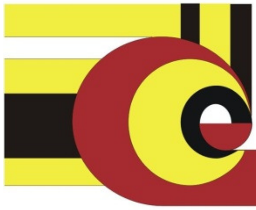


Instituto Politécnico Nacional

Centro de Investigación en Computación

Sensor virtual con entrenamiento en línea implementado sobre FPGA

Tesis que presenta el

ISC. Sergio Flores Velázquez

Para obtener el grado de

Maestro en Ciencias en Ingeniería de Cómputo

con opción en Sistemas Digitales.

Directores de Tesis:

Dr. Marco Antonio Moreno Armendáriz

Dr. Carlos Alberto Cruz Villar

México D.F., 18 de diciembre de 2012

SIP-14 bis

INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 11:00 horas del día 15 del mes de noviembre de 2012 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Sensor virtual con entrenamiento en línea implementado sobre FPGA"

Presentada por el alumno:

FLORES	**VELÁZQUEZ**	**SERGIO**
Apellido paterno	Apellido materno	Nombre(s)

Con registro: | B | 1 | 0 | 1 | 8 | 7 | 1 |

aspirante de: **MAESTRÍA EN CIENCIAS EN INGENIERÍA DE CÓMPUTO CON OPCIÓN EN SISTEMAS DIGITALES**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Directores de Tesis

Dr. Marco Antonio Moreno Armendáriz

Dr. Carlos Alberto Cruz Villar

Dr. Sergio Suárez Guerra

Dr. Luis Pastor Sánchez Fernández

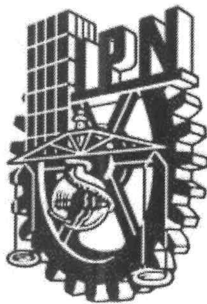
M. en C. Osvaldo Espinosa Sosa

Dr. Víctor Hugo Ponce Ponce

PRESIDENTE DEL COLEGIO DE PROFESORES

Dr. Luis Alfonso Villa Vargas

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION EN COMPUTACION
DIRECCION



INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA DE CESIÓN DE DERECHOS

En la ciudad de México, el día 12 del mes de noviembre del año 2012, el que suscribe **Sergio Flores Velázquez** alumno del Programa de Maestría en Ciencias en Ingeniería de Cómputo, opción en Sistemas Digitales, con número de registro B101871, adscrito al Centro de Investigación en Computación, manifiesta que es autor intelectual del presente trabajo de Tesis bajo la dirección del Dr. Marco Antonio Moreno Armendáriz y el Dr. Carlos Alberto Cruz Villar cede los derechos del trabajo titulado "Sensor Virtual con entrenamiento en línea implementado sobre FPGA", al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a las siguientes direcciones ing.tuzo.biker@gmail.com, mam_armendariz@cic.ipn.mx y cacruz@cinvestav.mx. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Sergio Flores Velázquez

Resumen

La presente tesis detalla el diseño e implementación un sensor virtual sobre un FPGA. El sensor virtual es del tipo *caja negra* y utiliza como metodología de diseño el aprendizaje de la red neuronal para adaptarse a variaciones que pudiesen presentarse a lo largo de la operación del sistema.

La implementación de la red neuronal comprende dos etapas: la propagación hacia adelante de las entradas y el algoritmo de aprendizaje. Las neuronas de las capas ocultas tienen función de activación sigmoide logarítmica, mientras que las de la capa de salida, la función lineal. La función sigmoide logarítmica se implementó mediante la linealización por partes. La propagación hacia adelante de la red neuronal se puede ejecutar de forma serial, por bloques o bien totalmente paralela, con lo que se consigue un menor tiempo de ejecución a costa de un mayor uso de recursos del FPGA.

El algoritmo de aprendizaje de la red neuronal tiene como finalidad ajustar los valores de los parámetros de la red neuronal, de esta manera se consigue un regresor capaz de ajustarse a una señal no lineal.

Esta implementación resultó tener un alto grado de precisión con un moderado uso de recursos, lo que da pie a su integración en proyectos que puedan ser desarrollados a futuro.

Abstract

This thesis describes the virtual sensor design and implementation on FPGA. The *black box* virtual sensor was designed using the neural network learning to adapt it to variations that may occur throughout the system operation.

The neural network implementation comprises two stages: the feedforward inputs and the learning algorithm. The hidden layers neurons have the logarithmic sigmoid function as activation function, while the output layer has the linear function. The logarithmic sigmoid function was implemented by using the piecewise linearization. The feedforward neural network can be run serially, blocks or fully parallel, thus result in lower execution time at the cost of increased use of FPGA resources.

The neural network learning algorithm is designed to adjust the neural network parameters to achieving a regressor capable of adjusting itself to a nonlinear signal.

This implementation has a very high degree of accuracy with a moderate use of resources, which leads to their integration in projects that may be developed in the future.

Agradecimientos

La presente tesis representa la culminación de un gran esfuerzo que no hubiese sido posible sin el apoyo, opiniones, ánimos y paciencia de varias personas.

Agradezco al Dr. Marco Antonio Moreno Armendáriz por su paciencia, apoyo y confianza que depositó en mi persona para la elaboración de este trabajo de tesis. Hubo momentos díficiles, de gran premura, los cuales me hicieron salir adelante y culminar con este documento todo el trabajo desarrollado.

Gracias también a mis compañeros por sus consejos, apoyo y asesorías que me fueron de gran ayuda en la realización de este trabajo. Gracias César por tu ejemplo de constancia, gracias Carlos por tus sugerencias. Gracias Pamela, Alejandro, Jesús, José Luis y Rodolfo por haberme acompañado durante estos casi tres años.

Gracias a todos aquellos amigos quienes me apoyaron durante este tiempo: con una palmada, un abrazo o simplemente, con una frase de ánimo. Quiero agradecerle a Mayra por su apoyo y compañía a lo largo de este tiempo.

Gracias al CONACYT y a la SIP-IPN por el apoyo económico brindado, sin el cual la elaboración de la presente tesis no habría sido posible.

Es momento de agradecer a mi familia: a mis padres Patricia y Sergio por sus palabras de aliento en aquellos momentos difíciles por los que pasé, por siempre estar al pendiente de mi trabajo; gracias por su apoyo incondicional. Gracias a mi hermana Betsabé por esos momentos de risa, enojos y tristezas; a mis abuelos por sus infinitas bendiciones.

Tío: donde sea que estés, sigue rodando... espérame en la cima...

A todos. . .
Gracias!!!

Índice general

Resumen I

Abstract II

Agradecimientos III

Glosario XI

1. Introducción 1

1.1. Antecedentes . . . 2
1.2. Planteamiento del problema . . . 4
1.3. Justificación . . . 5
1.4. Hipótesis . . . 5
1.5. Objetivos . . . 5
1.5.1. Objetivo general . . . 5
1.5.2. Objetivos específicos . . . 5
1.6. Alcances y límites . . . 6
1.7. Contribución . . . 6
1.7.1. Producto de la Investigación . . . 6
1.8. Organización de la tesis . . . 6

2. Estado del Arte 8

2.1. Resumen . . . 15

3. Marco Teórico 16
3.1. Neurona Biológica 17
3.2. Neurona Artificial 18
3.3. Tipos de Redes Neuronales Artificiales 21
3.3.1. Capas de neuronas 21
3.3.2. Redes multicapa 22
3.4. Perceptrón multicapa 23
3.4.1. Arquitectura del Perceptrón multicapa 23
3.4.2. Propagación hacia adelante 24
3.5. Aprendizaje de la Red Neuronal Artificial 24
3.5.1. Criterios de Finalización 24
3.5.2. Diseño de las Reglas de Aprendizaje 26
3.5.3. Ecuaciones Generales 31
3.5.4. Proceso de aprendizaje 32
3.6. Resumen 34

4. Desarrollo de la Investigación 35
4.1. Panorama General 36
4.2. Propagación hacia adelante de las señales 38
4.2.1. Módulo *Multadder* 42
4.2.2. Función de activación 43
4.2.3. Modos de ejecución 46
4.2.3.1. Ejecución Serial 46
4.2.3.2. Ejecución por bloques 47
4.2.3.3. Ejecución en Paralelo 49
4.3. Algoritmo de Aprendizaje 50
4.3.1. Cálculo de Sensitividades 52
4.3.2. Actualización de bias 53
4.3.3. Actualización de pesos sinápticos 54
4.3.4. Actualización de entradas 55
4.3.5. Cálculo del Error de Iteración (E_{It}) 56
4.4. Comunicación PC-FPGA 57
4.5. Resumen 58

5. Presentación de Resultados 59
5.1. Experimento 1: Propagación hacia adelante de las señales de entrada 59
5.1.1. Caso de estudio 65

5.2. Experimento 2: Aprendizaje de la red neuronal 70
5.3. Resumen 72

6. Conclusiones **73**
6.1. Conclusiones 73
6.2. Trabajo a Futuro 74

Referencias **75**

Apéndices **81**

A. Función sigmoidal y su derivada **81**

B. Interfaz desarrollada en MatLab **83**

Índice de figuras

1.1. Procedimiento para diseñar un sensor virtual. 3
1.2. Esquema de diseño de un sensor virtual adaptable [1]. 4

2.1. Referencias por año de publicación. 14
2.2. Referencias por tipo de publicación. 15

3.1. Neurona biológica. 17
3.2. Neurona artificial con una entrada. 18
3.3. Funciones de activación lineales. 19
3.4. Funciones de activación no lineales. 20
3.5. Neurona artificial con múltiples entradas. 21
3.6. Capa de S neuronas. 22
3.7. Red neuronal multicapa. 22
3.8. Perceptrón Multicapa en representación matricial. 24
3.9. Red neuronal [2-2-1]. 26

4.1. Tarjeta Altera DE2 70. 35
4.2. Formato de punto flotante de 32 bits. 36
4.3. Diagrama general de la Red Neuronal y su aprendizaje. 37
4.4. Máquina de Estados de la propagación hacia adelante de la Red Neuronal Artificial. 40
4.5. Arquitectura general de la propagación hacia adelante. 41
4.6. Módulo *Multadder*. 42
4.7. Linealización de la función sigmoidal. 44

4.8. Módulo *Sigmoid*. 44
4.9. Modo de ejecución serial. 47
4.10. Modo de ejecución por bloques. 48
4.11. Modo de ejecución paralelo. 50
4.12. Red Neuronal 1-n-1. 50
4.13. Proceso de aprendizaje. 51
4.14. Módulo *S_2*. 52
4.15. Módulo *S_1*. 53
4.16. Módulo *updt_b*. 54
4.17. Módulo *updt_w*. 55
4.18. Convertidor Serial - USB. 57

5.1. Tiempos con diversos modos de ejecución. 62
5.2. Ángulos Pitch y Roll del vehículo Baja SAE. 66
5.3. Sensor de distancia del amortiguador. 66
5.4. Sistema de suspensión activa. 67
5.5. Diagrama del sensor virtual. 68
5.6. Pista de pruebas. 68
5.7. Mediciones del sensor virtual sobre el FPGA. 69
5.8. Mediciones del sensor virtual en MatLab. 69
5.9. Comparativo del entrenamiento de MatLab y del FPGA. 71
5.10. Comparativo del error obtenido por MatLab y por el FPGA. . . 72

Índice de tablas

2.1. Longitudes de datos reportadas 13

5.1. Uso de recursos del FPGA. 60
5.2. Tiempos con diversos modos de ejecución. 61
5.3. Frecuencias máximas de operación. 63
5.4. Tiempos de ejecución. 64
5.5. Resultados obtenidos con el FPGA y con MatLab. 64
5.6. Error promedio de los sensores virtuales. 70

Lista de algoritmos

4.1. Conversión de una neurona simple a multientrada 42
4.2. Linealización de la función sigmoidal 45
4.3. Ejecución Serial 47
4.4. Ejecución a bloques 48
4.5. Ejecución en Paralelo 49
4.6. Cálculo de S^1 53
4.7. Actualización de las entradas 55
4.8. Cálculo de E_{It} 56

Glosario

CMOS

Del inglés *Complementary Metal Oxide Semiconductor*. Tipo de tecnología de semiconductores ampliamente usado. Los semiconductores CMOS utilizan circuitos NMOS (polaridad negativa) y PMOS (polaridad positiva). Dado que sólo un tipo de circuito está activo en un tiempo determinado, los chips CMOS requieren menos energía que los chips que usan sólo un tipo de transistor. 2.0

FPGA

Del inglés *Field Programmable Gate Array*, fue inventadas en 1984 por Ross Freeman y Bernard Vonderschmitt, co-fundadores de Xilinx, es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad se puede programar. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinacional hasta complejos sistemas en un chip. 1.2, 2.0, 4.0, 5.0

LE

Del inglés *Logic Element*. Es la unidad lógica más pequeña en la arquitectura del FPGA(Altera). Como principal característica contiene una tabla de búsqueda (LUT, look-up table) de cuatro entradas, que es un generador de funciones, por lo que puede implementar cualquier función de cuatro variables. 4.0, 5.1

LUT

Del inglés *Look-Up Table*, es una estructura de datos que se usa para substituir la ejecución una rutina mediante el indexado de los registros. Son muy útiles en el ahorro de tiempo y recursos de procesamiento. Un ejemplo práctico de la utilidad de una LUT es su uso para obtener resultados de funciones sin necesidad de hacer el cálculo, utilizando como índice el valor de entrada y como resultado, el valor almacenado en la localidad respectiva. 2.0, 4.2

VHDL

Del inglés *Very High Speed Integrated Circuit Hardware Description Language*, lenguaje creado a partir de un proyecto promovido por el gobierno de Estados Unidos en 1981 y fue homologado por el IEEE en 1987 en el estándar 1076-87. VHDL utiliza una metodología de diseño de arriba abajo (top-bottom) que permite describir el circuito de forma estructural (señales y sus conexiones) y funcional (qué hace). Cada bloque se puede definir empleando subcircuitos definidos en el mismo proyecto o disponibles en librerías propias o de terceros. 1.4

Capítulo 1

Introducción

Los sensores virtuales representan una herramienta que permite la estimación de variables cuyo proceso de medición presenta dificultades técnicas, o inclusive no es posible realizar dado que no se disponen de los sensores físicos necesarios. Su uso se extiende al control y/o seguridad de procesos, o bien, simplemente en tareas de monitoreo. Los sensores virtuales también conocidos como *sensores suaves* ya que son una combinación entre los conceptos software y sensor. El concepto software se refiere a que utilizan modelos computacionales en su operación, mientras que sensor se debe a que entrega información similar a un sensor físico [2].

La industria química ha encontrado en los sensores virtuales una alternativa para la estimación en línea de variables como la concentración de elementos químicos, ya que estas mediciones tardan alrededor de 30 minutos en estar listas. Otros ejemplos de industrias que se ha beneficiado del uso de los sensores virtuales es la industria minera y la automotriz.

1.1. Antecedentes

Los sensores virtuales están basados en un modelo que representa la dinámica del sistema, este modelo relaciona las entradas del sensor (señales medidas) y las señales deseadas. Conforme al trabajo de Jassar [3], existen tres principales metodologías para el diseño de sensores virtuales:

- ▷ Modelado matemático del sistema físico
- ▷ Redes neuronales artificiales
- ▷ Lógica difusa

De acuerdo a su principio de operación, se puede clasificar a los sensores virtuales en dos categorías [4]:

- ▷ Manejados por modelo: esos sensores conocidos como modelos de *caja blanca* se basan en la descripción mediante ecuaciones matemáticas los principios del proceso químico o físico. Aunque los modelos reflejan claramente el mecanismo de operación, su desarrollo requiere un profundo entendimiento y conocimiento del proceso, lo que resulta complicado em un proceso real.
- ▷ Manejados por datos: conocidos como modelos de *caja negra* se construyen a partir de observaciones empíricas del proceso, sin ningún tipo de información del proceso interno.

Algunos ejemplos de sensores virtuales manejados por datos son sensores basados en el filtro de Kalman extendido [5], o en observadores de estado [6], esta familia de sensores virtuales se basa en el modelo de los principios del sistema del cual se desea estimar las variables. Estos modelos fundamentados en leyes físicas que rigen el sistema son desarrollados y utlizados para diseño y control de sistemas, por ello se enfocan en los estados ideales del sistema, lo cual es un gran inconveniente para su operación en ambientes reales. Por ello, los sensores manejados por datos obtenidos de la operación del sistema en entornos no controlados han incrementado su uso.

Entre las técnicas empleadas para el diseño de sensores virtuales manejados por datos, se encuentran el análisis de componentes principales [7], su combinación con modelos de regresión [8], mínimos cuadrados parciales [9],

redes neuronales artificiales [10], sistemas neurodifusos [11] y máquinas de soporte vectorial [12]. Las aplicaciones más comunes de los sensores virtuales se encuentran en monitoreo en línea [13], control en lazo cerrado con bajos periodos de muestreo [14] y detección de fallas [15].

Los sensores virtuales son exitosos en pruebas en periodos cortos, sin embargo, con su uso en periodos prolongados de tiempo, se observó que las estimaciones empiezan a tener diferencias sustanciales con las mediciones reales del proceso obtenidas en laboratorios, esto debido a cambios clim'aticos, cambios en propiedades de fuentes de alimentación, entre otros factores, es decir, las condiciones con las que los sensores fueron entrenados cambiaron con el paso del tiempo con respecto a las condiciones de operación actuales. Esto dió origen al desarrollo de sensores virtuales adaptables, que son capaces de lidiar con las variaciones del entorno en el que fueron entrenados.

Los métodos de aprendizaje más comunes en sensores virtuales adaptables son técnicas de ventanas móviles, técnicas de adaptación recursiva y métodos basados en conjuntos [16].

Fortuna [17] propone la siguiente metodología para el diseño de sensores virtuales (ver figura 1.1).

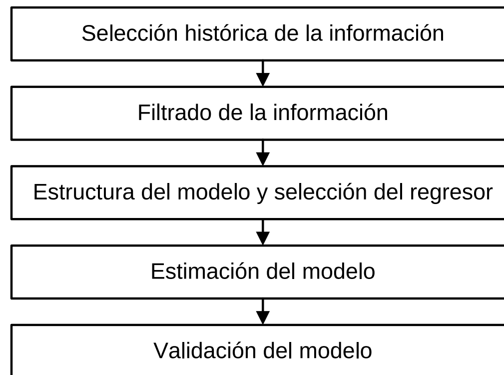


Figura 1.1: Procedimiento para diseñar un sensor virtual.

Como primer paso del diseño se seleccionan los datos medibles del sistema para obtener la salida deseada. Esto se puede realizar de diferentes maneras, pero es vital seleccionar aquellas variables que estén directamente relacionadas con el fenómeno.

Una vez que se han identificado las señales a medir, es necesario determinar qué sensores serán utilizados para dichas mediciones. Algunas veces

es necesario agregar una capa de filtrado para eliminar posibles ruidos del ambiente o del mismo sistema.

Posteriormente, se plantea el modelo que relaciona las señales medidas con las senãles a estimar. La validación del modelo consiste en hacer estimaciones de señales que no se hayan incluído en su diseño, y finalmente, se comparan las señales obtenidas con las esperadas.

A diferencia del trabajo de Fortuna [17], el trabajo realizado por Ma [1] presenta una nueva metodología para la creación de un sensor virtual adaptable, dicha metodología se muestra en la figura 1.2.

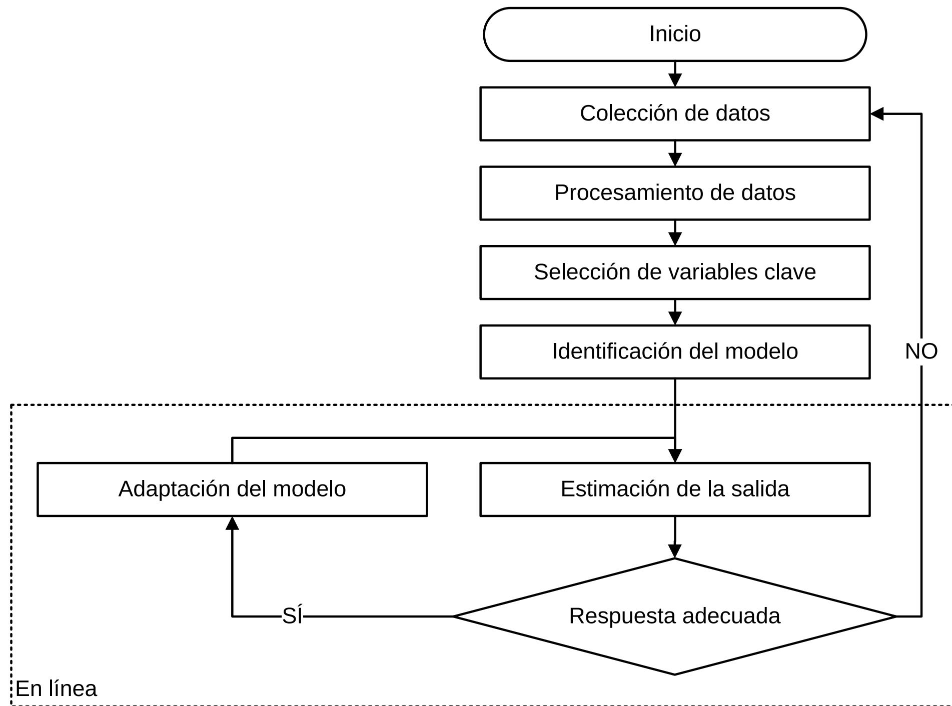


Figura 1.2: Esquema de diseño de un sensor virtual adaptable [1].

1.2. Planteamiento del problema

Actualmente existen algunos sistemas que presentan dificultades técnicas en el proceso de medición de algunas variables de gran importancia. Tales dificultades pueden ser que no se pueden localizar los sensores físicos en el lugar adecuado o bien, que la medición presenta graves retrasos, sin embargo

se dispone de información que se puede correlacionar y con esto determinar el comportamiento estas variables no medibles directamente. Para ello, una de las estrategias más comunes es el uso de sensores virtuales, los cuales permiten reducir costos o realizar un diagnostico rápido del funcionamiento del sistema.

