CAMPUS TERRA

Deep learning en el borde

Trabajo Fin de Grado
Para a titulación do
Grado en Robótica

Autor:
Alicia Jiajun Lorenzo Lourido

Titores:
Dr. Fernando R. Pardo Seco
Dpto. de Electrónica e Computación

Dr. Xosé R. Fdez-Vidal
Dpto. de Física Aplicada

USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

ESCOLA POLITÉCNICA SUPERIOR DE ENXEÑARÍA

Convocatoria:
Julio de 2023

UNIVERSIDAD DE SANTIAGO DE COMPOSTELA

TRABAJO FIN DE GRADO

DEEP LEARNING EN EL BORDE

AUTOR:
ALICIA JIAJUN LORENZO LOURIDO

TITORES:
Dr. FERNANDO R. PARDO SECO
Dpto. de Electrónica e Computación

Dr. XOSÉ R. FDEZ-VIDAL
Dpto. de Física Aplicada

Este trabajo se presenta para cumplir con los requisitos normativos requeridos para lograr el Grado en Robótica

na

Escuela Politécnica Superior de Ingeniería.

Fecha de la Convocatoria: Julio de 2023

USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
ESCOLA POLITÉCNICA SUPERIOR DE ENXEÑARÍA

Trate de no convertirse en una persona de éxito, sino en una persona de valor.

Albert Einstein

Resumo

DEEP LEARNING EN EL BORDE

por
ALICIA JIAJUN LORENZO LOURIDO

Palabras Chave: *Visión Artificial, Aprendizaje máquina, FPGA, Quantización, Detección de objetos*

Neste traballo investigouse a viabilidade das FPGAs (Matrices de portas programables en campo) como dispositivos de computación no borde (Edge Computing) para a detección de obxectos nunha cadea industrial, centrándose especificamente na detección de cubos. Para iso utilizouse o popular conxunto de datos COCO 2017 para realizar comparativas e, posteriormente, entrenouse un modelo adaptado a unha aplicación industrial de detección de cubos nunha cadea de produción.

Os resultados obtidos indicaron que as FPGAs consomen menos enerxía en comparación coas GPUs, o que as converte nunha opción eficiente desde o punto de vista enerxético. Con todo, tamén se observa unha perda de precisión na tarefa de encadrar os obxectos, aínda que esta perda non afecta de forma crítica á detección de cubos nunha cadea industrial

Resumen

DEEP LEARNING EN EL BORDE

por
ALICIA JIAJUN LORENZO LOURIDO

Palabras Clave: *Visión Artificial, Aprendizaje máquina, FPGA, Quantización, Detección de objetos*

En este trabajo se investigó la viabilidad de las FPGAs (Matrices de puertas programables en campo) como dispositivos de computación en el borde (Edge Computing) para la detección de objetos en una cadena industrial, centrándose específicamente en la detección de cubos. Para ello se utilizó el popular conjunto de datos COCO 2017 para realizar comparativas y, posteriormente, se entrenó un modelo adaptado a una aplicación industrial de detección de cubos en una cadena de producción.

Los resultados obtenidos indicaron que las FPGAs consumen menos energía en comparación con las GPUs, lo que las convierte en una opción eficiente desde el punto de vista energético. Sin embargo, también se observa una pérdida de precisión en la tarea de encuadrar los objetos, aunque esta pérdida no afecta de forma crítica a la detección de cubos en una cadena industrial.

Abstract

DEEP LEARNING EN EL BORDE

by
ALICIA JIAJUN LORENZO LOURIDO

Keywords: *Visión Artificial, Aprendizaje máquina, FPGA, Quantización, Detección de objetos*

This work investigated the feasibility of Field-Programmable Gate Arrays (FPGAs) as edge computing devices for object detection in an industrial chain, specifically focusing on cube detection. The popular COCO 2017 dataset was used for comparative analysis, followed by training a model adapted for industrial cube detection in a production chain.

The results obtained indicated that FPGAs consume less energy compared to GPUs, making them an energy-efficient option. However, there is also a loss of precision in object bounding, although this loss does not critically affect cube detection in an industrial chain.

Agradecementos

Deseo expresar mi más sincero agradecimiento hacia mi respetado tutor, el distinguido Dr. Fernando R. Pardo Seco , quien ha ejercido una destacada influencia en mi proceso de investigación. Asimismo, extiendo mi gratitud a otro estimado mentor, Dr. Xosé R. Fdez-Vidal, por brindarme su valiosa asistencia en el abordaje de inquietudes surgidas en el ámbito de la visión artificial.

También quiero expresar mi profundo agradecimiento a mi familia por su inquebrantable apoyo durante el desarrollo de este trabajo. Su constante aliento, comprensión y amor han sido fundamentales para alcanzar este logro. Estoy sumamente agradecida por su presencia y respaldo incondicional en cada paso del camino.

Índice General

Resumo III

Resumen V

Abstract VII

Agradecementos IX

1. Introducción 1
1.1. Propuesta y justificación . . . 1
1.2. Aportación . . . 2
Alcance y Objetivo . . . 2
Aplicación . . . 2

2. Estado del Arte 3
2.1. Técnicas de entrenamiento de redes . . . 3
Federated Learning . . . 3
Transferencia de Aprendizaje Profundo (DTL) . . . 3
Destilación de conocimiento (Knowledge Distillation) . . . 3
2.2. Redes neuronales . . . 4
SSD . . . 4
MobileNet . . . 5
SqueezeDet . . . 6
EfficientDet . . . 7
RetinaNet . . . 8
Fast R-CNN . . . 9
YOLO . . . 9
Comparativa . . . 10
2.3. Optimización de modelos . . . 13
2.3.1. Durante el entrenamiento . . . 13
Stochastic Gradient Descent (SGD) . . . 13
Regularización . . . 13
Compartir peso, Weight Sharing . . . 13
2.3.2. Ajuste de hiperparámetros . . . 14
2.3.3. Después del entrenamiento . . . 14
Pruning . . . 14
Quantización . . . 15
2.4. Dispositivos hardware . . . 16
GPU, Jetson nvidia. . . . 16
VPU, Intel Movidius Neural Compute Stick . . . 16
TPU, Cámara inteligente . . . 17

TPU, Google Coral Edge . . . 17
NPU, IMX8 family . . . 17
DPU, Xilinx kV260 . . . 18
2.4.1. Comparativa . . . 18

3. Metodología **21**
3.1. Hardware . . . 21
3.2. Software . . . 21
3.3. Dataset COCO . . . 22
3.4. Dataset . . . 23
Obtención de datos . . . 23
Etiquetado de datos . . . 23
Características . . . 24
3.5. Entrenamiento del modelo . . . 25
3.5.1. Modelo seleccionado . . . 25
Configuración seleccionada . . . 26
3.5.2. Hiperparametros . . . 27
3.5.3. Entrenamiento . . . 28
3.6. Optimización del modelo . . . 29
3.6.1. Pasos de la quantización posterior al entrenamiento . . . 29
Calibración . . . 29
Quantización de los parámetros . . . 29
Quantización de activaciones . . . 29
3.7. Despliegue en la placa . . . 30
3.7.1. Conexión . . . 30
3.7.2. Modelo . . . 30
3.8. Evaluación . . . 31
3.8.1. Precisión . . . 31
Métricas . . . 31
3.8.2. Consumo . . . 32
GPU . . . 32
FPGA . . . 33

4. Resultados **35**
4.0.1. Precisión . . . 35
4.0.2. GPU . . . 35
Modelo Flotante . . . 35
Modelo quantizado . . . 35
4.0.3. FPGA . . . 36
4.0.4. Consumo . . . 36
4.0.5. GPU . . . 37
4.0.6. FPGA . . . 37
4.0.7. Comparativa . . . 37
4.0.8. Inferencia del modelo de deteción . . . 38
4.0.9. Obtención del modelo . . . 38

5. Conclusiones **39**

A. Anexo **41**

Bibliografía **47**

Índice de figuras

2.1. Arquitectura basica de SSD [7] . . . 4
2.2. Arquitectura de MobileNet-Tiny. [11] . . . 6
2.3. Pipeline de la detección de objetos con el modelo SqueezeDet. [12] . . . 7
2.4. Arquitectura EfficientDet [13] . . . 8
2.5. Arquitectura del RetinaNet [16] . . . 8
2.6. Arquitectura de la Faste R-CNN siendo el modelo VGG el backbone [18] . . . 9
2.7. Arquitectura de YOLO [19] . . . 9
2.8. Stochastic Gradient Descent (SGD) vs Gradient Descent (GD) . . . 13
2.9. Ejemplo de Weight Sharing con un factor de compresión $\frac{1}{4}$ [43] . . . 14
2.10. Poda no estructurada vs Poda estructurada [60] . . . 15

3.1. (a) Número de categorías anotadas y (b) número de instancias anotadas, por imagen,se muestra el promedio de categorías e instancias entre paréntesis). (c) Número de categorías versus número de instancias por categoría para varios conjuntos de datos populares de reconocimiento de objetos. (d) Distribución de tamaños de instancia para los conjuntos de datos (MS COCO, ImageNet Detection, PASCAL VOC y SUN) [34]. . . . 22
3.2. Número de instancias anotadas por categoría para MS COCO y PASCAL VOC [34]. . . . 23
3.3. Imágenes de ejemplo del dataset . . . 24
3.4. Imagen de ejemplo del etiqutado . . . 25
3.5. a) DenseNet and (b) Cross Stage Partial DenseNet (CSPDenseNet)[21]. 25
3.6. Arquitectura de Yolov4 . . . 26
3.7. Arquitectura de Yolov4 selecionada [86] . . . 27
3.8. En la parte de arriba estan las capas de la red empleada en el TFG y las de abajo son las capas originales de la red YOLOV4 para la detección de objetos del dataset COCO. . . . 28
3.9. Consumo con multimetro [86] . . . 33

4.1. Resultados obtenidos de la precisión de la GPU en el modelo float [34]. 35
4.2. Resultados obtenidos de la precisión de la gpu en el modelo quantizado [34]. . . . 36
4.3. Resultados obtenidos de la precisión de la DPU en el modelo quantizado [34]. . . . 36
4.4. Consumo en watios de la GPU . . . 37
4.5. Consumo en watios de la KV260 . . . 37
4.6. Las pérdidas (loss) en las 2000 iteraciones de entrenamiento. . . . 38
4.7. Ejemplo de deteción en la FPGA . . . 38

A.1. Comparativa de los modelos en el dataset COCO 2017 43
A.2. Comparativa de los modelos en la detección de cubos 44
A.3. Arquitectura YoloV4 Compilada para la DPU 45

Índice de Tablas

2.1. Comparación de MobileNet con modelos populares [9] 6
2.2. Comparación de modelos en el dataset KITTI con una GPU TITAN X (C) coche, (B) Bicicleta, (P) Peón ,(T) todo [11], [12], [15] 11
2.3. Comparación de modelos en el dataset de COCO signos (+) NVIDIA Tesla T4 GPU (∗) Tesla V100. (−) i7-8565U[36] & Nvidia M40 con el entorno de TensorRT TRT [11], [12], [15], [26], [28], [31]-[33] 12
2.4. Especificaciones de NVidia Jetson Xavier NX, Google Coral Mini ,K26 SOM de Xilinx,Raspberry Pi 4 B, Axis Q1615-LE Mk III, y Jetson Nano.[74], [76]-[78] 19
2.5. Comparativa de Modelos Power: Xilinx K26 SOM, Nvidia Jetson Nano y Nvidia Jetson TX2 [74] 19
2.6. Comparativa de Modelos FPS: Xilinx K26 SOM, Nvidia Jetson Nano y Nvidia Jetson TX2[74] 20
2.7. Comparison of Detection Network Performance Metrics for Jetson Nano, Jetson Xavier NX, and Coral Mini (Transposed). con el modelo YOLOv4-Tiny 20
2.8. Comparison of Detection Network Performance Metrics for Jetson Nano, Jetson Xavier NX, and Coral Mini (Transposed). con el modelo SSD MobileNet V2 20

3.1. Configuración de hiperparámetros[87] 27

4.1. Comparación de rendimiento y características 37

A.1. Modelos Pre-Entrenados de Intel en el OpenVINO™ Toolkit [35]. . . . 41
A.2. Especificaciones de MT-1707 42

Lista de Abreviacións

AP	**A**verage **P**recision
AR	**A**verage **R**ecall
BRB	**B**ottleneck **R**esidual **B**lock
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
DPU	**D**ata **P**rocessing **U**nit
DTL	**D**eep **T**ransfer **L**earning
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FPN	**F**eature **P**yramid **N**etwork
GPU	**G**raphics **P**rocessing **U**nit
IoU	**I**Intersection **o**ver **U**nion
mAP	**m**edian **A**verage **P**recision
NMS	**N**on **M**ax **S**upress
NPU	**N**eural **P**rocessing **U**nit
QPT	**Q**uantization **P**ost **T**raining
QAT	**Q**uantization **A**ware **T**raining
RNN	**R**ecursive **N**eural **N**etwork
ROI	**R**egion **o**f **I**nterest
SoC	**S**ystem **o**n **C**hip
SoM	**S**ystem **o**n **M**odul
SSD	**S**ingle **S**hot MultiBox **D**etector
SGD	**S**tochastic **G**radient **D**escent
TPU	**T**ensor **P**rocessing **U**nit
TFG	**T**rabajo **F**in **G**rado
YOLO	**Y**ou **O**nly **L**ook **O**nce

Dedicado a mi familia . . .

1 Introducción

1.1. Propuesta y justificación

En la actualidad, el uso de técnicas de inteligencia artificial, especialmente el aprendizaje profundo o Deep Learning, ha tenido un impacto transformador en la resolución de problemas complejos relacionados con la visión por computadora en diversos ámbitos industriales. Estas técnicas han demostrado ser eficaces en aplicaciones como sistemas de vigilancia, seguridad y control de calidad en líneas de producción.

Sin embargo, la implementación del aprendizaje profundo en el contexto industrial y de automatización presenta desafíos significativos. Uno de los principales desafíos es el alto consumo de recursos que requiere el aprendizaje profundo, debido a la gran cantidad de parámetros y la potencia de cálculo necesaria para su ejecución.

En este campo, donde la eficiencia y la optimización de recursos son fundamentales, el consumo excesivo de los mismos puede obstaculizar la adopción de técnicas de inteligencia artificial. Los sistemas de automatización, como las líneas de producción y los sistemas de control de calidad, requieren un procesamiento rápido y eficiente de la información en tiempo real. La implementación del aprendizaje profundo puede requerir una infraestructura costosa y potente, lo que afecta la viabilidad y rentabilidad de su aplicación. Además, se trabaja con grandes volúmenes de datos, lo que plantea desafíos adicionales en términos de velocidad de procesamiento y capacidad de respuesta. Esto puede afectar significativamente el funcionamiento correcto del proceso en ejecución, lo cual puede resultar contraproducente.

Es crucial abordar estos desafíos y encontrar soluciones que permitan aprovechar los beneficios del aprendizaje profundo en el contexto industrial y de automatización, al mismo tiempo que se optimiza el consumo de recursos y se garantiza una ejecución eficiente y rápida de las tareas de procesamiento de datos. Por lo tanto, es necesario explorar nuevas formas de optimizar los recursos sin comprometer la precisión ni la velocidad de ejecución.

Hasta hace algunos años, la solución predominante consistía en ejecutar el procesamiento en la nube, lo cual no tenía en cuenta el consumo de recursos. Esto generaba una mayor carga en las redes y una mayor latencia en el procesamiento de información, limitando la capacidad de respuesta en tiempo real y afectando la eficiencia operativa.

En este sentido, resulta interesante investigar la viabilidad de utilizar FPGA (Field-Programmable Gate Array) para aplicar el aprendizaje profundo en la visión por computadora en el contexto industrial. Las FPGA permiten implementar algoritmos de aprendizaje profundo directamente en hardware especializado, lo que puede llevar a una ejecución más eficiente y rápida de las tareas de procesamiento de imágenes. Además, tienen un tamaño reducido, lo que permite integrarlas en sistemas

embebidos, lo que las hace aún más interesantes en el campo de la automatización industrial.

Una posible solución es utilizar FPGA como sistema de edge computing (es un enfoque de computación distribuida donde el procesamiento de datos ocurre cerca del punto de generación de los mismos). De esta manera, se busca superar estas limitaciones, al realizar el procesamiento cerca de la fuente de datos, lo que reduce la latencia y mejora la capacidad de respuesta.

1.2. Aportación

Alcance y Objetivo

El objetivo principal de este estudio es demostrar que el edge computing utilizando una FPGA es adecuado para desplegar técnicas de aprendizaje profundo (en este caso, para VA - Visión Artificial) en entornos industriales que requieren tiempo real. Se realizarán pruebas comparativas entre la FPGA y la GPU, utilizando un modelo entrenado como YOLOv4. Se medirán métricas clave como precisión de detección, tiempo de procesamiento y consumo de energía. Estas métricas permitirán comparar y analizar las ventajas y desventajas de cada plataforma en términos de rendimiento y eficiencia. El objetivo final es demostrar la viabilidad y eficiencia de las FPGAs en realizar tareas de detección en el edge computing en comparación con las GPUs.

Aplicación

La aplicación de este estudio en detección y aceleración de procesos en entornos industriales mediante el uso de FPGAs en el edge computing ofrece beneficios significativos, como detección en tiempo real, mayor eficiencia energética y mejora en la capacidad de respuesta. Al procesar los datos directamente en la FPGA, se reduce la latencia y la dependencia de la conectividad de red, lo que optimiza los procesos industriales. Esto permite una mayor productividad, calidad y eficiencia operativa en áreas como control de calidad, gestión de inventario y detección de anomalías.

2 Estado del Arte

Dentro del contexto de la optimización de recursos, se encuentran diversas técnicas que pueden contribuir a mejorar el rendimiento de la detección en el borde (edge). Estas técnicas se pueden clasificar en las siguientes categorías:

Optimización del modelo: Esta categoría engloba las técnicas relacionadas con el modelo en sí, como su entrenamiento, configuración de hiperparámetros y modificaciones en la arquitectura o el modelo destinadas a reducir el consumo de recursos.

Optimización del hardware: En esta categoría se incluyen las técnicas que se centran en el hardware. Esto puede implicar la selección de componentes más eficientes en términos de consumo energético, capacidad de procesamiento y memoria.

2.1. Técnicas de entrenamiento de redes

Para algunas aplicaciones es interesante ejecutar el entrenamiento en el edge pero esto requiere capacidad computacional, por lo que en la actualidad se utilizan algunas técnicas para salvar estas dificultades. Algunas de las más usadas son las siguientes

Federated Learning

El aprendizaje federado[1], [2] es un enfoque distribuido donde varios dispositivos de borde colaboran para entrenar un modelo global sin compartir sus datos locales. Se utiliza un modelo global inicial que se entrena localmente en cada dispositivo. Los pesos del modelo se envían a un servidor centralizado que los combina y actualiza el modelo global. Es beneficioso para aplicaciones con privacidad y sistemas basados en IoT.

