UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

PROGRAMA DE MAESTRÍA Y DOCTORADO EN INGENIERÍA
INGENIERÍA ELÉCTRICA – SISTEMAS ELECTRÓNICOS

IMPLEMENTACIÓN DE UNA RED NEURONAL CONVOLUCIONAL EN UN FPGA PARA LA CLASIFICACIÓN DE PIEZAS DE MANUFACTURA

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN INGENIERÍA

PRESENTA:
ING. RICARDO DAVID SILVA FLORES

TUTOR PRINCIPAL
DR. JUAN MARIO PEÑA CABRERA
IIMAS

CIUDAD UNIVERSITARIA, CDMX, MARZO 2022

JURADO ASIGNADO:

Presidente: Dr. Pérez Alcázar Pablo Roberto

Secretario: Dr. Lomas Barrie Víctor Manuel

1 er. Vocal: Dr. Peña Cabrera Juan Mario

2 do. Vocal: Dr. Neme Castillo José Antonio

3 er. Vocal: Dr. De la Rosa Nieves Saúl

Lugar o lugares donde se realizó la tesis: Facultad de Ingeniería, Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas.

TUTOR DE TESIS:

DR. JUAN MARIO PEÑA CABRERA

FIRMA

AGRADECIMIENTOS

A la Universidad Nacional Autónoma de México por todo lo que me ha dado.

Al Consejo Nacional de Ciencia y Tecnología (CONACYT) por el apoyo brindado durante mis estudios de maestría.

Al Dr. Víctor Manuel Lomas Barrie y al Dr. Juan Mario Peña Cabrera, por todo el apoyo que siempre me dieron, por creer en mí y en lo que era capaz y por luchar conmigo contra las adversidades que nos encontramos a lo largo de este camino.

¡Muchas gracias!

A mis padres, Adriana y Ricardo, por el apoyo incondicional a cada uno de los proyectos que se me ocurren y por los consejos que me han dado, y a mi hermano, Rodrigo, por aguantarme y estar siempre en los momentos importantes, este no es un logro solo mío, sino nuestro.

A mi compañera, a mi amiga, a mi confidente, a mi Cap, que siempre me ha apoyado y me ha dado la fuerza necesaria para seguir adelante, por creer en mí incluso cuando yo no lo hacía, por todo y mucho más.

A mis tíos y padrinos, Jus y Ruth, por abrirme las puertas de su hogar y permitirme ser uno más de su familia, les estaré siempre agradecido.

Resumen

En este trabajo se plantea el diseño e implementación de una Red Neuronal Convolucional (RNC) en un arreglo de compuertas programables de campo (FPGA, por sus siglas en inglés) para la clasificación de piezas de manufactura a partir de un conjunto de descriptores únicos. El diseño parte desde la creación de las piezas y la adquisición del conjunto de datos, atravesando por la experimentación con distintas arquitecturas y métodos de regularización para el entrenamiento de la RNC, y hasta la implementación de cada una de las capas de la red en el FPGA. Además, se realizó una comparación tanto de desempeño como de velocidad de clasificación entre la implementación realizada en Python frente a la implementación en el FPGA.

CONTENIDO

Índice de Figuras

Índice de Tablas

Índice de Gráficas

1 Introducción

A lo largo de la historia, el ser humano ha buscado la manera de obtener una vida más placentera y sencilla, para ello se ha apoyado fuertemente en la tecnología, ya sea utilizándola o desarrollándola. Partiendo desde la Primera Revolución Industrial en el siglo XVIII, donde pasó de fabricarse bienes a mano a la utilización de máquinas, hacia la segunda revolución en el siglo XIX, con la implementación de líneas de producción y producción en masa, llegando a la tercera revolución en el siglo XX, logrando la automatización de los procesos de fabricación con la llegada de las computadoras y los controladores lógicos programables (PLC, por sus siglas en inglés) [1], a la aún temprana Cuarta Revolución Industrial o también llamada Industria 4.0, la cual emerge a través de la integración de sistemas ciberfísicos (CPS, por sus siglas en inglés), el internet de las cosas (IoT, por sus siglas en inglés) y el internet de los servicios (IoS, por sus siglas en inglés) en los procesos de manufactura [2].

Desde la primera revolución industrial, la industria manufacturera ha buscado la manera de reducir sus costos de producción mediante el uso eficiente de estos, sin embargo, siempre ha dependido de los humanos y sus habilidades en distintas áreas para lograrlo, no obstante, la disponibilidad y el uso eficiente de este capital humano han sido algunos de los principales retos con los que la industria se ha enfrentado constantemente. Afortunadamente, el creciente desarrollo en el área de la Inteligencia Artificial (IA) y su implementación en la industria, apunta a ser la solución a estos problemas [3]. A esta integración de técnicas avanzadas de inteligencia en la manufactura para permitir una producción más eficiente, rápida y con una respuesta más dinámica a la demanda se le conoce como manufactura inteligente [4], estas técnicas, sumadas con la integración de sistemas embebidos con la capacidad de incorporar estos modelos, abre un gran abanico de oportunidades y posibilidades para llevar la industria a un nuevo nivel.

1.1 Planteamiento del Problema

Uno de los principales objetivos de la Industria 4.0 es mejorar los niveles de automatización dentro de las líneas de producción, aumentando su productividad y efectividad integrando tecnologías como IoT, IoS, CPS, entre otros [5]. Tomando en cuenta la Industria 4.0 y el gran rendimiento que han demostrado las Redes Neuronales Artificiales (RNA) en los últimos años, es necesario desarrollar implementaciones más eficientes, compactas y asequibles con el fin de mejorar los procesos de manufactura actuales, de tal manera que estas mejoras se puedan ver reflejadas en beneficios económicos y productos o servicios de calidad.

1.2 Justificación

Por lo anterior, se pretende diseñar un modelo de Red Neuronal Convolucional (RNC) implementada en un FPGA para la clasificación de piezas de manufactura de cuerpo rígido mediante vídeo, a partir de una familia de descriptores únicos. Dicha implementación lograría mejorar los procesos de producción al realizar tareas de clasificación en un menor tiempo y a través de un dispositivo más compacto.

Modificando las dimensiones de extracción de características dentro de la red neuronal e integrando la familia de descriptores únicos, se busca mejorar la velocidad de clasificación, así como

también disminuir el número de operaciones necesarias, dando como resultado una red más liviana y eficiente capaz de implementarse en un FPGA, con las ventajas que estos dispositivos conllevan.

1.3 Hipótesis

La implementación de una red neuronal convolucional, en un FPGA, para la clasificación de piezas de manufactura, a través de visión computacional, puede generar un impacto positivo en distintos aspectos como la velocidad de clasificación, consumo de energía y estructura del sistema.

1.4 Objetivo

Diseñar la arquitectura de una red neuronal convolucional que sea capaz de clasificar piezas de manufactura a través de visión por computadora y mediante la utilización de un conjunto de descriptores únicos, implementada en un FPGA.

1.5 Objetivos Específicos

- Diseñar un entorno controlado para la adquisición de imágenes de las piezas a clasificar.
- Establecer la cantidad de imágenes necesarias para la clasificación de los objetos y las propiedades que éstas tendrán.
- Analizar y determinar distintos modelos de RNC para elegir la que mejor rendimiento tenga.
- Integrar la familia de descriptores únicos con la RNC elegida para el entrenamiento y optimización.
- Diseñar la arquitectura de la RNC seleccionada y transferir los parámetros de aprendizaje al FPGA manteniendo una exactitud mayor al 90%.
- Realizar pruebas y analizar los resultados para llevar a cabo los ajustes necesarios.

1.6 Metodología

- Investigación sobre distintas cámaras por USB.
- Diseño de ambiente controlado para la adquisición de imágenes.
- Impresión 3D de los objetos a clasificar.
- Diseño de algoritmos para la captura de imágenes de los objetos y la extracción del conjunto de vectores descriptivos.
- Curso especializado de aprendizaje profundo.
- Investigación de distintas redes neuronales artificiales.
- Diseño e implementación por software de la red neuronal elegida.
- Entrenamiento y regularización de la red neuronal elegida con los conjuntos de vectores obtenidos de los objetos.
- Investigación de la tarjeta de desarrollo donde se implementará la red neuronal entrenada.
- Implementación de la red neuronal en la tarjeta de desarrollo.
- Experimentos y pruebas finales.

1.7 Estado del Arte

En los últimos años, investigaciones en redes neuronales artificiales han demostrado ventajas significativas en el aprendizaje de máquina en comparación con la programación tradicional donde el enfoque, la implementación y ejecución dependen en gran medida del desarrollador [6]. Las redes neuronales han sido ampliamente utilizadas en las áreas de reconocimiento de patrones en imágenes, voz y vídeo, sin embargo, la alta complejidad de cálculo y almacenamiento representan un gran obstáculo para su implementación y aplicación, por esta razón, el diseño de redes neuronales basadas en FPGA se ha convertido en un tema de investigación de gran interés. Con hardware diseñado específicamente, los FPGA pueden ser la posible solución que supere a las unidades de procesamiento gráfico (GPU, por sus siglas en inglés) en cuanto a velocidad, eficiencia energética, tamaño y peso [7]. A continuación, se presentan algunos de los trabajos más relevantes en la implementación de RNC enfocadas en FPGA:

En el 2017 [8], Marco Bettoni et al. desarrollaron una RNC a partir del modelo AlexNet para implementarlo en el FPGA Stratix V de Altera, el cual se encuentra embebido en la tarjeta DE5-Net de Terasic. A esta red la llamaron HW-CNN y tiene el propósito de reconocer distintas imágenes.

Compararon su RNC implementada en el FPGA con AlexNet, la cual fue implementada en una computadora de propósito general, y un sistema en chip (SoC, por sus siglas en inglés) GPU. Tomaron en cuenta el tiempo de desempeño y eficiencia energética de cada uno de ellos, llegando a la conclusión que en ambos ámbitos la implementación en FPGA obtuvo mejores resultados en comparación con el SoC GPU, sin embargo, con respecto a la computadora de propósito general los tiempos de desempeño fueron casi idénticos, pero teniendo una mayor eficiencia energética el FPGA.

En su propuesta de investigación de principios del 2018 [9], K. Abdelouahab et al. proporcionaron un estudio de distintas RNC aceleradas a través de FPGA. Analizan la carga de trabajo computacional, su paralelismo y el acceso a memoria implicado. A nivel de las neuronas, explican las optimizaciones de las capas convolucionales y las capas totalmente conectadas y comparan el desempeño de los diferentes métodos. En general, los métodos y herramientas investigadas en este estudio, como la optimización de algoritmos y rutas de datos, representan las recientes tendencias en la utilización de FPGA para la aceleración de las RNC.

En la segunda mitad del 2018 [10], R. A. Solovyev et al. desarrollaron una RNC a partir de la arquitectura de la red VGG, la cual llamaron *LWDD* (Low Weight Digit Detector, por su nombre en inglés), esta red fue entrenada para el reconocimiento de dígitos escritos a mano mediante el conjunto de datos llamado *MNIST* y fue implementada en la tarjeta de desarrollo DE0-Nano de Terasic, logrando una exactitud mayor al 95%.

En 2018, en la Conferencia Internacional en Comunicaciones en China (ICCC) [12], Guangju Wei et al. implementaron la red neuronal YOLO (You Only Look Once: Unified, Real-Time Object Detection, por su nombre en inglés) en un SoC de la familia Zynq-7000 de Xilinx para la detección de autos y peatones. Compararon el desempeño de la misma red en dos plataformas distintas: el

FPGA y una computadora de propósito general, resultando ser 46.9 veces más veloz la implementación en FPGA.

En el artículo publicado por Justin Sánchez et al. en 2018 [14], proponen una arquitectura de RNC para los dispositivos *on the edge*[1] utilizando la convolución de una sola dimensión, tomaron como base la RNC SqueezeNet y la implementaron en un FPGA de Xilinx Zynq-7000. Demostrando que la optimización a una dimensión de la convolución, en comparación con la de dos dimensiones, es 7.3 veces mayor y, en comparación con su implementación en la tarjeta de Nvidia Jetson TX2, 4.3 veces. Además, solo presentó una pérdida del 2% de la exactitud en su implementación en el FPGA.

En 2019, en la tercera Conferencia Internacional en Circuitos, Sistemas y Simulación (ICCSS) [13], Dai Rongshi et al. implementaron la RNC LeNet-5 para reconocer dígitos escritos a mano o impresos en el FPGA Zybo Z7-20 de Digilent. Con el objetivo de optimizar esta red y cumplir con los requerimientos de desempeño y eficiencia energética que algunas plataformas de *front-end*[2] requieren, implementaron este modelo a través de la herramienta Vivado® High-Level Synthesis (VHLS) donde la unidad lógica programable se encarga de ejecutar las operaciones y el sistema de procesamiento de suministrar los datos a la unidad lógica. Compararon el tiempo de ejecución de su implementación en FPGA con una computadora de propósito general, obteniendo una velocidad de clasificación 4.7 veces mayor en comparación con la computadora.

A finales del 2019 [15], Jinwei Xu et al. publicaron un artículo sobre un sistema simplificado para el reconocimiento de voz basado en una RNC de una dimensión e implementada en el FPGA de Xilinx VC709. Tomaron como base el modelo ResNet20 y lograron reducir su complejidad computacional en un 64% y la cantidad de parámetros en un 53%, afectando tan solo el 1% de la exactitud de reconocimiento, además, lograron superar en velocidad de reconocimiento a una computadora de propósito general siendo, por lo menos, 5 veces más rápida la implementación en FPGA.

Por otro lado, en 2020 David Gschwend diseñó en su trabajo de tesis de maestría [11] una RNC a partir de la topología de SqueezeNet, la cual llamó ZynqNet CNN, desarrollada para el *SoC* de Xilinx Zynq XC-7Z045. Esta red se probó mediante el reto de ImageNet[3] logrando una exactitud del 84.6%.

[1] Los dispositivos *on the edge* o en el borde son aquellos los cuales se encargan de controlar la conexión y el flujo de datos entre dos redes.

[2] Front-end es toda la parte de una aplicación o plataforma que tiene que ver con la interacción con los usuarios.

[3] El reto ImageNet es una competencia anual donde los participantes desarrollan algoritmos para clasificar imágenes a partir de un subconjunto de la base de datos de ImageNet.

2 Marco Teórico

Los seres humanos somos seres predominantemente visuales, dependemos en gran medida de nuestra visión para dar sentido al mundo que nos rodea, no solo observamos las cosas para poder identificarlas y clasificarlas, sino también para buscar diferencias y obtener una idea general y aproximada de una escena. Nuestro sentido visual ha evolucionado de tal manera que podemos reconocer una cara al instante, diferenciar colores y procesar una gran cantidad de información visual de manera muy rápida [16].

Para nuestros propósitos, una imagen es una representación visual en dos dimensiones de una escena en tres dimensiones la cual representa algo, puede ser una imagen de una persona, un animal, una microfotografía de un componente electrónico o una imagen médica [17], [18].

2.1 Procesamiento Digital de Imágenes

El procesamiento digital de imágenes es utilizado con el fin de mejorar las imágenes mediante la modificación de sus características como: el contraste, el color, formas, entre otras, a través de una computadora para que estas puedan ser más claras en cuanto a su información visual para la interpretación humana o más adecuadas para el análisis e interpretación a través de una computadora [18].

Sin embargo, si un procedimiento puede mejorar una imagen para la interpretación humana, no necesariamente puede funcionar para el caso de la computadora y viceversa.

Para ello, es necesario realizar un análisis de la imagen a través de algoritmos y métodos con los cuales se puedan extraer características importantes que la computadora pueda examinar, interpretar y obtener un resultado, un ejemplo se muestra en la Figura 1.

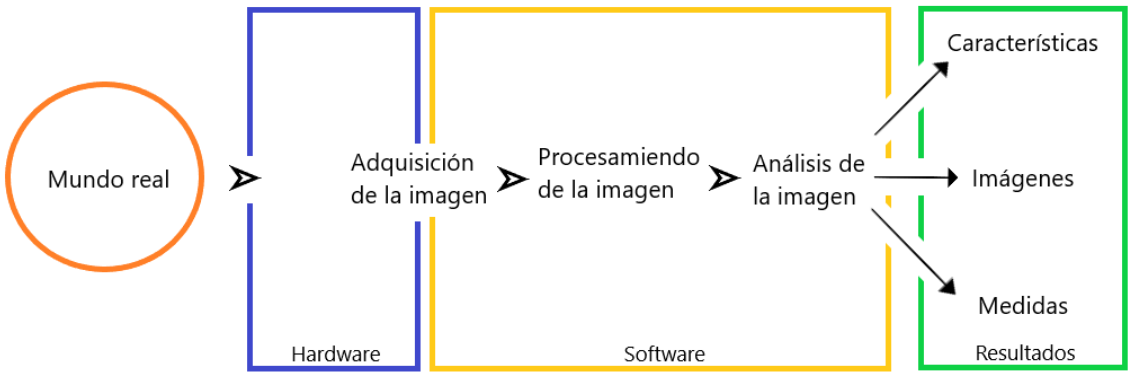


Figura 1. Principios del análisis de imágenes. Fuente: Elaboración propia.

2.1.1 Clasificación Digital de Imágenes

Un humano que intenta clasificar una imagen o el contenido dentro de ella lo logra mediante la identificación de grupos de píxeles homogéneos que comparten características en común, diferentes imágenes u objetos que poseen distintos grupos de características que los definen.

El objetivo de la clasificación digital de imágenes es categorizar automáticamente cada uno de los píxeles que conforman una imagen en clases. Para ello es necesaria la utilización de un clasificador, el cual es un programa en computadora que implementa ciertos procedimientos para lograr clasificar la imagen.

Los métodos tradicionales de clasificación siguen dos enfoques:

- Clasificación supervisada.
  - Es un enfoque que se define por el uso de conjuntos de datos etiquetados. Estos conjuntos son utilizados para entrenar o supervisar algoritmos para clasificar datos o predecir resultados.
- Clasificación sin supervisión.
  - Con este enfoque, los algoritmos analizan y agrupan conjuntos de datos sin etiquetar sin la necesidad de la intervención humana.

Una tarea fundamental en la clasificación es la selección correcta de las características a extraer ya que, si se elige cuidadosamente un buen conjunto de ellas, se puede lograr una clasificación con una mayor confiabilidad y un menor esfuerzo [17].

Algunos métodos utilizados para la clasificación son:

- K-NN (K-Nearest Neighbors, por su nombre en inglés).
- MDM (Minimum Distance to Mean, por su nombre en inglés).
- Árboles de decisión.
- Máxima verosimilitud.
- Clasificación Bayesiana.
- K-Means.

2.2 Inteligencia Artificial

La IA se puede definir como la ciencia e ingeniería que consiste en crear máquinas o programas informáticos inteligentes que aprenden sin ser explícitamente programados. [19]. Dentro del campo de la IA existe un área llamada aprendizaje de máquina, la cual se encarga de dotar a las computadoras de la habilidad de aprender a través de datos y algoritmos sin ser explícitamente programadas. Dentro del aprendizaje de máquina se encuentra una subárea conocida como aprendizaje profundo, la cual incorpora modelos computacionales y algoritmos que imitan la arquitectura de las redes neuronales biológicas en el cerebro para la resolución de problemas [20], ver Figura 2.

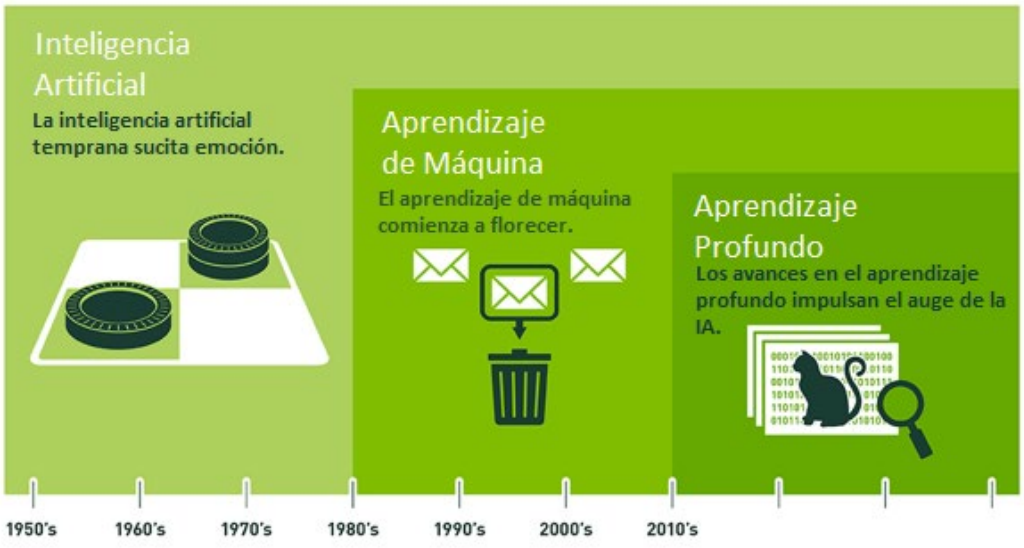


Figura 2. La inteligencia artificial y sus ramas. Fuente: https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/ (Último acceso: 21/06/20).

2.3 Redes Neuronales Artificiales

Una Red Neuronal Artificial puede considerarse como un modelo simplificado de la estructura de una red neuronal biológica. Consta de muchas unidades de procesamiento interconectadas llamadas neuronas, cada una produciendo una secuencia de activaciones. Las neuronas de entrada son activadas a través de sensores que perciben el entorno, otras neuronas son activadas a través de conexiones ponderadas de neuronas previamente activas, ver Figura 3. El aprendizaje consta de encontrar los pesos que hagan que la red tenga un comportamiento deseado [21].

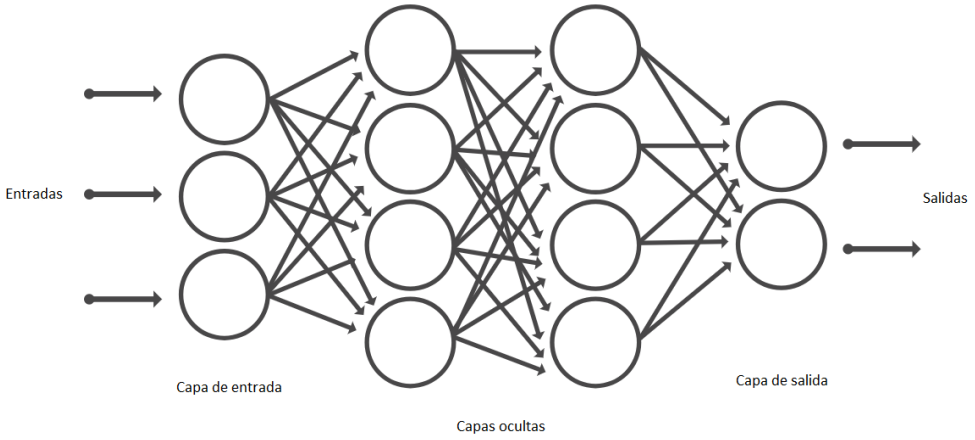


Figura 3. Estructura de una red neuronal artificial con 3 capas totalmente conectadas, 3 entradas y dos salidas. Fuente: https://la.mathworks.com/solutions/deep-learning/convolutional-neural-network.html (Último acceso: 21/06/20).

De la misma manera que en el tema 2.1.1, para las RNA también se tienen dos paradigmas de aprendizaje, el aprendizaje supervisado y el aprendizaje no supervisado [22].

2.3.1 Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales son análogas a las tradicionales RNA ya que están constituidas por neuronas que ajustan sus pesos a través del entrenamiento, la única diferencia notable es que las RNC son principalmente utilizadas en el campo de reconocimiento de patrones en imágenes, ya que eliminan la necesidad de extraer manualmente las características de la imagen y se pueden volver a entrenar para nuevas tareas de reconocimiento a partir de redes ya existentes [23].

