UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en ingeniería de tecnologías y servicios de telecomunicación

TRABAJO FIN DE GRADO

Algoritmos de aprendizaje profundo para procesamiento de video en dispositivos Xilinx Zynq UltraScale+ de bajo coste

Autor: Mario Gil Martínez
Tutor: Sergio López Buedo

JUNIO 2021

Algoritmos de aprendizaje profundo para procesamiento de video en dispositivos Xilinx Zynq UltraScale+ de bajo coste

AUTOR: Mario Gil Martínez
TUTOR: Sergio López Buedo

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2021

Resumen

Las redes neuronales han sido la revolución tecnológica de comienzos de este siglo, debido a su gran potencial anteriormente inalcanzable debido al alto nivel de computación requerido para ello. Es por eso que cada año aparece nueva documentación y estudios sobre redes neuronales, puesto que al aumentar la tecnología de procesado se aumenta la capacidad de implementar mejores redes. Aun así a veces no se busca la mejor o más potente solución de entrenamiento de redes más potente, sino una adaptada a cada caso de inferencia, por lo que se buscan opciones de bajo consumo, baja latencia o que puedan ir en sistemas *Edge*. Es ahí donde aparecen opciones como las FPGA para adaptarse a distintas necesidades, y con ellas Xilinx con el software de Vitis AI con su entorno de desarrollo para implementar redes neuronales sobre sus arquitecturas de hardware de forma más sencilla y dinámica.

En este trabajo se parte de la necesidad de explorar la implementación de redes neuronales sobre FPGA para ver su capacidad de dar soluciones a aplicaciones reales, como, en el caso a desarrollar, de clasificación de imágenes mediante CNN. Primero se hace un repaso al estado actual del arte de las redes neuronales desde sus orígenes hasta la actualidad. A continuación, se explica la forma de implementar estos sistemas mediante el software de Vitis AI y las herramientas que proporciona respecto a un desarrollo clásico. Por ultimo se realiza un caso practico del que se parte de una descripción de una red neuronal en Python hasta su despliegue en la placa ZCU104, detallando el flujo de trabajo realizado y obteniendo resultados de rendimiento y precisión para valorar el uso de estas tecnologías como solución de inferencia para redes neuronales.

Palabras clave

Aprendizaje automático, Aprendizaje profundo, Redes neuronales, CNN, computación edge, FPGA, Vitis AI, visión artificial

Abstract

Neural networks have been the technological revolution of the beginning of this century, due to their great potential previously unattainable due to the high level of computing required for it. That is why new documentation and studies on neural networks appear every year, since the new processing technology increases the capacity to implement better networks. Even so, sometimes the best or most powerful solution for the most powerful network training is not sought, but one adapted to each case of inference, which is why low consumption, low latency options are sought or that can be used in Edge systems. That is where options such as FPGAs appear to adapt to different needs, and that why Xilinx came with Vitis AI software and its development environment to implement neural networks on their hardware architectures in a simpler and more dynamic way.

This work is based on the need to explore the implementation of neural networks on FPGAs and to see their ability to provide solutions to real applications, such as, in the case to be developed, image classification through CNN. First, a review is made of the current state of the art of neural networks from its origins to the present. The following explains how to implement these systems using the Vitis AI software and the tools it provides compared to a classic development. Finally, a practical case is carried out, starting with a description of a neural network in Python until its deployment on the ZCU104 board, detailing the workflow carried out and obtaining performance and precision results to assess the use of these technologies as a solution. of inference for neural networks.

Keywords

Machine Learning, Deep Learning, Neural Networks, CNN, Edge Computing, FPGA, Vitis AI, Computer Vision

Agradecimientos

En agradecimiento de todas las personas que he conocido durante la carrera y me han ayudado, enseñado y compartido horas de trabajo conmigo, a mi tutor Sergio por guiarme en este trabajo, a mis grandes amigos que he conocido gracias al grado y por último y más importante a mi familia, mi padre y mi madre, por apoyarme siempre y estar conmigo cuando los he necesitado.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGO

ÍNDICE DE TABLAS

1 Introducción

La inteligencia artificial es un campo dentro de las ciencias de computación que ha aumentado increíblemente su relevancia en los últimos 50 años. Dentro de este campo el aprendizaje automático (*machine learning*) es la tecnología que permite que las computadoras aprendan de forma independiente sin la necesidad de ser programadas directamente para ello, pero para poder aprender y resolver problemas necesitan de datos para entrenar. El aprendizaje automático permite superar la problemática de describir todos los casos y comportamientos posibles ya que se convierte en un trabajo demasiado complejo de definir con la programación clásica. Por tanto, los programas de aprendizaje automático resuelven la tarea de simplificar un modelo complejo y utilizarlo de manera efectiva.

El aprendizaje profundo toma algoritmos de aprendizaje automático para avanzar en distintos campos como el de tratamiento avanzado de imágenes (reconocimiento de caras y objetos en imágenes), traducción inteligente y asistentes de conducción autónoma. Es por eso por lo que se busca distintas formas de hardware para implementarlos.

1.1 Motivación

Actualmente se necesita de grandes sistemas de procesamiento, tiempo y energía para entrenar y desarrollar IA, pero al desplegarlo en sistemas integrados lo que se busca es una menor latencia, coste y menor consumo energético. La motivación de este trabajo es entender las soluciones de inferencia de IA sobre sistemas de FPGA gracias a distintas herramientas.

Antiguamente las FPGA no eran tan convenientes por la necesidad de entender el funcionamiento del hardware, aunque ofrecen ventajas sobre las GPU en un consumo energético menor y son óptimos para soluciones integradas como pueden ser en asistentes avanzados de conducción.
Por eso hoy en día hay empresas que ofrecen herramientas como las que ofrece Xilinx con VITIS AI que permiten una implementación más simple y estandarizada.

1.2 Objetivos

En este trabajo se busca entender cómo se puede implementar aplicaciones de aprendizaje automático sobre hardware de FPGA, entender las mejoras que ofrece en latencia, consumo eléctrico, eficiencia y realizar un caso práctico implementando una red neuronal de clasificación de imágenes sobre una placa Xilinx ZCU104.

Para eso primero se comprende el funcionamiento de las redes neuronales profundas, como realizar la aceleración sobre hardware para sistemas de FPGA y finalmente desarrollar un caso real.

1.3 Organización de la memoria

La memoria consta de cinco capítulos:

El primero el cual es introductorio con la motivación del trabajo y los objetivos a desarrollar.

El segundo punto habla sobre el estado del arte en redes neuronales y su aceleración por hardware, explicando el aspecto teórico y las arquitecturas de hardware actuales para la aceleración.

El tercero de planteamiento se comprenderá el funcionamiento de implementar una red neuronal en una FPGA mediante las herramientas que ofrece Vitis AI y su entorno.

El cuarto punto será el de desarrollo de la red neuronal sobre la tarjeta ZCU104, en el que se explica el flujo de trabajo llevado a cabo junto con los resultados obtenidos de precisión y rendimiento.

El quinto y ultimo punto se redactan las conclusiones llevadas a cabo con este trabajo y su conveniencia, junto con el posible trabajo futuro a llevar a cabo para continuar implementando soluciones de IA sobre FPGA.

2 Estado del arte

En este apartado se profundizará en la actualidad de las tecnologías de aprendizaje automático y su implementación sobre hardware. Desde el aprendizaje automático hasta distintos tipos como el aprendizaje profundo o redes neuronales centrándonos en las redes neuronales convolucionales como estándar de la visión artificial.

2.1 Aprendizaje automático

Desde la Antigüedad, el tema de las máquinas pensantes ha preocupado a la gente. Este concepto está en la base de la reflexión de lo que luego se convertirá en inteligencia artificial, así como una de sus subramas: el aprendizaje automático.

La realización de esta idea se debe principalmente a Alan Turing y su concepto de la "máquina universal" en 1936 [2], que es la base de las computadoras actuales.
Continúa sentando las bases del aprendizaje automático, su artículo sobre "Computadoras e inteligencia" en 1950, en el que desarrolla, entre otras cosas, el test de Turing [3].

El aprendizaje automático permite que un sistema controlado o asistido por computadora, como un programa, una IA o un robot, adapte sus respuestas o comportamientos a las situaciones encontradas, basándose en el análisis de datos empíricos pasados de bases de datos, sensores o web.

El AA permite superar la dificultad que radica en el hecho de que el conjunto de todos los comportamientos posibles teniendo en cuenta todas las entradas posibles se vuelve rápidamente demasiado complejo para describir y programar de manera clásica.
Por lo tanto, a los programas AA se les asigna la tarea de adaptar un modelo para simplificar esta complejidad y utilizarlo operativamente.
Idealmente, el aprendizaje tendrá como objetivo que no esté supervisado, es decir, las respuestas a los datos de entrenamiento no se proporcionan en el modelo.
Estos programas, dependiendo de su grado de sofisticación, pueden incluir capacidades para procesamiento probabilístico de datos, análisis de datos de sensores, reconocimiento, minería de datos, cálculo teórico ...

Los algoritmos de aprendizaje se pueden clasificar de acuerdo con el modo de aprendizaje que emplean.:

Aprendizaje supervisado: Si las clases están predeterminadas y los ejemplos conocidos, el sistema aprende a clasificar según una clasificación o modelo de clasificación; luego hablamos de aprendizaje supervisado (o análisis discriminante) en el cual un experto debe primero etiquetar ejemplos y el proceso se implementa en dos fases. Durante la primera fase o aprendizaje, se busca determinar un modelo a partir de los datos etiquetados. La segunda fase o prueba consiste en predecir la etiqueta de un nuevo dato, partiendo del modelo aprendido previamente. A veces es preferible asociar un dato no con una sola clase, sino con una probabilidad de pertenecer a cada una de las clases predeterminadas (esto se llama aprendizaje supervisado probabilístico).

Aprendizaje sin supervisión: Cuando el sistema o el operador solo tiene ejemplos, pero sin etiqueta, y el número de clases y su naturaleza no han sido predeterminados, hablamos de aprendizaje no supervisado o *clustering* en inglés y no se requiere ningún experto. El algoritmo debe descubrir por sí mismo la estructura más o menos oculta de los datos. El análisis de grupos es un algoritmo de aprendizaje no supervisado.
El sistema debe, en el espacio de descripción (todos los datos), apuntar los datos de acuerdo con sus atributos disponibles, para clasificarlos en grupos homogéneos de ejemplos. La similitud se calcula generalmente usando una función de distancia entre pares de ejemplos. Depende entonces del operador asociar o deducir significados para cada grupo y para los patrones de la aparición de grupos, o grupos de grupos, en su "espacio". Si el enfoque es probabilístico, es decir, en lugar de estar clasificado en una sola clase, se caracteriza por un conjunto de probabilidades de pertenencia a cada una de las clases, entonces hablamos de agrupamiento suave.

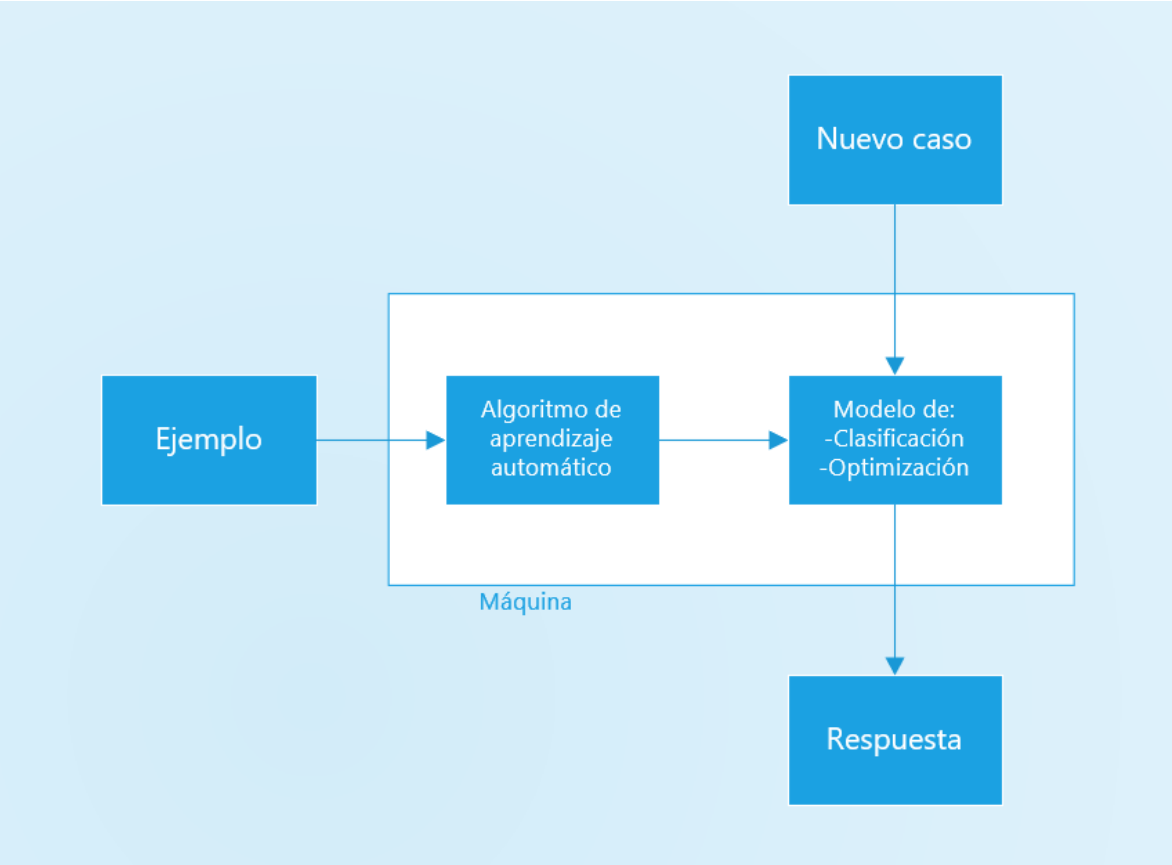


Figura 2-1. Diagrama simplificado de aprendizaje automático.

2.1.1 Aprendizaje profundo

El aprendizaje profundo es un conjunto de métodos para el aprendizaje automático que intenta modelar con un alto nivel de abstracción de datos a través de arquitecturas articuladas de diferentes transformaciones no lineales. Estas técnicas han permitido avances significativos y rápidos en los campos del análisis de la señal sonora o visual incluyendo el reconocimiento facial, el reconocimiento de voz., visión por computadora, procesamiento de lenguaje automatizado.

El aprendizaje profundo forma parte de una familia de métodos de aprendizaje automático basados en el aprendizaje a partir de modelos de datos. Una observación como puede ser una imagen puede ser representado de diferentes formas mediante un vector, una matriz o un tensor de datos. En particular para imágenes según la intensidad de los píxeles que lo componen, sus diferentes aristas, sus diferentes regiones, con formas particulares.

Ciertas representaciones y una buena capacidad de análisis automático de diferenciaciones hacen más eficiente la tarea de aprendizaje.

Una perspectiva de aprendizaje profundo de está sustituyendo las técnicas algunos trabajos todavía relativamente laboriosos, por modelos algorítmicos de aprendizaje supervisado, no supervisado (es decir, que no requieren conocimientos específicos sobre el problema estudiado) o por técnicas de extracción de características jerárquicas.

