UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO

CENTRO UNIVERSITARIO UAEM VALLE DE MÉXICO

DEEP LEARNING PARA LA DETECCIÓN DE PEATONES Y VEHÍCULOS SOBRE FPGA

TESIS

Para obtener el Grado de

MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

P r e s e n t a

Ing. Rigoberto Vizcaya Cárdenas

Asesor

Dr. en C. José Martín Flores Albino

Co asesores:

Dr. en I. Saúl Lazcano Salas

Dr. Víctor Manuel Landassuri Moreno

Centro Universitario UAEM Valle de México

Atizapán de Zaragoza, Edo. de México 22 de enero de 2018

Dedicatoria

A mi familia por el tiempo que no he compartido con ustedes y en especial a mis hijos Alan Iker y Alison Daniela, son los motores que mueven mi vida.

Agredecimientos

Agradezco a la UAEM Valle de México por permitirme culminar esta etapa de mi vida y a su planta docente por los comentarios para la aportación de este trabajo.

Al Dr. José Martín Flores Albino por su apoyo brindado en este proyecto, así como a mis co asesores Dr. Saúl Lazcano Salas y Dr. Víctor Manuel Landassuri Moreno por sus acertados comentarios.

Al CONACyT por los recursos económicos proporcionados, ya que sin su ayuda esto no hubiera sido posible.

Resumen

Dado el reciente desarrollo y el impacto que ha tenido el paradigma de *Deep Learning* en el campo de la Inteligencia Artificial, el presente trabajo tiene como base el interés en este paradigma de aprendizaje, en específico empleando redes neuronales convolucionales *(CNNs)*, para la detección o clasificación de objetos en imágenes; además se analiza las ventajas de implementar estos algoritmos en hardware.

El objeto de estudio del aprendizaje automático es tratar de emular la inteligencia humana de forma artificial. Se ha trabajado en este campo por años, con diferentes enfoques y algoritmos. En la última década, el paradigma del Deep Learning ha revolucionado el estado del arte en tareas como reconocimiento de voz, visión artificial y el procesamiento del lenguaje natural; que resultaban difíciles de llevar a cabo por una máquina. Las técnicas que predominan en este paradigma son las *CNNs*, se utilizan como principal algoritmo en tareas que involucran visión artificial, tales como la detección de objetos. Se ha logrado un despunte importante en el reconocimiento de patrones en imágenes y video empleando estas técnicas, al grado de superar la capacidad humana. Un factor importante para ese desarrollo es la capacidad de procesar altos volúmenes de información en aplicaciones exitosas, lo que ha derivado en que los dispositivos empleados para dicho propósito, como GPUs y CPUs multinúcleo requieran de gran cantidad de energía para su funcionamiento.

Recientemente, han surgido investigaciones enfocadas en buscar alternativas de hardware, sobre el cual implementar las CNNs de forma eficiente, sobre todo para aplicaciones embebidas. Una de estas alternativas son los *Arreglos de Compuertas Programables en Campo* (*FPGAs),* que ofrecen la capacidad de procesamiento en paralelo espacial y temporal, un menor tiempo de latencia y bajo consumo de potencia; lo que resulta ideal para ese tipo de aplicaciones.

El presente trabajo se divide en dos partes, por un lado se hace la implementación del paradigma Deep Learning con una CNN para clasificar imágenes de señales de tránsito vehicular (como primer caso de estudio), con el propósito de medir el tiempo de entrenamiento y su desempeño en la clasificación. Por otro lado, se investiga la tecnología relacionada con FPGAs, para determinar la forma en que se puede acelerar el cómputo implicado en ese tipo de redes con estos dispositivos, validándolos como una alternativa de implementación para *sistemas embebidos*.

Los resultados obtenidos en la presente investigación son: 1) La programación, entrenamiento y prueba de una CNN. Se realizaron una serie de experimentos, encontrando un error en la clasificación de 3.25% y un tiempo de entrenamiento de 0.33 horas, para los mejores casos de los ensayos realizados. 2) Se analizan las ventajas de implementar este tipo de algoritmos en FPGAs, sus restricciones, requisitos y tres alternativas de desarrollo.

Cabe señalar que resultados de esta investigación fueron publicados en una revista indizada.

Abstract

Given the recent development and impact of the Deep Learning paradigm in the field of Artificial Intelligence, this work is based on the interest in this learning paradigm, specifically using convolutional neural networks (CNNs), for detection or classification of objects in images; besides analyzing the advantages of implementing these algorithms in hardware.

Machine Learning has focused on trying to emulate human artificial intelligence. Researchers have worked in this field for years, with different approaches and algorithms. In the last decade, the Deep Learning paradigm has updated the state of the art in tasks such as voice recognition, artificial vision and natural language processing, which were difficult to carry out artificially. The algorithms that predominate in this paradigm are CNNs, we find them in any task that involves artificial vision, such as object detection. These techniques have achieved a crucial development in the area of pattern recognition in images or video, even beating the human capacity in some tasks. An important factor for this development is the ability to process high volumes of information in successful applications, which has meant that the devices used for this purpose, such as GPUs and multicore CPUs, require a large amount of energy for their operation.

Recently, research has emerged focused on finding other hardware alternatives, on which to implement CNN, especially for embedded applications. One of these options are *Field Programmable Gate Arrays (FPGAs)*, which offer spatial and temporal parallel processing capacity, a shorter latency time and lower power consumption; what is ideal for such applications.

This work is divided into two parts, first the implementation of the Deep Learning paradigm with a CNN, to classify images of vehicular traffic signals (as the first case of study), this with the purpose of measuring the training time and its performance in terms of accuracy for classification. On the other hand, the technology related to FPGAs is investigated, to determine the way in which the

computation implied in this type of networks can be accelerated with these devices, validating them as an implementation alternative for embedded systems.

The results obtained in this investigation are: 1) The programming, training and testing of a CNN. A series of experiments was carried out, finding an error in the classification of 3.25% and a training time of 0.33 hours, for the best cases of the tests carried out. 2) The advantages of implementing this type of algorithms in FPGAs, their restrictions, requirements and three development alternatives are analyzed.

It should be noted that the results of this research were published in an indexed journal.

Índice de contenido

Índice de figuras

Índice de tablas

Lista de acrónimos

AHDL	*Altera Hardware Description Language*
AI	*Aritificial Intelligence*
ALU	*Arithmetic Logic Unit*
API	*Application Programming Interface*
ASIC	*Aplication-Specific Integrated Circuit*
BRAM	*Block Random Access Memory*
CLB	*Configurable Logic Block*
CNN	*Convolutional Neural Networks*
CPLD	*Cmplex Programmable Logic Device*
CPU	*Central Processor Unit*
CUDA	*Compute Unified Device Architecture*
DDR	*Double Data Rate*
DL	*Deep Learning*
DMA	*Direct memory access*
DNN	*Deep Neural Network*
DRAM	*Dynamic Random Access Memory*
DSP	*Digital Signal Processing*
EEPROM	*Electrically Eresable Programmable Read-Only Memory*
EPROM	*Eresable Programmable Read Only Memory*
FFT	*Fast Fourier Transform*
FIFO	*First in, first out*
FLOPS	*Floating point operations per second*
FPGA	*Field Programmable Gate Array*
GAL	*Generic Array Logic*
GPU	*Graphics Processing Unit*
HDL	*Hardware Description Language*
HLS	*High Level Synthesis*

IEEE	*Institute of Electrical and Electronics Engineers*
ISE	*Integrated Synthesis Environment*
LUT	*Look-Up Table*
MLP	*Multi Layer Perceotron*
MNIST	Modified National Institute of Standards and Technology
NCD	*Native Circuit Description*
NGC	*Native Generic Circuit*
NGD	*Native Generic Database*
OpenCL	*Open Computing Language*
PAL	*Programmable Array Logic*
PLD	*Programmable Logic Divice*
RAM	*Random Acces Memory*
ReLU	*Rectified Linear Unit*
RTL	*Register Transfer Level*
SDR	*Softare Defined Radio*
SDRAM	*Double Data Rate Synchronous Dynamic Random-Access Memory*
SRAM	*Static Random Access Memory*
UCF	*User Constraints File*
UCF	*User Constraints File*
VHDL	*Very High-Level Description Language*

Capítulo 1 Introducción

1.1 Antecedentes

Algunas tareas que son naturales para el ser humano como la visión, el reconocimiento de la voz o el procesamiento del lenguaje natural resultan complicadas de llevar a cabo de manera artificial. Esta es un área que ha estado en constante desarrollo dentro del campo de la Inteligencia Artificial (AI) por décadas. Sin embargo, a partir del 2006 se ha logrado un despunte importante en el *estado del arte* de esas y otras tareas, gracias al paradigma del Deep Learning (DL), por respetar su nombre en inglés, o "Aprendizaje Profundo", ver (Conneau, Schwenk, & Le Cun, s.f.), (LeCun, Bengio, & Hinton, 2015), (Redmon & Farhadi, s.f.).

En la parte de visión, y en concreto respecto al reconocimiento de objetos en imágenes, las redes neuronales convolucionales (CNN) dominan el estado del arte desde 2012, prueba de ello son los resultados obtenidos en diferentes competencias a nivel mundial enfocadas en ese tipo de tareas, parte de esos resultados se publican en (Benenson, 2017). Si bien son variadas las tareas en las se ha tenido éxito con estas técnicas, lo que tienen en común, es que requieren de una CNN como parte fundamental. Por ejemplo, para la de detección de objetos (como peatones y vehículos) en imágenes o video, se requiere de dos secciones: la CNN con la que se hace la clasificación de objetos de interés, y la sección en la que se asignan las coordenadas de la ubicación de dichos objetos en la imagen.

Aunque este paradigma (incluidas las CNN) no es algo nuevo, "*ya que existe desde la década de los 90's*", los resultados actuales superan en gran medida a los de épocas anteriores, incluso se ha superado la capacidad del ser humano en algunas tareas, como se muestra en (Graham, s.f.) y (Springenberg, Dosovitskiy, Brox, & Riedmiller, s.f.). Una de las razones de ello, es que ahora se cuenta con potentes equipos de cómputo, capaces de llevar a cabo el procesamiento requerido por los algoritmos de este paradigma. Otro factor que ha originado un notable avance (en comparación con otras técnicas) es la creciente investigación en este

campo, con lo que se ha generado flexibilidad y diversidad en esos algoritmos. Otro ingrediente importante del que se ha beneficiado este enfoque, es la enorme cantidad de datos actualmente disponibles para procesar.

Una de las razones por las que las CNN resultan idóneas para aplicaciones que involucran imágenes o tramas de video, es que con esta técnica se procesa cada imagen por secciones, en lugar de tener un parámetro por cada pixel de la imagen, (como sería el caso de una red multi-layer perceptrón, MLP). Se crean grupos filtros de tamaño reducido, (en comparación con el tamaño de la imagen). Estos filtros se desplazan a lo largo de la imagen, haciendo operaciones de productos y sumas, que es la operación de convolución. El propósito de los filtros es generar mapas de características, a partir de las cuales se pueden crear diversas aplicaciones. Aun así, la cantidad de operaciones a realizar es muy grande, por ejemplo, en (Simonyan & Zisserman, 2015) se realizan 15470×10^6 operaciones multiplicación-acumulación (MAC) por imagen que entre a la red, lo que implica que se debe contar con un potente equipo de cómputo para implementar ese tipo de aplicaciones.

En una CNN, la mayor parte de los recursos computacionales, y en consecuencia la energía, son usados en las operaciones de convolución (más del 95%), por lo que es deseable tener formas eficientes de llevarlas a cabo, sin afectar el rendimiento de la red. Parte de la investigación actual se enfoca en buscar alternativas para implementar los algoritmos de Deep Learning en hardware, más que en software, esto encaminado a obtener una mayor eficiencia en cuanto al consumo de energía, menor tiempo de latencia y un menor tamaño espacial. Hablando en particular de las CNNs, las alternativas han sido la implementación sobre *Field Programmable Gate Arrays* (FPGA), debido a la capacidad que ofrecen estos dispositivos para el procesamiento en paralelo, tanto espacial como temporal; además de la flexibilidad de reconfigurar el comportamiento que tendrá el hardware para alguna aplicación específica. Otra opción ha sido la implementación sobre *Aplication-Specific Integrated Circuits* (ASIC), que al ser dispositivos hechos a la medida de cada aplicación ofrecen el menor tamaño espacial, menor consumo de

potencia y menor tiempo de latencia, Lo que tienen en común estas alternativas es que el desarrollador requiere de experiencia en el diseño de hardware, además de un profundo conocimiento sobre la aplicación a la que está dirigido dicho diseño, lo que no es algo trivial.

En este trabajo se investigan los principios de las redes neuronales convolucionales, su contexto y su impacto en el estado del arte. Como primer caso de estudio se entrena y prueba una CNN en la tarea de clasificación de señales de tránsito vehicular usadas en México y otros países. Por otro lado, se investiga la tecnología de FPGA para conocer su arquitectura, funcionamiento y programación. Con esto se tendrán las herramientas necesarias para plantear la forma en que se pueden implementar las CNNs en FPGAs. Esto va enfocado a propiciar el aceleramiento en el manejo de información requerido en el paradigma del Deep Learning, cuidando el consumo de potencia, tiempo de latencia y las dimensiones espaciales del hardware empleado, que es lo que se busca en las aplicaciones embebidas.

1.2 Planteamiento del problema

Las redes neuronales convolucionales, en los últimos años, han propiciado un importante avance en tareas que involucran visión artificial, tales como clasificación, localización, detección y segmentación de objetos, descripción de escenas, entre otras, ya sea en imágenes o video. Los resultados que se obtienen actualmente se puedan emplear en una gran variedad de aplicaciones, en las que ese tipo de tareas estarían prácticamente resueltas.

Un ejemplo de esas aplicaciones son los sistemas para el monitoreo y control de tránsito vehicular y video-vigilancia en general, así como la captura de evidencias de otros sucesos que ocurren tanto en avenidas principales como en lugares concurridos. Así, el uso de cámaras para la captura de video se ha incrementado enormemente, ya que permite la visualización de diferentes eventos para su posterior análisis o bien para el registro de evidencias de estos hechos. Por lo

general, es un operador el encargado de registrar los eventos que ocurren de forma manual, esto por la capacidad natural del ser humano para la detección visual en imágenes y video, claro está que a la larga esto provoca que el cansancio le ocasione no detectar todo lo que ocurre en la escena. Un sistema automático que lo pudiera auxiliar en esa tarea de manera ininterrumpida, mejoraría la eficiencia de la detección de eventos de interés, sobre todo si se pudiera implementar en un sistema embebido, esto es, incorporado al interior de la misma cámara.

Para el ejemplo anterior, sería ideal que esos datos ya procesados, en lugar de llegar al operador, sean tomados por otro sistema automático, que pueda detectar cuando esos eventos (tráfico, asaltos, robos, secuestros, etc.) se están llevando a cabo y que de manera inmediata se tome medidas al respecto por el propio sistema, como la coordinación de semáforos de forma automática, seguimiento de vehículos o peatones a través de la ciudad, según el caso. Claro que, para generar un sistema completo con ese grado de inteligencia, primero se requiere tener la sección que proporcione los datos procesados de la detección y clasificación de objetos en la escena, esto de manera adecuada en cuanto a la eficiencia para la detección y tiempo de procesamiento.

Las CNNs han mostrado ser adecuadas para ese tipo de problemas, el inconveniente que se origina al emplearlas (y el paradigma de Deep Learning en general), es que se requiere de potentes equipos de cómputo para llevar a cabo la gran cantidad de operaciones multiplicación-acumulación necesarias en sus algoritmos. Esto fue una limitante para investigadores de este campo en los 90’s; para solventar esa dificultad se ha optado por la implementación de los algoritmos teniendo en mente el hardware, más que el software. Con la evolución que se ha tenido en dispositivos como: Graphics Processing Units (GPUs), CPUs multi-núcleo y el uso de clústeres para el procesamiento de alto desempeño, se han podido implementar y probar estos algoritmos con éxito. Otra opción es diseñar un ASIC, en (Chen, y otros, 2014) reportan que alcanzan 452 GFLOPS y una reducción de energía de 21.08 veces en comparación con un procesador convencional. En trabajos como (Krizhevsky, Sutskever, & E. Hinton, 2012), (Potluri, Fasih, Kishore

Vutukuru, Al MachoT, & Kyamakya, 2011), (Redmon, Divvalay, Girshick, & Farhadi, s.f.), (Tomé, y otros, s.f.), se muestran resultados en diversas aplicaciones con el paradigma de Deep Learning, implementando los modelos en GPUs, de forma habitual. Esto implica dos factores inapropiados para ciertas aplicaciones; que son: un alto consumo de potencia y el tamaño dimensional del hardware.

Otras aplicaciones en las que por su naturaleza se proyectan importantes desarrollos, son las que tienen que ver con drones, pequeños robots, smartphones, sistemas embebidos para automóviles autónomos, y otras más, en las que se cuenta con una fuente de energía finita.

1.3 Objetivos

1.3.1 Objetivo general

Implementar el paradigma de Deep Learning mediante una CNN para clasificación de señales de tránsito vehicular y analizar su implementación en un FPGA.

1.3.2 Objetivos específicos

1. Programar la arquitectura de una red neuronal convolucional para la clasificación de señales de tránsito vehicular y evaluar su desempeño en cuanto al tiempo de entrenamiento y error en la clasificación.
2. Determinar las ventajas de implementar redes neuronales convolucionales sobre FPGAs.

Se compara el desempeño de la CNN contra lo reportado en el estado del arte en este tipo de aplicaciones y se analiza la implementación de la CNN en el FPGA Artix 7 de Xilinx®.

El interés esencial es acelerar el cómputo implicado en las CNNs para la aplicación en las que se utilicen. Como primer caso de estudio se realiza la clasificación de señales de tránsito vehicular tomadas de un banco de imágenes

público para el entrenamiento y prueba de la red. Esto como un paso previo a la detección de peatones u otros objetos; que tendría un proceso similar.

1.4 Hipótesis

El paradigma de Deep Learning permite procesar imágenes mediante CNNs para la detección de objetos con un alto grado de exactitud. Su implementación en un sistema embebido permitiría aprovechar su capacidad para resolver problemas relacionados con lugares públicos de grandes y medianas ciudades; como el tráfico vehicular. Por lo tanto, se requiere evaluar la capacidad de este paradigma y su implementación en hardware, con el objetivo de aplicarlo en la solución de problemas reales.

Teniendo en cuenta las ventajas del paralelismo espacial y temporal que ofrecen los FPGAs, la realización de este trabajo de investigación permitirá visualizar la implementación de CNNs sobre hardware reconfigurable para la clasificación de señales de tránsito vehicular, como primer caso de estudio.

Se tiene la siguiente pregunta de investigación:

¿Es factible y viable la implementación de CNNs aplicadas a la clasificación y detección de objetos de interés sobre FPGAs, proporcionando características de desempeño en cuanto al consumo de potencia, tamaño y tiempo de latencia adecuadas para el funcionamiento en un sistema embebido?

1.5 Justificación

Gracias a los avances que se ha tenido en los últimos años en los algoritmos de CNNs en tareas como la detección y clasificación de objetos en imágenes, por otro lado la creación de dispositivos adecuados para el cálculo intensivo que el paradigma requiere, además de la enorme cantidad de datos que en la actualidad se tiene disponible; se ha revolucionado en los resultados que se están obteniendo, incluso en otras áreas de la Inteligencia artificial.

Una tarea de particular interés, es la clasificación de objetos (señales de tránsito vehicular), previo a la detección en imágenes, esto en un sistema de tamaño reducido. Los FPGA, permiten la implementación de cualquier sistema digital, debido a que el comportamiento del hardware es reconfigurable por el usuario y gracias al paralelismo que maneja, integración de módulos de procesamiento digital de señales (DSP), tamaño y consumo de energía, es posible usarlos para llevar a cabo modelos computacionales que demandan gran cantidad de recursos en un sistema embebido.

Un reto a superar ha sido el manejar la cantidad de operaciones que se requiere ejecutar, en un tiempo adecuado para la aplicación. Esto implica una alta demanda de recursos computacionales. En (Fowers, Brown, Cooke, & Stitt, 2012) se hace un estudio comparativo entre un CPU serial, un CPU multi-núcleo, un GPU y un FPGA para evaluar su desempeño en aplicaciones de video que utilizan técnicas de ventana deslizante. Hacen pruebas con tramas de diferente resolución; reportan que la cantidad de tramas por segundo procesadas cambia para cada dispositivo, a excepción del FPGA, que presenta un comportamiento constante para cada caso. El dispositivo con el que obtienen una mayor cantidad de imágenes procesadas por segundo son los GPU con algoritmos de transformada rápida de Fourier (FFT), seguidos por los FPGA. En cuanto al consumo de potencia, los FPGA superan a los demás dispositivos.

La ventaja de usar un CPU en este tipo de tareas es la facilidad de programación, que incide en el tiempo de desarrollo. Las desventajas de este son su naturaleza serial, y su consumo de potencia dinámica, debido a las velocidades de reloj que se manejan. Un CPU comercial tiene ciclos de reloj de 3.6 GHz o más, mientras que en un FPGA se manejan unos 500 MHz en los mejores casos. La naturaleza serial y sobre todo el acceso a memoria, provocan un cuello de botella para aplicaciones que requieren de procesamiento en paralelo, como el caso de imágenes o video; lo que implica que el tiempo de procesamiento no sea el adecuado para aplicaciones de ese tipo.

Si se quiere llevar a cabo alguna aplicación de reconocimiento o clasificación de objetos sobre imagen o video en tiempo real, se necesita no sólo seleccionar el modelo o técnica más apropiada para cada tarea, sino también seleccionar el hardware adecuado. En trabajos como (Cope, Cheung, & Luk, 2007), (Pauwels, Tomasi, Díaz, Ros, & Van Hulle, 2012) *y* (Ratheesh Kalarot, John Morris, 2010) también han realizado estudios comparativos para evaluar el desempeño de GPUs y FPGAs en tareas de ese tipo. Coinciden en que un GPU supera al FPGA en la ejecución de operaciones de punto flotante, en la mayor cantidad de imágenes procesadas por unidad de tiempo, el tiempo de desarrollo para una aplicación y facilidad de programación. Por su parte un FPGA supera al GPU en el tiempo de latencia, flexibilidad, consumo de potencia y en sus dimensiones.

En ese sentido, un Circuito Integrado de Aplicación Específica (ASIC) sería el dispositivo más adecuado, debido a que se trata de un dispositivo hecho a la medida. La desventaja de este, es la falta de flexibilidad (ya que se diseña para una determinada aplicación), tiempo de desarrollo y sobre todo, el costo de implementación para producción en pequeños volúmenes.

En otros trabajos, y por sus características, los FPGA se han usado como aceleradores de CNNs, como en (Farabet, y otros, 2010), (Ovtcharov, y otros, 2015) (Rahman, Lee, & Choi, 2016), (Zhang, y otros, 2015) y algunos más, esto debido a que la arquitectura de estos dispositivos es ideal para ese tipo de redes artificiales.

En este proyecto se implementa una CNN en software, con el propósito de entrenarla y probar su eficiencia para la tarea de clasificación, así como medir el tiempo de entrenamiento. Por otro lado, se estudia la forma de implementar ese tipo de modelos en un FPGA, dadas las ventajas de estos dispositivos. El propósito es que este trabajo sirva como base para la creación de sistemas inteligentes, aplicados a la solución de diversas problemáticas que se originan en lugares concurridos.

La implementación de estas técnicas en FPGAs en lugar de otro dispositivo se justifica por su tamaño, tiempo de latencia y bajo consumo de potencia, lo que los

hace adecuados para sistemas embebidos. Además, por sus características, permiten procesar varias imágenes por segundo. Por el contrario, la implementación a través de sistemas basados en procesador convencional, atrasaría la generación de resultados en el procesamiento de las imágenes, debido a su naturaleza serial. En el caso de dispositivos de procesamiento gráfico en paralelo de varios núcleos dedicados (GPU), es el consumo de energía la característica que los excluye de su implementación en sistemas embebidos con restricciones de energía. El uso de un ASIC no se justifica debido al tiempo de desarrollo que se requiere, la flexibilidad que no ofrece y el costo de implementación.

1.6 Fundamentación inicial

El reconocimiento de objetos en imágenes y/o video se puede dividir en dos categorías en el desarrollo que se ha tenido, que son, a lo que algunos autores se refieren como *modelos tradicionales* para el reconocimiento de patrones y los modelos basados en el *aprendizaje automático* o *"machine learning"*. En este último ámbito, desde la década de los 90's se comenzó a dar buenos resultados en tres áreas principales: minería de datos (en predicciones médicas), reconocimiento de rostros y del habla que (son eran tareas difíciles de llevar a cabo) y aplicaciones de software personalizadas (Mitchell, 1997).

En este campo de la Inteligencia Artificial se ha tenido un alto desarrollo en años recientes, comparado con décadas anteriores. En este sentido, hablando de los *modelos tradicionales* o *heurísticos,* como *"Bag of Words (BoV)",* también conocido como *"Bag of Visual Words"* y "*Fisher Vector (FV)"* para la detección y clasificación de objetos en imágenes o video, se han hecho trabajos como (Bristow & Lucey, s.f.), (Dalal & Triggs, 2005), (M. Bertozzi, 2007), (Xuejie Nian, 2015) entre muchos más, en los que los objetos de interés han sido peatones. En este tipo de modelos y de manera general para la detección y clasificación de objetos, hay dos componentes fundamentales para su implementación, que son:

1) El extractor de características (o descriptor).

2) El clasificador.

Para este fin, los principales algoritmos empleados en la penúltima década para llevar a cabo la tarea del descriptor han sido principalmente*: HOG (Histograms of Oriented Gradients), SIFT (Scale Invariant Feature Transform) y SURF (Speed-up Robust Features)*. Como clasificadores se han usado ampliamente: *SVM (Support Vector Machines) y AdaBoost (Adaptive Boosting),* entre otros. También se han usado variantes o combinaciones de los mismos para obtener mejores resultados en el desempeño.

En general, esos algoritmos han mostrado buen comportamiento para los objetos de interés que se pensaron al ser diseñados. La principal desventaja que han tenido, es que al ser *diseñados a mano o usando heurística,* resultan adecuados para las aplicaciones que se tuvieron en mente en principio, pero no para otras. Lo anterior ocurre al cambiar el objeto que se desea detectar, lo que implica hacer cambios en el diseño del algoritmo, de forma que éste sea optimizado, *"The No Free Lunch Theorem"* (Wolpert & Macready, 1997). Además, se debe tener en cuenta la variación de las características de los objetos en transformaciones debidas a la traslación, rotación, escala, iluminación, oclusión, entre otras.

Por su parte, el aprendizaje automático y particularmente a lo que se conoce como Deep Learning, en los últimos años ha tenido un importante avance entre la comunidad de investigadores, debido a los resultados que se están logrando en comparación con los métodos tradicionales. En (LeCun Y. , 2012) obtuvieron resultados que superaban a los métodos tradicionales en la robustez y exactitud para la extracción de características, así como la clasificación por medio de una CNN, aplicado a la detección de cáncer mamario. En este paradigma ahora tanto el extractor como el clasificador son entrenables y se pueden ir ajustando de manera automática por el propio sistema.

En (Chatfield, Simonyan, Vedaldi, & Zisserman, s.f.) hacen una comparación entre métodos tradicionales y el de Deep Learning con CNNs en aspectos como la robustez en la extracción de características y la efectividad en la clasificación de

imágenes. Concluyen que a pesar del alto costo computacional que se requiere para las CNNs, estas superan a los métodos tradicionales en cuanto a la exactitud. En otros proyectos se han enfocado en competir en el *"ImageNet Large Scale Visual Recognition Challenge"* (Russakovsky, y otros, 2017), que desde el 2010 se ha llevado a cabo anualmente. Este reto consiste en clasificar 1.2 millones de imágenes, entre 1000 clases diferentes. Este concurso es un referente a nivel mundial. Los últimos ganadores han sido: Clarifai (2013), Google (2014), el centro de investigación de Microsoft (2015), y el CUImage en (2016). En estos trabajos han usado CNNs para la tarea de detección y clasificación. En este paradigma, más que diseñar un algoritmo para cada tarea, lo que se hace es implementar una técnica para que una máquina aprenda a extraer las características de interés, de forma que por sí misma haga la clasificación de objetos.

La característica principal de este tipo de red neuronal es que tiene una capa de entrada y una de salida, pero puede tener varias capas intermedias, de ahí su nombre de "profunda", y como su nombre lo indica; principalmente llevan a cabo operaciones de convolución entre una capa de entrada y un filtro. El resultado de esa operación es un mapa de características. Ese proceso se repite con varias capas de la CNN. El objetivo de las capas intermedias, es extraer características de los objetos (de los mapas de características) con diferentes niveles de abstracción. El propósito de la capa de salida es hacer la clasificación (si esa fuera la tarea). De forma que se tiene una aplicación end to end, es decir que el extractor de características y el clasificador son implementados por la propia red. Debido a la gran cantidad de operaciones que se llevan a cabo para obtener buenos resultados, se requiere que estas se realicen en paralelo y sea viable su implementación. De ahí la conveniencia de hacerlo en hardware.

1.7 Alcances y limitaciones

En este proyecto se estudia la implementación de CNNs en hardware. El hardware seleccionado para este propósito es un FPGA debido a sus ventajas sobre otros dispositivos. La investigación contempla dos puntos importantes como primer

caso de estudio: la implementación de una CNN para clasificar imágenes de tránsito vehicular y, analizar su implementación en un FPGA.

1.7.1 Alcances

1. La red podrá clasificar señales de tránsito vehicular comunes en México y otros países, como señales preventivas, restrictivas, de servicios y turísticas. El modelo obtenido se puede adecuar a clasificar peatones y vehículos, como una tarea previa a la detección de los mismos

2. En primera instancia la red contará con dos capas de convolución y dos de agrupamiento, además de la capa completamente conectada. La arquitecura obtenida se podrá mejorar al agregar más capas ocultas.

3. El entrenamiento y prueba de la red se hará en un sistema basado en CPU, posteriormente se hace el análisis de su implementación en el FPGA, con los parámetros establecidos.

4. El banco de imágenes es tomado de la página de la Secretaría de Comunicaciones y Transportes de México.

1.7.2 Limitaciones

1. No se pretende probar con diferentes arquitecturas de CNNs, esto con la finalidad de tener congruencia en los experimentos realizados.

2. La sección del sistema para la detección quedará pendiente como un segundo caso de estudio.

3. En primera instancia no se pretende probar con diferentes cantidades de capas de convolución, esto para tener un punto de comparación con una CNN de las reportadas en el estado del arte.

4. Ya que la estructura de cada dispositivo cambia según la familia y el fabricante, solo se estudiará la implementación en FPGA Artix 7 de Xilinx®.

1.8 Metodología

Para cumplir con los objetivos planteados, la presente investigación se dividió en las siguientes etapas:

1) Revisión de la teoría y conceptos relacionados al Deep Learning, así como el impacto que han tenido las CNNs en el estado del arte en cuanto a la detección y clasificación de objetos en imágenes se refiere.

2) Revisión de la tecnología afín a los FPGAs, en específico su aplicación a la implementación de algoritmos con naturaleza paralela.

3) Establecer la arquitectura de una CNN. Entrenar y probar dicha arquitectura en un sistema basado en CPU, mediante la realización de diversos experimentos, enfocados a medir el tiempo de entrenamiento y error en la clasificación.

4) Analizar la implementación de la CNN con sus parámetros previamente establecidos sobre hardware reconfigurable.

5) Finalmente, analizar los resultados y conclusiones obtenidas.

1.9 Publicaciones derivadas de la investigación

Parte de los resultados obtenidos se encuentran publicados en la revista "*Journal CIM Vol. 5, Núm. 2, Coloquio de Investigación Multidisciplinaria 2017"*, ISSN: 2007-8102, con el artículo: "*Desempeño de una Red Neuronal Convolucional para Clasificación de Señales de Tránsito Vehicular*". Otros artículos derivados de esta investigación son: "Deep Learning y sus aplicaciones" y "Deep Learning para la detección de peatones y vehículos sobre FPGA's", presentados en coloquios de UAEM Valle de México.

1.10 Organización del capitulado

Este trabajo tiene dos aportes principalmente, que son la implementación de una CNN y la tecnología de FPGA para la implementación de este tipo de redes en estos dispositivos. El capítulo 2 inicia con la descripción del problema de reconocimiento en imágenes, una breve introducción a lo que es el Deep Learning y sus áreas de aplicación. De particular interés son las CNNs, se estudia su principio de operación y el impacto que han tenido en este campo de la visión por computadora. Para finalizar este apartado se hace una breve introducción sobre los FPGAs, mencionando su arquitectura general, elementos básicos y su ciclo de diseño

El capítulo 3 está dedicado al estudio de las ventajas para el procesamiento en paralelo que tiene un FPGA en comparación con un CPU. Se investigan las técnicas desarrolladas con miras en hacer eficiente el procesamiento de las capas convolutivas de una CNN, tanto en el tiempo de procesamiento como en el consumo de energía. Se hace un análisis sobre la complejidad computacional y espacial de una CNN, esto con el propósito de determinar la cantidad de recursos computacionales necesarios en este tipo de redes artificiales. Este apartado concluye con tres alternativas con las que se cuenta para la implementación de este tipo de algoritmos en hardware.

En el capítulo 4 se hace el entrenamiento y prueba de una CNN clasificadora de señales de tránsito vehicular usadas en México y otros países. Se mide el tiempo de entrenamiento y desempeño en la clasificación para dos casos de experimentación. Adicionalmente se hace el análisis sobre los recursos necesarios para implementar la arquitectura obtenida en hardware, así como una propuesta de implementación de la CNN. Finalmente, en el capítulo 5 se analizan los resultados obtenidos y se dan las conclusiones de la presente investigación, así como el trabajo futuro. En el apéndice A se proporciona el código implementado para entrenar y probar la CNN.

Capítulo 2 Marco teórico y estado del arte

2.1 Reconocimiento en imágenes digitales

Una imagen en escala de grises se define como una función bidimensional *f (m, n)*, donde *m* y *n* representan las coordenadas espaciales. El valor de *f* en una coordenada *(m, n)* representa la intensidad de la imagen en ese punto. Una imagen analógica se representa mediante funciones continuas tanto en la posición como en la intensidad. Por lo regular una imagen se encuentra en un formato analógico (como sucede con la mayoría de los fenómenos en la naturaleza), pero el tratamiento de imágenes resulta más simple si la imagen se encuentra en un formato digital. Para digitalizar la imagen se debe muestrear la posición y cuantificar la intensidad, con lo que tanto la posición como la intensidad ahora tendrán valores discretos. Ahora a cada coordenada (*m, n*) de la imagen se le llama pixel. De forma que una imagen digital la podemos ver como una matriz de pixeles, de la forma:

$$f(m,n) = \begin{pmatrix} f(1,1) & f(1,2) & \cdots & f(1,N-1) & f(1,N) \\ f(2,1) & f(2,2) & & f(2,N-1) & f(2,N) \\ \vdots & & \ddots & \vdots & \\ f(M-1,1) & f(M-1,2) & & f(M-1,N-1) & f(M-1,N) \\ f(M,1) & f(M,2) & \cdots & f(M,N-1) & f(M,N) \end{pmatrix} \quad (2.1)$$

El tamaño de la imagen se define como la cantidad de pixeles que contiene, expresada en un formato (NxM), La *resolución* de la imagen se refiere a la cantidad de pixeles por unidad de longitud. Entre mayor sea la resolución, mejor la calidad de una imagen. La *profundidad de bits* se refiere a la cantidad de bits que se ocupan para representar un pixel, esto da la cantidad de tonos que puede contener una imagen. Por ejemplo, una imagen binaria tiene una profundidad de bit igual a uno, lo que significa que se requiere de un bit para representar un pixel. Esto origina una imagen “binaria” y significa que cada bit puede ser 0 o 1, equivalente a decir negro o blanco, respectivamente. El número de bits empleados para representar un pixel

da la cantidad de tonos que puede contener una imagen. La cantidad de tonos se determina con la expresión 2^n, donde n es la cantidad de bits empleados.

