} * x - 2^{-2} * \alpha - 2^{-3} * \alpha, & -3072 < x \leq -1024, \quad 921 \leq \alpha < 1280 \\
2^{-3} * x - 2^{-1} * \alpha - 2^{-3} * \alpha - 64, & -3072 < x \leq -1024, \quad 1280 \leq \alpha < 1536 \\
2^{-2} * x - 2^{-1} * \alpha - 2^{-4} * \alpha, & -3072 < x \leq -1024, \quad 1536 \leq \alpha < 2048 \\
-\alpha, & x \leq -3072, \quad -\infty < \alpha < \infty
\end{cases} \tag{5.6}
$$

En la Figura 5.10 se puede observar la comparación entre la aproximación y el comportamiento real de la función, estableciendo sólo algunos valores del hiperparámetro para su ejemplificación.

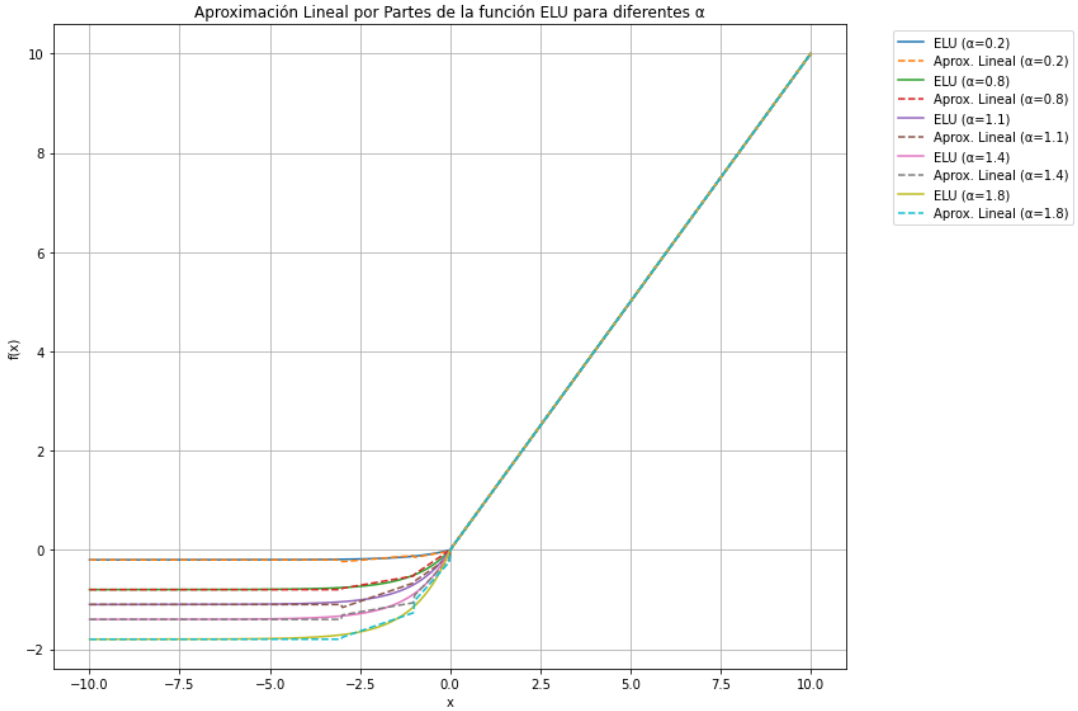


Figura 5.10: Aproximación lineal por partes de la función ELU.

TanH

La función Tangente hiperbólica, encontrada en la Tabla 2.1, al tener una gran similitud a la función Sigmoide, en cuanto al comportamiento de la gráfica, su aproximación está basada en la aproxima-

ción diseñada para la Sigmoide. Como se puede observar en la Figura 7a, la gráfica es una versión escalonada de la función Sigmoide, ampliando el rango de salida de [0, 1] a [-1, 1], por lo que la aproximación quedaría, para valores positivos, como se indica en la ecuación (5.7), mientras que para valores negativos sería de acuerdo a la ecuación (5.8).

$$TanH(x) = 2Sigmoide(2x) - 1 \tag{5.7}$$

$$TanH(x) = 1 - 2Sigmoide(2x) \tag{5.8}$$

Considerando el formato numérico establecido, se tiene entonces las siguientes ecuaciones.

$$TanH(x) = 2Sigmoide(2x) - 1024 \tag{5.9}$$

$$TanH(x) = 1024 - 2Sigmoide(2x) \tag{5.10}$$

Debido a que la aproximación de la función Sigmoide no es modificada de ninguna forma, únicamente se escala, entonces el error del diseño sigue siendo el mismo, un error promedio de 0.00587.

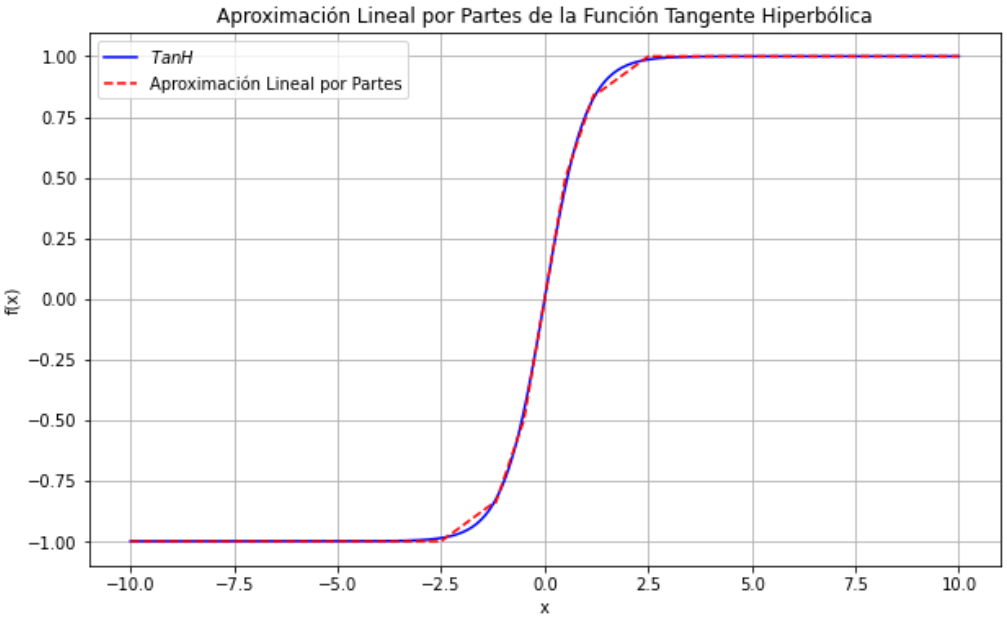


Figura 5.11: Aproximación lineal por partes de la función Tangente Hiperbólica.

Leaky ReLU

La función Leaky ReLU, encontrada en la Tabla 2.1, a pesar de su sencillez matemática, el hiperparámetro que contiene establece una multiplicación que se desea evitar, puesto que se busca que las funciones no consuman multiplicadores embebidos. Por ello, para solucionar el problema, se plantea implementar el hiperparámetro con valores de potencia 2 para que se realicen únicamente corrimientos. El beneficio de esta solución es que permite el ahorro de un multiplicador embebido, mientras que no se pierde precisión, gracias a que se mantiene una amplia gama de valores útiles, considerando

además que, de acuerdo con la mayoría de artículos, se iguala este parámetro a 0.01.

Se fija que este hiperparámetro tiene que ser algún valor entre (0, 1), por lo que, considerando el formato numérico establecido, se implementará un valor entre [2^{-12}, 2^{-1}]. La desventaja de este nuevo rango establecido es que acorta los valores a (0, 0.5], pero como se mencionaba anteriormente, típicamente se establecen valores menores de 0.1, puesto que da mejores resultados. Además, se establece en el límite inferior del rango un valor cercano a cero, pero sin estar tan próximo, debido a que con esto se espera evitar la muerte de neuronas al provocar que los valores se conviertan igual a cero.

A partir de esta propuesta, se tiene entonces un diseño listo para ser implementado, el cual, a diferencia de las demás funciones, no se está realizando una aproximación, únicamente se estableció un nuevo criterio que permite el ahorro de recursos en su implementación en Hardware. Por lo que, el comportamiento de la función sigue siendo el mismo. Con respecto al nuevo diseño, se desarrolla el diseño de detalle de la función, que se observa en la Figura 5.12, similar al de la función ReLU, con la diferencia de un resultado distinto a cero para los valores negativos, teniéndose en su lugar la suma ponderada recorrida 'α' veces, donde α es el exponencial de la base, $2^{-\alpha}$.

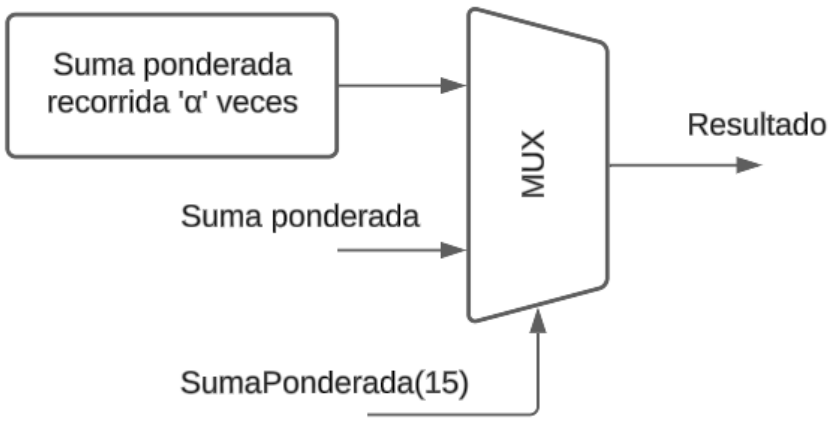


Figura 5.12: Diseño de detalle de la función Leaky ReLU.

Swish

La funcion Swish, encontrada en la Tabla 2.1, es una de las dos funciones que no cuentan con una aproximación previa, aunque a diferencia de la ELU, esta función sí se ha implementado en Hardware. Esto se ha logrado a partir de una equivalencia, tomando como referencia la función ReLU, sin embargo se considero la función *hard-swish* como punto de partida, la cual es una versión de la función Swish construida a partir de rectas y curvas, que toman como referencia la función *hard-sigmoide* [41]. Esta equivalencia se puede observar en la ecuación (5.11).

$$h - swish(x) = x * h - sigmoide(x) = x * \frac{ReLU6(x+3)}{6} \tag{5.11}$$

A pesar de la precisión de esta equivalencia, en cuestión al ahorro de recursos, no se está mejorando considerablemente su consumo, puesto que cuenta con una multiplicación y una división. Una

alternativa es hacer uso de la aproximación de la función Sigmoide, e integrar la multiplicación de x para cada parte, no obstante, seguiría existiendo un multiplicador embebido el cual se busca evitar, por lo que se decide realizar una aproximación lineal por partes a la función.