La arquitectura de una RNC se compone apilando cierto número de capas las cuales transforman mapas de características de entrada con ciertas dimensiones de altura, ancho y profundidad $(h_{en} \times w_{en} \times ch_{en})$ en mapas de características de salida con una distinta dimensión $(h_{sal} \times w_{sal} \times ch_{sal})$. Una RNC se compone básicamente por tres tipos de capas: convolucionales, de agrupamiento (pooling, por su nombre en inglés) y totalmente conectadas, ver Figura 4.

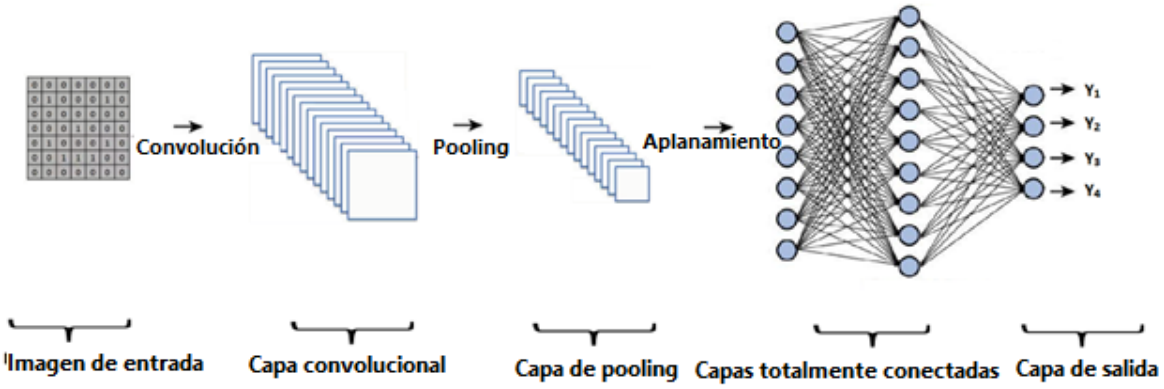


Figura 4. Estructura de una red neuronal convolucional con 4 capas, 1 convolucional, 2 totalmente conectadas y 1 de salida. Fuente: applsci-10-01245-g002.png (3560×1304) (mdpi.com) (Último acceso: 22/03/21).

Capa Convolucional

En la capa convolucional se aplican filtros convolucionales de tamaño $(k \times k)$ para generar un mapa de características de salida al cual se le aplicará una función de activación no lineal, como la función sigmoide, la unidad lineal rectificada (ReLU, por sus siglas en inglés) o la tangente hiperbólica, ver las ecuaciones (1), (2) y (3), respectivamente. Para filtros mayores a (1×1) las dimensiones del mapa de salida se ven reducidas en comparación con las del mapa de entrada, por lo cual, para evitar este efecto, al mapa de entrada se le agrega un marco de ceros (padding, por su nombre en inglés) para mantener las mismas dimensiones, además, al aplicar los filtros se puede

determinar el paso que tendrán (stride, por su nombre en inglés), entre mayor sea el stride menor será la dimensión del mapa de salida.

$$Sigmoide: \quad f(x) = \frac{1}{(1 + e^{-x})} \rightarrow x \in \mathbb{R} \tag{1}$$

$$ReLu: \quad f(x) = Max(0, x) \rightarrow x \in \mathbb{R} \tag{2}$$

$$Tangente\ hiperbólica: \quad f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \rightarrow x \in \mathbb{R} \tag{3}$$

Capa de Pooling

La capa de pooling se encarga de reducir las dimensiones del mapa de características de salida proveniente de la capa convolucional, con el objetivo de conservar las características más significativas. Dos opciones de pooling muy utilizadas son el max-pooling y avg-pooling, los cuales toman el valor máximo o el valor promedio de un área local dentro del mapa, respectivamente. El área local es de tamaño $(n \times n)$ y también se puede elegir su stride como en la capa convolucional.

Capa Totalmente Conectada

Esta capa es normalmente utilizada en las últimas capas de una RNC para calcular las puntuaciones de clase de las activaciones y ser utilizadas para la clasificación. Generalmente, estas capas son las que contienen la mayor cantidad de pesos en una RNC.

Entrenamiento

Para el entrenamiento de redes neuronales convolucionales es necesario un alto poder computacional con el fin de agilizar esta etapa y lograrlo en el menor tiempo posible, sin embargo, las computadoras modernas, con frecuencia, no son lo suficientemente rápidas y hardware especializado, tales como las GPU, son necesarias [24].

2.4 BOF

La función de frontera de objeto (BOF, por sus siglas en inglés), es un descriptor único utilizado principalmente en la manufactura avanzada para el reconocimiento de piezas el cual consiste en calcular el contorno de un objeto para su reconocimiento [26]. Propiedades como el perímetro, área, centroide y distancias entre el centroide y los puntos que conforman el contorno del objeto son necesarias para calcular la BOF.

Cabe resaltar que, aunque es un método relativamente sencillo no aplica para cualquier tipo de objetos, sino para aquellos cuya forma no es tan elaborada ya que al trazar los vectores desde el centroide hacia el contorno estos pueden quedar fuera del objeto, como se muestra en la Figura 5.

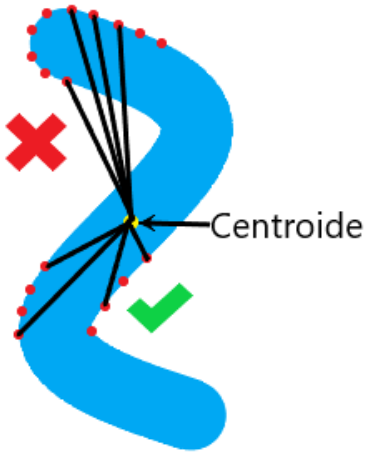


Figura 5. Ejemplo de la forma de un objeto donde los vectores lo atraviesan. Fuente: Elaboración propia.

Como primer paso, es necesario localizar al objeto dentro de la imagen, esto se logra mediante el análisis píxel a píxel de izquierda a derecha y de arriba hacia abajo, por lo tanto, el objeto que se encuentre más arriba será el primero en ser encontrado.

Perímetro

El perímetro será el conjunto de píxeles que conformen el contorno del objeto y se representará mediante la suma de cada uno de ellos, como se muestra en (4).

$$P = \sum_i \sum_j píxeles(i,j) \in contorno \qquad (4)$$

Área

El área del objeto se define como el conjunto de píxeles que conforman la totalidad del objeto, la cual se puede calcular mediante (5).

$$A = \sum_i \sum_j píxeles(i,j) \in forma \qquad (5)$$

Centroide

El centro de masa de una figura bidimensional se conforma por un par de coordenadas (x_c, y_c), en las cuales se considera que toda la masa se concentra y también donde todas las fuerzas resultantes actúan. Matemáticamente, el centroide se puede definir como se muestra en (6).

$$x_c = \frac{1}{A} \sum_i \sum_j i \qquad y_c = \frac{1}{A} \sum_i \sum_j j \qquad (6)$$

Distancia del Centroide al Contorno

Las distancias entre el centroide y los distintos puntos que conforman el contorno del objeto proporcionarán la información que conformará la BOF del objeto. Si $P_1(x_1, y_1)$ son las coordenadas del centroide (x_c, y_c) y $P_2(x_2, y_2)$ es un punto del perímetro, la distancia entre ambos se puede determinar a través de (7).

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \qquad (7)$$

Vector descriptivo BOF

El vector descriptivo se conforma por 180 elementos correspondientes a las distancias entre el centroide y 180 puntos, separados cada 2 grados, del perímetro del objeto. Estos puntos, además, son normalizados dividiéndolos entre la distancia mayor encontrada dentro de los 180, como se muestra en la Figura 6.

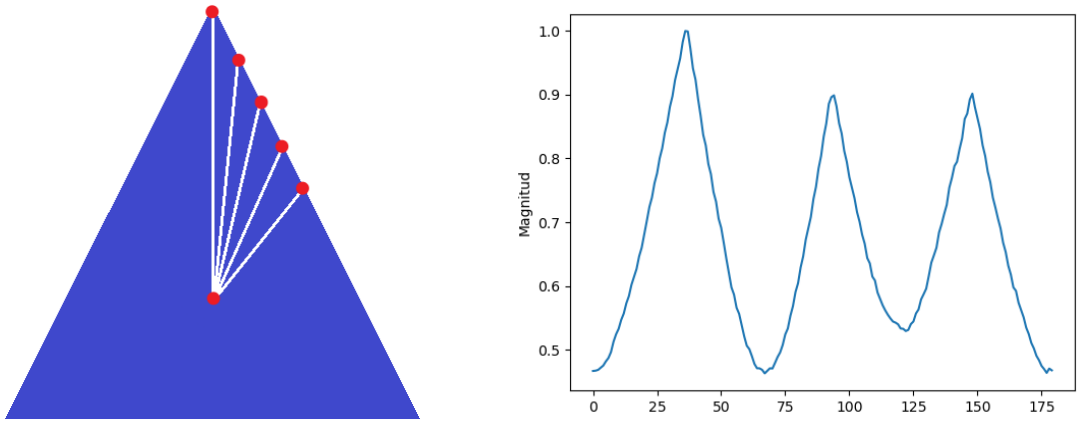


Figura 6. BOF de un triángulo. Fuente: Elaboración propia.

Cabe resaltar que, al normalizar el vector descriptivo, se logra que este sea invariante a la rotación, traslación y escalamiento [26].

2.5 FPGA

Los FPGA son dispositivos semiconductores basados en matrices de bloques lógicos configurables (CLB, por sus siglas en inglés) conectados a través de interconexiones programables. Los FPGA pueden ser reconfigurados para adaptarse a los requerimientos o funcionalidades de alguna aplicación, está característica los distingue de los circuitos integrados de aplicación específica (ASIC, por sus siglas en inglés), los cuales son fabricados a la medida para tareas específicas sin la posibilidad de reconfigurarlos. La configuración se mantiene típicamente en la memoria estática de acceso aleatorio (SRAM, por sus siglas en inglés), permitiendo que los FPGA puedan ser reconfigurados gran número de veces [27].

FPGA vs Procesadores de Propósito General

Una de las ventajas de los sistemas basados en FPGA por sobre las computadoras de escritorio, teléfonos inteligentes, sistemas embebidos e incluso GPU, es la posibilidad de poder configurar libremente los CLB. Estos pueden ser diseñados para realizar tareas específicas, acelerando la velocidad de procesamiento, mejorando el rendimiento y ahorrando energía, todo esto gracias a la naturaleza de procesamiento en paralelo que el FPGA posee. Sin embargo, esto conlleva la desventaja de aumentar la complejidad durante el desarrollo, dejando la tarea al diseñador de administrar con cuidado los recursos disponibles [28].

FPGA vs ASIC

Los circuitos integrados ASIC son circuitos diseñados para desempeñar una tarea en particular, resultando en sistemas más pequeños, rápidos y energéticamente eficientes, sin embargo, es necesaria una ardua etapa de desarrollo y evaluación antes de salir al mercado ya que, una vez fabricados, no pueden ser modificados y nuevos cambios requerirían la fabricación de nuevos circuitos. Los FPGA, con la posibilidad de reconfigurarlos, son más adecuados para la etapa de prototipado y desarrollos de corto plazo[28].

Aplicaciones

Debido a su naturaleza reconfigurable, los FPGA son ideales para distintas aplicaciones [27], como son:

- Aeroespacial y defensa.
- Prototipado ASIC.
- Sistemas automotrices.
- Electrónica de consumo.
- Centros de datos.
- Industrial.
- Médicas.
- Vídeo y procesamiento de imágenes.
- Comunicaciones alámbricas e inalámbricas.

2.5.1 Síntesis de Alto Nivel

Tradicionalmente, los FPGA son configurados utilizando un lenguaje de descripción de hardware (HDL, por sus siglas en inglés) tales como VHDL o Verilog. Estos lenguajes son utilizados con dos propósitos: simulación de diseños electrónicos y síntesis de dichos diseños, los cuales en su mayoría son descritos a nivel de transferencia de registro (RTL, por sus siglas en inglés), donde el diseñador especifica su algoritmo utilizando distintos procesos en paralelo. La síntesis es un proceso por el cual uno de estos lenguajes es condensado y mapeado hacia su implementación en hardware en un FPGA o un ASIC. Las descripciones RTL son muy parecidas a las compuertas lógicas y cableados que componen a los FPGA o ASIC, por lo que el hardware resultante de la síntesis a partir del diseño RTL puede ser estrechamente controlado, sin embargo, este proceso puede complicar el diseño y

es propenso a generar errores demandando intuición, experiencia y conocimiento por parte de los diseñadores [11], [29].

En contraparte, la Síntesis de Alto Nivel HLS es un método el cual permite que los diseñadores empleen sus algoritmos mediante un lenguaje de programación de alto nivel, tales como C, C++ o SystemC, facilitando y reduciendo el tiempo de desarrollo, pero comprometiendo el desempeño y la eficiencia que a través de RTL se podría obtener. Las implementaciones HLS son abstraídas y manejadas por el compilador, el cual transforma la descripción del algoritmo secuencial en una descripción de hardware a nivel de RTL [30].

La herramienta de Xilinx® Vivado® High-Level Synthesis es uno de los compiladores más populares a nivel comercial. Con VHLS, los diseñadores pueden utilizar ciclos, arreglos, estructuras, variables flotantes, la mayoría de las operaciones aritméticas, llamadas a funciones e incluso clases orientadas a objetos. Estos son transformados automáticamente en contadores, memorias y protocolos en conjunto con máquinas de estados. Las operaciones son programadas de tal manera que su ejecución se lleve a cabo lo más pronto posible. No obstante, aunque es una herramienta realmente útil para reducir el tiempo de desarrollo de manera eficiente, los resultados que se pueden alcanzar son altamente dependientes del estilo de programación del diseñador, además, el decantarse por este método conlleva un riesgo no despreciable ya que las fallas y deficiencias en el compilador solo se descubren durante el diseño [31], [11].

3 Adquisición de datos y BOF

Mediante la implementación de una RNC se conseguiría clasificar objetos con base en un conjunto de vectores que los describen, extraídos a través de la BOF. Se podría diseñar un sistema que capture imágenes de los objetos, extraiga su vector descriptivo y mediante una RNC adaptada para las dimensiones de ese vector, clasificar el objeto. Esto genera una ventaja ya que, al interesarnos solo el vector descriptivo, el tamaño de la imagen deja de ser tan relevante, a diferencia de las tradicionales RNC que emplean tamaños de imágenes reducidos, como se puede observar en la Tabla 1.

Modelo	**LeNet-5**	**AlexNet**	**VGG-16**	**GoogLeNet**	**ResNet-50(v1)**	**BOF + CNN**
Tamaño (píxeles)	32 x 32	227 x 227	231 x 231	224 x 224	224 x 224	N/A
Total (bytes)	1,024	51,529	53,361	50,176	50,176	180 * (#bits en punto fijo)

Tabla 1. Tamaños de imágenes con los que trabajan distintas RNC (asumiendo que las imágenes se encuentran en escala de grises a 8 bits).

Para lograrlo, se plantearon los siguientes pasos:

1. Determinar un conjunto de prueba compuesto por objetos comunes en la manufactura.
2. Crear un conjunto de imágenes por cada objeto.
3. Extraer el vector descriptivo de cada imagen para formar un conjunto de vectores por objeto.
4. Diseñar y entrenar una RNC con base en el conjunto de vectores extraídos.
5. Evaluar el entrenamiento y optimizar la red.

3.1 Adquisición del Conjunto de Datos

Se diseñaron 19 objetos tridimensionales mediante la herramienta de software Tinkercad de Autodesk. Tinkercad es una plataforma en línea en la cual se pueden diseñar modelos en 3D para su posterior impresión, en la Figura 7 se muestran algunos de los objetos diseñados con esta herramienta. Una vez diseñados los objetos, estos se importan para su creación mediante impresión 3D[4], algunos de ellos se pueden apreciar en la Figura 8.

[4] La impresión en 3D es una técnica de fabricación por adición donde un objeto tridimensional es creado mediante la superposición de capas sucesivas de material.

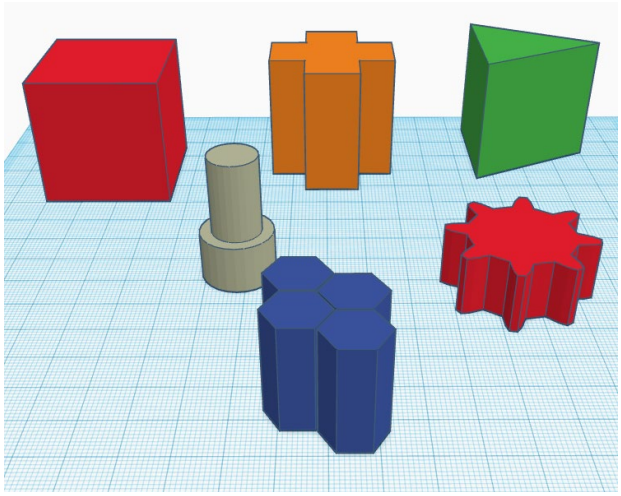


Figura 7. Objetos diseñados con Tinkercad. Fuente: Elaboración propia a través de tinkercad.com.

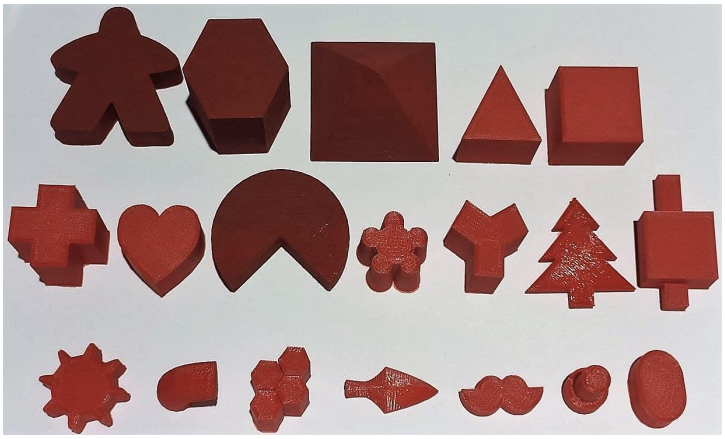


Figura 8. Figuras impresas mediante impresión 3D. Fuente: Elaboración propia.

Una vez impresos todos los objetos, se prosiguió a la obtención del conjunto de imágenes de cada uno, para ello se tomó la siguiente estrategia con el objetivo de poder reconocer al objeto en cualquier posición que se encontrase éste o la cámara: si el objeto es rotado sobre su propio eje cada α grados y, además, se captura una imagen de él cada θ grados, como se muestra en la Figura 9, se puede formar un conjunto de imágenes que representen al objeto visto desde diferentes ángulos.

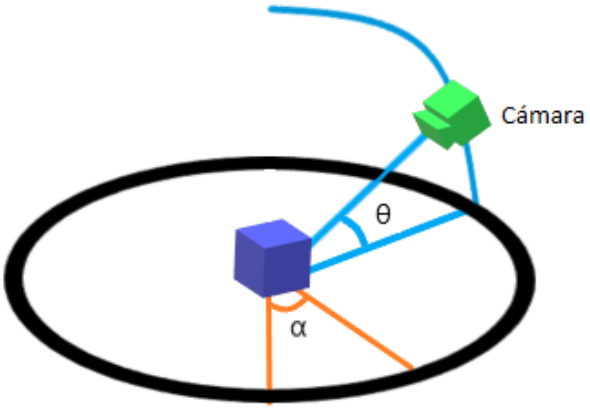


Figura 9. Secuencia en la que se mapea un objeto para obtener el su conjunto de vectores descriptivos. Fuente: Elaboración propia.

3.1.1 Estudio Fotográfico

Para la captura de las imágenes de los objetos, fue necesario construir un ambiente en el que el fondo y la iluminación estuvieran controlados con el propósito de generar las mejores condiciones fotográficas, ya que esto también facilitaría el desarrollo de las etapas siguientes.

Se diseñó un estudio fotográfico con dimensiones internas de 35 x 35 x 31 [cm], el cual se forró internamente con cartulina blanca mate para permitir que la luz se distribuyera uniformemente, como fuente principal de iluminación se colocaron dos tiras LED de color blanco en la parte superior, el resultado final se puede apreciar en la Figura 10.

Figura 10. Estudio fotográfico. Fuente: Elaboración propia.

La rotación del objeto se logró mediante la construcción de una base giratoria empleando el motor a pasos 28byj-48, como se muestra en la Figura 11, con el propósito de poder rotar el

objeto cierta cantidad de grados con mayor exactitud sin la necesidad de hacerlo manualmente y, además, facilitando y acelerando el proceso de captura de imágenes.

Figura 11. Base giratoria. Fuente: Elaboración propia.

La captura de imágenes se llevó a cabo mediante una cámara web de propósito general con las siguientes características:

- Modelo: Webcam HD FULL.
- Resolución: 1920 x 1080, 1280 x 720, 640 x 480, 320 x 240, entre otros.
- Frecuencia: Hasta 30 FPS.
- Interfaz: USB 2.0.

Se eligió esta cámara por su bajo costo, interfaz de conexión, resolución ajustable, facilidad de adquisición y soporte.

Para controlar el ángulo de captura de la cámara, se diseñó un arco de 90 grados, graduado cada 5 grados, una base para sostener dicho arco y una abrazadera. A la abrazadera, mediante piezas de LEGO® Technic™, se le adaptó una estructura con el fin de fijar la cámara, en la Figura 12 se muestra el soporte con la cámara.

Figura 12. Lado izquierdo, base y arco, lado derecho, abrazadera y acople para la cámara y en la parte inferior. Fuente: Elaboración propia.

3.1.2 Captura de Imágenes

Como primer paso, se decidió rotar a los objetos cada 10 grados y tomar fotografías con la cámara cada 10 grados a lo largo del arco, obteniendo un total de 360 fotografías por objeto, al ser 19 objetos el número total de fotografías sería de 6,840. Para esta tarea se desarrolló un programa basado en Python para la captura semiautomática de las imágenes.

Este programa se desarrolló e implemento en la tarjeta Raspberry Pi Zero W, ya que tiene todas las prestaciones que permiten el control del motor a pasos, a través de sus pines de propósito general, y la captura de las imágenes mediante la cámara web conectada por medio de un concentrador USB a la Raspberry.

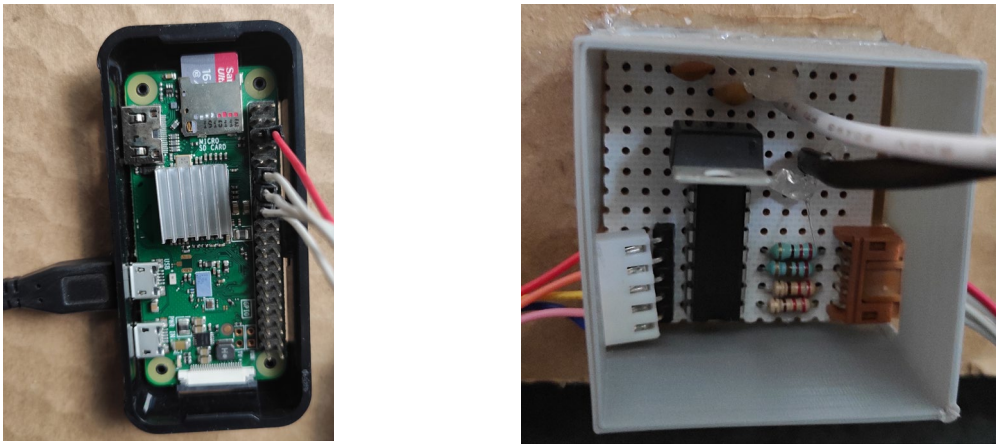


Figura 13. A la izquierda, Raspberry Pi Zero W, a la derecha, circuito de potencia para el motor a pasos, basado en el módulo ULN2003. Fuente: Elaboración propia.

El proceso de captura se lleva a cabo de manera semiautomática ya que es necesario ajustar la cámara a lo largo del arco y cambiar los objetos una vez que ha terminado su proceso de captura. La rotación de estos, la captura de la fotografía, el almacenamiento y el etiquetado se lleva a cabo de manera automática mediante la Raspberry.

