UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Cómputo eficiente de Redes Neuronales sobre Hardware Heterogéneo

Raúl Ignacio Marichal

Maestría en Informática
Facultad de Ingeniería, Programa de Desarrollo de las Ciencias Básicas
Universidad de la República

Montevideo – Uruguay
de 2024

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Cómputo eficiente de Redes Neuronales sobre Hardware Heterogéneo

Raúl Ignacio Marichal

Tesis de Maestría presentada al Programa de Posgrado en Informática, Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Magíster en Informática.

Directores:

Dr. Ing. Ernesto Dufrechou

Dr. Ing. Pablo Ezzatti

Montevideo – Uruguay

de 2024

Marichal, Raúl Ignacio

Cómputo eficiente de Redes Neuronales sobre Hardware Heterogéneo / Raúl Ignacio Marichal. - Montevideo: Universidad de la República, Facultad de Ingeniería, Programa de Desarrollo de las Ciencias Básicas, 2024.

XIV, 122 p. 29, 7cm.

Directores:

Ernesto Dufrechou

Pablo Ezzatti

Tesis de Maestría – Universidad de la República, Programa en Informática, 2024.

Referencias bibliográficas: p. 105 – 122.

1. redes neuronales, 2. cómputo eficiente, 3. plataformas de hardware heterogéneas. I. Dufrechou, Ernesto, Ezzatti, Pablo, . II. Universidad de la República, Maestría en Informática en Informática. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Dra. Adriana Marotta

Dr. Ing. Guillermo Moncecchi

Dr. Manuel F. Dolz

Montevideo – Uruguay
de 2024

RESUMEN

Las redes neuronales han revolucionado la resolución de problemas en diversos campos, como medicina, robótica e ingeniería. Estas técnicas utilizan grandes volúmenes de datos para aprender (etapa de entrenamiento) y deducir resultados (etapa de inferencia) ayudando así a tomar decisiones. Sin embargo, tanto la etapa de entrenamiento como la de inferencia requieren considerables recursos computacionales y, por lo tanto, un elevado consumo energético. Esta tesis aborda la creciente necesidad de optimizar estos procesos desde el punto de vista de los recursos computacionales que requieren, explorando dos enfoques principales: el uso de hardware especializado y la optimización a nivel de software para maximizar el aprovechamiento de las plataformas de hardware subyacentes.

En la primera línea de trabajo, se evalúa una variedad heterogénea de plataformas de hardware, tanto diseñadas específicamente para el cómputo de redes neuronales como plataformas de propósito general. Se logró establecer un entorno de desarrollo e implementación de redes neuronales eficiente, capaz de aprovechar plataformas de hardware de diversas características.

Por otro lado, se estudian y aplican diferentes optimizaciones por software utilizando herramientas que son el estado del arte, buscando explotar características específicas de cada plataforma de hardware. Esta línea mostró las potenciales mejoras en los tiempos de ejecución que ofrecen estas herramientas, alcanzando mejoras de hasta 10×.

Además, como evolución de las dos líneas de trabajo antes descritas, se abordó la optimización de una herramienta de simplificación de modelos, NetAdapt. Se estudió en profundidad la herramienta y se propusieron diferentes optimizaciones sobre la herramienta, los resultados experimentales muestran reducciones en los tiempos de ejecución de hasta 37× para la generación de modelos optimizados para plataformas de hardware específicas.

Por último, es necesario destacar que en un contexto de creciente importancia de la inteligencia artificial, donde se posicionan los requerimientos compu-

tacionales como una de las principales limitantes, tanto los esfuerzos estudiados como los incipientes aportes de la presente tesis muestran los niveles significativos de optimización que se pueden alcanzar en la implementación de redes neuronales a través del software como del hardware.

Palabras claves:
redes neuronales, cómputo eficiente, plataformas de hardware heterogéneas.

Tabla de contenidos

Capítulo 1

Introducción

Existe un gran número de sistemas o máquinas que tienen la capacidad de resolver problemas de una forma "similar" a como lo hacemos los humanos, alcanzando niveles de precisión o calidad en los resultados incluso mejores. A este campo se le conoce como inteligencia artificial (IA o AI por Artificial Intelligence), término acuñado por John McCarthy en los años 50 [83]. Las redes neuronales artificiales [76], o simplemente llamadas redes neuronales (NN por Neural Networks) integran este grupo con una importancia creciente principalmente debido a la elevada cantidad de aplicaciones en las que han mostrado ser efectivas al día de hoy. En particular, las Redes Neuronales Profundas [16] (DNNs por Deep Neural Networks) una sub-categoría de NNs, son empleadas para, por ejemplo, reconocimiento de imágenes [55], reconocimiento de voz [17], juegos [108], robótica y manejo automático de vehículos [14, 33] e incluso en campos como la biología y medicina [64].

Si bien las DNNs obtienen resultados destacados en muchas aplicaciones, su uso suele implicar un gran costo computacional (generando, por lo tanto, altos costos energéticos y tiempos de respuesta elevados -latencia-). La importancia práctica de este campo y su continuo crecimiento, ha motivado el desarrollo de múltiples plataformas de hardware o arquitecturas especializadas en la aceleración de NNs [37, 62], así como herramientas que posibilitan la evaluación y optimización de modelos de NNs en distintos tipos de plataformas [9, 60, 119].

En este contexto, interesa estudiar y evaluar diferentes técnicas que permitan realizar un uso eficiente de las plataformas de hardware subyacentes a la ejecución de dichas redes neuronales. Particularmente, la mayoría de las técnicas relevadas están concentradas en la etapa de inferencia, o dicho de otro modo

la ejecución de la red neuronal. Esto se debe, por un lado al hecho mencionado por Jensen Huang (CEO de NVIDIA) en 2019, de que el 80-90 % del consumo energético en tareas de Machine Learning (ML), es generado por la etapa de inferencia [94]. Por otro lado, muchas veces la implementación de soluciones utilizando NNs tiene asociada restricciones como pueden ser: la conectividad, privacidad, latencia y, en el caso de dispositivos edge y/o autónomos, restricciones de potencia energética, siendo necesario implementar soluciones que permitan obtener rendimientos con mejores relaciones de procesamiento por energía consumida y manteniendo los niveles aceptables de precisión.
En este contexto, el objetivo principal de esta tesis es avanzar en el estudio y comprensión de estrategias, herramientas y plataformas de hardware, asociadas a la optimización, aceleración y cómputo eficiente de redes neuronales. Específicamente, se pretende alcanzar una comprensión cabal de las posibilidades que ofrecen las diferentes plataformas de hardware disponibles, relevar las diferentes herramientas para optimizar la aplicación de redes neuronales y evaluar posibles propuestas para extenderlas sobre plataformas concretas.

El resto del documento se estructura de la forma que se describe a continuación. El Capítulo 2 presenta, de forma acotada, los conceptos y la teoría en la que está basada el resto de la tesis. Entre otros temas, se aportan detalles de las redes neuronales, sus bloques de construcción, etapas y operaciones involucradas en el cómputo de estas rutinas, así como las principales arquitecturas de hardware que han permitido avanzar con la implementación de las NNs. En el Capítulo 3 se resumen los principales antecedentes y esfuerzos relacionados con la optimización de redes neuronales, abordados desde dos enfoques: desde el software, buscando simplificar los modelos y así obtener cómputos más eficientes, y desde el hardware específicamente desarrollado para el cómputo eficiente de las NNs. Posteriormente, en los próximos dos capítulos, se presentan los esfuerzos realizados en el contexto de la maestría. Las propuestas incluyen, en primer lugar (Capítulo 4), una evaluación experimental de varias plataformas de hardware con diferentes mecanismos de optimización de redes neuronales. Luego, en el Capítulo 5, se propone y evalúa la optimización de uno de estos mecanismos, obteniendo importantes reducciones en el tiempo de cómputo para simplificar redes neuronales de gran porte. El documento se cierra con un resumen de las conclusiones que surgen del estudio de maestría, un listado de las publicaciones alcanzadas y la identificación de líneas de trabajo para extender el presente esfuerzo en el Capítulo 6.

Capítulo 2

Fundamentos teóricos

Este capítulo incluye un resumen de los enfoques, teorías y conceptos en los cuales se fundamenta el trabajo de la maestría. Se basa, principalmente, en la exposición de otros esfuerzos sobre los temas abordados en este estudio, buscando cierto nivel de auto-contención en el documento. Específicamente, en este capítulo se incluyen conceptos introductorios sobre redes neuronales, así como las bases desde el punto de vista de hardware para el cómputo de dicha técnica.

2.1. Conceptos básicos

A continuación se desarrollan conceptos básicos que permitirán entender la evolución de las redes neuronales profundas (DNNs, del inglés Deep Neural Networks [44]). Antes de ahondar en las DNNs, es necesario entender la posición de éstas dentro del campo de la Inteligencia Artificial (AI, del inglés Artificial Inteligence). En primer lugar, para referirse a inteligencia artificial, hay que remontarse al año 1956, cuando John McCarthy [83] define este concepto como: "*The science and engineering of creating intelligent machines*".

Posteriormente, dentro del área de IA, se hallan especializaciones o subcategorías, una de ellas, y quizás una de las más importantes, es el Machine Learning (ML) o a veces, en español, Aprendizaje Automático; definida por Arthur Samuel en 1959 como: "*The field of study that gives computers the ability to learn without being explicitly programmed*" [105], concepto que se diferencia bastante a la noción común que se tiene de un programa como un conjunto de heurísticas que definen explícita y estáticamente el comportamiento de un

algoritmo.

Las ventajas de los algoritmos de ML sobre los programas estáticos diseñados por los ingenieros o programadores, para ciertas tareas, son varias. Los algoritmos de ML son capaces de explotar al máximo grandes conjuntos de datos a través del aprendizaje estadístico, obteniendo representaciones eficaces de los mismos que permiten resolver en "poco tiempo" problemas complejos que necesitarían de mucha interpretación y experiencia si fuesen resueltos por expertos [114]. Este tipo de algoritmos en los últimos años ha logrado obtener notables resultados. Un claro ejemplo es el de ChatGPT [27] propuesto por OpenAI, a finales del año 2022, un chat interactivo con la capacidad de resolver problemas sumamente variados y complejos.

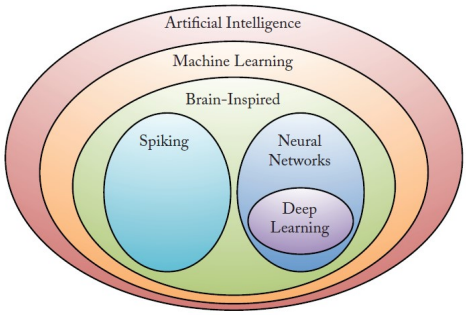


Figura 2.1: Aprendizaje profundo dentro de la Inteligencia Artificial. Extraído de [114].

Dentro de este esquema, en dirección hacia las DNNs, están los algoritmos Bioinspirados (o en inglés Bio-Inspired). Estos algoritmos toman como base el funcionamiento de contrapartes biológicas como puede ser el cerebro. En particular, para el caso del cerebro, no se intenta crear uno, sino que se apunta a, de alguna manera, emularlo (al menos de la forma que se cree que éste funciona). Considerando esta definición, se profundizará en la descripción del cerebro. La unidad mínima de cómputo del cerebro humano es la *neurona*. El cerebro posee, en promedio, alrededor de 86 billones de neuronas, que interconectadas entre sí, a través de las llamadas dendritas (entradas) y axones (salidas), dan como resultado una de las fuentes o máquinas de cómputo biológicas más sofisticadas que se conocen. A través de las dendritas, la neurona acepta o recibe las señales con las que realiza algún cómputo y genera una salida que emite a través del axón. Un esquema de lo anterior se puede observar en la Figura 2.2. El axón de las neuronas se ramifica permitiéndoles conectarse a las dendritas de otras neuronas. Esta conexión es llamada *sinapsis*, y se estima que el cerebro

humano tiene, en promedio, entre 10^{14} y 10^{15} de estas conexiones.

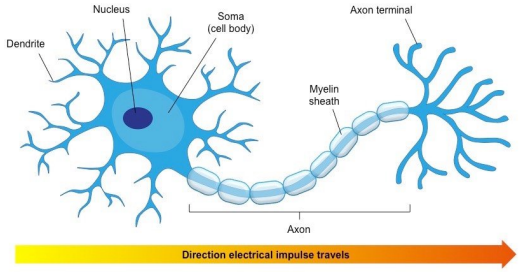


Figura 2.2: Anatomía de una neurona. Extraído de [90].

Como se verá en la siguiente sección, en este concepto general de neuronas e interconexiones, se basan las redes neuronales artificiales.

2.2. Redes neuronales y DNNs

Como se mencionó anteriormente, las NNs son redes artificiales diseñadas para resolver problemas imitando el funcionamiento del cerebro humano, mediante la aplicación de diversos algoritmos. De igual forma que en un programa convencional, es necesario mapear o discretizar la realidad en variables y parámetros que van a permitir resolver el problema. En este caso, en la red neuronal se pueden identificar las siguientes componentes, donde algunas tienen una correspondencia con su contraparte biológica: las entradas o *inputs* (x_i), los pesos o *weights* (w_i), *bias* (b), las salidas o *outputs* (y_j) y una función de activación (f). La Figura 2.3 representa gráficamente esta descripción. La fórmula que modela el comportamiento y principal operación que involucra los parámetros del ejemplo se presenta en la Ecuación 2.1 .

$$y = f(\sum_{i=0}^{2} w_i \times x_i + b) \tag{2.1}$$

Básicamente, una neurona o más específicamente un *perceptrón* [103], se encarga de computar una suma ponderada de sus entradas, donde, para que la salida sea un resultado útil, es necesario "ajustar" los parámetros[1] (pesos y bias) en base a la entrada y el resultado esperado. Este mismo cómputo de una sola neurona, puede ser agrupado en lo que se llaman capas o *layers*, donde cada neurona de una misma capa computa de forma muy similar a las demás con

[1] De forma muy similar a una regresión lineal.

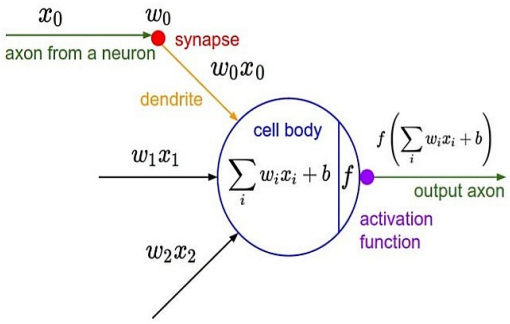


Figura 2.3: Esquema de una neurona (perceptrón), como bloque de construcción de una red neuronal, junto con sus componentes principales. Extraído de [40]

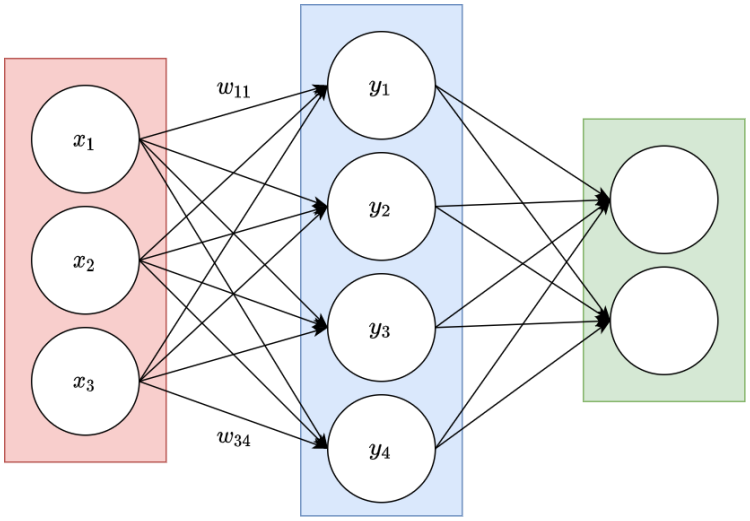


Figura 2.4: Neuronas asociadas por capas junto con las entradas y salidas.

un conjunto de pesos distintos, como se muestra en la Figura 2.4. Luego estas sumas ponderadas se pasan por una función no lineal, llamada comúnmente función de *activación*, algunos ejemplos de instancias de esta función son: Step, ReLU, Sigmoide, y SoftMax. Esto permite representar conjuntos más diversos de funciones, ya que en caso de prescindir de ellas, el resultado final de lo que modela la red, pasando por las diferentes capas, podría ser simplificado como una simple función lineal y, debido a lo complejas que pueden ser las realidades, esto en general no es suficiente.

2.3. Entrenamiento e Inferencia

El cómputo de las redes neuronales puede ser sintetizado en dos grandes procesos: entrenamiento (*training*) e inferencia (*inference*). El entrenamiento

refiere al proceso de ajuste y aprendizaje de los parámetros de la red neuronal para conseguir cierto objetivo, que se traduce en la minimización de cierta función $\mathcal{L}$ de costo o error. Como se menciona anteriormente, los parámetros de las redes neuronales consisten principalmente en los pesos y los bias. Luego de obtener los parámetros que verifican cierta tolerancia indicada en los hiper-parámetros[1] en el proceso de entrenamiento, se entiende que la red está "entrenada". En este punto, se considera que la red está en condiciones de realizar la tarea para la cual fue diseñada, computando para una determinada entrada[2], una salida aplicando un *forward-pass* o pasada hacia adelante de los datos de entrada a través de la red. A esta etapa, se la conoce como inferencia.

Una de las diferencias fundamentales entre el entrenamiento y la inferencia en redes neuronales se centra en el volumen de cálculos involucrados y la cantidad de datos necesarios para llevar a cabo cada proceso. Mientras que la inferencia implica una ejecución relativamente sencilla de la red para un solo elemento o un conjunto de ellos, el entrenamiento conlleva la manipulación y procesamiento de una cantidad de datos significativamente mayor, muchas veces varios órdenes de magnitud superior, dependiendo de la complejidad del problema que la red neuronal busca resolver.

Es importante destacar que la mayoría de las técnicas empleadas para la optimización de redes neuronales, están enfocadas en la etapa de inferencia. Esto está motivado principalmente por dos razones. En primer término, por estimaciones que indican cómo ésta etapa es causante de un gran porcentaje del consumo energético relativo a las NNs (ver por ejemplo el caso de Nvidia y el crecimiento en la demanda de GPUs [20]). Por otro lado, mientras que el entrenamiento se realiza, generalmente, en servidores o en la nube, la inferencia se tiende a realizar sobre dispositivos con recursos de cómputo limitados, y en muchos casos autónomos, siendo de suma importancia entender y optimizar esta etapa.

2.3.1. Entrenamiento

Si bien las técnicas de ML y redes neuronales se remontan a los años 50, en ese entonces no tuvieron el impacto que alcanzaron hoy en día. Esto se

[1] Los hiper-parámetros de una red neuronal son los parámetros establecidos para el entrenamiento de la misma. Algunos de ellos son el tamaño de batch, número de épocas, y tasa de aprendizaje.

[2] Generalmente, muestras no vistas durante el entrenamiento.

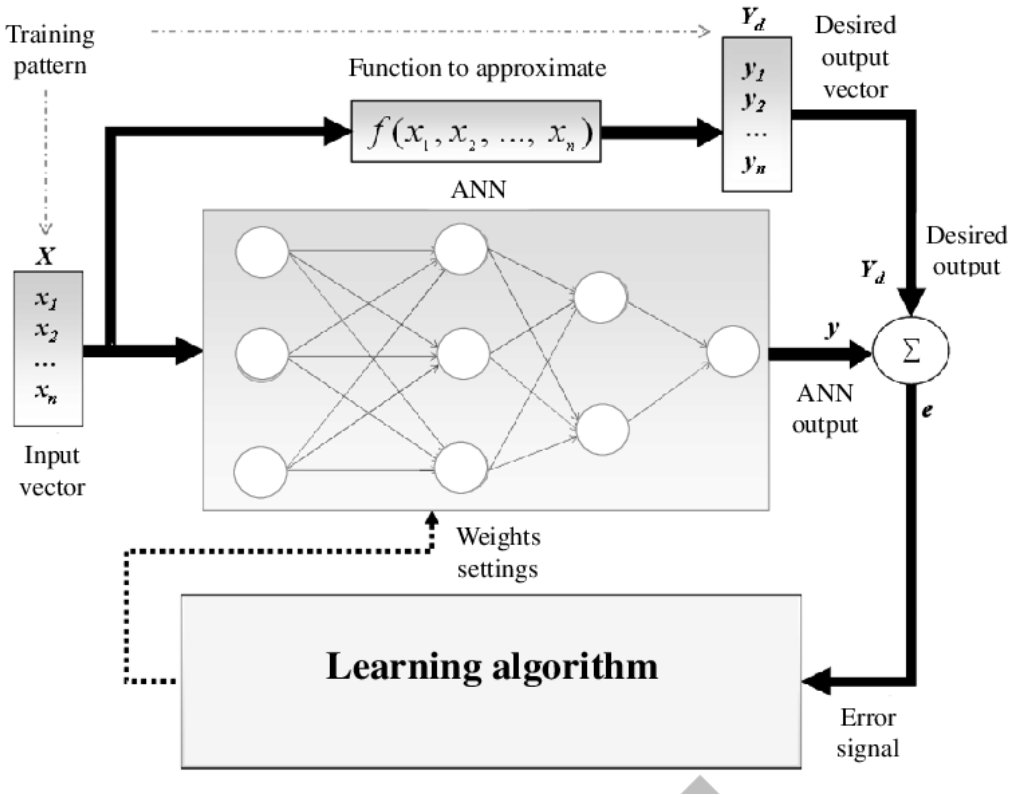


Figura 2.5: Esquema general de las etapa de entrenamiento en un contexto de aprendizaje supervisado. Extraído de [100].

debe, entre otros aspectos, a que uno de los requisitos para tener modelos que puedan generalizar o extraer cualidades de la realidad, es tener un conjunto de datos lo suficientemente grande, cualidad muy presente en la actualidad donde se tiene gran cantidad de "sensores".[1] La otra razón es la enorme capacidad de cómputo necesaria para manejar todo este flujo de datos, lo cual explica que el proceso de entrenamiento de la red se suela realizar en importantes centros de cómputo o simplemente en la nube (*cloud computing*).

Existen varias estrategias de aprendizaje o contextos que determinan las características del entrenamiento. Entre ellas están, por ejemplo, el aprendizaje supervisado, el no supervisado, por refuerzo, semi-supervisado y por transferencia. La estrategia adecuada depende fuertemente del problema que se intenta resolver. En este trabajo se abordará de forma breve las principales etapas de uno de los contextos de aprendizaje más comunes, el aprendizaje supervisado. La Figura 2.5 muestra de forma esquemática las principales etapas

[1]Sensores como concepto general, puede referir incluso hasta un celular que monitorea el nivel de actividad de un usuario frente a la pantalla hasta la cámara de un supermercado.

involucradas en este mecanismo de aprendizaje.

En el contexto de aprendizaje supervisado, al momento de ajustar los pesos del modelo de red neuronal para que esta aprenda, se busca minimizar una función de pérdida o de costo (L) que mide la diferencia entre las predicciones del modelo y los valores reales de los datos de entrenamiento. Para esto se debe contar con datos (por ejemplo, imágenes o textos) etiquetados con la salida esperada una vez procesado el mismo. Esta optimización se realiza de a pasos (*steps*) con cierto tamaño que indica la tasa de aprendizaje (*learning rate*).

Existen varias estrategias para minimizar la función de pérdida. El *descenso por gradiente*, es el método por excelencia y la base para el aprendizaje de los pesos de los modelos. Consiste en realizar la actualización de los pesos en el sentido contrario al gradiente de la función de pérdida de manera que esta se minimice.[1] Luego, al momento de implementar el descenso por gradiente existen distintas variantes, entre ellas el descenso por lotes (*Batch GD*), el estocástico (*Stochastic* GD) y la combinación de ambos (*Mini-batch GD*). El primero, por lotes, emplea para el cálculo del gradiente todas las muestras o datos de entrenamiento en cada paso. Esto impacta enormemente a los tiempos de entrenamiento, principalmente con datasets de grandes dimensiones. El SDG corrige esto empleando una muestra ($batch\_size = 1$) aleatoria del conjunto de datos. Esto acelera enormemente los tiempos de entrenamiento, con la desventaja de que no es tan regular al momento de minimizar la función de pérdida, debido a la componente aleatoria de los datos empleados para el cómputo. La última estrategia, combina estas dos empleando un $batch\_size > 1$, esto permite sacar partido de arquitecturas de hardware con optimizaciones de operaciones matriciales y cómputo paralelo, en específico las GPUs [42]. También existen otras variantes que tienen que ver con el ajuste del *learning rate* a lo largo del proceso de entrenamiento. Algunos ejemplos son, el Gradient Descent con Momentum y Adaptive Moment Estimation (*Adam*) [66].

A su vez, al momento de entrenar el modelo se realizan tantas recorridas sobre el dataset como se indique. Cada recorrida completa sobre el dataset se denomina *epoch*.

La técnica de auto-ajuste empleada en las redes neuronales para poder aprender una representación interna de los datos que tiene que procesar, así como la salida correcta o esperada para esa entrada, es conocida como Back

[1] El gradiente representa la derivada de la función de pérdida, es decir la dirección en la que crece.

Propagation [104] o propagación hacia atrás. Básicamente, a través del suministro de datos de entrenamiento válidos para el modelo, se obtiene una salida y en función a ésta, se computa el error cometido comparado con el valor esperado. Seguidamente el error es retro-propagado a lo largo de la red, penalizando a las neuronas de la capa previa en función de cuánto peso tuvieron sobre el resultado final, responsabilizando recursivamente a cada neurona por el error cometido.

2.3.2. Inferencia

El proceso de inferir un resultado a través de la ejecución de una red neuronal con algún dato de entrada es, como el proceso de entrenamiento, un cómputo costoso pero de mucha menor duración. En el entrenamiento se realizan tantas inferencias (forward-pass), además de las etapas de back-propagation, como sean necesarias para minimizar la función de coste o error. En general, la inferencia tiene, dependiendo de la aplicación, ciertas restricciones. Si bien esta etapa puede realizarse en grandes centros de cómputo, muchas veces es de interés que la inferencia sea realizada en equipos de bajo porte y en muchos casos en dispositivos *edge* o de borde. Las ventajas que presenta este enfoque son principalmente:

- **Privacidad**: evitar que los datos, a veces personales o sensibles, viajen a través de internet y sean enviados a centros de cómputo.
- **Conectividad**: muchas veces se cuenta con un ancho de banda limitado o incluso nulo para acceder a servicios de cómputo por red.
- **Latencia**: por ejemplo, en aplicaciones de tiempo real como pueden ser la conducción autónoma de vehículos, donde es necesario tomar decisiones en el "acto", los tiempos para realizar la inferencia tienen que ser menores que los tiempos de transferencia.

2.4. Modelos de redes neuronales

Lo que caracteriza a los distintos modelos de redes neuronales es básicamente su arquitectura. Dicha arquitectura[1] refiere a cuáles bloques de procesamiento emplea y cómo están dispuestos e interconectados en la red neuronal.

[1]Similar a la del hardware, haciendo referencia a cómo estos disponen memorias, transistores y unidades de cómputo, como bloques de construcción.

De esta forma no es necesario hablar de las neuronas como mínima unidad sino que, con cierto nivel de abstracción, es posible remitirse a los distintos tipos de capas que conforman la NN y cómo éstas están interconectadas.

2.4.1. NNs más utilizadas

A continuación se describen brevemente algunos de los tipos de redes más utilizados, las tareas que permiten resolver, las capas que emplean y como se interconectan.

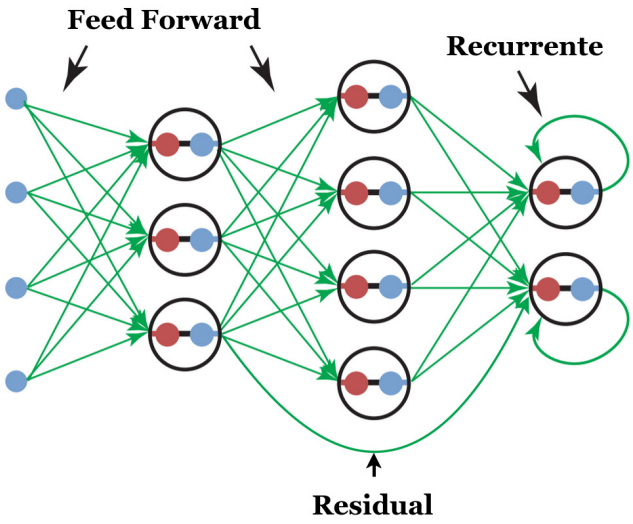


Figura 2.6: Tipos de conexiones en las redes neuronales (Figura adaptada de [114]).

Redes Feed-forward

Posiblemente el caso más común de redes neuronales son las *feed-forward*, en las que cada nodo de una capa recibe como entradas los valores de las salidas de la capa anterior y los datos fluyen "hacia adelante" (desde la capa de entrada a la de salida). Los Perceptrones Multicapa (MLP del inglés MultiLayer Perceptron) son un ejemplo de este tipo de redes. Son ideales para problemas donde la relación entre la entrada y la salida no cambia con el tiempo, como la clasificación de imágenes.

Redes Neuronales Recurrentes (RNNs)

A diferencia de las feed-forward, las redes neuronales recurrentes presentan neuronas con interconexiones a capas anteriores o hacia atrás (feed-back) o re-

currentes. Este tipo de redes son principalmente eficaces en tareas que implican secuencias de datos, como procesamiento del lenguaje natural (PLN), modelado del habla, traducción automática, entre otras. Las conexiones a capas anteriores funcionan, de alguna manera, como un estado, "memoria" o como entrada en cierto tiempo posterior del procesamiento de la secuencia de datos. Existen otras redes con grandes capacidades de procesar datos secuenciales como el texto, en particular los llamados LLM (*Large Language Models*). Aquellos con mayor impacto en los últimos tiempos son los basados en *Transformers*, que incorporan *mecanismos de atención* variable a los diferentes elementos a procesar [21, 123].

Redes Residuales (ResNets)

Las conexiones residuales, introducidas en las Redes Resiudales Profundas [51], consisten en enlaces dados entre capas no adyacentes. Este tipo de redes cuenta con la ventaja de sufrir una menor degradación del gradiente al momento de entrenar, en particular cuando las redes son muy profundas y la magnitud del error es pequeña.

Redes Neuronales Convolucionales (CNNs)

Este tipo de redes es muy utilizado para la visión por computadora, clasificación de imágenes, detección de objetos, segmentación de imágenes, entre otras tareas. Son redes de múltiples capas, involucrando principalmente capas convolucionales, totalmente conectadas (FC) y de pooling. Generalmente las capas convolucionales funcionan como extractores de propiedades sobre las imágenes, mientras que las capas de pooling reducen las dimensiones de las salidas de las capas anteriores y finalmente se obtiene el resultado a través de una capa FC.

2.4.2. Principales clases de capas y kernels

En las siguientes secciones se abordan los tipos de capas más usadas para implementar los modelos de aprendizaje profundo, además de un estudio de las principales operaciones involucradas para computar dichas capas.

Totalmente conectadas (FC - Fully Connected)

En las capas FC o totalmente conectadas como la de la Figura 2.4, cada neurona está conectada a todas sus predecesoras. Por lo tanto, agregar una neurona a este tipo de capa implica el manejo de tantos nuevos pesos como neuronas en la capa anterior. De similar forma impacta la adición de una neurona en la capa previa a la FC, volviéndose capas con un costo computacional sumamente elevado a medida que aumenta el tamaño de entrada. La utilidad y la capacidad de las capas FC para aprender y clasificar datos es notable [115], sin embargo la principal desventaja que presentan está dada por la alta conectividad entre las neuronas, que las vuelve poco prácticas a la hora de entrenarlas y computarlas para datos de gran tamaño, como por ejemplo imágenes en alta resolución.

Notar que en este caso, cada neurona de la capa FC, debe computar una suma ponderada de la salida o activación de las neuronas de la capa anterior, que en conjunto forman lo que se denomina *feature map* (*fmaps*). Entonces, el resultado de cada neurona puede pensarse como un producto escalar de dos vectores, la i-ésima neurona tendrá asociado el vector de pesos w_i (de tamaño $1 \times CHW$ donde CHW es la cantidad de neuronas de la capa anterior) que luego se multiplica por el vector de activaciones de la capa anterior $ifmap$ ($CHW \times 1$). Notar que cada neurona computa con el mismo vector de activaciones de entrada, por lo tanto, se puede generalizar y representar todo como una multiplicación matriz-vector, la matriz de los pesos asociados a la capa FC, de dimensiones $M \times CHW$, siendo M la cantidad de neuronas de la capa FC. Incluso es posible subir un nivel, generalizando el cómputo de esta capa de a batches. En lugar de tener un vector de activaciones ($CHW \times 1$) de la capa anterior, se tendría ahora una matriz ($CHW \times N$) donde N es el tamaño de batch, computando en este caso una multiplicación matriz-matriz, tal como se muestra en la Figura 2.7. Notar que las multiplicaciones de matrices, en todas sus variantes (matriz-matriz, matriz-vector, en formatos dispersos, etc.), son operaciones ampliamente abordadas por la comunidad científica, siendo estudiada y optimizada para múltiples plataformas de hardware [22][2].