Algo muy común es emplear 8 bits para representar la intensidad de cada pixel, lo que significa que se pueden representar 256 tonos ($2^8 = 256$). La figura 2.1 muestra una imagen en escala de grises, en esta la cantidad de tonos es 256 y se muestra el valor de cada pixel en una sección de 11x11 pixeles de la imagen. La representación de cada pixel está en un formato decimal.

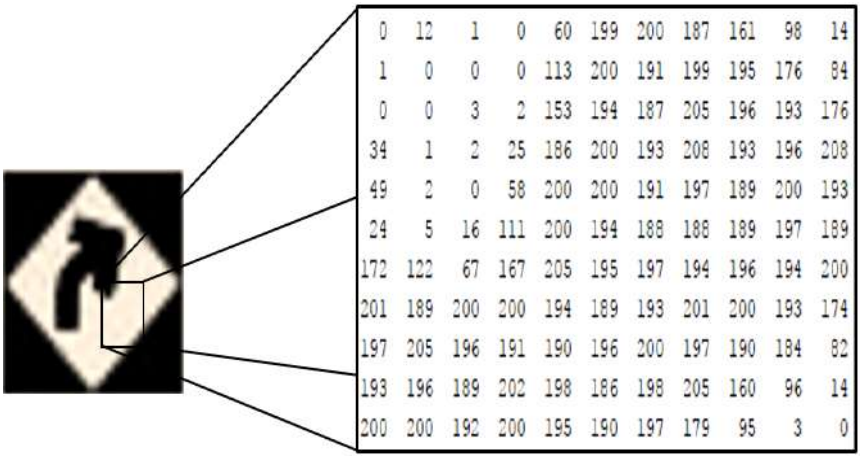

Figura 2.1. Pixeles de una imagen en escala de grises (8 bits por pixel).

Para una imagen a color se representa la misma información anterior, pero por cada uno de los tres canales, rojo, verde y azul (en el caso del formato RGB). De forma que una imagen a color se forma con la combinación de los tres canales. En la figura 2.2 se muestra el concepto.

El reconocimiento de patrones en general es un área de la Inteligencia Artificial que ha estado en constante evolución por sus diversas aplicaciones en las que se ha logrado éxito. En los últimos años se ha tenido un importante avance en diversos ámbitos, como en robótica, navegación de vehículos autónomos, aplicaciones con drones, aplicaciones para samartphones, además de la mencionadas en el capítulo uno. Esto gracias a la evolución que se ha conseguido, tanto en el hardware como software adecuado para esos propósitos.

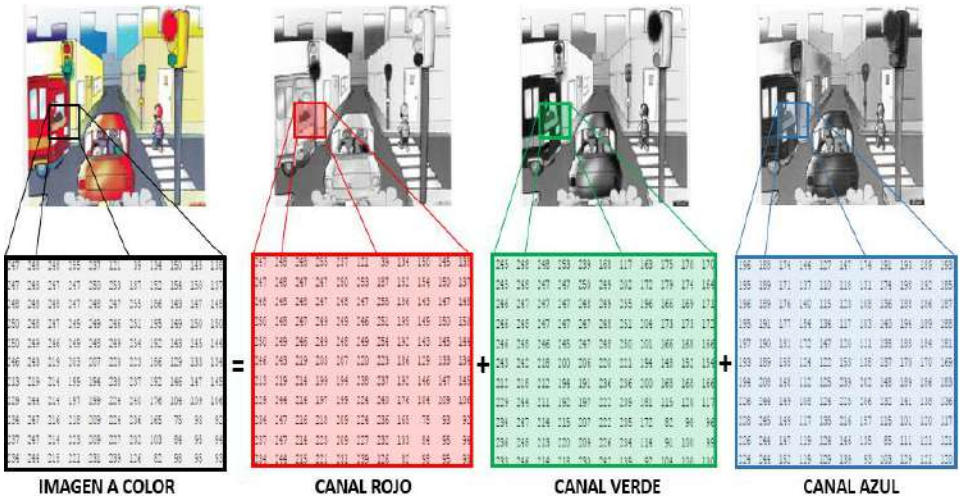

Figura 2.2. Formación de una imagen a color (8 bits por pixel).

Hablando en particular sobre el reconocimiento de objetos en imágenes o video, son dos los enfoques en los que se ha trabajado de forma intensa:

1) Los modelos realizados a mano o "*modelos tradicionales*" para algunos autores.

2) Los modelos basados en el *aprendizaje automático* (Redes Neuronales Artificiales).

Son diversas las tareas en las que se ha evolucionado en años recientes, como clasificación, detección de objetos, detección y clasificación, segmentación, descripción de escenas; en imágenes o video. Aunque este tipo de tareas son muy simples y naturales para un humano, resultan ser muy complicadas de llevar a cabo de forma artificial. Hablando de clasificación de imágenes, por ejemplo, hasta antes del 2013 una tasa de error del 26 % era lo reportado en el estado del arte por los autores de los mejores algoritmos para esa tarea. Esto significa que un sistema clasifica de forma incorrecta 26 imágenes de cada 100 que se le presenten. La figura 2.3 muestra la razón por la cual resulta complicado llevar esa tarea de forma artificial. Como se observa en la figura, para un humano es natural identificar que la imagen mostrada corresponde a una señal de tránsito preventiva, y que indica que se acerca una curva doble en una avenida. Sin embargo, "lo que ve el sistema" sólo

son números que indican la intensidad de cada pixel de la imagen. De alguna forma hay que hacer que el sistema clasifique el conjunto de esos números como una señal preventiva, y que no la confunda con ninguna otra señal de tránsito vehicular, aunque se le parezcan, u otro objeto.

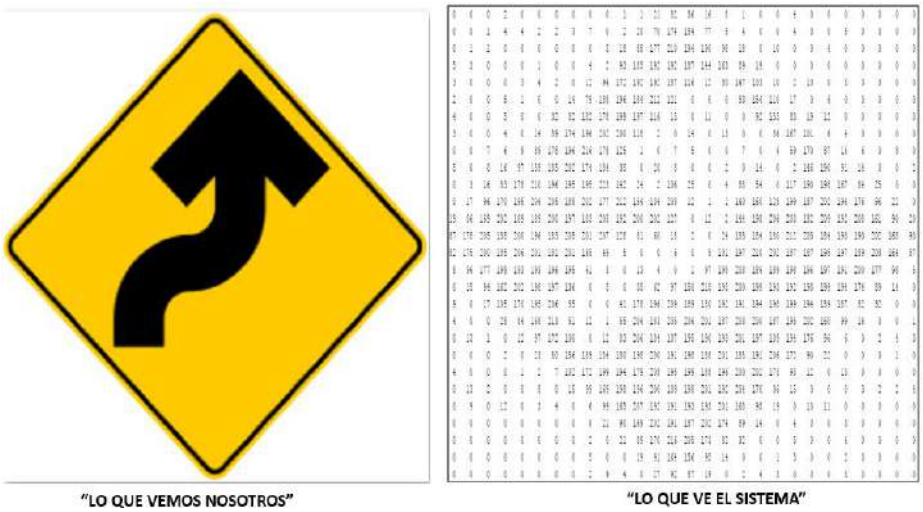

Figura 2.3. Diferencia entre lo que percibimos y lo que percibe una máquina.

Cualquiera que sea el enfoque, el esquema general que indica los dos componentes básicos (extractor de características o descriptor y el clasificador) para llevar a cabo una aplicación de este tipo, se muestra en la figura 2.4. En la figura se hace referencia a un sistema en el que los objetos de interés son vehículos y peatones, lo que se requiere es clasificar esos objetos, además de ubicar su posición. En la figura no se presenta una etapa de pre-procesado, que bien pudiera estar presente.

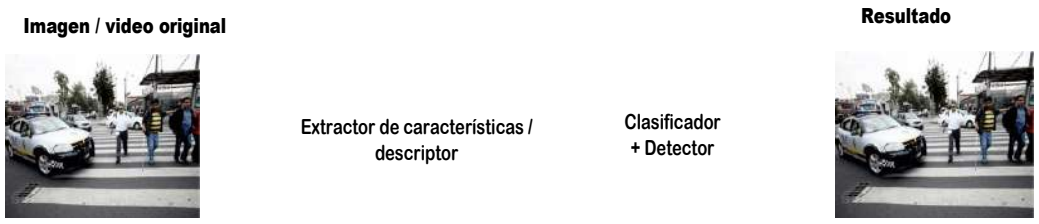

Figura 2.4. Esquema general para la detección y clasificación de objetos en imágenes.

Tanto en el modelo tradicional como en el basado en aprendizaje automático se intenta emular la inteligencia humana en cuanto a la visión se refiere. La diferencia entre ellos está en que en el primero, el descriptor es diseñado a mano por expertos en el área, y para el clasificador por lo regular han utilizado una Máquina de Soporte Vectorial (SVM) o el algoritmo de AdaBoost, como *en* (Sotelo, Parra, Fernández, & Naranjo, 2006) *y* (Gerónimo, Sappa, López, & Ponsa, 2006) por mencionar algunos. En cambio, en el enfoque basado en el aprendizaje automático con redes neuronales artificiales, se trata de que una máquina, mediante ejemplos, aprenda a extraer las características de interés de manera automática, y de la misma forma haga la clasificación de objetos, "como lo hace un humano". Claro que, aún no se ha logrado comprender a ciencia cierta el funcionamiento del cerebro, por consiguiente, el diseñar un modelo o algoritmo que lo emule está lejos de la realidad. Aún con esto, ambos enfoques han mostrado un desarrollo satisfactorio. Sin embargo, en años recientes, el aprendizaje automático, pero con redes neuronales profundas (DNN), ha mostrado mejores resultados, incluso en otras áreas del reconocimiento de patrones.

Autores como Geoffrey E. Hinton (pionero en este campo de investigación), opinan que la razón de esa mejora es que, "*en el enfoque tradicional se trata de modelar de manera formal la forma en que nuestro cerebro interpreta las cosas que son naturales para el ser humano, como la visión, el reconocimiento de la voz y el procesamiento del lenguaje natural, aun cuando no sabemos cómo lo hace"*. En la tarea de reconocimiento de objetos en imágenes o video se debe tener en cuenta variables como: los cambios en la escena, la escala, iluminación y oclusión, entre otras. Es aquí en donde los modelos tradicionales enfrentan desventajas, ya que habría que estar ajustando el algoritmo para cada cambio en las características de esas variables.

Por su parte, en el aprendizaje automático, ese problema se puede resolver haciendo que la red se aprenda los diferentes patrones que pueden ocurrir al variar las características. En este enfoque, se pretende que el propio sistema mejore conforme al incremento y variación de los datos, *de la misma forma que el ser*

humano mejora con la experiencia. Se ha tomado beneficio del Big Data en nuestros días, para esto ha sido necesario poder procesar la cantidad de información disponible. Hablando del aprendizaje profundo, y en particular de las Redes Neuronales Convolucionales, aplicaciones como la descrita en la figura 2.4 son end-to-end, esto es que la sección del descriptor y el clasificador se llevan cabo por la misma red, y no por separado.

2.2 Deep Learning

El paradigma del Deep Learning ha mejorado significativamente el estado del arte en tareas que resultaban difíciles de llevar a cabo por una máquina. Pero ¿Qué es el Deep Learning? Yann LeCun, Yoshua Bengio, y Geoffrey Hinton son pioneros investigadores en este campo, en su artículo (LeCun, Bengio, & Hinton, 2015) mencionan que "*el Deep Learning permite a modelos computacionales que están compuestos de múltiples capas de procesamiento, aprender representaciones de datos con deferentes niveles de abstracción*". Esos diferentes niveles de abstracción permiten hacer diversas representaciones (del mundo real), como resultado final del modelo computacional. Un ejemplo de ello sería la formación de un párrafo (texto), de manera artificial. La idea se muestra en la figura 2.5. En el ejemplo, la entrada son datos que representan caracteres, caracteres especiales, dígitos, espacios, etc.

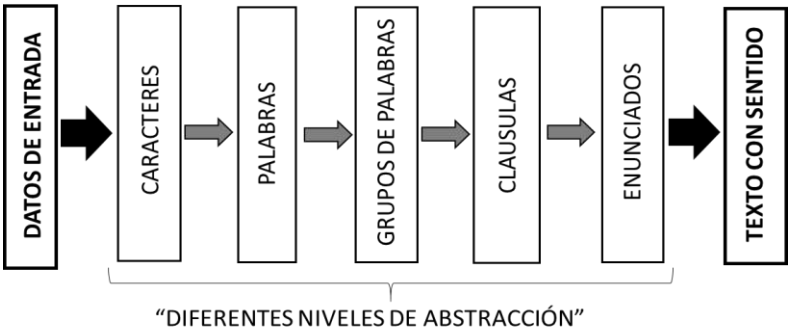

Figura 2.5. Idea del concepto de Deep Learning para formación de texto.

Cada etapa representada, extrae diferentes características, y como se observa, los datos de salida de una etapa son la entrada a la siguiente, el nivel de

abstracción se incrementa con forme se avanza. El resultado final es un texto que tiene sentido. Una característica de esto, es que la cantidad de datos necesaria para que esto funcione, debe ser muy grande.

El mismo concepto se puede usar en otras aplicaciones, por ejemplo, en el caso de visión por computadora, suponiendo que ahora la tarea es detectar y clasificar objetos en imágenes o video; los datos de entrada serían los pixeles que representan las imágenes o tramas de video. En lugar de caracteres, hablaríamos de pixeles individuales, las palabras ahora serían líneas de contornos formadas por grupos de pixeles, los de grupos de palabras serían formas que se generan a partir de los contornos, en lugar de cláusulas se tendrían partes de objetos, finalmente se tendría la formación de los objetos y su correspondiente clasificación, además de su ubicación en la imagen.

Evidentemente, para los ejemplos anteriores, la cantidad de capas varía de acuerdo a la aplicación y al nivel de abstracción que se desea tener. En este paradigma, lo que se tiene en cada una de las capas mencionadas en la figura 2.5 son principalmente redes neuronales artificiales. El término de “profundo” viene del hecho que se trata de más de una capa intermedia.

En (Goodfellow, Bengio, & Courville, 2016, pág. 8) se refieren al Deep Learning “*como un enfoque de la Inteligencia Artificial, un tipo de aprendizaje automático que alcanza gran potencia y flexibilidad mediante el aprendizaje de la representación del mundo, a través de conceptos jerárquicamente anidados. Se trata de formar conceptos complejos mediante la extracción y concatenación de conceptos muy simples*”. Para estos autores, el aprendizaje automático es el único enfoque viable, que permite construir sistemas de Inteligencia Artificial que pueden operar en los complicados ambientes del mundo real.

La constante investigación que se ha tenido en el aprendizaje automático a través de los años, ha dado lugar a una gran cantidad de algoritmos y modelos, muchos relacionados entre sí y generando infinidad de aplicaciones en conjunto. El Deep Learning es uno de estos modelos, la figura 2.6 muestra en donde queda

ubicado este paradigma. Se puede definir como un enfoque del aprendizaje automático basado en redes neuronales artificiales de varias capas, que se ha beneficiado del Big Data, por la gran cantidad de datos que se maneja en la actualidad y la reciente capacidad de procesamiento que se tiene con los dispositivos usados para ese propósito. Además, el desempeño en una aplicación debe de mejorar conforme se incrementa la cantidad de información, de la misma forma que los humanos aprendemos con la experiencia.

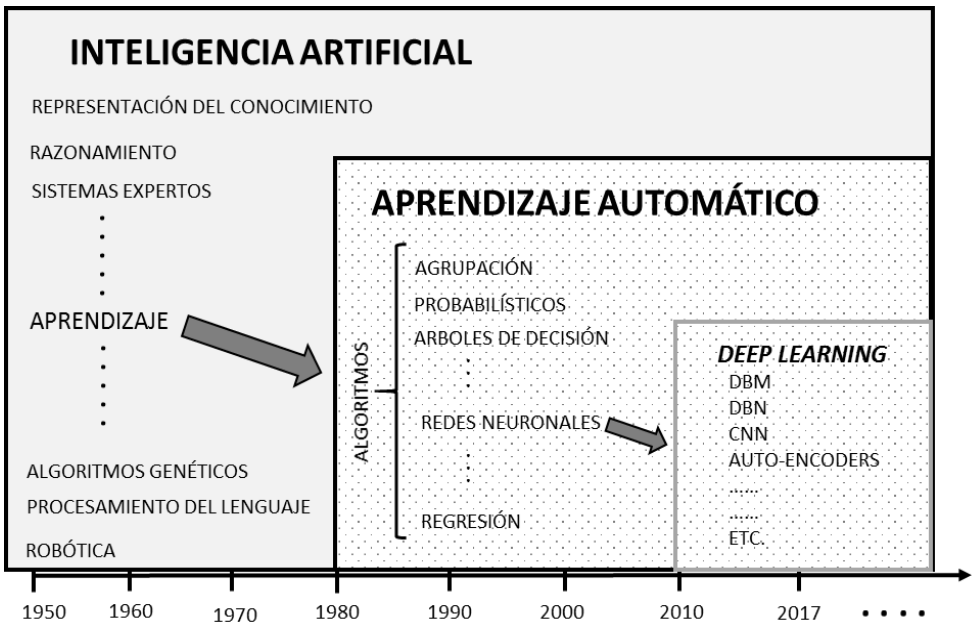

Figura 2.6. Ubicación del paradigma Deep Learning.

Como ya se mencionó, una característica que diferencia a las redes neuronales artificiales, del paradigma de Deep Learning, es la cantidad de capas y parámetros que se manejan en cada caso. Por ejemplo, en trabajos como (Coates, y otros, 2013) y (Le, y otros, 2012), trabajan con 10 millones de imágenes de 200x200 pixeles; con 11 billones de parámetros en el primer caso y un billón de parámetros en el segundo. El poder procesar esas cantidades de datos, es lo que da ventajas al Deep Learning, ya que con ello se pueden extraer características de interés (según la aplicación), con diferentes niveles de abstracción y exactitud. El uso adecuado de estas características en cada caso, ha propiciado los resultaos que se están obteniendo. Este desarrollo se ha enfocado en tres áreas

principalmente: visión por computadora, reconocimiento de voz y, más recientemente, el procesamiento de lenguaje natural. Sin embargo, las aplicaciones no están limitadas a ello. El presente trabajo se enfoca particularmente en el reconocimiento de objetos en imágenes y video.

2.2.1 Antecedentes de Deep Learning

Aunque este paradigma tiene sus orígenes en los 80's con Geoffrey Hinton y Yann LeCun, su potencial se ha desarrollado principalmente en la última década. Para tener una idea sobre el desarrollo en este campo de la Inteligencia Artificial y el aprendizaje automático, en la figura 2.7 se presenta información extraída de la base de datos de google® (google trends). En esta figura se muestra esa tendencia en el desarrollo, así como la evolución que ha tenido el Deep Learning.

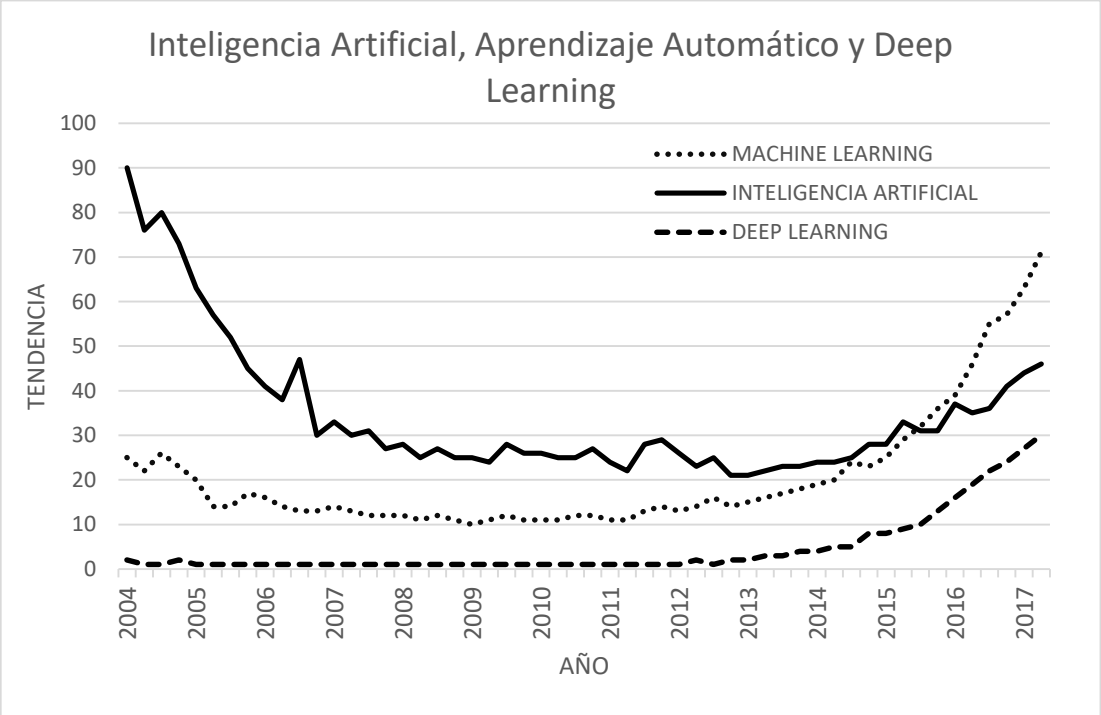

Figura 2.7. Tendencias del Deep Learning en la Inteligencia Artificial.

En la figura 2.7 se observa que antes del año 2004, el concepto de Deep Learning prácticamente no existía, salvo para la comunidad de investigadores que trabajaban en el área en esos años.

Sin embargo, las redes neuronales tienen sus orígenes desde 1943 con McCulloch y Pitts. Entre 1965 y 1971 surge el perceptrón multicapa, con sus creadores Ivakhnenco, Lapa y McDonough. En los 80's surge la segunda ola en la investigación de redes neuronales artificiales. La primera red neuronal que puede merecer el atributo de "profunda" fue el Neocognitrón por Kunihiko Fukushima en 1980, esta fue la inspiración para las redes neuronales convolucionales. El algoritmo de backpropagation ya existía cuando Rumelhart, Hinton y Williams lo popularizaron en 1986 con su trabajo "*Learning internal representations by error propagation*". En (LeCun, y otros, 1989), este clásico algoritmo fue aplicado a redes neuronales convolucionales, sus resultados fueron las bases para las modernas aplicaciones con este tipo de redes.

La investigación y el uso de redes neuronales artificiales profundas continuaron con los años, se percibían buenos resultados en diversos ámbitos y aplicaciones. En (LeCun, Bottou, Bengio, & Haffner, 1998), los autores implementaron una CNN para clasificar imágenes de dígitos escritos a mano, con la que demostraron clara evidencia de poder generar mejores resultados con este tipo de redes, sobre los métodos tradicionales. La limitante de aquellos años fueron los equipos de cómputo con los que se contaba. Trabajos como (Chellapilla, Puri, & Simard, 2006) y otros parecidos, fueron uno de los detonantes para el incremento en la investigación con este paradigma.

Por otro lado, está la disponibilidad de modernas bases de datos que permiten hacer investigación en múltiples áreas. En 2006 se incrementó la investigación en el aprendizaje no supervisado usando redes neuronales profundas, se obtuvo el record en la tasa de error en la clasificación de dígitos escritos a mano (MNIST). Se comenzó con la implementación de CNNs basadas en GPU, con lo que se superaba en cuatro veces la velocidad de ejecución, en relación con modelos basados en CPU.

Modernos equipos, permiten el incremento en el número de neuronas, así como la cantidad de conexiones por neurona. En (Goodfellow, Bengio, & Courville,

2016, pág. 26), presentan información de la cantidad de conexiones por neurona en las redes modernas (100,000 conexiones por neurona), igual que la cantidad de conexiones en un gato y sólo un poco por debajo del ser humano. Hasta el 2015 la cantidad de neuronas la ubican en poco más de 10,000,000 en ciertos trabajos. Estiman que con esa tendencia para el 2056 se estaría igualando al ser humano. Es debido a ese tipo de perspectivas que, compañías como Microsoft®, Google®, Facebook, Nvidia®, diversos centros de investigación y varios más, han apostado por esa área de investigación. Son variadas las aplicaciones en las que se enfocan, como navegación de vehículos autónomos, video juegos, procesamiento del lenguaje natural, traducción en tiempo real, reconocimiento y clasificación en imágenes o video y otras. Aún hay mucho por hacer, parte de los esfuerzos en la actualidad, se enfocan en el empleo de dispositivos capaces de procesar la cantidad de información necesaria, pero con un menor tiempo de latencia y bajo consumo de potencia.

2.2.2 Principales algoritmos de Deep Learning

Dependiendo del tipo de aplicación que se trate, hay que emplear el tipo de algoritmo más adecuado para ello. En (Deng & Yu, 2014), hacen una clasificación de las técnicas empleadas en el paradigma de Deep Learning, coinciden con la clasificación que hacen otros autores. Según el tipo de aprendizaje, de acuerdo a su arquitectura y a su finalidad que persiguen las clasifican en tres grupos:

1) **Redes profundas para aprendizaje no supervisado o generativo**. En este, el propósito principal es el análisis de patrones, la síntesis de los datos observados, o bien una agrupación, sin que se tenga una etiqueta para cada clase de patrón u objetivo.

2) **Redes profundas para aprendizaje supervisado**. En estas, la intención es una forma directa de discriminar, al contar con patrones conocidos y bien categorizados, con el fin de hacer clasificación de forma directa.

3) **Redes profundas híbridas**. Es una mezcla de las anteriores, con la meta de poder tener la capacidad de discriminar, auxiliado de aprendizaje no supervisado. Eso puede llevar a una mejor optimización y/o regularización que las redes supervisadas.

Otros autores sólo hacen la clasificación en las dos primeras categorías. Para ellos, la diferencia es en el tipo de técnica que utilizan, por lo regular el aprendizaje no supervisado o generativo está asociado con modelos probabilísticos. Como quiera que sea, los principales algoritmos empleadas en Deep Learning, se mencionan brevemente a continuación.

Deep Boltzmann Machine (DBM)

Las máquinas de Boltzmann (BM) tienen sus orígenes durante la segunda oleada de la investigación en redes neuronales (en los 80's), con lo que algunos autores nombran enfoque conexionista, se derivan de las redes Hopfield y son usadas para aprender distribuciones probabilísticas de vectores en forma arbitraria (Fahlman, Hinton, & Sejnowski, 1983). Se trata de modelos basados en energía, trabajan con un conjunto de vectores binarios aleatorios, básicamente lo que hacen es calcular la probabilidad de cada uno de los vectores binarios. Son útiles para determinar si otros conjuntos de vectores unitarios provienen de la misma distribución. Se pueden aplicar en la determinación de la ocurrencia de ciertas palabras en un texto, por ejemplo, o para monitorear sistemas complejos en búsqueda de comportamientos inusuales.

Una BM es difícil de entrenar y lenta para el aprendizaje, de ahí que surgen las Máquinas de Boltzmann Restringidas (RBM). Estas se consideran como uno de los bloques más comunes en los modelos probabilísticos. Tienen una arquitectura más simple, con una conectividad restringida para acelerar y facilitar el aprendizaje. Su arquitectura permite a las RBM, el apilarse una después de otra para formar modelos profundos, o Deep Boltzmann Machines (DBM), ver (Salakhutdinov & Hinton, 2009). La figura 2.8 muestra la estructura de una RBM profunda, las h_i representan capas ocultas, mientras qua V_i es una capa visible.

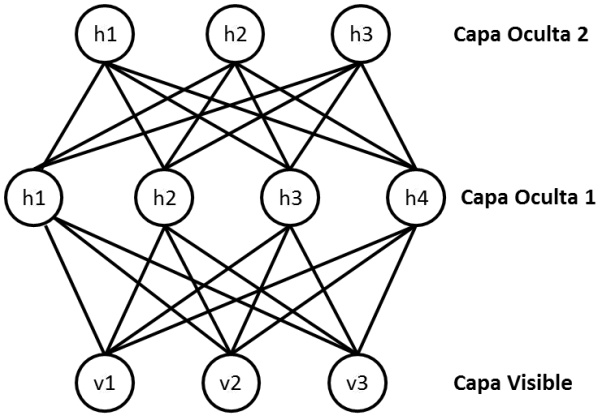

Figura 2.8. Máquina de Boltzmann Restringida con su capa visible y dos capas ocultas.

Deep Belief Networks (DBN)

Este tipo de redes fueron uno de los primeros modelos no convolucionales que incorporaron el entrenamiento de arquitecturas profundas *(Hinton et al 2006, Hinton 2007b)*. Con su introducción en 2006 se revolucionó el aprendizaje profundo, esto debido a que no resultaron difíciles de optimizar, a diferencia de otros modelos. Aunque en la actualidad son poco usadas, la aportación al aprendizaje profundo que tuvieron, hace que sean mencionadas como referencia en diversos documentos.

Auto-Encoders

Un auto-encoder, también conocido como auto-asociador, es una red neuronal que se entrena para intentar mapear sus entradas hacia sus salidas. Debido a que son más fácil de entrenar que una RBM, se han usado como bloques de construcción, para entrenar redes profundas, usando el aprendizaje no supervisado. Usualmente, tienen una capa de entrada con los datos originales, o un vector de características; una o más capas ocultas con las que se hace una transformación de características y una capa da salida, que se relaciona con la capa de entrada para hacer una reconstrucción de los datos originales. En (Deng & Yu, 2014) encontrará una explicación más detallada sobre auto-encoders, además de una aplicación en la que se extraen características de voz.

Convolutional Neural Networks (CNN)

Las CNNs son de los algoritmos más populares en el paradigma del Deep Learning, aunque también se utilizan en otras tareas; cuando se trata de reconocimiento de objetos en imágenes, estas son por defecto las usadas. El desempeño que han tenido se ubica entre los mejores en la actualidad. Tienen sus orígenes en (LeCun, Bottou, Bengio, & Haffner, 1998) y con el empleo de GPU's para su entrenamiento originaron un despunte importante en el estado del arte a partir del 2011, en cuanto a los resultados que se obtenían en las tareas de clasificación de objetos y reconocimiento de dígitos escritos a mano, ver (Ciresan, Meier, Masci, Gambardella, & Schmidhuber, 2011)*, y* (Krizhevsky, Sutskever, & E. Hinton, 2012). A partir de aquí se incrementó la investigación en este tipo de redes, hasta obtener las variantes en su arquitectura y resultados obtenidos de la actualidad. Dado que este trabajo está basado en este tipo de redes, en el apartado 2.3 se describen con más detalle.

Recurrent Neural Networks (RNN)

Las redes neuronales recurrentes son una familia de redes usadas para el procesamiento de datos secuenciales. Así como las CNN's son adecuadas para procesar datos que se encuentran en forma de una matriz, las RNN's se especializan en procesar datos que se encuentran en forma de un vector. A diferencia de las anteriores, en este tipo de redes, se permiten conexiones entre nodos de una misma capa. Esto las vuelve más complicadas para entrenarlas, pero lo que se obtiene a cambio, son redes con memoria, es decir, que son dinámicas; lo que conduce a aplicaciones como descripción de escenas en imágenes y otras más.

2.3 Redes Neuronales Convolucionales (CNN)

Las redes neuronales convolucionales son una variante del perceptrón multicapa, con la diferencia que las CNN's realizan operaciones de "convolución" entre los parámetros de la red y los datos de entrada, en lugar de productos punto. Son particularmente apropiadas para aplicaciones en las que los datos se encuentran

en forma de una rejilla, como matrices, por ejemplo. Si se quisiera trabajar con imágenes usando redes neuronales artificiales comunes (MLP), no sería muy adecuado por la cantidad de pesos que se necesitarían. Por ejemplo, para una imagen a color de 640x480 pixeles, la cantidad de pesos en la primera capa sería de 921,600 + 1, lo cual se haría inmanejable en una red multicapa, considerando que esa cantidad sería el valor aproximado con la que trabajaría cada capa. Para solventar ese problema, las CNNs procesan las imágenes por secciones, con lo que se reduce la cantidad de parámetros requeridos. Debido a esto, es que son utilizadas en aplicaciones que involucran visión artificial, tales como el reconocimiento de objetos en imágenes, clasificación de imágenes o video, segmentación, descripción de escenas, seguimiento de objetos en video y más.

El principio es manejar los datos que representan los pixeles en forma de volúmenes, y no como vectores. Las capas básicas para que funcione cualquier CNN son tres: capas de convolución, junto con su función de activación; capas de agrupación y las capas completamente conectadas. En la figura 2.9 se muestra la arquitectura básica de una CNN. En la figura, la entrada es una imagen a color de 254x254 pixeles. Enseguida vienen las diversas capas que integran la CNN; en este caso solo se representan las capas de convolución, agrupamiento, y además las que están completamente conectadas. Finalmente, se presenta la capa de salida.

Dependiendo de la aplicación, las últimas capas pueden variar, por ejemplo, si la tarea fuera clasificar imágenes, en la última capa tendría la misma cantidad de nodos que la cantidad de clases, un nodo por cada clase; los cuáles se activarían con diferente intensidad, indicando las probabilidades de las posibles clases a la que pertenece el objeto. En cambio, si la aplicación fuera localizar objetos en las imágenes, en la última capa, además de lo anterior habría que agregar la información sobre la ubicación de los objetos con las coordenadas, así como el ancho y altura del objeto. Para la tarea de detección, se agregaría información acerca de la predicción de cada objeto localizado, esto es, identificar la clase de objeto a la que pertenece. Si se tratara de describir lo que hay en una imagen, antes de las capas completamente conectadas, se insertaría una sección de redes

recurrentes, con lo que se pueda tener dinámica en la tarea. Las anteriores, y otras tareas posibles con el paradigma del Deep Learning, tienen a las CNN´s como base.

