Facsímil 01

Los cimientos

Capítulo 10

Entrenamiento frente a inferencia: dos mundos distintos

686f6c61 · 2026

Capítulo 10: Entrenamiento frente a inferencia: dos mundos distintos

Entrando en el tema

Has pasado nueve capítulos entendiendo cómo funciona una red neuronal por dentro. Neuronas, capas, retropropagación, pérdidas, optimizadores. Todo eso ocurre durante el **entrenamiento**: el proceso de crear el modelo.

Pero tú, como ingeniera, probablemente nunca entrenarás un LLM desde cero. Lo que harás, cientos de veces al día, es **inferencia**: usar un modelo ya entrenado para obtener respuestas. Y estos dos mundos son radicalmente distintos. Confundirlos es como confundir construir una fábrica con comprar el producto que sale de ella.

Este capítulo traza la frontera. Y de paso, explica qué hay entre medias: el *fine-tuning*, la cuantización, y el *pipeline* que convierte una idea en un modelo funcionando en producción.

Entrenamiento vs inferencia

ASPECTO	ENTRENAMIENTO	INFERENCIA
Qué hace	Ajusta parámetros para reducir una pérdida	Usa parámetros ya ajustados para responder
Duración	Desde minutos en un modelo pequeño hasta meses en un modelo fundacional	Desde milisegundos hasta minutos según modelo, contexto y carga
Hardware	CPU/GPU local para modelos pequeños; clústeres enormes para frontera	CPU, GPU local, servidor propio, API o clúster de inferencia
Coste	Barato en prácticas pequeñas; enorme en pre-entrenamiento a escala	Por petición, por token, por GPU/hora o por infraestructura propia
Quién	Estudiantes, equipos de datos, universidades y laboratorios; frontera: grandes organizaciones	Cualquier equipo que consuma o sirva un modelo
Frecuencia	Cuando cambian datos, objetivo o modelo	Cada petición de usuario o proceso automático

El entrenamiento es crear o modificar el modelo ajustando parámetros.[1] Si hablamos de un modelo fundacional de frontera, los datos curados se procesan durante semanas o meses en clústeres de GPUs. Si hablamos de un clasificador pequeño, una red de visión acotada o un *fine-tuning* con adaptadores, el entrenamiento puede ocurrir en un portátil potente, una única GPU o una instancia alquilada. La palabra es la misma; la escala no.

La inferencia es usar el modelo.[2] Tu *prompt* llega a un runtime, el modelo procesa tokens y recibes una salida. Puede ser una llamada de API, un modelo local con `llama.cpp`, un servidor vLLM, una función interna o un clúster con *batching*. En producción, inferir no es "apretar un botón": hay latencia, memoria de KV cache, límites de contexto, colas, cuotas, coste por token y monitorización.

El *fine-tuning* está a medio camino: no creas un modelo desde cero, pero sí reentrenas parcialmente uno existente con tus propios datos. Es mucho más barato que el entrenamiento completo, pero más caro que la simple inferencia. Cambia el estilo y comportamiento del modelo, no le «enseña datos nuevos» en tiempo real. Para acceder a datos cambiantes, RAG es mejor opción.

Hay una forma especialmente barata de hacer *fine-tuning* que conviene conocer: **LoRA** (*Low-Rank Adaptation*). En lugar de reentrenar los miles de millones de pesos del modelo, LoRA los congela y entrena solo unas matrices nuevas y pequeñas (de «bajo rango») que se suman a algunas capas. Como esas matrices son una fracción mínima de los parámetros, el ajuste cabe en una sola GPU y produce un adaptador de pocos megabytes en lugar de un modelo entero. **QLoRA** da un paso más: cuantiza el modelo base a 4 bits y entrena los adaptadores encima, lo que permite ajustar modelos enormes en hardware modesto.

Las tres fases del entrenamiento

El entrenamiento de un LLM moderno no es un solo paso: son tres fases encadenadas, cada una con un propósito distinto.

Fase 1: Pre-entrenamiento. El modelo se entrena sobre cantidades masivas de texto (billones de tokens) con un objetivo simple: predecir la siguiente palabra. No hay etiquetas humanas: el propio texto proporciona la señal. Esta fase consume el 99 % del coste total y produce un modelo que «sabe lenguaje» pero no sabe conversar. Es la fase que más se beneficia de las *scaling laws*: más datos + más parámetros + más cómputo = modelo más capaz.[3]

Fase 2: Post-entrenamiento (SFT). Se entrena al modelo con ejemplos de conversaciones de alta calidad escritas por personas. El modelo pre-entrenado ya sabe completar frases; el SFT le enseña a responder preguntas, seguir instrucciones y mantener un tono conversacional. Es mucho más barato que la fase 1: miles de ejemplos en lugar de billones de tokens.