Convolucionales (CONV)

Las capas convolucionales son aquellas que involucran una convolución en el cómputo de cada neurona. Esta operación no es nueva dentro del ámbito de

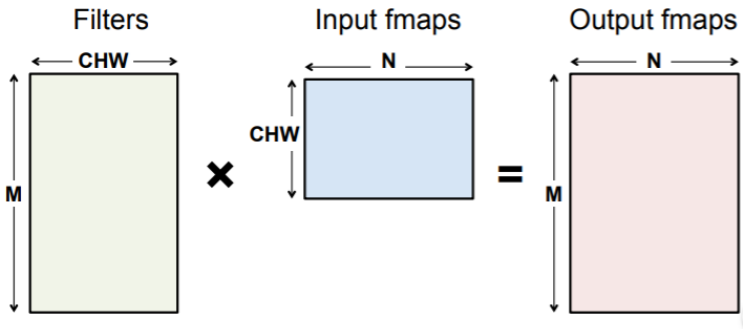


Figura 2.7: Cómputo de una capa FC (con N la cantidad de muestras del batch) expresado como una multiplicación matriz-matriz. Extraído de [115].

la ciencia y la ingeniería, ya que ha sido sumamente empleada en campos que requieren procesamiento y tratamiento de imágenes [43]. En el mismo principio se basan también las redes neuronales encargadas de procesar imágenes, que generalmente poseen varias de estas capas, donde las primeras están enfocadas en encontrar cualidades o *features* básicas de la imagen, y cuánto más profundo en los niveles de capas nos movemos, más específicas son las características que logran identificar. Por ejemplo, en redes neuronales encargadas de reconocer rostros, las primeras capas convolucionales generalmente van a detectar bordes y curvas genéricas que, combinadas, van a funcionar como entradas a otras capas siguientes que detectan características más específicas como: ojos, narices, bocas, y a su vez, estas unidas permiten detectar rostros. La convolución consiste, básicamente, en una rutina matricial que involucra un filtro (*filter*, en otras literaturas y contextos *kernel*) de tamaño $R \times S$ (generalmente cuadrado, p.ej. 3×3) y la imagen de entrada a procesar de dimensiones $H \times W$. Dicho filtro se multiplica elemento a elemento con cada porción de la imagen, a veces llamado ventana y luego este resultado es reducido, sumando cada multiplicación elemento a elemento, formando un nuevo píxel de salida, como se muestra en la Figura 2.8. Generalmente los *features* de entrada, cuentan con una dimensión más, llamada canales o *channels*, pasando a tener dimensiones $C \times H \times W$. En estos casos, los filtros involucrados en la convolución también agregan esta dimensión (p.ej. en imágenes RGB, un tamaño de filtro podría ser $3 \times 3 \times 3$). Posteriormente esta operación se repite moviendo la ventana de entrada sobre la imagen hasta recorrerla por completo.

Existen casos particulares de convoluciones, como lo es la convolución *depthwise* [55], donde cada filtro se aplica de manera independiente a un solo

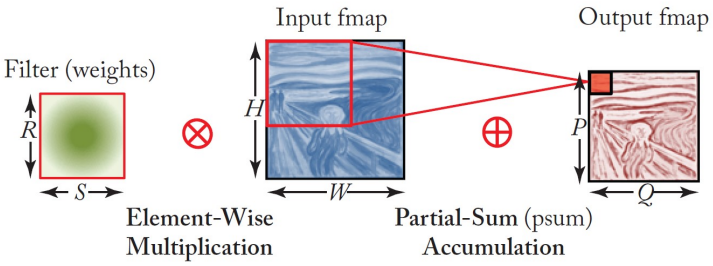


Figura 2.8: Convolución 2D. Extraído de [114].

canal de la entrada.

Funciones de no linealidad

En las secciones anteriores, cuando se introdujo la idea del perceptrón, de forma muy breve se introdujo también el concepto de funciones no lineales. Particularmente, las funciones de no linealidad o de activación son aplicadas a la salida de las neuronas, con el fin de permitir a la red representar una cantidad mas variada de funciones complejas, y no solo combinaciones lineales de las entradas. Entre las más utilizadas se encuentran: *Step*, *ReLU* (Rectified Linear Unit), *sigmoide* y *tanh* (Tangente Hiperbólica), expresadas en las ecuaciones de la Tabla 2.1. En la Figura 2.9 se muestran estos ejemplos de forma gráfica.

Step	$y = \begin{cases} 1 & \text{si } x > 0, \\ 0 & \text{en otro caso} \end{cases}$
ReLU	$y = \begin{cases} x & \text{si } x > 0, \\ 0 & \text{en otro caso} \end{cases}$
Sigmoide	$y = \frac{1}{1+e^{-x}}$
tanh	$y = tanh(x)$

Tabla 2.1: Ecuaciones de las funciones de activación más utilizadas.

Pooling y Unpooling

Son un tipo de capas no entrenables muy presentes en las redes neuronales. La idea de las capas pooling es reducir la dimensionalidad o resolución de los feature maps generados por capas anteriores, aplicando distintos operadores, como pueden ser el máximo o el promedio sobre regiones de la imagen. Por ejemplo, una capa MaxPool 2x2 y stride 2, permitiría pasar de una salida 4×4

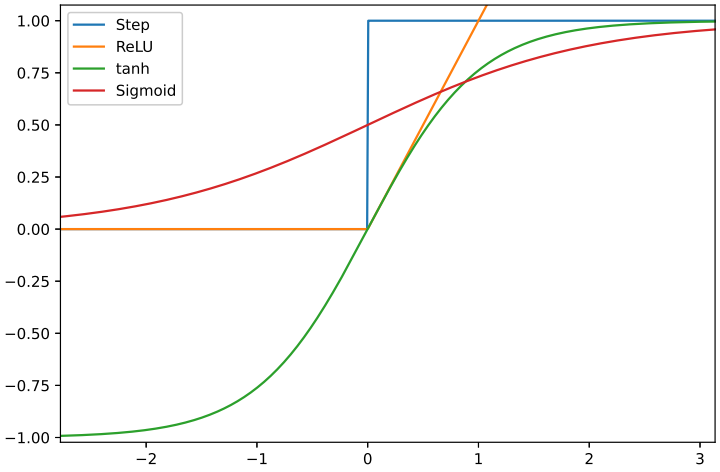


Figura 2.9: Gráfico de las funciones de activación más comunes.

a una de 2×2 (ver Figura 2.10). Entre las ventajas que aportan este tipo de capas, se destaca la reducción de complejidad y cantidad de parámetros que tiene que aprender la red, reduciendo así también la posibilidad de sobreajuste. En este sentido, tiende a ser ventajosa su aplicación antes de una capa FC para reducir la cantidad de pesos necesarios. También permite obtener cierta invariancia ante traslaciones y modificaciones pequeñas de las imágenes.

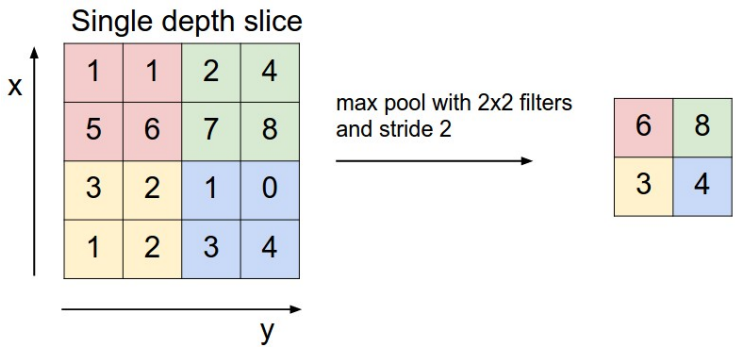


Figura 2.10: MaxPool (2x2, stride=2). Extraído de [40].

Por otra parte, las capas unpooling funcionan como la inversa de las pooling, permitiendo aumentar la dimensionalidad. En ocasiones se habla de que realizan un *upsampling* o re-muestreo de los datos. Existen varias estrategias, algunas replican valores, otros rellenan con cero.

Generalmente la combinación de estas capas involucra algún tipo de pérdida de información [42, p. 366].

Otras capas y combinaciones

Existen múltiples tipos de capas más que son utilizados para la implementación de redes neuronales. Muchos de ellos resultan de combinar algunas de las primitivas descritas anteriormente. Las capas de *atención* [123], por ejemplo, popularizadas en los Transformers, se basan en multiplicaciones de matrices, capas FC y conexiones feed-forward.

La capa *up-convolution* [36, 69] o des-convolución (a veces también llamada convolución traspuesta), es otro ejemplo de capa compuesta. Básicamente representa el inverso de la convolución, combina un unpooling para aumentar la dimensionalidad, seguida de convoluciones con varios filtros, típicamente se utiliza en redes como General Adversarial Networks (GANs) [45] y Auto Encoders (AEs) que procesan imágenes.

2.5. Métricas

La creciente importancia que se le ha dado a las redes neuronales, ha desencadenado el estudio e investigación de técnicas y arquitecturas que permitan mejorar el rendimiento de este tipo de programas. Para poder hablar de cómputo eficiente de redes neuronales, es necesario definir ciertas métricas y parámetros a medir y evaluar en el proceso de optimización de los *kernels* tanto, desde el punto de vista del modelo asociado a la red neuronal, así como del hardware encargado de ejecutar dicha red.

A continuación se describen, de forma breve, varias de estas métricas. Posiblemente, una de las métricas más comunes empleadas para evaluar una arquitectura, en la comunidad científica e industrial, es la cantidad de operaciones que esta puede realizar por segundo, medida en TOPS o Tera Operations Per Second. Cuando se tiene en cuenta la eficiencia energética, la tendencia es reportar la cantidad de operaciones por segundo que se pueden realizar con cierta cantidad de potencia, por ejemplo, TOPS/W. Autores como, Sze et al. [114], sostienen que estas métricas en muchos casos resultan insuficientes para referirse a cuán óptima y eficiente puede llegar a ser una implementación. Por lo tanto es importante evaluar otras métricas, como las desarrolladas en las siguientes sub-secciones.

2.5.1. Accuracy

La precisión (o del inglés, *accuracy*) es una métrica comúnmente utilizada para evaluar la calidad de un modelo de red neuronal. Específicamente, el accuracy (estándar) se define como "la proporción de predicciones correctas realizadas por un modelo en un conjunto de datos", generalmente reportada como un porcentaje. Su interpretación puede variar mucho según el contexto y la naturaleza del problema, por ejemplo, es especialmente importante en tareas de clasificación donde se busca minimizar el número de predicciones incorrectas. Sin embargo, se debe tomar en cuenta que la precisión por si sola puede no ser suficiente, por ejemplo, en situaciones donde se presente un desbalance de clases en las muestras usadas para el entrenamiento y evaluación. También, en algunos problemas, los errores de clasificación de diferentes clases pueden tener diferentes implicaciones y costos. Por ejemplo, en un sistema de detección de enfermedades, un falso negativo (una enfermedad no detectada) puede ser mucho más grave que un falso positivo (un paciente sano clasificado como enfermo). Para estos casos existen otras métricas de utilidad [42], algunas son el *recall* (sensibilidad), la especificidad y el *F1-score*.

De todos modos, a la hora optimizar un modelo de redes neuronales con posibles impactos sobre la calidad de los resultados, es deseable evaluar y reportar el accuracy. En este sentido se deben tener en cuenta varios factores, entre ellos, la precisión inicial del modelo, la dificultad de la tarea a resolver y también el conjunto de datos con el que se está evaluando el modelo.

Además de la precisión estándar, existen otras formas de precisión que pueden ser más apropiadas para ciertos problemas. Por ejemplo, la precisión Top-k mide la proporción de predicciones donde la clase correcta se encuentra entre las k primeras predicciones, especialmente útil para datasets muy grandes y con muchas clases o categorías distintas [95].

2.5.2. Latencia y Throughput (Productividad)

A diferencia de la anterior métrica, que depende únicamente de la arquitectura de la red neuronal y de su entrenamiento, a continuación se introducen dos métricas de interés, en este caso, asociadas a la implementación de la red neuronal sobre una plataforma de hardware específica. El *throughput* habla de la capacidad, principalmente desde el hardware, de procesar datos y se reporta en cantidad de operaciones por segundo. Algunas medidas de throughput en

el contexto de redes neuronales son la cantidad de TOPS (tera operaciones por segundo), la cantidad de muestras de entrada que se pueden procesar por unidad de tiempo (p.ej. en un segundo), o en aplicaciones que procesan imágenes, la cantidad de imágenes o *frames* procesados por segundo (fps). Cuando es necesario procesar grandes cantidades de información, a mayor throughput mejor va a ser el rendimiento total del sistema.

Otro concepto importante es el de *latencia*, métrica que hace referencia al tiempo necesario en procesar una sola muestra, generalmente medida en milisegundos (ms). La latencia es un punto crítico en los sistemas de tiempo real como, por ejemplo, la conducción autónoma, donde los sensores muestrean datos periódicamente y la plataforma de hardware debe ser capaz de procesar esa información y el cómputo de la red neuronal con la menor demora o latencia posible, para tomar decisiones a tiempo.

2.5.3. Consumo y eficiencia energética

Otras métricas muy importantes que deben ser reportadas, son aquellas asociadas al consumo energético de la red neuronal y la plataforma de hardware que se encarga de ejecutarla. La *potencia* entonces, indica la cantidad de energía insumida por unidad de tiempo de la plataforma de hardware, asociada a la cantidad máxima de calor disipable por el sistema de enfriamiento. Se reporta, generalmente, en Watts.

Por otro lado, la *eficiencia energética* está directamente asociada a la tarea en ejecución y describe la cantidad de datos o de ejecuciones de la tarea que se pueden realizar por unidad de energía. En el caso de las NNs y la inferencia, la eficiencia energética se expresa en inferencias por Joule, o con la inversa como Joules por inferencia.

Ambas métricas son fundamentales para la implementación de redes neuronales. En los dispositivos embebidos alimentados por baterías, por ejemplo, la eficiencia energética es de suma importancia, dada la limitante en el consumo y potencia que las baterías pueden proveer. Es igualmente importante en datacenters, debido a que la gran demanda computacional que implica el cómputo deriva en un importante consumo eléctrico, además de la generación de calor. La limitada capacidad de disipación del mismo en muchos casos ha impulsado el desarrollo de técnicas más complejas y costosas de enfriamiento [38] que también implican un elevado consumo eléctrico.

2.5.4. Otras métricas

Además de las antes descritas, pueden ser enumeradas otras métricas. Por ejemplo, en [114] son abordadas aquellas que están más relacionadas con la implementación de nuevas arquitecturas de hardware para la resolución de un problema específico. Una de ellas es el *costo*, que apunta, entre otras cosas, al tiempo y esfuerzo de diseño de una nueva plataforma de hardware, así como al precio de la pieza de hardware que se plantea usar como solución que, en muchos casos, puede ser una restricción en la realidad planteada.

Otra métrica relacionada al hardware es la *flexibilidad*, que indica el rango de redes neuronales y tareas con las que la plataforma puede operar de forma eficiente. En general, es preferible una gran flexibilidad, siempre y cuando no impacte drásticamente el desempeño. Esta puede ser medida, por ejemplo, evaluando su desempeño para un subconjunto variado de modelos, como pueden ser los disponibles en *benchmarks* como MLBench [118]. Por último, se puede agregar la *escalabilidad*, que señala la capacidad del diseño de adaptarse al momento de aumentar o expandir la capacidad de cómputo de la plataforma y que al mismo tiempo esto derive en un aumento de desempeño real.

2.6. Plataformas de hardware

Al abordar las redes neuronales es de suma importancia estudiar también las plataformas de hardware[1] utilizadas para computar estos programas. Como se menciona en la Sección 2.4 las redes neuronales, por lo general, tienden a quedar definidas por *kernels* básicos de cómputo asociados a las operaciones más costosas de la ejecución. Por eso, se discutirán las principales plataformas de hardware utilizadas, poniendo especial foco en la relación de cada una con respecto a la flexibilidad de configuración y la eficiencia que alcanzan a la hora de resolver los distintos kernels. Por sus características, son generalmente enmarcadas en el campo de la computación de alto desempeño (HPC por su sigla en inglés). Esta discusión es especialmente relevante en la actualidad, considerando que autores como John L. Hennessy y David A. Patterson [52] recientemente han destacado los principales desafíos del área en la actualidad, como el límite físico de la Ley de Moore, la computación en la nube, la im-

[1] En el documento se evitará utilizar el término arquitectura en el contexto de hardware para no incurrir en confusiones con las arquitecturas de las redes neuronales.

portancia del aprendizaje automático y las técnicas de inteligencia artificial, adelantando hacia dónde se dirige el futuro de las arquitecturas de computadoras y la necesidad de especialización.

Como se introdujo en las secciones anteriores, las principales operaciones que intervienen en la ejecución de una red neuronal, siendo computadas por cada neurona, son las multiplicaciones y acumulaciones (MACs). Por esta razón, las MACs, son operaciones esenciales en las NNs funcionando como un bloque mínimo para la construcción de estos sistemas y el cómputo de sumas ponderadas, así como la base de multiplicaciones vector-matriz y matriz-matriz. Si bien el cómputo de una MAC, que involucra tres parámetros, puede aparentar ser trivial, para realizar cada operación es necesario leer de memoria tres datos (un peso, una activación y una suma parcial), y luego de computada la multiplicación y acumulación se impacta ese resultado en memoria, como lo indica la Figura 2.11. Al igual que otros autores, Horowitz [53] indica que la

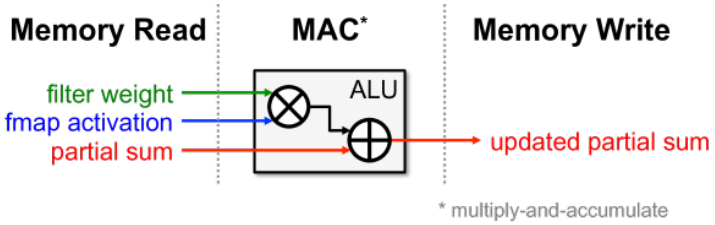


Figura 2.11: Accesos de lectura y escritura para el cómputo de una MAC, como bloque de una NN. Extraído de [114].

mayor parte del consumo energético en una importante cantidad de algoritmos no se da por la realización de MACs, sino por el movimiento de los datos a través de las jerarquías de memorias, como la trasferencia de datos o valores desde la DRAM hacia las unidades encargadas de ejecutar dichas MACs. En la Figura 2.12 están plasmados algunos de los costos que suponen los accesos a memoria, junto al costo de realizar distintas operaciones. Notar que, cuanto más "lejos" se vayan a buscar los datos, pasando desde las memorias on-chip como los registros o la SRAM, hasta memorias off-chip como la DRAM, mayor es el tiempo y la energía necesaria para obtener dichos datos con diferencias de varios órdenes de magnitud entre algunos tipos de memoria. Por ejemplo, para ejecutar la red neuronal MobileNetV2[1] son necesarias unos 300M de MACs, implicando, en el peor caso, 1200M accesos a DRAM, e impactando directa-

[1]Red eficiente muy utilizada y estudiada en la comunidad que se profundiza en el Capítulo 4

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3 pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1pJ
32 bit	4pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

Figura 2.12: Valores aproximados de energía consumida por operación (45nm). Extraído de [53].

mente en el desempeño de la red (tiempo de ejecución, consumo energético, etc.). Para estos casos, la solución ha sido emplear las ya conocidas jerarquías de memorias, incluyendo memorias más pequeñas pero más rápidas y eficientes (*buffers*, *scratchpads*, registros, cachés), donde la forma de explotar los beneficios que éstas ofrecen, es la reutilización de datos. Afortunadamente el esquema de las redes neuronales es sumamente apto para la reutilización de datos. Un ejemplo, durante la ejecución de capas de convolución, son los filtros, es decir los pesos que se aplican a las activaciones de entrada asociados a la capa anterior. Estos son compartidos por muchas de las unidades que ejecutan las multiplicaciones y acumulaciones, por lo que tenerlos en memorias intermedias, cercanas a los procesadores, permitiría reducir la cantidad de accesos a memorias más lentas, como la DRAM.

Otra técnica, que logra sacar partido de la mayoría de las plataformas, como se presentará en las siguientes secciones, y no es ajena a las redes neuronales, es el cómputo paralelo. Esto se debe, entre otras razones, a que muchas capas pueden ser tratadas como una multiplicación de matrices, y existen diversas técnicas para paralelizar este tipo de operaciones [2, 67]. Por ejemplo, en las capas FC el cálculo de cada entrada de la matriz de resultado es independiente. Esto sucede también en las capas convolucionales, donde el producto de cada peso del filtro, por cada entrada o activación, es independiente. Por esta razón, las plataformas que ofrecen la capacidad de computar en paralelo son las más empleadas para la implementación de redes neuronales.

La mayor parte de las plataformas de hardware que explotan el paralelismo, pueden ser incluidas en dos categorías, las basadas en arquitecturas de hardware temporales y espaciales, como se muestra en la Figura 2.13. Ejemplos

de arquitecturas de hardware temporales son la CPU y GPU, con una o, en algunos casos, varias unidades de control centralizadas, para un gran número de ALUs. Además, en esta familia de paralelismo los accesos a datos se realizan a través de lecturas directas a la jerarquía de memorias. Las arquitecturas de hardware espaciales, en cambio, presentan un esquema distinto de ejecución basado en un flujo de datos. En este caso todas las unidades aritméticas poseen una unidad de control, además de una memoria local, conformando una unidad de procesamiento independiente, a veces llamada Proccessing Element (PE). Este tipo de plataformas, cuenta también con una interfaz que interconecta estas unidades (Network on Chip - NoC), posibilitando que compartan datos sin tener que acceder necesariamente a los sistemas centralizados de memoria. Estas últimas son un esquema muy atractivo para la implementación de aceleradores de redes neuronales debido a la forma en que fluyen los datos y cómo estos pueden ser computados, además de compartidos, a través del arreglo sistólico de unidades (término tomado de medicina basado en el sistema circulatorio humano, por la forma en la que fluyen los datos como un bombeo rítmico -sístole- a través de la red que interconecta a los PEs [70]).

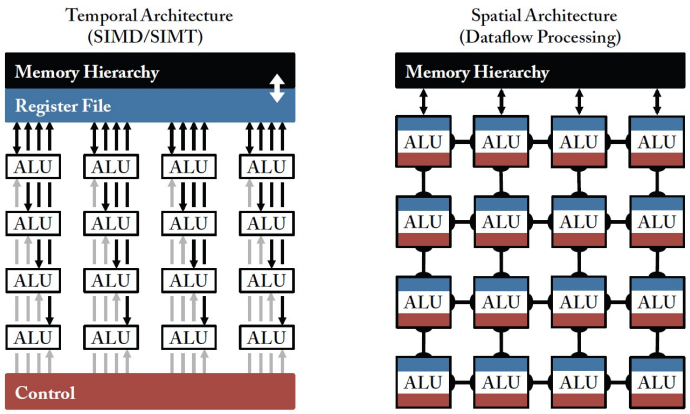


Figura 2.13: Comparación gráfica de arquitecturas que explotan paralelismo. Extraído de [115].

A continuación se presenta, de forma breve, una descripción de las plataformas de hardware, tanto espaciales como temporales, más consolidadas para este tipo de cómputo, diferenciándolas según los distintos paradigmas en cuanto al flujo de los datos y al paralelismo que ofrecen. Las plataformas serán presentadas en un orden decreciente de generalidad permitiendo introducir, a través del trayecto, conceptos claves para entender las plataformas de dominio específico.

2.6.1. CPU

La CPU o del inglés, Central Processing Unit es, posiblemente, la unidad de cómputo más conocida y extendida, encargada de ejecutar instrucciones en la mayoría de los computadores. Se basa en la estructura o arquitectura de *Von-Neumann* [111], propuesta en el año 1945 principalmente, para la ejecución secuencial de instrucciones, permitiendo resolver diferentes problemas.

Al día de hoy los diseños de CPU han experimentado modificaciones para abastecer la demanda de la mayoría de los usuarios, que consiste en la ejecución de múltiples programas al mismo tiempo. Por lo tanto, una gran parte de la capacidad de las CPUs se centra justamente en darle a los usuarios la mejor experiencia, funcionando como organizador de los recursos del sistema. Para esto están dotadas de una o más unidades de cómputo o cores, grandes memorias caché, permitiendo la ejecución de múltiples hilos al mismo tiempo. En particular, las CPUs tienen una gran mayoría de los recursos de hardware (los transistores) dedicados a tareas como: predicción de branches (para poder ejecutar secuencialmente y explotar las técnicas de *pipelining*), pre-fetch de memoria, y ejecución fuera de orden; técnicas que permiten mejorar la fluidez que percibe el usuario al ejecutar tareas de propósito general.

Por lo tanto, en su concepción, las CPUs son sin lugar a dudas sumamente flexibles y presentan gran generalidad en cuanto a los problemas que pueden resolver, pero el descenso o estancamiento de la ley de Moore [107], las ha convertido en una solución sub-óptima para la ejecución de redes neuronales con restricciones como latencia o potencia. Sin embargo, en los últimos años se han propuesto múltiples tipos de unidades de cómputo para ML, basadas en esta arquitectura de hardware. En [12] se presenta una breve discusión de plataformas emergentes para ML, entre ellas destacan varias soluciones que emplean CPUs con bajo consumo energético y poca área de circuitos que resulta en sistemas de bajo costo.

2.6.2. GPU

Las Unidades de Procesamiento Gráfico o (GPUs del inglés Graphics Proccessing Units), son un tipo de dispositivo diseñado para la resolución de tareas que pueden ser computadas de forma masivamente paralela, o dicho de otra forma, problemas que pueden ser divididos en una gran cantidad de subproblemas independientes entre sí, que pueden resolverse de forma simultánea.

Particularmente, las redes neuronales son un claro ejemplo de un problema que puede ser abordado con este paradigma.

La GPU también se basa en una arquitectura de hardware temporal como la CPU, en este caso, adopta la idea del *multicore* o múltiples unidades de cómputo, y lo explota a gran escala. A diferencia de las CPUs (que tienen generalmente decenas de cores), las GPUs dedican la mayoría de los transistores al cálculo, logrando agregar cantidades impresionantes de unidades de cómputo (por ejemplo, la GeForce RTX 3090 Ti de NVIDIA, con 10752 CUDA cores [24]). La Figura 2.14 muestra una comparativa, aproximada, de la proporción de transistores que está dedicado a cada componente dentro de una CPU y una GPU. Mientras que gran parte de la capacidad de las CPUs está dedicada a control, intentando mejorar el tiempo de ejecución secuencial, las GPUs enfocan estos transistores en agregar más unidades de cómputo, reduciendo las unidades de control.

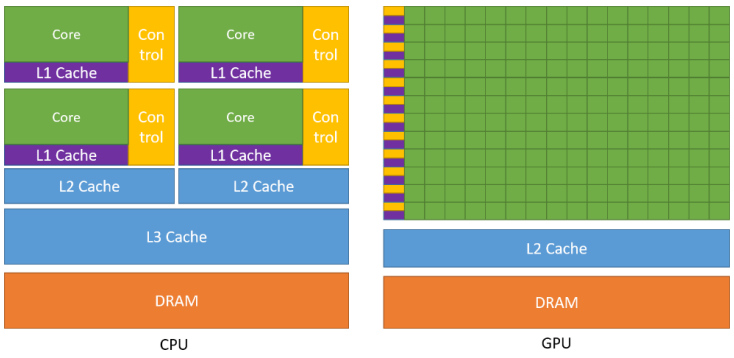


Figura 2.14: Diferencia de arquitectura entre una CPU y una GPU. Extraído de [3].

2.6.2.1. Tensor Cores

Uno de los cambios principales que presenta la generación de GPUs Volta [8], presentada por NVIDIA en el año 2017, con respecto a Pascal (su predecesora) es la inclusión de una nueva unidad de cómputo, los *Tensor Core*. Esta unidad funcional aparece para mitigar el creciente costo de entrenar redes neuronales que, debido al crecimiento en la complejidad de los problemas que éstas intentan resolver, aumentan la cantidad de atributos que manejan, y por consiguiente los parámetros a entrenar. Los Tensor Cores se encargan, en pocas palabras, de realizar multiplicaciones y acumulaciones de matrices, operación de suma importancia en las redes neuronales debido a que, como se pudo ob-

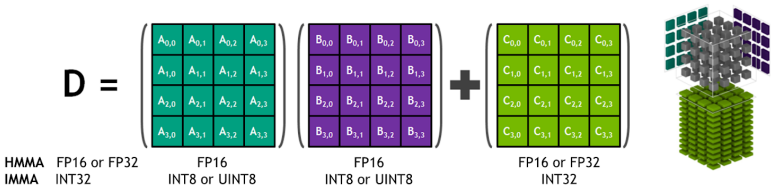


Figura 2.15: Multiplicación y acumulación de matrices en un Tensor Core. Extraída de [47].

servar en la Sección 2.4.2, las matrices cumplen un rol muy importante en los kernels de inteligencia artificial. Una particularidad destacable que introducen estas unidades, es el manejo de precisiones numéricas mixtas al momento de operar, por ejemplo, ofreciendo la posibilidad de realizar multiplicaciones con FP16[1] y acumulaciones en FP32, y permitiendo en algunos casos duplicar la cantidad de datos procesados por unidad de tiempo.

2.6.3. FPGA

Los FPGAs (Field Programmable Gate Arrays) son dispositivos que, a diferencia de los anteriores, presentan una arquitectura de hardware flexible formada por una matriz de elementos lógicos configurables, bloques dedicados (compuestos principalmente por memorias y multiplicadores), en conjunto con una estructura programable de enrutamiento que posibilita la interconexión entre estos bloques. Quizás una de las mayores ventajas que presentan es la capacidad de ser "reprogramadas"(re-diseñadas) luego de fabricadas, demostrando ser un tipo de dispositivo muy versátil para diferentes aplicaciones [39, 88].

Las FPGAs se encuentran, entonces, en un punto medio entre las arquitecturas de propósito general como son las CPUs y las GPUs y los ASICs, arquitecturas de hardware diseñados específicamente para una función. Si bien es posible que las FPGAs no alcancen un desempeño comparable al que potencialmente consiguen los ASICs en cuanto a consumo, velocidad y área, tienen la ventaja de que el costo de diseño, así como la puesta en producción de soluciones en pequeña escala, es considerablemente menor. Esto se suma a la flexibilidad que permite utilizar "software" para reconfigurar las características de este hardware (mediante la sintetización de nuevos diseños).

[1] Punto flotante con 16 bits, también conocido como media precisión o *half precision*.

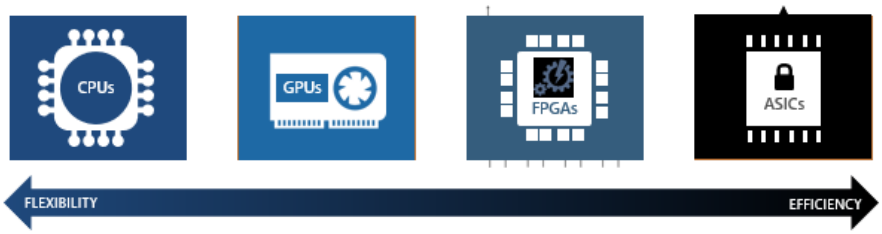


Figura 2.16: Comparación entre las distintas arquitecturas, desde un enfoque de flexibilidad y eficiencia. Extraído de [58].

En la Figura 2.16 se presenta, de forma gráfica, una comparación similar a la tratada en los apartados de las distintas plataformas de hardware con sus cualidades. A forma de resumen, la figura destaca la variabilidad existente a la hora de implementar una solución empleando estas tecnologías, evaluadas desde el punto de vista de la flexibilidad y eficiencia.

2.6.4. ASICs

Los ASICs o Application Specific Integrated Circuits, son circuitos diseñados específicamente para resolver una tarea en particular. Estos cuentan con la ventaja de resolver el problema con una gran eficiencia, obteniendo altas velocidades con un bajo consumo energético.

Una amplia gama de sistemas emplea soluciones que aprovechan este tipo de hardware. Los SoCs (System on Chip) presentes en teléfonos móviles y tablets integran componentes altamente especializados como unidades de procesamiento gráfico, procesadores de señales digitales e incluso aceleradores destinados a tareas de redes neuronales. Por ejemplo, el procesadores Apple A11 “Bionic”, integra un Image Signal Processor (ISP) llamado Neural Engine, que es un ASIC diseñado para ofrecer un rendimiento óptimo y una alta eficiencia energética en las tareas de identificación biométrica, como el FaceID y el TouchID [106]. Otro campo donde las soluciones basadas en ASICs son fundamentales es en aplicaciones automotrices, donde desempeñan funciones críticas como el control del despliegue del sistema de airbag, la gestión de la batería y el control del rendimiento del motor, entre otras tareas [23].

Aunque los ASICs representan soluciones sumamente optimizadas para una gran variedad de problemas, la implementación y desarrollo conlleva desafíos significativos en comparación con otras plataformas de hardware más flexi-

bles. Estos desafíos van desde el importante tiempo necesario para el estudio minucioso del problema que se quiere resolver de manera eficiente, hasta la dificultad de diseño, siendo necesario el manejo de bibliotecas y herramientas específicas de los fabricantes. Además, el costo de producción es un factor crítico a considerar, y se necesita un análisis exhaustivo de su rentabilidad, especialmente dado que la viabilidad económica de estos circuitos suele depender de su producción a gran escala [19].

2.6.5. Otras plataformas de hardware

Existen, además, otra gran cantidad de plataformas que emergieron o "re-emergieron" directamente para abordar problemas asociados a las redes neuronales. Un ejemplo bastante reciente es la inferencia de redes neuronales utilizando hardware y cómputo analógico. Este enfoque difiere fundamentalmente de las plataformas de hardware actuales, que en su mayoría utilizan una arquitectura de von Neumann y codifican información en representación binaria. La tecnología analógica permite codificar los pesos en el chip representando el flujo de datos para realizar una inferencia como el procesamiento de una señal eléctrica, lo que permite calcular "en memoria" y reducir drásticamente las transferencias de datos [53].

Otro ejemplo de plataforma de hardware que se han usado para la ejecución de modelos de redes neuronales es TinyML [124], donde P. Warden y D. Situnayake estudian la ejecución de modelos de redes neuronales en microcontroladores con restricciones extremas de consumo energético, con consumos en el rango de los mili-Watts y menores.

