Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid
Curso 2022-2023

Evaluación del tiempo de inferencia y análisis del error por cuantización sobre el procesador Edge TPU

Inference time evaluation and quantization error analysis on the Edge TPU processor

Autor:
Jorge Villarrubia Elvira

Directores:
Francisco Daniel Igual Peña
Luis María Costero Valero

Convocatoria: Febrero 2023. Calificación: 10.

Resumen

El auge de las redes neuronales ha motivado que aparezcan arquitecturas de propósito específico para sus cómputos. Los cálculos tensoriales que predominan en estas redes se pueden realizar de forma eficiente mediante unidades de procesamiento tensorial (TPUs). Es habitual que las inferencias de las redes estén sujetas a estrictas restricciones de tiempo y, para reducir su latencia en entornos IoT, se utilizan TPUs para computación en el borde (*edge computing*). En este trabajo se estudia el rendimiento del procesador Edge TPU, diseñado por Google para este tipo de computación. Dicho procesador realiza las inferencias con aritmética entera de 8 *bits*, lo que produce importantes beneficios en cuanto a rendimiento y eficiencia energética. No obstante, el uso de precisión reducida requiere la *cuantización* del modelo, que introduce cierto error en la inferencia. En este trabajo también se analiza el error provocado por la cuantización para modelos entrenados mediante aprendizaje por refuerzo.

El tamaño de memoria interna del Edge TPU (8 MiB) es insuficiente para almacenar modelos que no son excesivamente grandes. Si un modelo no cabe completamente en esta memoria, una parte se almacena en el *host* y, durante la inferencia, se realizan envíos a la TPU que degradan notablemente el rendimiento. Este cuello de botella se alivia considerablemente segmentando el modelo para ejecutar los fragmentos en un *pipeline* de TPUs. Frente al uso de una sola TPU, la segmentación con hasta cuatro de ellas ha producido mejoras de rendimiento de $\times 6$ en capas neuronales de convolución y casi $\times 50$ en capas neuronales densas.

Por otra parte, se observa y justifica la influencia que tiene sobre el error por cuantización la anchura de la distribución de pesos en relación a la dispersión de sus valores. Además, para varias arquitecturas de red, se aprecian los mismos patrones de evolución del error con el avance del entrenamiento. También se observa el impacto de este error en la recompensa obtenida por el modelo cuantizado frente al modelo sin cuantizar. Finalmente, se aprecia y justifica que un escalado en profundidad de la red neuronal (añadirle más capas neuronales) aumente notablemente el error por cuantización.

Palabras clave

Arquitecturas de dominio especifico, Edge TPU, aprendizaje profundo, segmentación de modelos, cuantización, aprendizaje por refuerzo.

Abstract

The rise of neural networks has led to the emergence of specific-purpose architectures for their computations. Tensor computations that dominate these networks can be efficiently performed by tensor processing units (TPUs). It is common for network inferences to be subject to strict time constraints and TPUs are used for edge computing to reduce latency in IoT environments. In this work we study the performance of the Edge TPU processor, designed by Google specifically for edge computing. This processor performs inference with 8 bit integer arithmetic, which yields significant performance and energy efficiency benefits. However, the use of reduced precision requires model quantization, that can potentially introduce numerical errors in the inference. This work also analyses the error caused by quantization for models trained by reinforcement learning.

The internal memory size of the Edge TPU (8 MiB) is insufficient to store models that are not excessively large. If a model does not fit completely in this memory, a portion is stored on the host and, during inference, dispatches are made to the TPU that degrade performance significantly. This bottleneck is alleviated considerably by segmenting the model to run the fragments in a pipeline of TPUs. Compared to using a single TPU, segmenting with up to four of them has yielded performance improvements of 6× in convolutional neural layers and almost 50× on dense neural layers.

On the other hand, the influence that the width of the weight distribution has on the quantization error in relation to the dispersion of its values is observed and justified. Moreover, for several network architectures, the same patterns of error evolution are observed as training progresses. The impact of this error on the reward obtained by the quantized model versus the unquantized model is also observed. Finally, it is observed and justified that a deep scaling of the neural network (adding more neural layers) significantly increases the quantization error.

Keywords

Domain-specific architectures, Edge TPU, deep learning, model segmentation, quantization, reinforcement learning.

Índice de contenidos

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

El aprendizaje automático (en inglés, *machine learning*) es una rama de la inteligencia artificial que actualmente está experimentando una evolución debido, entre otros factores, al auge de las redes neuronales. Su objetivo es que un computador aprenda automáticamente a generar salidas adecuadas para cierto dominio de entradas, sin programar explícitamente la función entre ambas. Para ello, se utilizan redes de neuronas artificiales, que tratan de emular la estructura y funcionamiento del cerebro humano en un computador. Estas redes calculan una función que se ajusta progresivamente con la ejecución de un algoritmo de entrenamiento para mejorar sus salidas. Después de un periodo de *entrenamiento*, la función se supone suficientemente ajustada y la red se utiliza en procesos de *inferencia* para predecir la salida de nuevas entradas. A lo largo del trabajo, estudiaremos el funcionamiento de diferentes tipos de redes neuronales (multiperceptrón y convolución), atendiendo especialmente al proceso de inferencia.

Las redes neuronales son aproximadores universales que teóricamente pueden aprender cualquier función continua (Teorema de Aproximación Universal [1]), pero que en la práctica están limitados por recursos computacionales finitos. Su reciente apogeo se debe, entre otras cosas, a la aparición de nuevas arquitecturas y algoritmos que han mejorado mucho su rendimiento y han ampliado sus posibles aplicaciones. Veremos que los cómputos requeridos por las redes neuronales admiten un elevado grado de paralelismo, que pueden aprovechar las GPUs, pero se optimizan mucho más con procesadores específicamente diseñados para cálculos tensoriales como las TPUs (del inglés, *Tensor Processing Unit*). En este trabajo estudiaremos la arquitectura de una TPU, analizando sus principales ventajas frente a otros esquemas de procesamiento.

La creciente demanda de aplicaciones IoT[1] que utilizan redes neuronales ha otorgado un papel relevante en este campo a procesadores *ad-hoc* para sus cómputos. Debido a su especificidad, estos procesadores alcanzan muy buenos rendimientos con un consumo energético reducido, lo que resulta ideal para este tipo de aplicaciones. Además, en muchos casos se requieren latencias muy bajas para la inferencia, lo que motiva su confluencia con la computación en el borde o *edge*[2]. En este sentido, Google comercializa diversos módulos que integran el Edge TPU, un procesador específico para la inferencia con redes neuronales orientado a este tipo de computación. Se trata de un modelo de bajas prestaciones con un consumo energético muy reducido y un precio bastante asequible. En este trabajo analizaremos con detalle este procesador, realizando una evaluación paramétrica del tiempo de inferencia según diversas características de las redes utilizadas. En este estudio, detectaremos un importante cuello de botella en el tamaño de memoria interna del dispositivo, y evaluamos la segmentación del modelo entre varias TPUs como posible solución.

Uno de los principales motivos por los que el Edge TPU obtiene un buen rendimiento con un consumo energético muy reducido es que opera en aritmética de enteros y realiza las operaciones más costosas usando 8 *bits* de precisión. Para garantizar la convergencia de los algoritmos, el entrenamiento de las redes debe realizarse con valores de punto flotante más precisos (`fp32` o `fp16`); no obstante, veremos que las características de las redes neuronales permiten realizar la inferencia con operaciones enteras de menor precisión sin alterar demasiado su comportamiento.

---

[1] El internet de las cosas o IoT (del inglés, *Internet of Things*) consiste en el conjunto de dispositivos con dirección IP que se conectan entre sí mediante una red (habitualmente inalámbrica). En este campo destacan los objetos "inteligentes" que, cada vez más, utilizan redes neuronales para una determinada tarea.

[2] El *edge computing* consiste en procesar los datos cerca del lugar donde se producen en lugar de enviarlos a la nube para procesarlos allí. Esto reduce la latencia de los envíos y favorece la protección de los datos.

En este sentido, se realiza un proceso de cuantización para convertir los valores de la red entrenada con punto flotante a números enteros de 8 *bits*. Explicaremos este proceso en la memoria y analizaremos el impacto que tiene en la calidad de los modelos. Para ello, es necesario establecer una aplicación y un método concreto de aprendizaje con los que entrenar[3]. Nosotros usaremos aprendizaje por refuerzo, que es un paradigma apropiado para problemas complejos donde hacen falta redes neuronales grandes que sacan un gran provecho de las TPUs. De hecho, las TPUs son conocidas por ayudar al sistema informático AlphaGo, que consta de enormes redes entrenadas intensivamente por refuerzo, a ganar por primera vez al campeón del mundo del juego Go. Nosotros utilizaremos el algoritmo de entrenamiento PPO para aprender automáticamente a jugar en un entorno de *ping-pong* virtual. En esas condiciones, estudiaremos el error por cuantización según la cantidad de iteraciones de entrenamiento en diferentes arquitecturas de red.

1.2. Objetivos

Por una parte, este trabajo pretende extraer conclusiones generales sobre factores importantes para el rendimiento del Edge TPU. En particular, trataremos de detectar cuellos de botella y, cuando sea posible, propondremos y evaluaremos soluciones para ellos. Más allá de las conclusiones que se deriven de un análisis teórico, realizaremos diversos experimentos con modelos sintéticos para evaluar su impacto. Este análisis no estará limitado a un solo dispositivo, sino que contemplará la posibilidad de utilizar varias TPUs conjuntamente.

Por otra parte, se pretende analizar el error provocado por la cuantización previa a ejecutar en esta TPU. Concretamente, trataremos de analizar este error en el marco del aprendizaje por refuerzo, aprovechando y ampliando conclusiones de un estudio relacionado. Se escogerá una buena métrica de error y se estudiará su evolución con el avance del entrenamiento para diferentes arquitecturas de red. En este análisis consideraremos también la distribución de pesos del modelo que, según la literatura, está íntimamente relacionada con el error.

Estos dos objetivos principales dependen de otros más específicos se enumeran a continuación:

- Comprender en profundidad los fundamentos teóricos del trabajo. En particular, es especialmente relevante la teoría sobre TPUs y sobre cuantización de redes neuronales.
- Crear sistemáticamente modelos neuronales con distintas arquitecturas de red, y realizar las transformaciones necesarias para ejecutar sus inferencia en Edge TPU. Esto incluye la conversión del modelo al formato simplificado TensorFlow Lite, su cuantización y su compilación para Edge TPU.
- Evaluar diferentes tipos de redes neuronales. Concretamente, utilizaremos redes multiperceptrón y redes de convolución.
- Entrenar modelos neuronales mediante aprendizaje por refuerzo. En particular, utilizaremos el algoritmo de entrenamiento PPO, que se ejecutará eficientemente con la biblioteca RLlib aprovechando el paralelismo *multicore* y las GPUs.

1.3. Trabajo relacionado

Desde su presentación al público en el año 2016, la comunidad investigadora ha prestado mucha atención a las TPUs. La mayor parte de estudios al respecto analizan las versiones *cloud*. Los diseñadores de la primera Cloud TPU presentan detalles arquitectónicos del procesador en [2]. Por su parte, [3] y [4] realizan un estudio en profundidad de las características y diferencias de las CPUs, las GPUs y las Cloud TPUs.

[3] Para evaluar el rendimiento de un modelo no hace falta entrenarlo porque da igual lo que la función compute, pero cuando se evalúa su calidad sí es necesario.

Edge TPU es una versión de 2019 que, hasta el momento, se ha evaluado para modelos y/o tareas concretas. Por ejemplo, en [5] se utiliza en pruebas de procesamiento de imágenes, en [6] y [7] se evalúa para tareas relacionadas con la detección de objetos y en [8] se aplica a la clasificación de espectrogramas de información ganadera. El único estudio con modelos sintéticos que hemos encontrado es un artículo reciente de miembros de Google [9], que no utilizan la versión comercial (usan versiones más potentes con diferentes frecuencias, memorias o buses). Además, no se ha encontrado ningún estudio que utilice conjuntamente varias Edge TPUs.

Por otra parte, hay muchos estudios que tratan el error asociado a la cuantización de redes neuronales, pero prácticamente todos utilizan aprendizaje supervisado. Destacamos [10], que fundamenta y aplica el mismo esquema de cuantización que se usará en este trabajo (el que implementa TensorFlow Lite). En el aprendizaje supervisado, las salidas de la red son independientes entre sí, pero en el aprendizaje por refuerzo, la salida determina cuál será la siguiente entrada. Esto hace pensar que el impacto de la cuantización pueda ser diferente para el aprendizaje por refuerzo. Sin embargo, el único estudio que hemos encontrado sobre cuantización de modelos entrenados con aprendizaje por refuerzo es [11]. En el presente trabajo se resumen y amplían los resultados de ese artículo.

1.4. Metodología y plan de trabajo

Este trabajo comenzó hace aproximadamente un año, estableciendo con los directores los objetivos principales. Desde el principio, los directores han mantenido reuniones semanales con el estudiante para guiarle en el desarrollo del trabajo. Los primeros 6 meses se dedicaron a estudiar diferentes fundamentos teóricos sobre TPUs, cuantización y aprendizaje por refuerzo. En ese periodo se realizaron también las primeras pruebas de entrenamiento con la biblioteca RLlib, y las primeras inferencias en Edge TPU. Durante los siguientes 4 meses se desarrollaron los experimentos para evaluar el tiempo de inferencia en el Edge TPU, y los últimos 2 meses se dedicaron a analizar el error por cuantización. A lo largo de estos últimos 6 meses también se redactó la memoria del trabajo.

Para desarrollar los experimentos se han elaborado diversos *scripts* en Python que pueden encontrarse en el siguiente repositorio de GitHub: `https://github.com/Jorgitou98/EdgeTPU-QuantRL-Evaluation`. En el presente documento se hará referencia a algunos ficheros de este repositorio con enlaces que se pueden *clickar* para acceder a ellos. El repositorio también contiene los resultados de los experimentos y dos *notebooks* en los que se generan las gráficas de esta memoria con la biblioteca Matplotlib[4]. Prácticamente todas las demás figuras han sido elaboradas por el estudiante con la herramienta Matcha[5]. La fuente de cualquier otra figura se referencia apropiadamente al pie de la misma.

1.5. Estructura del documento

El resto del documento se estructura en cinco capítulos como se indica a continuación:

- En el **capítulo 2** se explican los principios teóricos de una TPU. Concretamente, se estudia en profundidad su arquitectura y se explican sus principales ventajas frente a utilizar una CPU o una GPU para los cómputos asociados a las redes neuronales. Finalmente, se introducen las características concretas de la versión Edge TPU.
- En el **capítulo 3** se tratan otros fundamentos importantes: la cuantización, las redes de convolución y el aprendizaje por refuerzo. En relación a este aprendizaje, se explica el algoritmo PPO y se introducen de forma práctica las librerías Gym y RLlib. Para acabar, se exponen las transformaciones necesarias para ejecutar la inferencia de un modelo en Edge TPU.

[4] `https://matplotlib.org/`

[5] `https://www.mathcha.io/editor`

- En el **capítulo 4** se presentan los resultados de diversos experimentos sobre el tiempo de inferencia en Edge TPU. Para empezar, se evalúa una sola TPU atendiendo a la cantidad de operaciones del modelo y su uso de memoria. Después, como solución a un cuello de botella detectado, se analiza la ejecución segmentada de los modelos utilizando varias TPUs. Esta parte incluye experimentos con diferentes lotes de entrada y un perfilado de la segmentación.
- En el **capítulo 5** se estudia el error por cuantización para modelos entrenados con aprendizaje por refuerzo. Primero, se presentan los resultados de [11], que analizan este error cambiando el entorno de aplicación, el algoritmo de entrenamiento y la cantidad de *bits* de precisión. Estos resultados se aprovechan después para analizar, mediante experimentos propios, cómo evoluciona el error con el avance del entrenamiento en diferentes arquitecturas del red.
- En el **capítulo 6** se exponen las conclusiones haciendo un breve repaso de los resultados obtenidos y se realizar algunas propuestas para trabajo futuro.

Capítulo 2

La unidad de procesamiento tensorial

Para desarrollar un trabajo de estas características es imprescindible una buena fundamentación teórica que ayude a plantear los experimentos e interpretar los resultados. En este sentido, es esencial entender el funcionamiento del ASIC[1] con el que vamos a trabajar, el Edge TPU. Para ello, estudiaremos la arquitectura general de una TPU, destacando las principales ventajas frente a otro tipo de procesadores y especificaremos las particularidades del modelo Edge.

Una unidad de procesamiento tensorial o TPU (del inglés, *Tensor Processing Unit*) es un circuito integrado de propósito específico desarrollado por Google para acelerar los ingentes cálculos tensoriales[2] que surgen en los procesos de inferencia sobre redes neuronales. Es un *hardware* especializado que sigue las tendencias del sector para la era *post-Moore*. Con el fin de la ley de Moore y del escalado de Dennard, surgen muchos circuitos de aplicación específica que, gracias a un diseño arquitectónico *ad-hoc*, consiguen mejorar el rendimiento y eficiencia en una tarea concreta [12].

Las TPUs se presentaron al público en la conferencia *Google I/O* del año 2016, pero han tenido ya tiempo de desempeñar un papel relevante demostrando su enorme potencial. Antes del mencionado lanzamiento, ya se utilizaron en el sistema de inteligencia artificial AlphaGo, otorgando el extra de velocidad y profundidad en el análisis de movimientos que permitió ganar al campeón del mundo de Go. A día de hoy, Google también utiliza este *hardware* para mejorar la relevancia de sus resultados de búsqueda y para extraer texto en Street View [13].

Aunque las TPUs también se utilizan para el entrenamiento de redes neuronales, su propósito principal es optimizar el proceso de inferencia; de hecho, el Edge TPU solamente puede utilizarse para este otro proceso. Las inferencias con redes neuronales se realizan de forma masiva con restricciones de tiempo real. Además, tienen unos requisitos de precisión en cómputo que permiten disminuir el tamaño de representación de los datos, lo que reduce notablemente la complejidad y el consumo energético de las arquitecturas diseñadas paea su ejecución. Esto las hace muy buenas candidatas para entornos *edge*, donde el consumo energético es fundamental y realmente surge la necesidad de utilizar modelos de predicción.

Los servicios de búsqueda, traducción o navegación de Google utilizan redes neuronales que deben responder al instante para miles de peticiones por segundo durante todo el día [13]. No es justo comparar un entrenamiento con una inferencia, sino con millones de inferencias que deben realizarse instantáneamente. Esto justifica los esfuerzos de Google por desarrollar un acelerador de inferencias como la TPU que, además, está optimizado para su integración sencilla con la biblioteca TensorFlow. Esta librería de código abierto fue desarrollada por Google para construir redes neuronales más rápidas y robustas, y actualmente es la más utilizada para desarrollar este tipo de modelos.

Google alquila sus TPUs como servicio en la nube, aunque también permite utilizarlas de forma gratuita, pero con ciertas limitaciones, en entornos como Google Colab. Por otra parte, comercializa la línea de productos Coral, orientada a soluciones de aprendizaje automático en el ámbito del *edge computing*. Esta línea ofrece múltiples dispositivos que integran el Edge TPU, un modelo de bajo precio y bajo consumo que está pensado para proyectos IoT.

---

[1] Un ASIC (del inglés, *Application Specific Integrated Circuit*) es un *chip* diseñado a medida para una tarea concreta. Una CPU no es un ASIC porque tiene un diseño lógico orientado a desempeñar tareas muy diversas, pero una TPU sí porque, como veremos, está diseñado para calcular productos escalares entre vectores.

[2] Un tensor n-dimensional es un array organizado según n índices. Un vector es el caso particular para $n = 1$ y una matriz es el caso para $n = 2$ donde los índices denotan el número de fila y columna.

2.1. Arquitectura de una TPU

Aunque existen marcadas diferencias entre los distintos modelos de TPU, todos ellos se basan en el uso de *matrices sistólicas* para optimizar los ingentes cálculos tensoriales de la inferencia con redes neuronales. A continuación, sintetizamos las ideas de [14], [2] y [15] para explicar, de forma constructiva, cómo es una matriz sistólica y por qué exhibe tan buenos resultados para estos cómputos.

2.1.1. Cadena segmentada para inferencia en un nodo

Pensemos en el nodo de una red neuronal multiperceptrón con un vector de pesos $\vec{w_{ij}} = (w_{ij1}, ..., w_{ijm})$[3], un término de sesgo (*bias*) b_{ij} y una función de activación φ_{ij}. Durante una inferencia, este nodo recibe un vector de entrada $\vec{x_k} = (x_{k1}, ..., x_{km})$[4], realiza el producto escalar entre dicho vector y los pesos, suma el sesgo al resultado y finalmente evalúa en la función de activación para obtener la salida.

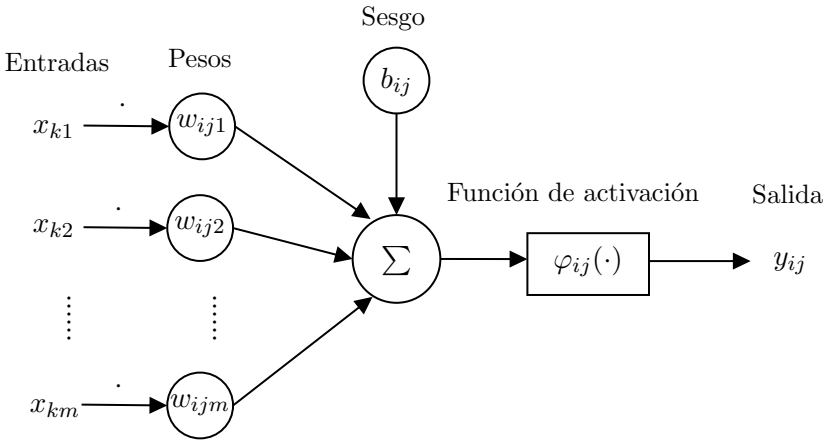


Figura 2.1: Inferencia en un nodo de una red neuronal tradicional.

Si extendemos el vector de pesos con un valor $w_{ij0} = b_{ij}$ y el vector de entrada con un valor $x_{k0} = 1$, podemos incluir la suma del sesgo en el producto escalar. De esta manera, el cómputo de un nodo puede expresarse de forma compacta como:

$$y_{ijk} = \varphi_{ij}\left(\sum_{l=0}^{m} x_{kl} \cdot w_{ijl}\right). \tag{2.1}$$

Esta expresión consta de multiplicaciones independientes entre escalares, un sumatorio para acumular todos los productos y la evaluación final en la función de activación. Para optimizarla, no hay mejor solución que desarrollar *hardware* específico para estas operaciones, haciendo especial énfasis en los productos y el sumatorio, que dependen del tamaño m de entrada (en las redes profundas este tamaño suele alcanzar varias unidades de millar).

En este sentido, no hay dificultad alguna en resolver las multiplicaciones que, al ser independientes, pueden calcularse a la vez en m circuitos distintos. El problema es acumular la suma de todos los productos, ya que un diseño de sumadores en cadena resulta muy ineficiente cuando su profundidad es grande. En un diseño tal, el último sumador de la cadena debe esperar todos los resultados intermedios antes de que su salida se estabilice, lo cual puede resultar dramático para el tiempo de propagación.

[3] Los subíndices i y j indican que los pesos corresponden a la j-ésima neurona de la i-ésima capa.

[4] El subíndice k indica que la entrada se corresponde con la k-ésima inferencia.

Por ejemplo, asumiendo que el tiempo de propagación de un sumador fuese de apenas 2 ns, con una cadena de 250 sumadores el tiempo de propagación ascendería a 500 ns y la frecuencia máxima de reloj sería de apenas 2 MHz.

La solución a este problema es una estructura de *pipeline* que permita segmentar la cadena en varias etapas. De esta forma, se pueden computar simultáneamente varios sumatorios en etapas distintas con una frecuencia de reloj más alta. Para ello, basta colocar un elemento de memoria entre cada par de etapas que almacene el resultado intermedio durante un ciclo de reloj. Así, la latencia para pasar por toda la cadena no sería el precio a pagar por un solo sumatorio sino por un conjunto de ellos (con el *pipeline* lleno, tantos como etapas). Además, la frecuencia máxima de reloj dependería exclusivamente del tiempo de propagación de una etapa, que es bastante menor que el de toda la cadena.

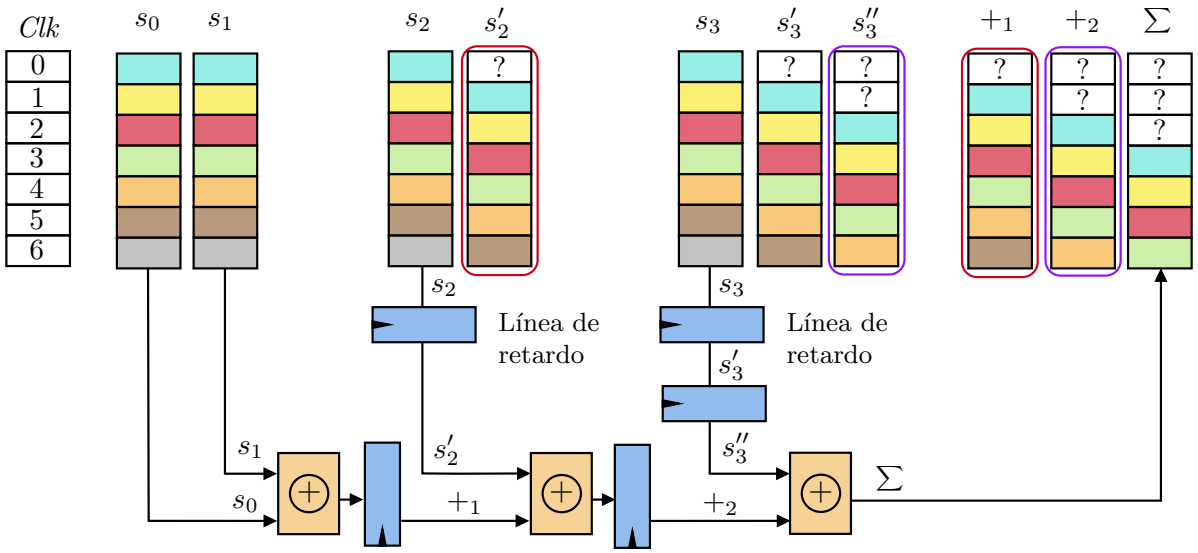


Figura 2.2: Estructura y funcionamiento de una cadena de 3 sumadores para acumular la suma de 4 operandos. Las cajas naranjas representan sumadores y las cajas azules registros *hardware*. Mediante *buffers* con colores se representa el ciclo en que cada sumatorio pasa por cada etapa del *pipeline*. Cada color representa uno de los sumatorios propagados.

Además de los registros colocados entre cada par de sumadores, en la figura 2.2 podemos observar la utilización de otros registros para formar pequeñas líneas de retardo que retrasan la entrada de los valores s_2 y s_3. Sin dejar de incorporar nuevos datos en cada ciclo de reloj, estas líneas permiten sincronizar adecuadamente la cadena para no mezclar valores de sumatorios distintos.

En la figura 2.2 podemos observar que los *buffers* de $+_1$ y s'_2 (correspondientes a la entrada del segundo sumador y destacados en rojo) y los *buffers* de $+_2$ y s''_3 (correspondientes a la entrada del tercer sumador y destacados en morado) son iguales gracias al retardo de estas líneas. Por su parte, la figura 2.3 muestra una representación más detallada de los 4 primeros ciclos de reloj donde se observa la sincronización de datos de los sumatorios a la entrada de los sumadores.

Con este diseño segmentado, el tiempo de propagación que define la frecuencia de reloj es el de cada etapa del *pipeline* y resulta independiente de la longitud de la cadena. Podrá utilizarse la misma frecuencia de reloj en el ejemplo con 250 sumadores (que sin segmentación era un problema) y en la pequeña cadena de nuestra figura.

Asumiendo que la entrada del registro debe mantenerse estable al menos 0.1 ns antes del flanco de reloj y que la propagación en el registro es de a lo sumo 0.4 ns, se garantiza una propagación estable de la señal en 2.5 ns: 2 ns de la etapa de suma, 0.1 ns para estabilizar el valor y 0.4 ns

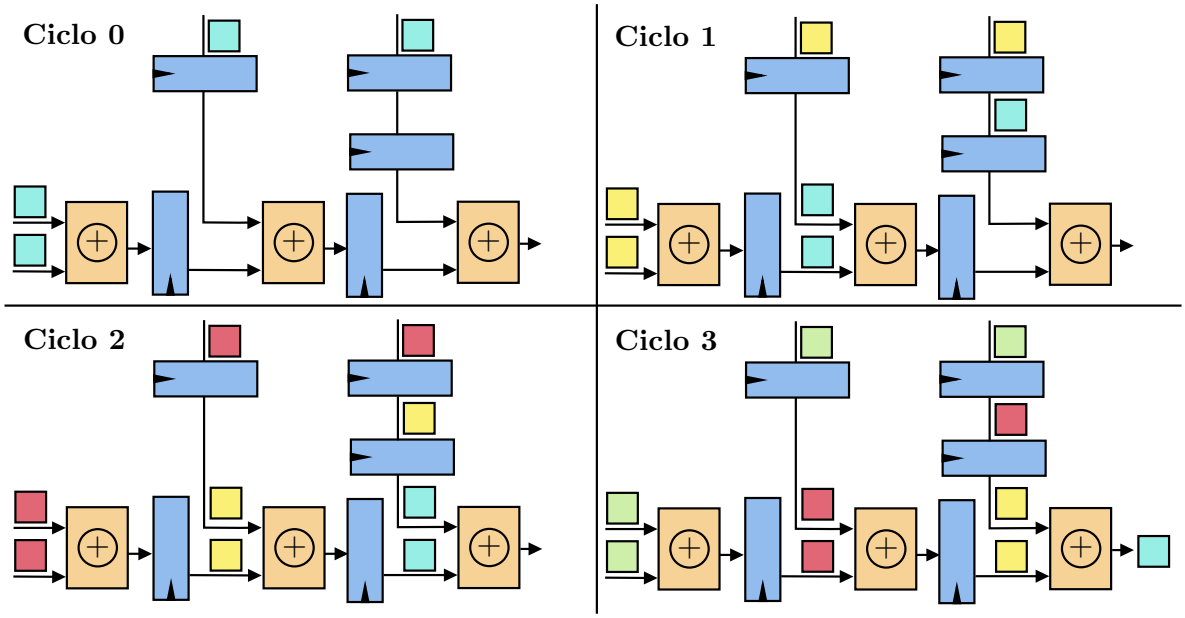


Figura 2.3: Propagación por el *pipeline* de los datos correspondientes a los distintos flujos (cada uno representado con un color) durante los primeros 4 ciclos de reloj.

para propagarlo en el registro. Con ello, la frecuencia máxima de reloj alcanzaría los 400 MHz y dejaría de ser un cuello de botella tan evidente.

Ahora, debemos tener en cuenta que, según la ecuación 2.1, cada uno de los sumandos es el producto entre una componente x_{kl} del vector de entradas y la correspondiente componente w_{ijl} del vector de pesos. Como ya hemos comentado, todos estos productos pueden realizarse en paralelo y basta añadir al diseño de la figura 2.2 un circuito multiplicador delante de cada sumador.

No obstante, debemos notar que los distintos flujos sumatorios ahora serán los diferentes vectores de entrada $\vec{x_k}$, y que el vector de pesos $\vec{w_{ij}}$ es el mismo para todos ellos: en la inferencia k-ésima se computa $\vec{x_k} \cdot \vec{w_{ij}}$, en la inferencia siguiente $\vec{x_{k+1}} \cdot \vec{w_{ij}}$, etc. Esto supone que el *pipeline* se llena con datos correspondientes a inferencias distintas pero los pesos se pueden reutilizar entre ellas.

De manera genérica, la componente l-ésima de las distintas inferencias $(x_{kl}, x_{(k+1)l}, ...)$ se multiplica por la componente l-ésima del vector de pesos w_{ijl}. Las componentes l-ésimas estarán en mismo *buffer* e irán a parar a la misma celda de multiplicación y suma. De esta forma, el peso w_{ijl} resultará un operando de entrada fijo para dicha celda[5] y resultará más apropiada la representación derecha de la figura 2.4 para ilustrar los multiplicadores-sumadores.

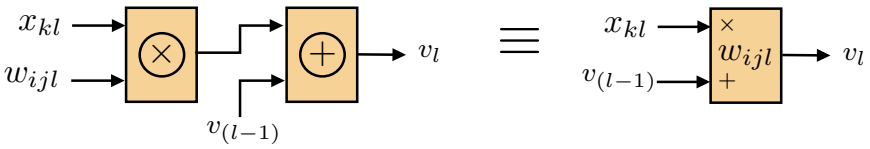
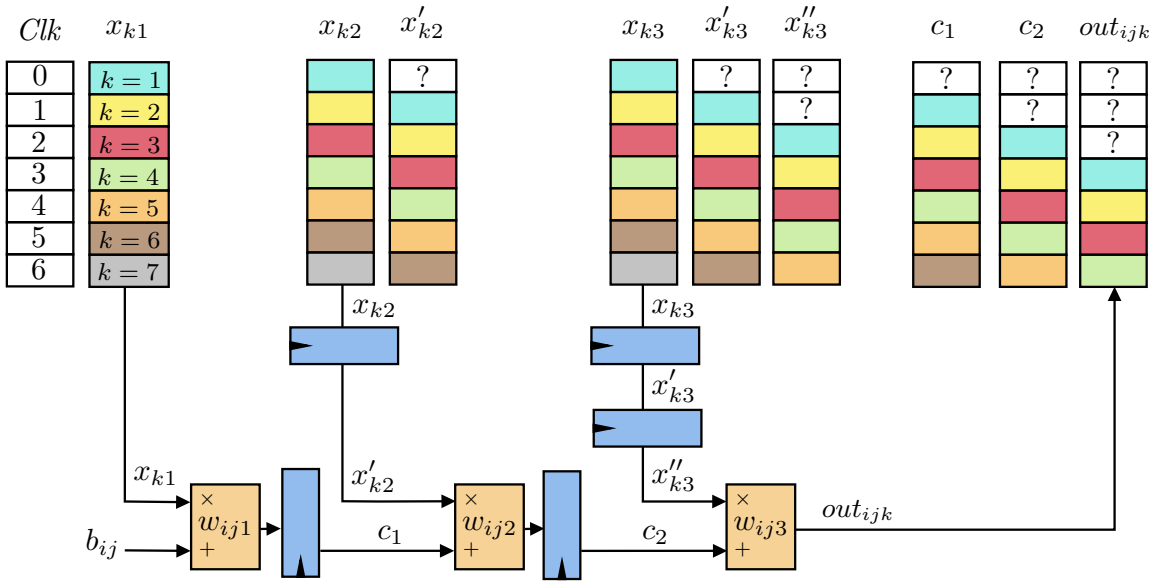


Figura 2.4: A la izquierda una representación de multiplicador-sumador con diferentes cajas para cada operación y el peso w_{ijl} como elemento de entrada. A la derecha una representación más compacta con una sola caja donde w_{ijl} se denota como parte suya.

[5] Más adelante veremos que el valor del peso en un multiplicador puede cambiar cuando el tamaño de las entradas exceda al ancho de la cadena. En tal caso, los cómputos se dividen en varias fases y los pesos se modifican de una a otra.

Para expresar la ecuación 2.1 de manera más ilustrativa y compacta, incluimos la suma del sesgo $b_{ij} = w_{ij0}$ en el producto escalar añadiendo la componente de entrada $x_{k0} = 1$. A nivel *hardware* esto no es necesario ya que el primer multiplicador-sumador no recibe valor de la celda anterior y puede computar directamente $b_{ij} + x_{k1} \cdot w_{ij1}$. A falta de evaluar en la función de activación φ_{ij}, el diseño de la figura 2.5 realiza el cómputo de inferencia para un nodo.



$$out_{ijk} = x_{k1} \cdot w_{ij1} + x_{k2} \cdot w_{ij2} + x_{k3} \cdot w_{ij3} + b_{ij}$$

Figura 2.5: Diseño *hardware* de una cadena con 3 multiplicadores-sumadores para el producto escalar de la inferencia en un nodo. Los colores se corresponden con entradas de distintas inferencias asociadas al subíndice k.