1.3. Justificación

La implementación de una red neuronal con entrenamiento en línea en hardware se puede integrar como un sensor virtual para la estimación de valores cuyas mediciones de sistemas reales no se puedan realizar. La implementación en hardware le da al sensor una mayor rapidez además de posibilitar su integración en proyectos a futuro.

1.4. Hipótesis

El sensor virtual implementado en este trabajo será ser capaz de adaptarse a variaciones en las señales no lineales que recibe como entrada para obtener un resultado a la salida con el que se puedan estimar mediciones de variables en sistemas reales.

1.5. Objetivos

1.5.1. Objetivo general

Este trabajo tiene como objetivo la implementación de un sensor virtual con entrenamiento en línea para estimar mediciones de variables dentro de sistemas reales que no sean posibles de medir directamente. La estimación se realizará mediante información de variables dentro del sistema que sí se pueden medir.

1.5.2. Objetivos específicos

▷ Revisar el estado del arte referente a la implementación de sensores virtuales basados en redes neuronales.

- ▷ Implementar sobre un FPGA un perceptrón multicapa junto con la regla de aprendizaje *backpropagation*.
- ▷ Validar el desempeño del sistema mediante la aproximación de señales no lineales.

1.6. Alcances y límites

En este trabajo se busca implementar un sensor virtual adaptable basado en el aprendizaje de un Perceptrón multicapa. La arquitectura de la red neuronal será de tres capas: capa de entrada, una capa oculta y la capa de salida. En teoría, la capa oculta puede tener cualquier cantidad de neuronas; pero en la práctica no es así, está limitada por el tamaño de la memoria en la que se almacenan los pesos sinápticos, un incremento en el tamaño de la memoria puede dar como resultado un consumo excesivo de recursos.

1.7. Contribución

Mediante el desarrollo de esta tesis se implementará un Perceptrón multicapa junto con sus algoritmo de aprendizaje sobre un FPGA en el cual se actualizarán los pesos sinápticos y umbrales de manera automática. Esta implementación será utilizada como un sensor virtual.

1.7.1. Producto de la Investigación

Se logró la publicación de un artículo en el congreso IASTED International Conference on Signal and Image Processing (SIP 2012) [18]:

Marco A. Moreno-Armendáriz, Carlos A. Duchanoy, Sergio Flores-Velázquez, Carlos A. Cruz-Villar. (2012). *FPGA Implementation of a Pitch and Roll Soft Sensors for Active Suspension System.* Proceedings of the IASTED International Conference Signal and Image Processing (SIP 2012) pp 99-106.

1.8. Organización de la tesis

Capítulo 1 Presenta al lector una introducción a los sensores virtuales. Asimismo establece de manera clara la problemática a resolver, plantea los

objetivos, los alcances y las limitaciones del trabajo por realizar.

Capítulo 2 Ofrece una semblanza del estado del arte referente a la implementación sensores virtuales y redes neuronales artificiales sobre FPGAs. Aquí se describen diversas arquitecturas utilizadas para el diseño de sensores virtuales, así como de redes neuronales artificiales.

Capítulo 3 Expone a profundidad los fundamentos teóricos que tienen como base tanto las redes neuronales artificiales como el algoritmo de aprendizaje basado en la propagación hacia atrás del error y los criterios que marcan el final del entrenamiento de las redes neuronales.

Capítulo 4 Contiene la parte medular de esta tesis, ya que a lo largo de este capítulo se describe el desarrollo de la implementación de la red neuronal artificial y del algoritmo de aprendizaje *Backpropagation* en el FPGA. Este capítulo contiene esquemas y algoritmos que ejemplifican todos aquellos procesos que están inmersos en la ejecución y aprendizaje de redes neuronales.

Capítulo 5 Muestra los resultados obtenidos mediante la ejecución de las etapas de las que se conforma el proyecto, así como se realiza una comparativa entre este trabajo y una aplicación desarrollada en un paquete de cálculo, para ello se incluyen gráficos y tablas que ayudan al lector a comprender dichos resultados.

Capítulo 6 Entrega las conclusiones dados los objetivos alcanzados y posibles opciones de trabajo para nuevos desarrollos no solo en en el campo de las redes neuronales, sino como su integración como un componente en un probable proyecto a futuro.

Apéndices Incluyen la manera de obtener la derivada de la función sigmoide logarítmica. Aquí también se presentan códigos fuente en MatLab y VHDL que fueron utilizados en el desarrollo de esta tesis.

Capítulo 2

Estado del Arte

Los sensores virtuales son ampliamente utilizados en casos donde las tareas de medición son difíciles de llevar a cabo. Existen algunos casos en los que la medición tiene retrasos de tiempo considerables. Los valores de interés son calculados de forma indirecta a partir de datos cuyas mediciones son más fáciles de obtener. En estos sensores, el uso de redes neuronales es un enfoque muy exitoso en el análisis de datos con mucho ruido, o bien en variables no lineales [19].

La industria minera ha utilizado a los sensores virtuales para determinar partículas de minerales a partir de imágenes; esto provee una herramienta muy valiosa para administrar los procesos de la minería de una mejor manera. El modelado mediante redes neuronales brinda resultados precisos en la estimación del tamaño de partículas de minerales [20].

La industria química también se ha beneficiado del uso de redes neuronales dentro de sensores virtuales. Una de sus aplicaciones corresponde a la detección de óxidos de Nitrógeno (NO_x) y Ozono (O_2) en calentadores industriales. En este caso se colocaron sensores físicos en las tuberías para monitorear su sobrecalentamiento, algunos otros sensores se colocaron en las entradas de combustible y de oxígeno a la cámara de combustión para medir la mezcla que se consume. La red neuronal determina la cantidad de contaminantes que son expedidos a la atmósfera [21].

Otra aplicación de los sensores virtuales dentro de la industria química se

utiliza para estimar el punto de congelamiento del queroseno proveniente del proceso de destilación del petróleo crudo. La característica más importante en este trabajo es el reducido conjunto de datos. El uso de máquinas de soporte vectorial (SVM) ayudó a los autores a conseguir una buena precisión en el aprendizaje para este caso, sin embargo, la generalización de la red neuronal mejora con la introducción de ruido a las muestras de entrenamiento [22].

Actualmente las redes neuronales artificiales constituyen una herramienta ampliamente utilizada para resolver una amplia gama de problemas. Debido a esta variedad, también existen distintos enfoques para su integración al problema que se desea resolver. Los avances tecnológicos que existen hoy en día incluyen diversos dispositivos que pueden ser empleados para implementar redes neuronales en hardware, sin nembargo, los FPGAs resultan ser una excelente solución para ello. Una de las mayores ventajas del uso de esta tecnología radica en su capacidad de ejecutar tareas de forma paralela, por otra parte, también destacan su velocidad de operación y la sencillez con la que el diseño se puede ser modificado inclusive en tiempo de ejecución (*reconfiguración*). Sin embargo, pueden ser implementadas también mediante circuitos analógicos (CMOS), estas implementaciones explotan el carácter no lineal de las redes neuronales, además de ser más rápidas y consumir menor cantidad de energía; pero se ven afectados por ruido, calor y su procesamiento no resulta ser muy exacto [23].

Las redes neuronales artificiales que se implementan en FPGA que poseen la capacidad de ser entrenadas, se pueden clasificar en dos tipos:

- ▷ Entrenamiento *On-Line*: el algoritmo de aprendizaje de la red neuronal se implementa en el dispositivo con lo que se logra un entrenamiento continuo a costa de un mayor consumo de recursos de hardware. [23]
- ▷ Entrenamiento *Offline*: consiste en obtener los valores de los pesos sinápticos y umbrales mediante una aplicación desarrollada en software. La red neuronal trabaja con los parámetros fijos a lo largo de su ejecución. [24, 25]

Existen trabajos que se valen de la capacidad de reconfiguración de las redes neuronales para trabajar en paralelo las tareas de aprendizaje de la red neuronal artificial. El proyecto Run-TIME Reconfiguration Artificial Neural

Network (RRANN) justamente aplica ese principio. Esta arquitectura divide la ejecución del algoritmo *backpropagation* en tres etapas que se ejecutan secuencialmente:

- ▷ Propagación hacia adelante: toma las entradas y las propaga a lo largo de la red para obtener una salida de la red.
- ▷ Propagación hacia atrás: toma el valor de salida de la red y obtiene el error de la salida y lo propaga hacia atrás para obtener el error de cada neurona de las capas ocultas.
- ▷ Actualización: se aplican las reglas de aprendizaje para obtener los nuevos valores de los pesos sinápticos.

El fin de la etapa de actualización, marca la terminación del cálculo de patrón de entrenamiento. Este proceso se repite para todos los patrones hasta que la red esté suficientemente entrenada.

RRANN utiliza un FPGA con múltiples procesadores neuronales, cada uno de los cuales contiene 6 neuronas en hardware cuyas subrutinas y memoria RAM son controladas por una máquina de estados. Los procesadores siguen las instrucciones de un controlador global. La RAM de cada procesador almacena los pesos y los resultados deseados, además de actuar como buffer para los valores de error. Para utilizar menor cantidad de recursos, se emplean tablas LUT para implementar las funciones de activación y sus respectivas derivadas.

La reconfiguración en tiempo de ejecución le da al usuario tres maneras de implementar la arquitectura.

1. Configura las tres etapas de operación en el FPGA para la ejecución totalmente en paralelo.
2. Configura la propagación hacia adelante y la actualización en el FPGA, mientras que el FPGA se *reconfigura* cuando se ejecuta la retropropagación.
3. Configura solamente una de las tres etapas en el FPGA, de manera que al finalizar la ejecución de una, se reconfigura la siguiente etapa.

La tercer etapa es la única en ser diseñada, implementada y probada. El proyecto fue implementado en una tarjeta National Technology Inc. que incluye un FPGA, las neuronas en hardware se implementaron en un FPGA Xilinx XC3090, ya que se valen de una técnica de reconfiguración en tiempo de ejecución. Los autores estiman que podría funcionar sobre los 40 MHz, aunque fue operado a una frecuencia de 14 MHz [26].

Algunas implementaciones de redes neuronales sobre FPGAs tienen como principal premisa un mínimo uso de recursos disponibles, sacrificando en cierto grado el niver de precisión y la velocidad de procesamiento. Aunque también es importante destacar que la función de activación utilizada en las neuronas de la red neuronal determina en gran parte el consumo total de recursos, por ello se vuelve indispensable el uso de algoritmos de aproximación a las funciones no lineales (tangente sigmoidal, sigmoide logarítmica, entre otras), entre los que destacan la linealización y el uso de tablas LUT [27, 28].

Savran y Ünsal [29] resuelven el problema de la XOR con 3 entradas mediante el uso de una red neuronal [3-5-1]. La red neuronal que es controlada por una máquina de estados que alimenta a las neuronas de la red. Cada neurona está compuesta por los siguientes elementos:

- ▷ Un multiplicador de 8 bits en la entradas y 16 en la salida.
- ▷ Una memoria ROM para almacenar los pesos.
- ▷ Un acumulador de 16 bits.

Los autores utilizaron una función de activación por capa ya que, debido al carácter serial de esta implementación, cada ciclo de reloj existe un resultado válido del acumulador. Es decir, para ejecutar una capa de 3 neuronas, es necesario que transcurran 3 ciclos de reloj para tener los resultados de las neuronas listas, aunque éstas se procesan en paralelo. Los resultados de las funciones de activación se calculan en MatLab y se vacían en archivos para inicializar memorias ROM en el FPGA Xilinx Spartan II. La red neuronal requiere de 20 ciclos de reloj para ser procesada completamente.

Sahin, Becerikli y Yazici [30] implementaron un modelo de red neuronal [2-3-1] en un FPGA Xilinx Spartan IIE. Utilizaron aritmética de punto flotante de 32 bits. Diseñaron un módulo de multiplicación y uno de suma en punto

flotante, los cuales utilizan la mayor cantidad de recursos del FPGA. Utilizan datos de 32 bits ya que determinaron que este ancho de datos mantiene un equilibrio entre los recursos utilizados y el tiempo requerido para realizar el cálculo de la red, ya que para minimizar los recursos se pudiesen utilizar datos de 8 bits, pero entonces la red no serviría para su uso como controlador de tiempo real. En esta estructura tienen un multiplicador y un sumador por capa, las salidas de una capa entran de manera paralela a la siguiente capa, mientras que los pesos lo hacen serialmente a los multiplicadores. Los resultados de la multiplicación se almacenan en la memoria ROM de cada neurona, estos resultados entran de forma serial al sumador, su resultado entra a una LUT. La función de activación es una sigmoide con un intervalo de -10 a 2 con pasos de 0,2. Debido a la estructura que fue implementada, solo una neurona, únicamente necesitan un multiplicador y un sumador, además de una tabla para almacenar los resultados de la función para cada neurona, cuya ejecución se repite tantas veces como neuronas existan en la red. En el artículo, los autores reportan un error de 0.059323 al calcular la red en el FPGA con respecto a una implementación en software (2.3274321322 en software contra 2.268109 en el FPGA) con entradas $g_1 = 0{,}5$ y $g_2 = -0{,}25$.

Otro modelo de red neuronal artificial implementado en FPGA es el propuesto por Himavathi, Anitha, y Muthuramalingam [31], éste se basa en el multiplexado de las capas, es decir sólo se tiene una capa en hardware, la cual tiene tantas neuronas como la capa que más neuronas tenga en la red.

Las neuronas se integran de los siguientes módulos:

- $MUL8$: módulo multiplicador de números de 8 bits con signo
- ADD: módulo sumador de dos números
- $SIGMA$: mdulo que obtiene la sumatoria de los productos de los pesos por las entradas y el umbral
- $SIGMA_CTRL$: unidad de control del procesamiento de la neurona, se encarga de obtener la sumatoria de todos los productos de los pesos por las entradas y el umbral de cada neurona
- LUT: es el módulo que devuelve el resultado de la función sigmoidal de acuerdo a la salida del módulo SIGMA

Con este modelo, una neurona puede tener 16 entradas sin presentar sobreflujo, aunque incrementando el ancho de palabra, se puede tener capacidad para un mayor número de entradas (en aplicaciones de tiempo real, raramente se excenden 16 entradas). Los datos tienen diferentes longitudes dependiendo de su funciń, como se aprecia en Tabla 2.1.

Tipo de Dato	**Signo** (bits)	**Parte Entera** (bits)	**Parte Fraccionaria** (bits)
Entradas	1	8	
Pesos	1	8	8
Umbrales	1	8	16
Producto Pesos-Entradas	1	24	
Resultado de Sumatoria	1	12	16
Salida de la función de activación	1	8	

Tabla 2.1: Longitud de datos utilizados en el modelo de la red neuronal artificial.

Este trabajo presenta una comparativa entre una neurona con función implementada en una LUT y una neurona con el cálculo de la función mediante la ecuación sigmoidal. Se obtuvo un ahorro de casi el 70 % de recursos mediante la LUT y un ahorro de casi 80 % en tiempo de ejecución.

Otra propuesta en cuanto a la implementación de redes neuronales artificiales en FPGAs es el trabajo de Maeda y Tada [32]. La propuesta se refiere al uso del *Método de perturbación simultánea*. La ventaja de este método con respecto al *backpropagation* radica en su simplicidad, ya que puede estimar el gradiente solamente con el uso de los valores de la función del error. En un experimento que reportan, se resuelve el problema de la XOR mediante una red [2-2-1] con un uso del FPGA del 24 % (60 000 compuertas de las 250 000 disponibles). La ejecución de la red neuronal inicia con la asignación aleatoria de los pesos sinápticos en el rango de [-255,+255] y una razón de cambio de $\alpha = 1{,}25$.

El trabajo realizado durante este semestre es capaz de resolver cualquier red neuronal, siempre y cuando está dentro de los límites de los recursos del FPGA, por lo que a diferencia del trabajo realizado por Savran y Ünsal [29], resulta una aplicación genérica. Una diferencia notable entre los artículos

anteriormente expuestos consiste en el hecho que se utilizaron LUTs para obtener el resultado de la función de activación (reduciendo de esta manera el tiempo de cálculo), mientras que este modelo linealiza la función almacenando en tablas solamente dos valores que serán procesados para obtener el resultado correcto. Esto hace más lento el procesamiento en general, pero a su vez disminuye de manera considerable la cantidad de recursos de memoria empleados.

A pesar de la existencia de distintas estrategias para implementar redes neuronales en FPGAs, su éxito depende de la implementación eficiente de una simple neurona. Los FPGAs son adecuados para albergar redes neuronales artificiales no solamente por el hecho de ofrecer cálculo en paralelo, sino que también pueden ser reprogramados fácilmente con la finalidad de modificar el diseño sin importar la magnitud de estos cambios [33].

En la figura 2.1 se muestra la cantidad de artículos publicados por año que son citados en esta tesis. Se observa que desde el año 1943 cuando McCulloch y Pitts [34] propusieron el primer modelo de neurona artificial hasta el 2006, se produjeron 20 artículos, sin embargo entre los años 2007 y 2009 se realizaron 17, y desde el 2010 hasta el presente año con el artículo de sensores virtuales de Galicia [9], se han publicado 14 artículos.

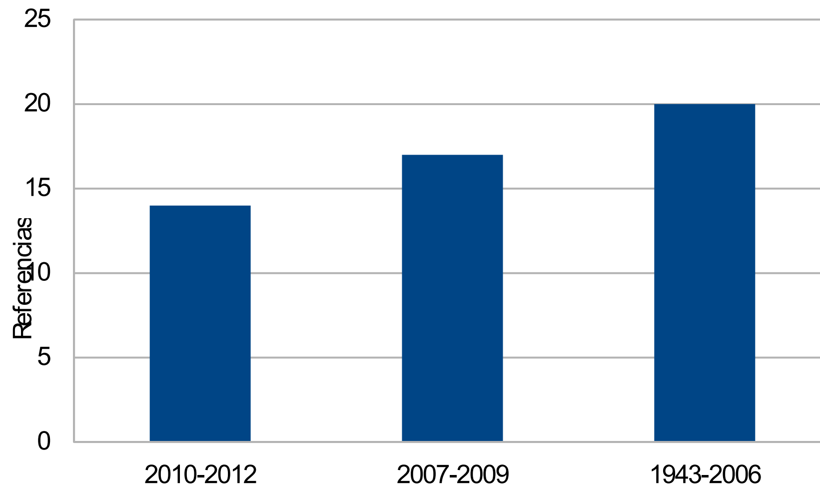


Figura 2.1: Referencias por año de publicación.

2.1. Resumen

A lo largo de este capítulo se hace mención de los trabajos previos que se han realizado en el campo de los sensores virtuales, donde se pone de manifiesto la diversidad de aplicaciones en las que pueden ser implementados con éxito. Los sensores virtuales reportados utilizan como base las redes neuronales artificiales, por ello también se incluyen trabajos de implementación en hardware de Redes Neuronales Artificiales. Los resultados reportados son muy variados, en algunos la precisión es el aspecto primordial mientras que en algunos otros lo importante es reducir al mínimo el uso de las unidades lógicas del dispositivo.

Otro aspecto importante de los trabajos es la implementación del algortimo *Backpropagation*, ya que se proponen arquitecturas muy diversas: múltiples FPGAs controlados por un microprocesador, reconfigración de módulos dentro del FPGA dependiendo de la etapa del proceso a ejecutar, entre otros.

Con el objetivo de resaltar la investigación realizada a lo largo de esta tesis, el siguiente gráfico muestra la cantidad de artículos de revistas científicas, congresos, libros, tesis y manuales que fueron referenciados.

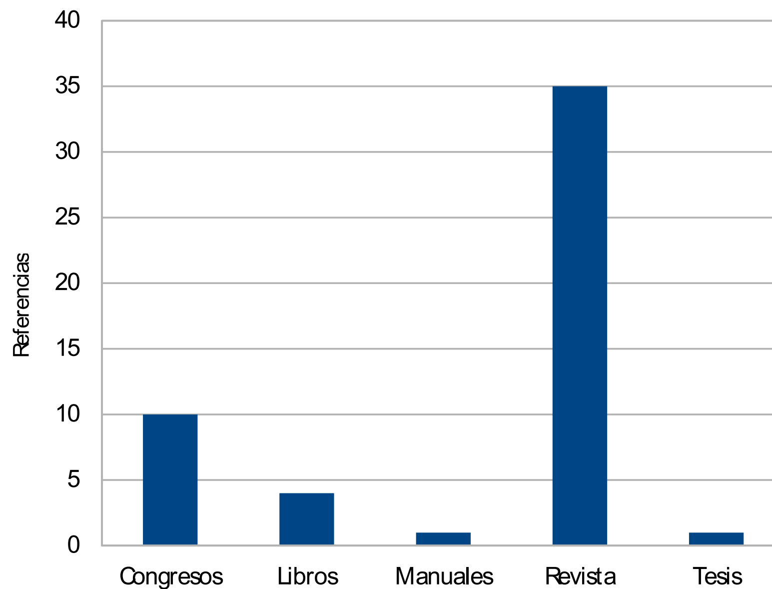


Figura 2.2: Referencias por tipo de publicación.

Capítulo 3

Marco Teórico

Las neuronas artificiales imitan el funcionamiento de las células neuronales biológicas que constituyen el cerebro de los seres vivos. Una red neuronal es un arreglo de neuronas dispuestas en capas interconectadas entre sí. Una red neuronal se compone, generalmente de tres capas: capa de entrada, capa oculta y capa de salida; aunque existen algunas aplicaciones que únicamente requieren el uso de la capa de entrada y la capa de salida.

La estructura de una neurona artificial es un modelo de la célula biológica, que se compone de entradas, sumatoria y una función de activación: ya sea la función lineal (utilizada en la capa de salida) o bien alguna función no lineal, ya sea la sigmoidal o la tangente hiperbólica, la cual se emplea frecuentemente en aplicaciones de clasificación.

Debido al carácter cognitivo de las redes neuronales arficiales, una de sus aplicaciones más extendidas es en la aproximación de señales no lineales. Esto se realiza mediante algoritmos de aprendizaje, en los que se ajustan iterativamente los pesos sinápticos y los umbrales de las neuronas para obtener los resultados esperados minimizando así el error de salida.

