Trabajo Fin de Máster
"Máster Universitario en Microelectrónica: Diseño y Aplicaciones de Sistemas Micro/Nanométricos"

Diseño de una red neuronal oscilatoria digital con capacidad de aprendizaje on-line sobre FPGA

Design of an oscillatory neural network with on-line Learning on an FPGA

Autor / Author: Daniel Vázquez Díaz

Tutor / Advisor: María José Avedillo de Juan
Manuel Jiménez Través
Juan Núñez Martínez

Fecha / Date: Julio 2023

Agradecimientos

Mis agradecimientos a María José Avedillo de Juan, Manuel Jiménez Través y Juan Núñez Martínez por la ayuda guía y apoyo brindado durante la realización de este trabajo fin de Máster.
También agradecer a mi mujer y mis hijos por su paciencia y ánimos durante todo este tiempo para terminar este trabajo Fin de Master.

Resumen

La inteligencia artificial es un concepto que cada vez está más integrado en nuestras vidas. Aunque no nos demos cuenta, esta nos rodea en nuestro día a día. Ejemplos de esto son los algoritmos que utilizan los buscadores de internet, los cuales permiten ofrecer una publicidad que se adaptada en función de la información recopilada durante las búsquedas o los algoritmos de reconocimiento de señales de tráfico que se están implementando en los vehículos actuales.

En general, estos algoritmos de inteligencia artificial están implementados sobre plataformas basadas en CPU y/o GPU, que tienen la desventaja de su gran consumo de potencia, que los hace inviables en aplicaciones de bajo consumo. Una alternativa es la implementación en hardware de estas redes neuronales, con frecuencia usando dispositivos y paradigmas de computación no convencionales. Entre estas últimas, se están estudiando redes neuronales oscilatorias (ONNs) con potencial para una computación eficiente energéticamente.

En este trabajo fin de master se ha desarrollado el estudio e implementación hardware de un algoritmo de aprendizaje iterativo supervisado para ONNs. Este trata de mejorar las prestaciones de los algoritmos de aprendizaje simples, como la regla de Hebb.

El punto de partida para el desarrollo de este trabajo ha sido una descripción Verilog a nivel RTL, sintetizable y parametrizable de una ONN digital. Las principales tareas llevadas a cabo han sido:

- Análisis del funcionamiento de la ONN digital mediante simulación del código proporcionado.
- Desarrollo y validación del nuevo algoritmo utilizando lenguajes de alto nivel (MATLAB).
- Diseño a nivel RTL y validación de una implementación hardware del algoritmo. Se ha generado una descripción Verilog sintetizable y parametrizable para este hardware. En el diseño se ha utilizado el entorno Vivado.
- Desarrollo de un sistema para su demostración experimental. Este usa una placa de desarrollo Zybo-7. La ONN y los módulos hardware que realizan el aprendizaje *on-line* se implementan en la lógica programable del APSoC Zynq del que dispone dicha placa. En su procesador corre una aplicación que permite conectar su puerto serie con el diseño hardware. Una aplicación MATLAB envía ordenes de aprendizaje o de inferencia a la ONN desde un PC.

El algoritmo desarrollado mejora las prestaciones de los algoritmos de aprendizaje simples, como la regla de Hebb y se ha demostrado la operación correcta del sistema experimental.

Palabras clave: FPGA, Zynq, Vivado, Vitis, MATLAB, AXI, ONN, Aprendizaje supervisado.

Índice de Contenidos

I. Índice de Figuras

II. Índice de Tablas

III. Glosario

AM	Associative Memory
APSoC	All Programmable System-On-Chip
ARM	Advanced RISC Machine
AXI	Advanced eXtensible Interface
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
GPU	Graphic Processing Unit
HW	Hardware
LUT	Look Up Table
ONN	Oscillatory Neural Network
PC	Personal Computer
PL	Programmable Logic
PS	Processing System
RTL	Register Transfer Level
SW	Software
USB	Universal Serial Bus
VO_2	Dióxido de vanadio

1. Introducción

Cada vez está más presente el uso de la inteligencia artificial como tecnología aplicada en nuestras vidas. Ejemplos de esto son el reconocimiento facial aplicado por las cámaras de los teléfonos móviles, los sistemas de reconocimiento de señales de tráfico utilizados en los vehículos actuales o el reconocimiento de voz.

Esta inteligencia artificial está basada en el uso de redes neuronales. Estas generalmente, están implementados en dispositivos como CPUs o GPUs, dispositivos que ofrecen una gran potencia de cálculo, pero también requieren de gran consumo para su funcionamiento, lo cual limita su uso en aplicaciones de bajo consumo.

Una alternativa al uso de CPUs o GPUs es la implementación en hardware de estas redes neuronales. El desarrollo de este hardware neuromórfico es un área de gran interés y actividad de investigación en el momento actual. Comprende aproximaciones muy diversas, incluyendo implementaciones digitales y analógicas. En estas últimas, el uso de dispositivos y paradigmas de computación no convencionales es muy prometedor.

Este TFM se enmarca en el contexto del proyecto europeo NeurONN, en el cual se explora la realización de redes neuronales usando redes de osciladores acoplados (paradigma de computación no convencional). Estas redes se denominan redes neuronales oscilatorias o ONNs del inglés *Oscillatory Neural Network*. Además, estos osciladores se implementan con dispositivos de transición de fases (dispositivos alternativos al transistor).

Las ONNs tiene potencial desde el punto de vista de su eficiencia energética. Por una parte, su computación se basa en la dinámica de osciladores acoplados, codificando y procesando la información en su relación de fases, por lo tanto, de forma completamente paralela. Esto se traduce en tiempos de cómputos reducidos. Por otro lado, los osciladores utilizados también presentan ventajas en términos de área y potencia respecto a otros tipos de osciladores. Todo ello se traduce en menor consumo de energía, lo que permite que se puedan utilizar en dispositivos de bajo consumo. Esto daría lugar al desarrollo de dispositivos que sean capaces de proporcionarles una inteligencia, como puede ser una cámara de bajo consumo capaz de proporcionar información sobre los objetos detectados.

1.1. Objetivos y Metodología

El objetivo de este trabajo es el diseño, realización y validación de una implementación hardware de un algoritmo de aprendizaje supervisado para ONNs.
En concreto, se persigue dotar de capacidad de aprendizaje on-line a una ONN digital desarrollada en el proyecto como prueba de concepto. El punto de partida para el desarrollo de este trabajo fue una descripción Verilog a nivel RTL, sintetizable y parametrizable de la ONN digital.
Las principales tareas llevadas a cabo han sido:

- Análisis del funcionamiento de la ONN digital mediante simulación del código proporcionado.
- Análisis del algoritmo de aprendizaje en el que se basa el desarrollado mediante modelos en lenguajes de programación de alto nivel.
- Desarrollo y validación del nuevo algoritmo utilizando también lenguajes de alto nivel.
- Diseño a nivel RTL y validación de una implementación hardware del algoritmo. Se ha generado una descripción Verilog sintetizable y parametrizable para este hardware.
- Desarrollo de un sistema para su demostración experimental.

1.2. Herramientas y recursos materiales

MATLAB

MATLAB [1] es un software de cómputo numérico con un entorno gráfico y un lenguaje propio (lenguaje M). MATLAB permite implementar de manera sencilla y rápida algoritmos y poder validarlos. También proporciona herramientas que permiten visualizar gráficamente los resultados obtenidos.

En este proyecto se ha utilizado MATLAB para la evaluación del modelo de aprendizaje, antes de implementarlo en HW y en la demostración experimental.

Vivado

Vivado [2] es el entorno de desarrollo del fabricante de FPGAs Xilinx. Este permite la síntesis, simulación, implementación y análisis de sistemas digitales.

En este trabajo se ha utilizado Vivado en el análisis de la ONN digital y en el diseño, simulación y síntesis de la implementación hardware del algoritmo de aprendizaje.

Vitis

Vitis [3] es el entorno de desarrollo software de Xilinx que se utiliza para la implementación de aplicaciones que corren en los procesadores disponibles en los dispositivos como FPGAs y APSoC de Xilinx.

En este proyecto se ha utilizado Vitis para la implementación de un software que permite leer e interpretar los comandos recibidos a través del puerto serie de la plataforma Hardware utilizada durante la demostración.

Tarjeta Zybo.

La placa de desarrollo Zybo Z7 [4] [5]es una tarjeta que dispone de una FPGA Zynq-7020 de Xilinx. Las características principales de esta tarjeta son:

- Dispone de un dispositivo Zynq- XC7Z020-1CLG400C
- 1 GB de DDR3L.
- Puertos USBs, Ethernet, video y audio.
- Conectores PMOD para usar módulos externos dedicados, leds y switches.

El dispositivo Zynq es un System On Chip (SoC) que está formado por un procesador Cortex A9 dual core embebido (PS) y una sección lógica programable (PL). Esta arquitectura permite explotar las ventajas de tener en un mismo dispositivo un procesador y la flexibilidad de la lógica programable. La intercomunicación entre ambas partes, PS y PL, a través del bus específico como el AXI (Advanced eXtensible Interface) hacen de la Zynq un dispositivo idóneo para este tipo de implementaciones.

1.3. Estructura de la Memoria

La memoria se estructura de la siguiente manera:

- En el capítulo 2 se introducen los fundamentos en los que se basa el concepto de las redes neuronales oscilatorias como son los osciladores y las redes de Hopfield.
- En el capítulo 3 se explica el funcionamiento de la ONN digital a la que se dota de capacidad de aprendizaje on-line. También se exponen los algoritmos de aprendizaje Hebbian y Hebbian iterativo.
- En el capítulo 4 se describe el proceso de implementación del algoritmo de aprendizaje supervisado completo, desde su desarrollo previo en MATLAB, hasta la simulación de su descripción Verilog.
- En el capítulo 5 se dedica a la validación experimental de la ONN con capacidad de aprendizaje on-line. Se describe el sistema desarrollado sobre la plataforma hardware Zybo Z7 basada en la FPGA Zynq-7020 de Xilinx y la aplicación MATLAB con la que se controla y se ilustra su operación.
- En el capítulo 6 se exponen las conclusiones obtenidas durante la elaboración de este TFM.
- La memoria finaliza con la sección de bibliografía y los anexos donde se incluyen los códigos y scripts que han sido desarrollados.

2. Fundamentos

Las ONN (Oscillatory Neural Network o Redes Neuronales Oscilatorias) son redes neuronales analógicas formadas por elementos oscilatorios acoplados entre sí [6], [7], [8], [9]. El acoplamiento entre los osciladores se realiza mediante elementos resistivos o capacitivos y el conjunto de los valores de estos elementos acopladores son lo que proporcionan la capacidad a la red neuronal de almacenar patrones [10].

En esta sección se introduce brevemente los osciladores y las redes de Hopfield, que son los conceptos en los que se basa el funcionamiento de las ONN. Finalmente, se explican los algoritmos de aprendizaje de Hebbian y el método de aprendizaje iterativo.

2.1. Redes neuronales oscilatorias

El elemento básico de las redes neuronales ONN son los osciladores. Estos elementos permiten generar señales repetitivas periódicas. Existen diversas formas de implementar un oscilador, entre las que se encuentran las siguientes:

- Microelectromecánicos (MEMS).
- Spin-Torque Oscillators (STO).
- Osciladores basados en PLL (Phased Locked Loops).
- Osciladores de anillo (RO).
- Empleando Materiales de Transición de Fase (Phase Transition Materials, PTM).

En NeurONN se utiliza el dióxido de vanadio (VO_2), un material que exhibe transiciones de fase, para implementar un oscilador muy compacto.

Oscilador basado en VO_2

La topología más básica de este tipo de oscilador se muestra en la Figura 1(b) [11], [12], [13]. Está formado por un elemento principal, que es el dióxido de vanadio (VO_2), una resistencia y un condensador.

El VO_2 es un material de cambio de fase que posee dos zonas o fases de trabajo, una fase aislante o de alta resistencia y una fase metálica o de baja resistencia. La Figura 1(a) muestra su característica I-V. En estado de reposo, en ausencia de estímulo externo, permanece en fase aislante.

Al someterse el VO_2 a una diferencia de tensión, la corriente que circula a través de este aumenta, haciendo que se produzca la transición de aislante a metal (IMT por sus siglas en inglés). En el estado metálico, al reducir la tensión del VO_2, se disminuye la corriente. Cuando cae por debajo de J_{C-MIT} se produce una transición del estado metálico al estado aislante (MIT).

El funcionamiento de los osciladores basados en VO_2 consiste en utilizar este cambio de fase para generar una señal oscilatoria. En la Figura 1(c) se ilustra su operación. Con el VO_2 en metálico, la capacidad de salida se carga (punto B en la forma de onda del oscilador). Esto reduce la tensión que cae en el VO_2 hasta que alcanza V_{MIT} y se produce la transición al estado aislante. En este estado, la capacidad se descarga por la resistencia (punto A), incrementando la tensión en el VO_2. Cuando esta alcanza V_{IMT} se produce la transición a metálico y el proceso se repite.

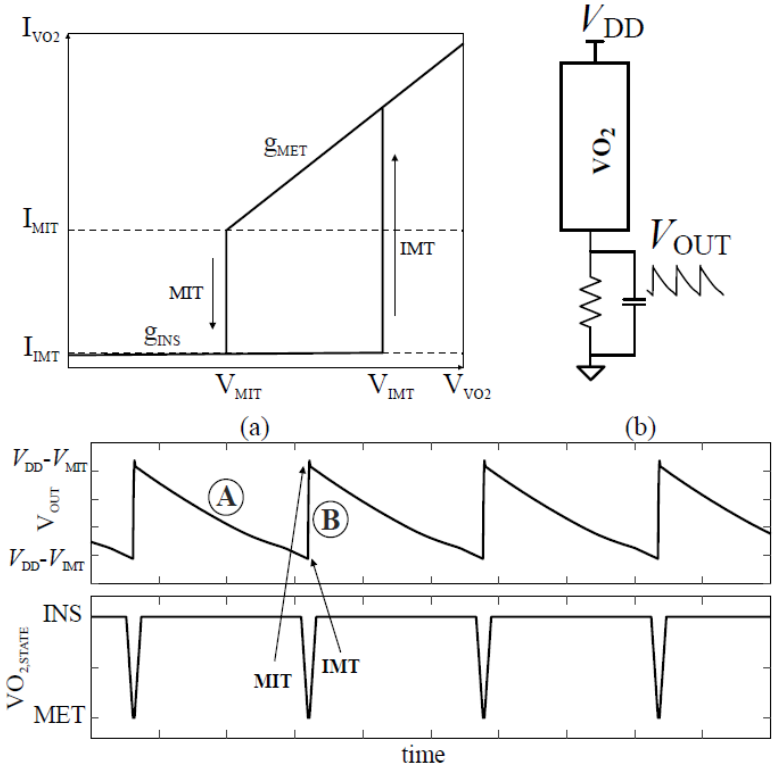


Figura 1: Funcionamiento oscilador VO_2 [14].

Funcionamiento de las ONN

Las ONN son redes neuronales oscilatorias, formadas por osciladores acoplados entre sí mediante resistencias y/o condensadores que representan los pesos sinápticos de las redes neuronales. Cuando dos osciladores de frecuencias similares se acoplan capacitivamente, se sincronizan y se bloquean en anti-fase. Si se acoplan resistivamente, se sincronizan en fase. En una ONN, una vez que se produce la sincronización, se tiene una determinada relación de fases entre los distintos osciladores que la componen que constituye el estado de la ONN. Esto es, en una ONN la información se codifica en la fase de señales oscilatorias. La relación de fase inicial, que se corresponde con el patrón de entrada de la red, se controla retrasando adecuadamente el encendido (activación de la polarización) de unos osciladores respecto a otros. Si se utiliza el ejemplo más simple, que consiste en una entrada binaria, el 0 estaría representado con una inicialización del oscilador en el instante T0 y el 1 estaría representado por una señal aplicada en el instante T0 + *T_OSC*/2, siendo *T_OSC* el periodo de oscilación.

Los acoplamientos se eligen de forma que se mapea la solución del problema que se quiere resolver con la red a la función de energía de dicho sistema dinámico. En consecuencia, a medida que el sistema evoluciona para minimizar su energía, naturalmente calculará la solución del problema. La manifestación y eficacia de este enfoque es evidente en el mundo físico; El procesamiento de la información en tiempo real se observa en los sistemas dinámicos naturales, como los patrones de activación de los circuitos neuronales o los mecanismos de señalización celular. Así, el sistema físico "computa" con su propia dinámica de forma paralela colectiva y, por tanto, con tiempos de cómputo competitivos, lo que también se traduce en un menor consumo de energía.

2.2. Redes de Hopfield

Las redes neuronales de Hopfield o HNN por sus siglas en inglés (Hopfield Neural Networks) son un tipo de redes neuronales recurrente en las que cada neurona están conectadas entre sí mediante unos pesos W_{ij}. Estos pesos indican la fuerza de conexión que hay entre las diferentes neuronas y los subíndices indican entre que dos neuronas se aplica tal peso, es decir, entre la neurona i y la j.

Este tipo de redes se han utilizado como memorias asociativas (AM) y para tareas de reconocimiento de patrones. La Figura 2 representa una red neuronal de Hopfield [15]. Cada neurona es representada por n_i y los pesos entre ellas se representan por W_{ij}. Con frecuencia se utilizan matrices de pesos simétricas ($W_{ij} = W_{ji}$) y con diagonal nula ($W_{ij} = 0 \ \forall \ i = j$). La Figura 3 representa una HNN de tres neuronas con estas restricciones de forma que se enfatiza su naturaleza recurrente.

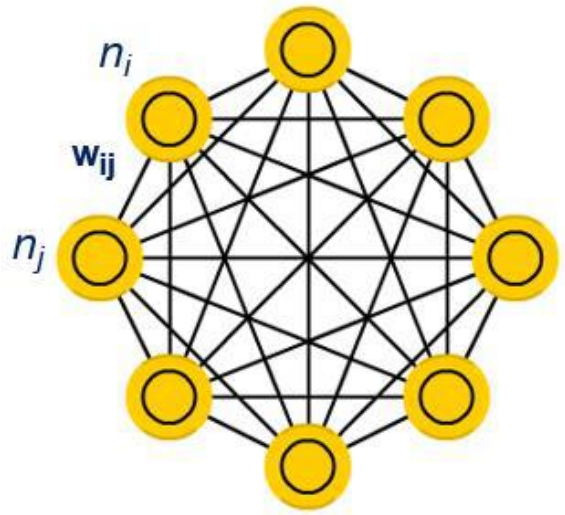


Figura 2: Modelo de Red Neuronal HNN (Hopfield Neural Network)

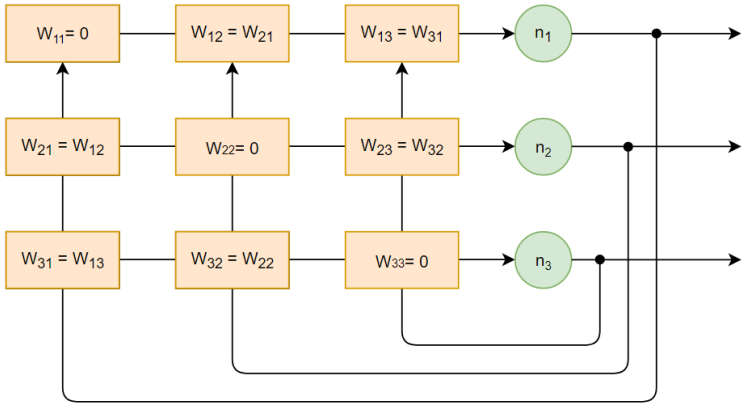


Figura 3: Red Neuronal de tres neuronas

Una representación simplificada y más próxima a las ONNs en las que las interacciones son bidireccionales se muestra en la Figura 4.

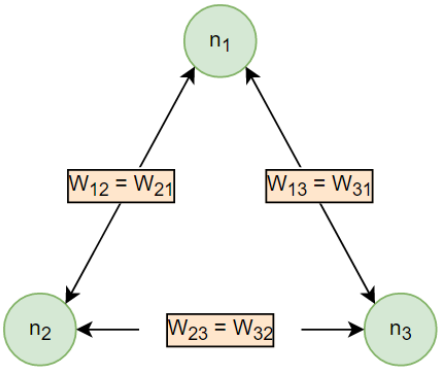


Figura 4: Red Neuronal de tres neuronas simplificada

En una red de Hopfield discreta, el estado de cada neurona se representa mediante S_i y este toma valores bipolares {-1, +1}.

La actualización del valor S_i correspondiente a cada neurona se realiza de acuerdo con:

$$S_{\mathrm{i}} = +1 \quad \text{Si} \quad \sum_{j \neq i} W_{ij} S_j > U_i$$

$$\mathrm{S_i} = -1 \quad \text{Si} \quad \sum_{j \neq i} W_{ij} S_j < U_i$$

Donde U_i es un valor umbral para cada una de las neuronas. En el caso más simple, este umbral es el mismo para todas las neuronas y se establece en 0. Por tanto, se puede resumir que la ecuación para actualizar el estado de cada neurona es:

$$S_i = Sign\left(\sum_j W_{ij} S_j\right)$$

Los pesos determinan cuáles son los estados estables de la red (puntos fijos de la expresión anterior de cada una de las neuronas). Los estados estables se corresponden con estados de mínima energía [15]:

$$E = -\frac{1}{2}\sum_i \sum_j W_{ij} S_i S_j$$

La HNN se comporta como una memoria asociativa si los pesos se fijan de forma que los patrones que se quieren guardar o almacenar se correspondan con estados estables de la misma. Se denomina aprendizaje o entrenamiento a la determinación de dichos pesos.

Cuando se aplica un patrón a la entrada de la red neuronal, esta converge al estado almacenado más cercano, ya que tiende a estabilizarse en un punto de mínima energía.

Si imaginamos una superficie curva con diferentes puntos de mínima energía, representados por X_1, X_2 y X_3 en la Figura 5, estos puntos representan los patrones almacenados por la red neuronal.

El patrón de entrada está representado por la bola verde y esta tendería al punto de mínima energía más próximo, en este caso X_1

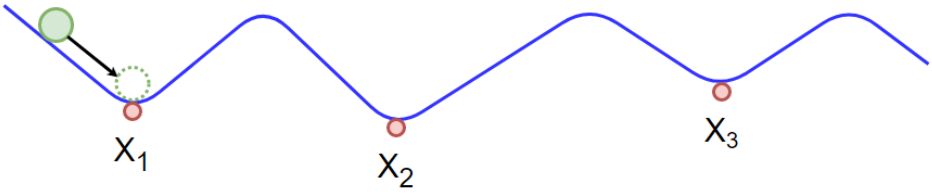


Figura 5: Redes de Hopfield. Punto de mínima energía

La AM puede aplicarse al reconocimiento de patrones almacenando los patrones que se quieren reconocer en la AM. Al aplicar una versión distorsionada de estos se recupera el más parecido.

2.3. Métodos de aprendizaje

2.3.1. Algoritmo de aprendizaje Hebbian

La regla de aprendizaje de Hebbian es uno de los algoritmos más conocidos para el cálculo de los pesos sinápticos en redes neuronales bipolares. Este fue propuesto por el neuropsicólogo Donald O. Hebb, quien postuló la forma en la que aprenden las neuronas [16]:

Cuando el axón de una célula A esta lo suficientemente cerca como para excitar a otra célula B y repetida o persistentemente toma parte en su activación, se produce un proceso o cambio metabólico en una o ambas células de tal manera que tanto eficiencia de A, como la de las células que activan a la célula B, se incrementan.

Esto quiere decir que, los pesos sinápticos o la conexión entre las neuronas se ven fortalecidas como respuesta a la experiencia, mientras que, en el caso contrario, los pesos sinápticos o la conexión se ve debilitada.

Esta regla permite entrenar una red HNN para guardar un determinado conjunto de patrones. Estos patrones se denominan patrones de entrenamiento. En este caso su significado es que, si las neuronas se activan juntas, la fuerza de su acoplamiento mutuo aumentará, mientras que de lo contrario la conexión será más débil.

Este tipo de algoritmo de aprendizaje es sin supervisión, lo que quiere decir que solo depende de los patrones de entrenamiento. Una de las limitaciones de este algoritmo es que es inestable, debido a que los pesos tienden a crecer sin límite, lo que da lleva a la saturación y por tanto da lugar la pérdida de su utilidad

Así, para un conjunto de P patrones de longitud N, $\xi^K \epsilon\{-1,+1\}^N$ que se desea almacenar en una HNN, la fuerza de acoplamiento entre las neuronas (peso) se determina mediante la siguiente ecuación:

$$W_{ij} = \sum_{K=1}^{P} \xi_i^K \xi_j^K$$

Donde:

- $W_{ij} = 0 \; \forall$ i = j
- N es el número de neuronas de la red neuronal
- ξ_i^K representa el estado de la neurona i en el patrón K.

La capacidad que ofrece este tipo de algoritmo viene determinada por la siguiente ecuación [17]:

$$C_{abs} = \frac{N}{2\ln(N)'}$$

La regla de Hebb admite una formulación incremental:

$$W_{ij}(n+1) = W_{ij}(n) + \Delta W_{ij}$$

Donde el incremento viene definido por la siguiente ecuación:

$$\Delta W_{ij} = \alpha x_i * x_j$$

Donde x_i , x_j corresponde a los valores de las posiciones *i* y *j* del nuevo patrón. α es un escalar que indica la ratio de aprendizaje, que se considera 1. Por tanto, la ecuación de actualización de la matriz de pesos sinápticos según el algoritmo de Hebbian queda la siguiente manera:

$$W(n+1) = W(n) + x^T * x$$

El algoritmo de Hebbian se pude resumir en los siguientes pasos:

1. Se inicializan todos los pesos de la matriz de pesos a 0. $W_{ij} = 0$
2. Repetir el paso 3 para cada nuevo patrón de entrada que se desea aprender.
3. Actualizar los pesos conforme a la ecuación $W(n+1) = W(n) + x^T * x$
4. Se ponen a cero los elementos de la diagonal de la matriz W.

Se ha utilizado un ejemplo sencillo para explicar el algoritmo de Hebbian. Por ejemplo, para un caso concreto de una red neuronal formada por 5 neuronas, el vector de entrada será de tamaño 5 y por tanto los patrones de aprendizaje utilizados serán vectores de 5 bits. El vector de salida también será de 5 bits.

En un caso concreto, si se desea aprender el siguiente patrón:

(+1, -1, +1, +1, -1)

El cálculo quedaría de la siguiente manera:

$$\begin{bmatrix} W_{11} & W_{12} & W_{13} & W_{14} & W_{15} \\ W_{21} & W_{22} & W_{23} & W_{24} & W_{25} \\ W_{31} & W_{32} & W_{33} & W_{34} & W_{35} \\ W_{41} & W_{42} & W_{34} & W_{44} & W_{45} \\ W_{51} & W_{52} & W_{35} & W_{54} & W_{55} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} +1 \\ -1 \\ +1 \\ +1 \\ -1 \end{bmatrix} * \begin{bmatrix} +1 & -1 & +1 & +1 & -1 \end{bmatrix}$$

La matriz quedaría, por tanto:

$$W = \begin{bmatrix} 0 & -1 & +1 & +1 & -1 \\ -1 & 0 & -1 & -1 & +1 \\ +1 & -1 & 0 & +1 & -1 \\ +1 & -1 & +1 & 0 & -1 \\ -1 & +1 & -1 & -1 & 0 \end{bmatrix}$$

Puede comprobarse que el patrón se ha almacenado, evaluando la salida de la red para dicho patrón de entrada:

$$y_i = Sign\left(\sum_j W_{ij} x_j\right)$$

Que es lo mismo que multiplicar el patrón de entrada por la matriz de pesos:

$$Y = Sign(\mathrm{H}) = Sign(X * W)$$

$$H = [+1 \quad -1 \quad +1 \quad +1 \quad -1] * \begin{bmatrix} 0 & -1 & +1 & +1 & -1 \\ -1 & 0 & -1 & -1 & +1 \\ +1 & -1 & 0 & +1 & -1 \\ +1 & -1 & +1 & 0 & -1 \\ -1 & +1 & -1 & -1 & 0 \end{bmatrix} = [+4 \quad -4 \quad +4 \quad +4 \quad -4]$$

Utilizando la función signo, se obtiene que el estado (salida) de la HNN es el propio patrón de entrada $[+1 \quad -1 \quad +1 \quad +1 \quad -1]$.

2.3.2. Algoritmo de aprendizaje iterativo

La regla de aprendizaje Hebbian es un algoritmo de aprendizaje no supervisado. Los pesos solo dependen de los patrones de entrenamiento, sin utilizar la respuesta de la HNN en su cálculo. A este tipo de aprendizaje se denomina sin supervisión.

Una variación de este tipo de aprendizaje consiste en introducir en el proceso la información de salida de la red neuronal. Este tipo de algoritmo de aprendizaje se denomina supervisado.

Partiendo del algoritmo de aprendizaje de Hebb, se propuso una versión supervisada [18], [19] que denominamos regla de Hebb iterativa. Las diferencias fundamentales son que antes de actualizar los pesos para un nuevo patrón de entrada, se evalúa si este es ya un punto fijo (está aprendido) con los pesos actuales y sólo se modifican los pesos asociados a las neuronas que no satisfacen esta condición. Además, el proceso se repite hasta que el conjunto de patrones de entrenamiento se almacena en la HNN.

El proceso de aprendizaje iterativo se resume en los siguientes pasos:

1. Se inicializa la matriz de pesos con todos sus elementos a 0.
2. Se establece un conjunto de patrones de entrenamiento.
3. La matriz de pesos se actualiza aplicando la regla de Hebb al primer patrón del conjunto.
4. Para cada uno de los patrones se evalúa la respuesta de la red neuronal. Si coincide con el propio patrón, este se ha aprendido y, por tanto, se vuelve al paso 4 para procesar otro patrón. En caso contrario, se aplica la regla de Hebb para dicho patrón antes de procesar el siguiente.
5. El procedimiento se repite hasta que todos los patrones se hayan aprendido.

3. Análisis de la ONN digital

Como ya se ha mencionado, este TFM parte de una ONN digital diseñada en el proyecto NeurONN como prueba de concepto [20]. El primer paso en nuestro trabajo fue el análisis de dicha ONN. Este paso es fundamental ya que, para abordar la tarea de dotarla de capacidad de aprendizaje on-line, era necesario disponer de una versión de la misma con pesos modificables. En este capítulo se describe este diseño y su funcionamiento.

3.1. Descripción general

La Figura 6 muestra un diagrama de bloques de alto nivel de la ONN digital que consta de los siguientes tres bloques:

- Neuronas "neurons", que contiene el conjunto de osciladores de la red (módulo "Neuron").
- Sinapsis "synapses", encargado de almacenar la matriz de pesos sinápticos y de actualizar los valores de entrada a red neuronal.
- Bloque de control "Control blocks", encargado de generar las señales necesarias para el correcto funcionamiento de la red neuronal.