2.1.2. Matriz sistólica para inferencia en múltiples nodos

Las redes neuronales habitualmente constan de millones de nodos distribuidos en diferentes capas con interconexiones entre capas consecutivas. Si el nodo n_{ij} está conectado con el nodo $n_{(i+1)s}$, entonces y_{ij} es una de las entradas para calcular $y_{(i+1)s}$. Esto significa que el cómputo de sus salidas no puede realizarse con facilidad en paralelo. Sin embargo, el cálculo de la salida para nodos que se encuentren en la misma capa es totalmente independiente y, por tanto, paralelizable. En este sentido, para poder computar en varios nodos de la misma capa a la vez, basta con replicar el diseño que tenemos para uno de ellos, como se muestra en la figura 2.6.

La estructura resultante es un conjunto de celdas de multiplicación y suma con apariencia de matriz, donde cada fila es la cadena correspondiente a un nodo y cada columna es una etapa del *pipeline*. Esta matriz se conoce como *sistólica* por la semejanza entre el empuje de sangre en un latido del corazón y el avance de los datos por el *pipeline* cuando llega el flanco de reloj. Se trata de una matriz implementada en *hardware* cuyas dimensiones son fijas. Sin embargo, el número de nodos de cada capa y el tamaño de entrada de cada nodo son parámetros dependientes del modelo que pueden no coincidir con estas dimensiones (la matriz de la figura 2.6 tiene dimensiones 3×3, pero los tamaños reales son 64×64, 128×128 y 256×256).

En caso de que el número de nodos de alguna capa sea inferior al número de filas de la matriz, o en caso de que el tamaño de entrada de un nodo sea inferior al número de columnas, simplemente no se aprovechan todas las celdas rellenando algunos tensores con ceros para que sus cálculos no influyan en el resultado. Tal y como apuntan desde Google, esta situación debe tratar de evitarse ya que causa una importante pérdida de rendimiento y eficiencia [16].

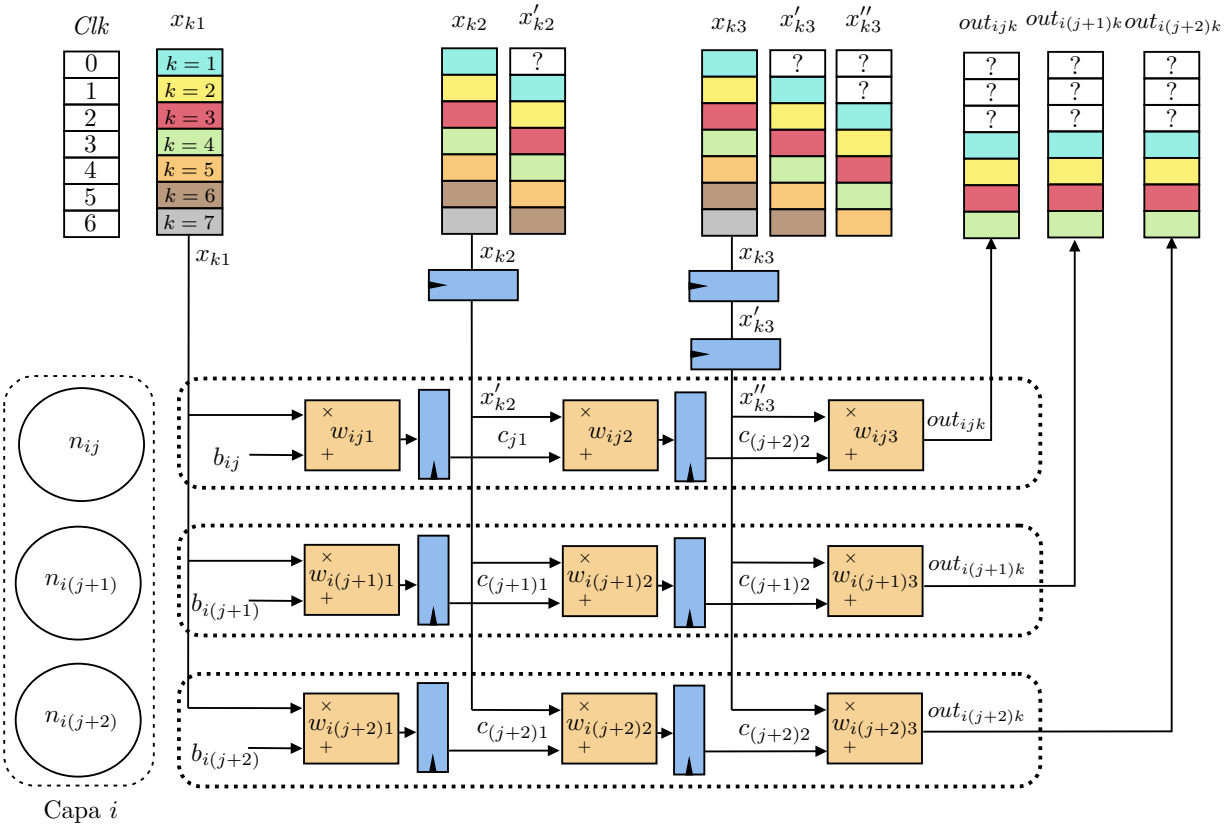


Figura 2.6: Matriz sistólica compuesta por 3 cadenas segmentadas en 3 etapas que permiten realizar en paralelo los cómputos correspondientes a 3 nodos de la misma capa.

Si el número de nodos de una capa es superior al número f de filas, solo se podrá paralelizar el cómputo de f nodos a la vez y serán necesarias varias fases para completar dicha capa. Si el tamaño m de entrada es superior al número c de columnas, cada vector de entrada $\vec{x_k} = (x_{k1}, ..., x_{km})$ se divide en segmentos $(x_{kl}, ..., x_{k(l+c-1)})$ de tamaño c[6], que se ejecutan secuencialmente en la matriz y producen resultados parciales que finalmente habrá que reducir.

Por ejemplo, supongamos que los nodos de la capa i de la figura 2.6 recibieran entradas de tamaño 8 para la matriz sistólica 3×3 de esa misma figura. El cómputo para cada una de las entradas $\vec{x_k} = (x_{k1}, .., x_{k8})$ tendría que dividirse en $\lceil \frac{8}{3} \rceil = 3$ fases distintas. Como la división no es exacta, las dos primeras fases aprovecharían todo el ancho de la matriz, pero la tercera fase solo computaría de forma útil en 2 celdas (rellenando con $x_{k9} = w_{ij9} = 0$ los valores de la tercera). Por otra parte, la suma del sesgo se realizaría exclusivamente en una de las fases, rellenando con ceros el valor del sesgo para las demás. Las siguientes expresiones muestran los valores intermedios calculados en las tres fases:

$$
\begin{aligned}
out_{ijk}^{(1)} &= x_{k1} \cdot w_{ij1} + x_{k2} \cdot w_{ij2} + x_{k3} \cdot w_{ij3} + b_{ij}, \\
out_{ijk}^{(2)} &= x_{k4} \cdot w_{ij4} + x_{k5} \cdot w_{ij5} + x_{k6} \cdot w_{ij6}, \\
out_{ijk}^{(3)} &= x_{k7} \cdot w_{ij7} + x_{k8} \cdot w_{ij8}.
\end{aligned}
$$

Para seguir aprovechando la segmentación de las cadenas, se colocan consecutivamente en los *buffers* de lectura segmentos correspondientes a inferencias distintas. En nuestro ejemplo, el *buffer* conectado con la primera columna guardará consecutivamente las primeras componentes de entrada de las distintas inferencias $(x_{k1}, x_{(k+1)1}, ...)$. Dichas componentes entran ciclo a ciclo

[6] Si la división no es exacta el último de estos vectores tendrá menos de c coordenadas. Para ajustar su tamaño a c sin afectar al resultado, se rellena el vector con ceros.

al *pipeline* para aprovechar su profundidad durante la primera fase. Con el mismo propósito, este *buffer* almacena consecutivamente las cuartas componentes $(x_{k4}, x_{(k+1)4}, , ...)$ para la segunda fase y finalmente las séptimas $(x_{k7}, x_{(k+1)7}, ...)$ para la tercera.

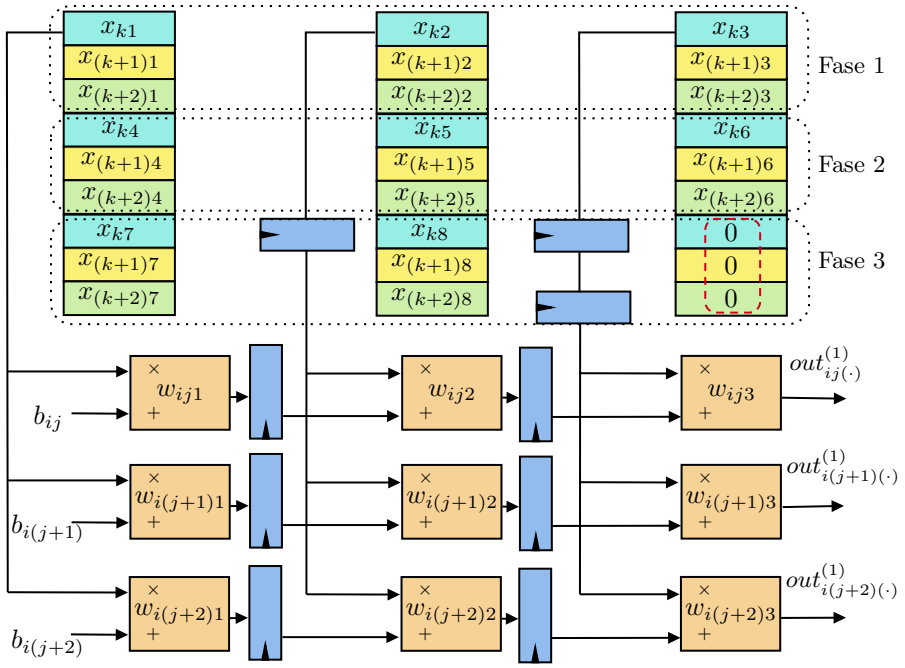


Figura 2.7: Reparto de la entrada en los diferentes *buffers* para nuestro caso de ejemplo. Se redondea en color rojo el relleno con ceros del último *buffer* debido a que el tamaño de entrada no es múltiplo del ancho de la matriz.

Cuando los primeros valores de una fase ingresan en la matriz, hay que cambiar los pesos correspondientes a algunas celdas. Por ejemplo, cuando ingrese el valor x_{k4}, los pesos de la primera columna de la matriz no deberían ser $(w_{ij1}, w_{i(j+1)1}, w_{i(j+2)1})$ sino $(w_{ij4}, w_{i(j+1)4}, w_{i(j+2)4})$. Los pesos de una nueva fase no se cargan de golpe, sino etapa a etapa, ya que cuando una etapa cambia de fase las etapas posteriores a ella siguen en la fase anterior con los mismos pesos.

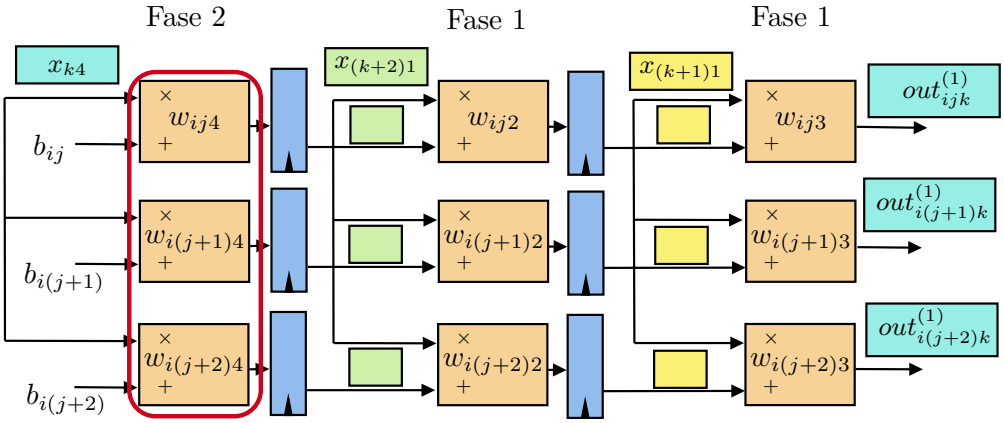


Figura 2.8: La primera etapa recibe el primer dato de la fase 2 y modifica sus pesos (señalado en rojo). Sin embargo, las siguientes etapas siguen en la fase 1 con los pesos que ya tenían.

La TPU cuenta con una memoria RAM que debería resultar suficiente para almacenar todos los pesos de la red. No obstante, suele acompañarla de un *buffer* de menor latencia donde cargar bloques de pesos que van a utilizarse próximamente.

También debemos notar que, para cada vector de entrada $\vec{x_k}$, la salida out_{ijk} es la suma de los valores intermedios calculados en las distintas fases:

$$out_{ijk} = out^{(1)}_{ijk} + out^{(2)}_{ijk} + out^{(3)}_{ijk}.$$

Para reducir los valores intermedios, en la salida de cada cadena se utiliza una estructura acumuladora compuesta por un sumador y una cola. El sumador recibe como operandos la salida de la cadena y el primer valor de la cola y almacena su resultado de nuevo en la cola. De esta forma, a la suma acumulada de las fases anteriores se le añade la salida de la última fase. El comportamiento FIFO de la cola (*First In First Out*, es decir, el primero en entrar es el primero en salir) es fundamental para no mezclar resultados intermedios de inferencias distintas.

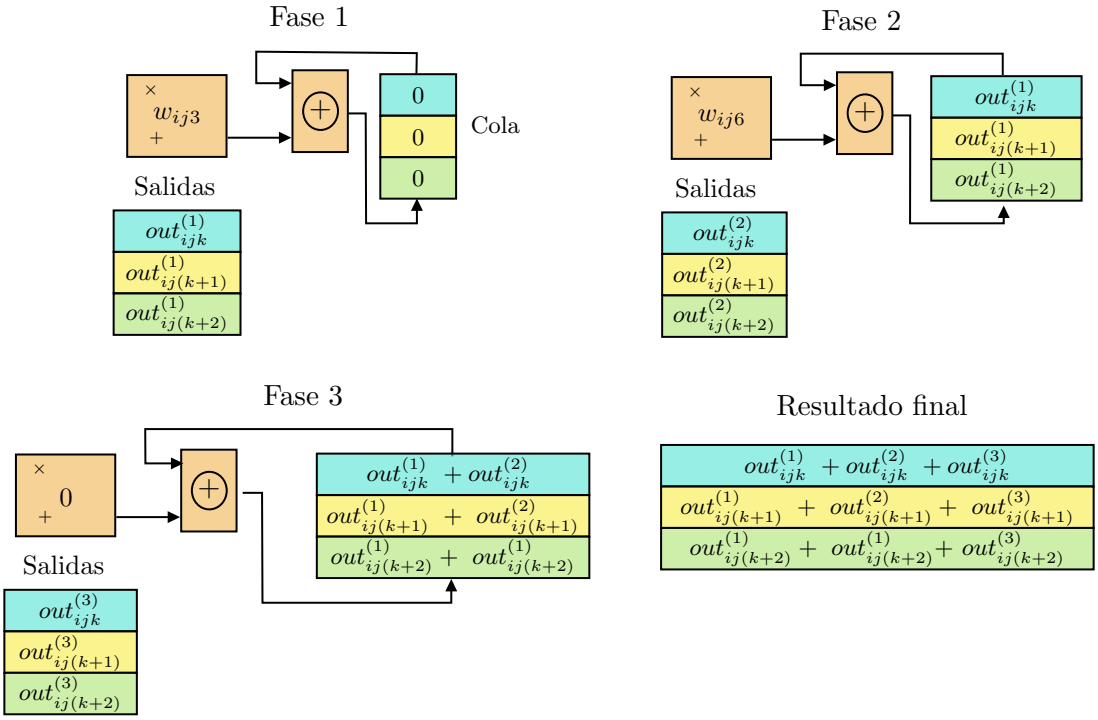


Figura 2.9: Estructura acumuladora a la salida de la primera cadena durante las 3 fases correspondientes al j-ésimo nodo de la i-ésima capa.

2.1.3. Función de activación, otras componentes y conexión con la CPU

Para completar el cómputo de un nodo correspondiente a la ecuación 2.1, falta por evaluar en la función de activación φ_{ij}. Como ya comentamos, la optimización de este cálculo no es tan relevante porque que se computa una sola vez por cada nodo e inferencia, independientemente del tamaño m de las entradas. No obstante, como el conjunto de funciones de activación que suele utilizarse en las redes neuronales es bastante limitado (fundamentalmente *ReLu*, *SoftMax*, sigmoides o tangentes hiperbólicas), las TPUs incorporan circuitos específicos para computarlas.

El resultado de estas evaluaciones es la entrada para los nodos de la siguiente capa, que se almacena en un *buffer* unificado según se completan los cómputos. Cuando termina la inferencia de la capa, se produce una pequeña sincronización para leer de forma segura los resultados y distribuirlos adecuadamente entre los *buffers* de la matriz sistólica. Una vez hecho esto, comienza la inferencia de la siguiente capa con la correspondiente carga y modificación de pesos.

Por otra parte, las TPUs incluyen *hardware* específico para ciertas operaciones propias de una red neuronal que no se ajustan bien al cálculo de productos escalares de la matriz sistólica. Este es el caso de la normalización de salidas de una capa o de las capas de agrupación (*pooling*) utilizadas en las redes de convolución que veremos en la sección 3.2.

Hay que hacer hincapié en que estas operaciones son particulares de cada modelo y, aunque valga la pena acelerarlas, no es el objetivo primordial de la TPU. Dependiendo del modelo, estas operaciones pueden utilizarse o no y habitualmente no resultan los cómputos más costosos. El cálculo de productos escalares es común a todos los modelos de red neuronal y habitualmente es el más demandante durante la inferencia. Por ello, la clave arquitectónica de la TPU no es otra que la matriz sistólica.

Todo este *hardware* se gestiona a través de una lógica de control que, como comentaremos en la próxima sección, es mínima en comparación con otro tipo de procesadores. La TPU realmente es un procesador complementario a una CPU que, junto a los datos del modelo y las entradas, le proporciona instrucciones máquina para alimentar esta lógica.

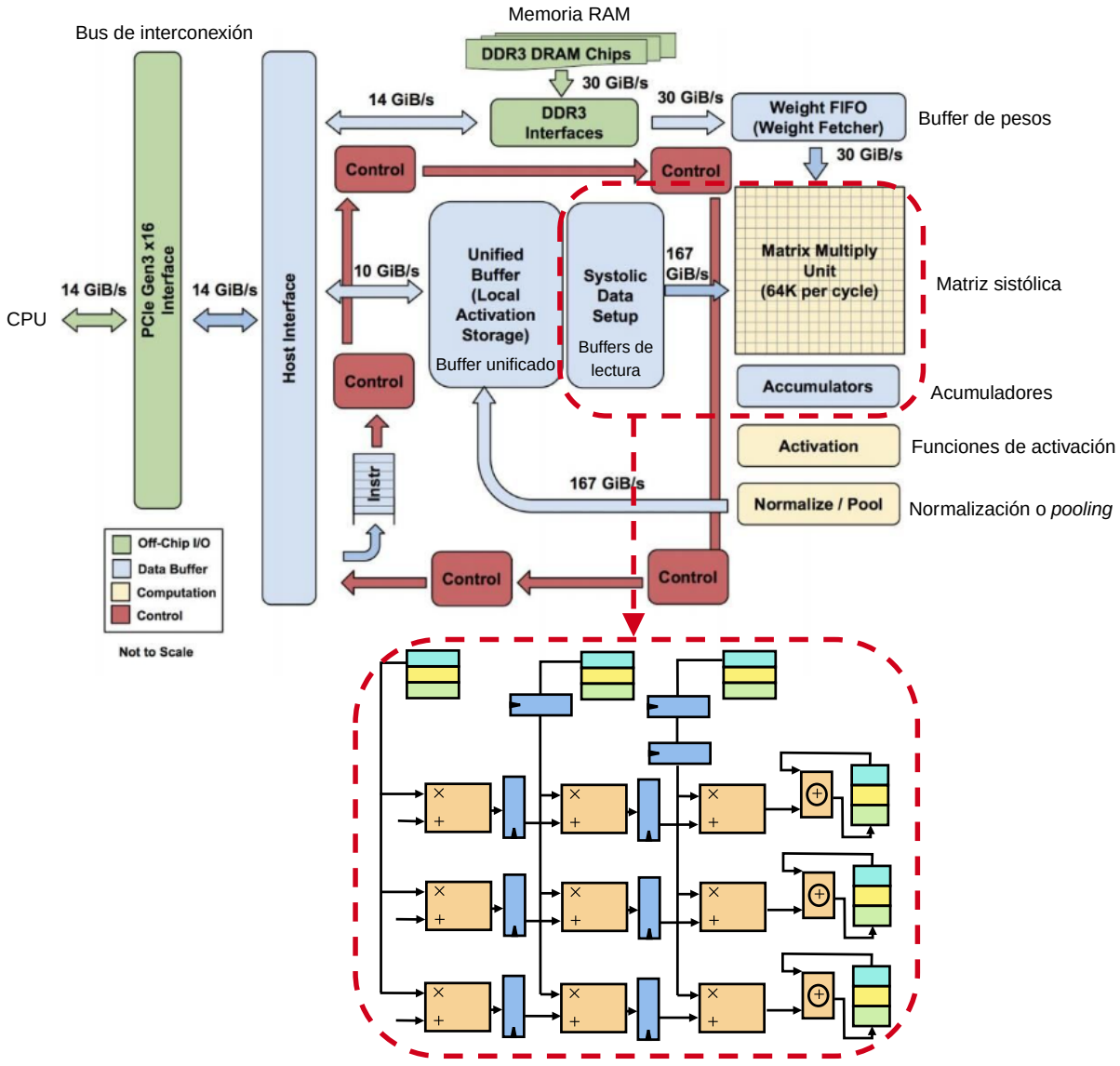


Figura 2.10: Arquitectura general de un modelo de TPU. Se indica el nombre con el que nos hemos referido a cada componente, ampliando las que hemos desarrollado con detalle en la sección (matriz sistólica con sus *buffers* y acumuladores). (Figura modificada extraída de [2])

La conexión entre CPU y TPU se realiza habitualmente con un bus *PCI Express* (bus especializado en conectar solo dos dispositivos) sin un ancho de banda demasiado grande. El trasiego de información por el bus debería ser moderado ya que apenas harían falta dos transferencias: la carga inicial de entradas, instrucciones y modelo, y la devolución de resultados a CPU.

Para esos intercambios de información no hace falta mucha velocidad, pero hay casos en los que se realizan más comunicaciones. En este sentido, hay que evitar cuellos de botella relacionados con cualquier otro trasvase de información. Por ejemplo, si algún cómputo no puede expresarse con instrucciones de TPU, la CPU se encargará de él y será necesario un intercambio de información continuado. Se produce una situación similar si el modelo no puede almacenarse íntegramente en la memoria de la TPU, en cuyo caso parte de los pesos se mantendrán en CPU y será necesario utilizar constantemente el bus para ir cargándolos.

2.1.4. Conclusiones arquitectónicas

Tras estudiar en profundidad la arquitectura de una TPU, podemos extraer algunas conclusiones generales sobre el rendimiento de este procesador ante ciertas características de las entradas y modelos neuronales utilizados:

- **La TPU está diseñada para resolver múltiples inferencias en una misma ejecución a partir de un lote de entradas distintas.**

  Los cómputos de diferentes entradas pueden realizarse a la vez en distintas etapas del *pipeline* sin necesidad de cambiar los pesos de las celdas. Si disponemos de muchas entradas diferentes, la mayor parte de los cómputos se realizarán con el *pipeline* lleno produciendo en cada ciclo tantos resultados como filas tenga la matriz sistólica. El tiempo de inferencia de ejecutar conjuntamente un lote debería ser mucho menor que el de una sola inferencia multiplicado por el tamaño del lote.

- **El rendimiento de la TPU debería mejorar al aumentar el tamaño de entrada de los nodos de un modelo. En este sentido, la TPU sería adecuada para redes con alto grado de interconexión y muchos nodos en cada capa.**

  Cuando el tamaño de entrada de un nodo excede el número de columnas de la matriz, las entradas se dividen en fragmentos de ese tamaño. Mayor tamaño de entrada significa mayor cantidad de fragmentos y más ciclos con el *pipeline* lleno a máximo rendimiento antes de la siguiente latencia por llenado. En este caso, sí hay que modificar los pesos de las celdas, pero los cambios son de baja latencia y tratan de minimizarse. Lo ideal es que el tamaño de entrada sea múltiplo de número de columnas para no rellenar con ceros los tensores del último fragmento y que todos los cómputos sean útiles.

  En una red multiperceptrón, el número de entradas de un nodo es la cantidad de neuronas de la capa anterior que se conectan a él. Por tanto, que haya muchos nodos en cada capa y muchas interconexiones entre ellas implica que las entradas sean grandes. Lo ideal es que el número de nodos de una capa sea múltiplo del número de filas para que durante la ejecución no se desaprovechen cadenas con cómputos de relleno.

- **El rendimiento de una TPU con una sola matriz sistólica no debería mejorar al aumentar el número de capas de un modelo.**

  Como las salidas de una capa son las entradas de la siguiente, la matriz sistólica computa las capas secuencialmente. Teóricamente, es posible comenzar los cómputos de una capa sin haber terminado los de la anterior (basta tener alguna salida calculada), pero esto no se ajusta bien a la arquitectura de la matriz, cuyas cadenas trabajan de forma independiente sin intercambiarse datos. No obstante, con varias matrices sistólicas se podría, por ejemplo, formar un pequeño *pipeline* de matrices que se encarguen de los cálculos de capas consecutivas.

En el capítulo 4 estudiaremos el tiempo de inferencia de una TPU orientada a *edge computing* según aspectos que no se desprenden directamente del diseño arquitectónico. Así, complementaremos las conclusiones generales que acabamos de presentar con particularidades de esa TPU.

2.2. Ventajas de una TPU frente a una CPU y una GPU

Con especial atención a la inferencia con redes neuronales, en esta sección explicamos las principales ventajas de una TPU frente a un procesador de propósito general como la CPU y frente a un procesador ampliamente utilizado para aprendizaje automático como la GPU.

2.2.1. Mayor rendimiento al calcular productos escalares

Como explicamos en la sección anterior, la inferencia con redes neuronales se basa en enormes productos escalares, muchos de los cuales pueden ejecutarse en paralelo. Vimos que la TPU ha sido específicamente diseñada para estas operaciones, pero vamos a tratar de entender los aspectos cruciales en los que obtiene ventaja.

Una CPU es un procesador mucho más versátil, enfocado a cálculos más generales. En este sentido, la CPU se orienta al producto de dos números en lugar de la multiplicación de dos vectores o matrices. No obstante, puede aprovechar una pequeña parte del paralelismo inherente a las operaciones vectoriales mediante las instrucciones SIMD (*Single Instruction Multiple Data*). Estas instrucciones permiten operar con varios datos a la vez aprovechando mejor la capacidad total de las ALUs.

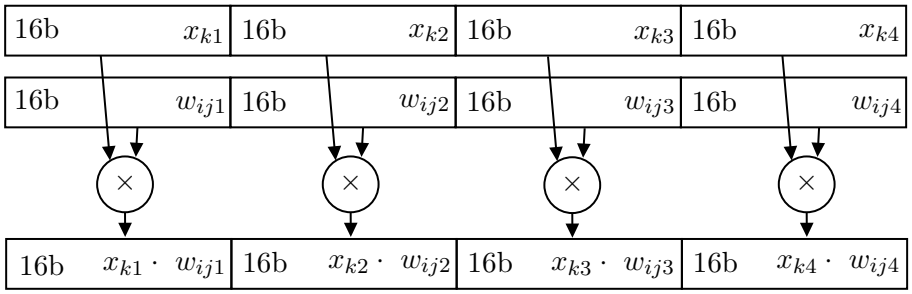


Figura 2.11: Instrucción SIMD de multiplicación para 64 *bits* de ancho que realiza en paralelo 4 productos entre entradas y pesos de una inferencia suponiendo componentes de 16 *bits*.

Este paralelismo a nivel de datos puede anidarse con una distribución *multicore*, de forma que se compute simultáneamente en varios núcleos. Incluso, puede usarse *multithreading* a nivel de núcleo para cambiar el hilo de ejecución, ocultando así las latencias por acceso a memoria de los demás. Las CPUs permiten incluso *simultaneous multithreading* para aprovechar mejor los recursos ejecutando a la vez instrucciones de diferentes hilos en una misma etapa del *pipeline* (instrucciones de diferentes hilos podrían utilizar las ALUs en el mismo ciclo).

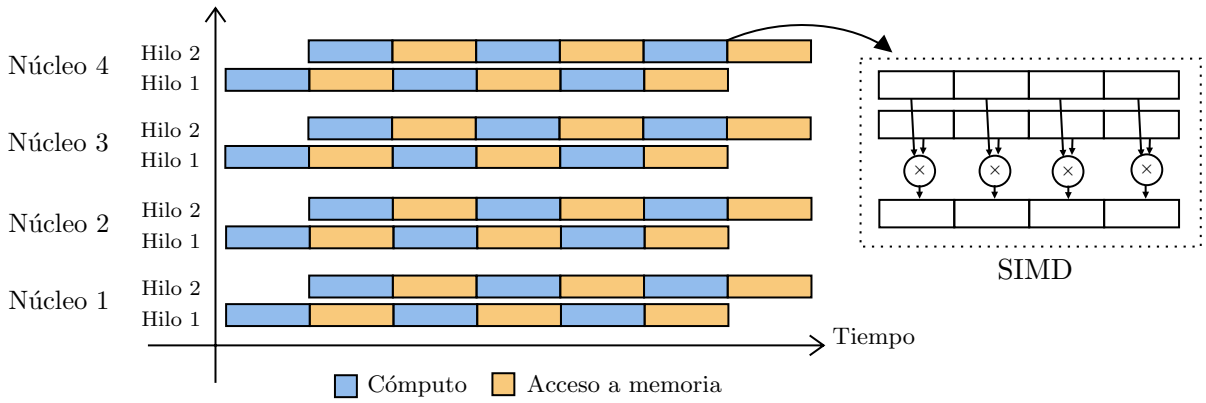


Figura 2.12: Representación temporal de la ejecución de 8 hilos repartidos por pares en 4 núcleos. Con *multithreading* en cada uno, el hilo con cómputo (color azul) podría ejecutarse mientras el otro hilo accede a memoria (color naranja). Cada cómputo se realizaría con el paradigma SIMD.

En cualquier caso, estos métodos son insuficientes para explotar el paralelismo que subyace en una enorme cantidad de grandes productos escalares independientes. El número de núcleos de una CPU de propósito general no suele ser muy grande (habitualmente 8 o 16) debido a que las aplicaciones no suelen presentar tanto paralelismo como para aprovecharlos. Estamos ante un procesador de propósito general, donde demasiados núcleos resultarían casi siempre un derroche absurdo de coste y energía. Por otra parte, la CPU no integra grandes optimizaciones *hardware* para acumular las multiplicaciones de un producto escalar, como sí tiene la TPU con las cadenas de sumadores. Las optimizaciones en este sentido son mediante algoritmos *software* que no resultan tan provechosos como la solución *hardware* de la TPU.

Esa capacidad paralela que se echa en falta en una CPU, es uno de los principales baluartes de las GPUs. Aunque estos dispositivos nacieron para procesamiento gráfico, han demostrado ser muy útiles en aplicaciones donde se pueda extraer un alto grado de paralelismo a nivel de datos.

Basándonos en la abstracción arquitectural ofrecida por CUDA, una GPU está formada por miles de ALUs, conocidas como *cores*, que se separan en diferentes bloques denominados *stream multiprocessors* (SMs). Cada SM recibe varios conjuntos de hilos llamados *warps*, con los que realiza *multithreading* para ocultar latencias de acceso a memoria. En este sentido, hay un paralelismo a tres niveles: en diferentes SMs se ejecutan a la vez *warps* distintos, en cada SM se ejecutan simultáneamente las instrucciones de los hilos de un *warp* y en cada SM se cambia de *warp* para ocultar los accesos a memoria de otros.

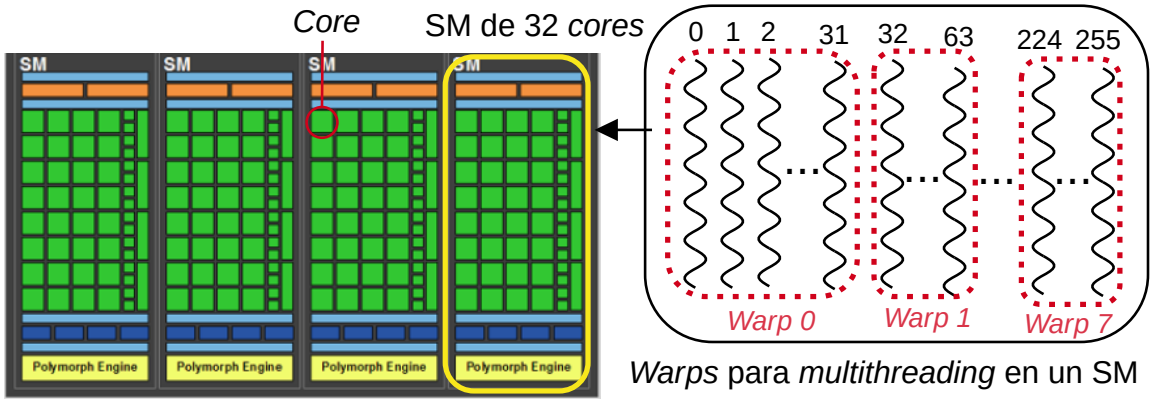


Figura 2.13: Esquema de un grupo de SMs con 32 *cores* cada uno (una GPU suele tener varios de estos grupos). Cada SM aloja 8 *warps* de 32 hilos cada uno (uno para cada *core* del SM). El planificador del SM conmuta de *warp* para ocultar los accesos memoria (*multithreading*).

Para generar suficientes hilos independientes que consigan aprovechar esta estructura es necesaria una enorme cantidad de datos y mucho paralelismo en los cómputos. Estas, son precisamente las características de los productos escalares que estábamos tratando: solemos tener muchos productos de vectores independientes, las multiplicaciones escalares de cada producto son a su vez independientes y los tensores involucrados son habitualmente muy grandes.

Sin embargo, las GPUs se orientan a aplicaciones con gran localidad espacial en el tratamiento de los datos. Los SMs incluyen pequeñas memorias *caché* de baja latencia que permiten un rápido acceso a los datos que suelen utilizar sus *warps*. Además, comparten varios niveles de memoria superior con un acceso bastante más lento. El problema es que la reducción para acumular la suma de valores intermedios del producto escalar no se adapta bien a esta arquitectura y puede convertirse en un cuello de botella. Para estas reducciones, hacen falta continuos accesos a memoria de nivel superior con bastantes transmisiones por red. Este es un aspecto muy relevante en el que la TPU obtienen ventaja ya que, como vimos al estudiar su arquitectura, los resultados intermedios se acumulan con la transmisión directa entre celdas sin necesidad de pasar por memoria (apenas los retiene el registro de segmentación durante un ciclo).

Tras estudiar cómo se realizaría un producto escalar en los tres dispositivos, las ventajas de la TPU parecen claras. La optimización fundamental frente a una GPU es la acumulación de los valores intermedios sin almacenar estos resultados en memoria. Frente a una CPU, además de esa ventaja, la matriz sistólica aporta una capacidad paralela muy superior para multiplicar las componentes de los vectores y computar varios productos escalares a la vez; técnicas como SIMD y diversos paralelismos a nivel de hilo (*multicore* o *multithreading*) resultan insuficientes para la cantidad de operaciones independientes que suele haber.