3.1. Neurona Biológica

El desarrollo de la teoría de las Redes Neuronales Artificiales comienza en 1943 cuando el neurobiólogo Walter Pitts y el matemático Warren McCulloch proponen en conjunto el primer modelo de la neurona artificial [34]. Posteriormente en 1958, Frank Rosenblatt desarrolla el *Perceptrón*, que consta de dos capas y es capaz de procesar la información en paralelo [35, 36].

En 1986 se agrega una capa oculta al Perceptrón con la finalidad de entrenar la red para resolver tareas más complejas de clasificación, a este nuevo modelo se le conoce en inglés como *Feed-Forward Back-Propagation Neural Network* [37]. El algoritmo de aprendizaje que fue presentado por McLelland y Rumelhart es el más ampliamente utilizado a pesar de que han surgido más reglas de aprendizaje.

La neurona artificial brinda un modelo matemático de la neurona biológica. Para comprender dicho modelo, la neurona biológica se muestra gráficamente en su forma general en la figura 3.1.

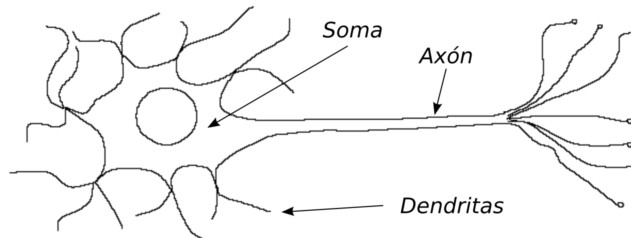


Figura 3.1: Modelo de la neurona biológica.

La neurona se compone de las partes básicas:

- ▷ *Soma* es el lugar donde se llevan a cabo las funciones de procesamiento de los estímulos recibidos, también es dónde se realiza la síntesis de proteínas.
- ▷ *Axón* es un nervio en el cual la señal transmitida por el Soma se convierte en impulsos nerviosos que serán propagados a otras células.
- ▷ *Dendritas* son fibras ramificadas conectadas al Soma que reciben información mediante la Sinápsis.

▷ *Sinápsis* son las terminales de los axones que se conectan a otras neuronas. Debido a la naturaleza inhibitoria o excitatoria de la sinápsis, es posible atenuar o incrementar el nivel de excitación de la neurona.

3.2. Neurona Artificial

La neurona artificial es una representación simplificada de la neurona biológica, cuyos procesos son tan complejos que actualmente no se conocen en su totalidad. Dicha representación es flexible con la finalidad de adaptarse a la función en la que se vaya a emplear [34].

Una red neuronal es una estructura que procesa información de forma paralela y distribuida, que consta de elementos de procesamiento interconectados entre sí, cada elemento tiene una memoria interna y procesa las senãles que recibe del exterior. Cada elemento tiene una salida que se ramifica en tantas conexiones como el problema a resolver lo requiera. La señal de salida del elemento puede ser de cualquier tipo matemático deseado. Todo el procesamiento que ocurre dentro de cada elemento debe ser completamente local: depende únicamente de los valores actuales de las señales de entrada llegan a la elemento a través de conexiones que inciden y en los valores almacenados en la memoria local del elemento de procesamiento [38].

Las neuronas artificiales imitan el funcionamiento del modelo de la neurona biológica presente en el cerebro de los seres vivos. Una neurona artificial de una sola entrada, como la que aparece en la figura 3.2 se puede relacionar fácilmente con una neurona biológica (figura 3.1): el peso sináptico (w) es la fuerza de la sinápsis, el cuerpo de la neurona es la sumatoria (Σ) y la función de activación (f), mientras que la salida (a) es la señal en el axón [39].

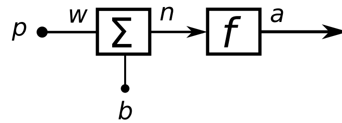


Figura 3.2: Neurona artificial con una entrada.

La salida de la neurona de una entrada se caracteriza en la siguiente ecuación:

$$n = f(wp + b)$$
$$a = f(n) \tag{3.1}$$

Función de Activación

La función $f(n)$ corresponde a la función de activación usada en la neurona. Éstas funciones pueden ser lineales o bien, no lineales dependiendo del uso en el que se vaya a utilizar la red neuronal. Las principales funciones de activación son las siguientes:

▷ Funciones Lineales

- *Lineal (pureline)*

$$f(n) = n \tag{3.2}$$

- *Escalón (hardlimit)*

$$f(n) = \begin{cases} 0 : n < 0 \\ 1 : n \geq 1 \end{cases} \tag{3.3}$$

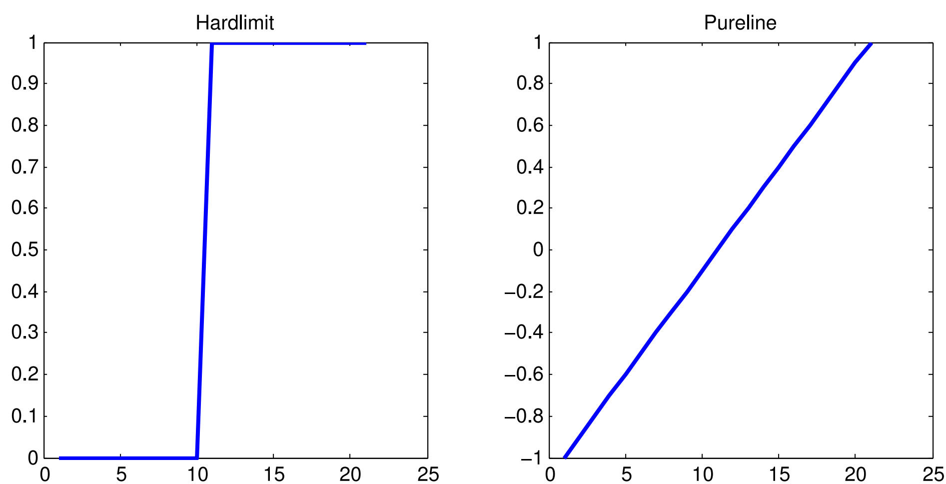


Figura 3.3: Funciones de activación lineales.

▷ Funciones No Lineales

- *Sigmoide logarítmica (logsig)*

$$f(n) = \frac{1}{1 + e^{-n}} \tag{3.4}$$

- *Sigmoide Tangente Hiperbólica (tansig)*

$$f(n) = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}} \tag{3.5}$$

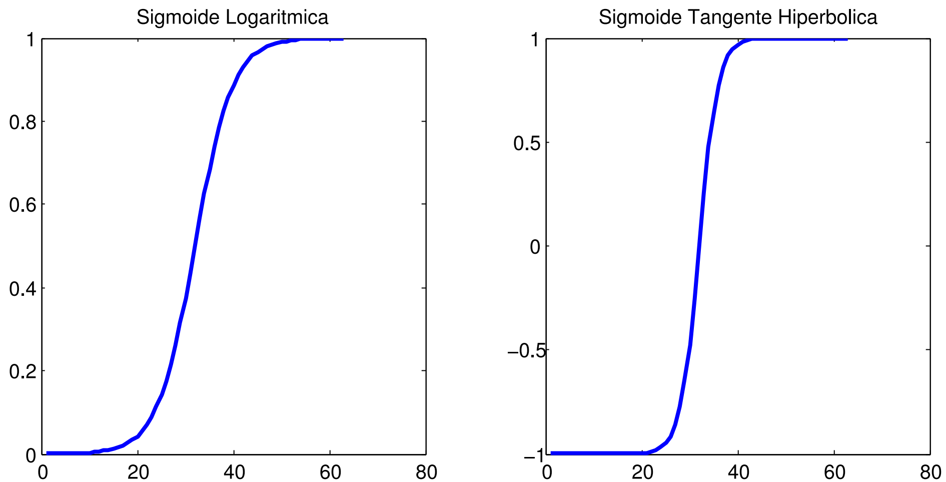


Figura 3.4: Funciones de activación no lineales.

Las funciones lineales se emplean para redes neuronales del tipo *ADALINE* por su sencillez, mientras que la función sigmoidal logarítmica es de uso mayoritariamente en redes neuronales multicapa entrenadas con el algoritmo *backpropagation*. La función Tangente hiperbólica sigmoidal es muy utilizada en redes multicapa para tareas de clasificación.

Neurona de múltiples entradas

Es muy común que una neurona artificial posea múltiples entradas, con pesos sinápticos y un bias, mientras a que a la salida tiene una función de activación $f(x)$.

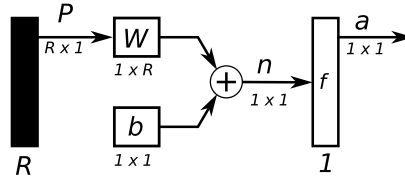


Figura 3.5: Neurona artificial con múltiples entradas.

El procesamiento de la neurona comienza con la suma de las entradas multiplicadas por sus pesos sinápticos respectivos más el bias:

$$n = \sum_{i=1}^{n} w_i \cdot p_i + b$$

donde $w_i \cdot p_i$ es la iésima conexión y b es el bias, mientras que la salida de la neurona se representa como:

$$a = f(n)$$

3.3. Tipos de Redes Neuronales Artificiales

Muchos de los problemas existentes no se pueden resolver aún con una neurona con muchas entradas, de aquí surgió la necesidad de disponer de un conjunto de neuronas, que al trabajar en paralelo forman una *capa*.

3.3.1. Capas de neuronas

Una red neuronal de una capa con R entradas y S neuronas se muestra en la figura 3.6. Cada entrada p está conectada a todas las neuronas con un peso w, toda neurona tiene un bias b, un sumador $\sum$, una función de activación f^1 y una salida a. Las neuronas dentro de una misma capa pueden tener diferentes funciones de activación.

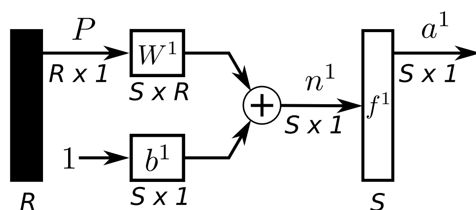


Figura 3.6: Capa de S neuronas.

3.3.2. Redes multicapa

Una red neuronal multicapa como la figura 3.7 agrega complejidad al desarrollo de la misma ya que cada capa tiene sus propios pesos sinápticos y sus bias. Estos parámetros que se agregan en una red multicapa se referencían mediante el superíndice que aparece en cada uno de los elementos, de esta manera se puede tener una red con R entradas, S^1 neuronas en su primer capa y S^2 neuronas en su segunda capa, aunque cada capa puede tener distinta cantidad de neuronas.

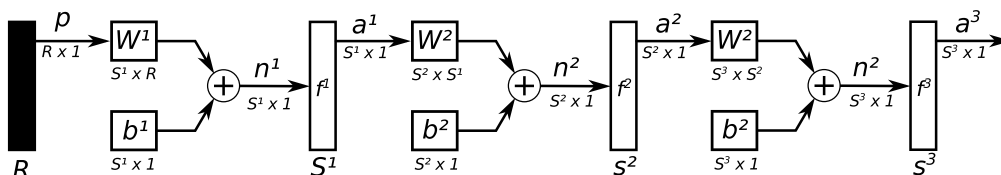


Figura 3.7: Red neuronal multicapa.

Como se puede observar en la figura 3.7, las salidas de la primer capa se convierten en las entradas a la segunda capa.

La capa que arroja los resultados de la red se conoce como *capa de salida* y las capas existentes entre las entradas y la capa de salida son *capas ocultas.*

Una gran ventaja de las redes neuronales multicapa frente a las redes de solo una capa radica en su poder: una red de dos capas con función de activación sigmoidal en su capa oculta y función lineal en la capa de salida puede ser entrenada para aproximar señales con un buen nivel de certeza, mientras que las redes de una sola capa no pueden hacer esto.

El diseño de una red neuronal puede llegar a ser problemático en sí, aunque pudiesen presentarse situaciones que ayuden en este punto: el número de entradas y salidas de la red depende enteramente de las especificaciones del problema a resolver y el tipo de la señal de salida también puede ayudar

a que función de activación utilizar. La arquitectura de una red de una capa está dada casi en su totalidad por el entorno, mientras que para una red multicapa se vuelve una tarea mucho más compleja. En una red multicapa, no es posible obtener la cantidad de neuronas de la capa oculta del exterior, aunque existen métodos para predecir este número.

3.4. Perceptrón multicapa

El modelo del Preceptrón multicapa (MLP, siglas en inglés de *Multilayer Perceptron*) es un aproximador universal, es decir, cualquier función continua puede ser aproximada con un Perceptrón multicapa, con al menos una capa oculta. Esta capacidad ha hecho del Perceptrón multicapa una de las herramientas más utilizadas para la solución de diversos problemas: clasificación y reconocimiento de patrones, control de procesos, modelado de sistemas, predicción de series, entre otros.

El hecho que el Perceptrón multicapa sea ampliamente utilizado, no quiere decir que sea una de las redes más potentes, entre las limitaciones que presenta destaca su largo proceso de aprendizaje en problemas muy complejos, por otra parte, muchas veces resulta muy complicado asignar valores numéricos a las variables del entorno.

3.4.1. Arquitectura del Perceptrón multicapa

Perceptrón multicapa se compone de tres tipos de capas: entradas, capas ocultas y capa de salida (ver figura 3.8). Esta arquitectura es una cascada de capas de neuronas artificiales. Las entradas ingresan las señales a la red, mientras que a capa de salida envía el resultado del procesamiento de los realizado por las neuronas de las capas ocultas. La estructura de un Perceptrón multicapa se suele representar con el número de entradas, seguido del número de neuronas de cada capa:

$$R - S^1 - S^2 - S^3$$

Las conexiones en el Perceptrón siempre van dirigidas hacia adelante, por lo que se le conoce como red *“feedforward”* y por lo general cada una de las neuronas de una capa están conectadas a todas las neuronas de la capa siguiente, por lo que se dice que es una red totalmente conectada.

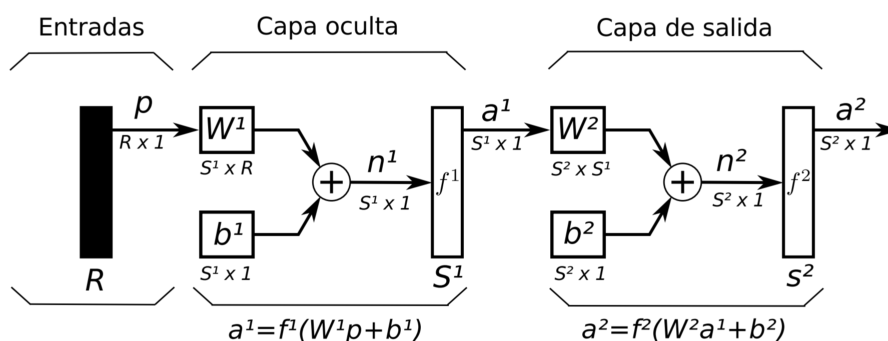


Figura 3.8: Perceptrón Multicapa en representación matricial.

3.4.2. Propagación hacia adelante

El Perceptrón multicapa define una relación entre las variables de entrada y las variables de salida. Dicha relación se establece mediante el cálculo que realiza cada neurona de las entradas que recibe y produce una salida que propaga mediante las conexiones respectivas a la siguiente capa.

El procesamiento realizado para esta arquitectura de red neuronal es el siguiente:

1. Activación de las entradas: se transmiten las señales recibidas del exterior hacia dentro de la red.

2. Activación de las neuronas de la capa oculta: las neuronas ocultas reciben las señales de entrada y los umbrales, son sumados y al resultado se le aplica la función de activación (tangente sigmoidal o sigmoide).

3. Activación de la capa de salida: se realiza un procesamiento similar al de las capas ocultas. En esta capa es común que la función de activación sea la función lineal.

3.5. Aprendizaje de la Red Neuronal Artificial

3.5.1. Criterios de Finalización

En muchos de los problemas que se resuelven mediante el uso del método *Backpropagation*, no se puede garantizar que el error de aprendizaje (E_{It}) converja a cero. Sin embargo existen criterios que pueden ser usados para

finalizar el ajuste de los parámetros de acuerdo al problema a resolver. A continuación se presentan los criterios de finalización más utilizados en la implementación de redes neuronales artificiales:

1. Existen algunos problemas cuya complejidad es poca, con lo que el error de la iteración llega a ser cero, es decir, se resuelve en su totalidad el problema, así que también este constituye un criterio de finalización.

$$E_{It} = 0$$

2. Un criterio muy utilizado para detener el entrenamiento de redes neuronales se refiere al hecho de fijar un valor (ε) lo suficientemente pequeño como para considerar que la red ha sido entrenada en el caso que el error de iteración caiga por debajo de este valor. Generalmente se tienen valores entre 0,1 y 0,01. Desafortunadamente este criterio puede terminar prematuramente el entrenamiento.

$$E_{It} \leq \varepsilon$$

3. El criterio más extremo utilizado en el entrenamiento de las redes neuronales consiste en terminar el entrenamiento de la red en el momento que se alcance una cantidad de iteraciones considerable (It_{max}), esto minimiza el riesgo de que el entrenamiento caiga en un mínimo local.

$$It = It_{max}$$

4. Adicional a estos criterios, se utiliza el algoritmo *Early stopping* que previene el sobreentrenamiento de la red neuronal. Consiste en que cada n intervalo de iteraciones, se ejecuta una iteración de prueba y se calcula el error, que entonces se nombra como *error de validación* (e_{val}); entonces se reanuda el entrenamiento durante las mismas n iteraciones, al final de las cuales se calcula nuevamente el error, se comparan ambos errores y entonces cuando el error crece dos o más veces consecutivas, se termina el entrenamiento. Los valores de los parámetros finales de la red serán los que se tenían en el instante previo a que el error comenzara a crecer.

3.5.2. Diseño de las Reglas de Aprendizaje

El aprendizaje de la Red Neuronal Artificial que se desarrolló en este trabajo está basado en el método del gradiente descendiente, en el que el error se minimiza sucesivamente para cada patrón de entrada. Para ello, es necesario establecer reglas de aprendizaje mediante las siguientes ecuaciones.

$$w(k+1) = w(k) - \alpha \frac{\partial e(k)}{\partial w} \tag{3.6}$$

$$b(k+1) = b(k) - \alpha \frac{\partial e(k)}{\partial b} \tag{3.7}$$

donde α representa la velocidad de convergencia al valor deseado (*razón de aprendizaje*): un valor muy grande ocasiona que la red tarde menos en llegar al objetivo, mientras que uno más pequeño brinda un ajuste más fino [38].

Los valores de $w(k)$ y $b(k)$ cambian mediante las derivadas parciales de $e(k)$ respecto a las variables w y b. A este proceso se les conoce como propagación hacia atrás del error (*backpropagation*).

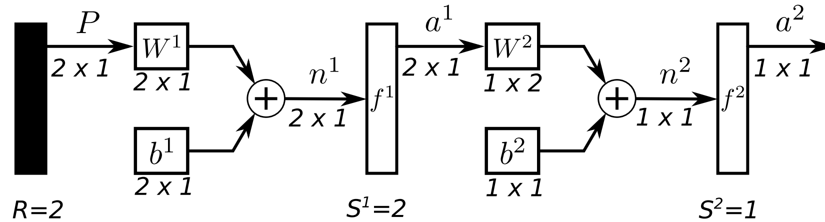


Figura 3.9: Red neuronal [2-2-1].

Para desarrollar de mejor manera el algoritmo *Backpropagation* se tomará como ejemplo una red neuronal [2-2-1] como la de la figura 3.9, sus ecuaciones de la propagación hacia adelante son las siguientes:

$$a_1^1 = f^1(w_{11}^1 \cdot p_1 + w_{21}^1 \cdot p_2 + b_1^1)$$

$$a_2^1 = f^1(w_{12}^2 \cdot p_1 + w_{22}^1 \cdot p_2 + b_2^1)$$

$$a^2 = f^2(w_{11}^2 \cdot a_1^1 + w_{21}^2 \cdot a_2^1 + b_1^2) \tag{3.8}$$

Es necesario tener en cuenta la ecuación del error que será empleada para obtener las reglas de aprendizaje, en este caso se trata del *Error cuadrático medio* descrito por la ecuación (3.9).

$$e(k) = \frac{1}{2}(a_{target} - a^2)^2 \tag{3.9}$$

Regla de la Cadena

El error en el Perceptrón multicapa no es función explícita de los pesos sinápticos en las capas ocultas, por lo tanto sus derivadas requieren cálculos no triviales. Esto implica utilizar la regla de la cadena para derivar los pesos sinápticos.

Se tiene una función f que es una función explícita de una variable n y se desea derivar f con respecto a una tercer variable w, entonces la regla de la cadena queda como sigue:

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \cdot \frac{dn(w)}{dw}$$

Esto es de utilidad para hallar las derivadas en 3.6 y 3.7.

