

Facsímil 03

Arquitecturas y modelos

Coleccionable completo · 8 capítulos

686f6c61 · 2026

Índice

01	Qué es un LLM: modelo, parámetros y escala
02	Transformer por dentro: de texto a tensores y atención
03	Q, K, V, máscara causal y softmax
04	MLP, residual, LayerNorm, logits y sampling
05	Arquitecturas modernas: familias, MoE, razonamiento y multimodalidad
06	Transfer learning, destilación y modelos abiertos
07	Inferencia optimizada, edge AI y hardware
08	Lo que deberías saber: arquitecturas y modelos

Capítulo 01: Qué es un LLM: modelo, parámetros y escala

El objeto que hay detrás del chat

Cuando alguien dice «he usado una IA», casi nunca se refiere al sistema completo. Se refiere a una experiencia: una caja de texto, una respuesta, quizá una voz o una imagen. Pero debajo de esa experiencia hay varias piezas: una interfaz, una API, reglas de producto, herramientas, bases de datos, filtros, cachés y, en el centro, un **modelo**.

Este capítulo va de ese centro. No del producto de chat. No del proveedor. No del envoltorio. Del objeto técnico que hace posible que escribas una frase y recibas otra frase: un **LLM**, un *Large Language Model*, o modelo de lenguaje de gran escala.

La pregunta no es solo «qué es». La pregunta buena es: ¿qué tipo de máquina matemática estoy usando?, ¿qué significan sus parámetros?, ¿por qué la escala cambia tanto el comportamiento?, ¿qué puedo esperar de él y qué no debería pedirle sin apoyo externo?

Qué no es un LLM

En el [facsimil 1, capítulo 1](#) vimos qué no es la inteligencia artificial en general: ni magia, ni una mente consciente, ni un buscador glorificado. No vamos a repetir aquello. Ahora afinamos el foco a un tipo concreto de modelo, el LLM, y deshacemos tres confusiones más técnicas, las que aparecen justo cuando empiezas a construir con uno.

Un LLM no es una base de datos. Puede devolver datos que parecen almacenados, pero su mecanismo no consiste en buscar una fila exacta y traerla de vuelta. Si necesitas el saldo de una cuenta, el precio de un pedido o el estado real de un envío, necesitas consultar el sistema que contiene ese dato. El LLM puede ayudar a formular, resumir o explicar; no sustituye a la fuente de verdad.

Tampoco es un programa clásico lleno de reglas escritas a mano. Nadie ha codificado millones de condiciones del tipo «si el usuario pregunta por una receta, responde así». El comportamiento emerge de muchos números ajustados durante entrenamiento, no de una lista legible de instrucciones.

Y, aunque produzca lenguaje muy convincente, un LLM no es una persona que recuerda, razona y decide con intención. La forma honesta de pensarlo es más sobria: recibe tokens, los transforma en vectores, aplica muchas capas de álgebra lineal y devuelve una distribución de probabilidad sobre el siguiente token. Esa frase suena menos espectacular, pero es justo lo que necesitamos para poder construir con criterio.

Qué sí es un LLM

Un LLM es un modelo probabilístico que aprende patrones del lenguaje a gran escala. Su tarea base es sencilla de enunciar: dado un contexto, estimar qué token viene después.¹

Si el contexto es:

La capital de Francia es

el modelo asigna puntuaciones a muchos tokens posibles. «París» debería recibir una puntuación alta. «azul» debería recibir una puntuación baja. «porque» quizá tenga una puntuación pequeña pero no imposible, dependiendo del contexto.

La idea moderna no empieza con los LLM actuales. Los modelos neuronales de lenguaje ya buscaban representar palabras como vectores y estimar probabilidades de secuencias antes de la explosión generativa reciente.² Lo que cambió después fue la arquitectura, los datos, el cómputo y la escala.

Por eso un LLM no es «una IA» en abstracto. Es una familia concreta de modelos: redes neuronales profundas entrenadas sobre enormes cantidades de texto y, en modelos multimodales, también otros tipos de datos. En este facsímil abriremos esa familia pieza a pieza.

Un poco de historia: cuándo empezó eso de «LLM»

Aquí hay una pequeña trampa histórica. Si preguntas «cuál fue el primer LLM», la respuesta depende de qué estés llamando LLM.

La expresión **large language model** aparece en la literatura bastante antes de ChatGPT. Un ejemplo temprano y muy claro es el paper de Google “**Large Language Models in Machine Translation**”, publicado en 2007, donde Brants y sus coautores hablaban de modelos de lenguaje enormes para mejorar traducción automática.³ Pero aquellos modelos no eran lo que hoy imaginas cuando dices LLM. Eran modelos estadísticos grandes, muy útiles, pero no redes Transformer generativas de propósito general.

La familia moderna se entiende mejor como una cadena:

AÑO	HITO	POR QUÉ IMPORTA
1948	Shannon modela el lenguaje como una fuente estadística.	Abre la puerta a pensar el texto como secuencia de símbolos con probabilidades.
2003	Bengio y coautores proponen un modelo neuronal probabilístico de lenguaje.	Las palabras empiezan a representarse como vectores aprendidos, no solo como cuentas discretas.
2017	Transformer.	La atención permite entrenar modelos grandes de secuencias de forma mucho más paralela. ⁴
2018	GPT-1.	OpenAI muestra que el preentrenamiento generativo seguido de adaptación a tareas funciona muy bien. ⁵
2019	GPT-2.	La escala empieza a enseñar una idea fuerte: un modelo de lenguaje puede comportarse como aprendiz multitarea sin entrenarlo de nuevo para cada tarea. ⁶
2020	GPT-3.	Con 175B parámetros, el paper populariza la idea de modelos capaces de hacer tareas con ejemplos en el propio prompt. ⁷
2022	Modelos instruidos y chat.	El post-entrenamiento para seguir instrucciones convierte modelos generativos en asistentes mucho más útiles.

Así que, si somos precisos: **el término «large language model» ya se usaba en papers antes de los Transformers modernos.** Pero **el LLM moderno que solemos tener en la cabeza** nace de la combinación de Transformer, preentrenamiento generativo, escala, post-entrenamiento e interfaz conversacional.

Tres capas que conviene no mezclar

Para entender bien un LLM hay que separar tres niveles que en una app aparecen pegados. Si no los separamos, acabamos discutiendo frases como «este modelo es educado», «este modelo sabe usar herramientas» o «este modelo recuerda mi proyecto» sin saber qué parte del sistema produce cada comportamiento.

Modelo base. Es el modelo entrenado sobre grandes cantidades de texto para predecir el siguiente token. Aprende regularidades del lenguaje, código, formatos, estilos, relaciones estadísticas y patrones de razonamiento presentes en los datos. Pero un modelo base no tiene por qué comportarse como un asistente amable. Si le escribes una pregunta, puede continuar la pregunta, imitar un documento, cerrar una lista o producir algo extraño. Su entrenamiento principal no dice «ayuda al usuario»: dice «predice bien la continuación».

Modelo instruido. Después del preentrenamiento, muchos modelos pasan por una fase de ajuste para seguir instrucciones, responder en formato conversacional, rechazar peticiones que no debe completar y preferir respuestas útiles según ejemplos humanos o sintéticos.⁸ Esta capa explica por qué un modelo puede contestar «claro, te ayudo» en vez de limitarse a completar texto.

Sistema completo. Es lo que tú usas en producción: modelo, prompt de sistema, contexto recuperado, herramientas, validadores, memoria conversacional, permisos, logs, interfaz, evaluaciones y reglas de negocio. Cuando un asistente consulta una base de datos o llama a una función, eso no significa que el LLM sea una base de datos. Significa que el sistema que rodea al LLM le ha dado una herramienta y ha decidido cómo usar su salida.

NIVEL	QUÉ APRENDE O CONTIENE	QUÉ NO CONVIENE ATRIBUIRLE
Modelo base	Patrones estadísticos del lenguaje y del código.	Obediencia conversacional fiable o conocimiento actualizado.
Modelo instruido	Preferencias de respuesta, formato, seguimiento de instrucciones y estilo de ayuda.	Acceso real a tus datos, herramientas o estado de la aplicación.
Sistema completo	Contexto externo, permisos, herramientas, validadores, trazas y experiencia de usuario.	Que todo el comportamiento venga «de dentro» del modelo.

Esta separación volverá mucho. En el [facsimil 4](#), cuando hablemos de APIs y RAG, estaremos construyendo parte del sistema completo. En el [facsimil 5](#), cuando hablemos de agentes, estaremos añadiendo decisiones, herramientas y memoria alrededor del modelo. En el [facsimil 7](#), cuando evaluemos, tendremos que medir cada capa por separado: no es lo mismo fallar por un mal modelo que fallar por un mal contexto o por una herramienta mal diseñada.

La arquitectura interna de un LLM base

El LLM base típico de la familia GPT es un **Transformer decoder-only autoregresivo**. Suena largo, pero se puede leer despacio:

Transformer porque usa bloques de atención, MLP, conexiones residuales y normalización.

Decoder-only porque usa la parte generativa de la arquitectura Transformer: recibe una secuencia previa y genera continuación.

Autoregresivo porque predice el siguiente token usando tokens anteriores, añade ese token al contexto y repite el proceso.

La arquitectura interna mínima es esta:

PIEZA	QUÉ HACE	QUÉ DEBES RECORDAR
Tokenizador	Convierte texto en IDs de token.	El modelo no recibe palabras; recibe enteros.
Embedding de token	Convierte cada ID en un vector.	Aquí el texto entra al espacio numérico.
Información de posición	Indica orden o distancia entre tokens.	Sin posición, la atención no sabe qué va antes o después.
Bloques Transformer	Repiten atención, MLP, residual y normalización.	Aquí se transforma la representación capa a capa.
Atención causal	Permite mirar hacia atrás, no hacia tokens futuros.	Hace honesta la predicción de izquierda a derecha.
MLP	Reprocesa cada token después de mezclar contexto.	No todo ocurre en atención; el MLP aporta mucha capacidad.
Cabeza de lenguaje	Proyecta el último vector al tamaño del vocabulario.	De aquí salen los logits.
Softmax y muestreo	Convierte logits en probabilidades y elige token.	Aquí aparecen temperature, top_p , top_k y decisiones de generación.

El flujo completo, dicho como sistema, sería:

```
texto -> tokens -> embeddings + posición -> bloques Transformer -> logits -> softmax -> siguiente token
```

En el capítulo 02 abriremos el tramo tokens -> embeddings -> tensores -> atención . En el capítulo 03 nos detendremos en Q, K, V y máscara causal. En el capítulo 04 veremos lo que pasa entre atenciones y cómo los logits se convierten en texto. Este capítulo solo coloca el mapa.

Dónde se encuentran los LLMs

Un LLM no vive solo en una web de chat. Puede aparecer en varios sitios, con formas muy distintas:

LUGAR	QUÉ SIGNIFICA	EJEMPLO DE USO
API de proveedor	El modelo vive en infraestructura externa y tú lo llamas por red.	Chat, extracción, clasificación, generación de código, análisis de documentos.
Producto final	El usuario no ve el modelo; ve una función integrada.	Asistente en un editor, buscador mejorado, soporte interno, copiloto de datos.
Repositorio de pesos abiertos	Descargas pesos o variantes cuantizadas y los sirves tú.	Prototipos locales, investigación, privacidad, control de coste.
Runtime local	Un programa carga el modelo en tu máquina o servidor.	Ollama, LM Studio, Transformers, vLLM, SGLang o TensorRT-LLM, según caso.
Dispositivo o edge	Modelo pequeño o cuantizado ejecutado cerca del usuario.	Móvil, portátil, navegador, dispositivo industrial, app sin conexión estable.
Sistema de agentes	El LLM decide o ayuda a decidir pasos dentro de un bucle con herramientas.	Revisar un repositorio, consultar documentos, preparar tareas, ejecutar flujos internos.

La diferencia práctica es enorme. Un modelo por API puede darte calidad alta sin ocuparte del hardware, pero dependes de red, coste por uso y condiciones del proveedor. Un modelo local te da control y privacidad, pero te obliga a pelear con memoria, cuantización, runtime y actualizaciones. Un modelo dentro de un producto puede parecer «inteligente» porque el sistema le da contexto y herramientas, no porque el modelo por sí solo lo sepa todo.

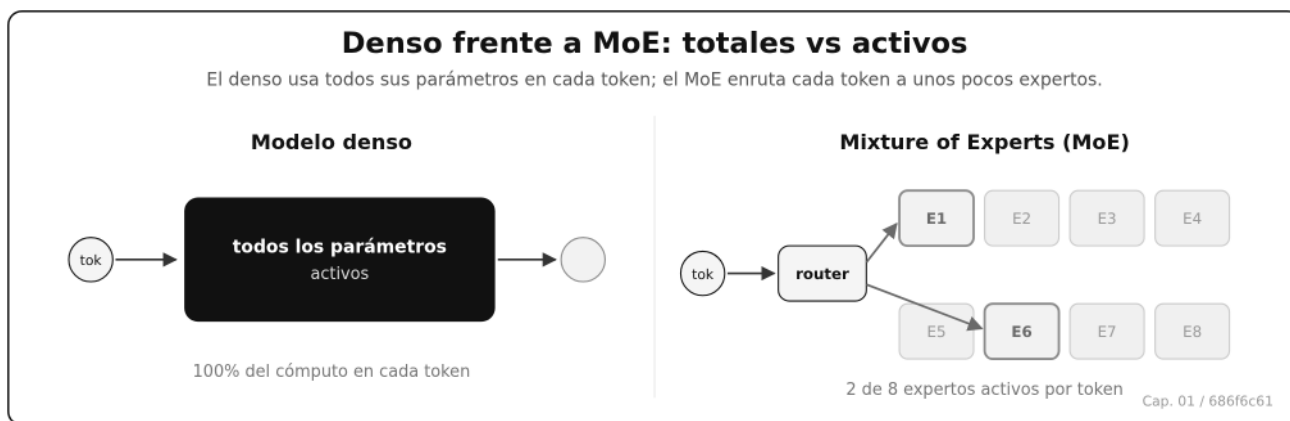
Qué tipos de LLM podrías encontrarte

No todos los LLMs son iguales. Algunos nombres describen arquitectura, otros describen acceso, otros describen entrenamiento y otros describen despliegue. Mezclarlos es una fuente deliciosa de confusión.

TIPO	QUÉ SIGNIFICA	PREGUNTA ÚTIL
Base	Entrenado para continuar texto.	¿Necesito adaptarlo o usar una versión instruida?
Instruct o chat	Ajustado para seguir instrucciones y conversar.	¿Sigue bien formato, tono, límites y herramientas?
Denso	Activa casi todos sus parámetros relevantes por token.	¿Cuánto cuesta cada token frente a un MoE?
MoE	Tiene expertos y activa solo algunos por token.	¿Cuántos parámetros son totales y cuántos activos?
Text-only	Entrada y salida principalmente texto.	¿Mi caso necesita imagen, audio, vídeo o documentos complejos?
Multimodal	Acepta o genera más de un tipo de dato.	¿Qué modalidades acepta y cuáles produce realmente?
Open weights	Los pesos están disponibles bajo una licencia.	¿Puedo servirlo, modificarlo o usarlo comercialmente?
Propietario cerrado	Se usa por API o producto, sin acceso a pesos.	¿Me compensa calidad y mantenimiento frente a dependencia externa?
Modelo pequeño	Optimizado para coste, latencia o dispositivo.	¿La tarea requiere un modelo grande o solo uno fiable y barato?
Modelo de razonamiento	Dedica más presupuesto a pasos internos o deliberación.	¿La mejora compensa coste y latencia en mi evaluación?

Una misma familia puede ocupar varias filas a la vez: puede ser `instruct` , `MoE` , `multimodal` , `open weights` y además tener variantes cuantizadas para local. Por eso conviene leer las fichas con calma. El nombre comercial rara vez basta.

Dos de esas filas merecen una imagen, porque marcan el coste de cada token. Un **modelo denso** activa casi todos sus parámetros para procesar cada token: si tiene 70B, los 70B trabajan en cada paso. Un **MoE** (*mixture of experts*) tiene muchos expertos, pero un *router* envía cada token solo a unos pocos: puede tener 600B totales y activar 30B por token. Por eso un MoE puede ser barato de ejecutar pese a ser enorme, y por eso al leer una ficha conviene distinguir parámetros **totales** de parámetros **activos**.



El mecanismo matemático mínimo

En el facsímil 1 vimos la predicción del siguiente token como tres pasos intuitivos: tokenizar, transformar con pesos aprendidos y muestrear. Aquí no nos quedamos en la intuición: le ponemos la notación que hay debajo, porque el resto del fascículo (atención, sampling, MoE, cuantización) se apoya en ella.

Un modelo de lenguaje autoregresivo factoriza una secuencia token a token. Si tenemos una secuencia:

$$x_1, x_2, \dots, x_T$$

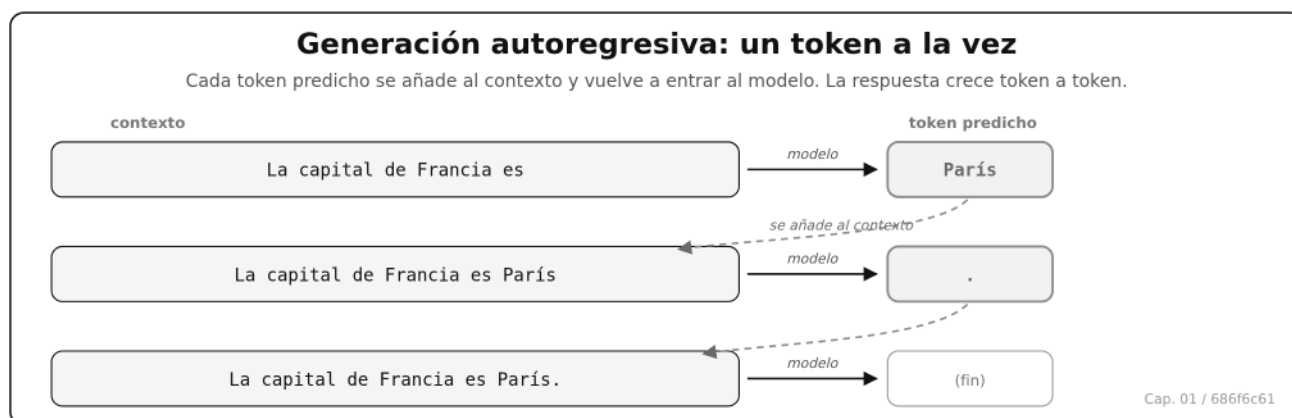
el modelo estima la probabilidad conjunta así:

$$P(x_1, x_2, \dots, x_T) = \prod_{t=1}^T P(x_t \mid x_{<t}; \theta)$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
x_t	Token en la posición t .	En «la casa azul», x_2 podría ser «casa».
$x_{<t}$	Todos los tokens anteriores a la posición t .	Para predecir «azul», el contexto previo es «la casa».
T	Longitud total de la secuencia.	Una frase de 12 tokens tiene $T = 12$.
θ	Conjunto de parámetros aprendidos del modelo.	Los miles de millones de pesos de una red.
$P(x_t x_{<t}; \theta)$	Probabilidad de un token dado el contexto anterior y los parámetros.	$P(\text{azul} \text{la casa})$.

La expresión puede intimidar, pero dice algo cotidiano: para escribir una frase, el modelo predice el primer token, luego el segundo condicionado al primero, luego el tercero condicionado a los dos anteriores, y así hasta terminar. La respuesta larga aparece porque este paso se repite muchas veces.

El siguiente diagrama lo hace visible: cada token que el modelo elige se **añade** al contexto y se vuelve a alimentar a la entrada. No hay un plan global de la frase entera; hay un mismo paso repetido, cada vez con un poco más de contexto.



Durante entrenamiento, el objetivo típico es minimizar la sorpresa del modelo ante el siguiente token correcto. Una forma de escribirlo es la **pérdida de entropía cruzada negativa**:

$$\mathcal{L}(\theta) = - \sum_{t=1}^T \log P_{\theta}(x_t | x_{<t})$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
$\mathcal{L}(\theta)$	Pérdida que queremos reducir durante entrenamiento.	Cuanto menor, mejor predice el texto de entrenamiento.
$\log$	Logaritmo de la probabilidad.	Penaliza mucho asignar probabilidad muy baja al token correcto.
P_θ	Probabilidad producida por el modelo con parámetros θ .	Si el modelo da 0,80 al token correcto, la pérdida baja.
$\sum_{t=1}^T$	Suma sobre todas las posiciones de la secuencia.	Se evalúa token por token.

Ejemplo pequeño. Imagina que el texto correcto es:

la casa azul

y el modelo asigna estas probabilidades a los tokens correctos:

PASO	CONTEXTO PREVIO	TOKEN CORRECTO	PROBABILIDAD ASIGNADA
1	Inicio	la	0,50
2	la	casa	0,25
3	la casa	azul	0,80

La pérdida sería:

$$\mathcal{L} = -(\log 0,50 + \log 0,25 + \log 0,80)$$

Con logaritmo natural:

$$\mathcal{L} \approx -(-0,693 - 1,386 - 0,223) = 2,302$$

Si después de entrenar el modelo asigna 0,80, 0,60 y 0,95 a esos mismos tokens, la pérdida baja. Eso es aprender: ajustar parámetros para que el texto observado resulte menos sorprendente.

De la pérdida a la perplexity

La pérdida es cómoda para entrenar, pero poco intuitiva para interpretar. Por eso se suele traducir a **perplexity**, que es la exponencial de la pérdida media por token:

$$\text{PPL} = e^{\mathcal{L}/T}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
PPL	Perplejidad: opciones efectivas por token.	$\approx 2,15$ en el ejemplo.
$\mathcal{L}/T$	Pérdida media por token.	$2,302/3 \approx 0,767$.
e	Exponencial, deshace el logaritmo de la pérdida.	$e^{0,767}$.

La perplexity tiene una lectura muy visual: es el **número efectivo de opciones** entre las que el modelo duda en cada paso. Una perplexity de 1 es certeza total (siempre acierta con probabilidad 1); una de 50 significa que el modelo está, en promedio, tan indeciso como si eligiera entre 50 tokens equiprobables. En nuestro ejemplo la pérdida media es 0,767, así que la perplexity es $e^{0,767} \approx 2,15$: el modelo duda como si tuviera poco más de dos opciones por token.

Esto conecta con una idea más profunda. Minimizar la entropía cruzada es minimizar la **sorpresa** en el sentido de Shannon: cada $-\log P$ mide la información (en *bits* si el logaritmo es en base 2, en *nats* si es natural) que cuesta codificar el token correcto. Un modelo que predice bien necesita pocos bits para describir el texto real; uno malo se sorprende mucho y gasta muchos. Entrenar es, literalmente, enseñar al modelo a sorprenderse menos.

De logits a probabilidades

El modelo no empieza devolviendo probabilidades limpias. Primero produce **logits**: puntuaciones sin normalizar, una por cada token del vocabulario. Después usamos softmax para convertir esas puntuaciones en probabilidades:

$$P(x_i \mid c) = \frac{e^{z_i}}{\sum_{j=1}^V e^{z_j}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x_i	Token candidato número i .	«París», «Madrid», «azul»...
c	Contexto que recibe el modelo.	«La capital de Francia es».
z_i	Logit del token x_i .	$z_{\text{París}} = 5,1$.
V	Tamaño del vocabulario del modelo.	50 000, 100 000 o más tokens posibles.
e^{z_i}	Exponencial del logit.	Convierte puntuaciones en valores positivos.

Supongamos tres candidatos:

TOKEN	LOGIT
París	5,0
Lyon	2,0
azul	0,0

Aplicamos softmax:

$$P(\text{París}) = \frac{e^5}{e^5 + e^2 + e^0} \approx \frac{148,41}{156,80} \approx 0,947$$

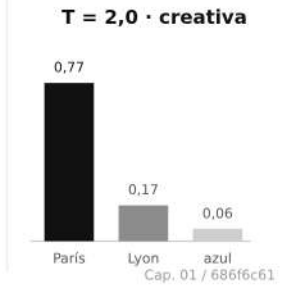
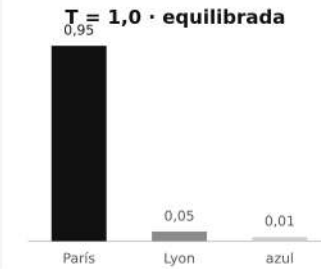
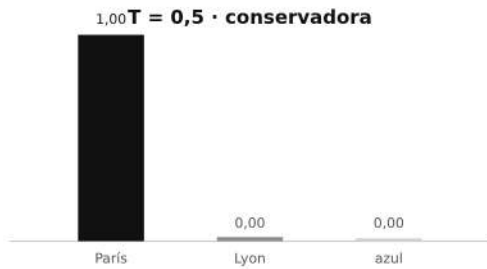
$$P(\text{Lyon}) \approx 0,047, \quad P(\text{azul}) \approx 0,006$$

El modelo no «elige la verdad». Calcula una distribución. Luego el sistema de generación decide si toma el token más probable, si muestrea con temperatura, si restringe candidatos con `top_p`, o si aplica alguna regla adicional. Volveremos a esto en el capítulo 04.

Esa decisión de generación tiene una palanca central: la **temperatura**. Antes del softmax, los logits se dividen por un número T . Con T baja la distribución se vuelve picuda (el modelo casi siempre elige el token más probable, salidas deterministas); con T alta se aplanan (reparte probabilidad, salidas más variadas y creativas, pero también más arriesgadas). Los mismos logits, distinta temperatura:

La temperatura moldea la distribución

Los mismos logits (París 5, Lyon 2, azul 0) dan distribuciones distintas según la temperatura T.



Baja T concentra (determinista); alta T reparte (variada, pero más arriesgada).

Anatomía de un LLM en una aplicación real

El modelo predice tokens; el sistema decide qué contexto entra, qué herramientas rodean la salida y cómo se evalúa.

SISTEMA COMPLETO

Prompt
del sistema

RAG

Herramientas
y funciones

Validadores
de salida

Observabilidad
trazas y métricas

Ventana de contexto

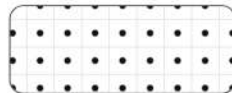
$c = \text{tokens visibles}$

instrucciones
conversación reciente
fragmentos recuperados
token anterior
No es memoria permanente.

entrada

Modelo base

$$h = f\theta(c)$$



θ : parámetros aprendidos

atención causal

MLP · residual · norma

logits

Distribución

$$P(x | c) = \text{softmax}(z)$$

París 0,947
Lyon 0,047
azul 0,006

token: París

El token elegido vuelve a la ventana de contexto y el ciclo continúa.

ESCALA

parámetros

datos

cómputo

contexto

coste y latencia

IA para gente curiosa / Facsímil 03 / Capítulo 01 / 686f6c61

Parámetros: la memoria no es una carpeta

La palabra **parámetro** suele crear confusión. Un parámetro no es una página guardada. No es una ficha con «París = capital de Francia». Es un número dentro de una matriz o un vector. Durante entrenamiento, esos números se ajustan para que las predicciones mejoren.⁹

Una capa lineal simple transforma una entrada así:

$$y = xW + b$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Vector o matriz de entrada.	Un token representado con $d_{in} = 4$ números.
W	Matriz de pesos aprendidos.	Una tabla de 4×3 números.
b	Sesgo aprendido que se suma al resultado.	Un vector de 3 números.
y	Salida transformada.	Nuevo vector de $d_{out} = 3$ números.

El número de parámetros de esa capa es:

$$N_{\text{params}} = d_{in} \cdot d_{out} + d_{out}$$

Si $d_{in} = 4$ y $d_{out} = 3$:

$$N_{\text{params}} = 4 \cdot 3 + 3 = 15$$

Doce pesos en la matriz y tres sesgos. En un LLM real, estas matrices tienen miles de dimensiones y se repiten capa tras capa. Por eso hablamos de millones, miles de millones o billones de parámetros.

De una capa a un modelo entero

Esa fórmula es de juguete, pero escala. Un bloque Transformer está hecho sobre todo de capas lineales: las proyecciones Q , K , V y la de salida de la atención (cada una $d \times d$), más el MLP, que suele expandir a $4d$ y volver. Sumando, cada bloque ronda los $12d^2$ parámetros, y un modelo de L capas se aproxima por:

$$N \approx 12 \cdot L \cdot d^2$$

donde d es la dimensión del modelo (`d_model`) y L el número de capas. Es una estimación, no el valor exacto (faltan embeddings, normalizaciones y sesgos, que pesan menos), pero acierta el orden de magnitud:

```
def parametros_transformer(n_capas, d_model):  
    return 12 * n_capas * d_model ** 2  
  
# Un modelo tipo 7B: 32 capas, d_model 4096  
print(parametros_transformer(32, 4096) / 1e9) # ~6.4 (mil millones)
```

Con 32 capas y `d_model 4096` salen unos 6,4 mil millones de parámetros, muy cerca de los 7B reales; lo que falta (sobre todo los embeddings de vocabulario) explica el resto. Lo útil: ahora puedes mirar la arquitectura de una ficha y estimar tú mismo el tamaño, en vez de creerte la etiqueta.

La intuición importante es esta: los parámetros son la forma en la que el modelo **ha comprimido regularidades del entrenamiento**. No guarda el mundo como una biblioteca; guarda transformaciones numéricas que suelen producir continuaciones plausibles.

Escala: tamaño, datos, cómputo y contexto

La escala de un LLM no es una sola cifra. Cuando alguien dice «este modelo es grande», puede estar mezclando al menos cinco dimensiones:

DIMENSIÓN	QUÉ MIDE	POR QUÉ IMPORTA
Parámetros	Cuántos números aprendidos tiene el modelo.	Aumentan capacidad, memoria necesaria y coste de servirlo.
Datos	Cuántos tokens se usaron durante entrenamiento.	Determinan cobertura de patrones, idiomas, estilos y dominios.
Cómputo de entrenamiento	Cuántas operaciones se invirtieron en ajustar parámetros.	Marca el coste de crear el modelo y parte de su calidad.
Contexto	Cuántos tokens puede leer de una vez.	Afecta a documentos largos, RAG, agentes y memoria de conversación.
Cómputo de inferencia	Cuánto trabajo cuesta generar cada token.	Impacta en latencia, precio y capacidad de escalar un producto.

Las leyes de escalado mostraron que, durante una etapa importante del desarrollo de modelos de lenguaje, aumentar parámetros, datos y cómputo de forma coordinada mejoraba de manera predecible la pérdida.¹⁰ Más tarde, el trabajo conocido como Chinchilla subrayó algo muy práctico: no basta con hacer modelos enormes; hay que equilibrar tamaño del modelo y cantidad de tokens de entrenamiento.¹¹

Esto nos deja una regla de lectura útil: **más parámetros no significa automáticamente mejor modelo para tu caso**. Un modelo menor, bien entrenado, bien alineado con tu idioma, más barato de servir y con buena ventana de contexto puede ser mejor elección que uno mucho mayor.