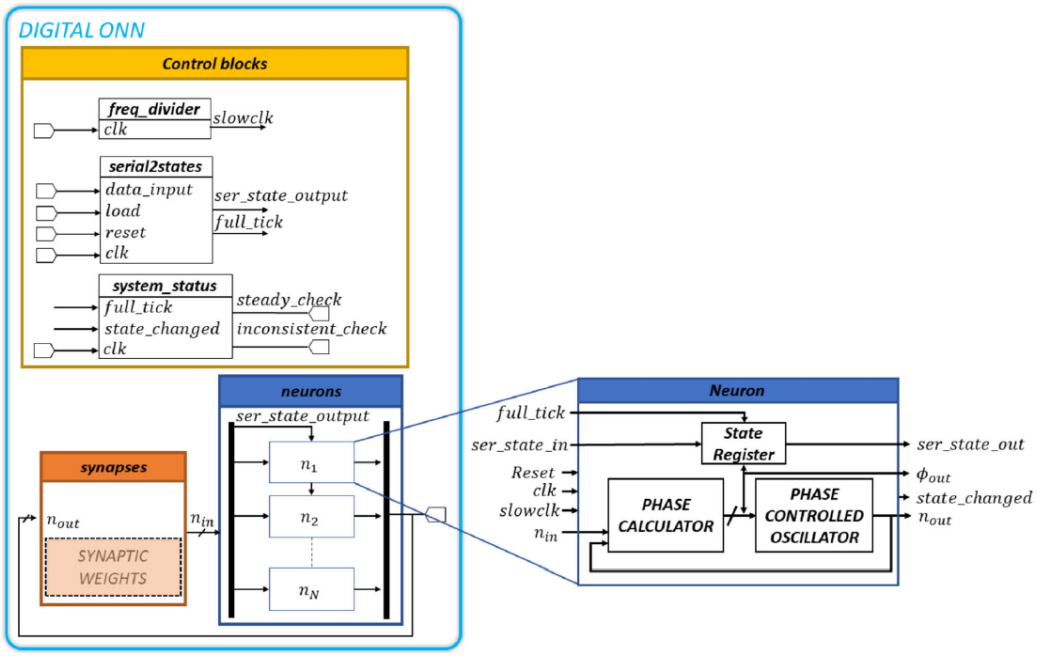


Figura 6: Diagrama de bloques de la ONN Digital

La descripción Verilog de partida está completamente parametrizada y es sintetizable.

A continuación, se explica en detalle el funcionamiento de cada uno de los módulos que componen la red neuronal. Además, se han incorporado algunas simulaciones de cada módulo con el objetivo de proporcionar un mejor entendimiento de su comportamiento.

Para las simulaciones se ha utilizado la herramienta Vivado del fabricante Xilinx. Esta herramienta permite sintetizar, implementar y simular diseños realizados en lenguajes de descripción de hardware, tales como VHDL o Verilog.

Se han desarrollado diferentes *testbenchs* que permiten estimular las entradas de los distintos módulos.

En las simulaciones se han utilizado los siguientes valores para los parámetros de la ONN:

- **Nrow = 5**. Este parámetro indica el número de filas de la red de neuronas.
- **Ncol = 8**. Indica el número de columnas que forma la red de neuronas.
- **NbitsState= 4**. Indica el número de bits usados para codificar el estado de cada neurona.
- **NbitsWeights = 5**. Indica el número de bits a utilizar para codificar los pesos de la matriz sináptica.

A partir de estos parámetros, se obtiene que el número total de neuronas que forman la ONN. Este es de *Nrow*Ncol* o 40 neuronas. La distribución de las neuronas de la red en filas y columnas se debe a la aplicación con la que se trabaja, con patrones bidimensionales correspondientes a imágenes. El significado de los restantes parámetros se introduce al describir el módulo hardware que los utiliza.

3.1.1. Módulo "Neurons"

El bloque "Neurons" es el que contiene los osciladores que proporcionan la salida de la red neuronal en función de la entrada.

Cada módulo "Neuron" (oscilador) detecta la diferencia de fase entre su señal de entrada y su salida y las alinea. La Figura 7 muestra un diagrama de la conexión entre los pesos sinápticos (bloque naranja) y las neuronas (bloque en azul).

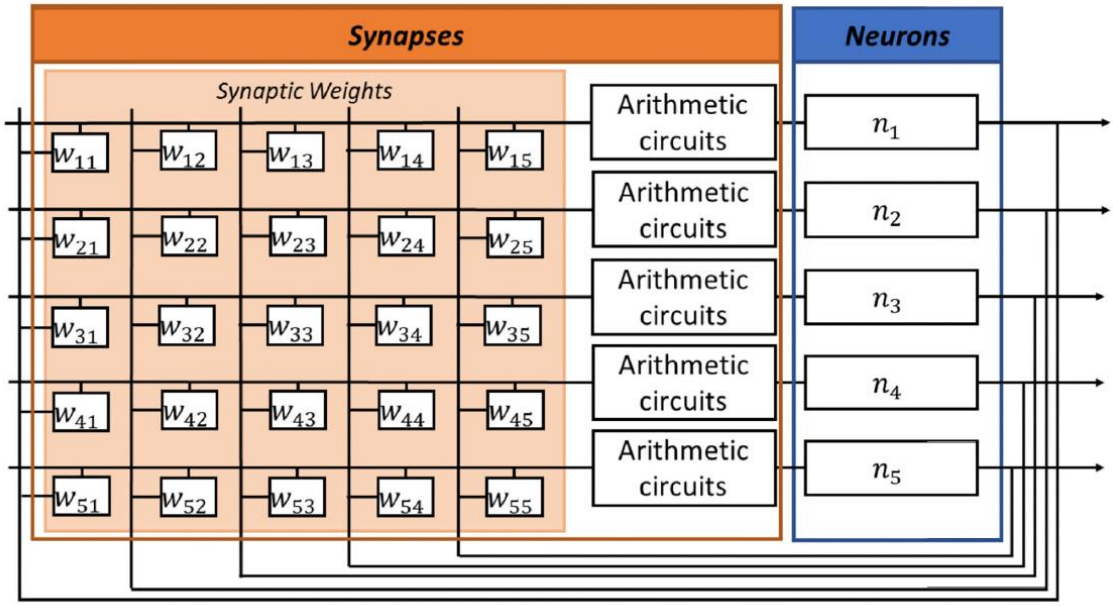


Figura 7: Conexión pesos sinápticos, aritmética y neuronas

Los puertos del módulo "**Neuron**" son los siguientes:

- **Ninput**: Señal de entrada que proviene de la operación de los pesos sinápticos por las salidas del resto de neuronas.
- **full_tick:** Señal de entrada que viene del módulo "serial2states". Se utiliza para inicializar las neuronas.
- **clk:** Señal de entrada de reloj.
- **slowclk:** Señal de entrada de reloj dividido derivado de "clk" y utilizado como fuente para el registro de desplazamiento con el que se genera la oscilación.
- **reset_async:** Señal de entrada de reset asíncrono.

- **bit_initstate:** Señal de entrada que proviene de la salida del módulo "serial2states" en el caso de la primera neurona y que almacena la información asociada al estado de las neuronas. Esta señal se conecta a la salida "bit_state_output" de la neurona predecesora en el caso de la segunda y posteriores y permite inicializar la ONN.
- **bit_state_output:** Señal de salida de estado para la siguiente neurona. Esta señal se genera mediante el desplazamiento de la señal de entrada "bit_initstate".
- **state_neuron:** Señal de salida señal proporciona el estado actual de la neurona. El tamaño del estado viene determinado por el parámetro "NbitsState"
- **Noutput:** Señal de salida del oscilador. Esta señal de salida de las neuronas y se utiliza para calcular la siguiente entrada de las neuronas.

Las funciones que este módulo realiza son las siguientes:

- Genera una señal cuadrada mediante un registro de desplazamiento de 16 bits a partir del reloj de entrada "slowclk".
- Calcula el estado de la neurona "stateint":
  - El estado inicial se introduce en la neurona, cuando "full_tick" está a nivel 0, desde el exterior a través de la entrada "bit_initstate".
  - La señal "bit_initstate" corresponde a la imagen serializada que se pretende analizar.
  - El estado se actualiza, cuando "full_tick" está a nivel 1, en función de la detección de los flancos de "Noutput" y "Ninput"
- Proporciona la señal de salida en "Noutput". Esta señal se genera a partir del registro de desplazamiento, mediante un multiplexor que selecciona la fase en función de la señal de estado de la neurona "stateint".
- Detecta los flancos de la señal de salida "Noutput" y de la señal de entrada "Ninput".
- En función de los flancos de las señales "Noutput" y "Ninput" adelanta o atrasa el estado de la neurona "stateint" y, por tanto, adelanta o atrasa la señal "Noutput" con el objetivo de alinearla con la señal de entrada "Ninput".

Simulaciones:

En la Figura 8 se muestra una simulación de "neuron". Se puede observar las diferencias de comportamiento con "full_tick" a cero y a uno.

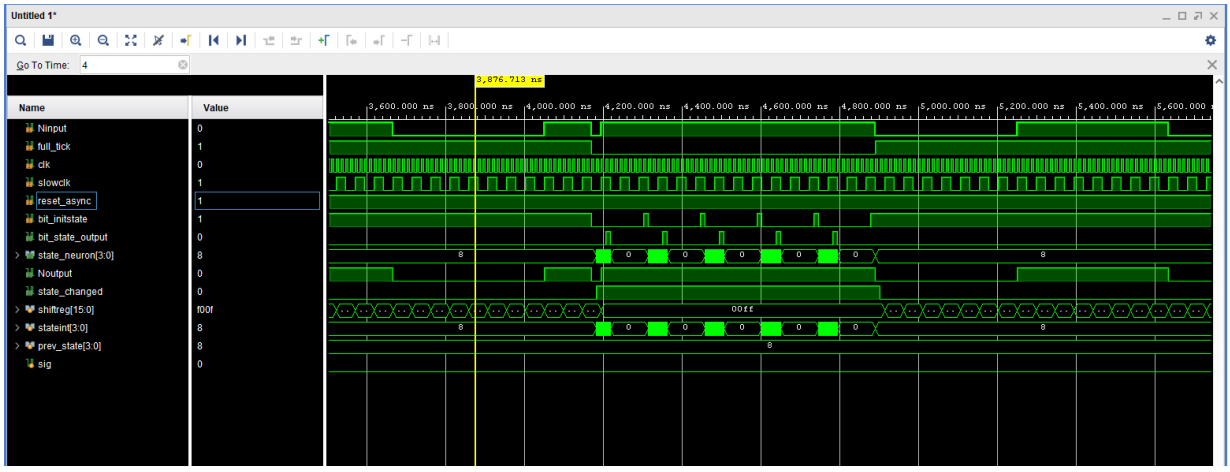


Figura 8: Comportamiento de "neuron" con full_tick = 0 y full_tick = 1.

En la Figura 9 se muestra cómo se propaga la señal "bit_initstate" (salida de la neurona predecesora) a la salida "bit_state_output" (entrada de la neurona siguiente) en función del reloj de entrada "clk". Puede observarse también como cambia el estado de la neurona ("señal state_neuron").

Se necesitan, en total, 160 ciclos de reloj para poder inicializar todas las neuronas de la red neuronal. Estos 160 ciclos corresponden a el número de neuronas multiplicado por el tamaño de codificación del estado de las neuronas.

$$Numero\ de\ Ciclos = Nrow\ (=5) * Ncol\ (=8) * NbitsState\ (=4) = 160\ ciclos$$

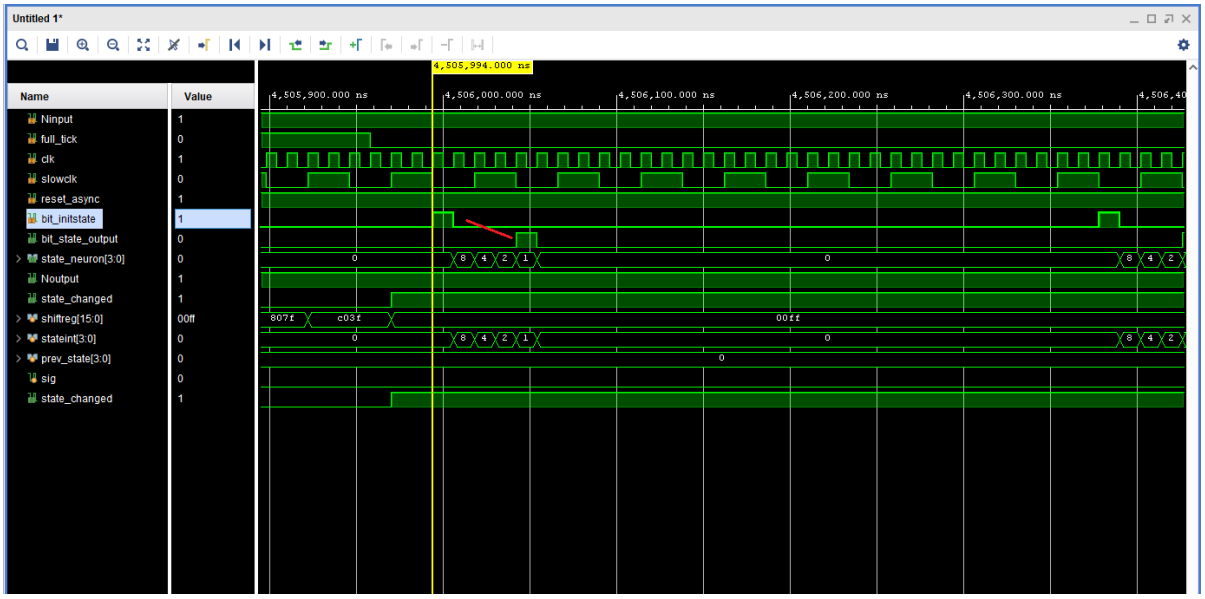


Figura 9: Operación de "neuron" con full_tick = 0

En la Figura 10 se muestra el caso en el que se produce una alineación entre la entrada y la salida. Se observa que “Ninput” y “Nouput” están desfasadas a partir del momento en el que “full_tick” vale 1, se calcula el desfase y se determina el nuevo estado “stateint” con un valor de 8. Con el nuevo valor del estado “stateint” se produce la alineación entre la entrada y la salida.

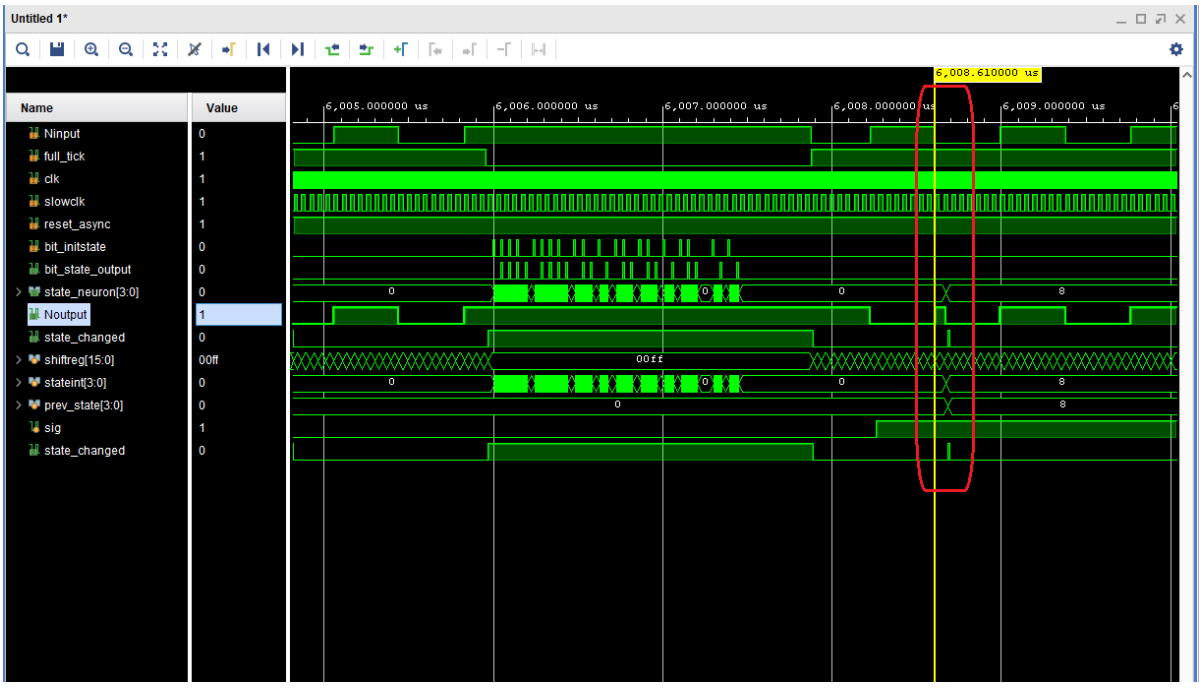


Figura 10: Alineamiento de Ninput con Noutput.

3.1.2. Módulo “Synapses”

Dentro del bloque *“synapses”* se encuentran almacenados los pesos, que emulan la función de acoplamiento entre los diferentes osciladores, y los circuitos aritméticos que se encargan de generar las entradas de las neuronas a partir de los pesos sinápticos y el estado actual de las neuronas.

El módulo “Synapses” es un bloque combinacional que actualiza las entradas de las neuronas a partir de sus salidas y los pesos almacenados. Esta actualización la realiza en función de los valores de salida de las neuronas.

Para obtener el valor de entrada de la neurona *i*, si la salida de la neurona *j* es 1, se suma el peso W_{ij}, en el caso contrario, si la salida de la neurona *j* es 0, se resta el peso W_{ij}. Con el signo de la suma sobre todas las neuronas *j* obtenidas se determina si la entrada de la neurona *i* es 1 o 0. Esto es, no es necesario realizar multiplicaciones al trabajar con la oscilación de salida de las neuronas (1 bit).

Simulaciones:

La Figura 11 muestra la simulación del proceso de actualización de los valores “Ninput” a partir de los pesos y los valores de “Noutput”. Como se puede observar, cuando “full_tick=1” los valores de “Ninput” se van actualizando. También se puede ver que la señal “steady_check_o” se pone a “1” cuando el valor de “Noutput” es estable y por tanto se puede capturar.

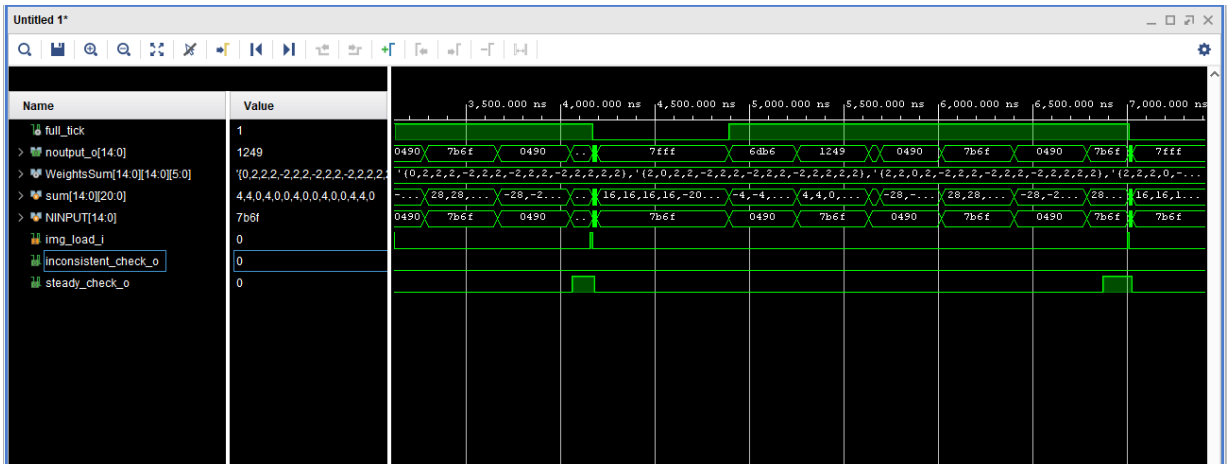


Figura 11: Simulación Synapses

3.1.3. Módulo de control

El control que se encarga de generar señales necesarias para el correcto funcionamiento de la red neuronal. Este está formado por los bloques “serial2states”, “freq_divider” y “system_status” como se muestra en la Figura 12.

El módulo de control se encarga de generar la señal de “slowclk” y “full_tick” que se utiliza en el bloque de las neuronas, registra la imagen de entrada y se la proporciona a la red de neuronas y, por último, también genera las señales “steady_check” e “inconsistent_check” que se utilizan para determinar cuando la salida de la red neuronal es estable y cuando el resultado de esta es inconsistente.

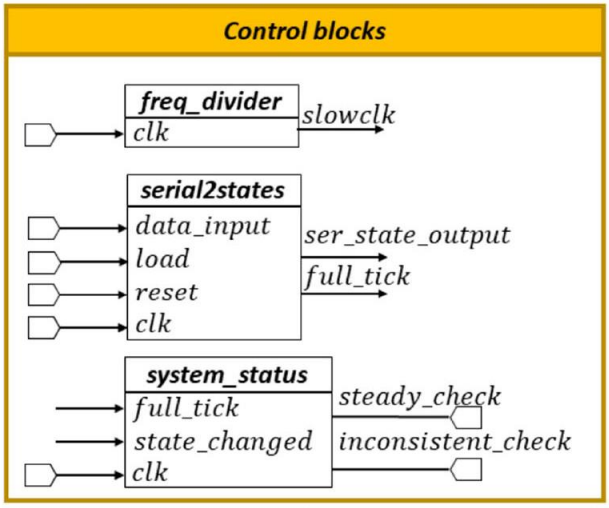


Figura 12: Bloques de control ONN Digital

A continuación, se explica en detalle el funcionamiento del bloque más complejo “serial2states”.

“serial2states”

Este módulo realiza las siguientes funciones:

- Inicia el proceso de aplicar un patrón de entrada a la ONN. Esto es, cargar un determinado estado “data_input” al recibir un pulso en la señal “load”. El proceso consiste en propagar la señal de entrada “data_input” a “ser_state_output” cuando se ordena la aplicación de un patrón a la ONN y generar las correspondientes señales de control para la ONN. En la señal “data_input” se proporciona desde el exterior la imagen o estado de cada una de las neuronas.
- Genera la señal “full_tick” para indicar que se carga una nueva imagen en la red neuronal.

Los puertos del módulo “**serial2states**” son los siguientes:

- **clk:** Señal de entrada de reloj maestro del módulo.
- **reset:** Señal de entrada de reset asíncrono.
- **load:** Señal de entrada que inicializa el proceso de serialización.
- **data_input:** Dato de entrada en serie de la imagen a procesar por la red neuronal. Este dato se propaga a la salida “ser_state_output” al iniciarse la máquina de estado a partir de un pulso en la señal de entrada “load”
- **full_tick:** Señal de salida que permanece a 0 durante la serialización del dato de entrada y que se pone a 1 una vez terminada la serialización. Esto permite iniciar la operación de inferencia en la ONN.
- **ser_state_output:** Señal de salida serializada. Esta señal se conecta a la entrada de la primera neurona y permite cargar el estado inicial de cada neurona.

Simulaciones:

La Figura 13 muestra una secuencia completa de serialización de una imagen de entrada.

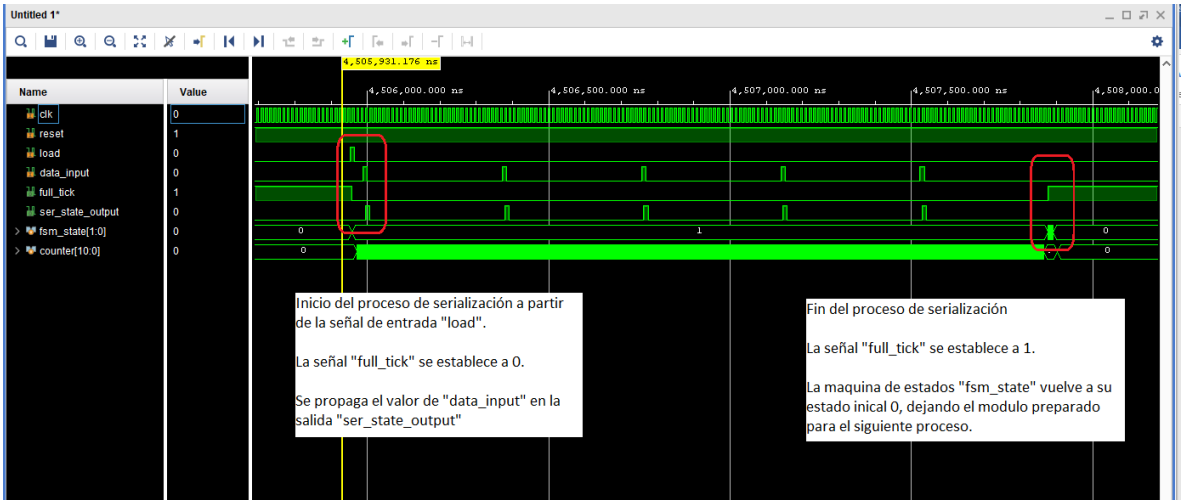


Figura 13: Simulación "serial2states" 1

La Figura 14 muestra la propagación de la señal “data_input”. El módulo asigna a la salida “ser_state_output” el estado de la señal “data_input” a partir del flanco de subida del reloj “clk”.

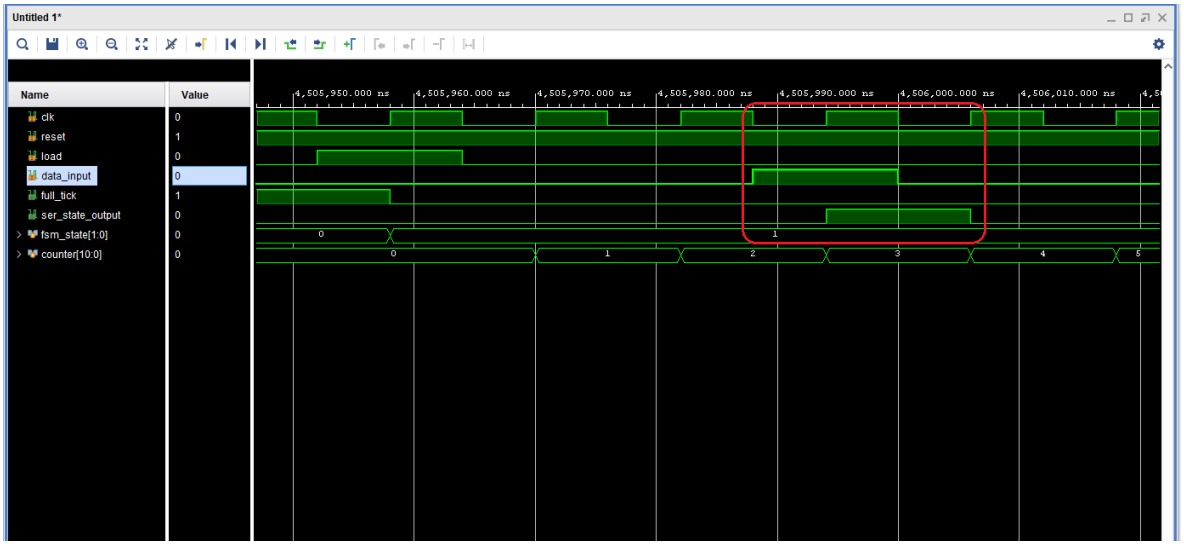


Figura 14: Simulación "serial2states" 2

4. Diseño del Sistema

En este capítulo se explica cómo se ha desarrollado el proceso de dotar a la ONN digital anteriormente analizada de capacidad de aprendizaje on-line. Esta capacidad se ha logrado implementando en hardware una modificación del algoritmo de aprendizaje iterativo supervisado descrito en el Capítulo 2.

Se ha comenzado modelando el algoritmo en MATLAB para validarlo conceptualmente y utilizarlo como punto de referencia para la posterior implementación en hardware.

El siguiente paso ha sido su implementación hardware mediante el diseño de diferentes módulos funcionales desarrollados a nivel RTL en Verilog y validados mediante simulaciones. Una vez chequeado su funcionamiento correcto, se ha procedido a validar la operación de la ONN con capacidad de aprendizaje on-line integrando los distintos bloques desarrollados junto con la ONN digital. Finalmente, el sistema se ha sintetizado e implementado para comprobar la viabilidad de su realización física en la plataforma Zybo en cuanto a recursos necesarios. El proceso de diseño se ha llevado a cabo en el entorno Vivado de Xilinx. Los módulos desarrollados en Verilog son completamente parametrizables para permitir escalar el diseño de una manera rápida y sencilla.

4.1. Diseño del algoritmo de aprendizaje iterativo

En este punto se explica el proceso de modelado del algoritmo de aprendizaje iterativo en MATLAB. Se parte desde el modelado más simple de aprendizaje de Hebb sin supervisión, luego continúa explicando las diferentes funciones que se han utilizado durante todo el proceso, y finalmente se termina con la explicación de cómo se ha implementado el algoritmo de aprendizaje iterativo.

4.1.1. Módulo TestMatrix

La función TestMatrix modela la red neuronal. Calcula su salida para un patrón de entrada.

Esta función tiene los siguientes argumentos como entrada:

- **M**: Matriz que contiene el patrón a testear.
- **W**: Matriz de pesos. Contiene la matriz de pesos con la que se testea el patrón de entrada.
- **N_iter**: Numero de iteraciones en el proceso de test.

Y proporciona las siguientes salidas:

- **Mout_HL**: Matriz de salida resultado de multiplicar la matriz de entrada M, que contiene el patrón a testear, por la matriz de pesos W. Mout_HL proporciona la salida equivalente a la red neuronal.
- **UpdateMatrix**: Matriz de salida del mismo tamaño que "Mout_HL". Esta matriz indica que posiciones de la matriz de salida Mout_HL son diferentes a la matriz de entrada M. No es necesario para calcular la respuesta de la red neuronal, pero se utiliza en la función de aprendizaje iterativo como se explicará posteriormente.

4.1.2. Módulo hebb_unsupervised

Este módulo implementa la regla de aprendizaje de Hebb sin supervisión. Como ya se ha indicado, este tipo de aprendizaje solo depende de los vectores de entrada y no tiene en cuenta el resultado de la red neuronal.

Se ha implementado en MATLAB un script dedicado a testear el aprendizaje de Hebb. Este lee desde un fichero externo los patrones a entrenar, determina el número total de patrones y su

longitud para enviarlos a la función *"function W = update(W,pattern)"* que se encarga de actualizar la matriz de pesos conforme a la ecuación:

$$W(n+1) = W(n) + x^T * x$$

Para probar el funcionamiento, se ha implementado un ejemplo con 15 neuronas en el que se ha utilizado un conjunto de tres patrones de entrada que están almacenados en un fichero de texto, con el siguiente contenido:

1,1,1,1,0,1,1,0,1,1,0,1,1,1,1
0,0,1,0,0,1,0,0,1,0,0,1,0,0,1
1,1,1,0,0,1,1,1,1,1,0,0,1,1,1

Estos valores representan cada uno de los pixeles de una imagen de 5x3, donde cada elemento indica si el color de píxel es negro, con un "1", o blanco, con un "0". Por ejemplo, el vector (1,1,1,1,0,1,1,0,1,1,0,1,1,1,1) representa el numero 0 sobre una matriz de 5x3. Ver Figura 15.

