Optimización de redes neuronales en Apple Silicon (M1, M2, M3) y Google Tensor con variedades diferenciables, SVD y optimizaciones de bajo nivel

La reciente generación de **chips Apple Silicon (M1, M2, M3)** y los **SoC Google Tensor** en dispositivos Pixel ha impulsado la **IA móvil**, permitiendo correr redes neuronales avanzadas directamente en teléfonos y portátiles. Para aprovechar plenamente este hardware es crucial **optimizar los modelos de IA**, reduciendo su complejidad matemática y afinando su implementación a nivel de máquina. En este artículo exploramos técnicas de optimización que combinan enfoques matemáticos (variedades diferenciables, SVD) con optimizaciones de bajo nivel (pruning, cuantización, destilación, código ensamblador y GPU) para exprimir el rendimiento del **Neural Engine** de Apple y la **TPU** de Google Tensor sin sacrificar precisión ni eficiencia energética. Presentamos los fundamentos teóricos, comparativa de métodos, detalles de implementación en C++/ARM64/Metal, **benchmarks** en CPU, GPU y aceleradores dedicados, y mejores prácticas para desarrolladores móviles.

Variedades diferenciables y reducción de dimensionalidad

En matemáticas, una *variedad diferenciable* es un espacio que localmente se comporta como un espacio euclidiano (plano) y en el cual se pueden aplicar operaciones de cálculo diferencial. En el contexto de *machine learning*, surge la llamada **hipótesis de la variedad**: muchos datos de alta dimensión (imágenes, audio, texto codificado, etc.) en realidad residen en una **variedad latente de baja dimensión** dentro del espacio original (Manifold hypothesis - Wikipedia). Es decir, aunque una imagen pueda tener millones de píxeles (dimensionalidad alta), las variaciones naturales (iluminación, pose, etc.) ocupan una subestructura de mucho menor dimensión. Esta idea explica por qué es posible representar datos complejos con relativamente pocos parámetros clave (las *coordenadas* en la variedad latente) (Manifold hypothesis - Wikipedia).

¿Cómo aprovecha esto la reducción de dimensionalidad? Muchas técnicas de reducción (PCA, t-SNE, autoencoders, etc.) asumen que los datos se concentran en una variedad de baja dimensión y buscan “desenrollar” esa variedad para encontrar una representación compacta. Por ejemplo, PCA proyecta los datos en las direcciones de mayor varianza (una variedad lineal) capturando la mayor parte de la información en menos componentes. Más allá de los datos, en las redes neuronales podemos pensar que **los parámetros entrenados también viven en variedades**: investigaciones han demostrado que las redes profundas están *sobreparametrizadas*, con grandes redundancias en sus pesos (). En otras palabras, los pesos de una capa suelen residir cerca de una **subvariedad** más simple, lo que abre la puerta a comprimir el modelo sin apenas alterar su comportamiento.

En la práctica, este concepto se aplica buscando **representaciones más compactas** de modelos y datos. Por ejemplo, un modelo entrenado puede ser proyectado a una variedad de menor dimensión mediante *factorizaciones matriciales* o *espacios latentes*, manteniendo casi el mismo rendimiento.

Técnicas como los **autoencoders** fuerzan a la red a codificar los datos de entrada en un espacio latente más pequeño (una variedad aprendida) y luego reconstruirlos; si la reconstrucción es buena, significa que la información relevante estaba contenida en esa variedad de menor dimensión. Esta compresión inherente no solo reduce la memoria y cálculos requeridos, sino que puede mejorar la **generalización**, al eliminar redundancias y ruido.

Descomposición en Valores Singulares (SVD) para redes neuronales

Una de las herramientas matemáticas más poderosas para la reducción de dimensionalidad en matrices es la **Descomposición en Valores Singulares (Singular Value Decomposition, SVD)**. SVD factoriza cualquier matriz W en el producto U , Σ , V^T, donde U y V son matrices ortogonales (básicamente bases de espacios columnales y fila) y Σ es diagonal con los **valores singulares** ordenados (SVD Compression for Neural Networks: A Practical Approach | by shashank Jain | AI Mind). Lo crucial es que podemos **truncar** esta descomposición tomando solo los k valores singulares más grandes y sus vectores asociados, obteniendo una aproximación de rango k de W que minimiza el error cuadrático (es la mejor aproximación de rango reducido según la teoría de Eckart-Young). En términos simples, SVD nos da la forma óptima de **aproximar una matriz grande por el producto de matrices más pequeñas**, capturando la "esencia" de los pesos originales con menos parámetros.

Aplicación en redes neuronales: Muchas capas de una red (especialmente las capas densas o totalmente conectadas, y también las convolucionales una vez reestructuradas) pueden verse como multiplicaciones de matrices. Tras entrenar la red, podemos aplicar SVD a los pesos de una capa para encontrar una representación de menor rango. Por ejemplo, si una capa densa tiene una matriz de pesos W de dimensión $m \times n$, la SVD puede descomponerla en $W \approx U_k , \Sigma_k , V_k^T$, con $k \ll \min(m,n)$. Esto se implementa como **dos capas consecutivas más pequeñas**: una capa con pesos U_k , Σ_k (de $m \times k$) seguida de otra con pesos V_k^T (de $k \times n$). El resultado es que en vez de computar $m \times n$ multiplicaciones, computamos $m \times k + k \times n$, que para k pequeño supone un gran ahorro de cómputo.

Varios estudios han demostrado que muchas redes profundas pueden comprimirse significativamente mediante SVD **sin apenas perder precisión**. Por ejemplo, Denton et al. (2014) aplicaron factorizaciones de baja r**a**n**g**o a los tensores de convolución y lograron acelerar la inferencia de CNNs en aproximadamente 2× con solo ~1% de pérdida de exactitud (Exploiting Linear Structure Within Convolutional Networks for ...). Esto es posible porque, como apuntábamos, hay mucha redundancia: Denil et al. (2013) encontraron que se puede predecir una gran fracción de los pesos de una red a partir de un subconjunto pequeño de ellos, evidenciando la sobreparametrización (). En otras palabras, *la información útil está concentrada en unas pocas combinaciones lineales de los pesos*, que SVD identifica como los vectores singulares principales. Al descartar los valores singulares pequeños, eliminamos componentes de peso menos importantes (ruido o detalles finos) y obtenemos un modelo más pequeño y rápido.