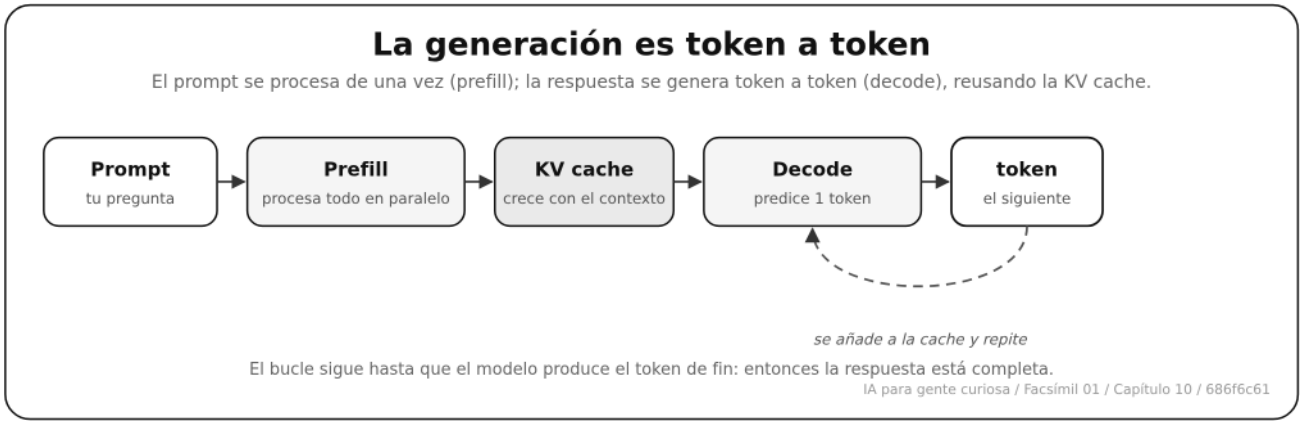
Fase 3: Alineamiento (RLHF/DPO). Personas comparan pares de respuestas del modelo y eligen cuál es mejor. Esa señal de preferencia se usa para refinar el comportamiento: ser útil pero no peligroso, admitir cuando no sabe algo, rechazar peticiones inapropiadas. Es la fase más barata y la que marca la diferencia entre un modelo técnicamente capaz y uno con el que realmente quieres interactuar.

Cómo funciona por dentro

Inferir es «usar el modelo», pero por dentro la generación de texto sigue un mecanismo concreto que conviene conocer, porque explica casi todas las decisiones de coste y latencia. Un LLM genera texto de forma **autoregresiva**: produce un token cada vez, y cada token depende de todos los anteriores. El proceso tiene dos fases.

Fase 1: prefill. Cuando envías tu *prompt*, el modelo lo procesa entero de una sola pasada. Para cada token del *prompt* calcula, en cada capa de atención, sus claves (*keys*) y sus valores (*values*), y los guarda en la **KV cache**. Esta fase es paralela: todos los tokens del *prompt* se procesan a la vez, así que es rápida aunque el *prompt* sea largo. Al terminar el prefill, el modelo predice el primer token de la respuesta.

Fase 2: decode. Aquí se genera la respuesta, token a token. En cada paso, el modelo toma el último token generado, calcula su clave y su valor, los añade a la KV cache, y usa toda la cache para predecir el siguiente token. Ese token se añade a la secuencia y el ciclo se repite, hasta que el modelo produce un token especial de fin. La clave está en la cache: gracias a ella, el modelo no recalcula la atención de todos los tokens anteriores en cada paso, solo procesa el token nuevo. El precio es la memoria, que crece con la longitud del contexto, justo lo que vimos al hablar de la KV cache.



Esta estructura explica dos métricas que oirás en producción. El ***time to first token*** (tiempo hasta el primer token) lo domina el prefill: procesar un *prompt* largo tarda más en arrancar. Los ***tokens por segundo*** los domina el decode: como cada token depende del anterior, la

generación es secuencial y no se puede paralelizar dentro de una misma respuesta, por muchas GPUs libres que tengas. Por eso un modelo puede ir «rápido» con respuestas cortas y sentirse «lento» cuando genera un texto largo: cada token es un eslabón más de la cadena.