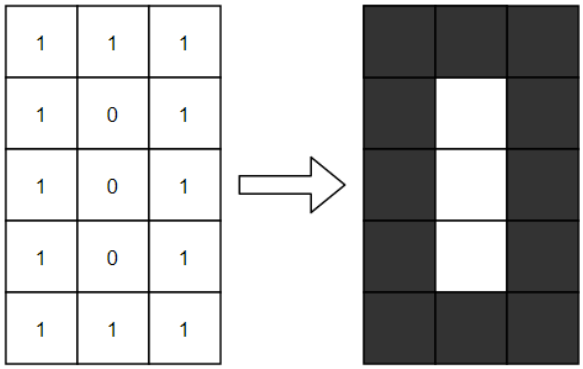


Figura 15: Representación número 0 sobre matriz de 5x3 píxeles.

Otro ejemplo es el vector (1,1,1,0,0,1,1,1,1,1,0,0,1,1,1) que representa el número 2 en una matriz de 5x3. Ver Figura 16.

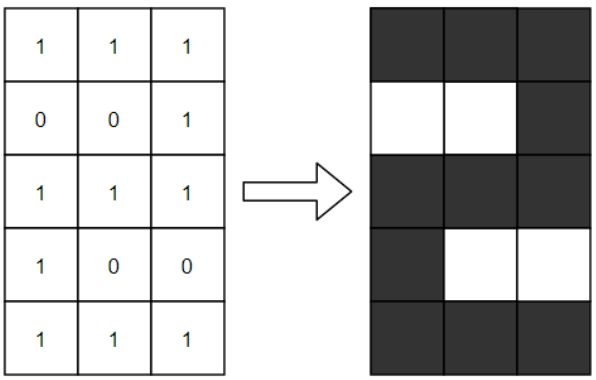


Figura 16: Representación número 2 sobre matriz de 5x3

Durante el aprendizaje, la matriz de pesos se va adaptando a medida que se introducen nuevos patrones para almacenar, hasta llegar un momento en el que pierde la capacidad de recuperar los patrones ya aprendidos.

A continuación, se ilustra este comportamiento. Se va a realizar un test para comprobar la capacidad de almacenamiento de nuevos patrones. Para ello, se va a partir de una matriz inicializada con todos sus valores a 0, se aprende un patrón, se calcula la respuesta de la red con la matriz de peso tras el aprendizaje para dicho patrón (se testea el patrón) y se va a comprobar la evolución a medida que se introducen nuevos patrones a aprender. Resumiendo, el procedimiento que se va a realizar es el siguiente:

- Se aprende el patrón 0.
- Se testea el patrón 0.
- Se aprende el patrón 1.
- Se testea el patrón 0.
- Se aprende el patrón 2.
- Se testea el patrón 0.

El resultado obtenido del test es el mostrado en la Tabla 1.

Tabla 1: Resultado test aprendizaje sin supervisión para el patrón 0

Patrón entrada "0"	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**
Resultado Test después de aprender patrón "0"	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1
Resultado Test después de aprender patrón "1"	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1
Resultado Test después de aprender patrón "2"	1	1	1	1	0	1	1	0	1	1	0	0	1	1	1

Observando la evolución de los resultados, se ve que, después de aprender el patrón 0, los resultados obtenidos del test son satisfactorios, pues se consigue recuperar el patrón de entrada.

En el segundo resultado, después de aprender el patrón “1”, se ve que se continúa recuperando el patrón “0”.

En el tercer test, después de introducir el patrón 2 en la matriz, se observa que el resultado para uno de los valores pasa a ser 0 (marcado en rojo). Esto implica que, la matriz de pesos empieza a perder la capacidad de recuperar los patrones aprendidos.

También es interesante analizar la evolución del resto de patrones. Para el caso del patrón de entrada 1 se muestra en la Tabla 2. Se observa que en la primera fase de aprendizaje el resultado no es correcto, ya que, el patrón 1 no se ha aprendido aún y en este caso, se obtiene como resultado el patrón 0. Tras aprender el patrón 1, se puede ver que el resultado es correcto, por lo que se comprueba que el algoritmo funciona para al menos dos patrones.

Finalmente, después de añadir el patrón 2, el resultado indica que el sistema ha dejado de recuperar el patrón “1”, ya que vuelve a introducir errores en la detección.

Tabla 2: Resultado test aprendizaje sin supervisión para el patrón 1

Patrón entrada "1"	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**
Resultado Test después de aprender patrón "0"	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1
Resultado Test después de aprender patrón "1"	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
Resultado Test después de aprender patrón "2"	1	1	1	0	0	1	1	0	1	1	0	1	1	1	1

Por último, en la Tabla 3 se analiza el patrón 2. En esta se observa que el sistema no es capaz de recuperar el patrón “2” al añadirlo a la matriz de pesos.

Tabla 3: Resultado test aprendizaje sin supervisión para el patrón 2

Patrón entrada "2"	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**
Resultado Test después de aprender patrón "0"	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1
Resultado Test después de aprender patrón "1"	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1
Resultado Test después de aprender patrón "2"	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1

El resultado obtenido en este test indica que solo se ha conseguido aprender 2 patrones de entrada, lo cual es coherente con el límite de 0.15*n indicado en el Capítulo de Fundamentos.

4.1.3. Módulo LearnHebb

Para evolucionar el módulo de aprendizaje de Hebb hacia el algoritmo de aprendizaje iterativo supervisado, se ha implementado la función LearnHebb. El objetivo de esta es implementar una función que pueda bloquear las posiciones de la matriz de pesos que no se desean actualizar.

Función “function [WeightMatrixOut] = LearnHebb(InputTrainingMatrix,WeightMatrix,UpdateMatrix, NumError)”

Esta función tiene los siguientes argumentos de entrada:

- **InputTrainingMatrix**: Matriz de entrada de tamaño *Nrow*Ncol* que contiene el patrón que se desea aprender.
- **WeightMatrix**: Es la matriz de pesos sinápticos de tamaño *n*n*, siendo *n = Nrow*Ncol*, que se desea actualizar según el patrón de entrada **InputTrainingMatrix**.
- **UpdateMatrix**: Es una matriz de tamaño *Nrow*Ncol* que indica que los pesos asociados a la neurona indicada en **UpdateMatrix** se puede actualizar o no.

La función proporciona como salida la matriz de pesos actualizada de tamaño *n*n*.

La función LearnHebb acepta en la entrada **InputTrainingMatrix** los patrones en formato de matriz con valores de -1 y 1. Ejemplo:

InputTrainingMatrix = [1 1 1; 1 -1 1; 1 -1 1; 1 -1 1; 1 1 1]

1	**1**	**1**
1	**-1**	**1**
1	**-1**	**1**
1	**-1**	**1**
1	**1**	**1**

En el ejemplo anterior, la matriz está representando el número 0 en una matriz de 5*3.

En la entrada **UpdateMatrix** recibe una matriz del mismo tamaño que **InputTrainingMatrix.** Esta matriz contiene 0 y 1 para indicar qué posiciones de la matriz **InputTrainingMatrix** se puede actualizar y cuáles no. Esto permite bloquear las posiciones que no se desean modificar.

- Si **UpdateMatrix(i,j)** = 0 la posición *(i,j)* de la matriz no se puede actualizar.
- Si **UpdateMatrix(i,j)** = 1 la posición *(i,j)* de la matriz se puede actualizar.

Ejemplo:

UpdateMatrix = [0 0 0; 1 0 0; 0 0 1; 0 0 0; 0 1 0]

0	**0**	**0**
1	**0**	**0**
0	**0**	**1**
0	**0**	**0**
0	**1**	**0**

Esto indica que los pesos correspondientes a las neuronas de las posiciones marcadas con "1" son las que se pueden modificar, mientras que los pesos de las neuronas marcadas con "0" quedan bloqueadas y, por tanto, no se pueden modificar.

En la entrada **WeightMatrix** se proporciona la matriz de pesos que se desea actualizar con tamaño *n*n*.

La función devuelve una matriz de pesos actualizada aplicando la regla de aprendizaje de Hebb a partir de la matriz de entrada a entrenar **InputTrainingMatrix** y de la matriz de pesos **WeightMatrix**, pero solo actualizando los pesos correspondientes a posiciones que no están bloqueadas, es decir, los marcados con "1" según la matriz de entrada **UpdateMatrix.**

4.1.4. Módulo Aprendizaje Iterativo LearnIterative

La función ***LearnIterative*** implementa el proceso de aprendizaje iterativo supervisado que hemos desarrollado. Esta función usa las funciones anteriormente descritas. El proceso de aprendizaje iterativo se muestra en el diagrama de flujo de la

Figura 17.

El proceso comienza con la inicialización de algunas variables necesarias para el funcionamiento del algoritmo. Los patrones para aprender se leen desde una variable y estos se envían para que se aprendan. Los patrones que se van aprendiendo se van almacenando en "Pattern_List".

El tratamiento para el primer patrón es ligeramente diferente que para el caso de los siguientes patrones. El primer patrón se envía directamente al aprendizaje mediante la función LearnHebb, mientras que los siguientes, primero se testean y luego, en función de los errores detectados, se aprenden.

Cuando el algoritmo intenta aprender un nuevo patrón, se produce una actualización de la matriz de pesos y esto puede dar lugar a que patrones que ya estaban aprendidos dejen de estarlo y, por tanto, requieran de nuevas correcciones en la matriz de pesos. Por esto, después de cada uno de los procesos de aprendizaje, se vuelve a testear desde la primera posición de la lista "Pattern_List".

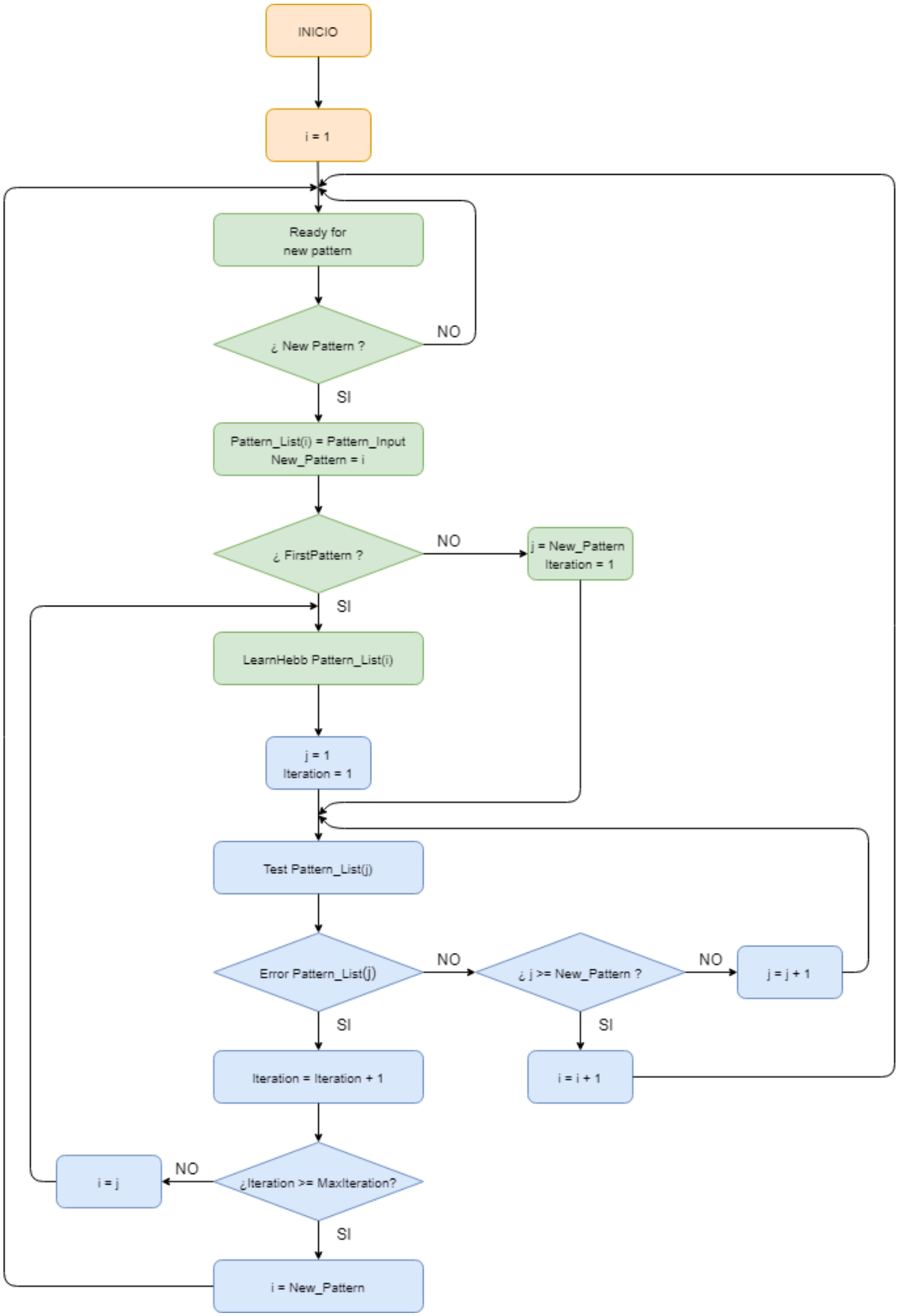


Figura 17: Diagrama Flujo aprendizaje iterativo

El proceso de test consiste en aplicar el patrón de entrada en la red neuronal. Esta comprobación devuelve si existen errores o no, y en función de esto se continúa con el siguiente patrón o se envía el patrón al aprendizaje. Este proceso continúa hasta que se corrigen los errores o de lo contrario hasta que se llega a un número máximo de intentos.

Cuando se llega a un límite de intentos con un patrón determinado, el algoritmo finaliza el aprendizaje de dicho patrón y vuelve al inicio para continuar con el siguiente patrón disponible.

Finalmente, para comprobar visualmente el resultado del proceso de aprendizaje, se muestran gráficamente los patrones aprendidos. En la Figura 18 se muestra en la izquierda el conjunto de patrones que se han enviado al algoritmo de aprendizaje y en la imagen de la derecha se muestra los patrones que han sido aprendidos ("pattern_list").

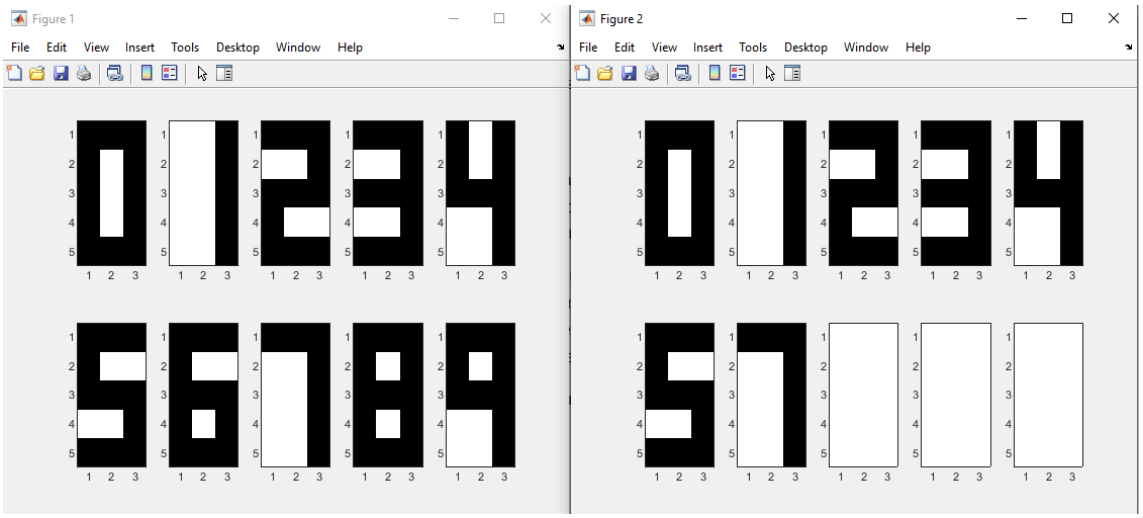


Figura 18: Resultado aprendizaje iterativo

De los resultados, se obtiene que se han aprendido un total de 7 patrones, lo que da lugar a una ratio de 0.46**n*, frente al 0.15**n* que se tiene con un proceso de aprendizaje no supervisado. Esto proporciona una mejora sustancial respecto al algoritmo de Hebb simple.

4.2. Implementación en HW

En este apartado se explica el funcionamiento de los dos módulos principales que se han desarrollado para la implementación hardware del algoritmo de aprendizaje iterativo.

El primero de los módulos, "mod_update_matrix", es el encargado de actualizar la matriz de pesos. La función de este módulo es equivalente a la función realizada por el módulo "LearnHebb" implementado en MATLAB, es decir, actualiza la matriz de pesos.

El segundo módulo que se trata en este apartado, es el encargado de realizar el proceso de aprendizaje iterativo, "mod_learn_hebb_iterative". La función de este módulo es la de recibir los patrones a aprender y realizar el proceso iterativo para actualizar la matriz de pesos teniendo en cuenta la respuesta de la red neuronal.

Para el desarrollo de estos módulos se ha utilizado Verilog y se ha elaborado también *testbenches* específicos con el objetivo de verificar el correcto funcionamiento. Todo esto se ha realizado mediante el entorno de desarrollo proporcionado por la herramienta Vivado de Xilinx.

Estos módulos se integrarán en un sistema que permite establecer el entorno necesario para poder testear el funcionamiento completo. Esto implica que habrá una interacción entre los diferentes módulos, que se describirá en el siguiente apartado.

4.2.1. Módulo mod_update_matrix

El módulo **"mod_update_matrix"** realiza las siguientes funciones:

- Almacenar la matriz de pesos sinápticos de la red neuronal.
- Recibir en su entrada el patrón que se pretende almacenar junto con una máscara que indica que posiciones de la matriz de pesos se pueden actualizar y una señal que indica en qué momento se pueden capturar los datos en la entrada del módulo.
- Almacenar una copia de la matriz de pesos sinápticos.
- Recuperar la copia de la matriz de pesos sinápticos.
- Actualizar la matriz de pesos sinápticos en función de un patrón de entrada y la máscara. La actualización de los pesos se realiza siguiendo el algoritmo de Hebbian.
- Proporcionar en la salida la matriz de pesos para que sea utilizada en la red de neuronas.

Los puertos del módulo "**mod_update_matrix**" son los siguientes:

- **clk_i:** Señal de reloj de entrada.
- **nrst_i:** Señal de reset de entrada.
- **learn_i:** Señal de entrada que indica el inicio de aprendizaje de un nuevo patrón.
- **savematrix_i:** Señal de entrada que permite salvar el estado actual de la matriz de pesos. Cuando se recibe un pulso el savematrix_i se actualiza WeightMatrix_backup con WeightMatrix.
- **recovermatrix_i:** Señal de entrada que permite recuperar la última matriz almacenada en WeightMatrix_backup. Cuando se activa esta señal, se actualiza el valor de WeightMatrix con el valor almacenado en WeightMatrix_backup.
- **pattern_i:** Patrón de entrada a aprender.
- **updateweight_i:** Máscara que indica que posiciones de la matriz de pesos se permiten actualizar.
- **weightmatrixparallel_o:** Señal de salida que proporciona el valor de la matriz de pesos.
- **busy_o:** Señal de salida que indica con un 1 cuándo el módulo está ocupado realizando una operación de actualización. Permite sincronizar el módulo con el resto del sistema.

El módulo permite, mediante parametrización, establecer el número de neuronas y el número de bits utilizado para los pesos.

Para verificar este módulo se ha desarrollado un *testbench* que permite ejercitar las entradas y comprobar su funcionamiento. Este *testbench* consiste en una máquina de estados que envía patrones de test al módulo "mod_update_matrix". Los patrones de test se obtienen mediante un fichero de texto externo donde están contenidos, estos se almacenan en un array y se leen mediante la máquina de estados.

La máquina de estados maneja las señales de entrada al módulo "learn_i", "pattern_i" y "updateweight_i" y para gestionar el envío de patrones, utiliza la señal de salida "busy_o" y un contador para controlar el número de patrones enviados.

El fichero de texto que contiene los patrones de test tiene el formato que se puede ver en la Figura 19. Cada patrón consiste en un array de 15 caracteres (este ejemplo está basado en una red de 5x3). Cada uno de los caracteres están representados por 0's y 1's que representan el contenido de la imagen, en este caso el patrón de entrada está formado por 10 vectores que representan los números de 0 a 9 en formato de 5x3.

```
1  111101101101111
2  001001001001001
3  111001111100111
4  111001111001111
5  101101111001001
6  111100111001111
7  111100111101111
8  111001001001001
9  111101111101111
10 111101111001001
```

Figura 19: Formato patrones de entrada

El envío de patrones al módulo lo gestiona una máquina de estados que lee las posiciones del array "read_training" y se las asigna al vector de entrada al módulo "pattern".

Una vez que se termina de enviar todos los patrones, se almacena el resultado de la matriz de pesos en un fichero externo "output.txt". Esto permite poder utilizar la matriz de pesos para hacer pruebas o verificar el resultado.

En la Figura 20 se observa el proceso de envío de los patrones al módulo. Se va actualizando el valor de los patrones en la señal "pattern_i" junto con cada pulso en la señal "learn_i". Estos patrones son los que están almacenados en el array "read_training". Al finalizar el envío de los patrones que hay en el fichero, la máquina de estados permanece en reposo.

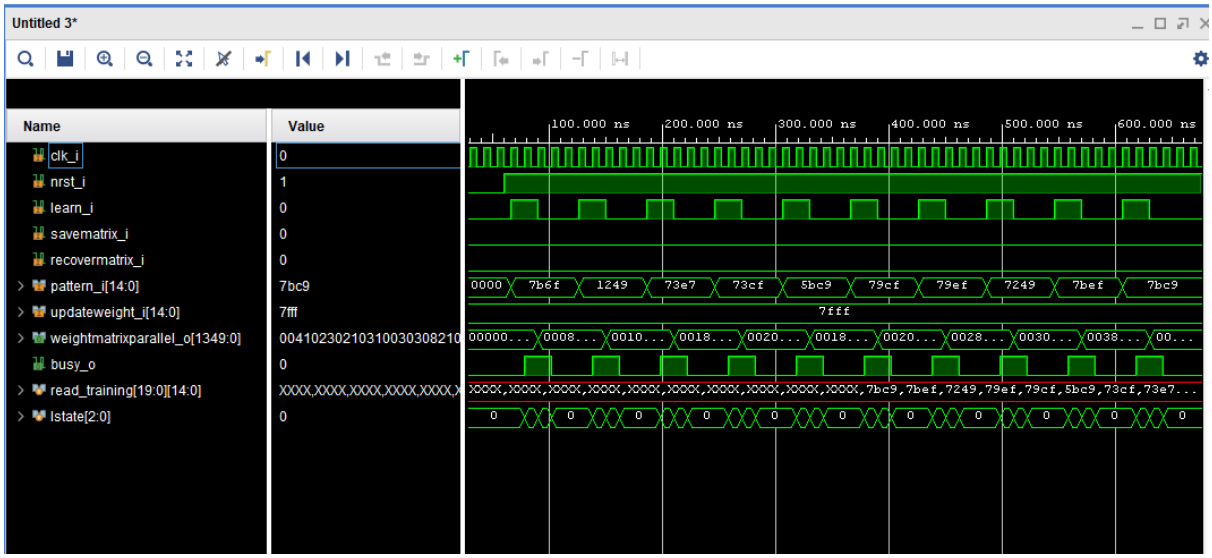


Figura 20: Testbench mod_update_matrix 1.

En la Figura 21 se muestra la evolución de la matriz de pesos sinápticos en función de los patrones de entrada “pattern_i” y de la señal “learn_i”. En esta se puede ver también el comportamiento de la función de copiar y recuperar la matriz al generarse las señales “savematrix_i” y “recovermatrix_i”.

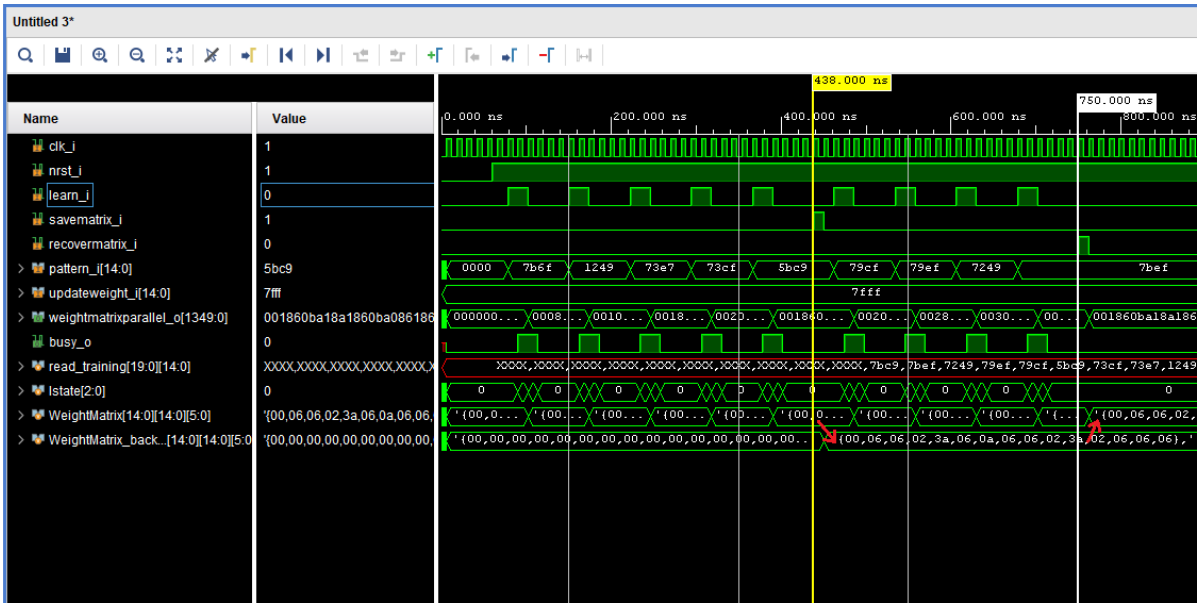


Figura 21: Testbench mod_update_matrix 3

Finalmente, se comprueba el resultado obtenido en el fichero “output.txt” generado. La Figura 22 muestra un ejemplo del aspecto del fichero de salida.

1	0	-16	-16	8	-16	8	-16	12	-16	0	-16	12	8	8	-16
2	-16	0	12	4	-12	4	12	8	12	4	-12	8	12	12	12
3	-16	12	0	4	12	12	12	8	-12	-4	12	-16	4	4	-12
4	8	4	4	0	-4	-4	12	8	4	4	-4	8	4	4	4
5	-16	-12	12	-4	0	-12	-12	-8	12	4	-12	-16	-4	-4	12
6	8	4	12	-4	-12	0	4	0	12	-4	-12	8	-4	-4	12
7	-16	12	12	12	-12	4	0	-16	12	4	-12	8	12	12	12
8	12	8	8	8	-8	0	-16	0	8	0	-8	4	8	8	8
9	-16	12	-12	4	12	12	12	8	0	-4	12	-16	4	4	-12
10	0	4	-4	4	4	-4	4	0	-4	0	4	-8	12	12	-4
11	-16	-12	12	-4	-12	-12	-12	-8	12	4	0	-16	-4	-4	12
12	12	8	-16	8	-16	8	8	4	-16	-8	-16	0	0	0	-16
13	8	12	4	4	-4	-4	12	8	4	12	-4	0	0	-12	4
14	8	12	4	4	-4	-4	12	8	4	12	-4	0	-12	0	4
15	-16	12	-12	4	12	12	12	8	-12	-4	12	-16	4	4	0

Figura 22: Fichero "output.txt" (NbitsWeights=5)

El resultado obtenido por el módulo “mod_update_matrix” se compara con el resultado obtenido en MATLAB. Al comparar se observa que existen diferencias entre ambos resultados. La

Figura 23 muestra el resultado obtenido en MATLAB.

La diferencia de resultados es debido a la limitación introducida por el número de bits utilizados en el código para codificar los pesos. Esta información es proporcionada por el parámetro “NbitsWeights”, que estaba establecido en 5 bits. Se actualiza a 6 bits, se vuelve a lanzar la simulación y el resultado obtenido es el mostrado en la Figura 24.

Se observa que el resultado obtenido con NbitsWeights=6 coincide con el resultado obtenido en MATLAB.

0	16	16	8	-16	8	16	12	16	0	-16	12	8	8	16
16	0	12	4	-12	4	12	8	12	4	-12	8	12	12	12
16	12	0	4	-20	12	12	8	20	-4	-20	16	4	4	20
8	4	4	0	-4	-4	12	8	4	4	-4	8	4	4	4
-16	-12	-20	-4	0	-12	-12	-8	-20	4	20	-16	-4	-4	-20
8	4	12	-4	-12	0	4	0	12	-4	-12	8	-4	-4	12
16	12	12	12	-12	4	0	16	12	4	-12	8	12	12	12
12	8	8	8	-8	0	16	0	8	0	-8	4	8	8	8
16	12	20	4	-20	12	12	8	0	-4	-20	16	4	4	20
0	4	-4	4	4	-4	4	0	-4	0	4	-8	12	12	-4
-16	-12	-20	-4	20	-12	-12	-8	-20	4	0	-16	-4	-4	-20
12	8	16	8	-16	8	8	4	16	-8	-16	0	0	0	16
8	12	4	4	-4	-4	12	8	4	12	-4	0	0	20	4
8	12	4	4	-4	-4	12	8	4	12	-4	0	20	0	4
16	12	20	4	-20	12	12	8	20	-4	-20	16	4	4	0

Figura 23: Matriz de pesos obtenida en MATLAB.

```
 1   0  16  16   8 -16   8  16  12  16   0 -16  12   8   8  16
 2  16   0  12   4 -12   4  12   8  12   4 -12   8  12  12  12
 3  16  12   0   4 -20  12  12   8  20  -4 -20  16   4   4  20
 4   8   4   4   0  -4  -4  12   8   4   4  -4   8   4   4   4
 5 -16 -12 -20  -4   0 -12 -12  -8 -20   4  20 -16  -4  -4 -20
 6   8   4  12  -4 -12   0   4   0  12  -4 -12   8  -4  -4  12
 7  16  12  12  12 -12   4   0  16  12   4 -12   8  12  12  12
 8  12   8   8   8  -8   0  16   0   8   0  -8   4   8   8   8
 9  16  12  20   4 -20  12  12   8   0  -4 -20  16   4   4  20
10   0   4  -4   4   4  -4   4   0  -4   0   4  -8  12  12  -4
11 -16 -12 -20  -4  20 -12 -12  -8 -20   4   0 -16  -4  -4 -20
12  12   8  16   8 -16   8   8   4  16  -8 -16   0   0   0  16
13   8  12   4   4  -4  -4  12   8   4  12  -4   0   0  20   4
14   8  12   4   4  -4  -4  12   8   4  12  -4   0  20   0   4
15  16  12  20   4 -20  12  12   8  20  -4 -20  16   4   4   0
```

Figura 24: Fichero "output.txt" (NbitsWeights=6)

4.2.2. Módulo mod_learn_hebb_iterative

El módulo **"mod_learn_hebb_iterative"** implementa el método de aprendizaje iterativo indicado en el diagrama de flujo de la

Figura 17. Las principales funciones que realiza son las siguientes:

- Almacena los patrones aprendidos.
- Genera las señales necesarias para controlar el bloque "mod_update_matrix" y la propia ONN.
- Generar, a partir de los patrones, las imágenes de test que se envían a la ONN.
- Recibir las salidas de la ONN y en función de este determinar si el patrón de aprendizaje esta aprendido o no.
- Para comprobar que los patrones se han aprendido, este módulo también realiza el cálculo de los errores detectados en los patrones. Esta información se utiliza para determinar si el patrón se ha aprendido y también para calcular la máscara a aplicar en una nueva iteración.

