Algoritmos de aprendizaje neurocomputacionales para su implementación hardware

Tesis Doctoral

D. Francisco Ortega Zamorano

Departamento de Lenguajes y Ciencias de la Computación
Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

Junio de 2015

Algoritmos de aprendizaje neurocomputacionales para su implementación hardware

Memoria que presenta para optar al título de Doctor por la Universidad de Málaga

D. Francisco Ortega Zamorano

Dirigida por los Doctores

Dr. Leonardo Franco y Dr. José Manuel Jerez Aragonés

Departamento de Lenguajes y Ciencias de la Computación
Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

Junio de 2015

Departamento de Lenguajes y Ciencias de la Computación
Escuela Técnica Superior de Ingeniera Informática
Universidad de Málaga

El Dr. D. Leonardo Franco, Profesor Titular de Universidad, y el Dr. D. José Manuel Jerez Aragonés, Profesor Titular de Universidad, ambos pertecientes al área de Ciencias de la Computación e Inteligencia Artificial de la E.T.S. Ingeniera Informática de la Universidad de Málaga,

Certifican que,

D. Francisco Ortega Zamorano, Ingeniero en Telecomunicaciones, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo su dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada:

Algoritmos de aprendizaje neurocomputacionales para su implementación hardware

Revisado el presente trabajo, estimamos que puede ser presentado al tribunal que ha de juzgarlo. Y para que conste a efectos de lo establecido en la legislación vigente, autorizamos la presentación de este trabajo en la Universidad de Málaga.

Málaga, Junio de 2015

Fdo.: Dr. Leonardo Franco

Fdo.: Dr. José Manuel Jerez Aragonés

A toda mi familia por confiar en mí.
Sin ellos nada de esto hubiera sido posible.

Agradecimientos

Me gustaría expresar mi más sincero agradecimiento a los directores de mi Tesis Doctoral, Leonardo Franco y José Manuel Jerez Aragonés, por su inestimable colaboración tanto en mi tesis doctoral como en mi carrera investigadora, gracias a su ayuda he podido afrontar todos los retos que han ido surgiendo y he podido iniciar una etapa profesional en la que espero seguir contando con su apoyo y amistad.

Quiero hacer una mención especial a Ignacio Molina por confiar en mí y darme la oportunidad de conocer el mundo de la investigación; a Marcelo Montemurro por su acogida durante mi estancia de investigación en Manchester, ofreciéndome la oportunidad de disfrutar de una gran experiencia que espero repercutan en futuros lazos de colaboración y a Paula Monasterio por ofrecerme de forma totalmente desinteresada su ayuda y sus conocimientos en los inicios de mi etapa doctoral.

Me gustaría también expresar mi más profunda gratitud a mis compañeros de laboratorio de Inteligencia Computacional en Biomedicina (ICB): Subi, Dani, Rafa, Esteban, Héctor y Julio; por echarme una mano cada vez que lo he necesitado, por acogerme con la mejor de las sonrisas y por su amistad que espero perdure en el tiempo.

No podía faltar en estos agradecimientos una mención especial para todos mis amigos que han soportado estoicamente mis comentarios e historias sobre mi tesis, Nio, Cristi (ella sabe lo duro que es una tesis), Pablo Tabo, Moni, Pipe, Rake, Nico, Arturo,... y a los que me dejo seguro en el tintero.

Para terminar, agradecer de una forma muy especial a Sandra el apoyo incondicional, con ella todos los agobios y problemas son más fáciles de llevar. A mis padres por su cariño y porque siempre han demostrado que puedo contar con ellos. También a mi Hermana que siempre está en los malos momentos y espero que siempre este en los buenos; y a mis niñas porque su alegría siempre me contagia. Gracias a todos.

Resumen

Las redes de neuronas artificiales son un paradigma de aprendizaje y procesamiento automático inspirado en el funcionamiento del sistema nervioso central de los animales. Han tenido una gran evolución desde que en 1943 McCulloch y Walter Pitts introdujeran el concepto de neurona artificial, gracias en gran medida a modelos y algoritmos más complejos publicados con posterioridad como el modelo de Hopfield y el algoritmo Backpropagation.

Hoy en día los sistemas neurocomputacionales se emplean en toda una variedad de aplicaciones en sectores tan importantes como el financiero, medico, energético, industrial, de la robótica o el científico. En la mayoría de estos campos la utilización de algoritmos neurocomputacionales se han ido extendiendo y ampliando en su uso, y en todos ellos van apareciendo nuevas aplicaciones donde la utilización de la programación tradicional sobre ordenadores no puede dar una solución de manera eficiente a un problema dado, ya sea por el elevado tiempo de cómputo en problemas complejos o por su consumo y dimensiones en sistemas empotrados.

Los sistemas en tiempo real y las redes de sensores son dos de las tecnologías más extendidas donde la utilización de modelos neurocomputacionales requiere un desarrollo en dispositivos y el empleo de técnicas de programación diferente a las convencionales. En este tipo de aplicaciones otros dispositivos hardware como las FPGAs (Field Programmable Gate Array) o microcontroladores son más adecuados a la hora de la implementación de redes neuronales artificiales.

Una FPGA es un circuito integrado semiconductor basado en una matriz de bloques de lógica configurables conectados entre sí y, a su vez, con celdas de entrada y salida. Dichas interconexiones (también programables) forman una matriz de enrutado modificable según la funcionalidad necesaria por parte del usuario. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica hasta sistemas combinacionales complejos, siendo utilizadas sobre todo en aplicaciones de sistemas en tiempo real que requieran un alto grado de paralelismo.

Por otro lado, un microcontrolador es un circuito integrado programable capaz de ejecutar las órdenes grabadas en su memoria. Está compuesto por varios bloques funcionales que cumplen una tarea específica, como la unidad de procesamiento, memoria, los puertos de entrada/salida, etc. Por su versatilidad son dispositivo que se encuentran en todo tipo de aplicaciones, desde instrumentos de la vida cotidiana a la más avanzada tecnología aeroespacial.

En ciencias de la computación, los sistemas en tiempo real son aquellos sistemas hardware y software que están sujetos a unas limitaciones temporales dadas por la naturaleza del propio sistema. Controlan o actúan sobre un entorno mediante la recepción de información, el procesamiento de la misma y la devolución de una respuesta con la suficiente rapidez (en un rango de tiempo determinado) como para actuar sobre

el entorno. Las respuestas en tiempo real a menudo son del orden de milisegundos y en ocasiones microsegundos. Por el contrario, un sistema sin restricciones temporales no puede garantizar una respuesta dentro de un período de tiempo prefijado.

La estructura y operatividad de las FPGAs ofrecen la posibilidad de realizar diseños eficientes de sistemas en tiempo real debido a que se pueden implementar funciones complejas para que sean ejecutadas de forma simultánea aprovechando su paralelismo, superando en potencia de cómputo a los procesadores digitales convencionales con paradigma de ejecución secuencial. De esta forma se pueden controlar las señales de entradas y salidas del dispositivo a nivel hardware, consiguiendo unos tiempos de respuesta muy acotados que coinciden con los requerimientos de una aplicación en tiempo real.

Existen multitud de aplicaciones en tiempo real y en muchas de ellas se emplean modelos neurocomputacionales. La FPGAs son dispositivos muy adecuados para implementaciones de algoritmos de redes neuronales ya que el tratamiento de información en este tipo de algoritmos se realiza de forma paralela motivando su utilización, más aún cuando se imponen restricciones temporales que las conviertan en un sistema neurocomputacional en tiempo real.

Existen diferentes estrategias de implementaciones neurocomputacionales en función de si se desarrolla o no el proceso de aprendizaje dentro de la FPGA, llamándose "on-chip" y "off-chip". En aplicaciones donde no se incluyan el proceso de aprendizaje en el propio dispositivo ("off-chip") todo el proceso se realiza generalmente en un ordenador personal, transmitiendo la red resultante a una FPGA que actúa a modo de acelerador hardware en la fase de explotación del modelo. Este tipo de implementación es más fácil de desarrollar ya que se puede reutilizar el software ya diseñado para trabajos previos, por el contrario ofrece una estructura muy rígida que no permite cambiar ni la arquitectura de la red ni modificar los pesos sinápticos de forma ágil y eficiente. Por otro lado, si el proceso de aprendizaje está incluido dentro de la propia FPGA ("on-chip") los recursos hardware utilizados son superiores, pero se obtienen estructuras flexibles que posibilitan hacer modificaciones de forma sencilla. La elección del algoritmo utilizado es determínate en este último tipo de sistemas ya que influirá en el tiempo de aprendizaje y la complejidad de la arquitectura a utilizar.

Por este motivo, una de las finalidades principales de esta tesis es el análisis y desarrollo de implementaciones hardware para sistemas neurocomputacionales en tiempo real con estructura de aprendizaje "on-chip". En esta línea de investigación existen dos posibles alternativas para este tipo de sistemas. Una es realizar implementaciones específicas de algoritmos ya existentes y muy conocidos en los que se realicen modificaciones para simplificar el proceso con el fin de conseguir modelos adaptados a los dispositivos hardware. Si bien los cambios realizados deben ser suficientes para simplificar el proceso, no deben alterar la estructura del algoritmo ya que la ventaja de esta alternativa es el uso de modelos muy estudiados que han demostrado su rendimiento. La otra opción es diseñar implementaciones específicas y eficientes de nuevos algoritmos que se adapten mejor a este tipo de dispositivos. Aunque a priori esta opción pueda parecer la más eficaz, puede resultar poco beneficiosa si el algoritmo presentado no genera un modelo neurocomputacional válido a pesar de obtener una implementación hardware eficiente. Una de las motivaciones principales para la realización de esta tesis es conocer las limitaciones y fortalezas de ambas alternativas de desarrollo y así determinar la opción más viable y productiva en este tipo de implementaciones.

Una de las tecnologías que más ha avanzado en los últimos años ha sido las redes de sensores inalámbricas, debido en mayor medida a los avances tecnológicos registrados

en la última década sobre la capacidad de los microcontroladores usados en este tipo de aplicaciones, así como la reducción en su coste y consumo energético lo que permite la utilización a gran escala de redes de este tipo de dispositivos. En la actualidad muchas aplicaciones precisan dotar de un actuador en los nodos sensores que convierta una señal eléctrica en un movimiento físico para modificar el entorno según la información recibida.

La programación tradicional de sensores/actuadores puede conducir a decisiones incorrectas cuando las condiciones ambientales evolucionan con el tiempo, por lo que es necesario cambiar o adaptar el proceso de toma de decisiones a las nuevas circunstancias. Una motivación de esta tesis es de dotar de inteligencia a las redes de sensores en este tipo de escenarios. Esta inteligencia consistirá en proporcionar cierta información adicional junto a las variables medidas para dar un soporte automatizado a la toma de decisiones y al procesamiento distribuido, aportando una respuesta variable en función del cambio producido.

A pesar del gran avance tecnológico producido en los últimos años en el campo de la microelectrónica, que permite disponer actualmente de microcontroladores con una capacidad de cómputo y memoria muy elevados a precios asequibles, estos recursos siguen siendo una importante limitación para la implementación de algoritmos de aprendizaje sofisticados sobre este tipo de dispositivos. Por ello es necesario diseñar implementaciones de modelos neurocomputacionales eficientes para desarrollar redes de sensores inteligentes.

Por lo tanto esta tesis doctoral tiene como objetivo avanzar en el conocimiento científico y tecnológico necesario para el diseño y desarrollo de modelos neurocomputacionales en dispositivos hardware específicos (microcontroladores y FPGAs), con el fin de permitir su utilización en aplicaciones de sistemas en tiempo real y redes de sensores. Para ello se investigan y desarrollan estrategias de diseño para este tipo de dispositivos que maximicen la eficiencia en la utilización de recursos.

De forma más detallada, esta tesis busca alcanzar los siguientes objetivos parciales: (i) analizar el conocido algoritmo de red neuronal Backpropagation con el fin de implementarlo en los dos dispositivos hardware a considerar, desarrollando y evaluando diferentes técnicas de optimización del algoritmo para cada dispositivo, comparando dichos resultados con una programación tradicional ejecutada en un PC y probar así la mejoría de las técnicas propuestas; (ii) evaluar una alternativa al algoritmo Backpropagation para poder realizar una implementación hardware maximizando el uso de recursos. Se estudia y compara el algoritmo de red neuronal constructivo, C-Mantec, como alternativa aprovechando sus singularidades a la hora de generar arquitecturas de red reducidas; (iii) realizar un estudio exhaustivo de una implementación eficiente hardware del algoritmo C-Mantec para evaluar su posible utilización en aplicaciones de sistemas en tiempo real, siendo los tiempos empleados en el aprendizaje y la explotación las variables críticas para determinar su utilización; y (iv) evaluar una implementación del algoritmo constructivo C-Mantec sobre microcontroladores para su utilización en redes de sensores con aplicación a predicción microclimáticas, alarma de emergencia y sensor de caídas.

Implementación del algoritmo Backpropagation en dispositivos hardware: FPGA y microcontrolador

En el capítulo 2 se expone un estudio detallado de las implementaciones del algoritmo de aprendizaje neurocomputacional Backpropagation para dos dispositivos hardware diferentes como son una FPGA y un microcontrolador, centrándose en la obtención de implementaciones eficientes en términos de uso de recursos y velocidad de cálculo. El algoritmo Backpropagation ha sido ampliamente estudiado en numerosos trabajos, con lo que en primer lugar se ha analizado su diagrama de flujo para determinar los procedimientos a diseñar. En este primer paso se identifica a la función sigmoidea como función de transferencia para la salida de dicho modelo.

El algoritmo se ha implementado en ambos casos utilizando la estrategia entrenamiento/validación/testeo con el fin de evitar el sobreajuste provocado por excesivas iteraciones. Para ello se divide el conjunto de datos a estudiar en tres subconjuntos, uno para realizar el entrenamiento con el que la red modifica sus valores, otro para comprobar la tendencia del error generado por la red neuronal y un último para verificar la capacidad de aprendizaje del modelo resultante. Cada vez que la red realiza una iteración (entrenar el modelo con todos los datos del subconjunto de entrenamiento) se comprueba el error cuadrático medio del subconjunto de validación; si éste es menor al mínimo almacenado por el sistema, se almacena la red que ha obtenido dicho error y el error mínimo se cambia por el obtenido. Por último la red resultante (aquella con el error cuadrático mínimo) se ejecuta con el subconjunto de testeo para comprobar su precisión en generalización.

Para el caso específico de la implementación hardware en la FPGA se ha introducido un nuevo diseño consistente en una neurona (Primera Capa) que combina los elementos de la entrada con las funcionalidades de la primera capa oculta, permitiendo reducir drásticamente el número de recursos utilizados para su implementación. Esto es debido a que los recursos empleados en los elementos de la entrada son absorbidos, con un pequeño incremento de uso de celdas lógicas, por las neuronas de la "Primera Capa" evitando el uso de la memoria y bloques específicos para realizar las operaciones inherentes a los elementos de entrada.

Además, debido a las características intrínsecas del algoritmo se ha introducido un esquema de división en el tiempo para la realización de las multiplicaciones derivadas de la ejecución del algoritmo. Este esquema de procesamiento es viable puesto que en el modelo propuesto las multiplicaciones se hacen de manera secuencial, con lo que sólo es necesario utilizar un bloque multiplicador en cada neurona. Dichos bloques pueden implementarse de dos modos diferentes, mediante combinación de celdas lógicas o mediante el uso de bloques específicos, siendo esta última la más eficiente al usar un DSP (digital signal processor) por cada multiplicador.

Otra estrategia diseñada a la hora de realizar la implementación hardware ha sido emplear una tabla de búsqueda junto a una interpolación para aproximar la función de transferencia. Para ello se utiliza una tabla donde se registran una serie de valores equiespaciados de la función sigmoidea y con el fin de reducir el error producido por el redondeo, se calcula una interpolación lineal de los valores tabulados obteniendo un error absoluto por debajo de 0,001. La interpolación es posible realizarla sin un incremento significativo de recursos ya que se dispone de un bloque multiplicador para realizar las operaciones en cada neurona, el cual está disponible en el momento del cálculo gracias a la multiplexación por división en el tiempo previamente descrita. Esta

estrategia se ha modificado para poder ser utilizada también en la implementación sobre microcontrolador, comprobando que se obtiene una mejora sustancial en la velocidad de operación del algoritmo en dicho dispositivo.

Para ambas implementaciones se ha redefinido la representación del tipo de datos utilizado, pasando de la clásica representación en punto flotante usada en este tipo de modelos a la representación de punto fijo, con la consiguiente reducción en la memoria utilizada para el manejo de las variables y aumento en la velocidad de procesamiento. La reducción de memoria se debe a que con representación entera o de punto fijo (si es necesaria una parte fraccionaria) se puede seleccionar el número de bits a utilizar, mientras que la mayoría de las representaciones de punto flotante se realizan con doble representación, es decir con 64 bits. La reducción de velocidad es intrínseca al tipo de datos, puesto que aunque en la actualidad existen unidades aritmético-lógicas muy potentes para las operaciones en punto flotantes, éstas siguen siendo inferiores a las de números enteros.

En una implementación hardware real la limitación en cuanto al tamaño de las arquitecturas de las redes neuronales que pueden ser simuladas viene determinada por el dispositivo FPGA utilizado (una placa Virtex V -XC5VLX110T- en este trabajo). Estudiando las características de la placa y los recursos necesario para cada neurona se observa que la principal limitación proviene de la cantidad de bloques DSP disponibles. La FPGA mencionada incluye 64 bloques DSP, por lo que el número total de neuronas que se pueden implementar de manera simultánea es 63 ya que es necesario un bloque DSP adicional para realizar el proceso de validación.

Se han realizado pruebas de diseño para una arquitectura determinada con el fin de demostrar la correcta implementación del modelo en una FPGA y un microcontrolador, para ello se ha realizado un estudio de la evolución de los errores cuadráticos medios en el proceso de aprendizaje, validación y testeo de las dos implementaciones junto a una tercera realizada en un PC, observando que los errores son similares para validación y testeo en las tres implementaciones. El uso de una representación con menor precisión afecta al error de aprendizaje (que es menor para la aplicación PC), pero no es relevante respecto a la exactitud de la predicción.

Implementación FPGA del algoritmo constructivo C-Mantec

En el capítulo 3 se analiza la implementación hardware de todo el proceso de aprendizaje ("on-chip") de un nuevo algoritmo de red neuronal constructivo que, a diferencia de los modelos basados en el algoritmo Backpropagation, generan la arquitectura de la red de forma automática mientras se ejecuta el proceso de entrenamiento. Este nuevo algoritmo es el C-Mantec, el cual genera arquitecturas muy compactas y posee una muy buena capacidad de generalización sin necesidad de una capa de salida. Además sustituye la función sigmoidea como función de transferencia de salida de la red por la función mayoría, calculando la activación de la red en función del estado de la mayoría de las neuronas.

Cada neurona dispone de una temperatura específica (T_{fac}) que va descendiendo conforme los datos del conjunto de entrada necesiten ser aprendidos (es decir, la decisión de la red sea incorrecta). Cada vez que un dato de entrada precise ser aprendido la neurona con mayor T_{fac} de las que hayan tomado la decisión incorrecta modifica su peso sináptico con el objetivo de aprender dicho dato de entrada. Cuando la neurona encargada de realizar el proceso tenga un T_{fac} tan pequeño que el cambio de su peso

sináptico sea mínimo entonces la arquitectura añade una neurona y se resetean todas las temperaturas. Una de las principales ventajas de utilizar este nuevo algoritmo es el hecho de que evita el problema de seleccionar la arquitectura adecuada, ya que este proceso se realiza automáticamente de acuerdo con la complejidad de la los datos de entrada, siendo una ventaja adicional su robustez respecto al ajustes de los parámetros.

La implementación de este algoritmo se ha diseñado buscando la reducción en el uso de recursos y simplificando los procesos complejos, prestando especial atención a los procedimientos del cálculo de la función mayoría, del valor de la temperatura específica en cada neurona y de las multiplicaciones del algoritmo.

El cálculo de la función mayoría se realiza comparando la suma de la salida de todas las neuronas con salida activas con respecto a la mitad de las neuronas disponibles en ese momento en la arquitectura. Este proceso puede realizarse en sólo un ciclo de reloj puesto que la división entre dos del número de neuronas activas se puede implementar con un registro de desplazamiento.

El cálculo de la temperatura específica (T_{fac}) involucra dos procedimientos; el primero el cálculo del propio valor del T_{fac}, que a su vez requiere de una función exponencial que se implementa mediante la interpolación de los valores tabulados, procedimiento similar al descrito para la función sigmoidea; el segundo procedimiento es especificar qué neurona con posibilidades de aprender tiene el mayor valor de T_{fac}, para lo que se realizan una serie de comparaciones en grupos de 16 neuronas con el fin de maximizar las comparaciones sin extra dimensionar el comparador.

Las multiplicaciones necesarias en el desarrollo del algoritmo se realizan con un solo bloque multiplicador con estructura de división en el tiempo para cada neurona. Se puede implementar con un solo bloque específico o mediante lógica combinacional. En este caso se optó por la lógica combinacional debido a la posibilidad de exportar dicha implementación a otras placas de diferentes familias y marcas en detrimento del rendimiento en cuanto a velocidad de procesamiento.

Se realizaron diversas pruebas sobre diferentes conjuntos de datos de referencia estudiados en múltiples trabajos, demostrando que la generalización obtenida en la implementación hardware es muy similar a la que se obtiene con una implementación realizada en el lenguaje de programación C sobre PC. Además se observa un claro incremento en la velocidad de cálculo en comparación con las implementaciones estándares realizadas mediante el PC. Para funciones complejas, que requieren una mayor arquitectura de red para su aprendizaje, los tiempos de cómputo en un PC tienen una tasa de crecimiento muy superior a los de la implementación en FPGA, lo que las hace más idóneas para implementaciones de problemas complejos.

Un estudio pormenorizado en la velocidad de cómputo revela que en la implementación en PC el tiempo empleado en añadir otra nueva neurona crece exponencialmente conforme el algoritmo va añadiendo neuronas a la capa oculta , mientras que en una FPGA el tiempo en añadir una nueva neurona crece linealmente; esto es debido principalmente al carácter paralelo de cómputo de la FPGA, haciéndola más eficiente en términos de velocidad cuanto mayor es la complejidad de la arquitectura a simular (más neuronas en la capa oculta).

Redes de sensores inteligentes basados en modelos neurocomputacionales

En el capítulo 4 se introduce un nuevo concepto en la reprogramación de nodos sensores. Las técnicas desarrolladas hasta la actualidad como método para la actualización

de los nodos en una red de sensores requieren generalmente del uso de reprogramación. Si los sensores se encuentran en entornos cambiantes dicha reprogramación puede llegar a ser habitual, incrementando de forma sustancial los costes en términos de tiempo y energía.

Se presenta una alternativa al enfoque tradicional para la reprogramación de nodos sensores/actuadores en entornos cambiantes, basada en un esquema de aprendizaje en el propio sensor ("on-chip") para que de forma automática adapte el comportamiento a las condiciones del entorno. El modelo de aprendizaje propuesto se basa en el C-Mantec, ya mencionado anteriormente, que al generar arquitecturas de tamaño muy compacto lo hacen especialmente adecuado para las implementaciones del microcontrolador al reducir significativamente el uso de memoria.

La placa microcontroladora utilizada ha sido el Arduino UNO, se trata de una placa muy popular de código abierto, económico y eficiente. Las ventajas de utilizar esta familia de placas es diverso pero cabe destacar que al ser de código abierto dispone de documentación y tutoriales muy completos que la hacen fácil de programar, además de presentar una comunidad de usuarios muy extensa que facilita el acceso a nuevos prototipos.

Con el objetivo de realizar la implementación íntegra del algoritmo C-Mantec en la placa Arduino se ha cambiado el paradigma de representación de datos del tradicional punto flotante usado en este tipo de algoritmos a la representación en punto fijo. Una vez realizada la implementación, su correcta ejecución se verifica con una comparación de los resultados obtenidos y los valores teóricos en generalización de una serie de conjuntos de datos de entrada.

El sistema neurocomputacional implementado se ha validado en una red de sensor/actuador en tres casos de estudios con el fin de demostrar la eficiencia y la versatilidad de la aplicación resultante. Los tres casos seleccionados son problemas definidos en entornos cambiantes y por lo tanto la toma de decisiones se debe adaptar acorde a los cambios observados, requiriendo una adaptación del modelo de red neuronal que controla el proceso de decisión.

El primer caso de estudio es una alarma de incendios. Generalmente las alarmas incorporan un proceso de toma de decisiones en función de los valores medidos, si dichos valores pasan un determinado umbral la alarma se activa. Para generar las reglas de decisión se realiza un estudio en un ambiente genérico, pero si las condiciones en la estancia a controlar no son los estándares la alarma sonará. Para no realizar un estudio de cada estancia a medir se ha diseñado un sensor con una red neuronal para tomar la decisión de activación, teniendo en cuenta que si toma una decisión incorrecta el propio sistema modifica su red neuronal para adecuarla a la estancia a sensar.

El segundo caso de estudio es la predicción meteorológica para decidir si activar el sistema de riego de un campo de regadío. Existen multitud de estudios relacionado con las variables climáticas y muchos de ellos utilizan modelos computacionales. El problema surge cuando se quiere realizar una acción sin tener en cuenta las variables microclimáticas de la zona. En este caso los estudios generalistas pueden no resultar adecuados y los estudios específicos pueden resultar muy costosos, por lo que se ha decidido la implementación de un modelo de red neuronal en el propio sensor para que capte las variables climáticas con el fin de modificar la red conforme se tengan datos de la zona sensada. Con el fin de maximizar la memoria disponible se ha discretizado las variables sensadas debido a que almacenar los datos reales es muy costoso en cuanto a memoria utilizada.

Por último se ha estudiado un sistema de detección de caídas en personas mayores. El sensor viene equipado con un acelerómetro que indica la posición en el eje x, y, z del individuo. El conjunto de datos del problema viene definido por medidas de una serie de caídas simuladas y otras medidas de acciones habituales que no son caídas. La dificultad surge cuando el sistema de caídas se coloca en diferentes tipos de personas con diferente morfología, comportamiento y hábitos de vida, en ese momento el sistema puede conducir a decisiones incorrectas. Si los sensores van equipados con un modelo neurocomputacional adaptarán su toma de decisiones en función de las variables anteriormente mencionadas dando lugar a un sensor específico para cada usuario.

Conclusiones y líneas de trabajos futuros

En conclusión, se puede decir que el algoritmo Backpropagation se implementó correctamente en dos dispositivos hardware, una FPGA y un microcontrolador, realizándose dichas implementaciones bajo un paradigma de aprendizaje que incluye un sistema de validación para evitar el sobreajuste.

La implementación FPGA precisó de diferentes actuaciones sobre la estructura del algoritmo para minimizar los recursos utilizados. Las modificaciones diseñadas, que se centran en Introducir el paradigma de una nueva neurona de la "Primera Capa", incorporar la multiplexación por división en el tiempo del bloque multiplicador, así como tabular e interpolar los valores de la función sigmoidea, han permitido un reducción de recursos, en media, de un 25,8 % de celdas lógicas y un 50'3 % de bloques específicos en una serie de arquitecturas de diferente tamaño en comparación con implementaciones hardware tradicionales sin mejoras.

Al analizar las limitaciones impuestas por la cantidad de recursos disponibles, se observa que la limitación principal es el número de bloques específicos DSP de la placa FPGA, limitando el número de neuronas disponibles puesto que cada neurona precisara de un bloque específico DSP. Al comparar los resultados con las publicaciones de trabajos previos se observa una reducción en términos de recursos de bloques específicos y celdas lógicas, lo que permite un incremento sustancial en el número máximo de neuronas que dispondría una arquitectura de red neuronal. Sin embargo, el aumento de dicho tamaño, para casos particulares, dependerá de la combinación de los valores de la arquitectura de red neural y recursos de la placa FPGA a utilizar.

En el caso específico del microcontrolador la modificación del tipo de representación de los datos permite un incremento en la velocidad de cómputo de entre 8 a 18 veces más rápido, además de una reducción importante en la cantidad de memoria utilizada. Estos resultados avalan el empleo de sistemas neurocompuacionales en el propio microcontrolador con estructura de aprendizaje "on-chip".

El estudio del algoritmo Backpropagation muestra que las implementaciones específicas de dispositivos hardware diferentes al PC deben diseñarse con modificaciones que las hagan más adecuadas a cada dispositivo en los que se desarrolla. De esta forma se consigue maximizar los recursos disponibles y reducir drásticamente los tiempos de cómputo en los dispositivos FPGAs y microcontroladores.

La implementación íntegra (con estructura de aprendizaje "on-chip") del algoritmo C-Mantec ha sido realizada de forma específica para su implantación en una placa FPGA y la eficacia del diseño se ha demostrado mediante una comparación frente a los valores teóricos de los resultados obtenidos de la generalización de una serie de conjunto de datos conocidos.

Además, haciendo un análisis del tiempo de cómputo de cada conjunto de datos se observa una disminución del tiempo empleado en las implementaciones sobre FPGA en relación a las realizadas sobre PC, señalando que dicho aumento de velocidad es proporcional al tamaño de la arquitectura de la red. Esto es debido a que el tiempo de cómputo en un ordenador crece de forma exponencial conforme se añaden neuronas a la capa oculta y de manera lineal en una FPGA, dando lugar a implementaciones hasta 47 veces más rápidas en arquitecturas más complejas.

Las pruebas anteriormente mencionadas muestran que la representación de datos utilizada (representación en punto fijo) es suficiente para lograr resultados comparables en cuanto generalización y tamaño de la arquitectura a los ejecutados en un PC con representación en punto flotante, siendo este algoritmo muy robusto en cuanto al tamaño de palabra utilizado en la representación de los datos.

La implementación hardware del algoritmo C-Mantec es un 15 % más eficientes en recursos hardware utilizados en comparación a la del algoritmo Backpropagation. El análisis de los resultados confirma que las arquitecturas implementadas en una misma placa FPGA del algoritmo C-Mantec permiten mayor número de neuronas que las realizadas para el algoritmo Backpropagation. Además, las ejecuciones del algoritmo C-Mantec precisan de menor tamaño de representación de datos para que sean precisas, por lo que se constata que dicha implementación es menos sensible a la longitud de palabra utilizada, lo que permite un ahorro de recursos al reducir la complejidad de la implementación.

Para finalizar es importante mencionar que el tiempo de aprendizaje de las implementaciones FPGAs en ambos algoritmos es notablemente menor que el tiempo empleado por un PC, siendo la del C-Mantec sustancialmente inferior al del Backpropagation. Además el tiempo de ejecución del modelo es considerablemente inferior para el C-Mantec, lo que supone una ventaja en la fase de explotación del modelo. A la luz de los resultados observados se puede concluir que el presente análisis demuestra la idoneidad del algoritmo C-Mantec para su aplicación en problemas industriales del mundo real en las que se precise de modelos neurocomputacionales en tiempo real.

El algoritmo C-Mantec se ha implementado en la placa Arduino, para lo que se ha modificado el paradigma de representación de datos reduciendo considerablemente la memoria utilizada para el almacenamiento de variables y aumentando el número de neuronas máximas disponible por la arquitectura. Además, un estudio de la velocidad de procesamiento revela que las implementaciones del mismo algoritmo en punto fijo (8 bits en parte decimal) son hasta cinco veces más rápidas que las de punto flotante para las arquitecturas más grandes, debido principalmente a que el tamaño de la arquitectura repercute negativamente de forma severa en la velocidad de cómputo para las representaciones de punto flotante ya que el manejo de memoria y el uso de una unidad aritmético lógica en este tipo de representación son más complejos.

La correcta implementación del algoritmo ha sido verificada con la comparación de los resultados obtenidos para una serie de conjuntos de datos. Se ha comparado la generalización y tamaño de las arquitecturas con los valores teóricos para dichos conjuntos, obteniendo como resultado que el tamaño de las arquitecturas en el microcontrolador es ligeramente superior a las teóricas conforme crece el número de entradas del conjunto de datos a entrenar, observándose también una pequeña reducción de la precisión. Si bien se aprecia una degradación del modelo debido principalmente a los efectos producidos por el redondeo de la representación del tipo de datos, esto no condiciona la eficiencia del sistema resultante ya que la generalización se mantiene en unos valores

cercanos a los óptimos.

El algoritmo implementado se ha empleado como una red de sensor/actuador en tres casos de estudios con el fin de demostrar la eficiencia y la versatilidad de la aplicación resultante. Los tres casos de estudios seleccionados son problemas definidos en entornos cambiantes, y por lo tanto la toma de decisiones del sensor/actuador ha de adaptarse en consecuencia a los cambios observados, por lo que requieren una reconversión del modelo de red neuronal que controla el proceso de decisión. Los tiempos de reprogramación observados son significativamente bajos en los tres casos de estudio, siendo en consecuencia el consumo de energía del dispositivo también bastante pequeño. Incluso sin una comparación exhaustiva con el caso tradicional en el que el nuevo código tiene que ser transmitido desde una unidad de control central, los resultados hacen evidente una reducción en el gasto energético, cualidad muy importante en este tipo de tecnología debida a la corta duración de las baterías que lo alimentan.

Como resultado, se ha demostrado la idoneidad del algoritmo C-Mantec para su aplicación en una tarea compleja utilizando un microcontrolador Arduino UNO. Hoy en día, dada la existencia de dispositivos con mucho más poder de cómputo y recursos que la placa considerada, el presente estudio permite confirmar la potencial aplicación del algoritmo propuesto en tareas reales que necesitan sensores/actuadores.

Finalmente, como conclusión global se puede decir que esta tesis doctoral profundiza en el estudio de implementaciones neurocompuacionales para dispositivos diferentes a los tradicionales (FPGA y microcontroladores) en tecnologías y aplicaciones que así lo requieran. Además demuestra las ventajas potenciales de las FPGAs como dispositivos usados a modo de aceleradores hardware para aplicaciones de neurocomputación dada sus capacidades intrínsecamente paralelas, mientras que en relación con el uso de las redes neuronales en microcontroladores destacamos la estructura de aprendizaje "on-chip" que permiten su uso en sensores remotos utilizando un modo de funcionamiento autónomo. Después de un análisis exhaustivo se puede determinar que las implementaciones de estos dispositivos deben diseñarse de acuerdo a sus particularidades de composición y configuración, consiguiendo unos resultados en términos de eficiencia de recursos hardware y velocidad de cómputo muy superiores a los modelos neurocomputacionales desarrollados en un PC. Además, se ratifica que existen otros modelos de red neuronal más adecuados a los tradicionales, con los que se consiguen arquitecturas superiores en tamaño y tiempo de respuesta inferiores.

Las futuras líneas de investigación que puede seguir el estudio realizado en esta tesis es muy variado, alguno de los iniciados son evolucionar la implementación hardware del algoritmo Backpropagation para disponer de arquitecturas más grandes gracias a la reutilización de una sola capa que permite simular diferentes capas ocultas de la arquitectura de red; o analizar la posibilidad de emplear las FPGAs como aceleradoras hardware para simulaciones de sistemas complejos con una necesidad de cómputo elevado como el modelo Ising, usado para estudiar el comportamiento de los materiales ferromagnéticos. Además existen otras posibilidades de futuros trabajos que necesitan ser explorados como son estudiar otros modelos computacionales con otras reglas de decisión para su uso en redes de sensores inteligentes con el fin de ahorrar recursos al no necesitar guardar una cantidad ingente de datos; o analizar la posibilidad de aplicar los sistemas neurocomputacionales en tiempo real en tareas complejas como un sistema de estabilización de un cuadricóptero.

Abstract

Artificial neural networks are a paradigm of learning inspired by the automatic processing and functioning of the central nervous system of animals. They have had a great evolution since 1943 when Warren McCulloch and Walter Pitts introduced the concept of artificial neuron, thanks largely to the introduction of more complex models and algorithms such as the Hopfield model and the Backpropagation algorithm in the 80s. These two models produced a neural network renaissance, and nowadays neurocomputational systems are used in a variety of applications in key sectors such as finance, medical, energy, industrial, robotics or science. In most of these fields the use of neurocomputational algorithms have been extended and expanded, and in almost all of them new applications are appearing where the use of traditional programming on computers cannot provide an efficient solution, partly due to the high computation needs for complex problems or factors like energy consumption and dimensions for the case of the implementation of models in embedded systems.

Real-time systems and sensor networks are two of the most widespread technologies in which neurocomputational applications may require the use of alternative devices with different programming techniques. In these kinds of applications hardware devices such as FPGA (Field Programmable Gate Array) or microcontrollers might be more adequate for the implementation of artificial neural networks.

FPGAs are semiconductor integrated circuits based on an array of configurable logic blocks interconnected between them and with input/output signals. The programmable interconnections generate a routing matrix, modifiable by the user according to the needed functionality. They can be programmed to generate a broad range of processes from very simple function, like those carried out by a logic gate to very complex combinational systems like functions needed for the operation of real time systems.

On the other hand, a microcontroller is an integrated programmable circuit able to execute orders etched into its memory. It consists of several functional blocks that execute a specific task as the processing unit, memory, input/output, etc. Due to its versatility, microcontrollers are found in several applications, ranging from everyday life devices to the most advanced aerospace technology.

In computer science, real-time systems are hardware and software systems subject to tight time restrictions determined by the nature of the application. They control or act according to environmental conditions by receiving information, processing it and returning a response quickly enough. Real-time responses are often on the order of milliseconds, even microseconds sometimes. By contrast, a non-real time system is not constrained to respond within a predetermined time period.

The structure and effectiveness of FPGAs offer the possibility for the efficient design of the real-time systems because they can implement complex functions that can be executed simultaneously by taking advantage of their intrinsic parallelism, having also

a computing power similar or higher than conventional sequential operated digital processors. FPGAs permit inputs and outputs to be controlled at hardware level, and can provide faster response times and specialized functionality that meet the requirements of real-time applications.

FPGAs are devices well suited for neurocomputational algorithms due to the parallel processing of information in such algorithms, motivating their use especially when time constraints are imposed for the case of real-time systems.

A classification of neurocomputational implementations can be performed depending on whether the learning process is developed within the FPGA ("on-chip") or not ("off-chip"). In applications where the learning process is not included in the device ("off-chip") this process is generally performed in a personal computer, to then transmit the resultant neural network to the FPGA which acts as a hardware accelerator in the execution phase of the model. This kind of implementation is easier to develop since only the final architecture is codified into the FPGA as training is done externally, usually in a PC with the possibility of reusing existing software. On the contrary, this scheme offers a very rigid structure that does not allow for the modification of the architecture and/or synaptic weights in a quick and efficient manner. The other situation in which the learning process is included within the FPGA ("on-chip" learning) larger hardware resource usage is needed, but this scheme offers more flexibility as several algorithm features can be easily modified on-line. Furthermore, for this last scheme, the choice of training algorithm is a very relevant feature as it strongly influences the FPGA code implementation, learning times and final model architecture, all critical factors in terms of the time and effort required for the problem implementation and the final execution model.

For the previously mentioned reasons, one of the main purposes of this thesis is the analysis and development of hardware implementations for real-time neurocomputational systems with an "on-chip" learning scheme. In this line of research two main alternatives are possible: a) The first one is to develop specific implementations of well-known existent algorithms introducing modifications to simplify the process in order to get the models adapted to the hardware devices; the changes should be sufficient to simplify the process although should not alter the structure of the algorithm since the advantage of this alternative is the use of well-studied models that have demonstrated their performance. b) The second option is to design implementations of new algorithms better suited to the devices in order to achieve efficient implementations. This option seems the most effective if the proposed algorithm does generate a valid neurocomputational model. One of the motivations of this thesis is to analyze the limitations and strengths of both alternatives to determine the most viable and productive option.

Apart from FPGAs, wireless sensor networks are one of the technologies that has advanced more in recent years due to the technological advances on the microcontrollers used in these kind of applications, together with a drastic reduction both in cost and power consumption, that allow for the use of large-scale networks. The traditional programming way of sensor/actuator systems can lead to wrong decisions when conditions change over time, so in several situations it is necessary to change or adapt the decision making process to new circumstances. Another motivation of this thesis is to include intelligence into sensor networks in changing environments, in order to use some additional information along with the measured variables to give an automated support decision-making and distributed processing, providing a smart environmental

dependent response according to the sensed information.

In spite of the great technological advances in recent years in the field of microelectronics that permitted to construct microcontrollers with high computing and memory capacities at very affordable prices, microcontrollers resources still remain a major constraint to implementations of sophisticated learning algorithms. Therefore it is necessary to design efficient neurocomputational implementations models to develop smart sensor networks.

As an overall goal, this thesis aims to advance in the scientific and technological knowledge necessary for the design and development of neurocomputational models for specific hardware devices (microcontrollers and FPGAs) to allow their use in real-time and sensor network applications. In more detail, this thesis seeks to achieve the following partial objectives: (i) analyzing the well-known neural network algorithm, Backpropagation, for its implementation in specific hardware devices, developing and evaluating different techniques of optimization in order to compare its performance with its standard PC version, and thus prove the improvement of the proposed techniques; (ii) evaluating alternatives to the Backpropagation algorithm to perform a hardware implementation in which device resources are maximized. The constructive neural network algorithm, C-Mantec is analyzed as an alternative since it produces very compact architectures; (iii) developing a comprehensive study of an efficient hardware implementation of the algorithm C-Mantec to assess their possible use in systems applications in real time, being the computational time in the learning and the execution processes the critical variables to determine the use; (iv) evaluating an implementation of C-Mantec constructive algorithm in microcontrollers to use as neural model in smart sensor networks by checking in real applications as microclimate predictions, emergency alarms and elderly people fall detection.

Implementation of the Backpropagation algorithm in hardware devices: FPGA and microcontroller

In Chapter 2 a detailed study of implementations of neural computational learning algorithm Backpropagation is carried out in two different hardware devices such as FPGA and a microcontroller, focusing on obtaining efficient implementations in terms of resource utilization and computing speed. The Backpropagation algorithm has been extensively studied in several works, so the first step is to analyze the flowchart to determine procedures which should be designed. In this step the sigmoid function is identified as the transfer function for the output of the model.

The algorithm has been implemented in the two cases using the training/validation/testing strategy to avoid overfitting caused by excessive training iterations; in order to obtain this scheme the dataset is divided into three subsets, a first to perform the training to modify the synaptic weight values of the model, another to check the evolution of the error, and the last one to verify the predictive capacity of the resulting network. Whenever the network carry out an iteration (training the model with all training data set) the mean squared error of the validation data set is checked; if it is less than the minimum stored by the system, the network which generates this error is saved and the stored minimum error is changing by the obtained one. Finally, the obtained network (the one with the minimum mean squared error) runs with the test data set to check the quality of generalization.

For the specific case of hardware implementation in FPGA a new design of a neuron

(First Layer) has been introduced that it combines the elements of the inputs with the functionalities of the first hidden layer, allowing a drastic reduction on the resources used for the implementation. In this new scheme, the resources used for the inputs are incorporated in the neurons of the "First Layer" with a small increase in use of logic cells, without needing the use of memory and specific blocks to perform the inherent operations for input elements.

Furthermore, due to the intrinsic characteristics of the algorithm a scheme of time-division multiplexing to perform the multiplications involved in the process is designed This processing scheme is feasible since the proposed model calculates sequentially the multiplications, so it is just necessary to use a single multiplier block for every neuron. These blocks can be implemented in two different ways, by combining logic cells or using specific blocks, the latter being the more efficient when using a DSP (digital signal processor) in each multiplier.

Another strategy designed when the hardware implementation is performed has been to develop a lookup table plus a linear interpolation to approximate the transfer function. For this purpose a table where several equidistant values are stored of the sigmoid function and in order to reduce the error produced by rounding of the values a linear interpolation of the tabulated values is calculated, obtaining an absolute error below 0.001. The interpolation is possible to carry out without a significant increase in resources since a multiplier block for operations in each neuron is available at the time of calculation due to described time-division multiplexing. This strategy has been modified to be also used in the implementation of the microcontroller, checking that a substantial improvement of the computing speed is obtained in this device.

For both implementations, the data type representation has been modified from the classical floating point representation used in these type of models to fixed-point representation, in order to reduce the used memory for the management of variables and for increasing the computation speed. Memory reduction is achieved when using an integer representation (also called fixed point representation when a fractional part is used) because the number of bits used for representing the values can be optimized by the programmer while in general for the floating point cases the double float representation is employed, that needs 64 bits. Speed reduction is intrinsic to data type, and even if powerful arithmetic-logic units for operations in floating point representation have been developed, their improvements are still inferior to using integer numbers.

In a real hardware implementation the limitation in the size of the neural network architectures which can be simulated is determined by the used FPGA board. In our case, a Virtex V (XC5VLX110T) board has been employed, and by analyzing the characteristics of this device and the resources necessary for implementing a single neuron, it is observed that the main constraint for the current implementation is the number of available DSP blocks used to compute products related to a neuron output. The employed FPGA includes 64 DSP blocks so the total number of neurons that can be simultaneously implemented is 63 because one DSP block is required to perform the validation process.

In order to demonstrate the correct implementation of the Backpropagation model both in the FPGA board and Arduino microcontroller several tests have been carried out, in particular by analyzing the evolution of the mean squared error for training, validation and test pattern sets. A comparison to the standard implementation of the algorithm on a PC is also carried out, checking that the level of error is similar for all considered cases. The use of a fixed point representation for the implemented models in

the FPGA and microcontrollers may produce rounding effects larger than in the case of the PC application, but the results demonstrate that the representation used is long enough as the accuracy of the implemented models is not much affected.

FPGA Implementation of the constructive algorithm C-Mantec

In chapter 3 the FPGA implementation of a new constructive neural network algorithm named C-Mantec is analyzed. Unlike models based on the Backpropagation algorithm, C-Mantec generates the network architecture automatically while the training process is executed. Furthermore, this new algorithm generates very compact architectures with a single hidden layer of threshold neurons that permit to obtain very good predictive capabilities. Another difference with the Backpropagation based architectures is that C-Mantec replaces the sigmoid function as output transfer function of the network by a majority function.

Every neuron has a specific temperature (T_{fac}) which will decrease as the input data set is learned. Whenever an input pattern is misclassified and requires to be learned the neuron with the largest T_{fac} among those wrongly computing its output, modifies its synaptic weights in order to learn the data. When the neuron in charge of carrying out the learning process has a T_{fac} too small so that the change of the synaptic weights of the architecture is almost negligible then the system adds one neuron to the network and all specific temperatures are reset. One of the main advantages of using this new algorithm is the fact that it avoids the problem of selecting the right architecture, as this process is performed automatically according to the complexity of the input data, with a further advantage regarding its robustness the parameter settings.

The implementation has been designed aiming at reducing the use of resources and simplifying the whole process, with particular attention to the procedures of calculating the majority function, set of values of the specific temperature of every neuron and the multiplications necessary for the execution of the algorithm.

The computation of the majority function is performed by comparing the number of active neurons in the hidden layer with the total number of neurons present in this layer divided by two, noting that this process can be carried out in only one clock cycle since the division by two of the total number of neurons can be implemented using a shift register.