La SVD se puede aplicar de distintas formas en una red neuronal:

- **Capas densas:** Es directo aplicar SVD a la matriz de pesos. Tras la compresión, generalmente se realiza una ligera afinación (*fine-tuning*) entrenando el modelo comprimido unas pocas épocas para recuperar cualquier pequeña degradación de precisión.

- **Capas convolucionales:** Un filtro convolucional 2D de tamaño $C_{\text{out}} \times C_{\text{in}} \times k \times k$ puede reordenarse en una matriz W de $C_{\text{out}} \times (C_{\text{in}}\cdot k \cdot k)$ y luego factorizarla. Alternativamente, se pueden usar descomposiciones específicas (por ejemplo, factorizar una conv $k\times k$ en dos conv secuenciales $k\times 1$ y $1\times k$). En ambos casos, el objetivo es reducir la carga computacional explotando un rango bajo efectivo. *Ejemplo:* una convolución 5×5 puede descomponerse en una 5×1 + 1×5 si la matriz de pesos tiene rango reducido, ahorrando operaciones (Efficient Neural Network Compression) (Efficient Neural Network Compression).
- **Regularización durante el entrenamiento:** En vez de aplicar SVD post-entrenamiento, hay enfoques para incentivar durante el entrenamiento que los pesos tengan decaimiento rápido de valores singulares. Por ejemplo, agregando una penalización que sume los valores singulares pequeños para que el modelo concentre la información en unos pocos valores singulares grandes (SVD Compression for Neural Networks: A Practical Approach | by shashank Jain | AI Mind). De esta forma, el modelo aprendido es *más comprimible* de entrada.

En resumen, la SVD ofrece una forma sistemática de **reducir la dimensionalidad de las matrices de peso** en las redes neuronales. Menos dimensión significa menos multiplicaciones y adiciones en tiempo de inferencia, lo que en hardware de dispositivos móviles se traduce en **menor uso de CPU/GPU y menos consumo energético**, factores críticos para aplicaciones en smartphones. A diferencia de otras técnicas, la SVD garantiza la mejor aproximación lineal posible para un dado número de parámetros, aunque no captura no-linealidades (por eso suele combinarse con un pequeño reentrenamiento). Veamos a continuación cómo se compara con otras estrategias de optimización de modelos.

Comparación con técnicas alternativas: poda, cuantización y destilación

La compresión mediante SVD no es la única vía para aligerar modelos. Existen técnicas complementarias o alternativas que abordan el problema desde distintos ángulos:

- **Poda (*pruning*):** Consiste en **eliminar conexiones o neuronas redundantes** de un modelo entrenado. Por ejemplo, fijar a cero ciertos pesos considerados poco importantes (poda no estructurada) o remover filtros/canales completos (poda estructurada). Al hacerlo, el modelo resultante tiene menos parámetros efectivos y, si se implementa eficientemente, menos operaciones en tiempo de inferencia. La poda puede basarse en criterios como la magnitud de los pesos (eliminar los más pequeños), sensibilidad (evaluar cuánto empeora la pérdida al quitar un peso) o patrones estructurados (Reduzca el tamaño de su modelo de IA manteniendo el rendimiento). Estudios clásicos mostraron que entre un 80-90% de los pesos de una red pueden prunearse con mínima pérdida de exactitud, aprovechando la gran redundancia. La **ventaja** es que podemos lograr modelos muy compactos; la **desventaja** es que la red resultante es dispersa (muchos ceros), lo que en algunas plataformas no acelera proporcionalmente a la reducción de pesos debido a las ineficiencias de memoria.
- **Cuantización (*quantization*):** Busca **reducir la precisión numérica** usada para representar pesos y activaciones (Reduzca el tamaño de su modelo de IA manteniendo el rendimiento).

Típicamente, se baja de 32-bit en coma flotante (FP32) a 16-bit flotante (FP16) o incluso a 8-bit enteros (INT8). Un modelo cuantizado INT8 ocupa ~4 veces menos memoria que su versión FP32 y puede aprovechar instrucciones vectoriales especializadas para enteros, logrando aceleraciones notables (Reduzca el tamaño de su modelo de IA manteniendo el rendimiento). Muchos aceleradores (incluyendo el *Neural Engine* de Apple y la TPU de Google) están diseñados específicamente para operaciones en INT8, ofreciendo grandes mejoras de rendimiento. La cuantización puede hacerse **post-entrenamiento** (ajustando escalar y offset de cuantización y asumiendo ligera pérdida de precisión) o con **entrenamiento cuantizado** (incorporando la cuantización en el bucle de entrenamiento para mantener la precisión). En la práctica, la cuantización suele lograr grandes reducciones de latencia y consumo con impacto mínimo en la accuracy, especialmente para tareas de visión. Es quizá la técnica más efectiva en cuanto a relación ganancia/perdida cuando el hardware la soporta (¿Qué es la destilación de conocimientos? - LeaderGPU).

- **Destilación de conocimiento (*knowledge distillation*):** En vez de modificar un modelo directamente, la destilación entrena un **modelo "estudiante" más pequeño** para imitar a un **modelo "profesor" grande** (Reduzca el tamaño de su modelo de IA manteniendo el rendimiento). Introducida por Hinton et al., la idea es que el modelo grande contiene conocimiento (por ejemplo, en las distribuciones de probabilidad de sus salidas) que puede transferirse al pequeño. Se entrena el modelo pequeño con las salidas "suavizadas" del grande como objetivo (además de las etiquetas verdaderas), de modo que aprenda no solo a acertar, sino a *comportarse* como el grande. Al final obtenemos una red de tamaño reducido que alcanza un rendimiento cercano al modelo original (Reduzca el tamaño de su modelo de IA manteniendo el rendimiento). La destilación es muy útil combinada con las técnicas anteriores: por ejemplo, tras comprimir un modelo con poda/SVD/cuantización, podemos destilar desde el original para recuperar exactitud. Su punto fuerte es que optimiza la *estructura* del modelo (diseñamos el estudiante como queramos) en lugar de los pesos, y puede incluso mejorar la generalización del estudiante. El punto débil es que requiere un proceso de entrenamiento adicional y bien diseñado.

