UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

IMPLEMENTACIÓN DE REDES NEURONALES PARA TOMOGRAFÍA DE EMISIÓN DE POSITRONES MEDIANTE FPGA

Autor: Jorge García Martínez

Tutor: Rafael Gadea Gironés

Trabajo Fin de Máster presentado en el Departamento de Ingeniería Electrónica de la Universitat Politècnica de València para la obtención del Título de Máster Universitario en Ingeniería de Sistemas Electrónicos

Curso 2019-20

Valencia, Enero de 2020

Resumen

Hoy en día existe gran controversia en la selección de dispositivos para la generación de algoritmos de Inteligencia Artificial y más en concreto los de Deep Learning, ya que necesitan grandes requerimientos computacionales. En este trabajo se propone la implementación de diferentes redes neuronales empleadas en Tomografía por Emisión de Positrones (PET) para la reconstrucción de imágenes sobre dispositivos FPGA. Estos dispositivos, a priori, están dotados de gran flexibilidad en los diseños, bajo consumo y alta capacidad de integración en sistemas embebidos.

Para ello, se empleará la plataforma OpenVINO. Ésta es una herramienta que integra todo el flujo de trabajo partiendo de una red pre-entrenada hasta su inferencia sobre el dispositivo seleccionado, en este caso una FPGA de Intel (PAC Arria 10 GX). Su funcionamiento se basa en obtener una representación intermedia de la red en cuestión con ayuda de la herramienta Model Optimizer, esta representación será procesada por un motor de inferencia que se encargará de realizar el proceso de ejecución de la red de manera iterativa. Finalmente, con ayuda del complemento Benchmark se puede obtener los resultados de througput y latencia en el proceso de inferencia.

En primer lugar, se realiza una primera comprobación del flujo de trabajo de la herramienta con una red convolucional entrenada para clasificar números escritos a mano. Finamente, se realiza todo el proceso para la ejecución de dos redes. La primera de ellas es una red convolucional capaz de predecir la profundidad del punto de choque del positrón en un sistema PET. La segunda es una red densa que trata de predecir el error cometido en la medida de la coordenada Z del punto de choque del positrón con respecto a un método analítico.

Tras realizar todo el proceso, se han podido observar las ventajas de la utilización de OpenVINO a la hora de implementar redes sobre plataformas heterogéneas con gran rapidez. Sin embargo, al tratarse de una herramienta con poco recorrido, está sujeta a mejoras como es la generación de bitstreams para FPGA que soporten mayor cantidad de capas diferentes.

Resum

Hui dia existeix gran controvèrsia en la selecció de dispositius per a la generació d'algorismes d'Intelligència Artificial i més en concret els de Deep Learning, ja que necessiten grans requeriments computacionals. En aquest treball es proposa la implementació de diferents xarxes neuronals empleades en Tomografia per Emissió de Positrons (PET) per a la reconstrucció d'imatges sobre dispositius FPGA. Aquests dispositius, a priori, estan dotats de gran flexibilitat en els dissenys, sota consum i alta capacitat d'integració en sistemes embeguts.

Per a això, s'emprarà la plataforma OpenVINO. Aquesta és una eina que integra tot el flux de treball partint d'una xarxa pre-entrenada fins a la seua inferència sobre el dispositiu seleccionat, en aquest cas una FPGA d'Intel (PAC Arria 10 GX). El seu funcionament es basa a obtindre una representació intermèdia de la xarxa en qüestió amb ajuda de l'eina Model Optimizer, aquesta representació serà processada per un motor d'inferència que s'encarregarà de realitzar el procés d'execució de la xarxa de manera iterativa. Finalment, amb ajuda del complement Benchmark es pot obtindre els resultats de througput i latència en el procés d'inferència.

En primer lloc, es realitza una primera comprovació del flux de treball de l'eina amb una xarxa convolucional entrenada per a classificar números escrits a mà. Finament, es realitza tot el procés per a l'execució de dues xarxes. La primera d'elles és una xarxa convolucional capaç de predir la profunditat del punt de xoc del positró en un sistema PET. La segona és una xarxa densa que tracta de predir l'error comés en la mesura de la coordenada Z del punt de xoc del positró respecte a un mètode analític.

Després de realitzar tot el procés, s'han pogut observar els avantatges de la utilització de OpenVINO a l'hora d'implementar xarxes sobre plataformes heterogènies amb gran rapidesa. No obstant això, en tractar-se d'una eina amb poc recorregut, està subjecta a millores com és la generació de bitstreams per a FPGA que suporten major quantitat de capes diferents.

Abstract

Nowadays, there is great controversy in the selection of devices for the generation of algorithms of Artificial Intelligence and more specifically those of Deep Learning, since they need large computational requirements. This paper proposes the implementation of different neural networks used in Positron Emission Tomography (PET) for the reconstruction of images on FPGA devices. These devices are endowed with great flexibility in the designs, low consumption and high capacity of integration in embedded systems.

For this, the OpenVINO platform will be used. This is a tool that integrates the entire workflow from a pre-trained network until its inference about the selected device, in this case an Intel FPGA (PAC Arria 10 GX). Its operation is based on obtaining an intermediate representation of the network with the help of the Model Optimizer tool, this representation will be processed by an inference engine that will be responsible for carrying out the process of executing the network iteratively. Finally, with the help of the Benchmark plug-in you can obtain the results of througput and latency in the inference process.

First, a first check of the workflow of the tool is performed with a convolutional network trained to classify handwritten numbers. Finally, the entire process for the execution of two networks is carried out. The first is a convolutional network capable of predicting the depth of the point of collision of the positron in a PET system. The second is a dense network that tries to predict the error made in the measurement of the Z coordinate with the collision's point of the positron with respect to an analytical method.

After carrying out the entire process, the advantages of using OpenVINO have been observed when implementing networks on heterogeneous platforms very quickly. However, being a tool with little travel, it is subject to improvements such as the generation of bitstreams for FPGA that support a greater number of different layers.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación

Hoy en día la Inteligencia Artificial, y en concreto el Deep Learning, se ha convertido en un componente fundamental de nuestras vidas. Por un lado, la gran cantidad de aplicaciones de esta materia como el diagnóstico precoz en hospitales, la detección de señales de trafico en visión por computador para automóviles o el procesado de imágenes en nuestros teléfonos móviles, y por otra parte, los grandes requerimientos computacionales que requiere su aplicación, marcan un reto al desarrollo de esta tecnología.

Existe gran controversia sobre la plataforma óptima para la ejecución de dichos algoritmos debido a diversos factores como son la velocidad de procesamiento de los datos, respuesta en tiempo real en sistemas críticos, consumo, tamaño o el precio del dispositivo.

En este contexto, las FPGAs juegan un papel fundamental por su gran aplicabilidad en sistemas de alta velocidad, su alta capacidad de integración para sistemas embebidos y bajo consumo con respecto a otras tecnologías. Por ello, y desde el punto de vista computacional, en el presente documento se estudia la actuación de diferentes algoritmos de aprendizaje profundo sobre dichos dispositivos.

Con todo ello, el grupo de investigadores J. Renner, J.M. Benlloch-Rodríguez, J.V. Carrión y R. Gadea entre otros, han desarrollado un sistema de reconstrucción de imágenes para la plataforma PETALO (Positron Emission TOF Apparatus based on Liquid xenOn) basado en la utilización de redes neuronales. Sus altos requerimientos computacionales en velocidad marcan el punto de partida de este trabajo fin de Máster.

1.2. Objetivos

En este trabajo se pretende realizar la inferencia de distintas redes neuronales empleadas en Tomografía de Emisión de Positrones (PET) desarrolladas para la plataforma PETALO sobre FPGA. Además, se comprobará los niveles de velocidad alcanzados en dicha inferencia sobre un dispositivo FPGA. Para ello, se estudiará el entorno de desarrollo OpenVINO cuya distribución realiza Intel, empleando un dispositivo específico del mismo fabricante.

Capítulo 2

Fundamentos del Deep Learning

A continuación se presentan los conceptos y principios básicos del Deep Learning, para ello se realiza un breve introducción al campo del conocimiento de referencia. Finalmente se justifica la presencia de dispositivos de altas prestaciones computacionales a la hora de utilizar este tipo de algoritmos.

2.1. Inteligencia artificial

El concepto de Inteligencia Artificial surge en 1950 de la mano del investigador Alan Turing con su artículo "Computing Machinery and Intelligence", donde se presenta el famoso Test de Turing en el cual se propone demostrar si una determinada máquina es inteligente o no en base a la capacidad de generación de respuestas a diferentes preguntas. No es hasta la década de 1990 cuando el resurgir de esta ciencia toma fuerza con grandes aportaciones de los investigadores de la época, grandes inversores apoyan esta tecnología ante la necesidad mejorar la capacidad de procesamiento y análisis de la enorme cantidad de datos con las que ya se trabajaban [1].

Hoy en día, se entiende por Inteligencia Artificial como la "Disciplina científica que se ocupa de crear programas informáticos que ejecutan operaciones comparables a las que realiza la mente humana, como el aprendizaje o el razonamiento lógico" [2].

En otras palabras, la Inteligencia Artificial estudia cómo trabaja el cerebro humano y cómo las personas tomamos decisiones cuando nos enfrentamos a un problema. Estos paradigmas son aplicados a desarrollar sistemas que desarrollen ciertas tareas.

Los problemas tradicionales de esta rama de la ciencia incluyen el razonamiento, representación del conocimiento, planificación, aprendizaje, procesado natural del lenguaje y percepción. Es evidente que la complejidad de los algoritmos es cada vez mayor debido a las mejoras continuas de los mismos y su influencia a aumentado en nuestras vidas. Sus aplicaciones están en todas partes y sus posibilidades se han visto incrementadas en los últimos años debido a las mejoras, también, en la tecnología [3].

2.2. Machine Learning

El Machine Learning es un subcampo de la Inteligencia Artificial, definido en 1959 por Arthur Samuel como "campo de estudio que da a los ordenadores la habilidad de aprender sin ser explícitamente programados ". Una vez creado el programa, éste es capaz de aprender de forma autónoma a realizar alguna tarea que requiera de cierta inteligencia. Esto contrasta con un programa que define el comportamiento del mismo de forma explícita, punto por punto.

Este punto de vista abre nuevas posibilidades y ventajas con respecto a otro tipo de algoritmos inteligentes. En lugar de crear un programa distinto y específico para resolver un problema en concreto, un único algoritmo de Machine Learning será capaz de aprender tareas diferentes a través de un proceso llamado 'Entrenamiento' (Training) [4].

Existen diversas técnicas para diseñar y especificar los parámetros de los algoritmos. Las principales técnicas para el aprendizaje son:

- Aprendizaje no supervisado: está basado en la capacidad de encontrar patrones en un flujo de entrada de datos. No se fijan etiquetas asociadas a las entradas ni objetivos al algoritmo de destino. En este tipo de aprendizaje el algoritmo por sí solo debe ser capaz de asociar las distintas entradas para crear una estructura determinada.

- Aprendizaje supervisado: crea un modelo matemático a partir de los datos de entrada asociados a unas salidas determinadas. Incluye algoritmos de clasificación y regresiones numéricas.

    - La Clasificación es utilizada cuando las salidas están restringidas a un número determinado de valores y se usa para determinar la categoría del flujo de datos de entrada. Previamente el algoritmo es entrenado con varios ejemplos de dichas categorías.
    - Regresión es el método por el cual se intenta producir una función que describa la relación entre las entradas y salidas del algoritmo con el fin de predecir cómo cambiarán las salidas a medida que varían las entradas.

- Aprendizaje semi-supervisado: combina los dos métodos anteriores, se basa en el entrenamiento de un algoritmo con un conjunto incompleto de datos de entrada y unas etiquetas o valores finales también incompletos.

- Aprendizaje por refuerzo: es un área del aprendizaje automático que mejora la efectividad de sus algoritmos en base a una recompensa acumulativa para resultados válidos o mejorados con respecto a los anteriores. De la misma forma puede emplear penalizaciones para los erróneos.

El Entrenamiento es una tarea intensiva de cómputo supervisada que trata de obtener el valor óptimo de ciertos parámetros del algoritmo en cuestión. Su ejecución puede dividirse en dos partes, en primer lugar, se parte de unos valores iniciales de los parámetros del algoritmo al cual se le aplica una señal de entrada, la salida obtenida se compara con el valor deseado u objetivo y se calcula el error obtenido. En segundo lugar, el error se relaciona con los parámetros para obtener un valor de los mismos cuya posterior ejecución, ante una señal de entrada similar, proporcione una salida de valor más próxima al objetivo [5].

2.3. Deep Learning

2.3.1. Conceptos básicos

El Deep Learning es una de las técnicas más utilizadas dentro del entorno del Machine Learning inspirada en las estructuras básicas del cerebro humano, las neuronas. Este enfoque trata de emular como entendemos el funcionamiento del cerebro, no a la creación de un cerebro. Dichos sistemas, comúnmente llamados Redes Neuronales Artificiales son capaces de realizar tareas de clasificación o predicción tras un entrenamiento previo, lo cual hace que sea un aprendizaje supervisado. En la figura 2.1 se muestra la jerarquía de cada una de las técnicas comentadas.

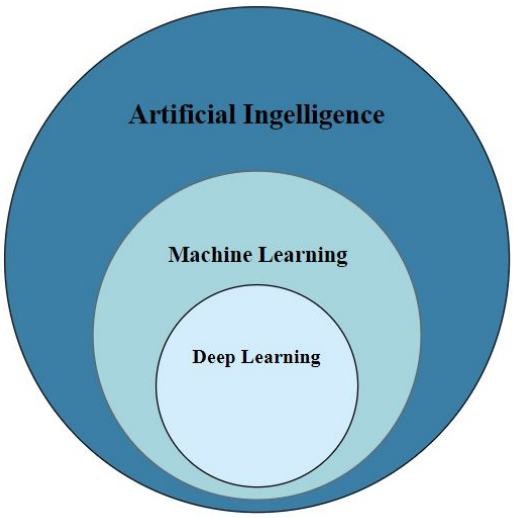

Figura 2.1: Jerarquía de áreas en Inteligencia Artificial.

En el campo de las Redes Neuronales, la unidad básica es el perceptrón o neurona artificial. Como ya se ha comentado, esta unidad debe su nombre al concepto en el que se basa su creación y funcionamiento. En la siguiente imagen se observa cada una de las partes de las que esta formado:

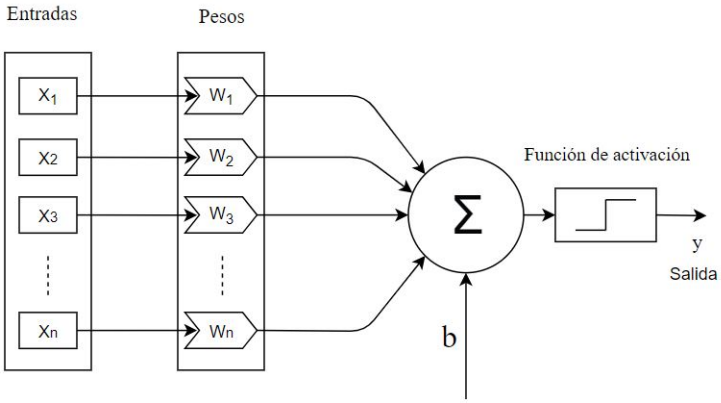

Figura 2.2: Esquema de un perceptrón simple.