Capítulo 3

Revisión del estado del arte

En este capítulo se relevan los principales esfuerzos dedicados al cómputo eficiente de redes neuronales, así como el diseño de plataformas de hardware específicas que logren explotar características de este tipo de programas.

De manera general, los esfuerzos se pueden clasificar en dos tipos o niveles de optimización, aquellos centrados en el software, aplicables muchas veces sobre varias plataformas de hardware, y los otros enfocados en el desarrollo de nuevo hardware optimizado para estas tareas. En el caso de las optimizaciones por software se aplican técnicas como transformaciones, podas y reordenamientos en los datos de entrada y operaciones de la red neuronal para aumentar el paralelismo o reducir el número de operaciones costosas. En el nivel de hardware, se destacan diseños de plataformas específicas para ejecutar redes neuronales, considerando factores como memoria, caché, multiprocesadores y unidades dedicadas, con el objetivo de lograr un equilibrio óptimo entre capacidad de cómputo y consumo energético.

3.1. Optimizaciones a nivel de hardware

Las optimizaciones enfocadas en el hardware requieren de un diseño específico de las plataformas encargadas de ejecutar redes neuronales. Para ello es necesario un estudio minucioso, tanto de las plataformas existentes, como del flujo esperado de este tipo de rutinas, permitiendo evaluar, entender y diseñar conjuntamente las redes y el uso de aceleradores de hardware. A continuación se describen varios de los esfuerzos centrados en el tópico, incluyendo entre otros, implementaciones de grandes potencias en el área del desarrollo de hardware

como son NVIDIA e Intel. Destacan los esfuerzos por reducir el impacto de las transferencias de memoria, por ejemplo empleando representaciones con una cantidad reducida de bits, o incluyendo unidades de cómputo paralelo con memorias unificadas, adoptando enfoques de flujo sistólico de datos.

3.1.1. Cuantificación o cuantización

Una categoría bastante importante dentro de los dispositivos capaces de ejecutar redes neuronales, son los dispositivos *edge* o de borde. Una de sus cualidades más importantes es que presentan memoria y capacidad de cómputo sumamente limitada. Con la gran demanda del IoT (Internet of Things [56]), los dispositivos edge deben ser capaces de computar inferencias de modelos de redes neuronales en tiempo real, intentando reducir al mínimo la latencia. La cuantificación es una técnica que se enfoca en la reducción de la precisión numérica o tipo de datos empleados para representar los parámetros de un modelo de red neuronal (p.ej. pesos y bias) y realizar las operaciones. Por lo general, los modelos y sus pesos son representados y computados empleando `FP32` (punto flotante de simple precisión, 32 bits). Sin embargo, dependiendo de la tolerancia en los niveles de precisión numérica, los parámetros de la red pueden ser representados en `FP16` (punto flotante de media precisión, 16 bits), enteros de 8 bits o incluso por bits [57]. El uso de esta técnica da como resultado un modelo de mucho menor tamaño, que no sólo puede ser almacenado en memorias más pequeñas, sino que presenta, dependiendo de la plataforma de hardware en la que se ejecute, menor latencia y costo energético.

En general, para emplear representaciones numéricas de baja cantidad de bits como `INT8` (entero de 8 bits) a partir de una red neuronal entrenada y parametrizada con `FP32` (proceso llamado *post-training quantization* [97]) es necesario aplicar algún mecanismo de re-ajuste de los parámetros. Luego de cuantificar el modelo, se calibran los pesos y bias de la red (ahora representados en `INT8`) utilizando un pequeño subconjunto de datos para minimizar la pérdida de información en el pasaje de `FP32` a `INT8`. Esto ayuda a evitar grandes errores de predicción al momento de inferir con modelos cuantizados luego de estar entrenados. Existen también mecanismos de entrenamiento que tienen en cuenta el error de cuantificar (transformar los parámetros de `FP32` a `INT8`) en el cómputo de la función de pérdida. A esto se lo llama *quantization aware training* [99].

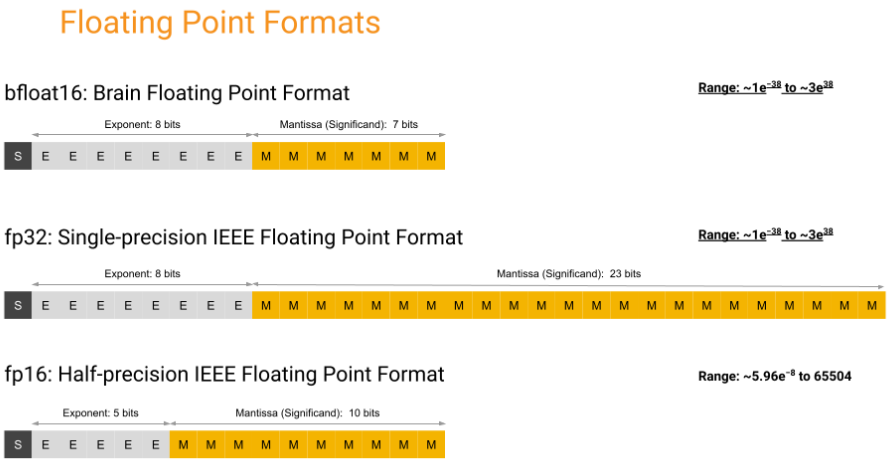


Figura 3.1: Comparación entre los formatos numéricos `FP32`, `BF16` y `FP16`. Muestra como el formato `BF16` permite representar números en el mismo rango que `FP32`, pero con menor precisión (menor cantidad de bit para la mantisa). Extraído de [46].

Los beneficios de la cuantificación, han impulsado también al desarrollo de nuevos formatos para la representación numérica de los modelos. Brain Floating-Point (`BF16`), por ejemplo, es un formato de 16-bits desarrollado por Google Brain [46], similar a `FP16`. La diferencia es que `BF16` designa una mayor cantidad de bits para el exponente, permitiendo representar un rango mucho mayor de números, designando menor cantidad de bits a la mantisa, es decir, para representar la parte fraccionaria (ver Figura 3.1). Si bien se tienen menos bits de precisión, `BF16` alcanza casi el mismo desempeño que `FP32` en términos de *accuracy* en tareas de IA [65]. Para que este tipo de optimizaciones funcione, las plataformas de hardware tienen que agregar unidades que manejen y computen estos formatos. Por ejemplo, los Tensor Cores de la generación Ampere de Nvidia, dan soporte a los formatos `BF16` y `FP16` [91], entre otros.

En trabajos recientes se han registrado notables resultados en cuanto a la ejecución de LLMs empleando únicamente los valores $\{-1, 0, 1\}$ para la representación de los parámetros del modelo. En [78] se propone BitNet b1.58, con la representación antes descrita, obteniendo resultados comparables a `FP16` y `BF16`, con una relación costo-beneficio más favorable en términos de latencia, throughput, consumo de memoria y energía.

Notar que esta técnica puede ser aplicada para casi cualquier modelo de redes neuronales, desencadenando en al menos una reducción de la memoria necesaria para representar el modelo, por lo que podría pensarse como una

optimización por software. Sin embargo, el mayor provecho de esto se obtiene cuando se cuenta con plataformas de hardware específicas que den soporte a precisiones reducidas. Por ese motivo los diseñadores de hardware han empezado a optar por diseñar los chips con operadores específicos de baja precisión. NVIDIA por ejemplo, recientemente lazó la arquitectura de GPU Blackwell [1], donde incorpora en el set de instrucciones operaciones en `FP4` (punto flotante de 4 bits).

3.1.2. Arquitecturas de hardware aceleradoras para DNNs

Hasta ahora, se analizaron algunos ejemplos de arquitecturas de hardware temporales (CPUs y GPUs) y arquitecturas espaciales (FPGAs). En esta sección se describen algunas plataformas de hardware, entre ellas ASICs, pensadas y optimizadas especialmente para redes neuronales. La mayoría de los esfuerzos se centran en el uso de memorias on-chip cercanas a las unidades de cómputo, así como optimizaciones en el ancho de banda de acceso a los niveles de memoria. Al mismo tiempo, también buscan explotar al máximo el paralelismo de datos. A continuación se presentan algunas plataformas destacadas.

TPU

Los Tensor Processing Units [63], o TPUs, son procesadores creados por Google que fueron diseñados para dotar a los servidores de dicha empresa de unidades de cómputo específico para IA. A diferencia de la mayoría de las plataformas de hardware antes evaluadas como las CPUs y GPUs, diseñadas para computar eficientemente con escalares y vectores de datos, esta ASIC de Google es capaz de manejar matrices de datos para computar eficientemente. Para ello agregaron específicamente una matriz o arreglo sistólico de unidades de cómputo llamada Matrix Multiply Unit (MXU). Esta se encarga de realizar multiplicaciones de matrices y acumular esos resultados de forma masiva, representando el corazón computacional del TPU. En la Figura 3.2 se resaltan las principales unidades que componen al TPU, destacando la MXU a la derecha, cuya entrada viene dada por las unidades Weight FIFO (para almacenar los pesos de la capa a computar) y la unidad Systolic Data Setup (con las activaciones), y cuya salida se impacta en Accumulators. Notar también la presencia de varias unidades específicas para el cómputo de activaciones, como

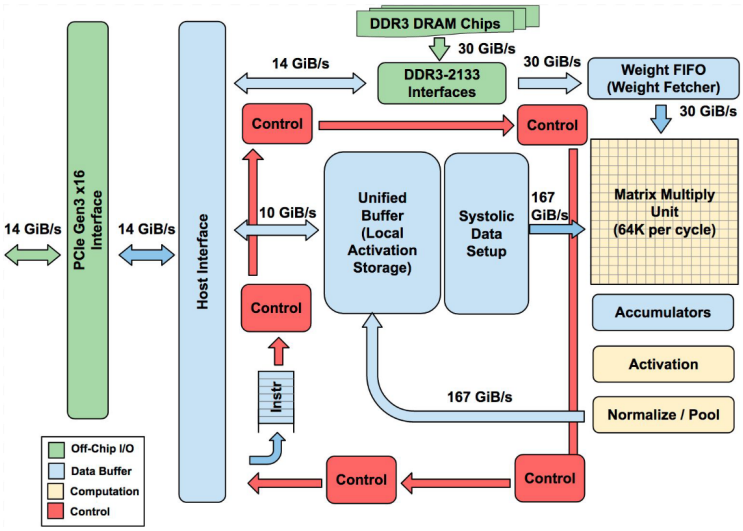


Figura 3.2: Diagrama de un TPU. Extraído de [63].

normalizaciones y capas de pooling. En la Figura 3.3 se muestra un resumen gráfico de la MXU. Básicamente es un arreglo bidimensional de 256×256 unidades MACs (en las últimas versiones de 128×128) que pueden realizar sumas y multiplicaciones de enteros de 8 bits, con o sin signo.

Con un diseño minimalista, la TPU destaca por optimizar la cantidad de transistores y, por ende, el consumo energético. En general, gran cantidad de transistores son empleados en CPUs para mejorar los casos de ejecución promedio, de menor importancia en cómputos específicos como las redes neuronales. Entre los mecanismos destacan, predicción de saltos, ejecución fuera de orden, multiprocesamiento, precarga probabilística de datos, manejo de multihilo, cambios de contexto, etc. El minimalismo es una virtud de los procesadores de dominio específico.

Actualmente los Cloud TPU, implementan soluciones escalables con varias versiones de estas unidades de procesamiento [113].

EdgeTPU

EdgeTPU [37], al igual que Cloud TPU, es una arquitectura diseñada por Google específicamente para tareas de inteligencia artificial. En este caso, la arquitectura de hardware está pensada para la implementación de modelos de ML en dispositivos edge, particularmente creada para poder llevar a cabo el proceso de inferencia (sin tener que pasar por la nube) pudiendo interactuar con sensores y otros dispositivos para obtener datos. Este acelerador optimiza

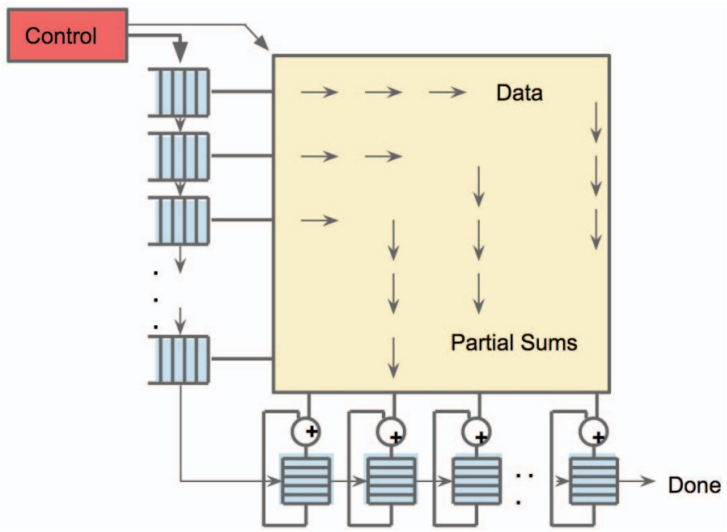


Figura 3.3: Flujo sistólico de datos de la MXU. Extraído de [63].

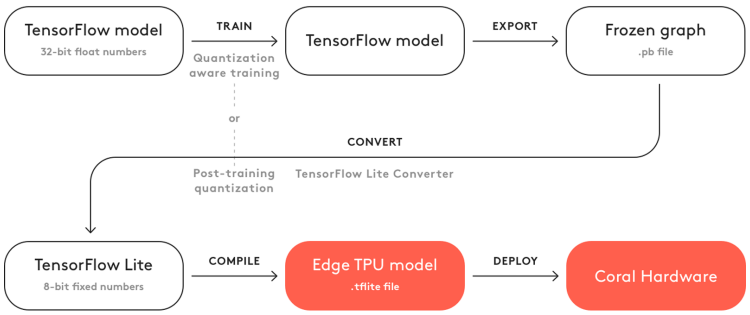


Figura 3.4: Flujo para crear un modelo de red neuronal ejecutable en un EdgeTPU. [120]

el proceso de inferencia haciendo uso de precisiones numéricas reducidas, operando con enteros de 8-bits (`INT8`) a un muy bajo costo energético, alcanzando un máximo de 4 TOPS empleando sólo 2W de potencia.

Los modelos de NNs que se pueden ejecutar en el EdgeTPU están limitados a un subconjunto de las operaciones que comúnmente se emplean en el diseño de NNs [120] y, en particular, solo pueden ser modelos cuantificados con TensorFlow Lite (framework que se describe en la siguiente sección). Un esquema gráfico del flujo que debe seguirse para implementar un modelo en el EdgeTPU se presenta en la Figura 3.4.

La unidad encargada de impulsar la capacidad de cómputo de este ASIC, es un arreglo bidimensional de PEs, donde cada PE consiste en múltiples pequeños cores con unidades MACs y una memoria local compartida entre estos cores.

Intel Xeon SP

Los procesadores Intel Xeon SP (Scalable Processors) de 4ta. Generación, integran unidades específicas como las Advanced Matrix Extensions (AMX) para acelerar tareas de *deep learning*, tanto en la fase de entrenamiento como en la inferencia. Las AMX son una extensión de las instrucciones AVX-512 que permite el procesamiento eficiente de operaciones matriciales, esenciales para el cómputo de redes neuronales.

Además de las AMX, los Xeon SP de 4ta generación incorporan varias otras tecnologías que contribuyen a su rendimiento superior en aplicaciones de inteligencia artificial. Por ejemplo, soportan `BF16` e incluyen el denominado **Intel Deep Learning Boost (DL Boost)**, instrucciones VNNI (Vector Neural Network Instructions) que optimizan el procesamiento de operaciones intensivas en cómputo, como las convoluciones en NNs. Esto reduce significativamente la latencia y mejora el rendimiento de la inferencia.

Estas y otras características hacen de los procesadores Intel Xeon SP una opción potente y eficiente para la computación de redes neuronales.

VPU

Los VPUs (Vision Processing Units) [62], son otro ejemplo de ASICs creadas para la aceleración de algoritmos de IA, enfocado particularmente en la visión artificial y el procesamiento de imágenes. La última versión de esta unidad, presentada por Intel, es el Movidius Myriad X VPU [61].

El Intel Neural Compute Stick 2 (NCS2), un dispositivo USB, equipado con el Movidius Myriad X VPU, está diseñado para llevar capacidades avanzadas de inferencia de IA a prácticamente cualquier entorno. Este dispositivo USB, conectado a un host, promete acelerar cómputos de IA a un bajo costo energético, siendo compatible con varios frameworks de inteligencia artificial. Junto con el Neural Compute Engine y 16 SHAVE (Streaming Hybrid Architecture Vector Engine) Cores, la arquitectura del Intel NCS2 está optimizada específicamente para tareas de visión por computadora, ofreciendo un rendimiento superior en este tipo de inferencias.

Otro aspecto a destacar es que si bien la memoria de los Intel NCS2 es algo limitada (500MB) acotando también los modelos que se pueden ejecutar de manera eficiente, existe la posibilidad de agrupar varios de estos dispositivos, escalando la ejecución para poder correr modelos de mayores dimensiones.

Nvidia **Jetson**

Los dispositivos Nvidia Jetson son una serie de sistemas en módulos (SoMs) y placas de desarrollo diseñadas específicamente para aplicaciones de inteligencia artificial y cómputo edge de alto rendimiento. Esta variedad de SoMs compactos, incluyen GPU, CPU, memoria, control de potencia e interfaces de alta velocidad, siendo ideales para tareas de visión por computadora, procesamiento de imágenes y otras aplicaciones de aprendizaje profundo en entornos embebidos con posibilidad de ser integrados directamente a productos finales. Desde su primer lanzamiento, las Jetson han evolucionado integrando nuevas y optimizadas características, por lo que existen múltiples versiones de estas plataformas. Igualmente, existe una serie de cualidades que las caracterizan como, por ejemplo, una GPU integrada, siguiendo un esquema de arquitectura CUDA con Tensor Cores, estos últimos presentes a partir de la familia Xavier NX Series. También, los Jetson Xavier introdujeron la **NVDLA**, una arquitectura de hardware dedicada a acelerar la inferencia de redes convolucionales a través de un diseño flexible, compuesto por diferentes bloques. Entre ellos destacan los Convolution Cores optimizados para la convolución, una serie de Data Processors optimizados para funciones de activación, capas pooling y de normalización, así como una memoria dedicada y hardware específico para redimensionar tensores.

Además, los Jetson presentan compatibilidad con varios frameworks de desarrollo de tareas de IA, como PyTorch, TensorFlow y Caffe, los cuales pueden ser complementados con el SDK Nvidia JetPack, que incluye, entre otros, TensorRT. Finalmente, estas plataformas están optimizadas para aplicaciones de visión por computadora.

Otras plataformas de hardware

Aparte de las propuestas de los principales fabricantes de hardware existen esfuerzos académicos en la misma dirección. Por ejemplo la arquitectura de hardware Eyeriss, desarrollada por Chen et al. [29] enfocada en la ejecución eficiente de CNNs, contemplando la eficiencia energética y niveles de latencia aceptables. Este acelerador se caracteriza también por su flexibilidad, siendo capaz de soportar redes convolucionales que presentan un elevado número de capas, millones de pesos, así como distintas configuraciones de filtros y canales. Además, se intenta explotar al máximo la reutilización de datos a través

del esquema de flujo Row Stationary DataFlow propuesto por algunos de los mismos autores [28].

Existen también otros esfuerzos dedicados al cómputo eficiente de NNs, en particular, a través del cómputo analógico. Con este enfoque, se busca explotar características físicas del hardware, representando, por ejemplo, los pesos de un modelo como magnitudes de corriente (físicas/analógicas), en lugar de señales digitales codificadas como un número en binario. De esta manera se implementan sumadores y multiplicadores empleando leyes físicas conocidas, como las leyes de nodos y mallas para circuitos de Kirchhoff [48]. La empresa Mythic [87] ha construido un acelerador de IA analógico llamado "Analog Matrix Processor" que, según Mythic, tiene un rendimiento de 25 TOPS, con un consumo de 4W. Además, en los últimos años IBM también ha invertido esfuerzos en el prototipado de un chip llamado Fusion que usa memoria de cambio de fase para almacenar sus datos directamente en el chip [59, 110].

3.1.3. Dataflows

Como se pudo observar en los esfuerzos anteriores, una gran parte de las estrategias de optimización por hardware consistieron en la aplicación de paralelismo y la reutilización de datos a través de memorias intermedias, cercanas a las unidades de cómputo. Los principales datos involucrados en la ejecución de NNs son: los pesos, las activaciones o entradas a una capa y las sumas parciales para computar la salida. Por lo tanto, estas memorias intermedias y los mecanismos de pasaje de datos entre PEs pueden tener distintos objetivos en cuanto al dato que se almacena. Chen et al. [28] desarrollaron una taxonomía de los diferentes esquemas de flujo de datos (*dataflows*) presentes en las plataformas empleadas para la ejecución de redes neuronales. En primera instancia, clasificaron los esquemas de flujo en tres categorías: *Weight Stationary* (WS), *Output Stationary* (OS) y *No Local Reuse* (NLR). A continuación se describen brevemente cada una.

- *Weight Stationary*: en esta familia la idea principal es minimizar el consumo energético asociado al acceso de los pesos, leyéndolos una vez de memoria y almacenándolos en los registros de cada unidad. Son plataformas que intentan maximizar la cantidad de operaciones que involucren estos parámetros, leyendo continuamente datos de entrada y computando, a través del arreglo de unidades, la suma ponderada. El procesador

TPU presentado por Google [63] es, posiblemente, el ejemplo más conocido con este esquema.

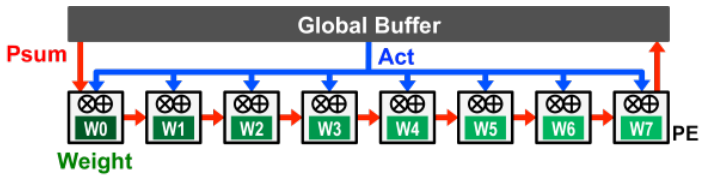


Figura 3.5: Weight Stationary dataflow. Extraído de [28].

- *Output Stationary* (OS): a diferencia del anterior, busca optimizar el costo de acceso a las sumas parciales. Existen varias implementaciones de este flujo.

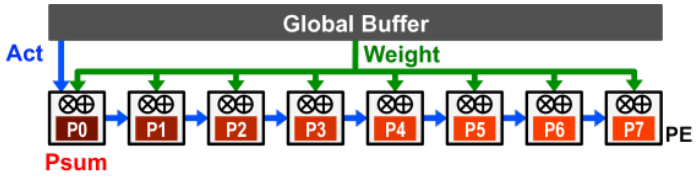


Figura 3.6: Output Stationary dataflow. Extraído de [28]

- *No Local Reuse* (NLR): es un esquema de flujo que logra reducir el tamaño físico de la arquitectura al eliminar la unidad de memoria, accediendo únicamente a través de la jerarquía.

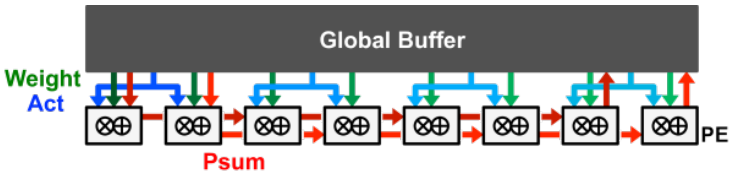


Figura 3.7: No Local Reuse dataflow. Extraído de [28]

3.2. Optimizaciones a nivel de software

La idea principal de los esfuerzos que buscan realizar optimizaciones por software consiste en entender cómo funcionan los distintos *kernels* que intervienen a la hora de ejecutar diferentes modelos de NNs. Se busca realizar transformaciones a dichos *kernels* que permitan obtener el mismo resultado (o un resultado equivalente, en algún sentido, y que sea aceptable) pero reduciendo la cantidad de operaciones costosas o ciclos. Este tipo de técnicas

surgen con foco en el aprovechamiento y uso óptimo de los recursos computacionales que las plataformas, principalmente las arquitecturas de hardware temporales como las CPUs y GPUs, ofrecen, buscando entonces accesos eficientes a las jerarquías de memorias y una buena distribución de las cargas entre las unidades de cómputo disponibles. Entre otras optimizaciones destacan transformaciones, podas y reordenamientos tanto al conjunto de datos de entrada como a las operaciones que intervienen en las etapas de cómputo de la red neuronal. Esto permite ajustar los cómputos de las redes neuronales, por ejemplo, logrando mayores niveles de paralelismo o reduciendo la cantidad de operaciones costosas a través de podas sobre la red.

3.2.1. Transformaciones de los *kernels*

Las multiplicaciones y acumulaciones (MAC) son las operaciones fundamentales a la hora de computar una red neuronal. En la Sección 2.4.2 se discutieron algunas de las capas más utilizadas en las redes neuronales, junto a los *kernels* asociados a las mismas, evidenciando la gran cantidad operaciones MAC que se deben realizar durante su ejecución. En particular, el orden de estas operaciones, la cantidad de datos, así como la agrupación de los mismos pueden ser modificados buscando un cómputo eficiente, medido desde distintos parámetros.

A continuación se presentan técnicas para la optimización de la ejecución de las NNs, en algunos casos, a través de la reducción de la cantidad de multiplicaciones con un aumento de la cantidad de sumas y accesos a memoria, y en otros casos aprovechando al máximo la reutilización de datos en memorias rápidas.

En primer lugar, una técnica interesante es la que permite el mapeo de una convolución en una multiplicación de matrices. El algoritmo *im2col* permite dada una matriz y unas dimensiones (generalmente las del filtro), rearmar una nueva matriz que tiene por columnas los sub-bloques de las dimensiones dadas, expresados como vectores. Luego, convirtiendo el filtro (en caso 2D) en un vector, la convolución se puede expresar como una multiplicación matriz-vector. En la izquierda de la Figura 3.8 se puede observar la convolución 2D normal, y a la derecha el mapeo a una multiplicación de matrices.

La ventaja de este enfoque radica, como se menciona anteriormente, en que las operaciones de multiplicación de matrices densas grandes (*kernel* conocido

Figura 3.8: Mapeo de Convolución 2D a Multiplicación de matrices empleando *im2col*. Extraído de [85].

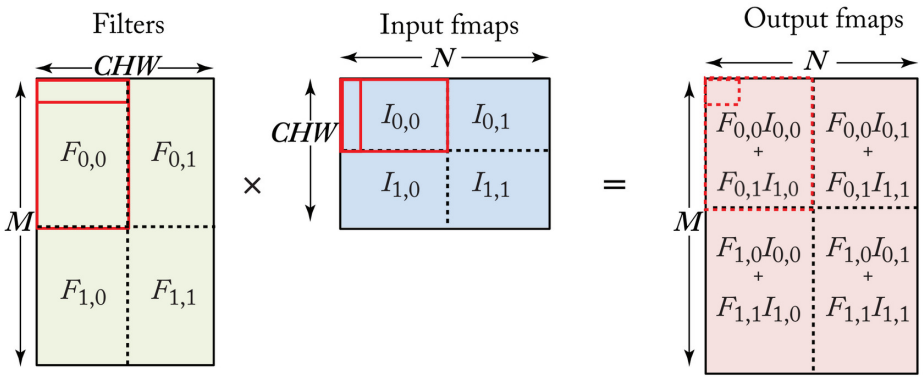


Figura 3.9: Ilustración de una multiplicación de matrices utilizando *tiling*. La línea punteada roja en la matriz de salida indica que es un producto parcial. Adaptada de [114].

como GEMM [6]) son rutinas que han sido y siguen siendo sumamente estudiadas y optimizadas para una gran cantidad de plataformas de hardware. La principal desventaja que presenta esta estrategia, es la redundancia de datos en al momento de aplicar *im2col* a la matriz de entrada, lo que puede resultar en patrones de acceso no uniformes y un consumo mayor de memoria, que puede no ser útil en ciertos contextos.

En el contexto de las multiplicaciones de matrices, también se destaca una técnica llamada *tiling*, que la gran mayoría de las bibliotecas ya implementan, y se basa en el aprovechamiento de las jerarquías de memoria. Básicamente, a través de una modificación en el orden de recorrido de las matrices al momento de computar la multiplicación, se particionan los datos de entrada y los pesos en bloques o *tiles* (rectangulares) de un tamaño que se pueda almacenar en las memorias más rápidas y cercanas a las unidades de cómputo, buscando la reutilización máxima de estos datos. La Figura 3.9 ilustra esto.

El algoritmo de *Strassen* [112] (o sus evoluciones) permite computar un producto de matrices cuadradas utilizando una menor cantidad de multipli-

caciones, añadiendo como costo adicional el almacenamiento de matrices intermedias. Por ejemplo, una multiplicación de matrices de dimensiones 2×2 aplicando el algoritmo de Strassen se puede resolver con 7 multiplicaciones en lugar de 8 (producto de matrices clásico), sin modificar el resultado final. Para ello se introducen 7 resultados intermedios, que permiten computar las cuatro entradas de la matriz de salida. En el caso general para matrices $n \times n$, donde la multiplicación normal de matrices tiene una complejidad de $O(n^3)$, Strassen permite reducirla a $O(n^{log_2(7)}) \sim O(n^{2.807})$. En estos casos, la estrategia se basa en Divide and Conquer (D&C), particionando las matrices que intervienen en la multiplicación en 4 sub-matrices y aplicando recursivamente el algoritmo hasta llegar a un caso base. A continuación se presenta dicho algoritmo, donde A y B son las matrices a multiplicar y el resultado se almacena en la matriz C. Este se puede dividir en tres fases principales: División, Resultados intermedios y Combinación.

1. **División**: Se dividen las matrices A y B en cuatro submatrices más pequeñas de tamaño $\frac{n}{2} \times \frac{n}{2}$.
2. **Resultados intermedios**: Se calculan siete productos intermedios utilizando estas submatrices:

$$
\begin{aligned}
P_1 &= A_{11} \cdot (B_{12} - B_{22}) \\
P_2 &= (A_{11} + A_{12}) \cdot B_{22} \\
P_3 &= (A_{21} + A_{22}) \cdot B_{11} \\
P_4 &= A_{22} \cdot (B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
P_6 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\
P_7 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12})
\end{aligned}
$$

3. **Combinación**: Se utilizan estos productos intermedios para calcular las submatrices del resultado final:

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

Si bien la cantidad de sumas a realizar aumenta, las multiplicaciones son generalmente más costosas, tanto desde el punto de vista de hardware, ocupando mayor área en el chip, como del energético (hasta 30 veces más que la suma, dependiendo del tipo de datos [53]), por lo que reducir la cantidad de multiplicaciones puede llegar a ser un gran beneficio dependiendo de las restricciones con las que se cuente. En FPGAs, por ejemplo, se podrían acomodar más unidades de suma que multiplicadores. Cong y Xiao [32] utilizaron este algoritmo para optimizar la complejidad de las convoluciones en una CNN, realizando el mapeo de una convolución a una multiplicación de matrices.

Existen también otros trabajos interesantes, por ejemplo, aquellos que emplean el algoritmo de filtrado mínimo de Winograd [125], y también otros que aplican técnicas como Fast Fourier Transform (FFT). Lavin y Grey [71] fueron de los primeros autores en aplicar el algoritmo de Winograd para NNs. En particular, buscaron minimizar la complejidad computacional de redes convolucionales con tamaños de filtros y batch pequeños. La idea detrás de esta estrategia, es buscar explotar las superposiciones que se presentan en el cómputo de una convolución con el método directo al momento de trasladar el filtro a través de los datos de entrada.

FFT es un algoritmo comúnmente utilizado para el procesamiento de señales. En el caso de las redes neuronales, también es otra opción para la reducción de la complejidad computacional de la convolución. Autores como Mathieu et al. [82] fueron precursores en incorporar esta técnica al campo de las NNs, posteriormente optimizada y mejorada en [122]. El proceso consiste en transformar los datos de entrada y los filtros al dominio de frecuencias, sustituyendo la convolución por una multiplicación de matrices y, por último, aplicar la transformada inversa de Fourier para obtener el resultado en el dominio del tiempo. Si bien las dimensiones una vez transformadas las matrices al dominio de frecuencias se mantienen, las entradas son representadas con complejos, por lo tanto se obtiene una reducción en el cómputo pero no en memoria ni ancho de banda.

Algunas de estas técnicas son implementadas en bibliotecas como cuDNN [7, 30] de Nvidia. Existen otras implementaciones, por ejemplo, oneDNN [92], biblioteca que se encuentra disponible en oneAPI [93] de Intel, que nuclea varias familias de optimizaciones para hardware heterogéneo.

3.2.2. Poda de parámetros de la red neuronal

El *pruning* o poda, es un concepto que se enfoca en la remoción o eliminación de parámetros de la NN, ya que una gran mayoría de modelos están sobre-parametrizados incurriendo en una gran redundancia. Dentro del campo de las NNs, la aplicación de podas sobre los modelos no es nueva. Muchos de los primeros trabajos se enfocaban en la eliminación o truncamiento de pesos de menor magnitud o cercanos a cero. La intuición detrás de este enfoque es que dichos pesos tienen menor impacto sobre la salida final del modelo. En 1990, LeCun et al. [72] ya proponían variantes para la eliminación de los pesos de las redes. En lugar de emplear la magnitud de los pesos como métrica para eliminar conexiones de la red, proponían emplear la segunda derivada de la función objetivo con respecto a los parámetros. Este fue uno de los trabajos que sentó las bases para la aplicación de *pruning* en el contexto de NNs, destacando las importantes ventajas de estas técnicas, como son mejoras en la generalización de los modelos y mejora en la velocidad de entrenamiento e inferencia.