La investigación en esta área se esfuerza por construir mejores representaciones de la realidad y por crear modelos capaces de aprender estas representaciones a partir de datos sin etiquetar a gran escala. Algunas de estas representaciones están inspiradas en los últimos avances de la neurociencia. En términos generales, se trata de interpretaciones de los modelos de procesamiento de información y comunicación del sistema nervioso, como la forma en que el sistema nervioso establece conexiones en función de los mensajes recibidos, la respuesta neuronal y el peso de las conexiones entre las neuronas del cerebro.

2.2 Redes Neuronales Profundas (DNN)

En 1943, el neurofisiólogo Warren McCulloch y el matemático Walter Pitts publicaron un artículo que describía cómo funcionan las neuronas representándolas mediante circuitos eléctricos [4]. Luego formaron un modelo simplificado de una neurona biológica comúnmente conocida como neurona formal. Demostraron que las redes neuronales formales simples teóricamente pueden realizar funciones lógicas, aritméticas y simbólicas complejas. La neurona formal se concibe como un autómata dotado de una función de transferencia que transforma sus entradas en salidas según reglas precisas. Por ejemplo, una neurona suma sus entradas, compara la suma resultante con un valor umbral y responde emitiendo una señal si esta suma es mayor o igual a este umbral.

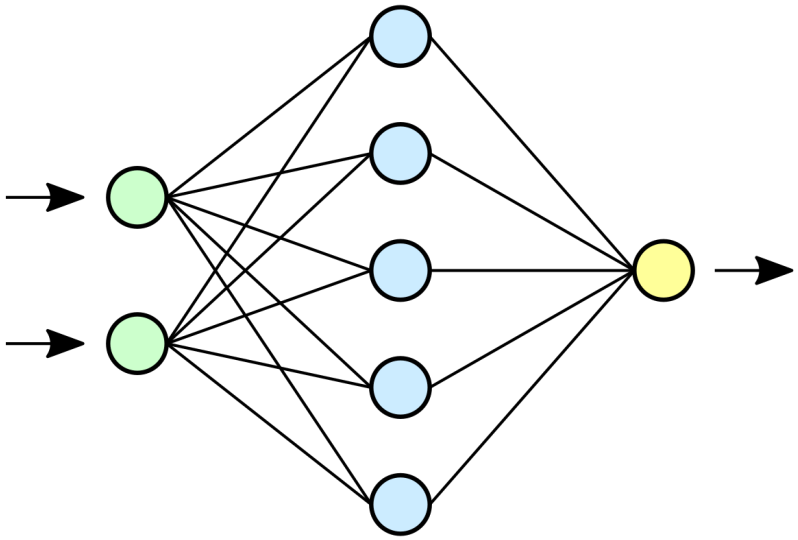


Figura 2-2: Esquema simplificado de una red neuronal.

Aun así, el trabajo de McCulloch y Pitts no dio ninguna indicación sobre un método para adaptar los coeficientes sinápticos de las neuronas y fue mas adelante gracias al físico fisiólogo canadiense Donald Hebb sobre el aprendizaje en 1949 descrito en su libro "*The Organization of Behavior*" [5]. Hebb propuso una regla simple que permite modificar el valor de los coeficientes sinápticos según la actividad de las unidades que conectan. Esta regla ahora conocida como la "regla de Hebb" está presente en casi todas partes en los modelos actuales, incluso en los más sofisticados. A partir de este artículo, la idea se afianzó con el tiempo en la mente y germinó en la mente de Frank Rosenblatt en 1957 con el modelo del perceptrón. Es el primer sistema artificial capaz de aprender de la experiencia, incluso cuando su instructor comete algunos errores (lo cual es claramente diferente de un sistema de aprendizaje lógico formal).

En 1982 , John Joseph Hopfield , físico de renombre, dio nueva vida a lo neuronal al publicar un artículo en el que presentaba un nuevo modelo de red neuronal (completamente recurrente) [6]. Este artículo tuvo éxito por varias razones, la principal de las cuales fue teñir la teoría de las redes neuronales con el rigor propio de los físicos. El neural volvió a ser un tema de estudio aceptable, aunque el modelo de Hopfield adolecía de las principales limitaciones de los modelos de la década de 1960, en particular la incapacidad para tratar problemas no lineales.
Las redes neuronales posteriormente experimentaron un auge considerable, y fueron de los primeros sistemas de beneficiarse de la comprensión de la teoría de la "regularización estadística" introducido por Vladimir Vapnik en la Unión Soviética y popularizado en occidente desde la caída de la Unión Soviética[7]. Esta teoría, una de las más importantes en el campo de la estadística, permite anticipar, estudiar y regular los fenómenos vinculados al sobreajuste.

De esta manera las redes neuronales fueron evolucionando con un nuevo tipo de red que no está completamente conectada, para aligerar los modelos en cuanto al número de parámetros, y mejorar el rendimiento y su capacidad de generalización, llegando hasta la actualidad con modelos de redes neuronales convolucionales las cuales tienen grandes aplicaciones en reconocimiento de imágenes y video, sistemas de sugerencias y procesamiento de la lengua.

Las redes neuronales generalmente se optimizan mediante métodos de aprendizaje probabilísticos, especialmente bayesianos. Se ubican en la serie de aplicaciones estadísticas, por un lado, lo que enriquece un conjunto de paradigmas que permiten la creación de clasificación rápida, y por otro lado en la serie de métodos de inteligencia artificial, brindan un mecanismo para percibir de forma independiente las propias ideas del implementador, y proporcionar información para el razonamiento lógico formal.

Una red neuronal generalmente se compone de una sucesión de capas, cada una de las cuales toma sus entradas de las salidas de la anterior. Cada capa (j) está compuesta por N_j neuronas, que toman sus entradas de las N_{j-1} neuronas de la capa anterior. Cada sinapsis tiene un peso sináptico asociado, de modo que el N_{j-1} se multiplican por este peso, luego se suman por las neuronas de nivel j, lo que equivale a multiplicar el vector de entrada por una matriz de transformación. Poner las diferentes capas de una red neuronal una detrás de la otra equivaldría a poner en cascada varias matrices de transformación y podría reducirse a una sola matriz, producto de las otras, si no hubiera en cada capa, la función de salida que introduce una no linealidad.

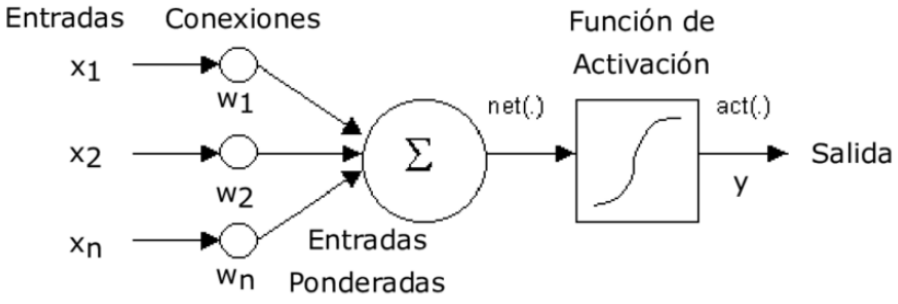


Figura 2-3: Estructura de una neurona artificial o neurona formal.

La neurona calcula la suma de sus entradas x, ponderadas por los pesos w, luego este valor pasa por la función de activación para producir su salida

Más allá de esta estructura simple, la red neuronal también puede contener bucles que cambian radicalmente sus posibilidades, pero también su complejidad. Así

como los bucles pueden transformar la lógica combinatoria en lógica secuencial, los bucles en una red neuronal transforman un dispositivo de reconocimiento de entrada simple en una máquina compleja capaz de todo tipo de comportamientos modificando fundamentalmente toda la red.

2.2.1 Red Neuronal Convolucional(CNN)

En el aprendizaje automático, la red neuronal convolucional (CNN) es una red neuronal artificial acíclica (*feedforward*) en la que el patrón de conexión entre las neuronas se inspira en la corteza visual animal. Al colocar en mosaico el campo de visión, las neuronas en esa área del cerebro se organizan para que se correspondan con el área superpuesta. Su funcionamiento está inspirado en procesos biológicos, están compuestos por perceptrones multicapa cuya finalidad es pre procesar una pequeña cantidad de información. Las redes neuronales convolucionales tienen una amplia gama de aplicaciones en el reconocimiento de imágenes y video, sistemas de recomendación y procesamiento del lenguaje natural.

Una red neuronal convolucional está formada por dos tipos de neuronas artificiales, dispuestas en capas que procesan información sucesivamente. Las primeras las neuronas de procesamiento, que procesan una parte limitada de la imagen (denominada "campo receptivo") a través de una función de convolución y a continuación las llamadas puesta en común de las neuronas de salida. Se puede aplicar un tratamiento correctivo no lineal y puntual entre cada capa para mejorar la relevancia del resultado.
Todas las salidas de una capa de procesamiento permiten reconstituir una imagen intermedia, que servirá de base para la siguiente capa.

Una arquitectura de red neuronal convolucional está formada por distintas capas de procesamiento:

Capa de convolución: Para esta capa se utilizan 3 hiperparámetros para dimensionar el volumen de la capa de convolución: profundidad de capa, paso y margen. Si el paso y el margen son aplicados a la imagen de entrada permiten controlar el número de campos receptores a gestionar (superficie de tratamiento), la profundidad permite tener una noción de volumen de salida, y de la misma forma que una imagen, puede tener un volumen, si tomamos una profundidad de 3 para los tres canales RGB de una imagen en color, la capa de convolución también presentará la misma profundidad en la salida. Es por eso que es más útil hablar de volumen entrada y salida ya que la salida de una capa convolucional puede ser la entrada de otra.

Capa de agrupación (*Pooling*): Un concepto importante de las CNN es la agrupación, que es una forma de reducción de la resolución de las imágenes. La imagen de entrada se corta en una serie de rectángulos con n píxeles en cada lado que no se superponen. Cada rectángulo puede verse como un mosaico. La señal en la salida del mosaico se define en función de los valores tomados por los diferentes píxeles del mosaico. Normalmente se toman valores de mosaico pequeño estilo 2x2 y hay dos tipos de agrupaciones la *max pooling*, la cual toma el valor máximo de cada agrupación de neuronas, y la *average pooling* que toma el valor medio. La agrupación reduce el tamaño espacial de una imagen intermedia, lo que reduce la cantidad de parámetros y cálculos en la red. Por lo tanto, es común insertar periódicamente una capa de agrupación entre dos capas convolucionales sucesivas de una arquitectura de red neuronal convolucional para reducir el sobre aprendizaje.

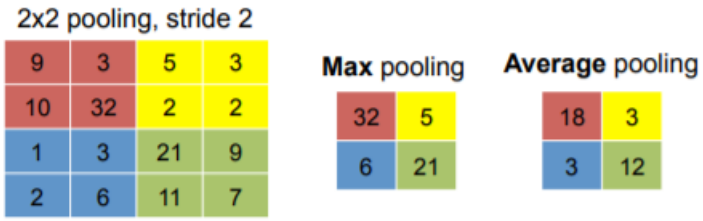


Figura 2-4: Las dos formas de pooling, max y average.

Figura extraída de [8]

Capas de corrección (funciones de activación): A menudo, es posible mejorar la eficiencia del procesamiento interponiendo entre las capas de procesamiento una capa que realizará una función no lineal (función de activación) en las señales de salida. Pueden ser corrección tipo ReLU (*Rectified linear unit*), tangente hiperbólica, hiperbólica de saturación, función sigmoidea, etc. A menudo, la corrección de Relu es preferible, ya que da como resultado la formación de la red neuronal varias veces más rápida [1], sin hacer una diferencia significativa en la precisión de la generalización.

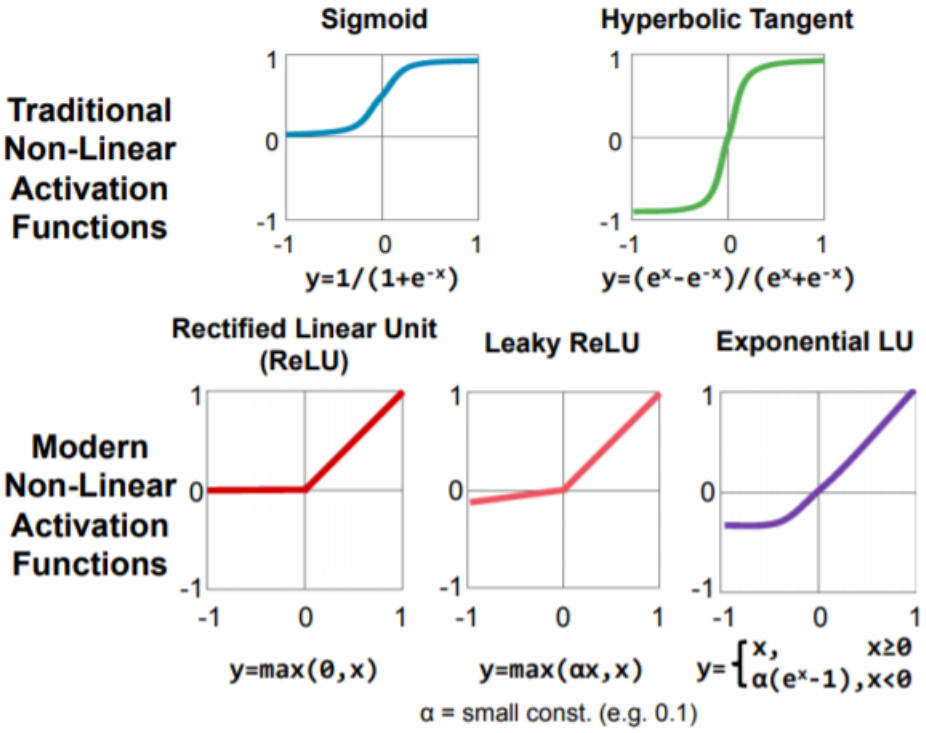


Figura 2-5: Diferentes funciones de activación para capas de corrección.

Figura extraída de [8]

Capa completamente conectada (*Fully Conected Layers*): Después de varias capas de convolución y agrupación máxima, el razonamiento de alto nivel en la red neuronal tiene lugar a través de capas completamente conectadas. Las neuronas en una capa completamente conectada tienen conexiones a todas las salidas de la capa anterior como se ve normalmente en las redes neuronales regulares.

Capa de pérdida (*LOSS*): capa de pérdida especifica cómo el arrastre de la red penaliza la brecha entre la señal esperada y real. Normalmente es la última capa de la red. Allí se pueden utilizar varias funciones de pérdida adecuadas para diferentes tareas.

2.3 Entrenamiento e Inferencia

Dado que las redes neuronales convolucionales son una clase de algoritmos de aprendizaje automático, el programa básico no cambia a medida que aprende a realizar sus tareas asignadas. En el caso específico de las redes neuronales convolucionales el aprendizaje implica determinar el valor de los pesos (y sesgos) de las conexiones

neuronales en la red, y se denomina entrenamiento de la red. Una vez entrenado, el programa puede realizar su tarea calculando la salida de la red utilizando los pesos determinados durante el proceso de entrenamiento. Ejecutando el programa con estos pesos se denominan inferencia.

2.4 Despliegue de redes neuronales en hardware

Hoy en día el estándar de implementación hardware de aprendizaje automático son las GPU (*Graphics Processing Units*), aunque en este trabajo nos centraremos en la implementación con FPGA conviene saber sobre la arquitectura de las GPU para comprender la implementación.