Del mismo modo en que se realizó para la función ELU, se deberá analizar el comportamiento de la gráfica de la función Swish, que se encuentra en la Figura 11a, la cual tiene el beneficio de no contar con algún hiperparámetro. De primera instancia, se seccionarán las regiones críticas en las cuales se sustituirá por rectas, por ende, a continuación se muestran las ocho secciones que se establecieron para este fin:

1. $x \geq 4$
2. $2.5 < x \leq 4$
3. $0.5 < x \leq 2.5$
4. $-0.5 < x \leq 0.5$
5. $-1.5 < x \leq -0.5$
6. $-3.5 < x \leq -1.5$
7. $-5 < x \leq -3.5$
8. $x \leq -5$

A partir de cada región se realizará la linealización de la curva, por lo que se obtiene la ecuación (5.12), en la cual se tiene la aproximación final de la función, que considerando el formato numérico, se tiene entonces la ecuación (5.13). En la Figura 5.13 se puede observar gráficamente la comparación entre la función real y su aproximación.

$$Swish(x) = \begin{cases} x, & x \geq 4 \\ x - 0.125, & 2.5 < x \leq 4 \\ x + 0.25, & 0.5 < x \leq 2.5 \\ 0.5 * x, & -0.5 < x \leq 0.5 \\ 0.125 * x - 0.125, & -1.5 < x \leq -0.5 \\ -0.125 * x - 0.5, & -3.5 < x \leq -1.5 \\ -0.015625 * x - 0.125, & -5 < x \leq -3.5 \\ 0, & x \leq -5 \end{cases} \tag{5.12}$$

$$Swish(x) = \begin{cases} x, & x \geq 4096 \\ x - 128, & 2560 < x < 4096 \\ x + 256, & 512 < x \leq 2560 \\ 2^{-1} * x, & -512 < x \leq 512 \\ 2^{-3} * x - 128, & -1536 < x \leq -512 \\ -2^{-3} * x - 512, & -3584 < x \leq -1536 \\ -2^{-6} * x - 128, & -5120 < x \leq -3584 \\ 0, & x \leq -5120 \end{cases} \tag{5.13}$$

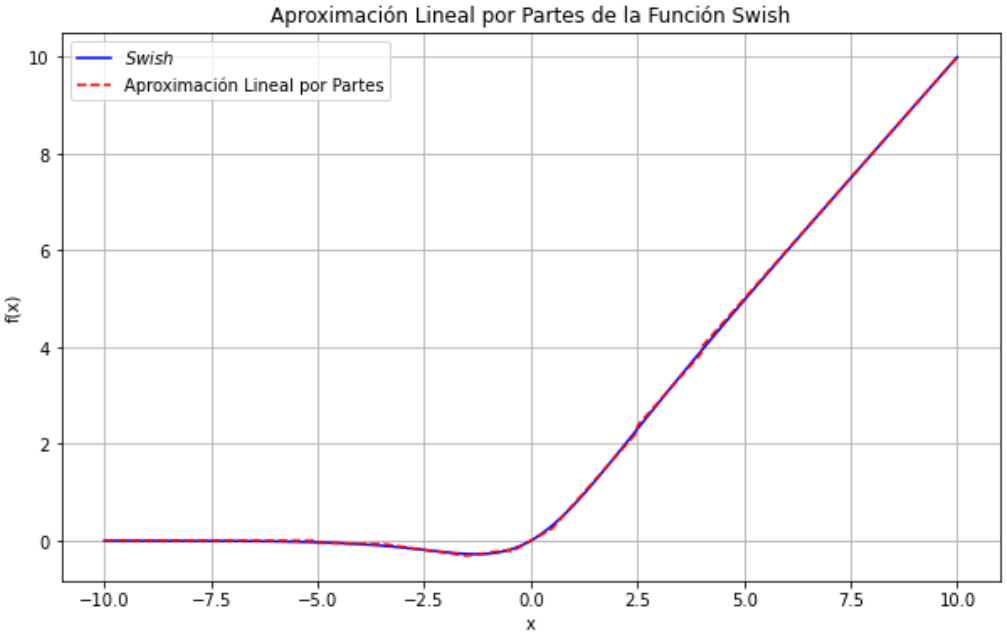


Figura 5.13: Aproximación lineal por partes de la función Swish.

Overflow

Un aspecto importante a considerarse en el diseño de las funciones es que se deberá considerar la posibilidad de que exista un *overflow*, haciendo referencia al desbordamiento de un resultado, específicamente hablando en la obtención de la suma ponderada existe el caso en que el resultado sea mas grande de lo que se pueda representar, por lo que el valor obtenido ya no representa un resultado real. En estos casos, se deberá identificar constantemente que el resultado no se encuentre desbordado y si es el caso, el resultado de la función de activación ya no será aquel procesado por la ecuación matemática, sino que se arrojará automáticamente el valor saturado de cada función, como se muestra en la Tabla 5.7.

Función de activación	**Valor saturado**	
	Valores positivos	**Valores negativos**
ReLU	Hex‘7C00’	0
Sigmoide	1	0
ELU	Hex‘7C00’	$-\alpha$
TanH	1	-1
Leaky ReLU	Hex‘7C00’	Corrimiento α veces del valor Hex‘7FFF’
Swish	Hex‘7C00’	0

Tabla 5.7: Consumo de elementos lógicos por cada función de activación.

Se puede observar en la Figura 5.14 el diseño que se deberá llevar a cabo para implementar la bandera que indique la existencia de *overflow*, mientras que en la Figura 5.15 se observa el diseño que se deberá implementar en cada función.

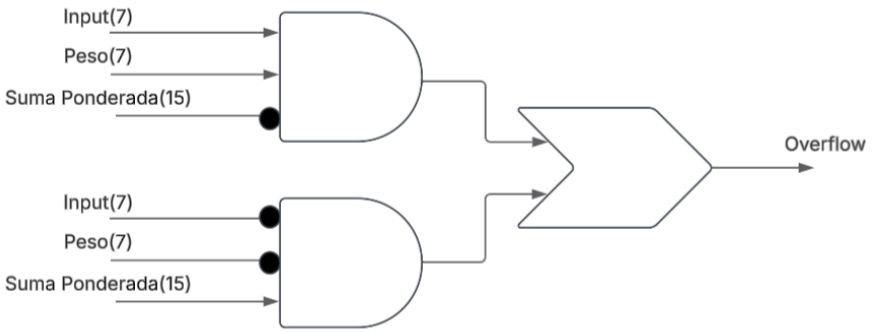


Figura 5.14: Diseño de detalle de la bandera de *Overflow*

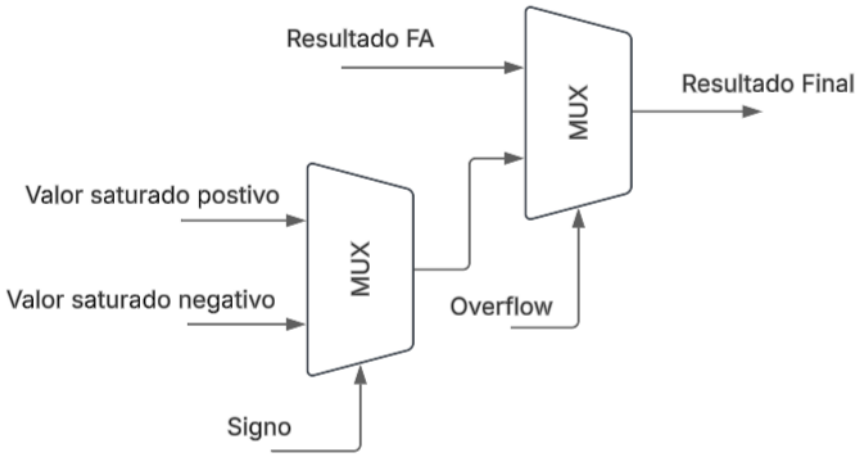


Figura 5.15: Diseño de detalle de la implementación del *Overflow* para cada función.

Capítulo 6

6. Implementación

Para el presente capítulo se espera mostrar el proceso de implementación en Hardware de la ANN, aunque como se ha mencionado en anteriores capítulos, el análisis del diseño de la red no es el objetivo principal en la optimización del consumo de recursos, puesto que no se está proponiendo algo nuevo. Por ende, en este capítulo se espera realizar un análisis de la implementación de las funciones de activación, esto con el objetivo de poner en práctica técnicas de optimización que promuevan el ahorro de recursos lógicos, además de las que ya se han implementado en el diseño. Posteriormente se describirá el proceso de instrumentación en FPGA, una vez ya teniéndose el diseño estructurado en Quartus.

6.1. Implementación en Quartus y RTL viewer

De acuerdo con los objetivos del presente trabajo se realizará un análisis del consumo de recursos de las funciones de activación, teniendo como prioridad generar el mayor ahorro que sea posible. En este punto del trabajo, el diseño desarrollado se encuentra optimizado a partir de modelos matemáticos que persiguen este interés, sin embargo las técnicas de optimización aplicables en esta etapa son diferentes a las impuestas en la etapa de diseño, siendo que para este proceso aquellas implementadas serán *pipeline*, reutilización de recursos, eliminación de redundancias, compresión de datos y principalmente, código HDL optimizado.

La red neuronal artificial fue implementada en *Quartus Prime*, el cual es el entorno de desarrollo integrado (IDE, por sus siglas en inglés) que proporciona Intel para el uso de sus dispositivos FPGA. Esta IDE ofrece lenguajes y formas diferentes de descripción de hardware, ya sea por medio de esquemático o lenguajes como Altera (AHDL, por sus siglas en inglés), Verilog, VHSIC (VHDL, por sus siglas en inglés), entre otros, siendo que para este trabajo se hizo uso de VHDL para la construcción total del sistema. Una de las herramientas más útiles para el análisis del desarrollo de proyectos que brinda Quartus es la visualización del hardware sintetizado a nivel de transferencia de registro (RTL, por sus siglas en inglés), por lo que será a partir de estos esquemas que se podrá comprender de una mejor manera el uso específico de hardware para cada función.

Se espera establecer aquel diseño de descripción de hardware que otorgue el menor uso de recursos para cada función de activación. Se hará énfasis en el desarrollo de un código HDL optimizado,

esto es debido a que existen dos enfoques diferentes que corresponden a dos estilos, el primero es un diseño a nivel compuertas, que es conocido como descripción estructural, mientras que el otro es un diseño a alto nivel, conocido como descripción comportamental [32]. El primer mencionado describe conexiones físicas usando primitivas lógicas (AND, OR, Flip-Flops, etc), por el contrario, el segundo diseño describe un comportamiento y hace uso de bloques procedurales (if-else, for, while, etc).