Al día de hoy las ventajas son aún más destacables, principalmente debido a que tener modelos reducidos permite, entre otras cosas, un menor manejo de memoria (que como se mencionó es el principal cuello de botella en este tipo de cómputo) así como la exportación de los modelos a plataformas edge y dispositivos IoT (donde la capacidad de almacenamiento y memoria son limitadas). Por otro lado, también se destaca la reducción de cómputo, ya que cuanto menor es la cantidad de pesos y parámetros que procesar en la red, menor es la cantidad de MACs que se tendrían que realizar en el proceso de inferencia. Cabe destacar que, en relación a este último punto, varios investigadores han concluido que una reducción en la cantidad de MACs generalmente no se ve reflejada linealmente en métricas importantes al momento de implementar las redes en sistemas reales como, por ejemplo, la eficiencia energética o la latencia [86, 129]. En este sentido, se destaca que los resultados van a depender fuertemente de la plataforma de hardware.

En los últimos años estas técnicas han sido estudiadas con mayor detenimiento, como es el caso de [50], donde se propone un mecanismo completo para podar las NNs. El proceso consta de tres etapas: en la primera se entrena la red densa ("sobre-parametrizada") para aprender cuáles conexiones son importantes, seguidamente se podan las conexiones de menor importancia, eliminando aquellos pesos que estén por debajo de cierto umbral y, por último,

se re-entrena el modelo con las conexiones restantes, a modo de refinamiento o *fine-tuning*. Entre los experimentos, se ejecutan de manera iterativa estas últimas dos etapas, buscando podar más aún el modelo, funcionando como una búsqueda de la arquitectura óptima. Los autores muestran cómo se pueden obtener incluso mejores resultados de precisión que con el modelo original.

Hasta ahora, los ejemplos de *pruning* mostrados trabajan con podas individuales de pesos o conexiones entre neuronas, llevando dichos pesos a cero, que en el proceso de predicción se traducen en multiplicaciones por cero. A este tipo de poda, se lo conoce como *Unstructured Pruning*. Estas podas, en general, dan como resultado estructuras dispersas (es decir, con una cantidad importante de valores nulos), que pueden ser explotadas por ciertas plataformas de hardware en forma de no-ops (no operations). El otro tipo de podas, *Structured Pruning* se centra en eliminar filtros enteros e incluso canales, directamente impactando en la dimensionalidad de la red. Desde el punto de vista de la calidad de la red, esto puede ser perjudicial debido a que junto a los pesos no importantes también se eliminan algunos que si tienen impacto sobre la salida, degradando la salida de la red. Desde el punto de vista del desempeño, estas podas producen nuevos modelos que ejecutan mucho más rápido en la mayoría de las plataformas de hardware, sin necesidad de aplicar mecanismos para explotar los valores nulos. Por lo tanto, ambos mecanismos de poda tienen ventajas y desventajas, recayendo en los equipos de desarrollo evaluar cuál de las podas aplicar.

3.2.3. Frameworks

Esta sección describe brevemente algunos de los principales *frameworks* y APIs específicas de desarrollo de NNs, que permiten realizar optimizaciones sobre modelos existentes aplicando varias de las técnicas discutidas en las secciones anteriores. En general, varios de estos frameworks son provistos por las mismas empresas encargadas del desarrollo de las plataformas de hardware. Algunos de ellos funcionan de forma similar a los compiladores, permitiendo aplicar optimizaciones sobre la carga de trabajo asociada a la red neuronal de manera sencilla.

Entre los principales *frameworks* para el desarrollo de redes neuronales se encuentran:

- TensorFlow: es una biblioteca de código abierto desarrollada por Goo-

gle para el aprendizaje automático y la IA. Ofrece una amplia gama de herramientas para construir y entrenar redes neuronales, incluyendo TensorFlow Lite para dispositivos móviles.

- Keras: es una API de alto nivel para redes neuronales que se ejecuta sobre TensorFlow. Está diseñada para ser fácil de usar, modular y extensible, permitiendo a los desarrolladores construir y prototipar rápidamente modelos de aprendizaje profundo.
- PyTorch: es otro popular *framework* de código abierto creado por Meta para el desarrollo de redes neuronales. Es conocido por su facilidad de uso y flexibilidad, permitiendo a los investigadores y desarrolladores construir modelos complejos con una sintaxis intuitiva, además de proveer directivas para su uso con GPUs.
- Caffe: Originalmente desarrollado por la Universidad de Berkeley, es un *framework* de aprendizaje profundo optimizado para la velocidad y la modularidad. Aunque no es tan flexible como TensorFlow o PyTorch, sigue siendo popular en aplicaciones de visión por computadora debido a su eficiencia [25].
- ONNX (Open Neural Network Exchange): ONNX es un ecosistema abierto para la interoperabilidad de modelos de aprendizaje automático. Permite a los desarrolladores mover modelos entre diferentes frameworks como TensorFlow y PyTorch, facilitando la optimización y el despliegue en diversas plataformas de hardware.

Algunos de estos *frameworks* no solo permiten la construcción y entrenamiento de NNs, sino que también ofrecen herramientas y bibliotecas para la optimización de modelos, asegurando que las NNs puedan ser ejecutadas de manera eficiente en diversas plataformas de hardware.

A continuación se detallan otros *frameworks* específicos para el desarrollo eficiente de modelos de NNs, en algunos casos necesarios para la implementación sobre plataformas de hardware específicas.

OpenVINO

OpenVINO [60] es una herramienta que, además de ser necesaria para ejecutar proyectos de IA en algunos dispositivos de Intel, permite realizarles optimizaciones centradas en el proceso de inferencia. Es compatible con varios frameworks de desarrollo de redes neuronales, como por ejemplo TensorFlow y

Caffe. Para poder ejecutar un modelo diseñado en alguno de estos frameworks, es necesario convertir el modelo entrenado a un formato intermedio antes de poder ejecutar el optimizador de OpenVINO y posteriormente, con el Inference Engine (IE) de OpenVINO ejecutar la red en el dispositivo.

TensorFlow Lite (TFL)

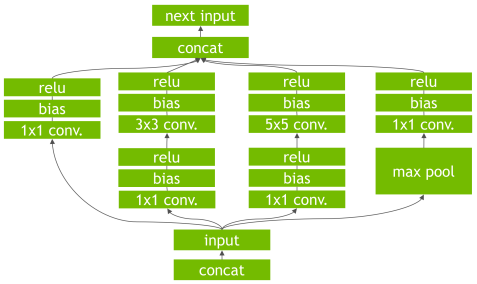
TensorFlow Lite [119] es un framework diseñado para implementar modelos de aprendizaje automático, especialmente centrado en la etapa de inferencia sobre dispositivos móviles y edge.

Una de las ventajas que presenta TFL con respecto a Keras/TF se basa en el formato de almacenamiento de sus modelos. Mientras que Keras y TF utilizan el formato Protocol Buffer [98] (protobuf -.pb-), TFL emplea el formato FlatBuffer [41] que, a diferencia del anterior, permite acceder a los datos serializados sin la necesidad de desempaquetarlos o analizar la estructura. Entre otras optimizaciones disponibles, se destaca la posibilidad de realizar cuantificaciones a los modelos empleando `INT8`, derivando en un uso mucho menor de memoria.

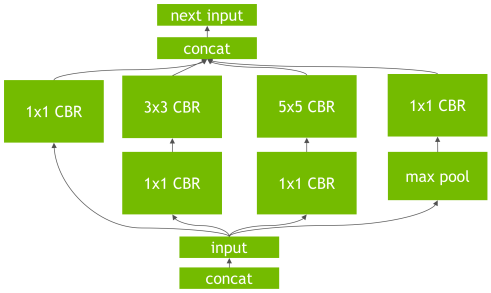
TensorRT

TensorRT [9] es un kit de desarrollo (SDK) provisto por NVIDIA que permite optimizar el proceso de inferencia en plataformas con GPUs. Además de contar con una basta cantidad de *kernels* optimizados e implementados en CUDA, el framework emplea varias de las técnicas introducidas en las secciones anteriores, entre ellas:

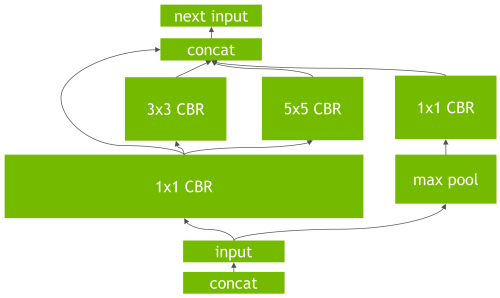
- Cuantificación: provee herramientas para cuantificar los parámetros de la red logrando emplear la precisión más baja posible sin perder significativamente precisión mediante técnicas como la calibración.
- Fusión de capas: TensorRT combina múltiples capas de la NN en una sola operación siempre que sea posible. Como se muestra en la Figura 3.10, esto reduce la cantidad de operaciones individuales y minimiza el sobrecoste de las operaciones de entrada/salida entre capas.
- *Kernel Tiling* y *Autotuning*: selección y ajuste de kernels optimizados para el hardware específico. TensorRT elige los mejores kernels y configura parámetros específicos para maximizar el rendimiento en la arquitectura de GPU utilizada.



(a) Red sin optimizar



(b) Red optimizada con TensorRT fusionando verticalmente



(c) Red optimizada con TensorRT con fusiones horizontales

Figura 3.10: Ejemplo de optimizaciones realizadas por TensorRT en un red neuronal por fusión de nodos y capas. Extraídas de [15].

- Optimización de datos y tensores: incluye la reorganización de datos y tensores en memoria para mejorar la eficiencia en el acceso y el ancho de banda.
- *Loop unrolling* y eliminación de capas inactivas: eliminar capas o nodos que no contribuyen al resultado final y desenrollar bucles para mejorar el rendimiento.

Varias de estas técnicas están asociadas a optimizaciones de hardware, evidenciando el desarrollo y diseño en conjunto de las plataformas de hardware, en este caso GPUs, junto con el de las redes neuronales.

La herramienta por linea de comandos `trtexec` [4] permite la creación de modelos especialmente optimizados (también llamados *engines*) para la plataforma NVIDIA donde se va a ejecutar el modelo.

3.2.4. Network Architecture Search (NAS) y otros algoritmos

Network Architecture Search (NAS) es una familia de algoritmos y soluciones que automatizan la búsqueda de nuevas arquitecturas de NNs en lugar de depender de su diseño manual por expertos. Esto ha tenido un impacto significativo en los últimos años y se vuelve especialmente relevante en situaciones en las que las dimensiones de las redes y la complejidad de los problemas crecen de manera exponencial. Por lo tanto, resulta una idea razonable desarrollar algoritmos que permitan explorar grandes espacios de búsqueda conformados por arquitecturas de NNs de forma automática.

Las técnicas de NAS, se pueden clasificar según el mecanismo que se utilice para implementar la búsqueda del óptimo. Las principales categorías son:

- Reinforcment Learning NAS
- Evolutionary NAS
- Bayesian NAS
- Differentiable NAS

A continuación se ahonda en cada una de estas categorías, describiendo los trabajos más relevantes, y discutiendo las ventajas y desventajas que cada técnica posee.

Reinforcement Learning (RL) NAS

Posiblemente uno de los primeros trabajos que emplean RL es el de Barret Zoph y Quoc V. Le [131], donde entrenan una red recurrente llamada *controller* utilizando RL, para que sea capaz de generar descripciones de redes neuronales intentando maximizar el accuracy para una tarea dada. Su red ConvNet producida por la RNN, obtiene mejores resultados que las redes diseñadas por expertos en el momento para el dataset CIFAR10.

Tan et al. [116] abordan el problema de encontrar modelos de redes neuronales lo suficientemente eficientes para las distintas plataformas de hardware disponibles buscando un balance entre la precisión y la latencia. Los autores proponen una solución utilizando aprendizaje por refuerzo para encontrar el frente de Pareto óptimo para este problema de búsqueda multi-objetivo, empleando mediciones empíricas de latencia para regularizar o guiar de alguna manera la búsqueda. El resultado de esta investigación es MnasNet, una red

convolucional especialmente optimizada para dispositivos móviles, 1,8 veces más rápida que MobileNet con ImageNet en un teléfono Google Pixel.

Evolutionary NAS

Real et al. [102] proponen evolucionar modelos para la clasificación de imágenes. Los autores emplean algoritmos y operaciones simples de evolución, como son la selección, cruzamiento y mutación, donde cada individuo es un modelo de red neuronal "entrenado", con el accuracy como función de fitness. El enfoque presenta la desventaja de requerir un alto costo computacional para la búsqueda.

En [101] se presenta una evolución con edad (*aging evolution* o *regularized evolution*), donde se propone asociar cada genotipo con una edad, y donde persisten los individuos más jóvenes. En particular, utilizan esta idea para obtener una nueva arquitectura que llaman AmoebaNet-A, compitiendo con los mejores clasificadores para ImageNet al momento (84.5 T1 / 97.0 T5). No se toman en cuenta optimizaciones que permitan reducir la latencia, o métricas por el estilo, para plataformas particulares. La arquitectura que obtienen es de tamaño y cantidad de parámetros similares a otros modelos obtenidos por NAS.

Differentiable NAS

En lugar de trabajar en un espacio discreto que involucra muchas arquitecturas y configuraciones distintas, en [75] se propone una relajación del problema a un espacio continuo, para que la función que guía la búsqueda de la arquitectura sea diferenciable, y así aplicar el descenso por gradiente. Este enfoque que llamaron DARTS, permite realizar la búsqueda de la arquitectura final de la red optimizando conjuntamente sus parámetros (p.ej. los pesos de una capa convolucional), órdenes de magnitud más rápido que los algoritmos NAS no diferenciables.

Si bien la búsqueda de arquitecturas sobre espacios diferenciables supone una importante mejora en términos de horas de GPU, estas técnicas hacen un uso intensivo de memoria. Para mitigar este problema, se hace uso de algún *proxy* o intermediario, por ejemplo resolviendo la búsqueda de una arquitectura para un dataset más simple, enfocando la búsqueda en una porción de la red, o entrenando por pocas épocas. Cai et al. [26] plantean el esquema ProxylessNAS

que permite resolver la búsqueda directamente sobre el problema en cuestión sin emplear ningún tipo de *proxy* para datasets de grandes dimensiones. Realizando podas a partir de una red hiper-parametrizada que representa todos los candidatos posibles, y a través del entrenamiento, buscan detectar qué caminos son redundantes. Incorporan también métricas y objetivos enfocados al hardware, como la latencia.

En [126] se propone una familia de modelos convolucionales, FBNets, descubiertas a través de DNAS sobre un espacio de búsqueda inspirado en MobileNetV2, utilizando convoluciones *depthwise* e *inverted residual blocks*. En particular, se incorporan medidas de latencia en la función de pérdida a minimizar (en este caso usando dos dispositivos móviles, un iPhone X y un Samsung S8) y asumiendo cómputos puramente secuenciales, permitiendo estimar de manera mucho más simple la latencia total como la suma de la latencia de las capas que componen a la arquitectura.

3.2.5. NetAdapt

Existen otros esfuerzos por encontrar arquitecturas óptimas que no son directamente NAS (en el sentido de que no crean una arquitectura de red neuronal nueva), y cuyo principal foco está en la eficiencia y el costo de computar dichas redes. Un ejemplo es NetAdpat [127], desarrollado por Yang et al., que aplicando podas estructuradas (eliminando filtros) a una arquitectura ya diseñada y entrenada, y a través de la comparación de distintas configuraciones de dicha red, busca una arquitectura lo suficientemente buena según cierta métrica para un dispositivo particular.

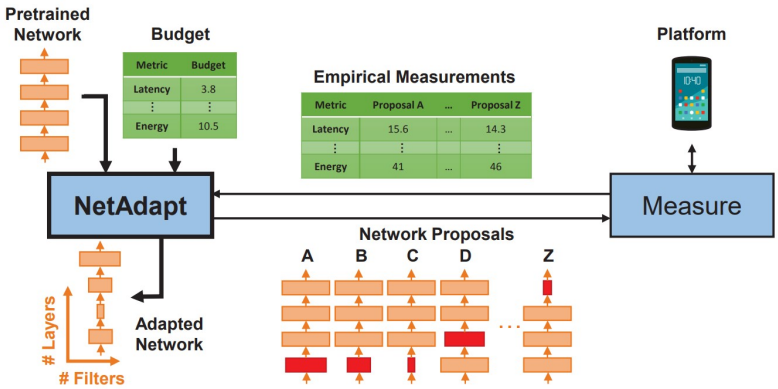


Figura 3.11: Proceso iterativo realizado por NetAdapt, para optimizar el modelo de red neuronal. Extraído de [127].

En otras palabras, NetAdapt iterativamente propone modelos simplificados de la red, que en cada paso del algoritmo son evaluados en términos de alguna métrica que se quiera optimizar (p.ej. latencia) y precisión, eligiendo el modelo que presente una mejor relación entre precisión y las métricas evaluadas. En particular, esta herramienta permite realizar optimizaciones sobre NNs desde un enfoque de eficiencia energética. Gráficamente este proceso se puede ver en la Figura 3.11.

A diferencia de otros métodos, NetAdapt basa sus optimizaciones en medidas empíricas, y no en métricas indirectas del desempeño como pueden ser la cantidad de MACs o la cantidad de pesos. Para ello, computa reiteradas veces sobre la plataforma de hardware objetivo distintas configuraciones de las capas de la red y mide, por ejemplo, la latencia de cada ejecución. Posteriormente estos resultados son almacenados en una tabla de búsqueda o Look-Up Table (LUT). Durante el proceso de optimización de la red neuronal, NetAdapt recorre las distintas configuraciones de filtros para dichas capas hasta encontrar una configuración cuya latencia cumpla con cierto criterio de reducción o restricción. En el algoritmo original esta búsqueda es realizada de forma lineal, comenzando con la cantidad actual de filtros (en un principio la configuración máxima) reduciendo y midiendo hasta alcanzar la restricción.

Una vez alcanzada la restricción para las distintas configuraciones (una por cada bloque), los modelos son entrenados (*fine-tuning*) y evaluados con el objetivo de obtener el mejor modelo en dicha iteración, comparando el ratio de precisión y latencia. Notar que podría pasar que la restricción nunca sea alcanzada para dicho bloque, en cuyo caso todas las configuraciones hasta el tamaño mínimo son recorridas. En casos como AlexNet, que tiene capas de 4096 filtros, esto puede resultar un gran inconveniente.

Debido a que la elección del modelo optimizado se realiza mediante medidas empíricas, esta herramienta permite realizar optimizaciones a modelos de NNs sin la necesidad de conocer al detalle la disposición, ni los recursos de la plataforma a la cual está dirigida, siendo útil en casos donde los fabricantes son poco explícitos en cuanto al diseño de la arquitectura de hardware o acelerador.

3.2.6. Optimizaciones en la búsqueda de redes óptimas

Existen varios esfuerzos por optimizar el costo de la búsqueda involucrada en los NAS o paradigmas similares, ya que son varios los puntos que hacen sumamente costosa esta tarea. En particular, se destaca la necesidad de realizar entrenamientos constantes a los individuos o posibles candidatos a ser la red neuronal buscada.

Cuando el objetivo de la búsqueda involucra también la optimización de alguna métrica además de la precisión, existen varias estrategias para mejorar esta parte del proceso. Por ejemplo, para evitar el cómputo de la latencia o evaluar continuamente el dispositivo para el cual se quiere el modelo, varios autores plantean la posibilidad de estimar dicho valor. Cuando los espacios de búsqueda son *chain-styled* (no hay capas recurrentes, ni saltos complejos, como conexiones residuales) estimar la latencia es relativamente simple, o incluso se puede utilizar una LUT, tal como se hace en NetAdapt, ProxylessNAS o FBNet. Estimar la latencia en lugar de medirla continuamente para de miles de inferencias reduce significativamente el tiempo de ejecución. Además, el tiempo necesario para la configuración y comunicación, especialmente en dispositivos edge como el EdgeTPU, puede ser considerable, ya que el pipeline para ejecutar un modelo (transformar los modelos a representaciones intermedias o aplicar cuantificaciones) puede derivar en un gran sobrecoste. Algunos autores también indican que computar el promedio de varias mediciones de latencia es muy costoso desde el punto de vista energético, dado que se invierte en cómputo solo para evaluar y no para inferir. Esto se puede notar especialmente en grandes servidores de datos. En [73], se aborda este problema y se implementa HELP, un predictor de latencia adaptativo empleando meta-learning. En este contexto, meta-learning, refiere a la noción de aprender a aprender. Además de obtener un modelo entrenado, se aprende a estimar la latencia para un dispositivo no evaluado (empleando una representación -hardware *embeddings*- para tratarlo como una caja negra sin tener que evaluarlo continuamente).

En otra categoría de optimizaciones, Pham et al. [96] presentan ENAS, un enfoque eficiente para la búsqueda de arquitecturas de NNs, en particular, compartiendo los pesos y los parámetros entre las arquitecturas generadas a partir de un super-grafo. El *controller*, encargado de la búsqueda de arquitecturas óptimas, se centra en un DAG (*directed acyclic graph*), donde los hijos son subgrafos de este, y los nodos representan cómputos locales, que conecta-

dos a otros nodos forman capas. Con este enfoque logran reducir en 1000× la cantidad de horas de GPU necesarias para entrenar modelos independientes, en comparación con [131].

Como evolución de NetAdapt, discutido anteriormente, se presentó NetAdaptV2 [128], que incorpora alguna de las ideas antes mencionadas, como el manejo de una super-red en lugar de redes disjuntas.

En [74], Li et al. evalúan métricas como energía y latencia para un conjunto de seis plataformas de hardware (desde edge comerciales, hasta ASICs como Eyeriss), sobre el espacio de arquitecturas de redes neuronales de FBNet y NAS-Bench-201 [35]. Como resultado, presentan una herramienta de benchmarking y búsqueda simple de nuevas arquitecturas optimizadas con cierto balance entre precisión y costo computacional, para plataformas de hardware específicas.

Mellor et al. proponen NAS-WOT [84], otra técnica para optimizar el tiempo de búsqueda en los algoritmos NAS. En particular abordan el costo del entrenamiento de las redes candidatas en cada etapa del NAS. La idea consiste en calcular una métrica para las redes sin entrenar a partir de una muestra o batch pequeño de datos (por defecto 128) que permita estimar el desempeño de dicha arquitectura finalmente entrenada, y así evitar el entrenamiento continuo y evaluación de todas las arquitecturas candidatas. Empleando esta métrica los autores logran encontrar arquitecturas competentes en fracciones de minutos, que entrenadas logran alrededor de 90 % de precisión en datasets como CIFAR-10, resultado satisfactorio (con un leve sacrificio de la precisión respecto a otros modelos) comparado con otras técnicas NAS que pueden llegar a demorar en el orden de días. El estimador que emplean, se basa en tomar los códigos binarios asociados a las capas ReLU del modelo para ese lote de datos, y computar la distancia de Hamming [49][18][121] entre los binarios, resultando en una matriz (*kernel*) que denotaron como K_H. El estimador computa el determinante de la matriz K_H, y se basa en que K_H parece contar con menos elementos fuera de la diagonal principal para aquellas redes que finalmente entrenadas tienen buena precisión.

Lopes et al. [77] proponen EPE-NAS, una idea similar a la estrategia anterior. En particular estudian la correlación de las salidas de los operadores lineales, también para un batch de 256 datos, incorporando los meta-datos del dataset como son la clase a la que pertenecen las muestras. Esto permite estudiar la correspondencia entre datos de una misma clase y baja correlación

con respecto a muestras de distintas clases. Para un importante conjunto de arquitecturas tomadas de otros NAS, muestran cierta correlación entre esta métrica aplicada sobre los modelos sin entrenar y la precisión de los mismos una vez entrenados.

3.3. Resumen del capítulo

En este capítulo se estudiaron los principales avances relacionados con el cómputo eficiente de NNs. Debido a la gran cantidad de aplicaciones que tienen hoy en día estas rutinas, se destacó la importancia de lograr una baja latencia y una alta productividad a la hora de computarlas y cómo, en muchos casos, lo anterior es una tarea muy compleja y recae de forma importante sobre las características del hardware utilizado. Sin embargo, también existen herramientas y estrategias que permiten optimizar modelos a través de software, permitiendo sacar el mayor partido posible a las plataformas de hardware que se tengan disponibles. La gran mayoría de las optimizaciones por hardware buscan explotar la reutilización de datos a través de memorias caché y buffers, reduciendo así las comunicaciones con la memoria principal, que tienden a ser un cuello de botella importante. También se han desarrollado procesadores específicos con esquemas de procesamiento espacial, donde generalmente se cuenta con un arreglo sistólico de unidades (PEs), por ejemplo, la TPU de Google.

Junto a la actual abundancia de sensores que captan información (por ejemplo cámaras o celulares) computar cerca de la fuente donde se adquiere la información se ha convertido en una opción muy atractiva. En lugar de tener que comunicar a un servicio en la nube, que tiene desventajas como la privacidad, conectividad y latencia, se han desarrollado múltiples dispositivos de hardware para la ejecución de inferencia a una baja latencia y coste energético. Está claro que la capacidad de estos dispositivos es mucho más limitada, por lo que son importantes las optimizaciones por software para sacarles el mayor partido posible.

Dentro de las optimizaciones por software se destacan las de bajo nivel, que consisten en reordenamientos y transformaciones para los principales *kernels* involucrados en el cómputo de las NNs, en particular derivando en operaciones sumamente estudiadas y optimizadas para múltiples plataformas de hardware, entre ellas la multiplicación de matriz-matriz y matriz-vector. Existen también

otras técnicas que buscan reducir la cantidad de multiplicaciones, aumentando la cantidad de sumas y variables intermedias. Una gran cantidad de estas optimizaciones se encuentran implementadas en *frameworks* y bibliotecas de los mismos fabricantes de hardware.

Luego se revisaron los algoritmos NAS, que buscan, recorriendo grandes espacios de arquitecturas de NNs, encontrar modelos óptimos para cierta métrica. El uso de la precisión como métrica, en general, deriva en algoritmos NAS sumamente costosos desde el punto de vista computacional debido al entrenamiento continuo de los modelos. Por esta razón se utilizan estimadores para los valores de precisión que impliquen menos cómputos. También destacan las técnicas que emplean podas sobre los modelos, buscando reducir la cantidad de cómputo eliminando parámetros que tienen poca o nula importancia sobre el resultado final. En este contexto NetAdapt, emplea podas específicas guiadas por mediciones empíricas de latencia para ajustar un modelo a cierta plataforma de hardware.

Capítulo 4

Evaluación experimental de hardware para redes neuronales

Uno de los objetivos de la tesis es el estudio y posterior implementación de entornos de ejecución para redes neuronales en distintos dispositivos de cómputo, permitiendo así una familiarización con este tipo de aplicaciones. Para ello, se abordaron una variedad de plataformas de hardware heterogéneas, combinadas con el uso de *frameworks* y algoritmos que permiten implementaciones eficientes de redes neuronales sobre las mismas.

En esta línea, los primeros esfuerzos consistieron en evaluar, de forma experimental, el desempeño de múltiples dispositivos, entre ellos CPUs Intel, GPUs Nvidia y algunos diseñados específicamente para el proceso de inferencia de redes neuronales como el EdgeTPU de Google. Particularmente, se llevaron adelante una gran variedad de experimentos, combinando la aplicación de distintas herramientas y mecanismos de optimización para poder explotar al máximo el rendimiento de cada plataforma de hardware. Las técnicas de optimización utilizadas en esta evaluación se pueden clasificar en, por un lado, aquellas diseñadas por varios de los fabricantes de plataformas de hardware y por otro, técnicas y algoritmos que pueden ser aplicados en conjunto para la simplificación de los modelos, como por ejemplo, NetAdapt. Entre las pruebas con *frameworks* para la optimización, se utilizaron TensorRT para plataformas Nvidia, OpenVINO Toolkit para dispositivos Intel, y TFLite en los dispositivos Google.

En la evaluación experimental se realizaron pruebas de inferencia para los diferentes dispositivos considerando distintas métricas, como latencia, produc-

tividad y precisión. Para el caso de la latencia, se evalúa el tiempo que le toma a cada par, modelo-dispositivo, realizar la inferencia para una sola muestra ($batch\_size = 1$). Luego, en cuanto a la evaluación de la productividad, se estudió la evolución de los tiempos de inferencia aumentando el tamaño del batch. Para las plataformas que lo soportan, se realizaron pruebas evaluando los modelos de redes neuronales utilizando `FP16` e `INT8`. Por último, se realizó una evaluación cruzada de los diferentes modelos generados por NetAdapt llevados a otras plataformas de hardware, con el objetivo de observar si las mejoras de un modelo optimizado para un dispositivo específico pueden aplicar para otro.

4.1. Redes evaluadas

Las redes evaluadas, en primera instancia, son dos clásicas y de diferente porte: AlexNet y MobileNet. Mobilenet [55] es una red diseñada especialmente para dispositivos con recursos limitados, con un bajo uso de memoria, siendo almacenada en alrededor de 13MiB dependiendo de la implementación. Aprovecha las capas *depthwise* y *pointwise* para realizar cálculos rápidos sobre capas convolucionales, sumado a capas de normalización por batches entre convoluciones para mejorar la precisión. Por otro lado, AlexNet [68] está compuesto por capas convolucionales profundas conectadas a un clasificador lineal que genera el resultado. En contraste con MobileNet, el almacenamiento de los pesos y sesgos de AlexNet requiere aproximadamente 220MiB. Si bien ambas arquitecturas pueden ser consideradas antiguas dentro del campo de redes neuronales, existen varios puntos a destacar a favor de utilizarlas en esta investigación. Los puntos principales se describen a continuación.

- **Disponibilidad de recursos:** La disponibilidad de recursos, como modelos pre-entrenados, implementaciones e información para ambas arquitecturas fue importante en la elección. Esta disponibilidad facilita el trabajo de desarrollo permitiendo destinar un mayor esfuerzo en la evaluación de las diferentes herramientas y optimizaciones en las que se enfoca este trabajo de maestría.
- **Comparación generacional y relevancia histórica:** La elección de AlexNet y MobileNet permite una comparación entre generaciones de modelos. AlexNet fue una de las primeras arquitecturas de DNNs exi-

tosas sobre ImageNet, mientras que MobileNet se diseñó posteriormente para abordar la eficiencia en dispositivos móviles. Este enfoque proporciona una perspectiva cronológica sobre la evolución de las NNs convolucionales.

- **Impacto continuo:** Por último, la influencia continua de ambas arquitecturas en la comunidad científica, evidenciada por su impacto en investigaciones y desarrollos más recientes es otro aspecto que refuerza la elección para este trabajo.

En resumen, la combinación de aspectos comparativos, relevancia histórica, disponibilidad de recursos, simplicidad y su impacto continuo son algunas de las razones que justifican la elección de AlexNet y MobileNet como casos de prueba para las evaluaciones empleando herramientas y técnicas de optimización de NNs desarrolladas en el marco de esta tesis. Una comparación de las dimensiones y características de ambas arquitecturas se presenta en la Tabla 4.1.

Tabla 4.1: Descripción y comparación entre AlexNet y MobileNet.

Parámetro	**AlexNet**	**MobileNetV1**
Profundidad	8 capas	28 capas
Número de parámetros	∼61 mill.	∼4.2 mill.
Tamaño (MiB)	∼220 MiB	∼13 MiB
Max FC Input Dims.	9216	1024
Max FC Output Dims.	4096	1000
Número de capas Conv.	5	13
Número de capas FC	3	1
Número de capas Pooling	3	1
Número de capas de Activación	5	13

4.2. Entorno de ejecución

Para los experimentos realizados utilizando el algoritmo NetAdapt, fue necesario un entorno "potente" que permitiera llevar a cabo las optimizaciones sobre las redes neuronales. Estos experimentos fueron realizados en un servidor equipado con un procesador Intel(R) Core(TM) i7-6700 @ 3.40GHz (con 4 cores, 64kB en cache L1), 64 GB de memoria RAM, y 3 GPUs: dos Nvidia GeForce GTX 980 Ti (6GB) y una Nvidia GeForce GTX 1060 (3GB).

Además, para la evaluación experimental se realizaron pruebas sobre varias plataformas heterogéneas de hardware. Estos van desde procesadores de propósito general, hasta plataformas de hardware específico desarrollados para tareas de IA. A continuación se listan de los dispositivos, clasificados en generales y específicos.

- Generales:
  - **CPU x86-64 Intel Core i7-4790**
  - Nvidia GPUs:
    - **GeForce GTX 970**
    - **Tesla P100**
- Específicos:
  - Nvidia Jetson:
    - **TX2**
    - **Xavier AGX**
  - **Intel NCS2**
  - **Google Edge TPU**

En las siguientes secciones se detallan, desde un enfoque técnico, las principales características de los dispositivos utilizados en la evaluación experimental, así como una breve introducción al flujo de trabajo empleado para la implementación de soluciones en cada caso.

4.2.1. Intel NCS2

El Intel NCS2 es un acelerador para tareas de IA, encapsulado en un dispositivo USB con una unidad de procesamiento visual (VPU). Cabe destacar que para ejecutar la inferencia en este dispositivo es obligatorio convertir el modelo al formato OpenVINO. Primero, se convierten los modelos PyTorch usando el convertidor ONNX de PyTorch. Luego, OpenVINO proporciona Model Optimizer (MO), una herramienta de línea de comandos que permite convertir modelos en formato ONNX para producir representaciones intermedias (IR). Por último, OpenVINO Runtime permite cargar estas representaciones intermedias en el NCS2. Los modelos resultantes generados por NetAdapt también se convirtieron a ONNX para ejecutarse en la VPU.

4.2.2. Google Coral AI EdgeTPU

Para la evaluación del Edge TPU, en este trabajo se empleó el Coral Dev Board de Google, que contiene el acelerador EdgeTPU como co-procesador. Específicamente, está equipado con un procesador ARM quad-core Cortex-A53 y un Cortex-M4F, un procesador de video GC7000 Lite, 1Gb de memoria LPDDR4 y varios puertos de E/S. De forma similar al Intel NCS2, debe seguirse un estricto flujo de trabajo para poder implementar una solución en el Edge TPU.