2.2.2. Dedicación específica de recursos y diseño determinista

La CPU debe proporcionar un buen rendimiento en una amplia gama de aplicaciones y, para ello, utiliza mecanismos que consumen muchos transistores y energía: ejecución fuera de orden, predicción de saltos, cambios de contexto, multiprocesamiento, SIMD, etc. En este sentido, hay mucho margen de mejora si nos centramos en una tarea concreta, pues basta invertir bien los transistores teniendo en cuenta que no hacen falta dichos mecanismos ni tanta lógica de control para gestionar varias aplicaciones.

En esta línea, la TPU dedica todos sus transistores a optimizar la inferencia con redes neuronales sin preocuparse de soportar otras tareas. La TPU reduce notablemente la lógica de control frente la CPU empleando la mayor parte del *chip* para estructuras de memoria como el *buffer* unificado y para elementos de cómputo como la matriz sistólica (figura 2.14).

De manera similar sucede con las GPUs que, aunque finalmente se han adaptado a otros usos, surgieron específicamente para procesamiento gráfico. Este tipo de procesamiento demanda una enorme capacidad de cómputo paralelo con una gran localidad en el tratamiento de los datos. Por ello, las GPUs dedican la mayor parte del *chip* a los *cores* y la RAM, reduciendo notablemente la lógica de control y el tamaño de caché (figura 2.14).

Otro motivo para que la lógica de control de una TPU resulte mínima es que su comportamiento es altamente determinista. Frente a la gestión dinámica de varios procesos con tareas potencialmente distintas que realiza la CPU, la TPU alberga un solo proceso de inferencias cuya ejecución se conoce por completo en tiempo de compilación [15].

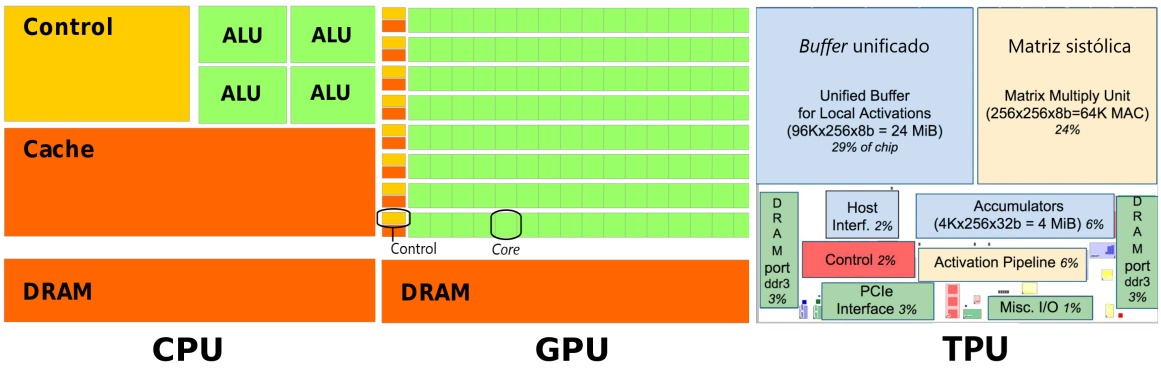


Figura 2.14: Comparativa del espacio de *chip* dedicado por cada procesador a sus componentes. Para la TPU se muestran porcentajes concretos del modelo Cloud TPUv1. (Figura creada a partir de una imagen de la guía de programación NVIDIA CUDA C[7]y una imagen de [2]).

La dedicación específica de recursos y el enfoque determinista ayudan a mejorar el rendimiento, pero también la eficiencia energética de la TPU. No solo es que los componentes *ad-hoc* sean más adecuados para realizar los cómputos más rápido (por todo el paralelismo disponible en la

[7] `https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`

matriz sistólica), sino que se desperdicia menos trabajo gracias a que la ejecución es altamente predecible. En este sentido, ya comentamos la importancia de mantener la matriz sistólica lo más llena posible de cálculos útiles, por ejemplo, ubicando consecutivamente en sus *buffers* de lectura segmentos correspondientes a entradas distintas.

2.2.3. Aritmética de baja precisión

Otra de las claves para que una TPU mejore el rendimiento y la eficiencia energética al trabajar con redes neuronales es que utiliza menos *bits* para representar los datos de lo que resulta habitual en otros procesadores y aplicaciones.

Las CPUs suelen manejar formatos de punto flotante de 32 o 64 *bits*, aunque algunas aplicaciones utilizan menor precisión para sacarle más provecho a las extensiones SIMD (caben más operaciones en paralelo). En las GPUs comienza a ser habitual operar con punto flotante de 16 *bits*, aunque muchas tareas gráficas siguen necesitando el tradicional punto flotante de 32. Por su parte, en las TPUs también es habitual utilizar representaciones de punto flotante de 16 *bits*, aunque algunos modelos son capaces de realizar las inferencias con punto fijo de tan solo 8 *bits*.

Representar los datos utilizando menos *bits* presenta importantes ventajas que las TPUs tratan de aprovechar. Por una parte, el tamaño de las unidades aritméticas es menor y hace falta conmutar menos transistores en cada operación. Debido a ello, caben más unidades de cálculo en el mismo espacio y la eficiencia energética aumenta porque cada operación consume menos energía. Por otro lado, los datos requieren menos espacio de almacenamiento y, por lo tanto, caben más valores en memorias de baja latencia como registros o *caché* [17].

A pesar de estas ventajas, hay importantes perjuicios asociados que impiden reducen la precisión en muchas aplicaciones. El inconveniente más obvio es que la cantidad de valores representados se reduce exponencialmente (con 16 *bits* se representan 2^{16} números, mientras que con 8 *bits* apenas 2^8 valores). En el caso de los números enteros esto simplemente supone un rango de representación menor, pero con los números reales también implica mayor error por redondeo.

Un computador no puede representar los infinitos números de un intervalo real. Al operar con ellos, las máquinas pueden introducir error tanto en los operandos como en el resultado (el valor exacto al operar números máquina puede no ser representable y redondearse a uno que sí lo sea). En este sentido, se define $\varepsilon_{\text{máq}}$ como el error máximo relativo al operar con cierto formato de representación. Por su definición, se tiene que:

$$\big|\text{OpMáquina}(x,y) - \text{OpExacta}(x,y)\big| \leq \big|\text{OpExacta}(x,y)\big| \cdot \varepsilon_{\text{máq}} \quad \forall x, y \in \mathbb{R}_{\text{representable}}.$$

Este error aumenta cuando el número de *bits* de la representación disminuye. En particular, $\varepsilon_{\text{máq}} = 1.2 \cdot 10^{-7}$ para el estándar de punto flotante de 32 *bits* (`fp32`) y $\varepsilon_{\text{máq}} = 9.8 \cdot 10^{-4}$ para el correspondiente formato de 16 *bits* (`fp16`) [18]. Muchas aplicaciones requieren altos niveles de exactitud y no pueden permitirse reducir la cantidad de *bits* para operar. Un valor $\varepsilon_{\text{máq}} = 9.8 \cdot 10^{-4}$ no es aceptable en muchos casos, sobre todo teniendo en cuenta que es una medida relativa al valor del resultado correcto.

En el mundo del aprendizaje automático es habitual lidiar con muchas imprecisiones y estos errores resultan insignificantes en comparación; por ejemplo, es habitual arrastrar bastante error de medida al utilizar datos del mundo real. Cuando se entrena un modelo de aprendizaje automático, el objetivo es captar los patrones que son evidencia útil, descartando las variaciones sin sentido y los datos irrelevantes. Los modelos bien entrenados generalizan su aprendizaje y generan las suficientes redundancias para que pequeñas desviaciones en los datos prácticamente no afecten al resultado [19]. En este sentido, diversos estudios concluyen que el acierto de las redes neuronales apenas se degrada al reducir la precisión en entrenamiento e inferencia [20, 21].

En el entrenamiento es habitual utilizar números reales para garantizar la convergencia del algoritmo. Además, se ha comprobado que realizar las operaciones de acumulación de gradientes con formatos de menor precisión reduce notablemente la velocidad de convergencia. El problema es que los gradientes se mueven en rangos muy pequeños y resultan muy sensibles a los redondeos (suelen ser prácticamente nulos y se redondean a cero). Aunque actualmente existen variaciones de los algoritmos que resuelven estos problemas [22], sigue siendo habitual aplicar soluciones más conservadoras. En este sentido, se opta por realizar las acumulaciones en precisión simple (32 *bits*), reduciendo a media precisión (16 *bits*) el tamaño de entrada de los multiplicadores.

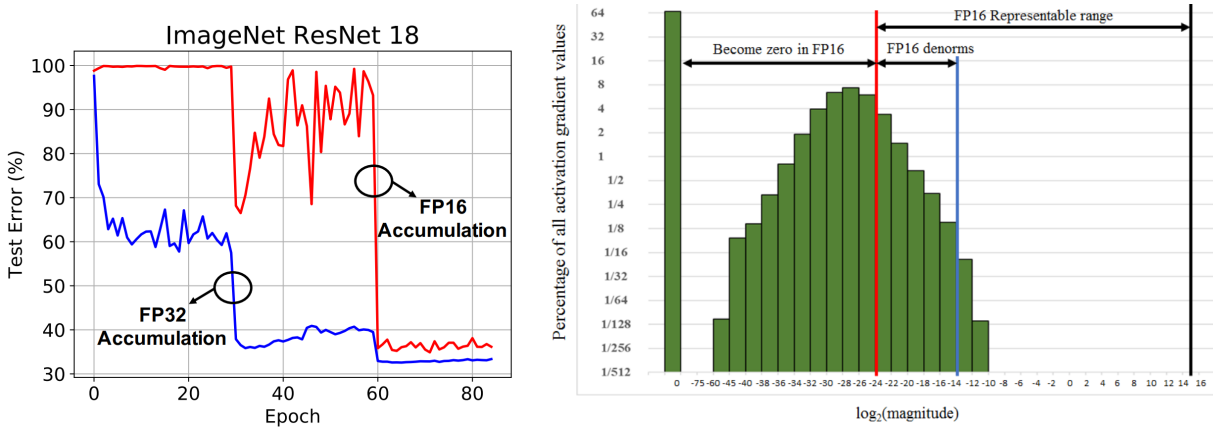


Figura 2.15: La gráfica izquierda, extraída de [22], muestra la convergencia de una red realizando las acumulaciones con `fp16` y `fp32`. La imagen derecha, extraída de [23], es un histograma de porcentaje con las magnitudes (en escala logarítmica) de los gradientes que se acumulan en una retropropagación; se indica cuáles de ellos se redondearían a cero con `fp16`.

La utilización mixta de formatos obliga a realizar abundantes conversiones que las TPUs tratan de abaratar. En lugar de utilizar `fp16`, las TPUs utilizan un formato conocido como `bfloat16` con el mismo tamaño de exponente que `fp32` (8 *bits*). El exponente determina el rango representable, y es interesante que ambos formatos le dediquen el mismo tamaño para evitar reescalados durante la conversión. Además, según afirman desde Google, las redes neuronales son mucho más sensibles al tamaño del exponente que al de la mantisa [24]. La mantisa consiste en los *bits* enteros y fraccionarios del número representado, y simplemente habrá que truncarla o extenderla para realizar la conversión. El tamaño de los multiplicadores depende cuadráticamente de la longitud de la mantisa y, por lo tanto, para `bfloat16` es la mitad que para `fp16` ($8^2/11^2 \simeq 0.5$, considerando el bit principal implícito de mantisa) [24].

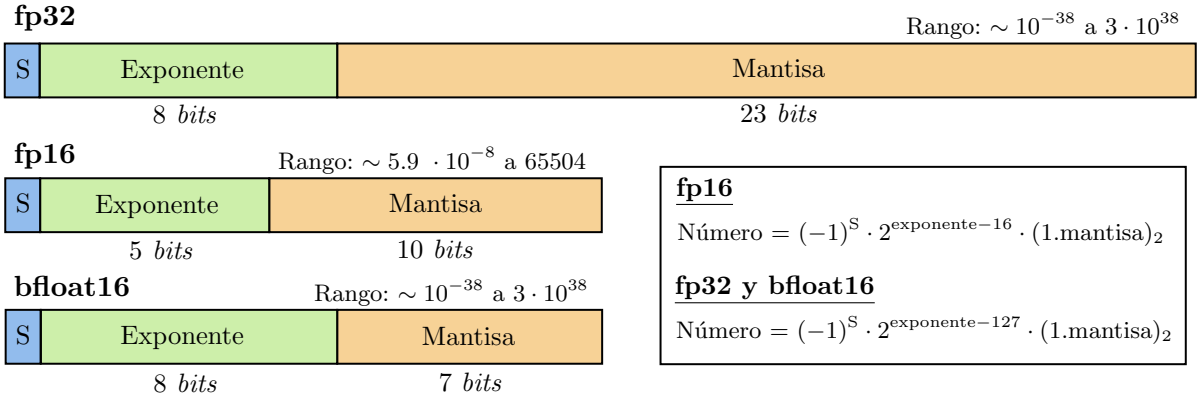


Figura 2.16: Formatos de punto flotante `fp32`, `fp16` y `bfloat16` junto a los rangos que alcanzan. Se muestra la expresión que permite determina el valor representado con cada uno.

Para realizar inferencias, es posible prescindir de los números reales y reducir aún más el tamaño de representación pues los pesos ya están ajustados y no hay que garantizar la convergencia del algoritmo que los modifica. Los modelos se convierten a una representación menos precisa (habitualmente enteros de 8 *bits*) mediante un proceso de 'cuantización', cuyo impacto debería ser mínimo sobre redes robustas bien entrenadas.

Los números enteros no se representan con formatos de punto flotante como los de la figura 2.16, sino con formatos de punto fijo. La principal diferencia es que no dedican *bits* al exponente ya que la coma binaria ocupa la misma posición para todos los números[8]. Aunque este tipo de formato es menos flexible y los rangos representables son menos amplios, utiliza aritmética entera cuyo *hardware* es más barato y eficiente. En el caso particular de los números enteros, es habitual interpretarlos en complemento a 2 en lugar de con magnitud y signo.

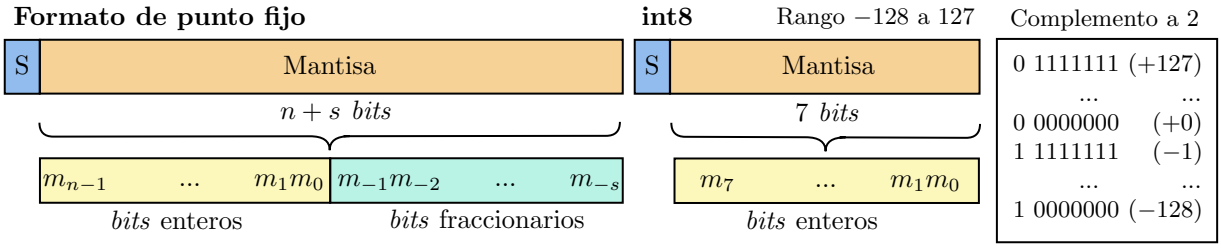


Figura 2.17: Formato genérico de punto fijo y caso particular para `int8` para representar enteros de 8 *bits* interpretando el número en complemento a 2.

Para evitar los errores de cuantización, las últimas versiones de Cloud TPU infieren con la misma precisión del entrenamiento (combinación de `bfloat16` y `fp32`). Sin embargo, la versión comercializada para *edge computing* va un paso más allá y trabaja mayoritariamente con enteros de 8 *bits*. La aritmética entera de baja precisión reduce notablemente el coste y el consumo energético del dispositivo. Esta versión se orienta a entornos donde estas características priman frente a pequeños errores por cuantización. En el capítulo 5, estudiaremos la pérdida de calidad en la inferencia al utilizar enteros de 8 *bits* para modelos entrenados con `fp32`.

2.3. Edge TPU

Una vez estudiada la arquitectura general de una TPU y sus principales ventajas frente a otros procesadores, es momento de centrarnos en el modelo que vamos a utilizar, el Edge TPU. Como ya comentamos, se trata de una versión orientada a realizar las inferencias cerca de los dispositivos que generan la entrada (*edge computing*). Este tipo de procesamiento presenta importantes ventajas frente a tratar los datos en la nube: reduce la latencia para transferir los datos (los envíos son a menor distancia), beneficia a otros servicios que usan el *cloud* al disminuir la congestión de la red y aumenta el control sobre los datos evitando compartirlos con terceros.

Figura 2.18: Esquema que muestra la diferencia entre la computación *cloud* y *edge computing*.

[8] El exponente coloca la coma binaria porque eleva a una potencia de 2 que multiplica a la mantisa (ver figura 2.16).

2.3.1. Características del Edge TPU

El Edge TPU es un modelo de gama baja pensado para entornos donde la eficiencia energética, el tamaño o el precio priman sobre obtener el máximo rendimiento. Mientras un Cloud TPUv3 tiene 16 matrices sistólicas con un rendimiento pico de 480 TOPS (teraoperaciones por segundo, es decir, 10^{12} ops/s), el Edge TPU consta de una sola matriz que idealmente podría alcanzar 4 TOPS[9]. Esta enorme diferencia refleja el increíble poder de cómputo de la versión *cloud*, que no debe desmerecer el rendimiento del Edge TPU. Un rendimiento de TOPS (10^{12} ops/s) es bastante superior al de muchas CPUs de gama alta que se mueven en las decenas o como mucho centenas de GOPS (10^9 ops/s).

Más adelante estudiaremos el tiempo de inferencia en función de diferentes parámetros y veremos que, para muchos modelos, el Edge TPU rinde mejor que una CPU de altas prestaciones. Por el momento, podemos fijarnos en los *benchmarks* de Google Coral[10], que muestran importantes aceleraciones respecto a una potente CPU como el Intel Xeon Gold 6154[11]. La aceleración al utilizar el Edge TPU varía entre un $\times 8$ y un $\times 20$ para los principales modelos de predicción con imágenes, pero aumenta a $\times 1000$ para los modelos 'EfficientNet-EdgeTPU'. Estas redes neuronales, que ponen de manifiesto la importancia del modelo, han sido diseñadas por Google para optimizar tanto la latencia como el acierto en esta TPU [25].

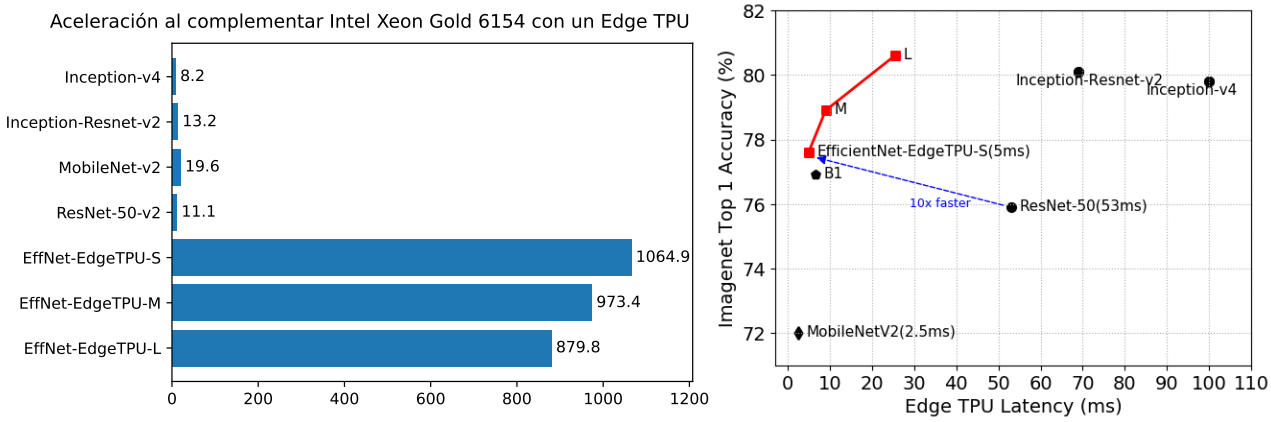


Figura 2.19: A la izquierda, un gráfico de barras que muestra, para varios modelos famosos y los 'EfficientNet-EdgeTPU', el *speedup* de inferencia al complementar el Intel Xeon Gold 6154 con un Edge TPU. A la derecha, un gráfico extraído de [25] que muestra la latencia y el acierto de los anteriores modelos, destacando los 'EfficientNet-EdgeTPU'.

Como ya explicamos, una TPU es un procesador complementario a una CPU y, por ello, Google comercializa el modelo Edge en módulos fáciles de conectar a otros sistemas. Aunque puede adquirirse como *chip* soldable a la placa, la línea de productos Coral[12] ofrece esta TPU integrada en módulos de tipo M.2, mini PCIe o incluso USB. También existe la opción de adquirirla integrada un sistema completo con una CPU de bajo consumo, una GPU, Wi-Fi y Bluetooth. Nosotros utilizaremos el módulo `M.2 A+E key`[13], que se conectan fácilmente a través de un puerto PCIe y puede adquirirse por apenas 24.99$. Su conexión con el *host* es más rápida que la del módulo USB y, como se muestra en la figura 2.20, su tamaño es muy reducido.

[9] `https://coral.ai/docs/edgetpu/faq/#how-is-the-edge-tpu-different-from-cloud-tpus`

[10] `https://coral.ai/docs/edgetpu/benchmarks/`

[11] `https://www.intel.es/content/www/es/es/products/sku/120495/intel-xeon-gold-6154-processor-24-75m-cache-3-00-ghz/specifications.html`

[12] `https://coral.ai/products/`

[13] `https://coral.ai/products/m2-accelerator-ae`

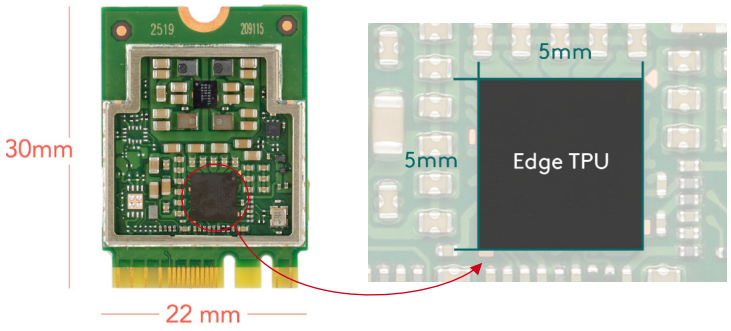


Figura 2.20: Módulo `M.2 A+E key` que integra un *chip* Edge TPU.

Según las especificaciones[14], el Edge TPU podría alcanzar su rendimiento máximo (4 TOPS) con una potencia disipada de 2 W, lo que supone una eficiencia energética de 2 TOPS/W. Para hacernos una idea de lo que esto significa, pensemos que la potencia promedio de una CPU habitualmente está entre 55 y 95 W, mientras que la potencia térmica de diseño (valor a partir de la cual se reduce su rendimiento) suele estar entre 100 y 150 W. En la práctica, la TPU no alcanza el máximo rendimiento y la potencia ni siquiera alcanza los 2 W. En la tabla 3 de las especificaciones se muestra un consumo sostenido a máxima frecuencia de 1.4 W para el modelo MobileNet v2 y de 0.7 W para Inception v3.

El Edge TPU cuenta con una memoria RAM de 8 MiB que se utiliza para almacenar las instrucciones, las entradas, las salidas y, sobre todo, los pesos del modelo. Cuando los pesos no caben completamente en esta memoria se utiliza el almacenamiento de la CPU *host*. Los pesos almacenados en memoria externa se cargan a través del bus PCIe que conecta ambos dispositivos. En los experimentos del capítulo 4 veremos la importancia del uso de cada memoria para el rendimiento.

Mientras las Cloud TPUs están optimizadas para modelos TensorFlow, el Edge TPU trabaja con modelos de tipo 'Lite'. TensorFlow Lite es un conjunto de herramientas específicamente orientado a *edge computing*, que reduce los costes de ejecución sobre placas como Rasberry Pi, dispositivos móviles o el propio Edge TPU. Sus modelos son más ligeros y están adaptados a las necesidades particulares de este tipo de computación y dispositivos. No hay problema en utilizarlos ya que, con las siguientes líneas de código Python, se puede convertir un modelo TensorFlow a su correspondiente versión 'Lite':

```
import tensorflow as tf
# Indicamos la ruta al directorio con el modelo TensorFlow
converter = tf.lite.TFLiteConverter.from_saved_model(model_saved_dir)
# Convertimos a TensorFlow Lite
tflite_model = converter.convert()
```

Como ya anticipamos, el Edge TPU opera exclusivamente con valores enteros, de tan solo 8 *bits* para las multiplicaciones. Esto genera importantes beneficios en cuanto a rendimiento y eficiencia energética, pero implica que esta TPU solamente sirve para inferencia (las del *cloud* se usan también para entrenamiento). Además, es necesario adaptar los modelos entrenados con mayor precisión para poder ejecutarlos en esta TPU. En la sección 3.1, explicaremos cómo realiza TensorFlow Lite el proceso de 'cuantización' para transformar un modelo de punto flotante a enteros de 8 *bits*.

[14] `https://coral.ai/docs/m2/datasheet/`

Capítulo 3

Otros fundamentos teórico-prácticos

En este capítulo tratamos otros aspectos importantes para el trabajo como la cuantización de modelos con TensorFlow Lite, las redes neuronales de convolución o el aprendizaje por refuerzo. Además, explicaremos el proceso para crear modelos en el Edge TPU de cara a los capítulos 4 y 5. Previamente trataremos aspectos de las librerías Gym y RLlib, utilizadas para entrenar los modelos de los experimentos del capítulo 5.

3.1. Cuantización de modelos con TensorFlow Lite

3.1.1. Tipos de cuantización

TensorFlow Lite ofrece dos opciones para el proceso de cuantización: entrenamiento consciente de cuantización[1] y cuantización post-entrenamiento[2]. En el primer caso, durante el entrenamiento se simulan inferencias de baja precisión para que el error por cuantización sea ruido que el propio algoritmo trata de minimizar [26]. De esta forma, aunque el algoritmo tarda más en converger, los parámetros del modelo son más robustos a la posterior cuantización. En el segundo caso, el entrenamiento es ajeno a la cuantización y no sufre ninguna alteración.

El entrenamiento consciente de cuantización tiene bastante sobrecoste y es poco flexible dado que se limita a algoritmos de la API Keras[3]. Además, aunque produce menos error que la cuantización post-entrenamiento, la diferencia resulta muy sutil. De hecho, si la pérdida de acierto es un problema, la recomendación es no cuantizar o hacerlo con post-entrenamiento para precisiones más altas (p.ej. `fp16` en lugar de `int8`). En el capítulo 5, evaluaremos la cuantización post-entrenamiento que, por lo que acabamos de comentar, es la técnica más habitual.

3.1.2. Cuantización desde un punto de vista teórico

El proceso de cuantización de TensorFlow Lite se explica en sus especificaciones[4] y en un artículo allí referenciado [10]. Esta cuantización se basa en aproximar los valores reales r de los parámetros de la red (entradas y pesos) mediante valores cuantizados q de tipo `int8`. Dicha aproximación, sigue la relación lineal

$$r \simeq S\,(q - Z), \tag{3.1}$$

donde S denota el factor de escala entre los rangos, mientras Z es el valor cuantizado para $r = 0$. Con estas aproximaciones, se mapea un intervalo real $[r_{\text{mín}}, r_{\text{máx}}]$ a los enteros del intervalo $[-128, 127]$ (rango representable con `int8`). Se garantiza la cuantización Z de $r = 0$ porque, como ya comentamos, algunos tensores deben rellenarse con valores nulos. De hecho, esto supone que 0 pertenezca al intervalo $[r_{\text{mín}}, r_{\text{máx}}]$ y, siendo R el conjunto de números reales a cuantizar, los extremos se toman como $r_{\text{mín}} = \text{mín}(0, \text{mín}\, R)$ y $r_{\text{máx}} = \text{máx}(0, \text{máx}\, R)$.

Con estos extremos, se calcula la proporción entre el tamaño de los dos intervalos para determinar el valor de la constante S:

$$S = \frac{r_{\text{máx}} - r_{\text{mín}}}{127 - (-128)} = \frac{r_{\text{máx}} - r_{\text{mín}}}{255}.$$

[1] `https://www.tensorflow.org/model_optimization/guide/quantization/training`

[2] `https://www.tensorflow.org/model_optimization/guide/quantization/post_training`

[3] `https://keras.io/api/optimizers/`

[4] `https://www.tensorflow.org/lite/performance/quantization_spec`

Observamos que S puede tomar un valor real bastante grande y, por ello, se representa con un número de punto flotante (a diferencia de q y Z que son de tipo `int8`). Además, para minimizar el error por cuantización, las constantes S y Z se determinan a nivel de tensor para las entradas o incluso a nivel de eje para los pesos. Así, un tensor t puede tener su propio (S_t, Z_t) o cada valor i de la proyección en una de las dimensiones de t su propio (S_{t_i}, Z_{t_i}):

$$\begin{aligned}
&\texttt{t[:, 0, :, :]} \text{ tiene } S_{t_0} = 1.4 \text{ y } Z_{t_0} = -5,\\
&\texttt{t[:, 1, :, :]} \text{ tiene } S_{t_1} = 2.1 \text{ y } Z_{t_1} = 2,\\
&\texttt{t[:, 2, :, :]} \text{ tiene } S_{t_2} = 1.7 \text{ y } Z_{t_2} = -8.
\end{aligned}$$

En cualquier caso, todos los valores S y Z se precalculan al cuantizar, al igual que los valores q de los pesos. Durante la inferencia solamente se determinan los q propagados por la red. La propagación se basa en productos escalares entre vectores de entrada $\vec{x} = (x_1, ..., x_n)$ y vectores de pesos $\vec{w} = (w_1, .., w_n)$, que se calculan de forma aproximada:

$$x_k \simeq S_x\,(q_{x_k} - Z_x),\ w_k \simeq S_w\,(q_{w_k} - Z_w) \implies \sum_{k=1}^{n} x_k\,w_k \simeq \sum_{k=1}^{n} S_x\,(q_{x_k} - Z_x)\,S_w\,(q_{w_k} - Z_w).$$

El resultado de estos productos se interpreta de nuevo como valor aproximado $S_{\text{out}}\,(q_{\text{out}} - Z_{\text{out}})$, donde S_{out} y Z_{out} son conocidos (se fijaron al cuantizar), pero hay que determinar q_{out}. Su cálculo puede deducirse igualando y despejando de las expresiones aproximadas:

$$\begin{aligned}
&\sum_{k=1}^{n} S_x\,(q_{x_k} - Z_x)\,S_w\,(q_{w_k} - Z_w) = S_{\text{out}}\,(q_{\text{out}} - Z_{\text{out}}) \implies\\
&q_{\text{out}} = Z_{\text{out}} + \frac{S_x\,S_w}{S_{\text{out}}}\left(n\,Z_x\,Z_w - Z_x\sum_{k=1}^{n} q_{w_k} - Z_w\sum_{k=1}^{n} q_{x_k} + \sum_{k=1}^{n} q_{x_k}\,q_{w_k}\right). \qquad (3.2)
\end{aligned}$$

Para que el cálculo de la ecuación 3.2 resulte eficiente, los cómputos de inferencia deben ser productos escalares entre vectores de componentes `int8` (es para lo que está diseñada la matriz sistólica). En la práctica, solo importan los cálculos cuya complejidad dependa de n, pues aquellos de coste constante resultarán despreciables en comparación (salvo valores muy pequeños de n). Analizando los cómputos no constantes de la ecuación 3.2:

- $Z_x \sum_{k=1}^{n} q_{w_k}$ puede precalcularse al cuantizar tras determinar Z_x y $(q_{w_1}, ..., q_{w_n})$.
- $Z_w \sum_{k=1}^{n} q_{x_k}$ debería calcularse en inferencia porque los valores q_{x_k} son fruto de la propagación. Sin embargo, puede evitarse su cómputo cuantizando con $Z_w = 0$ para que se anule.
- $\sum_{k=1}^{n} q_{x_k}\,q_{w_k}$ tiene que calcularse en inferencia pues los valores q_{x_k} son fruto de la propagación.

Deducimos que, de forma análoga al caso sin cuantizar, el cómputo de inferencia es el producto escalar entre las entradas y los pesos de un nodo. No obstante, ahora se utilizan valores `int8` en lugar de un punto flotante más costoso.

No hay que hacer un producto escalar adicional porque los pesos se cuantizan con $Z_w = 0$ para evitarlo. Este tipo de cuantización se conoce como 'simétrica' porque si q es el valor cuantizado para r, entonces $-q$ es el valor cuantizado para $-r$. Resulta una cuantización razonable para distribuciones como las de los pesos, cuyos extremos se aproximan a ser simétricos respecto al 0[5]. Sin embargo, suele producir más error que escoger libremente el valor del parámetro.

[5] $Z = 0$ es una elección razonable cuando $r_{\text{mín}}$ es próximo a $-r_{\text{máx}}$ ya que, como $[r_{\text{mín}}, 0]$ se mapea a $[-128, Z]$ y $[0, r_{\text{máx}}]$ se mapea a $[Z, 127]$, la asignación de número enteros es equitativa con la amplitud de los intervalos reales. En otro caso, tomando $Z = 0$ se infrautilizan la mitad de los enteros y los valores cuantizados se algolpan en exceso en la otra mitad.

Para completar el cálculo de la ecuación 3.2, hay un par de asuntos aún por resolver: la acumulación de sumandos de los productos escalares `int8` sin desbordamientos (sin salirse del rango $[-128, 127]$) y la multiplicación por el valor real $S_x \cdot S_w \,/\, S_{\text{out}}$ usando aritmética de enteros.

Para evitar los desbordamientos, las acumulaciones se realizan con enteros de mayor rango `int32` [10]. Aunque el tamaño de representación sea superior, se sigue utilizando aritmética de enteros que resulta mucho más eficiente que operar con reales de punto flotante. Además, el principal coste de los productos escalares está en las multiplicaciones que sí se realizan con 8 *bits*:

`int32 += uint8 * uint8`.

Por otra parte, hay que tratar de multiplicar el valor de punto flotante $M := S_x \cdot S_w \,/\, S_{\text{out}}$ como entero. Según explican en [10], empíricamente se ha comprobado que su valor pertenece al intervalo $(0, 1)$ y, por lo tanto, existe $n \geq 0$ tal que $M_0 = 2^n \cdot M$ pertenece a $[0.5, 1)$[6]. Despejando M y multiplicando por el neutro $2^{-31} \cdot 2^{31}$, se obtiene la expresión:

$$M = 2^{-(n+31)}\, 2^{31}\, M_0, \text{ con } n \in \mathbb{N} \text{ y } M_0 \in [0.5, 1). \tag{3.3}$$

Como $M_0 < 1$, entonces $2^{31} M_0 < 2^{31}$ y podemos representarlo con `int32` con un error inferior a $1/2$. Siguiendo la ecuación 3.3, se multiplica por este número entero y luego se efectúa el producto por la exponencial $2^{-(n+31)}$ desplazando la coma binaria. De esta forma, se habrá cometido un error inferior a $2^{-(n+32)}$ respecto al valor original M de punto flotante[7]. Finalmente, el resultado `int32` de la multiplicación se convierte a `int8` saturando al intervalo $[-128, 127]$.

Para completar el cómputo de un nodo, el valor q_{out} se evalúa en la función de activación, produciendo una salida de tipo `int8`. Esta salida se almacena como componente de entrada de la siguiente capa o como salida de la red si es la capa final.