La resolución elegida para las imágenes fue de 640 x 480 píxeles, no obstante, se pudo haber elegido la resolución más alta permitida por la cámara se optó por esta ya que resultaba ser suficiente para poder extraer el vector descriptivo de cada objeto.

3.2 BOF

Una vez obtenido el conjunto de imágenes de cada objeto, se prosiguió a la generación del conjunto de vectores descriptivos, a través de la BOF, de cada uno de ellos. Esta tarea se desarrolló e implementó mediante un programa en una computadora de propósito general, a través de Python, el cual se detalla a continuación:

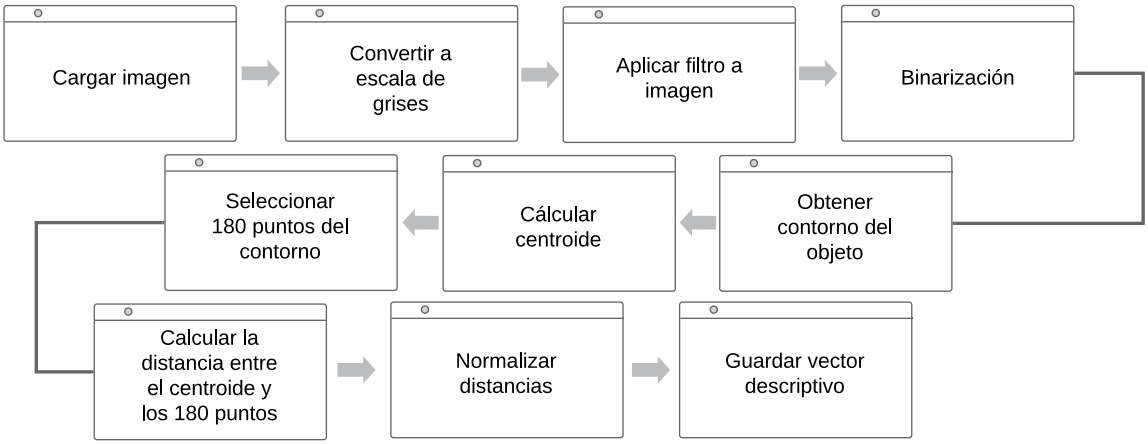


Figura 14. Algoritmo para extraer un vector descriptivo de una imagen mediante BOF. Fuente: Elaboración propia.

1. Se carga cada conjunto de imágenes de cada objeto.
2. Cada imagen de cada conjunto es convertida a escala de grises.
3. A todas las imágenes se les aplica un filtro Gaussiano para eliminar las altas frecuencias.
4. Cada imagen es binarizada asignando un '1' al fondo y un '0' a los píxeles que comprenden el objeto.
5. Se extraen las coordenadas que conforman el contorno del objeto de cada imagen binarizada analizando el cambio en el valor en el valor de los píxeles.
6. Se calcula el centroide del objeto encontrado en la imagen.
7. Se toman solo 180 puntos del contorno del objeto.
8. Se calculan las distancias que hay entre los 180 puntos del contorno y el centroide.
9. Se busca la mayor distancia dentro de las 180 y se normalizan.
10. Se guarda en un archivo todos los vectores obtenidos.

3.3 FPGA

El FPGA elegido para la implementación de todo el sistema fue la tarjeta de desarrollo Zybo Z7-20 de Digilent, construida en torno al SoC Zynq-7000 (XC7Z020-1CLG400C) de Xilinx®, la cual proporciona distintos recursos como memoria RAM, periféricos de entrada y salida de audio y vídeo, puertos USB y Ethernet y 6 puertos Pmod[5] disponibles.

[5] La interfaz Pmod es un estándar abierto definido por Digilent para periféricos usados en tarjetas de desarrollo de FPGAs o microcontroladores.

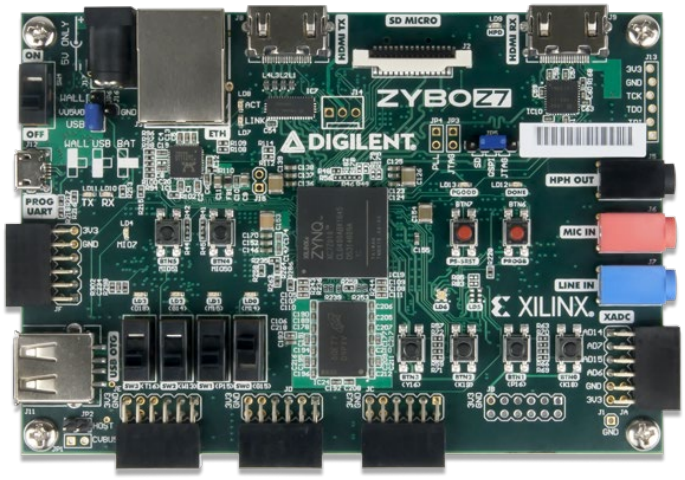


Figura 15. Tarjeta de desarrollo Zybo Z7-20. Fuente: zybo-z7-1.png (600×600) (digilentinc.com) (Último acceso: 04/04/21).

Se eligió esta tarjeta ya que, al ser diseñada con base en el SoC Zynq-7000 es posible utilizar las herramientas HLS de Vivado, además, cuenta con un procesador y un puerto e interfaz dedicados para la utilización de una cámara y salida de vídeo a través de HDMI, los cuales amplían las posibilidades de implementar la RNC, así mismo, posee un puerto USB donde se pueden conectar otros dispositivos que lo complementen y un puerto Ethernet que lo vuelve capaz de conectarse a Internet.

La tarjeta posee las siguientes especificaciones:

- **SoC ZYNQ 7000**
  - ARM Cortex-A9.
  - Protocolos de comunicación: SPI, UART, CAN, I2C.
  - Programable mediante JTAG, Quad-SPI flash y tarjeta microSD.
  - FPGA Artix-7.
- **Memoria**
  - 1 GB DDR3L con un ancho de bus de 32-bits.
  - Ranura para microSD.
- **Alimentación**
  - Alimentado desde USB o cualquier fuente externa de 5V.
- **USB y Ethernet**
  - Gigabit Ethernet PHY.
  - USB-JTAG.
  - USB-UART.
  - USB 2.0 OTG PHY.
- **Audio y Vídeo**
  - Conector para cámara Pcam con soporte MIPI CSI-2.
  - HDMI de entrada.

- HDMI de salida.
- Enchufe de 3.5 mm para entrada y salida de audio.

- **Conectores**
  - Puertos Pmod.

3.3.1 Sistema de Adquisición

Para la adquisición de vídeo a través de la tarjeta Zybo se decidió utilizar el módulo Pcam 5C de Digilent (Figura 16), el cual se encuentra diseñado con base en el sensor de imagen OV5640 de Omnivision. Este módulo es compatible con el puerto e interfaz de la tarjeta convirtiéndola en una opción viable y práctica para su implementación.

Figura 16. Módulo de cámara Pcam 5C de Digilent. Fuente: Pcam 5C: 5 MP Fixed-Focus Color Camera Module - Digilent (digilentinc.com) (Último acceso: 05/04/21).

El módulo posee distintas funciones internas de procesamiento que pueden ayudar a obtener una calidad de imagen superior, como balance automático de blanco, calibración automática de negro y controles para ajustar la saturación, matiz y nitidez.

Los datos son transferidos a través de la interfaz MIPI CSI-2, la cual provee suficiente ancho de banda (hasta 2000 Mbps[6]) para transmitir formatos de vídeo de 1080p (30 FPS[7]) y 720p (60 FPS) [32].

A partir del proyecto de código abierto de Digilent para la tarjeta Zybo y el módulo Pcam 5C [33] se comenzó el diseño de la adquisición de vídeo, el cual está constituido en el diagrama de bloques mostrado en la Figura 17.

MIPI D-PHY RX

Implementa la capa física de la interfaz MIPI D-PHY. Los datos son deserializados y las dos vías son integradas.

[6] Megabits por segundo.
[7] Frames Per Second o cuadros por segundo

MIPI CSI2 RX

Implementa el protocolo de la interfaz MIPI Camera Serial Interface 2 (CSI 2). Interpreta los datos como píxeles, líneas y fotogramas. Los datos de vídeo son formateados como RAW[8] RGB y empaquetados para su transmisión a través de la interfaz AXI-Stream, de esta manera se genera la entrada hacia el *pipeline* de procesamiento de vídeo.

AXI-S Bayer a RGB

Interpola los datos RAW RGB filtrados en datos estándar RGB de 32bits: 2 bits de padding y 10 bits para el canal rojo, verde y azul, respectivamente.

AXI-S Corrección Gama

Aplica corrección gama a la transmisión de vídeo mientras reduce la profundidad de los canales RGB de 10 bits a 8 bits por canal (24 bits: 8 bits rojo, 8 bits verde y 8 bits azul). Este bloque aplica uno de los cinco factores gama a la imagen de entrada: $1, \frac{1}{1.2}, \frac{1}{1.5}, \frac{1}{1.8}$ y $\frac{1}{2.2}$.

AXI Vídeo DMA

Implementa tres *buffers* de fotogramas en la memoria DDR para permitir el acceso desde dentro del procesador, si es necesario. El bloque es conectado al subsistema de memoria Zynq a través de dos puertos de memoria de alto rendimiento. Con el almacenamiento de fotogramas se consigue duplicar la frecuencia de fotogramas de entrada de 30Hz a 60 Hz de salida.

AXI-S a Salida de Vídeo

Convierte la transmisión de vídeo AXI-Stream a una interfaz de vídeo de salida.

RGB a DVI

Codifica los datos de RGB a DVI y los envía a través del conector de salida de HDMI.

[8] Es un formato de archivo digital de imágenes que contiene la totalidad de los datos de la imagen tal y como ha sido capturada por el sensor digital de la cámara.

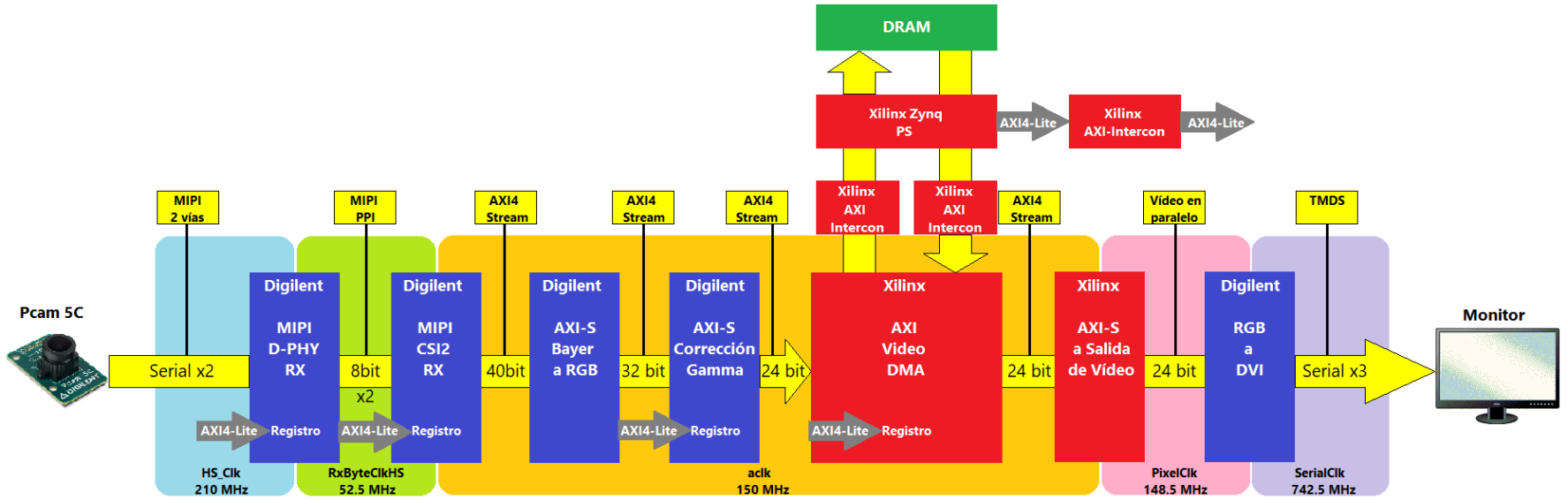


Figura 17. Diagrama de bloque de la captura y despliegue de vídeo a través del módulo Pcam 5C. Fuente: *Zybo Z7-20 Pcam 5C Demo with YCbCr422/RGB565 - FPGA - Digilent Forum (digilentinc.com)* (Último acceso: 05/04/21).

4 Implementación y Resultados

4.1 Sistema de Adquisición

El sistema de adquisición de vídeo se basó en el proyecto mencionado en el apartado 3.3.1, el cual captura vídeo en distintas resoluciones y fotogramas por segundo, además, permite realizar un preprocesamiento a la imagen ya sea configurando los registros internos del sensor o las opciones que el proyecto posee como:

- Resolución.
- Formato de imagen.
- Corrección gama.
- Balance automático de blancos.

Para la captura de vídeo se probó con distintas configuraciones y estas fueron las que incrementaban la calidad de la imagen:

- Resolución: 1280 x 720p a 60 FPS.
- Formato: RGB.
- Corrección gama: $\frac{1}{1.2}$.
- Registro 0x3821: Efecto espejo activado.
- Registro 0x5000: Interpolación de color, cancelación de píxel blanco y negro habilitados.
- Registro 0x5001: Balance automático de blanco, matriz de color y promedio UV habilitados.

El proyecto despliega el vídeo a través del puerto HDMI TX de la tarjeta para su observación, sin embargo, para poder extraer el conjunto de vectores descriptivos a través de la BOF es necesario realizar otra serie de pasos como el mencionado en la sección 3.2. Mediante Vivado® HLS se diseñó el módulo “gray_blur_bw”, el cual se encarga de realizar las siguientes acciones:

- Convertir la imagen de color (RGB) a escala de grises (8 bits).
- Aplicar filtro Gaussiano con un kernel de tamaño 5 x 5.
- Binarizar la imagen con un umbral de 115.

Una vez binarizada la imagen, es posible comenzar el procedimiento para extraer el conjunto de vectores a través de la BOF. En la Figura 18 se muestran los objetos desplegados a través de la cámara y el FPGA en un monitor, del lado izquierdo se aprecia la imagen antes del módulo “gray_blur_bw”, y a la derecha el resultado después del módulo.

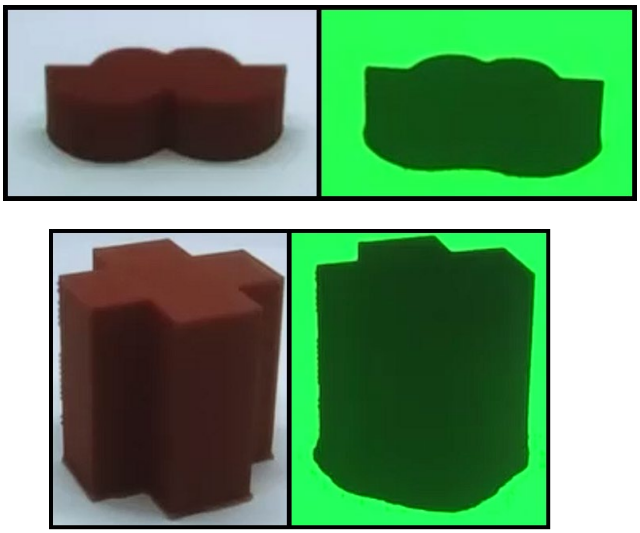


Figura 18. Visualización de los objetos mediante el sistema de adquisición por la cámara en el FPGA, desplegada en un monitor. Fuente: Elaboración propia.

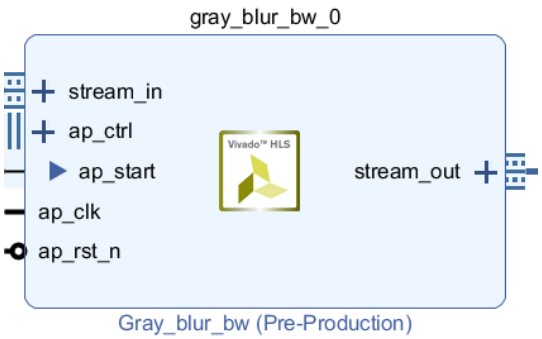


Figura 19. Modulo gray_blur_bw para la conversión de vídeo RGB a binario. Fuente: Elaboración propia en Vivado®.

4.2 RNC

El desarrollo de la RNC para la clasificación de objetos basado en un conjunto de vectores descriptivos se llevó a cabo tomando como punto de partida la red llamada LeNet-5, desarrollada por Yann LeCun et al. [34], se eligió esta red por su tamaño e implementaciones que se han realizado en otros FPGA [10], [13]. En la Figura 20 se muestra la arquitectura de esta red, la cual está compuesta por dos capas convolucionales y tres capas totalmente conectadas, fue diseñada con el propósito de reconocer dígitos escritos a mano e impresos.

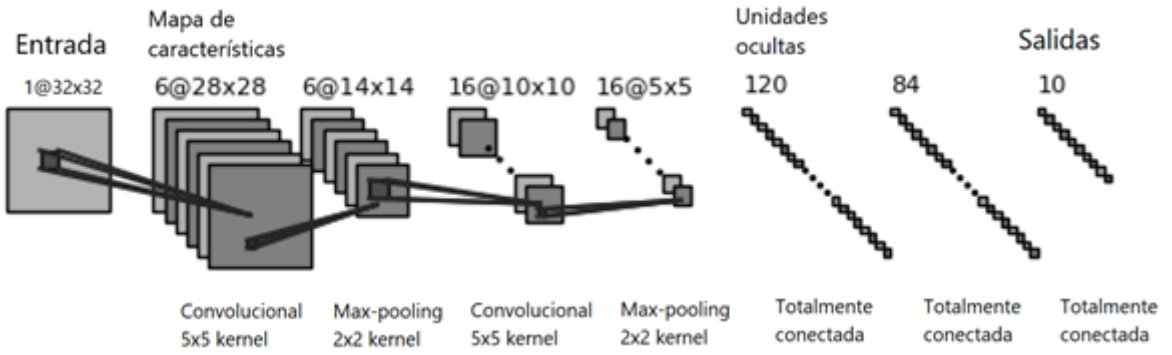


Figura 20. Arquitectura de la RNC LeNet-5. Fuente: D. Rongshi and T. Yongming, "Accelerator Implementation of Lenet-5 Convolution Neural Network Based on FPGA with HLS," 2019 3rd Int. Conf. Circuits, Syst. Simulation, ICCSS 2019, pp. 64–67, 2019, doi: 10.1109/CIRSYSSIM.2019.8935599.

Como se mencionó en el tema 3.1.2, por cada objeto se obtienen 360 imágenes, sin embargo, se determinó que las imágenes generadas cuando el ángulo θ de la cámara es igual a 0°, la perspectiva que se obtiene entre distintos objetos es muy similar, por lo tanto, únicamente se tomaron en cuenta las imágenes tomadas a partir de los 10° en adelante, obteniendo un total de 324 imágenes por objeto y 6,156 imágenes en total.

Como se puede observar en la Figura 20, los kernels de las capas convolucionales y de pooling son de tamaño cuadrado, por lo tanto, no es posible aplicarlos de esta manera al vector descriptivo y es necesario cambiar su tamaño como se muestra en la Tabla 2.

Capa	**Conv1**	**Max-pooling1**	**Conv2**	**Max-pooling2**
Tamaño original	5x5	2x2	5x5	2x2
Tamaño adaptado	1x5	1x2	1x5	1x2

Tabla 2. Tamaños adaptados de los kernels de las capas convolucionales y de pooling.

Además, todo el conjunto de vectores descriptivos se dividió de la siguiente manera para las etapas de entrenamiento, validación y prueba:

Vectores descriptivos: 6,156

Conjunto de entrenamiento	**Conjunto de validación**	**Conjunto de prueba**
60%	20%	20%
3,694	1,231	1,231

Tabla 3. División del conjunto de vectores descriptivos para el entrenamiento.

Para implementar y entrenar la red LeNet-5 se utilizó la API[9] Keras en Python, en la Gráfica 1 se muestran los resultados obtenidos durante el entrenamiento para las métricas de exactitud y pérdida.

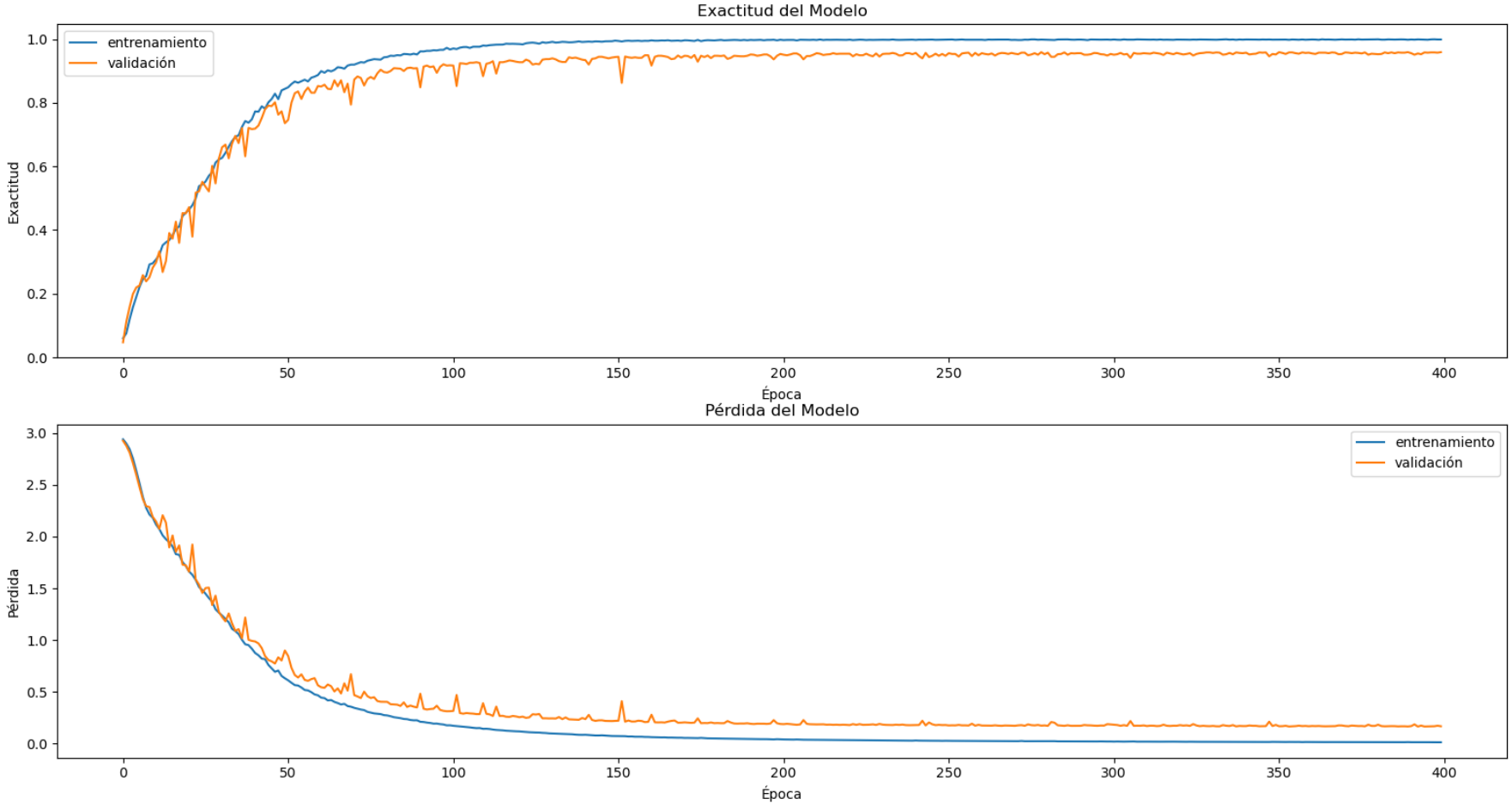
La exactitud se refiere a que tan cercana se encuentra la clase predicha a la clase verdadera y el valor de pérdida nos indica que tan mal son las predicciones de la red.

