Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

Trabajo de Fin de Máster

Análisis de rendimiento de modelos basados en aprendizaje por transferencia con TensorFlow y TensorRT

Performance analysis of learning transfer based models with TensorFlow and TensorRT

Nicolás Hernández González

La Laguna, 9 de marzo de 2023

D. **Francisco Almeida Rodríguez**, con N.I.F. 42831571-M profesor Catedrático de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Vicente José Blanco Pérez**, con N.I.F. 42.171.808-C profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"*Análisis de rendimiento de modelos basados en aprendizaje por transferencia con TensorFlow y TensorRT*"

ha sido realizada bajo su dirección por D. **Nicolás Hernández González**, con N.I.F. 54.108.335-F.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 9 de marzo de 2023

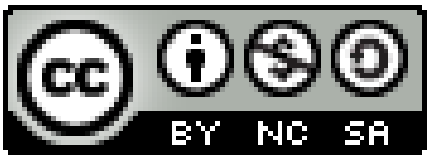
Agradecimientos

Le quiero agradecer a mi familia que asintieran con una sonrisa cuando les contaba las dificultades que me encontraba durante el desarrollo del proyecto. Y también a mis tutores, sobretodo, por tenerme paciencia.

Financiación

Este trabajo ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación de España con los proyectos PID2019-107228RB-I00, TED2021-131019B-I00 y PDC2022-134013-I00; y por el Gobierno de Canarias con el proyecto ProID2021010012.

Licencia



Resumen

En los últimos años hemos podido observar un vertiginoso auge de la inteligencia artificial, tal es así, que nos resultaría difícil encontrar algún campo en el que no se haya aplicado de una u otra forma. Este crecimiento ha estado encabezado de forma general por el Machine Learning, y de forma particular por el Deep Learning, subcampo del citado Machine Learning, que se diferencia de los modelos tradicionales por el uso de redes neuronales y un abanico de aplicaciones más amplio. No obstante, las redes neuronales tienen un serio inconveniente que puede provocar que muchos se replanteen su uso, el diseño de una red desde cero es una tarea ardua y costosa.

En este trabajo se pretende mostrar cómo podemos adaptar una red preentrenada a un propósito específico para el cual estaba preparada inicialmente, lo cual se conoce como transferencia de conocimiento. La aplicación seleccionada para la prueba de concepto es el reconocimiento facial, y forma parte del objetivo del proyecto que la aplicación pueda ser ejecutada en dispositivos IoT. Se evaluará el rendimiento y eficiencia de los modelos generados empleando para ello tanto un ordenador de propósito general como un dispositivo NVIDIA Jetson AGX Orin, que está específicamente está diseñado para optimizar procesos de Inteligencia Artificial. Para el desarrollo de los modelos emplearemos la biblioteca TensorFlow, mientras que para la optimización de los procesos de inferencia usaremos la SDK de NVIDIA TensorRT.

Palabras clave: TensorFlow, TensorRT, Deep Learning, NVIDIA Jetson

Abstract

In recent years we have seen a dizzying rise in artificial intelligence, so much so that it would be difficult to find any field in which it has not been applied in one way or another. More specifically, Deep Learning, the subfield of artificial intelligence that uses neural networks, is the one that has been developed the most due to the possibilities it offers compared to more traditional models. However, neural networks have a severe drawback that may cause many to rethink their use. Designing a network from scratch is an arduous and costly task.

In this work, we intend to show how we can adapt a pre-trained network to a specific purpose for which it was initially prepared, known as knowledge transfer, where we have chosen an application that has a place within the IoT, such as facial recognition. The performance and efficiency of the generated models will be evaluated using a general-purpose computer and the NVIDIA Jetson AGX Orin device, designed to optimize Artificial Intelligence processes. We will use the TensorFlow library to develop the models and the NVIDIA TensorRT SDK to optimize the inference processes.

Keywords: TensorFlow, TensorRT, Deep Learning, NVIDIA Jetson

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Motivación

Entre los constantes avances tecnológicos que se llevan produciendo desde finales del siglo pasado, sin duda hay un campo que en los últimos tiempos está en auge y no parece que vaya a dejar de estarlo en los próximos años; nos referimos a la Inteligencia Artificial (IA), un concepto que no es nuevo pero que siempre se ha visto relacionada con supercomputadores o centros con altas capacidades de cómputo. No obstante, el presente nos muestra el cómo es posible que esta tecnología se haga un hueco en nuestro día a día al incorporarse en nuestros aparatos móviles y demás dispositivos del Internet de las Cosas (IoT, por sus siglas en inglés).

Este apogeo de la IA está encabezado por un subcampo dentro de la IA propiamente dicha, el *Deep Learning* o Aprendizaje Profundo. Dentro de la IA podemos encontrar una rama denominada *Machine Learning*(ML) o Aprendizaje Automático, que agrupa las técnicas destinadas a permitir que las máquinas aprendan de forma autónoma, en la que a su vez hay un subcampo en el que se proponen arquitecturas basadas en redes neuronales artificiales, el *Deep Learning*. En cuanto al por qué éste último ha ganado importancia frente a las técnicas clásicas, el *Deep Learning* ha demostrado una mejor capacidad de aprendizaje y un menor error de generalización ante problemas con grandes cantidades de datos, por ejemplo en problemas de Visión por Computador [33].

Una vez dicho esto, debemos hacer mención a la mayor dificultad asociada al *Deep Learning*, el diseño de una red neuronal artificial desde cero es una tarea altamente costosa, en especial si se pretende que lleve a cabo tareas de gran complejidad. Si se tiene la intención de ejecutar un modelo de Inteligencia Artificial en dispositivos IoT, se debe tener en cuenta que el modelo debe ser lo suficientemente complejo como para realizar su función con la suficiente precisión, todo ello sin que suponga una carga de trabajo inasumible para el dispositivo objetivo.

Por este motivo se propone el uso de una red neuronal entrenada para llevar a cabo una tarea tan compleja como es el reconocimiento facial, para ello nos valdremos de la capacidad conocida como transferencia de conocimiento a fin

de emplear una red neuronal pre-entrenada y adaptarla a la tarea que nos ocupa, con el fin de mantener un alto nivel de precisión y rendimiento, empleando un conjunto de datos relativamente pequeño para su entrenamiento.

1.2. Objetivos

El propósito de este trabajo es el desarrollo y estudio de modelos de IA mediante la transferencia de conocimiento de modelos pre-entrenados, con el objetivo de que dichos modelos sean capaces de identificar el rostro de una persona concreta frente a cualquier otro rostro, todo ello con un rendimiento razonable en un dispositivo IoT. Para este desarrollo se ha decidido emplear la librería TensorFlow, que nos permite llevar a cabo la transferencia de conocimiento gracias a las numerosas redes pre-entrenadas que pone a nuestra disposición.

Para poder realizar un estudio completo del problema, se han definido las siguiente metas:

- Diseño de modelos mediante transferencia de conocimiento.
- Análisis de rendimiento de las etapas de entrenamiento.
- Análisis de rendimiento en la etapa de inferencia.
- El análisis de rendimiento se llevará a cabo sobre distintos dispositivos, CPU, GPU, TPU, ...
- Emplear la SDK TensorRT para optimizar el proceso de inferencia, a fin de cuantificar la mejora de rendimiento.

En cuanto al dispositivo especializado mencionado, se ha optado por el kit de desarrollo NVIDIA Jetson AGX Orin, que está pensado para desarrollar proyectos de robótica avanzada y ejecutar aplicaciones de IA en el borde (*AI at the edge*). Más adelante en este documento entraremos en detalle sobre las características de este dispositivo.

Capítulo 2

Estado del arte

Hemos hecho mención con anterioridad al auge y expansión de la IA más allá de los campos con los que se relacionaba en sus inicios, como son la investigación o la computación de altas prestaciones, no en vano, uno de los mayores hitos logrados en la breve historia de la Inteligencia Artificial fue logrado por el supercomputador de IBM Deep Blue[1]. Así pues, podemos preguntarnos a qué se debe esta escalada de la IA, y encontraremos que la respuesta puede resumirse en dos aspectos clave, el incremento en potencia de cálculo de los dispositivos actuales y la ingente cantidad de información almacenada de la que disponemos hoy día.

Resulta intuitivo que estos dos factores están estrechamente relacionados entre sí, por razones obvias, puesto que procesar grandes cantidades de información requiere grandes capacidades de cómputo y este proceso ya no está limitado a grandes centros de datos o al uso de supercomputadores. La proliferación de la IA también ha hecho hecho que sus casos de uso aumenten proporcionalmente en tareas cada vez más complejas, como es la visión artificial o el procesamiento de lenguaje natural, y es precisamente ahí donde el *Deep Learning* gana un protagonismo incontestable frente a los algoritmos clásicos de IA[16].

Difícilmente los algoritmos tradicionales de IA dejarán de emplearse, pero el sorpasso que han sufrido éstos por parte de aquellos que componen el *Deep Learning* es palpable e incontestable. Medir el impacto que tendrán los sistemas que empleen *Deep Learning* en el futuro es un tema de constante estudio y debate[24], donde se tienden a realizar previsiones optimistas. El mejor ejemplo que podemos encontrar es sin duda ChatGPT, el *chatbot* que unos meses antes de la redacción de este documento asombró al mundo, debido a que, según palabras del CEO de una gigante tecnológica como NVIDIA, Jensen Huang, en diez años los modelos predictivos de IA serán un millón de veces más potentes [9] que el propio ChatGPT.

A continuación veremos algunas de las desventajas del *Deep Learning*, que por mucho que su potencial nos quiera hacer ignorar, sin lugar a dudas existen, y que hemos decidido tratar en este proyecto. Además veremos diversos casos de uso en los que se han aplicado y estudiado algoritmos de *Deep Learning*, tanto en sectores profesionales como en dispositivos específicos.

2.1. Problemática

Ya ha sido mencionado la dificultad asociada a la elaboración, desde cero, de una red neuronal. Usando uno de los muchos frameworks disponibles que hay para este propósito, podríamos diseñar una red sin mucho esfuerzo y en poco tiempo añadiendo una serie de capas una tras de otra, no obstante, esa red no podría ni compararse a las que se emplean en las aplicaciones que vemos en el mercado, que cuentan por millones las neuronas y conexiones entre éstas que conforman dichas redes, como veremos en el capítulo 4.

Con el aumento en la complejidad de los problemas a resolver, se necesitan modelos cada vez más específicos, lo que supone que la red debe diseñarse con cierto grado de especificidad. No obstante, los modelos diseñados para una tarea específica se pueden emplear para realizar otro cometido de naturaleza similar, aspecto al que sacaremos provecho. En este trabajo seleccionaremos redes neuronales pre-entrenadas en la tarea de reconocimiento de imágenes para, mediante la transferencia de conocimiento, adaptarla a nuestra propia tarea de reconocimiento facial.

Otro inconveniente al que nos enfrentamos es el tamaño de nuestro conjunto de datos. En cualquier proyecto de inteligencia artificial es importante recabar la mayor cantidad posible de datos a fin de poder entrenar a nuestros modelos con datos los más variados posibles, para que generalizar de forma eficaz, y es precisamente cuando procesamos una gran cantidad de datos cuando los algoritmos de *Deep Learning* destacan sobre el resto. En este trabajo se ha diseñado un sistema de reconocimiento facial que diferencie un rostro en específico frente al resto, así que por un lado necesitamos rostros de diferentes personas, algo fácil de conseguir de bases de datos públicas, pero para el rostro específico debemos recabar las imágenes de forma manual, así que nuestro conjunto de datos final no será de una magnitud en absoluto similar a las bases de datos existentes.

Como última dificultad a afrontar será examinar el cómo puede encajar nuestro modelo en el entorno de del IoT. Por norma general, en el IoT nos encontramos con dispositivos limitados en cuanto a sus especificaciones técnicas.¿Puede nuestro modelo ejecutarse en un dispositivo de este estilo? ¿Su ejecución alcanza un rendimiento aceptable? Estas preguntas aparecen cuando el *Deep Learning* y el IoT se relacionan, puede que en ámbitos como el industrial esté más desarrollado[14], pero en otros como el doméstico, sin duda quedan barreas por superar[20].

Con la intención de estudiar el impacto que tienen estas dificultades en el desarrollo, haremos uso de redes pre-entrenadas mediante la transferencia de conocimiento, con lo que podemos hacer uso de una red compleja ya entrenada con gran volumen de datos, a la vez analizamos su rendimiento en un dispositivo IoT frente a un ordenador de escritorio (DC por sus siglas en inglés); si bien es cierto que el dispositivo elegido, la NVIDIA Jetson, tiene unas especificaciones relativamente más potentes que los dispositivos más comunes en el IoT.

2.2. Trabajos existentes

En esta sección veremos algunos trabajos e investigaciones existentes en diferentes ámbitos en los que se ha intentado implantar una solución basada en *Deep Learning*, que se encuentran en el contexto de nuestra aproximación. Algunos de los proyectos son relevantes por el uso de redes neuronales en dispositivos específicos, similar a nuestro caso con la NVIDIA Jetson, y la investigación de formas distintas de su aceleración, con el objetivo de conseguir mejoras de rendimiento.

Como hemos mencionado anteriormente, existen pocos campos que no hayan aplicado, o cuanto menos investigado, una solución basada en *Deep Learning* ya sea para resolver un problema o realizar un proceso existente de una manera más eficiente. Podemos empezar por la industria automovilística, en la que podemos encontrar múltiples casos de uso enfocados en la visión por computador, como muestran en su artículo diversos expertos, como por ejemplo el trabajo detallado en [15], entre cuyas contribuciones podemos destacar la creación de un *dataset* con varios tipos de vehículos desde distintas perspectivas, con el cual ayuda a la realización de futuros proyectos en este área, además de probar la efectividad de sistemas como el de la inspección basada en visión artificial, al conseguir una precisión en torno al 85 %.

Uno de los sectores que más provecho está sacando de los sistemas basados en *Deep Learning* es el industrial, que está relacionado con el sector automovilístico que mencionamos recientemente. Tomemos como ejemplo el estudio desarrollado en [32], en el que se nos muestra un caso de uso de interés que consiste en determinar tiempo de vida útil que le resta al equipamiento y elementos de una línea de producción. Introduciendo el principio de electrocardiograma de dispositivos, que en pocas palabras consiste en proporcionar un visión detallada del estado de los dispositivos a la vez que se reduce la dependencia del conocimiento por parte de expertos. Los resultados muestran una mejora notable en la capacidad de respuesta y fiabilidad frente a los sistemas tradicionales, con una precisión muy alta a la hora de predecir la vida útil restante.