Los puertos del módulo **"mod_learn_hebb_iterative"** son los siguientes:

- **clk_i:** Señal de reloj de entrada.
- **nrst_i:** Señal de reset de entrada.
- **learn_i:** Señal de entrada que indica el inicio del aprendizaje de un nuevo patrón.
- **pattern_i:** Patrón de entrada a aprender.
- **busy_o:** Señal de salida que indica cuando el módulo está ocupado. Si busy_o = 1, el módulo está ocupado y no acepta nuevos patrones.
- **bit_status_o:** Señal de salida que se pone a uno cuando se termina el proceso de aprendizaje del patrón introducido.
- **pattern_nok_status_o:** Señal de salida que indica que el patrón introducido no se ha podido aprender. Esta señal genera un pulso a 1 al final del proceso si el patrón introducido no se ha podido aprender.
- **learn_o:** Señal de salida que se genera para indicar al módulo **"mod_update_matrix":** cuando hay un patrón valido en la salida "pattern_o".
- **savematrix_o:** Señal de salida que se conecta a la señal "savematrix_i" del módulo "mod_update_matrix".
- **recovermatrix_o:** Señal de salida que se conecta a la señal "recovermatrix_i" del módulo "mod_update_matrix".
- **pattern_o:** Señal de salida que contiene el patrón a enviar al módulo "mod_update_matrix".
- **updateweight_o:** Señal de salida que indica la máscara con las posiciones de la matriz se permiten actualizar. Si updateweight_i = 1, se permite actualizar, en caso contrario no se permite actualizar el peso. Esta señal se conecta al bus "updatematrix_i" del módulo "mod_update_matrix".
- **img_load_o:** Señal de salida que indica la carga de una nueva imagen a la ONN.
- **serial_out_o:** Señal de salida serie que proporciona la imagen a la red neuronal. Esta imagen que se utiliza para testear el patrón y permite obtener el comportamiento de la red neuronal para el proceso iterativo.
- **ninput_i:** Señal de entrada que se conecta a la salida de la red neuronal y proporciona el estado de esta.
- **steady_i:** Señal de entrada que proporciona la red de neuronas e indica cuando el estado de estas es estable. Permite determinar el momento en el que capturar el estado en el bus "ninput_i".
- **incosistent_check_i:** Señal de entrada que viene de la red neuronal e indica que el estado de esta no es consistente y no se de utilizar.

Este módulo también se ha parametrizado.

Para testear este módulo se ha utilizado el módulo con la red neuronal para generar las señales de entrada proveniente de este módulo: ninput_i, steay_i e inconsistent_check_i.

Para ello, se ha integrado el módulo "mod_update_matrix" dentro del módulo de contiene la red neuronal. El setup para testear el módulo "mod_learn_hebb_iterative" se muestra en la Figura 25.

El *testbench* consiste en el envío de patrones al módulo de aprendizaje iterativo. Estos patrones son leídos a partir de un fichero de texto externo que contiene los datos, luego se envían de manera secuencial al módulo de aprendizaje iterativo y este se encarga de realizar el proceso a partir de una máquina de estados. Para el control del envío de los patrones se utilizan las señales "busy_o" que indica cuando el módulo está ocupado.

En la Figura 26 se muestra el instante en el que se inicia la carga del primer patrón para aprender, se observa como la señal "busy_o" se pone a 1 indicando que el módulo está ocupado y, a continuación, se replica el patrón y la señal aprendizaje en las señales de salida "patrón_o" y "learn_o". Estas señales son las que entran al módulo "mod_update_matrix" para actualizar el estado de la matriz de pesos sinápticos.

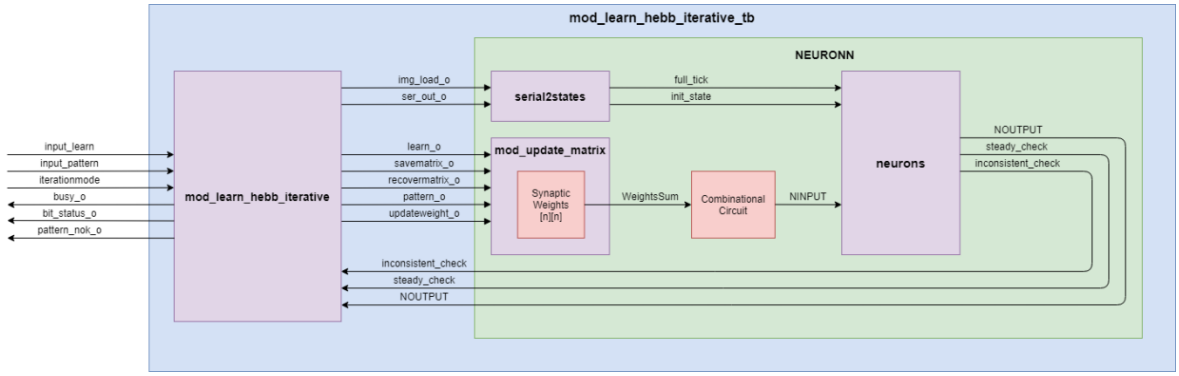


Figura 25: Setup test "mod_learn_hebb_iterative"

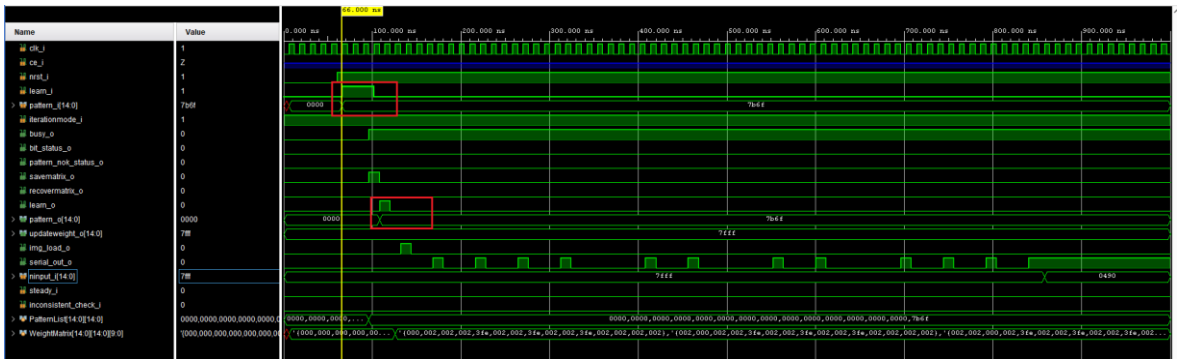


Figura 26: Test "mod_learn_hebb_iterative" 1

En la Figura 27 se observa que justo después de generarse la señal “learn_o” se produce la actualización de la matriz de pesos sinápticos.

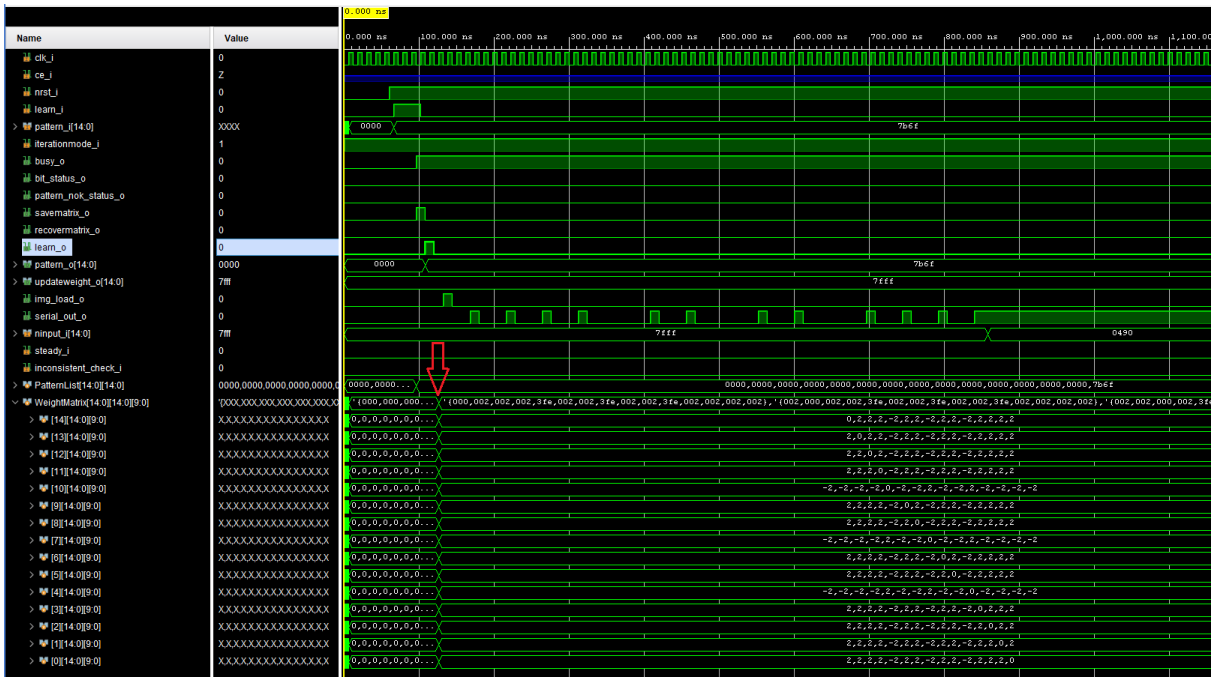


Figura 27: Test "mod_learn_hebb_iterative" 2

Justo después de actualizar la matriz de pesos, el módulo de aprendizaje iterativo envía una imagen de test mediante las señales “img_load_o” y “serial_out_o”. Estas señales se conectan al módulo de serialización “serial2states” que inicializa el estado de las redes neuronales oscilatorias (Figura 28).

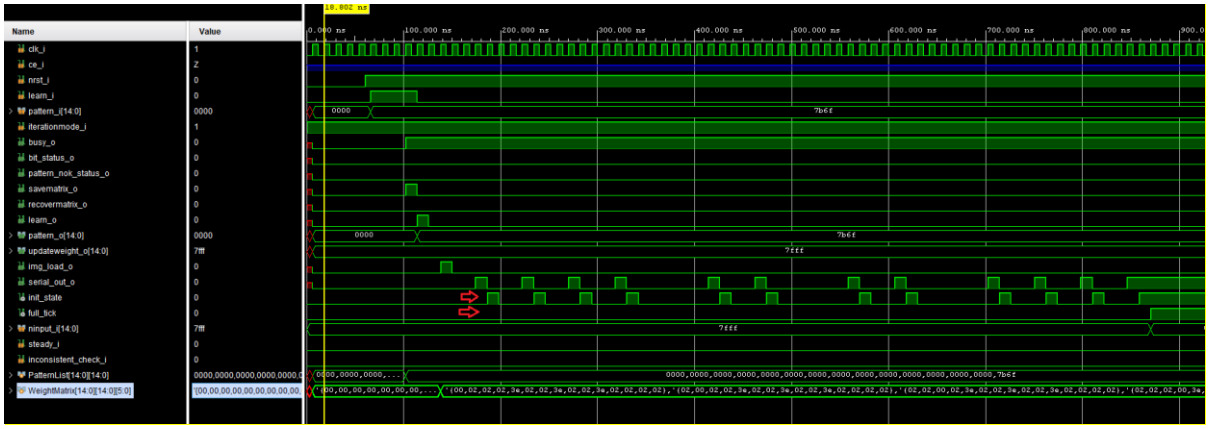


Figura 28: Test "mod_learn_hebb_iterative" 3

Desde el momento en el que la señal “full_tick” se pone a nivel 1, comienza el proceso de alineación de la red neuronal hasta que se genera la señal de “steady_i” que indica cuando la señal de salida “ninput_i” es estable. Es en este momento en el que el módulo de aprendizaje iterativo captura la salida de la red neuronal para comprobar si el patrón ha sido aprendido (Figura 29).

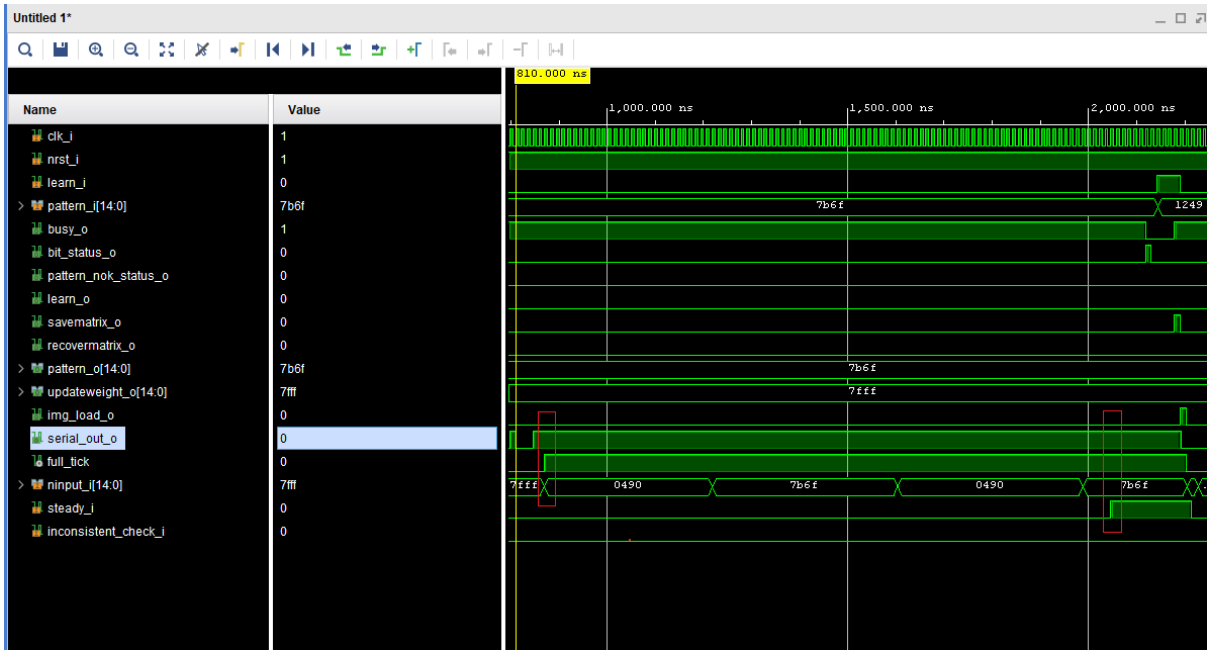


Figura 29: Test "mod_learn_hebb_iterative" 4

El test continúa hasta que se termina enviando los 10 patrones almacenados en el fichero de texto. Al terminar el proceso se puede ver que el número de patrones que ha sido capaz de aprender el sistema es de 7 patrones (guardados en las posiciones 0 a 6 de “pattern_list”). Se ve también el comportamiento de la señal “pattern_nok_status_o”. Esta se activa en tres ocasiones indicando que los tres patrones correspondientes no se han podido aprender. Esto coincide con el número de patrones enviados y aprendidos. (Figura 30).

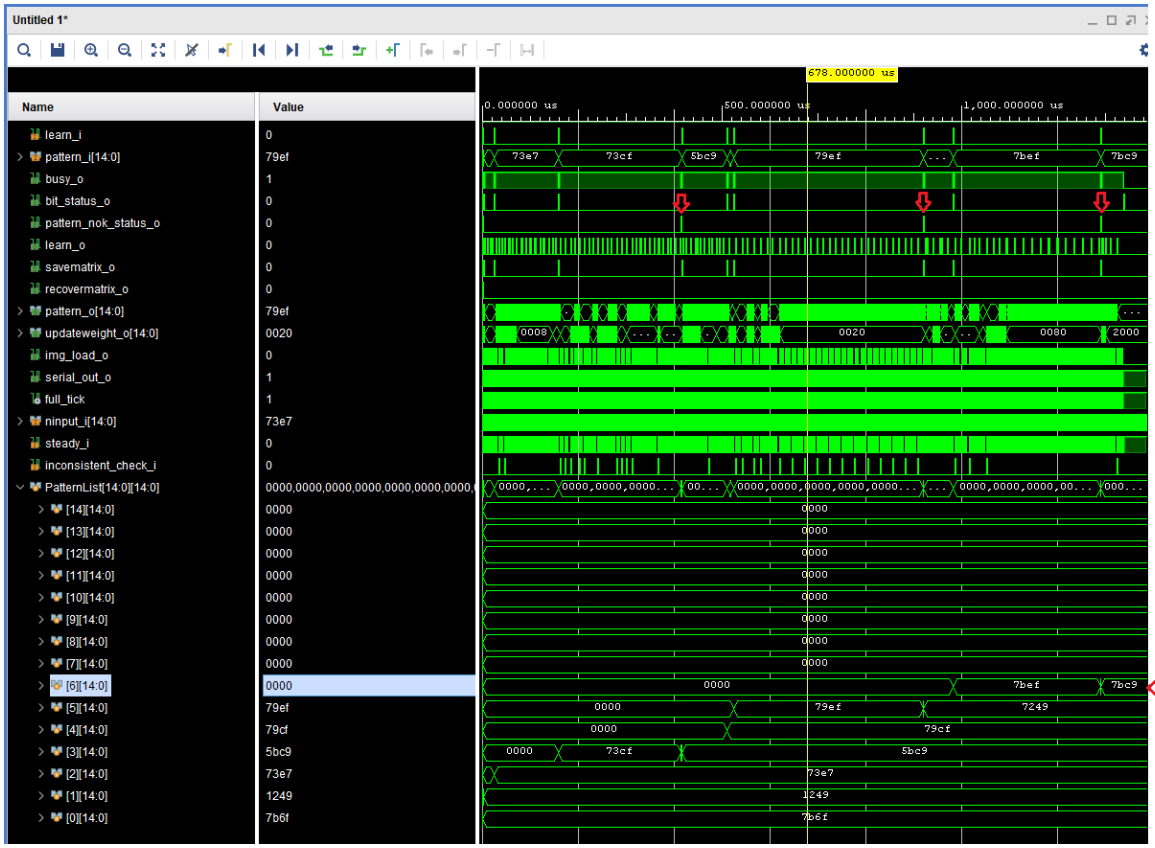


Figura 30: Test "mod_learn_hebb_iterative" 5

4.3. Integración del sistema

En este apartado se explica cómo se ha realizado la integración de todo el sistema formado con los módulos desarrollados y la ONN. Para ello, se ha elaborado un código que interconecta entre sí los módulos.

Además, para permitir tanto la funcionalidad de aprendizaje como la operación normal de la ONN, ha sido necesario añadir algunos otros elementos. Así, se ha utilizado multiplexores que seleccionan si los patrones de entrada de la ONN provienen desde el exterior o bien desde el módulo de aprendizaje iterativo.

Para proporcionar la salida de la ONN al exterior, se ha desarrollado el módulo "mod_data_capture" que se encarga de realizar la captura del dato de salida de las neuronas a partir de la señal "steady_check" y lo presenta en la salida. La Figura 31 muestra el sistema completo.

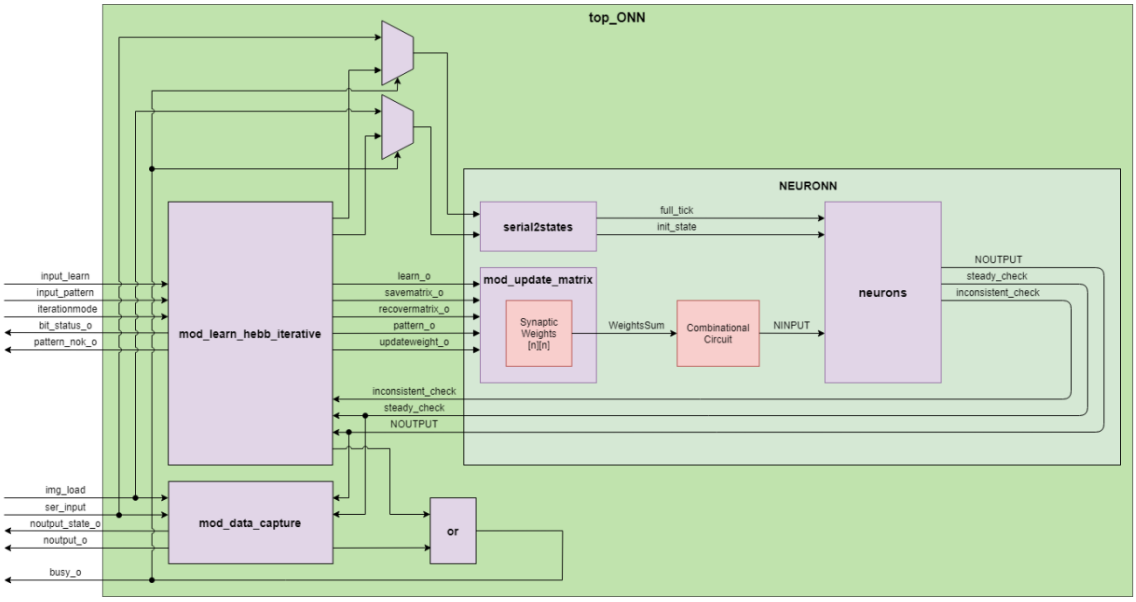


Figura 31: Diagrama de bloques del sistema completo.

Para testear el módulo se ha generado un *testbench* que permite enviar patrones de aprendizaje y patrones de test para comprobar el funcionamiento. En este caso se ha utilizado un *testbench* que es capaz de leer los patrones de test y de aprendizaje a partir de un fichero de texto. Estos se introducen en el fichero de texto con valores de 0's y 1's, donde el primer valor indica que tipo de operación se va a realizar y los siguientes contienen el patrón.

La operación se clasifica de la siguiente manera:

- 1 = Operación de aprendizaje.
- 0 = Operación de test.

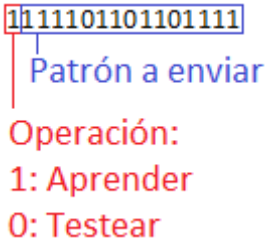


Figura 32: Codificación patrones de entrada en el testbench del sistema completo

Por ejemplo, el siguiente comando: 1111101101101111 indica que se va a aprender el patrón "111101101101111".

En la Figura 33 se muestra el diagrama de flujo del *testbench* para el sistema completo. Este *testbench* lo primero que hace es leer los datos del fichero externo y almacenarlo en el array de vectores "read_training". Este array contiene la operación y el vector que se utilizará para aprenderlo o testearlo. También se determina el número de vectores que hay en el fichero y se almacena en la variable "n_training".

La máquina de estados maneja las señales correspondientes en función de si la operación es de aprendizaje o de test. El proceso termina en el estado de reposo "Idle" cuando se han procesado todos los vectores.

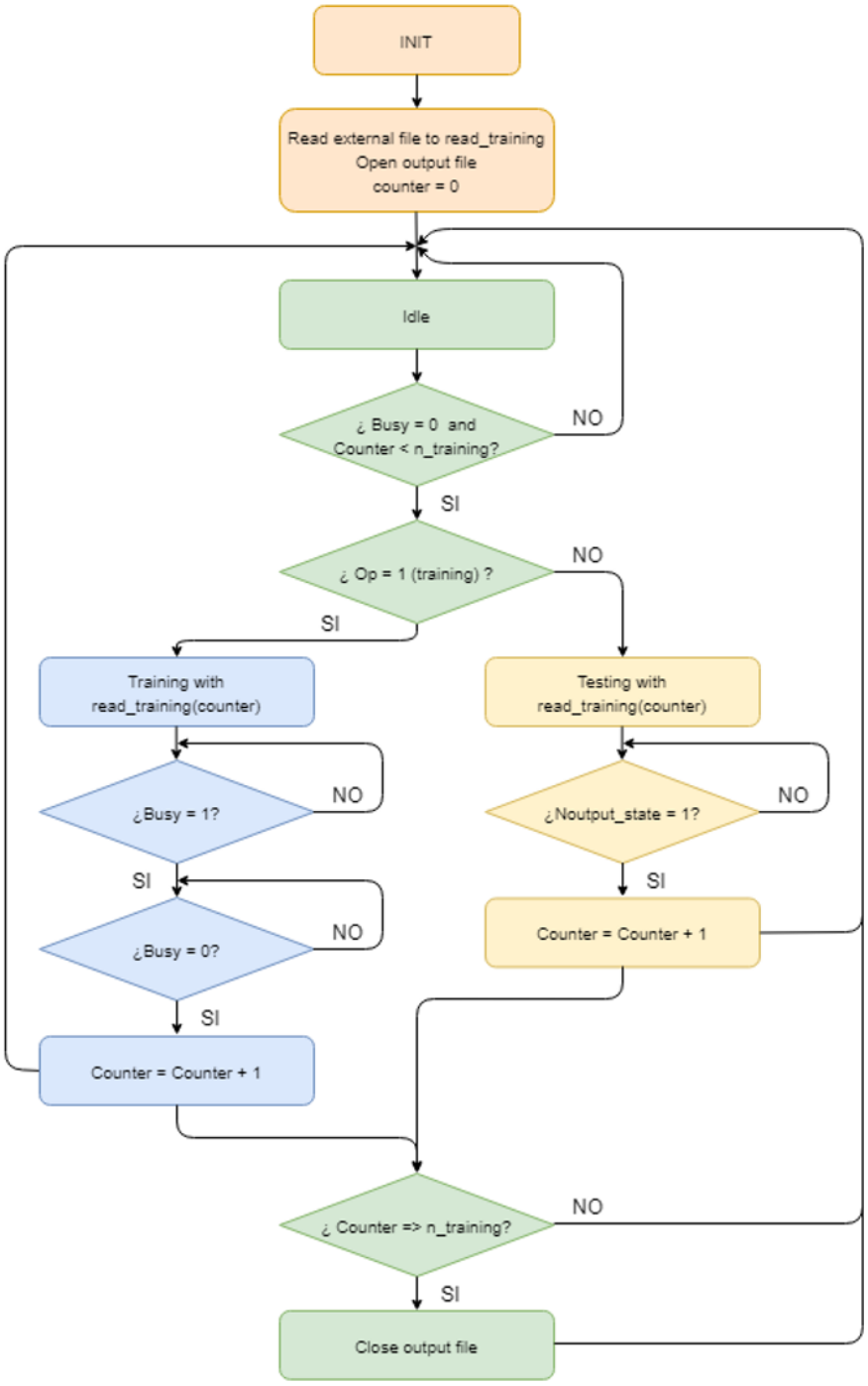


Figura 33: Diagrama flujo del testbench

Para comprobar el funcionamiento del módulo se ha preparado el fichero mostrado en la Figura 34. Este fichero de texto realiza las siguientes operaciones:

- Aprender el patrón correspondiente al número 0.
- Testear el patrón correspondiente al 0.
- Testear el patrón correspondiente al 1. Se espera que no se reconozca.
- Aprender el patrón correspondiente al número 1.
- Testear el patrón correspondiente al 1.
- Aprender los patrones correspondientes desde el numero 2 al 9.
- Testear los patrones correspondientes desde el numero 2 al 9.

```
1  1111101101101111    //Learn pattern 1111101101101111 (0)
2  0111101101101111    //Test pattern  1111101101101111 (0)
3  0001001001001001    //Test pattern  0010010010010010 (1)
4  1001001001001001    //Learn pattern 0010010010010010 (1)
5  0001001001001001    //Test pattern  0010010010010010 (1)
6  1111001111100111    //Learn pattern 1110011111001111 (2)
7  1111001111001111    //Learn pattern 1110011111001111 (3)
8  1101101111001001    //Learn pattern 1011011111001001 (4)
9  1111100111001111    //Learn pattern 1111001110011111 (5)
10 1111100111101111    //Learn pattern 1111001111011111 (6)
11 1111001001001001    //Learn pattern 1110010010010010 (7)
12 1111101111101111    //Learn pattern 1111011111011111 (8)
13 1111101111001001    //Learn pattern 1111011110010010 (9)
14 0111001111100111    //Test pattern  1110011111001111 (2)
15 0111001111001111    //Test pattern  1110011111001111 (3)
16 0101101111001001    //Test pattern  1011011111001001 (4)
17 0111100111001111    //Test pattern  1111001110011111 (5)
18 0111100111101111    //Test pattern  1111001111011111 (6)
19 0111001001001001    //Test pattern  1110010010010010 (7)
20 0111101111101111    //Test pattern  1111011111011111 (8)
21 0111101111001001    //Test pattern  1111011111001001 (9)
```

Figura 34: Patrones de entrada en el testbench del sistema completo

En la Figura 35 se muestran los cuatro primeros comandos enviados al módulo. Se aprende el patrón correspondiente al 0 (“7bff”), se testea y se observa que el resultado obtenido (“noutput_o”) es el correspondiente al patrón 0. Luego se testea el patrón 1 y se ve que el resultado obtenido vuelve a ser el correspondiente al 0.

A continuación, se envía el comando para aprender el patrón 1 y se testea. En este caso, se puede ver que el resultado obtenido es correcto (Figura 36).

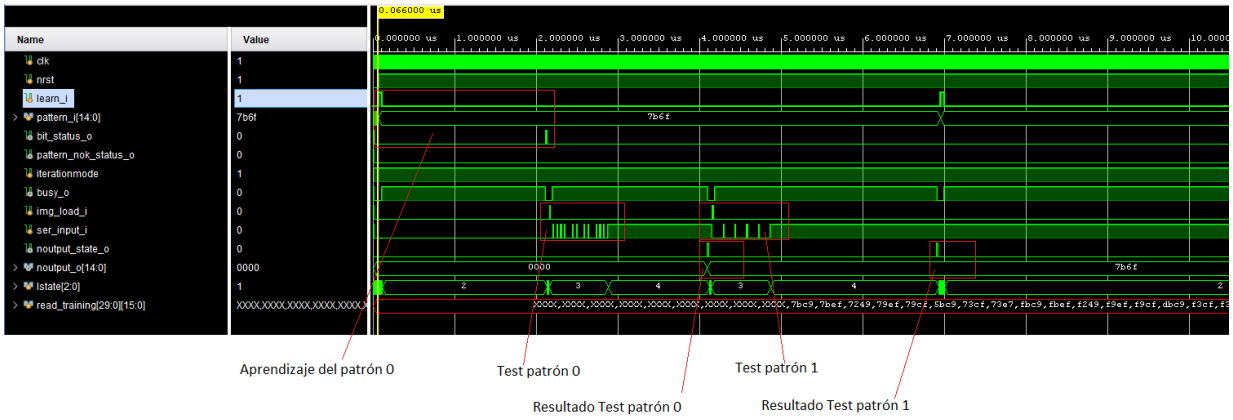


Figura 35: Simulación sistema completo 1

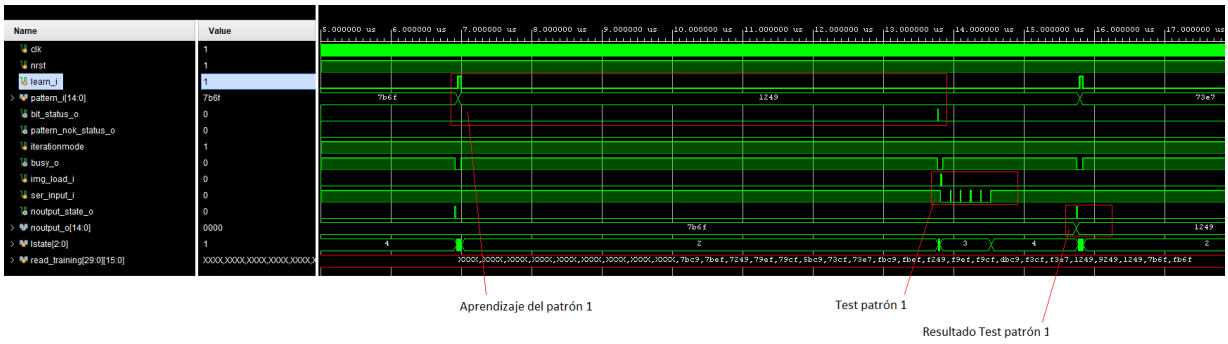


Figura 36: Simulación sistema completo 2

En la Figura 37 se muestra el proceso completo de todos los patrones del fichero de texto. Se observa que durante el aprendizaje se han producido 3 fallos de patrones (activación de la señal pattern_nok_status_o), los correspondientes al "79ef", "7bef" y "7bc9". La señal "learn_i" se activa por cada nuevo patrón que se desea aprender y se puede ver también que la distancia entre cada vez que se activa "learn_i" se va distanciando, esto es debido a que la ONN cada vez le cuesta más aprender un nuevo patrón. Al final de todo el proceso de aprendizaje se realiza el proceso de test, durante este lo que se hace es enviar las imágenes correspondientes a los patrones y se captura el resultado.

