

# Universidad de Alcalá Escuela Politécnica Superior

## Máster Universitario en Ingeniería de Telecomunicación

### Trabajo Fin de Máster

Diseño de arquitecturas empotradas para la implementación en  
tiempo real de redes neuronales profundas

ESCUELA POLITECNICA  
SUPERIOR

**Autor:** Jorge Martín Catalán

**Tutor:** Álvaro Hernández Alonso

2022



UNIVERSIDAD DE ALCALÁ  
ESCUELA POLITÉCNICA SUPERIOR

**Máster Universitario en Ingeniería de Telecomunicación**

**Trabajo Fin de Máster**

**Diseño de arquitecturas empotradas para la implementación en tiempo real de redes neuronales profundas**

Autor: Jorge Martín Catalán

Tutor: Álvaro Hernández Alonso

**Tribunal:**

**Presidente:** Daniel Pizarro Pérez

**Vocal 1º:** M<sup>º</sup> del Pilar Jarabo Amores

**Vocal 2º:** Álvaro Hernández Alonso

Fecha de depósito: 17 de julio de 2022



**A mi familia, profesores y amigos**

*“Conoce a tu enemigo y concéete a ti mismo; en cien batallas, nunca saldras derrotado.”*  
Sun Tzu



# Resumen

Este documento detalla la implementación de algoritmos de *Deep Learning* en distintos tipos de plataformas embebidas como son los *SoC/FPGA* y los microprocesadores. Además se comparan las diferencias en términos de latencias, errores de cuantificación y recursos consumidos. Se parte de una red ya implementada y probada, con sus bases de datos etiquetadas y separadas para realizar la implementación. El problema que trata de resolver se utilizará como base, pero debido a que se quieren probar diversas arquitecturas como perceptron multicapa, redes convolucionales y las recurrentes, se resolverá modificando la estructura de la red neuronal con redes alternativas a las propuestas en dicho trabajo.

**Palabras clave:** FPGA/SoC, Deep Learning, Microprocesadores, NILM, Sistemas empotrados.





# Abstract

This document details the implementation procedure of different Deep Learning algorithms in embedded platforms such as SoC/FPGA or Microprocessors. Also, the differences in terms of latency, quantification errors and resources are compared. The start point is a tested network and its tagged database. This network will be modified in order to try different types of networks such as Multilayer perceptron, Convolutional Network and Recurrent Network.

**Keywords:** FPGA/SoC, Deep Learning, Microprocesors, NILM, embedded system.



# Resumen extendido

El campo del *Deep Learning* es cada vez más importante dentro del campo de la inteligencia artificial y de la vida diaria. Su implementación no tiene cabida únicamente dentro de los servidores con aceleradores para realizar las inferencias en tiempo real; en múltiples casos, por cuestiones de latencias, seguridad o de comunicaciones es necesario realizar los cálculos en el *edge*. Por lo tanto, su implementación dentro de dispositivos embebidos como las *FPGA* o los microcontroladores es un aspecto primordial.

El propósito de este trabajo se centra en la implementación de este tipo de redes dentro de los dispositivos empujados comentados previamente. Por parte de la *FPGA*, el desarrollo del algoritmo en un lenguaje de descripción *hardware* como *VHDL* serviría para obtener un funcionamiento óptimo. Aun así, este desarrollo sería complejo y tendría un tiempo de implementación largo. Por lo tanto, para el desarrollo del trabajo se observarán técnicas de alto nivel que permiten un desarrollo más rápido y sencillo con el paquete *hls4ml*, que permite desde *python* generar código *hls* que herramientas como *Vivado HLS* son capaces de convertir en *firmware*.

Por su parte, con el microcontrolador también se observarán herramientas destinadas a la implementación de redes neuronales. Por ejemplo, los procesadores de *STM* disponen de un paquete denominado *X-Cube-AI* que permite importar las redes neuronales y convertirlas en una *API* compatible con sus dispositivos.

Todas estas herramientas se evaluarán a la práctica y se explicará el funcionamiento y utilidades de las que disponen. Como caso práctico se ha seleccionado la implementación de técnicas *NILM* para la monitorización de hogares, en concreto, para la clasificación de electrodomésticos en función de sus características eléctricas. Con el objetivo de explorar las distintas arquitecturas de *Deep Learning* se utilizarán distintos tipos de neuronas como perceptrones multicapa, convolucionales y recurrentes con distintos tipos de funciones de activación.

Con los resultados obtenidos se compararán las principales características de cada dispositivo en función del tamaño y tipo de red, y también se compararán entre los dos tipos de componentes, destacando los puntos fuertes y débiles de las *FPGA* frente a los microcontroladores en la implementación del sistema.



# Índice general

Resumen	vii
Abstract	ix
Resumen extendido	xi
Índice general	xiii
Índice de figuras	xv
Índice de tablas	xvii
Lista de acrónimos	xvii
Lista de símbolos	xix
<b>1 Introducción</b>	<b>1</b>
1.1 Objetivos . . . . .	1
1.2 Estructura global . . . . .	2
<b>2 Estudio teórico</b>	<b>3</b>
2.1 Redes Neuronales . . . . .	3
2.2 Técnicas NILM . . . . .	4
2.3 Red neuronal base . . . . .	5
<b>3 Arquitecturas basadas en dispositivos FPGA</b>	<b>9</b>
3.1 Red neuronal básica . . . . .	10
3.1.1 Diseño de la red neuronal básica en <i>Keras</i> . . . . .	10
3.1.2 Conversión de una red neuronal a HLS . . . . .	12
3.1.2.1 Análisis de recursos y latencia resultante . . . . .	16
3.1.3 Pruebas realizadas en HLS . . . . .	17
3.1.4 Diseño del sistema en <i>Vivado</i> . . . . .	17
3.1.5 Diseño del sistema en <i>Vitis</i> . . . . .	19

3.2	Red neuronal de dos capas densamente conectadas . . . . .	22
3.2.1	Diseño de la red neuronal de dos capas en Keras . . . . .	22
3.2.2	Conversión de la red de dos capas densas a HLS . . . . .	25
3.2.2.1	Estudio de la cuantificación en el modelo de dos capas densas . . . . .	25
3.2.2.2	Análisis de recursos y latencia resultante . . . . .	27
3.2.3	Pruebas realizadas en HLS . . . . .	29
3.2.4	Implementación en Vivado . . . . .	30
3.2.5	Implementación en Vitis . . . . .	31
3.3	Red neuronal de dos capas utilizando CNN . . . . .	34
3.3.1	Diseño de la red neuronal convolucional en <i>Keras</i> . . . . .	34
3.3.2	Convertir la red neuronal convolucional a HLS . . . . .	35
3.3.2.1	Estudio de la cuantificación en el modelo convolucional . . . . .	35
3.3.2.2	Análisis de recursos y latencia resultante . . . . .	37
3.3.3	Implementación en Vivado . . . . .	38
3.4	Comparativa de las distintas estructuras en FPGA . . . . .	40
<b>4</b>	<b>Implementación de Redes Neuronales en sistemas basados en Microprocesador</b>	<b>43</b>
4.1	Red neuronal básica en uC . . . . .	43
4.1.1	Utilización del paquete STMicroelectronics-X-Cube-AI . . . . .	44
4.1.2	Utilización de la <i>API</i> y resultados . . . . .	46
4.2	Red neuronal de dos capas densamente conectadas . . . . .	47
4.3	Red neuronal convolucional . . . . .	49
4.4	Red neuronal recurrente . . . . .	50
4.5	Comparativa entre la implementación de las distintas neuronas en $\mu C$ . . . . .	54
<b>5</b>	<b>Conclusiones y líneas futuras</b>	<b>55</b>
<b>6</b>	<b>Presupuesto</b>	<b>57</b>
	<b>Bibliografía</b>	<b>59</b>

# Índice de figuras

2.1	Esquema del perceptrón [1]. . . . .	3
2.2	Ejemplo de una red neuronal densamente conectada [2]. . . . .	4
2.3	División de arquitectura en NILM [3]. . . . .	6
2.4	Red original descrita para la detección NILM [3]. . . . .	7
3.1	Esquema de desarrollo de la aplicación. . . . .	10
3.2	Placa de desarrollo <i>Zedboard</i> de <i>Digilent</i> . [4]. . . . .	10
3.3	Estructura de la <i>Zynq</i> . . . . .	11
3.4	Topología red neuronal básica. . . . .	12
3.5	Resultados de entrenamiento de la red neuronal básica. . . . .	13
3.6	Significado del Reuse factor [5]. . . . .	14
3.7	Diagrama implementado en la red básica. . . . .	15
3.8	Utilización en función del <i>reuse factor</i> para la red neuronal básica. . . . .	16
3.9	Latencia en función del <i>reuse factor</i> para la red neuronal básica. . . . .	17
3.10	Resultados en <i>Vivado HLS</i> de la Red Básica. . . . .	17
3.11	Configuración de puertos del IP básico. . . . .	18
3.12	Esquema básico. . . . .	18
3.13	Potencia consumida por la implementación. . . . .	19
3.14	Análisis temporal del esquema básico para distintas frecuencias. . . . .	20
3.15	Resultados de la red neuronal básica implementada en la tarjeta. . . . .	21
3.16	Resultados en ILA de la red neuronal básica implementada en la tarjeta. . . . .	22
3.17	Topología de la red neuronal de dos capas densas. . . . .	23
3.18	Resultados de entrenamiento de la red neuronal de dos capas densas. . . . .	24
3.19	Diagrama del modelo HLS con dos capas densamente conectadas. . . . .	25
3.20	Cuantificación de los parámetros en la red neuronal de dos capas. . . . .	26
3.21	Respuesta a la variación de la cuantificación en la estructura de dos capas variando la coma fija. . . . .	26
3.22	Respuesta a la variación de la cuantificación en la estructura de dos capas. . . . .	27

3.23 Recursos consumidos en la fase de síntesis variando el <i>Reuse factor</i> en la estructura de dos capas. . . . .	28
3.24 Latencia variando el <i>Reuse factor</i> en la estructura de dos capas. . . . .	29
3.25 Resultados en <i>HLS</i> de estructura con dos capas. . . . .	30
3.26 Bloque IP de la red de dos capas densamente conectadas. . . . .	30
3.27 Esquema de conexionado de bloques en Vivado con dos capas. . . . .	31
3.28 Análisis temporal del esquema de dos capas densamente conectadas para distintas frecuencias. . . . .	32
3.29 Potencia en la arquitectura de dos capas densamente conectadas. . . . .	33
3.30 Resultados de la red neuronal implementada en la <i>Zedboard</i> . . . . .	33
3.31 Diagrama red neuronal convolucional. . . . .	34
3.32 Resultados de entrenamiento de la red neuronal convolucional. . . . .	36
3.33 Cuantificación de los parámetros de la red neuronal convolucional. . . . .	37
3.34 Variación del tamaño de palabra de la red neuronal convolucional. . . . .	37
3.35 Variación del tamaño de palabra con la parte entera fija en 11 bits de la red neuronal convolucional. . . . .	38
3.36 Utilización de la CNN en función del <i>Reuse Factor</i> . . . . .	39
3.37 Latencia de la CNN en función del <i>Reuse Factor</i> . . . . .	39
3.38 Análisis temporal de la red CNN para distintas frecuencias . . . . .	40
3.39 Latencia de la CNN en función del <i>Reuse Factor</i> . . . . .	40
4.1 Interfaz X-Cube-AI. . . . .	44
4.2 Arquitectura red básica X-Cube-AI. . . . .	45
4.3 Tiempo en realizar la inferencia en la validación de la red básica. . . . .	45
4.4 Precisión de la red básica según X_cubeX_AI. . . . .	45
4.5 Captura de los resultados de la red básica. . . . .	47
4.6 Análisis de la red de dos capas. . . . .	48
4.7 Comparativa de resultados de la red de dos capas. . . . .	48
4.8 Comparativa de tiempos de la red de dos capas. . . . .	49
4.9 Análisis de la red convolucional. . . . .	49
4.10 Error de cuantificación red convolucional para $\mu C$ . . . . .	50
4.11 Tiempo en ordenador de la red convolucional. . . . .	50
4.12 Esquema de la red recurrente. . . . .	51
4.13 Resultados de entrenamiento de la red neuronal básica. . . . .	52
4.14 Análisis de la red recurrente. . . . .	53
4.15 Tiempo en la evaluación en ordenador de la red recurrente. . . . .	53
4.16 Precisión entre el modelo de STM y el teórico. . . . .	53



# Índice de tablas

3.1	Resumen de la red neurona básica. . . . .	11
3.2	Utilización básica con <i>Reuse Factor</i> = 4. . . . .	16
3.3	Recursos de la red neuronal en Vivado. . . . .	19
3.4	Resumen de la red neurona de dos capas densamente conectadas. . . . .	22
3.5	Recursos de NN en Vivado para estructura de dos capas con <i>Reuse factor</i> de 1. . . . .	28
3.6	Recursos de NN en Vivado para estructura de dos capas con interfaz AXI. . . . .	29
3.7	Recursos de NN en implementación del diagrama de dos capas. . . . .	31
3.8	Resumen de la red neurona convolucional. . . . .	35
3.9	Recursos de NN en Vivado para estructura CNN. . . . .	38
3.10	Recursos de NN en Vivado para estructura CNN en la implementación. . . . .	39
4.1	Compresión obtenida para la red básica. . . . .	44
4.2	Resultados en $\mu C$ para la red básica. . . . .	47
4.3	Compresión obtenida para la red de dos capas. . . . .	48
4.4	Resultados en $\mu C$ para la red de 2 capas. . . . .	49
4.5	Compresión obtenida para la red de convolucional. . . . .	50
4.6	Resultados en $\mu C$ para la red convolucional. . . . .	51
4.7	Resumen de la red neurona básica. . . . .	51
4.8	Compresión obtenida para la red de recurrente. . . . .	52
4.9	Resultados en $\mu C$ para la red recurrente. . . . .	54
6.1	Costes Materiales . . . . .	57
6.2	Costes Profesionales . . . . .	57
6.3	Costes Totales . . . . .	58



# Glosario de Acrónimos y abreviaturas

API	Application Programming Interface
DSP	Digital Signal Processor
E/S	Entrada/Salida
FPGA	Field Programmable Gate Array
HLS	High-Level Synthesis
LSTM	Long Short-Term Memory
LUT	Look-Up Table
NILM	Non-Intrusive Load Monitoring
PS	Processing System
PL	Programmable Logic
RAM	random access memory
RTL	Real Time Logic
SoC	System On Chip
uC / MCU / $\mu$ C	Microcontrolador



# Capítulo 1

## Introducción

A día de hoy, las redes neuronales se aplican en la inmensa mayoría de aspectos de nuestra vida. Por ejemplo, al utilizar plataformas de vídeo como *Youtube* o *TikTok*, o redes sociales como *Facebook* o *Twitter*, hay múltiples redes neuronales encargadas de predecir el próximo contenido a visualizar en función de nuestros gustos o intereses. Otro ámbito en el que está encontrando gran relevancia es en el vehículo autónomo, aspectos como la identificación de peatones o guiado del vehículo toman especial importancia.

Otra rama en la que han mejorado bastante la calidad de vida es en la sanitaria. Se aplican en campos como la detección previa de enfermedades o en el reconocimiento de hábitos de vida saludables. Este último tema se detallará en la sección 2.3 dado que ha servido como iniciativa del presente trabajo con la aplicación de redes neuronales dentro de *Smart Meters* para la clasificación de electrodomésticos en función de sus cualidades eléctricas.

En cuanto a la ejecución de las redes neuronales se parte de dos paradigmas, desarrollar la algoritmia en el *edge* o desarrollarla en el cloud. Para el primer caso, se suelen utilizar microcontroladores o microprocesadores para llevar a cabo la inferencia de la red como puede ser en un móvil. Sin embargo, las *FPGA* también pueden resultar de gran utilidad en este terreno ya que permiten reducir los tiempos de cálculo debido a su alto grado de paralelización. Aunque su uso no se centra únicamente al *edge*, empresas como *Xilinx* tienen ramas de productos como las tarjetas aceleradoras *Alveo*[6] o Virtex especiales para *Data Centers*. Por ejemplo, las instancias Amazon EC2 F1 [7] de *Amazon Web Services* utilizan *FPGAs* de *Xilinx* para realizar una aceleración en *hardware* de los algoritmos.

Este trabajo se centra en aplicaciones sobre el *edge* para ofrecer una visión de la implementación de redes neuronales en microprocesadores y *FPGA*, a fin de comparar el comportamiento y las características de ambos entornos.

### 1.1 Objetivos

Los objetivos que trata de cubrir el TFM son:

- Diseño de redes con distintos tipos de arquitecturas como perceptrón multicapa, convolucionales o recurrentes.

- Implementación en *FPGA* de las distintas redes neuronales partiendo del fichero en python y probándolo en la tarjeta.
- Implementación en microcontrolador de las distintas redes neuronales partiendo del fichero en python y probándolo en la tarjeta.
- Evaluación de las herramientas utilizadas.
- Comparativa de resultados entre las diferentes arquitecturas de redes neuronales en cada sistema.
- Comparativa de resultados entre los dos sistemas.

## 1.2 Estructura global

En el capítulo 2 se comenzará con el análisis del estado del arte. En el que se explicará los trabajos previos o a la situación actual con respecto a las redes neuronales y a los *Smart Meters*. Además, también se explicará el trabajo previo que ha servido como punto de partida a este documento.

A continuación, en el capítulo 3 se comentará el proceso de implementación para la *FPGA* analizando las herramientas, programas utilizados y resultados. A lo largo de dicho capítulo se analizarán de igual manera las redes neuronales que se van a llevar a la práctica en la *FPGA* y en el micro. El proceso en el microprocesador se continuará en el apartado siguiente presentando de igual manera el *software* empleado y los resultados obtenidos teóricamente y sobre la tarjeta.

Para comparar los resultados obtenidos en los dos capítulos anteriores se dispondrá del capítulo 5 donde también se anotarán las conclusiones obtenidas y las líneas de trabajo futuras que surgen de este TFM.

# Capítulo 2

## Estudio teórico

### 2.1 Redes Neuronales

El terreno de la inteligencia artificial ha tenido un gran crecimiento en las últimas décadas. En concreto, en esta última con la llegada de las redes neuronales profundas (*Deep Neuronal Networks*). Son inspiradas en el modelo del cerebro y de las conexiones entre distintas neuronas para emular la inteligencia humana. La neurona más sencilla implementada es el perceptrón que se puede visualizar en la figura 2.1. Consiste en una capa de entradas  $X_n$  a las que se les aplica una serie de pesos  $W_n$  y se suma su resultado. Adicionalmente, se le puede insertar un *bias*. Finalmente, al resultado de esta expresión se suele aplicar múltiples funciones no lineales con el objetivo de evitar un comportamiento lineal. Por lo tanto la función implementada por un perceptrón será:

$$Y = f(X_n \cdot W_n + b) \tag{2.1}$$

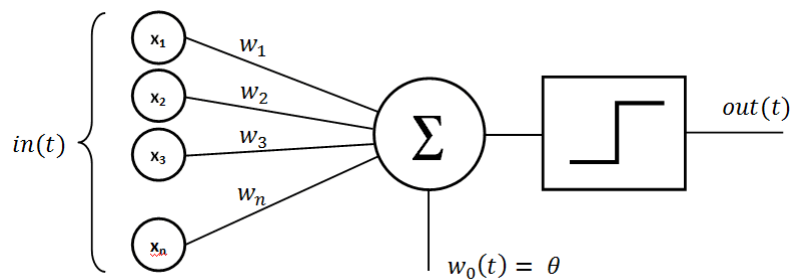


Figura 2.1: Esquema del perceptrón [1].

El funcionamiento de una única neurona sería demasiado sencillo por lo que se agregan más en la misma capa o en distintas con el objetivo de aprender comportamientos más complejos de un problema de regresión o de clasificación. Las redes neuronales cuentan con una capa de entrada a la red conectada con distintas neuronas ocultas densamente conectadas. Todas estas pueden tener distintos tipos de neuronas de activación. Finalmente se dispondrá de una capa de salida con el resultado de la clasificación. Un ejemplo de esta red conectada se observa en la figura 2.2. Se define como densamente conectada por que todas las salidas de la capa anterior están conectadas a cada neurona de la capa siguiente.

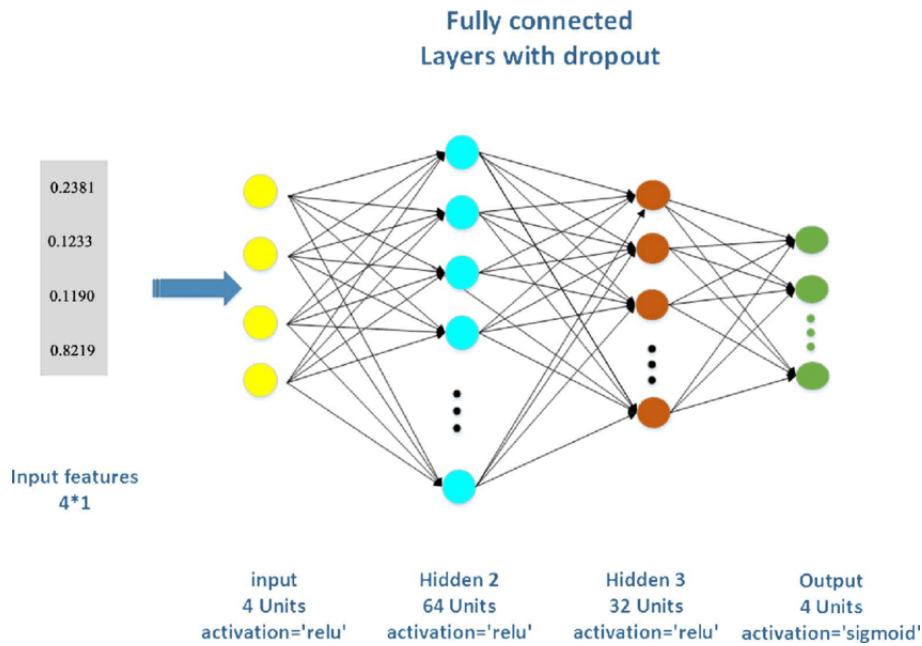


Figura 2.2: Ejemplo de una red neuronal densamente conectada [2].