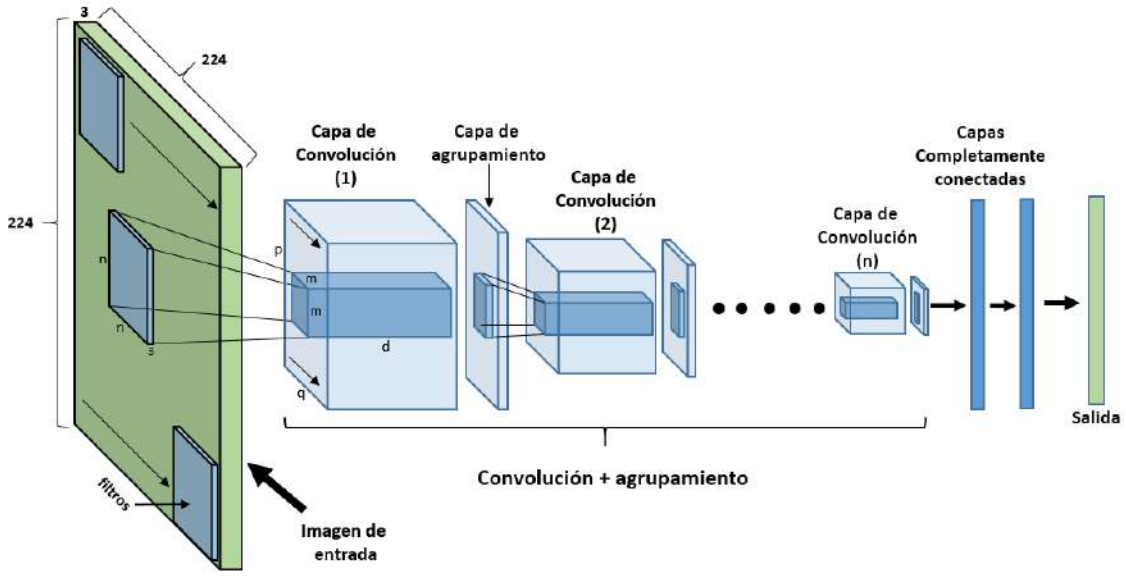

Figura 2.9. Arquitectura básica de una red neuronal convolucional.

Capas de convolución. En sí son el fundamento de las CNNs, se trata de una ventana deslizante. La operación de convolución, para nuestro caso, es digital y de dos dimensiones, queda definida como:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \qquad (2.2)$$

Donde I representa la imagen y K un filtro o kernel de tamaño mxn. Por lo regular $m = n$, de ahí que el tamaño del filtro representado en la figura 2.9 es de nxn. Los subíndices *i, j* representan la posición de los pixeles en la imagen sobre la que se hace la operación de convolución. Básicamente, consisten en un conjunto de parámetros entrenables, de tamaño reducido (nxn), típicamente de 3x3, 5x5, 7x7 u 11x11 pesos, además de que cada filtro debe tener la profundidad de los datos de entrada. Si la entrada es una imagen a color, la profundidad de los filtros sería tres, que corresponden a los canales RGB. Estos filtros se desplazan por toda la imagen

de entrada, desde la esquina superior izquierda hasta la esquina inferior derecha, como se observa en la figura 2.9. A la vez que se hace el desplazamiento por la imagen, se van haciendo las operaciones de producto punto y sumas, entre los pixeles de la imagen y los pesos de los filtros, que es la operación de convolución. De lo anterior resulta otro grupo de datos, en forma de un volumen de tamaño (*p*x*q*x*d*). Donde *d*, corresponde a la cantidad de filtros usados en la capa anterior. A estos datos se le puede repetir el mismo procedimiento, por varias capas, con nuevos conjuntos de filtros (*m*x*m*x*c*), donde *c* corresponde a la cantidad de filtros de la capa anterior. En el transcurso de estas capas, la dimensión de los cubos se va reduciendo en cuanto al ancho y altura.

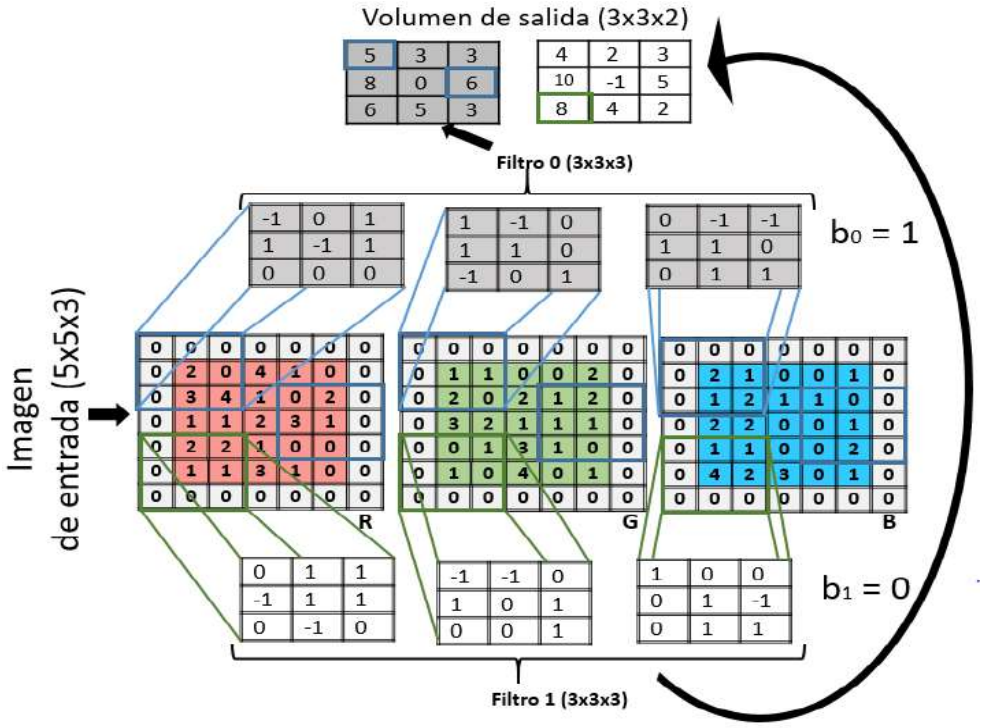

Figura 2.10. Ejemplo de convolución con dos filtros de 3x3 en una imagen RGB de 5x5.

La figura 2.10 muestra un ejemplo sencillo de la operación de convolución. En este, la los datos de entrada corresponden a una imagen RGB de 5x5 pixeles y se representa cada canal por separado. Se emplean dos filtros (filtro 0 y filtro 1) de 3x3, representados en gris y blanco respectivamente. Dado que el volumen de entrada (imagen RGB) tiene una profundidad de 3, se deben emplear tres canales

en cada filtro. Además, se representa el bias de los filtros como b_0 y b_1 respectivamente. En la parte superior se observa el volumen de salida, que es de 3x3x2. Esos datos de salida son el resultado de multiplicar y sumar los parámetros de cada filtro por los datos de entrada según la posición, y sumar el resultado de cada canal junto con el bias.

El propósito de los filtros, es producir mapas de activación, o características, los cuáles se activarán al pasar los filtros por regiones de la imagen que contengan las características buscadas en cada capa, como bordes, líneas, contornos, etc. En la figura 2.11 se muestra un ejemplo de lo que resulta al aplicar un filtro a una imagen. El fundamento al entrenar este tipo de redes, consiste en actualizar los parámetros de esos filtros, de forma que cada filtro extraiga las correspondientes características buscadas, según la aplicación. Esas características son las que eventualmente formarán objetos conforme se avanza en la red. En cada capa de convolución se requiere de una función de activación, aunque no se representa en las figuras anteriores.

Figura 2.11. Derecha: Efecto del filtro $K = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ *a la imagen. Izquierda: Imagen original.*

En cuanto a las dimensiones del volumen de los datos de salida para cada capa, depende de cuatro factores: la **cantidad de filtros** (K) usados como entrada, ***el paso*** del filtro (S), el ***relleno de ceros*** (P) y el ***campo visual*** (F) del filtro. La *profundidad* de los datos en la salida corresponde a la cantidad de filtros que se está empleando. El *paso* del filtro se refiere a la cantidad de pixeles que avanzará

el filtro al desplazarse cada vez por la imagen en cada operación. En cuanto al *relleno de ceros*, en ocasiones se debe rellenar los bordes de una imagen con ceros, de forma que el tamaño de la imagen, se ajuste al tamaño de los filtros y se pueda hacer el barrido completo. En el ejemplo de la figura 2.10 se hizo un relleno de ceros en los bordes de los datos de entrada. Si los datos de entrada a una capa de convolución son: $W_1xH_1xD_1$, que corresponden al ancho, altura y profundidad de los datos respectivamente, el volumen de datos en la salida queda determinado por las ecuaciones 2.3 a 2.5, para el ancho, altura y profundidad, respectivamente.

$$W_2 = \frac{(W_1 - F + 2P)}{S} + 1 \qquad (2.3)$$

$$H_2 = \frac{(H_1 - F + 2P)}{S} + 1 \qquad (2.4)$$

$$D_2 = K \qquad (2.5)$$

Funciones de activación. Para producir los mapas de activación, los datos resultantes de las operaciones de convolución se pasan por una función de activación. En las CNN publicadas recientemente se ha empleado una ReLU (Unidad Lineal Rectificada) como función de activación, esto, por un lado debido al cómputo que implica usar una sigmoide o $\tanh(x)$ como función de activación (muy comunes en otro tipo de redes), y por otro lado, ayuda a reducir el problema del descenso del gradiente al hacer el entrenamiento de la red. La capa ReLU aplica la función: $f(x) = \max(0, x)$ a todos los valores del volumen de entrada. En términos básicos, esta capa sólo cambia todas las activaciones negativas a 0, dejando los valores positivos igual.

Capas de agrupamiento. El propósito de estas capas, es reducir el tamaño espacial de los datos progresivamente, con el fin de reducir la cantidad de parámetros a tratar y consecuentemente la cantidad de cálculos, ayudando a

prevenir el overfitting (demasiados parámetros). Estas capas, operan de forma independiente a los datos de entrada de la capa anterior, sólo reducen el tamaño espacial de los datos usando una operación *MAX*, que consiste en dividir los datos en secciones, para extraer los valores máximos de cada sección, discriminando los demás. O bien extrayendo el promedio de cada grupo de datos. Estas capas reciben un volumen de entrada de $W_1xH_1xD_1$, que corresponden al ancho, altura y profundidad de los datos de entrada. Producen un volumen de salida $W_2xH_2xD_2$, que corresponden al ancho, altura y profundidad del volumen de salida, estos quedan determinados por las ecuaciones (2.6) a (2.8), respectivamente.

$$W_2 = \frac{(W_1 - F)}{S} + 1 \qquad (2.6)$$

$$H_2 = \frac{(H_1 - F)}{S} + 1 \qquad (2.7)$$

$$D_2 = D_1 \qquad (2.8)$$

La figura 2.12 muestra la forma de llevar a cabo esta operación. Para el ejemplo de la figura, a la izquierda se muestra un volumen de datos de entrada de *24x24x6*, y como resultado se obtiene un volumen de *12x12x6*. A la derecha se muestra un ejemplo con la operación *MAX*, como se observa en la figura, $W_1 = 4$, $H_1 = 4$, $F = 2$ y $S = 2$ (algo muy común en la práctica).

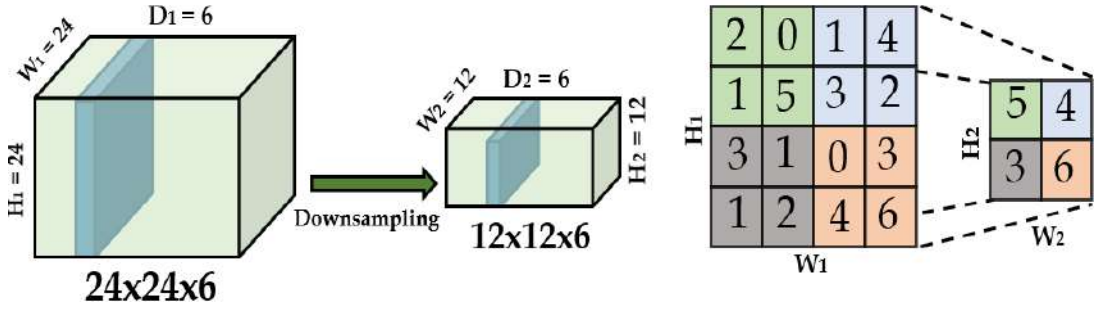

Figura 2.12. Ejemplo de capa de agrupamiento.

Según las ecuaciones 2.6 y 2.7, W_2 y H_2, que corresponden al ancho y altura de salida respectivamente, es 2. Como se observa, el tamaño espacial de los datos permanece igual en cuanto a la profundidad de los mismos se refiere, pero se reducen en cuanto al ancho y altura.

Capas completamente conectadas. Estas funcionan igual que en un MLP, es decir, cada nodo se encuentra completamente conectado a las salidas de la capa anterior. En cada nodo se hace una suma ponderada de sus parámetros por el valor de entrada, además de la entrada independiente o bias. Con esto, el que tenga la mayor puntuación, será el que tenga una mayor probabilidad. En sí, hacen una clasificación indicando la probabilidad de cada clase.

Las capas anteriores son las básicas requeridas para que una CNN funcione. En modernas redes, además de incrementar la cantidad de capas de convolución en su arquitectura, también se han incrustado otras capas más, con el propósito de prevenir el overfitting, el análisis de este tipo de capas está fuera del propósito de este documento.

2.3.1 Frameworks para desarrollo de CNN

Existen varios Frameworks para el desarrollo de redes neuronales convolucionales, estos facilitan en gran medida el trabajo para diseñar una red neuronal. *En* (Shi, Wang, Xu, & Chu) hacen un estudio comparativo entre cinco de los más populares actualmente, como Caffe, CNTK, MXNet, TensorFlow y Torch, todos ellos de código abierto y con diferentes plataformas para programación. No señalan a ninguno de ellos como el mejor, en cuanto al desempeño en general, ya que se crearon con diferentes propósitos y características en mente. La característica que comparten entre ellos y otros más, es que fueron creados teniendo en cuenta que el hardware sobre el cuál se implementarán las aplicaciones creadas es fijo, esto es CPU's multinúcleo o GPU's. Bajo este enfoque hay algunos otros que se crearon teniendo en mente un software en particular, como (MatConvNet, 2017) por ejemplo. En la

tabla 2.1 se muestran las características de los frameworks para desarrollo de CNNs más populares.

Tabla 2.1. Comparación de Frameworks para desarrollo de CNN populares.

Framework	Creador	Licencia	Código abierto	Plataforma	Interfaz	Soporte OpenCL	Soporte CUDA
Caffe	Berkeley Vision and Learning Center	BSD license	X	Linux, macOS, Windows	Python, MATLAB	En desarrollo	X
TensorFlow	Google Brainteam	Apache 2.0	X	Windows	Python (Keras), C/C++, Java, Go, R	En desarrollo	X
Microsoft Cognitive Toolkit	Microsoft Research	MIT license	X	Windows, Linux (macOS via Docker on roadmap)	Python (Keras), C++, Command line		X
Theano	Universidad de Montréal	BSD license	X	Cross-platform	Python (Keras)	En desarrollo	X
MatConvNet	Andrea Vedaldi, Karel Lenc	BSD license	X	Windows, Linux (macOS via Docker on roadmap)	MATLAB, C++		X
Torch	Ronan Collobert, Koray Kavukcuoglu, Clement Farabet	BSD license	X	Linux, macOS, Windows, Android, iOS	Lua, LuaJIT, C, utility library for C++/OpenCL	Implementaciones de terceros	X
Deeplearning4j	Skymind engineering team; Deeplearning4j community; originally Adam Gibson	Apache 2.0	X	Linux, macOS, Windows, Android (Cross-platform)	Java, Scala, Clojure, Python(Keras), Kotlin	En desarrollo	X
Keras	François Chollet	MIT license	X	Linux, macOS, Windows	Python, R	En desarrollo	X
Wolfram Mathematica	Wolfram Research	Proprietary		Windows, mac OS, Linux, Cloud computing	Wolfram Language		X

Dada la naturaleza de las CNN, es deseable que se implementen sobre hardware con la capacidad de procesar en paralelo, de ahí el éxito que se ha tenido con los GPU´s. En ciertas aplicaciones, y en particular para sistemas embebidos, es deseable hardware con dimensiones pequeñas y bajo consumo de potencia. Esto ha conducido al interés por parte de investigadores y diversos centros de investigación, en crear Frameworks para CNN con el enfoque en ese tipo de hardware, como FPGA's, (Zhu, Liu, Wang, & Xie, s.f.), (Sharma, y otros, 2016). Sin embargo aún se está en el camino de crear una herramienta que permita la

implementación de CNN directamente en FPGA's. Los trabajos citados lo hacen sobre sistemas en un chip (SoC's), en los que se tiene un CPU además del FPGA integrados en el mismo chip.

Otra opción es usar OpenCL™ (Open Computing Language), que consta de una interfaz para crear aplicaciones y de un lenguaje de programación. Juntos permiten implementar algoritmos con paralelismo a nivel de datos y de tareas que pueden ejecutarse tanto en CPU's como en GPU's. El lenguaje está basado en C, eliminando cierta funcionalidad y extendiéndolo con operaciones vectoriales.

Otro caso es CUDA™ (Compute Unified Device Architecture), que hace referencia a una plataforma de computación en paralelo incluyendo un compilador y un conjunto de herramientas de desarrollo creadas por NVIDIA® que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPU's de NVIDIA®.

2.4 Estado del arte de reconocimiento en imágenes y video

A partir del éxito de las redes neuronales convolucionales aplicadas al reconocimiento de objetos en imágenes, se tuvo un notable avance en el estado del arte en ese tipo de tareas, una evidencia de la anterior afirmación son los resultados que reportan en (Krizhevsky, Sutskever, & E. Hinton, 2012), sobre el reto ImageNet LSVRC-2010, en la tarea de clasificación.

Tabla 2.2. Comparación de resultados en el LSVRC-2010 fuente: (Krizhevsky, Sutskever, & E. Hinton, 2012)

Modelo	Tasa de error Top-1	Tasa de error Top-5
Sparse coding	47.1%	28.2%
SIFT + FVs	45.7%	25.7%
CNN	**37.5%**	**17.0%**

Este trascendente reto consiste en clasificar 1.2 millones de imágenes entre 1,000 clases diferentes, con 50,000 imágenes para validación y 10,000 para prueba.

Usaron una red neuronal convolucional con 60 millones de parámetros y 650,000 neuronas para la tarea. Parte de esos resultados se muestran en la tabla 2.2. En la tabla se observa una clara mejora en la tasa de error en la clasificación al usar una CNN, en comparación con los dos primeros métodos (modelos tradicionales), con los que se habían obtenido los mejores resultados reportados hasta ese año.

En general, esa tendencia de mejora con este paradigma se ha conservado hasta la actualidad, en la figura 2.13 se observa el desarrollo de este reto en cuanto a la tarea de clasificación (Russakovsky, y otros, 2017), además de los resultados en otras competencias populares con CNN's.

Las nuevas tareas en las que se compite actualmente son: detección y localización de objetos en imágenes y video. Los últimos resultados reportados, (competencia de 2017) en (Russakovsky, y otros, 2017), indican al equipo ganador con un porcentaje de error en la localización de 6.22%. Existen más competencias de ese estilo, como MINIST, CIFAR-10, CIFAR-100, STL-10 y SVHN por mencionar algunas. En el caso de CIFAR-10 (Object Recognition in Images), compitieron 231 equipos en el último evento, el ganador tiene una exactitud de 96.53% en la tarea, que supera al ser humano estimado en 94.8% aproximadamente para esta tarea (indicado en la figura 2.13). En esta, se trata de clasificar 60,000 imágenes a color de 32x32 pixeles entre 10 clases, en (Benenson, 2017) se muestran los resultados de estas competencias.

En los dos últimos años se han llevado a cabo la detección de objetos en video, con una precisión promedio de 80.83%, el seguimiento de objetos en video, así como la clasificación y análisis de escenas. En otros trabajos se han enfocado en clasificar video, como en (Karpathy, y otros, 2014), el seguimiento de objetos en video (Wang, Ouyang, Wang, & Lu, 2015) y la descripción del contexto en una escena (Donahue, y otros, s.f.), por mencionar algunos.

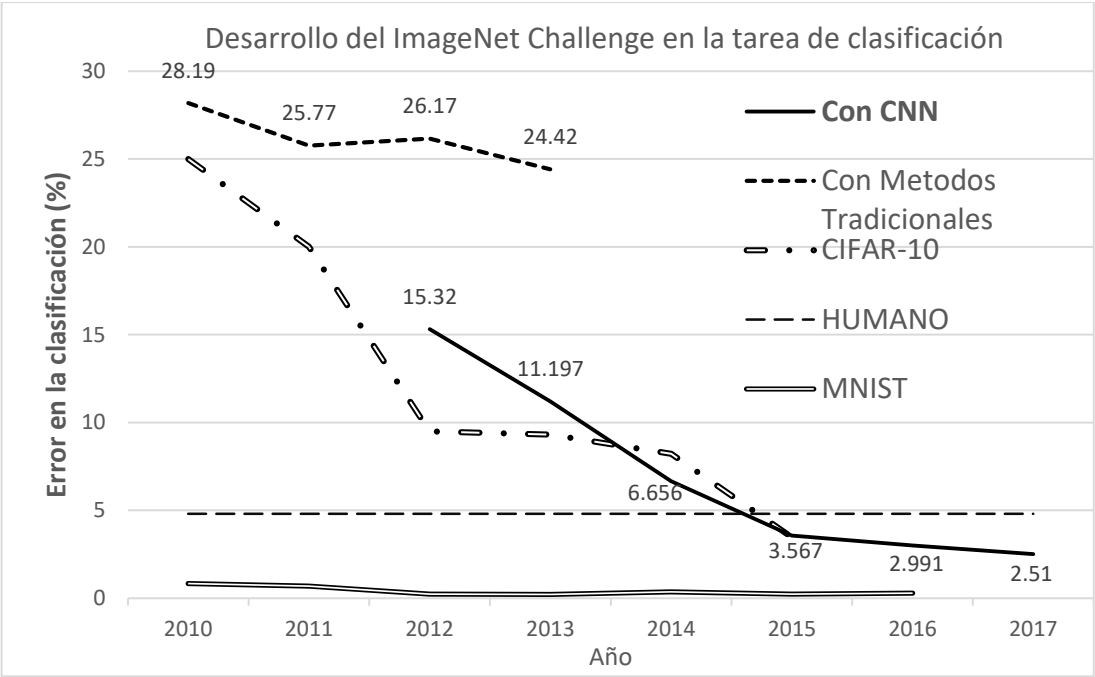

Figura 2.13. Desarrollo del ImageNet Challenge y otros concursos populares.

Del 2012 a la fecha se ha acentuado la investigación en cuanto a CNNs se refiere. En la tabla 2.3 se muestra una comparación entre las redes más populares publicadas y sus características.

Tabla 2.3. Comparación entre diferentes Redes Neuronales Convolucionales populares.

	# Capas Conv.	MACCs [x 10^6]	Parámetros [x 10^6]	Activaciones [x 10^6]	ImageNet top-5 error %
AlexNet (2012)	5	1140	62.4	2.4	19.7
Network-in-Network (2013)	12	1100	7.6	4.0	19.0
VGG-16 (2014)	16	15470	138.3	29.0	8.1
GoogLeNet (2015)	22	1600	7.0	10.4	9.2
ResNet-50 (2015)	50	3870	25.6	46.9	7.0
Inception v3 (2016)	48	5710	23.8	32.6	5.6
Inception-ResNet-v2 (2016)	96	9210	31.6	74.5	4.9
SqueezeNet (2016)	18	860	1.2	12.7	19.7

Se observa de la tabla, que con los años, se fue incrementando la cantidad de capas de convolución en las redes, esto trajo como resultado la reducción en la tasa de error, pero obviamente un incremento en la cantidad de operaciones multiplicación-acumulación (MACC) necesarias. Aquí solo se indica la cantidad de capas de convolución que contienen las redes, no así las demás capas.

La constante investigación ha originado diversidad entre los algoritmos empleados, y las aplicaciones anteriores han tenido su éxito gracias a dos factores principalmente con los que ahora se cuenta: La gran cantidad de datos de información, el llamado “Big data” de hoy en día, y sobre todo los equipos de cómputo que pueden manejar la gran cantidad de cálculos en tiempos adecuados.

2.5 Arreglos de Compuertas Configurables en Campo (FPGA’s)

Ya se ha señalado que para poder llevar a cabo las aplicaciones que tienen que ver con el paradigma del Deep Learning, se necesita, entre otras cosas, contar con equipos de procesamiento con la capacidad de operar en paralelo. Entre los dispositivos más adecuados para este fin se encuentran los CPU Multi-núcleo, GPU, FPGA y ASIC. De estos, los más populares, y para los que fueron pensados los frameworks anteriormente mencionados, son los GPU y CPU. Esto debido a que permiten el desarrollo de aplicaciones de una forma más directa y simple, se cuenta con soporte por parte del fabricante y son de uso más común. Las desventajas que tienen, en el caso se sistemas embebidos, son el consumo de potencia y espacio dimensional.

En ese sentido, el dispositivo más adecuado sería un ASIC, ya que está hecho a la medida de la aplicación, por lo tanto, el tamaño y cantidad de componentes en el chip sería el ideal, lo que se traduce en un menor consumo de potencia y menor tiempo de propagación de información. Las desventajas que se tiene con este son: el costo de implementación cuando la producción es en pequeña escala, el ciclo de diseño es largo, pero lo más importante es que no permite la

flexibilidad al momento de probar diversos algoritmos, o bien hacer cambios y actualizaciones a algún diseño determinado.

Los FPGA permiten esa flexibilidad, ya que son reconfigurables, y en la parte de consumo de potencia, son más eficientes que los GPU y CPU. En los siguientes apartados se menciona de manera general los conceptos relacionados a la tecnología de los FPGA, así como su relación con las CNN.

2.5.1 Antecedentes de FPGA's

Los primeros dispositivos programables por el usuario fueron los *Dispositivos Lógicos Programables* (PLDs) creados en los 70's. Estos se podían programar una sola vez, de ahí que se crearon los PLDs basados en tecnologías EEPROM o EPROM, con lo que ya se contaba con dispositivos reprogramables, pero limitados en su escala de integración, y por lo tanto en las aplicaciones en las que se pueden emplear (PAL, GAL). De ahí que surgen los CPLD's (Complex PLD), que contenían alrededor de 50 PLD en el mismo chip.

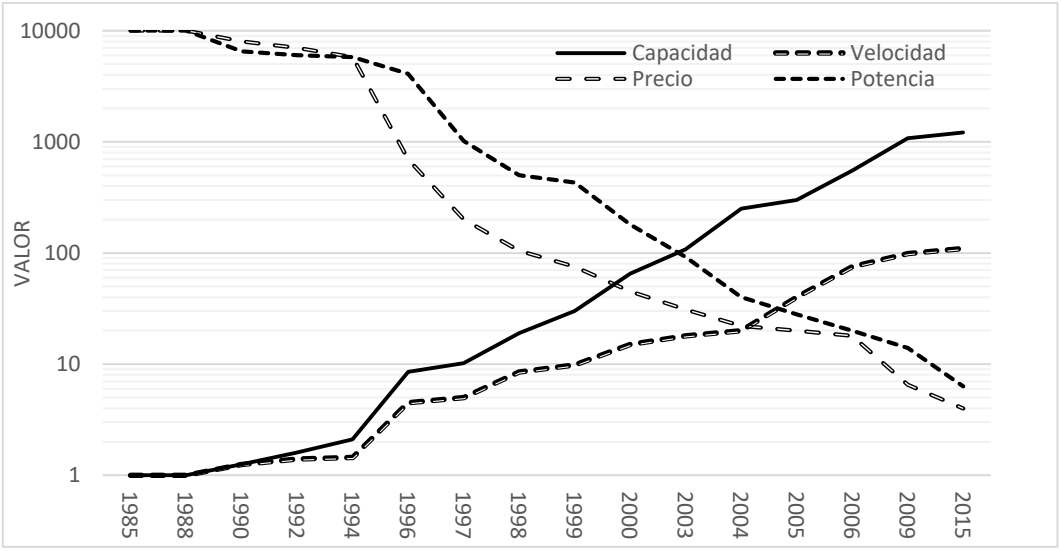

Figura 2.14. Desarrollo de los FPGA (Trimberger, 2015).

En 1984 surgen los FPGA, con una mayor escala de integración, además de que son basados en memoria RAM y con la capacidad de reconfigurar el

comportamiento que tendrá el hardware para cada aplicación. Han tenido un importante desarrollo desde su aparición, en (Trimberger, 2015) el autor hace una retrospectiva de la evolución en cuanto a la capacidad, velocidad, precio y potencia de 30 años que han tenido los FPGA de Xilinx (uno de los principales fabricantes), parte de esta información se muestra en la figura 2.14. De la figura resalta que han tenido un incremento en un factor de 10,000 y 100 en su capacidad y velocidad respectivamente, mientras que el costo y consumo de potencia se han reducido en un factor de 1,000 y un poco más de 1,000 respectivamente.

Son dispositivos que se pueden usar para resolver cualquier problema que sea computable, esto y otros factores han dado lugar a que las aplicaciones en las que se utilizan los FPGA son muy variadas, como: procesamiento de imágenes digitales, prototipado de ASIC's, manejo de imágenes médicas, visión por computadora, reconocimiento de voz, radio-astronomía, robótica, aplicaciones de control automático, radio definido por software (SDR), aplicaciones militares, entre otras. Al ser reconfigurables por el usuario, permiten flexibilidad al hacer cambios en algoritmos, y ofrecen la posibilidad hacer procesamiento de datos en paralelo, esto de forma temporal y espacial. De ahí el interés por usar un FPGA para acelerar el procesamiento involucrado en una CNN, ya que básicamente se trata con conjuntos de datos en forma matriz, con los que se realizan operaciones de convolución, activación, agrupación y otras. Es deseable que esas operaciones se realicen en paralelo y de forma eficiente. Lo mismo sucede con la adquisición de imágenes en el dispositivo.

Una desventaja que puede tener el usar un dispositivo configurable por el usuario como hardware para llevar a cabo cualquier aplicación, es la forma en la que se configura el dispositivo. Esto se hace mediante un lenguaje de descripción de hardware (HDL), que es el equivalente a usar un lenguaje de bajo nivel en un ambiente de software. Eso sería una limitante para aplicaciones sofisticadas, como el tratamiento de imágenes o video, por ejemplo. Una consecuencia del desarrollo que han tenido los FPGA's, son las herramientas que se han creado para generar aplicaciones, como los entornos de desarrollo de los propios fabricantes, así como

herramientas de terceros. Un ejemplo de ello es el HDL Coder® de Mathworks®, el cual permite crear aplicaciones “complicadas” en lenguajes de alto nivel, como Matlab® o Simulink® y de manera automática generar el código correspondiente en un lenguaje HDL, ya sea VHDL o Verilog. En cierta medida, esto simplifica la programación del dispositivo, pero no implica que el diseño final sea el óptimo en cuanto al uso y asignación de recursos, así como la lógica implementada.

2.5.2 Arquitectura general

La arquitectura de un FPGA cambia según la familia a la que pertenece, al igual que entre un fabricante y otro. Lo que no cambia para cualquiera de ellos, son los elementos básicos con los que cuentan estos dispositivos, que son:

1. **Bloques de entrada / salida (I / O):** Puertos de interconexión para datos de entrada y salida.
2. **Tablas de consulta (LUT):** Elemento que implementa operaciones lógicas.
3. **Flip-Flop (FF):** Registros para almacenar resultados de las LUT.

La combinación de estos elementos da como resultado la arquitectura básica de un FPGA, que se muestra en la figura 2.15 (Xilinx, SDAccel Enviroment, 2017). Los bloques lógicos configurables (CLB) son los componentes fundamentales, estos implementan la funcionalidad lógica del hardware y contienen a las LUT y FF en su interior. Los modernos FPGA’s, además de los elementos mostrados en la figura 2.15, cuentan con:

1. **Memoria embebida** para el almacenamiento de datos.
2. **Bucle de fijación de fase (PLL)** para accionar la estructura del FPGA a diferentes velocidades de reloj.
3. **Transceptores seriales** de alta velocidad para comunicaión.
4. **Controladores de memoria** fuera de chip.
5. **Bloques de procesamiento digital (DSP)** para operaciones de multiplicación-acumulación de forma eficiente.

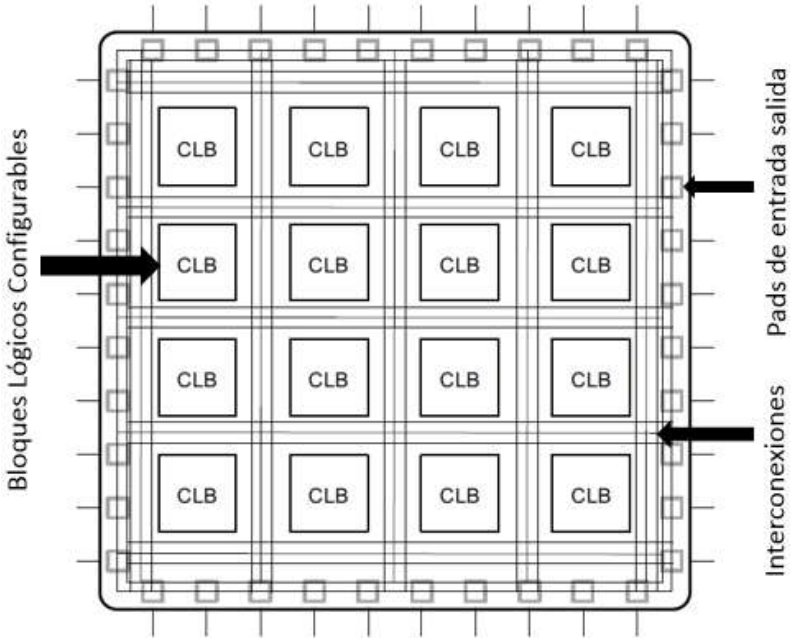

Figura 2.15 Arquitectura básica de un FPGA.

La LUT es el componente elemental de un FPGA, es capaz de implementar cualquier función lógica de N variables booleanas. Esencialmente, este elemento es una tabla de verdad en la que las diferentes combinaciones de las entradas seleccionan diferentes funciones (previamente asignadas), para producir valores de salida. Si N representa el número de entradas a la LUT, la cantidad de posiciones de memoria a las que accede la tabla es 2^N. Este mecanismo permite ahorrar tiempo de procesamiento, ya que es mucho más rápido copiar un dato de memoria que hacer el cálculo de dicho dato.

La implementación de hardware de una LUT se puede ver como una colección de celdas de memoria conectadas a un conjunto de multiplexores. Las entradas a la LUT actúan como bits selectores en el multiplexor, para elegir el resultado en un punto dado en el tiempo. Una LUT puede ser utilizada tanto como un motor de cálculo de funciones, como un elemento de almacenamiento de datos o como un registro de desplazamiento. La figura 2.16 muestra la representación funcional de una LUT.

El flip-flop es la unidad de almacenamiento básica dentro del tejido de un FPGA. Este elemento siempre está acompañado de una LUT para ayudar en la lógica de canalización y almacenamiento de datos. La estructura básica de un flip-flop incluye una entrada de datos, entrada de reloj, habilitación de reloj, reinicio y salida de datos.

Durante la operación normal, cualquier valor en el puerto de entrada de datos se enclava y pasa a la salida en cada pulso del reloj. El propósito del pin de habilitación del reloj, es permitir que el flip-flop sostenga un valor específico para más de un pulso de reloj. Las nuevas entradas de datos solo se retienen y pasan al puerto de salida de datos cuando tanto el reloj, como la habilitación del reloj son iguales a uno.

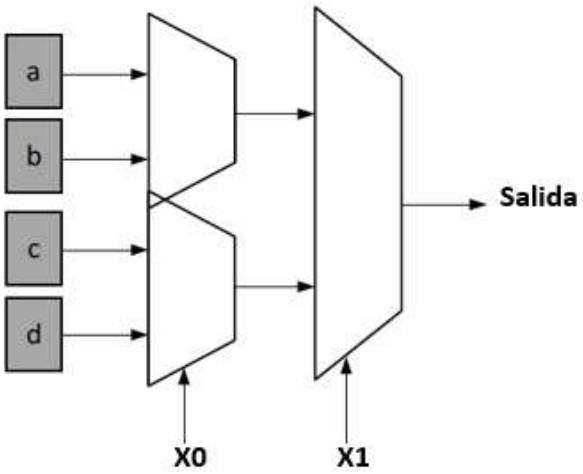

Figura 2.16. Representación de una LUT.

