Universidad Zaragoza
1542

Trabajo Fin de Grado

Implementación con FPGA de redes neuronales binarias

FPGA-based accelerator for Binary Neural Network

Autor/es

Luis Lozano Romeo

Director/es

Denis Navarro Tabernero

Escuela de Ingeniería y Arquitectura

2019

Repositorio de la Universidad de Zaragoza – Zaguan http://zaguan.unizar.es

0.- Resumen:

Este trabajo de fin de grado consiste en observar los beneficios de utilizar un menor tamaño de palabra para la cuantificación de los pesos en una red neuronal convolucional, o en inglés Convolutional Neuronal Network, sin perder eficiencia y precisión. Por ello se ha decidido utilizar para su implementación una FPGA donde podemos trabajar a nivel de bit.

En la mayoría de ocasiones que se encuentran propuestas de optimizaciones para redes neuronales artificiales se nos ofrece la posibilidad de reducir el número de filtros en las capas convolucionales o de neuronas en la capa fully conected, o de añadir más capas reduciendo su tamaño o viceversa. Pero rara vez se ha tratado de encontrar una solución a dichas optimizaciones mediante la reducción del tamaño de palabra para agilizar tiempos y costes de recursos.

La red neuronal utilizada ha sido LeNet, que consta de siete capas de los tipos convolucional (Conv), relu, full conected (FC) y max pooling (max). Esta red junto con la base de datos MNIST se encarga de identificar en una imagen un número del 0 al 9.

Para poder realizarlo, se han llevado a cabo una serie de procesos en los cuales se han utilizado herramientas como Caffe Ristretto, GuinnessMaster y Vivado HLS. La primera de ellas ha sido necesaria para poder entrenar la red y obtener los pesos cuantificados con el menor tamaño posible sin perder eficiencia en la red. Mediante GuinnessMaster se ha obtenido una plantilla de la red y sus capas en C sintetizable que nos ha permitido dar forma a dicha red. Por último, mediante Vivado HLS, hemos conseguido traducir a VHDL la red en C para poder introducirla en una FPGA y poder comprobar su tiempo de ejecución, área de silicio utilizada y parámetros de memoria que nos definirán lo útil que puede llegar a ser nuestra propuesta.

Índice

1.- Introducción:

1.1.- Redes neuronales convolucionales:

Una red neuronal convolucional o CNN, es un algoritmo de aprendizaje profundo (Deep learning), que es un tipo de aprendizaje automático (machine learning) en el que un modelo aprende a realizar tareas de clasificación directamente a partir de imágenes. Estas CNNs son realmente útiles en el reconocimiento de imágenes y su clasificación.

Este tipo de redes se ha vuelto cada vez más utilizado debido a que gracias a ellas, no se necesita una extracción manual de las características de las imágenes a clasificar, los resultados obtenidos de las redes neuronales artificiales bien entrenadas son muy precisos, con eficiencias de hasta el 99'8%, y nos permite reutilizar un mismo modelo de red entrenándolo con una diferente base de datos para realizar diferentes tareas.

Una CNN está compuesta por varias capas que permiten extraer diversas características de la imagen, tales como los bordes, el brillo o los tonos. Esta sucesión de capas tiene un carácter secuencial, puesto que la entrada de una capa es la salida de la anterior.

1.2.- LeNet-5:

En nuestro caso, vamos a desarrollar la arquitectura de LeNet-5 *[3]*, que es el modelo que hemos implementado. Esta red fue creada por Yann LeCun en 1988. Fue utilizada por una gran cantidad de bancos para reconocer los dígitos escritos a mano en los cheques. También es utilizada para la clasificación de imágenes, así como para la identificación de rostros y formas.

LeNet-5 está compuesta por 7 capas, contando la capa de entrada a la red, todas ellas con parámetros entrenables, es decir pesos. En nuestro caso, la entrada está formada por una imagen de 28X28 pixeles en blanco y negro. Sería posible adaptar la red para tratar con imágenes a color, utilizando una cuantificación RGB, pero supondría tratar tres veces cada imagen, una para cada color primario.

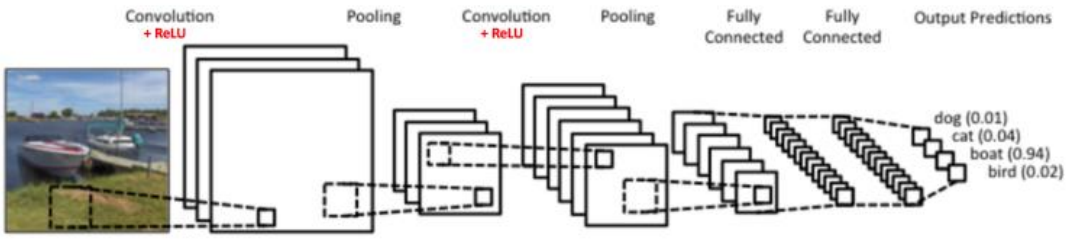

Figura 1: Esquema de LeNet-5

Las capas de dicha red pertenecen a tres tipos diferenciando sus funciones, convolución, max pooling y fully conected. Como se puede apreciar en la *Figura 1: Esquema de LeNet-5* utilizaremos una capa de entrada para cuantificar la imagen, dos capas convolucionales para utilizar diversos filtros, dos capas max pooling y dos capas fully conected. Siguiendo el esquema mencionado anteriormente, el tamaño de la imagen de entrada va a ser modificado en cada capa de la siguiente forma.

Partiremos de una imagen en blanco y negro de 28X28 pixeles, que será cuantificada en una matriz de 28X28 coeficientes. Dicha matriz será introducida en la primera capa convolucional que tiene una profundidad de 20 filtros con un núcleo de5X5 generando para cada filtro una salida de 24X24 coeficientes. Esta sucesión de salidas verá reducido su tamaño en la siguiente capa max pooling que presenta un núcleo 2X2 consiguiendo a su salida 20 imágenes con un tamaño 12X12. Dichas imágenes pasarán por la siguiente capa convolucional, que presenta una profundidad de 50 filtros generando 50 salidas de tamaño 8X8 que como sucedía anteriormente, se introducen en una capa max pooling 2X2 que reduce su tamaño a 50 imágenes 4X4.

A partir de este momento cada una de las muestras de las 50 imágenes 4X4 será considerada una entrada para cada neurona de la primera capa fully conected, por lo que tendremos un total de 800 entradas a dicha capa. Esta primera capa presenta una salida de 500 neuronas que serán la entrada de la segunda y última capa fully conected que presentará las 10 posibles salidas finales.

Para el tratamiento de esta gran cantidad de información, se ha utilizado en cada capa convolucional y en las fully conected una función de filtrado para la salida conocida como ReLu, que nos permite quedarnos únicamente con los valores positivos, puesto que los valores negativos no nos van a aportar ninguna información y por lo tanto son descartables y llevados al valor nulo, es decir, 0.

A continuación, se explicarán cada una de las capas más en profundidad y con las funciones matemáticas en las cuales están basadas.