The calculation of the internal temperature of every neuron (T_{fac}) involves two procedures; the first for computing the value of T_{fac} itself that in turn involves an exponential function which is implemented by using tabulated values plus a linear interpolation scheme, a similar method as described previously for the sigmoid function. The second procedure is dedicated to the selection of the neuron with the largest T_{fac} value (the candidate neuron that will try to learn the current input pattern), and for this task a series of comparisons is implemented in groups of 16 neurons, in order to optimize resource usage.

The implementation of the C-Mantec algorithm involves several multiplications that are performed in a single block using a time-division multiplexing scheme for every neuron. Each of these products can be implemented with one specific block or by using combinational logic. This last option has been selected in this case due to the possibility of exporting the implementations to other different boards.

Several tests on different benchmark data sets previously analyzed in several works have been performed, demonstrating that the obtained generalization using the new

C-Mantec hardware implementation is quite similar to the one obtained from a implementation performed with traditional programming by language C in an PC, noting that also a significant increment of computing speed is obtained in comparison to the traditional PC implementation. In particular, the computation of complex functions requires large network architectures for which longer computational times are required, and thus FPGA implementations are suitable for these cases.

A detailed study on the computation speed reveals that the rate at which a new neuron is added to the network grows exponentially in the PC implementation, while for the FPGA this time grows linearly, confirming the advantage of the parallel nature of the FPGA, being more efficient in terms of speed when the architecture is more complex (more neurons are present in the hidden layer).

Smart sensor networks based on neurocomputational models

Chapter 4 introduces a new concept in relationship to sensor nodes reprogramming. Standard techniques for updating nodes in a sensor network require the reprogramming of the nodes software, and thus if sensors are located in changing environments such reprogramming might be done frequently, increasing substantially the involved costs in terms of time and energy.

This chapter presents an alternative to the traditional approach for sensor/actuator node reprogramming in changing environments based on a scheme of training the sensor node itself to adapt automatically its behavior to the environment conditions. The proposed learning model is based on neural network constructed using the C-Mantec algorithm described in the previous chapter, which generates very compact architectures that makes the approach particularly suitable for microcontroller implementations as memory resources are limited in these type of devices. The microcontroller board chosen has been the Arduino UNO that is a very popular open source computer hardware, economic and efficient. Among the advantages of using this family of boards, the fact that it is developed under an open source format means that very complete documentation and tutorials are easily available as there is a large user community that facilitates the access to new prototypes.

In order to develop the whole implementation of the C-Mantec algorithm into the Arduino board the traditional data type representation has been changed from the standard floating point one used in this type of algorithms to a fixed-point representation that helps to optimize resource utilization. In order to check the correct implementation of the algorithm several checks have been carried out verifying that the results obtained for the generalization ability on benchmark data sets are the same as those obtained with the PC version of the algorithm.

The implemented neurocomputational system has been used as a smart sensors/actuators network in three case studies to demonstrate the efficiency and versatility of the resultant application. The chosen cases correspond to three problems set in changing environments, and therefore a decision-making approach should be adopted to deal with the observed changes, requiring a re-training of the neural network model that controls the decision process.

The first case study is a fire alarm. Usually alarm systems incorporate a decision process that depends on the measured values such that if these values exceed a set threshold value then the alarm is activated. An analysis can be carried out in a generic environment to generate decision rules, although the alarm could be activated when

conditions in the space to control are not the standards. In order to avoid analyzing every possible case, the system modifies its neural based decision model when a wrong choice has been taken, adapting it to the sensed conditions.

The second case study is the problem of weather forecasting for deciding whether to activate or not the irrigation system of a given field. Several studies about climatic prediction have been performed based on neural network models, noting that many complications arise from microclimatic conditions when they are not properly considered. In the present case general studies may not be appropriate, and specific studies can be very expensive. A neural network model within the sensor itself has been implemented in order to sense the climatic variables and modify the network according to the measured data of a specific area. As storing the data set in real values format is very costly, the measurements have been discretized into categories in order to reduce the amount of information to be saved.

Finally, a fall detection system for elderly or disabled people has been analyzed. In this case, the sensor is equipped with an accelerometer which indicates the position of a person using spatial x, y, z coordinates. Several measurements are obtained for simulated falls and for normal situations, to obtain a training/test data set for the problem. As the system might be used by different kinds of people in different environments where their behavior and lifestyles of the user are also different, the system can lead to wrong decisions, so to avoid possible malfunctioning, the sensor will train the neural network based decision making system according to the aforementioned variables in order to obtain a specific adjustment for each case.

Conclusions and future lines of research

In conclusion, it can be said the Backpropagation algorithm has been implemented successfully into two different hardware devices, a FPGA and a microcontroller; these implementations have been developed from a learning scheme with validation process to avoid overfitting problems.

The FPGA implementation requires several changes in the structure of the algorithm in order to minimize the employed resources. Introducing a new paradigm for neurons " First Layer ", incorporating time-division multiplexing of the multiplier block as well as tabular and interpolate the values of the sigmoid function. These changes have allowed an average reduction of resources 25.8 % of logic cells and 50.3 % of specific blocks in a series of architectures of different size in comparison to previous hardware implementations without upgrades.

In the moment to analyze the constraints imposed by the available resources, it can be observed that the main limitation is the number of specific DSP blocks of the FPGA board, therefore DSP blocks restrict the number of available neurons for the architecture as a DSP block is required in each neuron and another for the validation process. Comparing these results with previous published works, a reduction in terms of resources specific blocks and logic cells is observed, allowing a substantial increase in the maximum number of neurons that the architecture of a neural network can manage. However the increase of the size of the architecture, in particular cases, depends on the combination of the neural network values and FPGA board resources used.

In the specific case of the microcontroller, introducing change in data type representation allows an increase in the computing speed of between 8 and 18 times faster, and a reduction in the amount of used memory. These results permit carry out neu-

rocomputational models in the own microcontroller with scheme of "on-chip" learning also in this device.

The study of Backpropagation algorithm shows that the specific implementations in different device to PC should be performed with changes to adapt it to each device in which they are carried out. These methods maximise available resources and dramatically reduce computation times, being shown for FPGAs and microcontrollers.

The full implementation (with "on-chip" learning scheme) of the C-Mantec algorithm has been specifically performed for developing in an FPGA board, the precise design has been demonstrated by a comparison against theoretical values of the generalization of a series of well-known data sets.

In addition a reduction of spent time to learning process in FPGA in relation to PC is observed when a detailed analysis of computation time of each data set is carried out; noting that the speed increase is proportional to the size of the network architecture. This is due to the fact that computation time in a computer grows exponentially when neurons are added to the hidden layer and linearly in an FPGA, resulting implementations up to 47 times faster for more complex architectures.

The aforementioned tests show that the used data type representation (fixed-point representation) is sufficient to obtain similar results in terms of generalization and size of the architecture in comparison to PC implementations with floating point representation, being this algorithm very robust as regards the word size used in the data type representation.

The hardware implementation of the C-Mantec algorithm is 15 % more efficient in terms of the used hardware resources compared to the Backpropagation algorithm. By analyzing these results it can be confirmed that the generated architectures in a FPGA board of the C-Mantec algorithm allow greater number of neurons than for the Backpropagation algorithm.

In addition, the C-Mantec algorithm executions require less word length in data representation for being precise, so it can be verified that such implementation is less sensitive to the used word length, allowing resource savings since the complexity is reduced in the implementation.

To complete the comparison it can be demonstrated that the learning time is substantially smaller in both implementations compared to the process in a PC, noting that this time is notably smaller in the C-Mantec implementation than the Backpropagation. In addition, runtime of the C-Mantec model is considerably lower which is an advantage in the execution phase of the neural network. In light of the observed results it can be concluded that this analysis demonstrates the suitability of the C-Mantec algorithm for application in real-world industrial problems in which real-time neurocomputational models is required.

The C-Mantec algorithm has been implemented in the Arduino, for this the paradigm of data type representation has been changed, greatly reducing the used memory for storing variables as a consequence the maximum number of neurons available for the architecture of the network increases. In addition a study of computing speed reveals that for the same data set, the fixed point implementations (8-bit decimal part) are up to five times faster than floating point for the larger architecture, mainly due to the size of the architecture affects the computational speed in a negative way for floating point representation since memory management and the use of an arithmetic logic unit in this kind of representation are more complex.

Successful implementation of the algorithm is verified by comparing the obtained results with those from previous studies using several benchmark data sets. The generalization accuracy and size of the obtained architectures have been analyzed, observing that the microcontroller architectures are slightly larger than the theoretical values, also noting a small reduction in training accuracy. These two observed changes can be justified as a consequence of rounding effects generated by the data type representation used, noting that the differences are always small and generalization values are quite close to the optimal ones previously obtained using a real data type representation.

The implemented algorithm has been used as a sensors / actuators network in three case studies to demonstrate the efficiency and versatility of the resulting application. The three selected case studies are problems defined in changing environments where the decisions sensor / actuator should be tuned to the sensed conditions, requiring in this way an almost continuous modification of the underlying neural network models used. The observed reprogramming times were significantly low in the three cases and as a consequence the power consumption of the device is also quite low.

Even if a thorough comparison with the traditional case where a new code has to be transmitted from a central control unit were not carried out, the results indicate that a clear energy consumption reduction can be obtained, constituting a very important aspect of this technology. As a result, the suitability of the C-Mantec algorithm for its use in a complex task has been demonstrated using a microcontroller Arduino UNO. Further, given that nowadays there are also newer devices with more computing power and resources than the considered board, the present study confirms the potential of the proposed algorithm for real applications in tasks where sensors/actuators are needed.

Finally, as an overall conclusion we can say that the present thesis extends the study and application of neurocomputational models to FPGA and microcontrollers, devices that constitutes an alternative to standard computers for applications where a dynamic adaptation is needed to the actual conditions. It also demonstrates the potential benefits of FPGAs used as hardware accelerators in neurocomputational applications given their inherently parallel capabilities, while in relation to the use of neural networks in microcontrollers highlight the "on-chip" learning scheme utilized which allows for the use of remote sensors in stand-alone operation. From the carried out analysis, it can be concluded that implementations for these types of devices should be designed according to their specific configuration, as this might permit to achieve important improvements in terms of efficiency of hardware resource usage and computation speed, well above neurocomputational models developed on a PC computer. Further, the results indicate that alternative neurocomputational models might be more suited than the traditional ones for the implementation of larger architectures and in order to obtain faster response times.

Future lines of research that the made study in this thesis can follow is varied. Ones of which have been initiated are evolving the hardware implementation of Backpropagation the algorithm to obtain larger architectures by a layer multiplexing to simulate different hidden layers of the architecture; or analyzing the possibility of using FPGAs as hardware accelerators for simulations of complex systems with a high level of computational needs as the Ising model used to study the behavior of ferromagnetic materials. There are other possibilities for future work which will be studied as the use of the other neurocomputational models with different learning method for being developed in smart sensor networks in order to save resources by not storing a huge amount of data; or analyze the possibility of applying the neurocomputational systems

in real time on complex tasks such as a stabilization system of a quadcopter.

Índice

"La estupidez real siempre vence a la inteligencia artificial"

Terry Pratchett

Capítulo 1

Introducción

1.1. Conceptos generales previos

1.1.1. Redes Neuronales Artificiales

Las redes de neuronas artificiales (ANN: Artificial Neural Networks) son un paradigma de aprendizaje y procesamiento automático inspirado en el funcionamiento del sistema nervioso central de los animales (el cerebro en particular). Las ANN se presentan generalmente como un sistema de "interconexionado de neuronas" que calculan una respuesta a partir de un conjunto de estímulos de entradas. Un sistema de estas características se puede entender como procesadores paralelos masivamente distribuidos cuya función es almacenar conocimiento y representarlo para que esté disponible. Se basan en los siguientes principios:

1. El procesamiento de la información se ejecuta a través de múltiples elementos simples llamados unidades de proceso (neuronas).

2. Las neuronas están conectadas a través de conexiones sinápticas, las cuales transmiten las señales entre ellas.

3. Cada conexión sináptica entre dos neuronas tiene asociado un peso (peso sináptico) que tiene un efecto multiplicador sobre la señal transmitida.

4. Cada neurona dispone de una función de activación (o transferencia) para determinar su señal de salida.

Historia

- *1943:*Warren McCulloch y Walter Pitts publican un modelo de neurona que puede computar funciones aritméticas y lógicas [McCulloch y Pitts (1943)].

- *1949:* Donald Hebb plasma sus ideas sobre el aprendizaje neuronal en la conocida "regla de Hebb".

- *1958:* Frank Rosenblatt empieza el trabajo que conduce a la creación del concepto de perceptrón, siendo la red neuronal más antigua capaz de reconocer patrones [Rosenblatt (1962)].

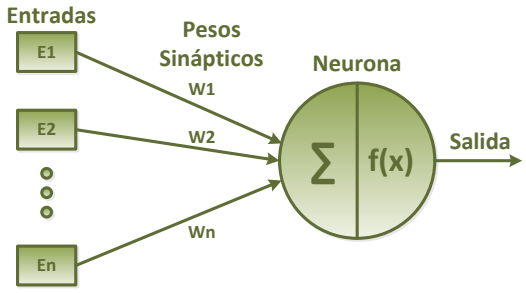

Figura 1.1: Representación de la arquitectura más básica de redes neuronales, un perceptrón simple con n entradas.

- *1960:* Widrow y Hoff desarrollan el "ADALINE", que fue la primera aplicación industrial real.
- *1967:* Marvin Minsky y Seymour Papert frenan esta primera etapa en el desarrollo de redes neuronales por la publicación sobre las limitaciones del perceptrón [Minsky y Papert (1969)].
- *1982:* Jhon Hopfield, con la publicación de su modelo [Hopfield (1982)] junto con la invención del algoritmo "BackPropagation" [Rumelhart et al. (1986)], consigue devolver el interés en el campo de la computación neuronal.
- *1987:* Se celebra en San Diego la primera conferencia abierta sobre redes neuronales (IEEE International Conference on Neural Networks), con más de 1700 participantes, y se constituye la International Neural Network Society (INNS).
- *1988:* Nace la revista Neural Networks; le siguen la revista Neural Computation en 1989 y la IEEE Transaction on Neural Networks en 1990. Posteriormente han ido apareciendo otras muchas y se han creado Institutos de Investigación y programas de formación en Neurocomputación.

Estructura

Una ANN está compuesta de elementos básicos de procesamiento (neuronas) que integran información desde múltiples entradas. Esta información es normalmente procesada mediante un sumador cuyo resultado actúa como entrada de una función de transferencia que calcula la respuesta de dicha neurona. La salida de una neurona se puede conectar a la entrada de otras mediante conexiones ponderadas (pesos sinápticos) que representan la eficacia de la sinapsis de las conexiones neuronales. En la Fig 1.1 se puede observar una representación del esquema más básico de una neurona artificial (perceptrón simple) . Las funciones de transferencias más utilizadas son la función escalón para salidas binarias y la función sigmoidea o la función tangente hiperbólica para salidas reales.

Una red neuronal consiste en un conjunto de neuronas interconectadas de una forma específica. El procesamiento en las ANN no reside solamente en el modelo de las neuronas sino en la manera en la que estas se conectan. Esta manera o estructura se conoce como arquitectura de la red y generalmente se organiza en grupos de neuronas llamados niveles o capas. Una de las arquitecturas más usadas es la de conexionado hacia delante

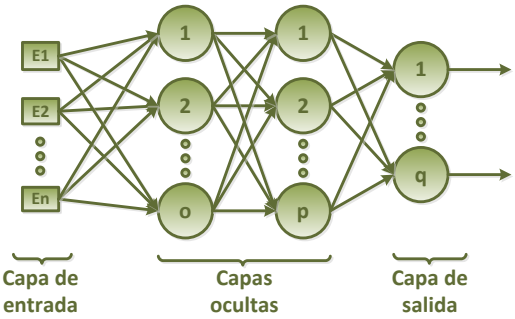

Figura 1.2: Representación gráfica de la estructura genérica de una arquitectura de red Feed-forward con n entradas 2 capas ocultas con o y p número de neuronas en cada capa y con q salidas.

o "feed-forward" que consiste en una capa de entrada, que proporciona información a la red, una serie de capas ocultas, que mapean la información de las entradas y una capa de salida que proporciona la respuesta de la red. La Fig 1.2 representa un modelo genérico de ANN con una arquitectura de red Feed-forward.

Propiedades

El poder de cómputo de una red neuronal deriva principalmente de su estructura masivamente paralela y de su capacidad de aprendizaje y generalización. Además, las redes neuronales disponen de las siguientes propiedades y capacidades que las hacen especialmente atractivas [Haykin (1998)]:

- *Sistemas distribuidos no lineales:* Una neurona puede ser un elemento de proceso lineal o no lineal, con lo que una red neuronal también puede simular sistemas no lineales y caóticos.
- *Relación entrada-salida:* Las ANN son capaces de vincular una salida en función de unos parámetros de entrada bajo la premisa de aprendizaje supervisado.
- *Adaptabilidad:* Una red tiene capacidad de adaptación al entorno a través de la modificación del conjunto de pesos sinápticos que conforman la red.
- *Autoorganización:* Las ANN usan su capacidad de aprendizaje adaptativo para organizar la información que reciben durante el aprendizaje y/o la operación.
- *Tolerancia a fallos:* Una red neuronal es un sistema computacionalmente robusto con alta tolerancia a fallos. Su rendimiento se degrada lentamente bajo un funcionamiento adverso (pérdidas de neuronas).
- *Implementación VLSI:* Gracias al paralelismo intrínseco, las ANN pueden conseguir gran rapidez de cómputo en la realización de determinadas tareas, lo que permite que puedan ser aplicados a sistemas en tiempo real, permitiendo simular sistemas biológicos mediante circuitos de silicio.

Reglas de aprendizaje

Una de las características más importantes de las redes neuronales es su capacidad de aprender interactuando con su entorno o con alguna fuente de información. El aprendizaje de la red es un proceso adaptativo mediante el cual se van modificando los pesos sinápticos de la red para mejorar el comportamiento de la misma. Se distinguen tres paradigmas de aprendizaje:

- *Aprendizaje supervisado:* Se dispone de un conjunto de patrones de entrenamiento para los que se conocen la salida deseada de la red. Un objetivo de este aprendizaje podría ser minimizar el error cuadrático medio cometido entre la salida de la red y la deseada.

- *Aprendizaje no supervisado (competitivo o autoorganizado):* Se dispone de un conjunto de patrones de entrenamiento pero no se conoce la salida que debe generar. La red por sí misma buscará su comportamiento más adecuado atendiendo a cierto criterio para encontrar estructuras o prototipos en el conjunto de patrones.

- *Aprendizaje por refuerzo:* basado en un proceso de prueba y error que busca maximizar el valor esperado de una función criterio conocida como una señal de refuerzo.

Aplicaciones

Las características de los modelos de sistema neurocomputacionales permiten que sean utilizadas en una gran variedad de aplicaciones, las cuales se puede enmarcar dentro de las siguientes categorías:

- Aproximación de funciones o análisis de regresión, incluyendo predicción de series temporales.

- Clasificación, comprendiendo reconocimiento de patrones y secuencias además de toma de decisiones.

- Procesamiento de datos, abarcando filtrado, agrupación, separación y compresión.

- Robótica, implicando manipulación directa y diseño de prótesis.

- Control, incluyendo control numérico por ordenador

Algunas de las aplicaciones en las que las ANN son utilizadas son: predicciones del mercado financiero debido al comportamiento no lineal del mismo [Bahrammirzaee (2010)]; diagnóstico médico para la predicción de enfermedades como el cáncer [Baena et al. (2013)]; tratamiento de imágenes como compresión de imágenes y video [Palomo et al. (2013)] o reconocimiento facial [Zhao et al. (2003)]; robótica [Chaoui et al. (2009)]; planificación y toma de decisiones en la teoría de juegos [Memmert y Perl (2009)], modelado de sistemas complejos en el sector energético [Kalogirou (2001)].

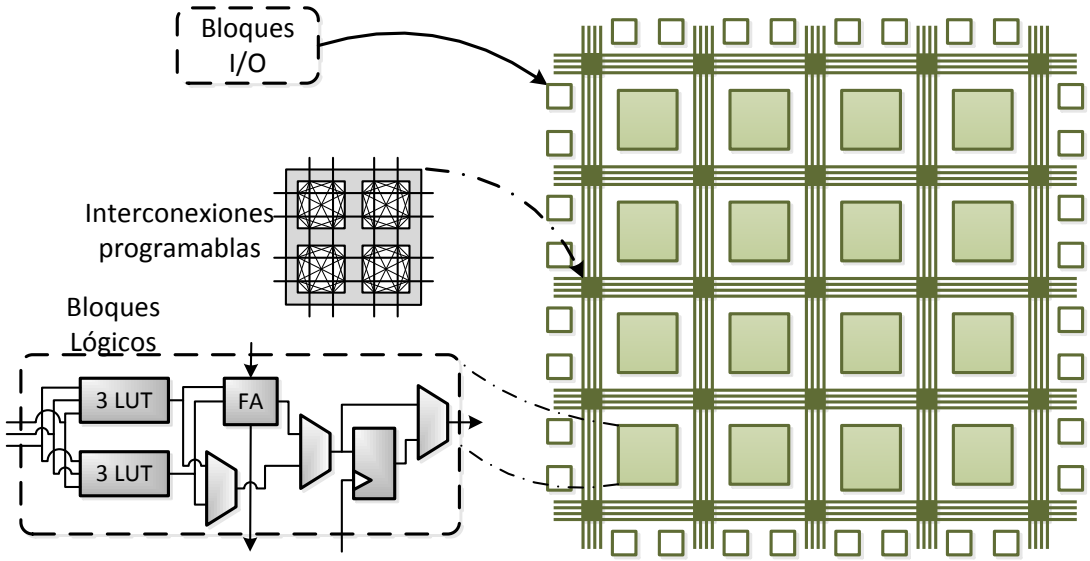

Figura 1.3: Representación de la estructura básica de una FPGA, describiendo los elementos básicos que la componen como son los bloques lógicos, las interconexiones programables y los bloques de entrada salida

1.1.2. FPGA

Una FPGA (Field Programmable Gate Array) es un circuito integrado semiconductor basado en una matriz de bloques de lógica configurables (CLBs) conectados entre sí, y a su vez, con celdas de entrada y salida (I/O) como puede observarse en la Fig. 1.3. Dichas interconexiones (también programables) forman una matriz de enrutado modificable según la funcionalidad necesaria por parte del usuario. Los CLBs se componen generalmente de varias tablas LUTs (Look-Up Table) cuyas salidas están multiplexadas y los bloques I/O comunican a la FPGA con el exterior.

A diferencia de una ASIC (Application-Specific Integrated Circuit), en el que el dispositivo está diseñado a medida para una aplicación determinada, las FPGAs se pueden programar cambiando su configuración para un uso determinado. La lógica programable puede reproducir desde funciones tan sencillas, como las llevadas a cabo por una puerta lógica, hasta sistemas combinacionales complejos. Las FPGAs se utilizan en aplicaciones similares a los ASICs, y aunque son más lentas y presentan un mayor consumo de potencia, proporcionan la gran ventaja de ser reprogramables. Esta característica añade una enorme flexibilidad al flujo de diseño y minimiza tiempos, coste de desarrollo y adquisición para pequeñas cantidades de dispositivos.

Los lenguajes de descripción de hardware más empleados en el diseño de FPGAS son VHDL y Verilog. Ambos son lenguajes que permiten diseñar la FPGA desde un punto de vista abstracto, funcional, aunque también se puede definir la estructura del hardware a bajo nivel. Existen además componentes predefinidos, los IPs, descritos en estos lenguajes para simplificar el diseño de la FPGA. Los principales fabricantes facilitan las herramientas para hacer más sencillo su proceso de diseño.

Reseña histórica

Las FPGA son el resultado de la convergencia de dos tecnologías diferentes, los dispositivos lógicos programables (PLDs Programmable Logic Devices) y los circuitos integrados de aplicación específica (ASIC). La tecnología de circuitos programables se desarrolló en los años 80 a partir de diferentes trabajos que derivaron en patentes, como la de Steve Casselman en 1992, el cual desarrolló un computador con 600.000 puertas programables, o las de David W. Page and LuVerne R. Peterson en 1985 que introdujeron conceptos fundamentales en puertas lógicas programables y bloques lógicos. Sin embargo fue la empresa Xilinx de manos de sus fundadores Ross Freeman y Bernard Vonderschmit la primera en comercializar una FPGA (modelo XC2064) que apenas contaba con 64 bloques lógicos programables con 2 LUTs de 3 entradas. Actualmente, la empresa Xilinx sigue siendo el principal fabricante de placas FPGAs, aunque existen otros fabricantes como Altera (principal competidor), Lattice Semiconductor, Actel ,etc. El mercado de FPGA a nivel mundial en 2013 fue de 5.386 millones de dólares, esperando llegar a 9.552 millones en 2020 [FPG (2014)].

Aplicaciones

Las FPGAs pueden implementar cualquier circuito específico siempre que se disponga de recursos para esa implementación. Están siendo utilizadas cada vez en más aplicaciones, sobre todo en las que es necesario disponer de sistemas de tiempo real y/o sistemas con un alto grado de paralelismo. Un ejemplo de estas aplicaciones pueden ser automoción, sistemas aeroespaciales y de defensa, prototipos de ASICs, procesamiento de imágenes, equipamiento médico, sistemas de visión para computadoras, instrumentación científica, bioinformática, emulación de hardware de computadora, entre otras.

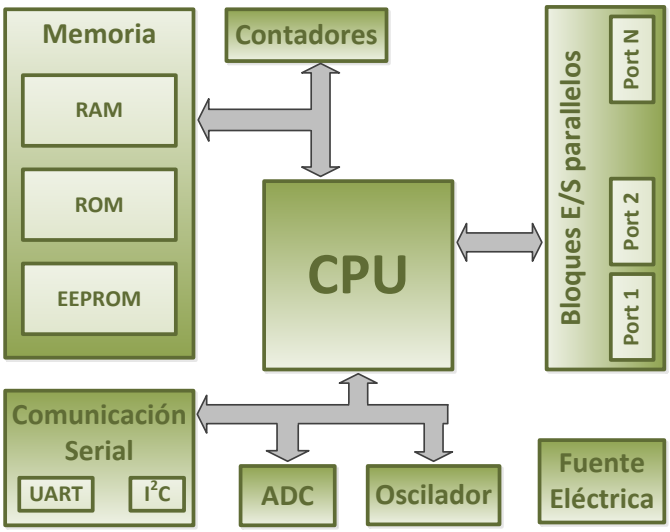

Figura 1.4: Representación gráfica de la estructura genérica de un microcontrolador, pudiéndose observar las diferentes elementos que lo componen

1.1.3. Microcontrolador

Un microcontrolador (μC, UC o MCU) es un circuito integrado programable capaz de ejecutar las órdenes grabadas en su memoria. Está compuesto por las siguientes unidades funcionales cada una con una función específica (Ver Fig. 1.5):

- *Una unidad central de procesamiento (CPU):* Se encarga principalmente de leer la instrucción seleccionada, decodificarla y finalmente ejecutarla. También se encarga de conectar cada parte del microcontrolador.

- *Memoria:* Se utiliza para almacenar datos y programas. Un microcontrolador tiene generalmente una cierta cantidad de memoria RAM y ROM (EEPROM: Electrically Erasable Programmable Read-Only Memory, EPROM: Erasable Programmable Read-Only Memory, etc.) o memorias flash para almacenar los códigos fuente del programa.

- *Puertos paralelos de Entrada/Salida (I/O):* Se utilizan principalmente para manejar o leer interfaces externos, como diferentes tipos de sensores, LCDs, motores, etc.

- *Contadores:* Entre sus principales funciones se encuentran actuar como reloj, modulador, generador de pulso, medidor de frecuencia, generador de oscilación, etc.

- *Comunicaciones serie:* La función principal de esta unidad es dotar de un interfaz serie entre el microcontrolador y otros periféricos mediante comunicación UART o bus I2C.

- *Oscilador:* Genera una señal periódica normalmente sinusoidal o cuadrada para ser empleada a modo de reloj del sistema.

En el momento de fabricación la memoria de un microcontrolador no contiene datos, por lo que es necesario cargar el programa que se desee ejecutar en la memoria EEPROM (o similar) del microcontrolador. Para facilitar el proceso de carga del programa la mayoría de los fabricantes ofrecen un "Bootloader", programa que se graba en el microcontrolador una sola vez para facilitar la carga de posteriores programas sin necesidad de extraerlo del zócalo de la placa de desarrollo, permitiendo grabar los programas por los puertos de comunicaciones. El programa puede ser escrito en diferentes lenguajes de programación, desde lenguajes de bajo nivel como el ensamblador hasta más complejos como JAVA, aunque el más utilizados por los desarrolladores de este tipo de tecnología es el lenguaje C. No obstante, el programa debe ser codificado en sistema numérico hexadecimal antes de ser grabado en la memoria del microcontrolador.

Reseña histórica

El primer microcontrolador data del año 1971, cuando Texas Instruments crea el TMS 1000, que no comercializa hasta 1974. En esa misma fecha Intel lanza el Intel 4004, que sin embargo éste necesitaba circuitos adicionales para funcionar, con lo que poco tiempo después comercializó el Intel 8048, con el que logró un gran volumen de ventas gracias a que combinaba memoria RAM y ROM.

En 1993 la empresa General Instrument dio un salto de calidad en los microcontroladores al desarrollar la familia PIC, los cuales empezaron por primera vez a incluir memoria EEPROM (Electrically Erasable Programmable Read-Only Memory), permitiendo en ese momento un borrado eléctrico y rápido sin necesidad del complicado proceso que precisaban hasta la fecha.

En la actualidad, la reducción de costes provocada por el auge de la tecnología y abaratamiento del diseño está permitiendo la aparición de una gran variedad de microcontroladores, que se clasifican en función de la longitud de palabra utilizada (4, 8, 16, 32, 64 o 128 bits), las diferentes frecuencias de trabajo permitidas (desde varios kilohercios hasta frecuencias elevadas) y los consumos (del orden de milivatios e incluso de microvatios). Por lo general todos incluyen la función de espera, que reduce significativamente tanto la funcionalidad del microcontrolador como el consumo, estado inducido y/o anulado por algún evento como una interrupción o la pulsación de un botón.

Aplicaciones

Los microcontroladores son dispositivos que no sólo se han introducido en todos los ámbitos de la vida cotidiana, como la mayoría de electrodomésticos comunes, aparatos portátiles y de bolsillo, teléfonos móviles, inclusive en muchos elementos de juguetería electrónica, sino que también forman parte de todos los dispositivos de periféricos y dispositivos auxiliares de computación. Además, se utilizan en componentes industriales de instrumentación, sistemas de automoción o elementos diversos sistemas de seguridad y domótica en general y están involucrados en los elementos de tecnología más avanzadas, como sistemas de navegación espacial, en electromedicina, sistemas de control industrial y robótica.

1.2. Motivación

En muchos de los campos de la ciencia se ha generalizado la utilización de algoritmos neurocomputacionales, y en todos ellos van aparecido nuevas aplicaciones donde la utilización de la programación tradicional sobre ordenadores convencionales no es suficiente para poder implementar de manera eficiente una solución a un problema dado, ya sea por el elevado tiempo de cómputo de los ordenadores convencionales en problemas complejos o por su consumo y dimensiones en sistemas empotrados. Los sistemas en tiempo real y las redes de sensores son dos exponentes de aplicaciones donde la utilización de ANN debe realizarse sobre dispositivos y con programación diferente a los convencionales, como las FPGA o los microcontroladores que resultan más eficientes a la hora de una implementación de un modelo neurocomputacional.

1.2.1. Sistemas tiempo real

En ciencias de la computación los sistemas en tiempo real son aquellos sistemas hardware y software que están sujetos a unas limitaciones temporales que vienen dadas por naturaleza del propio sistema. Estos sistemas controlan o actúan sobre un entorno mediante la recepción de datos, el procesamiento de los mismos y la devolución con la suficiente rapidez para actuar sobre el entorno. Las respuestas en tiempo real a menudo son del orden de milisegundos, y en ocasiones microsegundos. Por el contrario, un sistema sin instalaciones en tiempo real no puede garantizar una respuesta dentro de cualquier período de tiempo.

La estructura y operatividad de las FPGAs ofrecen la posibilidad de realizar diseños eficientes de sistema en tiempo real debido a que se puede implementar funciones complejas para que sean ejecutadas de forma paralela, superando la potencia de cómputo de los procesadores digitales convencionales con paradigma de ejecución secuencial. De esta forma, se pueden controlar entradas y salidas (I/O) a nivel hardware ofreciendo tiempos de respuesta más veloces y funcionalidades especializadas que coinciden con los requerimientos de una aplicación en tiempo real.

Debido a la estructura intrínsecamente paralela de la FPGA, este tipo de dispositivos son muy adecuados para implementaciones de algoritmos de redes neuronales, ya que el tratamiento de información se realiza de forma paralela, lo que motiva su uso sobre todo en aplicaciones en tiempo real donde las restricciones temporales son un factor determinante. Existen numerosos ejemplos de aplicaciones en tiempo real [Kuo et al. (2006)] y en las que se emplean ANN [bin Huang et al. (2006)].

En este tipo de implementaciones es determinante el algoritmo utilizado para generar el modelo de ANN puesto que define la arquitectura empleada en el proceso de aprendizaje influyendo directamente en el tiempo de cómputo empleado y en la complejidad del diseño resultante.

Por lo tanto, una de las principales motivaciones de esta tesis es estudiar el uso de las FPGAS como dispositivo en tiempo real y conocer las limitaciones de estos dispositivos en este tipo de aplicaciones, además de diseñar algoritmos neurocomputacionales eficientes desde el punto de vista de los recursos empleados con el fin de poder implementar arquitecturas de ANN cada vez más complejas.

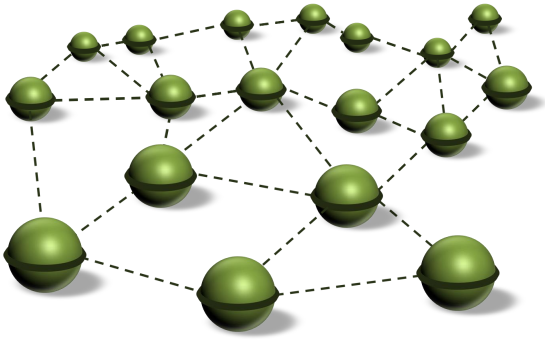

Figura 1.5: Esquema básico de interconexiones generadas por una red de sensores entre sus nodos

1.2.2. Redes de sensores

Una de las tecnologías que más ha avanzado en los últimos años ha sido las redes de sensores debido en mayor medida a los avances tecnológicos registrados en la última década respecto a la capacidad de los microcontroladores usados en este tipo de tecnología, así como la reducción en su costo y consumo energético. Esto ha permitido la utilización a gran escala de redes de este tipo de dispositivos, de hecho, fuentes especializadas estiman en un trillón la cifra de sensores que se alcanzará en 2015 monitorizando todo tipo de actividad [Kharif (2013)].

En los casos en que las condiciones ambientales evolucionan con el tiempo, la programación original de sensores/actuadores puede conducir a decisiones incorrectas, por lo que es necesario cambiar o adaptar el proceso de toma de decisiones a las nuevas condiciones [Sayed-Mouchaweh y Lughofer (2012)]. Por dicho motivo es necesario dotar de "inteligencia" a las redes de sensores, considerando inteligencia en este ámbito la aportación de información adicional a los datos obtenidos para dar soporte a la toma de decisiones y al procesamiento distribuido [Frank (2000)]. En este ámbito, la incorporación a este tipo de sensores de mecanismos cada vez más complejos de adaptación al entorno requiere del diseño de algoritmos específicos, o modificaciones de los ya existentes, para ser implementados en estos dispositivos que presentan limitaciones de recursos en comparación con sistemas de computación comunes (ej. PC).

A pesar del gran avance tecnológico producido en los últimos años en el campo de la microelectrónica, que permite disponer actualmente de microcontroladores con una capacidad de cómputo muy elevada a precios accesibles, los recursos de memoria y cómputo siguen siendo una importante limitación para la implementación de algoritmos de aprendizaje sofisticados sobre este tipo de dispositivos, ya que el tamaño y precio de las motas son aspectos importantes a tener en cuenta a la hora de su implementación física. En este aspecto se centra una de las motivaciones principales en esta tesis, diseñar modelos neurocompuatacionales eficientes con los que dotar de inteligencia a las redes de sensores.

1.3. Estado del arte

1.3.1. Sistemas neurocomputacionales en tiempo real

Los sistemas neurocomputacionales han tenido una gran repercusión en el ámbito científico desde su creación, pero no fue hasta los años 80 y principios de los 90 cuando se produjo un importante avance gracias en su mayor parte al desarrollo de la red de Hopfield, y en especial al algoritmo de aprendizaje de retropropagación (BackPropagation) ideado por Rumelhart y McLellan en 1986 que fue aplicado en el desarrollo de los perceptrones multicapa. Durante estos años se dedicaron grandes esfuerzos a la implementación hardware de sistemas neurocomputacionales sin demasiado éxito. La principal razón de este fracaso se puede atribuir a que la mayoría del trabajo realizado estaba dedicado a la implementación sobre circuitos específicos con tecnología ASIC. El rendimiento obtenido no fue competitivo para justificar la realización de circuitos específicos a gran escala, con lo que las implementaciones hardware sobre ASIC se descartaron en estos inicios. En aquellos momentos la tecnología FPGA no estaba lo suficientemente desarrollada para ser una opción competitiva a la hora de realizar implementaciones neurocomputacionales.

A principios de siglo hubo un resurgimiento de las implementaciones de sistemas neurocomputacionales en tiempo real, con la proliferación de nuevas tecnologías que daban lugar a nuevas familias de FPGAs con más poder de cómputo que permitían la implementación eficiente de este tipo de sistemas.

El libro [Omondi y Rajapakse (2006)] aportó una visión global de los trabajos realizados hasta el momento debido a que se trata de una recopilación de publicaciones de diferentes autores. Los trabajos tratan diferentes cuestiones a la hora de diseñar implementaciones hardware, siendo la más importante la representación del tipo de datos. La naturaleza intrínseca de la FPGA da lugar a utilizar representación entera (punto fijo si es necesaria parte fraccionaria) debido a la eficiencia de este tipo de representación, otro tipo de representación como punto flotante puede ser utilizada pero es necesario el uso de bloques específicos para su utilización [Ho et al. (2009); Jovanovic y Milutinovic (2012)], el trabajo [Savich et al. (2007)] describe un interesante análisis de la implementación de algoritmos neuronales en punto flotante con aritmética de punto fijo.

En estos primeros trabajos ya aparecieron dos posibles alternativas como solución al problema de implementación de modelos de aprendizaje en dispositivos hardware, el concepto de entrenamiento "on-chip" y "off-chip", en función de si incluye o no el proceso de aprendizaje dentro del dispositivo.

En implementaciones "off-chip" el aprendizaje del modelo de red neuronal se realiza generalmente en un ordenador personal (PC) y sólo los pesos sinápticos se transmiten a la FPGA que actúa como un acelerador de hardware. La principal ventaja de este esquema es la simplicidad del diseño hardware ya que todo el proceso de aprendizaje puede realizarse de forma convencional en un ordenador personal estándar y sólo es necesario hacer la implementación hardware de la red resultante. La principal desventaja es la imposibilidad de re-ajustar el modelo directamente en el dispositivo [Himavathi et al. (2007); Jung y Kim (2007); Orlowska-Kowalska y Kaminski (2011)].

Por el contrario, las implementaciones de aprendizaje "on-chip" incluyen el proceso completo de entrenamiento y ejecución en la propia FPGA, permitiendo entrenar los modelos de forma automática independiente del PC, ofreciendo además una gran

flexibilidad y eficiencia en los sistemas resultantes. Como contrapartida, este tipo de implementaciones requieren mucho más recursos de la FPGA [Dinu et al. (2010); Gomperts et al. (2010)].

En la actualidad, se ha extendido el uso de sistemas neurocomputacionales con esquema de aprendizaje "on-chip" en implementaciones hardware debido a que ha demostrado su superioridad en cuanto a la velocidad de cómputo ofreciendo sistemas más eficientes, aunque se hace evidente un incremento en la complejidad del diseño de las aplicaciones resultantes.

Una vez decidido el esquema de aprendizaje "on-chip" como el más adecuado, se plantean diferentes estrategias para implementar ANN en dispositivos FPGAs. Las dos estrategias principales para este tipo de implementaciones son:

1. Mejorar y sacar mayor rendimiento de los algoritmos clásicos ampliamente utilizados en múltiples aplicaciones. Ejemplos de trabajos donde se adoptan esta estrategia son:

   - *Lotrič y Bulić (2012):* Realiza una variación del algoritmo "BackPropagation" para conseguir un incremento de la potencia de cálculo y eficiencia del algoritmo. Para ello se sustituyen los multiplicadores, necesarios en este tipo de algoritmos, por una función aproximada más simple que requiere menos circuitería y que permite obtener mejores resultados sobre diferentes conjuntos de datos.
   - *Gomperts et al. (2011):* Implementa una arquitectura del algoritmo de perceptrón multicapa "BackPropagation" minimizando los costes del hardware y maximizando el rendimiento, la precisión y la parametrización. Presenta además la implementación real de una aplicación de sistemas neurocomputacionales en tiempo real

2. Diseñar nuevos algoritmos que se adapten mejor al tipo de implementación utilizada optimizando la utilización de recursos disponibles. Algunos ejemplos son los trabajos de:

   - *Shawash y Selviah (2013):* Describe una nueva implementación del algoritmo Levenberg-Marquardt, el cual es un algoritmo de aprendizaje no lineal que converge con precisión y rapidez. Como ejemplo, la función XOR es resuelta con tan sólo 13 iteraciones.
   - *Pan y Lan (2014):* Presenta una variación del MLP en la que combina una tercera capa con un algoritmo genético y el método máxima pendiente para hacer una búsqueda global de los pesos sinápticos de forma más rápida y eficiente.
   - *Bahoura (2014):* Propone una red neuronal dinámica con arquitectura en pipeline de alta velocidad para el modelado del comportamiento del amplificador de potencia. La novedad de la arquitectura propuesta reside en una mayor frecuencia de funcionamiento, la latencia de salida inferior y menor grado de utilización de recursos.

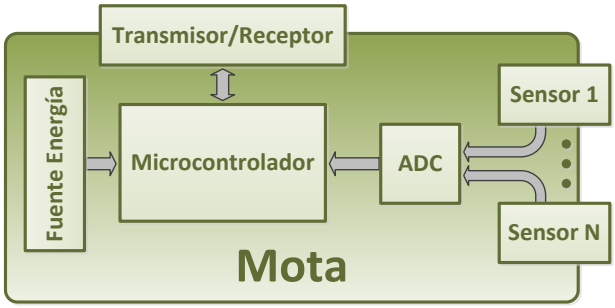

Figura 1.6: Representación gráfica de una mota en la que los sensores envían información al microcontrolador para procesarla y este se comunica con el exterior mediante un Transmisor/Receptor

1.3.2. Redes de sensores inteligentes

Los sensores (o detectores) son dispositivos que permiten la medición de variables químicas o físicas y las transforman en señales eléctricas, que son transmitidas por diferentes medios. Estos dispositivos son unidades autónomas que constan de un microcontrolador, una fuente de energía (usualmente una batería), un transmisor/receptor y un elemento sensor (Fig. 1.6).

Las redes de sensores inalámbricas (WSN) están formadas por un grupo de sensores con ciertas capacidades sensitivas y de comunicación inalámbrica los cuales permiten formar redes ad hoc sin infraestructura física preestablecida ni administración central.

Se conoce poco sobre el nacimiento de esta tecnología, ya que se produjo bajo el auspicio de la industria militar. El desarrollo moderno de pequeñas redes de sensores se remonta al proyecto Smartdust en 1998 y el proyecto de la NASA Sensor Webs. Aunque estos proyectos no tuvieron demasiada repercusión, el proyecto Smartdust desencadenó la ejecución de otros proyectos de investigación, NEST y CENS, desarrollados en la universidad de Berkeley, pioneros en este área de conocimiento y responsables de acuñar el término mota (mote) para referirse a los dispositivos nodo sensor.

Hoy en día el esfuerzo en el desarrollo de las WSN se centra en solucionar los siguientes problemas:

- *Cobertura:* Existen diferentes estrategias que abordan el problema en la fase de implementación [Wang (2011)]: (i) estrategia basada en la fuerza bruta que consiste en distanciar o acercar las motas hasta que queden en una situación de equilibrio en el que se abarque el máximo espacio posible; (ii) estrategia del punto de cuadrícula, que consiste en dividir el área en cuadrículas y poner cada mota en el centro de una; (iii) estrategia de geometría computacional para la optimización, que usa principalmente el diagrama de Voroi y la triangulación de Delaunay.

- *Desarrollo:* Existen 3 niveles de abstracción en cuanto al diseño de la tecnología de WSN para facilitar la labor de los desarrolladores [Laukkarinen et al. (2012)]: nodo, red e infraestructura, siendo ésta última la más importante debido a que se refiere típicamente al "middleware" de la red de sensores. El propósito principal de la abstracción de infraestructura es separar al usuario final de la heterogeneidad de una red de sensores, haciendo el desarrollo de aplicaciones más rápido y fácil.

- *Consumo energético:* La eficiencia energética de una WSN es, quizás, el aspecto más importante a centrarse en el desarrollo de este tipo de tecnología, dado que la recarga de baterías en una red de cierto tamaño puede resultar una tarea altamente costosa en tiempo, por lo que un reto claro en este tipo de redes es reducir significativamente el consumo energético de los dispositivos [Rault et al. (2014)]. Existen diferentes estrategias para reducir el consumo energético, siendo la más habitual la reducción de la transmisión de datos entre los nodos de la red, aunque existen otras vías para reducir el consumo como un enrutamiento energéticamente eficiente, la optimización radio o la estrategia reposo/activación.

Recientemente, en muchas aplicaciones se está considerado el equipamiento de los nodos sensores con un actuador, el cual convierte una señal eléctrica en un movimiento físico. Ejemplo de actuadores son algunos que incluyen válvulas que controlan el agua o la salida de gas, motores eléctricos que abran/cierren puertas y ventanas, interruptores para encender/apagar algún instrumento eléctrico o señales de alarma [Salarian et al. (2012)].

La red sensor/actuador inalámbrica (WSAN) resultante no sólo se compone de nodos de sensores que miden parámetros específicos sino que algunos de ellos tienen la capacidad de actuar sobre el medio ambiente, siendo una característica de este tipo de nodos la necesidad de una cantidad superior de recursos [Melodia et al. (2007)].

En ciertos escenarios donde las variables a sensar evolucionan con el tiempo, la programación original de sensores/actuadores puede conducir a decisiones incorrectas, por lo que es necesario cambiar o adaptar el proceso de toma de decisiones a las nuevas condiciones [Sayed-Mouchaweh y Lughofer (2012)]. La solución tradicional pasa por enviar los datos sensados a una unidad central, donde una persona los interpreta y vuelve a programar el microcontrolador con el nuevo conjunto de reglas Han et al. (2005); Wang et al. (2006); Shaikh et al. (2010).