Un campo que siempre mira con escepticismo los sistemas en los que las máquinas toman decisiones sin supervisión es la medicina, que una mala predicción no supone únicamente un riesgo económico o material, sino que afecta directamente a la salud de las personas. Por este motivo, las soluciones propuestas comúnmente so valoran con recelo, a la par que exigen unos valores de precisión mínimos bastante elevados. Relacionado con el IoT, el algoritmo propuesto en [30] está basado en algoritmos tradicionales de aprendizaje de diccionario y consiste en analizar información multimedia captada por dispositivos móviles para la clasificación de enfermedades. Es importante destacar que si bien el algoritmo alcanza un rendimiento notable en cuanto a precisión y especificidad, su función es orientar a los expertos en el diagnóstico de la enfermedad, no en realizar el diagnóstico de forma autónoma.

Veamos un último ejemplo de campo en el que el *Deep Learning* puede tener cabida, la educación. Podríamos afirmar que junto a la medicina, la educación es el ámbito que más complejidad puede haber al implantar soluciones basadas en esta tecnología. En el artículo [18] se describe la compleja relación que tienen ya no solo los sistemas de *Deep Learning*, sino la IA en conjunto en el ámbito educativo, en el que se analizan diversas aproximaciones hechas para introducir elementos de IA en entornos de aprendizaje online; junto a otros trabajos realizados por científicos de datos para predecir indicadores de rendimiento empleando *Deep Learning*. En definitiva, todas estas aproximaciones no están exentas de problemas que ponen en duda la capacidad de la IA de realizar las evaluaciones objetivas y precisas que requiere el sistema educativo, ya sea por problemas técnicos como datos defectuosos o imprecisos, o por la falta de consenso en la elección de los criterios de evaluación.

Dejando a un lado los ámbitos de aplicación, otro campo de investigación de interés en torno al *Deep Learning* es el uso de dispositivos destinados a la aceleración de las redes neuronales. Por un lado podríamos hablar de la diferencia de rendimiento entre la CPU y la GPU, lo que trataremos en este proyecto, pero los trabajos más recientes y de interés están realizándose sobre los dispositivos conocidos como FPGA. Una matriz de puertas lógicas programable, o FPGA, no es tecnología nueva, fue inventada por Ross Freeman en 1984[31], pero los avances tecnológicos la han convertido en un hardware a tener en cuenta. Al igual que en este proyecto emplearemos la NVIDIA Jetson para medir el rendimiento de las redes, se han hecho numerosas investigaciones en torno las posibilidades que ofrecen las FPGAs, principalmente en la tarea de inferencia. Un buen ejemplo de esto es el proyecto de investigación desarrollado en [3], en el que vemos cómo diferentes redes pre-entrenadas obtienen aceleraciones entre 1,2 y 1,5 respecto a sus implementaciones más modernas.

Los aceleradores y su análisis, ya no solo GPU y FPGA, sino también otros como los ASICs (circuito integrado específico de aplicación, por sus siglas en inglés), continúan siendo un campo de investigación bastante fructífero, en parte debido a que están estrechamente ligados a un elemento que está en constante evolución como el hardware. En el trabajo descrito en [22] podemos encontrar un buen análisis del rendimiento y consumo que consiguen estos dispositivos frente una CPU Intel, tipo de procesador utilizado con frecuencia en aplicaciones integradas.

Hemos visto la aceleración mediante hardware, terminaremos esta sección con diversos estudios en la aceleración mediante software. En este trabajo realizaremos la optimización de los modelos mediante la biblioteca de NVIDIA TensorRT, la que explicaremos con más detalle en capítulos posteriores, no obstante, las aproximaciones que hay para este propósito son tan abundantes como diversas. Si bien hemos hablado de emplear software para conseguir una aceleración de rendimiento, esto puede traducirse en diseñar instrucciones específicas para obtener un mejor rendimiento del hardware. El desarrollo realizado en [4] es un buen ejemplo de

esto, en él se describe como mediante la introducción de instrucciones en dos procesadores Intel y ARM pueden explotar su capacidades multi-hilo y de paralelismo, consiguiendo un rendimiento competitivo con sólo añadir un coste asumible en cuanto al rendimiento computacional.

Un proyecto dedicado a la optimización de los algoritmo de *Deep Learning* en el campo del IoT que merece la pena hacer mención, es el proyecto 'VEDLIoT' [11]. Este es un proyecto de la unión Europea destinado a aprovechar al máximo el aumento de la capacidad de procesamiento de los dispositivos de IoT para el diseño y optimización de algoritmos que operen de manera eficiente para satisfacer las creciente necesidades en campos como la industria, el transporte y el hogar.

Terminamos con la revisión de proyectos de ML sobre el reconocimiento facial. Podemos tomar como ejemplo el estudio desarrollado en [7] en el que se presenta un sistema de reconocimiento facial con dispositivos IoT, a fin de mejorar la seguridad de los sistemas de autenticación en sistemas médicos inteligentes. Los resultados muestran un alto nivel de precisión con distintas redes neuronales en un dispositivo de bajo consumo como la Raspberry Pi 3, a costa de un tiempo de procesamiento alto.

También desarrollado sobre un módulo Raspberry Pi 3, podemos encontrar el desarrollo de un sistema que emplea el reconocimiento facial para controlar la apertura de la entrada de un hogar inteligente, tal y como se describe en [21]. Con el inconveniente de un tiempo de procesamiento excesivo, el sistema desarrollado es perfectamente funcional empleando un módulo de especificaciones limitadas.

Los trabajos antes mencionados constatan que el rendimiento es un problema común al que los desarrolladores deben enfrentarse cuando se quiere hacer de sistemas basados en *Deep Learning* sobre dispositivos IoT. Razón por la que el proceso de optimización es un aspecto relevante de este proyecto.

Capítulo 3

Datos y dispositivos

En este capítulo haremos un repaso al *dataset* o conjunto de datos que emplearemos para entrenar y validar los diferentes modelos de IA, junto a los dispositivos en que serán ejecutados a fin de comparar los rendimientos obtenidos.

3.1. Datos

Esta sección esta destinada a explicar cómo se divide nuestro *dataset*, las diferentes fuentes de los datos y por último el tratamiento que se les ha dado antes de ser usados en el desarrollo de los modelos.

3.1.1. Composición

Para este proyecto se ha elegido como aplicación un sistema de reconocimiento facial, que, al formar parte de la disciplina de la visión artificial, puede trabajar tanto con vídeos o imágenes, aunque en la práctica el primero de éstos es tratado como una sucesión de imágenes; en nuestro caso estamos analizando imágenes, por lo que elaboraremos un conjunto de datos compuesta por imágenes.

Como hemos mencionado con anterioridad, la idea es desarrollar un sistema que reconozca un rostro frente al resto, siendo más precisos, reconocer cuándo un rostro pertenece y cuando no al autor de este documento. En otras palabras, tenemos dos clases de imágenes que por sencillez hemos nombrado como *'nico'* y *'not_nico'*, tal y como se indica en el nombre de cada clase, la primera está destinada al rostro del autor y la segunda a cualquier otro.

En total nuestro conjunto de datos está compuesto por un total de mil cien imágenes que se distribuyen uniformemente entre ambas clases. Como es un conjunto de datos relativamente pequeño, se han dividido manualmente, manteniendo la proporción de imágenes de cada clase, en los tres grupos siguientes:

- Setecientas imágenes para el entrenamiento.
- Trescientas imágenes para la validación.
- Cien imágenes para la verificación final del modelo.

3.1.2. Recolección

Para la clase *'not_nico'*, al simplemente se necesitan rostros sin ningún requisito específico, se han escogido imágenes de un repositorio público. En concreto se ha elegido el *'Flickr-Faces-HQ dataset'*[12], que fue creado originalmente para verificar el trabajo descrito en [13] elaborado por miembros de NVIDIA. Este conjunto de datos está compuesto por más de 70.000 imágenes extraídas del sitio web destinado al almacenamiento de contenido audiovisual Flickr. De esta fuente se terminó por escoger 550 imágenes de la categoría miniaturas, que tienen una resolución de 128x128 en formato PNG.

Para la clase *'nico'* se han tenido que recabar los datos manualmente por razones obvias. Como este proceso solo consiste en la toma de fotografías, intentando que sean lo más diversas posibles dentro de lo que cabe, no incidiremos más ello. Este conjunto también se compone de 550 imágenes, lo que nos deja las imágenes divididas uniformemente entre ambas clases.

3.1.3. Preparado

Antes de emplear las imágenes para el entrenamiento de los modelos es necesario realizar algunas modificaciones. La clase *'not_nico'* está preparada para su uso, puesto que son imágenes de proporción cuadrada con una resolución baja, lo más común al alimentar redes neuronales es emplear imágenes de una resolución como máximo de 320x320[29], y son imágenes centradas en el rostro; por otro lado, las imágenes de la clase *'nico'* son imágenes directamente tomadas de una cámara de alta resolución, por lo que debemos recortar una sección cuadrada en torno al rostro y luego redimensionar las imágenes.

Nuestro *dataset* estará listo cuando todas las imágenes sean de proporción cuadrada, con resolución 128x128 y centradas en el rostro. El redimensionado de las imágenes es posible realizarlo empleando funciones de la biblioteca TensorFlow, pero para evitar la carga que supondría realizar este proceso en cada ejecución, se optó por realizar manualmente la operación antes del desarrollo.

3.2. Dispositivos

Seguidamente haremos un breve repaso a las especificaciones de los dispositivos que emplearemos para el desarrollo y medición de los modelos, que serán uno de propósito general y otro dedicado. Nos centraremos en los componentes que más influyen en el rendimiento de las operaciones como son la unidad de procesamiento y la memoria RAM.

3.2.1. Desktop Computer (DC)

Empezando por el DC, veamos las especificaciones de la CPU, GPU y RAM:

- CPU: dispone de un procesador Intel® Core™ i9-10900 que cuenta con 10 núcleos que gracias al Hyper-Threading actúan en la práctica como 20, ade-

más dispone de una caché de 20 MB y alcanza una frecuencia máxima de 5,2 GHz.

- GPU: se trata de una GeForce RTX 3080 de NVIDIA que cuenta con 8704 núcleos CUDA y 272 núcleos tensoriales, con 10 GB de VRAM, alcanzando los 29,77 TFLOPS en punto flotante de 32 bits y con una frecuencia de reloj máxima de 1,71 GHz. Todo esto lo consigue con un consumo de 320 W.
- RAM: la memoria RAM se compone de dos módulos DDR5 que suman un total de 16 GB.

3.2.2. NVIDIA Jetson

Veamos ahora algunas de las especificaciones de la NVIDIA Jetson AGX Orin:

- CPU: Arm® Cortex®-A78AE v8.2 de 12 núcleos con dos memorias caché de 3 y 6 MB, capaz de alcanzar una frecuencia máxima de 2,2 GHz.
- GPU: se trata de arquitectura NVIDIA Ampere integrada de 2048 núcleos CUDA con 64 núcleos tensoriales, con la que alcanza una frecuencia máxima de 1,3 GHz. Cuenta con tres modos de energía con desde un mínimo de 15W hasta un máximo de 50W.
- RAM: memoria LPDDR5 de 64 GB.

En este dispositivo es importante destacar dos aspectos. En primer lugar el consumo, que varía entre los 15 y los 50 W dividido en tres configuraciones, un consumo relativamente bajo, característica común entre los dispositivos en el borde. Y luego tenemos los aceleradores, y es que la Jetson dispone de varios, algunos centrados en el procesamiento de contenido multimedia, pero el que nos atañe a nosotros es el denominado como *Deep Learning Accelerator* (DLA) que está destinado a acelerar la operaciones de inferencia de modelos de *Deep Learning*. El modelo Orin dispone de dos de estos aceleradores, que no están exentos de restricciones al ejecutar un modelo, como el tipo de capas de la red; en capítulos posteriores veremos cómo emplear esta tecnología y qué rendimiento ofrece.

3.2.3. Comparativa

La tabla 3.1 recoge las especificaciones de ambos dispositivos.

	DC	Jetson
CPU	Intel® Core™ i9-10900 - 20 núcleos - 5,2 GHz	Arm® Cortex®-A78AE v8.2 - 12 núcleos - 2,2 GHz
GPU	GeForce RTX 3080 - 8704 núcleos CUDA - 272 núcleos tensoriales	GPU de arquitectura NVIDIA Ampere - 2048 núcleos CUDA - 68 núcleos tensoriales
RAM	16 GB DDR5	64 GB LPDDR5
DLA	-	2x NVDLA v2

Tabla 3.1: Comparativa de especificaciones

La comparación componente a componente muestra una clara ventaja del DC frente a la Jetson, a excepción de la memoria RAM, por lo que inicialmente se espera que la diferencia de rendimiento está a favor del primero, sin embargo, si bien no hacemos un análisis en profundidad sobre ello, el consumo energético de la Jetson es notablemente menor, puesto que la GPU del DC consume hasta seis veces más que el máximo consumo de la Jetson.

Capítulo 4

Tecnologías y entorno

En este capítulo haremos un repaso a los conceptos y tecnologías que están estrechamente relacionadas con el desarrollo de nuestros modelos. El objetivo no es profundizar en los aspectos técnicos de cada uno, sino proporcionar una introducción para que los lectores, en caso de no tener conocimientos previos, conozcan las nociones básicas que les faciliten la comprensión.

4.1. Conceptos teóricos

En esta sección se explicarán algunos términos que están directamente relacionados con el desarrollo del proyecto, algunos comunes en los trabajos de *Deep Learning* y otros más específicos de este campo cuando trabajamos en aspectos relacionados con la visión artificial.

4.1.1. Aumentado artificial de datos

En el capítulo anterior se trataron los detalles relativos a nuestro *dataset*, que está compuesto por mil cien imágenes que se distribuyen uniformemente en dos clases. El tamaño de nuestro *dataset*, si bien no es excesivamente pequeño, no se puede comparar con los conjuntos de datos empleados para el entrenamiento de las redes neuronales de mayor entidad, como por ejemplo el *dataset* ImageNet [26], compuesto por más de catorce millones de imágenes divididas en más de veinte mil categorías, sobre el que se han entrenado las redes neuronales que emplearemos en este desarrollo. Cuanto mayor es el *dataset* mejor será la capacidad de generalización del modelo, por lo que en los casos como el nuestro, en que disponemos de pocos datos, se recurre al aumentado artificial de los datos.