Aunque los valores de exactitud del entrenamiento, validación y prueba son altos, la diferencia entre los valores de pérdida del entrenamiento y la validación es muy grande, indicando que hay una alta varianza y un sobreajuste (overfitting, por su nombre en inglés) sobre el conjunto de entrenamiento. Por lo tanto, fue necesario aplicar ciertas técnicas de regularización para obtener un mejor desempeño de la red, las cuales se enlistan a continuación:

1. Utilización de la función de activación ReLU en lugar de la tangente hiperbólica.
2. Aplicación del optimizador Adam en lugar de descenso de gradiente estocástico (SGD, por sus siglas en inglés).
3. Creación de mini lotes (mini-batches, por su nombre en inglés).
4. Implementación de la técnica abandonar (dropout, por su nombre en inglés) en la primera capa totalmente conectada.

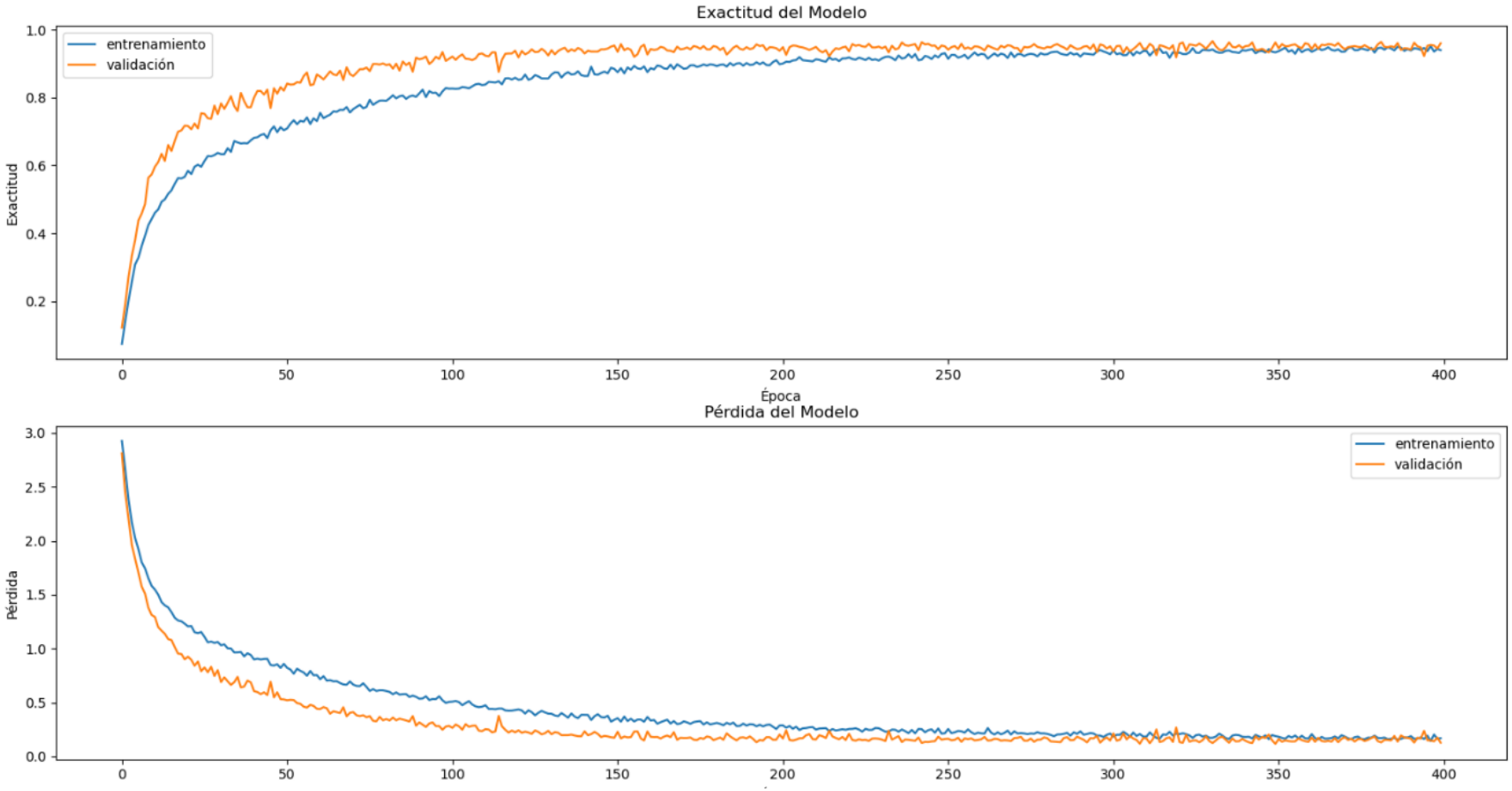
Como se puede observar en la Gráfica 2, se logró disminuir la diferencia entre los valores de pérdida del entrenamiento y la validación, no obstante, el valor de pérdida de la validación resulta menor en comparación con el de entrenamiento, por lo tanto, existe un infrajuste (underfitting, por su nombre en inglés) y un alto sesgo. Para tratar de disminuir el sesgo se implementaron 2 nuevas capas convolucionales de 12 y 25 filtros con kernels de tamaño de 1x5 para cada una, además, se disminuyó la cantidad de filtros de las primeras dos capas convolucionales a 5 y 8, respectivamente, y a solo 90 neuronas para la primera capa totalmente conectada.

[9] Del inglés, Application Programming Interface, es un conjunto de subrutinas, funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como capa de abstracción.



	Pérdida	Exactitud
Entrenamiento	0.0111	0.9996
Validación	0.1657	0.9594
Prueba	0.1361	0.9594

Gráfica 1. Resultados de entrenamiento de la red LeNet-5 con el conjunto de vectores descriptivos a 400 épocas. Fuente: Elaboración propia.



	Pérdida	Exactitud
Entrenamiento	0.1713	0.9409
Validación	0.1277	0.9602
Prueba	0.1407	0.9521

Gráfica 2. Resultados tras la aplicación de técnicas de regularización. Fuente: Elaboración propia.

En la Gráfica 3 se muestran los resultados de la RNC después de agregar las dos capas convolucionales y modificar la cantidad de filtros y neuronas en algunas de ellas. La diferencia entre los valores de pérdida de entrenamiento y validación ahora es menor en comparación con los resultados anteriores, no obstante, el valor de pérdida de entrenamiento es menor al de validación, lo cual se puede interpretar como un sesgo y una varianza bajos.

Además de los valores de pérdida y exactitud, existen otras métricas que nos permiten determinar si el desempeño de la red es el adecuado, las cuales son:

- Precisión.
- Exhaustividad.

Para calcular estas métricas es necesario ordenar los resultados de clasificación en las siguientes categorías:

1. Verdadero Positivo (VP)
    - El valor real es positivo y la red clasificó al objeto como tal.
2. Verdadero Negativo (VN)
    - EL valor real es negativo y la red clasificó al objeto como tal.
3. Falso Positivo (FP)
    - El valor real es negativo y la red clasificó al objeto como verdadero.
4. Falso Negativo (FN)
    - El valor real es positivo y la red clasificó al objeto como falso.

Precisión

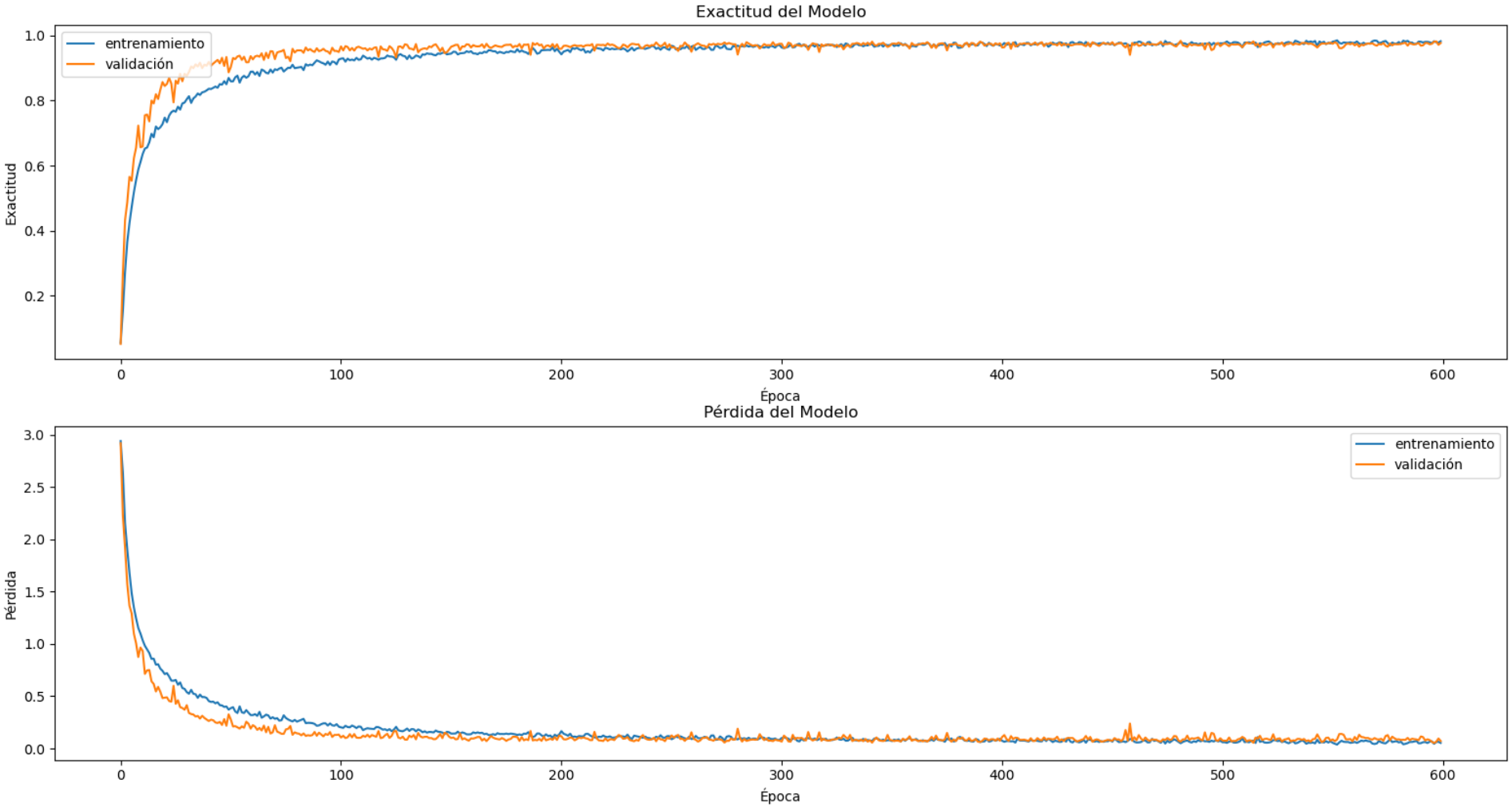
La precisión se refiere a la dispersión de un conjunto de valores obtenidos a partir de mediciones repetidas de una magnitud, en términos más prácticos, el porcentaje de verdaderos positivos detectados de todo el conjunto.

$$precisión = \frac{VP}{VP + FP} \tag{8}$$

Exhaustividad

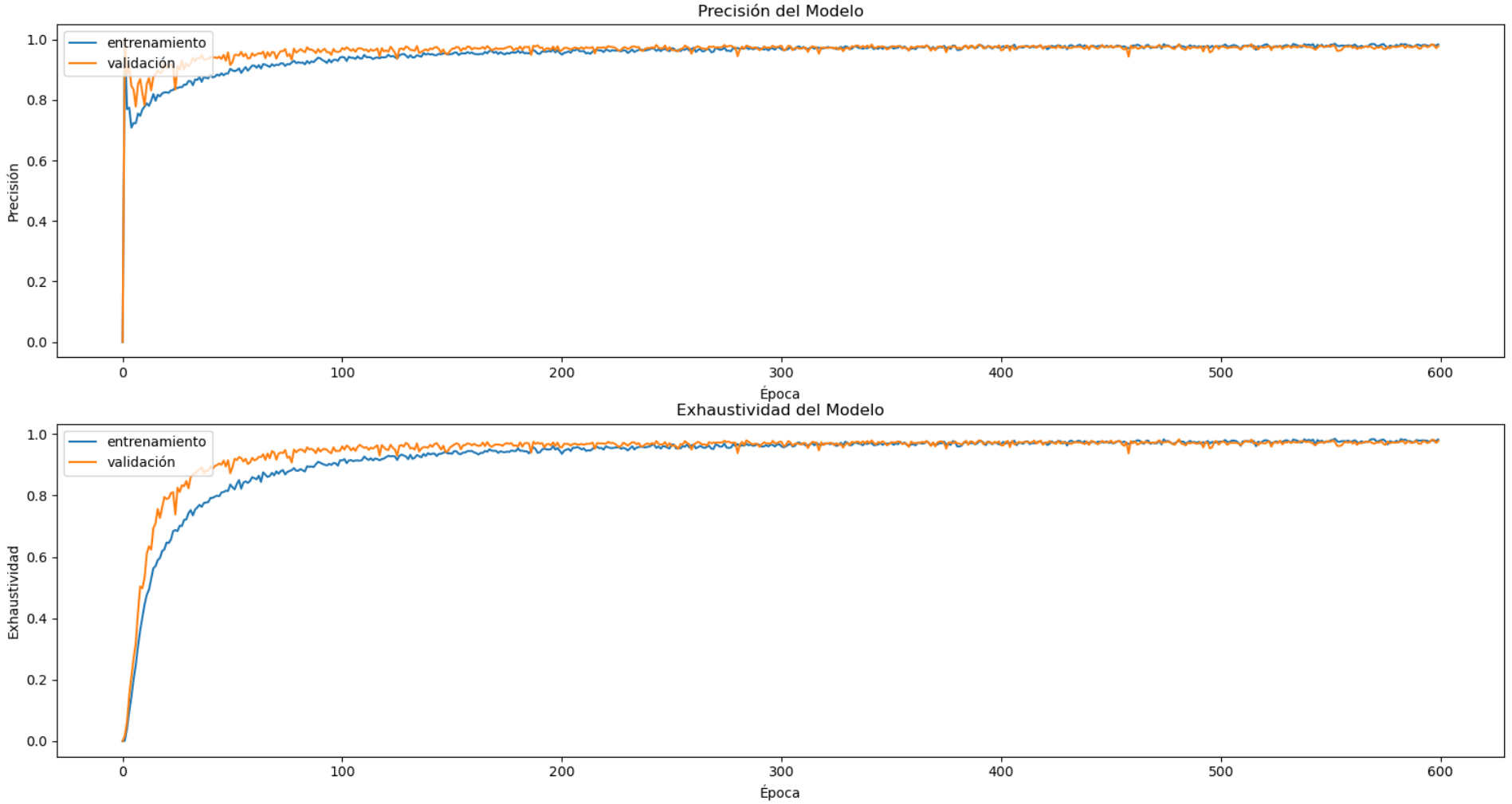
El valor de exhaustividad indica la capacidad que tiene la red para discriminar los casos positivos de los negativos, es decir, la proporción de casos positivos que fueron correctamente identificados de todos los casos positivos que podían predecirse.

$$exhaustividad = \frac{VP}{VP + FN} \tag{9}$$



	Pérdida	Exactitud
Entrenamiento	0.0312	0.9897
Validación	0.0545	0.9805
Prueba	0.0845	0.9781

Gráfica 3. Resultados finales del desempeño de la RNC diseñada para la clasificación de objetos a través un conjunto de vectores descriptivos. Fuente: Elaboración propia.



Clase	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Precisión	0.9935	0.9705	0.9939	0.9818	0.9786	0.9969	0.9729	0.9068	1.0	0.9667	0.9938	1.0	0.9878	0.9937	0.967	1.0	0.991	0.9818	0.9906
Exhaustividad	0.9444	0.9136	1.0	0.997	0.9876	0.9876	0.997	0.991	1.0	0.9846	0.997	0.961	0.997	0.9691	0.9938	0.9938	0.9784	1.0	0.9753

Gráfica 4. Precisión y exhaustividad del modelo por clase. Fuente: Elaboración propia.

En conjunto con estas métricas, se puede construir la matriz de confusión del modelo para observar de manera más clara su desempeño, como se muestra e la Figura 21. En la diagonal principal se ubican los VP de cada clase, es decir, la cantidad de veces que la red clasificó al conjunto de vectores en la clase correcta. Fuera de la diagonal principal se encuentran los FP, en otras palabras, el número de veces que la red clasificó al conjunto de vectores perteneciente a la clase que se está analizando cuando no lo era, y FN, las veces que la red no clasificó al conjunto de vectores dentro de la clase a la que se está analizando cuando sí formaba parte.

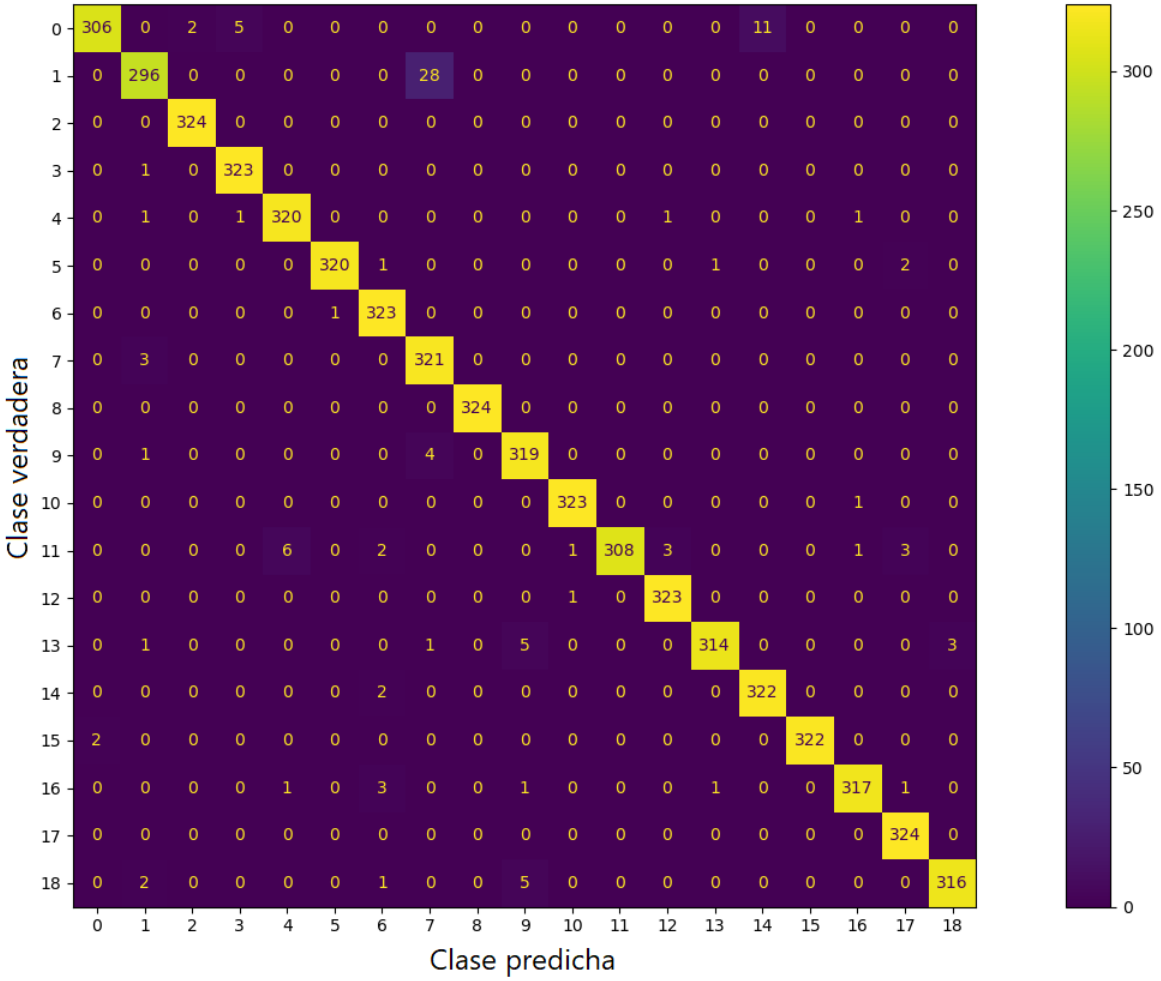


Figura 21. Matriz de confusión del modelo. Fuente: Elaboración propia.

En la Tabla 4 se muestra el tiempo requerido por la red para realizar la clasificación de un solo conjunto de vectores utilizando el siguiente hardware:

- Procesador: AMD Ryzen™ 5 3400G con Gráficos Radeon™ RX Vega 11.
- Memoria RAM: 2 x 8GB a 3000 MHz en Dual Channel.
- IDE: Visual Studio Code con Python 3.8.5 de 64 bits.

	Python
Latencia	37.97 [ms]

Tabla 4. Latencia de clasificación de la RNC implementada en Python.

En la Figura 22 se muestra la arquitectura final de la RNC, la cual está compuesta por 4 capas convolucionales, con su etapa de pooling cada una de ellas, y 3 capas totalmente conectadas, obteniendo un total de 7 capas con 27,354 parámetros entrenables, ver Tabla 5.

Capa	**Dimensión de salida**	**# Parámetros**
Conv1	(176, 5)	30
Max_pooling1	(88, 5)	0
Conv2	(84, 8)	208
Max_pooling2	(42, 8)	0
Conv3	(38, 12)	492
Max_pooling3	(19, 12)	0
Conv4	(15, 25)	1525
Max_pooling4	(7, 25)	0
Dense1	(1, 90)	15,840
Dense2	(1, 84)	7,644
Dense2	(1, 19)	1,615

Tabla 5. Dimensiones de salida y parámetros entrenables de capa.

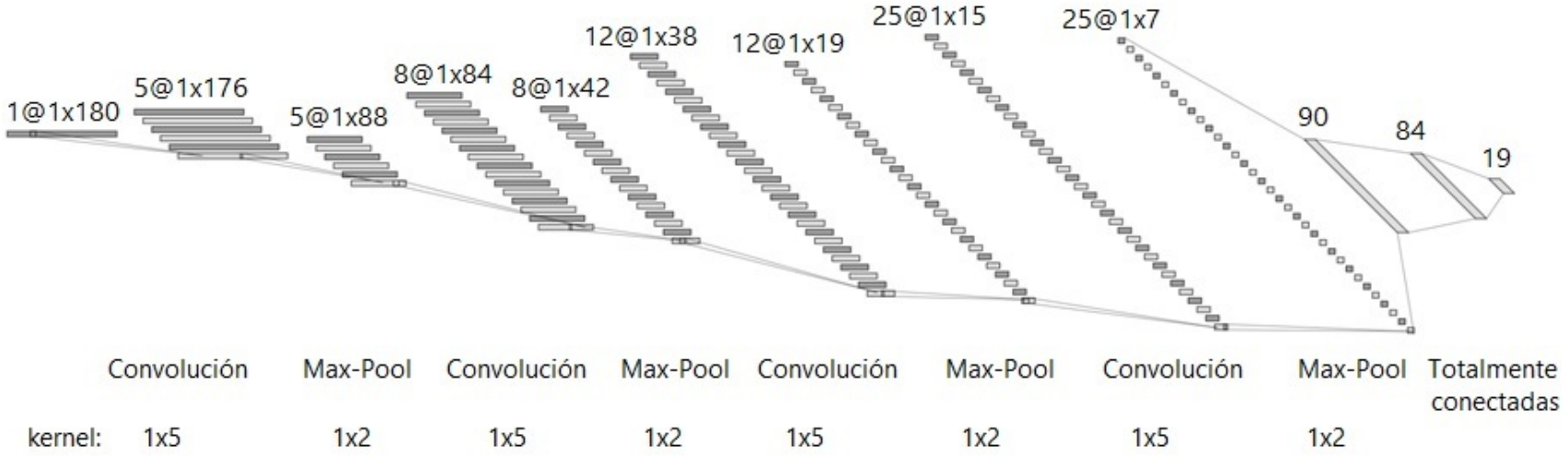


Figura 22. Arquitectura final de la RNC. Fuente: Elaboración propia en NN SVG (alexlenail.me) (Último acceso: 13/08/21).