Librerías populares sobre aprendizaje automático como Tensorflow funcionan mediante CUDA (*Compute Unified Device Architecture*) para procesar los datos dentro de la GPU aprovechando su capacidad de procesamiento en paralelo. Este proceso se le conoce como GPGPU (*General Purpose GPU*) y ha sido adaptado para modelos de aprendizaje profundo debido a los miles de cálculos aritméticos necesarios para ejecutarse.

Dentro de una red convolucional profunda, como muestra la siguiente imagen, regiones de pixeles pasas a través de filtros mientras se genera una suma ponderada por iteración. Esto se repite miles de veces por filtro en cada capa de la red, por eso que los modelos de redes convolucionales profundas tienen una alta carga computacional.

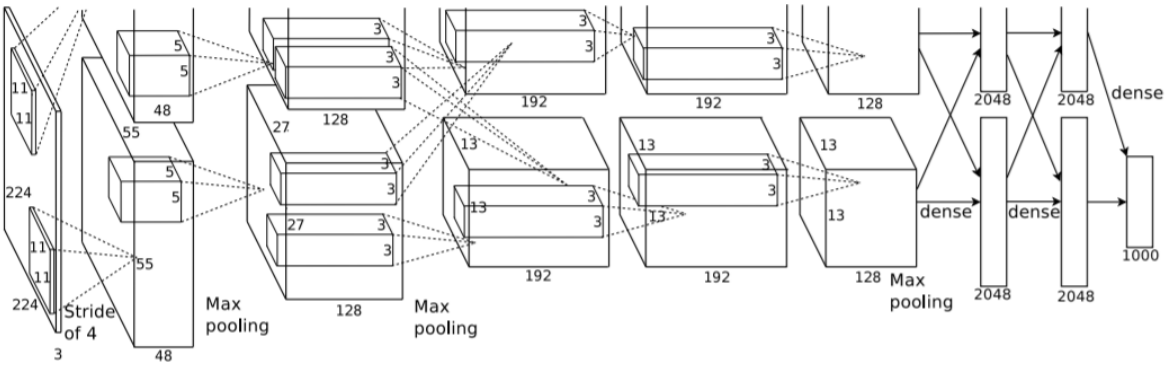


Figura 2-6: Arquitectura de CNN ejecutada por una GPU por capas.

La entrada de la red es de 150.528 dimensiones y la cantidad de neuronas en las capas restantes de la red viene dada por 253,440–186,624–64,896–64,896–43,264–4096–4096–1000. Figura extraída de [1].

La facilidad que ofrecen librerías como Tensor Flow es que se trabaja sobre CUDA, es que el desarrollador no necesite implementar el código en paralelo ni comprender la arquitectura de su chip haciéndolo muy conveniente y fácil de implementar.

Antiguamente las FPGA no eran tan convenientes por la necesidad de entender el funcionamiento del hardware, aunque ofrecen ventajas sobre las GPU en un consumo energético menor y son óptimos para soluciones integradas como pueden ser en asistentes avanzados de conducción.
Por eso hoy en día hay herramientas como las que ofrece Xilinx con VITIS AI que permiten una implementación más simple y estandarizada.

2.4.1 Implementación sobre GPU y CPU

Las CPU (unidades centrales de procesamiento) están diseñadas para operaciones en serie y admiten lógica avanzada. Esto se refleja en su diseño, que contiene menos núcleos y más memoria caché para obtener rápidamente instrucciones complejas. Sin embargo, las GPU (Unidad de procesamiento gráfico) tienen cientos de núcleos más pequeños para un cálculo simple y, por lo tanto, un rendimiento más alto en comparación con las CPU.

Figura 2-7: Diferencia de arquitectura entre GPU y CPU.

Figura extraída de [12].

Un problema frecuente en las redes neuronales son las altas demandas de potencia computacional de ejecutar algoritmos. Los desarrolladores han recurrido al concepto de computación paralela para superar estas dificultades. Sin embargo, los

enfoques tradicionales basados en CPU pueden no ser la mejor solución para realizar computación en paralelo debido a sus problemas de costo y escalabilidad. La GPU es hasta ahora una de las mejores soluciones rentables en esta era para enfrentar este problema con la tecnología de computación paralela masiva. Las actuales GPU de NVIDIA contienen cientos de núcleos que son capaces de ejecutar miles de subprocesos livianos en lugar de unos pocos subprocesos disponibles en la CPU actual. La diferencia de arquitectura entre CPU y GPU se puede identificar simplemente mirando la Fig. 2-7. Una CPU normal tiene varias ALU (unidades lógicas aritméticas), una unidad de control para controlar esas ALU, memoria caché rápida y DRAM (memoria dinámica de acceso aleatorio) cuando se considera una GPU que tiene cientos de ALU, varias unidades de control, varias memorias caché y una DRACMA. Estos cientos de núcleos se agrupan en varios multiprocesadores para su fácil manejo.

2.4.2 Arquitecturas principales de aprendizaje profundo

A la hora de elegir el hardware adecuado para nuestra IA debemos de pensar en el objetivo del sistema, ya sea entrenamiento de redes neuronales o aceleración de la inferencia y también si va a estar situado en servidores (*Cloud*) o va a ser implementado en dispositivos *Edge*, móviles o sistemas integrados en vehículos.
Dependiendo de los diferentes objetivos que se busquen se utilizaran distintas arquitecturas para acelerar la implementación de IA.

Cloud-Servidores: Utilizan grandes GPU de NVIDIA, con un gran rendimiento para el entrenamiento de redes neuronales. Por ejemplo, la NVIDIA A100[15].
Dispositivos móviles: Se busca bajo consumo para desempeñar funciones de inferencia. Por ejemplo: Qualcomm Hexagon, Samsung, Apple Core-ML,Huawei Da Vinci
FPGA: Muy útiles por la baja latencia y bajo consumo, se integran sobre otros hardware para aplicaciones inferencia. EjXilinx DPU, Intel OpenVino + DLDT, Lattice sensAI
Edge: Múltiples opciones desde sistemas de chips integrados hasta sistemas de consumo mínimo de energía.

3 Planteamiento

Para entender el trabajo hace falta plantear las bases y objetivos de implementar una red neuronal, como funciona e implementarlo sobre un hardware especifico. Por eso se describe la herramienta que nos permitirá llevar a cabo el trabajo, Vitis AI y los sistemas de los que dispone y como implementarlos.

3.1 Visión artificial mediante CNN

La visión artificial (*Computer Vision*) es un campo interdisciplinario de la ciencia que tiene como objetivo hacer que las computadoras procesen, analicen imágenes y videos y extraigan detalles de la misma manera que lo hace una mente humana.
A diario no es perceptible, pero estamos siendo asistidos por las aplicaciones de la visión por computadora en los servicios automotrices, minoristas, bancarios y financieros, atención médica, etc. Las redes neuronales profundas (DNN) tienen mayores capacidades para el reconocimiento de patrones de imágenes y se utilizan ampliamente en algoritmos de visión artificial. Y, la red neuronal convolucional (CNN) es una clase de DNN que se aplica más comúnmente para analizar imágenes visuales. Se utiliza no solo en visión artificial, sino también para la clasificación de textos en el procesamiento del lenguaje natural.

La mayoría de las tareas de visión artificial están creadas en arquitecturas de CNN, ya que la base de la mayoría de los problemas es clasificar una imagen en etiquetas conocidas. Los algoritmos para la detección de objetos como SSD (detección de múltiples cajas de disparo único) y YOLO (Solo mira una vez) también se construyen alrededor de CNN.

Las redes neuronales artificiales (ANN) son excelentes herramientas para el aprendizaje automático, pero para procesar imágenes con capas ocultas completamente conectadas, las ANN tardan mucho en ser entrenadas. Debido a esto, se utilizó CNN para reducir primero el tamaño de las redes utilizando capas convolucionales y capas agrupadas, y luego alimentar los datos reducidos a capas completamente conectadas.

Hoy en día, la mayoría los algoritmos de visión artificial, son implementados en librerías específicas, es decir, no es necesario desarrollar un algoritmo para utilizarlo. Es por esto por lo que un desarrollador de sistemas de visión artificial se debe centrar en utilizar las funcionalidades que ofrecen las distintas librerías para implementar de forma óptima y adecuada a cada proyecto. Existen distintas librerías centradas solo en visión artificial como puede ser OpenCV, siendo la más conocida y gratuita, y estando disponible en C++ y Python. Aun así, en este trabajo nos centraremos en las librerías más

especializadas en aprendizaje profundo, como son Caffe, Pytorch o Tensorflow (desarrollada por Google en C++ y Python).

Figura 3-1: Librerías de aprendizaje automático.

3.2 VITIS AI

Esta herramienta de Xilinx se presenta como una solución para la inferencia de inteligencia artificial óptima desde el *edge* hasta la nube. El entorno de desarrollo de Vitis AI acelera la inferencia de AI en las plataformas de hardware Xilinx, incluidos los dispositivos Edge y las tarjetas aceleradoras Alveo. Consiste en núcleos IP optimizados, herramientas, bibliotecas, modelos y diseños de ejemplo. Está diseñado teniendo en cuenta una alta eficiencia y facilidad de uso para liberar todo el potencial de la aceleración de IA en FPGA de Xilinx y en plataformas de aceleración de cómputo adaptativas (ACAP). Facilita a los usuarios sin conocimientos de FPGA el desarrollo de aplicaciones de inferencia de aprendizaje profundo, al abstraer las complejidades de la FPGA y ACAP subyacentes lo cual resulta muy conveniente para nuestro trabajo.

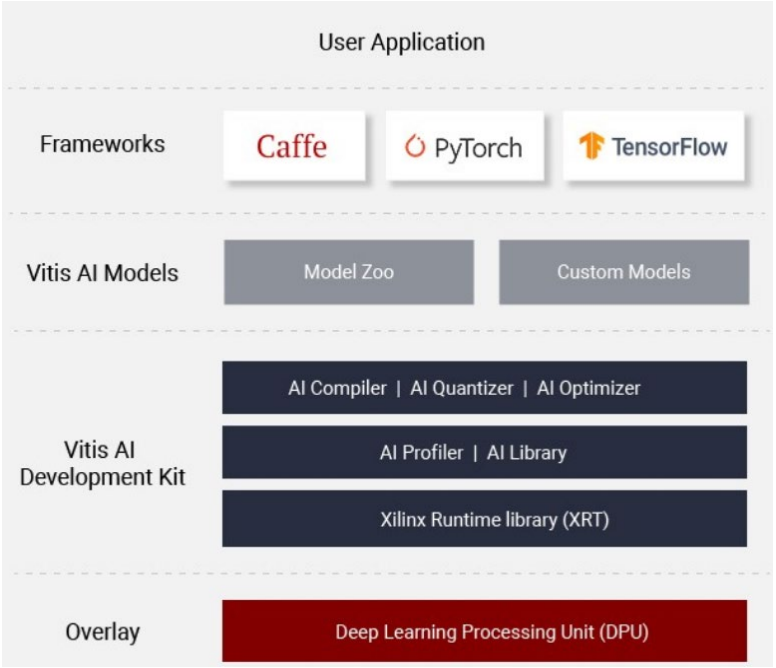


Figura 3-2. Stack de Vitis AI desde nivel de usuario hasta el despliegue.

Figura extraída de [11].

La plataforma de software Vitis tiene una arquitectura en capas, de esta forma ayuda a los ingenieros a centrarse en su propio objetivo, pero es fácil intercambiar resultados de trabajo. Por ejemplo, los *Data scientist* o los ingenieros de aprendizaje automático solo necesitan trabajar en los entornos de desarrollo específicos del dominio. Después el resultado se puede convertir al formato de Vitis-AI para una mayor implementación. Los desarrolladores de aplicaciones toman el modelo de aprendizaje automático de los *Data scientist* y lo adoptan a las bibliotecas aceleradas proporcionadas por Xilinx, al tiempo de ejecución y la arquitectura en capas de plataforma garantizando que la API de servicio sea estable y brinde soporte multiplataforma. De esta forma la carga de trabajo para habilitar una nueva plataforma en hardware sería mínima.

Vitis AI incluye los siguientes componentes clave:

- AI Model Zoo: un conjunto completo de modelos optimizados previamente que están listos para implementarse en dispositivos Xilinx.
- AI Optimizer: un optimizador de modelo opcional que puede podar un modelo hasta en un 90%. Con esta herramienta se reduce desde un x5 hasta un x50 teniendo un impacto mínimo en la precisión.

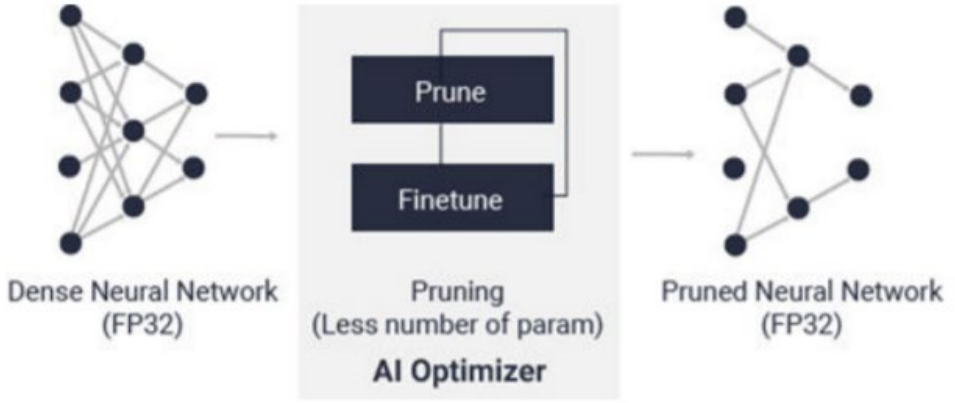


Figura 3-3: Optimizador de Vitis AI.

Figura extraída de [11].

- AI Quantizer: un cuantificador potente que admite la cuantificación, calibración y ajuste fino de modelos. Al convertir los pesos y activaciones de punto flotante de 32 bits en puntos fijos como INT8, el cuantificador de IA puede reducir la complejidad informática sin perder precisión de predicción. El modelo de red de punto fijo requiere menos ancho de banda de memoria, lo que proporciona una velocidad más rápida y una mayor eficiencia energética que el modelo de punto flotante

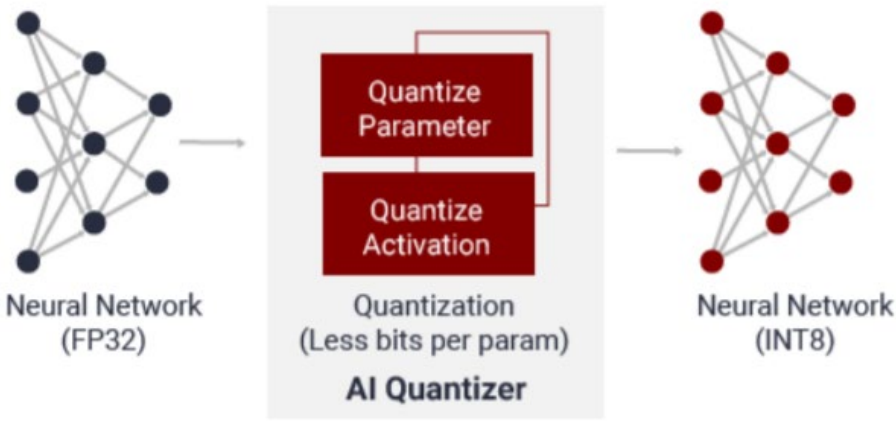


Figura 3-4. Cuantizador de Vitis AI.

