DEPÓSITO LEGAL ZU2020000153
ISSN 0041-8811
E-ISSN 2665-0428

Revista de la Universidad del Zulia

Fundada en 1947
por el Dr. Jesús Enrique Lossada

Ciencias del Agro Ingeniería y Tecnología

Año 12 N° 32
Enero - Abril 2021
Tercera Época
Maracaibo-Venezuela

DOI: http://dx.doi.org/10.46925//rdluz.32.16

Red neuronal convolucional usando VHDL para entrenar un clasificador de objetos en una imagen

Cesar Arturo Niño Carmonal *
Manuel-Jesús Sánchez-Chero **
Emanuel Ortiz Ortiz ***
Juan Carlos Sernaque Julca ****
Cecilia Lizeth Risco Ipanaqué *****

RESUMEN

El objetivo del presente trabajo fue implementar una Red neuronal convolucional en hardware usando VHDL. En cuanto a su diseño fue experimental, la investigación inicia con el diseño de una red neuronal convolucional en Software usando Python, donde se utilizó Tensorflow y Keras. Este diseño necesitó un entrenamiento de 6 épocas, para superar el 90% de exactitud al momento de clasificar las imágenes del dataset MNITS. De este diseño se obtienen los parámetros e hiperparámetros, necesarios para el diseño en hardware. Para la implementación del algoritmo en hardware, fue necesario conocer el funcionamiento matemático de las operaciones de convolución, maxpooling y de las redes neuronales, ya que en el software estas operaciones están resumidas en una línea de código. Cada una de estas operaciones fue implementada en bloques diferentes, siguiendo el enfoque modular. La respuesta que se obtiene en el hardware, se muestra en una pantalla usando la comunicación interna de la placa entre el ARM y la FPGA. Esta respuesta obtenida en hardware es similar a la que se obtiene en software y el tiempo en el software es mucho mayor al del hardware. Para esta investigación se utilizó la plataforma SoC basada en FPGA, De-10 Nano.

PALABRAS CLAVE: Red neuronal Convolucional, FPGA, Python, SoC, VHDL

* Docente Asociado. Universidad Nacional de Piura. Perú. ORCID: https://orcid.org/0000-0002-0981-0822

** Docente Investigador. Universidad Nacional de Frontera. ORCID: Perú. https://orcid.org/0000-0003-1646-3037. E-mail: manuelsanchezchero@gmail.com

*** Bachiller. Universidad Nacional de Piura. Perú. ORCID: https://orcid.org/0000-0002-4222-7372

**** Bachiller. Universidad Nacional de Piura. Perú. ORCID: https://orcid.org/0000-0002-3157-8935

***** Jefa (E) de la Unidad de Tecnología de Información y Comunicación. Universidad Nacional de Frontera. Perú. ORCID: https://orcid.org/0000-0002-7936-1495

Recibido: 18/09/2020 Aceptado: 24/11/2020

DOI: http://dx.doi.org/10.46925//rdluz.32.16

Convolutional neural network using VHDL to train an object classifier on an image

ABSTRACT

The objective of the present work was to implement a convolutional neural network in hardware using VHDL. Regarding its design, it was experimental, the research begins with the design of a convolutional neural network in Software using Python, where Tensorflow and Keras were used. This design required a 6-epoch training to exceed 90% accuracy when classifying the images from the MNITS dataset. From this design, the parameters and hyperparameters, necessary for hardware design, are obtained. For the implementation of the algorithm in hardware, it was necessary to know the mathematical operation of the convolution, maxpooling and neural network operations, since in the software these operations are summarized in a line of code. Each of these operations was implemented in different blocks, following the modular approach. The response obtained in the hardware is displayed on a screen using the internal communication of the board between the ARM and the FPGA. This response obtained in hardware is similar to that obtained in software and the time in software is much longer than in hardware. The FPGA-based SoC platform, De-10 Nano, was used for this research.

KEYWORDS: Convolutional Neural Network, FPGA, Python, SoC, VHDL

Introducción

A medida que las redes neuronales convolucionales continúan aplicándose para resolver problemas aún más complejos, sus demandas computacionales y de almacenamiento están aumentando enormemente. Convencionalmente, las redes neuronales convolucionales se han ejecutado en CPU y GPU, sin embargo, su bajo rendimiento y/o eficiencia energética presentan un cuello de botella en su uso, como menciona (Mittal. 2018). De acuerdo con (Ovtcharov et al. 2015), las FPGA son plataformas prometedoras para la aceleración hardware de redes neuronales convucionales, debido a sus características. En general, los FPGA proporcionan una mayor eficiencia energética que las GPU y las CPU y un mayor rendimiento que las CPU como menciona (Mittal, Vetter, 2015).

La investigación consiste en 2 partes fundamentales, como es el diseño de una red neuronal convolucional (CNN) en software, la cual se implementará en hardware usando una plataforma SoC De-10 Nano. La primera parte del proyecto se enfocó en el aprendizaje de la red

DOI: http://dx.doi.org/10.46925//rdluz.32.16

neuronal convolucional, utilizando el lenguaje de programación Python, donde se obtuvieron los parámetros e hiperparámetros. A esta etapa también se le conoce como backpropagation en Inteligencia Artificial. En la segunda parte del proyecto se utilizaron los datos obtenidos del aprendizaje para realizar la implementación de la red neuronal convolucional en la plataforma SoC De-10 Nano.

Se diseñó la arquitectura de la red neuronal convolucional usando un lenguaje de descripción de hardware como VHDL para luego implementarla en la FPGA. Esto se realizó para aprovechar el paralelismo que ofrecen la FPGA y la red neuronal convolucional, para lograr obtener una respuesta más eficiente y rápida. A esta etapa se le conoce como forward-propagation en Inteligencia Artificial que es donde se aplica lo aprendido en la etapa anterior. El resultado obtenido se envió al ARM. Se usaron los protocolos de comunicación, para poder enviar datos desde la FPGA hacia el ARM de la plataforma SoC De-10 Nano, donde se ejecutó el sistema operativo en el cual se visualizó la respuesta de la red neuronal convolucional en hardware para clasificación de un objeto en una imagen.

1. Material y métodos

La presente investigación, según de Cross (2001), es aplicada; y de acuerdo con Hernández et al, (2014), la investigación desarrollada es de tipo cuantitativa, debido a que se realizó un análisis de la respuesta de la Red neuronal convolucional en Python y su respuesta cuando fue implementada en una plataforma SoC basada en FPGA. En cuanto a su diseño fue experimental (Hernández et al, (2014).

En esta investigación se utilizó el dataset de MNIST. La cual se divide en 2 grupos: uno enfocado en los dígitos escritos a mano del 0 a 9 y otro con diferentes tipos de ropa (10 clases). Estos dataset consisten en un conjunto de entrenamiento de 60,000 ejemplos y un conjunto de prueba de 10,000 ejemplos. Cada ejemplo es una imagen en escala de grises de 28x28, asociada con una etiqueta de 10 clases.