Transferencia de Aprendizaje Profundo (DTL)

La transferencia de aprendizaje profundo [3], [4] utiliza conocimiento de modelos pre-entrenados para tareas nuevas. Se parte de un modelo pre-entrenado y se ajusta para la tarea específica. Ayuda a reducir el tiempo y recursos de entrenamiento en dispositivos de borde con limitaciones.

Destilación de conocimiento (Knowledge Distillation)

La Knowledge Distillatio [4], [5] transfiere el conocimiento de un modelo grande a uno más pequeño. El modelo grande se pre-entrena, mediante diferentes técnicas, se destila para mejorar la precisión del modelo pequeño en dispositivos con recursos limitados.

2.2. Redes neuronales

Otra manera de reducir el coste computacional es utilizar modelos que requieran menos recursos. En el campo de la detección de objetos, el estado del arte ha avanzado significativamente en los últimos años, gracias al desarrollo de diversas técnicas y modelos de redes neuronales. Estas técnicas tienen como objetivo lograr un equilibrio entre la precisión, el consumo y la velocidad de detección, especialmente en dispositivos con recursos limitados. A continuación, se presentan algunos de los modelos más relevantes:

SSD

SSD (Single Shot MultiBox Detector)[6] es una red neuronal utilizada para detectar y clasificar objetos en una imagen en una sola pasada. Aportando una alta precisión y velocidad razonable, lo que la hace ideal para aplicaciones en tiempo real en dispositivos con recursos limitados.

Este modelo se compone de una columna vertebral (backbone) y una cabeza (head), para detectar divide la imagen en regiones y colocando cajas de anclaje (anchor boxes) predefinidas con diferentes tamaños y relaciones de aspecto en esas regiones. Luego, utiliza una red neuronal para detectar y clasificar objetos dentro de cada caja. Es una técnica eficiente y rápida que se utiliza en aplicaciones en tiempo real para la detección de objetos. Su arquitectura se detalla en Fig.2.1.

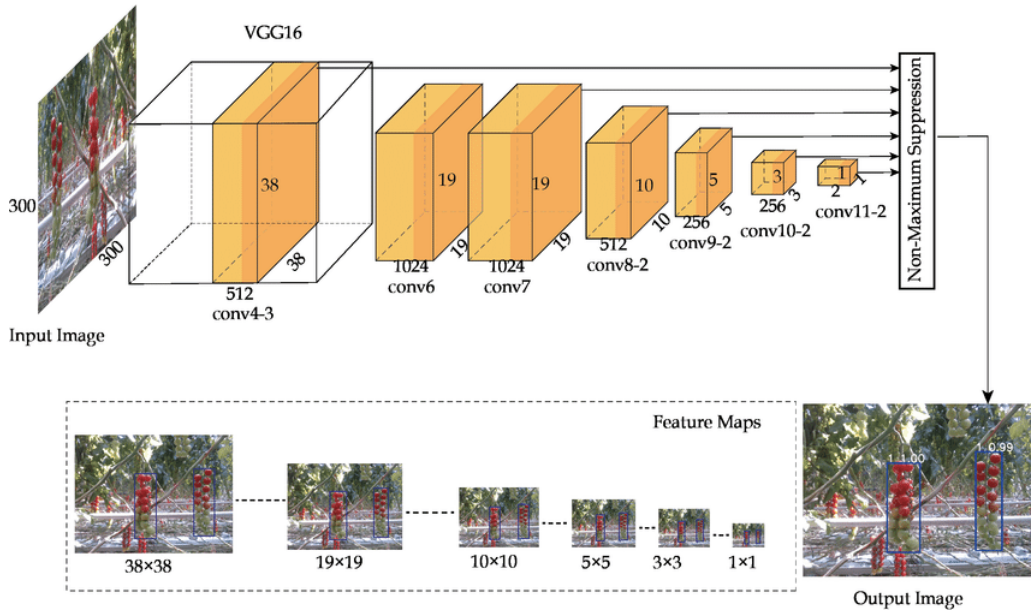


FIGURA 2.1: Arquitectura basica de SSD [7]

El 'backbone', suele ser una red neuronal convolucional preentrenada en clasificación de imágenes, se utiliza como extractor de características. Esta red es capaz de extraer el significado semántico de la imagen de entrada mientras preserva la estructura espacial de la imagen a una resolución más baja. Por ejemplo, se puede utilizar una red como ResNet [8], entrenada en el conjunto de datos ImageNet, eliminando la capa de clasificación totalmente conectada final. Esto da como resultado una red

neuronal profunda que puede extraer el significado de la imagen de entrada a una resolución reducida, como mapas de características de 7x7.

La 'head' de SSD consiste en una o varias capas convolucionales adicionales conectadas a la columna vertebral. Estas capas convolucionales se encargan de interpretar las salidas de la columna vertebral y generar las cajas delimitadoras y las clases de los objetos en las ubicaciones espaciales de las activaciones finales. Cada caja de anclaje en una región determinada de la imagen es responsable de predecir la clase y la ubicación de un objeto dentro de esa región.

MobileNet

MobileNet es una arquitectura de red neuronal diseñada especialmente para dispositivos móviles [9]. Para lograr su eficiencia y uso óptimo de recursos, MobileNet emplea el concepto de convolución en profundidad separable (depthwise separable convolution). Esta técnica descompone una convolución en dos etapas separadas: la convolución espacial y la convolución en profundidad.

En la etapa de convolución espacial, se aplica un filtro a cada canal de la imagen de entrada de forma separada. Esto permite capturar las correlaciones espaciales en la imagen, es decir, las relaciones entre los píxeles vecinos dentro de cada canal, generando un mapa de características.

Luego, en la etapa de convolución en profundidad, se ejecuta una convolución lineal sobre los mapas de características obtenidos en la capa anterior. Esta convolución en profundidad ayuda a combinar las características extraídas de diferentes canales, permitiendo una representación más rica y compleja de la información.

Al separar la convolución en dos etapas, MobileNet reduce significativamente la cantidad de cálculos necesarios en comparación con las convoluciones tradicionales. Esto hace que la inferencia en MobileNet sea altamente eficiente en términos de recursos computacionales, lo cual es especialmente importante en dispositivos móviles y embebidos donde se busca optimizar el rendimiento y la eficiencia energética. Se puede ver la comparativa con otros modelos en la Tab. 2.1

Una de las versiones mejoradas de MobileNet es **MobileNetV3** [10], que se presentó en 2019, introduce bloques de optimización de estructura llamados "búsqueda de arquitectura", lo que permite que el propio algoritmo o sistema explore y descubra automáticamente las mejores configuraciones de arquitectura para una tarea específico. También combina operaciones de convolución en profundidad separable con operaciones de convolución de punto fijo, lo que implica que los valores numéricos se representan con una cantidad fija de bits, lo que permite una representación más compacta y eficiente en términos de memoria y cálculos.

Una versión adicional enfocada en la velocidad de detección y la reducción del consumo es **MobileNet-Tiny** [11], fue implementada y desplegada en una Raspberry Pi, su arquitectura se detalla en Fig. 2.2. La arquitectura de detección se basa en dos componentes principales: SSD-Lite y MobileNetV2. SSD-Lite es una versión optimizada del detector SSD que utiliza menos recursos sin comprometer la precisión. Por su parte, MobileNetV2 es una mejora de MobileNet que incorpora bloques de activación lineal para una mejor discriminación y precisión, así como bloques residuales

TABLA 2.1: Comparación de MobileNet con modelos populares [9]

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6 %	569	4.2
GoogleNet	69.8 %	1550	6.8
VGG 16	71.5 %	15300	138
0.50 MobileNet-160	60.2 %	76	1.32
Squeezenet	57.5 %	1700	1.25
AlexNet	57.2 %	720	60

que permiten la propagación de información a través de conexiones de salto para capturar características complejas y aumentar la precisión.

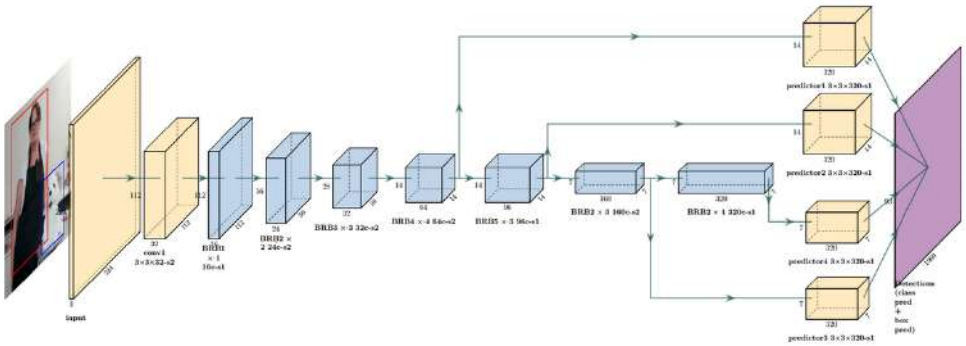


FIGURA 2.2: Arquitectura de MobileNet-Tiny. [11]

En cuanto al proceso de detección, se recibe una imagen RGB y se procesa mediante convoluciones y bloques residuales de bottleneck (BRB4, BRB5 y BRB6), generando un mapa de características. Este mapa se combina con otros mapas de características y se pasa a capas predictoras SSDLite para generar las detecciones, que luego se filtran mediante una capa de supresión de no máximo. De esta manera, se logra una detección precisa y eficiente de objetos en imágenes, manteniendo un equilibrio entre recursos computacionales y precisión.

SqueezeDet

SqueezeDet [12] es una red neuronal diseñada con CNNs y con una RNNs (recurrent neural networks, redes neuronales recurrentes) para lograr una detección y clasificación eficiente de objetos en dispositivos de baja potencia o edge. Para lograr una detección y clasificación eficiente reduce la cantidad de parámetros y operaciones,

utilizando capas de convolución 'squeeze' que se encarga de reducir la dimensionalidad de los datos de entrada al aplicar convoluciones 1x1 que actúan como filtros lineales para combinar características de diferentes canales en un espacio de menor dimensionalidad y 'expand' que se encarga de expandir la dimensionalidad de los datos aplicando convoluciones 1x1 seguidas de convoluciones 3x3 para generar una mayor variedad de características. Estas capas comprimen (en la capa squeeze) y luego expanden (en la capa expand) la información de las características de la imagen para realizar predicciones sobre la posición y clase de los objetos. El funcionamiento de detección se muestra en Fig. 2.3, que consiste en una CNN (Convolutian Neural Network) que extrae un mapa de características de la imagen de entrada y lo pasa a la capa ConvDet. Esta capa calcula cajas delimitadoras centradas alrededor de una cuadrícula distribuida uniformemente. Cada caja tiene una puntuación de confianza de si contiene o no objeto y una probabilidad de pertenencia a cada clase. Luego, se seleccionan las mejores cajas según su confianza y se aplica NMS para obtener la detección final.

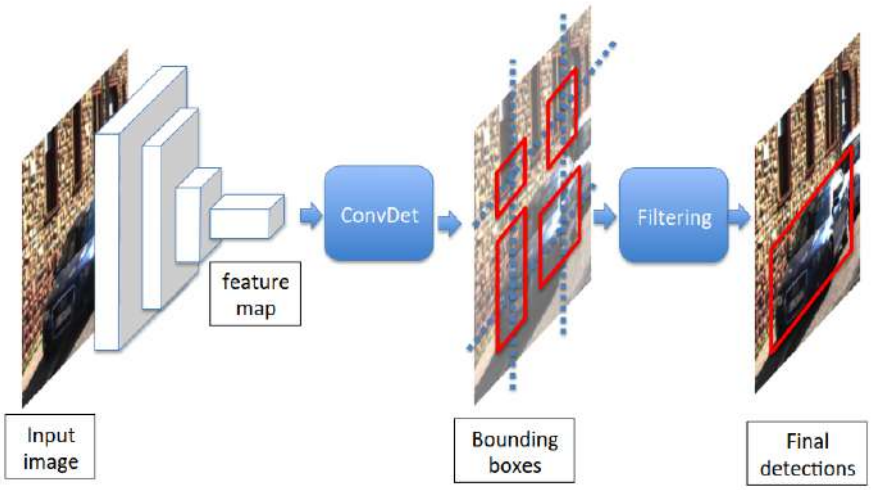


FIGURA 2.3: Pipeline de la detección de objetos con el modelo SqueezeDet. [12]

EfficientDet

EfficientDet[13] es una familia de redes neuronales basadas en EfficientNet[14]. Estas redes están diseñadas para ofrecer un rendimiento óptimo en términos de precisión y eficiencia computacional, siendo adecuadas para dispositivos edge.

La eficiencia de EfficientDet se logra mediante técnicas como la optimización de la arquitectura de red, el uso de bloques de convolución eficientes y la aplicación de técnicas de regularización y compresión del modelo. Estas técnicas permiten reducir el número de operaciones computacionales y los parámetros del modelo sin comprometer significativamente la precisión en la detección de objetos. Su arquitectura se detalla en Fig. 2.4. y conste en una red principal, BiFPN (que utiliza conexiones bidireccionales y un proceso iterativo para fusionar características a múltiples escalas, permitiendo una propagación de información más eficiente y un mejor manejo de escalas en la detección de objetos), como la red de características la cual a su salida

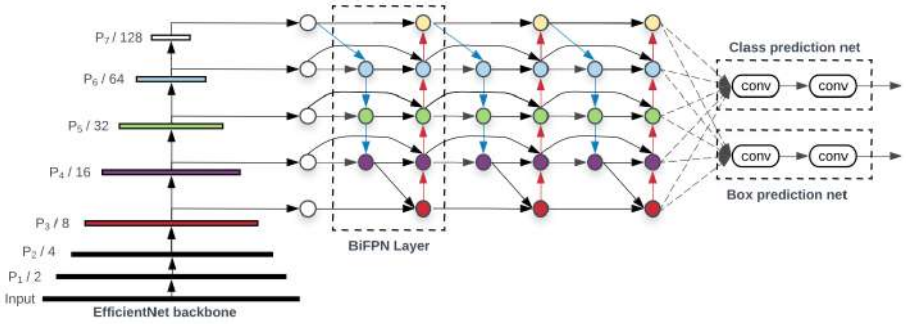


FIGURA 2.4: Arquitectura EfficientDet [13]

le pasa estas a una red de predicción de clases/cajas a través de una conexión total o fully connected. Tanto las capas de BiFPN como las capas de la red de predicción de clases/cajas se repiten varias veces según diferentes restricciones de recursos

RetinaNet

RetinaNet [15] es una red neuronal especializada en la detección de objetos. Está basada en Feature Pyramid Network (FPN). La principal fortaleza de RetinaNet es su capacidad para detectar tanto objetos pequeños como grandes con alta precisión. Esto se debe a su diseño y estructura interna, que permite capturar y analizar características de diferentes escalas en una imagen.

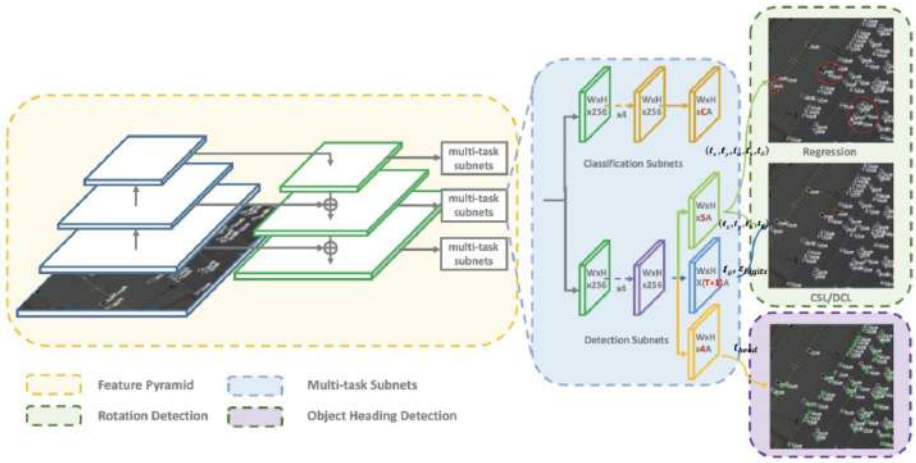


FIGURA 2.5: Arquitectura del RetinaNet [16]

RetinaNet utiliza una combinación de técnicas y capas especializadas para lograr su alto rendimiento en la detección de objetos. La red consta de tres componentes principales: una red de características que extrae información relevante de la imagen, una red de pirámide de características que captura detalles finos y contextos más amplios, y una red de predicción de clases/cajas que asigna clases y ajusta las cajas delimitadoras de los objetos. Utiliza una estrategia llamada "enfoque de anclaje focal" (focal anchoring approach) para asignar pesos a las regiones propuestas, lo que mejora la detección de objetos difíciles y garantiza un equilibrio en la detección de objetos de diferentes tamaños. Su arquitectura se detalla en Fig. 2.5.

Fast R-CNN

Fast R-CNN [17] (Region-based Convolutional Neural Network) es un enfoque popular para la detección de objetos que utiliza una red neuronal convolucional para procesar la imagen y generar características relevantes. Estas características se utilizan para crear propuestas de regiones que podrían contener objetos. Luego, se utiliza una red llamada Región de Interés (ROI) para clasificar y ajustar estas propuestas de regiones con el fin de detectar objetos precisamente.

La ventaja de Faster R-CNN es que combina la extracción de características y la detección de objetos en una sola arquitectura, lo que hace que el proceso sea más eficiente y preciso. Además, al utilizar una región de interés, se pueden seleccionar las áreas más relevantes de la imagen para realizar la detección, lo que mejora aún más la precisión.Su arquitectura se detalla en Fig.2.5.

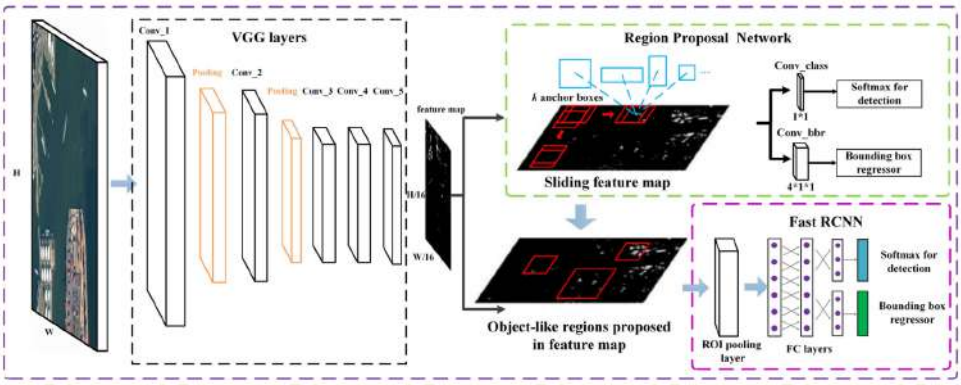


FIGURA 2.6: Arquitectura de la Faste R-CNN siendo el modelo VGG el backbone [18]

YOLO

YOLO (You Only Look Once) [19] Es un enfoque popular de detección de objetos en tiempo real que ofrece un buen equilibrio entre precisión y velocidad. Se adapta para funcionar en dispositivos con recursos limitados, lo que lo hace ampliamente utilizado en la detección de objetos. Su arquitectura se detalla en Fig.2.7.