Regla δ generalizada

Con base en las ecuaciones (3.8) y (3.9) se determinan las reglas de aprendizaje de los pesos sinápticos y el umbral de la capa de salida:

$$w_{11}^2(k+1) = w_{11}^2(k) - \alpha \frac{\partial e(k)}{\partial w_{11}^2}$$

$$w_{21}^2(k+1) = w_{21}^2(k) - \alpha \frac{\partial e(k)}{\partial w_{21}^2}$$

$$b_1^2(k+1) = b_1^2(k) - \alpha \frac{\partial e(k)}{\partial b_1^2}$$

Ahora se determinan las reglas de aprendizaje para la capa oculta:

$$w_{11}^1(k+1) = w_{11}^1(k) - \alpha \frac{\partial e(k)}{\partial w_{11}^1}$$

$$w_{12}^1(k+1) = w_{12}^1(k) - \alpha \frac{\partial e(k)}{\partial w_{12}^1}$$

$$w_{21}^1(k+1) = w_{21}^1(k) - \alpha \frac{\partial e(k)}{\partial w_{21}^1}$$

$$w_{22}^1(k+1) = w_{22}^1(k) - \alpha \frac{\partial e(k)}{\partial w_{22}^1}$$

$$b_1^1(k+1) = b_1^1(k) - \alpha \frac{\partial e(k)}{\partial b_1^1}$$

$$b_2^1(k+1) = b_2^1(k) - \alpha \frac{\partial e(k)}{\partial b_2^1}$$

En las expresiones anteriores aparecen términos que incluyen derivadas parciales, se trata de la *regla δ generalizada*, que serán resueltos mediante la regla de la cadena antes expuesta. La regla δ generalizada tendrá las siguientes formas:

▷ Capa de salida

- Pesos sinápticos

$$\begin{aligned} \frac{\partial e(k)}{\partial w_{i1}^2} &= \frac{\partial e(k)}{\partial a} \cdot \frac{\partial a}{\partial f^2} \cdot \frac{\partial f^2}{\partial w_{i1}^2} \\ &= -(a_{target} - a^2) \cdot 1 \cdot a_i^1 \end{aligned}$$

$$= -(a_{target} - a^2) \cdot a_i^1 \tag{3.10}$$

- Bias

$$\begin{aligned} \frac{\partial e(k)}{\partial b_1^2} &= \frac{\partial e(k)}{\partial a} \cdot \frac{\partial a}{\partial f^2} \cdot \frac{\partial f^2}{\partial b_1^2} \\ &= -(a_{target} - a^2) \cdot 1 \cdot 1 \end{aligned}$$

$$= -(a_{target} - a^2) \tag{3.11}$$

▷ Capa oculta

- Pesos sinápticos

$$\begin{aligned}\frac{\partial e(k)}{\partial w_{ij}^1} &= \frac{\partial e(k)}{\partial a^2}\cdot\frac{\partial a^2}{\partial f^2}\cdot\frac{\partial f^2}{\partial a_j^1}\cdot\frac{\partial a_j^1}{\partial f^1}\cdot\frac{\partial f^1}{\partial w_{ij}^1}\\ &= -(a_{target}-a^2)\cdot 1\cdot w_{ij}^1\cdot f^1(1-f^1)\cdot p_i\end{aligned}$$

$$= -(a_{target}-a^2)\cdot w_{ij}^1\cdot f^1(1-f^1)\cdot p_i \tag{3.12}$$

- Bias

$$\begin{aligned}\frac{\partial e(k)}{\partial b_j^1} &= \frac{\partial e(k)}{\partial a^2}\cdot\frac{\partial a^2}{\partial f^2}\cdot\frac{\partial f^2}{\partial a_j^1}\cdot\frac{\partial a_j^1}{\partial f^1}\cdot\frac{\partial f^1}{\partial b_j^1}\\ &= -(a_{target}-a^2)\cdot 1\cdot w_{ij}^1\cdot f^1(1-f^1)\end{aligned}$$

$$= -(a_{target}-a^2)\cdot w_{ij}^1\cdot f^1(1-f^1) \tag{3.13}$$

En las ecuaciones anteriores hay dos índices: i y j, éstos se refieren al número de la neurona origen y a la neurona destino respectivamente. Una vez que se tienen desarrollados los términos mediante la regla de la cadena, las reglas de aprendizaje quedan como sigue:

Capa de salida

$$w_{11}^2(k+1) = w_{11}^2(k) - \alpha(-(a_{target}-a^2)\cdot a_1^1)$$

$$w_{21}^2(k+1) = w_{21}^2(k) - \alpha(-(a_{target}-a^2)\cdot a_2^1)$$

$$b_1^2(k+1) = b_1^2(k) - \alpha(-(a_{target}-a^2))$$

Capa oculta

$$\begin{aligned}w_{12}^1(k+1) &= w_{12}^1(k) - \alpha\cdot(-(a_{target}-a^2)\cdot w_{11}^2\cdot f_1(1-f_1)\cdot p_1\\ w_{11}^1(k+1) &= w_{11}^1(k) - \alpha\cdot(-(a_{target}-a^2)\cdot w_{11}^2\cdot f_1(1-f_1)\cdot p_1\end{aligned}$$

$$b_1^1(k+1) = b_1^1(k) - \alpha \cdot (-(a_{target} - a^2) \cdot w_{11}^2 \cdot f_1(1-f_1)$$

$$\begin{aligned} w_{21}^1(k+1) &= w_{21}^1(k) - \alpha \cdot (-(a_{target} - a^2) \cdot w_{21}^2 \cdot f_1(1-f_1) \cdot p_2 \\ w_{22}^1(k+1) &= w_{22}^1(k) - \alpha \cdot (-(a_{target} - a^2) \cdot w_{21}^2 \cdot f_1(1-f_1) \cdot p_2 \end{aligned}$$

$$b_2^1(k+1) = b_2^1(k) - \alpha \cdot (-(a_{target} - a^2) \cdot w_{11}^2 \cdot f_1(1-f_1)$$

Cálculo de Sensitividades

De las ecuaciones 3.10, 3.11, 3.12 y 3.13 es posible factorizar algunos términos comunes y así, facilitar el procedimiento de cálculo. El primer término a factorizar es la Sensitividad de la capa de salida (S^2).

$$S^2 = -(a_{target} - a^2) \tag{3.14}$$

Entonces se redefinen las reglas de aprendizaje de la capa de entrada, a las ecuaciones 3.12 y 3.13 se factorizan nuevamente los términos en común. El nuevo término será la Sensitividad de la capa oculta (S_i^1).

$$\frac{\partial e(k)}{\partial w_{ij}^1} = \frac{\partial e(k)}{\partial a} \cdot \frac{\partial a}{\partial f^2} \cdot \frac{\partial f^2}{\partial a_j^1} \cdot \frac{\partial a_j^1}{\partial f^1} \cdot \frac{\partial f^1}{\partial w_{ij}^1}$$

$$S_i^1 = S^2 \cdot w_{ij}^2 \cdot f^1(1-f^1) \tag{3.15}$$

El término $f^1(1-f^1)$ corresponde a la derivada de la función de activación de las neuronas de la capa oculta (ver Apéndice A); dado que la función presente en la neurona de la capa de salida es la función lineal y su primer derivada es 1, no aparece explícita. Como resultado de las factorización de las Sensitividades se tienen las siguientes reglas de aprendizaje:

▷ Capa de salida

$$\begin{aligned} w_{11}^2(k+1) &= w_{11}^2(k) - \alpha \cdot S^2 \cdot a_1^1 \\ w_{21}^2(k+1) &= w_{21}^2(k) - \alpha \cdot S^2 \cdot a_2^1 \end{aligned}$$

$$b_1^2(k+1) = b_1^2(k) - \alpha \cdot S^2$$

▷ Capa oculta

$$
\begin{aligned}
w_{11}^{1}(k+1) &= w_{11}^{1}(k) - \alpha \cdot S_{1}^{1} \cdot p_{1} \\
w_{12}^{1}(k+1) &= w_{12}^{1}(k) - \alpha \cdot S_{2}^{1} \cdot p_{1}
\end{aligned}
$$

$$b_{1}^{1}(k+1) = b_{1}^{1}(k) - \alpha \cdot S_{1}^{1}$$

$$
\begin{aligned}
w_{21}^{1}(k+1) &= w_{21}^{1}(k) - \alpha \cdot S_{1}^{1} \cdot p_{2} \\
w_{22}^{1}(k+1) &= w_{22}^{1}(k) - \alpha \cdot S_{2}^{1} \cdot p_{2}
\end{aligned}
$$

$$b_{2}^{1}(k+1) = b_{2}^{1}(k) - \alpha \cdot S_{2}^{1}$$

3.5.3. Ecuaciones Generales

En el momento en el que ya se tienen las reglas de aprendizaje, es posible obtener formas generales para cada uno de los parámetros de la red neuronal [40].

Propagación hacia adelante

Las ecuaciones generales de la propagación hacia adelante de las señales de entrada para una red neuronal de M capas son las siguientes:

$$a^{0} = P$$

$$a^{m+1} = F^{m+1}(W^{m+1} \cdot a^{m} + b^{m+1})$$

$$a = a^{M} \tag{3.16}$$

donde $m = 0, 1, 2, M - 1$.

Aprendizaje

Como parte de la retropropagación, es necesario conocer las Sensitividades tanto de las capas ocultas como de la capa de salida, las cuales están definidas por:

$$S^M = -\dot{F}^M(n^M) \cdot (a_{target} - a^2) \tag{3.17}$$

$$S^m = \dot{F^m}(n^m) \cdot (W^{m+1})^T (S^{m+1}) \tag{3.18}$$

para $M = M - 1, \cdots, 2, 1$ y donde $\dot{F^m}$ es la derivada de la función de activación de las capas ocultas, mientras que $\dot{F^M}$ es la derivada de la función de la capa de salida. Las Reglas de aprendizaje de los pesos sinápticos y los bias de la red neuronal artificial quedan de la siguiente manera:

$$W^m(k+1) = W^m(k) - \alpha S^m (a^{m-1})^T \tag{3.19}$$

$$b^m(k+1) = b^m(k) - \alpha S^m \tag{3.20}$$

3.5.4. Proceso de aprendizaje

Se tiene un conjunto $\{(p_1(1), a_{target1}(1)), \cdots, (p_n(k), a_{targetn}(k))\}$ de patrones de entrada para el problema, entonces el proceso de aprendizaje se describe a continuación:

Paso (1). Se toma un conjunto de muestras y se divide de la siguiente manera:

- ▷ 70 % datos de entrenamiento
- ▷ 15 % datos de validación
- ▷ 15 % de prueba

Paso (2). Se inicializan de forma aleatoria los pesos sinápticos y los umbrales dentro del rango [-1,1].

Paso (3). Se aplica como entrada a la red un patrón del conjunto de entrenamiento obteniendo una salida a.

Paso (4). Se calcula el error mediante la ecuación (3.9).

Paso (5). Utilizando el error obtenido en el paso anterior, se aplican las reglas de aprendizaje para cada uno de los parámetros de la red mediante las ecuaciones (3.19) y (3.20).

Paso (6). Se repiten desde Paso (3) hasta Paso (5) para todos los valores del conjunto de entrenamiento.

Paso (7). Se calcula el error de iteración (E_{it}), que en este punto toma en nombre de *Error de entrenamiento.*

Paso (8). Se repite Paso (3) a Paso (7) hasta cumplir alguno de los cuatro criterios de finalización:

a) $E_{it} = 0$

b) $E_{it} \leq \varepsilon$

c) Alcanzar un It_{max}

d) *Early Stopping*

En el segundo y tercer criterio de finalización se utilizan dos parámetros importantes que se definen a continuación:

- ε: valor mínimo que se establece para continuar con el proceso de entrenamiento de la red neuronal.

- It_{max}: número máximo de iteraciones que serán ejecutadas en el entrenamiento de la red neuronal.

Estos criterios previenen que la red neuronal caiga en mínimos locales, o bien, que la red sea sobreentrenada y que sus resultados para la propagación hacia adelante de las entradas presente grandes fallas [41].

3.6. Resumen

A lo largo de este capítulo se describieron todos aquellos aspectos teóricos que rigen el comportamiento de las redes neuronales artificiales, en especial el modelo *Perceptrón multicapa*. Al describir este modelo, se incluyen las ecuaciones útiles para realizar la propagación hacia adelante (*Feed-Forward*) del mismo. Esto se debe al uso de las redes neuronales como metodología para diseñar el sensor virtual propuesto.

El algoritmo de aprendizaje conocido como *Backpropagation* también fue descrito, con la finalidad de brindarle al lector el fundamento matemático que se tomó como base para el desarrollo de la implementación en un FPGA del algoritmo de aprendizaje.

Capítulo 4

Desarrollo de la Investigación

La implementación del sensor virtual basado en el algoritmo de aprendizaje *Backpropagation* para una Red Neuronal Artificial se realizó en una tarjeta Altera DE2-70 (figura 4.1) que incluye un FPGA de la serie Cyclone II, el cual contiene cerca de $70,000$ LEs y poco más de 1 MB de memoria RAM [42].

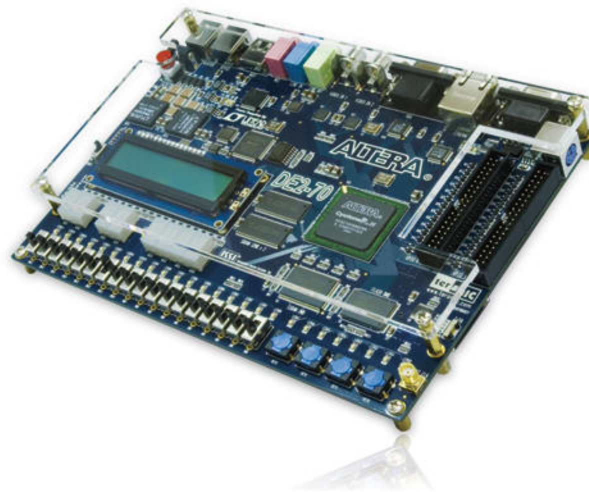


Figura 4.1: Tarjeta Altera DE2 70.

4.1. Panorama General

Esta implementación consta de dos etapas principalmente: la propagación hacia adelante de las señales de entrada (*feedforward*) y la propagación hacia atrás del error que se presenta a cada ejecución de un patrón de entrada a la red neuronal (*backpropagation*).

La estructura del proyecto se basa en la construcción. Algunos de estos módulos fueron programados, los demás corresponden a plantillas que provee Altera mediante el entorno de desarrollo Quartus II.

Las platillas corresponden a memorias ROM y RAM, unidades de aritmética de punto flotante (sumadores, restadores, multiplicadores, divisores) y a convertidores de *Integer* a punto flotante y viceversa. Todos los datos con los que realiza cálculos la red neuronal tienen un formato de punto flotante de 32 bits, cuya estructura se muestra en la figura 4.2.

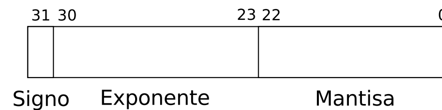


Figura 4.2: Formato de punto flotante de 32 bits.

El bit más significativo corresponde al signo (1: negativo, 0: positivo), los siguientes 8 bits (30-23) corresponden al exponente y los restantes 23 bits (22-0) serán utilizados para la mantisa, que define la parte fraccionaria del número.

En la figura 4.3 se muestra el proyecto de una forma general, en la que se observa el procesamiento que lleva a cabo el FPGA para realizar el entrenamiento:

▷ PC

1. Configuración de la RNA con interfaz MatLab: la aplicación desarrollada en MatLab permite al usuario el modelado de la red neuronal, elige la estructura, e ingresa el conjunto de entrenamiento y los parámetros para fijar parámetros del entrenamiento (ε y γ).
2. Envío de datos al FPGA: convierte los datos (pesos sinápticos, bias, parámetros de configuración) que se encuentran en un formato comprensible por la PC a un formato de punto flotante IEEE que es utilizado por los módulos del FPGA y los envía vía puerto serial al FPGA.

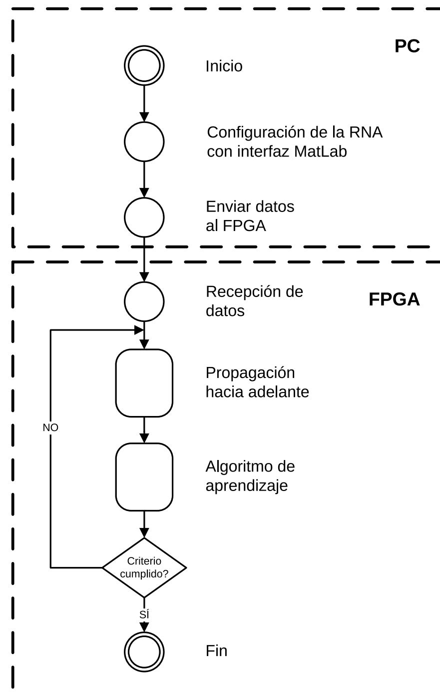


Figura 4.3: Diagrama general de la Red Neuronal y su aprendizaje.

▷ FPGA

1. Recepción de datos: el FPGA recibe los datos y los almacena en sus memorias respectivas para su posterior uso en los cálculos.
2. Propagación hacia adelante: calcula la salida de la red con los pesos sinápticos, bias y entradas que en ese momento tenga la red.
3. Algoritmo de aprendizaje: modifica los pesos sinápticos y bias conforme al error obtenido en cada iteración, también actualiza el valor de las entradas a los siguientes para que sean ejecutados en una futura propagación hacia adelante.
4. Verificación de criterios de finalización: se revisan los criterios de finalización y su relación con los parámetros que han sido dados por el usuario para una posible terminación del entrenamiento de la red neuronal.

4.2. Propagación hacia adelante de las señales

La etapa de propagación hacia adelante consiste en el hecho de aplicar las ecuaciones de propagación hacia adelante de las señales de entrada a la red neuronal, por ello se implementó una máquina de estados finitos (FSM, siglas del inglés *Finite State Machine*). Dicha FSM controla todas las transiciones entre los estados que realizan diversas funciones: carga de datos en memorias, cálculos, entre otros.

La figura 4.4 ilustra la ejecución de la FSM, en la que se observan estados cuyo color más oscuro, esto indica que son estados de retardo para ajustarse a tiempos de almacenamiento o lectura en memoria, o bien, de espera de fin de ejecución de módulos. Por otra parte, se observan también bloques que delimitan las etapas por las que pasa lan los datos para llegar al punto final de la ejecución de la red neuronal en su totalidad. Los bloques que intervienen en el procesamiento son los siguientes:

- ▷ Llenado de memorias: los datos recibidos vía puerto serial se almacenan en las memorias respecivas. La interfaz de MatLab envía los datos en una secuencia que el FPGA mantiene con la finalidad de guardar los datos donde corresponde. La mayoría de las memorias son de tipo RAM, a excepción de dos ROM, que son las tablas LUT para los para la *Abscisa* y la *Pendiente* utilizadas para la función sigmoidal.

- ▷ Carga de entradas en memorias temporales: se cargan los datos que utiliza la red neuronal como entradas

- ▷ Carga de la estructura: los valores de la cadena de la estructura se cargan para ejecutar la red neuronal, existen dos índices: *origen* es el número de neuronas de la capa actual y *destino*, el número de neuronas de la capa siguiente.

- ▷ Carga de bias: asigna el valor del bias de la neurona a las variables temporales. Este bloque se ejecuta cada vez que una nueva neurona será calculada.

- ▷ Carga de pesos sinápticos: se alistan los valores de los pesos sinápticos a utilizar para resolver el cálculo de las neuronas.

- ▷ Cálculo de neurona: una vez que se tienen listos los valores de las entradas al módulo que realiza las funciones de una neurona, se procede a su

respectivo cómputo. Cuándo existe más de una entrada, se ejecuta la carga de pesos y entradas hasta que se resuelve la sumatoria de todas las entradas y el bias. En apartados posteriores se aborda con mayor profundidad el procesamiento realizado por el módulo ***Multadder***.

- ▷ Función de activación: este bloque calcula la salida de una neurona, ya sea mediante la función lineal o la sigmoidal. Este bloque se revisa con mayor detenimiento en secciones posteriores.

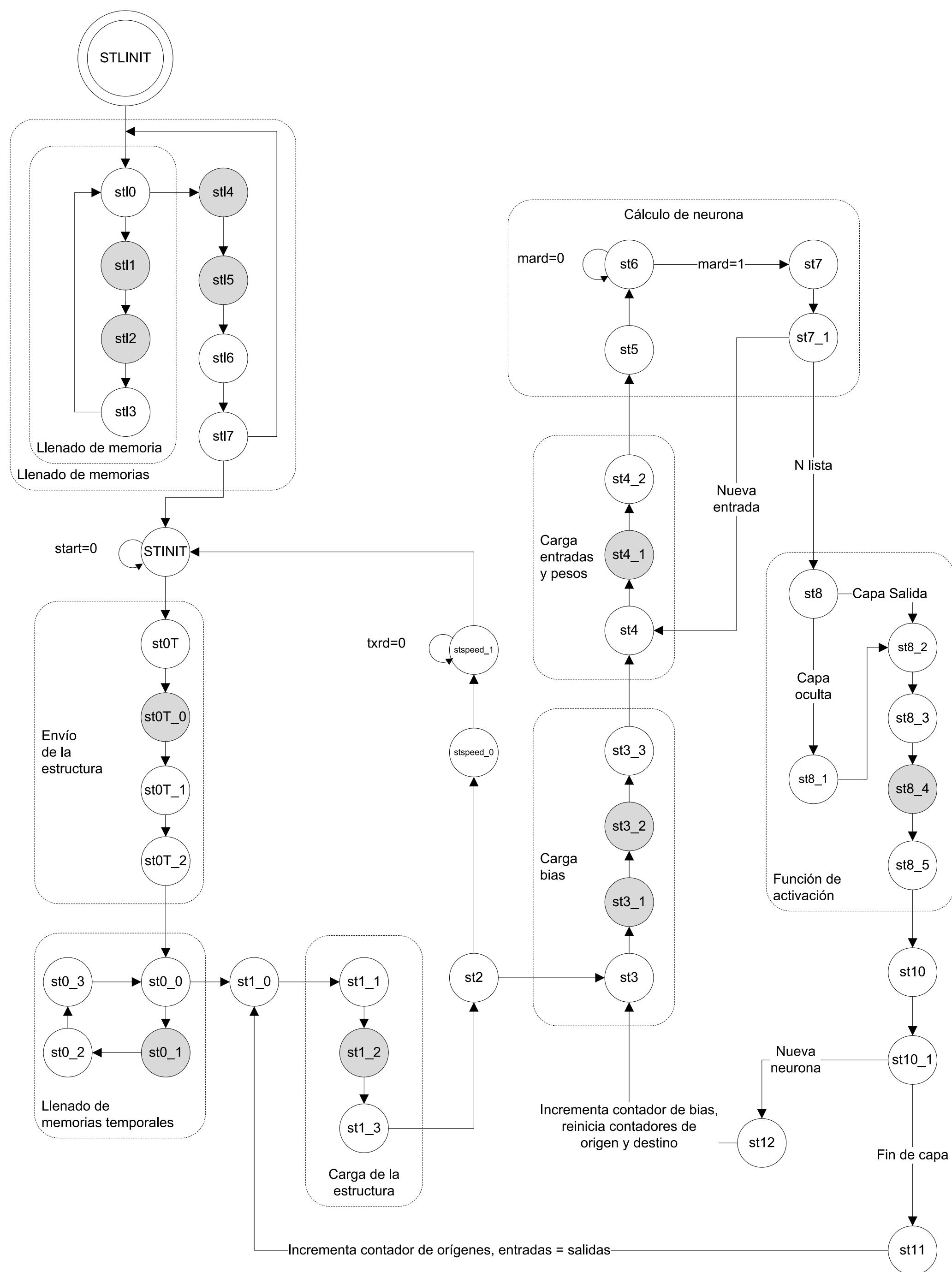


Figura 4.4: Máquina de Estados de la propagación hacia adelante de la Red Neuronal Artificial.