De izquierda a derecha podemos asociar cada una de las partes del perceptrón con su relación directa con una neurona real:

- Las entradas X_i están asociadas a un numero determinado de dendritas de la neurona cuya función es proveer al perceptrón de nuevas señales de excitación.
- Los pesos W_i imitan la sinapsis de una neurona y su valor determinará la importancia final de la entrada por la que ha sido multiplicado.
- A continuación, se realiza el sumatorio de todos los valores obtenidos a partir de la multiplicación de las entradas por los pesos, además de un valor fijo 'b' llamado bias. Este cometido se relaciona con el soma en una neurona real.
- Finalmente para obtener la salida se emplea una función de activación que devolverá un valor acotado en un rango determinado. Estas funciones pueden ser de diferentes tipos, desde una función escalón o rampa, hasta una tangente hiperbólica.

Matemáticamente se podría resumir el funcionamiento de una neurona artificial como:

$$y = f(b + \sum X_i \cdot W_i) \tag{2.1}$$

donde y es la salida, f la función de activación, X_i las entradas y W_i los pesos sinápticos del perceptrón.

Al igual en el cerebro humano, una unidad básica no es capaz de realizar una tarea compleja, en Deep Learning es necesario la asociación de distintas neuronas básicas para la resolución de problemas linealmente separables como es el caso de una función XOR. De este hecho, nace el concepto de Red Neuronal o Perceptrón Multicapa cuyo ejemplo se muestra en la figura 2.3 [6].

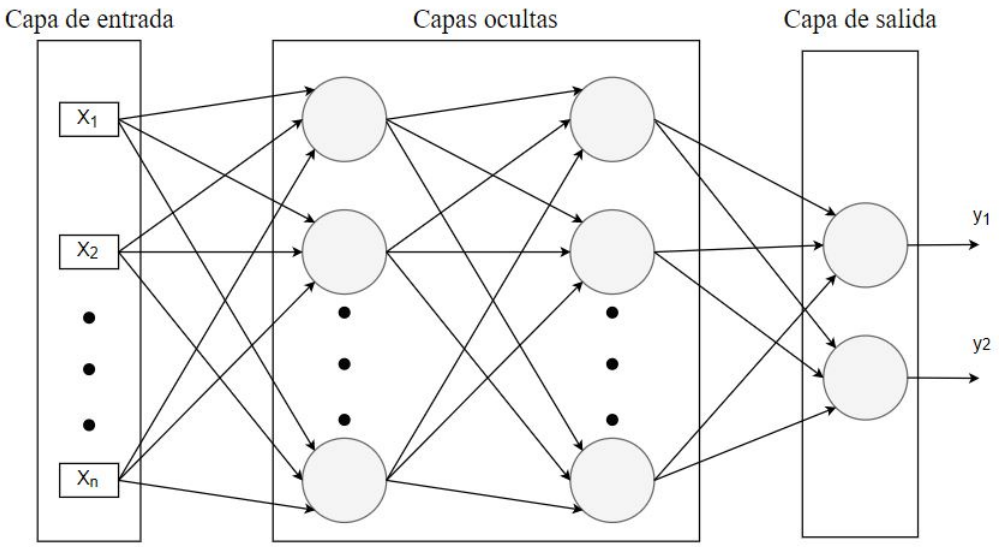

Figura 2.3: Esquema de un perceptrón multicapa con dos capas ocultas y salida doble.

El correcto funcionamiento de una Red Neuronal se basa principalmente en el método de aprendizaje del propio sistema, cuyo objetivo será el de obtener el valor óptimo de los diferentes parámetros que lo forman.

2.3.2. Algoritmo de aprendizaje

El algoritmo de aprendizaje por excelencia para estos sistemas es el llamado 'Forward and Backward propagation' o lo que es lo mismo, propagación hacia adelante o hacia atrás. Este algoritmo puede resumirse de forma simplificada a partir del esquema de la figura 2.4 [7].

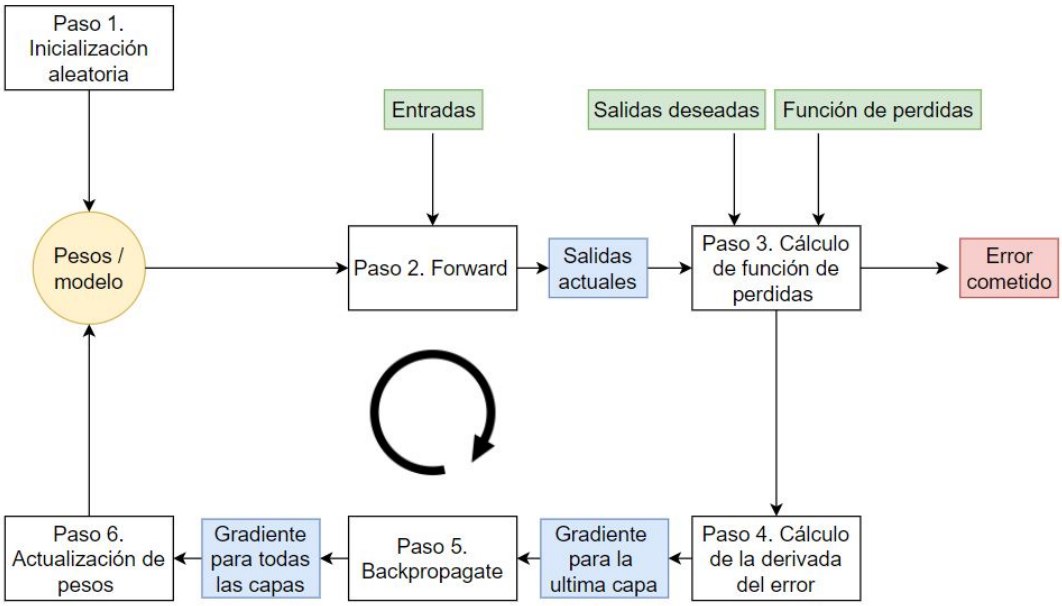

Figura 2.4: Esquema algoritmo 'Forward and Backward propagation'.

Partiendo de un conjunto de datos de entrada cuyas salidas son conocidas, ya que se trata de un aprendizaje supervisado, se emplea el algoritmo, anteriormente nombrado, para obtener el valor óptimo de los pesos de la red y cuyos pasos se muestran a continuación:

- Paso 1. Se inicializan los parámetros de la red en un valor determinado y los pesos de cada neurona a un valor aleatorio. Este paso únicamente se realiza una vez en la fase de entrenamiento. Algunos de los parámetros básicos el entrenamiento de redes neuronales son:
  - Época: se trata de una ejecución completa del algoritmo de aprendizaje.
  - Batch: es el número de ejemplos de señales de entrada que se le muestra a la red en una iteración del algoritmo. Dicho valor es directamente proporcional al tamaño de la memoria requerida para el procesado de los datos.
  - Número de iteraciones: veces que el algoritmo se repite para actualizar los pesos de la red.

- Paso 2. Partiendo del valor aleatorio de los pesos de cada neurona, se obtiene la salida o salidas para cada una de las entradas determinadas por el batch. Esta etapa se propaga de forma natural desde la entrada hacia la salida.

- Paso 3. De acuerdo con la función de perdidas empleada y los valores de las salidas deseadas se calcula el error cometido en cada salida en función de los pesos que se han empleado en la red.

- Paso 4. Existen gran cantidad de métodos de optimización para la actualización de los pesos con el fin de minimizar la función de pérdidas, una de las más utilizadas es la de descenso por gradiente, ya que computacionalmente es muy eficiente. Este método trata de alcanzar el valor mínimo de la función de perdidas a través del cálculo de su derivada.

- Paso 5. El método de optimización se propaga desde la salida hacia la entrada y se calcula su gradiente de la función de perdidas en cada capa. Este hecho da el nombre de Backpropagation al algoritmo de aprendizaje.

- Paso 6. Finalmente se obtienen los pesos de cada neurona y se actualizan a su nuevo valor. Todo el proceso anterior es iterativo hasta conseguir un valor de pérdidas mínimo [7].

A continuación se muestra un ejemplo de cómo se realizaría una iteración del algoritmo anterior con el fin de justificar los requerimientos computacionales que supone su implementación:

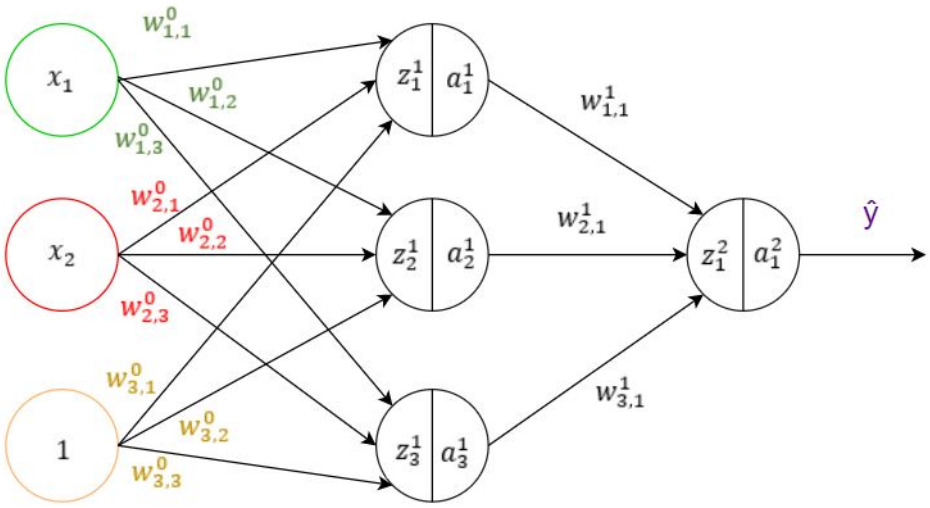

Figura 2.5: Red neuronal con salida simple y una capa oculta.

De acuerdo con la figura 2.5 la variables X_i son las entradas, el valor 1 es un valor constante en la red que sirve de sesgo en el cálculo estadístico total, las variables Z_i son las funciones sumatorio de los resultados de multiplicar los pesos W_i por las entradas. Las funciones de activación de cada neurona son a_i y finalmente la variable $\hat{y}$ hace referencia a la salida obtenida.

Si se expresan las matrices de pesos para la capa cero o de entrada y para la capa oculta:

$$W^0 = \begin{bmatrix} w^0_{1,1} & w^0_{1,2} & w^0_{1,3} \\ w^0_{2,1} & w^0_{2,2} & w^0_{2,3} \\ w^0_{3,1} & w^0_{3,2} & w^0_{3,3} \end{bmatrix}; W^1 = \begin{bmatrix} w^1_{1,1} \\ w^1_{2,1} \\ w^1_{3,1} \end{bmatrix} \tag{2.2}$$

Podría calcularse la salida de la red como el producto escalar de la función de activación de cada neurona de la capa anterior con la matriz traspuesta de los pesos de la capa anterior. Este cálculo corresponde con el paso Forward del algoritmo de aprendizaje.

$$A^2 = f_{activacion}((W^1)^T * A^1) \tag{2.3}$$

Siendo A^1:

$$A^1 = f_{activacion}((W^0)^T * A^0) \tag{2.4}$$

y representándose las matrices de las funciones de activación como:

$$A^0 = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}; A^1 = \begin{bmatrix} a^1_1 \\ a^1_2 \\ a^1_3 \end{bmatrix}; A^2 = \begin{bmatrix} a^2_1 \end{bmatrix} \tag{2.5}$$

Una vez obtenida la salida, se calcula el error a partir del valor de la salida deseada. En este caso el error absoluto:

$$Error = E = a^2_1 - \hat{y} \tag{2.6}$$

Además, se comienza el paso Backpropagation para actualizar los pesos minimizando el error anterior con el método de descenso por gradiente. Las matrices de gradiente para cada capa pueden representarse matricialmente como:

$$\Delta^2 = \begin{bmatrix} \delta_{a^2_1} \end{bmatrix} = E * (A^2)' \tag{2.7}$$

$$\Delta^1 = \begin{bmatrix} \delta_{a^1_1} \\ \delta_{a^1_2} \\ \delta_{a^1_3} \end{bmatrix} = \Delta^2 * (W^1)^T \tag{2.8}$$

Finalmente la actualización de los pesos para la iteración actual puede obtenerse como:

$$W^1(t+1) = W^1(t) - \eta \Delta^2 * A^1 \tag{2.9}$$

$$W^0(t+1) = W^0(t) - \eta (A^0)^T * \Delta^1 \tag{2.10}$$

para la tasa de aprendizaje η con valor entre 0 y 1 [7].

2.3.3. Tipos de Redes Neuronales y aplicaciones

Existen gran cantidad de tipos de Redes Neuronales y su clasificación puede realizarse desde el método de aprendizaje hasta su topología o estructura. En esta sección se describen algunas de las redes más utilizadas en la actualidad clasificadas en función de su estructura.

Perceptrón multicapa

Este tipo de redes son similares a las de la figura 2.3, a diferencia de un perceptrón simple, ésta dispone de un conjunto de capas intermedias además de las de entrada y salida a la red. Dependiendo del número de conexiones entre las distintas neuronas de una capa con las de la siguiente puede estar total o parcialmente conectada.

Sus aplicaciones más comunes son las de resolver problemas de asociación de patrones, segmentación de imágenes o compresión de datos entre otras.

Redes Neuronales Convolucionales (CNN)

A diferencia del resto de redes neuronales, en lugar de tener pesos independientes por cada neurona, los pesos se utilizan para definir un filtro convolucional que se aplica a la información de entrada para a continuación, aplicar la función de activación al resultado de la convolución. Su principal aplicación es la de procesado de imagen o vídeo y su estructura combina las convoluciones con capas de reducción. Hoy en día son muy utilizadas debido a sus buenos resultados aunque sus requerimientos computacionales son mayores que con otro tipo de redes.

Las principales capas de las que están compuestas estas redes son:

- Capa convolucional: capa de filtrado para la extracción de características de la imagen. Está formada por un número definido de filtros a los cuales se define su tamaño, estos se desplazarán por toda la imagen. A su salida se generan matrices de datos que contienen las características extraídas, su tamaño dependerá del tamaño del filtro y por lo tanto también de la cantidad de pesos que se incluyan, también dependerá del tipo de desplazamiento del filtro y la posibilidad de incluir un padding externo.
- Capa de reducción: también llamada 'pooling', se encarga de reducir el tamaño de las matrices anteriormente obtenidas para disminuir el número de datos a procesar. La reducción dependerá del tamaño de ventana seleccionado. Se pueden emplear diversas técnicas como la media o máximo del conjunto de valores de la ventana.
- Capa de entrada totalmente conectada: también llamada 'flatten', su función será transformar las matrices de datos en un único vector que contiene toda la información característica de la entrada.
- Primera capa totalmente conectada: proporciona un nivel mayor de profundidad a la red de predicción de características. Es opcional.
- Capa de salida totalmente conectada: muestra los niveles de probabilidad para cada clase de salida.

Redes Neuronales Recurrentes (RNN)

Una RNN no tiene una estructura de capas definida, las neuronas tienen conexiones arbitrarias. De hecho, su enlace puede generar ciclos con el fin de crear temporalidad, permitiendo que las neuronas tengan memoria. Su aplicación principal es el análisis de secuencias en textos, sonido o vídeo [8].

Redes Autoencoder

Estas estructuras tienen el objetivo de generar nuevos datos en dos pasos. Primero se comprime la entrada en un espacio de variables determinado y luego se reconstruye la salida en función de la información anterior. Las dos partes de las que se forman estas redes son:

- Encoder: zona de inicio de la red donde se reducen los valores de entrada a un tamaño determinado.
- Decoder: parte final de la red donde se reconstruyen los valores con unas características determinadas en función de los datos anteriores.