Figura extraída de [11].

- Compilador AI: compile el modelo cuantificado en un conjunto de instrucciones y un flujo de datos de alta eficiencia.

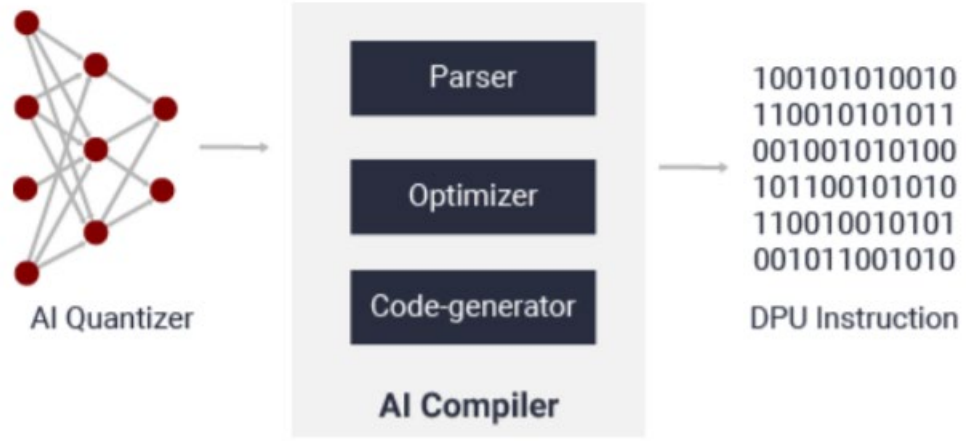


Figura 3-5. Compilador de Vitis AI.

Figura extraída de [11].

- AI Profiler: realice un análisis en profundidad de la eficiencia y la utilización de la implementación de la inferencia de AI.
- Biblioteca de IA: ofrece API de C ++ de alto nivel, pero optimizadas para aplicaciones de IA desde el borde hasta la nube.
- DPU: los núcleos IP escalables y eficientes se pueden personalizar para satisfacer las necesidades de muchas aplicaciones diferentes. Se explica con profundidad en el punto 3.3.

Para habilitar la aceleración de hardware, el código se divide entre un host y uno o varios núcleos. El programa host, escrito en C / C ++ utiliza abstracciones API, se

ejecuta en un procesador host (un servidor x86 o un procesador ARM en el caso de plataformas integradas). Los núcleos acelerados por hardware se ejecutan en la lógica programable de un dispositivo Xilinx.

Las llamadas a la API, procesadas por el tiempo de ejecución de Xilinx (XRT), administran las transacciones de proceso entre el programa host y los aceleradores de hardware (*kernels*). La comunicación entre el host y el (los) Kernel (s), incluido el control y las transferencias de datos, se produce a través del bus PCIe o un bus AXI para plataformas integradas.

En una aplicación típica, el host primero transfiere los datos para ser operados por el Kernel desde la memoria del host a la memoria global. Posteriormente, el Kernel opera con los datos y almacena los resultados en la memoria global. Una vez completado el Kernel, el host transfiere los resultados a la memoria del host. Las transferencias de datos entre el host y la memoria global introducen latencia, lo que puede resultar costoso para la aplicación en general. Para lograr la aceleración en un sistema real, los beneficios logrados por los núcleos de aceleración de hardware deben superar la latencia adicional de estas transferencias de datos.
El flujo de trabajo que se realiza en se puede simplificar en tres pasos como se ve en la figura 3-6.

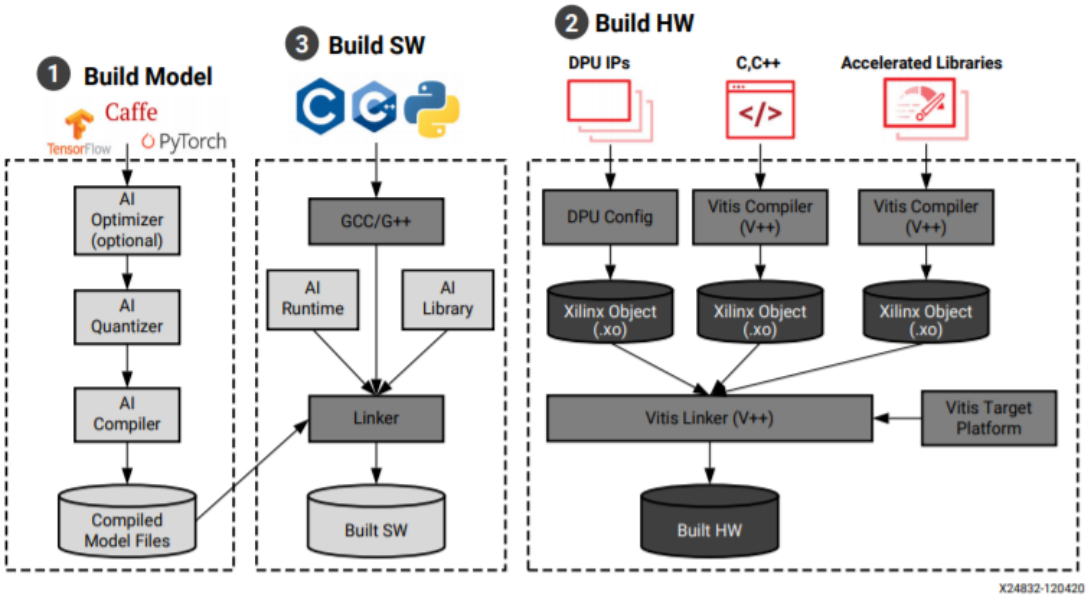


Figura 3-6: Flujo de trabajo en Vitis AI.

Figura extraída de [11].

Primero la *toolchain* de Vitis AI en la máquina host se utiliza para construir el modelo. Toma los modelos flotantes previamente entrenados como entrada y los ejecuta a través del Optimizador de IA (opcional). A continuación, se construye una plataforma de hardware personalizada utilizando la plataforma de software Vitis. El hardware generado incluye la DPU IP y otros núcleos. En el paquete de lanzamiento de Vitis AI, se incluyen imágenes de tarjetas SD predefinidas como para nuestro caso la ZCU104 para un inicio rápido y desarrollo de aplicaciones. Por ultimo se crea el software ejecutable que se ejecute en el hardware integrado. Las aplicaciones pueden ser escritas con C ++ o Python el cual hará llamadas a Vitis AI Runtime y Vitis AI Library para cargar y ejecutar los archivos de modelo compilados.

3.3 Xilinx DPU

La Unidad de Procesador de Aprendizaje Profundo (DPU) de Xilinx es un motor de cálculo configurable dedicado para redes neuronales convolucionales. El grado de paralelismo utilizado en el motor es un parámetro de diseño y una aplicación. Incluye un conjunto de instrucciones altamente optimizadas y es compatible con la mayoría de las redes neuronales convolucionales, como VGG, ResNet, GoogleNet, YOLO, SSD, MobileNet, FPN y otras.

La DPU IP se implementa en la lógica programable (PL) del Zynq® UltraScale + MPSoC con conexiones directas al sistema de procesamiento (PS). La DPU requiere instrucciones para implementar una red neuronal y ubicaciones de memoria accesibles para imágenes de entrada, así como datos temporales y de salida. También se requiere que un programa se ejecute al mismo tiempo en la unidad de procesamiento de aplicaciones (APU) para atender las interrupciones y coordinar las transferencias de datos.

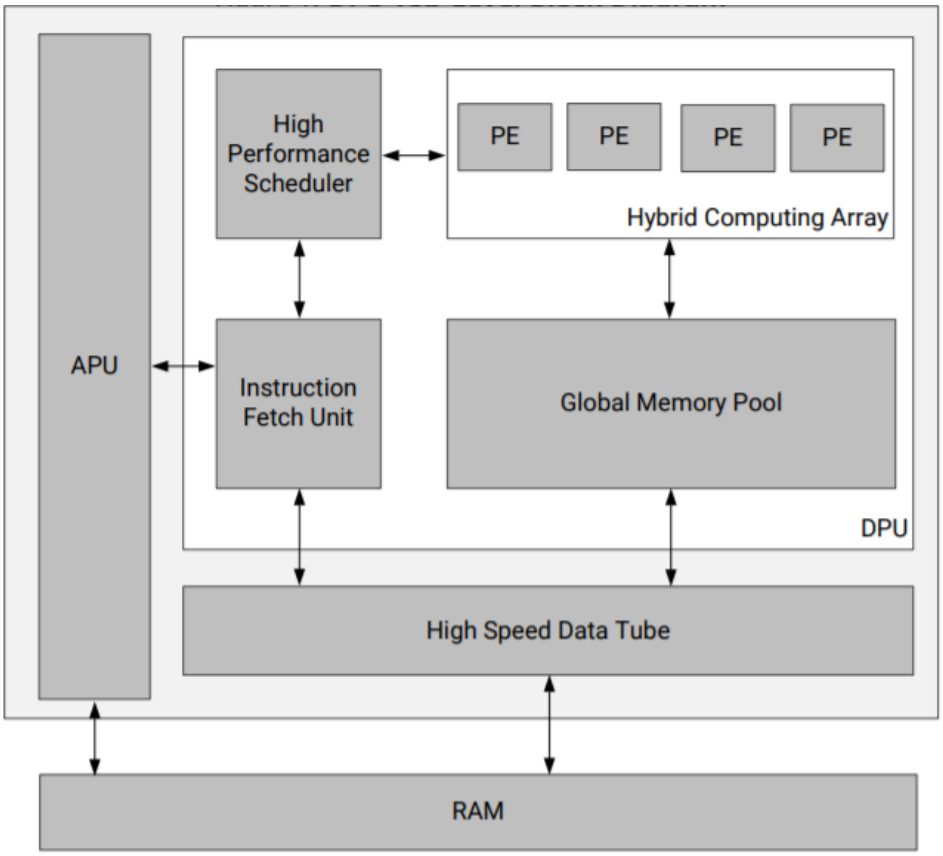


Figura 3-7: Diagrama de bloques de alto nivel de DPU.

Figura extraída de [10].

Las características más destacables de esta unidad son la interfaz AXI(*Advanced eXtensible Interface*) esclava para acceder a los registros y configuración y otro AXI maestro para acceder a las instrucciones, siendo configurable el AXI con 64 o 128 bits de acceso a datos dependiendo del dispositivo. Admite también la configuración individual de cada canal y la generación de solicitudes de interrupción opcional. Algunos aspectos principales de la funcionalidad DPU pueden ser la arquitectura de hardware configurable (incluye B512, B800, B1024, B1152, B1600, B2304, B3136 y B4096), máximo de tres núcleos, convolución y desconvolución, convolucion en profundidad, *max poolling y average poolling*, ReLU, RELU6 y leaky ReLU, Concat, Suma de elementos, Dilatación, Reorg, Capa completamente conectada, Softmax, Normalización de Bach y Separar.

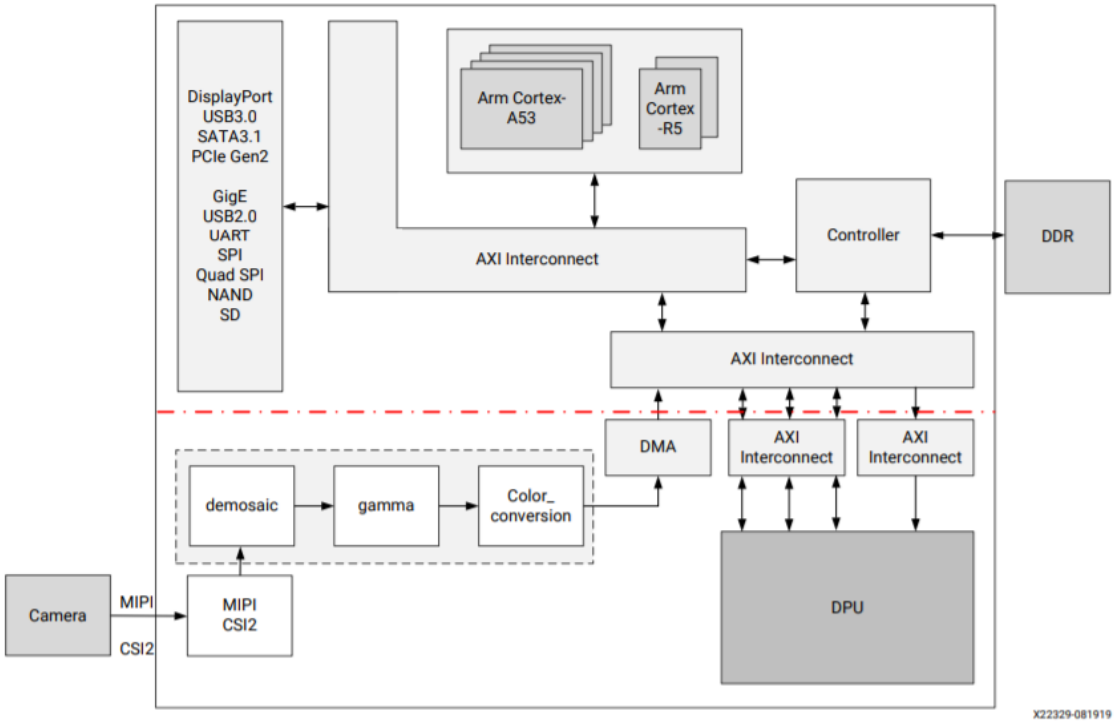


Figura 3-8: Diagrama de bloques del sistema con Xilinx UltraScale + MPSoC

La DPU está integrada en el sistema a través de una interconexión AXI para realizar tareas de inferencia de aprendizaje profundo como clasificación de imágenes, detección de objetos y segmentación semántica. Figura extraída de [10].

La arquitectura de hardware detallada de la DPU se muestra en la figura 3.3. Después de la puesta en marcha, la DPU obtiene instrucciones de la memoria fuera del chip para controlar el funcionamiento del motor informático (*computing engine*). Las instrucciones son generadas por el DNNC donde se han realizado optimizaciones sustanciales. La memoria en el chip se usa para almacenar datos de entrada, intermedios y de salida para lograr un alto rendimiento y eficiencia. Los datos se reutilizan tanto como sea posible para reducir el ancho de banda de la memoria. Se utiliza un diseño de *pipeline* profundo para el motor informático. Los elementos de procesamiento (PE) aprovechan al máximo los bloques de construcción de grano fino, como multiplicadores, sumadores y acumuladores en los dispositivos Xilinx.