Cada estilo tiene sus ventajas y desventajas, aunque en proyectos de gran escala, siempre será recomendable el uso de descripción comportamental debido a su portabilidad y facilidad de entendimiento, así como de modificación, es por ello que la ANN fue implementada en este tipo de descripción, debido a que una de las prioridades del trabajo es la portabilidad de la red. En la Figura 6.1 se observa el esquema en un alto nivel de abstracción de la red neuronal artificial completamente implementada con las diferentes funciones.

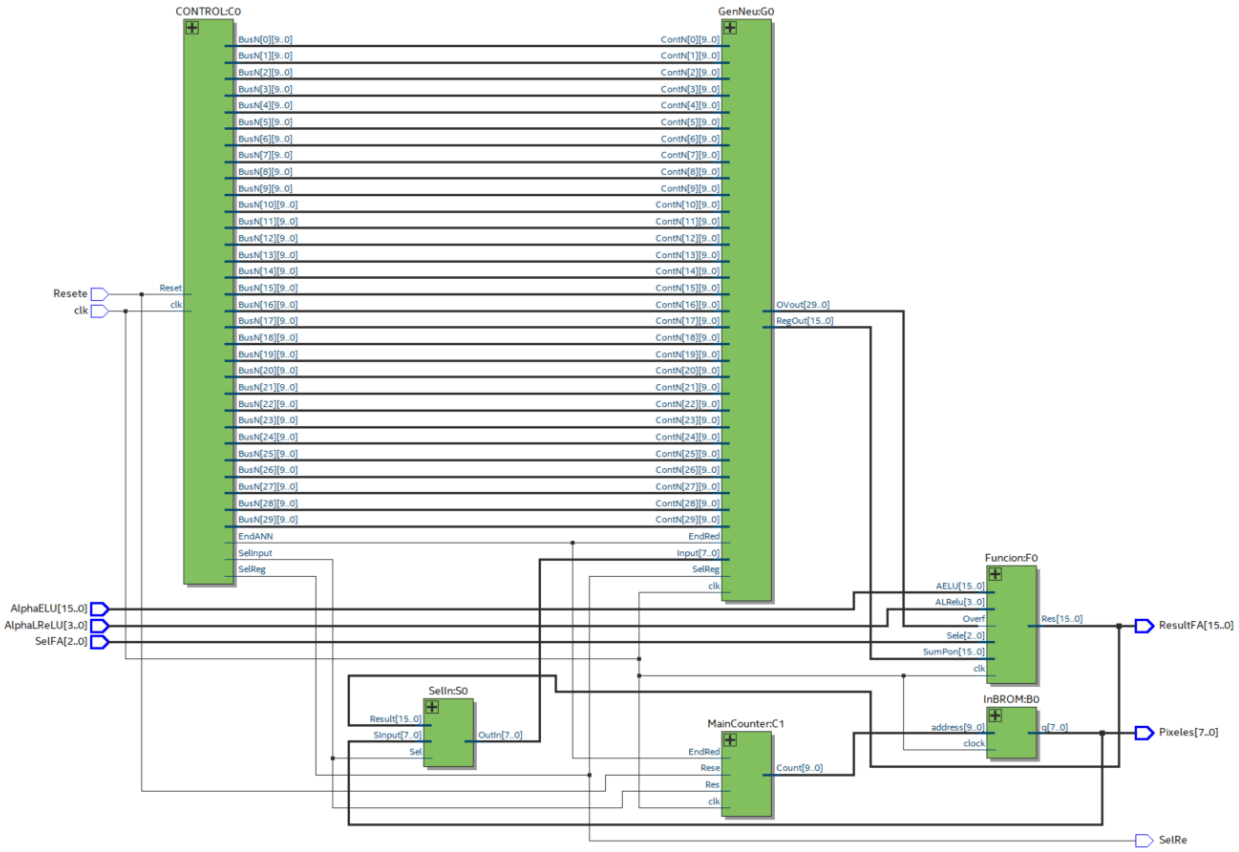


Figura 6.1: RTL *viewer* de la ANN.

Sin embargo, las funciones de activación al tener como prioridad el ahorro de recursos se realizaron diferentes versiones de cada función con el fin de observar y analizar su consumo, comparando cada enfoque de descripción para obtener aquella versión más eficiente.

Es importante analizar de primera instancia las implicaciones de desarrollar un diseño totalmente estructural, esto quiere decir que en operandos matemáticos como por ejemplo la suma, que es la operación más utilizada en las aproximaciones lineales, se deberá realizar con circuitos combinacionales

conocidos como *full-adder* y *half-adder*, de los cuales se obtienen dos elementos: el resultado de la suma y su acarreo, por lo que se está generando por cada suma dos circuitos combinacionales para la obtención de los dos resultados, como se observa en la Figura 6.2.

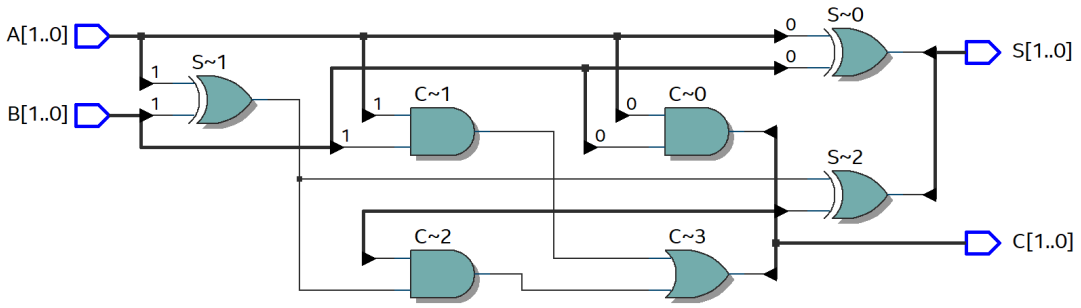


Figura 6.2: RTL *viewer* de una suma de dos bits a nivel compuertas.

Comprendiendo este consumo que genera una suma a bajo nivel, se compara entonces con una implementación a alto nivel, la cual en la síntesis del código se traduce a un segmento especializado que integra un circuito específico para la suma algebraica, observándose en la Figura 6.3. A nivel sistema, un FPGA está compuesto de varios Bloques de lógica configurable (CLB, por sus siglas en inglés), que se observa en la Figura 6.14, que a su vez estos bloques están compuestos por diversos Módulos lógicos, en ellos se puede encontrar los circuitos reprogramables que el FPGA utiliza para desarrollar cualquier circuito que se desee implementar.

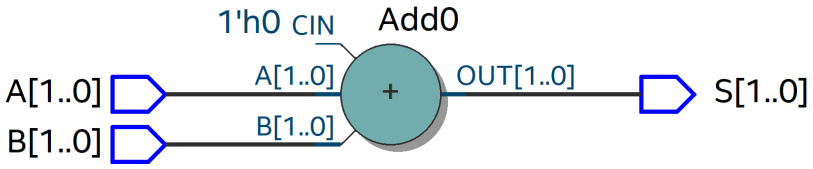
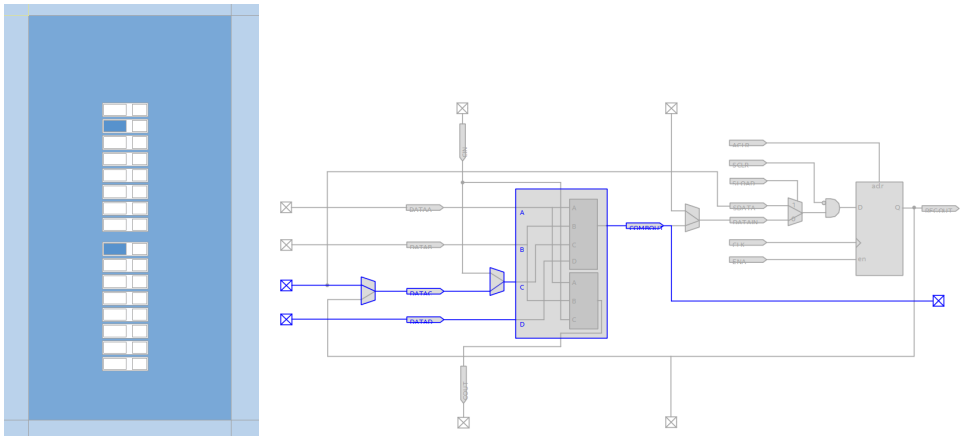


Figura 6.3: RTL *viewer* de una suma de dos bits a alto nivel.

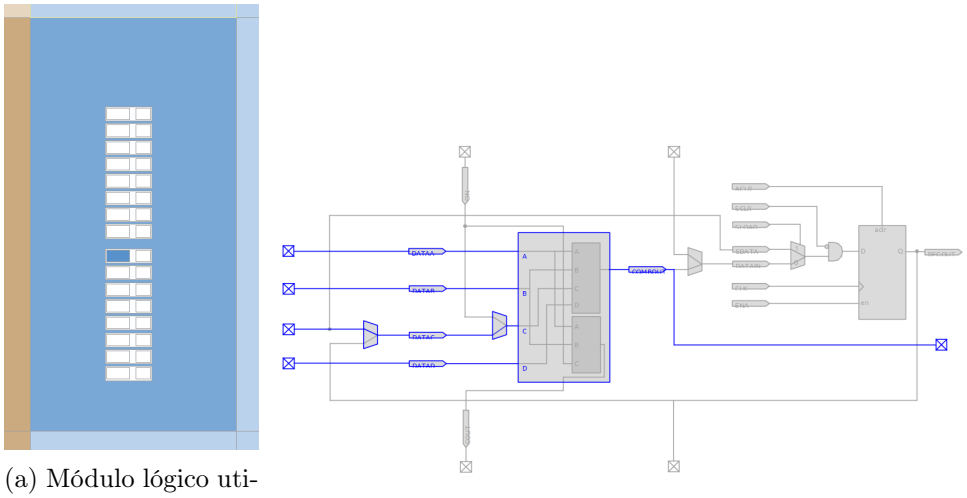
Es a partir del conocimiento de la estructuración del FPGA que se observa que la suma descrita estructuralmente hace uso de dos módulos lógicos, para el circuito combinacional del acarreo y el resultado de la suma, mientras que la suma descrita comportamentalmente reduce el uso de dos módulos lógicos, a uno solo. Esta comparación se puede observar en la Figura 6.4 y la Figura 6.5, observándose los módulos que hace uso una suma de un bit para cada tipo de descripción y su circuito reprogramable correspondiente. Por lo que, con este ejemplo, se concluye que un diseño totalmente estructural no puede proveer la versión con menor consumo de recursos, teniéndose que implementar un enfoque a alto nivel para ciertas funciones obligatoriamente.



(a) Módulos lógicos utilizados en el CLB.