La figura 4.5 muestra los puertos con los que se comunican los módulos con la FSM principal, en la parte superior están las dos memorias RAM en las que se almacena la estructura para controlar el flujo de las operaciones entre las capas de la red neuronal. En el lado izquierdo de la figura, se encuentran las memorias de datos que son utilizadas para los cálculos: pesos sinápticos, bias, entradas, abscisa y pendiente; en la parte derecha aparece el módulo ***Multadder*** y la memoria de los resultados de las neuronas de la red; mientras que en la parte inferior se observa el módulo ***Sigmoid***, que se encarga de obtener el valor de la función de activación de las neuronas de la red neuronal.

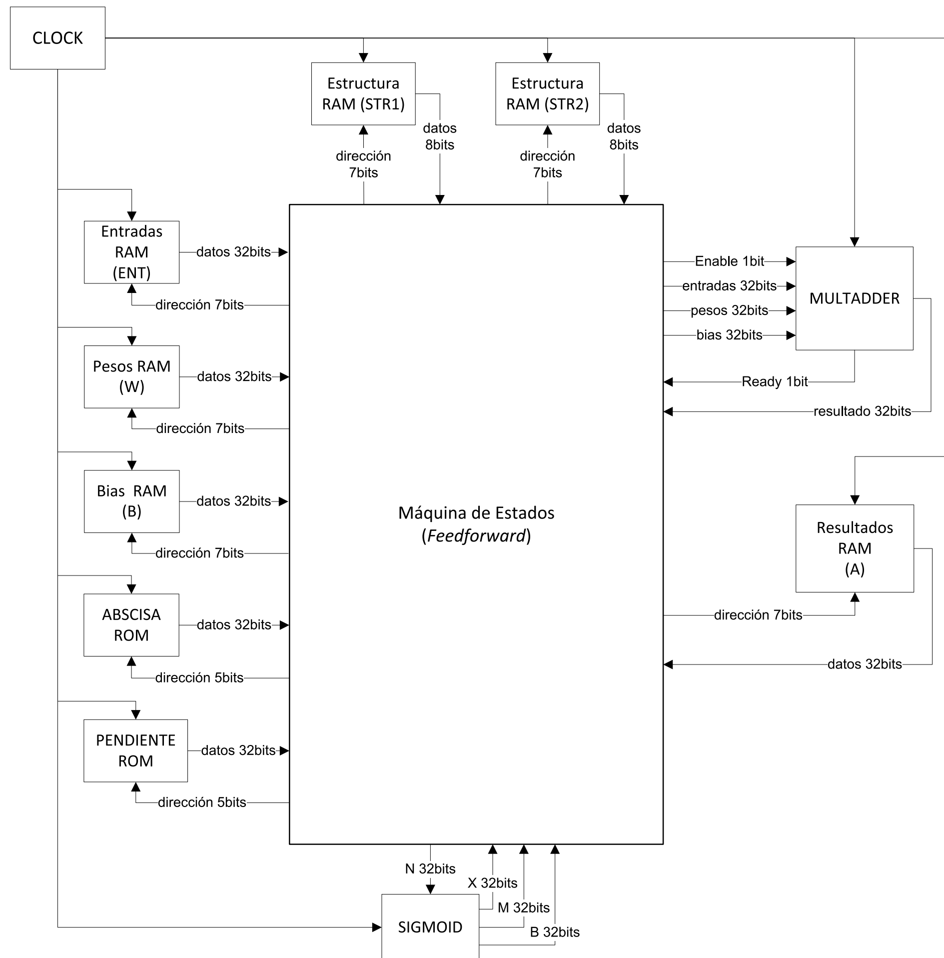


Figura 4.5: Arquitectura general de la propagación hacia adelante.

4.2.1. Módulo *Multadder*

Durante el cálculo de la propagación hacia adelante de las señales de entrada a la red neuronal destaca por su importancia el módulo ***Multadder*** (figura 4.6), que es fundamentalmente una neurona artificial en hardware con una sola entrada. Este módulo funciona de la misma manera en que la figura 3.5, acumula los productos de entradas-pesos sinápticos y bias.

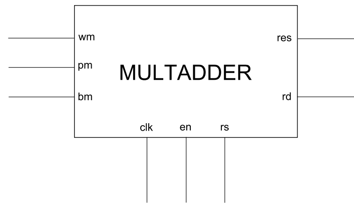


Figura 4.6: Módulo *Multadder*.

Una neurona de una sola entrada se puede convertir a una neurona multientrada conforme el Algoritmo 4.1.

Algoritmo 4.1 Conversión de una neurona simple a multientrada

Entrada: R : número de entradas,
 $w[i]$: pesos sinápticos de la neurona,
 $p[i]$: entradas a la neurona,
 b : bias de la neurona

Salida: A : salida de la neurona

1: $s_0 \leftarrow b$
2: **mientras** $i \leq R$ **hacer**
3: $s_i \leftarrow w_i * p_i + s_{i-1}$
4: **fin mientras**
5: $n \leftarrow s_n$
6: $A \leftarrow f(n)$

El módulo *Multadder* devuelve el valor n en el que se encuentran acumulados todos los productos de las entradas y el bias de la neurona.

Este módulo es capaz de calcular desde una neurona de una sola entrada con su respectivo bias, hasta una neurona con múltiples entradas. Esto es posible mediante un procesamiento en serie de las entradas: esto es, la entrada se multiplica por su peso sináptico correspondiente, entonces se suma al bias

de la neurona y este resultado será el acumulado donde los productos de las entradas subsecuentes serán almacenados.

4.2.2. Función de activación

La función de activación convierte una señal n en un resultado a de una neurona artificial. Esta implementación tiene dos diferentes funciones de activación para las neuronas de la red dependiendo de la capa en la que se encuentren:

- ▷ Función sigmoidal para las neuronas de la capa oculta.
- ▷ Función lineal para la neurona de la capa de salida.

La función sigmoidal se obtiene mediante el procedimiento conocido como *Piecewise Linearization* [43], el cual consiste en dividir el conjunto en varios segmentos del mismo tamaño, con lo que se logra una buena precisión y un menor uso de recursos que en el caso que se implementase la función tal cual está descrita por la ecuación (4.1) debido a la existencia de una función exponencial en el denominador.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4.1}$$

Esta linealización se muestra en la figura 4.7, el intervalo comprendido entre [-8,+8] fue dividido en 64 partes, asegurando un error muy bajo, ya que se observa que los puntos (linealización) se colocan sobre la curva que corresponde a la función sigmoidal. Una mayor segmentación del intervalo implica un menor error, sin embargo con los valores actuales es suficiente para asegurar una correcta aproximación.

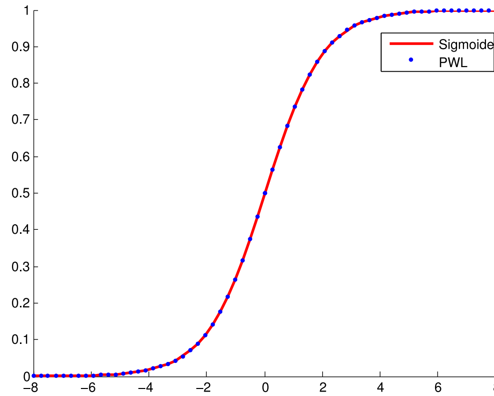


Figura 4.7: Linealización de la función sigmoidal.

Este procedimiento se realiza mediante el uso del módulo ***Sigmoid*** que se muestra en la figura 4.8.

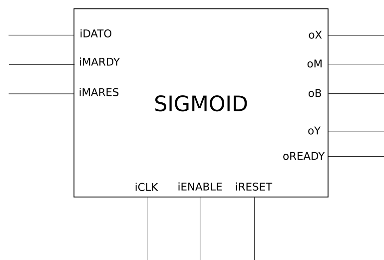


Figura 4.8: Módulo *Sigmoid*.

El algoritmo 4.2 describe el procedimiento que este módulo sigue para linealizar la función sigmoidal mediante el método *Piecewise Linear*. Los valores de la *pendiente* (oM) y la *abscisa* (oB) fueron calculados previamente mediante una aplicación en MatLab, con los que se cargan las memorias ROM respectivas. Estas memorias tienen capacidad para almacenar 32 datos, es decir, se pueden calcular 32 rectas diferentes, cada una de las cuales corresponde a dos segmentos. El hecho que una localidad corresponda a dos segmentos ya que se truncó a la mitad el conjunto de datos, debido a que la sigmoide es una función simétrica: la mitad positiva se calcula directamente, mientras que la mitad negativa se obtiene restando a 1 el valor obtenido con el dato positivo.

Para el desarrollo de este procedimiento se reutiliza el módulo ***Multadder*** descrito en la sección anterior, esto se debe a que el cálculo de la recta

mediante la ecuación (4.2) solamente requiere de tres parámetros: el valor de la pendiente, de la abscisa y el punto de interés. Además, se observa que esta ecuación es exactamente la misma que la utilizada para calcular el valor n de una neurona de una sola entrada.

$$y = mx + b \tag{4.2}$$

Algoritmo 4.2 Linealización de la función sigmoidal

Entrada: $iDATO$: n de la neurona (*punto flotante*)
Salida: oY : a de la neurona (*punto flotante*)

1: $addr \leftarrow float2int(iDATO)$
2: $oX \leftarrow iDATO$
3: $oM \leftarrow pendiente(addr)$
4: $oB \leftarrow abscisa(addr)$
5: **si** $overflow(iDATO) = 0$ **entonces**
6: $\quad oY \leftarrow oM * oX + oB$
7: $\quad$ **si** $oY < 0$ **entonces**
8: $\quad\quad oY \leftarrow 1 - oY$
9: $\quad$ **si no**
10: $\quad\quad oY \leftarrow oY$
11: $\quad$ **fin si**
12: **si no**
13: $\quad$ **si** $over_under = 0$ **entonces**
14: $\quad\quad$ **si** $iDATO < 0$ **entonces**
15: $\quad\quad\quad oY \leftarrow 0$
16: $\quad\quad$ **si no**
17: $\quad\quad\quad oY \leftarrow 1$
18: $\quad\quad$ **fin si**
19: $\quad$ **si no**
20: $\quad\quad$ **si** $oY < 0$ **entonces**
21: $\quad\quad\quad oY \leftarrow 1 - oY$
22: $\quad\quad$ **si no**
23: $\quad\quad\quad oY \leftarrow oY$
24: $\quad\quad$ **fin si**
25: $\quad$ **fin si**
26: **fin si**

Este procedimiento se realiza exclusivamente para las neuronas de la capa oculta, ya que las neuronas de la capa de salida tienen como función de activación la función lineal. Cuando se ejecutan las neuronas de la capa de salida, el módulo ***Sigmoid*** no es habilitado y el valor de la sumatoria de los productos pesos-entradas más el bias (n) se convierte directamente en la salida de la neurona (a).

4.2.3. Modos de ejecución

La propagación hacia adelante de las entradas se puede ejecutar de tres formas distintas, dependiendo de la aplicación, o bien de la cantidad de recursos del dispositivo del que se disponga.

En los modos de ejecución híbrido y paralelo destaca el cálculo de neuronas inexistentes en la red neuronal, cuyo resultado se desecha para no interferir en el resultado de la red neuronal. Estas neuronas son conocidas como neuronas *dummy*, el tiempo requerido para ejecutar estas neuronas es despreciable ya que se ejecuta de forma paralela a las neuronas útiles cuyos resultados sí serán utilizados posteriormente.

Los tres modos de ejecución ser revisan a continuación con mayor detalle.

4.2.3.1. Ejecución Serial

El modo de ejecución serial consiste en sintetizar solamente una neurona en hardware, es decir, se tiene un módulo ***Multadder*** que ejecutará todas las neuronas de la red de forma secuencial. Este modo es el que más tiempo utiliza para ejecutar la neurona, en cambio, es el que menos recursos del FPGA requiere. El algoritmo 4.3 describe este proceso.

Algoritmo 4.3 Ejecución Serial

Entrada: $str[]$: cadena de estructura de la red
Salida: A : salidas de las neuronas (*punto flotante*)

1: $i \leftarrow 1$ {*Apuntador a la estructura*}
2: $j \leftarrow 0$ {*Contador de neuronas*}
3: **mientras** $str(i) \neq FF$ **hacer**
4:     **mientras** $j < str(i)$ **hacer**
5:         $a_j^i \leftarrow f(multadder_1(W, P, b))$
6:         $j \leftarrow j + 1$
7:     **fin mientras**
8:     $i \leftarrow i + 1$
9: **fin mientras**

La figura 4.9 muestra de forma gráfica el funcionamiento del algoritmo anterior. Las neuronas oscuras representan a aquellas que son calculadas en una iteración, por lo que las blancas son las neuronas que aún no se resuelven, de esta manera se observa que se requieren de 4 ciclos para terminar de ejecutar una red neuronal [2-3-1] con una sola neurona sintetizada en el FPGA.

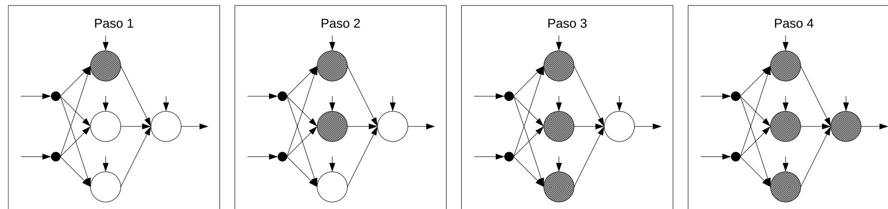


Figura 4.9: Modo de ejecución serial.

4.2.3.2. Ejecución por bloques

La ejecución por bloques se refiere al hecho que fueron sintetizadas dos o más neuronas, pero sin llegar al número máximo de neuronas en una capa. Esto le brinda a la red neuronal una gran flexibilidad para ser ajustada a la cantidad de recursos utilizados y reducir en cierto grado el tiempo total de ejecución. Aquí se presentan neuronas inútiles en el momento de terminar la ejecución de una capa, ya que en ocasiones el número de neuronas de una capa no es divisible entre el número de neuronas en hardware, el resultado de estas

células conocidas como *dummy* no será utilizado en los cálculos posteriores. El algoritmo 4.4 muestra el procedimiento.

Algoritmo 4.4 Ejecución a bloques

Entrada: $str[]$: estructura de la red,
nfn : número de neuronas hardware

Salida: A : salidas de las neuronas (*punto flotante*)

1: $i \leftarrow 1$ $\{Apuntador\ a\ la\ estructura\}$
2: $j \leftarrow 0$ $\{Contador\ de\ neuronas\}$
3: **mientras** $str(i) \neq FF$ **hacer**
4: **mientras** $j < str(i)$ **hacer**
5: $a^i_j \leftarrow f(multadder_1(W_i, P_i, b_j))$
$a^i_{j+1} \leftarrow f(multadder_2(W_{i+1}, P_{i+1}, b_{j+1}))$
$\vdots$
$a^i_{j+nfn} \leftarrow f(multadder_{nfn}(W_{i+nfn}, P_{i+nfn}, b_{j+nfn}))$
6: $j \leftarrow j + nfn$
7: **fin mientras**
8: $i \leftarrow i + nfn$
9: **fin mientras**

La ejecución por bloques de la red neuronal se muestra en la figura 4.10, las neuronas que tienen un sombreado más oscuro son las neuronas calculadas, mientras que las neuronas con el sombreado claro representan las neuronas *dummy*. El cálculo de las neuronas *dummy* se realiza ya que de esta manera se reduce la complejidad en la ejecución genérica de redes neuronales artificiales.

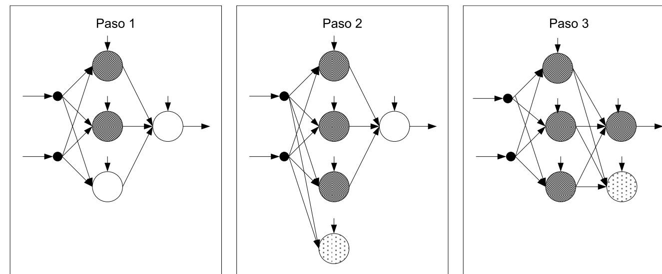


Figura 4.10: Modo de ejecución por bloques.

4.2.3.3. Ejecución en Paralelo

Este es un ambiente totalmente paralelizado, en el que se ha sintetizado una neurona en hardware por cada neurona en la capa oculta, con esto, se reduce considerablemente el tiempo de ejecución de la red neuronal en la etapa de propagación hacia adelante. Sin embargo, la reducción de tiempo requiere un incremento en el uso de recursos del FPGA. Por ejemplo, si se ejecuta una red [3-8-1]: en el FPGA serán sintetizados 8 módulos ***Multadder***.

La elección de uno de los tres métodos depende tanto de los recursos disponibles como de la velocidad que se desee o que se requiera. El hecho de paralelizar toda la etapa de la propagación hacia adelante, no significa que la red neuronal se ejecutará en el tiempo que una red [1-1-1] sería ejecutada, esto se debe a que existen operaciones cuyo flujo necesariamente tiene que ser en serie, como es el ejemplo de la escritura en memorias.

Algoritmo 4.5 Ejecución en Paralelo

Entrada: $str[]$: estructura de la red,
nfn : número de neuronas hardware
Salida: A : salidas de las neuronas (*punto flotante*)
1: $i \leftarrow 1$ $\{Apuntador\ a\ la\ estructura\}$
2: $j \leftarrow 0$ $\{Contador\ de\ neuronas\}$
3: **mientras** $str(i) \neq FF$ **hacer**
4: $a^i_j \leftarrow f(multadder_1(W_i, P_i, b_j)$
$a^i_{j+1} \leftarrow f(multadder_2(W_{i+1}, P_{i+1}, b_{j+1}))$
$\vdots$
$a^i_{j+nfn} \leftarrow f(multadder_{nfn}(W_{i+nfn}, P_{i+nfn}, b_{j+nfn}))$
5: $i \leftarrow i + 1$
6: **fin mientras**

La figura 4.11 muestra la misma red neuronal de los ejemplos anteriores resuelta mediante la implementación de 3 neuronas en el FPGA. De esta manera se logra el menor tiempo de ejecución, ya que solamente se requieren de 2 iteraciones para tener el resultado del cálculo a la salida. Lo anterior no significa que los tiempos se reduzcan proporcionalmente, ya que conforme más neuronas se sinteticen, mayores recursos de hardware para el control será utilizado y con ello, se habrán de realizar más operaciones que influyen en el tiempo total y que no necesariamente están directamente relacionados con el cálculo de la red neuronal.

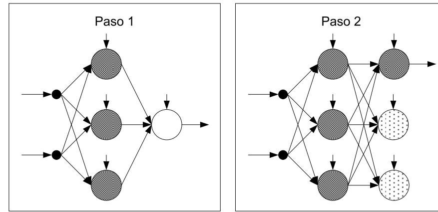


Figura 4.11: Modo de ejecución paralelo.

4.3. Algoritmo de Aprendizaje

El algoritmo de aprendizaje que entrena a la red neuronal utiliza el procedimiento de retropropagación del error descrito en el capítulo anterior, se fundamenta en la actualización de los pesos sinápticos y bias conforme al error. El aprendizaje de las redes neuronales que fue desarrollado, solamente permite el entrenamiento con redes del tipo [1-n-1], como el esquema dado en la figura 4.12.

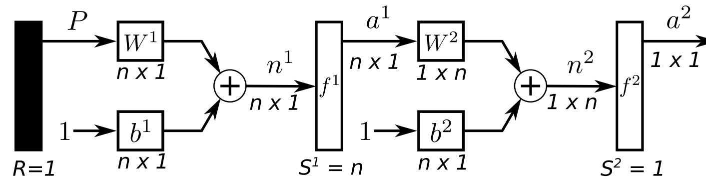


Figura 4.12: Red Neuronal [1-n-1].

El procedimiento que se realiza en el aprendizaje de la red neuronal se muestra en la figura 4.13: básicamente se trata de la actualización de forma serial de los peso sinápticos y de los bias de la red mediante el cálculo de las reglas de aprendizaje que se valen de los valores de las sensitividades de las capas (previamente calculadas) para propagar el error hacia atrás y minimizarlo a cada iteración.