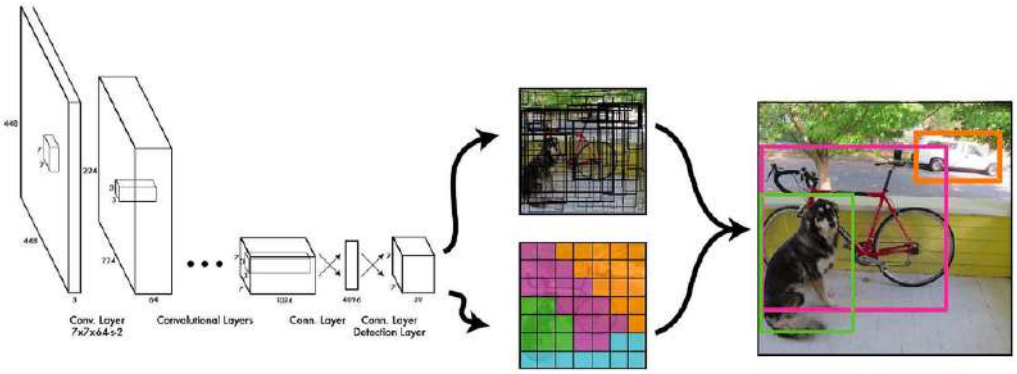


FIGURA 2.7: Arquitectura de YOLO [19]

YOLOv4[20] Es una versión mejorada de YOLO que se lanzó en 2020. Se enfoca en mejorar tanto la precisión como la velocidad de detección. Incorporando capas CSP-Darknet[21] (CSP: Cross-Stage Partial connections), consiste en dividir la red en dos rutas o etapas paralelas, una ruta principal y una ruta de derivación. La ruta principal se encarga de extraer características a través de convoluciones y otros procesamientos, mientras que la ruta de derivación se utiliza para agregar características de bajo nivel al final de la ruta principal), bloques PANet[20] (Path Aggregation Network, permiten fusionar características de diferentes escalas en diferentes niveles de la red, utiliza un enfoque de pirámide de características para fusionar características de diferentes niveles y escalas, lo que proporciona una representación) y SPP[22] (Spatial Pyramid Pooling,divide la imagen o las características en varias regiones a diferentes escalas y luego realiza el procesamiento de cada región por separado.), así como optimizaciones de red.

YOLOv5[23][24]: es una versión de YOLO que se lanzó en 2020. Esta versión se enfoca en ser más ligera y eficiente en términos de recursos computacionales. Se basa en la arquitectura de YOLOv4. En la que se han realizado ajustes y optimizaciones en la arquitectura para reducir aún más el tamaño del modelo para ello.

YOLOv6[25][26] es una versión mejorada del framework de detección de objetos YOLO. En esta versión, se han introducido mejoras en el cuello del detector, incorporando el módulo de Concatenación Bidireccional (BiC) para mejorar la precisión de la localización. También ha implementado una estrategia de entrenamiento asistida por anclas (anchors) y ha aumentado la profundidad de la red. Estas mejoras posicionan a YOLOv6 como una opción avanzada y eficiente en la detección de objetos.

YOLO-TINY[27]: Es una la versión compacta y ligera de YOLO diseñada para dispositivos con recursos limitados. Aunque sacrifica algo de precisión en comparación con las versiones completas, ofrece una inferencia más rápida y es más adecuada para aplicaciones que requieren baja latencia y recursos computacionales limitados. Existen para todas las versiones de YOLO mencionadas.

DAMO-YOLO [28][29][30]es un método de detección de objetos avanzado. Utiliza tecnologías como la Búsqueda de Arquitectura Neural (NAS), estructuras eficientes de cuellos y cabezas, asignación de etiquetas mejorada y destilación de modelos.

Comparativa

En las tablas Tab. 2.2 y Tab. 2.3 se presenta una comparativa de varios modelos de detección de objetos basada en los datos recopilados de múltiples fuentes empleadas [11], [12], [15], [26], [28], [31]-[33]. Estas tablas proporcionan una visión general de los resultados de rendimiento de los modelos en términos de precisión de las cajas delimitadoras (AP, mAP(median average preciśión) que se calcula con la eq. 3.3) sobre el dataset de COCO 2017[34]. En la Tab. A.1 se pueden observar más comparativas de modelos obtenidas de los modelos zoo de OpenVINO™[35]

COCO 2017 es un conjunto de datos ampliamente utilizado en visión por computadora y aprendizaje automático. Contiene imágenes etiquetadas con objetos comunes en diversos contextos. Consta de 80 categorías de objetos y anotaciones detalladas de ubicación y contorno. COCO 2017 es un recurso de referencia para tareas de

reconocimiento, detección y segmentación de objetos. Es valioso para el desarrollo de algoritmos de visión por computadora.

Es importante tener en cuenta que las referencias mencionadas proporcionan información adicional y detalles sobre los métodos y configuraciones específicas utilizadas en cada modelo. Consultar estas referencias puede brindar una comprensión más completa de los enfoques y técnicas utilizadas en el campo de la detección de objetos.

Estas tablas también nos permiten comprender las razones de la selección de modelos para realización de este TFG, puesto que se busca un modelo que tenga una buena relación con su coste computacional y precisión.

TABLA 2.2: Comparación de modelos en el dataset KITTI con una GPU TITAN X (C) coche, (B) Bicicleta, (P) Peón ,(T) todo [11], [12], [15]

Modelo	mAP_C	mAP_B	mAP_P	mAP_T	Model size (MB)	Speed (FPS)	FLOPs
FRCNN + VGG16[17]	86.0	-	-	-	485	1.7	-
FRCNN + AlexNet[17]	82.6	-	-	-	240	2.9	-
SqueezeDet [12]	82.9	76.8	70.4	76.7	7.9	57.2	9.7 G
SqueezeDet+ [12]	85.5	82.0	73.7	80.4	26.8	32.1	77.2 G
VGG16-Det [12]	86.9	79.6	70.7	79.1	57.4	16.6	288.4 G
ResNet50-Det [12]	86.7	80.0	61.5	76.1	35.1	22.5	61.3 G

TABLA 2.3: Comparación de modelos en el dataset de COCO signos (+) NVIDIA Tesla T4 GPU (∗) Tesla V100. (−) i7-8565U[36] & Nvidia M40 con el entorno de TensorRT TRT [11], [12], [15], [26], [28], [31]-[33]

Method	Input Size	AP	AP_{50}	FPS_{50} (bs=1s)	FPS (bs=32s)	Latency (bs=1s)	Params	FLOPs
+TRT YOLOv5-N [24]	640	28.0 %	45.7 %	602	735	1.7 ms	1.9 M	4.5 G
+TRT YOLOv5-S [24]	640	37.4 %	56.8 %	376	444	2.7 ms	7.2 M	16.5 G
+TRT YOLOX-Tiny [33]	416	32.8 %	50.3 %*	717	1143	1.4 ms	5.1 M	6.5 G
+TRT YOLOX-S [33]	640	40.5 %	59.3 %*	333	396	3.0 ms	9.0 M	26.8 G
+TRT DAMO-YOLO-T [28]	640	42.0 %	58.0 %	-	397	2.78 ms	8.5 M	18.1 G
+TRT DAMO-YOLO-S [28]	640	46.0 %	61.9 %	-	325	3.83 ms	16.3 M	37.8 G
+TRT DAMO-YOLO-M [28]	640	49.2 %	65.5 %	101	127	5.62 ms	28.2 M	61.8 G
+TRT YOLOv7-Tiny [37]	416	33.3 %*	49.9 %*	787	1196	1.3 ms	6.2 M	5.8 G
+TRT YOLOv7-Tiny [37]	640	37.4 %*	55.2 %*	424	519	2.4 ms	6.2 M	13.7 G*
+TRT YOLOv6-N [26]	640	35.9 %	51.2 %	802	1234	1.2 ms	4.3 M	11.1 G
+TRT YOLOv6-T [26]	640	40.3 %	56.6 %	449	659	2.2 ms	15.0 M	36.7 G
+TRT YOLOv6-S [26]	640	43.5 %	60.4 %	358	495	2.8 ms	17.2 M	44.2 G
∗ YOLOv4 [20]	608	43.5 %	65.7 %	-	62.0	-	-	-
∗ YOLOv4-CSP FLops in Backbone y neck	640	47.5 %	66.2 %	-	73	-	53M	109B
∗ TRT YOLOv3 +ASFF* [38]	608	42.4 %	63.0 %	-	45.5	-	-	-
∗ TRT YOLOv3 - ultralytics2 [33]	640	44.3 %	64.6 %	-	95.2	-	63.00 M	157.3G
∗ EfficientDet-D0 [13]	512	33.8 %	52.2 %	-	98.0	-	3.9M	2.5B
∗ EfficientDet-D [13]1	640	39.6 %	58.6 %	-	74.1	-	6.6M	6.1B
− MobileNet-Tiny [11]	608	19 %	-	-	19	-	-	0.41G
& RetinaNet-50 [15]	500	32.5 %	-	-	-	73 ms	-	0.41G
& RetinaNet-101[15]	500	34.4 %	-	-	-	90 ms	57 M	273 B

2.3. Optimización de modelos

Además de seleccionar un modelo que consuma pocos recursos se pueden utilizar varios métodos para conseguir una optimización del mismo. A continuación, se explican los más relevantes [39].

2.3.1. Durante el entrenamiento

Stochastic Gradient Descent (SGD)

El descenso de gradiente estocástico (SGD) es un algoritmo de optimización utilizado para minimizar la función de costo en una red neuronal. A diferencia del descenso de gradiente por lotes, el SGD actualiza los pesos y sesgos después de cada dato individual, lo que lo hace especialmente adecuado para su implementación en dispositivos con recursos limitados [40]. El SGD ha sido ampliamente utilizado en el entrenamiento de redes neuronales debido a su eficiencia y capacidad para evitar mínimos locales. Este enfoque de optimización ha demostrado ser efectivo en una variedad de tareas de aprendizaje profundo.

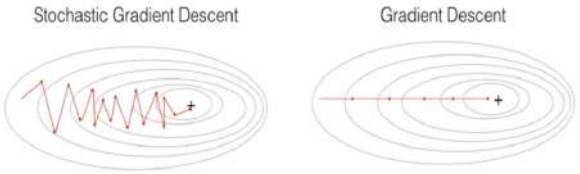


FIGURA 2.8: Stochastic Gradient Descent (SGD) vs Gradient Descent (GD)

Regularización

La regularización es conjunto de técnica utilizada para abordar el problema del sobreajuste en los modelos de redes neuronales[41]. Se han explorado diferentes métodos de regularización, incluyendo la regularización L1 (Lasso. Least Absolute Shrinkage and Selection Operator,penaliza la suma del valor absoluto de los coeficientes del modelo, establece coeficientes exactamente a cero), la regularización L2 (Ridge, penaliza la suma de los cuadrados de los coeficientes del modelo no establece coeficientes exactamente a cero, sino que los reduce significativamente.) y la técnica de dropout (aleatoriamente "apaga" , es decir establece a cero, un porcentaje de las unidades (neuronas) en cada capa oculta). Estas técnicas ayudan a reducir la complejidad del modelo y mejorar su capacidad de generalización al introducir restricciones en los pesos y sesgos. La regularización ha demostrado ser eficaz para mejorar el rendimiento de los modelos de aprendizaje profundo y evitar el sobreajuste.

Compartir peso, Weight Sharing

El weight sharing es una técnica que reduce la redundancia en los pesos de una red neuronal, permitiendo que múltiples conexiones compartan el mismo peso. Siendo efectiva para reducir la cantidad de pesos almacenados y reconstruir el modelo completo utilizando un número menor de pesos efectivos [42]. Se puede realizar el weight sharing de forma aleatoria o utilizando una función de hash para agrupar los pesos [43], [44] y además utilizando técnicas como la codificación Huffman [45], [46]

los pesos se pueden reducir al comprimirlos. Esta técnica ha sido utilizada en diversos enfoques y ha demostrado su eficacia en la reducción del tamaño del modelo.

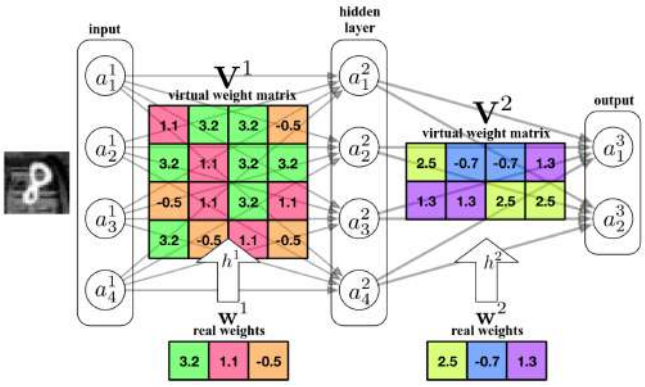


FIGURA 2.9: Ejemplo de Weight Sharing con un factor de compresión $\frac{1}{4}$ [43]

2.3.2. Ajuste de hiperparámetros

En el aprendizaje automático, el ajuste de hiperparámetros es crucial para obtener un rendimiento óptimo del modelo. Hay varios métodos para ajustar los, como la búsqueda en cuadrícula, la búsqueda aleatoria y la optimización bayesiana. La búsqueda en cuadrícula realiza una búsqueda exhaustiva de todas las combinaciones posibles de valores, mientras que la búsqueda aleatoria selecciona valores al azar. La optimización bayesiana utiliza modelos de probabilidad para encontrar los mejores valores basándose en las evaluaciones previas. También existen herramientas como: scikit-learn[47], [48], Scikit-optimize [49], BayesianOptimization[50], [51], GPyOpt[52], Hyperopt[53], Keras [54], Ray Tune [55], Optuna[56], MOE[57] y Spearmint[58], [59] que facilitan el proceso de ajuste de hiperparámetros. Es importante validar los hiperparámetros optimizados mediante técnicas como la validación cruzada. El ajuste de hiperparámetros busca encontrar un equilibrio entre el sesgo y la varianza del modelo.

2.3.3. Después del entrenamiento

Pruning

El Pruning (Poda) es una técnica utilizada para reducir el tamaño de los modelos de redes neuronales al eliminar conexiones o neuronas no importantes. Existen dos tipos de poda: la poda no estructurada, que simplemente elimina neuronas, y la poda estructurada, que también elimina conexiones, pesos y canales asociados [60]. Al eliminar las conexiones o neuronas menos importantes, el modelo resultante se vuelve más eficiente en términos de memoria, energía y tiempo de cálculo, sin una pérdida significativa de precisión [61]. La poda puede realizarse considerando el valor de los pesos, la correlación de las salidas de las neuronas o mediante el uso de filtros no importantes [62], [63]. La evaluación de la poda se realiza en términos del tamaño del modelo, la precisión y el tiempo de cálculo [64].

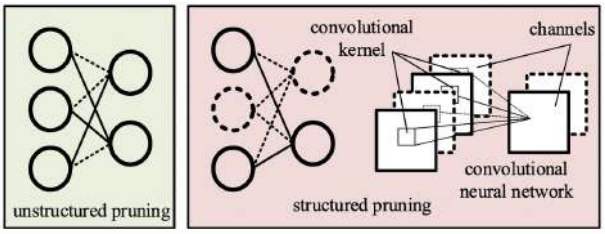
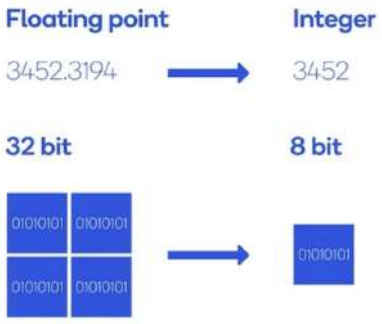


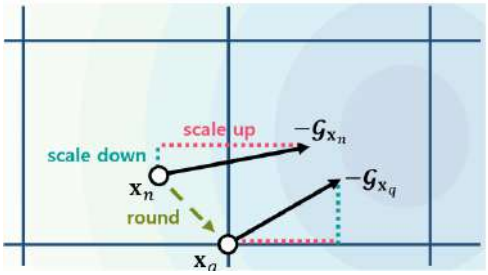
FIGURA 2.10: Poda no estructurada vs Poda estructurada [60]

Quantización

La quantización implica reducir la precisión de los pesos, parámetros y activaciones de una red neuronal para ahorrar memoria al pasar valores de punto flotante de 32 bits a una precisión menor como enteros de 8 bits, aplicando técnicas como la quantización de escala y la granularidad para ajustar el rango de los valores [65], Esta técnica se suele emplear después de entrenar el modelo (post-training quantization) aunque también se puede realizar durante el entrenamiento (QAT Quantization aware training). El estudio [65] muestra que la precisión no se ve afectada significativamente al quantizar. Se han propuesto métodos como la aproximación binaria y el reentrenamiento por grupos para minimizar el error de quantización [66], [67]. Otras estrategias proponen dividir los pesos de una capa en dos partes, una parte de los pesos se aproxima a valores de bajo número de bits, mientras que la otra parte se mantiene con la precisión original de 32 bits en punto flotante y se vuelve a entrenar para minimizar la pérdida de precisión causada por la quantización de la primera parte [68].



(A) Ejemplo de quantización



(B) Propagación de gradiente Element-wise gradient scaling [69]

Además, se ha propuesto el escalado de gradientes, **Gradient Scaling in Network Quantization** [70] propaga el gradiente sin considerar los errores de discretización, esta técnica ajusta de forma adaptativa cada elemento del gradiente en contra parte de SGD. En el paper [69] se ha desarrollado un método novedoso que permite entrenar modelos de aprendizaje profundo utilizando representaciones de 4 bits. Este enfoque, que incluye la quantización de enteros y el escalado de valores de gradiente, ha demostrado una aceleración significativa de más de 7 veces en comparación con la representación de punto flotante de 16 bits.

2.4. Dispositivos hardware

En las últimas décadas, el mercado industrial ha reconocido la creciente necesidad de desarrollar nuevos dispositivos edge que permitan procesar grandes cantidades de datos de manera eficiente. Estos dispositivos se han vuelto cada vez más importantes en el ámbito de la detección en la industria y la automatización, ya que abordan los desafíos de consumo excesivo de recursos y ralentización del sistema al realizar el procesamiento en la nube.