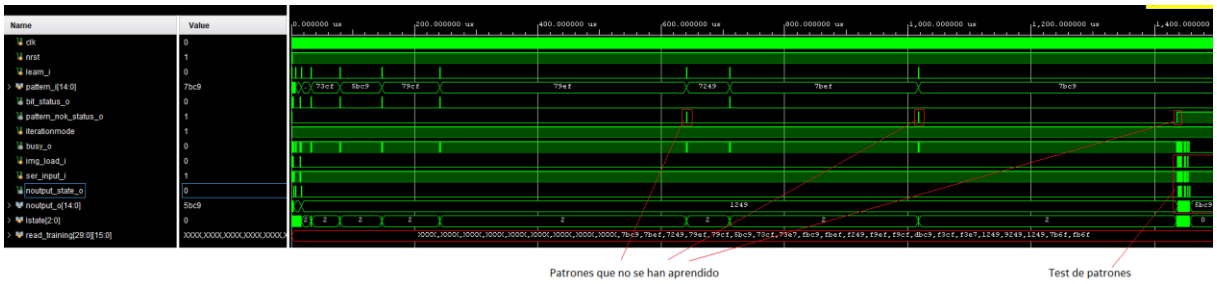


Figura 37: Simulación sistema completo 3

Por último, se ha sintetizado y se ha realizado una implementación del sistema para comprobar el nivel de ocupación en el dispositivo que se ha elegido y el resultado obtenido se muestra en la Figura 38.

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	6745	53200	12.68
FF	3756	106400	3.53
IO	39	125	31.20
BUFG	2	32	6.25

Figura 38: Recursos implementación ONN con capacidad de aprendizaje on-line con 15 neuronas.

5. Implementación física

En este apartado se explica cómo se ha implementado el setup para verificar el funcionamiento de la ONN con aprendizaje *on-line*. Para ello, se ha utilizado una placa de desarrollo de Digilent.

5.1. Setup experimental

El setup para la verificación del sistema consiste en una tarjeta Zybo Z7 7020 conectada a un PC a través del puerto USB. La tarjeta Zybo Z7 7020 tiene una FPGA de la serie ZYNQ de Xilinx en la que se ha cargado el *bitstream* correspondiente a la plataforma hardware diseñada para la prueba de nuestro sistema y la aplicación que corre en el procesador ARM incluido en la FPGA ZYNQ de la tarjeta.

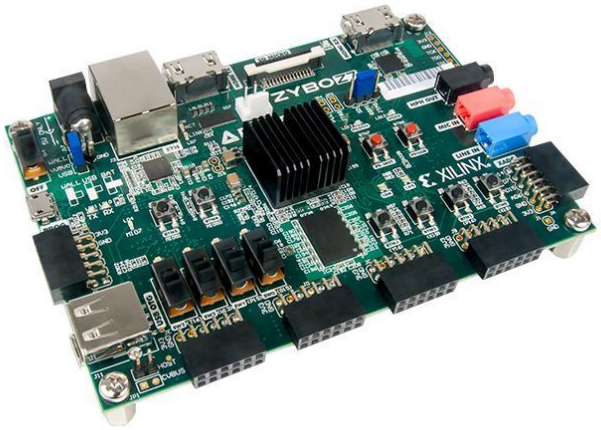


Figura 39: Tarjeta Zybo Z20 de Xilinx

Para acceder a las entradas y salidas de la ONN, se ha utilizado el ARM. La comunicación con el PC se ha realizado mediante un puerto serie, gestionado por el ARM.

El diagrama de bloques de este setup se muestra en la Figura 40.

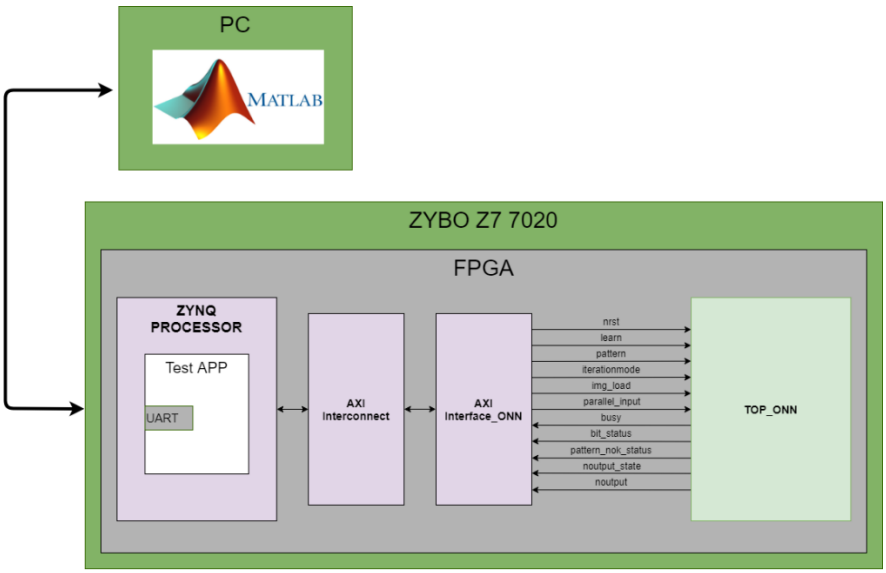


Figura 40: Diagrama de bloques del setup experimental.

Además del módulo TOP_ONN, que contiene la ONN con aprendizaje on-line, se ha implementado un bloque, AXI Interface ONN, para la comunicación entre el ARM y la ONN. Este bloque permite acceder a los puertos de la ONN desde el procesador ARM del ZYNQ de una manera sencilla.

El resto son módulos que se encuentran en el catálogo de Xilinx y que se han instanciados y configurados para esta aplicación. En la Figura 41 se muestra el diagrama de bloques del setup de prueba en la herramienta de Xilinx Vivado.

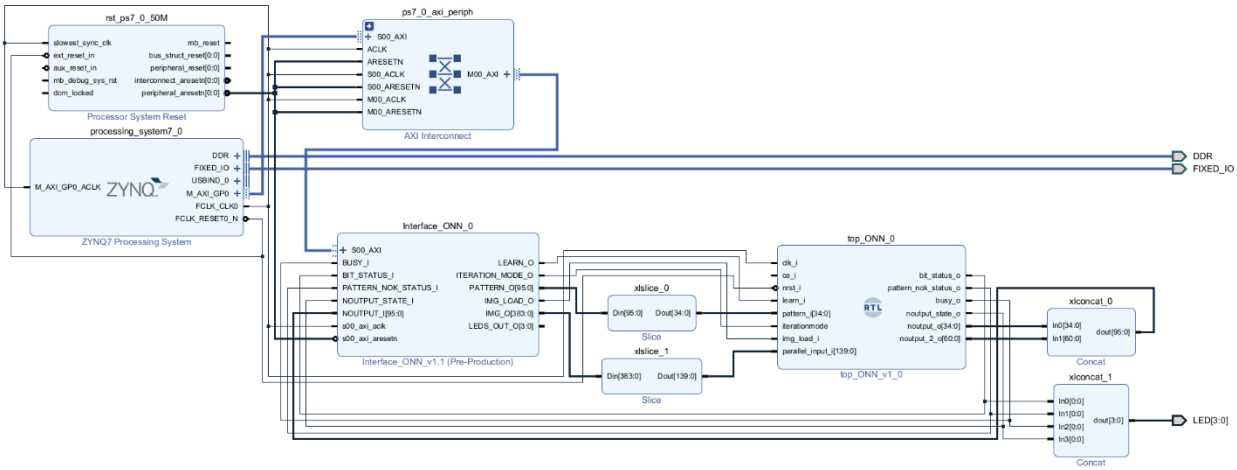


Figura 41: Diagrama de bloques de la plataforma hardware que se implementa en el FPGA.

El procesador ARM de la ZYNQ se comunica con el exterior mediante un puerto UART. Desde el PC se accede a este desde una herramienta desarrollada en MATLAB que permite el envío de patrones (imágenes en nuestra aplicación), así como recibir la respuesta de la ONN para poder analizarlos y compararlos con lo que se espera.

- **<u>Módulo TOP_ONN.</u>**

El módulo TOP_ONN es el que contiene el diseño del algoritmo de aprendizaje iterativo y la red neuronal ONN. Para proporcionar las imágenes de entrenamiento al módulo de aprendizaje iterativo, se ha propuesto enviar estas imágenes de forma paralelo a través del puerto "parallel_input_i". Una imagen de test está formada por tantos bits como neuronas tenga la ONN, multiplicada por el número de bits utilizados para definir el estado de cada neurona. Por ejemplo, para el caso de 15 neuronas y 4 bis de ancho para la definición del estado de cada neurona, da un total de 60 bits necesarios para definir una imagen. Por tanto, lo que se ha hecho es definir un bus paralelo de hasta 384 bits para poder transmitir las imágenes de test al módulo de aprendizaje iterativo.

Esta imagen en paralelo que le llega al módulo de aprendizaje iterativo necesita ser serializada para poderla introducir en la red de neuronas. Por este motivo, se ha modificado el módulo de captura "mod_data_capture" para que, al recibir la imagen en paralelo, la serialice y la envíe a la red neuronal.

La Figura 42 muestra un diagrama detallado de cómo está compuesto este módulo. Este diagrama es similar al mostrado en la Figura 31 pero con la modificación para capturar los datos de la imagen en paralelo.

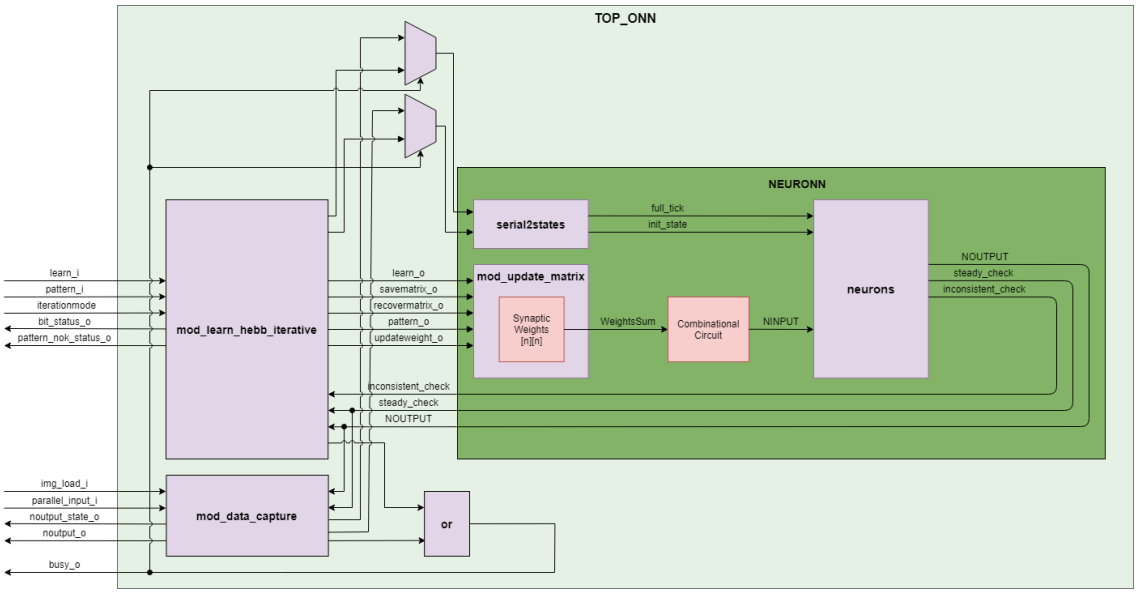


Figura 42: Diagrama de bloques TOP_ONN

A partir de este diseño, se realiza la síntesis e implementación que genera el fichero *bitstream*, el cual se utiliza para cargarse en el dispositivo.

- **<u>Módulo AXI Interface_ONN.</u>**

Este módulo permite acceder a los puertos de entrada / salida del módulo TOP_ONN mediante un bus AXI (Advanced eXtensible Interface) conectado al procesador. El Bus AXI es un protocolo desarrollado por ARM que permite interconectar de una manera estandarizada un procesador con módulos o periféricos.

El módulo Interface_ONN consiste en un banco de 32 registros de 32 bits cada uno de ellos, a los cuales se pueden acceder en modo lectura o escritura desde el procesador. Estos registros se han mapeado a puertos de entrada/salida del módulo y de esta manera se puede controlar el acceso a los puertos del módulo Top_ONN. En la Figura 43 Se muestra el detalle de la interconexión de este módulo. Se puede observar cómo se conecta su bus AXI S00_AXI al procesador mediante un módulo de interconexión. Se observa también los puertos de entrada y salida del bus Interface_ONN que se conectan principalmente al módulo TOP_ONN.

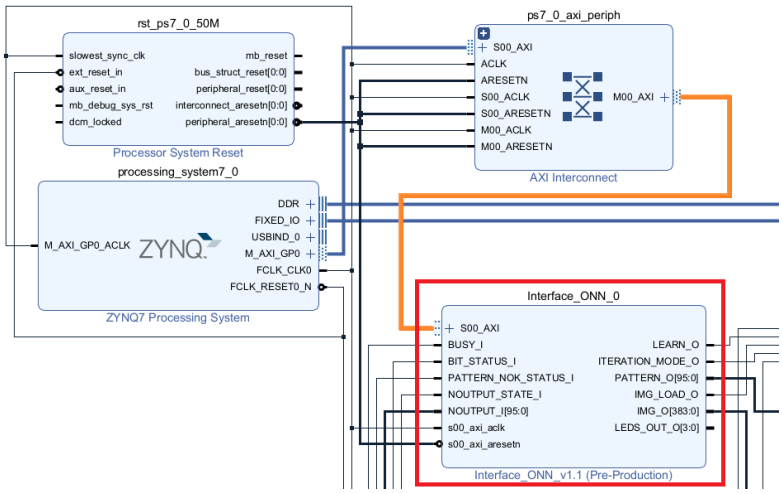


Figura 43: Interconexión del Módulo Interface_ONN.

- **<u>Código procesador Zynq</u>**

La aplicación software que se ha desarrollado para correr en el ARM se encarga de hacer de puente entre la ONN y el puerto serie que tiene disponible la tarjeta.

Para la implementación del código se ha utilizado el entorno Vitis de Xilinx, que es un entorno de programación que permite crear una aplicación y cargarla junto el *bitstream* en el procesador del Zynq. La Figura 44 muestra una captura de dicho entorno.

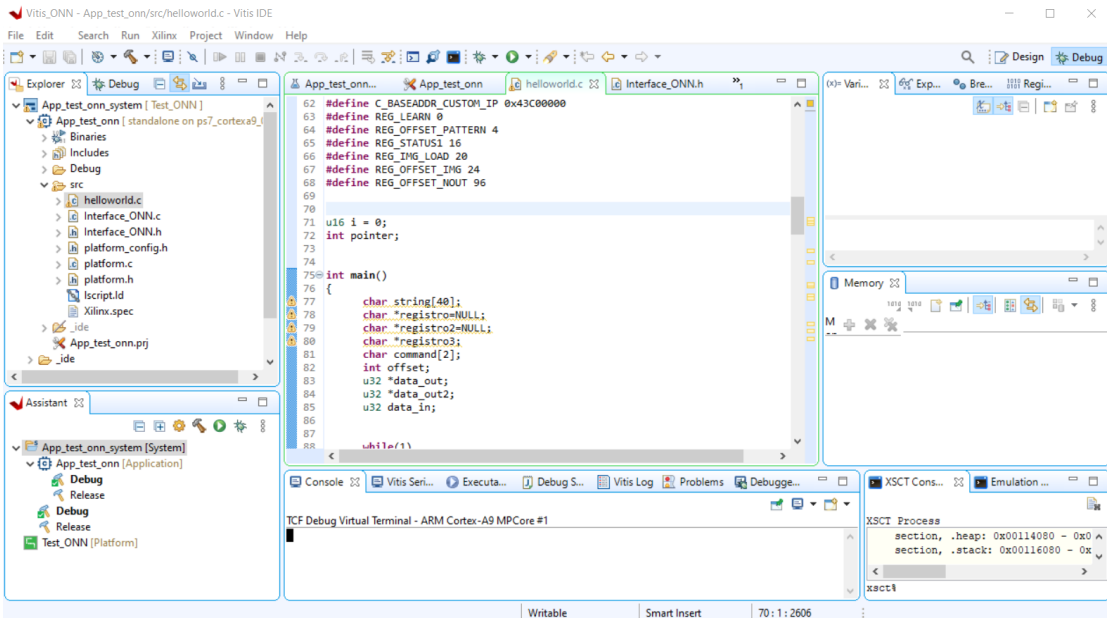


Figura 44: Aplicación Vitis de Xilinx

La aplicación que se ha implementado está continuamente a la espera de recibir comandos por el puerto serie, y en función de este se procede a escribir o leer a los registros correspondientes.

En la aplicación MATLAB que corre en el PC se han definido dos comandos:

- **<u>Comando Escribir (w)</u>**: Este comando permite escribir en el banco de registros del AXI Module y este a su vez está conectado al módulo TOP_ONN.

"w REG DATA", donde:

"w" indica que es el comando de escritura.

"REG" es el registro que se desea escribir.

"DATA" es el dato en formato hexadecimal de 32 bits que se va a escribir en el registro "REG".

Ejemplo: "w 8 0x76B6". Escribir en el registro 8 el dato 0x76B6.

- **<u>Comando LEER (r)</u>**: Este comando permite leer desde el banco de registros del AXI Module y, por tanto, a su vez permite acceder al módulo TOP_ONN.

"r REG", donde:

"r" indica que es el comando de lectura.

"REG" es el registro que se desea acceder.

Ejemplo: "r 4". Leer el registro 4.

5.2. Aplicación de test en MATLAB

Para acceder al módulo TOP_ONN desde el exterior se ha creado una aplicación sobre MATLAB que se conecta mediante puerto serie a la tarjeta Zybo Z7.

Se parte de un conjunto de matrices asociadas a imágenes de entrenamiento. Estas matrices requieren de una adaptación del formato para que se pueda enviar a la tarjeta. Es por esto por lo que se han creado una serie de funciones que permite estas conversiones.

- Función **ResultNok = LearnMatrix(InputMatrix,device)**

Esta función envía la imagen al dispositivo. Devuelve el resultado del aprendizaje en ResultNok, Siendo 0 para indicar que la imagen de entrada se ha aprendido y 1 indicando que no se ha aprendido.

- Función **MatrixOut = TestMatrix(InputMatrix,Ncol,Nrow,NbitsState,device)**

Esta función toma la matriz InputMatrix con el formato de Ncol columnas, Nrow filas, con NbitsState numero de bits para cada estado de la matriz y la envía al dispositivo device para testear. Devuelve en MatrixOut una matriz con el resultado de la prueba.

Descripción del proceso de test

Partiendo de todas estas herramientas, se ha implementado el setup de prueba que realiza lo siguiente:

- Lectura de las imágenes de entrenamiento desde un fichero.
- Preparación de las imágenes para darles un formato que permita enviarlas por el puerto serie.
- Envío de las imágenes de entrenamiento a través del puerto serie a la tarjeta Zybo Z7 para su aprendizaje.
- Envía de imágenes de test para inferencia en la ONN. Estas pueden ser tanto las propias imágenes de entrenamiento como versiones de ellas con ruido.
- Recepción de las imágenes tras la inferencia.
- Finalmente, muestra gráficamente el conjunto de imágenes de entrenamiento y el conjunto de imágenes recibidas desde el módulo.

5.3. Pruebas con la versión 5x3

Se han realizado pruebas con imágenes de entrada de 5x3 pixeles. Las prueban consisten en enviar los patrones al módulo ONN para que los aprenda, luego se testean, se comprueba la salida obtenida y se compara con los resultados obtenidos en las simulaciones. Para las pruebas, se han utilizado patrones de entrada que representan los números desde el 0 al 9. Estos patrones son los mismos que se han utilizado en las simulaciones en el Capítulo 4 y que volvemos a mostrar en la Figura 45.

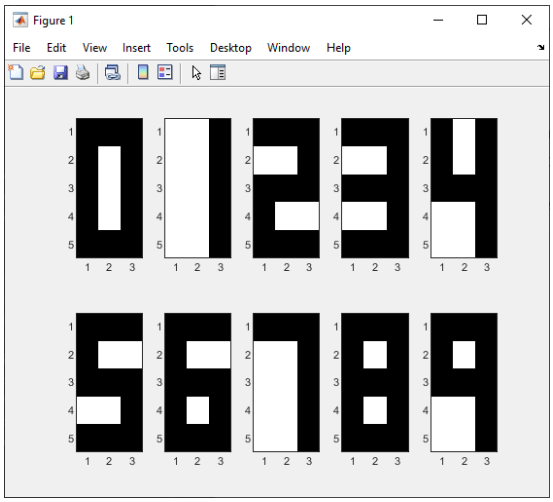


Figura 45: Patrones de entrenamiento 5x3

Una vez finalizado el proceso de aprendizaje, los mismos patrones se envían de nuevo para ser procesados por la ONN. Se ha obtenido el resultado mostrado en la Figura 46.

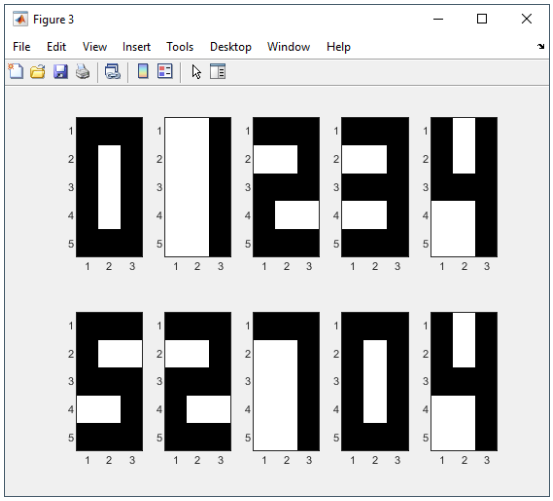


Figura 46: Resultado aprendizaje red ONN 5x3

Se puede observar que el sistema ha sido capaz de aprender 7 patrones de los 10 lo que indica que el resultado obtenido es idéntico al obtenido en la fase de simulación.

Esta prueba era interesante para comparar los resultados del sistema físico con las simulaciones. Sin embargo, almacenar 7 patrones con 15 neuronas, aunque es interesante desde el punto de vista de la potencia del algoritmo, es poco relevante desde el punto de vista de su aplicación práctica. Con tantos patrones, su capacidad de recuperarlos desde versiones ruidosas de los mismos va a ser muy limitada. Por ello, el sistema se ha probado con conjuntos reducidos de patrones de forma que experimentalmente comprobemos su capacidad para recuperar patrones.

5.3.1. Test 5x3 con 2 patrones

Este test consiste en el aprendizaje de dos patrones correspondientes al “0” y al “1”. Ver Figura 47.

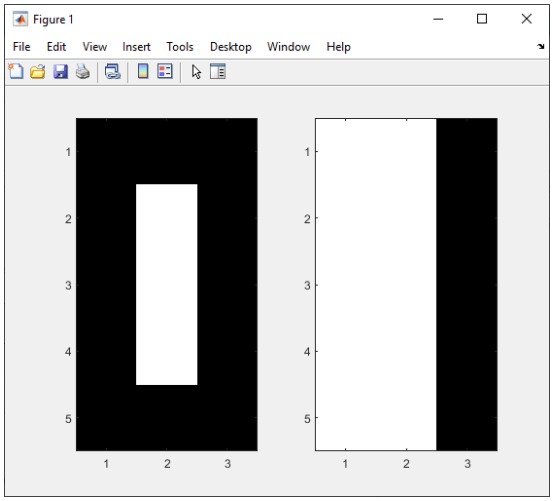


Figura 47: Test 5x3 con 2 patrones

Estos se envían a la ONN y luego se testean con los patrones de entrada que van desde el “0” al “9”. El resultado obtenido se puede ver la Figura 48. Se puede observar que, en los resultados, los únicos patrones reconocidos son los correspondientes al “0” y al “1”.

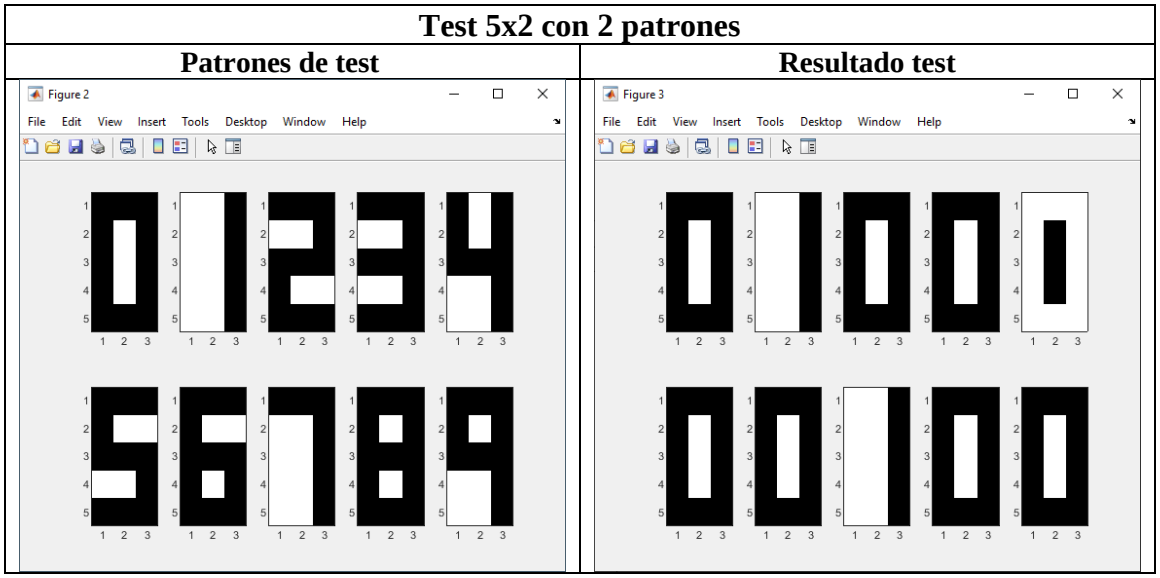
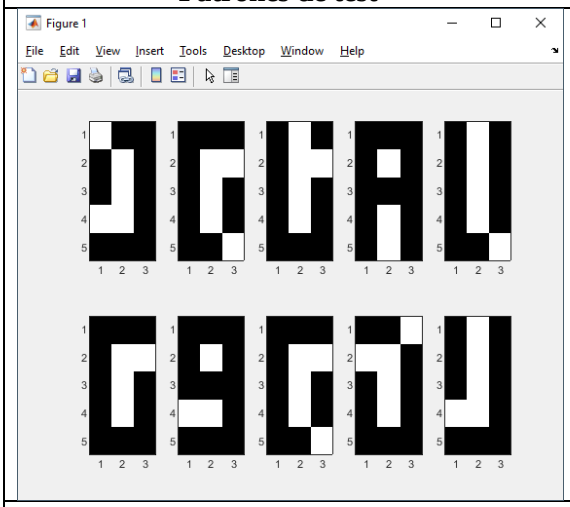
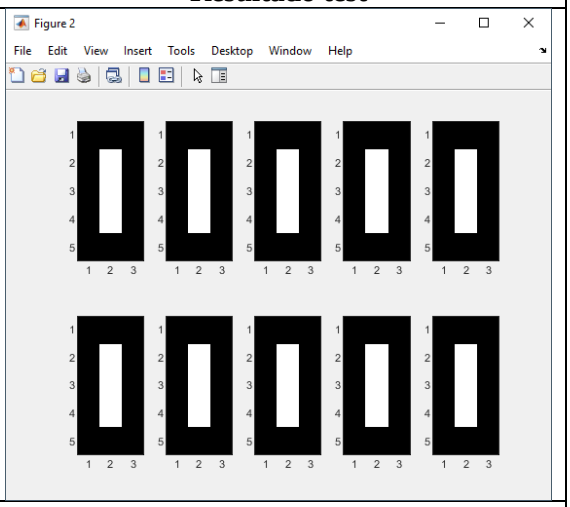


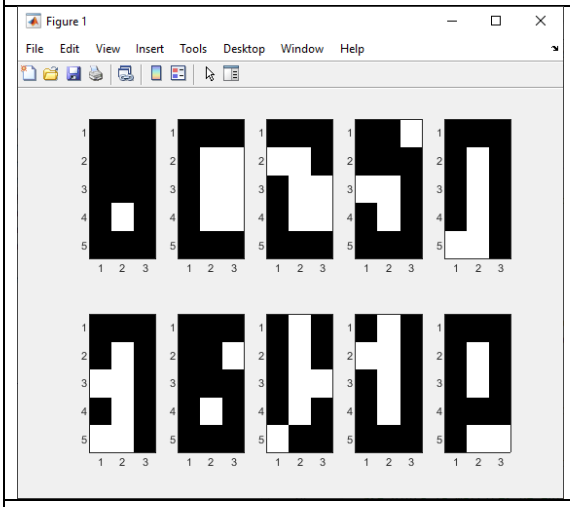
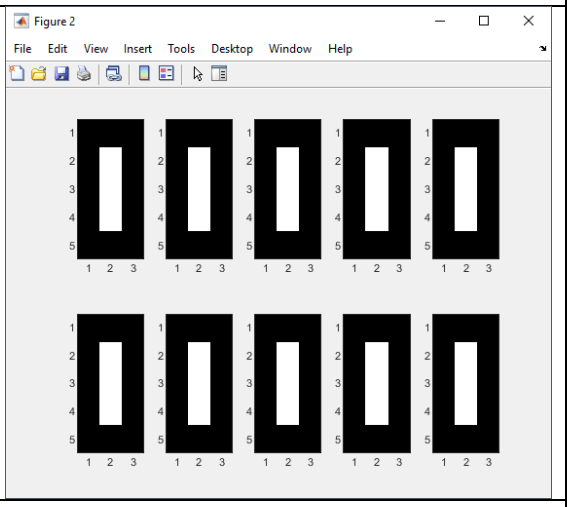
Figura 48: Resultado test 5x3 con 2 patrones.