(b) Circuito reprogramable establecido por cada módulo lógico activo.

Figura 6.4: Recursos utilizados por la suma a bajo nivel.



(a) Módulo lógico utilizado en el CLB.

(b) Circuito reprogramable establecido por el módulo lógico.

Figura 6.5: Recursos utilizados por la suma a alto nivel

Entonces, a continuación se describirá el diseño que se llevó a cabo para cada función de activación, el cual fue un proceso de selección entre diferentes versiones de una misma función seleccionando el que genere el menor consumo. Para cada función se desarrolló una versión con descripción comportamental y diferentes versiones que son una combinación entre ambos enfoques. Es importante tomar en cuenta que cada cambio o implementación a cada versión es una decisión a consciencia considerando el hardware que cada sentencia genera.

Además de lo mencionado, se debe considerar que lo que se muestra en la Figuras de cada función corresponde únicamente a la implementación de la aproximación, faltando la obtención del absoluto de x que se utiliza en la gran mayoría de las funciones, así como la obtención del resultado en complemento a 2 para números negativos. Estos bloques adicionales se implementan en archivos diferentes, esto con el objetivo de promover la reutilización de recursos, puesto que en varios diseños se requieren

las mismas funciones, por lo que dividir las en diferentes archivos evitar redundancias.

ReLU

La función ReLU es la única función que se establece desde un principio el diseño a bajo nivel de lo que se implementará en hardware, puesto que conlleva la menor complejidad matemática, así como el menor uso de recursos. En la Figura 6.6 se puede observar el hardware implementado, un primer multiplexor que satisface a la función ReLU y un segundo para arrojar el valor saturado si es que existe un *overflow*. Adicionalmente hay un registro que permite la coordinación de todo el sistema al reloj, esto se realiza de este modo para evitar errores por desincronización.

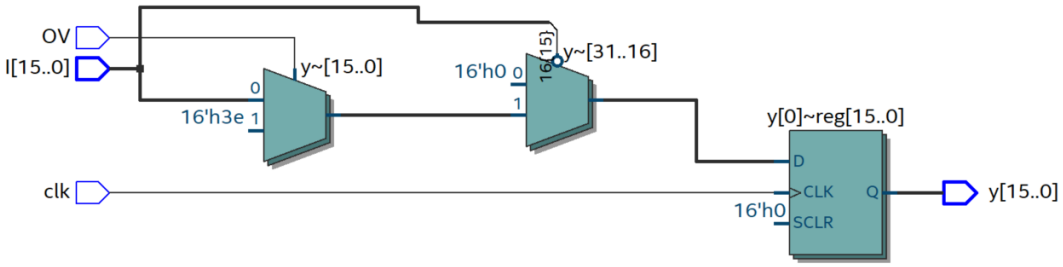


Figura 6.6: RTL *viewer* de la función ReLU.

Sigmoide

Para la función Sigmoide se tiene únicamente el diseño de su aproximación, por lo que se deberá encontrar aquel implementación que haga el menor consumo de recursos. Se realizó la aproximación a nivel compuerta, a partir de la construcción de comparadores y multiplexores a la medida, se integraron sentencias preestablecidas como *with-select*, hasta sustituir gran parte por sentencias *if-else*. Se encontró que la versión que menos recursos consume es aquella en la que se utilizó en su mayoría sentencias *if-else*, puesto que realizar comparadores a bajo nivel conlleva un mayor uso de recursos desde el momento en que se requieren un par de pasos extra para lograrlo, esto debido al compilador de Quartus y su interpretación.

En la Figura 6.7 se puede observar los recursos lógicos que consume la función, los cuáles representan los mínimos necesario. Contiene los ocho comparadores necesarios de la aproximación, tres compuertas AND correspondiente a aquellas regiones en las que se tienen que cumplir dos condiciones, cinco multiplexores, de los cuales cuatro corresponden a las cuatro regiones que abarca la aproximación y un multiplexor para arrojar el resultado saturado si existe un *overflow*, así como un registro para sincronizar la función al reloj.

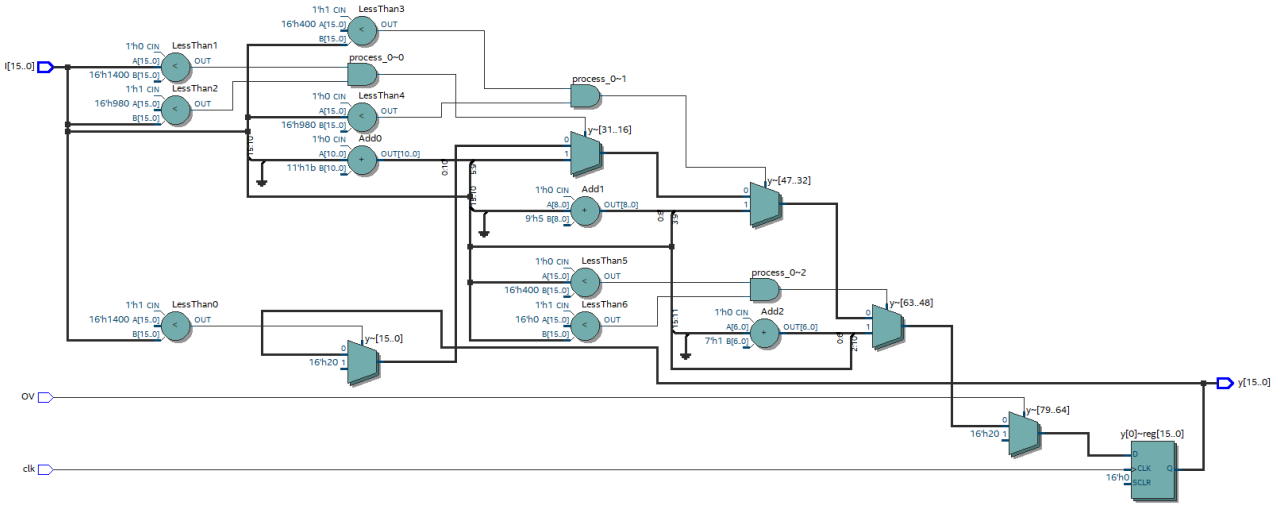


Figura 6.7: RTL *viewer* de la función Sigmoide.

ELU

La función ELU desde el diseño de su aproximación se prevé que será la función que más consuma recursos debido a que es aquella en la que existe más casos. Por ello, es aquella función en la cual se crearon más versiones para reducir ese consumo que ya se estima que será mayor. Se crean comparadores a bajo nivel, así como multiplexores, pero así como se observo con la función Sigmoide, se encuentra que la mejor versión es aquella en la que se encuentra implementada la aproximación con las sentencias *if-else*. En la Figura 6.8 se puede contemplar el consumo de recursos de la función.

Se analiza, de acuerdo con la Figura, que se tienen 12 comparadores correspondientes a las regiones creadas, optimizando tales comparadores al evitar redundancias; se cuenta con 13 multiplexores, de los cuales 12 son de los casos con los que cuenta la función y uno es para el caso en que exista *overflow*, llevando el resultado al valor de saturación negativo o positivo; además se examina que los componentes faltantes son las sumas y algunas compuertas lógicas, siendo estas últimas herramientas útiles para reducir el uso de más componentes.

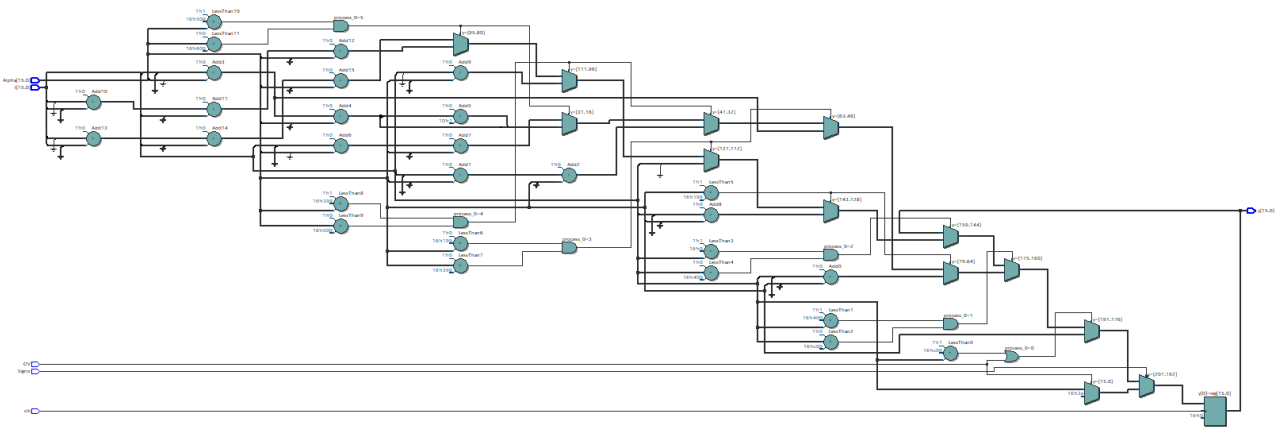


Figura 6.8: RTL *viewer* de la función ELU.

TanH

La función Tangente hiperbólica presenta una ventaja que no tiene ningún otra función, su aproximación se basa en otra función, por lo que, con el objetivo de evitar redundancias, y promover la reutilización de recursos, se mandará a llamar la función Sigmoide agregando únicamente la multiplicación indicada como un corrimiento a la derecha y la suma o resta, siendo que para este trabajo todas las restas son una suma en complemento a 2. Como se observa en la Figura 6.9, se observa la función Sigmoide, tres sumas que equivalen al complemento a 2 y finalmente un multiplexor que de acuerdo al signo arroja uno o otro resultado, así como un registro para solucionar la sincronización del sistema. Para esta función, como se mencionaba anteriormente, la suma debe ser necesariamente implementada en alto nivel.

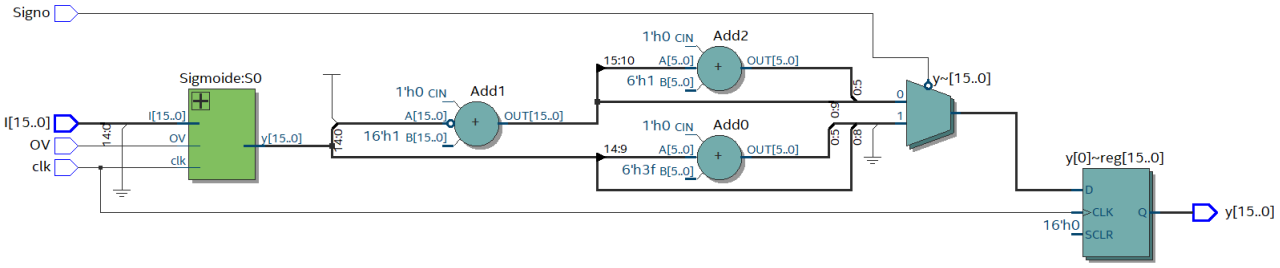


Figura 6.9: RTL *viewer* de la función TanH.