El perceptrón es el tipo de neurona más sencillo, encargado de desarrollar una combinación lineal con una función de activación, pero existen otro tipo de neuronas con un mejor funcionamiento en ciertas aplicaciones. Por ejemplo las redes convolucionales [8] destacan en el procesamiento de imagen y se han utilizado en sistemas como AlexNet [9] o Yolo [10]. Este tipo de neurona se encarga de aplicar un filtro sobre la imagen en lugar de la combinación lineal realizada por el perceptrón. Las capas neuronales suelen venir seguidas con capas de *pooling* que se encargan de evitar que aumente demasiado el número de parámetros de la red. Para ello seleccionan el número mayor o medio de un subconjunto. Otra clase de redes que funcionan especialmente bien con series temporales son las recurrentes [11], entre las que destacan las LSTM. Su principal característica es considerar valores previos para las siguientes clasificaciones.

Otro concepto relativamente importante dentro de las redes neuronales es el de entrenamiento y *backpropagation*[12]. Dado que se trata de aprendizaje supervisado, se requiere un conjunto de datos para alimentar la red. Con ello realiza el algoritmo de *backpropagation* con el que intenta, a partir del error cometido entre el predicho y el correcto, corregir los parámetros de la red para aprender a resolver el problema. A lo largo del entrenamiento pueden suceder dos problemas distintos denominados *overfitting* y *underfitting*. El último consiste en que no hay suficientes datos o que las iteraciones del algoritmo son insuficientes para resolver el problema. Mientras que el otro consiste en que la red memoriza los datos de entrenamiento pero es incapaz de comportarse correctamente ante nuevos datos. Este problema se soluciona separando el conjunto de datos en entrenamiento y validación que servirán para probar que no sé este realizando este sobreentrenamiento.

## 2.2 Técnicas NILM

El área de aplicación en la que se enfoca este trabajo es la monitorización de carga, que permite extraer las características de un electrodoméstico del sistema. Para ello existen las técnicas NILM, por las que tomando medidas de energía eléctrica del edificio se puede estimar el consumo y estado de un único



dispositivo. La estimación se suele dividir en dos fases, una de bajo nivel consistente en la adquisición de datos y otra de alto nivel que separará las cargas de los diferentes elementos que componen la red.

Para la adquisición de los datos se requiere de un medidor conectado a la red eléctrica llamado *Smart Meter* [13]. En cuanto a la separación de la energía hay dos diferencias principales. La primera basada en eventos y la segunda que no está basada en eventos. Los eventos consisten en las transiciones de ON a OFF o viceversa de los dispositivos y su algoritmia se enfoca en detectarlos y clasificarlos. Mientras, los que no están basados en eventos se encargan de predecir el estado de los dispositivos. Puesto que la tarea de alto nivel consiste en la clasificación principalmente, se emplea la inteligencia artificial para la resolución del problema. Se pueden optar por métodos supervisados o no supervisados, aunque para este trabajo se hará empleo, como se vio en el apartado anterior, de redes neuronales.

Las aplicaciones de estos sistemas se dividen principalmente en consultar el consumo eléctrico de cierto electrodoméstico por parte del usuario, permitiendo modificar sus hábitos de consumo o cambiar de dispositivo a uno con menor gasto eléctrico. El otro objetivo de los *Smart Meters* y las técnicas *NILM* es el soporte a las personas de la tercera edad, habilitando de manera no intrusiva visualizar y monitorizar las actividades que desarrollan para constatar que están en un buen estado de salud.

## 2.3 Red neuronal base

Como se ha podido apreciar en capítulos previos, este trabajo tiene como antecedente otro TFG[3] de la universidad. Su documento se fundamenta en la definición de técnicas no intrusivas para separar el consumo eléctrico de distintos electrodomésticos para el cuidado de personas de tercera edad. Divide por lo tanto la arquitectura de procesamiento en dos zonas principales que se observan en la figura 2.3. La primera es la de bajo nivel que ya se comentó en la sección pasada que se encargará de la adquisición de muestras, un preprocesado encargado del cálculo de la corriente de la red y un último bloque dedicado a encontrar los eventos de ON-OFF que sucedan en las muestras. Para la adquisición, en lugar de conectar el dispositivo a la red se empleó el *Blued Dataset* consistente en una base de datos etiquetada de una familia de Pensylvania, US. La frecuencia de muestreo con la que se adquirió la biblioteca fue con 12 KHz y el experimento duro una semana.

El apartado de alto nivel es el encargado de la clasificación de los electrodomésticos. Para este tipo de señales temporales eléctricas se optó por las redes neuronales recurrentes como mecanismo de clasificación. Siendo las *LSTM* la arquitectura seleccionada para su implementación.

En cuanto al diseño de la red neuronal, en el trabajo se presentan dos opciones, siendo la segunda la más avanzada y representada en la figura 2.4. Como se observa, se emplean para la clasificación dos capas LSTM que destacan por su buen funcionamiento en señales temporales junto con capas densamente conectadas para extraer el porcentaje de pertenencia a cada clase. La precisión de esta red para eventos de *ON/OFF* es del 93.90 % mientras que para eventos únicamente de *ON* es del 98.9 %.

Dado que el presente trabajo se dirige a la implementación de redes neuronales dentro de dispositivos embebidos, el apartado de adquisición queda fuera del estudio y no se llevará ni a la *FPGA* ni al microcontrolador, quedando como una línea de trabajo futuro. Por lo tanto, el TFM consistirá en la implementación de distintos clasificadores para extraer el tipo de electrodoméstico conectado a la red

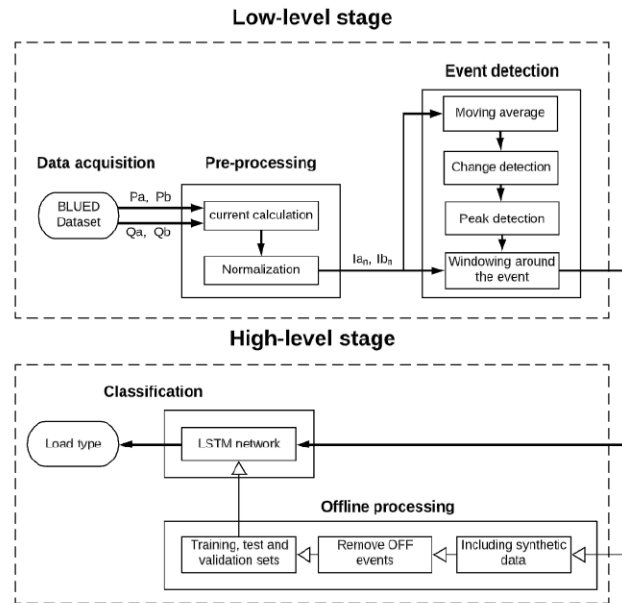


Figura 2.3: División de arquitectura en NILM [3].

eléctrica. El punto de partida será el Dataset BLUED preprocesado y con la detección de los eventos para el entrenamiento de diversas redes neuronales.

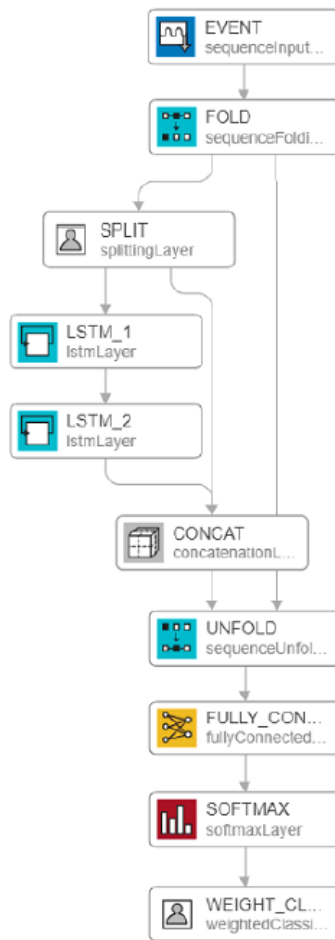


Figura 2.4: Red original descrita para la detección NILM [3].



## Capítulo 3

# Arquitecturas basadas en dispositivos FPGA

A partir de la red neuronal descrita en la sección previa, en este capítulo se adaptará para ser implementadas en una FPGA. Para ello habrá que tener en cuenta los distintos niveles a los que se trabajará y la cuantificación, ya que esta irá vinculada con la utilización de recursos, la latencia y el error cometido con respecto a la red teórica.

El diseño de los distintos sistemas partirá de los resultados obtenidos teóricamente en python y se empleará el paquete *hls4ml* para adaptar la red a *HLS*. Esto permitirá que pueda ser probado y sintetizado por el programa *Vivado HLS de Xilinx* para las distintas arquitecturas de *FPGAs*. Posteriormente, estas serán sintetizadas e implementadas para el componente seleccionado y finalmente, dado que se introducirán en una arquitectura *SoC* como periférico, también se tendrá que desarrollar el código correspondiente a la *PS* que gestione el periférico de la *PL*. Para este último apartado se empleará el software *Vitis* de *Xilinx*. Este esquema de desarrollo se puede visualizar en la 3.1. Además, se probará cada etapa del sistema, pudiéndose volver a fases anteriores del diseño con el objetivo de subsanar los problemas posibles.

El dispositivo seleccionado para el desarrollo de la aplicación es el componente *Zynq-7020* de *Xilinx* implementado en la placa de desarrollo *Zedboard* de *Digilent* de la figura 3.2. Se ha seleccionado este dispositivo para probar la red neuronal con recursos limitados, haciendo que la comparación con la plataforma en  $\mu C$  analizada en el próximo capítulo sea más justa.

El dispositivo *SoC* dispone de 2 núcleos de ARM-A9 formando la parte de procesamiento (PS) y una lógica programable con 85.000 celdas lógicas, 4.9 Mb de bloques de RAM y 220 DSP que tendrán una gran importancia como se verá más adelante. Además incorpora más módulos y periféricos como se observa en la figura 3.3 como la UART que se utiliza para visualizar los resultados del test.

La arquitectura de la red neuronal variará para analizar la dificultad de la implementación desde una red neuronal básica con 4 neuronas de clasificación, hasta una red convolucional. La red presentada en el estado del arte no es posible su síntesis a *FPGA* debido a que *HLS4ML* no tiene compatibilidad [14] con las capas recurrentes *LSTM*.

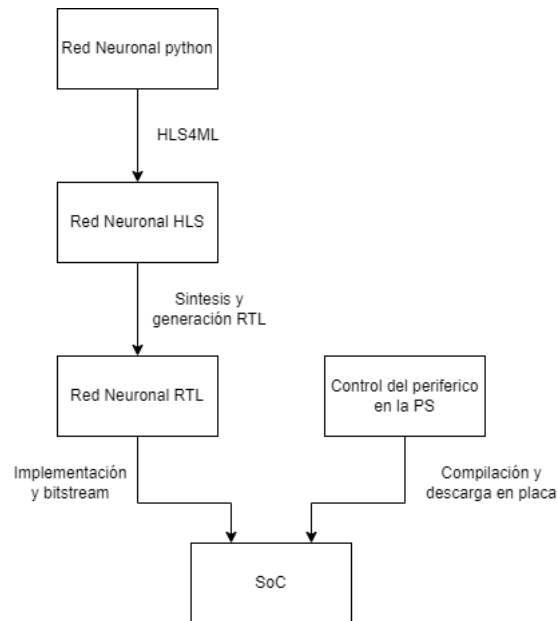


Figura 3.1: Esquema de desarrollo de la aplicación.

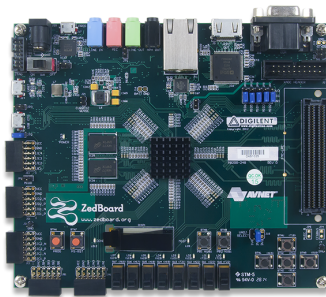


Figura 3.2: Placa de desarrollo Zedboard de Digilent.[4].

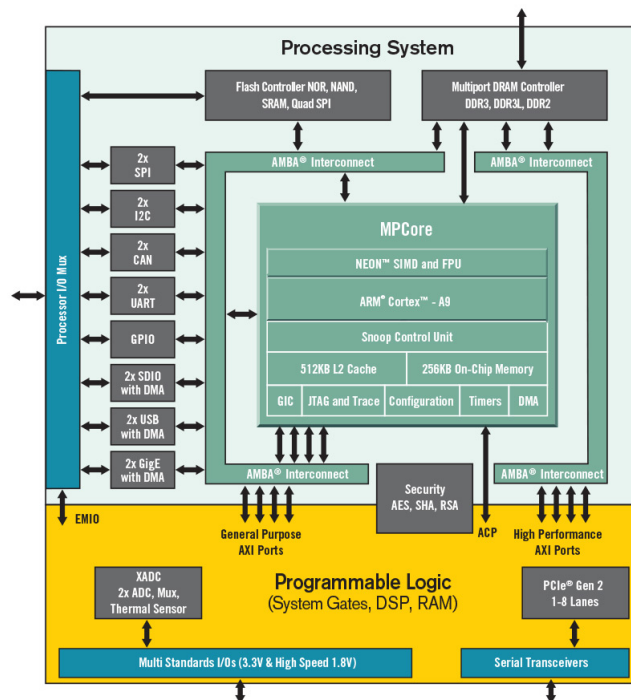
## 3.1 Red neuronal básica

Para comenzar con la implementación de redes neuronales sobre la *FPGA* se partirá desde un escenario simple, en el que el número de parámetros no sea muy elevado pero la red neuronal consiga aprender y posteriormente se aumentará el número de capas y el tipo de estas. Para ello se empleará diseño de red neuronal orientado a los *Smart Meters* descrita en el capítulo 2.3 y las fases de implementación vistas en la figura 3.1.

### 3.1.1 Diseño de la red neuronal básica en *Keras*

Como se ha comentado previamente, esta solución tratará de resolver el problema de clasificación de cuatro electrodomésticos con el menor número de parámetros posibles. Para ello se utilizarán únicamente las 4 neuronas de clasificación que entregarán como resultado la probabilidad de pertenencia a cada electrodoméstico.

La entrada de la red neuronal serán las 160 muestras temporales de la señal de intensidad preprocesadas, por lo que, teniendo cuatro neuronas densamente conectadas se dispondrá de 644 parámetros en la red. Posteriormente, para la tarea de clasificación se le aplicará una función de activación de tipo *Soft-Max*

Figura 3.3: Estructura de la *Zynq*.

Layer(type)	Output Shape	Param
input_1 (Input Layer)	[160,1]	0
flatten (Flatten)	[160]	0
dense (Dense)	[4]	644
Total params: 644		
Trainable params: 644		
Non-trainable params: 0		

Tabla 3.1: Resumen de la red neurona básica.

para conseguir el porcentaje de pertenencia a la clase. Estas interconexiones se observan en la topología de la red de la figura 3.4 y en el resumen de los parámetros de la tabla 4.7, obtenida utilizando el método *summary* del modelo creado con *Keras*. Por otra parte, la capa *flatten* es prescindible en el modelo simplemente definiendo la entrada como un vector de 160 en vez de una matriz de 160x1, pero dado que su introducción no afecta a los recursos consumidos en la implementación no se ha eliminado.

Posteriormente, la red neuronal es entrenada utilizando los mismos hiperparámetros que para el modelo original. El resultado es una precisión de entrenamiento cercana al 80 % a partir de la época 15 y al 77 % en validación como se observa en las dos primeras gráficas de la figura 4.13. Se observa además que a partir de la época 15 se empieza a producir sobreentrenamiento (*overfitting*) dado que la precisión de aprendizaje sigue aumentando pero la de validación permanece constante. A continuación, se evalúa la mejor red neuronal con los datos de test y se obtiene una precisión del 79.70 % con la correspondiente matriz de confusión que se representa en la última gráfica de la figura 4.13. En la matriz de confusión se observa la producción de falsos positivos destaca entre las dos primeras clases.

Aunque este modelo tiene una precisión bastante mejorable, es bastante sencillo y fácil de implementar, por lo que servirá de utilidad para evaluar el funcionamiento de la herramienta de exportación a HLS que se verá en la siguiente subsección.

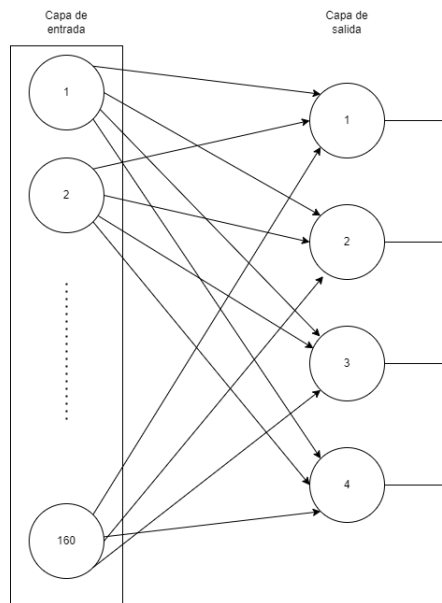


Figura 3.4: Topología red neuronal básica.

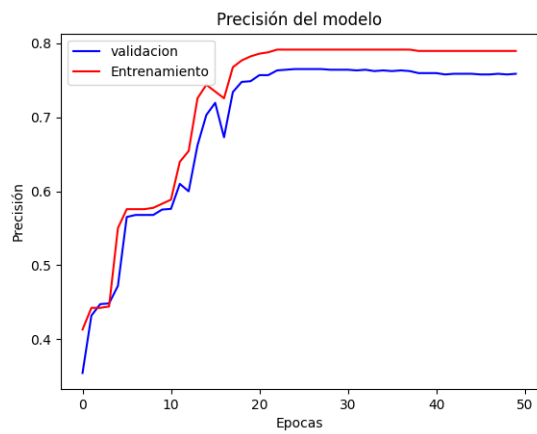
### 3.1.2 Conversión de una red neuronal a HLS

Dado que las FPGAs no emplean la notación en coma flotante ya que gastarían muchos recursos disponibles, se emplea la coma fija. La utilización de esta forma de representación permite utilizar números decimales con tamaños de palabra arbitrarios y menores a los estándares de 32 bits de coma flotante. Por lo tanto, un primer paso para implementar la red neuronal en la *FPGA* es obtener la coma fija a partir de la coma flotante, y un segundo paso es traducir el código de *Keras* que implementa la predicción de la red neuronal con su estructura de capas a un lenguaje de descripción *hardware* como *VHDL* o *Verilog*. Aunque también se puede convertir en primer lugar a lenguaje de alto nivel como *C* o *C++* y que posteriormente sea traducido a *RTL* mediante alguna herramienta como el sintetizador de *Vivado HLS*. Esta última opción será la que se utilizará más adelante pero primero habrá que obtener el proyecto *HLS*.

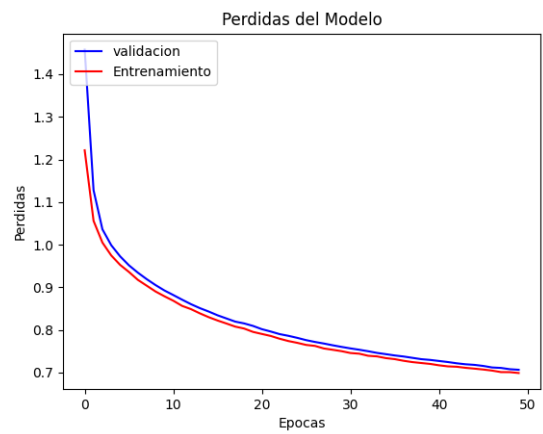
Para ello, se empleará el paquete de *HLS4ML* compatible con *Vivado* que permitirá tanto el paso de coma flotante a fija, como crear el proyecto *HLS* que utilizará *Vivado* por debajo. Por lo tanto, el primer paso para su obtención es la importación del paquete de trabajo como se observa en el código 3.1 y en el que además se incluye una librería para realizar gráficas de manera sencilla y la del sistema operativo para obtener la ruta del programa *Vivado 2019.2*. Posteriormente en el código se creará la configuración correspondiente mediante el comando `hls4ml.converters.convert_from_keras_model`, en el que se introducirá en primer lugar el modelo y posteriormente se fijará la granularidad. Este es un parámetro importante ya que define a que nivel se harán las optimizaciones del proyecto. Tiene tres valores válidos que son: 'model' que permitirá optimizaciones a nivel de modelo, 'type' que las hará para cada tipo de capa, y 'name' que permitirá realizarlas por nombre de capas, siendo esta última la que más grado de detalle alcanza. El segundo parámetro es el `default_reuse_factor` que define el número de veces que volverá a utilizar el mismo recurso. Esto se observa claramente en la figura 3.6 en la que a medida que se aumenta el parámetro menos multiplicadores se emplean, pero a su vez aumenta la latencia obtenida.

Finalmente el último parámetro de configuración es el tamaño de coma fija que empleará el *datapath* del modelo, que en este caso se deja por defecto a 16 bits en los que 10 bits estarán destinados a la

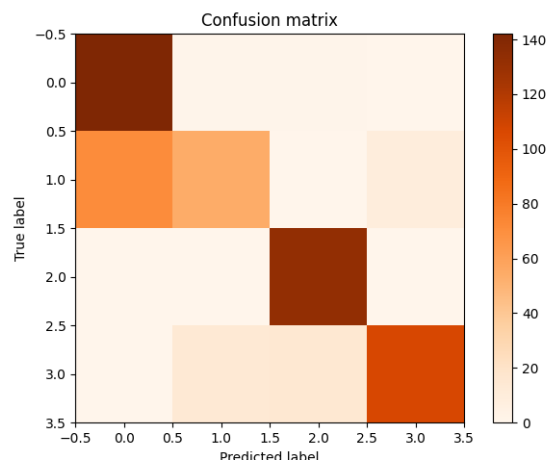




(a) Precisión del modelo básico.



(b) Pérdidas del modelo básico.



(c) Matriz de confusión del modelo básico.

Figura 3.5: Resultados de entrenamiento de la red neuronal básica.

parte fraccionaria y 6 a la parte entera. Cabe destacar que este apartado es importante de cara a que la precisión del modelo sea similar a la obtenida de manera teórica como se observará en futuros proyectos.

```
import hls4ml
import plotting
import os
os.environ['PATH'] = '/opt/Xilinx/Vivado/2019.2/bin:' + os.environ['PATH']

config = hls4ml.utils.config_from_keras_model(model, granularity='model',
                                              default_reuse_factor=4,
                                              default_precision = 'ap_fixed<16,6>')

print("-----")
print(" Configuration ")
plotting.print_dict(config)
print("-----")
hls_model = hls4ml.converters.convert_from_keras_model(model,
                                                       hls_config=config,
                                                       output_dir='model_1/hls4ml_prj',
                                                       part='XC7Z020-CLG484-1')
```

Código 3.1: Configuración del modelo.