3.1.3. Cuantización desde un punto de vista práctico

En la práctica, la cuantización de modelos es muy sencilla, porque todo el proceso lo realiza la biblioteca de manera transparente al programador. Lo más destacable es que, para determinar correctamente los valores S y Z, hace falta calibrar el rango de los valores propagados por la red. En este sentido, hay que implementar una función que genere un *dataset* de entradas representativas para estimar la magnitud de los valores. Además, hay que indicar el tipo de las operaciones cuantizadas que, para ejecutar en el Edge TPU, deben ser `int8`.

```
# Función que genera 100 ejemplos de posibles entradas del modelo
def representative_data_gen():
   for input_value in tf.Dataset.from_tensor_slices(inputs).batch(1).take(100):
       yield [input_value]
# Se configura la conversión a TF Lite y se establecen optimizaciones por defecto
converter = tf.lite.TFLiteConverter.from_saved_model(model_saved_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
# Se indica la función que proporciona el dataset representativo
converter.representative_dataset = representative_data_gen
# Se especifica que las operaciones, entradas y salidas serán int8
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
# Se realiza la cuantización
tflite_model_quant = converter.convert()
```

[6] Moviendo n posiciones hacia la derecha la coma binaria de M hasta que el primer *bit* fraccionario sea 1.

[7] $\left|M - M_{\text{aprox}}\right| = \left|2^{-(n+31)}\, 2^{31} M_0 - 2^{-(n+31)}\, \texttt{int32}(2^{31} M_0)\right| = 2^{-(n+31)} \left|2^{31} M_0 - \texttt{int32}(2^{31} M_0)\right| \leq 2^{-(n+31)}\, 2^{-1} = 2^{-(n+32)}$.

3.2. Redes neuronales de convolución

En el capítulo 4 y el capítulo 5 trabajaremos con un tipo de redes neuronales conocidas como redes de convolución o CNNs (del inglés, *Convolutional Neural Networks*). Estas redes son actualmente muy utilizadas por su buen desempeño para automatizar tareas cuyas entradas son imágenes: reconocimiento facial, detección de objetos, clasificación de imágenes, etc. En esta sección, sintetizamos las ideas de [27] y [28] para explicar sus fundamentos básicos.

3.2.1. Intuición y funcionamiento de una capa de convolución

Las redes de convolución emulan la manera en que el cerebro humano analiza las imágenes. Cada neurona del cerebro trabaja con su propio campo receptivo y solo responde a los estímulos de una pequeña región del campo visual. Como las neuronas están interconectadas, los patrones detectados en las distintas regiones se combinan entre sí creando una visión más general. La idea es que cada neurona extrae las características de una parte de la imagen, que luego se combinan entre sí para formar patrones más complejos y amplios. Por ejemplo, en la imagen de un rostro, algunas neuronas se fijarían en los ojos, otras en la nariz, otras en la boca, siendo la combinación de estas características lo que permitiría reconstruir la cara entera.

En este sentido, las redes de convolución utilizan pequeñas matrices de pesos entrenables, con las que filtran diferentes regiones de la imagen, extrayendo características de las mismas. La idea es que las matrices se va desplazando por la imagen, aplicando cierto filtro sobre diferentes regiones. Los resultados de aplicar cada filtro sobre cada región se evalúan en la función de activación y se van guardando en una matriz de salida conocida como mapa de características. Los distintos mapas de características son la salida de una capa convolucional que puede utilizarse como entrada para otra. Los filtros de la capa inicial se aplican directamente sobre la imagen, pero los filtros de las demás capas trabajan sobre las características que reciben de la capa anterior. De esta forma, las características se combinan usando nuevos filtros sobre diferentes regiones de los mapas, proporcionando otras características más complejas y abstractas. Así, las primeras capas detectan patrones simples como líneas o curvas, que se van combinando para que las últimas puedan detectar patrones sofisticados como caras u objetos.

Una imagen se representa con varias matrices que contienen los valores de sus canales de color. Por ejemplo, usando formato RGB, hay una matriz con el valor de las componentes rojas de los píxeles, otra con las componentes verdes y otra con las azules. Sobre la matriz de cada canal se desplaza otra matriz más pequeña conocida como *kernel*. El *kernel* se aplica sobre submatrices del canal que tienen su misma dimensión, multiplicando sus valores por la correspondientes posiciones de la submatriz y acumulando los resultados (un producto escalar si aplanasemos las matrices). Los *kernels* de los diferentes canales conforman un filtro de convolución con un mapa de características como salida. Los resultados de aplicar los *kernels* de un filtro sobre la misma submatriz en diferentes canales, se acumulan entre sí proporcionando un escalar que se guarda en el mapa de características (ver figura 3.1). Después, los *kernels* se desplazan *stride* posiciones hacia la derecha (mismas filas pero aumentando *stride* las columnas) y vuelven a aplicarse para obtener la siguiente característica del mapa. Cuando el desplazamiento del *kernel* excede el ancho de la entrada, se hace el módulo con el número de columnas y se avanzan las filas. Los mapas de características obtenidos con los distintos filtros se convierten en los canales de entrada de la siguiente capa.

Habitualmente, cada capa de convolución consta de muchos filtros que extraen mapas de características diferentes combinados después de muchas formas. Los *kernels* de los filtros constan de parámetros que el algoritmo de entrenamiento trata de ajustar para mejorar la extracción y combinación de características en la tarea que el modelo aprende (los valores de los *kernels* son los pesos de la red). Los algoritmos de entrenamiento son descensos de gradiente con una función objetivo, análogos a los que se utilizan con otro tipo de redes.

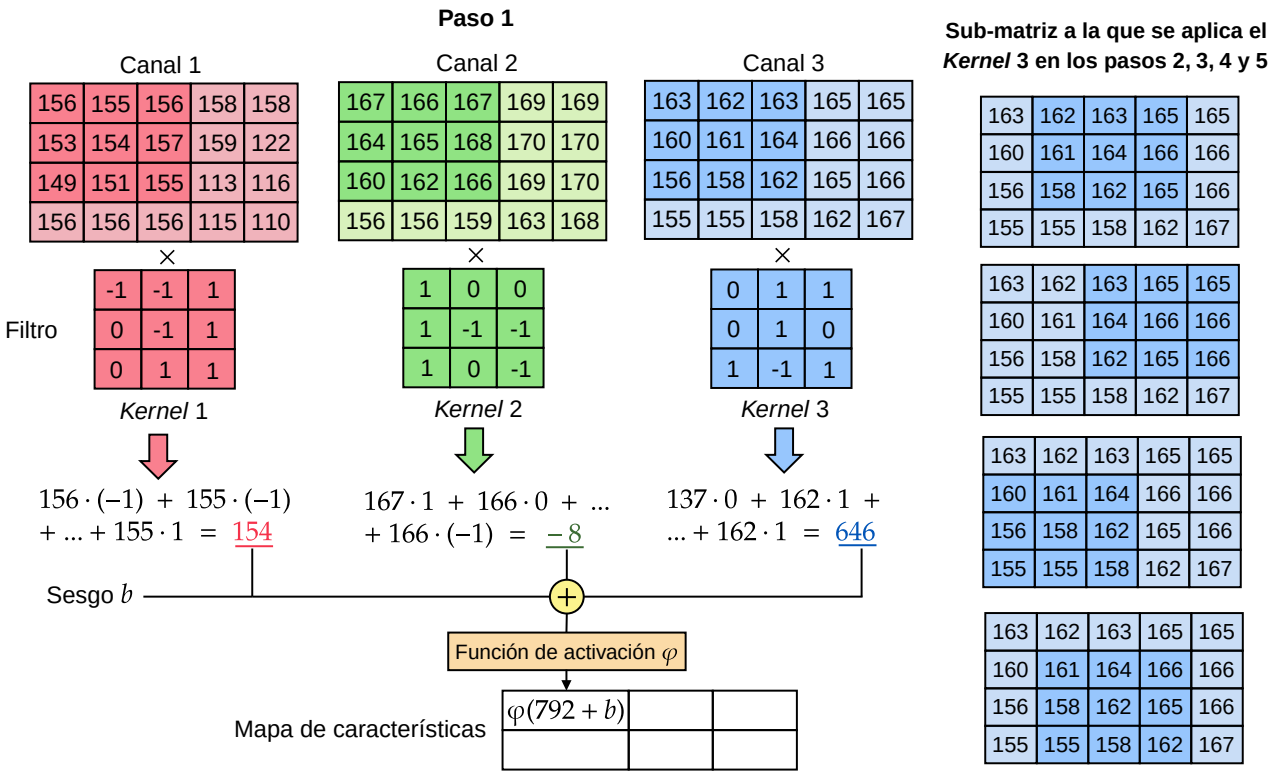


Figura 3.1: Primer paso al aplicar un filtro de dimensiones 3×3 con 3 canales de entrada (podrían ser los canales de una imagen RGB). Se destaca en una tonalidad más oscura la submatriz del canal sobre la que se aplica cada uno de los *kernels* del filtro. Se muestra también cómo cambiaría esta submatriz en el canal 3 cuando el *kernel* se desplaza con *stride* $= 1$.

3.2.2. Relleno de la entrada, capas de agrupación y capa densa final

En el desplazamiento de un *kernel* por la entrada, los valores que están en la primera o última fila, y los que están en la primera o última columna, se utilizan menos que los demás. Por ejemplo, en el desplazamiento ilustrado en la figura 3.1, el valor que está en la posición $(0, 0)$ solo se usa una vez, pero el que está en la posición $(1, 1)$ se usa cuatro veces. De esta forma, se extrae menos información de los bordes, lo que muchas veces no resulta deseable. Para que esto no suceda, se pueden rellenar las matrices de entrada con una fila y columna de ceros al principio y al final, de manera que los elementos que originalmente estaban en un borde dejen de estarlo y se utilicen las mismas veces que los demás. En este caso, la dimensión del mapa de características coincidirá con la dimensión de entrada si se utiliza *stride* $= 1$. Es por eso que a esta técnica de relleno se conoce como *same padding*, mientras que se habla de *valid padding* cuando no se utiliza relleno (ver figura 3.2).

Para reducir el coste computacional del modelo sin perder información importante, las redes de convolución suelen incluir capas de agrupación (en inglés, *pooling layers*). Uno de los objetivos de estas capas es disminuir la dimensión de los mapas de características, reduciendo sus valores mediante operaciones como la media o el máximo. De forma análoga a los filtros de convolución, estas reducciones se aplican sobre submatrices de un determinado tamaño con un cierto *stride* de desplazamiento (ver figura 3.3). Además de reducir la dimensiones, estas capas permiten que el modelo sea más flexible en cuanto la posición de los objetos a detectar. Digamos que favorecen que la red pueda detectar objetos ligeramente trasladados en la imagen de entrada (si el objeto está cerca de donde se espera que esté, es capaz detectarlo). En este sentido, constituyen un mecanismo de regularización para evitar el sobreaprendizaje.

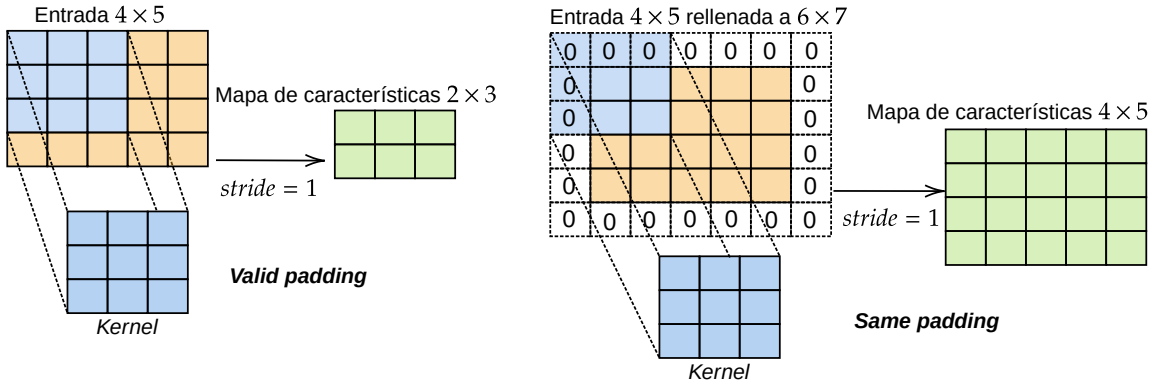


Figura 3.2: Primer paso al aplicar un *kernel* sobre una entrada sin relleno (*valid padding*) o con una fila y columna de relleno nulo por los bordes (*same padding*). Se muestran las dimensiones del mapa de características que se obtendría, asumiendo *stride* = 1.

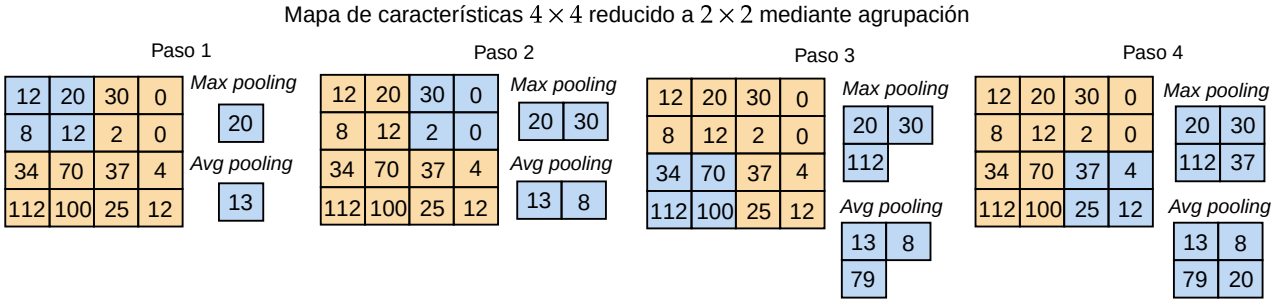


Figura 3.3: Aplicación paso a paso de un filtro de agrupación 2×2 a una entrada de tamaño 4×4 con *stride* $= 2$. Se destaca la submatriz que se reduce en cada paso y se muestra el mapa de características que se va formando tanto si se usa agrupa escogiendo el máximo (*max pooling*), como si se usa agrupa tomando la media (*average pooling*).

Por último, para que la salida tenga las dimensiones deseadas, se suele colocar al final de la red una capa densa de neuronas (todas las entradas de la capa son entradas de todas las neuronas). En general, la salida no debe ser un mapa de características, sino un vector de cierta dimensión que resulte de combinarlas una última vez. Por ejemplo, en un problema de clasificación esta capa permite obtener un vector de salida con las probabilidades de pertenencia de la imagen a cada categoría. En particular, esta capa permite por primera vez una combinación no lineal de características alejadas en el mapa (las combinaciones de las capas de convolución solo afectan a características cercanas que están en la misma submatriz).

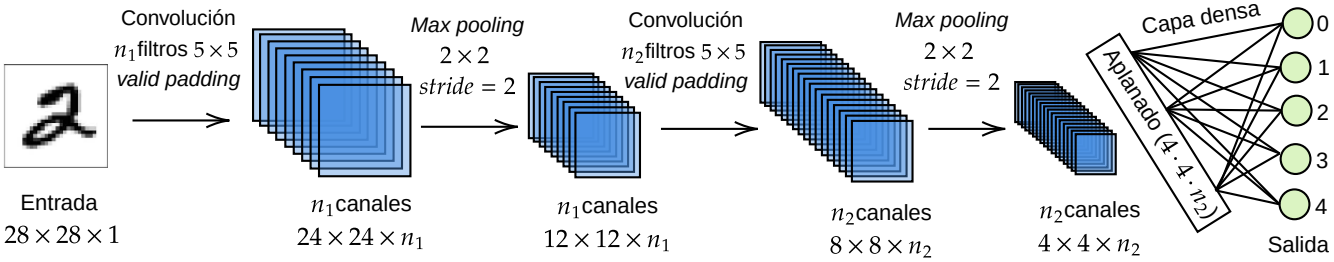


Figura 3.4: Visión general de una red neuronal de convolución con dos capas convolucionales seguidas de capas de agrupación que reducen la dimensión de los canales, y una capa densa al final para obtener un vector con 5 salidas.

3.3. Aprendizaje por refuerzo

En el capítulo 5 estudiaremos el error por cuantización en modelos neuronales entrenados mediante aprendizaje por refuerzo. En esta sección, sintentizamos las ideas básicas de [29] y [30] sobre este aprendizaje. Además, exponemos las bases del algoritmo de entrenamiento PPO (Proximal Policy Optimization), que utilizaremos en dicho capítulo. Terminamos hablamos de las librerías empleada para el entrenamiento con este tipo de aprendizaje.

3.3.1. Conceptos generales de aprendizaje por refuerzo

El aprendizaje por refuerzo es una variedad de aprendizaje automático en la que un agente mejora su desempeño en un entorno mediante la experiencia de actuación. El agente recibe una observación con el estado del entorno, que le permite decidir qué acción realizar sobre él. La acción le proporciona una recompensa y modifica el estado del entorno dando lugar a una nueva observación. El objetivo del agente es maximizar la recompensa acumulada con las acciones que realiza durante un episodio. Un episodio es la sucesión de pasos de actuación hasta que sucede un determinado evento que marca el final. Por ejemplo, en el caso de los juegos, el episodio termina cuando el jugador gana o pierde definitivamente la partida.

Figura 3.5: Esquema de interacción entre el agente y el entorno con un ejemplo de acción, recompensa y nueva observación.

Aunque el resultado de las acciones suele ser determinista en entornos sintéticos simples, en muchos entornos reales la misma acción sobre el mismo estado puede conducir a estados diferentes y proporcionar recompensas distintas. En este sentido, el entorno de aprendizaje por refuerzo se modela formalmente como un proceso de decisión de Markov, donde las transición entre estados y las recompensas obtenidas siguen distribuciones de probabilidad (proceso estocástico). Se trata de un proceso de tiempo discreto cuyos instantes se corresponden con los pasos de decisión del agente (un paso será el instante t y el paso siguiente el instante $t+1$). Además, en estos procesos se satisface la propiedad de Markov, es decir, las probabilidades no dependen de los estados anteriores sino solamente del estado y acción actual. En este sentido, la probabilidad de transitar a un estado s' realizando una acción a desde el estado s se define mediante una distribución de probabilidad condicionada por el estado y acción actuales: $\delta(s' \,|\, s, a) = \mathbb{P}_\delta(s_{t+1} = s' \,|\, s_t = s, a_t = a)$. Por su parte, la recompensa r obtenida depende de otra distribución de probabilidad completamente análoga: $R(r \,|\, s, a) = \mathbb{P}_R(R_{t+1} = r \,|\, s_t = s, a_t = a)$.

La tarea del agente en un entorno de estas características es decidir qué acción realizar en cada estado. Para ello, el agente utiliza una función π, conocida como política, que asocia a los estados una distribución de probabilidad para sus posibles acciones. Formalmente $\pi : S \to \Delta(A)$, donde S es el conjunto de posibles estados, A el conjunto de posibles acciones y $\Delta(A)$ el conjunto de distribuciones de probabilidad para A (las funciones $A \to [0, 1]$ cuyos valores sumen 1). Así, $\pi(s)(a) = \mathbb{P}(a \,|\, s)$ denota la probabilidad de tomar la acción a en el estado s con la política π.

El objetivo del agente es tomar acciones que maximicen su recompensa acumulada y, para ello, en cada paso se ajusta la política que rige su comportamiento. A través del *feedback* que proporciona el entorno en forma de recompensa, se modifica la política actual π_t a una nueva política π_{t+1} para el siguiente paso. Esta modificación se realiza siguiente cierta estrategia definida por el algoritmo de entrenamiento. Además, el algoritmo no solo tiene en cuenta el valor de la recompensa inmediata para cada acción, sino también las posibles recompensas que puede obtener en el futuro. Una acción que de forma inmediata esté poco recompensada puede llevar a grandes recompensa en futuros pasos, y una acción muy recompensada a corto plazo puede ser contraproducente más adelante. En este sentido, se utilizan también las recompensas futuras, pero reduciendo su valor con la cantidad de pasos que faltan pasa obtenerla. Se utiliza una constante de descuento $\gamma \in (0, 1)$ para que la recompensa r, que podría obtenerse n pasos más tarde, tenga actualmente un valor de $\gamma^n \cdot r$. De esta forma, podemos definir el valor de cada política π_t en cada estado s como la esperanza matemática de la recompensas descontadas acumuladas según las probabilidades de acción dadas por la política:

$$V_\pi(s) = \mathop{\mathbb{E}}_{a_t \sim \pi(s_t)} \left[\sum_{t=0}^{\infty} \gamma^t \cdot R(s_t, a_t) \,\middle|\, s_0 = s \right] [8].$$

El algoritmo de entrenamiento modificará las políticas para que, a la larga, se maximice su valor, de forma que las recompensas acumuladas del agente mejoran. El objetivo es que la sucesión de funciones de política $\{\pi_t\}_{t=0}^{\infty}$ converja a una política óptima π^*, es decir, tal que $V_{\pi^*}(s) \geq V_\pi(\mathrm{s})$ para toda política π y estado s. El teorema 17.8 de [29] garantiza la existencia de una política óptima para cualquier proceso de decisión de Markov finito (el conjunto de estados S y el conjunto de acciones A son finitos). Además, garantiza que existe una política óptima determinista π^*, es decir, tal que para cada estado s existe una acción a de forma que $\pi^*(s)(a) = 1$. Esta política indica la secuencia exacta de acciones que maximiza el valor esperado.

Para aproximarse a una política óptima, los algoritmos de entrenamiento tienen que balancear entre exploración y explotación. Por una parte deben explorar estados desconocidos o poco visitados para ganar información sobre su valor y recompensa. Por otra parte deben explotar la información conocida para maximizar el valor de la política. Lo habitual es que las primeras iteraciones de entrenamiento apuesten mucho más por la exploración que por la explotación, y con el avance de las iteraciones la explotación se vaya imponiendo.

3.3.2. Algoritmo PPO (Proximal Policy Optimization)

A continuación sintetizamos las ideas de [31] y [32] para explicar los fundamentos teóricos del popular algoritmo de aprendizaje por refuerzo PPO (Proximal Policy Optimization). Emplearemos este algoritmo para el entrenamiento de los modelos del capítulo 5.

El algoritmo PPO mejora los resultados del algoritmo TRPO (Trust Region Policy Optimizacion), abordando de manera distinta el mismo problema de optimización. El problema que plantea el algoritmo TRPO implica un estimador temporal $\hat{A}_t$, para una función de ventaja A que mide el valor relativo de una determinada acción en un estado:

$$A(s, a) = \mathop{\mathbb{E}}_{a_t \sim \pi(s_t)} \left[R(s, a) + \sum_{t=1}^{\infty} \gamma^t \cdot R(s_t, a_t) \,\middle|\, s_0 = s, a_0 = a \right] - V_\pi(s).$$

La expresión anterior mide la ventaja de la acción a en el estado s, ya que calcula la diferencia entre el valor esperado realizando a como acción inmediata y el valor sin fijar la próxima acción. Utilizar esta función es preferible a usar el valor, porque se reduce la varianza de la estimación.

[8] Para simplificar la notación y no complicar innecesariamente las fórmulas, no se refleja explícitamente la aleatoriedad de la función de transición δ y de la función de recompensa R (modelada con las distribuciones de probabilidad $\mathbb{P}_\delta$ y $\mathbb{P}_R$). La notación utilizada es la correspondiente a entornos deterministas donde la transición y recompensa siempre son las mismas dada una acción en un estado.

Además, el estimador temporal de ventaja $\hat{A}_t$ se calibra mediante la relación de probabilidades entre la nueva política y la política anterior: $\pi_\theta(s_t)(a_t) \,/\, \pi_{\theta_{\text{old}}}(s_t)(a_t)$, donde θ_{old} y θ son los parámetros que definen la política antes y después de un paso de actualización. El algoritmo trata de encontrar nuevos parámetros θ que maximicen la esperanza empírica media de la ventaja calibrada en función del tiempo:

$$\max_{\theta} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(s_t)(a_t)}{\pi_{\theta_{old}}(s_t)(a_t)} \cdot \hat{A}_t \right].$$

La función objetivo aproxima localmente la ventaja de la política π_θ, pero se vuelve menos precisa cuanta mayor diferencia exista entre la política antigua y la nueva. La función objetivo puede indicar que una política muy distinta a la actual tiene una enorme ventaja respecto a ella, pero esta estimación es poco fiable. En este sentido, se intenta restringir la optimización para que π_θ no cambie demasiado respecto a $\pi_{\theta_{old}}$. Para ello, se utiliza la divergencia KL, que permite cuantificar la diferencia entre dos distribuciones de probabilidad[9]. Fijando una cota superior δ para esta divergencia entre políticas, se obtiene el problema de optimización con restricciones:

$$\begin{aligned} \max_{\theta} \quad & \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(s_t)(a_t)}{\pi_{\theta_{old}}(s_t)(a_t)} \cdot \hat{A}_t \right] \\ \text{sujeto a} \quad & \hat{\mathbb{E}}_t \left[\text{KL} \left(\pi_\theta(s_t), \pi_{\theta_{\text{old}}}(s_t) \right) \right] \leq \delta. \end{aligned}$$

Para resolver este problema, se realizan desarrollos de Taylor de la función objetivo y de la restricción. En el caso de la función objetivo, es suficiente un desarrollo de orden 1 (hasta la primera derivada), ya que el término de segundo orden es prácticamente nulo. Sin embargo, para la divergencia KL el término de segundo orden puede tener un valor relativo importante y hace falta calcular la matriz de derivadas parciales segundas. Obtener las derivadas segundas es muy costoso computacionalmente y, además, la resolución analítica del problema exige calcular la inversa de la matriz que forman, lo que también tiene un coste elevado. Para abordar estos inconvenientes surgen dos grandes estrategias:

- Aproximar algunos cálculos de las derivadas de segundo orden y la inversa matricial.
- Añadir restricciones en la propia función objetivo, que limiten la optimización a un pequeño entorno donde los desarrollos de Taylor de primer orden sean suficientemente precisos y no haga falta calcular derivadas segundas.

Mientras el algoritmo TRPO opta por la primera estrategia, el algoritmo PPO apuesta por la segunda. Concretamente, existen dos versiones del PPO que limitan de forma distinta la optimización a una estrecha región de confianza. A continuación explicamos estas versiones.

PPO con coeficiente de penalización KL adaptativo

La primera versión del PPO cambia la restricción de la divergencia KL por una penalización en la función objetivo. Si al estimador de ventaja le restamos la divergencia KL, la propia maximización tenderá a reducir la divergencia siempre que no sea muy malo para la ventaja.

Para ponderar la penalización, la divergencia se multiplica por un coeficiente β. Elegir adecuadamente este coeficiente es realmente difícil, y el PPO trata de adaptarlo dinámicamente según las circunstancias. El objetivo es que la penalización sea suficientemente restrictiva para que el entorno sea pequeño, pero a la vez no sea demasiado estricta para que realmente se exploren cambios de política. En este sentido, si se detecta una divergencia muy grande entre políticas consecutivas, se aumenta el valor de β para endurecer la restricción. Por su parte, si la divergencia es muy pequeña se reduce el valor de β para relajar la restricción ampliando el entorno de optimización. A continuación se definen los pasos para actualizar la política y la penalización:

[9] La divergencia KL entre dos distribuciones de probabilidad P y Q se define como: $\text{KL}(P, Q) = \mathop{\mathbb{E}}_{x} \left[ln \frac{P(x)}{Q(x)} \right]$.

1. Realizar varias etapas de descenso de gradiente estocástico (que solo usan derivadas primeras) para maximizar la función objetivo penalizada:

$$L^{\text{KL-PEN}}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(s_t)(a_t)}{\pi_{\theta_{old}}(s_t)(a_t)} \cdot \hat{A}_t - \beta \cdot \text{KL}\left(\pi_\theta(s_t), \pi_{\theta_{\text{old}}}(s_t)\right) \right].$$

2. Actualizar el coeficiente β como se indica a continuación. Sea $d = \hat{\mathbb{E}}_t \left[\text{KL}\left(\pi_{\theta_{\text{opt}}}(s_t), \pi_{\theta_{\text{old}}}(s_t)\right) \right]$ la divergencia respecto a la política elegida en el paso 1 y d_{ref} una divergencia de referencia:
   - Si $d < d_{\text{ref}} \,/\, 1.5$ entonces $\beta \leftarrow \beta \,/\, 2$.
   - Si $d < d_{\text{ref}} \cdot 1.5$ entonces $\beta \leftarrow 2 \cdot \beta$.

Con la actualización de política anterior se pretende mantener siempre una divergencia similar a d_{ref}. Aunque el valor de β puede no ser el adecuado inicialmente, el algoritmo lo ajusta rápidamente. Las constantes 1.5 y 2 utilizadas para su actualización son elegidas heurísticamente, pero el algoritmo no es muy sensible a ellas.

PPO con objetivo recortado

Otra versión del algoritmo PPO, que funciona incluso mejor que la anterior, consiste en recortar la relación de probabilidades entre políticas de la función objetivo. Esta relación indica cuánto cambia la política, de manera que, limitando su valor, restringimos la optimización a un entorno de la política actual.

Cuanto más se parezcan las políticas entre sí, más se aproximará a 1 la relación de probabilidades $r_t(\theta) = \pi_\theta(s_t)(a_t) \,/\, \pi_{\theta_{\text{old}}}(s_t)(a_t)$. En este sentido, se limita la relación al intervalo $[1-\varepsilon, 1+\varepsilon]$ para cierto $\varepsilon > 0$ pequeño. Este recorte se realiza solamente para evitar cambios de política bruscos que mejoren la estimación de ventaja. Si el cambio de política es a peor, es que no hay forma de mejorar en un entorno de la política actual (porque maximizamos), y quizás nos interese mirar más allá permitiendo mayores cambios. En este sentido, se realiza el mínimo entre la función de ventaja recortada y la función sin recortar, obteniendo así una cota pesimista. Con todo ello, la función objetivo a maximizar mediante un descenso de gradiente estocástico es:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\text{mín}\left(r_t(\theta) \cdot \hat{A}_t, \text{clip}\big(r_t(\theta), 1-\varepsilon, 1+\varepsilon\big) \cdot \hat{A}_t \right) \right],$$

donde 'clip' denota la siguiente función definida a trozos:

$$\text{clip}\big(r_t(\theta), 1-\varepsilon, 1+\varepsilon\big) = \begin{cases} 1-\varepsilon & \text{si} \quad r_t(\theta) \leq 1-\varepsilon, \\ 1+\varepsilon & \text{si} \quad r_t(\theta) \geq 1+\varepsilon, \\ r_t(\theta) & \text{en otro caso.} \end{cases}$$

La siguiente figura muestra una representación gráfica de la función con objetivo recortado.

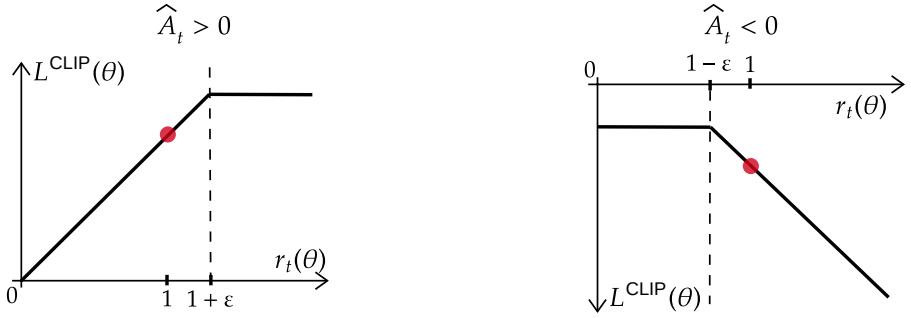


Figura 3.6: Valores de la función objetivo L^{CLIP} para distintas relaciones de probabilidad r en un instante de tiempo t concreto. Cuando el estimador de ventaja $\hat{A}_t$ es positivo, el objetivo solo se recorta por encima $(1+\varepsilon)$. Cuando el estimador es negativo, solo se recorta por debajo $(1-\varepsilon)$. Figura modificada a partir de la que muestra [31].

3.3.3. Aprendizaje por refuerzo con redes neuronales

Los algoritmos de aprendizaje por refuerzo pueden implementarse de diversas formas. Para problemas sencillos es habitual calcular tablas con las funciones de valor y política para todos los posibles estados y parejas estado-acción. No obstante, esto no es posible cuando el espacio de estados o acciones es demasiado grande. Este es el caso de juegos cuya dificultad reside en la enorme cantidad de situaciones a considerar (p. ej. el ajedrez o el juego Go). En estos entornos es imposible evaluar todas las posibilidades estado-acción, y hace falta aproximar políticas óptimas con una cantidad limitada de datos. En este sentido, resultan idóneas las redes neuronales como aproximadores universales de funciones. Las redes neuronales permiten aproximar políticas óptimas sin pasar por todos los estados, pero aún así necesitan grandes cantidades de datos (p. ej. miles de horas en un juego para explorar una cantidad de estados suficientemente grande). No obstante, su inclusión en el aprendizaje por refuerzo permite abordar problemas que son irresolubles con implementaciones clásicas.

Otra ventaja de utilizar redes neuronas en aprendizaje por refuerzo es que se pueden tomar decisiones sobre grandes entradas sin estructurar. Con otra implementación, habría que transformar las observaciones que recibe el agente a una representación simplificada; por ejemplo, en lugar de proporcionar directamente la imagen de un tablero, habría que construir un array con las posiciones de las fichas. Es muy común que las observaciones con este aprendizaje sean directamente imágenes del entorno y, por lo tanto, lo más apropiado es utilizar redes neurales de convolución (se explicaron estas redes en la sección 3.2).

La salida de la red que aproxima la política del agente debe proporcionar las probabilidades de realizar cada acción y, por tanto, termina con una capa densa con tantos nodos como posibles acciones (véase la figura 3.4). Los pesos de la red constituyen los parámetros que el algoritmo de entrenamiento ajusta para cambiar de política. El ajuste de los pesos se realiza con un descenso de gradiente estocástico que optimiza cierta función objetivo (p. ej. la función de objetivo recortado que vimos en el PPO). La función objetivo depende del valor de los estados[10], que se suele estimar con otra red neuronal. Esta otra red recibe las mismas observaciones y proporciona un único valor de salida: el valor estimado para el estado recibido.

3.3.4. Librería Gym

Gym [33] es una librería de código abierto desarrollada por la compañía OpenAI para Python. Esta biblioteca permite desarrollar y comparar algoritmos de aprendizaje por refuerzo y destaca por la variedad de entornos de aprendizaje que expone su API[11]. Gym ofrece entornos con videojuegos clásicos de la marca Atari (p. ej. PacMan o Pong) o problemas de control de *robots* simulados en el motor físico MuJoCo. Además, los entornos Gym se pueden utilizar en muchas otras bibliotecas, como la librería RLlib que utilizaremos nosotros para el entrenamiento.

Para crear un entorno con Gym basta llamar a la función `make` proporcionando el nombre como parámetro (p. ej. `env = gym.make("Pong-v0")`). Este entorno se puede resetear, obteniendo una observación inicial con la que empezar a interaccionar. El reseteo depende de una semilla de aleatoriedad que se puede fijar para comenzar un episodio en concreto, y devuelve un diccionario con metadatos de la observación (`obs, info = env.reset(seed = 0)`). Luego, se puede llamar a la función `step` con el número de la acción que se desea realizar, simulando así la interacción del agente con el entorno. Esta llamada devuelve la siguiente observación, la recompensa obtenida por la acción realizada, un *booleano* que indica si el episodio ha terminado y metadatos varios (`obs, reward, end, info = env.step(action)`). Cuando el episodio acaba, es necesario hacer un `reset` para iniciar uno nuevo. A continuación se muestra un código para la ejecución de 500 episodios fijos siguiendo las acciones que indica un modelo de aprendizaje.

[10] En el caso del PPO depende de la función de ventaja, que a su vez depende de la función de valor.