4.3 FPGA

Una de las principales ventajas de implementar y ejecutar una RNC en un FPGA es la cantidad de operaciones simultáneas que se pueden llevar a cabo en un solo ciclo de reloj, sin embargo, es necesario tomar en cuenta la cantidad de recursos que se tienen disponibles con el objetivo de poder diseñar una red que sea eficiente no solo en su funcionamiento sino también en la utilización de estos.

4.3.1 RNC

El primer paso para implementar la RNC en el FPGA fue elegir la representación que se utilizaría para los datos, para lo cual se decidió emplear el punto fijo ya que es una representación que permite:

- Reducir la utilización de recursos como los procesadores digitales de señal (DSP, por sus siglas en inglés), tablas de búsqueda (LUT, por sus siglas en inglés), bancos de memoria, etc.
- Disminuir el consumo energético.
- Mejorar la latencia.
- Utilizar los recursos adicionales disponibles para agregar otras características [35].

La representación en punto fijo, a diferencia del punto flotante, mantiene una cantidad fija de dígitos antes y después del punto decimal, no obstante, el utilizar esta representación conlleva un costo en la precisión, ya que una vez que se realiza una operación el resultado debe ser redondeado o truncado al valor más cercano que se puede representar, este redondeo o truncamiento produce un error de cuantificación [36].

Fue necesario probar distintos formatos de punto fijo para establecer los que mejor se ajusten a las necesidades de la implementación y eficiencia en el FPGA, para ello se diseñó la etapa de propagación hacia adelante desde cero en Python con el fin de poder evaluar y manipular cada paso. También se utilizó la biblioteca “fxpmath” la cual permite representar números en punto fijo, entre otras funciones [37].

Para la representación de los vectores descriptivos y pesos de las capas de la red neuronal se utilizaron 9 bits:

- 1 bit para el signo.
- 1 bit para la parte entera.
- 7 bits para la parte decimal.

Se empleó este formato ya que, con una menor cantidad de bits para la parte decimal la exactitud de clasificación de la red disminuye en gran medida, ver Tabla 6, en caso contrario, aunque aumenta la exactitud, los recursos utilizados en el FPGA también aumentarían.

Exactitud RNC con punto flotante	Exactitud RNC con 7 bits	Exactitud RNC con 9 bits	Exactitud con 11 bits
99.0%	92.0%	96.0%	98.0%

Tabla 6. Exactitud de clasificación de 100 muestras con distintos formatos para los vectores descriptivos y pesos.

En la representación de las operaciones se probaron distintos formatos con los cuales se pudiera obtener una exactitud aceptable sin comprometer los recursos del FPGA, para las capas convolucionales y totalmente conectadas (a excepción de la etapa de clasificación de la última capa) se emplearon 16 bits con los siguientes formatos:

- Capas convolucionales
  - 1 bit para el signo.
  - 3 bits para la parte entera.
  - 12 bits para la parte decimal.
- Capas totalmente conectadas
  - 1 bit para el signo.
  - 7 bits para la parte entera.
  - 8 bits para la parte decimal.

Exactitud RNC con punto flotante	Exactitud RNC con 9 bits en ambas capas	Exactitud RNC con 16 bits en ambas capas
99.0%	40.0%	96.0%

Tabla 7. Exactitud de clasificación de 100 muestras con distintos formatos en las capas de la RNC.

En la capa de clasificación y debido a la naturaleza de la función de activación Softmax, fue necesario utilizar un formato mayor ya que el resultado de las operaciones realizadas producía valores muy grandes o pequeños. El formato utilizado fue de 32 bits:

- 24 bits para la parte entera.
- 8 bits para la parte decimal.

Exactitud RNC con punto flotante	Exactitud RNC con 32 bits en la última etapa
99.0%	94.0%

Tabla 8. Exactitud de clasificación de 100 muestras con un formato de 32 bits en la capa de clasificación.

Aunque el porcentaje de exactitud disminuyó al 94%, aún es un valor válido para nuestros propósitos.

Una vez establecidos los formatos de punto fijo para cada capa de la RNC, se comenzó con el diseño e implementación en el FPGA.

4.3.2 Paquetería de Punto Fijo en VHDL

De manera análoga a la biblioteca de punto fijo en Python, fue necesario emplear una paquetería que nos permitiera declarar y realizar operaciones con representación en punto fijo en VHDL, para tales propósitos existe la paquetería llamada "FIXED_PKG.VHDL", con esta paquetería se agregaron dos nuevos tipos de variables:

- ufixed: punto fijo sin signo.
- sfixed: punto fijo con signo.

Estos tipos de variables muestran la posición del punto decimal mediante la utilización del signo negativo cuando son declaradas, se asume que el punto decimal se encuentra entre el 0 y -1. A continuación, se muestran ejemplos para declarar ambos tipos de variables:

Tipo de dato sin signo con 3 bits para la parte entera y 3 bits para la parte decimal.

signal a : ufixed (2 downto -3);

Tipo de dato con signo con 1 bit para el signo, dos bits para la parte entera y 3 bits para la parte decimal.

signal b : sfixed (2 downto -3);

Al realizar operaciones entre datos de punto fijo existen dos escenarios para almacenar el resultado:

1. Declarar otro dato en punto fijo con las dimensiones correctas utilizando las reglas de dimensiones.

Reglas para las dimensiones de los resultados

Operación	Dimensión
A + B	(Max(A'left, B'left) + 1 downto Min(A'right, B'right))
A – B	(Max(A'left, B'left) + 1 downto Min(A'right, B'right))
*A * B*	(A'left + B'left + 1 downto A'right + B'right)

Tabla 9. Reglas para las operaciones básicas.

2. Utilizar la función *resize* donde se aplican las reglas de redondeo y desbordamiento según se especifique.

*a <= resize(a * b, x'high, x'low)*

- Estilos de redondeo: *fixed_round* (por defecto) y *fixed_truncate.*
- Estilos de desbordamiento: *fixed_saturate* (por defecto) y *fixed_wrap.*

Para mantener el mismo formato de punto fijo tras cada operación realizada, se empleó la función *resize* en cada módulo con distintos estilos de redondeo y desbordamiento.

Otra función muy útil es la conversión a datos de punto fijo, la cual es posible mediante las siguientes funciones:

a <= to_sfixed(-3.125, 7, -6)
b <= to_ufixed(a, b)

Cabe mencionar que la paquetería no permite mezclar variables signadas con variables sin signo al realizar operaciones, es decir, no podemos sumar, multiplicar, etc. variables *ufixed* con variables *fixed*. Se debe elegir un solo formato para realizar las operaciones y una vez llevadas a cabo convertir el resultado al tipo de variable requerido.

4.3.3 Memoria del Conjunto de Vectores Descriptivos

Los valores del vector descriptivo del objeto a clasificar se almacenan en un arreglo con el formato descrito en el tema 4.3.1 y se mandan a través de cada capa de la red hasta completar la propagación hacia adelante. En un principio, se diseñó la primera capa convolucional de tal manera que realizara la convolución de todos los filtros en un solo ciclo de reloj, es decir, si en la primera capa convolucional las dimensiones de los kernels son de 1x5 y en total se tienen 5 filtros, se realizarían 25 multiplicaciones, 5 por cada filtro, y 20 sumas, 4 por cada filtro, en total. Sin embargo, siguiendo este diseño en las siguientes capas convolucionales, la cantidad de recursos empleados, principalmente las LUT, resultaban ser insuficientes cuando se llegaba a la cuarta capa, por lo tanto, fue necesario cambiar la manera en que se realizarían las convoluciones.

Como la primera capa convolucional está constituida por 5 filtros, se decidió realizar la convolución de un filtro completo por ciclo de reloj, necesitando solo 5 ciclos para aplicar todos los filtros; esto resulta también en una menor utilización de los recursos, principalmente LUT, ya que por ciclo de reloj solo se realizarían 5 multiplicaciones y 4 sumas.

Al aplicar la convolución de un filtro por ciclo de reloj en las demás capas convolucionales y comprobar que los recursos empleados por éstas no era excesivamente alto, se determinó que esta estrategia era la adecuada para el diseño de las capas convolucionales, definiendo también la cantidad de datos que se transferirían de módulo a módulo iniciando desde la memoria BOF.

Como se mencionó anteriormente, al ser todos los kernels de tamaño 1x5 de los filtros de la primera capa convolucional, la memoria enviará 5 valores (9 bits por cada valor) del vector descriptivo, estos valores se unirán en un paquete de datos de 45 bits y se enviará para comenzar la propagación hacia adelante. Mientras la primera capa convolucional mantenga en “1” la señal de *start_in*, el módulo de memoria realizará el mismo proceso hasta haber enviado todos los valores del vector descriptivo. En la Tabla 10, se muestran los recursos utilizados por este módulo.

Nombre	Slice LUTs	Slice Registers	Multiplexores F7	Multiplexores F8	Slice
BOF	177	16	42	12	64

Tabla 10. Recursos utilizados por el módulo de memoria del conjunto de vectores descriptivos.

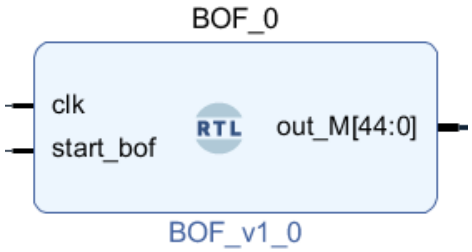


Figura 23. Módulo de memoria del conjunto de vectores descriptivos. Fuente: Elaboración propia en Vivado®.

En la Tabla 11 se muestra el tiempo que tarda el módulo en entregar un solo valor a la salida, así como también el tiempo total que tarda en enviar todos los datos. Es importante mencionar que la frecuencia del reloj a la que se trabajó fue de 100 MHz, esta frecuencia puede ser cambiada de acuerdo con los requerimientos necesarios, por lo tanto, también se muestran los ciclos de reloj empleados para los casos antes mencionados.

BOF	Ciclos de reloj	Tiempo
Latencia de salida	6	60 [ns]
Latencia total	1044	10.44

Tabla 11. Latencias del módulo de memoria.

4.3.4 Capas Convolucionales

Cada capa convolucional se compone de 4 etapas:

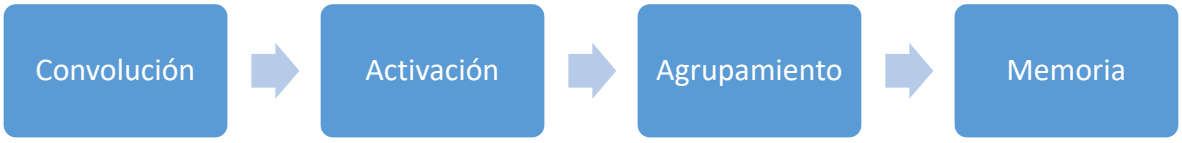


Figura 24. Etapas de una capa convolucional. Fuente: Elaboración propia.

En la primera etapa se realiza la convolución con los datos entrantes, en la segunda etapa se aplica la función de activación elegida, en la tercera se agrupan los valores más representativos de acuerdo con la estrategia seleccionada y en la cuarta se almacenan los valores resultantes.

Para cada una de las capas convolucionales se aplicaron los siguientes parámetros:

Kernel	**Stride**	**Padding**	**Activación**
1x5	1	-	ReLU

Tabla 12. Parámetros para la etapa de convolución y activación.

Agrupamiento	**Kernel**	**Stride**	**Padding**
Max-pooling	1x2	2	-

Tabla 13. Parámetros para la etapa de agrupamiento.

Primera Capa Convolucional

La primera capa convolucional está compuesta por 5 filtros con kernels de dimensiones 1x5 (25 pesos en total), por lo tanto, al recibir los 5 valores provenientes de la memoria estos son multiplicados por los pesos del kernel correspondiente, comenzando por el primer filtro, y los resultados son sumados junto con el sesgo perteneciente al mismo. Cuando los 5 filtros son aplicados a los 5 valores entrantes, el módulo pone en "1" la señal de *start_bof*, para ordenarle a la memoria que envíe los siguientes 5 valores, y la señal *start_out,* para indicarle al módulo de agrupación que puede leer los 5 valores resultantes. Al enviar los últimos valores, el módulo coloca en "1" la señal *last_out* y reinicia sus señales internas y externas.

Cabe mencionar que, debido a la cantidad de filtros que se tienen en la capa convolucional, la cantidad de datos de salida será igual a esa misma cantidad, es decir, si la primera capa convolucional tiene 5 filtros, a la salida se tendrán 5 datos de 16 bits cada uno, conformando un paquete de 80 bits en total.

Nombre	**Slice LUTs**	**Slice Registers**	**Slice**
Conv_1	706	174	250

Tabla 14. Recursos utilizados por la primera convolución.

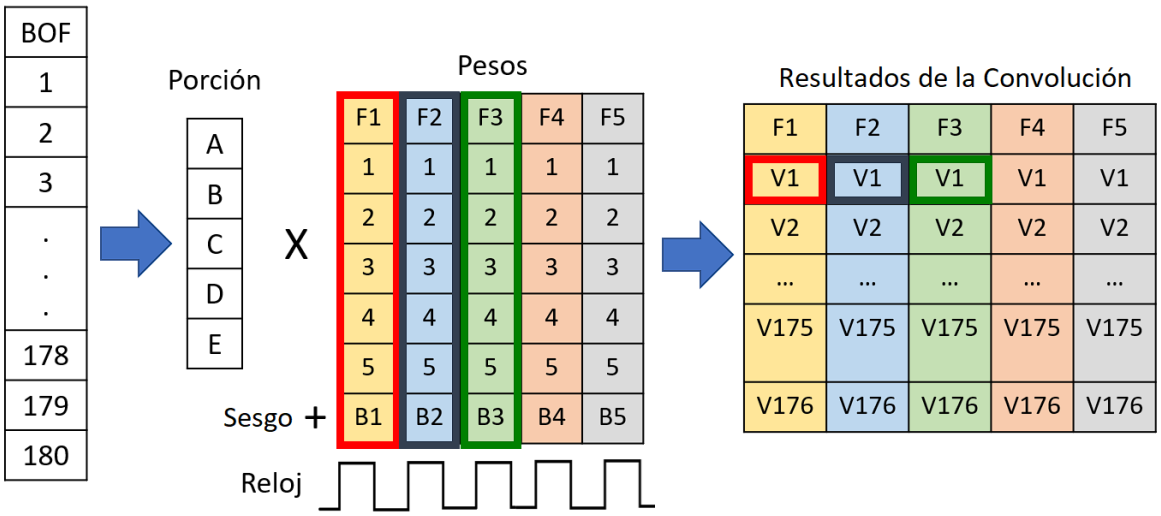


Figura 25. Diagrama del funcionamiento de la primera convolución. Fuente: Elaboración propia.

En la Figura 25 se puede observar el funcionamiento de la primera convolución, cada valor resultante es la suma de productos de los 5 valores de entrada por los pesos de cada filtro más el sesgo correspondiente.

Dentro de cada módulo de convolución se encuentra la etapa de activación, la cual se conforma por la función ReLU, esta función prácticamente convierte en cero a todos los valores que son negativos mientras que los positivos permanecen igual.

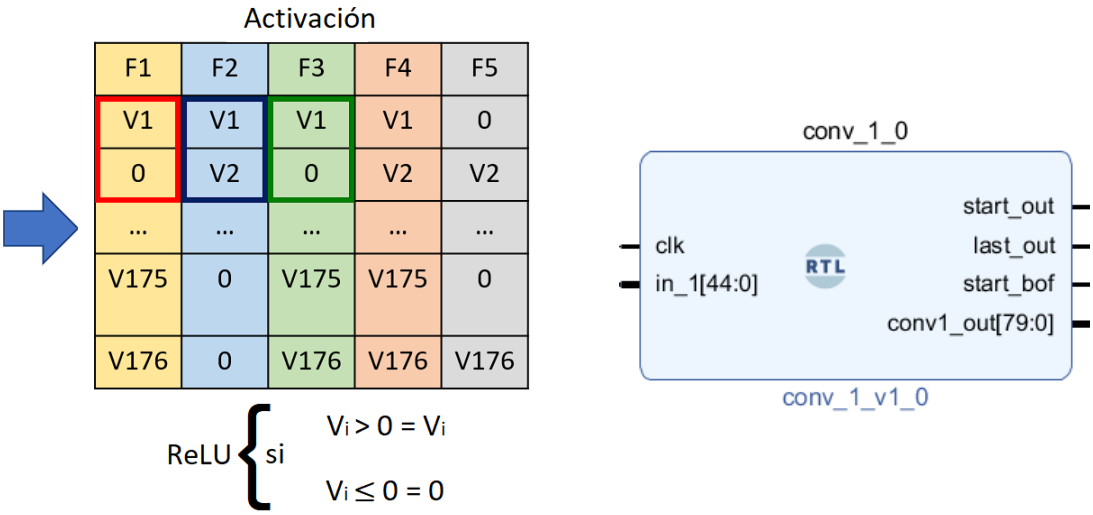


Figura 26. A la izquierda, diagrama de funcionamiento y a la derecha, módulo de la primera convolución. Fuente: A la izquierda, elaboración propia y a la derecha, elaboración propia en Vivado®.

Conv_1	Ciclos de reloj	Tiempo
Latencia de salida	6	60 [ns]
Latencia total	1056	10.56 [µs]

Tabla 15. Latencias de la primera convolución.

La siguiente etapa corresponde al agrupamiento, en específico al max-pooling o agrupamiento máximo, el cual consiste en tomar el valor máximo dentro de un conjunto de valores. En esta etapa, el kernel es de tamaño 1x2 y se tiene un stride de 2, la dimensión total de datos de salida es igual a la mitad de la dimensión de los datos de salida de la etapa de convolución.

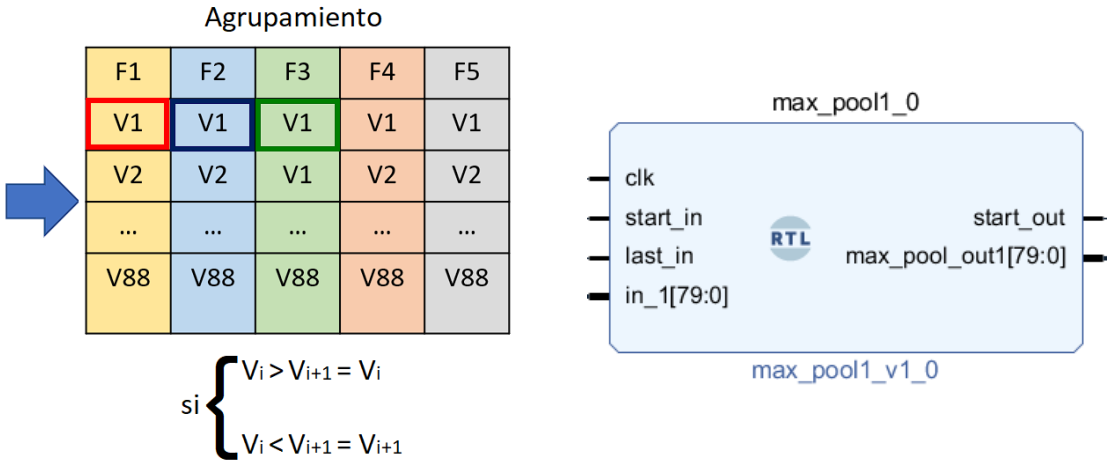


Figura 27. A la izquierda, diagrama de funcionamiento de la primera agrupación y a la derecha, el primer módulo.
Fuente: A la izquierda, elaboración propia y a la derecha, elaboración propia en Vivado®.

Nombre	Slice LUTs	Slice Registers	Slice
Max_pool1	83	162	44

Tabla 16. Recursos utilizados por el primer módulo de agrupamiento.

Max_pool1	Ciclos de reloj	Tiempo
Latencia de salida	12	120 [ns]
Latencia total	1051	10.51 [µs]

Tabla 17. Latencias del primer módulo de máximo agrupamiento.

Por último, los valores resultantes de la etapa de agrupamiento son almacenados en una memoria intermedia entre capas con el propósito de no perder los valores de salida de la etapa anterior. Son necesarias estas memorias intermedias ya que, conforme se avanza entre capas, la cantidad de operaciones va aumentando y, por lo tanto, también la latencia.

Nombre	Slice LUTs	Slice Registers	Slice	BRAM
Mem_1	33	33	12	1.5

Tabla 18. Recursos utilizados por la primera memoria intermedia.

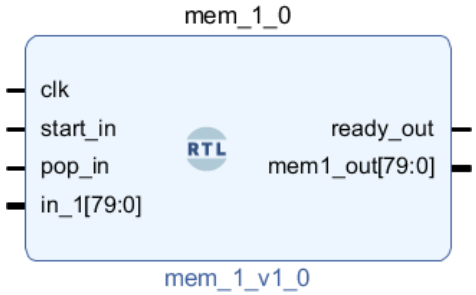


Figura 28. Módulo de la primera memoria intermedia. Fuente: Elaboración propia en Vivado®.

Mem_1	Ciclos de reloj	Tiempo
Latencia de salida	9	90 [ns]
Latencia total	3810	38.1 [µs]

Tabla 19. Latencias de la primera memoria intermedia.

Segunda, Tercera y Cuarta Capa Convolucional

Estas tres capas convolucionales se comportan de manera similar a la primera ya que reciben datos provenientes de una memoria intermedia y realizan sus 4 etapas correspondientes (convolución, activación, agrupamiento y almacenamiento). Es importante resaltar que, como la cantidad de filtros aumenta conforme se avanza en las capas convolucionales, el paquete de datos a la salida de los módulos de convolución también se ve incrementado y este tamaño de paquete permanece hasta que se alcanza el siguiente módulo de convolución.

Dentro de los módulos de memoria intermedia existe la señal de salida *ready_out*, la cual se encarga de informar al módulo de convolución que puede leer los datos que se encuentran a la salida. El módulo de memoria intermedia debe almacenar primero cierta cantidad de datos para que, conforme se avanza en la convolución, la memoria no envíe datos basura que se puedan encontrar en las localidades vacías.

Después del módulo de agrupamiento de la cuarta capa convolucional ya no es necesaria la utilización de una memoria intermedia, ya que los datos a la salida de este módulo pueden pasar directamente a la primera capa totalmente conectada.

En el Apéndice A se muestran los recursos utilizados por cada una de las capas convolucionales restantes junto con los módulos de agrupamiento y memorias intermedias, además, también se muestran sus latencias correspondientes.

4.3.5 Capas Totalmente Conectadas

Las capas totalmente conectadas se componen esencialmente por 3 etapas:

Figura 29. Etapas de una capa totalmente conectada. Fuente: Elaboración propia.

En la primera etapa, los datos entrantes son multiplicados por los pesos correspondientes de cada neurona y sumados junto con su correspondiente sesgo, en la segunda, a cada resultado de la suma se le aplica la función de activación ReLU, como en las capas convolucionales, y el resultado es almacenado en memoria.

Cabe mencionar que los pesos de este tipo de capas no son almacenados en un arreglo dentro de cada módulo, a diferencia de los módulos de convolución, ya que la cantidad de valores es muchísimo mayor, como se puede apreciar en la Tabla 5. Para almacenarlos se emplearon módulos de propiedad intelectual (IP, por sus siglas en inglés) de Vivado® llamados memoria de solo lectura (ROM, por sus siglas en inglés), estas memorias fueron diseñadas a la medida con el propósito de aprovechar de mejor manera los recursos disponibles.

Primera y Segunda Capa Totalmente Conectadas

La primera capa totalmente conectada recibe en total 175 datos de la cuarta capa convolucional y, al estar formada por 90 neuronas, recibe de su memoria ROM un total de 15,750 pesos, sin embargo, solo se toma una porción (25 datos) de estos 175 valores junto con los pesos correspondientes y se comienza a operar con ellos. Una vez que las multiplicaciones y sumas se han llevado a cabo, se descarta la porción entrante y se recibe una nueva, así hasta recibir los 175 datos.