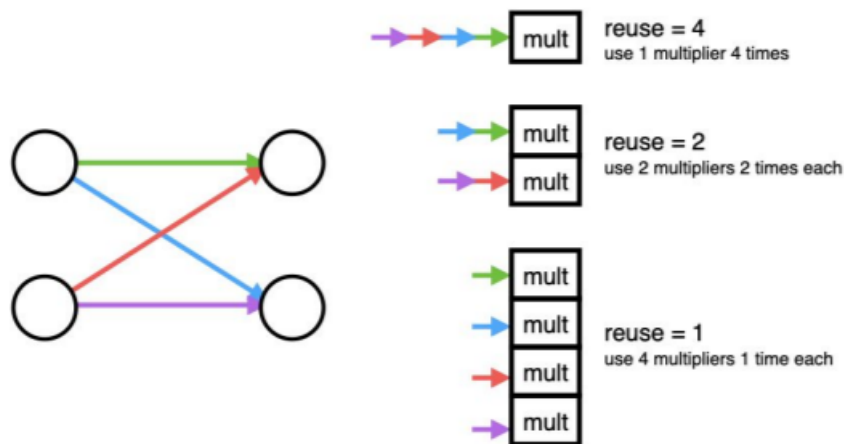


Figura 3.6: Significado del Reuse factor [5].

El resumen de las distintas capas generadas previamente convirtiendo el modelo de *keras* en el de *hls4ml* se observa en la figura 3.7. El diagrama se muestra mediante la función *hls4ml.utils.plot\_model*. En ella se puede comprobar que tanto el número de capas como de parámetros es correcto y el tipo de dato que utilizan, que como se ha comentado previamente, es coma fija de 16 bits y 6 de parte entera.

Tras comprobar que el diseño de la red en *hls4ml* es correcto se procede a generar el proyecto de *Vivado HLS* que contendrá la red neuronal. Para ello se hará uso de la función *hls\_model.compile()*. Cabe destacar que esta función únicamente permite ser utilizada en una máquina *Linux*, por lo que mientras que todo lo demás se ha desarrollado en *Windows*, será necesario importarlo al otro sistema operativo.

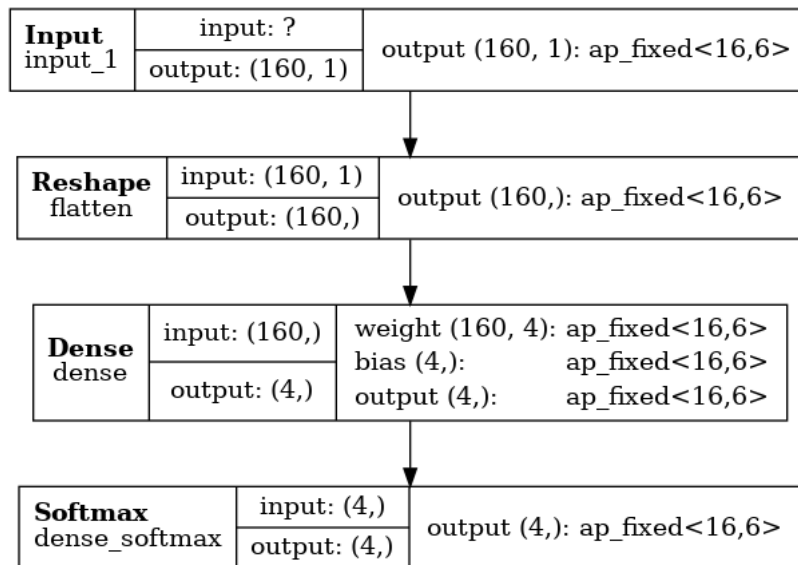


Figura 3.7: Diagrama implementado en la red básica.

Debido a problemas con la creación del entorno, se ha procedido a realizar este procesado en la nube con un entorno de *Jupyter Hub*[15]. Este sistema permite acceder a recursos computacionales sin necesidad de disponer del *hardware* y sirve al usuario de un entorno preconfigurado para el *Data Science*. Además, es la herramienta recomendada por *hls4ml* para seguir sus tutoriales y dispone de *Vivado 2019.2* preinstalado que se empleará como *backend* de la aplicación.

Por lo tanto, al ejecutar la compilación del sistema se generará el proyecto en *Vivado HLS*, aún así se continuará empleando *python* para verificar el comportamiento del sistema. Para ello, mediante el método `hls_model.predict()` se calcula con la base de datos de test las soluciones del sistema. En este caso se obtiene una precisión del 79.70%, igual a la obtenida de manera teórica. Por lo tanto, se puede concluir que la cuantificación es correcta para este sistema.

A continuación, se atiende al tema de conexión con el *IP* que se generaría si se sintetizara y se exportara el sistema. Por defecto, *HLS4ML* genera una interfaz paralelo con un puerto `ap_vld` para la validación de los puertos de entrada mediante las siguientes sentencias:

```
#pragma HLS ARRAY_RESHAPE variable=input_1 complete dim=0
#pragma HLS ARRAY_PARTITION variable=layer4_out complete dim=0
#pragma HLS INTERFACE ap_vld port=input_1,layer4_out
#pragma HLS PIPELINE
```

Código 3.2: Directivas predeterminadas para I/O.

Dado que para la arquitectura deseada las distintas entradas de la base de datos de test se encuentran en el *PS*, resulta más sencillo disponer de una interfaz AXI para escribir los datos. Por lo tanto, manualmente se modifican las líneas previamente vistas del fichero `myproject.cpp` en el que se define la función por las siguientes:

```
#pragma HLS INTERFACE s_axilite port=input_1 bundle=myproject_io
#pragma HLS INTERFACE s_axilite port=layer4_out bundle=myproject_io
```

Código 3.3: Directivas para la interfaz AXI para I/O.

El único paso antes de continuar con *Vivado HLS* es la síntesis del sistema. Esto se realiza mediante la función `hls_model.build()` y tras ello se reproducirán los resultados, lo cual permitirá extraer una comparativa de los recursos empleados, la latencia y si se cumplen las restricciones temporales.

### 3.1.2.1 Análisis de recursos y latencia resultante

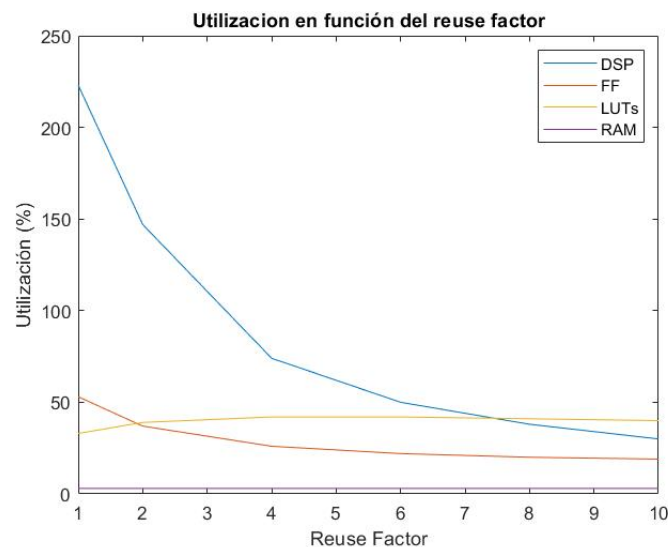
En función del *reuse factor* se puede modificar el consumo de recursos del sistema en la fase de síntesis. Cabe destacar que estos se pueden modificar en el futuro en la fase de la implementación en tarjeta debido a las optimizaciones que realiza el software Vivado, pero dan una idea de los recursos consumidos y la latencia del sistema

Con el *reuse factor* de 4 se obtiene la utilización mayor con la menor latencia por lo que será el seleccionado para las siguientes etapas. En concreto su utilización de recursos se muestra en la tabla 3.2.

Name	BRAM_18K	DSP48E	FF	LUT
Total	3	164	28420	22808
Available	280	220	106400	53200
Utilization(%)	1	74	26	42

Tabla 3.2: Utilización básica con *Reuse Factor* = 4.

La utilización de *BRAM\_18K* permanece constante a medida que se varía el parámetro de reutilización pero sin embargo los otros tres varían al modificar su valor como se observa en la figura 3.8. En ella se observa cómo el número de DSP cae abruptamente desde el máximo en el 223% para el *reuse factor* de 1 hasta menos del 30%. Los FF también disminuyen al aumentar el valor pero no es tan significativo. Sin embargo los LUTs sufren un ligero aumento al principio y se mantienen.

Figura 3.8: Utilización en función del *reuse factor* para la red neuronal básica.

Por otra parte la latencia en ciclos de reloj del sistema aumenta como se evidencia en la figura 3.9, en la que en primer lugar disminuye un poco pero a partir de ahí aumenta. Por lo tanto, se selecciona la configuración con un *reuse factor* de 4 ya que es el primero en convertirse sintetizable en cuanto al número de DSP y es el que tiene una latencia mínima de 18 ciclos de reloj.

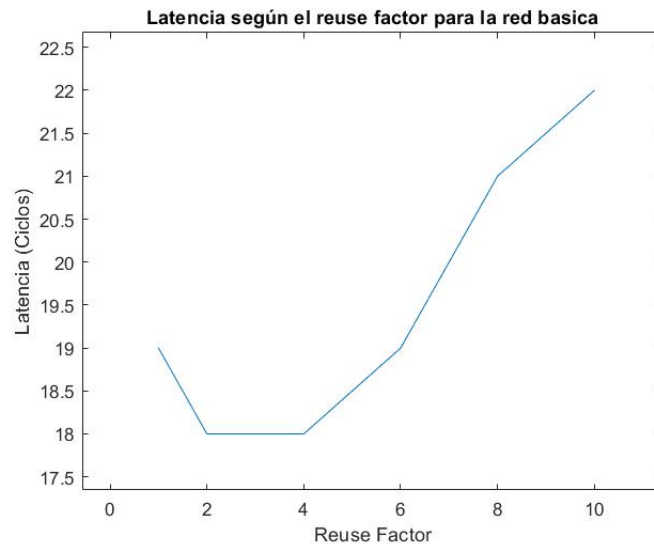


Figura 3.9: Latencia en función del *reuse factor* para la red neuronal básica.

### 3.1.3 Pruebas realizadas en HLS

Todas las pruebas previas se podían haber realizado desde el programa de *HLS*, pero se ha preferido expresar al máximo las funciones de *HLS4ML* ya que permiten gestionar el proyecto de Vivado desde alto nivel. Aún así, a modo de comprobación se prueba el sistema ya sintetizado en *Vivado HLS* para observar que el código es correcto. Para ello se ha generado un *testbench* en el que se le alimente a la red neuronal con las muestras de test. En primer lugar se ha almacenado en una variable las  $547 \times 160$  de la señal de entrada en formato de coma flotante que convertirá el compilador en *ap\_fixed<16,6>* y las 547 salidas. Estas se han introducido en la red y se han obtenido los mismos resultados obtenidos en la subsección anterior como se evidencia en la figura anterior 3.10.

```
Console | Tasks | Problems | Executables | Debugger Console
<terminated> (exit value: 0) myproject_prj.Debug [C/C++ Application] csim.exe
Numero de muestras de test: 547
El numero de errores es: 111 y de aciertos 436
Accuracy: 0.797075
```

Figura 3.10: Resultados en *Vivado HLS* de la Red Básica.

### 3.1.4 Diseño del sistema en Vivado

A partir de la síntesis en *HLS* se exporta el *RTL* para poder ser utilizado en *Vivado*. Con esto se generará un módulo *IP* con la configuración de entrada/salida de la figura 3.11. Se dispone de un puerto AXI desde el que se controlará la entrada y la salida de la red, los puertos de reloj y reset y finalmente una interfaz *ap\_ctrl*. Esta interfaz manejará el inicio de operación del módulo mediante su puerto de

entrada y sus tres puertos de salida. La señal *ap\_start* marcará el comienzo del cálculo siempre y cuando *ap\_ready* esté a nivel alto. Por otra parte, *ap\_idle* y *ap\_ready* indican cuando se está realizando las operaciones y cuando se han terminado respectivamente. Por otra parte, tiene otras cuatro salidas que indican el tamaño de los datos de entrada y salida y su validez, que no se emplearán.

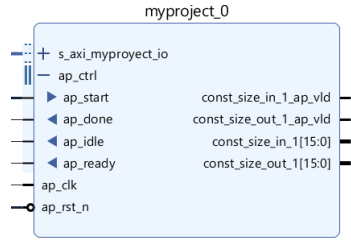


Figura 3.11: Configuración de puertos del IP básico.

Puesto que se empleará un *SoC* será necesario conectar el módulo con la *PS* como se observa en la figura 3.12, en la que el *AXI* de la red neuronal se conecta mediante un módulo *AXI interconnect* al igual que los cuatro puertos del *ap\_ctrl*, aunque estos necesitarán pasar previamente por un módulo *AXI GPIO*. Este estará configurado de manera que tenga 2 puertos, el primero de ellos de 1 bit de salida y el segundo de 3 bits de todo entradas, lo que permitirá controlar fácilmente estos puertos desde el procesador. Además, estos puertos llevan a un *ILA* para ser depurados en caso de observarse errores. Finalmente, el pin de reloj es compartido entre todos los módulos y es de 100 MHz. La señal de reset del sistema se creará mediante el *IP Processor System Reset*.

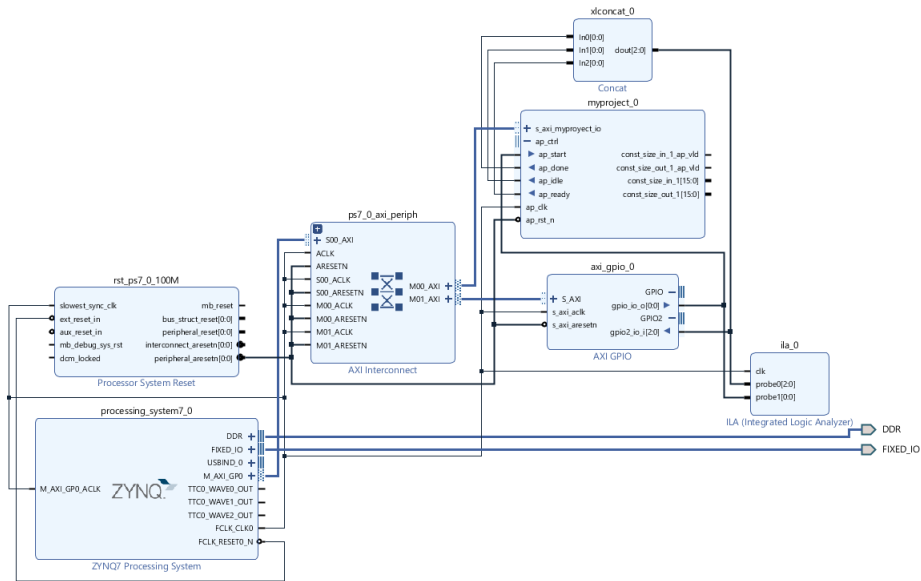


Figura 3.12: Esquema básico.

Posteriormente se crea el *bitstream* y se obtiene la utilización de recursos del sistema implementado de la tabla 3.3, en la que se observa que el consumo de recursos con respecto a *BRAM* y *DSP* es el mismo con respecto a la tabla 3.2, pero el número de *FF* y *LUTs* disminuye significativamente debido a las optimizaciones realizadas por *Vivado* para la síntesis e implementación en *FPGA*. Por otra parte, en esta figura no se encuentra únicamente el módulo *IP* diseñado sino que también habrá otros como el *ILA* o el *AXI Interconnect* que consumen recursos, pero su influencia es inferior al 5% de los recursos disponibles.

Resource	Total	Available	Utilization (%)
LUT	11775	53200	22.13
FF	15718	106400	14,77
BRAM	3.5	140	2.5
DSP	164	220	74.55

Tabla 3.3: Recursos de la red neuronal en Vivado.

Otro aspecto importante es la potencia consumida por la placa y la temperatura de operación. Vivado ofrece estos resultados como se observa en la figura 3.13. La potencia consumida por el apartado de *PL* está entorno al 15 % mientras que el de la *PS* es del 85 %, aunque este consumo se puede modificar en función de las tareas que realice este apartado.

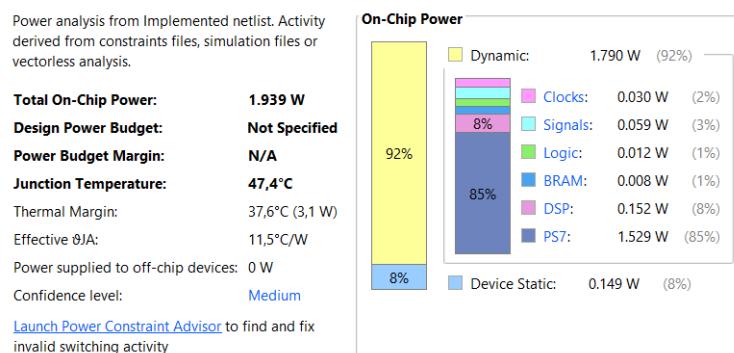


Figura 3.13: Potencia consumida por la implementación.

Finalmente hay que analizar el aspecto temporal. Se realizará un barrido desde los 100 MHz hasta que se produzcan errores para observar la máxima frecuencia de reloj que puede soportar el sistema con la implementación realizada. Los resultados para 100 MHz se observan en la figura 3.14a y son positivos ya que se cumplen las restricciones. Para 110 MHz sucede igual como se percibe en la figura 3.14b aunque el margen es mucho menor y muy cercano a cero. Por lo tanto, no se podrán llevar grandes subidas de frecuencia. Esto se corrobora con la figura 3.14c, en la que el tiempo de *setup* falla para 125 MHz.

A la vista de estos resultados se extrae que la frecuencia máxima es de 110 MHz con la que se tendría una latencia mínima de 162ns. Sin embargo, se emplea una frecuencia de 100 MHz para futuros apartados para tener un margen temporal mayor con una latencia de 180ns.

### 3.1.5 Diseño del sistema en *Vitis*

Tras diseñar el apartado de *hardware* para la *PL* se exporta el *bitstream* y se comienza con la fase de diseño del *software* en la *PS*. Esta se realizará en el programa *Vitis* de *Xilinx*, en primer lugar se importará la plataforma de *hardware* previamente creada y a continuación se procederá con la aplicación. Esta realizará un trabajo similar a las pruebas realizadas en *HLS*, se les introducirá las 547 muestras de test y se extraerán los resultados. Para extraer los resultados desde la placa se leerán desde la UART.

En primer lugar, se almacenará la matriz  $547 \times 160$  como unidimensional con todos los datos que se utilizarán como entradas en formato de coma fija con 16 bits y 10 de parte fraccional y las 547 salidas.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,467 ns	Worst Hold Slack (WHS): 0,028 ns	Worst Pulse Width Slack (WPWS): 3,750 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 38666	Total Number of Endpoints: 38650	Total Number of Endpoints: 16111

**All user specified timing constraints are met.**

(a) Análisis temporal para el esquema básico a 100MHz.

#### Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,142 ns	Worst Hold Slack (WHS): 0,021 ns	Worst Pulse Width Slack (WPWS): 3,250 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 38666	Total Number of Endpoints: 38650	Total Number of Endpoints: 16111

**All user specified timing constraints are met.**

(b) Análisis temporal para el esquema basico a 110MHz.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0,087 ns	Worst Hold Slack (WHS): 0,019 ns	Worst Pulse Width Slack (WPWS): 2,750 ns
Total Negative Slack (TNS): -0,087 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 1	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 38804	Total Number of Endpoints: 38788	Total Number of Endpoints: 16180

**Timing constraints are not met.**

(c) Análisis temporal para el esquema básico a 125MHz.

Figura 3.14: Análisis temporal del esquema básico para distintas frecuencias.

Estas serán constantes del sistema y estarán almacenadas en ficheros *.h*. En el *main* en primer lugar se inicializará la plataforma. Después habrá que arrancar el periférico correspondiente a la red neuronal. Para ello, cuando se generó el *IP* en *Vivado HLS*, se creó automáticamente un *driver* con funciones que permiten utilizar el periférico. Para la funcionalidad previamente comentada se empleará la función *int XMyproject\_Initialize(XMyproject \*InstancePtr, u16 DeviceId)*; en la que se introducirá la estructura de tipo *XMyproject* y el *ID* del dispositivo. A continuación, hay que hacer lo mismo con los *XGPIO* que permiten controlar el comienzo y la lectura del periférico con la función *int XGpio\_Initialize(XGpio \*InstancePtr, u16 DeviceId)* y fijar la dirección de sus puertos con *void XGpio\_SetDataDirection(XGpio \*InstancePtr, unsigned Channel, u32 DirectionMask)*, en los que el canal 1 será de salida y el 2 de entrada. Por lo que, en el canal 1 se fijará el bit *ap\_start* y en el canal 2 se leerán los bits del resto de puertos en la interfaz.

Tras esa primera fase de inicialización se procederá con la fase de pruebas. En primer lugar se leerán los 160 números correspondientes a cada fila de la base de datos. A continuación, hay que destacar que el periférico tiene 2 modos de lectura y escritura, en modo palabra (4 bytes) y en modo byte. Por velocidad se empleará el modo palabra, por lo que, dado que las variables están empleando 16 bits, habrá que agrupar 2 variables en una mediante el siguiente código para los datos de entrada:

```
for (i=0; i < 160/2; i++)
{
    x_final[i] = x_row[2*i] + (x_row[2*i+1] << 16);
}
```

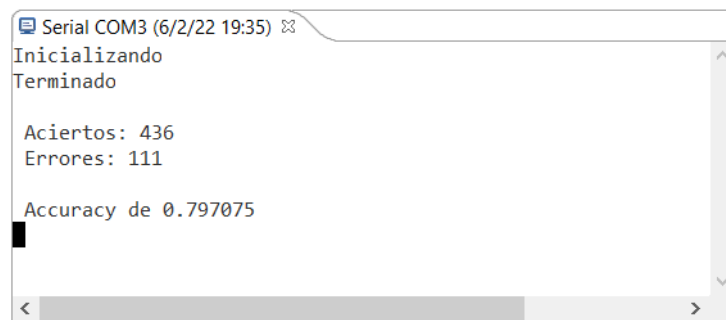


Código 3.4: Agrupación de dos muestras de 16 bits en una sola palabra de 32 bits.