[11] `https://www.gymlibrary.dev/`

```
import gym
# Creación del entorno
env = gym.make("Pong-v0")
for episode in range(500):
    # Reseteo del entorno (primer estado del episodio actual)
    obs, info = env.reset(seed=episode)
    # Indicamos que el episodio no se ha acabado
    end = False
    # Mientras no se haya acabado el episodio
    while not end:
        # Un modelo predice la acción a realizar usando la observación
        action = model.predict(obs)
        # Se realiza la acción. Se actualiza la observación, recompensa y fin
        obs, reward, end, info = env.step(action)
env.close()
```

Con una estructura de código similar a la anterior se realizan los experimentos del capítulo 5 para estudiar el error por cuantización con diferentes modelos en el entorno "Pong-v0". A continuación, explicamos las características principales de este entorno.

Entorno Pong-v0

Pong-v0[12] es la primera versión del entorno Gym que, inspirado en un famoso videojuego de Atari, simula una partida de tenis de mesa entre dos jugadores. La simulación consiste en un tablero 2D con una pala a cada lado correspondiente a cada jugador, y una pelota que se mueve. El jugador izquierdo es controlado automáticamente por la máquina, mientras que el jugador derecho se controla con las acciones realizadas en el entorno. Las palas se pueden mover hacia arriba o hacia abajo, y chocan con la pelota cuando está en su misma posición. Cuando una pala choca con la pelota, cambia su sentido de movimiento, pero también su dirección según con qué parte haya impactado. Además, es posible golpear la pelota imprimiéndole más velocidad que simplemente dejando que choque con la pala. La pelota también rebota contra las paredes superior e inferior del tablero. Cuando la pelota sobrepasa a alguno de los jugadores, el oponente gana un punto. La partida (o episodio) se disputa hasta que algún jugador alcance 21 puntos.

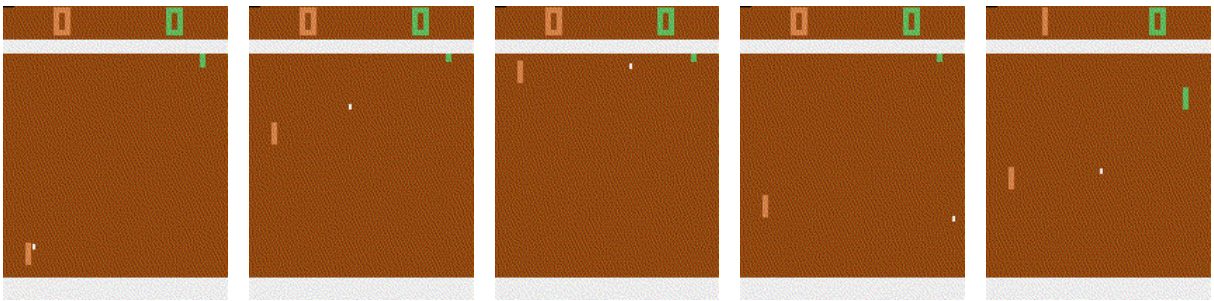


Figura 3.7: Secuencia de imágenes de Pong-v0 extraídas del *gif* de la documentación[12]. El jugador izquierdo golpea la pelota que, tras rebotar en la pared superior, sobrepasa al jugador derecho y supone el primer punto.

Las observaciones asociadas a este entorno son imágenes RGB de tamaño 210×160 (es decir, tensores $210 \times 160 \times 3$). En general, estas imágenes serán preprocesadas para reducir sus dimensiones. En la próxima sección veremos cómo funciona el envoltorio que utiliza RLlib para preprocesar estas imágenes.

[12] `https://www.gymlibrary.dev/environments/atari/pong/`

Por otra parte, existen 6 posibles acciones que los jugadores pueden realizar: no moverse, golpear, moverse hacia arriba, moverse hacia abajo, moverse hacia arriba golpeando y moverse hacia abajo golpeando. Así, la red neuronal que aproxime la política de un agente entrenado en este entorno tendrá 6 salidas con la probabilidades de tomar cada una de las acciones. Finalmente, las acciones previas a ganar un punto tienen una recompensa de $+1$, las acciones previas a perder un punto una recompensa de -1 y las demás acciones una recompensa de 0. Como las partidas se juegan hasta 21 puntos, la recompensa acumulada del agente será un valor entero entre -21 y $+21$.

3.3.5. Librería RLlib

RLlib[13] es una biblioteca de código abierto para aprendizaje por refuerzo que forma parte del *framework* para aplicaciones distribuidas Ray. Este *framework* trata de repartir las tareas entre varios actores que trabajan en paralelo. Está especialmente orientado a distribuir el trabajo entre varias máquinas, pero también permite explotar de forma eficiente el paralelismo *multicore* o el paralelismo de las GPUs.

La librería tiene una API para Python que utilizaremos para entrenar modelos neuronales con el algoritmo PPO. Esta API usa la biblioteca TensorFlow como *backend* para facilitar el entrenamiento de redes neuronales que aproximen la política y el valor de un agente. RLlib soporta una gran variedad de algoritmos de entrenamiento, entre los que se encuentran las versiones del PPO estudiadas en la sección 3.3.2. Además, permite utilizar redes de convolución cuando las observaciones son imágenes, y soporta una gran variedad de entornos, entre los que se incluyen los entornos de la librería Gym.

Para entender el funcionamiento básico de la API, a continuación se muestra un ejemplo de entrenamiento en el entorno Pong-v0 con la configuración por defecto del algoritmo PPO (la versión por defecto es con objetivo recortado). El ejemplo muestra cómo modificar la arquitectura de las redes neuronales que aproximan la política y el valor del agente modificando el campo `conv_filters` de la configuración. En este campo, se describen las capas de convolución como una lista de tuplas (`num_filtros`, `dim_filtros`, `stride`). Tras las capas descritas, RLlib implícitamente pondrá un capa densa con tantas salidas como acciones para formar la red de política, y una capa con una sola salida para formar la red de valor. El ejemplo también muestra cómo guardar el estado del agente en cierto punto del entrenamiento en un directorio externo y cómo cargarlo después (el estado del agente incluye las dos redes neuronales entrenadas).

```
import ray
import ray.rllib.agents.ppo as ppo
# Inicializamos ray indicando que podrá paralelizar hasta en 16 cores
ray.init(num_cpus=16)
# Copiamos la configuración por defecto del algoritmo PPO
conf = ppo.DEFAULT_CONFIG.copy()
# Red de convolución con 3 capas descritas como (num_filtros, dim_filtros, stride)
conf["model"]["conv_filters"] = [(16, (8, 8), 4), (8, (4, 4), 2), (256, (11, 11), 1)]
# Asignamos la configuración y entorno para obtener el agente
agent = ppo.PPOTrainer(conf, env="Pong-v0")
# Realizamos 1000 iteraciones de entrenamiento
for i in range(1, 1001):
   agent.train()
   # Cada 50 iteraciones, guardamos un checkpoint del modelo en un directorio
   if i % 50 == 0:
       agent.save(f"checkpoint_{i}")
# Restablecemos desde el directorio con el agente de la iteración 1000
agent.restore(f"checkpoint_1000")
```

[13] `https://docs.ray.io/en/latest/rllib/index.html`

Las imágenes que reciben las redes neuronales no son directamente las observaciones del entorno sino que, para facilitar el aprendizaje, RLlib las preprocesa con el envoltorio *deepmind*. En la sección anterior comentamos que las observaciones del entorno Pong-v0 eran tensores $210 \times 160 \times 3$ correspondientes a imágenes RGB. Como el color es un detalle innecesario para el aprendizaje, estas imágenes se convierten a escala de grises reduciendo así el número de canales a uno. Además, se reducen sus dimensiones a un tamaño cuadrado $dim \times dim$, que se puede configurar en el agente, pero por defecto es 84×84. Finalmente, se apila este tensor junto con los de las últimas 3 observaciones, ya que en algunas situaciones es interesante recordar los estados anteriores (un patrón clave en un juego puede estar en la sucesión de estados). Para ello, se quita la observación más antigua en cada paso de actualización (la de hace ya 4 pasos) y se añade la nueva observación preprocesada. Así, los modelos reciben tensores $dim \times dim \times 4$.

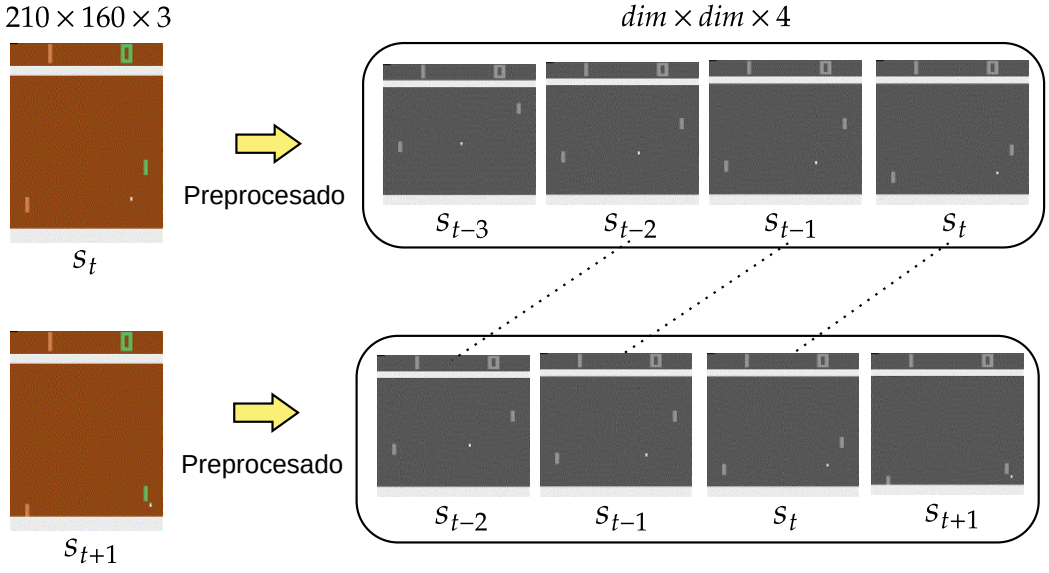


Figura 3.8: Observación correspondiente a estados consecutivos del entorno Pong-v0 antes y después del preprocesado *deepmind*. Nótese cómo se convierte la imagen a escala de grises, se redimensiona a un tamaño cuadrado y se apila junto a las imágenes de los 3 estados anteriores. Las imágenes para hacer esta figura se han extraído del *gif* de la documentación de Pong-v0.

Aunque las salidas de la red de política proporcionan las probabilidades de las distintas acciones, no están directamente en el intervalo $[0, 1]$ sino que se espera que el usuario las normalice usando la función *softmax*. Se trata de una exponencial normalizada que permite transformar valores reales arbitrarios de un vector k-dimensional a valores en $[0, 1]$:

$$\text{softmax} : \mathbb{R}^k \rightarrow [0, 1]^k$$

$$\text{softmax}(z_1, ..., z_k) = \frac{(e^{z_1}, ..., e^{z_k})}{\sum_{i=1}^{k} e^{z_i}}$$

La implementación del algoritmo PPO en RLlib utiliza dos tipos de actores para acelerar el proceso de entrenamiento: los *rollout workers* y el *driver*. En cada iteración de entrenamiento se crean varios *workers* que recopilan información sobre el entorno ejecutando acciones en paralelo según la política del agente. La información obtenida por los *workers* se centraliza y concatena en un *driver*, que la utiliza para realizar varias etapas de descenso de gradiente estocástico que actualizan la política. Tanto la cantidad de *workers* como los recursos computacionales disponibles para los actor se pueden configurar (cantidad de *cores* y GPUs). Recomendamos el Trabajo de Fin de Grado de Javier Guzmán Muñoz [34], que hizo un interesante estudio sobre el impacto en el tiempo de entrenamiento del número de *workers* y los recursos disponibles.

3.4. Flujo de transformación de modelos

Para convertir un modelo TensorFlow a un formato ejecutable en Edge TPU hay que realizar las transformaciones que indican en la página de Google Coral[14]. A continuación, explicamos el proceso que hemos seguido nosotros para ejecutar inferencias en Edge TPU con modelos creados directamente en la API de TensorFlow Keras (capítulo 4) o en la biblioteca RLlib (capítulo 5).

3.4.1. Creación, conversión a TensorFlow Lite y cuantización

En los experimentos del capítulo 4 solo vamos a evaluar el tiempo de inferencia en Edge TPU, de forma que no hace falta entrenar los modelos (importa la arquitectura de la red pero no los pesos concretos que tenga). En este sentido, podemos crear directamente la red neuronal especificando sus capas con la API de TensorFlow Keras como muestra en el siguiente código. Así, obtenemos un modelo que se convierte a TensorFlow Lite y cuantiza a `int8` como vimos en la sección 3.1.3. La única diferencia es que el modelo no se carga de un directorio sino que directamente se usa la variable del programa que lo contiene. Finalmente, el modelo cuantizado se guarda en un fichero con extensión `.tflite`.

```
from keras.models import Sequential
from tensorflow.keras import layers
# Red de 5 capas de convolución de filtros 3x3, relleno same y una entrada 64x64x3
model = Sequential()
model.add(layers.Conv2D(num_filters, (3,3), padding="same", input_shape=(64, 64, 3)))
for _ in range(4):
  model.add(layers.Conv2D(num_filters, (3,3), padding="same"))
# Configuramos el conversor directamente con la variable con el modelo keras
converter = tf.lite.TFLiteConverter.from_keras_model(model)
# Aquí iría el resto de configuración de conversión y cuantización
# ...
# Se guarda el modelo cuantizado en un fichero .tflite
open("model_quant.tflite", "wb").write(converter.convert())
```

En el capítulo 5, los modelos se entrenan con RLlib mediante aprendizaje por refuerzo, y para convertirlos a TensorFlow Lite y cuantizarlos hace falta exportar su grafo a un fichero externo (que tendrá extensión `.pb`). Una vez hecho esto, se configura el objeto conversor con la ruta del directorio donde está el fichero `.pb`, indicando las etiquetas de los nodos de entrada y salida de la red de políticas para quedarnos solo con ella. En la inferencia solo hace falta la política y, por lo tanto, desechamos intencionadamente la red de valor. La conversión y cuantización se hacen exactamente como se mostró en la sección 3.1.3, y se guardan ficheros de extensión `.tflite` para el modelo cuantizado y sin cuantizar (se usarán ambos).

```
# Aquí se crea y entrena el modelo con RLlib
# ...
# Se guarda el grafo del modelo en el directorio indicado (se creará un fichero .pb)
agent.get_policy().export_model("dir_of_model_graph")
# Conversor con dir del grafo y etiquetas de entrada-salida de la red de política
converter = tf.lite.TFLiteConverter.from_saved_model(saved_dir,
                            input_arrays=["default_policy/obs"],
                            output_arrays=["default_policy/model/conv_out/BiasAdd"])
# Se guarda el modelo sin cuantizar en un fichero .tflite
open("model.tflite", "wb").write(converter.convert())
# Aquí iría la configuración de cuantización
# ...
# Se guarda el modelo cuantizado en un fichero .tflite
open("model_quant.tflite", "wb").write(converter.convert())
```

[14] https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview

3.4.2. Compilación para Edge TPU

El siguiente paso es compilar el modelo cuantizado para poder ejecutarlo en Edge TPU. Para ello, se invoca al compilador de este dispositivo[15] por línea de comandos, proporcionando como argumento el fichero `.tflite` a compilar y opcionalmente ciertos *flags*. El resultado de la compilación es otro fichero `.tflite` con el nombre del fichero de entrada terminado en `"_edgetpu"`.

```
# Comando para compilar el modelo "model_quant.tflite".
# Genera un modelo "model_quant_edgetpu.tflite" en el directorio "compiled"
$ edgetpu_compiler --out_dir compiled model_quant.tflite
```

Durante la compilación se escribe información de reporte tanto en la salida estándar como en un fichero de *log*. Concretamente, la salida estándar muestra el uso que hará el modelo de la memoria interna y externa durante la inferencia (la memoria interna es la de la propia TPU y la externa la de su *host*). Por su parte, el fichero de *log* indica si las operaciones del modelo se realizarán en la TPU o hay algún problema para convertirlas y se realizarán en la CPU *host*.

```
--- Salida estándar ---
Input model: conv_model_quant.tflite
Input size: 1.05MiB
Output model: conv_model_quant_edgetpu.tflite
Output size: 1.19MiB
On-chip memory used for caching model parameters: 1.15MiB
On-chip memory remaining for caching model parameters: 5.77MiB
Off-chip memory used for streaming uncached model parameters: 0.00B
Number of Edge TPU subgraphs: 1
Total number of operations: 6
Operation log: conv_model_quant_edgetpu.log
Number of operations that will run on Edge TPU: 5
Number of operations that will run on CPU: 1
See the operation log file for individual operation details.
Compilation child process completed within timeout period.
Compilation succeeded!

--- Fichero conv_model_quant_edgetpu.log ---
CONV_2D       5     Mapped to Edge TPU
QUANTIZE      1     Operation is otherwise supported
```

Figura 3.9: Reporte de la compilación de un modelo en la salida estándar y en el fichero de *log*. La salida estándar muestra el uso de memoria del modelo e indica que cinco operaciones se ejecutarán en Edge TPU y una operación en CPU. El fichero de *log* muestra que la TPU ejecutará las 5 capas de convolución y la CPU solo la cuantización de las entradas.

3.4.3. Ejecución de inferencias en Edge TPU y CPU

Finalmente, para los experimentos del capítulo 4 se ejecutan inferencias en Edge TPU con el modelo compilado. Para los experimentos del capítulo 5, además se realizan inferencias en CPU con el modelo TensorFlow Lite sin cuantizar. En ambos casos se crea un intérprete que permite gestionar los tensores de entrada y salida del modelo, así como realizar la inferencia sobre el tensor de entrada establecido con el método `invoke()`. Al crear este intérprete se indica la ruta al fichero con el modelo a utilizar y, si se quiere ejecutar en Edge TPU, un parámetro adicional proporcionando el delegado `libedgetpu.so.1`.

El siguiente código muestra cómo jugar una partida a Pong-v0 realizando las sucesivas inferencias en Edge TPU con un modelo previamente cuantizado y compilado. Los comentarios del código explican cada paso. Nótese cómo las salidas se convierten a probabilidades con la función *softmax* y cómo se elige la siguiente acción según la distribución de probabilidades resultante.

[15] `https://coral.ai/docs/edgetpu/compiler/`

```
# Intérprete con delegado para ejecutar en Edge TPU el modelo cuantizado y compilado
interpreter = tflite.Interpreter(model_path="model_quant_edgetpu.tflite",
                    experimental_delegates=[tflite.load_delegate("libedgetpu.so.1")])
# Reservamos memoria para los tensores de ambos intérpretes
interpreter.allocate_tensors()
# Obtenemos las características de entrada y salida del modelo
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
# Creamos el entorno Gym especificando la dimensión de entrada del modelo
# Le colocamos el envoltorio "deepmind" con el que RLlib preprocesa las imágenes
env = wrappers.wrap_deepmind(gym.make("Pong-v0"), dim = input_details[0]["shape"][1])
# Objeto de preprocesamiento a partir del espacio de acciones del entorno
prep = get_preprocessor(env.observation_space)
# Jugamos 500 partidas correspondiente a 500 semillas fijas
for episode in range(500):
obs, info = env.reset(seed=episode)
end = False
while not end:
    # Estableceos la observación preprocesada como tensor de entrada
    interp_tpu.set_tensor(input_details_tpu[0]["index"], prep.transform(obs))
    # Se realiza la inferencia con el modelo (en Edge TPU)
    interp_tpu.invoke()
    # Obtenemos la salida fruto de la inferencia
    output_quant = interp_tpu.get_tensor(output_details_tpu[0]["index"])
    # Elegimos la acción según las probabilidades tras aplicar la función softmax
    random.choices(range(len(output_quant)), weights = softmax(output_quant))
    # Se realiza la acción. Se obtiene la siguiente observación, recompensa y fin
    obs, reward, end, info = env.step(action)
```

Capítulo 4

Tiempo de inferencia en Edge TPU

En este capítulo analizamos, a través de los resultados de diversos experimentos, el tiempo de inferencia en el Edge TPU según algunas características del modelo utilizado y sus entradas: el tipo de capas neuronales, el número de operaciones, el espacio de memoria utilizado o tamaño del lote de entradas. A lo largo de todo el capítulo iremos presentaremos en paralelo los resultados obtenidos con modelos de capas densas[1] y los resultados para capas de convolución. El objetivo es mantener la comparativa visual en todo momento puesto que, como veremos, hay bastantes disparidades por el tipo de capa.

En relación al espacio de memoria, veremos la importancia de guardar completamente los pesos del modelo en los apenas 8 MiB de almacenamiento interno de la TPU. En este sentido, analizaremos en profundidad la solución que ofrece Google Coral para reducir el tiempo de inferencia cuando un modelo no cabe en esta memoria: su segmentación entre varias TPUs formando un *pipeline*. Realizaremos pruebas de la segmentación por defecto que permite el compilador de Edge TPU, y veremos que hay bastante margen de mejora. En este sentido, trataremos de optimizar el reparto del modelo mediante un perfilado exhaustivo de la segmentación.

Para realizar estos experimentos se ha utilizado de forma remota el equipo 'Patones' del Departamento de Arquitectura de Computadores y Automática. Este equipo tiene instaladas 8 tarjetas M.2 por PCI-Express, con un Edge TPU cada una. En particular, se utilizan varias TPUs a la vez en los experimentos de segmentación. Por su parte, la CPU que ejerce de *host* es un procesador Intel Core i9-9900K[2].

4.1. Cantidad de operaciones MAC

Como comentamos en la sección 2.1, la inferencia de una red neuronal se basa en productos escalares calculados en las celdas de la matriz sistólica mediante operaciones de multiplicación-acumulación (MAC). Por este motivo, es razonable pensar que la cantidad de operaciones MAC estará estrechamente relacionada con el tiempo de inferencia.

Elaboramos los *scripts* `FC_MACs_memory_test.py` y `CONV_MACs_memory_test.py`, que automatizan la creación y evaluación de modelos aumentando progresivamente su cantidad de operaciones MAC. El primero genera modelos con L_{FC} capas densas, variando el número de nodos n de cada una entre $N_{\text{mín}}$ y $N_{\text{máx}}$ con paso S_N. El segundo hace lo propio con L_{CONV} capas de convolución variando el número de filtros f de cada una entre $F_{\text{mín}}$ y $F_{\text{máx}}$ con paso S_F (todos ellos 3×3).

En las capas densas, cada peso se multiplica y acumula exactamente una vez y, por tanto, el número de operaciones MAC coincide la cantidad de pesos de la red[3]. La primera capa oculta consta de $I \cdot n$ pesos (siendo I el número de entradas), el resto de capas ocultas constan de $n \cdot n$ pesos y la capa de salida consta de $n \cdot O$ pesos (siendo O el número de salidas). Así, obtenemos la expresión $\#\text{MAC}\,(n) = n \cdot (O + I + n \cdot (L_{\text{FC}} - 2))$, que aumenta linealmente al incrementar n si $L_{\text{FC}} = 2$ o cuadráticamente si $L_{\text{FC}} > 2$.

---

[1] Una capa densa (o totalmente conectada) es una capa oculta de neuronas tradicionales (multiperceptrón) en que cada nodo está conectado con todos los de la capa anterior.

[2] `https://www.intel.es/content/www/es/es/products/sku/186605/intel-core-i99900k-processor-16m-cache-up-to-5-00-ghz/specifications.html`

[3] No consideramos las operaciones de sesgo porque no son MAC (no hay multiplicación). En cualquier caso, hay una cantidad lineal respecto a n que será despreciable frente a las n^2 MACs de una capa oculta.

Para las capas de convolución, los filtros de nuestro *script* se toman con desplazamiento 1 y relleno '*same*' (ver la sección 3.2). Por tanto, cada peso se multiplica y acumula una vez por cada posición de cada matriz de entrada. Así, la capa de entrada consta de $C \cdot W \cdot H \cdot f \cdot F_w \cdot F_h$ operaciones MAC (siendo C el número de canales de entrada, $W \times H$ las dimensiones de cada canal y $F_w \times F_h$ las dimensiones de cada filtro). En el resto de capas el número de canales de entrada coincide con el número de filtros f de la capa anterior. Por tanto, tenemos $f \cdot W \cdot H \cdot f \cdot F_w \cdot F_h$ operaciones en estas capas. Con todo ello, deducimos la expresión $\#\text{MACs}\,(n) = W \cdot H \cdot f \cdot f_w \cdot f_h \cdot (c + f \cdot (L_{\text{CONV}} - 1))$, que aumenta linealmente al incrementar f si $L_{\text{CONV}} = 1$ o cuadráticamente si $L_{\text{CONV}} > 1$.

Con las expresiones anteriores, los *scripts* calculan el número de operaciones MAC para los modelos generados al variar n o f. Además, miden el tiempo de inferencia de cada uno de estos modelos en el Edge TPU. Para que los resultados sean fidedignos y estables, se mide el tiempo de inferencia varias veces y se calcula la media de los valores obtenidos (el *script* recibe un argumento que indica cuántas veces ejecutar cada modelo). El tiempo de la primera inferencia se descarta porque incluye el coste de cargar el modelo en la memoria de la TPU (esto no sucede en el resto de inferencias que utilizan el modelo ya cargado).

La siguiente figura muestra, para ambos tipos de capas, el tiempo de inferencia medio y el rendimiento obtenido según el número de operaciones MAC. Para generar los modelos se ha fijado el valor de los parámetros indicado al pie de la figura.

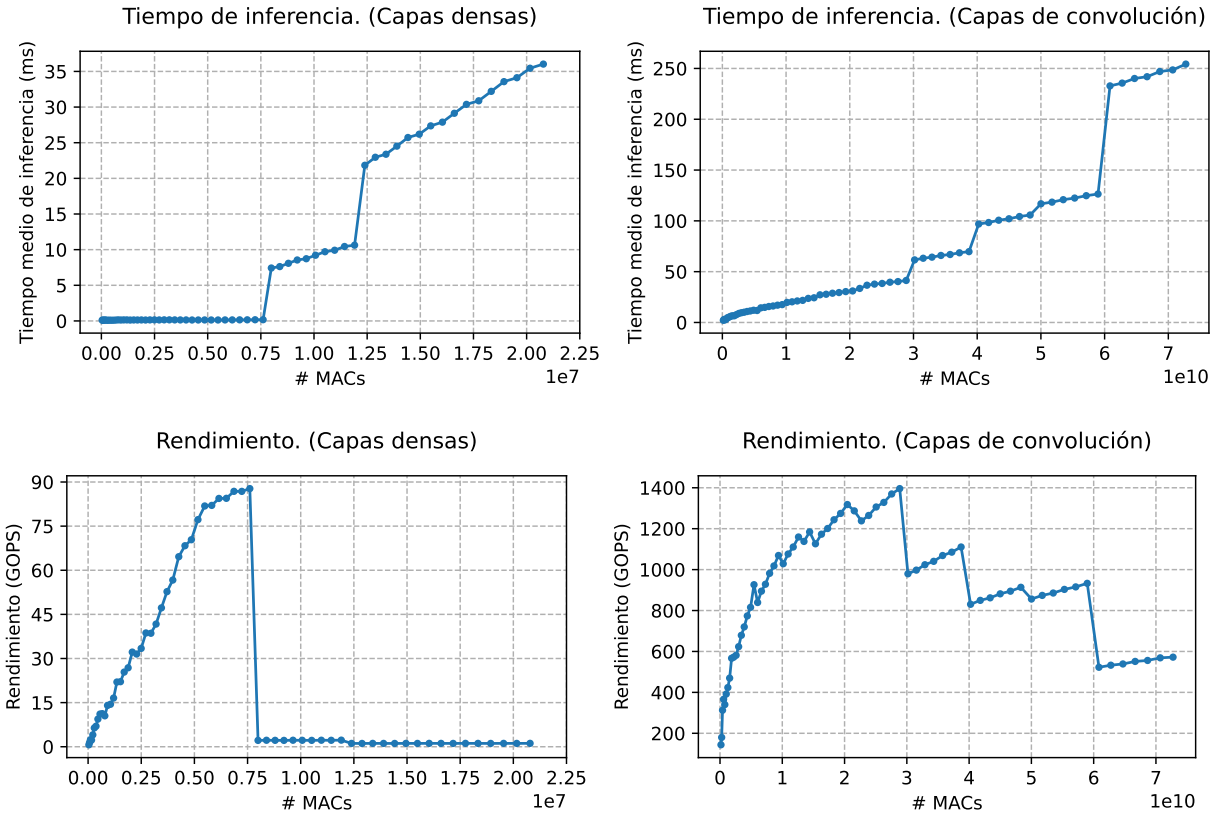


Figura 4.1: Tiempo de inferencia y rendimiento en Edge TPU en función del número de operaciones MAC. Resultados para 50 ejecuciones de cada modelo. Los modelos de capas densas se obtienen al variar el número de nodos por capa n fijando: $L_{\text{FC}} = 5$, $I = 64$, $O = 10$, $N_{\text{mín}} = 100$, $N_{\text{máx}} = 2640$ y $S_N = 40$. Los modelos de capas convolucionales se obtienen al variar el número de filtros por capa f fijando: $L_{\text{CONV}} = 5$, $C = 3$, $W \times H = 64 \times 64$, $F_h \times F_w = 3 \times 3$, $F_{\text{mín}} = 32$, $F_{\text{máx}} = 702$ y $S_F = 10$.

A continuación realizamos algunas observaciones sobre estos primeros resultados.

- **Comportamiento escalonado.**

  Para ambos tipos de capas, observamos importantes saltos que reflejan drásticos aumentos del tiempo de inferencia entre cantidades de operaciones MAC consecutivas. Dichos saltos constituyen fronteras de degradación de rendimiento que próximamente caracterizaremos y trataremos de mitigar. Los saltos delimitan "escalones" en los que el tiempo de inferencia evoluciona de forma mucho más moderada respecto al número de operaciones; de hecho, observamos que el rendimiento tiende a aumentar en dichos escalones.

- **Rendimiento muy por debajo del ideal y muy diferente según el tipo de capa.**

  En ambos casos el rendimiento de los modelos está muy por debajo de las 4 TOPS (4000 GOPS) que podría alcanzar la matriz sistólica del Edge TPU. Esta situación da a entender que las ejecuciones están limitadas por los accesos a memoria (son *memory bound*)[4]. En esa situación, la intensidad aritmética de la aplicación (relación entre el número de operaciones y los *bytes* transferidos con memoria) determina el rendimiento que podría alcanzar la aplicación: a menor intensidad aritmética menor rendimiento. Esto explicaría la enorme diferencia de rendimiento que apreciamos también entre ambos tipos de capas.

  El rendimiento de las capas densas es muy inferior al de las capas convolucionales (casi un $\times 17$ si comparamos los máximos), ya que su intensidad aritmética es bastante menor. Como ya comentamos, los pesos de las capas densas se utilizan en una sola operación MAC a lo largo de la inferencia. Sin embargo, los pesos de los filtros de convolución se reutilizan en muchas operaciones (sus núcleos se desplazan por las matrices de entrada). Así, en una capa densa se carga un dato por cada operación, mientras en una capa convolucional se carga un dato cada m operaciones (siendo $m \gg 1$).

4.2. Uso de memoria interna y memoria externa

A raíz de estas observaciones, surge interés por analizar los resultados considerando también los accesos a memoria. Dado que no existe ninguna herramienta de perfilado que permita obtener este tipo de métrica, tratamos de caracterizarla mediante datos conocidos. Concretamente, aprovechamos que el compilador genera un reporte con la cantidad de memoria interna y externa que utiliza el Edge TPU para almacenar los pesos del modelo. Esta información es un buen indicador del coste de las operaciones con memoria ya que la lectura de pesos es la operación predominante. Frente a dicha operación, la lectura de las entradas y la escritura de las salidas tienen un coste despreciable porque que sus tensores son mucho más pequeños[5].

Por otra parte, es interesante analizar el reporte del compilador sobre el uso de memoria porque, como se destaca en la documentación, las comunicaciones entre TPU y *host* son un cuello de botella que hay que tratar de evitar. En caso de almacenar parte de los pesos en memoria externa (de la CPU *host*), harán falta comunicaciones para enviar los datos. Sin embargo, cuando los pesos del modelo no caben íntegramente en memoria interna, el compilador no tiene más remedio que utilizar el almacenamiento externo. La memoria interna es de apenas 8 MiB, y no es difícil encontrar modelos TensorFlow Lite cuantizados que excedan su tamaño (p. ej. muchos modelos populares que se proporcionan en la página de Coral compilados para Edge TPU[6]).

La figura 4.2 muestra los tiempos de inferencia de los modelos anteriores junto a la cantidad de memoria interna y externa utilizada para almacenar sus pesos.

[4] Lo ideal sería ilustrarlo mediante el modelo *roofline*, pero no se ha revelado el ancho de banda pico del Edge TPU y, por tanto, nos falta información para construirlo. No obstante, los resultados aportan motivos sobrados para pensar que estamos en la zona *memory bound* del modelo.

[5] Por ejemplo, si pensamos en una capa densa con n nodos y m entradas, la capa tendrá $n \cdot m$ pesos (cada uno de los n nodos tiene m pesos distintos) y n salidas (una por nodo). Si n y m son grandes, la lectura de las m entradas y la escritura de las n salidas son despreciables frente a la lectura de los $n \cdot m$ pesos.

[6] `https://coral.ai/models/`

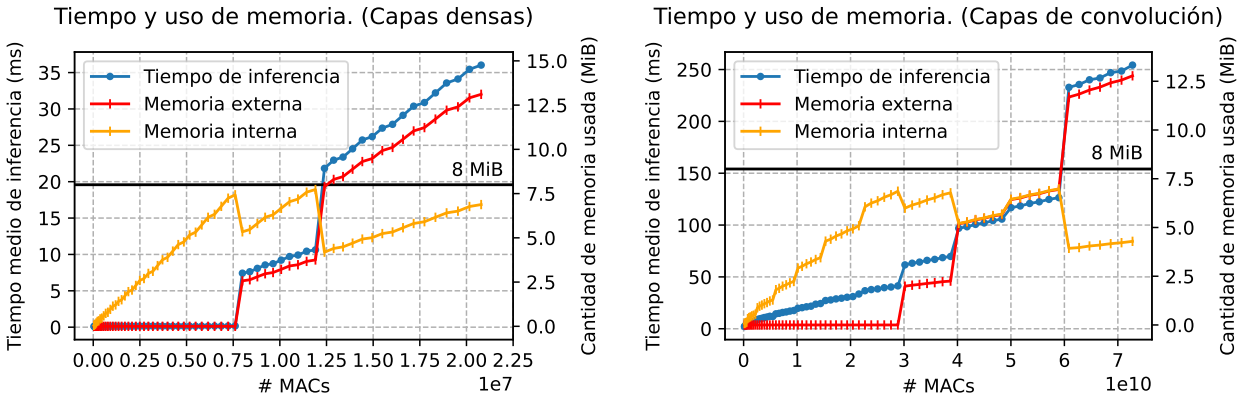


Figura 4.2: Gráficas con dos ejes de ordenadas que reflejan, junto al tiempo de inferencia de los modelos, la cantidad de memoria interna y externa utilizada para los pesos de la red.

4.2.1. Sobrecoste por almacenamiento externo

El uso de memoria para los pesos explica perfectamente el comportamiento escalonado del tiempo de inferencia. Durante un escalón, la cantidad de memoria interna aumenta progresivamente hasta alcanzar un valor próximo a los 8 MiB (tamaño de memoria de la TPU). En ese momento, el uso de memoria interna decae bruscamente coincidiendo con un incremento en forma de salto del uso de memoria externa y del tiempo de inferencia. Lo que sucede en el salto es que parte del modelo deja de almacenarse en memoria de la TPU y pasa a guardarse en el *host*.

La figura 4.2 refleja el elevado coste de las comunicaciones con el *host* para cargar los pesos de los modelos que no caben en la TPU. De hecho, por la diferencia de intensidad aritmética, el sobrecoste por comunicación tiene un peso relativo mucho mayor en las capas densas que en las capas de convolución. En una capa densa, la cantidad de pesos a comunicar entre dispositivos es mucho mayor en relación al coste computacional del modelo. Esto se aprecia claramente al comparar la ejecución en TPU con la del *host*, donde no hay sobrecoste por comunicación y el incremento de tiempo es consecuencia casi exclusiva del aumento del cómputo.