Nombre	Slice LUTs	Slice Registers	Slice	BRAM
Blk_mem0	7	2	7	6.5

Tabla 20. Recursos utilizados por el módulo de memoria ROM de la primera capa totalmente conectada.

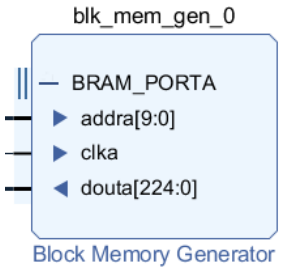


Figura 30. Módulo de memoria ROM para almacenar los pesos de la primera capa totalmente conectada. Fuente: Elaboración propia en Vivado®.

Blk_mem0	Ciclos de reloj	Tiempo
Latencia de salida	1	10 [ns]
Latencia total	1653	16.53 [µs]

Tabla 21. Latencias de la primera memoria ROM.

Al terminar la etapa de suma de productos, esta capa produce 90 datos de salida a los cuales se les aplica la función de activación ReLU y son almacenados para ser enviados posteriormente a la siguiente capa totalmente conectada. Nótese que este tipo de capas no involucran una etapa de agrupamiento después de la etapa de activación.

Nombre	Slice LUTs	Slice Registers	Multiplexores F7	Multiplexores F8	Slice	DSPs
Dense_1	5648	3209	403	120	1739	25

Tabla 22. Recursos utilizados por la primera capa totalmente conectada.

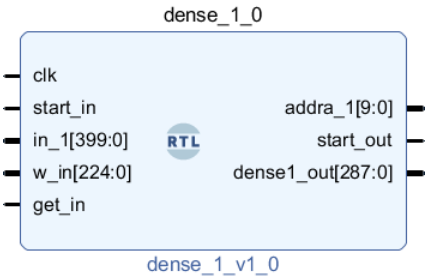


Figura 31. Módulo de la primera capa totalmente conectada. Fuente: Elaboración propia en Vivado®.

Dense_1	**Ciclos de reloj**	**Tiempo**
Latencia de salida	85	850 [ns]
Latencia total	1989	19.89 [µs]

Tabla 23. Latencias de la primera capa totalmente conectada.

La segunda capa totalmente conectada funciona de la misma manera que la primera, pero con su respectiva memoria de pesos y cantidad de neuronas, esta capa produce 84 datos de salida que son almacenados en el mismo módulo y enviados posteriormente a la tercera capa.

Nombre	**Slice LUTs**	**Slice Registers**	**Multiplexores F7**	**Multiplexores F8**	**Slice**	**DSPs**
Dense_2	5140	2941	384	108	1656	18

Tabla 24. Recursos utilizados por la segunda capa totalmente conectada.

Dense_2	**Ciclos de reloj**	**Tiempo**
Latencia de salida	20	200 [ns]
Latencia total	516	5.16 [µs]

Tabla 25. Latencias de la segunda capa totalmente conectada.

Tercera Capa Totalmente Conectada

La tercera capa totalmente conectada o capa de clasificación solo se diferencia de la primera y la segunda en la función de activación. En esta capa se emplea la función Softmax o función exponencial normalizada (10), la cual es una función utilizada para la clasificación multiclase [38].

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad para\ j = 1, \dots, k \qquad (10)$$

El módulo de esta capa solo lleva a cabo la suma de productos y sesgo y, al ser una capa con 19 neuronas, se obtienen 19 valores de salida a los cuales se les aplica esta función.

Nombre	Slice LUTs	Slice Registers	Multiplexores F7	Multiplexores F8	Slice	DSPs
Dense_3	749	344	33	1	255	14

Tabla 26. Recursos utilizados por el módulo de la tercera capa totalmente conectada.

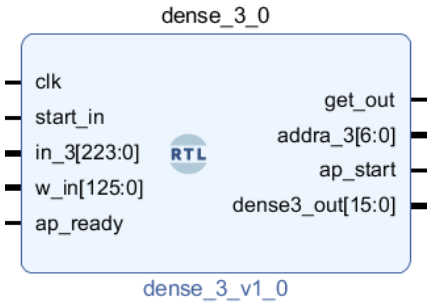


Figura 32. Módulo de la tercera capa totalmente conectada. Fuente: Elaboración propia en Vivado®.

Dense_3	Ciclos de reloj	Tiempo
Latencia de salida	18	180 [ns]
Latencia total	439	4.39 [µs]

Tabla 27. Latencias de la tercera capa totalmente conectada.

Función Softmax

Para emplear la función Softmax es necesario poder calcular la exponencial, desafortunadamente, la paquetería empleada para punto fijo en VHDL no cuenta con esta función, por lo que fue necesario emplear otras herramientas que nos permitieran calcular la exponencial de números en punto fijo con un formato específico. Para tal tarea se decidió utilizar Vivado® HLS, ya que es una herramienta que permite describir hardware a partir de otros lenguajes como C y C++ de una manera más rápida y sencilla.

El primer paso para crear la función exponencial fue definir el programa fuente (source, por su nombre en inglés) que la ejecutará, para ello se creó un archivo .cpp (C++) donde se incluyeron las bibliotecas y se definieron las variables y métodos necesarios.

Para manejar los valores en punto fijo fue necesario emplear la biblioteca *"ap_fixed.h"*, la cual ya se encuentra dentro de las bibliotecas disponibles en Vivado® y solo es necesario declararla, y la biblioteca *"hls_math.h"* con el fin de poder realizar la función exponencial.

Como el formato de salida de la tercera capa totalmente conectada es de 16 bits, se declaró una variable de entrada de 16 bits que contenga el mismo formato en punto fijo:

typedef ap_fixed<16, 8, AP_TRN, AP_SAT> FixedTypeIn;

Al recibir el dato en punto fijo, el siguiente paso fue convertirlo a punto flotante, lo cual se consigue mediante la función *"float()"*, para posteriormente efectuar la exponencial.

Finalmente, se convierte el resultado a punto fijo de 32 bits para continuar con la representación de punto fijo en la etapa de clasificación, cabe mencionar que se decidió utilizar 32 bits ya que los valores que pueden resultar de la exponencial se encuentran en un rango de entre los cientos de millones hasta valores muy pequeños cercanos a cero.

typedef ap_fixed<32, 24, AP_TRN, AP_SAT> FixedTypeOut;

Una vez terminado el programa fuente, el siguiente paso fue crear un programa de prueba (*test bench*) donde se podría evaluar la funcionalidad del programa fuente antes de comenzar con la síntesis.

Si los resultados del programa de prueba son satisfactorios, se puede continuar con la síntesis en C, la cual consta de sintetizar el código fuente de C++ en una implementación RTL.

Después que la síntesis es completada, Vivado® automáticamente crea reportes de síntesis donde se puede comprender y analizar el desempeño de la implementación. Como se creó un programa de prueba para propósitos de simulación, se puede utilizar la opción de co-simulación C/RTL con el objetivo de verificar que el funcionamiento del diseño RTL es idéntico al del programa fuente.

Por último, se exporta el diseño RTL en un formato que pueda ser utilizado por otras herramientas de Vivado®. En la Tabla 28 se pueden observar los recursos utilizados por la función exponencial desarrollada en Vivado® HLS.

Nombre	**Slice LUTs**	**Slice Registers**	**Multiplexores F7**	**Multiplexores F8**	**Slice**	**DSPs**
Exponential	1330	588	195	2	423	7

Tabla 28. Recursos utilizados por el módulo exponencial.

En la Figura 33 se pueden apreciar las señales de control del módulo, las cuales se pueden identificar ya que comienzan con el término "ap", estas son generadas automáticamente por Vivado® con el objetivo de manipularlo correctamente.

Señal	**Función**	**Tipo**
ap_start	Comenzar operaciones	Entrada
ap_done	Cálculos completados	Salida
ap_idle	Módulo en operación o inactivo	Salida
ap_ready	Listo para la siguiente entrada	Salida
ap_clk	Reloj del módulo	Entrada

ap_rst	Reinicio del módulo	Entrada
ap_vld	El resultado a la salida está listo	Salida

Tabla 29. Señales de control del módulo exponencial.

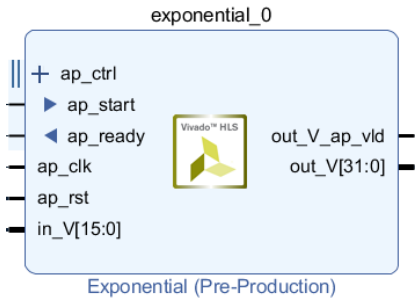


Figura 33. Módulo para calcular la exponencial. Fuente: Elaboración propia en Vivado®.

Exponential	**Ciclos de reloj**	**Tiempo**
Latencia de salida	18	180 [ns]
Latencia total	324	3.24 [µs]

Tabla 30. Latencias del módulo exponencial.

Clasificación

En esta última etapa se realiza la clasificación y se determina a qué clase pertenece el conjunto de vectores descriptivos mediante la función Softmax:

- Se reciben y almacenan las exponenciales calculadas del módulo exponencial.
- Al mismo tiempo, en otra variable se suman las exponenciales entrantes.
- Una vez recibidos los 19 datos, correspondientes a las 19 clases, se divide cada uno de ellos entre la suma total y se almacenan los resultados en un arreglo.
- Se revisan los resultados y el índice que contenga el valor más grande será la clase a la pertenecerá el conjunto de vectores.

Nombre	**Slice LUTs**	**Slice Registers**	**Multiplexores F7**	**Multiplexores F8**	**Slice**
Classification	11765	1675	18	9	3390

Tabla 31. Recursos utilizados por el módulo de clasificación.

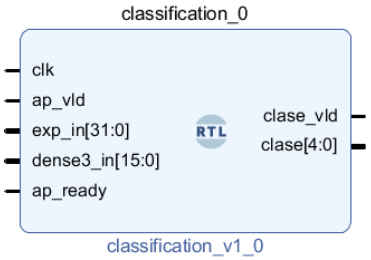


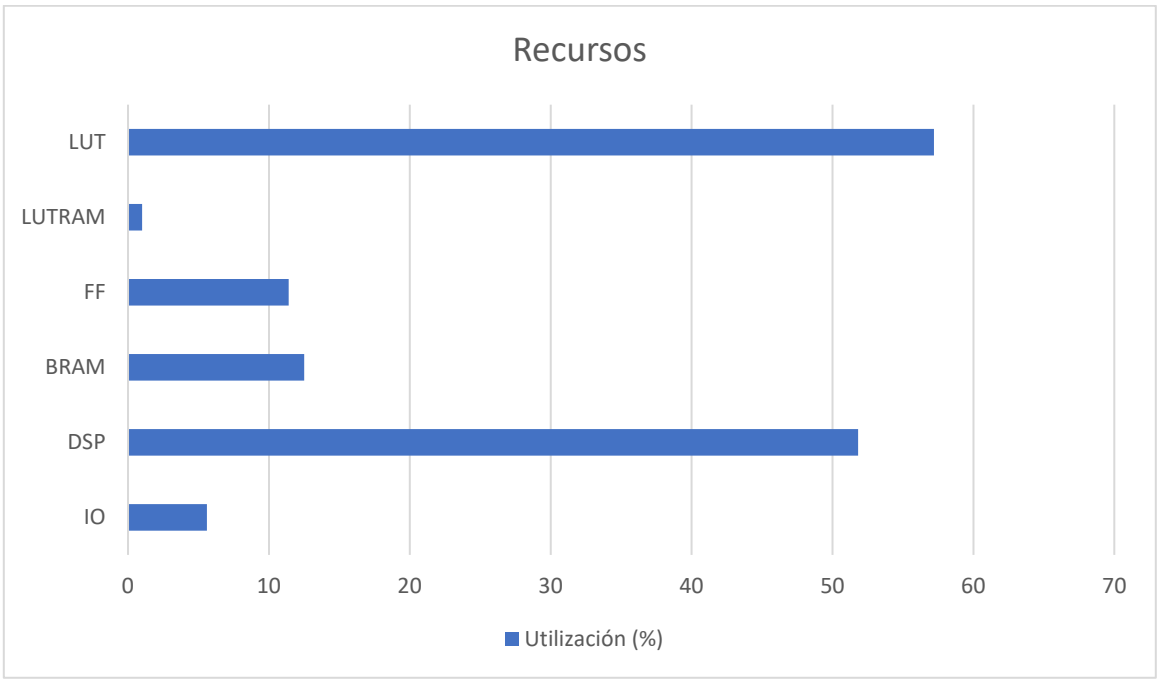
Figura 34. Módulo de clasificación. Fuente: Elaboración propia en Vivado®.

Como el módulo de clasificación solo produce una salida, la cual corresponde a la clase a la que pertenece el conjunto de vectores, la latencia de salida y total son las mismas.

Classification	**Ciclos de reloj**	**Tiempo**
Latencia de salida	11	110 [ns]
Latencia total	11	110 [ns]

Tabla 32. Latencias del módulo de clasificación.

En la Gráfica 5 se puede observar el total de los recursos utilizados por toda la RNC una vez implementada en el FPGA, además, en la Figura 35 se muestra el diagrama de tiempos de cada una de las capas que componen la red al realizar una sola clasificación.



Gráfica 5. Porcentaje total utilizado por la RNC implementada en el FPGA. Fuente: Elaboración propia.

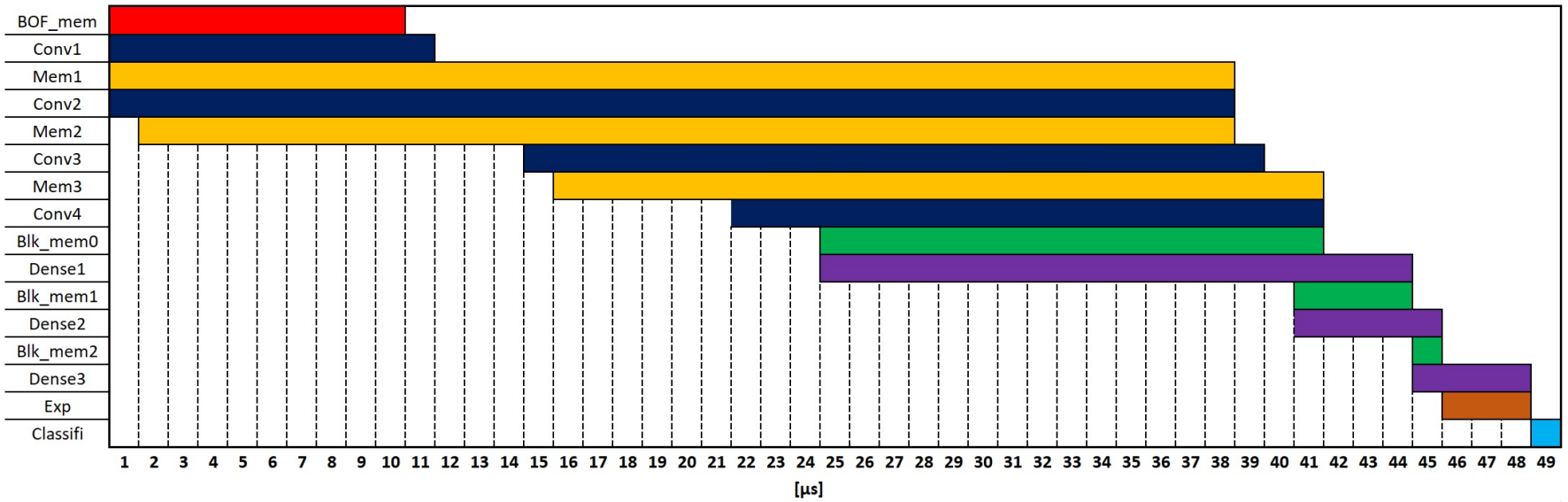


Figura 35. Diagrama de tiempo de las capas de la RNC. Fuente: Elaboración propia.

En el diagrama de tiempos, las etapas de pooling se encuentran embebidas en las capas de convolución. En la Tabla 33 se puede observar la latencia total de una sola clasificación de la red implementada en el FPGA.

	FPGA
Latencia	49.95 [µs]

Tabla 33. Latencia de la red para lograr una sola clasificación a partir de un conjunto de vectores

4.4 Resultados de Clasificación en el FPGA

Para evaluar el desempeño de la red implementada en el FPGA se tomaron de manera aleatoria el 5% de datos (60 conjuntos de vectores) pertenecientes al conjunto de prueba y se introdujeron a la red para clasificarlos. Es importante señalar que, debido a la cantidad de conjuntos de vectores (6156) y el tiempo que se necesita para clasificar solo uno de ellos dentro del simulador de Vivado® se decidió solo evaluar la cantidad antes mencionada.

En la Figura 35 se muestra la matriz de confusión de la RNC implementada en Python para clasificar los 60 conjuntos de vectores. Como se puede observar, se incluye al menos un conjunto por cada clase y solo se obtuvo un error de clasificación para la clase 7 (pirámide).

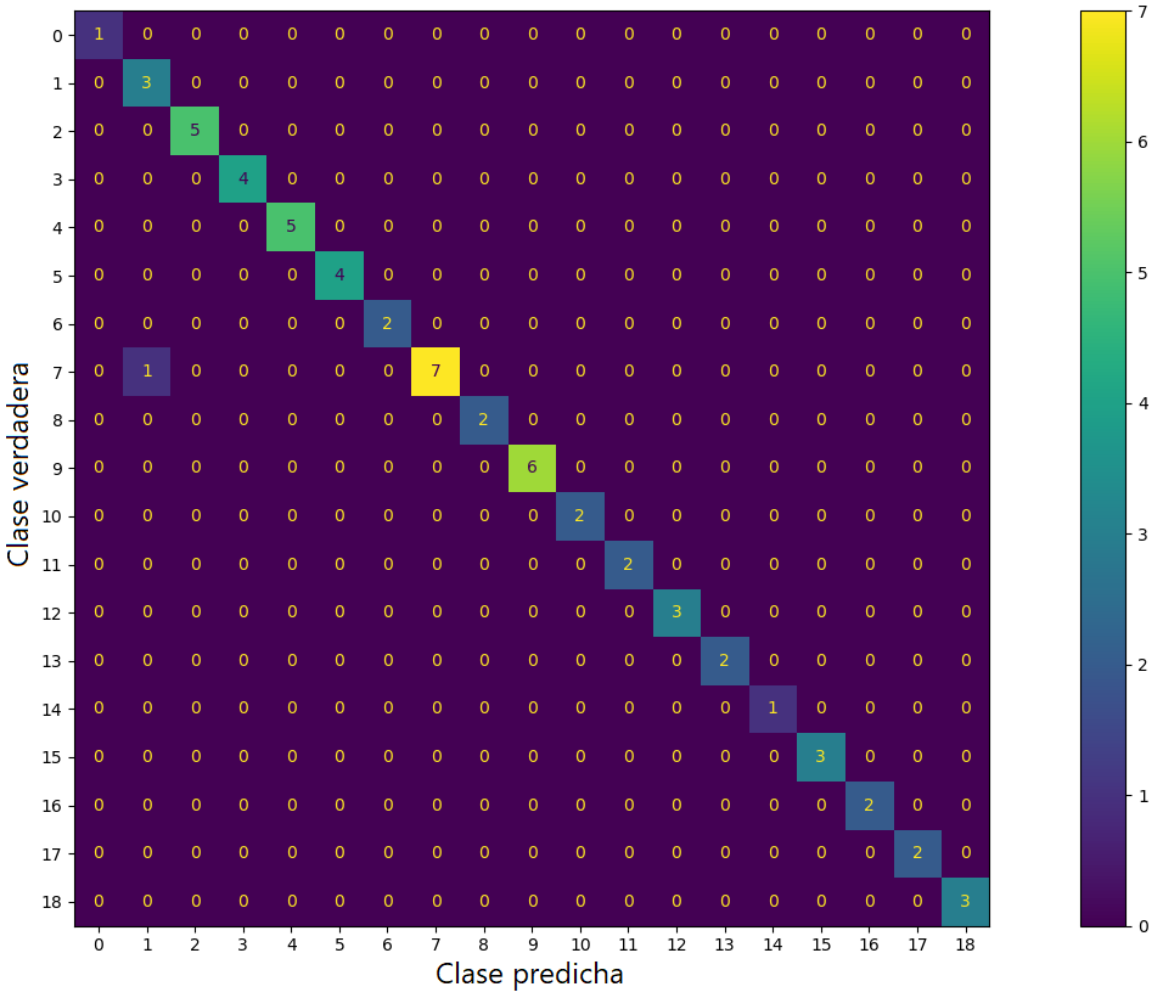


Figura 36. Matriz de confusión de la red implementada en Python de las 60 muestras. Fuente: Elaboración propia.

Las mismas 60 muestras se clasificaron mediante la implementación en el FPGA y se obtuvieron los resultados mostrados en la Figura 36. En este caso se obtuvieron dos errores de

clasificación, en la clase 0 (cubo) y en la clase 13 (corazón), nótese también que, el error producido por la red en Python para la clase 7 no sucede para la implementación en FPGA.