El aumentado artificial de los datos, o *data augmentation*, consiste en incrementar de forma sintética nuestro conjunto de datos aplicando diferentes transformaciones sobre las imágenes originales. Como ejemplo de dichas transformaciones están la rotación, el efecto espejo, la ampliación o zoom, la inversión de colores, etc. En la figura 4.1 tenemos un ejemplo práctico de esto, en el que conseguimos imágenes nuevas a partir la original.

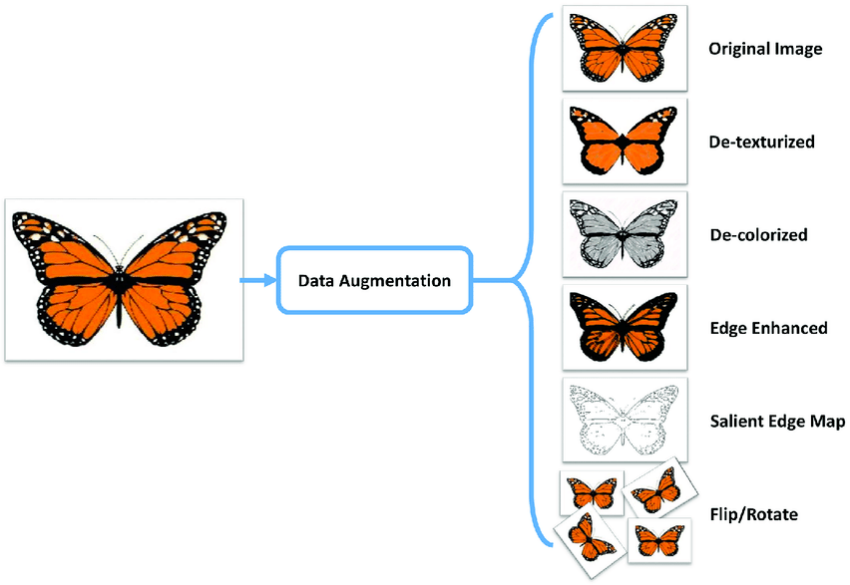


Figura 4.1: Aumentado artificial datos
Fuente: `https://medium.com`

En nuestro caso, como veremos en el capítulo posterior, aplicamos únicamente las transformaciones de espejo y una ligera rotación a nuestro *dataset*.

4.1.2. Red neuronal convolucional

Una red neuronal prealimentada puede ser descrita según el enfoque tradicional como un conjunto de nodos agrupados en capas completamente interconectadas cada una con la siguiente. No obstante, en la visión artificial no se suele emplear, sino que se opta por las conocidas como redes neuronales convolucionales. Podemos resumir las diferencias entre ambas en dos puntos:

- Si en la red neuronal prealimentada los nodos se agrupan en vectores, en la red neuronal convolucional se agrupan en matrices denominadas como tensores.
- En la red prealimentada todos los nodos de una capa están conectados con todos los nodos de la siguiente, mientras que en las convolucionales los nodos se conectan únicamente con un número reducido de la capa posterior.

Si se piensa detenidamente, es lógico que al trabajar con imágenes no se trabaje con vectores sino con tensores, puesto que una imagen es una matriz de píxeles con una profundidad según los canales de color. En la imagen 4.2 podemos ver una comparativa entre una red neuronal prealimentada (a) y una red neuronal convolucional (b).

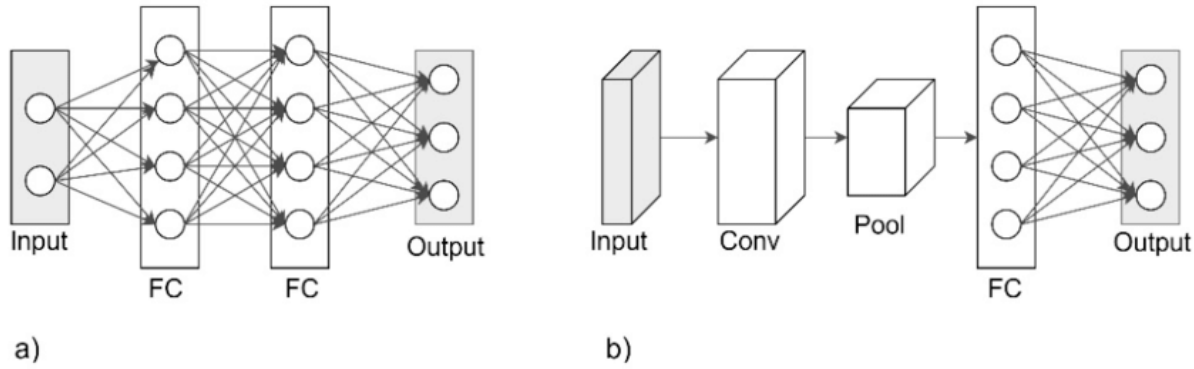


Figura 4.2: Red prealimentada vs convolucional
Fuente: `https://www.researchgate.net` [28]

4.1.3. Transferencia de conocimiento

El aprendizaje por transferencia puede ser el concepto de mayor importancia en este proyecto, ya que nos evita la tarea de crear un red neuronal desde cero, al utilizar modelos pre-entrenados para adaptarlos a nuestra tarea específica. El proceso consiste en seleccionar un modelo que haya sido entrenado para una tarea similar, en nuestro caso serán modelos que clasifican imágenes de ImageNet [26] en diferentes clases, se congelan los valores de cada nodo, conocidos como pesos, y se sustituyen las capas superiores de la red, que se encargan de la clasificación, por unas propias de nuestra tarea.

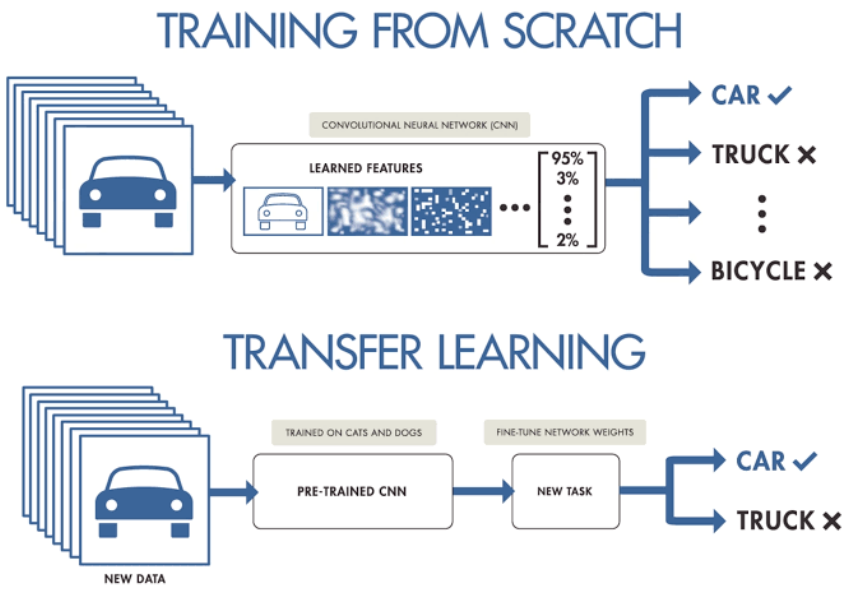


Figura 4.3: Transferencia de conocimiento
Fuente: `https://www.viewnext.com`

Como vemos en la figura 4.3, otra importante ventaja es que estamos empleando un modelo que ha aprendido a extraer características usando un gran conjunto de datos, lo que significa que podemos conseguir un modelo con gran capacidad de generalización incluso con un *dataset* pequeño como el nuestro.

4.1.4. Parada temprana

En el ámbito del aprendizaje automático, la parada temprana, o *early stopping*, es un método de regularización destinado e evitar que los sistemas iterativos como el nuestro sufran el efecto de sobre-ajuste. Consiste en dejar de entrenar el modelo en el punto en que se haya alcanzado el menor valor de error, conocido como pérdida o coste, sobre el conjunto de validación, después del cuál el entrenamiento sólo cause el sobre-ajuste en el conjunto de entrenamiento.

El criterio empleado puede variar según el problema y es objeto de estudio [19], en nuestro caso escogemos un enfoque simple en el que detenemos el entrenamiento si la pérdida no ha disminuido en un número determinado de iteraciones, con esto se puede deducir que el tiempo total de entrenamiento está directamente influenciado por la iteración en que se produce la parada temprana. En la figura 4.4 se puede observar el efecto de sobre-ajuste que se quiere evitar con la parada temprana.

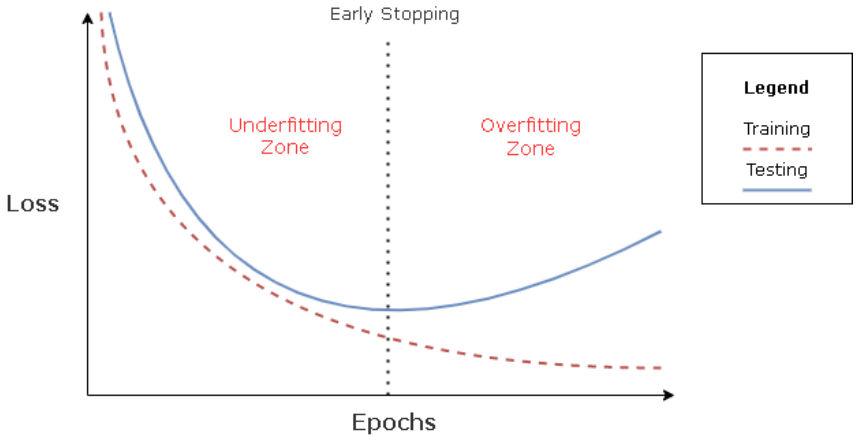


Figura 4.4: Parada temprana
Fuente: `https://towardsdatascience.com`

4.1.5. Cuantización

La cuantización o cuantificación es el proceso por el que se aproximan las operaciones en coma flotante de una red neuronal a otras con una anchura de bits menor. Con esto se busca reducir el coste computacional así como los requisitos de memoria. Este proceso está relacionado con la inferencia de los modelos y no con el entrenamiento, puesto que se busca convertir operaciones que han sido entrenadas en 32 bits en punto flotante a 16 bits en punto flotante y 8 bits de enteros.

La conversión de 32 bits en punto flotante a 16 bits es un proceso que se lleva a cabo con operaciones matemáticas independientes de la entrada, no así la conversión a enteros de 8 bits. Como vemos en la figura 4.5, la diferencia entre ambos rangos es notoria, por lo que es necesario definir tanto una función de redondeo

más compleja como el tratamiento que se le dará a las anomalías o *outliers*. La definición de estas operaciones se realiza empleando datos reales a modo de calibración.

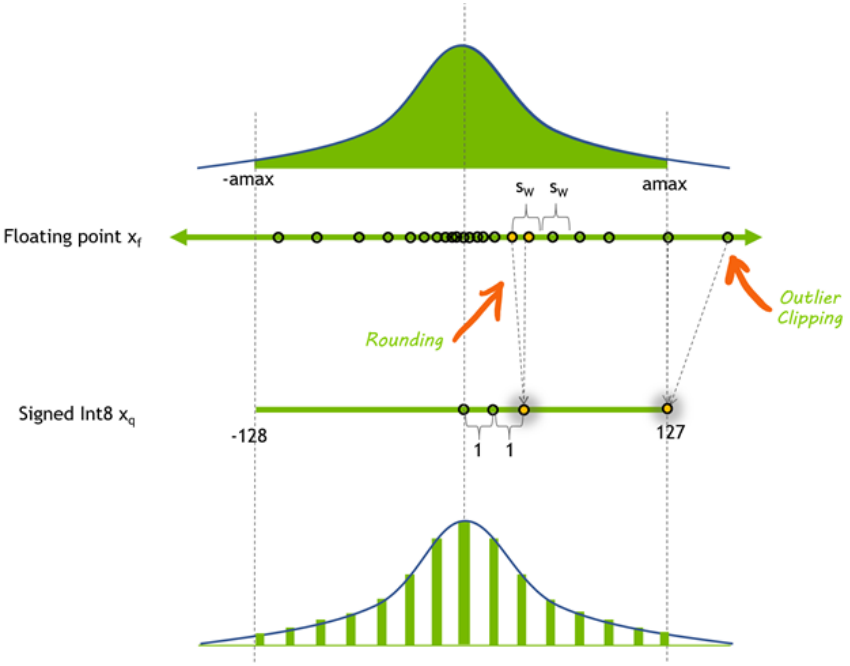


Figura 4.5: Cuantización de Fp32 a INT8
Fuente: `https://developer.nvidia.com/blog`

Las técnicas de cuantización pueden clasificarse entre cuantificación post entrenamiento (PTQ) o entrenamiento consciente de la cuantificación (QAT). Se ha optado por descartar el proceso QAT, que se basa en simular el error de cuantificación en los nodos durante el entrenamiento, debido a que el chip DLA usado en el desarrollo de este proyecto no soporta este tipo de cuantización.

El método PTQ como su nombre indica, se realiza después de haber entrenado un modelo de alta precisión. Con PTQ, cuantificar los pesos es fácil ya que se tiene acceso a los pesos de los tensores y se pueden medir sus distribuciones, sin embargo, cuantificar las activaciones es más difícil porque las distribuciones de activación deben medirse utilizando datos de entrada reales.

Para ello, el modelo entrenado en coma flotante se evalúa utilizando un pequeño conjunto representativo de los datos de entrada reales, y se recopilan estadísticas sobre las distribuciones de activación entre capas. Como paso final, se determinan las escalas de cuantización de los tensores de activación del modelo utilizando un objetivo de optimización. Este proceso se denomina calibración y el conjunto de datos representativo utilizado se le conoce como conjunto de calibración.

Como es de esperar, reducir la precisión de las operaciones de esta manera no solo provoca una reducción de la carga computacional, también implica que la

precisión final del modelo pueda verse afectada. Una parte importante de este trabajo es medir el impacto positivo y negativo que provoca este método sobre nuestros modelos.

4.2. Tecnologías

A continuación daremos a conocer de forma concisa las principales tecnologías empleadas en el proyecto, TensorFlow y TensorRT, para las fases de desarrollo y aceleración de los modelos respectivamente.

4.2.1. TensorFlow

Desarrollado originalmente por investigadores e ingenieros que trabajaban en el equipo Google Brain, TensorFlow es una plataforma integral de código abierto para el aprendizaje automático. Cuenta con un ecosistema completo y flexible de herramientas, bibliotecas y recursos comunitarios que permite a los investigadores impulsar el estado del arte en ML y a los desarrolladores crear y desplegar fácilmente aplicaciones potenciadas por ML.