Programación clásica vs machine learning

El *machine learning* no sustituye a la ingeniería de *software*: cambia dónde escribes la lógica.

En **software clásico**, una persona escribe reglas explícitas: «si el *email* contiene 'gratis' y 'clic aquí', sube la puntuación de *spam*». La lógica vive en código mantenido por personas.

En **machine learning**, das ejemplos etiquetados y dejas que el modelo descubra las reglas. En lugar de escribir «si contiene X, entonces Y», le das diez mil *emails* etiquetados como *spam* o no *spam* y dejas que encuentre los patrones. La lógica queda codificada en parámetros aprendidos, no en sentencias `if`.

En **inferencia**, usas el modelo entrenado: llega un *email* nuevo y el modelo devuelve una probabilidad de *spam*.

ASPECTO	SOFTWARE CLÁSICO	MACHINE LEARNING
Fuente de la lógica	Reglas escritas por personas	Patrones aprendidos de datos
Cómo se corrige	Modificar código	Mejorar datos, etiquetas o arquitectura
Testing	Casos deterministas	Evaluaciones estadísticas
Explicación	Se rastrea la rama de código	Se analizan pesos, datos y *prompts*

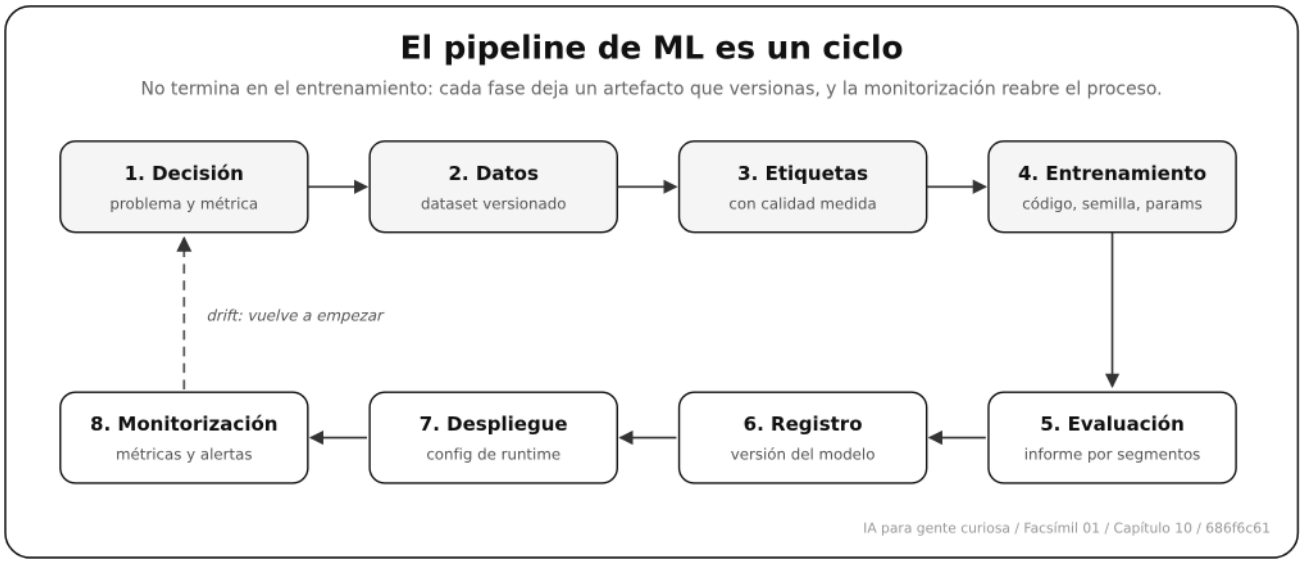
El riesgo principal: un *bug* en ML puede vivir en el *dataset,* no en el código. El modelo puede aprender atajos espurios (correlaciones que funcionan en entrenamiento pero fallan en producción) y tu código compilará perfectamente mientras el modelo toma decisiones incorrectas.