1.2.1.- Capa de entrada

Como se ha mencionado anteriormente, la entrada de esta CNN será una imagen de 28X28 pixeles. Estos pixeles deberán ser cuantificados para poder ser tratados por la red. Para ello, deberemos tratar a la imagen como una matriz 28X28 en la que cada pixel tenga un equivalente numérico.

En nuestro caso, utilizaremos imágenes en escala de grises. Por ello, se ha establecido que el rango de valores entre los cuales trataremos los pixeles será de 0 para el color negro y 255 para el blanco. Por convención, se ha establecido que el rango de colores en una escala de grises es de 9 tonos diferenciados, por lo que únicamente nos encontraremos con 9 posibles valores en la entrada.

En el supuesto caso de que la imagen estuviera en color, se trataría cada uno de los colores primarios (RGB) por separado, como si se tratasen de tres imágenes diferentes y se obtendría una estimación del conjunto una vez tratadas las tres imágenes.

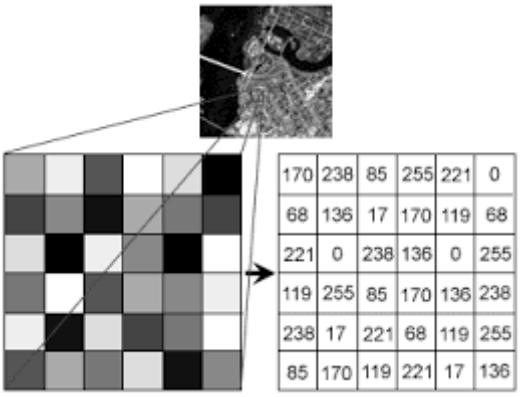

Figura 2: Codificación de una imagen en escala de grises

1.2.2.- Capa convolucional

Las redes neuronales convolucionales, extraen su nombre del operador convolución, que es el que permite el principal cometido de estas redes, el extraer las características de la imagen. Para ello se basan su funcionamiento en aplicar distintos filtros a determinadas partes de la imagen.

El funcionamiento de estas capas es el siguiente. Se realiza una convolución de una matriz de dimensiones NxN sobre la entrada de la capa tantas veces como posiciones distintas pueda tener dentro de la matriz MxM de entrada, dando lugar a una salida LxL. Este proceso se repite para Z filtros si queremos obtener Z salidas LxL.

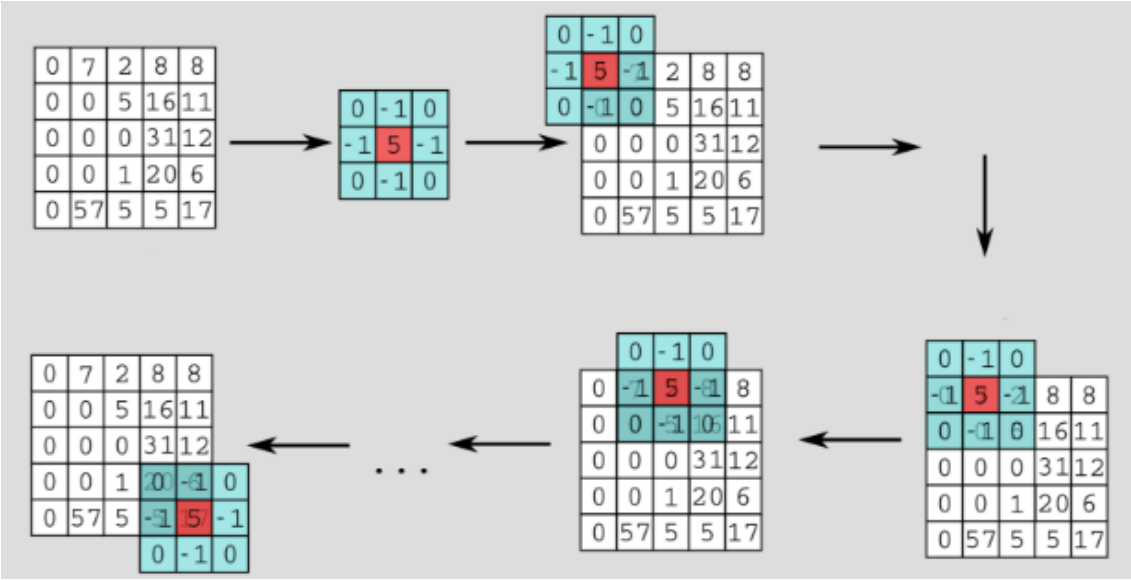

Figura 3: Ejemplo de una convolución gráfica de matrices

En términos de las CNNs, la matriz NxN se llama filtro y en nuestro caso las dimensiones que tomará serán de N = 5. Y la matriz de salida de la capa será conocida como mapa de activaciones con unas dimensiones equivalentes a las de la entrada. Además, se conoce como profundidad al número Z de filtros que utilizamos en cada capa convolucional, que son los que determinan el número de salidas por capa. Para la red que estamos utilizando, trataremos la primera convolución con una profundidad Z igual a 20 y la segunda convolución con una profundidad Z' equivalente a 50.

Hay que destacar que cuanta más profundidad presente nuestra red, mayor precisión potencial tiene, aunque supondrá una mayor cantidad de coste de computación por lo que podrá perder prestaciones en cuanto a tiempo de ejecución y procesamiento de la imagen.

La capa convolucional a su vez incorpora la función ReLu. Esta función trata de obtener una rectificación de la señal, de forma que los valores negativos pasen a ser cero y los positivos conserven su valor. Es equiparable a la función de rectificación de media onda en señales analógicas, y sigue la ecuación:

ReLu = max(0,X) siendo X la entrada a la función.

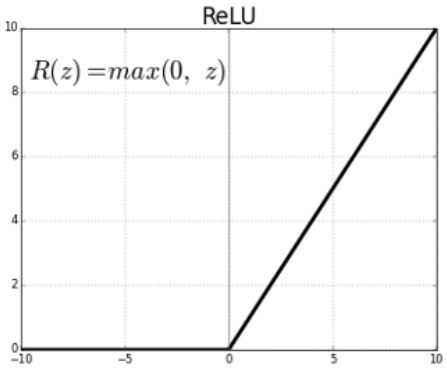

Figura 4: Función ReLu

1.2.3.- Capa MaxPooling:

La capa de MaxPooling es una capa de submuestreo no lineal. Esta capa divide la entrada en matrices no solapadas, en nuestro caso 5X5, de las cuales se queda el mayor de los datos.

La capa MaxPooling sigue la función:

$$B_n = Max(A_{ij}) \in n \leq L \,\&\, i,j \leq 5$$

Esta capa se utiliza por dos razones principales, reducir el número de operaciones que realizarán las capas superiores a esta, puesto que la matriz de salida será mucho menor que la entrada, y para proporcionar robustez adicional a la posición. En resumen, MaxPooling es una manera eficiente de reducir las dimensiones de representaciones intermedias que no aportan una información útil para la red.