Un CLB puede comprender un único elemento lógico básico (BLE) o un grupo de BLE localmente interconectados. La figura 2.17 muestra la estructura general de un CLB, compuesto de 4 elementos lógicos básicos, en la misma figura se indica la estructura interna de un BLE simple, que comprende una tabla de consulta de 4 entradas (LUT-4).

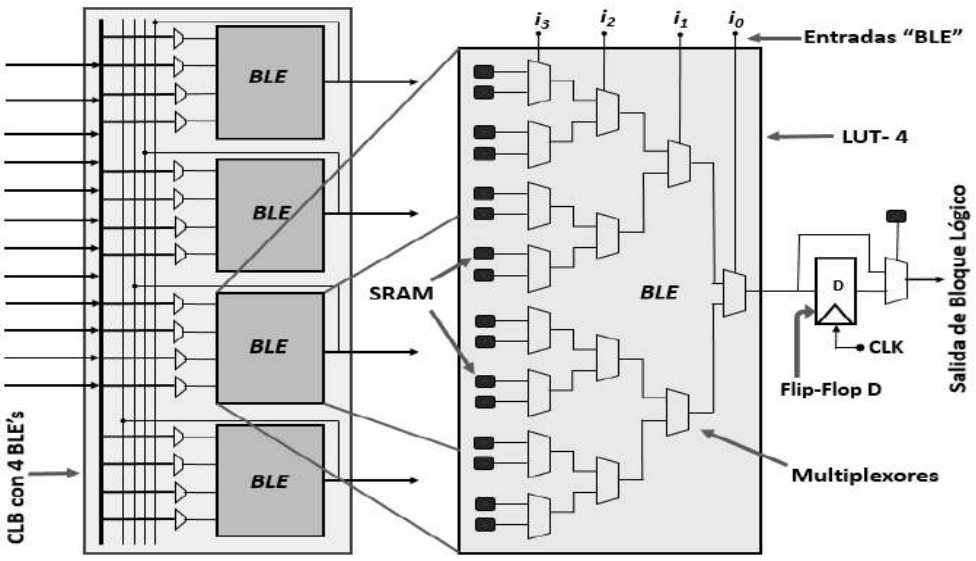

Figura 2.17. Estructura de un Bloque Lógico Configurable (CLB).

En la figura 2.17 se observa que el BLE cuenta con 16 unidades de memoria de acceso aleatorio estática (SRAM), en estas unidades se almacenan los datos de la tabla para que estén disponibles; de esta forma se puede implementar cualquier función booleana de 4 entradas. En la salida de LUT- 4 está conectado a un Flip-Flop tipo D, que es opcional. Un multiplexor selecciona la salida BLE, para que sea la salida del Flip-Flop o el LUT-4. Una tabla de búsqueda con mayor cantidad de entradas reduce la cantidad total de LUT's, requeridas para mapear un circuito de hardware. Se pueden mapear más funciones lógicas en una sola LUT. Esta finalmente reduce la intercomunicación entre las LUT, y por lo tanto la velocidad del hardware mejora. Sin embargo, una LUT con mayor cantidad de entradas aumenta su área exponencialmente.

El bloque computacional más complejo disponible en un FPGA Xilinx® es el bloque DSP48. Este bloque es el equivalente a una unidad lógica aritmética (ALU), según el modelo del dispositivo que se trate, la cantidad de estos bloques varía, pero en todos los casos hablamos de cientos de estos bloques.

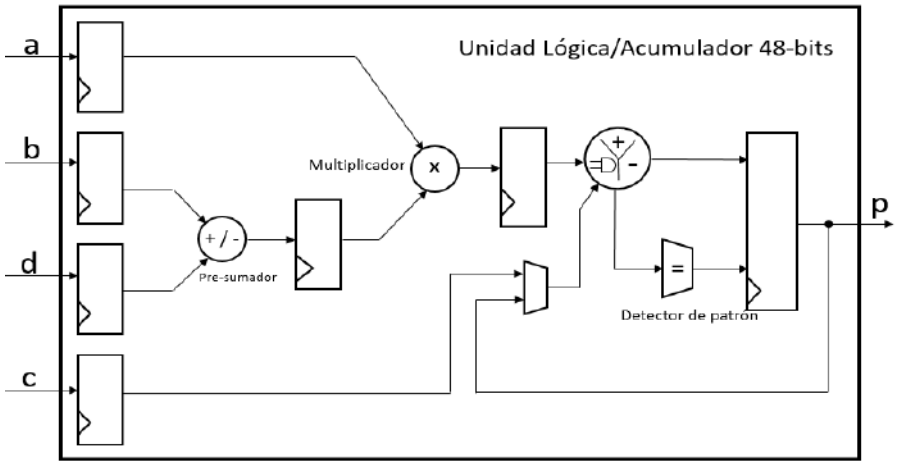

Figura 2.18. Estructura de un bloque DSP48 de Xilinx. Fuente (Xilinx)

La figura 2.18 muestra la estructura de este bloque, se compone de una unidad de suma/resta conectada a un multiplicador, y este conectado a un motor de suma/resta/acumulación final. Esta estructura permite implementar de forma eficiente funciones de la forma:

$$p = a\ x\ (a + b) + c \qquad \text{y} \qquad p+ = a\ x\ (b + d) \tag{2.9}$$

Los FPGA incluyen elementos de memoria incorporados que se pueden usar como memoria de acceso aleatorio (RAM), memoria de solo lectura (ROM) o registros de desplazamiento. Estos elementos son bloques RAM (BRAM), LUT y registros de desplazamiento. El BRAM es un módulo de doble puerto instanciado en el FPGA, para proporcionar almacenamiento en el chip para un conjunto relativamente grande de datos. Los tipos de memorias BRAM disponibles en el dispositivo pueden contener 18 o 36k bits. El número de estas memorias disponibles en el dispositivo depende de la familia de FPGA. La naturaleza de doble puerto de este tipo de memorias permite el acceso a diferentes ubicaciones en el mismo ciclo de reloj. Como se mencionó anteriormente, la LUT es una pequeña memoria, en la que el contenido de una tabla de verdad se escribe durante la configuración del dispositivo. Estos bloques se pueden usar como memorias de 64 bits y se conocen comúnmente como memorias distribuidas. Este es el tipo más rápido de memoria

disponible en el dispositivo, porque se puede instanciar en cualquier parte, de forma que mejore el desempeño del circuito implementado.

Existen varias opciones para seleccionar el dispositivo adecuado para implementar CNN's en FPGA. Los principales fabricantes de estos son Xilinx® e Intel (antes Altera), que juntos deben superar el 80 % del mercado a nivel mundial. Estos fabricantes ofrecen las opciones más inmediatas. Factores importantes a considerar son las dimensiones del sistema a implementar, el FPGA se debe ajustar a esos tamaños. Otra opción que pareciera incluso más viable es usar un System on a Chip (SoC), ya que este dispositivo además del FPGA tiene incorporado un CPU al interior del mismo chip, lo que ofrece mayores posibilidades para el control del flujo de datos, además de la capacidad de procesamiento que tiene el FPGA.

La figura 2.19 muestra la arquitectura general de un SoC (Crockett, Elliot, Enderwitz, & Stewart, 2014). Como se observa en la figura, al interior del mismo chip encontramos los elementos de un microprocesador, como el sistema operativo, memoria, interfaces con el harware, y demás, pero también se tiene la parte del hardware configurable (FPGA). En el caso del Zynq-7000™ de Xilinx® se combina la capacidad de procesamiento de un FPGA (Artix-7™) con la facilidad del control que ofrece un CPU (ARM® Cortex® A-9).

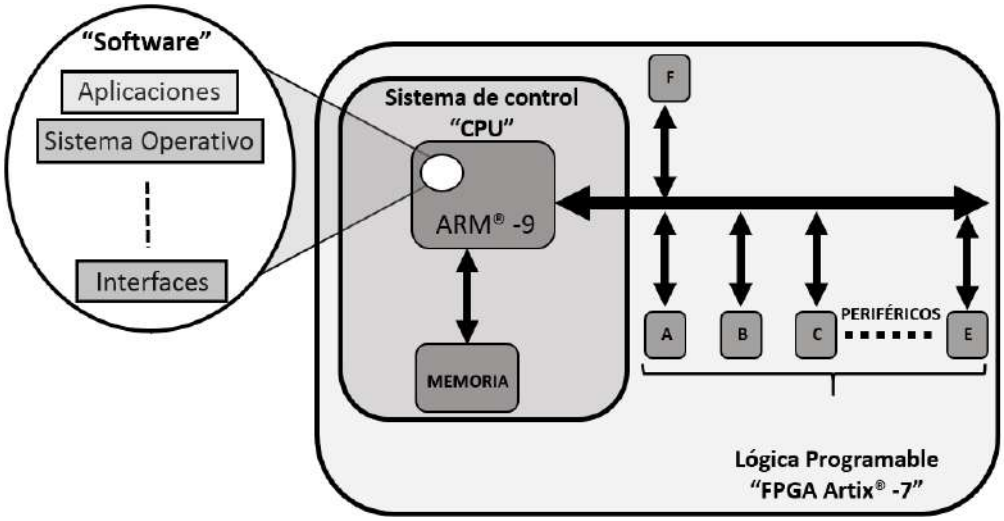

Figura 2.19 Arquitectura Zinq-7000 de Xilinx.

2.5.3 Flujo de diseño en FPGA

En la figura 2.20 se muestra el flujo de diseño general para FPGAs. Los pasos indicados en la parte central son requeridos, mientras que los que se encuentran a los costados son opcionales. A continuación se describe cada paso:

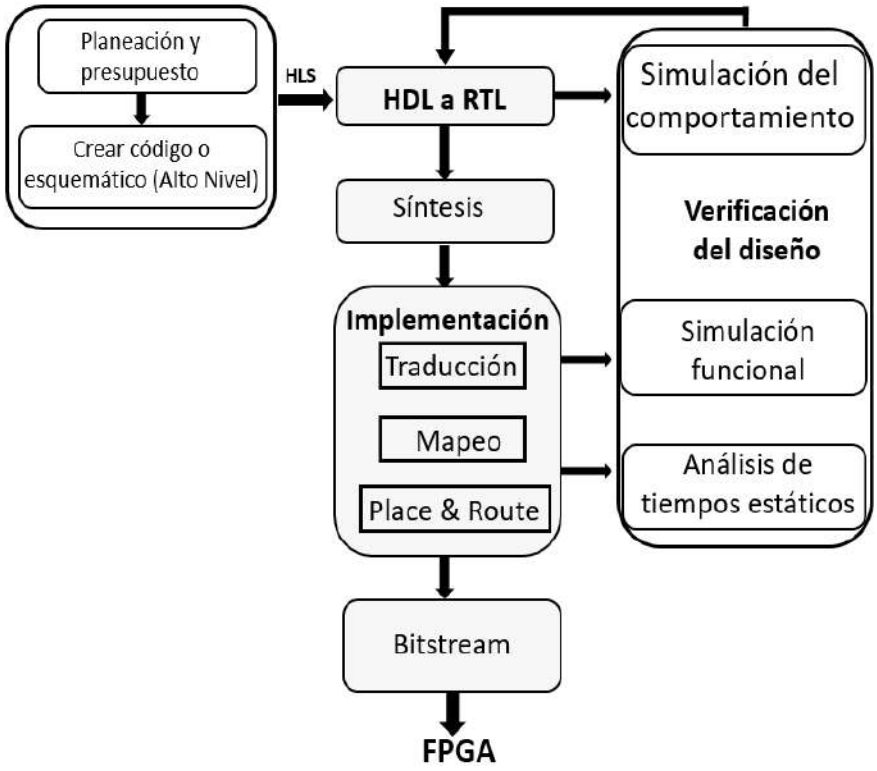

Figura 2.20. Flujo de diseño para FPGA.

1) HDL a RTL. Una vez que se hace la planeación y presupuesto del proyecto, se hace la codificación de la implementación. Esta codificación se puede hacer en un lenguaje de alto nivel o bien con un esquemático, y mediante un software *de* síntesis del alto nivel (HLS) se hace una traducción a un lenguaje de descripción de hardware (HDL), que es el código aceptado por un entorno de desarrollo de FPGA. El entorno de desarrollo convierte el código HDL en otro archivo llamado nivel de transferencia de registro (RTL). El archivo resultante contiene la base de tiempos de todas las operaciones a implementar por el diseño y es el archivo que puede ser sintetizable.

Otra opción es hacer la codificación directamente en HDL. Los estándares aceptados por la IEEE para HDL son VHDL (*Very High-Level Description Language)* basado en el lenguaje de programación ADA, y verilog que está basado en C. Estos son los lenguajes más comúnmente usados, aunque existen otros menos conocidos como ABEL (Advanced Boolean Expression Language) y otros propios de los fabricantes, como AHDL™ (Altera Hardware Description Language) y Active-HDL™ por ejemplo. La elección anterior depende del diseñador y su experiencia.

2) Síntesis. En la síntesis se traduce el código HDL a un formato netlist, que es una descripción de los elementos lógicos que contiene el circuito a implementar. Si el diseño contiene más de un sub-diseño, por ejemplo, para implementar un procesador se requiere de un CPU como un elemento individual, una RAM como otro elemento, y así sucesivamente. El proceso de síntesis genera un netlist por cada elemento. En este proceso se verifica la sintaxis del código y analiza la jerarquía del diseño, con lo que se asegura que el mismo sea optimizado para la arquitectura que el diseñador implementó. El resultado de todo esto es otro archivo NGC (Native Generic Circuit).

3) Implementación. Él proceso de implementación se divide en tres partes: la traducción, mapeo y el place and route. El proceso de traducción combina el netlist y las restricciones en un archivo de diseño lógico. Esta información se almacena en un archivo NGD (Native Generic Database). La definición de restricciones es asignar los puertos del diseño a los elementos físicos, tales como pines, switches, botones etc, del dispositivo sobre el cuál se implementará el diseño (FPGA en particular), especificando los requerimientos de tiempo del diseño. Esta información se guarda en un archivo UCF (User Constraints File). El proceso de mapeo divide el circuito lógico completo en sub-bloques, de forma que se ajusten a los bloques que en particular tiene el dispositivo sobre el que se implementará el diseño (CLBs, IOBs, etc). La salida de esta etapa es un archivo NCD (Native Circuit Description), que físicamente representa el diseño mapeado a los componentes del FPGA. El proceso

de place and route hace la asignación física de los elementos del diseño en el dispositivo.

4) Bitstream. Antes de hacer la configuración del FPGA, el diseño se debe convertir a un formato que pueda aceptar el dispositivo. El archivo NCD de entrada a este proceso es convertido a un archivo tipo .bit, que es el que contiene la información necesaria para configurar el dispositivo.

Como se observa a la derecha de la figura 2.20, de manera opcional se puede hacer una simulación de los procesos anteriores, esto con el propósito de verificar el comportamiento que tendrá el circuito, previo a la implementación física. El análisis de tiempos estáticos se puede hacer luego del mapeo y también luego del proceso place and route. Luego del proceso de mapeo, se obtiene un reporte sobre los retardos que hay en las trayectorias del diseño lógico, mientras que luego del place and route se obtiene la suma de todos los tiempos de retardo.

Existe software dedicado a realizar cada uno de los procesos anteriores, pero no sería adecuado tenerlos por separado, de forma que se integran en uno solo, como el caso de Vivado™ o ISE™ en el caso de Xilinx® o Quartus™ para Intel®. Existen varios softwares de uso libre como: Icarus Verylog, GPL Cver, LIFTING, VBS y otros. Claro que para implementar un diseño en un FPGA no se tendrán las herramientas necesarias, ya que depende del fabricante. Pero funcionan bien para el caso de simulaciones y como ambiente académico.

Capítulo 3 Redes Neuronales Convolucionales en FPGA

3.1 Trabajos relacionados

Aunque desde su aparición los FPGA se han empleado en infinidad de aplicaciones, para el caso de Redes Neuronales Convolucionales se comenzaron a usar al final de la década pasada (Farabet, Poulet, & Han, 2009). La principal razón por la que estos dispositivos anteriormente no se emplearan en este tipo de aplicaciones, es que se requiere de un conocimiento en hardware, mientras que el desarrollo se ha llevado a cabo principalmente en software. Recientemente, entre las herramientas para prototipado basado en FPGA se han adoptado modelos de programación a nivel de software, como OpenCL y herramientas de desarrollo de los propios fabricantes, lo que los ha hecho más atractivos para desarrolladores de software. Es así como recientemente se han hecho aportaciones en el paradigma de Deep Learning con FPGAs.

Muchas empresas y centros de investigación han enfocado parte de sus esfuerzos en este campo de la Inteligencia Artificial. En 2010 Microsoft® anuncio su proyecto “Catapult” (Microsoft, 2011), en el que el objetivo fue la investigación en “*cómo utilizar circuitos integrados personalizados y programables para acelerar las operaciones computacionalmente caras en su Datacenter*”. Su solución, sobre la que aún se sigue investigando fueron los FPGAs, en (Ovtcharov, y otros, 2015) muestran parte su investigación, en la parte de acelerar el cómputo requerido en las CNN empleadas en su Datacenter. En (Qiao, y otros, 2016) presentan resultados sobre la aceleración de CNN en la tarea de clasificación de imágenes, alcanzando un flujo de salida de 77.8 GFLOPS y un ahorro de energía de 4.7 veces en comparación con un GPU.

En (Qiu, y otros, 2016) implementan la arquitectura VGG16-SVD (una CNN profunda) en un SoC, aunque alcanzan apenas una tasa de 4.45 fps y una exactitud de 86.66% en la clasificación de imágenes en el ImageNet Challenge, (bajo para la exactitud del último concurso de 96.6 %). Otro caso en el que se tiene una situación

parecida es en (Zhang, y otros, 2015), en el que también utilizan un FPGA para acelerar el procesamiento de un CNN profunda alcanzando un flujo de salida de 61.62 GFLOPS. Xilinx® es uno de los principales fabricantes de FPGAs, los siguientes conceptos son extraídos de (Xilinx, 2013), aunque los mismos conceptos aplican para dispositivos de otros fabricantes.

3.2 Paralelismo en FPGA vs CPU

Como se ha mencionado anteriormente, un punto importante a tener en cuenta al entrenar o implementar redes neuronales convolucionales (o cualquier aplicación que demande capacidad de procesamiento de forma eficiente) es el hardware sobre el cuál se ejecutarán los algoritmos. La ventaja de un CPU es su facilidad de programación al usar un lenguaje de alto nivel para el desarrollo de la aplicación y dejar la traducción a código ejecutable al compilador, aunque eso no sea lo más adecuado si se habla de eficiencia. Aún y cuando las velocidades con las que trabajan los CPU's superan por mucho a las de los FPGA's (más de 3.6 GHz contra 500 MHz en los mejores casos), los FPGA ejecutan instrucciones de forma más eficaz. Por ver un ejemplo de esto, para llevar a cabo una simple instrucción de suma de dos operandos como:

$$P = A + B; \ (a) \qquad ADD \ AX, BX; \quad (b) \tag{3.1}$$

En la ecuación 3.1 (a) indica la suma de los operandos A y B en algún lenguaje de alto nivel, mientras que la ecuación 3.1 (b) indica su representación en ensamblador. Sin embargo, operaciones tan simples como esa requieren de varios ciclos de reloj para llevarla a cabo, ya que para que funcione, el compilador debe generar el código extra necesario para que se lleve a cabo la operación en el microprocesador. En ensamblador, el código sería algo como:

```
MOV     AX,   A;
MOV     BX,   B;
ADD     AX,   BX;
MOV     P,    AX;
```

Como se observa, una simple operación en realidad implica que el procesador ejecute varias instrucciones para llevarla a cabo, lo que implica consumo de tiempo al ejecutar una instrucción en cada ciclo de reloj. Según el tipo de operación, el tiempo de latencia de cada una de ellas cambia. Del ejemplo anterior, dependiendo de la ubicación de los operandos A y B, la cantidad de ciclos de reloj para hacerlo cambia. Si los operandos se encuentran en la cache del microprocesador, la cantidad de ciclos será la mínima, pero si estos operandos se encuentran la memoria DDR, le llevará unos cientos de ciclos de reloj hacerlo, si se encuentran en el disco duro o alguna unidad de almacenamiento, esta cantidad crece drásticamente.

Al igual que un microprocesador, un FPGA puede implementar cualquier operación lógica o aritmética. La principal diferencia es que el HLS (compilador del FPGA) transforma las descripciones del software en RTL (nivel de transferencia de registro), este no se ve obstaculizado por restricciones de una cache o un espacio de memoria unificada. Para calcular el valor de P en el ejemplo anterior, se requiere de una cantidad de LUT's igual a la cantidad de bits necesarios para representar P, (una LUT por bit). Las LUT's utilizadas para el cálculo de P sólo pertenecen a esta operación. A diferencia de un procesador, en el que todos los cálculos comparten la misma ALU, una implementación en FPGA crea instancias independientes de LUT para cada cálculo. Además de asignar recursos únicos de LUT por cálculo, el FPGA difiere de un procesador en la arquitectura de la memoria y el costo de acceso a la misma.

El compilador HLS organiza la memoria en múltiples bancos de almacenamiento, tan cerca como sea posible del punto que se requieren para la operación. Esto da como resultado un ancho de banda de memoria instantáneo, que en ese sentido supera las capacidades de un procesador. El compilador HLS despliega las capacidades del dispositivo a través de los procesos de programación, canalización y flujo de datos. Aunque son transparentes para el usuario, estos procesos son etapas integrales de la compilación del programa, que extrae la mejor

implementación posible a nivel de circuito de la aplicación. A continuación se presentan las diferencias entre usar un FPGA y un CPU para cómputo eficiente.

3.2.1 Ventajas del FPGA para cómputo eficiente

1) Programación. Es el proceso de identificación de las dependencias de datos y control entre diferentes operaciones, para determinar cuándo se ejecutará cada una. En el diseño FPGA tradicional, este es un proceso manual que también se conoce como paralelización del algoritmo de software, para una implementación en hardware. HLS analiza las dependencias entre operaciones adyacentes, así como a través del tiempo. Esto le permite al compilador agrupar operaciones en el mismo ciclo de reloj, y configurar el hardware, para permitir la superposición de llamadas a funciones. La superposición de ejecuciones de llamadas de función elimina la restricción del procesador, que requiere que la llamada a la función actual se cumpla por completo, antes de que pueda comenzar la siguiente llamada de función al mismo conjunto de operaciones.

2) **Flujo de datos.** En un FPGA, el flujo de datos tiene una naturaleza paralela. Para lograr esto, el HLS extrae este nivel de paralelismo al evaluar las interacciones entre las diferentes funciones de un programa en función de sus entradas y salidas. El caso más simple de paralelismo es cuando las funciones interactúan con diferentes conjuntos de datos y no se comunican entre sí. En este caso, el HLS asigna recursos de lógica del FPGA para cada función y luego ejecuta los bloques de forma independiente. El caso más complejo, que es típico en los programas de software, es cuando una función proporciona resultados para otra función. Este caso se conoce como el escenario consumidor-productor. HLS admite dos modelos de uso para este escenario.

En el primer modelo de uso, el productor crea un conjunto completo de datos antes de que el consumidor pueda comenzar su operación. El paralelismo se logra instanciando un par de memorias BRAM dispuestas como bancos de memoria ping y pong. Cada función puede acceder solo a un banco de memoria, ping o pong,

durante la duración de una llamada de función. Cuando comienza una nueva llamada de función, el circuito generado por HLS conmuta las conexiones de memoria tanto para el productor como para el consumidor. Este enfoque garantiza la corrección funcional, pero limita el nivel de paralelismo alcanzable para llamadas a través de funciones.

En el segundo modelo, el consumidor puede comenzar a trabajar con resultados parciales del productor, y el nivel de paralelismo alcanzable se extiende, para incluir la ejecución dentro de una llamada a función. Los módulos generados por HLS para ambas funciones se conectan mediante el uso de un circuito de memoria primero en entrar, primero en salir (FIFO). Este circuito de memoria, que actúa como una cola en la programación de software, proporciona sincronización a nivel de datos entre los módulos. En cualquier punto durante una llamada de función, ambos módulos de hardware están ejecutando su programación. La única excepción es que el módulo del consumidor espera a que haya algunos datos disponibles del productor, antes de comenzar el cálculo. En la terminología de HLS, el tiempo de espera del módulo del consumidor se conoce como intervalo de iniciación.

3) Frecuencia de reloj. Es uno de los primeros elementos a considerar al determinar la plataforma de ejecución de un algoritmo específico. Una pauta comúnmente utilizada es que una alta frecuencia de reloj se traduce en una tasa de ejecución de alto rendimiento de un algoritmo. Aunque esta podría ser una buena regla de primer orden para elegir entre procesadores, en realidad es engañosa y puede llevar al diseñador a tomar la decisión equivocada al seleccionar entre un procesador y un FPGA. La razón de esto se relaciona con la diferencia nominal en la frecuencia del reloj entre un procesador y un FPGA, 3.6 GHz o más, contra 500 MHz, respectivamente. Un procesador convencional ejecuta las instrucciones una a la vez, por cada una de ellas, por lo regular sigue el ciclo de instrucción (Fetch-Decode-Execute) indicado en la figura 3.1. En el ciclo indicado en la figura, primero se primero se obtiene la instrucción de la memoria de programa (IF), se decodifica (ID), se ejecuta (EXE), se obtiene la siguiente instrucción usando operaciones de

memoria (MEM) y finalmente se escriben los resultados de la operación en un registro local o en la memoria global.

Figura 3.1. Ciclo de instrucción en un procesador, fuente (Xilinx, 2013).

La mayoría de los procesadores modernos incluyen múltiples copias de la ruta de ejecución de la instrucción y son capaces de ejecutar instrucciones con cierto grado de solapamiento. Ya que las instrucciones por lo general tienen dependencia, la superposición entre las copias del hardware de ejecución de la instrucción no es perfecta. En el mejor de los casos, solo las etapas generales introducidas al usar un procesador pueden superponerse. Las etapas EXE, que es el tiempo en que se hace un cálculo, se ejecutan secuencialmente. Las razones para esta ejecución secuencial están relacionadas con recursos limitados en la etapa EXE y dependencia entre instrucciones. La figura 3.2 muestra la forma en que se ejecutaría instrucciones en semi-paralelo. Este es el mejor caso para un procesador multinúcleo, en el que todas las instrucciones se ejecutan lo más rápido posible. Incluso en este mejor caso, el procesador está limitado a solo una etapa EXE por ciclo de reloj.

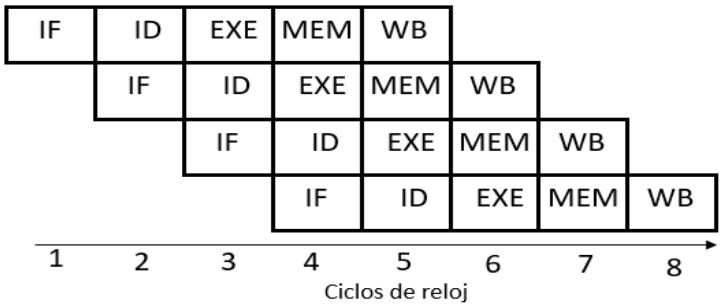

Figura 3.2. Ejecución de instrucciones en semi-paralelo, fuente (Xilinx, 2013).

Esto significa que la aplicación del usuario avanza en una operación por ciclo de reloj. Incluso si el compilador determinara que las cinco etapas EXE podrían ejecutarse en paralelo, la estructura del proceso lo evitaría.

Por el contrario, un FPGA no ejecuta todo el procesamiento en una sola plataforma de cálculo común, si no que ejecuta un solo programa a la vez, en un circuito personalizado para ese programa. Si se cambiara la aplicación del usuario, cambiaría el circuito en el FPGA. El equivalente a la figura 3.2 en un FPGA sería como lo que se muestra en la figura 3.3. La presencia de la etapa MEM depende de la aplicación.

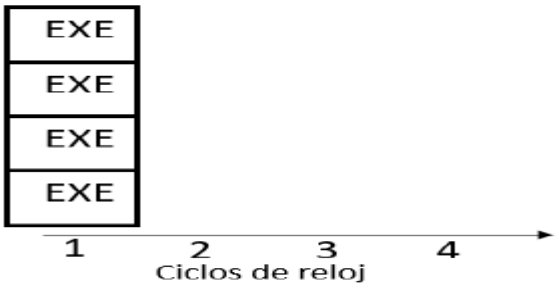

Figura 3.3. Múltiples unidades de ejecución en un FPGA, fuente (Xilinx, 2013).

Comparando las figuras 3.2 y 3.3, se nota que el FPGA tiene una clara ventaja de rendimiento nominal de 9x (Xilinx, 2013), en comparación con el procesador. Los números reales de esto dependen de la aplicación, pero los FPGA generalmente muestran al menos 10 veces el rendimiento de cómputo de un procesador, para aplicaciones computacionalmente intensivas.

4) Consumo de potencia. Es otra característica que no se percibe al enfocarse solo en la frecuencia del reloj, el consumo de potencia dinámica de un dispositivo se aproxima con la ecuación 3.1:

$$P = \frac{cF\alpha V^2}{2} \qquad (3.1)$$

Donde P es la potencia dinámica en watts, c es la capacitancia producida por los componentes del dispositivo en faradios, F es la frecuencia de operación en Hz

del procesador, α un factor que depende de la actividad del dispositivo y V es el voltaje de alimentación en volts. Como se muestra en la ecuación 3.1, el consumo de potencia dinámica es directamente proporcional a la frecuencia del reloj con la que trabaja el dispositivo. Considerando los otros factores constantes, esto indica un mayor consumo de potencia en un CPU que un FPGA para la misma carga de trabajo computacional. Creando un circuito personalizado por el usuario, un FPGA puede ejecutarse a una frecuencia de reloj más baja con el máximo paralelismo entre las operaciones y sin la sobrecarga de interpretación de instrucciones que se encuentra en un procesador.

5) Latencia y pipelining. La latencia se define como el número de ciclos de reloj que le lleva a un dispositivo completar una instrucción o un conjunto de instrucciones y generar un resultado en una aplicación. Por ejemplo, en la figura 3.1, la latencia sería de cinco ciclos para una instrucción en un procesador convencional. Si una operación tuviese cuatro instrucciones, se requeriría de 20 ciclos de reloj para generar el resultado.

EL motivo para usar pipelining es mejorar el desempeño temporal de una aplicación en cuanto a la velocidad de procesamiento. Esta técnica consiste en que cada parte de un sistema se encargue de llevar a cabo sólo un tipo de tarea, y que entre varias secciones se lleve a cabo una instrucción completa (ciclo completo de instrucción). Por ejemplo, en la figura 3.1, para una instrucción, a un procesador le toma cinco ciclos de reloj, en la figura 3.2 se representa la misma situación empleando pipelining en el procesador y como se ve, cuatro instrucciones se ejecutan en 8 ciclos de reloj, y no en 20. Para el caso del FPGA, en la figra 3.3 se ve que las mismas cuatro instrucciones las lleva a cabo en un ciclo de reloj, con lo que se reduce significativamente el tiempo de latencia.

6) Memoria. La arquitectura de memoria disponible en la plataforma de implementación seleccionada, es uno de los elementos físicos que pueden afectar el rendimiento de una aplicación de software. Esta arquitectura determina el límite superior en el rendimiento alcanzable. En algún punto del desarrollo, todas las

aplicaciones (ya sea en un procesador o en un FPGA), se vuelven vinculadas a la memoria, independientemente del tipo y la cantidad de recursos computacionales disponibles. Una estrategia en el diseño con FPGAs, es comprender dónde se encuentra el límite de la memoria, y cómo puede verse afectado por el tipo y organización de los datos, además de la organización de la misma. En un sistema basado en procesador, el ingeniero de software debe ajustar la aplicación, tomando en cuenta la arquitectura de la memoria con la que se cuenta, independientemente del tipo específico de procesador. Esta característica común simplifica el proceso de migración de la aplicación, a costas del rendimiento.

Tabla 3.1. Clasificación de memoria, según la cantidad de ciclos necesarios para extraer y escribir datos.

Categoría	Ejemplo
Lenta	Dispositivos de almacenamiento masivo, como discos duros, unidades de almacenamiento
Velocidad media	Memorias DDR
Rápida	Memorias de tipo caché, en el interior del propio chip. Diferentes tamaños según el procesador específico.

Se puede hacer una clasificación de la memoria en función de ciclos de reloj necesarios para transferir los datos al y del procesador como memorias lentas, medianas o rápidas. Esta clasificación se muestra en la tabla 3.1. La ubicación física de los datos y cómo se mueven entre los diferentes niveles en la jerarquía, es manejada por la plataforma de cómputo, y es transparente para el usuario. En este tipo de sistema, la única forma de aumentar el rendimiento es reutilizar los datos en la memoria caché tanto como sea posible. Para lograr este objetivo, el ingeniero de software debe pasar grandes cantidades de tiempo mirando las huellas de caché, reestructurando el algoritmo de software para hacer eficiente la ubicación de los datos y administrando la asignación de la memoria, con el fin de minimizar su uso y en consecuencia reducir el tiempo de procesamiento. La ventaja de la asignación

de memoria estática, es que puede ser implementada de diferentes maneras para cada necesidad. Según el cálculo necesario en el algoritmo, el compilador de HLS puede implementar la los recursos de memoria como registros, desplazamiento de registros, FIFO (First In First Out) o BRAM.

7) Registros. La implementación de una memoria con registros, es la estructura de memoria más rápida posible. En este estilo de implementación, cada entrada de un elemento (una matriz, por ejemplo), se convierte en una entidad independiente del mismo elemento. Cada entidad independiente está integrada en el cálculo donde se use, sin la necesidad de abordar la lógica o retrasos adicionales para tener acceso a los datos de una memoria externa.

8) FIFO. La estructura FIFO se puede ver como una cola con un único punto de entrada y un único punto de salida. Este tipo de estructura se usa generalmente para transmitir datos entre ciclos de programas o funciones. No hay lógica de direccionamiento involucrada, y los detalles de implementación son completamente manejados por el compilador de HLS.

9) BRAM. Un BRAM (Block Random Access Memory) es una memoria de acceso aleatorio que está integrada en la rejilla del FPGA, la cantidad exacta de bloques de memoria es específica del dispositivo. En términos de programación en un procesador, este tipo de memoria se puede considerar como una caché con las siguientes limitaciones:

1. No implementa la coherencia de caché, la colisión y la lógica de seguimiento de fallas de caché que normalmente se encuentran en una caché de un procesador convencional.
2. Mantiene sus valores solo mientras el dispositivo esté encendido.
3. Admite el mismo acceso de ciclo paralelo a dos ubicaciones de memoria diferentes.

Una de las diferencias más significativas entre un FPGA y un CPU es que en el primero, la memoria está disponible en el interior de cada CLB, con lo que se

elimina la restricción del tiempo necesario para acceder a los recursos de la memoria, como en el caso del CPU.