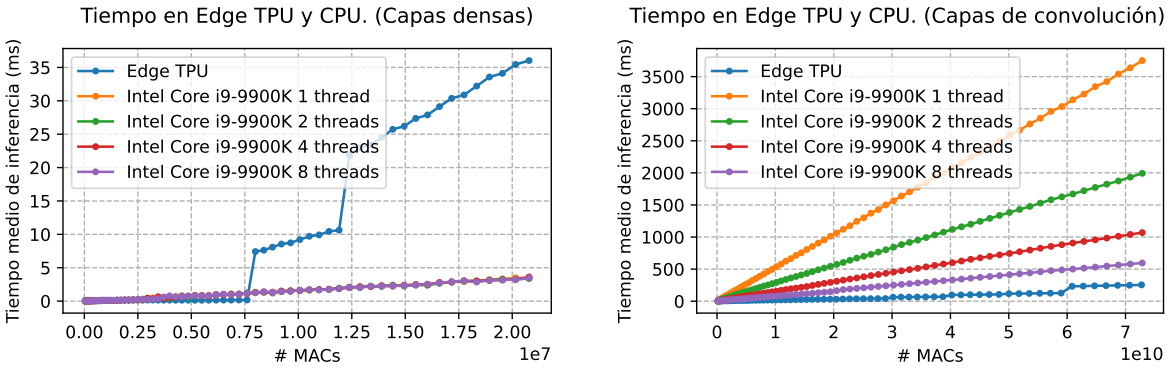


Figura 4.3: Tiempo de inferencia en Edge TPU y en la CPU *host* para ambos tipos de capas.

En las capas densas, los saltos son de gran magnitud respecto al tiempo de inferencia en CPU. Esto sucede con modelos tan ligeros que los tiempos en CPU son prácticamente iguales independientemente del número de hilos. De hecho, el tiempo de los modelos más grandes en CPU ($\sim$ 3 ms) es menor que la diferencia de tiempos en los saltos ($\sim$ 10 ms). Se confirma así el enorme coste relativo de las comunicaciones por la baja intensidad aritmética de estas capas. Por contra, los saltos de las capas de convolución son insignificantes frente a los tiempos en CPU (especialmente si limitamos la ejecución a pocos *cores*). La TPU destaca en estas capas porque el cómputo tiene mayor peso relativo y se notan más las ventajas de la matriz sistólica frente a un procesamiento más general. La diferencia es abismal a pesar de que estamos enfrentando un dispositivo de gama baja lejos de su rendimiento pico con una CPU de altas prestaciones.

Criterio de almacenamiento en el *host*

En la figura 4.2 observamos que el vuelco de datos hacia el *host* no es progresivo, sino que, llegado el punto, el compilador decide guardar en memoria externa gran parte de los pesos. Analizando el tamaño de los saltos (ver tabla 4.1 y tabla 4.2), llegamos a la conclusión de que la unidad mínima de almacenamiento es la capa neuronal. Al superar el tamaño de memoria del *chip*, se guarda en el *host* una capa neuronal completa en lugar de fraccionar sus tensores.

En las capas densas, nuestros modelos tienen $64n$ pesos en la primera capa oculta, n^2 pesos en las otras tres capas ocultas y $10n$ pesos en la capa de salida. En cuanto n es grande, el tamaño de las tres capas ocultas intermedias constituye prácticamente el total de memoria utilizada. Como podemos ver en el primer salto de la tabla 4.1, el uso de memoria del *host* (2.63 MiB) es aproximadamente la mitad del uso de memoria del *chip* (5.27 MiB) porque una de las tres capas intermedias ocultas se ha trasladado al *host* quedando las otras dos capas almacenadas en la TPU. En el segundo salto, se traslada una segunda capa oculta al *host* cuyo uso (8.04 MiB) pasa a ser aproximadamente el doble que el de memoria interna (4.04 MiB).

En nuestros modelos de convolución, el tamaño de la primera capa es despreciable frente al de las otras cuatro. La primera capa consta de $3f$ filtros (recibe siempre 3 canales), mientras las demás capas tienen f^2 filtros (reciben una cantidad de canales f). En el primer salto de la tabla 4.2, observamos el trasvase de una de las cuatro capas no iniciales hacia el *host*; por ello, el uso de memoria externa (1.99 MiB) es prácticamente la cuarta parte del uso de memoria interna (5.99 MiB). Análogamente sucede con una distribución de dos capas en cada memoria en el segundo salto (usos casi empatados). Finalmente, tenemos una distribución de una capa en TPU y tres en el *host* en el tercer salto (el *host* almacena aproximadamente el triple que la TPU).

Salto	#MACs	MiB TPU	MiB Host	Tiempo (ms)
1	0.76e7	7.43	0	0.17
	0.79e7	5.27	2.63	7.42
2	1.19e7	7.66	3.82	10.62
	1.24e7	4.04	8.04	21.83

Tabla 4.1: Saltos en los modelos de capas densas.

Salto	#MACs	MiB TPU	MiB Host	Tiempo (ms)
1	2.88e10	6.86	0	41.34
	3.01e10	5.99	1.99	61.60
2	3.87e10	6.78	2.25	69.71
	4.02e10	5.21	5.19	96.89
3	5.89e10	6.98	6.95	126.41
	6.08e10	3.93	11.69	232.82

Tabla 4.2: Saltos en los modelos de convolución.

Modelos que cabe completamente en memoria interna

Hasta el momento, hemos apreciado la influencia del uso de memoria externa, pero ¿qué sucede con los modelos que caben completamente en la memoria del *chip*? Para responder a ello, representamos solamente los modelos generados que no utilizan memoria externa (*zoom* en la figura 4.2 antes de los primeros saltos).

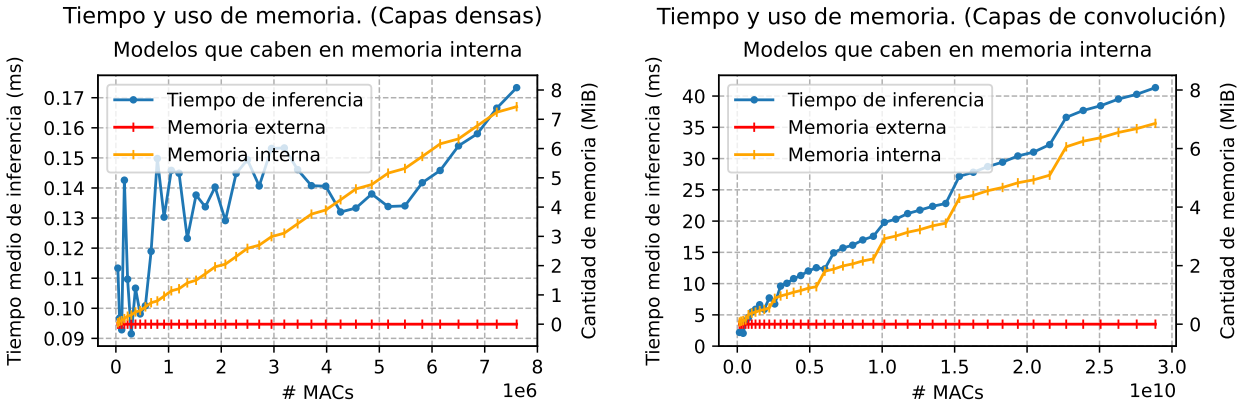


Figura 4.4: Tiempo de inferencia y uso de memoria de los modelos íntegramente almacenados en el *chip* (antes del primer salto por uso de memoria externa).

En el caso de las capas densas no apreciamos un comportamiento claro porque los modelos que caben en memoria son excesivamente ligeros y sus tiempos de inferencia son demasiado pequeños (< 0.2 ms). El problema es que el uso de memoria crece tan rápido respecto a la cantidad de cómputo que se supera el tamaño de memoria interna con muy poca carga de trabajo. La diferencia entre los modelos es tan sutil que es casi imposible captarla con las medidas de tiempo.

Para las capas convolucionales, la figura 4.4 muestra claramente cómo el tiempo de inferencia evoluciona igual que el uso de memoria interna cuando el modelo cabe íntegramente en ella. Concretamente, ambas curvas presentan el mismo comportamiento escalonado, marcado por saltos mucho más leves que los que provocaba el uso de memoria externa. El tamaño del modelo crece linealmente con el número de operaciones MAC y, por tanto, la única explicación posible para estos saltos es que se rellenen los tensores para ajustar sus tamaños a las dimensiones de la matriz sistólica (esto fue comentado al estudiar la arquitectura de una TPU en la sección 2.1). En cada salto, las dimensiones de algunos tensores superarían por primera vez cierto múltiplo del tamaño de la matriz y se estarían rellenando con ceros para alcanzar el múltiplo siguiente.

4.2.2. Segmentación en varias TPUs para reducir el uso de memoria externa

La solución propuesta por Google Coral para reducir el uso de memoria externa es segmentar el modelo en varios grupos de capas consecutivas que se ejecuten en diferentes TPUs. Así, se intenta repartir el modelo entre las memorias internas de varios *chips* y se puede aspirar a almacenar localmente más de 8 MiB (idealmente #TPUs $\cdot$ 8 MiB). Para ejecutar una inferencia basta utilizar las salidas de cada TPU como entradas para la que tiene el fragmento siguiente. Con ello, formamos un *pipeline* de TPUs en el que podemos explotar cierto paralelismo ejecutando simultáneamente distintas inferencias en diferentes TPUs.

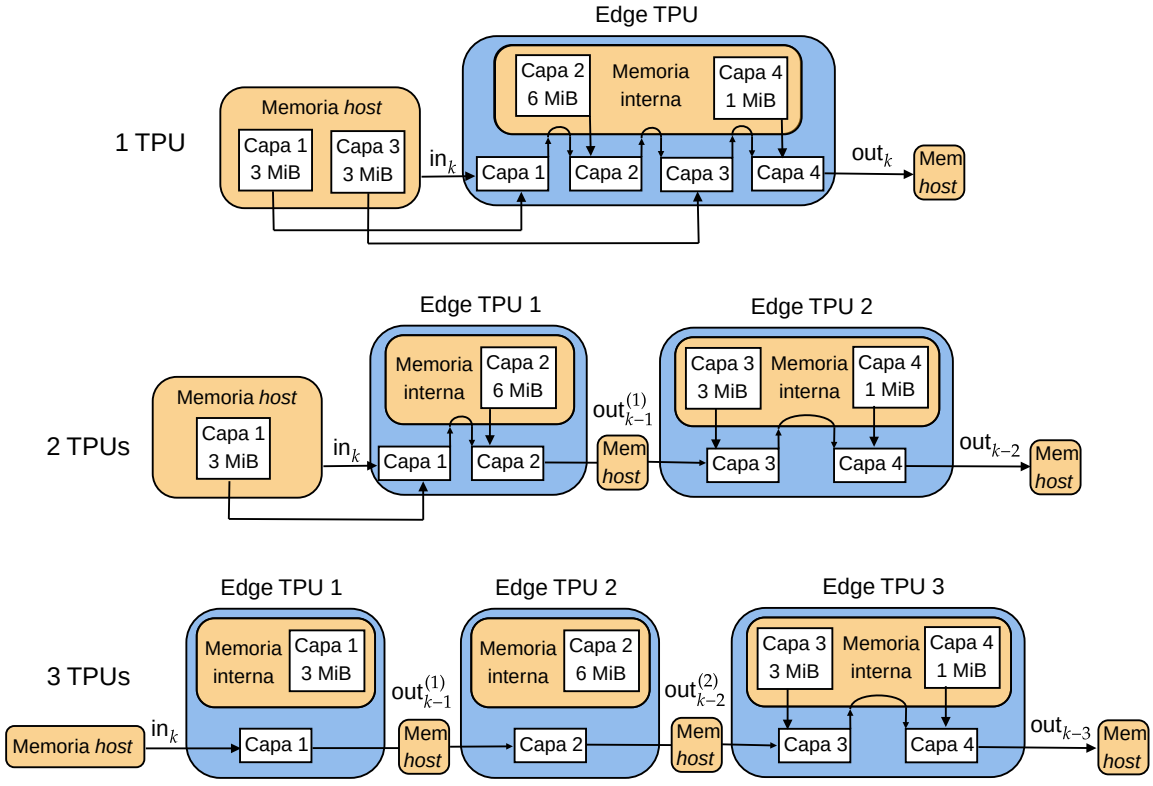


Figura 4.5: Ubicación y flujo de datos en la inferencia de un modelo para una TPU, y para segmentación en dos y tres TPUs. Los subíndices denotan el número de inferencia y los superíndices el cardinal como valor intermedio: $out^{(1)}_{k-1}$ es el primer valor intermedio de la inferencia $(k-1)$.

En la figura 4.5 apreciamos que se reduce el uso de memoria externa al segmentar el modelo entre varis TPUs (pasa de 7 MiB a 4 MiB con dos segmentos y deja de utilizarse con tres segmentos). A pesar de ello, no se aprovecha al máximo la memoria interna disponible debido a la distribución de pesos en las capas. Si no hubiese restricciones, con dos TPUs podríamos almacenar los 13 MiB del modelo. Cuando utilizamos dos segmentos, los 3 MiB de la capa 1 caben en la segunda TPU pero, por la necesidad de albergar capas consecutivas, no pueden alojarse ahí (ahí está la capa 3 pero no la 2). Como la capa 2 ocupa 6 MiB, no puede almacenarse junto a otra capa en la misma TPU y la mejor solución es guardar la capa 1 en el *host*. Este ejemplo muestra las limitaciones de esta técnica cuando hay capas muy grandes (en la práctica es raro que lo sean tanto) y los inconvenientes de que la unidad mínima de almacenamiento sea la capa.

Por otra parte, al utilizar segmentación, observamos que se realizan comunicaciones con el *host* para enviar las salidas de una TPU como entradas para la siguiente. Esto pudiera parece contradictorio porque son precisamente las comunicaciones con el *host* para enviar los datos de memoria externa las que tratamos de paliar con esta técnica. No obstante, debemos tener en cuenta que las dimensiones de los tensores que se envían al utilizar memoria externa (los pesos de las capas) son mucho mayores que las que se envían para comunicar entre sí las TPUs (las salidas intermedias). Por ejemplo, en una capa densa de n nodos con m entradas, tendremos $n \cdot m$ pesos y tan solo n salidas. En cualquier caso, el coste de comunicación de las TPUs se reflejará en nuestros próximos experimentos.

Finalmente, debemos destacar que las TPUs pueden calcular diferentes inferencias en paralelo como en un *pipeline*. Por ejemplo, en el caso con tres TPUs de la figura 4.5, observamos que la primera TPU puede ejecutar sobre la entrada de la inferencia k-ésima, mientras la segunda TPU ejecuta sobre el primer valor intermedio de la inferencia $(k-1)$-ésima y la tercera TPU ejecuta sobre el segundo valor intermedio de la inferencia $(k-2)$-ésima. Para aprovecharlo, lo más conveniente será ejecutar un lote de inferencia en forma de *pipeline* en lugar de esperar a que termine cada una toda el proceso antes de empezar con la siguiente.

Proceso de segmentación y ejecución de modelos

Para segmentar los modelos, podemos indicar al compilador de Edge TPU cuántos fragmentos formar mediante el parámetro `--num_segments`[7]. Como resultado, obtendremos tantos ficheros en formato TensorFlow Lite como fragmentos hayamos indicado. El nombre de los ficheros de salida indica el del número de fragmento y permite saber qué posición ocupa en la cadena.

```
# Genera 3 segmentos compilados: model_quant_0_of_3_edgetpu.tflite,
# model_quant_1_of_3_edgetpu.tflite y model_quant_2_of_3_edgetpu.tflite.
$ edgetpu_compiler --num_segments=3 model_quant.tflite
```

La compilación con el *flag* de segmentación funcionó perfectamente con los modelos de convolución, pero arrojó errores al utilizar capas densas. Se abrió una incidencia en el repositorio `edgetpu` de Google Coral[8], para la cual no recibimos respuesta. Después, se envió un correo a la dirección `coral-support@google.com`, al que nos contestaron indicando que se trata de un fallo interno del compilador para este tipo de capas.

Decidimos segmentar nosotros mismos estos modelos hasta que arreglasen el problema. Para ello, seguimos la misma estrategia que el compilador, que consiste en repartir uniformemente el número de capas entre los fragmentos. Como no encontramos forma de segmentar los modelos TensorFlow Lite, fragmentamos directamente el modelo Keras. La única diferencia es que el compilador segmenta el modelo cuantizado, y nosotros segmentamos y después cuantizamos. A nivel de rendimiento (que es lo que estudiamos ahora) no hay diferencia, y creemos que a nivel de comportamiento tampoco (segmentar y cuantizar serían acciones conmutables).

[7] `https://coral.ai/docs/edgetpu/pipeline/#segment-a-model`

[8] `https://github.com/google-coral/edgetpu/issues/669`

Una vez segmentado el modelo, se crea un objeto de la clase `PipelinedModelRunner` para montar el *pipeline* de ejecución. Este objeto se construye a partir de una lista de intérpretes de los segmentos del modelo. En cada intérprete se especifica el fichero con el fragmento a utilizar (parámetro `model_path`) y la TPU donde se ejecutará (campo `device` del parámetro `options`).

```
import pycoral.pipeline.pipelined_model_runner as pipeline
# Lista de intérpretes para los diferentes segmentos
interpreters = [tflite.Interpreter(
                model_path=f"{model_prefix}_quant_{seg}_of_{num_seg}_edgetpu.tflite",
                experimental_delegates=[tflite.load_delegate("libedgetpu.so.1",
                options={"device": f":{seg}"})])
                for seg in range(num_seg)]
for interpreter in interpreters:
  interpreter.allocate_tensors()
# Objeto para la ejecución del modelo en forma de pipeline
runner = pipeline.PipelinedModelRunner(interpreters)
```

Después, tal y como muestra en el ejemplo del repositorio de `pycoral`[9], ejecutamos en el *pipeline* utilizando un productor para introducir los datos y un consumidor para los extraerlos. Además, tal y como explica la documentación[10], se introduce una última entrada vacía para cerrar adecuadamente el *pipeline*.

```
def producer():
   for input_value in input_values:
     # Introduce la entrada en el pipe
     runner.push({name: input_value})
   runner.push({})

def consumer():
  for i in range(len(input_values)):
    # Se bloquea hasta recibir una salida
    output = runner.pop()
    print(f"Output {i}:", output)

start = time.perf_counter()
producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer)
producer_thread.start()
consumer_thread.start()
# Espera a que finalice la ejecución de todo el lote
producer_thread.join()
consumer_thread.join()
average_time_ms = (time.perf_counter() - start) / len(input_values) * 1000
```

Este procedimiento funcionó correctamente con los modelos de convolución segmentados por el compilador, pero dio nuevamente problemas con los modelos de capas densas que segmentamos nosotros (concretamente errores al crear el objeto con el *pipeline*). Al igual que antes, decidimos programar nosotros una ejecución equivalente, creando un hilo para la inferencia en cada TPU y utilizando colas para comunicarlas (enviar las salidas de una como entradas para la siguiente). Creamos las colas con tantas posiciones como tamaño tuviese el lote de entradas para garantizar que no había paradas por intentar escribir en una cola llena (esa acción es bloqueante).

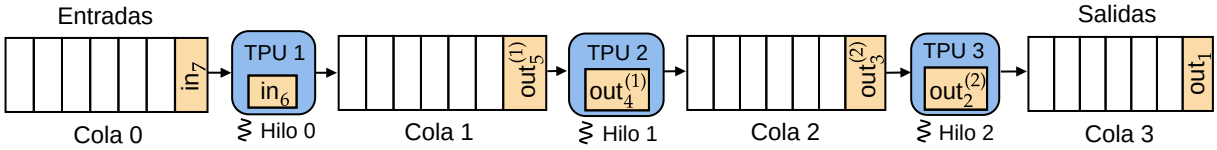


Figura 4.6: Ejemplo de una posible situación de ejecución de nuestro *pipeline* con colas para 3 TPUs y un lote de 8 entradas.

[9] `https://github.com/google-coral/pycoral/blob/master/examples/model_pipelining_classify_image.py`

[10] `https://coral.ai/docs/edgetpu/pipeline/#run-a-pipeline-with-python`

```
def seg_inference(num_segment, batch_size):
  input_details = interpreters[num_segment].get_input_details()
  output_details = interpreters[num_segment].get_output_details()
  for _ in range(batch_size):
    # Leemos de la cola de entrada (si está vacía se bloquea)
    input_val = queues[num_segment].get()
    # Ejecutamos el segmento sobre la entrada
    interpreters[num_segment].set_tensor(input_details[0]["index"], input_val)
    interpreters[num_segment].invoke()
    # Escribimos la salida en la siguiente cola
    output = interpreters[num_segment].get_tensor(output_details[0]["index"])
    queues[num_segment+1].put(output)

# Creamos las colas y colocamos las entradas en la primera
batch_size = len(input_values)
queues = [queue.Queue(batch_size) for _ in range(num_segments+1)]
for input_values in input_values:
   queues[0].put(input_values)
# Creamos, iniciamos y esperamos la finalización de los hilos
threads = [threading.Thread(target=seg_inference, args=(num_segment, batch_size))
           for num_segment in range(num_segments)]
start = time.perf_counter()
for thread in threads:
  thread.start()
for thread in threads:
  thread.join()
average_time_ms = (time.perf_counter() - start) / batch_size * 1000
```

Los propios *scripts* con los que generemos los modelos de los experimentos anteriores contienen el código para segmentarlos y en función de un parámetro que les indica el número de segmentos. Por otra parte, elaboramos los *scripts* `rollout_pipeline_batch_FC.py` y `rollout_pipeline_batch_CONV.py` que se invocan para ejecución en forma de *pipeline* si el número de segmentos es superior a uno.

Ejecución con un lote de entrada unitario

Comenzamos ejecutando los modelos de los experimentos anteriores segmentados en dos, tres y cuatro TPUs con un lote de una sola entrada. Con este tamaño de lote, no hay paralelismo en la ejecución y observamos el impacto positivo de reducir las comunicaciones por el uso de memoria externa junto al impacto negativo de los nuevos costes de comunicación entre TPUs.

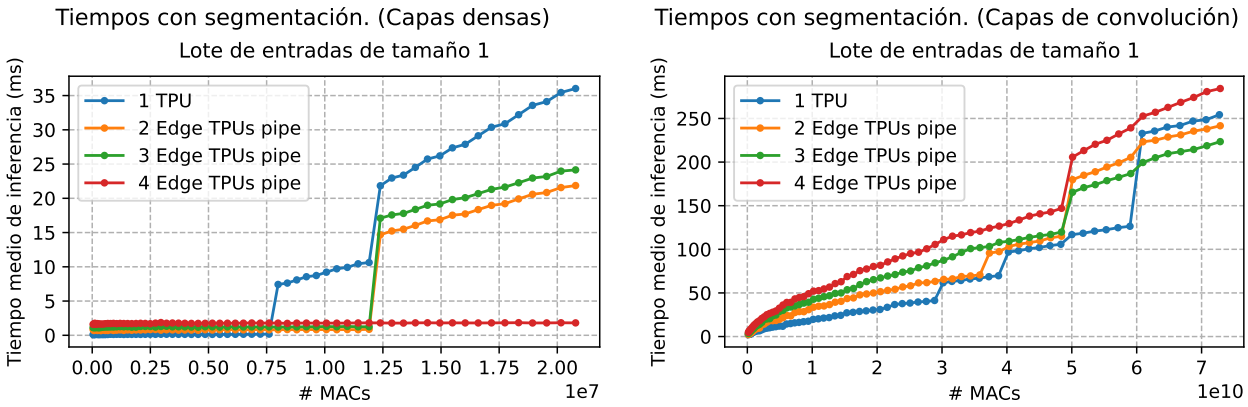


Figura 4.7: Tiempo medio de inferencia para 50 ejecuciones de cada modelo sobre un lote de entrada unitario.

En las capas densas observamos resultados mucho mejores al segmentar que en la capas de convolución. Esto era de esperar porque las comunicaciones por uso de memoria externa tienen un peso relativo mucho mayor en las capas densas. Tanto es así que, aunque no estemos explotando el paralelismo del *pipeline*, la segmentación ha resultado muy rentable cuando estos modelos no cabían en una TPU. No sucede así antes del primer salto, donde apreciamos muy ligeramente los sobrecostes de comunicación entre TPUs. Por otra parte, en las capas de convolución observamos tiempos de inferencia mayores al segmentar que sin hacerlo. No obstante, en los modelos más grandes ya empieza a resultar ligeramente mejor utilizar dos o tres segmentos.

La utilización de más segmentos retrasa la necesidad de alojar capas en el *host*, desplazando los saltos hacia modelos con cargas de trabajos mayores (hacia la derecha en nuestras gráficas). No obstante, a mayor es el número de segmentos mayor es el coste de comunicación. En este sentido, lo ideal es utilizar el mínimo número de segmentos para que el modelo use solamente memorias internas (se eviten todos los saltos). Por ejemplo, en el caso de los modelos de capas densas con $\sim 10^7$ operaciones MAC, basta segmentar en dos TPUs para alojar todo el modelo en memoria local y, además, es mejor que segmentar en tres o cuatro TPUs. Sin embargo, cuando el modelo ya se mueve en en las $\sim 1.5 \cdot 10^7$ operaciones, usar dos e incluso tres TPUs no es suficiente para almacenarlo entero y vale la pena usar cuatro TPUs para ello (el tiempo se reduce aproximadamente a la mitad).

Nos llama bastante la atención que se comporten igual la segmentaciones con dos y tres TPUs para capas densas. A priori, con tres TPUs debería producirse el salto por uso de memoria externa más tarde que con dos. Además, con un aprovechamiento ideal de la memoria interna, los dos saltos que se producían en nuestros modelos de capas densas deberían subsanarse con tres segmentos; sin embargo, hacen falta cuatro. De forma similar, los tres saltos que se producían en los modelos de convolución podrían solventarse con cuatro segmentos, pero en la práctica con esa cantidad se sigue produciendo un salto. Para analizar estos asuntos, nos fijamos en los datos de uso de memoria con varios segmentos.

En la tabla 4.3 apreciamos las consecuencias del reparto uniforme del número de capas entre los segmentos. Nuestros modelos de capas densas tienen una primera capa oculta pequeña ($64n$ pesos), tres capas ocultas grandes (n^2 pesos) y una capa de salida pequeña ($10n$ pesos). Al utilizar dos segmentos, el reparto de las cinco capas asigna las dos primeras al primer segmento (una de ellas grande) y las últimas tres al segundo segmento (dos de ellas grandes). Por eso, el uso de memoria en el segundo *chip* es aproximadamente el doble que en el del primero antes del salto (hay el doble de capas grandes) y se iguala justo después (cuando el segundo *chip* deja de alojar una capa grande). Con tres segmentos, se envía la primera capa a la primera TPU (capa pequeña), las dos siguientes a la segunda TPU (dos capas grandes) y las dos últimas a la tercera TPU (una capa grande y una pequeña). Por eso, se utiliza tan poca memoria interna en la primera TPU y la segmentación se comporta igual que en el caso de dos TPUs.

		2 TPUs pipe (MiB de memoria)				3 TPUs pipe (MiB de memoria)					
n	#MACs	TPU 1	TPU 2	Host 1	Host 2	TPU 1	TPU 2	TPU 3	Host 1	Host 2	Host 3
180	0.01e7	0.04	0.07	0	0	0.01	0.07	0.04	0	0	0
420	0.05e7	0.21	0.38	0	0	0.03	0.35	0.21	0	0	0
660	0.13e7	0.49	0.93	0	0	0.04	0.88	0.48	0	0	0
900	0.25e7	0.88	1.70	0	0	0.06	1.65	0.88	0	0	0
1140	0.40e7	1.32	2.57	0	0	0.07	2.5	1.32	0	0	0
1380	0.58e7	1.94	3.79	0	0	0.09	3.71	1.94	0	0	0
1620	0.80e7	2.67	5.24	0	0	0.10	5.14	2.67	0	0	0
1860	1.05e7	3.52	6.93	0	0	0.12	6.81	3.52	0	0	0
2100	1.33e7	4.36	4.36	0	4.23	0.13	4.23	4.36	0	4.23	0
2340	1.65e7	5.43	5.43	0	5.28	0.14	5.28	5.43	0	5.28	0
2580	2.01e7	6.62	6.95	0	6.46	0.16	6.48	6.61	0	6.46	0

Tabla 4.3: Uso de memoria de una muestra de modelos de capas densas con dos y tres segmentos.

En la tabla 4.4 observamos que se produce una situación similar para las capas de convolución con cuatro segmentos a la que vimos en las capas densas con tres. Nuestros modelos de convolución tienen una primera capa pequeña ($3f$ filtros) seguida de cuatro capas grandes (f^2 filtros). Con el reparto uniforme del número de capas que realiza el compilador, la primera TPU recibe la capa pequeña, la segunda TPU recibe una capa grande, la tercera TPU otra capa grande y la última TPU dos capas grandes. Por ello, la memoria interna de la primera TPU apenas se utiliza y el tamaño de memoria interna de la cuarta TPU acaba siendo insuficiente. Con un reparto que hubiese mandado las dos primeras capas a una TPU y una capa a las demás, todos los modelos hubieran cabido en memoria y no se habría producido ningún salto.

		4 TPUs pipe (MiB de memoria)							
f	#MACs	TPU 1	TPU 2	TPU 3	TPU 4	Host 1	Host 2	Host 3	Host 4
52	0.04e10	0.003	0.03	0.03	0.06	0	0	0	0
112	0.18e10	0.005	0.12	0.12	0.25	0	0	0	0
172	0.44e10	0.008	0.29	0.29	0.57	0	0	0	0
232	0.80e10	0.010	0.51	0.51	1.02	0	0	0	0
292	1.26e10	0.013	0.80	0.80	1.61	0	0	0	0
352	1.83e10	0.016	1.16	1.16	2.33	0	0	0	0
412	2.51e10	0.018	1.59	1.59	3.18	0	0	0	0
472	3.30e10	0.021	2.08	2.08	4.16	0	0	0	0
532	4.19e10	0.024	2.63	2.63	5.27	0	0	0	0
592	5.19e10	0.026	3.26	3.26	3.26	0	0	0	3.26
652	6.29e10	0.029	3.95	3.95	3.95	0	0	0	3.95

Tabla 4.4: Uso de memoria de una muestra de modelos de convolución con cuatro segmentos.

En definitiva, observamos que un reparto uniforme del número de capas no es buena estrategia cuando el uso de memoria es muy desequilibrado. A raíz de estos resultados, parece lógico pensar en un reparto de capas que trate de uniformizar el uso de memoria entre los segmentos. No obstante, esta solución no consideraría que, ante usos de memoria externa similares, es preferible aquel que distribuya la carga de trabajo de forma más equitativa. Lo ideal es que las fases de nuestro *pipeline* tengan una latencia similar ya que el rendimiento estará limitado por el segmento más lento. En este sentido, Google Coral ofrece una herramienta de perfilado que testea la latencia de los fragmentos para diferentes repartos y trata de minimizar la diferencia entre el más rápido y el más lento. En breve analizaremos un reparto basado en perfilado, pero antes vamos a probar la segmentación por defecto con un lote de entradas más grande.

Ejecución con un lote de entrada grande

Para aprovechar el potencial paralelo del *pipeline*, realizamos el mismo experimento con un lote de 50 entradas. En este caso, dividimos el tiempo de ejecución de todo el lote entre su tamaño para obtener el tiempo por inferencia. La figura 4.8 incluye gráficas con la aceleración respecto al uso de un lote unitario y respecto al uso de una sola TPU.

En ambas capas, la aceleración respecto a una entrada está lejos del ideal (en un *pipeline* con n TPUs podríamos esperar cerca de un $\times n$). El problema es que el reparto de la carga de trabajo es bastante desequilibrado y hay etapas mucho más lentas que otras ejerciendo de cuello de botella. Esto se une a los sobrecostes por comunicación entre TPUs para que, en los modelos de convolución y en los de capas densas que caben en el *chip* o siguen usando memoria externa, las aceleraciones respecto a una TPU sean paupérrimas. En estos casos, con tres o incluso cuatro TPUs la aceleración no llega a $\times 2$ y, por tanto, la segmentación no es para nada rentable. No obstante, sí que resulta muy útil en los modelos de capas densas cuando permite evitar por completo el uso de memoria *host* (porque su coste relativo es muy elevado). En estos casos, llegamos a ver aceleraciones de hasta $\times 36$ con cuatro TPUs, que justifican sobradamente la utilización de los dispositivos para repartir el modelo.

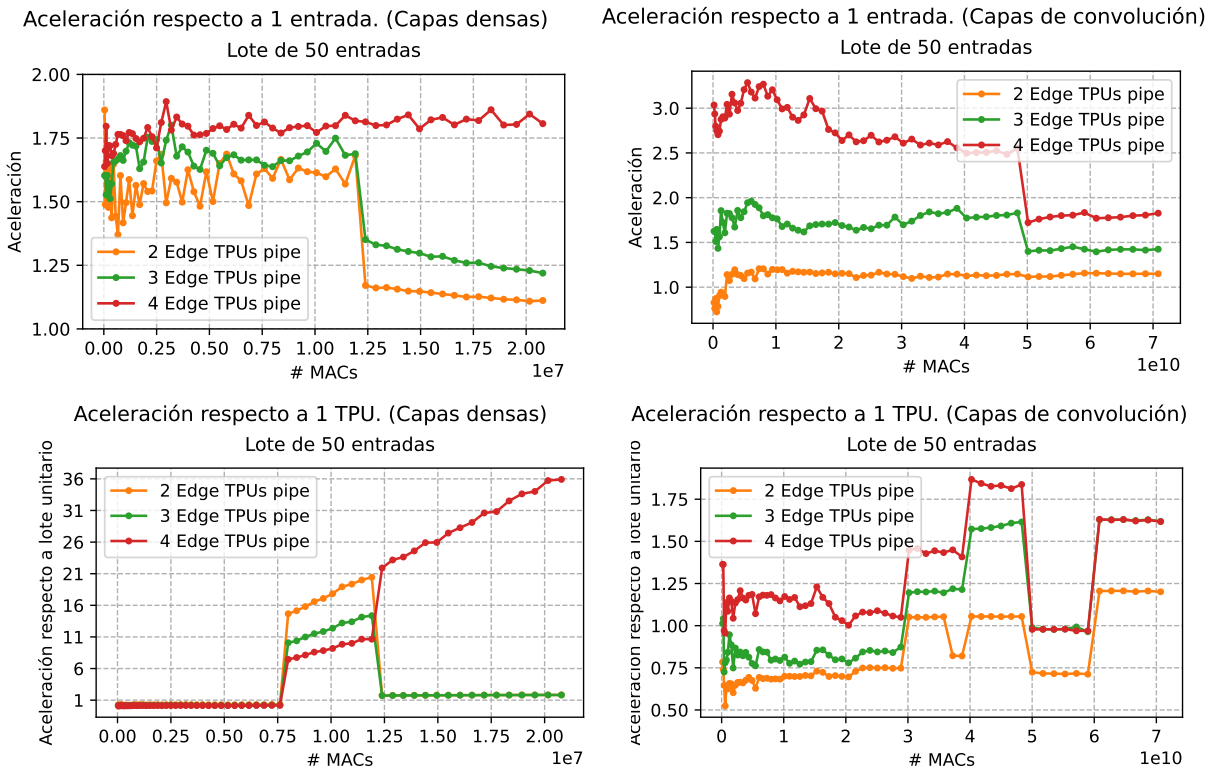


Figura 4.8: Tiempo medio por inferencia para 50 ejecuciones de cada modelo sobre un lote de 50 entradas. Gráficas con las aceleraciones respecto al caso con una sola entrada y una sola TPU.