En general, estas técnicas **no son excluyentes**. De hecho, los mejores resultados suelen obtenerse combinándolas: primero entrenar un modelo grande, luego podar pesos, cuantizar a INT8, y destilar en el proceso de reafinamiento. La SVD misma se puede ver como una forma de poda estructurada de pesos (eliminando componentes lineales enteros en lugar de pesos individuales). Cada método ataca la eficiencia desde una perspectiva distinta:

- *SVD/low-rank:* reduce la complejidad lineal de las capas (menor rango).
- *Pruning:* elimina pesos/neuronas completas (estructura más esparsa).
- *Quantization:* reduce el costo por operación (operar con datos más pequeños).
- *Distillation:* entrena un modelo intrínsecamente más pequeño para ser inteligente.

A la hora de implementar en hardware real, la cuantización y la poda estructurada suelen tener impacto más inmediato (ya que hardware como TPU/ANE aprovecha bien matrices pequeñas densas en INT8). La SVD proporciona esas matrices más pequeñas densas; la destilación te da un modelo reducido ya optimizado. En dispositivos como Apple Silicon y Google Tensor, **se aprovechan intensamente la cuantización y operaciones vectoriales**, por lo que tener modelos de baja precisión y bajo número de dimensiones hace que más cálculo caiga dentro de los "puntos dulces" del hardware.

Implementación técnica en Apple Silicon (C++, ARM64 ASM, Metal)

Apple Silicon (M1, M2, M3) ofrece un entorno heterogéneo con CPU de alto rendimiento, GPU integrada potente, y un **Neural Engine (ANE)** dedicado para ML. Para optimizar redes neuronales en estos chips, los desarrolladores pueden actuar en varios niveles:

- **Optimización en CPU (C++/ASM):** Las CPU de Apple (núcleos Firestorm, Avalanche, etc.) implementan la arquitectura ARM64 con extensiones SIMD (NEON/ASIMD). Un código C++ eficiente debería vectorizar las operaciones para aprovechar registros de 128 bits (o 256 bits con SVE2 en chips más nuevos) que permiten procesar múltiples elementos a la vez. Por ejemplo, utilizando intrínsecos NEON podemos multiplicar vectores de 4 floats en una sola instrucción. Un pequeño fragmento ilustrativo en C++ con intrínsecos ARM NEON:

```
#include <arm_neon.h>
void vector_mul(const float *A, const float *B, float *C) {
    float32x4_t a = vld1q_f32(A);         // Carga 4 floats de A en un registro
SIMD
    float32x4_t b = vld1q_f32(B);         // Carga 4 floats de B
    float32x4_t result = vmulq_f32(a, b); // Multiplica elemento a elemento (4 a
la vez)
    vst1q_f32(C, result);                // Almacena los 4 resultados en C
}
```

En este ejemplo simple, en lugar de 4 multiplicaciones escalar sucesivas, realizamos las 4 en **paralelo SIMD** ([

```
Apple’s deep learning frameworks: BNNS vs. Metal CNN
```

](https://machinethink.net/blog/apple-deep-learning-bnns-versus-metal-cnn/#:~:text=,the%20CPU%E2%80%99s%20fast%20vector%20instructions)). De igual manera, librerías como **Accelerate/VecLib** de Apple internamente usan NEON/ASIMD para rutinas BLAS (multiplicación de matrices, etc.), sacando partido de instrucciones especializadas (por ejemplo, `fmla` para multiply-add y, en chips recientes, instrucciones de *dot product* para INT8). Para obtener el máximo rendimiento, a veces se recurre a escribir secciones críticas en ensamblador ARM64 directamente, ajustando el *pipeline* de instrucciones, uso de caches, *prefetch*, etc. Sin embargo, para la mayoría de casos, usar intrínsecos o librerías optimizadas es suficiente y más portable.

- **Optimización en GPU (Metal):** La GPU integrada en M1/M2/M3 es muy potente en cómputo paralelo (por ejemplo, la GPU de 8 núcleos del M1 alcanza ~2.6 TFLOPs en FP32 (Apple unleashes M1 - Apple)) y cuenta con la ventaja de la *memoria unificada* (CPU y GPU comparten RAM física, evitando copias) (Apple unleashes M1 - Apple). Apple provee varias formas de usar la GPU para ML. Una es a través de **Metal Performance Shaders (MPS)**, un conjunto de *kernels* optimizados (convoluciones, MLP, activaciones, etc.) que ejecutan en GPU de forma altamente optimizada ([

  ```
  Apple’s deep learning frameworks: BNNS vs. Metal CNN
  ```

](https://machinethink.net/blog/apple-deep-learning-bnns-versus-metal-cnn/#:~:text=that%20take%20full%20advantage%20of,the%20CPU%E2%80%99s%20fast%20vector%20instructions)). Otra es escribir directamente *shaders* de cómputo en **Metal** para implementar operaciones custom. Por

ejemplo, podríamos escribir un kernel Metal que calcule la activación de una capa convolucional tomando bloques de datos en paralelo. La GPU de Apple Silicon usa un enfoque *tile-based deferred rendering* que también aplica a cómputo: conviene estructurar los accesos a memoria de forma coalescente y aprovechar la alta cantidad de hilos. En general, operaciones de **matriz-densa** grandes (como multiplicar una matriz 1024x1024) pueden correr más rápido en la GPU que en la CPU, siempre que la latencia de lanzamiento y transferencia sea compensada por el paralelismo. Apple ha analizado durante años cargas típicas para afinar su GPU; de hecho, su biblioteca MPS está "calibrada" para muchas cargas de redes neuronales comunes, superando a la librería BNNS (CPU) en operaciones grandes ([

```
Apple’s deep learning frameworks: BNNS vs. Metal CNN
```

](https://machinethink.net/blog/apple-deep-learning-bnns-versus-metal-cnn/#:~:text=,instead%20of%20on%20the%20CPU)) ([

```
Apple’s deep learning frameworks: BNNS vs. Metal CNN
```

](https://machinethink.net/blog/apple-deep-learning-bnns-versus-metal-cnn/#:~:text=Apple%E2%80%99s%20deep%20learning%20frameworks%20are,possible%20through%20the%20network%E2%80%99s%20layers)). En nuestras pruebas, la GPU suele ser ideal para *batches* grandes o capas convolucionales pesadas, mientras que la CPU (especialmente los núcleos de alto rendimiento) puede bastar para MLP pequeños o lógica secuencial.