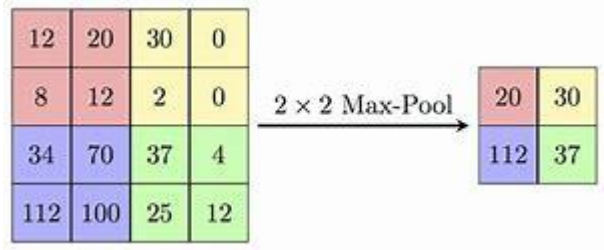

Figura 5: Ejemplo de MaxPooling

1.2.4.- Capa Fully Conected:

Esta es una de las últimas capas de la red y se utiliza para clasificar todas las salidas y realizar el trabajo de decidir cuál es el resultado más probable y que por lo tanto se considerará como correcto. En esta capa, tenemos una neurona por cada salida y por cada tipo de posible salida que tengamos presente en nuestra red, es decir si tenemos A salidas de la capa anterior, y B posibles salidas de la red, tendremos que tener A*B coeficientes en esta última capa.

Para poder entender esta capa correctamente, primero precisamos en explicar el funcionamiento de una neurona artificial. El concepto de neurona artificial, o elemento de proceso, fue definido por Rumelhart y McClelland [6] en 1986 como un sistema que a partir de un conjunto de entradas finitas genera una única salida.

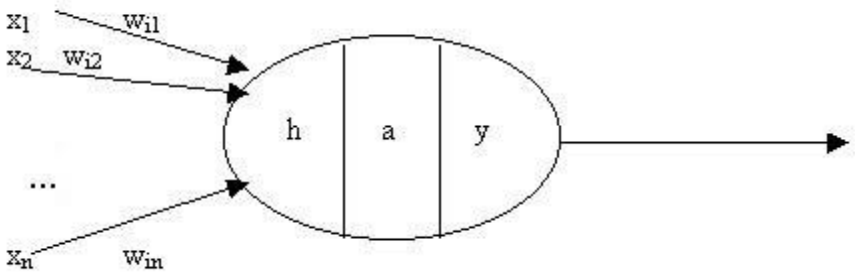

Figura 6: Modelo de una neurona artificial

Siguiendo la notación de la *Figura 6: Modelo de una neurona artificial*, cada neurona sigue la expresión matemática:

$$y = \sum_{n=0}^{N} x_n * w_n + a$$

Siendo Xn cada una de las entradas a la neurona, Wn los coeficientes para cada conexión y a el valor de activación de la bia.

Dependiendo de qué tipo de salidas presente la neurona artificial se podrá clasificar en neuronas binarias con salidas 0/1, neuronas bipolares con salidas -1/+1 o neuronas continuas con salidas de números reales.

Ante lo mencionado anteriormente, la principal característica de una neurona artificial es la capacidad de obtener una única salida a partir de múltiples entradas, por lo que, si conectamos las A entradas de nuestra capa a cada una de las B neuronas de salida de la capa, obtenemos nuestra red neuronal artificial de A*B coeficientes. De esta forma, obtenemos la definición de la capa fully conected, la cual expresa que esta capa presenta un conjunto de entradas completamente conectadas con todas las salidas como se puede apreciar en la *Figura 7: Diagrama de una Fully Conected layer*

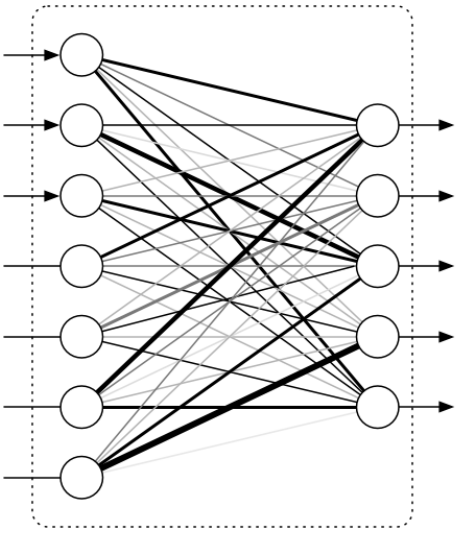

Figura 7: Diagrama de una Fully Conected layer

Esta capa es la más similar al concepto biológico de red neuronal, dado que nuestro cerebro está formado por neuronas interconectadas entre sí. Cada una de estas neuronas tiene una relevancia en función de su entrenamiento y de la cantidad de información que sea capaz de aportar. Esta relevancia se ve reflejada en el valor que posee su peso y su valor de activación. Esto mismo ocurre con las neuronas del cerebro humano, puesto que, para cada actividad, o entrenamiento específico, se recurre a la información obtenida por un área del cerebro específico o conjunto de neuronas.

Matemáticamente, la definición de esta capa es la extrapolación de la definición de la neurona artificial a un conjunto de neuronas formando la red, siendo:

$$y_n = \sum_{k=0}^{K} x_k * w_{nk} + a_n$$

Además, esta es la capa que más memoria necesita utilizar puesto que al estar todas interconectadas, se requieren grandes cantidades de bits para codificar todos los coeficientes, los cuales intentaremos reducir. En nuestro caso, partiremos de una entrada de 50 planos de dimensiones 4X4, lo que hace un total de 800 neuronas de entrada. Dichas neuronas están completamente conectadas mediante los 400.000 coeficientes necesarios con las 500 neuronas intermedias. Dichas neuronas intermedias tienen asociados las 10 salidas finales de la capa mediante 5.000 coeficientes. Todo ello genera una cantidad de 405.000 coeficientes en la red.

2.- Entrenamiento y extracción de los pesos:

En este apartado trataremos el entrenamiento de la red mediante la herramienta Caffe-Ristretto, la cual nos permite hacer un análisis tras su entreno para ver hasta qué punto podemos reducir el tamaño de las palabras manteniendo, casi en su totalidad, el porcentaje de éxito.

2.1.- Caffe-Ristretto:

Caffe es un entorno de aprendizaje profundo creado por Berkeley AI Research *[1]*. Caffe está diseñado en base a cinco aspectos clave que son el lenguaje, la velocidad, la modularidad, el código abierto y la comunidad. El lenguaje de módulos y optimizaciones está definido como esquemas de código en texto. La alta velocidad de la herramienta permite manejar grandes cantidades de datos y modelos en un tiempo razonable. Gracias a la capacidad de separar en módulos las funciones de la herramienta se consigue una gran flexibilidad y profundidad en la herramienta. Al ser un entorno de código abierto, se pueden utilizar una gran cantidad de módulos de referencia y código común para alcanzar los objetivos. Por último, la comunidad académica e industrial que utiliza la herramienta no deja de crecer ofreciendo diversos recursos y soluciones.

Dentro de este entorno, se han desarrolladas diferentes herramientas, pero la que a nosotros nos incumbe es Ristretto.

La herramienta Ristretto *[2]* realiza la cuantificación y evaluación automática de la red, utilizando diferentes anchos de bits para la representación numérica, para encontrar un buen equilibrio entre la compresión y la precisión de la red. Gracias a la integración de Ristretto en Caffe, los archivos de descripción de red se pueden cambiar para cuantificar diferentes capas. El ancho de bits utilizado para diferentes capas, así como otros parámetros, se puede establecer en el archivo prototxt de la red. Esto permite probar y entrenar directamente redes convolucionales, sin necesidad de recompilar.