Qué no arregla la escala

Conviene una dosis de honestidad: hacer el modelo más grande mejora muchas cosas, pero no todas. La escala no aporta **datos actualizados** (un modelo entrenado hasta cierta fecha no conoce lo posterior, por enorme que sea), no elimina la posibilidad de **alucinar** (afirmar con seguridad un dato falso) y puede **amplificar sesgos** presentes en los datos en vez de corregirlos. Bender y sus coautoras lo resumieron con una metáfora incómoda: un modelo de lenguaje, por grande que sea, puede comportarse como un «loro estocástico» que recombina patrones sin entenderlos. La lección práctica es directa: para hechos verificables, datos frescos o decisiones críticas, la escala no sustituye a las fuentes externas, la recuperación y la validación. Por eso el resto del facsímil insiste tanto en RAG, herramientas y evaluación.

Leer una ficha de modelo sin marearse

Una *model card* es una ficha técnica. Puede estar en Hugging Face, en la documentación de un proveedor o en un paper. No siempre trae todo lo que querríamos, pero cuando está bien hecha responde a una pregunta muy práctica: «¿puedo usar este modelo para mi caso sin engañarme?».

La forma de leerla no es empezar por el benchmark más vistoso. Es ir de lo más operativo a lo más fino:

DATO DE LA FICHA	PREGUNTA QUE RESPONDE	DÓNDE VOLVERÁ
Arquitectura	¿Es denso, MoE, multimodal, encoder-decoder, decoder-only?	Facsimil 3 , capítulo 2 y capítulo 5 .
Parámetros totales	¿Cuánta capacidad y memoria bruta puede requerir?	Facsimil 3 , capítulo 5 y capítulo 7 .
Parámetros activos	Si es MoE, ¿cuánto cómputo se activa por token?	Facsimil 3 , capítulo 5 .
Context window	¿Cuánto texto puede recibir sin RAG ni resumen previo?	Facsimil 4 , facsimil 5 y facsimil 6 .
Tokenizer y vocabulario	¿Cómo trocea idiomas, código, números o nombres raros?	Facsimil 1 , capítulo 9 .
Precisión y cuantización	¿FP16, BF16, FP8, INT8, 4-bit? ¿Cabe y con qué pérdida?	Facsimil 3 , capítulo 7 .
Licencia	¿Puedo usarlo, modificarlo, servirlo o vender un producto encima?	Facsimil 4 y bloque de licencia del proyecto .
Benchmarks	¿Qué mide realmente la tabla y con qué metodología?	Facsimil 7 .
Runtime recomendado	¿Transformers, vLLM, SGLang, Ollama, TensorRT-LLM?	Facsimil 4 , facsimil 6 y facsimil 7 .
Limitaciones declaradas	¿En qué idiomas, dominios o tareas falla más?	Facsimil 7 , facsimil 8 y facsimil 9 .

Ejemplo: si una ficha dice «70B, contexto 128k, instruct, FP16», no has terminado de leer. Has empezado. Falta preguntar: ¿tengo memoria para servirlo?, ¿necesito realmente 128k tokens?, ¿qué latencia tolera mi usuario?, ¿hay versión cuantizada?, ¿funciona en español?, ¿qué licencia tiene?, ¿qué benchmark se parece a mi tarea?, ¿necesito RAG aunque tenga mucho contexto?

Otra trampa: comparar una ficha de modelo base con una ficha de modelo instruido. Puede que compartan arquitectura y tamaño, pero no comportamiento. El modelo base puede ser excelente para continuar texto y torpe para seguir instrucciones; el instruido puede ser mucho más útil en chat aunque tenga el mismo número de parámetros. Por eso, cuando alguien diga «uso tal modelo», la pregunta completa es: ¿base o instruct?, ¿qué versión?, ¿qué cuantización?, ¿qué contexto?, ¿qué parámetros de generación?, ¿qué sistema lo rodea?

Cuánto pesa un modelo

La primera consecuencia de los parámetros es física: ocupan memoria. Si un modelo tiene N parámetros y cada parámetro ocupa B bytes, la memoria aproximada solo para pesos es:

$$M_{\text{pesos}} \approx N \cdot B$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
M_{pesos}	Memoria necesaria para guardar los pesos del modelo.	14 GB para 7B en FP16, aproximadamente.
N	Número de parámetros.	7 000 000 000.
B	Bytes por parámetro según precisión.	FP16 usa 2 bytes; INT8 usa 1 byte; 4-bit usa 0,5 bytes.

Ejemplo:

$$M_{\text{pesos}} \approx 7\,000\,000\,000 \cdot 2 = 14\,000\,000\,000 \text{ bytes}$$

Es decir, unos 14 GB decimales solo para los pesos. Y eso no incluye todo lo demás: memoria de activaciones, caché KV durante generación, buffers del runtime, batch, contexto ni el sistema que rodea al modelo.

Para verlo con código:

```
def memoria_pesos(parametros_miles_millones, bytes_por_parametro):
    parametros = parametros_miles_millones * 1_000_000_000
    gb_decimales = parametros * bytes_por_parametro / 1_000_000_000
    gib_binarios = parametros * bytes_por_parametro / (1024 ** 3)
    return gb_decimales, gib_binarios

for nombre, bytes_param in [("FP16", 2), ("INT8", 1), ("4-bit", 0.5)]:
    gb, gib = memoria_pesos(7, bytes_param)
    print(f"{nombre}: {gb:.1f} GB decimales / {gib:.1f} GiB binarios")
```

Salida esperada:

```
FP16: 14.0 GB decimales / 13.0 GiB binarios
INT8: 7.0 GB decimales / 6.5 GiB binarios
4-bit: 3.5 GB decimales / 3.3 GiB binarios
```

Este cálculo no decide si un modelo cabe en tu máquina, pero evita autoengaños. Cuando alguien dice «tengo una GPU de 8 GB, ¿puedo servir un 7B?», la respuesta empieza aquí: pesos, precisión, contexto, batch y runtime.

El contexto también pesa

Hay una segunda memoria que al principio se nos escapa: la memoria de generación. Cuando el modelo genera token a token, no recalcula todo desde cero. Guarda una caché de claves y valores de atención, normalmente llamada **KV cache**. Esa caché crece con el número de capas, la longitud de contexto, la dimensión interna, la precisión y el batch.

Una aproximación útil es:

$$M_{KV} \approx 2 \cdot L \cdot n \cdot d \cdot B$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
M_{KV}	Memoria aproximada de la caché de atención.	Alrededor de 2 GiB en el ejemplo de abajo.
2	Guardamos claves K y valores V .	Dos tensores por capa.
L	Número de capas del modelo.	$L = 32$.
n	Tokens de contexto almacenados.	$n = 4096$.
d	Dimensión interna aproximada por token.	$d = 4096$.
B	Bytes por valor.	FP16 usa $B = 2$.

Con esos números:

$$M_{KV} \approx 2 \cdot 32 \cdot 4096 \cdot 4096 \cdot 2 = 2\,147\,483\,648 \text{ bytes}$$

Unos 2 GiB solo para la caché, en batch 1 y con una aproximación simplificada. En modelos reales hay optimizaciones, variantes de atención y detalles de implementación, pero la intuición se mantiene: **más contexto no es gratis**. Si duplicas el contexto, sube la memoria

de la caché. Si subes el batch, también. Por eso un modelo puede caber con un prompt corto y quedarse sin memoria cuando le pasas un documento largo.

Hay un truco de ingeniería que cambia mucho esta cuenta: **compartir las claves y los valores entre cabezas de atención**. En la atención clásica (MHA) cada cabeza tiene su propio K y V ; en *grouped-query attention* (GQA) varias cabezas comparten un mismo par K, V , y en el extremo *multi-query attention* (MQA) lo comparten todas. Eso divide la KV cache por el factor de compartición, a veces por ocho o más, sin tocar los pesos ni la calidad apenas. Por eso casi todos los modelos de contexto largo usan GQA: es lo que hace asequible guardar 128k tokens en memoria. Lo veremos por dentro en el [capítulo 3](#).

Esta idea conecta directamente con RAG, agentes y producto. No todo documento largo debe meterse entero en contexto. A veces conviene recuperar fragmentos, resumir, estructurar, hacer varias llamadas o diseñar una memoria externa. El contexto es una herramienta, no un trastero infinito.

Modo ingeniero: ¿cabe este modelo en mi GPU?

Ahora podemos juntar las dos memorias en la pregunta que de verdad importa antes de servir un modelo. La memoria total durante inferencia es, a grandes rasgos:

$$M_{\text{total}} \approx M_{\text{pesos}} + M_{\text{KV}} + M_{\text{activaciones}} + M_{\text{overhead}}$$

Las activaciones y el overhead del runtime son más difíciles de predecir, pero una regla prudente es reservar un 20-30 % extra sobre pesos más caché. Veámoslo con el caso clásico: un 7B en una GPU de 8 GB.

```
pesos_fp16 = 7e9 * 2 / 1e9          # 14.0 GB
pesos_4bit = 7e9 * 0.5 / 1e9        # 3.5 GB
kv_4k = 2 * 32 * 4096 * 4096 * 2 / 1e9 # ~2.1 GB (contexto 4k, FP16)
overhead = 0.25                      # 25% extra prudente

total_fp16 = (pesos_fp16 + kv_4k) * (1 + overhead) # ~20 GB
total_4bit = (pesos_4bit + kv_4k) * (1 + overhead) # ~7 GB
print(round(total_fp16, 1), round(total_4bit, 1))
```

La conclusión es honesta: un 7B en FP16 **no** entra en 8 GB (necesita unos 20 GB contando caché y overhead), pero cuantizado a 4 bits **entra justo** (unos 7 GB), siempre que no alargues mucho el contexto. Esto es exactamente lo que decide si puedes correr un modelo localmente, y por qué la cuantización y la gestión del contexto pesan tanto en el [capítulo 7](#).

En el día a día

Entender qué es un LLM cambia cómo eliges tecnología.

Elegir modelo para un producto. No empiezas por «el más grande». Empiezas por la tarea: extracción, soporte, programación, razonamiento, redacción, búsqueda, resumen, clasificación. Luego preguntas: ¿qué calidad necesito?, ¿qué latencia tolero?, ¿cuánto contexto requiere?, ¿qué presupuesto por llamada tengo?, ¿necesito modelo local por privacidad o coste? La arquitectura viene después del problema.

Leer una ficha de modelo. Cuando veas parámetros, contexto, tamaño de vocabulario, arquitectura densa o MoE, precisión, cuantización y benchmarks, no lo leas como marketing suelto. Léelo como restricciones de ingeniería. Cada número te dice algo sobre memoria, latencia, coste, calidad esperable o compatibilidad con tu runtime.

Diseñar una aplicación con IA. Un LLM no debería vivir solo. A su alrededor suelen aparecer recuperación de documentos, herramientas, validadores, observabilidad, pruebas y revisión humana cuando la salida importa. El modelo predice tokens; el sistema completo convierte esa predicción en una experiencia útil y responsable.

Diagnosticar un fallo. Si una respuesta sale mal, no basta con decir «el modelo falla». Puede fallar el prompt, el contexto, el retrieval, el formato esperado, el parámetro de generación, la herramienta llamada, la validación posterior o la expectativa del producto. Separar modelo base, modelo instruido y sistema completo te permite depurar con lupa en vez de cambiar de modelo cada vez que algo se tuerce.

Por qué debería importarte

Porque el error más común al empezar con LLMs es tratarlos como cajas opacas intercambiables. «Ponemos un modelo potente y ya está». Esa frase suele ocultar casi todas las decisiones importantes.

Si sabes qué es un parámetro, entiendes por qué un modelo ocupa memoria. Si sabes qué es un logit, entiendes por qué la salida no es una verdad sino una distribución. Si sabes qué es escala, entiendes por qué comparar modelos exige mirar más que una cifra. Si sabes qué es una KV cache, entiendes por qué el contexto largo tiene coste real. Y si sabes que el modelo predice tokens, entiendes por qué muchas aplicaciones necesitan datos externos, restricciones, evaluación y trazabilidad.

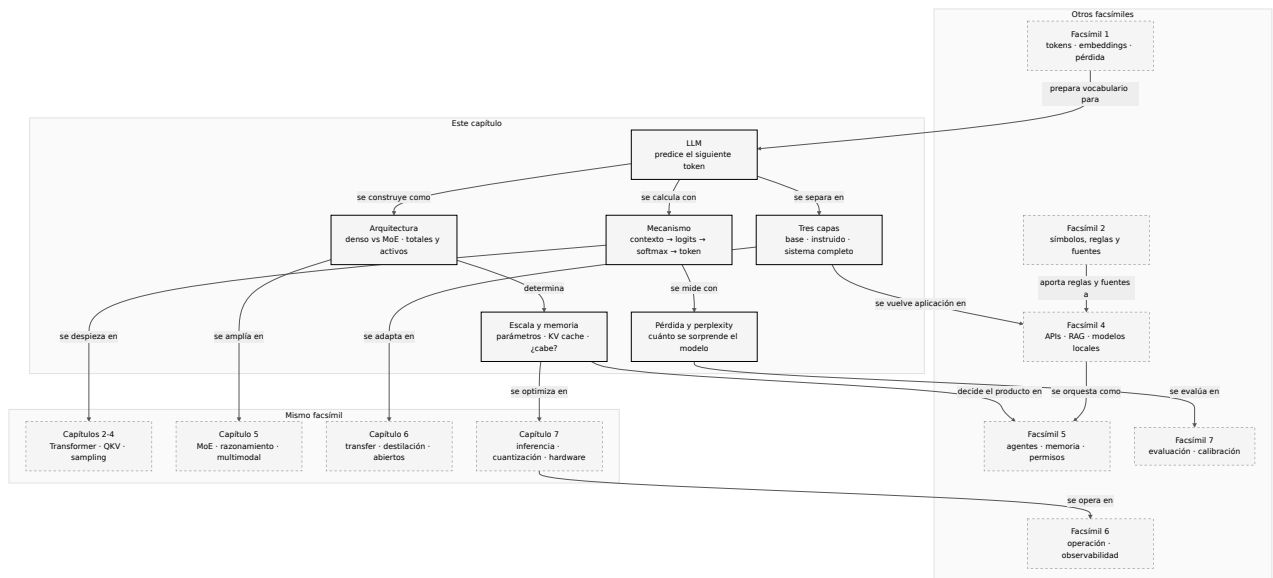
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Confundir modelo con producto	El chat que usas incluye interfaz, herramientas, reglas y memoria de conversación. El LLM es solo una pieza. Si comparas experiencias completas como si fueran modelos puros, sacas conclusiones débiles.	Pregunta siempre qué parte estás evaluando: modelo base, modelo instruido, API, app, RAG, tool use o sistema completo.
Creer que parámetros son conocimiento o literal	Un parámetro no guarda una frase ni una tabla. Es un número dentro de una transformación. Hablar de «memoria» del modelo puede ser útil como metáfora, pero mala como descripción técnica.	Traduce «memoria» por «regularidades comprimidas en pesos». Para hechos exactos, usa fuentes externas verificables.
Pensar que más grande siempre gana	Más parámetros suelen dar más capacidad, pero también más coste, memoria y latencia. Además, datos, entrenamiento, post-entrenamiento y evaluación importan muchísimo.	Compara con una eval propia y restricciones reales: calidad, coste, latencia, contexto, privacidad y mantenimiento.
Olvidar que genera token a token	Una respuesta larga no aparece de golpe. Cada token depende de lo anterior y consume cómputo. Por eso salidas largas cuestan más y tardan más.	Mide tokens de entrada y salida. Diseña límites, streaming y presupuestos antes de producción.
Leer softmax como certeza	Un 94 % de probabilidad para un token no significa que la afirmación sea verdadera. Significa que ese token encaja muy bien con el contexto según el modelo.	Diferencia probabilidad lingüística de veracidad factual. Para hechos, conecta fuentes, citas y validadores.
No separar base, instruct y sistema	Puedes culpar al modelo de un fallo que viene del contexto, de una herramienta o de una regla de producto. También puedes atribuir al modelo capacidades que en realidad vienen del sistema que lo envuelve.	Cuando algo funcione o falle, pregunta: ¿lo produjo el modelo base, el post-entrenamiento, el prompt, el contexto, una herramienta o un validador?

Cómo encaja todo

Este mapa agrupa tres cosas: lo que toca este capítulo, lo que abre en el resto del fascículo y lo que conecta con los demás. Recupera los cimientos del facsímil 1, separa qué es el modelo y qué es el sistema, y muestra por qué arquitectura, herramientas, operación y evaluación no son temas sueltos.

La decisión nueva de este capítulo es llamar a cada cosa por su nombre: parámetros, contexto, logits, KV cache y sistema completo. Si esa separación queda clara aquí, los próximos capítulos dejan de parecer una colección de siglas.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
LLM	Modelo de lenguaje de gran escala entrenado para predecir el siguiente token a partir de un contexto.
Modelo	Función matemática con parámetros aprendidos que transforma entradas en salidas.
Parámetro	Número aprendido durante entrenamiento. Los parámetros determinan cómo se transforman los vectores dentro de la red.
Decoder-only	Arquitectura autoregresiva que genera texto de izquierda a derecha prediciendo el siguiente token.
Escala	Conjunto de tamaño del modelo, datos, cómputo, contexto e inferencia. No es solo número de parámetros.
Logit	Puntuación no normalizada para un token candidato.
Softmax	Función que transforma logits en probabilidades que suman uno.
Temperatura	Parámetro que divide los logits antes del softmax; baja concentra la distribución, alta la aplanada.
Perplexity	Exponencial de la pérdida media por token; opciones efectivas entre las que duda el modelo en cada paso.
Contexto	Tokens disponibles para que el modelo calcule la siguiente predicción.
KV cache	Memoria que guarda claves y valores de atención para no recalcular todo el contexto en cada token generado.
Modelo base	Modelo entrenado principalmente para continuar texto, antes de adaptarlo para conversar o seguir instrucciones.
Modelo instruido	Modelo ajustado para seguir instrucciones y producir respuestas más útiles en interacción humana.
Model card	Ficha técnica que documenta arquitectura, tamaño, contexto, licencia, evaluación y límites de un modelo.
MoE	Arquitectura con varios expertos donde un router activa solo una parte del modelo para cada token.
Cuantización	Técnica que guarda pesos con menos bits para reducir memoria y acelerar inferencia, a cambio de posible pérdida de precisión.

Antes de pasar página

- ☐ ¿Puedo explicar la diferencia entre un LLM, una app de chat y un sistema completo con herramientas? (Si no, vuelve a «El objeto que hay detrás del chat».)
- ☐ ¿Entiendo por qué un LLM no es una base de datos? (Si no, vuelve a «Qué no es un LLM».)
- ☐ ¿Sé distinguir modelo base, modelo instruido y sistema completo? (Si no, vuelve a «Tres capas que conviene no mezclar».)
- ☐ ¿Puedo leer la fórmula $P(x_1, \dots, x_T) = \prod_t P(x_t | x_{<t}; \theta)$ sin asustarme? (Si no, vuelve a «El mecanismo matemático mínimo».)
- ☐ ¿Sé qué es un logit y por qué softmax lo convierte en probabilidad? (Si no, vuelve a «De logits a probabilidades».)
- ☐ ¿Entiendo qué es la perplexity y cómo se relaciona con la pérdida y con la sorpresa de Shannon? (Si no, vuelve a «De la pérdida a la perplexity».)
- ☐ ¿Entiendo cómo la temperatura concentra o aplanla la distribución de salida? (Si no, vuelve a «De logits a probabilidades».)
- ☐ ¿Distingo un modelo denso de un MoE y la diferencia entre parámetros totales y activos? (Si no, vuelve a «Qué tipos de LLM podrías encontrarte».)
- ☐ ¿Puedo leer una ficha de modelo y separar arquitectura, parámetros, contexto, licencia, runtime y benchmarks? (Si no, vuelve a «Leer una ficha de modelo sin marearse».)
- ☐ ¿Puedo calcular cuánta memoria ocupan los pesos de un modelo de 7B en FP16? (Si no, vuelve a «Cuánto pesa un modelo».)
- ☐ ¿Sabría estimar los parámetros de un Transformer a partir de capas y `d_model` ? (Si no, vuelve a «De una capa a un modelo entero».)
- ☐ ¿Sé juntar pesos, KV cache y overhead para decidir si un modelo cabe en una GPU? (Si no, vuelve a «Modo ingeniero: ¿cabe este modelo en mi GPU?».)
- ☐ ¿Entiendo por qué el contexto largo consume memoria aunque los pesos no cambien? (Si no, vuelve a «El contexto también pesa».)
- ☐ ¿Puedo relacionar tokens, parámetros, contexto, logits y escala con otras partes del temario? (Si no, vuelve a «Cómo encaja todo».)
- ☐ ¿Entiendo por qué «más parámetros» no equivale automáticamente a «mejor modelo para mi caso»? (Si no, vuelve a «Dónde solía tropezar yo».)

En resumen

IDEA FUERZA	DETALLE
Un LLM predice el siguiente token.	La respuesta larga aparece porque esa predicción se repite una y otra vez, token a token.
Los parámetros son números aprendidos, no recuerdos literales.	Comprimen regularidades del entrenamiento en matrices y vectores que transforman representaciones.
La salida es una distribución, no una garantía de verdad.	Softmax convierte logits en probabilidades lingüísticas; la veracidad requiere contexto, fuentes y validación.
Modelo base, modelo instruido y sistema completo no son lo mismo.	Separarlos permite depurar fallos y elegir mejor arquitectura, contexto, herramientas y evaluación.
La escala tiene varias dimensiones.	Parámetros, datos, cómputo, contexto, KV cache e inferencia afectan calidad, coste, latencia y viabilidad técnica.

Para saber más

Bender, E. M., Gebru, T., McMillan-Major, A. y Shmitchell, S. (2021). On the dangers of stochastic parrots: can language models be too big? En *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* (pp. 610-623).
<https://doi.org/10.1145/3442188.3445922>

Bengio, Y., Ducharme, R., Vincent, P. y Janvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3, 1137-1155.
<https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf>

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/1457c0d6bfc4967418bfb8ac142f64a-Abstract.html>

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

Hoffmann, J. y otros (2022). Training compute-optimal large language models. En *Advances in Neural Information Processing Systems 35*. <https://doi.org/10.48550/arXiv.2203.15556>

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

Ouyang, L. y otros (2022). Training language models to follow instructions with human feedback. En *Advances in Neural Information Processing Systems 35* (pp. 27730-27744). <https://arxiv.org/abs/2203.02155>

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3), 379-423. <https://doi.org/10.1002/j.1538-7305.1948.tb01338.x>

Vaswani, A. y otros (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. Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3), 379-423. <https://doi.org/10.1002/j.1538-7305.1948.tb01338.x>. Shannon formalizó el lenguaje como una fuente estadística de símbolos, una idea que está en la raíz de los modelos probabilísticos de secuencias. ↵
2. Bengio, Y., Ducharme, R., Vincent, P. y Janvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3, 1137-1155. <https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf>. El artículo introdujo una forma neuronal de modelar lenguaje mediante representaciones distribuidas. ↵
3. Brants, T., Popat, A. C., Xu, P., Och, F. J. y Dean, J. (2007). Large language models in machine translation. En *Proceedings of EMNLP-CoNLL 2007* (pp. 858-867). <https://aclanthology.org/D07-1090/>. Ojo: estos «large language models» eran modelos estadísticos a escala masiva, no LLMs Transformer conversacionales como los actuales. ↵
4. Vaswani, A. y otros (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>. El Transformer hizo viable una arquitectura basada en atención que acabaría dominando los LLM modernos. ↵
5. Radford, A., Narasimhan, K., Salimans, T. y Sutskever, I. (2018). *Improving language understanding by generative pre-training*. OpenAI. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf. El paper introdujo el enfoque GPT original: preentrenar un Transformer decoder y adaptarlo después a tareas. ↵
6. Radford, A. y otros (2019). *Language models are unsupervised multitask learners*. OpenAI. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. ↵
7. 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/1457c0d6bfc4967418bfb8ac142f64a-Abstract.html>.



8. Ouyang, L. y otros (2022). Training language models to follow instructions with human feedback. En *Advances in Neural Information Processing Systems 35* (pp. 27730-27744). <https://arxiv.org/abs/2203.02155>. El trabajo muestra cómo el ajuste con instrucciones y feedback humano cambió el comportamiento observable de modelos de lenguaje sin cambiar la idea base de predicción token a token. ↵
9. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. <https://www.deeplearningbook.org>. El libro explica los parámetros como pesos aprendidos que definen transformaciones entre representaciones. ↵
10. Kaplan, J. y otros (2020). Scaling laws for neural language models. *arXiv preprint*. <https://doi.org/10.48550/arXiv.2001.08361>. El trabajo formalizó relaciones empíricas entre tamaño del modelo, datos, cómputo y pérdida. ↵
11. Hoffmann, J. y otros (2022). Training compute-optimal large language models. En *Advances in Neural Information Processing Systems 35*. <https://doi.org/10.48550/arXiv.2203.15556>. El artículo mostró que muchos modelos grandes estaban infraentrenados respecto al cómputo disponible y propuso un equilibrio distinto entre parámetros y datos. ↵

Capítulo 02: Transformer por dentro: de texto a tensores y atención

El momento en que el texto deja de ser texto

En el capítulo anterior vimos el LLM desde fuera: contexto, parámetros, logits, escala y sistema completo. Ahora vamos a abrir una puerta más pequeña y más concreta: ¿qué ocurre justo después de tokenizar?

Para nosotros, una frase sigue siendo una frase. Para el modelo, ya no. La frase se convierte en una tabla de números. Esa tabla tiene filas, columnas, tamaño, memoria y operaciones permitidas. Cuando un modelo «lee» una oración, en realidad está multiplicando matrices, sumando vectores y calculando pesos de atención.

Este capítulo va de ese paso. No vamos todavía a expresar Q, K, V, máscara causal y softmax con todo detalle; eso será el capítulo 03. Aquí queremos entender la carretera principal: texto → tokens → tensores → atención → representación contextual.

Qué no es un Transformer

Un Transformer no es una lista de reglas gramaticales. No tiene una tabla escrita a mano con «si aparece sujeto, busca verbo». Aprende transformaciones numéricas durante entrenamiento y las aplica a vectores.

Tampoco es una memoria que recorra el texto palabra por palabra como una persona leyendo en voz alta. Antes de los Transformers, muchas arquitecturas secuenciales procesaban una posición tras otra y llevaban un estado interno hacia delante.¹ Eso funcionaba, pero tenía un cuello de botella: si cada paso depende del paso anterior, paralelizar se vuelve más difícil.

Y no es «atención humana» en sentido psicológico. La palabra atención es útil, pero técnica: significa calcular pesos para mezclar información. Si un token atiende a otro, no está «pensando» en él. Está usando su vector para decidir cuánto de otro vector debe entrar en la nueva representación.

Qué sí es un Transformer

Un Transformer es una arquitectura de red neuronal diseñada para procesar secuencias mediante atención y operaciones paralelas sobre matrices.²

La idea central es sencilla de decir y potente de ejecutar: cada token empieza con un vector, y cada capa transforma ese vector usando información de otros tokens. Al final, el vector de cada posición ya no representa solo ese token aislado, sino ese token **dentro de su contexto**.

Por ejemplo, en estas dos frases:

Me senté en el banco
El banco aprobó el préstamo

El token «banco» podría empezar con un embedding parecido en ambos casos. Pero tras pasar por atención, su representación debería cambiar: en la primera frase se acerca a asiento; en la segunda, a entidad financiera. El Transformer no recibe una definición explícita: aprende a usar el contexto.

La atención moderna no apareció de la nada. Antes del Transformer, Bahdanau, Cho y Bengio ya habían mostrado que un modelo de traducción podía aprender a alinear partes de una frase origen con partes de una frase destino.³ Luong, Pham y Manning estudiaron variantes eficaces de atención para traducción neuronal.⁴ El salto del Transformer fue llevar esa intuición al centro de la arquitectura.

El paper original: por qué fue tan raro y tan importante

El paper “**Attention Is All You Need**” se envió a arXiv el 12 de junio de 2017 y fue presentado en NeurIPS 2017.⁵ Lo firmaron ocho autores: **Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser e Illia Polosukhin**.

La propuesta era incómoda para su época porque quitaba dos muletas habituales en modelos de secuencia: recurrencia y convolución. Hasta entonces, para traducir o generar secuencias era normal procesar tokens en orden, mantener un estado oculto y apoyarse en atención como complemento. El Transformer dijo: probemos a construir toda la arquitectura alrededor de atención.

El paper original no nació pensando en chats modernos. Su tarea principal era **traducción automática**. La entrada era una frase en un idioma y la salida una frase en otro. La arquitectura completa tenía dos mitades:

PARTE DEL PAPER	QUÉ HACÍA	CÓMO LO LEEMOS HOY
Encoder	Leía toda la frase de entrada y construía representaciones contextuales.	Se parece a modelos de comprensión como BERT.
Decoder	Generaba la frase de salida token a token mirando lo ya generado.	Se parece a la familia GPT cuando hablamos de generación autoregresiva.
Atención encoder-decoder	Permitía que cada token generado mirase partes relevantes de la entrada.	Es una forma temprana de conectar generación con contexto fuente.
Multi-head attention	Usaba varias atenciones en paralelo para capturar relaciones distintas.	En el capítulo 03 veremos por qué varias cabezas miran patrones distintos.
Feed-forward, residual y normalización	Completaba cada bloque para hacerlo entrenable y expresivo.	En el capítulo 04 lo abriremos despacio.

Esa tabla esconde una distinción que conviene nombrar: no toda la atención es igual. Cuando un token mira a otros de **su misma secuencia** (los de su propio contexto), hablamos de *self-attention*: es la que usan tanto el encoder como, en generación, la familia GPT. Cuando los tokens que se generan miran a los de **otra secuencia** (por ejemplo, la frase de entrada que se está traduciendo), hablamos de *cross-attention*: es el puente encoder-decoder del paper original. Los LLMs decoder-only modernos son, en esencia, una pila de self-attention; la cross-attention reaparece en arquitecturas encoder-decoder y en algunos modelos multimodales, donde el texto atiende a una imagen.

La idea que debemos llevarnos no es «inventaron los LLMs de golpe». Es más precisa: inventaron una arquitectura que hacía mucho más natural entrenar modelos grandes de secuencias con paralelismo y atención directa. Los LLMs modernos heredan esa carretera y la llevan a otra escala.

De tokens a tensores

Empezamos con texto:

El banco aprobó el préstamo

El tokenizador lo convierte en identificadores. No importa ahora si salen cinco o seis tokens; usemos cinco para entender el mecanismo:

$$I = [i_1, i_2, i_3, i_4, i_5]$$

Inteligencia artificial para gente curiosa · 686f6c61 · CC BY 4.0

SÍMBOLO	SIGNIFICADO	EJEMPLO
I	Secuencia de identificadores de token.	$[42, 981, 774, 42, 1503]$.
i_t	Identificador del token en la posición t .	$i_2 = 981$ para «banco».
t	Posición dentro de la secuencia.	$t = 1$ es el primer token.

Cada identificador selecciona una fila de la matriz de embeddings:

$$E \in \mathbb{R}^{V \times d_{\text{model}}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
E	Matriz de embeddings aprendida.	Una tabla gigante de vectores.
V	Tamaño del vocabulario.	100 000 tokens posibles.
d_{model}	Dimensión interna del modelo.	4096 números por token.
$\mathbb{R}$	Conjunto de números reales.	Valores como 0,37 o -1,12.