Estas dos funciones pueden emplearse de forma separada como medio de compresión y descompresión. Las aplicaciones de este tipo de redes pueden ser la de reducción de ruido en imagen, reducción o amplificación de la dimensionalidad para la visualización o la adaptación a otro medio. Además de, compresión y descompresión de vídeo e imagen [9].

2.3.4. Dispositivos para la ejecución de algoritmos de Deep Learning

Como se ha podido observar, tanto en el entrenamiento como en la inferencia de redes neuronales se necesitan grandes requerimientos computacionales. Además, la mayoría de aplicaciones a las que se destinan estos algoritmos procesan gran cantidad de datos con tiempos de respuesta muy acotados. Por todo ello, existe la necesidad de utilizar dispositivos capaces de atender a todas estas exigencias. Aunque algunos fabricantes disponen de dispositivos integrados específicos para la ejecución de este tipo de algoritmos, a continuación se comparan únicamente los de propósito general:

- CPU: su principal ventaja es su facilidad de programación con gran cantidad de recursos software para el desarrollo de aplicaciones como C/C++, Python, Java o cualquier otro lenguaje. Con estos dispositivos es muy rápido el desarrollo de redes neuronales de pequeña escala, con lotes de datos reducidos y con modelos cuyo tiempo de entrenamiento y ejecución no sean determinantes en la aplicación. Además, estos dispositivos se emplean como host de GPUs y FPGAs.
- GPU: son unidades de procesamiento especializadas en para el procesado de vídeo e imagen. Están formadas por gran cantidad de unidades de procesamiento más simples que en una CPU. Son ideales para el procesado de datos en paralelo, como es el caso de píxeles. Por otra parte, su consumo es mucho mayor que el de otros dispositivos y su programación está limitada a ciertos lenguajes como CUDA u OpenCL.

- FPGA: son dispositivos cuyo hardware reconfigurable permite tener una gran flexibilidad en el desarrollo de aplicaciones de Deep Learning. Dispone de unidades de procesamiento muy potentes como DSPs y su memoria distribuida permite el acceso inmediato a los datos. Además, su alta capacidad de integración y su bajo consumo posibilitan su aplicabilidad en sistemas embebidos. En un inicio su programación únicamente podía realizarse con lenguajes de descripción de hardware como VHDL o Verilog, sin embargo, hoy en día existen lenguajes de más alto nivel como OpenCL o HLS que permiten obtener aplicaciones con tiempos de desarrollo mas cortos. Aunque actualmente se necesita de un host externo para su configuración y control en estas aplicaciones, su futuro es prometedor ya que tienen la capacidad de integrar microprocesadores de altas prestaciones en su hardware [10] [11].

Capítulo 3

Herramientas y Metodología

En este capítulo se describen las herramientas software y hardware utilizadas para la consecución de los objetivos marcados por el trabajo. Además, se presenta el flujo de trabajo llevado a cabo para el mismo fin.

3.1. Herramientas

3.1.1. Software

Para el desarrollo de este trabajo se ha utilizado la siguiente distribución de Linux:

- CentOS Linux release 7.6.1810

Sobre ésta, se han instalado una serie de paquetes que se enumeran y describen a continuación:

3.1.1.1. Anaconda

Anaconda es una distribución de los lenguajes Python y R, libre y abierta, empleada para ciencias de datos y Machine Learning. En este trabajo se emplean distintas librerías para el procesado de las señales de entrada a las redes empleadas, así como la generación de las mismas. Además, junto a este paquete se emplea Jupyter Notebook, una aplicación web de código abierto que permite crear y compartir documentos que contienen código en vivo, ecuaciones, visualizaciones y texto narrativo [12].

3.1.1.2. Keras

Keras es una librería para el desarrollo de redes neuronales de alto nivel, escrita en Python y capaz de ejecutarse sobre diferentes librerías de bajo nivel como Tensorflow, entre otras. Permite la creación de algoritmos de Deep Learning en tiempos de desarrollo relativamente cortos debido a su modularidad y pueden emplearse distintos dispositivos para la aceleración de la fase de entrenamiento o ejecución [13].

3.1.1.3. OpenVINO

Del inglés “ Open Visual Inferencing and Neural Network Optimization” es una herramienta que permite desarrollar soluciones orientadas a la visión sobre distintas plataformas. Se centra en aplicar Deep Learning, en concreto para Redes Neuronales Convolucionales (CNN), a sistemas de procesado de imagen y vídeo acelerado con distinto hardware.

Permite la optimización de ciertos procesos incluyendo librerias de OpenCV, OpenCL u OpenVX. Además, dota de cierta abstracción al proceso de inferencia gracias a los kernels pre-optimizados para gran cantidad de aplicaciones, eso supone menor tiempo de desarrollo con la finalidad de ser más competitivo. El presente documento centrará su atención en la distribución que hace Intel sobre esta herramienta [14].

Sus principales campos de aplicación son:

- Atención sanitaria: procesado de imágenes médicas para diagnostico.
- Vigilancia: Sistemas de seguridad inteligentes.
- Venta al por menor: clasificación de objetos y detección de los mismos.
- Industria: Robótica inteligente para la industria 4.0.
- Ciudades inteligentes: control de tráfico, multitudes...
- Transporte: detección de señales, visión artificial entre vehículos…

3.1.2. Hardware

3.1.2.1. Host

Las características principales del host son:

Dispositivo	*Características*
CPU	Intel® Xeon(R) Gold 5118 CPU @ 2.30GHz × 48
RAM	64 GB
Cache	Tamaño: 16896 KB

Tabla 3.1: Características del host

3.1.2.2. FPGA

El modelo de tarjeta aceleradora del que se dispone es la ’Intel® Programmable Acceleration Card con Intel Arria® 10 GX FPGA’ mostrada en la figura 3.1. Esta tarjeta aceleradora FPGA basada en PCI Express * (PCIe *) proporciona el rendimiento y la versatilidad de la aceleración FPGA y es totalmente compatible con la pila de aceleración para CPU Intel® Xeon® con FPGA. Se puede implementar en muchos segmentos del mercado, como análisis de grandes cantidades

de datos, inteligencia artificial, transcodificación de vídeo, ciberseguridad y comercio financiero [15].

Algunas de las características más importantes de este dispositivo son:

Dispositivo	*Características*
FPGA	Intel® Arria® 10 GX
Logic Elements	1150000
On chip Memory	65.7 Mb
DSP Blocks	3036
External On board DDR4 Memory	8GB (4GB x 2 banks)
Potencia de diseño térmico (TDP)	66 W

Tabla 3.2: Características principales tarjeta Intel PAC Arria 10 GX [16].

Figura 3.1: Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA.

3.2. Metodología

A continuación se muestran los diferentes pasos llevados a cabo para la consecución de los objetivos marcados en el trabajo:

1. Estudio del flujo de trabajo de OpenVINO.
2. Verificación del entorno FPGA.
3. Demo para clasificación.
4. CNN para clasificación de números escritos a mano.
5. Plataforma PETALO.
6. CNN para predicción de profundidad de positrones.
7. Red densa para predicción del error con respecto al método analítico.

Capítulo 4

Desarrollo

4.1. Flujo de trabajo en OpenVINO

La implementación de redes de neuronales desde el entorno de diseño hasta plataformas integradas para inferencia puede ser una tarea compleja que presenta una serie de desafíos técnicos que deben abordarse:

- Existen varios marcos de aprendizaje profundo ampliamente utilizados en la industria, como Caffe, TensorFlow, MXNet, Kaldi, etc.
- Normalmente, la creación y entrenamiento de redes se realiza sobre centros de datos, mientras que la ejecución se puede llevar a cabo en plataformas integradas, optimizadas para el rendimiento y consumo de energía dependiendo de la aplicación de destino. Las plataformas de entrenamiento suelen estar limitadas tanto desde la perspectiva del software, lenguajes de programación o consumos de memoria, como la del hardware, diferentes tipos de datos o potencia limitada. Por todo ello, no se recomienda emplear el marco de aprendizaje para realizar la inercia. La solución pasa por utilizar entornos dedicados y optimizados para cada tipo de hardware.
- Además, a todo ello se le une la complejidad que están alcanzando las estructuras de redes neuronales para el procesado de los datos. Garantizar su precisión no es una tarea trivial.

En este contexto surge la herramienta OpenVINO cuyo flujo de trabajo se presenta en la figura 4.1. El objetivo principal de este entorno será realizar la inferencia o ejecución de un modelo de red neuronal pre-entrenado sobre diferentes dispositivos. Para ello, se emplean diversas herramientas y técnicas. A continuación, se desarrollan cada una de sus fases [17].

Al tratarse de una plataforma novedosa y en continuo desarrollo, es preciso comentar que se ha trabajado con la versión R3 de la misma.

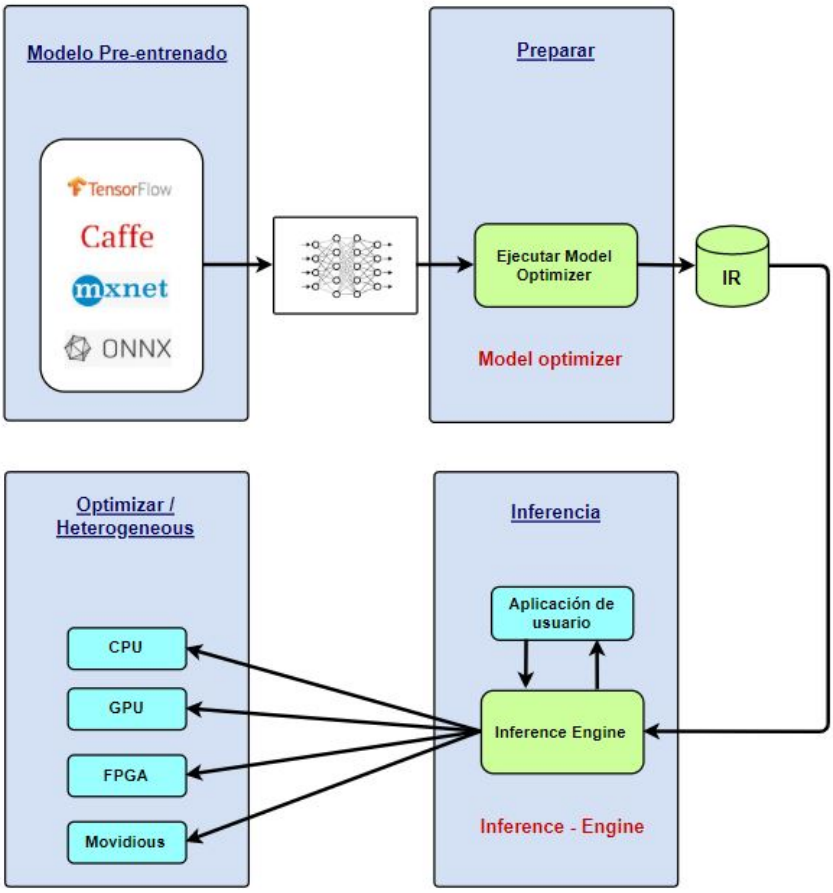

Figura 4.1: Esquema simplificado del flujo de trabajo con OpenVINO.

4.1.1. Modelo pre-entrenado

Se parte de una red neuronal entrenada con una estructura y parámetros definidos. En OpenVINO se aceptan como entrada al flujo de trabajo algunas de las infraestructuras más importantes en el desarrollo de redes. La extensión de archivos soportada, donde deben definirse la estructura, valores de los pesos y el resto de parámetros, son:

- .caffemodel - modelos Caffe.
- .pb - modelos TensorFlow.
- .params - modelos MXNet.
- .onnx - modelos ONNX.
- .nnet - modelos Kaldi.

4.1.2. Preparar

Antes de poder utilizar la red diseñada es necesario preparar y ajustar el modelo a la plataforma donde va a ser ejecutado. Para ello, se emplea una herramienta offline llamada 'Model Optimizer' que transforma la red inicial en un modelo compatible con la herramienta de inferencia.

Su modo de trabajo es el siguiente; carga el modelo en la memoria, lo lee y crea una representación intermedia del mismo en el formato necesario para la plataforma de ejecución utilizada. Los dos propósitos principales de dicha herramienta son:

- Producir una representación intermedia (IR): si este objetivo no se cumple no se podrá ejecutar la red. La responsabilidad principal de la herramienta será obtener dos archivos (.xml y .bin) que formarán la IR. El primero describe la topología de la red y el segundo contiene el valor de los parámetros, como el formato de las entradas, pesos...

- Optimizar la IR: esto se consigue sustituyendo determinados patrones que definen la estructura de una red por operaciones matemáticas o eliminando ciertas capas que no van a ser utilizadas y puede omitirse. El resultado es una IR que tiene menos capas que el modelo original y por lo tanto disminuirá el tiempo de ejecución de la red.

Para que la herramienta pueda generar una IR lo más simplificada posible es necesario que la red con la que se trabaja esté generada con una topología de capas que sea reconocible por la misma. En caso contrario, no se podrá obtener dicha representación y la herramienta de inferencia no podrá ejecutar la red en ningún dispositivo. Estas topologías son las que en su amplia mayoría utilizan los usuarios y están listadas en la documentación de OpenVINO [18].

Los parámetros de entrada más importantes que acepta esta herramienta son:

$--inputmodel$: localización del archivo del modelo de red pre-entrenado. En el caso de tensorflow, archivo .pb.

$--modelname$: nombre del modelo de representación intermedia (IR) que será creado.

$--outputdir$: directorio de salida para los archivos .xml y .bin. Por defecto es el directorio donde se ejecuta la herramienta.

$--inputshape$: formato de los datos de entradas. Para tensorflow se especifica como [N,H,W,C] donde N es el número de batches, H y W el tamaño en pixeles de la imagen y C el número de canales para definirla.

$--datatype$: se especifica entre FP32 o FP16 la cuantificación que se quiere emplear de los pesos y biases para crear la representacion intermedia.

Un ejemplo de lanzamiento de esta herramienta es el que se muestra a continuación:

mo_tf.py - - input_model /home/t_model0.pb - - input_shape [1,28,28,1] - - data_type FP16 - - model_name tf_model0_fp16

4.1.3. Inferencia

Después de haber utilizado el Optimizador de Modelos, se emplea la herramienta Inference Engine o Motor de inferencia para incluir datos de entrada y realizar la ejecución de la aplicación. OpenVINO dota de una librería de funciones escritas en C++ y Python para realizar todo el proceso. El flujograma de programación del Motor de Inferencia es el que se muestra en la figura 4.2 y cada una de sus etapas tienen la siguiente función:

1. Inicializar el Motor de Inferencia: se crea el núcleo del proceso para administrar los dispositivos disponibles y sus complementos de forma interna.
2. Leer IR: se leen los archivos .xml y .bin creados anteriormente. Contienen todos los datos necesarios para generar la red.
3. Configurar entradas y salidas: se toma la información de la topología de la entra y la salida. Opcionalmente, se puede configurar el formato (precisión) y el tamaño de memoria empleado para ellas. Al tratarse de de imágenes el motor de inferencia supone un formato de color BGR, sin embargo puede admitir las siguientes conversiones:
   - RGB –>BGR
   - RGBX –>BGR
   - BGRX –>BGR
   - NV12 –>BGR

   Donde X es un canal que será ignorado.
4. Cargar modelo: se añade al core del proceso el modelo de red anteriormente leído.
5. Solicitud de inferencia. Se crea el punto de partida para comenzar el proceso de inferencia.
6. Preparar las entradas. Se ajustan las imágenes al tamaño de las entradas. Existen varias funciones para implementaciones de una sola red o redes en cascada.
7. Inferencia. Se ejecuta la red con una entrada determinada. Puede especificarse si el método es síncrono o asíncrono.
8. Obtención de salidas: en cada ciclo de inferencia se obtiene un valor o valores de salida para cada entrada. Estos son almacenados en un buffer.