Leaky ReLU

La función Leaky ReLU cuenta con un diseño de detalle descrito en la Figura 5.12, en el cual no se profundiza en el modelo del corrimiento de la suma ponderada, esto es debido a que se espera encontrar la versión que represente el menor consumo de recursos, aunque bien se tiene ya como base el multiplexor que seleccionará el resultado de acuerdo al signo. Resultó en una mejor eficiencia el uso de sentencias *if-else* como se ha observado en las demás funciones de activación, puesto que se logra tener un control total de los recursos que se están utilizando, a diferencia de las sentencias *with-select*, esto debido al interprete de Quartus que con descripción a bajo nivel no logra comprender las estructuras que se desarrollan, realizando un circuito literal, sin utilizar su hardware especializado que optimiza el consumo de recursos.

En la Figura 6.10 se observa una cantidad considerable de multiplexores, de los cuales dos de ellos corresponden al caso por si existe *overflow* llevando el valor saturado si es negativo o positivo, mientras que 10 multiplexores corresponden al número de corrimientos posibles y finalmente el último multiplexor es el indicado en el diseño de detalle de la Figura 5.12, así como un registro final que sincroniza el sistema al reloj. Además, es importante considerar que los corrimientos se deben realizar en señales positivas forzosamente, por lo que, el valor de entrada deberá ser el valor absoluto, y posteriormente se deberá obtener el complemento a 2 de aquellos valores negativos, sin embargo, estas funciones se encuentran en archivos diferentes debido a que se reutilizan en cada función en los que haya corrimientos.

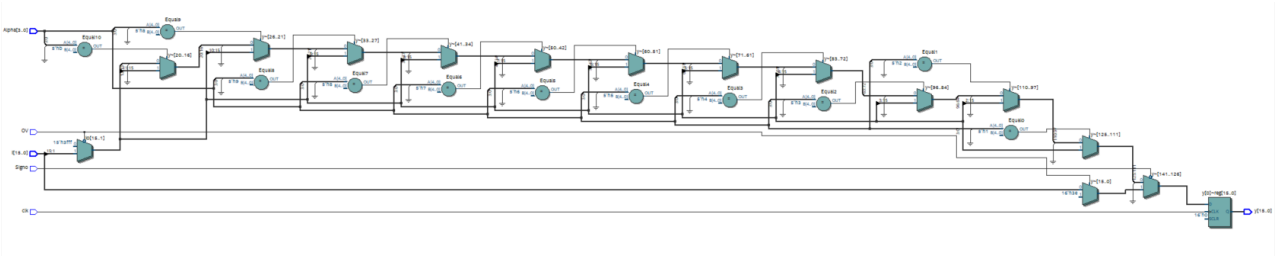


Figura 6.10: RTL *viewer* de la función Leaky ReLU.

Swish

Por último, se tiene únicamente la aproximación lineal de la función Swish, la cual se espera implementar con el mínimo de recursos necesarios. Para ello, se realizaron diferentes versiones realizando comparadores, multiplexores a nivel compuerta, funciones como *with-select* o *if-else*, y se encontró que la versión que menos recursos consumía es aquella en la que se usa la función *if-else*, existiendo una gran diferencia de un 50 % menos de elementos lógicos. En la Figura 6.11 se observa el consumo de recursos de la función, teniéndose un único registro que sincroniza el sistema, 13 comparadores correspondientes a las regiones establecidas, así como sus respectivas compuertas AND cuando se requiere que se establecen las fronteras de las regiones y diez multiplexores, de los cuales ocho son correspondientes a los casos establecidos y dos corresponden a los casos en que exista *overflow* arrojando los valores saturados para la región positiva o negativa.

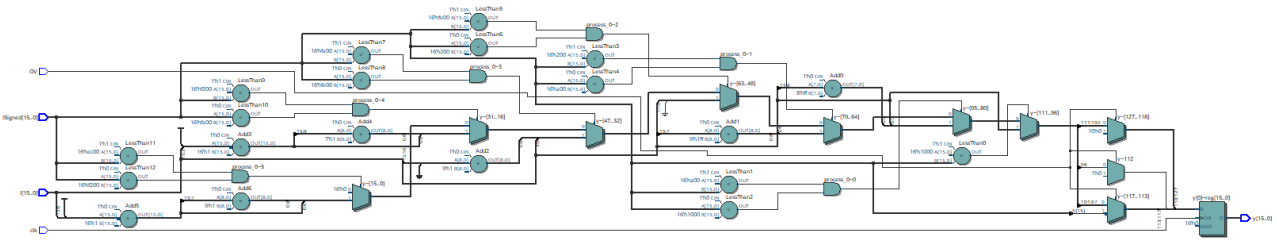


Figura 6.11: RTL *viewer* de la función Swish.

Complemento a 2

De acuerdo a la implementación desarrollada por cada función de activación, se observa que para la gran mayoría de funciones, exceptuando únicamente la ReLU, se necesita trabajar con valores absolutos debido a que todo cambio que se aplique deberá ser con valores positivos, por lo que se deberá obtener el complemento a 2 a las entradas de las funciones a aquellos valores negativos, implementándose este mismo proceso a las salidas negativas. Por lo que, con un solo bloque desarrollado se integrará al inicio y al final de ciertas funciones, reutilizando y ahorrando los recursos disponibles. En la Figura 6.12 se puede observar un multiplexor que arroja la salida igual si es positivo u obtiene el complemento a 2 si es negativo; así como también brindará el signo para indicar a las funciones si el valor que entro era negativo o positivo, además de un registro para cada salida que sirve para sincronizar la máquina.

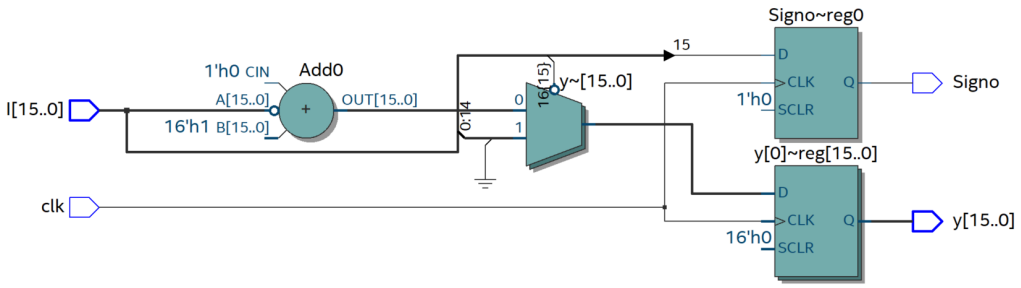


Figura 6.12: RTL *viewer* del complemento a 2.

Complemento de la Sigmoide

Este bloque se requiere únicamente para la función Sigmoide, el cual tiene la función de obtener el resultado para los valores negativos, representando la implementación de la ecuación (5.3). Como se menciona anteriormente, este bloque se desarrolla por separado, ya que la Tangente Hiperbólica hace uso de la Sigmoide con valores positivos únicamente. En la Figura 6.13 se contempla los recursos utilizados por este bloque el cual es un multiplexor que arroja como resultado la entrada sin cambios si es positivo u obtiene el complemento a 2 y se suma el '1' si el valor es negativo, así como un registro de salida que sincroniza la red.

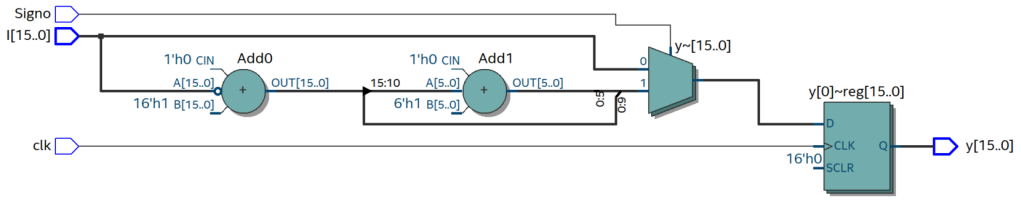


Figura 6.13: RTL *viewer* del complemento de la Sigmoide.

6.2. Implementación en FPGA

Considerando que ya se cuenta con la implementación total del sistema, el cual incluye la red neuronal así como cada función anteriormente descrita, entonces se procede con la construcción del sistema en el FPGA, que consiste en la conexión correcta entre lo diseñado en el IDE y los pines reales del dispositivo. Se tiene entonces que se establecen como entradas físicas el reloj del sistema y la entrada por UART de los parámetros de configuración y de entrada, mientras que las salidas físicas serán los LED integrados en el dispositivo, utilizando cuatro de los ocho disponibles, con el fin de simular un número binario de cuatro bits, que representa la respuesta de la red.

En la Figura 6.14 se muestra el FPGA que se utilizará para el presente trabajo, mientras que en la Figura 6.15 se observa la interfaz del IDE que permite realizar asignación de pines, a partir de una herramienta llamada *pin planner*, el cual asigna salidas del sistema con salidas físicas del dispositivo, en este caso, asociada a los LED que integra la tarjeta de desarrollo. En la Tabla 6.1 se registra de mejor manera la conexión que se desarrolló.

Figura 6.14: Dispositivo FPGA a utilizar [39].