Para los datos de salida la operación es la contraria, hay que seleccionar únicamente los primeros 16 bits o los siguientes. A continuación, mediante la función del driver `u32 XMyproject_Write_input_2_V_Words(XMyproject *InstancePtr, int offset, int *data, int length)` se escribirán los datos en el periférico. Para ello, en los datos se introduce la matriz de entrada colocada en formato palabra y se fijará la longitud en 80. Tras terminar la operación se fijará el `GPIO` correspondiente al `ap_start` a nivel alto y acto seguido a nivel bajo, dando comienzo al procesado.

En este caso se opta por una espera activa por facilidad, quedando el programa en un bucle `while` hasta que el periférico esté en estado de `done`. Posteriormente se leen los resultados, se selecciona el que mayor probabilidad tenga como solución del sistema y se almacena en una matriz. Este proceso se repite 547 veces y tras completarse se procede a calcular el número de aciertos y errores ocurridos y la precisión. Cuando dispone de esos datos se los trasmite por la `UART` al ordenador para observar los resultados obtenidos.

Por lo tanto, al ejecutar el código se observará en un terminal unos resultados iguales a los de la figura 3.15 que son exactamente iguales a los obtenidos de manera teórica y en HLS.



```
Serial COM3 (6/2/22 19:35) ✕
Iniciando
Terminado

Aciertos: 436
Errores: 111

Accuracy de 0.797075
```

Figura 3.15: Resultados de la red neuronal básica implementada en la tarjeta.

Por otra parte, si mediante el `ILA` se observa los puertos de `ap_ctrl` para una iteración se obtendrán los resultados de la figura 3.16, en los que se observa que la latencia de reloj es aproximadamente los 80 ciclos de reloj que se calcularon teóricamente en HLS.

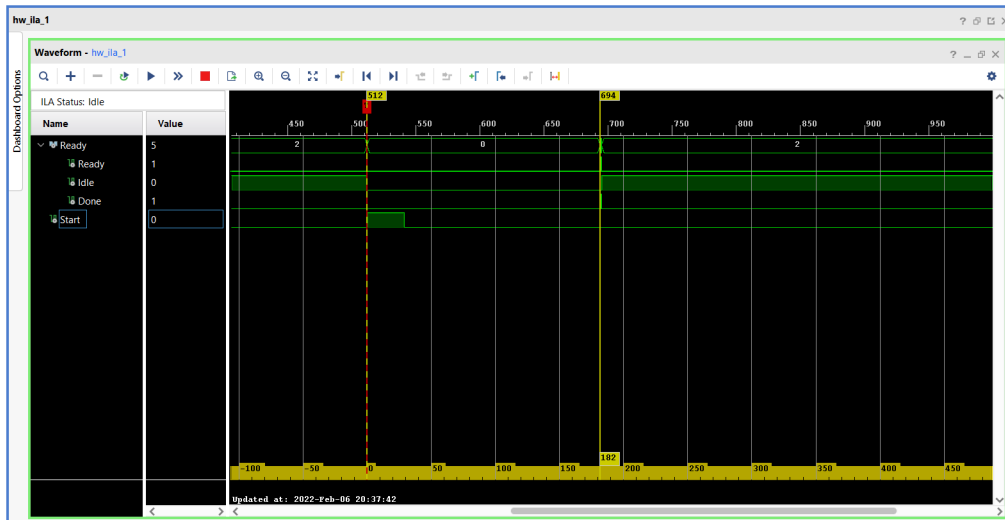


Figura 3.16: Resultados en ILA de la red neuronal básica implementada en la tarjeta.

## 3.2 Red neuronal de dos capas densamente conectadas

La sección anterior permitía visualizar un ejemplo sencillo de red neuronal que llegaba a aprender el comportamiento del sistema pero su falta de parámetros no permitía que estos progresaran. En este capítulo se analizará una red neuronal más profunda con dos capas y un número mayor de neuronas densamente conectadas permitiendo un mejor nivel de aprendizaje. Las fases de diseño en este caso serán las mismas que el anterior. Se partirá del diseño en *Keras* de la red y posteriormente se bajará de niveles, primero convirtiendo el proyecto en *HLS*, posteriormente en *RTL* y finalmente uniéndolo con el *processing system* del *SoC*.

### 3.2.1 Diseño de la red neuronal de dos capas en Keras

El modelo desarrollado en este apartado tendrá las mismas 160 entradas que en el apartado anterior, pero verá aumentado el número de neuronas. Se ha apostado por hacer dos capas densamente conectadas. La primera tendrá 8 neuronas con una función de activación *Relu* mientras que la segunda serán 4 neuronas con función de activación *Soft-max* iguales que en la sección anterior. La topología de la red neuronal se observa en la figura 3.17.

Por lo tanto, en este caso el número de parámetros entrenables para la primera capa serán 1288 y en la segunda 36 como se observa en la tabla 3.4. Para este modelo se han duplicado el número de parámetros con respecto al caso anterior.

Layer(type)	Output Shape	Param
input_1 (Input Layer)	[160,1]	0
dense (Dense)	[8]	1288
dense_1 (Dense)	[4]	36
Total params: 1324		
Trainable params: 1324		
Non-trainable params: 0		

Tabla 3.4: Resumen de la red neurona de dos capas densamente conectadas.

Posteriormente se procede al entrenamiento de la red neuronal empleando los mismos hiperparámetros del modelo original y se consigue una precisión cercana al 90 % para el entrenamiento y la validación a partir

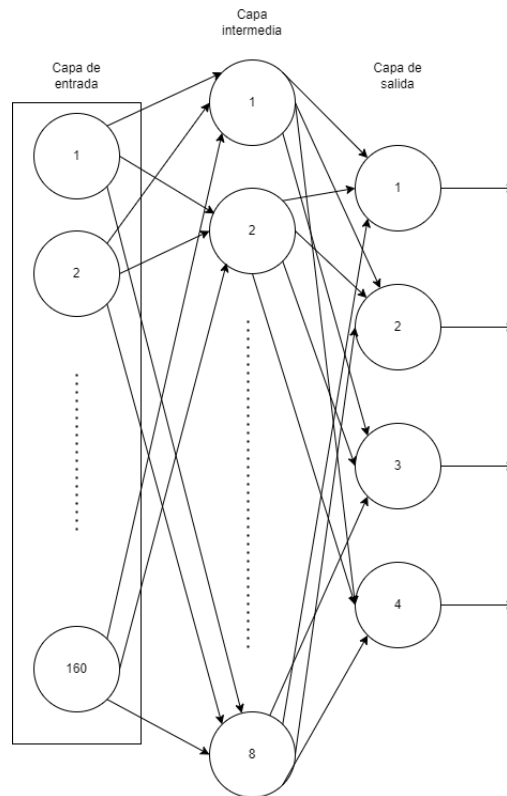
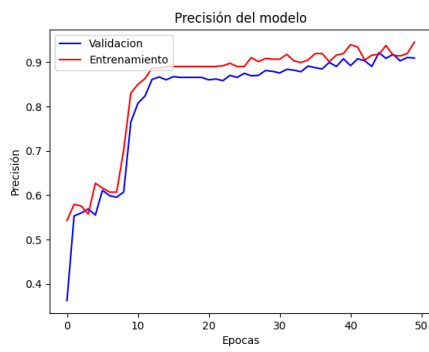
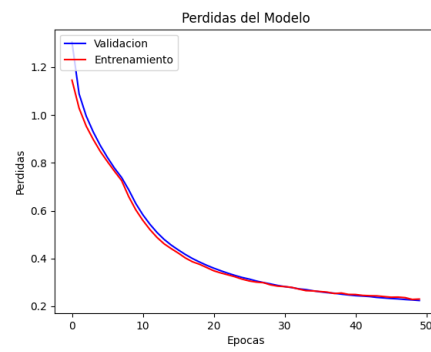


Figura 3.17: Topología de la red neuronal de dos capas densas.

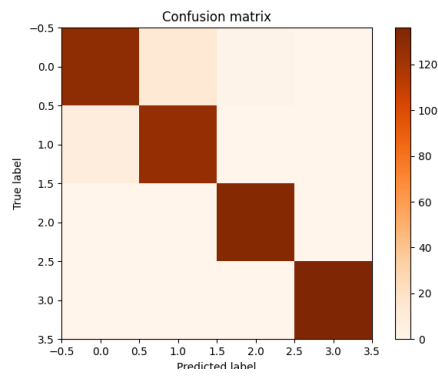
de la época 15 como se observa en la figura 3.18a. Sin embargo, para test se obtiene una precisión del 95.79% y una matriz de confusión en la que la mayoría de errores se producen entre el primer y segundo electrodoméstico como se visualiza en la figura 3.18c. Por lo tanto, este modelo es bastante positivo y se llevará a la implementación en *FPGA*.



(a) Precisión del modelo de dos capas densas.



(b) Pérdidas del modelo de dos capas densas.



(c) Matriz de confusión del modelo de dos capas densas.

Figura 3.18: Resultados de entrenamiento de la red neuronal de dos capas densas.

### 3.2.2 Conversión de la red de dos capas densas a HLS

A continuación se procederá a convertir el modelo de *Keras* en un modelo de *HLS* siguiendo los mismos pasos que para la red anterior. Se tomarán las mismas variables con un *reuse\_factor* inicial de 1 y una cuantificación de 16 bits con 6 bits de parte entera. Con ello se obtiene la arquitectura del modelo de la figura 3.19 con los tamaños de cada variable de la red.

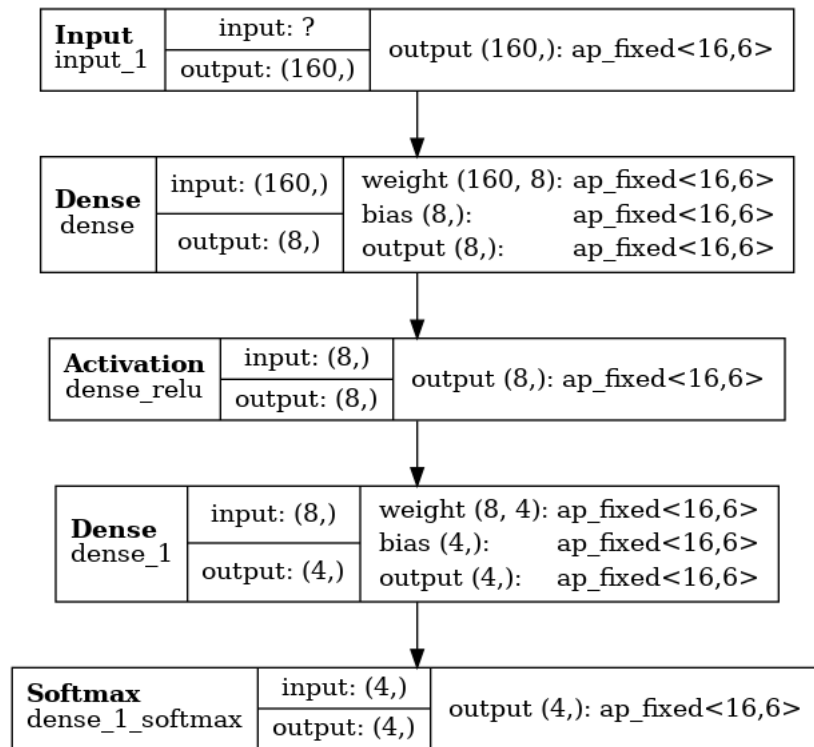


Figura 3.19: Diagrama del modelo HLS con dos capas densamente conectadas.

Se sigue avanzando los pasos convirtiendo el diseño en un proyecto de *Vivado HLS* y se prueba la precisión de la predicción con las muestras de test. Al realizar este proceso se observa que hay un fallo en la cuantificación ya que mientras la precisión teórica era de 95% con el modelo HLS se obtiene un 80%.

Este error puede ser debido a múltiples factores como cuantificación, redondeo o desbordamiento. Para ello en la siguiente sección se llevará a cabo un estudio a nivel de modelo para disminuir el error cometido entre el modelo teórico y el que se va a implementar.

#### 3.2.2.1 Estudio de la cuantificación en el modelo de dos capas densas

Como se ha comentado anteriormente, hay un problema con la cuantificación del sistema. Para solucionarlo, *hls4ml* ofrece múltiples herramientas. En primer lugar, se observará la cuantificación necesaria para implementar cada uno de los parámetros entrenados necesarios. Para ello se empleará la función `hls4ml.model.profiling.numerical(model=model, hls_model=hls_model)` que mostrará la cuantificación necesaria para cada uno de los parámetros y la que se está garantizando con la configuración seleccionada de manera gráfica como se observa en la figura 3.20. Se aprecia que la cuantificación de coma fija con 16 bits y 10 de parte decimal son correctos a excepción de los pesos de la primera capa densa, que necesitan de hasta 16 bits de parte decimal para estar correctamente representados. Por lo tanto, el primer paso a adoptar es cambiar el tipo de optimización a nivel de capa y fijar el tamaño de bits de los pesos de la capa densa

para que cumplan las restricciones manualmente con `config['LayerName']['dense']['Precision']['weight'] = 'ap_fixed<20,2>'`. Con ello se obtiene un aumento de la precisión desde el 80 % al 83.72 %, lo cual es significativo pero sigue sin igualar las prestaciones iniciales de la red de diseño.

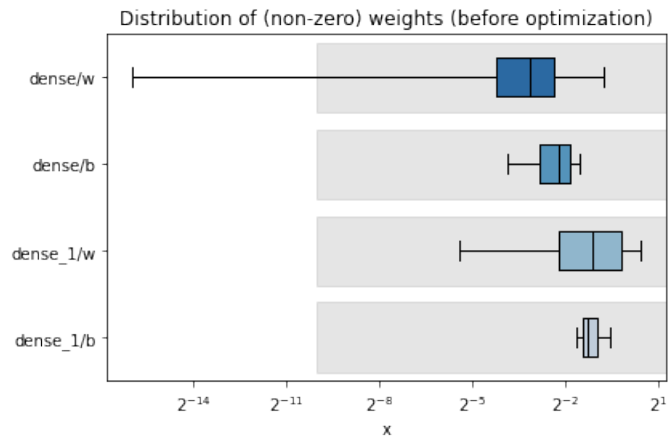


Figura 3.20: Cuantificación de los parámetros en la red neuronal de dos capas.

Para solucionarlo se adopta una estrategia distinta en la que se realizan optimizaciones al modelo completo barriendo la utilización de bits desde 16 hasta 32 bits con distintos repartos en el tamaño de las palabras. En concreto, se subirá desde 0 bits de parte entera hasta el casi el número de bits totales. Como figura de mérito se utilizará la precisión obtenida para cada combinación de número de bits y parte entera asignada. La respuesta obtenida se evidencia en la figura 3.22, en la que se observa que cuando la parte entera es menor a 4 el error es máximo y que a partir de ese valor crece abruptamente hasta llegar al máximo con 5 bits. Esto se debe a que antes de 4 bits, no había resolución suficiente para todas las representaciones y en 5 bits se alcanza la resolución de la parte entera óptima. A partir de este valor, la solución empeora dado que cada vez se dedica menos bits a la parte fraccionaria y los añadidos a la parte entera no mejoran la respuesta.

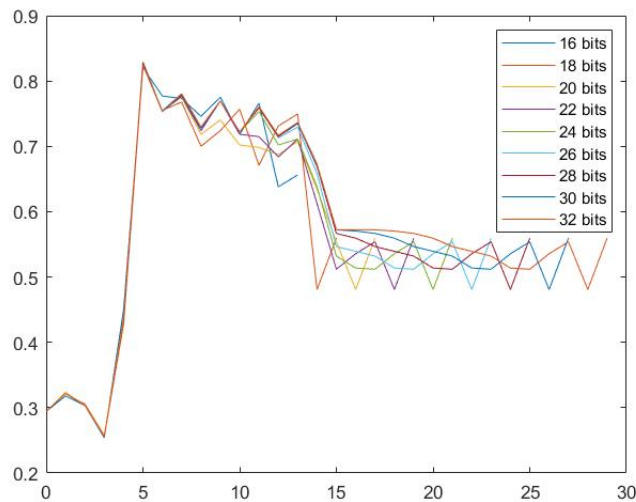


Figura 3.21: Respuesta a la variación de la cuantificación en la estructura de dos capas variando la coma fija.

Por otra parte, esta respuesta es muy similar aunque se aumente el número de bits, en todos los casos se alcanzará un valor de 83 % como máximo a los 5 bits de coma fija. Para comprobar esa poca relación entre el número de bits y la mejora de rendimiento se realiza un segundo barrido. En él se fija el tamaño de la parte entera en 5 bits y se aumenta el tamaño de la palabra desde 10 hasta 64 obteniendo un máximo con un tamaño de 12 bits de casi el 85 %. A partir de ahí sufre fluctuaciones alrededor del 82.5 % y a partir de los 24 bits se fija constante en ese valor.

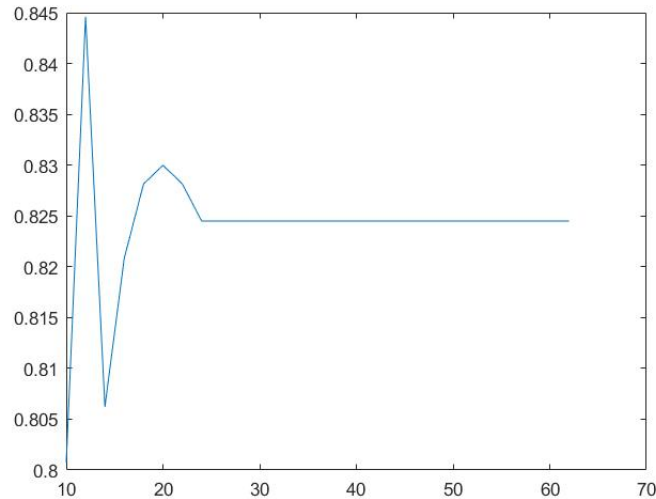


Figura 3.22: Respuesta a la variación de la cuantificación en la estructura de dos capas.

Dado que la resolución a nivel global parece estar bien, se debe analizar más en concreto la capa sobre la que sucede la falta de cuantificación. En concreto, analizando las respuestas de la red se ha observado que a la salida de la función de activación *Soft-max* hay números negativos, lo cual desde el punto de vista teórico es incorrecto. Por lo tanto, de cara a un diseño e implementación final es necesario hacer una optimización de todas las capas hasta obtener unos resultados cercanos a los obtenidos desde el punto de vista teórico.

Finalmente, dado que este trabajo se basa en el análisis de las herramientas para implementar algoritmos de *Deep Learning* en sistemas empotrados se toma para la implementación en el sistema tamaños que no son óptimos pero que facilitan el uso del sistema. En primer lugar se selecciona el tamaño de palabra de 16 bits para que la interfaz AXI escriba 2 números por ciclo agilizando las comunicaciones y facilitando el paso de datos desde la *PS* a la *PL*. Además, permite que el *testbench* de la subsección anterior sean reutilizables para esta. Por otra parte, se asignan 7 bits a la parte entera ya que para este valor se alcanza el máximo de la función de 82.44 % para las muestras de test.

### 3.2.2.2 Análisis de recursos y latencia resultante

De manera análoga al apartado de las cuatro neuronas de clasificación, será necesario realizar un análisis de los recursos que consumirá la red neuronal dentro de la lógica programable. Para ello, se recuerda que el parámetro de peso para modificar la latencia y el consumo de recursos es el *reuse factor*.

Si se sintetiza el proyecto de *HLS* para un *reuse factor* de 1 se obtienen los datos de la tabla 3.5. En ellos se observa que la ocupación del sistema es muy elevado, consumiendo gran cantidad de los *LUTs* disponibles de la placa y excediendo los *DSP* disponibles. Sin embargo, para esta estructura se obtiene

una latencia de 12 ciclos de reloj, lo que empleando un reloj de 100 MHz serían 120 ns. Por lo que, para otra *FPGA* con un mayor número de recursos sería una implementación muy buena en términos de latencia.

Resource	Total	Available	Utilization (%)
LUT	43204	53200	81
FF	25450	106400	23
BRAM	3	280	1
DSP	821	220	373

Tabla 3.5: Recursos de NN en Vivado para estructura de dos capas con *Reuse factor* de 1.

A pesar de estos datos, los recursos no son suficientes por lo que es necesario aumentar el *Reuse factor* para conseguir implementarlo. Para ello se realizan variaciones desde 1 hasta 16, obteniendo los resultados de la gráfica 3.23, en la que se observa que el consumo de *FF* y *BRAM* permanece prácticamente constante mientras que el número de *DSP* disminuye abruptamente siendo implementable con un reuse factor de 8. Por otra parte, los *LUTs* tienen un comportamiento opuesto, cuanto mayor es el factor de reutilización, más recursos consumen para implementar el control de uso de los *DSP*.

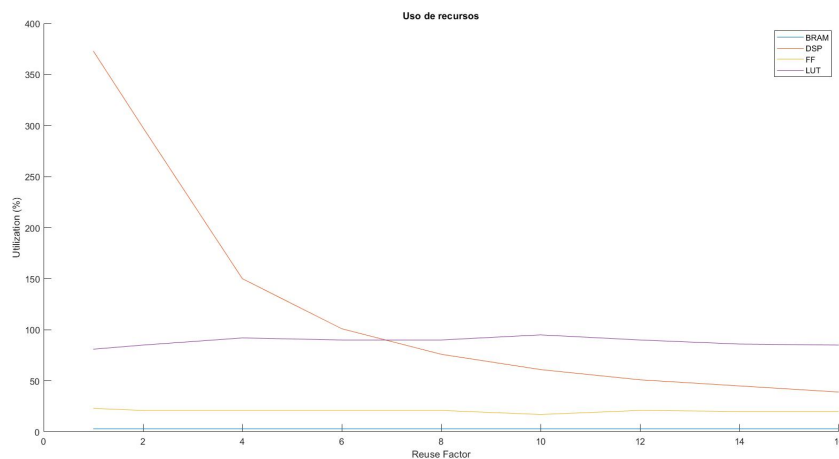


Figura 3.23: Recursos consumidos en la fase de síntesis variando el *Reuse factor* en la estructura de dos capas.