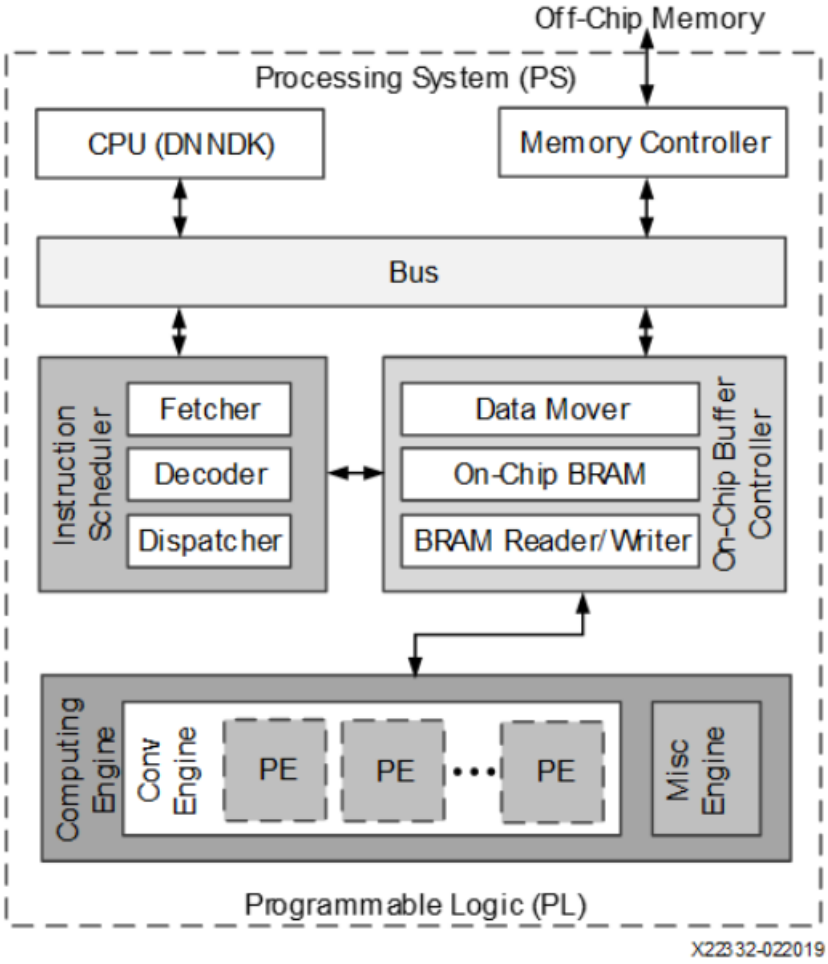


Figura 3-9: Arquitectura hardware del Xilinx DPU.

Figura extraída de [10].

Hay tres dimensiones de paralelismo en la arquitectura de convolución DPU: paralelismo de píxeles, paralelismo de canal de entrada y paralelismo de canal de salida. El paralelismo del canal de entrada es siempre igual al paralelismo del canal de salida. Las diferentes arquitecturas requieren diferentes recursos lógicos programables. Las arquitecturas más grandes pueden lograr un mayor rendimiento con más recursos.

4 Desarrollo, implementación y resultados

Este apartado demuestra el proceso de diseño de Vitis AI con librerías Tensorflow y se ilustra como una llegar desde la descripción del modelo en Python a un modelo compilado para Xilinx DPU.

Todo el código de la aplicación está escrito en Python y utilizada las API unificadas y el VART en la versión de VITIS AI 1.3. La figura 4-1 muestra el flujo de trabajo de capitulo.

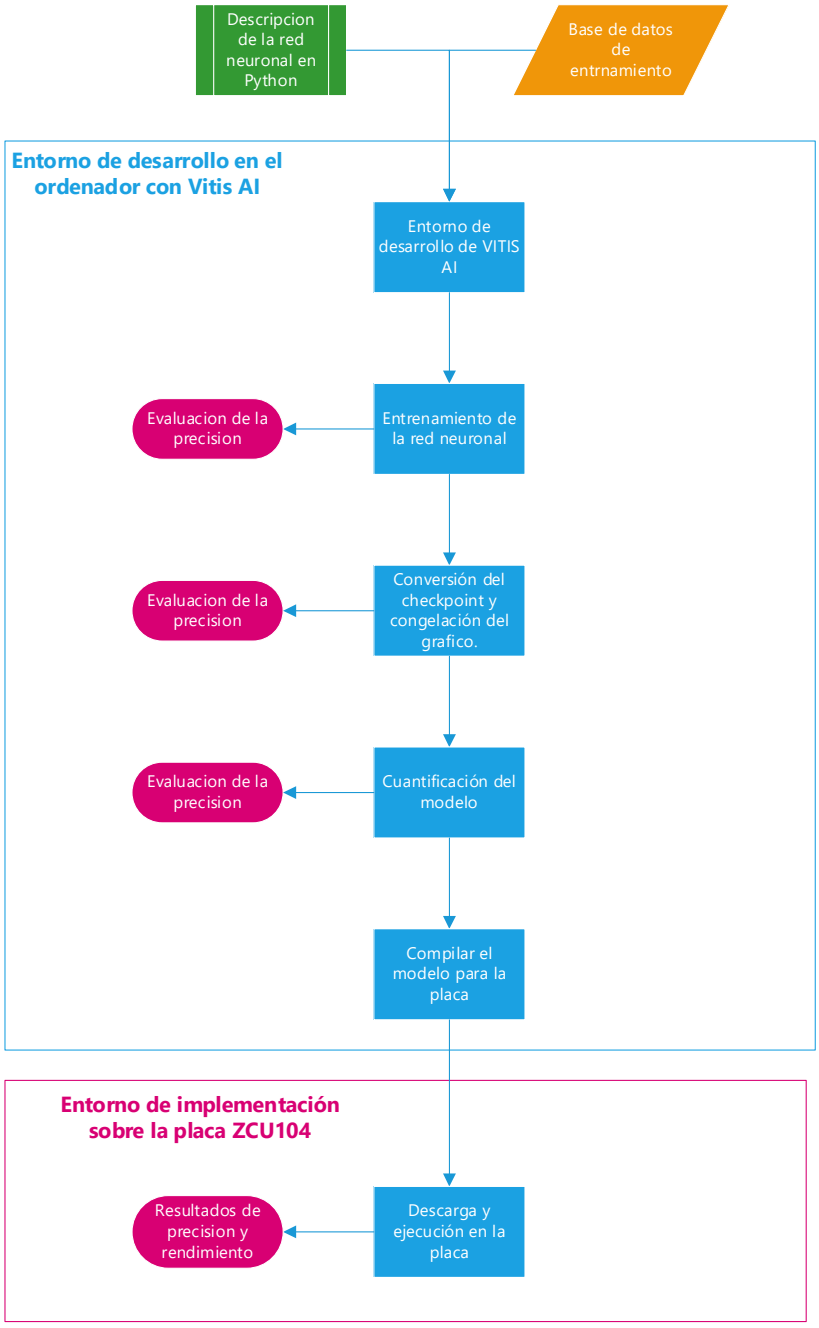


Figura 4-1. Diagrama de bloques del flujo de trabajo.

4.1 Conjunto de datos CIFAR-10

CIFAR-10 es un conjunto de datos(*dataset*) disponible públicamente que contiene un total de 60k imágenes RGB, cada una de las cuales tiene 32 píxeles x 32 píxeles x 8 bits por canal de color. El pequeño tamaño de imagen de 32 x 32 significa que no es muy útil para aplicaciones del mundo real, pero el conjunto de datos CIFAR-10 es un buen punto de partida para estudiar el aprendizaje automático. El conjunto de datos completo de imágenes de 60k normalmente se divide en imágenes de 50k para entrenamiento e imágenes de 10k para prueba / validación. Hay un total de 10 clases como se puede ver en la tabla 4-1.

Índice de clase	0	1	2	3	4	5	6	7	8	9
Nombre de clase	Avión	Coche	Pájaro	Gato	Ciervo	Perro	Rana	Caballo	Barco	Camión

Tabla 4-1: Clases e índices del CIFAR-10 dataset.

4.2 DenseNetX CNN

Trabajos recientes han demostrado que las redes convolucionales consiguen ser mucho más profundas, eficientes y precisas de entrenar si tienen conexiones más próximas entre capas cercanas a la entrada y cercanos a la salida. Y es por eso que en Huang presento por primera vez esta arquitectura en su artículo *'Densely Connected Convolutional Networks'*[9].

En este punto, se introduce la Red convolucional densa (DenseNet), que conecta cada capa a todas las demás capas en forma *feed-forward*. A diferencia de las redes convolucionales clásicas que tienen capas con L conexiones, una entre cada capa y su siguiente capa, la red DenseNet tiene L (L + 1) / 2 conexiones directas. Para cada capa, los mapas de características de todas las capas anteriores se utilizan como entradas, y sus propios mapas de características se utilizan como entradas en todas las capas posteriores. DenseNet tiene muchas ventajas destacables: alivia el problema del gradiente de desaparición, fortalecen la propagación de características, promueven la reutilización de características y disminuyen sustancialmente la cantidad de parámetros necesarios. Es por eso por lo que presenta mejores resultados que redes anteriores, reduciendo el coste computacional y obteniendo mejores resultados.

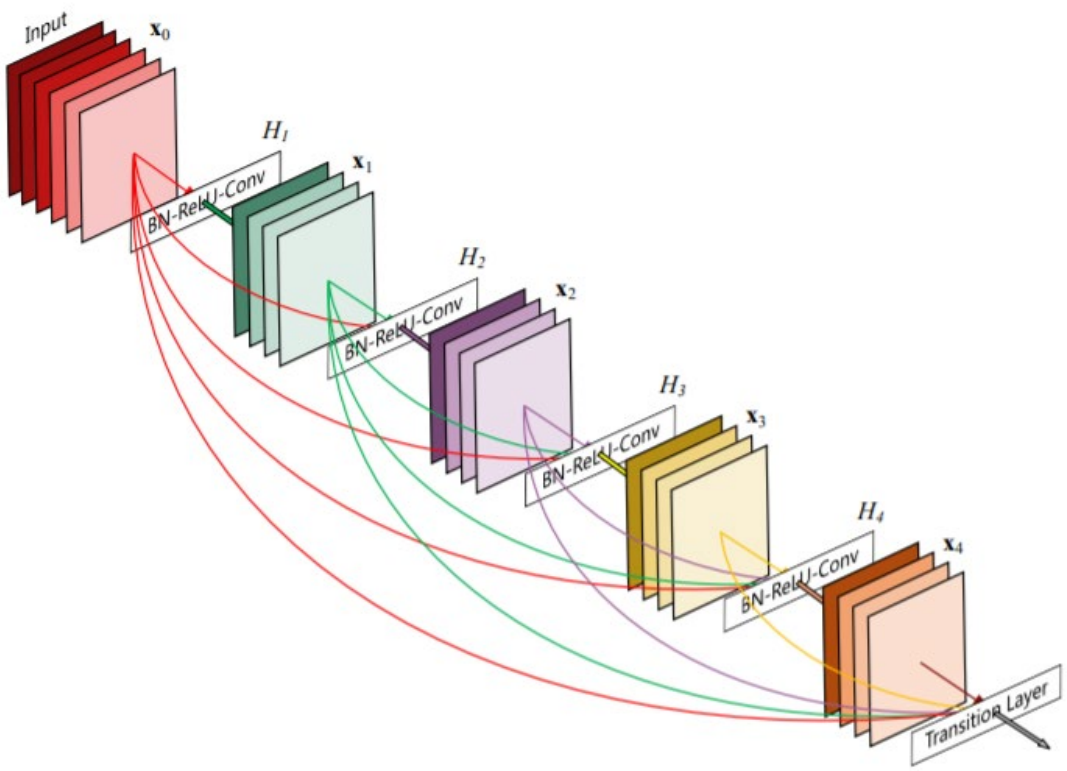


Figura 4-2: Un bloque de Densenet de 5 capas en la cual todas las capas toman el mapa de características de sus anteriores.

Figura extraída de [9].

El código de DenseNet del que parto esta implementado en Python y se basa en la versión 5 del artículo e incluye las capas de cuello de botella (*BottleNeck*) y el factor de compresión, por lo que estrictamente hablando es una implementación de DenseNet-BC. También incluye algunas modificaciones menores para hacerlo compatible con el cuantificador y compilador Vitis AI. En particular, el orden de las capas de normalización de Batch (BatchNorm), funciones de activación RelU y las capas de convolución se ha modificado de BN-> ReLU-> Conv a Conv-> BN-> ReLU.

Los autores del artículo original utilizaron un optimizador de descenso de gradiente estocástico (SGD), mientras que el script de entrenamiento en este ejemplo (train.py) utiliza RMSProp(*Root Mean Square Propagation*) aunque también se proporciona el código necesario para la optimización de SGD.

```
'''
Optimizer
RMSprop used in this example.
SGD  with Nesterov momentum was used in original paper
'''
#opt = SGD(lr=learnrate, momentum=0.9, nesterov=True)
opt = RMSprop(lr=learnrate)
```

Código 4-1: Distintas opciones de optimizador

El código de DenseNetX.py está configurado para el conjunto de datos CIFAR10, por lo que la primera capa convolucional usa un kernel 3x3 con una longitud de paso de 1 y se omite la primera capa de agrupación máxima(máx. pooling). Para otras bases de datos se debe cambiar la primera capa de convolución para adaptarse a la entrada especifica, por ejemplo en la presentada en el paper original la primera capa convolucional se configuró para un kernel de 7x7, longitudes de zancada de 2 y se incluyó una capa de agrupación máxima para el conjunto de datos de ImageNet.

```
# Use this for CIFAR-10, CIFAR-100
# first convolutional layer + BN + ReLU (Imagenet style)
net = Conv2D((2*k),3,strides=1,use_bias=False,kernel_initializer='he_uniform',kernel_regularizer=l2(weight_decay),padding='same')(input_layer)
net = BatchNormalization()(net)
net = Activation('relu')(net)


'''
# Use this for IMAGENET
# first convolutional layer + BN + ReLU
net = Conv2D((2*k), 7, strides=2, use_bias=False)(input_layer)
net = BatchNormalization()(net)
net = Activation('relu')(net)

# max pooling layer
net = MaxPooling2D(3, 2)(net)
'''
```

Código 4-2*: Configuración de la primera capa convolucional*

Se le denomina DenseNetX y la X del nombre es un parámetro que indica la profundidad de la red que en artículo original eran DenseNet-121, 169, 201 y 264. Es interesante entender como decidieron usar esa nomenclatura para entender mejor la arquitectura. Utilizaron el parámetro 'X' para indicar la profundidad de sus redes, pero contaron solo las capas convolucionales y completamente conectadas, no incluyeron las capas de agrupación(*pooling)*, BatchNorm o activación. Eso lo podemos observar en la tabla 4.1

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv; 3 × 3 conv] × 24	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48	[1 × 1 conv; 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv; 3 × 3 conv] × 16	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Tabla 4-2: Diferentes arquitecturas de Densenet.

Figura extraída de [9]

Por ejemplo, DenseNet-121 se compone de:

- la primera capa convolucional = 1
- 1 bloque denso de 6 capas convolucionales 1x1 y 3x3 = 6 x 2 = 12.
- 1 capa de transición con 1 capa convolucional = 1.
- 1 bloque denso de 12 capas convolucionales 1x1 y 3x3 = 12 x 2 = 24.
- 1 capa de transición con 1 capa convolucional = 1.
- 1 bloque denso de 24 capas convolucionales 1x1 y 3x3 = 24 x 2 = 48.
- 1 capa de transición con 1 capa convolucional = 1.
- 1 bloque denso de 16 capas convolucionales 1x1 y 3x3 = 16 x 2 = 32.
- 1 capa de clasificación que contiene 1 capa completamente conectada = 1

La suma de 1 + 12 + 1 + 24 + 1 + 48 + 1 + 32 + 1 = 121

El modelo DenseNet utilizado es el 121. Los 4 bloques son creados por el argumento *convlayers* que debe proporcionarse como una lista de Python y si se deseara utilizar otro se cambiarían los valores a la arquitectura deseada.

```
densenetx(input_shape=(224,224,3),classes=1000,k=32,drop_rate=0.2,theta=0.5,weight_decay=1e-4,convlayers=[6,12,24,16]):
```

Código 4-3. Parámetros de la red Densenetx

4.3 Implementación del diseño

Para implementar el diseño se utilizará un ordenador con sistema operativo Ubuntu y la placa Xilinx ZCU104. En este punto se explica las preparaciones del entorno de desarrollo necesarias. Seguidamente se explica la implementación paso a paso desde la descripción de la red en Tensor Flow hasta su despliegue e inferencia en la placa.