La implementación de NetAdapt para la optimización de los modelos sobre la Edge TPU es una tarea laboriosa. Para poder obtener las mediciones de latencia necesarias para el algoritmo NetAdapt, cada capa de la red y sus variantes removiendo filtros necesita ser convertida de PyTorch a TensorFlow Lite, cuantizada, y posteriormente compilada con el `edgetpu_compiler`. Por esta razón sólo se evaluó MobileNet en conjunto con NetAdapt. Además, puesto que solo soporta tamaños de batch igual a 1, sólo se evaluó la latencia para este caso.

4.2.3. Dispositivos Nvidia CUDA

Se realizaron pruebas sobre cuatro dispositivos Nvidia de características muy distintas:

- Nvidia **GTX 970:** es una tarjeta gráfica diseñada para usuarios y jugadores. Lanzada en 2014, ofrece un rendimiento sólido en juegos y aplicaciones gráficas, con 4 GB de memoria GDDR5 y soporte para tecnologías como Nvidia GameWorks [5].
- Nvidia **Tesla P100:** es la primer GPU en servidores dedicados al cómputo de alto desempeño específicamente orientada a la inteligencia artificial. Con 16 GB de memoria HBM2 y la arquitectura Pascal, esta tarjeta está diseñada para cargas de trabajo intensivas en cálculos, como el aprendizaje profundo y la investigación científica [10].
- Nvidia **Jetson TX2:** es una plataforma de computación embebida diseñada para aplicaciones de inteligencia artificial y aprendizaje profundo en dispositivos integrados. Ofrece un equilibrio entre rendimiento y eficiencia energética, permitiendo la ejecución de algoritmos complejos en sistemas embebidos y robots.

- NVIDIA **Jetson AGX Xavier:** similar a la TX2, es una potente plataforma de IA diseñada para sistemas embebidos de alto rendimiento. Incorpora la arquitectura Volta y está equipada con ocho núcleos de CPU ARM, junto con una GPU NVIDIA Volta con 512 núcleos CUDA y 64 Tensor Cores, incluyendo además 2 aceleradores de inteligencia artificial (NVDLA). Esta plataforma es ideal para aplicaciones que requieren procesamiento intensivo, como vehículos autónomos y robots avanzados.

Se empleó la API de TensorRT para que la implementación de las arquitecturas AlexNet y MobileNet pudieran ejecutarse de forma nativa por el runtime de TensorRT de los dispositivos. Para evaluar los modelos sin TensorRT, se empleó la extensión CUDA de PyTorch. Para este caso particular, para ejecutar los modelos optimizados por NETADAPT, se crearon funciones que analizan el modelo capa por capa y los re-codifica utilizando las primitivas provistas por la API de TensorRT. Particularmente, todas las evaluaciones de las plataformas NVIDIA fueron realizadas empleando `FP32`, mientras que en el caso de los Jetson, se utilizó la configuración de máxima potencia disponible.

Característica	**Jetson TX2**	**Jetson AGX Xavier**	**GTX 970**	**P100**
Memoria	4GB 128-bit LPDDR4	32GB 256-bit LPDDR4x	4GB 256-bit GDDR5	12GB 3072-bit HBM2
Arq. GPU	NVIDIA Pascal	NVIDIA Volta	NVIDIA Maxwell	NVIDIA Pascal
CUDA cores	384	512	1664	3584
Tensor Cores	-	64	-	-
GPU Max. freq.	1300 MHz	1377 MHz	1178 MHz	1328 MHz
Power	7.5/15W	10/15/30W	145W	250W
Acelerador de DL	-	2× NVDLA	-	-

Tabla 4.2: Comparación de características de GPUs NVIDIA de los diferentes dispositivos utilizados.

4.3. Resultados experimentales

En las secciones anteriores fueron introducidas las redes utilizadas como casos de pruebas (Sección 4.1) y los dispositivos utilizados para la evaluación experimental (Sección 4.2). A continuación se presentan los resultados obtenidos para las distintas métricas mencionadas al principio de este capítulo, entre ellas: latencia, productividad y evaluación cruzada de los modelos generados por NETADAPT.

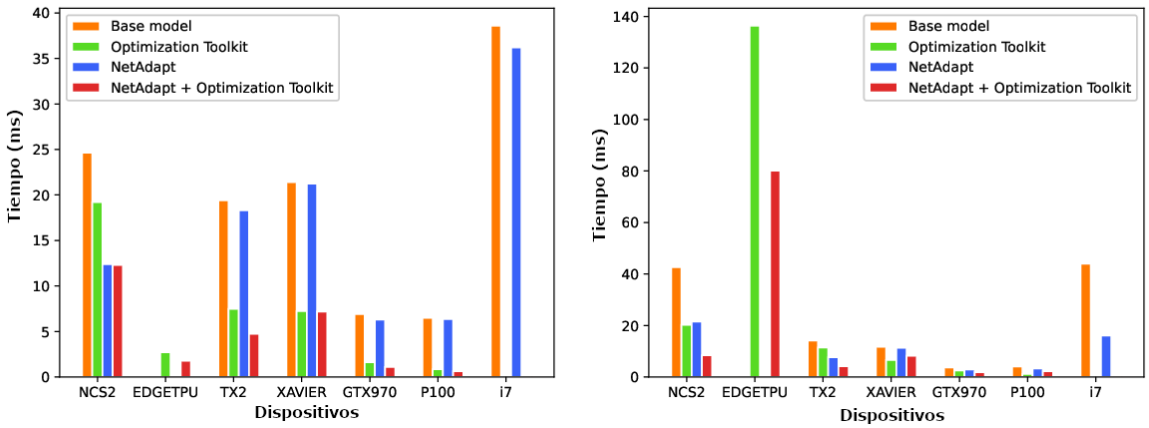
4.3.1. Estudio de latencia

En el primer experimento, se busca medir las latencias en los diversos dispositivos mediante la aplicación de distintas herramientas de optimización. Para ello, se ejecutan ambos casos de prueba, con la opción de aplicar o no las optimizaciones, y combinando estas con NetAdapt. Estas combinaciones generan múltiples modelos que se evalúan en cada plataforma.

La Figura 4.1 resume, de manera gráfica, los resultados de latencias al combinar tanto el modelo base como el simplificado por NetAdapt con las diferentes herramientas de optimización.

Analizando los resultados se puede observar que todas las optimizaciones propuestas derivan en una reducción en la latencia cuando se compara contra el modelo base. Entre otros resultados se destaca la mejora cercana a 2× que obtiene NetAdapt optimizando MobileNet para el NCS2.

En segundo lugar, para algunas plataformas de hardware existe una importante reducción de la latencia para varias de las estrategias. Por ejemplo, observando los resultados para AlexNet, en los dispositivos Intel NCS2 y la TX2, las mejores configuraciones permitieron registrar tiempos de inferencia 5,3× y 3,1× más rápidos comparados con el modelo base.



(a) MobileNet **(b)** AlexNet

Figura 4.1: Latencia (en ms), para las redes MobileNet y AlexNet, agrupada por dispositivos. Los dispositivos TX2, Xavier, GTX970 y la P100 emplean TensorRT como herramienta de optimización y PyTorch CUDA para la ejecución de NetAdapt y el modelo base. Para el Intel NCS2 se utilizan las optimizaciones realizadas por OpenVINO. NetAdapt y el modelo base se ejecutan sin las optimizaciones del OpenVINO.

Para el dispositivo NCS2 ejecutando MobileNet se puede observar que la combinación del NetAdapt con OpenVINO no aporta mejora alguna, mientras que para AlexNet esta combinación permite alcanzar una mejora en la velocidad de 2,8× con respecto a NetAdapt cuando no se emplean las optimizaciones de OpenVINO.

En cuanto a las diferencias de latencia entre las distintas plataformas de hardware, se puede observar que, en general, la Nvidia P100 tiene un desempeño mayor a los demás dispositivos. Esto era de esperar teniendo en cuenta la capacidad de cómputo, el costo y el consumo de energía de los diferentes dispositivos. Como excepción se identificó que AlexNet optimizada con NetAdapt +TensorRT presenta un tiempo de inferencia bastante más bajo en la GTX 970.

En cuanto a los dispositivos edge, el TX2 tiene más de 2× mejor rendimiento con NetAdapt y TensorRT aplicados para MobileNet, que NetAdapt con OpenVINO en el NCS2. Esto destaca la importancia de la optimización del modelo para alcanzar resultados competitivos. Se puede observar que TensorRT funciona particularmente bien en los modelos MobileNet.

La Figura 4.1 también permite destacar que la aplicación de NetAdapt con PyTorch no produjo una reducción significativa de la latencia en las plataformas de Nvidia. Además, existe una diferencia de más de 4× al ejecutar el modelo optimizado por NetAdapt con y sin la aplicación de TensorRT, alcanzando la máxima mejora en la P100, con un factor de 10×. A partir de estos resultados, de forma prematura, se puede establecer que la aplicación de TensorRT y las optimizaciones de OpenVINO ofrecen beneficios para las redes convolucionales, independientemente de las diferentes plataformas de hardware utilizadas. Sin embargo, lo que no se puede asegurar con certeza es el nivel de mejora aportado por la aplicación de estas herramientas de forma genérica.

Otro aspecto a destacar es el desempeño de la EdgeTPU para MobileNet que, como se muestra en la Figura 4.1a, supera a los demás dispositivos edge. Es más, el EdgeTPU casi se comporta como los dispositivos de mayor porte, i.e., la GTX 970 y la P100. No obstante, la comparación puede resultar algo injusta puesto que el EdgeTPU emplea para la inferencia modelos cuantificados para enteros de 8-bits. A pesar de esto, los resultados son positivos, dado que el proceso de cuantificación sólo implicó una reducción de la precisión en los resultados en un 2 %. También se puede observar la incompatibilidad del EdgeTPU con modelos de dimensiones que superan las capacidades del

dispositivo (observar cómo con AlexNet se obtienen los peores resultados de latencia). Si bien la aplicación de NetAdapt resulta positiva, no es suficiente para obtener latencias comparables con los demás dispositivos.

Como resultado final, las mediciones de latencia para el Intel Core i7 resaltan la importancia del uso de aceleradores de hardware especializados para la inferencia de NNs, ya que casi todas las plataformas de borde superan a este procesador. A pesar de eso, NetAdapt realiza un gran trabajo al optimizar AlexNet, aunque casi no hubo cambios de latencia con el modelo optimizado de MobileNet.

4.3.2. Evaluación del throughput

En muchas ocasiones un requerimiento importante en el contexto de la IA son valores de productividad altos junto con una baja latencia. En este sentido, la siguiente etapa del estudio consiste en evaluar cómo evoluciona la latencia mientras se aumenta el tamaño del batch para los diferentes modelos optimizados.

El experimento diseñado consiste en, para cada dispositivo, tomar mediciones de tiempo de inferencia a medida que se incrementaba el tamaño del batch (BS), hasta que el dispositivo se quede sin memoria para realizar la inferencia. En este sentido, para la P100 se midió hasta $BS = 200$; $BS = 28$ para la GTX 970; hasta $BS = 18$ en el NCS2; y $BS = 32$ en el Jetson TX2; para el Xavier, $BS = 130$. Por último, el EdgeTPU, solo soporta inferencias de a una muestra ($BS = 1$), por lo tanto no se realizaron estas pruebas. Por cuestiones prácticas y claridad, las líneas de las gráficas son continuas, aunque en realidad el tamaño de batch es una variable discreta. Los resultados se resumen de manera gráfica en las Figuras 4.2, 4.3, 4.4, 4.5 y 4.6.

De la Figura 4.2 se puede destacar que en la P100 para tamaños de batch pequeños ($BS < 15$) los modelos optimizados con TensorRT (el original y el simplificado por NetAdapt) presentaron una latencia mucho menor que los modelos originales, tanto para AlexNet como MobileNet. El comportamiento es distinto para tamaños de batch más grandes, para los cuales el modelo con mejor desempeño es el optimizado por NetAdapt. En el caso particular de AlexNet, para este dispositivo, se puede observar cómo los resultados aplicando las optimizaciones de TensorRT con tamaños de batch grandes son los peores. Los factores que pueden influir en este comportamiento son varios.

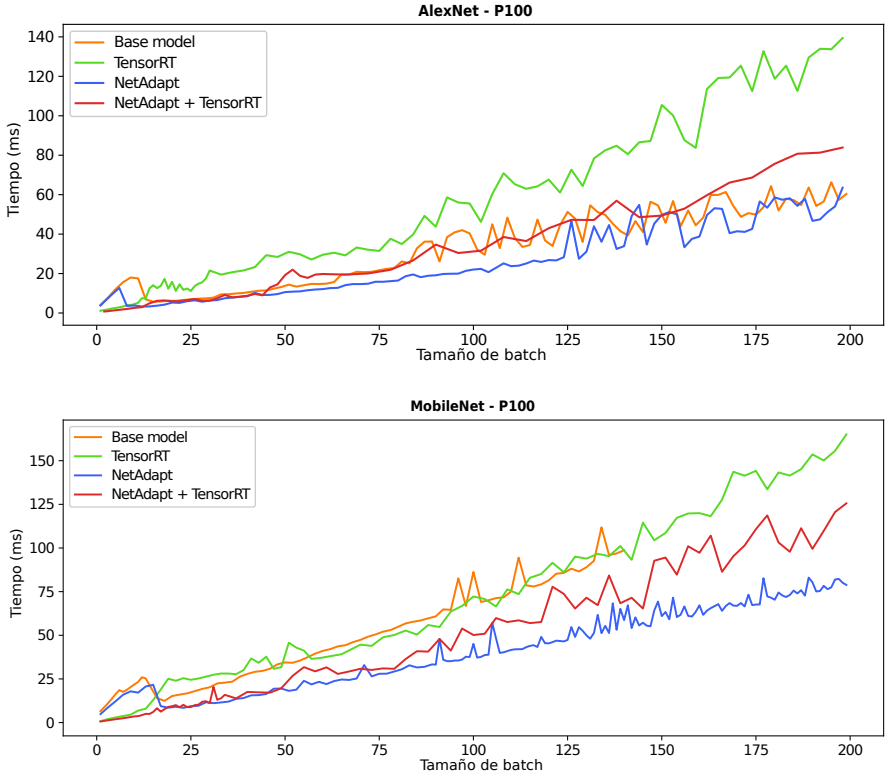


Figura 4.2: Tiempo de ejecución en función del tamaño de batch para AlexNet y MobileNet en la NVIDIA Tesla P100.

Por ejemplo, TensorRT aplica optimizaciones importantes como la fusión de capas. En general estas optimizaciones funcionan bien para tamaños pequeños de batch (algunos refieren a un tamaño de batch óptimo [11]) pero tamaños más grandes pueden causar que se alcance el límite de memoria. Asociado a esto, entra también en juego la elección del tamaño del *workspace* (parámetro de memoria del `trtexec` empleado para cómputos intermedios). En los experimentos realizados no se varió este parámetro, empleando un tamaño cercano al máximo de memoria disponible. Al incrementar el tamaño de batch es posible que se trabaje muy cerca del límite de espacio disponible en el workspace, degradando el desempeño del *engine* generado por TensorRT. Para entender más a fondo estos resultados son necesarias pruebas exhaustivas utilizando herramientas para poder ver, por ejemplo, los *kernels* empleados por TensorRT, lo que deriva en una posible linea de trabajo futuro.

En la Figura 4.3 se resumen los resultados para el dispositivo NCS2. Para ambas redes, se observa un crecimiento lineal de la latencia al aumentar el tamaño de batch para cada una de las configuraciones. Esta evaluación se

da en todos los casos con reducciones en el tiempo de ejecución cuando se compara con el modelo base. Para AlexNet, OpenVINO combinado con NetAdapt supera a todas las demás optimizaciones para los tamaños evaluados. En el caso del modelo MobileNet simplificado por NetAdapt se observa una mejora importante contra la ejecución del modelo base con las optimizaciones de OpenVINO, y se puede notar que para este modelo más pequeño la adicción de OpenVINO, apenas presenta una leve mejora para tamaños de batch crecientes.

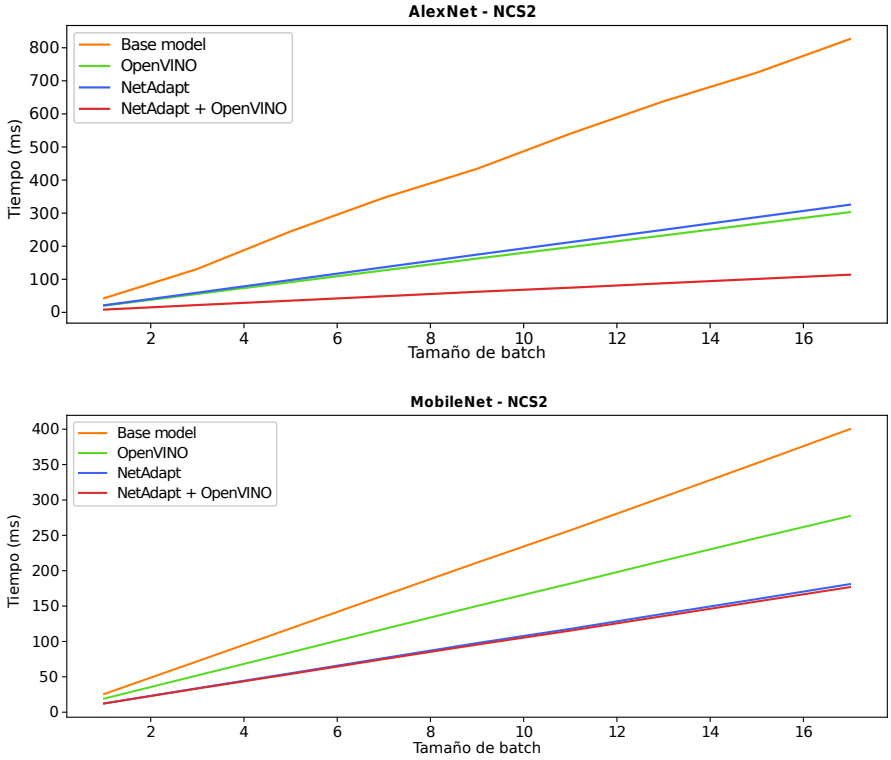


Figura 4.3: Tiempo de ejecución en función del tamaño de batch para AlexNet y MobileNet en el Intel NCS2.

Las mediciones de tiempo de inferencia sobre la GTX 970 se presentan en la Figura 4.4. Los resultados también presentan un crecimiento lineal para tamaños crecientes de batch, para AlexNet hasta $BS \sim 25$, y para MobileNet, hasta $BS \sim 20$. Para los tamaños más grandes de batch, en el modelo original se pueden observar tiempos de latencia mucho más elevados. Utilizando el Nvidia System Management Interface, se pudo observar cómo PyTorch CUDA estaba utilizando casi toda la memoria de la GPU. Por esta razón, es posible que haya una penalización fuerte asociada a esta tarjeta cuando trabaja muy cerca de

los límites de la cantidad de VRAM. Otro fenómeno que se puede observar es que, de forma similar a la P100, AlexNet simplificado con NetAdapt y TensorRT para tamaños de batch grandes ($BS > 5$) presenta un desempeño peor, indicando que las optimizaciones de TensorRT para las GPUs P100 y la GTX 970 tienen sus mejores resultados para batches pequeños. En el caso de AlexNet, también se puede observar un comportamiento similar al que aparece en la P100, donde a partir de cierto tamaño del batch ($7 < BS < 25$), el modelo original presenta una menor latencia que los modelos optimizados con TensorRT. Una cualidad que estas GPUs comparten y puede estar relacionada al desempeño de TensorRT, es la carencia de unidades especializadas en tareas de IA, como son los Tensor Cores.

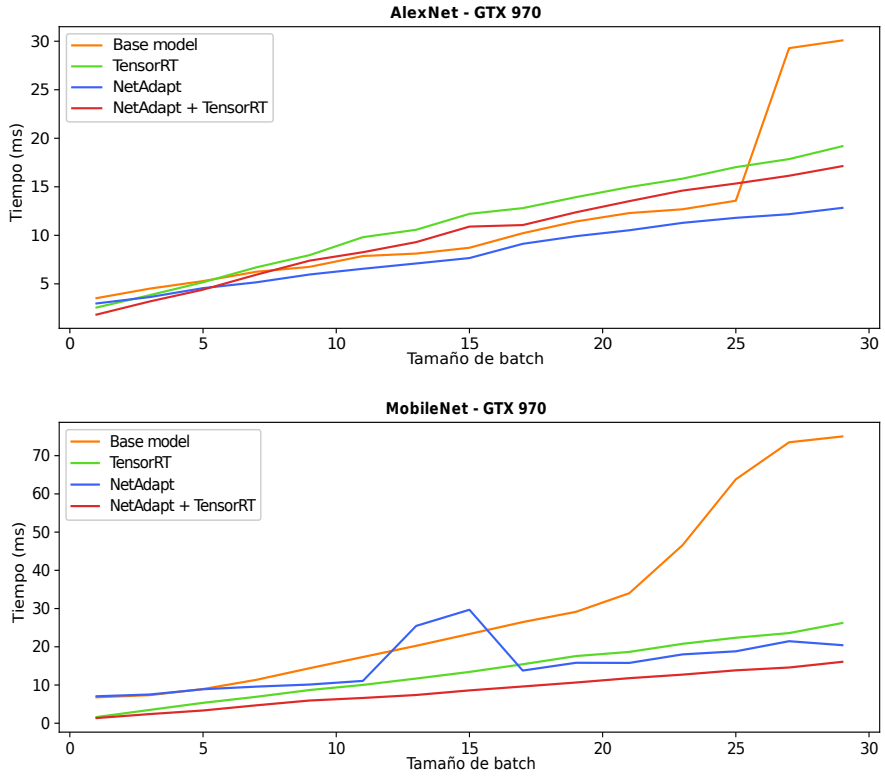


Figura 4.4: Tiempo de ejecución en función del tamaño de batch para AlexNet y MobileNet en la Nvidia GTX 970.

Por último, restan los resultados de las plataformas Jetson de Nvidia. La Figura 4.5 presenta las mediciones sobre el TX2, y en ella se pueden observar leves mejoras utilizando TensorRT en el caso de AlexNet para tamaños de batch menores a 10, tanto para la versión base, como la simplificada por NetAdapt. Luego de ese tamaño, se puede notar como la aplicación de TensorRT degrada

el desempeño. De forma contraria, el desempeño de las variantes indican que, para MobileNet, TensorRT aporta importantes mejoras en tamaños crecientes de batch, siendo el mejor modelo el simplificado por NetAdapt junto con TensorRT.

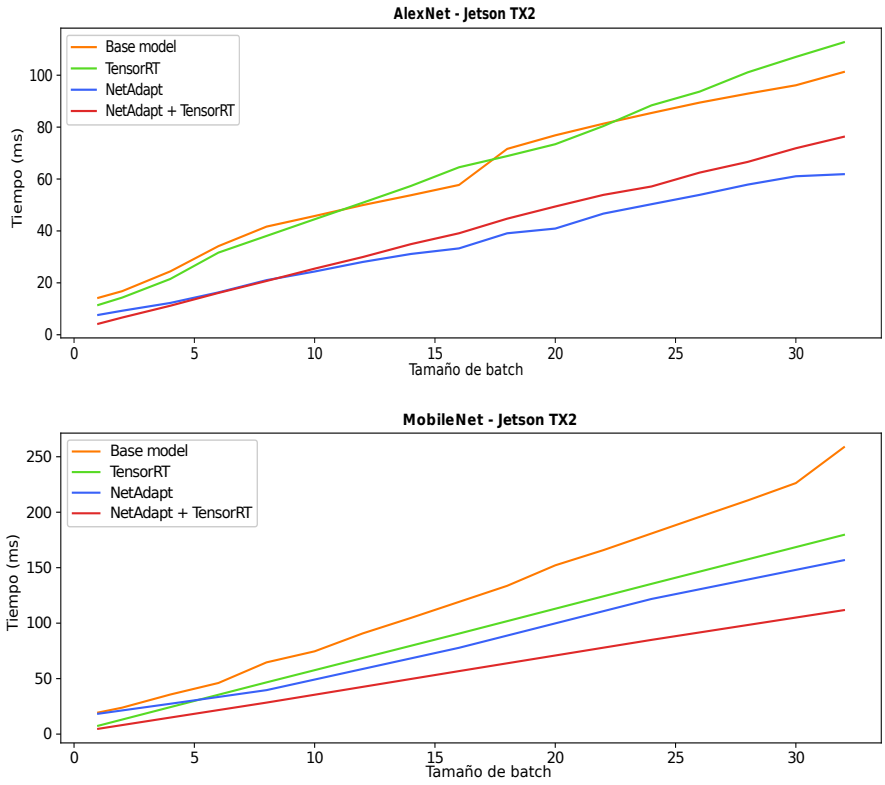


Figura 4.5: Tiempo de ejecución en función del tamaño de batch para AlexNet y MobileNet en la Nvidia Jetson TX2.

Los resultados para el Jetson Xavier se pueden observar en la Figura 4.6. A diferencia del TX2, el Xavier presenta un comportamiento similar para ambos modelos. En el caso de AlexNet, la aplicación de TensorRT supone una mejora de desempeño, lo cual se puede deber a la cantidad de memoria y unidades de cómputo del Xavier, incluidos los Tensor Cores y los aceleradores de Deep Learning, a los que TensorRT puede sacar mayor partido para hacer frente a la gran cantidad de parámetros que tiene AlexNet. Tanto para las variantes de AlexNet como para MobileNet, los mejores resultados están dados por la combinación de TensorRT y NetAdapt.

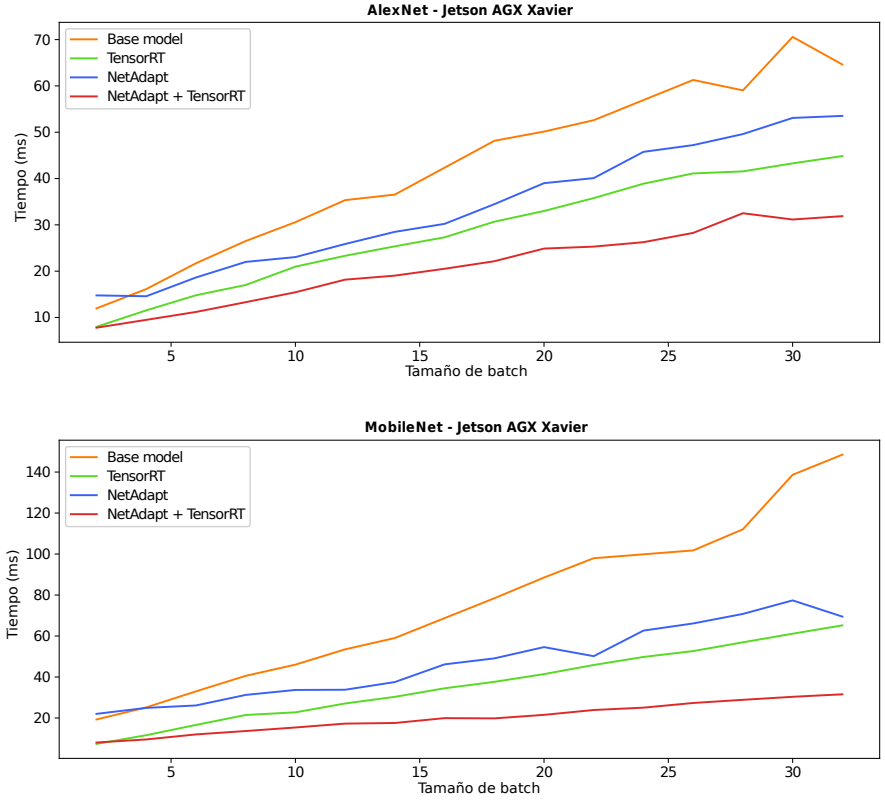


Figura 4.6: Tiempo de ejecución en función del tamaño de batch para AlexNet y MobileNet en la NVIDIA Jetson AGX Xavier.

4.3.3. Evaluación de uso de precisiones reducidas

Esta sección presenta los resultados de la aplicación de técnicas de cuantificación de modelos. Específicamente, representar los parámetros de las redes empleando, en lugar de `FP32`, precisiones numéricas reducidas como `INT8` y `FP16`. Esta evaluación se realiza sobre los dispositivos NVIDIA Jetson empleando el SDK TensorRT. En este contexto, los experimentos se realizaron con un tamaño de batch de 32.

Las Tablas 4.3 y 4.4 presentan los resultados de latencia, destacando las mejoras derivadas de la reducción de la precisión utilizada para el cómputo de redes neuronales. Los resultados más destacados se observan en el dispositivo Xavier, especialmente con AlexNet (Tabla 4.3), donde se logran mejoras en el tiempo de ejecución de hasta 3,1× con `INT8` y de 2,5× con `FP16` en comparación con el modelo original. Por el contrario, en el caso del TX2, las mejoras no son tan pronunciadas, particularmente para la red AlexNet.

Tanto la versión original como la simplificada con NETADAPT muestran

AlexNet	Base	Original + TRT			NA	NA + TRT		
	FP32	FP32	FP16	INT8	FP32	FP32	FP16	INT8
TX2	99,6	103,4	78,2		60,4	66,1	54,4	
Xavier	60,0	40,8	23,3	19,3	45,9	28,1	21,7	19,4

Tabla 4.3: Latencia (en ms) con un tamaño de batch de 32 para AlexNet en dispositivos NVIDIA Jetson variando la representación de parámetros mediante cuantificación.

MobileNet	Base	Original + TRT			NA	NA + TRT		
	FP32	FP32	FP16	INT8	FP32	FP32	FP16	INT8
TX2	243,7	157,4	145,4		140,5	93,3	92,2	
Xavier	130,2	64,5	26,0	23,4	59,4	25,2	22,8	22,1

Tabla 4.4: Latencia (en ms) con un tamaño de batch de 32 para MobileNet en dispositivos NVIDIA Jetson variando la representación de parámetros mediante cuantificación.

mejoras sustanciales al optimizar con TensorRT y FP16. Sin embargo, las mejoras entre ambas versiones optimizadas por TensorRT no son tan notables en el TX2, indicando que las precisiones numéricas reducidas no generan beneficios tan significativos en este dispositivo. En cambio, en el Xavier, las mejoras son más importantes, aunque la diferencia entre FP16 e INT8 no es tan marcada. Además, estos resultados sugieren que para el dispositivo Xavier hay poca diferencia práctica entre el modelo original y el modelo NETADAPT después de aplicar técnicas de cuantificación (notar que utilizando FP32 sí se pueden apreciar mejoras).

Un punto importante a destacar es que este tipo de técnicas, en general, acarrea reducciones de precisión en los resultados. En este caso, las reducciones de la precisión fueron en el entorno del 2 al 4 %.

A modo de resumen de esta evaluación, las técnicas de cuantificación parecen funcionar mejor en el Xavier. Una de las posibles razones para justificar este comportamiento está dado por la presencia de los Tensor Cores y los NVDLA, unidades a las que TensorRT saca partido.

4.3.4. Evaluación cruzada de modelos

Como se mencionó anteriormente, ejecutar el algoritmo NETADAPT es costoso y no siempre conduce a los resultados esperados en reducción de latencia.

Además, debido a las adaptaciones del esquema de ejecución de NetAdapt para los diferentes dispositivos, el costo de construir la tabla de búsqueda no es igual para todos. Por ejemplo, para el Intel NCS2 y el EdgeTPU la transformación de los modelos sobre estas plataformas involucra un costo para cada configuración que se evalúe. Por lo tanto, resulta interesante analizar hasta qué punto las optimizaciones del modelo obtenidas para una plataforma específica pueden llevar a un rendimiento mejorado en otras. Esto puede permitir la ejecución de NetAdapt para un dispositivo donde la construcción de la tabla de búsqueda sea más rápida y la ejecución del modelo en otro, o varios, dispositivos.

En la Tabla 4.5 se presenta el porcentaje de reducción de tamaño de los modelos simplificados por NetAdapt, tanto para AlexNet como MobileNet, para los diferentes dispositivos. Rápidamente se puede observar cómo, para ambas arquitecturas, NetAdapt reduce sustancialmente el peso para el caso del Intel NCS2. De forma similar, también se reduce el tamaño de AlexNet en un 84 % para el Nvidia Jetson TX2.

Un fenómeno más general que se puede observar en estos resultados es que la dimensión de la red AlexNet es reducida en mayor medida que MobileNet sobre la gran mayoría de los dispositivos evaluados. Esta reducción se puede deber a la diferencia de dimensiones y tamaño de AlexNet, siendo una red más antigua y posiblemente sobredimensionada para el juego de datos empleado, además de las características de los dispositivos, muchas veces limitados por memoria, siendo necesaria la remoción de mayor cantidad de filtros y neuronas para obtener reducciones de latencia equivalentes. Como resultado, el algoritmo NetAdapt reduce aún más las capas que tienen el menor impacto sobre la precisión de la red a la vez que tienen una mayor reducción de la latencia, por consecuencia reduciendo el tamaño final del modelo. En este caso, AlexNet implica un número mayor de capas profundas con una alta dimensionalidad que pueden ser reducidas.

Las Tablas 4.6 y 4.7 muestran la reducción de latencia de los modelos NetAdapt optimizados para diferentes dispositivos en comparación con la latencia original de AlexNet y MobileNet. La estrategia fue, para cada plataforma, ejecutar cada uno de estos modelos y computar la latencia promedio para 1000 iteraciones. En dichas tablas, cada entrada de una fila indica el porcentaje de reducción de latencia al ejecutar el modelo optimizado por NetAdapt en el dispositivo asociado.

Modelo	MobileNet	AlexNet
NetAdapt-NCS2	82 %	86 %
NetAdapt-GTX970	64 %	52 %
NetAdapt-i7	20 %	49 %
NetAdapt-TX2	25 %	84 %
Modelo original	12,4MiB	217,6MiB

Tabla 4.5: Porcentaje de reducción del tamaño de los modelos correspondientes a AlexNet y MobileNet construidos por NetAdapt para algunas de las plataformas.