Después de buscar esas filas, obtenemos una matriz:

$$X = E[I] + P_{1:n}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
X	Tensor de entrada al Transformer.	Una matriz de 5×4096 .
$E[I]$	Filas de embedding seleccionadas por los IDs.	Los vectores de «El», «banco», «aprobó»...
$P_{1:n}$	Información de posición para las n posiciones.	Permite distinguir «perro muerde hombre» de «hombre muerde perro».
n	Número de tokens de la secuencia.	$n = 5$.

De dónde sale la posición

El sumando $P_{1:n}$ merece una parada, porque la atención tiene un punto ciego: por sí sola **no sabe el orden**. Si solo sumáramos embeddings de token, «el perro muerde al hombre» y «el hombre muerde al perro» entrarían como el mismo conjunto de vectores, porque la atención mira a todos los tokens a la vez, sin noción de antes o después. La información de posición es lo que rompe ese empate. Hay tres formas habituales de generarla:

FORMA	CÓMO FUNCIONA	DÓNDE SE USA
Sinusoidal	A cada posición le asigna un patrón fijo de senos y cosenos de distintas frecuencias. No se aprende, se calcula. La propuso Vaswani en el paper original.	El Transformer original.
Aprendida	Una fila de embedding por posición, entrenada como cualquier otro parámetro. Simple, pero limitada a la longitud vista en entrenamiento.	BERT, GPT-2.
Rotatoria (RoPE)	Rota los vectores Q y K un ángulo proporcional a la posición, de modo que la atención depende de la <i>distancia relativa</i> entre dos tokens. Generaliza mejor a contextos largos. ⁶	La mayoría de LLMs modernos.

La idea de fondo es la misma en las tres: que el vector de cada token lleve, además de «qué palabra soy», un «en qué lugar estoy». Sin eso, el Transformer no sería más que una bolsa de palabras muy cara.

Este punto es importante: el Transformer no recibe una frase como texto. Recibe un tensor. Esa forma de pensar (representar datos como vectores, matrices y transformaciones diferenciables) es la base común del *deep learning* moderno.⁷ En la práctica, si procesamos una sola frase con 5 tokens y $d_{\text{model}} = 4$ para simplificar, podríamos imaginar algo así:

TOKEN	VECTOR INTERNO SIMPLIFICADO
El	[0,20, 0,10, -0,30, 0,40]
banco	[0,80, -0,20, 0,10, 0,50]
aprobó	[0,10, 0,70, 0,20, -0,10]
el	[0,18, 0,08, -0,25, 0,36]
préstamo	[0,60, 0,40, 0,70, 0,20]

En un modelo real no habría 4 columnas, sino cientos o miles. Pero la idea es la misma: una fila por token, una columna por dimensión interna.

La atención como mesa de mezclas

La atención responde a una pregunta: para actualizar la representación de este token, ¿cuánto debería mezclar información de los demás tokens?

En una capa de atención simplificada, el modelo calcula tres proyecciones desde X :

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
Q	Consultas (<i>queries</i>): qué busca cada token.	«banco» pregunta qué contexto le da sentido.
K	Claves (<i>keys</i>): qué ofrece cada token para ser encontrado.	«préstamo» ofrece pista financiera.
V	Valores (<i>values</i>): información que se mezclará si recibe atención.	Vector que aporta contenido al resultado.
W_Q, W_K, W_V	Matrices aprendidas que proyectan X .	Parámetros entrenados.

¿Por qué tres proyecciones distintas y no usar X directamente? Porque lo que un token **busca** no tiene por qué ser lo mismo que lo que **ofrece** ni lo que **aporta**. Separar Q , K y V le da al modelo esa libertad: «banco» puede *buscar* contexto financiero (su Q), *ofrecerse* como lugar físico (su K) y *aportar* otro matiz al mezclarse (su V). Con un solo vector, esos tres papeles colapsarían en uno y la atención sería simétrica y pobre. Tres matrices aprendidas permiten además que la relación entre dos tokens no sea recíproca: que A mire mucho a B no obliga a que B mire mucho a A.

Después se calculan puntuaciones comparando consultas con claves:

$$S = \frac{QK^T}{d_k}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
S	Matriz de puntuaciones de atención.	Una tabla $n \times n$.
QK^T	Producto entre consultas y claves.	Mide afinidad entre posiciones.
d_k	Dimensión de las claves.	64, 128 u otra dimensión por cabeza.
d_k	Escalado para estabilizar valores.	Si $d_k = 64$, dividimos por 8.

Ese d_k del denominador no es un capricho. El producto QK^T suma d_k términos: cuanto mayor es d_k , más grandes tienden a ser las puntuaciones. Y si las puntuaciones que entran al softmax son muy grandes, el softmax se **satura**, reparte casi toda la probabilidad a un único token y deja a los demás casi en cero. En esa zona los gradientes son minúsculos y el modelo aprende muy despacio. Dividir por d_k mantiene las puntuaciones en un rango sano, donde el softmax sigue siendo sensible y el entrenamiento estable. Es un detalle pequeño con un efecto grande, y por eso aparece tal cual en el paper original.

Convertimos esas puntuaciones en pesos con softmax:

$$A = \text{softmax}(S)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
A	Matriz de atención.	Cada fila suma 1.
softmax	Convierte puntuaciones en pesos positivos.	2,0 se vuelve más importante que 0,2.
A_{ij}	Peso con el que el token i mira al token j .	«banco» mira a «préstamo» con peso alto.

Por último, mezclamos los valores:

$$O = AV$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
O	Salida de atención.	Nueva representación contextual.
A	Pesos de mezcla.	Cuánto mirar a cada token.
V	Valores disponibles para mezclar.	Información que aporta cada posición.

Leído sin símbolos: cada token produce una pregunta, compara esa pregunta con las claves de todos los tokens, convierte las coincidencias en pesos y mezcla valores según esos pesos. Eso es atención.

Varias cabezas a la vez

Lo que acabamos de describir es **una** cabeza de atención, pero un Transformer real no usa una sola: usa varias en paralelo, lo que se llama *multi-head attention*. Cada cabeza tiene sus propias matrices W_Q , W_K , W_V y, por tanto, aprende a mirar un patrón distinto: una puede especializarse en la concordancia entre sujeto y verbo, otra en a qué nombre se refiere un pronombre, otra en la puntuación. Se calculan todas a la vez, se concatenan sus salidas y una última proyección las combina. La intuición: en vez de una sola mesa de mezclas, hay un panel de varias mesas mirando relaciones diferentes del mismo texto, y luego se juntan. Veremos por dentro cómo y por qué en el capítulo 3.

Cómo funcionaría una pasada completa

Ahora juntemos las piezas sin perdernos. Imagina que queremos generar una respuesta a partir del prompt:

Resume este contrato en tres puntos

Una pasada de inferencia, simplificada, seguiría estos pasos:

1. **Tokenizar.** El texto se trocea en IDs: el modelo no recibe caracteres, recibe números.
2. **Buscar embeddings.** Cada ID selecciona una fila de la matriz E .
3. **Añadir posición.** A cada vector se le suma o aplica información de orden para distinguir la primera posición de la segunda.
4. **Construir el tensor X .** La entrada queda como una matriz $n \times d_{\text{model}}$.
5. **Proyectar a Q , K , V .** Cada capa aprende tres formas de mirar la misma entrada: qué busca, qué ofrece y qué información puede mezclar.
6. **Calcular atención.** Se comparan consultas y claves, se aplica softmax y se obtiene una matriz de pesos.
7. **Mezclar valores.** Cada token recibe una combinación ponderada de información del contexto.
8. **Pasar por residual, normalización y MLP.** La representación se estabiliza y se transforma dentro del bloque.
9. **Repetir capas.** Un modelo real tiene muchas capas; cada una refina las representaciones.

10. **Proyectar a logits.** El último vector se convierte en puntuaciones para todos los tokens del vocabulario.
11. **Elegir el siguiente token.** El sistema aplica temperatura, `top_p`, `top_k` u otra política de generación.
12. **Volver a empezar.** El token elegido se añade al contexto y la siguiente pasada genera el siguiente token.

Si lo escribimos como pseudocódigo conceptual:

```
ids = tokenizador(texto)
X = embeddings(ids) + posicion(ids)

para cada bloque Transformer:
    Q = X W_Q
    K = X W_K
    V = X W_V
    A = softmax(Q K^T / sqrt(d_k))
    X = normalizacion(X + A V)
    X = normalizacion(X + MLP(X))

logits = X_ultimo W_vocab
siguiente_token = muestrear(logits)
```

Ese pseudocódigo no sirve para entrenar un modelo real, pero sí para orientarse. Si alguna vez te pierdes, vuelve a la pregunta: ¿en qué punto estoy? ¿Tokenizando, mezclando contexto, transformando representación o eligiendo el siguiente token?

Un ejemplo pequeño con números

Usemos solo tres tokens para mirar la fila de atención de «banco»:

```
El banco aprobó
```

Imagina que, después de proyectar consultas y claves, las puntuaciones de «banco» hacia cada token son:

TOKEN AL QUE MIRA «BANCO»	PUNTUACIÓN
El	0,2
banco	2,0
aprobó	1,0

Aplicamos softmax:

$$\text{softmax}([0,2, 2,0, 1,0]) \approx [0,108, 0,652, 0,240]$$

VALOR	SIGNIFICADO	LECTURA
0,108	Peso hacia «El».	Aporta poco.
0,652	Peso hacia «banco».	Mantiene mucho de sí mismo.
0,240	Peso hacia «aprobó».	Aporta contexto de acción.

Ahora supongamos valores simplificados de dos dimensiones:

TOKEN	VALOR V
El	[0,1, 0,0]
banco	[1,0, 0,2]
aprobó	[0,4, 1,0]

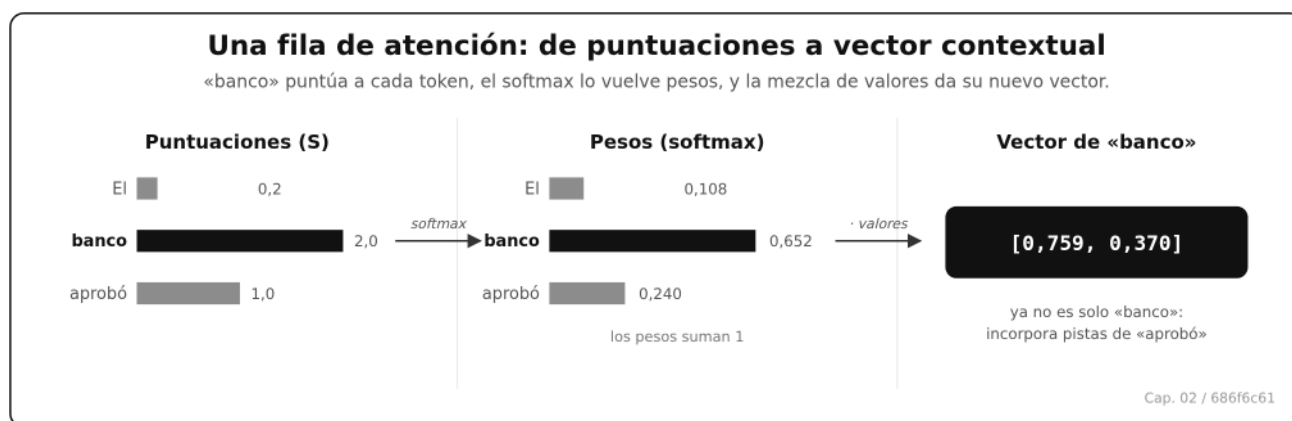
La nueva representación de «banco» es la mezcla ponderada:

$$o_{\text{banco}} = 0,108[0,1, 0,0] + 0,652[1,0, 0,2] + 0,240[0,4, 1,0]$$

Resultado:

$$o_{\text{banco}} \approx [0,759, 0,370]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
o_{banco}	Vector contextualizado del token «banco».	Ya no es solo el embedding inicial.
$0,652[1,0, 0,2]$	Parte que conserva de sí mismo.	El token sigue siendo «banco».
$0,240[0,4, 1,0]$	Parte que recibe de «aprobó».	El verbo cambia el sentido posible.



Esto es pequeño, casi de juguete, pero contiene la intuición completa. La atención no reemplaza el token: lo contextualiza.

Modo ingeniero: la fila de atención en código

Ese cálculo a mano es, casi línea por línea, lo que hace una librería real. En `numpy` cabe en muy pocas líneas: puntuaciones, softmax estable y mezcla ponderada.

```
import numpy as np

def softmax(x):
    x = x - np.max(x)      # estabilidad: restar el máximo evita desbordes
    e = np.exp(x)
    return e / e.sum()

puntuaciones = np.array([0.2, 2.0, 1.0])      # banco -> [El, banco, aprobo]
valores = np.array([[0.1, 0.0],
                    [1.0, 0.2],
                    [0.4, 1.0]])

pesos = softmax(puntuaciones)                 # [0.108, 0.652, 0.240]
o_banco = pesos @ valores                     # mezcla ponderada
print(pesos.round(3), o_banco.round(3))      # [0.108 0.652 0.24] [0.759 0.37]
```

El `@` es el producto matriz-vector: multiplica cada peso por su fila de valores y los suma, justo lo que hicimos a mano. El único truco de ingeniería es el `x - np.max(x)` dentro del softmax: restar el máximo antes de exponenciar evita que `exp` desborde con puntuaciones grandes, sin cambiar el resultado. Esto es exactamente lo que practica el lab de este capítulo, pero con varios tokens y comprobando que los pesos suman 1.

Dentro de un bloque Transformer hay más piezas además de atención. Las conexiones residuales permiten que la información original siga circulando a través de capas profundas, una idea que se volvió central tras ResNet.⁸ La normalización por capa ayuda a estabilizar las activaciones dentro de la red.⁹ En el [capítulo 04](#) entraremos con calma en residual, normalización, MLP y logits.

Simuladores para verlo funcionando

Hay conceptos que se entienden mejor cuando los ves moverse. Estos recursos no sustituyen al capítulo, pero ayudan mucho a mirar la arquitectura con las manos:

RECURSO	QUÉ PUEDES MIRAR	CUÁNDO ABRIRLO
Transformer Explainer	Visualiza un GPT-2 pequeño en el navegador, con flujo de tokens, componentes internos y predicción del siguiente token. ¹⁰	Después de «Cómo funcionaría una pasada completa».
LLM Visualization	Un recorrido 3D por un Transformer estilo GPT, con capas, atención y operaciones internas. ¹¹	Cuando quieras una intuición espacial de las capas.
The Illustrated Transformer	Explicación visual paso a paso del Transformer original, muy buena para fijar encoder, decoder y atención. ¹²	Antes o después de leer el bloque del paper original.
The Annotated Transformer	Implementación anotada del paper, con código y explicación matemática. ¹³	Cuando ya quieras pasar de dibujo a código.

Una forma útil de usarlos: primero abre Transformer Explainer y escribe una frase corta. Mira cómo cambia la distribución del siguiente token. Después vuelve al pseudocódigo de este capítulo y localiza en qué parte del simulador está cada paso. Si una visualización te abruma, no intentes entender todo a la vez: busca solo tokens, atención y logits.

Transformer: de texto a atención

La frase se convierte en una matriz; la atención decide cómo se mezcla la información entre posiciones.

TEXTO Y TOKENS

El

banco

aprobó

préstamo

lookup + posición

TENSOR X

$$X \in \mathbb{R}^{(n \times d_{\text{model}})}$$

El				
banco				
aprobó				
préstamo				

dimensiones internas

Bloque Transformer

atención

residual + norma

MLP

$$X \rightarrow X'$$

MATRIZ DE ATENCIÓN

$$A \in \mathbb{R}^{(n \times n)}$$

cada fila dice qué mezcla cada token

salida contextualizada

el vector de «banco» ya incorpora pistas de «aprobó» y «préstamo»

IA para gente curiosa / Facsímil 03 / Capítulo 02 / 686f6c61

Por qué el Transformer cambió el juego

La ventaja práctica del Transformer está en dos ideas que se refuerzan.

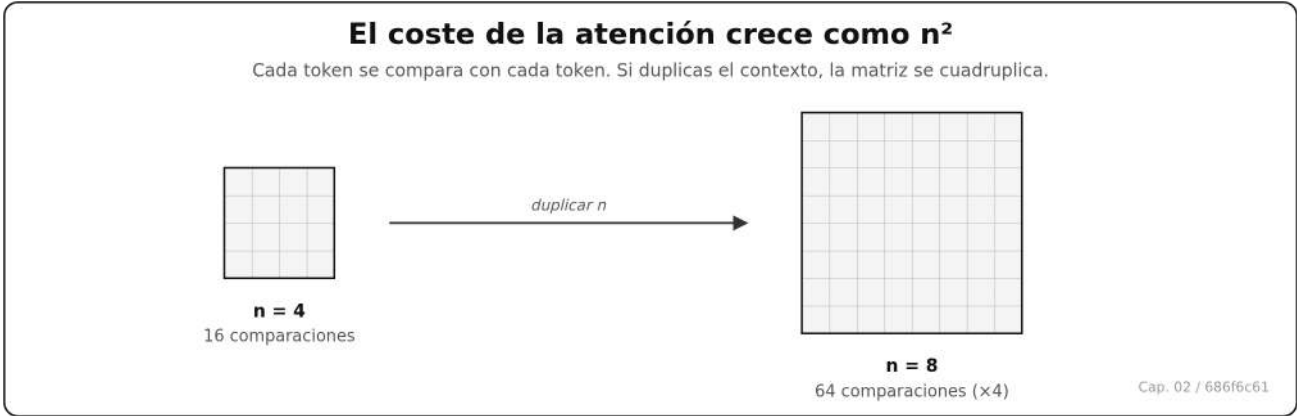
Primera: paralelismo. En una arquitectura recurrente clásica, procesar el token 200 depende de haber procesado antes los 199 tokens previos. En un Transformer, muchas operaciones se expresan como multiplicaciones de matrices. Eso encaja muy bien con GPU y TPU, que son máquinas excelentes para operar con bloques grandes de números.

Segunda: acceso directo al contexto. Si el token de la posición 200 necesita información de la posición 12, la atención puede crear una conexión directa dentro de la matriz. No necesita transportar todo a través de un único estado oculto. Esa es una de las razones por las que el Transformer se volvió la arquitectura dominante para modelos grandes de lenguaje.

Pero no todo es gratis. La atención estándar compara cada token con cada token. Si hay n tokens, la matriz de atención tiene $n \times n$ entradas:

$$\text{coste de atención} \propto n^2$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
n	Número de tokens de contexto.	1 000 tokens.
n^2	Comparaciones entre posiciones.	1 000 000 relaciones posibles.
$\propto$	«Proporcional a».	Si duplicas n , el coste crece aproximadamente por cuatro.



Esto explica por qué el contexto largo es tan valioso y tan caro. No basta con decir «metamos más texto». El diseño del sistema debe decidir qué entra, qué se resume, qué se recupera con RAG y qué conviene dejar fuera.

Además, no todos los Transformers leen igual. BERT usa una arquitectura orientada a entender texto mirando a ambos lados del contexto, muy útil para tareas de comprensión.¹⁴ La familia GPT usa el enfoque autoregresivo: predice el siguiente token mirando el contexto disponible hacia atrás, que es la base de los LLMs generativos modernos.¹⁵

En el día a día

Cuando integras un modelo en una aplicación, rara vez manipulas Q , K , V a mano. Pero sí pagas sus consecuencias.

Diseñar prompts largos. Si añades instrucciones, documentos, historial y ejemplos, estás aumentando n . Más tokens significan más memoria, más latencia y más coste. Este capítulo te ayuda a entender por qué el contexto debe diseñarse, no solo rellenarse.

Elegir entre modelos. Dos modelos pueden tener calidades parecidas, pero arquitecturas internas distintas. Uno puede ser más rápido con contexto corto; otro puede estar optimizado para contexto largo. Cuando leas una ficha técnica, «context window», «attention implementation», «KV cache» y «throughput» dejan de ser palabras decorativas.

Depurar respuestas raras. A veces el problema no está en que el modelo «no sepa», sino en que la información relevante no está en el contexto, queda enterrada entre demasiado ruido o llega en un formato difícil de usar. Entender atención te recuerda que el modelo mezcla señales: si la señal útil está muy diluida, la salida también puede estarlo.

Conectar con RAG. En RAG, no meteremos todos los documentos en el prompt. Recuperaremos fragmentos. La razón no es solo comodidad: es arquitectura. El contexto es una ventana cara y limitada.

Por qué debería importarte

Porque el Transformer es la bisagra entre «texto» y «sistema de IA moderno». Si solo sabes que los LLMs predicen tokens, entiendes el comportamiento externo. Si además entiendes tensores y atención, empiezas a leer por dentro por qué el contexto importa, por qué el orden importa, por qué el coste crece y por qué algunos errores se arreglan cambiando el contexto en vez de cambiando de modelo.

También te prepara para el capítulo siguiente. Q, K, V, máscara causal y softmax no son nombres sueltos: son las piezas concretas que hacen posible esta mesa de mezclas.

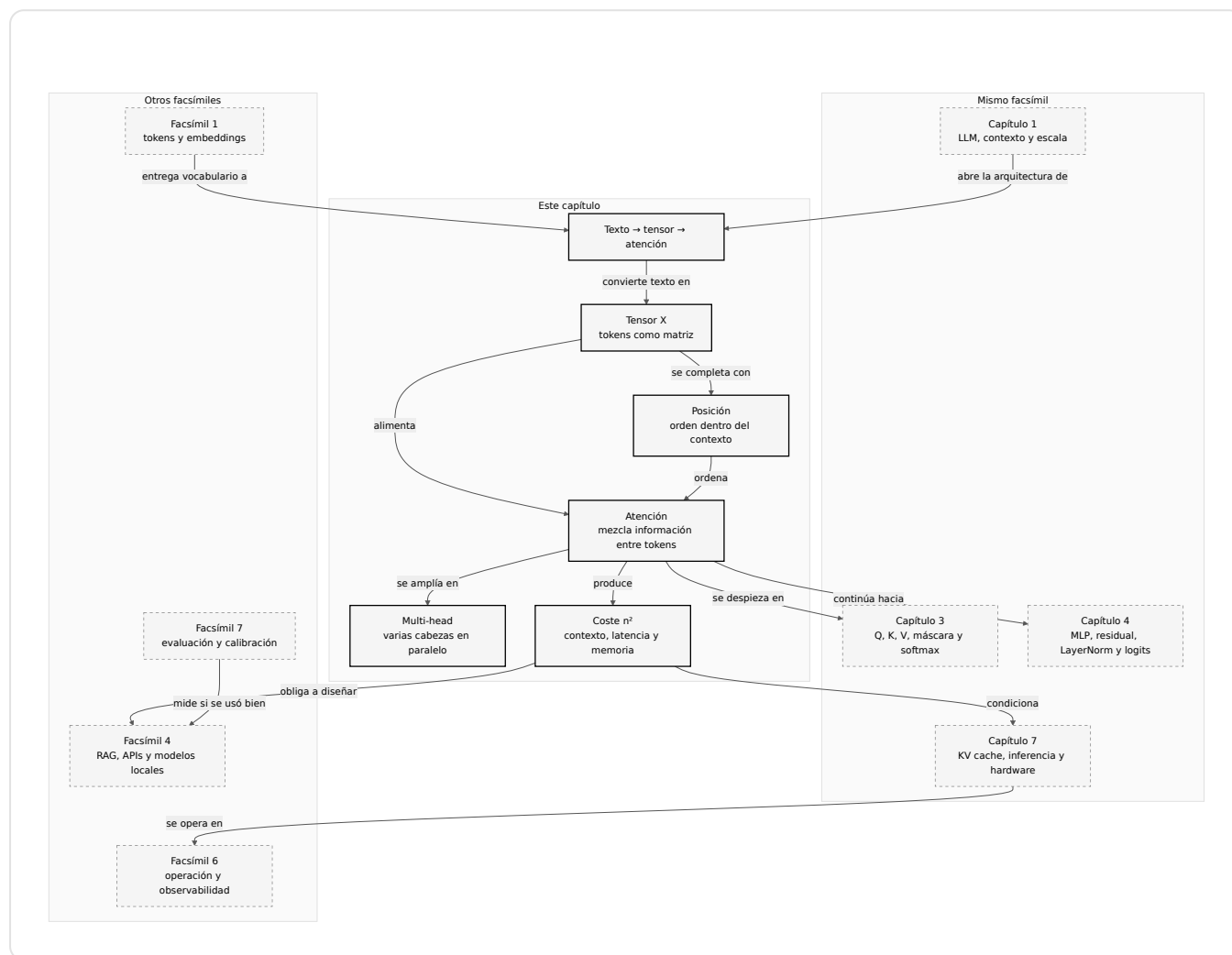
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Pensar que el tensor es un detalle de implementación	El tensor X define cuántos tokens hay, cuántas dimensiones maneja el modelo y qué coste tendrá procesar la entrada.	Cuando leas una arquitectura, pregunta siempre por las formas: n , d_{model} , capas y cabezas.
Creer que atención es explicación perfecta	Un peso de atención alto indica mezcla de información, pero no equivale automáticamente a una explicación humana completa.	Úsala como pista técnica, no como prueba definitiva de causalidad.
Olvidar la posición	Si solo sumas embeddings de tokens sin posición, pierdes orden. Y en lenguaje, el orden cambia el significado.	Repite mentalmente: token + posición = entrada útil para secuencia.
Meter todo en contexto por tranquilidad	Más contexto puede añadir ruido, coste y latencia. No todo lo disponible es útil.	Recupera, resume y ordena. El contexto es una superficie de diseño.
Separar demasiado los capítulos	Tokens, embeddings, atención, logits y evaluación parecen temas distintos, pero son una cadena.	Sigue el flujo: texto → tensor → atención → logits → decisión de producto.

Cómo encaja todo

Este mapa baja un nivel respecto al capítulo anterior. Ya no miramos solo el LLM como producto matemático: miramos cómo una frase se vuelve tensor, cómo la posición evita perder el orden y cómo la atención empieza a mezclar señales.

La conexión importante es práctica: cada vez que en facsímiles posteriores hablemos de contexto, RAG, coste o latencia, estaremos pagando decisiones que nacen aquí, en el tamaño de n , d_{model} y la matriz de atención.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Tensor	Estructura numérica con dimensiones. Una matriz $n \times d$ es un tensor de dos dimensiones.
d_{model}	Dimensión interna del modelo: cuántos números representan cada token dentro del Transformer.
Embedding posicional	Información que indica la posición de cada token para que el modelo no pierda el orden.
Atención	Mecanismo que calcula pesos para mezclar información entre posiciones de una secuencia.
Multi-head attention	Varias cabezas de atención en paralelo, cada una con sus proyecciones, que miran patrones distintos.
Matriz de atención	Tabla $n \times n$ donde cada fila indica cuánto mira un token a los demás.
Consulta, clave y valor	Tres proyecciones aprendidas que permiten preguntar, comparar y mezclar información.
Residual	Conexión que suma la entrada del bloque a su salida para no perder señal y entrenar mejor.
Normalización	Operación que estabiliza activaciones dentro de la red.
Paralelismo	Capacidad de ejecutar muchas operaciones a la vez, especialmente multiplicaciones de matrices.

Antes de pasar página

- ☐ ¿Puedo explicar por qué el Transformer recibe una matriz y no texto en bruto? (Si no, vuelve a «De tokens a tensores».)
- ☐ ¿Entiendo qué significa $X \in \mathbb{R}^{n \times d_{\text{model}}}$? (Si no, vuelve a «De tokens a tensores».)
- ☐ ¿Sé decir para qué sirve la información de posición? (Si no, vuelve a «De tokens a tensores».)
- ☐ ¿Distingo las tres formas de codificar la posición (sinusoidal, aprendida, RoPE)? (Si no, vuelve a «De dónde sale la posición».)

- ☐ ¿Puedo explicar la atención como una mezcla ponderada de valores? (Si no, vuelve a «La atención como mesa de mezclas».)
- ☐ ¿Entiendo por qué Q, K y V son tres proyecciones distintas y por qué se divide por d_k ? (Si no, vuelve a «La atención como mesa de mezclas».)
- ☐ ¿Sé qué aporta multi-head attention frente a una sola cabeza? (Si no, vuelve a «Varias cabezas a la vez».)
- ☐ ¿Puedo calcular una softmax pequeña y leer sus pesos? (Si no, vuelve a «Un ejemplo pequeño con números».)
- ☐ ¿Puedo narrar una pasada completa desde el texto hasta el siguiente token? (Si no, vuelve a «Cómo funcionaría una pasada completa».)
- ☐ ¿He abierto al menos un simulador y sé localizar tokens, atención y logits? (Si no, vuelve a «Simuladores para verlo funcionando».)
- ☐ ¿Entiendo por qué el coste de atención crece con n^2 ? (Si no, vuelve a «Por qué el Transformer cambió el juego».)
- ☐ ¿Sé conectar este capítulo con Q, K, V y máscara causal del capítulo siguiente? (Si no, vuelve a «Cómo encaja todo».)
- ☐ ¿Puedo explicar por qué meter más contexto no siempre mejora una aplicación? (Si no, vuelve a «Dónde solía tropezar yo».)

En resumen

IDEA FUERZA	DETALLE
El texto entra al Transformer como tensor.	Después de tokenizar y buscar embeddings, la entrada tiene forma $n \times d_{\text{model}}$: filas para tokens, columnas para dimensiones internas.
La atención contextualiza cada token.	Cada posición calcula pesos sobre otras posiciones y mezcla valores para construir una representación dependiente del contexto.
El Transformer ganó por paralelismo y acceso directo.	Procesa secuencias con operaciones matriciales y permite que posiciones lejanas se conecten sin pasar por un único estado oculto.
El contexto largo tiene coste real.	La matriz de atención crece como n^2 , así que diseñar contexto es una decisión técnica y de producto.