2.2.- Definición de la red en Caffe:

Para definir una red en Caffe, como se ha mencionado anteriormente, hay que definir un archivo prototxt. En este archivo se descompone la red en las diferentes capas estableciendo sus propiedades y su orden.

Ver anexo 1: Definición de la red - lenet.prototxt

Todas las capas de la red comparten la misma estructura, nombre de la capa, tipo de datos, definición de los parámetros de entrada y salida de la capa y capa superior e inferior. En el caso de las capas convolucionales, aparecen además los parámetros de entrada y salida como el tamaño de la máscara del filtro N y la profundidad Z.

Como se puede apreciar en la primera capa convolucional, se le definen los parámetros de número de entradas, tamaño de la máscara y número de salidas. El número de entradas en este caso es 1 puesto que la entrada a esta capa es la imagen de entrada a la red. En el caso del número de salida, como ya se ha mencionado anteriormente, implica el número de filtros que se utilizarán para extraer las características de la imagen, en este caso serán 20 los filtros aplicados. Por último, el tamaño de la máscara es de 5, eso quiere decir que cada filtro tendrá asociada una matriz de pesos de 5X5.

2.3.- Entrenamiento de la red:

En este paso ya tenemos definidas una serie de capas genéricas y una estructura de la red, pero aun no podemos trabajar con ella como elemento útil puesto que no contiene ninguna información y por lo tanto si la hiciésemos clasificar una imagen no obtendríamos una salida correcta y fiable. Esto se debe a que los pesos de la red están “vacíos”, es decir con valor 0.

Para solucionar esto debemos entrenar la red. Hay diferentes formas de entrenar una CNN pero en nuestro caso utilizaremos un entrenamiento de tipo Backward propagation. Este método de entrenamiento utiliza una base de datos, en nuestro caso MNIST, la cual se introduce en la red con una etiqueta del resultado que se espera en la salida. Mediante una determinada serie de iteraciones, la red va auto gestionando sus pesos para poder obtener reglas mediante las cuales sea capaz de clasificar las imágenes sin necesidad de etiqueta. Este auto aprendizaje es la base de las redes neuronales artificiales, y su resultado final es una serie de pesos que se mantienen en un continuo estado de evolución ampliando su capacidad de clasificación.

En este punto ya tenemos una red funcional que tiene en cada capa un conjunto de pesos de un tamaño estándar de 32 bits en coma flotante. Esta red será la que tomemos como medida para ver si reduciendo el tamaño de palabra obtendremos los beneficios esperados, o si por el contrario solo conseguiremos perder eficiencia.

2.3.1.- MNIST:

Como se ha citado anteriormente, MNIST es una base de datos que contiene un conjunto de muestras de números del 0 al 9 escritos a mano por diferentes personas de todo el mundo. La base de datos tiene aproximadamente 10000 muestras diferentes recopiladas por Yan LeCun, Chris Burgese y Corinna Cortes *[4]*.

Esta base de datos fue la utilizada por la mayoría de bancos para el entrenamiento de las redes neuronales convolucionales que se utilizaron para el reconocimiento de dígitos escritos a mano u ordenador en los cheques bancarios.

Figura 8: Ejemplo MNIST

Además, esta base de datos y el trabajo de Yan LeCun sirvieron de inspiración para el trabajo de diversos investigadores que implantaron esta idea con el reconocimiento de diversos caracteres escritos a mano con bases de datos aún más grandes gracias a los avances en las técnicas de almacenamiento de información y bases de datos.

2.4.- Cuantificación de los pesos:

Una vez ya tenemos los pesos calculados, se genera un fichero “solvestate”, en el cual nos encontramos el estado final de la red, con la definición de los coeficientes y demás recursos de la red. En él se encuentran almacenados entre otras cosas los pesos tras el número de iteraciones que hayamos asignado. Este fichero nos sirve tanto para reiniciar el entrenamiento de una red desde un punto preciso, o como es nuestro caso, para introducir la red y los pesos en Ristretto y así poder realizar su cuantificación.

En la ejecución de Ristretto, debemos definirle de nuevo una serie de propiedades como si la red se va a entrenar el CPU o GPU, el tipo de datos que posee la red y como parámetros importantes, el número de ciclos que debe realizar para entrenar la red y la precisión que puede perder la red. En nuestro caso, se ha establecido que a partir de los 100 ciclos ya no se obtenía ningún cambio en los resultados obtenidos de esta herramienta, y que como era requisito al inicio del proyecto, la precisión perdida debía ser como máximo del 1%.

De esta ejecución obtendremos la cantidad de bits asignados a cada tipo de peso y la precisión obtenida con cada tamaño de palabra.

Para poder extraer los pesos de la herramienta se ha debido modificar una de las funciones del entorno Ristretto para que nos cree un fichero de texto con los pesos a los que accede la red. En concreto se trata de la función "*conv_layer.cpp*" en la cual se recuperan los pesos del fichero "solvestate" y se realiza la cuantificación de los mismos en función del número de bits que se le asignan a cada tipo de coeficiente. Por ello, este es el único punto de la red en el que podemos obtener los pesos del solvestate en formato digital, lo que nos facilita sustancialmente el tratamiento posterior de estos pesos.

Una vez modificado el entorno procederemos a ejecutar un único ciclo de la red con la herramienta Ristretto para que se proceda al llamado de la función "conv_layer.cpp" y de esta forma obtener el fichero. Para que los pesos coincidan con los que realmente ejecutaría la red convolucional, se ha de utilizar el fichero solvestate que nos ha generado el entrenamiento de la red ya que como ha sido mencionado anteriormente, es en ese fichero donde se encuentra la información de los pesos de la red.

Una vez finalizada la ejecución de la herramienta, se dispondrá de un fichero txt con todos los pesos que utiliza la red de forma que podremos extraerlos con facilidad a la hora de ejecutar realmente la red. Cabe mencionar que para que estos pesos luego sean introducidos en la red real, se deberán cuantificar en función del tamaño de palabra.

2.5.- Eficiencia de la red en función del tamaño de palabra:

Para poder tener una medida de la eficiencia de la red, se ha utilizado al igual que para la cuantificación de los pesos, la herramienta Caffe-Ristretto que nos permite reducir el tamaño de palabra de ciertos parámetros en una CNN.

En la herramienta vamos a tratar los 3 parámetros que más memoria utilizan en la CNN que son los pesos de las capas convolucionales, los pesos de las neuronas de las capas Fully Conected y los valores de activación de las bias. Por ello realizaremos un tratamiento individual de cada uno de estos parámetros, y por último uno en conjunto para encontrar la combinación óptima.