A medida que las aplicaciones industriales requieren capacidades de procesamiento más robustas y en tiempo real, se ha producido un avance significativo en el desarrollo de dispositivos edge. Estos dispositivos están diseñados para realizar tareas de procesamiento y análisis de datos cerca de la fuente de origen sin la necesidad de enviarlos a la nube, lo que reduce la latencia y mejora la capacidad de respuesta en tiempo real.

A continuación, enumeraré algunos de los dispositivos más interesantes en el contexto de este Trabajo de Fin de Grado (TFG) relacionado con el tema:

GPU, Jetson nvidia.

La serie Jetson de NVIDIA [71] ofrece dispositivos de computación en el edge con potentes capacidades de procesamiento de visión por computadora. Por ejemplo, el Jetson Nano es un sistema en un módulo (SoM) que incorpora una GPU NVIDIA y está diseñado para aplicaciones de IA y visión en dispositivos pequeños y de bajo consumo, como robots y cámaras inteligentes. El Jetson Xavier NX es un dispositivo de la serie Jetson que cuenta con una GPU NVIDIA de alto rendimiento, diseñado para tareas de visión por computadora en aplicaciones industriales, médicas y de transporte. Las especificaciones se pueden observar en la Tab. 2.4

VPU, Intel Movidius Neural Compute Stick

El Intel Movidius Neural Compute Stick es un dispositivo de aceleración de inteligencia artificial (IA) desarrollado por Intel. Es una unidad de procesamiento de visión (VPU, Video Process Unity) compacta y portátil que permite a los desarrolladores ejecutar cargas de trabajo de aprendizaje profundo en dispositivos de borde.

El Neural Compute Stick se basa en la tecnología de Myriad, un procesador de visión diseñado específicamente para aplicaciones de IA. Ofrece un alto rendimiento y una baja potencia, lo que lo convierte en una solución ideal para aplicaciones de inteligencia artificial en dispositivos de borde con recursos limitados.

Una de las principales ventajas del Neural Compute Stick es su capacidad para realizar inferencias de IA de manera eficiente y rápida sin requerir una conexión a la nube. Esto permite que los dispositivos de borde, como cámaras de seguridad, drones y sistemas de videovigilancia, ejecuten algoritmos de aprendizaje profundo en tiempo real y tomen decisiones de manera autónoma.

La Raspberry Pi + Intel, junto con la unidad de procesamiento de visión (VPU) Intel NCS 2, ofrece una solución de bajo costo y alto rendimiento para aplicaciones de visión en el edge. La Raspberry Pi es una placa de computadora de tamaño reducido que se puede utilizar en una amplia gama de proyectos. Al combinarla con la VPU

Intel NCS 2, se mejora su capacidad de procesamiento de visión, lo que permite realizar tareas como detección de objetos, reconocimiento facial y análisis de vídeo en tiempo real. Esto es especialmente útil en aplicaciones de IoT, automatización del hogar y sistemas de seguridad. Las especificaciones se pueden observar en la tabla Tab. 2.4

TPU, Cámara inteligente

Las cámaras inteligentes equipadas con una unidad de procesamiento tensorial (TPU, Tensor Process Unity) son dispositivos diseñados para aplicaciones de visión en el edge. Por ejemplo, la cámara Google Nest Cam IQ o la Axis Q1615-LE Mk III utiliza una TPU integrada para realizar el procesamiento de vídeo y análisis de imágenes en el propio dispositivo, lo que permite funciones avanzadas como detección de personas, seguimiento de objetos y alertas inteligentes. Estas cámaras son ideales para sistemas de seguridad residencial o empresarial, monitorización de espacios públicos y aplicaciones de reconocimiento facial. Las especificaciones de la cámara Axis Q1615-LE [72] Mk III se pueden observar en la Tab. 2.4

TPU, Google Coral Edge

Google Coral Edge es una plataforma de aceleración de inteligencia artificial (IA) desarrollada por Google. Está compuesta por hardware y software diseñados específicamente para habilitar el procesamiento de IA en dispositivos de borde. El componente central de la plataforma es la GPU TPU de Google Coral, que ofrece un rendimiento excepcional y una eficiencia energética destacada. Las especificaciones se pueden observar en la Tab. 2.4

La GPU TPU de Google Coral está optimizada para cargas de trabajo de IA, como reconocimiento de imágenes, detección de objetos y procesamiento de lenguaje natural. Proporciona una potencia de cálculo intensiva para realizar inferencias de IA en tiempo real en dispositivos de borde, sin depender de una conexión a la nube. Esto permite que los dispositivos, como cámaras de seguridad, sistemas de videovigilancia y robots, realicen tareas de IA en el mismo lugar donde se generan los datos, lo que garantiza una mayor privacidad y una menor latencia.

NPU, IMX8 family

La NPU (Neural Process Unity) de la familia IMX8 [73] es un componente clave en los procesadores de aplicaciones de la familia i.MX8, desarrollados por NXP Semiconductors. Esta unidad especializada está diseñada específicamente para acelerar y optimizar las operaciones relacionadas con redes neuronales, brindando un rendimiento eficiente en aplicaciones de inteligencia artificial en dispositivos de borde.

La NPU de la familia IMX8 desempeña un papel fundamental al proporcionar capacidades de inferencia de IA en tiempo real en dispositivos como automóviles, sistemas industriales, robots y dispositivos inteligentes. Al integrar una NPU en los procesadores i.MX8, NXP ofrece a los desarrolladores una solución potente y eficiente para ejecutar tareas de aprendizaje automático en el dispositivo mismo, sin necesidad de depender de servicios en la nube.

Esta unidad especializada utiliza hardware optimizado para realizar operaciones intensivas de cálculo requeridas en las redes neuronales, como multiplicaciones de

matrices y funciones de activación. Esto permite que los dispositivos basados en la familia IMX8 realicen tareas complejas de inferencia, como el reconocimiento de imágenes, el procesamiento de voz y la detección de objetos en tiempo real, todo ello con una eficiencia energética destacada.

DPU, Xilinx kV260

Los dispositivos FPGA con DPU (Data Process Unity) son soluciones de procesamiento de visión en el edge altamente flexibles y personalizables. Por ejemplo, la placa Xilinx KV260[74], que combina un FPGA con una unidad de procesamiento de aprendizaje profundo (DPU). La placa Xilinx KV260 está diseñada específicamente para aplicaciones de visión y reconocimiento de imágenes en tiempo real. Su FPGA permite la implementación de algoritmos personalizados y optimizados para tareas de visión por computadora. Por lo que, la DPU integrada acelera el procesamiento de imágenes y la ejecución de modelos de aprendizaje profundo, proporcionando un rendimiento excepcional en aplicaciones exigentes.

El Xilinx Kria KV260 Vision AI Starter Kit es un sistema completo que combina el DPU con una plataforma de desarrollo y herramientas de software. Está especialmente diseñado para aplicaciones de visión artificial en tiempo real, como detección de objetos, seguimiento de objetos, reconocimiento facial y análisis de video. El kit ofrece una solución escalable y flexible para desarrolladores que desean implementar capacidades de IA avanzadas en dispositivos de borde [75].Las especificaciones se pueden observar en la Tab. 2.4

Xilinx utiliza técnicas de quantización y calibración para optimizar el rendimiento y la eficiencia energética de las tareas de inferencia de IA y para poder desplegarse en la DPU. El proceso de quantización reduce la precisión numérica requerida para los cálculos de la red neuronal, lo que ahorra recursos computacionales y permite una ejecución más rápida. La calibración ajusta los parámetros de la red neuronal para adaptarse a las características específicas del hardware y garantizar una precisión aceptable.

2.4.1. Comparativa

Para realizar la comparativa se utilizaron de referencia los siguientes estudios [74], [76]-[78]. Como se muestra en las tablas en un principio la FPGA es uno dispositivo que más rápido ejecuta los modelos y tiene un consumo relativamente bajo comparable a la Nvidia Jetson TX2[79] la cual tiene un tiempo de ejecución similar pero un precio mayor cerca de los 500$. Y comparado con el resto, aunque consumen menos tienen un tiempo de ejecución significativamente menor.

TABLA 2.4: Especificaciones de NVidia Jetson Xavier NX, Google Coral Mini ,K26 SOM de Xilinx,Raspberry Pi 4 B, Axis Q1615-LE Mk III, y Jetson Nano.[74], [76]-[78]

	Jetson Xavier NX	Coral Mini	Xilinx's K26 SOM
Processor	Hexa-core Carmel ARM v8.2 CPU	Quad-core ARM Cortex-A35	Quad-core Arm Cortex-A53 MPCore up to 1.5GHz
Accelerator	GPU (384 CUDA cores, 48 Tensor cores, 21 TOPs)	Edge TPU (Systolic array, 4 TOPs)	Mali-400 MP2 up to 667MHz ,DPU configurations B4096 @ 333MHz, (1.36TOPS)
Memory	8 GB LPDDR4x	2 GB LPDDR3	4GB 64-bit DDR4
Flash Memory	16 GB eMMC	8 GB eMMC	16GB
Supported Frameworks	TensorFlow Lite	-	Vitis-Ai y PYNQ (TensorFlow, PyTorch, and Caffe)
Networking	10/100/1000 BASE-T Ethernet, Wi-Fi 5, Bluetooth 5.0	-	10/100/1000 BASE-T Ethernet, 802.11a/b/g/n/ac 2×2 867Mb/s, Bluetooth 4.1
TDP	10-20 W	12.5-15 W	8-15 W
Price	$399	$99.99	$250
	Raspberry Pi 4 B	Axis Q1615-LE Mk III	Jetson Nano
Processor	ARM A72, Quad- Core @ 1.5 GHz e @ 1.5 GHz	ARTPEC-7 DL Accelerator	ARM A57, Quad-Core @ 1.43 GHz
Accelerator	Intel Movidius Myriad VPU (4000 GFLOPS)	Google Coral EdgeTPU (4000 GFLOPS)	NVIDIA (128-core Maxwell , 472 GFLOPs)
Memory	4 GB LPDDR4	8 MB SRAM	4 GB (shared)
Flash Memory	16 GB eMMC	-	-
Supported Frameworks	OpenVINO	TensorFlow Lite	TensorRT
Networking	-	-	10/100/1000 BASE-T Ethernet, Wi-Fi 5, Bluetooth 5.0
TDP	5-10 W	5-10 W	5-10W
Price	$110 €	$1250 €	$80

TABLA 2.5: Comparativa de Modelos Power: Xilinx K26 SOM, Nvidia Jetson Nano y Nvidia Jetson TX2 [74]

Modelo	Image Size	Xilinx K26 B3136 DPU	Xilinx K26 B4096 DPU	Nvidia Jetson Nano	Nvidia Jetson TX2
Inception V4	299x299	8.09	10.10	7.40	11.20
VGG-19	224x224	8.55	11.28	8.10	13.10
Tiny Yolo V3	416x416	8.26	11.08	7.80	12.30
ResNet-50	224x224	7.47	9.28	7.70	11.70
SSD Mobilenet-V1	300x300	7.67	9.29	7.30	10.80

TABLA 2.6: Comparativa de Modelos FPS: Xilinx K26 SOM, Nvidia Jetson Nano y Nvidia Jetson TX2[74]

Modelo	Image Size	Xilinx K26 B3136 DPU	Xilinx K26 B4096 DPU	Nvidia Jetson Nano	Nvidia Jetson TX2
Inception V4	299x299	19/19.1	30.3/30.4	11/13	24/32
VGG-19	224x224	17.9/17.9	17.4 / 17.4	10/ 12	23/29
Tiny Yolo V3	416x416	88.2 / 92.6	148.0 / 161.3	48 / 49	107/112
ResNet-50	224x224	49 / 49.1	75.6 / 75.9	37 / 47	84 112
SSD Mobilenet-V1	300x300	129.6 / 133.4	192.1 / 200.4	43 /48	92/109

TABLA 2.7: Comparison of Detection Network Performance Metrics for Jetson Nano, Jetson Xavier NX, and Coral Mini (Transposed). con el modelo YOLOv4-Tiny

	Accuracy (mAP)	Latency (ms)	Energy Efficiency
Jetson Nano	0.24	12.8	8.13
Jetson Xavier NX	0.29	7.3	13.62
Coral Mini	0.21	13.1	13.45

TABLA 2.8: Comparison of Detection Network Performance Metrics for Jetson Nano, Jetson Xavier NX, and Coral Mini (Transposed). con el modelo SSD MobileNet V2

	Accuracy (mAP)	Latency (ms)	Energy Efficiency
Jetson Nano	0.26	14.2	7.92
Jetson Xavier NX	0.29	10.1	9.47
Coral Mini	0.23	14.1	10.36

3 Metodología

3.1. Hardware

Durante el desarrollo del TFG, se utilizó un ordenador HP-Omenn como host para llevar a cabo tareas como el la quantización y la compilación de los modelos. El ordenador estaba equipado con una tarjeta gráfica NVIDIA GeForce RTX 3070 con tecnología Q-Max, lo que proporcionaba una capacidad de procesamiento adecuada para estas tareas.

Se utilizó la plataforma Colab para entrenar los modelos. Colab es un entorno de desarrollo en línea que proporciona recursos computacionales, como unidades de procesamiento gráfico (GPU), para llevar a cabo tareas intensivas en términos de cómputo, como el entrenamiento de modelos de aprendizaje automático.

Además del ordenador host, se contó con una placa Kria KV260 de Xilinx [74], [80] para realizar las evaluaciones y medir las métricas especificadas. La placa Kria KV260 es un dispositivo especializado diseñado para acelerar la inferencia de modelos de aprendizaje automático. Sus especificaciones detalladas se pueden encontrar en la Tab. 2.4.

3.2. Software

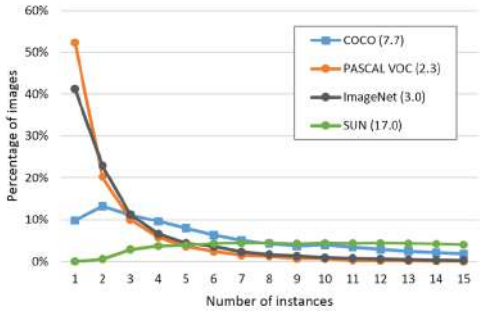
TensorFlow fue elegido como el framework principal de aprendizaje automático debido a su popularidad y su amplio conjunto de herramientas y funcionalidades. TensorFlow proporciona una interfaz intuitiva y flexible para construir y entrenar modelos de aprendizaje profundo. Además, cuenta con una gran comunidad de desarrolladores y una documentación extensa, lo que facilita su uso y resolución de problemas.

Para facilitar el entorno de desarrollo y garantizar la reproducibilidad de los resultados, se utilizó Docker. Docker es una plataforma de virtualización ligera que permite crear y gestionar contenedores de software. Xilinx proporciona imágenes de Docker en su repositorio oficial de DockerHub, lo que facilitó la configuración y el uso de las herramientas de Vitis-AI[81][82]. En particular, se utilizaron diferentes versiones de las imágenes de Docker de Xilinx, como la versión 3.0 para la quantización del modelo y la versión 2.5 para la compilación.

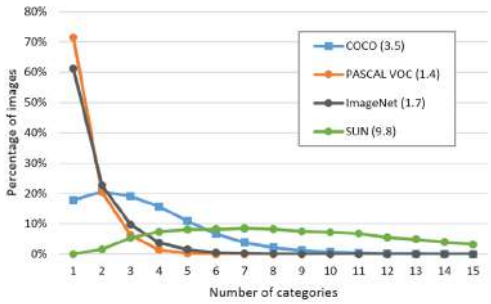
Además, se configuró el sistema operativo con los controladores de NVIDIA para aprovechar el poder de la GPU y utilizar CUDA. Se eligió la versión 12.1 de CUDA, que es compatible con las herramientas de Xilinx y permite una integración fluida entre el entrenamiento y la inferencia en la placa Kria KV260.

3.3. Dataset COCO

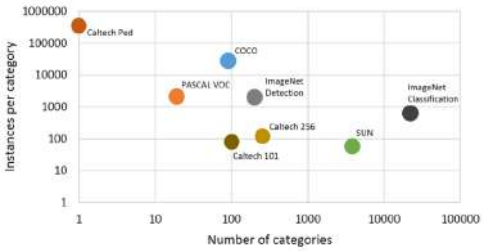
El conjunto de datos COCO (Common Objects in Context) se ha convertido en un referente en el campo de la visión por computadora debido a su amplio alcance y calidad de anotaciones. Contiene un total de más de 330,000 imágenes, que abarcan una variedad de escenarios y situaciones del mundo real. Estas imágenes están anotadas con información detallada, como categorías de objetos, descripciones de escenas y anotaciones de segmentación, lo que brinda una base sólida para una amplia gama de tareas de visión por computadora. En la Fig. 3.1 se muestra este datset en comparación con otros utilizados para la detección de objetos.



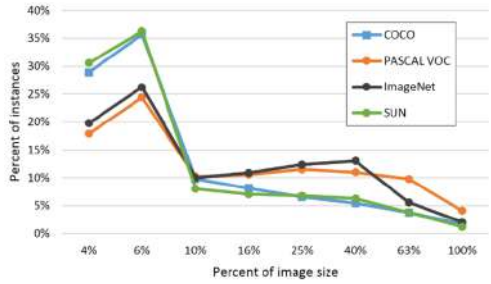
(A) instancias por imagen



(B) categorias por imagen



(C) Número de categorías vs. número de instancias por categoría



(D) tamaño de instancia

FIGURA 3.1: (a) Número de categorías anotadas y (b) número de instancias anotadas, por imagen,se muestra el promedio de categorías e instancias entre paréntesis). (c) Número de categorías versus número de instancias por categoría para varios conjuntos de datos populares de reconocimiento de objetos. (d) Distribución de tamaños de instancia para los conjuntos de datos (MS COCO, ImageNet Detection, PASCAL VOC y SUN) [34].

Una de las características clave de COCO es su diversidad de clases de objetos. El conjunto de datos incluye 80 categorías de objetos diferentes, que van desde personas, bicicletas y automóviles hasta animales, alimentos y muebles. Esta diversidad permite a los investigadores y desarrolladores abordar una amplia variedad de problemas de reconocimiento de objetos y detección de instancias. Cada imagen está etiquetada con múltiples anotaciones de objetos, lo que garantiza una cobertura exhaustiva de los diversos elementos presentes en la escena.