El Motor de Inferencia utiliza una arquitectura de plugin. El plugin Inference Engine es un componente de software que contiene una implementación completa para la inferencia en un determinado dispositivo de hardware, en este caso de Intel®: CPU, GPU, VPU, FPGA, etc. Cada plugin implementa la API unificada y proporciona recursos adicionales específicos de cada hardware.

Como es evidente, la inferencia puede realizarse sobre diferentes dispositivos, por defecto al lanzar esta herramienta se ejecutará sobre CPU, para especificar otra plataforma se incluye un comando de entrada en la inicialización del proceso. Dicho comando se especifica como '$--device$' o '$-d$' seguido del dispositivo deseado. El objetivo seleccionado debe se compatible con la

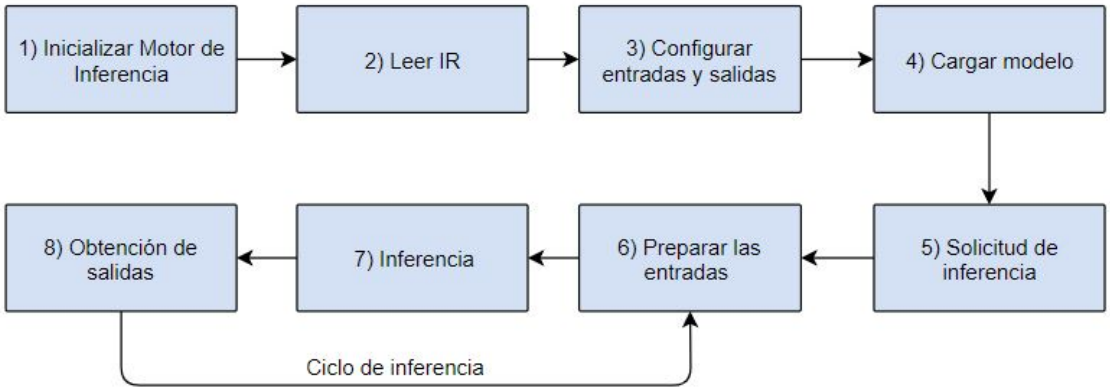

Figura 4.2: Flujograma del Motor de Inferencia.

plataforma y previamente configurado. Además, todas las estructuras de red deben estar soportadas por el mismo, de otra forma no podrá realizarse la inferencia [19].

Actualmente existen gran cantidad de estructuras definidas en redes neuronales, algunas de ellas no están soportas sobre dispositivos como GPU o FPGA. Para solventar este problema, OpenVINO proporciona un plugin llamado Heterogeneous.

Con todo ello, los argumentos más importantes que acepta esta herramienta son:

$-m$: localización del archivo del modelo de representación intermedia .xml.

$-i$: localización de la imagen o archivos de entrada a la red.

$-d$: especifica el dispositivo sobre el que se realiza la medición. CPU, GPU, FPGA, HETERO...

$--labels$: en caso de una red de clasificación se añade la ruta del archivo que contiene las clases de la red.

Un ejemplo del lanzamiento del proceso de inferencia:

python3 archivo_inferenecia.py - m /home/IR_model.xml - i /home/imagen.png - d CPU

En este caso, se lanza desde un archivo que contiene el pre-procesado de las imágenes o datos de entrada y el proceso de inferencia visto en esta sección y descrito con la librería de Python proporcionada por OpenVINO. El proceso se realiza sobre la CPU.

4.1.3.1. Heterogeneous Plugin

Este complemento permite la computación para la inferencia de una red en varios dispositivos. El propósito de ejecutar redes en modo heterogéneo es utilizar la potencia de aceleradores para estructuras con alta carga computacional y emplear la CPU para capas que no están soportadas. Además de emplear todo el hardware disponible durante la inferencia para ser más eficientes [20].

La utilización de esta herramienta auxiliar puede dividir su lanzamiento en dos pasos independientes:

- Vincular las capas soportadas a cada dispositivo, priorizando sobre el dispositivo determinado previamente.
- Ejecutar cada capa de la red en la inferencia de forma automática.

La forma de lanzar una ejecución heterogénea es a través del siguiente comando en la definición de los parámetros del motor de inferencia y puede realizarse para el caso de ejecución priorizada sobre FPGA como:

$$--device\ HETERO:FPGA,CPU \text{ ó } -d\ HETERO:FPGA,CPU$$

Para añadir más dispositivos, únicamente es necesario añadir las siglas del mismo con una coma previa [21].

4.1.3.2. FPGA Bitstream

El funcionamiento de las FPGAs se basan en la configuración del hardware que se realiza de las mismas. Dicha configuración, en ultima estancia, depende del bitstream que se carga sobre la misma y el cual define las interconexiones y valores de los recursos que se requieren. Por ello, antes de realizar la inferencia de una red sobre sobre una FPGA es necesario la carga de un bitstream sobre ella. OpenVINO proporciona gran cantidad de bitstreams pre-compilados para la gran mayoría de redes neuronales más comunes en la industria.

El diseño de las estructuras de redes neuronales sobre FPGA están creados de forma modular para que cualquier red con estructuras compatibles pueda ser ejecutada sobre el mismo bitstream. Esta modularidad puede observarse en la figura 4.3. Las capas soportadas se ejecutan en el orden determinado por la configuración del motor de inferencia y cuyos valores están almacenados en la memoria de lectura y escritura. La comunicación de las estructuras con la memoria se realiza a través de una matriz de interconexión llamada 'Crossbar'. Las capas de convolución, que son las que computacionalmente son más complejas, se comunican directamente con la memoria.

Con todo ello, se puede concluir que el bitstream óptimo para una determinada aplicación será el que en mayor medida comparta las mismas estructuras de red. Por ello será necesario estudiar las redes cuyos bitstreams están pre-compilados y seleccionar el que mayor número de capas tengan en común. Además, estos deben ser cargados sobre la FPGA previo a realizar la inferencia.

Para programar la FPGA es necesario emplear un compilador compatible con el dispositivo. En este caso, OpenVINO proporciona el compilador AOCL de Intel. Un ejemplo de carga de bitstream sobre el dispositivo sería:

$$aocl\ program\ device - name\ bitsteram.aocx$$

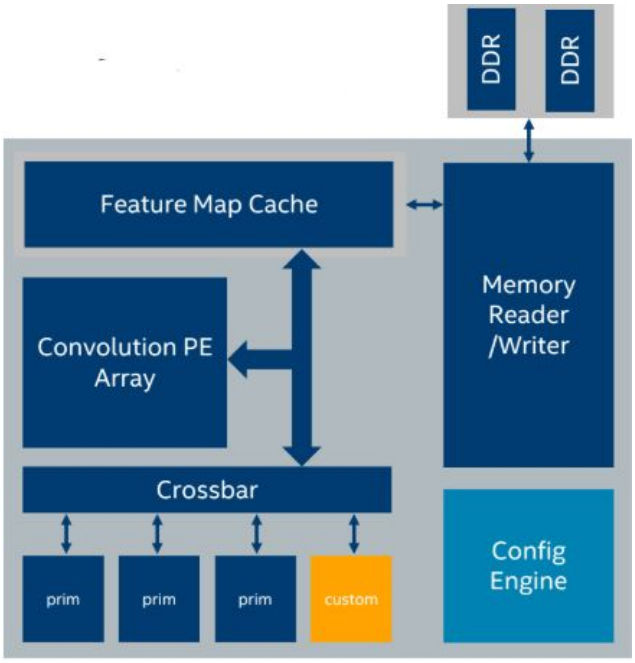

Figura 4.3: Estructura bitstream en FPGA [22].

4.1.4. Optimizaciones

OpenVINO propone la posibilidad de optimizar ciertas estructuras que o bien no están soportadas para un dispositivo o su actuación esta sujeta a mejoras. Estas optimizaciones pueden realizarse tanto para GPU como para GPU. Su programación debe realizarse en OpenCl y para ello el entorno incluye librerías y controladores específicos.

En este trabajo no se incluyen optimizaciones para el proceso de inferencia. Se trabaja con las estructuras soportadas por defecto y se miden las velocidades de trabajo alcanzadas con ello.

4.1.5. Mediciones de velocidad y latencia

Para obtener un resultado objetivo de las prestaciones en velocidad y latencia del proceso de inferencia en diferentes dispositivos y poder comparar los resultados entre si, el entorno proviene de una herramienta especialmente creada para ello. Del inglés 'Benchmark Application' o 'punto de referencia' es el instrumento que se empleará para realizar la evaluación comparativa ya comentada.

La actuación del proceso de inferencia puede ser medido en dos modos de inferencia distintos, síncrono o asíncrono. Al iniciarse, la aplicación lee los parámetros de la línea de comandos y carga una red e imágenes de entrada en el Motor de Inferencia. El número de solicitudes inferidas puede definirse con un parámetro específico en la línea de comandos al lanzar la aplicación.

Modo síncrono

En este caso la métrica fundamental es la latencia. La aplicación crea una petición de inferencia

y ejecuta el proceso. El número de ejecuciones es definido por uno de estos dos valores:

- Número de ejecuciones definido con el comando $-niter$ en los argumentos de lanzamiento de la aplicación.
- Duración predefinida si se omite el comando anterior. El valor de duración predefinido dependerá del dispositivo.

Además, durante la ejecución esta herramienta recogerá dos tipos de medidas:

- Latencia para cada petición de inferencia. Este proceso se especifica con el método $Infer$ en el transcurso de la inferencia.
- Duración de todas las ejecuciones.

El valor de latencia finál es calculado como el valor medio de todas las latencias recopiladas. El valor de rendimiento es un derivado de la latencia y además dependerá del tamaño de los paquetes de datos (batches).

Modo asíncrono

En este caso la medida principal son los frames por segundos que la red es capaz de procesar en (FPS), también llamado rendimiento de la red. De igual manera que el método anterior, se especifican el número de iteraciones a realizar.

La petición de inferencia se ejecuta de forma asíncrona. El método $Wait$ es usado para esperar a que la ejecución previa se complete. La aplicación mide todas las ejecuciones y devuelve la medida de 'throughput' basada en el tamaño de los datos (batches) y la duración total de la ejecución.

Algunos de los argumentos de entrada más importantes que acepta este complemento de OpenVINO son:

- i: ruta requerida con la imagen o imágenes de entrada. Si no se especifica se inicializa con valores aleatorios.
- m: ruta del modelo de representación intermedia de la red. Archivo .xml.
- api: especifica el modo de medida de la aplicación, $sync$ para modo síncrono y $async$ para asíncrono.
- $niter$: número de iteraciones. Sin no se especifica, se calcula en función del dispositivo.
- d: especifica el dispositivo sobre el que se realiza la medición. CPU, GPU, FPGA, HETERO...
- b: tamaño del batch. Si no se especifica, es tomado de la representación intermedia de la red [23].

Un ejemplo de lanzamiento de esta herramienta es:

$./benchmarkApp -i\ /imagen.bmp -m\ /red.xml -d\ HETERO:FPGA,CPU -api\ sync$

Se lanza la aplicación con la imagen de entrada y modelo de red por defecto en el mismo directorio que la aplicación. Se realizará una medición síncrona, para la medición específica de latencia, de forma heterogénea priorizando sobre las capas soportadas en FPGA y el resto en CPU.

4.2. Verificación del entorno FPGA

Antes de realizar la inferencia de una red sobre la FPGA es necesario comprobar si el dispositivo está instalado correctamente, para ello se ejecuta el siguiente comando:

```
(base) [jorga20j@vm1next ~]$ aocl diagnose
-------------------------------------------------------------------
Device Name:
acl0

Package Pat:
/home/jorga20j/tools/intelrtestack/a10_gx_pac_ias_1_2_pv/opencl/opencl_bsp

Vendor: Intel Corp

Physical Dev Name   Status           Information

pac_a10_ec00000     Passed           PAC Arria 10 Platform (pac_a10_ec00000)
                                     PCIe 59:00.0
                                     FPGA temperature = 50 degrees C.

DIAGNOSTIC_PASSED
-------------------------------------------------------------------
```

Figura 4.4: Diagnóstico de los dispositivos compatibles.

Se trata de un diagnostico que realiza el compilador de OpenCL (en su versión RTE) que OpenVINO proporciona en su kit de herramientas. Cuando se ejecuta, se realiza un diagnostico a los dispositivos compatibles con el mismo. Como se puede observar, el dispositivo FPGA, con nombre 'acl0', está preparado para cualquier tarea que se le requiera.

4.3. Demo para clasificación

Con el fin de comprobar el correcto funcionamiento de la herramienta y comparar los resultados obtenidos entre realizar una inferencia con CPU o con FPGA, se ejecuta una demo a través de la plataforma. Se realizará con una red neuronal de clasificación llamada SqueezeNet en su versión 1.1. Esta red es capaz de clasificar el objeto principal de una imagen entre 1000 clases pre-definidas.

La demo tiene programada en un script de Linux todos los comandos necesarios para obtener la inferencia de la red sobre varios dispositivos partiendo de un modelo pre-entrenado de la red sobre la plataforma de desarrollo de redes neuronales Caffe.

Se realizan varias ejecuciones de esta demo variando los parámetros más importantes, como son las iteraciones y el dispositivo. Como enterada a la red se propone la figura 4.5.

Figura 4.5: Imagen de entrada a demo de clasificación de imágenes (SqueezeNet).

4.3.1. Demo para clasificación sobre CPU

Se realizan tres ejecuciones de dicha demo para diferente número de iteraciones, una, cien y mil. Los resultados para cada ejecución pueden observarse en las tablas que aparecen a continuación.

Parámetro	*Valor*
Dispositivo	CPU
Iteraciones	1
Formato	FP32
Tiempo total de inferencia	3.73 ms
Tiempo de ejecución medio para una iteración	3.73 ms
Throughput	268.055 FPS
Clase	Coche deportivo
Probabilidad	0.8363

Tabla 4.1: Resultados para demo de clasificación. Una iteración en CPU.

Parámetro	*Valor*
Dispositivo	CPU
Iteraciones	100
Formato	FP32
Tiempo total de inferencia	264.82 ms
Tiempo de ejecución medio para una iteración	2.6481 ms
Throughput	377.6201 FPS
Clase	Coche deportivo
Probabilidad	0.8363

Tabla 4.2: Resultados para demo de clasificación. Cien iteraciones en CPU.

Parámetro	*Valor*
Dispositivo	CPU
Iteraciones	1000
Formato	FP32
Tiempo total de inferencia	2517.34 ms
Tiempo de ejecución medio para una iteración	2.517 ms
Throughput	397.245 FPS
Clase	Coche deportivo
Probabilidad	0.8363

Tabla 4.3: Resultados para demo de clasificación. Mil iteraciones en CPU.

Como se puede observar, la probabilidad es constante para todas las ejecuciones, obteniendo como clase principal un coche deportivo. El tiempo total de inferencia aumenta de forma proporcional con el número de iteraciones. Por otra parte, al aumentar el número de iteraciones aumenta la velocidad media de procesado de las imágenes de entrada. Este efecto se debe a que la herramienta realiza la medición de dicha velocidad, teniendo en cuenta la inicialización del dispositivo donde se realiza la inferencia [24].