10) Compilador. El HLS no se está limitado por una plataforma de procesamiento fija y crea una plataforma específica para cada algoritmo, basada en la entrada del usuario. Esto permite que un diseñador maximice el rendimiento de la aplicación, en términos de cantidad de operaciones por segundo, latencia y potencia. La figura 3.4 muestra la forma en que se llevaría a cabo el siguiente código de tres instrucciones, en un procesador convencional y en un FPGA.

```
A[i]  =  B[i] * C[i];
D[i]  =  B[i] * E[i];
F[i]  =  A[i] + D[i];
```

En la figura 3.4 sólo se indica la sección de *ejecución* del ciclo Fetch-Decode-Execute de un procesador convencional. Aun así, la implementación de HLS supera a la ejecución del procesador convencional, debido a la arquitectura de memoria personalizada creada para el algoritmo. En el procesador, las matrices A, B, C, D, E y F se almacenan en un solo espacio de memoria y solo se puede acceder a ellas una a la vez. En cambio, el compilador HLS detecta estas memorias y crea un banco de memoria independiente para cada matriz, lo que resulta en una superposición entre las operaciones de lectura de la matriz B y la matriz C. La ejecución de la operación de lectura de la matriz E en el ciclo de reloj 2 muestra una de las optimizaciones automáticas de recursos de HLS.

Para las operaciones de memoria, el compilador analiza los bancos que contienen los datos y dónde se toma el valor durante el cálculo. Aunque la lectura de la matriz E pudiera ocurrir durante el primer ciclo de reloj, el HLS coloca automáticamente la operación de memoria lo más cerca posible de la ubicación donde se necesitan los datos, para reducir la cantidad de almacenamiento temporal de datos en el circuito. Debido a que el multiplicador que usa el valor de E no se

ejecuta hasta el ciclo de reloj 3, no hay beneficio en programar el acceso de lectura para que ocurra antes que el ciclo de reloj 2.

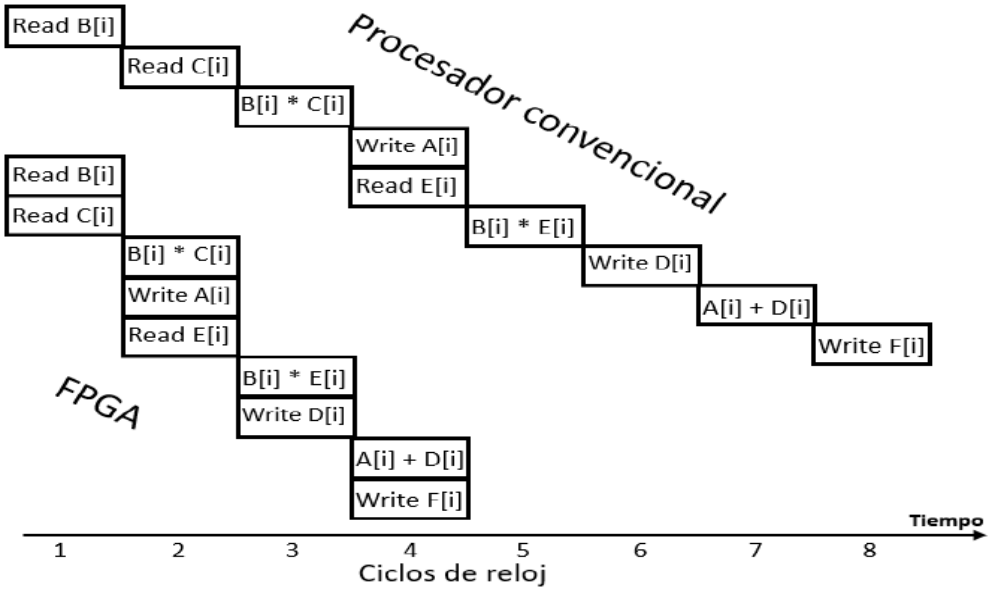

Figura 3.4. Ejecución de instrucciones en un FPGA y un procesador.

Otra forma en que HLS ayuda al usuario a controlar el tamaño del circuito generado, es proporcionando los tipos de datos para el tamaño de las variables. De forma similar a otros compiladores, HLS ofrece al usuario acceso a tipos de datos enteros, de simple precisión y de doble precisión. Esto permite una migración rápida del software al FPGA, pero podría enmascarar las ineficiencias de los algoritmos, que son el resultado de las rutas de datos de 32 y 64 bits disponibles en los procesadores. Por ejemplo, suponiendo que el código mostrado solo requiere valores de 20 bits en las matrices B, C y E, en el código del procesador original, estos tamaños de bits requerirían que las matrices A, D y F sean capaces de almacenar valores de 64 bits para evitar cualquier pérdida de precisión. HLS puede compilar el código como este, pero esto da como resultado una ruta de datos de 64 bits ineficaz que consume más recursos de los que requiere el algoritmo.

11) Sentencias condicionales y ciclos. En un compilador de procesador convencional, las sentencias condicionales se traducen en un conjunto de operaciones que pueden dar o no como resultado un cambio del contexto. Esto

afecta el desempeño al emplear pipelining, al introducir una dependencia que afecta la instrucción que se busca a continuación en la memoria. En un FPGA, un enunciado condicional no tiene el mismo impacto potencial en el rendimiento que en un procesador convencional. El HLS crea todos los circuitos descritos por cada rama del enunciado condicional. Por lo tanto, la ejecución de una declaración de software condicional implica la selección entre dos resultados posibles, en lugar de un cambio de contexto.

Los ciclos son una construcción de programación común para expresar el cálculo iterativo. HLS soporta completamente los ciclos, e incluso puede hacer transformaciones que están más allá de las capacidades de un compilador de procesador estándar. Por ejemplo, para el siguiente ciclo:

```
for  i=0; i<10; i++)
{
        A = A + (B[i] * C[i]);
}
```

Suponiendo que cada iteración del ciclo se ejecuta en cuatro ciclos de reloj, en un procesador convencional, el compilador programa cada iteración de forma secuencial, para un total de 40 ciclos de reloj necesarios para terminar el ciclo del código anterior. El HLS no tiene esta limitante, debido a que crea el hardware para el algoritmo, puede alterar el perfil de ejecución de un ciclo repartiendo iteraciones. Un ciclo con pipelining extiende el concepto de paralelización en una iteración del ciclo a todas las iteraciones. La figura 3.5 muestra el concepto.

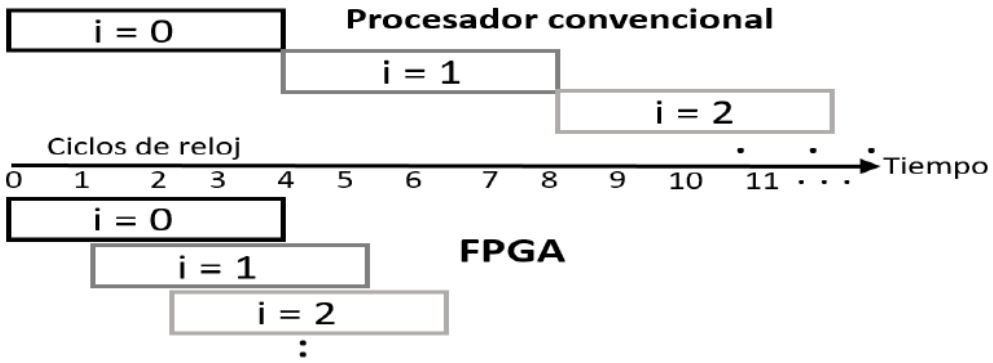

Figura 3.5. Ejecución de ciclos en un procesador y un FPGA.

Para reducir el tiempo de latencia en cada iteración, la primera optimización automática aplicada por HLS, es la paralelización del operador en la iteración del ciclo. La segunda optimización es el pipelining de la iteración en el ciclo. Esta optimización afecta el consumo de recursos y las tasas de los datos de entrada de la implementación en el FPGA.

El comportamiento por default es ejecutar ciclos en el mismo tiempo que un procesador, como se muestra en la figura 3.5. Esto significa que el código implementado tiene una latencia de procesamiento de 40 ciclos y una velocidad de datos de entrada de una vez cada 4 ciclos. En este ejemplo, la velocidad de datos de entrada se define por la rapidez con la que se pueden muestrear los valores de B y C desde la entrada. HLS puede paralelizar o canalizar (pipeline) las iteraciones de un ciclo para reducir la latencia en el cálculo, y aumentar la tasa de datos de entrada. El efecto del pipeline de ciclos en las características de ejecución se resume en la tabla 3.2.

Tabla 3.2. Perfil de ejecución de ciclos en diferentes compiladores.

Compilador.	Latencia en la ejecución del ciclo. (Ciclos de reloj)	Tasa en los datos de entrada.
Procesador convencional	40	Cada 4 ciclos
HLS por default	40	Cada 4 ciclos
HLS con pipeline	14	Cada ciclo

3.3 Arquitecturas para procesamiento eficiente de CNNs.

Como se expone en el capítulo 2, la parte crítica para tener aplicaciones exitosas con CNNs es procesar altos volúmenes de datos, con los que se llevan a cabo operaciones de convolución principalmente. En la figura 2.10 vemos que básicamente se requiere una forma de controlar el desplazamiento de los filtros a lo largo de las imágenes (el paso del filtro S), a la vez que se van haciendo las operaciones de productos y sumas, entre los datos de cada capa del volumen de entrada y los parámetros de los filtros. Claro que además hay que implementar las

funciones de activación, por cada capa del volumen de entrada. La otra operación fundamental que se requiere, son los productos punto para las capas completamente conectadas. Lo que se busca, es una forma eficiente de hacer ese procesamiento, y dado que el algoritmo es inherentemente paralelo, se pueden aprovechar las ventajas que ofrecen los FPGAs. En (Ovtcharov, y otros, 2015), exponen la arquitectura empleada en el proyecto de Microsoft (Microsoft, 2011) implementada para este propósito, esta arquitectura se muestra en la figura 3.6.

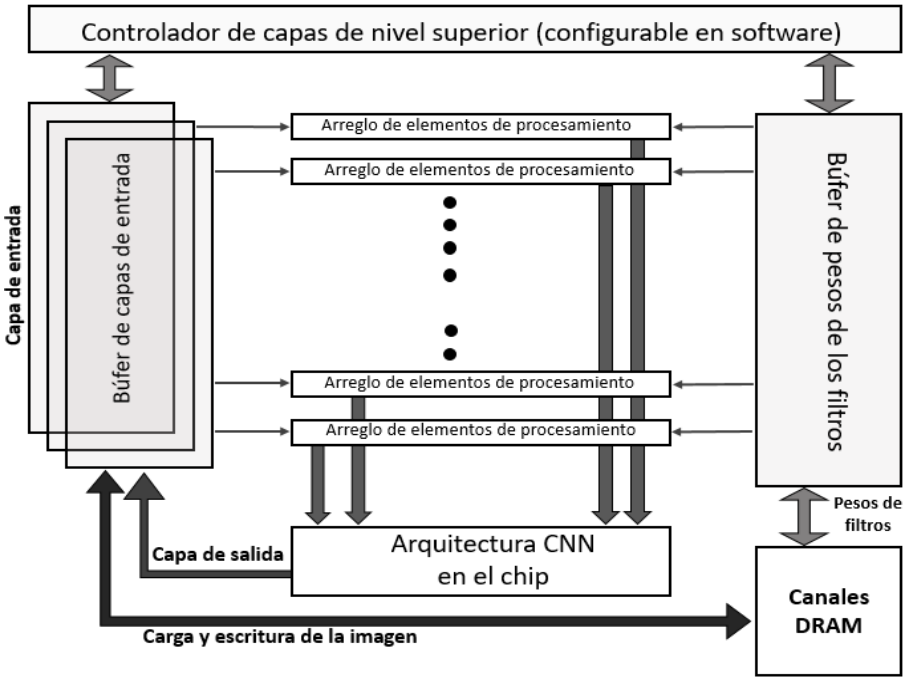

Figura 3.6. Arquitectura para acelerar el cómputo de capas en una CNN.

El principio de la arquitectura indicada en la figura 3.6 es un motor configurable por software que pueda manejar la configuración de las múltiples capas en tiempo de ejecución (sin que sea necesaria la reconfiguración de hardware). Un esquema eficiente de transporte de datos, junto con una red de redistribución on-chip que minimice el tráfico fuera de la memoria. Arreglos para procesamiento distribuidos espacialmente, donde la cantidad de estos es variable.

En una operación normal, la CNN acepta una imagen y procesa las diferentes capas de convolución. Para la capa inicial, los pixeles de la imagen de entrada son

llevados al chip desde la memoria DRAM local y luego al búfer de las múltiples capas de entrada. Estos datos son llevados a los múltiples arreglos de procesamiento, donde se implementan las operaciones de convolución en tres dimensiones. El controlador superior organiza, direcciona y entrega los datos a cada arreglo de procesamiento. Finalmente, los resultados acumulados son enviados al chip que contiene la arquitectura de la red, esta se encarga de recircular los datos obtenidos en esta capa, para repetir el procedimiento con la siguiente capa de datos.

La configuración anterior fue pensada para los servidores empleados en los centros de datos de Microsoft, cuentan tarjetas para aceleramiento del cómputo y FPGAs Stratix V D5 y 8GB de memoria DDR3 incorporados en las mismas. Esta arquitectura se probó en el ImageNet Challenge, y CIFAR-10 implementando una CNN de las reportadas en el estado del arte, (Krizhevsky, Sutskever, & E. Hinton, 2012), además de cambiar el FPGA y ver el efecto. Los resultados reportados se muestran en la tabla 3.3 y se hace una comparación con un desarrollo para la misma aplicación, pero con GPUs. De la tabla se observa una notable ventaja de los FPGAs en el consumo de potencia en todos los casos, en comparación con los GPUs. La cantidad de imágenes procesadas por segundo por cada FPGA (tres primeras filas) es más que suficiente para una aplicación de video. Por otro lado, se observa que procesan una mayor cantidad de imágenes por unidad de watt.

Tabla 3.3. Comparación de la arquitectura implementada con otros sistemas, (Ovtcharov, y otros, 2015).

	CIFAR-10	ImageNet 1K	ImageNet 22K	Potencia
Catapult Server + Stratix V D5	2318 im/s	134 im/seg	91 images/seg	25 W
Catapult Server + Arria 10 GX1150	X	233 im/seg	158 im/seg	25 W
Best prior CNN on Virtex 7 485T	X	46 im/seg	X	X
Caffe+cuDNN on Tesla K20	X	376 im/seg	X	235 W
Caffe+cuDNN on Tesla K40	X	500-824 im/seg	X	235 W

En muchos trabajos se han enfocado en buscar alternativas para hacer más eficiente el cómputo de las capas de convolución, sin afectar el rendimiento de la red, tales como la Transformada Rápida de Fourier (FFT) en (Mathieu, Henaff, & LeCun, s.f.), (Dubout & Fleuret, 2012) y el algoritmo Winograd (Lavin & Gray, 2016). Sin embargo los requerimientos computacionales no se han bajado lo suficiente y en cambio se han tenido otras desventajas con esos enfoques. En este trabajo interesa analizar la arquitectura más adecuada para la operación de convolución sin cambiar la estructura de una CNN.

Para el cómputo en paralelo de alto rendimiento se manejan dos paradigmas:

1) Arquitectura temporal (CPU y GPU)

2) Arquitectura espacial (FPGA y ASIC)

La primera es adecuada para operaciones de simple instrucción multiples datos (SIMD) y simple instrucción múltiples hilos (SIMT). En la arquitectura espacial se pueden aprovechar las características físicas de FPGAs, al tener recursos de memoria y control de forma local. La figura 3.7 muestra estos enfoques.

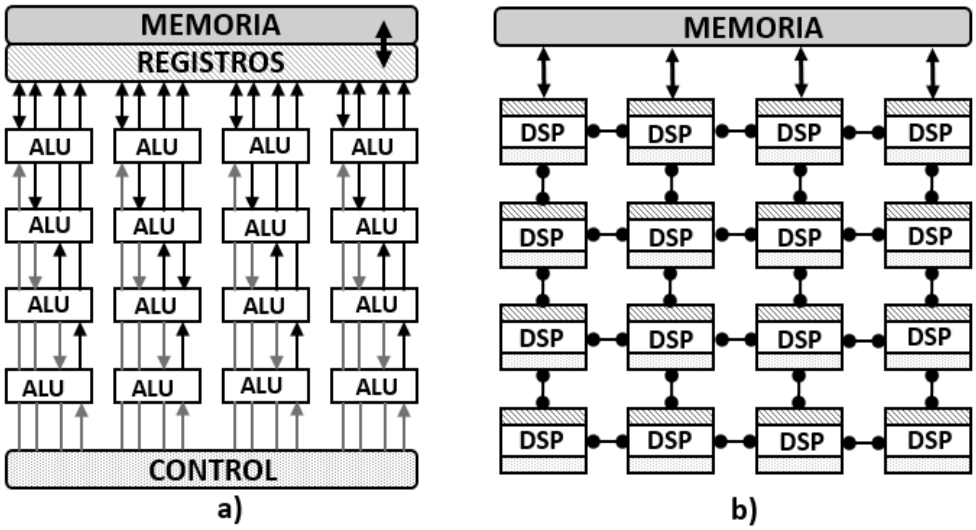

Figura 3.7. Paradigmas de cómputo en paralelo. a) Esquema adecuado para CPU y GPU. b) Adecuado para FPGA y ASIC. (Sze, Chen, Yang, & Emer, 2017).

En la operación de convolución básicamente se ejecutan operaciones MAC, esto es, multiplicar los pesos de los filtros con los mapas de activación y hacer sumas parciales de esos productos (en paralelo). Dado que el cuello de botella en este tipo de aplicaciones se origina en gran medida por el acceso a memoria, sería ideal reducir ese acceso en la medida de lo posible y hacer la mayor parte de operaciones localmente.

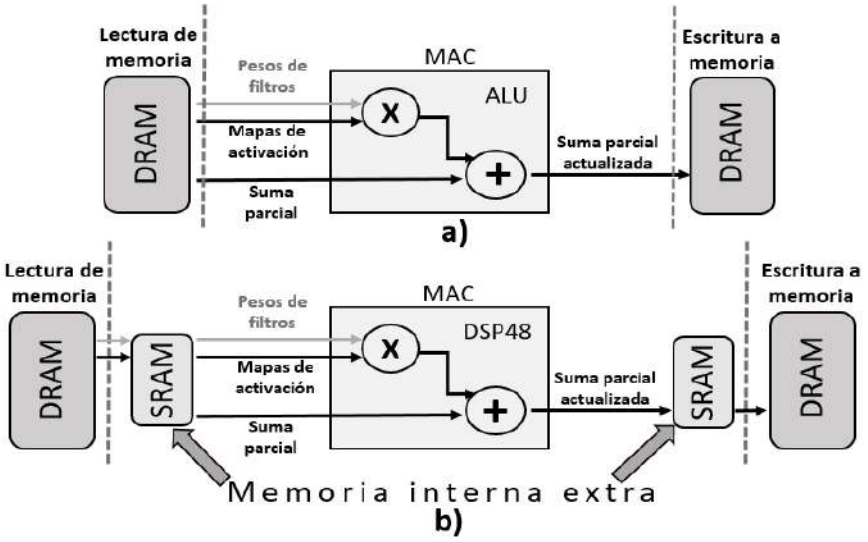

Figura 3.8. Arquitectura para reducir el acceso a memoria a) Hardware para la operación multiplicar-acumular, b) Implementación eficiente de MAC en un FPGA.

Un FPGA se ajusta a esa necesidad, dada la estructura de los CLBs (Figura 2.17) y los DSPs (figura 2.18). La idea se muestra en la figura 3.8, fuente (Sze, Chen, Yang, & Emer, 2017). En la parte superior de la figura se indica el hardware necesario para implementar la operación MAC. Se toman los datos de los pesos de los filtros, mapas de activación y la suma parcial, teniendo tres accesos a memoria. Mediante una ALU se llevan a cabo las operaciones actualizando la suma parcial, el resultado se escribe a memoria, teniendo otro acceso a memoria. Esto se hace por cada operación necesaria, lo que implica un incremento en el tiempo para generar resultados y también en la energía requerida.

Por otro lado, en la figura 3.7 b) se agrega memoria local extra (SRAM contenida en los CLBs del FPGA), con esta arquitectura se tienen tres beneficios

importantes: 1) Re-uso de datos, tanto de los pesos de los filtros como de los mapas de activación. 2) Acumulación local, ya que para la suma parcial no se requiere acceso a la memoria. En (Sze, Chen, Yang, & Emer, 2017) reportan una reducción de 2896 megas a 68 megas de accesos a memoria con este tipo de arquitectura. 3) La reducción en el consumo de energía, al estar presentes los datos localmente.

La arquitectura espacial para aceleramiento de CNNs de la figura 3.7 b) ofrecen la posibilidad de reducir el costo de energía del movimiento de datos al introducir varios niveles de jerarquía de memoria local con diferentes costos de energía como se muestra en la figura 3.9. Esto incluye un buffer global con un tamaño de varios cientos de kilobytes que se conecta a la DRAM, una red interna de elementos de procesamiento (PE) que puede pasar datos directamente entre las ALU y un archivo de registro (RF) dentro de cada elemento de procesamiento con un tamaño de algunos kilobytes o menos. Los múltiples niveles de jerarquía de memoria ayudan a mejorar la eficiencia energética al proporcionar acceso a datos de bajo costo. Por ejemplo, obtener los datos de la RF o de los PE vecinos tiene un costo energético de 200 veces menor que traerlos de la DRAM.

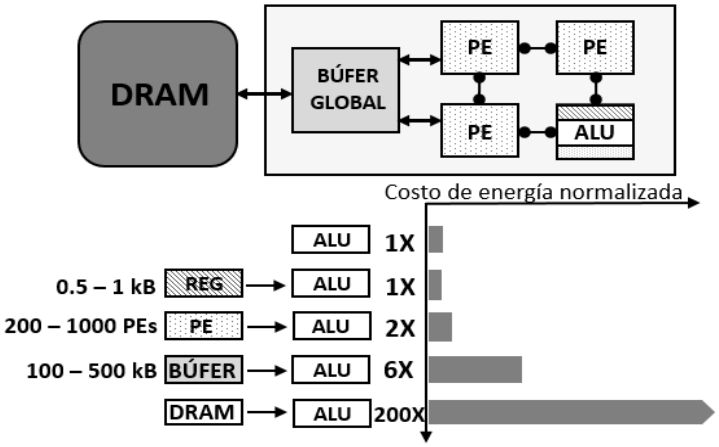

Figura 3.9. Costo de energía del acceso a memoria desde diferentes puntos para una arquitectura para aceleramiento de CNNs (Sze, Chen, Yang, & Emer, 2017).

Para poder explotar el re uso de datos y la acumulación local con una arquitectura de acceso a memoria de bajo costo energético se debe hacer un procesamiento con el flujo de datos adecuado. El tema de optimización del flujo de

datos se aborda brevemente en el siguiente apartado. Las ventajas en el paralelismo de los FPGA, así como las arquitecturas descritas indican que es viable el uso de estos dispositivos para el procesamiento de CNN en hardware embebido. Se puede adaptar la arquitectura descrita para nuestro propósito, tomando los parámetros de una red ya entrenada, de forma que lo que faltaría es la parte encargada de la adquisición de las imágenes, o video si fuera el caso. El sistema propuesto se muestra en la figura 3.10.

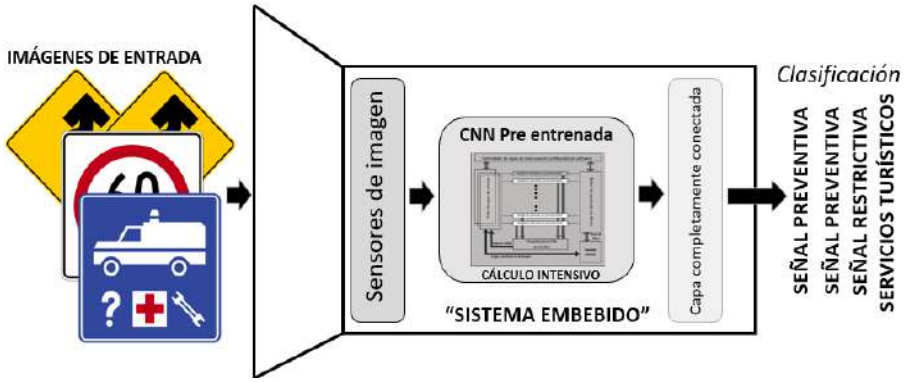

Figura 3.10. Sistema embebido para clasificación de imágenes de tránsito vehicular.

3.4 Optimización del flujo de datos

Para lograr el mejor desempeño en una aplicación, es necesario optimizar el sistema, en tantos aspectos como sea posible. El entorno SDAccel™ es un entorno de desarrollo de software completo para la creación, compilación y optimización de aplicaciones OpenCL™ que se acelerarán en un FPGA de Xilinx. A continuación se describen las recomendaciones de Xilinx para optimizar una aplicación basada en FPGA, en cuanto al flujo de datos, obtenido de (Xilinx, 2017).

En el modelo de programación de OpenCL™, primero todos los datos se transfieren de la memoria del host a la memoria global en el dispositivo, y luego de la memoria global al núcleo para su cálculo. Los resultados de cálculo se escriben desde el kernel a la memoria global y, por último, desde la memoria global a la memoria del host. La forma en que los datos pueden moverse de manera eficiente en este modelo de programación, es un factor clave para determinar las estrategias

de optimización del cómputo del kernel, por lo que se recomienda optimizar el movimiento de los datos en la aplicación, antes de tomar la optimización del cálculo.

Durante la optimización del flujo de datos, es importante aislar el código de transferencia de datos, del código de cálculo porque la ineficiencia en el cálculo puede causar paradas en el movimiento de datos. Xilinx recomienda que modifique el código de host y los núcleos, con el código de transferencia de datos, solo para este paso de optimización. El objetivo es maximizar el rendimiento de los datos del nivel del sistema, maximizando la utilización del ancho de banda PCIe, y la utilización del ancho de banda DDR. Por lo general, se requieren varias iteraciones de ejecución de emulación de CPU, emulación de hardware y ejecución en FPGA para lograr el objetivo.

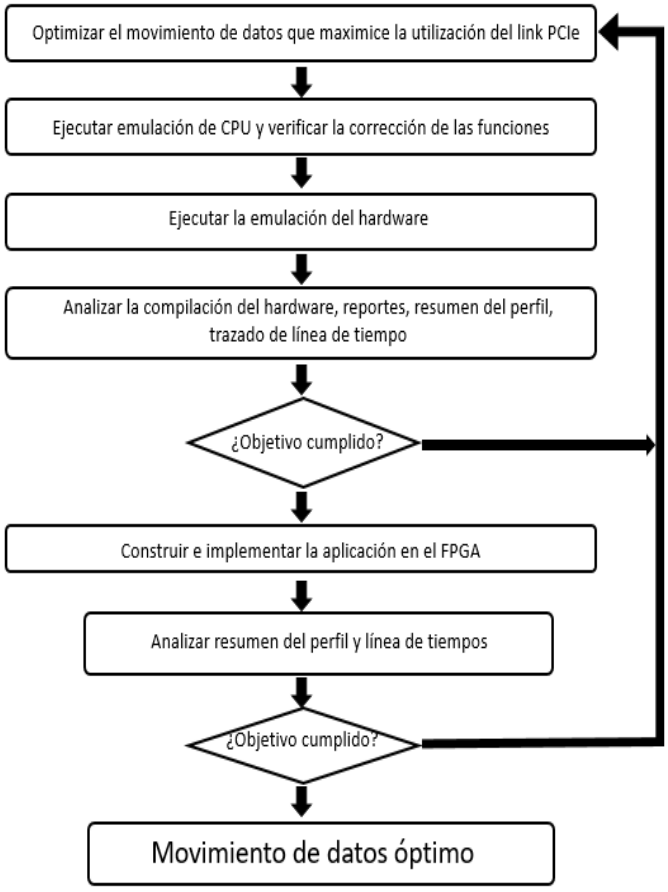

Figura 3.11. Optimización del flujo de datos

En el mismo documento se encuentra la guía para optimizar el rendimiento de funciones y el cómputo del kernel.

3.5 Opciones para la implementación de CNN en FPGA

A través de un análisis del funcionamiento de las CNN y de las características y ventajas de los FPGA, además de las herramientas de desarrollo, a continuación, se presentan las recomendaciones para la implementación de CNN en FPGAs. Son tres opciones: la primera se basa en OpenCL para paralelizar los procesos y etapas necesarias. La segunda se basa en el uso de un framework, que son herramientas que permiten la transferencia a un lenguaje HDL de arquitecturas de procesamiento. La otra opción es usar herramientas de terceros.

Sería muy complicado emplear un leguaje de descripción de hardware (HDL) para hacer implementaciones sofisticadas, como lo es la visión artificial. Es análogo a resolver un problema *"complicado"* usando ensamblador en un ambiente de software. La principal dificultad que se tiene en el caso de aplicaciones de visión artificial es la programación en paralelo, y hacer la traducción a HDL, que es lo que finalmente acepta el compilador HLS. Con la evolución que se está teniendo en este campo de la Inteligencia Artificial, se han creado diversas plataformas para este fin. Tres de las principales son:

1. Modelos basados en OpenCL
2. Frameworks para desarrollo basado en hardware
3. HDL Coder® de Matworks®

A continuación, se hace una breve descripción de estas opciones de implementación.

3.5.1 Modelo basado en OpenCL®

El estándar OpenCL® para programación paralela, ha sido desarrollado por el consorcio industrial del Grupo Khronos, para abordar los desafíos de la

programación de plataformas informáticas multi-núcleo y heterogéneas. La especificación OpenCL® define un modelo de programación único, y un conjunto de abstracciones a nivel de sistema que son compatibles con todas las plataformas de hardware que cumplen con el estándar. Esto significa que un ingeniero de software puede aprender un solo modelo de programación y usarlo directamente en dispositivos de múltiples proveedores. La figura 3.12 muestra la estructura general de una aplicación para FPGA basada en OpenCL®.

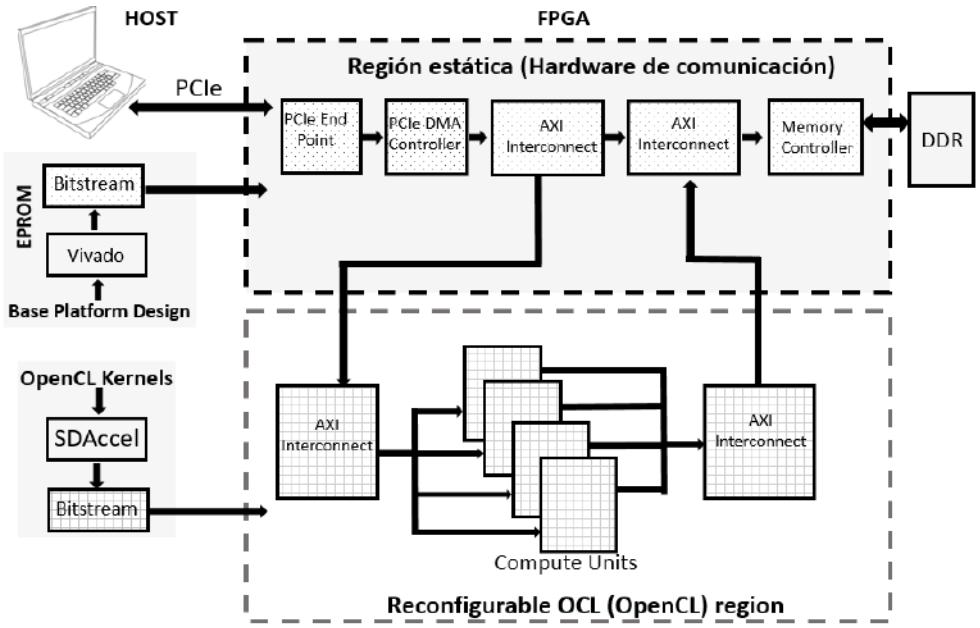

Figura 3.12. Modelo basado en OpenCL

Usar el entorno SDAccel OpenCL para realizar cálculos en un FPGA implica tanto el código del host como el del kernel. El código de host se utiliza para programar el FPGA, pasar datos entre la memoria del host, la memoria global del FPGA, y ejecutar el kernel en el FPGA. Como se ve en la figura 3.12, el FPGA está segmentado en dos regiones, la región programable y la región estática. La región estática se programa al momento del encendido y contiene las interfaces para la memoria global y PCIe. La región programable contiene el kernel, el cálculo que se debe acelerar. El código del núcleo se sintetiza en hardware y se configura en la región programable del FPGA. El kernel sintetizado puede contener una o más unidades de cálculo (CU), donde una CU corresponde a la unidad de hardware

responsable del cálculo requerido. La parte del problema manejado por un solo CU se conoce como el tamaño del grupo de trabajo local, mientras que el tamaño de la tarea general que esté terminado se conoce como el tamaño global de grupo de trabajo. Cada CU tiene su propia memoria local a la que solo puede acceder la CU, mientras que todas las CU comparten la memoria global fuera de chip de la plataforma.

3.5.2 Frameworks para CNNs en FPGA

Otra opción para implementar el sistema es usar frameworks con un enfoque en FPGAs. Ejemplos de estos son trabajos como (Zhu, Liu, Wang, & Xie, s.f.), (Sharma, y otros, 2016) *y* (DiCecco, y otros, 2016). La figura 3.13 representa la idea para la implementación usando un framework. El concepto es tomar alguna arquitectura de una CNN probada (de las comunes en el estado del arte) y mediante el framework obtener el código equivalente en HDL para, mediante un HLS hacer el ciclo de diseño para FPGA.

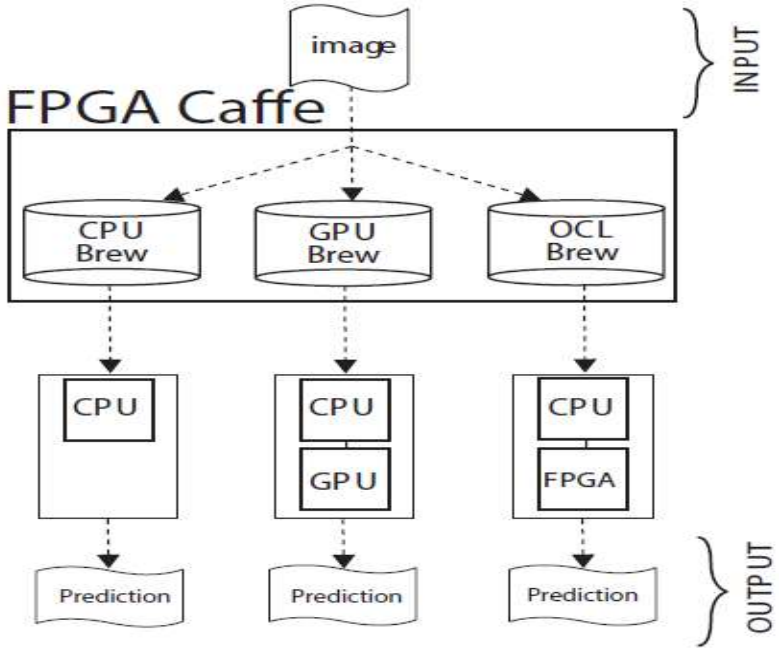

Figura 3.13. Modelo basado en un framework, tomado de (DiCecco, y otros, 2016).