Al aumentar el factor también aumenta la latencia como se observa en la figura 3.24 desde los 12 ciclos de reloj hasta los 32 de manera casi lineal. Por lo tanto, para la siguiente fase de diseño se utilizará un *reuse factor* de 10, ya que se considera que con uno de 8, al consumir cerca del 75 % de los *DSP*, puede complicar bastante el enrutado del dispositivo. Con este valor se obtiene una latencia de 22 ciclos de reloj y cerca del 60 % de los *DSP* consumidos.

Por otra parte, los resultados ofrecidos se analizan para un interfaz de E/S paralelo por sencillez, pero de cara a su implementación en el *SoC* no es eficaz para comunicarse con la *PS*. Por lo tanto, de manera análoga al apartado anterior se cambia la interfaz de comunicaciones a un *AXI Lite*. Esto aumenta el número de recursos observados en la figura 3.6 y la latencia hasta 183 ciclos de reloj pero el consumo de *DSP* permanece constante.



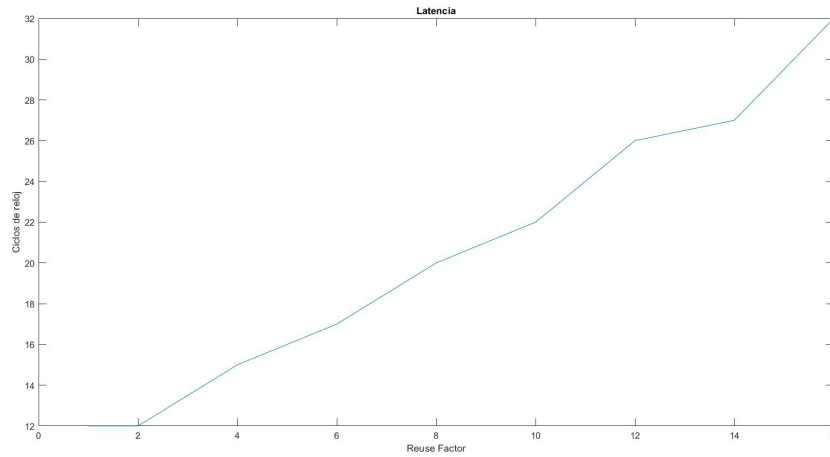


Figura 3.24: Latencia variando el *Reuse factor* en la estructura de dos capas.

Resource	Total	Available	Utilization (%)
LUT	50616	53200	95
FF	18411	106400	23
BRAM	6	280	2
DSP	136	220	62

Tabla 3.6: Recursos de NN en Vivado para estructura de dos capas con interfaz AXI.

### 3.2.3 Pruebas realizadas en HLS

Tras completar la conversión y síntesis del proyecto el sistema queda preparado para su desarrollo *RTL*. Para ello, en primer lugar se prueba el sistema en el programa *Vivado HLS*. El *testbench* empleado es exactamente igual, dado que las entradas al sistema son las mismas, únicamente cambia la distribución de bits en coma fija pero su conversión se realiza de manera automática por el simulador. Los resultados observados en la captura de la figura 3.25 tienen un ligero aumento en la precisión con respecto a los obtenidos en *python* con la cuantificación utilizada. Se asume que es debido a cómo se implementa el método *argmax* en python y al desarrollado en C++ mediante el siguiente código.

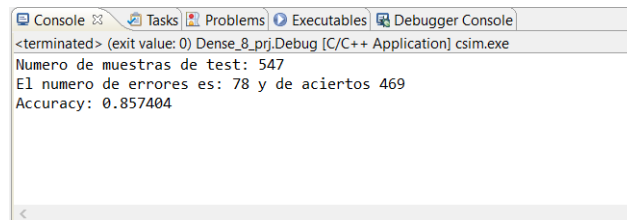
```
int argmax(result_t y[4])
{
    int i;
    int location=0;
    float maximo=0;
    float y_actual=0;

    for (i = 0; i < 4; i++)
    {
        y_actual=y[i].to_float();
        if (y_actual > maximo)
        {
            location = i;
            maximo = y_actual;
        }
    }
}
```

```

    }
}
return location+1;
}

```

Código 3.5: Código de la función *argmax*Figura 3.25: Resultados en *HLS* de estructura con dos capas.

Dado que este aumento mejora el resultado original sin modificar las probabilidades obtenidas a la salida de la red, los resultados se considerarán positivos y se continúa con la fase siguiente del diseño. Para ello se exportará el *RTL* de *HLS* a *Vivado*.

### 3.2.4 Implementación en Vivado

Al exportar el *RTL* se obtiene un bloque con las mismas entradas y salidas que en la red anterior como se observa en la figura 3.26.

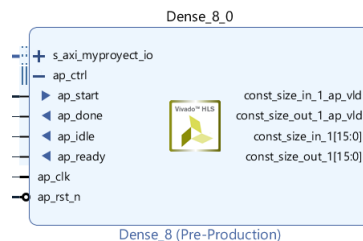


Figura 3.26: Bloque IP de la red de dos capas densamente conectadas.

La funcionalidad de estos puertos es exactamente la misma, por lo tanto se excluye su explicación en este apartado. Su conexionado con el resto de circuitos es exactamente igual al anterior como se observa en la figura 3.27 pero en este caso no se incluirá el *ILA* dado que los resultados obtenidos son iguales a los obtenidos en el apartado anterior y así se excluye su consumo de recursos en la implementación.

Considerando únicamente estos bloques, que harían que el sistema funcionará a la perfección se incluyen los resultados de la figura 3.7, en los que se observa que al igual que en el apartado anterior, se reducen desde el 95 % hasta el 38.67 %, siendo significativa esta reducción. El número de LUTs también se verá afectado, pero únicamente disminuirá un 5%. Por otra parte, el número de DSP y BRAM permanece constante.

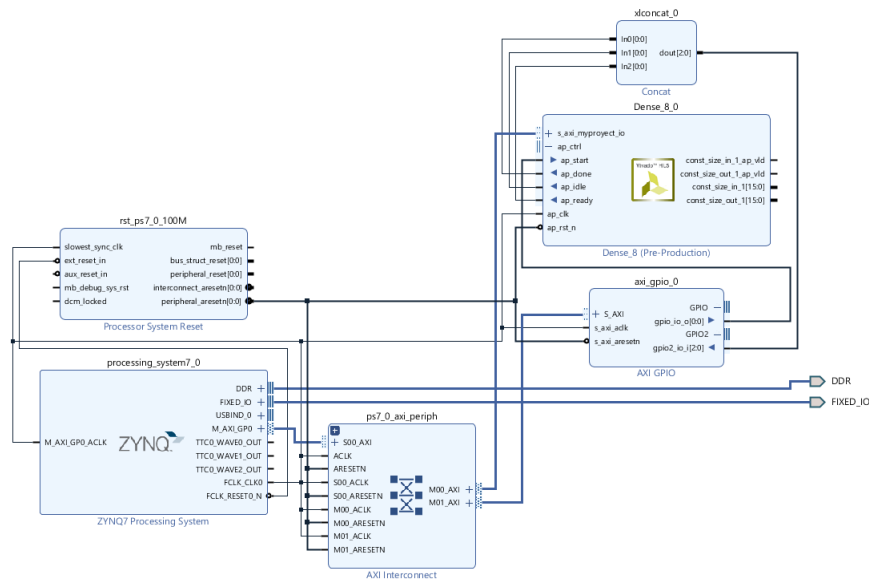


Figura 3.27: Esquema de conexionado de bloques en Vivado con dos capas.

Resource	Total	Available	Utilization (%)
LUT	20575	53200	38.67
FF	19157	106400	18
BRAM	3	280	2.14
DSP	136	220	61.82

Tabla 3.7: Recursos de NN en implementación del diagrama de dos capas.

El análisis de espacio de Silicio ocupado por el sistema ya se ha realizado pero aún faltan dos aspectos importantes por tratar, el consumo de potencia y el análisis temporal. En cuanto a este último se realiza un barrido desde los 50 MHz hasta los 100 MHz en saltos de 25 MHz. Se observa por tanto que para 50 MHz en la figura 3.28a se consiguen los objetivos temporales con facilidad. Para el caso de 75 MHz en 3.28b el margen se ve muy reducido pero todavía se cumplen tanto los tiempos de *setup* y *hold*. Finalmente en 3.28c el sistema falla en el tiempo de *setup* en gran cantidad de nodos. Por lo tanto, se seleccionará el reloj de 75 MHz para continuar con el diseño del sistema. Con él se obtendrá una latencia de 2.44  $\mu$ s.

Esta frecuencia de reloj se empleará para obtener un consumo de potencia acorde como el que se observa en la figura 3.29. En el consumo estimado total es de 1.992 W y estará a una temperatura de la unión de 48°C. En torno al 90% del consumo será dinámico y un 82% de él será producido por el *PS*. La red neuronal únicamente empleará 0.33 W de potencia del consumo dinámico para su aplicación.

### 3.2.5 Implementación en Vitis

Tras completar el bloque de *hardware* y fijar los parámetros de frecuencia de reloj y utilización de recursos se continua con el diseño de la *PS* en el programa de *Vitis* de *Xilinx*. El código del proyecto será muy similar al realizado para la red neuronal anterior ya que el tamaño de la entrada es el mismo y el tamaño de la palabra también.

El único cambio substancial se encuentra en el reparto de palabra que se almacenará como muestras de entrada. Dado que en el caso previo había 6 bits de parte entera y en este caso serán 7. Por lo tanto,

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 1,955 ns	Worst Hold Slack (WHS): 0,018 ns	Worst Pulse Width Slack (WPWS): 9,020 ns	
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns	
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	
Total Number of Endpoints: 39170	Total Number of Endpoints: 39170	Total Number of Endpoints: 19241	

All user specified timing constraints are met.

(a) Análisis temporal para el esquema de dos capas densamente conectadas a 50MHz.

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 0,515 ns	Worst Hold Slack (WHS): 0,030 ns	Worst Pulse Width Slack (WPWS): 5,519 ns	
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns	
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	
Total Number of Endpoints: 39170	Total Number of Endpoints: 39170	Total Number of Endpoints: 19241	

All user specified timing constraints are met.

(b) Análisis temporal para el esquema de dos capas densamente conectadas a 75MHz.

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): -2,097 ns	Worst Hold Slack (WHS): 0,023 ns	Worst Pulse Width Slack (WPWS): 4,020 ns	
Total Negative Slack (TNS): -7147,364 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns	
Number of Failing Endpoints: 6964	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	
Total Number of Endpoints: 39170	Total Number of Endpoints: 39170	Total Number of Endpoints: 19241	

Timing constraints are not met.

(c) Análisis temporal para el esquema de dos capas densamente conectadas a 100MHz.

Figura 3.28: Análisis temporal del esquema de dos capas densamente conectadas para distintas frecuencias.

empleando *Matlab* como herramienta auxiliar se convertirán todos los datos de test en formato coma fija con 16 bits y 7 de parte entera. Posteriormente se escribirán estos datos en un fichero *.h* y se leerán fila a fila.

Otra modificación es que los números a la salida de la red pueden ser negativos debido a los errores de cuantificación. Por lo tanto, es necesario adaptar la variable a la salida del periférico a coma fija con signo mediante el siguiente código:

```
for (i=0;i<2;i++)
{
    y_neta[2*i]=(float)(y_bruta[i] & 0x0000FFFF)/512;
    if (y_neta[2*i]>64)
        y_neta[2*i]=y_neta[2*i]-128;
    y_neta[2*i+1]=(float)((y_bruta[i] & 0xFFFF0000)>>16)/512;
    if (y_neta[2*i+1]>64)
        y_neta[2*i+1]=y_neta[2*i+1]-128;
}
```

Código 3.6: Código de la función conversión de dos palabras de 32 bits a 4 palabras de 16 bits en coma fija con signo.

Una vez realizados los cambios al código de la *PS*, este estará listo para la implementación en la *Zedboard*. Con ello se obtendrán los resultados de la figura 3.30 extraído de la *UART*. Con una precisión

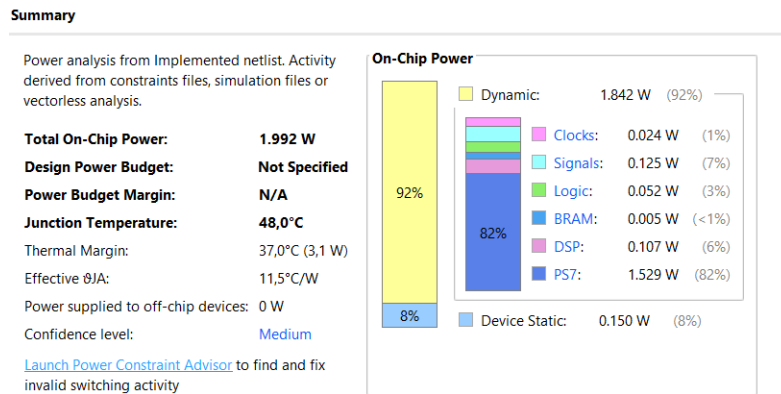


Figura 3.29: Potencia en la arquitectura de dos capas densamente conectadas.

del 85% al igual que en los resultados de *HLS* y superando los descritos al convertir la red neuronal suponiendo que por la definición de la función *argmax*.

```

Terminal
Serial COM3 (20/2/22 16:08)
Hello World
Terminado

Aciertos: 468
Errores: 79

Accuracy de 0.855576

```

Figura 3.30: Resultados de la red neuronal implementada en la *Zedboard*.

### 3.3 Red neuronal de dos capas utilizando CNN

En este último apartado se empleará una neurona más compleja denominada *convolutional neural network*, consistente en un filtro aplicado a la entrada para permitir extraer las características o *features* del problema. Es comúnmente utilizada para el tratamiento de imágenes, pero en este caso se va a aplicar a las muestras temporales de intensidad eléctrica del problema tratado en el TFM. Al igual que en los otros apartados se diseñará su funcionamiento teórico con *Keras* y se convertirá en *RTL* mediante *HLS4ML*.

#### 3.3.1 Diseño de la red neuronal convolucional en *Keras*

Este apartado es uno de los más complejos de vista a la implementación ya que desde el punto de vista teórico se puede aumentar de gran manera el número de neuronas pero la mayoría de estas no se podrán llevar a una *FPGA* debido al gran número de bucles que tiene que deshacer *Vivado HLS* y no únicamente a los recursos consumidos como pasaba en los casos anteriores.

Por ejemplo, se han diseñado redes con 100 neuronas convolucionales con un kernel de 15 que alcanzan una precisión de test del 99.26% en test o 40 neuronas con un *kernel* de 9 parametros, pero ambos no han sido implementables. A estas capas se las ha acompañado de un *Max-Pooling* para mejorar la respuesta y reducir el tamaño de la red, pero aún así no han terminado siendo implementables. Por lo tanto, se ha ido reduciendo el número de neuronas hasta llegar a una red sintetizable con 10 neuronas y un *kernel* de 20 y un *Max-Pooling* de 20 muestras. El diagrama de esta red neuronal se observa en la figura 3.31 con los parámetros previamente comentados.

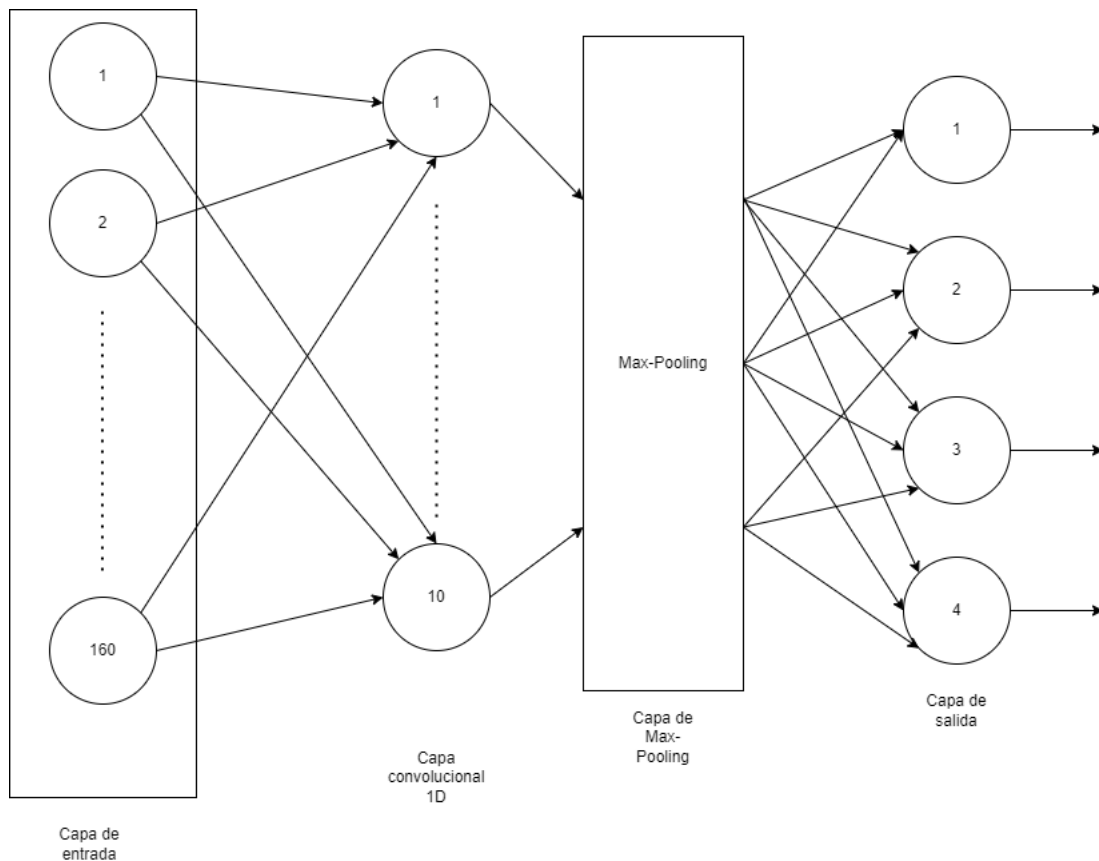


Figura 3.31: Diagrama red neuronal convolucional.

Esta red neuronal contará por lo tanto de 4 capas contando la de entrada y tendrá los parámetros de la tabla 3.8 en los que se añade una capa de *flatten* para adaptar el tamaño de la salida de la *MaxPooling1D* a la capa densa de salida. Cabe destacar que el número de parámetros es mucho menor a los dos casos anteriores debido a los problemas en implementación y a la ocupación de la placa. Esto es debido a la complejidad de la neurona que requiere de un factor de desenrollado mayor que no es capaz de soportar *Vivado HLS*. La red mínima que se ha conseguido sintetizar contaba con 20 neuronas con *kernel* de 13 y *max-pooling* de 4, teniendo un número de parámetros igual a 3204. Sin embargo, el consumo para la tarjeta de estudio era del 207% de los LUTs disponibles. Por lo tanto, para implementar esta red sería necesario tener un factor de reuso muy elevado y se ha llevado a la práctica la siguiente red de 494 parámetros.

Layer(type)	Output Shape	Param
input_1 (Input Layer)	[160,1]	0
conv1D (Conv1D)	[141,10]	210
max_pooling1d (MaxPooling1D)	[7,10]	0
flatten (Flatten)	[70]	0
dense_1 (Dense)	[4]	284
Total params: 494		
Trainable params: 494		
Non-trainable params: 0		

Tabla 3.8: Resumen de la red neurona convolucional.

La red se entrena con los mismos hiperparámetros del resto y se obtienen los resultados de la figura 3.32. En la que se observa que la precisión de entrenamiento se estabiliza sobre el 90% aproximadamente a partir de la época 30 en la imagen 3.32a y que las pérdidas se estabilizan sobre 0.2. En la matriz de confusión de la figura 3.32c se observa lo mismo que en los casos anteriores, entre las clases 3 y 4 no hay problemas de distinción pero entre las clases 1 y 2 se producen la mayoría de los problemas para este sistema. Por otra parte, la precisión para el conjunto de datos de text es del 93%.

### 3.3.2 Convertir la red neuronal convolucional a HLS

Al igual que las otras redes neuronales desarrolladas, será necesario convertir el modelo de *Keras* a uno sintetizable por la *FPGA* mediante *HLS4ML*. Por lo tanto, en primer lugar habrá que hacer un análisis de la cuantificación necesaria para que el sistema y posteriormente analizar los recursos que consume.

Como ocurrió con las dos capas densamente conectadas de la sección anterior, se producen errores de cuantificación, dado que si se parte de los valores por defecto de la conversión del modelo con 16 bits y 6 de parte entera, se obtienen una precisión del 84% frente a los 93% teóricos. Por lo tanto, es necesario dedicar un apartado al análisis de la cuantificación del modelo a exportar.

#### 3.3.2.1 Estudio de la cuantificación en el modelo convolucional

Como se ha comentado, esta red neuronal también tiene problemas por lo que el primer paso será analizar el número de bits que requieren los distintos parámetros de la red para ser implementados. Estos se observan en la figura 3.33. Se evidencia que todos los valores con el reparto de  $\langle 16,6 \rangle$  están contenidos a excepción de los pesos de la convolución que requerirían al menos de 1 bit más en la parte fraccionaria. En caso de probar con una configuración de  $\langle 16,5 \rangle$  en la que todos los parámetros estuvieran bien cuantificados, la precisión sería del 83.3%. Por lo tanto, el error no se encuentra en este apartado.