Los resultados muestran que el modelo con el mejor rendimiento es, en la mayoría de los casos, el construido para el dispositivo correspondiente. Esto es un indicador de que las podas guiadas por las mediciones de latencia, obtenidas para el dispositivo de destino, son efectivas y específicas de cada plataforma. De hecho, en muchos casos, los modelos optimizados por NetAdapt de MobileNet han mostrado un rendimiento inferior al del modelo base si el dispositivo objetivo no coincide con el dispositivo que ejecuta el modelo (marcado como 0 % en la Tabla 4.7). Sin embargo, es importante destacar que los modelos NetAdapt generados para el TX2 obtienen reducciones de latencia en otros dispositivos también. En particular, en AlexNet, la reducción para el procesador Intel i7 y el NCS2 está cerca de la mejor, y para el i7 es incluso mayor que la obtenida para el TX2.

Considerando los tamaños de red, el modelo NetAdapt de AlexNet para la GTX 970 reduce la latencia en un 82 %, mientras que la versión para el NCS la reduce un 60 %, aunque esta última es 3,5× más pequeña. Esto confirma que reducir el tamaño de la red no implica necesariamente una reducción proporcional de la latencia. Esto se explica en el documento original de NetAdapt [127].

Device	NA-NCS2	NA-GTX	NA-TX2	NA-i7	Modelo original
NCS2	59 %	38 %	57 %	45 %	20ms
GTX	23 %	82 %	23 %	5 %	4ms
TX2	35 %	18 %	36 %	17 %	20ms
i7	62 %	39 %	64 %	64 %	44ms

Tabla 4.6: Porcentaje de reducción de la latencia para las versiones simplificadas (por NetAdapt) de AlexNet ejecutadas en diferentes dispositivos.

Device	NA-NCS2	NA-GTX	NA-TX2	NA-i7	Modelo original
NCS2	49 %	31 %	27 %	4 %	19ms
GTX	1 %	4 %	0 %	1 %	8ms
TX2	6 %	6 %	5 %	1 %	20ms
i7	0 %	0 %	0 %	6 %	39ms

Tabla 4.7: Porcentaje de reducción de la latencia para las versiones simplificadas (por NetAdapt) de MobileNet ejecutadas en diferentes dispositivos.

4.4. Resumen del capítulo

Debido a la extensión de las pruebas realizadas en la evaluación experimental, esta sección busca resumir y sintetizar los principales resultados obtenidos, destacando algunas conclusiones generales.

En primer lugar, la evaluación de latencia permitió observar los importantes beneficios de aplicar técnicas para el cómputo eficiente de NNs. Además estas técnicas no comprometen drásticamente la precisión de los resultados finales. En particular, se pudo notar que tanto las herramientas desarrolladas por los mismos creadores de hardware, como NetAdapt, permiten mejorar el desempeño de los modelos en muchos casos. En el estudio, se destacan mejoras significativas en la velocidad de inferencia para AlexNet en dispositivos como Intel NCS2 y la Jetson TX2, resultado destacable teniendo en cuenta que son dispositivos de bajo porte y con limitantes energéticas. En esta evaluación NetAdapt por sí solo no obtiene los mejores resultados. Sin embargo, es importante tener en cuenta que los resultados pueden variar según la arquitectura específica de la red neuronal, el hardware utilizado y las estrategias de optimización aplicadas.

Respecto a la segunda evaluación, midiendo los tiempos de inferencia variando el tamaño de batch procesado, destacan claramente las diferencias entre los portes y capacidades de procesamiento de cada plataforma. Específicamente, la P100 es capaz de procesar hasta 200 muestras por batch, mientras que la Intel NCS2 llega a un máximo de 16 para las redes evaluadas. En esta instancia, para tamaños crecientes de batch se pudo notar la ventaja de NetAdapt, posiblemente debido a que las podas sobre las redes reducen la memoria necesaria para computar el modelo y permiten ejecutar la inferencia con más muestras a un menor tiempo que el modelo base, e incluso que las otras versiones optimizadas por las demás herramientas.

El siguiente experimento importante consistió en aplicar técnicas de cuantificación de modelos en los dispositivos NVIDIA Jetson, utilizando precisiones reducidas como `INT8` y `FP16` en lugar de `FP32` para el cómputo de la inferencia. En el dispositivo Xavier, especialmente con AlexNet, se observan mejoras significativas de hasta $3,1\times$ en el tiempo de ejecución con `INT8` y $2,5\times$ con `FP16`. Las mejoras en TX2 son menos pronunciadas, especialmente para AlexNet. Para MobileNet, en el TX2 las mejoras son notables, especialmente al utilizar TensorRT y `FP16`. En general, estas técnicas parecen funcionar mejor en el dispositivo Xavier, posiblemente debido a la presencia de Tensor Cores diseñados para computar eficientemente el producto de matrices en varias precisiones.

Por último, en la evaluación cruzada de los modelos optimizados por NETADAPT se puede observar que, en primer lugar, se logran reducciones sustanciales en el tamaño de los modelos, siendo particularmente eficiente en dispositivos como Intel NCS2 y NVIDIA Jetson TX2, donde alcanza reducciones del orden del 85 % para AlexNet. Otro resultado interesante es que la reducción en el tamaño de las redes no garantiza una reducción proporcional en la latencia. Se pudo observar, en la evaluación cruzada, que los modelos NETADAPT diseñados específicamente para un dispositivo generalmente muestran el mejor rendimiento en ese dispositivo, indicando la efectividad y especificidad de las podas guiadas por mediciones de latencia. Además, se pudo observar cómo los modelos generados para el TX2 obtuvieron importantes reducciones de latencia para otros dispositivos.

En resumen, las ventajas clave de NETADAPT incluyen la capacidad de reducir significativamente el tamaño de los modelos, adaptarse a las características de hardware específicas de cada dispositivo, y lograr mejoras notables en la latencia, especialmente cuando el modelo donde se realiza la inferencia es el dispositivo objetivo durante el proceso de optimización.

Capítulo 5

Optimizaciones en NetAdapt

Los algoritmos de optimización de NNs abordados en los capítulos anteriores pueden ser muy demandantes desde el punto de vista del costo computacional. En este sentido, el costo de optimizar la red es una limitante en sí misma, ya que de poco sirve un algoritmo de optimización varios órdenes de magnitud más costoso que utilizar una solución no optimizada. Por esta razón, el segundo gran esfuerzo de la tesis fue dedicado a desarrollar implementaciones más eficientes de una de las herramientas de optimización de NNs empleadas por la comunidad y estudiadas en la evaluación experimental resumida en el capítulo anterior. Específicamente, se busca optimizar el tiempo de ejecución de la implementación oficial con PyTorch del NetAdapt [89] a través de distintas propuestas evaluadas de forma incremental.

El resto del capítulo se organiza de la siguiente manera, en primer lugar se realiza un análisis más profundo del NetAdapt, intentando entender los parámetros que influyen en los tiempos de ejecución. Luego se describe el entorno para las pruebas y se especifican los experimentos realizados sobre la implementación original de NetAdapt. Por último, se desarrollan y evalúan algunas propuestas de mejora, abriendo caminos para la optimización de esta herramienta y la implementación de soluciones con redes neuronales optimizadas.

5.1. Análisis de la herramienta

Como se mencionó anteriormente, NetAdapt se basa en un algoritmo iterativo que, en cada paso, genera un conjunto de redes candidatas (tantas

como capas simplificables tenga el modelo), las entrena y evalúa, y finalmente elige la que tenga mejor desempeño teniendo en cuenta un balance entre la precisión y la latencia. En el capítulo anterior no se hizo especial énfasis en los posibles ajustes disponibles a la hora de ejecutar NetAdapt. Existen varios parámetros que influyen sobre el tiempo de cómputo, tanto del proceso de optimización, como de las redes neuronales resultantes.

Algoritmo 1 Resumen del algoritmo NetAdapt

1: $bestModel \leftarrow$ Modelo pre-entrenado (semilla).
2: $step \leftarrow 0$
3: **while** $step < MAXITER$ **y** $latencia(bestModel) > budget$ **do**
4: Generar un conjunto de modelos candidatos (tantos como capas simplificables)
5: **for each** candidato **do (*parallel*)**
6: Simplificar el modelo con podas (reducir la latencia en al menos δ)
7: Entrenar el modelo candidato por T iteraciones (Fine-tune).
8: Evaluar y guardar el accuracy del modelo.
9: **end for**
10: $bestModel \leftarrow$ Seleccionar el mejor candidato (en latencia y accuracy).
11: $step \leftarrow step + 1$
12: **end while**

Una versión simplificada del algoritmo para NetAdapt se presenta en el Algoritmo 1. El proceso de simplificación comienza con un modelo inicial (línea 1), seguido por un bucle que itera hasta alcanzar el número máximo de iteraciones (parámetro de NetAdapt) o hasta que se alcance la latencia objetivo (por ejemplo, la mitad de la latencia original) (línea 3). Dentro de este bucle el primer paso es generar los modelos candidatos, es decir, copias independientes del mejor modelo para la iteración actual (línea 4). Estos candidatos se simplifican, entrenan y evalúan de manera independiente en el bucle interno (línea 5). Como este proceso es independiente de cada modelo, en la implementación se simplifican, entrenan y evalúan al mismo tiempo tantos modelos como GPUs disponibles existan en el entorno de ejecución. Esto se realiza con un sistema de *workers* y una cola. Específicamente, cada worker está asociado a un candidato, cuando se libera una GPU se le asignan los recursos para la ejecución al primer worker de la cola. El proceso de simplificación (línea 6) consiste en podar el modelo candidato hasta que se alcance una reducción de latencia de al menos δ. Por ejemplo, $\delta = 0{,}05$ significa que la reducción de latencia de los modelos candidatos será de $0{,}05 \times |L|$ o un

5 % de la latencia original, en cada iteración del algoritmo. El cálculo de la latencia para los modelos se realiza mediante la LUT de latencia tomando la entrada correspondiente a cada configuración o si la configuración no fue medida, interpolando a partir de los valores en la LUT (ya que evaluar todas las combinaciones posibles puede tomar en el orden de semanas).

El proceso de calcular la latencia a partir de la LUT está documentado en el Algoritmo 3. Una vez obtenida la configuración que satisface la restricción de latencia del paso, el algoritmo procede a entrenar el modelo candidato durante T iteraciones (línea 7) para tener una precisión comparable al modelo original después de la poda. El último paso de este bucle es evaluar el modelo y almacenar su precisión (línea 8). Una vez que se evalúan todos los modelos propuestos, se comparan para seleccionar aquel con la mejor relación $\frac{\Delta\text{Precisión}}{\Delta\text{Latencia}}$ con respecto al mejor modelo de la iteración anterior.

Es notorio que la herramienta cuenta con varios parámetros, y la definición de estos influye en el tiempo de cálculo del algoritmo. Concretamente, T controla el número de iteraciones de *fine-tuning* que se realizan sobre los modelos candidatos, y debe ser mayor para conjuntos de datos más grandes o modelos con muchos pesos a ajustar. También está el parámetro δ, el cual hace referencia a la tasa de reducción de latencia por iteración. Valores más grandes de este parámetro implican podas más importantes para cumplir con la restricción de latencia en cada iteración. Aunque el resultado puede no ser tan bueno como con un δ pequeño, se necesita menos cálculo para δ más alto, ya que la restricción se alcanza en menos iteraciones.

Para tomar como ejemplo de uso de parámetros, en el estudio de Howard et al. [54], NetAdapt se utiliza junto con un algoritmo NAS para la búsqueda de una nueva versión de MobileNet (V3). Específicamente, los autores aprovechan el algoritmo para reducir el tamaño de la arquitectura encontrada por el NAS, resultando en una versión más compacta del modelo. El trabajo mencionado, establece los parámetros $T = 10.000$ (para ImageNet) y $\delta = 0{,}01$, reduciendo el 1 % de la latencia del modelo inicial en cada iteración. Cabe destacar que para emplear un δ "pequeño" y un T "grande", la infraestructura donde se ejecuta tiene que ofrecer gran capacidad de cómputo (en dicho trabajo los modelos se entrenaron en un TPU Pod de 4×4).

Otro aspecto importante del algoritmo es la construcción de la LUT para cada dispositivo. Se cuenta con una rutina encargada de realizar las mediciones de latencia para las diferentes configuraciones posibles que puede tener la red.

Específicamente, para cada capa simplificable (en este caso capas Convolucionales y Fully Connected) partiendo de la configuración original, se reduce la dimensionalidad de dicha capa de forma lineal en una cantidad fija dada por los parámetros *min_conv_feature_size* y *min_fc_feature_size*.

Si se considera a las capas como cajas que parten con ciertas dimensiones de entrada (*num_in_channels*), y generan una salida de dimensiones (*num_out_channels*), el algoritmo de evaluación primero fija las dimensiones de entrada y luego variando la cantidad de canales de salida realiza las mediciones hasta la mínima cantidad de canales de salida. Luego, se vuelve a repetir este proceso pero ahora reduciendo la cantidad de canales de entrada, midiendo nuevamente las distintas configuraciones variando las dimensiones de salida. Esto se repite hasta el mínimo de canales de entrada. El procedimiento descrito implica un muestreo (ya es discreto) del espacio de evaluación debido a que los espacios pueden ser muy extensos dependiendo de las dimensiones de las capas. Por lo tanto, la elección del *min_feature_size* (generalizando para las distintas capas) es crucial para poder computar la LUT en tiempos razonables, más teniendo en cuenta que se tiene que computar el promedio de varios cálculos para tener más certeza en las mediciones.

Algoritmo 2 Construcción de la LUT de latencias

```
1:  LUT ← {}                                   ▷ Estructura de datos para la Lookup Table
2:  simplifiable_layers ← get_simplifiable_layers(network)
3:  for all layer in simplifiable_layers do
4:      LUT[layer] ← {}                        ▷ Inicializar entrada para la capa en la LUT
5:      layer_properties ← get_layer_properties(layer)
6:      for in_channels from layer_properties['num_in_channels'] to 1 with
    step min_feature_size do
7:          for out_channels from layer_properties['num_out_channels'] to 1
    with step min_feature_size do
8:              latency_measure ← measure_latency(layer, in_channels, out_channels)
9:              LUT[layer][(in_channels, out_channels)] ← latency_measure
10:         end for
11:     end for
12: end for
```

El Algoritmo 2 representa la rutina descrita anteriormente en forma de pseudocódigo.

5.2. Entorno de ejecución

En esta etapa del trabajo se empleó el mismo servidor descrito en la Sección 4.2. Los experimentos se limitaron a generar modelos específicamente optimizados para la plataforma Nvidia Jetson Xavier[1], utilizando PyTorch CUDA.

Los parámetros utilizados para la ejecución de NetAdapt, tanto para la versión original como las variantes propuestas, fueron $T = 500$ (razonable para las dimensiones y complejidad de CIFAR10) y para el valor de δ, que controla la reducción de latencia de los candidatos en cada paso, se utilizaron los valores $\{0{,}01, 0{,}025, 0{,}05\}$[2]. La latencia objetivo de los modelos, es decir el budget, fue de $0{,}5 \times L$ (la mitad de la latencia original).

5.3. Evaluación experimental de NetAdapt

El costo de computar la versión original de NetAdapt se puede observar en la Tabla 5.1. También, se puede notar que el tiempo de ejecución varía según el modelo y el parámetro δ. Para MobileNet el tiempo transcurrido se correlaciona con el número de iteraciones en todos los casos (alrededor de 15 minutos por iteración). Por otro lado, para AlexNet, se puede observar que para un valor de δ más alto, el objetivo de latencia se alcanza, como era de esperar, en menos iteraciones. Sin embargo, el tiempo transcurrido no se relaciona directamente con estos resultados, ya que para 0,01 el promedio por iteración es de 32 minutos, para 0,025 es de 69 y para 0,05 es de 101 minutos. Esto se debe principalmente a que cuanto mayor es el valor de δ, más veces se estima la latencia del modelo en búsqueda de una configuración que satisfaga las restricciones.

Los costos del cómputo del NetAdapt se distribuyen principalmente en la búsqueda de la configuración que satisfaga la restricción de latencia en el paso para el bloque que se está optimizando, el entrenamiento luego de realizadas las podas, y la evaluación de la precisión. En cada iteración de NetAdapt, entre los modelos candidatos, estas etapas se pueden computar en paralelo. Por lo tanto, el mejor de los casos se da cuando se cuenta con tantas GPUs,

[1] Notar que, en este caso, la única parte en la que influye el dispositivo de destino es en la generación de la tabla de latencias.

[2] Es decir 1 %, 2,5 % y 5 % de reducción en cada paso.

Tabla 5.1: Tiempo de ejecución (en segundos) e iteraciones para computar NetAdapt (original) para distintos valores de δ, con $T = 500$.

	MobileNet		**AlexNet**	
δ	Tiempo de ejecución	Iteraciones	Tiempo de ejecución	Iteraciones
0,01	$3{,}434 \times 10^4$	40	$1{,}343 \times 10^5$	68
0,025	$1{,}662 \times 10^4$	18	$1{,}003 \times 10^5$	24
0,05	$1{,}010 \times 10^4$	11	$7{,}282 \times 10^4$	12

como modelos candidatos, permitiendo poder simplificar, entrenar y evaluar todos los candidatos al mismo tiempo (no es el caso de este trabajo).

Las Figuras 5.1 y 5.2 presentan un desglose de los tiempos de ejecución (promedio) de las etapas más demandantes, divididas por bloque optimizable de AlexNet. Comparando los resultados para los distintos valores de δ se puede

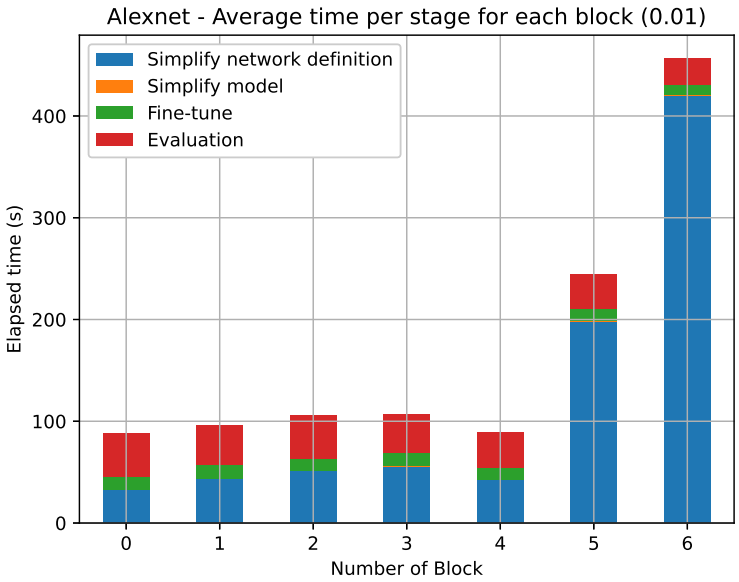


Figura 5.1: Distribución de tiempos promedio de ejecución (en segundos) por etapa en NetAdapt (original) para AlexNet divididos por bloques simplificables. $\delta = 0{,}01$.

observar cómo este afecta significativamente el tiempo necesario para computar la definición simplificada de AlexNet, principalmente en los últimos dos bloques, asociados a las capas Fully Connected, las de mayores dimensiones. Por otro lado, las etapas de entrenamiento y evaluación ocupan una gran parte del cómputo debido a que la ejecución de inferencias, así como la propagación hacia atrás en el caso del fine-tune, son operaciones costosas. Además de estas fases, es importante notar que también se destaca el tiempo empleado en sim-

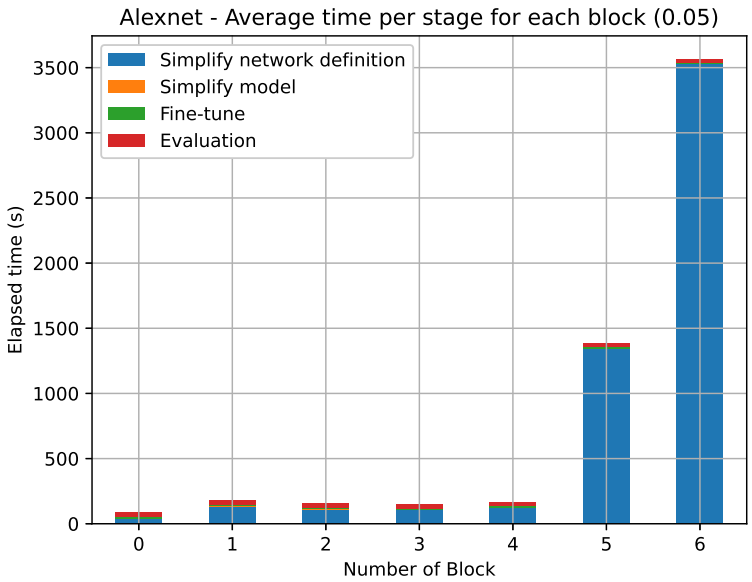


Figura 5.2: Distribución de tiempos promedio de ejecución (en segundos) por etapa en NetAdapt (original) para AlexNet divididos por bloques simplificables. $\delta = 0{,}05$.

plificar la red para cumplir con los requerimientos de latencia. En particular, notar que para el valor de $\delta = 0{,}05$ el costo de simplificar la red pasa a ser incomparable con los tiempos de las demás etapas. El problema surge cuando los puntos a interpolar son muchos, como en el caso de las capas FC en AlexNet, con dimensiones (4096,9126).

Para poder entender el por qué del alto costo que presenta la etapa de búsqueda de la configuración, el Algoritmo 3 presenta el procedimiento con mayor detalle. Básicamente, el algoritmo `compute_latency` se encarga de computar la latencia total de un modelo de red neuronal tomando en cuenta su definición (capas y cantidad de filtros) y una LUT, la cual fue computada previamente sobre el dispositivo para el que se quiere optimizar el modelo. Para ello, se recorren todas las capas de la definición de la red, se obtiene la cantidad de canales de entrada y salida *(in_channels,out_channels)* y seguidamente se evalúa la existencia de una medición en la LUT para dicha configuración. En caso de no existir, la latencia es estimada a través de una interpolación utilizando todos los puntos de tres dimensiones *((in_channels,out_channels) : measure)* asociados a dicha capa. Una vez obtenida la función de interpolación, se estima la latencia y se agrega al total. Es importante destacar que si la definición de la red para la cual se calculó la latencia no cumple con la restric-

Algoritmo 3 Cómputo de la latencia a partir de la LUT

```
procedure COMPUTE_LATENCY(net_def, lut_path)
    latency ← 0.0
    latency_lut ← load(lut_path)
    for layer in net_def do
        in_chs, out_chs ← get_num_channels(layer)
        if (in_chs, out_chs) in lookup_table(layer) then
            elat ← lookup_table(layer)(in_chs, out_chs)
        else
            lay_int ← interpolate(lookup_table(layer))
            elat ← lay_int(in_chs, out_chs)
        end if
        latency ← latency + elat
    end for
    return latency
end procedure
```

ción de la iteración, se vuelve a repetir el proceso. En este caso, se genera una nueva configuración reduciendo la dimensionalidad del bloque asignado y se recalcula la latencia utilizando el procedimiento `compute_latency`. Se puede observar cómo este enfoque presenta un problema significativo, y es el del costo asociado con la simplificación de la definición de la red. Si existe un bloque cuyas simplificaciones no derivan en una red que cumpla con la restricción y, además, presenta configuraciones que no se encuentran en la LUT, se llevan a cabo cálculos repetidos para computar la función de interpolación.

5.4. Búsqueda por bipartición

Para reducir el costo asociado a la simplificación de la red, se propuso una estrategia diferente de evaluación. En lugar de reducir linealmente la cantidad de filtros en una cantidad fija dada por *min_feature_size*, se optó por realizar una búsqueda binaria o por bipartición. La premisa para poder aplicar esta estrategia es que la latencia, o tiempo de ejecución, depende linealmente de la cantidad de filtros, es decir, a mayor cantidad de filtros, mayor es el tiempo de ejecución, lo que permitiría asumir cierto orden entre las configuraciones. Las Figuras 5.3 y 5.4 sustentan esta hipótesis, al menos para las redes evaluadas.

Parte del algoritmo utilizado se muestra en la Figura 5.5. Se puede apreciar que es equivalente a un algoritmo de búsqueda por bipartición tradicional,

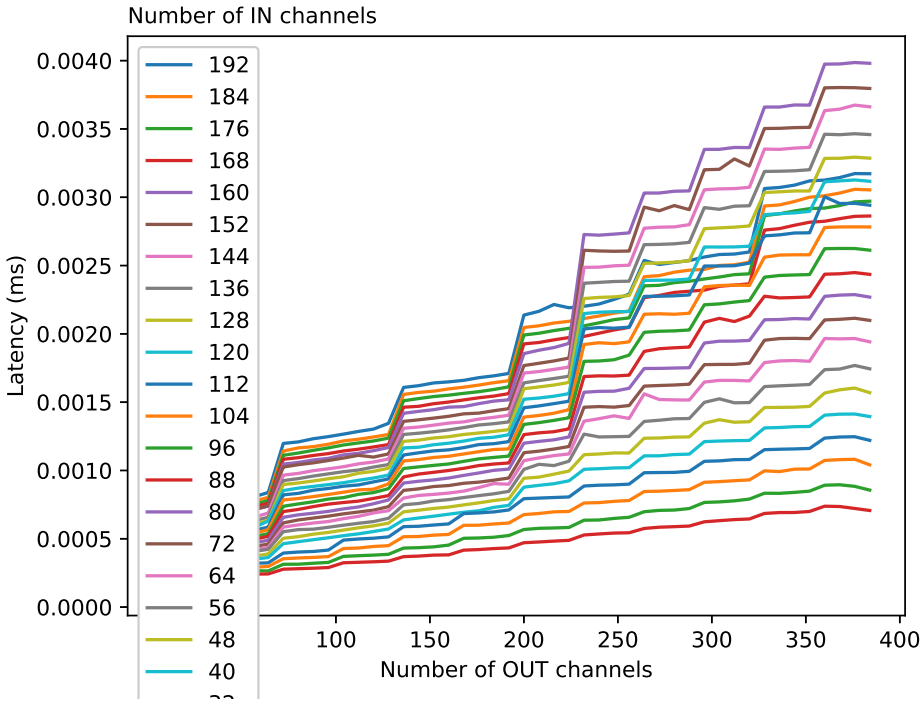


Figura 5.3: Latencia (en ms) de una capa convolucional variando la cantidad de canales de entrada y salida, en el NVIDIA Jetson AGX Xavier.

agregando un parámetro extra ***threshold***, o un umbral, que es empleado como condición de parada de la búsqueda. Es decir, cuando se llega a un vector de dimensiones menores o iguales al *threshold*, la búsqueda por bipartición termina y se realiza una evaluación secuencial sobre esta última porción del espacio.

Para clarificar aún más la idea, la Figura 5.6 presenta un caso muy simplificado de búsqueda por bipartición. En este ejemplo, cada fila representa el vector de las medidas de latencia (los índices arriba del primer vector indican las dimensiones de la capa medida) y su paso por la bipartición. Notar que la primer fila tiene todas las entradas en azul (menos una, la que evalúa la el algoritmo de bipartición) que son los posibles valores a evaluar. Paso a paso se va reduciendo la cantidad de entradas a evaluar. En lugar de tener que realizar 10 evaluaciones para encontrar la restricción 0,5, sólo son necesarias 5. Si se generaliza el problema para uno de tamaño n, siendo n la cantidad de dimensiones posibles a evaluar para la capa a simplificar, en el peor de los casos para la búsqueda secuencial derivaría en un orden de ejecución lineal $O(n)$. En cambio, utilizando la búsqueda modificada por bipartición con el umbral la complejidad se reduce a $O(log(n) + threshold)$. Esto es especialmente

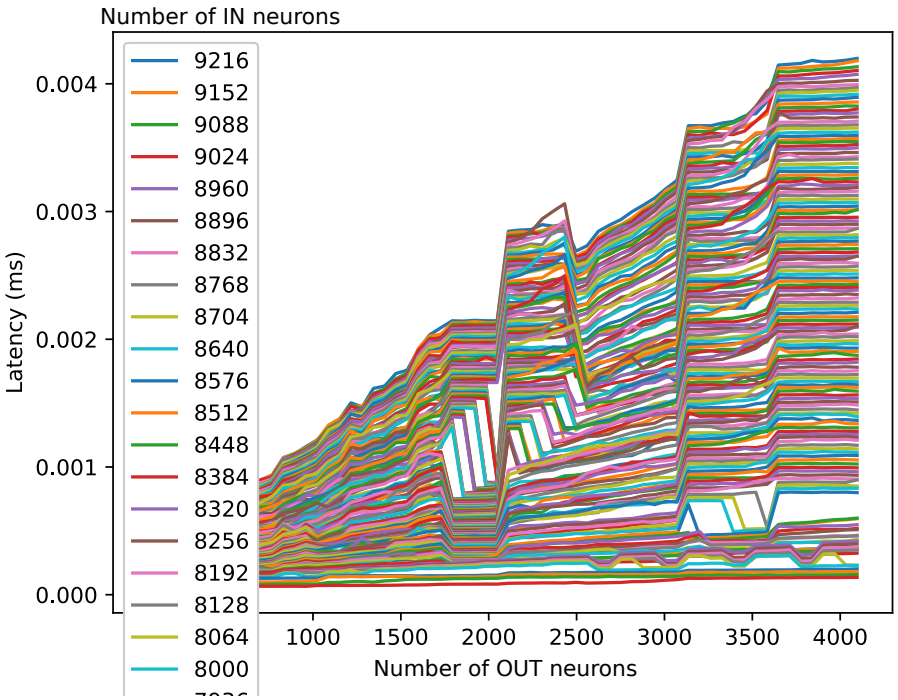


Figura 5.4: Latencia (en ms) de una capa Fully Connected variando el tamaño de entrada y salida, para un Nvidia Jetson AGX Xavier.

importante cuando la cantidad de canales distintos a evaluar es grande.

5.4.1. Resultados

En las Tablas 5.2 y 5.3 se puede observar el resultado de sustituir en NetAdapt la búsqueda lineal por la bipartición (con umbral). En cuanto a la ejecución del algoritmo sobre MobileNet (Tabla 5.2) no se observan mejoras, lo que es consistente con el análisis hecho en las secciones previas, donde el tiempo promedio por iteración no presentaba variaciones al emplear diferentes δ. En cambio, para AlexNet (Tabla 5.3) las mejoras empleando distintos valores de δ van desde $\sim 3\times$ hasta $7\times$.

5.5. Interpolación

El procedimiento de interpolación para estimar los valores de latencia que no se encuentran en la LUT es invocado múltiples veces para los diferentes workers durante la ejecución del algoritmo (ver el Algoritmo 3), y particular-

```
threshold = 16
left = 0
right = len(num_out_channels_try) - 1
while right - left > threshold:
    mid = (right + left)//2
    current_num_out_channels =
        num_out_channels_try[mid]
    # Get the current resource consumption
    simplified_resource = compute_resource(
                            simplified_network_def,
                            resource_type,
                            lookup_table)
    if simplified_resource < constraint:
        right = mid
    else:
        left = mid
num_out_channels_try =
    num_out_channels_try[left:right+1]
```

Figura 5.5: Pseudocódigo (porción simplificada de *python*) del nuevo algoritmo de búsqueda para alcanzar la restricción de latencia de la iteración.

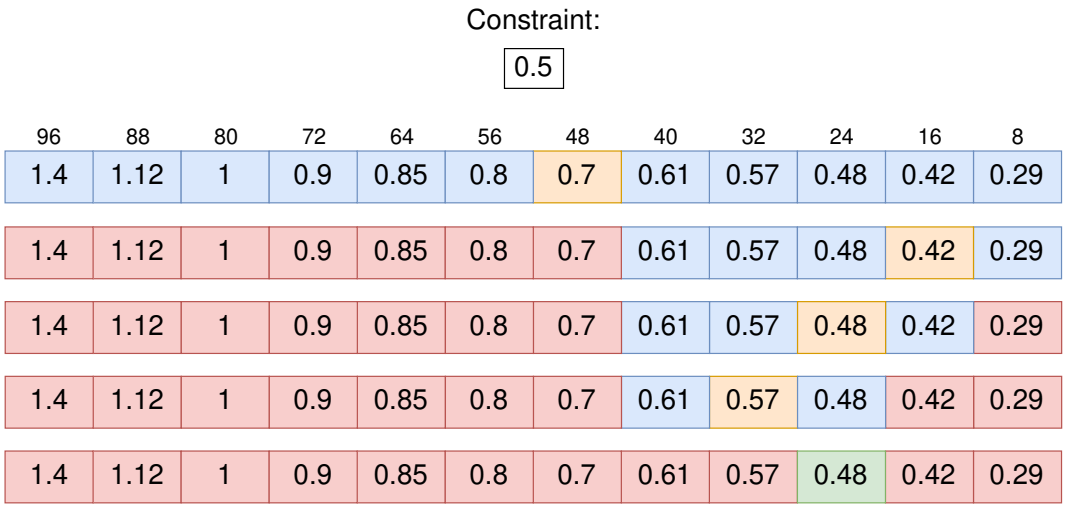


Figura 5.6: Ejemplo de búsqueda por bipartición con un *threshold* $= 1$, intentando alcanzar la restricción 0,5.

Tabla 5.2: Tiempo de ejecución (en minutos) para computar NetAdapt en sus dos versiones: Original y utilizando la búsqueda por bipartición, para distintos valores de δ y $T = 500$, sobre MobileNet.

δ	**Original**	**B. por bipartición**	**SpeedUp**
0,01	572	622	0,9×
0,025	270	268	1,0×
0,05	168	167	1,0×

Tabla 5.3: Tiempo de ejecución (en minutos) para computar NetAdapt en sus diferentes versiones: Original y utilizando la búsqueda por bipartición, para distintos valores de δ y $T = 500$, sobre AlexNet.