4.3.2. Demo para clasificación sobre FPGA

Antes de poder realizar la inferencia sobre el dispositivo FPGA y una vez verificado el entorno, es necesario programarla con un bitstream compatible con la red de la que se pretende realizar la inferencia. En este caso OpenVINO proporciona el bitstream:

$$2019R1\_RC\_FP16\_ResNet\_SqueezeNet\_VGG.aocx$$

Que contiene soporte para la mayoría de estructuras de la red Squeezenet 1.1. y que debe ser cargado como:

$$aocl\ program\ acl0\ \ 2019R1\_RC\_FP16\_ResNet\_SqueezeNet\_VGG.aocx$$

Una vez realizado, se modifica la demo para que el dispositivo de inferencia sea la FPGA. Únicamente es necesario modificar el parámetro $target\ =\ "CPU"$ por $target\ =\ "FPGA"$.

Los resultados de la ejecución para una iteración se observan en la siguiente imagen:

```
Layer (Name: prob, Type: SoftMax) is not supported:
        custom or unknown.


[ ERROR ] Graph is not supported on FPGA plugin due to existance of layer (Name: prob, Type: SoftMax)
in topology. Most likely you need to use heterogeneous plugin instead of FPGA plugin directly.
```

Figura 4.6: Ejecución demo 1 sobre FPGA.

El terminal de Linux muestra como una de las estructuras de la red, en concreto la función de activación Softmax, no esta soportada en el bitstream de la FPGA y por lo tanto debe emplearse la herramienta Heterogeneus para poder realizar la inferencia compartida entre CPU y FPGA.

4.3.3. Demo para clasificación sobre Heterogeneous

Para ejecutar de forma homogénea la demo, se modifica el parámetro $target = "FPGA"$ por $target = "HETERO:FPGA,CPU"$, de esta forma se prioriza la inferencia sobre la FPGA dejando la CPU como soporte para ejecutar las capas no admitidas por el bitstream.

Se realizan tres ejecuciones de dicha demo para diferente número de iteraciones, una, cien y mil. Los resultados para cada ejecución pueden observarse en las tablas que aparecen a continuación:

Parámetro	*Valor*
Dispositivo	HETERO:FPGA,CPU
Iteraciones	1
Formato	FP32
Tiempo total de inferencia	4.39 ms
Tiempo de ejecución medio para una iteración	4.39 ms
Throughput	227.68 FPS
Clase	Coche deportivo
Probabilidad	0.82967

Tabla 4.4: Resultados para demo de clasificación. Una iteración en FPGA/CPU.

Parámetro	*Valor*
Dispositivo	HETERO:FPGA,CPU
Iteraciones	1
Formato	FP32
Tiempo total de inferencia	160 ms
Tiempo de ejecución medio para una iteración	1.6 ms
Throughput	FPS 623.73
Clase	Coche deportivo
Probabilidad	0.82967

Tabla 4.5: Resultados para demo de clasificación. Cien iteraciones en FPGA/CPU.

Parámetro	*Valor*
Dispositivo	HETERO:FPGA,CPU
Iteraciones	1
Formato	FP32
Tiempo total de inferencia	1534.96 ms
Tiempo de ejecución medio para una iteración	1.535 ms
Throughput	651.484 FPS
Clase	Coche deportivo
Probabilidad	0.82967

Tabla 4.6: Resultados para demo de clasificación. Mil iteraciones en FPGA/CPU.

Se puede comprobar como al aumentar el número de iteraciones el throughput alcanzado es mayor, por otra parte la precisión es similar en todos los casos así como la clase seleccionada. Estos resultados muestran como la herramienta calcula los frames por segundo incluyendo la inicialización del dispositivo.

4.4. CNN para clasificación de números escritos a mano

4.4.1. Estructura

En esta sección se pretende definir una Red Neuronal Convolucional con estructura y parámetros conocidos con la que se trabajará en el proceso de inferencia de OpenVINO en apartados posteriores. La red seleccionada tiene la finalidad de clasificar números escritos a mano tomados de un repositorio de datos o 'dataset' llamado Mnist. Dicho paquete fue creado Instituto Nacional de Estándares y Tecnología para el entrenamiento de redes neuronales y contiene 60000 imágenes de entrenamiento y 10000 de test. En la siguiente figura se observa un ejemplo de este tipo de imágenes.

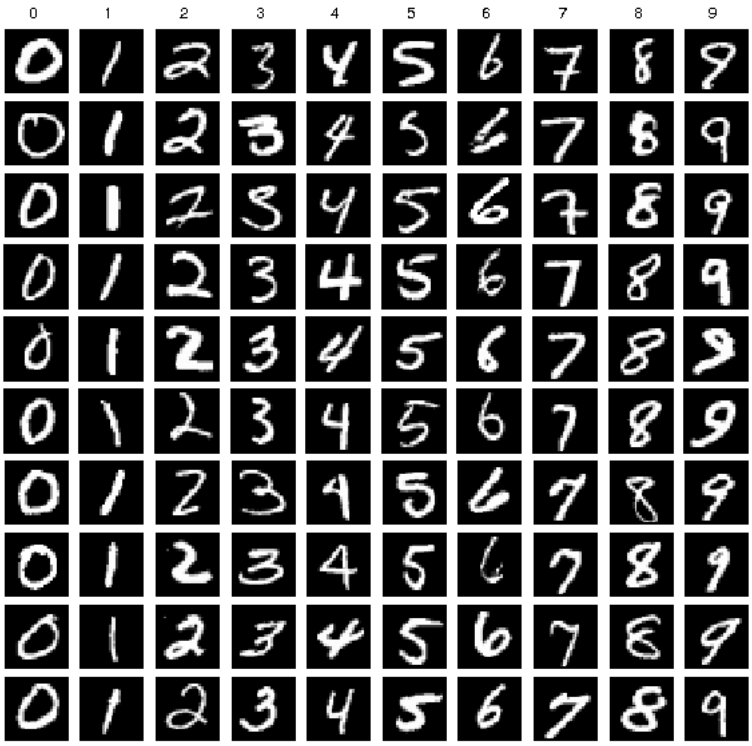

Figura 4.7: Ejemplo del dataset de números escritos a mano.

Tal como se vio en el Capítulo 2, las CNN combinan capas con filtros convolucionales para la extracción de características con capas de 'pooling' o reducción para disminuir el número de parámetros a entrenar. En este el caso de esta red, se ha seleccionado la estructura que aparece en la figura 4.8.

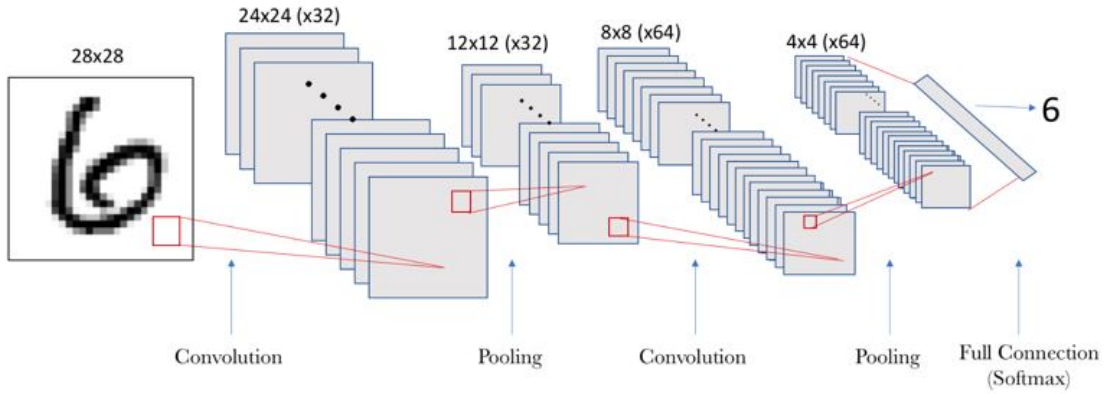

Figura 4.8: Esquema de la estructura de la CNN para clasificación de números escritos a mano.

Las imágenes de entrada tienen un tamaño de 28x28 y una profundidad de 1 puesto que son imágenes en blanco y negro. Cada capa convolucional hace las veces de filtro para extraer características más generales en función que aumentamos la profundidad de la red. La primera capa está formada por 32 filtros o kernels, que empleando una ventana de 5x5 y un desplazamiento de un pixel, forman un tamaño de 24x24 a su salida. Con el fin de reducir el tamaño de los filtros y por lo tanto la cantidad de datos a procesar, se añade una capa de 'pooling' con una ventana de 2x2.

Para obtener una estructura más compleja y más precisa, se añade una segunda capa interna de convolución con 64 filtros, empleando también una ventana de 5x5. A esta se añade otra capa de reducción para obtener 64 filtros de tamaño 4x4.

Con el fin de adaptar las dimensiones de la salida de esta segunda capa de reducción (4x4x64), a una función de activación final, se emplea una capa llamada 'Flatten' para transformar su entrada, a una salida de una única dimensión de tamaño 1024.

Como capa final se añade una capa totalmente conectada o densa con 10 salidas, una por clase. Alimentará a la función de activación que determinará que clase tiene mayor probabilidad.

Esta estructura puede definirse en Keras como:

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32,(5,5),activation='relu', input_shape=(28,28,1)))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (5, 5), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax'))
```

Código 4.1: Definición CNN de clasificación en Keras.

Con la función 'Sequential' se da comienzo a la definición de la estructura de la red 'model' tal como se ha comentado anteriormente. Para las funciones de activación de las capas convolucionales se emplea la función rampa llamada 'Relu' y como funciones de reducción se emplean las funciones 'MaxPooling', cuyo valor de salida es el máximo de los píxeles de la ventana 2x2. Finalmente se emplea una función de activación llamada Softmax, de uso muy común en capas de clasificación.

4.4.2. Entrenamiento

Resulta evidente que para la obtención de unos valores óptimos de los pesos es necesario un proceso de entrenamiento de la red. En primer lugar se dividen en dos grupos las imágenes de test y entrenamiento del 'dataset' Mnist:

```
from keras.datasets import mnist
from keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255
print(train_images.shape)

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255
print(test_images.shape)

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
>>(60000, 28, 28, 1)
>>(10000, 28, 28, 1)
```

Código 4.2: Obtener y configurar el dataset Mnist en Keras.

Además, se ha reajustado las dimensiones a las mismas que la entrada a la red y a valores normalizados de los píxeles. A continuación, se compila el modelo para poder ser entrenado. Se empleará una función de pérdidas llamada 'categorical_crossentropy', el optimizador de descenso por gradiente 'sdg'. También se medirá la precisión en el proceso.

Finalmente se entrena la red con el conjunto de imágenes de entrenamiento y sus etiquetas asociadas. Este proceso se realiza en paquetes de datos de 100 imágenes y para 5 épocas.

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

train_mod = model.fit(train_images, train_labels,
              batch_size=100,
              epochs=5,
              verbose=1,
```

```
            validation_data=(test_images, test_labels),)
```

Código 4.3: Compilar y entrenar el modelo de CNN para clasificación en Keras.

Los resultados de entrenamiento para cada época pueden examinarse en la siguiente figura:

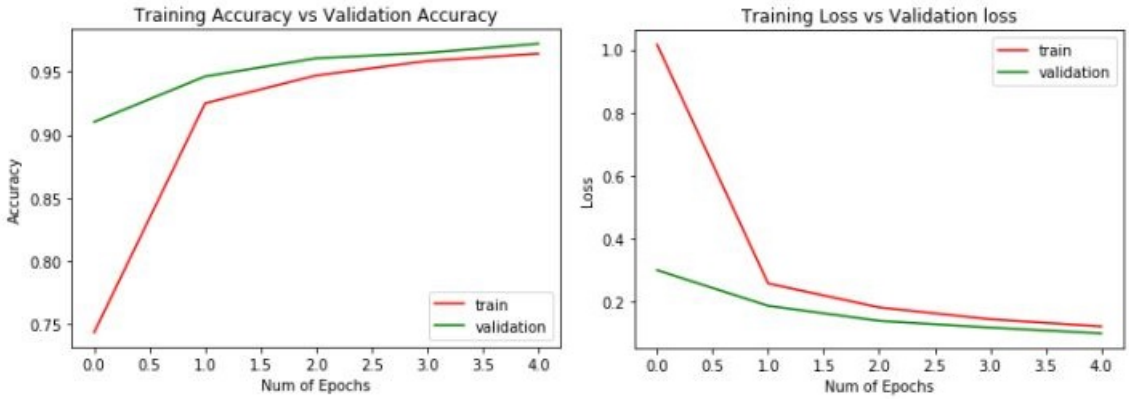

Figura 4.9: Evolución comparativa de la precisión y pérdidas en la fase de entrenamiento y validación.

Como se puede observar, para la fase de entrenamiento, la precisión aumenta desde valores inferiores al 80 % hasta alcanzarse un valor asintótico superior al 95 %. De manera complementaria, la función de pérdidas proporciona valores menores en cada época, partiendo de un máximo para la primera época y alcanzando un valor asintótico menor del 2 %.

Una vez ajustados los pesos finales, en la fase de validación se emplea el conjunto de test para comprobar la precisión alcanzada. Finalmente se muestran los valores finales de pérdidas y precisión alcanzados en el entrenamiento de la red.

```
test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy:', test_acc)
print('Test loss:', test_loss)

>>10000/10000 [==============================] - 1s 62us/step
>>('Test accuracy:', 0.9721999764442444)
>>('Test loss:', 0.09957398469634354)
```

Código 4.4: Valor de precisión y perdidas después del entrenamiento.

4.4.3. Predicción

Con el objetivo de obtener una comparación entre los resultados obtenidos en la predicción de clases sobre la plataforma de desarrollo de la red y OpenVINO, se realiza la predicción de una imagen seleccionada del conjunto de test.

De acuerdo con el código 4.6, se selecciona la segunda imagen número 2 del conjunto de test que corresponde con el número uno tal como indica su etiqueta.

```
import numpy as np
import matplotlib.pyplot as plt

def display_digit(num):
    plt.title('Ejemplo: %d - Label: %d' %
        (num,test_labels[num,:].argmax(axis=0)))
    plt.imshow(test_images[num,:].reshape([28,28]), cmap=plt.get_cmap('gray_r'))


Ejemplo = int(2)
display_digit(Ejemplo)
```

Código 4.5: Mostrar ejemplo de imagen de predicción.

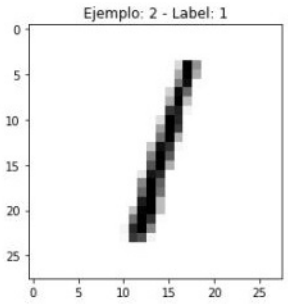

Figura 4.10: Imagen de ejemplo para la predicción.

Realizando la predicción de este ejemplo, se observa como la clase con mayor probabilidad es la correspondiente a la segunda posición del vector de predicción o lo que es lo mismo al número uno.

```
prediccion = model.predict(test_images[Ejemplo:(Ejemplo+1),:,:,:])
clase = model.predict_classes(test_images[Ejemplo:(Ejemplo+1),:,:,:])
print('Valor de cada Clase --> ',prediccion)
print('Clase de mayor probabilidad --> ',clase)

>>('Valor de cada Clase --> ', array([[2.75377970e-04, 9.89952147e-01,
    1.48330897e-03, 5.67623239e-04,
       2.89157708e-03, 1.11641995e-04, 6.56777178e-04, 1.94653368e-03,
       1.89476623e-03, 2.20270857e-04]], dtype=float32))
>>('Clase de mayor probabilidad --> ', array([1]))
```

Código 4.6: Predicción del ejemplo mostrado.