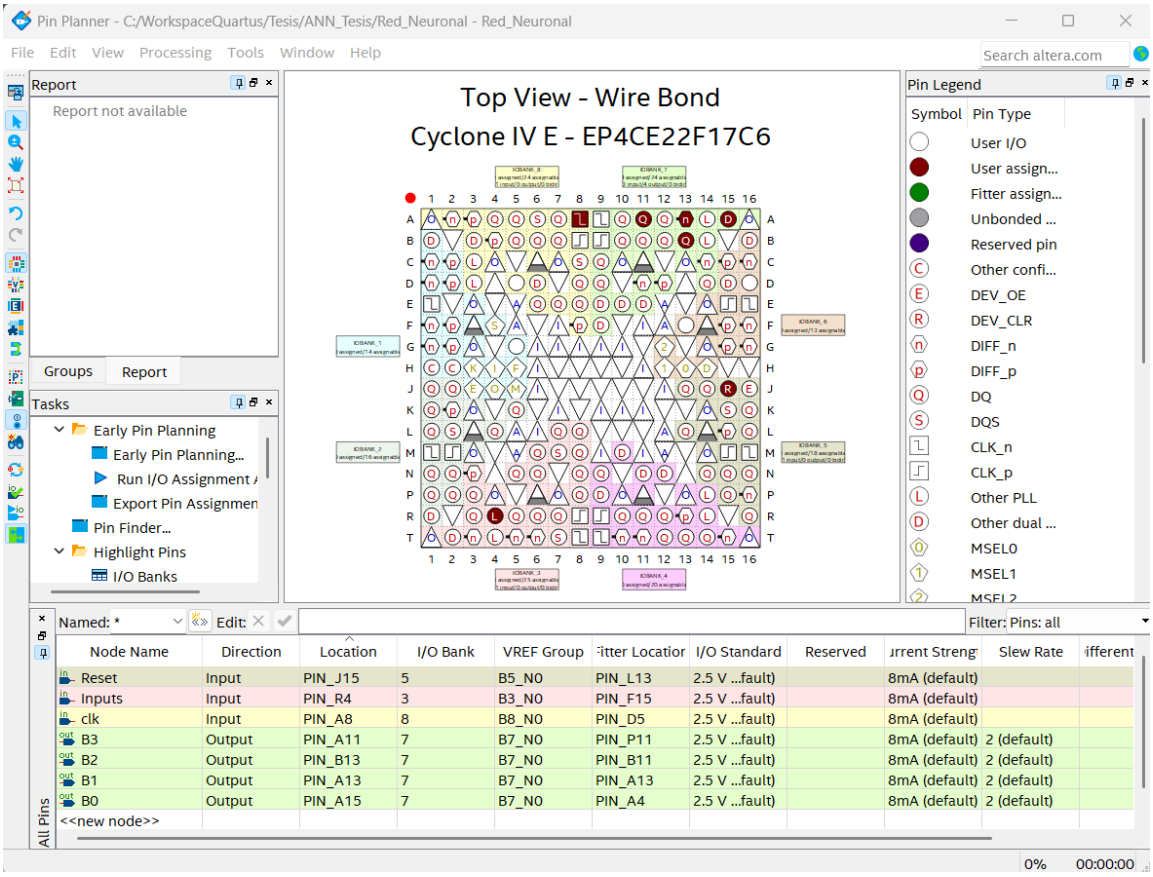


Figura 6.15: Herramienta *Pin Planner* para la asignación de pines.

Tipo	**Pin**	**Función**
Entradas	PIN_A8	Reloj de 50[MHz]
	PIN_R4	Receptor de parámetros de configuración y entrada
	PIN_J15	Botón de reinicio de la red
Salidas	PIN_A15	Led 01
	PIN_A13	Led 02
	PIN_B13	Led 03
	PIN_A11	Led 04

Tabla 6.1: Relación de pines asignados.

Capítulo 7

7. Pruebas y resultados

A partir de la implementación total del diseño presentado se procedió con la realización de pruebas y el análisis de resultados. De primera instancia, se probaron las funciones de activación con el fin de observar el error que producen, para así verificar si cumplen los estándares de este trabajo, en el que se plantea que todas las funciones deben tener un error promedio menor al 5 %. En la Tabla 7.1 se puede observar el error de cada función siendo que todas cumplen los estándares, considerando que la función ELU y Swish son diseños realizados desde cero, la Sigmoide y la Tangente hiperbólica son diseños que se encuentran en el Estado del Arte, la Leaky ReLU es el mismo modelo, pero con una restricción en la elección del hiperparámetro, y finalmente, la ReLU es una implementación exacta de la ecuación matemática.

Función de activación	**Error promedio**
ReLU	0
Sigmoide	0.00587
ELU	0.01526
TanH	0.00587
Leaky ReLU	0
Swish	0.01529

Tabla 7.1: Error promedio de cada función de activación.

De acuerdo con estos resultados se prosiguió con su implementación en Hardware. Analizando el consumo de recursos que produce cada diseño o modelo se puede observar la Tabla 7.2, en la cual se comprende que la función que menos recursos consume es la ReLU, resultado que se esperaba desde un inicio debido a su definición matemática, mientras que la función más robusta es la ELU, que de igual forma, se esperaba debido a que era la más compleja matemáticamente hablando y que más factores considera. Como se observa en la Tabla, los recursos que utiliza cada función de activación es un conjunto de los elementos lógicos consumidos por la implementación del diseño de la función, la obtención de su complemento a 2 y el complemento de la Sigmoide.

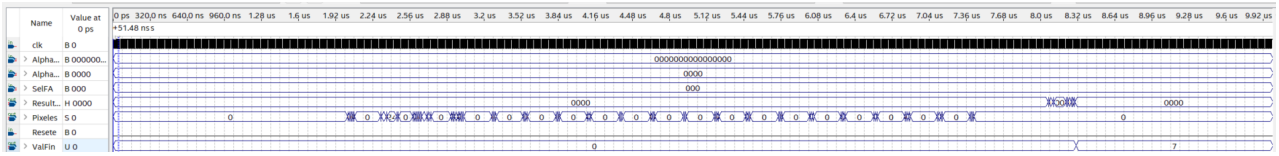
Diseño de la F.A	**Elementos lógicos consumidos**			
	Diseño de la función	**Complemento a 2**	**Complemento de la Sigmoide**	**Total**
ReLU	15	-	-	15
Sigmoide	44	18	31	93
ELU	388	18	-	406
TanH	68	18	-	86
Leaky ReLU	110	18	-	128
Swish	103	18	-	121

Tabla 7.2: Consumo de elementos lógicos por cada función de activación.

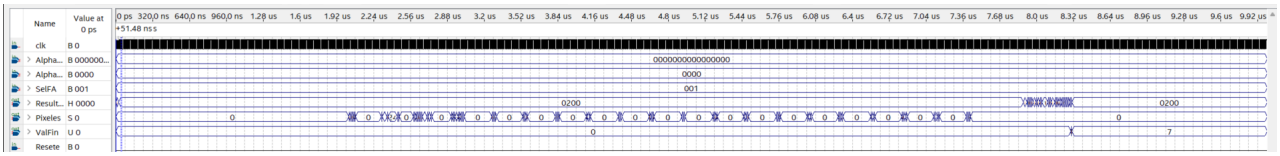
Gracias a la implementación de técnicas de optimización, el bloque de complemento a 2, se define una sola vez en el sistema, pero se utiliza repetidamente, provocando un uso eficiente de los recursos. De acuerdo con la Tabla, se observa como los recursos requeridos por cada diseño representan un consumo mínimo en comparación con su ecuación matemática que las define. Sin considerar las funciones de las cuales no existe una aproximación PWL que sirva como punto de comparación y la función ReLU y Leaky ReLU, las cuales su implementación es exacta a su definición, se compara el consumo de las funciones Sigmoide y Tangente hiperbólica realizados en este trabajo con el Estado del Arte y se encuentra que de acuerdo con el artículo [1], la implementación en este trabajo es más eficiente, puesto que consume menos recursos.

Finalmente, integrando la red en su totalidad se obtiene que consume 2,897 elementos lógicos, considerando ya todas las funciones de activación, representando un total del 13 % de todos los elementos lógicos disponibles, de los cuales 849 elementos lógicos corresponden a las funciones de activación, representando el 29 % de los elementos lógicos utilizados.

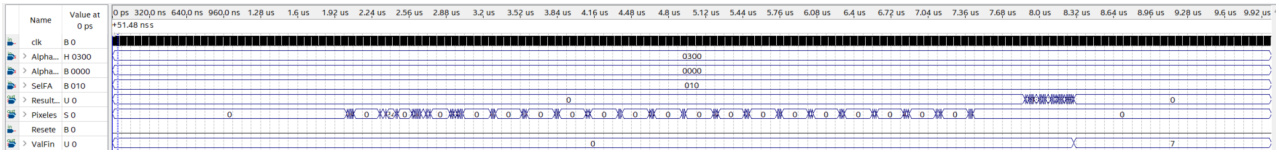
La prueba de la red se hará a partir de la base de datos de números manuscritos MNIST, que como se mencionaba en las especificaciones del objetivo, son imágenes del tamaño de 28x28, un total de 784 pixeles, con 10 clases de salida (0-9). De primera instancia, se probará con una imagen para corroborar un correcto funcionamiento de la red, esto se realizará a partir del simulador que provee Quartus Prime, con la herramienta *University Program VWF*. En la Figura 7.1 se observa un correcto funcionamiento de la red con cada función de activación, realizando una correcta clasificación, partiendo de una imagen prueba de un 7 manuscrito, que se ve en la Figura 7.2.



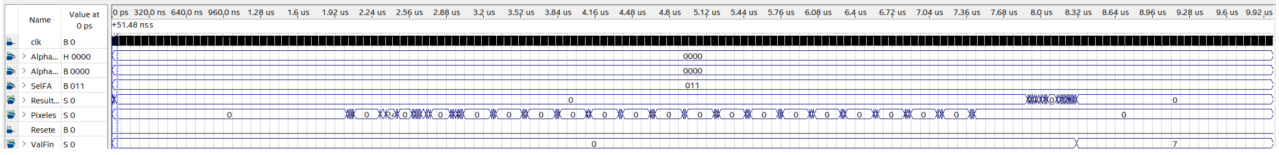
(a) Prueba de la función ReLU.



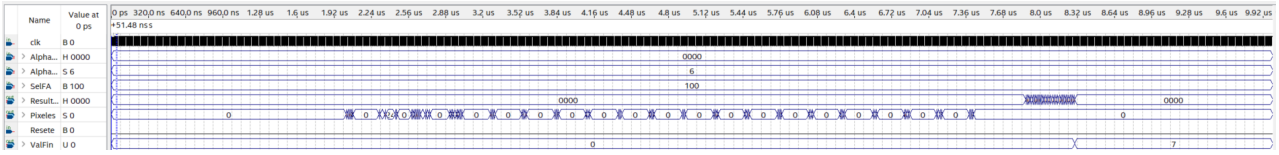
(b) Prueba de la función Sigmoide.



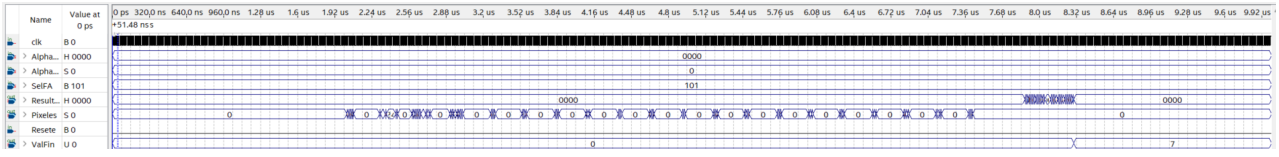
(c) Prueba de la función ELU con $\alpha = 0.75 = Hex'0300'$.



(d) Prueba de la función TanH.



(e) Prueba de la función Leaky ReLU con $\alpha = 6 \rightarrow (2^{-6})$.



(f) Prueba de la función Swish.

Figura 7.1: Resultados de la clasificación de la red.