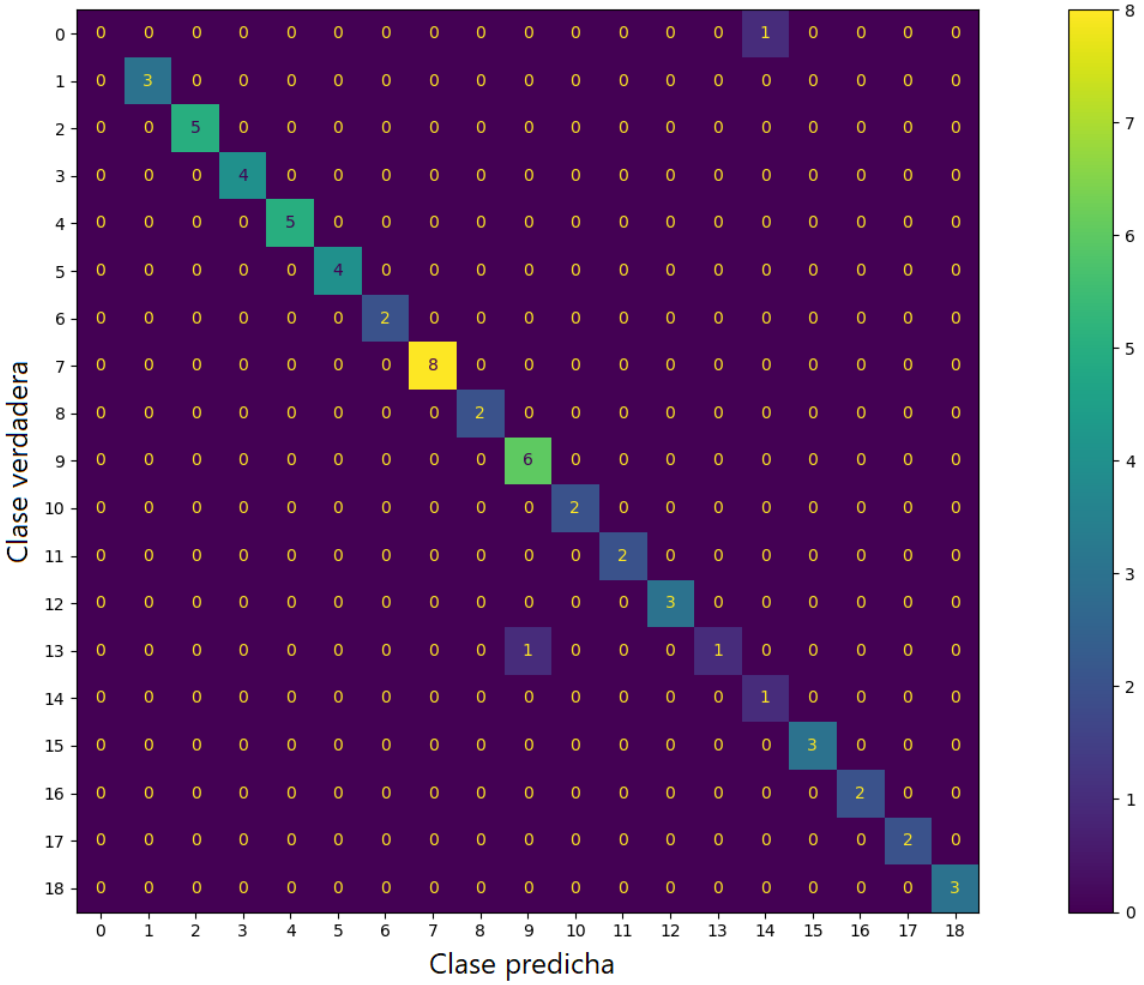


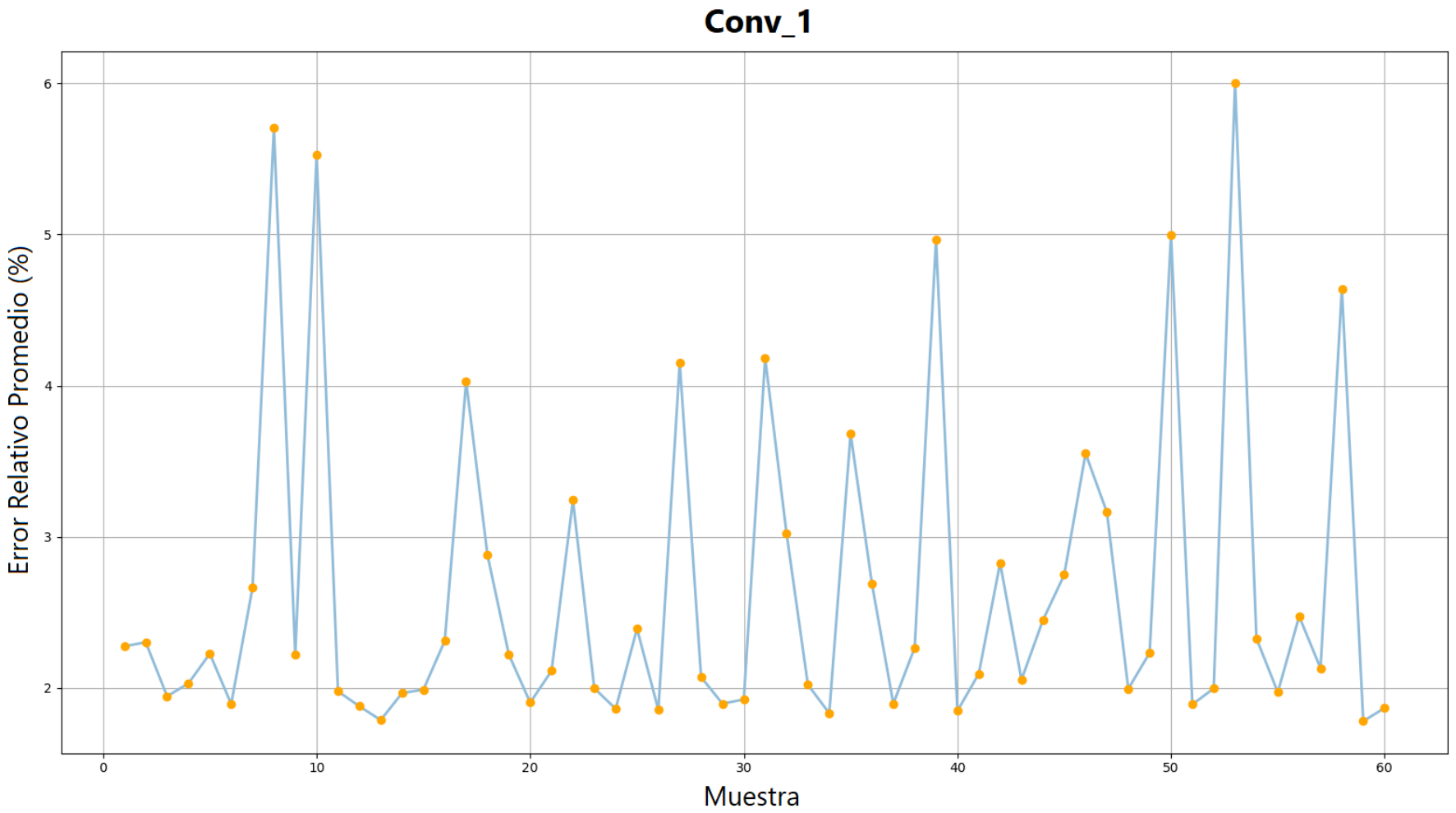
Figura 37. Matriz de confusión para la red implementada en el FPGA para las 60 muestras. Fuente: Elaboración propia.

Con base en estas clasificaciones, en la Tabla 34 se muestra la exactitud de clasificación de ambas implementaciones.

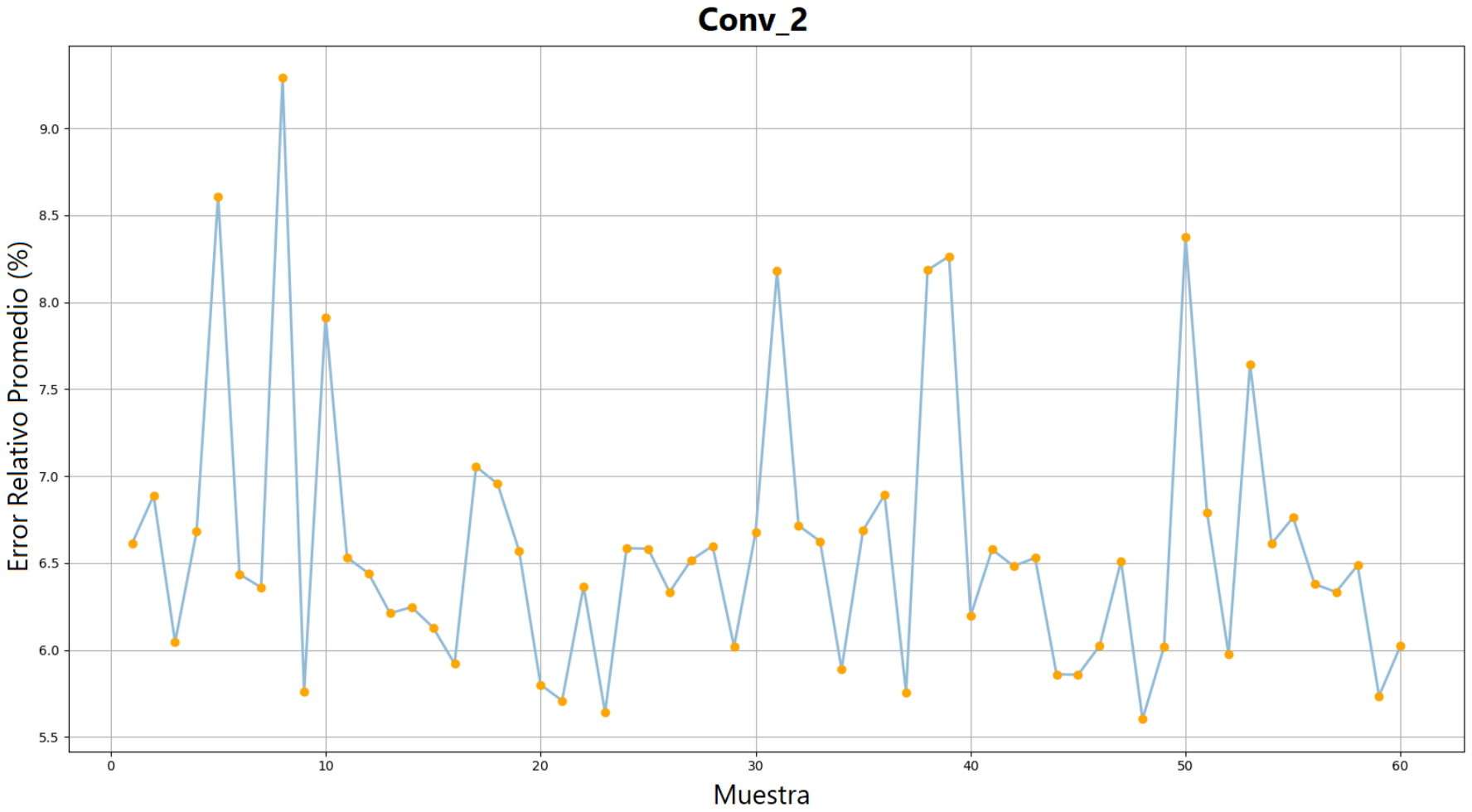
Implementación	**Exactitud**
Python	98.33%
FPGA	96.67%

Tabla 34. Exactitud de ambas implementaciones para las 60 muestras.

Adicionalmente, se calculó el porcentaje de error que produce cada uno de los módulos que conforman la red implementada en el FPGA en comparación con los resultados arrojados por la red en Python para cada una de las 60 clasificaciones. Los resultados de los módulos de las capas de convolución 1 y 2 se muestran en la Gráfica 6 y Gráfica 7, respectivamente.



Gráfica 6. Error relativo promedio de la primera convolución para las 60 muestras. Fuente: Elaboración propia.



Gráfica 7. Error relativo promedio de la segunda convolución para las 60 muestras. Fuente: Elaboración propia.

5 Conclusiones

Lo expuesto a lo largo de este trabajo permite arribar a las siguientes conclusiones:

El diseño de un entorno controlado para la adquisición de imágenes de los objetos facilitó en gran medida la misma adquisición y extracción del conjunto de vectores descriptivos ya que se tenía control sobre la iluminación y posicionamiento de los objetos, además, permitió tener un preprocesamiento más simple al tener un buen contraste entre objeto y fondo.

Gracias al proyecto de código abierto compartido por Digilent [33] y a las bibliotecas que posee Vivado® HLS para el procesamiento de imágenes, fue posible realizar el despliegue y tratamiento del vídeo de una manera más sencilla, desafortunadamente, no fue viable diseñar los módulos para la extracción de los conjuntos de vectores a través de la BOF.

A pesar de que se obtuvieron 6,840 conjuntos de vectores (360 por objeto) solo se utilizaron 6,156 (324 por objeto) ya que, cuando la cámara se encontraba en $\theta = 0°$, la perspectiva que presentaban algunos de los objetos era muy similar, lo cual ocasionaba que la red cometiera más errores durante el entrenamiento.

Como se comentó en el apartado 4.2, se tomó como base la arquitectura de la red LeNet-5 ya que es una red que se ha implementado en FPGA anteriormente [10], [13] y, afortunadamente, se obtuvieron resultados favorables sin realizar ninguna modificación adicional a la adaptación de las dimensiones de las capas con las dimensiones de los datos de entrada, quedando solo la tarea de mejorar su desempeño con base a los resultados obtenidos al aplicar alguna técnica de regularización.

Tras el análisis de la matriz de confusión de la implementación de la red en Python para todas las muestras, se puede observar que los mayores errores de clasificación pertenecen a las clases “1” (cubo) y “7” (pirámide), esto debido a que, cuando la cámara se encuentra a 90° en θ, ambos objetos muestran una forma muy similar (cuadrado), lo cual provoca estas confusiones.

Durante la implementación y pruebas realizadas de la red en el FPGA, se compararon los resultados arrojados por cada capa con los resultados de la red en Python y se calculó el error relativo promedio, mostrando que conforme se avanza en las capas de la red en el FPGA, el error que se produce también se ve incrementado debido a que este se va acumulando. Sin embargo, aunque el error llega a ser bastante grande en la última capa, no afecta considerablemente la exactitud de clasificación. No obstante, es recomendable hacer un análisis más completo utilizando el 100% del conjunto de prueba para comprobar que la métrica de exactitud se mantiene arriba del 90%.

Por último, como se ha podido comprobar, se logró no solo diseñar una red neuronal convolucional capaz de clasificar objetos con una gran exactitud a partir de un conjunto de vectores, sino que también se consiguió implementarla en un FPGA con una pérdida en la exactitud de tan solo el 2%, obteniendo un 96% en total y una velocidad 760 veces mayor, asimismo, esta implementación no solo resulta ser más rápida, sino también más eficiente energéticamente y una

opción más que interesante, en comparación de las computadoras de propósito general, para la implementación de redes neuronales.

6 Trabajo a Futuro

Este proyecto abre un gran abanico de posibilidades de investigación y perfeccionamiento para sistemas de clasificación basados en vectores descriptivos, los cuales se enlistan a continuación:

1. El proyecto que comparte Digilent para la adquisición de vídeo emplea distintas funciones para su uso en general, sin embargo, acotando el proyecto para obtener una implementación más eficiente, contribuiría a tener un mejor uso de los recursos disponibles en el FPGA y así unificarlo con la red neuronal convolucional.
2. Es necesaria una mayor investigación y experimentación en el uso de Vivado® HLS y los distintos protocolos de transmisión de datos que emplea, esto con el objetivo de diseñar los módulos faltantes para la extracción de los vectores descriptivos a partir de las imágenes binarizadas y su posterior propagación hacia la red neuronal convolucional.
3. Es más que conveniente experimentar con otro tipo de redes neuronales, tales como totalmente conectadas o redes neuronales recurrentes, ya que es posible encontrar una red que emplee una menor cantidad de capas y, por ende, una cantidad menor de pesos y recursos.
4. En este proyecto solo se emplearon 19 objetos para clasificar, es aconsejable experimentar y aumentar la cantidad de objetos o clases que la red puede manejar sin disminuir su exactitud. Además, es preciso realizar otras pruebas de la red en el FPGA incrementando la cantidad de bits que se utilizan para la representación de los pesos de cada capa y el impacto que tiene en su desempeño.
5. Por último, la tarjeta al poseer conectividad a través del puerto Ethernet, sería muy interesante diseñar alguna manera de poder actualizar los pesos de las capas por medio de Internet. Esto con la intención de “actualizar” la red si es necesario clasificar un nuevo objeto.

7 Referencias

[1] "The Fourth Industrial Revolution: what it means and how to respond | World Economic Forum." https://www.weforum.org/agenda/2016/01/the-fourth-industrial-revolution-what-it-means-and-how-to-respond/ (accessed Nov. 15, 2021).

[2] R. Y. Zhong, X. Xu, E. Klotz, and S. T. Newman, "Intelligent Manufacturing in the Context of Industry 4.0: A Review," *Engineering*, vol. 3, no. 5, pp. 616–630, 2017, doi: 10.1016/J.ENG.2017.05.015.

[3] "Importance of Artificial Intelligence in Manufacturing | HCL Blogs." https://www.hcltech.com/blogs/manufacturing-and-artificial-intelligence (accessed Jun. 19, 2020).

[4] X. Yao, J. Zhou, J. Zhang, and C. R. Boer, "From Intelligent Manufacturing to Smart Manufacturing for Industry 4.0 Driven by Next Generation Artificial Intelligence and Further on," *Proceedings - 2017 5th International Conference on Enterprise Systems: Industrial Digitalization by Enterprise Systems, ES 2017*, pp. 311–318, 2017, doi: 10.1109/ES.2017.58.

[5] B. Ślusarczyk, "Industry 4.0 – Are we ready?," *Polish Journal of Management Studies*, vol. 17, no. 1, pp. 232–248, 2018, doi: 10.17512/pjms.2018.17.1.19.

[6] A. Alassadi, T. Ivanauskas, S. Kamilla, K. Examiner, and N. Gador, "Classification Performance Between Machine Learning and Traditional Programming in Java Title Classification performance between machine learning and traditional programming in Java".

[7] E. Nurvitadhi *et al.*, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?," *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 5–14, 2017, doi: 10.1145/3020078.3021740.

[8] M. Bettoni, G. Urgese, Y. Kobayashi, E. Macii, and A. Acquaviva, "A convolutional neural network fully implemented on FPGA for embedded platforms," in *Proceedings - 2017 1st New Generation of CAS, NGCAS 2017*, Sep. 2017, pp. 49–52. doi: 10.1109/NGCAS.2017.16.

[9] K. Abdelouahab, "Accelerating CNN inference on FPGAs : A Survey arXiv : 1806 . 01683v1 [cs . DC] 26 May 2018 Accelerating CNN inference on FPGAs : A Survey," no. May, 2018.

[10] R. A. Solovyev, A. A. Kalinin, A. G. Kustov, D. v. Telpukhov, and V. S. Ruhlov, "FPGA Implementation of Convolutional Neural Networks with Fixed-Point Calculations," pp. 1–9, 2018, doi: 10.1109/EIConRus.2019.8656778.

[11] D. Gschwend, "ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network," no. August 2016, 2020, [Online]. Available: http://arxiv.org/abs/2005.06892

[12] G. Wei, Y. Hou, Q. Cui, G. Deng, X. Tao, and Y. Yao, “YOLO Acceleration using FPGA Architecture,” *2018 IEEE/CIC International Conference on Communications in China, ICCC 2018*, no. Iccc, pp. 734–735, 2019, doi: 10.1109/ICCChina.2018.8641256.

[13] D. Rongshi and T. Yongming, “Accelerator Implementation of Lenet-5 Convolution Neural Network Based on FPGA with HLS,” *2019 3rd International Conference on Circuits, System and Simulation, ICCSS 2019*, pp. 64–67, 2019, doi: 10.1109/CIRSYSSIM.2019.8935599.

[14] J. Sanchez, N. Soltani, R. Chamarthi, A. Sawant, and H. Tabkhi, “A Novel 1D-Convolution Accelerator for Low-Power Real-time CNN processing on the Edge,” *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018*, pp. 0–7, 2018, doi: 10.1109/HPEC.2018.8547530.

[15] J. Xu, S. Li, J. Jiang, and Y. Dou, “A Simplified Speaker Recognition System Based on FPGA Platform,” *IEEE Access*, vol. 8, pp. 1507–1516, 2020, doi: 10.1109/ACCESS.2019.2944644.

[16] J. H. Kaas and P. Balaram, “Current research on the organization and function of the visual system in primates,” *Eye and Brain*, vol. 6, no. Suppl 1, p. 1, 2014, doi: 10.2147/EB.S64016.

[17] W. K. Pratt, *Digital Image Processing*, vol. 19, no. 3. 1994. doi: 10.1080/03043799408928319.

[18] M. Alasdair, “An Introduction to Digital Image Processing with Matlab, Notes for SCM2511 Image Processing 1,” *Jurnal Ilmiah Elite Elektro*, vol. 2, no. 2, pp. 83–87, 2014.

[19] E. Wolfgang, *Introduction to AI.pdf*. 2009.

[20] D. Jakhar and I. Kaur, “Artificial intelligence, machine learning and deep learning: definitions and differences,” *Clinical and Experimental Dermatology*, vol. 45, no. 1, pp. 131–132, 2020, doi: 10.1111/ced.14029.

[21] J. Schmidhuber, “Deep Learning in neural networks: An overview,” *Neural Networks*, vol. 61, pp. 85–117, 2015, doi: 10.1016/j.neunet.2014.09.003.

[22] K. O’Shea and R. Nash, “An Introduction to Convolutional Neural Networks,” pp. 1–11, 2015, [Online]. Available: http://arxiv.org/abs/1511.08458

[23] “Redes Neuronales Convolucionales - MATLAB & Simulink.” https://la.mathworks.com/solutions/deep-learning/convolutional-neural-network.html (accessed Jun. 20, 2020).

[24] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015, doi: 10.1038/nature14539.

[25] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size,” 2016, Accessed: Feb. 24, 2020. [Online]. Available: https://github.com/DeepScale/SqueezeNet

[26] M. Peña-Cabrera, M. Ontiveros, R. Osorio, G. Lefranc, and V. Lomas, “Contourn descriptor generator algorithm implemented in embedded system,” *2017 CHILEAN Conference on*

Electrical, Electronics Engineering, Information and Communication Technologies, CHILECON 2017 - Proceedings, vol. 2017-Janua, pp. 1–8, 2017, doi: 10.1109/CHILECON.2017.8229681.

[27] "What is an FPGA? Field Programmable Gate Array." https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html (accessed Apr. 02, 2021).

[28] R. Elliot Crockett, Luis, "The_Zynq_Book_ebook," 2014, [Online]. Available: All Papers/E/Elliot,Ross 2014 - The_Zynq_Book_ebook.pdf

[29] O. K. Chinedu, E. C. Genevera, and O. O. Akinyele, "Hardware description language (HDL): An efficient approach to device independent designs for VLSI market segments," *3rd IEEE International Conference on Adaptive Science and Technology, ICAST 2011, Proceedings*, no. November, pp. 262–267, 2011, doi: 10.1109/ICASTech.2011.6145181.

[30] M. Leinenger, "UC San Diego UC San Diego Electronic Theses and Dissertations by," *UC San Diego Electronic Theses and Dissertations*, vol. 290, p. 214, 2016.

[31] VivadoUG1, "Vivado Design Suite User Guide IP Subsystems," vol. 901, pp. 1–120, 2013.

[32] "Pcam 5C: 5 MP Fixed-Focus Color Camera Module - Digilent." https://store.digilentinc.com/pcam-5c-5-mp-fixed-focus-color-camera-module/ (accessed Apr. 04, 2021).

[33] "Zybo Z7 Pcam 5C Demo - Digilent Reference." https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-z7-pcam-5c-demo/start (accessed Apr. 04, 2021).

[34] Y. LeCun *et al.*, "Backpropagation applied to digit recognition," *Neural computation*, vol. 1, no. 4. pp. 541–551, 1989. [Online]. Available: https://www.ics.uci.edu/~welling/teaching/273ASpring09/lecun-89e.pdf

[35] A. Finnerty and H. Ratigner, "Reduce Power and Cost by Converting from Floating Point to Fixed Point (WP491)," 2017, Accessed: Sep. 05, 2021. [Online]. Available: www.xilinx.com1

[36] "Fixed-Point vs. Floating-Point Digital Signal Processing | Analog Devices." https://www.analog.com/en/technical-articles/fixed-point-vs-floating-point-dsp.html (accessed Sep. 05, 2021).

[37] "GitHub - francof2a/fxpmath: A python library for fractional fixed-point (base 2) arithmetic and binary manipulation with Numpy compatibility." https://github.com/francof2a/fxpmath#readme (accessed Sep. 05, 2021).

[38] "Multi-Class Neural Networks: Softmax | Machine Learning Crash Course." https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax?hl=es_419 (accessed Sep. 06, 2021).

Apéndice A

Nombre	Slice LUTs	Slice Registers	Multiplexores F7	Multiplexores F8	Slice	DSPs
Conv_2	461	272	67	25	175	10
Conv_3	558	399	50	0	216	16
Conv_4	2993	814	791	346	925	24

Tabla 35. Recursos utilizados por los módulos de convolución y activación.

Conv_2	**Ciclos de reloj**	**Tiempo**
Latencia de salida	36	360 [ns]
Latencia total	3786	37.86 [µs]
Conv_3		
Latencia de salida	52	520 [ns]
Latencia total	2480	24.8 [µs]
Conv_4		
Latencia de salida	104	1.04 [µs]
Latencia total	1973	19.73 [µs]

Tabla 36. Latencias de los módulos de convolución.

Nombre	Slice LUTs	Slice Registers	Slice
Max_pool2	130	258	63
Max_pool3	194	386	101
Max_pool4	403	802	194

Tabla 37. Recursos utilizados por los módulos de agrupamiento.

Max_pool2	**Ciclos de reloj**	**Tiempo**
Latencia de salida	72	720 [ns]
Latencia total	3736	37.36 [µs]
Max_pool3		
Latencia de salida	104	1.04 [µs]
Latencia total	2406	24.06 [µs]
Max_pool4		
Latencia de salida	208	2.08 [µs]
Latencia total	1691	16.91 [µs]

Tabla 38. Latencias de los módulos de máximo agrupamiento.

Nombre	**Slice LUTs**	**Slice Registers**	**Slice**	**BRAM**
Mem_2	25	24	9	2
Mem_3	22	22	9	3

Tabla 39. Recursos utilizados por las memorias intermedias.