- **Neural Engine (ANE):** Es el acelerador específico de ML de Apple, con 16 núcleos diseñados para ejecutar **operaciones matriciales entero/fp16 a gran throughput** (por ej., 11 TOPS en M1, 15.8 TOPS en M2) (Neural Engine | Apple Wiki | Fandom) (Neural Engine | Apple Wiki | Fandom). Para aprovechar la ANE, Apple proporciona un nivel de abstracción más alto: **Core ML**. Los desarrolladores no pueden programar la ANE directamente via API pública (machine learning - How to leverage the Neural Engine on Apple Silicon/M1 processors as a developer? - Stack Overflow); en su lugar, deben convertir sus modelos entrenados a **Core ML format** (usando `coremltools`) y dejar que el *runtime* decida dónde ejecutar cada capa. Core ML en Apple Silicon es muy inteligente: es capaz de distribuir partes de la red entre CPU, GPU y ANE de manera transparente para optimizar rendimiento (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research). Por ejemplo, en un iPhone, Core ML puede ejecutar convoluciones en la ANE, pero si la red tiene alguna capa no soportada por ANE, esa capa se ejecutará en CPU/GPU, coordinando todo de forma híbrida. En macOS, frameworks como **ML Compute** también intentan utilizar el ANE cuando disponible. La recomendación es: **usar CoreML/ML Compute** siempre que sea posible para inferencia, ya que Apple ha optimizado ese camino para exprimir la ANE sin esfuerzo adicional del desarrollador. Apple ha mostrado ejemplos notables: un modelo Transformer (DistilBERT) optimizado para ANE logró hasta **10× más velocidad y 14× menos memoria** que su implementación original en CPU/GPU (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research). Esto se logró ajustando el modelo (p. ej. transformando capas Dense en Convs 1x1 para ajustarse mejor al formato preferido del ANE, y cuantizando pesos) y dejando que Core ML lo ejecute principalmente en la Neural Engine.

(Apple unleashes M1 - Apple) *Chip Apple **M1** con memoria unificada (los dos chips de DRAM a los lados) y el SoC en el centro. La arquitectura unificada permite que CPU, GPU y Neural Engine*

accedan a la misma memoria sin copias, mejorando rendimiento y eficiencia (Apple unleashes M1 - Apple).

En cuanto a *low-level*, aunque no podamos programar la ANE manualmente, sí podemos influir en el rendimiento para que Core ML la use: por ejemplo, utilizar tipos de datos compatibles (FP16/INT8) y capas soportadas. También es posible utilizar la librería BNNS (Basic Neural Network Subroutines) en CPU ([

```
Apple’s deep learning frameworks: BNNS vs. Metal CNN
```

](https://machinethink.net/blog/apple-deep-learning-bnns-versus-metal-cnn/#:~:text=,the%20CPU%E2%80%99s%20fast%20vector%20instructions)), que a veces es acelerada automáticamente por ANE cuando corresponde (casos de inferencia sencillos). Sin embargo, en general **la mejor práctica en Apple Silicon** es: 1) **cuantizar** el modelo (INT8) si es posible para que el ANE lo aproveche, 2) compilarlo a CoreML y 3) dejar que el sistema lo ubique en ANE/CPU/GPU según convenga (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research). Si se requiere personalización máxima (por ejemplo, una arquitectura no soportada por CoreML), entonces se puede optar por escribir kernels custom en Metal (GPU) o usar Accelerate en CPU, pero esto suele ser la excepción.

Optimización en Google Tensor (TPU móvil de Pixel)

Los chips **Google Tensor** (como el GS101 del Pixel 6, G2 del Pixel 7, G3 del Pixel 8, etc.) incorporan una **TPU integrada** diseñada para acelerar cargas de trabajo de IA en Android. Google suele referirse a ella simplemente como *TPU* o *Edge TPU*, y está especializada en operar con tensores cuantizados de forma muy eficiente. Para optimizar redes neuronales en este contexto, se deben considerar varias particularidades:

- **Uso de NNAPI y delegados:** En Android, el camino habitual para aprovechar aceleradores de ML es a través de la **Neural Networks API (NNAPI)** (Google's IP: Tensor TPU/NPU - Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance & Efficiency). Los desarrolladores pueden utilizar frameworks como **TensorFlow Lite** (TFLite) que ofrecen *delegados* NNAPI: esto permite que, al cargar un modelo, las capas soportadas sean derivadas al hardware especializado (TPU, DSP, GPU) del dispositivo. En el caso de Pixel 6/7/8, Google provee un driver NNAPI para su TPU, de modo que al habilitar NNAPI en TFLite, casi toda la red se ejecutará en la TPU en lugar del CPU. Es importante diseñar el modelo de forma **compatible con NNAPI** (por ejemplo, usando operaciones estándares y tipos compatibles, como INT8 o FP16). Google ha mencionado que en Pixel 6, los modelos ejecutados vía NNAPI en la TPU lograron aproximadamente **10×** la aceleración respecto a ejecutarlos en CPU (Improved On-Device ML on Pixel 6, with Neural Architecture Search). Por tanto, es fundamental asegurarse de compilar e inferir el modelo con NNAPI en vez de la CPU por defecto.
- **Cuantización a INT8:** La Edge TPU de Google está optimizada para enteros de 8 bits (similar a la Coral EdgeTPU externa de Google, que ofrece ~4 TOPS a 2W (Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance & Efficiency)). Para obtener máxima velocidad, se recomienda **cuantizar el modelo a INT8**. TFLite facilita esto con *post-training quantization*. Un modelo cuantizado INT8 podrá utilizar kernels acelerados en

la TPU, mientras que si se deja en float32 puede acabar ejecutándose en CPU (o en GPU con OpenGL/Vulkan). La diferencia de rendimiento es enorme: la TPU puede ejecutar **miles de millones de operaciones INT8 por segundo con bajo consumo**, mientras que en float tendría que caer a CPU o usar emulación. En Pixel 6, muchos modelos integrados (traducción en vivo, procesamiento de imágenes) usan INT8 internamente. Un dato técnico: la TPU de Pixel opera a frecuencias elevadas (~1 GHz+) y puede llegar a ~5 W en picos (Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance & Efficiency), entregando una potencia de cómputo efectiva que rivaliza o supera a la CPU/GPU del dispositivo para tareas de ML.