Se han propuesto diferentes técnicas de reprogramación dinámica para cambiar el comportamiento de los sensores sin tener que reprogramar manualmente, ya que la reprogramación tradicional requiere en la mayoría de los casos la interrupción del proceso para cargar el nuevo código binario, con la consiguiente pérdida de tiempo y energía que participan en el proceso de comunicación entre la unidad central [Rassam et al. (2013); Aiello et al. (2011)].

Un primer paso hacia la reducción de los efectos anteriores ha sido la incorporación de sistemas de aprendizaje automático en el proceso de toma de decisiones, la automatización de la respuesta del microcontrolador sin interrumpir su ejecución y el envío de sólo una pequeña fracción de código para el microcontrolador [Urda et al. (2012); E. Cañete et al. (2012); Farooq et al. (2010)]. Sin embargo, los recientes avances en la potencia de cálculo de los microcontroladores permite la inclusión de sistemas de aprendizaje "on-chip", adaptando el comportamiento de sensores/actuadores dinámicamente según los datos sensados [Aleksendrić et al. (2012); Mahmoud et al. (2013)].

1.4. Objetivos

Esta tesis doctoral tiene como objetivo avanzar en el conocimiento científico y tecnológico necesario para el diseño e implementación de modelos neurocomputacionales en dispositivos hardware específicos (microcontroladores y FPGAs), con el fin de permitir su utilización en aplicaciones de sistemas en tiempo real y redes de sensores.

Se investigan y desarrollan estrategias de diseño para los dispositivos que maximicen la eficiencia en la utilización de recursos, por lo que se evalúan posibles alternativas al clásico algoritmo de aprendizaje para redes neuronales artificiales (algoritmo Backpropagation) utilizado habitualmente en las aplicaciones que precisan neurocomputación. Una alternativa clara parece ser los algoritmos constructivos de red neuronal (CoNN: Constructive Neural Networks) [Franco et al. (2010)], debido a que generan de forma automáticas arquitecturas de red compactas que consumen significativamente menos recursos del dispositivo. De forma más detallada, esta tesis busca alcanzar los siguientes objetivos parciales:

- Analizar el algoritmo de red neuronal "BackPropagation" con la finalidad de evaluar las fortalezas y deficiencias de éste para su diseño e implantación en los dos dispositivos hardware a analizar (FPGA y Microcontroladores). Desarrollar y evaluar diferentes técnicas de optimización del algoritmo para cada dispositivo y comparar dichos resultados con una programación tradicional realizada sobre un PC, con el fin de comprobar la mejoría en eficiencia de las técnicas propuestas.

- Evaluar una alternativa al algoritmo "BackPropagation" para realizar diferentes implementaciones sobre los dos dispositivos hardware estudiados con el objetivo de maximizar el uso de recursos en cada uno de ellos. (i) Analizar los algoritmos constructivos de red neuronal que generan arquitecturas de red de forma automática como alternativa para los modelos neurocomputacionales. (ii) Estudiar el nuevo algoritmo C-Mantec [Subirats et al. (2012)] con la función mayoría como función de salida de la red para comprobar si existe una reducción de la complejidad del modelo que repercuta en la simplicidad de la implementación hardware.

- Realizar un estudio exhaustivo de una implementación eficiente en una FPGA, sobre VHDL, del algoritmo C-Mantec a fin de evaluar su posible utilización en aplicaciones de sistemas en tiempo real. Desarrollar estrategias de diseño del algoritmo para reducir los recursos empleados en su implementación que, sin alterar la integridad del algoritmo, modifiquen la complejidad de los procedimientos. Además realizar un análisis de los tiempos empleados en el proceso de aprendizaje así como de la explotación del modelo, dado que estas variables van a determinar la viabilidad de utilización de la implementación resultante como sistema en tiempo real.

- Evaluar una implementación del algoritmo C-Mantec sobre una placa microcontroladora específica (Arduino UNO) para su utilización en redes de sensores. El objetivo principal es conseguir un aumento significativo en el tamaño de las arquitecturas resultantes así como reducir la memoria utilizada del dispositivo. Otro objetivo es evaluar dicha implementación en aplicaciones reales de redes de sensores como la predicción microclimática, la gestión de alarmas de emergencia y los sensores de caídas.

1.5. Estructura de la tesis

Tras este capítulo introductorio, la memoria de esta tesis se encuentra estructurada en los siguientes cinco capítulos.

El capítulo 2 expone la implementación del algoritmo "BackPropagation" en dos dispositivos hardware como son el microcontrolador y la FPGA, especificando las singularidades de dicho algoritmo para poder diseñar diferentes técnicas con el objetivo de conseguir implementaciones eficientes desde el punto de vista de la velocidad de cómputo y recursos. Se realiza además una comparativa para conocer si esta implementación mejora a las tradicionales de este algoritmo realizadas en un PC. En el apéndice B se describe detalladamente una de las técnicas utilizadas para maximizar la eficiencia, la implementación por medio de tablas de búsqueda e interpolación lineal de las funciones de transferencias usadas en este tipo de algoritmos.

El capítulo 3 describe la implementación de un nuevo algoritmo de red neuronal constructivo, C-Mantec, en una FPGA con el fin de mejorar la velocidad de los cálculos neurocompuacionales. Las características específicas de este nuevo algoritmo hace de él un candidato idóneo para la implementación hardware debido a la reducción de las arquitecturas que genera y a la simplicidad de su estructura, sin perder capacidad de generalización respecto a los algoritmos tradicionales y comprobando que las velocidades de cómputo son más elevadas que las conseguidas en un PC. En el apéndice C se describe una comparativa de la implementación hardware del algoritmo C-Mantec en comparación con la del algoritmo Backpropagation, describiendo las ventajas y desventajas de las dos implementaciones hardware.

El capítulo 4 estudia y diseña una red de sensores inteligentes para su uso en aplicaciones donde las condiciones ambientales evolucionan con el tiempo. Se implementa el novedoso algoritmo de red neuronal constructivo C-Mantec en un microcontrolador Arduino UNO, con el fin de dotar de inteligencia a los nodos sensores y que éstos puedan reprogramarse de forma automática sin necesidad de interacción exterior ahorrando un coste en tiempo y energía. Este proceso está basado en trabajos preliminares recogidos en el apéndice A. Como resultado se demuestra que esta implementación es más eficiente al ser desarrollada con éxito para para tres casos de estudios específicos de la vida real.

Finalmente, el capítulo 5 expone las conclusiones derivadas de esta tesis doctoral junto a las posibles líneas futuras de trabajo en este campo.

Capítulo 2

Implementaciones eficientes del algoritmo Backpropagation en FPGAs y microcontroladores

Francisco Ortega-Zamorano, José M. Jerez, Daniel Urda, Rafael Luque-Baena and Leonardo Franco: Efficient implementation of the Backpropagation algorithm in FPGAs and microcontrollers. **IEEE Transactions on Neural Networks and Learning Systems** (IN PRESS).

Posición JCR: Q1 (7/121) en Computer science: Artificial Intelligence.
Q1 (11/248) en Engineering: Electrical & Electronic.
Factor de impacto: 4,370

Resumen: El algoritmo de aprendizaje BackPropagation, el cual ha sido ampliamente analizado en múltiples estudios, se ha implementado en una FPGA y en un microcontrolador de forma eficiente en términos de uso de recursos y velocidad de cómputo. El planteamiento utilizado en ambos casos para evitar el sobreentrenamiento ha sido la estrategia entrenamiento/validación/testeo. Para el caso específico de la implementación hardware en la FPGA, se ha introducido un nuevo diseño de una neurona que combina los elementos de la entrada con las funcionalidades de la primera capa oculta, permitiendo reducir drásticamente el número de recursos hardware utilizados; además, se ha introducido un esquema de división en el tiempo para realizar todas las multiplicaciones implicadas en el algoritmo con un solo bloque multiplicador para cada neurona. Para ambas implementaciones se ha redefinido la representación del tipo de datos utilizado pasando de la clásica representación en punto flotante, utilizada en este tipo de modelos, a la representación de punto fijo, consiguiendo así una reducción en la memoria utilizada para almacenar las variables del proceso y un aumento en la velocidad de procesamiento. Los resultados muestran que las modificaciones propuestas producen un claro incremento de la velocidad de cómputo en comparación con la implementacion estándar realizadas en un PC, demostrando la utilidad del paralelismo intrínseco de una FPGAs en las tareas neurocomputacionales y la idoneidad de ambas implementaciones del algoritmo para problemas del mundo real.

Efficient implementation of the Backpropagation algorithm in FPGAs and microcontrollers

Francisco Ortega-Zamorano, José M. Jerez, Daniel Urda, Rafael M. Luque-Baena, and Leonardo Franco *Senior Member, IEEE.*

***Abstract*— The well known backpropagation learning algorithm is implemented in a FPGA board and a microcontroller, focusing in obtaining efficient implementations in terms of resource usage and computational speed. The algorithm was implemented in both cases using a training/validation/testing scheme in order to avoid overfitting problems. For the case of the FPGA implementation, a new neuron representation that reduces drastically the resource usage was introduced by combining the input and first hidden layer units in a single module. Further, a time-division multiplexing scheme was implemented for carrying out product computations taking advantage of the built-in DSP cores. In both implementations, the floating point data type representation normally used in a PC has been changed to a more efficient one based on a fixed point scheme, reducing system memory variable usage and leading to an increase in computation speed. The results show that the modifications proposed produced a clear increase in computation speed in comparison to the standard PC-based implementation, demonstrating the usefulness of the intrinsic parallelism of FPGAs in neurocomputational tasks and the suitability of both implementations of the algorithm for its application to real world problems.**

Keywords: Hardware implementation, FPGA, Microcontrollers, Supervised learning, embedded systems.

I. Introduction

The backpropagation algorithm (BP) is the most used learning procedure for training multilayer neural networks architectures. Even if the algorithm was originally proposed by Werbos in 1974 [1], it was not until 1986 that it become popularized through the work of Rumelhart et al. [2]. The BP algorithm is a gradient descent based method that minimizes the error between target and actual network outputs, computing the derivatives of the error in an efficient way [3], [4], [5]. As a gradient descent algorithm the search for a solution can get stuck in a local minima but in practice the algorithm is quite efficient, and as so it has been applied to a wide range of areas from pattern recognition [6], medical diagnosis [7], stock market prediction [8], etc. Real-time applications require extra computational resources [9], involving in some cases also energy consumption restrictions, and thus the use of embedded (dedicated) systems [10] or low power consumption devices are needed, as in those cases a PC might not be the most adequate device for executing neural network models.

The authors are within the Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Campus de Teatinos S/N, 29071, Málaga, SPAIN. Corresponding author: L. Franco, e-mail: lfranco@lcc.uma.es, Tel.: +34-952-133304, Fax: +34-952-131397.

Field Programmable Gate Arrays (FPGA) are reconfigurable hardware devices that can be reprogrammed to implement different combinational and sequential logic created with the aim of prototyping digital circuits, as they offer flexibility and speed. In recent years, the advance in technology have permitted to construct FPGAs with considerable large amounts of processing power and memory storage, and as so they have been applied in several domains (telecommunications, robotics, pattern recognition tasks, infrastructure monitoring, etc.) [11], [12], [13]. In particular FPGAs seem quite suitable for neural network implementations as they are intrinsically parallel devices as is the processing of information in neural network models. Several studies have analyzed the implementation of neural networks models in FPGAs [14], [15], [16], [17], [18], [19], but it is worth noting the difference between *off* and *on* chip implementations. In off-chip learning implementations [20], [21] the training of the neural network model is usually performed externally in a personal computer (PC), and only the synaptic weights are transmitted to the FPGA that acts as a hardware accelerator, while on-chip learning implementations includes both training and execution phases of the algorithm [22], [23], [18]. Existing specific implementations of the artificial neural network backpropagation algorithm in FPGA boards include the works of [24], [25], [26], noting that despite recent advances on the computational power of these boards, still the size of the neural architectures that can be implemented is quite limited. FPGA boards are predominantly programmed using hardware description languages such as VHDL (VHSIC Hardware Description Language) or Verilog and programming them is usually very time consuming.

Apart from FPGAs, other devices very much used in neural network applications are microcontrollers, which are small and low-cost computers built on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals built for dealing with specific tasks. These devices are commonly used in sensor nodes (widely used in wireless sensor network (WSN) [27]) usually under environmental changing conditions, because they are an economic, small and flexible solutions to interpret signals from various sensors and take a decision according to the inputs received [28], [29], [30], [31], [32], [19]. An advantage of microcontrollers is that they can be easily programmed using standard programming languages such as C, C++, Java, etc., while their main limitations are memory size and computing speed.

In the present work, we have implemented the backpropagation algorithm in a VIRTEX-5 XC5VLX110T FPGA and

an Arduino Due microcontroller. Our aim was two fold: first, to obtain efficient implementations on both types of devices that permit its practical application in real life problems and second to compare the efficiency between them and to a standard PC based implementation. The organization of the present work is as follows: next section includes details about the BP algorithm. The FPGA implementation is described in Section III, which contains four parts: the first three subsections describe each one of the three blocks used for the algorithm implementation, while the fourth subsection deals with specific implementation details. Section IV contains the microcontroller implementation of the algorithm followed by the Results section that presents results of both implementations on a set of benchmark functions, together with a detailed analysis of the computational costs involved (number of cycles and execution times) and of the general functioning of the algorithm. The work finishes with the discussion and conclusions.

II. The Backpropagation algorithm

The backpropagation algorithm is a supervised learning method for training multilayer artificial neural networks, and even if the algorithm is very well known, we summarize in this section the main equations in relationship to the implementation of the backpropagation algorithm, as they are important in order to understand the current work.

Let's consider a neural network architecture comprising several hidden layers. If we consider the neurons belonging to a hidden or output layer, the activation of these units, denoted by y_i, can be written as:

$$y_i = g\left(\sum_{j=1}^{L} w_{ij} \cdot s_j\right) = g(h)\ , \tag{1}$$

where w_{ij} are the synaptic weights between neuron i in the current layer and the neurons of the previous layer with activation s_j. In the previous equation, we have introduced h as the synaptic potential of a neuron. g is a sigmoid activation function given by:

$$g(x) = \frac{1}{1 - e^{-\beta x}} \tag{2}$$

The objective of the BP supervised learning algorithm is to minimize the difference between given outputs (targets) for a set of input data and the output of the network. This error depends on the values of the synaptic weights, and so these should be adjusted in order to minimize the error. The error function computed for all output neurons can be defined as:

$$E = \frac{1}{2}\sum_{k=1}^{p}\sum_{i=1}^{M}(z_i(k) - y_i(k))^2, \tag{3}$$

where the first sum is on the p patterns of the data set and the second sum is on the M output neurons. $z_i(k)$ is the target value for output neuron i for pattern k, and $y_i(k)$ is the corresponding response output of the network. By using the method of *gradient descent*, the BP attempts to minimize this error in an iterative process by updating the synaptic weights upon the presentation of a given pattern. The synaptic weights between two last layers of neurons are updated as:

$$\Delta w_{ij}(k) = -\eta\frac{\partial E}{\partial w_{ij}(k)} = \eta[z_i(k) - y_i(k)]g_i'(h_i)s_j(k), \tag{4}$$

where η is the learning rate that has to be set in advance (a parameter of the algorithm), g' is the derivative of the sigmoid function and h is the synaptic potential previously defined, while the rest of the weights are modified according to similar equations by the introduction of a set of values called the "deltas" (δ), that propagate the error form the last layer into the inner ones.

Training and validation processes: The training procedure is executed a certain number of times (epochs) using the training patterns. In one epoch, the training patterns are all presented once in random ordering, adjusting the synaptic weights in an on-line manner. A well known and severe problem affecting all predictive algorithms is the problem of overfitting, caused by an overspecialization of the training procedure on the training set of patterns [33]. In order to alleviate this effect, one straightforward alternative is to split the set of available training patterns in training, validation and test sets. The training set will then be used to adjust the synaptic weights according to Eq. 4, while the validation set is used to control overfitting effects, storing in memory the values of the synaptic weights that have so far led to the lowest validation error, so when the training procedure ends, the algorithm returns the stored set of weights. The test set is used to estimate the performance of the algorithm in unseen data patterns.

III. FPGA implementation of the BP algorithm

FPGAs [34] are reprogrammable silicon circuits, using prebuilt logic blocks and modifiable routing resources that can be configured to implement custom hardware. Besides the fact that FPGAs can be completely reconfigured allowing to change its behavior almost instantaneously by loading a new circuitry configuration, they can be also used as hardware accelerators, in particular for neural based applications given their intrinsic parallel computational capabilities. FPGAs are usually programmed using a hardware description language (VHDL). For the current implementation we used the Virtex-5 OpenSPARC Evaluation Platform (ML509) that includes a Xilinx Virtex-5 XC5VLX110T FPGA. The board was programmed using the "Xilinx ISE Design Suite 12.4" environment within the "ISim M.81d" simulator. Fig. 1 shows a picture of a Virtex-5 OpenSPARC board, and Table I shows some characteristics of the Virtex-5 XC5VLX110T FPGA, indicating its main logic resources. The table indicates for the mentioned board the number of slice registers, Look-Up Tables (LUTs), Bonded Input-Output Banks (IOB), Block RAM (BR) and DSP48 cores. Given that every computation in the FPGA has to be defined from first principles, they usually contain DSPs for helping to perform certain operations. A Digital Signal Processor, or DSP, is a specialized microprocessor that has an architecture optimized for the fast operational needs of digital signal processing. A Digital Signal Processor (DSP) process

Fig. 1. Picture of a Virtex-5 OpenSPARC Platform used for the implementation of the C-Mantec algorithm.

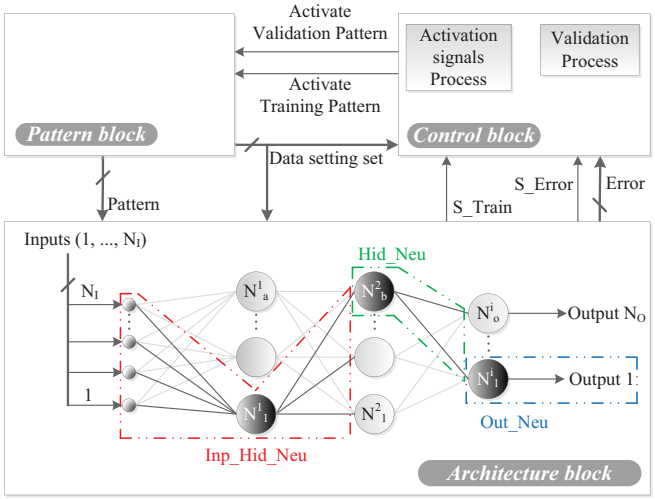

Fig. 2. Scheme of the FPGA design where control, pattern and architecture blocks are shown together with the information exchanged between them.

data in real time, making it ideal for applications that can not tolerate delays [35], [36].

TABLE I
MAIN SPECIFICATIONS OF THE VIRTEX-5 XC5VLX110T FPGA RELATED TO ITS AVAILABLE SLICE LOGIC.

Device	Slice Registers	Slice LUTs	Bonded IOBs	BR	DSP48
Virtex-5 XC5VLX110T	69,120	69,120	34	148	64

The FPGA implementation of the BP algorithm was carried out using three blocks: control, pattern and architecture blocks. The control block organizes the whole information process by sending and processing the information from the architecture and pattern blocks. The pattern block manages the exchange of information between the PC and FPGA for reading the set of patterns to be stored in blocks RAM, and also is used to send a given pattern to the architecture block, that it will be in charge of the training process. Circuit computations have been programmed using fixed point arithmetic which is the standard way to work with FPGA boards. Floating point operations can be codified in a FPGA but they tend to be inefficient in comparison to fixed point representation [25]. We describe below the organization of each one of the three blocks in separate subsections, followed by a fourth subsection that comments on specific implementations details. Fig. 2 shows a diagram of the FPGA design, where the three blocks used are shown together with the information that is exchanged between them.

A. Pattern block

The pattern block manages the data exchange between the PC and the FPGA board through the serial communication RS-232 port of the device. This port has been used because it can be easily implemented in VHDL and ported to other architectures.

To start the process, the user sends the set of parameters of the algorithm and the training patterns. The set of parameters specifies the number of training ($\#Train$) and validation patterns ($\#Valid$), the number of neurons in each layer ($\#N_i$), the number of epochs ($\#Epoch$), and the learning rate value η. The training and validation data sets are stored in the distributed-RAM block of the FPGA, storing the training set in the first positions and the validation in the following ones. Two bytes (1 byte = 8 bits) of memory are used for representing each attribute and each class of a pattern, and thus the total occupied memory of the data set is defined by the equation:

$$\#bytes = 2 \cdot (N_I + N_O) \cdot (\#Valid + \#Train), \quad (5)$$

where N_I is the number of inputs (attributes) and N_O is the number of output classes.

During the execution of the algorithm the pattern block might receive two different signals from the control block in order to send a random training or validation pattern. To avoid training several times a given pattern, the memory position of the last sent pattern is switched with the one corresponding to the final eligible memory position, while the number of eligible memory positions is reduced by 1. This action is repeated until the eligible memory is null, finishing the epoch at this moment and starting a following one. The same process is applied for both training and validation sets independently.

B. Control block

The control block organizes the whole information flow process within the FPGA board by sending and processing the information from the architecture and pattern blocks. The structure of this block is organized into two main processes: i) First, the main function of this block is to control two activation signals that indicate whether a training or a validation pattern should be sent to the architecture block. In order to perform this action the control block receives a signal value from the pattern block that indicates the total number of training (#train) and validation (#val) patterns set for the whole learning procedure. ii) The secondary process of this block is to execute the validation process, included with the aim of avoiding overfitting effects. In essence, this process computes an error value using the validation set of patterns to store the synaptic weight values that have led to the smallest validation error thus far as the training of the network proceeds. The

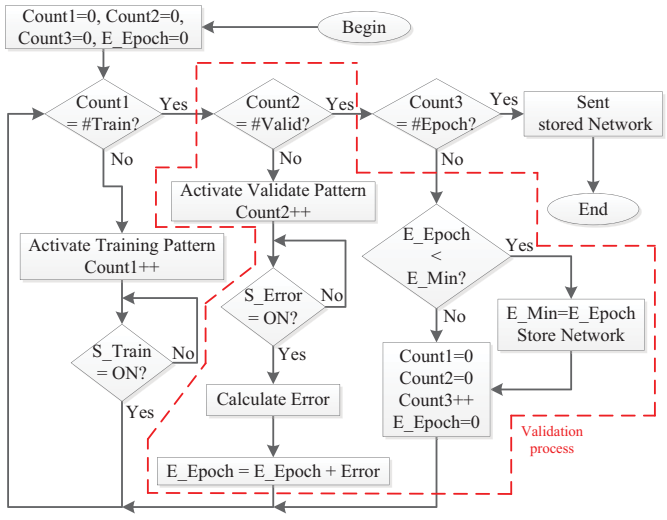

Fig. 3. Flowchart of the FPGA control block operations. The region inside the dashed line corresponds to the validation procedure implemented.

implementation of the whole validation process in the FPGA is detailed in section III-B.

When the computations starts, the set of patterns are loaded into the pattern block that sends a signal to the control block in order to start the execution of the algorithm. Fig. 3 shows a flowchart of the control block operations. At the beginning of the process, a set of counters related to the number of training patterns, number of validation patterns and number of epochs are initialized to zero. If the number of actual training patterns has not reached the value # Train (set by the user), the training procedure starts by sending a signal to the pattern block indicating that a random chosen training pattern should be sent to the architecture block. The architecture block will then train the network, sending back a signal (S_Train) to the control block when the training of this pattern finishes, increasing the trained pattern counter *Count1*. When this value gets equal to the total number of training patterns, then the validation process start (this step is described in detail below). After the validation process, an epoch counter is used for checking whether the whole training-validation procedure should continue or not, as the previous steps are repeated until the maximum number of epochs (# Epoch) is reached.

Validation process: The validation process, included to prevent overfitting problems, is executed after finishing a training epoch. This process requires the storing of the lowest validation error obtained so far (as the training procedure advances) together with the synaptic weights that led to this error. When this procedure is activated at the end of a training epoch, it computes the mean square error (MSE) for the validation set, and if this value is lower than the stored one, then it is saved together with all synaptic weights in a block RAM using a FIFO procedure. As the neurons included in the network architecture are indexed, the control block demands sequentially the set of synaptic weights associated to each neuron so they can be stored in a single FIFO RAM while preventing memory collision problems. The flowchart of the validation process is included in Fig. 3.

TABLE II

NUMBER OF EMPLOYED RESOURCES FOR THE REALIZATION OF EACH TYPE OF POSSIBLE NEURON.

Resource	Resource for type of Neuron			
	Input	InpHid	Hidden	Output
LUTs	524	1126	923	502
Register	254	396	391	255
DSP48	1	1	1	1
Block RAM	1	1	1	1

C. Architecture block

The architecture block is in charge of the physical implementation of the neural network architecture. The number of layers and the maximum number of neurons in each layer has to be predefined by the user before the execution of the algorithm.

Previous works [37], [38], [23] use three different types of neurons, corresponding to input, hidden and output layer neurons, as they all have different functionalities. Nevertheless, we decide to use a different approach in order to optimize further the FPGA resources, and thus the proposed implementation eliminate input layer neurons as they are included together with the first hidden layer neurons in a new module that we name input-hidden neurons. The definition of this new type of module is possible, mainly because the input layer neurons do not process the information as they simple act as input to the network. The implementation of the neurons consists of a group of LUTs (LookUp Table) with a specific functionality of the backpropagation algorithm. The input-hidden neurons manage the input data, the synaptic weights between the input and first hidden layer, the output computation of the first hidden neuron and also the synaptic weights connecting to the output or to a further hidden layer. The hidden layer modules (in case they are included) compute the neuron activation and store the values of the synaptic weights connecting to further neurons. Finally, the output modules evaluate the value of the output units in order to compute the error of the presented pattern. A scheme of a two hidden layer neural network is shown in Fig. 2 where the three different types of modules are indicated. The table II shows the number of employed resources by each type of neuron with a word size of 32 bits, 16 for the integer part (N_1) and 16 to the decimal part (N_2). The election of the word size is described in section III-D.1

To specify a given architecture the number of active neurons in each layer should be sent to the FPGA as part of the setting data set. However the system are composed of a determined number of neurons and layers, so that the architecture of the network and the maximum number of neurons are predetermined and delimited by the resources of the device. The novel layer (the first layer blocks), that is employed in this implementation, reduces the required resources for any architecture. Table III shows board resources needed by the proposed method (Prop.) for different size neural architectures in comparison to a conventional implementation (Conv.) , and also to the published results in Ref. [38] (Gomp.).

TABLE III

NUMBER OF EMPLOYED RESOURCES FOR THE REALIZATION OF EACH TYPE OF POSSIBLE NEURON.

Architecture	*Type*	*Resource*			
		LUTs	Regis.	BR	DSP
$10-3-1$	Conv.	11044	6676	79	15
	Gomp.	8043	2243	–	70
	Prop.	6413	4151	69	5
$10-6-3-2$	Conv.	17084	9277	86	22
	Gomp.	20021	5342	–	169
	Prop.	13062	6767	76	12
$10-50-1$	Conv.	54425	25053	126	62
	Prop.	59335	22763	116	52
$30-30-10-2$	Conv.	56177	26478	137	73
	Prop.	46547	19008	107	43
$50-10-10-5$	Conv.	49703	24503	140	76
	Prop.	25533	11853	90	26
$60-15-10-5$	Conv.	59558	28998	155	91
	Prop.	33163	13833	95	31
Reduction mean		25.8%	35.2%	23.5%	50.1%

D. Implementation details

We describe below details related to the choice of synaptic weights precision, for carrying out the implementation of products, and about the computation of the sigmoid function used as the transfer function of the neurons.

1) Synaptic weights precision: The representation of the synaptic weights can be chosen according to the available resources, taking into account that requiring higher accuracy may need a larger representation, leading also to an increase in the number of LUTs per neuron (consequently reducing the maximum number of available neurons), and a decrease in the maximum operation frequency of the board. On the other hand, synaptic weights accuracy cannot be much reduced, as a proper operation of the backpropagation algorithm requires a certain level of precision [39], [25]. A synaptic weight is represented by a bit array with integer and fractional parts of length N_1 and N_2. N_1 determines the minimum and maximum values that can be represented $-2^{(N_1-1)}$ to $2^{(N_1-1)}$ while N_2 defines the accuracy $2^{(-N_2)}$. The number of bits needed to represent all possible discrete values within a certain range of positive values depends on the difference between the maximum and minimum of the interval, and can be obtained from the following equation:

$$\#bits = log_2((1 + max(w_{ij}))/(min(w_{ij}))). \quad (6)$$

Table IV shows the number of LUTs needed to represent each type of neuron modules as a function of the number of bits used for representing the synaptic weights. N_1 and N_2 indicate the integer and fractional parts of the synaptic weight representation. The last column shows the maximum whole system operation frequency allowed for the chosen representation.

TABLE IV

NUMBER OF LUTS PER EACH TYPE OF POSSIBLE NEURON MODULE ACCORDING TO THE NUMBER OF BITS USED FOR REPRESENTING THE SYNAPTIC WEIGHTS.

N_1	N_2	LUTs per type of Neuron				f_{max}
		Input	InpHid	Hidden	Output	(MHz)
8	8	347	723	592	343	191.2
8	12	390	871	671	409	186.7
8	16	433	1028	763	448	183.7
12	12	430	913	740	425	183.7
12	16	476	1031	859	475	180.6
16	16	524	1126	923	502	177.3

2) Product implementation: The execution of the backpropagation algorithm requires the computation of several products, mainly between neuron activations and synaptic weights values (see Eqs. 1 - 4), and so an efficient implementation of this operation is crucial in order to optimize board resources (affecting the number of LUTs per neuron required and the operation frequency of the FPGA). Multipliers can be implemented by shifters and adders, following the approach presented in [40] or by available specific DSP cores in the FPGA. The number of required LUTs for the first type of implementation is proportional to the bit size of the input data, as for example, for two vectors with N_a and N_b bits length respectively, the product requires $N_a \times N_b$ LUTs while for the second type of implementation, one DSP for each neuron is needed (clearly this puts a limitation in the maximum number of neurons in the system). We decided to use the DSP based strategy as the board frequency operation can be up to four times faster, as we measured the operation of the board without using the DSPs.

For an efficient use of the DSP resources, we implemented a time-division multiplexing scheme, using only one multiplier block per neuron and thus performing sequentially the computation of several products. This time-division multiplexing scheme is shown in Fig. 4. The neuron module comprises a first block (we named it neuron block) that includes several LUTs, registers and one BR, while a second block consists just of a DSP. Both blocks are synchronous but the DSP uses a frequency two times larger than the one used by the neuron block, so that a product operation could be completed in one operation cycle of the FPGA.

3) Implementation of the Sigmoid function: The operation of the neurons involves the computation of sigmoid functions for obtaining the output value of the neurons. As we are using a fixed point representation (as it is more efficient than the floating point one), then the computation of the sigmoid function needs the use of an approximation. A look-up table containing equispaced values of the function has been created to obtain certain number of output values. Neverthless, as high precision values are needed for the correct execution of the algorithm, the computation of the function approximation was further complemented by a linear interpolation procedure using the two adjacent tabulated values (lower and larger) with respect to the input. Storing table values requires large amounts of memory, and as one table per neuron is needed, this number should be optimized. 64 tabulated values were used, as this ensures the obtention of absolute errors lower than 10^{-3}.

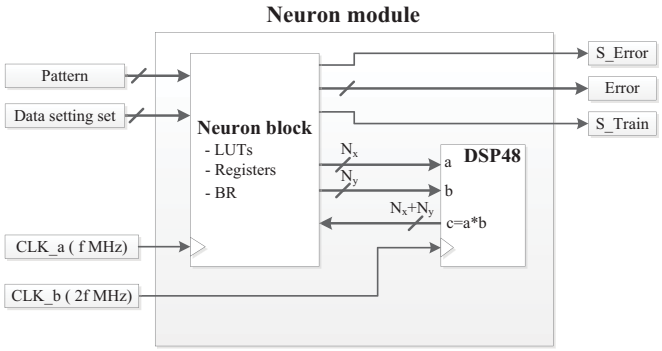

Fig. 4. Scheme of an implemented neuron that uses a time-division multiplexed strategy to execute several multiplications. Neuron and DSP blocks are synchronous but the DSP uses a frequency two times larger than the used by the neuron block, so that a product operation could be completed in one operation cycle of the FPGA

These values start from -8 to 8 with 0.25 increasing steps (values lower than -8 and higher than 8 were set to 0 and 1 respectively). Fig. 5 (Top) plots real, tabulated and interpolated values for the sigmoid function, with the inset plot showing an enlargement of a portion of the curve. Fig. 5 (Middle and Bottom graphs) shows absolute and relative errors involved in the computation of the sigmoid function in the range from -10 to 10.

IV. Microcontroller (μC) implementation

We have further implemented the backpropagation algorithm in an Arduino DUE microcontroller. We describe below in several subsections all the details of the implementation process, highlighting the results of a comparison carried out between using a fixed point representation or a floating point one.

A. The Arduino board

Arduino is a single-board microcontroller designed to make the process of using electronics in multidisciplinary projects more accessible [41]. The hardware consists of a simple open source board designed around an 32-bit Atmel ARM core microcontroller, and the software includes a standard programming language compiler that runs in a standard PC and a boot loader for loading the compiled code on the microcontroller. Arduino is a descendant of the open-source *Wiring* platform and is programmed using a Wiring-based language (syntax and libraries), similar to C++ with some slight simplifications and modifications, and a processing-based integrated development environment.

The Arduino DUE is based on the SAM3X8E ARM Cortex-M3 CPU [42], and it has 54 digital input/output pins (of which 12 can be used as PWM outputs), 12 analog inputs, 4 UARTs (hardware serial ports), a 84 MHz clock, an USB OTG capable connection, 2 DAC (digital to analog), and a reset and erase buttons. The SAM3X has 512 KB (2 blocks of 256 KB) of flash memory for storing code, it also comes with a preburned bootloader that is stored in a dedicated ROM memory. The available SRAM amounts to 96 KB in two contiguous banks

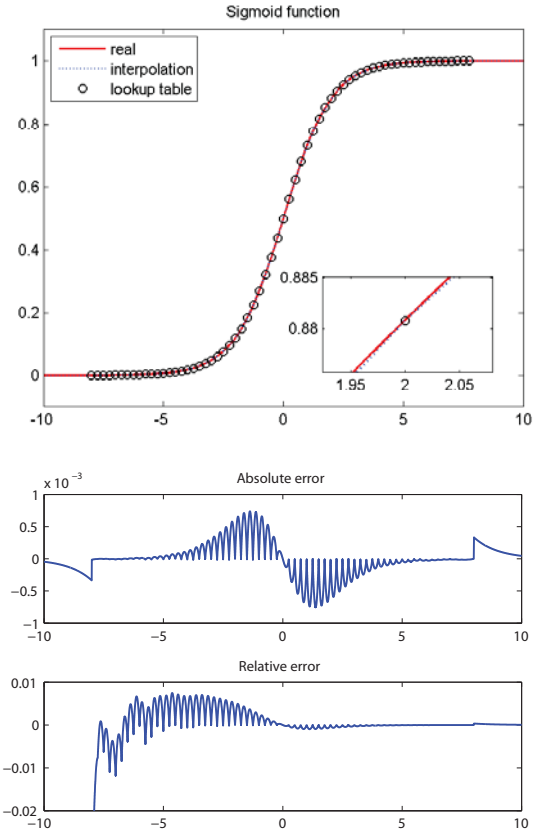

Fig. 5. The computation of the exponential function and its approximation based on a lookup table plus a linear interpolation scheme (Top graph). Absolute (Middle graph) and relative errors (Bottom graph) committed in the approximation of the function (See text for more details).

Fig. 6. Picture of an Arduino DUE board used for the implementation of the C-Mantec algorithm.

of 64 and 32 KB. A picture of the Arduino DUE board is shown in Fig. 6.

B. Learning and execution phases of the algorithm

The implementation of the backpropagation neural network learning model comprises two phases: the learning phase where the synaptic weights of the chosen architecture are adjusted according to the set of patterns presented to the network, and the execution phase in which the microcontroller outputs a signal in response to sensed input data according to the model previously adjusted.

The learning phase has been divided into two different processes: loading of input patterns and neural network training. Data can be loaded into the microcontroller memory on-line by I/O pins or by a serial communication USB port (this last option was the used one in the current implementation for simplicity reasons), but in both cases, the patterns have to be stored into the memory board because the learning process works in cycles in which patterns are repeatedly used. The microcontroller memory has been further divided in two parts, one of 64 KB for the loading process, which is used for storing the inputs patterns, and a second part comprising 32 KB of memory to be used for system variables. This second part of memory is used for storing the value of the synaptic weights, and all other variables of the algorithm, like the activation value of the neurons, the deltas, the learning rate, etc. The neural network training phase consists of the backpropagation algorithm itself that was implemented with a validation phase to avoid overfitting effects. Once the training phase finishes the synaptic are stored in the memory block of 64 KB.

The execution phase is programmed to be carried out from external data, as usually the microprocessor will be used as an independent sensor. In this mode, a pattern would be read from a sensor connected to one of the ports of the board, and the previously trained model will be executed to obtain the neural network output.

C. Pattern storage

The number of bits used for representing each of the inputs of a pattern has to be decided in advance of the implementation. In the present case, 8 bits have been used to represent each variable, taking into account that these input values have to be previously normalized between 0 and 255. Using this representation for the patterns, the maximum number of samples that can be stored in a 64KB memory is given by the following equation:

$$N_P \cdot (N_I + N_O) \leq 65536\ , \tag{7}$$

where N_P is the number of patterns, N_I is the number of input variables (the dimension of the patterns) and N_O is the number of outputs of the patterns that determines the number of output of the neural architecture.

D. Data type representation

The microcontrollers are devices with limited computing power so in order to speed up the learning process, we decided to utilize a fixed point data representation. We note that floating point is the usual data type representation used in this kind of device but this representation is not always the most efficiency. This paradigm shift involves important changes in the way the BP algorithm is programmed but in return offers a faster learning process and a smaller size representation of variables. The following list give details of the type of representation used for variables related to the implementation of the neurons:

- deltas (δ): 2 bytes *integer*.
- Synaptic weights(w): 2 bytes *integer*.
- Outputs (y): 2 bytes *integer*.

The previous choice for the representation of the neural network related variables affects the maximum network size that can be utilized. The total number of neurons (N_N) in the whole architecture can be expressed as the sum of the number of neurons in each layer ($N_N = N_{N1} + N_{N2} + ...$), so the maximum number of neurons used in each layer should verify the following constraint:

$$\begin{aligned} 2 \cdot (N_{N1} + N_{N2} + ...) + \\ +2 \cdot (N_I \cdot N_{N1} + N_{N1} \cdot N_{N2} + N_{N2} \cdot N_{N3} + ...) + \\ +2 \cdot (N_{N1} + N_{N2} + ...) \leqslant 32768, \end{aligned} \tag{8}$$

where the first term relates to the variable storage space for the δs, the second term account for the synaptic weights between all the layers of neurons, and the last term is related to the output value of the neurons in each layer. (N_{N1} represents the number of neurons in the first layer, N_{N2} of the second layer and so on, while N_I is used for the number of inputs).

From the 2 bytes used for representing the variables of the system, 10 bits were used for the decimal part, and the remaining 6 for the integer part, and thus the value of the system variables ranges between 32 and -32. A special case was the representation used for the variable computing the summation of the synaptic potential, because in order to avoid saturation effects a 4 bytes representation was used.

E. Computation of the Sigmoid function

The computation of the sigmoid function can be implemented using the specific ALU (arithmetic and logic unit) for resolving the exponential function. The previous computation involves two different variable conversions, the first related to the input values of the sigmoid (casting from integer to a floating point representation), and the second conversion is done to the output value in a reverse casting. The computational cost of implementing the previous method is high, with an approximate time of operation of $62\mu s$, and thus an alternative method based on a lookup table plus linear interpolation of adjacent values, similar to the one used in the FPGA implementation, was chosen. The method is explained in detail in section III-D.3, and in this case the computation time employed is reduced to $2\mu s$ (97% reduction in comparison to the first mentioned method).

F. Fixed point vs floating point representation comparison

Figs. 7 Top, middle and bottom show the number of times that the implementation based on integer data type is faster in comparison to the floating point representation as a function of the number of neurons in the different layers of the neural architecture. An architecture with only one hidden layer has been used to compute the values represented in the figure, where N_N represents the number of neurons, N_I the number of inputs and N_O is used for the number of outputs. The Fig. 7 top shows the comparison for variable values of N_N and N_I (keeping fixed N_O equal to 1), Fig. 7 middle is computed for different values of N_N and N_O with N_I equals to 10, and finally Fig. 7 bottom represents the values obtained as a function of N_I and N_O for N_N equals to 5.

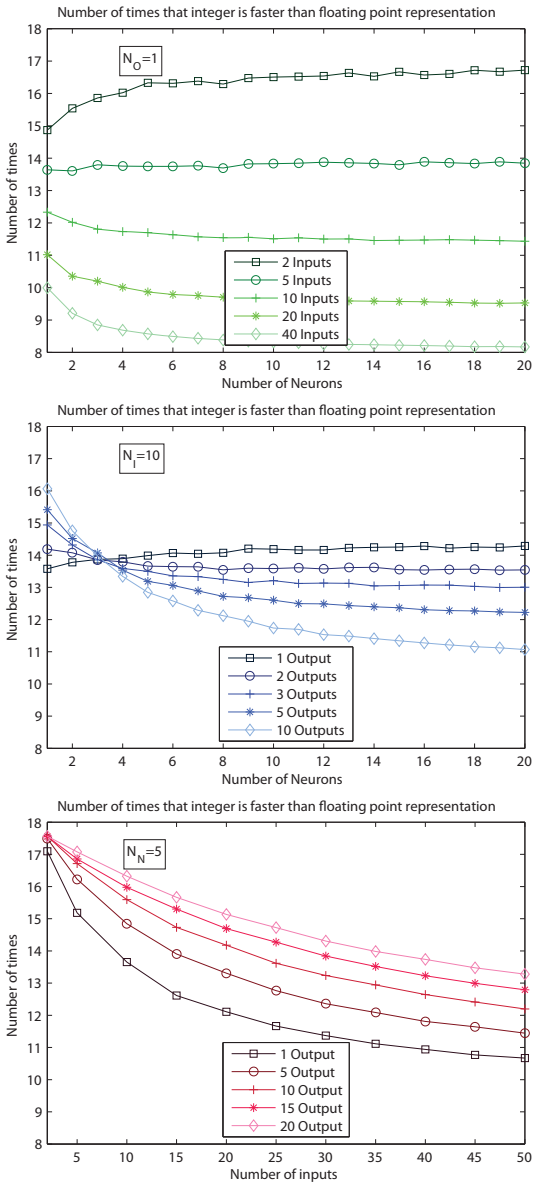

Fig. 7. Number of times that the fixed point representation used in the microcontroller is faster than the floating point one for different number of neurons in the layers of a one hidden layer architecture (see text for more details).

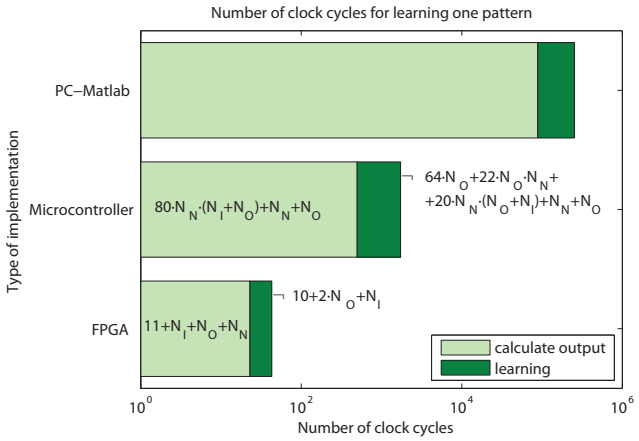

Fig. 8. Number of clock cycles involved in the learning process of a single pattern according to the type of implementation used for the case of a $4-5-3$ neural network architecture.

V. Results

We analyze in this section several aspects in relationship to the two implementations of the backpropagation algorithm carried out in a FPGA board and in an Arduino DUE microcontroller, considering also a third implementation of the algorithm in personal computer (PC) for comparison purposes. The PC implementation of the algorithm has been executed under Matlab code and run in an *Intel(R) core (TM) i5-3330 CPU @3.00GHZ* with 16 GB of RAM memory. All three implementations of the algorithm follow the same operation steps and the only evident differences between them are the random number generator used for the initialization of the synaptic weights, the type of data representation used in each case, and the computation of the sigmoid function. The FPGA implementation uses a LFSR random number generation routine, while the microcontroller and the Matlab code use the built-in *random* and *randn* functions respectively.

Fig. 8 shows the estimated number of clock cycles that each implementation employs for the learning of an input pattern for a single hidden layer neural network architecture with a 4-5-3 structure. Each bar in the graph is further divided in two parts: computation of the output of the network in response to the input pattern (clear part of the bar), and cycles involved in the modification of the synaptic weights including the backward phase of the BP algorithm (dark part of the bar) (Note the logarithmic scale of the graph). The number of clock cycles in the FPGA implementation has been measured straightforwardly using the ISIM simulator. The estimation of the number of cycles for the microcontroller has been done by computing the time that takes the computation of each instruction of the algorithm, multiplying this value by the clock frequency of the microcontroller (80 MHZ) and then summing over all the instructions involved in the algorithm. The number of cycles used in the PC-Matlab implementation cannot be computed directly and has been estimated by measuring the total computation time and multiplying this value by the CPU frecuency (3 GHZ), even if a strict evaluation of the number of cycles should involve taking into account other factors like instructions of the operative system, libraries, etc. The method used in the FPGA and microcontroller implementations permits to estimate the number of operation cycles as a function of the number of neurons in one hidden layer architecture (N_I, N_N, and N_O) and is represented inside the bars in Fig. 8. It is worth noting that even if in principle microcontroller and computer codes used are quite similar, there are differences regarding the implementations as the data representation used is different (fixed and floating point respectively), and the computation of the sigmoid function values is done in a different way (tabulated values plus interpolation for the microcontroller vs ALU in the case of a computer). To test the correct implementation of the BP algorithm in the FPGA and microcontroller devices, we tested the training, validation and test errors on a set of benchmark problems from the UCI

database [43] frequently used in the literature. Table V shows the accuracy (generalization ability) values obtained for the three implementations of the algorithm for eleven benchmark problems. The first three columns indicate the data set name, number of inputs and outputs respectively, while the last three columns shows the accuracy obtained by using neural network architectures with 5 neurons in the single hidden layer, as the number of inputs and outputs is determined by the problems themselves. We have not optimized the neural architecture for each problem as our aim is to demonstrate the correct implementation of the algorithm and not to obtain optimal values of prediction accuracy. For carrying out the simulations a training, validation and test sets splitting was used in a 50-20-30 % scheme; in which the validation set was used to find the number of epochs for evaluating the test error, the maximum number of epochs was set to 1000, and the learning rate was equal to 0.2.