Ver anexo 2: Reporte herramienta Ristretto

Comenzaremos con los pesos de las capas convolucionales, los cuales se han parametrizado con tamaños desde 32 bits hasta 2 bits:

Tamaño de palabra	Eficiencia de la red
32 bits	0.9906
16 bits	0.9906
8 bits	0.9905
4 bits	0.9904
2 bits	0.9659

Tabla 1.- Precisión de la red frente a tamaño de pesos de las capas convolucionales

En el caso de los pesos de las neuronas de la capa Fully Conected se ha probado con tamaños entre 32 bits y 2 bits:

Tamaño de palabra	Eficiencia de la red
32 bits	0.9906
16 bits	0.9906
8 bits	0.9906
4 bits	0.9899
2 bits	0.9346

Tabla 2.- Precisión de la red frente a tamaño de pesos de las neuronas de la capa Fully Conected

Para los valores de activación de las bias, se ha optado por realizar una parametrización entre 16 y 4 bits:

Tamaño de palabra	Eficiencia de la red
16 bits	0.9892
8 bits	0.9894
4 bits	0.982
2 bits	0.1009

Tabla 3.- Precisión de la red frente a tamaño de la activación de las bias.

Como cómputo global de la red, dado que la eficiencia es una estadística que refleja la posibilidad de que la red nos devuelva como resultado el valor correcto para los datos de entrada, se establece que la probabilidad conjunta debe ser menor que cada una de las probabilidades individuales de cada caso. Por ello, la combinación que mejores resultados a obtenido, es la conformada por 8 bits para los pesos de las capas convolucionales y Fully Conected, y 4 para los valores de activación de las bias, con una probabilidad de éxito del 98%.

3.- Traducción y pruebas de la red en VHDL:

Para llevar a cabo las pruebas, se ha definido la red en C utilizando la ayuda de la herramienta GUINNES la cual nos ha permitido obtener el esqueleto de la red neuronal convolucional para su posterior implementación.

Ver anexo 3: Estructura de la RNC en C.

Para su traducción al VHDL se ha utilizado la herramienta VIVADO HLS, la cual nos permite optimizar el código generado mediante una serie de directrices o "PRAGMAS". El más utilizado en nuestro caso es UNROLL, que se utiliza para desplegar los bucles. Este recurso de VIVADO HLS nos permite paralelizar recursos como las convoluciones de cada filtro o las operaciones de la capa fully conected.

Como se ha mencionado anteriormente, se va a proceder a probar 6 combinaciones diferentes de tamaños de palabra para los pesos de las dos capas convolucionales, los pesos de las neuronas en la capa Fully Conected y para la activación de la bias.

Estas combinaciones van a tener las siguientes dimensiones:

Número de ejecución	Pesos para convoluciones	Pesos para neuronas de Fully Conectd	Activación de las bias
1	32 bits	32 bits	16 bits
2	16 bits	16 bits	8 bits
3	8 bits	8 bits	4 bits
4	4 bits	4 bits	4 bits

Tabla 4.- Comparación de los tamaños de palabra para cada caso de prueba

Como se puede observar, el tamaño de memoria necesario es significativamente menor en la ejecución 4 que en la 1 pero su eficiencia como se ha demostrado anteriormente, es menor que el límite definido en las condiciones iniciales del 1%, por lo que no es una solución óptima para nuestro desarrollo y no será probada su eficiencia en términos de costes de ejecución.

3.1.- FPGA utilizada para las pruebas:

Para llevar a cabo las pruebas se ha optado por la utilización de la FPGA de Xilinx XCVU13P [5], la cual se nos presenta como una de las FPGAs más potentes en cuanto a recursos dado su tecnología Ultra Scale de 16 nm. Esta FPGA cuenta con un total de 5376 bloques de memoria BRAM de 18Kbytes, lo que nos supone 96.768Kbytes de memoria BRAM donde almacenaremos los pesos y demás constantes de nuestra red.

Otro aspecto importante para la selección de esta FPGA para realizar las pruebas, es el periodo de reloj alcanzable. Este parámetro es de 5 ns, por lo que la velocidad de cálculo es lo suficientemente elevada como para poder prestar un servicio adecuado a las necesidades técnicas. Como se podrá observar tras las pruebas, las diferentes ejecuciones se realizan en aproximadamente 3 ms, por lo que la experiencia a nivel de usuario sería óptima.

Además, la FPGA cuenta con suficientes bloques de memoria LUT para las operaciones y el suficiente número de DSPs como multiplicadores. Estos DSPs que se utilizarán como multiplicadores se nos presentan como multiplicadores de 25x18 bits.

3.2.- Caso de prueba 1:

Para este caso de prueba utilizaremos el siguiente conjunto de parametrizaciones para los datos:

Tipo de dato	Nº de bits
Pesos de las capas convolucionales	32
Pesos de las neuronas de la capa FullyConected	32
Indicadores de actvación de las bias	16

Tabla 5.- Tamaño de palabra para el caso de prueba 1

Con este conjunto de datos, sintetiza con herramienta Vivado HLS, la cual genera el siguiente conjunto de reportes:

Ver anexo 4.1: Reporte caso de prueba 1

Como podemos comprobar en el cuadro de tiempos y utilizaciones de recursos, se consumen 640823 ciclos de reloj entre que una imagen entra en la red y se obtienen los resultados. Como estamos trabajando a una frecuencia de reloj de 5 ns, este tiempo es equivalente a 3,2 ms.

Además, el número necesario de memorias BRAM_18K para el almacenaje de las constantes como los pesos, es de 515, lo que supone únicamente un 9% de la capacidad total de la FPGA que hemos seleccionado para hacer el estudio. Con respecto al número de DSPs utilizados como multiplicadores, hemos utilizado un total de 103 DSPs.

3.3.- Caso de prueba 2:

Para este caso de prueba utilizaremos el siguiente conjunto de parametrizaciones de datos:

Tipo de dato	Nº de bits
Pesos de las capas convolucionales	16
Pesos de las neuronas de la capa FullyConected	16
Indicadores de actvación de las bias	8

Tabla 6.- Tamaño de palabra para el caso de prueba 2

Con este conjunto de datos, sintetiza con herramienta Vivado HLS, la cual genera el siguiente conjunto de reportes:

Ver anexo 4.2: Reporte caso de prueba 2

Como podemos comprobar en el cuadro de tiempos y utilizaciones de recursos, se consumen 641316 ciclos de reloj entre que una imagen entra en la red y se obtienen los resultados. Como estamos trabajando a una frecuencia de reloj de 5 ns, este tiempo es equivalente a 3,2 ms.

Además, el número necesario de memorias BRAM_18K para el almacenaje de las constantes como los pesos, es de 282, lo que supone únicamente un 5% de la capacidad total de la FPGA que hemos seleccionado para hacer el estudio. Con respecto al número de DSPs utilizados como multiplicadores, hemos utilizado un total de 52 DSPs.