- **Optimización del modelo para la TPU:** Google ha ido más allá del hardware puro; ha aplicado *Neural Architecture Search (NAS)* para crear modelos *MobileNetEdgeTPU* optimizados para su TPU (Improved On-Device ML on Pixel 6, with Neural Architecture Search) (Improved On-Device ML on Pixel 6, with Neural Architecture Search). Estos modelos tienen bloques de convolución diseñados para ajustarse a la forma en que la TPU maneja la memoria y la computación (por ejemplo, usan más operaciones que favorecen enteros y evitan patrones poco eficientes en la TPU). En general, al portar un modelo a Pixel, conviene revisar si existe una variante optimizada (Google suele publicar en TensorFlow Hub modelos "...EdgeTPU" adaptados). Si no, al menos asegurarse de **evitar capas no soportadas por NNAPI** (como ciertas operaciones personalizadas) o de dividir la tarea en sub-tareas compatibles. Cabe destacar que la TPU del Tensor ha mostrado un desempeño sobresaliente en tareas de **procesamiento de lenguaje natural**: en benchmarks de MLPerf, el Pixel 6 (Tensor) alcanzó ~70 inferencias/seg en un modelo BERT compacto, superando **3×** al Snapdragon 888 y **5×** al Apple A15 en esa prueba (Google's IP: Tensor TPU/NPU - Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance & Efficiency).

(Google's IP: Tensor TPU/NPU - Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance & Efficiency) *Gráfico comparativo de rendimiento en tareas de **procesamiento de lenguaje natural** (MobileBERT, MLPerf 1.0.1). El Google **Tensor** (Pixel 6 Pro) alcanza ~69.8 inferencias/segundo, aventajando claramente al Snapdragon 888 (~23.2) y Apple A15 (~14) en dicho test (Google's IP: Tensor TPU/NPU - Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance & Efficiency). El diseño de la TPU de Google está muy focalizado en este tipo de cargas.*

- **Programación de bajo nivel:** A diferencia de Apple, donde la ANE no es accesible directamente, en la TPU de Google tampoco existe una API pública de "bajo nivel" para programarla; pero Google ofrece el **Edge TPU Compiler** (para la versión discreta Coral) y en Android la idea es que NNAPI es la capa de abstracción. Un desarrollador avanzado podría, en teoría, usar la biblioteca **Android Neural Networks API** directamente en C/C++ para cargar y ejecutar modelos, pero es más común dejar que TFLite lo maneje. No se acostumbra escribir código assembly específico para la TPU; en su lugar, se prepara el modelo adecuadamente. Sí es posible optimizar a nivel de **CPU ARM** en Tensor también (los Pixel usan núcleos ARM Cortex-X1/A78/A55 o similares). Si por algún motivo una parte de la red corre en CPU (por ej., una capa no soportada por NNAPI), aplicar igualmente técnicas SIMD NEON y multihilo (vía Android NDK) ayudará. Asimismo, la **GPU (Mali)** del Pixel puede usarse via **TFLite GPU delegate** si la TPU no soporta algo, aunque la GPU

Mali suele ser menos eficiente en INT8 que la TPU. En resumen, en Google Tensor el mayor rendimiento vendrá de **ejecutar todo lo posible en la Edge TPU**; esto implica modelos bien cuantizados, usar NNAPI, y posiblemente trocear la tarea para que la mayor parte caiga en la TPU.

- **Medición y ajustes:** Google proporciona la herramienta **Android Profiler** y utilidades en `adb` para medir el uso de la TPU/CPU/GPU. Es importante probar el modelo en un Pixel real y confirmar (por logs de NNAPI) que la mayoría de operaciones se asignan a la **unidad PIXEL_NPU** (como aparece en los logs). Si partes del modelo caen a CPU, considerar reescribir esas partes (p.ej. ciertas activaciones no soportadas se pueden reemplazar por ReLU6, etc.). Además, prestar atención a la **termalización**: la TPU compartirá el presupuesto térmico del SoC, y en cargas sostenidas largas podría throttlear. Aun así, su eficiencia es mayor que CPU/GPU para ML sostenido, así que mantendrá mejor el rendimiento bajo restricciones de potencia.

Benchmarks de rendimiento: CPU vs GPU vs aceleradores

Para entender el impacto real de estas optimizaciones, veamos comparativas de rendimiento en distintos escenarios:

- **Apple Silicon (M1/M2/M3):** Apple anunció que al incorporar el Neural Engine de 16 núcleos en Mac (M1), ciertas tareas de ML podían ser hasta **15× más rápidas** que ejecutándolas solo en CPU (Apple unleashes M1 - Apple). Esto se debe a la capacidad de la ANE de ejecutar **paralelamente 11 trillones de operaciones/s** (TOPS) en M1, liberando a la CPU de esa carga. En pruebas prácticas, un modelo como MobileNet v2 realiza inferencia en ~10 ms en ANE frente a ~100-150 ms en CPU, corroborando ~10-15× de mejora. En el M2 (gracias a un ANE mejorado de 15.8 TOPS) se logra otro salto de ~40% (Neural Engine | Apple Wiki | Fandom). En cuanto a la **GPU vs ANE**, depende del tipo de cálculo: la **GPU** puede manejar FP32/FP16 con flexibilidad, pero la **ANE** es extremadamente rápida en INT8/FP16 cuantizados. Por ejemplo, en GeekBench ML (pruebas estandarizadas de inferencia), un **M4 Pro** obtuvo en una tarea cuantizada un score ~3.6 veces mayor en ANE comparado con su GPU integrada ([Geekbench ML Results
- Geekbench](https://browser.geekbench.com/ai/v1#:~:text=ML%20Neural%20Engine%20%201874,Product%20Name%20NVIDIA%20GeForce%20RTX)). Sin embargo, en precisión simple (FP32) la GPU supera a la ANE (ya que la ANE está optimizada para baja precisión). Esto sugiere una recomendación clara: para **maximizar throughput en Apple** usar datos FP16 o INT8, permitiendo a la ANE lucirse, mientras que la GPU puede apoyar en aquello que requiera FP32 o gráficos.
- **Google Tensor (Pixel TPU) vs CPU/GPU:** Como mencionamos, Google reportó hasta ~10× aceleración de la TPU vs CPU en Pixel 6 (Improved On-Device ML on Pixel 6, with Neural Architecture Search). En benchmarks independientes, la Pixel TPU suele situarse por encima de la GPU Adreno de Qualcomm en la mayoría de tareas ML y a menudo por encima de la CPU en órdenes de magnitud. En MLPerf Mobile 1.1, el Pixel 6 Pro logró ~942 inferencias/seg en clasificación de imágenes, superando al Exynos 2100 (~804/s) y al Apple A15 (~613/s) cuando estos usaban sus NPUs/CPU respectivas (Google's IP: Tensor TPU/NPU - Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance &

Efficiency). Solo el Snapdragon 888 (con DSP altamente optimizado) pudo superarlo en clasificación tras optimizaciones posteriores (Google's IP: Tensor TPU/NPU - Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance & Efficiency). Pero en *lenguaje natural*, la ventaja del Pixel fue masiva: ~70 vs ~14 inferencias/s sobre A15 (Google's IP: Tensor TPU/NPU - Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance & Efficiency) (ver gráfico anterior). Esto refleja que la TPU de Google está muy optimizada para modelos tipo Transfomer/BERT (central en funcionalidades de Pixel como *Live Translate*). En términos de **eficiencia energética**, estos aceleradores permiten realizar tareas intensivas con una fracción de la energía que consumiría la CPU/GPU. Apple indicó que el A12 Bionic podía realizar tareas de ML 9× más rápido gastando 1/10 de la energía de A11 (Neural Engine | Apple Wiki | Fandom), un salto enorme en eficiencia. De forma similar, Google diseñó su TPU móvil para ejecutar algoritmos de cámara y voz continuamente sin drenar la batería. Un ejemplo práctico: el Pixel 6 puede transcribir voz a texto en tiempo real con el modelo de Google Assistant íntegramente en dispositivo; algo inviable en CPU a igual nivel de precisión por consumo y latencia.

En general, los **aceleradores especializados (NPU/TPU)** brillan cuando el modelo se adapta a ellos (por eso insistimos en cuantización y compatibilidad). Las CPU modernizadas (como los núcleos Avalanche del M2) siguen siendo muy potentes en ML a pequeña escala y ofrecen baja latencia para modelos pequeños, pero escalan mal en energía si intentan igualar a las NPUs en throughput. Las GPUs ofrecen un término medio, con buena flexibilidad y bastante paralelismo, útiles si el modelo no encaja del todo en la NPU o si se combinan tareas gráficas/ML. Lo ideal es **repartir el trabajo**: usar la NPU/TPU para el grueso matemático (convoluciones, matmuls grandes) y dejar a CPU/GPU tareas auxiliares o partes no soportadas. Apple ya hace esto automáticamente (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research). En Android, depende más del developer integrar bien NNAPI y delegar correctamente.

Por último, mencionar que los **benchmark sintéticos** (como GeekBench ML) dan una idea de la capacidad máxima, pero en escenarios reales las cifras pueden variar. Aún así, es claro que un modelo optimizado (compacto y cuantizado) puede inferirse en, digamos, 5-10 ms en un teléfono moderno, mientras que un modelo no optimizado (grande y en FP32) podría tardar 100-200 ms en las mismas condiciones. Esa diferencia determina si una función de IA se siente instantánea o lenta, o si es viable ejecutarla continuamente en segundo plano.

Aplicaciones prácticas en IA móvil y eficiencia energética

Las optimizaciones descritas no son solo ejercicios académicos; permiten **nuevas experiencias de IA en dispositivos móviles** manteniendo un consumo de energía razonable y respetando la privacidad (al no requerir servidor). Algunas aplicaciones destacadas:

- **Fotografía computacional:** Tanto Apple como Google emplean redes neuronales en sus cámaras para mejoras en tiempo real. Por ejemplo, Google Pixel utiliza la TPU para **Night Sight** (fotografía nocturna con varias tomas), **Magic Eraser** (borrar objetos) y clasificar contenido en la galería. Apple aplica redes para detección de escenas, segmentación de personas en retratos y recientemente para **Live Text** (reconocimiento de texto en imágenes) en iOS. Estas tareas requieren procesar megapíxeles rápidamente; gracias a aceleradores dedicados y modelos comprimidos, pueden hacerse en milisegundos sin agotar la batería.

- **Asistentes de voz y traducción:** Los modelos de reconocimiento de voz y NLP (ej. **Siri**, **Google Assistant**, **Live Translate**) se han llevado on-device. Pixel 6 estrenó la traducción en tiempo real de audio a texto y de un idioma a otro directamente en el teléfono, algo posible por la TPU. Apple, con iOS 15+, también permite el dictado continuo en el dispositivo. Un modelo grande de reconocimiento de voz que antes corría en la nube, ahora corre localmente en una variedad optimizada (distilled) usando la ANE. El resultado es menor latencia (no depende de red) y garantiza funcionamiento offline, todo dentro de límites de consumo aceptables.
- **Realidad aumentada y visión por computadora en vivo:** Aplicaciones AR como **Ikea Place** o funciones como **Detección de movimiento** en video se benefician de redes optimizadas para detectar planos, objetos o pose humana en tiempo real. Con un modelo pequeño y ANE/TPU, un teléfono puede detectar una persona en la escena a 30 FPS, habilitando efectos AR creíbles. Apple ha promocionado cómo su chip puede ejecutar ML para reconocer gestos y posiciones corporales sin accesorios adicionales. La eficiencia aquí es clave: se quiere correr estas redes continuamente mientras la cámara está activa, y eso implica decodificar video + inferencia ML + rendering gráfico. Cada optimización suma para lograrlo dentro del presupuesto de ~5W disponible para todo el sistema en un smartphone sin sobrecalentarlo.
- **Salud y aplicaciones always-on:** Los wearables y móviles monitorean constantemente señales (ej. latidos, actividad física) usando modelos diminutos de ML. Apple Watch, por ejemplo, detecta caídas y ritmos cardíacos anómalos con redes neuronales entrenadas y comprimidas para su acelerador. Estos modelos corren **siempre activos** en segundo plano, por lo que deben ser extremadamente eficientes. Apple Neural Engine y la arquitectura big.LITTLE del SoC permiten dedicar un núcleo eficiente o el NPU para estas tareas sin despertar los núcleos potentes, prolongando la batería. En Pixel, existe un *Context Hub* de muy bajo consumo para IA continua (por ejemplo, la detección de frases como “Hey Google” usa un modelo pequeño en un DSP aparte). En todos los casos, la *eficiencia energética* viene dada tanto por hardware especializado como por modelos optimizados (pequeños y de baja precisión). Un modelo 10× más ligero puede significar la diferencia entre 1 día o 1/2 día de batería si se usa intensivamente.