We further computed execution times for the whole learning process (training and validation) for the same set of benchmark functions mentioned above, and the results are shown in Fig. 9 for the FPGA, microcontroller and PC versions of the backpropagation algorithm (note the logarithmic scale used in the Y-axis of the figure). We also perform an analysis to see how FPGA and microcontroller performances behaves in comparison to the PC implementation as the complexity of the functions grow. We have used the execution time needed in the PC implementation $time_{PC}$ as an estimation for the complexity of the benchmark functions, to obtain that the number of times that the FPGA implementation is faster than the PC $(\#times_{FPGA})$ grows as $\#times_{FPGA} = 94 + 2 \cdot time_{PC}$ (Pearson correlation coefficient equals to 0.753), while the analysis for the microcontroller shows not significant performance increase: $\#times_{\mu C} = 1.6 + 0.0083 \cdot time_{PC}$ (Pearson correlation coefficient equals to 0.25).

TABLE V
ACCURACY

Function	$\#I$	$\#O$	PC	FPGA	μC
Diabetes	8	2	78.31	79.35	79.13
Cancer	9	2	95.63	95.73	95.60
Statlog (Heart)	13	2	78.52	78.27	78.26
Climate	18	2	93.27	94.14	94.51
Ionosphere	34	2	88.21	87.57	87.14
HeartC	35	2	78.80	80.11	80.22
Iris	4	3	92.22	92.77	90.89
Balance Scale	4	3	87.93	87.82	87.61
Seeds	7	3	97.62	96.51	96.66
Wine	13	3	88.89	87.04	86.67
Glass	10	6	93.85	91.54	92.31
Average			88.48	88.26	88.09

Fig. 10 shows the Root Mean Square Error (RMSE) obtained for training and validation processes when the BP algorithm is implemented in the FPGA, μC and PC as a function of the number of epochs for the Iris data set using a 4-5-3 architecture (similar results were observed for all data sets). It can be seen that the training error always decreases as training advances being lower for the PC implementation than for other two, indicating that a more precise representation (32 bit floating point) helps to adjust the synaptic weights during

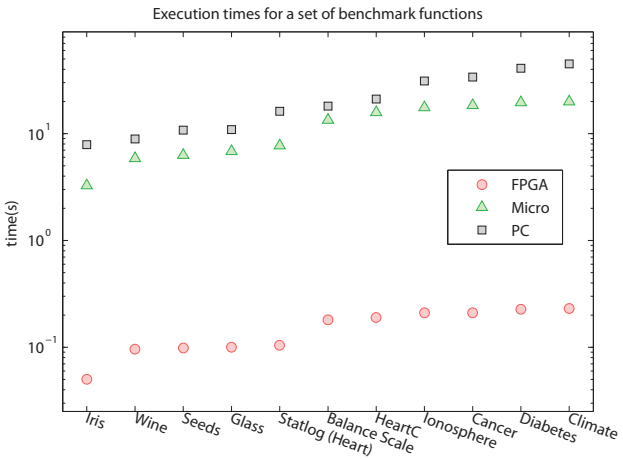

Fig. 9. Execution times (in Seconds) for the whole learning process (training and validation) for the set of benchmark functions used for verifying the correct implementation of the algorithm in the FPGA and Microcontroller using neural network architectures with a single hidden layer (see text more details).

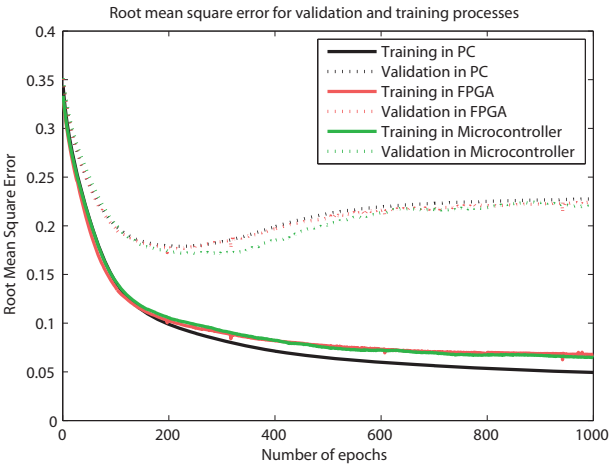

Fig. 10. Root mean square error for the training and validation process when learning the Iris data set using a 4-5-3 architecture for the three different implementations used.

training. However for the validation RMSE, the three curves are quite similar noting also that the values increase at certain point of the process (approximately at 200 epochs), indicating overfitting effects and justifying the use of a validation set. As in general the interest on the application of supervised neural networks to practical problems is related to prediction, validation and test errors are the important features and thus the results confirm that the representation used for the FPGA and μC (16 bits fixed point) is adequate.

VI. Discussion and Conclusions

The backpropagation algorithm has been successfully implemented in a FPGA board and in an Arduino microcontroller, in a learning paradigm that includes a validation scheme in order to prevent overfitting effects. The implementation of the algorithm in the FPGA board involved several challenges as hardware programming is a totally different

paradigm approach in comparison to standard software programming, and as such, we have first introduced a new neuron representation that permits to increase the efficiency of the traditional implementation, obtaining an average reduction of 25.8% in the total number of LUTs needed to implement different architectures, as shown in Table III. Further improvements are related to a time-division multiplexing scheme for carrying out product computations taking advantage of the FPGA DSP built-in cores, and a lookup table plus linear interpolation scheme for computing the transfer function of the neurons (the Sigmoid function). At the time of a real implementation, the limitation in terms of the size of the neural network architectures that can be simulated would come from the specific FPGA board used, and this analysis can be done from the results shown in table II. In our case in which we are using a Xilinx Virtex V XC5VLX110T board the main limitation comes from the number of available DSP cores, as the mentioned board includes 64 DSP cores, and thus this factor limits the maximum number of neurons in the architecture to 63, as an extra core is needed in the validation process. If we compare these new results with previous published works, for example those appearing in [38], we see a significant efficiency increase that will permit the utilization of much larger neural architectures. Nevertheless the efficiency increase for a particular case would depend on the values of the combination of neural network architecture parameters and resources of the FPGA board used.

Considering the microcontroller implementation, the standard floating point data type representation has been changed to a more efficient one based on a fixed point scheme, reducing system memory variable usage and an increase in computation speed, obtaining that the Fixed point representation is approximately 10 times faster than the floating point one (see Fig. 7).

The results shown in Fig. 10 indicate that the representation used for the FPGA and microcontroller was adequate as validation and test errors were similar to the PC implementation of the algorithm. The use of a lower precision representation affects the learning error (it is lower for the PC implementation) but it is not relevant regarding prediction accuracy. We hypotethize that this fact might be related to the effect observed when noise is added both to input and synaptic weight values [44], that instead of having negative effects, it can help to improve generalization by preventing overfitting effects.

An estimation of the computation time (Fig. 9 involved in relationship with the three implementations of the BP algorithm (FPGA, μC and PC) shows the potential advantages of using a FPGA board as a hardware accelerator device for neurocomputing applications, obtaining a speed up of hundred of times with an improvement increasing with the complexity of the problem, highlighting the intrinsic parallelism of the device.

As an overall conclusion, the present work shows the potential advantages of using FPGA boards as hardware accelerator devices for neurocomputing applications giving their intrinsic parallel capabilities, while in relationship to the use of neural networks in microcontrollers we highlight the "on-chip" characteristic of the presented implementation that will permit its use in remote sensors using a stand-alone operation mode.

VII. Acknowledgements

The authors acknowledge support from Junta de Andaluca through grants P10-TIC-5770 and P08-TIC-04026, and from CICYT (Spain) through grant TIN2010-16556 (all including FEDER funds).

References

[1] P. J. Werbos, “Beyond regression: New tools for prediction and analysis in the behavioral sciences,” Ph.D. dissertation, Harvard University, 1974.

[2] D. Rumelhart, G. Hinton, and R. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[3] K. Mehrotra, C. K. Mohan, and S. Ranka, *Elements of Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1997.

[4] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

[5] R. D. Reed and R. J. Marks, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1998.

[6] J. Li, K. Ouazzane, H. Kazemian, and M. Afzal, “Neural network approaches for noisy language modeling,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 24, no. 11, pp. 1773–1784, Nov 2013.

[7] K. Misra, S. Chattopadhyay, and D. Kanhar, “A hybrid expert tool for the diagnosis of depression.” *Journal of medical imaging and health informatics.*, vol. 3, no. 1, pp. 42–47, 2013.

[8] H. Mo and J. Wang, “Volatility degree forecasting of stock market by stochastic time strength neural network.” *Mathematical Problems in Engineering.*, vol. 2013, pp. 1–11, 2013.

[9] G.-B. Huang and C.-K. Siew, “Real-time learning capability of neural networks,” *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 863–878, July 2006.

[10] S.-M. Baek and J.-W. Park, “Hessian matrix estimation in hybrid systems based on an embedded ffnn,” *IEEE Transactions on Neural Networks*, vol. 21, no. 10, pp. 1533–1542, Oct 2010.

[11] E. Monmasson, L. Idkhajine, M. Cirstea, I. Bahri, A. Tisan, and M. Naouar, “Fpgas in industrial control applications,” *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, pp. 224–243, May 2011.

[12] D. Bacon, R. Rabbah, and S. Shukla, “Fpga programming for the masses,” *Queue*, vol. 11, pp. 40–52, 2013.

[13] P. Conmy and I. Bate, “Component-based safety analysis of fpgas,” *IEEE Transactions on Industrial Informatics*, vol. 6, no. 2, pp. 195–205, May 2010.

[14] D. Le Ly and P. Chow, “High-performance reconfigurable hardware architecture for restricted boltzmann machines,” *IEEE Transactions on Neural Networks*, vol. 21, no. 11, pp. 1780–1792, Nov 2010.

[15] Q. N. Le and J.-W. Jeon, “Neural-network-based low-speed-damping controller for stepper motor with an fpga,” *IEEE Transactions on Industrial Electronics*, vol. 57, no. 9, pp. 3167–3180, Sept 2010.

[16] W. Mansour, R. Ayoubi, H. Ziade, R. Velazco, and W. E. Falouh, “An optimal implementation on fpga of a hopfield neural network,” *Advances in Artificial Neural Systems*, vol. 2011, pp. 1–9, 2011.

[17] L.-W. Kim, S. Asaad, and R. Linsker, “A fully pipelined fpga architecture of a factored restricted boltzmann machine artificial neural network,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 1, pp. 5–23, Feb. 2014.

[18] F. Ortega-Zamorano, J. Jerez, and L. Franco, “Fpga implementation of the c-mantec neural network constructive algorithm,” *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1154–1161, May 2014.

[19] F. Ortega-Zamorano, J. M. Jerez, J. L. Subirats, I. Molina, and L. Franco, “Smart sensor/actuator node reprogramming in changing environments using a neural network model,” *Engineering Applications of Artificial Intelligence*, vol. 30, no. 0, pp. 179 – 188, 2014.

[20] 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*, vol. 18, no. 3, pp. 880–888, May 2007.

[21] T. Orlowska-Kowalska and M. Kaminski, "Fpga implementation of the multilayer neural network for the speed estimation of the two-mass drive system," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 3, pp. 436–445, Aug 2011.

[22] A. Dinu, M. Cirstea, and S. Cirstea, "Direct neural-network hardware-implementation algorithm," *IEEE Transactions on Industrial Electronics*, vol. 57, no. 5, pp. 1845–1848, May 2010.

[23] J. Shawash and D. Selviah, "Real-time nonlinear parameter estimation using the levenberg-marquardt algorithm on field programmable gate arrays," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 1, pp. 170–176, Jan 2013.

[24] A. Omondi and J. Rajapakse, *FPGA Implementations of Neural Networks*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[25] A. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing mlp-bp on fpgas: A study," *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp. 240–252, Jan 2007.

[26] A. Gomperts, A. Ukil, and F. Zurfluh, "Implementation of neural network on parameterized fpga." in *AAAI Spring Symposium: Embedded Reasoning*. Stanford University, CA, USA, 2010, pp. 45–51.

[27] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Comput. Netw.*, vol. 52, no. 12, pp. 2292–2330, Aug. 2008.

[28] M. Sayed-Mouchaweh and E. Lughofer, *Learning in non-stationary environments: Methods and Applications*. Springer, New York, 2012.

[29] D. Urda, E. Canete, J. L. Subirats, L. Franco, L. Llopis, and J. M. Jerez, "Energy-efficient reprogramming in wsn using constructive neural networks," *International Journal of Innovative, Computing, Information and Control*, vol. 8, pp. 7561–7578, 2012.

[30] E. E. Canete, J. Chen, R.Luque, and B. Rubio, "Neuralsens: A neural network based framework to allow dynamic adaptation in wireless sensor and actor networks." *J. Network and Computer Applications*, vol. 35, no. 1, pp. 382–393, 2012.

[31] S. Mahmoud, A. Lotfi, and C. Langensiepen, "Behavioural pattern identification and prediction in intelligent environments," *Appl. Soft Comput.*, vol. 13, no. 4, pp. 1813–1822, Apr. 2013.

[32] M. Rassam, A. Zainal, and M. Maarof, "An adaptive and efficient dimension reduction model for multivariate wireless sensor networks applications," *Appl. Soft Comput.*, vol. 13, no. 4, pp. 1978–1996, Apr. 2013.

[33] D. M. Hawkins, "The problem of overfitting," *Journal of Chemical Information and Computer Sciences*, vol. 44, no. 1, pp. 1–12, 2004.

[34] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007.

[35] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, 3rd ed. Springer-Verlag New York, NJ, USA, 2007.

[36] *Virtex-6 FPGA DSP48E1 Slice USer Guide (v1.3)*, Xilinx inc., 2011.

[37] C.-J. Lin and C.-Y. Lee, "Implementation of a neuro-fuzzy network with on-chip learning and its applications," *Expert Syst. Appl.*, vol. 38, no. 1, pp. 673–681, Jan. 2011.

[38] A. Gomperts, A. Ukil, and F. Zurfluh, "Development and implementation of parameterized fpga-based general purpose neural networks for online applications," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 1, pp. 78–89, Feb 2011.

[39] M. Moussa, S. Areibi, and K. Nichols, *On the Arithmetic Precision for Implementing Back-Propagation Networks on FPGA: A Case Study*, A. R. Omondi and J. C. Rajapakse, Eds. Springer US, 2006.

[40] P. P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. John Wiley & Sons, 2006.

[41] J. Oxer and H. Blemings, *Practical Arduino: Cool Projects for Open Source Hardware*. Berkeley, CA, USA: Apress, 2009.

[42] Atmel, "Datasheet atmel sam3x8e arm cortex-m3 cpu," http://www.atmel.com/Images/doc11057.pdf.

[43] U. of California Irvine, "Machine learning repository," http://archive.ics.uci.edu/ml/.

[44] G. An, "The effects of adding noise during backpropagation training on a generalization performance," *Neural Comput.*, vol. 8, no. 3, pp. 643–674, Apr. 1996.

Capítulo 3

Implementación en una FPGA del Algoritmo constructivo de red neuronal C-Mantec

Francisco Ortega-Zamorano, José M. Jerez and Leonardo Franco: FPGA Implementation of the C-Mantec Neural Network Constructive Algorithm. **IEEE Transactions on Industrial Informatics** 10(2): 1154-1161 (2014).

Posición JCR: Q1 (1/102) en Computer science: Interdisciplinary applications.
Q1 (1/43) en Engineering: Industrial.
Factor de impacto: 8,785

Resumen: Se ha realizado la implementación hardware en una FPGA del algoritmo recientemente propuesto de red neuronal constructivo (CoNN: Constructive Neural Networks) C-Mantec que genera arquitecturas muy compactas y con una muy buena capacidad de generalización. Una clara diferencia del algoritmo C-Mantec con respecto a la mayoría de las implementaciones de redes neuronales basadas en el algoritmo Backpropagation es que el algoritmo C-Mantec genera la arquitectura de red automáticamente, añadiendo neuronas a la arquitecture de la red conforme el modelo lo requiera; además, no precisa de capa de salida con lo que resta complejidad a la estructura de la red. Todos los pasos involucrados en la ejecución (incluyendo la fase de aprendizaje) se han descrito con detalle. Realizando un análisis en profundidad de los resultados obtenidos, se observa claramente un incremento en la velocidad de cálculo en comparación con las implementaciones estándares realizadas mediante un ordenador personal, lo que demuestra la utilidad del paralelismo intrínseco de FPGAs en las tareas neurocomputacionales y la idoneidad de la versión de hardware del algoritmo C-Mantec para su aplicación a los problemas del mundo real.

FPGA Implementation of the C-Mantec Neural Network Constructive Algorithm

Francisco Ortega-Zamorano, José M. Jerez, and Leonardo Franco, *Senior Member*, *IEEE*

***Abstract*—Competitive majority network trained by error correction (C-Mantec), a recently proposed constructive neural network algorithm that generates very compact architectures with good generalization capabilities, is implemented in a field programmable gate array (FPGA). A clear difference with most of the existing neural network implementations (most of them based on the use of the backpropagation algorithm) is that the C-Mantec automatically generates an adequate neural architecture while the training of the data is performed. All the steps involved in the implementation, including the on-chip learning phase, are fully described and a deep analysis of the results is carried on using the two sets of benchmark problems. The results show a clear increase in the computation speed in comparison to the standard personal computer (PC)-based implementation, demonstrating the usefulness of the intrinsic parallelism of FPGAs in the neurocomputational tasks and the suitability of the hardware version of the C-Mantec algorithm for its application to real-world problems.**

***Index Terms*—Circuit complexity, constructive neural networks (CoNN), on-chip learning, threshold networks.**

I. Introduction

ARTIFICIAL neural networks (ANNs) are the mathematical models inspired in the functioning of the brain that can be utilized in clustering and classification problems, and which have been successfully applied in several fields, including pattern recognition, stock market prediction, control tasks, medical diagnosis, prognosis, etc. Despite years of research in the field of ANN, selecting a proper architecture for a given problem remains a difficult task [1]–[3].

Among the several strategies to solve or alleviate this problem, constructive neural networks (CoNNs) offer the possibility of generating networks that grows as the input data are analyzed, and then they can match the complexity of the set of data [4]. Moreover, the training procedure in CoNN, considered a computationally expensive problem in the standard feedforward neural networks, can be done online and relatively fast. Competitive majority network trained by the error correction (C-Mantec) is a CoNN algorithm recently introduced [5] that implements competition between the neurons permitting them to learn during the whole training process as it does not freeze the synaptic weights of the previous incorporated neurons as most CoNN usually do, and also incorporates a built-in filtering scheme to avoid the overfitting problems. These two characteristics permit the algorithm to generate the compact neural architectures with very good generalization capabilities, making the algorithm suitable for its application to devices with limited resources such as microcontrollers, embedded systems, sensor networks, and field programmable gate arrays (FPGAs) [6].

FPGAs are the hardware devices created with the aim of prototyping digital circuits as they offer flexibility and speed. In the recent years, the advancement in technology has permitted to construct FPGAs with a considerable amount of processing power and memory storage, and so they have been applied in several domains (telecommunications, robotics, pattern recognition tasks, infrastructure monitoring, etc.) [7]–[9]. In particular, FPGAs seem quite suitable for neural network implementations as they are intrinsically parallel devices as is the processing of information in neural network models. Several studies have analyzed the implementation of neural networks models in FPGAs [10]–[12]. A broad classification of them can be done according to whether or not they include the learning process on-chip [3], [13]. In the off-chip learning implementations, the training of the neural network model is usually performed in a personal computer (PC), and only the synaptic weights are transmitted to the FPGA that acts as a hardware accelerator [14]–[16]. In contrast, the on-chip learning implementations [17]–[20] permit to train the models autonomously, independently of a PC, consuming much more FPGA resources but offering more flexibility and efficiency. Among the works in this category, we highlight the recent work by Lotrič and Bulić [19] where the backprogation algorithm is fully implemented and applied to a large set of benchmark functions, even though there is no significant performance through the use of an FPGA was found in comparison to the software models.

An important aspect at the time of the implementation of an algorithm in an FPGA is the data-type representation. The nature of the FPGAs encourages the use of an integer data type, or fixed point representation if a fractional part is needed, because this type of representation is more efficient. A floating point representation might be used but this would require the utilization of specific cores [21], [22]. In this sense, the work of Savich *et al.* (2007) [23] describes an interesting analysis of the implementation of floating point neural algorithms in the fixed point arithmetic, being the one used in this work.

Programming an FPGA is not a trivial task. A straightforward approach would be the use of standard programming languages (C, C++, Fortran, etc.) but unfortunately this option requires the use of language translators that so far tends to be quite inefficient. Instead, FPGAs are predominantly programmed

Manuscript received June 24, 2013; revised September 27, 2013; accepted November 29, 2013. Date of publication December 05, 2013; date of current version May 02, 2014. This work was supported in part by Junta de Andalucía under Grant P10-TIC-5770 and Grant P08-TIC-04026, and from CICYT (Spain) under Grant TIN2010-16556 [all include Fondo Europeo de Desarrollo Regional (FEDER) funds]. Paper no. TII-13-0416.

The authors are with the Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Campus de Teatinos S/N, 29071 Málaga, Spain (e-mail: lfranco@lcc.uma.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2013.2294137

using the hardware description languages such as VHDL or Verilog. These languages are complex and thus its programming is usually very time consuming. Furthermore, the existence of several FPGAs manufacturers and a lack of a common standard makes the situation a little bit more complicated [8].

In the present work, we have fully implemented the C-Mantec CoNN model in a VIRTEX-5 XC5VLX110T FPGA but using a general programming approach that may facilitate its portability to other board models. Our aim was twofold: first, to analyze the advantage in processing the speed that can be achieved through the FPGA parallel processing, and second to analyze the actual limitations in the utilization of an FPGA in terms of maximum number of patterns that can be stored, the dimension of the dataset, the differences in using a fixed point representation, etc., and, in particular, analyzing those aspects that arise only when a real hardware implementation is carried on.

The organization of the present work is as follows. Section II includes the details about the C-Mantec algorithm. The FPGA implementation is described in Section III, which contains four parts: the first three subsections describe the implementation of the blocks in which the FPGA is divided, and the fourth subsection deals with specific implementation details. Section IV contains the results of testing the implementation on two sets of benchmark functions (Boolean and real-valued problems), together with a detailed comparison of the computational speed between the current FPGA implementation and the original PC-based one. Finally, the paper ends with the discussion of the results and the conclusions obtained.

II. C-Mantec and CoNN Algorithm

C-Mantec [5] is a novel neural network constructive algorithm that utilizes competition between the neurons and a modified perceptron learning rule (thermal perceptron [24]) to build the single hidden layer compact architectures with good prediction capabilities for the supervised classification problems. As a CoNN algorithm, C-Mantec generates the network topology online during the learning phase, avoiding the complex problem of selecting an adequate neural architecture. The novelty of the C-Mantec in comparison to the previously proposed constructive algorithms is that the neurons in the single hidden layer compete for learning the incoming data, and this process permits the creation of very compact neural architectures. The binary activation state (S) of the neurons in the hidden layer depends on the N input signals ψ_i, and on the actual value of the N synaptic weights (ω_i) and bias (b) as follows:

$$S = \begin{cases} 1(\text{ON}), & \text{if } h \geq 0 \\ 0(\text{OFF}), & \text{otherwise} \end{cases} \tag{1}$$

where h is the synaptic potential of the neuron defined as

$$h = \sum_{i=1}^{N} \omega_i \, \psi_i - b. \tag{2}$$

In the thermal perceptron rule, the modification of the synaptic weights $\Delta\omega_i$ is done online (after the presentation of a single input pattern) according to the following equation:

$$\Delta\omega_i = (t - S)\psi_i T_{\text{fac}} \tag{3}$$

where t is the target value of the presented input and ψ represents the value of input unit i connected to the output by weight ω_i. The difference to the standard perceptron learning rule is that the thermal perceptron incorporates the T_{fac} factor. This factor, whose value is computed as shown in (4), depends on the value of the synaptic potential and on an artificially introduced temperature (T)

$$T_{\text{fac}} = \frac{T}{T_0} e^{-\frac{h}{T}}. \tag{4}$$

The value of T decreases as the learning process advances according to (5), similarly to a simulated annealing process

$$T = T_0 \cdot \left(1 - \frac{I}{I_{\max}}\right) \tag{5}$$

where I is a cycle counter that defines an iteration of the algorithm on one learning cycle and $I_{\max}$ is the maximum number of iterations allowed. One learning cycle of the algorithm is the process that starts when a chosen pattern is presented to the network and finishes after checking that all neurons respond correctly to the input or when the synaptic weights of the neuron chosen to learn the actual pattern (whether an existing or a new neuron) modifies its synaptic weights.

The C-Mantec algorithm has three parameters to be set at the time of starting the learning procedure, and several experiments have shown the robustness of the algorithm that operates fairly well in a wide range of parameter values. The algorithm has the following three parameters.

1) $I_{\max}$: maximum number of learning iterations allowed for each neuron in one learning cycle.
2) g_{fac}: growing factor that determines when to stop a learning cycle and include a new neuron in the hidden layer.
3) ϕ: determines in which case an input example is considered as noise and removed from the training dataset according to the following condition:

$$\text{delete}(x_i) | N_{\text{LT}} \geq (\mu + \phi \cdot \sigma) \tag{6}$$

where x_i represents an input pattern, N is the total number of patterns in the dataset, N_{LT} is the number of times that pattern x_i has been presented to the network on the current learning cycle, and μ and σ correspond to the mean and variance of the distribution for all patterns on the number of times where the algorithm has tried to learn each pattern in a learning cycle, respectively. The learning procedure starts with one neuron present in the single hidden layer of the architecture and an output neuron that computes the majority function of the responses of the hidden neurons (a voting scheme). The process continues by presenting an input pattern to the network and if it is misclassified, it will be learned by one of the present neurons whose output did not match the target pattern value if certain conditions are met, otherwise a new neuron will be included in the architecture to learn it. Among all neurons that misclassified the input pattern, the one with the largest T_{fac} will learn it but only if this T_{fac} value is larger than the g_{fac} parameter of the algorithm, a condition included to prevent the

unlearning of the previous stored information. If no thermal perceptron meeting these criteria is found, a new neuron is added to the network, starting a new learning cycle that includes the resetting of all neurons' temperature to T_0. Also, at the end of a cycle, the noisy patterns filtering procedure (6) is applied. The algorithm continues its operation iteratively by repeating the previous stages until all patterns in the training set are correctly classified by the network.

III. FPGA Implementation

FPGAs [25] are reprogrammable silicon chips, using prebuilt logic blocks and programmable routing resources. They can be configured to implement the custom hardware functionality, and in this sense, FPGAs are completely reconfigurable and can almost instantly change its behavior by recompiling a new circuitry configuration.

The board used for the current implementation is the Virtex-5 OpenSPARC evaluation platform (ML509). This device includes a Xilinx Virtex-5 XC5VLX110T FPGA that provides different connector devices: 2 USB (host and peripheral) ports, 2 PS/2 (keyboard, mouse) ports, RJ-45 (10/100/1000 networking) and RS-232 (male serial port) connectors, 2 Audio inputs (line, microphone), 2 audio outputs (line, Amp, SPDIF), video input, video output (DVI/VGA), and single-ended and differential I/O expansion. Table I shows some characteristics of the Virtex-5 XC5VLX110T FPGA, indicating its main logic resources. A picture of the Virtex-5 OpenSPARC platform is shown in Fig. 1.

The VHDL [26], [27] (VHSIC hardware description language) language is used for programming the FPGA, under the "Xilinx ISE Design Suite 12.4" environment using the "ISim M.81d" simulator. VHDL is a hardware description language widely used in the electronic design automation to describe the digital and mixed-signal systems such as FPGAs and integrated circuits, and can also be used as a general purpose parallel programming language. Our design strategy was to avoid the usage of specific Xilinx cores, in order to obtain a general design that can be potentially used in FPGAs from other manufacturers.

All computations have been performed using the fixed point arithmetic, which is the standard way to work with FPGA boards. Even if floating point operations can be codified in an FPGA [23], they tend to be more inefficient, as it is also the case for most digital circuits.

The implementation of the C-Mantec algorithm in the Virtex 5 FPGA was carried by dividing the board resources in three main blocks. Fig. 2 shows the control, pattern, and neuron blocks created, and the flow of information between them, necessary for the execution of the algorithm. We describe below the organization of each one of the three blocks followed by a subsection that comment on the specific implementations details.

A. Neurons Block

Each neuron consists of a group of look-up tables (LUTs) containing all information to compute its output (S), to calculate its T_{fac} value, and to modify its synaptic weights. Neurons receive the information about the input patterns from the pattern

TABLE I
Main Specifications of the Virtex-5 XC5VLX110T FPGA Related to its Available Slice Logic

Device	Slice registers	Slice LUTs	Bonded IOBs	Block RAM/FIFO
Virtex-5 XC5VLX110T	69 120	69 120	34	148

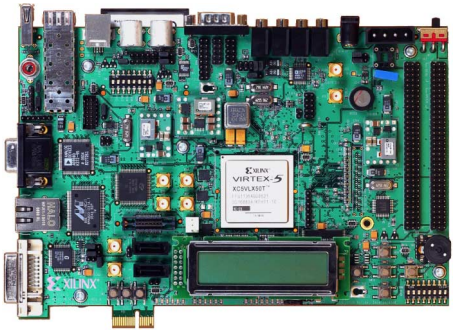

Fig. 1. Picture of a Virtex-5 OpenSPARC platform used for the implementation of the C-Mantec algorithm.

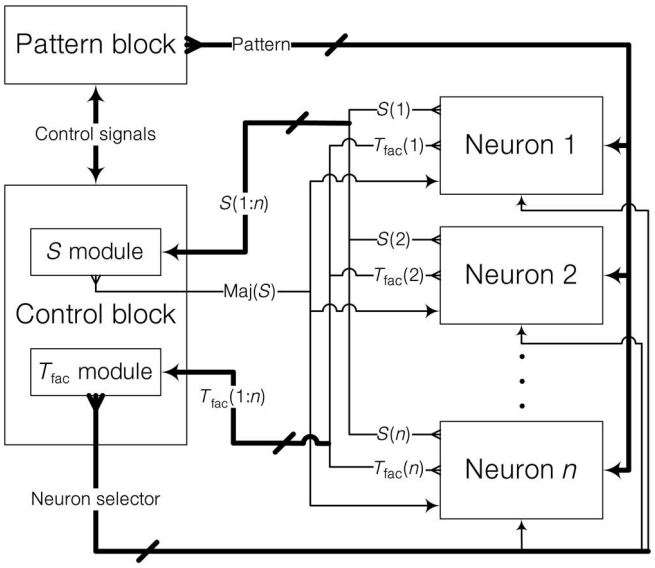

Fig. 2. Scheme of the FPGA design, where the control and pattern blocks are shown together with the neurons.

block and compute their synaptic potential value h to obtain a neuron's output through the following equation:

$$S = \begin{cases} 1(\text{ON}), & \text{if } h > 0 \\ 0(\text{OFF}), & \text{otherwise.} \end{cases} \tag{7}$$

Note that this equation is similar to the original C-Mantec equation (1) but the inequality condition for activating or not the neuron according to the value of h was modified, as now the neuron will be active only if $h > 0$ and not in the case $h = 0$. This change permits to compute the majority function of all neurons (the network output) in a much faster way (see control block section). The S values of all hidden neurons are sent by the neurons block to the control block for computing the whole network output value that is returned back to the neurons for

deciding whether to compute the T_{fac} values (in case the network output does not match the pattern target value) or to wait for the next input pattern (if network output match the pattern target value), starting the whole process again. In case the network output does not match the target of the current pattern, each neuron computes its T_{fac} value (4), and these values are sent to the control block that will return the information about which neuron has the largest T_{fac}, so this neuron modifies its synaptic weights if its T_{fac} is larger than the value of the parameter g_{fac}. Synaptic weight values are stored in the registers of the FPGA instead of using the RAM block in order to reduce the set-up problems because in this way they are kept close to the neurons.

B. Pattern Block

Training patterns are managed by the pattern block. Through the serial port pattern, the values are received from the PC and stored in the FPGA distributed RAM block. Patterns are represented using 2^n bytes, with the value of n determined by the input dimension of the patterns, noting that one extra bit is reserved for the pattern class (the target output).

During the execution of the algorithm after a signal from the control block is received, the pattern block sends one randomly selected pattern to all neurons. In order to avoid repeated training of a given pattern, the memory position of the last sent pattern is switched with the one corresponding to the final eligible memory position, whereas the number of eligible memory positions is reduced by 1. This action is repeated until a pattern is found for which the network output is different from its target value, as this case involves modifying the synaptic weights and also the beginning of a new cycle.

The C-Mantec algorithm incorporates a filtering scheme for removing patterns considered as "noisy" samples, task also performed by the pattern block. Every time a new neuron is added to the architecture, patterns that needed a number of weight updates larger than the mean plus ϕ standard deviation of the whole set of patterns [see (6) and related text] are considered as noise and removed from the training set. In practice, the removal procedure works by storing these "noisy" patterns in the last memory positions.

C. Control Block

The control block organizes the whole information process by sending and processing the information from the neurons and pattern blocks. Once the patterns have been loaded into the pattern block, the control receives a signal in order to start the execution of the algorithm. The process start by sending a signal to the pattern block indicating that a random chosen pattern should be sent to the neurons.

As C-Mantec is a constructive ANN model, the architecture grows as the training phase proceeds, but as FPGA programming needs all elements to be specified before execution, the maximum number of possible neurons in the architecture is created at the beginning with a flag managed by the control block, which determines whether a neuron is activated (included in the actual architecture) or inactivated (waiting for its potential inclusion). At the beginning of the learning process, all synaptic weights are set to zero and the control unit activates only one neuron.

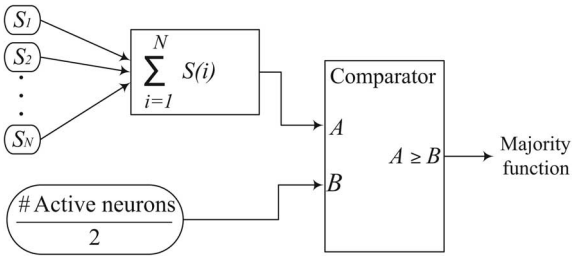

Fig. 3. Functional diagram of the S module for computing the majority function of the hidden neurons outputs.

When a random chosen pattern is sent to the neurons, their outputs (S) are computed and the control block calculates the majority function of these signals, which is the whole network output in response to the presented input. This output value is compared with the target value and if they match, the control block sends a signal for the release of a new pattern, but if they are different, T_{fac} values are obtained for each of the neurons in order to choose the one with the largest value and in case if its T_{fac} is larger than the value of parameter g_{fac}, start the modification of its synaptic weights. T_{fac} values are received from the neurons and the control unit chooses among the wrong ones, the neuron with the largest T_{fac}. At this point, the control block sends a signal to the chosen neuron so this unit modifies its synaptic weights. The modification of the weights is done serially, updating every synapse according to (3).

The control block contains two modules: the T_{fac} and the S (Output) modules. The former one calculates the largest T_{fac} of a group of neurons, whereas the S module calculates the majority function of the outputs (S). A detailed description of both the modules is given in the following section.

D. Implementation Details

1) S Module: This module computes the majority function of the hidden neurons outputs (S). A diagram showing the schematic operation of this module is shown in Fig. 3. Hidden neurons output (S_i) are first added and then this value is compared to the number of active neurons divided by 2, so the majority function can be computed. The output of this module is "1" when the number of activated neurons is larger than the inactivated ones and "0" otherwise. Regarding the logical operation of the circuit, half of the number of active neurons is obtained using a right logical shift operation applied to the number of active neurons. The operation of this module can be performed in just one clock cycle, because computations within this block are not synchronized.

2) T_{fac} Module: This module is in charge of computing the largest T_{fac} value among all neurons whose output does not match the target value for a presented pattern ("wrong" neurons). Since it is more efficient to compute the T_{fac} value among all neurons, we set this value to 0 for the "correct" neurons to obtain the largest value among all (also the T_{fac} value of inactive neurons is set to 0). The highest T_{fac} is calculated in blocks of 16 neurons, as this number was the largest one for which it is not necessary to reduce the operative frequency of the system. If the number of active neurons is larger than 16, the obtained value of the first block is saved and compared to the one obtained from

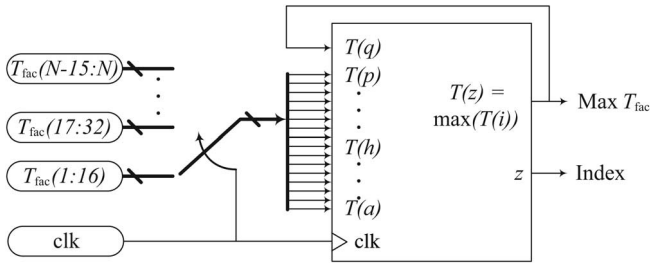

Fig. 4. Functional diagram of the T_{fac} module, which calculates the largest T_{fac} value among active neurons and the index of the neuron for which this value was obtained.

the second block, saving the largest value found so far and proceeding in this way until all active neurons are analyzed. Fig. 4 shows an schematic drawing of the procedure used to find the neuron with the largest T_{fac} among those that incorrectly classify the input pattern. This figure shows on its bottom left of the clock signal that synchronizes the module operation. On top of it, T_{fac} values grouped in blocks of 16 that are iteratively sent to the logic circuit that computes the maximum of this block, taking into account the maximum value obtained so far [indicated by $T(z)$ in the figure]. After all blocks are analyzed, the T_{fac} module outputs the maximum value found (Max T_{fac}) together with a value (*Index*) that indicates the neuron for which this value was found.

3) Product Implementation: The execution of the algorithm requires the computation of several products, such as the computation of the T_{fac} values, the obtention of the synaptic weights, and the neuron potential (h). In particular, the computation of the T_{fac} value requires three multiplications; one related to the calculation of T, another for the interpolation of the exponential function, and the last for obtaining its final value [cf. (4)]. Every neuron has only one multiplier block that is used in a time-division multiplexing scheme to minimize the LUTs' usage. Xilinx specific multiplication cores are not used because we have decided to use the exportable code, allowing its potential application to other FPGAs. The multiplication block is based on the use of shifters and adders, following the approach introduced in [28]. The number of LUTs required in this scheme is proportional to the bit size of the input data, e.g., for two vectors with $\boldsymbol{N}_a$ and $\boldsymbol{N}_b$ bits length, respectively, the product requires $\boldsymbol{N}_a \times \boldsymbol{N}_b$ LUTs, whereas the output has a size of $\boldsymbol{N}_a + \boldsymbol{N}_b$ bits.

4) Synaptic Weights Precision: The representation of the synaptic weights can be chosen according to the available resources, taking into account that obtaining a higher accuracy may require a larger representation, which will imply an increase in the number of LUTs per neuron (consequently a reduced number of available neurons) and a decrease in the maximum operation frequency of the board. Synaptic weight accuracy is important so that the resulting values are similar to those obtained using the floating point representation used in the PC-based code. A synaptic weight is represented by a bit array with integer and fractional parts of lengths N_1 and N_2. N_1 determines the minimum and maximum values that can be obtained as $-2^{(N_1-1)}$ to $2^{(N_1-1)}$, whereas N_2 defines the accuracy $2^{(-N_2)}$. The number of bits needed to represent all possible discrete values within a certain range of positive values depends on the difference between the maximum and minimum values of the interval, and can be obtained from the following equation:

$$\#\text{bits} = \log_2((1 + \max(w_{ij}))/(\min(w_{ij}))). \tag{8}$$

Table II shows the number of LUTs, maximum operating frequency, number of available neurons, and accuracy according to the number of bits used for representing the synaptic weights $N_1 + N_2$, where N_1 and N_2 indicate the integer and fractional parts of the representation, respectively.

5) Number of Inputs: The number of inputs in a C-Mantec network is determined by the dimension of the training patterns. The input dimension strongly influences the whole FPGA implementation as it essentially modifies the number of LUTs per neuron and the maximum number of training patterns that can be stored in the memory board. Further, changing the number of inputs involves a complete modification of the programming code, and in this sense, our approach was to develop different codes according to the maximum number of inputs needed. Of course, one nonoptimal option is to choose the largest input code for all cases, clearly reducing other capabilities of the board. Table III shows the number of LUTs per neuron, the number of RAM blocks used every 1024 patterns, the maximum number of patterns that can be stored ($1024 \times \#$ RAM blocks), and the number of LUTs and registers of the pattern block, all as a function of the number of inputs. The size of the inputs shown in the table corresponds to values of a power of 2 minus 1, as the remaining input is used for the output class.

6) Exponential Function: The calculation of the T_{fac} value involves the computation of an exponential functions involving a negative number. As there is no floating point operations defined in the FPGA, integer fixed point arithmetic should be used. For the exponential function, a table was created that contains the value of the function for some selected inputs.

TABLE II
NUMBER OF LUTS, MAXIMUM OPERATING FREQUENCY, NUMBER OF AVAILABLE NEURONS AND ACCURACY ACCORDING TO THE NUMBER OF BITS USED FOR REPRESENTING THE SYNAPTIC WEIGHTS

N_1	N_2	LUTs per neuron	Max f (MHz)	# Available neurons	Accuracy
8	8	689	74.184	94	$3.9 \cdot 10^{-3}$
12	8	757	74.184	85	$3.9 \cdot 10^{-3}$
12	12	943	53.735	68	$2.4 \cdot 10^{-4}$
16	8	826	74.184	78	$3.9 \cdot 10^{-3}$
16	12	1033	53.735	62	$2.4 \cdot 10^{-4}$
16	16	1299	42.715	50	$1.5 \cdot 10^{-5}$

N_1 and N_2 indicate the integer and fractional parts of the representation.

TABLE III
LUTS PER NEURON, NUMBER OF RAM BLOCKS USED, MAXIMUM NUMBER OF PATTERNS AND NUMBER OF LUTS AND REGISTERS USED IN THE PATTERN BLOCK AS A FUNCTION OF THE INPUT SIZE

Inputs	LUTs per neuron	Pattern block		Block RAM	Max patterns
		# Registers	#LUTS		
7	665	505	609	2	72 728
15	689	764	629	4	37 888
31	729	1029	674	8	18 944
63	793	1797	930	15	9216

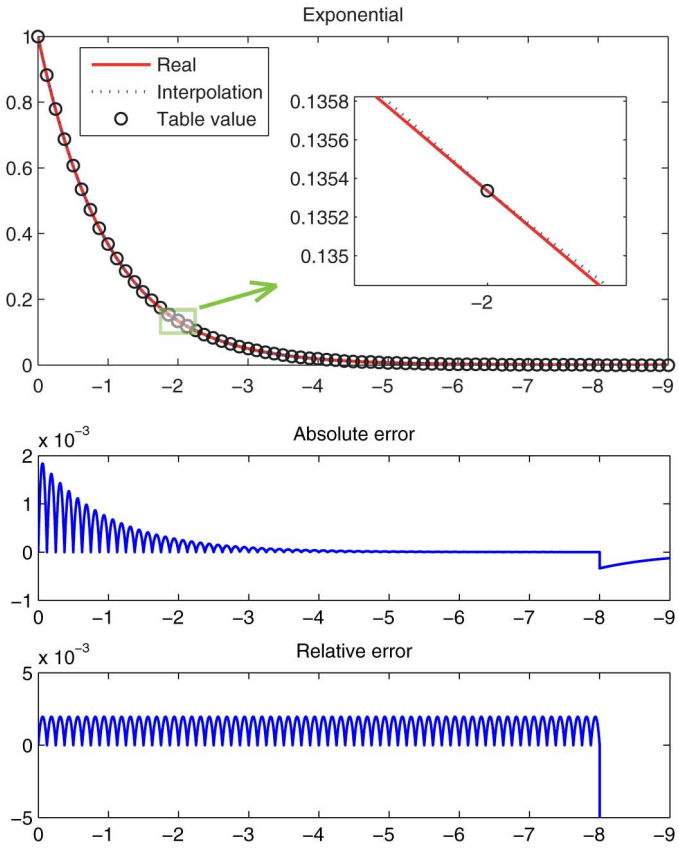

Fig. 5. Exponential function and approximated value (top graph). Absolute (middle) and relative errors committed in the approximation of the exponential function (see text for more details).

Storing the table values requires large amounts of memory, and as one table per neuron is needed, only selected input values were stored, and thus the table was built with 64 values starting from 0 down to -8 with 0.125 decreasing steps (values lower than -8 were set to 0). To increase the precision of the exponential calculation, a linear interpolation procedure was implemented. Fig. 5 (top) plots the real, table, and interpolated values for the exponential function, where the inset plot shows an enlargement of a portion of the curve. Fig. 5 (middle and bottom graphs) shows the absolute and relative errors of the exponential function in the range from 0.0 to -9.0.

7) $I_{\max}$: This value, which determines the maximum number of learning modifications that a neuron can suffer in a learning cycle, has been modified with respect to the original C-Mantec code. As this value is employed for calculating the temperature (T) value, through a division operation with a high associated computational cost, $I_{\max}$ was limited power of 2 values, allowing a conversion of the division operation in a right logical shift. The maximum allowed value of $I_{\max}$ is $2^{17} = 131072$.

IV. Results

Different comparisons have been performed between the implementation in VHDL for the selected FPGA and the implementation of the C-Mantec algorithm in C programming language [29], noting that this language is considered among the fastest that can be used in a PC [30]–[32]. The CPU used for running the C code is an Intel (R) core (TM) Quad CPU Q6600 @ 2.4 GHz.

TABLE IV
Mean and Standard Deviation of the Number of Neurons in the Generated Architectures and the Time (in Seconds) Needed by the Algorithm for a Set of 16 Boolean Functions, Both for FPGA and PC Implementations

Funct.	# Neurons C		# Neurons FPGA		Mean time (s)	
	$\overline{x}$	σ	$\overline{x}$	σ	C	FPGA
Xor2	2	0	2	0	0.012	0.008
Xor3	3	0	3	0	0.041	0.03
Alu2k	10.8	0.77	10.66	0.66	9.64	1.31
Alu2l	19.15	0.99	18.6	1.18	94	5.9
Alu2m	2	0	2	0	0.013	0.03
Alu2n	1	0	1	0	0.001	0.01
Alu2o	11.7	0.58	11.52	0.73	36.7	3.69
Alu2p	2	0	3	0	0.016	0.056
Alu4o	10.8	0.7	10.7	0.48	9.5	1.5
Alu4p	19.5	1.4	19.3	1.16	120.5	6.38
Alu4q	33.85	2.06	34.6	2.26	995	28.2
Alu4r	48.95	4.99	49.23	4.57	5711	121.2
Alu4s	2	0	2	0	0.032	0.031
Alu4t	1	0	1	0	0.013	0.014
Alu4u	24.3	1.53	22.2	1.5	1026	33.6
Alu4v	2	0	3	0	0.11	0.018
Average	12.13	0.81	12.11	0.78	500.2	12.62

A set of 16 single output Boolean functions was used to test the speed and the architectures generated by the C-Mantec algorithm. The functions used for the tests comprises two sets of arithmetic logic unit (ALU) functions that include 6 *Alu2* functions of 10 inputs and 8 *Alu4* functions of 14 inputs, both from the MCNC (Microelectronics Center of North Carolina) benchmark [33], together with the exclusive disjunction function (XOR) of two and three inputs. The C-Mantec algorithm was run with the following parameter values: $g_{\text{fac}} = 0.01$ and $I_{\max} = 16\ 384$. (No noise elimination step was applied in this case, i.e., $\phi = \infty$.) Also, a phase-locked loop (PLL) block has been used to set the frequency of the system to 72.72 MHz for all tests. Table IV shows the number of neurons and time (s) needed by the VHDL–FPGA and the C-PC implementations, computed as the mean over 20 simulations run for each function [the standard deviation (σ) is also shown]. A comparison of the learning times is displayed in Fig. 6 using a logarithmic y-axis. All the results shown in Table IV were obtained using a 16-bit fixed point representation for the synaptic weights ($N_1 = 8, N_2 = 8$), except the values for function *Alu4r* for which a larger representation was needed ($N_1 = 12, N_2 = 8$) in order to avoid the saturation effects.