Sin embargo, una limitación importante del conjunto de datos COCO es el desequilibrio de clases. Algunas categorías de objetos tienen una presencia mucho más

prominente en comparación con otras. Por ejemplo, la clase "persona" puede tener una cantidad significativamente mayor de instancias en comparación con la clase "avión". Este desequilibrio puede afectar negativamente el rendimiento de los modelos de visión por computadora, ya que pueden sesgarse hacia las clases más frecuentes y tener dificultades para generalizar correctamente en las clases menos representadas. Se muestra en comparación a VOC en la Fig. 3.2

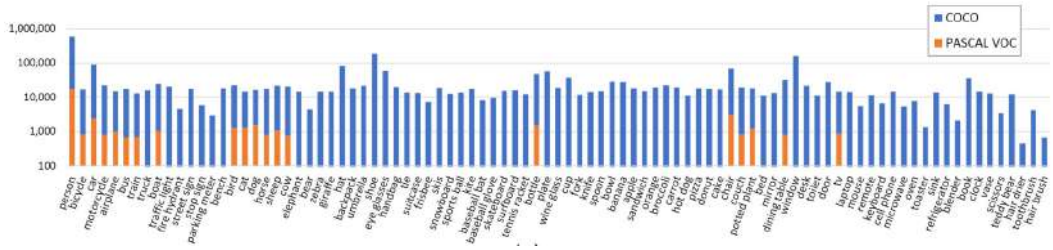


FIGURA 3.2: Número de instancias anotadas por categoría para MS COCO y PASCAL VOC [34].

En cuanto a la resolución de las imágenes, la relación de imágenes que más se repite en el conjunto de datos COCO es de 640 x 480 píxeles. Esto indica que la mayoría de las imágenes tienen una resolución de 640 píxeles de ancho por 480 píxeles de alto, en términos generales. Sin embargo, es importante tener en cuenta que las imágenes en el conjunto de datos pueden tener diferentes resoluciones y proporciones, ya que se trata de una colección diversa de imágenes recopiladas de varias fuentes.

El dataset se divide en 118.000/5.000 para entrenamiento/validación. Se utilizan exactamente las mismas imágenes y no se proporcionan nuevas anotaciones para detección/puntos clave. También cuenta con 120.000 imágenes no etiquetadas de COCO que siguen la misma distribución de clases que las imágenes etiquetadas; esto puede ser útil para el aprendizaje semi-supervisado en COCO.

3.4. Dataset

Obtención de datos

Se utilizó un sensor Sony IMX766 tipo CMOS con una resolución, 50 Mpx, apertura focal de f/1.8 para la recogida de datos. Las fotos de muestra para el dataset se realizaron en el entorno de trabajo designado para la implementación del modelo, con buena iluminación y sin distorsiones del ambiente.

Etiquetado de datos

En el proceso de etiquetado de las fotografías, se utilizó la herramienta VIA (VGG Image Annotator)[83] de manera online. VIA es una herramienta popular para el etiquetado de imágenes y se utiliza comúnmente en tareas de visión por computadora. Permite agregar etiquetas a las imágenes de manera interactiva y almacenar la información de anotaciones en un formato COCO (Common Objects in Context). En la Fig.3.4 se muestra como se realizó.

Con la herramienta VIA, se pudo seleccionar cada imagen y definir las cajas delimitadoras alrededor de los objetos de interés. Se asignaron etiquetas a cada caja, lo que

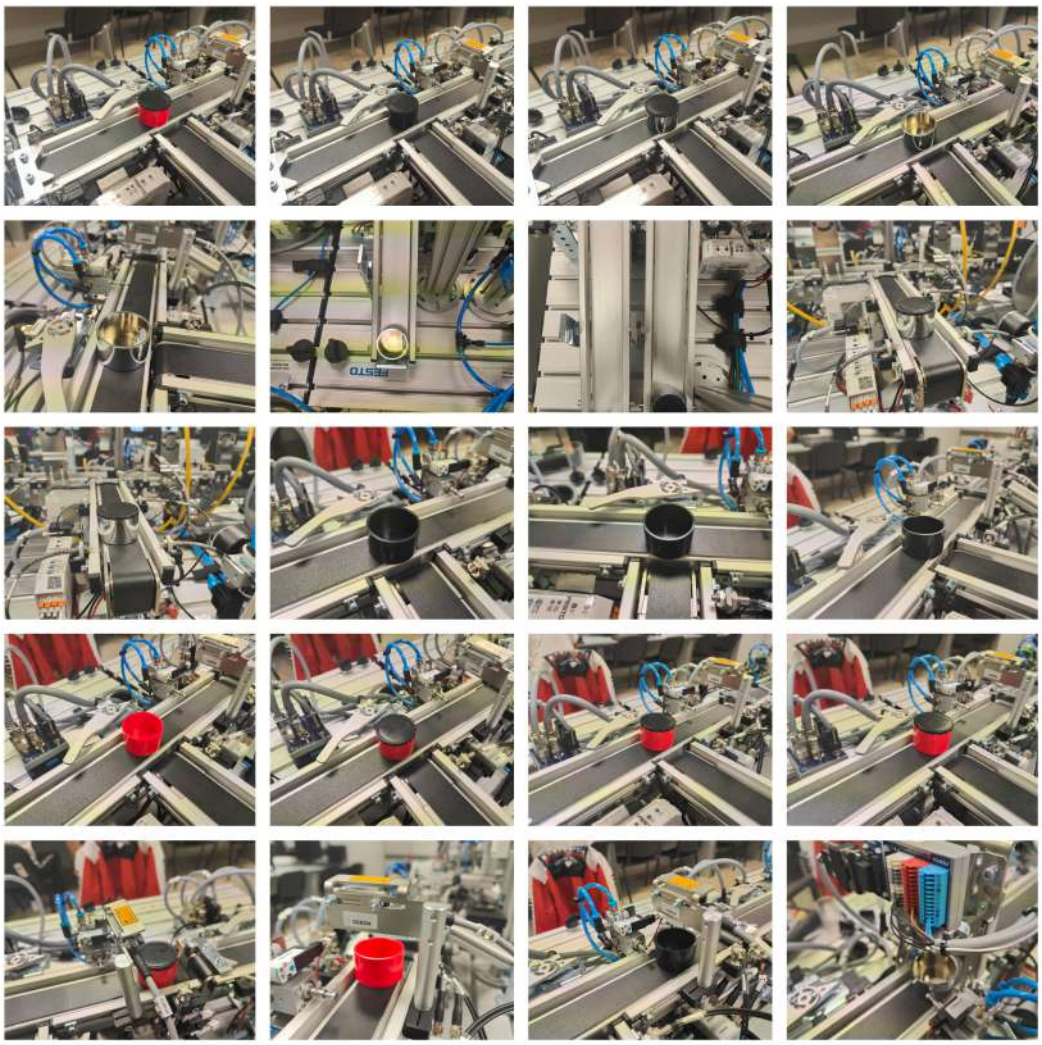


FIGURA 3.3: Imágenes de ejemplo del dataset

permitió identificar y categorizar los objetos presentes en las fotografías. Además de las cajas delimitadoras, también fue posible agregar otras anotaciones, como puntos de referencia, segmentaciones o cualquier otra información adicional requerida para abordar el problema específico.

Una vez completado el proceso de etiquetado en VIA, se generó un archivo JSON que contenía toda la información de las anotaciones realizadas en las imágenes. Este archivo COCO en formato JSON se utilizó para almacenar y organizar la información de manera estructurada, lo que facilitó su posterior procesamiento y análisis.

Características

El dataset está compuesto por un conjunto de 200 imágenes de 512x512 en la que hay 6 clases (cubo de metal, cubo rosa, cubo negro, cubo de metal con tapa, cubo rosa con tapa, cubo negro con tapa). Se empleo el 'data augmentation' rotando, en 45º, 90º y 180º; usando recortes (crops) y voltean (flips) las imágenes para dar un total resultante de 1000 imágenes.

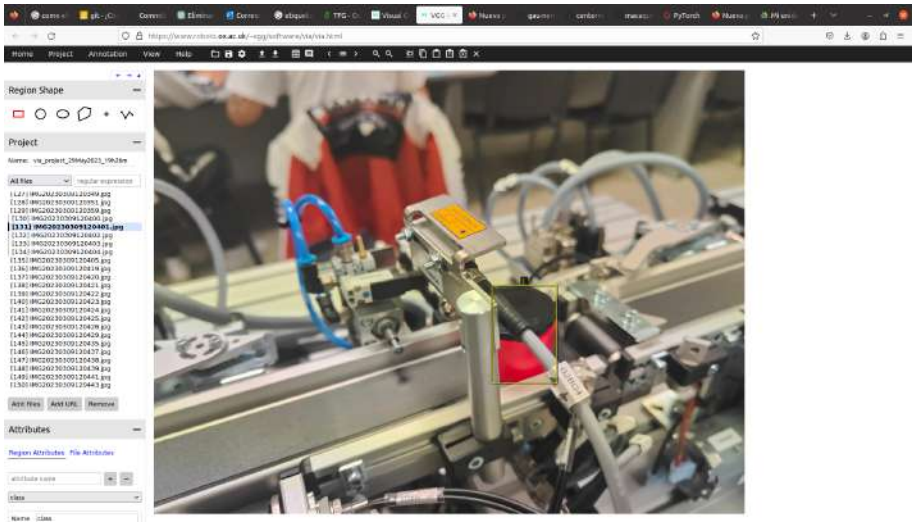


FIGURA 3.4: Imagen de ejemplo del etiqutado

3.5. Entrenamiento del modelo

3.5.1. Modelo seleccionado

- YOLOv4 se compone de:
  - Backbone:

    CSPDarknet53 [21] es una arquitectura basada en DenseNet que utiliza el patrón de conectividad densa para mejorar el flujo de gradientes en las capas. Consiste en dos bloques principales: una capa base de convolución y un bloque Cross Stage Partial (CSP).

    El bloque CSP divide el mapa de características en dos partes y las fusiona mediante una jerarquía entre etapas, lo que soluciona el problema del "Gradiente desvaneciente". La capa base de convolución es la entrada completa y el bloque CSP procesa la mitad de la entrada mientras que la otra mitad se pasa directamente al siguiente paso. Esto permite preservar características detalladas, fomentar la reutilización de características y reducir la cantidad de parámetros de la red.

    Solo el último bloque convolucional en la red base es denso, ya que agregar más capas densamente conectadas puede afectar la velocidad de detección. En general, CSPDarkNet53 optimiza el flujo de gradientes y mejora la eficiencia de la red al preservar características importantes.

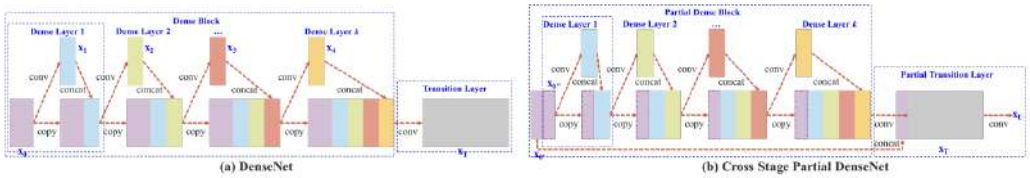


FIGURA 3.5: a) DenseNet and (b) Cross Stage Partial DenseNet (CSPDenseNet)[21].

  - Neck:

SPP [22] (Spatial Pyramid Pooling) entre el backbone CSPDarkNet53 y la red agregadora de características (PANet). Esto se realiza para aumentar el campo receptivo y separar las características de contexto más significativas.El campo receptivo se refiere al área de la imagen que es expuesta a un kernel o filtro en un instante dado. A medida que se apilan más capas convolucionales, el campo receptivo aumenta de forma lineal, pero se incrementa de manera exponencial al emplear convoluciones dilatadas.

PAN [84] suma las capas vecinas. PANet [20] concatena capas vecinas cuando se utiliza el agrupamiento de características adaptable. Se encarga de mejorar el proceso de segmentación de instancias al mantener la información espacial, lo que a su vez ayuda en la localización adecuada de píxeles para la predicción de máscaras.

- Head: YOLOv3 [85]

- YOLOv4 utiliza:
  - Bag of Freebies (BoF) para el backbone: aumentos de datos CutMix y Mosaic, regularización DropBlock, suavizado de etiquetas de clase
  - Bag of Specials (BoS) para el backbone: activación Mish, conexiones parciales de etapas cruzadas (CSP), conexiones residuales ponderadas de múltiples entradas (MiWRC)
  - Bag of Freebies (BoF) para el detector: pérdida CIoU, CmBN, regularización DropBlock, aumentos de datos Mosaic, entrenamiento autoadversario, eliminación de sensibilidad a la cuadrícula, uso de múltiples anclas para un solo valor de referencia, programador de enfriamiento cíclico coseno [20], hiperparámetros óptimos, formas de entrenamiento aleatorias
  - Bag of Specials (BoS) para el detector: activación Mish, bloque SPP, bloque SAM, bloque de agregación de ruta PAN, NMS DIoU

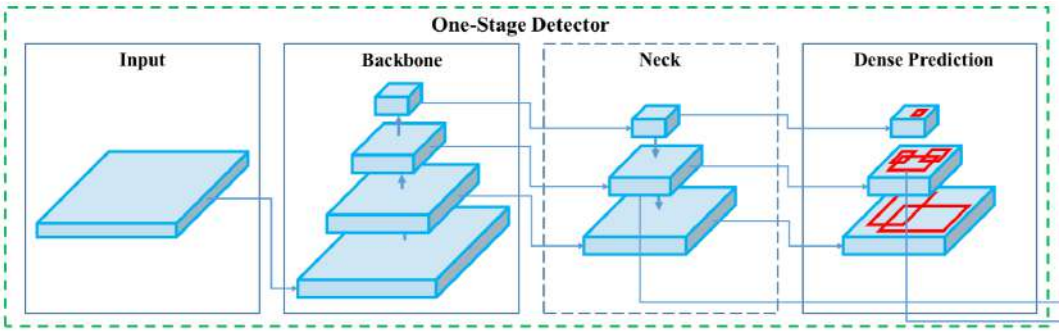


FIGURA 3.6: Arquitectura de Yolov4

Configuración seleccionada

YOLOv4 El modelo presentado en el artículo de YOLOv4 es el siguiente:

- Backbone: CSPDarknet53 con activación Leaky
- Neck: PANet con activación Leaky

- Módulos de complemento: SPP
- Predictor: YoloV3
- BFLOPs: 128.5@512x512

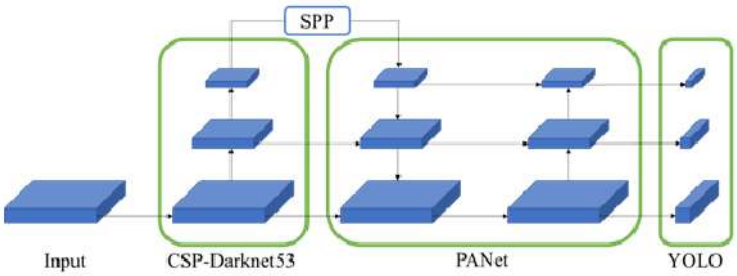


FIGURA 3.7: Arquitectura de Yolov4 selecionada [86]

3.5.2. Hiperparametros

Scikit-learn ha sido elegido como la herramienta principal para la configuración de hiperparámetros debido a su amplia adopción, su conjunto completo de algoritmos y utilidades, y su sólido respaldo por parte de la comunidad de aprendizaje automático. Esta elección garantiza una configuración eficiente y efectiva de los hiperparámetros, lo que conduce a modelos de mayor calidad y rendimiento. Los hiperparametros utilizados para el entrenamiento del modelo se muestran en la Tab. 3.1

Se modificaron las activaciones, original propuesta en el papear, 'mish' descripta en la eq. 3.1 por la activación 'leakyrelu' descripta en la eq. 3.2 para que pudieran ser soportas por la DPU.

Ecuación de activación 'mish':

$$f(x) = x \cdot \tanh(\ln(1 + e^x)) \tag{3.1}$$

Ecuación de activación 'leaky relu':

$$f(x) = \text{máx}(0{,}01 \cdot x, x) \tag{3.2}$$

TABLA 3.1: Configuración de hiperparámetros[87]

Hyperparameter	**Value**
Learning rate	0.001
Epochs	200
Optimizer	Adam
Batch size	8
Activation	leaky relu
Input image size	[512, 512, 3]

3.5.3. Entrenamiento

Se llevó a cabo el entrenamiento del modelo utilizando el dataset COCO 2017. El dataset COCO es ampliamente reconocido en la comunidad de investigación de visión por computadora y proporciona un conjunto de imágenes anotadas con múltiples categorías de objetos.

Para el entrenamiento se seleccionó la subdivisión de entrenamiento del dataset COCO 2017, que consta de aproximadamente 118,000 imágenes. Estas imágenes abarcan una amplia variedad de escenas y contienen una diversidad de objetos anotados. Antes de comenzar el proceso de entrenamiento, se redimensionaron las imágenes para que tuvieran una resolución uniforme de 512x512 píxeles.

En cuanto al enfoque de entrenamiento para el modelo de detección de cubos, se empleó la técnica de transfer learning, también conocida como aprendizaje de transferencia. Esta metodología se basa en la utilización de modelos previamente entrenados en tareas relacionadas y adaptarlos a un nuevo problema específico. En este caso, se utilizó un modelo preentrenado en un conjunto de datos COCO.

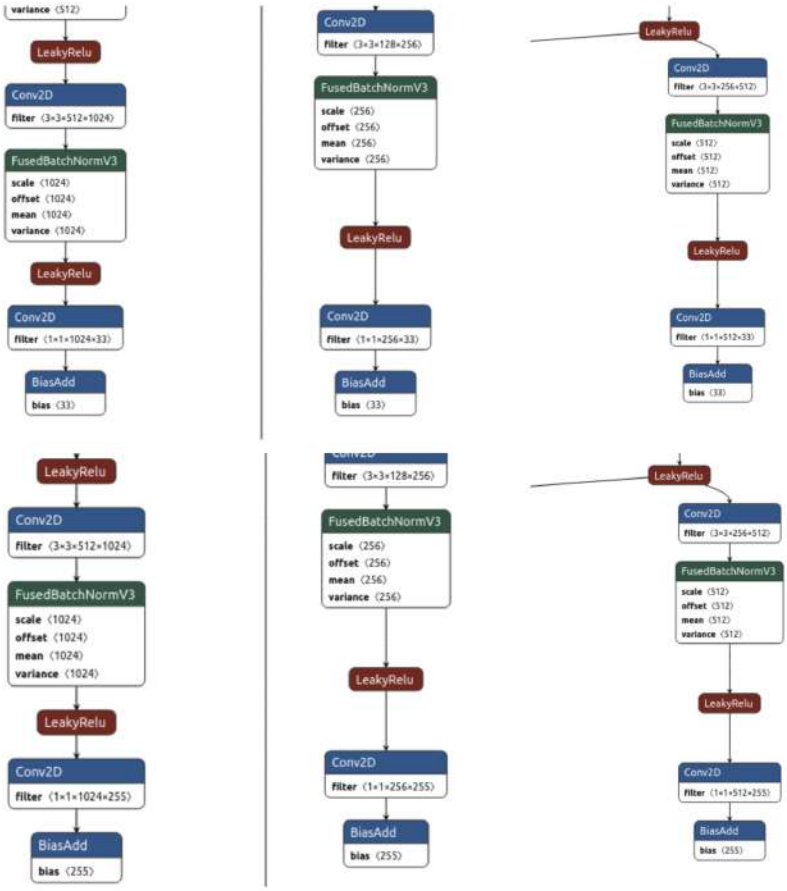


FIGURA 3.8: En la parte de arriba estan las capas de la red empleada en el TFG y las de abajo son las capas originales de la red YOLOV4 para la detección de objetos del dataset COCO.