El procedimiento para hacer la implementación de la red neuronal convolucional en una plataforma SoC basada en FPGA, es el siguiente:

DOI: http://dx.doi.org/10.46925//rdluz.32.16

1. Diseño de una red neuronal convolucional en Python.

1.1. Importación de librerías.

1.2. Diseño de la red neuronal convolucional con Tensorflow y Keras.

1.3. Descargas de los parámetros (Pesos y Bias) de la red neuronal convolucional entrenada en formato .MIF.

2. Diseño de la arquitectura de una red neuronal convolucional en VHDL.

2.1. Diseño del módulo de convolución 1.

2.2. Diseño del módulo de maxpooling.

2.3. Diseño del módulo de convolución 2.

2.4. Diseño del módulo de Fully Connected.

3. Desarrollo del módulo de red neuronal convolucional en una FPGA.

4. Diseño del hardware para la plataforma SoC.

Se utilizó la técnica de recolección de datos donde se obtuvo el porcentaje de acierto y el tiempo de respuesta en la implementación de la red neuronal convolucional, tanto en Software como en Hardware. La instrumentación de recolección de datos fue mediante fichas o guías de observación.

2. Resultados

2.1. Diseño de una red neuronal convolucional en Python

Según (Goodfellow at. 2016) las redes neuronales convolucionales son simplemente redes neuronales que utilizan la convolución en lugar de la multiplicación matricial general en al menos una de sus capas. La red neuronal convolucional propuesta se basó en la arquitectura LeNet-5 propuesta por (Lecun et al. 1998), porque las imágenes de entradas que se analizaron están en escala de grises y tienen una dimensión de 28x28. En la figura 1 se muestra el modelo de la arquitectura LeNet-5, la cual está compuesta por 2 capas de convolución que utilizan un filtro de 3x3, porque requieren una mejor carga computacional como menciona (Géron, 2019). En la convolución 1 se utilizaron 5 filtros y en la convolución 2 se utilizó 16 filtros. Las etapas de maxpooling usa un stride de 2x2, por lo que, por cada 4 píxeles, se escogerá el más grande. La

DOI: http://dx.doi.org/10.46925//rdluz.32.16

etapa de Fully Connected está compuesta por 3 sub etapas de redes neuronales, siendo la primera de 120 neuronas, la segunda de 84 neuronas y la última de 10 neuronas. La función de activación utilizada fue Relu en las etapas de convolución y en las 2 primeras redes neuronales. En la última red neuronal se utilizó la función de activación Softmax.

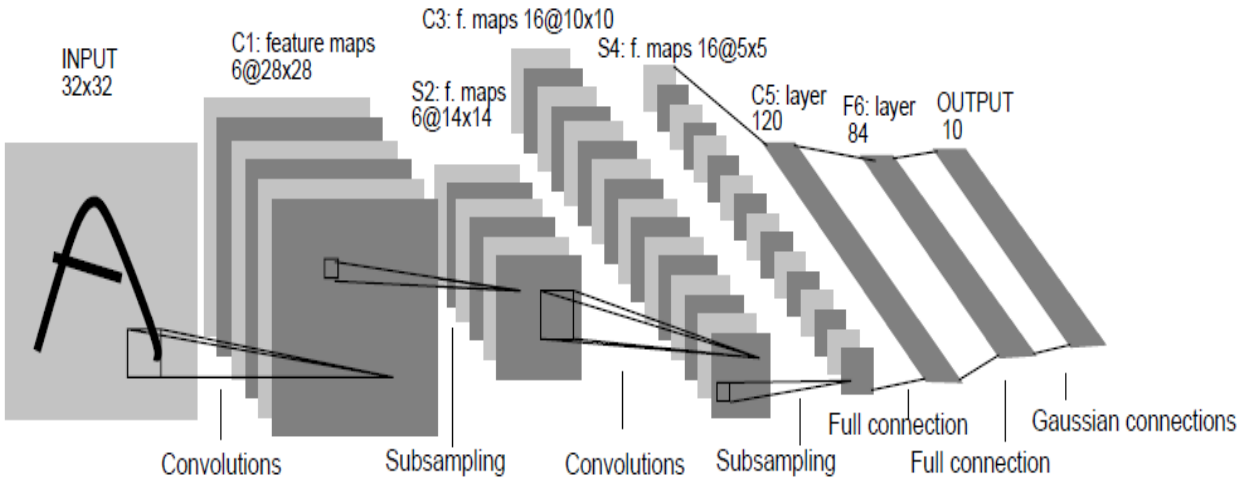

Figura 1. Arquitectura de LeNet-5 para reconocimiento de dígitos.
Fuente: Gradient-based learning applied to document recognition (Lecun et al. 1998)

Como el modelo (basado en la arquitectura LeNet-5) ya fue definido, se realizó la construcción del modelo donde se especificó las condiciones del entrenamiento. Para construir el modelo se usó la función **compile()**, propia de la librería Tensorflow. Lo primero que se declaró en la función, fue la operación de optimización. Se utilizó el optimizador "**ADAM**", el cual se utiliza para encontrar los valores correctos de los parámetros. Se escogió a "**Sparce Categorical Crossentropy**" como la función de costo. Se escogió está función por las características de la entrada, ya que se tiene etiquetas dispersas y las clases son exclusivas.

Antes de entrenar la red neuronal convolucional, se usó la función **summary()**, la cual nos muestra las capas del modelo, sus características y también la cantidad de parámetros de cada capa, como se observa en la Figura 2.

DOI: http://dx.doi.org/10.46925//rdluz.32.16

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 5)         50
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 5)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 16)        736
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 400)               0
_________________________________________________________________
dense (Dense)                (None, 120)               48120
_________________________________________________________________
dense_1 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_2 (Dense)              (None, 10)                850
=================================================================
Total params: 59,920
Trainable params: 59,920
Non-trainable params: 0
```

Figura 2. Resumen de las capas de un CNN
Fuente: Elaboración propia

Para entrenar el modelo creado, se utiliza la función **fit()**, en la cual se especificó el conjunto de datos que se van a entrenar: **model.fit (training_images ,training_labels ,epochs=6)**. Para indicar cuantas veces se va repetir el entrenamiento, se iguala a epochs (palabra reservada del método) al valor deseado. En este caso se igualó a 6 porque con esta cantidad de entrenamiento el modelo supera el 90% de exactitud, como se muestra en la figura 3.

Para verificar si la red logró aprender, se usa la función evaluate (). A esta función se le ingresan datos que la red no vio durante el entrenamiento. En esta función es donde se hizo uso de las imágenes y etiquetas de prueba: **test_loss = model.evaluate (test_images ,test_labels).** Estos resultados se muestran en la tabla 1 y en la figura.

DOI: http://dx.doi.org/10.46925//rdluz.32.16