TensorFlow tiene en su haber cientos de modelos pre-entrenados con múltiples versiones, que se dividen en los dominios de texto, imágenes, vídeo y audio. En la sección de Redes pre-entrenadas de este capítulo veremos con más detalle qué redes se han escogido como base para este proyecto.

4.2.2. NVIDIA TensorRT

NVIDIA TensorRT es una SDK para la inferencia de *Deep Learning* de alto rendimiento, incluye un optimizador de inferencia de *Deep Learning* y de tiempo de ejecución que ofrece tanto baja latencia como un alto rendimiento en aplicaciones de inferencia. Para su uso, en este proyecto nos valdremos de la API disponible en el lenguaje Python.

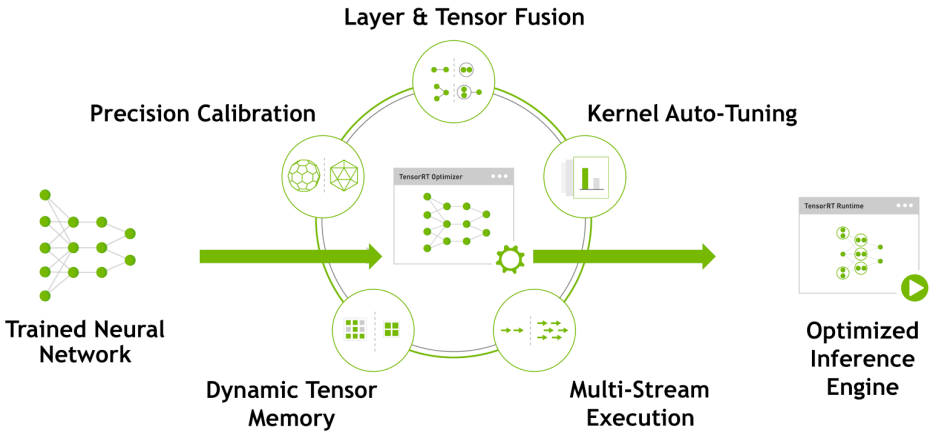


Figura 4.6: Flujo de trabajo de TensorRT
Fuente: `https://developer.nvidia.com/blog`

TensorRT trabaja en dos fases. En la primera fase se proporciona a TensorRT una definición del modelo, que será optimizado para una GPU de destino; en la figura 4.6 podemos ver los diferentes procesos llevados a cabo por TensorRT en esta etapa. En la segunda fase, se utiliza el modelo optimizado para ejecutar la inferencia. El modelo resultante del proceso de mejora sólo obtendrá el máximo rendimiento si se ejecuta en el mismo dispositivo en el que ha sido optimizado.

Los aceleradores de *Deep Learning* disponibles en la NVIDIA Jetson, comentados en el capítulo anterior, tienen una forma de uso recomendada basada en el entrenamiento de un modelo para que posteriormente sea optimizado empleando TensorRT, aunque es cierto que existen alternativas como la API cuDLA, que no es más que una extensión de CUDA para programar la ejecución de redes neuronales en los chips DLA.

Dependencias

La librería TensorRT tiene dos dependencias que son lo suficientemente relevantes como para ser mencionadas, aunque sea someramente. En primer lugar está CUDA Toolkit, en el que podemos crear aplicaciones de alto rendimiento aceleradas en la GPU. En otras palabras, contiene las herramientas para poder programar aplicaciones que hagan uso de la GPU.

En segundo lugar está cuDNN, *CUDA Deep Neural Network*, librería para la aceleración en la GPU de primitivas para redes neuronales profundas. Esta librería proporciona implementaciones optimizadas para rutinas estándar como la convolución hacia delante y hacia atrás, la agrupación, la normalización y las capas de activación. Como se puede comprobar, ambas dependencias se centran en el uso de la GPU para la aceleración, poniendo el foco en las redes neuronales, lo que nos sirve de indicativo de por qué son un requisito fundamental dentro de TensorRT.

4.2.3. Versiones del entorno

Como el uso de una u otra versión puede influir en los resultados o producir incompatibilidades, en la tabla 4.1 se muestran las versiones de las herramientas empleadas, de forma que puedan reproducirse los resultados en caso que el lector lo requiera:

Herramienta	**Versión**
Python	3.8.10
TensorFlow	2.11.0
CUDA Toolkit	11.4.4
cdDNN	8.6.0
TensorRT	8.5.2

Tabla 4.1: Versiones del entorno de desarrollo

4.3. Redes pre-entrenadas

Terminamos el capítulo describiendo las rede que redes emplearemos como base para la transferencia de conocimiento en la fase de desarrollo. En la anterior sección se hizo alusión a la gran cantidad de modelos disponibles en TensorFlow, razón por la que se hizo una revisión concisa a fin de determinar cuáles emplear [10]. El grupo definitivo está compuesto por las redes MobileNet-v2, ResNet50, ResNet152, Xception y Vgg16; se han buscado redes con cierta variación en cuanto al número de parámetros que las componen y la carga computacional de sus operaciones.

4.3.1. MobileNet-v2

La red MobileNet-v2 [6] es una red neuronal convolucional con 53 capas de profundidad y ha sido entrenada para clasificar imágenes en 1000 categorías de objetos. Como su nombre indica, el modelo MobileNet está diseñado para ser utilizado en aplicaciones móviles, y es el primer modelo de visión por ordenador móvil de TensorFlow.

MobileNet utiliza convoluciones separables en profundidad. Reduce significativamente el número de parámetros en comparación con la red con convoluciones regulares con la misma profundidad en las redes. Como resultado se obtienen neuronales profundas y ligeras.

4.3.2. ResNet50

ResNet-50 [5] es una red neuronal de tipo residual de 50 capas. Las redes neuronales residuales son un tipo de red neuronal artificial (RNA) que forma redes apilando bloques residuales.

La ResNet de 50 capas utiliza un diseño de cuello de botella para el bloque de construcción. Se emplea un bloque residual [23] que utiliza convoluciones 1×1, lo que se conoce como 'cuello de botella', a fin de reducir el número de parámetros y multiplicaciones de matrices. Esto permite un entrenamiento mucho más rápido de cada capa respecto a su definición original de 34 capas, la red ResNet34.

4.3.3. ResNet152

Otra implementación de la red ResNet [5] en la que, como su nombre indica, el número de capas asciende hasta las 152; por lo demás, no hay diferencias dignas de mención respecto de la red ResNet50.

4.3.4. Xception

La red Xception [2] es una arquitectura de red neuronal convolucional profunda que implica convoluciones separables en profundidad. Xception son las siglas de 'extreme inception', que lleva al extremo los principios de la red Inception [27]. En Inception, se utilizaban convoluciones 1x1 para comprimir la entrada original, y a partir de cada uno de esos espacios de entrada se utilizaban distintos tipos de

filtros en cada uno de los espacios de profundidad. Xception simplemente invierte este paso. En su lugar, primero aplica los filtros en cada uno de los mapas de profundidad y luego comprime el espacio de entrada utilizando la convolución 1X1 aplicándola en toda la profundidad.

4.3.5. Vgg16

La red Vgg16 [25], como su nombre indica, es una red neuronal profunda de 16 capas. VGG16 es una red relativamente extensa, con un total de 138 millones de parámetros, enorme incluso para los estándares actuales. Sin embargo, la sencillez de la arquitectura de VggNet16 es su principal atractivo. Una red Vgg está formada por pequeños filtros de convolución. Vgg16 tiene tres capas totalmente conectadas y 13 capas convolucionales.

4.3.6. Comparativa

Para finalizar este capítulo hacemos este capítulo haciendo una breve comparativa de las redes neuronales antes mencionadas utilizando como referencia las implementaciones disponibles en TensorFlow. En la tabla 4.2 ponemos el foco sobre la profundidad de cada red y la cantidad de parámetros que la conforman. En esta sección hemos hecho referencia en varias ocasiones al número de parámetros, con esto nos estamos refiriendo a la suma de los nodos o neuronas y las interconexiones de la red al completo. Veremos que en la tabla la profundidad de la red en ocasiones es distinta a la descrita anteriormente, esto se debe al cómo se ha implementado en TensorFlow, no obstante, el funcionamiento de la red en sí mismo no varía.

Red	Profundidad (capas)	Parámetros
MobileNet	105	3.538.984
ResNet50	107	25.636.712
ResNet152	311	60.419.944
Xception	81	22.910.480
Vgg16	16	138.357.544

Tabla 4.2: Comparativa de redes neuronales

Dado que en este proyecto estamos sustituyendo las capas superiores del modelo original, destinadas a la clasificación, por unas propias, los números finales de nuestro modelo pueden variar pero no de manera significativa.

Capítulo 5

Desarrollo y metodología

Este capítulo abarca los aspectos relativos al desarrollo del proyecto. Se ha dividido en dos tareas fundamentales, el entrenamiento de los modelos, en el que se detallan los pasos seguidos y los componentes empleados en este proceso empleando TensorFlow, y la inferencia, que entre otros elementos recoge la información relativa al uso de TensorRT. Por último veremos un prototipo inicial en que se realizan predicciones en tiempo real sobre una entrada de vídeo.

5.1. Entrenamiento

Tal y como se explicó en el Capítulo 3, para las tareas de entrenamiento, validación y testeo de los modelos disponemos de mil cien imágenes que se dividen uniformemente en dos clases, '*nico*' y '*not_nico*'. Como desde TensorFlow cargamos los conjuntos desde un directorio, las clases se diferencian según el nombre del directorio , de esta forma el conjunto de entrenamiento es un directorio que a su vez se compone de dos directorios con los nombres '*nico*' y '*not_nico*', que contienen las imágenes. En el siguiente bloque de código puede observarse la carga de los datos de entrenamiento:

```
1  import tensorflow as tf
2  from tensorflow import keras
3  from keras.utils import image_dataset_from_directory
4
5  BATCH_SIZE = 32
6  IMG_SIZE = (128, 128)
7
8  train_ds = image_dataset_from_directory(train_dir,
9                                          shuffle=True,
10                                         batch_size=BATCH_SIZE,
11                                         image_size=IMG_SIZE)
```

Listado 5.1: Carga de los datos de entrenamiento

La primera fase del entrenamiento, después de haber cargado los tres conjuntos de imágenes, consiste en importar el modelo desde el repositorio de TensorFlow,

en el que indicaremos que ha sido entrenado sobre la base de datos de ImageNet y que queremos prescindir de las capas superiores del mismo. También se necesita especificar las dimensiones de los datos de entrada, que son de la forma '(128,128,3)' por las dimensiones de las imágenes y los tres canales de color. Puede verse a continuación un ejemplo de la red MobileNet:

```
tf.keras.applications.MobileNetV2(  input_shape=input_shape,
                                    include_top=False,
                                    weights='imagenet')
```

Listado 5.2: Importando la red MobileNet

Una vez tenemos el modelo base podemos crear nuestro propio modelo. Este modelo se compone de cinco elementos, el aumentado artificial de datos, el preprocesamiento de los datos, el modelos base, una capa de agrupación y una última capa que se encarga de la clasificación. El aumentado de los datos, como ya se mencionó, consiste en transformaciones de rotación y de espejo; en cuanto al preprocesado, en TensorFlow cada red neuronal pre-entrenada tiene su propia tarea de pre-procesamiento, en el caso de MobileNet se reescalan los valores del rango [0,255] a [-1,1]. Tras estas dos etapas se alimenta al modelo base.

El modelo base se dedica a la extracción de características de la entrada proporcionada, y produce una salida que se envía a una capa de agrupación que se encarga de calcular las medias para convertir un tensor en un vector. El vector está completamente conectado a la última capa que se encarga de la clasificación, que produce un único resultado. A pesar de tener dos clases, nuestro modelo sólo retorna un valor, esto se debe a que el resultado obtenido es una unidad que se conoce como '*logit*', que no es otra cosa que un valor de predicción en bruto. Los valores positivos pertenecerán a la clase 1 mientras que los negativos a la clase 0; esto podemos hacerlo gracias a que nuestro modelo es binario, es decir, sólo tiene dos clases. La ventaja de hacerlo de esta forma es que no nos hace falta emplear una función de activación [8], que es necesaria para permitir que las redes neuronales aprendan patrones complejos, aunque añaden más carga computacional a la red.

Veamos cómo se traduce todo esto en el código:

```
# Data augmentation
data_augmentation = tf.keras.Sequential([
tf.keras.layers.RandomFlip('horizontal'),
tf.keras.layers.RandomRotation(0.2),
])

# Extract initial features from base_model
feature_batch = base_model(image_batch)
# Set layers to non-trainable
```

```
base_model.trainable = False

# Add custom layers on top
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
feature_batch_average = global_average_layer(feature_batch)
prediction_layer = tf.keras.layers.Dense(1)
prediction_batch = prediction_layer(feature_batch_average)

inputs = tf.keras.Input(shape=(128, 128, 3))
x = data_augmentation(inputs)
x = preprocess_input(x)
x = base_model(x, training=False)
x = global_average_layer(x)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = prediction_layer(x)
model = tf.keras.Model(inputs,outputs,name="FaceClassifier")
```

Listado 5.3: Diseño del modelo

Podemos destacar el cómo debemos especificar que no modifique los pesos del modelo con *'base_model.trainable = False'*, puesto que no tendría sentido usar un modelo pre-entrenado si lo modificamos al completo con cada iteración. Dicho esto, disponemos de un modelo de gran complejidad prácticamente listo para ser entrenado con sólo añadir unas pocas capas.

Para que el modelo sea completamente operativo es necesario compilarlo. El proceso de compilado consiste en detallar cómo se debe realizar el entrenamiento mediante la especificación de tres componentes:

- El optimizador: técnica o algoritmo utilizado para disminuir la pérdida (el error) ajustando diversos parámetros y ponderaciones, con lo que se minimiza la función de pérdida, proporcionando una mayor precisión del modelo con mayor rapidez.
- La función de pérdida: función que nos indica cuánto nos hemos equivocado con nuestras predicciones, es decir, cuanto de alejadas están nuestras predicciones de la realidad.
- Las métricas: en sentido estricto, este elemento no es necesario, pues sólo sirve como retroalimentación para el desarrollador pero difícilmente se conocerá la bondad del modelo modelo si no se dispone de elementos para cuantificarlo.

Para el optimizador haremos uso de la estimación adaptativa del momento (Adam, por sus siglas en inglés), que, sin entrar en detalles de las operaciones matemáticas que realiza, calcula los gradientes actuales a partir de gradientes anteriores. Los optimizadores emplean un parámetro para medir cuánto de bruscos

pueden ser los cambios en los pesos de las neuronas, cuyo principio básico es que sea menor cuanto más fino es el ajuste que queremos realizar; en nuestro caso, como sólo queremos ajustar las capas superiores, emplearemos un valor considerablemente bajo.