δ	**Original**	**B. por bipartición**	**SpeedUp**
0,01	2239	798	2,8×
0,025	1672	319	5,2×
0,05	1214	174	7,0×

mente la interpolación es siempre calculada con los mismos datos medidos. Para capas de alta dimensionalidad, como la mayoría de las FC de los modelos evaluados, son muchos los puntos de tres dimensiones de la forma *(in_features, out_features) : measure* que deben ser interpolados, esto puede llevar en el orden de varios segundos, que acumulados a través de repetidas iteraciones supone un costo computacional no despreciable.

5.5.1. Propuesta

Considerando los resultados del apartado anterior se propone computar una única vez las interpolaciones al inicio de la ejecución del NetAdapt. Más en detalle, se almacena la función usada para calcular las interpolaciones en un archivo binario, permitiendo así que sea cargada por los distintos workers a lo largo de las iteraciones en el algoritmo. En el Algoritmo 3 se puede observar cómo el parámetro para computar la latencia corresponde a una dirección de un archivo (*path*) almacenado en un formato serializable (en este caso, `pickle`) y no a una variable con los datos ya cargados de la LUT. Por lo tanto, para cada medición o estimación de la latencia que hace uso de la LUT es necesaria la lectura de dicho archivo. La lectura de archivos, en muchas ocasiones, puede ser determinante en en el desempeño de los programas introduciendo cierto sobrecoste. Varios son los factores que pueden afectar cuán importante es el impacto sobre el tiempo total del programa, entre ellos: comunicaciones a

memorias lentas, como el disco, el tamaño del archivo leído y también el tiempo de des-serialización. Por estas razones, la otra estrategia propuesta en este apartado que permitiría reducir el costo de estos accesos es, en lugar de usar un *path* al archivo, usar directamente la variable con el resultado de cargar la LUT.

5.5.2. Resultados

Las Tablas 5.4 y 5.5 presentan los resultados de las ejecuciones junto con los speed-ups alcanzados empleando la estrategia de pre-computar la interpolación, comparados con la versión original. Rápidamente se puede observar cómo para MobileNet este enfoque no produce mejoras, mientras que para AlexNet, para los valores de δ evaluados se aprecia una mejora creciente, desde $\sim 5\times$ para 0,01, hasta $\sim 15\times$ para 0,05.

Tabla 5.4: Tiempo de ejecución (en minutos) para computar NetAdapt en sus diferentes versiones: Original y utilizando la pre-Interpolación, para distintos valores de δ y $T = 500$, sobre MobileNet.

	Original	**Interpolación**	**SpeedUp**
0,01	572	585	1,0×
0,025	270	272	1,0×
0,05	168	167	1,0×

Tabla 5.5: Tiempo de ejecución (en minutos) para computar NetAdapt en sus diferentes versiones: Original y utilizando la pre-Interpolación, para distintos valores de δ y $T = 500$, sobre AlexNet.

	Original	**Interpolación**	**SpeedUp**
0,01	2239	468	4,8×
0,025	1672	137	12,2×
0,05	1214	82	14,8×

Las Tablas 5.6 y 5.7 muestran el tiempo promedio empleado en la etapa de simplificación del modelo combinando bipartición y la reutilización de la carga del archivo de la LUT con la interpolación.

Los resultados de la Tabla 5.6 permiten enfatizar el fuerte impacto en el rendimiento de la nueva estrategia para hacer los cálculos de interpolación, reduciendo enormemente el costo de la búsqueda de definición de arquitectura simplificada, con una mejora de 2060× para el último bloque FC de AlexNet.

Tabla 5.6: AlexNet - Comparación entre los tiempos promedio de ejecución (en segundos) para simplificar la definición de red por bloque con las diferentes estrategias.

Bloque	Estrategia			
	Original	Interpolación	B. por bip. (+Int.)	Archivo (+Int.)
0	43,927	0,040	0,046	0,033
1	74,880	0,066	0,070	0,034
2	73,461	0,054	0,071	0,030
3	77,266	0,076	0,088	0,043
4	71,977	0,048	0,055	0,027
5	868,283	0,371	0,111	0,072
6	1586,191	0,770	0,070	0,122

Por otro lado, en la Tabla 5.7 puede observarse cómo para MobileNet no hubo aceleraciones en esta etapa y, en algunos casos, la estrategia con la interpolación precalculada es peor por una pequeña fracción. Estos resultados indican que el procedimiento de interpolación original es bastante económico para MobileNet debido a su LUT más pequeña y por ende la menor cantidad de configuraciones posibles a evaluar. Por lo tanto, la optimización de la búsqueda de definición de arquitectura simplificada no afecta a esta red.

Para profundizar en los resultados obtenidos para AlexNet, se incluyó la evaluación de otra red de grandes dimensiones, VGG19 [109]. En este caso, los resultados presentados en la Tabla 5.8 debido a los tiempos elevados, no son promedios sino el costo de una sola iteración. Estos resultados permiten dar una idea de cuán costoso es computar NetAdapt para redes grandes. Específicamente, cuando no se tienen todos los valores de latencia medidos, i.e., es necesario interpolar, el comportamiento es bastante similar al de AlexNet, donde las últimas capas FC suponen un costo extensamente superior con el resto. A diferencia de MobileNet, AlexNet tiene entradas de mayor dimensión en las últimas capas, y VGG19, con una capa de 25088 entradas y 4096 neuronas de salida, tiene 25088 puntos para interpolar cada vez que no se encuentra una configuración en la LUT de latencia. Además, se necesitan más interpolaciones para otras capas según las conexiones y cómo la reducción de una capa afecta a otras.

Tabla 5.7: MobileNet - Comparación entre los tiempos promedio de ejecución (en segundos) para simplificar la definición de red por bloque con las diferentes estrategias.

Bloque	Estrategia			
	Original	Interpolación	B. por bip. (+Int.)	Archivo (+Int.)
0	0,035	0,032	0,039	0,037
1	0,050	0,059	0,051	0,034
2	0,054	0,054	0,057	0,032
3	0,066	0,074	0,072	0,026
4	0,085	0,087	0,095	0,027
5	0,104	0,104	0,081	0,027
6	0,169	0,171	0,096	0,027
7	0,137	0,139	0,073	0,028
8	0,143	0,140	0,067	0,028
9	0,143	0,141	0,071	0,031
10	0,149	0,137	0,066	0,028
11	0,197	0,187	0,123	0,027
12	0,371	0,399	0,077	0,031
13	0,657	0,684	0,091	0,036

5.6. Evitar el entrenamiento de la red

Otro costo importante en cada iteración de NetAdapt es el asociado al entrenamiento (*fine-tuning*) y evaluación de los modelos candidatos. Esto requiere múltiples ejecuciones del modelo, así como recorridas de retropropagación. Dependiendo del conjunto de datos y la arquitectura de la red, estas operaciones conllevan una gran cantidad de movimiento de datos y cálculos. Este costo se aborda generalmente con una gran cantidad de GPUs. Por ejemplo, existen algoritmos NAS que utilizan cientos de GPUs y semanas de cómputo [131]. En el caso de NetAdapt, la cantidad de candidatos se limita al número de capas simplificables y el número de iteraciones para el entrenamiento está dado por el parámetro T. Por lo tanto, si bien el T es ajustable y la cantidad de candidatos es mucho menor comparado con algoritmos NAS, el entrenamiento y evaluación representan un costo considerable en el proceso de optimización. Por ejemplo, sobre CIFAR10 con un tamaño de lote de 128 para AlexNet, la ejecución de 500 iteraciones tomó alrededor de 60 segundos. Aún así, si se escala este resultado a conjuntos de datos más grandes como ImageNet [34], donde la complejidad es mayor, el número de iteraciones debe aumentar. En [54], T se establece en 10000 iteraciones, lo que representa un aumento de al menos

Tabla 5.8: VGG19 - Comparación entre los tiempos promedio de ejecución (en segundos) para simplificar la definición de red por bloque con las diferentes estrategias.

Bloque	Estrategia			
	Original	Interpolación	B. por bip. (+Int.)	Archivo (+Int.)
0	0,034	0,035	0,071	0,031
1	0,037	0,037	0,043	0,033
2	0,050	0,048	0,058	0,026
3	0,090	0,095	0,102	0,027
4	0,091	0,094	0,103	0,027
5	0,091	0,091	0,078	0,026
6	0,091	0,089	0,140	0,029
7	0,167	0,169	0,075	0,030
8	0,151	0,149	0,058	0,030
9	0,215	0,237	0,119	0,031
10	0,409	0,409	0,139	0,027
11	0,426	0,380	0,146	0,027
12	10172,725	0,786	0,111	0,210
13	61980,530	8,808	0,272	1,519
14	1334,882	4,353	0,234	0,189

20× en el costo de entrenar los modelos candidatos.

5.6.1. Propuesta

En el contexto antes descrito, se propone aplicar conceptos de No-Training NAS para mejorar la eficiencia del algoritmo NetAdapt evitando, en la medida de lo posible, el entrenamiento y la evaluación de la precisión. Se busca introducir un enfoque *híbrido* para simplificar los modelos de redes neuronales. La idea es utilizar los estimadores de No-Training NAS como un mecanismo para elegir el candidato con mejor rendimiento en cada iteración de NetAdapt sin tener que reentrenar los candidatos y evaluarlos para conocer su precisión. En particular, se realiza una búsqueda inicial con estos estimadores (sin entrenamiento), y las iteraciones finales se realizan guiadas por la precisión (obtenida mediante entrenamiento y evaluación) como una forma de refinar la búsqueda. La idea es evaluar diferentes versiones del esquema híbrido utilizando distintos estimadores de No-Training NAS para la simplificación inicial del modelo. En particular, se utilizan estimadores de las dos principales familias presentes en la literatura. De los que utilizan la cantidad de regiones lineales, se seleccionó nas_wot [84], y para las que emplean el gradiente de la red

neuronal *grad_norm* [13] y *synflow* [117].

5.6.2. Resultados

En primer lugar, para validar la aplicación de estos estimadores como sustituto del cómputo de la precisión, se realizaron ejecuciones del NetAdapt original monitoreando los estimadores, con especial atención a su evolución con respecto a la latencia. Notar que la latencia es la característica que se busca optimizar sobre las redes. En particular, la hipótesis del algoritmo es que a medida que se poda la red para reducir la latencia, se reduce la precisión, por eso se busca el modelo con mejor balance entre latencia y precisión.

Las Figuras 5.7 y 5.8 muestran la evolución de los modelos candidatos a lo largo de las iteraciones en términos de latencia y los valores de nas_wot. Cada punto representa un modelo. La línea verde (la de mayor grosor) representa la trayectoria de NetAdapt o la puntuación del modelo seleccionado por NetAdapt en cada iteración. Aunque esta búsqueda está dirigida por la precisión, se puede observar que la trayectoria corresponde a un descenso regular de nas_wot, siguiendo una trayectoria general entre los bloques, donde además coincide que el modelo seleccionado en cada iteración en muchos casos se encuentra en valores más altos del estimador. También muestra cómo una reducción de la latencia (aumentando $1/L$) generalmente resulta en una reducción de los valores estimados para la calidad de la red.

Por otro lado, en la Figura 5.9 se puede observar la importante reducción de esfuerzo computacional por bloque que supone la sustitución de las etapas de entrenamiento y evaluación de la precisión por el cómputo del estimador nas_wot.

Siguiendo esta línea se evaluaron otros estimadores disponibles, estudiando los distintos desempeños en la tarea de clasificar los modelos sin el entrenamiento. En particular se realizaron pruebas con *grad_norm* y *synflow*. La evolución de *grad_norm* en NetAdapt guiado por la precisión se resume en las Figuras 5.10 y 5.11 .

A partir de estos resultados, se instancia una versión híbrida de NetAdapt guiado en los primeros pasos por los diferentes estimadores, con una última etapa de refinamiento guiada por la precisión que incluye etapas de *fine-tuning* intermedias. Los resultados para AlexNet se muestran en la Figura 5.14. En dicha figura se puede observar la precisión obtenida por el modelo optimizado

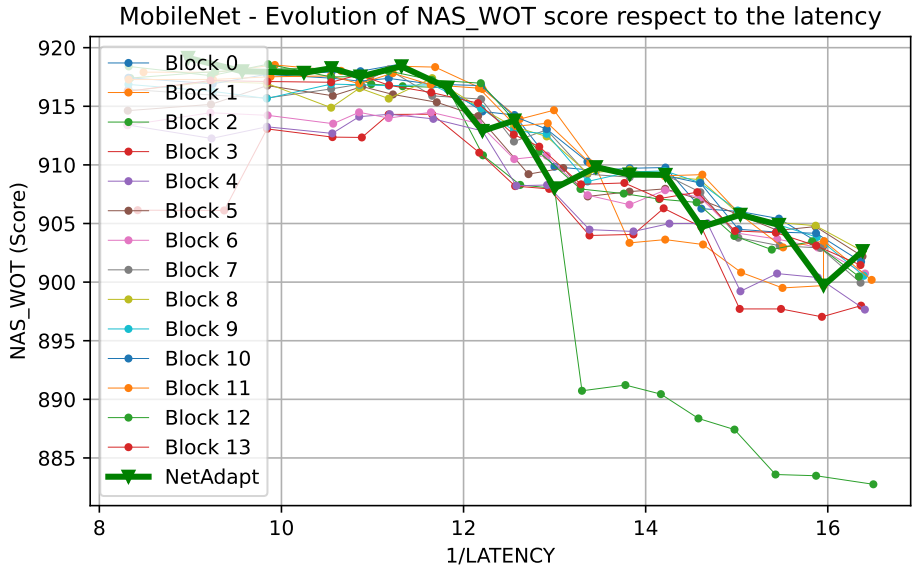


Figura 5.7: MobileNet ($\delta = 0{,}025$): Evolución de NAS_WOT versus latencia, a través de la ejecución de NetAdapt guiada por accuracy.

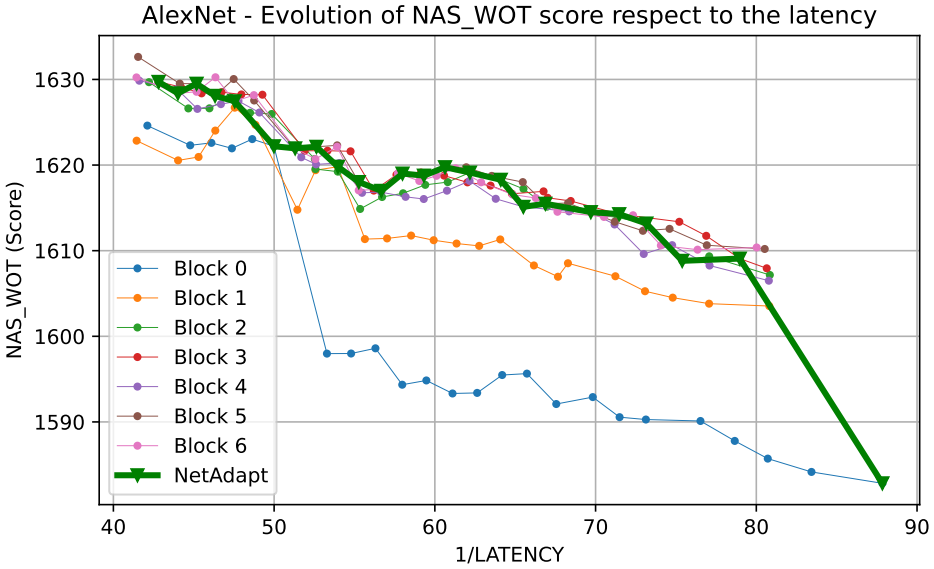


Figura 5.8: AlexNet ($\delta = 0{,}025$): Evolución de NAS_WOT versus latencia, a través de la ejecución de NetAdapt guiada por accuracy.

con cada estimador, además del tiempo necesario para la ejecución del proceso de optimización. HN, donde N es un número, referencia al mecanismo híbrido. Es decir, H30 equivale a la estrategia híbrida donde el último 30 % de las iteraciones para simplificar el modelo se realizan guiadas por la precisión.

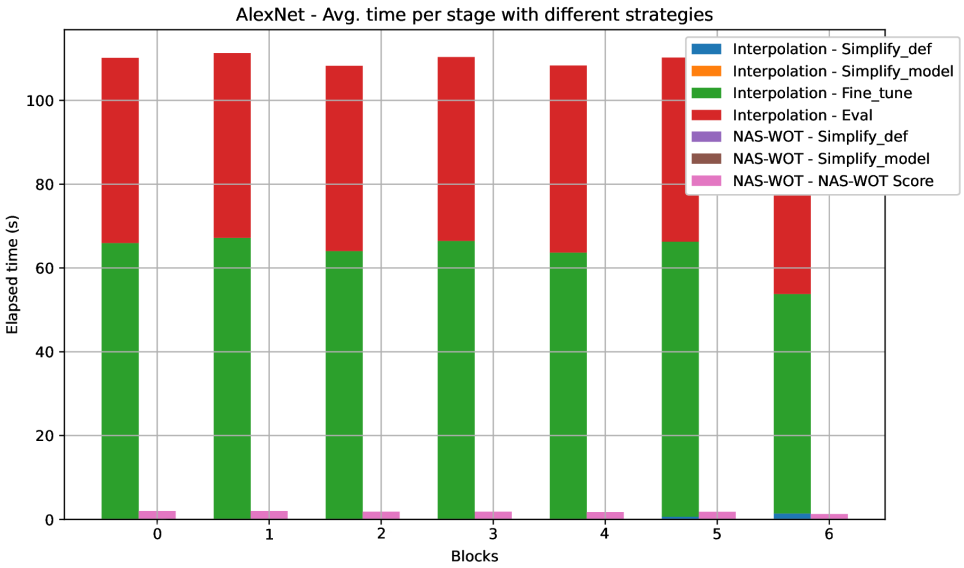


Figura 5.9: Tiempos promedio (en segundos) por etapa para cada bloque en el proceso del NetAdapt original vs. usando nas_wot como estimador. Con $T = 500$ y $\delta = 0{,}01$.

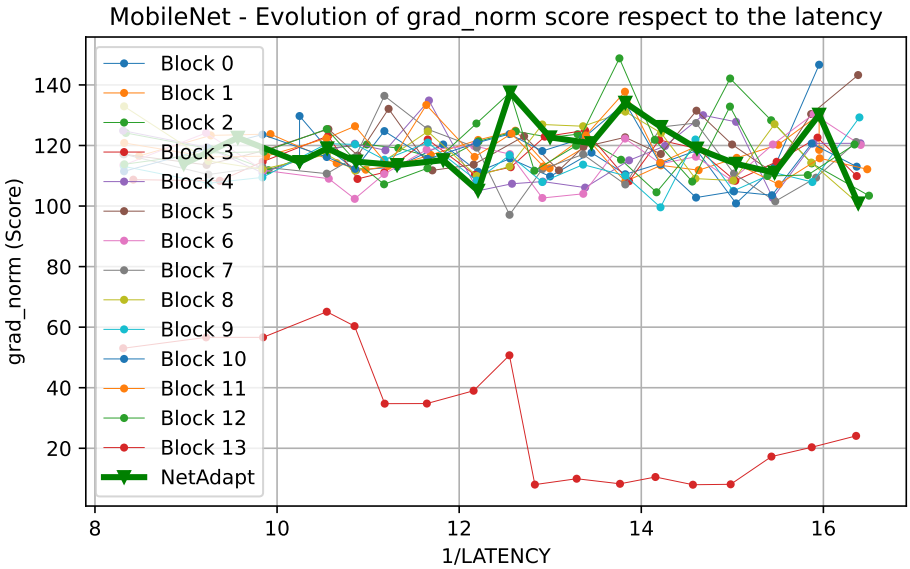


Figura 5.10: MobileNet ($\delta = 0{,}025$): Evolución de *grad_norm* versus latencia, a través de la ejecución de NetAdapt guiada por accuracy.

En primer lugar se puede destacar la notable diferencia en los tiempos de ejecución de los esquemas propuestos en este proyecto con respecto a la versión original (incorporando la estrategia de la pre-interpolación, Sección 5.5). En

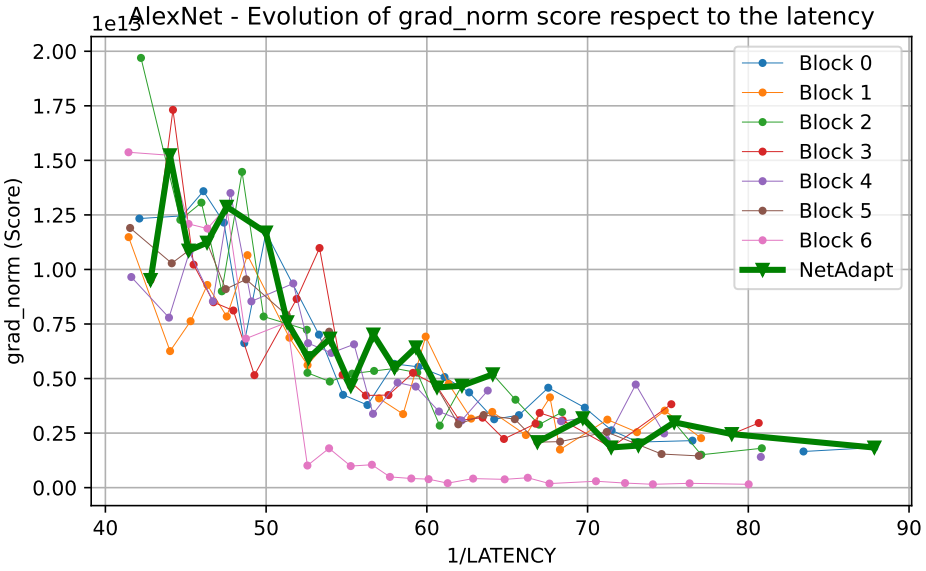


Figura 5.11: AlexNet ($\delta = 0{,}025$): Evolución de *grad_norm* versus latencia, a través de la ejecución de NetAdapt guiada por accuracy.

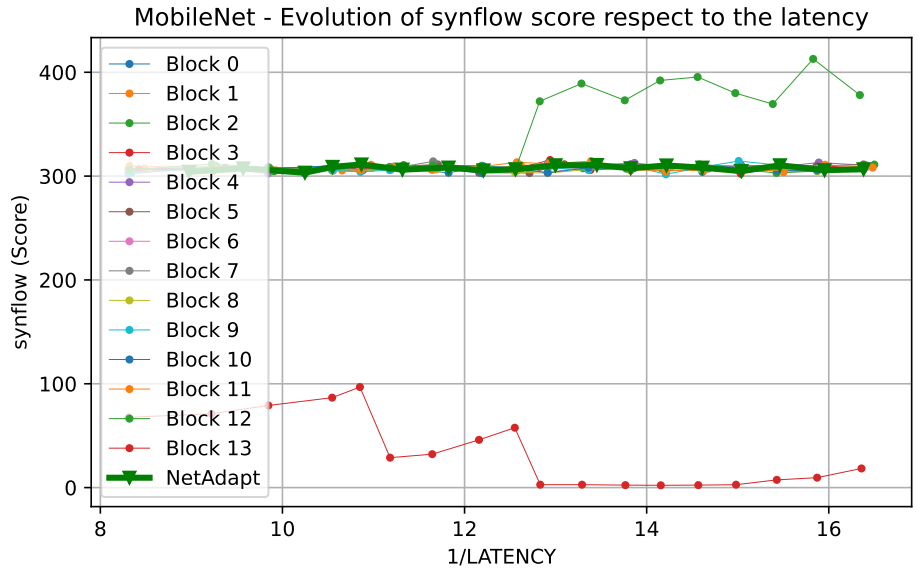


Figura 5.12: MobileNet ($\delta = 0{,}025$): Evolución de *synflow* versus latencia, a través de la ejecución de NetAdapt guiada por accuracy.

cuanto a la precisión obtenida para AlexNet con las estrategias híbridas, si bien es menor, es comparable con la obtenida con la implementación original de NetAdapt. Por ejemplo, la estrategia `H30_nas_wot` logró casi la misma precisión (una diferencia de 0.08 %), con una aceleración de 8× en compara-

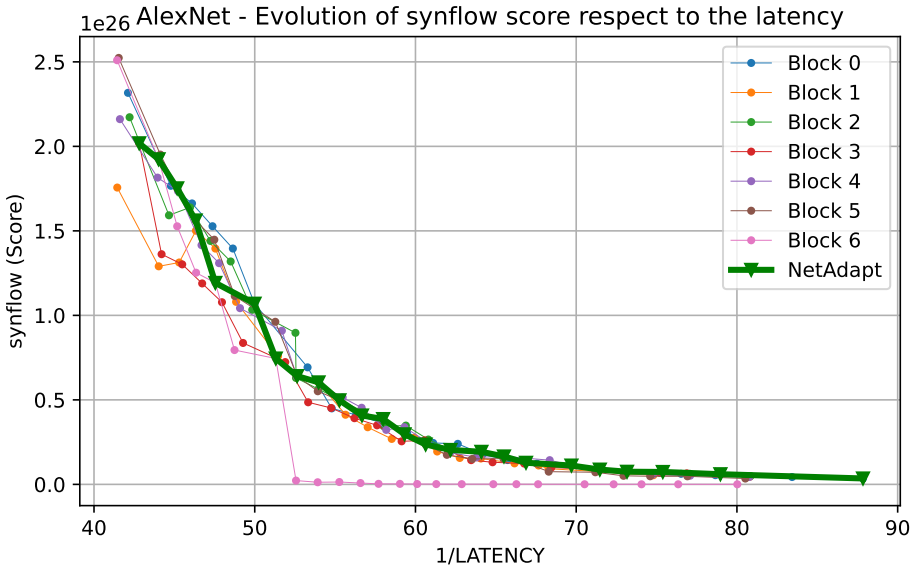


Figura 5.13: AlexNet ($\delta = 0{,}025$): Evolución de *synflow* versus latencia, a través de la ejecución de NetAdapt guiada por accuracy.

ción con la implementación original con pre-interpolación, y la mejora es de 37× si se compara con el tiempo original sin las mejoras planteadas.

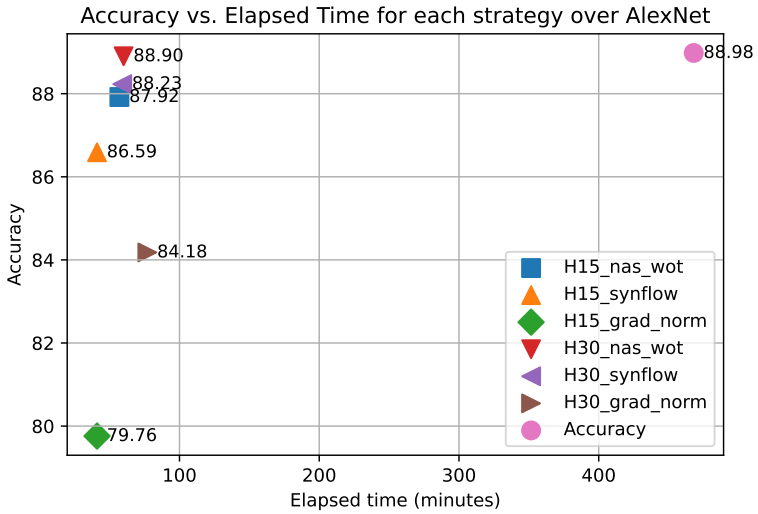


Figura 5.14: Accuracy versus Tiempo de ejecución (en minutos) para cada estrategia, con $T = 500$ y $\delta = 0{,}01$ sobre AlexNet.

La Figura 5.15 muestra resultados similares a los de AlexNet presentados antes pero para MobileNet. En este caso algunas de las estrategias híbridas incluso obtuvieron mejores precisiones al momento de inferir comparados con

el modelo obtenido con la versión original de NetAdapt (guiado solo por la precisión). Las mejoras en tiempo de ejecución de nuevo son notables.

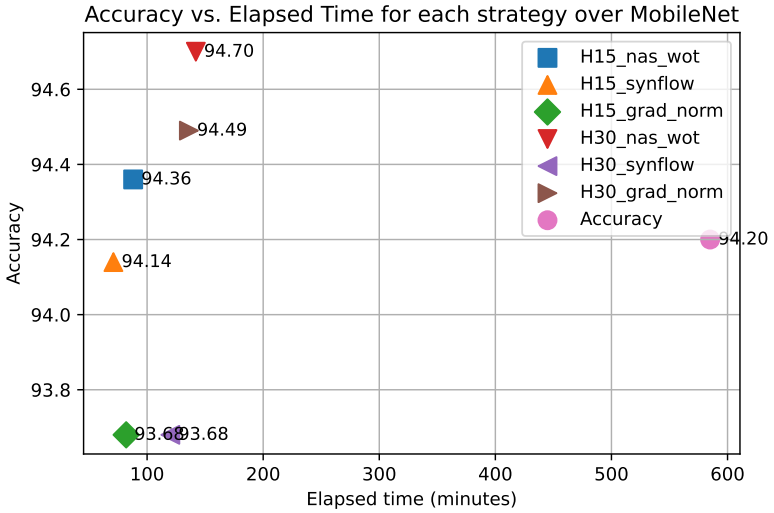


Figura 5.15: Accuracy versus Tiempo de ejecución (en minutos) para cada estrategia, con $T = 500$ y $\delta = 0{,}01$ sobre MobileNet.

5.7. Resumen del capítulo

Este capítulo se centró en abordar el costo computacional de NetAdapt, identificando sus principales rutinas, parámetros y cuellos de botella y proponiendo varias optimizaciones que permitieron acelerar el cómputo subyacente en la herramienta.

En primer lugar, se realizaron diferentes pruebas ajustando posibles parámetros que podrían permitir una aceleración en la ejecución del algoritmo. Esta prueba evidenció los importantes costos, además de crecimientos irregulares en los tiempos promedios de ejecución para los diferentes bloques simplificables, en particular para redes de mayores dimensiones. En este sentido, los costos que destacaron fueron los de las rutinas encargadas de la simplificación de los candidatos, el entrenamiento y la evaluación de la precisión de los modelos.

Primero se abordó el costo de simplificación de los modelos, creciente con el tamaño de δ, en particular para AlexNet, y de manera incremental se propusieron distintas estrategias para mitigarlo. Específicamente, se trabajó en sustituir la evaluación lineal de configuraciones para la capa asociada al candidato, por una variante de una evaluación por bipartición (asumiendo un orden

creciente en las medidas de latencia). Esto permitió reducir considerablemente el costo de simplificación.

Luego, a través de un análisis profundo de los procedimientos, se observó que gran parte del costo del proceso de simplificación derivaba de computar interpolaciones para configuraciones no medidas. En esta línea, se modificó el procedimiento para poder computar una única vez las interpolaciones, lo que derivó en una mejora del tiempo de ejecución entre 3× y 7×.

Por último, se enfocaron los esfuerzos en reducir el costo de las etapas restantes, es decir, el entrenamiento y la evaluación de la precisión. Para abordar esta parte, se buscó integrar estimadores derivados de algoritmos No-Training NAS. Se realizaron estudios sobre la evolución de los estimadores a través de la ejecución de NetAdapt, obteniendo resultados positivos. En base a los resultados obtenidos, se implementaron soluciones híbridas, guiando una primer etapa de las iteraciones de NetAdapt empleando estimadores (nas_wot, *synflow* y *grad_norm*), y para las últimas iteraciones, aplicando el esquema original de simplificación guiado por la precisión. En este caso, se lograron resultados notables, consiguiendo aceleraciones de hasta 37× con valores de precisión comparables.

Capítulo 6

Conclusiones y trabajo futuro

En este capítulo se detallan las conclusiones más importantes inferidas a través del trabajo realizado en esta tesis, se enumeran las publicaciones relacionadas, y se resumen algunas de las posibles líneas de trabajo futuro identificadas que permitirían extender el presente esfuerzo.

6.1. Conclusiones

El proyecto de tesis tuvo por objetivo principal avanzar en el estudio y comprensión de estrategias de optimización de redes neuronales en plataformas de hardware heterogéneas. Poniendo especial foco en la etapa de inferencia de las NNs, así como en algoritmos y técnicas que permitan optimizar las redes para su ejecución sobre plataformas de bajo porte y/o que impliquen consumos energéticos acotados.

En primera instancia, se destaca que en el desarrollo de la tesis se logró cumplir con los objetivos originalmente planteados. En particular, se realizó un relevamiento del estado del arte en materia de NNs, con especial foco en su cómputo eficiente. El relevamiento considera los diferentes métodos y técnicas que permiten optimizar las etapas involucradas en su cómputo, pero con especial foco en la etapa de inferencia. Se pudo observar cómo las optimizaciones propuestas van desde un enfoque centrado en el desarrollo de plataformas de hardware específicas para la ejecución de las redes neuronales, hasta optimizaciones por software para poder sacar el mayor provecho posible a los dispositivos de cómputo disponibles.

También se avanzó en el diseño y despliegue de un entorno de ejecución

para NNs, con el objetivo de evaluar técnicas eficientes de cómputo para estas rutinas. En particular, se lograron configuraciones de entornos para una gran variedad de plataformas de hardware de distintas características. Entre estos destacan dispositivos de bajo porte como el Intel NCS2 y el EdgeTPU de Google, así como también dos plataformas Jetson de Nvidia.

Sobre la base de los entornos alcanzados, se logró la adaptación de un algoritmo de optimización de NNs ampliamente reconocido en la comunidad, NetAdapt. Esta herramienta permite, partiendo de modelos existentes, adaptarlos a través de podas guiadas por medidas empíricas de latencia, explorando así modelos que tienen en cuenta la plataforma de hardware subyacente. Este algoritmo también fue combinado con frameworks desarrollados por los distintos fabricantes, así como técnicas de cuantificación usando tipos de datos como `INT8` y `FP16`. Las evaluaciones experimentales llevadas adelante arrojaron resultados variados, pero en general con importantes mejoras en los tiempos de cómputo.