```
Epoch 1/6
1875/1875 [==============================] - 24s 13ms/step - loss: 0.5287 - accuracy: 0.8046
Epoch 2/6
1875/1875 [==============================] - 24s 13ms/step - loss: 0.3625 - accuracy: 0.8658
Epoch 3/6
1875/1875 [==============================] - 24s 13ms/step - loss: 0.3218 - accuracy: 0.8809
Epoch 4/6
1875/1875 [==============================] - 24s 13ms/step - loss: 0.2954 - accuracy: 0.8898
Epoch 5/6
1875/1875 [==============================] - 24s 13ms/step - loss: 0.2729 - accuracy: 0.8974
Epoch 6/6
1875/1875 [==============================] - 24s 13ms/step - loss: 0.2557 - accuracy: 0.9034
313/313 [==============================] - 2s 6ms/step - loss: 0.3084 - accuracy: 0.8861
```

Figura 3. Datos del entrenamiento de la red neuronal convolucional.
Fuente: Elaboración propia

Con la red neuronal convolucional entrenada, se procedió a extraer sus parámetros como los Pesos y Bias de cada capa. Estos parámetros se guardaron en un archivo con formato .MIF para inicializar las memorias RAM en la FPGA. Los datos de los parámetros fueron convertidos a un binario de punto flotante, como se requieren en el formato .MIF. Para cada Peso y Bias se inicializa una memoria. Como se muestra en la figura 2, se tiene 5 etapas y en cada una se obtiene los Pesos y Bias de la CNN. Como menciona (Belean, 2018) la imagen a filtrar se almacena en una memoria. Los parámetros como los pesos y bias de las capas de convolución y Fully Connected también son almacenadas en una memoria. Entonces fue necesario inicializar 11 memorias independientes en la FPGA. Son 10 memorias que corresponden a los Pesos y Bias, y 1 memoria donde se inicializa la imagen que será procesada.

2.2. Diseño de la arquitectura de una red neuronal convolucional en VHDL

En el diseño del hardware se utilizaron IP Cores proporcionados por el Software Quartus, para realizar las operaciones de punto flotante e inicializar bloques de memorias. Se utilizó un enfoque de diseño modular, para subdividir el sistema en partes más pequeñas. La subdivisión de la red neuronal convolucional se basa en la arquitectura creada en Python.

2.2.1. Convolución 1

DOI: http://dx.doi.org/10.46925//rdluz.32.16

El bloque de convolución 1, se subdivide en 3 módulos. El primer módulo se encarga de realizar la operación de filtrado, el segundo módulo se encarga de inicializar las memorias de los Pesos y Bias de la convolución 1 y el último modulo es el que controla todo el proceso de asignar los valores de los Pesos y Bias a la operación correspondiente. Antes de diseñar el filtro de 3x3 se configuran los IP Cores de Suma y Multiplicación de punto flotante. Luego se generar las conexiones entre ellos; como el filtro es de dimensión 3x3, la cantidad de parámetros correspondientes a los pesos son 9. Como ni una de las 9 multiplicaciones depende de la otra, estas se pueden realizar en paralelo. De la misma forma se procedió a sumar los resultados de las multiplicaciones.

En la figura 4 se muestra el hardware generado por VHDL. El módulo filtro_3x3 tiene como entradas los valores de la imagen, los Pesos, Bias y como salida el resultado de la operación de convolución para una parte de la imagen.

El segundo módulo se encarga de calcular el tiempo que demora el módulo de filtro, a partir del valor de la entrada. El tiempo que demora el módulo en darnos una respuesta está en el rango de 7 a 42 ciclos de reloj. Este tiempo depende del tipo de entrada que tengamos como se muestra en la figura 5. El módulo para el filtro de 3x3 se desarrolló la convolución en 5 etapas. La primera etapa consiste en la multiplicación de los pesos con los valores de la imagen. En las siguientes etapas se implementó la suma de los resultados de las multiplicaciones. En la última etapa se implementó la suma del Bias con el resultado total de las sumas anteriores.

En el módulo de control se implementó una máquina de estados compuesta por cuatro estados. El estado 1 es donde se asignan las direcciones para acceder a los datos de las memorias inicializadas. En el caso de la dirección de los pesos, los cinco primeros datos le corresponden al Peso 1, los cinco siguientes al Peso 2 y así sucesivamente. Por este motivo, la dirección del primer peso es “0 + k”, el segundo peso es “5 + k”, y así hasta el noveno peso que es “40 + k”. Se utilizó una variable “k” inicializada en 0, que aumente de 1 en uno. De la misma forma se hizo para acceder a los datos de la imagen, utilizando dos variables que representan el movimiento del filtro sobre la imagen, como se muestra en la figura 6. Este estado tiene una condición, que cuando K sea igual a 5 todo se detiene. Este 5 representa la cantidad de filtros en la primera convolución.

DOI: http://dx.doi.org/10.46925//rdluz.32.16

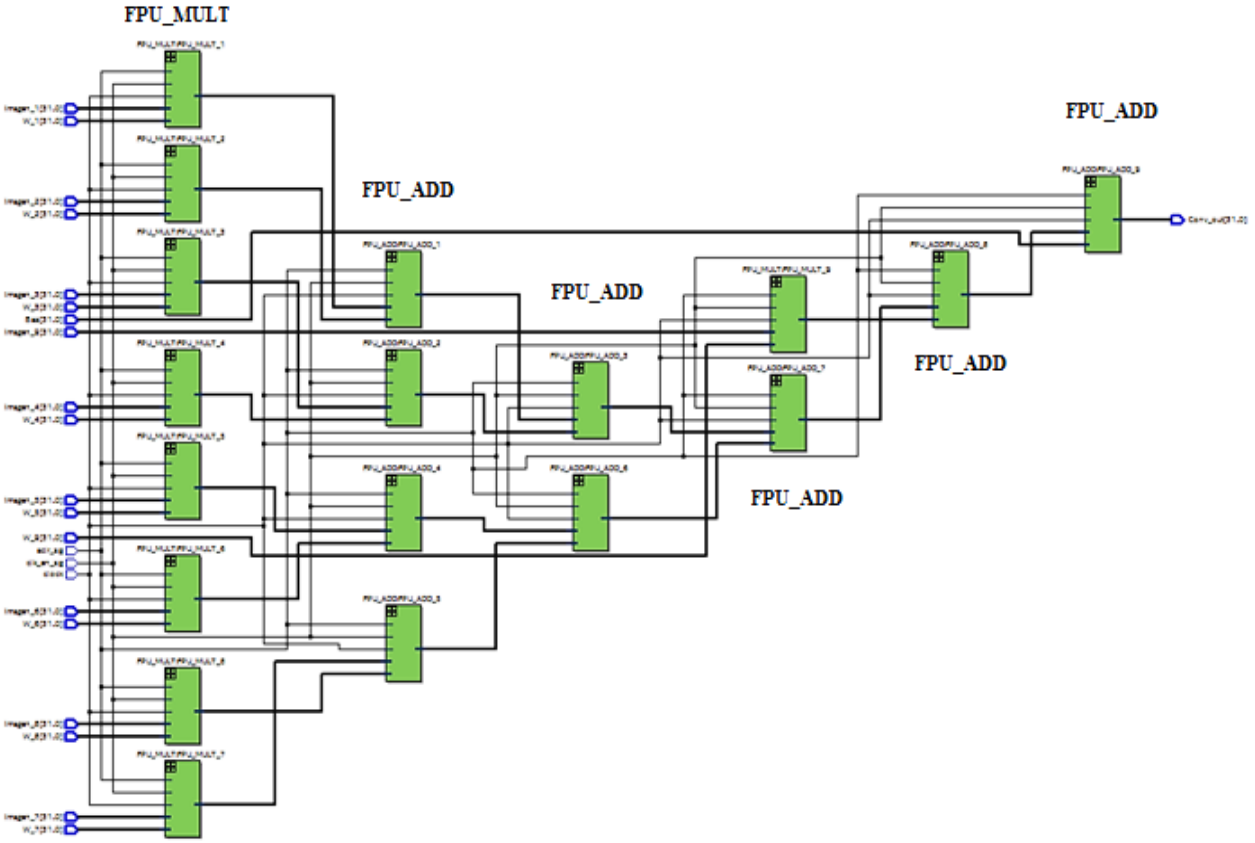

Figura 4. Modulo Filtro 3x3.
Fuente: Elaboración propia