La función de pérdida se escoge en base a la clase de nuestro problema, que podemos definir como categórico y binario. Para este tipo de problemas TensorFlow nos proporciona la función *BinaryCrossentropy*, en la que podemos especificar que se mida a partir de '*logits*'. Para las medidas, indicaremos que queremos que nos muestre la pérdida como el valor de la función anterior, junto a la precisión binaria.

El resultado final es el siguiente:

```
base_learning_rate = 0.0001
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=
    base_learning_rate),
              loss=tf.keras.losses.BinaryCrossentropy(from_logits
                 =True),
              metrics=[tf.keras.losses.BinaryCrossentropy(
                      from_logits=True, name='binary_crossentropy
                         '),
                      keras.metrics.BinaryAccuracy()])
```

Listado 5.4: Compilación del modelo

Como último paso antes de comenzar el entrenamiento, definiremos una *callback* para la ejecución de la parada temprana que vimos en el anterior capítulo. TensorFlow dispone de varias *callbacks* que podemos usar, entre las que está *EarlyStopping*, que simplemente requiere que se le indique el valor a monitorizar, las ejecuciones que pueden realizarse sin mejora antes de la parada, y que retorne los pesos a cuando se obtuvo el mejor resultado.

```
from tensorflow.keras.callbacks import EarlyStopping

earlystop = EarlyStopping(monitor='val_binary_crossentropy',
                          restore_best_weights=True,
                          patience=10,
                          verbose=1)
callbacks = [earlystop]
```

Listado 5.5: Definición de parada temprana

Ahora podemos ejecutar el entrenamiento, que se trata de un proceso iterativo en el que alimentamos la red con el conjunto de entrenamiento y ajustamos los valores de la red usando el conjunto de validación como referencia. El último paso

es evaluar el modelo ya entrenado sobre el conjunto de testeo, que aportará los valores finales de precisión y pérdida.

```
model.fit(
  train_ds,
  epochs=300,
  callbacks=callbacks,
  validation_data=val_ds)

# TESTING THE MODEL
loss, crossentropy, accuracy = model.evaluate(test_ds)
```

Listado 5.6: Entrenamiento del modelo

Para finalizar, es necesario indicar la unidad de procesamiento, CPU o GPU, que debe realizar el entrenamiento. TensorFlow facilita en gran medida esto, puesto que permite hacerlo haciendo uso de la instrucción *with tf.device(/GPU:0")*:. De esta forma, podemos especificar no sólo el emplear (CPU o GPU), sino en caso de tener varias se puede especificar una en concreto, con TensorFlow encargándose de las tareas de sincronización y transferencia de datos cuando se usa la GPU. Existen estrategias para la sincronización si quieren emplearse de forma simultánea múltiples GPUs, máquinas o TPUs, sin embargo están fuera del alcance de este proyecto.

Por último es necesario tratar el resultado de la predicción, el '*logit*'. Siguiendo la documentación de la librería, una vez tenemos el resultado le aplicamos una función sigmoide, de forma que obtenemos un resultado acotado dentro del rango [0,1]. Con este rango definimos las siguientes reglas:

- Si es menor de 0,5 pertenece a la clase 0.
- Si es mayor o igual a 0,5 pertenece a la clase 1.

El siguiente es un ejemplo de cómo seleccionamos un lote de imágenes desde el conjunto de testeo, para ejecutar la predicción y su posterior tratamiento. El resultado final es un vector con las clases en las que han sido etiquetadas las imágenes de entrada.

```
# Retrieve a batch of images from the test set
image_batch, label_batch = test_ds.as_numpy_iterator().next()
predictions = model.predict_on_batch(image_batch)

# Apply a sigmoid since our model returns logits
predictions = tf.nn.sigmoid(predictions)
predictions = tf.where(predictions < 0.5, 0, 1)
```

Listado 5.7: Ejemplo de predicción

5.2. Inferencia

La sección de inferencia será dividida en dos partes. En la primera se expondrá cómo hacer uso de TensorFlow para medir la latencia y capacidad de procesamiento de los modelos generados. En la segunda se detallará el proceso para crear lo que en TensorRT se conoce como motor (*engine*) a partir del modelo original de TensorFlow, para luego poder tomar las mismas medidas que en el caso anterior.

Antes de continuar, aclaremos algunos aspectos relevantes relativos a la inferencia:

- Por latencia entendemos el tiempo que se tarda en realizar la inferencia, donde en el caso de la GPU esto implica sumar el tiempo de copia al y desde dicho dispositivo junto al tiempo de inferencia en sí mismo. El resultado es medido en milisegundos.
- Por capacidad de procesamiento nos referimos a la cantidad de imágenes que nuestro modelo es capaz de inferir por unidad de tiempo. En inglés se le conoce como *throughput* y lo medimos como imágenes por segundo.
- La inferencia, a fin de obtener el máximo rendimiento, la realizamos sobre lotes de imágenes en lugar de imágenes individuales.
- Todas las inferencias serán medidas en la GPU, puesto que TensorRT se ejecuta únicamente en dichos dispositivos además del DLA.

5.2.1. Inferencia con TensorFlow

El proceso de inferencia con el modelo original de TensorFlow puede llevarse a cabo realizando las predicciones con el ejemplo visto en la sección anterior, no obstante, a fin de asemejar este proceso con el empleado con TensorRT, llevaremos a cabo la inferencia con unos ligeros cambios a lo visto en el ejemplo.

```
1  model = tf.saved_model.load(input_saved_model, tags=[
       tag_constants.SERVING])
2  infer = model.signatures['serving_default']
3
4  N_warmup_run = 50
5  N_run = 1000
6  elapsed_time = []
7
8  print('Warming up for 50 batches...')
9  for i in range(N_warmup_run):
10   labeling = infer(image_batch)
11
12 print('Benchmarking inference engine...')
13 for i in range(N_run):
14     start_time = time.time()
```

```
15      labeling = infer(image_batch)
16      end_time = time.time()
17      elapsed_time = np.append(elapsed_time, end_time -
            start_time)
```

Listado 5.8: Ejecución de inferencias en TensorFlow

En la primera parte estamos cargando el modelo y guardando la función de inferencia por defecto, tras lo que inicializamos algunas variables. Seguidamente comenzamos la inferencia donde, como podemos observar, estamos realizando un número reducido de inferencias a modo de calentamiento, esto es necesario porque la GPU entra en modo reposo cuando no se utiliza para reducir el consumo energético. Con el calentamiento terminado realizamos un total de mil inferencias sobre el mismo conjunto de imágenes, midiendo el tiempo que tarda en completarse la operación.

Multiplicando el número de iteraciones por el tamaño del lote imágenes y dividiendo por el tiempo total, tenemos la capacidad de procesamiento total del modelo, mientras que la media de los tiempos almacenados nos da el tiempo medio de inferencia.

```
1       mean_step_time = elapsed_time.mean() * 1000
2       throughput = N_run * BATCH_SIZE / elapsed_time.sum()
3
4       print('Throughput: {:.0f} images/s'.format(throughput))
5       print('Step time: {:4.3f}ms'.format(mean_step_time))
```

Listado 5.9: Resultado de la medición en TensorFlow

La precisión la medimos de manera muy similar, pero en lugar de emplear un único lote, empleamos el conjunto de testeo. La precisión final es la comparación entre las predicciones y las clases reales, con el inconveniente de que deben especificarse a mano en forma de vector con ceros y unos. Con la función *evaluate* podemos conseguir el mismo resultado, así que el código para medir la precisión lo podemos ver en la sección de TensorRT, ya que no dispone una función predefinida para ello.

5.2.2. Inferencia con TensorRT

Existen dos formas principales de emplear TensorRT sobre un modelo de TensorFlow, la primera y más directa es emplear las funciones integradas en TensorFlow para ello, que se conoce como TF-TRT. Esta integración se incluye por defecto en TensorFlow, no obstante, se descartó su uso debido a que en el momento del desarrollo del proyecto esta integración estaba disponible sólo hasta la versión 7 de TensorRT.

Esto nos deja la conversión manual, que puede realizarse mediante la herramienta *trtexec* o con la API de TensorRT. La herramienta de línea de comandos

trtexec es la manera sencilla de emplear TensorRT sin tener que desarrollar una aplicación, y cuya finalidad es la creación de motores a partir de modelos y medir el rendimiento de dichos motores. Como nosotros buscamos un control más preciso que el que podemos obtener través de sólo la línea de comandos, se descartó esta opción para la creación de los motores, no obstante, no así para la medición de rendimiento.

Una vez decidido el uso de la API en Python de TensorRT, podemos iniciar la conversión de los modelos, de lo que surge un nuevo problema, no hay una conversión directa de un modelo TensorFlow a TensorRT. Es necesaria la creación de un modelo intermedio del formato ONNX, estándar abierto para representar modelos de *Deep learning*. El flujo de trabajo puede verse en la figura 5.1.

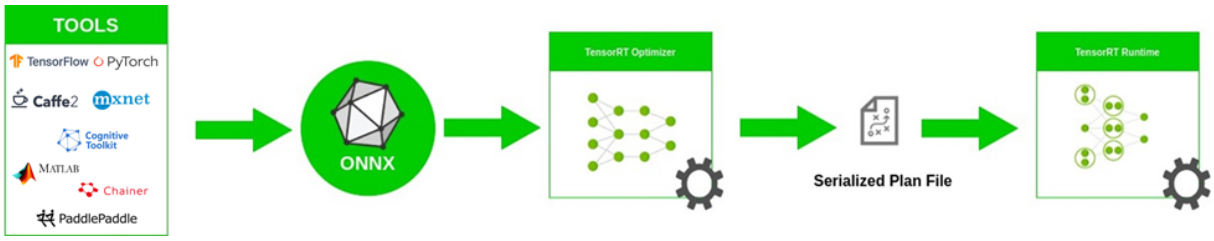


Figura 5.1: Flujo de conversión desde TensorFlow a TensorRT
Fuente: `https://developer.nvidia.com/blog`

Convertir un modelo de TensorFlow al formato ONNX no requiere de gran esfuerzo gracias a que en Python podemos encontrar el paquete *tf2onnx* destinado a este propósito. Así pues nos centraremos en la construcción del motor a partir del modelo ONNX, más concretamente en la creación del motor con precisión INT8. Ya ha sido mencionado que la cuantización del estilo PTQ a INT8 requiere de un conjunto de calibración, sin el que no puede llevarse a cabo el proceso de forma fiable. Veamos a continuación un fragmento del proceso de creación del motor, en el que omitimos los pasos previos de inicialización y carga del modelo para centrarnos en los aspectos relativos a la conversión a INT8.

```
1     if not builder.platform_has_fast_int8:
2         log.warning("INT8 is not supported natively")
3     else:
4         config.set_flag(trt.BuilderFlag.INT8)
5         config.int8_calibrator = EngineCalibrator(calib_cache)
6         if not os.path.exists(calib_cache):
7             calib_shape = [calib_batch_size] + list(inputs[0].
                shape[1:])
8             calib_dtype = trt.nptype(inputs[0].dtype)
9             config.int8_calibrator.set_image_batcher(
10                ImageBatcher(...) )
```

Listado 5.10: Creación de motor en INT8

Obsérvese que primero debe comprobarse que la GPU es compatible con la precisión indicada, tras lo que activamos el *flag* de INT8. Para la calibración, primero se comprueba si ya se ha creado un caché en alguna conversión anterior, que no es más que un archivo con la extensión .cache; en caso de no haber una caché, se configura el calibrador y se le indica un objeto de la clase *ImageBatcher*. Esta clase simplemente es inicializada con una serie de imágenes, que se preprocesan si es necesario, y devuelve un lote de imágenes del tamaño que se le especificó en su construcción; está basada en clases similares empleadas en los ejemplos de la documentación de TensorRT. A partir de aquí, sólo resta crear el motor y guardarlo en disco.

Una vez creado el motor, debe crearse un proceso similar al visto en el modelo de TensorFlow, la mayor diferencia radica en definir manualmente la función de inferencia.

```
1  def infer(batch):
2      # Prepare the output data
3      output = np.zeros(*output_spec())
4
5      # Process I/O and execute the network
6      cuda.memcpy_htod(inputs[0]["allocation"], np.
           ascontiguousarray(batch))
7      context.execute_v2(allocations)
8      cuda.memcpy_dtoh(output, outputs[0]["allocation"])
9
10     return output
```

Listado 5.11: Función de inferencia en TensorRT

Aquellos que estén familiarizados con CUDA observarán que el procedimiento les es conocido, ya que estamos alojando en la memoria de la GPU el lote de imágenes, ejecutamos la inferencia y transferimos de vuelta el resultado. Podríamos considerar que ésta es la metodología estándar a la hora de trabajar en la GPU. Con la función de inferencia creada, el código para calcular el tiempo de inferencia es el mismo al empleado con el modelo de TensorFlow, así que no es necesario repetirlo.

Si en lugar de emplear nuestra propia clase para la inferencia se opta por hacer uso de la herramienta *trtexec*, la instrucción es la siguiente:

```
1  trtexec --useCudaGraph --noDataTransfers --iterations=100 --
       avgRuns=100 --loadEngine=engine.trt
```

Listado 5.12: Uso de trtexec para la mdeición

Los parámetros que se observan se describen a continuación:

- *useCudaGraph*: utilizar CUDA Graphs para capturar la ejecución del motor y luego lanzar la inferencia.
- *noDataTransfers*: desactiva las transferencias de acceso directo a memoria desde y hacia el dispositivo.
- *iterations*: número de iteraciones a ejecutar.
- *avgRuns*: las mediciones se muestran como el promedio de las iteraciones indicadas.
- *loadEngine*: ruta al motor de TensorRT.

Terminamos la sección con una breve mención a las diferencias que hay en la construcción del modelo específico para su uso en DLA, porque es necesario introducir algunos cambios para poder hacer uso del acelerador. En concreto sólo es necesario indicar tres parámetros de configuración al constructor.

```
1 config.set_flag(trt.BuilderFlag.GPU_FALLBACK)
2 config.default_device_type = tensorrt.DeviceType.DLA
3 config.DLA_core = 0
```

Listado 5.13: Construcción de motor para DLA

En primer lugar tenemos el *flag GPU_FALLBACK* para indicarle que si una capa de la red no es compatible con el DLA, lo ejecute en la GPU en su lugar. En segundo lugar se indica que el dispositivo por defecto para la ejecución de la inferencia es el DLA, y por último hay que indicar cuáles de los DLA vamos a emplear, ya que nuestra Jetson Orin dispone de dos de estos chips. Si bien se han omitido, en el código se incluyeron comprobaciones para determinar si la máquina cuenta con estos chips antes de indicarle los parámetros.