Este mismo ejemplo servirá como entrada a la red en la inferencia con OpenVINO.

4.4.4. Ejecución de CNN con OpenVINO

4.4.4.1. Modelo pre-entrenado de CNN

Como se ha visto en el flujo de trabajo de OpenVINO, es necesario la obtención de un archivo que contenga las características de la red entrenada. Al trabajar con Keras-Tensorflow se genera un gráfico cuyos parámetros se representan como variables congeladas para poder ser utilizadas por Model Optimizer [25].

Dicho gráfico congelado, se obtiene en un archivo '.pb' con el siguiente código.

```
def freeze_session(session, keep_var_names=None, output_names=None,
    clear_devices=True):

    from tensorflow.python.framework.graph_util import
        convert_variables_to_constants
    graph = session.graph
    with graph.as_default():
        freeze_var_names = list(set(v.op.name for v in
                                tf.global_variables()).difference(keep_var_names
                                    or []))
        output_names = output_names or []
        output_names += [v.op.name for v in tf.global_variables()]
        # Graph -> GraphDef ProtoBuf
        input_graph_def = graph.as_graph_def()
        if clear_devices:
            for node in input_graph_def.node:
                node.device = ""
        frozen_graph = convert_variables_to_constants(session, input_graph_def,
                                                      output_names, freeze_var_names)
        return frozen_graph


from keras import backend as K
import tensorflow as tf

frozen_graph = freeze_session(K.get_session(), output_names=[out.op.name for
                                                             out in model.outputs])
tf.train.write_graph(frozen_graph, "model", "tf_model0.pb", as_text=False)
```

Código 4.7: Congelar gráfico del modelo de red

Únicamente es necesario marcar el nombre del modelo de red creado y el nombre del archivo que se genera.

4.4.4.2. Preparar CNN

El objetivo de esta sección será obtener la representación intermedia (IR) de la red previamente entrenada para obtener una estructura y parámetros reconocibles por el motor de inferencia. Se emplea la herramienta Model Optimizer de la siguiente maneara:

$$mo\_tf.py\ -$$
$$-input\_model\ /home/jorga20j/Redes\_Jorge/cero\_red\_CNN/notebooks/model/tf\_model0.pb\ -$$
$$-input\_shape\ [1,28,28,1]\ --data\_typeFP32\ --model\_name\ tf\_model0\_fp32$$

Se emplea la herramienta optimizada para estructuras de Tensorflow, como modelo de entrada se incluye el modelo de red anteriormente creado, se especifica las dimensiones y tamaño de los valores de entrada. su formato y finalmente se propone un nombre para la IR de salida. De esta forma se obtienen los archivos .xml y .bin.

4.4.4.3. Inferencia CNN

La inferencia de la red se realiza siguiendo la metodología descrita en el flujo de trabajo, para ello se emplean las librerías de Python que la plataforma ofrece. Además, se emplean ciertas librerías de visión por computador (CV) para el preprocesado de las imágenes y así adaptarlas al formato de la inferencia.

A continuación, se muestran cada una de las partes del código que implementa el Motor de Inferencia para la CNN de clasificación de números escritos a mano. Para el resto de redes se trabaja de forma similar. En primer lugar se muestra las librerías y argumentos de entrada a la función principal donde se desarrolla todo el proceso:

```
from __future__ import print_function
import sys
import os
from argparse import ArgumentParser, SUPPRESS
import cv2
import numpy as np
import logging as log
from time import time
from openvino.inference_engine import IENetwork, IECore


def build_argparser():
    parser = ArgumentParser(add_help=False)
    args = parser.add_argument_group('Options')
    args.add_argument('-h', '--help', action='help', default=SUPPRESS,
        help='Show this help message and exit.')
    args.add_argument("-m", "--model", help="Required. Path to an .xml file with
        a trained model.", required=True,
                      type=str)
    args.add_argument("-i", "--input", help="Required. Path to a folder with
        images or path to an image files",
                      required=True,
                      type=str, nargs="+")
    args.add_argument("-l", "--cpu_extension",
                      help="Optional. Required for CPU custom layers. "
                           "MKLDNN (CPU)-targeted custom layers. Absolute path to a
                               shared library with the"
                           " kernels implementations.", type=str, default=None)
```

```
    args.add_argument("-d", "--device",
                      help="Optional. Specify the target device to infer on; CPU,
                          GPU, FPGA, HDDL, MYRIAD or HETERO: is "
                           "acceptable. The sample will look for a suitable plugin
                               for device specified. Default "
                           "value is CPU",
                      default="CPU", type=str)
    args.add_argument("--labels", help="Optional. Path to a labels mapping
        file", default=None, type=str)
    args.add_argument("-nt", "--number_top", help="Optional. Number of top
        results", default=10, type=int)

    return parser
```

Código 4.8: Librerias y argumentos de entrada a la función principal del motor de inferencia

Estos argumentos principales son la ayuda (-h), la ruta del modelo de red que se pretende realizar la inferencia (-m), la imagen o imágenes de entrada (-i), el dispositivo donde se ejecutará (-d), el archivo de etiquetas si es una red de clasificación (–labels) y el número de resultados con mayor probabilidad (-nt).

```
def main():
    log.basicConfig(format="[ %(levelname)s ] %(message)s", level=log.INFO,
        stream=sys.stdout)
    args = build_argparser().parse_args()
    model_xml = args.model
    model_bin = os.path.splitext(model_xml)[0] + ".bin"

    # Plugin initialization for specified device and load extensions library if
        specified
    log.info("Creating Inference Engine")
    ie = IECore()
    if args.cpu_extension and 'CPU' in args.device:
        ie.add_extension(args.cpu_extension, "CPU")
    # Read IR
    log.info("Loading network files:\n\t{}\n\t{}".format(model_xml, model_bin))
    net = IENetwork(model=model_xml, weights=model_bin)

    if "CPU" in args.device:
        supported_layers = ie.query_network(net, "CPU")
        not_supported_layers = [l for l in net.layers.keys() if l not in
            supported_layers]
        if len(not_supported_layers) != 0:
            log.error("Following layers are not supported by the plugin for
                specified device {}:\n {}".
                      format(args.device, ', '.join(not_supported_layers)))
            log.error("Please try to specify cpu extensions library path in
                sample's command line parameters using -l "
                      "or --cpu_extension command line argument")
            sys.exit(1)
```

```
    assert len(net.inputs.keys()) == 1, "Sample supports only single input
        topologies"
    assert len(net.outputs) == 1, "Sample supports only single output topologies"

    log.info("Preparing input blobs")
    input_blob = next(iter(net.inputs))
    out_blob = next(iter(net.outputs))
    net.batch_size = len(args.input)
    print("PARAMETROS RED")
    print("net.imputs: ", input_blob)
    print("net.outputs: ", out_blob)
    print("batch_size: ", net.batch_size)
```

Código 4.9: Parte 1 función principal del Motor de Inferencia

Como se muestra en el código 4.9, en primer lugar se cargan los argumentos de la IR y el plugin del dispositivo donde se realizará la inferencia. Se crea la red con la función 'IENetwork()' y se comprueba si todas las capas están soportadas sobre CPU. Finalmente se crean las variables de entrada, salida y el batch en función de la estructura de la red.

Seguidamente se crea el vector de imágenes con las dimensiones de los datos de entrada a la red. El preprocesado de las imágenes es un bucle donde cada imagen es transformada del formato de origen a las dimensiones y tamaño de la capa de entrada a la red. En este caso, las imágenes entran con un formato [28,28,3] proveniente de un archivo '.png' y se transforman a una topología [3,28,28].

De forma general sería de [H,W,C] a [C, H, W], donde C es el número de canales (RGB), H los píxeles verticales y W los horizontales. Al tratarse de un formato de entrada a la red de [1, 28, 28] y la imagen tener triplicada sus valores en sus dimensiones RGB, se iguala una de sus dimensiones al vector de entrada final (images[i]). Esto puede observarse en el código 4.10.

```
# Read and pre-process input images
    n, c, h, w = net.inputs[input_blob].shape
    images = np.ndarray(shape=(n, c, h, w))
    print("Valor de n: ", n)
    print("Valor de c: ", c)
    print("Valor de h: ", h)
    print("Valor de w: ", w)
    print("Images", images.shape)

    for i in range(n):
        image = cv2.imread(args.input[i])
        print(("Formato imagen",image.shape))
        if image.shape[:-1] != (h, w):
            log.warning("Image {} is resized from {} to {}".format(args.input[i],
                image.shape[:-1], (h, w)))
            image = cv2.resize(image, (w, h))
        image = image.transpose((2, 0, 1)) # Change data layout from HWC to CHW
        images[i] = image[2,:,:]
    log.info("Batch size is {}".format(n))
```

Código 4.10: Parte 2 función principal del Motor de Inferencia

Por último, se carga el modelo de red sobre el dispositivo y se realiza la inferencia de todas las imágenes. La salida es reportada de manera que cada imagen de entrada tiene asociadas las probabilidades de cada clase. La de mayor probabilidad es mostrada en primer lugar así como su relación con cada etiqueta.

```
# Loading model to the plugin
    log.info("Loading model to the plugin")
    exec_net = ie.load_network(network=net, device_name=args.device)

    # Start sync inference
    log.info("Starting inference in synchronous mode")
    res = exec_net.infer(inputs={input_blob: images})

    # Processing output blob
    log.info("Processing output blob")
    res = res[out_blob]
    log.info("Top {} results: ".format(args.number_top))
    if args.labels:
        with open(args.labels, 'r') as f:
            labels_map = [x.split(sep=' ', maxsplit=1)[-1].strip() for x in f]
    else:
        labels_map = None
    classid_str = "classid"
    probability_str = "probability"
    for i, probs in enumerate(res):
        probs = np.squeeze(probs)
        top_ind = np.argsort(probs)[-args.number_top:][::-1]
        print("Image {}\n".format(args.input[i]))
        print(classid_str, probability_str)
        print("{} {}".format('-' * len(classid_str), '-' * len(probability_str)))
        for id in top_ind:
            det_label = labels_map[id] if labels_map else "{}".format(id)
            label_length = len(det_label)
            space_num_before = (len(classid_str) - label_length) // 2
            space_num_after = len(classid_str) - (space_num_before + label_length)
                + 2
            space_num_before_prob = (len(probability_str) - len(str(probs[id])))
                // 2
            print("{}{}{}{}{:.7f}".format(' ' * space_num_before, det_label,
                                          ' ' * space_num_after, ' ' *
                                              space_num_before_prob,
                                          probs[id]))
```

Código 4.11: Parte 3 función principal del Motor de Inferencia

Para comprobar el correcto funcionamiento del motor de inferencia, se propone como imagen de entrada, la misma que la mostrada en la figura **??**. Por lo tanto, los argumentos de entrada al

motor de inferencia son:

$$python3\ cero\_IE.py\ -m\ /home/jorga20j/Redes\_Jorge/cero\_red\_CNN/Inference\_Engine\_CPU/FP32/tf\_model0\_fp32.xml\ -i\ /home/jorga20j/Redes_jorge/cero\_red\_CNN/Inference\_Engine\_CPU/FP32/Images/uno.png\ --device\ CPU\ --labels\ number.labels$$

y cuyos resultados se muestran en la figura 4.11

```
[ INFO ] Top 10 results:
Image /home/jorga20j/Redes_Jorge/cero_red_CNN/
no.png

classid probability
------- -----------
   1          1.0000000
   9          0.0000000
   8          0.0000000
   7          0.0000000
   6          0.0000000
   5          0.0000000
   4          0.0000000
   3          0.0000000
   2          0.0000000
   0          0.0000000
```

Figura 4.11: Resultado ejemplo Motor de Inferncia para CNN.

Como se puede observar la clase con la probabilidad mayor es la case correspondiente al número 1. Se puede concluir que el proceso de inferencia de la CNN de clasificación de números escritos a mano sobre OpenVINO ha sido correcta.

4.5. Plataforma PETALO

PETALO es un sistema basado en liquido xenon (Lxe) para la Emisión de Positrones en Tomografía (PET), una técnica no invasiva que escanea la actividad metabólica del cuerpo para el diagnóstico.

El objetivo principal de este sistema será la determinación del punto de choque de los rayos gamma para la reconstrucción de imágenes PET. En concreto, PETALO está formado por un total de 3500 sensores dispuestos en forma de anillo y repartidos matricialmente tal y como se muestra en la siguiente figura:

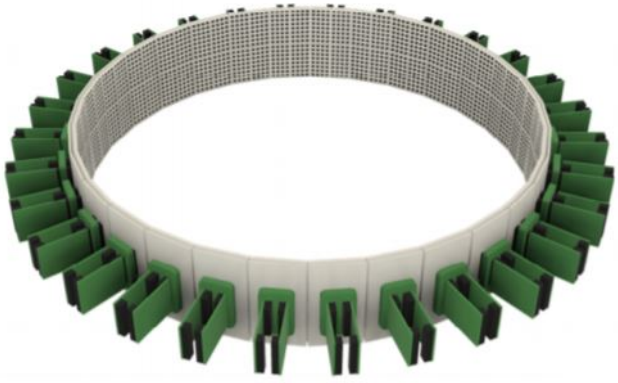

Figura 4.12: Disposición de los sensores en PETALO.

La cantidad de información generada por evento detectado es mucho más alto que el de un detector segmentado (común), y la electrónica de adquisición de datos debe ser capaz de procesar esto para mejorar el rendimiento del sistema.

Además, un problema potencial en un PET basado en LXe es la llamada dispersión de Compton. Un choque de rayos gamma a 511 keV puede sufrir dispersión Compton:

- Fuera de la región activa. Tales eventos pueden eliminarse mediante un corte de energía, y por lo tanto, la eficiencia con la que esto se puede hacer es dependiente de la resolución energética del detector.
- Dentro de la región activa. Para estos rayos gamma toda la energía producida será observada, pero el patrón de centelleo observado en los sensores no será únicamente interacción puntual. Esto podría afectar a la ubicación de la interacción que se pretende reconstruir y, por lo tanto, proporciona una línea de respuesta (LOR) reconstruida incorrectamente. Este efecto se observa en la figura 4.13.

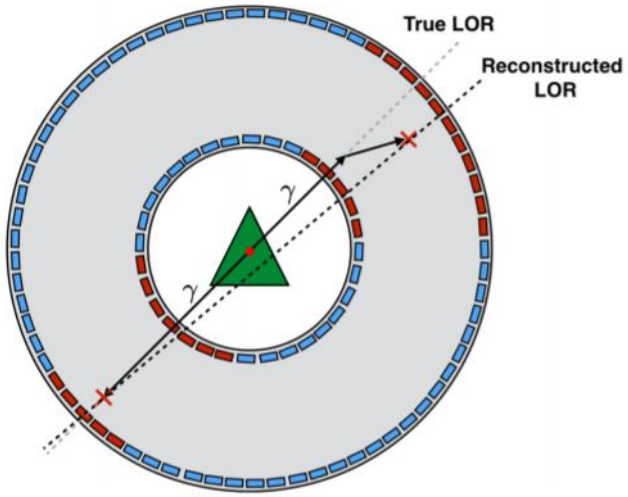

Figura 4.13: Efecto Compton dentro de la región activa.

Para solventar este ultimo problema, se emplea un sistema basado en redes neuronales que tratará de predecir la posición correcta o real del choque de dichos rayos gamma [26].

4.6. CNN para predicción de profundidad de positrones

En primer lugar se presenta una red convolucional cuyo propósito es la predicción de la profundidad del positrón en función del valor de los sensores. El dataset con el conjunto de datos de entrenamiento y test es proporcionado en la fase inicial del proyecto y obtenido a partir de la simulación de un modelo del sistema PETALO.