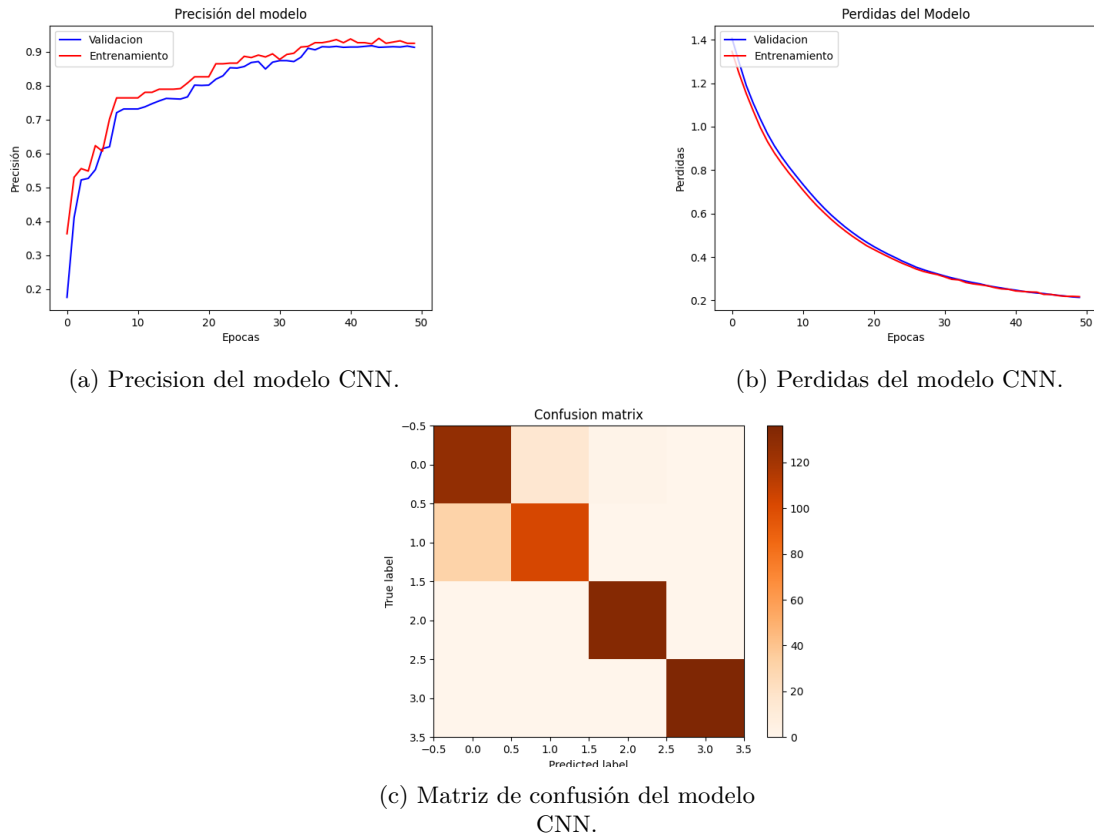


Figura 3.32: Resultados de entrenamiento de la red neuronal convolucional.

El siguiente aspecto a analizar es como varía la precisión en función del número de bits asignados al modelo en conjunto, para ello se realiza una variación en el número de bits y en la parte entera que se dedica de la palabra al igual que con la red densamente conectada. En este caso se probará con tamaños de palabra de 16 a 26 bits y la parte entera desde 0 hasta casi el tamaño máximo de palabra. Realizando este análisis se obtienen los resultados de la figura 3.34, en los que para partes enteras inferiores a 6 bits se obtiene una precisión insuficiente pero a partir de ahí permanece estable entre el 85 % y 7 % para caer cuando no hay suficiente parte decimal.

En la figura 3.34 también se observa que el máximo se obtiene con una parte entera de 11 para casi todos los tamaños de palabra, por lo que se procede a hacer un análisis para ese tamaño de la parte entera con distintos tamaños de palabra desde 10 hasta 32 bits. Como se observa en la figura 3.35, al principio las precisiones son muy pequeñas pero a partir de los 18 bits ya se alcanza el máximo de 86.6 % sobre el que se estabiliza la respuesta para todos los tamaños de palabra. Por lo tanto, la cuantificación del modelo no se puede analizar desde el punto de vista de modelo sino que se debe tratar en cada capa por separado intentando que sea igual a la respuesta teórica. Por lo que se deja como línea para futuros trabajos el alcanzar una cuantificación más fina para alcanzar la precisión teórica. Además, para futuras partes se utilizará un tamaño de palabra de  $\langle 16,6 \rangle$  dado que no introduce ninguna modificación en la capa de entrada con respecto a la red inicial, pudiendo usar los mismos conjuntos de pruebas y de escritura de datos.



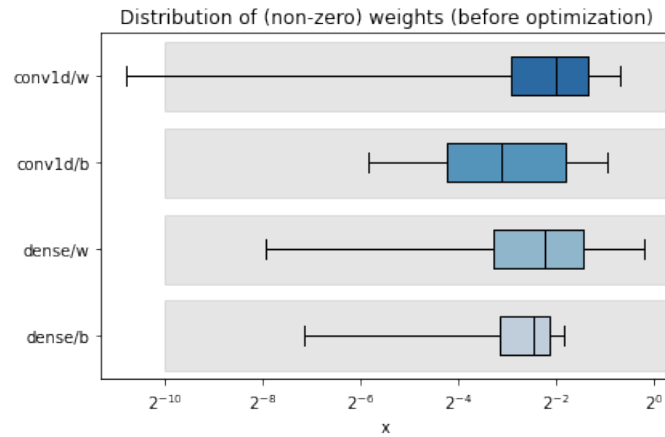


Figura 3.33: Cuantificación de los parámetros de la red neuronal convolucional.

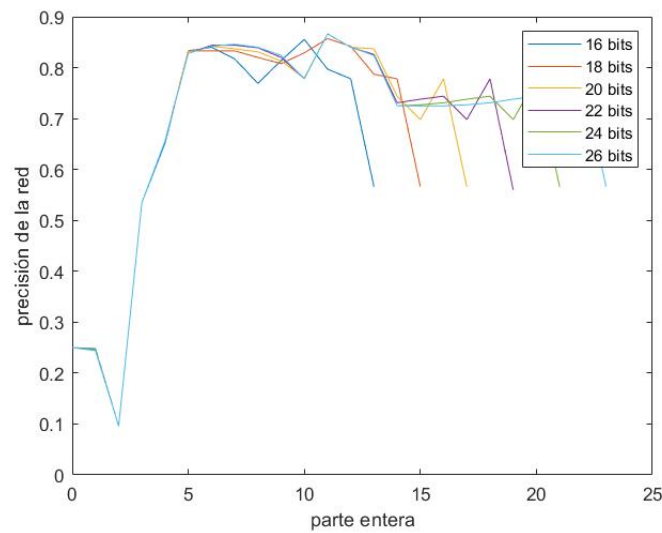


Figura 3.34: Variación del tamaño de palabra de la red neuronal convolucional.

### 3.3.2.2 Análisis de recursos y latencia resultante

Una vez se ha decidido la cuantificación del modelo, es hora de convertir el modelo en una síntesis en lógica. Para ello, como en los pasados capítulos, hay que establecer el valor del *reuse factor* correcto para que ocupe en Silicio el tamaño adecuado y cumpla con las restricciones de latencia. En el caso en cuestión estas limitaciones son únicamente la implementabilidad de la red con un menor número de ciclos de latencia.

Si se realizan múltiples síntesis del proyecto para valores del factor de reutilización de 1 a 10 se obtienen los resultados de la figura 3.36, en la que el consumo de DSP comienza alrededor del 200% para un *reuse factor* de 1 pero que decrece de manera hiperbólica hasta alrededor del 40%. Para un factor de reuso 3 la red es completamente sintetizable ya que el único aspecto limitante son los DSP, el resto de parámetros permanecen prácticamente constantes.

El otro aspecto a analizar es la latencia de la red. Esta es mucho mayor que en los casos anteriores, ya que para la implementación de esta red no es válida la utilización de una interfaz AXI o paralelo sino

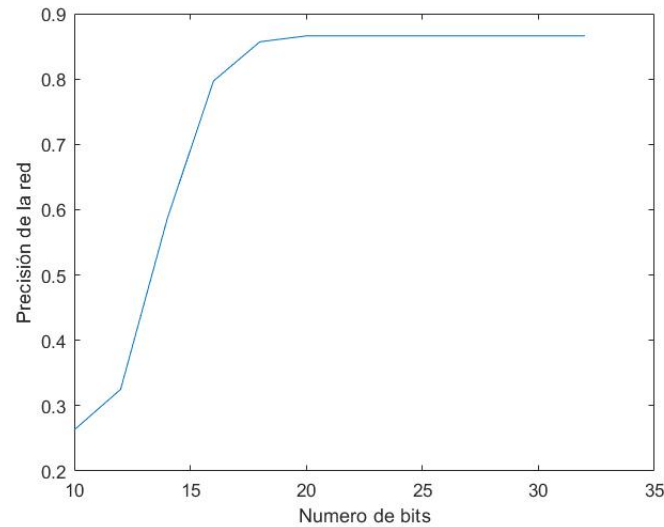


Figura 3.35: Variación del tamaño de palabra con la parte entera fija en 11 bits de la red neuronal convolucional.

que requiere de un *io\_stream* para ser sintetizable por *Vivado HLS*. La latencia resultante se observa en la figura 3.39 y es completamente lineal con una pendiente de 160 ciclos de reloj por reuse factor.

A la vista de los dos resultados previos, la primera red implementable será para un *reuse factor* igual a 3 que ocupará los recursos de la tabla 3.9. Para ella se obtendrá una latencia de 496 ciclos de reloj, equivalente a  $4.96 \mu\text{s}$  para un reloj de 100 MHz. Esta red neuronal será puesta en práctica en *Vivado* en la siguiente subsección.

Resource	Total	Available	Utilization (%)
LUT	26901	53200	50
FF	18284	106400	17
BRAM	22	280	7
DSP	163	220	74

Tabla 3.9: Recursos de NN en Vivado para estructura CNN.

### 3.3.3 Implementación en Vivado

Se ha exportado el IP de *HLS* a *Vivado* y se ha realizado un esquema similar a los de las secciones anteriores. Se ha llevado un esquema básico a implementar con el objetivo de observar los recursos consumidos y la máxima frecuencia que se puede alcanzar. Debido a la necesidad de adaptar la salida del *SoC* al módulo *Stream* que hay que realizar, este esquema no se llevará a un escenario real.

Sin embargo, en la tabla 3.10 se reflejan los detalles de la implementación. Se observa que el número de todos los recursos a excepción de los DSP, varían. En este caso, al igual que para las dos capas densas, no se han alcanzado los parámetros de *timing* para 100 MHz como se observa en la figura 3.38c. La mayor frecuencia que se alcanza es de 90 MHz con los resultados de la figura 3.38b. Para esta frecuencia se tendrá una latencia al realizar la inferencia de  $5.51 \mu\text{s}$ . El consumo de la placa completa es de 2.025W de los que el 82% de la potencia dinámica se corresponde al procesador y el resto al interfaz y computo de la red neuronal.

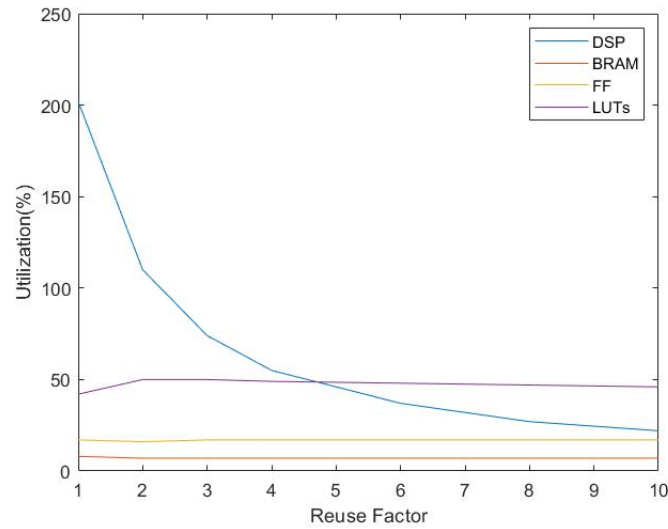


Figura 3.36: Utilización de la CNN en función del *Reuse Factor*.

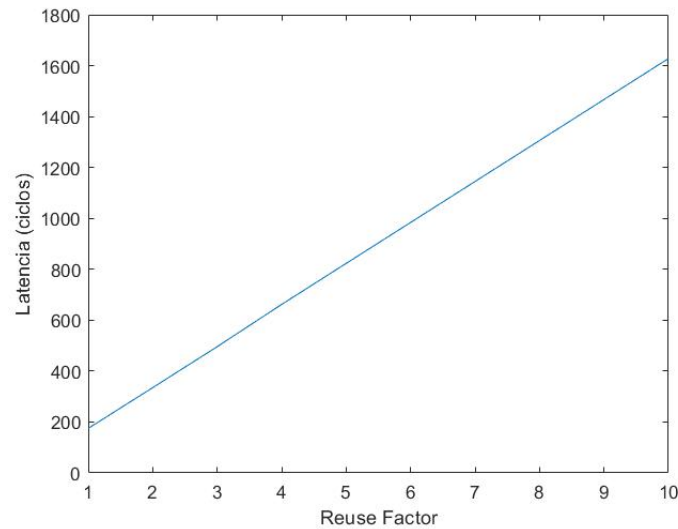


Figura 3.37: Latencia de la CNN en función del *Reuse Factor*.

Resource	Total	Available	Utilization (%)
LUT	14434	53200	27.13
FF	19519	106400	18.34
BRAM	11	280	7.86
DSP	163	220	74.09

Tabla 3.10: Recursos de NN en Vivado para estructura CNN en la implementación.

**Design Timing Summary**

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,405 ns	Worst Hold Slack (WHS): 0,007 ns	Worst Pulse Width Slack (WPWS): 5,519 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 41494	Total Number of Endpoints: 41494	Total Number of Endpoints: 19721

All user specified timing constraints are met.

(a) Análisis temporal para la red CNN a 75MHz.

**Design Timing Summary**

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,116 ns	Worst Hold Slack (WHS): 0,014 ns	Worst Pulse Width Slack (WPWS): 4,520 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 41506	Total Number of Endpoints: 41506	Total Number of Endpoints: 19727

All user specified timing constraints are met.

(b) Análisis temporal para la red CNN a 90MHz.

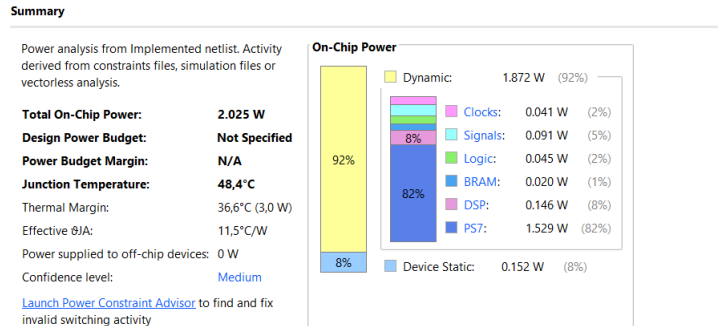
**Design Timing Summary**

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0,295 ns	Worst Hold Slack (WHS): 0,015 ns	Worst Pulse Width Slack (WPWS): 4,020 ns
Total Negative Slack (TNS): -19,170 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 212	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 41704	Total Number of Endpoints: 41704	Total Number of Endpoints: 19826

Timing constraints are not met.

(c) Análisis temporal para la red CNN a 100MHz.

Figura 3.38: Análisis temporal de la red CNN para distintas frecuencias

Figura 3.39: Latencia de la CNN en función del *Reuse Factor*.

### 3.4 Comparativa de las distintas estructuras en FPGA

Las FPGAs se han visto como un medio muy acertado para la implementación de redes neuronales. Para una tarjeta de recursos limitados se ha observado la posibilidad de implementar estos algoritmos. A pesar de eso, se ha percibido que el diseño no es tan sencillo dado que hay que prestar especial atención a la cuantificación para obtener unos resultados similares a los teóricos. También se ha observado la limitación del tipo de neuronas que se pueden implementar dado que por ejemplo la recurrente no estaba disponible dentro del paquete *hls4ml*. Una red recurrente podía haber sido diseñada a mano mediante lenguajes de descripción *hardware* como *vhdl* o en lenguajes de alto nivel como C y una posterior síntesis a *Hardware*, pero estos tendrían una mayor complejidad y abarcarían un TFM en su totalidad.

Otro aspecto apreciable dentro del paquete *hls4ml* es la disponibilidad de interfaces. Para diseños sencillos y con neuronas densamente conectadas se permitía la utilización de cualquiera de ellos pero al utilizar redes convolucionales se ha restringido únicamente a interfaces *stream* debido a su mayor complejidad computacional.

Al igual, a medida que crecía la complejidad y el tamaño de la red, el *timing* se ha visto comprometido, obligando a reducir la frecuencia de operación. Vinculado a esto está la latencia que tiene una alta dependencia del número de parámetros y operaciones y del factor de reuso. Aún así, para las redes que se han implementado no se han superado en ningún momento los 6  $\mu$ s.

La utilización a su vez también va ampliamente ligada con los parámetros vistos previamente, pero cabe destacar que variando el factor de reuso lo que más variaba era el número de *DSP* utilizados. En caso de que la red tuviera insuficientes *LUTs* o *FF* sería difícil alcanzar un diseño implementado variando ese parámetro. Obligando al diseño de una implementación a medida en *VHDL* mejorando las optimizaciones de *hls4ml* y *VIVADO HLS* o a cambiar de dispositivo a uno con más recursos.



## Capítulo 4

# Implementación de Redes Neuronales en sistemas basados en Microprocesador

En el anterior apartado se analizó el despliegue de las redes neuronales dentro de las *FPGA* de *Xilinx*, en este sin embargo se analizará en el otro tipo de sistemas empujados, los  $\mu\text{C}$  y en concreto los de *STMicroelectronics*. Para ello se utilizará la tarjeta Nucleo-L496ZG-P [16] con un MCU STM32L496 basado en un procesador ARM Cortex-M4.

Al igual que en el apartado anterior, se comenzará con el despliegue de la red neuronal más básica que permitirá explicar las distintas herramientas que se emplean y posteriormente se aumentará la complejidad como se hizo con las *FPGA*. Las distintas redes que se emplearán ya se han visto en el capítulo anterior, por lo tanto no se volverán a analizar de nuevo si no que se partirá de su modelo *h5*, aunque se analizará también una red recurrente desde el diseño de la misma, explicando su arquitectura, las curvas de aprendizaje y pérdidas y la precisión alcanzada para el conjunto de test.

### 4.1 Red neuronal básica en uC

La primera red neuronal a analizar es la que se vio en la sección 3.1 consistente únicamente en las 4 neuronas de clasificación de las 4 clases correspondientes con funciones de activación *Soft-Max*. El IDE empleado para la programación de los micros será el de *STMicroelectronics* llamado *STM32IDE* [17]. Este ofrece funciones para la programación completa de la tarjeta junto con un interfaz gráfica capaz de generar código correspondiente a librerías asociadas. Además permite la utilización de distintos *Software Packs* que permiten facilitar la implementación de distintos servicios. Cabe destacar el componente *STMicroelectronics-X-Cube-AI* [18] encargado de convertir la red neuronal en formato *h5* a una simple API, facilitando las tareas del desarrollador e informando de los recursos utilizados por la red.

Este paquete no admitía la versión de *Keras* empleada en el capítulo previo. Por lo tanto, se ha tenido que bajar de versión de *software* y reentrenar las redes para que sean guardadas de manera correcta para el programa de *STM*. Debido a esto la precisión de las redes neuronales puede variar ligeramente con respecto a las ya vistas.

Para analizar este apartado se dividirá en dos bloques, el primero enfocado en la utilización del paquete *STMicroelectronics-X-Cube-AI* culminando en la generación de la *API*, y el segundo en el empleo de la *API* y los resultados obtenidos en cada caso.

### 4.1.1 Utilización del paquete *STMicroelectronics-X-Cube-AI*

Esta herramienta, como se comentó previamente, permite convertir los pesos de la red neuronal y su funcionamiento únicamente en una serie de funciones dentro de una *API*. El interfaz de este software se observa en la figura 4.1.

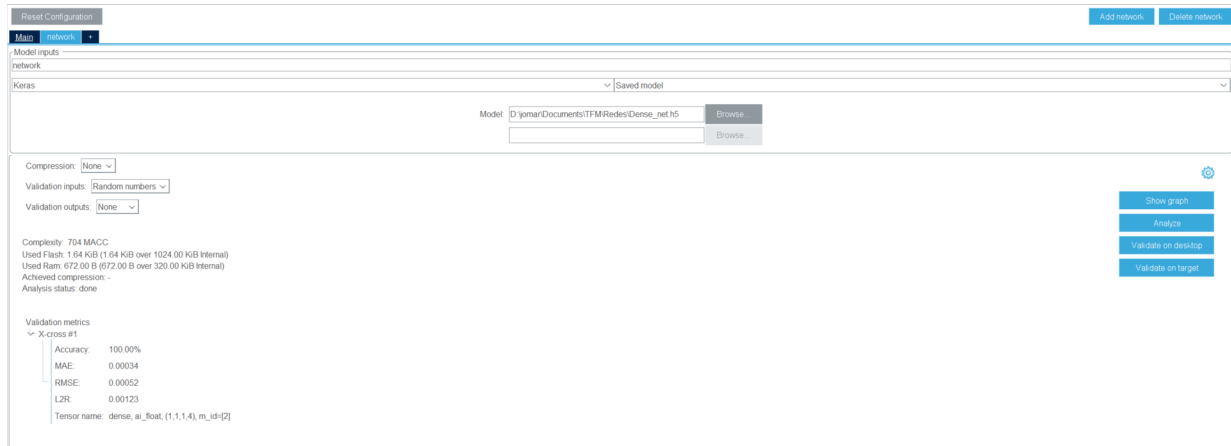


Figura 4.1: Interfaz X-Cube-AI.

La herramienta permite seleccionar distintos factores de compresión para reducir el consumo en memoria utilizado por los pesos del sistema. Los resultados de variar la compresión para la red básica que se está analizando se observan en la tabla 4.1. Como se aprecia el número de MACC utilizado para todos los valores de compresión disponibles es el mismo. Por otra parte, la utilización de *Flash* y *RAM* aumenta en los mecanismos con compresión con respecto al que no tiene. Esto es debido a que, para una red tan sencilla con tan poca ocupación en memoria, utiliza más memoria el mecanismo de compresión que los pesos de la red. Por lo tanto, se continuará con el sistema normal.

	Sin Compresión	Compresión = 4	Compresión = 8
Complexity(MACC)	704	704	704
Used Flash	1.64 KB	12.45 KB	11.20 KB
Used RAM	672B	1.76KiB	1.76 KiB
Achieved Compression	0%	3.93%	7.67%

Tabla 4.1: Compresión obtenida para la red básica.

También permite visualizar la arquitectura de la red con la cantidad de MAC y memoria dedicados a cada capa de la red como se observa en la figura 4.2 en la que se observa la capa densa con 644 MAC y su capa de activación con 60.