3.5.3 HDL Coder™ de Mathworks®

Otra opción con un enfoque totalmente diferente es el usar software de terceros para generar el código HDL necesario para la aplicación. HDL Coder™ genera código portátil sintetizable Verilog y VHDL a partir de funciones de MATLAB™, modelos en Simulink™ y gráficos de Stateflow™. En (MathWorks, 2017) está una guía completa sobre este concepto.

En la tabla 3.4 se resumen las ventajas y desventajas que tiene el usar las diferentes herramientas para implementación de CNNs en FPGAs. Cualquiera que sea el modelo que se tenga en mente, una cosa muy importante a considerar es el dispositivo seleccionado para la aplicación. Cada uno de estos cuenta con diferentes recursos, tanto de memoria como de módulos DSP, necesarios para la implementación.

Tabla 3.4. Ventajas y desventajas de los diferentes modelos.

Modelo de desarrollo	Pros	Contras
Modelo basado en OpenCL	• Optimización del código, movimiento de datos. • Optimización de recursos del dispositivo	• Se requiere de conocimiento en OpenCL y hardware. • Tiempo de desarrollo.
Frameworks para CNN en FPGA	• Se implementan arquitecturas previamente probadas. • Mayor facilidad para implementar una aplicación	• Se requiere de conocimiento del framework con el que se trabaja y el dispositivo a usar. • Menor optimización en general.
HDL Coder® de Mathworks	• Tiempo de desarrollo. • Uso de lenguaje gráfico y librerías para el desarrollo. • Simplicidad	• Uso de software de terceros. • La menor optimización de recursos. • Algunas librerías no soportadas. • Diversidad en las aplicaciones

Capítulo 4 Resultados experimentales

En este apartado se muestran los resultados derivados de la investigación. Primero se expone la arquitectura de la CNN que se implementó y probó en software, para luego hacer el análisis de los requerimientos y forma en que se puede implementar la red en el FPGA Artix 7.

Con el propósito de tener una referencia dierecta con una CNN existente se implementó una arquitectura similar a la LeNet 5 (primer CNN reportada en el estado del arte). Como ya se mencionó, el propósito de la red es clasificar imágenes de señales de tránsito vehicular para evaluar para evaluar su desempeño en cuanto a la precisión y tiempo de entrenamiento. Para esto se hicieron las siguientes etapas:

1) Obtención del banco de imágenes digitales.

2) Acondicionamiento de las imágenes para la CNN.

3) Diseño de la arquitectura de la CNN.

4) Implementación del código para entrenamiento y prueba de la red en CPU.

5) Realización de dos experimentos con diferentes cantidades de imágenes.

Para el caso de estudio, el banco de imágenes de señales de tránsito se tomó de la página de la Secretaría de Comunicaciones y Transportes (Secretaria, 2017). En este sitio se encuentran las imágenes de tránsito agrupadas por categorías. En este caso se emplearon las señales denominadas como “señalamiento vertical”, que se dividen en señales restrictivas, preventivas, informativas y servicios turísticos; además de las “señales de indicación de obras”. Se trabajó con estos grupos de imágenes (360 en total) debido a que son las que más comúnmente se encuentran en las vías públicas. Estas imágenes se encuentran en un formato png a color, con tamaños variables. Se acondicionaron las imágenes para adecuarlas a la CNN, el acondicionamiento consistió en pasarlas a escala de grises y escalarlas a 28x28

pixeles, que es el tamaño que se maneja en las bases de datos de imágenes de dígitos escritos a mano MNIST.

Para tener una base de datos lo suficientemente grande y con diversidad, se generaron más imágenes, a partir de las que se tenían. Estas nuevas imágenes se formaron agregando ruido gaussiano, ruido de Poisson y de sal y pimienta al 10% a las originales, aplicarles un proceso de dilatación y rotarlas a 90,180 y 270 grados. Estos procesos, el entrenamiento y prueba de la CNN se hicieron en Matlab™, debido a la facilidad de trabajar con imágenes con este software. Con el propósito de comparar resultados para diferente cantidad de datos, se obtuvieron dos conjuntos de imágenes, con el propósito de realizar dos experimentos generales. El resultado final fue un banco con 12,000 muestras y de seis clases diferentes, 2,000 imágenes por clase en un experimento. En el otro caso se tienen 8,000 imágenes con 8 clases. En la figura 4.1 se muestra una representación de lo descrito. En el anexo A se muestran ejemplos de los códigos empleados en la generación de los bancos de datos.

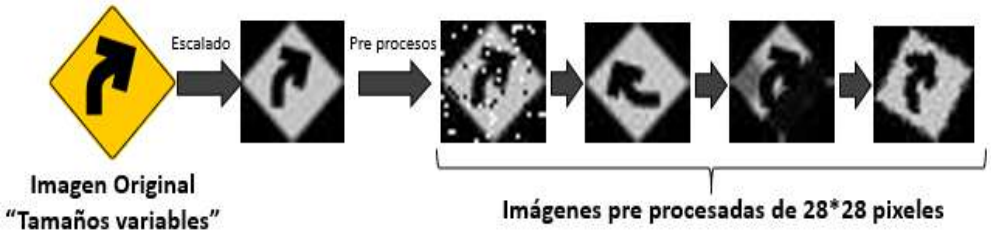

Figura 4.1. Pre procesamiento del banco de imágenes.

Para tener un punto de comparación con resultados reportados en el estado del arte en este tipo de tareas (LeNet 5), se implementó una CNN con dos capas de convolución y dos de agrupamiento, además de la capa de salida completamente conectada. En el siguiente apartado de muestra la arquitectura de la CNN con la que se trabajó.

4.1 Arquitectura de la red implementada

Antes de hacer el análisis de la implementación de la CNN en el FPGA o bien en el SoC, primero se entrena y prueba la red en un CPU, para ver el desempeño que tiene, en cuanto a la precisión y el tiempo de entrenamiento. El hardware empleado en los experimentos fue un CPU Quad-Core Intel Xeon E5 a 3.7 GHz y 12 GB de memoria RAM. La CNN cuenta con de dos capas de convolución y dos de agrupamiento, esto para tener un punto de comparación con una CNN de las reportadas en el estado del arte, (aunque diseñada para clasificar dígitos escritos a mano). En la literatura se menciona que conforme se incrementa la cantidad de información, mejora el desempeño en las aplicaciones con el paradigma de Deep Learning. Para ver el efecto de incrementar la cantidad de información en el desempeño de la CNN, se hicieron dos pruebas con diferentes cantidades de información. La tarea elegida es la clasificación imágenes de señales de tránsito vehicular comunes en México y otros países. En un caso se trata con 8,000 imágenes y en el otro con 12,000.

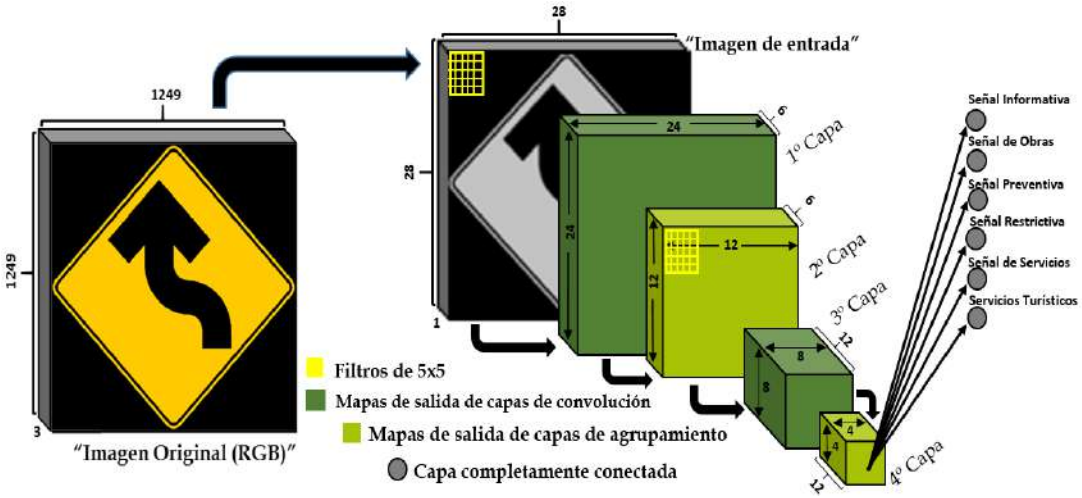

Figura 4.2. Arquitectura de la CNN implementada.

Luego de pre procesamiento, las imágenes de entrada a la CNN se encuentran en escala de grises, por lo tanto volumen de entrada es de 28x28x1, para cada imagen. Se emplean seis filtros (K=6) de 5x5 parámetros, no habrá relleno de ceros,

por lo que P = 0 en las ecuaciones 2.3 y 2.4. El paso del filtro (S) es 1. Con esto, el volumen de salida será de 24x24x6, según las ecuaciones 2.3, 2.4 y 2.5. Las capas de agrupamiento, o “downsampling”, que resultan de las de convolución serán de 12x12x6, según las ecuaciones 2.6, 2.7 y 2.8, (F= S = 2). El proceso se repite con las capas resultantes, ahora se emplean 12 filtros para obtener nuevos volúmenes de 8x8x12 y 4x4x12, respectivamente. En la figura 4.2 se muestra el proceso y la arquitectura de la CNN implementada.

4.1.1 Entrenamiento y prueba de la red

El entrenamiento de la red se hace usando el criterio 80-20, esto es usar 80% de las muestras para entrenar y 20% para probar. Con esto, en el caso del experimento de 8,000 imágenes, se usaron 6,400 en el entrenamiento y 1,600 para probar la CNN. Para el caso del experimento de 12,000 imágenes fueron 9,600 y 2,400 respectivamente.

El entrenamiento se hace por lotes y las imágenes se toman de forma aleatoria, tanto para seleccionar los conjuntos de entrenamiento y prueba, como para entrenar la CNN. Se utiliza el clásico backpropagation como algoritmo de aprendizaje y se deja fija la tasa de aprendizaje para todos los experimentos. Dado que la estructura de la red es pequeña en comparación con las reportadas en el estado del arte, se empleó la función sigmoide como función de activación, indicada en la ecuación 4.1. Para el caso de las capas de agrupamiento se usó el criterio del promedio para hacer el downsampling.

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

Los siguientes pasos muestran de manera general el procedimiento empleado para la generación de los conjuntos de imágenes, así como el entrenamiento y prueba de la CNN.

1. Obtención y acondicionamiento de las imágenes

Esta etapa consistió en bajar los conjuntos de imágenes de la página de la Secretaría de Comunicaciones y Transportes. Como se menciona anteriormente, estas 360 imágenes se encuentran en un formato png a color y fueron creadas empleando el software Autocad® por los autores. Dado que esa cantidad de imágenes es muy pequeña para los modelos del paradigma de Deep Learning, se crearon más imágenes a partir de las existentes, obteniendo 12,000 y 8,000 imágenes en total, todas previamente procesadas, quedando con un formato jpg en escala de grises y de 28X28 pixeles.

2. Generar los conjuntos de entrenamiento y prueba de forma aleatoria, a partir del banco de imágenes previamente obtenido.

Aquí se generaron los conjuntos para entrenamiento y prueba de la CNN. Esto se hizo tomando las muestras previamente guardadas de forma aleatoria y con el criterio 80-20. De esta sección se obtuvo un arreglo que contiene esos dos conjuntos juntos, junto con la etiqueta de cada muestra.

3. Programa principal:

1. Cargar los conjuntos de imágenes de señales de tránsito para el entrenamiento y prueba de la CNN, previamente acondicionadas y sus respectivas etiquetas.

2. Inicializar la semilla para generar los números aleatorios.

3. Definir la arquitectura de la CNN como sigue:

    a) Capa de entrada 28x28 pixeles, en escala de grises.

    b) Mapas de salida de la primera capa de convolución = 6, filtros de 5x5.

    c) Capa de agrupamiento, escala = 2.

    d) Mapas de salida de la segunda capa de convolución = 12, filtros de 5x5.

    e) Capa de agrupamiento, escala = 2.

4. Llamar a la función para inicializar aleatoriamente los parámetros de la red.

5. Definir la tasa de aprendizaje, tamaño de los lotes y cantidad de épocas de entrenamiento.

6. Llamar a la función del entrenamiento de la CNN.

7. Llamar a la función para probar la CNN previamente entrenada.

8. Calcular el número de muestras clasificadas correctamente.

9. Imprimir los resultados.

3.4 Función para inicializar los parámetros de la CNN

1. Para cada capa de entrada, calcular la dimensión de los mapas de salida.

    a) Para mapas de agrupamiento ecuaciones 2.6 a 2.8.

    b) Para mapas de convolución ecuaciones 2.3 a 2.5

2. Inicializar cada parámetro de los filtros, además del bias.

3. Inicializar los pesos de la capa de salida de forma aleatoria.

3.6. Función de entrenamiento de la CNN:

1. Inicializar aleatoriamente los parámetros de la red.

2. Verificar que la cantidad de lotes seleccionado sea un número entero.

3. Por cada época de entrenamiento, repetir:

    3.1 Seleccionar aleatoriamente el orden del conjunto de entrenamiento que se presenta a la red.

    3.2 Por cada lote de entrenamiento seleccionado, repetir:

    a) Llamada a la función para hacer el cálculo de todos los parámetros de la CNN (Feed Forward).

    b) Llamada a la función para calcular los gradientes usando el algoritmo de backpropagation (Feed Backward).

    c) Llamada a la función para actualizar los parámetros de la CNN aplicando el gradiente.

3.7. Función para prueba de la CNN:

1. Recibe la configuración actual de la CNN, el conjunto de prueba y sus etiquetas.

2. Realiza el pase de conjunto por la CNN (Feed Forward) con los datos recibidos.

3. Asignar la categoría a la que pertenece cada imagen, según el máximo score obtenido.

4. Determinar las imágenes clasificadas de forma incorrecta.

5. Calcula y retorna el error, según la cantidad de imágenes clasificadas incorrectamente.

Función FeedForward:

1. Recibe la estructura de la CNN con sus parámetros actualizados y las imágenes de entrada.

2. Para cada capa de la red, excepto la de entrada, repetir:

a) Si es capa de convolución:

i) Para cada mapa de salida (en este caso 6):

* Calcula la matriz de pesos mediante la operación de convolución entre los datos de entrada y los filtros asignados.

** Se agrega el bias y se aplica la función sigmoidal a los datos.

ii) Calcula la cantidad de mapas de entrada de la siguiente capa.

b) Si es capa de agrupamiento:

i) Calcula el promedio de los mapas de entrada y genera los nuevos mapas de características con estos datos (downsampling).

3. Regresa la nueva configuración de la CNN.

Función BackPropagation

1. Calcular el error y la función de pérdida.

2. Calcular los pesos de cada parámetro.

3. remodelar los deltas de vectores de características en el estilo del mapa de salida.

4. Calcular el gradiente.

5. Actualizar los pesos.

En el anexo A se muestra el código implementado para los pasos anteriores, junto con las funciones que no se indican.

4.2 Resultados con 8,000 imágenes y 8 clases

Con el propósito de ver la diferencia en cambiar el tamaño del lote para el entrenamiento, así como la cantidad de épocas con las que se entrena la CNN, en cada experimento también se hizo una variación de estos factores. De esta forma, para cada experimento se hicieron pruebas con lotes de 32, 100 y 200 muestras, y cada uno de estos con diferentes cantidades de épocas de entrenamiento, dejando la tasa de aprendizaje fija en todos los casos.

Tabla 4.1. Resultados obtenidos hasta 2200 épocas de entrenamiento, en el caso de 8,000 imágenes.

# de Épocas	Exactitud (%)			Tiempo de entrenamiento (Segundos)			Tiempo Ent./Época (Segundos)		
	L32	L100	L200	L32	L100	L200	L32	L100	L200
200	82.94	73.50	64.25	1623.1	1281.2	**1200.98**	8.12	6.41	**6.00**
600	89.69	84.06	77.56	4870.2	3854.7	**3620.9**	8.12	6.42	**6.03**
1000	88.50	85.75	83.81	8116.3	6420.8	**6032.5**	8.12	6.42	**6.03**
1400	**91.13**	88.75	86.19	**11361.5**	8993.9	**8454.8**	8.12	6.42	**6.04**
1800	89.56	88.50	86.81	14610.4	11554.4	**10887.5**	8.12	6.42	**6.05**
2200	90.19	**89.62**	**88.50**	17842.2	14110.0	**13229.4**	8.11	6.41	**6.01**

En la tabla 4.1 se muestran los resultados obtenidos hasta 2200 épocas de entrenamiento y para cada tamaño de lote. En el caso de los experimentos con 32 en el tamaño del lote, significa que cada 200 lotes, se le muestra el conjunto de 6400 muestras de entrenamiento a la CNN, con lo que se tendría una época de

entrenamiento. Para 100 en el tamaño del lote, una época de entrenamiento será cada 64 lotes, y en el caso de 200 cada 32 lotes.

En la figura 4.3 se muestra el comportamiento del *Error Cuadrático Medio* obtenido durante el entrenamiento, para el experimento de 2200 épocas y 100 en el tamaño del lote. Ese mismo comportamiento se observa en los demás casos de experimentación. En la figura 4.4 se muestra la *tendencia del error en la clasificación* al probar la CNN para cada prueba realizada.

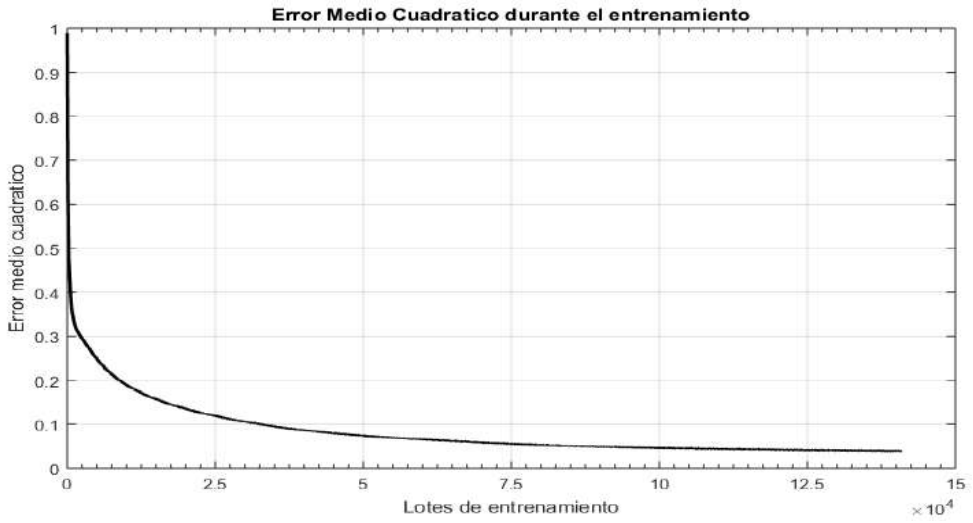

Figura 4.3. Comportamiento del Error Cuadrático Medio durante el entrenamiento.

Con el propósito de ver el desempeño de la red para una mayor cantidad de épocas, se hizo un experimento más, ahora el tamaño de los lotes fue de 200 y la cantidad de épocas de 3800. La tendencia del Error Cuadrático Medio fue la misma que la mostrada en la figura 4.3, la exactitud que se alcanzó en este último experimento fue de 90.19%, en un tiempo de 22,947.17 segundos (6.37 horas), con un promedio de 6.04 segundos por época.

Los tamaños de los lotes se eligieron tomando en cuenta la cantidad de imágenes de entrenamiento (6400), de la tabla 4.1 se observa que con forme se reduce el tamaño del lote, se mejora la exactitud en la clasificación, pero se incrementa el tiempo de entrenamiento. Esto sugiere que para tamaños de 16, 8 o

4 muestras en los lotes, se obtendría un mejor desempeño en cuanto a la exactitud, pero tiempos de entrenamiento más grandes. En cuanto al comportamiento del *Error Cuadrático Medio,* es el esperado al entrenar cualquier red neuronal artificial.

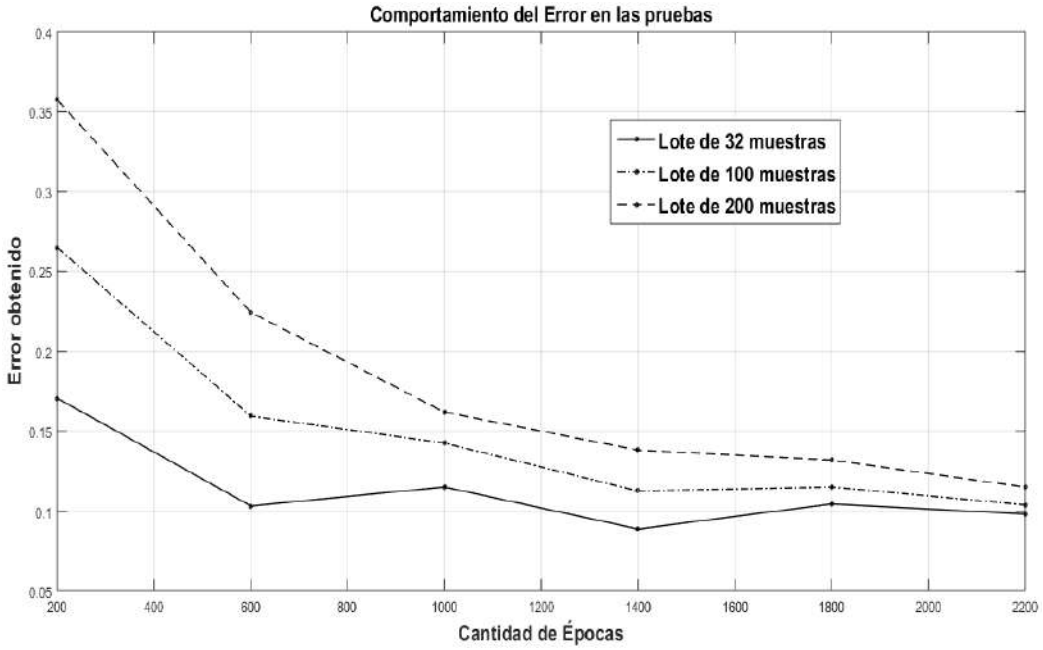

Figura 4.4. Tendencia del error en la clasificación para el experimento de 8,000 imágenes.

4.3 Resultados con 12,000 imágenes y 6 clases

En este experimento se hizo básicamente el mismo procedimiento que en el caso anterior, sólo que ahora, al tener 12,000 imágenes en total, los tamaños de los lotes se eligieron de 20, 40, 100 y 200. En lugar de presentar el error obtenido en las pruebas se presenta la exactitud alcanzada con fines de visualización. En la figura 4.5 se presentan los resultados obtenidos hasta 1200 épocas de entrenamiento para cada caso.

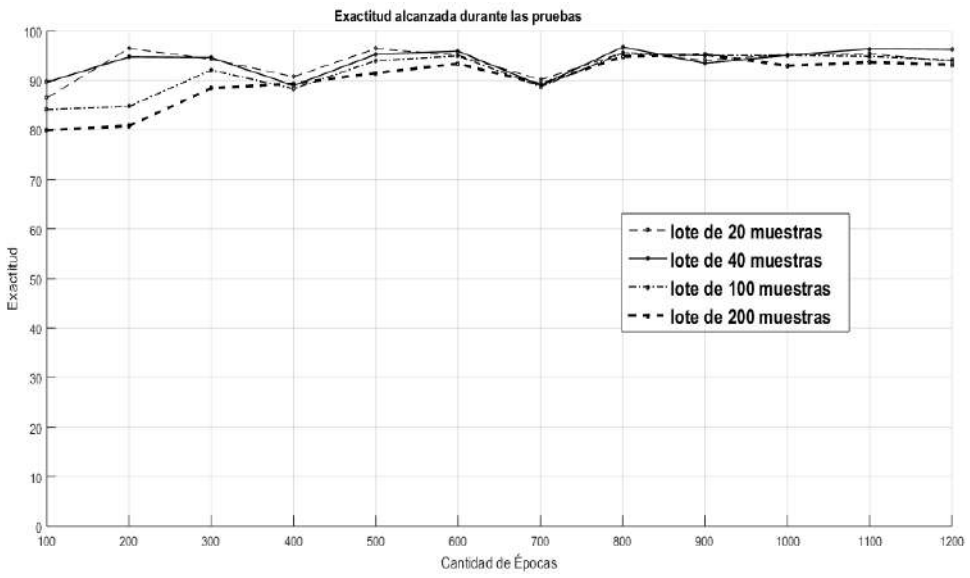

Figura 4.5. Exactitud obtenida durante las pruebas.

La tabla 4.2 muestra los errores mínimo, máximo y promedio obtenido en cada experimento, además del tiempo de entrenamiento obtenido por época para cada caso.

Tabla 4.2. Valores extremos obtenidos en las pruebas.

Tamaño de lote (muestras)	Error mínimo (%)	Error máximo (%)	Error promedio (%)	Tiempo de entrenamiento/Época (Seg)
20	3.54	13.54	5.31	14.21
40	3.25	11.04	5.08	11.22
100	4.46	15.88	5.94	9.61
200	4.75	20.04	7.77	9.01

En cuanto al error medio cuadrático para estas pruebas se muestra en la figura 4.6, para el caso de 20 en el tamaño del lote. Analizando los resultados anteriores, se observa que el error más pequeño alcanzado, para estos experimentos fue de 3.25%, que la CNN converge más rápido cuando se tiene mayor cantidad de información, que el error mejora con forme se reduce el tamaño del lote y que el tiempo de entrenamiento se reduce con forme se incrementa el tamaño del lote, esto para cada experimento.

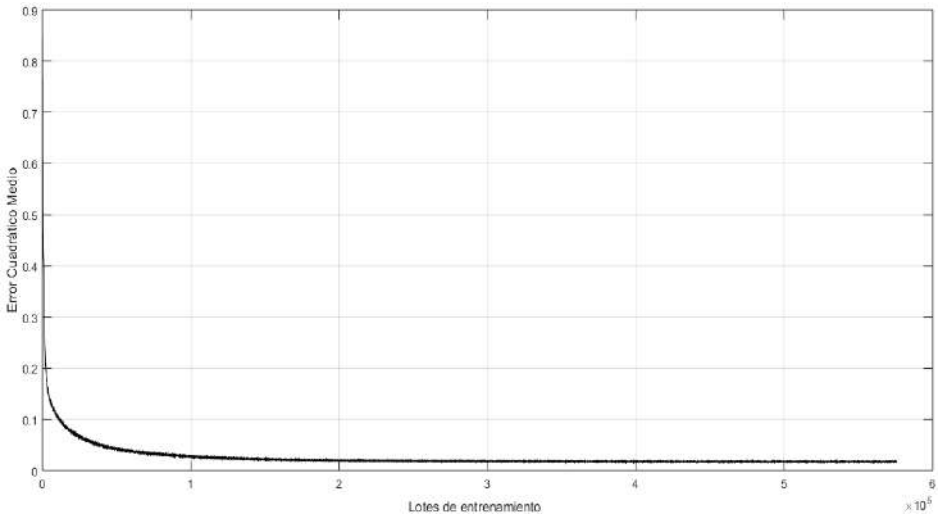

Figura 4.6. Comportamiento del Error Cuadrático Medio durante el entrenamiento del experimento con 20 muestras por lote.

No es la intención en el presente trabajo hacer una descripción sobre los algoritmos de Backpropagation, descenso del gradiente, funciones de activación y demás características que se llevan a cabo para el funcionamiento de la CNN, ya que estos conceptos son loa más comúnmente empleados en otros tipos de redes.

Una vez que se han obtenido los parámetros de la red que provocarán el desempeño mostrado, el siguiente paso sería la implementación en hardware de esta arquitectura y proceder a ingresar imágenes nunca antes vistas por la red. La intención es que estas imágenes ya provengan del exterior de sistema embebido. Como ya se mencionó, el propósito de emplear un FPGA es acelerar el cómputo necesario en las capas de la red, sobre todo las capas de convolución, que es en donde se requiere la mayor parte del cómputo. En el siguiente apartado se da la propuesta de implementación de la CNN en hardware.

4.4 Análisis de la implementación de la CNN en hardware

Dada la naturaleza paralela de las CNNs y tomando en cuenta las ventajas que ofrecen los FPGAs en comparación con otros dispositivos para la implementación de algoritmos con esa característica, en esta sección se hace el análisis sobre la

implementación de la CNN, esto ya con sus parámetros determinados en la sección anterior.

Si bien las dimensiones de la arquitectura de la CNN implementada en software son pequeñas en comparación con otras reportadas en el estado del arte para este tipo de tareas, el siguiente análisis puede ser extendido para tener un procedimiento general sobre la implementación de cualquier topología de CNN en hardware reconfigurable. Primero se hace un análisis sobre la complejidad computacional y espacial requerida para obtener la cantidad de recursos computacionales necesarios, así como la cantidad de memoria.

4.4.1 Complejidad computacional

La complejidad computacional de una capa es la suma de cada operador en esta capa. Como se menciona en el capítulo 2, en una capa de convolución cada mapa de características de entrada se convoluciona con un filtro de dimensiones W_FxH_F, lo que da como resultado un mapa de características de salida W_{cout}xH_{cout}; el número de capas de entrada y salida es Nin, Ncout respectivamente. Dado que cada elemento necesita dos operaciones para multiplicación y adición, entonces la complejidad de una capa de convolución es:

$$C_{conv} = 2 * N_{in} * W_F * H_F * W_{cout} * H_{cout} * N_{cout} \quad (4.2)$$

Para el caso de la función de activación de la misma capa:

$$C_{act} = N_{out} * \ W_{cout} * H_{cout} \quad (4.3)$$

Para la capa de agrupación, los mapas de características de entrada se muestrean a través de un filtro W_PxH_P, generando salidas N_{pout} de W_{pout}xH_{pout}. La complejidad de cálculo de esta capa es:

$$C_{agrup} = W_P * H_P * W_{pout} * H_{pout} * N_{pout} \quad (4.4)$$

Para la capa completamente conectada, los nodos de entrada N_{FCin} se multiplican y suman con N_{FCout} parámetros, entonces:

$$C_{FC} = 2 * N_{FCin} * N_{FCout} \quad (4.5)$$

4.4.2 Complejidad espacial

El espacio en memoria necesario se determina con la complejidad espacial. El parámetro principal en la CNN son los pesos que se usan en las capas convolucionales y las completamente conectadas. El número de pesos en una capa convolucion se puede expresar como:

$$S_{conv} = W^2 * N_{cin} * N_{cout} \quad (4.6)$$

Y para la capa completamente conectada:

$$S_{FC} = N_{cin} * N_{cout} \quad (4.7)$$

Tabla 4.3. Configuración de la CNN.

Configuración de la CNN										
	Entrada			Filtro		Salida				
Capa	N_{in}	W	H	W_F	H_F	N_{out}	W_P	H_P	# MAC	# Parámetros
Convolución	1	28	28	5	5	6	24	24	235200	150
Activación						6	24	24	3456	0
Agrupación	6	24	24	2	2	6	12	12	3456	0
Convolución	6	12	12	5	5	12	8	8	518400	1800
Activación						12	8	8	768	0
Agrupación	12	8	8	2	2	12	4	4	768	0
FC	768					8			12288	6144
								TOTAL	**774336**	**8094**

En la

Tabla 4.3 se hace el cálculo de la cantidad de operaciones MAC a ejecutar y el espacio de memoria requerido para cada capa, así como los totales. Con esto se obtiene la cantidad de recursos necesarios de la CNN. En la tabla se observa que el 98.4 % de las operaciones MAC se encuentra en las capas de convolución y agrupación, mientras que la capa completamente conectada es la que requiere de mayor cantidad de memoria para almacenamiento con 75.9 %. Esto ratifica la prioridad de implementar las operaciones MAC de forma eficiente en una aplicación con CNNs.

4.4.3 Arquitectura propuesta

Claro está que ningún FPGA en el mercado cuenta con la cantidad de bloques DSP para hacer todas las operaciones MAC al mismo tiempo (aún para estas dimensiones de la red), por lo que se requiere una sección de control para turnar el cálculo de las operaciones, así como el ingreso de las imágenes al sistema. Lo anterior haciendo el re uso de bloques de procesamiento, a la vez de maximizar el paralelismo del procesamiento. Como se menciona en el capítulo anterior, otro factor importante para hacer eficiente el proceso es el flujo de los datos. En la Figura 4.7 se muestra el sistema propuesto para acelerar la CNN y minimizar el consumo de energía.

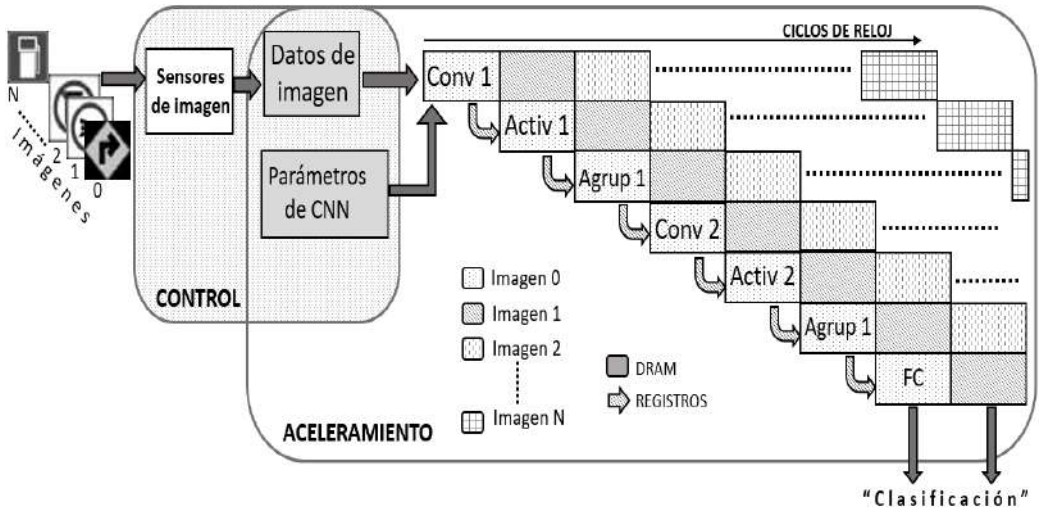

Figura 4.7. Arquitectura del sistema propuesto para el FPGA.

En la figura 4.7 se tienen las imágenes de entrada (o tramas de video) que luego de ser adquiridas pasan a la memoria global DRAM del FPGA. De este sitio se toman los datos junto con los parámetros de la CNN previamente entrenada para llevar a cabo el procesamiento descrito anteriormente. La sección de control será la responsable de hacer el desplazamiento de los filtros a lo largo de la imagen para cada una de las capas.

Para el caso de estudio, dado que los filtros son de 5X5 y las imágenes de 28X28, se requerirán de 25 módulos de DSP-48 por cada mapa de activación. Dado que en la primera capa de convolución se tienen 6 filtros, entonces se requiere de 150 módulos de DSP-48 para llevar a cabo el cómputo y generar las seis capas de características en paralelo. Para el caso de la segunda capa de convolución se requiere de 25X12 = 300 módulos DSP. Lo que en total resultan 450 módulos para las capas de convolución. En el caso de las capas de agrupamiento se requiere de 6 + 12 = 18 módulos.