Durante esta etapa, se ajustaron los parámetros y pesos del modelo para que se especializara en la detección precisa de cubos en las imágenes. Si bien los componentes iniciales del modelo se mantuvieron intactos, las capas finales se adaptaron para

aprender las características relevantes de los cubos, en la Fig. 3.8 se muestran los cambios en las ultimas capas.

3.6. Optimización del modelo

En este trabajo se tomó la decisión de utilizar solo la técnica de quantización para optimizar el modelo, puesto que en este caso se quería comprobar que perdidas se tiene a la hora de ejecutar un modelo en una Kv260, la cual cuenta con DPUs que están creadas para procesar cálculos de las redes neuronales, pero necesitan que todos estos estén en una precisión de 8 bits. Por lo que se decidió utilizar la quantización para convertir los modelos en punto flotante a enteros, además específicamente se utiliza la quantización después del entrenamiento, ya que como se comentó, se utiliza el tranfering learning a la hora de entrenar en modelo de deteción de cubos, debido a las limitaciones que se cuentan en el hardware.

Para esta tarea se utilizó las herramientas aportadas por xilinx en el docker de Vitis-AI. El método de quantización utilizado por Xilinx en Vitis-AI es conocido como ”post-training quantization” (quantización posterior al entrenamiento). En esta quantización posterior al entrenamiento se puede realizar en diferentes niveles, como quantización de peso, quantización de activación o quantización de ambos.

3.6.1. Pasos de la quantización posterior al entrenamiento

Calibración

Después de entrenar el modelo (normalmente en 32 bits), se necesita un conjunto de datos de calibración para determinar el rango de valores que toman los parámetros del modelo. Este conjunto de datos debe ser representativo de los datos de entrada que el modelo encontrará durante la inferencia. Durante la calibración, se ejecuta el modelo en los datos de calibración y se registran los rangos de los valores de los parámetros.

Quantización de los parámetros

Una vez que se ha realizado la calibración, se aplica la quantización real a los parámetros del modelo. El rango de valores determinado durante la calibración se utiliza para escalar y cuantizar los parámetros de manera adecuada.

Quantización de activaciones

Además de cuantizar los parámetros, también es común cuantizar las activaciones de las capas intermedias durante la inferencia. Esto implica quantizar los valores que se propagan a través del modelo después de aplicar las operaciones de cada capa, al igual que con los parámetros.

Como se mencionó anteriormente se utilizó el docker de Xilinx, el cual se descargó de su repositorio oficial de docker en este caso se utilizaron varias versiones la 3.0 CPU tensor-flow para quantizar el modelo y la 2.5 para compilarlo en la placa.

La quantización se llevó a cabo utilizando herramienta integrada en el docker de Xilinx con el siguiente comando.

```
vai_q_tensorflow quantize \
  --input_frozen_graph ../my_model/xilinxs/yolov4-leaky.pb \
  --input_fn yolov4_graph_input_keras_fn.calib_input \
  --output_dir ../yolov4_xilinx_quantized \
  --output_nodes conv2d_93/BiasAdd,conv2d_101/BiasAdd,conv2d_109/BiasAdd \
  --input_nodes image_input \
  --input_shapes ?,512,512,3 \
  --gpu 0 \
  --calib_iter 100
```

Indicando los datos de calibración (yolov_graph_input_keras_fn.calib_input) que se usó el subconjunto (subset) de test que se proporciona con COCO, el modelo que queremos quantizar (../my_model/xilinxs/yolov4-leaky.pb) , las entradas (nombre image-input , dimensión ?,512,512,3) y los nodos de salidas de nuestro modelo (conv2d_93/BiasAdd, conv2d_101/BiasAdd,conv2d_109/BiasAdd). Para saber que capas extraer se visualizó el modelo con la herramienta netron [88]

3.7. Despliegue en la placa

3.7.1. Conexión

Para realizar la comunicación y el control remoto de una placa desde un dispositivo Linux en un entorno de trabajo, en principio se creó una red local LAN, a través de un cable Ethernet, en la que se designó una ip fija a la placa y al ordenador para poder enrutarlos y se utilizó el protocolo SSH (Secure Shell).

SSH es un protocolo de red que permite acceder y administrar de forma segura dispositivos remotos a través de una conexión encriptada. Proporciona una forma segura de establecer sesiones interactivas con otros dispositivos en una red.

Luego se utilizó un router Xiaomi Mi AIoT Router AX3600[89] para que se encargase del control de la red LAN y permitiese a los dos dispositivos conectarse entre ellos a la vez que conectarse a internet.

3.7.2. Modelo

Para permitir la ejecución del modelo en la placa, es necesario compilarlo teniendo en cuenta la arquitectura específica requerida. En este caso, se almacenó la huella digital (fingerprint) de la arquitectura en un archivo JSON. La huella digital utilizada fue "0x101000056010407".

El proceso de compilación del modelo se llevó a cabo utilizando un comando específico que utiliza las herramientas integradas en el docker de xilinxs, el cual se detalla a continuación:

```
TARGET=KV260
model_name=quantize_eval_model.pb
NET_NAME=yolov4-coco-xilinx_kv260_xilinx # yolov4-mod-xilinx_kv260
ARCH=./arch.json
PATH_OUT=./model_data/coco_xilinx/

vai_c_tensorflow --frozen_pb ../my_model/coco_xilinxs/${model_name} \
```

```
                --arch ${ARCH} \
                --output_dir ${PATH_OUT} \
                --net_name ${NET_NAME} \
                --options "{'mode':'normal','save_kernel':'',
                'input_shape':'1,512,512,3'}"

xir png ${PATH_OUT}/${NET_NAME}.xmodel ${PATH_OUT}/${NET_NAME}.png
```

En este comando, se especifica el grafo congelado del modelo quantizado, la ubicación del archivo JSON que contiene la huella digital de la arquitectura, el directorio de salida para el modelo compilado, el nombre de la red (net_name) utilizado en la compilación, y varias opciones adicionales como el modo de compilación, la opción de guardar los kernels y la forma de entrada del modelo.

Una vez completada la compilación del modelo, se obtuvo el modelo compilado en el directorio especificado "./model_data/coco_xilinx/". Este modelo compilado está listo para su implementación y ejecución en la placa específica que corresponda a la arquitectura definida por la huella digital.

3.8. Evaluación

3.8.1. Precisión

Para medir la diferencia entre el modelo en punto flotante, el quantizado y el desplegado se utilizó el subconjunto de COCO 2017, 5000 imágenes de evaluación, y se utilizó la herramienta de COCOeval,es una biblioteca de código abierto ampliamente utilizada para la evaluación de modelos de detección y segmentación de objetos basados en el formato COCO, proporciona métricas de evaluación estándar, como precisión de detección, precisión de localización y puntuación media de precisión, que permiten cuantificar el rendimiento de los modelos. En el trabajo se utilizó una modificación de la misma la cual pertenece a [90].

Métricas

Las siguientes 12 métricas se utilizan para caracterizar el rendimiento de un detector de objetos en COCO:

Precisión promedio (AP): Mide la capacidad del detector de objetos para localizar y clasificar correctamente los objetos en la imagen a distintas escalas.

$$\text{Precisión} = \frac{\text{Número de objetos correctamente detectados}}{\text{Número total de objetos detectados}} \tag{3.3}$$

- % AP IoU=0.50:0.05:0.95 (primary challenge metric)
- % $AP^{IoU=0,50}$ (PASCAL VOC metric)
- % $AP^{IoU=0,75}$ (strict metric)

Precisión promedio en diferentes escalas (AP Across Scales): Mide la capacidad del detector de objetos para localizar y clasificar correctamente los objetos en la imagen a distintas escalas.

- % AP para objetos pequeños: área <322 (APsmall).
- % AP para objetos medianos: 322 <área <962 (APmedium).
- % AP para objetos grandes: área >962 (APlarge).

Recall promedio (AR): Mide la capacidad del detector para encontrar todos los objetos relevantes en una imagen.

$$\text{Recall} = \frac{\text{Número de objetos correctamente detectados}}{\text{Número total de objetos presentes}} \tag{3.4}$$

- % AR con 1 detección por imagen (ARmax=1). Es el recall máximo cuando se permite solo una detección por imagen. Mide la capacidad del detector de objetos para recuperar todos los objetos relevantes en la imagen.
- % AR con 10 detecciones por imagen (ARmax=10). Es el recall máximo cuando se permiten hasta diez detecciones por imagen.
- % AR con 100 detecciones por imagen (ARmax=100). Es el recall máximo cuando se permiten hasta cien detecciones por imagen.

Recall promedio en diferentes escalas (AR Across Scales):

- % AR para objetos pequeños: área <322 (ARsmall).
- % AR para objetos medianos: 322 <área <962 (ARmedium).
- % AR para objetos grandes: área >962 (ARlarge).

Estas métricas se obtienen mediante el cálculo de la precisión y el recall para cada categoría de objetos en el conjunto de datos COCO.

3.8.2. Consumo

GPU

En este trabajo se realizó una evaluación del rendimiento del sistema, incluyendo la medición del consumo de la GPU, la RAM y el porcentaje de uso de ambas. Para lograr esto se utilizó la herramienta **nvidia-smi** [91] proporcionada por NVIDIA, la cual permite obtener información detallada sobre el estado del sistema.

El comando nvidia-smi fue empleado para obtener datos precisos sobre el consumo de la GPU y la RAM. Esto incluye información sobre la potencia utilizada, la cantidad de memoria y la utilización porcentual. La ejecución de este comando proporcionó una visión general del rendimiento del sistema, permitiendo realizar análisis comparativos y observar cambios en el consumo en tiempo real.

```
watch -n0.1 nvidia-smi --query-gpu=memory.used,power.draw,
utilization.gpu  --format=csv > gpu_info.txt
```

FPGA

Para poder estimar el consumo de energía que se produce al ejecutar el modelo en la FPGA, se llevaron a cabo una serie de pruebas considerando diferentes números de hilos en el DPU. Para este propósito se empleó la ley de la potencia eléctrica eq. 3.5.

La ecuación de potencia es:

$$P = I * V \tag{3.5}$$

Donde P representa la potencia, I es la intensidad de corriente y V es el voltaje.

Para medir la intensidad de corriente, se utilizó el multímetro Pro'skit MT-1707[92], las especificaciones del dispositivo se pueden ver en la Tab. A.2, el cual se conectó en serie en el circuito, permitiéndonos obtener una lectura precisa de la corriente de entrada. Así mismo, el voltaje de entrada se midió utilizando el multímetro, el cual se conectó en paralelo al circuito. Esto nos permitió medir con precisión el voltaje que consume a el sistema.

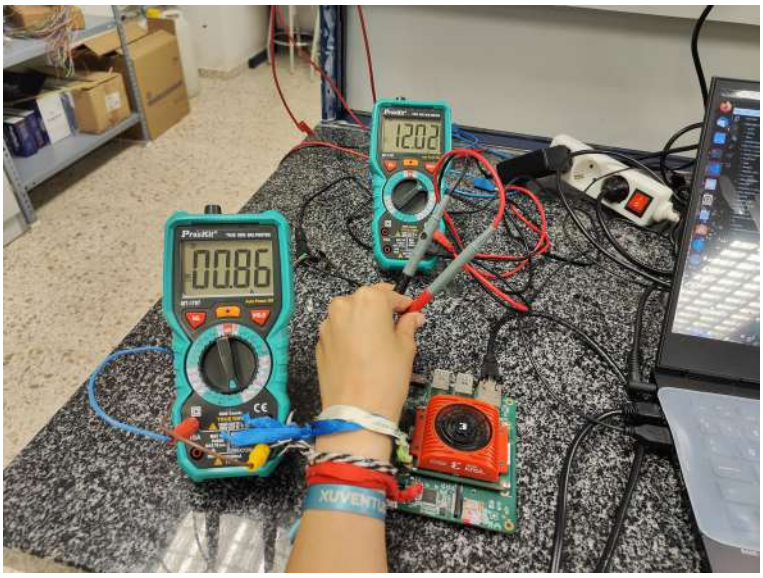


FIGURA 3.9: Consumo con multimetro [86]

Este método también se utiliza para comparar el consumo obtenido a través de los sensores con el que da el multímetro para poder saber cuánto error se produce a la hora de medir estas métricas.

Con sensores

Kria KV260 cuenta con sensores integrados que permiten monitorizar varias variables (temperatura, consumo SoC) y que se pueden acceder a través de la interfaz hwmon [93] , por lo que podemos utilizar 'lm_sesnors'[94] (es una biblioteca y conjunto de utilidades de software de código abierto, que se utiliza para monitorear los sensores de hardware en sistemas Linux para visualizar la información de estos sensores de manera sencilla). Otra manera es utilizar el xmutil, que es otra herramienta de monitoreo que está integrada en el sistema operativo aportado por Xilinxs.

Por lo que se empleó la librería 'lm_sesnors' para generar un script que lance un hilo durante la ejecución de la inferencia del modelo y almacene los datos en un documento para obtener el consumo de la placa.

4 Resultados

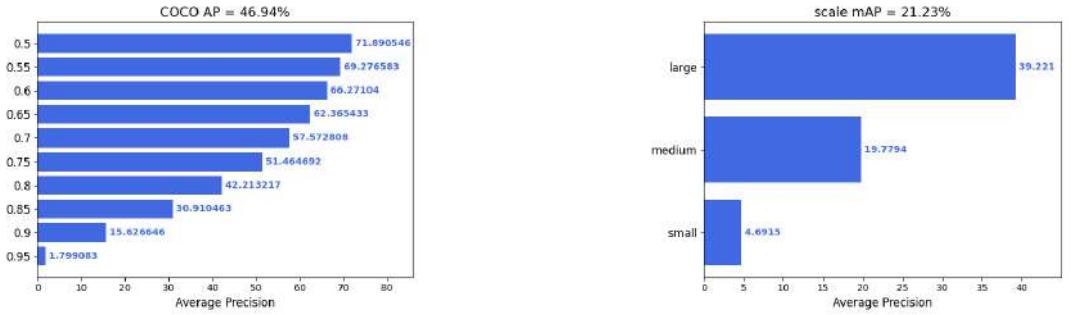
4.0.1. Precisión

Para medir el desempeño del modelo se utilizó el subconjunto de COCO eval 2017. Se llevaron a cabo dos evaluaciones principales en GPU: Una sobre una resolución de los pesos en punto flotante tras su entrenamiento y otra sobre el quantizado con PTQ. La evaluación permitió analizar la influencia de la quantización en el desempeño del modelo. Y por último se volvió a ejecutar dicha prueba sobre el compilado en la placa para comprobar la diferencia de precisión con los anteriores.

4.0.2. GPU

Modelo Flotante

Como se puede observar en la Fig. 4.1, los resultados obtenidos demuestran que la quantización realizada al modelo no implica una perdida significativa en la precisión.



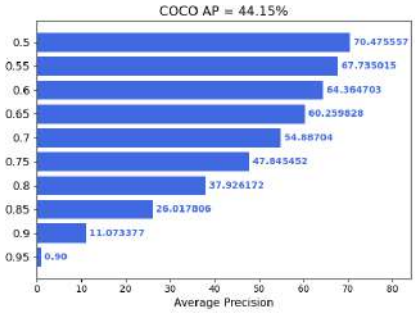
(A) MS COCO AP para distintos porcentajes de IoU

(B) MS COCO AP para distintos tamaños de objetos

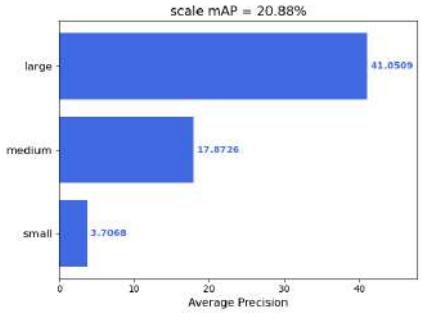
FIGURA 4.1: Resultados obtenidos de la precisión de la GPU en el modelo float [34].

Modelo quantizado

Como se pude observar en la Fig. 4.2 los resultados obtenidos para el modelo en punto flotante son muy similares a los del estado del arte. Pero tiene un peor comportamiento para objetos pequeños y mejora según se aumenta el tamaño del objeto. Lo cual implica que puede ser un buen modelo para la tarea que queremos realizar, que es la detección de cubos en una red de producción puesto que la cámara estará integrada cerca del objeto.



(A) MS COCO AP para distintos porcentajes de IoU

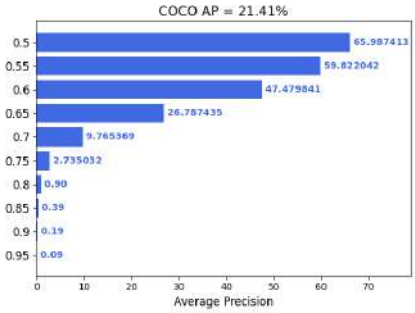


(B) MS COCO AP para distintos tamaños de objetos

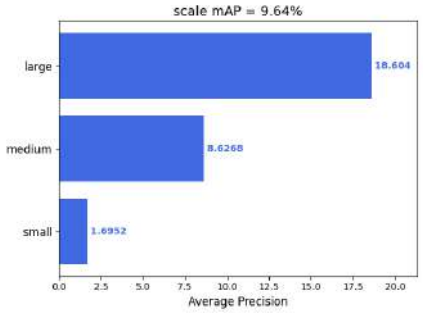
FIGURA 4.2: Resultados obtenidos de la precisión de la gpu en el modelo quantizado [34].

4.0.3. FPGA

En la Fig. 4.2 se visualizan las pérdidas de precisión a la hora de ejecutar el modelo en la DPU. Las cuales son bastantes significativas en comparación a la precisión que aporta el modelo en punto flotante, en concreto a la hora de encuadrar correctamente los objetos.



(A) MS COCO AP para distintos porcentajes de IoU



(B) MS COCO AP para distintos tamaños de objetos

FIGURA 4.3: Resultados obtenidos de la precisión de la DPU en el modelo quantizado [34].