Este dataset contiene el valor de los sensores para una profundidad determinada. En este caso el conjunto de datos ha sido previamente procesado para obtener paquetes de datos de tamaño 20 x 33.

4.6.1. Estructura

La estructura de la red desarrollada en Keras es la que se muestra en el siguiente código:

```
# input image dimensions
img_rows, img_cols = 20, 33
# number of convolutional filters to use
nb_filters = 32
# size of pooling area for max pooling
pool_size = (2, 2)
# convolution kernel size
kernel_size = (4, 4)

model = Sequential()

model.add(Conv2D(16, kernel_size=kernel_size, padding='same',
    input_shape=(img_rows,img_cols,1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))
model.add(Conv2D(16, kernel_size, padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))
model.add(Conv2D(32, kernel_size, padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, kernel_size, padding='same'))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(1))
```

Código 4.12: Estructura de CNN para profundidad de positrones

Como se puede observar se trata de una red con cuatro capas convolucionales seguidas todas ellas de una capa de reducción o pooling. Además, al final se emplea una capa 'Flatten' para

transformar a una única dimensión los valores obtenidos de la última capa de pooling. Finalmente se utiliza una única neurona totalmente conectada para determinar el valor de predicción a la salida.

En este caso se parte de un modelo pre-entrenado por Rafael Gadea Gironés y cuyos pesos han sido cargados a la red de la siguiente manera:

```
experimento="CNN_original"
algoritmo='Nadam'
model.compile(loss='mean_squared_error', optimizer=algoritmo)

model.load_weights('../redes_CNN_R/defs/archivo_pesos.167505')
```

Código 4.13: Modelo compilado y carga de pesos para CNN predicción de profundidad.

4.6.2. Predicción

Con el objetivo de obtener una comparación entre los resultados obtenidos en la predicción con plataforma de desarrollo de la red y OpenVINO, se realiza la predicción de un ejemplo seleccionado del conjunto de test. Este ejemplo es el que se muestra a continuación:

```
import matplotlib.pyplot as plt

def display_image(num):
    plt.title('Ejemplo: %d - valor: %d' % (num,Y_test[num]))
    plt.imshow(np.reshape(X_test[num], [img_rows, img_cols]), cmap='viridis')

Ejemplo = int(3)
display_image(Ejemplo)
```

Código 4.14: Ejemplo para predicción del conjunto de test.

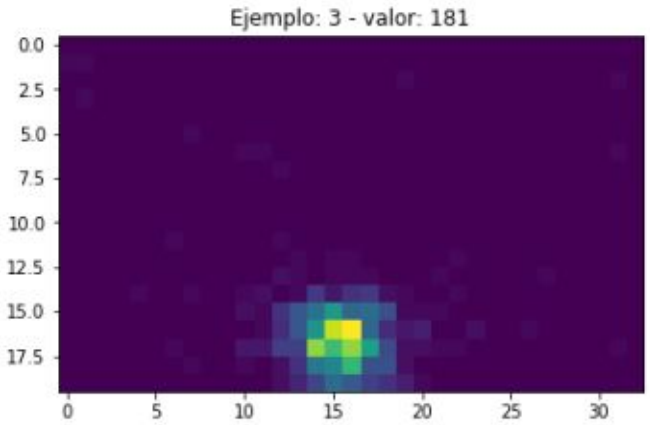

Figura 4.14: Ejemplo para predicción del conjunto de test.

Realizando la predicción de dicha entrada sobre la red creada en Keras se obtiene el siguiente resultado:

```
Y_test_predicted=model.predict(X_test[Ejemplo:(Ejemplo+1),:,:,:])
print(Y_test_predicted)

>>[[180.381]]
```

Código 4.15: Predicción del ejemplo del conjunto de test.

4.6.3. Inferencia sobre OpenVINO

Tras congelar el gráfico de la red tal como se vio en la CNN para la predicción de números escritos a mano, se realiza todo el proceso de obtención de la IR con la herramienta Model Optimizer y se realiza la inferencia procediendo de igual manera que en los códigos 4.8, 4.9, 4.10 y 4.11, sin embargo en este caso los datos de entrada a la red no se incluyen como un argumento de entrada al motor de inferencia sino que previamente se han guardado con extensión 'csv' por medio de la librería Pandas en Python y se cargan en el motor de inferencia como:

```
import pandas as pd

 # Read and pre-process input images
    n, c, h, w = net.inputs[input_blob].shape
    input_data = dato_in=pd.read_csv('X_test.csv',
        sep=',',header=0,index_col=0).to_numpy()

    # Loading model to the plugin
    log.info("Loading model to the plugin")
    exec_net = ie.load_network(network=net, device_name=args.device)

    # Start sync inference
    log.info("Starting inference in synchronous mode")
    res = exec_net.infer(inputs={input_blob: input_data})

    # Processing output blob
    log.info("Processing output blob")
    res = res[out_blob]
    log.info("TOP 1 result")
    print("\n")
    print("Value --> ", res)
```

Código 4.16: Inferencia CNN para predicción de profundidad sobre OpenVINO.

Como se puede observar, únicamente se trabaja con una clase a la salida, siendo su valor el correspondiente a la predicción de profundidad del positrón.

Finalmente, se lanza el motor de inferencia con la entrada de ejemplo mostrada en el apartado anterior obteniendo los siguientes resultados:

```
net.outputs:  dense_1/MatMul
batch_size:  1
[ INFO ] Loading model to the plugin
[ INFO ] Starting inference in synchronous mode
[ INFO ] Processing output blob
[ INFO ] TOP 1 result


Value -->  [[180.38101]]
```

Figura 4.15: Resultado ejecución de ejemplo para CNN profundidad del positrón en OpenVINO.

Se puede concluir que la red se ha procesado correctamente sobre OpenVINO.

4.7. Red densa basada en el error

Como se ha comentado, la finalidad de emplear redes neuronales en este tipo de reconstrucción de imagen es mejorar las prestaciones computacionales y obtener un valor más próximo a la realidad. Con esta segunda red se pretende realizar una predicción del error cometido en la obtención de la coordenada Z del choque de los rayos gamma con respecto al valor obtenido a través de un método analítico. Dicha coordenada mide el desplazamiento vertical del choque de acuerdo con la disposición de sensores visto en la figura 4.12.

El dataset de entrenamiento y test ha sido obtenido a partir de los resultados de la simulación de un modelo de PETALO y los cálculos necesarios para obtener una predicción basada en un método analítico. Es preciso comentar que la red ha sido pre-entrenada por el investigador Josh Renner.

4.7.1. Estructura

La estructura de la red densa definida en keras es la siguiente:

```
model = Sequential()

model.add(Flatten(input_shape=(20,20,1)))
model.add(Dense(units=64, kernel_initializer="glorot_normal", activation="relu"))
model.add(Dense(units=32, kernel_initializer="glorot_normal", activation="relu"))
model.add(Dense(units=16, kernel_initializer="glorot_normal", activation="relu"))
model.add(Dense(units=8, kernel_initializer="glorot_normal", activation="relu"))
model.add(Dense(units=1, kernel_initializer="glorot_normal", activation="relu"))

model.compile(loss='mse', optimizer=optimizers.Nadam(lr=5e-4,
beta_1=0.9, beta_2=0.999, epsilon=1e-09, schedule_decay=0.001),
    metrics=['accuracy'])
```

Código 4.17: Estructura red densa basada en el error

Como se puede observar, se parte de un formato de entrada de 20x20, este tamaño deja alrededor del 4 % de los rayos gamma descartados para el entrenamiento y ejecución de la red. Su

estructura consiste en una capa de entrada de 400 posiciones que son conectadas a cuatro capas densas de 64, 32, 16 y 8 neuronas respectivamente. a la salida se encuentra una única neurona totalmente conectada que proporciona el valor de predicción del error z normalizado en un rango de 0 a 1.

Al tratarse de un modelo pre-entrenado, se cargan los pesos de la red como:

```
model.load_weights('weights_zerr_FC_IEEE.h5')
```

Código 4.18: Carga de pesos en red densa basada en el error.

4.7.2. Predicción

Con el objetivo de obtener una comparativa entre los resultados obtenidos sobre Keras en la predicción del error con OpenVINO, se muestra un ejemplo de dicha predicción:

```
import matplotlib.pyplot as plt
img_rows = 20
img_cols = 20

def display_image(num):
    plt.title('Ejemplo: %d - valor: %f' % (num,y_pred[num]))
    plt.imshow(np.reshape(x_test[num], [img_rows, img_cols]), cmap='viridis')

Ejemplo = int(1)
display_image(Ejemplo)
```

Código 4.19: Ejemplo predicción para red densa basada en el error

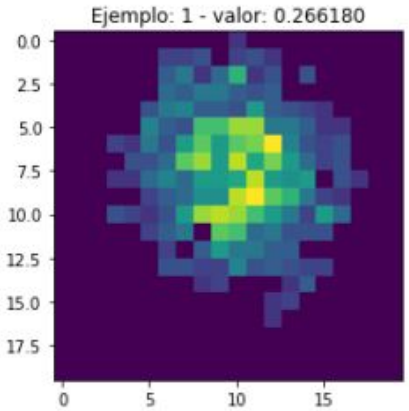

Figura 4.16: Ejemplo predicción para red densa basada en el error.

Realizando la predicción de dicha entrada sobre la red creada en Keras se obtiene el siguiente resultado:

```
Y_test_predicted=model.predict(x_test[Ejemplo:(Ejemplo+1):,:,:])
print(Y_test_predicted)

>>[[0.2661804]]
```

Código 4.20: Resultado predicción para red densa basada en el error

4.7.3. Inferencia sobre OpenVINO

Procediendo de igual forma que en casos anteriores se congela el gráfico de la red con sus parámetros, se obtiene la representación intermedia con Model Optimizer y se procede a realizar la inferencia del modelo sobre OpenVINO. Para ello se obtiene la entrada a la red en formato 'csv' y se carga al motor de inferencia. Procediendo como en los códigos 4.8, 4.9, 4.10 y 4.11 se definen las librerías pertinentes y argumentos de entrada, se inicializan los plugin específicos del dispositivo de inferencia y se lee la IR. Finalmente, se cargan las entradas y el modelo de red sobre el plugin de inferencia y se comienza el proceso como:

```
# Read and pre-process input images
    n, c, h, w = net.inputs[input_blob].shape
    input_data = dato_in=pd.read_csv('x_test_segunda.csv',
        sep=',',header=0,index_col=0).to_numpy()

    # Loading model to the plugin
    log.info("Loading model to the plugin")
    exec_net = ie.load_network(network=net, device_name=args.device)

    # Start sync inference
    log.info("Starting inference in synchronous mode")
    res = exec_net.infer(inputs={input_blob: input_data})

    # Processing output blob
    log.info("Processing output blob")
    res = res[out_blob]
    log.info("TOP 1 result")
    print("\n")
    print("Z error --> ", res)
```

Código 4.21: Inferencia red basada en el error sobre OpenVINO.

Tras lanzar el motor de inferencia con la configuración anterior y el ejemplo mostrado en la figura 4.17 se obtiene:

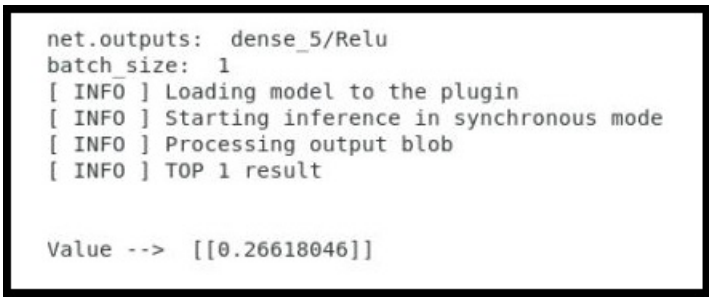

Figura 4.17: Predicción para red densa basada en el error sobre OpenVINO.

Se puede concluir nuevamente que el resultado de la inferencia con OpenVINO ha sido correcto.

Capítulo 5

Resultados

Los resultados mostrados en este capítulo hacen referencia al throughput y latencia conseguidos para las tres redes tratadas anteriormente, llamadas Red 0 a la red convolucional para la clasificación de números escritos a mano, Red 1 a la red convolucional del sistema PETALO para la predicción de la profundidad de los positrones y Red 2 a la red densa, también de la plataforma PETALO, para la predicción del error con con el método analítico de la coordenada z del positrón.

Dichas redes han sido transformadas a su representación intermedia en formatos de 16 y 32 bits en coma flotante. La ejecución de las mismas ha sido aplicada a CPU, FPGA y Heterogeneous. Para el caso de FPGA y Heterogeneous se ha empleando el bitstream proporcionado por la herramienta '2019R1_RC_FP16_ResNet _SqueezeNet_VGG.aocx' sobre la placa 'Intel® Programmable Acceleration Card con Intel Arria® 10 GX FPGA'.

En concreto, la herramienta Benchmark permite realizar la inferencia de las redes para obtener dichos valores, configurada en modo síncrono. Se han efectuado 10 ejecuciones para cada red con un total de 10000 iteraciones del proceso en cada ejecución para reducir la influencia de los tiempos de inicialización de los dispositivos dónde se han realizado las inferencias.

Es preciso comentar que la ejecución completa sobre FPGA no ha sido posible debido a que ciertas estructuras de las redes no están soportadas por el dispositivo, en concreto las capas 'Flatten'. Por otra parte, el formato de 16 bits en coma flotante no está soportado para la ejecución única sobre CPU. Con todo ello, se presentan los resultados obtenidos en las siguientes secciones.

5.1. Resultados primera CNN sobre OpenVINO

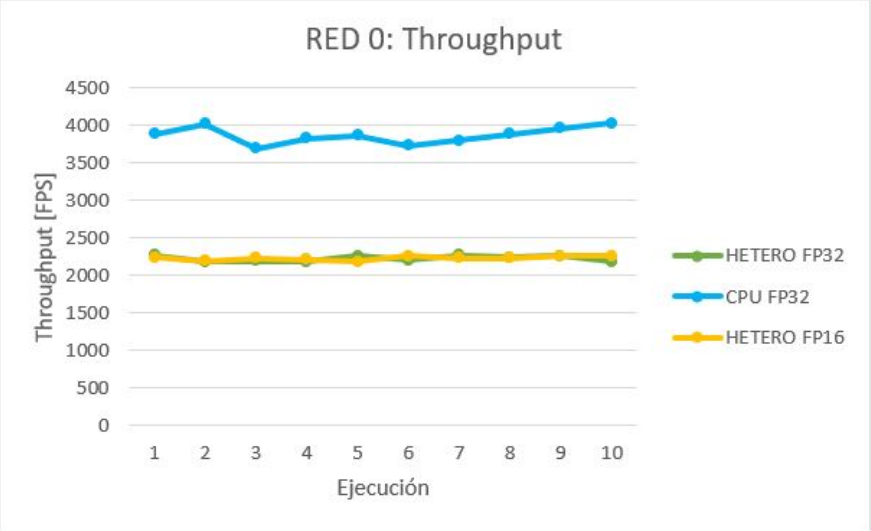

Figura 5.1: Througput red 0.