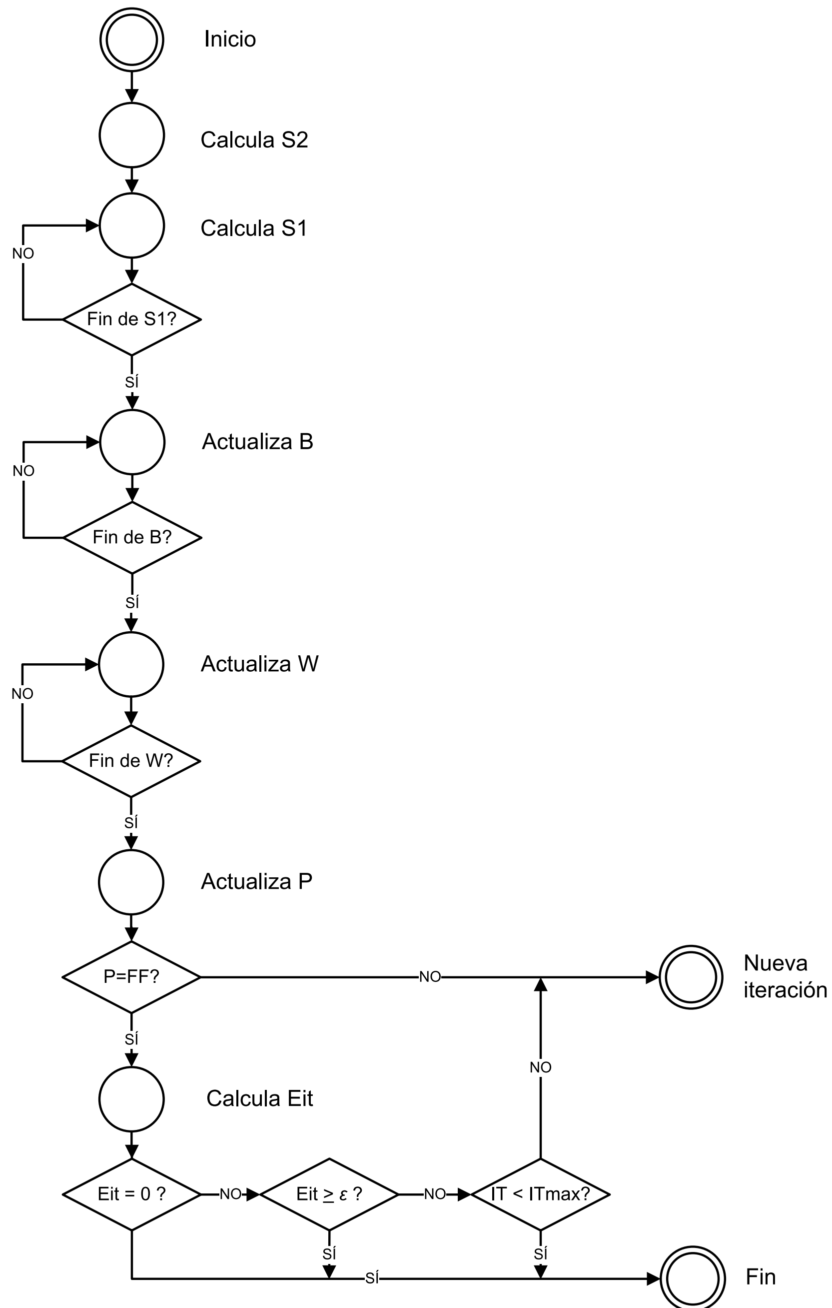


Figura 4.13: Proceso de aprendizaje.

4.3.1. Cálculo de Sensitividades

Para iniciar este proceso, es necesario calcular primero las sensitividades: la cantidad de los valores depende del número de capas que tenga la red neuronal artificial. En una red neuronal [2-4-1], por ejemplo, se tienen dos valores de sensitividades: una para la capa de salida, conocida como S^2, y otra para la única capa oculta existente que será S^1. Sin embargo, dado que la capa de salida tiene una neurona, se tiene solo un valor para S^2; pero la capa oculta tiene cuatro neuronas, lo que implica la existencia de cuatro valores (S_1^1, S_2^1, S_3^1, S_4^1), para las neuronas respectivas.

El valor de S^2 se obtiene mediante la ecuación (3.14), por lo que se implementó el módulo ***S_ 2*** (figura 4.14) que ejecuta éste cálculo. Dada la sencillez de las operaciones que se realizan y a que la función de activación de la capa de salida es la función lineal, solamente se necesitan dos valores: el valor de la salida de la red neuronal (iA) y el valor deseado de la salida de la red ($iTARGET$). Las salidas del módulo corresponden a la Sensitividad de la capa de salida ($oRESULT$) y a una señal de aviso que las operaciones se han completado satisfactoriamente y que el dato está listo para se utilizado para operaciones posteriores ($oREADY$).

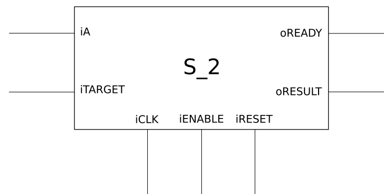


Figura 4.14: Módulo *S_ 2*.

La sensitividad de la capa oculta requiere de una cantidad mayor de procesamiento, debido a que implica el uso de la primer derivada de la función de la capa oculta (ver Apéndice A). Sin embargo, esto es transparente para el usuario, por lo que solamente se observan las entradas referentes al peso sináptico (iW), a la salida de la red neuronal (iA) y a la sensitividad de la capa de salida ($iSENS2$). Por otra parte, la selección de los datos que serán utilizados por el módulo, se lleva a cabo por la máquina de estados principal.

El procedimiento realizado por el módulo ***S_ 1*** realiza los cálculos de la ecuación (3.15), que consisten en las siguientes operaciones:

Algoritmo 4.6 Cálculo de S^1

Entrada: iW, iA, iSENS2 : *punto flotante*
Salida: oRESULT : *punto flotante*
1: $t1 \leftarrow iW * iSENS2$
2: $t1 \leftarrow t1 * iA$
3: $t2 \leftarrow 1 - iA$
4: $oRESULT \leftarrow t1 * t2$

El módulo 4.15 se ejecuta tantas veces como neuronas en la capa oculta existan, dado que es la misma cantidad de valores que se requieren para la sensitividad de la capa oculta.

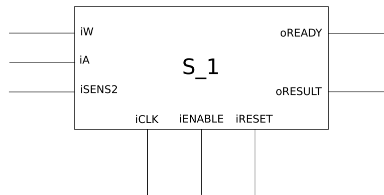


Figura 4.15: Módulo *S_1*.

4.3.2. Actualización de bias

Una vez que los valores de las sensitividades ya se tienen listos, se realiza la actualización de los parámetros de la red: pesos sinápticos y bias. Este es un proceso sencillo ya que se desarrollaron módulos que se encargan de obtener los nuevos valores, sin embargo la secuencia de actualización es algo más complicado y se controla por la FSM.

La actualización de los bias de la red se realiza de manera serial, ya que la memoria RAM en la que se almacenan los valores, solamente permite un acceso a memoria a la vez. El módulo ***updt_b*** obtiene los nuevos valores para los bias de cada capa de acuerdo a la entrada de selección (iS_SEL) existente también en este módulo. Cuando el bias a actualizar corresponde a la capa de salida, entonces se toma el valor de $iSENS2$, de otra manera, se ocupa $iSENS1$. Cuando el procesamiento de las señales de entrada ha terminado, la salida $oREADY$ indica que el nuevo valor se encuentra listo en la salida $oRESULT$.

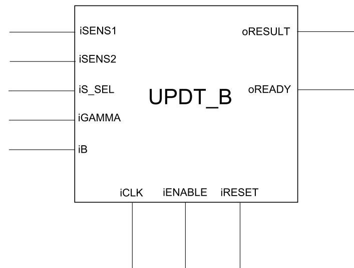


Figura 4.16: Módulo *updt_ b*.

Una entrada presente en ambos módulos para ajustar los pesos sinápticos y bias es $iGAMMA$, que es la constante para la razón de cambio (γ) de dichos parámetros de las ecuaciones descritas en el capítulo anterior. Este módulo realiza el cálculo conforme a la ecuación (3.20).

4.3.3. Actualización de pesos sinápticos

El procedimiento para la actualización de los pesos sinápticos de la red neuronal, es similar al realizado para los bias, es decir, existe un módulo que se encarga de realizar las operaciones de actualización para los pesos sinápticos que procesa los datos de forma serial.

El módulo ***updt_ w*** obtiene los nuevos valores, este componente es capaz de actualizar tanto valores de la capa de entrada como de la capa de salida, esto se hace mediante la selección de la capa con la entrada iS_SEL:

- ▷ El peso corresponde a la capa de salida, toma el valor de salida de la neurona de la capa oculta(iA) y de la sensitividad de la capa de salida ($iSENS2$).
- ▷ El peso corresponde a la capa oculta, se utilizan los valores de entrada ($iENT$) y la sensitividad de la capa de entrada ($iSENS1$).

El módulo ***updt_ w*** se muestra en la figura 4.17 con las entradas previamente expuestas.

El módulo realiza el cálculo conforme a la ecuación (3.19).

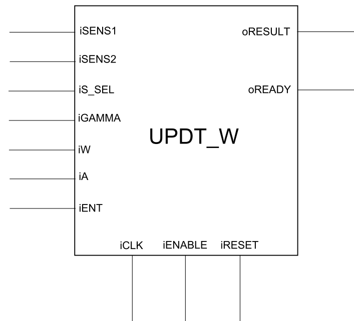


Figura 4.17: Módulo *updt_w*.

4.3.4. Actualización de entradas

Posterior a la actualización de los parámetros de la red neuronal, es necesario actualizar el valor de la entrada al siguiente valor en el conjunto de entrenamiento, con la finalidad de recorrer todos los patrones de entrenamiento, y que una vez realizado el procesamiento con todos los valores, se ejecute una nueva iteración de ser necesaria. El procedimiento que se encarga de este proceso se describe en el Algoritmo 4.7.

Algoritmo 4.7 Actualización de las entradas

Entrada: $training_set[]$: conjunto de entrenamiento,
Fin_{It} : bandera de fin de iteración
Salida: p : entrada siguiente
1: $p \leftarrow training_set(i)$
2: **si** $p = FF$ **entonces**
3: $Fin_{IT} \leftarrow 1$
4: $i \leftarrow 0$
5: $p \leftarrow training_set(0)$
6: **si no**
7: $Fin_{IT} \leftarrow 0$
8: $i \leftarrow i + 1$
9: **fin si**

4.3.5. Cálculo del Error de Iteración (E_{It})

El Error de Iteración (E_{It}) se obtiene con la ecuación (4.3) mediante el uso del módulo ***Multadder***, el algoritmo 4.8 explica la manera en la que realiza esta operación.

$$E_{It} = \frac{\sum_{i=1}^{n_{pat}} (a_{target(i)} - a_{(i)})^2}{n_{pat}} \tag{4.3}$$

El error obtenido con la ejecución de cada patrón (E_{pat}), se acumula hasta que se haya recorrido todo el conjunto de entrenamiento, al cumplirse esta condición, se divide el resultado entre el número de patrones del conjunto.

Algoritmo 4.8 Cálculo de E_{It}

Entrada: E_{pat} : error de patrón (*punto flotante*),
n_{pat} : número de patrones (*integer*)
Salida: E_{It} : error de iteración (*punto flotante*)
1: $t1 \leftarrow multadder(E_{pat}, E_{pat}, t1)$
2: **si** $Fin_{It} = 1$ **entonces**
3: $E_{IT} \leftarrow \frac{t1}{float2int(n_{pat})}$
4: **fin si**

Finalmente, se revisa si el E_{It} cumple con alguno de los criterios de paro, de otra manera, se ejecutará otra iteración hasta que se cumpla uno. Los criterios de finalización que fueron implementados en el prototipo fueron los siguientes:

- ▷ $E_{it} = 0$
- ▷ $E_{it} \leq \varepsilon$
- ▷ Alcanzar un It_{max}

El primer criterio de finalización establece que cuando el error sea nulo, la red ha sido entrenada satisfactoriamente. Para el segundo criterio, se elige un valor ε, que es el valor mínimo de error con el que debe operar el entrenamiento de la red neuronal, en el momento que se iguale o bien, se baje de este valor, la red ha terminado su ejecución. El tercer criterio requiere de

un número máximo de iteraciones a realizar por la red, en el caso que no se finalice la ejecución por llegar a un valor del error menor a ε, la red ya no se entrenará más cuando se alcancen It_{max} iteraciones, evitando de esta manera que la red caiga en mínimos locales.

4.4. Comunicación PC-FPGA

La aplicación desarrollada en MatLab es una interfaz entre la computadora y el FPGA. La comunicación se establece mediante el comando *serial* que utiliza los siguientes parámetros para comunicarse correctamente con el FPGA:

- ▷ Velocidad: 9600 baudios.
- ▷ Bits de datos: 8.
- ▷ Puerto: COM5.

Posterior al establecimiento de la conexión entre ambos dispositivos, se realiza la apertuda del puerto con la instrucción *fopen* con lo que se le asigna a una variable la información obtenida del puerto serie: para escribir en el puerto se utiliza *fwrite* y para leer datos recibidos en él, se utiliza *fread*. Cuando se ha terminado de escribir y leer el puerto, el comando *fclose* cierra el puerto, por lo que ya no se puede escribir ni leer más datos desde el puerto serie.

No obstante que se programó el puerto serial de la computadora para su comunicación con el FPGA, fue necesario el uso de un convertidor Serial-USB (figura 4.18) ya que la computadora con la que se cuenta no tiene puerto serial, pero sí puertos USB cuya comunicación resulta transparente con la utilización de dicho convertidor.

Figura 4.18: Convertidor Serial - USB.

4.5. Resumen

Este capítulo explica de manera detallada la arquitectura propia del trabajo desarrollado. El proyecto fue desglosado en dos etapas:

- ▷ Propagación hacia adelante de las señales
- ▷ Algoritmo de aprendizaje

En la primer etapa, se ejecuta la red utilizando el valor de la entrada, los pesos sinápticos y los bias que en ese instante tiene la red, dicha ejecución se puede realizar de tres modos:

- ▷ Serial
- ▷ Por bloques
- ▷ Paralela

La segunda etapa ajusta dichos parámetros con la finalidad de reducir el error y tener un mínimo error a la salida. Por último, al obtener el error de iteración se puede tomar la decisión de terminar o continuar con la ejecución de la red para llevar el error a cero, que se rebaje el error mínimo establecido (ε), o en su caso, terminar la ejecución por haber alcanzado un número máximo de iteraciones (It_{max}).

Capítulo 5

Presentación de Resultados

Los resultados que se presentan a lo largo de este capítulo se derivan de la ejecuciíon de las dos etapas de la red neuronal artificial implementada en el FPGA:

- ▷ Propagación hacia adelante de las señales de entrada
- ▷ Aprendizaje de la red neuronal

A continuación se explican de forma detallada los experimentos realizados.

5.1. Experimento 1: Propagación hacia adelante de las señales de entrada

La red neuronal se puede ejecutar de tres maneras distintas, dependiendo de la disponibilidad de recursos con los que el FPGA cuente:

- ▷ Serial: se puede ejecutar en un FPGA con pocas unidades de memoria y de LEs disponibles; esto se debe a que únicamente se requiere una unidad de cálculo para ejecutar todo el cómputo de la red neuronal.
- ▷ Por bloques: aquí se ejecutan varias neuronas en paralelo para calcular una parte de la capa de la red, para ejecutar completamente la capa, se requiere otra iteración para calcular el resto de las neuronas. Por ello, se dice que tiene una ejecución en paralelo y a su vez, también se ejecuta en serie.

- ▷ Paralelo: este es un entorno en el que la ejecución de la red neuronal requiere que se realice en el menor tiempo posible, para ello se utiliza una mayor cantidad de recursos del FPGA.

El dispositivo que fue utilizado para la implementación de este proyecto es un FPGA Altera Cyclone II DE2-70, la cual cuenta con 68,416 LEs, 1,152,000 bits de memoria en total, así como 300 multiplicadores de 9 bits, que operan a un reloj interno de 50 MHz. La tabla 5.1 muestra el consumo de recursos con la síntesis desde 1 hasta 10 neuronas el dispositivo.

Neuronas Hardware	**Recursos**		
	LEs	**Memoria**	**Multiplicadores**
1	8,781	9,288	7
2	11,186	15,528	14
3	17,910	21,756	21
4	16,012	27,984	28
5	25,092	34,212	35
6	21,032	40,440	42
7	32,014	46,668	49
8	23,534	52,896	56
9	29,935	59,124	63
10	36,858	65,352	70

Tabla 5.1: Uso de recursos del FPGA.

De lo anterior destaca que aún y cuando se implementen 10 neuronas en el FPGA, el consumo de recursos es moderado, se utiliza ligeramente más del 53 % de los elementos lógicos disponibles y un 5 % del uso de la memoria total. El ahorro en los recursos del dispositivo se debe a la implementación de algoritmos de linealización para obtener el resultado de la función de activación de las neuronas.

La tabla 5.2 muestra una comparación entre los diferentes modos de ejecución antes expuestos con distintas configuraciones de redes neuronales. La red neuronal se sintetizó con un máximo de 5 neuronas en hardware ya que con esto se consigue un buen panorama de los tiempos y recursos requeridos para la configuración de cualquier RNA.

Red neuronal	Neuronas Hardware				
	1	**2**	**3**	**4**	**5**
[1-1-1]	2.54	2.54	2.54	2.48	2.54
[1-2-1]	4.44	3.10	3.10	3.10	3.10
[1-3-1]	6.34	5.00	3.66	3.66	3.66
[1-4-1]	8.24	5.56	5.56	4.22	4.22
[1-5-1]	10.14	7.46	6.12	6.12	4.78
[2-1-1]	3.18	3.18	3.18	3.18	3.18
[2-2-1]	5.64	3.74	3.74	3.74	3.74
[2-3-1]	8.10	6.20	4.30	4.30	4.30
[2-4-1]	10.56	6.76	6.76	4.86	4.86
[2-5-1]	13.02	6.76	7.32	7.32	5.42
[3-1-1]	3.60	3.82	3.82	3.82	3.82
[3-2-1]	6.84	4.38	4.38	4.38	4.38
[3-3-1]	9.86	7.40	4.94	4.94	4.94
[3-4-1]	12.88	7.96	7.96	5.50	5.50
[3-5-1]	15.90	10.98	8.52	8.52	6.06
[4-1-1]	4.46	4.46	4.46	4.46	4.46
[4-2-1]	8.04	5.02	5.02	5.02	5.02
[4-3-1]	11.62	8.60	5.58	5.58	5.58
[4-4-1]	15.20	9.16	9.16	6.14	6.14
[4-5-1]	18.78	12.74	9.72	9.72	6.70
[5-1-1]	5.10	5.10	5.10	5.10	5.10
[5-2-1]	9.24	5.66	5.66	5.66	5.66
[5-3-1]	13.38	9.80	6.22	6.22	6.16
[5-4-1]	17.52	10.36	10.36	6.78	6.78
[5-5-1]	21.66	14.50	10.92	10.92	7.34

Tabla 5.2: Tiempos de ejecución (ms) con neuronas hardware.

En la primer columna se encuentran los tiempos más altos, lo que corresponde a una ejecución puramente serial, mientras que la quinta representa a una implementación totalmente paralela ya que el número máximo de neuronas en hardware utilizadas para los experimentos fue justamente de 5 neuronas. Estos resultados se pueden apreciar de manera gráfica en la figura 5.1, en la que las redes con una neurona en su capa oculta y una salida tienen tiempos de ejecución similares o incluso iguales. Este comportamiento se debe al *"efecto dummy"* en el que se ejecutan en paralelo todas las neuronas aún y cuando su resultado no sea utilizado para cálculos posteriores, esto evita agregar complejidad que pudiese ser innecesaria al diseño.

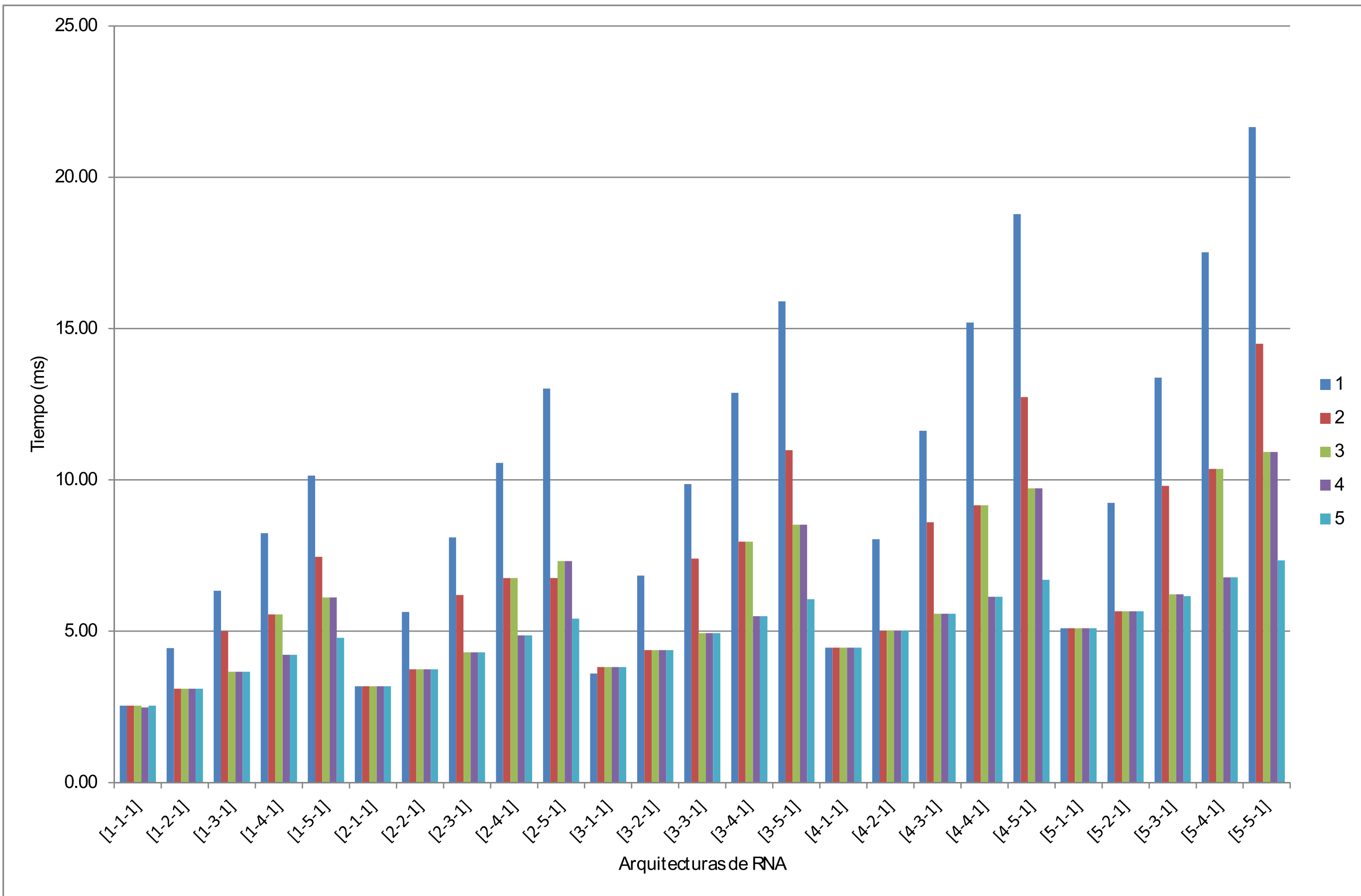


Figura 5.1: Tiempos de ejecución (ms) con neuronas hardware.