Durante este desarrollo se seguirán los siguientes pasos:

- Entrenamiento y evaluación de una versión personalizada de la red DenseNet usando TensorFlow Keras.
- Conversión del *checkpoint* de Keras en formato HDF5 en un *checkpoint* compatible con TensorFlow.
- Eliminación de los nodos de entrenamiento y conversión de las variables del gráfico en constantes (denominado "congelación del gráfico").
- Evaluación del modelo congelado utilizando el conjunto de datos de prueba CIFAR-10.
- Cuantificación del modelo congelado de punto flotante.
- Evaluación del modelo cuantificado utilizando el conjunto de datos de prueba CIFAR-10.
- Compilación del modelo cuantificado para crear el archivo .xmodel listo para su ejecución en la IP del acelerador DPU.
- Descarga y ejecución de la aplicación en la placa ZCU104

4.3.1 Preparación de la maquina anfitriona y la placa de destino

Hay que hacer un trabajo previo de creación y configuración del sistema operativo sobre el que vamos a trabajar(host), en nuestro caso será sobre un sistema operativo en Ubuntu 18.04.

Para utilizar al máximo lo que ofrece Xilinx, se necesita instalar en el sistema operativo tres herramientas:

- Vitis Software Plataform 2020: La plataforma de software unificada Vitis es una nueva herramienta que combina todos los aspectos del desarrollo de software Xilinx en un entorno unificado.
- Petalinux 2020 Tools: Las herramientas PetaLinux facilitan el desarrollo de productos basados en Linux desde el inicio del sistema hasta la ejecución.
- Vitis AI v1.3: El entorno de desarrollo que vamos a utilizar.

Para instalar Vitis AI lo primero es instalar Docker una aplicación con la cual se automatiza el despliegue de contenedores software. Copiamos el repositorio de Vitis AI de Github[14] en el host y lanzamos el contenedor de Vitis AI mediante Docker.

Además, es necesario preparar la placa instalando en ella el sistema operativo. Se ha descargado el entorno de la placa con DPU y el tiempo de ejecución correspondiente integrado. Se comprueba su correcto funcionamiento copiando repositorios de imágenes y comprobando que funcione la inferencia de las demos.

También es imprescindible preparar el entorno de compilación cruzada en en host instalando un archivo skd, el cual al ejecutarse se aplica en el host para configurar los encabezados sysroot, las bibliotecas y los archivos de inclusión para la compilación cruzada de aplicaciones que se ejecutan en las plataformas integradas. Para más detalles sobre la instalación ver Anexo[A].

4.3.2 Flujo de trabajo con Vitis AI

Comenzamos lanzando una terminal y haciendo el comando cd a la carpeta de trabajo DensenetX. Como parte del procedimiento de configuración del host se clona el repositorio de Vitis AI en la máquina host. En la carpeta Vitis AI de ese repositorio hay un script de shell llamado docker_run.sh que lanzará el contenedor Docker. Se tiene que elegir lanzar la versión de GPU o CPU, siendo preferible la versión GPU si se dispone de tarjeta gráfica en el host y siendo imprescindible para el entrenamiento de la red. En la terminal introduzco el siguiente comando:

```
# Navego hasta a la carpeta de trabajo
cd ~/Documents/DenseNetX

# para lanzar el GPU docker
home/electra/Vitis-AI/docker_run.sh xilinx/vitis-ai-gpu:latest

# ...o para lanzar el CPU docker
home/electra/Vitis-AI/docker_run.sh xilinx/vitis-ai-cpu:latest
```

A continuación, se observa la figura 4-2 en la terminal

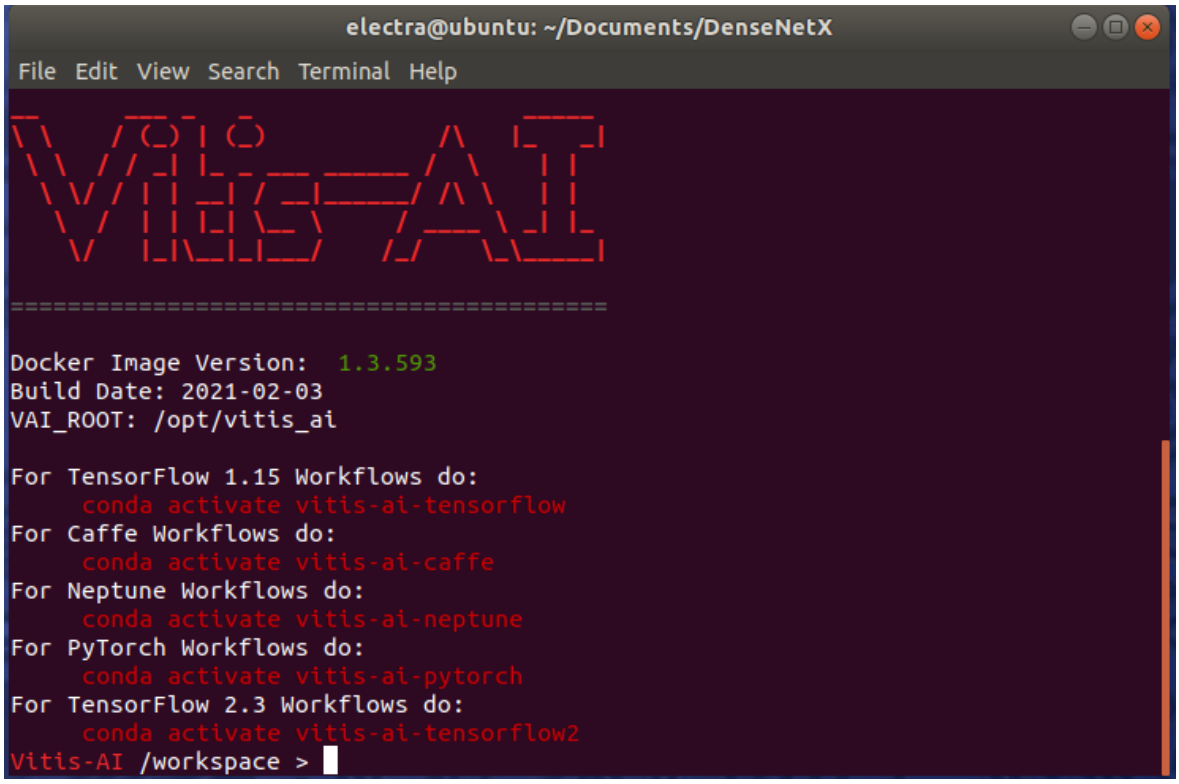


Figura 4-3. Terminal tras lanzar el Docker de Vitis AI

A continuación, se elige el entorno de Tensorflow mediante el comando: *conda activate vitis-ai-tensorflow* y se configuran todas las variables de entorno, principalmente punteros a carpetas y archivos. También creará las carpetas para los registros y el punto de control de keras entrenado *k_model.h5*.

Entrenamiento y evaluación. El entrenamiento lleva un tiempo considerable, entre 8 y 12 horas, según la GPU. Se puede reducir el número de épocas editando la línea export EPOCHS = 160 para reducir su tamaño y tiempo de ejecución. Obviamente, menos épocas de entrenamiento tendrán un impacto negativo en la precisión final.

El proceso de entrenamiento está definido por el script train.py Python. Primero, el conjunto de datos CIFAR10 es descargado y preprocesado por el script datadownload.py que es llamado por train.py. Todas las imágenes se someten a un preprocesamiento de evaluación y calibración de cuantificación simple antes de ser utilizadas para entrenamiento. Las imágenes se normalizan para llevar todos los valores de píxeles en el rango de 0 a 1,0 dividiéndolos por 255.

Las imágenes también se someterán a un aumento en forma de rotación aleatoria, volteo horizontal (es decir, alrededor del eje vertical), desplazamiento hacia arriba y hacia abajo, cizallamiento y zoom.

```
data_augment = ImageDataGenerator(rotation_range=10,
                                  horizontal_flip=True,
                                  height_shift_range=0.1,
                                  width_shift_range=0.1,
                                  shear_range=0.1,
                                  zoom_range=0.1)

train_generator = data_augment.flow(x=x_train,
                                    y=y_train,
                                    batch_size=batchsize,
                                    shuffle=True)
```

Código 4-4. Trasformaciones de las imágenes para el entrenamiento

Al final del entrenamiento se calcula la precisión del modelo entrenado según el código 4-5

```
scores = model.evaluate(x=x_test,y=y_test,batch_size=50, verbose=0)
print ('Evaluation Loss     : ', scores[0])
print ('Evaluation Accuracy: ', scores[1])
```

Código 4-5. Evaluación del modelo

El modelo entrenado Keras checkpoint se guarda como archivo HDF5 en el archivo creado anteriormente *k_model.h5* .

Conversión del *checkpoint* y congelación del gráfico (*freezing the graph*): Las herramientas de Vitis AI no pueden operar directamente en los puntos de control de Keras y requieren un gráfico congelado compatible con TensorFlow como formato de entrada. Para crear el gráfico congelado se necesitan dos pasos:

- El archivo HDF5 se convierte en un *checkpoint* de TensorFlow.
- El *checkpoint* de TensorFlow se convierte en un 'gráfico congelado' en formato protobuf binario.

```
keras_2_tf() {
  python keras_2_tf.py \
    --keras_hdf5 ${KERAS}/${K_MODEL} \
    --tf_ckpt    ${TFCKPT_DIR}/${TFCKPT}
}


freeze() {
  freeze_graph \
    --input_meta_graph  ${TFCKPT_DIR}/${TFCKPT}.meta \
    --input_checkpoint  ${TFCKPT_DIR}/${TFCKPT} \
    --output_graph      ${FREEZE}/${FROZEN_GRAPH} \
    --output_node_names ${OUTPUT_NODE} \
    --input_binary      true
}
```

Código 4-6: Funciones de transformar el Keras y congelar el grafico.

El archivo *frozen_graph.pb* de salida se conoce generalmente como un "gráfico congelado", ya que todas las variables se convierten en constantes y los nodos de gráficos asociados con el entrenamiento, como el optimizador y las funciones de pérdida, se eliminan.

Evaluación del modelo congelado: Este es un paso opcional ya que el gráfico congelado todavía está en formato de punto flotante y debería dar resultados de precisión casi idénticos a los de la evaluación realizada durante la fase de entrenamiento. Todas las 10k imágenes de prueba CIFAR10 se pasan a través del modelo congelado y se calcula la precisión.

```
eval_graph() {
  dir_name=$1
  graph=$2
  python eval_graph.py \
    --graph        $dir_name/$graph \
    --input_node   ${INPUT_NODE} \
    --output_node  ${OUTPUT_NODE} \
    --batchsize    100
}
```

Código 4-7: Función de evaluación de precisión.

Cuantificación del modelo: El acelerador de DPU IP ejecuta todos los cálculos en formato entero de 8 bits, por lo que debemos cuantificar nuestro gráfico congelado de punto flotante. Esto se realiza mediante las herramientas de Vitis AI, en particular mediante el comando 'vai_q_tensorflow quantize' al cual es necesario pasarle distintos argumentos con valores.

```
vai_q_tensorflow quantize \
  --input_frozen_graph ${FREEZE}/${FROZEN_GRAPH} \
              --input_fn           image_input_fn.calib_input \
              --output_dir         ${QUANT} \
        --input_nodes        ${INPUT_NODE} \
              --output_nodes       ${OUTPUT_NODE} \
              --input_shapes       ${INPUT_SHAPE} \
              --calib_iter         10
```

Código 4-8: Función para cuantizar la red

Son necesarios un conjunto de datos de muestra para calibrar el proceso de cuantificación. Estos datos se pasarán a través del modelo en una pasada hacia adelante y, por lo tanto, deben procesarse exactamente de la misma manera que los datos se procesan previamente en el entrenamiento. La función que apunta a el argumento --input_fn deberá contener todos los los pasos de preprocesamiento.

Las imágenes para la calibración son creadas por el script de Python tf_gen_images.py y luego almacenadas en la carpeta.

El script de Python image_input_fn.py contiene una única función llamada calib_input que abre las imágenes con OpenCV, las transforma a RGB desde BGR ya que el modelo se entrenó en imágenes RGB y luego las normaliza para tener todos los píxeles en el rango de 0 a 1.0, exactamente como se hizo en el entrenamiento y la evaluación.

```
# open image as BGR
image = cv2.imread(calib_image_name)

# change to RGB
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# normalize
image = image/255.0
```

Código 4-9. Transformar imágenes y normalizarlas

El número de imágenes generadas para su uso en la calibración se establece mediante la variable de entorno CALIB_IMAGES. Se debe tener cuidado de que el número de iteraciones de calibración (--calib_iter) multiplicado por el tamaño del lote de calibración no exceda el número total de imágenes disponibles (CALIB_IMAGES).

Una vez completada la cuantificación, tendremos el modelo de implementación cuantificado (deploy_model.pb) y el modelo de evaluación (quantize_eval_model.pb).

Evaluación del modelo: La conversión de un modelo de punto flotante donde los valores pueden tener un rango dinámico muy amplio a un modelo de 8 bits donde los valores solo pueden tener uno de 256 valores conduce casi inevitablemente a una pequeña pérdida de precisión. Usamos el modelo de evaluación cuantificado para ver exactamente cuánto impacto ha tenido la cuantificación.

Es la misma función de Python que en la evaluación anterior, código 4-7, por lo que se calcula igual.

Compilación del modelo cuantificado: La DPU IP es una IP de soft-core cuya única función es acelerar la ejecución de redes neuronales convolucionales. Es un coprocesador que tiene su propio conjunto de instrucciones las cuales se pasan a la DPU en formato de archivo Xmodel. El compilador de Vitis AI convertirá y optimizará, cuando sea posible, el modelo de implementación cuantificado en un conjunto de microinstrucciones y luego las enviará a un archivo Xmodel.

Las instrucciones generadas son específicas para la configuración particular de la DPU. Los parámetros de la DPU están contenidos en un archivo arch.json que debe crearse para cada placa de destino; como se ve en la guía del usuario de Vitis AI [11].

En el caso específico del ZCU104 y el archivo de imagen de la tarjeta SD preparada que instalamos, el archivo arch.json se incluye en el contenedor docker y su ubicación se pasa al comando vai_c_tensorflow a través del argumento --arch.

```
compile() {
  vai_c_tensorflow \
    --frozen_pb  ${QUANT}/quantize_eval_model.pb \
    --arch       ${ARCH} \
    --output_dir ${COMPILE} \
    --net_name   ${NET_NAME} \
    --options    "{'mode':'${DPU_MODE}'}"
}
```

Código 4-10: Compilador para desplegar en la placa

Una vez que se completa la compilación, se crea el archivo *densenetx.xmodel*.

Descarga y ejecución de la aplicación en la placa ZCU104: El paso final consiste en preparar una carpeta *objetivo* con todo lo necesario para ejecutar la aplicación en la placa y transferírsela. Se envían 10k imágenes del conjunto de test de CIFAR10, el archivo xmodel y parámetros para la DPU en archivos json.