4.2.3. Perfilado para optimizar la segmentación de modelos

Hemos visto un par de cuestiones a mejorar en la segmentación mediante una mejor distribución del modelo. Por una parte, queremos aprovechar mejor el espacio de memoria de las TPUs con un reparto de capas más equitativo en cuanto al tamaño. De esta forma, conseguiremos una mayor reducción del uso de memoria *host* para una misma cantidad de dispositivos. Por otra parte, queremos que la carga de trabajo se reparta de forma más equitativa para que las etapas del *pipeline* tengan latencias similares y su paralelismo sea más eficiente.

En nuestros modelos, ambos propósitos coinciden ya que todas las capas son del mismo tipo (todas son densas o todas son convolucionales). Al tener todas la misma intensidad aritmética, una mayor carga de trabajo equivale a un mayor uso de memoria. Sin embargo, es muy habitual que los modelos mezclen capas de distinto tipo y algunas ocupen mucho más que otras en relación al trabajo que realizan. En este sentido, para atajar ambos aspectos usando parámetros como la cantidad de memoria y el número de operaciones MAC, habría que plantear un problema de optimización multivariable difícil de resolver. En su lugar, se plantea automatizar la evaluación de diferentes repartos en las propias TPUs para escoger cuál utilizar.

Google Coral ofrece una herramienta de compilación que perfila la segmentación de diferentes repartos para la cantidad de segmentos que especifiquemos[11]. La herramienta recibe un parámetro con la diferencia máxima deseada entre el tiempo del segmento más rápido y el segmento más lento. Se prueban diferentes repartos hasta que uno cumple con la restricción y, por lo tanto, resulta elegido. En caso de que ningún reparto cumpla la restricción, hemos observado que la herramienta se queda con el último que haya probado. En este sentido, nos parece más lógico elegir el que haya funcionado mejor y, por lo tanto, nosotros lo haremos así. A continuación podemos ver un ejemplo de compilación basada en perfilado a través de la herramienta.

[11] `https://coral.ai/docs/edgetpu/compiler/#profiling-partitioner`

```
# Segmentación con perfilado para 3 segmentos con una diferencia máxima de 500 ns
# entre el más rápido y el más lento.
$ ./partition_with_profiling  --edgetpu_compiler_binary  /usr/bin/edgetpu_compiler
    --diff_threshold_ns 500 --model_path model_quant.tflite  --num_segments 3
```

Suponemos que Google Coral plantea el perfilado con un umbral objetivo para evitar que, si las restricciones del usuario son poco exigentes, la compilación se haga muy lenta por probar todas las opciones. No obstante, si el modelo tiene pocas capas, el número de posibles repartos es asequible para una exploración exhaustiva: hay $(l-1)!\,/\,((s-1)!\cdot(l-s)!)$ posibilidades[12], siendo l el número de capas y s el número de segmentos. En nuestros modelos de 5 capas hay solo 4 posibilidades con 2 segmentos, 6 posibilidades con 3 segmentos y 4 posibilidades con 4 segmentos. Por ello, decidimos implementar el perfilado exhaustivo que explora todas las opciones y se queda con la mejor. En lugar de usar la diferencia entre el segmento más rápido y más lento, es factible utilizar directamente nuestra función objetivo: el tiempo de una inferencia ejecutando como *pipeline* sobre un lote de entradas grande. A continuación se muestra un esbozo de nuestra implementación.

```
inf_times = []
# Exploramos todas las posibles separaciones
all_separators = itertools.combinations(range(1, num_layers), num_segments-1)
for separator in all_separators:
  # Añadimos los extremos para tener el inicio-1 y el fin de cada segmento
  split = [0] + separator + [num_layers]
  # Construimos los segmentos según los índices de partición
  segments = [Model(model.get_layer(f"layer{split[num_segment]+1}").input,
                    model.get_layer(f"layer{split[num_segment+1]}").output)
              for num_segment in range(num_segments)]
  # Cuantizamos y compilamos todos los segmentos
  for num_segment in range(num_segments):
    model_file_prefix = f"model_prefix_seg{num_segment}_of_{num_segments}"
    quantize(model_file_prefix, segments[num_segment], num_segment)
    compile_cmd = f"edgetpu_compiler {model_file_prefix}_quant.tflite"
    subprocess.Popen(compile_cmd.split()).communicate()
  # Medimos el tiempo de inferencia de la segmentación en el pipe
  inf_times.append(pipeline.execute(model_prefix, num_segments, steps, batch_size))
# Elegimos la segmentación con la inferencia más rápida
best_split = [0] + all_separators[inf_times.index(min(inf_times))] + [num_layers]
```

Como podemos ver en las gráficas de la figura 4.9, el tiempo de inferencia de muchos de nuestros modelos se reduce significativamente con la segmentación basada en perfilado. En el caso de las capas densas, no se incluye la segmentación con dos y cuatro TPUs porque no hay ninguna diferencia entre el reparto con perfilado y el que hace por defecto el compilador (los repartos por defecto ya eran los mejores). Sin embargo, en el caso con tres TPUs se consigue evitar el uso de memoria externa mediante el perfilado. Como vimos en la tabla 4.3, la segmentación por defecto infrautilizaba la memoria del primer *chip* guardando en ella los pesos de una sola capa pequeña. Mientras tanto, la segunda TPU almacenaba dos capas grandes y terminaba necesitando la memoria *host* para guardar una de ellas. En la tabla 4.5 apreciamos cómo la segmentación incluye una de esas capas grandes en la primera TPU, de forma que la segunda solo almacena la otra y no tiene que utilizar memoria externa. Como vemos, el perfilado selecciona una distribución de las capas muy equilibrada en cuanto al uso de memoria.

[12] Se trata de partir las l capas en s segmentos. Esto es equivalente a elegir $s-1$ separadores (para formar los s segmentos) entre las $l-1$ posiciones que hay entre capas. Es decir, las posibilidades son $\binom{l-1}{s-1} = \frac{(l-1)!}{(s-1)!\cdot(l-s)!}$.

En las capas de convolución también apreciamos un mejor aprovechamiento de espacio de memoria interna, que retrasa la necesidad de guardar capas en el *host* (de hecho, con cuatro TPUs los modelos generados no necesitan memoria externa). En las gráficas se aprecia cómo los saltos producidos cuando una capa comienza a guardarse en el *host* se desplazan hacia modelos más grandes (hacia la derecha en el eje de abcisas). En la tabla 4.6, vemos cómo el reparto para cuatro TPUs es bastante equilibrado en cuanto al uso de memoria, especialmente si lo comparamos con el reparto por defecto visto en la tabla 4.4.

Además de eso, en las capas de convolución observamos una reducción de tiempos debida a una mejor paralelización. Los tiempos de inferencia cuando no se ha producido todavía ningún salto son menores con perfilado que sin él. Esto es porque el reparto de la carga de trabajo es más equilibrado y, consecuentemente, la latencia de las etapas del *pipeline* también. La ejecución en el *pipeline* está limitada por la etapa más lenta y, por tanto, conviene un reparto del trabajo lo más equitativo posible. En la segmentación por defecto destacamos un importante desbalanceo de carga que el perfilado consigue mitigar.

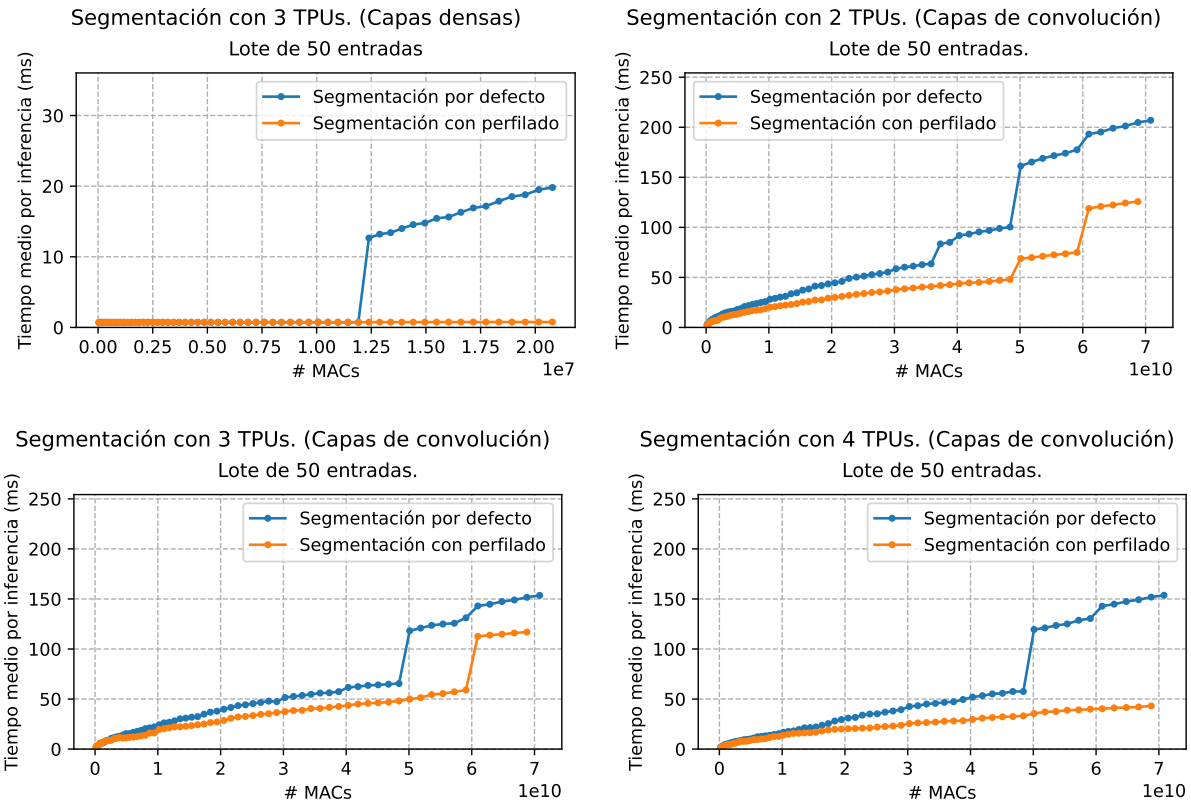


Figura 4.9: Tiempo de inferencia con segmentación por defecto vs perfilado para 50 ejecuciones. Para que se note la evolución hasta aquí, los límites del eje y son los que había sin segmentación.

		3 TPUs perfilado (MiB)		
n	#MACs	TPU 1	TPU 2	TPU 3
1140	0.40e7	1.32	1.18	1.32
1380	0.58e7	1.94	1.85	1.94
1620	0.80e7	2.67	2.57	2.67
1860	1.05e7	3.52	3.29	3.52
2100	1.33e7	4.36	4.23	4.36
2340	1.65e7	5.42	5.28	5.42
2580	2.01e7	6.62	6.46	6.62

Tabla 4.5: Uso de memorias internas en segmentación con perfilado de capas densas para tres TPUs. La memoria externa no se usa.

		4 TPUs perfilado (MiB)			
f	#MACs	TPU 1	TPU 2	TPU 3	TPU 4
292	1.26e10	0.82	0.80	0.80	0.80
352	1.83e10	1.18	1.16	1.16	1.16
412	2.51e10	1.61	1.59	1.59	1.59
472	3.30e10	2.10	2.08	2.08	2.08
532	4.19e10	2.66	2.63	2.63	2.63
592	5.19e10	3.28	3.26	3.26	3.26
652	6.29e10	4.00	3.97	3.97	3.97

Tabla 4.6: Uso de memorias internas en segmentación con perfilado de capas de convolución para cuatro TPUs. La memoria externa no se usa.

Para acabar, nos fijamos en la aceleración de la ejecución paralela tras perfilado respecto al caso inicial con una sola TPU (figura 4.10). Ya vimos que la segmentación por defecto era muy provechosa en los modelos de capas densas cuando evitaba utilizar memoria externa por completo (aceleraciones de varias decenas con apenas dos, tres o cuatro dispositivos). Con el perfilado, esta situación se extiende a todos los repartos para tres TPUs, que resultan la mejor opción para modelos que, sin segmentación, guardarían dos capas en el *host*. Aunque cuatro TPUs también evitan utilizar memoria externa, sus resultados son peores debido al innecesario sobrecoste de comunicación de una TPU extra. Lo mismo sucede en los modelos que solo guardarían una capa en el *host*, cuya máxima aceleración se obtiene con dos TPUs (con tres y cuatro TPUs tampoco se usa memoria *host* pero hay mayor coste de comunicación). En definitiva, lo óptimo es utilizar el mínimo número de TPUs que evite usar memoria externa.

En el caso de las capas de convolución, el perfilado mejora la segmentación por defecto hasta el punto de que sea ligeramente rentable segmentar los modelos más grandes (con cuatro TPUs se evita guardar tres capas en memoria externa y obtenemos cerca de un $\times 6$ respecto a una TPU). Esto sucede a pesar de que el uso de memoria *host* no tiene tanto impacto como en las capas densas y los costes de comunicación entre TPUs son relativamente altos en relación a las cargas de trabajo. No obstante, en los modelos que solo guardaría una o dos capas en el *host* con una TPU, no vale la pena invertir el *hardware* en segmentar (otro tipo de paralelización, como replicar el modelo y repartir el lote de entrada, podría ser más eficaz).

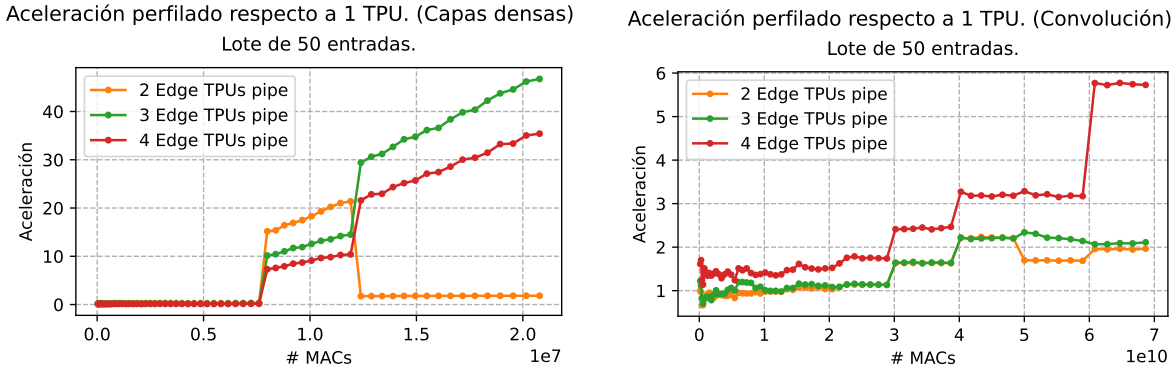


Figura 4.10: Aceleración de inferencias usando un *pipeline* basada en perfilado para un lote de 50 entradas respecto a la ejecución con una sola TPU.

Capítulo 5

Error por cuantización

Para ejecutar la inferencia de un modelo en el Edge TPU es necesario que el modelo esté cuantizado a enteros de 8 *bits*. La cuantización, explicada en la sección 3.1, es muy beneficiosa desde el punto de vista energético y de rendimiento, pero induce un error numérico asociado que se analiza en este capítulo. Concretamente, nos enfocamos en modelos de redes neuronales entrenados en el marco de procesos de *aprendizaje por refuerzo no consciente de cuantización*.

El aprendizaje por refuerzo es muy interesante para nosotros porque se utiliza en juegos y tareas de control que requieren inferencias constantes, optimizables mediante aceleradores como el Edge TPU. Dicha optimización depende bastante de la cuantización del modelo, que habitualmente se realiza tras el entrenamiento (es la técnica más flexible y la implica un menor sobrecoste en términos temporales). No obstante, esta opción introduce un mayor error asociado que un entrenamiento consciente de cuantización y, por ello, este capítulo es de interés.

5.1. Influencia del entorno, algoritmo y *bits* de precisión

Hemos encontrado estudios sobre el error por cuantización en aprendizaje por refuerzo [11] que usan un modelo muy entrenado para analizar la influencia del entorno de aplicación, el algoritmo utilizado y el número de *bits* de precisión[1]. En ellos, se trabaja con juegos Atari de la librería Gym (sección 3.3.4), que arrojan importantes diferencias en el error por cuantización debidas a la anchura de la distribución de pesos. El entorno Pong presenta distribuciones menos anchas que BeamRider o BreakOut y, por ello, induce menos error asociado para el mismo modelo y algoritmo. De forma análoga sucede con los algoritmos; se observa menos error por cuantización con PPO o A2C que con DQN porque los primeros generan distribuciones de pesos menos anchas. Este estudio también concluye que una representación entera de 8 *bits* es suficiente para que el error por cuantización sea muy pequeño en diferentes entornos con un algoritmo adecuado.

Entorno	Algoritmo	Error int8
BreakOut	DQN	63.55 %
BeamRider	DQN	22.05 %
Pong	DQN	0 %

Algoritmo	Entorno	Recompensa fp32	Error int8	Error fp16
DQN	BreakOut	214	63.55 %	-1.40 %
PPO	BreakOut	400	8.00 %	0.00 %
A2C	BreakOut	379	7.65 %	2.11 %

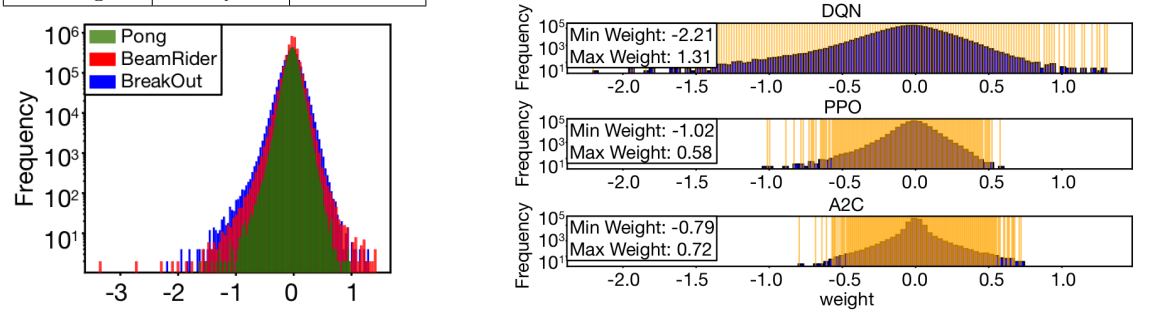


Figura 5.1: A la izquierda, error por cuantización `int8` para diferentes entornos e histogramas de las distribuciones de pesos. A la derecha, resultados análogos variando el algoritmo de entrenamiento y añadiendo cuantización `fp16`. Las líneas amarillas marcan los valores `fp32` asociados a los 256 números `int8`. Los datos e ilustraciones han sido extraídos de [11].

[1] El estudio mide el error por cuantización como el porcentaje medio de pérdida de recompensa al cuantizar de varias partidas.

Desde un punto de vista teórico, tiene sentido que la cuantización presente más error cuando la distribución de pesos es más amplia. Cuanto más amplia es la distribución, más amplios son los rangos que se truncan al mismo valor, pues existe más distancia entre valores `fp32` de números cuantizados consecutivos (líneas amarillas consecutivas de la figura 5.1). Si el número $q_1 \in$ `int8` está asociado al valor real r_1 y su consecutivo $q_2 = q_1 + 1$ está asociado al valor real r_2, todos los $r \in [r_1, r_1 + (r_2 - r_1)/2)$ se asocian al valor q_1 y todos los $r \in [r_1 + (r_2 - r_1)/2, r_2]$ se asocian a q_2. Así, la diferencias entre los valores de esos intervalos se pierde al cuantizar, lo que influye en el comportamiento de la red durante la inferencia. Esto resulta especialmente significativo cuando los rangos son amplios y los redondeos relativamente más grandes.

5.2. Evolución del error con el entrenamiento

Como el error por cuantización está muy influido por la distribución de pesos, surge interés por analizar cómo evoluciona con el entrenamiento. Los pesos se ajustan hacia valores que mejoran la recompensa del agente, lo que modifica su distribución e influye en la calidad de la cuantización. En esta sección analizamos si existe alguna relación entre la cantidad de entrenamiento y el error por cuantización. En este sentido, entrenamos una red neuronal con la librería RLlib, usando la configuración por defecto del algoritmo PPO con objetivo recortado, en el entorno de aprendizaje Pong-v0 de la biblioteca Gym. Cada cierto número de iteraciones guardamos un *checkpoint* del modelo, que después evaluamos cuantizado y sin cuantizar.

Como las observaciones de Pong-v0 son imágenes de la partida actual, lo más adecuado es utilizar una red de convolución. Concretamente, utilizamos la red que establece por defecto RLlib para este entorno, cuyas tres capas convolucionales se describen mediante ternas (`num_filtros`, `dim_filtros`, `stride`) como sigue: (16, (8, 8), 4), (32, (4, 4), 2), (256, (11, 11), 1). Esta red recibe imágenes $84 \times 84 \times 4$ tras el preprocesamiento que implementa RLlib (explicado en la sección 3.3.5). Después de estas capas, la red de política incluye una capa densa de 6 salidas con las probabilidades de tomar las 6 posibles acciones del juego. La figura 5.2 muestra la visualización de la red con la herramienta Netron[2], junto al número de parámetros entrenables desglosado por capas.

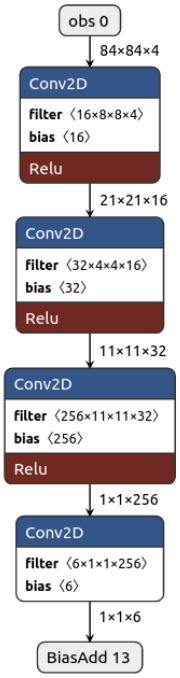
En el estudio expuesto en la sección previa [11], el error por cuantización se mide como el porcentaje medio de pérdida de recompensa para varias partidas del juego. El problema de esta medida es que, en cuanto el modelo cuantizado tome una acción distinta al modelo sin cuantizar, ambos empezarán a recibir observaciones diferentes y jugarán partidas distintas. Aunque es un enfoque realista (refleja lo que sucedería si sustituimos un modelo por otro), quizás no es el mejor para analizar el error propiamente dicho. Con un error pequeño, la decisión del agente puede cambiar en situaciones cruciales que le hagan ganar o perder, mientras que con un error más grande es posible que la decisiones del agente no cambien o lo hagan en situaciones irrelevantes para la recompensa. Esta métrica evalúa el impacto del error, que por supuesto es muy interesante, pero altamente dependiente del entorno y el contexto. Tendremos en cuenta esta medida, pero vemos la necesidad de evaluar el error propiamente dicho de una manera alternativa.

Una manera de medir el error puede ser la diferencia entre los vectores de salida con cuantización y sin ella para las mismas observaciones. En este sentido, decidimos utilizar el error relativo entre los vectores calculado como:

$$\text{Error}_{\text{cuant}} = \frac{\left\| \vec{\text{out}}_{\text{sin cuant}} - \vec{\text{out}}_{\text{cuant}} \right\|}{\left\| \vec{\text{out}}_{\text{sin cuant}} \right\|}. \tag{5.1}$$

Para que las observaciones cubran situaciones diferentes, también jugamos partidas con los modelos, pero tomando en ambos la acción indicada solamente por uno (concretamente, la del modelo sin cuantizar). Así, ambos se evalúan sobre las mismas entradas. Además, para que sean las mismas partidas entre modelos de iteraciones diferentes, fijamos las semillas de inicialización.

[2] `https://netron.app/`




```
Model: "model 1"
_________________________________________________________________________________________________
 Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
 observations (InputLayer)    [(None, 84, 84, 4)]       0           []

 conv_value_1 (Conv2D)        (None, 21, 21, 16)        4112        ['observations[0][0]']

 conv1 (Conv2D)               (None, 21, 21, 16)        4112        ['observations[0][0]']

 conv_value_2 (Conv2D)        (None, 11, 11, 32)        8224        ['conv_value_1[0][0]']

 conv2 (Conv2D)               (None, 11, 11, 32)        8224        ['conv1[0][0]']

 conv_value_3 (Conv2D)        (None, 1, 1, 256)         991488      ['conv_value_2[0][0]']

 conv3 (Conv2D)               (None, 1, 1, 256)         991488      ['conv2[0][0]']

 conv_value_out (Conv2D)      (None, 1, 1, 1)           257         ['conv_value_3[0][0]']

 conv_out (Conv2D)            (None, 1, 1, 6)           1542        ['conv3[0][0]']

 lambda (Lambda)              (None, 1)                 0           ['conv_value_out[0][0]']

=================================================================================================
Total params: 2,009,447
Trainable params: 2,009,447
Non-trainable params: 0
```

Figura 5.2: A la izquierda visualización con Netron de la red neuronal utilizada. A la derecha, resumen del modelo Keras con los parámetros entrenables de cada capa. Aunque se implementa con una capa "Conv2D", la capa de salida es una capa densa con 256 entradas y 6 salidas.

En nuestro análisis conviene incluir también alguna medida sobre cómo evoluciona la distribución de pesos con el entrenamiento. Al fin y al cabo, el cambio de distribución es el que influye directamente en el error por cuantización. En la sección previa, presentamos resultados que achacan el error a la anchura de pesos. No obstante, no es simplemente la anchura de la distribución lo que aumenta el error, sino el hecho de que, estando más lejos los extremos, los pesos no se dipersan en la misma proporción (ocupan posiciones relativas más próximas). Por ejemplo, una distribución el doble de amplia en que se duplique la distancia entre pesos consecutivos se cuantiza exactamente igual: el factor de escala se reduce a la mitad pero las distancias relativas son el doble y, por tanto, los mapeos son los mismos. La cuestión está cuando la separación de los extremos supera, de forma relativa, la dispersión entre algunos (es posible que pasen a mapearse al mismo valor cuantizado).

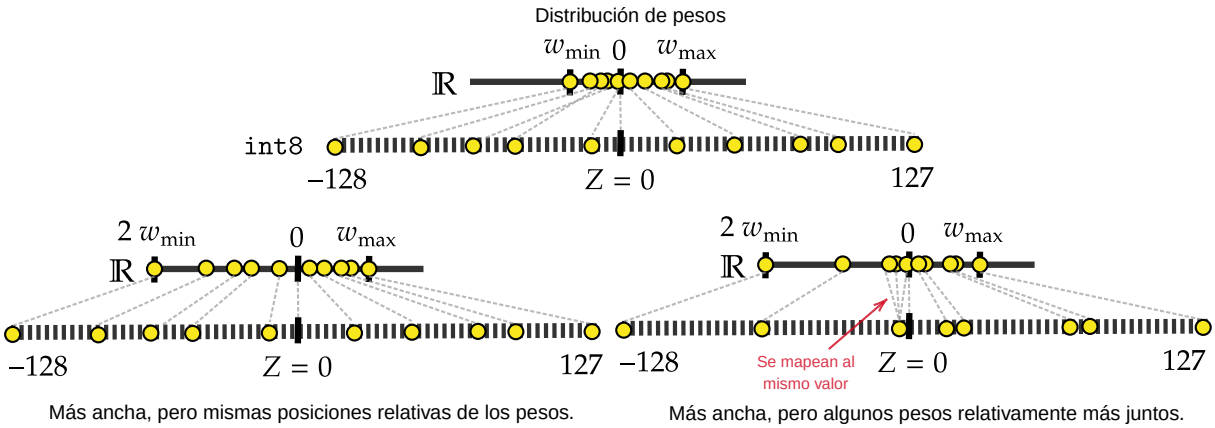


Figura 5.3: Cuantización el doble de ancha por el lado negativo en que los valores ocupan las mismas posiciones relativas o posiciones relativas más próximas en algunos casos. En el segundo caso, varios valores se mapean al mismo valor cuantizado. Se trata de una cuantización simétrica (el 0 real se mapea al 0 cuantizado) como la de TensorFlow Lite (ver sección 3.1).

Proponemos una medida que calcule el ancho de la distribución de pesos en relación a la dispersión entre sus valores. Como las distribuciones de pesos se asemejan a normales (ver histogramas de la figura 5.1), los valores se concentran en torno a la media, siendo una buena medida de la dispersión entre ellos la desviación típica (dispersión respecto a la media). En cuanto a la anchura, debemos tener en cuenta que la cuantización de TensorFlow Lite es simétrica (el 0 real se mapea al 0 cuantizado) y, por lo tanto, se dedican los mismos valores cuantizados a la parte positiva que a la parte negativa. Por este motivo, el error al cuantizar se debe principalmente al intervalo real más amplio entre $[w_{\text{mín}}, 0)$ y $(0, w_{\text{máx}}]$. En la figura 5.3 podemos ver los pesos que están muy juntos en el intervalo $[w_{\text{mín}}, 0)$ se mapean al mismo valor, mientras que pesos igual de juntos en términos absolutos del intervalo $(0, w_{\text{máx}}]$ se mapean a valores diferentes. Esto es porque $[w_{\text{mín}}, 0)$ es mucho más amplio que $(0, w_{\text{máx}}]$ y, en términos relativos, en un caso la diferencia es suficiente para distinguir los valores y en otro no. Tras estas observaciones, se justifica la siguiente expresión para calcular la anchura en relación a la dispersión:

$$\text{ancho-disp}(\vec{w}) = \frac{\text{máx}\left(|w_{\text{mín}}|, |w_{\text{máx}}|\right)}{\sigma\left(\vec{w}\right)}{}^{3}.$$

Finalmente, debemos pensar que los pesos del modelo se distribuyen en varios tensores multidimensionales y, como vimos en la sección 3.1, TensorFlow Lite cuantiza los pesos a nivel de eje del tensor (con parámetros S propios para cada índice de una de las dimensiones). Particularmente, hemos comprobado que el modelo a estudiar se cuantiza con factores S propios para la primera dimensión de cada tensor (los valores $w[i, :, ..., :]$ de cada tensor w tienen su propio S_{w_i}). En este sentido, realizamos la media ponderada de los valores de anchura-dispersión de las primeras dimensiones de los tensores obteniendo una medida global:

$$\text{ancho-disp}(W) = \sum_{\substack{w \in \text{tens}(W) \\ i \in \dim_0(w)}} \frac{\text{ancho-disp}\left(w[i, :, ..., :]\right) \cdot \text{num\_val}(w[i, :, ..., :])}{\sum_{\substack{w \in \text{tens}(W) \\ i \in \dim_0(w)}} \text{num\_val}(w[i, :, ..., :])}{}^{4}. \tag{5.2}$$

En los *scripts* `rel_error_quant.py` y `width-disp_weights.py` calculamos el error relativo entre los vectores de salida para cada *checkpoint* de entrenamiento y su medida de anchura-dispersión. Además, con el *script* `avg_rwd_dequant_vs_quant.py` calculamos también la recompensa media por partida que se obtiene con los modelos antes y después de cuantizar.

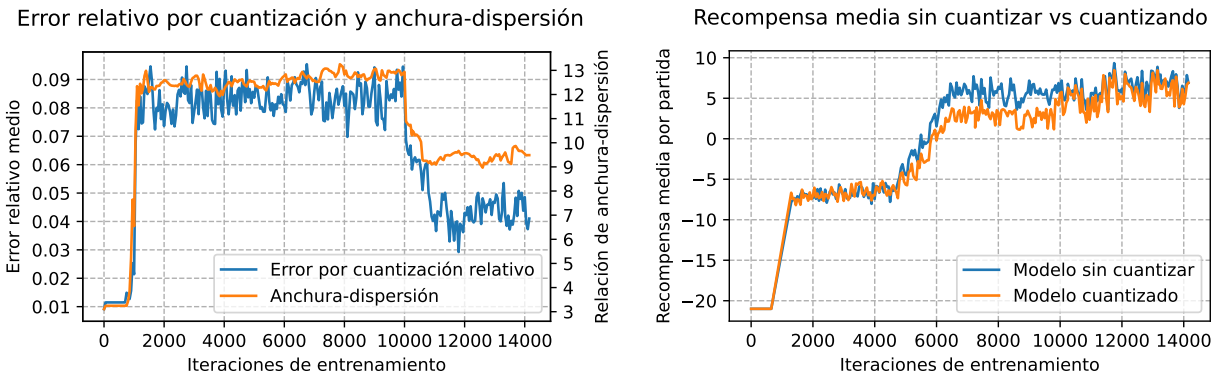


Figura 5.4: Error relativo medio por cuantización (ecuación 5.1) para 500 partidas fijas y anchura-dispersión (ecuación 5.2) cada 50 iteraciones de entrenamiento. Recompensa media del modelo cuantizado y sin cuantizar para esas partidas.

[3] Hacemos el valor absoluto de $w_{\text{máx}}$ por rigor (teóricamente podrían ser todos los pesos negativos), pero en la práctica siempre será un valor positivo.

[4] La función "num_val" proporciona un escalar con el número de valores que hay en un tensor, es decir, para un tensor n-dimensional t, $\text{num\_val}(t) = \prod_{i=0}^{n-1} |\dim_i(t)|$.

En la figura 5.4 observamos que el error por cuantización y la medida de anchura-dispersión evolucionan de manera muy similar. Como venimos explicando, un mayor valor de anchura-dispersión supone que haya más casos de pesos diferentes que se mapean al mismo valor cuantizado, lo que cambia bastante la lógica del modelo. En tal caso, la predicción cuantizada difiere más de la predicción sin cuantizar y el error es mayor. Apreciamos un error prácticamente nulo en las primeras iteraciones de entrenamiento porque la anchura de la distribución de pesos es muy pequeña en relación a su dispersión. Aproximadamente en 1 000 iteraciones, la medida de anchura-dispersión crece rápidamente y provoca un drástico aumento del error. Finalmente, en torno a las 10 000 iteraciones, el valor de anchura-dispersión vuelve a decaer de forma brusca, reduciendo el error por cuantización.

Si nos fijamos también en la recompensa media de los modelos, apreciamos que el aumento de error coincide con el primer escalón en que aumenta la recompensa. Parece que el primer escalón de aprendizaje se debe a una modificación en la distribución de pesos que aumenta la amplitud-dispersión, incrementando colateralmente el error por cuantización. No obstante, esto no parece tener un impacto crucial en el desempeño de los modelos, cuyas recompensas con cuantización son muy parecidas a las obtenidas sin cuantizar (en algunos casos incluso mejores).

El segundo escalón de aprendizaje no está relacionado con un cambio en los pesos que afecte a la anchura-dispersión, y el error por cuantización no se ve prácticamente alterado. Aunque la diferencia relativa entre las salidas cuantizadas y sin cuantizar no aumenta, en términos absolutos el error se nota más porque los modelos alcanzan puntuaciones más altas y las partidas son más largas. A partir de este momento, se aprecia una recompensa en los modelos cuantizados claramente peor que la recompensa sin cuantizar. Esta situación se prolonga hasta aproximadamante las 10 000 iteraciones, cuando la distribución de pesos comienza a ser menos ancha en relación a su dispersión. Entonces, el error por cuantización se reduce notablemente y la recompensa media que obtiene el modelo cuantizado vuelve a aproximarse a la del modelo sin cuantizar.

5.2.1. Análisis de diferentes arquitecturas de red

Finalmente, realizamos el mismo experimento con diferentes arquitecturas de red. No es posible evaluar tantos modelos como en el capítulo anterior ya que, a pesar de acelerar el entrenamiento en RLlib mediante GPUs, hacen falta muchas horas de ejecución para que el modelo tome buenas decisiones. Evaluaremos un par de redes de convolución que se obtienen alterando un parámetro muy concreto del modelo usado anteriormente. Así, este modelo servirá como referencia para comparar los resultados y extraer conclusiones. El cambio en una de las redes consiste en reducir el número de filtros de la segunda capa de 32 a 8, manteniendo sus dimensiones. El cambio en la otra red consiste en añadir dos capas de convolución con 16 filtros 8×8 y $stride = 1$.