En resumen, las optimizaciones permiten **llevar la IA al día a día**: traducciones instantáneas de carteles con la cámara, filtros de realidad aumentada en videollamadas, notificaciones inteligentes que entienden contexto, etc., sin depender de la nube. Además, al ejecutarse localmente, preservan mejor la privacidad y funcionan sin conexión. Pero todo ello sería impracticable si cada inferencia tomara mucho tiempo o energía; por eso se requiere tanto el avance en **modelos eficientes** como en **chips especializados**.

Recomendaciones y mejores prácticas para desarrolladores

Finalmente, recopilamos una serie de recomendaciones prácticas para desarrolladores que deseen optimizar redes neuronales en hardware Apple Silicon o Google Tensor:

1. **Aplica compresión de modelo antes de la implantación:** Si entrenaste un modelo grande en GPU, considera técnicas de **podado y SVD** para reducir su tamaño. Incluso un retraining

corto con los pesos comprimidos puede mantener la precisión. Menos parámetros suelen equivaler a inferencia más rápida en dispositivos móviles.

2. **Cuantiza a menor precisión (INT8/FP16):** Siempre que la tarea lo permita, convierte tu modelo a FP16 o INT8. En Apple, un modelo Core ML en FP16 podrá utilizar la ANE al máximo (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research) (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research). En Android, exporta modelos TFLite cuantizados a INT8 para usar NNAPI en la Edge TPU. Asegúrate de calibrar bien la cuantización para minimizar la pérdida de accuracy.

3. **Usa las bibliotecas y frameworks nativos:** No reinventes la rueda en bajo nivel si no es necesario. En Apple, utiliza **Core ML** para inferencia – el sistema decidirá la mejor combinación de CPU/GPU/ANE por ti (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research). En Android, utiliza **TensorFlow Lite** con NNAPI o los APis de **ML Kit**; te ahorrarán manejar detalles de drivers y garantizan portabilidad. Estas herramientas suelen incorporar optimizaciones al compilar el modelo (p. ej., fusionar operaciones, reordenar tensores) específicas para cada hardware.

4. **Perfil de rendimiento y cuellos de botella:** Prueba tu aplicación con las herramientas de profiling (Instruments en iOS, Android Profiler) para ver dónde se consume tiempo. ¿Está la inferencia ejecutándose en el acelerador esperado? ¿La CPU se queda esperando transferencias de memoria? Detectar si quizás una capa está cayendo fuera del acelerador (p. ej., una operación no soportada en ANE/TPU) te permitirá ajustar el modelo (quizá reemplazar esa operación por otra equivalente). Observa también el uso de energía: en iOS, los Logs de energía te pueden indicar si la ANE está activa o si la CPU estuvo a tope.

5. **Optimiza el uso de la CPU y subprocesos:** Si parte de la inferencia ocurre en CPU (por diseño o por falta de soporte), asegúrate de usar **vectorización** (por intrínsecos o libs optimizadas) y **multithreading**. Las redes suelen ser fácilmente paralelizables por lote o por capa. En Apple, `Accelerate.framework` (vDSP, BNNS) aprovechará NEON y varios núcleos automáticamente. En Android, configura el número de hilos de TFLite según la carga (por ej., 4 threads en big cores para una tarea heavy si no hay NNAPI).

6. **Aprovecha la GPU si corresponde:** Si tu tarea es gráfica o combina gráficos+ML (ej: un filtro estilo Prisma aplicado a cámara en vivo), la GPU puede manejar ambos aspectos. En Apple, usar **MPS** para convoluciones pesadas mantendrá los datos en GPU, evitando copias ([

   `Apple’s deep learning frameworks: BNNS vs. Metal CNN`

](https://machinethink.net/blog/apple-deep-learning-bnns-versus-metal-cnn/#:~:text=,instead%20of%20on%20the%20CPU)). En Android, un delegado GPU puede ser útil si la TPU no está disponible (por ejemplo, en un móvil sin acelerador dedicado). La GPU suele rendir mejor con lotes más grandes; si procesas de uno en uno, la NPU/CPU tal vez sea mejor.

7. **Diseña pensando en el hardware:** Ten en cuenta las *limitaciones de los aceleradores*. Por ejemplo, el **ANE** prefiere tensores 4D en formato channels-first (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research); el **Pixel TPU** prefiere ciertas dimensiones de tensores (múltiplos de 8, etc.) y evita operaciones no soportadas como convoluciones dilatadas muy grandes. A veces, una pequeña modificación en la arquitectura

(p. ej., usar 2 capas 3×3 en vez de 1 capa 5×5) puede hacer tu modelo compatible y más eficiente en cierto hardware. Revisa documentación de Apple (Tech Notes de CoreML) o de Google (guías de NNAPI) para conocer las mejores prácticas de diseño de modelos para cada acelerador.

8. **Mantén los modelos lo más pequeños posible:** Además de la velocidad, en móviles importa el **uso de memoria**. Un modelo más pequeño ocupa menos RAM (importante en dispositivos con 4GB o menos) y cache. También reduce el tamaño de la app descargada o las actualizaciones OTA de modelos. Considera técnicas como **distillation** para obtener una red pequeña desde el inicio en lugar de llevar siempre el modelo gigante. Muchas veces varios modelos pequeños especializados pueden ser mejores que un monolítico grande.
9. **Prueba en distintos dispositivos:** La fragmentación de Android implica que tu app puede correr en móviles con o sin acelerador ML. Diseña con **degradación graciosa**: si no hay NNAPI disponible, quizás usar la GPU o en el peor caso CPU con un modelo aún más pequeño. En iOS, todos los A12 en adelante tienen Neural Engine, pero la potencia varía (p.ej. ANE de A14 vs A11 es 18× más rápida (Neural Engine | Apple Wiki | Fandom)). Asegúrate de que incluso en dispositivos más antiguos la app funciona (quizá con menor frecuencia de inferencia o usando un modelo de menor tamaño). Core ML y TFLite delegados suelen hacer esta adaptación automáticamente, pero vale la pena confirmarlo.
10. **Considera la temperatura y eficiencia sostenida:** Si tu aplicación hará uso intensivo continuo del ML (ej: navegación AR por 30 min), realiza pruebas de estrés. Un modelo que corre a 30 FPS el primer minuto pero baja a 15 FPS tras 5 minutos por *thermal throttling* puede ser problemático. En tal caso, tal vez debas optar por un modelo aún más pequeño o limitar la frecuencia de inferencia. Apple Silicon en Macs tiene buen margen térmico (ventilación en modelos Pro), pero en móviles la disipación es limitada. Google Tensor ha sido criticado por calentarse en tareas prolongadas; puede ser útil ceder parte del trabajo a hardware menos caliente (ej. periodicamete usar GPU en lugar de TPU para bajar temperatura, si la latencia lo permite). Estas decisiones son complejas, pero en apps de misión crítica conviene monitorizar la temperatura y adaptar cargas dinámicamente.