El pipeline: de los datos a producción

Un modelo útil no nace cuando termina el entrenamiento. Nace cuando puedes repetir el proceso, comparar versiones y desplegar con control:

```
Decisión → Datos → Etiquetas → Entrenamiento → Evaluación → Registro → Despliegue →
Monitorización
```

Y no es una línea recta, sino un **ciclo**. La última fase, la monitorización, vigila el modelo en producción y, cuando detecta que algo ha cambiado (que los datos del mundo real se han desviado de los de entrenamiento, lo que se llama *drift*), reabre el proceso: vuelves a los datos, reentrenas y despliegas una versión nueva. Un sistema de ML maduro no es un modelo, es esta rueda girando con control en cada vuelta.



Cada fase produce artefactos que deberías versionar, porque sin versionar no puedes ni reproducir un resultado ni comparar dos versiones:

FASE	ARTEFACTO	PREGUNTA CLAVE
Decisión	Problema y métrica objetivo escritos	¿Qué resuelvo y cómo sabré que funciona?
Datos	*Dataset* versionado	¿Con qué datos exactos se entrenó?
Etiquetas	Etiquetas con su calidad medida	¿Son fiables las respuestas que aprende el modelo?
Entrenamiento	Código, semilla, hiperparámetros	¿Puedo reproducir el modelo?
Evaluación	Informe con métricas y segmentos	¿Dónde mejora y dónde empeora?
Registro	Versión del modelo guardada y etiquetada	¿Qué versión exacta está en cada entorno?
Despliegue	Configuración de *runtime*	¿Qué latencia, coste y *fallback* tendrá?
Monitorización	Métricas y alertas	¿Cómo sé que en producción ha cambiado algo?

El criterio para desplegar no es «el modelo es mejor». Es una conjunción: solo se despliega si se cumplen las cuatro condiciones a la vez.

```
release_ok = data_ok AND eval_ok AND safety_ok AND rollback_ok
```

- **`data_ok`**: los datos de entrenamiento son los que crees, están versionados y no tienen fugas ni contaminación.
- **`eval_ok`**: el modelo mejora en la evaluación, y no solo en la media, también en los segmentos que importan.
- **`safety_ok`**: pasa las comprobaciones de seguridad y de comportamiento que tu caso exige.
- **`rollback_ok`**: puedes volver a la versión anterior en minutos si algo sale mal.

Fíjate en el `AND`: que el modelo puntúe mejor (`eval_ok`) no basta. Si no puedes volver atrás (`rollback_ok`), no despliegues, por muy bueno que parezca el resultado.

Entrenar no es desplegar

El modelo que gana en métricas *offline* puede ser inviable en producción. Quizás tarda demasiado, consume demasiada memoria, no explica sus fallos o no se puede monitorizar. Desplegar no es copiar un archivo: es integrar un sistema.

Cinco técnicas cierran la brecha entre el modelo entrenado y el servicio desplegado:

TÉCNICA	QUÉ HACE	CUÁNDO AYUDA	CONTRAPARTIDA
Pruning	Elimina pesos o neuronas poco útiles	Reducir tamaño cuando hay redundancia	Puede degradar calidad
Cuantización	Guarda pesos con menos bits (FP16→INT8→INT4)	Inferencia local, *edge*	Puede perder precisión
Destilación	Un modelo grande enseña a uno pequeño	Casos repetitivos de bajo coste	El alumno hereda sesgos del maestro
Compilación	Optimiza para *hardware* concreto	Servir mucho tráfico con GPU/TPU específica	Más dependencia del *runtime*
Batching	Agrupa peticiones	*Throughput* alto	Puede aumentar latencia individual

Cuantización: modelos que caben en tu portátil

La cuantización merece un apartado propio porque es la técnica que ha democratizado el acceso a los LLMs.[4]

La idea es simple: en lugar de guardar cada peso como un número de 16 bits (FP16), lo guardas con 8 bits (INT8), 4 bits (INT4) o incluso 2 bits. Ocupa la mitad, un cuarto o un octavo de memoria. Un modelo de 7B parámetros que en FP16 ocupa ~14 GB, cuantizado a 4 bits ocupa ~3,5 GB. Cabe en un portátil.