We have also computed the time (s) that the system employs for three important tasks involved in a learning cycle: computing the majority function of the hidden neurons' responses, determining the largest T_{fac} value, and modifying the synaptic weights. For obtaining the majority function, the system needs $8 + 2N_I$ cycles, for determining the largest T_{fac} $34 + \text{ceil}(N_N/16)$, and for modifying the synaptic weights $4 + 2N_I$ cycles (N_I is the number of inputs and N_N is the number of neurons). Fig. 7 shows a graphical representation of the running times for the values of $N_I = 10$ and $N_N = 25$.

We further computed the execution times related to the addition of a new neuron as the constructive network grows, comparing the times needed by the FPGA and PC-based implementations using the 10-input function *alu2l* (30 repetitions

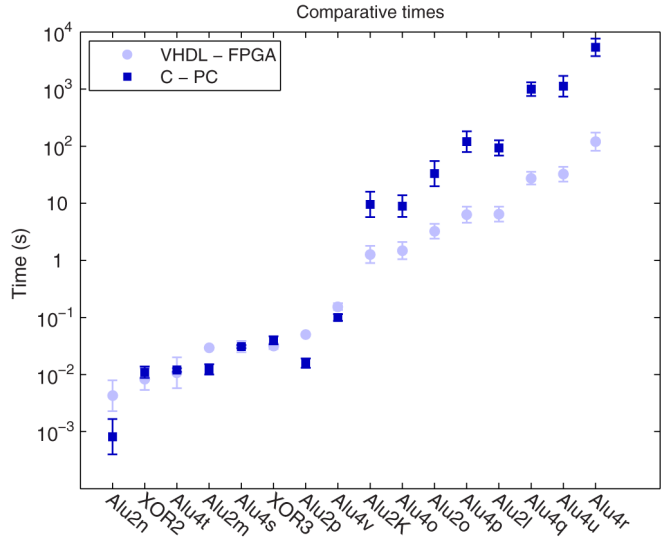

Fig. 6. Mean and standard deviation of the learning times (in seconds, logarithmic scale) for a set of 16 Boolean functions for VHDL–FPGA and C-PC implementations of the C-Mantec algorithm (these values are also represented in Table IV).

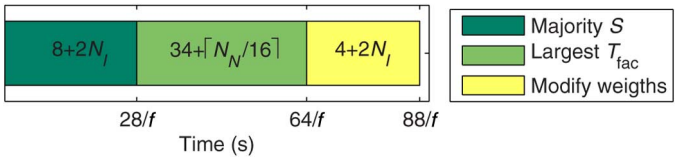

Fig. 7. Run time of computing the majority function, determining the largest T_{fac} and modifying the synaptic weights in a learning cycle (see text for more details).

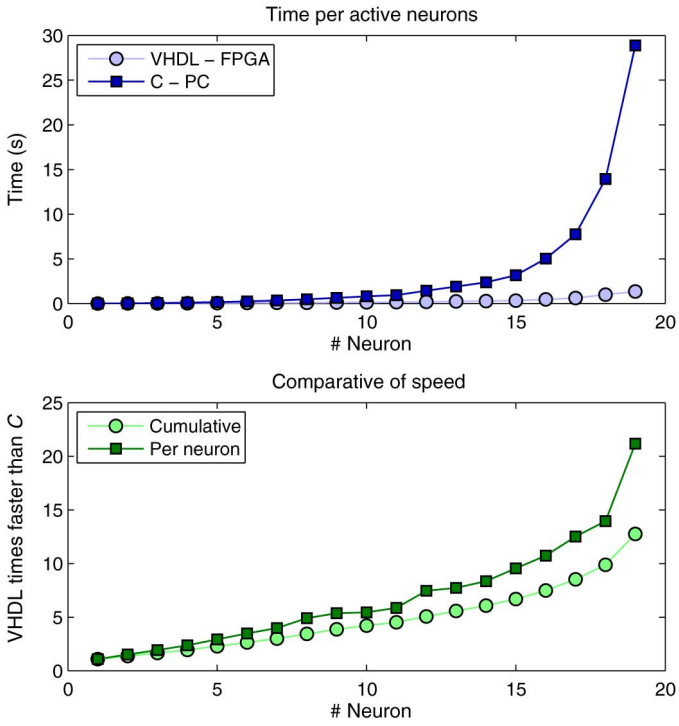

Fig. 8. Execution times needed by the FPGA and PC implementations of C-Mantec related to the addition of a new neuron. Absolute times (top graph) and relative times (bottom graph) computed for the *alu2l* function (see the text for more details).

were performed in order to obtain an average value). Fig. 8 (top) shows the time needed by the FPGA and PC implementation, whereas Fig. 8 (bottom) shows the relative speed increase between the two platforms. The two curves in Fig. 8 (bottom) corresponds to the relative time needed by the algorithm to achieve the shown number of neurons (cumulative), whereas the upper curve (per neuron) corresponds to the comparison when the algorithm is executed with a constant number of neurons. The cumulative comparison clearly shows that the relative performance of the FPGA (in comparison to the PC implementation) increases almost linearly as the number of neurons in the architecture increases.

In another experiment, we analyzed the implementation of real-world functions, using a set of eight binary output benchmark problems from the UCI[1] repository. Table V shows the results for the FPGA and PC implementations of the C-Mantec algorithm. The first two columns of the table show the name and number of inputs of the used benchmark problems. The third and fourth columns show the number of neurons obtained for both the implementations, and in the last two columns the generalization ability is shown for both the approaches. The generalization ability was computed using a 10-fold training/test scheme with the following parameters for the C-Mantec algorithm: ($g_{\text{fac}} = 0.1$, $I_{\max} = 65\ 536$, $\phi = 2$).

[1]www.ics.uci.edu/ mlearn/.

TABLE V
RESULTS OF THE IMPLEMENTATION OF EIGHT BENCHMARK PROBLEMS IN THE FPGA VERSION OF THE C-MANTEC ALGORITHM

Function	Inputs	# Neurons		Generalization	
		PC	FPGA	PC	FPGA
Diabetes	8	5.0	5.0	76.6 ± 2.7	74.2 ± 4.9
Cancer1	9	1.0	2.0	96.9 ± 1.2	97.1 ± 2.6
Heart1	13	2.7	2.2	82.6 ± 2.5	80.6 ± 6.7
Ionosphere	34	2.0	2.0	87.4 ± 0.1	87.2 ± 3.7
Heart-c	35	2.0	2.0	82.5 ± 3.3	82.8 ± 6.9
Kr-vs-Kp	40	3.0	2.0	98.5 ± 0.6	96.9 ± 0.9
Card	51	1.8	2.1	85.2 ± 2.5	85.9 ± 3.6
Sonar	60	1.0	1.2	75.0 ± 14	75.7 ± 15.3
Average	–	2.3	2.3	85.6 ± 3.4	85.1 ± 5.6

The first two columns show name and number of inputs of the function and the rest of the columns shows the number of neurons and the generalization ability obtained with both platforms.

V. DISCUSSION AND CONCLUSION

We have presented and analyzed in this work an FPGA on-chip learning implementation of the recently introduced C-Mantec neural network constructive algorithm. One of the main advantages of using this new algorithm is the fact that in comparison to backpropagation training (the standard choice in the ANN field) it avoids the problem of selecting the adequate architecture, as this process is done automatically according to the complexity of the input data. A further advantage of C-Mantec is its robustness regarding the parameter settings [5].

Several tests carried on two well-known benchmark datasets (cf. Tables IVand V) show that the fixed precision representation

used (16 bits fixed point) for most cases is enough to achieve the results comparable to the floating point original PC-based execution of the algorithm, as almost indistinguishable results were obtained for the size of the generated architectures and for the generalization ability of the algorithm.

Regarding the advantage of using the FPGA version of the algorithm instead of standard software, a clear significant speed increase has been observed, noting that this increase grows approximately linear as the complexity of the problem augments (cf. Fig. 7 bottom graph), thus obtaining a factor increase of up to 47 times in the case of the most complex function analyzed (function *Alu4r* in Table IV).

In the light of the observed results, we can conclude that the present analysis demonstrates the suitability of the C-Mantec algorithm for its application in the real-world industrial problems in which FPGAs are commonly used [34], such as industrial motor control [35], machine vision [36], industrial networking [37], robotics [38], etc. Further, the experiments carried out confirm the great potential that FPGAs have for neurocomputational tasks given its intrinsic parallel processing.

Acknowledgments

The authors would like to acknowledge P. Monasterio-Huelin, A. I. Molina, and M. González for fruitful discussions.

References

[1] I. Gómez, L. Franco, and J. Jerez, "Neural network architecture selection: Can function complexity help?," *Neural Process. Lett.*, vol. 30, pp. 71–87, 2009.

[2] D. Hunter, Y. Hao, M. Pukish, J. Kolbusz, and B. Wilamowski, "Selection of proper neural network sizes and architectures—A comparative study," *IEEE Trans. Ind. Appl.*, vol. 8, no. 2, pp. 228–240, May 2012.

[3] K. Lakshmi and M. Subadra, "A survey on FPGA based MLP realization for on-chip learning," *Int. J. Sci. Eng. Res.*, vol. 4, 2013.

[4] L. Franco, D. Elizondo, and J. Jerez, *Constructive Neural Networks*, Berlin, Germany: Springer-Verlag, 2009.

[5] J. Subirats, L. Franco, and J. Jerez, "C-Mantec: A novel constructive neural network algorithm incorporating competition between neurons," *Neural Netw.*, vol. 26, pp. 130–140, 2012.

[6] D. Urda, E. Canete, J. L. Subirats, L. Franco, L. Llopis, and J. M. Jerez, "Energy-efficient reprogramming in WSN using constructive neural networks," *Int. J. Innov. Comput. Inf. Control*, vol. 8, pp. 7561–7578, 2012.

[7] E. Monmasson, L. Idkhajine, M. Cirstea, I. Bahri, A. Tisan, and M. W. Naouar, "FPGAs in industrial control applications," *IEEE Trans. Ind. Informat.*, vol. 7, no. 2, pp. 224–243, May 2011.

[8] D. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Queue*, vol. 11, pp. 40–52, 2013.

[9] P. Conmy and I. Bate, "Component-based safety analysis of FPGAs," *IEEE Trans. Ind. Informat.*, vol. 6, no. 2, pp. 195–205, May 2010.

[10] J. Zhu and P. Sutton, "FPGA implementations of neural networks—A survey of a decade of progress," *Lecture Notes Comput. Sci.*, vol. 22778, pp. 1062–1066, 2003.

[11] A. Gomperts, A. Ukil, and F. Zurfluh, "Development and implementation of parameterized FPGA-based general purpose neural networks for online applications," *IEEE Trans. Ind. Informat.*, vol. 7, no. 1, pp. 78–89, Feb. 2011.

[12] Q. Le and J. Jeon, "Neural-network-based low-speed-damping controller for stepper motor with an FPGA," *IEEE Trans. Ind. Appl.*, vol. 57, no. 9, pp. 3167–3180, Sep. 2010.

[13] A. Omondi and J. Rajapakse, *FPGA Implementations of Neural Networks*, New York, NY, USA: Springer, 2006.

[14] S. Himavathi, D. Anitha, and A. Muthuramalingam, "Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization," *IEEE Trans. Neural Netw.*, vol. 18, no. 3, pp. 880–888, May 2007.

[15] S. Jung and S. S. Kim, "Hardware implementation of a real-time neural network controller with a DSP and an FPGA for nonlinear systems," *IEEE Trans. Ind. Electron.*, vol. 54, no. 1, pp. 265–271, Feb. 2007.

[16] T. Orlowska-Kowalska and M. Kaminski, "FPGA implementation of the multilayer neural network for the speed estimation of the two-mass drive system," *IEEE Trans. Ind. Informat.*, vol. 7, no. 3, pp. 436–445, Aug. 2011.

[17] A. Dinu, M. Cirstea, and S. E. Cirstea, "Direct neural-network hardware-implementation algorithm," *IEEE Trans. Ind. Electron.*, vol. 57, no. 5, pp. 1845–1848, May 2010.

[18] A. Gomperts, A. Ukil, and F. Zurfluh, "Implementation of neural network on parameterized FPGA," in *Proc. AAAI Spring Symp.: Embedded Reasoning*, Stanford, CA, USA, 2010.

[19] U. Lotrič and P. Bulić, "Applicability of approximate multipliers in hardware neural networks," *Neurocomputing*, vol. 96, pp. 57–65, 2012.

[20] J. Shawash and D. Selviah, "Real-time nonlinear parameter estimation using the Levenberg-Marquardt algorithm on field programmable gate arrays," *IEEE Trans. Ind. Electron.*, vol. 60, no. 1, pp. 170–176, Jan. 2013.

[21] C. H. Ho, C. W. Yu, P. Leong, W. Luk, and S. J. E. Wilton, "Floating-point FPGA: Architecture and modeling," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 12, pp. 1709–1718, Dec. 2009.

[22] Z. Jovanovic and V. Milutinovic, "FPGA accelerator for floating-point matrix multiplication," *IET Comput. Digit. Technol.*, vol. 6, no. 4, pp. 249–256, 2012.

[23] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 240–252, Jan. 2007.

[24] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," *Neural Comput.*, vol. 2, no. 2, pp. 198–209, Apr. 1990.

[25] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*, Hoboken, NJ, USA: Wiley, 2007.

[26] P. Ashenden, *The Designer's Guide to VHDL, Vol. 3, (Systems on Silicon)*, 3rd ed., Burlington, MA, USA: Morgan Kaufmann Publishers, 2008.

[27] P. P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*, Hoboken, NJ, USA: Wiley, 2008.

[28] P. P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, Hoboken, NJ, USA: Wiley, 2006.

[29] D. M. Ritchie, "The development of the C language," *Proc. 2nd ACM SIGPLAN Conf. History Program. Lang*, ser. HOPL-II, New York, NY, USA: ACM, 1993, pp. 201–208.

[30] B. Fulgham, "The computer language benchmarks game" [Online]. Available: http://benchmarksgame.alioth.debian.org/ Last accessed on Dec. 27, 2013.

[31] Bioinformatics.org, "Language benchmark" [Online]. Available: http://www.bioinformatics.org/benchmark/results.html Last accessed on Dec. 27, 2013.

[32] R. Hundt, (2011). "Loop recognition in C++/java/go/scala," in *Proc. Scala Days 2011* [Online]. Available: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf.

[33] ACM/SIGDA benchmarks [Online]. Available: http://www.cbl.ncsu.edu:16080/benchmarks/.

[34] Y. Lin, *Using FPGAs to Solve Challenges in Industrial Applications, WP410 (V2.0)*, Xilinx, Oct. 2012.

[35] S. Maiti, V. Verma, C. Chakraborty, and Y. Hori, "An adaptive speed sensorless induction motor drive with artificial neural network for stability enhancement," *IEEE Trans. Ind. Informat.*, vol. 8, no. 4, pp. 757–766, Nov. 2012.

[36] D. Kim, T. Nguyen, M. Kim, and J. Jeon, "Design and implementation of a pipelined datapath for high-speed face detection using FPGA," *IEEE Trans. Ind. Informat.*, vol. 8, no. 1, pp. 158–167, Feb. 2012.

[37] P. Ferrari, A. Flammini, D. Marioli, and A. Taroni, "A distributed instrument for performance analysis of real-time ethernet networks," *IEEE Trans. Ind. Informat.*, vol. 4, no. 1, pp. 16–25, Feb. 2008.

[38] X. Shao and D. Sun, "Development of a new robot controller architecture with FPGA-based IC design for improved high-speed performance," *IEEE Trans. Ind. Informat.*, vol. 3, no. 4, pp. 312–321, Nov. 2007.

Francisco Ortega-Zamorano, photograph and biography not available at time of publication.

José M. Jerez, photograph and biography not available at time of publication.

Leonardo Franco (M'06–SM'13), photograph and biography not available at time of publication.

Capítulo 4

Reprogramación de sensores inteligentes en entornos cambiantes usando un modelo de red neuronal

Francisco Ortega-Zamorano, José M. Jerez, José Luis Subirats, Ignacio Molina and Leonardo Franco: Smart sensor/actuator node reprogramming in changing environments using a neural network model. **Engineering Applications of Artificial Intelligence** 30: 179-188 (2014).

Posición JCR: Q2 (31/121) en Computer science: Artificial Intelligence.
Q1 (15/87) en Engineering: Multidisciplinary.
Factor de impacto: 1,962

Resumen: Las técnicas desarrolladas en la actualidad como método para la actualización de los nodos de sensores requieren generalmente del uso de reprogramación; si éstos se encuentran en entornos cambiantes dicha reprogramación podría llegar a ser habitual, incrementando de forma sustancial los costes en términos de energía y tiempo. El trabajo presenta una alternativa al enfoque tradicional para la reprogramación de nodos sensores/actuadores en entornos cambiantes basada en un esquema de aprendizaje en el propio sensor para que de forma automática adapte el comportamiento a las condiciones del entorno. El modelo de aprendizaje propuesto se basa en el C-Mantec, un nuevo algoritmo de red neuronal constructivo especialmente adecuado para las implementaciones del microcontrolador, ya que genera arquitecturas de tamaño muy compacto reduciendo significativamente el uso de memoria del microcontrolador. La placa seleccionada ha sido el Arduino UNO, una placa microcontroladora muy popular de código abierto, económica y eficiente. Además este trabajo aporta un análisis en profundidad de las soluciones adoptadas para superar las limitaciones de recursos hardware en la implementación del algoritmo de aprendizaje junto con una evaluación de la eficiencia de este enfoque, probando el algoritmo en un conjunto de datos de funciones de referencia. Finalmente la utilidad y versatilidad del sistema se prueban en tres casos de estudios de diferente naturaleza en los cuales las condiciones ambientales evolucionan con el tiempo, cambiando el comportamiento del sistema.

Engineering Applications of Artificial Intelligence 30 (2014) 179–188

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Smart sensor/actuator node reprogramming in changing environments using a neural network model

Francisco Ortega-Zamorano [a,*], José M. Jerez [a], José L. Subirats [a], Ignacio Molina [b], Leonardo Franco [a]

[a] Department of Computer Science, E.T.S.I. Informatica, University of Malaga, Bulevar Louis Pasteur, 35, 29071 Malaga, Spain
[b] Max Planck Institute, Munich, Germany

ARTICLE INFO

Article history:
Received 17 September 2013
Received in revised form
23 December 2013
Accepted 9 January 2014
Available online 1 February 2014

Keywords:
Constructive Neural Networks
Microcontroller
Arduino

ABSTRACT

The techniques currently developed for updating software in sensor nodes located in changing environments require usually the use of reprogramming procedures, which clearly increments the costs in terms of time and energy consumption. This work presents an alternative to the traditional reprogramming approach based on an on-chip learning scheme in order to adapt the node behaviour to the environment conditions. The proposed learning scheme is based on C-Mantec, a novel constructive neural network algorithm especially suitable for microcontroller implementations as it generates very compact size architectures. The Arduino UNO board was selected to implement this learning algorithm as it is a popular, economic and efficient open source single-board microcontroller. C-Mantec has been successfully implemented in a microcontroller board by adapting it in order to overcome the limitations imposed by the limited resources of memory and computing speed of the hardware device. Also, this work brings an in-depth analysis of the solutions adopted to overcome hardware resource limitations in the learning algorithm implementation (e.g., data type), together with an efficiency assessment of this approach when the algorithm is tested on a set of circuit design benchmark functions. Finally, the utility, efficiency and versatility of the system is tested in three different-nature case studies in which the environmental conditions change its behaviour over time.

1. Introduction

Sensors (or detectors) are devices that permit the measurement of chemical or physical variables, transforming them into electrical signals, in order to interpret the signals from various sensors and send the sensed data or make a decision according to them. Microcontroller boards are an economic, small and flexible solution, and thus are the most common controller used in a sensor node, also known as a Mote, commonly used in Wireless sensor network (Yick et al., 2008; Sengupta et al., 2013), but also used in other important technologies such as Embedded systems (Marwedel, 2006; Mamdoohi et al., 2012) and Real-time systems (Kopetz, 1997; Wang et al., 2010). Motes are nowadays widely employed in all kind of industrial applications, in several of them, the problem requires an action upon the environmental conditions, in this case a sensor/actuator node is required.

In cases when the environmental conditions evolve over time, the original sensor/actuator programming can lead to incorrect decisions, and thus it is necessary to change or adapt the decision-making process to the new conditions (Sayed-Mouchaweh and Lughofer, 2012). The traditional option to solve this problem has been to send the sensed data to a central unit, where a person interprets the data and reprogram the microcontroller with the new set of rules (Han et al., 2005; Wang et al., 2006; Shaikh et al., 2010). Different reprogramming techniques have been proposed as a way of dynamically changing the behaviour of the sensors without having to manually reprogram them, because traditional reprogramming requires in most cases the interruption of the process for loading the new binary code, with the consequent loss of time and energy, involved in the communication process to the central unit (Rassam et al., 2013; Aiello et al., 2011). A first step towards reducing the previous effects has been to incorporate machine learning systems in the decision-making process, automating the response of the microcontroller without interrupting its execution and sending just a small fraction of code to the microcontroller (Urda et al., 2012; Canete et al., 2012; Farooq et al., 2010). However, recent advances in the computing power of sensors permit the inclusion of learning systems in the microcontroller ("on-chip" learning), adapting the sensor/actuator behaviour dynamically according to the sensed data (Aleksendrić et al., 2012; Mahmoud et al., 2013).

Artificial Neural Networks (ANNs) (Haykin, 1994) are a kind of machine learning models, inspired on the functioning of the brain,

* Corresponding author. Tel.: +34 952 13 28 47; fax: +34 952 131 397.
E-mail addresses: fortega@lcc.uma.es, FOZamorano@gmail.com (F. Ortega-Zamorano).

0952-1976/$ - see front matter © 2014 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.engappai.2014.01.006

that can be utilised in clustering and classification problems, having been applied successfully in several fields, including pattern recognition (Dhanalakshmi et al., 2011), stock market prediction (Park and Shin, 2013), control tasks (Zhai and Yu, 2009), medical diagnosis and prognosis (Kodogiannis et al., 2007), and so on. Despite years of research in the field of ANN, selecting a proper architecture for a given problem remains a difficult task (Gómez et al., 2009; Hunter et al., 2012; Lakshmi and Subadra, 2013), and several strategies have been proposed for solving or alleviating this issue. In particular, Constructive Neural Networks (CoNNs) offer the possibility of generating networks that grows as input information is received, matching the complexity of the data (Franco et al., 2009). Moreover, the training procedure in CoNN, considered a computationally expensive problem in standard feedforward neural networks, can be done on-line and relatively fast. C-Mantec is a recently introduced CoNN algorithm that implements competition between neurons, also incorporating a built-in filtering scheme to avoid overfitting problems. These two characteristics permit the algorithm to generate compact neural architectures with very good generalisation capabilities, making the algorithm suitable for its application to devices with limited resources like microcontrollers. The main limitations of these devices are memory size and computing speed, and thus an efficient implementation of the algorithm is needed. Despite the existence of alternative evolving models (Lughofer, 2011; Angelov, 2010; Huang et al., 2005), C-Mantec has been selected based mainly on the three following features: dynamic generation of compact architectures, good prediction ability and robustness to parameter setting.

In the present work, we have fully implemented the C-Mantec (Subirats et al., 2012) constructive neural network model in an Arduino UNO board, including the whole learning process to implement the automatic reprogramming process for decision-making into the sensor/actuator in changing environments, avoiding communication to other devices.

The Arduino UNO board was used (Oxer and Blemings, 2009) as it is a popular, economic and efficient open source single-board microcontroller that allows easy project development (Lian et al., 2013; Cela et al., 2013; Ortega-Zamorano et al., 2013; Kornuta et al., 2013). We have also proposed three case studies of different nature which require reprogramming to the decision-making process, demonstrating that the time involved in the sensor reprogramming is significantly lower than in the traditional case, without the need to send any information to another device (as a control unit), saving a large fraction of the required energy resources.

The paper is structured as follows: first, we briefly describe in Section 2 the C-Mantec constructive neural network algorithm used, followed by a description of the Arduino UNO microcontroller board in Section 3. Section 4 includes the details of the implementation with the results obtained shown in Section 5. Thereafter, three case studies are evaluated in Section 6, checking the efficiency of the system in them, to finally extract the conclusions in Section 7.

2. C-Mantec, constructive neural network algorithm

C-Mantec (Subirats et al., 2012) (Competitive Majority Network Trained by Error Correction) is a novel neural network constructive algorithm that utilises competition between neurons and a modified perceptron learning rule (thermal perceptron Frean, 1990) to build single hidden layer compact architectures with good prediction capabilities for supervised classification problems. As a CoNN algorithm, C-Mantec generates the network topology on-line during the learning phase, avoiding the complex problem of selecting an adequate neural architecture. The novelty of C-Mantec in comparison to previous proposed constructive algorithms is that the neurons in the single hidden layer compete for learning the incoming data, and this process permits the creation of very compact neural architectures. The binary activation state (S) of the neurons in the hidden layer depends on N input signals, ψ_i, and on the actual value of the N synaptic weights (ω_i) and bias (b) as follows:

$$S = \begin{cases} 1(ON) & \text{if } h \geq 0 \\ 0(OFF) & \text{otherwise} \end{cases} \quad (1)$$

where h is the synaptic potential of the neuron defined as

$$h = \sum_{i=0}^{N} \omega_i \psi_i \quad (2)$$

In the thermal perceptron rule, the modification of the synaptic weights, $\Delta\omega_i$, is done on-line (after the presentation of a single input pattern) according to the following equation:

$$\Delta\omega_i = (t - S)\psi_i T_{fac}, \quad (3)$$

where t is the target value of the presented input, and ψ represents the value of input unit i connected to the output by weight ω_i. The difference in the standard perceptron learning rule is that the thermal perceptron incorporates the T_{fac} factor. This factor, whose value is computed as shown in Eq. (4), depends on the value of the synaptic potential and on an artificially introduced temperature (T).

$$T_{fac} = \frac{T}{T_0} e^{-|h|/T}, \quad (4)$$

The value of T decreases as the learning process advances according to Eq. (5), similar to a simulated annealing process.

$$T = T_0 \cdot \left(1 - \frac{I}{I_{max}}\right), \quad (5)$$

where I is a cycle counter that defines an iteration of the algorithm on one learning cycle, and I_{max} is the maximum number of iterations allowed. One learning cycle of the algorithm is the process that starts when a random chosen pattern is presented to the network and finishes after checking that the output of the network is equal to the target for this pattern, or when a chosen neuron (the neuron with largest T_{fac} value or a new added neuron) modifies its synaptic weights to learn the actual presented pattern.

The C-Mantec algorithm has three parameters to be set at the time of starting the learning procedure, and several experiments have shown the robustness of the algorithm that operates fairly well in a wide range of parameter values. The algorithm has the following three parameters:

- I_{max}: Maximum number of learning iterations allowed for each neuron in one learning cycle.
- g_{fac}: Growing factor that determines when to stop a learning cycle and includes a new neuron in the hidden layer.
- ϕ: Determines in which case an input example is considered as noise and removed from the training dataset according to the following condition:

$$\text{delete}(x_i) | N_{LT} \geq (\mu + \phi\sigma), \quad (6)$$

where x_i represents an input pattern, N is the total number of patterns in the dataset, N_{LT} is the number of times that pattern x_i has been presented to the network on the current learning cycle, and where μ and σ correspond to the mean and variance of the distribution for all patterns on the number of times that the algorithm has tried to learn each pattern in a learning cycle. The learning procedure starts with one neuron present in the single hidden layer of the architecture and an output neuron that computes the majority function of the responses of the hidden

neurons (a voting scheme). The process continues by presenting an input pattern to the network and if it is misclassified, it will be learned by one of the present neurons whose output did not match the target pattern value if certain conditions are met, otherwise a new neuron will be included in the architecture to learn it. Among all neurons that misclassified the input pattern, the one with the largest T_{fac} will learn it but only if this T_{fac} value is larger than the g_{fac} parameter of the algorithm, a condition included to prevent the unlearning of previous stored information. If no thermal perceptron meeting these criteria are found, a new neuron is added to the network, starting a new learning cycle that includes the resetting of all neurons temperature to T_0. Also at the end of a cycle the noisy patterns filtering procedure (Eq. (6)) is applied. The algorithm continues its operation iteratively repeating the previous stages until all patterns in the training set are correctly classified by the network. During the learning process catastrophic forgetting is prevented as synaptic weights are only modified if the change involved is small (controlled by the value of g_{fac} and by an annealing process that reduces the temperature as learning proceeds), as if this is not the case the algorithm introduces a new neuron in the architecture (Subirats et al., 2012).

The Flowchart of C-Mantec algorithm is shown in Fig. 1, the most relevant function is represented in boxes, the decision-making in diamonds and the most important states (start and finish) in ovals.

3. The Arduino UNO board

Arduino is a single-board microcontroller designed to make the process of using electronics in multidisciplinary projects more accessible. The hardware consists of a simple open source hardware board designed around an 8-bit Atmel AVR microcontroller, though a new model has been designed around a 32-bit Atmel ARM. The software consists of a standard programming language compiler and a boot loader that executes on the microcontroller.

Arduino is a descendant of the open-source *Wiring* platform and is programmed using a Wiring-based language (syntax and libraries); similar to C++ with some slight simplifications and modifications, and a processing-based integrated development environment. Arduino boards can be purchased pre-assembled or do-it-yourself kits, and hardware design information is available. The maximum length and width of the Arduino UNO board are 6.8 and 5.3 cm respectively, with the USB connector and power jack extending beyond the former dimension.

The Arduino UNO is based on the ATmega328 chip (Atmel, Datasheet 328). It has 14 digital input/output pins, which can be used as input or outputs, and in addition, has some pins for specialised functions, for example 6 digital pins can be used as PWM outputs. It also has 6 analogy inputs, each of which provide 10 bits of resolution, together with a 16 MHz ceramic resonator, USB connection with serial communication, a power jack, an ICSP header, and a reset button. The ATmega328 chip has 32 KB of memory storage (0.5 KB are used for the bootloader). It also has 2 KB of SRAM and 1 KB of EEPROM. A picture of the Arduino UNO board is shown in Fig. 2.

The Arduino UNO has a communication protocol for its interaction with a computer, with another Arduino board or other microcontrollers. The ATmega328 provides serial communication (UART) which is available on digital pins 0 (RX) and 1 (Tx), also has I2C and SPI communication.

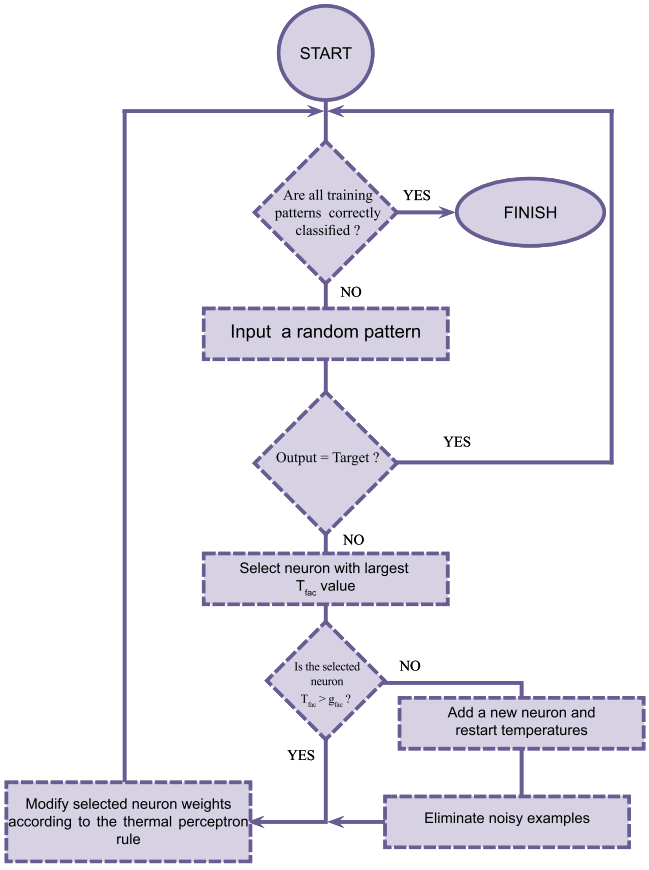

Fig. 1. Flow diagram of the C-Mantec constructive learning algorithm.

4. Implementation of the C-Mantec algorithm

The neural network model comprises two phases (learning and execution). In the learning phase, the synaptic weights for the neural network are computed from a set of patterns stored in the memory of the microcontroller, while in the execution phase, the microcontroller obtains the response to sensed input data according to the previously learned model. The learning phase comprises two different states (loading of input patterns and neural network learning). Data can be loaded into the microcontroller EEPROM memory on-line by I/O pins or by a serial communication USB port, but in both cases, the patterns have to be stored into the EEPROM memory, after, the neural network learning state starts.

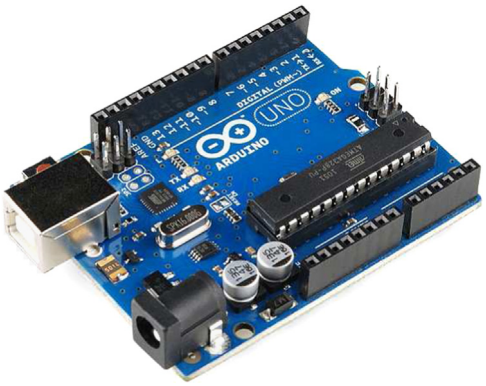

Fig. 2. Picture of an Arduino UNO board used for the implementation of the C-Mantec algorithm.

We explain next, the main technical issues considered for the implementation of the algorithm according to the two learning modes mentioned before:

4.1. Loading of patterns

The patterns have to be stored in the memory microcontroller because the learning process works in cycles and uses the pattern set repeatedly. For Boolean functions, it is only necessary to store the true value function outputs because the inputs are represented by the memory position. For example, for the pattern "01101001"→ '0', the input "01101001" corresponds to the decimal number 105 and thus the value '0' is stored in the position of memory 105. The microcontroller has 1 KB of EPROM memory, i.e., 8192 bits (2^{13}) limiting the number of Boolean inputs to 13. For the case of using an incomplete truth table, because the noisy pattern elimination stage is used or because of the nature of the function, the memory is divided into two parts, a first one that corresponds to the function outputs and a second part to indicate the inclusion or not of a given pattern in the learning set. In the case of using an incomplete truth table the maximum size of the input dimension is reduced to 12.

For the case of using real-value patterns is necessary to know in advance the actual number of bits that are used to represent each input variable. Eight bits have been used to represent each variable, taking into account that these values have to be normalised between 0 and 255. The following equation permits to compute the maximum number of input patterns:

$$N_P \cdot N_I + N_P/8 \leq 1024, \tag{7}$$

where N_I is the number of inputs and N_P is the number of patterns. N_P depends on the number of entries and the number of bits used for each entry.

4.2. Neural network learning

C-Mantec is an algorithm which adds neurons when they are become necessary, action that is not easily implemented in microcontroller, because the use of memory is static so the maximum number of neurons, which are stored in SRAM, must be previously defined. From this memory, with a capacity of 2 KB, we will employ less than 1 KB for storing the variables of the program; and thus saving at least 1 KB of free memory for saving the variables related to the neurons.

The microcontrollers are devices with limited computing speed so for obtaining more velocity in the learning process we have changed the data type of the variables associated to neurons. The floating point representation is the usual data type used in this kind of algorithm to represent all variables but this representation is not the most efficiency. We have selected fixed point to represent the variables associated to neurons for this we have to change the data type of these variables to integer. This paradigm shift has incited to essential changes in the way to program this algorithm but in return a higher learning speed and a smaller size of each variable has been obtained.

Regarding the variables associated to neurons, it is given next some details about the representation used:

- T_{fac}: Represented as a *float* type and occupying 4 bytes.
- Number of iterations: An integer value with a range between 1000 and 65,535 iterations, so a 2 bytes *integer* variable was used.
- Synaptic weights: Almost all calculations are based on these variables, so to speed up the computations an *integer* variable of 2 bytes long was used.
- Synaptic potential (h): It is calculated as a result of a summation of the synaptic weights, so not to saturate this value a type *long* of 4 bytes in length is used.

According to the previous definitions, the maximum number of neurons (N_N) that can be implemented should verify the following constraint:

$$4 \cdot N_N + 2 \cdot N_N + 2 \cdot N_N \cdot (N_I + 1) + 4 \cdot N_N \leq 1024, \tag{8}$$

where N_I is the number of inputs. For a worst case (maximum number of inputs is 13), the maximum number of neurons is 28. If during the learning process, the architecture reaches this maximum number of neurons, the algorithm finishes and the sensor outputs an error message.

The synaptic weights and synaptic potential have been implemented with 10 bits precision for the decimal part, so that the value of the weights will be between 32 and −32. Integer data types with values between −32,768 and 32,767 were used. The representation of the synaptic potential is done in a similar way, except that for this value 4 bits were used, allowing values between 2,097,152 and −2,097,152. The computation of T_{fac} is done using a float data type because it involves an exponential operation that can only be done with this type of data, but as its computation involves integer values, two different conversions must be done. The first conversion is done in Eq. (3), where T_{fac} values must be converted to fixed point representation, operation done by multiplying the T_{fac} value by 1024. The second conversion is performed in Eq. (4) where the synaptic potential (h) has to be converted to floating point representation for the calculation of an exponential function. In this case, ten right logical shifts were used in the synaptic potential, to then convert its data type to floating point for the final calculation of the T_{fac}. In order to avoid the saturation of the synaptic weights value, when any of these values are larger than 32 or lower than −32, all weights are divided by two (a right logical shift) when any of them reach an absolute value of 30. This change does not affect the algorithm execution because neural networks are invariant to this kind of scaling.

5. Results

We first tested the correct functioning of the microcontroller implementation of the C-Mantec algorithm comparing to the original published results (Subirats et al., 2012) in terms of the number of neurons generated in the neural network architectures, analysing as well the execution time for the microcontroller using floating point and integer representation. A set of 14 Boolean functions have been used for the comparison, including 12 single output functions from the MCNC circuits testing benchmark plus two XOR functions with two and three inputs. The C-Mantec algorithm was run with the following parameter configuration: $g_{fac}=0.05$ and $I_{max}=1000$. The results are shown in Table 1 where the first two columns indicate the function reference name and its number of inputs, and the third, fourth and fifth columns show that the number of neurons obtained in the original C-Mantec publication (Subirats et al., 2012), and the integer and floating point microcontroller implementation, respectively. Two last columns in Table 1 show the learning times in seconds (s) for the integer and floating point implementation of the microcontroller. The displayed values (mean ± standard deviation) are computed from 50 random samples.

Fig. 3 shows the learning time for all 14 benchmark functions included in Table 1 both for the integer and floating point implementation. The top graph in the figure shows on a logarithmic scale for the y-axis the learning time while the bottom graph shows the comparative speed between both representations used (integer and floating point).

Further, Fig. 4 shows the temporal evolution for the most complex analysed function (Alu2k), where the top graph shows the execution times related to the addition of a new neuron in the

Table 1
Number of neurons and learning time of the synthesis of a set of functions for fixed-point (integer) and floating point implementations compared with the original paper.

Funct.	# Inp.	# Neurons			Time (s)	
		Theory	Integer	Floating	Integer	Floating
XOR2	2	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	0.36 ± 0.01	0.57 ± 0.03
XOR3	3	3.0 ± 0.0	3.0 ± 0.0	3.0 ± 0.0	1.4 ± 0.11	2.22 ± 0.26
cm82af	5	3.0 ± 0.0	3.0 ± 0.0	3.0 ± 0.0	1.84 ± 0.28	3.15 ± 0.64
cm82ag	5	3.0 ± 0.0	3.6 ± 0.6	3.6 ± 0.7	4.11 ± 1.88	8.10 ± 4.41
cm82ah	5	3.0 ± 0.0	3.0 ± 0.0	3.0 ± 0.0	1.85 ± 0.21	3.35 ± 0.63
z4ml24	7	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	0.23 ± 0.05	0.47 ± 0.13
z4ml25	7	3.1 ± 0.7	3.1 ± 0.4	3.1 ± 0.4	3.36 ± 1.04	6.43 ± 2.05
z4ml26	7	3.0 ± 0.0	3.0 ± 0.0	3.0 ± 0.0	2.67 ± 0.45	5.39 ± 1.11
9symml	9	3.0 ± 0.0	3.0 ± 0.0	3.0 ± 0.0	4.43 ± 0.58	12.8 ± 2.6
alu2k	10	11.2 ± 0.9	12.7 ± 0.9	12.3 ± 0.7	220 ± 50	969 ± 260
alu2m	10	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	3.94 ± 0.21	13.1 ± 0.9
alu2n	10	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	0.41 ± 0.15	0.99 ± 0.41
alu2o	10	11.2 ± 0.9	13.0 ± 0.8	12.4 ± 0.8	312 ± 59	1444 ± 824
alu2p	10	3.0 ± 0.0	3.0 ± 0.0	3.0 ± 0.0	20.8 ± 43.7	84.1 ± 17.2

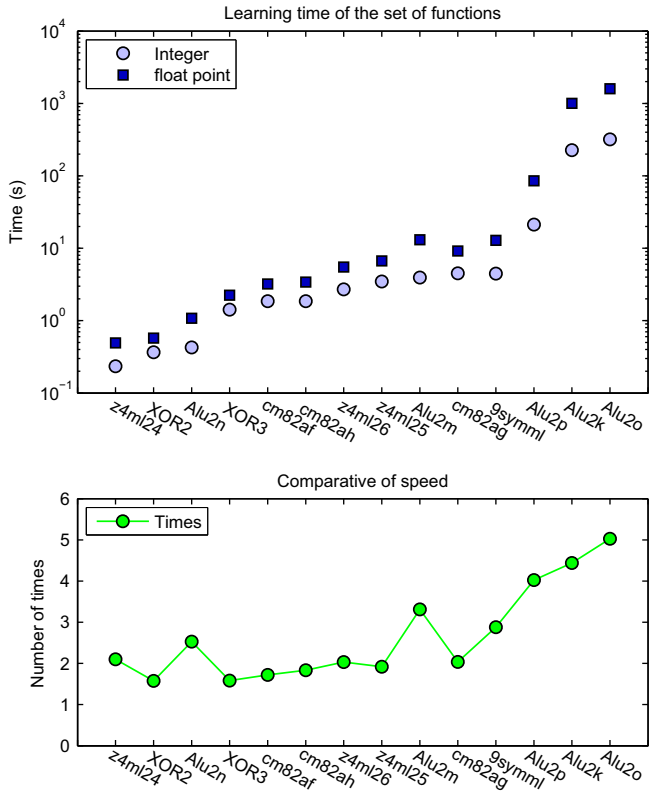

Fig. 3. Learning time (in seconds, logarithmic scale) for each function of the Boolean function benchmark data set with the two data type representation (top graph) and the relative speed between them (bottom graph).

constructive network architecture while Fig. 4 bottom shows the relative speed between the two implementations (integer and floating point). The time shown in the top graph of Fig. 4 includes several learning cycles until an input pattern wrongly classified by the network is presented and there is no neuron in the architecture that can be selected to learn it (according to the values of T_{fac} and g_{fac}).

The C-Mantec algorithm includes three procedures that are modified depending on the data type (floating or integer) representation used. These three functions that can be observed in Fig. 1 are the following: *Input a random pattern*, *Select neuron with largest* T_{fac} *value*, and *Modify selected neuron weights*. The algorithm

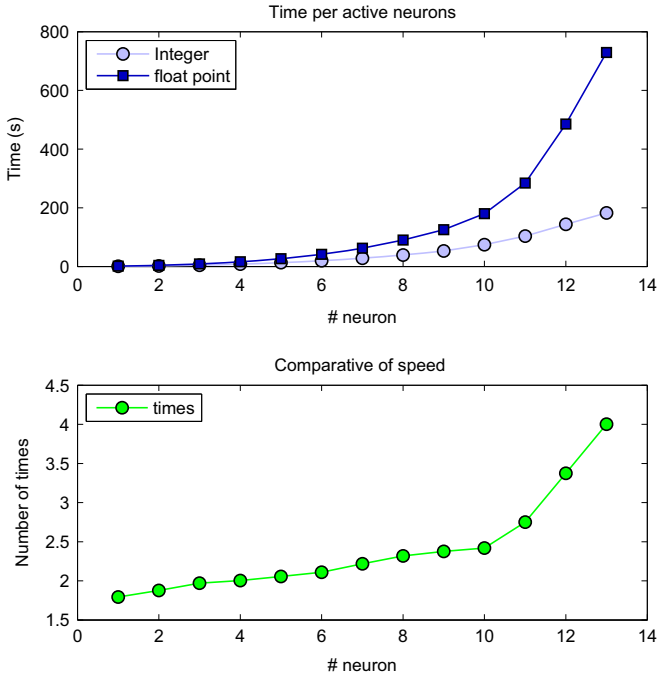

Fig. 4. Temporal evolution for the function Alu2k related to the addition of a new neuron to the network architecture. Absolute time (top graph) and relative time (bottom graph) for both representations (integer and floating point) used.

includes other two procedures (*Add a new neuron* and *Eliminate noisy examples*) that are not altered if the data representation is changed.

For each of the procedures that change according to the representation used we computed the mean execution time averaged across 50 samples for both cases. Fig. 5 shows the results for each procedure for integer and floating point representations. Fig. 5a shows the whole network execution time as the number of neurons increases from 1 to 20 for different input pattern dimensions (2, 5, 10, 15), while Fig. 5b shows a comparison between both implementations, where it can be appreciated the number of times that the integer representation is faster than the floating point one. Fig. 5c and d shows the execution time and relative comparison (respectively) for the computation of the maximum value of the T_{fac} for both representations as the number of neurons present in the architecture increases from 1 to 20. Fig. 5e and f shows the same as the previously described case but for the procedure *Modify selected neuron weights*.