5.3.1.1. Test 5x2 con 2 patrones: Test ruido con 2 pixeles

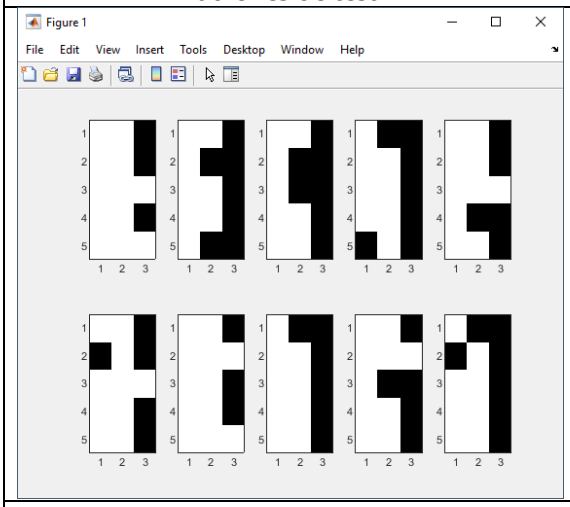
En esta sección se realiza los test de ruido en los que se comprueba la capacidad de recuperación de la ONN. En este caso, se han generado los patrones correspondientes a los aprendidos con ruido aleatorio, luego se han enviado a la ONN y se ha comprobado la cantidad de estos que es capaz de recuperar.

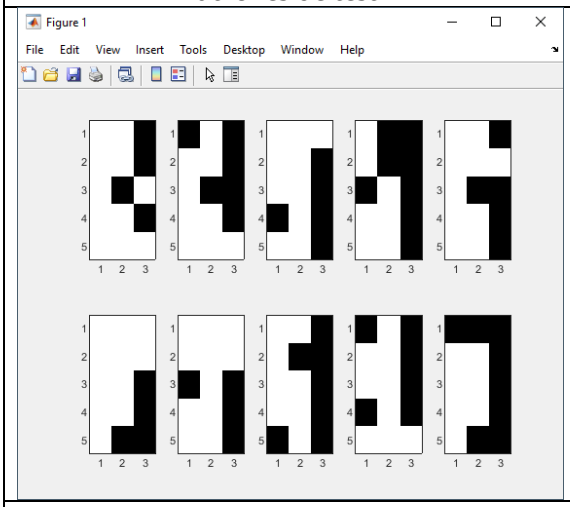
Resultados patrón "0":

Test 5x2 con 2 patrones: Patrón "0" con ruido 2 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
10 de 10	100 %

Test 5x2 con 2 patrones: Patrón "0" con ruido 3 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
10 de 10	100 %

Resultados patrón “1”:

Test 5x2 con 2 patrones: Patrón “1” con ruido 2 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
8 de 10	80 %

Test 5x2 con 2 patrones: Patrón “1” con ruido 3 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
8 de 10	80 %

Resumen resultados:

Tabla 4: Resumen resultados test ONN 5x3 con 2 patrones.

Patrón testeado	Número de errores introducidos en los patrones			
	2 píxeles		3 píxeles	
	Recuperados	Porcentaje	Recuperados	Porcentaje
Patrón "0"	10 de 10	100%	10 de 10	100%
Patrón "1"	8 de 10	80%	8 de 10	80%
Todos los test se han realizado con 10 patrones diferentes generados de manera aleatoria				

5.3.2. Test 5x3 con 3 patrones

En este caso el test se realiza con los patrones correspondientes al “0”, “1” y al “2”. Ver Figura 47.

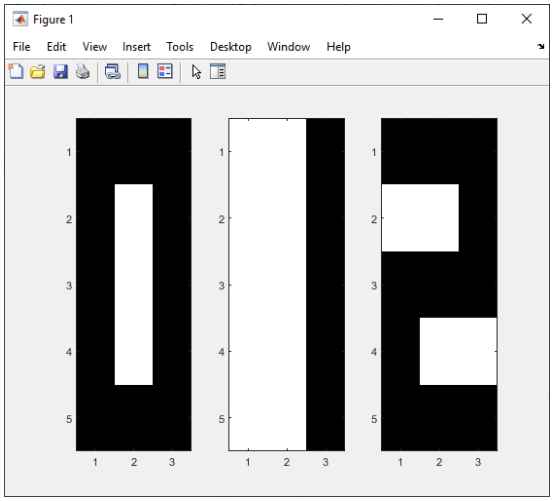


Figura 49: Test 5x3 con 3 patrones

Se testea la ONN con todos los patrones y los resultados son los mostrados en la Figura 50

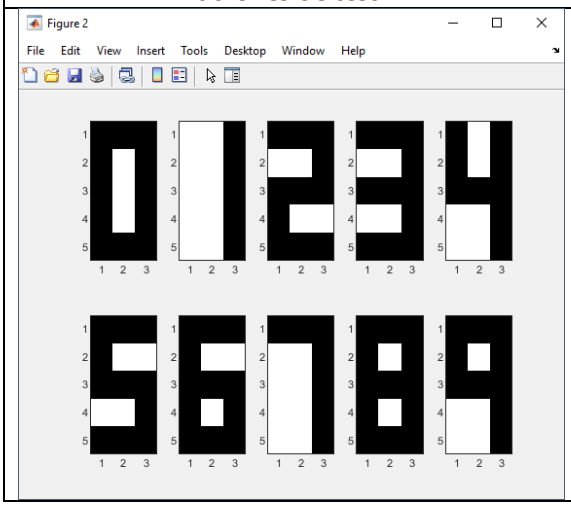
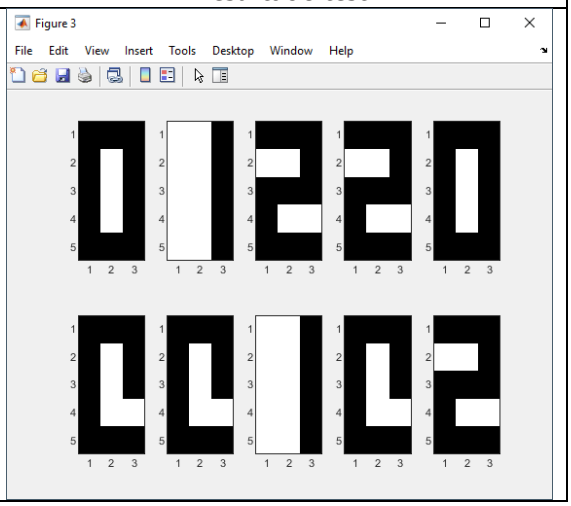
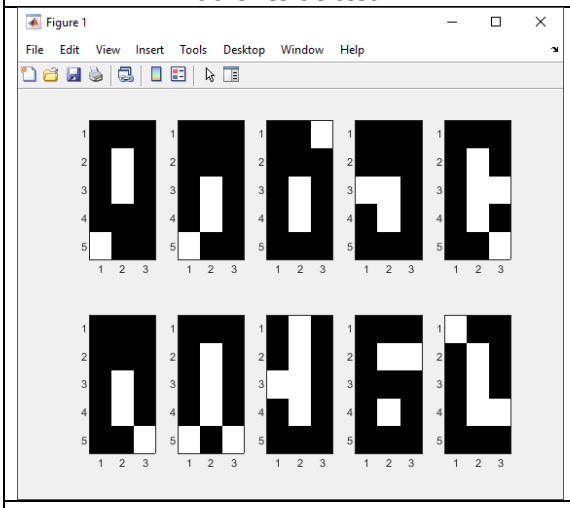
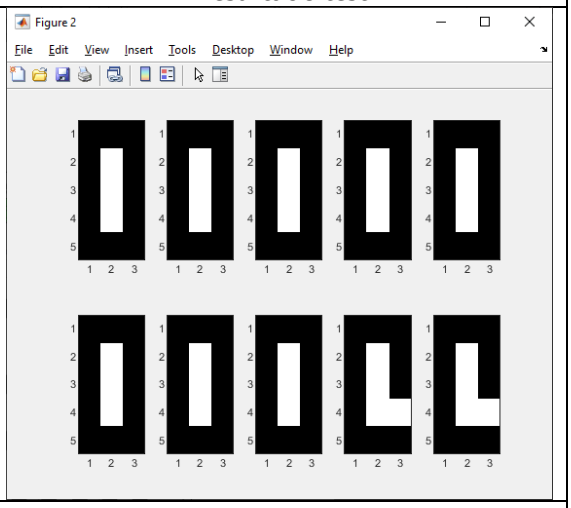
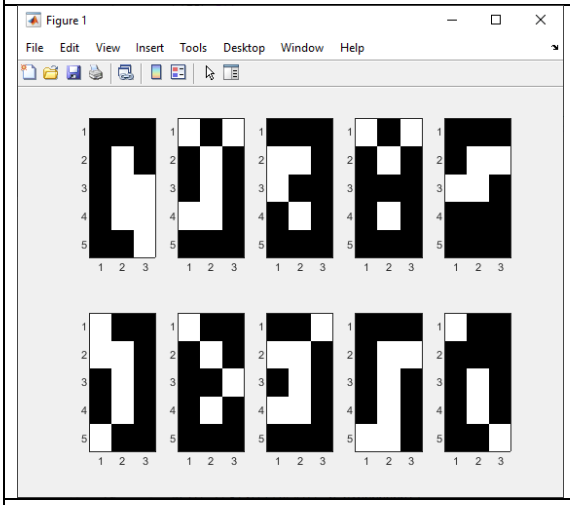
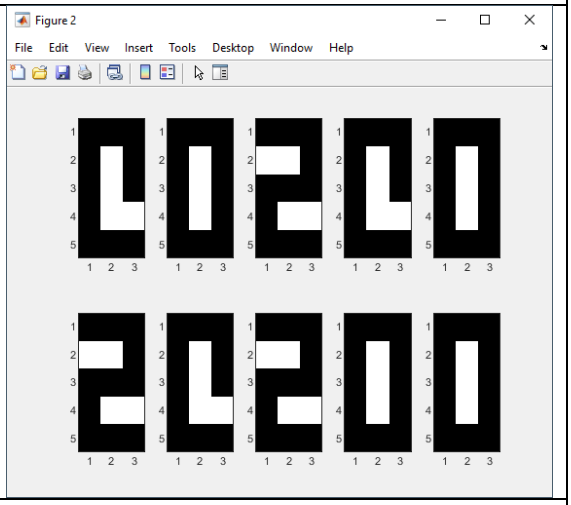
Test 5x2 con 3 patrones	
Patrones de test	**Resultado test**
	

Figura 50: Resultado test 5x3 con 3 patrones.

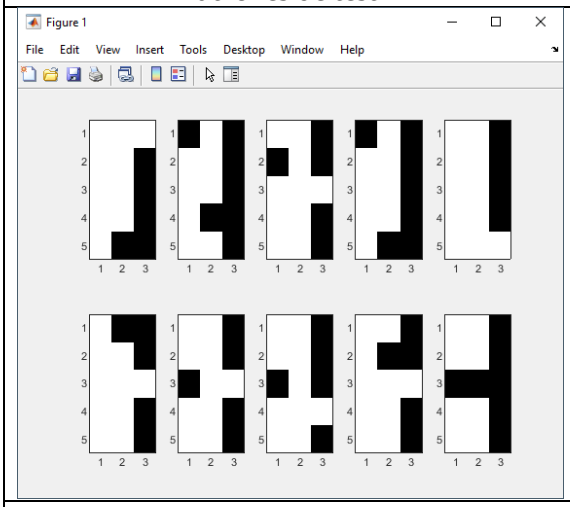
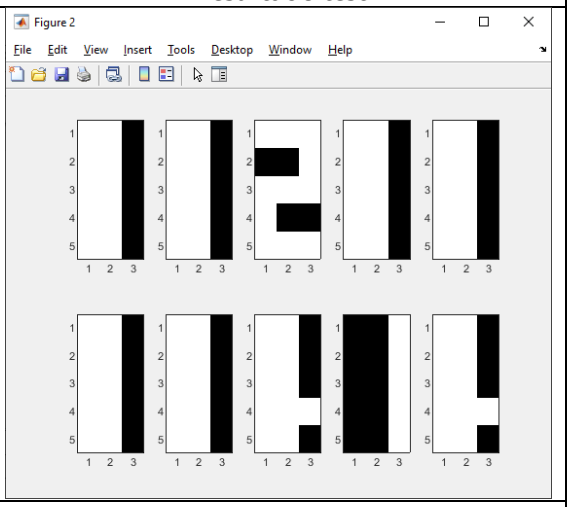
5.3.2.1. Test 5x2 con 3 patrones: Test ruido con 2 pixeles

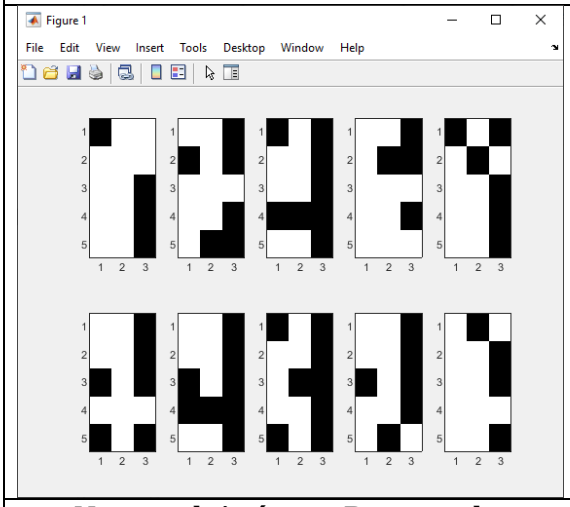
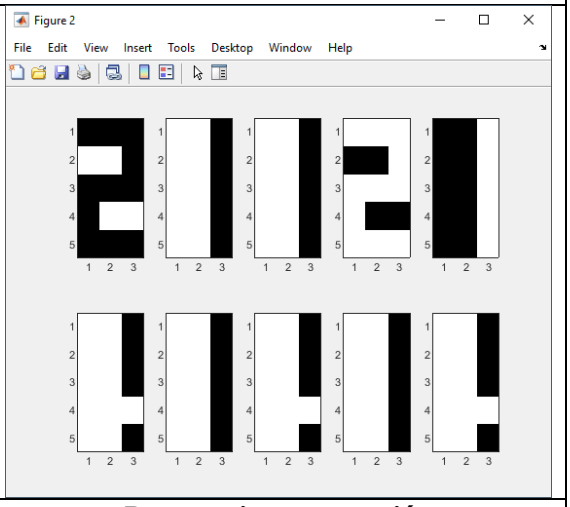
Resultados patrón "0":

Test 5x2 con 3 patrones: Patrón "0" con ruido 2 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
8 de 10	80 %

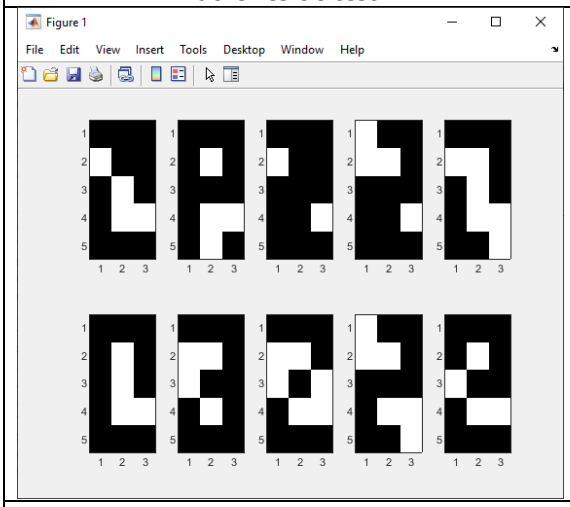
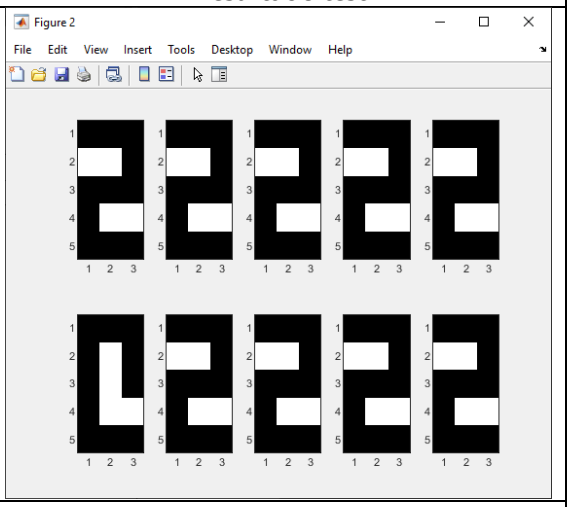
Test 5x2 con 3 patrones: Patrón "0" con ruido 3 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
4 de 10	40 %

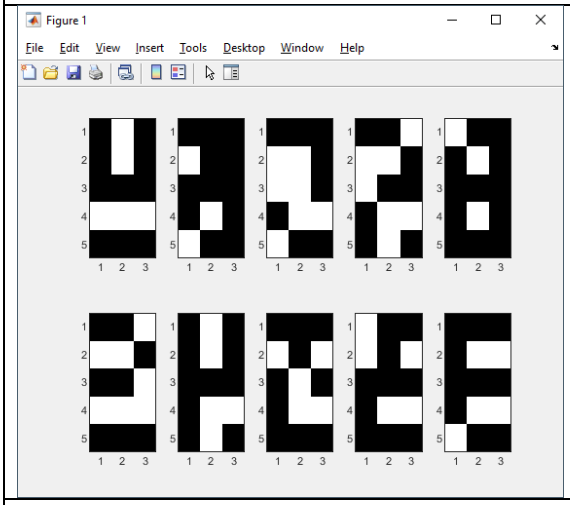
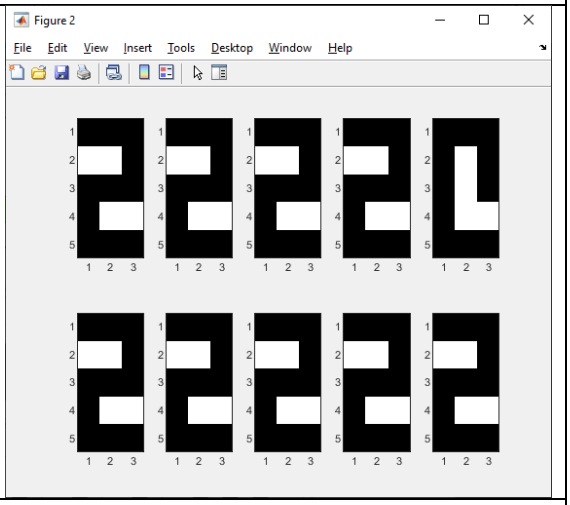
Resultados patrón "1":

Test 5x2 con 3 patrones: Patrón "1" con ruido 2 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
7 de 10	70 %

Test 5x2 con 3 patrones: Patrón "1" con ruido 3 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
5 de 10	50 %

Resultados patrón "2":

Test 5x2 con 3 patrones: Patrón "2" con ruido 2 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
9 de 10	90 %

Test 5x2 con 3 patrones: Patrón "2" con ruido 3 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
9 de 10	90 %

Resumen resultados:

Tabla 5: Resumen resultados test ONN 5x3 con 3 patrones.

Patrón testeado	Número de errores introducidos en los patrones			
	2 píxeles		3 píxeles	
	Recuperados	Porcentaje	Recuperados	Porcentaje
Patrón "0"	8 de 10	80%	4 de 10	40%
Patrón "1"	7 de 10	70%	5 de 10	50%
Patrón "2"	9 de 10	90%	9 de 10	90%
Todos los test se han realizado con 10 patrones diferentes generados de manera aleatoria				

Comparativa entre test con 2 y 3 patrones:

En la Tabla 6 se muestra un resumen comparativo entre los test realizados con 2 y 3 patrones. En este se muestra una media de los resultados obtenidos en ambos casos. Como se puede observar, el resultado con 2 patrones es del 90% de recuperación, tanto con la introducción de 2 pixeles como 3 pixeles de errores. En el caso de 3 patrones aprendidos, se obtiene un resultado de 80% y 60% respectivamente para 2 y 3 pixeles de errores. Estos resultados son coherentes, ya que, a mayor número de patrones aprendidos, se espera un empeoramiento de la capacidad de recuperación.

Tabla 6: Comparativa test 2 y 3 patrones.

Comparativa entre test con 2 y 3 patrones		
Test Setup	Número de errores introducidos en los patrones	
	2 píxeles	3 píxeles
	Porcentaje recuperación	Porcentaje recuperación
Test con 2 patrones	90%	90%
Test con 3 patrones	80%	60%

5.4. Pruebas con versión 7x5

En esta sección se exponen los resultados obtenidos con la versión de ONN de 7x5 neuronas. Los test presentados en este punto van enfocados a conocer la capacidad de recuperación que tiene la ONN frente a patrones con ruido y por otro lado se expone un test con objeto de determinar el máximo número de patrones que es capaz de almacenar la ONN. Para estos test se van a utilizar patrones correspondientes a las letras del abecedario que se muestran en la Figura 51.

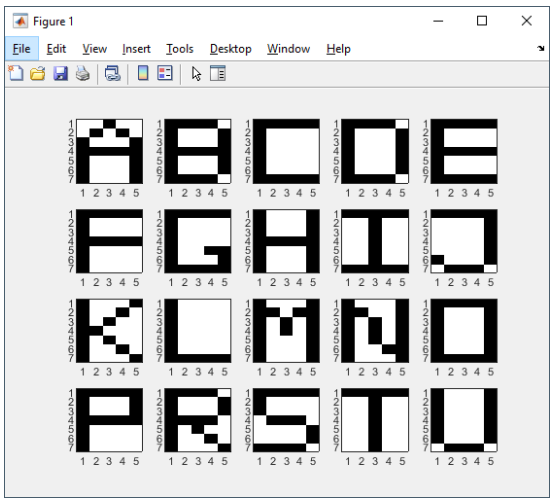


Figura 51: Patrones test ONN 7x5.

5.4.1. Test 7x5 con 4 patrones

El test consiste en aprender el patrón correspondiente a las letras “A”, “B”, “C” y “D”. Estos patrones se envían a la ONN y, posteriormente, se testea con 10 patrones correspondientes desde la “A” a la “J” para comprobar el comportamiento frente a estos. En la Figura 52 se puede observar en la figura de la derecha que los patrones “A”, “B”, “C” y “D” se han aprendido correctamente, para el resto de los patrones la ONN ha proporcionado el resultado que más se aproxima. Por ejemplo, en el caso de la “G” y “E” que se obtiene la “C” o en el caso de la “H” que se obtiene una “A”.

Test 7x5 con 4 patrones

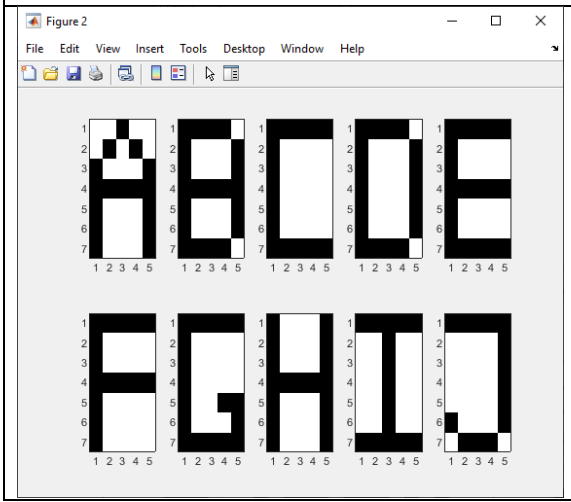
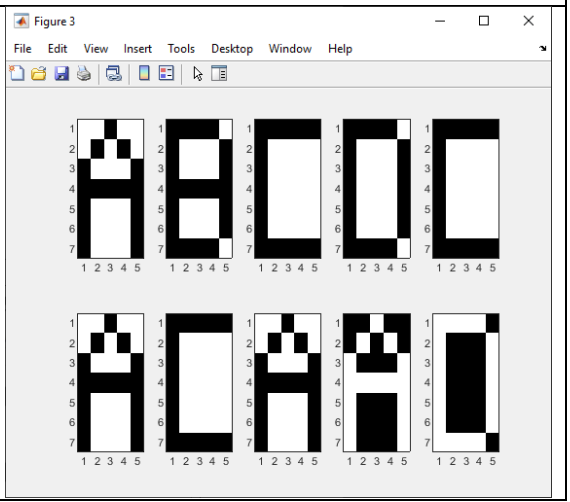
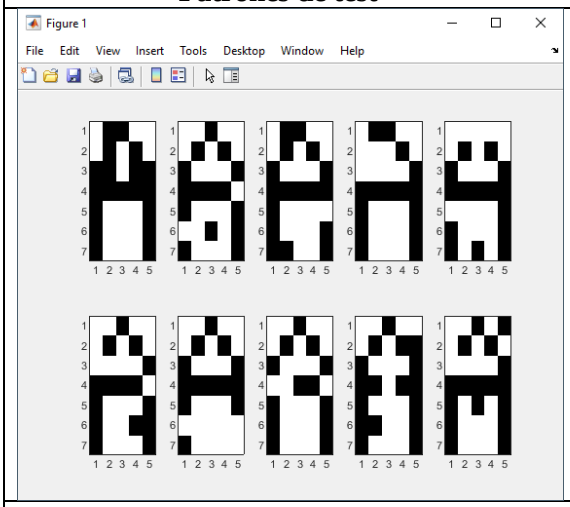
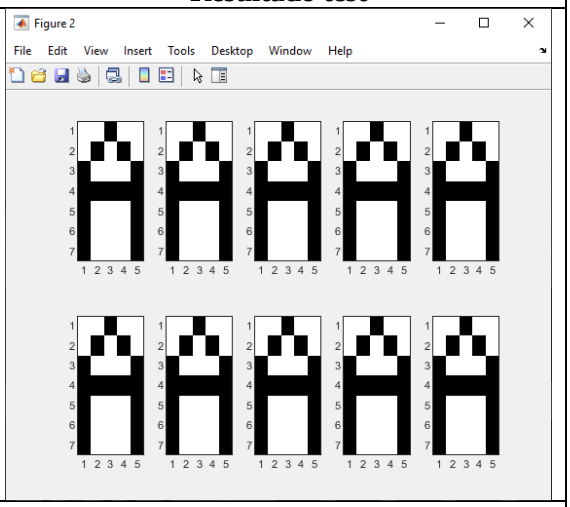
Patrones de test de entrada	**Resultado del test**
	

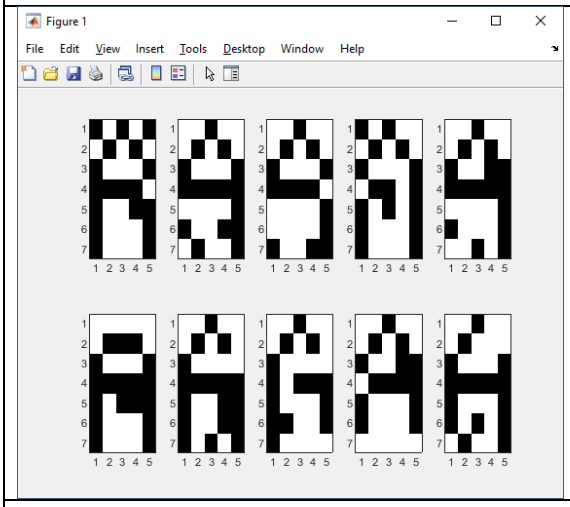
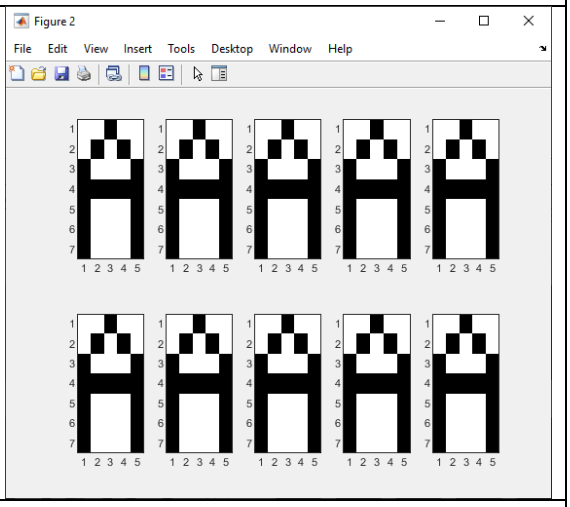
Figura 52: Test 7x5 con 4 patrones.

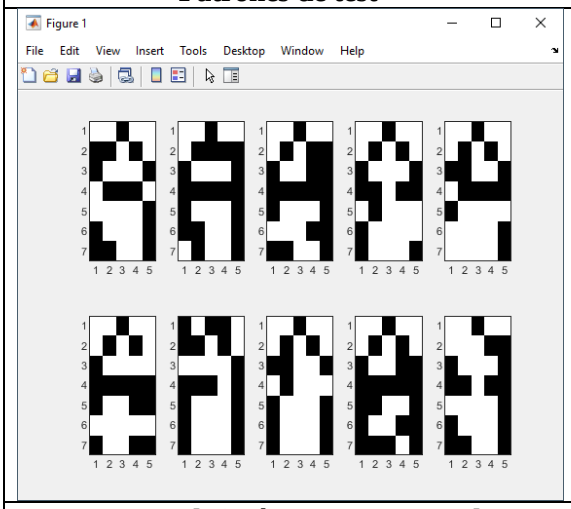
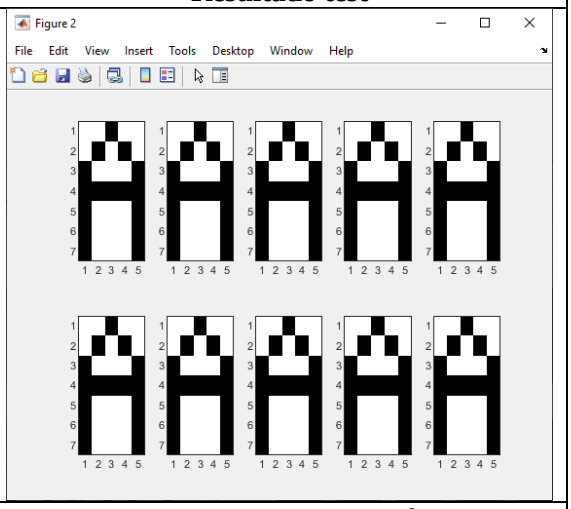
5.4.1.1. Test 7x5 con 4 patrones: Test ruido

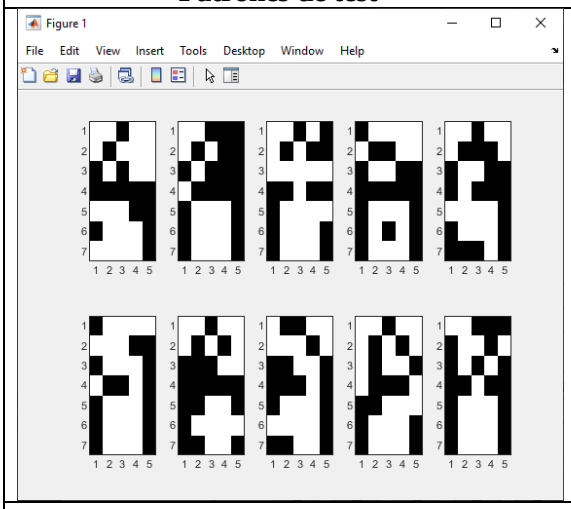
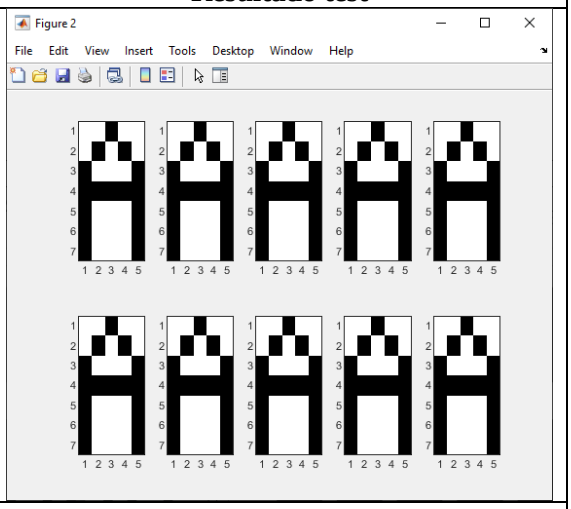
En este punto se comprueba la capacidad de recuperación de la ONN frente a patrones con ruido. Para cada uno de los patrones aprendidos, se generan patrones con cuatro niveles diferente de ruido, desde 3 pixeles hasta 6 pixeles. Los patrones con ruido son generados mediante una función que genera valores pseudoaleatorios.