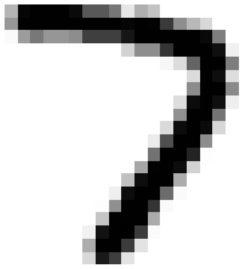


Figura 7.2: Numero 7 manuscrito de la base de datos MNIST.

Las señales que se observan en la prueba son:

- *Alpha ELU*, que representa el hiperparámetro de la función ELU
- *Alpha LReLU*, que representa el hiperparámetro de la función Leaky ReLU
- *SelFA*, es el selector de la función de activación
- *ResultFA*, es el resultado de cada neurona en cada capa
- *Resete*, es una señal creada para reiniciar el sistema si el usuario así lo desea
- *Pixeles*, es la impresión de las entradas que ingresan a la red, los 784 pixeles de la imagen, con el fin de verificar una correcta lectura.
- *ValFin*, será la variable que después de comparar los resultados de la última capa, encuentre el valor más grande e indique a que clase corresponde.

Analizando más de cerca los resultados que arroja la red, en la Figura 7.3 se puede observar el resultado de las neuronas de la última capa, así como que el valor más grande se encuentra en la posición 7, indicando que el resultado final es un 7.

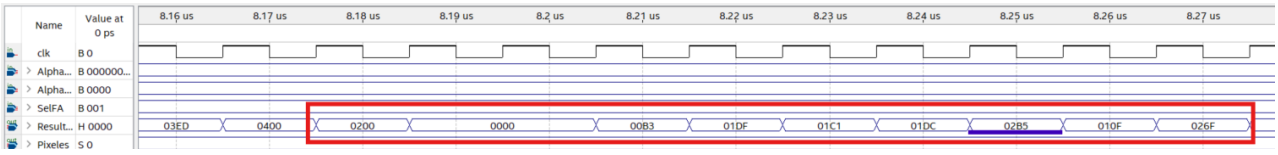


Figura 7.3: Resultados de las neuronas de la última capa con la función Sigmoide.

En la Figura 7.1 se observa como todos los resultados finales se obtienen en el mismo tiempo, en 828 ciclos de reloj, con un reloj de 50 [MHz], indicando que el FPGA toma 16.5 [μs] en clasificar una imagen.

Partiendo de un correcto funcionamiento de la red y sus funciones de activación, se realizan pruebas de consumo energético gracias a la herramienta *Power Analyzer Tool*, con el objetivo de reafirmar que se diseñó una red neuronal de bajo consumo energético. En la Figura 7.4 se observa que Quartus nos indica que existe un consumo de 116.94 [mW].

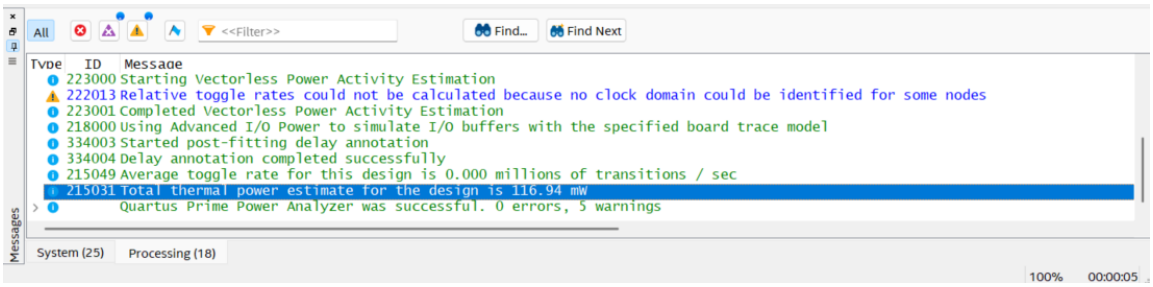


Figura 7.4: Consumo energético total estimado en el FPGA.

Instrumentando la red en FPGA, para visualizar los resultados en el dispositivo, se realiza la conexión pertinente, y se obtiene el resultado a partir de los LED integrados, que se observa en la Figura 7.5, que se tiene los tres LED menos significativos prendidos refiriéndose al valor en binario *"0111"*, el cual es un 7 en decimal.

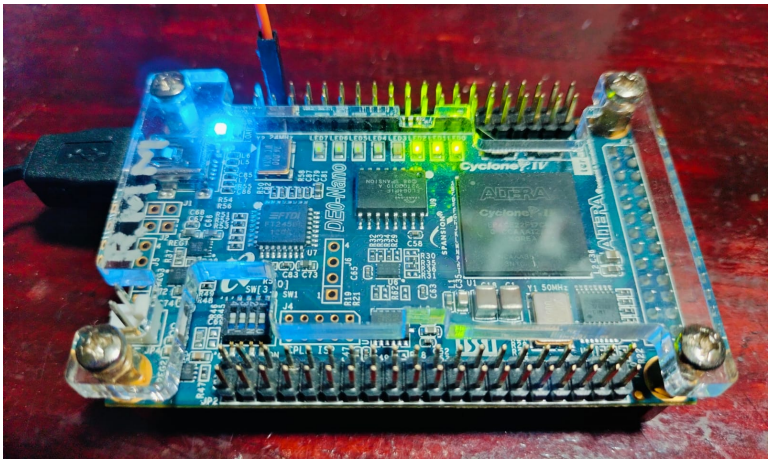


Figura 7.5: Resultado de la red en el FPGA, evaluando el 7.

A continuación se mostrará la matriz de confusión de la red para cada función, desarrollada a partir de los resultados dados, observando sus resultados correctos e incorrectos. Para la matriz se probaron 100 imágenes para cada función, considerando 10 imágenes para cada dígito. Como se observa en la Figura 7.6 la red realiza una correcta clasificación en su gran mayoría, cambiando los resultados de acuerdo a la función, pero catalogando bien. Para obtener un porcentaje certero de precisión de la red, se debería probar con un mayor número de imágenes, aunque el objetivo en el trabajo es comprobar un correcto funcionamiento de las funciones de activación, que queda demostrado en las matrices.

Real \ Observado	0	1	2	3	4	5	6	7	8	9
0	9	0	0	0	0	0	0	0	1	0
1	0	8	0	0	0	0	0	2	0	0
2	0	0	9	0	0	0	0	0	0	0
3	1	0	0	9	0	1	0	0	1	0
4	0	0	0	0	9	0	0	0	0	3
5	0	0	1	0	0	8	1	0	0	0
6	0	0	0	0	0	1	9	0	0	0
7	0	2	0	0	0	0	0	8	0	0
8	0	0	0	1	0	0	0	0	8	0
9	0	0	0	0	1	0	0	0	0	7

(a) Función ReLU.

Real \ Observado	0	1	2	3	4	5	6	7	8	9
0	8	0	0	1	0	0	0	0	0	0
1	0	8	0	0	0	0	0	1	0	0
2	0	0	9	0	0	0	0	0	0	0
3	1	0	0	7	0	1	0	0	2	0
4	0	0	0	0	8	0	0	0	0	1
5	0	0	1	0	0	9	1	0	0	0
6	0	0	0	0	0	0	9	0	0	0
7	0	2	0	0	0	0	0	9	0	0
8	1	0	0	2	0	0	0	0	8	0
9	0	0	0	0	2	0	0	0	0	9

(b) Función Sigmoide.

Real \ Observado	0	1	2	3	4	5	6	7	8	9
0	9	0	0	1	0	0	0	0	0	0
1	0	8	0	0	0	0	0	1	0	0
2	0	0	8	0	0	0	0	0	0	0
3	0	0	0	8	0	0	0	0	1	0
4	0	0	0	0	9	0	0	0	0	1
5	0	0	1	0	0	9	0	0	0	0
6	0	0	0	0	0	1	9	0	1	0
7	0	2	1	0	0	0	0	9	0	0
8	1	0	0	1	0	0	1	0	8	0
9	0	0	0	0	1	0	0	0	0	9

(c) Función ELU con $\alpha = 0.75$.

Real \ Observado	0	1	2	3	4	5	6	7	8	9
0	9	0	0	0	0	0	0	0	0	0
1	0	9	0	0	0	0	0	0	0	0
2	0	0	8	0	0	0	0	0	0	0
3	0	0	0	9	0	0	0	0	2	0
4	0	0	0	0	9	0	0	0	0	2
5	0	0	2	0	0	9	1	0	0	0
6	0	0	0	0	0	1	8	0	0	0
7	0	1	0	0	0	0	0	9	0	1
8	1	0	0	1	0	0	2	0	8	0
9	0	0	0	0	1	0	0	1	0	7

(d) Prueba de la función TanH.

Real \ Observado	0	1	2	3	4	5	6	7	8	9
0	8	0	0	0	0	0	0	0	0	0
1	0	9	0	0	0	0	0	1	0	0
2	0	0	9	0	0	0	0	0	0	0
3	1	0	0	9	0	0	0	0	0	0
4	0	0	0	0	9	0	0	0	0	1
5	0	0	0	0	0	9	0	0	0	0
6	0	0	1	0	0	1	8	0	1	0
7	0	1	0	0	0	0	0	9	0	1
8	1	0	0	1	0	0	2	0	8	0
9	0	0	0	0	1	0	0	0	0	8

(e) Función Leaky ReLU con $\alpha = 6$.

Real \ Observado	0	1	2	3	4	5	6	7	8	9
0	8	0	0	1	0	0	0	0	0	0
1	0	8	0	0	0	0	0	1	0	0
2	0	1	9	0	0	0	0	0	0	0
3	2	0	0	8	0	1	0	0	1	0
4	0	0	0	0	8	0	0	0	0	2
5	0	0	0	0	0	9	1	0	0	0
6	0	0	1	0	0	0	8	0	1	0
7	0	1	0	0	0	0	0	9	0	0
8	0	0	0	1	0	0	1	0	8	0
9	0	0	0	0	2	0	0	0	0	8

(f) Función Swish.

Figura 7.6: Matriz de confusión de cada función de la red.

Capítulo 8

8. Conclusiones

Se logró implementar una red neuronal *Feed-Foward* re-configurable en cuanto al número de capas, número de neuronas por capa y función de activación, diseñada de tal modo que existiera una optimización de recursos, pensada para la aplicación de proyectos que tienen limitados elementos lógicos disponibles, de bajo consumo energético y considerando el pequeño tamaño del FPGA que se utilizó, ideal para proyectos con limitaciones de espacio. Se obtuvo entonces una ANN que representó únicamente el 13 % de consumo de recursos considerando una configuración de 784 entradas, 30 neuronas de la capa oculta y 10 neuronas de la capa de salida, existiendo la posibilidad de agrandar mucho mas la red, aunque esto no quiere decir que por ello se obtendrán mejores resultados como se observo en la Tabla 5.2.