La cuenta es directa, y es la primera que harás para saber si un modelo cabe en tu GPU:

EJEMPLO, NO NOTACIÓN OFICIAL.

$$\text{memoria de pesos} \approx P \times b$$

> *No es una fórmula con nombre del campo, sino la cuenta directa de cuánto ocupan los pesos: número de parámetros por los bytes de cada uno. Y es solo una cota inferior: la memoria real de servir el modelo es mayor, porque suma la KV cache, los estados intermedios y el overhead del runtime, como se ve unas líneas más abajo.*

SÍMBOLO	SIGNIFICADO	EJEMPLO
P	Número de parámetros del modelo	7.000 millones (7B)
b	Bytes por parámetro según la precisión	FP16 = 2, INT8 = 1, INT4 = 0,5
memoria de pesos	Lo que ocupan los pesos en memoria	$7 \times 10^9 \times 2 = 14$ GB en FP16

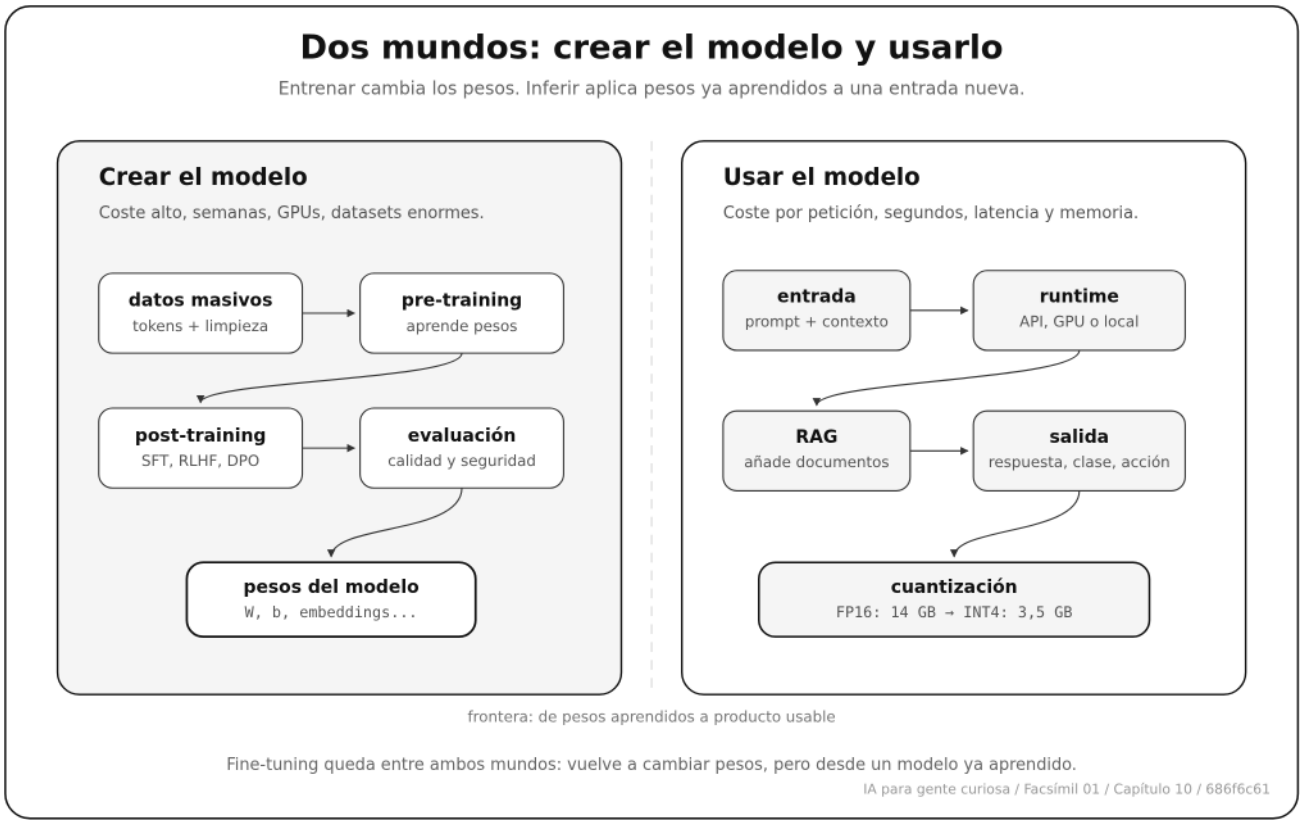
En palabras: multiplicas el número de parámetros por los bytes de cada uno. Un modelo de 7B en FP16 son 7.000 millones × 2 bytes = 14 GB; a 4 bits (medio byte) bajan a 3,5 GB.