Un detalle muy interesante que se puede observar es que existen resultados que permanecen constantes en varias columnas. En una red [1-1-1] los tiempos son muy cercanos, inclusive algunos iguales, esto se debe a que a pesar de que se implementen más neuronas en hardware, el tiempo de ejecución de una neurona es el mismo. Por otra parte la quinta columna resulta ser la más rápida, debido a que muestra los resultados de la síntesis de 5 neuronas en el FPGA, esto se debe a que se tiene una implementación totalmente en paralelo.
En la tabla 5.3 se muestran las frecuencias máximas de operación para distinto número de neuronas sintetizadas en el FPGA.
La tabla 5.4 muestra con una mayor claridad la disminución de tiempos de

Neuronas hardware	**Reloj**	
	iCLK50	**clk96**
1	111.86	302.11
2	112.45	294.9
3	107.32	285.47
4	108.12	290.53
5	108.35	296.91
6	105.2	284.33
7	106.43	288.02
8	106.3	291.21
9	107.02	292.57
10	105.73	275.48

Tabla 5.3: Frecuencias máximas de operación (MHz).

ejecución mediante el uso de varias neuronas sintetizadas en el FPGA. Es importante destacar el hecho que una red [1-6-1] resuelta mediante 2 neuronas en el FPGA utiliza prácticamente el mismo tiempo que una red neuronal [6-6-6] con 6 neuronas en hardware. Otro punto a tomar en cuenta es que una implementación con 6 neuronas en el dispositivo reduce considerablemente los tiempos en los que la red neuronal se calcula, en el caso de la red neuronal [6-6-6] se ven disminuidos a un 18 % los tiempos de ejecución con respecto a una implementación totalmente serial.

Red neuronal	Neuronas Hardware			
	1	2	3	6
[1-6-1]	12.04	8.02	6.68	5.34
[6-6-6]	46.94	23.9	16.22	8.54

Tabla 5.4: Tiempos de ejecución (ms).

La tabla 5.5 presenta una comparación entre los resultados obtenidos mediante la implementación de la red neuronal artificial en el FPGA y una aplicación desarrollada en MatLab.

Red neuronal	Resultados obtenidos	
	FPGA	MatLab
[1-1-1]	1.8808	1.8808
[1-2-1]	2.7616	2.7616
[1-3-1]	3.6424	3.6424
[1-4-1]	4.5232	4.5232
[1-5-1]	5.4040	5.4040
[2-1-1]	1.9526	1.9526
[2-2-1]	2.9051	2.9051
[2-3-1]	3.8577	3.8577
[2-4-1]	4.8103	4.8103
[2-5-1]	5.7629	5.7629
[3-1-1]	1.9820	1.9820
[3-2-1]	2.9640	2.9640
[3-3-1]	3.9460	3.9460
[3-4-1]	4.9281	4.9281
[3-5-1]	5.9101	5.9101
[4-1-1]	1.9933	1.9933
[4-2-1]	2.9866	2.9866
[4-3-1]	3.9799	3.9799
[4-4-1]	4.9732	4.9732
[4-5-1]	5.9665	5.9665
[5-1-1]	1.9975	1.9975
[5-2-1]	2.9951	2.9951
[5-3-1]	3.9926	3.9926
[5-4-1]	4.9901	4.9901
[5-5-1]	5.9876	5.9876

Tabla 5.5: Resultados obtenidos con el FPGA y con MatLab.

Los datos resultan ser los mismos con una precisión de 4 decimales, lo que significa que la red neuronal además de ejecutarse a una alta velocidad, provee resultados certeros con un consumo de recursos bastante aceptable.

5.1.1. Caso de estudio

Los automóviles a finales del siglo XIX eran principalmente carruajes motorizados que eran extremadamente lentos que no requerían un sistema de suspensión. Cuando los motores de combustión adquirierion mayor potencia, la mejora en el comfort y en el manejo tomaron mayor importancia. Las primeras suspensiones que qparecieron en 1898 eran simples ballestas como las de los carros jalados por caballos.

Alrededor de los años 1898 y 1899 se incrementó la necesidad de absorber el movimiento oscilatorio generado por las ballestas, esto originó el surgimiento de los primeros amortiguadores que consistian en dos brazos unidos por un disco de fricción. Debido a su baja durabilidad y eficiencia, la investigación de nuevos modelos de suspensión cobró gran importancia,

Actualmente, la industria automotriz utiliza amortiguadores hidráulicos, de los cuales hay varios tipos pero todos tienen características en común: el muelle y los parámetros de amortiguamiento varían de una forma no lineal y no hay manera de cambiar sus valores.

Las suspensiones que utilizan amortiguadores con parámetros fijos se llaman *"suspensiones pasivas"*. Sin embargo, estos sistemas resultan ineficientes para brinder seguridad a altas velocidades. Esto motiva el desarrollo de suspensiones provistas de algunos mecanismos que adaptan sus parámetros para conseguir un mejor desempeño en cualquier circunstancia. Estas fueron llamadas *"suspensiones activas"*.

Este problema tiene dos aspectos principales: primero, la dificultad de proveer las leyes de control para ese tipo de sistemas altamente no lineales, esto se ha resuelto de muchas formas, desde un modelo de control clásico propuesto por [44], donde el modelo de control clásico es asistido por sistemas inteligentes: algoritmos genéticos [45]. Finalmente, en control inteligente, como contro difuso que ha reportado buenos resultados [46] y [47]. Otra solución es el uso de controladores híbridos; donde se combinan las estrategias antes mencionadas. En [48] se propone un controlador PID ajustado por un algoritmo genético asistido por el control difuso.

En este trabajo se utiliza la propagación hacia adelante de las señales de entrada de una red neuronal artificial para medir los ángulos más importantes

de un vehículo *"off-road"* tipo Baja SAE [49]. Los ángulos que son medidos son el *pitch* y el *roll*. La figura 5.2 ilustra estos ángulos.

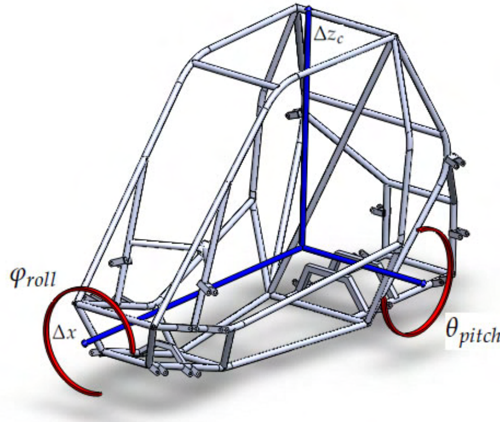


Figura 5.2: Ángulos Pitch y Roll del vehículo Baja SAE.

Los sensores virtuales utilizados en este trabajo fueron diseñados para ser usados en suspensiones activas. Para logar ese objetivo, se implementaron dos sensores virtuales: uno para el ángulo Pitch y otro para el ángulo Roll. Los sensores virtuales generan los ángulos a partir de las señales de distancia de los cuatro sensores que miden la compresión de cada amortiguador del vehículo. Un ejemplo de la medición de distancia en los amortiguadores se muestra en la figura 5.3.

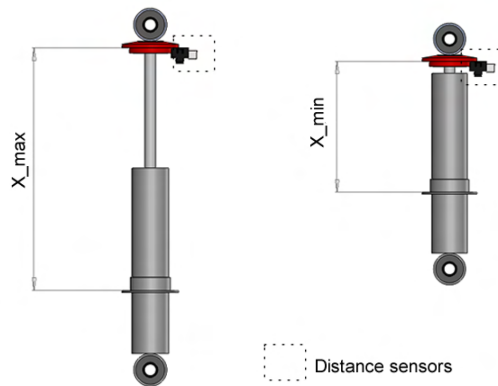


Figura 5.3: Sensor de distancia del amortiguador.

En la figura 5.4 se muestra el sistema propuesto. La figura tiene cuatro acrónimos para los sensores de distancia:

1. Amortiguador trasero derecho (DS RRD).
2. Amortiguador trasero izquierdo (DS RLD).
3. Amortiguador delantero derecho (DS FRD).
4. Amortiguador delantero izquierdo (DS FLD).

Los sensores de distancia fueron seleccionados como entradas al sensor virtual debido a que la distancias de compresión de los amortiguadores son las variables más importantes que indican al modelo los ángulos de inclinación, un trabajo similar que utiliza sensores de distancia se presenta en [50]. Cada sensor de distancia se conecta a los sensores virtuales de los ángulos Pitch y Roll embebidos con el controlador en el FPGA.

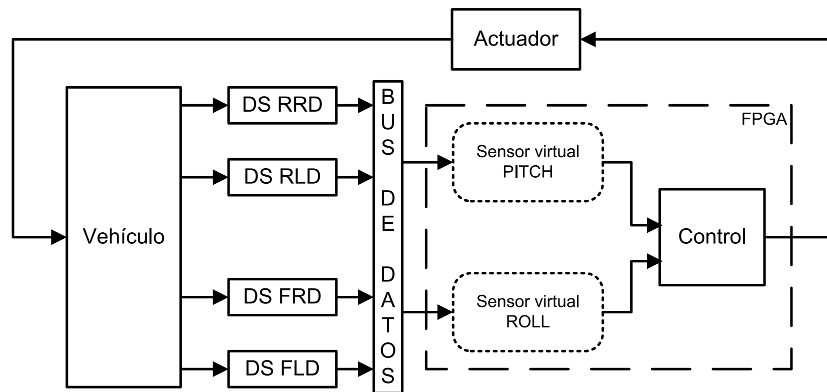


Figura 5.4: Sistema de suspensión activa.

Cada sensor virtual está hecho con un perceptrón multicapa con una estructura [4-22-1] como se muestra en la figura 5.5.

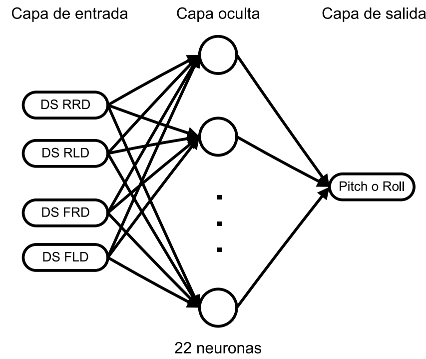


Figura 5.5: Diagrama del sensor virtual.

Los datos utilizados para entrenar el perceptrón multicapa son las mediciones de los sensores de distancia cuando el vehículo se mueven en una pista de pruebas. La figura 5.6 muestra el recorrido de pruebas para obtener las mediciones del sistema, este trayecto está formado por un arreglo de topes.

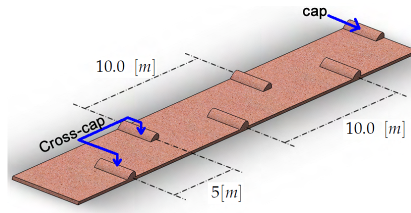


Figura 5.6: Pista de pruebas.

La arquitectura de la red neuronal implementada para el diseño de estos sensores virtuales se basa en el modelo presentado anteriormente en estre trabajo en la figura 4.4.

El perceptrón multicapa fue adaptado utilizando un conjunto de 920 muestras de las cuales se utilizó un 70 % para entrenamiento, 13.4 % para validación y un 16.6 % para pruebas. Con la finalidad de analizar el comportamiento decada sensor, se realizaron dos experimentos: la primera consistió en una simulación en la computadora mediante MatLab, la segunda prueba fue hecha en el FPGA usando la técnica *hardware-in-the-loop* [51].

Los resultados obtenidos en los experimentos antes mencionados se observan en las figuras 5.7 y 5.8.

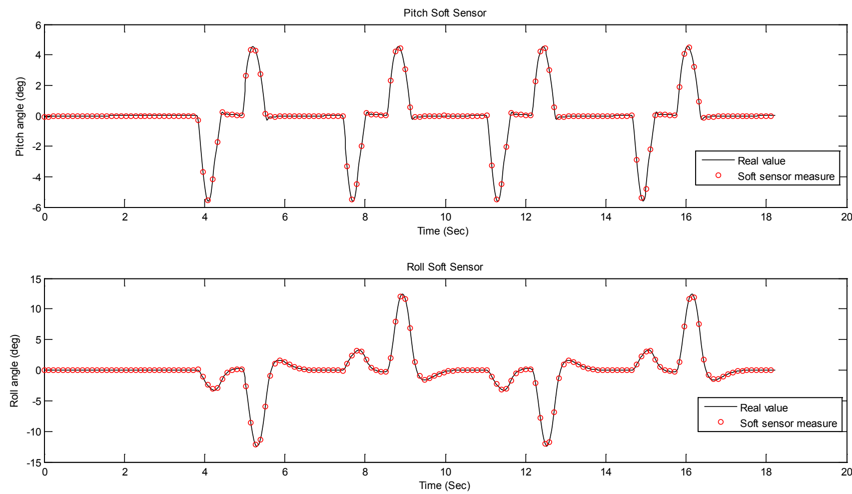


Figura 5.7: Mediciones del sensor virtual sobre el FPGA.

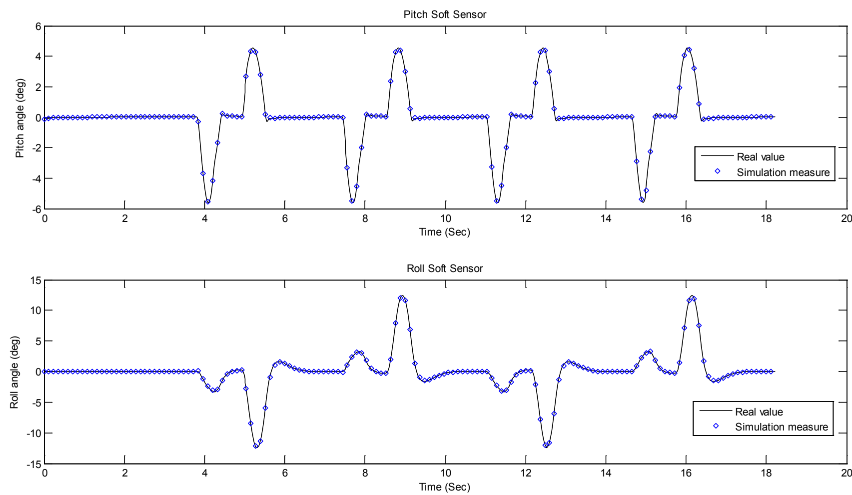


Figura 5.8: Mediciones del sensor virtual en MatLab.

El esquema *Hardware-in-the-loop* (HIL) es una herramienta indispensable para el desarrollo de controladores para los vehículos modernos porque brindan un ambiente extremadamente seguro para probar el control de la dinámica del vehículo. La técnica HIL juega un rol muy importante en el desarrollo de automóviles más seguros, ya que han sido implementados para tareas críticas como sistemas de frenado antibloqueo (ABS), sistema de frenado asistido (BAS) y sistemas electrónicos de de estabilidad (ESP).

	MatLab	FPGA
Sensor Pitch	0.090	0.0139
Sensor Roll	0.0412	0.0467

Tabla 5.6: Error promedio de los sensores virtuales.

En la tabla 5.1.1 se muestran los errores obtenidos por ambos experimentos. Las mediciones de los sensores virtuales se obtuvieron en grados, cada grado corresponde a una unidad en la escala del sensor virtual. Tomando esto en consideración, el error promedio para el sensor de Pitch es de 0,0139 grados, mientras que para el sensor de Roll es de 0,046. Estos valores de error resultan aceptables para esta aplicación.

5.2. Experimento 2: Aprendizaje de la red neuronal

Una red neuronal artificial con aprendizaje se convierte en un aproximador universal, como se mencionó en capítulos anteriores. Esta es la razón que los primeros experimentos que se llevaron a cabo consisten en la aproximación de señales no lineales.

En primer lugar, se utilizó la ecuación 5.1, que representa una señal senoidal. Para ello se introdujo a la red neuronal un conjunto de 61 muestras comprendidas en el intervalo [-2,+2] con una diferencia de $\frac{1}{15}$.

$$a_{target} = 1 + Sin(\frac{\pi}{2}x) \tag{5.1}$$

Los resultados que se obtuvieron fueron comparados con una aplicación desarrollada en MatLab, la figura 5.9 muestra que los datos obtenidos por

el FPGA se encuentran muy cercanos a los obtenidos por la aplicación en software. El aprendizaje requirió de 700 iteraciones para que la red terminase su procedimento de entrenamiento con $E_{It} < \varepsilon = 0{,}001$. La red neuronal artificial utilizada para el experimento tiene una arquitectura [1-3-1]. Los parámetros de la red neuronal (pesos sinápticos y bias son aleatorios dentro del intervalo [-1,+1]). El tiempo total de procesamiento fue de 2,448 segundos en los que se realizaron 1145 iteraciones en total para entrenar la red neuronal en el FPGA, mientras que la aplicación hecha en MatLab requirió de 13,592 segundos para realizar el mismo proceso.

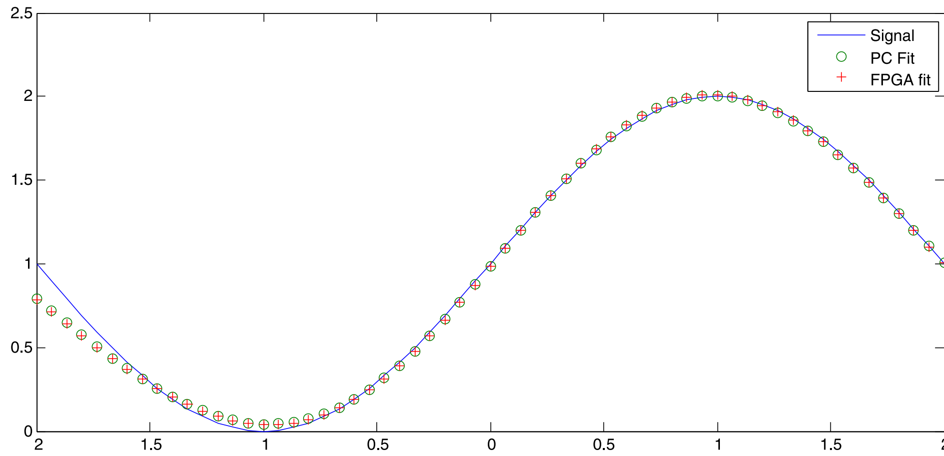


Figura 5.9: Comparativo del entrenamiento de MatLab y del FPGA.

Del procedimiento de aprendizaje anterior, se obtienen los valores de error de aprendizaje (E_{It}) que se muestran en la figura 5.10 a manera de comparación entre ambas implementaciones.

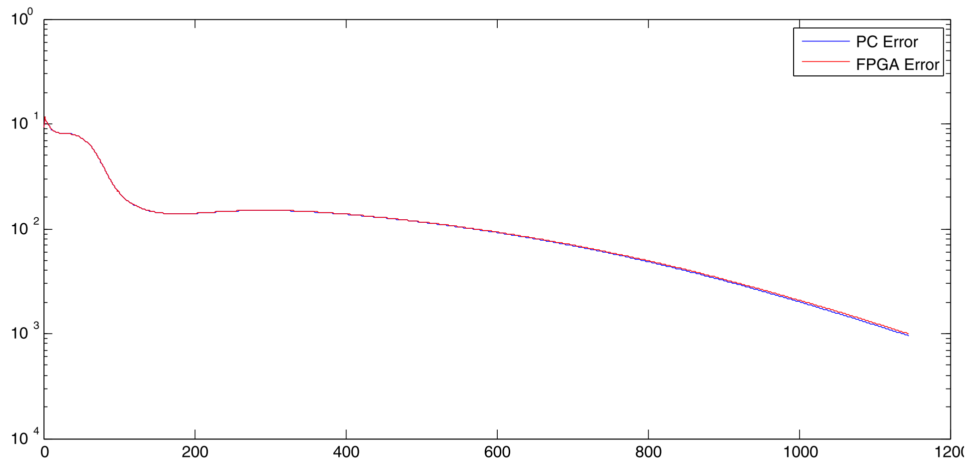


Figura 5.10: Comparativo del error obtenido por MatLab y por el FPGA.

La figura previa es una clara muestra de los resultados obtenidos por la ejecución del proyecto en su totalidad: tiene un alto grado precisión manteniendo una alta velocidad en el procesamiento de ambas etapas de la red neuronal, propagación hacia adelante de las señales de entrada y propagación hacia atrás del error.

5.3. Resumen

El uso de los recursos del FPGA es una medida que le brinda a esta implementación la posibilidad de ser integrado en algún proyecto a futuro, ya que solamente en el caso de una síntesis totalmente paralela se llega a utilizar más del 50 % de los elementos lógicos disponibles, mientras que el uso de memoria se mantiene en un 5 %. Los resultados obtenidos mediante la medición de tiempos en el FPGA arroja que la propagación hacia adelante se mantiene en el orden de los milisegundos, lo que le brinda a esta implementación una posible utilización en ambientes cuya velocidad cobre gran importancia. La propagación hacia atrás del error de la red neuronal resulta tener una gran velocidad, por lo que mediante un proceso de optimización para reducir aún más el tiempo de aprendizaje, este desarrollo sería muy útil para tareas de tiempo real, en las que el sistema deba reaccionar a posibles variaciones en su entorno manteniendo un buen desempeño.

Capítulo 6

Conclusiones

6.1. Conclusiones

Como parte del presente trabajo, se realizó la implementación de dos sensores virtuales para la estimación de los ángulos de Pitch y Roll de un vehículo Baja-SAE. Los sensores virtuales se diseñaron mediante redes neuronales artificiales multicapa (*perceptrón multicapa*).

En un primer experimento, la propagación hacia adelante de las señales de entrada a la red neuronal provenientes de los sensores ubicados en los amortiguadores se utilizó como regresor para estimar los ángulos antes mencionados. Las señales de entrada corresponden a la medición de la compresión de cada uno de los amortiguadores con los que cuenta el vehículo. Los valores de los pesos sinápticos y los bias fueron calculados mediante MatLab. En una comparativa para determinar la exactitud de la implementación en el FPGA y su contraparte en software, los datos obtenidos en ambos ejercicios resultaron ser muy cercanos.