El FPGA Artix 7 cuenta con 740 de estos módulos, que serían más que suficientes para implementar el cómputo de estas capas en paralelo. No se hace el análisis de la cantidad de ciclos totales que se requeriría para el cómputo completo de estas capas, ya que son de diferente tamaño con forme se avanza en la

trayectoria del flujo de la red. En la figura se observa que luego del cálculo de 7 capas se obtiene el resultado para la primera imagen, pero luego de esto sólo se necesitará el cálculo de una capa para generar el resultado de la siguiente imagen y esto se repite continuamente. Esto por la implementación del pipelining entre las capas. También se hace referencia al uso de la memoria local mediante los registros y el re uso de recursos computacionales. Claro está que a las imágenes ingresadas al sistema les hará falta el pre procesamiento, que para el caso del entrenamiento se hizo de forma manual. Este pre procesamiento se debe hacer en la sección de "datos de la imagen" de la figura 4.7, básicamente es escalar las imágenes a 28X28 pixeles y en escala de grises, que es lo que finalmente acepta la red.

Capítulo 5 Conclusiones

Uno de los propósitos del presente trabajo fue entrenar y probar una red neuronal convolucional, con la intensión de valorar los tiempos de entrenamiento y el error en la clasificación de imágenes de señales de tránsito vehicular empleadas en México y otros países. La CNN que se implementó es pequeña comparada con las reportadas en el estado del arte en este tipo de aplicaciones, esto fue así para hacer el entrenamiento en un CPU, y analizar su implementación en un FPGA. De los resultados obtenidos en las Figura 4.3 y 4.4, se observa que para el experimento en el que se cuenta con una mayor cantidad de información (banco de 12,000 imágenes), el entrenamiento converge en menor cantidad de épocas que en el caso que se tiene menor cantidad de información (8,000 imágenes), esto para todos los casos de experimentación. La exactitud en la clasificación es mejor en el caso que se tiene mayor cantidad de información, 96.75% contra 91.13 %. En los dos casos, el desempeño mejora con forme se reduce el tamaño del lote. Como es de esperarse, el tiempo de entrenamiento al incrementar la cantidad de información, esto se indica en la tabla 4.1 y tabla 4.2.

En la tabla 4.1 se observa que el resultado de variar el tamaño del lote para el mismo experimento provoca que la red tenga un desempeño diferente. Para tamaños de lote más pequeños se tiene un mejor desempeño en cuanto a la exactitud en la clasificación (91.13%, 89.62% y 88.5% para lotes de 32, 100 y 200 imágenes respectivamente), pero el tiempo de entrenamiento se incrementa por la mayor cantidad de actualizaciones de pesos de la CNN. El comportamiento del Error Cuadrático Medio obtenido en todos los experimentos muestra una tendencia esperada al entrenar cualquier red neuronal artificial.

Si tomamos el menor error para todos los experimentos en la clasificación (3.25 %), se aleja un tanto de los resultados reportados en el estado del arte en ese tipo de tareas. En el reto de clasificación de dígitos escritos a mano (MNIST) también se manejan imágenes de 28x28 en escala de grises y con la misma cantidad de capas, aunque con filtros de tamaño diferente. Lo reportado en el estado del arte en ese reto es un error de 0.22%, “prácticamente está resuelto ese problema”. Una diferencia marcada es la cantidad de información, en ese reto se maneja un banco de 60,000 imágenes para entrenamiento y 10,000 para prueba, además de que la red implementada para obtener ese desempeño es más profunda y se agregan otro tipo de capas a la red. La tendencia encontrada muestra que el desempeño de una CNN mejora al incrementar la cantidad de información. Otro factor a tener en cuenta en el caso de hacer alguna comparación con el MNIST, es la variabilidad de las imágenes, ya que este trabajo una señal de STOP pertenece a la misma clase que la de un límite de velocidad, por ejemplo.

En cuanto al análisis de la implementación de la arquitectura programada en un FPGA se encontró que es viable, no sólo para el caso de estudio, si no que el mismo concepto se puede generalizar para arquitecturas más grandes, la única limitante que se tiene es la cantidad de memoria requerida para guardar tanto los parámetros de la CNN como la imagen de entrada. Para el caso de experimentación se requiere poco más de 8 KB de memoria, mientras que el Artix 7 de Xilinx cuenta con unos 13 MB. Esto sugiere espacio suficiente para integrar la parte del control de información en el mismo chip. Otra opción, que para la arquitectura propuesta

sería más simple, es usar un SoC, en el que se cuenta con un DSP además del FPGA, lo que facilitaría la parte del control de información.

Por la investigación realizada, para redes neuronales convolucionales con arquitecturas mucho más grandes (reportadas en el estado del arte), parece viable su implementación en FPGAs, sólo que no es algo transparente para un ingeniero de software. Un factor importante a considerar es el tamaño de la CNN a implementar, en comparación con los recursos disponibles en el dispositivo seleccionado. Con la constante investigación que se realiza en esta área, es de esperarse que en un corto plazo se cuente con frameworks enfocados en ese punto, y con ello facilitar la implementación de CNN existentes directamente en FPGAs. Con esto se podrían obtener soluciones directas para aplicaciones embebidas, en las que el consumo de potencia, tiempo de latencia y tamaño espacial son factores muy importantes.

5.1 Trabajo futuro

Dada la discusión presentada aquí, existen diversos trabajos pendientes, como el incrementar la cantidad de capas de convolución de la CNN y ver la diferencia en el desempeño, así como incrementar la cantidad y variabilidad de las imágenes. Otro punto, es probar la red con imágenes tomadas en avenidas o autopistas que contengan señales de tránsito, así como probar la configuración obtenida en un sistema embebido.

El punto más importante para el trabajo futuro es la implementación de la arquitectura obtenida en un FPGA.

Referencias

Benenson, R. (06 de 03 de 2017). *What is the class of this image ? Discover the current state of the art in objects classification.* Obtenido de Classification datasets results: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#494c5356524332303132207461736b2031

Berg Palm, R. (20 de Mayo de 2017). *GitHub Repository.* Recuperado el 12 de Mayo de 2017, de GitHub Repository: https://github.com/rasmusbergpalm/DeepLearnToolbox

Bristow, H., & Lucey, S. (s.f.). Why do linear SVMs trained on HOG features perform so well? *arXiv:1406.2419v1 [cs.CV] 10 Jun 2014*.

Chatfield, K., Simonyan, K., Vedaldi, A., & Zisserman, A. (s.f.). Return of the Devil in the Details: Delving Deep into Convolutional Nets. *arXiv:1405.3531v4 [cs.CV] 5 Nov 2014*.

Chellapilla, K., Puri, S., & Simard, P. (Octubre de 2006). High Performance Convolutional Neural Networks for Document Processing. (G. Lorette, Ed.) *Tenth International Workshop on Frontiers in Handwriting Recognition*. Obtenido de https://hal.inria.fr/inria-00112631

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., & Chen, Y. (2014). DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (págs. 269-284). Salt Lake City, Utah, USA: ACM New York, NY, USA.

Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., & Schmidhuber, J. (Julio de 2011). Flexible, High Performance Convolutional Neural Networks for Image Classification. *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, 1237-1242. doi:10.5591/978-1-57735-516-8/IJCAI11-210

Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., & Ng, A. (2013). Deep learning with COTS HPC systems. *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 1337-1345.

Conneau, A., Schwenk, H., & Le Cun, Y. (s.f.). Very Deep Convolutional Networks for Text Classification. *arXiv:1606.01781v2 [cs.CL]*.

Cope, B., Cheung, P. Y., & Luk, W. (2007). Bridging the Gap between FPGAs and Multi-Processor Architectures: A Video Processing Perspective. *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 308-313.

Crockett, L. H., Elliot, R. A., Enderwitz, M. A., & Stewart, R. W. (2014). *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC.* Glasgow, Scotland, UK: Strathclyde Academic Media.

Dalal, N., & Triggs, B. (2005). Histograms of Oriented Gradients for Human Detection. *CVPR '05 Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 886-893.

Deng, L., & Yu, D. (20014). *Deep Learning, Methods and Applications.* Redmond, WA 98052; USA: ow the essence of knowledge. doi:10.1561/2000000039

DiCecco, R., Lacey, G., Vasiljevic, J., Chow, P., Taylor, G., & Areibi, S. (2016). Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks. *arXiv:1609.09671v1 [cs.CV]*.

Donahue, J., Hendricks, L. A., Rohrbach, M., Venugopalan, S., Guadarrama, S., Saenko, K., & Darrell, T. (s.f.). Long-term Recurrent Convolutional Networks for Visual Recognition and Description. *arXiv:1411.4389v4 [cs.CV]*.

Dubout, C., & Fleuret, F. (2012). Exact Acceleration of Linear Object Detectors . *In Proceedings ECCV*, 301-311.

Fahlman, S. E., Hinton, G. E., & Sejnowski, T. J. (1983). Massively Parallel Architectures for Al: NETL, Thistle, and Boltzmann Machines. *Proceedings of the National Conference on Artificial Intelligence*, 109-113.

Farabet, C., Martini, B., Akselrod, P., Talay, S., LeCun, Y., & Culurciello, E. (2010). Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems. *Proceedings of 2010 IEEE International Symposium*, 257- 260.

Farabet, C., Poulet, C., & Han, J. Y. (2009). CNP: An FPGA-based processor for Convolutional Networks. *19th International Conference on Field Programmable Logic and Applications*, (págs. 32-37). Prague, Czech Republic. doi:10.1109/FPL.2009.5272559

Fowers, J., Brown, G., Cooke, P., & Stitt, G. (2012). A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications. *FPGA '12 Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays* (págs. 47-56). Monterey, California, USA: ACM New York, NY, USA.

Gerónimo, D., Sappa, A. D., López, A., & Ponsa, D. (2006). Pedestrian detection using AdaBoost learning of features and vehicle pitch estimation. *Proceedings of the Sixth IASTED International Conference Visualization, Imaging, and Image Processing* (págs. 400-405). Palma de Mallorca, Spain: IASTED.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* Cambridge, Massachusetts, United States of America: MIT Press.

Graham, B. (s.f.). Fractional Max-Pooling. *arXiv:1412.6071v4 [cs.CV]*.

Griffin, L., Graham, T., & Shawki, A. (s.f.). Deep Learning on FPGAs: Past, Present, and Future. *arXiv:1602.04283v1 [cs.DC] 13 Feb 2016*.

Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., & Fei-Fei, L. (2014). Large-scale Video Classification with Convolutional Neural Networks. *CVPR '14 Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, 1725-1732.

Krizhevsky, A., Sutskever, I., & E. Hinton, G. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems 25*, 1097-1105.

Lavin, A., & Gray, S. (2016). Fast Algorithms for Convolutional Neural Networks. *In Proceedings CVPR*, 4013-4021.

Le, Q., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G., . . . Ng, A. (2012). Building high-level features using large scale unsupervised learning. *Proceedings of the 29 th International Conference on Machine Learning*, 81-88.

LeCun, Y. (2012). Learning Invariant Feature Hierarchies. *European Conference on Computer Vision (ECCV 2012)* (págs. 496-505). Florence, Italy: Springer.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *NATURE*, 436-444.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 541-551.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 2278--2324.

LeCun, Y., Kavukcuoglu, K., & Farabet, C. (2010). Convolutional Networks and Applications in Vision. *IEEE*, 253-256.

M. Bertozzi, A. B. (2007). A Pedestrian Detector Using Histograms of Oriented Gradients and a Support Vector Machine Classifier. *Intelligent Transportation Systems Conference* (págs. 143-148). Seattle, WA, USA: IEEE.

MatConvNet. (28 de Mayo de 2017). *MatConvNet: CNNs for MATLAB*. Obtenido de http://www.vlfeat.org/matconvnet/

Mathieu, M., Henaff, M., & LeCun, Y. (s.f.). Fast Training of Convolutional Networks through FFTs. *arXiv:1312.5851v5 [cs.CV]*.

MathWorks. (20 de Octubre de 2017). *HDL Coder*. Obtenido de Generate VHDL and Verilog code for FPGA and ASIC designs: https://www.mathworks.com/products/hdl-coder.html

Microsoft. (1 de junio de 2011). *Project Catapult*. Obtenido de https://www.microsoft.com/en-us/research/project/project-catapult/

Mitchell, T. M. (1997). Does Machine Learning really work? *AI Magazine*, 11-21.

Ovtcharov, K., Ruwase, O., Kim, J.-Y., Fowers, J., Strauss, K., & Chung, E. S. (2015). Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. *Microsoft Research*, 1- 4.

Pauwels, K., Tomasi, M., Díaz, J., Ros, E., & Van Hulle, M. M. (2012). A Comparison of FPGA and GPU for Real-Time Phase-Based Optical Flow, Stereo, and Local Image Features. *IEEE TRANSACTIONS ON COMPUTERS*, 999-1012.

Potluri, S., Fasih, A., Kishore Vutukuru, L., Al MachoT, F., & Kyamakya, K. (2011). CNN Based High Performance Computing for Real Time Image Processing on GPU. *Autonomous Systems: Developments and Trends*, 255-266.

Qiao, Y., Shen, J., Xiao, T., Yang, Q., Wen, M., & Zhang, C. (6 de Mayo de 2016). *FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency.* doi:10.1002/cpe.3850

Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., . . . Yang, H. (2016). Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. *ACM*, 26-35. doi:10.1145/2847263.2847265

Rahman, A., Lee, J., & Choi, K. (2016). Efficient FPGA Acceleration of Convolutional Neural Networks Using Logical-3D Compute Array. *Proceedings of the 2016 Conference on Design, Automation & Test in Europe* (págs. 1393 - 1398). Dresden, Germany: EDA Consortium San Jose, CA, USA.

Ratheesh Kalarot, John Morris. (2010). Comparison of FPGA and GPU implementations of Real-time Stereo Vision. *IEEE*, 1-6.

Redmon, J., & Farhadi, A. (s.f.). YOLO9000: Better, Faster, Stronger. *arXiv:1612.08242v1 [cs.CV]*.

Redmon, J., Divvalay, S., Girshick, R., & Farhadi, A. (s.f.). You Only Look Once: Unified, Real-Time Object Detection. *arXiv:1506.02640v5 [cs.CV]*.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., . . . Fei-Fe, L. (27 de Mayo de 2017). *Large Scale Visual Recognition Challenge 2017 (ILSVRC2017)*. Obtenido de http://image-net.org/challenges/LSVRC/2017/results

Salakhutdinov, R., & Hinton, G. (2009). Deep Boltzmann Machines. (D. v. Welling, Ed.) *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, 448--455.

Secretaria, d. C. (12 de Mayo de 2017). *SCT*. Obtenido de Banco Digital de Señalización Vial: http://www.sct.gob.mx/bancodigital/

Sharma, H., Park, J., Amaro, E., Thwaites, B., Kotha, P., Gupta, A., . . . Esmaeilzadeh, H. (2016). From High-Level Deep Network Models to FPGA Acceleration. *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, 2016*, 1-12.

Shi, S., Wang, Q., Xu, P., & Chu, X. (s.f.). Benchmarking State-of-the-Art Deep Learning Software Tools. *arXiv:1608.07249v6 [cs.DC] 25 Jan 2017*.

Simonyan, K., & Zisserman, A. (4 de Septiembre de 2015). VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION. *arXiv:1409.1556v6 [cs.CV]*. Obtenido de https://arxiv.org/abs/1409.1556v6

Sotelo, M. A., Parra, I., Fernández, D., & Naranjo, E. (2006). Pedestrian Detection using SVM and Multi-feature Combination. *2006 IEEE Intelligent Transportation Systems Conference* (págs. 104-108). Toronto, Canada: IEEE.

Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (s.f.). STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET. *arXiv:1412.6806v3 [cs.LG]*.

Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (Diciembre de 2017). Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE, 105*(12), 2295-2329. doi:10.1109/JPROC.2017.2761740

Tomé, D., Monti, F., Baroffio, L., Bondi, L., Tagliasacchi, M., & Tubaro, S. (s.f.). Deep convolutional neural networks for pedestrian detection. *arXiv:1510.03608v5 [cs.CV]*.

Trimberger, S. (Marzo de 2015). Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE, 103*(3), 318-331. doi:10.1109/JPROC.2015.2392104

Wan, L., Zeiler, M., Zhang, S., LeCun, Y., & Fergus, R. (2013). Regularization of Neural Networks using DropConnect. *Proceedings of the 30 th International Conference on Machine Learning*.

Wang, L., Ouyang, W., Wang, X., & Lu, H. (2015). Visual Tracking with Fully Convolutional Networks. *2015 IEEE International Conference on Computer Vision*, 3119-3127.

Wolpert, D. H., & Macready, W. G. (1997). No Free Lunch Theorems for Optimization. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 67-82.

Xilinx. (2 de Julio de 2013). Introduction to FPGA Design with Vivado High-Level Synthesis. USA: Xilinx.

Xilinx. (3 de 11 de 2017). *SDAccel Enviroment*. Obtenido de SDAccel Development Environment Help: https://www.xilinx.com/html_docs/xilinx2017_2/sdaccel_doc/topics/devices/con-fpga-architecture.html

Xilinx. (9 de Marzo de 2017). SDAccel Environment Optimization Guide. Xilinx.

Xuejie Nian, K. X. (2015). A Pedestrian Detection Method Based on MB_LBP Features and Intersection Kernel SVM. *Proceedings of the 2015 Chinese Intelligent Automation Conference*, 361-369.

Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. (2015). Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (págs. 161-170). Monterey, California, USA: ACM New York, NY, USA.

Zhao, W., Fu, H., Luk, W., Yu, T., Wang, S., Feng, B., . . . Yang, G. (2016). F-CNN: An FPGA-based Framework for Training Convolutional Neural Networks. *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*.

Zhu, M., Liu, L., Wang, C., & Xie, Y. (s.f.). CNNLab: a Novel Parallel Framework for Neural Networks using GPU and FPGA. *arXiv:1606.06234v1 [cs.LG]*.

Anexo A. Código implementado

El código implementado en Matlab® se muestra a continuación. Previo a la implementación de la CNN se hace el acondicionamiento de las imágenes. Estas se encuentran en un formato .png a color y con diferentes tamaños. Para el ingreso al programa, las imágenes deben de estar en un formato de 28*28 pixeles, en escala de grises. Además de esto, se les agregan los procesos de diversificación de las imágenes para tener un banco de 12000 muestras para un caso y 8000 para el otro.

1) Acondicionamiento de imágenes

Primero se escalaron las imágenes a 28*28 y en escala de grises. El siguiente código es un ejemplo de esta parte del acondicionamiento para 68 imágenes de la clase preventivas:

```
cd C:\Users\RVC\Documents\MATLAB\imagenes\preventivas;
for (imnumero=1:68)
imnumero=12;
imfile=['C:\Users\RVC\Documents\MATLAB\imagenes\preventivas\sp-' num2str(imnumero) '.png'];
imagenfile='C:\Users\RVC\Documents\MATLAB\imagenes\preventivas\sp-6.png';
imagen=imread(imfile);
imagen=imagen(:,:,1);
imshow(imagen);
numcols = 28;
numrows = 28;
figure(2);
imagen2= imresize(imagen, [numrows numcols]);
imagen2= rgb2gray(imagen2);
imshow (imagen2);
imfile=['C:\Users\RVC\Documents\MATLAB\imagenes\preventivas28x28\sp-' num2str(imnumero) '.jpg'];
imwrite(imagen2, imfile);
end
```

El mismo proceso se hizo con todas las imágenes para en primer instancia escalarlas a como se necesitan en el programa. El siguiente código sólo es un ejemplo de uno de los procesos que se aplicó a las imágenes. En el ejemplo se toman 200 imágenes de la clase “turísticas”, previamente guardadas y se giran -45°.

```
cd C:\Users\RVC\Documents\MATLAB\imagenes\bancoimagenes\turisticas;
for (i=1:200)
   imfile = ['C:\Users\RVC\Documents\MATLAB\imagenes\bancoimagenes\turisticas\sit-' num2str(i) '.jpg'];
   imagen=imread(imfile);
   imagen1 = imrotate(imagen, 345, 'crop');
   imfile=['C:\Users\RVC\Documents\MATLAB\imagenes\bancoimagenes\turisticas\sit-'
num2str(i+1800) '.jpg'];
   imwrite(imagen1, imfile);
end
```

Procesos similares se hacen para agregar ruido, dilatarlas, escalarlas, etc. De forma que se obtienen bancos de 8000 y 12000 imágenes para los diferentes experimentos.

2) Generación de los conjuntos de entrenamiento y prueba

A continuación, se muestra parte del código implementado para generar los conjuntos de entrenamiento y prueba para el caso del experimento de 8,000 muestras y 8 clases.

```
numclase1 = 990;
numclase2 = 1018;
numclase3 = 990;
numclase4 = 970;
numclase5 = 990;
numclase6 = 1050;
numclase7 = 990;
numclase8 = 1002;
numclases = 8;
testx1=[];
testx2=[];
testx3=[];
testx4=[];
testx5=[];
testx6=[];
testx7=[];
testx8=[];

etiquetas=zeros(numclase1+numclase2+numclase3+numclase4+numclase5+numclase6+numclas
e7+numclase8, numclases);

for j=1:numclase1
ruta = 'C:\Users\RVC\Documents\MATLAB\imagenes\bancoimagenes1\preventivas28x28\';
archivo = 'sp-';
rutaarchivo = [ruta archivo num2str(j) '.jpg'];
imagentemp = imread (rutaarchivo);
testx = [];
```

```
for i=1:28
    testx = [testx imagentemp(i,:)];
end
testx1=[testx1;testx];
end

for j=1:numclase2
ruta = 'C:\Users\RVC\Documents\MATLAB\imagenes\bancoimagenes1\preventivas128x28\';
archivo = 'sp1-';
rutaarchivo = [ruta archivo num2str(j) '.jpg'];
imagentemp = imread (rutaarchivo);
testx = [];
for i=1:28
    testx = [testx imagentemp(i,:)];
end
testx2=[testx2;testx];
end
.....
.....
for j=1:numclase8
ruta = 'C:\Users\RVC\Documents\MATLAB\imagenes\bancoimagenes1\serviciosB128x28\';
archivo = 'sit1-';
rutaarchivo = [ruta archivo num2str(j) '.jpg'];
imagentemp = imread (rutaarchivo);
testx = [];
for i=1:28
    testx = [testx imagentemp(i,:)];
end
testx8=[testx8;testx];
end

for i=1:numclase1
    etiquetas (i,:) = [1 0 0 0 0 0 0 0];
end

for i= numclase1+1: numclase1+numclase2
    etiquetas (i,:) = [0 1 0 0 0 0 0 0];
end

aleatorio=fix(1+(8000-1)*rand);
selec(1)=aleatorio;

while(size(selec,1)<6400)
    aleatorio=fix(1+(8000-1)*rand);
    tmp=find(selec==aleatorio);
    if(isempty(tmp))
        selec=[selec; aleatorio];
    end
end
total_imagenes = vertcat(testx1,testx2,testx3,testx4,testx5,testx6,testx7,testx8);

train_x = total_imagenes(selec, :);
```

```
train_y = etiquetas(selec,:);
test_x = removerows(total_imagenes,selec);
test_y = removerows(etiquetas,selec);
```

Con lo anterior se crean los conjuntos de entrenamiento con 6,400 muestras y prueba con 1,600, de forma aleatoria (criterio 80% entrenamiento y 20% prueba) para el caso de los experimentos de 8,000 imágenes. Las variables train_x y train_y contienen el conjunto de entrenamiento y sus etiquetas, mientras que test_x y test_y contienen el conjunto para prueba y sus etiquetas respectivamente. Para el caso de 12,000 se hizo un proceso similar. Estos conjuntos se guardan en un arreglo llamado database, que es el que se cargará en el programa principal.

3) Programa principal para entrenar y probar la CNN

En las siguientes líneas se muestra el código para entrenar y probar la CNN, esto luego de tener los conjuntos de entrenamiento y prueba, junto con sus etiquetas.

```
%% Carga de las imágenes.

load C:\Users\RVC\Documents\MATLAB\MINTS\database;

% Acondicionamiento de la base de datos, tanto para entrenamiento como para prueba, las
%imágenes deben estar.
% en un formato de 28*28*6400 y 28*28*1600 respectivamente. Cada pixel hay que normalizarlo de
%a 0-1.

train_x = double (reshape(train_x',28,28,6400))/255;
test_x = double (reshape(test_x',28,28,1600))/255;
train_y = double (train_y');
test_y = double (test_y');

% Inicialización de la red
% La arquitectura contiene una capa de entrada, dos capas de convolución junto con su función de
%activación, dos  % Capas de agrupamiento y una completamente conectada a la salida
% Cada capa se diferencia por i para entrada, c para convolutiva y s para agrupamiento
% Para capas de convolución; outputmaps contiene la cantidad de filtros o características a aprender
% kernelsize guarda el tamaño del filtro.
%  Para capas de agrupamiento, scale es la escala a la que se reducirán las capas

cnn.layers = {
    struct('type', 'i') % capa de entrada
    struct('type', 'c', 'outputmaps', 6, 'kernelsize', 5) % capa de convolución
    struct('type', 's', 'scale', 2) % capa de agrupamiento
    struct('type', 'c', 'outputmaps', 12, 'kernelsize', 5) % capa de convolución
    struct('type', 's', 'scale', 2) % capa de agrupamiento
};

% Reseteo de la semilla para el número aleatorio.
rand('state', 0)

% Opciones para el entrenamiento, alpha – tasa de aprendizaje, batchsize – tamaño del lote,
% numepochs – cantidad de épocas de entrenamiento.
opts.alpha = 1;
opts.batchsize = 100;
```

```
i = 1; %guarda el número de iteración

for numepocas = 100:100:2200        % ciclo para ejecutar el entrenamiento hasta 2200 épocas
opts.numepochs = numepocas;

% LLamada a la función cnnsetup para inicializar los parámetros de la red aleatoriamente
% Los parámetros son la estructura de la red, el conjunto de entrenamiento y sus etiquetas
cnn = cnnsetup(cnn, train_x, train_y);

fprintf('Entrenando la CNN...\n');

startTime = tic();

% LLamada a la función de entrenamiento de la CNN.
cnn = cnntrain(cnn, train_x, train_y, opts);

% Impresión del tiempo de entrenamiento
fprintf('...Done. Training took %.2f seconds\n', toc(startTime));

%%=========================================================================
%%  Prueba de la CNN
%%=========================================================================

fprintf('Evaluating test set...\n');

% Evaluación de la CNN entrenada
[er(i), bad] = cnntest(cnn, test_x, test_y);

% Cálculo de la cantidad de muestras clasificadas correctamente.
numRight = size(test_y, 2) - numel(bad);
fprintf('Accuracy: %.2f%%\n', numRight / size(test_y, 2) * 100);

figure(1);
plot(cnn.rL);
title('Mean Squared Error');
xlabel('Training Batch');
ylabel('Mean Squared Error');
end;
figure(2);
plot (er);
```

4) Configuración de la red

La siguiente función se utiliza para inicializar los parámetros de la CNN. Recibe como parámetros la estructura de la CNN, la matriz de muestras de entrenamiento y sus respectivas etiquetas. Regresa la estructura de la CNN con los parámetros inicializados.

```
function net = cnnsetup(net, x, y)

%  net – Estructura  actual de la CNN
%  net.layers{l}.k{i}{j} - Kernel para mapas de salida 'j' de los mapas de entrada 'i'.
%  net.layers{l}.b{j} – Bias para los mapas de salida 'j'.
%  net.ffW, net.ffb – Matriz de pesos y el bias para la capa de salida.
inputmaps = 1;
```

```
mapsize = size(squeeze(x(:, :, 1)));
for l = 1 : numel(net.layers)
    if strcmp(net.layers{l}.type, 's')
        % Calculo del tamaño del mapa de salida para esta capa.
        mapsize = mapsize / net.layers{l}.scale;
        assert(all(floor(mapsize) == mapsize), ['Layer ' num2str(l) ' size must be integer. Actual: '
num2str(mapsize)]);
        % Inicialización del bias para todos los mapas de salida
        for j = 1 : inputmaps
            net.layers{l}.b{j} = 0;
        end
    end

    % Para capa convolutiva...
    if strcmp(net.layers{l}.type, 'c')
        mapsize = mapsize - net.layers{l}.kernelsize + 1;
        fan_out = net.layers{l}.outputmaps * net.layers{l}.kernelsize ^ 2;
        for j = 1 : net.layers{l}.outputmaps
            fan_in = inputmaps * net.layers{l}.kernelsize ^ 2;
            for i = 1 : inputmaps
                net.layers{l}.k{i}{j} = (rand(net.layers{l}.kernelsize) - 0.5) * 2 * sqrt(6 / (fan_in + fan_out));
            end

         net.layers{l}.b{j} = 0;
        end
        inputmaps = net.layers{l}.outputmaps;
    end
end
fvnum = prod(mapsize) * inputmaps;
onum = size(y, 1);
net.ffb = zeros(onum, 1);
net.ffW = (rand(onum, fvnum) - 0.5) * 2 * sqrt(6 / (onum + fvnum));
end
```

5) Entrenamiento de la red

Esta función realiza el entrenamiento de la CNN. Recibe la estructura actual de la CNN, con sus parámetros creados e inicializados, las muestras para el entrenamiento, sus etiquetas y las opciones de número de épocas y tamaño del lote. Retorna la nueva configuración de la red.

```
function net = cnntrain(net, x, y, opts)
m = size(x, 3);
numbatches = m / opts.batchsize;
if rem(numbatches, 1) ~= 0
    error('numbatches not integer');
end

net.rL = [ ];
for i = 1 : opts.numepochs
    disp(['epoch ' num2str(i) '/' num2str(opts.numepochs)]);
    tic;
```

```
    kk = randperm(m);
    for l = 1 : numbatches
        batch_x = x(:, :, kk((l - 1) * opts.batchsize + 1 : l * opts.batchsize));
        batch_y = y(:,    kk((l - 1) * opts.batchsize + 1 : l * opts.batchsize));
        % Evaluación feed-forward de la red actual con el lote de entrenamiento
        %para actualizar las variables de salida de la estructra de la red.
        net = cnnff(net, batch_x);
        % Llamada a la función para el cálculo del gradiente con back-propagation.
        net = cnnbp(net, batch_y);
        % Actualización de los pesos con el gradiente.
        net = cnnapplygrads(net, opts);
        if isempty(net.rL)
            net.rL(1) = net.L;
        end
        net.rL(end + 1) = 0.99 * net.rL(end) + 0.01 * net.L;
    end
    % Impresion del tiempo de entrenamiento de la presente época.
    toc;
  end
end
```

6) Prueba de la red

Esta función lleva a cabo la prueba de la CNN previamente entrenada. Recibe la estructura de la red, las muestras para prueba y sus etiquetas. Regresa el porcentaje de error de las muestras mal clasificadas y los índices de estas muestras. Para la evaluación llama a la función FeedForward.

```
function [er, bad] = cnntest(net, x, y)
net = cnnff(net, x);
 [~, h] = max(net.o);
 [~, a] = max(y);
 bad = find(h ~= a);
 er = numel(bad) / size(y, 2);
end
```

7) FeedForward

Esta función hace la evaluación de la CNN, recibe como parámetros la estructura actual de la CNN y las muestras de prueba. Regresa la nueva estructura de la CNN con sus parámetros actualizados.

```
function net = cnnff(net, x)
n = numel(net.layers);
net.layers{1}.a{1} = x;
inputmaps = 1;
  for l = 2 : n
    if strcmp(net.layers{l}.type, 'c')
      for j = 1 : net.layers{l}.outputmaps

        z = zeros(size(net.layers{l - 1}.a{1}) - [net.layers{l}.kernelsize - 1, net.layers{l}.kernelsize -
1, 0]);
        for i = 1 : inputmaps
```

```
            z = z + convn(net.layers{l - 1}.a{i}, net.layers{l}.k{i}{j}, 'valid');
          end
          net.layers{l}.a{j} = sigm(z + net.layers{l}.b{j});
        end
        inputmaps = net.layers{l}.outputmaps;
      elseif strcmp(net.layers{l}.type, 's')
        for j = 1 : inputmaps
          z = convn(net.layers{l - 1}.a{j}, ones(net.layers{l}.scale) / (net.layers{l}.scale ^ 2), 'valid');
          net.layers{l}.a{j} = z(1 : net.layers{l}.scale : end, 1 : net.layers{l}.scale : end, :);
        end
      end
    end

    net.fv = [ ];
    for j = 1 : numel(net.layers{n}.a)
      sa = size(net.layers{n}.a{j});
      net.fv = [net.fv; reshape(net.layers{n}.a{j}, sa(1) * sa(2), sa(3))];
    end
    net.o = sigm(net.ffW * net.fv + repmat(net.ffb, 1, size(net.fv, 2)));
end
```

8) BackPropagation

Esta función implementa el Back Propagation en la CNN.

```
function net = cnnbp(net, y)
    n = numel(net.layers);
    net.e = net.o - y;
    % función de pérdida
    net.L = 1/2* sum(net.e(:) .^ 2) / size(net.e, 2);

    net.od = net.e .* (net.o .* (1 - net.o));
    net.fvd = (net.ffW' * net.od);
    if strcmp(net.layers{n}.type, 'c')
      net.fvd = net.fvd .* (net.fv .* (1 - net.fv));
    end

    sa = size(net.layers{n}.a{1});
    fvnum = sa(1) * sa(2);
    for j = 1 : numel(net.layers{n}.a)
      net.layers{n}.d{j} = reshape(net.fvd(((j - 1) * fvnum + 1) : j * fvnum, :), sa(1), sa(2), sa(3));
    end
    for l = (n - 1) : -1 : 1
      if strcmp(net.layers{l}.type, 'c')
        for j = 1 : numel(net.layers{l}.a)
                net.layers{l}.d{j} = net.layers{l}.a{j} .* (1 - net.layers{l}.a{j}) .* (expand(net.layers{l +
              1}.d{j}, [net.layers{l +          1}.scale net.layers{l + 1}.scale 1]) / net.layers{l +
              1}.scale ^ 2);
        end
      elseif strcmp(net.layers{l}.type, 's')
        for i = 1 : numel(net.layers{l}.a)
          z = zeros(size(net.layers{l}.a{1}));
          for j = 1 : numel(net.layers{l + 1}.a)
```

```
                z = z + convn(net.layers{l + 1}.d{j}, rot180(net.layers{l + 1}.k{i}{j}), 'full');
            end
            net.layers{l}.d{i} = z;
        end
    end
end
for l = 2 : n
    if strcmp(net.layers{l}.type, 'c')
        for j = 1 : numel(net.layers{l}.a)
            for i = 1 : numel(net.layers{l - 1}.a)
                net.layers{l}.dk{i}{j} = convn(flipall(net.layers{l - 1}.a{i}), net.layers{l}.d{j}, 'valid') /
size(net.layers{l}.d{j}, 3);
            end
            net.layers{l}.db{j} = sum(net.layers{l}.d{j}(:)) / size(net.layers{l}.d{j}, 3);
        end
    end
end
net.dffW = net.od * (net.fv)' / size(net.od, 2);
net.dffb = mean(net.od, 2);
function X = rot180(X)
    X = flipdim(flipdim(X, 1), 2);
end
end
```

9) Aplicación del gradiente

En esta función se implementa el cálculo del gradiente.

```
function net = cnnapplygrads(net, opts)
    for l = 2 : numel(net.layers)
        if strcmp(net.layers{l}.type, 'c')
            for j = 1 : numel(net.layers{l}.a)
                for ii = 1 : numel(net.layers{l - 1}.a)
                    net.layers{l}.k{ii}{j} = net.layers{l}.k{ii}{j} - opts.alpha * net.layers{l}.dk{ii}{j};
                end
                net.layers{l}.b{j} = net.layers{l}.b{j} - opts.alpha * net.layers{l}.db{j};
            end
        end
    end
    net.ffW = net.ffW - opts.alpha * net.dffW;
    net.ffb = net.ffb - opts.alpha * net.dffb;
end
```

Anexo B. Artículo publicado en revista indizada

Desempeño de una Red Neuronal Convolucional para Clasificación de Señales de Tránsito Vehicular

R. Vizcaya Cárdenas[1], J. M. Flores Albino, V. M. Landassuri Moreno y S. Lazcano Salas

Maestría en Ciencias de la Computación, Universidad Autónoma del Estado de México. Centro Universitario UAEM Valle de México., Km. 11.5 Carretera Atizapán de Zaragoza-Nicolás Romero S/N, México.

rvizcayac@uaemex.mx, jmfloresa@uaemex.mx, vmlandassurim@uaemex.mx, slazcanos@uaemex.mx

***Área de participación**: Sistemas Computacionales*

Resumen

El paradigma del **Deep Learnig**, o **Aprendizaje Profundo**, se ha beneficiado del incremento de información de la actualidad, así como del notable avance de las **Redes Neuronales Convolucionales** o **CNN's**. En los últimos cinco años, las CNN's han estado al frente en aplicaciones de reconocimientos de patrones usando imágenes o video, debido a las ventajas que tienen en comparación con otras técnicas; incluso, en algunos casos, llegando a superar la capacidad humana, como se muestra en el trabajo de Graham [2015]. En el presente trabajo se emplean señales de tránsito empleadas en México, para investigar el tiempo de entrenamiento y error de clasificación (desempeño) de una CNN de dos capas de convolución, el entrenamiento y prueba se llevada cabo en un CPU.