5.3. Prototipo

En una fase temprana del desarrollo se elaboró prototipo destinado a realizar predicciones en tiempo real sobre una entrada de vídeo. Dado que nuestro modelo clasifica rostros y una entrada de vídeo contiene otros muchos elementos, es necesario indicarle al modelo dónde se encuentra el rostro en cada *frame* de la entrada.

Para manejar la entrada de vídeo y la detección de rostros se emplea la librería de código abierto para visión artificial OpenCV, puesto que dispone de forma nativa de un método para identificar rostros en una imagen. El proceso llevado a cabo por el prototipo es el siguiente:

1. En primer lugar se realiza la búsqueda de rostros sobre la entrada de vídeo.

2. Una vez identificado el rostro, se usan las coordenadas de la detección para crear una imagen centrada en el rostro.

3. Se realiza el proceso de inferencia.

4. En base al resultado de la inferencia, en la entrada de vídeo se le añade un marcador al rostro con la clase a la que pertenece.

A continuación, en la figura 5.2, se puede observar el funcionamiento del prototipo sobre dos imágenes, que pertenecen a la clase *nico* (izquierda) y *not_nico* (derecha).

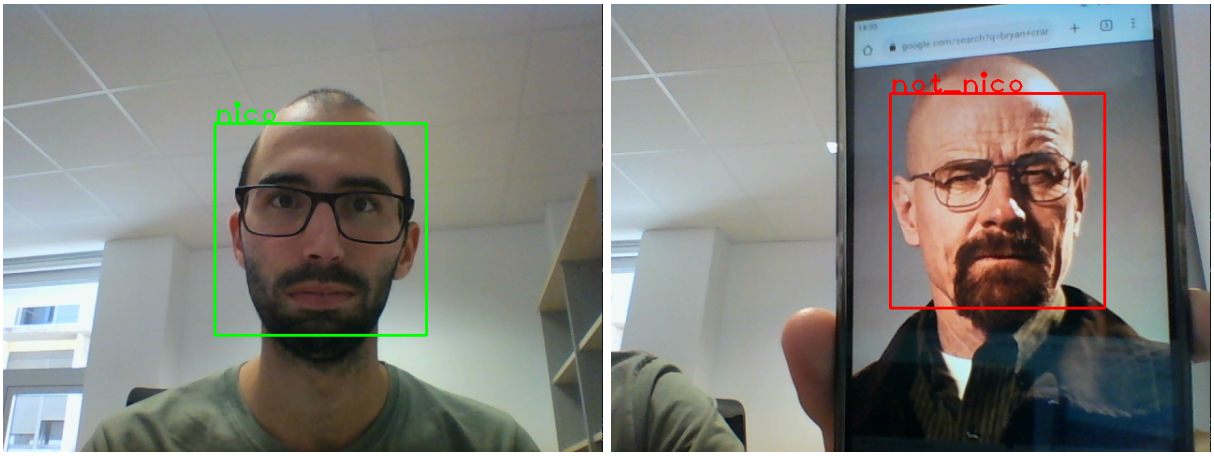


Figura 5.2: Funcionamiento del prototipo

Capítulo 6

Resultados y evaluación

Este capítulo está destinado a mostrar y evaluar los resultados recabados durante el transcurso del proyecto. Como en el capítulo anterior, los procesos de entrenamiento e inferencia se tratarán por separado, para terminar haciendo una puesta en común de las conclusiones obtenidas.

6.1. Entrenamiento

Para la fase de entrenamiento vamos a cuantificar realizar mediciones sobre la precisión, el valor de pérdida y la duración, en base a la red pre-entrenada escogida y en qué dispositivo y unidad de procesamiento se ejecuta.

6.1.1. Precisión

Al emplear el mismo código en todo momento, la precisión no se ve afectada por el dispositivo en que la ejecutemos, ya sea el DC o la Jetson, ni por su ejecución en la CPU o la GPU. No obstante, el valor de precisión obtenido no es siempre el mismo debido a que la inicialización tiene un componente aleatorio, los valores iniciales de los pesos influyen directamente en el resultado final, es por ello que los valores finales corresponden a la media de múltiples ejecuciones.

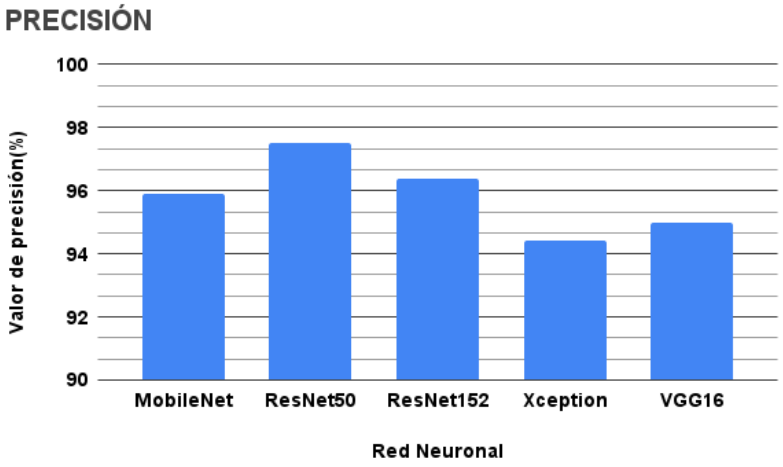


Figura 6.1: Precisión en la fase de entrenamiento

Como se puede observar en la figura 6.1, todos los modelos obtienen valores de precisión que superan el 90 por ciento, lo que deja patente el enorme potencial de la transferencia de conocimiento incluso con un *dataset* reducido. Al comparar los valores obtenidos según la red base, no se aprecian diferencias significativas.

6.1.2. Pérdida o coste

A modo de recapitulación, el valor de la función de pérdida, conocida como *loss function* en inglés, hace referencia a la diferencia entre la predicción y el valor real. Si con la precisión buscamos acercarnos al 100 por ciento, con la pérdida nuestro valor objetivo es el 0; aunque según qué función se emplee el rango de valores varía, los valores cercanos a 0 están asociados a la buena calidad del modelo.

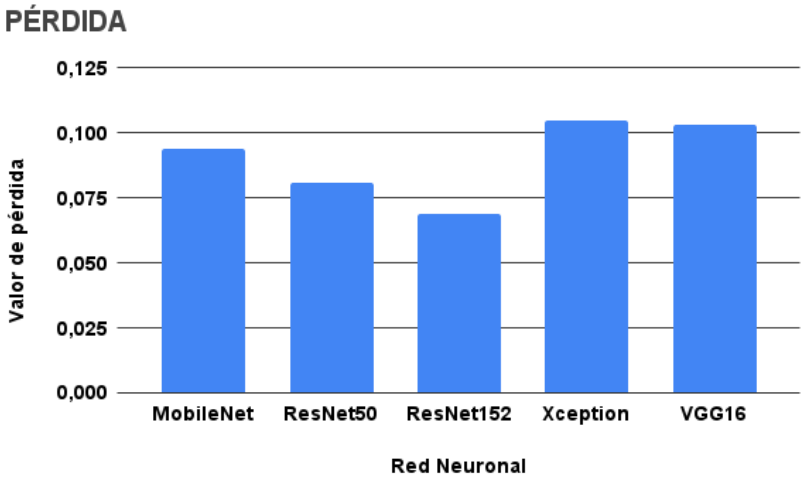


Figura 6.2: Pérdida en la fase de entrenamiento

Lo valores obtenidos son bajos, con la red ResNet152 destacando sobre el resto por un corto margen, no obstante, si tomamos como referencia para la función de entropía cruzada como la nuestra que los valores para una predicción de gran calidad es menor a 0,02 o incluso 0,05 [17], podemos concluir que los resultados son satisfactorios pero notablemente mejorables.

6.2. Tiempo

En esta sección sí que haremos diferencia entre procesador y dispositivo, a diferencia de las dos anteriores, debido a que el resultado sí está directamente influenciado según dónde y quién lo lleve a cabo. Como las capacidades de paralelización de las GPU son notablemente superiores a las de la CPU, mostraremos los tiempos obtenidos enfrentado las GPU y CPU de ambos dispositivos por la diferencia en escala.

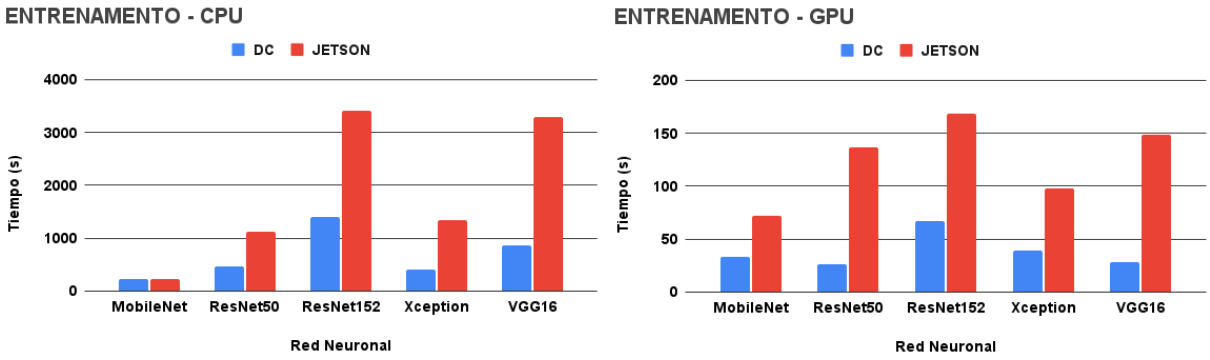


Figura 6.3: Tiempo total de entrenamiento

Como era de esperar la diferencia de tiempo entre CPU y GPU es notoria cuanto menos, con la DC, gracias a sus especificaciones superiores, realizando el entrenamiento con mayor celeridad, si bien es cierto que la diferencia se reduce considerablemente en la red MobileNet, que si bien recordamos está diseñada para ser más ligera que sus homólogas. Esto último nos plantea la duda de por qué la red MobileNet tiene un tiempo de entrenamiento superior en la GPU del DC que las redes ResNet50 y Vgg16, considerablemente más pesadas; para explicar esto tenemos que traer a colación la parada temprana que explicamos en capítulos anteriores.

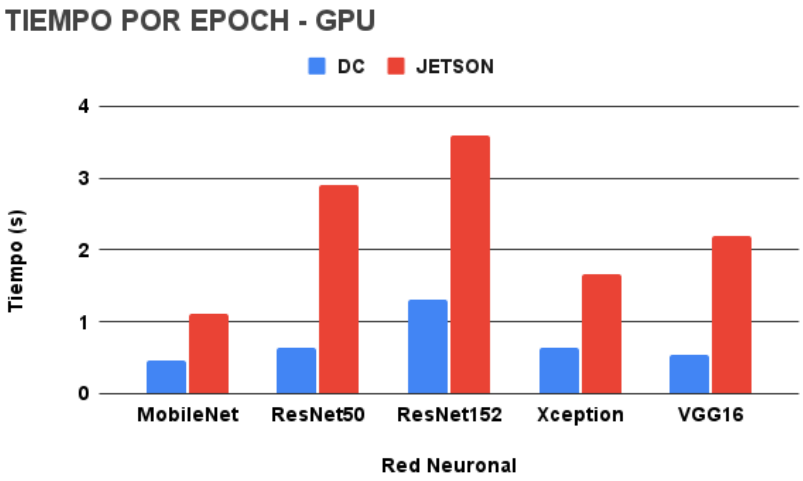


Figura 6.4: Tiempo total por epoch en la GPU

Por *epoch* entendemos una iteración dentro del proceso de entrenamiento, el que, valga la redundancia, es un proceso iterativo en el que alimentamos y ajustamos la red consecutivamente. En la figura 6.4, con los tiempos por *epoch* en la GPU para cada red neuronal, observamos que de hecho la red MobileNet sigue siendo más rápida que las ResNet50 y Vgg16, lo que nos indica que éstas últimas únicamente alcanzan la solución óptima antes, consiguiendo así un tiempo menor. Como el tiempo del *epoch* está directamente relacionado con el tamaño de nuestro *dataset*, si una iteración consiste en alimentar y ejecutar la red, más

imágenes implica más tiempo por razones obvias, podemos afirmar que aumentar la cantidad de imágenes provocaría con total seguridad que la red MobileNet conseguiría el menor tiempo de entrenamiento.

6.3. Inferencia

En la inferencia, como ya habíamos comentado anteriormente, se medirán únicamente sobre la GPU debido a que la librería de TensorRT sólo es aplicable sobre dicha unidad y como el proceso de inferencia está centrado mayoritariamente en dicha librería, la CPU pierde relevancia en favor de la GPU. Las medidas a comparar son la pérdida de precisión en el proceso de cuantización, y el rendimiento de los modelos originales y optimizados, en forma de la latencia y capacidad de procesamiento, para cada una de las precisiones disponibles. Adicionalmente se medirá el rendimiento al emplear el chip DLA para realizar la inferencia.

6.3.1. Precisión

Los resultados de precisión en el proceso de inferencia hacen referencia a cómo se ve afectada tras el proceso de optimización llevado a cabo por TensorRT. Cómo el proceso de cuantización reduce la amplitud de bits empleados por las operaciones de la red, por ejemplo el paso de 32 bits a 8, es esperable que la precisión se vea afectada negativamente.

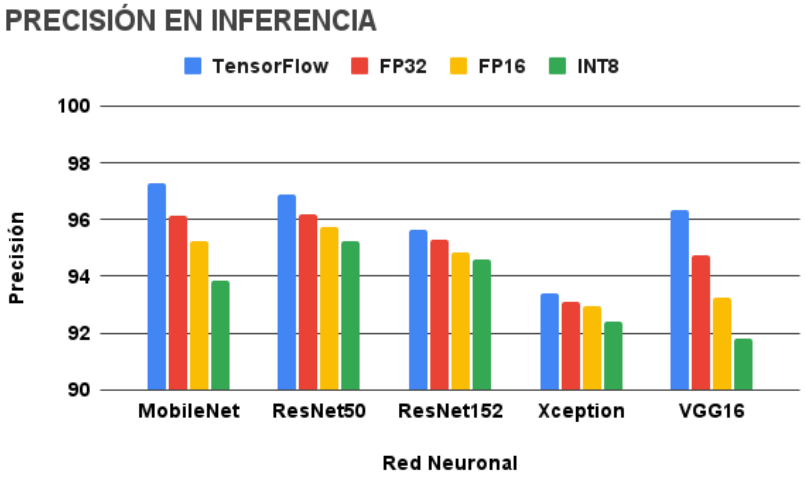


Figura 6.5: Comparativa de precisión en GPU de la Jetson tras la optimización

Tras la ejecución en la GPU del dispositivo Jetson, los datos obtenidos en la Jetson que se muestran en la figura 6.5 ponen de manifiesto una reducción de la precisión en torno al 5 por ciento en el peor de los casos. La redes MobileNet y Vgg16 se muestran más susceptibles a la pérdida de precisión, mientras que el resto presentan una disminución notablemente baja. Los resultados no han mostrado diferencias dignas de mención entre ambos dispositivos.