```
Process(clock) is
   variable comparador : STD_LOGIC_VECTOR(31 DOWNTO 0);
     begin
       if rising_edge(clock) then
         comparador := Imagen_1_sig or Imagen_2_sig or Imagen_3_sig or Imagen_4_sig
                       or Imagen_5_sig or Imagen_6_sig or Imagen_7_sig or Imagen_8_sig;
         if(Imagen_9_sig = "00000000000000000000000000000000") then
            if(comparador = "00000000000000000000000000000000" ) then
               limite_out_sig<= 7;
            else
               limite_out_sig<= 42;
            end if;
         else
            if(comparador = "00000000000000000000000000000000" ) then
               limite_out_sig<= 21;
            else
               limite_out_sig<= 42;
            end if;
         end if;
       end if;
end Process;
limite_out<= limite_out_sig;
```

Figura 5. Máquina de Estados para determinar el tiempo del módulo Filtro_3x3
Fuente: Elaboración propia

DOI: http://dx.doi.org/10.46925//rdluz.32.16

```
when ESTADO_1 =>

if (k = 5) then
   stop <= '1';
else

 control_out<= '0';
 counter<=0;
 result_sig <= "00000000000000000000000000000000";
 enable<='1';
 reinicio<='0';
 address_W_1 <= std_logic_vector(to_unsigned(0 +k,6));
 address_W_2 <= std_logic_vector(to_unsigned(5+k,6));
 address_W_3 <= std_logic_vector(to_unsigned(10+k,6));
 address_W_4 <= std_logic_vector(to_unsigned(15+k,6));
 address_W_5 <= std_logic_vector(to_unsigned(20+k,6));
 address_W_6 <= std_logic_vector(to_unsigned(25+k,6));
 address_W_7 <= std_logic_vector(to_unsigned(30+k,6));
 address_W_8 <= std_logic_vector(to_unsigned(35+ k,6));
 address_W_9 <= std_logic_vector(to_unsigned(40+k,6));
 address_Bias <= std_logic_vector(to_unsigned(0+k,3));
 address_Imagen_1 <= std_logic_vector(to_unsigned(i + j*28,10));
 address_Imagen_2 <= std_logic_vector(to_unsigned(i+1 + j*28,10));
 address_Imagen_3 <= std_logic_vector(to_unsigned(i+2+ j*28,10));
 address_Imagen_4 <= std_logic_vector(to_unsigned(i+28+ j*28,10));
 address_Imagen_5 <= std_logic_vector(to_unsigned(i+29+ j*28,10));
 address_Imagen_6 <= std_logic_vector(to_unsigned(i+30+ j*28,10));
 address_Imagen_7 <= std_logic_vector(to_unsigned(i+56+ j*28,10));
 address_Imagen_8 <= std_logic_vector(to_unsigned(i+57+ j*28,10));
 address_Imagen_9 <= std_logic_vector(to_unsigned(i+58 + j*28,10));
 State   <= ESTADO_2;

end if;
```

Figura 6. Estado 1 del módulo de convolución 1
Fuente: Elaboración propia

En el estado 2 se espera que el módulo filtro 3x3 nos entregue el resultado. El tiempo que se permanece en este estado depende de lo datos de la imagen. Como se trabaja con imágenes a escala de grises, mucho de los datos de la imagen van a ser 0. El tiempo que demore el módulo filtro 3x3 puede ser 7, 21 o 42 ciclos de reloj, como se muestra en la figura 5.

En el estado 3 se realizan 2 sentencias "IF", como se muestra en la figura 7. La primera sentencia es para verificar las variables que realizan el desplazamiento, en este caso la variable "i" representa el desplazamiento horizontal y "j" el desplazamiento vertical. La segunda sentencia es para realizar la función de activación ReLU, la cual se implementó con una función IF evaluando el bit 32 del resultado del filtro_3x3. Este bit representa el signo del resultado. Si el resultado es negativo la salida de la convolución es 0 pero si es positiva, la salida de la convolución es el resultado obtenido por el módulo filtro 3x3.

DOI: http://dx.doi.org/10.46925//rdluz.32.16

```
when ESTADO_3 =>

   if (i = 26) then
    i <= 0;
    j <= j+1;
   end if;

   if result_sig(31) = '0' then
      relu <= result_sig;
   else
      relu <= "00000000000000000000000000000000";
   end if;