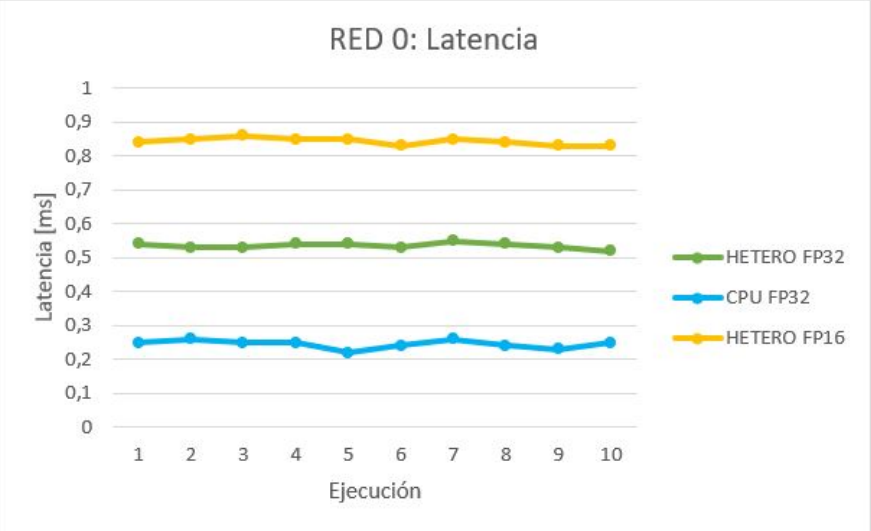

Figura 5.2: Latencia red 0.

5.2. Resultados CNN PETALO sobre OpenVINO

Figura 5.3: Througput red 1.

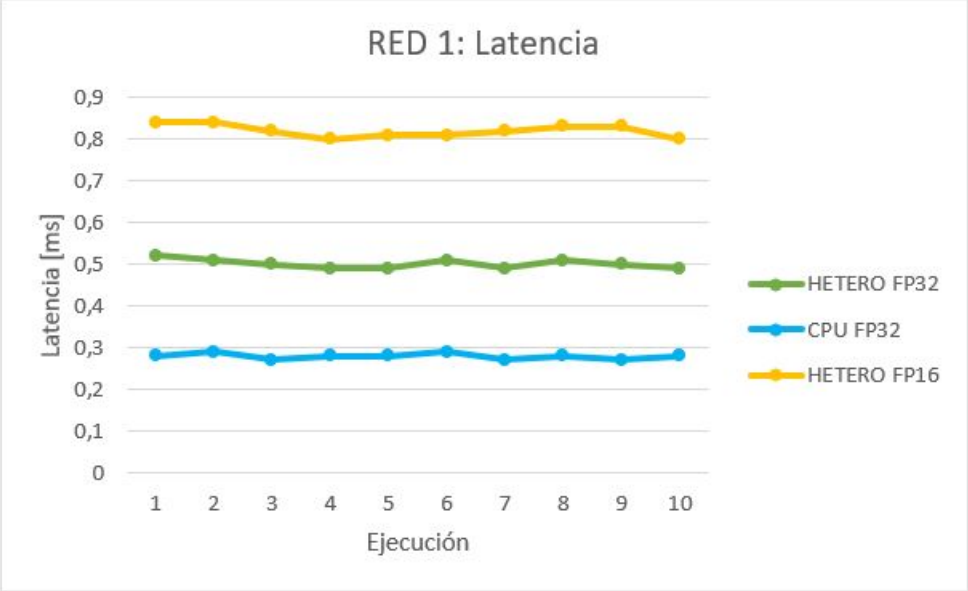

Figura 5.4: Latencia red 1.

5.3. Resultados Red Densa PETALO sobre OpenVINO

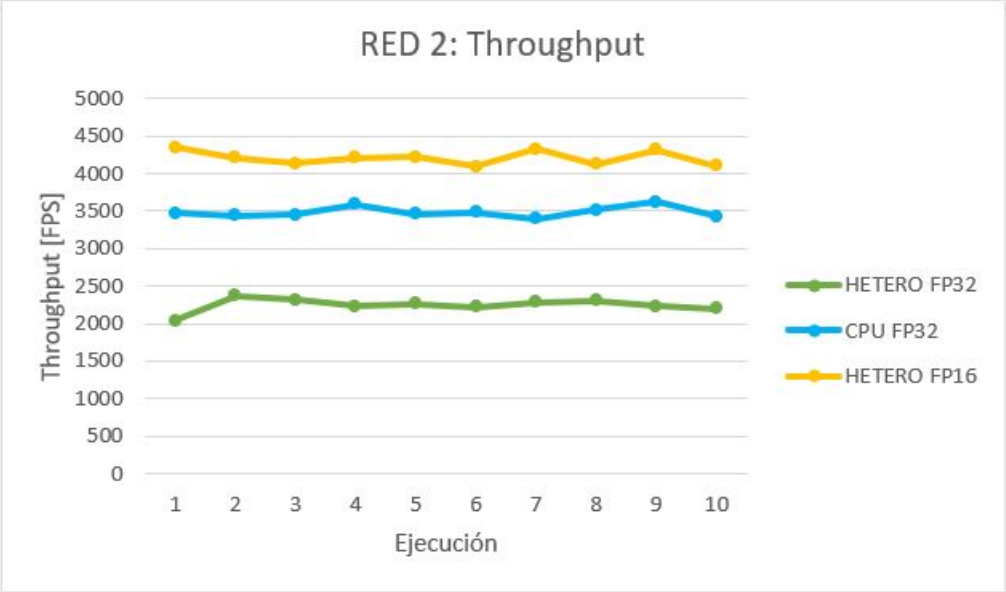

Figura 5.5: Througput red 2.

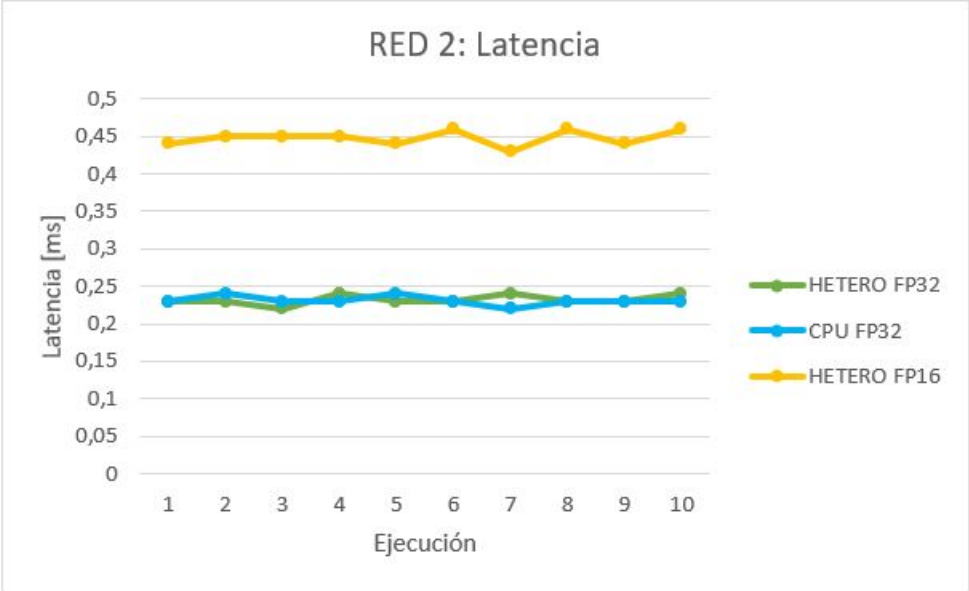

Figura 5.6: Latencia red 2.

Capítulo 6

Conclusiones

Tras finalizar este trabajo y atendiendo al planteamiento inicial del mismo, se ha logrado realizar la inferencia de las redes neuronales para PETALO. Estudiando el flujo de trabajo de OpenVINO y empleando diversas herramientas que proporciona la plataforma, se ha conseguido obtener la comparativa de throughput y latencia en diferentes formatos y dispositivos.

Alguna de las ventajas que proporciona la utilización de OpenVINO es por ejemplo la capacidad de abstracción para la implementación de redes neuronales sobre plataformas heterogéneas, desde CPU hasta FPGA. Esto proporciona una gran reducción en los tiempos de desarrollo para lograr obtener aplicaciones sobre sistemas embebidos. También existen gran cantidad de ejemplos de redes de uso común en distintas aplicaciones que están optimizadas para su ejecución sobre FPGA y que proporciona la herramienta, por lo tanto es muy útil para aplicaciones de uso común en Deep Learning.

Por una parte, se han comprobado algunos de los inconvenientes de emplear esta herramienta. Sin embargo, al tratarse de una plataforma con poco recorrido todavía esta sujeta a grandes mejoras. Por ejemplo, tanto la herramienta Model Optimizer como los bitstream precompilados para FPGA tienen la capacidad de soportar mayor número de estructuras diferentes para así obtener mayor flexibilidad en el diseño de redes propias y mejores prestaciones a nivel computacional.

Finalmente, atendiendo a los resultados obtenidos, tanto para la red convolucional de clasificación de números escritos a mano como para la red convolucional de predicción de la profundidad de positrones, se ha comprobado como la ejecución heterogénea sobre FPGA y CPU, tanto para 16 como 32 bit, proporciona peores resultados de throughput y latencia que una inferencia sobre CPU únicamente. La convolución es una de las estructuras que mayores requisitos computacionales requiere y la herramienta, en la versión empleada, todavía esta sujeta a mejoras sobre FPGA. Sin embargo, para la red densa se ha visto como con una ejecución heterogénea FPGA - CPU de 16 bits es mucho más rápida que en su versión de 32 bits. Este efecto también se observa sobre la latencia obtenida.

6.1. Estudios futuros

En primer lugar, se propone la implementación de las redes sobre versiones superiores de la herramienta con el fin de observar las mejoras realizadas sobre la plataforma. A fecha de finalización de este trabajo ya se dispone de la versión 3 de la misma. En segundo lugar, OpenVINO dispone de una herramienta llamada 'Deep Learning Acceleration Suite' que permite la generación de kernels propios con OpenCL. Esta puede emplearse para generar las capas que todavía no son compatibles en FPGA de las redes trabajadas, de esta forma se realizaría un ejecución total sobre FPGA. También podría emplearse para optimizar capas como la convolución, cuyos requerimientos computaciones podrían ser críticos en el proceso de inferencia.

Con el fin de comprobar si los dispositivos FPGA son óptimos para la implementación de redes neuronales se propone realizar una comparativa con la ejecución sobre GPU. Además, se abre la posibilidad a estudiar herramientas similares, por ejemplo Vitis de Xilinx.

Bibliografía

[1] *Historia de la Inteligencia Artificial*. http://ocw.uc3m.es/ingenieria-telematica/inteligencia-en-redes-de-comunicaciones/material-de-clase-1/01-historia-de-la-inteligencia-artificial. (Accessed on 11/28/2019). 2018.

[2] *inteligencia | Definición | Diccionario de la lengua española | RAE - ASALE*. https://dle.rae.es/inteligencia?m=. (Accessed on 11/28/2019).

[3] Andrea Piccione. *Introduction to Deep Learning: a practical point of view*. http://hdl.handle.net/10251/107792. (Accessed on 11/28/2019). Agosto de 2018.

[4] John McCarthy. *Arthur Samuel*. http://infolab.stanford.edu/pub/voy/museum/samuel.html. (Accessed on 11/28/2019).

[5] *Machine learning - Wikipedia*. https://en.wikipedia.org/wiki/Machine_learning. (Accessed on 11/28/2019). Nov. de 2019.

[6] Bernard Widrow y Michael A Lehr. "30 years of adaptive neural networks: perceptron, madaline, and backpropagation". En: *Proceedings of the IEEE* 78.9 (1990), págs. 1415-1442.

[7] Sudarshan Nandy, Partha Pratim Sarkar y Achintya Das. "Training a feed-forward neural network with artificial bee colony based backpropagation method". En: *arXiv preprint arXiv:1209.2548* (2012).

[8] *Clasificación de redes neuronales artificiales - Diego Calvo*. http://www.diegocalvo.es/clasificacion-de-redes-neuronales-artificiales/. (Accessed on 12/02/2019). Jul. de 2017.

[9] *Deep Learning Básico Parte Final: Autoencoders - Jaime Crispi - Medium*. https://medium.com/@jcrispis56/deep-learning-basico-parte-final-autoencoders-3d5c6fff2966. (Accessed on 12/02/2019).

[10] *CPU, GPU, FPGA or TPU: Which one to choose for my Machine Learning training? – InAccel*. https://inaccel.com/cpu-gpu-fpga-or-tpu-which-one-to-choose-for-my-machine-learning-training/. (Accessed on 12/03/2019).

[11] *fpt16-accelerating-bnn.pdf*. https://www.jaewoong.org/pubs/fpt16-accelerating-bnn.pdf. (Accessed on 12/03/2019).

[12] *Anaconda | The World's Most Popular Data Science Platform*. https://www.anaconda.com/. (Accessed on 12/04/2019).

[13] *Home - Keras Documentation*. https://keras.io/. (Accessed on 12/04/2019).

[14] *Intel® Distribution of OpenVINO™ Toolkit | Intel® Software*. https://software.intel.com/en-us/openvino-toolkit. (Accessed on 12/04/2019).

[15] *Intel® Programmable Acceleration Card with Intel Arria® 10 GX FPGA - Overview*. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx/overview.html. (Accessed on 12/04/2019).

[16] *Intel® PAC con FPGA Arria® 10 GX Especificaciones de productos*. https://ark.intel.com/content/www/es/es/ark/products/149169/intel-pac-with-arria-10-gx-fpga.html. (Accessed on 12/04/2019).

[17] *Introduction to Intel® Deep Learning Deployment Toolkit - OpenVINO Toolkit*. https://docs.openvinotoolkit.org/latest/_docs_IE_DG_Introduction.html. (Accessed on 12/06/2019).

[18] *Model Optimizer Developer Guide - OpenVINO Toolkit*. https://docs.openvinotoolkit.org/latest/_docs_MO_DG_Deep_Learning_Model_Optimizer_DevGuide.html. (Accessed on 12/06/2019).

[19] *Inference Engine Developer Guide - OpenVINO Toolkit*. https://docs.openvinotoolkit.org/latest/_docs_IE_DG_Deep_Learning_Inference_Engine_DevGuide.html. (Accessed on 12/06/2019).

[20] *Heterogeneous Plugin - OpenVINO Toolkit*. https://docs.openvinotoolkit.org/latest/_docs_IE_DG_supported_plugins_HETERO.html. (Accessed on 12/06/2019).

[21] *FPGA Plugin - OpenVINO Toolkit*. https://docs.openvinotoolkit.org/latest/_docs_IE_DG_supported_plugins_FPGA.html. (Accessed on 12/06/2019).

[22] *intel-vision-accelerator-design-with-FPGA-wp.pdf*. https://www.mouser.com/pdfdocs/intel-vision-accelerator-design-with-FPGA-wp.pdf. (Accessed on 12/06/2019).

[23] *Benchmark Application Demo - OpenVINO Toolkit*. https://docs.openvinotoolkit.org/2018_R5/_samples_benchmark_app_README.html. (Accessed on 12/10/2019).

[24] *OpenVINO Toolkit and FPGAs - Tech.Decoded powered by Intel Software*. https://techdecoded.intel.io/resources/openvino-toolkit-and-fpgas. (Accessed on 12/11/2019).

[25] *Converting a TensorFlow* Model - OpenVINO Toolkit*. https://docs.openvinotoolkit.org/latest/_docs_MO_DG_prepare_model_convert_model_Convert_Model_From_TensorFlow.html. (Accessed on 12/12/2019).

[26] Vicente Herrero y col. "PETALO read-out: A novel approach for data acquisition systems in PET applications". En: mayo de 2019. DOI: 10.1109/NSSMIC.2018.8824293.