3.4.- Caso de prueba 3:

Para este caso de prueba utilizaremos el siguiente conjunto de parametrizaciones de datos:

Tipo de dato	Nº de bits
Pesos de las capas convolucionales	8
Pesos de las neuronas de la capa FullyConected	8
Indicadores de actvación de las bias	4

Tabla 7.- Tamaño de palabra para el caso de prueba 3

Con este conjunto de datos, sintetiza con herramienta Vivado HLS, la cual genera el siguiente conjunto de reportes:

Ver anexo 4.3: Reporte caso de prueba 3

Como podemos comprobar en el cuadro de tiempos y utilizaciones de recursos, se consumen 629299 ciclos de reloj entre que una imagen entra en la red y se obtienen los resultados. Como estamos trabajando a una frecuencia de reloj de 5 ns, este tiempo es equivalente a 3,15 ms.

Además, el número necesario de memorias BRAM_18K para el almacenaje de las constantes como los pesos, es de 179, lo que supone únicamente un 3% de la capacidad total de la FPGA que hemos seleccionado para hacer el estudio. Con respecto al número de DSPs utilizados como multiplicadores, hemos utilizado un total de 53 DSPs.

4.- Conclusiones:

Una vez analizados los diferentes resultados por separado, vamos a proceder a compararlos para observar la supuesta mejora del rendimiento de la red. Para poder comparar los resultados, nos vamos a centrar en los criterios que han sido descritos anteriormente y que son el tiempo de ejecución, la memoria BRAM necesaria, el número de DSPs utilizados como multiplicadores y la eficiencia o rendimiento de la red.

Comenzaremos analizando el tiempo de ejecución para cada caso de prueba. Para cuantificar el tiempo de ejecución de la red se utilizará el número de ciclos de reloj que le cuesta a cada imagen ser procesada por la red. Para calcular el tiempo real medido en milisegundos, se deberá multiplicar el periodo del reloj con el número de ciclos de ejecución. Como se puede apreciar en la *Tabla 8*, el tiempo de ejecución en cada una de las diferentes pruebas es muy similar.

Caso de Prueba	Ciclos de ejecución	Tiempo de ejecución
1	640823 ciclos de reloj	3,2 ms
2	641316 ciclos de reloj	3,2 ms
3	629299 ciclos de reloj	3,15 ms

Tabla 8.- Comparativa de tiempos de ejecución entre pruebas

Debemos destacar que las variaciones en el número de ciclos en los cuales se ejecuta cada prueba varían en función de la forma en la que la FPGA divide la memoria para almacenar los datos.
Con respecto al tamaño de memoria BRAM utilizada por cada ejecución, sucede lo previsto en la descripción del proyecto, dado que, a un menor tamaño de palabra para el mismo número de datos, el tamaño que se debe almacenar es mucho menor, como lo podemos comprobar en la *Tabla 9*.

Caso de prueba	Bloques de memoria BRAM_18K
1	515
2	282
3	179

Tabla 9.- Comparativa de memoria BRAM entre pruebas

En el apartado de DSPs como multiplicadores, hay que destacar que los resultados dependen en gran parte de la FPGA utilizada a diferencia de los otros elementos de esta comparativa, el número de DSPs depende de su tamaño. En la FPGA que se ha utilizado el tamaño del multiplicador es de 25x18 bits, que no nos permitiría multiplicar dos palabras de 32 bits con un solo DSP.
Una vez aclarado este punto, podemos apreciar lo descrito anteriormente en la *Tabla 10* donde se puede observar que el caso de prueba 1 utiliza el doble de DSPs que el resto de casos, puesto que tiene multiplicaciones de 32bits por 32bits.

Caso de prueba	Número de DSPs
1	102
2	52
3	53

Tabla 10.- Comparativa de número de DSPs en cada prueba

Como último elemento comparativo habíamos determinado que la eficiencia o rendimiento de la red no podía verse reducida en más de un 1%, motivo por el cual descartamos el caso de prueba 4 y todos aquellos con un tamaño de palabra menor, puesto que las pérdidas de rendimiento serían mayores. La diferencia de rendimiento es inferior al 1% con la red original que presentaba un rendimiento equivalente al 0.9899 para todos los tres casos de prueba, siendo el tercero el que ha sufrido un mayor descenso en el rendimiento, alcanzando una eficiencia de 0.982, un 0.7% inferior al rendimiento original.

Tras haber analizado los resultados de las diferentes pruebas realizadas, podemos deducir que la optimización de la red en el caso de prueba 3 es la que mejores prestaciones en cuanto a costes de ejecución y pérdida de rendimiento ofrece. Esto se debe a que hemos sido capaces de reducir los tamaños de palabra lo suficiente como para reducir costes sin llegar a sacrificar la eficiencia de la red.

Personalmente, analizando los resultados obtenidos, creía que al reducir el tamaño de palabra en un factor 4, como ha sido el caso de prueba 3, la velocidad de ejecución iba a ser más rápida en esta prueba mencionada. Tras estudiarlo nos podemos dar cuenta que al estar la FPGA ejecutando bloques de código en paralelo, la FPGA consume es más recursos, en nuestro caso multiplicadores DSP, pero no un mayor tiempo. También considero que el rendimiento alcanzado en el caso de prueba 3 con solo una pérdida del 0.7% es sorprendente, puesto que la cuantificación de los pesos se realiza con una precisión bastante menor que la que cabría esperar. El cuantificar con 32 bits, en lugar que con 8 bits nos hace ignorar los 24 bits menos significativos de información, los cuales ha quedado demostrado que solo aportan un 0.7% de la información que se nos presenta en la entrada.

Durante el tiempo en el cual se ha llevado a cabo este trabajo de fin de grado, Xilinx ha invertido esfuerzos en este mismo tema, y ha conseguido crear una herramienta que permite realizar los procesos que se han seguido en este proyecto. La herramienta consiste en optimizar los tamaños de palabra de una red neuronal para reducir sus costes y mantener su eficiencia y a diferencia de este trabajo, lo realiza todo desde un mismo entorno facilitando a los usuarios el no tener que estar realizando cambios de lenguajes de programación como si sucedía en este proyecto. Dicha herramienta se ha abierto al público recientemente. El hecho de que una empresa de la relevancia y tamaño como Xilinx esté trabajando en este tema, nos indica que la idea con la que se ha estado trabajando en este proyecto puede llegar a ser muy interesante.

5.- Bibliografía:

[1] Jia; Yangqing; Shelhamer; Evan; Donahue; Jeff; Karayev; Sergey; Long; Jonathan; Girshick; Ross; Guadarrama; Sergio; Darrell y Trevor (2014). *Caffe: Convolutional Architecture for Fast Feature Embedding*.

[2] Gysel; Philipp; Pimentel; Jon; Motamedi; Mohammad; Ghiasi y Soheil (2018). *Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks.* IEEE Transactions on Neural Networks and Learning Systems. Doi 10.1109/TNNLS.2018.2808319.

[3] Y. LeCun; L. Bottou; Y. Bengio y P. Haffner (1998). Gradient-Based Learning Applied to Document Recognition, *Proceedings of the IEEE, 86(11):2278-2324* Recuperado de: *http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf*

[4] Y. LeCun; C. Cortes y C. Burges (1998). *THE MNIST DATABASE of handwritten digits* Recuperado de: http://yann.lecun.com/exdb/mnist/

[5] Xilinx FPGA Ultra Scale + (Xilinx XCVU13P) https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf

[6] D.E.Rumelhart,J.L.MacClelland(eds.)(1986).Parallel Distributed Processing.Vol 1. Foundations, MITPress.