6.3.2. Latencia y capacidad

A diferencia de la precisión, es esta sección se diferenciarán los resultados obtenidos en el DC y la Jetson. Por un lado veremos la latencia, la suma del tiempo copia a la GPU, el tiempo de inferencia y la copia desde la GPU, y por otro la capacidad de procesamiento o *troughput*, que hace referencia a la cantidad de imágenes que capaz de procesar la red en un intervalo de tiempo determinado.

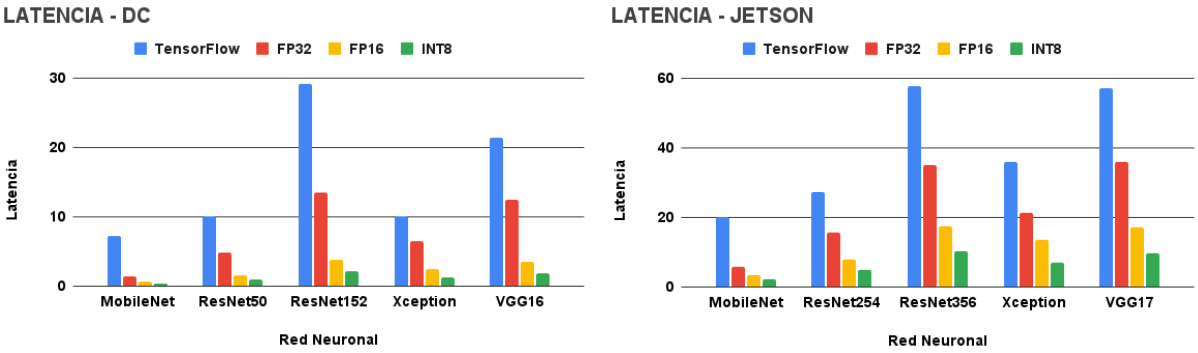


Figura 6.6: Latencia en inferencia

En la figura 6.6 se puede observar los valores de latencia para cada red, modelo y dispositivo. Podemos advertir que los resultados mantienen la tendencia vista en los tiempos de entrenamiento y por *epoch*, con MobileNet nuevamente a la cabeza, y con una reducción de en torno al 50 por ciento en el DC en relación a la Jetson. Al comparar los motores de TensorRT en diferentes precisiones, tamaño en bits de las operaciones, se pone de manifiesto una reducción notoria de la latencia incluso con el punto flotante de 32 bits, que es la precisión original del modelo; las operaciones en INT8 son capaces de alcanzar una reducción de en torno al 90 por ciento respecto al modelo original.

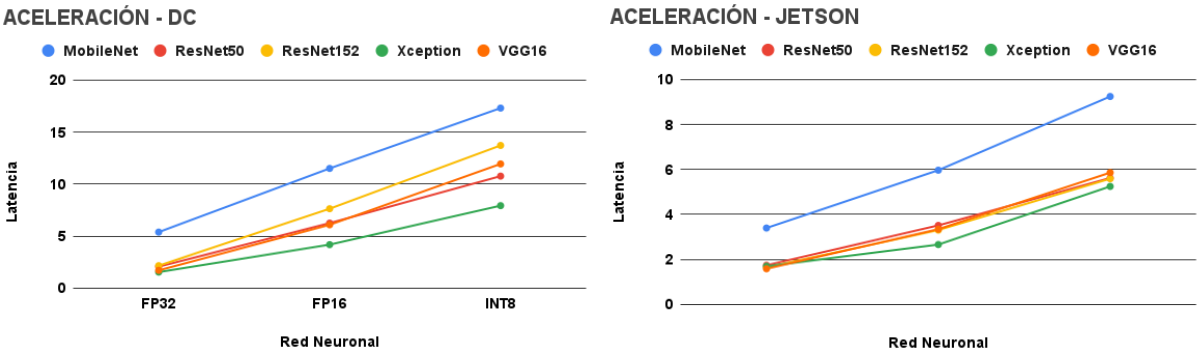


Figura 6.7: Aceleración en inferencia

La figura 6.7 muestra los valores de aceleración obtenidos a partir los datos de latencia, dejando patente que los valores obtenidos en la DC son aproximadamente el doble a los obtenidos con la Jetson. La aceleración mejora de forma lineal a medida que la anchura en bits de las operaciones disminuye, donde podemos destacar que en la Jetson la mejora del INT8 respecto al FP16 es mayor que en

el DC; además de que nuevamente la red MobileNet consigue los mejores valores en contraposición con la red Xception, que en comparación con el resto tiene una peor optimización con TensorRT. La redes restantes consiguen resultados similares entre sí, tanto en el DC como en la Jetson.

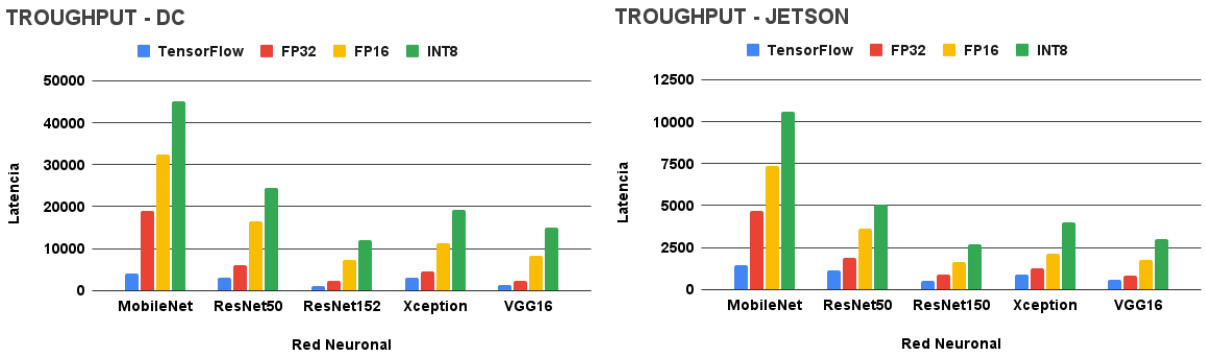


Figura 6.8: Capacidad de procesamiento en inferencia

Dejando a un lado latencia, en la figura 6.8 vemos los resultados de la capacidad de procesamiento o *troughput*, que nos muestran un resultado inverso al visto anteriormente, es decir, con un descenso en la precisión de las operaciones estamos consiguiendo un mayor número de predicciones por segundo. Sin duda alguna, la red MobileNet es la que más se ve beneficiada por el proceso de optimización, seguida por le red ResNet50, y con el resto consiguiendo valores similares. Si en la latencia se observaba una diferencia en la reducción de la latencia de entorno al 50 por ciento entre ambos dispositivos, la diferencia del aumento de rendimiento muestra una mejora de entorno al 450 por ciento en favor del DC. Esta diferencia entre latencia y capacidad de procesamiento podemos asociarla a que el proceso de inferencia se realiza sobre un lote de imágenes en lugar de una única imagen.

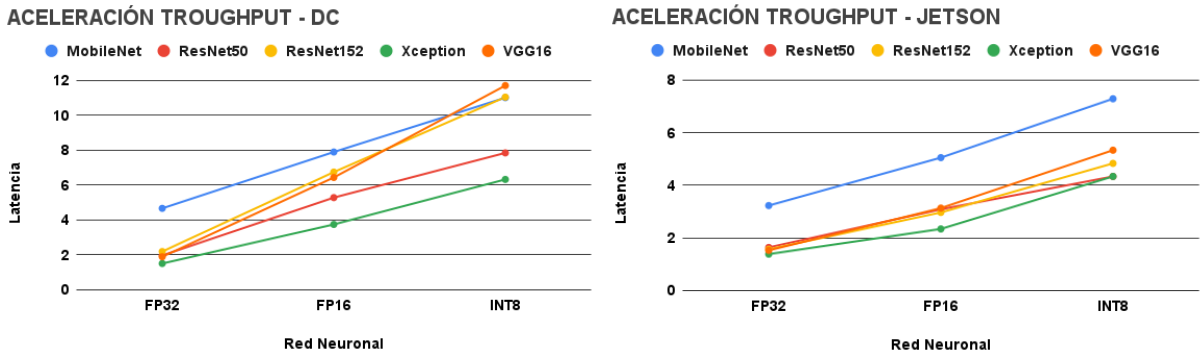


Figura 6.9: Aumento en capacidad de procesamiento

En la figura 6.9 se muestra gráficamente el aumento en la capacidad de procesamiento obtenido con el proceso de optimización con TensorRT. Si bien en la Jetson las aceleraciones son similares a las obtenidas para con la latencia, en el DC, con una GPU con capacidad de computación superior, las aceleraciones de ResNet50 y VGG16 consiguen una aceleración superior al de la red MobileNet en

la precisión de INT8, la manera en que aumenta el rendimiento a medida que reduce la precisión es notablemente superior en estas dos redes en comparación con el resto. Otro punto a destacar es el cómo la diferencia de aceleración entre ambos dispositivos ya no es cercana al doble, como en la latencia, sino que ronda el 30 por ciento; por último, la red Xception sigue mostrando los peores resultados por un escaso margen.

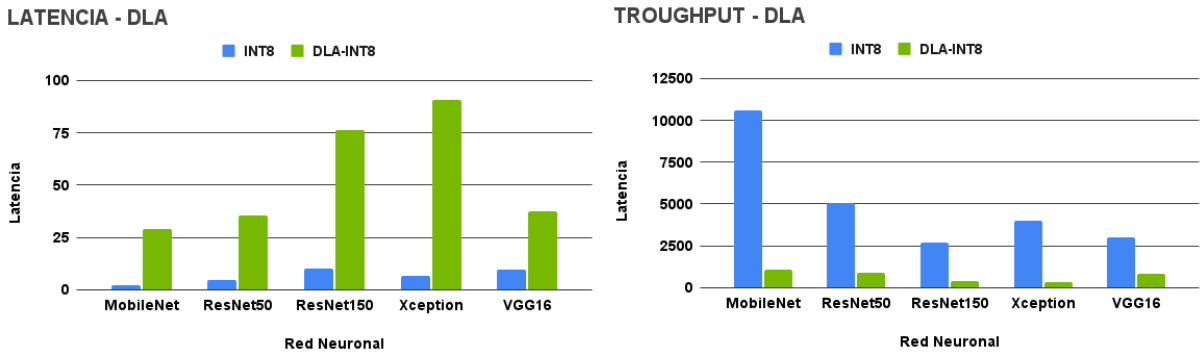


Figura 6.10: Comparativa de resultados con DLA

Por último tenemos la figura 6.10 con los datos comparativos entre los resultados obtenidos por los motores con INT8 usando y sin usar el DLA, ambos en la Jetson. Los resultados muestran que el acelerador no hace honor a su nombre y reduce el rendimiento los modelos optimizados con TensorRT de forma significativa. La causa de este pobre rendimiento es la incompatibilidad de gran parte de las capas de las redes pre-entrenadas, a la hora de ejecutar el modelo en el chip de aceleración necesita enviar los datos de vuelta a la GPU de forma constante, lo que repercute negativamente en el rendimiento de forma ostensible.

Capítulo 7

Conclusiones y líneas futuras

La Inteligencia Artificial es una tecnología que está en boga gracias al incremento de la capacidad computacional de los dispositivos modernos y la ingente cantidad de información disponible. En particular, el campo del *Deep Learning*, que abarca los modelos de IA basados en redes neuronales artificiales, han acaparado la atención de investigadores y desarrolladores de forma mayoritaria, en detrimento de los modelos tradicionales, debido tanto a las posibilidades que ofrecen como su mejor rendimiento a la hora de emplear grandes cantidades de datos.

Por estos motivos el *Deep Learning* está siendo utilizado en un múltiples disciplinas, no obstante, desarrollar una red neuronal desde cero que cumpla unos estándares de rendimiento y eficiencia, es una tarea exigente que no está al alcance de cualquiera. Si hablamos de campos como el IoT, donde la velocidad y el consumo son factores determinantes, esta dificultad se convierte en un obstáculo imperante.

Dado este inconveniente, en este proyecto hemos desarrollado un modelo de *Deep Learning* mediante el uso de la transferencia de conocimiento, que nos permite adaptar modelos existente de alta complejidad y rendimiento contrastado, para su ejecución en un equipo de escritorio y otro equipo específico para la Computación en el Borde (NVIDIA Jetson), que muchos dispositivos IoT ya permiten realizar. A pesar de entrar en la categoría de IoT, las capacidades del dispositivo Jetson son considerablemente altas a los que podemos encontrar en sistemas más tradicionales, sin embargo, éstas siguen siendo inferiores al computador estándar.

Empleando la capacidad de transferencia de conocimiento y un *dataset* de tamaño reducido, los resultados obtenidos muestran los extraordinarios beneficios de dicha técnica, consiguiendo altos niveles de precisión en los cinco modelos desarrollados. Con valores que rondan el 95 por ciento, los modelos tienen una precisión sustancialmente alta con un valor de pérdida aceptable aunque mejorable.

Los tiempos de entrenamiento nos muestran los datos esperados, un mayor rendimiento de la GPU frente a la CPU, y del DC frente a la Jetson, donde es destacable que la red más ligera de todas la MobileNet, se ve perjudicada en la GPU

de mayor potencia por su mayor cantidad de iteraciones en calcular los mejores parámetros, lo que nos indica que en conjunto de datos pequeño y una GPU de gran rendimiento, esta red está desaprovechada. Por otra parte, si bien la Jetson consigue peores resultados, queda de manifiesto que es perfectamente asumible entrenar el modelo en el dispositivo, aspecto de gran valor en un dispositivo en el borde.