Para saber más

Alammar, J. (2018). *The Illustrated Transformer*. <https://jalammar.github.io/illustrated-transformer/>

Ba, J. L., Kiros, J. R. y Hinton, G. E. (2016). *Layer normalization*. <https://arxiv.org/abs/1607.06450>

Bahdanau, D., Cho, K. y Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. En *International Conference on Learning Representations*. <https://arxiv.org/abs/1409.0473>

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/1457c0d6bfc4967418bfb8ac142f64a-Abstract.html>

Bycroft, B. (2023). *LLM Visualization*. <https://bbycroft.net/llm>

Cho, A., Kim, G. C., Karpekov, A., Helbling, A., Wang, Z. J., Lee, S., Hoover, B. y Chau, D. H. (2025). Transformer Explainer: interactive learning of text-generative models. En *Proceedings of the AAAI Conference on Artificial Intelligence*. <https://ojs.aaai.org/index.php/AAAI/article/download/35347/37502>

Devlin, J., Chang, M. W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. En *Proceedings of NAACL-HLT* (pp. 4171-4186). <https://doi.org/10.18653/v1/N19-1423>

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

He, K., Zhang, X., Ren, S. y Sun, J. (2016). Deep residual learning for image recognition. En *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770-778). <https://doi.org/10.1109/CVPR.2016.90>

Luong, M.-T., Pham, H. y Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. En *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (pp. 1412-1421). <https://doi.org/10.18653/v1/D15-1166>

Rush, A. M. (2018). *The Annotated Transformer*. <https://nlp.seas.harvard.edu/annotated-transformer/>

Sutskever, I., Vinyals, O. y Le, Q. V. (2014). Sequence to sequence learning with neural networks. En *Advances in Neural Information Processing Systems 27* (pp. 3104-3112). <https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks>

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. Sutskever, I., Vinyals, O. y Le, Q. V. (2014). Sequence to sequence learning with neural networks. En *Advances in Neural Information Processing Systems 27* (pp. 3104-3112). <https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks>. Los modelos sequence-to-sequence con redes recurrentes fueron un paso clave para traducir y generar secuencias antes del dominio del Transformer. ↵
2. 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>. El artículo presentó una arquitectura sin recurrencia ni convolución, basada en atención, capaz de entrenar modelos de secuencia de forma mucho más paralela. ↵
3. Bahdanau, D., Cho, K. y Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. En *International Conference on Learning Representations*. <https://arxiv.org/abs/1409.0473>. El trabajo introdujo un mecanismo de atención para traducción neuronal que permitía al decodificador consultar partes relevantes de la entrada. ↵
4. Luong, M.-T., Pham, H. y Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. En *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (pp. 1412-1421). <https://doi.org/10.18653/v1/D15-1166>. El artículo comparó formas prácticas de calcular atención en traducción automática neuronal. ↵
5. 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://arxiv.org/abs/1706.03762>. La página de arXiv muestra la fecha de envío, los autores y el resumen del trabajo. ↵
6. Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B. y Liu, Y. (2021). *RoFormer: Enhanced Transformer with Rotary Position Embedding*. <https://arxiv.org/abs/2104.09864>. RoPE codifica la posición rotando las consultas y las claves, lo que captura posiciones relativas; es la opción dominante en los LLMs recientes. ↵
7. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. <https://www.deeplearningbook.org>. El libro presenta las redes profundas como composiciones de funciones diferenciables sobre tensores, justo el marco matemático que usamos aquí para leer el Transformer. ↵

8. He, K., Zhang, X., Ren, S. y Sun, J. (2016). Deep residual learning for image recognition. En *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770-778). <https://doi.org/10.1109/CVPR.2016.90>. Aunque ResNet nació en visión por computador, popularizó las conexiones residuales que ayudan a entrenar redes muy profundas. ↵
9. Ba, J. L., Kiros, J. R. y Hinton, G. E. (2016). *Layer normalization*. <https://arxiv.org/abs/1607.06450>. LayerNorm normaliza activaciones dentro de una capa y se convirtió en una pieza habitual de arquitecturas Transformer. ↵
10. Cho, A., Kim, G. C., Karpekov, A., Helbling, A., Wang, Z. J., Lee, S., Hoover, B. y Chau, D. H. (2025). Transformer Explainer: interactive learning of text-generative models. En *Proceedings of the AAAI Conference on Artificial Intelligence*. <https://ojs.aaai.org/index.php/AAAI/article/download/35347/37502>. El artículo describe la herramienta como una visualización web interactiva que ejecuta un modelo GPT-2 localmente en el navegador. ↵
11. Bycroft, B. (2023). *LLM Visualization*. <https://bbycroft.net/llm>. Visualización interactiva 3D de un modelo Transformer estilo GPT. ↵
12. Alammr, J. (2018). *The Illustrated Transformer*. <https://jalammar.github.io/illustrated-transformer/>. Recurso visual ampliamente usado para entender la arquitectura Transformer. ↵
13. Rush, A. M. (2018). *The Annotated Transformer*. <https://nlp.seas.harvard.edu/annotated-transformer/>. Implementación comentada de la arquitectura Transformer original. ↵
14. Devlin, J., Chang, M. W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. En *Proceedings of NAACL-HLT* (pp. 4171-4186). <https://doi.org/10.18653/v1/N19-1423>. BERT popularizó el preentrenamiento bidireccional con Transformer encoder para comprensión de lenguaje. ↵
15. 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/1457c0d6bfc4967418bfb8ac142f64a-Abstract.html>. GPT-3 mostró que un Transformer decoder autoregresivo a gran escala podía realizar muchas tareas mediante ejemplos en el propio prompt. ↵

Capítulo 03: Q, K, V, máscara causal y softmax

La mesa de mezclas ya tiene mandos

En el capítulo anterior dijimos que la atención era una mesa de mezclas: cada token decide cuánto tomar de otros tokens para construir una representación contextual. Pero dejamos sin abrir los mandos concretos.

Ahora entran tres letras que aparecen por todas partes cuando alguien explica Transformers: **Q**, **K** y **V**. Al principio parecen una contraseña. En realidad son tres formas distintas de mirar el mismo tensor: qué busca cada token, qué ofrece cada token y qué información se mezcla al final.

También aparece una regla silenciosa pero decisiva: la **máscara causal**. Sin ella, un modelo generativo haría trampas durante entrenamiento: para predecir el token 3 podría mirar el token 4, que en generación real todavía no existe.

Qué no son Q, K y V

Q, K y V no son tres bases de datos internas. No hay una tabla llamada «preguntas», otra llamada «claves» y otra llamada «valores» con significado humano directo. Son proyecciones lineales aprendidas a partir del tensor de entrada.

Tampoco son conceptos nuevos separados del capítulo anterior. En el [capítulo 02](#) ya teníamos una matriz X con una fila por token. Q, K y V salen de esa misma matriz, multiplicándola por tres matrices de pesos distintas.

Y la máscara causal no es un filtro moral ni una regla de producto. Es una restricción geométrica sobre qué posiciones pueden verse entre sí. Su función es mantener honesta la generación de izquierda a derecha.

Qué sí son Q, K y V

En self-attention, cada token produce tres vectores: una consulta (*query*), una clave (*key*) y un valor (*value*). El Transformer original formalizó este mecanismo como atención escalada por producto punto y lo convirtió en la pieza central de la arquitectura.¹

La intuición:

LETRA	NOMBRE	PREGUNTA INFORMAL
Q	Query / consulta	¿Qué estoy buscando desde esta posición?
K	Key / clave	¿Qué tipo de información ofrezco para que otros me encuentren?
V	Value / valor	Si alguien me mira, ¿qué información le entrego?

Como lectura pedagógica complementaria, Alyona Vert resume esta intuición de forma muy útil: Q busca, K permite decidir si algo merece atención y V transporta el contenido que se mezcla; además conecta esa separación con la KV cache que veremos en inferencia.²

Imagina la frase:

El banco aprobó el préstamo

Para la posición «banco», la consulta puede aprender a buscar señales que aclaren el sentido. Las claves de «aprobó» y «préstamo» pueden ofrecer pistas financieras. Los valores son la información que realmente se mezcla cuando esas posiciones reciben peso.

Las proyecciones lineales

Partimos de la matriz X :

$$X \in \mathbb{R}^{n \times d_{\text{model}}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
X	Tensor de entrada a la capa de atención.	4 tokens por 3 dimensiones en el ejemplo pequeño.
n	Número de tokens.	$n = 4$.
d_{model}	Dimensión interna del modelo.	En un modelo real, 768, 4096 o más.

Calculamos:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
W_Q	Matriz aprendida para consultas.	Transforma cada fila de X en lo que busca.
W_K	Matriz aprendida para claves.	Transforma cada fila de X en lo que ofrece.
W_V	Matriz aprendida para valores.	Transforma cada fila de X en lo que se mezclará.
Q, K, V	Tres matrices derivadas de la misma entrada.	Tres versiones útiles del contexto.

Estas multiplicaciones son capas lineales, una idea básica de redes neuronales profundas.³ Lo interesante no es que la operación sea exótica, sino que el entrenamiento aprende qué proyecciones convienen para comparar tokens.

Puntuaciones: comparar lo que busco con lo que ofreces

Para saber cuánto debe mirar cada token a cada otro token, comparamos consultas con claves:

$$S = \frac{QK^T}{d_k}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
S	Matriz de puntuaciones antes de softmax.	Una tabla $n \times n$.
QK^T	Producto entre consultas y claves.	Afinidad entre cada par de posiciones.
K^T	Transpuesta de K .	Convierte columnas en filas para multiplicar.
d_k	Dimensión de las claves.	Si las claves tienen 64 dimensiones, $d_k = 64$.
d_k	Factor de escala.	Evita puntuaciones demasiado grandes.

El escalado por d_k ayuda a que las puntuaciones no crezcan demasiado cuando la dimensión aumenta. Si las puntuaciones llegan enormes a softmax, la distribución puede volverse muy extrema y los gradientes menos útiles. Esto lo verás también al estudiar optimización y estabilidad numérica.

La máscara causal: no mirar el futuro

En un modelo generativo autoregresivo, la posición i solo puede mirar posiciones anteriores o la misma posición. No puede mirar posiciones futuras. La máscara causal se puede escribir así:

$$M_{ij} = \begin{cases} 0 & \text{si } j \leq i \\ -\infty & \text{si } j > i \end{cases}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
M_{ij}	Valor de la máscara para la fila i , columna j .	Fila 2 no puede mirar columna 4.
$j \leq i$	Posición visible.	El token actual puede mirar lo anterior.
$j > i$	Posición futura.	Se bloquea antes de softmax.
$-\infty$	Valor ideal que softmax convierte en peso 0.	El futuro queda con probabilidad cero.

¿Por qué $-\infty$ y no simplemente poner el peso a cero después del softmax? Porque sumar $-\infty$ **antes** del softmax hace el trabajo solo: como $e^{-\infty} = 0$, esa posición sale con peso exactamente cero y el resto de la fila se normaliza solo entre las posiciones permitidas. Si en cambio calculáramos el softmax con todas las posiciones y luego borraríamos las futuras, la fila ya no sumaría 1 y habría que renormalizar a mano. En código real no se usa $-\infty$ literal (produciría operaciones inválidas), sino un número muy negativo como -10^9 : al exponenciar queda prácticamente cero, con el mismo efecto y sin romper la aritmética.

Aplicamos la máscara a las puntuaciones:

$$\tilde{S} = S + M$$

Después:

$$A = \text{softmax}(\tilde{S})$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
$\tilde{S}$	Puntuaciones ya enmascaradas.	Futuro sustituido por $-\infty$.
A	Matriz final de atención.	Cada fila suma 1.
softmax	Función que convierte puntuaciones en pesos positivos.	Muy usada en clasificación y en LLMs. ⁴

La máscara causal es una de las diferencias importantes entre un modelo que genera texto de izquierda a derecha y un modelo de comprensión que puede mirar ambos lados del texto. GPT usa atención causal para generación autoregresiva.⁵ BERT, en cambio, usa una arquitectura bidireccional pensada para comprensión.⁶

Conviene poner nombre a las tres formas de atención que aparecen en la práctica, porque se confunden con facilidad:

VARIANTE	QUIÉN MIRA A QUIÉN	MÁSCARA	DÓNDE SE USA
Self-attention	Los tokens de una secuencia se miran entre sí.	Sin máscara causal: cada token ve toda la secuencia.	El encoder de BERT, comprensión bidireccional.
Self-attention enmascarada	Los tokens de una secuencia se miran, pero solo hacia atrás.	Máscara causal: el futuro queda bloqueado.	El decoder de GPT, generación autoregresiva.
Cross-attention	Los tokens de una secuencia miran a los de otra .	Sin máscara causal (mira toda la otra secuencia).	Encoder-decoder (traducción) y modelos multimodales (texto que mira una imagen).

La diferencia entre las dos primeras es solo la máscara. La misma maquinaria de Q, K, V sirve para las tres; lo que cambia es a qué se permite mirar.

Cómo leer la matriz sin perderse

La matriz de atención se lee **fila a fila**. Cada fila responde a una pregunta concreta: «desde este token, ¿cuánto miro a cada posición disponible?». Por eso no tiene sentido mirar un número aislado sin saber en qué fila vive.

En una frase de cuatro tokens, la máscara causal deja esta forma:

FIL A	TOKEN QUE MIRA	PUEDE MIRAR	NO PUEDE MIRAR TODAVÍA	LECTURA
1	El	El	banco, aprobó, préstamo	No hay pasado; solo se ve a sí mismo.
2	banco	El, banco	aprobó, préstamo	Tiene un poco de contexto, pero no sabe lo que viene después.
3	aprobó	El, banco, aprobó	préstamo	Puede usar el sujeto y el verbo, pero no el objeto futuro.
4	préstamo	El, banco, aprobó, préstamo	(ninguna)	Ya puede mirar toda la frase disponible.

La máscara causal: cada fila solo ve su pasado

Filas: el token que mira. Columnas: el token mirado. El triángulo superior (el futuro) queda bloqueado.

	El	banco	aprobó	préstamo
El	1,000	×	×	×
banco	0,501	0,499	×	×
aprobó	0,319	0,315	0,366	×
préstamo	0,218	0,214	0,284	0,284

☐ visible (pasado y presente)
☒ bloqueado (futuro)

Cada fila visible suma 1.

Cap. 03 / 686f6c61

Después de aplicar softmax, cada fila suma 1. Eso significa que el token reparte toda su atención entre las posiciones permitidas. Si una fila tiene cuatro posiciones posibles, no significa que las cuatro importen igual; significa que el reparto completo se hace dentro de esas cuatro posiciones.

Una forma sana de leerlo es esta:

1. Elige una fila.
2. Mira qué columnas permite la máscara.
3. Aplica softmax solo a esas puntuaciones útiles.
4. Usa esos pesos para mezclar los valores V .

Si sigues esos cuatro pasos, la fórmula deja de ser una pared y se convierte en una receta.

Valores: mezclar la información

Una vez tenemos los pesos de atención A , mezclamos los valores:

$$O = AV$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
O	Salida de la atención.	Nueva matriz con representaciones contextualizadas.
A	Pesos de atención.	Cuánto mira cada token a cada posición permitida.
V	Valores.	Información que se mezcla.

Esta fórmula es la parte más importante para la intuición: Q y K deciden **cuánto** mirar; V decide **qué información** se toma cuando miras.

Cuánto cuesta esta operación

Merece la pena ver de dónde sale el coste. Para puntuar, QK^T compara las n consultas con las n claves, cada una de dimensión d_k : son del orden de $n^2 \cdot d_k$ operaciones, y produce una matriz $n \times n$. La mezcla AV vuelve a recorrer esa matriz $n \times n$ por cada dimensión de los valores, otro $n^2 \cdot d$. El término que manda es el n^2 : cada token se compara con cada token. Por eso, como vimos en el capítulo anterior, **duplicar el contexto cuadruplica el trabajo de la atención**, y por eso casi toda la investigación de contexto largo busca formas de no pagar ese n^2 entero.

De Q , K , V a la KV cache

Aquí está el germen de una optimización capital. En generación, el modelo produce un token, lo añade al contexto y repite. Pero gracias a la máscara causal, los tokens ya generados **solo miran hacia atrás**: su K y su V no cambian cuando llega un token nuevo. Recalcularlos en cada paso sería un derroche. La solución es guardarlos: la **KV cache** almacena las claves y los valores de todo el pasado, y en cada token nuevo solo se calcula su Q , su K y su V y se compara contra la caché. Sin máscara causal esto no sería posible, porque el pasado podría depender del futuro. Veremos el coste en memoria de esa caché en el [capítulo 7](#).

Un ejemplo pequeño con máscara causal

Usemos cuatro tokens:

El banco aprobó préstamo

Tras las proyecciones, supongamos que obtenemos estas matrices pequeñas:

TOKEN	Q	K	V
El	$[-0,01, -0,04]$	$[0,18, -0,19]$	$[-0,02, 0,24]$
banco	$[0,47, 0,00]$	$[0,17, -0,23]$	$[0,51, -0,04]$
aprobó	$[-0,03, 0,51]$	$[0,51, 0,21]$	$[0,23, 0,50]$
préstamo	$[0,43, 0,58]$	$[0,45, 0,25]$	$[0,75, 0,16]$

La matriz de puntuaciones enmascarada queda así, redondeada:

TOKEN QUE MIRA	EL	BANCO	APROBÓ	PRÉSTAMO
El	0,004	n/d	n/d	n/d
banco	0,060	0,056	n/d	n/d
aprobó	-0,072	-0,087	0,065	n/d
préstamo	-0,023	-0,043	0,241	0,239

La marca «n/d» significa «posición futura bloqueada por la máscara». Cuando aplicamos softmax por fila:

TOKEN QUE MIRA	EL	BANCO	APROBÓ	PRÉSTAMO
El	1,000	0,000	0,000	0,000
banco	0,501	0,499	0,000	0,000
aprobó	0,319	0,315	0,366	0,000
préstamo	0,218	0,214	0,284	0,284

Mira la segunda fila. El token «banco» solo puede mirar «El» y «banco». Aunque «aprobó» y «préstamo» serían pistas útiles, todavía son futuro para esa posición en generación autoregresiva. Por eso su peso es 0.

Después multiplicamos por V :

$$O = AV$$

La salida redondeada es:

TOKEN	SALIDA CONTEXTUALIZADA
El	$[-0,020, 0,240]$
banco	$[0,245, 0,100]$
aprobó	$[0,238, 0,247]$
préstamo	$[0,383, 0,231]$

Esta tabla enseña una idea que cuesta ver al principio: el mismo mecanismo sirve para todos los tokens, pero cada fila tiene una frontera temporal distinta.

Modo ingeniero: la atención causal en código

Todo lo anterior cabe en una función. Con `numpy`, la atención causal es: puntuar con QK^T / d_k , sumar la máscara, aplicar softmax por fila y mezclar con V .

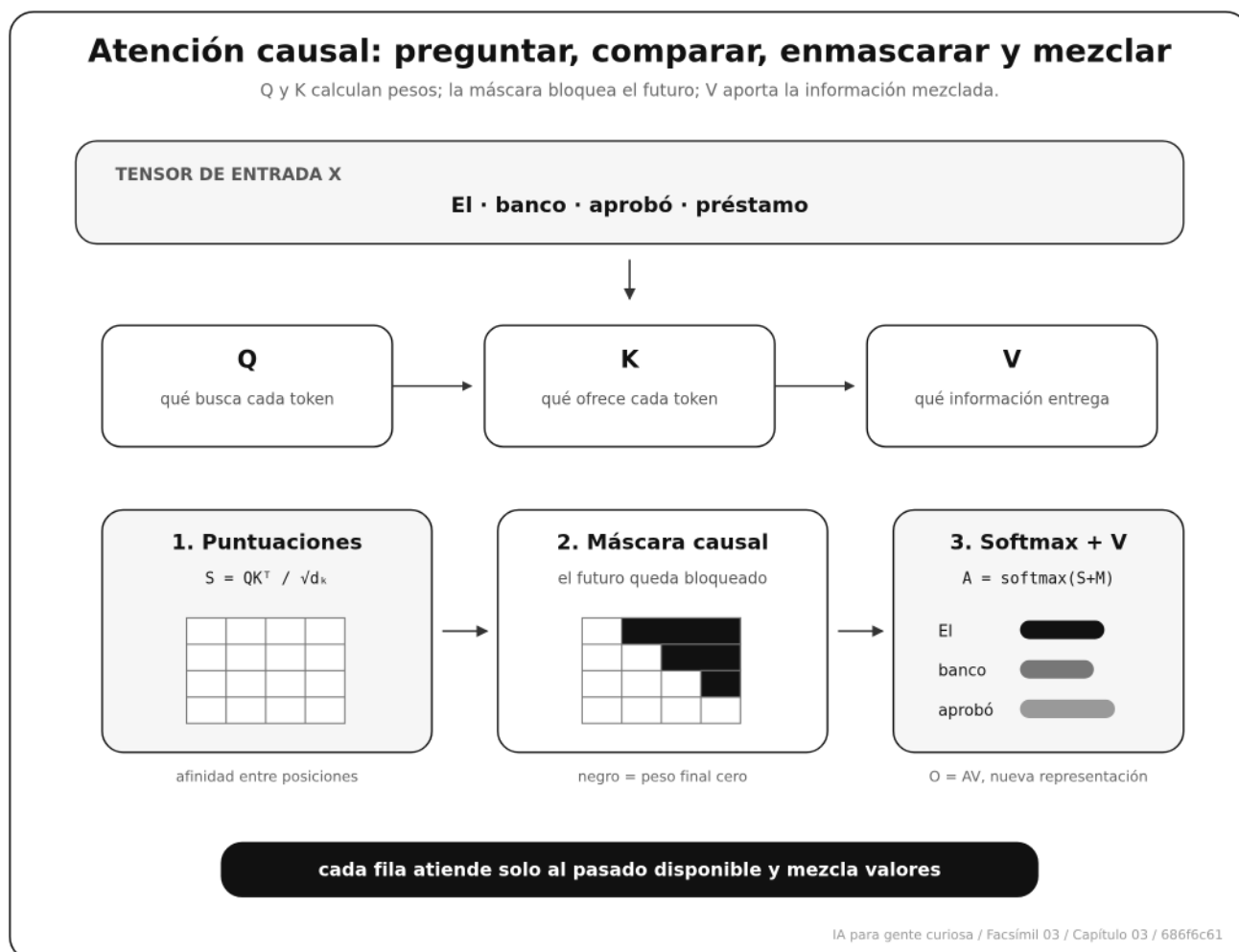
```
import numpy as np

def softmax_filas(S):
    S = S - S.max(axis=1, keepdims=True)          # estabilidad por fila
    e = np.exp(S)
    return e / e.sum(axis=1, keepdims=True)

def atencion_causal(Q, K, V):
    d_k = Q.shape[1]
    S = Q @ K.T / np.sqrt(d_k)                    # puntuaciones n x n
    n = S.shape[0]
    mascara = np.triu(np.full((n, n), -1e9), k=1)   # -1e9 en el triángulo superior
    A = softmax_filas(S + mascara)                  # el futuro queda en 0
    return A @ V                                    # mezcla de valores

# con las Q, K, V del ejemplo, A[1] (banco) sale [0.501, 0.499, 0, 0]
```

`np.triu(..., k=1)` construye la máscara: pone `-1e9` por encima de la diagonal (el futuro) y deja `0` en la diagonal y por debajo (el pasado y el presente). Tras sumar y aplicar softmax, esas posiciones futuras quedan en cero exacto. Es, línea por línea, lo que comprueba el lab de este capítulo: que ninguna fila reparte peso hacia el futuro.



Varias cabezas: mirar relaciones distintas

El Transformer no usa una sola atención. Usa varias cabezas en paralelo:

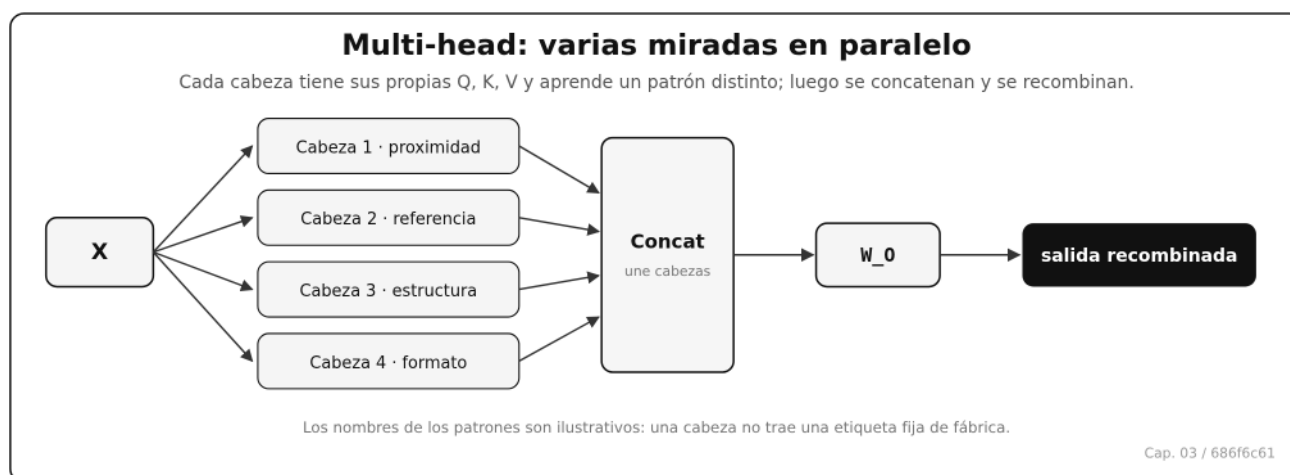
$$\text{head}_h = \text{Attention}(XW_Q^{(h)}, XW_K^{(h)}, XW_V^{(h)})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
h	Índice de la cabeza de atención.	Cabeza 1, cabeza 2, etc.
$W_Q^{(h)}, W_K^{(h)}, W_V^{(h)}$	Proyecciones propias de la cabeza h .	Cada cabeza aprende una mirada distinta.
head_h	Salida de una cabeza.	Una matriz contextual.

Después concatena las cabezas y proyecta:

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \dots, \text{head}_H) W_O$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
H	Número de cabezas.	12, 32, 64... según modelo.
Concat	Concatenación de salidas.	Une las cabezas por dimensión.
W_O	Matriz de salida aprendida.	Recombina lo que miraron las cabezas.



La intuición: una cabeza puede aprender relaciones de proximidad, otra relaciones sintácticas, otra dependencias de nombres, otra patrones de formato. No hay que imaginárselo como roles fijos garantizados, pero sí como capacidad de mirar varios tipos de relación en paralelo.

Recursos como *The Illustrated Transformer* ayudan a ver esta estructura con dibujo.⁷ *The Annotated Transformer* la baja además a código comentado.⁸

Cuando las cabezas comparten claves y valores

La atención multi-cabeza clásica (MHA) da a cada cabeza sus propias Q , K y V . Funciona muy bien, pero tiene un coste oculto que se paga en inferencia: como vimos, K y V se guardan en la KV cache, y con muchas cabezas esa caché se multiplica. De ahí nacen dos variantes que dominan los modelos modernos:

VARIANTE	QUÉ COMPARTEN LAS CABEZAS	EFECTO
MHA (multi-head)	Nada: cada cabeza tiene su Q , K , V .	Máxima expresividad, máxima KV cache.
GQA (grouped-query)	Varias cabezas comparten un mismo par K , V .	Casi la misma calidad con una caché mucho menor.
MQA (multi-query)	Todas las cabezas comparten un único K , V .	Caché mínima, ligeramente menos expresiva.

Las consultas Q siguen siendo independientes en las tres; lo que se comparte son las claves y los valores. Por eso GQA es hoy la opción por defecto en los LLMs de contexto largo: divide la KV cache por el factor de compartición casi sin tocar la calidad. Lo veremos con números en el [capítulo 7](#).

Qué aprende una cabeza de atención

Una cabeza no viene de fábrica con una misión escrita. No podemos decir «esta cabeza es la de fechas» como si fuera una etiqueta estable. Lo que sí podemos decir es que cada cabeza tiene sus propias matrices $W_Q^{(h)}$, $W_K^{(h)}$ y $W_V^{(h)}$, así que dispone de una forma propia de comparar y mezclar.

Eso permite que distintas cabezas se especialicen de manera útil:

PATRÓN QUE PUEDE CAPTURAR	QUÉ SIGNIFICA EN UNA FRASE	POR QUÉ AYUDA
Proximidad	Mirar mucho a tokens cercanos.	Muchas dependencias locales viven cerca: artículo, nombre, puntuación, formato.
Referencia	Conectar una palabra con otra anterior.	Ayuda a mantener entidades: «Marta llegó. Ella...».
Estructura	Detectar patrones de frase o de código.	Un cierre de paréntesis, una coma o una indentación pueden depender de señales previas.
Tema	Mantener pistas semánticas repartidas por el contexto.	Si una conversación habla de medicina, finanzas o cocina, algunas posiciones arrastran ese marco.
Formato	Seguir listas, tablas, JSON o Markdown.	La forma del texto también es información que el modelo usa para continuar.

La clave para el lector es no convertir esto en una caja negra. Una cabeza calcula pesos; varias cabezas calculan varios repartos; la proyección final recombina esos repartos. La complejidad emerge de repetir este patrón muchas veces, no de que una pieza aislada «entienda» el texto como una persona.

Para verlo en pantalla

Aquí conviene jugar un poco:

RECURSO	QUÉ MIRAR EN ESTE CAPÍTULO
Transformer Explainer	Escribe una frase corta y observa cómo la predicción del siguiente token cambia con el contexto. Busca la parte de atención y fíjate en qué posiciones reciben más peso. ²
The Annotated Transformer	Localiza la función de atención y compara el código con la fórmula $\text{softmax}(QK^T / d_k)V$.
The Illustrated Transformer	Repasa el dibujo de Q, K y V después de leer este capítulo.

No intentes verlo todo a la vez. Una buena práctica es abrir el simulador con una frase de cuatro o cinco tokens y preguntar: ¿qué fila estoy mirando?, ¿qué columnas quedan permitidas?, ¿qué pesos suman 1?

En el día a día

Diseñar contexto. Si sabes que cada fila de atención reparte peso entre posiciones disponibles, empiezas a cuidar dónde pones la información importante. No es lo mismo abrir un prompt con instrucciones claras que enterrarlas al final de un bloque enorme.

Entender por qué existe la KV cache. En inferencia, K y V de tokens ya procesados pueden guardarse para no recalcularlos una y otra vez. En el [capítulo 07](#) conectaremos esta idea con memoria, latencia y hardware.

Leer parámetros de generación. Después de la atención y las capas siguientes llegan logits. Softmax reaparece cuando convertimos puntuaciones en probabilidades y elegimos tokens. Métodos como `top_p` y `top_k` se entienden mejor cuando ya has visto qué significa normalizar una distribución.¹⁰

No sobrerreaccionar a un mapa de atención. Mirar pesos de atención puede ser útil, pero no basta para explicar todo un comportamiento. La red tiene muchas capas, cabezas, MLPs y proyecciones. La atención es una ventana técnica, no una explicación completa del sistema.

Por qué debería importarte

Porque Q , K , V y la máscara causal son el punto donde la arquitectura deja de ser una caja negra total. No necesitas memorizar cada matriz, pero sí entender la película: el modelo proyecta, compara, bloquea lo que no debe ver, normaliza y mezcla.

Si entiendes eso, muchas decisiones dejan de sonar arbitrarias: por qué los modelos autoregresivos generan token a token, por qué el contexto largo cuesta, por qué la KV cache importa, por qué el orden del prompt afecta y por qué softmax no significa «verdad», sino distribución de pesos.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Pensar que Q, K y V tienen significado humano fijo	Son proyecciones aprendidas. Una dimensión o una cabeza no trae una etiqueta estable como «sujeto» o «fecha».	Usa las metáforas como andamio, no como definición literal.
Olvidar que softmax se aplica por fila	Cada token reparte su atención entre posiciones permitidas. Si normalizas toda la matriz de golpe, rompes la interpretación.	Pregunta siempre: ¿qué fila estoy leyendo?
No ver la máscara causal	Sin máscara, parece que todos miran a todos. En generación autoregresiva, eso sería mirar tokens futuros.	Dibuja el triángulo inferior de la matriz antes de calcular softmax.
Confundir atención alta con importancia total	Un peso alto indica mezcla en una cabeza y una capa, pero el comportamiento final depende de muchas operaciones.	Lee atención como señal local, no como explicación completa.
Creer que V es menos importante que Q y K	Q y K deciden pesos, pero V contiene lo que se mezcla. Sin V, no hay salida contextual.	Recita la película: comparar con QK^T , normalizar con softmax, mezclar con V.