Resultados patrón "A":

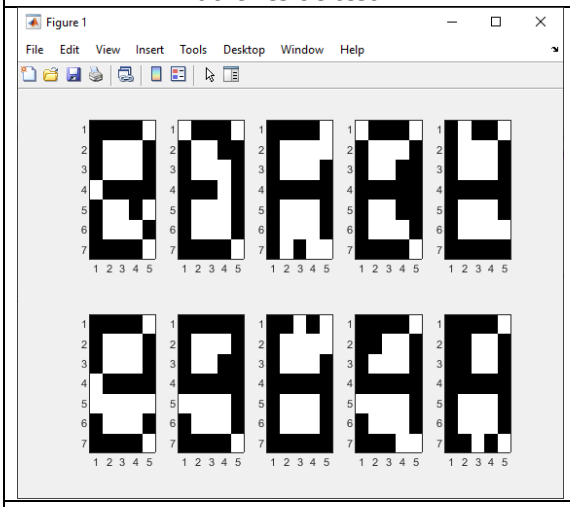
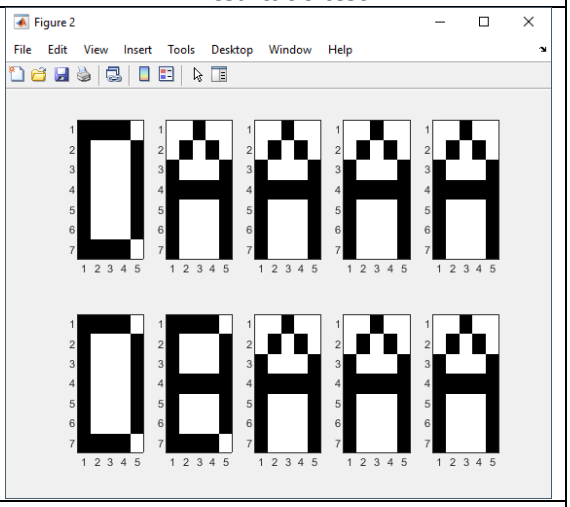
Test 7x5 con 4 patrones: Patrón "A" con ruido 3 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
10 de 10	100 %

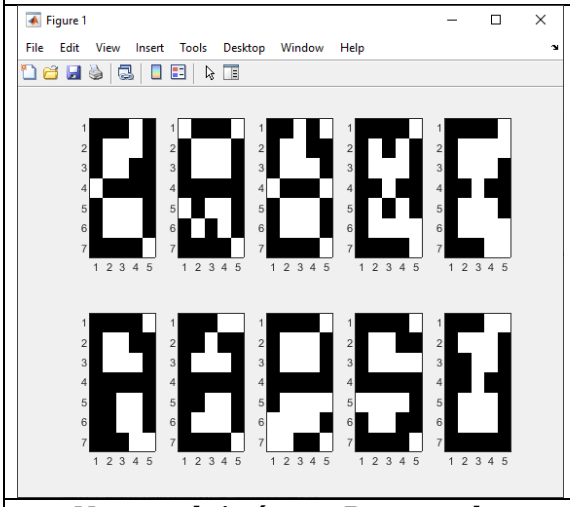
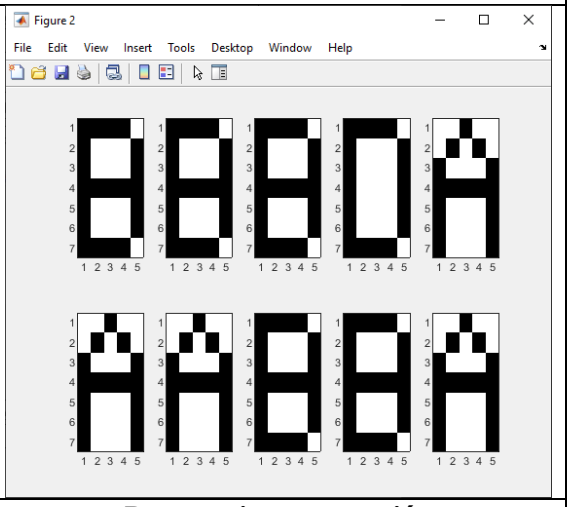
Test 7x5 con 4 patrones: Patrón "A" con ruido 4 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
10 de 10	100 %

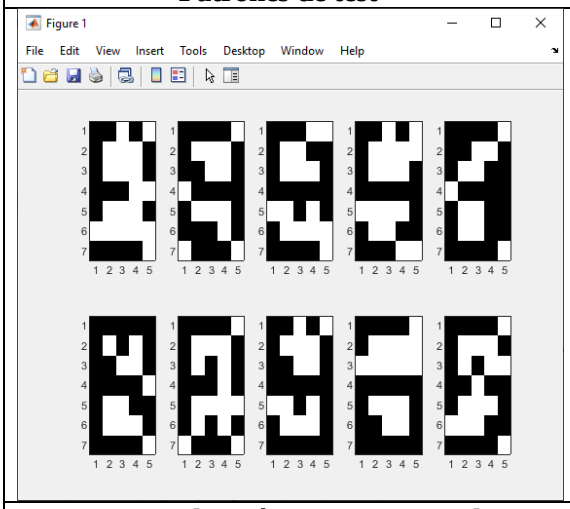
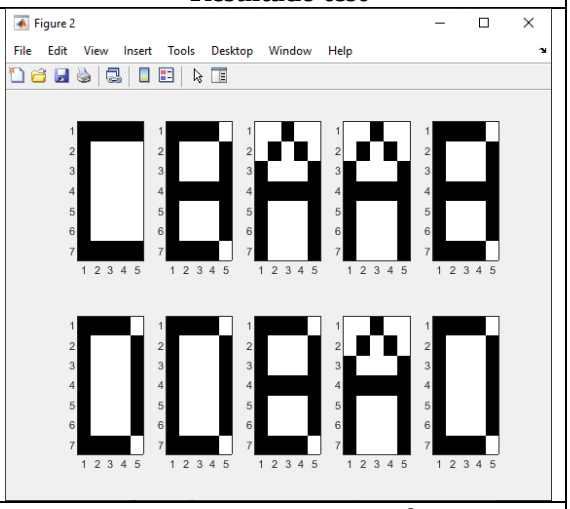
Test 7x5 con 4 patrones: Patrón “A” con ruido 5 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
10 de 10	100 %

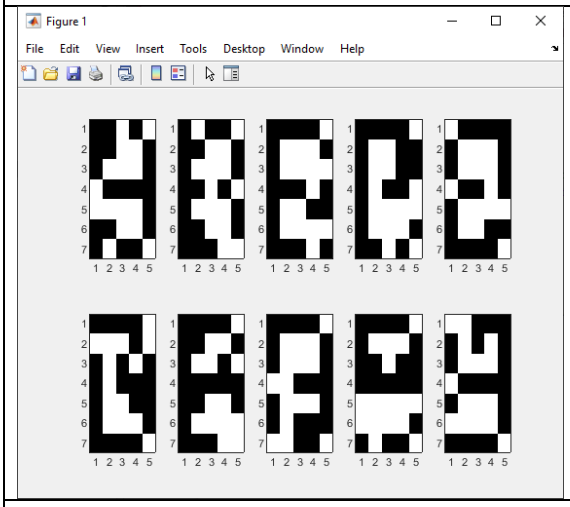
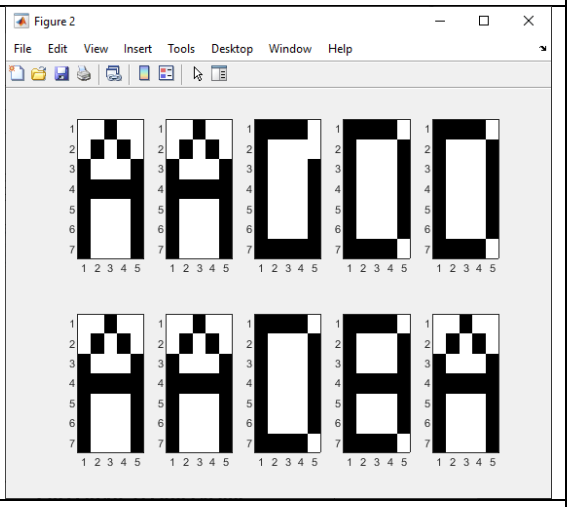
Test 7x5 con 4 patrones: Patrón “A” con ruido 6 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
10 de 10	100 %

Resultados patrón "B":

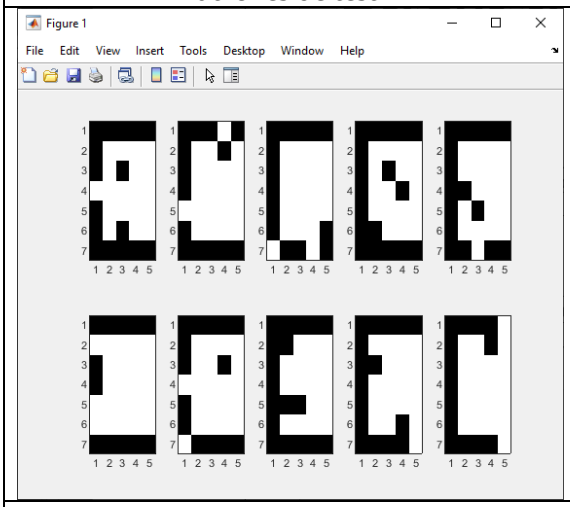
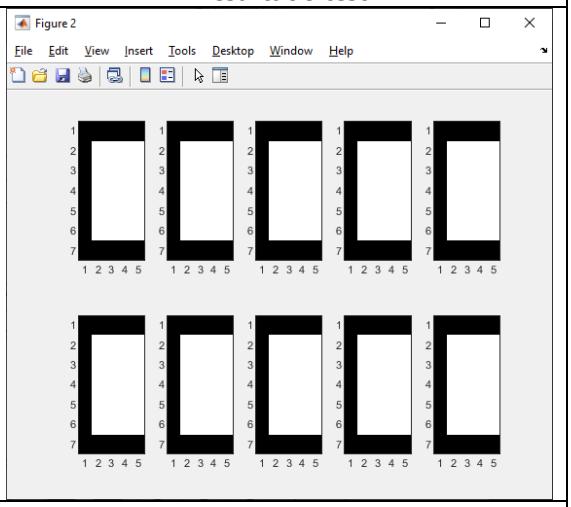
Test 7x5 con 4 patrones: Patrón "B" con ruido 3 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
1 de 10	10 %

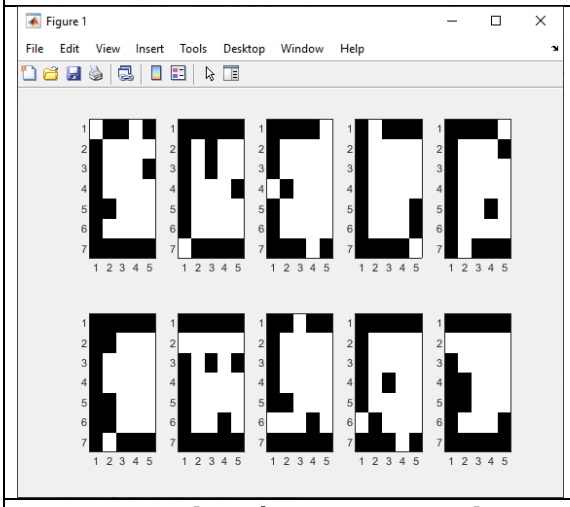
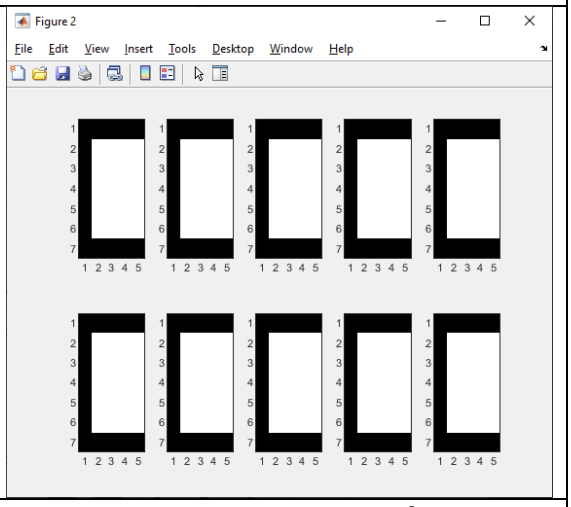
Test 7x5 con 4 patrones: Patrón "B" con ruido 4 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
5 de 10	50 %

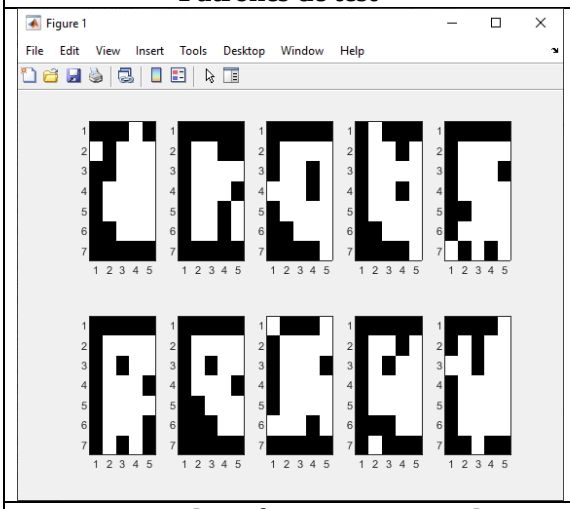
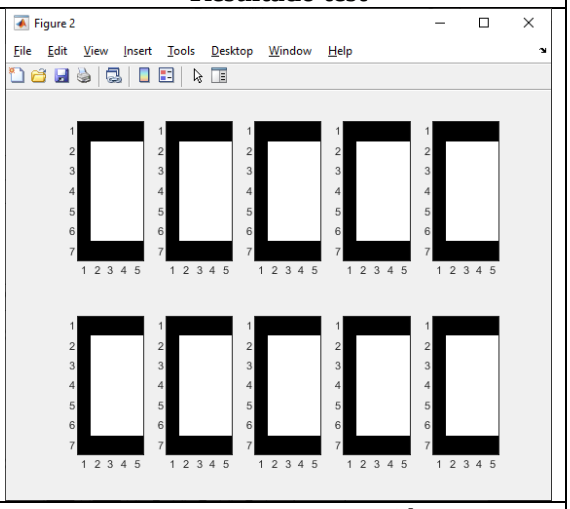
Test 7x5 con 4 patrones: Patrón “B” con ruido 5 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
3 de 10	30 %

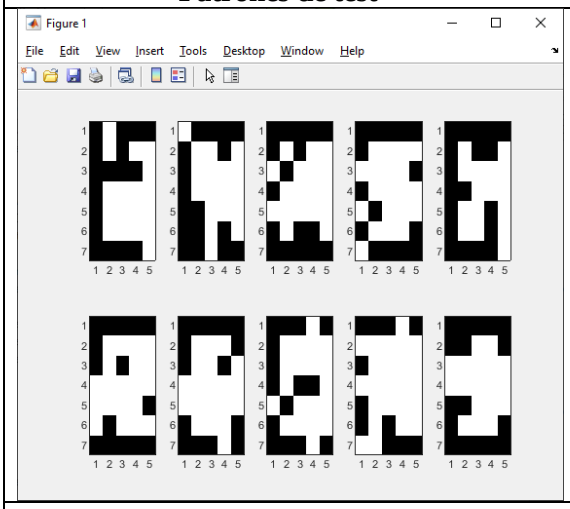
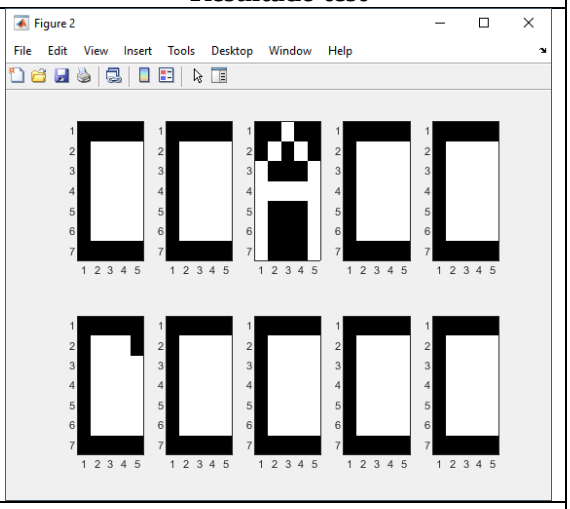
Test 7x5 con 4 patrones: Patrón “B” con ruido 6 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
1 de 10	10 %

Resultados patrón "C":

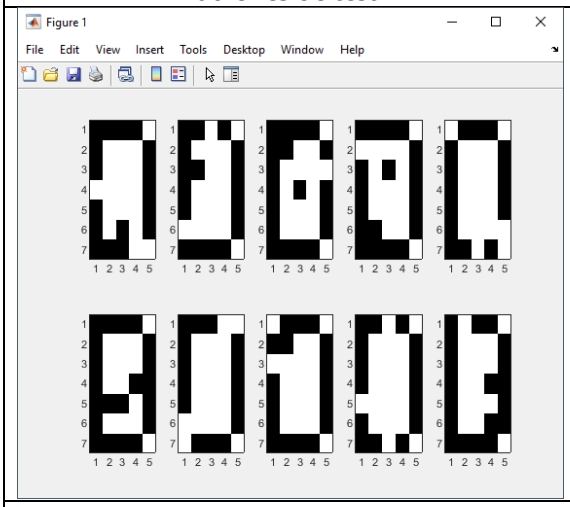
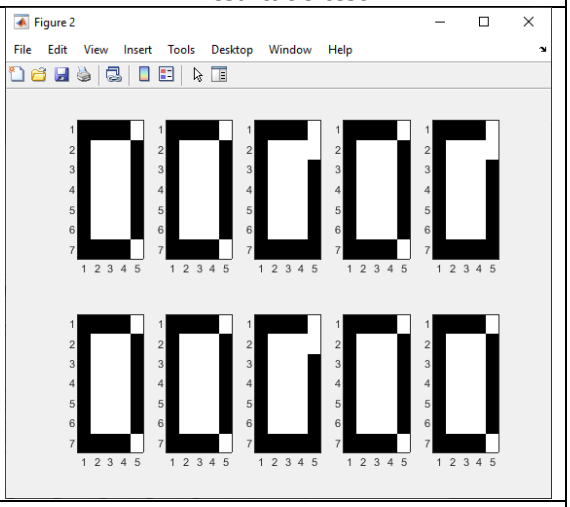
Test 7x5 con 4 patrones: Patrón "C" con ruido 3 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
10 de 10	100 %

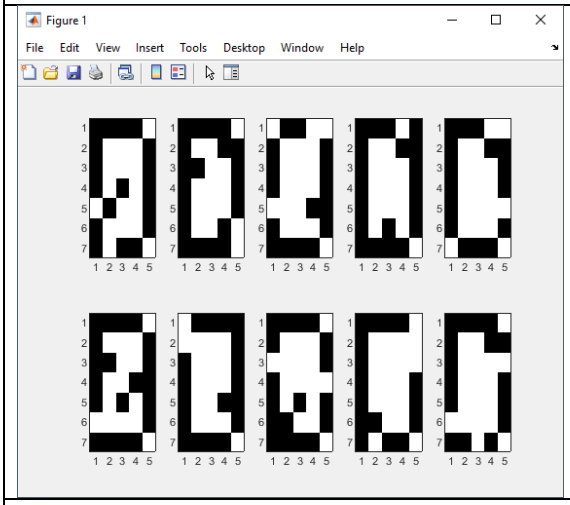
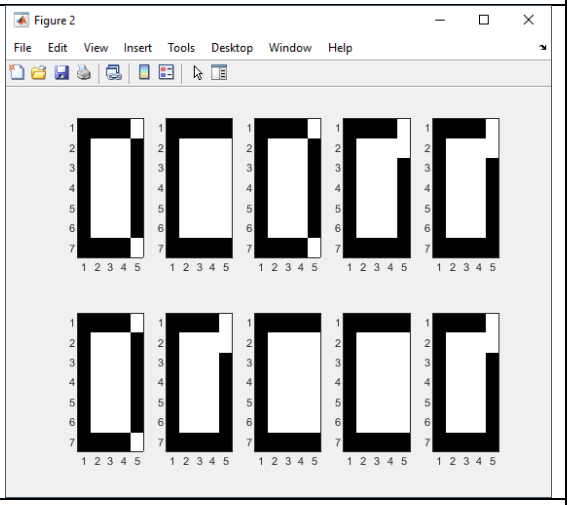
Test 7x5 con 4 patrones: Patrón "C" con ruido 4 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
10 de 10	100 %

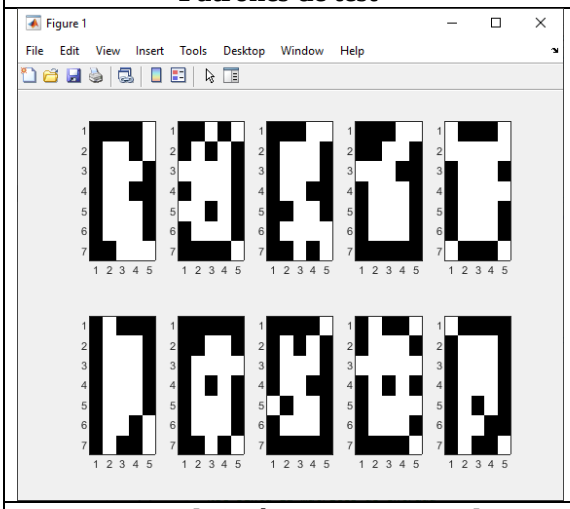
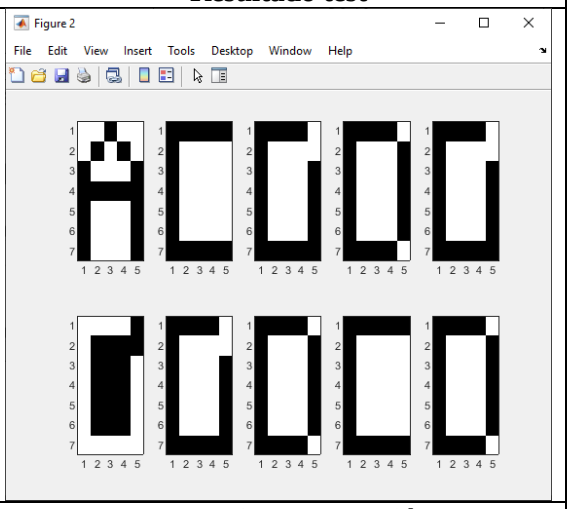
Test 7x5 con 4 patrones: Patrón “C” con ruido 5 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
10 de 10	100 %

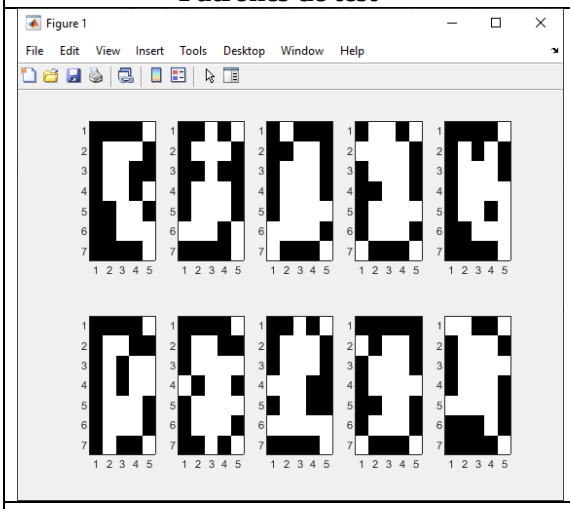
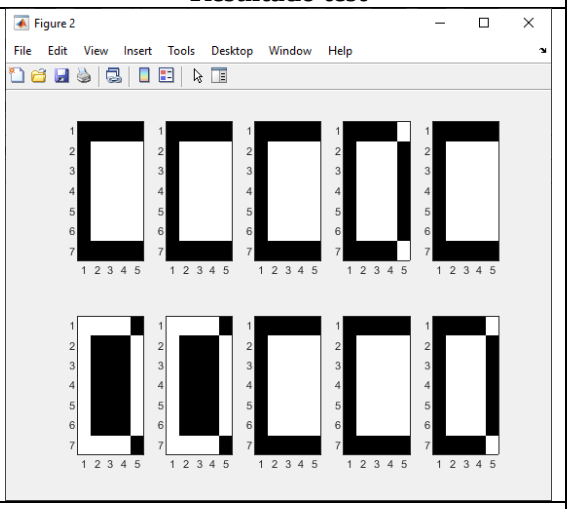
Test 7x5 con 4 patrones: Patrón “C” con ruido 6 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
9 de 10	90 %

Resultados patrón "D":

Test 7x5 con 4 patrones: Patrón "D" con ruido 3 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
10 de 10	100 %

Test 7x5 con 4 patrones: Patrón "D" con ruido 4 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
3 de 10	30 %

Test 7x5 con 4 patrones: Patrón “D” con ruido 5 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
3 de 10	30 %

Test 7x5 con 4 patrones: Patrón “D” con ruido 6 pixeles	
Patrones de test	**Resultado test**
	
Numero de imágenes Recuperadas	**Porcentaje recuperación**
4 de 10	40 %

Resumen de los resultados:

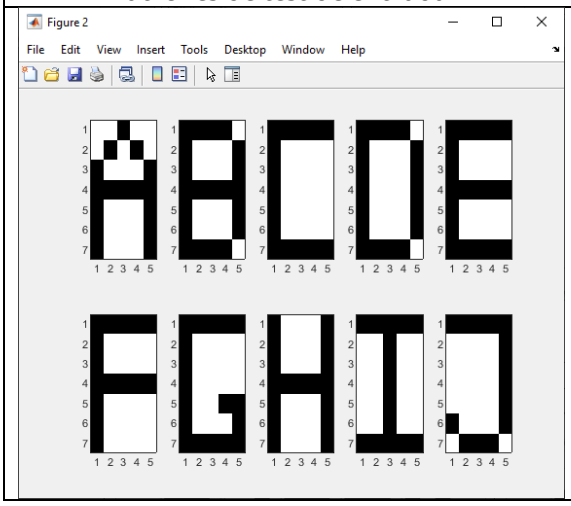
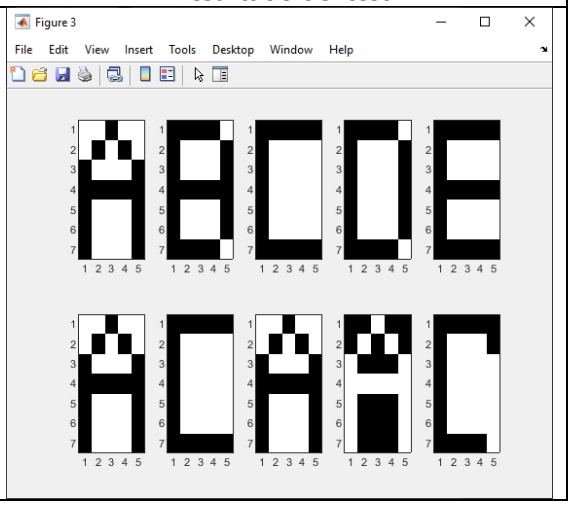
Patrón testeado	Número de errores introducidos en los patrones							
	3 píxeles		4 píxeles		5 píxeles		6 píxeles	
	Recuperados	Porcentaje	Recuperados	Porcentaje	Recuperados	Porcentaje	Recuperados	Porcentaje
Patrón "A"	10 de 10	100%	10 de 10	100%	10 de 10	100%	10 de 10	100%
Patrón "B"	1 de 10	10%	5 de 10	50%	3 de 10	30%	1 de 10	10%
Patrón "C"	10 de 10	100%	10 de 10	100%	10 de 10	100%	9 de 10	90%
Patrón "D"	10 de 10	100%	3 de 10	30%	3 de 10	30%	4 de 10	40%
Todos los test se han realizado con 10 patrones diferentes generados de manera aleatoria								

De los resultados, se puede ver que la ONN tiene una buena capacidad de recuperación para los patrones “A” y “C”, con resultados de 100% para ambos casos, excepto en el caso de 6 píxeles de errores en el caso del patrón “C”. Por otro lado, para los casos del patrón “B” y “D” la capacidad de recuperación es bastante baja, pues se obtiene unos resultados de éxito de aproximadamente un 30% de los casos.

5.4.2. Test 7x5 con 5 patrones

Este test es igual al anterior, pero en este caso se aprenden 5 patrones, los correspondientes a las letras “A”, “B”, “C”, “D” y “E”. Después de entrenar la ONN, los resultados obtenidos se muestran en la Tabla 7.

Tabla 7: Resultados Test 7x5 con 5 patrones.

Test 7x5 con 5 patrones	
Patrones de test de entrada	**Resultado del test**
	

5.4.2.1. *Test 7x5 con 5 patrones: Test ruido*

En este caso, se suprime el conjunto de imágenes obtenidas durante el test y se va a presentar directamente la tabla de resultados. Estos se pueden observar en la Tabla 8.

Tabla 8: ONN 7x5 con 5 patrones. Test ruido.

Patrón testeado	**Número de errores introducidos en los patrones**							
	3 píxeles		**4 píxeles**		**5 píxeles**		**6 píxeles**	
	Recuperados	**Porcentaje**	**Recuperados**	**Porcentaje**	**Recuperados**	**Porcentaje**	**Recuperados**	**Porcentaje**
Patrón "A"	10 de 10	100%	10 de 10	100%	10 de 10	100%	10 de 10	100%
Patrón "B"	9 de 10	90%	10 de 10	100%	6 de 10	60%	5 de 10	50%
Patrón "C"	10 de 10	100%	9 de 10	90%	7 de 10	70%	8 de 10	80%
Patrón "D"	6 de 10	60%	3 de 10	30%	5 de 10	50%	3 de 10	30%
Patrón "E"	8 de 10	80%	6 de 10	60%	7 de 10	70%	3 de 10	30%
Todos los test se han realizado con 10 patrones diferentes generados de manera aleatoria								

5.4.3. Test 7x5 capacidad

En este test se evalúa la capacidad que tiene la ONN de aprender diferentes patrones. En este caso se han enviado los 20 patrones diferentes a la ONN para aprender y el resultado se puede ver en la Figura 53. En esta se observa que la ONN ha sido capaz de aprender hasta 14 patrones diferentes.