Para poder transferirlo es necesario que se encuentren en la misma subred la placa y el host puesto que se conectaran con el comando scp. El comando es el siguiente:
scp -r ./objetivo root@192.168.1.227:~/

Suponiendo que 192.168.1.227 es la dirección IP de la placa ZCU104 y objetivo la carpeta con lo necesario para la ejecución. Para realizar la conexión nos pide una contraseña la cual es: *root.*

4.3.3 Entorno de despliegue

Como se puede ver en la figura 4-4 la placa está conectada a la corriente con su transformador C), a una misma red de internet que el ordenador host por cable ethernet B) y con un cable micro USB a un puerto del host para una conexión serie D). De esta forma se puede observar en la pantalla del ordenador(host) un emulador de terminal de la placa ZCU104 en la pantalla E), al cual hemos accedido gracias la herramienta PUTTY de emulador de terminal mediante conexión serie con el USB. En este ordenador es en el cual tenemos todo el entorno de diseño de Vitis AI junto a otras herramientas necesarias para el desarrollo del trabajo. Es así como hemos ejecutado la red neuronal en la placa y obtenido los resultados de precisión y eficacia.

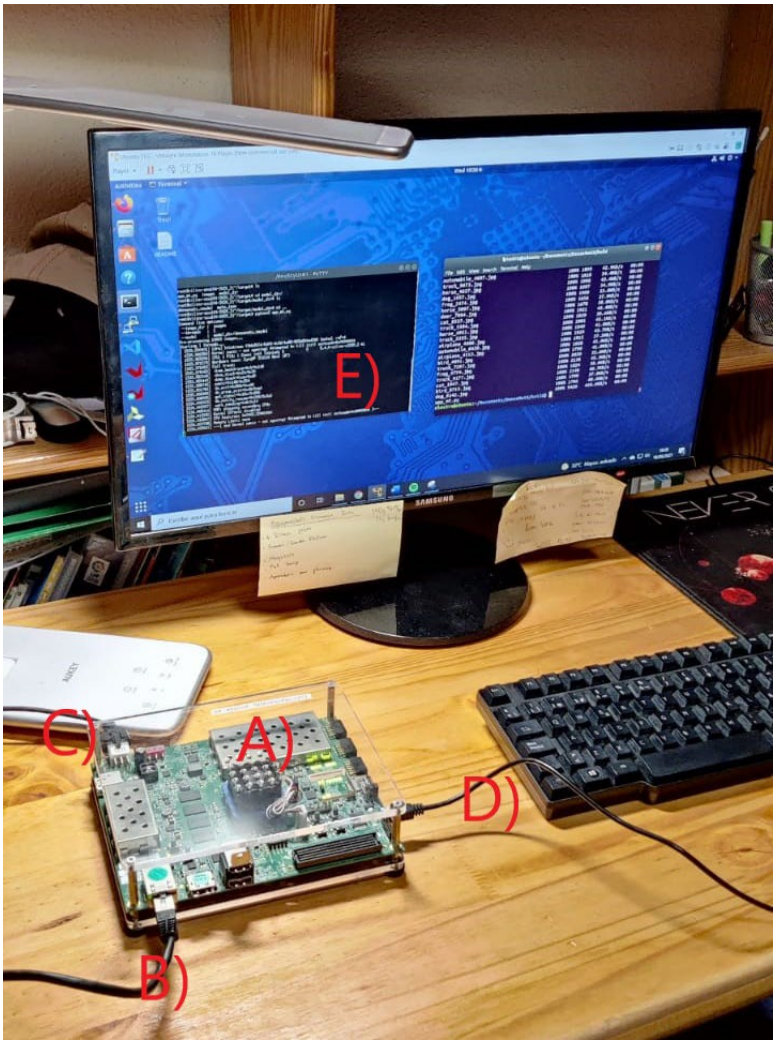


Figura 4-4: Conexión entre la placa y el ordenador.

A) La placa ZCU104, B) Conexión por cable Ethernet, C) Cable de alimentación, D) Conexión serie micro usb-USB al host, E) Terminal de la placa de la conexión en serie USB

También en la siguiente figura 4-5 podemos ver algunos de los detalles y componentes principales de la placa. A) Corresponde al chip MPSoC, el cual combina *processing system*(PS) y *programable logic*(PL). El PS este compuesto de un ARM CORTEX-A53 de 64 bits con 4 núcleos, un CORTEX-R5 doble núcleo para el procesamiento de datos en tiempo real y un procesador MAIL-400 encargado del procesado de pixeles [13]. La memoria de la parte del sistema de procesador es el F), el cual tiene 4GB de DDR4. El C) es la interfaz de ethernet de alta velocidad, D) es un puerto serie USB 3.0 y E) es la entrada de alimentación. El B) es la entrada de la interfaz microSD desde la cual lanzamos el sistema operativo de la placa basado en Linux según explicamos en el Anexo. Esta configuración esta diseñad para poder cargar aplicaciones en la ZCU104 de forma sencilla desde el ordenador con Ubuntu(host) en el cual esta preparado todo el entorno de diseño de VITIS AI.

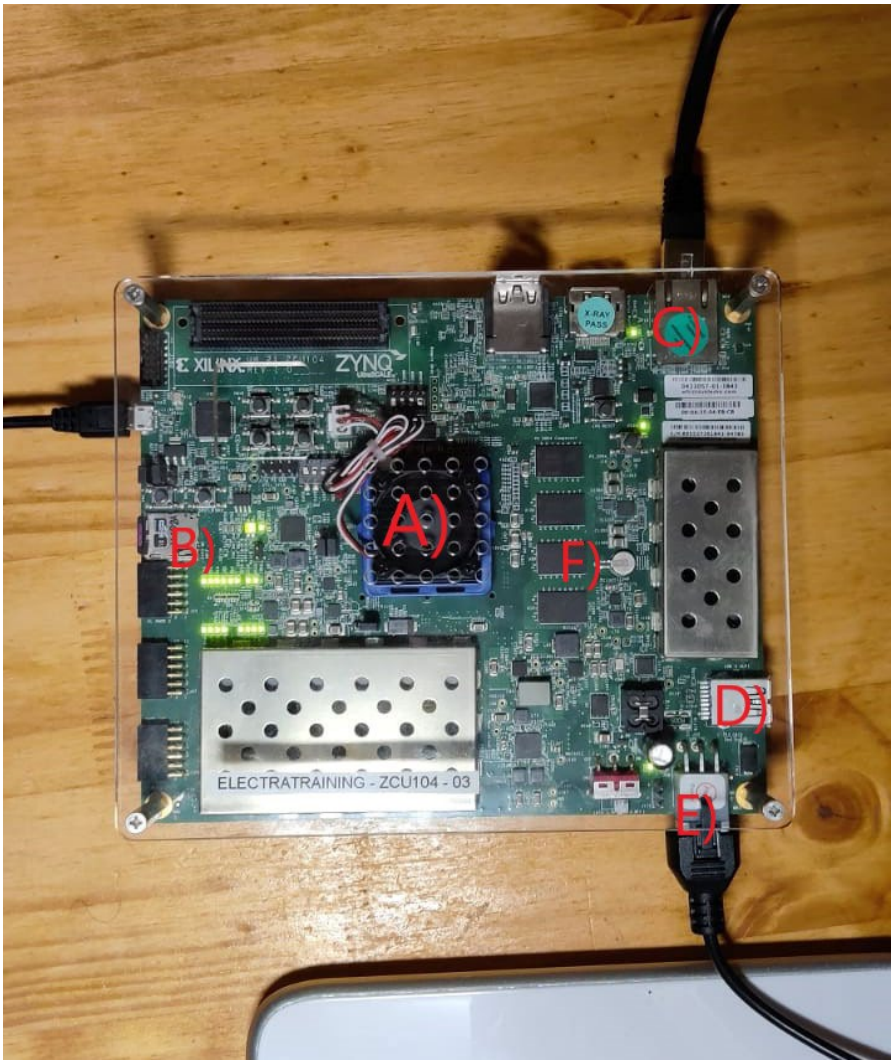


Figura 4-5: Detalle de la ZCU104.

En ella se aprecia A) El chip MPSoC, B) Interfaz micro USB, C) Interfaz de ethernet, D) Interfaz USB 3.0, E) Entrada de alimentación, F) Memoria SDRAM

4.3.4 Resultados

En este punto se muestran dos tipos de resultados muy importantes para comprender la implementación, precisión y rendimiento (o *througput*). El primero es la precisión que tiene la red neuronal al ejecutarla con las imágenes de prueba del *dataset* a lo largo del proceso de pasar de una red descrita en Python hasta un *bitstream* cargado en la FPGA de la placa ZCU104.

Es necesario entender los motivos por los cuales puede variar la precisión debido a eso y si realmente supone una gran diferencia.
La precisión que mantiene la red tras transformarla en distintos puntos del proceso depende principalmente del formato de los datos de los cuales se pasa de uno a otro. La red recién entrenada parte con una buena precisión y la primera transformación que

sufre es la de pasar los estados variables a estados fijos de la red en el paso de congelación del grafo. Como se observa en la tabla 4-3 no afecta significativamente este paso ya que se sigue manteniendo el formato de datos de punto flotante.

En la siguiente transformación al pasar al modelo cuantizado ya si se pierde precisión debido al hecho de pasar de un rango muy grande de datos en punto flotante a un modelo de 8bits con 256 posibilidades de valor. Aun así, pierde poca precisión respecto al estado anterior.

Por último para pasar al modelo hardware Vitis AI cuantiza y modela de forma que pueda ser desplegado en la placa ZCU104 y pierde un poco de precisión pero no lo suficiente para que sea importante. Podemos concluir que, a pesar de múltiples transformaciones, cuantizaciones y compilados la red desplegada final consigue mantener una muy buena precisión respecto a la original, haciendo muy viable su uso en aplicaciones reales.

Estado	Post entrenamiento	Grafo congelado	Modelo cuantizado (INT8)	Modelo hardware
Precisión	92.80%	93.03%	92.63%	91.37%

Tabla 4-3: Resultados de precisión

Una parte muy importante del trabajo es la eficiencia que tiene la placa al ser ejecutada y es por eso que resulta imprescindible medir el rendimiento o *througput.*

Al ejecutar la red en la ZCU104 podemos elegir el tiempo que estará procesando imágenes y el número de hilos que usará para ello. Para un tiempo estándar de 60 segundos de tiempo de ejecución y un solo hilo se comprueba el rendimiento de la red en frames por segundo. Según aumentamos los hilos aumenta consigo el rendimiento rápido al principio y luego disminuye. Como se puede ver en la tabla 4-4 en la cual la diferencia entre un hilo y dos es muy grande, pero disminuye progresivamente entre en las siguientes. Es por eso que observamos que a partir de 4 hilos no es mucho más beneficioso seguir aumentando los hilos ya que se comienza a llegar al límite de paralelización posible en la placa y aumentar el número de hilos no sirve para aumentar la eficiencia.

Hilos	1	2	3	4	5
Eficiencia(fps)	482.65	704.91	791.14	822.37	831.05

Tabla 4-4: Resultados de Eficiencia

5 Conclusiones y trabajo futuro

Por último, se exponen las conclusiones finales del trabajo y el posible trabajo futuro a realizar siguiendo lo iniciado con este trabajo.

5.1 Conclusiones

Comienza el trabajo buscando nuevas formas de mejorar la inferencia de aprendizaje automático, ya fuese mejorando la latencia, el consumo eléctrico, el rendimiento o incluso la viabilidad económica, dejando de lado el hardware clásico de CPU y GPU y buscando opciones para placas de FPGA. Se presenta la herramienta de Vitis AI como el elemento ideal para implementarlo de forma directa y sencilla sobre plataformas hardware. Como se ve durante el trabajo es muy conveniente su utilización sirviendo para implementar una red neuronal que si hubiese sido creada de forma clásica hubiese sido una tarea casi imposible y requeriría unos conocimientos avanzados de diseño hardware por encima de los de un estudiante de grado.

Se utilizan la mayoría de las herramientas que ofrece Vitis AI, demostrando su viabilidad para su uso por parte de otros estudiantes con limitados conocimientos sobre hardware de base y que su aplicación en el mundo real puede ser muy beneficiosa. Y finalmente se puede ver que se puede implementar una red neuronal sobre la placa ZCU104 y además con unos buenos resultados tanto de rendimiento como de precisión. Viendo los resultados obtenidos se demuestra que sería una gran herramienta para proceso de datos en tiempo real debido a los altos fps a los que funciona.

Por último, destacar el trabajo y esfuerzo que conlleva implementar una red neuronal sobre un sistema hardware FPGA incluso con la ayuda de las herramientas de Vitis, debido a la alta complejidad de los sistemas. Durante el grado se estudia la parte de teórica de las tecnologías, herramientas y técnicas de diseño hardware imprescindibles para realizar este proyecto, pero sin llegar a realizar un proyecto tan avanzado como el de este trabajo.

5.2 Trabajo Futuro

Como trabajo futuro sería interesante poder llegar a realizar más avances dentro del campo de la inferencia de redes neuronales con sistemas de FPGA, puesto que según lo visto presentan grandes ventajas en sistemas de Edge e implementarlo en conjunto con sistemas en la nube.

Aunque se han utilizado la mayoría de las herramientas de Vitis AI se podría llegar a trabajar la herramienta del AI Optimizer de Vitis AI, que solo está disponible bajo compra de licencia para ver más posibilidades de las que ofrece Xillinx, como podría ser la poda de redes neuronales para aumentar la inferencia manteniendo una buena precisión.

También sería interesante incluir el trabajo sobre la placa en un sistema integrado de algún tipo, para realizar un uso más practico aparte de la pura experimentación. Se podría intentar la integración en otras placas más diseñadas a la inferencia, ya que como se ve en el desarrollo la dificultad de cambiar de una placa a otra es mínima, simplemente cambiando los archivos específicos de cada una en los pasos finales hace sencilla su implementación en distintas placas. Una de esas opciones seria Xilinx Kria260, una placa basada en la arquitectura Zynq UltraScale+ MPSoC, especialmente diseñada para aplicación de de visión artificial, ya que cuenta con distintos sensores e interfaces y muy interesante como solución de bajo coste.

Bibliografía

[1] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", Part of Advances in Neural Information Processing Systems 25 (NIPS 2012)

[2] A. M. TURING". "ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM" https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

[3] "COMPUTING MACHINERY AND INTELLIGENCE By A. M. Turing" https://www.csee.umbc.edu/courses/471/papers/turing.pdf

[4] Lettvin, JY, Maturana, HR, McCulloch, WS y Pitts, WH; What the Frog's Eye Tells the Frog's Brain, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.4995&rep=rep1&type=pdf

[5] Donald O. Hebb, La organización del comportamiento: una teoría neuropsicológica , Wiley , al. "Libro de Wiley en psicología clínica",1966

[6] Hopfield, J. J. (1982). "Neural networks and physical systems with emergent collective computational abilities". Proc. Natl. Acad. Sci.