Pero los pesos no son lo único que ocupa memoria. Al generar texto, el modelo guarda en una **KV cache** las claves (*keys*) y los valores (*values*) de la atención de todos los tokens ya procesados, para no recalcularlos en cada paso. Su tamaño crece con la longitud del contexto: cuanto más largo son el prompt y la respuesta, más memoria consume, al margen del tamaño de los pesos. Por eso un modelo puede caber con un prompt corto y dejar de caber cuando lo usas con documentos largos: la cuantización reduce los pesos, pero la KV cache sigue ahí. Por eso conviene calcular las dos memorias por separado.

La contrapartida de la cuantización es la precisión. Al reducir el número de bits, algunos pesos pierden matices. Para la mayoría de tareas, la pérdida puede ser pequeña. Para tareas que requieren cálculo preciso, seguimiento largo de instrucciones o respuestas muy sensibles al detalle, la degradación puede ser notable.[5] La clave está en evaluar con tus datos y tu tarea.

Formatos comunes:

- **GGUF**: para inferencia en CPU con llama.cpp. Ideal para portátiles.
- **GPTQ / AWQ**: para inferencia en GPU con menor latencia.
- **BitsAndBytes**: para cargar modelos cuantizados en Python con Hugging Face.



En el día a día

No entrenas, infieres. Tu día a día con IA es casi todo inferencia: llamadas a una API, *prompts* y respuestas. El entrenamiento de los modelos grandes lo hacen los proveedores, con sus clústeres y sus datos. Saber esto cambia cómo planteas un problema: antes de pensar en «entrenar un modelo», la pregunta correcta suele ser «¿qué le pido al modelo que ya existe y cómo le doy el contexto que necesita?».

Haces *fine-tuning* cuando necesitas especialización, no conocimiento nuevo. Si el modelo base no domina tu jerga, tu formato o tu estilo, el *fine-tuning* (a menudo con LoRA) es la herramienta. Pero antes evalúa si RAG resuelve tu problema sin el coste de reentrenar: si lo que falta es información que cambia (catálogo, normativa, documentos), RAG suele ser la respuesta y el *fine-tuning* no. Confundir las dos cosas es el error más caro de esta fase.

Cuantizas para abaratar el despliegue. Si usas modelos *open weights*, la cuantización te permite ejecutarlos en hardware más modesto: pasar de 14 GB a 3,5 GB es la diferencia entre necesitar una GPU de 24 GB y poder usar una de 8 GB, o incluso correr en CPU. La decisión no es automática: cuantizas, evalúas con tu tarea y compruebas si la pérdida de precisión es asumible. Y recuerda que la KV cache del contexto largo no se cuantiza con los pesos.

Versionas todo. El *dataset*, el código de entrenamiento, los hiperparámetros y la configuración de despliegue. Parece burocracia, pero es lo que distingue un experimento reproducible de una corazonada: si no puedes reconstruir exactamente el modelo que está en

producción, tampoco podrás depurarlo cuando falle.

Por qué debería importarte

La frontera entre entrenamiento e inferencia es la frontera entre lo que haces tú y lo que hace el proveedor. Si no la entiendes, tomarás decisiones equivocadas: intentarás «entrenar» un modelo cuando bastaba con un buen *prompt*, o asumirás que el modelo «sabe» cosas que solo aprendió durante el entrenamiento y no puede actualizar.

Entender el *pipeline* completo, de los datos a la monitorización, es lo que separa un prototipo que funciona en una *demo* de un sistema que funciona en producción. Y la cuantización es la herramienta que convierte «necesito un clúster de GPUs» en «cabe en mi portátil».