En conclusión, optimizar redes neuronales para Apple Silicon y Google Tensor requiere un enfoque multidisciplinario: entender la matemática (variedades, SVD), aplicar estrategias de compresión (poda, cuantización, distilación) y conocer las interioridades del hardware (SIMD CPU, GPU paralela, NPU/TPU especializada). Al seguir las mejores prácticas anteriores, los desarrolladores pueden lograr **modelos que corren en el borde de forma rápida y eficiente**, brindando experiencias de IA fluidas a los usuarios de dispositivos móviles modernos. La sinergia entre *diseño de modelos* y *capacidades de hardware* es la clave: modelos más pequeños y formatos adecuados sacarán a relucir la potencia de chips como el M1 o el Google Tensor, cumpliendo la promesa de la **inteligencia artificial ubicua y de bajo consumo** directamente en nuestros bolsillos.

Referencias:

1. Bengio, Y. et al. *"Manifold hypothesis in machine learning"*. Explicación del supuesto de que los datos de alta dimensión se distribuyen en variedades de baja dimensión (Manifold hypothesis - Wikipedia) (Manifold hypothesis - Wikipedia).
2. Denton, E. et al. *"Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation"*, NeurIPS 2014. Uso de SVD para acelerar redes convolucionales, logrando ~2×

compresión con mínima pérdida (Exploiting Linear Structure Within Convolutional Networks for ...).

3. Denil, M. et al. *"Predicting Parameters in Deep Learning"*, NIPS 2013. Evidencia de sobreparametrización: muchos pesos son predecibles a partir de unos pocos ().
4. Konfuzio Blog. *"Reduzca el tamaño de su modelo de IA manteniendo el rendimiento"*. Resumen en español de poda, cuantización y destilación de modelos (Reduzca el tamaño de su modelo de IA manteniendo el rendimiento) (Reduzca el tamaño de su modelo de IA manteniendo el rendimiento) (Reduzca el tamaño de su modelo de IA manteniendo el rendimiento) (Reduzca el tamaño de su modelo de IA manteniendo el rendimiento).
5. Apple Inc. *Apple Newsroom (2020) – "Apple unleashes M1"*. Comunicado sobre M1: 16-core Neural Engine, 11 TOPS, hasta 15× aceleración en ML (Apple unleashes M1 - Apple) y memoria unificada para CPU/GPU/NPU (Apple unleashes M1 - Apple).
6. Apple Machine Learning. *"Deploying Transformers on the Apple Neural Engine"* (2022). Detalles sobre ANE en A15/M1, 15.8 TFlops en FP16 (26× vs A11) (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research) (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research) y guía para optimizar modelos Core ML en ANE (Deploying Transformers on the Apple Neural Engine - Apple Machine Learning Research).
7. StackOverflow: *"How to leverage the Neural Engine on M1?"* (2021). Confirma que no hay API pública para ANE, se accede vía Core ML / Accelerate (machine learning - How to leverage the Neural Engine on Apple Silicon/M1 processors as a developer? - Stack Overflow).
8. Hollemans, M. *"Apple's deep learning frameworks: BNNS vs. Metal CNN"* (2017). Explica BNNS (CPU NEON) vs MPSCNN (GPU) ([

   `Apple's deep learning frameworks: BNNS vs. Metal CNN`

](https://machinethink.net/blog/apple-deep-learning-bnns-versus-metal-cnn/#:~:text=,the%20CPU%E2%80%99s%20fast%20vector%20instructions)) ([

   `Apple's deep learning frameworks: BNNS vs. Metal CNN`

](https://machinethink.net/blog/apple-deep-learning-bnns-versus-metal-cnn/#:~:text=that%20take%20full%20advantage%20of,the%20CPU%E2%80%99s%20fast%20vector%20instructions)) y su uso en inferencia.
9. Google AI Blog. *"Improved On-Device ML on Pixel 6, with Neural Architecture Search"* (2021). Describe la TPU del Pixel 6, ~10× más rápida que CPU (Improved On-Device ML on Pixel 6, with Neural Architecture Search), y modelos MobileNetEdgeTPU optimizados (Improved On-Device ML on Pixel 6, with Neural Architecture Search) (Improved On-Device ML on Pixel 6, with Neural Architecture Search).
10. Frumusanu, A. *"Google's Tensor SoC: Performance & Efficiency"*, AnandTech (2021). Análisis del Pixel 6: la TPU (EdgeTPU) proviene de la ASIC 4 TOPS@2W (Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance & Efficiency) y arrasa en tareas de lenguaje (MobileBERT) frente a Snapdragon/Exynos/Apple (Google's IP: Tensor

TPU/NPU - Google's Tensor inside of Pixel 6, Pixel 6 Pro: A Look into Performance & Efficiency).

11. GeekBench ML Browser (2025). Resultados comparativos de inferencia en distintos dispositivos. Ejemplo: en Mac Mini M4 Pro, ANE (quantized) ≈3.6× GPU ([Geekbench ML Results

- Geekbench](https://browser.geekbench.com/ai/v1#:~:text=ML%20Neural%20Engine%20%201874,Product%20Name%20NVIDIA%20GeForce%20RTX)); en iPhone 12 Pro (A14), ANE >> CPU en Half precision ([Geekbench ML Results
- Geekbench](https://browser.geekbench.com/ai/v1#:~:text=Sun%2C%2002%20Mar%202025%2015%3A10%3A23,5751%20%2036240%20%2049374)).

12. Apple Wiki – *"Neural Engine"* (actualizado 2023). Historial de Neural Engine: A11 0.6 TOp/s, A14/M1 11 TOp/s, A15/M2 15.8 TOp/s, etc., y eficiencias generacionales (Neural Engine | Apple Wiki | Fandom) (Neural Engine | Apple Wiki | Fandom) (Neural Engine | Apple Wiki | Fandom).