Palabras clave: *Redes N*euronales Convolucionales (CNN), Deep Learning, Reconocimiento de Patrones, Señales de Tránsito, Clasificación de Imágenes.

Abstract

The Deep Learning paradigm has benefited from the data increase of today. Convolutional Neural Networks or CNN's, are among the most widely used algorithms of this paradigm. In the last five

[1] Becario CONACYT con No. CVU 712316

years, CNN's have been leading in pattern recognition applications using images or video because of the advantages they have compared to other techniques; even beating human capacity, Graham [2015]*. Traffic signals employed in Mexico are utilized in this article to investigate the training time and classification error (performance) of a CNN. Where the architecture of the network has been fixed with two layers of convolution, and the training and testing are carried out on a CPU.*

Keywords: *Convolutional Neural Networks (CNN), Deep Learning, Pattern Recognition, Traffic Signs, Image Classification*

Introducción

Desde los primeros trabajos sobre **Redes Neuronales Convolucionales** (**CNN**) tales como: LeCun, Bottou, Bengio, & Haffner [1998], se proyectaba que las mismas impactarían de manera notable sobre el desarrollo de la Inteligencia Artificial, concretamente en aplicaciones como el reconocimiento de patrones en imágenes; la primera CNN se diseñó para la tarea de clasificar dígitos escritos a mano (MNIST). Sin embargo, este desarrollo se ve limitado por la capacidad de cómputo disponible en aquellos años, además de que otras técnicas como las Máquinas de Soporte Vectorial en trabajos como Schölkopf, Burges, & Smola [1999], Vapnik [1995] o las llamadas redes neuronales superficiales Neal & Zhang [2006], presentaban resultados de buena calidad, razón por la que las CNN no se estudiaron de manera muy intensa. Sin embargo, no se abandonó la investigación en este y otros sub-campos con el **aprendizaje profundo** o **Deep Learning**, y en particular con las **Redes Neuronales Convolucionales** Simard, Steinkraus, & Platt [2003]. Para 2006, las **CNN's** ya presentaban mejores resultados en cuanto a la precisión y la tasa de error en las tareas de clasificación de dígitos escritos a mano, en comparación con otras técnicas, Ranzato, Poultney, Chopra, & LeCun [2007]. Las desventajas que presentaban fueron tanto en el tiempo de entrenamiento requerido, como en la implementación de la CNN para aplicaciones prácticas, esto debido a la cantidad de parámetros y cálculos a manejar. En ese mismo año se comenzó a utilizar los procesadores gráficos (GPU) para esos propósitos, mostrando un incremento de 4 veces en la velocidad, en comparación con un CPU convencional, Chellapilla, Puri, & Simard, [2006]. A partir de esos resultados, la implementación de los diferentes algoritmos de Deep Learning basados en GPU se volvió una constante Ranzato, Huang, Boureau, & LeCun [2007], Fernandez, Graves, & Schmidhuber [2007], Raina, Madhavan, & Ng, [2009], Ciresan D. C., Meier, Gambardella, & & Schmidhuber [2010], Ciresan D. C., Meier, Masci, Gambardella, & Schmidhuber [2011].

En el caso del reconocimiento de patrones en general, sobre imágenes o video, las CNN dominan el estado del arte desde el 2012, siendo el parteaguas el trabajo realizado en Krizhevsky, Sutskever, & E. Hinton [2012]. En este último trabajo, se entrenó una CNN para clasificar 1.2 millones de imágenes de 227x227 pixeles, entre 1000 clases diferentes. La arquitectura usada para tal propósito utiliza 60 millones de parámetros y 650,000 neuronas, los autores reportan que el tiempo de entrenamiento, fue de poco más de cinco días empleando dos GPUs (tarjetas NVIDIA GTX 580 de 3GB en RAM). Esta red se sometió por primera vez (empleando CNN) en el ImageNet Challenge LSVRC-2010, superando los mejores resultados obtenidos hasta ese entonces con métodos tradicionales. En la **Tabla 1** se muestra parte de dichos resultados.

Tabla 1. Comparación de resultados en el ImageNet Challenge LSVRC-2010, Krizhevsky et al. [2012]

Modelo	Tasa de error Top-1	Tasa de error Top-5
Sparse coding	47.1%	28.2%
SIFT + FVs	45.7%	25.7%
CNN	37.5%	17.0%

En general, esa tendencia de mejora continua con CNN's se ha conservado hasta la actualidad. Las categorías en las que se compite anualmente en el ImageNet Challenge, son: clasificación, detección y localización de objetos en imágenes y video. Los últimos resultados reportados, competencia de 2017, Russakovsky, y otros [2017], indican al equipo ganador con un porcentaje de error en la localización de 6.19% y en la tarea de clasificación de 2.711%. Existen otras competencias similares, como MINIST, CIFAR-10, CIFAR-100, STL-10 y SVHN por mencionar algunas. En el caso de Reconocimiento de Objetos en Imágenes CIFAR-10, compitieron 231 equipos en el último concurso y el equipo ganador implementó una CNN que alcanzó una exactitud de 96.53%, que supera al ser humano estimado en 94% aproximadamente para esta tarea. En dicho concurso, se trata de clasificar 60,000 imágenes a color de 32x32 pixeles en 10 clases; Benenson [2017] muestra los resultados de estas competencias, donde los equipos ganadores utilizan CNN's como principal algoritmo. Otras tareas en las que el estado del arte es dominado por CNN's son: descripción de escenas, clasificación de video, seguimiento de objetos en video, entre otras. Del 2012 a la fecha se ha incrementado la investigación en cuanto a CNN se refiere. En la **Tabla 2** se muestra una comparación de las redes más populares publicadas y sus características.

Tabla 2. Comparación entre diferentes Redes Neuronales Convolucionales.

	# Capas de Convolución	MACCs [x 10^6]	Parámetros [x 10^6]	Activaciones [x 10^6]	ImageNet top-5 error %
AlexNet (2012)	5	1140	62.4	2.4	19.7
Network-in-Network (2013)	12	1100	7.6	4.0	19.0
VGG-16 (2014)	16	15470	138.3	29.0	8.1
GoogLeNet (2015)	22	1600	7.0	10.4	9.2
ResNet-50 (2015)	50	3870	25.6	46.9	7.0
Inception v3 (2016)	48	5710	23.8	32.6	5.6
Inception-ResNet-v2 (2016)	96	9210	31.6	74.5	4.9
SqueezeNet (2016)	18	860	1.2	12.7	19.7

Se observa de la **Tabla 2** que, con los años, se fue incrementando la cantidad de "capas de convolución" empleadas, esto trajo como resultado un mejor desempeño en cuanto al error, pero obviamente un incremento en la cantidad de operaciones multiplicación-acumulación (MACC) necesarias. Aquí solo se muestra la cantidad de capas de convolución que contienen las redes, no así las demás capas, donde la columna de activaciones indica la cantidad de funciones de salida de las neuronas, y en la última columna se muestra el porcentaje de error en la clasificación.

Está claro que las CNN's han tenido un notable éxito en los últimos cinco años, y las podemos encontrar en casi cualquier aplicación de reconocimiento de patrones que involucre imágenes o visión artificial. Una característica importante de estas, es que requieren de varias capas de convolución (entre otras capas), una gran cantidad parámetros y de operaciones a implementar, de ahí la necesidad de usar GPU's u otros dispositivos capaces de procesar en paralelo el cómputo requerido, para poder ser implementas.

El principal objetivo del presente documento es evaluar el desempeño, en cuanto al tiempo de entrenamiento y al error en la clasificación de señales de tránsito vehicular, usadas en México, de una CNN que cuenta con sólo dos capas de convolución y dos de agrupamiento. Lo que se pretende con esto es, determinar si es posible emplear Redes Neuronales Convolucionales con menor cantidad de capas, que las reportadas en el estado del arte, y aun así obtener resultados satisfactorios al entrenarlas y probarlas en un CPU. El entrenamiento y prueba de la red, se hace en un CPU Quad-Core Intel Xeon E5 a 3.7 GHz y 12 GB de memoria RAM, en lugar de algún otro dispositivo con el cuál acelerar el proceso. El banco de imágenes de señales de tránsito se generó

a partir de imágenes obtenidas de la página de la Secretaría de Comunicaciones y Transportes, Secretaria [2017]. A este conjunto inicial, se les hizo un tratamiento para obtener variantes de las mismas y con ello conseguir un conjunto de 8,000 imágenes en total. Estas imágenes se dividieron en ocho clases. La separación de los conjuntos de entrenamiento y prueba se hizo de forma aleatoria, se consideró un criterio de 80% de muestras para entrenamiento y 20% para prueba.

Redes Neuronales Convolucionales

Las redes neuronales convolucionales son una extensión del perceptrón multicapa, con la diferencia de que las CNN's realizan operaciones de "convolución" entre los parámetros y los datos de la red, en lugar de productos punto. Son apropiadas para aplicaciones en las que los datos se encuentran en forma de una rejilla, como matrices, por ejemplo. A diferencia de una Multi Layer Perceptron (MLP), las CNN procesan las imágenes por secciones y han tenido un notable éxito, ver, Lin, Chen, & Yan [2013], Chatfield, Simonyan, Vedaldi, & Zisserman [2014], Szegedy y otros [2015], He, Zhang, Ren, & Sun [2015], Szegedy, Vanhoucke, Ioffe, & Shlens [2015], Szegedy, Ioffe, & Vanhoucke, [2016]. Debido a su arquitectura, que opta por expandir una arquitectura bidimensional de una MLP, por una tridimensional de las CNN's, es que son utilizadas en aplicaciones que involucran visión artificial. El principio es manejar los datos que representan los pixeles en forma de volúmenes, y no como vectores. Las capas básicas para que funcione cualquier CNN son tres: capas de convolución, junto con su función de activación, capas de agrupación y las capas completamente conectadas. En la **Figura 1** se muestra la arquitectura básica de una red neuronal convolucional, en esta, la entrada es una imagen a color de 254x254 pixeles. Le siguen las capas que integran la CNN, se muestran las capas de convolución, agrupamiento, las capas completamente conectadas y la capa de salida.

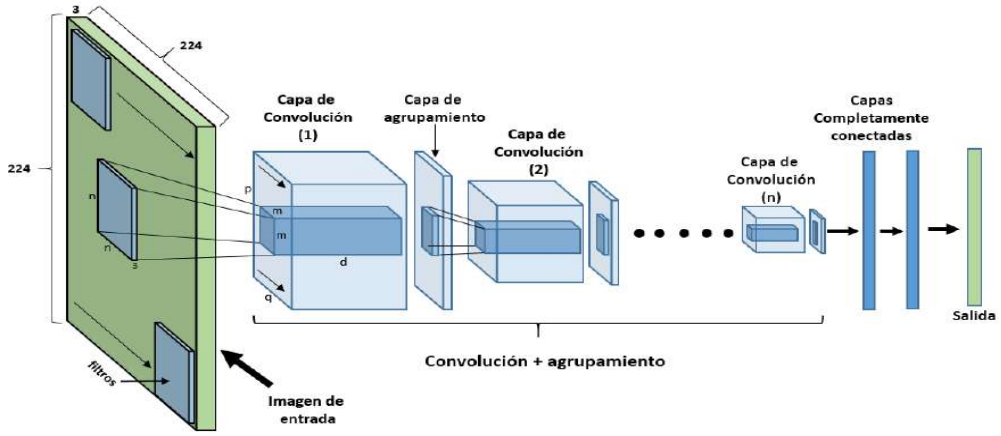

Figura 1. Arquitectura básica de una red neuronal convolucional.

Dependiendo de la aplicación, las últimas capas pueden variar. Para el caso de la tarea de clasificación de imágenes, en la última capa se tendría la misma cantidad de nodos que la cantidad de clases, un nodo por cada clase, los cuáles se activarían con diferente intensidad, indicando el grado de pertenencia de la instancia a la clase. Estos nodos están completamente conectados a cada nodo de la capa anterior.

Capas de convolución

En este concepto de basa el fundamento de las CNN. La operación de convolución, para nuestro caso, es digital y de dos dimensiones, queda definida como:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \qquad (1)$$

Donde I representa la imagen y K un filtro o kernel de tamaño *m x n*. Por lo regular *m* = *n*, de ahí que el tamaño del filtro representado en la **Figura 1** es de *n x n*. Los subíndices *i, j* representan la posición de los pixeles en la imagen sobre la que se hace la operación de convolución. Los filtros consisten en conjuntos de parámetros, típicamente de tamaños: 3x3, 5x5, 7x7 u 11x11, además de que cada filtro debe tener la profundidad de los datos de entrada, por ejemplo, si la entrada es una imagen a color, la profundidad de cada filtro sería tres, que corresponden a los canales (**RGB**). Estos filtros se desplazan barriendo toda la imagen de entrada, desde la esquina superior izquierda, hasta la esquina inferior derecha, como se observa en la **Figura 1**. Conforme se hace el desplazamiento, se van haciendo las operaciones de producto punto entre los valores de los pixeles de la imagen y los que corresponden a los parámetros de los filtros. De la operación de convolución, resulta otro grupo de datos en forma de un volumen, de tamaño (*p* ***x*** *q* ***x*** *d*). Donde *d*, corresponde a la cantidad de filtros usados en la capa anterior. A estos datos se le puede repetir el mismo procedimiento, por varias capas, con nuevos conjuntos de filtros (*m* ***x*** *m* ***x*** *c*), donde *c* corresponde a la cantidad de filtros de la capa anterior. En el transcurso de estas capas, la dimensión de los cubos se va reduciendo en cuanto al ancho y altura, pero la profundidad se conserva igual a la cantidad de filtros usados en la capa anterior.

El propósito de los filtros es producir mapas de activación, o de características, los cuáles se activarán al pasar los filtros por regiones de la imagen que contengan las características buscadas, como bordes, líneas, contornos, etc. El fundamento al entrenar este tipo de redes consiste en actualizar los parámetros de esos filtros, de forma que cada filtro extraiga las correspondientes características buscadas.

En cuanto a las dimensiones del volumen de los datos de salida para cada capa, depende de cuatro factores: **la *cantidad de filtros*** (K) usados en esta capa, ***el paso*** del filtro (S), el ***relleno de ceros*** (P) y el ***campo visual*** (F) del filtro. La *profundidad* de los datos en la salida corresponde a la cantidad de filtros que se está empleando. El *paso* del filtro se refiere a la cantidad de pixeles que se desplazará el filtro al moverse por la imagen. En cuanto al relleno de ceros, en ocasiones se debe rellenar los bordes de una imagen con ceros, de forma que el tamaño de la imagen se ajuste al tamaño de los filtros y se pueda hacer el barrido completo. Si los datos de entrada a una capa de convolución son: $W_1xH_1xD_1$, que corresponden al ancho, altura y profundidad de los datos respectivamente, el volumen de datos en la salida queda determinado por las ecuaciones 2 a 4, para el ancho, altura y profundidad, respectivamente.

$$W_2 = \frac{(W_1 - F + 2P)}{S} + 1 \qquad (2)$$

$$H_2 = \frac{(H_1 - F + 2P)}{S} + 1 \qquad (3)$$

$$D_2 = K \qquad (4)$$

Capas de agrupamiento

El propósito de estas capas es reducir el tamaño espacial de los datos progresivamente, con el fin de reducir la cantidad de parámetros a tratar y consecuentemente la cantidad de cálculos, ayudando a prevenir el overfitting (demasiados parámetros). Estas capas, operan de forma independiente a los datos de entrada de la capa anterior, sólo reducen el tamaño espacial de los datos usando una operación *MAX*, que consiste en dividir los datos en secciones, para extraer los valores máximos de cada sección, discriminando los demás. O bien extrayendo el promedio de este grupo de datos. Estas capas reciben un volumen de entrada de $W_1xH_1xD_1$, que corresponden al ancho, altura y profundidad de los datos de entrada. Producen un volumen de salida $W_2xH_2XD_2$, que corresponden al ancho, altura y profundidad del volumen de salida, estos quedan determinados por las ecuaciones (5) a (7), respectivamente.

$$W_2 = \frac{(W_1 - F)}{S} + 1 \quad (5)$$

$$H_2 = \frac{(H_1 - F)}{S} + 1 \quad (6)$$

$$D_2 = D_1 \quad (7)$$

La **Figura 2** muestra la forma de llevar a cabo esta operación. Para el ejemplo de la figura, a la izquierda se muestra un volumen de datos de entrada de *24x24x6*, y como resultado se obtiene un volumen de *12x12x6*. A la derecha se muestra un ejemplo con la operación *MAX*, como se observa en la figura, $W_1 = 4$, $H_1 = 4$, $F = 2$ y $S = 2$ (algo muy común en la práctica). Según las ecuaciones 5 y 6, W_2 y H_2, que corresponden al ancho y altura de salida respectivamente, es 2. Como se observa, el tamaño espacial de los datos permanece igual en cuanto a la profundidad de los mismos se refiere, pero se reducen en cuanto al ancho y altura.

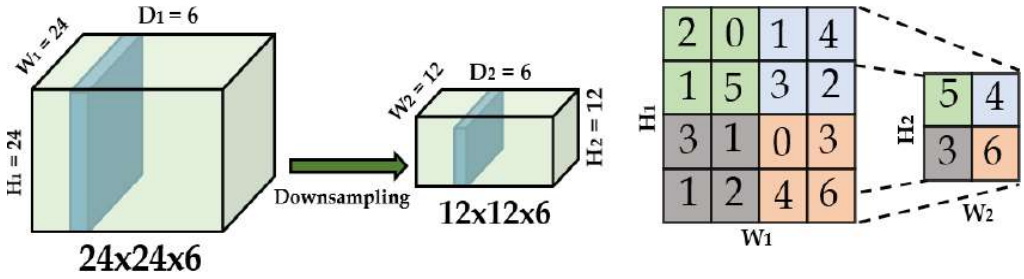

Figura 2. Ejemplo de capa de agrupamiento.

Capas completamente conectadas

Estas funcionan igual que en un MLP, es decir, cada nodo se encuentra completamente conectado a las salidas de la capa anterior. Cada nodo hace una suma ponderada de sus parámetros por el valor de entrada, además de la entrada independiente o bias. Con ello, el que tenga la mayor puntuación, será el que tenga una mayor probabilidad. En sí, hacen una clasificación indicando la probabilidad de cada clase.

Entrenamiento y prueba de la CNN

Para nuestro caso de estudio, el banco de imágenes de señales de tránsito se tomó de la página de la Secretaría de Comunicaciones y Transportes. Estas imágenes se encuentran en un formato PNG a color, con un tamaño de 1249x1249 pixeles. Se acondicionaron las imágenes para adecuarlas a la CNN. El acondicionamiento consistió en pasarlas a escala de grises y escalarlas a 28x28 pixeles. Además, para tener una base de datos lo suficientemente grande y con diversidad, se generaron más imágenes, a partir de las que se tenían. Estas nuevas imágenes se formaron agregando ruido de Poisson y de sal y pimienta al 10% a las originales, rotarlas a 90, 180 y 270 grados, y aplicando un proceso de dilatación. El resultado final es una base de datos con 8,000 imágenes y de ocho clases diferentes. Dado que las imágenes de entrada a la CNN se encuentran en escala de grises, el volumen de entrada en nuestro caso es de 28x28x1. Se emplean seis filtros (K=6) de 5x5, no habrá relleno de ceros, por lo que P = 0 en las ecuaciones 2 y 3. El paso del filtro (S) es 1. Con esto, el volumen de salida será de 24x24x6, según las ecuaciones 2 a 4. Las capas de agrupamiento, o "downsampling", que resultan de las de convolución serán de 12x12x6, según las ecuaciones 5 y 6 (F= S = 2). El proceso se repite con las capas resultantes para obtener nuevos volúmenes de 8x8x12 y 4x4x12, respectivamente. En la **Figura 3** se muestra el proceso y la arquitectura de la CNN implementada. Tanto para la generación de imágenes, como para el entrenamiento y prueba de la CNN, se empleó el software Matlab R2017a® como lenguaje de programación, dada la facilidad de trabajar con imágenes, operaciones con matrices y el toolbox de Deep Learning, Berg Palm [2017].

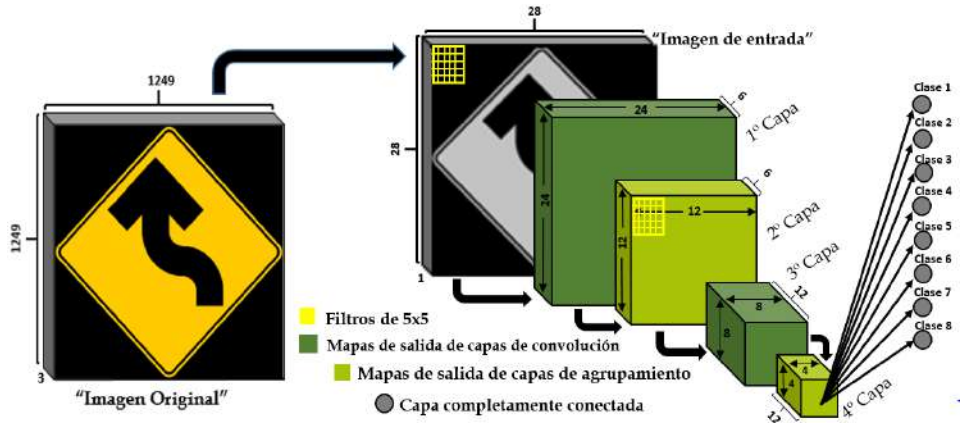

Figura 3. Arquitectura implementada.

Resultados

Para ver el desempeño de la CNN se hicieron varios experimentos. Estos consisten en entrenar la red durante diferentes cantidades de épocas, esto con el propósito de medir el tiempo de entrenamiento y la exactitud para cada caso. El entrenamiento se hace por lotes, para ver el efecto de variar el tamaño del lote que se le presentan a la red durante el entrenamiento, los experimentos anteriores se hicieron para lotes de 32, 100 y 200 muestras, dejando la tasa de aprendizaje fija en todos los casos. En la tabla 3 se muestran los resultados obtenidos hasta 2200 épocas de entrenamiento y para cada tamaño de lote. En el caso de los experimentos con 32 en el tamaño del lote, significa que cada 200 lotes se la muestra el conjunto de 6400 muestras de entrenamiento a la CNN, con lo que se tendría una época de entrenamiento. Para 100 en el tamaño del lote, una época de entrenamiento será cada 64 lotes, y en el caso de 200 cada 32 lotes.

Tabla 3. Resultados obtenidos hasta 2200 épocas.

# de Épocas	Exactitud (%)			Tiempo de entrenamiento (Segundos)			Tiempo Ent./Época (Segundos)		
	L32	L100	L200	L32	L100	L200	L32	L100	L200
200	82.94	73.50	64.25	1623.1	1281.2	**1200.98**	8.12	6.41	**6.00**
600	89.69	84.06	77.56	4870.2	3854.7	**3620.9**	8.12	6.42	**6.03**
1000	88.50	85.75	83.81	8116.3	6420.8	**6032.5**	8.12	6.42	**6.03**
1400	**91.13**	88.75	86.19	**11361.5**	8993.9	**8454.8**	8.12	6.42	**6.04**
1800	89.56	88.50	86.81	14610.4	11554.4	**10887.5**	8.12	6.42	**6.05**
2200	90.19	**89.62**	**88.50**	17842.2	14110.0	**13229.4**	8.11	6.41	**6.01**

En la **Figura 4** se muestra el comportamiento del Error Cuadrático Medio obtenido durante el entrenamiento, para el experimento de 2200 épocas y 100 en el tamaño del lote. Ese mismo comportamiento se observa en los demás casos de experimentación. En la **figura 5** se muestra la tendencia del error en la clasificación al probar la CNN para cada experimento realizado.

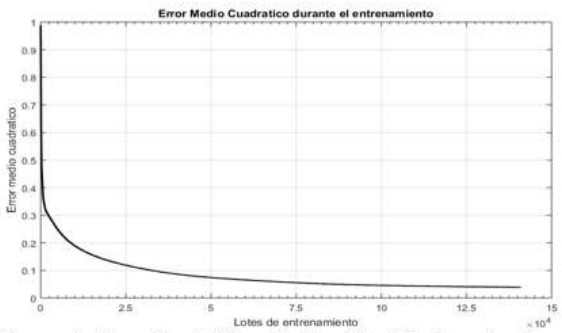

Figura 4. Error Cuadrático Medio obtenido durante el entrenamiento, para el experimento de 2200 épocas y lote de 100 muestras.

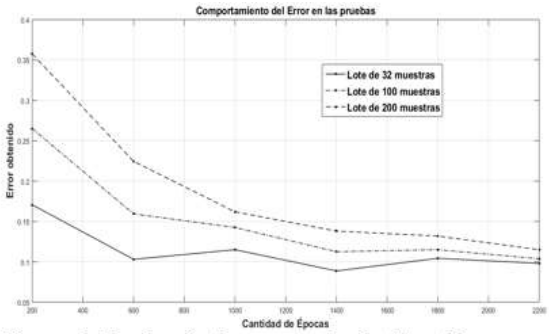

Figura 5. Tendencia de error en la clasificación para cada experimento.

Con el propósito de ver el desempeño de la red para una mayor cantidad de épocas, se hizo un experimento más, el tamaño de los lotes fue de 200 y la cantidad de épocas de 3800. La tendencia del Error Cuadrático Medio fue la misma que la mostrada en la **figura 4**, la exactitud que se alcanzó en este último experimento fue de 90.19%, en un tiempo de 22,947.17 segundos (6.37 horas), un promedio de 6.04 segundos por época.

Los tamaños de los lotes se eligieron tomando en cuenta la cantidad de imágenes de entrenamiento (6400), de la **tabla 3** se observa que con forme se reduce el tamaño del lote, se mejora la exactitud en la clasificación, pero se incrementa el tiempo de entrenamiento. Esto sugiere que para tamaños de 16, 8 o 4 muestras en los lotes, se obtendría un mejor desempeño en cuanto a la exactitud, pero tiempos de entrenamiento más grandes. En cuanto al comportamiento del Error Cuadrático Medio, es el esperado al entrenar cualquier red neuronal.

Conclusiones

El presente trabajo se llevó a cabo con el propósito de evaluar si es posible implementar Redes Neuronales Convolucionales, que tengan pocas capas de convolución y aun así obtener resultados adecuados en la tarea que realizan. Esto tiene la ventaja de poderlas implementar con dispositivos que no requieran de un alto consumo de energía, como es el caso de los GPU's, además de un tamaño reducido. Lo que sería ideal para cualquier aplicación embebida. El error en la clasificación más bajo que se obtuvo fue de 8.87%, cuando el tamaño del lote fue de 32 y el experimento de 1400 épocas (en 3.16 horas de entrenamiento), lo que significa que de cada 100 imágenes de tránsito vehicular que se le presenten a la CNN, esperamos que aproximadamente 91 de ellas las clasifique correctamente. El menor tiempo de entrenamiento se obtiene en los experimentos de 200 en el tamaño del lote, alcanzando 90.19% de exactitud en el experimento de 3800 épocas. Aunque este tipo de problemas están prácticamente resueltos con CNN's (empleando GPUs), según los resultados que se muestran en Benenson [2017], no se puede hacer una comparación directa contra el presente trabajo, debido, por un lado, al tipo y cantidad de imágenes que se tratan en cada caso y, sobre todo, la cantidad de capas de convolución (y otras más) que se manejan. Una referencia directa que si pudiera ser un punto de comparación es con Berg Palm [2017], donde se maneja la misma arquitectura, sólo que ahí la tarea es clasificar imágenes de dígitos escritos a mano (MNIST), el autor reporta una exactitud de 98.8%. La diferencia que se observa puede ser explicada por la mayor diversidad que se tiene, en cuanto a las imágenes que pertenecen a una misma clase. Por ejemplo, para este trabajo, imágenes de vuelta a la izquierda, vuelta a la derecha, un entronque y otras, pertenecen a la misma clase. O los límites de velocidad de 70, 80, 90, 100 km/hr, pertenecen a la misma clase que una señal de stop, o de no rebasar, por ejemplo. Otro factor importante por investigar, es la cantidad de imágenes de entrenamiento en cada caso (6400 contra 60,000), en la literatura se menciona que el desempeño de una CNN mejora con forme se incrementa la cantidad

de información. En cuanto a los tiempos de entrenamiento que se obtuvo en los experimentos, son congruentes con la plataforma que se empleó para este propósito y la configuración de la red obtenida pudiera implementarse en un dispositivo con poca capacidad, en cuanto a los recursos computacionales.

Trabajo futuro

Dada la discusión presentada aquí, existen diversos trabajos pendientes, como el incrementar la cantidad de capas de convolución de la CNN y ver la diferencia en el desempeño, así como incrementar la cantidad y variabilidad de las imágenes. Otro punto, es probar la red con imágenes tomadas en avenidas o autopistas que contengan señales de tránsito, así como probar la configuración obtenida en un sistema embebido.

Referencias

1. Benenson, R. (06 de 03 de 2017). *What is the class of this image ? Discover the current state of the art in objects classification.* Obtenido de Classification datasets results: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#494c5356524332303132207461736b2031
2. Berg Palm, R. (20 de Mayo de 2017). *GitHub Repository.* Recuperado el 12 de Mayo de 2017, de GitHub Repository: https://github.com/rasmusbergpalm/DeepLearnToolbox
3. Chatfield, K., Simonyan, K., Vedaldi, A., & Zisserman, A. (4 de Septiembre de 2014). VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION. *arXiv:1409.1556v6*, 1-14. Obtenido de https://arxiv.org/abs/1409.1556v6
4. Chellapilla, K., Puri, S., & Simard, P. (2006). High performance convolutional neural networks for document processing. En *Tenth International Workshop on Frontiers in Handwriting Recognition.* La Baule, France: Guy Lorette. Obtenido de https://hal.inria.fr/inria-00112631/file/p1038112283956.pdf
5. Ciresan, D. C., Meier, U., Gambardella, L. M., & & Schmidhuber, J. (2010). Deep big simple neural nets for handwritten digit recogntion. *Neural Computation,, 22*(12), 3207-3220.
6. Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., & Schmidhuber, J. (Julio de 2011). Flexible, High Performance Convolutional Neural Networks for Image Classification. *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, 1237-1242. doi:10.5591/978-1-57735-516-8/IJCAI11-210
7. Fernandez, S., Graves, A., & Schmidhuber, J. (2007). Sequence Labelling in Structured Domains with Hierarchical Recurrent Neural Networks. En *Proceedings of the 20th International Joint Conference on Artificial Intelligence.*
8. Graham, B. (2015). Fractional Max-Pooling. *arXiv:1412.6071v4 [cs.CV]*, 1-10.
9. He, K., Zhang, X., Ren, S., & Sun, J. (10 de Diciembre de 2015). Deep Residual Learning for Image Recognition. *arXiv:1512.03385v1*, 1-12. Obtenido de https://arxiv.org/abs/1512.03385
10. Krizhevsky, A., Sutskever, I., & E. Hinton, G. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems 25*, 1097-1105.
11. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 2278--2324.
12. Lin, M., Chen, Q., & Yan, S. (16 de Diciembre de 2013). Network In Network. *arXiv:1312.4400v3*, 1-10. Obtenido de https://arxiv.org/abs/1312.4400
13. Neal, R. M. (2006). Classification with Bayesian neural networks. *Lecture notes in computer science, 3944*, 28-32.
14. Neal, R. M., & Zhang, J. (2006). High dimensional classification with Bayesian neural networks and Dirichlet diffusion trees. En *Feature Extraction. Studies in Fuzziness and Soft Computing*

(Vol. 207, págs. 265-296). Berlin, Heidelberg: Springer. doi:https://doi.org/10.1007/978-3-540-35488-8_11

15. Raina, R., Madhavan, A., & Ng, A. Y. (2009). Large-scale Deep Unsupervised Learning using Graphics Processors. *Proceedings of the 26th annual International Conference on Machine Learning*, 873-880.
16. Ranzato, M. A., Huang, F., Boureau, Y., & LeCun, Y. (2007). Unsupervised learning of invariant feature hierarchies with applications to object recognition. *In Proc. computer vision and pattern recognition conference* , 1-8.
17. Ranzato, M., Poultney, C., Chopra, S., & LeCun, Y. (2007). Efficient learning of sparse representations with an energy-based model. *Advances in Neural Information Processing Systems 19*, 1137-1144. Obtenido de http://papers.nips.cc/paper/3112-efficient-learning-of-sparse-representations-with-an-energy-based-model.pdf
18. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., . . . Fei-Fe, L. (27 de Mayo de 2017). *Large Scale Visual Recognition Challenge 2017 (ILSVRC2017)*. Obtenido de http://image-net.org/challenges/LSVRC/2017/results
19. Schölkopf, B., Burges, C. J., & Smola, A. J. (1999). *Advances in kernel methods: support vector learning.* Cambridge, MA, USA: MIT Press.
20. Secretaria, d. C. (12 de Mayo de 2017). *SCT*. Obtenido de Banco Digital de Señalizaciòn Vial: http://www.sct.gob.mx/bancodigital/
21. Simard, P., Steinkraus, D., & Platt, J. (2003). Best practices for convolutional neural networks applied to visual document analysis. *Seventh international conference on document analysis and recognition*, (págs. 958-963).
22. Szegedy, C., Ioffe, S., & Vanhoucke, V. (23 de Febrero de 2016). Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *arXiv:1602.07261v2*, 1-12. Obtenido de https://arxiv.org/abs/1602.07261
23. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., . . . Rabinovich, A. (2015). Going Deeper with Convolutions. *IEEE Xplore "Open Access version, Computer Vision Fundation"*, 1-9.
24. Szegedy, C., Vanhoucke, V., Ioffe, S., & Shlens, J. (2 de Diciembre de 2015). Rethinking the Inception Architecture for Computer Vision. *arXiv:1512.00567*, 1-10. Obtenido de https://arxiv.org/abs/1512.00567
25. Vapnik, V. N. (1995). *The nature of statistical learning theory.* New York: Springer.