Cómo encaja todo

Este mapa coloca Q, K, V y la máscara causal en la cadena completa. El capítulo anterior entrega el tensor; este capítulo decide qué posiciones pueden mirarse, cómo se normalizan los pesos y qué información se mezcla.

La conexión hacia delante es igual de importante: las mismas claves y valores que aquí parecen una fórmula serán memoria real en la KV cache, coste real en serving y criterio real cuando diseñemos contexto, RAG o evaluación.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Query (Q)	Proyección que representa lo que una posición busca en el contexto.
Key (K)	Proyección que representa lo que una posición ofrece para ser encontrada.
Value (V)	Proyección que contiene la información que se mezclará al final.
Producto punto	Suma de productos entre dos vectores; mide afinidad en atención.
Máscara causal	Regla que pone peso cero a posiciones futuras durante generación autoregresiva.
Softmax por fila	Normalización de cada fila de puntuaciones para convertirla en pesos que suman 1.
Cabeza de atención	Una atención independiente con sus propias proyecciones Q , K y V .
Multi-head attention	Varias cabezas trabajando en paralelo y recombinadas al final.
KV cache	Memoria que guarda las claves y valores del pasado para no recalcularlos en cada token generado.
GQA	Variante en la que varias cabezas comparten un mismo par de claves y valores, reduciendo la KV cache.

Antes de pasar página

- ☐ ¿Puedo explicar Q , K y V sin decir solo «consulta, clave y valor»? (Si no, vuelve a «Qué sí son Q , K y V ».)
- ☐ ¿Entiendo por qué QK^T produce una matriz $n \times n$? (Si no, vuelve a «Puntuaciones».)
- ☐ ¿Puedo escribir la máscara causal y explicar qué significa $j > i$? (Si no, vuelve a «La máscara causal».)
- ☐ ¿Entiendo por qué la máscara suma $-\infty$ antes del softmax, y por qué en código se usa -10^9 ? (Si no, vuelve a «La máscara causal».)
- ☐ ¿Distingo self-attention, self-attention enmascarada y cross-attention? (Si no, vuelve a «La máscara causal».)
- ☐ ¿Sé por qué softmax se aplica por fila? (Si no, vuelve a «La máscara causal».)

- ☐ ¿Puedo explicar por qué $O = AV$ mezcla información? (Si no, vuelve a «Valores».)
- ☐ ¿Puedo recorrer el circuito $Q \rightarrow K \rightarrow \text{máscara} \rightarrow \text{softmax} \rightarrow V$ sin saltarme pasos? (Si no, vuelve a «Cómo leer la matriz sin perderse».)
- ☐ ¿Entiendo por qué BERT y GPT no miran el texto de la misma forma? (Si no, vuelve a «La máscara causal».)
- ☐ ¿Sé explicar por qué aparecen ceros en la parte derecha de cada fila de la matriz de atención? (Si no, vuelve a «La máscara causal».)
- ☐ ¿Entiendo por qué la atención cuesta n^2 y cómo la KV cache lo alivia en generación? (Si no, vuelve a «Cuánto cuesta esta operación».)
- ☐ ¿Distingo MHA, GQA y MQA y por qué comparten claves y valores? (Si no, vuelve a «Cuando las cabezas comparten claves y valores».)
- ☐ ¿Sé conectar K y V con la KV cache? (Si no, vuelve a «De Q, K, V a la KV cache».)

En resumen

IDEA FUERZA	DETALLE
Q, K y V son tres proyecciones aprendidas de la misma entrada.	Q busca, K permite comparar y V aporta la información que se mezcla.
La atención compara todos los pares permitidos de posiciones.	QK^T / d_k produce puntuaciones; softmax las convierte en pesos.
La máscara causal hace honesta la generación.	Una posición no puede mirar tokens futuros, así que esos pesos acaban siendo cero.
Multi-head attention permite varias miradas en paralelo.	Distintas cabezas pueden aprender patrones de relación diferentes y luego recombinarse.

Para saber más

Alammar, J. (2018). *The Illustrated Transformer*. <https://jalammar.github.io/illustrated-transformer/>

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

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/1457c0d6bfc4967418bfb8ac142f64a-Abstract.html>

Cho, A., Kim, G. C., Karpekov, A., Helbling, A., Wang, Z. J., Lee, S., Hoover, B. y Chau, D. H. (2025). Transformer Explainer: interactive learning of text-generative models. En *Proceedings of the AAAI Conference on Artificial Intelligence*.
<https://ojs.aaai.org/index.php/AAAI/article/download/35347/37502>

Devlin, J., Chang, M. W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. En *Proceedings of NAACL-HLT* (pp. 4171-4186). <https://doi.org/10.18653/v1/N19-1423>

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

Holtzman, A., Buys, J., Du, L., Forbes, M. y Choi, Y. (2020). The curious case of neural text degeneration. En *International Conference on Learning Representations*.
<https://openreview.net/forum?id=rygGQyrFvH>

Rush, A. M. (2018). *The Annotated Transformer*. <https://nlp.seas.harvard.edu/annotated-transformer/>

Vert, A. (2026, 13 de mayo). *AI 101: Your Ultimate Guide to Attention: Mechanism, QKV, and KV Cache*. Turing Post. <https://www.turingpost.com/p/your-ultimate-guide-to-attention-mechanism-qkv-and-kv-cache>

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. 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>. El paper define la atención como $\text{softmax}(QK^T/d_k)V$, que es la fórmula que vamos a desmenuzar aquí. ↵
2. Vert, A. (2026, 13 de mayo). *AI 101: Your Ultimate Guide to Attention: Mechanism, QKV, and KV Cache*. Turing Post. <https://www.turingpost.com/p/your-ultimate-guide-to-attention-mechanism-qkv-and-kv-cache>. Consultado el 10 de junio de 2026. ↵
3. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. <https://www.deeplearningbook.org>. Las capas lineales y las composiciones de funciones diferenciables forman la base matemática de las redes profundas. ↵

4. Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer. Bishop explica softmax como transformación de puntuaciones en probabilidades normalizadas, una pieza común en modelos probabilísticos y clasificación. ↵
5. 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/1457c0d6bfc4967418bfb8ac142f64a-Abstract.html>. GPT-3 es un ejemplo de Transformer decoder autoregresivo a gran escala. ↵
6. Devlin, J., Chang, M. W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. En *Proceedings of NAACL-HLT* (pp. 4171-4186). <https://doi.org/10.18653/v1/N19-1423>. BERT popularizó el uso de Transformer encoder bidireccional para comprensión de lenguaje. ↵
7. Alammr, J. (2018). *The Illustrated Transformer*. <https://jalammar.github.io/illustrated-transformer/>. Recurso visual para seguir el flujo de Q, K, V y multi-head attention. ↵
8. Rush, A. M. (2018). *The Annotated Transformer*. <https://nlp.seas.harvard.edu/annotated-transformer/>. Implementación anotada del Transformer original con código y explicación matemática. ↵
9. Cho, A., Kim, G. C., Karpekov, A., Helbling, A., Wang, Z. J., Lee, S., Hoover, B. y Chau, D. H. (2025). Transformer Explainer: interactive learning of text-generative models. En *Proceedings of the AAAI Conference on Artificial Intelligence*. <https://ojs.aaai.org/index.php/AAAI/article/download/35347/37502>. La herramienta permite visualizar componentes de un GPT-2 pequeño en navegador. ↵
10. Holtzman, A., Buys, J., Du, L., Forbes, M. y Choi, Y. (2020). The curious case of neural text degeneration. En *International Conference on Learning Representations*. <https://openreview.net/forum?id=rygGQyrFvH>. El artículo analiza cómo distintas estrategias de muestreo afectan a la calidad del texto generado. ↵

Capítulo 04: MLP, residual, LayerNorm, logits y sampling

Después de mirar, hay que pensar y decidir

En el capítulo anterior abrimos la atención por dentro: Q , K , V , máscara causal y softmax. Vimos cómo cada token mezcla información de posiciones permitidas. Pero un bloque Transformer no termina ahí.

Después de la atención queda trabajo: estabilizar números, dejar pasar información antigua, transformar cada posición con una red interna y, al final de muchas capas, convertir el último vector en una lista enorme de puntuaciones para tokens posibles.

Este capítulo va de ese tramo menos vistoso y muy importante. La atención decide **de dónde** tomar información. El MLP ayuda a transformar **qué significa** esa información. Las conexiones residuales permiten que las capas se apilen sin destruir lo anterior. LayerNorm mantiene los números en una zona manejable. Los logits y el sampling convierten todo eso en una palabra, un signo, un salto de línea o un fragmento de código.

Qué no es esta parte del Transformer

El MLP no es una memoria externa. No guarda documentos ni hechos como una base de datos. Es una transformación aprendida que se aplica a cada posición del tensor.

LayerNorm tampoco es una limpieza semántica del texto. No corrige ideas ni comprueba si algo es cierto. Normaliza números para que el entrenamiento y la inferencia sean más estables.

Y sampling no es una votación de verdad. Cuando el modelo elige el siguiente token, está usando una distribución de probabilidad lingüística. Que un token tenga mucha probabilidad significa que encaja con el contexto y con lo aprendido, no que la frase resultante sea necesariamente cierta.

Qué sí ocurre dentro de un bloque

Un bloque Transformer moderno suele seguir una idea parecida a esta:

```
entrada
-> normalización
-> atención
-> suma residual
-> normalización
-> MLP
-> suma residual
salida
```

El Transformer original ya combinaba atención, red feed-forward, normalización y conexiones residuales.¹ Muchas implementaciones actuales usan una variante llamada **pre-norm**: normalizan antes de la atención y antes del MLP.

Una forma compacta de escribirlo es:

$$a_l = x_l + \text{Attention}(\text{LN}(x_l))$$

$$x_{l+1} = a_l + \text{MLP}(\text{LN}(a_l))$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
x_l	Entrada del bloque l .	Tensor después del bloque anterior.
a_l	Estado intermedio después de atención y residual.	Lo que sale antes del MLP.
x_{l+1}	Salida del bloque.	Entrada del siguiente bloque.
LN	LayerNorm.	Normaliza cada vector de posición.
Attention	Atención causal vista en el capítulo 03.	Q, K, V, máscara y softmax.
MLP	Red interna aplicada por posición.	Dos o tres proyecciones con una activación.

Cómo se lee esto sin saber matemáticas. La fórmula se lee de dentro hacia afuera, como una receta de tres pasos. Tomemos la primera línea, $a_l = x_l + \text{Attention}(\text{LN}(x_l))$:

1. $\text{LN}(x_l)$: coge lo que llega al bloque, x_l , y ordena sus números a una escala estable. Es como nivelar el volumen antes de mezclar.
2. $\text{Attention}(\dots)$: sobre esa versión ordenada, deja que cada token mire a los demás y recoja contexto.

3. $x_l + \dots$: suma ese resultado a lo que ya había, **sin borrarlo**. Ese total es a_l .

La segunda línea repite la misma jugada, pero con el MLP en lugar de la atención: normaliza a_l , lo pasa por la red interna y vuelve a sumar. El resultado, x_{l+1} , es lo que entra al siguiente bloque.

El símbolo que conviene no perder de vista es el $+$: el bloque **nunca reemplaza** lo que venía, siempre **añade** una corrección encima. Esa suma (la conexión residual) es una de las razones por las que se pueden apilar muchas capas sin que la información se pierda por el camino.

Pre-norm o post-norm: dónde va la normalización

Te habrás fijado en un detalle de las fórmulas: la LayerNorm está **dentro** del paréntesis, antes de la atención y antes del MLP. Eso es **pre-norm**, y no siempre fue así. El Transformer original normalizaba **después** de cada subcapa (*post-norm*): $x_{l+1} = \text{LN}(x_l + \text{Attention}(x_l))$. Funcionaba, pero al apilar muchas capas el entrenamiento se volvía inestable y exigía calentar el *learning rate* con cuidado. El pre-norm cambió eso: al normalizar la entrada de cada subcapa y dejar la vía residual **limpia** (sin normalizar), el gradiente fluye sin obstáculos desde la última capa hasta la primera. Por eso casi todos los LLMs profundos de hoy son pre-norm: es lo que hace viable entrenar decenas o cientos de capas sin que los números se descontrolen.

Un ejemplo entendible: autocompletar un mensaje

Imagina que escribes en una aplicación de correo:

Nos vemos mañana a las

El modelo no piensa en castellano con una libreta al lado. Lo que tiene es una representación numérica del contexto y debe decidir el siguiente token. Aun así, podemos traducir las piezas del bloque a una historia bastante humana:

PIEZA	EN EL EJEMPLO	QUÉ APORTA
Atención	Mira «vemos», «mañana» y «a las».	Recupera que probablemente viene una hora.
Residual	Conserva la frase completa mientras añade señales nuevas.	No pierde que estamos hablando de una cita.
LayerNorm	Pone las señales numéricas en una escala comparable.	Evita que una dimensión grande domine solo por tamaño.
MLP	Transforma la representación de la última posición.	Refuerza patrones como «a las» → hora o número.
Logits	Asigna puntuaciones a tokens candidatos.	«10», «9», «reunión», «azul» reciben valores distintos.
Sampling	Elige el siguiente token según la distribución y la configuración.	Puede elegir «10» de forma conservadora o explorar alternativas.

Supongamos que, tras muchas capas, la cabeza de lenguaje produce estos logits:

TOKEN CANDIDATO	LOGIT	LECTURA INFORMAL
10	4,2	Muy compatible con «a las».
9	3,9	También muy compatible.
reunión	1,1	Tiene relación con el contexto, pero encaja peor justo después de «a las».
azul	-0,7	Gramatical y semánticamente flojo aquí.

Si usamos temperatura baja, el sistema tenderá a elegir entre «10» y «9». Si subimos la temperatura o abrimos mucho `top_p`, aumenta la posibilidad de tokens menos esperados. Esto no vuelve al modelo más profundo; cambia el margen de exploración.

Este ejemplo resume el capítulo entero: el bloque procesa la frase, los logits proponen continuaciones y el sampling decide cuánto riesgo aceptamos al elegir una.

Residual: dejar una vía abierta

Una conexión residual suma la entrada original con una transformación:

$$y = x + F(x)$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
x	Representación que llega a una subcapa.	Vector de un token antes de la atención.
$F(x)$	Transformación aprendida.	Atención o MLP.
y	Resultado después de sumar entrada y transformación.	Vector actualizado.

La idea se popularizó con ResNet en visión por computador: una red profunda entrena mejor si cada bloque puede aprender una corrección sobre lo que ya venía circulando.²

En un LLM, la intuición práctica es esta:

SIN RESIDUAL	CON RESIDUAL
Cada subcapa tiene que reconstruir todo lo útil.	Cada subcapa puede añadir una corrección.
Es más fácil degradar información anterior.	La representación anterior sigue disponible.
Entrenar muchas capas se vuelve más frágil.	Apilar capas profundas es más viable.

Ejemplo pequeño:

$$x = [2, -1, 0, 5]$$

$$F(x) = [0, 1, 0, 4, -0, 2]$$

$$y = x + F(x) = [2, 1, -0, 6, 0, 3]$$

La transformación ha modificado el vector, pero no lo ha sustituido por completo. Ese matiz importa mucho en redes profundas.

LayerNorm: poner los números en una escala manejable

LayerNorm normaliza cada vector de activaciones usando su propia media y desviación típica.³ Para un vector x , calculamos:

$$\mu = \frac{1}{d} \sum_{i=1}^d x_i$$

$$\sigma = \frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2 + \epsilon$$

$$\text{LN}(x)_i = \gamma_i \frac{x_i - \mu}{\sigma} + \beta_i$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x_i	Componente i del vector.	$x_2 = -1$.
d	Dimensión del vector.	$d = 3$.
μ	Media del vector.	Media de $[2, -1, 0,5]$.
σ	Desviación típica con estabilidad numérica.	Incluye ϵ .
ϵ	Número pequeño para evitar división por cero.	10^{-5} .
γ_i	Escala aprendida.	Permite ajustar la salida normalizada.
β_i	Sesgo aprendido.	Permite desplazar la salida.

Si tomamos:

$$x = [2, -1, 0,5]$$

La media es:

$$\mu = \frac{2 - 1 + 0,5}{3} = 0,5$$

Las diferencias respecto a la media son:

$$[1,5, -1,5, 0]$$

La desviación típica, ignorando ϵ para leer el ejemplo, es:

$$\sigma = \frac{1,5^2 + (-1,5)^2 + 0^2}{3} \approx 1,225$$

Si $\gamma = [1, 1, 1]$ y $\beta = [0, 0, 0]$, la salida aproximada es:

$$\text{LN}(x) \approx [1,225, -1,225, 0]$$

LayerNorm no cambia el orden de las ideas por sí sola. Cambia la escala de los números para que las siguientes operaciones trabajen en una zona más estable.

La versión moderna: RMSNorm

Muchos LLMs recientes (LLaMA, Mistral, Qwen y compañía) usan una variante más simple, **RMSNorm**.⁴ La idea: la parte cara, y según se vio poco necesaria, de LayerNorm es restar la media. RMSNorm se la salta y normaliza solo por la magnitud del vector, su raíz cuadrática media:

$$\text{RMSNorm}(x)_i = \gamma_i \frac{x_i}{\sqrt{\frac{1}{d} \sum_{j=1}^d x_j^2 + \epsilon}}$$

No resta μ ni usa β : solo reescala. Hace casi lo mismo que LayerNorm, con menos operaciones y, por tanto, algo más rápido en entrenamiento e inferencia. Es un buen ejemplo de cómo los modelos modernos recortan piezas que costaban más de lo que aportaban.

MLP: transformar cada posición por dentro

Después de la atención, el bloque Transformer aplica una red feed-forward por posición. En muchos textos se llama MLP, aunque técnicamente es una red pequeña aplicada a cada token por separado.

Una versión clásica es:

$$\text{MLP}(x) = W_2 \phi(W_1 x + b_1) + b_2$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Vector de una posición.	Representación contextual de «banco».
W_1	Primera matriz aprendida.	Expande la dimensión interna.
b_1	Sesgo de la primera proyección.	Vector sumado antes de la activación.
ϕ	Función de activación.	GELU, ReLU o una variante con puerta.
W_2	Segunda matriz aprendida.	Devuelve la dimensión al tamaño del modelo.
b_2	Sesgo final.	Ajuste aprendido de salida.

Las redes profundas usan activaciones no lineales para poder representar funciones más ricas que una sola multiplicación de matrices.⁵ En Transformers modernos, GELU se hizo muy común en modelos de lenguaje.⁶ También hay variantes con puertas, como GLU y SwiGLU, que permiten modular qué información pasa.⁷

Dónde viven los parámetros: el MLP se ensancha

Hay un detalle de tamaño que sorprende: la matriz W_1 no mantiene la dimensión, la **expande**, normalmente a $4 \times d_{\text{model}}$, y W_2 la devuelve. Es decir, dentro de cada token el MLP trabaja en un espacio cuatro veces más ancho antes de volver. Esa expansión es cara: en un Transformer denso, **alrededor de dos tercios de los parámetros viven en los MLP**, no en la atención. Por eso, cuando en el capítulo 1 estimamos el tamaño de un modelo con $N \approx 12 \cdot L \cdot d^2$, buena parte de ese 12 es el MLP.

Por qué GELU, y no solo ReLU

La activación ϕ es la pieza que da la no-linealidad. La clásica es ReLU (deja pasar lo positivo y corta lo negativo a cero), pero los LLMs adoptaron **GELU**, una versión suave que no corta en seco sino que atenúa gradualmente los valores cercanos a cero; esa suavidad ayuda al gradiente. Y los modelos más recientes usan **SwiGLU**, una activación *con puerta*: en vez de aplicar una sola función, multiplica dos proyecciones, una de las cuales actúa como compuerta que decide cuánta señal deja pasar. A cambio de algo más de cómputo, suele mejorar la calidad, y por eso es la opción por defecto en muchos LLMs abiertos modernos.

La intuición:

PIEZA	QUÉ HACE
Atención	Mezcla información entre posiciones.
MLP	Transforma cada posición internamente.
Residual	Conserva una vía para lo que ya estaba.
LayerNorm	Mantiene los números en una escala razonable.

Si la atención es una mesa de mezclas entre tokens, el MLP es el taller interno de cada token. No decide a qué otros tokens mirar; trabaja con lo que ya recibió.

Del último vector a los logits

Tras repetir muchos bloques, el modelo tiene una representación final. Para predecir el siguiente token, toma el vector de la última posición y lo proyecta al tamaño del vocabulario:

$$z = hW_{\text{vocab}} + b$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
h	Vector final de la última posición.	Representación de todo el contexto disponible.
W_{vocab}	Matriz que lleva de dimensión interna a vocabulario.	De d_{model} a 50 000 tokens.
b	Sesgo de vocabulario.	Una puntuación adicional por token.
z	Vector de logits.	Una puntuación por token candidato.

Un logit no es una probabilidad. Puede ser negativo, positivo, grande o pequeño. Para convertir logits en una distribución usamos softmax, igual que vimos en el [capítulo 01](#) y en el [capítulo 03](#):

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^V e^{z_j}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
p_i	Probabilidad asignada al token i .	$p_{\text{Madrid}} = 0,256$.
z_i	Logit del token i .	$z_{\text{París}} = 4,0$.
V	Tamaño del vocabulario.	50 000 tokens candidatos.
$\sum_{j=1}^V$	Suma sobre todos los tokens.	Normaliza para que las probabilidades sumen 1.

Bishop presenta softmax como una transformación estándar de puntuaciones en probabilidades normalizadas.⁸ En LLMs, esa transformación no juzga la verdad de una frase: convierte puntuaciones de continuación en una distribución de tokens.

Temperatura: abrir o cerrar el abanico

La temperatura modifica los logits antes de softmax:

$$p_i(T) = \frac{e^{z_i/T}}{\sum_{j=1}^V e^{z_j/T}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
T	Temperatura.	$T = 0,5, T = 1, T = 2.$
z_i/T	Logit ajustado por temperatura.	Si $T < 1$, las diferencias crecen.
$p_i(T)$	Probabilidad después de aplicar temperatura.	Distribución más concentrada o más suave.

Supongamos cuatro tokens candidatos:

TOKEN	LOGIT
París	4,0
Madrid	3,0
Lyon	1,0
azul	0,0

Con $T = 1$:

TOKEN	PROBABILIDAD APROX.
París	0,696
Madrid	0,256
Lyon	0,035
azul	0,013

Con $T = 0,5$, las diferencias se agrandan:

TOKEN	PROBABILIDAD APROX.
París	0,879
Madrid	0,119
Lyon	0,002
azul	0,000

Con $T = 2$, las diferencias se suavizan:

TOKEN	PROBABILIDAD APROX.
París	0,509
Madrid	0,309
Lyon	0,114
azul	0,069

Temperatura baja no significa «más inteligente». Significa más conservadora en la elección del siguiente token. Temperatura alta no significa «mejor creatividad». Significa más dispersión en la distribución. En producto, esto se decide según la tarea: extracción estructurada, redacción, lluvia de ideas, código, resumen o conversación.

Top-k y top-p: limitar candidatos

Además de temperatura, se suelen aplicar reglas para limitar el conjunto de tokens candidatos.

Top-k conserva los k tokens con mayor probabilidad y pone el resto a cero:

$$C_k = \text{TopK}(p, k)$$

$$p'_i = \begin{cases} \frac{p_i}{\sum_{j \in C_k} p_j} & \text{si } i \in C_k \\ 0 & \text{si } i \notin C_k \end{cases}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
C_k	Conjunto de tokens conservados.	Con $k = 2$: París y Madrid.
p'_i	Probabilidad renormalizada.	Probabilidad después de eliminar candidatos.
$i \notin C_k$	Token descartado por top-k.	Lyon y azul reciben 0.

Top-p, también llamado nucleus sampling, conserva el conjunto mínimo de tokens cuya probabilidad acumulada alcanza un umbral p . Holtzman y coautores popularizaron esta estrategia para reducir degeneraciones en generación abierta.⁹

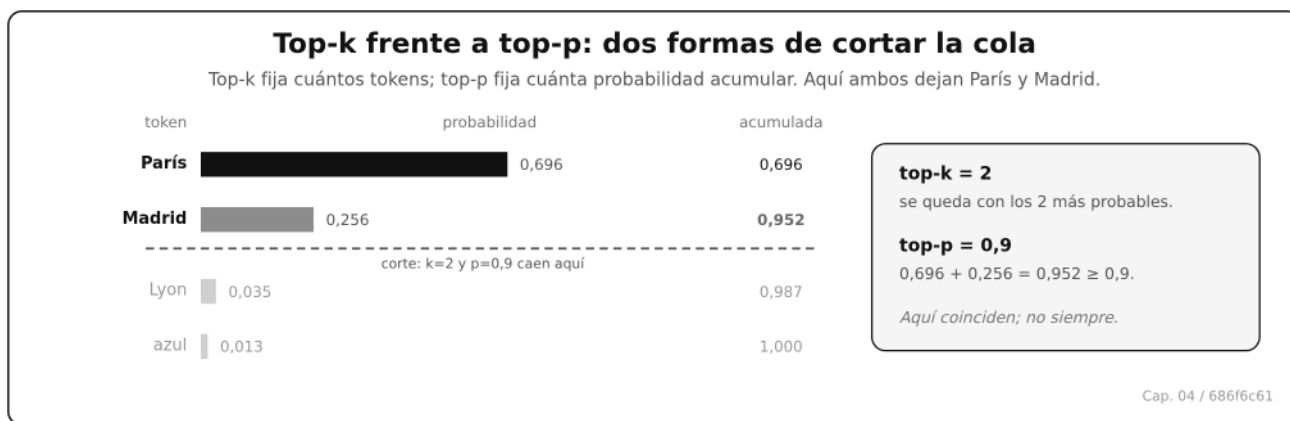
Si ordenamos de mayor a menor:

TOKEN	PROBABILIDAD	ACUMULADA
París	0,696	0,696
Madrid	0,256	0,952
Lyon	0,035	0,987
azul	0,013	1,000

Con $p = 0,9$, top-p conserva París y Madrid, porque con esos dos ya alcanza 0,952. Después renormaliza solo dentro de ese conjunto.

La diferencia importante:

REGLA	QUÉ FIJA	QUÉ CAMBIA
Top-k	Número de candidatos.	La masa de probabilidad conservada.
Top-p	Masa de probabilidad mínima.	El número de candidatos.



En una distribución muy concentrada, top-p puede conservar pocos tokens. En una distribución muy plana, puede conservar muchos. Esa elasticidad es la gracia.

El orden de los mandos, y qué viene después

En la práctica estos controles se aplican en cadena, y el orden importa: primero la **temperatura** reescala los logits, después **top-k** o **top-p** recortan la cola, y al final se muestra de lo que queda. Calentar y luego recortar no da el mismo resultado que recortar y luego calentar; por eso conviene saber qué hace tu librería por debajo.

Y top-k y top-p no son el final de la historia. Estrategias más recientes como **min-p** ajustan el corte a la *forma* de la distribución: en vez de un número fijo o una masa fija, descartan los tokens cuya probabilidad cae por debajo de una fracción de la del token más probable. La idea de fondo es siempre la misma: decidir cuánta cola de la distribución dejamos viva antes de tirar el dado.

Modo ingeniero: LayerNorm y los mandos en código

Todas estas piezas son cortas en `numpy`. Aquí están LayerNorm, el softmax con temperatura y el recorte top-p:

```

import numpy as np

def layernorm(x, gamma, beta, eps=1e-5):
    mu = x.mean()
    sigma = np.sqrt(((x - mu) ** 2).mean() + eps)
    return gamma * (x - mu) / sigma + beta

def softmax_temp(logits, T=1.0):
    z = logits / T
    z = z - z.max()          # estabilidad
    e = np.exp(z)
    return e / e.sum()

def top_p(probs, p=0.9):
    orden = np.argsort(probs)[::-1]          # de mayor a menor
    acum = np.cumsum(probs[orden])
    corte = np.searchsorted(acum, p) + 1     # cuántos tokens hasta alcanzar p
    keep = orden[:corte]
    out = np.zeros_like(probs)
    out[keep] = probs[keep]
    return out / out.sum()                  # renormaliza dentro del núcleo

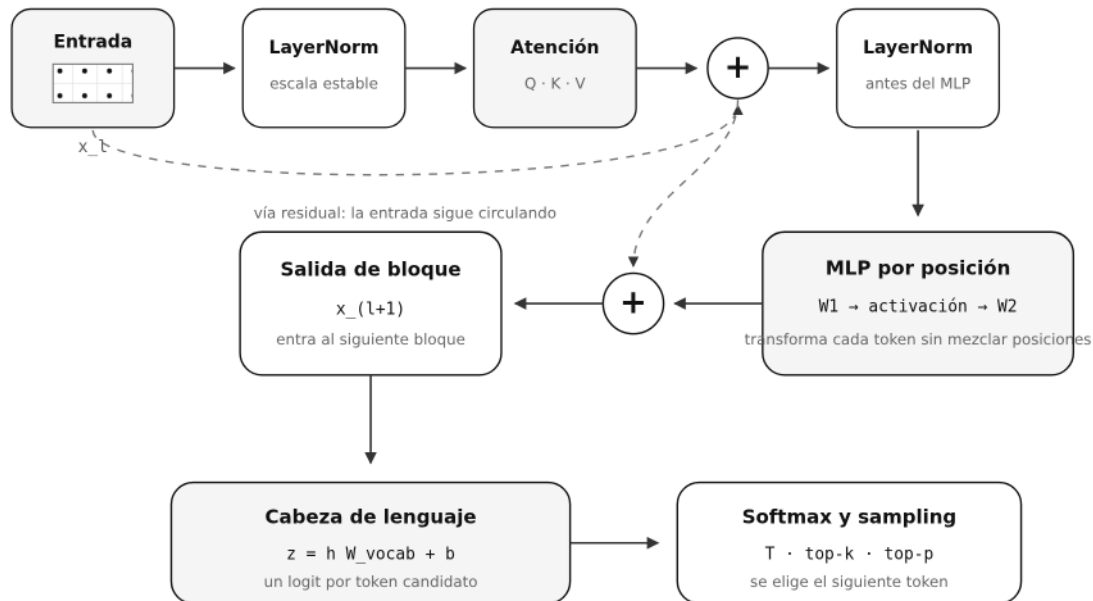
logits = np.array([4.0, 3.0, 1.0, 0.0])      # París, Madrid, Lyon, azul
p = softmax_temp(logits, T=1.0)             # [0.696, 0.256, 0.035, 0.013]
print(top_p(p, 0.9).round(3))               # [0.731 0.269 0. 0.]

```

`layernorm` con `x=[2, -1, 0.5]` devuelve `[1.225, -1.225, 0.0]`, justo el ejemplo de antes; y `top_p` con `p=0.9` deja a París y Madrid y renormaliza entre ellos. Es exactamente lo que aísla el lab de este capítulo: cada mando por separado, para ver qué cambia.