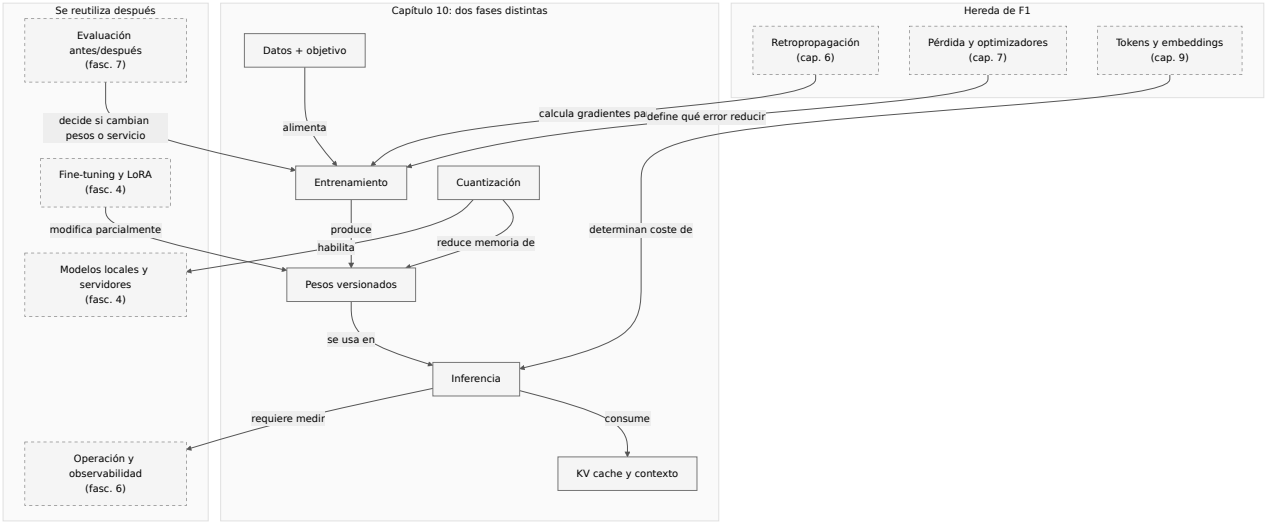
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Confundir *fine-tuning* con «enseñar datos nuevos»	El *fine-tuning* ajusta el estilo y comportamiento, no inyecta conocimiento factual actualizable. Para datos cambiantes, usa RAG.	Si tus datos cambian cada día, RAG. Si necesitas que el modelo hable con tu jerga, *fine-tuning*.
Cuantizar sin evaluar	Pasar de FP16 a INT4 puede degradar el rendimiento en tareas que requieren precisión. Asumir que «no se nota» es peligroso.	Evalúa con tus datos y tu tarea antes y después de cuantizar. No delegues esta decisión.
Desplegar sin *rollback*	Si el modelo nuevo funciona peor en producción, necesitas volver al anterior en minutos, no en días.	Diseña el despliegue con *rollback* desde el primer día. No es opcional.
Tratar el modelo como una caja negra	Si no sabes con qué datos se entrenó, no puedes explicar sus sesgos. Si no versionas el *pipeline*, no puedes reproducir resultados.	Versiona datos, código, hiperparámetros y configuración. La trazabilidad no es burocracia: es ingeniería.

Cómo encaja todo

Este mapa separa tres decisiones que suelen mezclarse. Primero, de dónde viene el modelo: entrenamiento, post-entrenamiento o ajuste parcial. Segundo, cómo se usa: inferencia con una ventana de contexto, una KV cache y una política de servicio. Tercero, cómo se lleva a producción: presupuesto, cuantización, despliegue y observabilidad.

La herencia viene de backpropagation, pérdida y optimizadores: esas piezas explican cómo cambian los pesos. La reutilización aparece en los próximos facsímiles cuando elijas entre API, modelo local, RAG, fine-tuning o despliegue propio. Si no separas estas capas, confundirás “necesito más conocimiento” con “necesito entrenar” y gastarás tiempo en el sitio equivocado.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Entrenamiento	Proceso de ajustar parámetros con datos y una función de pérdida. Puede ir desde una práctica pequeña hasta pre-entrenamiento de frontera.
Inferencia	Uso de un modelo ya entrenado para generar o calcular salidas a partir de entradas nuevas.
Fine-tuning	Reentrenamiento parcial de un modelo existente con datos propios para especializarlo.
Cuantización	Reducción de la precisión numérica de los pesos para ahorrar memoria y acelerar inferencia.
Pipeline ML	Secuencia de fases desde la decisión inicial hasta la monitorización en producción.
Pre-entrenamiento	Primera fase del entrenamiento de un LLM: aprender lenguaje prediciendo el siguiente token sobre billones de tokens. Consume casi todo el coste.
Post-entrenamiento (SFT)	Ajuste con ejemplos de conversaciones de alta calidad para que el modelo siga instrucciones y mantenga un tono conversacional.
Alineamiento (RLHF/DPO)	Fase que refina el comportamiento con preferencias humanas: ser útil, admitir lo que no sabe y rechazar lo inapropiado.
KV cache	Memoria donde el modelo guarda las claves y los valores de la atención de los tokens ya vistos durante la inferencia. Crece con la longitud del contexto.
LoRA	Técnica de *fine-tuning* eficiente: congela los pesos del modelo y entrena solo unas matrices pequeñas de bajo rango.