Otro aspecto que se puede seleccionar es como realizar la validación del sistema, que se puede hacer tanto con valores aleatorios o con la librería de validación del sistema en formato *dat*. Dado que se tienen los ficheros del conjunto de datos de test, se realizará la validación en el ordenador con esta biblioteca. Al realizar dicho análisis se obtienen distintos resultados. El primero de ellos es el tiempo que tarda en



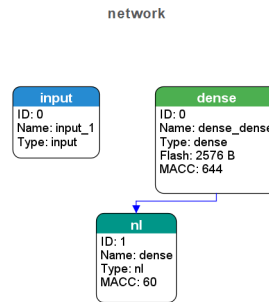


Figura 4.2: Arquitectura red básica X-Cube-AI.

realizar la inferencia y que se aprecia en la figura 4.4. Se observa que con la simulación de la placa el tiempo que tarda en realizar la inferencia es de 0.004 ms y que la multiplicación y suma de los pesos ocupa un mayor tiempo que la aplicación de la función de activación. Cabe destacar que estos datos no se corresponden con la realidad ya que dependen del procesador y de la ocupación del mismo.

```

-----
Results for 547 inference(s) - average per inference
device           : AMD64 Intel64 Family 6 Model 165 Stepping 2, GenuineIntel (Windows)
duration         : 0.004ms
c_nodes         : 2
c_id  m_id  desc          output          ms      %
-----
0     2     Dense (0x104)  (1,1,1,4)/float32/16B  0.003  70.8%
1     2     NL (0x107)    (1,1,1,4)/float32/16B  0.001  29.2%
-----
                                           0.004 ms
NOTE: duration and exec time per layer is just an indication. They are dependent of the HOST-machine work-load
  
```

Figura 4.3: Tiempo en realizar la inferencia en la validación de la red básica.

Otros datos de interés son la precisión del sistema, que se compara con los datos de validación y el modelo a alto nivel. Con los datos de referencia se obtienen la misma precisión que en el modelo teórico del 79.71%. Como se observa en la comparación con el modelo original la precisión es del 100%, es decir, con la cuantificación de *float32* empleada por STM no varían los resultados con respecto a los teóricos.

```

Accuracy report #1 for the reference model
-----
notes: - computed against the provided ground truth values
        - 547 samples (4 items per sample)
acc=79.71%, rmse=0.303721756, mae=0.207871869, l2r=0.828545868
4 classes (547 samples)
-----
C0    142  1  1  .
C1    71  54  .  9
C2    .  .  133  .
C3    .  14  15  107
Cross accuracy report #1 (reference vs C-model)
-----
notes: - the output of the reference model is used as ground truth/reference value
        - 547 samples (4 items per sample)
acc=100.00%, rmse=0.000000068, mae=0.000000041, l2r=0.000000185
4 classes (547 samples)
-----
C0    213  .  .  .
C1    .  69  .  .
C2    .  .  149  .
C3    .  .  .  116
Evaluation report (summary)
-----
Output      acc      rmse      mae      l2r      mean      std      tensor
-----
x86 c-model #1  79.71%  0.303721756  0.207871869  0.828545868  -0.000000000  0.303791195  dense, ai_float, (1,1,1,4), m_id=[2]
original model #1  79.71%  0.303721756  0.207871869  0.828545868  0.000000001  0.303791195  dense, ai_float, (1,1,1,4), m_id=[2]
X-cross #1    100.00%  0.000000068  0.000000041  0.000000185  0.000000000  0.000000068  dense, ai_float, (1,1,1,4), m_id=[2]
-----
Creating txt report file C:\Users\jcomar\.stm32cubeux\network_output\network_validate_report.txt
elapsed time (validate): 1.819s
Validation ended
  
```

Figura 4.4: Precisión de la red básica según X\_cubeX\_AI.

Estos mismos resultados se pueden obtener sobre la placa utilizando la funcionalidad de validación sobre el *target*, pero para analizar el funcionamiento de la API generada se recreará la misma funcionalidad generando el código correspondiente.

#### 4.1.2 Utilización de la API y resultados

Tras la utilización de X-Cube-AI, se genera una carpeta con el mismo nombre dentro del *workspace* con la configuración, pesos y funciones para la utilización de la red neuronal. De las funciones existentes, en primer lugar esta la que se encargará de crear e inicializar la red que será *ai\_network\_create\_and\_init*. A partir de ahí se generan los punteros de entrada y salida del sistema con *ai\_network\_inputs\_get* y *ai\_network\_outputs\_get*.

A partir de ahí todo el proceso de inicialización habría terminado. Por lo tanto, solo quedaría la inferencia que se realiza con la función *ai\_network\_run*, que recibe la red y la entrada y salida del sistema y devuelve el tamaño del *batch*.

Por lo tanto, para la evaluación de la inferencia se hará pasar por la red todos los elementos de la base de datos de test. La problemática de esto es que el tamaño completo de las entradas son 87.520 elementos en formato *float* de 32 bits. Por lo tanto la ocupación total es de 350 KB aproximadamente, superior al tamaño de la RAM que es de 256 KB. Se podría emplear la flash para almacenar los datos dado que esta es de 1 MB pero se opta por la división de las muestras de test en cinco subconjuntos de 100 elementos y uno de 47 que se analizarán en 6 ejecuciones diferentes. Ocupando aproximadamente 62 KB cada uno. Las salidas de la red para realizar la evaluación de la precisión dentro de la tarjeta ocuparán 400 Bytes ya que se almacenan en formato *float* aunque podrían reducirse empleando otros tipos menores dado que sus valores se encuentran entre 1 y 4.

Por consiguiente, la cadena de ejecución del problema será en primer lugar inicializar el procesador junto con el *UART* y la red, realizar la inferencia de las 100 muestras y almacenar sus resultados. A partir de esa respuesta se calculará la precisión de la red para compararla con la teórica. También antes de entrar en el bucle encargado de la inferencia se tomará una marca de tiempo para calcular el tiempo del bucle completo en *ms*. Este tiempo no es únicamente destinado a la inferencia sino que también va destinado a mover los datos del *array* donde se guardan al buffer de entrada del procesador, pero permite aproximar dicho tiempo.

La *UART* se empleará para comunicarse con la tarjeta y comprobar la salida que va teniendo en cada momento. Funcionará a 19200 baudios y un tamaño de palabra de 8 bits con un único bit de parada y sin paridad. En concreto sobre esta interfaz se mostrarán 3 mensajes. El primero para indicar que se ha iniciado correctamente el sistema y posteriormente se mostrará el número de aciertos realizados en la inferencia y el tiempo de esta en milisegundos. Estos resultados se muestran en la figura 4.5 con una captura del programa *Tera Term* con la ejecución del primer subconjunto de datos.

La imagen previa muestra los resultados para un subconjunto de datos, para el resto se muestran en la tabla 4.2. Se observa en ella que ha habido 436 aciertos de 547 ejecuciones suponiendo el 79.70%, igual a los obtenidos teóricamente. Por lo tanto, no se comete ningún error en cuanto a cuantificación y resultados al realizar la inferencia en el dispositivo. Por otra parte, se observa que la ejecución de 100 inferencias con sus respectivos movimientos de memoria a los *buffers* tarda 12 ms, por lo tanto el tiempo por inferencia será de aproximadamente 120  $\mu$ s para esta red neuronal.

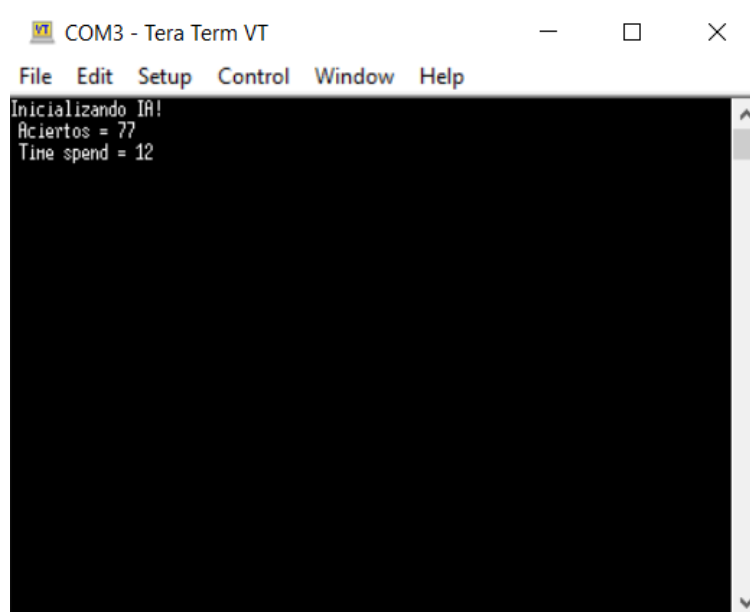


Figura 4.5: Captura de los resultados de la red básica.

Subconjunto	1	2	3	4	5	6
Aciertos	77	79	82	78	81	39
Tiempo utilizado (ms)	12	12	12	12	12	5

Tabla 4.2: Resultados en  $\mu C$  para la red básica.

## 4.2 Red neuronal de dos capas densamente conectadas

La pasada red neuronal era la más sencilla de acuerdo a la clasificación de 4 clases. Por lo tanto, al igual que en la FPGA se va a complicar el diseño de la red para comprobar los resultados que ofrece. Se va a utilizar en concreto la diseñada para el apartado 3.2, consistente en una red neuronal de 2 capas con 8 neuronas en la primera capa y 4 en la capa de salida. Debido a los problemas de compatibilidad entre las versiones de *Keras* comentados antes se ha tenido que reentrenar la red para guardar el modelo y no se han conseguido los mismos resultados. En este caso la precisión de las muestras de test es del 92.32%, ligeramente inferior al 95 % que se obtuvo antes.

Al igual que en la sección anterior en primer lugar se mostrarán los resultados ofrecidos por el software *X-CUBE-AI* de *STM* sobre el ordenador y posteriormente se ejecutará la inferencia de los datos de test sobre la misma placa *Nucleo-L496ZG*.

En primer lugar, en la herramienta *X-CUBE-AI* se introduce la red neuronal y se obtienen los resultados del análisis de la red en la figura 4.6. En este caso el número de MACC es de 1392, el doble que en la red previa y el 93 % se encuentran en la primera capa densa.

En cuanto a la memoria utilizada por el sistema se muestra en la siguiente tabla 4.3. Se observa que en este caso la RAM utilizada no varía en función del factor de compresión, pero la Flash utilizada sí que se reduce para los distintos casos. Aún así, como el sistema sin compresión ocupa una zona pequeña de memoria y se asume una menor latencia que con compresión, será la que se utilizará para los posteriores resultados.

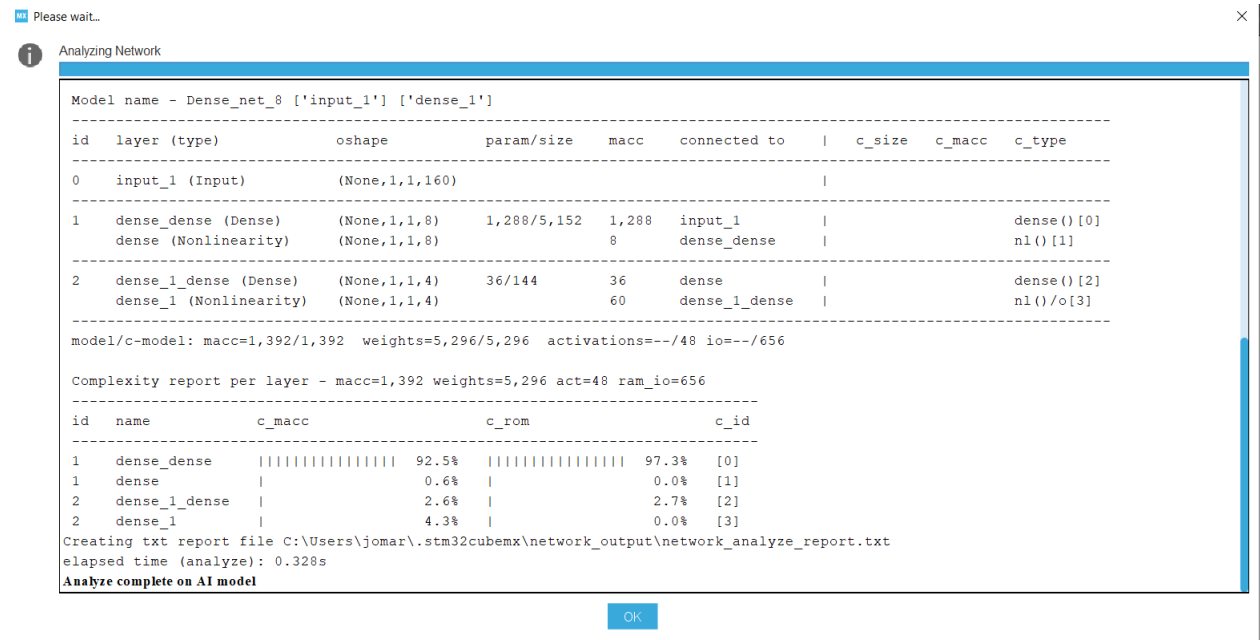


Figura 4.6: Análisis de la red de dos capas.

	Sin Compresión	Compresión = 4	Compresión = 8
Used Flash	17.12 KB	14.37 KB	12.81 KB
Used RAM	2.23 KB	2.23 KB	2.23 KB
Achieved Compression	0 %	3.64 %	6.5 %

Tabla 4.3: Compresión obtenida para la red de dos capas.

Posteriormente se realiza el análisis de la red en el ordenador y en primer lugar se observan los resultados de precisión de la figura 4.7. En él la teórica y la de la placa simulada son exactamente las mismas, por lo que no hay un error en cuanto a la cuantificación usada. Por otra parte, también se tiene que analizar el aspecto temporal que ofrece este análisis como se evidencia en la figura 4.8. A pesar de que este tiempo es sumamente orientativo, dado que depende del ordenador utilizado y de los procesos del sistema operativo, indica que el 66 % del tiempo consumido estará en la primera capa y que la duración por inferencia es de 0.012 ms.

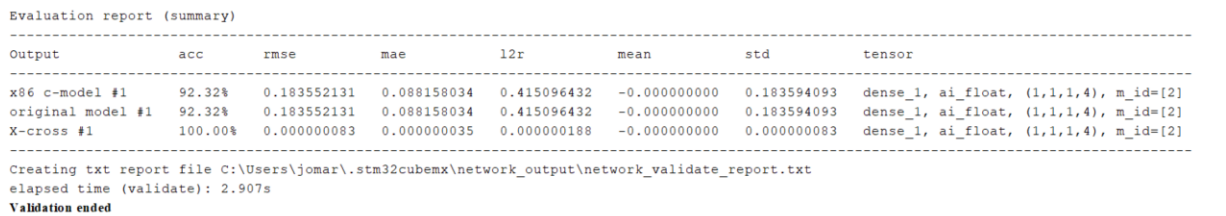


Figura 4.7: Comparativa de resultados de la red de dos capas.

Tras comprobar el funcionamiento en el ordenador, se volverá a reproducir el comportamiento sobre la placa con el mismo código que en la sección anterior pero cambiando únicamente el nombre de las funciones de la API. Se va a medir en las 6 ejecuciones de los subconjuntos de test el número de aciertos y el tiempo empleado en la inferencia. Esto se referencia en la tabla 4.4 con un total de 505 aciertos suponiendo una precisión del 92.32 % y un tiempo medio por inferencia de 260  $\mu$ s con una frecuencia de reloj de 80 MHz.

```
Results for 547 inference(s) - average per inference
device      : AMD64 Intel64 Family 6 Model 165 Stepping 2, GenuineIntel (Windows)
duration    : 0.012ms
c_nodes     : 4
c_id  m_id desc          output          ms          %
-----
0     1   Dense (0x104) (1,1,1,8)/float32/32B  0.006  50.8%
1     1   NL (0x107)    (1,1,1,8)/float32/32B  0.002  16.5%
2     2   Dense (0x104) (1,1,1,4)/float32/16B  0.002  13.5%
3     2   NL (0x107)    (1,1,1,4)/float32/16B  0.002  19.1%
-----
                                0.012 ms
```

Figura 4.8: Comparativa de tiempos de la red de dos capas.

Subconjunto	1	2	3	4	5	6
Aciertos	93	97	92	93	90	40
Tiempo utilizado (ms)	26	26	26	26	26	12

Tabla 4.4: Resultados en  $\mu C$  para la red de 2 capas.

### 4.3 Red neuronal convolucional

De igual manera que en el capítulo anterior, primero se probó con las redes neuronales *fully connected* y posteriormente se avanzó con las neuronas convolucionales, se hará lo mismo en esta sección. Se empleará para ello la misma arquitectura de red convolucional aunque reentrenada con una precisión del 91.96%.

Siguiendo el mismo procedimiento que con las redes densas, se importa el modelo de la red convolucional al paquete *X-CUBE-AI*. Analizando el modelo generado se observa que en este caso requiere 31.364 operaciones de multiplicación y acumulación, siendo casi 22 veces mayor que la sección anterior. Por lo tanto, se observa que la complejidad de esta red neuronal es mucho mayor. Con respecto a la capa que más complejidad tiene es la convolucional como se observa en la figura 4.9.

```
Model name - CNN_net ['input_1'] ['dense']
-----
id  layer (type)          oshape          param/size  macc  connected to
-----
0   input_1 (Input)      (None,160,1,1)
-----
1   convld_conv2d (Conv2D) (None,141,1,10)  210/840    28,210  input_1
   convld (Nonlinearity) (None,141,1,10)  1,410    convld_conv2d
-----
2   max_pooling1d (Pool)  (None,7,1,10)    1,400    convld
-----
3   flatten (Reshape)    (None,1,1,70)
-----
4   dense_dense (Dense)   (None,1,1,4)     284/1,136  284    flatten
   dense (Nonlinearity) (None,1,1,4)     60        dense_dense
-----
model/c-model: macc=31,364/31,364 weights=1,976/1,976 activations=-/1,080 io=-/656

Complexity report per layer - macc=31,364 weights=1,976 act=1,080 ram_io=656
-----
id  name          c_macc          c_rom          c_id
-----
1   convld_conv2d | 98.9% | 42.5% | [0]
4   dense_dense   | 0.9% | 57.5% | [1]
4   dense         | 0.2% | 0.0% | [2]
```

Figura 4.9: Análisis de la red convolucional.

En cuanto al consumo de memoria no ha habido grandes cambios con respecto a la anterior red, puesto que en ocupación de flash antes era de 17.12 KB y en este caso de 18.91 KB; y en caso de RAM pasa de 2.23 KB a 3.30 KB. Con respecto a la variación sufrida al cambiar el factor de compresión se observa en la tabla 4.5. Se percibe que en este caso la compresión es mucho menor. Para un factor de compresión de

4 no consigue disminuir la ocupación y para 8 lo reduce ligeramente. Por lo tanto, se continuará con la red sin compresión.

	Sin Compresión	Compresión = 4	Compresión = 8
Used Flash	18.91 KB	18.91 KB	18.02 KB
Used RAM	3.30 KB	3.30 KB	3.30 KB
Achieved Compression	0%	0%	1.99%

Tabla 4.5: Compresión obtenida para la red de convolucional.

Tras el análisis se pasará a probar sobre el ordenador la red neuronal al igual que se hizo previamente. Los resultados de cuantificación que se observan en la figura 4.10. Como se ve, el error cometido en este apartado también es nulo. En cuanto al tiempo que tarda en la ejecución de la base de datos en este tipo de evaluación, se percibe en la figura 4.11, donde un 98.2% del tiempo se dedica a la capa convolucional.

```

-----
Output          acc      rmse      mae      12r
-----
x86 c-model #1  91.96%   0.169161782  0.078541294  0.378186882
original model #1 91.96%   0.169161797  0.078541294  0.378186911
X-cross #1      100.00%  0.000000058  0.000000025  0.000000129
-----

```

Figura 4.10: Error de cuantificación red convolucional para  $\mu C$ .

```

Results for 547 inference(s) - average per inference
device          : AMD64 Intel64 Family 6 Model 165 Stepping 2, GenuineIntel (Windows)
duration        : 0.290ms
c_nodes        : 3
c_id  m_id  desc          output          ms      %
-----
0     2     Conv2dPool (0x109) (1,7,1,10)/float32/280B  0.285  98.2%
1     4     Dense (0x104)  (1,1,1,4)/float32/16B   0.003   1.0%
2     4     NL (0x107)    (1,1,1,4)/float32/16B   0.002   0.8%
-----

```

Figura 4.11: Tiempo en ordenador de la red convolucional.

Tras completar este paso, se procederá a realizar la inferencia dentro de la tarjeta. Para ello se vuelve a emplear el código de las secciones anteriores con modificaciones en el nombre de las funciones de la API pero con el mismo funcionamiento. De la misma manera se evaluarán los 6 subconjuntos y sus resultados temporales y la cantidad de aciertos se observan en la tabla 4.6. Al igual que en las otras redes el error de cuantificación es nulo ya que la precisión es del 91.96%, igual a la teórica. Por otra parte, el tiempo por operación ha aumentado bastante, en este caso tardará 16.11 ms en procesar una única inferencia.

## 4.4 Red neuronal recurrente

La red convolucional ya suponía una gran cantidad de operaciones para el  $\mu C$ . Aún así la red original del artículo base y que se comentó en el capítulo 2.3, se trataba de una red recurrente. Esta no se pudo implementar en FPGA debido a no ser compatible con el paquete *hls4ml*, pero por el contrario STM sí permite la utilización de este tipo de neuronas.

Subconjunto	1	2	3	4	5	6
Aciertos	91	93	88	91	97	43
Tiempo utilizado (ms)	1611	1611	1611	1611	1611	757

Tabla 4.6: Resultados en  $\mu C$  para la red convolucional.

Layer(type)	Output Shape	Param
input_1 (Input Layer)	[160,1]	0
lstm (LSTM)	[160,10]	480
lstm_1 (LSTM)	[20]	2480
flatten(Flatten)	[25]	0
dense (Dense)	[4]	84
Total params: 3,044		
Trainable params: 3,044		
Non-trainable params: 0		

Tabla 4.7: Resumen de la red neurona básica.

Por lo tanto, se ha partido del diseño original simplificándolo para no suponer un gran tamaño en cuestión de operaciones y memoria, y facilitando la implementación únicamente teniendo que realizar breves modificaciones en las funciones desarrolladas.