Del bloque Transformer al siguiente token

La atención mezcla contexto; residual, normalización y MLP refinan; logits y sampling deciden la continuación.



texto → capas → logits → distribución → token

IA para gente curiosa / Facsímil 03 / Capítulo 04 / 686f6c61

Simuladores para verlo en pantalla

Dos recursos ayudan especialmente aquí:

RECURSO	QUÉ MIRAR
LLM Visualization	Recorre las capas y localiza el paso desde bloques internos a logits. ¹⁰
Transformer Explainer	Cambia texto de entrada y observa cómo varía la distribución del siguiente token. ¹¹

Úsalos con una pregunta concreta: ¿dónde se convierte el vector final en logits?, ¿dónde aparece softmax?, ¿qué cambia si modifico el texto anterior?

En el día a día

Cuando quieres salida estable. Para extracción JSON, clasificación o tareas con formato rígido, normalmente quieres menos variación: temperatura baja, pocos candidatos y validación posterior. No porque el modelo «sepa más», sino porque reduces el abanico de continuaciones posibles.

Cuando quieres explorar. Para escritura creativa, alternativas de copy o lluvia de ideas, puedes permitir más variedad. Ahí temperatura y top-p se convierten en mandos de exploración. Conviene medir resultados, no elegir parámetros por superstición.

Cuando depuras una respuesta extraña. Si una salida cambia mucho entre ejecuciones, no siempre es un problema de prompt. Puede haber temperatura alta, top-p amplio o muestreo activado. Antes de cambiar de modelo, mira la configuración de generación.

Cuando eliges arquitectura. MLP, normalización y residuales parecen detalles internos, pero afectan a estabilidad, velocidad, memoria y calidad. En el [capítulo 05](#) veremos cómo modelos modernos cambian estas piezas: MoE, activaciones con puerta y variaciones de arquitectura.

Por qué debería importarte

Porque esta parte une dos mundos: arquitectura interna y comportamiento visible. El lector ve texto, pero por debajo hay logits, temperatura, filtros de candidatos y muestreo. Entender esa cadena evita explicaciones vagas.

Si un modelo responde siempre igual, puede ser por una distribución muy concentrada o por una configuración conservadora. Si responde con demasiada dispersión, quizá la temperatura o top-p están abriendo demasiado el abanico. Si un modelo grande cuesta mucho, parte del coste está en repetir bloques con atención, MLP y proyecciones a vocabulario miles de veces por segundo.

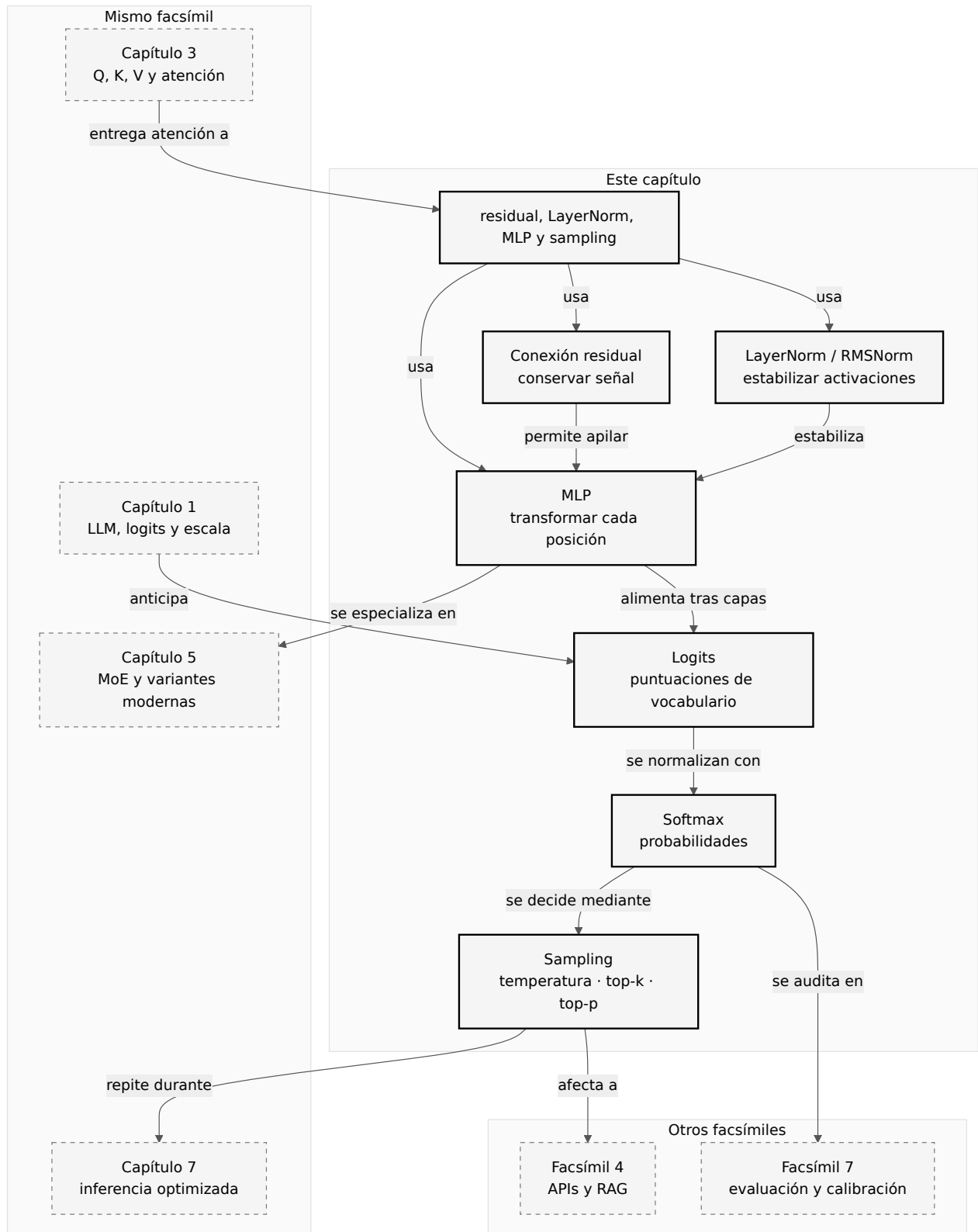
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Pensar que LayerNorm entiende el texto	Normaliza números; no valida significado.	Pregunta siempre si una pieza trabaja con escala numérica o con decisión de contenido.
Confundir logit con probabilidad	Un logit es una puntuación cruda; puede ser negativa y no suma nada.	No hables de probabilidad hasta aplicar softmax.
Creer que temperatura baja equivale a verdad	Solo concentra la distribución en tokens más probables.	Para verdad factual, usa contexto, fuentes y validación.
Olvidar que el MLP no mezcla posiciones	La mezcla entre tokens la hizo la atención; el MLP transforma cada posición.	Repite: atención mezcla entre posiciones, MLP transforma dentro de una posición.
Usar top-k y top-p como si fueran lo mismo	Top-k fija número de candidatos; top-p fija masa acumulada.	Mira una tabla ordenada de probabilidades y calcula ambos a mano una vez.

Cómo encaja todo

Este mapa enseña el tramo que suele quedar invisible entre “atención” y “respuesta”. La atención entrega una representación contextual; residual, LayerNorm y MLP la estabilizan y transforman; logits, softmax y sampling la convierten en comportamiento visible.

Por eso el capítulo conecta arquitectura con producto: temperatura, `top_k` y `top_p` no son adornos de API, sino controles sobre una distribución que nace de todo el bloque Transformer.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Conexión residual	Suma que conserva la entrada y añade una transformación aprendida.
LayerNorm	Normalización que usa media y desviación típica dentro de cada vector.
RMSNorm	Normalización que reescala por la raíz cuadrática media del vector, sin restar la media.
Pre-norm	Aplicar la normalización antes de cada subcapa, dejando limpia la vía residual.
MLP	Red interna aplicada por posición después de la atención.
Activación	Función no lineal que permite representar transformaciones más ricas.
SwiGLU	Activación con puerta usada en los MLP modernos; multiplica dos proyecciones para modular la señal.
Logit	Puntuación cruda de un token antes de softmax.
Temperatura	Parámetro que concentra o suaviza la distribución de salida.
Top-k	Muestreo que conserva solo los k candidatos principales.
Top-p	Muestreo que conserva candidatos hasta alcanzar una probabilidad acumulada.
Sampling	Proceso de elegir el siguiente token a partir de una distribución.

Antes de pasar página

- ☐ ¿Puedo explicar por qué una conexión residual suma en vez de reemplazar? (Si no, vuelve a «Residual: dejar una vía abierta».)
- ☐ ¿Sé calcular la media y desviación típica de una LayerNorm pequeña? (Si no, vuelve a «LayerNorm».)
- ☐ ¿Distingo pre-norm de post-norm y sé qué cambia RMSNorm frente a LayerNorm? (Si no, vuelve a «Pre-norm o post-norm» y «La versión moderna: RMSNorm».)
- ☐ ¿Entiendo por qué el MLP transforma cada posición pero no mezcla tokens? (Si no, vuelve a «MLP».)

- ☐ ¿Sé por qué el MLP se ensancha a $4d$ y por qué GELU o SwiGLU en vez de solo ReLU? (Si no, vuelve a «Dónde viven los parámetros» y «Por qué GELU».)
- ☐ ¿Puedo distinguir logit, probabilidad y token elegido? (Si no, vuelve a «Del último vector a los logits».)
- ☐ ¿Puedo explicar el ejemplo de «Nos vemos mañana a las» usando residual, LayerNorm, MLP, logits y sampling? (Si no, vuelve a «Un ejemplo entendible».)
- ☐ ¿Sé qué cambia cuando bajo o subo la temperatura? (Si no, vuelve a «Temperatura».)
- ☐ ¿Puedo calcular top-k y top-p con una tabla de cuatro tokens? (Si no, vuelve a «Top-k y top-p».)
- ☐ ¿Sé en qué orden se aplican temperatura, top-k y top-p, y qué hace min-p? (Si no, vuelve a «El orden de los mandos».)
- ☐ ¿Sé qué parámetro de muestreo tocar y cómo cambia la salida? (Si no, vuelve a «Temperatura» y «Top-k y top-p».)
- ☐ ¿Puedo conectar estos mandos con APIs, agentes y evaluación futura? (Si no, vuelve a «Cómo encaja todo».)

En resumen

IDEA FUERZA	DETALLE
Un bloque Transformer no es solo atención.	Residual, LayerNorm y MLP hacen que la información circule, se estabilice y se transforme.
Los logits son puntuaciones, no probabilidades.	Softmax convierte esas puntuaciones en una distribución sobre el vocabulario.
Sampling es una decisión de comportamiento.	Temperatura, top-k y top-p cambian cuánta variación permitimos al generar.
La arquitectura interna afecta al producto visible.	Estabilidad, coste, latencia y estilo de respuesta dependen de estas piezas.

Para saber más

Ba, J. L., Kiros, J. R. y Hinton, G. E. (2016). *Layer normalization*.
<https://arxiv.org/abs/1607.06450>

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

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/1457c0d6bfc4967418bfb8ac142f64a-Abstract.html>

Bycroft, B. (2023). *LLM Visualization*. <https://bbycroft.net/llm>

Cho, A., Kim, G. C., Karpekov, A., Helbling, A., Wang, Z. J., Lee, S., Hoover, B. y Chau, D. H. (2025). Transformer Explainer: interactive learning of text-generative models. En *Proceedings of the AAAI Conference on Artificial Intelligence*.

<https://ojs.aaai.org/index.php/AAAI/article/download/35347/37502>

Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press.

<https://www.deeplearningbook.org>

He, K., Zhang, X., Ren, S. y Sun, J. (2016). Deep residual learning for image recognition. En *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770-778). <https://doi.org/10.1109/CVPR.2016.90>

Hendrycks, D. y Gimpel, K. (2016). *Gaussian Error Linear Units (GELUs)*.

<https://arxiv.org/abs/1606.08415>

Holtzman, A., Buys, J., Du, L., Forbes, M. y Choi, Y. (2020). The curious case of neural text degeneration. En *International Conference on Learning Representations*.

<https://openreview.net/forum?id=rygGQyrFvH>

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. y Sutskever, I. (2019). *Language models are unsupervised multitask learners*. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Shazeer, N. (2020). *GLU variants improve Transformer*. <https://arxiv.org/abs/2002.05202>

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. 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>. El artículo introduce capas de atención multi-cabeza, redes feed-forward por posición, conexiones residuales y normalización. ↵

2. He, K., Zhang, X., Ren, S. y Sun, J. (2016). Deep residual learning for image recognition. En *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770-778). <https://doi.org/10.1109/CVPR.2016.90>. Las conexiones residuales ayudaron a entrenar redes mucho más profundas al permitir que la información y el gradiente circularan con menos degradación. ↵
3. Ba, J. L., Kiros, J. R. y Hinton, G. E. (2016). *Layer normalization*. <https://arxiv.org/abs/1607.06450>. LayerNorm normaliza las activaciones de una capa para cada ejemplo, lo que resulta útil en arquitecturas de secuencia y Transformer. ↵
4. Zhang, B. y Sennrich, R. (2019). Root Mean Square Layer Normalization. *Advances in Neural Information Processing Systems 32*. <https://arxiv.org/abs/1910.07467>. RMSNorm normaliza por la raíz cuadrática media del vector, sin restar la media, y resultó casi igual de eficaz que LayerNorm con menos cómputo. ↵
5. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. <https://www.deeplearningbook.org>. El libro explica cómo las capas lineales y activaciones no lineales permiten componer funciones complejas. ↵
6. Hendrycks, D. y Gimpel, K. (2016). *Gaussian Error Linear Units (GELUs)*. <https://arxiv.org/abs/1606.08415>. GELU propone una activación suave que se adoptó en varias arquitecturas de lenguaje. ↵
7. Shazeer, N. (2020). *GLU variants improve Transformer*. <https://arxiv.org/abs/2002.05202>. El trabajo estudia variantes con puertas para mejorar el bloque feed-forward de Transformers. ↵
8. Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer. Softmax aparece como una forma de convertir puntuaciones reales en probabilidades multinomiales. ↵
9. Holtzman, A., Buys, J., Du, L., Forbes, M. y Choi, Y. (2020). The curious case of neural text degeneration. En *International Conference on Learning Representations*. <https://openreview.net/forum?id=rygGQyrFvH>. El artículo analiza problemas de decodificación y propone nucleus sampling como alternativa a estrategias demasiado rígidas. ↵
10. Bycroft, B. (2023). *LLM Visualization*. <https://bbycroft.net/llm>. Visualización interactiva 3D de un Transformer estilo GPT. ↵
11. Cho, A., Kim, G. C., Karpekov, A., Helbling, A., Wang, Z. J., Lee, S., Hoover, B. y Chau, D. H. (2025). Transformer Explainer: interactive learning of text-generative models. En *Proceedings of the AAAI Conference on Artificial Intelligence*. <https://ojs.aaai.org/index.php/AAAI/article/download/35347/37502>. La herramienta permite visualizar componentes de un GPT-2 pequeño en navegador. ↵

Cuadernos para practicar

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



Cómo elige palabras un LLM: temperatura, top-k y top-p

<https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%C3%ADmil%2003/02-sampling-temperatura-topk-topp.ipynb>

Capítulo 05: Arquitecturas modernas: familias, MoE, razonamiento y multimodalidad

Tres formas de hacer que un modelo sea más capaz

Hasta ahora hemos mirado el Transformer clásico por dentro: tokens, tensores, atención, Q , K , V , residual, LayerNorm, MLP, logits y sampling. Ya tenemos la maquinaria básica. Ahora aparece una pregunta natural: si eso funciona, ¿cómo lo hacemos más capaz sin limitarnos a hacerlo todo más grande?

Las arquitecturas modernas suelen tocar tres palancas:

PALANCA	PREGUNTA QUE RESPONDE	EJEMPLO
Capacidad	¿Cómo meto más conocimiento y patrones sin pagar todo el coste en cada token?	Mixture of Experts, o MoE.
Proceso	¿Cómo doy más tiempo de cálculo a problemas que necesitan varios pasos?	Razonamiento, varias muestras, verificación.
Entrada	¿Cómo hago que el modelo trabaje con texto, imágenes, audio o acciones?	Modelos multimodales.

Este capítulo va de esas tres palancas. No son trucos aislados. Son respuestas distintas a la misma tensión: queremos modelos más útiles, pero tenemos límites de memoria, latencia, coste, datos y evaluación.

Qué no significan estas palabras

MoE no significa que haya expertos humanos dentro del modelo. La palabra experto engaña un poco. En la práctica, un experto suele ser una red interna, normalmente parecida al MLP del capítulo anterior, y un router decide qué expertos se activan para cada token.

Razonamiento no significa que el modelo piense como una persona. En IA moderna, muchas veces llamamos razonamiento a dedicar más pasos de generación, probar varias rutas, verificar resultados o entrenar el modelo para resolver tareas paso a paso. Eso puede mejorar resultados, pero no convierte al modelo en una mente humana.

Multimodal no significa que el modelo vea como tú. Un modelo puede procesar una imagen, pero lo hace transformándola en representaciones numéricas. A veces esas representaciones son muy útiles; otras veces fallan en detalles espaciales, texto pequeño, conteos o relaciones visuales delicadas.

Más moderno no significa automáticamente mejor para tu caso. Un modelo MoE puede ser excelente, pero más complejo de servir. Un modelo multimodal puede ser muy útil, pero más caro y más difícil de evaluar. Un modelo con razonamiento puede resolver mejor ciertas tareas, pero tardar más o necesitar controles de calidad distintos.

El mapa amplio de arquitecturas

Antes de entrar en MoE, razonamiento y multimodalidad, conviene abrir el plano completo. No existe una lista cerrada de «todas las arquitecturas», porque la investigación mezcla familias continuamente. Pero sí hay un mapa útil para no perderse.

Una primera separación:

NIVEL	QUÉ ES	EJEMPLO
Arquitectura neuronal	La forma interna de la red.	Transformer, CNN, RNN, MoE, Mamba, difusión.
Modelo entrenado	Una arquitectura con pesos aprendidos en datos concretos.	BERT, GPT-2, T5, Mixtral, CLIP.
Arquitectura de sistema	Cómo conectas modelos, datos, herramientas y validadores.	RAG, agente con herramientas, pipeline multimodal.

Esta distinción evita una confusión muy común: RAG, por ejemplo, no es una capa neuronal como LayerNorm. Es una arquitectura de sistema que combina recuperación de información con generación.¹

Familias neuronales clásicas y actuales

FAMILIA	IDEA CENTRAL	SE USA MUCHO EN	QUÉ DEBES RECORDAR
MLP	Capas densas que transforman vectores.	Tabular, bloques internos de Transformers.	No mezcla posiciones; transforma representaciones.
CNN	Filtros locales que detectan patrones espaciales.	Imagen, audio, visión industrial.	Muy fuertes cuando importa la vecindad local. ²
RNN / LSTM / GRU	Procesan secuencias manteniendo un estado.	Series temporales, lenguaje antes del dominio Transformer.	Buen punto histórico para entender secuencia, aunque hoy compiten con Transformers y SSM. ³
Transformer	Atención para mezclar información entre posiciones.	LLMs, traducción, visión, multimodalidad.	Es la familia dominante en lenguaje moderno, pero no la única.
GNN	Mensajes entre nodos de un grafo.	Moléculas, redes, conocimiento estructurado.	Sirven cuando la estructura no es una secuencia sino un grafo. ⁴
Autoencoders / VAE	Comprimir y reconstruir datos mediante una representación latente.	Generación, compresión, representación.	Enseñan la idea de espacio latente. ⁵
Diffusion models	Aprenden a quitar ruido paso a paso.	Imagen, audio, vídeo, generación visual.	Generan invirtiendo un proceso de ruido. ⁶
State Space Models	Mantienen un estado y procesan secuencias con coste favorable.	Secuencias largas, lenguaje, audio, señales.	Mamba reabrió el interés por alternativas al Transformer puro. ⁷

Una de esas familias merece una nota aparte, porque es la alternativa al Transformer que más interés ha despertado: los **State Space Models** (SSM), popularizados por Mamba. Su atractivo es de coste. Recuerda que la atención del capítulo 3 compara cada token con cada token, un coste $O(n^2)$. Un SSM, en cambio, procesa la secuencia manteniendo un **estado** que resume el pasado y se actualiza token a token, como una RNN moderna: su coste crece **lineal** con la longitud, $O(n)$, no cuadrático. Eso lo hace muy atractivo para secuencias muy largas (audio, genómica, documentos enteros). El precio es que comprimir todo el pasado en un estado fijo puede perder el detalle fino que la atención sí captura, y por eso muchas arquitecturas recientes son **híbridas**: combinan capas de atención con capas de SSM para quedarse con lo mejor de cada una.

Familias Transformer en lenguaje

Dentro de Transformers hay varias arquitecturas que conviene distinguir:

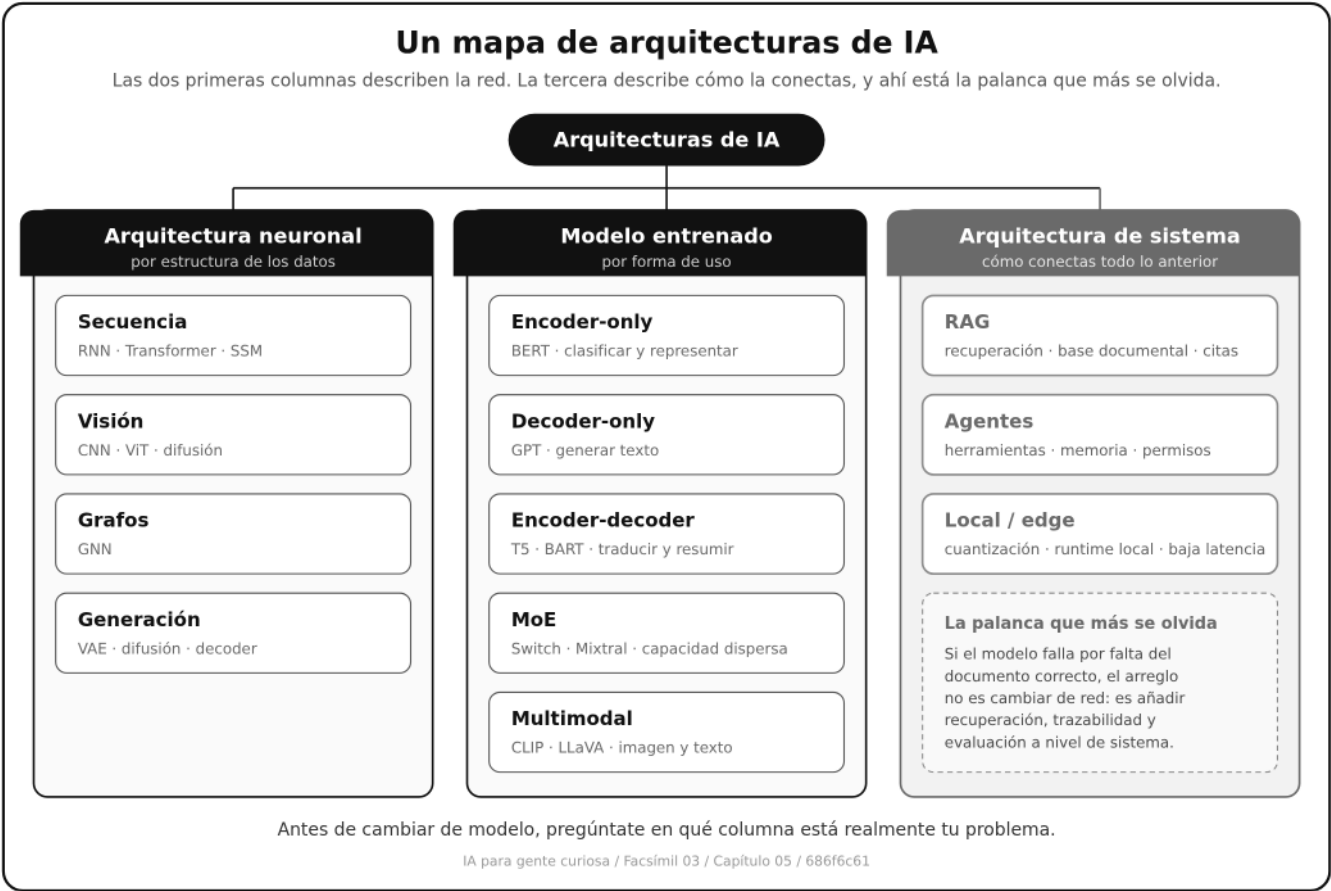
FAMILIA	CÓMO LEE	CÓMO GENERA	EJEMPLO	USO TÍPICO
Encoder-only	Mira el texto completo a ambos lados.	No está pensado principalmente para generar de izquierda a derecha.	BERT. ⁸	Clasificación, búsqueda semántica, extracción, embeddings.
Decode-only	Mira solo el pasado disponible.	Predice el siguiente token.	GPT-2. ⁹	Chat, completado, código, generación libre.
Encoder-decoder	Un encoder lee la entrada; un decoder genera la salida.	Genera condicionándose en lo leído.	T5 y BART. ¹⁰ BART combina un encoder bidireccional y un decoder autoregresivo. ¹¹	Traducción, resumen, transformación de texto.
Vision Transformer	Divide imágenes en parches tratados como tokens.	Depende de la tarea: clasificar, detectar, generar en sistemas híbridos.	ViT. ¹²	Visión y multimodalidad.
MoE Transformer	Usa atención y sustituye partes densas por expertos.	Igual que su base, pero con cómputo disperso.	Switch, Mixtral.	Más capacidad con menos parámetros activos por token.
Long-context / eficiente	Cambia atención, memoria o estado para manejar contextos largos.	Puede ser decoder, encoder o híbrido.	Mamba, variantes de atención eficiente.	Documentos largos, audio, series, agentes con memoria.

La pregunta útil no es «¿cuál es mejor?», sino «¿qué forma de leer y generar necesita mi problema?».

Arquitecturas de sistema que parecen arquitectura neuronal, pero no lo son

SISTEMA	QUÉ COMBINA	CUÁNDO APARECE
RAG	Recuperador, base documental, modelo generativo y citas o validadores.	Preguntas sobre conocimiento cambiante o privado.
Agente con herramientas	Modelo, herramientas, permisos, memoria, evaluadores y orquestación.	Tareas de varios pasos con acciones externas.
Pipeline multimodal	Codificadores de imagen/audio/documento, conectores y LLM.	Facturas, fotos, capturas, vídeo, voz.
Modelo local cuantizado	Pesos comprimidos, runtime local y límites de hardware.	Privacidad, coste, edge AI, baja latencia local.

Esto importa porque a veces se intenta resolver con arquitectura neuronal lo que en realidad es un problema de sistema. Si tu modelo falla porque no tiene el documento correcto, quizá no necesitas cambiar de LLM: necesitas recuperación, trazabilidad y evaluación.



Un ejemplo cotidiano: el ticket de soporte

Imagina que una tienda recibe este mensaje:

Compré unos auriculares ayer. Han llegado rotos y necesito cambiarlos antes del viernes porque son para un regalo.

Un sistema sencillo podría mandar todo el texto al mismo modelo denso y pedir una respuesta. Eso funciona muchas veces. Pero una arquitectura moderna puede repartir mejor el trabajo:

PARTE DEL PROBLEMA	QUÉ NECESITA ENTENDER
«han llegado rotos»	Incidencia de producto o envío.
«cambiarlos»	Política de cambios y devoluciones.
«antes del viernes»	Urgencia temporal.
«son para un regalo»	Tono empático y prioridad percibida.

Un MoE intentaría activar rutas internas útiles para ese tipo de patrón. Un sistema con razonamiento podría separar pasos: identificar intención, comprobar restricciones, decidir respuesta, revisar si falta información. Un modelo multimodal entraría si el usuario adjunta una foto de los auriculares dañados o una captura del pedido.

Esta es la idea del capítulo: no nos interesan las siglas por sí mismas. Nos interesa qué problema resuelve cada arquitectura.

MoE: más capacidad sin activar todo

En el capítulo 04 vimos que cada bloque Transformer tiene un MLP. En muchos modelos MoE, esa parte se sustituye por varios MLPs llamados expertos. Para cada token, un router elige uno o varios expertos y solo esos se ejecutan.

La idea de activar partes del modelo según la entrada aparece en las capas sparsely-gated Mixture-of-Experts de Shazeer y coautores.¹³

Después, GShard llevó esta idea a escalado con sharding automático y cómputo condicional.¹⁴ Switch Transformer simplificó el routing activando un experto por token.¹⁵

La fórmula básica:

$$s = W_r x$$

$$g = \text{softmax}(s)$$

$$C_k = \text{TopK}(g, k)$$

$$\text{MoE}(x) = \sum_{i \in C_k} \frac{g_i}{\sum_{j \in C_k} g_j} E_i(x)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Vector del token o posición que entra al bloque.	Representación de «rotos».
W_r	Matriz del router.	Produce puntuaciones para expertos.
s	Logits del router.	Una puntuación por experto.
g	Pesos tras softmax.	Probabilidad de usar cada experto.
C_k	Expertos seleccionados.	Si $k = 2$, se activan dos expertos.
$E_i(x)$	Salida del experto i .	MLP especializado por entrenamiento.
$\text{MoE}(x)$	Mezcla final de expertos activados.	Sustituye o complementa al MLP denso.