6. Case studies

In this section, in order to test the efficiency of using neural networks in applications where microcontrollers for sense and/or act on the environment might be required, three case studies of different nature have been considered. The first case deals with detecting fires by an alarm system that can be reprogrammed according to the security level required. The second analysed case is a control system for the activation of a solenoid valve that depends on environmental variables, while the third case corresponds to a person fall detection problem with the constraint of using only a limited set of data patterns. The parameter settings of the C-Mantec algorithm were the same for all analysed cases: $g_{fac}=0.05$, $I_{max}=1000$ and $\phi=2$. We verified that changes on these values did not improve significantly the performance on each of the problems and then decided to use this set of parameter in all three cases. We note that it has been previously verified that

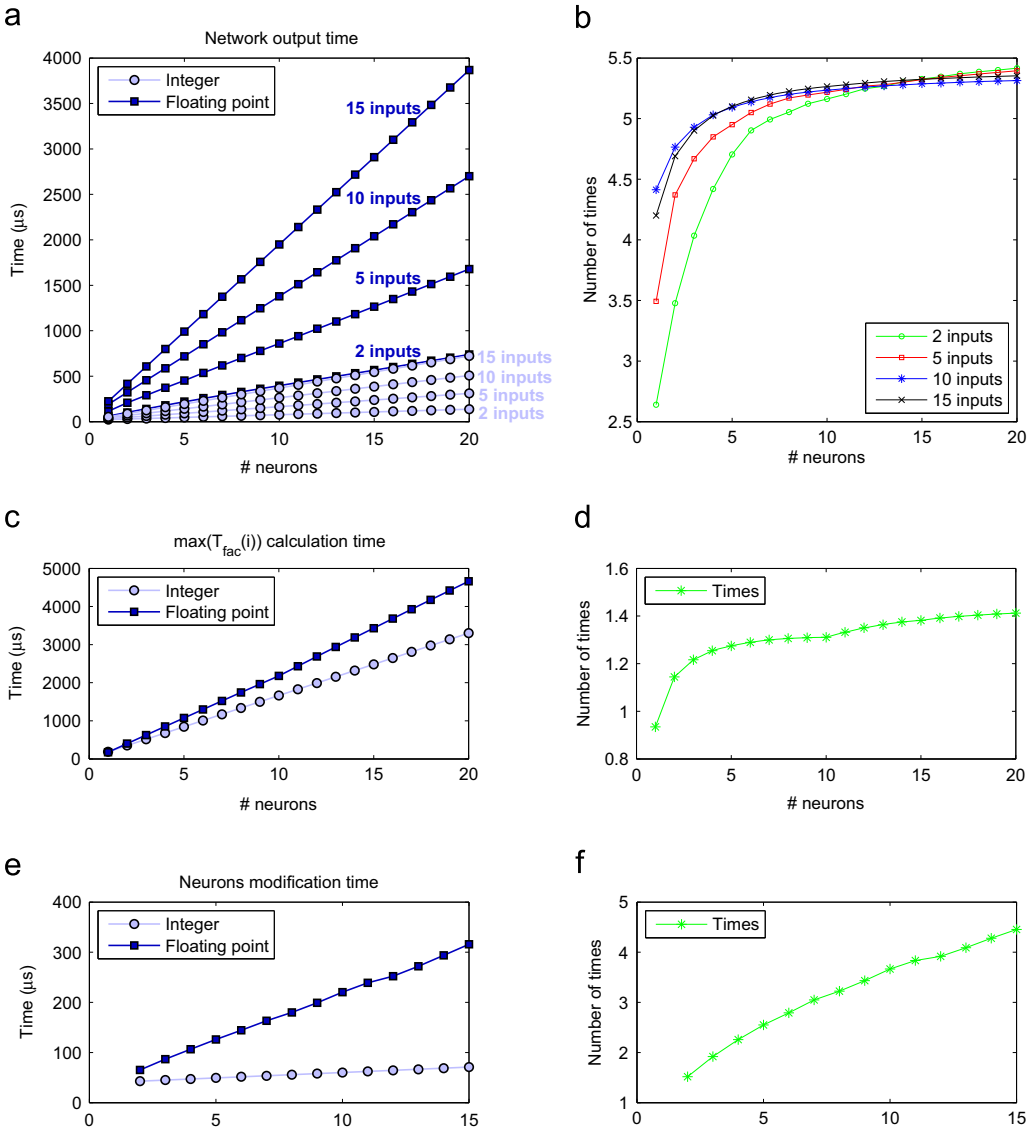

Fig. 5. Execution times of the three procedures included in the C-Mantec algorithm that changed depending on the data type representation used. Graphs in the first column show the absolute times measured in µs while the graphs in the second column show a comparison between both execution times (see the text for more details).

C-Mantec is very robust against changes on the parameter values (Subirats et al., 2012, 2013; Urda et al., 2013; Luque-Baena et al., 2013).

6.1. Fire alarm system

Fire alarm systems are widely used in inhabited premises. We analyse the case of a system whose security levels can be reprogrammed according to the user convenience, by defining activation thresholds as a function of the sensed environmental variables (temperature, smoke and gas). A schematic drawing of a room where a fire alarm is installed is shown in Fig. 6 including the three typical variables that can affect the system.

We have represented the different states of the fire alarm system using a truth table which is shown in Table 2.

Each different sensor (Gas, Smoke and Temperature) can be active ('1') or not ('0'), and these levels determine the Alarm state in the range from 1 to 8. As C-Mantec is a binary classifier, the alarm levels were codified in binary notation, as indicated in the last column of the table. A given functioning state of the alarm system includes setting the alarm levels for each of the detectors states, i.e., choosing the alarm level for a given sensor inputs.

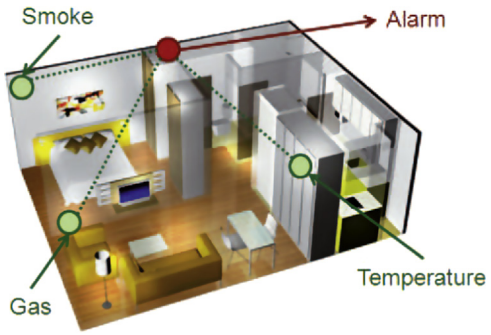

Fig. 6. Schematic representation of a room in which is installed a fire alarm.

Table 2 corresponds to a case in which each alarm level is different for every different input, as this is the worst possible case.

We first checked the time employed by the system to learn the initial state, measuring also the number of neurons included by the C-Mantec constructive algorithm and the energy consumption. Table 3 shows the results averaged across 50 samples, where

Table 2
Truth table of the initial state of the system for fire detection.

Gas	Smoke	Temperature	Alarm	Representation
0	0	0	1	0 0 0
0	0	1	2	0 0 1
0	1	0	3	0 1 0
0	1	1	4	0 1 1
1	0	0	5	1 0 0
1	0	1	6	1 0 1
1	1	0	7	1 1 0
1	1	1	8	1 1 1

Table 3
Time, energy consumption and number of neurons in the constructed neural network architecture for learning the initial state of a reprogrammable alarm system.

# Neurons	Time (ms)	Consumption (nAh)
3.0 ± 0.0	8.2 ± 4.8	73.1 ± 43.4

execution time is measured in milliseconds (ms) while energy consumption is measured in nano Amperes per hour (nAh).

Next, we have tested the time needed to relearn a modification of the state of the alarm, that corresponds to a change of values in the fourth column in Table 2. The results are shown in Table 4 where mean, maximum and minimum reprogramming time are displayed together with the measured consumption.

6.2. Weather prediction

Weather prediction is a relevant issue in human daily life, related for example for making good decisions in agriculture (moment of harvest, time of watering and crop type). Weather prediction is a very complex problem and neural network based system has been widely used for this task (Taylor and Buizza, 2002; Chakraborty et al., 2004).

We have consider a system that consists in a sensor/actuator that measures five environmental variables (Temperature (T), Wind speed (W_s), Wind direction (W_d), Humidity (H) and Solar Irradiance (I)) and takes a decision, which can be to irrigate or not the surrounding land. Fig. 7 displays a real picture of a constructed sensor/actuator node that may be used for the described problem.

To analyse a possible scenario where this node may be used, we consider the case of determine whether to water or not the surrounding land according to the value of the five environmental variables mentioned before. For the implementation these variables have been discretised using a 12 bits representation where the number of bits used for each variable can be seen from Table 5 together with the discretisation intervals.

The evaluation of the implementation of the neural network model for controlling the actuator system has been carried out starting from a null initial state. Afterwards, random input conditions were considered. An example of a generic instance can be:

$$\text{if}(T = 17.5\ ^\circ\text{C}) \wedge (W_s = 2\ \text{m/s}) \wedge (W_d = \text{“N”}) \wedge (H = 55\%) \wedge (I = 900\ \text{W/m}^2) \rightarrow \text{solenoid valve “open”}.$$

In the previous case, the indicated instance corresponds to a input of the truth table of the form “0100 00 00 10 11” with output ‘1’ as it was indicated that the solenoid valve, controlling the watering system, should open for the indicated condition. The whole truth table for the current discretisation used comprises 4096 different instances.

Table 4
Mean, maximum and minimum time involved in reprogramming a microcontroller and its corresponding energy consumption for a fire alarm system.

	Max	Min	Mean
Time (ms)	20.27	3.13	10.97
Consumption (nAh)	180.17	27.82	97.51

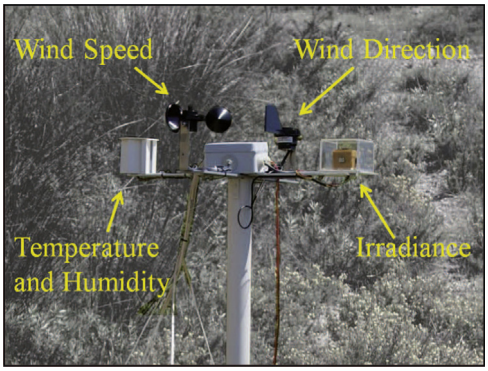

Fig. 7. Picture of a sensor/actuator system responsible for sensing different environmental variables and acting accordingly.

Table 5
Environmental variables used in a weather prediction problem, indicating the number of bits used in their representations and the discretisation intervals used.

Var.	# bits	Discretisation
T	4	[−2.5, 0, 2.5, 5, 7.5, 10, 12.5, 15, 17.5, 20, 22.5, 25, 27.5, 30, 32.5, 35] °C
W_s	2	[0, 2, 5, 9] m/s
W_d	2	N: [335°–45°), E: [45°–135°), S: [135°–225°), W: [225°–335°)
H	2	[5, 30, 55, 80]%
I	2	[100, 300, 600, 900] W/m^2

We measured the evolution of the time needed by the system as the number of defined instances is increased, together with the level of accuracy. Fig. 8 top shows time and consumption averaged across 30 random executions (Mean) and also one randomly chosen execution of the system in order to appreciate the existing variability of the learning process. Accuracy, computed as the fraction of correct responses over the whole truth table is shown in table Fig. 8 bottom. Note that to compute the accuracy, we suppose that the final whole truth table is known at every given time, even if the actuator only gets this information one pattern at a time, implying that in practice the accuracy (as it is computed) can be only analysed at the end of the process. If we had computed the accuracy over the presented pattern, this would have always been equal to 1.

A singular behaviour in the initial instances can be observed in Fig. 8, the neural network model has to be modified because almost all instances are misclassified thus at the beginning of the process, the learning time is higher and the accuracy increases because the number of classified instances grows. When the model has already learned almost all instances the learning time grows linearly depending on the number of instances as the results showed, and thus the accuracy is one in practice. In this moment if a instance are misclassified then the neural network model have to learn this instances and the learning time is the largest as Fig. 8 top shows for one execution.

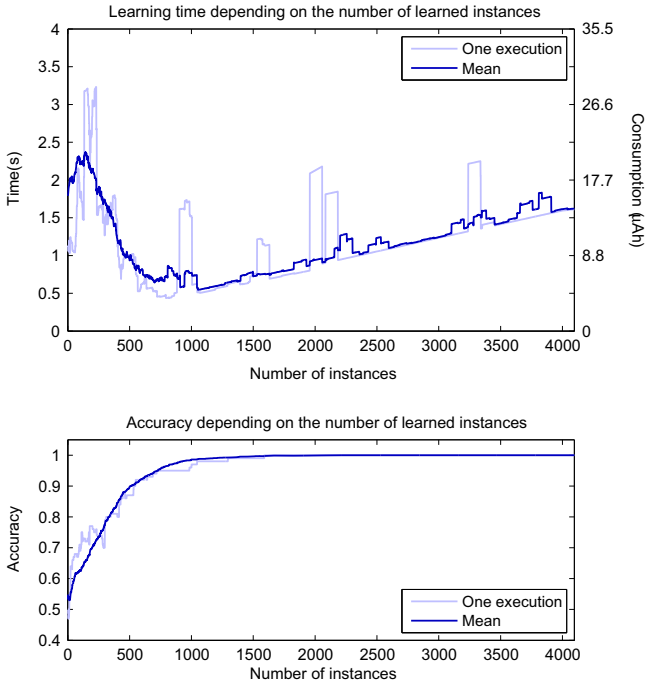

Fig. 8. Learning time, consumption (top graph) and accuracy (bottom graph) of a sensor/actuator used for an automatic watering system. (See text for details).

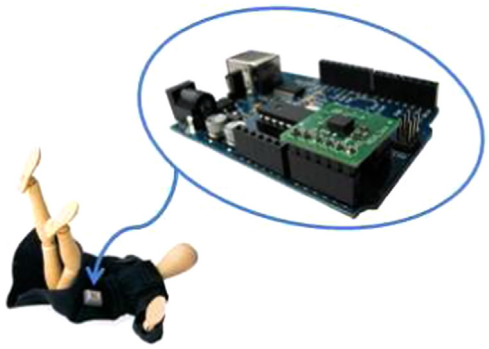

Fig. 9. A sensor based system attached to a person body for the implementation of a person fall detection system.

6.3. Fall detection system

Monitoring elderly or disabled people is in growing demand in modern societies, mainly because these people are dependent and unable to care for themselves. In particular, one important problem is to detect efficiently when a person falls. Different ways of addressing this problem have been proposed, like using a video surveillance systems, although if such system has the drawback of a strong sensitivity to light changes. A more efficient system to detect a person fall can be constructed using a sensor attached to the person body. The sensor, in this case, should include a 3-axis accelerometer together with a microcontroller that analyses the person relative inclination angle and movement, in order to detect abnormal situations, such as a strong downward movement or any sudden or violent displacement. Fig. 9 shows a schematic drawing of the described system.

Deducing the logic which describes the behaviour of the falls is a complex and very time consuming task. However, using a neural network based system simplifies enormously this task as a training process based on observed patterns can be implemented.

In such a system, the accelerometer senses the position of the device, acquiring the position in relationship to axes x, y, z. These coordinates are then sent to the microcontroller, in values normalised between 0 and 255 so they can be stored in a variable of "byte" type. A movement is considered a fall when in a short period of time (1 s), the position of the device changes from an initial state (vertical) to a final state (horizontal). Thus, a pattern to the system consists in the initial and final position of the movement plus the output that indicates whether the person has suffered or not a fall ($(x_0, y_0, z_0, x_1, y_1, z_1) \rightarrow fall?$). The number of patterns that can be stored on the system is delimited by Eq. (7), that in the present case leads to a maximum number of patterns to be stored equal to 167 patterns. As this number is quite small for such a complex problem, only patterns that a "supervisor" considered wrong will be used. For example, a pattern is sensed every second and the system determines whether or not a fall has occurred. If the supervisor (that during the training of the system can be the person carrying the device) resolves that the decision of the neural network model is wrong, this pattern is stored in memory and a new neural network model is calculated. The previous description is a modification to the standard C-Mantec algorithm that includes a more efficient storage of patterns for maximising the use of the limited memory resources of a microcontroller. For testing the system, a sensor node has been attached to a subject producing common movements such as sitting down, walking and stopping. The microcontroller has periodically gathered accelerometer data every second and 30,000 patterns have been obtained as data set.

Afterwards, the neural network based systems were checked, starting the process with no patterns in the memory, to only store one instance of the problem when the classification obtained does not match the supervisor output, that in our case, was the person wearing the system.

Fig. 10 top shows the time evolution of the system as the number of patterns presented increase. In the figure, the mean of

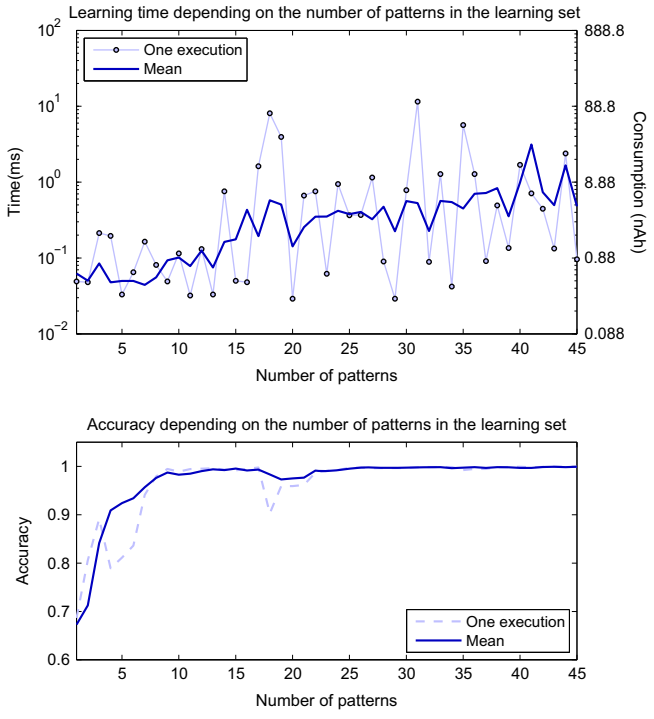

Fig. 10. Training time (top) and accuracy (bottom) of a fall detection system as a function of the number of observed patterns. Each graph includes average results over 30 independent samples, together with results corresponding to a single case.

30 executions is displayed together with a single case to observe the level of variability existing in the problem. Fig. 10 bottom shows the level of accuracy obtained as the patterns are presented to the system (the graph also includes the mean and a single observed case).

It is worth mentioning that the way this problem is treated is different from the previous two cases, as patterns are stored in the microcontroller memory only if they are misclassified.

7. Conclusions

The C-Mantec constructive neural network algorithm has been successfully implemented in a microcontroller board, adapting it to overcome the limitations imposed by the limited resources of memory and computing speed of the hardware device. The correct implementation of the algorithm has been verified in comparison to the original published results, obtaining that as the number of inputs is increased, the microcontroller implementation needs a small number of extra neurons and it is possible to observe a little reduction in performance accuracy due to rounding effects. Furthermore, a thorough comparison of the differences of using floating point or fixed precision representations has be carried out, concluding that better results can be obtained with an 8-bit fix precision representation leading to computation times approximately five times faster than using the standard floating point representation.

The implemented algorithm has been employed as a sensor/actuator system and applied to three case studies in order to demonstrate the efficiency and versatility of the resulting application. The three case studies chosen are problems defined in changing environments, and thus the decision-making of the sensor/actuator has to be adapted accordingly to the observed changes, thus needing a retraining of the neural network model that controls the decision process.

The observed reprogramming times are significantly low in the three case studies, being the energy consumption of the device also quite small, and even if a comparison to the traditional case in which the new code has to be transmitted from a central control unit has not been analysed, the results suggest a very important potential reduction.

As an overall conclusion, we have shown the suitability of C-Mantec for its application in a dynamic task using an Arduino UNO microcontroller. Nowadays, given the existence of devices with much more powerful computing resources than the considered board, the present study confirms the potential of the proposed algorithm for its application in real tasks where sensors/actuators are needed.

Acknowledgements

The authors acknowledge support from Junta de Andalucía through Grant Nos. P10-TIC-5770 and P08-TIC-04026, and from CICYT (Spain) through Grant No. TIN2010-16556 (all including FEDER funds).

References

Aiello, F., Bellifemine, F.L., Fortino, G., Galzarano, S., Gravina, R., 2011. An agent-based signal processing in-node environment for real-time human activity monitoring based on wireless body sensor networks. Eng. Appl. Artif. Intell. 24, 1147–1161.

Aleksendrić, D., Jakovljević, I., Irović, V., 2012. Intelligent control of braking process. Expert Syst. Appl. 39.

Angelov, P., 2010. Evolving Takagi–Sugeno fuzzy systems from data streams (eTS+). In: Angelov, P., Filev, D., Kasabov, N. (Eds.), Evolving Intelligent Systems. John Wiley and Sons and IEEE Press, IEEE Press Series in Computational Intelligence, pp. 21–50

Atmel, Datasheet 328. http://www.atmel.com/Images/doc8161.pdf.

Canete, E., Chen, J., Luque, R., Rubio, B., 2012. Neuralsens: a neural network based framework to allow dynamic adaptation in wireless sensor and actor networks. J. Netw. Comput. Appl. 35, 382–393.

Cela, A., Yebes, J.J., Arroyo, R., Bergasa, L.M., Barea, R., López, E., 2013. Complete low-cost implementation of a teleoperated control system for a humanoid robot. Sensors 13, 1385–1401.

Chakraborty, S., Ghosh, R., Ghosh, M., Fernandes, C.D., Charchar, M.J., Kelemu, S., 2004. Weather-based prediction of anthracnose severity using artificial neural network models. Plant Pathol. 53, 375–386.

Dhanalakshmi, P., Palanivel, S., Ramalingam, V., 2011. Pattern classification models for classifying and indexing audio signals. Eng. Appl. AI 24, 350–357.

Farooq, U., Amar, M., ulHaq, E., Asad, M.U., Atiq, H.M., 2010. Microcontroller based neural network controlled low cost autonomous vehicle. In: Proceedings of the 2010 Second International Conference on Machine Learning and Computing, IEEE Computer Society, Washington, DC, USA, pp. 96–100.

Franco, L., Elizondo, D., Jerez, J., 2009. Constructive Neural Networks. Springer-Verlag, Berlin

Frean, M., 1990. The upstart algorithm: a method for constructing and training feedforward neural networks. Neural Comput. 2, 198–209.

Gómez, I., Franco, L., Jerez, J., 2009. Neural network architecture selection: can function complexity help?. Neural Process. Lett. 30, 71–87.

Han, C.C., Kumar, R., Shea, R., Srivastava, M., 2005. Sensor network software update management: a survey. Int. J. Netw. Manag. 15, 283–294.

Haykin, S., 1994. Neural Networks: A Comprehensive Foundation. Prentice Hall

Huang, G.B., Saratchandran, P., Sundararajan, N., 2005. A generalized growing and pruning rbf (ggap-rbf) neural network for function approximation. IEEE Trans. Neural Netw. 16, 57–67.

Hunter, D., Hao, Y., Pukish, M., Kolbusz, J., Wilamowski, B., 2012. Selection of proper neural network sizes and architectures – a comparative study. IEEE Trans. Ind. Appl. 8, 228–240.

Kodogiannis, V.S., Boulougoura, M., Wadge, E., Lygouras, J.N., 2007. The usage of soft-computing methodologies in interpreting capsule endoscopy. Eng. Appl. AI 20, 539–553.

Kopetz, H., 1997. Real-Time Systems: Design Principles for Distributed Embedded Applications, 1st edition Kluwer Academic Publishers, Norwell, MA, USA

Kornuta, J.A., Nipper, M.E., Dixon, Brandon, 2013. Low-cost microcontroller platform for studying lymphatic biomechanics in vitro. J. Biomech. 46 (1), 183–186.

Lakshmi, K., Subadra, M., 2013. A survey on FPGA based MLP realization for on-chip learning. Int. J. Sci. Eng. Res., 1–9.

Lian, K.Y., Hsiao, S.J., Sung, W.T., 2013. Intelligent multi-sensor control system based on innovative technology integration via zigbee and WI-FI networks. J. Netw. Comput. Appl. 36, 756–767.

Lughofer, E., 2011. Evolving Fuzzy Systems – Methodologies, Advanced Concepts and Applications. Studies in Fuzziness and Soft Computing, vol. 266. Springer, http://dx.doi.org//10.1007/978-3-642-18087-3.

Luque-Baena, R.M., Urda, D., Subirats, J.L., Franco, L., Jerez, J.M., 2013. Analysis of cancer microarray data using constructive neural networks and genetic algorithms. In: Rojas, I., Guzman, F.M.O. (Eds.), Proceedings of the IWBBIO, International Work-Conference on Bioinformatics and Biomedical Engineering, pp. 55–63.

Mahmoud, S., Lotfi, A., Langensiepen, C., 2013. Behavioural pattern identification and prediction in intelligent environments. Appl. Soft Comput. 13, 1813–1822.

Mamdoohi, G., Fauzi Abas, A., Samsudin, K., Ibrahim, N.H., Hidayat, A., Mahdi, M.A., 2012. Implementation of genetic algorithm in an embedded microcontroller-based polarization control system. Eng. Appl. Artif. Intell. 25, 869–873.

Marwedel, P., 2006. Embedded System Design. Springer-Verlag, New York, Inc., Secaucus, NJ, USA

Ortega-Zamorano, F., Subirats, J.L., Jerez, J.M., Molina, I., Franco, L., 2013. Implementation of the C-Mantec neural network constructive algorithm in an arduino UNO microcontroller. Lect. Notes Comput. Sci. 7902, 80–87.

Oxer, J., Blemings, H., 2009. Practical Arduino: Cool Projects for Open Source Hardware. Apress, Berkeley, CA, USA

Park, K., Shin, H., 2013. Stock price prediction based on a complex interrelation network of economic factors. Eng. Appl. AI 26, 1550–1561.

Rassam, M., Zainal, A., Maarof, M., 2013. An adaptive and efficient dimension reduction model for multivariate wireless sensor networks applications. Appl. Soft Comput. 13, 1978–1996.

Sayed-Mouchaweh, M., Lughofer, E., 2012. Learning in Non-Stationary Environments: Methods and Applications. Springer, New York

Sengupta, S., Das, S., Nasir, M., Panigrahi, B.K., 2013. Multi-objective node deployment in WSNS: in search of an optimal trade-off among coverage, lifetime, energy consumption, and connectivity. Eng. Appl. Artif. Intell. 26, 405–416.

Shaikh, R., Thakare, V., Dharaskar, R., 2010. Efficient code dissemination reprogramming protocol for WSN. Int. J. Comput. Netw. Secur. 2, 116–122.

Subirats, J., Franco, L., Jerez, J., 2012. C-mantec: a novel constructive neural network algorithm incorporating competition between neurons. Neural Netw. 26, 130–140.

Subirats, J.L., Baena, R.M.L., Urda, D., Ortega-Zamorano, F., Jerez, J.M., Franco, L., 2013. Committee C-Mantec: a probabilistic constructive neural network. Lect. Notes Comput. Sci. 7902, 339–346.

Taylor, J.W., Buizza, R., 2002. Neural network load forecasting with weather ensemble predictions. IEEE Trans. Power Syst. 17, 626–632.

Urda, D., Canete, E., Subirats, J.L., Franco, L., Llopis, L., Jerez, J.M., 2012. Energy-efficient reprogramming in WSN using constructive neural networks. Int. J. Innov. Comput. Inf. Control 8, 7561–7578.

Urda, R.M., Luque, M.J., Jiménez, I., Turias, L. Franco, Jerez, J. M., 2013 A constructive neural network to predict pitting corrosion status of stainless steel. Lecture Notes in Computer Science, 7902, pp. 88-95, (2013). ISBN: 978-3-642-38678-7.
Wang, J., Xu, W., Gong, Y., 2010. Real-time driving danger-level prediction. Eng. Appl. Artif. Intell. 23, 1247–1254.
Wang, Q., Zhu, Y., Cheng, L., 2006. Reprogramming wireless sensor networks: challenges and approaches. Netw. IEEE 20, 48–55.
Yick, J., Mukherjee, B., Ghosal, D., 2008. Wireless sensor network survey. Comput. Netw. 52, 2292–2330.
Zhai, Y.J., Yu, D.L., 2009. Neural network model-based automotive engine air/fuel ratio control and robustness evaluation. Eng. Appl. Artif. Intell. 22, 171–180.

Capítulo 5

Conclusiones y líneas de trabajo futuras

5.1. Conclusiones

Una vez estudiadas las diferentes implementaciones neurocomputacionales realizadas para los dos dispositivos hardware analizados (FPGA y microcontroladores), se exponen tanto las conclusiones derivadas del trabajo realizado en esta tesis doctoral como las posibles líneas de investigaciones futuras.

Sistemas neurocomputacionales en tiempo real

A la hora de implementar sistemas neurocomputacionales en tiempo real sobre FPGAs existen al menos dos estrategias posibles: modificar un algoritmo ya existente y optimizar su implementación con el objetivo de maximizar el uso de recursos, y una segunda opción que consiste en desarrollar un algoritmo nuevo que se adapte a las características de los dispositivos. A tal efecto se han analizado dos algoritmos diferentes, Backpropagation y C-Mantec, como exponentes de estas estrategias para estudiar su idoneidad y rendimiento en este tipo de sistemas.

El algoritmo de Backpropagation se ha implementado de forma eficiente en una FPGA bajo un paradigma de aprendizaje "on-chip", incluyendo además un sistema de validación del error para evitar el efecto de sobreajuste, e incorporando un diseño con modificaciones estructurales que permite minimizar los recursos utilizados. Se ha introducido un nuevo tipo de neurona, denominada "primera capa" que incorpora tanto las funciones de los elementos de entrada ("inputs"), como de las neuronas de la primera capa oculta, reduciendo drásticamente los recursos necesarios en su desarrollo, principalmente en arquitecturas con un gran número de entradas. Además, se han realizado mejoras adicionales en el diseño como son la multiplexación por división en el tiempo del bloque multiplicador y la interpolación de valores tabulados para aproximar la función sigmoidea. Estas modificaciones han permitido un ahorro de más de un 25 % en el uso de celdas lógicas y de un 50 % en bloques específicos en comparación con los resultados obtenidos en otros trabajos publicados en relación a la implementación del mismo algoritmo. La principal ventaja de las reducciones obtenidas es que permiten un incremento sustancial en el número máximo de neuronas que pueden crearse en la arquitectura de red neuronal. Un análisis detallado de la implementación realizada y de

los recursos de la placa utilizada muestra que la principal limitación en cuanto al número máximo de neuronas que pueden incorporarse viene dada por el número de bloques específicos DSP de la placa FPGA ya que se precisa un bloque específico DSP para cada una de las neuronas, haciéndose notar que de todos los bloques DSP existentes un bloque debe guardarse para su uso en el proceso de validación.

A fin de probar el correcto diseño de la implementación en FPGA, se han comparado los valores obtenidos en el proceso de entrenamiento de la red neuronal frente a los valores correspondientes de una implementación en un ordenador utilizando programación tradicional. Se observa que el error cuadrático medio evoluciona de forma similar para una serie de conjuntos de datos de prueba en ambos dispositivos. Los errores en la placa FPGA para el subconjunto de entrenamiento son ligeramente superiores a los obtenidos mediante el PC debido a los efectos producidos en el redondeo ocasionado por el tipo de representación de datos, pero sin embargo el modelo resultante final no se ve afectado como demuestran los valores obtenidos para los errores del subconjunto de test. En este primer análisis realizado se aprecia que los tiempos de entrenamiento son aproximadamente 100 veces menores en el caso de las implementaciones en FPGA.

Como conclusión de la estrategia en implementaciones hardware de algoritmos conocidos, decir que en el caso considerado fue posible incorporar modificaciones al algoritmo de Backpropagation para adecuarlo al dispositivo utilizado sin afectar la estructura del modelo, consiguiéndose optimizar los recursos disponibles y reducir drásticamente los tiempos de cómputo.

Como una alternativa al algoritmo de Backpropagation se ha estudiado un modelo de red neuronal de tipo constructivo para su implementación hardware en una FPGA. El algoritmo seleccionado ha sido C-Mantec, ya que posee una muy buena capacidad de predicción y permite seleccionar la arquitectura de forma automática conforme se realiza el proceso de entrenamiento. Este algoritmo se ha implementado de manera íntegra (con estructura de aprendizaje "on-chip") en una placa FPGA. Con el fin de evaluar su correcto diseño, los valores obtenidos en generalización para una serie de conjuntos de datos booleanos frecuentemente utilizados han sido comparados con los valores teóricos, constatando, además del correcto funcionamiento por parte de la implementación en FPGA, una disminución del tiempo empleado en el aprendizaje en relación al PC. Un examen pormenorizado de los tiempos de cómputo refleja que el número de veces que una FPGA es más rápida que un PC es proporcional al tamaño de la arquitectura resultante. Este interesante resultado puede entenderse en base a que el tiempo de cómputo en un ordenador crece de forma exponencial conforme se añaden neuronas a la capa oculta, mientras que lo hace de manera lineal en una placa FPGA, dando lugar a implementaciones hasta 47 veces más rápidas en los problemas más complejos.

Varias pruebas realizadas sobre los conjuntos de datos anteriormente mencionados muestran que la representación de datos utilizada (representación en punto fijo) es en la mayoría de los casos suficiente para lograr resultados comparables a los ejecutados en un PC con representación en punto flotante desde el punto de vista de la capacidad de generalización del algoritmo y del tamaño de las arquitecturas resultantes, siendo este algoritmo muy robusto en cuanto al tamaño de palabra utilizado en la representación de los datos.

En comparación con una implementación hardware del algoritmo de Backpropagation, la implementación del algoritmo C-Mantec resulta un 15 % más eficiente en cuanto a los recursos hardware utilizados; analizando los resultados se confirma que

las arquitecturas permitidas para el algoritmo C-Mantec son mayores en número de neuronas que las permitidas para el algoritmo Backpropagation en la misma placa FPGA. Además, las ejecuciones del algoritmo C-Mantec precisan una menor longitud en la representación de los datos para ser precisas, por lo que se constata que dicha implementación es menos sensible al tamaño de palabra utilizada, lo que supone un ahorro de recursos al reducir la complejidad de la implementación. Para finalizar la comparación destacar que el tiempo empleado para el aprendizaje de un conjunto de datos es inferior en las dos implementaciones hardware en comparación al empleado en un PC, destacando que es sensiblemente inferior para el algoritmo C-Mantec. Además, el tiempo de ejecución de la red donde sólo se calcula la salida de la red sin el proceso de aprendizaje es notablemente menor para el algoritmo C-Mantec, lo que supone una ventaja en la fase de explotación del modelo.

A la luz de los resultados observados se puede concluir que el presente análisis demuestra la idoneidad del algoritmo C-Mantec como modelo neurocomputacional para este tipo de dispositivos. Además, este estudio demuestra las ventajas potenciales de las tarjetas FPGA para actuar aceleradores hardware para su aplicación en problemas industriales del mundo real que precisen de la implementación de modelos neurocomputacionales.

Redes de sensores inteligentes

Se ha investigado también en esta tesis doctoral la posibilidad de incluir los dos algoritmos mencionados anteriormente (Backpropagation y C-Mantec) como posibles soluciones a modelos neurocomputacionales en redes de sensores inteligentes con los que dotar de cierto razonamiento el proceso de toma de decisiones necesaria para adaptarse a entornos cambiantes. Para ello se analizan las mejores estrategias a la hora de implementar redes neuronales artificiales en una placa microcontroladora. Se ha seleccionado para llevar a cabo los experimentos la placa Arduino por su versatilidad y eficiencia, siendo además de código abierto.

Los algoritmos Backpropagation y C-Mantec han sido implementados en el microcontrolador con estructura de aprendizaje "on-chip". Se ha cambiado el paradigma de representación de datos del tradicional punto flotante usado en este tipo de dispositivos a la representación en punto fijo. Este cambio ha reducido considerablemente la memoria utilizada para el almacenamiento de variables, permitiendo que el número de neuronas máximas del que disponer para la construcción de la arquitectura de red se vea incrementado en ambos algoritmos.

Las correctas implementaciones de los algoritmos en un microcontrolador han sido verificadas con diferentes conjuntos de datos, un conjunto para cada algoritmo, en los que los resultados obtenidos en el aprendizaje han sido comparados con los resultados teóricos para cada conjunto. Se observa una pequeña reducción de la precisión en la generalización de los datos, debido principalmente a los efectos producidos por el redondeo ocasionado por la representación del tipo de datos sin que ello condicione la eficacia del modelo neurocomputacional resultante. Además, para el caso concreto del algoritmo C-Mantec para el cual la arquitectura de red se determina de forma automática se observa que las arquitecturas resultantes son ligeramente más grandes a las obtenidas en un PC conforme crece el número de entradas, efecto también causado por el redondeo. Este incremento no afecta como ya se ha visto a la generalización de los datos pero repercute negativamente en la memoria utilizada sin que suponga un

perjuicio significativo.

Se han analizado detenidamente los tiempos de cómputo del aprendizaje para los dos tipos de representación de datos en ambos algoritmos. Para el Backpropagation se consiguen incrementos en la velocidad de entre 8 y 18 veces más rápido para las representaciones en punto fijo (8 bits en parte decimal) mientras que para el algoritmo C-Mantec se consiguen mejoras de hasta cinco veces más velocidad en comparación con la de punto flotante. La razón de este incremento se debe a que el tamaño de la arquitectura de la red repercute negativamente en la velocidad de cómputo de forma más severa para las representaciones de punto flotante ya que el manejo de memoria y el uso de la unidad aritmético lógica de este tipo de representaciones son más complejas. Estos resultados posibilitan estructuras de aprendizaje "on-chip" también en este dispositivo para diseñar sistemas neurocomputacionales en el propio microcontrolador, permitiendo su implantación en sensores remotos que precisan de un modo de trabajo autónomo.

Además, el algoritmo C-Mantec se ha empleado en una red de sensor/actuador para tres casos de estudios con el fin de demostrar la eficiencia y la versatilidad del sistema resultante. Los casos de estudios son problemas definidos en entornos cambiantes por tanto la toma de decisiones del sensor/actuador ha de adaptarse en consecuencia a los cambios observados, necesitando una reconversión del modelo de red neuronal que controla el proceso de decisión.

Los tiempos de reprogramación observados son relativamente bajos en los tres casos de estudio siendo, en consecuencia, el consumo de energía del dispositivo también bastante reducido. Incluso sin una comparación exhaustiva con el caso tradicional en el que el nuevo código tiene que ser transmitido desde una unidad de control central, los resultados hacen evidente una reducción muy importante en el gasto energético, cualidad muy importante en este tipo de tecnología debida a la corta duración de las baterías que lo alimentan. Los resultados han demostrado la idoneidad del algoritmo C-Mantec para su aplicación en tareas reales donde se precisen sensores/actuadores, notándose que se ha utilizado un microcontrolador Arduino UNO y que su utilización en casos reales es altamente factible dada la existencia en el mercado de dispositivos con prestaciones superiores a la placa considerada.

Conclusiones generales

Como conclusión global de esta tesis doctoral se puede afirmar que los resultados obtenidos en ella dan una visión detallada de implementaciones hardware de modelos neurocomputacionales en dos tecnologías muy extendidas como son los sistemas en tiempo real y las redes de sensores. Después del análisis exhaustivo realizado se puede concluir que las ANN pueden utilizarse en diversas aplicaciones de las tecnologías citadas, empleando dispositivos más apropiados que los ordenadores convencionales. Se ha demostrado que es posible la implementación de diferentes modelos en FPGAs y microcontroladores consiguiendo unos resultados muy superiores a los mismos modelos desarrollados en un PC.

Además se ha demostrado que si bien modificaciones efectuadas en los algoritmos conocidos los hacen más eficientes para su implementación en dispositivos específicos, existen otros modelos de redes neuronales más adecuados para el tipo de dispositivo analizado que los tradicionales con los que es posible construir arquitecturas de mayor tamaño y obtener tiempos de respuesta más cortos.

5.2. Líneas de trabajo futuras

Dada la importancia de las tecnologías estudiadas en esta tesis, las posibles líneas de investigación y aplicaciones derivadas son numerosas, algunas ya han comenzado a desarrollarse por el grupo de investigación y otras pueden llevarse a cabo en el futuro para ampliar y mejorar los resultados de la tesis. Algunos ejemplos de futuros trabajos son:

- Progresar en la implementación hardware del algoritmo Backpropagation con el fin de poder utilizar arquitecturas de red con N capas ocultas, donde N es un número elevado no predefinido. Para ello se precisa diseñar una sola capa oculta que haga la funcionalidad de todas las capas y que permita la iteración de dicha capa (N veces) para simular la arquitectura completa. Esta implementación podría ser utilizada junto con un algoritmo evolutivo para encontrar arquitecturas de red óptimas en función de diferentes conjuntos de entrada.

- Analizar la posibilidad de usar otros modelos neurocomputacionales con diferentes reglas de aprendizaje para diseñar nuevas toma de decisiones en redes de sensores inteligentes. El mapa auto-organizado (SOM: Self-Organizing Map) es un tipo de ANN no supervisado cuyo entrenamiento produce una representación discreta del espacio de las muestras de entrada. Este modelo genera la red en función de la estructura de los datos de entrada sin necesidad de tener un conocimiento previo de su salida con lo que no precisa guardar el conjunto de datos con el consiguiente ahorro de recursos y tiempo en el uso de memoria.

- Investigar la posibilidad de utilizar las placas FPGAs como acelerador hardware de otros sistemas complejos con una necesidad de cómputo muy elevada. El modelo de Ising es un modelo físico propuesto para estudiar el comportamiento de materiales ferromagnéticos que evalúa el comportamiento de cada partícula en función de sus vecinas. En sistemas con interacciones de largo alcance, en los que se necesite conocer la evolución de una partícula a partir del conjunto total de las mismas, la implementación algorítmica en una FPGA puede reducir drásticamente el tiempo de modelado del sistema.

- Explorar la posibilidad de realizar nuevas aplicaciones con redes de sensores inteligentes, como la utilización de modelos neurocomputacionales para el control de huertos urbanos que precisan de un control exhaustivo de los recursos hídricos. Cada semilla y cada planta necesitan un riego diferente en función de las condiciones microclimáticas y en este escenario los modelos neurocomputacionales podrían suprimir la necesidad de estudios previos costosos, proporcionando un control automático para un sistema de plantación complejo.

- Analizar la posibilidad de utilizar sistemas de neurocomputación en tiempo real en áreas complejas, como los sistemas de estabilización de un cuadricóptero que permita una rápida adaptación a cambios estructurales en el dispositivo en caso, por ejemplo, de pérdida de un rotor o de condiciones de vuelo bajo inclemencias meteorológicas.

Capítulo 6

Conclusions and future lines of research

6.1. Conclusions

We have analyzed in this thesis the implementation of two different neural network models (C-Mantec and Back-propagation algorithms) in FPGA and microcontroller devices, and now we present in this chapter the final conclusions and possible future extensions of the work done.

Neurocomputational models for real time systems

At the time of the implementation of neurocomputational models for real time systems on a FPGA board there are at least two possible strategies: adapt an existing neural model to the hardware resources trying to optimize resource usage, and secondly develop a new algorithm taking into account the characteristics of the FPGA. With these two different strategies in mind, we have analyzed in this work the Backpropagation and C-Mantec neural models, studying their adaptation and performance for FPGA boards.

The Backpropagation algorithm has been efficiently implemented in a FPGA board using an "on-chip" learning scheme that includes a validation process to avoid overfitting; and several optimization strategies have been developed in order to minimize resource usage. In particular, a new type of neuron named "Inp-Hid" layer has been created. This new element incorporates the functions from both the input elements and the first hidden layer neurons, reducing drastically the needed resources, especially in architectures with a large number of inputs. Furthermore, the use of a time-division multiplexing scheme for implementing a multiplier block, together with a scheme based on tabulation plus interpolation of values for the computation of the sigmoid function has enabled a saving of more than 25 % in the number of logic cells used and 50 % in specific blocks in comparison to previous published results. An important aspect of the implemented modifications is that they allow for a substantial increase in the maximum number of neurons that can be included in a network architecture. An analysis of the developed implementation in terms of the constraints imposed by the available resources of the used FPGA board (Xilinx Virtex V) indicates that the main limitation regarding the maximum number of neurons that can be created is the number of DSP

specific blocks included in the board, as each neuron requires one DSP block for its implementation.

In order to check for the correct implementation of the Backpropagation algorithm in the FPGA board, a comparison with the original PC implementation of the algorithm is carried out. It is observed that the mean square error evolves in the same manner in both cases, existing small differences as training errors are lower for the PC implementations, fact that can be justified by rounding errors present in the FPGA implementation that do not affect the predictive accuracy. Regarding computational times, training a neural model using the FPGA code is about 100 times faster than using the PC one.

As a partial conclusion regarding hardware implementations based on existing algorithms we say that neural models should be adapted and modified to suit the device but taking care of not affecting the model structure. In particular in our specific case, it has been possible to maximize the available resources and at the same time reduce significantly the computational times.

As an alternative to the traditional Backpropagation algorithm, we have studied a second neurocomputational model for its FPGA hardware implementation. The selected algorithm is C-Mantec, a neural network constructive model that generates the architecture automatically as training is performed leading to compact neural architectures with very good predictive capabilities. The algorithm has been completely implemented ("on-chip" learning scheme) in a FPGA board, adapting it specifically for this device. The correct implementation of the algorithm has been checked by analyzing and comparing the training and generalization error curves for well-known Boolean data sets, observing besides its proper operation a decrease in the training times in relation to the PC implementation. A detailed examination of these times shows that the number of times an FPGA is faster than a computer is proportional to the size of the resultant architecture, and as the computing time for a PC grows exponentially as neurons are added to the hidden layer while linearly for the FPGA, in some cases, results up to 47 times faster can be observed for the more complex architectures. Several tests with the previously mentioned data sets confirm that the used data type representation (fixed point) is sufficient to achieve similar results to those obtained from a PC using a floating point representation.

In comparison with the hardware implementation of the Backpropagation algorithm, C-Mantec performance is 15 % more efficient in terms of used hardware resources permitting to create for a given FPGA board larger neural networks. In addition, the execution of the C-Mantec algorithm needs shorter length for the data representation in order to operate correctly, indicating as expected that C-Mantec is less sensitive to number precision than Backpropagation. To complete the comparison between both algorithms FPGA implementation training time are significantly lower for the C-Mantec algorithm, and in particular the runtime of the network (after the training phase is done) is significantly lower for this algorithm, an important advantage at the exploitation phase of the algorithm.

The observed results confirm the suitability of the C-Mantec algorithm as a valid neurocomputational model for its implementation in FPGA boards, demonstrating the potential advantages of FPGAs to act as hardware accelerator devices for application to real-world industrial problems.

Smart sensor networks

Smart wireless sensor networks is another type of technology that we have analyzed for the application of neurocomputational models, with the aim of providing intelligence to the decision making process that it is needed in order for systems to adapt to changing environments. The implementations of the two previously studied algorithms have been analyzed in order to demonstrate their suitability for the application in sensor networks. Backpropagation and C-Mantec algorithms have been implemented in an Arduino board with "on-chip" learning scheme. The data type representation has been changed from traditional floating-point representation used in this type of algorithms to fixed-point representation with the idea of reducing memory storage and in order to build as large as possible neural architectures. To verify the correct implementation two data sets have been employed and the obtained results have been compared with the theoretical results. A small reduction in accuracy is observed for the generalization of data sets due mainly to the effects of rounding in the data type representation without altering the effectiveness of the resultant neurocomputational model. In addition, for the specific case of the C-Mantec algorithm where the network architecture is generated automatically it can be observed that the resultant architectures are slightly larger than for the PC implementation, being this effect caused by rounding. Nevertheless, the increase does not affect the generalization ability. Further, we have carefully analyzed the learning computational times for the two types of data representation in both algorithms. An increase in speed between 8 and 18 is obtained for the Backpropagation algorithm, while up 5 times improvement is obtained for the C-Mantec algorithm using 8 bits in the decimal part of the fixed point representation in comparison to the floating point one.