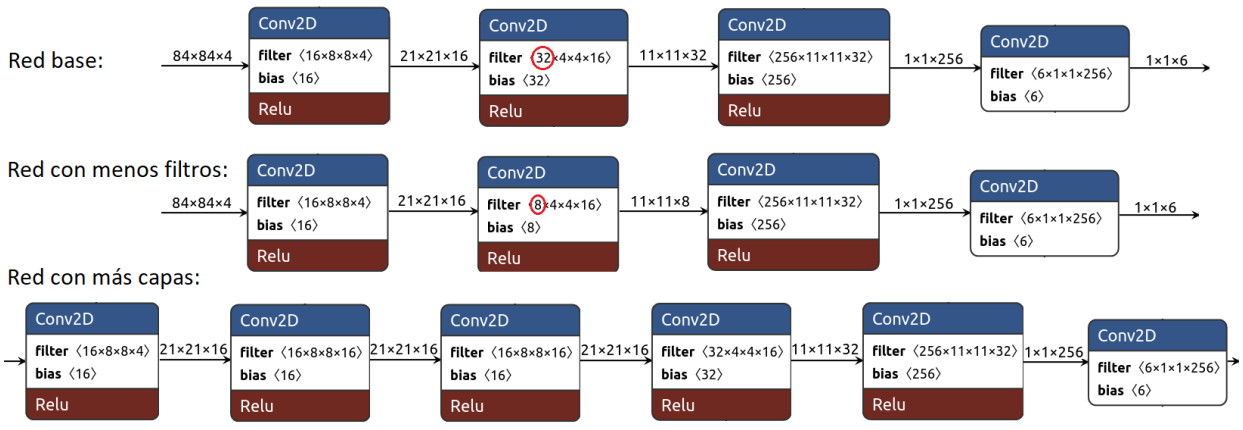


Figura 5.5: Arquitectura de la red de convolución base utilizada en la sección anterior y de las redes ligeramente modificadas que se utilizan en el próximo experimento.

La cantidad de pesos del modelo con menos filtros es aproximadamente la cuarta parte del número de pesos del modelo de referencia (510 263 frente a 2 009 447). Por su parte, la red con dos capas de convolución extra tiene aproximadamente el mismo número de pesos que el modelo de referencia (2 075 047 frente a 2 009 447). La Figura 5.6 muestra los resultados obtenidos.

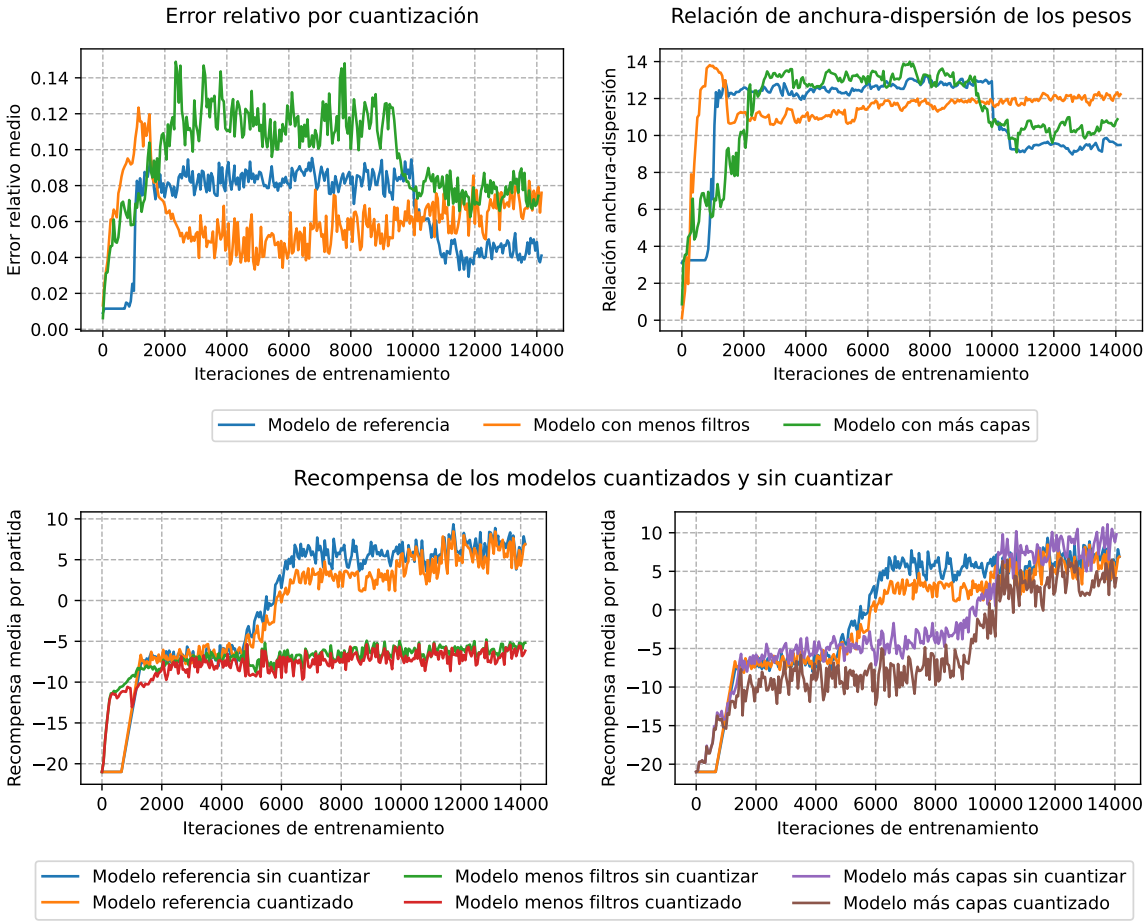


Figura 5.6: Resultados para las diferentes arquitecturas de red, en función del número de iteraciones de entrenamiento, del error relativo por cuantización, la relación de anchura-dispersión y la recompensa con/sin cuantización. El error y la recompensa se obtienen como la media de los resultados de 500 partidas fijas (las mismas para todos los modelos).

En todos los modelos parece haber dos fases del entrenamiento que afectan drásticamente a la anchura-dispersión de los pesos. Al principio del entrenamiento la anchura-dispersión crece de forma pronunciada y, tras cierta cantidad de iteraciones, su valor disminuye drásticamente hasta estabilizarse. Esto revierte directamente en el error, que presenta una evolución similar.

En el modelo con menos filtros, la reducción del error se produce aproximadamente tras 2 000 iteraciones de entrenamiento, pero en el modelo de referencia hacen falta 10 000. Pensamos que el hecho de tener menos pesos provoca que el entrenamiento del primer modelo evolucione más rápido. Por ello, el error por cuantización decrece antes y, durante bastantes iteraciones, es menor en este modelo que en el de referencia. Sin embargo, en este modelo no se observa el segundo escalón de aprendizaje que sí sucede en el modelo de referencia. Achacamos esto a la pérdida de expresividad que se deriva de una menor cantidad de pesos, lo que provoca que sus recompensas acaben siendo mucho peores (aprox. -6 frente a aprox. $+6$). En definitiva, aunque reducir el número de pesos pueda hacer que la cuantización sea más precisa durante bastantes iteraciones, no es rentable si se pierde tanta capacidad de aprendizaje como en este caso. Además, con el entrenamiento suficiente no sería interesante ni siquiera en términos de cuantización.

En el modelo con más capas, observamos que las variaciones del error se produce aproximadamente en las mismas iteraciones que en el modelo de referencia (suponemos que porque sus cantidades de pesos son similares). No obstante, apreciamos valores bastante mayores de anchura-dispersión y error, que se traducen en mayor diferencia entre la recompensa del modelo cuantizado y sin cuantizar (diferencia entre la curva marrón y morada vs diferencia entre la curva naranja y azul). La diferencia es tan importante que provoca que, a partir de las 10 000 iteraciones, la recompensa de dicho modelo cuantizado sea menor que la del modelo de referencia cuantizado (curva marrón vs naranja), a pesar de que su recompensa sin cuantizar es mayor que la del modelo de referencia sin cuantizar (curva morada vs azul).

Al estudiar el proceso de cuantización se explicó que se realiza una aproximación para calcular los productos escalares de cada una de las capas. El resultado de esta aproximación es la entrada de la siguiente capa que, en cierto modo, acumula el error cometido por las capas anteriores. Por ello, creemos que hacer el modelo más profundo provoca mayor error por cuantización. Para escalar en modelo en profundidad habría que valorar, al menos en términos de recompensa, si la capacidad de aprendizaje extra que se adquiere rentabiliza el aumento del error por cuantización. En el caso que hemos analizado, añadir capas no es rentable para la recompensa si se utiliza cuantización (aunque sin ella la recompensa mejore).

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este trabajo hemos analizado el rendimiento del Edge TPU para la inferencia de redes neuronales con capas densas y capas de convolución. Debido al elevado coste relativo de los accesos a memoria, en ambos casos hemos obtenido un rendimiento muy por debajo del ideal. Las ejecuciones son claramente *memory bound* con ambos tipos de capas, pero su rendimiento es mucho peor con las capas densas debido a que su intensidad aritmética es menor.

Por otra parte, hemos detectado un importante cuello de botella cuando los pesos del modelo no se pueden almacenar completamente en la memoria interna de la TPU. Esta memoria es de apenas 8 MiB y es bastante habitual que una red neuronal ocupe más tamaño (incluso en formato TensorFlow Lite cuantizado). En tal caso, una parte de los pesos se guarda en la memoria del *host* y, durante la ejecución, se envía al dispositivo mediante un bus PCIe. Estas transmisiones afectan dramáticamente al tiempo de inferencia, que crece de forma escalonada con el tamaño del modelo. El comportamiento escalonado se debe a que la unidad mínima de almacenamiento es la capa neuronal y, cuando la memoria del dispositivo es insuficiente, se almacenan capas completas en el *host* (en ningún caso se fraccionan los tensores de pesos de una capa).

El sobrecoste por utilizar memoria externa es especialmente notable en las capas densas ya que la relación entre la carga de trabajo y la cantidad de pesos es menor que en las convoluciones. Debido al sobrecoste de cargar pesos almacenados en el *host*, se han obtenido tiempos de inferencia de varias decenas de milisegundos con modelos de capas densas cuyos cómputos tardan menos de un milisegundo. En este tipo de modelos es imprescindible evitar el uso de memoria externa a la TPU. También se ha apreciado el impacto de utilizar memoria externa en las capas de convolución, pero con un peso relativo mucho menor. De hecho, a pesar de este sobrecoste, el Edge TPU ha mostrado mucho mejor rendimiento con este tipo de capas que una CPU de altas prestaciones.

Para mitigar el uso de memoria externa, los modelos se han segmentado y ejecutado en *pipelines* formados por varios Edge TPUs. La segmentación que ofrece por defecto el compilador de Edge TPU realiza un reparto uniforme del número de capas sin tener en cuenta la cantidad de pesos ni la carga de trabajo de cada una. No obstante, esta segmentación proporciona *speedups* de hasta $\times 36$ en modelos de capas densas con lotes de varias entradas usando 4 TPUs en lugar de una. Con las capas de convolución, el uso de memoria externa no tiene un peso relativo tan importante, y el *speedup* está muy por debajo de la cantidad de TPUs utilizadas (otro esquema de paralelización podría obtener mejores resultados).

Para optimizar la segmentación de los modelos se ha perfilado el tiempo de inferencia de las posibles particiones y se ha escogido la mejor. Si el modelo no tienen muchas capas es factible analizar todas las posibilidades. Con la segmentación basada en perfilado, los resultados han mejorado bastante, alcanzando casi un $\times 50$ en modelos de capas densas con 3 TPUs y hasta un $\times 6$ en modelos de capas de convolución con 4 TPUs. Lo idóneo es utilizar el mínimo número de TPUs que permitan almacenar el modelo sin usar memoria *host*, y con esta segmentación ha hecho falta una TPU menos que con la segmentación por defecto para ello (ya que obtiene una partición más equilibrada en cuanto al tamaño de memoria de los segmentos).

Por otra parte, la ejecución de un modelo en el Edge TPU requiere previamente cuantizarlo con enteros de 8 *bits*, lo que tiene un error asociado que hemos analizado en el marco del aprendizaje por refuerzo. Primero, se han presentado los resultados de un estudio que achaca mayor error por cuantización a mayor anchura en la distribución de pesos del modelo. Después, se han realizado experimentos propios usando el algoritmo de entrenamiento PPO en el entorno Pong-v0.

Se ha explicado que, cuando la anchura de la distribución de pesos es mayor en relación a la dispersión de sus valores, se mapean más pesos distintos al mismo entero. Cuando esto sucede, el error provocado por la cuantización aumenta. En este sentido, hemos diseñado métricas para la anchura de los pesos en relación a su dispersión y también para el error por cuantización. Hemos comparado estas métricas para un mismo modelo en función de la cantidad de iteraciones de entrenamiento, observando en ambas una evolución análoga. De esta forma, parece concluirse que la relación anchura-dispersión caracteriza el error.

En diversas arquitecturas de red se ha observado que el error crece bruscamente en las primeras iteraciones de entrenamiento, coincidiendo con el primer escalón en que la recompensa mejora. Después de cierto número de iteraciones, el algoritmo acaba ajustando una distribución de pesos con menor anchura-dispersión y, por lo tanto, el error disminuye. Esto tiene un impacto directo en las recompensas del modelo cuantizado que, con el suficiente entrenamiento, se acercan más a las del modelo sin cuantizar.

Finalmente, se ha observado que un modelo con menos pesos requiere menos iteraciones de entrenamiento para que disminuya el error por cuantización. No obstante, esta disminución de error anticipada puede no compensar la pérdida de recompensa debida a reducir la cantidad de pesos. Por otra parte, se ha observado un importante incremento del error por cuantización al aumentar las capas del modelo. Cada capa introduce cierto error al cuantizar que, en cierto modo, se acumula con el error de las capas anteriores. Por ello, un modelo más profundo sería peor para la cuantización. En el caso analizado, la pérdida de recompensa por el aumento del error ha superado a la mejora derivada de las capas adicionales.

6.2. Trabajo futuro

Debido a la importancia de almacenar íntegramente el modelo en la memoria del Edge TPU, proponemos explorar técnicas para reducir su tamaño. Estas técnicas se podrían complementar muy bien con la segmentación estudiada en este trabajo. En particular, sería interesante evaluar la poda (*pruning*) de pesos del modelo que sean poco significativos para su lógica. La librería TensorFlow ofrece diversas herramientas para realizar poda[1], por ejemplo al ejecutar los algoritmos de entrenamiento de su API Keras. Además, hay muchos otros esquemas de poda que sería interesante analizar. Esta técnica no solo reduce el tamaño del modelo sino también su tiempo de inferencia porque los cálculos con pesos podados no tiene que realizarse. Podría ser interesante analizar en qué medida se pueden reducir el tamaño y la carga de trabajo sin que la precisión del modelo se vea demasiado afectada.

Por otra parte, sería interesante evaluar el sobrecoste y la mejora que proporciona el entrenamiento consciente de cuantización frente a la cuantización post-entrenamiento. Esta comparativa podría incluir otros formatos de representación más precisos que `int8` como `fp16`, aunque entonces no serviría para Edge TPU (no obstante, hay otros aceleradores que puede trabajar con precisiones más altas). En el marco del aprendizaje por refuerzo, también sería interesante analizar cómo influye la definición de estados, acciones o el prepocesmiento de los datos en el error por cuantización. El objetivo sería obtener unas pautas para modelar un problema mediante aprendizaje por refuerzo minimizando el impacto de la cuantización en la calidad de la inferencia.

[1] `https://www.tensorflow.org/model_optimization/guide/pruning`

Chapter 1

Introduction

1.1. Motivation

Machine learning is a branch of artificial intelligence that is currently undergoing an evolution due, among other factors, to the rise of neural networks. Its aim is for a computer to automatically learn to generate suitable outputs for a certain domain of inputs, without explicitly programming the function between them. To do this, artificial neural networks are used, which try to emulate the structure and functioning of the human brain in a computer. These networks compute a function that is progressively adjusted by running a training algorithm to improve their outputs. After a period of *training*, the function is assumed to be sufficiently adjusted and the network is used in *inference* to predict the output of new inputs. Throughout the document, we will study the performance of different types of neural networks (multiperceptron and convolution), paying special attention to the inference process.

Neural networks are universal approximators that theoretically can learn any continuous function (Universal Approximation Theorem [1]), but in practice are limited by finite computational resources. Their recent heyday is due, among other things, to the emergence of new architectures and algorithms that have greatly improved their performance and broadened their potential applications. We will see that the computations required by neural networks support a high degree of parallelism, which can be exploited by GPUs, but are much more optimised with processors specifically designed for tensor computations such as TPUs (Tensor Processing Units). In this work we will study the architecture of a TPU, analysing its main advantages over other processing schemes.

The growing demand for IoT (Internet of Things[1]) applications using neural networks has given a significant role in this field to ad-hoc processors for their computations. Due to their specificity, these processors achieve very good performance with low power consumption, which is ideal for this type of applications. Moreover, in many cases very low latencies are required for inference, which motivates their confluence with edge computing[2]. In this sense, Google sells various modules that integrate the Edge TPU, a specific processor for inference with neural networks oriented towards this type of computation. This is a low-performance model with very low power consumption and a very affordable price. In this work we will analyse this processor in detail, carrying out a parametric evaluation of the inference time according to different characteristics of the networks used. In this study, we will detect an important bottleneck in the internal memory size, and we evaluate the segmentation of the model among several TPUs as a possible solution.

One of the main reasons the Edge TPU achieves good performance with very low power consumption is that it operates in integer arithmetic and performs the most expensive operations using 8 bits of precision. To ensure convergence of the algorithms, the training of the networks must be done with more accurate floating point values (`fp32` or `fp16`); however, we will see that the characteristics of the neural networks allow inference to be performed with lower precision integer operations without altering their behaviour too much.

[1] The internet of things is the collection of devices with IP addresses that are connected to each other via a network (usually wireless). This field includes 'smart' objects that increasingly use neural networks for a specific task.

[2] Edge computing is about processing data close to where it is produced rather than sending it to the cloud for processing there. This reduces the latency of sending and enhances data protection.

In this sense, a quantization process is performed to convert the values of the floating-point trained network to 8 *bits* integers. We will explain this process in the memory and analyse the impact it has on the quality of the models. To do this, it is necessary to establish an application and a concrete learning method with which to train the model. [3]. We will use reinforcement learning, which is an appropriate paradigm for complex problems where large neural networks are needed that make great use of TPUs. In fact, TPUs are known for helping the AlphaGo computer system, which consists of huge reinforcement-intensively trained networks, to win the world champion of the game Go for the first time. We will use the PPO training algorithm to automatically learn to play in a virtual ping-pong environment. Under these conditions, we will study the quantization error depending on the number of training iterations in different network architectures.

1.2. Objectives

On the one hand, this work aims to draw general conclusions on factors important for the performance of the Edge TPU. In particular, we will try to detect bottlenecks and, where possible, propose and evaluate solutions for them. Beyond the conclusions to be drawn from a theoretical analysis, we will perform various experiments with synthetic models to assess their impact. This analysis will not be limited to a single device, but will consider the possibility of using several TPUs together.

On the other hand, we intend to analyse the error caused by the quantization necessary to execute in this TPU. Specifically, we will try to analyse this error in the framework of reinforcement learning, taking advantage of and extending conclusions from a related study. We will choose a good error metric and study its evolution with training progress for different network architectures. In this analysis we will also consider the weight distribution of the model, which, according to the literature, is closely related to the error.

These two main objectives depend on more specific objectives, which are listed below:

- To understand in depth the theoretical foundations of the work. In particular, the theory on TPUs and on quantization of neural networks is especially relevant.
- Systematically create neural models with different network architectures, and perform the necessary transformations to run their inference in Edge TPU. This includes converting the model to the simplified TensorFlow Lite format, quantifying it and compiling it for Edge TPU.
- Evaluate different types of neural networks. Specifically, we will use multiperceptron networks and convolutional networks.
- Train neural models by reinforcement learning. In particular, we will use the PPO training algorithm, which will run efficiently with the RLlib library taking advantage of multicore parallelism and GPUs.

1.3. Related work

Since their public launch in 2016, TPUs have received a lot of attention from the research community. Most of the studies on TPUs analyse the Cloud versions. The designers of the first Cloud TPU present architectural details of the processor in [2]. Meanwhile, [3] and [4] provide an in-depth study of the features and differences between CPUs, GPUs and Cloud TPUs.

[3] To evaluate the performance of a model it is not necessary to train it because it does not matter what the function computes, but when evaluating its quality it is necessary

Edge TPU is a 2019 release that has been so far evaluated for specific models and/or tasks. For example, in [5] image processing tests are performed with this device, in [6] and [7] it is evaluated for tasks related to object detection and in [8] it is applied to the classification of spectrograms of livestock information. The only study with synthetic models that we have found is a recent article by Google [9] members, who do not use the commercial version (they use more powerful versions with different frequencies, memories or buses). In addition, no studies have been found that use multiple Edge TPUs together.

On the other hand, there are many studies that analyse the error associated with the quantization of neural networks, but practically all of them use supervised learning. We highlight [10], which founds and applies the same quantization scheme that will be used in this work (the one implemented by TensorFlow Lite). In supervised learning, the outputs of the network are independent of each other, but in reinforcement learning, the output determines what the next input will be. This suggests that the impact of quantization may be different for reinforcement learning. However, the only study we have found on quantization of trained models with reinforcement learning is [11]. This paper summarises and extends the results of that article.

1.4. Methodology and work plan

This work started about a year ago, establishing with the directors the main objectives. From the beginning, the directors have held weekly meetings with the student to guide him in the development of the work. The first 6 months were devoted to studying different theoretical foundations on TPUs, quantization and reinforcement learning. During this period, the first training tests with the RLlib library and the first inferences in Edge TPU were also carried out. During the following 4 months, the experiments to evaluate the inference time in the Edge TPU were developed, and the last 2 months were dedicated to analyse the quantization error. During these last 6 months, the working memory was also written.

To develop the experiments, several Python scripts have been developed and can be found in the following GitHub repository: `https://github.com/Jorgitou98/EdgeTPU-QuantRL-Evaluation`. This document will refer to some of the files in this repository with links that can be clicked to access them. The repository also contains the results of the experiments and two notebooks in which the document plots are generated using the Matplotlib library[4]. Almost all other figures have been produced by the student using the Matcha tool[5]. The source of any other figure is appropriately referenced at the bottom of the figure.

1.5. Document structure

Rest of the document is structured in five chapters as follows:

- **Chapter 2** explains the theoretical principles of a TPU. In particular, its architecture is studied in depth and its main advantages for neural network computations compared to a CPU or a GPU are explained. Finally, the specific characteristics of the Edge TPU version are introduced.
- **Chapter 3** covers other important fundamentals: quantization, convolutional networks and reinforcement learning. In relation to reinforcement learning, the PPO algorithm is explained and the Gym and RLlib libraries are introduced in a practical way. Finally, the transformations necessary to perform model inference in Edge TPU are presented.

[4] `https://matplotlib.org/`

[5] `https://www.mathcha.io/editor`

- **Chapter 4** presents the results of several experiments on inference time in Edge TPU. To begin with, a single TPU is evaluated in terms of the number of model operations and its memory usage. Then, the segmented execution of models using several TPUs is analysed as a solution to a detected bottleneck. This part includes experiments with different input batches and profiling-based segmentation.
- **Chapter 5** studies the quantization error for models trained with reinforcement learning. First, the results of [11] are presented, which analyse this error by changing the application environment, the training algorithm and the number of precision bits. These results are then used to analyse, by means of our own experiments, how the error evolves with the progress of training in different network architectures.
- **Chapter 6** presents the conclusions with a brief review of the results obtained and makes some proposals for future work.

Chapter 6

Conclusions and future work

6.1. Conclusions

In this work we have analysed the performance of the Edge TPU for the inference of neural networks with dense layers and convolutional layers. Due to the high relative cost of memory accesses, in both cases we have obtained a performance far below the ideal. The executions are clearly memory bound with both types of layers, but the performance is much worse with dense layers due to their lower arithmetic intensity.

We have detected a major bottleneck when the model weights cannot be completely stored in the internal memory of the TPU. This memory features 8 MiB of internal memory, and it is quite common for a neural network to take up more size (even in quantized TensorFlow Lite format). In such a case, a portion of the weights is stored in the host memory and, during execution, is sent to the device via a PCIe bus. These transmissions dramatically affect the inference time, which grows stepwise with the size of the model. The stepwise behaviour is due to the fact that the minimum storage unit is the neural layer, and when the device memory is insufficient, complete layers are stored in the host (the weight tensors of a layer are never split).

The overhead of using external memory is especially noticeable in dense layers as the ratio between the workload and the number of weights is lower than those observed in convolutional layers. Due to the overhead of loading weights stored in the host, inference times of several tens of milliseconds have been obtained with models whose computations take less than a millisecond. For such models, it is essential to avoid using memory external to the TPU. The impact of using external memory in convolutional layers has also been observed, but with a much smaller relative weight. In fact, despite this overhead, the Edge TPU has shown much better performance with these layers than a high-performance CPU.

To mitigate the use of external memory, models have been segmented and executed in pipelines consisting of several Edge TPUs. The default segmentation provided by the Edge TPU compiler performs an even distribution of the number of layers without taking into account the number of weights and the workload of each layer. However, this segmentation provides speedups of up to 36× on dense layers with multi-input batches using 4 TPUs instead of one. With convolutional layers, the use of external memory does not have such an important relative weight, and the speedup is well below the amount of TPUs used (another parallelisation scheme would obtain better results).

To optimise the segmentation of the models, the inference time of the possible partitions has been profiled and the best one has been chosen. If the model does not have many layers, it is feasible to analyse all possibilities. With the profiling-based segmentation, the results have improved a lot, reaching almost 50× with dense layers using 3 TPUs and up to 6× with convolutional layers using 4 TPUs. Ideally, it is best to use the minimum number of TPUs to store the model without using host memory, and with this segmentation it took one less TPU than with the default segmentation to do this (as it gives a more balanced partition in terms of the memory size of the segments).

On the other hand, running a model on the TPU Edge requires quantization with 8-bit integers, which has an associated error that we have analysed with a focus on reinforcement learning. First, we have presented the results of a study that attributes greater quantization error to

greater width in the distribution of model weights. Then, we conducted our own experiments using the PPO training algorithm in the Pong-v0 environment.

It has been explained that, when the width of the distribution of weights is larger relative to the spread of their values, more different weights map to the same integer. When this happens, the error caused by quantization increases. In this sense, we have designed metrics for the width of the weights in relation to their dispersion and also for the quantization error. We have compared these metrics for the same model as a function of the number of training iterations, observing an analogous evolution in both. Thus, it seems to be concluded that the width-dispersion relationship characterises the error.

In several network architectures it has been observed that the error grows sharply in the first training iterations, coinciding with the first step in which the reward improves. After a certain number of iterations, the algorithm eventually adjusts to a distribution of weights with lower dispersion-width and, therefore, the error decreases. This has a direct impact on the quantized model's rewards, which, with sufficient training, become closer to those of the unquantized model.

Finally, it has been observed that a model with fewer weights requires fewer training iterations for the quantization error to decrease. However, this anticipated decrease in error may not make up for the loss of reward due to reducing the number of weights. On the other hand, a significant increase in quantization error has been observed as the number of layers in the model increases. Each layer introduces some quantization error which, in a sense, accumulates with the error of the previous layers. Therefore, a deeper model would be worse for quantization. In the case analysed, the loss of reward from the increased error has outweighed the improvement from the additional layers.

6.2. Future work

Due to the importance of storing the entire model in the memory of the Edge TPU, we propose to explore techniques to reduce its size. These techniques could complement very well with the segmentation studied in this work. In particular, it would be interesting to evaluate the pruning of model weights that are insignificant for its logic. The TensorFlow library offers several tools to perform pruning[1], for example when running the training algorithms of its Keras API. In addition, there are many other pruning schemes that would be interesting to analyse. This technique not only reduces the size of the model but also its inference time because the calculations with pruned weights do not have to be performed. It might be interesting to analyse how the size and the workload can be reduced without affecting the model accuracy too much.

On the other hand, it would be interesting to evaluate the overhead and improvement provided by quantization-aware training versus post-training quantization. This comparison could include other representation formats that are more accurate than `int8` such as `fp16`, although then it would not be useful for Edge TPU (however, there are other accelerators that can work with higher accuracies). Focusing on reinforcement learning, it would also be interesting to analyse how the definition of states, actions or the pre-fitting of data influences the quantization error. The objective would be to obtain guidelines for modelling a problem using reinforcement learning while minimising the impact of quantization on the inference quality.

[1] `https://www.tensorflow.org/model_optimization/guide/pruning`

Bibliografía

[1] Kurt Hornik, Maxwell Stinchcombe y Halbert White. «Multilayer feedforward networks are universal approximators». En: *Neural Networks* 2.5 (1989), págs. 359-366. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8.

[2] Norman P. Jouppi, Cliff Young, Nishant Patil *et al.* «In-datacenter performance analysis of a tensor processing unit». En: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, págs. 1-12. DOI: 10.1145/3079856.3080246.

[3] Goran S. Nikolić, Bojan R. Dimitrijević, Tatjana R. Nikolić *et al.* «A Survey of Three Types of Processing Units: CPU, GPU and TPU». En: *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*. 2022, págs. 1-6. DOI: 10.1109/ICEST55168.2022.9828625.

[4] P Raj y Ch Sekhar. «Comparative Study on CPU, GPU and TPU». En: *International Journal of Computer Science and Information Technology for Education* 5 (mayo de 2020), págs. 31-38. DOI: 10.21742/IJCSITE.2020.5.1.04.

[5] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou *et al.* «Machine Learning at the Network Edge: A Survey». En: *ACM Computing Surveys* 54.8 (oct. de 2021), págs. 1-37. DOI: 10.1145/3469029.

[6] Pilsung Kang y Athip Somtham. «An Evaluation of Modern Accelerator-Based Edge Devices for Object Detection Applications». En: *Mathematics* 10 (nov. de 2022), pág. 4299. DOI: 10.3390/math10224299.

[7] Ahmad Ammar Asyraaf Jainuddin, Yew Cheong Hou, Mohd Zafri Baharuddin *et al.* «Performance Analysis of Deep Neural Networks for Object Classification with Edge TPU». En: *2020 8th International Conference on Information Technology and Multimedia (ICIMU)*. 2020, págs. 323-328. DOI: 10.1109/ICIMU49871.2020.9243367.

[8] Seyedehfaezeh Hosseininoorbin, Siamak Layeghy, Brano Kusy *et al.* «Scaling Spectrogram Data Representation for Deep Learning on Edge TPU». En: *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. 2021, págs. 572-578. DOI: 10.1109/PerComWorkshops51409.2021.9431041.

[9] Kiran Seshadri, Berkin Akin, James Laudon *et al.* «An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks». En: *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 2022, págs. 79-91. DOI: 10.1109/IISWC55918.2022.00017.

[10] Benoit Jacob, Skirmantas Kligys, Bo Chen *et al. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. 2017. DOI: 10.48550/ARXIV.1712.05877. URL: https://arxiv.org/abs/1712.05877.

[11] Srivatsan Krishnan, Sharad Chitlangia, Maximilian Lam *et al. Quantized Reinforcement Learning (QUARL)*. 2019. DOI: 10.48550/ARXIV.1910.01055. URL: https://arxiv.org/abs/1910.01055v3.

[12] John Shalf. «The future of computing beyond Moore's Law». En: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378 (2020). DOI: 10.1098/rsta.2019.0061.

[13] Norman P. Jouppi. «Google supercharges machine learning tasks with TPU custom chip». En: *Google Blogs* (2016). URL: https://cloud.google.com/blog/products/ai-machine-learning/google-supercharges-machine-learning-tasks-with-custom-chip.

[14] Q-Engineering. *Google Coral Edge TPU explained in depth*. URL: https://qengineering.eu/google-corals-tpu-explained.html.

[15] Kaz Sato, Cliff Young y David Patterson. «An in-depth look at Google's first Tensor Processing Unit (TPU)». En: *Google Cloud Blog* (2017). URL: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[16] «Cloud TPU performance guide». En: *Cloud TPU documentation*. URL: https://cloud.google.com/tpu/docs/performance-guide.

[17] «The pros and cons of lower precision». En: *ARM developer documentation*. URL: https://developer.arm.com/documentation/102502/0100/The-pros-and-cons-of-lower-precision.

[18] Christopher De Sa. «Lecture 23: Low-Precision Machine Learning». En: *Cornell University - Principles of Large-Scale Machine Learning*. 2019. URL: https://www.cs.cornell.edu/courses/cs4787/2019sp/notes/lecture23.pdf.

[19] Pete Warden. «Why are Eight Bits Enough for Deep Neural Networks?» En: *Pete Warden's Blog* (2015). URL: https://petewarden.com/2015/05/23/why-are-eight-bits-enough-for-deep-neural-networks/.

[20] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan *et al. Deep Learning with Limited Numerical Precision*. 2015. DOI: 10.48550/ARXIV.1502.02551. URL: https://arxiv.org/abs/1502.02551.

[21] Jiong Gong, Haihao Shen, Guoming Zhang *et al. Highly Efficient 8-bit Low Precision Inference of Convolutional Neural Networks with IntelCaffe*. 2018. DOI: 10.48550/ARXIV.1805.08691. URL: https://arxiv.org/abs/1805.08691.

[22] Charbel Sakr, Naigang Wang, Chia-Yu Chen *et al. Accumulation Bit-Width Scaling For Ultra-Low Precision Training Of Deep Networks*. 2019. DOI: 10.48550/ARXIV.1901.06588. URL: https://arxiv.org/abs/1901.06588.

[23] Paulius Micikevicius, Sharan Narang, Jonah Alben *et al. Mixed Precision Training*. 2017. DOI: 10.48550/ARXIV.1710.03740. URL: https://arxiv.org/abs/1710.03740.

[24] Shibo Wang y Pankaj Kanwar. «BFloat16: The secret to high performance on Cloud TPUs». En: *Google Cloud Blog* (2019). URL: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

[25] Suyog Gupta y Mingxing Tan. «EfficientNet-EdgeTPU: Creating Accelerator-Optimized Neural Networks with AutoML». En: *Google AI Blog* (2019). URL: https://ai.googleblog.com/2019/08/efficientnet-edgetpu-creating.html?m=1.

[26] «Quantization Aware Training with TensorFlow Model Optimization Toolkit - Performance with Accuracy». En: *TensorFlow Blog* (2020). URL: https://blog.tensorflow.org/2020/04/quantization-aware-training-with-tensorflow-model-optimization-toolkit.html.

[27] Sumit Saha. «A comprehensive Guide to Convolutional Neural Networks». En: *Towards Data Science* (2018). URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[28] Mayank Mishra. «Convolutional Neural Networks, Explained». En: *Towards Data Science* (2020). URL: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939.

[29] Mehryar Mohri, Afshin Rostamizadeh y Ameet Talwalkar. «Reinforcement Learning». En: *Foundations of Machine Learning*. 2nd. The MIT Press, 2018, págs. 379-405. ISBN: 0262039400. URL: https://cs.nyu.edu/~mohri/mlbook/.

[30] Aske Plaat. *Deep Reinforcement Learning*. Singapore: Springer Nature Singapore, 2022, págs. 1-24. ISBN: 978-981-19-0638-1. DOI: 10.1007/978-981-19-0638-1_1.

[31] John Schulman, Filip Wolski, Prafulla Dhariwal *et al. Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

[32] Jonathan Hui. «RL — Proximal Policy Optimization (PPO) Explained». En: *Medium* (2018). URL: https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12.

[33] Greg Brockman, Vicki Cheung, Ludwig Pettersson *et al. OpenAI Gym*. 2016. DOI: 10.48550/ARXIV.1606.01540. URL: https://arxiv.org/abs/1606.01540.

[34] Javier Guzmán Muñoz. «Evaluación de rendimiento de arquitecturas paralelas y de propósito específico para el aprendizaje por refuerzo en juegos». Trabajo de Fin de Grado en Ingeniería Informática. 2021. URL: https://eprints.ucm.es/id/eprint/67270/.