Como conclusión de esta etapa se puede destacar que, para la gran mayoría de las investigaciones relevadas y experimentos realizados, estas herramientas de optimización y en particular NetAdapt, proveen interesantes resultados en términos de optimización, pero muchas veces el costo de aplicar estas técnicas es una restricción especialmente importante. Por esta razón, en el marco del proyecto se abordó la optimización de la herramienta NetAdapt.

En este contexto, uno de los principales aportes de la tesis es la propuesta de varias estrategias para acelerar NetAdapt, obteniendo importantes reducciones en los tiempos de cómputo. En particular, se aplicaron técnicas de búsqueda sobre datos ordenados, factorización de etapas ejecutadas múltiples veces sobre mismos datos y, por último, la integración de estimadores de calidad de modelos de NNs, para suplantar el uso de la precisión, reduciendo las evaluaciones y el *fine-tuning*. El avance en estas técnicas permitió optimizar el proceso de NetAdapt hasta 37× con resultados comparables de precisión.

6.2. Publicaciones

En el marco del trabajo de maestría se publicaron los siguientes trabajos:

- *"Evaluation of architecture-aware optimization techniques for Convolutional Neural Networks"* [81]. Este trabajo resume la exploración del uso de plataformas de hardware para el cómputo de redes neuronales, en conjunto con herramientas de optimización variadas. Los resultados de este artículo resumen los presentados en el Capítulo 4.
- *"Assessing the Performance of an Architecture-Aware Optimization Tool for Neural Networks"* [79]. El artículo muestra los primeros avances y optimizaciones propuestos para NetAdapt. Específicamente, se enfoca en el análisis de las principales rutinas que intervienen en NetAdapt y su costo computacional (Sección 5.3). Se desarrolla la propuesta de emplear una búsqueda binaria o por bipartición, presentada en la Sección 5.4.
- *"Avoiding Training in the Platform-Aware Optimization Process for Faster DNN Latency Reduction"* [80]. Este último trabajo de difusión funciona como continuación del anterior en el que, con un análisis detallado, se ahonda aún más en los costos de NetAdapt. Se proponen las modificaciones para pre-computar las interpolaciones de las LUTs, así como la incorporación de estimadores para la calidad de los candidatos para reducir el tiempo de entrenamiento y evaluación, resultados discutidos en las Secciones 5.5 y 5.6.

6.3. Trabajo futuro

En forma adicional a los avances logrados en el desarrollo del proyecto de maestría se identificaron diferentes opciones para extender el actual trabajo. En este sentido, a continuación se detallan algunas de estas ideas:

- Extender el estudio a otras plataformas de hardware. Si bien en el trabajo se evaluaron diversas plataformas de cómputo, el campo de hardware para redes neuronales es muy extenso y diverso. Por mencionar algunos casos, no fueron comprendidas en el trabajo plataformas basadas en FPGAs (por ejemplo, las ofrecidas por Xilinx [130]) o los procesadores Xeon de Intel que tienen integrado una FPGA [31].

- Evaluación de una familia de redes más variada, así como juegos de datos de mayor complejidad. En esta tesis solo se trabajó con redes convolucionales enfocadas en el procesamiento de imágenes, utilizando únicamente el dataset CIFAR10. Existen otros datasets más interesantes y de mayores dimensiones (también más costosos para computar), como por ejemplo ImageNet.
- Abordar problemas que impliquen el uso de transformers. Debido al gran impacto que están teniendo en los últimos años los transformers dentro del mundo de la inteligencia artificial (por ejemplo, ChatGPT para lenguaje, ViT [36] en visión por computadora, o AlphaFold [64] para el plegado de proteínas), evaluar y desarrollar mecanismos que permitan optimizar este tipo de rutinas (que hacen uso intensivo de grandes centros de cómputos) representa un desafío más que interesante.
- Profundizar en el estudio del uso de estimadores y lograr caracterizar sus casos de uso para diferentes familias de redes neuronales. En la Sección 5.6 se evaluaron varios estimadores para sustituir el cálculo de la precisión, donde el uso se basó en el estudio experimental de dichos estimadores. Sería interesante poder caracterizar cada estimador y sus sesgos asociados de forma tal que la aplicación de técnicas híbridas combinando estos estimadores permita compensar las debilidades de cada uno.
- Incorporar otras métricas a optimizar, no solo latencia. La incorporación de mediciones de, por ejemplo, consumo energético o el espacio en memoria que ocupan los distintos modelos como métrica a optimizar es un desafío. Es más, esta ampliación de objetivos es un primer paso en el camino de optimizar los modelos según varias métricas.

Por último, parece importante avanzar en conjugar los avances conseguidos en el trabajo de maestría, así como las distintas líneas de trabajo futuro antes planteadas, en la aplicación de técnicas de redes neuronales para la resolución de un problema en particular. Este enfoque permitirá validar las técnicas de optimización exploradas con un grado mayor de detalle al conseguido utilizando problemas genéricos (benchmarks). Además, esta línea implica trabajar en forma conjunta con investigadores de otras disciplinas, lo que permitirá ampliar los criterios y profundizar en la relevancia y efectividad del uso de las técnicas de optimización.

Bibliografía

[1] NVIDIA. *Blackwell Architecture for Generative AI*. URL: https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/ (Accedido el 05-04-2024).

[2] NVIDIA. *cuBLAS :: CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/cublas/index.html (Accedido el 17-08-2020).

[3] NVIDIA. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (Accedido el 13-03-2022).

[4] NVIDIA. *Developer Guide :: NVIDIA Deep Learning TensorRT Documentation*. URL: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#trtexec (Accedido el 15-09-2024).

[5] NVIDIA. *GeForce GTX 900 Series Graphics Cards*. URL: https://www.nvidia.com/en-us/geforce/900-series/ (Accedido el 26-04-2023).

[6] NVIDIA. *Matrix Multiplication Background User's Guide*. URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html (Accedido el 14-05-2023).

[7] NVIDIA. *NVIDIA cuDNN v9.2.1 documentation*. URL: https://docs.nvidia.com/deeplearning/cudnn/ (Accedido el 23-02-2024).

[8] NVIDIA. *NVIDIA Volta - Artificial Intelligence Architecture*. URL: https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/ (Accedido el 23-03-2022).

[9] NVIDIA. *TensorRT SDK | NVIDIA Developer*. URL: https://developer.nvidia.com/tensorrt (Accedido el 23-03-2022).

[10] NVIDIA. *Tesla P100 Data Center Accelerator*. URL: https://www.nvidia.com/en-us/data-center/tesla-p100/ (Accedido el 26-04-2023).

[11] NVIDIA. *The larger the batch size, the better when build engine?* URL: https://forums.developer.nvidia.com/t/the-larger-the-batch-size-the-better-when-build-engine/142021/2 (Accedido el 15-09-2024).

[12] *A Comparison of Neural Network Computation Architectures for Low-Power Edge Applications*. URL: https://fabianpeddinghaus.com/docs/vlsi_seminar.pdf (Accedido el 13-03-2022).

[13] Mohamed S. Abdelfattah, Abhinav Mehrotra, Lukasz Dudziak y Nicholas Donald Lane. "Zero-Cost Proxies for Lightweight NAS". En: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=0cmMMy8J5q.

[14] *AI & Robotics | Tesla*. URL: https://www.tesla.com/AI (Accedido el 10-03-2024).

[15] Ryan Olson Allison Gray Chris Gottbrath y Shashank Prasanna. *Deploying Deep Neural Networks with NVIDIA TensorRT*. URL: https://developer.nvidia.com/blog/deploying-deep-learning-nvidia-tensorrt/ (Accedido el 13-03-2022).

[16] Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, J. Santamaría, Mohammed A. Fadhel, Muthana Al-Amidie y Laith Farhan. "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions". En: *Journal of Big Data* 8.1 (mar. de 2021). ISSN: 2196-1115. DOI: 10.1186/s40537-021-00444-8. URL: http://dx.doi.org/10.1186/s40537-021-00444-8.

[17] S. Ambrogio y col. "An analog-AI chip for energy-efficient speech recognition and transcription". En: *Nature* 620.7975 (ago. de 2023), págs. 768-775. ISSN: 1476-4687. DOI: 10.1038/s41586-023-06337-5. URL: http://dx.doi.org/10.1038/s41586-023-06337-5.

[18] R.B. Ash. *Information Theory*. Dover books on advanced mathematics. Dover Publications, 1990. ISBN: 9780486665214. URL: https://books.google.com.uy/books?id=nJ3UmGvdUCoC.

[19] Ian Lankshear. *The Economics of ASICs: At What Point Does a Custom SoC Become Viable?* https://www.electronicdesign.com/embedded-revolution/economics-asics-what-point-does-custom-soc-become-viable. (Accedido el 09-09-2021).

[20] *AWS to Offer Nvidia's T4 GPUs for AI Inferencing*. URL: https://www.hpcwire.com/2019/03/19/aws-upgrades-its-gpu-backed-ai-inference-platform/ (Accedido el 23-03-2022).

[21] Gonzalo Berger, Tatiana Rischewski, Luis Chiruzzo y Aiala Rosá. "Generation of English Question Answer Exercises from Texts using Transformers based Models". En: *2022 IEEE Latin American Conference on Computational Intelligence (LA-CCI), Montevideo, Uruguay, November 23-25, 2022*. IEEE, 2022, págs. 1-5. DOI: 10.1109/LA-CCI54402.2022.9981171. URL: https://doi.org/10.1109/LA-CCI54402.2022.9981171.

[22] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry y col. "An updated set of basic linear algebra subprograms (BLAS)". En: *ACM Transactions on Mathematical Software* 28.2 (2002), págs. 135-151.

[23] *Automotive system ICs | System ICs are the key to detect sensor events, process information and drive actuator controllers in real time*. https://www.bosch-mobility.com/en/solutions/electronic-components/integrated-circuits/. (Accedido el 20-07-2023).

[24] Andrew Burnes. *GeForce RTX 3090 Ti Is Here: The Fastest GeForce GPU For The Most Demanding Creators & Gamers*. URL: https://www.nvidia.com/en-us/geforce/news/geforce-rtx-3090-ti-out-now/ (Accedido el 13-03-2022).

[25] *Caffe | Deep Learning Framework*. URL: https://caffe.berkeleyvision.org/ (Accedido el 17-06-2024).

[26] Han Cai, Ligeng Zhu y Song Han. "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware". En: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=HylVB3AqYm.

[27] *ChatGPT: Optimizing Language Models for Dialogue*. URL: https://openai.com/blog/chatgpt/ (Accedido el 13-01-2023).

[28] Yu-Hsin Chen, Joel Emer y Vivienne Sze. "Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators". En: *IEEE Micro* 37.3 (2017), págs. 12-21. DOI: 10.1109/MM.2017.54.

[29] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer y Vivienne Sze. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". En: *IEEE Journal of Solid-State Circuits* 52.1 (2017), págs. 127-138. DOI: 10.1109/JSSC.2016.2616357.

[30] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro y Evan Shelhamer. "cuDNN: Efficient Primitives for Deep Learning". En: *CoRR* abs/1410.0759 (2014). arXiv: 1410.0759. URL: http://arxiv.org/abs/1410.0759.

[31] Philip Colangelo, Enno Luebbers, Randy Huang, Martin Margala y Kevin Nealis. "Application of convolutional neural networks on Intel® Xeon® processor with integrated FPGA". En: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 2017, págs. 1-7. DOI: 10.1109/HPEC.2017.8091025.

[32] Jason Cong y Bingjun Xiao. "Minimizing Computation in Convolutional Neural Networks". En: *Artificial Neural Networks and Machine Learning – ICANN 2014*. Springer International Publishing, 2014, págs. 281-290. DOI: 10.1007/978-3-319-11179-7_36. URL: https://doi.org/10.1007/978-3-319-11179-7_36.

[33] *Continental puts its own supercomputer for vehicle AI system training, powered by NVIDIA DGX, into operation*. URL: https://www.continental.com/en/press/press-releases/continental-puts-its-own-supercomputer-into-action/ (Accedido el 10-03-2024).

[34] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li y Li Fei-Fei. "Imagenet: A large-scale hierarchical image database". En: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, págs. 248-255.

[35] Xuanyi Dong y Yi Yang. "NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search". En: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=HJxyZkBKDr.

[36] Alexey Dosovitskiy, Jost Tobias Springenberg, Maxim Tatarchenko y Thomas Brox. "Learning to Generate Chairs, Tables and Cars with Convolutional Networks". En: *IEEE Trans. Pattern Anal. Mach. Intell.* 39.4 (2017), págs. 692-705. DOI: 10.1109/TPAMI.2016.2567384. URL: https://doi.org/10.1109/TPAMI.2016.2567384.

[37] *Edge TPU*. URL: https://cloud.google.com/edge-tpu (Accedido el 23-03-2022).

[38] M.J. Ellsworth, L.A. Campbell, R.E. Simons, M.K. Iyengar, R.R. Schmidt y R.C. Chu. "The evolution of water cooling for IBM large server systems: Back to the future". En: *2008 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*. IEEE, mayo de 2008. DOI: 10.1109/itherm.2008.4544279. URL: https://doi.org/10.1109/itherm.2008.4544279.

[39] Federico Favaro, Ernesto Dufrechou, Juan P. Oliver y Pablo Ezzatti. "Optimizing the Performance of the Sparse Matrix–Vector Multiplication Kernel in FPGA Guided by the Roofline Model". En: *Micromachines* 14.11 (oct. de 2023), pág. 2030. ISSN: 2072-666X. DOI: 10.3390/mi14112030. URL: http://dx.doi.org/10.3390/mi14112030.

[40] Ehsan Adeli Fei-Fei Li y col. *Stanford CS class CS231n: Deep Learning for Computer Vision*. URL: https://cs231n.stanford.edu/ (Accedido el 05-04-2024).

[41] *FlatBuffers*. URL: https://google.github.io/flatbuffers/ (Accedido el 23-03-2022).

[42] A. Géron. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017. ISBN: 9781491962299. URL: https://books.google.com.uy/books?id=I6qkDAEACAAJ.

[43] Rafael C. Gonzalez, Richard E. Woods y Steven L. Eddins. *Digital image processing using MATLAB*. Third edition. Knoxville: Gatesmark Publishing, 2020. ISBN: 9780982085417.

[44] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[45] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville y Yoshua Bengio. "Generative Adversarial Nets". En: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. Ed. por Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence y Kilian Q. Weinberger. 2014, págs. 2672-2680. URL: https://proceedings.neurips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html.

[46] *BFloat16: The secret to high performance on Cloud TPUs*. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus. (Accedido el 20-05-2023).

[47] Geetika Gupta. *Using Tensor Cores for Mixed-Precision Scientific Computing | NVIDIA Technical Blog*. URL: https://developer.nvidia.com/blog/tensor-cores-mixed-precision-scientific-computing/ (Accedido el 13-03-2022).

[48] David Halliday y Robert Resnick. *Fisica - Volumen 2*. en. 4. Compania Editorial Continental, feb. de 1999.

[49] R. W. Hamming. "Error detecting and error correcting codes". En: *The Bell System Technical Journal* 29.2 (1950), págs. 147-160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[50] Song Han, Jeff Pool, John Tran y William J. Dally. “Learning both weights and connections for efficient neural networks”. En: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’15. Montreal, Canada: MIT Press, 2015, págs. 1135-1143.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren y Jian Sun. “Deep Residual Learning for Image Recognition”. En: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, págs. 770-778. DOI: 10.1109/CVPR.2016.90. URL: https://doi.org/10.1109/CVPR.2016.90.

[52] John L. Hennessy y David A. Patterson. “A New Golden Age for Computer Architecture”. En: *Commun. ACM* 62.2 (ene. de 2019), págs. 48-60. ISSN: 0001-0782. DOI: 10.1145/3282307. URL: https://doi.org/10.1145/3282307.

[53] Mark Horowitz. “1.1 Computing’s energy problem (and what we can do about it)”. En: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, feb. de 2014. DOI: 10.1109/isscc.2014.6757323. URL: https://doi.org/10.1109/isscc.2014.6757323.

[54] Andrew Howard y col. “Searching for MobileNetV3”. En: *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 2019, págs. 1314-1324. DOI: 10.1109/ICCV.2019.00140. URL: https://doi.org/10.1109/ICCV.2019.00140.

[55] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto y Hartwig Adam. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. En: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[56] Hantao Huang y Hao Yu. *Compact and fast machine learning accelerator for IoT devices*. en. 1.ª ed. Computer Architecture and Design Methodologies. Singapore, Singapore: Springer, dic. de 2018.

[57] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv y Yoshua Bengio. "Quantized neural networks: training neural networks with low precision weights and activations". En: *J. Mach. Learn. Res.* 18.1 (ene. de 2017), págs. 6869-6898. ISSN: 1532-4435.

[58] *HW Acceleration - AI Community*. URL: https://microsoft.github.io/ai-at-edge/docs/hw_acceleration/ (Accedido el 23-03-2022).

[59] IBM. *The hardware behind analog AI | IBM Research Blog*. 2021. URL: https://research.ibm.com/blog/the-hardware-behind-analog-ai.

[60] *Intel Distribution of OpenVINO Toolkit*. URL: https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html (Accedido el 23-03-2022).

[61] *Intel® Movidius™ Myriad™ X Vision Processing Unit*. URL: https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu/movidius-myriad-x.html (Accedido el 23-03-2022).

[62] *Intel® Movidius™ Vision Processing Units (VPUs)*. URL: https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu.html (Accedido el 23-03-2022).

[63] Norman P. Jouppi y col. "In-Datacenter Performance Analysis of a Tensor Processing Unit". En: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, jun. de 2017. DOI: 10.1145/3079856.3080246. URL: https://doi.org/10.1145/3079856.3080246.

[64] John Jumper y col. "Highly accurate protein structure prediction with AlphaFold". En: *Nature* 596.7873 (jul. de 2021), págs. 583-589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2. URL: http://dx.doi.org/10.1038/s41586-021-03819-2.

[65] Dhiraj D. Kalamkar y col. "A Study of BFLOAT16 for Deep Learning Training". En: *CoRR* abs/1905.12322 (2019). arXiv: 1905.12322. URL: http://arxiv.org/abs/1905.12322.

[66] Diederik P. Kingma y Jimmy Ba. “Adam: A Method for Stochastic Optimization”. En: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. por Yoshua Bengio y Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[67] David B. Kirk y Wen-mei W. Hwu. *Programming Massively Parallel Processors*. Elsevier, 2013. DOI: 10.1016/c2011-0-04129-7. URL: https://doi.org/10.1016/c2011-0-04129-7.

[68] Alex Krizhevsky, Ilya Sutskever y Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. En: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. Ed. por Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou y Kilian Q. Weinberger. 2012, págs. 1106-1114. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[69] Tejas D. Kulkarni, William F. Whitney, Pushmeet Kohli y Joshua B. Tenenbaum. “Deep Convolutional Inverse Graphics Network”. En: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. Ed. por Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama y Roman Garnett. 2015, págs. 2539-2547. URL: https://proceedings.neurips.cc/paper/2015/hash/ced556cd9f9c0c8315cfbe0744a3baf0-Abstract.html.

[70] Hsiang Tsung Kung y Charles E Leiserson. “Systolic arrays (for VLSI)”. En: *Sparse Matrix Proceedings 1978*. Vol. 1. Society for industrial y applied mathematics Philadelphia, PA, USA. 1979, págs. 256-282.

[71] Andrew Lavin y Scott Gray. “Fast Algorithms for Convolutional Neural Networks”. En: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, págs. 4013-4021. DOI: 10.1109/CVPR.2016.435. URL: https://doi.org/10.1109/CVPR.2016.435.

[72] Yann LeCun, John S. Denker y Sara A. Solla. “Optimal Brain Damage”. En: *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*. Ed. por David S. Touretzky. Morgan Kaufmann, 1989, págs. 598-605. URL: http://papers.nips.cc/paper/250-optimal-brain-damage.

[73] Hayeon Lee, Sewoong Lee, Song Chong y Sung Ju Hwang. “Hardware-adaptive Efficient Latency Prediction for NAS via Meta-Learning”. En: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. Ed. por Marc’Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang y Jennifer Wortman Vaughan. 2021, págs. 27016-27028. URL: https://proceedings.neurips.cc/paper/2021/hash/e3251075554389fe91d17a794861d47b-Abstract.html.

[74] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, Cong Hao y Yingyan Lin. “HW-NAS-Bench: Hardware-Aware Neural Architecture Search Benchmark”. En: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=_0kaDkv3dVf.

[75] Hanxiao Liu, Karen Simonyan y Yiming Yang. “DARTS: Differentiable Architecture Search”. En: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=S1eYHoC5FX.

[76] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu y Fuad E. Alsaadi. “A survey of deep neural network architectures and their applications”. En: *Neurocomputing* 234 (abr. de 2017), págs. 11-26. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2016.12.038. URL: http://dx.doi.org/10.1016/j.neucom.2016.12.038.

[77] Vasco Lopes, Saeid Alirezazadeh y Luís A. Alexandre. “EPE-NAS: Efficient Performance Estimation Without Training for Neural Architecture Search”. En: *Lecture Notes in Computer Science*. Springer International Publishing, 2021, págs. 552-563. DOI: 10.1007/978-3-030-86383-8_44. URL: https://doi.org/10.1007/978-3-030-86383-8_44.

[78] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue y Furu Wei. "The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits". En: *CoRR* abs/2402.17764 (2024). DOI: 10.48550/ARXIV.2402.17764. arXiv: 2402.17764. URL: https://doi.org/10.48550/arXiv.2402.17764.

[79] Raúl Marichal, Ernesto Dufrechou y Pablo Ezzatti. "Assessing the Performance of an Architecture-Aware Optimization Tool for Neural Networks". En: *International Symposium on Computer Architecture and High Performance Computing Workshops , SBAC-PADW 2023, Porto Alegre, Brazil, October 17-20, 2023*. IEEE, 2023, págs. 1-8. DOI: 10.1109/SBAC-PADW60351.2023.00011. URL: https://doi.org/10.1109/SBAC-PADW60351.2023.00011.

[80] Raúl Marichal, Ernesto Dufrechou y Pablo Ezzatti. "Avoiding Training in the Platform-Aware Optimization Process for Faster DNN Latency Reduction". En: *International Workshop on Accelerators and Hybrid Emerging Systems (AsHES), IPDPS-AsHES 2024, San Francisco, United States, May 27-31, 2024*. IEEE, 2024, págs. 1-10.

[81] Raúl Marichal, Guillermo Toyos, Ernesto Dufrechou y Pablo Ezzatti. "Evaluation of architecture-aware optimization techniques for Convolutional Neural Networks". En: *31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2023, Naples, Italy, March 1-3, 2023*. Ed. por Raffaele Montella, Javier García Blas y Daniele D'Agostino. IEEE, 2023, págs. 177-184. DOI: 10.1109/PDP59025.2023.00036. URL: https://doi.org/10.1109/PDP59025.2023.00036.

[82] Michaël Mathieu, Mikael Henaff y Yann LeCun. "Fast Training of Convolutional Networks through FFTs". En: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. por Yoshua Bengio y Yann LeCun. 2014. URL: http://arxiv.org/abs/1312.5851.

[83] John McCarthy, Marvin L. Minsky, Nathaniel Rochester y Claude E. Shannon. "A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, August 31, 1955". En: *AI Magazine* 27.4 (dic. de

2006), pág. 12. DOI: 10.1609/aimag.v27i4.1904. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/1904.

[84] Joe Mellor, Jack Turner, Amos J. Storkey y Elliot J. Crowley. "Neural Architecture Search without Training". En: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Ed. por Marina Meila y Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, págs. 7588-7598. URL: http://proceedings.mlr.press/v139/mellor21a.html.

[85] *Microsoft / AI-System*. URL: https://github.com/microsoft/AI-System/blob/main/docs/SystemforAI-4-Computer%20architecture%20for%20Matrix%20computation.pdf (Accedido el 24-04-2022).

[86] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila y Jan Kautz. "Pruning Convolutional Neural Networks for Resource Efficient Inference". En: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=SJGCiw5gl.

[87] Mythic. *Power-efficient analog compute for Edge Ai*. https://mythic.ai/.

[88] Anouar Nechi, Lukas Groth, Saleh Mulhem, Farhad Merchant, Rainer Buchty y Mladen Berekovic. "FPGA-based Deep Learning Inference Accelerators: Where Are We Standing?" En: *ACM Trans. Reconfigurable Technol. Syst.* 16.4 (2023), 60:1-60:32. DOI: 10.1145/3613963. URL: https://doi.org/10.1145/3613963.

[89] *NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications - Official PyTorch implementation*. URL: https://github.com/denru01/netadapt (Accedido el 26-04-2022).

[90] *Neuron Anatomy*. URL: https://askabiologist.asu.edu/neuron-anatomy (Accedido el 23-03-2022).

[91] NVIDIA. *Tuning CUDA Applications for NVIDIA Ampere GPU Architecture*. https://docs.nvidia.com/cuda/ampere-tuning-guide/. (Accedido el 20-05-2023).

[92] *Intel® oneAPI Deep Neural Network Library | Increase Deep Learning Framework Performance on CPUs and GPUs*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.html. (Accedido el 20-06-2023).

[93] *oneAPI: A New Era of Heterogeneous Computing*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html. (Accedido el 20-06-2023).

[94] David A. Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier y Jeff Dean. "Carbon Emissions and Large Neural Network Training". En: *CoRR* abs/2104.10350 (2021). arXiv: 2104.10350. URL: https://arxiv.org/abs/2104.10350.

[95] Felix Petersen, Hilde Kuehne, Christian Borgelt y Oliver Deussen. "Differentiable Top-k Classification Learning". En: *Proceedings of the 39th International Conference on Machine Learning*. Ed. por Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu y Sivan Sabato. Vol. 162. Proceedings of Machine Learning Research. PMLR, jul. de 2022, págs. 17656-17668. URL: https://proceedings.mlr.press/v162/petersen22a.html.

[96] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le y Jeff Dean. "Efficient Neural Architecture Search via Parameter Sharing". En: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. por Jennifer G. Dy y Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, págs. 4092-4101. URL: http://proceedings.mlr.press/v80/pham18a.html.

[97] *Post Training Quantization (PTQ)*. URL: https://pytorch.org/TensorRT/tutorials/ptq.html (Accedido el 26-04-2023).

[98] *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers (Accedido el 23-03-2022).

[99] *Quantization-aware Training (QAT)*. URL: https://pytorch.org/blog/quantization-in-practice/#quantization-aware-training-qat (Accedido el 26-04-2023).

[100] Yadira Quiñonez, M Ramirez, Carmen Lizarraga, I Tostado y Juan Bekios-Calfa. "Autonomous Robot Navigation Based on Pattern Recognition Techniques and Artificial Neural Networks". En: vol. 9108. Jun. de 2015, págs. 320-329. ISBN: 978-3-319-18832-4. DOI: 10.1007/978-3-319-18833-1_34.

[101] Esteban Real, Alok Aggarwal, Yanping Huang y Quoc V. Le. "Regularized Evolution for Image Classifier Architecture Search". En: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, págs. 4780-4789. DOI: 10.1609/AAAI.V33I01.33014780. URL: https://doi.org/10.1609/aaai.v33i01.33014780.

[102] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka I. Leon-Suematsu, Jie Tan, Quoc V. Le y Alexey Kurakin. "Large-Scale Evolution of Image Classifiers". En: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. por Doina Precup y Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, págs. 2902-2911. URL: http://proceedings.mlr.press/v70/real17a.html.

[103] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." En: *Psychological Review* 65.6 (1958), págs. 386-408. DOI: 10.1037/h0042519. URL: https://doi.org/10.1037/h0042519.

[104] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. "Learning representations by back-propagating errors". En: *Nature* 323.6088 (oct. de 1986), págs. 533-536. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038%2F323533a0.

[105] A. L. Samuel. "Some studies in machine learning using the game of checkers". En: *IBM Journal of Research and Development* 44.1.2 (2000), págs. 206-226. DOI: 10.1147/rd.441.0206.

[106] Subham Sarkar. “A survey of Apple A11 Bionic processor”. En: (mar. de 2018).

[107] R.R. Schaller. “Moore’s law: past, present and future”. En: *IEEE Spectrum* 34.6 (1997), págs. 52-59. DOI: 10.1109/6.591665.

[108] David Silver y col. “Mastering the game of Go with deep neural networks and tree search”. En: *Nature* 529.7587 (ene. de 2016), págs. 484-489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: http://dx.doi.org/10.1038/nature16961.

[109] Karen Simonyan y Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. En: *International Conference on Learning Representations*. 2015.

[110] Katie Spoon y col. “Toward Software-Equivalent Accuracy on Transformer-Based Deep Neural Networks With Analog Memory Devices”. En: *Frontiers in Computational Neuroscience* 15 (2021). ISSN: 1662-5188. DOI: 10.3389/fncom.2021.675741.

[111] William Stallings. *Computer organization and architecture : designing for performance*. Boston: Pearson-Prentice Hall, 2016. ISBN: 978-0-13-410161-3.

[112] Volker Strassen. “Gaussian elimination is not optimal”. En: *Numerische Mathematik* 13.4 (ago. de 1969), págs. 354-356. DOI: 10.1007/bf02165411. URL: https://doi.org/10.1007/bf02165411.

[113] *System Architecture | Cloud TPU | Google Cloud*. URL: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm (Accedido el 13-03-2022).

[114] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang y Joel S. Emer. *Efficient Processing of Deep Neural Networks*. Vol. 15. 2. Morgan & Claypool Publishers LLC, jun. de 2020, págs. 1-341. DOI: 10.2200/s01004ed1v01y202004cac050. URL: https://doi.org/10.2200/s01004ed1v01y202004cac050.

[115] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang y Joel S. Emer. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. En: *Proc. IEEE* 105.12 (2017), págs. 2295-2329. DOI: 10.1109/JPROC.2017.2761740. URL: https://doi.org/10.1109/JPROC.2017.2761740.

[116] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard y Quoc V. Le. "MnasNet: Platform-Aware Neural Architecture Search for Mobile". En: (2018). DOI: 10.48550/ARXIV.1807.11626. URL: https://arxiv.org/abs/1807.11626.

[117] Hidenori Tanaka, Daniel Kunin, Daniel L. K. Yamins y Surya Ganguli. "Pruning neural networks without any data by iteratively conserving synaptic flow". En: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. por Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan y Hsuan-Tien Lin. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/46a4378f835dc8040c8057beb6a2da52-Abstract.html.

[118] MLBench development team. *MLBench: Distributed Machine Learning Benchmark*. URL: https://mlbench.github.io/ (Accedido el 04-09-2024).

[119] *TensorFlow Lite*. URL: https://www.tensorflow.org/lite (Accedido el 23-03-2022).

[120] *The basic workflow to create a model for the Edge TPU*. URL: https://coral.ai/docs/edgetpu/models-intro/ (Accedido el 23-03-2022).

[121] Thomas M. Cover y Joy A Thomas. *Elements of Information Theory*. en. 2.ª ed. Nashville, TN: John Wiley & Sons, jun. de 2006.

[122] Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino y Yann LeCun. "Fast Convolutional Nets With fbfft: A GPU Performance Evaluation". En: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. por Yoshua Bengio y Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.7580.

[123] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser e Illia Polosukhin. "Attention is All You Need". En: 2017. URL: https://arxiv.org/pdf/1706.03762.pdf.

[124] Pete Warden y Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. ISBN: 9781492052043.

[125] Shmuel Winograd. *Arithmetic Complexity of Computations*. Society for Industrial y Applied Mathematics, 1980. DOI: 10.1137/1.9781611970364. eprint: https://epubs.siam.org/doi/book/10.1137/1.9781611970364. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611970364.

[126] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia y Kurt Keutzer. "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search". En: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, págs. 10734-10742. DOI: 10.1109/CVPR.2019.01099. URL: http://openaccess.thecvf.com/content%5C_CVPR%5C_2019/html/Wu%5C_FBNet%5C_Hardware-Aware%5C_Efficient%5C_ConvNet%5C_Design%5C_via%5C_Differentiable%5C_Neural%5C_Architecture%5C_Search%5C_CVPR%5C_2019%5C_paper.html.

[127] Tien-Ju Yang, Andrew G. Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze y Hartwig Adam. "NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications". En: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part X*. Ed. por Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu y Yair Weiss. Vol. 11214. Lecture Notes in Computer Science. Springer, 2018, págs. 289-304. DOI: 10.1007/978-3-030-01249-6_18. URL: https://doi.org/10.1007/978-3-030-01249-6%5C_18.

[128] Tien-Ju Yang, Yi-Lun Liao y Vivienne Sze. "NetAdaptV2: Efficient Neural Architecture Search With Fast Super-Network Training and Architecture Optimization". En: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*. Computer Vision Foundation / IEEE, 2021, págs. 2402-2411. DOI: 10.1109/CVPR46437.2021.00243. URL: https://openaccess.thecvf.com/content/CVPR2021/html/Yang%5C_NetAdaptV2%5C_Efficient%5C_Neural%5C_Architecture%5C_Search%5C_With%5C_Fast%5C_Super-Network%5C_Training%5C_and%5C_CVPR%5C_2021%5C_paper.html.

[129] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das y Scott Mahlke. “Scalpel: Customizing DNN pruning to the underlying hardware parallelism”. En: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, págs. 548-560. DOI: 10.1145/3079856.3080215.

[130] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao y Jason Cong. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks”. En: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’15. Monterey, California, USA: Association for Computing Machinery, 2015, págs. 161-170. ISBN: 9781450333153. DOI: 10.1145/2684746.2689060. URL: https://doi.org/10.1145/2684746.2689060.

[131] Barret Zoph y Quoc V. Le. “Neural Architecture Search with Reinforcement Learning”. En: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=r1Ue8Hcxg.