La red empleada dispondrá de la entrada de 160 muestras como las anteriores y después tendrá una primera capa recurrente con 10 neuronas. Seguida de esta tendrá una segunda capa recurrente de 20 y finalmente las 4 neuronas de clasificación densamente conectadas con activación *SoftMax*. El esquema de la red se presenta en la figura 4.12

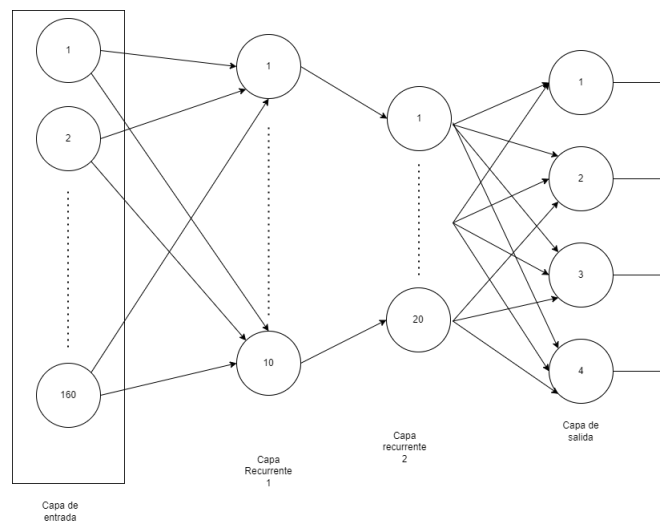
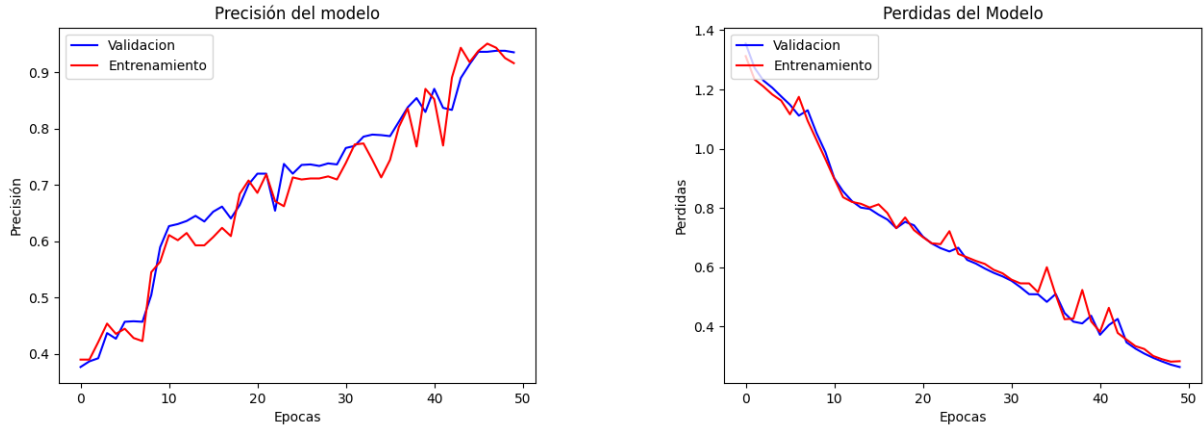


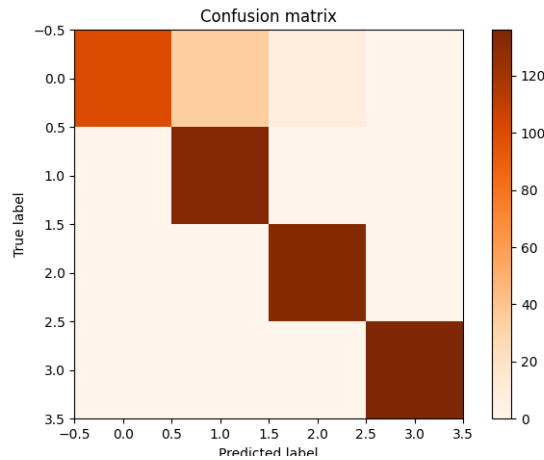
Figura 4.12: Esquema de la red recurrente.

Se observa que esta red es también la más grande que se va a implementar en cuanto a número de parámetros y número de capas y se verán sus características dentro del dispositivo. Aunque lo primero es realizar el entrenamiento. En él se observa en la figura 4.13a y 4.13b la curva de precisión y de pérdidas respectivamente con los mismos hiperparámetros que en las redes previas. Se consigue una red neuronal recurrente con una precisión del 91.95 % y la matriz de confusión de la figura 4.13c. Se percibe en las dos primeras figuras que se está produciendo *underfitting* ya que las pérdidas siguen disminuyendo y la precisión aumentando en los dos conjuntos de validación y entrenamiento. Aún así como la precisión



(a) Precisión del modelo recurrente.

(b) Pérdidas del modelo recurrente.



(c) Matriz de confusión del modelo recurrente.

Figura 4.13: Resultados de entrenamiento de la red neuronal básica.

alcanzada ha sido suficiente en comparación con el resto y para evitar reentrenar con un mayor número de épocas se utiliza para el resto de la sección.

Tras el diseño de la red, esta se importa dentro de *X-CUBE-AI*. Como se observa en la figura 4.14, el número de MACs en este caso ha aumentado con respecto a la red convolucional. De estos, la inmensa mayoría se encuentran en la segunda capa recurrente con el 83.6% de ellas.

En este caso el consumo de memoria es en torno al doble con respecto a la convolucional. Con la compresión tampoco se consigue una disminución clara como se observa en la tabla 4.8. Para un factor de 4 no consigue disminuirlo y para 8 alcanza un 1.03 de compresión. Por lo tanto se continúa con la red sin ninguna compresión como en el resto de secciones.

	Sin Compresión	Compresión = 4	Compresión = 8
Used Flash	32.23 KB	32.23 KB	32.02 KB
Used RAM	9.71 KB	9.79 KB	9.79 KB
Achieved Compression	0%	0%	1.99%

Tabla 4.8: Compresión obtenida para la red de recurrente.



```

Model name - RNN ['input_1'] ['dense']
-----
id  layer (type)          oshape          param/size      macc           connected to   |
-----
0   input_1 (Input)       (None,160,1,1)                               |
-----
1   lstm (LSTM)           (None,160,1,10) 480/1,920      78,400         input_1        |
-----
2   lstm_1 (LSTM)         (None,1,1,20)   2,480/9,920    400,000        lstm           |
-----
3   flatten (Reshape)     (None,1,1,20)                               lstm_1         |
-----
4   dense_dense (Dense)   (None,1,1,4)     84/336          84             flatten        |
    dense (Nonlinearity) (None,1,1,4)     60             dense_dense    |
-----
model/c-model: macc=478,544/478,544 weights=12,176/12,536 +360(+3.0%) activations=--/

Complexity report per layer - macc=478,544 weights=12,536 act=7,040 ram_io=656
-----
id  name          c_macc          c_rom          c_id
-----
1   lstm          |||             16.4%         ||||           16.3%         [0]
2   lstm_1        |||             83.6%         |||            81.0%         [1]
4   dense_dense  |              0.0%         |              2.7%         [2]
4   dense        |              0.0%         |              0.0%         [3]
    
```

Figura 4.14: Análisis de la red recurrente.

Posteriormente se realiza la validación en el escritorio. El aspecto temporal se refleja en la figura 4.15, en la que se observa que la duración en tiempo estimada está en 2.329 ms, 10 veces mayor que la convolucional, por lo que se espera que cuando se realice la inferencia en la tarjeta aumentará en gran medida el tiempo. Por otra parte, el 73.4% de los tiempos se gastan en la segunda capa que es la que más operaciones realiza.

```

STM.IO: 0%|          | 0/547 [00:00<?, ?it/s]
STM.IO: 69%|#####0 | 376/547 [00:00<00:00, 3758.9lit/s]
STM.IO: 89%|#####9 | 489/547 [00:00<00:00, 946.06it/s]
Results for 547 inference(s) - average per inference
device      : AMD64 Intel64 Family 6 Model 165 Stepping 2, GenuineIntel (Windows)
duration    : 2.329ms
c_nodes     : 4
c_id  m_id desc          output          ms          %
-----
0   1   LSTM (0x181) (1,160,1,10)/float32/6400B 0.614 26.4%
1   2   LSTM (0x181) (1,1,1,20)/float32/800B 1.710 73.4%
2   4   Dense (0x104) (1,1,1,4)/float32/16B 0.002 0.1%
3   4   NL (0x107) (1,1,1,4)/float32/16B 0.002 0.1%
    
```

Figura 4.15: Tiempo en la evaluación en ordenador de la red recurrente.

Por otra parte desde el punto de vista de cuantificación se consigue alcanzar la misma precisión que en el aspecto teórico como sucedía en el resto de redes, como se observa en la figura 4.16.

```

Evaluation report (summary)
-----
Output      acc      rmse      mae      l2r      mean      std      tensor
-----
x86 c-model #1  91.96%  0.191099256  0.099530734  0.441928416 -0.000000001  0.191142946  dense, ai_float, (1,1,1,4), m_id=[4]
original model #1  91.96%  0.190617830  0.099696316  0.441227406  0.000000000  0.190661401  dense, ai_float, (1,1,1,4), m_id=[4]
X-cross #1    100.00%  0.004081715  0.002433658  0.009439209 -0.000000001  0.004082648  dense, ai_float, (1,1,1,4), m_id=[4]
    
```

Figura 4.16: Precisión entre el modelo de STM y el teórico.

Finalmente se introduce la red neuronal dentro del dispositivo para las pruebas dentro de la placa. Se utilizará para ello el mismo código que para las secciones anteriores pero cambiando el nombre de las funciones de las APIs. En la tabla 4.9 se muestra la respuesta del sistema para los 6 subconjuntos con

una precisión del 91.96%, igual a la teórica. Por otra parte la tabla también muestra que el tiempo de inferencia medio es de 0.095 segundos.

Subconjunto	1	2	3	4	5	6
Aciertos	93	97	93	92	87	41
Tiempo utilizado (ms)	16028	16054	16048	16055	16017	7531

Tabla 4.9: Resultados en  $\mu C$  para la red recurrente.

## 4.5 Comparativa entre la implementación de las distintas neuronas en $\mu C$

Todas las redes que se han implementado a excepción de la primera son capaces de alcanzar precisiones a partir de 90%. Por lo tanto, para tres tipos de neuronas diferentes se ha podido observar que la complejidad de la red varia en función del tipo de capa. Para la red únicamente densa de 10 neuronas se tenían 1400 operaciones de multiplicación y suma frente a las 32000 para 10 neuronas convolucionales y 500000 en 30 neuronas recurrentes. Por ello, es fácilmente observable que cada neurona tendrá una complejidad característica asociada con su número de entradas aunque no sé puede analizar de manera directa ya que la arquitectura de cada una de las redes es diferente.

Otro aspecto relevante que se ha analizado son los tiempos que tarda la inferencia que suponen un aspecto critico a la hora de diseñar la aplicación, ya que marcarán su velocidad de operación, consumo de potencia, tiempo que se puede dedicar a otros recursos etc. Destacando que a medida que la red sea más grande o más compleja, aumentará el número de operaciones y por lo tanto el tiempo. Esto se observa claramente en el diseño de la red densa de 8 neuronas en la primera capa y la convolucional de 10 neuronas. A pesar de tener una arquitectura similar, la convolucional hace 22 veces más operaciones y tarda 61 veces más tiempo. En paralelo a este aspecto irá la ocupación en memoria que también aumentará en función de la arquitectura y estructura de las neuronas de la red.

Por lo tanto, el diseño de la red neuronal ha de ser orientado a su futura implementación dentro de un microcontrolador. Procurando cumplir las restricciones de memoria asociadas a la placa y restricciones temporales vinculadas a la aplicación. Con respecto a la memoria se dispone de la compresión de los pesos, que no se ha visto como un metodo efectivo para esta red en cuestión pero para otras con otra distribución de pesos pueden suponer una gran disminución de ocupación de memoria.

## Capítulo 5

# Conclusiones y líneas futuras

Cada uno de los dispositivos analizados destaca por ser ventajoso en el campo que el otro es más débil. Se ha observado que el desarrollo y la implementación en *FPGA* es más complejo y hay gran cantidad de parámetros que permiten variar el resultado, a pesar de utilizar un paquete como *hls4ml* que permite facilitarlo. La gran ventaja de su implementación son las bajas latencias gracias a la concurrencia, optimización del diseño y su desarrollo en *hardware*. Estos campos han sido el gran problema que se ha observado en los microprocesadores. Ahí se ha comprobado que la latencia de la red aumentaba bastante según esta crecía y era mucho mayor que en la *FPGA*. En concreto, el tiempo en el micro ha sido de 120  $\mu\text{s}$  en el caso de la red más pequeña y en la *FPGA* ha sido de 5.5  $\mu\text{s}$  la más grande.

Sin embargo, el desarrollo ha sido mucho más sencillo debido a lo guiado que era el paquete de desarrollo y la API. Por otra parte, el número de grados de libertad a la hora de implementar la red neuronal es bastante reducido en esta plataforma, se dispone únicamente de la compresión de los parámetros para tener una menor ocupación de memoria. Sin embargo los grados de libertad de la *FPGA* prestan especial atención a la cuantificación y al *trade-off* del factor de reuso, que marcará la utilización y latencia del sistema.

Otro aspecto es la cantidad de información accesible por parte de cada paquete. *STM* dispone de una gran cantidad de documentos y vídeos explicativos al descargar el paquete que permiten comprender la totalidad del mismo. Por otra parte *hls4ml* dispone únicamente de su página web y *Github* [19], siendo más difícil descubrir muchas de las funcionalidades del conjunto de funciones.

Además, hay una diferencia en costes entre las dos plataformas también evidente. A nivel de tarjeta la *Zedboard* cuesta entorno a 499\$ [20] en la web de Digilent, mientras que la *NUCLEO-L496ZG-P* se encuentra en 19.60\$ [21] en la web de STMicroelectronics. Si se atiende ahora al precio de cada chip, el *SoC* Zynq 7020 tiene un precio unitario en AVNET de 179.63\$ [22] mientras que el microprocesador cuesta 10.57\$ [23]. Ambos evaluados para la misma cantidad de dispositivos comprados.

De este trabajo pueden surgir una gran cantidad de trabajos futuros dado que se limita a evaluar las herramientas para la implementación en los dos tipos de dispositivos, por lo que cualquier red neuronal de cualquier temática puede ser implementada empleando los paquetes utilizados. Sin embargo, en este trabajo han quedado los siguientes temas que se han podido probar:

- Solucionar los problemas de cuantificación en *FPGA* para alcanzar la misma precisión que la teórica. Se puede optimizar utilizando la herramienta de *hls4ml* o utilizar paquetes de cuantificación como *QKeras* [24].

- Comparar los resultados con el IP de Vivado encargado de implementar capas convolucionales, pooling o capas densas llamado *DPU for Convolutional Neural Network* [25].
- Comparar con otros lenguajes de alto nivel sintetizados como puede ser *HDL Coder* de Matlab.
- Evaluar otras herramientas de implementación de redes neuronales para otros microprocesadores.
- Como se describió en el capítulo del estudio teórico se dividían las técnicas en dos aspectos, las de bajo nivel consistentes en la adquisición, preprocesado y detección de eventos y una de alto nivel orientada a la clasificación. Con lo visto en el trabajo el apartado de clasificación ha quedado cerrado pero no el de la etapa de bajo nivel que se propone como línea futura. También la representación de la salida de la red en un interfaz para el usuario.

## Capítulo 6

# Presupuesto

En este apartado se definirán los costes de cada material atendiendo al precio actual si sigue el producto en existencias y si no al precio de compra. Si se requiere un cambio de dolar a euro se realizará atendiendo al cambio a fecha 02/07/2022 de 0.958 dolares por euro.

Tabla 6.1: Costes Materiales

Objeto	Precio Unitario(€)	Unidades	Precio Total (€)
Omen 015 i7	1 300	1	1 300
Tarjeta de Evaluación Zedboard	478.42	1	478.42
Tarjeta de Evaluación Nucleo-L496ZGP	22.72	1	22.72
Licencia Matlab HOME	119	1	119
Licencia Office 365 Personal	69.99	1	69.99
Vivado	0	1	0
STM32CubeIDE	0	1	0
Pycharm Community Edition	0	1	0
TexMaker	0	1	0
Total			1990.13

Para extraer los costes profesionales de la actividad realizada se extrae de [26] el salario medio de un ingeniero de telecomunicaciones recién salido de la carrera en 20.600 €/año. Puesto que el artículo no especifica si el salario es para un titulado de grado o de master, se considera que se trata de uno de grado y se aumenta el salario medio a 24.000€/año. Con respecto al tiempo trabajado, se considera el semestre completo. Por lo tanto los gastos profesionales se reflejan en la siguiente tabla:

Tabla 6.2: Costes Profesionales

Actividad	Precio(€/mes)	Tiempo(meses)	Precio Total (€)
Ingenieria	2 000	6	12 000

Finalmente los costes totales del proyecto se muestran en la siguiente tabla aplicando unos gastos generales del 10% y unos beneficios industriales del 5%:

Tabla 6.3: Costes Totales

Costes Materiales	1990.13€
Costes Profesionales	12 000€
Costes Totales	13 990.13€
Gastos Generales(10 %)	1399.01€
Beneficio industrial (5 %)	699,50€
Presupuesto total	16088,64€

# Bibliografía

- [1] Mike A, “Perceptron: qué es y para qué sirve,” <https://datascientest.com/es/perceptron-que-es-y-para-que-sirve>, accessed on 2022-06-25.
- [2] Z. Lu and K.-C. Chou, “iATC\_DeepmISF: A MultiLabel Classifier for Predicting the Classes of Anatomical Therapeutic Chemicals by Deep Learning,” *Advances in BioScience and BioTechnology*, no. 11, pp. 153–159, May 2020.
- [3] Laura de Diego, “Definition of NILM techniques for energy disaggregation in AIIL environments,” *Trabajo Final de Grado, Universidad de Alcalá*, 2020.
- [4] Digilent, “Zedboard,” <https://digilent.com/reference/programmable-logic/zedboard/start>, accessed on 2022-03-05.
- [5] Jennifer Ngadiuba and Javier Duarte, “Overview of hls4ml project,” [https://indico.cern.ch/event/855626/contributions/3600304/attachments/1927281/3190876/hls4ml\\_BGU\\_meeting\\_16.10.2019.pdf](https://indico.cern.ch/event/855626/contributions/3600304/attachments/1927281/3190876/hls4ml_BGU_meeting_16.10.2019.pdf), accessed on 2022-03-05.
- [6] AMD Xilinx, “Xilinx Alveo,” <https://www.xilinx.com/products/boards-and-kits/alveo.html>, accessed on 2022-06-26.
- [7] AWS, “Amazon EC2 F1 Instances,” [https://aws.amazon.com/ec2/instance-types/f1/?nc1=h\\_ls](https://aws.amazon.com/ec2/instance-types/f1/?nc1=h_ls), accessed on 2022-06-26.
- [8] S. Saha, “A Comprehensive Guide to Convolutional Neural Networks the ELI5 way,” <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>, accessed on 2022-06-26.
- [9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>
- [10] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” 2015. [Online]. Available: <https://arxiv.org/abs/1506.02640>
- [11] A. Sherstinsky, “Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network,” *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167278919305974>
- [12] J. McGonagle, G. Shaikouski, and C. Williams, “Backpropagation,” <https://brilliant.org/wiki/backpropagation/#:~:text=Backpropagation%2C%20short%20for%20%22backward%20propagation,to%20the%20neural%20network's%20weights.>, accessed on 2022-06-26.

- 
- [13] Endesa, “The smart meter: the new digital smart meter,” <https://www.endesa.com/en/the-e-face/electrical-grid/smart-meter>, accessed on 2022-07-12.
- [14] Duchstf, “Keras to HLS: supported layers,” <https://github.com/fastmachinelearning/models/blob/master/keras/README.md>, accessed on 2022-03-05.
- [15] Jupyter, “Project jupyter,” <https://jupyter.org/>, accessed on 2022-03-05.
- [16] STMicroelectronics, “Nucleo-L496ZG-P,” <https://www.st.com/en/evaluation-tools/nucleo-l496zg-p.html>, accessed on 2022-07-03.
- [17] STMicroelectronics, “STM32CubeIDE,” <https://www.st.com/en/development-tools/stm32cubeide.html>, accessed on 2022-07-03.
- [18] STMicroelectronics, “X-Cube-AI,” <https://www.st.com/en/embedded-software/x-cube-ai.html>, accessed on 2022-07-03.
- [19] CERN, “Github HLS4ML,” <https://github.com/fastmachinelearning/hls4ml>, accessed on 2022-03-05.
- [20] Digilent, “ZedBoard Zynq-7000 ARM/FPGA SoC Development Board,” <https://digilent.com/shop/zedboard-zynq-7000-arm-fpga-soc-development-board/>, accessed on 2022-06-29.
- [21] Tienda de STMicroelectronics, “NUCLEO-L496ZG-P,” <https://estore.st.com/en/nucleo-l496zg-p-cpn.html>, accessed on 2022-06-29.
- [22] AVNET, distribuidor de Xilinx, “XC7Z020-L1CLG484I,” <https://www.avnet.com/shop/us/products/amd-xilinx/xc7z020-l1clg484i-3074457345626108626/>, accessed on 2022-06-29.
- [23] AVNET, distribuidor de Xilinx, “STM32L496ZGT6P,” <https://www.avnet.com/shop/us/products/amd-xilinx/xc7z020-l1clg484i-3074457345626108626/>, accessed on 2022-06-29.
- [24] QKeras, “Github of QKeras,” <https://github.com/google/qkeras>, accessed on 2022-06-29.
- [25] Xilinx, “DPU for Convolutional Neural Network,” <https://www.xilinx.com/products/intellectual-property/dpu.html>, accessed on 2022-06-29.
- [26] Jobted, “Ingeniero de telecomunicaciones-sueldo medio,” <https://www.jobted.es/salario/ingeniero-telecomunicaciones>, accessed on 2022-07-03.





Universidad de Alcalá  
Escuela Politécnica Superior



ESCUELA POLITECNICA  
SUPERIOR



Universidad  
de Alcalá