Ejemplo pequeño con cuatro expertos:

EXPERTO	PESO DEL ROUTER G_I
E_1	0,62
E_2	0,24
E_3	0,09
E_4	0,05

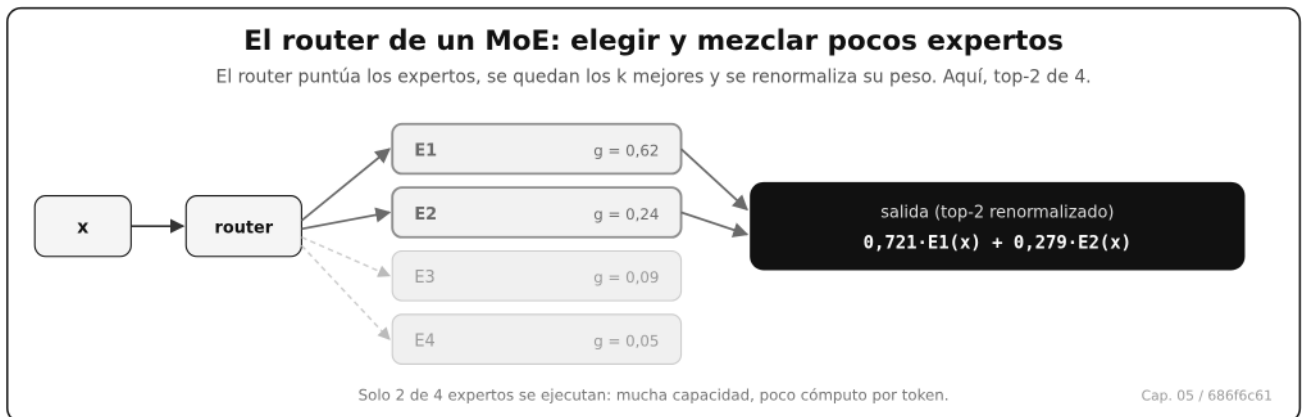
Si usamos top-2, conservamos E_1 y E_2 . Renormalizamos:

$$\hat{g}_1 = \frac{0,62}{0,62 + 0,24} \approx 0,721$$

$$\hat{g}_2 = \frac{0,24}{0,62 + 0,24} \approx 0,279$$

La salida sería:

$$\text{MoE}(x) \approx 0,721E_1(x) + 0,279E_2(x)$$



La clave: el modelo puede tener muchos parámetros, pero no los usa todos en cada token. Eso aumenta capacidad sin multiplicar de la misma forma el coste por token. Mixtral 8x7B es un ejemplo conocido de modelo de lenguaje sparse MoE: tiene varios expertos por capa y activa solo una parte para cada token.¹⁶

Parámetros totales frente a activos: los números reales

Aquí está la contabilidad que hace especial al MoE, y conviene verla con cifras. Mixtral 8x7B no son «ocho modelos de 7B»: comparte la atención y solo desdobra los MLP en ocho expertos, así que suma unos **47B de parámetros totales** pero, al activar dos expertos por token, usa solo unos **13B activos** en cada paso. La generación cuesta como un modelo de 13B, pero el modelo «sabe» como uno mucho mayor. El caso llevado al extremo es DeepSeek-V3: **671B de parámetros totales y apenas unos 37B activos por token**, alrededor del 5,5 %.¹⁷ Por eso, al leer la ficha de un MoE, la pregunta no es «¿cuántos parámetros tiene?» sino **dos**: cuántos totales (cuánta memoria hace falta para cargarlo) y cuántos activos (cuánto cuesta cada token). A 2025-2026 esta separación es ya la norma: los modelos abiertos punteros convergieron en combinar MoE con una atención eficiente, del tipo GQA del capítulo 3, para mantener a raya tanto la memoria de pesos como la KV cache.¹⁸

La parte incómoda de MoE

MoE suena precioso en una frase: más capacidad, menos cómputo activo. Pero la ingeniería no sale gratis.

PROBLEMA	QUÉ OCURRE	POR QUÉ IMPORTA
Balanceo	El router puede mandar demasiados tokens a pocos expertos.	Algunos expertos se saturan y otros apenas aprenden.
Comunicación	En entrenamiento distribuido, los tokens viajan hacia expertos en distintos dispositivos.	Puede aumentar latencia y complejidad.
Servicio	No basta con contar parámetros totales.	Importa cuántos expertos se activan, dónde viven y cómo se cargan.
Interpretación	Un experto activado no siempre tiene una etiqueta humana clara.	No podemos decir sin prueba: «este experto sabe medicina».

Una analogía útil: MoE no es una empresa con departamentos perfectamente etiquetados. Se parece más a un edificio con varias salas de trabajo y un recepcionista que aprende a enviar cada ficha a unas salas concretas. Algunas salas acaban especializándose, pero esa especialización no tiene por qué coincidir con nuestras categorías humanas.

¿Y cómo se evita que el recepcionista mande casi todo a dos salas y deje el resto vacías? Con un empujón durante el entrenamiento: una **pérdida auxiliar de balanceo** (*load-balancing loss*) que penaliza al router cuando reparte de forma muy desigual y lo anima a usar todos los expertos de manera parecida. Sin ese término, el MoE tiende a colapsar (unos pocos expertos acaparan el trabajo y los demás nunca aprenden); con él, la carga se distribuye y la capacidad del modelo se aprovecha. Es un detalle pequeño en la fórmula pero decisivo en la práctica: buena parte del arte de entrenar un MoE está en equilibrar la calidad de las respuestas con ese reparto de la carga.

Razonamiento: más proceso, no solo más parámetros

En el uso cotidiano decimos que un modelo razona cuando resuelve algo que parece necesitar pasos: un problema matemático, una planificación, una comparación de opciones, una depuración de código o una decisión con restricciones.

En términos técnicos, muchas mejoras de razonamiento vienen de cambiar **cómo se usa** el modelo durante inferencia o entrenamiento:

TÉCNICA	QUÉ HACE	EJEMPLO
Chain-of-thought	Usa ejemplos o instrucciones con pasos intermedios.	«Primero calcula el tiempo, luego comprueba la restricción».
Self-consistency	Genera varias soluciones y elige la respuesta más consistente.	Cinco rutas llegan a «14:30»; una ruta llega a «14:00».
Tree of Thoughts	Explora varias ramas de solución y evalúa avances parciales.	Planificar una agenda probando varias secuencias.
Verificación	Comprueba el resultado con reglas, código, tests o herramientas.	Ejecutar una consulta SQL o un test unitario.

El paper de chain-of-thought mostró que dar ejemplos con pasos intermedios podía mejorar tareas de razonamiento en modelos grandes.¹⁹ Self-consistency propuso generar varias rutas y escoger la respuesta que aparece de forma más consistente.²⁰ Tree of Thoughts extendió esa idea hacia búsqueda sobre pasos intermedios.²¹

Podemos escribir self-consistency de forma simple:

$$r^{(m)} \sim p(r \mid x)$$

$$a^{(m)} = \text{extraer_respuesta}(r^{(m)})$$

$$\hat{a} = \text{moda}(a^{(1)}, a^{(2)}, \dots, a^{(M)})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Problema de entrada.	«¿A qué hora debo salir?»
$r^{(m)}$	Ruta de solución número m .	Un intento de resolver paso a paso.
M	Número de rutas generadas.	$M = 5$.
$a^{(m)}$	Respuesta extraída de la ruta m .	«14:30».
$\hat{a}$	Respuesta final por consistencia.	La que más se repite o mejor verifica.

Ejemplo:

RUTA	RESPUESTA
1	14:30
2	14:30
3	14:00
4	14:30
5	14:45

La moda es 14:30. No garantiza verdad absoluta, pero reduce la dependencia de una sola muestra. En tareas profesionales, lo fuerte no es pedir «razona más» sin más. Lo fuerte es combinar pasos, herramientas, verificaciones y criterios de aceptación.

El giro de 2025: pensar más en inferencia

Estas técnicas apuntan a una idea que ha reorganizado el campo: el **escalado en tiempo de inferencia** (*test-time compute*). Durante años, mejorar un modelo significaba entrenarlo más grande. En 2025 se consolidó una vía complementaria: dejar fijos los pesos y **gastar más cómputo al generar la respuesta**, produciendo cadenas de razonamiento mucho más largas, explorando varias y verificándolas.²² Los **modelos de razonamiento** (OpenAI o1 y o3, DeepSeek-R1 o Gemini con modo de pensamiento) nacen de aquí: generan una larga deliberación interna antes de responder. DeepSeek-R1 enseñó algo notable: esa capacidad puede incentivarse con **aprendizaje por refuerzo puro**, sin trayectorias de razonamiento escritas a mano, y alcanzó el nivel de o1 en matemáticas y código.²³ El matiz de ingeniería: no sale gratis. Más cómputo en inferencia es más latencia y más coste por respuesta, y la mejora **satura**: rinde mucho en problemas de varios pasos, como matemáticas o código, y bastante menos en preguntas de sentido común. La decisión, otra vez, es un compromiso medible, no un eslogan.

Modo ingeniero: routing y consistencia en código

Las dos primeras palancas caben en muy poco `numpy`. El router de un MoE es softmax, top-k y renormalizar; la self-consistency es muestrear y quedarse con la moda.

```
import numpy as np
from collections import Counter

def router_moe(x, W_r, k=2):
    s = W_r @ x # logits del router, uno por experto
    g = np.exp(s - s.max()); g = g / g.sum() # softmax
    top = np.argsort(g)[::-1][:k] # los k expertos de mayor peso
    pesos = g[top] / g[top].sum() # renormalizar entre los elegidos
    return top, pesos # qué expertos y con qué peso mezclar

# con g = [0.62, 0.24, 0.09, 0.05] y k=2 -> expertos (0,1), pesos [0.721, 0.279]

def self_consistency(respuestas):
    return Counter(respuestas).most_common(1)[0][0] # la moda

print(self_consistency(["14:30", "14:30", "14:00", "14:30", "14:45"])) # 14:30
```

El router devuelve qué expertos se activan y con qué peso se mezclan sus salidas, exactamente la fórmula de MoE; y `self_consistency` se queda con la respuesta que más se repite entre varias rutas. Es lo que aísla el lab de este capítulo: las tres palancas con datos pequeños, para ver el mecanismo sin un modelo gigante delante.

Un ejemplo entendible: planificar una salida

Supongamos:

La reunión empieza a las 16:00. Tardo 25 minutos en taxi. Quiero llegar 10 minutos antes. Necesito 15 minutos para preparar la mochila. ¿A qué hora empiezo a prepararme?

Un modelo que responde rápido podría saltar a una respuesta plausible. Un proceso más cuidadoso separa restricciones:

PASO	CÁLCULO	RESULTADO
Llegar 10 minutos antes	16:00 - 10 min	15:50
Restar taxi	15:50 - 25 min	15:25
Restar preparación	15:25 - 15 min	15:10

La respuesta es: empezar a prepararse a las **15:10**.

La lección no es que el modelo tenga una calculadora perfecta dentro. La lección es que algunas tareas mejoran cuando obligamos al sistema a separar pasos, comprobar restricciones y, si hace falta, usar una herramienta externa. Esto conecta directamente con [facsímil 5](#), donde hablaremos de agentes y herramientas, y con [facsímil 7](#), donde hablaremos de evaluación.

Multimodalidad: convertir otros mundos en tokens útiles

Un modelo de lenguaje trabaja con tokens. Para trabajar con imágenes, audio, vídeo o acciones, necesita convertir esas señales en representaciones que puedan relacionarse con el lenguaje.

¿Cómo se convierte una imagen en algo parecido a tokens? La receta dominante viene del **Vision Transformer** (ViT): se corta la imagen en una cuadrícula de **parches** (por ejemplo, recuadros de 16×16 píxeles), cada parche se aplana y se proyecta a un vector, y a partir de ahí esos vectores se tratan **como si fueran tokens** de una frase.²⁴ Una foto deja de ser píxeles y pasa a ser una secuencia de «palabras visuales» que un Transformer puede atender igual que atiende texto. Sobre esa idea se construye casi toda la visión moderna en LLMs.

Hay varias familias de aproximaciones:

FAMILIA	IDEA	EJEMPLO
Alineación contrastiva	Aprender que una imagen y su texto cercano deben quedar cerca en un espacio común.	CLIP.
Conectores visuales	Usar un codificador de imagen y proyectar sus salidas al LLM.	LLaVA.
Cross-attention	Dejar que el modelo de lenguaje atienda a representaciones visuales.	Flamingo.
Modelos incorporados en entornos	Combinar lenguaje, visión y señales de acción u observación.	PaLM-E.

CLIP entrenó un codificador de imagen y otro de texto para acercar pares imagen-texto correctos y separar pares incorrectos.²⁵ Una similitud típica es el coseno:

$$\text{sim}(v, t) = \frac{v \cdot t}{\|v\| \|t\|}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
v	Vector de imagen.	Representación de una foto de un perro.
t	Vector de texto.	Representación de «foto de un perro».
$v \cdot t$	Producto punto.	Mide alineación entre vectores.
$\ v\ , \ t\ $	Normas de los vectores.	Sirven para normalizar magnitudes.
$\text{sim}(v, t)$	Similitud coseno.	Cerca de 1 si encajan mucho.

Ejemplo:

TEXTO CANDIDATO	SIMILITUD CON LA IMAGEN
«auriculares rotos»	0,82
«caja de zapatos»	0,31
«perro en un parque»	0,08

El sistema no «ve» como una persona; compara representaciones. Esa comparación puede ser muy útil para buscar, clasificar o describir, pero hay que evaluarla en el dominio real.

Modelos posteriores conectaron visión y lenguaje de formas más generativas. Flamingo combinó modelos visuales y de lenguaje para manejar secuencias intercaladas de imágenes, vídeo y texto.²⁶ LLaVA conectó un codificador visual con un LLM y usó ajuste con instrucciones visuales para diálogo imagen-texto.²⁷ PaLM-E exploró modelos que integran lenguaje, visión y señales de entornos físicos.²⁸

El patrón común: adaptar representaciones

Aunque las arquitecturas cambien, hay una receta recurrente:

```
entrada no textual -> codificador -> vectores -> conector -> modelo de lenguaje -> salida
```

Para una imagen:

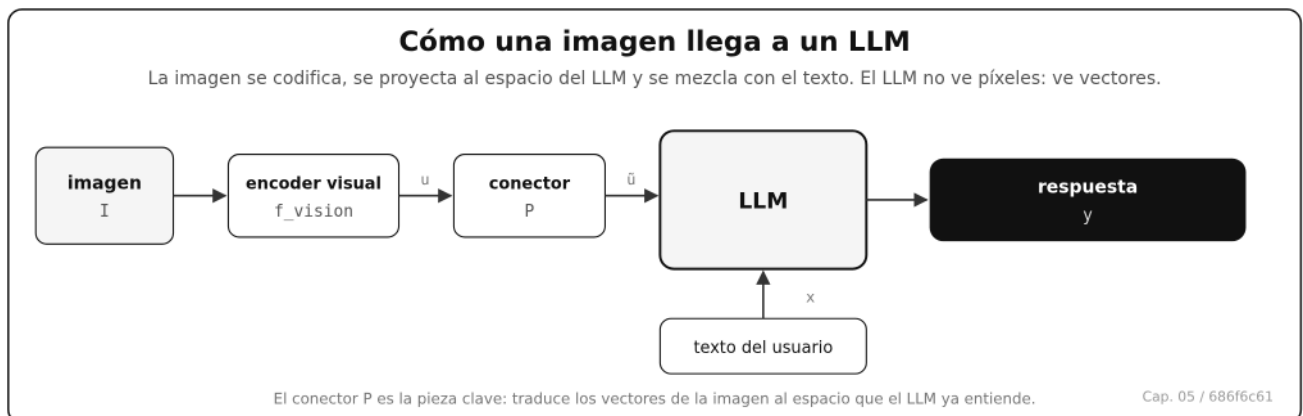
$$u = f_{\text{vision}}(I)$$

$$\tilde{u} = P(u)$$

$$y \sim p(y \mid \tilde{u}, x_{\text{text}})$$

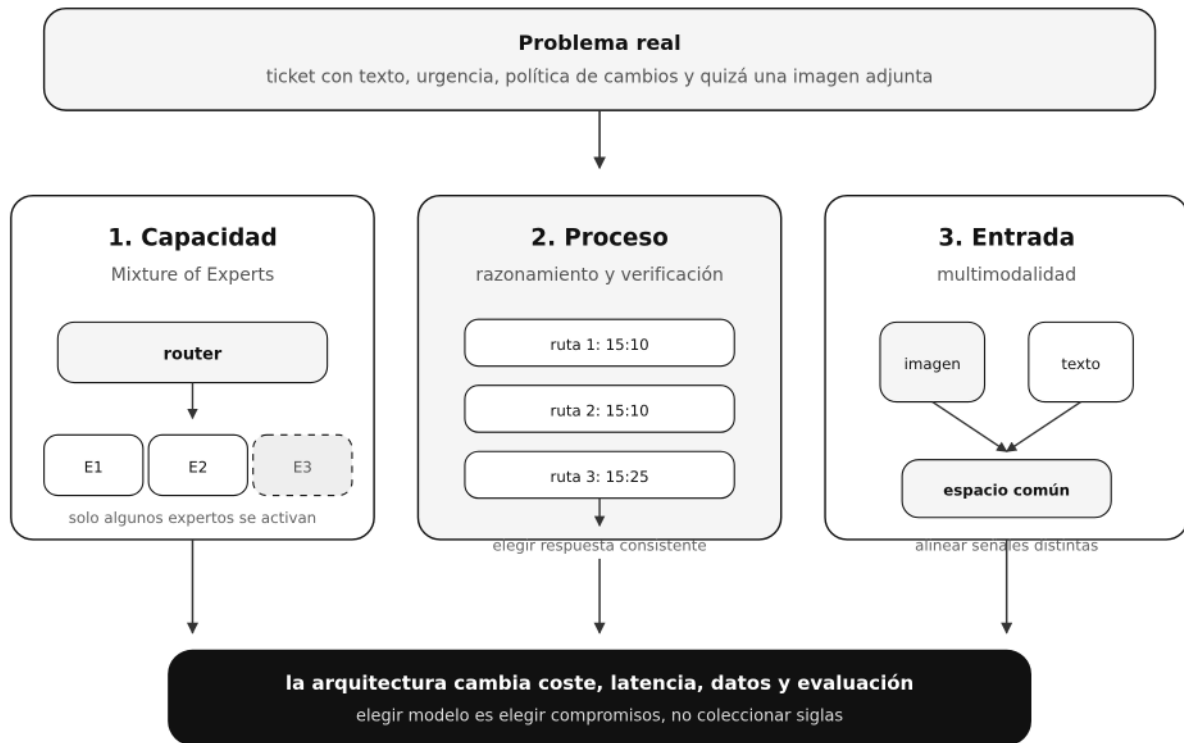
SÍMBOLO	SIGNIFICADO	EJEMPLO
I	Imagen de entrada.	Foto del producto dañado.
f_{vision}	Codificador visual.	Convierte píxeles en vectores.
u	Tokens o vectores visuales.	Representaciones de regiones o parches.
P	Proyector o conector.	Adapta dimensión y formato al LLM.
$\tilde{u}$	Representación visual ya conectada.	Lo que el LLM puede consumir.
x_{text}	Texto del usuario.	«¿Me lo cambian?»
y	Respuesta generada.	Explicación o siguiente acción.

El punto importante: multimodalidad no es solo añadir una imagen al prompt. Hay que decidir cómo se codifica, cómo se alinea, en qué capas se integra, qué datos se usaron para entrenar y cómo se evalúa el resultado.



Tres palancas de las arquitecturas modernas

Más capacidad, más proceso o más tipos de entrada; no son lo mismo y se evalúan distinto.



IA para gente curiosa / Facsímil 03 / Capítulo 05 / 686f6c61

Hacia dónde va esto

Conviene cerrar mirando la dirección del campo, con la cautela de que esto cambia muy rápido (fecha de corte: 10 de junio de 2026). Las tres palancas de este capítulo no se han quedado quietas: se han combinado, y de esa combinación salen las tendencias que hoy dominan el estado del arte.

Convergencia hacia MoE disperso con atención eficiente. Los modelos abiertos punteros han convergido en una receta común: capas MoE para tener muchos parámetros sin pagarlos todos en cada token, más una atención de bajo coste de memoria (GQA o variantes como la *multi-head latent attention*). DeepSeek-V3, con 671B totales y unos 37B activos, ejemplifica hacia dónde va la escala: enorme en conocimiento, contenida en cómputo por token.²⁹

Del entrenamiento a la inferencia: razonamiento por test-time compute. El segundo eje es gastar el cómputo donde más rinde. Tras DeepSeek-R1 y los modelos o1 y o3, el razonamiento por refuerzo y el escalado en inferencia se han vuelto centrales: en lugar de un

único modelo gigante, un modelo que delibera más cuando el problema lo merece. La pregunta abierta es hasta dónde escala esto antes de saturar y cómo controlar su coste.³⁰

Multimodalidad nativa. La multimodalidad ha pasado de «pegar» un encoder a un LLM a entrenar un único modelo que procesa texto, imagen y audio en un espacio común desde el principio (*early fusion*). Familias como Qwen-VL, InternVL o Llama 4 apuntan a que aceptar varias modalidades sea la norma, no una extensión.

Alternativas y arquitecturas híbridas. El Transformer sigue siendo dominante, pero no es un dogma: los State Space Models y los diseños híbridos de atención más SSM buscan domar el coste del contexto largo, y la investigación en atención eficiente no se detiene.

La lectura de ingeniería no cambia con las modas: cada avance mueve un compromiso (memoria, latencia, datos, evaluación), y la pregunta sigue siendo cuál te conviene mover para tu problema, no cuál es el modelo más nuevo del mes.

En el día a día

Elegir un modelo para soporte. Si atiendes miles de tickets diarios, un MoE puede parecer atractivo por capacidad y coste activo. Pero tienes que medir latencia real, disponibilidad, memoria, precio por token y calidad en tus casos. No basta con leer «8x7B» o «experts» en una ficha.

Diseñar un flujo con razonamiento. Para una tarea legal, médica, financiera o de ingeniería, no deberías conformarte con una respuesta directa si hay pasos verificables. Puedes pedir estructura, usar herramientas, ejecutar comprobaciones y registrar evidencias. El razonamiento útil se diseña como proceso, no como decoración textual.

Añadir imágenes a un producto. Si un usuario sube una foto de un daño, una factura o una pizarra, el sistema debe saber qué espera extraer. No es lo mismo describir la imagen que localizar un número de serie, leer una tabla o decidir si falta información.

Evaluar de verdad. Cada palanca trae fallos distintos. MoE puede fallar por routing o servicio. Razonamiento puede fallar por pasos plausibles pero incorrectos. Multimodalidad puede fallar por lectura visual, alineación o detalle espacial. La evaluación debe cubrir esos modos, no solo una puntuación global.

Por qué debería importarte

Porque aquí empiezas a leer fichas técnicas con criterio. Cuando alguien dice «modelo MoE», ya no preguntas solo cuántos parámetros tiene: preguntas cuántos expertos se activan, cómo se sirve y qué latencia tiene. Cuando alguien dice «modelo de razonamiento», preguntas si

mejora en tus tareas, cuánto tarda y cómo se verifica. Cuando alguien dice «multimodal», preguntas qué modalidades entran, cómo se alinean y qué errores has medido.

También importa por coste. Una arquitectura moderna no es una pegatina comercial; cambia memoria, inferencia, datos, evaluación y producto. Elegir mal puede darte un sistema caro, lento o difícil de depurar. Elegir bien puede convertir una tarea imposible en una herramienta útil.

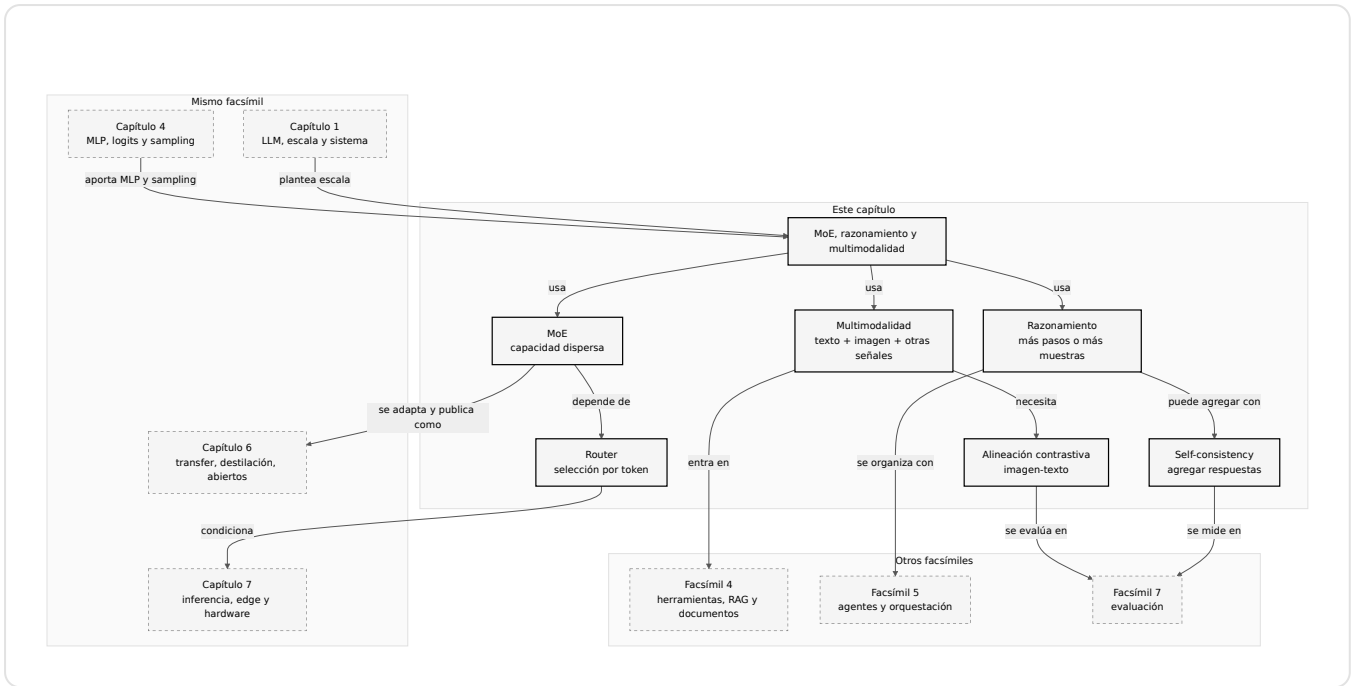
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Pensar que los expertos tienen profesiones humanas	Un experto MoE es una subred aprendida, no un departamento etiquetado.	Habla de routing y activación, no de «el experto de derecho» sin evidencia.
Comparar parámetros totales sin mirar parámetros activos	En MoE, no todos los parámetros se usan en cada token.	Pregunta cuántos expertos se activan y cuál es el coste por token.
Creer que razonar es escribir más texto	Un texto largo puede sonar convincente y estar mal.	Diseña verificación: cálculo, tests, fuentes o consistencia.
Tratar multimodalidad como OCR con esteroides	Una imagen no siempre se reduce a leer texto. Puede implicar objetos, posición, contexto y ambigüedad.	Define qué tarea visual necesitas y evalúala con ejemplos reales.
Meter una arquitectura moderna sin cambiar evaluación	Cada arquitectura trae fallos propios.	Añade pruebas específicas para routing, latencia, pasos y entrada visual.

Cómo encaja todo

Este mapa relaciona las familias modernas con lo aprendido antes y con lo que vendrá después. MoE hereda el MLP y cambia coste activo; razonamiento hereda sampling y añade proceso; multimodalidad hereda embeddings y exige alineación.

La decisión de ingeniería no es “usar lo más moderno”, sino saber qué compromiso estás moviendo: memoria, latencia, datos, evaluación, herramientas o UX.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Mixture of Experts	Arquitectura con varios expertos donde solo algunos se activan por token o ejemplo.
Router	Módulo que asigna tokens a expertos mediante puntuaciones aprendidas.
Cómputo disperso	Uso de una parte del modelo en cada paso, no de todo el modelo.
Parámetros activos	Parámetros que participan realmente en una pasada concreta.
Encoder-only	Transformer orientado a comprender una entrada completa y producir representaciones.
Decoder-only	Transformer causal orientado a generar el siguiente token.
Encoder-decoder	Arquitectura que lee una entrada con encoder y genera salida con decoder.
State Space Model	Modelo de secuencia que mantiene un estado interno y puede escalar bien en secuencias largas.
Diffusion model	Modelo generativo que aprende a quitar ruido progresivamente.
Graph Neural Network	Red que opera sobre nodos y aristas de un grafo.
Arquitectura de sistema	Diseño que combina modelos, datos, herramientas, memoria, recuperación y validadores.
Chain-of-thought	Técnica que usa pasos intermedios para resolver tareas de varios pasos.
Self-consistency	Generar varias soluciones y escoger la respuesta más consistente.
Test-time compute	Escalado del cómputo en inferencia (más muestras o pasos) para mejorar la respuesta sin cambiar los pesos.
Multimodalidad	Integración de texto, imagen, audio, vídeo u otras señales.
Vision Transformer	Transformer que parte una imagen en parches y los trata como tokens.
Alineación contrastiva	Entrenar representaciones para acercar pares relacionados y separar pares no relacionados.
Conector multimodal	Proyección que traduce representaciones visuales u otras señales al espacio que usa el LLM.

Antes de pasar página

- ☐ ¿Puedo explicar MoE sin decir que los expertos son humanos en miniatura? (Si no, vuelve a «MoE».)
- ☐ ¿Puedo distinguir arquitectura neuronal, modelo entrenado y arquitectura de sistema? (Si no, vuelve a «El mapa amplio de arquitecturas».)
- ☐ ¿Sé diferenciar encoder-only, decoder-only y encoder-decoder con un ejemplo de uso? (Si no, vuelve a «Familias Transformer en lenguaje».)
- ☐ ¿Sé calcular una ruta top-2 con pesos de router? (Si no, vuelve a «MoE: más capacidad sin activar todo».)
- ☐ ¿Puedo distinguir parámetros totales de parámetros activos con un ejemplo (Mixtral, DeepSeek-V3)? (Si no, vuelve a «Parámetros totales frente a activos».)
- ☐ ¿Sé por qué un SSM escala $O(n)$ y la atención $O(n^2)$? (Si no, vuelve a «El mapa amplio de arquitecturas».)
- ☐ ¿Entiendo por qué razonamiento no es solo escribir más texto, y qué es el test-time compute? (Si no, vuelve a «Razonamiento» y «El giro de 2025».)
- ☐ ¿Puedo resolver el ejemplo de la salida a las 15:10 paso a paso? (Si no, vuelve a «Un ejemplo entendible».)
- ☐ ¿Entiendo cómo una imagen se convierte en tokens con un Vision Transformer (parches)? (Si no, vuelve a «Multimodalidad».)
- ☐ ¿Sé explicar cómo una imagen llega a un modelo de lenguaje? (Si no, vuelve a «El patrón común».)
- ☐ ¿Puedo nombrar hacia dónde va el campo (MoE disperso, test-time compute, multimodalidad nativa)? (Si no, vuelve a «Hacia dónde va esto».)
- ☐ ¿Sé cómo cambiaría el resultado al variar un ejemplo de routing o de alineación? (Si no, vuelve a «MoE: más capacidad sin activar todo» y «El patrón común».)
- ☐ ¿Puedo relacionar MoE, razonamiento y multimodalidad con coste, latencia y evaluación? (Si no, vuelve a «Por qué debería importarte».)

En resumen

IDEA FUERZA	DETALLE
No hay una sola familia de arquitectura.	MLP, CNN, RNN, Transformer, GNN, difusión, SSM y sistemas RAG resuelven problemas distintos.
Hay que separar arquitectura neuronal, modelo y sistema.	BERT, GPT, T5 o RAG no son el mismo tipo de cosa aunque aparezcan juntos en conversaciones de producto.
MoE aumenta capacidad activando solo parte del modelo.	El router decide qué expertos procesan cada token, pero eso añade retos de balanceo y servicio.
Razonamiento es proceso, no eslogan.	Más pasos, varias muestras y verificación pueden mejorar tareas complejas si se evalúan bien.
Multimodalidad convierte señales distintas en representaciones compatibles.	Imagen y texto deben alinearse antes de que el LLM pueda usarlos de forma útil.
Las arquitecturas modernas cambian compromisos.	Calidad, coste, latencia, memoria y evaluación se mueven a la vez.

Para saber más

Alayrac, J.-B. y otros (2022). Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems* 35, 23716-23736.