El resto del trabajo desarrollado comprende el proceso de aprendizaje de la red neuronal artificial. Lo que requirió el desarrollo de componentes en lenguaje VHDL para la actualización de los pesos sinápticos y de los bias. Los componentes se realizaron de manera modular con la idea de mantener el consumo de recursos al mínimo, sin sacrificar tiempo de ejecución. Los componentes realizan los cálculos indicados en las reglas de aprendizaje

obtenidas de manera analítica mediante los fundamentos teóricos de la propagación hacia atrás del error a la salida de la red neuronal. Para ello, también fue necesario implementar componentes que calculan las sensitividades de las capas de la red neuronal, así como los respectivos errores por cada entrada a la red y el error de aprendizaje obtenido una vez que se ha terminado de ejecutar una iteración.

La red neuronal resultó ser capaz de aprender conforme a los valores que recibe como entradas y al conjunto de datos objetivo que deberá de seguir la red. Para validar el desempeño del sistema se realizaron experimentos que tienen como meta el aproximar señales no lineales. Como resultado de estos experimentos se comprobó que el Perceptrón multicapa con una capa oculta y una capa de salida cumple con el carácter de *Aproximador universal*, ya que tiene la capacidad de obtener resultados muy cercanos al conjunto de entrenamiento. Por otra parte, se realizó una aplicación en MatLab para comprobar el correcto funcionamiento del aprendizaje de la red neuronal en el FPGA; los resultados no pudieron ser mejores, en todos los casos se logró total similitud entre ambas implementaciones con cuatro cifras decimales.

6.2. Trabajo a Futuro

Algunos aspectos en los que es posible realizar más trabajo tomando como base este trabajo son los siguientes:

- ▷ Implementar el criterio de finalización *Early stopping* para prevenir el sobreentrenamiento de la red neuronal.
- ▷ Modificar el esquema actual de entrenamiento en línea para posibilitar el aprendizaje en redes neuronales genéricas, lo que implica las siguientes modificaciones:
  - Entrenar redes neuronales con más de una entrada y/o más de una salida.
  - Entrenar redes neuronales con más de una capa oculta.

Referencias

[1] Ming-Da Ma, Jing-Wei Ko, San-Jang Wang, Ming-Feng Wu, Shi-Shang Jang, Shien-Shu Shieh, and David Shan-Hill Wong. Development of adaptive soft sensor based on statistical identification of key variables. *Control Engineering Practice*, 17(9):1026–1034, 2009.

[2] Petr Kadlec, Bogdan Gabrys, and Sibylle Strandt. Data-driven Soft Sensors in the process industry. *Computers and Chemical Engineering*, 33(4):795–814, April 2009.

[3] Surinder Jassar, Zaiyi Liao, and Lian Zhao. Data Quality in Hybrid Neuro-Fuzzy based Soft-Sensor Models: An Experimental Study. *International Journal of Computer Science (AENG)*, 37(1), 2010.

[4] Yao Wu and Xionglin Luo. A Design of Soft Sensor based on Data Fusion. In *International Conference on Information Engineering and Computer Science 2009. (ICIECS 2009)*, pages 1–4, 2009.

[5] Yao Wu and Xionglin Luo. A novel calibration approach of soft sensor based on multirate data fusion technology. *Journal of Process Control*, 20:1252–1260, December 2010.

[6] David Cecil and Magdalena Kozlowska. Software sensors are a real alternative to true sensors. *Environmental Modelling and Software*, 25(5):622–625, May 2010.

[7] Wilmar Hernandez. A Survey on Optimal Signal Processing Techniques Applied to Improve the Performance of Mechanical Sensors in Automotive Applications. *Sensors*, 7(1):84–102, January 2007.

[8] Zhiqiang Ge and Zhihuan Song. A comparative study of just-in-time-learning based methods for online soft sensor modeling. *Chemometrics and Intelligent Laboratory Systems*, 104(2):306–317, December 2010.

[9] Hector J. Galicia, Q. Peter He, and Jin Wang. Comparison of the performance of a reduced-order dynamic PLS soft sensor with different updating schemes for digester control. *Control Engineering Practice*, 20(8):747–760, August 2012.

[10] Luigi Fortuna, Salvatore Graziani, and Maria G. Xibilia. Comparison of Soft-Sensor Design Methods for Industrial Plants Using Small Data Sets. *IEEE Transactions on Instrumentation and Measurement*, 58(8):2444–2451, August 2009.

[11] Claudio Garcia, Cássio de Carvalho Berni, and Carlos EduardoÑeri de Oliveira. Hardware/firmware implementation of a soft sensor using an improved version of a fuzzy identification algorithm. *ISA Transactions*, 47(2):157–70, April 2008.

[12] Guohai Liu, Dawei Zhou, Haixia Xu, and Congli Mei. Model optimization of SVM for a fermentation soft sensor. *Expert Systems with Applications*, 37(4):2708–2713, April 2010.

[13] Mika Liukkonen, Eero Hälikkä, Teri Hiltunen, and Yrjö Hiltunen. Adaptive soft sensor for fluidized bed quality: Applications to combustion of biomass. *Fuel Processing Technology*, 105:46–51, January 2013.

[14] J.C.B. Gonzaga, L.a.C Meleiro, C. Kiang, and Ruben Maciel-Filho. ANN-based soft-sensor for real-time process monitoring and control of an industrial polymerization process. *Computers and Chemical Engineering*, 33(1):43–49, January 2009.

[15] Guillermo D. Gonzalez, J.P. Redard, Renato Barrera, and Moisés Fernández. Issues in soft-sensor applications in industrial plants. *IEEE International Symposium on Industrial Electronics, 1994. Symposium Proceedings (ISIE '94)*, 17(9):380–385, 1994.

[16] Petr Kadlec, Ratko Grbić, and Bogdan Gabrys. Review of adaptation mechanisms for data-driven soft sensors. *Computers and Chemical Engineering*, 35(1):1–24, January 2011.

[17] Luigi Fortuna, Salvatore Graziani, Alessandro Rizzo, and Maria G. Xibilia. *Soft sensors for monitoring and control of industrial processes.* 2007.

[18] Marco A. Moreno-Armendáriz, Carlos A. Duchanoy, Sergio Flores-Velázquez, and Carlos A. Cruz-Villar. FPGA implementation of a pitch and roll soft sensors for active suspension system. In *Proceedings of the IASTED International Conference Signal and Image Processing (IP 2012)*, pages 99–106, Honolulu, USA, 2012.

[19] Arjpolson Pukrittayakamee, Martin Hagan, Lionel Raff, Satish T.S. Bukkapatnam, and Ranga Komanduri. Practical Training Framework for Fitting a Function and Its Derivatives. *IEEE Transactions on Neural Networks*, 22(6):936–947, 2011.

[20] Young-Don Ko and Helen Shang. A neural network-based soft sensor for particle size distribution using image analysis. *Powder Technology*, 212(2):359–366, October 2011.

[21] Muhammad Shakil, Moustafa Elshafei, Mohamed A. Habib, and Farzaneh A. Maleki. Soft sensor for NOx and O2 using dynamic neural networks. *Computers and Electrical Engineering*, 35(4):578–586, 2009.

[22] Giuseppe Napoli and Maria G. Xibilia. Soft Sensor design for a Topping process in the case of small datasets. *Computers and Chemical Engineering*, 35:2447–2456, 2011.

[23] Chun Lu, Bingxue Shi, and Lu Chen. Hardware implementation of an on-chip BP learning neural network with programmable neuron characteristics and learning rate adaptation. *Proceedings of the International Joint Conference on Neural Networks, 2001. (IJCNN '01)*, 1:212–215, 2001.

[24] Tamás Szabó, Lörinc Antoni, Gábor Horváth, and Béla Fehér. A full-parallel digital implementation for pre-trained NNs. *Neural Networks*, 2:49–54, 2000.

[25] André L.S. Braga, Carlos H. Llanos, Maurício Ayala-Rincón, and Ricardo P. Jacobi. VANNGen: a Flexible CAD Tool for Hardware Implementation of Artificial Neural Networks. In *Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2005)*, pages 8–13, 2005.

[26] James G. Eldredge and Brad L. Hutchings. RRANN: a hardware implementation of the backpropagation algorithm using reconfigurable FPGAs. *IEEE International Conference on Neural Networks, 1994. IEEE World Congress on Computational Intelligence*, 4(1):2097–2102, 1994.

[27] Rafid Ahmed Khalil. Hardware Implementation of Backpropagation Neural Networks on Field programmable Gate Array (FPGA). *AL Rafdain Engineering Journal*, 16-3(May 2007):62–70, 2007.

[28] Srÿan Coric, Ilija Latinovic, and Aleksandra Pavasovic. A neural network FPGA implementation. *5th Seminar on Neural Network Applications in Electrical Engineering (NEUREL-2000)*, pages 117–120, 2000.

[29] Aydoğan Savran and Serkan Ünsal. Hardware Implementation of a Feed forward Neural Network Using FPGAs. 18(3):1045–9227, 2006.

[30] Suhap Sahin, Yasar Becerikli, and Suleyman Yazici. Neural network implementation in hardware using FPGAs. *Lecture Notes in Computer Science*, pages 1105–1112, 2006.

[31] S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, May 2007.

[32] Yutaka Maeda and Toshiki Tada. FPGA implementation of a pulse density neural network with learning ability using simultaneous perturbation. *IEEE Transactions on Neural Networks*, 14(3):688–695, January 2003.

[33] Haitham Kareem Ali and Esraa Zeki Mohammed. Design Artificial Neural Network Using FPGA. *International Journal of Computer Science and Network Security (IJCSNS)*, 10(8):88–92, 2010.

[34] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of mathematical biology*, 5(4):115–133, 1943.

[35] Daqi Zhu. The Research Progress and Prospects of Artificial Neural Networks. *Southern Yangtze University Transaction*, 3(1), 2004.

[36] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[37] James L. McClelland and David E. Rumelhart. *Explorations in parallel distributed processing: A handbook of models, programs, and exercises*, volume 344 of *Computational models of cognition and perception*. MIT Press, 1988.

[38] Robert Hecht-Nielsen. Theory of the Backpropagation Neural Network. In *International Joint Conference on Neural Networks, 1989. (IJCNN 1989)*, pages 593–605, 1989.

[39] Carl Latino, Marco A. Moreno-Armendáriz, and Martin Hagan. Realizing General MLP Networks with Minimal FPGA Resources. In *Proceedings of International Joint Conference on Neural Networks, 2009. (IJCNN 2009)*, pages 1722–1729, 2009.

[40] Martin Hagan, Howard B. Demuth, and Mark Beale. *Neural network design*. University of Colorado Bookstore, 1996.

[41] Pedro Isasi Viñuela and Inés M. Galván León. *Redes de Neuronas Artificiales: Un Enfoque Práctico*. Pearson Educación S.A., Madrid, 2004.

[42] Terasic Technologies. Altera DE2-70 User manual, 2007.

[43] Mariusz Bajger and Amos Omondi. Low-error, High-speed Approximation of the Sigmoid Function for Large FPGA Implementations. *Journal of Signal Processing Systems*, 52(2):137–151, October 2007.

[44] Yahaya Md. Sam and Johari Halim Shah Bin Osman. Modeling and Control of the Active Suspension System Using Proportional Integral Sliding Mode Approach. *Asian Journal of Control*, 7(2):91–98, October 2008.

[45] Feng Jun-ping, Bei Shao-yi, Yuan Chuan-yi, and Zhang Lan-chun. Research on Wheelbase Preview Control for Vehicle Semi-active Suspension Based on Neural Networks. *2009 Third International Symposium on Intelligent Information Technology Application*, 3:290–293, 2009.

[46] Toshio Yoshimura, Kazuhiko Nakaminami, Masao Kurimoto, and Junichi Hino. Active suspension of passenger cars using linear and fuzzy-logic controls. *Control Engineering Practice*, 7(1):41–47, January 1999.

[47] Toshio Yoshimura and Itaru Teramura. Active suspension control of a one-wheel car model using single input rule modules fuzzy reasoning and a disturbance observer. *Journal of Zhejiang University SCIENCE*, 6A(4):251–256, April 2005.

[48] J. Z. Feng, J. Li, and F. Yu. GA-based PID and Fuzzy Logic control for active vehicle suspension system. *International Journal of Automotive Technology*, 4(4):181–191, 2003.

[49] Carlos A. Duchanoy. Desarrollo de un modelo dinámico integral de un vehículo todo terreno con 6 subsistemas, su validación y estudio de maniobrabilidad y confort. Master's thesis, CIC-IPN, 2012.

[50] Kwangsuck Boo, Jaewoo Park, Dongrak Lee, Bonggun Ch, and Sooman Son. Development of a Vehicle Height Sensor for Active Suspension. *International Conference on Control, Automation and Systems, 2007. (ICCAS '07)*, pages 278–282, 2007.

[51] Herbert Schuette and Peter Waeltermann. Hardware-In-The-Loop Testing of Vehicle Dynamics Controllers- A Technical Survey. *System*, (724), 2005.

Apéndice A

Función sigmoidal y su derivada

La función de activación utilizada en las neuronas de la capa oculta es la función sigmoidal.

$$f(x) = \frac{1}{1+e^{-x}}$$

En el proceso de aprendizaje de la red neuronal, es necesario conocer la derivada de la función, por lo que a continuación se presenta el desarrollo matemático respectivo.

$$\begin{aligned} \frac{df(x)}{dx} &= \frac{(1+e^{-x})(0)-(1)(-e^{-x})}{(1+e^{-x})^2} \\ &= \frac{e^{-x}}{(1+e^{-x})^2} \end{aligned}$$

Entonces se propone el término de la derecha se puede descomponer de la siguiente manera:

$$\frac{df(x)}{dx} = \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}}$$

Aquí se observa algo importante, $\frac{e^{-x}}{1+e^{-x}}$ se puede expresar como sigue:

$$\frac{e^{-x}}{1+e^{-x}} = 1 - \frac{1}{1+e^{-x}}$$

Sustituyendo lo anterior en la derivada obtenida, se tiene lo siguiente:

$$\begin{aligned}\frac{df(x)}{dx} &= \frac{1}{1+e^{-x}} \cdot (1 - \frac{1}{1+e^{-x}}) \\ &= f(x) \cdot (1 - f(x))\end{aligned}$$

Apéndice B

Interfaz desarrollada en MatLab

El siguiente código corresponde a la interfaz desarrollada en MatLab con la que es posible configurar la red neuronal que será entrenada en el FPGA.

```
1  clear all
2    neur = 1;               % Hardware neurons
3    n_p = -2:0.4/6:2;       % Training set
4    in = 1;                 %NN inputs
5    pin = n_p(1:in);        % Filling ANN input set
6    la = 1;                 % Hidden layers
7    str(1) = in;
8    for i = 1:la
9      str(i+1) = 3;         % Neurons in the ith hidden layer
10   end
11   sl = 1;                 %NN inputs
12   [m n] = size(str);
13   str(n+1) = sl;
14   [m n] = size(str);
15   target = 1+sin((2*pi/4)*n_p);  % Target set
16   % Gamma
17   gama = 0.1;             % Update rate
18   % Epsilon
19   gama(2) = 0.001;
20   mx = 1;                  % MAX limit random
21   mn = -1;                 % MIN limit random
22   %BIAS
23   bc = 0;
24   for j = 1:la+1                  %Bias generation by layer
25       for k = 1:str(j+1)          %Bias generation by neuron
```

```
26              bc = bc+1;
27              bias(bc) = mn + (mx-mn) * rand();
28          end
29      end
30  % WEIGHTS
31  wc = 0;
32  for j = 1:la+1
33      for i = 1:str(j+1)     % Weight generation by layer
34          for k = 1:str(j)  % Weight generation by neuron
35              wc = wc+1;
36              wei(wc) = mn + (mx-mn) * rand();
37          end
38      end
39  end
40  % Preparing memories
41  str
42  memcont = 1;
43  stps = 0;
44  w = [];
45  b = [];
46  sel = 1;
47  contw1 = 0;
48  contw = 0;
49  contb1 = 0;
50  contb = 1;
51  [mstr,nstr] = size(str);
52  for k = 1:nstr-1
53      pre = str(memcont);
54      post = str(memcont+1);
55      for i = 0:neur:post-1
56          for j = 1:pre
57              contw = contw+1;
58              contw1 = contw1+1;
59              for p= 1:neur
60                  if ((post-i)<p)
61                      w(p,contw1) = 0;
62                  else
63                      w(p,contw1) =wei(contw+(pre*(p-1)));
64                      stps = stps+1;
65                  end
66              end
67          end
68          contw = contw+(stps-pre);
69          stps = 0;
70          contb1 = contb1+1;
```

```
71          for p= 1:neur
72              if ((post-i)<p)
73                  b(p,contb1) = 0;
74              else
75                  b(p,contb1) = bias(contb);
76                  contb = contb+1;
77              end
78          end
79      end
80      memcont = memcont+1;
81  end
82  disp('Memories ready to be send to the FPGA');
83  disp('Push KEY0 to reset the FPGA');
84  pause
85  %Sending thru serial port
86  se = input('\nWhich COM are you using? COM# [COM5]: ','s');
87  disp(' ');
88  if isempty(se)
89      se = 'COM5';            %se: serial port identifier
90  end
91  s2 = serial(se,'BaudRate',9600,'DataBits',8);
92  fopen(s2)
93  %-----------------------------------------------------------
94  %sending data str
95  %begin
96  str(end+1) = 255;   %FF
97  str
98  [m,n] = size(str);
99  for i = 1:n
100     HHB = 0;                %High-High (24-31)
101     HLB = 0;                %High-Low  (16-23)
102     LHB = 0;                %Low-High  (8-15)
103     LLB = str(i);           %Low-Low   (0-7)
104     fwrite(s2,HHB)
105     fwrite(s2,HLB)
106     fwrite(s2,LHB)
107     fwrite(s2,LLB)
108 end
109 %end
110 fwrite(s2,255)
111 fwrite(s2,255)
112 fwrite(s2,255)
113 fwrite(s2,255)
114 %-----------------------------------------------------------
115 %sending inputs
```

```
116 % begin
117 [m,n] = size(pin);
118 for i = 1:n
119      a = int2float_32(pin(i));
120      HHB = a(1 :8);
121      HLB = a(9 :16);
122      LHB = a(17:24);
123      LLB = a(25:32);
124      fwrite(s2, bin2dec(HHB))
125      fwrite(s2, bin2dec(HLB))
126      fwrite(s2, bin2dec(LHB))
127      fwrite(s2, bin2dec(LLB))
128 end
129 % end
130 fwrite(s2,255)
131 fwrite(s2,255)
132 fwrite(s2,255)
133 fwrite(s2,255)
134 for p = 1 : neur
135 %------------------------------------------------------
136 %sending weight1
137 % begin
138      [m,n] = size(w);
139      for i = 1:n
140           a = int2float_32(w(p,i));
141           HHB = a(1 :8);
142           HLB = a(9 :16);
143           LHB = a(17:24);
144           LLB = a(25:32);
145           fwrite(s2, bin2dec(HHB))
146           fwrite(s2, bin2dec(HLB))
147           fwrite(s2, bin2dec(LHB))
148           fwrite(s2, bin2dec(LLB))
149      end
150 % end
151      fwrite(s2,255)
152      fwrite(s2,255)
153      fwrite(s2,255)
154      fwrite(s2,255)
155 %------------------------------------------------------
156 %sending bias1
157 % begin
158      [m,n] = size(b);
159      for i = 1:n
160           a = int2float_32(b(p,i));
```

```
161          HHB = a(1  :8);
162          HLB = a(9  :16);
163          LHB = a(17:24);
164          LLB = a(25:32);
165          fwrite(s2, bin2dec(HHB))
166          fwrite(s2, bin2dec(HLB))
167          fwrite(s2, bin2dec(LHB))
168          fwrite(s2, bin2dec(LLB))
169      end
170  % end
171      fwrite(s2,255)
172      fwrite(s2,255)
173      fwrite(s2,255)
174      fwrite(s2,255)
175  %————————————————————————————————————————————————
176  % sending target
177  % begin
178      [m,n] = size(target);
179      for i = 1:n
180          a = int2float_32(target(p,i));
181          HHB = a(1  :8);
182          HLB = a(9  :16);
183          LHB = a(17:24);
184          LLB = a(25:32);
185          fwrite(s2, bin2dec(HHB))
186          fwrite(s2, bin2dec(HLB))
187          fwrite(s2, bin2dec(LHB))
188          fwrite(s2, bin2dec(LLB))
189      end
190  % end
191      fwrite(s2,255)
192      fwrite(s2,255)
193      fwrite(s2,255)
194      fwrite(s2,255)
195  %————————————————————————————————————————————————
196  % sending gamma
197  % begin
198      [m,n] = size(gama);
199      for i = 1:n
200          a = int2float_32(gama(p,i));
201          HHB = a(1  :8);
202          HLB = a(9  :16);
203          LHB = a(17:24);
204          LLB = a(25:32);
205          fwrite(s2, bin2dec(HHB))
```

```
206             fwrite(s2, bin2dec(HLB))
207             fwrite(s2, bin2dec(LHB))
208             fwrite(s2, bin2dec(LLB))
209         end
210  % end
211         fwrite(s2,255)
212         fwrite(s2,255)
213         fwrite(s2,255)
214         fwrite(s2,255)
215  %----------------------------------------------------------
216  % sending patterns
217  % begin
218         [m,n] = size(n_p);
219         for i = 1:n
220             a = int2float_32(n_p(p,i));
221             HHB = a(1  :8);
222             HLB = a(9  :16);
223             LHB = a(17:24);
224             LLB = a(25:32);
225             fwrite(s2, bin2dec(HHB))
226             fwrite(s2, bin2dec(HLB))
227             fwrite(s2, bin2dec(LHB))
228             fwrite(s2, bin2dec(LLB))
229         end
230  % end
231         fwrite(s2,255)
232         fwrite(s2,255)
233         fwrite(s2,255)
234         fwrite(s2,255)
235  end
236  fclose(s2)
237  delete(s2)
238  clear all
239  end
```