En la Fig. A.1 se muestra la comparativa de los modelos sobre un conjunto de imágenes

4.0.4. Consumo

La medición de consumo se realizó mientras ejecutando la prueba de evaluación anterior

4.0.5. GPU

Como se muestra en la gráfica de la Fig. 4.4 el consumo de la GPU ronda sobre los 75 W

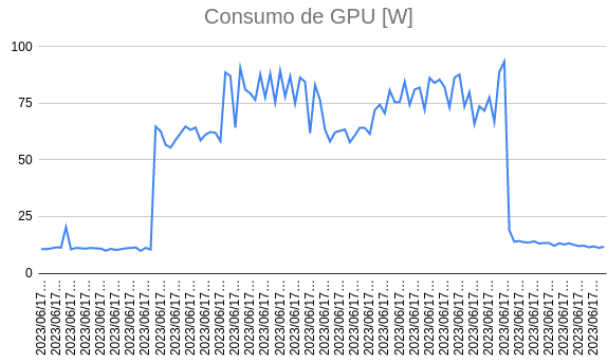


FIGURA 4.4: Consumo en watios de la GPU

4.0.6. FPGA

Como se muestra en la gráfica de la Fig. 4.4 el consumo de la placa KV260 ronda sobre los 15 W, en la gráfica también se observa una discrepancia entre la medición con los sensores y con los tomados con el multímetro, esta es debido a que los sensores solo toma en cuenta el consumo del SOM (Sistem on Modul) y no de la placa entera.

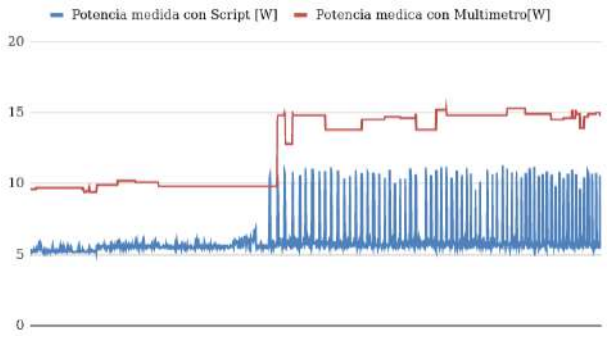


FIGURA 4.5: Consumo en watios de la KV260

4.0.7. Comparativa

TABLA 4.1: Comparación de rendimiento y características

	Precisión (mAP) %	FPS	Time (s)	Consumo medio [W]
GPU	46.94	17	308	75
CPU quantizado	44.15	5	896	70
KV260	21.41	-	-	15

4.0.8. Inferencia del modelo de deteción

4.0.9. Obtención del modelo

En la Fig. 4.6 se muestra la trama de la progresión de las pérdidas (loss) en las 2000 iteraciones de entrenamiento.

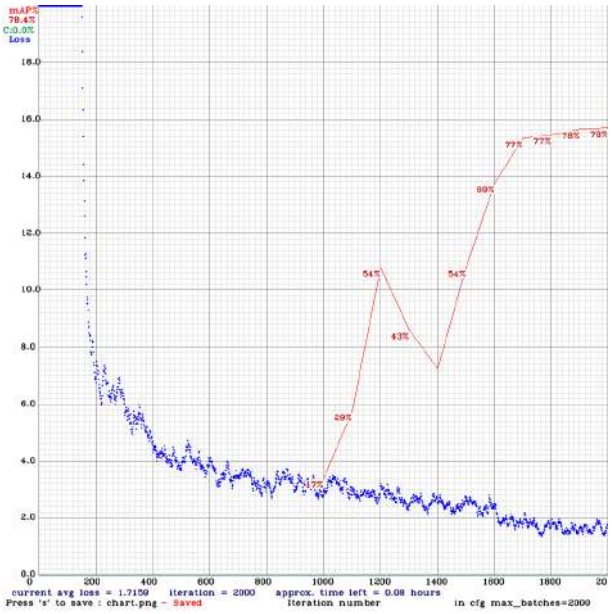


FIGURA 4.6: Las pérdidas (loss) en las 2000 iteraciones de entrenamiento.

En la Fig.A.3 se muestra la arquitectura del modelo que se ejecuta en la DPU como se puede observar en la figura el modelo se compiló para ser ejecutado en una única DPU y con 3 salidas. Y en la Fig. 4.7 se muestra un ejemplo de la detección del modelo compilado y desplegado en la placa, en la Fig. A.2 se muestra una comparativa para la detección de cubos.

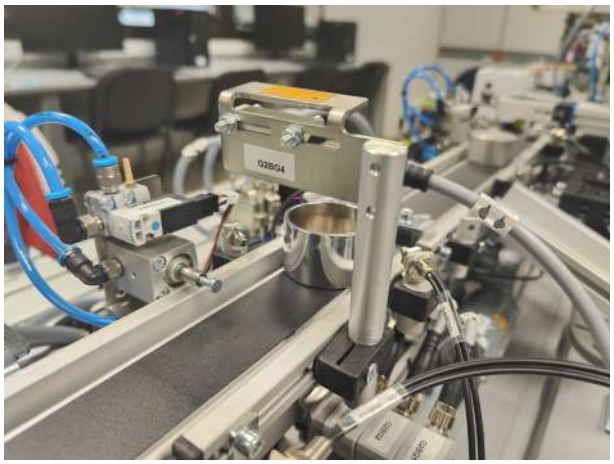


FIGURA 4.7: Ejemplo de deteción en la FPGA

5 Conclusiones

En conclusión, se llevó a cabo una evaluación para determinar si las FPGAs son dispositivos válidos como dispositivos de borde (edge) en aplicaciones de procesamiento de datos. Cumpliendo el objetivo principal que era investigar si el uso de FPGAs podría ofrecer una combinación óptima de consumo de energía y precisión en comparación con otras plataformas de procesamiento.

Los resultados obtenidos revelaron que, si bien las FPGAs ofrecen una ventaja significativa en términos de reducción del consumo de energía en comparación con otros dispositivos, también se observa una pérdida de precisión en el procesamiento de datos. Esta pérdida de precisión se atribuye principalmente al proceso de compilación del modelo para trabajar en la unidad de procesamiento digital (DPU) de la FPGA.

Se observó qué para ejecutar el modelo en la DPU, era necesario agregar capas de corrección (fix) al modelo original. Estas capas de corrección permitieron que el modelo se adaptara a la arquitectura específica de la DPU, pero al mismo tiempo resultaron en una disminución de la precisión de los resultados sobre la DPU. También cabe destacar que el modelo continúa detectando la mayoría de los objetos en comparación con el modelo en punto flotante, En la mayoría de los casos la caja de detección se redujo el tamaño o se desplazó lo que también explica la reducción en los test sintéticos.

Así mismo cabe destacar que el tiempo de ejecución del modelo era significativamente mayor del esperado, alejándose de la detección en tiempo real. Pero cabe destacar que principalmente esto se debe a la necesidad de un post procesado de las salidas de la DPU que se ejecutan en la CPU utilizando el lenguaje python para su implementación lo que justifica este aumento de tiempo.

Estos hallazgos indican que, si bien las FPGAs pueden ser una opción viable en términos de eficiencia energética, es importante considerar el impacto en la precisión de los resultados. En aplicaciones donde la precisión es de suma importancia, es posible que las FPGAs no sean la mejor opción, ya que la pérdida de precisión puede afectar negativamente el rendimiento general del sistema.

No obstante, en escenarios donde el consumo de energía es un factor crítico y la precisión puede ser tolerada hasta cierto punto, las FPGAs siguen siendo una opción atractiva. Es importante evaluar cuidadosamente los requisitos específicos de la aplicación y considerar los compromisos entre consumo de energía y precisión antes de decidir utilizar FPGAs como dispositivos de borde.

Para el caso expuesto en el TFG, el modelo que se desplegó en la placa obtuvo una precisión aceptable para la tarea requerida, aunque la velocidad de ejecución no era la esperada. Por lo que se considera que habría que utilizar algún tipo de aceleración

de cálculos del post procesado para poder considerar las FPGAs como dispositivos edge.

A Anexo

TABLA A.1: Modelos Pre-Entrenados de Intel en el OpenVINO™ Toolkit [35].

Model Name	Implementation	Accuracy	GFlops	mParams
CTPN	TensorFlow*	73.67 %	55.813	17.237
CenterNet (CTDET with DLAV0) 512x512	ONNX*	44.2756 %	62.211	17.911
DETR-ResNet50	PyTorch*	39.27 % / 42.36 %	174.4708	41.3293
EfficientDet-D0	TensorFlow*	31.95 %	2.54	3.9
EfficientDet-D1	TensorFlow*	37.54 %	6.1	6.6
FaceBoxes	PyTorch*	83.565 %	1.8975	1.0059
Face Detection Retail	Caffe*	83.00 %	1.067	0.588
Faster R-CNN with Inception-ResNet v2	TensorFlow*	40.69 %	30.687	13.307
Faster R-CNN with ResNet 50	TensorFlow*	31.09 %	57.203	29.162
MobileFace Detection V1	MXNet*	78.7488 %	3.5456	7.6828
Mobilenet-yolo-v4-syg	Keras*	86.35 %	65.981	61.922
MTCNN	Caffe*	48.1308 % / 62.2625 %	3.3715 0.0031 0.0263	0.0066
NanoDet with ShuffleNetV2 1.5x, size=416	PyTorch*	27.38 %/26.63 %	2.3895	2.0534
NanoDet Plus with ShuffleNetV2 1.5x, size=416	PyTorch*	34.53 %/33.77 %	3.0147	2.4614
Pelee	Caffe*	21.9761 %	1.290	5.98
RetinaFace with ResNet 50	PyTorch*	91.78 %	88.8627	27.2646
RetinaNet with Resnet 50	TensorFlow*	33.15 %	238.9469	64.9706
R-FCN with Resnet-101	TensorFlow*	28.40 % / 45.02 %	53.462	171.85
SSD 300	Caffe*	ssd300	87.09 %	26.285
SSD 512	Caffe*	ssd512	91.07 %	27.189
SSD with MobileNet	Caffe* TensorFlow*	67.00 % 23.32 %	2.31	36.807
SSD with MobileNet FPN	TensorFlow*	35.5453 %	123.309	36.188
SSD lite with MobileNet V2	TensorFlow*	24.2946 %	1.525	4.475
SSD with ResNet 34 1200x1200	PyTorch*	20.7198 % / 39.2752 %	433.411	20.058
Ultra Lightweight Face Detection RFB 320	PyTorch*	84.78 %	0.2106	0.3004
Ultra Lightweight Face Detection slim 320	PyTorch*	83.32 %	0.1724	0.2844
Vehicle License Plate Detection Barrier	TensorFlow*	99.52 %	0.271	0.547
YOLO v1 Tiny	TensorFlow.js*	54.79 %	6.9883	15.8587
YOLO v2 Tiny	Keras*	27.3443 % / 29.1184 %	5.4236	11.2295
YOLO v2	Keras*	53.1453 % / 56.483 %	63.0301	50.9526
YOLO v3	Keras* ONNX*	62.2759 % / 67.7221 % 48.30 % / 47.07 %	65.9843	61.9221
YOLO v3 Tiny	Keras* ONNX*	35.9 % / 39.7 % 17.07 % / 13.64 %	5.582	8.848
YOLO v4	Keras*	71.23 %/77.40 % / 50.26 %	129.5567	64.33
YOLO v4 Tiny	Keras*	6.9289	6.0535	-
YOLOF	PyTorch*	60.69 % / 66.23 % / 43.63 %	175.38	48.228
YOLOX Tiny	PyTorch*	47.85 % / 52.56 % / 31.82 %	6.4813	5.0472

TABLA A.2: Especificaciones de MT-1707

Medición	**Precisión**
DCV	600mV/6V/60V/600V/1000V ±(0.5 %+3d)
ACV TrueRMS	6V/60V ±(0.8 %+3d) 600V/750V ±(1 %+10d)
Frequency Response	40Hz-1KHz
DCA	600uA/60mA/600mA ±(0.8 %+3d) 10.00A ±(1.5 %+10d)
ACA True RMS	60mA/600mA ±(1 %+3d) 10.00A ±(2 %+10d)
Frequency Response	40Hz-1KHz
Frequency (Hz)	9.999Hz-9.999MHz ±(1 %Reading+3 digits)

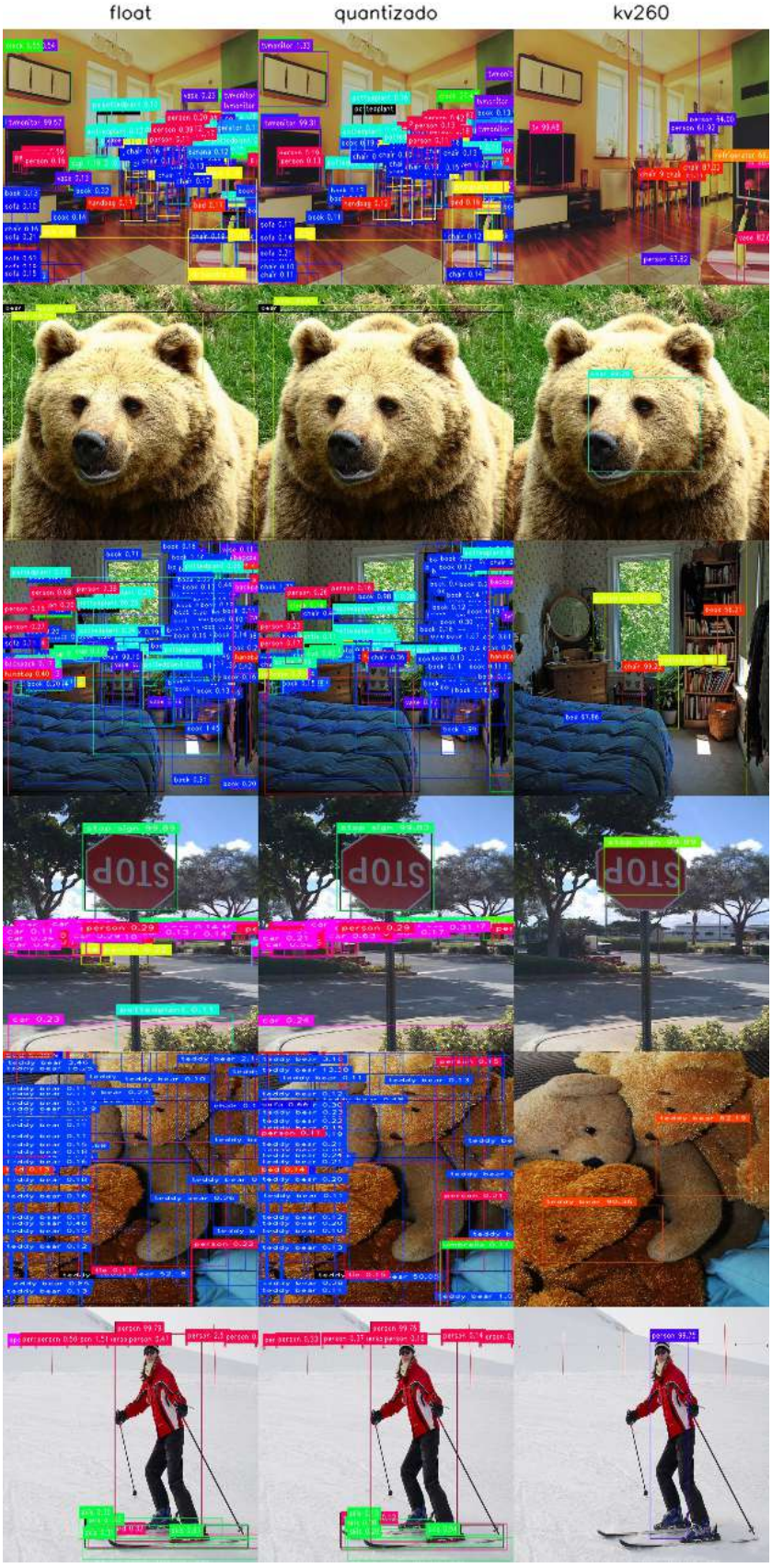


FIGURA A.1: Comparativa de los modelos en el dataset COCO 2017

FIGURA A.2: Comparativa de los modelos en la detección de cubos

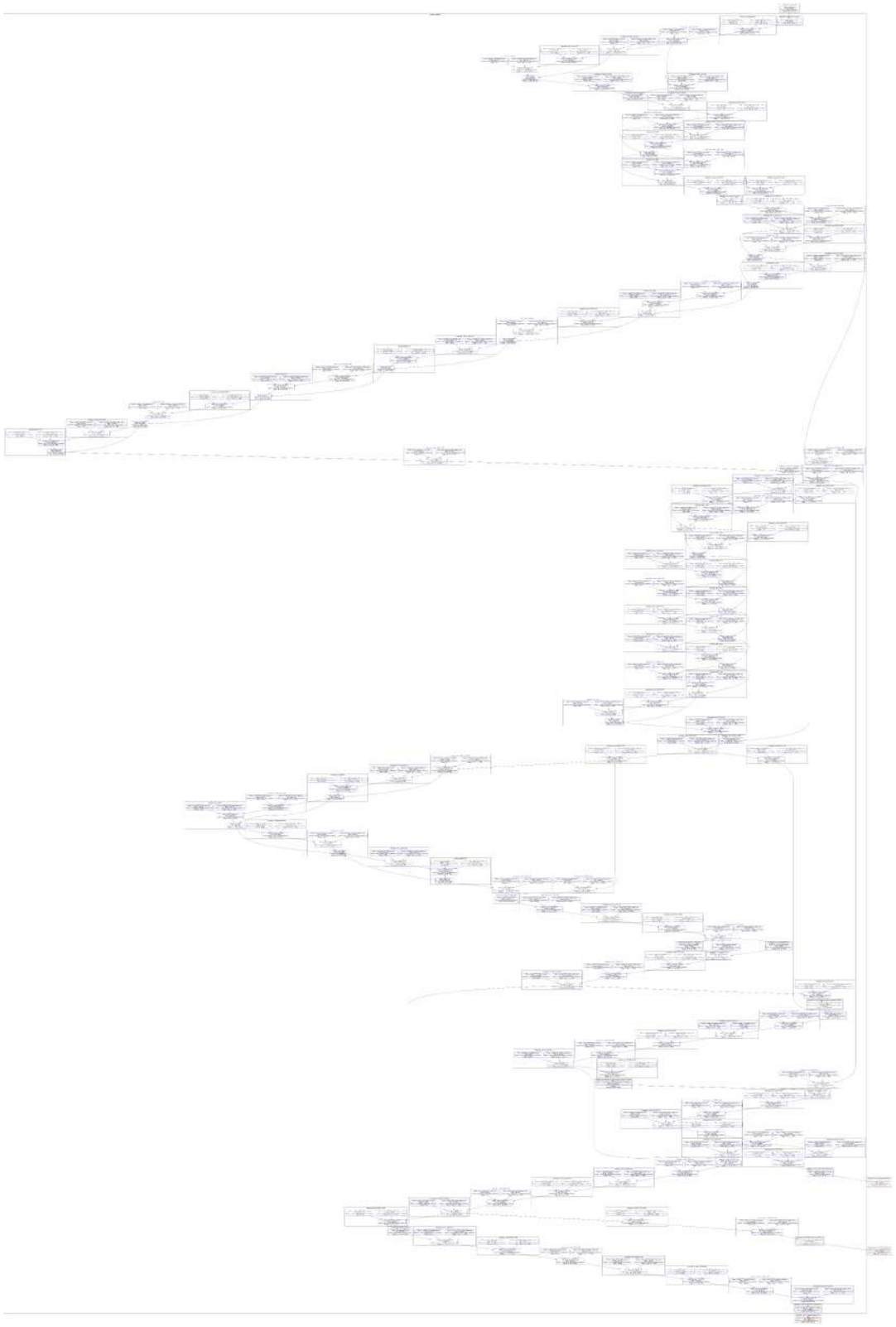


FIGURA A.3: Arquitectura YoloV4 Compilada para la DPU

Bibliografía

[1] Q. Xia, W. Ye, Z. Tao, J. Wu y Q. Li, «A survey of federated learning for edge computing: Research problems and solutions,» *High-Confidence Computing*, vol. 1, pág. 100 008, 2021. DOI: 10.1016/j.hcc.2021.100008.