The obtained results enable to build "on-chip" learning schemes that permits to implement the neurocomputational models in the microcontroller, that allows for their use in wireless sensor networks in autonomous mode. Furthermore, the C-Mantec algorithm has been employed in a sensors/actuators network for three case studies to demonstrate the efficiency and versatility of the resultant application. Case studies are defined problems in changing environments where a sensor/actuator should be adjusted accordingly to the observed changes, requiring a re-training of the neural network model that controls the decision process. The observed reprogramming times are significantly short in the three cases, and therefore the power consumption of the device is also quite low. Even without a thorough comparison with the traditional case where the new code should be transmitted from a central control unit, the results highlight a noteworthy reduction in energy supply, very important quality in this type of technology due to the short life of batteries. As part of result, it has demonstrated the suitability of C-Mantec algorithm to operate with using a microcontroller Arduino UNO on concrete applications for complex tasks. Even more, given the availability of similar devices with much more hardware resources, this study confirms the potential of the proposed algorithms for their application in real tasks where sensors/actuators are required.

Final Conclusion

As an overall conclusion of this thesis, it can be said that the problems analyzed together with the obtained results give a complete overview of the features to take into account for carrying out neurocomputational implementations in the two studied technologies (real-time systems and sensor networks). After the detailed analysis done,

it can be concluded that neurocomputational models can be used in real-time systems and sensor networks applications, in cases where is better to use alternative devices rather than traditional PCs. It has been shown that it is possible to implement different neural network models on FPGA boards and microcontrollers obtaining much faster computational times than when the same models are implemented on a PC.

Further, it has also been demonstrated that while adaptations made to known algorithms (the Backpropagation algorithm in our case) can make them much more efficient for their implementation in non PC hardware device, the use of alternative models like C-Mantec can be more suitable for FPGAs and microcontrollers, permitting to build larger network architectures with shorter response times.

6.2. Lines of future research

Given the variety of applications which can be used according to the studied implementations, many possible lines of research can be carried on in the future. Some of them have already begun and others could start in the future in order to improve the exposed data. Some items could be the following:

- Progressing in the hardware implementation of the Backpropagation algorithm in order to allow architectures of neural network of N hidden layers, where N is a non-predetermined large number. This process requires designing a single hidden layer with the functionality of every layer, and then this layer would be able to simulate the whole architecture with N iterations. This implementation could be used with an evolutionary algorithm in order to generate optimal network architectures for different input data sets.

- Analyzing the possibility of using other neurocomputational models with other learning rules to design decision-making in smart sensor networks. A self-organizing map (SOM), that is trained using unsupervised learning, seems a suitable neural network for these devices because the model does not require knowing the output of data sets since the model is modified according to the structure of these sets. So it is not necessary to store the above data, being a saving of resources and time by not using the memory and its management.

- Investigating the possibility of using FPGAs as hardware accelerator boards of other complex systems with a need for very high computation. The Ising model is proposed to study the behavior of ferromagnetic materials that evaluates the behavior of each particle as a function of its neighbors'physical model. In systems that need to know the status of all the neighbors to know the behavior of a single particle, processing can take a huge amount of time. A application based on an FPGA system can reduce the time taken to model the ferromagnetic behavior.

- Exploring the possibility of new real applications to networks of intelligent sensors. An application startup is to use neurocomputational models for the control of urban gardens that require a very thorough control of water resources. Every seed and every plant require different irrigation based on microclimatic conditions; neurocomputational models eliminate the need for a prior study and can make automatic control of such a complex system of plantation.

- Analyzing the possibility of using neurocomputational systems in real time on complex areas. One possibility would be to develop a system of stabilization in quadricopter that allows quick adaptation to structural changes in the device such as the loss of a rotor or inclement weather.

Apéndice A

Implementación del algoritmo de red neuronal constructivo en un microcontrolador Arduino UNO

Francisco Ortega-Zamorano, José Luis Subirats, José M. Jerez, Ignacio Molina, Leonardo Franco: Implementation of the C-Mantec Neural Network Constructive Algorithm in an Arduino Uno Microcontroller. **Lecture Notes in Computer Science** 7902, pp. 80-87, (2013). ISBN: 978-3-642-38678-7.

Resumen:

Un algoritmo constructivo propuesto recientemente de red neuronal, denominado C-Mantec, se diseña de forma íntegra (proceso de aprendizaje incluido) para una placa microcontroladora Arduino UNO. Dicho algoritmo genera arquitecturas de red muy compactas con buenas capacidades de predicción que lo hacen idóneo para ser implementado en un microcontrolador a fin de ser usado como dispositivo neurocomputacional sin necesidad de transmitir información a una unidad de control central para efectuar el proceso de aprendizaje.

Se detalla la implementación de los procesos más complejos y se realiza un análisis del correcto funcionamiento de la aplicación resultante mediante la verificación del aprendizaje de un conjunto de datos de referencia usados normalmente en el diseño de circuitos.

Implementation of the C-Mantec Neural Network Constructive Algorithm in an Arduino Uno Microcontroller

Francisco Ortega-Zamorano[1], José Luis Subirats[1], José Manuel Jerez[1], Ignacio Molina[2], and Leonardo Franco[1]

[1] Universidad de Málaga, Department of Computer Science, ETSI Informática, Spain
{fortega,jlsubirats,jja,lfranco}@lcc.uma.es
[2] Max Planck Institute, Munich, Germany
imol@uma.es

Abstract. A recently proposed constructive neural network algorithm, named C-Mantec, is fully implemented in a Arduino board. The C-Mantec algorithm generates very compact size neural architectures with good prediction abilities, and thus the board can be potentially used to learn on-site sensed data without needing to transmit information to a central control unit. An analysis of the more difficult steps of the implementation is detailed, and a test is carried out on a set of benchmark functions normally used in circuit design to show the correct functioning of the implementation.

Keywords: Constructive Neural Networks, Microcontroller, Arduino.

1 Introduction

Several technologies like Wireless Sensor Networks [1], Embedded Systems [2] and Real-time Systems [3] are nowadays being extensively used in all kind of industrial applications, most of which use microcontrollers [4] to implement. The recent advances in the computing power of this kind of systems are starting to permitting the use of learning systems, that are able to adjust its functioning as the input data is received, to manage the microcontrollers present in their structure. Neural networks [5] are a kind of flexible and widely used learning systems that are natural candidates for this task as they are very flexible. Nevertheless a disadvantage of neural networks is that learning needs intensive computing power and tends to be prohibitive even for modern systems. In this sense, a recently proposed neural network constructive algorithm has the advantage of being very fast in comparison to standard neural network training and further it creates very compact neural architectures that is useful given the limited memory resources of the microcontrollers.

In this work the C-Mantec[6] algorithm has been fully implemented in a microcontroller, as the training process is part of the software of the controller and it is not carried out externally as it is usually done. We have chosen the Arduino

I. Rojas, G. Joya, and J. Cabestany (Eds.): IWANN 2013, Part I, LNCS 7902, pp. 80–87, 2013.

UNO board [7] as it is a popular, economic and efficient open source single-board microcontroller. C-Mantec is a neural network constructive algorithm designed for supervised classification tasks. One of the critical factors at the time of the implementation of the C-Mantec algorithm is the limited resources of memory of the microcontroller used (32 KB Flash, 2KB RAM & 1KB EPROM memory) and in this sense the implementation has been done with integer arithmetic except for one of the parameters of the algorithm. The paper is structured as follows: we first, briefly describe the C-Mantec algorithm and the Arduino board, secondly we give details about the implementation of the algorithm, to finish with the results and the conclusions.

2 C-Mantec, Constructive Neural Network Algorithm

C-Mantec (Competitive Majority Network Trained by Error Correction) is a novel neural network constructive algorithm that utilizes competition between neurons and a modified perceptron learning rule (termal perceptron) to build compact architectures with good prediction capabilities. The novelty of C-Mantec is that the neurons compete for learning the new incoming data, and this process permits the creation of very compact neural architectures. The activation state (S) of the neurons in the single hidden layer depends on N input signals, ψ_i, and on the actual value of the N synaptic weights (ω_i) and the bias (b) as follows:

$$y = \begin{cases} 1(ON) & \text{if } \phi \geq 0 \\ 0(OFF) & \text{otherwise} \end{cases} \tag{1}$$

where ϕ is the synaptic potential of the neuron defined as:

$$\phi = \sum_{i=1}^{N} \omega_i \; \psi_i - b \tag{2}$$

In the thermal perceptron rule, the modification of the synaptic weights, $\Delta\omega_i$, is done on-line (after the presentation of a single input pattern) according to the following equation:

$$\Delta\omega_i = (t - S) \; \psi_i \; T_{fac} \tag{3}$$

Where t is the target value of the presented input, and ψ represents the value of input unit i connected to the output by weight ω_i. The difference to the standard perceptron learning rule is that the thermal perceptron incorporates the factor T_{fac}. This factor, whose value is computed as shown in Eq. 4, depends on the value of the synaptic potential and on an artificially introduced temperature (T) that is decreased as the learning process advances.

$$T_{fac} = \frac{T}{T_0} e^{-\frac{|\phi|}{T}} \tag{4}$$

C-Mantec, as a CNN algorithm, has in addition the advantage of generating online the topology of the network by adding new neurons during the training

phase, resulting in faster training times and more compact architectures. The C-Mantec algorithm has 3 parameters to be set at the time of starting the learning procedure. Several experiments have shown that the algorithm is very robust against changes of the parameter values and thus C-Mantec operates fairly well in a wide range of values. The three parameters of the algorithm to be set are:

- I_{max}: maximum number of iterations allowed for each neuron present in the hidden layer per learning cycle.
- g_{fac}: growing factor that determines when to stop a learning cycle and include a new neuron in the hidden layer.
- F_{itemp}: determines in which case an input example is considered as noise and removed from the training dataset according to the following condition:

$$\forall X \in \{X_1, X_2, ..., X_N\}, delete(X) \mid NTL \geq (\mu + F_{itemp} \cdot \sigma), \tag{5}$$

where N represents the number of input patterns of the dataset, NTL is the number of times that the pattern X has been presented to the network on the current learning cycle, and the pair $\{\mu, \sigma\}$ corresponds to the mean and variance of the normal distribution that represents the number of times that each pattern of the dataset has been learned during the learning cycle. This learning procedure is essentially based on the idea that patterns are learned by those neurons, the thermal perceptrons in the hidden layer of the neural architecture, whose output differs from the target value (wrongly classified the input) and for which its internal temperature is higher than the set value of g_{fac}. In the case in which more than one thermal perceptron in the hidden layer satisfies these conditions at a given iteration, the perceptron with the highest temperature is the selected candidate to learn the incoming pattern. A new single neuron is added to the network when there is no thermal perceptron that complies with these conditions and a new learning cycle starts.

3 The Arduino UNO Board

The Arduino Uno is a popular open source single-board microcontroller based on the ATmega328 chip [8]. It has 14 digital input/output pins, which can be used as input or outputs, and in addition, has some pins for specialized functions, for example 6 digital pins can be used as PWM outputs. It also has 6 analog inputs, each of which provide 10 bits of resolution, together with a 16 MHz ceramic resonator, USB connection with serial communication, a power jack, an ICSP header, and a reset button. The ATmega328 chip has 32 KB (0.5 KB are used for the bootloader). It also has 2 KB of SRAM and 1 KB of EEPROM. Arduino is a descendant of the open-source *Wiring* platform and is programmed using a Wiring-based language (syntax and libraries); similar to C++ with some slight simplifications and modifications, and a processing-based integrated development environment. Arduino boards can be purchased pre-assembled or do-it-yourself kits, and hardware design information is available. The maximum length and width of the Uno board are 6.8 and 5.3 cm respectively, with the USB connector and power jack extending beyond the former dimension. A picture of the Arduino UNO board is shown in Fig. 1.

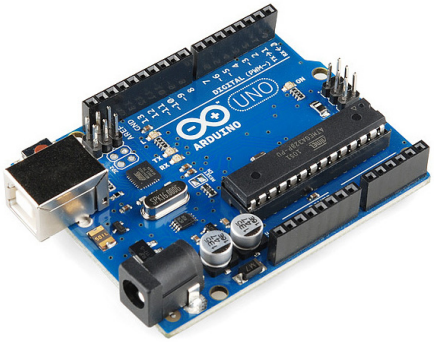

Fig. 1. Picture of an Arduino UNO board used for the implementation of the C-Mantec algorithm

4 Implementation of the C-Mantec Algorithm

The C-Mantec algorithm implemented in the wiring code is transferred by USB from the development framework from the PC to the board. The execution of the algorithm comprises two phases or states, because first, the patterns to be learnt have to be loaded into the EEPROM, and then the neural network learning process can begin. The microcontroller state is selected using a digital I/O pin. We explain next, the main technical issues considered for the implementation of the algorithm according to the two phases mentioned before:

4.1 Loading of Patterns

It is necessary to have the patterns stored in the memory board because the learning process work in cycles and use the pattern set repeateadly. The truth (output) value of a given Booelan pattern is stored in the memory position that corresponds to the input. For example, for the case of pattern of 8 inputs, the input pattern "01101001" that corresponds to the decimal number 105 and has a truth value of 0 would be stored by saving a value of 0 in the EEPROM memory position 105. The Arduino Uno EPROM has 1KB of memory, i.e., 8192 bits (2^{13}) and thus this limits the number of Boolean inputs to 13. For the case of using an incomplete truth table, the memory is divided into two parts, a first one to identify the pattern output and a second part to indicate its inclusion or not in the learning set. In this case, of an incomplete truth table, the maximum number of inputs is reduced to 12.

For the case of using real-valued patterns is necessary to know in advance the actual number of bits that will be used to represent each variable. If one byte is used to represent each variable then from the following equation the maximum number of input patterns permitted can be computed:

$$N_P \cdot N_I + N_P/8 \leq 1024\,, \tag{6}$$

where N_I is the number of inputs and N_P is the number of patterns. N_P depends on the number of entries and the number of bits used for each entry.

4.2 Neural Network Learning

C-Mantec is an algorithm which adds neurons as they become necessary, action that is not easily implemented in microcontroller, so we decided to set a value for the maximum number of permitted neurons, that will be stored in the SRAM memory. From this memory, with a capacity of 2 KB, we will employ less than 1 KB for storing the variables of the program; and thus saving at least 1 KB of free memory for saving the following variables related to the neurons:

- T_{fac}: must be a variable of *float* type and occupies 4 bytes.
- Number of iterations: an integer value with a range between 1000 and 100000 iterations, so it must be of type *long*, 4 bytes.
- Synaptic weights: almost all calculatios are based on these variable, so to speed up the computations we choose *integer* types of 2 bytes long.

According to the previous definitions, the maximum number of neurons (N_N) that can be implemented should verify the following constraint:

$$4 \cdot N_N + 2 \cdot N_N + 2 \cdot N_N \cdot (N_I + 1) \leqslant 1024, \tag{7}$$

where N_I is the number of inputs. For the maximum number of permitted inputs (13), the maximum number of neurons is 30. The computation of T_{fac} is done using a float data type because it requires an exponential operation that can be done only with this type of data, but as its computation involves other data types (integers) , a conversion must be done. To make this change without losing accuracy, we multiply the value of T_{fac} by 1000, leading to values in the range between 0 and 1000. When we convert to integer data type, precision is lost starting from the fourth digital number. Weights are of integer type in the range from -32768 to 32767, and as they are multiplied by the value of T_{fac}, we compensate this change by dividing them by 1000. When any synaptic weight value is greater than 30, or less than -30, all weights are divided by 2. This change does not affect at all the procedure of the network as neural network are invariant to this type of rescaling. To avoid the overflow of the integer data type, we apply the previous transformation whenever a synaptic weight reach the maximum or minimum permitted values. One very important thing in the implementation is the the execution time needed by the algorithm. In our case, this value depends strongly on the number of neurons actually used, as this time grows exponentially as a function of the number of used neurons. Fig 2 shows the execution time as a function of the neurons used in an architecture generated by the C-Mantec algorithm.

5 Results

We have tested the correct implementation of the C-Mantec algorihtm in the Arduino board by comparing the obtained results, in terms of the number of neurons generated and the generalization accuracy obtained, with those previously observed when using the PC implementation. The test is also carried out

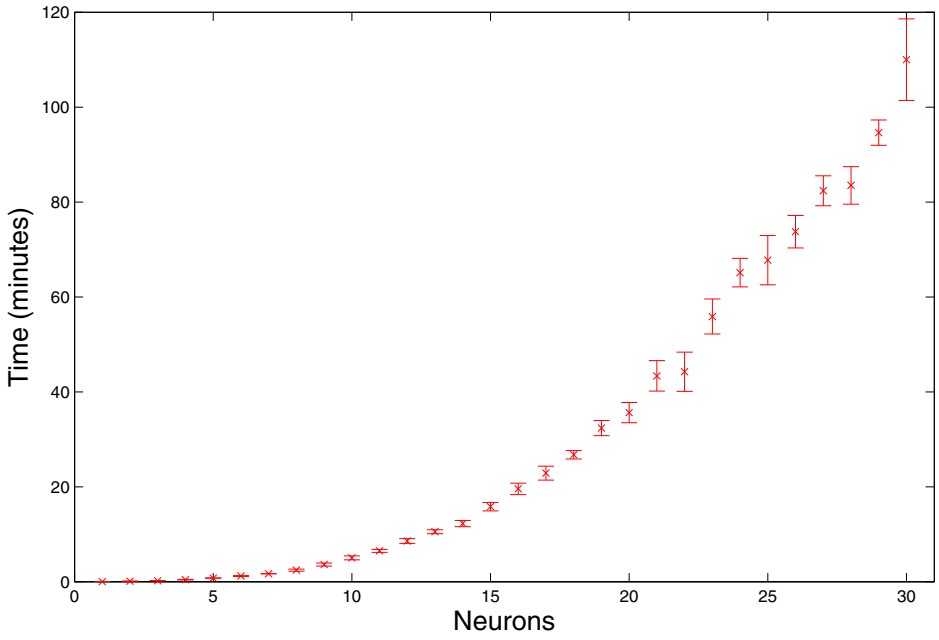

Fig. 2. Mean and standard deviation (indicated by error bars) of the execution time of the learning process as a function of the number of neurons used in a network created by the C-Mantec algorithm. The values shown are averages across 20 samples.

Table 1. Number of neurons and generalization ability obtained for a set of benchmark function for the implementation of the C-Mantec algorithm in an Arduino Uno board. (See text for more details).

Function	# Inputs	# Neurons		Accuracy generalization	
		Theory	Arduino	Theory	Arduino
cm82af	5	3,0±0,0	3,0±0,0	93,3±11,1	87,2±5,3
cm82ag	5	3,0±0,0	3,0±0,0	60.0±37,3	72,5±12,3
cm82ah	5	1,0±0,0	3,0±0,0	100.0±0,0	95,3±4,7
z4ml24	7	3,0±0,0	3,0±0,0	98,3±3,7	97,9±1, 1
z4ml25	7	3,1±0,9	3,1±0,9	90,8±12,3	86,0±0,9
z4ml26	7	3,0±0,0	3,0±0,0	96,7±5,9	94,6±0,4
z4ml27	7	3,0±0,0	3,0±0,0	99,2±2,8	99,9±0,9
9symml	9	3,0±0,0	3,0±0,0	99,4±0,9	97,5±1,2
alu2k	10	11,2±0,9	11,8±1,2	97,4±1,9	95,5±0,9
alu2l	10	18,9±1,5	19,3±1,3	79,2±5,5	70,3±1,3
alu2o	10	11,2±0,9	12,8±0,2	90,2±2,3	85,8±2,2

to analyze the effects of using a limited precision representation for the synaptic weights. A set of 10 single output Boolean functions from the MCNC benchmark were used to test the generalization ability of the C-Mantec algorithm. The C-Mantec algorithm was run with the following parameter values: $g_{fac} = 0.05$ and $I_{max} = 10000$. Table 1 shows the results obtained with the microcontroller for the

set of benchmark functions. The first two columns indicate the function reference name and its number of inputs. Third and fourth columns shows the number of neurons obtained by the PC and Arduino implementations, while fifth and last column shows the generalization ability obtained both for the PC and Arduino cases. The averages are computed from 20 samples and the standard deviation is indicated. The generalization ability shown in the table was computed using a ten-fold cross validation procedure.

6 Conclusion

We have successfully implemented the C-Mantec neural network constructive algorithm in an Arduino Uno board. The main issues at the time of the implementation are related to the memory limitations of the board. In this sense, we have analyzed the maximum number of Boolean and Real patterns that can be used for the learning process. For the case of Boolean patterns, we carried out a comparison against published results, showing that the algorithm works almost exact in comparison to the original PC implementation. As the number of inputs of the test functions increases, the Arduino implementation needs just a small extra number of neurons, and also a small degradation in the generalization accuracy is observed. These effects can be related to the limited numerical precision of the synaptic weights. The rounding effects should not in principle degrade the functioning of the algorithm, but affects the number of iterations needed to achieve convergence. Thus, we have also analyzed an important factor as it is the execution time of the algorithm. The results (cf. Figure 2) shows an exponential execution time increase as a function of the number of neurons in the constructed algorithms, and so for networks of approximately 15 neurons the execution time is around 20 minutes, while for 30 neurons this time increases up to two hours.

As a conclusion, and despite the previously mentioned limitations, we believe that the current implementation can be used in several practical applications, and we are planning to incorporate the C-Mantec algorithm in WSN in a near future.

Acknowledgements. The authors acknowledge support from Junta de Andalucía through grants P10-TIC-5770 and P08-TIC-04026, and from CICYT (Spain) through grant TIN2010-16556 (all including FEDER funds).

References

1. Yick, J., Mukherjee, B., Ghosal, D.: Wireless sensor network survey. Comput. Netw. 52(12), 2292–2330 (2008)
2. Marwedel, P.: Embedded System Design. Springer-Verlag New York, Inc., Secaucus (2006)
3. Kopetz, H.: Real-Time Systems: Design Principles for Distributed Embedded Applications, 1st edn. Kluwer Academic Publishers, Norwell (1997)

4. Andersson, A.: An Extensible Microcontroller and Programming Environment. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science (2003)
5. Haykin, S.: Neural networks: a comprehensive foundation. Prentice Hall (1994)
6. Subirats, J.L., Franco, L., Jerez, J.M.: C-mantec: A novel constructive neural network algorithm incorporating competition between neurons. Neural Netw. 26, 130–140 (2012)
7. Oxer, J., Blemings, H.: Practical Arduino: Cool Projects for Open Source Hardware. Apress, Berkely (2009)
8. Atmel: Datasheet 328, `http://www.atmel.com/Images/doc8161.pdf`

Apéndice B

Implementaciones FPGA de alta precisión de funciones de transferencia de redes neuronales

Francisco Ortega-Zamorano, José M. Jerez, Gustavo Júarez, Jorge O. Pérez and Leonardo Franco: High precision FPGA implementation of neural network activation functions. **Proceedings of the IEEE Symposium Series on Computatitional Intelligence (SSCI'2014),**pp. 55-60, (2014). ISBN: 978-1-4799-4486-6.

Resumen:

Las implementaciones hardware de modelos neurocomputacionales en FPGAs requieren afrontar varios problemas que afectan en gran medida al resultado final del sistema para disponer de aplicaciones eficientes. Uno de los más evidentes es el cálculo de la función de transferencia de la neurona. Se ha efectuado un análisis de las implementaciones de las funciones Sigmoidea y Exponencial en las que se ha utilizado una estructura que consiste en tabular los valores de la función combinada con un procedimiento de interpolación lineal.

Además se ha utilizado un esquema de división en el tiempo para el bloque multiplicador con el objetivo de implementar un solo bloque por neurona, para ejecutar todas las multiplicaciones asociadas al algoritmo para ahorrar recursos de la placa.

Los resultados se han evaluado en términos de error absoluto y relativo obtenidos para la aproximación y a través de un factor de calidad, demostrando una clara mejoría en relación a los trabajos publicadas con anterioridad.

High precision FPGA implementation of neural network activation functions

Francisco Ortega-Zamorano, José M. Jerez, Gustavo Juárez, Jorge O. Pérez, Leonardo Franco

Abstract— **The efficient implementation of artificial neural networks in FPGA boards requires tackling several issues that strongly affect the final result. One of these issues is the computation of the neuron's activation function. In this work, a detailed analysis of the FPGA implementations of the Sigmoid and Exponential functions is carried out, in a approach combining a lookup table with a linear interpolation procedure. Further, to optimize board resources utilization, a time division multiplexing of the multiplier attached to the neurons was used. The results are evaluated in terms of the absolute and relative errors obtained and also through measuring a quality factor and the resource utilization, showing a clear improvement in relationship to previously published works.**

I. Introduction

FPGAs [1] are reprogrammable silicon chips, using prebuilt logic blocks and programmable routing resources. They can be configured to implement custom hardware functionality, and in this sense, FPGAs are completely reconfigurable and can almost instantly change its behavior by recompiling a new circuitry configuration. In recent years, the advance in technology made possible to construct FPGAs with considerable large amounts of processing power and memory storage, permitting their application in several areas such as Telecommunications, Robotics, Pattern recognition tasks, Infrastructure monitoring, etc. [2]. As FPGAs are intrinsically parallel devices, they are quite suitable for Neural Network implementations, and so several studies have analyzed their application [3], [4], [5]. A broad classification of FPGA neural network applications can be done according to whether they include the learning process ("on-chip implementations") [6], [5] or if the training of the neural network model is performed externally in a Personal Computer (PC) where the FPGA acts as a hardware accelerator ("off-chip implementations") [7], [8]. Programming a FPGA is not a trivial task as they are predominantly codified using hardware description languages such as VHDL or Verilog, languages that are complex making the programming process very time consuming in most cases. An important aspect at the time of the implementation of an algorithm in a FPGA regards the data type representation. The nature of the FPGAs encourages the use of a fixed point representation because this type of representation is more efficient. A floating point representation might be used but this would require the utilization of specific cores [9], [10]. Regarding this issue, the work of Savich et al. (2007) [11] describes an interesting analysis of the implementation of floating point neural algorithms in fixed point arithmetic.

In the present work, we analyze the implementation in a FPGA board of Sigmoid and Exponential functions, two functions related to artificial neural network implementations. The Sigmoid function (σ) is one of the most used activation function in neural networks [12] , and as such its implementation in FPGA has been analyzed by several authors in the past [13], [14], [4]. The Exponential function is less commonly used in neural network models but it is considered in this work as it is related to the Upstart and C-Mantec constructive neural network algorithms that constitute a valid alternative to traditional backpropagation trained neural networks [16].

In the present work, we develop a method based on using a lookup table approach combined with a linear interpolation scheme. Similar approaches have not been used much in the past due to the memory requirements for the storage of table values, but given the actual specifications of FPGA boards it is a very interesting possibility [13], that may lead to fast and accurate results.

II. Methods

We analyze in this work the implementation on a FPGA board of two mathematical functions involved in the implementation of neural network models: the Sigmoid and Exponential functions. The board used for the current implementation is the Virtex-5 OpenSPARC Evaluation Platform (ML509). This device includes a Xilinx Virtex-5 XC5VLX110T FPGA that provides different connector devices that includes 2 USB ports, 2 PS/2 ports, RJ-45 (10/100/1000 Networking) and RS-232 connectors. The VIRTEX 5 board used has 69120 programmable LUTs (Look-Up Tables), 148 RAM/FIFO available blocks of memory and 64 DSP48 modules. Each DSP48 slice contains a 25 x 18 multiplier, an adder, and an accumulator, containing extensive cascade capabilities to efficiently implement high-speed DSP algorithms.

The Sigmoid function (σ) is one of the most commonly used activation function in neural networks [12] and is defined as:

$$\sigma(x) = \frac{1}{1 + exp(-x)}. \quad (1)$$

The Exponential function is a basic matematical function usually used to model a quantity that grows or decays at

F. Ortega-Zamorano, J.M Jerez and L. Franco are with the Department of Computer Science, Malaga University, Spain (email: {lfranco,jja,ortega}@lcc.uma.es), and J. Pérez and G. Juárez are with the Department of Computer Science, Tucumán National University, Argentina (email: {jperez,gjuarez }@herrera.unt.edu.ar)

This work was supported by grants from Junta de Andalucía P10-TIC-5770 and P08-TIC-04026, and from CICYT (Spain) through grant TIN2010-16556 (all including FEDER funds).

978-1-4799-4485-9/14/$31.00 ©2014 IEEE

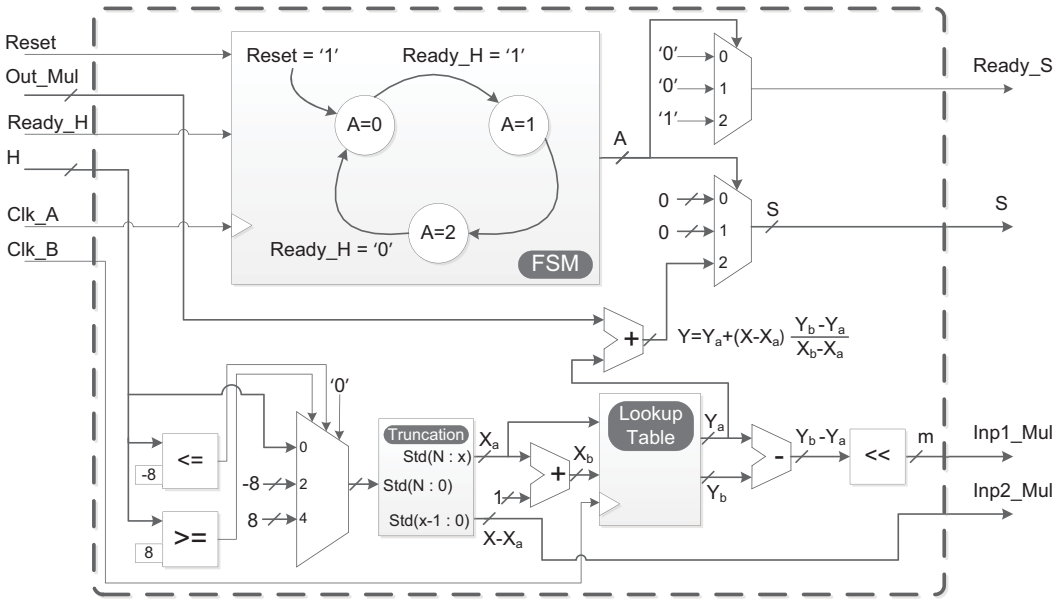

Fig. 1. Hardware representation of the interpolation block used for the computation of the functions.

a rate proportional to its current value, and it is used in several neural network algorithms like the Upstart [15]), and C-Mantec Constructive Neural Network algorithm [16].

The implementation of the Exponential and Sigmoid functions was carried out using a lookup table approach combined with a linear interpolation procedure. The lookup table contains equispaced values of the function in a range of input values of interest and the linear interpolation procedure uses two adjacent tabulated values (lower and larger) with respect to the input in order to compute the approximation, according to the following equation:

$$Y = Y_a + (X - X_a)\frac{(Y_b - Y_a)}{(X_b - X_a)}, \tag{2}$$

where X_a and X_b are the closest lower and upper values in relationship to the input one (X) that are stored in a lookup table, while Y_a and Y_b are the corresponding values of the function being computed.

The implementation has been carried out using fixed point arithmetic, as this type of representation is most suitable when using FPGAs [17].

Figure 1 shows hardware representation of the interpolation block. The inputs for the computation of the function are indicated on the upper left corner of the diagram: From top to bottom we have first a Reset signal that initialize the finite state machine (FSM), the multiplier output (Out_Mul), a signal (Ready_H) to indicate that the value of H is the correct one to be used, H the argument of the Sigmoid function (also known as the synaptic potential of the neuron), and two clock signals, Clk_A the system frequency and Clk_B the double of Clk_A. Within the figure, the upper part shows the finite state machine that controls the timing of the process, while in the lower part of the figure it is shown the procedure in charge of computing the approximation of the function, first by checking the closest values stored in the lookup table, and secondly by the computation of the linear interpolation (Eq. 2). On the right side of the diagram the outputs of the procedure are shown: Ready_S is a signal to indicate that the value of the function S is available, and near the bottom left part Inp1_Mul and Inp2_Mul are the inputs to a multiplier that will return Out_Mul. When the FSM is in the state $A = 1$ Inp1_Mul contains the value of $m = \frac{(Y_b - Y_a)}{(X_b - X_a)}$ while Inp2_Mul is equal to $X - X_A$. These two values are sent to a multiplier and the result is returned to the block as Out_Mul, so the value of S can be returned when the FSM is in state A=2 according to Eq. 2. As it can be appreciated in the FSM the whole procedure can be executed in two system clock cycles (from A=0 to A=2), because in this last state S is ready to be used in further computations.

The resources employed for the implementation of the interpolation process are shown in Table I as a function of the length of the representation used, formed by the integer part (N_a) and the decimal part (N_b). (Note that the values shown in the table do not take into account resources involved in the construction of the lookup table that are considered separately). Regarding the operation frequency of the block, in the worst case (N_a=N_b=16) the interpolation block generates a delay of 1.152 ns that implies a frequency of 868.056 MHz, higher than the maximum allowed frequency, so the system operating frequency will be determined by the rest of the hardware resources.

A. Multiplier

The strategy for the implementation of both functions was to use a time-division multiplexing strategy for the multiplier module permitting to compute sequentially the values and saving board resources. Figure 2 shows a schematic representation of a neuron in the FPGA in which the time-division

TABLE I

EMPLOYED RESOURCES IN THE IMPLEMENTATION OF THE INTERPOLATION PROCESS USED FOR THE COMPUTATION OF THE SIGMOID AND EXPONENTIAL FUNCTIONS.

N_a	N_b	LUTs	Registers
8	8	43	26
8	12	55	30
8	16	72	34
12	12	60	30
12	16	75	34
16	16	78	34

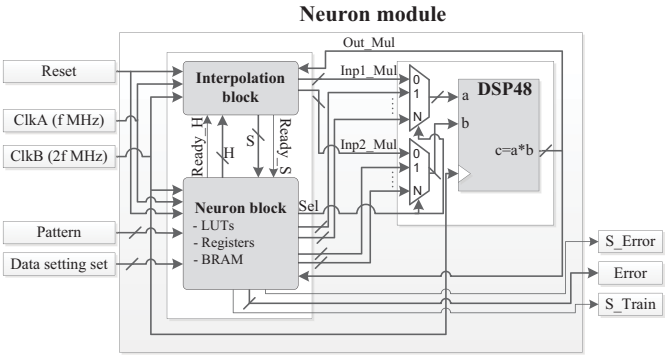

Fig. 2. Scheme of an implemented neuron that uses a time-division multiplexed strategy to execute several multiplications without adding complexity to the interpolation process.

TABLE II

FPGA MEMORY REQUIREMENTS IN LUTS FOR DIFFERENT VALUES OF N_L AND N_T.

N_T \ N_L	16	32	64	128	256	512	1024
8	8	12	16	30	32	210	388
12	12	18	24	46	52	434	796
16	20	26	32	62	84	490	924

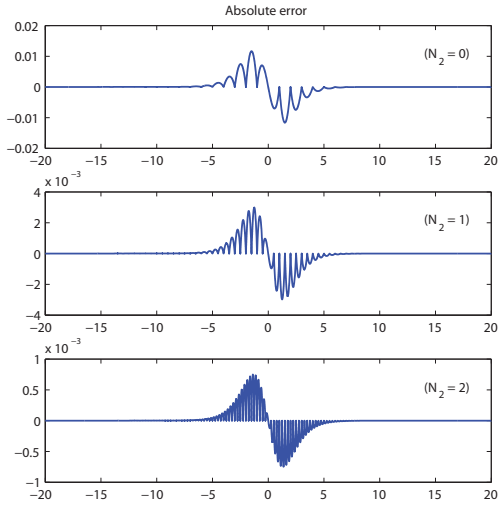

Fig. 3. Results obtained for the implementation of the Sigmoid functions for different values of N_2.

multiplexing of the DSP multiplier has been implemented. For an optimal utilization of the resources, the multiplier of each neuron has been performed with a synchronous DSP48 with an operation frequency two times faster than the one of the finite state machine, in order to perform the multiplication operation in one clock cycle.

B. Lookup Table

The computation of the sigmoid function utilizes a lookup table for storing tabulated values to be used in the interpolation procedure. The size of the table (N_L) is determined by the precision (number of bits) of the inputs. These values (X_a and X_b in Eq. 2) have been represented using a fixed point scheme with N_1 bits for the integer part and N_2 for the decimal part. In this way the total number of values in the table is defined by $N_L = 2^{N1+N2}$ for functions with only positive inputs or $N_L = 2^{1+N1+N2}$ for the case of functions with negative and positive inputs like the sigmoid function. To determine adequate values of N_1 and N_2, first, the value of N_2 is increased until the absolute error of the output function is lower than the desired value, to then apply the same process for N_1. To finish, the number of bits for representing the tabulated values (N_T) has to be determined. Table II shows the number of used LUTs for the different representation for the implementation of a lookup table, depending on $N_T = \{8, 12, 16\}$, and $N_L = \{16, 32, 64, 128, 256, 512, 1024\}$. A lookup table with any configuration of N_T and N_L can also be implemented by one block RAM instead of using LUTs, thus allowing for two different implementation possibilities that can be chosen according to the resource needs of the entire system.

III. RESULTS

A. Sigmoid function approximation

The approximation results obtained for the Sigmoid function are shown in Figs. 3, 4, and 5 where the absolute errors are shown as a function of the input value. Fig. 3 shows the result for three different values of $N_2 = \{0, 1, 2\}$, while Fig. 4 shows the result for three different values of $N_1 = \{1, 2, 3\}$. Fig. 5 shows the final approximation results as a function of $N_T = \{8, 12, 16\}$.

From an analysis of the results shown in Figs. 3, 4, and 5, our choice for the implementation of the Sigmoid function is $N_1 = 3$, $N_2 = 2$, and $N_T = 16$, as this choice permits to obtain absolute errors below 10^{-3} that guarantees a correct operation of most artificial neural networks algorithm. Table III shows maximum error (Max) and Root Mean Square Error (RMSE) for different values of N_1 and N_2. The first row of the table shows the results obtained for the chosen set of values ($N_1 = 3$, $N_2 = 2$) while for the rest of the rows larger values of N_1 and N_2 are used for comparison.

B. Comparison with previous approaches

We compare below the results obtained in the the current approach with two previous works where different aspects are taken into account in the analysis. Comparisons with other existing approaches can not be performed because the

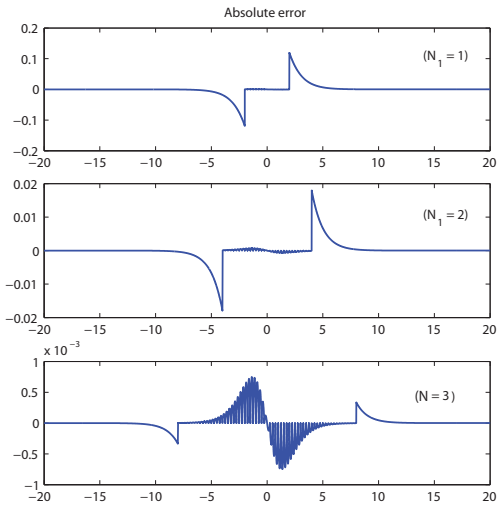

Fig. 4. Results obtained for the implementation of the Sigmoid functions for different values of N_1.

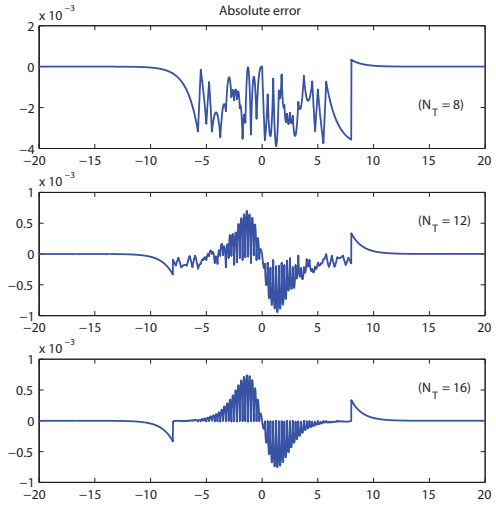

Fig. 5. Results obtained for the implementation of the Sigmoid functions for different values of N_T.

implementation details necessary for a reliable comparison were not given.

1) Comparison using a quality factor: In the work by Tommiska et al., 2003 [13] a quality factor Q has been introduced to analyze the accuracy and usability of a function implemented in a FPGA board:

$$Q = \frac{f_{max}}{LEs \cdot E_{ave} \cdot E_{max}}, \tag{3}$$

where f_{max} = clock rate, LEs = number of logic elements, E_{ave} = average error in per cent, and E_{max} = maximum error in per cent.

The quality factor (Q) for the Sigmoid approximation has been computed in two different cases, first by using the lookup table approximation alone (Q_1) and second when the interpolation scheme is used in combination with the lookup table (Q_2). The values of Q_1 and Q_2 are reported in Table IV together with previous published results [13]. For computing

TABLE III

MAXIMUM ERROR (MAX) AND ROOT MEAN SQUARE ERROR (RMSE) FOR DIFFERENT VALUES OF N_1 AND N_2 IN THE IMPLEMENTATION OF THE SIGMOID FUNCTION.

N_1	N_2	Max	RMSE
3	**2**	$7.548 \cdot 10^{-4}$	$2.447 \cdot 10^{-4}$
3	3	$3.353 \cdot 10^{-4}$	$9.477 \cdot 10^{-5}$
4	3	$1.982 \cdot 10^{-4}$	$4.428 \cdot 10^{-5}$
4	4	$6.091 \cdot 10^{-5}$	$1.394 \cdot 10^{-5}$
4	5	$2.669 \cdot 10^{-5}$	$8.783 \cdot 10^{-6}$
4	6	$1.755 \cdot 10^{-5}$	$8.362 \cdot 10^{-6}$

TABLE IV

QUALITY FACTOR FOR THE SIGMOID FUNCTION APPROXIMATIONS

Approximation	Q
Lookup table + interpolation	Q_2= 491.46
$sig_3 37p$ [13]	25.61
$sig_2 36p$ [13]	12.30
PLAN approximation [18]	1.743
Approximation of Alippi and StortiGajani[14]	1.085
Lookup table	Q_1 = 0.644
Approximation of Zhang [19].	0.227
A-law based approximation[20]	0.134

Q_1 the following values were used $f_{max} = 100$MHz, LEs= 32, $E_{ave} = 6.22$, $E_{max} = 0.78$, while for Q_2 the arguments were: $f_{max} = 100$MHz, LEs= $32 + 78 = 110$ (32 LEs for representing the table values, and 78 LEs for the interpolation procedure), $E_{ave} = 0.0755$, and $E_{max} = 0.0245$.

2) Comparsion based on resource utilization: A scheme of the Sigmoid function based on Taylor's theorem and Lagrange forms was proposed in [21], where a maximum allowable error(ϵ) of 0.01 was permitted in the hardware implementation. The necessary resources were one DSP and 11 LUTs, with the circuit performing the computation in 7 clock cycles at a maximum frequency of 373.5 MHz. The present work leads to a maximum allowable error of 0.001, using a DSP but in a multiplexing scheme involving no extra resources. Regarding the number of LUTS, the current implementation needs a larger resource utilization (up to 78 LUTs) because the circuit performs the computation in only 2 clock cycles with a maximum operation frequency of 868.056 MHz.

C. Exponential function approximation

The results for the exponential function approximation are shown in Figs. 6,7, and 8. Fig. 6 shows the result for three different values of $N_2 = \{0, 1, 2, 3\}$, while Fig. 7 shows the result for three different values of $N_1 = \{1, 2, 3\}$. Fig. 8 shows the final approximation results as a function of $N_T = \{8, 12, 16\}$.

From the observed results presented in Figs. 6,7, and 8 the choice made for the implementation of the Sigmoid function is $N_1 = 3$, $N_2 = 3$, and $N_T = 16$, as these values are the lowest ones permitting to obtain absolute errors below 2×10^{-3}.

Table V shows maximum error (Max) and Root Mean Square Error (RMSE) for different values of N_1 and N_2.

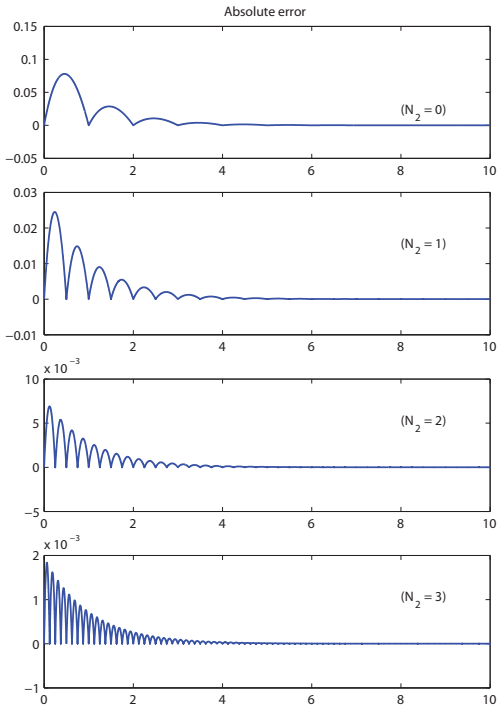

Fig. 6. Results obtained for the implementation of the exponential functions for different values of N_2.

TABLE V

MAXIMUM ERROR (MAX) AND ROOT MEAN SQUARE ERROR (RMSE) FOR DIFFERENT VALUES OF N_1 AND N_2 IN THE IMPLEMENTATION OF THE EXPONENTIAL FUNCTION.

N_1	N_2	Max	RMSE
3	**3**	$1.833 \cdot 10^{-3}$	$3.249 \cdot 10^{-4}$
3	4	$4.704 \cdot 10^{-4}$	$7.632 \cdot 10^{-5}$
4	4	$4.704 \cdot 10^{-4}$	$5.501 \cdot 10^{-5}$
4	5	$1.151 \cdot 10^{-4}$	$1.358 \cdot 10^{-5}$
4	6	$2.530 \cdot 10^{-5}$	$6.493 \cdot 10^{-6}$

The first row of the table shows the results obtained for the chosen set of values ($N_1 = 3$, $N_2 = 3$) while for the rest of the rows larger values of N_1 and N_2 are used for comparison.

In order to analyze the efficiency of the approximation done we have computed absolute and relative errors for the Exponential function in an arbitrarily selected interval between two tabulated values. Table VI displays the results for input values in the interval [2.000 2.125] showing in the columns the input values, the value of the exponential function with 6 significant digits, the value obtained with the approximation made and the absolute and relative errors. The extreme values of the interval considered corresponds to tabulated values used to compute the interpolation and so they appear in the table in boldface font.

IV. CONCLUSIONS

We have analyzed in this work the FPGA implementation of two basic functions related to the functioning of neural network algorithms by using a time division multiplexing strategy for carrying the multiplication operations in combination with a lookup table and interpolation scheme. A quality comparison done with respect to previously published results [13] for the Sigmoid function shows that the current implementation is very efficient, as we obtained a quality factor 40 times larger than the best previous published value, noting that this value is obtained without a significant increase of resource utilization. Regarding the lookup table plus interpolation scheme, it is also possible to conclude that the interpolation procedure helps very much to improve the approximation results, as the quality factor without the interpolation scheme reduces from 491 to 0.64. A further comparison to the results published in [21] shows that the current implementation is much faster, performing the computation of the Sigmoid function in only 2 clock cycles instead of the 7 used in the mentioned work.