6.- Anexos:

6.1.- Anexo 1: Definición de la red - lenet-prototxt

A continuación, se muestra la definición de la red en la herramienta Caffe-Master:

```
name: "LeNet"
layer {
     name: "data"  type: "Input"   top: "data"  input_param { shape: { dim: 64
           dim: 1 dim: 28 dim: 28 } }
}
layer {
  name: "conv1"  type: "Convolution"  bottom: "data"  top: "conv1"  param {
       lr_mult: 1
  }
  param {
     lr_mult: 2
  }
  convolution_param {
     num_output: 20  kernel_size: 5  stride: 1  weight_filler {
          type: "xavier"
     }
     bias_filler {
          type: "constant"
     }
  }
}
layer {
  name: "pool1"   type: "Pooling"  bottom: "conv1"  top: "pool1"  pooling_param
{
          pool: MAX  kernel_size: 2  stride: 2
  }
}
layer {
  name: "conv2"   type: "Convolution"  bottom: "pool1"  top: "conv2"  param {
        lr_mult: 1
  }
  param {
     lr_mult: 2
  }
  convolution_param {
     num_output: 50  kernel_size: 5  stride: 1  weight_filler {
                type: "xavier"
     }
     bias_filler {
          type: "constant"
     }
  }
}
```

```
layer {
  name: "pool2"   type: "Pooling"  bottom: "conv2"  top: "pool2"  pooling_param
{
      pool: MAX  kernel_size: 2  stride: 2
  }
}
layer {
  name: "ip1"  type: "InnerProduct"  bottom: "pool2"  top: "ip1"  param {
        lr_mult: 1
  }
  param {
      lr_mult: 2
  }
  inner_product_param {
      num_output: 500 weight_filler {
            type: "xavier"
      }
      bias_filler {
            type: "constant"
      }
  }
}
layer {
  name: "relu1"   type: "ReLU"  bottom: "ip1"  top: "ip1"
}

layer {
  name: "ip2"   type: "InnerProduct"  bottom: "ip1"  top: "ip2"  param {
      lr_mult: 1
  }
  param {
      lr_mult: 2
  }
  inner_product_param {
      num_output: 10  weight_filler {
            type: "xavier"
      }
      bias_filler {
            type: "constant"
      }
  }
}
layer {
  name: "prob"  type: "Softmax"  bottom: "ip2"  top: "prob"
}
```

6.2.- Anexo 2: Reporte herramienta Ristretto

A continuación, se mostrará el reporte de la ejecución de la herramienta Ristretto mediante la cual se han obtenido tanto los pesos como la eficiencia de la red:

```
I0122 01:26:36.055649  5498 layer_factory.hpp:77] Creating layer mnist
I0122 01:26:36.062642  5498 quantization.cpp:106] Running for 100 iterations.
I0122 02:11:26.213404  5498 quantization.cpp:277] Network accuracy analysis for
I0122 02:11:26.213412  5498 quantization.cpp:278] Convolutional (CONV) and
fully
I0122 02:11:26.213416  5498 quantization.cpp:279] connected (FC) layers.
I0122 02:11:26.213423  5498 quantization.cpp:280] Baseline 32bit float: 0.9906
I0122 02:11:26.213431  5498 quantization.cpp:281] Dynamic fixed point CONV
I0122 02:11:26.213471  5498 quantization.cpp:282] weights:
I0122 02:11:26.213477  5498 quantization.cpp:284] 16bit:  0.9906
I0122 02:11:26.213490  5498 quantization.cpp:284] 8bit:  0.9905
I0122 02:11:26.213498  5498 quantization.cpp:284] 4bit:  0.9904
I0122 02:11:26.213510  5498 quantization.cpp:284] 2bit:  0.9659
I0122 02:11:26.213516  5498 quantization.cpp:287] Dynamic fixed point FC
I0122 02:11:26.213526  5498 quantization.cpp:288] weights:
I0122 02:11:26.213532  5498 quantization.cpp:290] 16bit:  0.9906
I0122 02:11:26.213543  5498 quantization.cpp:290] 8bit:  0.9906
I0122 02:11:26.213551  5498 quantization.cpp:290] 4bit:  0.9899
I0122 02:11:26.213562  5498 quantization.cpp:290] 2bit:  0.9346
I0122 02:11:26.213570  5498 quantization.cpp:292] Dynamic fixed point layer
I0122 02:11:26.213587  5498 quantization.cpp:293] activations:
I0122 02:11:26.213596  5498 quantization.cpp:295] 16bit:  0.9892
I0122 02:11:26.213618  5498 quantization.cpp:295] 8bit:  0.9894
I0122 02:11:26.213627  5498 quantization.cpp:295] 4bit:  0.982
I0122 02:11:26.213639  5498 quantization.cpp:295] 2bit:  0.1009
I0122 02:11:26.213646  5498 quantization.cpp:298] Dynamic fixed point net:
I0122 02:11:26.213655  5498 quantization.cpp:299] 8bit CONV weights,
I0122 02:11:26.213661  5498 quantization.cpp:300] 8bit FC weights,
I0122 02:11:26.213670  5498 quantization.cpp:301] 4bit layer activations:
I0122 02:11:26.213676  5498 quantization.cpp:302] Accuracy: 0.981
I0122 02:11:26.213687  5498 quantization.cpp:303] Please fine-tune.
```

6.3.- Anexo 3: Estructura de la RNC en C

```
void RED(const datos_t datos_in[1][X_size][Y_size],datos_t salida[10] )
{


datos_t conv1[20][24][24] ;

conv_2d<1,20,28,32>(datos_in,conv1);

#pragma HLS ARRAY_PARTITION variable=conv1 cyclic factor=5 dim=2
#pragma HLS ARRAY_PARTITION variable=conv1 cyclic factor=5 dim=3

datos_t conv2[20][12][12] ;

max_pooling_layer<20,24>(conv1,conv2) ;

datos_t conv3[50][8][8] ;

conv_2d<20,50,12,32>(conv2,conv3);
#pragma HLS ARRAY_PARTITION variable=conv2 cyclic factor=5 dim=2
#pragma HLS ARRAY_PARTITION variable=conv2 cyclic factor=5 dim=3

datos_t conv4[50][4][4] ;
max_pooling_layer<50,8>(conv3,conv4) ;


datos_t fc1[800] ;
datos_t fc2[500] ;

#pragma HLS ARRAY_PARTITION variable=fc1 cyclic factor=50 dim=1
#pragma HLS ARRAY_PARTITION variable=fc2 cyclic factor=5 dim=1

conver_array<50,4>(conv4,fc1);

fc_800x500<800,500,32>( fc1,fc2) ;

fc_layer2<500,10,32>( fc2,salida) ;
}
```

6.4.- Anexo 4: Reportes de HLS del análisis sintáctico y simulación.