[2] G. Abreha, M. Hayajneh y M. A. Serhani, «Federated Learning in Edge Computing: A Systematic Survey,» *Sensors*, vol. 22, pág. 450, 2022. DOI: 10.3390/s22020450.

[3] Q. Chen, Z. Zheng, C. Hu, D. Wang y F. Liu, «On-edge multi-task transfer learning: Model and practice with data-driven task allocation,» *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, págs. 1357-1371, 2020. DOI: 10.1109/TPDS.2019.2962435.

[4] A. Alkhulaifi, F. Alsahli e I. Ahmad, «Knowledge distillation in deep learning and its applications,» *PeerJ Computer Science*, vol. 7, e474, 2021. DOI: 10.7717/peerj-cs.474.

[5] A. Alkhulaifi, F. Alsahli e I. Ahmad, «Knowledge distillation in deep learning and its applications,» *PeerJ Computer Science*, vol. 7, n.º 474, 2021. DOI: 10.7717/peerj-cs.474.

[6] W. Liu, D. Anguelov, D. Erhan et al., «SSD: Single Shot MultiBox Detector,» 2016. DOI: 10.1007/978-3-319-46448-0_2.

[7] T. Yuan, L. Lv, F. Zhang et al., «Robust Cherry Tomatoes Detection Algorithm in Greenhouse Scene Based on SSD,» *Agriculture*, vol. 10, n.º 5, pág. 160, 2020. DOI: 10.3390/agriculture10050160.

[8] K. He, X. Zhang, S. Ren y J. Sun, «Deep Residual Learning for Image Recognition,» inf. téc., 2015. DOI: 10.48550/arXiv.1512.03385.

[9] A. G. Howard, M. Zhu, B. Chen et al., «MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,» 2017. DOI: 10.48550/arXiv.1704.04861.

[10] A. G. Howard, M. Zhu, B. Chen et al., «Searching for MobileNetV3,» 2019. DOI: 10.48550/arXiv.1905.02244.

[11] N. Singh Sanjay y A. Ahmadinia, «MobileNet-Tiny: A Deep Neural Network-Based Real-Time Object Detection for Raspberry Pi,» en *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, 2019. DOI: 10.1109/ICMLA.2019.00118.

[12] B. Wu, A. Wan, F. Iandola, P. H. Jin y K. Keutzer, «SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving,» 2019. DOI: 10.48550/arXiv.1612.01051.

[13] M. Tan, R. Pang y Q. V. Le, «EfficientDet: Scalable and Efficient Object Detection,» 2020. DOI: 10.48550/arXiv.1911.09070.

[14] M. Tan y Q. V. Le, «Efficientnet: Rethinking model scaling for convolutional neural networks,» *International Conference on Machine Learning*, 2019. DOI: 10.48550/arXiv.1905.11946.

[15] T.-Y. Lin, P. Goyal, R. Girshick, K. He y P. Dollár, «Focal Loss for Dense Object Detection,» 2018. DOI: 10.48550/arXiv.1708.02002.

[16] X. Yang y J. Yan, «On the Arbitrary-Oriented Object Detection: Classification based Approaches Revisited,» *International Journal of Computer Vision*, 2022. DOI: 10.48550/arXiv.2003.05597.

[17] M. Tan, R. Pang y Q. V. Le, «Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,» 2016. DOI: 10.1109/CVPR.2017.123.

[18] Z. Deng, H. Sun, S. Zhou, J. Zhao, L. Lei y H. Zou, «Multi-scale object detection in remote sensing imagery with convolutional neural networks,» *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 145, 2018. DOI: 10.1016/j.isprsjprs.2018.04.003.

[19] J. Redmon, S. Divvala, R. Girshick y A. Farhadi, «You Only Look Once: Unified, Real-Time Object Detection,» 2016. DOI: 10.1109/CVPR.2016.91.

[20] H.-Y. M. L. Alexey Bochkovskiy Chien-Yao Wang, «YOLOv4: Optimal Speed and Accuracy of Object Detection,» 2020.

[21] C.-Y. Wang, H.-Y. M. Liao, Y.-H. Wu, P.-Y. Chen, J.-W. Hsieh e I.-H. Yeh, «CSPNet: A new backbone that can enhance learning capability of CNN,» en *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshop (CVPR Workshop)*, 2020. DOI: 10.48550/arXiv.1911.11929.

[22] K. He, X. Zhang, S. Ren y J. Sun, «Spatial pyramid pooling in deep convolutional networks for visual recognition,» *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2015.

[23] G. G. Ljudevit Jelečević Marko Horvat, «A comparative study of YOLOv5 models performance for image localization and classification,» 2021.

[24] G. Jocher, *YOLOv5 release v6.1*, https://github.com/ultralytics/yolov5/releases/tag/v6.1, 2022.

[25] C. Li, L. Li, Y. Geng et al., «YOLOv6 v3.0: A Full-Scale Reloading,» 2023. DOI: 10.48550/arXiv.2301.05586.

[26] C. Li, L. Li, H. Jiang et al., «YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications,» 2021. DOI: 10.48550/arXiv.2209.02976.

[27] I. Khokhlov, E. Davydenko, I. Osokin et al., «Tiny-YOLO object detection supplemented with geometrical data,» 2020. DOI: 10.48550/arXiv.2008.02170.

[28] X. Xu, Y. Jiang, W. Chen, Y. Huang, Y. Zhang y X. Sun, «DAMO-YOLO: A Report on Real-Time Object Detection Design,» 2022. DOI: 10.48550/arXiv.2211.15444.

[29] Z. Sun, M. Lin, X. Sun, Z. Tan, H. Li y R. Jin, «Mae-det: Revisiting maximum entropy principle in zero-shot nas for efficient object detection,» en *International Conference on Machine Learning*, PMLR, 2022, págs. 20 810-20 826. DOI: 10.48550/arXiv.2111.13336.

[30] yiqi jiang, Z. Tan, J. Wang, X. Sun, M. Lin y H. Li, «GiraffeDet: A Heavy-Neck Paradigm for Object Detection,» en *International Conference on Learning Representations*, 2022. DOI: 10.48550/arXiv.2202.04256.

[31] S. Kim y H. Kim, «Zero-Centered Fixed-Point Quantization With Iterative Retraining for Deep Convolutional Neural Network-Based Object Detectors,» 2020.

[32] Z. Ge, S. Liu, F. Wang, Z. Li y J. Sun, «YOLOX: Exceeding YOLO Series in 2021,» 2021. DOI: 10.48550/arXiv.2107.08430.

[33] M. L. Yoshitomo Matsubara, «Neural Compression and Filtering for Edge-assisted Real-time Object Detection in Challenged Networks,» 2021. DOI: 10.48550/arXiv.2007.15818.

[34] T.-Y. Lin, M. Maire, S. Belongie et al., «Microsoft COCO: Common Objects in Context,» 2014. DOI: 10.48550/arXiv.1405.0312.

[35] Intel Corporation, *OpenVINO: Open Visual Inference and Neural Network Optimization*, Accessed: September 2021, 2021. dirección: https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html.

[36] Dell Inc., *XPS 13 9370 Setup and Specifications*, Disponible en: https://i.dell.com/sites/doccontent/corporate/secure/en/Documents/dell-xps-13-brochure.pdf, 2018.

[37] C.-Y. Wang, A. Bochkovskiy y H.-Y. M. Liao, «YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors,» págs. 2, 6, 7, 8, 1, 2022. DOI: 10.48550/arXiv.2207.02696.

[38] S. Liu, D. Huang e Y. Wang, «Learning Spatial Fusion for Single-Shot Object Detection,» *arXiv preprint arXiv:1911.09516*, 2019. DOI: 10.48550/arXiv.1911.09516.

[39] C. Surianarayanan, J. J. Lawrence, P. R. Chelliah, E. Prakash y C. Hewage, «A Survey on Optimization Techniques for Edge Artificial Intelligence (AI),» 2023. DOI: 10.3390/s23031279.

[40] M. Merenda, C. Porcaro y D. Iero, «Edge Machine Learning for AI-Enabled IoT Devices: A Review,» *Sensors*, vol. 20, pág. 2533, 2020. DOI: 10.3390/s20092533.

[41] J. Kukačka, V. Golkov y D. Cremers, «Regularization for Deep Learning: A Taxonomy,» 2017. DOI: 10.48550/arXiv.1710.10686.

[42] M. Denil, B. Shakibi, L. Dinh, M. Ranzato y N. de Freitas, «Predicting Parameters in Deep Learning,» 2013. DOI: 10.48550/arXiv.1306.0543.

[43] W. Chen, J. Wilson, S. Tyree, K. Weinberger e Y. Chen, «Compressing Neural Networks with the Hashing Trick,» 2015. DOI: 10.48550/arXiv.1504.04788.

[44] X. Gou, L. Qing, Y. Wang, M. Xin y X. Wang, «Re-training and parameter sharing with the Hash trick for compressing convolutional neural networks,» 2020. DOI: 10.1016/j.asoc.2020.106783.

[45] J. Van Leeuwen, «On the construction of huffman trees,» 1976.

[46] S. Han, H. Mao y W. Dally, «Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,» 2016. DOI: 10.48550/arXiv.1306.0543.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort et al., «Scikit-learn: Machine Learning in Python,» *Journal of Machine Learning Research*, vol. 12, págs. 2825-2830, 2011.

[48] L. Buitinck, G. Louppe, M. Blondel et al., «API design for machine learning software: experiences from the scikit-learn project,» en *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, págs. 108-122.

[49] T. Head, MechCoder, G. Louppe et al., *scikit-optimize/scikit-optimize: v0.5.2*, ver. v0.5.2, mar. de 2018. DOI: 10.5281/zenodo.1207017. dirección: https://doi.org/10.5281/zenodo.1207017.

[50] F. Nogueira, *Bayesian Optimization: Open source constrained global optimization tool for Python*, 2014–. dirección: https://github.com/fmfn/BayesianOptimization.

[51] J. Snoek, H. Larochelle y R. P. Adams, «Practical Bayesian Optimization of Machine Learning Algorithms,» *arXiv preprint arXiv:1206.2944*, 2012. DOI: 10.48550/arXiv.1206.2944.

[52] T. G. authors, *GPyOpt: A Bayesian Optimization framework in python*, http://github.com/SheffieldML/GPyOpt, 2016.

[53] J. Bergstra, D. Yamins y D. D. Cox, «Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures,» 2013. dirección: https://github.com/hyperopt/hyperopt.

[54] F. Chollet et al. «Keras.» (2015), dirección: https://github.com/fchollet/keras.

[55] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez e I. Stoica, «Tune: A Research Platform for Distributed Model Selection and Training,» 2018. DOI: 10.48550/arXiv.1807.05118.

[56] T. Akiba, S. Sano, T. Yanase, T. Ohta y M. Koyama, «Optuna: A Next-generation Hyperparameter Optimization Framework,» 2019. DOI: 10.48550/arXiv.1907.10902.

[57] Z. Chen, Y. Deng, Y. Wu, Q. Gu e Y. Li, «Towards Understanding Mixture of Experts in Deep Learning,» 2022. DOI: 10.48550/arXiv.2208.02813.

[58] Jasper Snoek, *Spearmint*, Software package, 2021. dirección: https://github.com/JasperSnoek/spearmint.

[59] K. Ullrich, E. Meeds y M. Welling, «Soft Weight-Sharing for Neural Network Compression,» 2017. DOI: 10.48550/arXiv.1702.04008.

[60] C. Qi, S. Shen, R. Li et al., «An efficient pruning scheme of deep neural networks for Internet of Things applications,» *EURASIP J. Adv. Signal Process.*, vol. 2021, pág. 31, 2021. DOI: 10.1186/s13634-021-00744-4.

[61] S. Han, J. Pool, J. Tran y W. Dally, «Advances in Neural Information Processing Systems 28 (NIPS 2015), Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS),» en *Advances in Neural Information Processing Systems 28 (NIPS 2015), Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*, MIT Press, vol. 1, 2015, págs. 1135-1143.

[62] K. Kamma y T. Wada, «REAP: A Method for Pruning Convolutional Neural Networks with Performance Preservation,» *IEICE Trans. Inf. Syst.*, vol. 104, págs. 194-202, 2021. DOI: 10.1587/transinf.2020EDP7049.

[63] J.-H. Luo, J. Wu y W. Lin, «ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression,» en *Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, págs. 5068-5076. DOI: 10.1109/ICCV.2017.541.

[64] Y. He, X. Zhang y J. Sun, «Channel pruning for accelerating very deep neural networks,» en *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, págs. 1389-1397. DOI: 10.48550/arXiv.1707.06168.

[65] S. Gupta, A. Agrawal, K. Gopalakrishnan y P. Narayanan, «Deep Learning with Limited Numerical Precision,» en *Proceedings of the 32nd International Conference on International Conference on Machine Learning (ICML)*, 2015. DOI: 10.48550/arXiv.1502.02551.

[66] L. Hou y J. T.-Y. Kwok, «Loss-aware weight quantization of deep networks,» en *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018. DOI: 10.48550/arXiv.1802.08635.

[67] A. Zhou, A. Yao, K. Wang e Y. Chen, «Explicit Loss-Error-Aware Quantization for Low-Bit Deep Neural Networks,» en *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. DOI: 10.1109/CVPR.2018.00982.

[68] A. Zhou, A. Yao, Y. Guo y L. Xu, «Incremental network quantization: Towards lossless CNNs with low-precision weights,» en *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, Toulon, France, 2017.

[69] X. Sun, N. Wang, C.-Y. Chen et al., «Ultra-low precision 4-bit training of deep neural networks,» en *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[70] J. Lee, D. Kim y B. Ham, «Network Quantization with Element-wise Gradient Scaling,» 2021. DOI: 10.48550/arXiv.2104.00903.

[71] *Nvidia Embedded Systems*, https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/, Accedido el 12 de junio de 2023.

[72] Axis Communications, *AXIS Q1615-LE Mk III Network Camera*, 2022.

[73] NXP Semiconductors, *i.MX 8 Series Applications Processors Multicore Arm® Cortex® Processors*, 2023. dirección: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i.mx-applications-processors:IMX_SERIES.

[74] *Kria K26 SOM: The Ideal Platform for Vision AI at the Edge*, Xilinx, Inc. WP529 (v1.0), 2021. dirección: https://docs.xilinx.com/v/u/en-US/wp529-som-benchmarks.

[75] Xilinx, *Kria KV260 Vision AI Starter Kit Data Sheet (DS986)*, Xilinx Inc., 2021. DOI: 10.1234/DS986.

[76] V. Mazzia, A. Khaliq, F. Salveti y M. Chiaberge, «Real-Time Apple Detection System Using Embedded Systems With Hardware Accelerators: An Edge AI Application,» *IEEE Access*, vol. 8, págs. 3615-3622, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2964608.

[77] P. C. Gómez, «Implementation of a Convolutional Neural Network (CNN) on a FPGA for Sign Language's Alphabet recognition,» 2018, Trabajo de Fin de Grado,Universidad Politécnica de Madrid.

[78] J. Hoong, «IMPLEMENTATION OF CONVOLUTIONAL NEURAL NETWORKS ON FPGA FOR HUMAN ACTION RECOGNITION,» 2020, Trabajo de Fin de Master,California State Polytechnic University.

[79] NVIDIA Corporation, *NVIDIA Jetson Partner Products*, Disponible en: https://developer.nvidia.com/embedded/jetson-partner-products?t1_supported-jetson=TX2, Accessed: Fecha de acceso, 2023.

[80] Xilinx Inc., *Xilinx Kria KV260 Product Brief*, Accessed on May 30, 2023, 2021.

[81] Xilinx, *Vitis AI Optimizer User Guide (UG1333)*, Año no especificado. dirección: https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/.

[82] Xilinx, *Vitis-AI Repository*, Disponible en: https://github.com/Xilinx/Vitis-AI, Accessed: Fecha de acceso, Año de acceso.

[83] A. Dutta y A. Zisserman, *VIA (VGG Image Annotator)*, Accessed: June 18, 2023, 2019. dirección: http://www.robots.ox.ac.uk/~vgg/software/via/.

[84] S. Liu, L. Qi, H. Qin, J. Shi y J. Jia, «Path aggregation network for instance segmentation,» en *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, págs. 8759-8768.

[85] Joseph, Redmon and Ali ,Farhadi, *YOLOv3: An Incremental Improvements*, 2018. DOI: 10.48550/arXiv.1804.02767.

[86] W. Joo, J.-H. Baek, S.-H. Jo, S. Y. Kim y J.-H. Jeong, «A Study on Object Detection Performance of YOLOv4 for Autonomous Driving of Tram,» *Sensors*, vol. 21, 2021.

[87] A. no disponibles, «Deep-Learning-Based Automatic Monitoring of Pigs' Physico-Temporal Activities at Different Greenhouse Gas Concentrations,» *Animals*, 2021. DOI: 10.3390/ani11113089.

[88] L. Roeder, *Netron*, Accessed: June 18, 2023, 2016. dirección: https://github.com/lutzroeder/Netron.

[89] *Mi AIoT Router AX3600*, https://www.mi.com/global/product/mi-aiot-router-ax3600/specs, Accessed: 17 June 2023.

[90] Adamdad, *Keras-YOLOv3-MobileNet*, https://github.com/Adamdad/keras-YOLOv3-mobilenet, Repositorio de GitHub, fecha de última actualización.

[91] *NVIDIA System Management Interface (nvidia-smi)*, https://developer.nvidia.com/nvidia-system-management-interface, Accedido el 12 de junio de 2023.

[92] I. Eclipse Enterprises, *MT-1707 30 Positions, Full Functions*, Eclipse Enterprises, Inc. Phone: 804-561-2610 Fax: 804-561-2642 Web Site: https://eclipsetools.com/default/mt-1707.html, 2018.

[93] Kernel.org, *Documentation/hwmon/sysfs-interface*, https://www.kernel.org/doc/Documentation/hwmon/sysfs-interface, [Fecha de acceso: DD de mes de AAAA].

[94] Jean Delvare, *lm_sensors*, lm_sensors Development Team, 2021. dirección: https://github.com/groeck/lm-sensors.