   control_out<= '1';
   enable<='0';
   reinicio<='1';
   State   <= ESTADO_4;
```

Figura 7. Estado 3 del módulo de convolución 1
Fuente: Elaboración propia

Las primeras CNN fueron entrenados con funciones TanH o Sigmoides, pero los modelos recientes emplean la función de Unidad Lineal Rectificada (ReLU) que otorga tiempos de entrenamiento más rápidos y menos complejidad computacional, como se destaca en (Krizhevsky et al. 2012). Para implementar la función de activación ReLU, se comparó el bit más significativo del resultado. Si es '0', se envía el resultado, si no se envía un 0 en su lugar. Cuando el bloque de convolución 1 tiene una respuesta, esta se guarda en un bloque intermedio antes de ser procesado por el bloque de Maxpooling 1. En el último estado evaluamos a la variable "j". Si la variable j es igual a 26, significa que el filtro ya se desplazó por toda la imagen. Si esto se cumple, a la variable "k" se le aumenta en 1.

2.2.2. Etapa intermedia entre la convolución 1 y maxpooling 1.

Para guardar los datos de la convolución, una opción es ir guardándola en una memoria y luego realizar la operación de Maxpooling. Si se realiza esta opción, el módulo del Maxpooling tendría que esperar hasta que termine la convolución, cuando para implementar la operación de Maxpooling no se necesita tener todos los resultados de la convolución, sino una parte de ellos. Como el Maxpooling es de 2x2, solo necesita las dos primeras filas de la nueva imagen que se genera a partir de aplicar una convolución. Como los resultados de la convolución van generando

DOI: http://dx.doi.org/10.46925//rdluz.32.16

fila por fila de la nueva imagen, se decidió usar un FIFO ya que funciona como una cadena de datos donde el primer dato en entrar, es el primero en salir y no necesita una dirección para acceder a los datos.

2.2.3. Maxpooling 1

El bloque de Maxpooling 1 se diseñó para que espere una señal del bloque anterior antes de iniciar su operación. En este bloque se reciben 4 datos, los cuales se comparan hasta encontrar el mayor. Este bloque realiza sus operaciones al mismo tiempo que el bloque de convolución 1 trabajando. Cuando los FIFOs del bloque anterior se quedan sin datos, envían una señal para que las operaciones de Maxpooling se detengan, hasta que se tenga los siguientes datos que se van a procesar.

Los resultados de la convolución 1 y el maxpooling 1 se guardan en una memoria. Se decidió guardar en una memoria en lugar de un FIFO porque este resultado se convertirá en la entrada de la segunda convolución y en la memoria podemos acceder a sus datos muchas veces, a diferencia de un FIFO, que solo se puede acceder una sola vez a sus datos.

2.2.4. Convolución y Maxpooling 2

El bloque de convolución 2 utiliza un módulo más que la convolución 1. Esto se debe a que la entrada para la convolución 2, ya no es de una profundidad de 1, si no que ahora es de 5. En el módulo de filtrado, se retira la suma del bias y el módulo de inicializaron de los parámetros se retira la inicialización de los valores del bias. En estos 2 módulos solo esta modificación se realizó, manteniendo el mismo funcionamiento que en los módulos anteriores. El módulo que se agrega en este bloque se encarga de sumar los 5 resultados de aplicar un filtro de dimensión 3x3x5. En este módulo se inicializa los valores del bias y se le suma el valor correspondiente al resultado, para luego aplicar la función de activación Relu.

También se utilizó un bloque intermedio para guardar los resultados de la Convolución 2. Este bloque mantiene la lógica del bloque intermedio anterior. Solo se modificó la dimensión del FIFO. El bloque de Maxpooling 2, se mantiene igual en configuración y en lógica que el

DOI: http://dx.doi.org/10.46925//rdluz.32.16

bloque de Maxpooling 1. Al igual que cuando se obtuvieron los resultados de convolución y Maxpooling 1, los resultados de la convolución y maxpooling 2, se almacenan en una memoria.

2.2.5. Fully Connected

Las redes neuronales son la tecnología de inteligencia artificial más efectiva y apropiada para el reconocimiento de patrones como menciona (Panchal et al. 2011). A diferencia de los bloques de convolución que necesitan los 9 datos al mismo tiempo, el bloque de Fully Connected se diseñó para que procese los datos de 4 en 4. Este bloque se divide en 3 módulos: Red neuronal 1, Red neuronal 2 y Red neuronal 3.