Antes de pasar página

- [ ] ¿Puedo explicar las diferencias entre entrenamiento, inferencia y *fine-tuning*? (Si no, vuelve a «Entrenamiento vs inferencia».)
- [ ] ¿Entiendo la diferencia fundamental entre programación clásica y ML? (Si no, vuelve a «Programación clásica vs machine learning».)
- [ ] ¿Puedo enumerar las fases del *pipeline* ML? (Si no, vuelve a «El pipeline».)
- [ ] ¿Sé qué es la cuantización y cuándo usarla? (Si no, vuelve a «Cuantización».)

- [ ] ¿Sé explicar por qué la KV cache también consume memoria? (Si no, vuelve a «Cuantización: modelos que caben en tu portátil».)

En resumen

IDEA FUERZA	DETALLE
Entrenar cambia pesos; inferir usa pesos.	Pre-entrenar un modelo fundacional puede costar millones y tardar meses; entrenar modelos pequeños o ajustar adaptadores puede ser mucho más accesible. La inferencia también escala: puede ser una llamada barata o un sistema complejo de servicio.
El ML no sustituye al *software*: cambia dónde vive la lógica.	De reglas escritas por personas a patrones aprendidos de datos. El *bug* puede estar en el *dataset*, no en el código.
La cuantización democratiza el acceso a los LLMs.	De 14 GB a 3,5 GB con 4 bits. Evalúa siempre antes de desplegar: la pérdida de precisión depende de tu tarea.

Para saber más

Brown, T. B. y otros (2020). Language models are few-shot learners. En *Advances in Neural Information Processing Systems 33* (pp. 1877-1901). https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

Dettmers, T., Pagnoni, A., Holtzman, A. y Zettlemoyer, L. (2023). QLoRA: efficient finetuning of quantized LLMs. En *Advances in Neural Information Processing Systems 36*. https://arxiv.org/abs/2305.14314

Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org

Jacob, B. y otros (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. En *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2704-2713). https://doi.org/10.1109/CVPR.2018.00286

Kaplan, J. y otros (2020). Scaling laws for neural language models. arXiv:2001.08361. https://doi.org/10.48550/arXiv.2001.08361

Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. y Polosukhin, I. (2017). Attention is all you need. En *Advances in Neural Information Processing Systems 30* (pp. 5998-6008). https://papers.nips.cc/paper/7181-attention-is-all-you-need

---

Notas

1. Kaplan, J. y otros (2020). Scaling laws for neural language models. arXiv:2001.08361. https://doi.org/10.48550/arXiv.2001.08361. Los autores establecieron las leyes de escala que relacionan el tamaño del modelo, los datos de entrenamiento y la computación necesaria, demostrando que el rendimiento mejora de forma predecible con la inversión. ↩

2. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org. El capítulo 12.3 aborda las diferencias prácticas entre la fase de entrenamiento y la de inferencia, incluyendo técnicas de optimización específicas para cada una. ↩

3. Brown, T. B. y otros (2020). Language models are few-shot learners. En *Advances in Neural Information Processing Systems 33* (pp. 1877-1901). https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html. GPT-3 demostró que el pre-entrenamiento a escala masiva produce modelos capaces de realizar tareas para las que no fueron explícitamente entrenados (*few-shot learning*). ↩

4. Jacob, B. y otros (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. En *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2704-2713). https://doi.org/10.1109/CVPR.2018.00286. Los autores establecieron el marco para la cuantización de redes neuronales, demostrando que es posible mantener la precisión reduciendo la representación numérica de 32 a 8 bits. ↩

5. Dettmers, T., Pagnoni, A., Holtzman, A. y Zettlemoyer, L. (2023). QLoRA: efficient finetuning of quantized LLMs. En *Advances in Neural Information Processing Systems 36*. https://arxiv.org/abs/2305.14314. QLoRA combina cuantización de 4 bits con adaptadores LoRA, permitiendo hacer *fine-tuning* de modelos de 65B parámetros en una sola GPU de 48 GB. ↩

Cuadernos para practicar

Escanea el código con el móvil para abrir el cuaderno en Google Colab y ejecutarlo sin instalar nada.

Entrenamiento vs inferencia: el sobreajuste, en directo

https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%C3%ADmil%2001/03-sobreajuste-en-directo.ipynb