Mem_2	**Ciclos de reloj**	**Tiempo**
Latencia de salida	13	130 [ns]
Latencia total	3722	37.22 [µs]
Mem_3		
Latencia de salida	26	260 [ns]
Latencia total	2577	25.77 [µs]

Tabla 40. Latencias de las memorias intermedias.

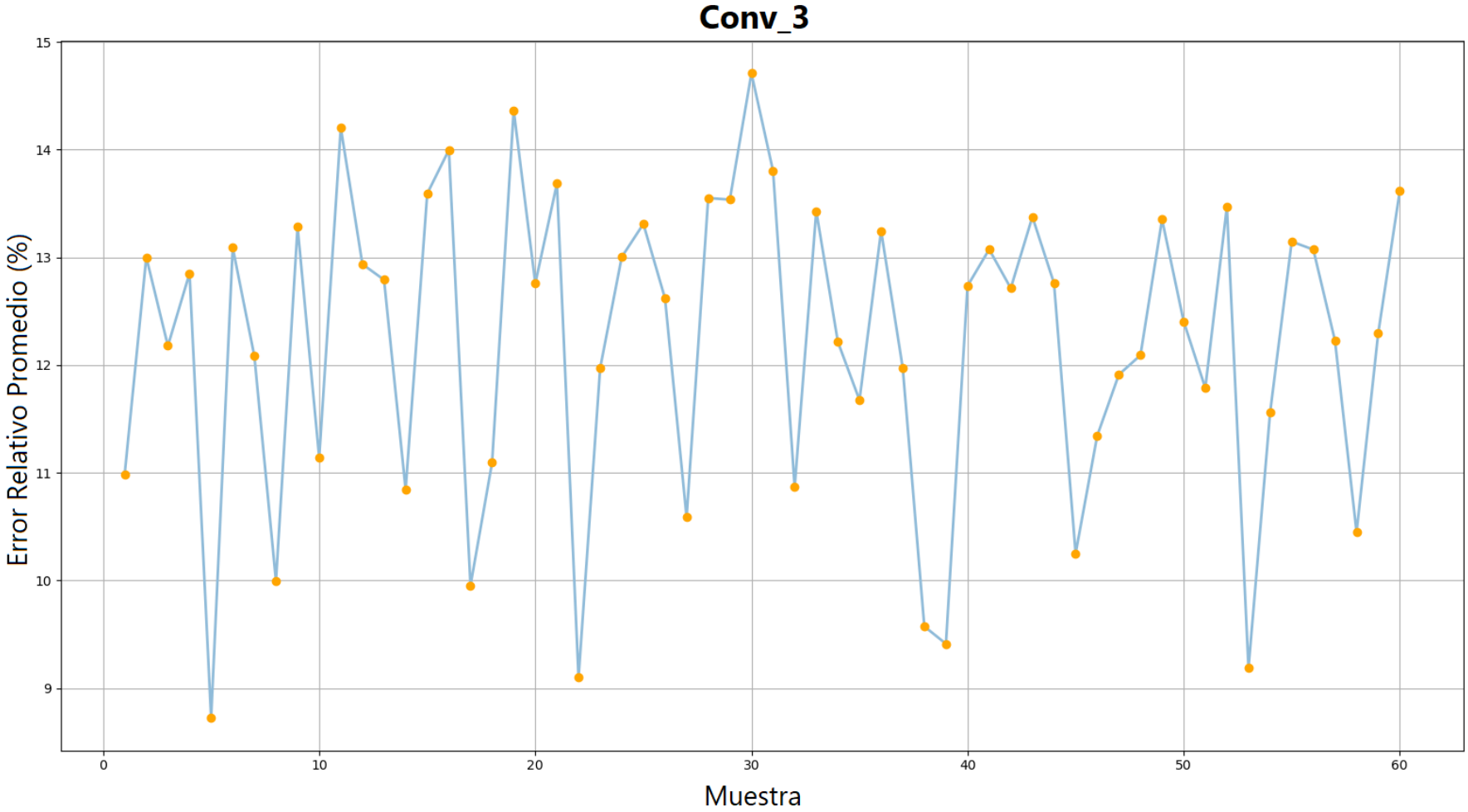
Nombre	**Slice LUTs**	**Slice Registers**	**Slice**	**BRAM**
Blk_mem1	6	2	4	2.5
Blk_mem2	2	2	1	2

Tabla 41.Recursos utilizados por las memorias de las capas totalmente conectadas 2 y 3, respectivamente.

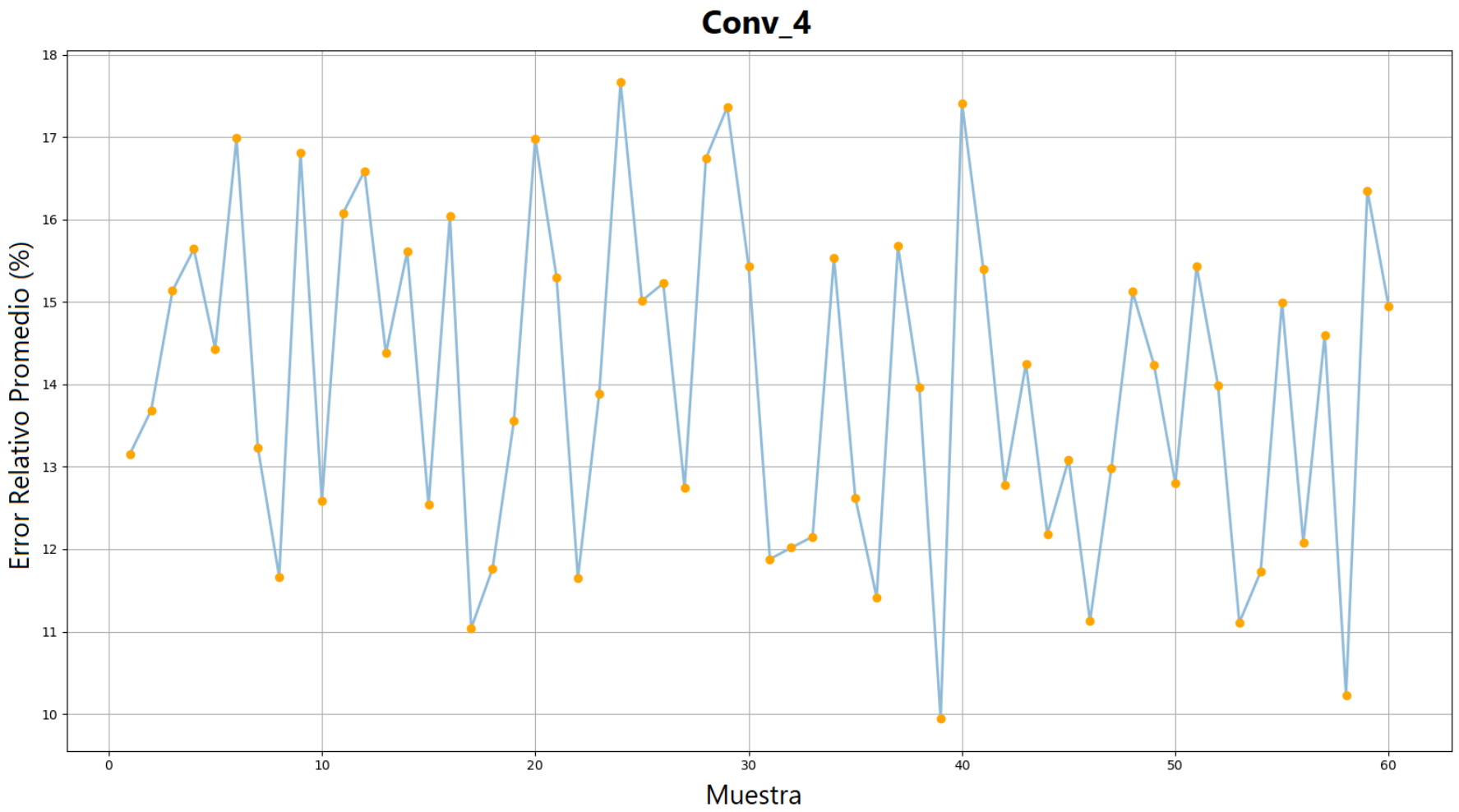
Blk_mem1	**Ciclos de reloj**	**Tiempo**
Latencia de salida	1	10 [ns]
Latencia total	418	4.18 [µs]
Blk_mem2		
Latencia de salida	1	10 [ns]
Latencia total	112	1.12 [µs]

Tabla 42. Latencias de las memorias de las capas totalmente conectadas 2 y 3.

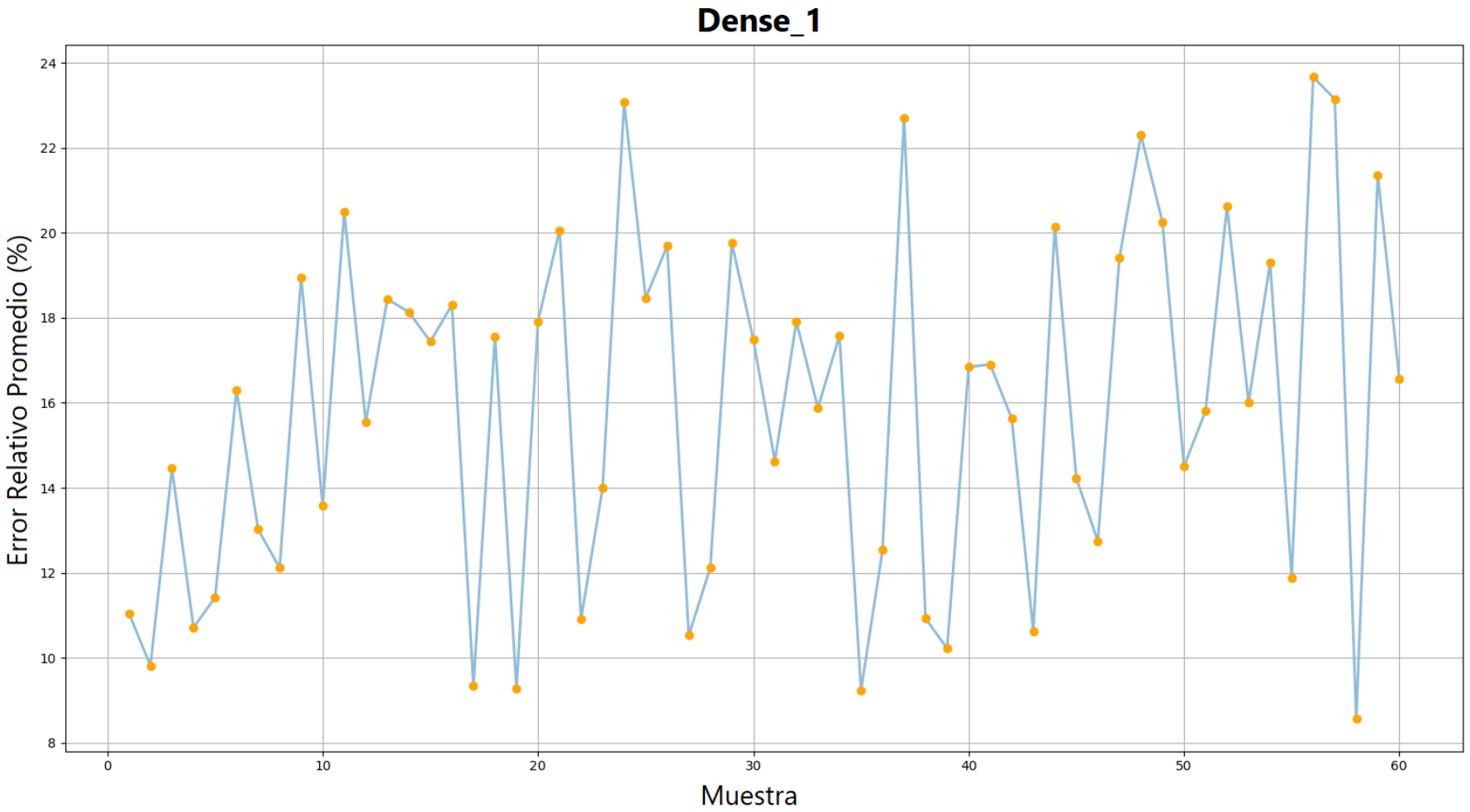
Apéndice B



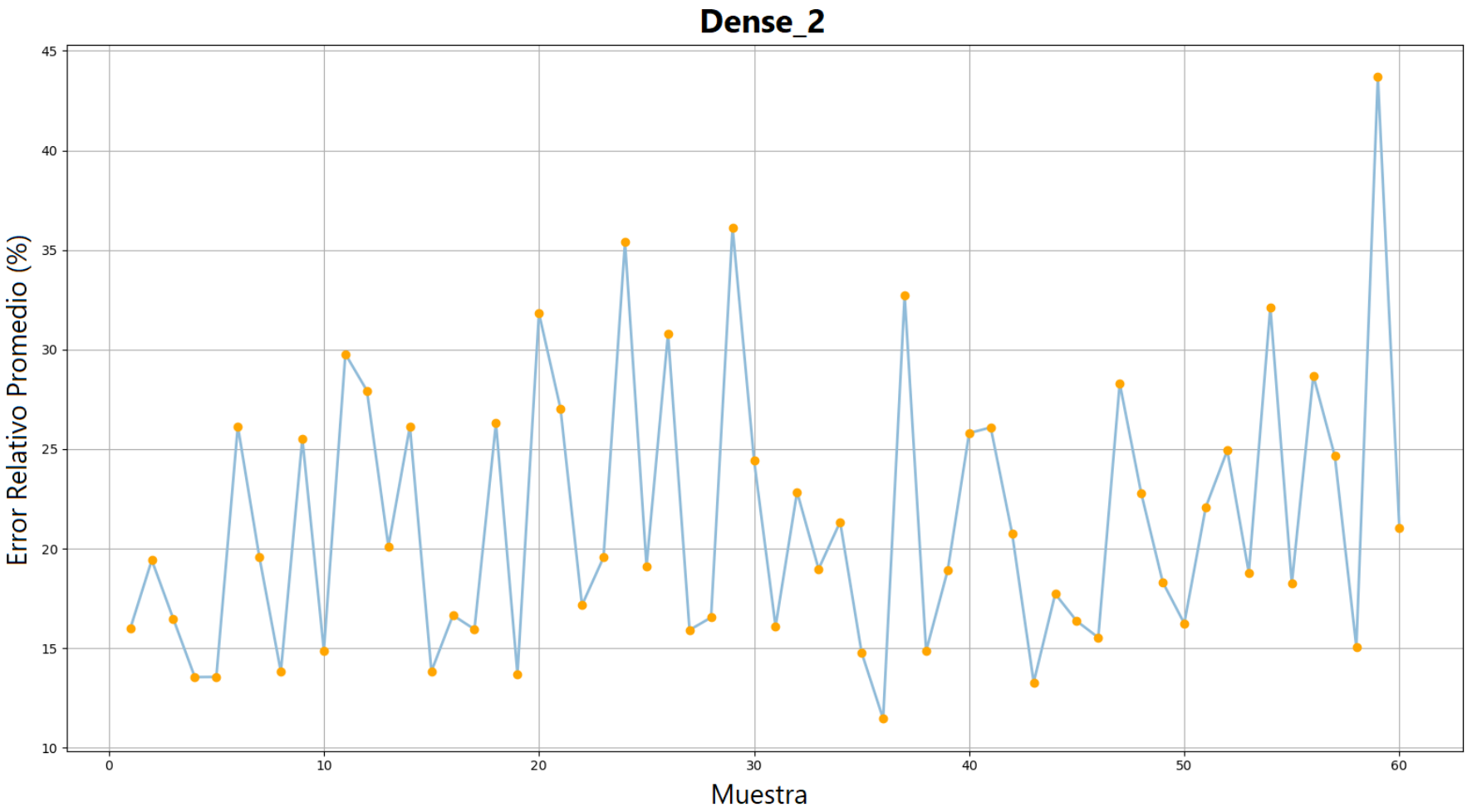
Gráfica 8. Error relativo promedio de la tercera convolución para las 60 muestras. Fuente: Elaboración propia.



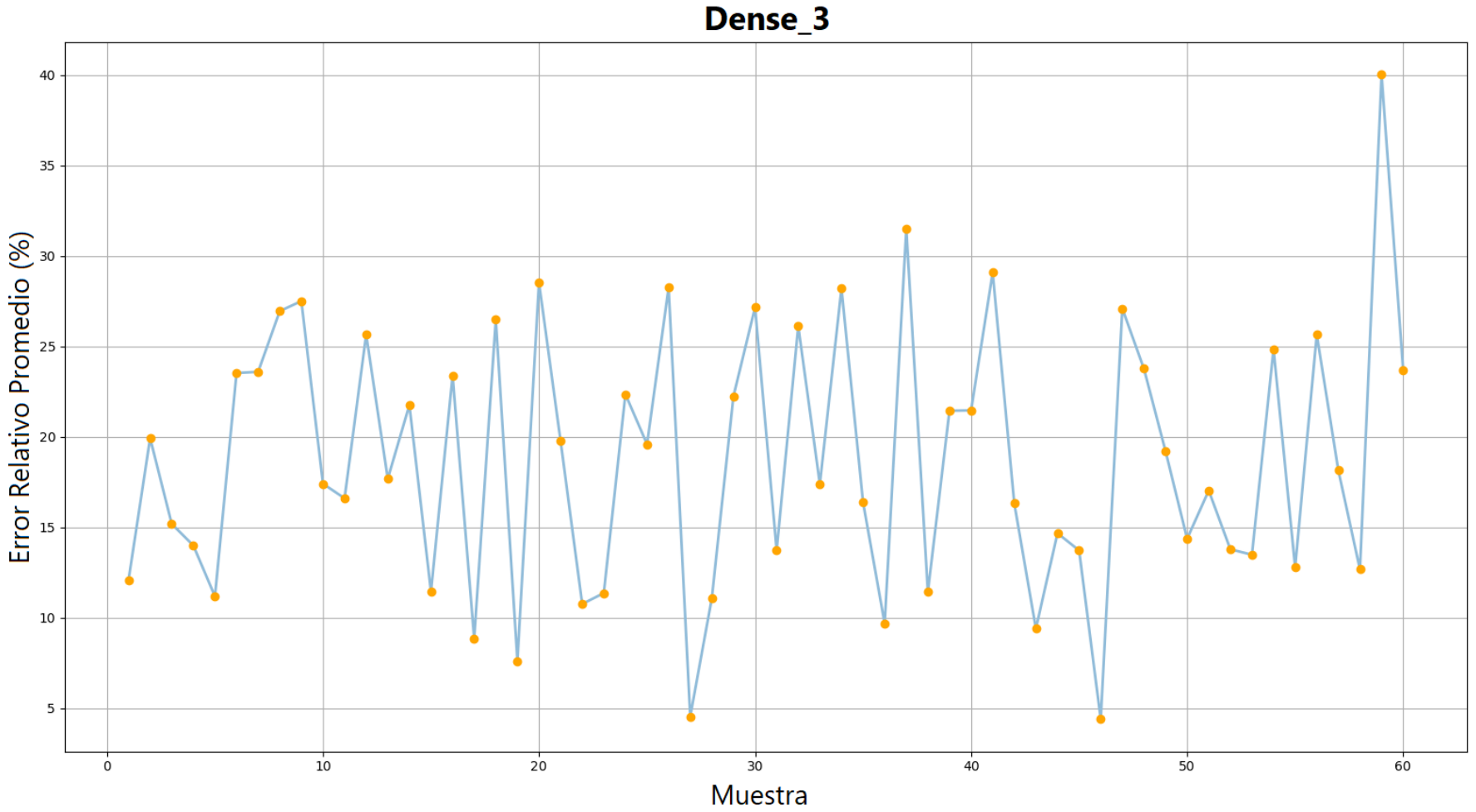
Gráfica 9. Error relativo promedio de la cuarta convolución para las 60 muestras. Fuente: Elaboración propia.



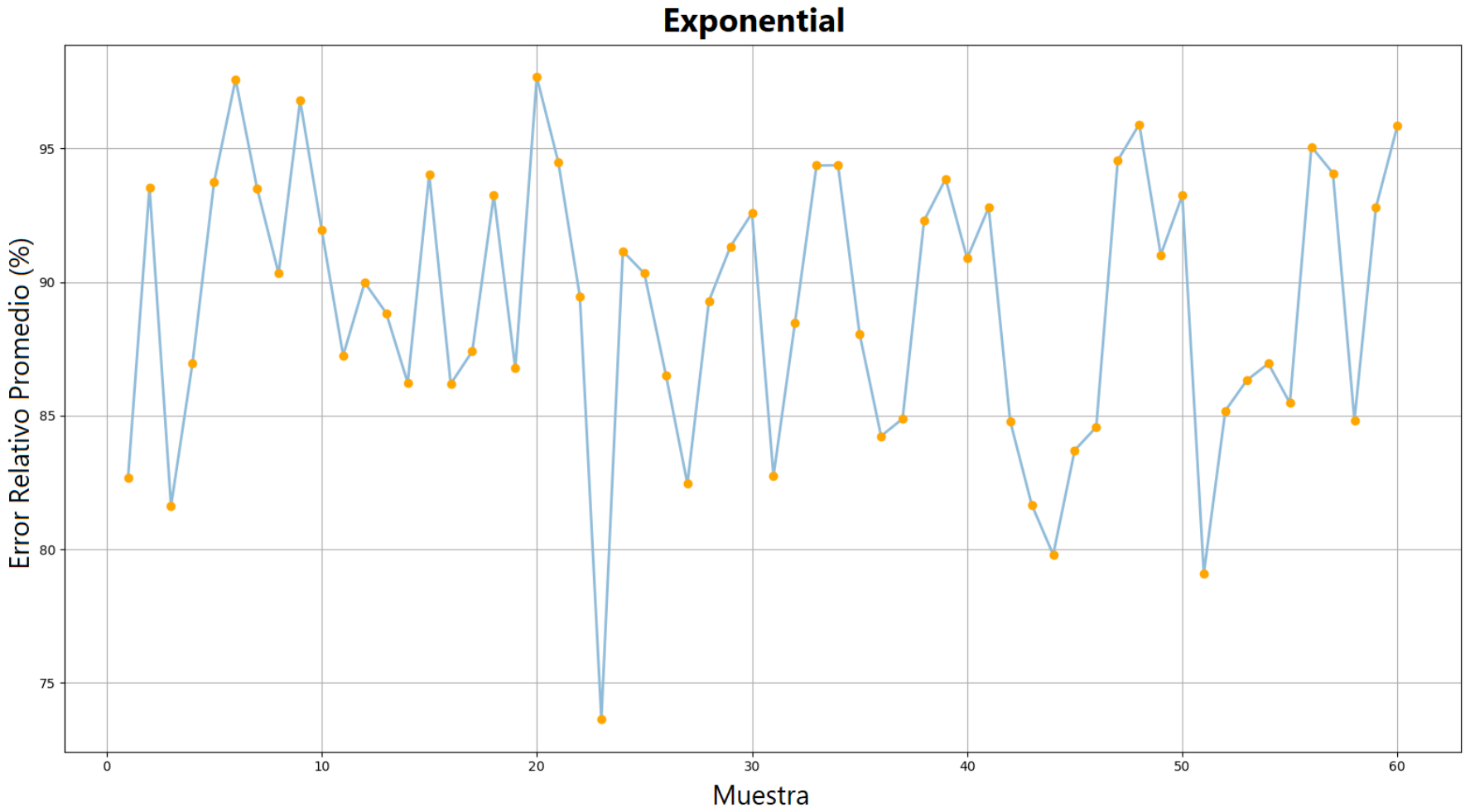
Gráfica 10. Error relativo promedio de la primera capa totalmente conectada para las 60 muestras. Fuente: Elaboración propia.



Gráfica 11. Error relativo promedio de la segunda capa totalmente conectada para las 60 muestras. Fuente: Elaboración propia.



Gráfica 12. Error relativo promedio de la tercera capa totalmente conectada para las 60 muestras. Fuente: Elaboración propia.



Gráfica 13. Error relativo promedio de la función exponencial para las 60 muestras. Fuente: Elaboración propia.