- Red Neuronal 1

Este módulo de Red neuronal 1 se subdivide en 3, como se muestra en la figura 8. Estos módulos se encargan de multiplicar la entrada por los Pesos, sumar el Bias y usar la función de activación Relu. En el primer módulo es el encargado de la multiplicación entre la entrada y sus pesos. En el segundo módulo se suma los datos obtenidos de las multiplicaciones y se realiza en 2 etapas, donde se suman 4 datos en paralelo como se muestra en la figura 9. En el tercer módulo, al dato que entra se le sumo el valor del Bias y se usó una función de activación Relu. Los resultados de las 120 neuronas se guardan en una memoria para la siguiente capa.

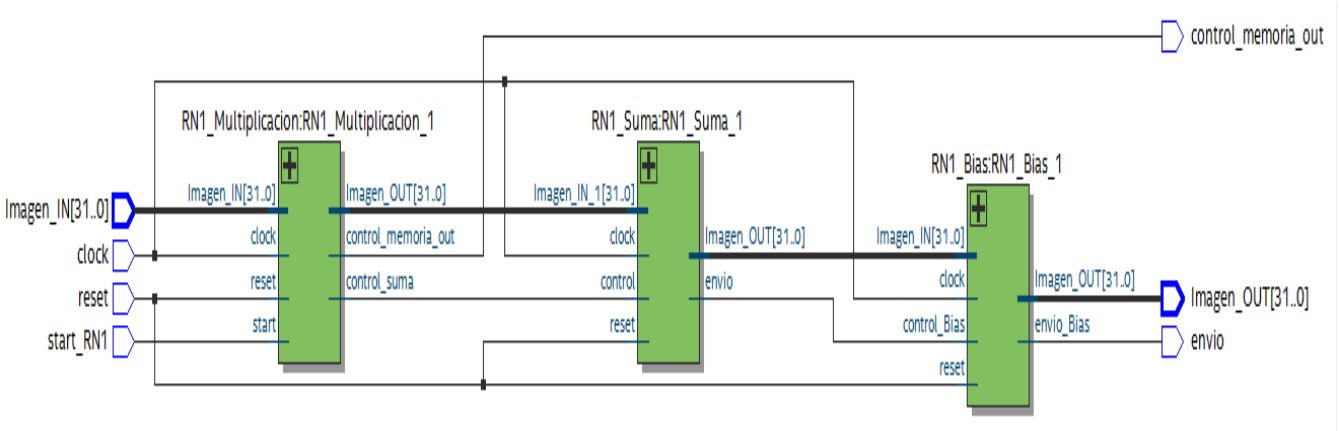

Figura 8. Red Neuronal en Hardware.
Fuente: Elaboración propia.

DOI: http://dx.doi.org/10.46925//rdluz.32.16

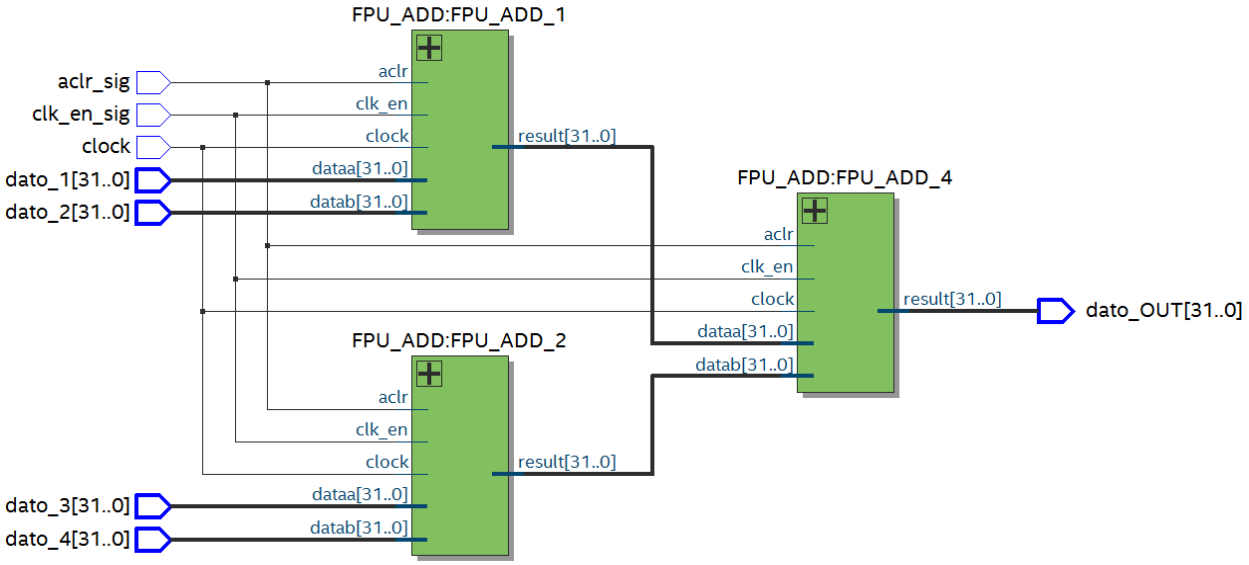

Figura 9. Módulo sumador de 4 datos
Fuente: Elaboración propia.

- Red Neuronal 2 y 3

La red neuronal 2 y 3 tienen la misma lógica que la red neuronal 1. La red neuronal 2 se compone por 84 neuronas y la red neuronal 3 por 10 neuronas. La modificación que se realizó para llevar a cabo estas 2 capas fue en el segundo módulo, ya que en la red neuronal 2 la capa de entrada tiene 120 datos, entonces la suma de todos estos datos se realizó en 29 sumas. Para la red neuronal 3, se realizan 20 sumas porque su entrada son 84 datos.

A diferencia de la Red neuronal 1 y 2, en la red neuronal 3 no se implementa la función de activación Relu. En su lugar se implementa la función Softmax, la cual nos indica a que clase pertenece la entrada, dependiendo de la neurona que tenga el valor más alto. Debido a que es un método clasificador, la función de activación Softmax obliga a la salida de la red neuronal a representar la probabilidad de que la entrada caiga en cada una de las clases. Sin el Softmax, los resultados de la neurona son simplemente valores numéricos (Heaton, 2015). Para implementar la operación de e^x , se utilizó el IP Core ALTFP_EXP. Es IP Core tiene como entrada el clock y el exponente. Adicionalmente se le agregaron 2 entradas más para tener un mayor control del IP Core. La respuesta se obtiene después de 17 ciclos de reloj, como se indica en (Altera

DOI: http://dx.doi.org/10.46925//rdluz.32.16

Corporation. 2018). Para la operación de división de punto flotante, se utilizó el IP Core ALTFP_DIV. A este IP Core también se le agrego 2 entradas para un mayor control. Se seleccionó una salida con una latencia de 6.

2.2.6. Diseño de la comunicación entre el HPS y la FPGA

Intel proporciona algunas herramientas para que el diseño de hardware. Entre estas herramientas se tienen algunos proyectos que sirven como ejemplos. Uno de estos proyectos es "Control panel". En este proyecto que se encuentra disponible en De10-Nano CD-ROM, ya se estable una comunicación entre el HPS y la FPGA. A este proyecto se le hicieron algunas modificaciones para que se pueda interactuar entre la CNN implementada en la FPGA con el HPS.

La CNN implementada en la FPGA, tiene 1 entrada y 2 salidas. La entrada cumple la función de activar el sistema para que inicie sus operaciones. La entrada tiene una dimensión de 1 bit ya que solo se requiere que sea "1" ó "0". Las salidas son la etiqueta que nos indica a que clase pertenece la imagen evaluada y el contador que nos indica la cantidad de ciclos por reloj que demora el sistema en darnos la respuesta.

3. Discusión

Usando hardware reconfigurable como las FPGA se resuelve el problema el desequilibrio entre la eficiencia energética y el rendimiento, que se presentan al usar otras arquitecturas de Hardware como menciona (Tu et al. 2017). Según (Cong, Xiao, 2014) el rendimiento en la CPU no puede cumplir los requisitos de procesamiento en tiempo real en los dispositivos integrados. El consumo de energía de la GPU es demasiado alto para la informática integrada (235 W para la GPU NVIDIA Tesla K40) como menciona (NVIDIA, 2013).

Para comparar si la CNN implementada en una plataforma SoC basada en FPGA nos da una respuesta cercana o igual al Software y si esta respuesta es más rápida, se evaluaron imágenes de cada clase y se compararon.

DOI: http://dx.doi.org/10.46925//rdluz.32.16