En el diseño de la red se implementaron las mismas técnicas de optimización que se observaron en modelos de redes neuronales en el Estado del Arte, las cuales integran un procesamiento de información serial-paralelo, la reutilización de neuronas, implementando únicamente el número de neuronas de la capa más grande, la implementación de un sólo bloque de la función de activación, la descripción de las funciones por medio de una aproximación lineal por partes (PWL) y un formato numérico de punto fijo.

Adicionalmente al desarrollo de una red optimizada, se logró implementar seis funciones de activación, la ReLU, la Sigmoide, la ELU, la Tangente hiperbólica, la Leaky ReLU y la Swish, las cuales representan las ecuaciones más mencionadas y con mejores resultados del Estado del Arte. Cada función cuenta con múltiples beneficios y alguna será mejor que la otra de acuerdo a la aplicación, agregando mayor valor a la red implementada, puesto que permite ser aplicada para una gran cantidad de problemas, debido a la capacidad de aprendizaje de cada función de patrones complejos, permitiendo su aplicación a distintas pruebas muy diferentes entre sí, resolviendo exitosamente problemas con una misma red.

Las funciones de activación fueron implementadas, como se mencionó, por aproximación PWL con el objetivo de reducir su consumo de recursos lógicos, así como de multiplicadores embebidos, aplicando técnicas de optimización no sólo en su diseño, sino también en su implementación, realizándose un análisis del hardware que genera cada sentencia para eficientar su código. Se logró diseñar

aproximaciones nunca antes hechas, para el caso la función ELU y Swish, un diseño modificado para el caso de la Leaky ReLU, y la implementación de aproximaciones ya conocidas como el de la Sigmoide y la Tangente hiperbólica, obteniendo mejores resultados que los encontrados en el Estado del Arte. Diseños que representaron un consumo mínimo de recursos, de acuerdo a la Tabla 7.2.

Finalmente se concluye que gracias al trabajo desarrollado se obtiene una red capaz de resolver un gran número de problemas, teniendo la capacidad de expandir o reducir su tamaño y profundidad fácilmente, por medio de una interfaz gráfica de usuario, sin necesidad de realizar modificaciones al proyecto, brindando una herramienta útil y viable de implementar en una gran cantidad de proyectos. Además de un diseño de cada función de activación capaz de ser implementada en cualquier dispositivo por muy limitados recursos que tenga, brindando modelos que nadie ha desarrollado de funciones que nunca se han implementado en Hardware de bajo consumo, pudiendo brindar un parteaguas para futuros proyectos.

Bibliografía

[1] A. Vaisnav, S. Ashok, S. Vinaykumar, and R. Thilagavathy, "Fpga implementation and comparison of sigmoid and hyperbolic tangent activation functions in an artificial neural network," in *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, 2022, pp. 1–4.

[2] V. Sumayyabeevi, J. J. Poovely, N. Aswathy, and S. Chinnu, "A new hardware architecture for fpga implementation of feed forward neural networks," in *2021 2nd International Conference on Advances in Computing, Communication, Embedded and Secure Systems (ACCESS)*, 2021, pp. 107–111.

[3] A. Rana, A. Singh Rawat, A. Bijalwan, and H. Bahuguna, "Application of multi layer (perceptron) artificial neural network in the diagnosis system: A systematic review," in *2018 International Conference on Research in Intelligent and Computing in Engineering (RICE)*, 2018, pp. 1–6.

[4] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence and Expert Systems (IJAE)*, vol. 1, no. 4, pp. 111–122, 2011.

[5] M. A. Mercioni and S. Holban, "A brief review of the most recent activation functions for neural networks," in *2023 17th International Conference on Engineering of Modern Electric Systems (EMES)*, 2023, pp. 1–4.

[6] S. Jeyanthi and M. Subadra, "Implementation of single neuron using various activation functions with fpga," in *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, 2014, pp. 1126–1131.

[7] R. M. Melamed, "Diseño de técnicas de optimización para la descripción de redes neuronales artificiales en fpga," Tesis para obtener el grado de maestría, Facultad de Ingeniería. Universidad Nacional Autónoma de México., Ciudad de México, México, 2024, tesis para optar por el grado de Maestría en Ingeniería.

[8] T. Rashid, *Make Your Own Neural Network*, 1st ed. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016.

[9] M. Dwijaya, U. Ali Ahmad, R. Wijayanto, and R. A. Nugrahaeni, "Model design of the image recognition of lung ct scan for covid-19 detection using artificial neural network," *JURNAL NASIONAL TEKNIK ELEKTRO*, vol. 11, pp. 21–28, 03 2022.

[10] D. B. Mehta, P. A. Barot, and S. G. Langhnoja, "Effect of different activation functions on eeg signal classification based on neural networks," in *2020 Fourth International Conference on Computing Methodologies and Communication (ICCMC)*, 2020, pp. 132–135.

[11] J. Wang, X. Zhang, L. Cheng, and Y. L. and, "An overview and metanalysis of machine and deep learning-based crispr grna design tools," *RNA Biology*, vol. 17, no. 1, pp. 13–22, 2020, pMID: 31533522. [Online]. Available: https://doi.org/10.1080/15476286.2019.1669406

[12] J. Moon, S. Park, S. Rho, and E. Hwang, "A comparative analysis of artificial neural network architectures for building energy consumption forecasting," *International Journal of Distributed Sensor Networks*, vol. 15, no. 9, p. 1550147719877616, 2019. [Online]. Available: https://doi.org/10.1177/1550147719877616

[13] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," *http://yann.lecun.com/exdb/mnist/*, 1998.

[14] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[dl] a survey of fpga-based neural network inference accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 1, Mar. 2019. [Online]. Available: https://doi.org/10.1145/3289185

[15] Y. Singh, M. Saini, and Savita, "Impact and performance analysis of various activation functions for classification problems," in *2023 IEEE International Conference on Contemporary Computing and Communications (InC4)*, vol. 1, 2023, pp. 1–7.

[16] F. Kamalov, A. Nazir, M. Safaraliev, A. K. Cherukuri, and R. Zgheib, "Comparative analysis of activation functions in neural networks," in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2021, pp. 1–6.

[17] R. Mahima, M. Maheswari, S. Roshana, E. Priyanka, N. Mohanan, and N. Nandhini, "A comparative analysis of the most commonly used activation functions in deep neural network," in *2023 4th International Conference on Electronics and Sustainable Communication Systems (ICESC)*, 2023, pp. 1334–1339.

[18] A. D. Rasamoelina, F. Adjailia, and P. Sinčák, "A review of activation function for artificial neural network," in *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, 2020, pp. 281–286.

[19] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," 2018. [Online]. Available: https://arxiv.org/abs/1811.03378

[20] D. C. Marcu and C. Grava, "The impact of activation functions on training and performance of a deep neural network," in *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*, 2021, pp. 1–4.

[21] S. Hitziger, "Modeling the variability of electrical activity in the brain," Theses, Université Nice Sophia Antipolis, Apr. 2015. [Online]. Available: https://theses.hal.science/tel-01175851

[22] J.-W. Lin, "Artificial neural network related to biological neuron network: a review," *Advanced Studies in Medical Sciences*, vol. 5, pp. 55–62, 01 2017.

[23] D. J. Matich, *Redes Neuronales: Conceptos Básicos y Aplicaciones.* Rosario, Argentina: Universidad Tecnológica Nacional, Facultad Regional Rosario, 2001, an introductory work on neural networks, covering foundational concepts and practical applications.

[24] R. Alkhatib, "Artificial neural network activation functions in exact analytical form (heaviside, relu, prelu, elu, selu, elish)," 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:238776182

[25] D. Misra, "Mish: A self regularized non-monotonic activation function," 2020. [Online]. Available: https://arxiv.org/abs/1908.08681

[26] T. Song, "Comparative analysis of activation functions in simple convolutional neural networks," in *2024 IEEE 6th International Conference on Civil Aviation Safety and Information Technology (ICCASIT)*, 2024, pp. 1031–1036.

[27] D. Pedamonti, "Comparison of non-linear activation functions for deep neural networks on mnist classification task," 2018. [Online]. Available: https://arxiv.org/abs/1804.02763

[28] D. Florek and M. Miłosz, "Comparison of an effectiveness of artificial neural networks for various activation functions," *Journal of Computer Sciences Institute*, vol. 26, pp. 7–12, 03 2023.

[29] H. Yang, J. Zhang, J. Sun, and L. Yu, "Review of advanced fpga architectures and technologies," *Journal of Electronics (China)*, vol. 31, no. 5, pp. 371–393, 2014.

[30] C. Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows.* Newnes, 2004. [Online]. Available: https://books.google.com.mx/books?id=dnuwr2xOFpUC

[31] K. S. Zaman, M. B. I. Reaz, S. H. Md Ali, A. A. A. Bakar, and 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, no. 11, pp. 6068–6088, 2022.

[32] S. Hariprasath and T. N. Prabakar, "Fpga implementation of multilayer feed forward neural network architecture using vhdl," in *2012 International Conference on Computing, Communication and Applications*, 2012, pp. 1–6.

[33] S. Jeyanthi and M. Subadra, "Implementation of single neuron using various activation functions with fpga," in *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, 2014, pp. 1126–1131.

[34] R. Maurya, D. Aggarwal, T. Gopalakrishnan, and N. N. Pandey, "Enhancing deep neural network convergence and performance: A hybrid activation function approach by combining relu and elu activation function," in *2023 Second International Conference on Informatics (ICI)*, 2023, pp. 1–5.

[35] M. B. B. G, A. Sarangi, A. Kashyap, and H. Yadav, "Comparative analysis of deep learning architectures for multilingual digit recognition," in *2024 International Conference on Data Science and Network Security (ICDSNS)*, 2024, pp. 1–6.

[36] D. Myers and R. Hutchinson, "Efficient implementation of piecewise linear activation function for digital vlsi neural networks," *Electronics Letters*, vol. 25, pp. 1662–1663, 1989. [Online]. Available: https://digital-library.theiet.org/doi/abs/10.1049/el%3A19891114

[37] K. M. . C. H.Amin and B. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEE Proc-Circuits Devices Systems*, vol. 144, no. 6, pp. 313 – 317, 1997.

[38] F. Pardo and J. Boluda, *VHDL: Lenguaje para Síntesis y Modelado de Circuitos*, 1st ed. RA-MA, 1999.

[39] Terasic Technologies Inc., *DE0-Nano User Manual*, Terasic Technologies, 2011, altera FPGA-based Development Board. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=139&No=593

[40] I. Tsmots, O. Skorokhoda, and V. Rabyk, "Hardware implementation of sigmoid activation functions using fpga," in *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*, 2019, pp. 34–38.

[41] J. Gao, Y. Qian, Y. Hu, X. Fan, W.-S. Luk, W. Cao, and L. Wang, "Leta: A lightweight exchangeable-track accelerator for efficientnet based on fpga," in *2021 International Conference on Field-Programmable Technology (ICFPT)*, 2021, pp. 1–9.