[7] Statistical Learning Theory by Vapnik, Vladimir

[8] Vivienne Sze, Senior Member, IEEE, Yu-Hsin Chen, Student Member, IEEE, Tien-Ju Yang, Student Member, IEEE, Joel Emer, Fellow, IEEE "Efficient Processing of Deep Neural Networks: A Tutorial and Survey"

[9] Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger "Densely Connected Convolutional Networks"

[10] Zynq DPU v3.2 IP Product Guide – Xilinx https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_2/pg338-dpu.pdf

[11] Xilinx, "Vitis AI User Guide" UG1414 (v1.3) February 3, 2021

[12] D. R. V. L. B. Thambawita , Roshan Ragel and Dhammika Elkaduwe, Department of Computer Engineering, University of Peradeniya "To Use or Not to Use: Graphics Processing Units for Pattern Matching Algorithms"

[13] ZCU104 Evaluation Board User Guide, UG1267 (v1.1) October 9, 2018

[14] Repositorio de Vitis AI, Xilinx. https://github.com/Xilinx/Vitis-AI

[15] NVIDIA Ampere Architecture In-Depth https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

Anexos

A Manual de instalación

Preparación 1. Carga de la imagen de sistema en la placa ZCU104

En este paso vamos a graban en una SD la imagen de la palaca ZCU104 con el DPU preconstruida para nuestra placa.

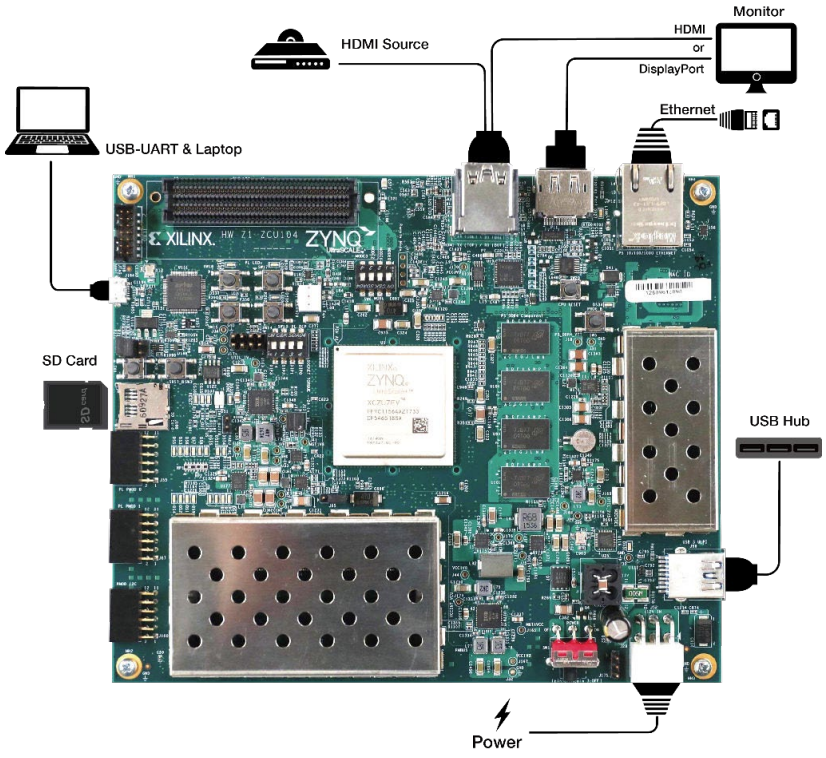
Primero descargamos la imagen de la placa ZCU104: ZCU104 prebuilt image para Vitis AI y la grabamos en una SD mediante una herramienta de grabado de imágenes.

Insertamos la SD en la placa ZCU104, y colocamos los pines en el modo de arranque por SD. SW6 [4:1]: OFF, OFF, OFF, ON.

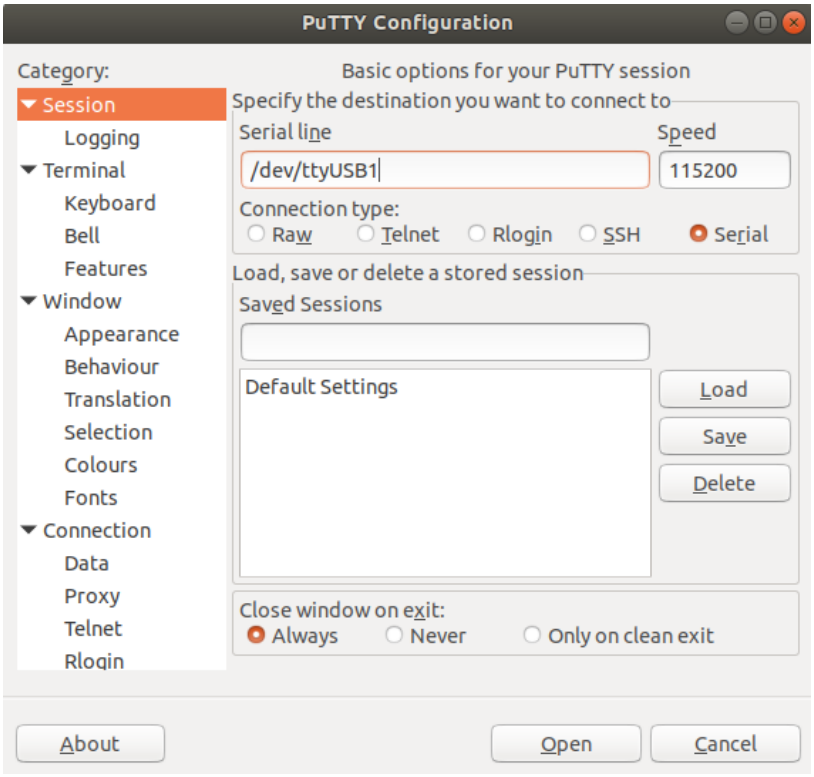
Para arrancar desde una tarjeta SD:
1. Almacene un archivo de imagen de arranque Zynq UltraScale + MPSoC válido en una tarjeta SD (enchufada en la toma SD J100) conectada a la interfaz MIO SD.
2.Configure los pines del modo de arranque SW6 [4: 1] PS_MODE [3: 0] como se indica en la Tabla 2-4 para SD1.3.
3.Apague y encienda o presione el botón SW4 de reinicio de encendido (POR). SW4 es la llamada 20 en la Figura 2-1.

Ref:
https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf

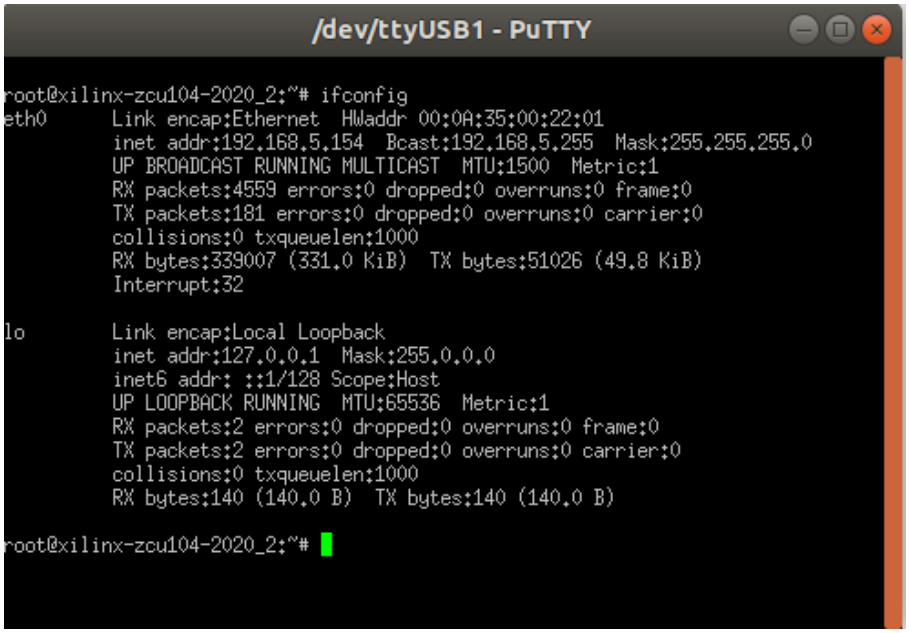
Conectamos la placa según el siguiente esquema extraído de[13]:



Luego con la aplicacion Putty de emulador de terminal, la conectaremos en serie con el ordenador, conectándonos a traves del puerto del USB1 con una velocidad(baud):115200

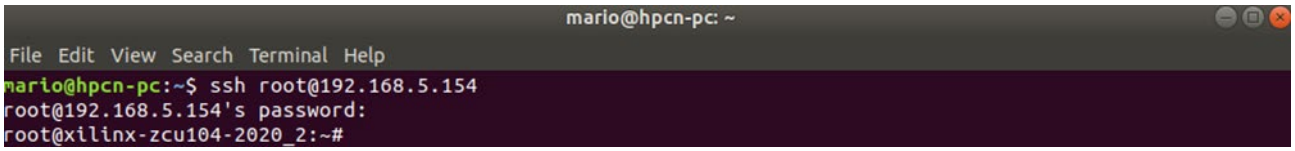


Despues averiguamos la direccion IP de la placa:
root@xilinx-zcu104-2020_2:~# ifconfig



De esta manera averiguamos la IP:192.168.5.154 y ya podremos operar la placa mediante la terminal de Ubuntu con el comando SSH.

Desde la consola del ordenador:
$ ssh root@(IP de la placa)
Nos pide contraseña, la contraseña es root.



Paso 2. Preparando el entorno de la placa y demo

En este paso copiaremos un repositorio de imágenes y video desde las librerías de Vitis AI para comprobar su correcto funcionamiento.
Como nuestra placa tiene conexión a internet con ejecutar los siguientes comandos sirve:

root@xilinx-zcu104-2020_2:~# wget -O vitis_ai_library_r1.3.0_images.tar.gz https://www.xilinx.com/bin/public/openDownload?filename=vitis_ai_library_r1.3.0_images.tar.gz

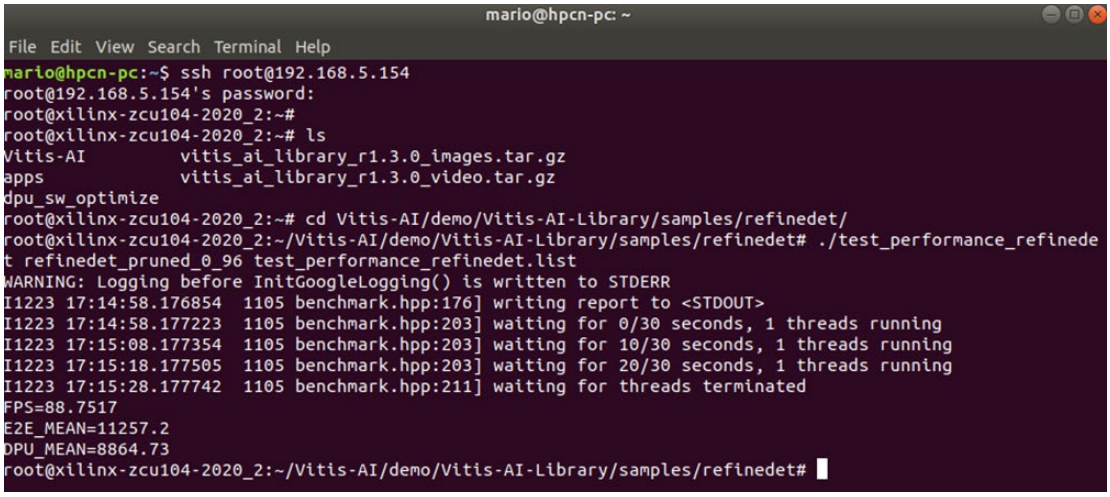
root@xilinx-zcu104-2020_2:~# wget -O vitis_ai_library_r1.3.0_video.tar.gz https://www.xilinx.com/bin/public/openDownload?filename=vitis_ai_library_r1.3.0_video.tar.gz

Descomprimimos las imágenes y videos:

root@xilinx-zcu104-2020_2:~# tar -xzvf vitis_ai_library_r1.3.0_images.tar.gz -C ~/Vitis-AI/demo/Vitis_AI_Library
root@xilinx-zcu104-2020_2:~# tar -xzvf vitis_ai_library_r1.3.0_video.tar.gz -C ~/Vitis-AI/demo/Vitis_AI_Library

Ejecutamos una demo para comprobar el funcionamiento:

root@xilinx-zcu104-2020_2:~# cd Vitis-AI/demo/Vitis_AI_Library/samples/refinedet
root@xilinx-zcu104-2020_2:~/Vitis-AI/demo/Vitis_AI_Library/samples/refinedet# ./test_performance_refinedet refinedet_pruned_0_96 test_performance_refinedet.list



también se pueden usar parámetros para definir el número de hilos y el tiempo de ejecución.
-t: <num_of_threads>
-s: <num_of_seconds>

Paso 3. Preparación del entorno de compilación cruzada

Se descarga el script sdk-2020.2.0.0.sh en el host. El archivo sdk se genera mediante el comando petalinux-build --sdk y se aplica para configurar los encabezados sysroot, las bibliotecas y los archivos de inclusión para la compilación cruzada de aplicaciones que se ejecutan en las plataformas integradas.

Se edita el archivo sdk-2020.2.0.0.sh para poder ejecutarlo:

$ chmod u+x sdk-2020.2.0.0.sh

Luego ejecuto el archivo:
$ ./sdk-2020.2.0.0.sh

Por defecto se instala en ~/opt/petalinux. Cuando se complete la instalación, siga las instrucciones y ejecute el siguiente comando para instalar el entorno de compilación cruzada de ARM integrado en el host.

$source /opt/petalinux/2020.2/environment-setup-aarch64-xilinx-linux

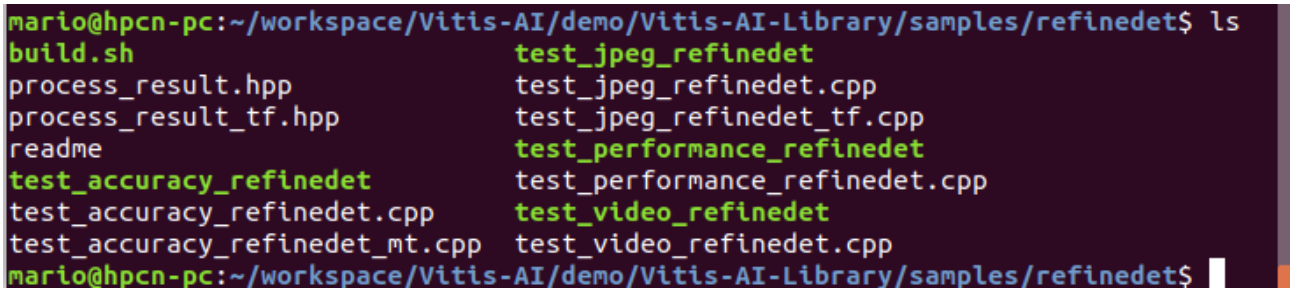
El comando anterior tiene que ser reejecutado cada vez que se lance una nueva terminal.

Descargamos este archivo de librerías y dependencias de Vitis AI (vitis_ai_2020.2-r1.3.0.tar.gz) y lo descomprimimos en la carpeta de /opt/petalinux/2020.2/environment-setup-aarch64-xilinx-linux, mediante el comando:

$ sudo tar -xzvf vitis_ai_2020.2-r1.3.0.tar.gz -C /opt/petalinux/2020.2/sysroots/aarch64-xilinx-linux

De esta forma la configuración del host estaría terminada y hacemos un ejemplo de cross-compile en la librería de Vitis AI.

$cd ~/Vitis-AI/demo/Vitis-AI-Library/samples/refinedet
$bash -x build.sh



Al ejecutar esto se crean 4 ejecutables nuevos en la carpeta(en verde).