```
f1 = activation_model.predict(training_images[1].reshape(1, 28, 28, 1))[7]
print(f1)
```

```
[[9.96830523e-01 2.12138973e-10 2.54809549e-07 7.54501457e-07
  2.05184825e-09 8.44953827e-13 3.16854287e-03 1.21883252e-17
  2.35816297e-11 1.08030785e-13]]
```

Figura 10. Resultados de evaluar la Imagen de la clase 1 en Software
Fuente: Elaboración propia

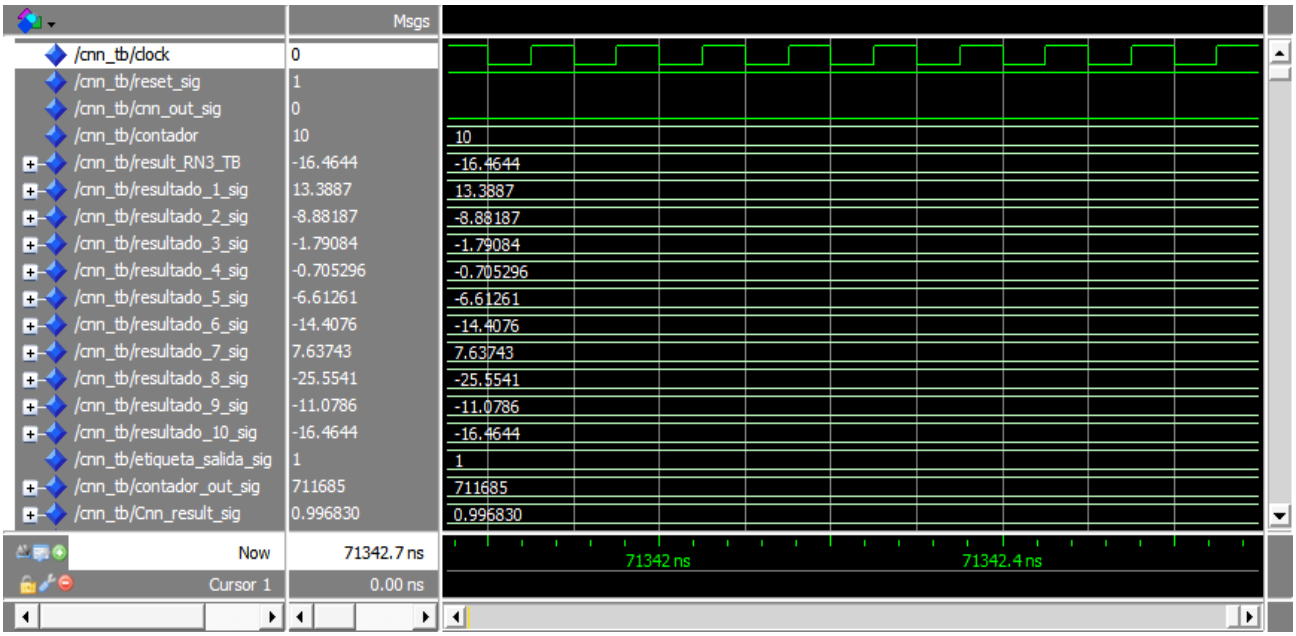

Figura 11. Resultados de evaluar la Imagen de la clase 1 en Hardware
Fuente: Elaboración propia

En la figura 10, se muestra los resultados de la red neuronal 3 implementada en Software, luego de aplicar la función de activación Softmax a cada una de las neuronas. Como se observa en la figura, el primer valor es el mayor entre los 10 valores obtenidos. Este valor es de 0.996830. Esto hace referencia a que la imagen evaluada pertenece a la clase 1.

En la figura 11, se muestra los resultados obtenidos de evaluar la Imagen 1 en Hardware. Las señales que se presentan en la figura 11 corresponden a los resultados obtenidos en la última capa de la CNN(Red Neuronal 3) previamente a ser ingresados por una función de activación. Como se describió en el módulo de Softmax, el sistema en hardware solo nos da el dato de la

DOI: http://dx.doi.org/10.46925//rdluz.32.16

neurona con el valor mayor. La variable **Cnn_result_sig** nos muestra el resultado obtenido, el cual es 0.996830 igual al resultado obtenido en Software. La variable **etiqueta_salida_sig** nos indica que el resultado que se obtuvo en hardware corresponde la imagen de la clase 1. El hardware también nos da 10 resulta más, los cuales corresponde a cada neurona de la red neuronal 3, antes de implementarles la función de activación Softmax. Estos valores se muestran con motivos de verificación del sistema. La variable **contador_out_sig** nos da como respuesta 711685 los cuales representan los ciclos de reloj que le toma al sistema evaluar la imagen.

Como los resultados obtenidos en Software y Hardware son los mismos, se procedió a verificar los tiempos que le toman a cada uno darnos la respuesta. En Python las clases están en numeradas de 0 a 9 y en Hardware se consideraron de 1 a 10, siendo 0 en Python equivalente a 1 en Hardware y así sucesivamente hasta 9 equivalente a 10. En la figura 3 muestra la evaluación de la imagen perteneciente a la clase 1. En este código fue ejecutado en Python y nos muestra que para evaluar la imagen de la clase 1, le tomó 32.5534 ms.

En cambio, en la figura 12 se muestra la evaluación en hardware, 14.2337 ms, el tiempo que le toma a la FPGA evaluar la imagen. En este tiempo no se tomó en cuenta la demora en la comunicación entre la FPGA y el HPS. El sistema implementado en Hardware trabaja a una frecuencia de 50MHZ, la cual puede ser aumentada, pero para estas evaluaciones se usó la de 50MHZ que es la frecuencia regularmente utilizada en implementaciones con FPGA.

```
import time
inicio_de_tiempo = time.time()
classifications = model.predict(training_images[1:2])
tiempo_final = time.time()
tiempo_transcurrido = tiempo_final - inicio_de_tiempo
print ("Tomo milisegundos : " + str(tiempo_transcurrido*1000))
print("Categoria de la imagen evaluada : " + str(np.argmax(classifications[0])))
```