En lo que respecta al proceso de inferencia, emplear la librería TensorRT para optimizar los modelos reduciendo la precisión de las operaciones entraña un impacto reducido en el índice de acierto de los modelos, con las redes MobileNet y `Vgg16` ligeramente más susceptibles que el resto al empeoramiento. Estos resultados muestran una alta eficiencia por parte del proceso de optimización, a la vez que ponen en valor las bondades de la transferencia de conocimiento una vez más, gracias a la robustez de los modelos base, que mantienen un alto índice de aciertos incluso con operaciones de precisión reducida.

Si el impacto sobre la fiabilidad es escaso, nos encontramos justo lo contrario con el rendimiento. La latencia y la capacidad de procesamiento o *troughput* disminuyen y aumentan de forma considerable respectivamente en ambos dispositivos, aunque con un orden mayor en el DC. Nuevamente la red MobileNet consigue los mejores valores, todas las redes obtienen una mejora de rendimiento notoria, es especial con el nivel de precisión INT8, con la red Xception, por norma general, mostrando el peor grado de optimización.

Todos estos resultados muestran que el dispositivo Jetson, aunque inferior al DC, alcanza unos niveles de rendimiento, como mínimo, aceptables que deberían permitirle desempeñar su función con fiabilidad en entornos de producción. No obstante, una de las mayores ventajas de este dispositivo es el grupo de aceleradores que tiene en su haber, de ahí que podamos considerarle especializado, y es precisamente en este aspecto donde la transferencia de conocimiento muestra su mayor debilidad.

Los modelos pre-entrenados se componen en su mayoría de capas incompatibles con el chip de aceleración de *Deep Learning* del dispositivo Jetson, lo que provoca que el aceleración consiga un efecto contrario al esperado y reduzca el rendimiento de forma considerable del modelo. Aprovechar al máximo la Jetson implica el desarrollo de un modelo teniendo en cuenta las limitaciones del acelerador, algo incompatible con la transferencia de conocimiento.

7.1. Líneas futuras

Una posible línea de desarrollo es incrementar la complejidad del modelo añadiendo más clases a su clasificación. El nuestro es un modelo binario que sólo reconoce un rostro frente a los demás, en otras palabras, sólo ha aprendido las características de un rostro. Añadir más clases implicaría que nuestro modelo se vería

obligado a realizar un proceso de extracción de características más completo, tras lo que sería de interés medir cómo se ven afectados los valores de rendimiento.

Sin duda, otra aspecto de interés que ha quedado sin tratar en este proyecto es la medición del consumo energético, tema de suma relevancia en el IoT. Se ha mostrado que el DC obtiene un rendimiento superior, sin embargo, su consumo energético es considerablemente superior, sólo los 320W de la GPU es seis veces mayor al consumo máximo de 50W de la Jetson.

En relación con el consumo de energía, está la medición según el modo de potencia de la Jetson, que tiene dos modos adicionales de 30 y 15 vatios además del de 50W que hemos utilizado en el desarrollo de este proyecto. Medir si el rendimiento por vatio se mantiene es especialmente trascendente en entornos donde se prima reducir el consumo eléctrico, como es común en el IoT.

Por último, si el tamaño del *dataset* aumenta conforme lo hace la escala del proyecto, como por ejemplo con la adición de nuevas clases a la clasificación, es de esperar que el tiempo de entrenamiento lo haga en consecuencia. Con esto en mente, podemos sugerir explorar diferentes formas de entrenamiento distribuido, ya sea en múltiples GPU o dispositivos, o incluso el uso de otras unidades como las TPUs. Una mayor paralelización del entrenamiento puede reducir considerablemente el tiempo total, pero sería necesario medir en qué grado se ven afectados el rendimiento y el consumo energético.

Capítulo 8

Summary and Conclusions

As explained in the introduction, Artificial Intelligence is a technology currently in vogue thanks to the increased computational power of modern devices and the vast amount of information available today. Specifically, the field of Deep Learning, which encompasses AI models based on artificial neural networks, has captured the attention of researchers and developers to a greater extent than traditional models due to the possibilities they offer and their better performance when using large amounts of data.

Deep Learning has carved out a niche in multiple disciplines for these reasons. However, developing a neural network from scratch that meets performance and efficiency standards is a demanding task that is not within everyone's reach. If we talk about fields like IoT, where speed and consumption are determining factors, this difficulty becomes a prevailing obstacle.

With this issue in mind, in this project, we have developed a Deep Learning model using knowledge transfer, which allows us to adapt existing models of high complexity and proven performance for execution on a desktop computer and another specific for IoT at the edge, such as the NVIDIA Jetson. Although it falls into the IoT category, the Jetson device's capabilities are considerably high compared to those found in more traditional systems. However, they are still inferior to those of a standard computer.

Using the knowledge transfer capacity and a small-sized dataset, the results clearly demonstrate this technique's extraordinary benefits, achieving high levels of precision in any of the networks. With values around 95 percent, the models have substantially high accuracy, albeit with the drawback of having an improvable but acceptable loss value.

The training times show us the expected data, greater GPU performance over CPU, and DC over Jetson, where it is noteworthy that the lightest network of all, MobileNet, is hindered in the more powerful GPU due to its greater number of iterations in calculating the best parameters, indicating that with small datasets and high-performance GPUs, this network is underutilized. On the other hand,

while the Jetson achieves worse results, it is perfectly feasible to train the model on the device, which is of great value in an edge device.

Regarding the inference process, using the TensorRT library to optimize the models by reducing the precision of the operations has a reduced impact on the accuracy index of the models, with MobileNet and Vgg16 networks being slightly more susceptible to worsening than the others. These results demonstrate high efficiency in the optimization process while highlighting the benefits of knowledge transfer, thanks to the robustness of the base models, which maintain a high accuracy index even with reduced precision operations.

If the impact on reliability is low, we find just the opposite with performance. Latency and throughput decrease and increase considerably on both devices, although to a greater extent on the DC. Once again, MobileNet achieves the best values, with all networks obtaining a notable performance improvement, especially with the INT8 precision level. The Xception network generally shows the worst degree of optimization.

All these results show that the Jetson device, although inferior to the DC, achieves at least acceptable performance levels that should allow it to function smoothly in production environments. However, one of the most significant advantages of this device is the various accelerators it has at its disposal, which is why we can consider it specialized. It is precisely in this aspect where knowledge transfer shows its greatest weakness.

Pre-trained models are mainly composed of layers incompatible with the Deep Learning acceleration chip of the Jetson device, which causes the acceleration to have the opposite effect and significantly slow down the model. Maximizing the use of Jetson involves developing a model considering the accelerator's limitations, something incompatible with knowledge transfer.

8.1. Future Work

An approximation for future development is the increase of the complexity of the models by adding more classes to its classification. Ours are binary models that only recognise one specific face. The inclusion of more classes would mean that our model would be forced to perform a more thorough feature extraction process, after which it would be of interest to measure how performance values are affected.

Undoubtedly, another aspect of interest that has been left unaddressed in this project is the measurement of power consumption, a topic of utmost relevance in IoT. It has been shown that the DC achieves superior performance, however, its power consumption is considerably higher, the 320W of the GPU alone is six times higher than the 50W of maximum consumption of the Jetson device.

In relation to power consumption, there is the measurement according to the power mode of the Jetson, which has two additional modes of 30 and 15 watts in addition to the 50W mode that we have used in the development of this project. Measuring whether performance per watt is maintained is especially important in environments where power consumption needs to be reduced, as is common in the IoT.

Finally, if the size of the *dataset* increases as the scale of the project does, such as adding new classes to the classification, we can expect the training time to do so accordingly. With this in mind, we can suggest exploring different forms of distributed training, either across multiple GPUs or devices, or even the use of other units such as TPUs. Further parallelization of training can significantly reduce the total time, but it would be necessary to measure to what extent performance and energy consumption are affected.

Capítulo 9

Presupuesto

Para el desarrollo de este trabajo se han empleado herramientas software gratuitas que no repercuten en el presupuesto del mismo, en cambio, se ha hecho uso y mención de diversos elementos hardware sin los que no hubiera sido posible su realización, por lo que procedemos a enumerar los elementos hardware utilizados en el proyecto junto a su coste.

Dispositivo	**Coste**
Desktop Computer	2.520,24€
NVIDIA Jetson AGX Orin	1.877,06€
Monitor, teclado y ratón	139,98€
Total	4.537,28 €

Tabla 9.1: Resumen de dispositivos

Para el coste de los recursos humanos tomaremos como referencia las 300 horas de trabajo que se estipulan en la guía docente de la asignatura de Trabajo de Fin de Máster. Como sueldo medio de un desarrollador de software en España usaremos el total de 16,92€ por hora.

Tarea	**Horas invertidas**	**Coste**
Búsqueda bibliográfica	15	253,80€
Desarrollo del *dataset*	40	676,80€
Desarrollo de los modelos	120	2.030,40€
Evaluación	80	1.353,60€
Elaboración de la memoria	45	761,40€
Total	300	5.076,00€

Tabla 9.2: Horas invertidas y coste

Bibliografía

[1] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

[2] François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016.

[3] Ziaul Choudhury, Shashwat Shrivastava, Lavanya Ramapantulu, and Suresh Purini. An fpga overlay for cnn inference with fine-grained flexible parallelism. *ACM Trans. Archit. Code Optim.*, 19(3), may 2022.

[4] Manuel F. Dolz, Hector Martinez, Adrián Castelló, Pedro Alonso, and Enrique S Quintana-Orti. Efficient and portable winograd convolutions for multi-core processors. *The Journal of Supercomputing*, pages 1–22, 02 2023.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[6] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[7] Tahir Hussain, Dostdar Hussain, Israr Hussain, Hussain lSalman, Saddam Hussain, Syed Sajid Ullah, and Suheer Al-Hadhrami. Internet of things with deep learning-based face recognition approach for authentication in control medical systems. *Computational and Mathematical Methods in Medicine*, 2022, Feb 2022.

[8] Vandit Jain. Everything you need to know about "activation functions" in deep learning models. `https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-8` Accessed: 2022-02-25.

[9] Shawn Johnson. Nvidia predicts ai model to be a million times more powerful than chatgpt within 10 years. `https://biz.crast.net/nvidia-predicts-ai-model-to-be-a-million-times-more-powerful-than-chatgpt-within` Accessed: 2023-02-25.

[10] Jeremy Jordan. Common architectures in convolutional neural networks. https://www.jeremyjordan.me/convnet-architectures/. Accessed: 2022-02-14.

[11] M. Kaiser, R. Griessl, N. Kucza, C. Haumann, L. Tigges, K. Mika, J. Hagemeyer, F. Porrmann, U. Rückert, M. vor dem Berge, S. Krupop, M. Porrmann, M. Tassemeier, P. Trancoso, F. Qararyah, S. Zouzoula, A. Casimiro, A. Bessani, J. Cecilio, S. Andersson, O. Brunnegard, O. Eriksson, R. Weiss, F. Mcierhöfer, H. Salomonsson, E. Malekzadeh, D. Ödman, A. Khurshid, P. Felber, M. Pasin, V. Schiavoni, J. Ménétrey, K. Gugala, P. Zierhoffer, E. Knauss, and H. Heyn. Vedliot: Very efficient deep learning in iot. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 963–968, 2022.

[12] Tero Karras and Janne Hellsten. Flickr-faces-hq dataset. https://github.com/NVlabs/ffhq-dataset. Accessed: 2022-02-20.

[13] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, abs/1812.04948, 2018.

[14] Shahid Latif, Maha Driss, Wadii Boulila, Zil e Huma, Sajjad Shaukat Jamal, Zeba Idrees, and Jawad Ahmad. Deep learning for the industrial internet of things (iiot): A comprehensive survey of techniques, implementation frameworks, potential applications, and future directions. *Sensors*, 21(22), 2021.

[15] Andre Luckow, Matthew Cook, Nathan Ashcraft, Edwin Weill, Emil Djerekarov, and Bennie Vorster. Deep learning in the automotive industry: Applications and tools. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3759–3768, 12 2016.

[16] Sambit Mahapatra. Why deep learning over traditional machine learning? https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063. Accessed: 2023-02-22.

[17] Christophe Pere. What are loss functions? https://towardsdatascience.com/what-is-loss-function-1e2605aeb904. Accessed: 2022-02-27.

[18] Carlo Perrotta and Neil Selwyn. Deep learning goes to school: toward a relational understanding of ai in education. *Learning, Media and Technology*, 45(3):251–269, 2020.

[19] Lutz Prechelt. *Early Stopping - But When?*, pages 55–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[20] Kevin Purdy. Appliance makers sad that 50 % of customers won't connect smart appliances. https://arstechnica.com/gadgets/2023/01/half-of-smart-appliances-remain-disconnected-from-internet-makers-lament/. Accessed: 2023-02-20.

[21] Syafeeza Ahmad Radzi, MK Mohd Fitri Alif, Y Nursyifaa Athirah, AS Jaafar, AH Norihan, and MS Saleha. Iot based facial recognition door access control home security system using raspberry pi. *International Journal of Power Electronics and Drive Systems*, 11(1):417, 2020.

[22] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey and benchmarking of machine learning accelerators. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2019.

[23] Sabyasachi Sahoo. Residual blocks — building blocks of resnet. `https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec`. Accessed: 2022-03-08.

[24] Terrence J. Sejnowski. *The Deep Learning Revolution*. The MIT Press, 10 2018.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, 2015.

[26] Princeton University Stanford Vision Lab, Stanford University. Imagenet. `https://www.image-net.org`. Accessed: 2022-02-24.

[27] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[28] Lyudmila Tuzova, Dmitry Tuzoff, Sergey Nikolenko, and Alexey Krasnov. *Teeth and Landmarks Detection and Classification Based on Deep Neural Networks*, pages 129–150. IGI Global, 01 2019.

[29] P. Warden and D. Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*, chapter 10. O'Reilly, 2020.

[30] Chunxue Wu, Chong Luo, Naixue Xiong, Wei Zhang, and Tai-Hoon Kim. A greedy deep learning method for medical disease analysis. *IEEE Access*, 6:20021–20030, 2018.

[31] Juan Manuel Manchado Ortega y Jorge Antonio García Pérez. Fpga: qué es y cuáles son las características de este componente. `https://www.akka-technologies.com/fpga/`. Accessed: 2022-03-07.

[32] Hehua Yan, Jiafu Wan, Chunhua Zhang, Shenglong Tang, Qingsong Hua, and Zhongren Wang. Industrial big data analytics for prediction of remaining useful life based on deep learning. *IEEE Access*, 6:17190–17197, 2018.

[33] Zhiliang Zhang, Limei Zhao, and Tao Yang. Research on the application of artificial intelligence in image recognition technology. *Journal of Physics: Conference Series*, 1992(3):032118, aug 2021.