6.4.1.- Anexo 4.1: Reporte caso de prueba 1

Synthesis Report for 'RED'

General Information

Date: *Mon Feb 11 21:16:18 2019*
Version: *2018.3 (Build 2405991 on Thu Dec 06 23:56:15 MST 2018)*
Project: *HLS8*
Solution: *solution6*
Product family: *virtexuplus*
Target device: *xcvu13p-fhga2104-1-i*

Performance Estimates

- ***Timing (ns)***
  - ***Summary***

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	4.275	0.63

- ***Latency (clock cycles)***
  - ***Summary***

Latency		Interval		Type
min	max	min	max	
640823	640823	320018	320018	dataflow

  - ***Detail***
    - ***Instance***

Instance	Module	Latency		Interval		Type
		min	max	min	max	
conv_2d_U0	conv_2d	320017	320017	320017	320017	none
fc_800x500_U0	fc_800x500	11001	11001	11001	11001	none
max_pooling_layer_1_U0	max_pooling_layer_1	83081	83081	83081	83081	none
conv_2d_1_U0	conv_2d_1	200701	200701	200701	200701	none
fc_layer2_U0	fc_layer2	5015	5015	5015	5015	none
Block_arrayctor_loop_U0	Block_arrayctor_loop	16501	16501	16501	16501	none
max_pooling_layer_U0	max_pooling_layer	4501	4501	4501	4501	none

Utilization Estimates

- ***Summary***

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	438	-
FIFO	-	-	-	-	-
Instance	455	103	11215	19052	-
Memory	60	-	1600	200	-
Multiplexer	-	-	-	945	-
Register	-	-	105	-	-
Total	515	103	12920	20635	0
Available	5376	12288	3456000	1728000	1280
Utilization (%)	9	~0	~0	1	0

- ***Detail***
  - ***Instance***

Instance	Module	BRAM_18K	DSP48E	FF	LUT
Block_arrayctor_loop_U0	Block_arrayctor_loop	0	1	312	387
conv_2d_U0	conv_2d	25	25	4463	10453
conv_2d_1_U0	conv_2d_1	25	25	1302	1574
fc_800x500_U0	fc_800x500	400	50	3158	3184
fc_layer2_U0	fc_layer2	5	2	654	620
max_pooling_layer_U0	max_pooling_layer	0	0	88	249
max_pooling_layer_1_U0	max_pooling_layer_1	0	0	1238	2585
Total	7	455	103	11215	19052

6.4.2.- Anexo 4.2: Reporte caso de prueba 2

Synthesis Report for 'RED'

General Information

Date: *Mon Feb 11 15:36:46 2019*
Version: *2018.3 (Build 2405991 on Thu Dec 06 23:56:15 MST 2018)*
Project: *HLS8*
Solution: *solution5*
Product family: *virtexuplus*
Target device: *xcvu13p-fhga2104-1-i*

Performance Estimates

- ***Timing (ns)***
  - ***Summary***

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	4.322	0.63

- ***Latency (clock cycles)***
  - ***Summary***

Latency		Interval		Type
min	max	min	max	
641316	641316	320013	320013	dataflow

  - ***Detail***
    - ***Instance***

Instance	Module	Latency		Interval		Type
		min	max	min	max	
conv_2d_U0	conv_2d	320012	320012	320012	320012	none
fc_800x500_U0	fc_800x500	11501	11501	11501	11501	none
max_pooling_layer_1_U0	max_pooling_layer_1	83081	83081	83081	83081	none
conv_2d_1_U0	conv_2d_1	200701	200701	200701	200701	none
fc_layer2_U0	fc_layer2	5013	5013	5013	5013	none
Block_arrayctor_loop_U0	Block_arrayctor_loop	16501	16501	16501	16501	none
max_pooling_layer_U0	max_pooling_layer	4501	4501	4501	4501	none

Utilization Estimates

- ***Summary***

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	438	-
FIFO	-	-	-	-	-
Instance	250	51	8684	20786	-
Memory	32	-	1156	390	-
Multiplexer	-	-	-	945	-
Register	-	-	105	-	-
Total	282	51	9945	22559	0
Available	5376	12288	3456000	1728000	1280
Utilization (%)	5	~0	~0	1	0

- ***Detail***
  - ***Instance***

Instance	Module	BRAM_18K	DSP48E	FF	LUT
Block_arrayctor_loop_U0	Block_arrayctor_loop	0	1	309	387
conv_2d_U0	conv_2d	25	1	3532	11421
conv_2d_1_U0	conv_2d_1	25	16	1046	1870
fc_800x500_U0	fc_800x500	200	32	2154	3902
fc_layer2_U0	fc_layer2	0	1	365	408
max_pooling_layer_U0	max_pooling_layer	0	0	88	247
max_pooling_layer_1_U0	max_pooling_layer_1	0	0	1190	2551
Total	7	250	51	8684	20786

6.4.3.- Anexo 4.3: Reporte caso de prueba 3

Synthesis Report for 'RED'

General Information

Date: Tue Feb 12 02:05:24 2019
Version: 2018.3 (Build 2405991 on Thu Dec 06 23:56:15 MST 2018)
Project: HLS8
Solution: solution7
Product family: virtexuplus
Target device: xcvu13p-fhga2104-1-i

Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	4.345	0.63

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
629299	629299	320014	320014	dataflow

  - **Detail**
    - **Instance**

Instance	Module	Latency		Interval		Type
		min	max	min	max	
conv_2d_U0	conv_2d	320013	320013	320013	320013	none
fc_800x500_U0	fc_800x500	11001	11001	11001	11001	none
max_pooling_layer_1_U0	max_pooling_layer_1	83081	83081	83081	83081	none
conv_2d_1_U0	conv_2d_1	189181	189181	189181	189181	none
fc_layer2_U0	fc_layer2	5015	5015	5015	5015	none
Block_arrayctor_loop_U0	Block_arrayctor_loop	16501	16501	16501	16501	none
max_pooling_layer_U0	max_pooling_layer	4501	4501	4501	4501	none

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	438	-
FIFO	-	-	-	-	-
Instance	152	52	7927	19971	-
Memory	27	-	630	264	-
Multiplexer	-	-	-	945	-
Register	-	-	105	-	-
Total	179	52	8662	21618	0
Available	5376	12288	3456000	1728000	1280
Utilization (%)	3	~0	~0	1	0

- **Detail**
  - **Instance**

Instance	Module	BRAM_18K	DSP48E	FF	LUT
Block_arrayctor_loop_U0	Block_arrayctor_loop	0	1	305	387
conv_2d_U0	conv_2d	25	1	3459	11383
conv_2d_1_U0	conv_2d_1	25	16	913	1547
fc_800x500_U0	fc_800x500	100	32	1389	3284
fc_layer2_U0	fc_layer2	2	2	607	588
max_pooling_layer_U0	max_pooling_layer	0	0	88	245
max_pooling_layer_1_U0	max_pooling_layer_1	0	0	1166	2537
Total	7	152	52	7927	19971