```
Tomo milisegundos : 32.55343437194824
Categoria de la imagen evaluada : 0
```

Figura 12. Evaluación del tiempo para dar una respuesta en Software.
Fuente: Elaboración propia

DOI: http://dx.doi.org/10.46925//rdluz.32.16

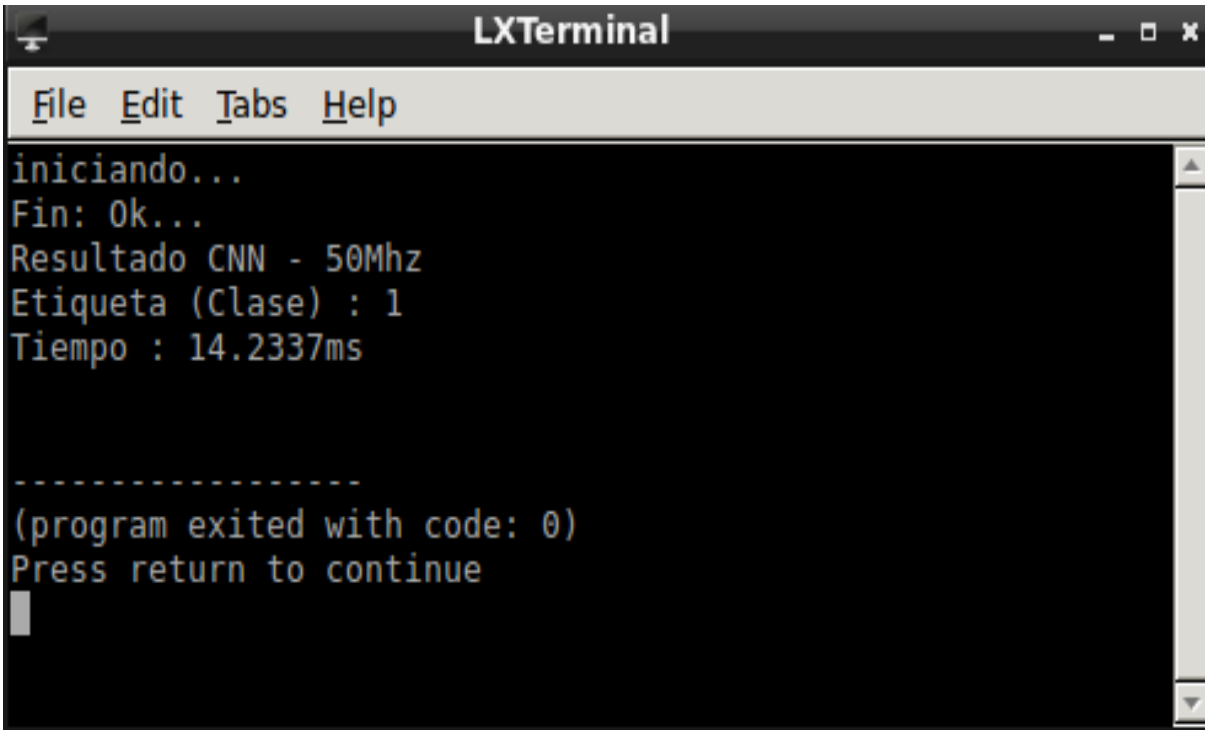

Figura 13. Evaluación del tiempo para dar una respuesta en Hardware.
Fuente: Elaboración propia

En la tabla 1, se muestra una comparación para analizar el porcentaje de acierto en la implementación de la red neuronal convolucional en Software y Hardware. Además, se indica el análisis de las imágenes que pertenecen a cada clase de la base de datos MNIST. Asimismo, también se ofrece la comparación para analizar el tiempo de respuesta de la implementación de la red neuronal convolucional en Software y Hardware. Como se observa en la tabla, al evaluar una imagen de cada clase, la respuesta que se obtiene usando un hardware reconfigurable como la FPGA es mucho menor que la respuesta obtenida en software. El tiempo en la FPGA está condicionado por la frecuencia de reloj y si esta frecuencia se aumenta, la respuesta podría ser tener tiempo menor al que se obtuvo usando la frecuencia de 50Mhz.

En esta tabla se evidencia que la implementación en Hardware nos puede dar un resultado igual al implementado en Software, pero en un menor tiempo.

DOI: http://dx.doi.org/10.46925//rdluz.32.16

Tabla 1. Comparación para analizar el porcentaje de acierto y el tiempo de respuesta en la implementación de la red neuronal convolucional en Software y Hardware.

N°	Categoría de la imagen	Posición de la imagen evaluada en el dataset	Porcentaje de acierto por Software (%)	Porcentaje de acierto por Hardware (%)	Tiempo de respuesta por Software(ms)	Tiempo de respuesta por Hardware a 50MHZ (ms)
1	Camiseta	Training_Imagen[1]	99.68304	99.6830	32.5534	14.2337
2	Pantalón	Training_Imagen[16]	99.99855	99.9986	32.4750	13.5308
3	Chompa	Training_Imagen[5]	99.62780	99.6278	33.6964	14.5511
4	Vestido	Training_Imagen[3]	85.52240	85.5224	32.7420	14.08348
5	Abrigo	Training_Imagen[19]	76.37630	76.3763	33.5312	14.45896
6	Sandalia	Training_Imagen[8]	99.99999	1.0000	32.0008	13.4847
7	Camisa	Training_Imagen[18]	86.86295	86.8630	33.1018	14.0915
8	Zapatilla de deporte	Training_Imagen[6]	99.88767	99.8877	31.3692	12.9391
9	Cartera	Training_Imagen[23]	99.10351	99.1035	33.7610	14.6761
10	Bota	Training_Imagen[0]	99.44180	99.4418	32.8994	13.8432

Conclusiones

- El diseño de la red neuronal convolucional en Python, hizo posible entrenar un clasificador de objetos en una imagen. Este diseño en software nos permitió comprobar la respuesta de la estructura antes de ser implementada en hardware, siendo en gran parte del diseño en la FPGA, un comparador de resultado de cada módulo implementado.

- En esta investigación se demuestra que, si se desea implementar estos algoritmos y obtener una respuesta rápida, las FPGAs son ideales para estas tareas, ya que no solo nos dan

DOI: http://dx.doi.org/10.46925//rdluz.32.16

una respuesta rápida, sino que también nos da respuesta con una gran exactitud, siendo similar en Software como en Hardware.

- Con el diseño del hardware de la plataforma SoC para comunicar la FPGA con el HPS, se obtuvieron los resultados de la red neuronal convolucional implementada en la FPGA, en un sistema operativo que se ejecuta en el HPS. Estos datos fueron expuestos en un terminal para comprobar que la comunicación es correcta. Los resultados de esta comunicación fueron satisfactorios, mostrando la etiqueta de manera correcta, correspondiente a la imagen de entrada.

Referencias

ALTERA CORPORATION. (2018). Floating Point Exponent (ALTFP_EXP) - Megafunction User Guide. 1st ed. Altera corpotation.

Belean, B. (2018). Application-Specific Hardware Architecture Design with VHDL. 1st ed. Cham: Springer.

Cong, J., Xiao, B. (2014). Minimizing computation in convolutional neural networks, in International Conference on Artificial Neural Networks (ICANN). Springer, pp. 281–290.

Cross, N. (2001). Métodos de diseño: estrategias para el diseño de productos. México, D.F.: Limusa.

Géron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems (2.a ed.). California, United State: O'Reilly Media.

Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep Learning} (1.a ed.). Cambridge, Massachusetts, Estados Unidos: MIT Press.

Heaton, J. (2015). Artificial Intelligence for Humans, Volume 3: Deep Learning and neural network (1.a ed., Vol. 3). Scotts Valley, California, Estados Unidos: CreateSpace Independent Publishing Platform.

Hernández, R., Fernández, C., Baptista, M. (2014). Metodología de la Investigación (6ta ed.). México: McGranw-Hill/ interamericana.

DOI: http://dx.doi.org/10.46925//rdluz.32.16

Krizhevsky, A., Sutskever, I., Hinton Geoffrey E., Geoffrey E Hinton. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems - NIPS'12, pp. 1–9.

Lecun, Y., Bottou, I,. Bengio, Y., Haffner, P. (1998). Gradient-based learning applied to document recognition". Proceedings of the IEEE, 86(11). 2278-2324.

Mittal, S. (2018). A Survey of FPGA-based Accelerators for Convolutional Neural Networks. Neural Computing and Applications. 10.1007/s00521-018-3761-1.

Mittal, S. y Vetter, J. (2015). A Survey of Methods for Analyzing and Improving GPU Energy Efficiency, ACM Computing Surveys.

NVIDIA. 2013. Tesla k40 GPU active accelerator. Report, 2013.

Ovtcharov, K., Ruwase, O., Kim, J.-Y., Fowers, J., Strauss, K., Chung, E. S. (2015). Accelerating deep convolutional neural networks usingspecialized hardware, Microsoft Research Whitepaper, 2 (11).

Panchal, G., Ganatra, A., Kosta, Y., Panchal, D. (2011). Behaviour analysis of multilayer perceptrons with multiple hidden neuronsand hidden layers. International Journal of Computer Theory and Engineering. 3 (2), 332–337.

Tu, F., Yin, S., Ouyang, P., Tang, S., Liu, L., Wei, S. (2017). Deep convolutional neural network architecture with reconfigurable computation patterns, IEEE Transactions on Very Large Scale Integration (VLSI)Systems.