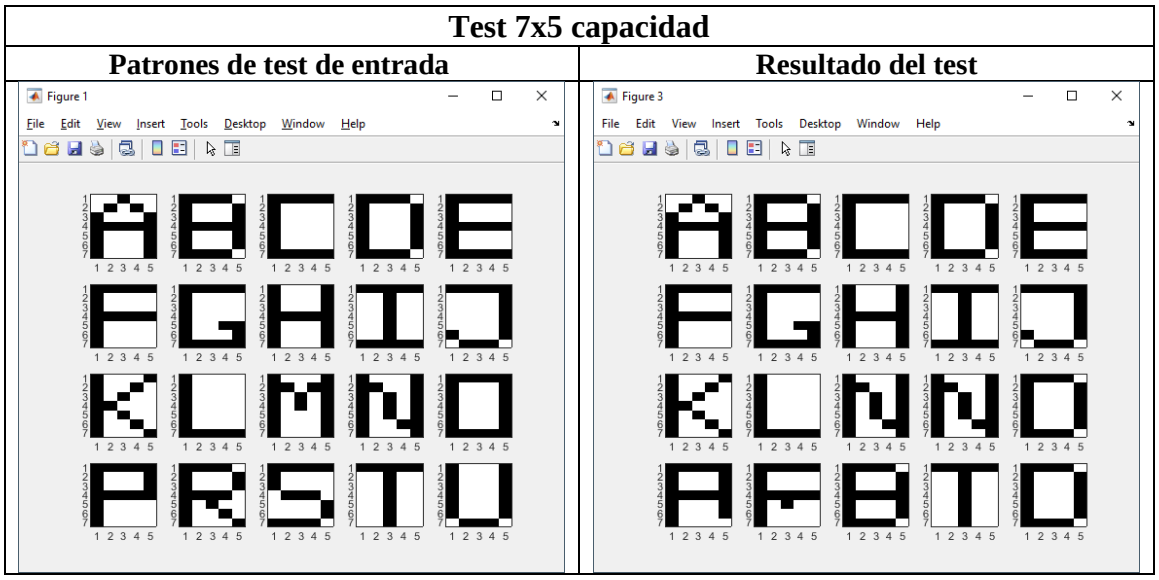


Figura 53: Test 7x5 capacidad

5.5. Recursos

En este punto se analizan los recursos utilizados para la implementación de los dos sistemas cuyos resultados se han reportado, es decir, las versiones de 5x3 y 7x5 neuronas.

Recursos sistema experimental completo para ONN 5x3

En la Figura 54 se muestran los recursos utilizados y en la Figura 55 se muestra el *layout* de la FPGA.

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	7443	53200	13.99
LUTRAM	60	17400	0.34
FF	5282	106400	4.96
IO	4	125	3.20
BUFG	2	32	6.25

Figura 54: Recursos lógica para sistema 5x3

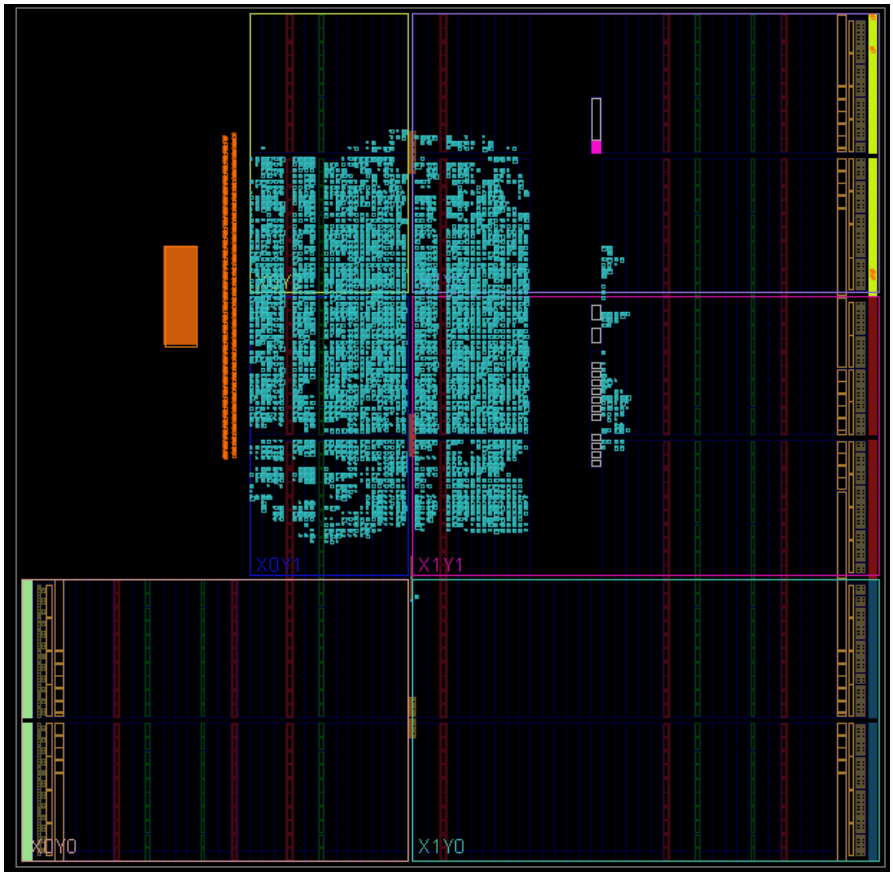


Figura 55: Ocupación FPGA para sistema experimental 5x3

La herramienta de Vivado también tiene la capacidad de indicar como se reparten los recursos en función de cada módulo. Esto permite saber cuál de estos es el que más recursos consume. En la Figura 56 se ve que el módulo "mod_updatematrix" es el que más recursos en LUT se lleva, seguido del módulo "mod_lear_hebb_iterative".

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Bonded IOB (125)	Bonded IOPADs (130)	BUFGCTRL (32)
design_1_wrapper	7443	5298	167	2605	7383	60	4	130	2
design_1_i (design_1)	7443	5298	167	2605	7383	60	0	0	2
Interface_ONN_0 (design_1_Interface_ONN_0_0)	412	974	128	302	412	0	0	0	0
processing_system7_0 (design_1_processing_system7_0_0)	0	0	0	0	0	0	0	0	1
ps7_0_axi_periph (design_1_ps7_0_axi_periph_0)	358	426	0	137	299	59	0	0	0
rst_ps7_0_50M (design_1_rst_ps7_0_50M_0)	17	33	0	8	16	1	0	0	0
top_ONN_0 (design_1_top_ONN_0_0)	6656	3865	39	2185	6656	0	0	0	1
inst (design_1_top_ONN_0_0_top_ONN)	6580	3865	39	2185	6580	0	0	0	1
dut (design_1_top_ONN_0_0_neuronn)	5016	3242	30	1864	5016	0	0	0	1
dut_mod_updatematrix (design_1_top_ONN_0_0_mod_	3833	2523	0	1534	3833	0	0	0	0
NeuronNetwork[0].NCell (design_1_top_ONN_0_0_neu	143	45	2	68	143	0	0	0	0
NeuronNetwork[1].NCell (design_1_top_ONN_0_0_neu	147	45	2	66	147	0	0	0	0
NeuronNetwork[2].NCell (design_1_top_ONN_0_0_neu	124	45	2	90	124	0	0	0	0
NeuronNetwork[3].NCell (design_1_top_ONN_0_0_neu	156	45	2	63	156	0	0	0	0
NeuronNetwork[4].NCell (design_1_top_ONN_0_0_neu	67	45	2	42	67	0	0	0	0
NeuronNetwork[5].NCell (design_1_top_ONN_0_0_neu	54	45	2	37	54	0	0	0	0
NeuronNetwork[6].NCell (design_1_top_ONN_0_0_neu	153	45	2	77	153	0	0	0	0
NeuronNetwork[7].NCell (design_1_top_ONN_0_0_neu	120	45	2	75	120	0	0	0	0
NeuronNetwork[8].NCell (design_1_top_ONN_0_0_neu	63	45	2	44	63	0	0	0	0
NeuronNetwork[9].NCell (design_1_top_ONN_0_0_neu	106	45	2	71	106	0	0	0	0
NeuronNetwork[10].NCell (design_1_top_ONN_0_0_ne	46	45	2	28	46	0	0	0	0
NeuronNetwork[11].NCell (design_1_top_ONN_0_0_ne	48	45	2	27	48	0	0	0	0
NeuronNetwork[12].NCell (design_1_top_ONN_0_0_ne	150	45	2	58	150	0	0	0	0
NeuronNetwork[13].NCell (design_1_top_ONN_0_0_ne	63	45	2	45	63	0	0	0	0
NeuronNetwork[14].NCell (design_1_top_ONN_0_0_ne	67	45	2	51	67	0	0	0	0
state_dut (design_1_top_ONN_0_0_serial2states)	103	18	0	54	103	0	0	0	0
dut_iterative_module (design_1_top_ONN_0_0_mod_learr	1515	505	1	810	1515	0	0	0	0
dut_mod_data_capture (design_1_top_ONN_0_0_mod_da	50	118	8	47	50	0	0	0	0
xlconcat_0 (design_1_xlconcat_0_0)	0	0	0	0	0	0	0	0	0
xlconcat_1 (design_1_xlconcat_1_0)	0	0	0	0	0	0	0	0	0
xlslice_0 (design_1_xlslice_0_0)	0	0	0	0	0	0	0	0	0
xlslice_2 (design_1_xlslice_2_0)	0	0	0	0	0	0	0	0	0

Figura 56: Reparto recursos sistema 5x3

Recursos sistema experimental completo para ONN 7x5

La Figura 57 muestra los recursos utilizados para el caso de la red neuronal de 7x5.

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	38147	53200	71.70
LUTRAM	60	17400	0.34
FF	19626	106400	18.45
IO	4	125	3.20
BUFG	3	32	9.38

Figura 57: Recursos lógica para sistema 7x5

En la Tabla 9 se muestra una comparativa de los recursos utilizado en los diferentes modelos de redes neuronales.

Tabla 9: Comparativa recursos

Modelo	**Incrementos en recursos**				
	Neuronas	**LUT**	**LUTRAM**	**FF**	**BUFG**
5x3	15	7443	60	5282	2
7x5	35 **Δ 130 %**	38147 **Δ 410 %**	60 **Δ 0 %**	19626 **Δ 270 %**	3 **Δ 50 %**

Se observa que en cuanto a LUT (Look Up Table) se produce un incremento de más del 410 % al pasar de 15 neuronas a 35 neuronas, que supone un incremento de 130% en cuanto al número de neuronas. Se ve que el consumo de LUT se dispara al incrementar el número de neuronas, esto hace pensar que este recurso es el que más va a limitar a la hora de escalar el número de neuronas. Además, como se puede ver en el caso de la red neuronal de 7x5 los recursos en cuanto a LUT de la Zynq se van a 71% del total.

6. Conclusiones

Se ha diseñado, implementado y verificado experimentalmente una ONN digital con capacidad de aprendizaje supervisado con mejores prestaciones que la regla de Hebb. Durante el desarrollo de este trabajo se han abordado las siguientes tareas:

- Familiarización con determinados conceptos relativos a las redes neuronales y su aprendizaje. En particular con el modelo de red de Hopfield y reglas de aprendizaje Hebbian.
- Familiarización con el funcionamiento de los elementos básicos que forman las redes neuronales oscilatorias.
- Comprensión del funcionamiento de la ONN digital desarrollada en el proyecto europeo NeurONN.
- Se ha desarrollado un algoritmo de aprendizaje supervisado y se ha comparado su comportamiento con la regla de aprendizaje simple de Hebb. Los resultados que se han obtenido han sido positivos, ya que, se ha observado una mejora de entorno a tres veces más capacidad con respecto al algoritmo simple.
- Se ha diseñado un bloque hardware que implementa dicho algoritmo de aprendizaje. Se ha validado su comportamiento comparando los resultados de simulación con los obtenidos en MATLAB.
- Se ha desarrollado un sistema hardware-software en un dispositivo Zynq para verificar experimentalmente la ONN digital con capacidad de aprendizaje on-line. Se ha hecho uso del procesador embebido del que dispone el dispositivo ZYNQ, como pasarela de comunicación entre el diseño del algoritmo de aprendizaje iterativo con la ONN implementado en la lógica y el exterior.
- Se han desarrollado una serie de funciones en MATLAB para controlar la ONN y el bloque de aprendizaje implementado en el Zynq.
- Se han realizado pruebas con ONNs de 5x3 y 7x5 para una aplicación de reconocimiento de dígitos. El entorno proporciona los medios para poder automatizar estos test y poder comprobar de forma visual los resultados.
- Durante el desarrollo de este TFM se ha detectado algunas posibles líneas de estudio y/o mejoras que se podrían plantear en futuras investigaciones:
    - Adaptación del sistema y setup de test desarrollados en este TFM para reemplazar la versión digital de la ONN por la ONN analógica desarrollada en NeurONN. Esta línea permitiría dotar a la ONN analógica de la capacidad de aprendizaje on-line, además de proporcionar un entorno para la implementación de test y la verificación de su funcionamiento.
    - Aunque el principal objetivo de este TFM no era el estudio en profundidad del algoritmo de aprendizaje iterativo implementado, sería interesante estudiarlo con más detalle y comparar su comportamiento con respecto a otro tipo de algoritmos de aprendizaje con supervisión reportados en la literatura.
    - Implementación de una aplicación real mediante el uso del sistema desarrollado en este TFM. Una posible aplicación sería utilizar el entono en el que la alimentación de datos de entrada al algoritmo sea proporcionada desde un sensor real, como podría ser imágenes provenientes de una cámara.

7. Referencias

[1] MathWorks, «https://es.mathworks.com/products/matlab.html,» [En línea].

[2] Vivado. [En línea]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[3] XILINX, «Vitis,» [En línea]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[4] DIGILENT. [En línea]. Available: https://digilent.com/reference/programmable-logic/zybo-z7/start.

[5] Digilent. [En línea]. Available: https://digilent.com/reference/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf.

[6] Hoppensteadt, F.C. y E. M. Izhikevich, «Oscillatory neurocomputers with dynamic connectivity,» *Phys. Rev. Lett.,* vol. 82, nº (14), 2983 doi:doi.org/10.1103/PhysRevLett.82.2983, 1999.

[7] A. e. a. Raychowdhury, «Computing with Networks of Oscillatory Dynamical Systems,» *Proc. of the IEEE,* vol. 107, nº 1, pp. 73-89, 2019.

[8] Hoppensteadt, F.C. y E. Izhikevich, «Pattern recognition via synchronization in phase-locked loop neural networks,» *IEEE Transactions on Neural Networks,* vol. 11, nº 3, pp. 734-738. doi: 10.1109/72.846744, 2000.

[9] G. Csaba, «Computing with Coupled Oscillators: Theory, Devices, and Applications,» *IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy,* pp. 1-5, 2018.

[10] Nikonov, D.E. y e. al., «Coupled-Oscillator Associative Memory Array Operation for Pattern Recognition,» *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits,* vol. 1, pp. 85-93. doi: 10.1109/JXCDC.2015.2504049, 2015.

[11] P. Maffezzoni, L. Daniel, N. Shukla, S. Datta y A. Raychowdhury, «Modeling and Simulation of Vanadium Dioxide Relaxation Oscillators,» *IEEE Transactions On Circuits-I,* vol. 62, pp. 2207-2215. doi: 10.1109/TCSI.2015.2452332, 2015.

[12] A. Parihar, N. Shukla, S. Datta y A. Raychowdhury, «Synchronization of pairwise-coupled, identical, relaxation oscillators based on metal-insulator phase transition devices: a model study,» *J. Appl. Phys.,* vol. 117, nº 5, 2015.

[13] H.-T. Kim, «Electrical oscillations induced by the metal-insulator transition in VO2,» 2010.

[14] J. Núñez, J. M. Quintana, M. J. Avedillo, M. Jiménez, A. Todri-Sanial, E. Corti, S. Karg y B. Linares-Barranco, «Insights Into the Dynamics of Coupled VO2 Oscillators for ONNs,» *IEEE Trans. Circuits Syst. II: Express Briefs,* vol. 68, nº 10, pp. 3356-3360, 2021.

[15] J. Hopfield, «Neural networks and physical systems with emergent collective computational capabilities,» *Proceedings of the national academy of sciences,* vol. 79, nº 8, pp. 2554-2558, 1982.

[16] D. Hebb, The Organization of Behavior, DOI: 10.1016/S0361-9230(99)00182-3, 1949.

[17] R. McEliece, E. Posner, E. Rodemich y S. Venkatesh, «The Capacity of the Hopfield Associative Memory,» *IEEE Transactions on Information Theory,* vol. 33, nº 4, pp. 461-482, July 1987.

[18] S. Diederich y M. Opper, de *“Learning of correlated patterns in spin-glass networks by local learning rules,” Phys. Rev. Lett., vol. 58, no. 9*, Mar. 1987, p. pp. 949–952.

[19] G. Tanaka, «Spatially Arranged Sparse Recurrent Neural Networks for Energy Efficient Associative Memory,» *IEEE Trans. Neural Netw. Learning Syst.,* vol. 31, nº 1, pp. 24-38, Jan. 2020.

[20] M. Abernot, T. Gil, M. Jiménez, J. Núñez, M. J. Avellido, B. Linares-Barranco, T. Gonos, T. Hardelin y A. Todri-Sanial, «Digital Implementation of Oscillatory Neural Network for Image Recognition Applications,» *Frontiers in Neuroscience,* vol. 15, nº 713054, 2021.

8. Anexos

8.1. Anexo I. Código MATLAB “hebb_unsupervised”

```
clear all;

%import patterns from external file "training_data.txt"
pattern = importdata("training_data.txt");
%extract pattern's lenght and the number of patterns
pattern_length= length(pattern(1,:));
n_pattern = length(pattern(:,1));
n_pattern = 2
%initialize the Weigth matrix to zero
W = zeros(pattern_length,pattern_length);
%process of update the Weigth matrix with the patterns
for i=1:n_pattern
   W=update(W,pattern(i,:));
end

%test the weitgh matrix with pattern
for i=1:n_pattern
   test(i,:) = pattern(i,:)*W;
   for j=1:pattern_length
      if test(i,j) > 0
         result(i,j) = 1;
      else
         result(i,j) = 0;
      end
   end
end

%compare result with origianl pattern
for i=1:n_pattern
   if pattern(i,:) == result(i,:)
      compare(i) = 1;
   else
      compare(i) = 0;
   end
end


%function in charge of updating the weight
function W = update(W,pattern)
   for j = 1: length(pattern)      % j used to sweep the rows
      for i = 1: length(pattern)  % i used to sweep the colums
         if i == j
            W(j,i) = 0;
         else
            if pattern(j) == pattern(i)  % if input equals target, increase the weight
               W(j,i) = W(j,i) + 1;
            else              % if input differs from target, decrease the weight
               W(j,i) = W(j,i) - 1;
            end
         end
      end
   end
end
```

8.2. Anexo II. Código MATLAB “hebb_unsupervised_test1”

```
clear all;

%import pattern from external file
pattern = importdata("training_data.txt");

%extract pattern lenght and number of pattern
pattern_length= length(pattern(1,:));
n_pattern = length(pattern(:,1));

%initialize the Weigth matrix to zero
W = zeros(pattern_length,pattern_length);

%Learn the pattern 0
W=update(W,pattern(1,:));

%Test the pattern 0
test0(1,:) = pattern(1,:);
test0(2,:) = pattern(1,:)*W;
for i = 1:pattern_length
   if test0(2,i) > 0
      test0(2,i) = 1;
   else
      test0(2,i) = 0;
   end
end
%Test the pattern 1
test1(1,:) = pattern(2,:);
test1(2,:) = pattern(2,:)*W;
for i = 1:pattern_length
   if test1(2,i) > 0
      test1(2,i) = 1;
   else
      test1(2,i) = 0;
   end
end
%Test the pattern 2
test2(1,:) = pattern(3,:);
test2(2,:) = pattern(3,:)*W;
for i = 1:pattern_length
   if test2(2,i) > 0
      test2(2,i) = 1;
   else
      test2(2,i) = 0;
   end
end

%Learn the pattern 1
W=update(W,pattern(2,:));

%Test the pattern 0
test0(3,:) = pattern(1,:)*W;
for i = 1:pattern_length
   if test0(3,i) > 0
```

```
        test0(3,i) = 1;
    else
        test0(3,i) = 0;
    end
end
%Test the pattern 1
test1(3,:) = pattern(2,:)*W;
for i = 1:pattern_length
    if test1(3,i) > 0
        test1(3,i) = 1;
    else
        test1(3,i) = 0;
    end
end
%Test the pattern 2
test2(3,:) = pattern(3,:)*W;
for i = 1:pattern_length
    if test2(3,i) > 0
        test2(3,i) = 1;
    else
        test2(3,i) = 0;
    end
end

%Learn the pattern 2
W=update(W,pattern(3,:));

%Test the pattern 0
test0(4,:) = pattern(1,:)*W;
for i = 1:pattern_length
    if test0(4,i) > 0
        test0(4,i) = 1;
    else
        test0(4,i) = 0;
    end
end
%Test the pattern 1
test1(4,:) = pattern(2,:)*W;
for i = 1:pattern_length
    if test1(4,i) > 0
        test1(4,i) = 1;
    else
        test1(4,i) = 0;
    end
end
%Test the pattern 2
test2(4,:) = pattern(3,:)*W;
for i = 1:pattern_length
    if test2(4,i) > 0
        test2(4,i) = 1;
    else
        test2(4,i) = 0;
    end
end
```

```
%function in charge of updating the weight
function W = update(W,pattern)
   for j = 1: length(pattern)      % j used to sweep the rows
      for i = 1: length(pattern)  % i used to sweep the colums
         if i == j
            W(j,i) = 0;
         else
            if pattern(j) == pattern(i)  % if input = target, increase the weight
               W(j,i) = W(j,i) + 1;
            else              % if input != target, decrease the weight
               W(j,i) = W(j,i) - 1;
            end
         end
      end
   end
end
```

8.3. Anexo III. Código MATLAB "LearnHebb"

```
% InputTrainingMatrix: Input matrix with the pattern to be learnt
% WeightMatrix: Weight matrix
% UpdateMatrix: Matrix with elements that indicates if updated or not the
% weight
% NumError: Indicate the Number of possitions that could be correcte for iteration
function [WeightMatrixOut] = LearnHebb(InputTrainingMatrix,WeightMatrix,UpdateMatrix,
NumError)

%InputMatrix = zeros(size(InputTrainingMatrix));
InputMatrix=reshape(InputTrainingMatrix(:,:,1)',[],1);

InputUpdateMatrix=reshape(UpdateMatrix(:,:,1)',[],1);

InputMatrixLength=length(InputMatrix);
MatrixNew = zeros(InputMatrixLength,InputMatrixLength);
MatrixNew=WeightMatrix;
Counter = 0;
   for j = 1: InputMatrixLength       % j used to sweep the rows
      for i = 1: InputMatrixLength    % i used to sweep the colums
         if i == j
            MatrixNew(j,i) = 0;
         else
            if InputUpdateMatrix(j) == 1
               if InputMatrix(j) == InputMatrix(i)  % if input = target, increase the weight
                  MatrixNew(j,i) = MatrixNew(j,i) + 1;
                  MatrixNew(i,j) = MatrixNew(i,j) + 1;
               else              % if input != target, decrease the weight
                  MatrixNew(j,i) = MatrixNew(j,i) - 1;
                  MatrixNew(i,j) = MatrixNew(i,j) - 1;
               end
            end
         end
      end
         if InputUpdateMatrix(j) == 1
            Counter = Counter + 1;
            if Counter == NumError
               break
            end
         end
   end
WeightMatrixOut=MatrixNew;
end
```

8.4. Anexo IV. Código MATLAB “matrix_fig”

```
% Modification Date: 11/05/2021
% Changes:  Ticks 2-steps for image size > 20
%          Custom Title as cell with position

function fig_h = matrix_fig(M,ntitle,ptitle)
if nargin < 2, ntitle = ''; end
if nargin < 3, ptitle = []; end

p_im = -M; % To print -1 as white and 1 as black
nb_pat = size(p_im,3);
n_row = 1;

if nb_pat > 20
   warndlg('Too many images introduced')
   nb_pat = 20;
   n_row = 4;
elseif nb_pat > 15
   n_row = 4;
elseif nb_pat > 10
   n_row = 3;
elseif nb_pat > 5
   n_row = 2;
end

n_col = ceil(nb_pat/n_row);
fig_h = figure; colormap('gray')
for i=1:nb_pat
%    subplot(1+floor(nb_pat/5),nb_pat,i)
   subplot(n_row,n_col,i)
   imagesc(p_im(:,:,i))
   if length(unique(p_im(:,:,i)))==1
      colormap(gca,gray(-unique(p_im(:,:,i))))
   end
   if nb_pat > 2 && (size(M,1)>20 || size(M,2)>20),
     if size(M,1) > 20
        set(gca,'YTick',2:2:size(M,1))
     end
     if size(M,2) > 20
        set(gca,'XTick',2:2:size(M,2))
     end
   else
      set(gca,'YTick',1:size(M,1))
      set(gca,'XTick',1:size(M,2))
   end

   %grid
end

if isstr(ntitle)
   subplot(n_row,n_col,ceil(n_col/2))
   title(ntitle)
%    set(gca,'fontsize',14)
elseif iscell(ntitle)
```

```
%     if length(ntitle) == length(ptitle),
   if length(ntitle) ~= length(ptitle), ptitle = 1:length(ntitle); end
   for k=1:length(ntitle)
      subplot(n_row,n_col,ptitle(k))
      title(ntitle{k})
   end

end
```

8.5. Anexo V. Código MATLAB "TestMatrix"

```
function [Mout_HL, steadyOut, UpdateMatrix] = TestMatrix(M,W,n_iter)
Mout_HL = zeros(size(M));
steadyOut = zeros(3,size(M,3));
tic
aux2=0; % Auxiliar variable to count the number of zero-eval cases.


   for i=1:size(M,3)
      aux_en=1;
      Mti=reshape(M(:,:,i)',[],1);
      Mout = Mti; %Mout store the input Matrix
%        for k=1:n_it
      for k=1:n_iter
         Mout_next = W*Mout;
         aux=find(Mout_next==0);
         if aux, Mout_next(aux) = Mout(aux);
            if aux_en
               aux2=aux2+1;
               aux_en=0;
            end
         end
         Mout_next = sign(Mout_next);
%           Mout_next = hardlims(Mout_next);
         M_compare = zeros(size(Mout_next));
         %Compare the value of Mout_next with the value of input Matrix
         for j=1:size(Mout_next)
            if Mout_next(j) == Mout(j)
               M_compare(j) = 0;   %If equal 0
            else
               M_compare(j) = 1;   %If different 1
            end
         end

         if isequal(Mout,Mout_next)
            if ~steadyOut(1,i)
               steadyOut(1,i) = k
            else
               break
            end
         else
            if steadyOut(1,i)
               steadyOut(2,i) = -k;
               steadyOut(3,i) = steadyOut(1,i);
               steadyOut(1,i) = 0;
            end
         end
         Mout = Mout_next; % Input: +1/-1
         Mout_HL2=reshape(Mout',[size(M,2),size(M,1)])';
         UpdateMatrix=reshape(M_compare',[size(M,2),size(M,1)])';
      end
      Mout_HL(:,:,i) = Mout_HL2;
%       isequal(Mout,Mti), aux2
   end
```

```
%     matrix_fig(Mout_HL,sprintf('Iteration %s',num2str(k)));
aux2; % Uncomment to display value while running

toc
display('TestMatrix Evaluation Finished')

end
```

8.6. Anexo VI. Código MATLAB "LearnIterative"

```
clear all;
load('Train_5x3.mat');
% MTrain = Mtrain;
PatternFirst = 1;   %indica el primer patron que se va a aprender
PatternEnd = 10;     %indica el ultimo patron que se va a aprender
IterationByStep = 1;    %indica si el proceso iterativo es por pasos o  no
%Se obtiene los parametros de tamaño y numero de matrices de entrada
MatrixSize=size(MTrain(:,:,:));
MatrixNumber = MatrixSize(3);
MatrixSize = MatrixSize(1)*MatrixSize(2);
%se genera la matriz de pesos nueva y se inicia a 0
NewWeightMatrix = zeros(MatrixSize);
%Se crea la matriz de unos para el primer proceso de aprendizaje
UpdateMatrix = ones(size(MTrain(:,:,1)));
%Se crean matrices para la salida y los errores
MOut(:,:,MatrixNumber) = zeros(size(MTrain(:,:,1)));
MatrixError(:,:,MatrixNumber) = zeros(size(MTrain(:,:,1)));
%se inicializa la primera matriz todo a 1, para el primer aprendizaje
MatrixError(:,:,1) = ones(size(MTrain(:,:,1)));
NRow = length(MTrain(:,1,1));
NColumn = length(MTrain(1,:,1));
NMatrix = length(MTrain(1,1,:));
Pattern_List = zeros(NRow,NColumn,NMatrix);
pointer = 1;
i = 1;
state = 1;
FirstPattern = 1;

while (pointer < NMatrix)

switch state

   case 1  %Idle
      %Nuevo patron
      Pattern_Input = MTrain(:,:,pointer);
      Pattern_List(:,:,i) = Pattern_Input;
      pointer = pointer + 1;
      New_Pattern = i;
      if FirstPattern == 1
         %se inicializa la primera matriz todo a 1, para el primer aprendizaje
```

```
        MatrixError(:,:,1) = ones(size(MTrain(:,:,1)));
        FirstPattern = 0;
        state = 2;
    else
        j = New_Pattern;
        Iteration = 1;
        state = 3;
    end

  case 2 %Learn
    %se aprende el patron i

NewWeightMatrix=LearnHebb(Pattern_List(:,:,i),NewWeightMatrix,MatrixError(:,:,i),MatrixSi
ze);
    j = 1;
    state = 3;

  case 3 %Test 1
    %se testea el patron j
    [MOut(:,:,j), steadyOut, MatrixError(:,:,j)]=TestMatrix(Pattern_List(:,:,j),
NewWeightMatrix, 1);
    a = MatrixError(:,:,j);
    Error  = numel(a(a==1));
      if Error == 0 %no errores
        if j >= New_Pattern
          i = New_Pattern + 1;
          state = 1;
        else
          j = j +1;
          state = 3;
        end
      else  %hay errores
        Iteration = Iteration + 1;
        if Iteration >= 50
          i = New_Pattern;
          state = 1;
        else
          %j = j + 1;
          i = j;
          state = 2;
        end
      end
```

```
    otherwise
        state = 1;
end
end
%Se pinta el resultado
matrix_fig(MTrain(:,:,PatternFirst:PatternEnd),"entrada", "entrada")
matrix_fig(MOut(:,:,PatternFirst:PatternEnd))
```