Regarding the approximation of the Exponential function, the results can be considered of similar quality to those

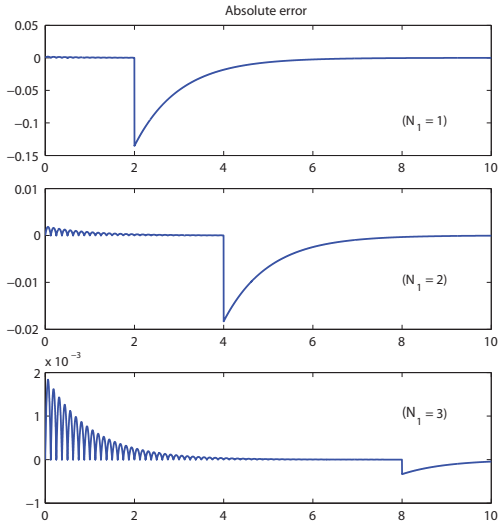

Fig. 7. Results obtained for the implementation of the exponential functions for different values of N_1.

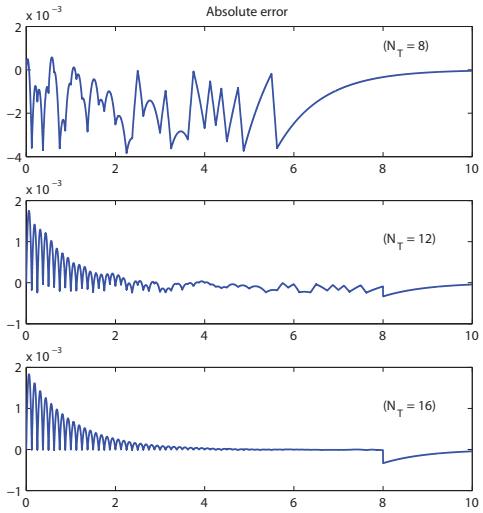

Fig. 8. Results obtained for the implementation of the exponential functions for different values of N_T.

TABLE VI

ABSOLUTE AND RELATIVE ERRORS FOR THE EXPONENTIAL FUNCTION FOR INPUT VALUES IN THE INTERVAL [2.000 2.125] FOR THE APPROXIMATION MADE USING A LOOKUP TABLE COMBINED WITH A LINEAR INTERPOLATION.

x	$exp(-x)$	Approximation	Absolute error	Relative error
2.000	**0.135335**	0.135330	$5 \cdot 10^{-6}$	$4 \cdot 10^{-5}$
2.025	0.131994	0.132150	$1 \cdot 10^{-4}$	$1 \cdot 10^{-3}$
2.050	0.128735	0.128970	$2 \cdot 10^{-4}$	$2 \cdot 10^{-3}$
2.075	0.125556	0.125790	$2 \cdot 10^{-4}$	$2 \cdot 10^{-3}$
2.100	0.122456	0.122610	$1 \cdot 10^{-4}$	$1 \cdot 10^{-3}$
2.125	**0.119433**	0.119430	$3 \cdot 10^{-6}$	$3 \cdot 10^{-5}$

obtained for the Sigmoidal function, even if the quality factor has not been compared as there were no previous published results. The results shown in table VI confirm the accuracy of the approximation, as very low error values were obtained. It is worth noting that the proposed techniques used for the functions approximation were obtained maximizing the utilization of FPGA resources, so that they can be replicated several times in the same board allowing for the construction of larger networks of neurons.

REFERENCES

[1] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007.

[2] E. Monmasson, L. Idkhajine, M. Cirstea, I. Bahri, A. Tisan, and M. W. Naouar, "Fpgas in industrial control applications." *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, pp. 224–243, 2011.

[3] J. Zhu and P. Sutton, "Fpga implementations of neural networks - a survey of a decade of progress," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL 2003*. Springer-Verlag, 2003, pp. 1062–1066.

[4] A. Gomperts, A. Ukil, and F. Zurfluh, "Development and implementation of parameterized fpga-based general purpose neural networks for online applications." *IEEE Transactions on Industrial Informatics*, vol. 7, no. 1, pp. 78–89, 2011.

[5] A. Omondi and J. Rajapakse, *FPGA Implementations of Neural Networks*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[6] K. Lakshmi and M. Subadra, "A survey on fpga based mlp realization for on-chip learning," in *International Journal of Scientific & Engineering Research*, vol. 4, 2013.

[7] 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*, pp. 880–888, 2007.

[8] S. Jung and S. S. Kim, "Hardware implementation of a real-time neural network controller with a dsp and an fpga for nonlinear systems," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 1, pp. 265–271, 2007.

[9] C. H. Ho, C. W. Yu, P. Leong, W. Luk, and S. J. E. Wilton, "Floating-point fpga: Architecture and modeling." *IEEE Trans. VLSI Syst.*, vol. 17, no. 12, pp. 1709–1718, 2009.

[10] Z. Jovanovic and V. Milutinovic, "Fpga accelerator for floating-point matrix multiplication," *IET Computers & Digital Techniques*, vol. 6, no. 4, pp. 249+, 2012.

[11] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing mlp-bp on fpgas: A study," *IEEE Transactions on Neural Networks*, vol. 18, pp. 240–252, 2007.

[12] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall, 1994.

[13] M. Tommiska, "Efficient digital implementation of the sigmoid function for reprogrammable logic," *IEE Proc. Comput. Digit. Techn.*, vol. 150, no. 6, pp. 403–411, 2003.

[14] C. Alippi and G. Storti Gajani, "Simple approximation of sigmoidal functions: realistic design of digital neural networks capable of learning," in *Proc. IEEE Int. Symp. on Circuits and Systems, Singapore*. IEEE Press, 1991, pp. 1505–1508.

[15] M. Frean, "The upstart algorithm: a method for constructing and training feedforward neural networks," *Neural Computation*, vol. 2, no. 2, pp. 198–209, Apr. 1990.

[16] J. Subirats, L. Franco, and J. Jerez, "C-mantec: A novel constructive neural network algorithm incorporating competition between neurons," *Neural Networks*, vol. 26, pp. 130–140, 2012.

[17] P. P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. John Wiley & Sons, 2008.

[18] H. Amin, K. Curtis, and B. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEE Proceedings - Circuits, Devices and Systems*, vol. 144, pp. 313–317(4), December 1997.

[19] M. Zhang, S. Vassiliadis, and J. G. Delgado-Frias, "Sigmoid generators for neural computing using piecewise approximations." *IEEE Trans. Computers*, vol. 45, no. 9, pp. 1045–1049, 1996.

[20] D. Myers and R. Hutchinson, "Efficient implementation of piecewise linear activation function for digital vlsi neural network," *Electron. Lett.*, no. 25, pp. 1662–1663, 1989.

[21] I. Campo, R. Finker, J. Echanobe, and K. Basterretxea, "Controlled accuracy approximation of sigmoid function for efficient fpga-based implementation of artificial neurons," *Electronics Letters*, vol. 49, no. 25, pp. 1598–1600, 2013.

Apéndice C

Comparativa de implementaciones hardware dos algoritmos de aprendizaje de redes neuronales

F. Ortega-Zamorano, José M. Jerez, Gustavo Juárez and Leonardo Franco:FPGA implementation comparison between C-Mantec and Back-Propagation neural network algorithms. **Lecture Notes in Computer Science** 9095, pp. 197-208, (2015). DOI: 10.1007/978-3-319-19222-2_17

Resumen:

Las implementaciones hardware de los modelos neurocomputacionales pueden orientarse a dos estrategias diferentes para conseguir diseños eficientes que reduzcan el consumo de recursos: modificar el algoritmo Backpropagation usado normalmente en este tipo de implementaciones o desarrollar nuevos algoritmos que se adapten mejor a los dispositivos.

Se ha realizado una comparación de las dos estrategias de diseño representadas por dos implementaciones diferentes. Una implementación ha sido una modificación del algoritmo Backpropagation, al definir una nueva primera capa para reducir los recursos utilizados; y la otra implementación ha sido el algoritmo C-Mantec que es un modelo de red neuronal constructivo que genera la arquitectura de red de forma automática.

Se analizan varios aspectos fundamentales de la aplicaciones sobre FPGAs, como son los recursos hardware utilizados, utilización de bloque específicos DSP, tiempos de Cómputo, número de LUTs, implementación de la función de transferencia, etc. Además se estudian las ventajas y desventajas de ambos métodos en el contexto de aplicaciones prácticas.

FPGA Implementation Comparison Between C-Mantec and Back-Propagation Neural Network Algorithms

Francisco Ortega-Zamorano[1], José M. Jerez[1], Gustavo Juárez[2], and Leonardo Franco[1(✉)]

[1] Department of Computer Science, ETSI Informática, Universidad de Málaga, Málaga, Spain
{fortega,jja,lfranco}@lcc.uma.es
[2] Universidad Nacional de Tucumán, San Miguel de Tucumán, Argentina
gjuarez@herrera.unt.edu.ar

Abstract. Recent advances in FPGA technology have permitted the implementation of neurocomputational models, making them an interesting alternative to standard PCs in order to speed up the computations involved taking advantage of the intrinsic FPGA parallelism. In this work, we analyse and compare the FPGA implementation of two neural network learning algorithms: the standard Back-Propagation algorithm and C-Mantec, a constructive neural network algorithm that generates compact one hidden layer architectures. One of the main differences between both algorithms is the fact that while Back-Propagation needs a predefined architecture, C-Mantec constructs its network while learning the input patterns. Several aspects of the FPGA implementation of both algorithms are analysed, focusing in features like logic and memory resources needed, transfer function implementation, computation time, etc. Advantages and disadvantages of both methods are discussed in the context of their application to benchmark problems.

Keywords: Constructive neural networks · FPGA · Hardware implementation

1 Introduction

Artificial Neural Networks (ANN) [1] are mathematical models inspired in the functioning of the brain that can be utilized in clustering and classification problems, and that have been successfully applied in several fields, including pattern recognition, stock market prediction, control tasks, medical diagnosis and prognosis, etc. The implementation of ANN in digital circuits (standard PCs, embedded systems, etc.) has been limited by the computational power needed, mainly given its intrinsic parallelism. In this sense, the capacity and performance of current FPGAs are a realistic alternative for the real time implementation of ANN. FPGAs [2] are reprogrammable silicon chips, using prebuilt logic blocks and

I. Rojas et al. (Eds.): IWANN 2015, Part II, LNCS 9095, pp. 197–208, 2015.
DOI: 10.1007/978-3-319-19222-2_17

programmable routing resources, that can be configured to implement custom hardware functionality, being able also to change almost instantly its behaviour by recompiling a new circuitry configuration. Recent advances in technology have permitted to construct FPGAs with considerable large amounts of processing power and memory storage, and as so they have been applied in several domains (Telecommunications, Robotics, Pattern recognition tasks, Infrastructure monitoring, etc.) [3–5]. In particular FPGAs seem quite suitable for Neural Network implementations as they can be programmed to operate in a parallel way [6–8].

Within the area of supervised pattern recognition, the efficient implementation of neurocomputational models into hardware can be done in principle following two different strategies: Modifying and adapting the traditional Back-Propagation algorithm or developing new algorithms that are better suited to the hardware constraints. In this work these two different possibilities are explored, first by doing a hardware optimization of the standard Back-Propagation algorithm [9], and secondly through the implementation of an alternative algorithm (C-Mantec) based on an incremental constructive architecture [10]. The overall idea of the work is to do a comparative analysis of the two approaches in order to identify the pros and cons for each case, and with this information take a decision depending on the specific application and the resources available. The Back-Propagation algorithm (BP) is the standard learning procedure for training multilayer neural networks architectures [11] [12] but one of the main problems associated to its implementation is the lack of a clear methodology for determining the network topology before training starts. On the other hand, C-Mantec [13] is a novel neural network constructive algorithm that utilizes competition between the neurons and a modified perceptron learning rule (thermal perceptron [14]) to build single hidden layer compact architectures with good prediction capabilities for the supervised classification problems. The organization of the present work is as follows: Section 2 includes the hardware implementation details and description about the Back-propagation and C-Mantec algorithms. Results from several comparison features are presented in Section 3, to finally present the discussion of the results and the conclusions obtained.

2 Algorithms Description and Implementation Details

We describe in this section the main functioning aspects of both algorithms, including also specific details of the FPGA implementation. An important issue and a big difference in relationship to standard PC implementations regards the number representation used in the FPGA. While for standard PCs floating point number representation is the standard choice, this type is not usually the most efficient for FPGAs and a fixed point number representation is preferred [15].

2.1 The Back-Propagation Algorithm

The Back-Propagation algorithm is a supervised learning method for training multilayer artificial neural networks based on the gradient descent strategy.

The network architecture for the implementation of the algorithm has to be decided previously and there is no standard methodology for this step, being the trial-and-error method one of the most used strategies. In the most general case the neural architecture comprises an input layer with a number of inputs determined by the problem at hand, several hidden layers, and one or many output neurons depending whether a binary or multi-output problem is analysed (for simplicity the first case is considered in this work).

The objective of the BP supervised learning algorithm is to minimize the difference between given outputs (targets) for a set of input data and the output of the network. This error depends on the values of the synaptic weights, and so these should be adjusted in order to minimize the error. The error function computed for the case of a single output neuron can be defined as:

$$E = \frac{1}{2}\sum_{i=1}^{M}(z_i - y_i)^2, \tag{1}$$

where the sum is over all training patterns, and z_i and y_i refers to target and network outputs for a given pattern i.

If we consider the neurons belonging to a hidden or output layer, the activation of these units, denoted by y_i, can be written as:

$$y_i = g\left(\sum_{j=1}^{L} w_{ij} \cdot s_j\right) = g(h)\ , \tag{2}$$

where w_{ij} are the synaptic weights between neuron i in the current layer and the neurons of the previous layer with activation s_j. In the previous equation, we have introduced h as the synaptic potential of a neuron. g is a sigmoid activation function given by:

$$g(x) = \frac{1}{1 - e^{-\beta x}} \tag{3}$$

By using the method of *gradient descent*, the BP algorithm attempts to minimize the error Eq. 1 in an iterative process by updating the synaptic weights upon the presentation of a given pattern. The synaptic weights between two last layers of neurons are updated as:

$$\Delta w_{ij}(k) = -\eta\frac{\partial E}{\partial w_{ij}(k)} = \eta[z_i(k) - y_i(k)]g_i'(h_i)s_j(k), \tag{4}$$

where η is the learning rate that has to be set in advance (a parameter of the algorithm), g' is the derivative of the sigmoid function and h is the synaptic potential previously defined, while the rest of the weights are modified according to similar equations by the introduction of a set of values called the "deltas" (δ), that propagate the error form the last layer into the inner ones.

The BP neural networks have been trained under a training / validation / testing strategy to avoid overfitting effects, caused mainly by excessive training

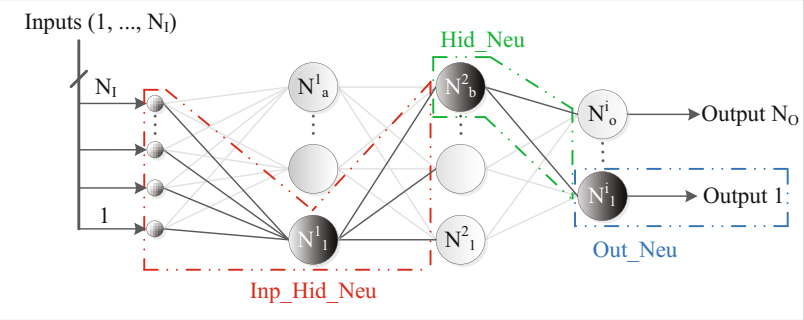

Fig. 1. Scheme of the FPGA architecture design for the implementation of the BP algorithm

iterations in the learning process, and thus a validation set is used to check the evolution of the mean square error between target and output values.

Regarding the FPGA implementation of the BP algorithm three main aspects have been carefully analyzed for increasing the efficiency of resource utilization: a) The introduction of a new input-hidden neurons block, b) A new scheme for computing the sigmoid transfer function, and c) A strategy of time division for using only a single multiplier block for each neuron.

Fig. 1 shows a scheme of the neural architecture, where three types of blocks are used for the implementation of the different parts of the neural architecture. The proposed implementation do not consider the input layer of neurons separately as this is included together with the first hidden layer neurons in a module named input-hidden neurons ("inp-hid"). The definition of this new type of module is possible because the input layer neurons do not process the information as they simple act as input to the network.

Another important FPGA design aspect regarding the hardware implementation of a neural network algorithm is the way of computing the activation function of the neurons, usually a sigmoid-type function. An scheme based on a lookup table approach plus linear interpolation scheme permits to obtain an efficient representation in terms of the resources needed together with low absolute and relative errors. In the previous work [16] a complete study about size of the table, employed resources and precision results has been presented. As conclusion the more efficient dimension of the table for a sigmoid is 2 bits for the decimal part, 3 bits for the integer part and one more bit foe the sign of the function. This parameters produce a table of $2^6 = 64$ inputs with 16 bits of word length as size of each input. The total resources necessary to implement this table are 32 bits or 1 block memory. Plus, Fig. 2 shows the approximation obtained for the sigmoid function (top graph) and the errors committed in its approximation (bottom graph).

Furthermore, a third important aspect considered during the FPGA implementation regards a time division scheme for performing the multiplications involved in the algorithm [16]. The multiplier blocks can be implemented both as a combination of logic cells or using specific DSP blocks. We have selected the first choice in this work for a fair comparison with the C-Mantec algorithm

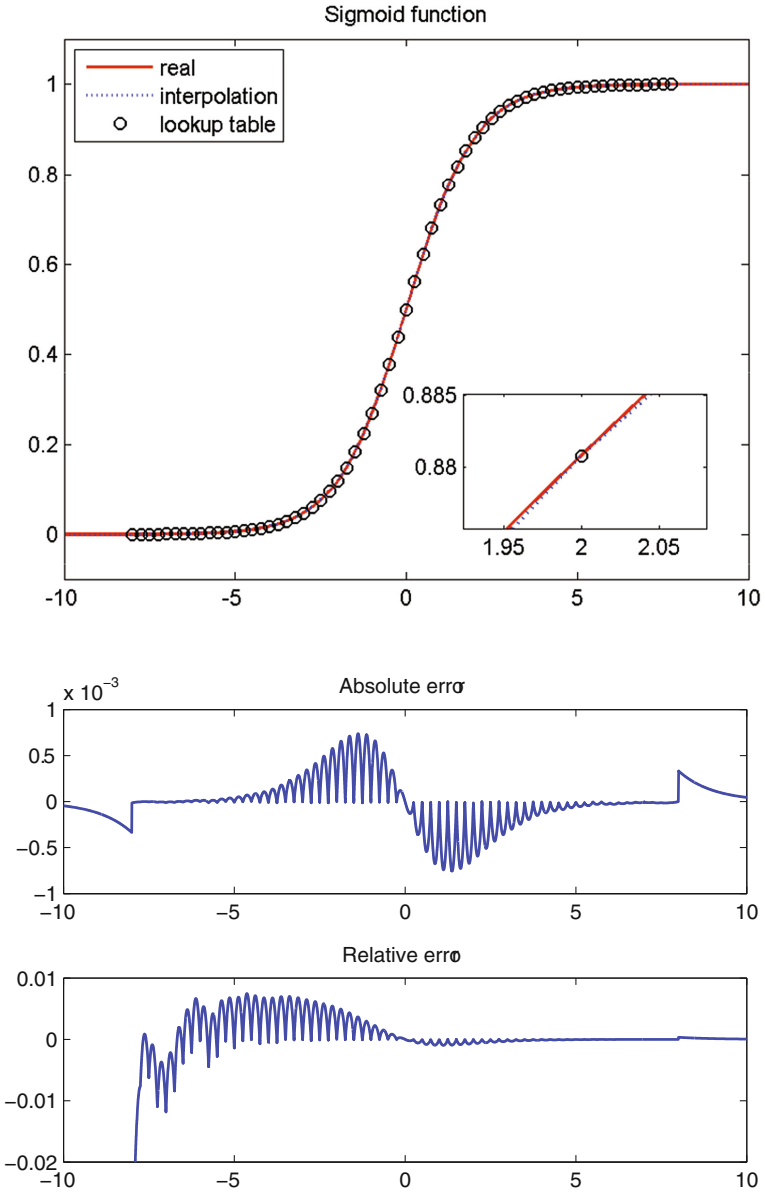

Fig. 2. FPGA sigmoid function approximation based on a lookup table plus linear interpolation scheme (top graph). Absolute (middle graph) and relative errors (bottom graph) committed in the approximation of the function.

implementation, as this was done without DSPs. Further the implementation using DSP is specific to each board and thus the present choice gives more generality to the results. The strategy consists in using a single multiplier for each neuron (built using logic blocks [17]) and then through using a time division multiplexing scheme compute all the multiplications related to the neuron.

2.2 The Constructive Neural Network Algorithm C-Mantec

C-Mantec as a constructive neural network algorithm generates the network topology in an on-line manner during the learning phase, avoiding the complex problem of selecting an adequate neural architecture [13]. The novelty of

C-Mantec in comparison to previous proposed constructive algorithms is that the neurons in the single hidden layer compete for learning the incoming data, and this process permits the creation of very compact neural architectures. The binary activation state (S_j) of each of the neurons in the hidden layer depends on N input signals, ψ_i, and on the actual value of the N synaptic weights (ω_{ji}) and bias (b_j) as follows:

$$S_j = \begin{cases} 1 & \text{if } h_j \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where h is the synaptic potential of the neuron defined as:

$$h_j = \sum_{i=1}^{N} \omega_{ji}\, \psi_i - b_j \tag{6}$$

The weight updating in the C-Mantec algorithm at the single neuron level is done using the thermal perceptron rule [14], in which the modification of the synaptic weights, $\Delta\omega_i$, is done on-line (after the presentation of a single input pattern) according to the following equation:

$$\Delta\omega_{ji} = (t - S_j)\, \psi_i\, T_{fac}, \tag{7}$$

where t is the target value (desired output of the whole network for the presented input), and ψ represents the value of input unit i connected to the hidden neuron S_j by synaptic weight ω_{ji}. The difference to the standard perceptron learning rule is that the thermal perceptron incorporates the T_{fac} factor. This factor, whose value is computed as shown in Eq. 8, depends on the value of the synaptic potential and on an artificially introduced temperature (T):

$$T_{fac} = \frac{T}{T_0} e^{-\frac{jh_j}{T}}, \tag{8}$$

The computation of the T_{fac} factor involves the FPGA implementation of the exponential function, task that was done using the same approach applied for the computation of the sigmoid function needed for the BP algorithm. Section 3 includes Table 2 that shows a comparison between the approximation results obtained for both functions (the sigmoid and exponential functions).

Following with the description of the C-Mantec algorithm, the value of the temperature T decreases as the learning process advances according to Eq. 9, similarly to a simulated annealing process.

$$T = T_0 \cdot (1 - \frac{I}{I_{max}}), \tag{9}$$

where I is a cycle counter that defines an iteration of the algorithm on one learning cycle, and I_{max} is the maximum number of iterations allowed. One learning cycle of the algorithm is the process that starts when a chosen pattern is presented to the network and finishes after checking that all neurons respond correctly to the input or when the synaptic weights of the neuron chosen to

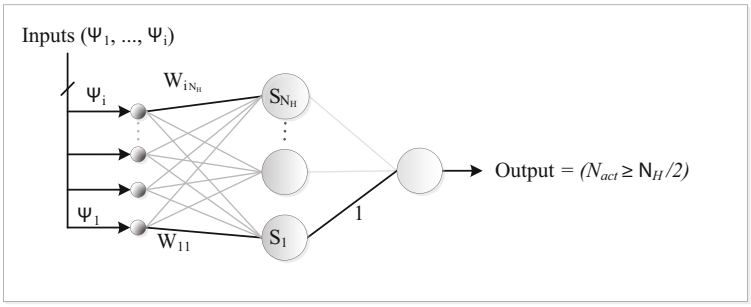

Fig. 3. Example of network architecture constructed by the C-Mantec algorithm

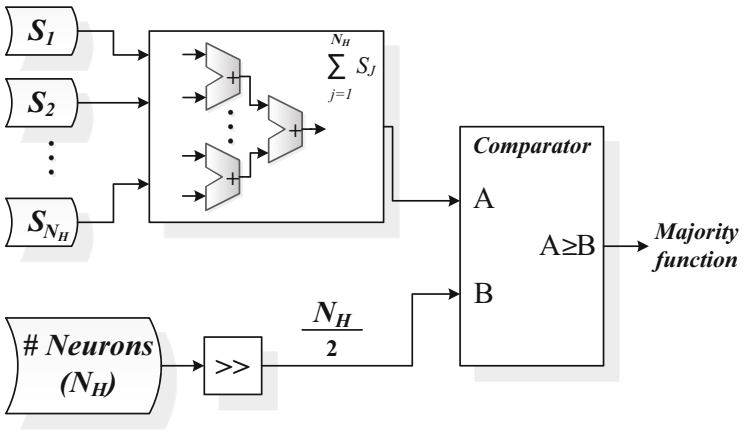

Fig. 4. Hardware implementation of the majority function that corresponds to the output neuron activation function of a network trained by the C-Mantec algorithm

learn the actual pattern (whether an existing or a new neuron) modifies its synaptic weights. The C-Mantec algorithm has three parameters to be set at the time of starting the learning procedure, and several experiments have shown the robustness of the algorithm that operates fairly well in a wide range of parameter values.

The output of a C-Mantec network consists in a single output that computes the majority function (see Eq.10) of the neuron activation of the hidden layer units, like in a voting process. The network output is active (1) if more than half of the N_H hidden neurons are active:

$$\text{Output} = \begin{cases} 1 & \text{if } \sum_j^{N_H} S_j \geq \frac{N_H}{2} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Fig. 3 shows a network architecture of the type built by the C-Mantec algorithm. The network contains a single hidden layer of threshold neurons (S_j) with output values $\{0, 1\}$.

The FPGA implementation of the majority function is shown in Fig. 4. On the left part of the figure the activation value of all N_H hidden layer neurons S_i are shown, followed by the computation of the sum of their activation. In the module indicated by "comparator" the obtained value is compared

with the value of $\frac{N_H}{2}$ and the whole network output is computed following Eq. 10. The whole process can be executed in less than one clock cycle of the FPGA because all operations involved are implemented with logic cells that introduce only minor delays.

3 Results

We present in this section results from the implementation of both algorithms (BP and C-Mantec) in a Xilinx Virtex-5 board. Table 1 shows some characteristics of the Virtex-5 XC5VLX110T FPGA, indicating its main logic resources. VHDL [17,18] (VHSIC Hardware Description Language) language was used for programming the FPGA, under the "Xilinx ISE Design Suite 12.4" environment using the "ISim M.81d" simulator.

Table 1. Main specifications of the Xilinx Virtex-5 XC5VLX110T FPGA board

Device	Slice Registers	Slice LUTs	Bonded IOBs	Block RAM
Virtex-5 XC5VLX110T	69,120	69,120	34	148

Table 2 shows the Maximum and Root Mean Square errors for different values of the integer N_a and decimal parts N_b obtained for the implementation of the exponential and sigmoidal functions used in the C-Mantec and Back-Propagation algorithms respectively through a lookup table plus linear interpolation scheme. As it can be appreciated from the the table both errors are quite low for both functions for almost all values of N_a and N_b being lower for the Sigmoidal function.

Table 2. Maximum error (Max) and Root Mean Square Error (RMSE) for different values of N_a and N_b in the implementation of the Exponential and Sigmoidal functions

N_a	N_b	Exponential		Sigmoidal	
		Max	RMSE	Max	RMSE
3	3	$1.833 \cdot 10^{-3}$	$3.249 \cdot 10^{-4}$	$3.353 \cdot 10^{-4}$	$9.477 \cdot 10^{-5}$
3	4	$4.704 \cdot 10^{-4}$	$7.632 \cdot 10^{-5}$	$1.982 \cdot 10^{-4}$	$4.428 \cdot 10^{-5}$
4	4	$4.704 \cdot 10^{-4}$	$5.501 \cdot 10^{-5}$	$6.091 \cdot 10^{-5}$	$1.394 \cdot 10^{-5}$
4	5	$1.151 \cdot 10^{-4}$	$1.358 \cdot 10^{-5}$	$2.669 \cdot 10^{-5}$	$8.783 \cdot 10^{-6}$
4	6	$2.530 \cdot 10^{-5}$	$6.493 \cdot 10^{-6}$	$1.755 \cdot 10^{-5}$	$8.362 \cdot 10^{-6}$

One of the most interesting results of this work is the comparison done for the maximum number of neurons that can be implemented with the board used, and also the resources associated with the implementation of a single neuron in

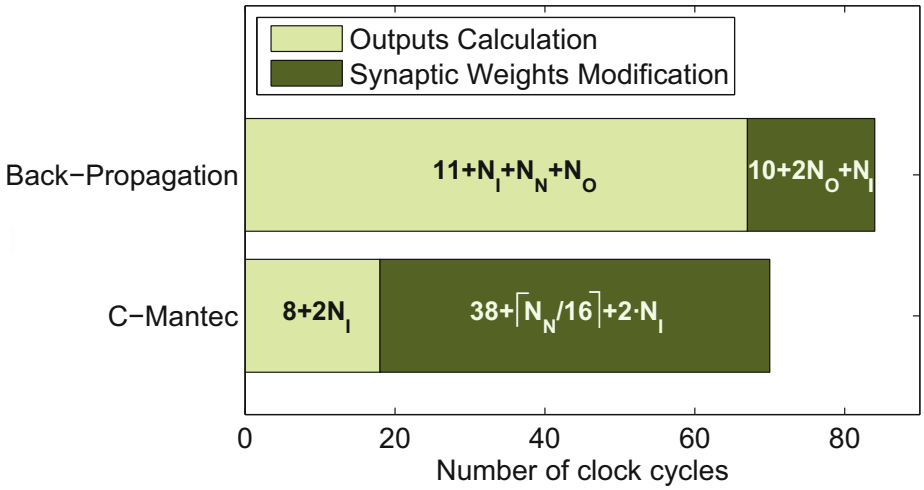

Fig. 5. Execution cycles needed to learn and compute the output for a single input pattern for the case of a 5-50-1 neural network architecture

relationship to both algorithms. Table 3 shows the values obtained for BP and C-Mantec for different values for the integer N_1 and decimal part N_2 of the fixed point representations tested.

Table 3. Number of LUTs and maximum number of neurons that can be implemented in a Virtex-5 board for the two algorithms and as a function of different fixed-point representations

N_1	N_2	LUTs/Neuron		# Neurons	
		BP	C-M	BP	C-M
8	8	787	689	82	94
8	12	967	757	67	85
8	16	1124	943	57	68
12	12	1057	826	61	78
12	16	1223	1033	53	62
16	16	1382	1299	47	50

We have also analysed the number of FPGA clock cycles involved in the computations related to training a network with one input pattern. The total number of cycles is divided in two parts related to the number cycles related to compute the output of the network for a given input and for the modification of the synaptic weights (cf. Eqs. 4 and 7). The results displayed in the table correspond to two neural networks with 50 neurons in the single hidden layer of the architecture.

Using a set of benchmark functions from the UCI repository, we have computed the generalization ability and computation time (ms) for both algorithms. Table 4 shows these results together with the number of neurons used in each

Table 4. Generalization ability (%) obtained and number of neurons used in the architectures for the implementation of seven classic benchmark problems

Function	C-Mantec			Back-Propagation		
	Gen.	# Neu.	time (ms)	Gen.	# Neu.	time (ms)
Diabetes	76.6	5	97	79.3	5	227
Cancer	96.9	2	52	95.7	5	210
Heart	82.6	3	71	78.2	5	104
Ionosphere	87.4	2	56	87.5	5	210
Heart-c	82.5	2	55	80.1	5	190
Card	85.2	3	72	83.1	5	195
Sonar	75.0	1	43	75.2	5	223
Average	83.7	3	63	82.7	5	194

case, noting that C-Mantec sets this number automatically while a constant size architecture comprising 5 neurons was used for Back-Propagation.

4 Conclusion

We have presented and analysed the implementation in a FPGA board of two neural network learning algorithms: Back-Propagation and C-Mantec. The algorithms operates from different principles as BP is a gradient based algorithm minimizing an error function for pre-determined architecture that has to be defined in advance, while C-Mantec is an error correcting method that constructs the network architecture automatically as it learns the input patterns. In terms of the FPGA implementation, both methods require the implementation of continues functions (the sigmoid and the exponential functions), process that is very simple for standard computers (PCs) but much more complex for hardware devices using a fixed point representation like FPGAs. An analysis of the resources needed to implement both functions efficiently indicate that similar error levels are obtained for both cases when using a lookup table plus linear interpolation scheme, with slightly lower error values for the case of the sigmoid function used in the BP algorithm. Nevertheless, it is worth noting that as C-Mantec is an error correcting algorithm the precision needed for the arithmetic representation is lower that for BP that might require high precision levels as it involves the accurate computation of the derivatives of the activation functions for its correct operation.

Another very important issue regarding the comparison of both algorithms is the amount of hardware resources needed for the implementation of single neurons in both algorithms, and in this aspect the advantage is on the C-Mantec side as a lower number of LUTs is required, permitting for a given board the construction of larger neural network architectures, that in our case resulted in approximately a 18.7% increase in the maximum number of neurons that can be included (cf. Table 3). Further, we have also estimated the average computation time needed for training both algorithms using a set of benchmark functions,

finding that C-Mantec operates faster than BP, needing in average a third of the computational time (cf. Table 4).

As an overall conclusion the present work shows a comparison regarding the possibilities of the application of neurocomputational algorithms using FPGA boards. The comparison of both algorithms is a little bit in favour of C-Mantec as first it does not need the a priori specification of the neural architecture to be used, and second as it is less demanding in terms of hardware resource utilization.

Acknowledgments. The authors acknowledge support from Junta de Andalucía through grants P10-TIC-5770 and P08-TIC-04026, and from CICYT (Spain) through grant TIN2010-16556 (all including FEDER funds).

References

1. Haykin, S.: Neural networks: a comprehensive foundation. Prentice Hall (1994)
2. Kilts, S.: Advanced FPGA Design: Architecture, Implementation, and Optimization. Wiley-IEEE Press (2007)
3. Monmasson, E., Idkhajine, L., Cirstea, M., Bahri, I., Tisan, A., Naouar, M.W.: Fpgas in industrial control applications. IEEE Transactions on Industrial Informatics **7**(2), 224–243 (2011)
4. Bacon, D., Rabbah, R., Shukla, S.: Fpga programming for the masses. Queue **11**, 40–52 (2013)
5. Conmy, P., Bate, I.: Component-based safety analysis of fpgas. IEEE Transactions on Industrial Informatics **6**(2), 195–205 (2010)
6. Zhu, J., Sutton, P.: Fpga implementations of neural networks - a survey of a decade of progress. In: Cheung, P.Y.K., Constantinides, G.A. (eds.) FPL 2003. LNCS, vol. 2778, pp. 1062–1066. Springer, Heidelberg (2003)
7. Gomperts, A., Ukil, A., Zurfluh, F.: Development and implementation of parameterized fpga-based general purpose neural networks for online applications. IEEE Trans. Industrial Informatics **7**(1), 78–89 (2011)
8. Le, Q., Jeon, J.: Neural-network-based low-speed-damping controller for stepper motor with an fpga. IEEE Transactions on Industrial Applications **57**, 3167–3180 (2010)
9. Ortega-Zamorano, F., Jerez, J., Urda, D., Luque-Baena, R., Franco., L.: Efficient implementation of the backpropagation algorithm in fpgas and microcontrollers. IEEE Transactions on Neural Networks and Learning Systems (2015) (in press)
10. Ortega-Zamorano, F., Jerez, J., Franco, L.: Fpga implementation of the c-mantec neural network constructive algorithm. IEEE Transactions on Industrial Informatics **10**(2), 1154–1161 (2014)
11. Werbos, P.J.: Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University (1974)
12. Rumelhart, D., Hinton, G., Williams, R.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986)
13. Subirats, J.L., Franco, L., Jerez, J.M.: C-mantec: A novel constructive neural network algorithm incorporating competition between neurons. Neural Netw. **26**, 130–140 (2012)
14. Frean, M.: The upstart algorithm: a method for constructing and training feedforward neural networks. Neural Computation **2**(2), 198–209 (1990)

15. Savich, A., Moussa, M., Areibi, S.: The impact of arithmetic representation on implementing mlp-bp on fpgas: A study. IEEE Transactions on Neural Networks **18**(1), 240–252 (2007)
16. Ortega-Zamorano, F., Jerez, J., Juarez, G., Perez, J., Franco, L.: High precision fpga implementation of neural network activation functions. In: 2014 IEEE Symposium on Intelligent Embedded Systems (IES), pp. 55–60, December 2014
17. Chu, P.P.: FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version. John Wiley & Sons (2008)
18. Ashenden, P.: The Designer's Guide to VHDL (Systems on Silicon), vol. 3, 3rd edn. Morgan Kaufmann Publishers Inc., San Francisco (2008)

Bibliografía

FPGA (Field-Programmable Gate Array) Market Analysis By Application (Automotive, Consumer Electronics, Data Processing, Industrial, Military And Aerospace, Telecom) And Segment Forecasts To 2020, 2014.

Aiello, F., Bellifemine, F. L., Fortino, G., Galzarano, S. y Gravina, R. An agent-based signal processing in-node environment for real-time human activity monitoring based on wireless body sensor networks. *Eng. Appl. Artif. Intell.*, vol. 24(7), páginas 1147–1161, 2011.

Aleksendrić, D., Jakovljević, I. y Irović, V. Intelligent control of braking process. *Expert Syst. Appl.*, vol. 39(14), 2012.

Baena, R. M. L., Urda, D., Subirats, J. L., Franco, L. y Jerez, J. M. Analysis of cancer microarray data using constructive neural networks and genetic algorithms. En *IWBBIO* (editado por I. Rojas y F. M. O. Guzman), páginas 55–63. Copicentro Editorial, 2013.

Bahoura, M. Fpga implementation of high-speed neural network for power amplifier behavioral modeling. *Analog Integrated Circuits and Signal Processing*, vol. 79(3), páginas 507–527, 2014.

Bahrammirzaee, A. A comparative survey of artificial intelligence applications in finance: artificial neural networks, expert system and hybrid intelligent systems. *Neural Computing and Applications*, vol. 19(8), páginas 1165–1195, 2010.

Chaoui, H., Sicard, P. y Gueaieb, W. Ann-based adaptive control of robotic manipulators with friction and joint elasticity. *Industrial Electronics, IEEE Transactions on*, vol. 56(8), páginas 3174–3187, 2009.

Dinu, A., Cirstea, M. y S.E.Cirstea. Direct neural-network hardware-implementation algorithm. *IEEE Transactions on Industrial Electronics*, vol. 57(5), páginas 1845–1848, 2010.

E. Cañete, E., Chen, J., Luque, R. y Rubio, B. Neuralsens: A neural network based framework to allow dynamic adaptation in wireless sensor and actor networks. *J. Network and Computer Applications*, vol. 35(1), páginas 382–393, 2012.

Farooq, U., Amar, M., ul Haq, E., Asad, M. U. y Atiq, H. M. Microcontroller based neural network controlled low cost autonomous vehicle. En *Proceedings of the 2010 Second International Conference on Machine Learning and Computing*, ICMLC '10, páginas 96–100. IEEE Computer Society, Washington, DC, USA, 2010. ISBN 978-0-7695-3977-5.

FRANCO, L., ELIZONDO, D. A. y JEREZ, J. *Constructive neural network*, vol. 258. Springer, 2010.

FRANK, R. *Understanding Smart Sensors, Second Edition*. Artech House, Inc., Norwood, MA, USA, 2nd edición, 2000. ISBN 0890063117.

GOMPERTS, A., UKIL, A. y ZURFLUH, F. Implementation of neural network on parameterized fpga. En *AAAI Spring Symposium: Embedded Reasoning: Stanford University,*, páginas 45–51. AAAI Spring Symposium 2010 - Embedded Reasoning: Stanford University, CA, USA, 2010.

GOMPERTS, A., UKIL, A. y ZURFLUH, F. Development and implementation of parameterized fpga-based general purpose neural networks for online applications. *IEEE Transactions on Industrial Informatics*, vol. 7(1), páginas 78–89, 2011.

HAN, C.-C., KUMAR, R., SHEA, R. y SRIVASTAVA, M. Sensor network software update management: a survey. *Int. J. Netw. Manag.*, vol. 15(4), páginas 283–294, 2005.

HAYKIN, S. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edición, 1998.

HIMAVATHI, S., ANITHA, D. y MUTHURAMALINGAM, A. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, páginas 880–888, 2007.

HO, C. H., YU, C. W., LEONG, P., LUK, W. y WILTON, S. J. E. Floating-point fpga: Architecture and modeling. *IEEE Trans. VLSI Syst.*, vol. 17(12), páginas 1709–1718, 2009.

HOPFIELD, J. J. Neural networks and physical systems with emergent collective computational abilities. *National Academy of Sciences of the USA*, vol. 79(8), páginas 2554–2558, 1982.

BIN HUANG, G., MEMBER, S., YU ZHU, Q. y KHEONG SIEW, C. Real-time learning capability of neural networks. *IEEE Trans. Neural Networks*, páginas 863–878, 2006.

JOVANOVIC, Z. y MILUTINOVIC, V. Fpga accelerator for floating-point matrix multiplication. *IET Computers & Digital Techniques*, vol. 6(4), páginas 249+, 2012.

JUNG, S. y KIM, S. S. Hardware implementation of a real-time neural network controller with a dsp and an fpga for nonlinear systems. *IEEE Transactions on Industrial Electronics*, vol. 54(1), páginas 265–271, 2007.

KALOGIROU, S. A. Artificial neural networks in renewable energy systems applications: a review. *Renewable and Sustainable Energy Reviews*, vol. 5(4), páginas 373–401, 2001.

KHARIF, O. Trillions of smart sensors will change life. bloomberg news. http://www.bloomberg.com/news/2013-08-05/trillions-of-smart-sensors-will-change-life-as-apps-have.html, 2013.

KUO, S. M., LEE, B. H. y TIAN, W. *Real-Time Digital Signal Processing: Implementations and Applications*. Wiley, 2006.

LAUKKARINEN, T., SUHONEN, J. y HANNIKAINEN, M. A survey of wireless sensor network abstraction for application development. *International Journal of Distributed Sensor Networks*, vol. 2012, páginas 1–12, 2012.

LOTRIČ, U. y BULIĆ, P. Applicability of approximate multipliers in hardware neural networks. *Neurocomputing*, vol. 96, páginas 57–65, 2012.

MAHMOUD, S., LOTFI, A. y LANGENSIEPEN, C. Behavioural pattern identification and prediction in intelligent environments. *Appl. Soft Comput.*, vol. 13(4), páginas 1813–1822, 2013.

MCCULLOCH, W. S. y PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, vol. 5(4), páginas 115–133, 1943.

MELODIA, T., POMPILI, D., GUNGOR, V. y AKYILDIZ, I. Communication and coordination in wireless sensor and actor networks. *Mobile Computing, IEEE Transactions on*, vol. 6(10), páginas 1116–1129, 2007.

MEMMERT, D. y PERL, J. Game creativity analysis using neural networks. *Journal of Sports Sciences*, vol. 27(2), páginas 139–149, 2009.

MINSKY, M. y PAPERT, S. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

OMONDI, A. y RAJAPAKSE, J. *FPGA Implementations of Neural Networks*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387284850.

ORLOWSKA-KOWALSKA, T. y KAMINSKI, M. Fpga implementation of the multilayer neural network for the speed estimation of the two-mass drive system. *IEEE Transactions on Industrial Informatics*, vol. 7(3), páginas 436–445, 2011.

PALOMO, E. J., DOMINGUEZ, E., LUQUE-BAENA, R. M. y MUÑOZ, J. Image compression and video segmentation using hierarchical self-organization. *Neural Processing Letters*, vol. 37(1), páginas 69–87, 2013.

PAN, S.-T. y LAN, M.-L. An efficient hybrid learning algorithm for neural network-based speech recognition systems on fpga chip. *Neural Computing & Applications*, vol. 24(7–8), páginas 1879–1885, 2014.

RASSAM, M., ZAINAL, A. y MAAROF, M. An adaptive and efficient dimension reduction model for multivariate wireless sensor networks applications. *Appl. Soft Comput.*, vol. 13(4), páginas 1978–1996, 2013.

RAULT, T., BOUABDALLAH, A. y CHALLAL, Y. Energy efficiency in wireless sensor networks: A top-down survey. *Computer Networks*, vol. 67(0), páginas 104 – 122, 2014.

ROSENBLATT, F. *Principles of neurodynamics; perceptrons and the theory of brain mechanisms*. Spartan Books Washington, 1962.

RUMELHART, D., HINTONT, G. y WILLIAMS, R. Learning representations by back-propagating errors. *Nature*, vol. 323(6088), páginas 533–536, 1986.

SALARIAN, H., CHIN, K.-W. y NAGHDY, F. Coordination in wireless sensor/actuator networks: A survey. *Journal of Parallel and Distributed Computing*, vol. 72(7), páginas 856 – 867, 2012.

SAVICH, A. W., MOUSSA, M. y AREIBI, S. The impact of arithmetic representation on implementing mlp-bp on fpgas: A study. *IEEE Transactions on Neural Networks*, vol. 18, páginas 240–252, 2007.

SAYED-MOUCHAWEH, M. y LUGHOFER, E. *Learning in non-stationary environments: Methods and Applications*. Springer, New York, 2012.

SHAIKH, R., THAKARE, V. y DHARASKAR, R. Efficient code dissemination reprogramming protocol for wsn. *International Journal of Computer and Network Security*, vol. 2(2), páginas 116–122, 2010.

SHAWASH, J. y SELVIAH, D. Real-time nonlinear parameter estimation using the levenberg-marquardt algorithm on field programmable gate arrays. *IEEE Transactions on Industrial Electronics*, vol. 60(1), páginas 170–176, 2013.

SUBIRATS, J., FRANCO, L. y JEREZ, J. C-mantec: A novel constructive neural network algorithm incorporating competition between neurons. *Neural Networks*, vol. 26, páginas 130–140, 2012.

URDA, D., CANETE, E., SUBIRATS, J. L., FRANCO, L., LLOPIS, L. y JEREZ, J. M. Energy-efficient reprogramming in wsn using constructive neural networks. *International Journal of Innovative, Computing, Information and Control*, vol. 8, páginas 7561–7578, 2012.

WANG, B. Coverage problems in sensor networks: A survey. *ACM Comput. Surv.*, vol. 43(4), páginas 32:1–32:53, 2011.

WANG, Q., ZHU, Y. y CHENG, L. Reprogramming wireless sensor networks: challenges and approaches. *Network, IEEE*, vol. 20(3), páginas 48–55, 2006.

ZHAO, W., CHELLAPPA, R., PHILLIPS, P. J. y ROSENFELD, A. Face recognition: A literature survey. *ACM Comput. Surv.*, vol. 35(4), páginas 399–458, 2003.