<https://arxiv.org/abs/2204.14198>

Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, 4171-4186. <https://doi.org/10.18653/v1/N19-1423>

Dosovitskiy, A. y otros (2021). An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations*. <https://arxiv.org/abs/2010.11929>

Driess, D. y otros (2023). *PaLM-E: An embodied multimodal language model*. <https://arxiv.org/abs/2303.03378>

Fedus, W., Zoph, B. y Shazeer, N. (2022). Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120), 1-39. <https://jmlr.org/papers/v23/21-0998.html>

Gu, A. y Dao, T. (2023). *Mamba: Linear-time sequence modeling with selective state spaces*. <https://arxiv.org/abs/2312.00752>

- Ho, J., Jain, A. y Abbeel, P. (2020). Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems 33*, 6840-6851.
<https://papers.nips.cc/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Abstract.html>
- Jiang, A. Q. y otros (2024). *Mixtral of experts*. <https://arxiv.org/abs/2401.04088>
- Kingma, D. P. y Welling, M. (2014). Auto-encoding variational Bayes. *International Conference on Learning Representations*. <https://arxiv.org/abs/1312.6114>
- Kipf, T. N. y Welling, M. (2017). Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations*.
<https://arxiv.org/abs/1609.02907>
- Lepikhin, D. y otros (2021). GShard: Scaling giant models with conditional computation and automatic sharding. *International Conference on Learning Representations*.
<https://arxiv.org/abs/2006.16668>
- Lewis, M. y otros (2020). BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *Proceedings of ACL*, 7871-7880.
<https://doi.org/10.18653/v1/2020.acl-main.703>
- Lewis, P. y otros (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems 33*, 9459-9474.
<https://papers.nips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html>
- Liu, H., Li, C., Wu, Q. y Lee, Y. J. (2023). Visual instruction tuning. *Advances in Neural Information Processing Systems 36*. <https://arxiv.org/abs/2304.08485>
- Radford, A. y otros (2021). Learning transferable visual models from natural language supervision. *Proceedings of the 38th International Conference on Machine Learning*, 8748-8763. <https://arxiv.org/abs/2103.00020>
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. y Sutskever, I. (2019). *Language models are unsupervised multitask learners*. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
- Raffel, C. y otros (2020). Exploring the limits of transfer learning with a unified text-to-text Transformer. *Journal of Machine Learning Research*, 21(140), 1-67.
<https://www.jmlr.org/papers/v21/20-074.html>
- Shazeer, N. y otros (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *International Conference on Learning Representations*.
<https://arxiv.org/abs/1701.06538>
- Sutskever, I., Vinyals, O. y Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems 27*, 3104-3112.
<https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks>

Wang, X. y otros (2023). Self-consistency improves chain of thought reasoning in language models. *International Conference on Learning Representations*.
<https://arxiv.org/abs/2203.11171>

Wei, J. y otros (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems 35*, 24824-24837.
<https://arxiv.org/abs/2201.11903>

Yao, S. y otros (2023). *Tree of thoughts: Deliberate problem solving with large language models*. <https://arxiv.org/abs/2305.10601>

Notas

1. Lewis, P. y otros (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems 33*, 9459-9474.
<https://papers.nips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html>.
RAG combina memoria paramétrica del modelo con recuperación externa de documentos. ↵
2. LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444.
<https://doi.org/10.1038/nature14539>. El artículo revisa CNNs, redes recurrentes y aprendizaje profundo moderno. ↵
3. Sutskever, I., Vinyals, O. y Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems 27*, 3104-3112.
<https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks>. El trabajo mostró modelos secuencia-a-secuencia basados en redes recurrentes para traducción. ↵
4. Kipf, T. N. y Welling, M. (2017). Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations*.
<https://arxiv.org/abs/1609.02907>. El trabajo presenta una forma eficiente de aplicar redes convolucionales a datos con estructura de grafo. ↵
5. Kingma, D. P. y Welling, M. (2014). Auto-encoding variational Bayes. *International Conference on Learning Representations*. <https://arxiv.org/abs/1312.6114>. El VAE aprende una representación latente probabilística que permite generación y reconstrucción. ↵
6. Ho, J., Jain, A. y Abbeel, P. (2020). Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems 33*, 6840-6851.
<https://papers.nips.cc/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Abstract.html>.
DDPM formaliza generación como un proceso de denoising iterativo. ↵
7. Gu, A. y Dao, T. (2023). *Mamba: Linear-time sequence modeling with selective state spaces*.
<https://arxiv.org/abs/2312.00752>. Mamba propone modelos de espacio de estados selectivos para modelar secuencias en tiempo lineal. ↵

8. Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, 4171-4186. <https://doi.org/10.18653/v1/N19-1423>. BERT usa una arquitectura encoder bidireccional orientada a comprensión. ↵
9. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. y Sutskever, I. (2019). *Language models are unsupervised multitask learners*. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. GPT-2 popularizó el Transformer decoder autoregresivo a escala. ↵
10. Raffel, C. y otros (2020). Exploring the limits of transfer learning with a unified text-to-text Transformer. *Journal of Machine Learning Research*, 21(140), 1-67. <https://www.jmlr.org/papers/v21/20-074.html>. T5 convierte tareas de NLP a un formato texto-a-texto con arquitectura encoder-decoder. ↵
11. Lewis, M. y otros (2020). BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *Proceedings of ACL*, 7871-7880. <https://doi.org/10.18653/v1/2020.acl-main.703>. ↵
12. Dosovitskiy, A. y otros (2021). An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations*. <https://arxiv.org/abs/2010.11929>. ViT aplica Transformers a imágenes dividiéndolas en parches. ↵
13. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G. y Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *International Conference on Learning Representations*. <https://arxiv.org/abs/1701.06538>. El trabajo introduce una capa MoE con una red de gating que selecciona una combinación escasa de expertos por ejemplo. ↵
14. Lepikhin, D. y otros (2021). GShard: Scaling giant models with conditional computation and automatic sharding. *International Conference on Learning Representations*. <https://arxiv.org/abs/2006.16668>. ↵
15. Fedus, W., Zoph, B. y Shazeer, N. (2022). Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120), 1-39. <https://jmlr.org/papers/v23/21-0998.html>. ↵
16. Jiang, A. Q. y otros (2024). *Mixtral of experts*. <https://arxiv.org/abs/2401.04088>. El artículo presenta Mixtral 8x7B como un modelo sparse MoE donde cada capa usa varios expertos feed-forward y selecciona algunos por token. ↵
17. DeepSeek-AI. (2024). *DeepSeek-V3 Technical Report*. <https://arxiv.org/abs/2412.19437>. DeepSeek-V3 es un modelo MoE de 671B de parámetros con unos 37B activos por token. ↵
18. Raschka, S. (2025). *The State of LLMs 2025*. <https://magazine.sebastianraschka.com/p/state-of-llms-2025>. Repaso del año en arquitecturas de LLM, con la convergencia hacia MoE y atención eficiente. ↵

9. Wei, J. y otros (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems 35*, 24824-24837. <https://arxiv.org/abs/2201.11903>. El trabajo estudia cómo ejemplos con razonamiento paso a paso mejoran el rendimiento en tareas que requieren varios pasos. ↵
10. Wang, X. y otros (2023). Self-consistency improves chain of thought reasoning in language models. *International Conference on Learning Representations*. <https://arxiv.org/abs/2203.11171>. El método muestrea múltiples cadenas de solución y agrega respuestas para mejorar robustez. ↵
11. Yao, S. y otros (2023). *Tree of thoughts: Deliberate problem solving with large language models*. <https://arxiv.org/abs/2305.10601>. El marco explora unidades de pensamiento como nodos de búsqueda y permite evaluar caminos parciales. ↵
12. Snell, C., Lee, J., Xu, K. y Kumar, A. (2024). *Scaling LLM Test-Time Compute Optimally Can Be More Effective Than Scaling Model Parameters*. <https://arxiv.org/abs/2408.03314>. El trabajo muestra que repartir más cómputo en inferencia puede rendir más que aumentar los parámetros. ↵
13. Guo, D. y otros (DeepSeek-AI). (2025). *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. <https://arxiv.org/abs/2501.12948>. R1 muestra que el razonamiento paso a paso puede emerger de refuerzo puro y escalar con el cómputo de inferencia. ↵
14. Dosovitskiy, A. y otros (2021). An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations*. <https://arxiv.org/abs/2010.11929>. El título lo dice literalmente: una imagen vale 16x16 palabras, los parches que se tratan como tokens. ↵
15. Radford, A. y otros (2021). Learning transferable visual models from natural language supervision. *Proceedings of the 38th International Conference on Machine Learning*, 8748-8763. <https://arxiv.org/abs/2103.00020>. CLIP aprende representaciones visuales usando supervisión en lenguaje natural y permite transferencia zero-shot. ↵
16. Alayrac, J.-B. y otros (2022). Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems 35*, 23716-23736. <https://arxiv.org/abs/2204.14198>. Flamingo integra información visual y textual con mecanismos que permiten few-shot learning multimodal. ↵
17. Liu, H., Li, C., Wu, Q. y Lee, Y. J. (2023). Visual instruction tuning. *Advances in Neural Information Processing Systems 36*. <https://arxiv.org/abs/2304.08485>. LLaVA conecta un encoder visual y un LLM, y ajusta el sistema con datos de instrucciones multimodales. ↵
18. Driess, D. y otros (2023). *PaLM-E: An embodied multimodal language model*. <https://arxiv.org/abs/2303.03378>. PaLM-E integra observaciones multimodales en un modelo de lenguaje para tareas encarnadas. ↵

9. Raschka, S. (2025). *The State of LLMs 2025*. <https://magazine.sebastianraschka.com/p/state-of-llms-2025>. El repaso del año describe la convergencia de los modelos abiertos hacia MoE más atención eficiente. ↩
10. Snell, C., Lee, J., Xu, K. y Kumar, A. (2024). *Scaling LLM Test-Time Compute Optimally Can Be More Effective Than Scaling Model Parameters*. <https://arxiv.org/abs/2408.03314>. ↩

Capítulo 06: Transfer learning, destilación y modelos abiertos

Casi nadie empieza desde cero

Entrenar un modelo grande desde cero es caro, lento y exige datos, infraestructura, evaluación y experiencia. La mayoría de equipos no empieza ahí. Empieza con un modelo que ya sabe mucho y lo adapta a una tarea concreta.

Ese cambio de mentalidad es enorme. En vez de preguntar «¿cómo entreno mi propio LLM desde cero?», preguntamos:

PREGUNTA PRÁCTICA	TÉCNICA QUE SUELE APARECER
¿Puedo reutilizar un modelo ya entrenado?	Transfer learning.
¿Puedo ajustarlo a mi dominio?	Fine-tuning o instruction tuning.
¿Puedo entrenar pocos parámetros?	LoRA, adapters, QLoRA.
¿Puedo hacerlo más pequeño?	Destilación y cuantización.
¿Puedo ejecutarlo y modificarlo legalmente?	Modelos abiertos, licencias y documentación.

Este capítulo une esas piezas. Es menos espectacular que hablar de billones de parámetros, pero mucho más cercano a la realidad profesional: elegir una base, adaptarla, medirla, comprimirla si hace falta y entender qué te permite su licencia.

Cuando aparezcan herramientas, catálogos, modelos abiertos o librerías de evaluación, toma la sección como estado observado a **10 de junio de 2026**. Las ideas estables son el mecanismo y el criterio de decisión; los nombres, versiones, licencias y capacidades concretas deben revisarse antes de usarlos en un proyecto real.

Qué no es adaptar un modelo

Adaptar un modelo no es «meterle documentos» como quien mete libros en una estantería. Si haces fine-tuning, cambias pesos. Si haces RAG, conectas recuperación externa. Si haces prompt engineering, cambias el contexto. Son mecanismos distintos.

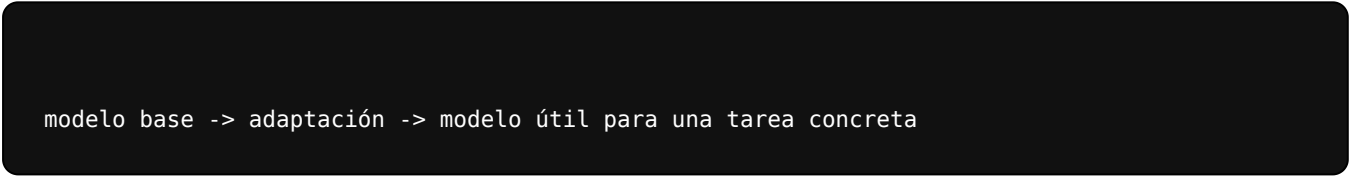
Tampoco conviene pensar que fine-tuning arregla cualquier problema. Si el modelo falla porque no tiene el dato actualizado, puede que necesites recuperación. Si falla porque no sigue un formato, quizá necesitas mejores ejemplos, validación o una interfaz más estricta. Si falla porque la tarea requiere cálculo exacto, puede que necesites una herramienta.

Y un modelo abierto no siempre significa lo mismo. A veces puedes descargar pesos, pero no tienes datos de entrenamiento. A veces la licencia permite investigación pero limita uso comercial. A veces hay código, tokenizer y evaluación; otras veces solo un checkpoint. La palabra «abierto» exige mirar detalles.

Transfer learning: reutilizar una base

Transfer learning significa usar conocimiento aprendido en un contexto para mejorar otro. En NLP moderno, el patrón se volvió muy claro con modelos preentrenados: primero se entrena una base con grandes cantidades de texto; después se adapta a tareas concretas.¹ T5 lo llevó a una formulación texto-a-texto muy amplia.²

Una forma simple de verlo:



FASE	QUÉ APRENDE	EJEMPLO
Preentrenamiento	Patrones generales del lenguaje y del mundo textual.	Continuar texto, relacionar conceptos, estilo, idiomas.
Instruction tuning	Seguir instrucciones humanas en formato pregunta/respuesta.	«Resume esto», «extrae estos campos», «clasifica».
Fine-tuning de dominio	Patrones de un sector o tarea concreta.	Soporte técnico, documentos jurídicos, código interno.
Evaluación	Si la adaptación ayuda o daña.	Tests propios, casos reales, comparación contra base.

Instruction tuning se volvió central para convertir modelos base en asistentes útiles.³ Pero no hay que confundirlo con una capacidad espontánea: es entrenamiento sobre ejemplos de comportamiento deseado.

Datasets: qué se entrena realmente

Cuando hablamos de entrenamiento, la palabra “datos” se queda corta. Un LLM no se entrena con “internet” como una masa uniforme. Se entrena con mezclas de documentos, código, conversaciones, instrucciones, comparaciones, problemas, imágenes, etiquetas, filtros y evaluaciones. Cada familia de datos empuja una capacidad distinta.

Un dataset no es solo una carpeta con archivos. Es una decisión de producto y de ingeniería:

PREGUNTA	POR QUÉ IMPORTA
¿De dónde sale?	Derechos, licencia, idioma, sesgo de dominio y calidad.
¿Qué formato tiene?	Texto libre, pares pregunta-respuesta, preferencias, imagen-texto, código, tablas.
¿Qué enseña?	Continuar texto, seguir instrucciones, razonar, preferir una respuesta, ver imágenes.
¿Qué no enseña?	Ningún dataset cubre todo; cada mezcla tiene huecos.
¿Cómo se filtra?	Duplicados, baja calidad, datos sensibles, idioma, toxicidad, contenido roto.
¿Cómo se mezcla?	La proporción entre web, código, matemáticas o diálogo cambia el modelo.
¿Cómo se evalúa?	Sin conjunto de evaluación separado, puedes entrenar para memorizar tus pruebas.

La arquitectura de entrenamiento cambia según el tipo de dataset.

TIPO DE DATASET	FORMA TÍPICA	QUÉ ENTRENA	EJEMPLOS CONOCIDOS
Preentrenamiento textual	Texto largo no etiquetado.	Predicción del siguiente token o reconstrucción.	C4/T5, The Pile. ⁴
Código	Repositorios, archivos, funciones, comentarios.	Sintaxis, APIs, patrones de programación.	The Stack. ⁵
SFT / instrucciones	instrucción -> respuesta esperada .	Seguir tareas, formato, tono, extracción, resumen.	FLAN, Dolly-15K. ⁶
Preferencias	prompt, respuesta A, respuesta B, preferida .	Elegir entre salidas plausibles.	HH-RLHF, UltraFeedback. ⁷
Razonamiento	Problema, solución, pasos o verificador.	Matemáticas, cadenas de inferencia, comprobación.	GSM8K. ⁸
Multimodal	Imagen-texto, imagen-instrucción-respuesta, vídeo-texto.	Alinear señales visuales y lenguaje.	LAION-5B, LLaVA. ⁹
Evaluación	Preguntas con respuesta esperada o rúbrica.	No entrena; mide.	MMLU, BEIR, MTEB. ¹⁰

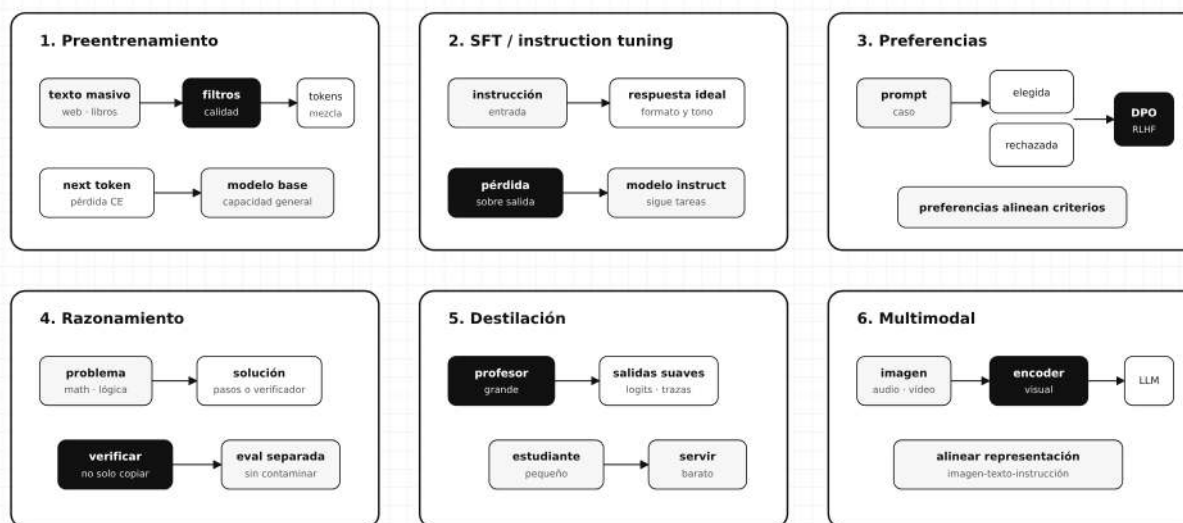
No confundas dataset de entrenamiento con dataset de evaluación. El primero mueve el modelo. El segundo mide si el modelo se comporta mejor. Si una pregunta de evaluación se cuela en entrenamiento, la nota puede subir sin que el modelo haya aprendido a generalizar. A eso se le suele llamar contaminación de datos.

Arquitecturas de datos para entrenar y adaptar

La palabra arquitectura no solo habla de capas del Transformer. También habla de cómo conectas datos, filtros, pérdidas, modelo base, evaluadores y artefactos finales. Aquí tienes el mapa de las arquitecturas de entrenamiento más comunes.

Arquitecturas de datos para entrenar y adaptar LLMs

Cada tipo de dataset cambia entrada, pérdida, artefacto final y riesgo principal.



La evaluación no se mezcla con entrenamiento

si el test entra en el train, el modelo puede memorizar la prueba y parecer mejor de lo que es
guarda versión de datos, filtros, mezcla, tokenizer, receta y métrica

IA para gente curiosa / Facsimil 03 / Capítulo 06 / 686f6c61

El SVG muestra seis arquitecturas porque no todas optimizan lo mismo. Preentrenar busca capacidad general. SFT enseña formato y seguimiento de instrucciones. Preferencias enseñan criterios comparativos. Razonamiento necesita problemas verificables. Destilación pasa señales de un profesor a un estudiante. Multimodalidad alinea señales no textuales con lenguaje.

Qué optimiza cada arquitectura

Para leer bien un dataset hay que mirar tres cosas a la vez: entrada, pérdida y artefacto final. Dos datasets pueden tener el mismo tamaño y servir para cosas muy distintas.

ARQUITECTURA	ENTRADA REAL	OBJETIVO MATEMÁTICO	ARTEFACTO QUE PRODUCE	QUÉ DEBES COMPROBAR
Preentrenamiento causal	Texto tokenizado $x_1, x_2, \dots, x_T$.	Predecir el siguiente token: $\mathcal{L} = -\sum_t \log p(x_t x_{<t})$.	Modelo base.	Mezcla, deduplicación, idiomas, licencias y benchmarks filtrados.
Preentrenamiento texto-a-texto / denoising	Texto con huecos o transformación entrada-salida.	Reconstruir partes ocultas o salida esperada.	Modelo base transferible.	Cómo se generan corrupciones, tareas y plantillas.
SFT / instruction tuning	Instrucción, contexto opcional y respuesta ideal.	Entropía cruzada sobre los tokens de respuesta.	Modelo que sigue formato y tarea.	Calidad de respuestas, diversidad de instrucciones y ejemplos contradictorios.
Preferencias	Prompt, respuesta elegida y respuesta descartada.	Aumentar la probabilidad relativa de la elegida; en DPO aparece una pérdida logística sobre diferencias de log-probabilidad.	Modelo o adaptador alineado con una rúbrica.	Quién decide la preferencia, con qué criterio y si hay sesgo de longitud o estilo.
Razonamiento verificable	Problema, solución, pasos y a veces verificador.	Premiar soluciones que llegan a una respuesta comprobable.	Modelo mejor en problemas estructurados.	Separar entrenamiento y test; comprobar que no aprende plantillas memorizadas.
Destilación	Entradas y salidas del profesor: logits, respuestas o trazas.	Acercar distribución del estudiante a la del profesor, por ejemplo con KL.	Modelo estudiante más barato.	Qué capacidades pierde el estudiante fuera de la tarea.
Multimodal	Pares imagen-texto, imagen-instrucción-respuesta, audio-texto o vídeo-texto.	Alinear representaciones y generar texto condicionado por otra señal.	Modelo capaz de razonar sobre varias modalidades.	Calidad de pares, resolución, licencias y si la descripción textual corresponde a la señal.

ARQUITECTURA	ENTRADA REAL	OBJETIVO MATEMÁTICO	ARTEFACTO QUE PRODUCE	QUÉ DEBES COMPROBAR
Evaluación	Preguntas, respuestas esperadas, qrels, rúbricas o casos no respondibles.	No debería optimizar el modelo; mide una versión.	Métrica, traza y decisión de publicación.	Que no se mezcle con entrenamiento y que represente uso real.

La mezcla de datos también es una arquitectura. Si tienes familias $D_1, D_2, \dots, D_n$, no basta con juntarlas.

Ejemplo de fórmula. Para razonar de forma pedagógica sobre una mezcla, podemos escribir un conjunto de entrenamiento como suma ponderada de familias:

$$D_{\text{train}} = \lambda_1 D_1 + \lambda_2 D_2 + \dots + \lambda_n D_n$$

Aquí D_i representa una familia de datos (web filtrada, código, instrucciones, preferencias, razonamiento, imagen-texto) y λ_i representa su peso relativo en la mezcla. Esta fórmula no describe por sí sola cómo se implementa un *data loader* de entrenamiento real: sirve para recordar que cambiar proporciones cambia capacidades, sesgos, coste y contaminación posible. En un sistema real habría que documentar muestreo, filtros, deduplicación, licencias, idioma, fechas, splits y evaluación.

SÍMBOLO	SIGNIFICADO
D_i	Familia de datos: web, código, matemáticas, conversación, imagen-texto...
λ_i	Peso o proporción de esa familia en la mezcla.
D_{train}	Conjunto efectivo que verá el entrenamiento.

Subir $\lambda_{\text{código}}$ puede mejorar programación y cambiar estilo de respuesta. Subir λ_{math} puede ayudar en problemas formales y no mover demasiado tareas conversacionales. Por eso los informes técnicos serios hablan de mezcla, filtros y etapas; no solo de “tamaño del dataset”.

Tipos de dataset con ejemplos

1. Preentrenamiento textual. Aquí el modelo aprende a predecir texto. No recibe una tarea humana explícita; recibe secuencias y aprende regularidades. C4 y The Pile son ejemplos conocidos de corpora grandes. Lo importante no es solo el tamaño: importan filtros, deduplicación, idiomas, licencias, dominios y contaminación de evaluación.

2. Código. Entrenar con código no es igual que entrenar con prosa. Un dataset como The Stack contiene repositorios y lenguajes de programación. Eso ayuda a aprender sintaxis, patrones de APIs, comentarios, tests y estructura de proyectos. También obliga a mirar licencias y duplicados con mucho cuidado.

3. SFT o instruction tuning. Aquí cada ejemplo suele tener una instrucción y una respuesta deseada. FLAN mezcla tareas y plantillas para enseñar a seguir instrucciones. Dolly-15K ilustra un dataset más pequeño y humano de instrucciones/respuestas. Esta familia enseña comportamiento, no conocimiento vivo.

4. Preferencias. En DPO/RLHF no basta con una respuesta correcta. El dataset dice cuál de dos respuestas se prefiere. HH-RLHF y UltraFeedback ilustran esta familia. La arquitectura cambia: el modelo aprende de comparaciones, no solo de una salida ideal.

5. Razonamiento y verificación. GSM8K contiene problemas matemáticos con soluciones. Sirve para entrenar o evaluar razonamiento paso a paso, pero hay que evitar que los ejemplos de evaluación entren en entrenamiento. En razonamiento, la solución no debe ser una decoración: debe poder comprobarse.

6. Multimodalidad. LAION-5B ilustra pares imagen-texto a gran escala; LLaVA usa instrucción visual para conectar imagen y diálogo. Aquí el dataset no solo enseña palabras. Enseña alineación entre una representación visual y lenguaje.

7. Evaluación. MMLU, BEIR o MTEB no deberían tratarse como entrenamiento de conveniencia. Son instrumentos de medida. Puedes inspirarte en ellos, pero tu producto necesita evaluación propia, como vimos en el [capítulo 10 del facsímil 4](#).

Qué debe mirar un ingeniero en un dataset

La ficha de un dataset debería leerse como una ficha de arquitectura:

CAMPO	QUÉ PREGUNTAS HACER
Procedencia	¿Quién creó los datos? ¿Con qué permisos? ¿De qué dominios vienen?
Formato	¿Texto libre, instrucciones, preferencias, código, imagen-texto, tablas?
Tamaño	¿Número de ejemplos, tokens, documentos o pares?
Calidad	¿Hay filtros, deduplicación, revisión humana o heurísticas claras?
Licencia	¿Permite entrenamiento, redistribución, uso comercial o derivados?
Cobertura	¿Qué idiomas, dominios, estilos y tareas incluye?
Huecos	¿Qué casos no cubre o cubre mal?
Riesgos	¿Datos personales, duplicados, baja calidad, sesgos de fuente?
Contaminación	¿Se mezcló con benchmarks o tests que luego se reportan?
Trazabilidad	¿Puedo reconstruir versión, filtros y mezcla usada?

Una regla simple: un dataset pequeño y muy bueno puede ganar a un dataset enorme y sucio en una adaptación concreta. Para preentrenamiento, el volumen importa mucho; para SFT, preferencias o destilación, la calidad y la cobertura del caso real pesan muchísimo.

Software que ayuda a auditar datasets y evaluar modelos

Sí, existe software para ayudar, pero conviene no meterlo todo en el mismo saco. Una herramienta puede limpiar datos, otra puede detectar ejemplos sospechosos, otra puede ejecutar benchmarks y otra puede evaluar un RAG. “Evaluar el dataset” no es una única operación.

HERRAMIENTA	QUÉ EVALÚA O CURA	CUÁNDO USARLA	QUÉ NO DEBES PEDIRLE
DataTrove	Procesa, filtra, deduplica y calcula estadísticas sobre texto a gran escala. ¹¹	Antes de preentrenar o construir un corpus grande.	No decide por sí sola si una respuesta SFT es pedagógicamente buena.
NVIDIA NeMo Curator	Curation multimodal: texto, imagen, vídeo y audio; filtrado, deduplicación y calidad a escala. ¹²	Cuando el volumen exige pipelines distribuidos o multimodales.	No sustituye tu criterio de dominio ni una licencia clara.
Cleanlab TLM / Studio	Detecta ejemplos problemáticos en pares instrucción-respuesta: baja calidad, prompts vagos, PII, lenguaje tóxico, gramática, etc. ¹³	Para revisar SFT, datasets de preferencias o conjuntos de evaluación con respuestas.	No convierte automáticamente una mala rúbrica en una buena.
Deepchecks	Integridad de datos, distribuciones, splits, comparación de modelos y validación clásica de ML. ¹⁴	En datasets tabulares, visión o ML tradicional; también para detectar problemas de splits.	No es el evaluador principal de un LLM conversacional.
Hugging Face Evaluate	Métricas, mediciones y comparaciones reproducibles para modelos y datasets. ¹⁵	Cuando necesitas métricas conocidas y resultados reproducibles.	No entiende tu negocio si no defines bien la tarea.
LightEval	Evaluación de LLMs en múltiples backends, tareas existentes, tareas propias y resultados muestra a muestra. ¹⁶	Después de adaptar un modelo o comparar variantes.	No audita por sí solo el dataset de entrenamiento.
Im-evaluation-harness	Benchmarks académicos y tareas reproducibles para modelos de lenguaje. ¹⁷	Para comparar contra MMLU, GSM8K, BBH y otros benchmarks públicos.	No sustituye tu evaluación privada de producto.
HELM	Evaluación holística: escenarios, métricas, robustez, transparencia y comparación amplia. ¹⁸	Como marco mental o benchmark amplio.	No te dice si tu asistente responde bien a tus usuarios concretos.

Un flujo profesional mínimo sería:

- Antes de entrenar:** perfilar, filtrar, deduplicar, detectar idioma, longitud, PII, duplicados y baja calidad con DataTrove, NeMo Curator, Cleanlab o checks propios.
- Antes de tocar pesos:** congelar splits `train`, `validation` y `test`, con hashes y versión de dataset.

3. **Después de adaptar:** evaluar con LightEval, lm-evaluation-harness, Hugging Face Evaluate o un harness propio.
4. **Antes de publicar:** pasar tu dataset privado de evaluación, no solo benchmarks públicos.
5. **Si es RAG:** usar evaluación por capas como la del [facsimilar 4, capítulo 10](#): retrieval, contexto, groundedness, citas y abstención.

La idea importante es esta: el software ayuda a encontrar problemas, pero no define por ti qué significa “bueno”. Esa definición sigue siendo del equipo: tarea, rúbrica, usuario, coste, licencia y riesgo de error.

Fine-tuning completo: mover todos los pesos

En fine-tuning completo, partimos de unos pesos θ y los actualizamos con nuevos datos:

$$\theta' = \theta - \eta \nabla_{\theta} \mathcal{L}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
θ	Pesos iniciales del modelo base.	Pesos de un LLM preentrenado.
θ'	Pesos después del ajuste.	Modelo especializado.
η	Tasa de aprendizaje.	2×10^{-5} .
$\nabla_{\theta} \mathcal{L}$	Gradiente de la pérdida respecto a los pesos.	Dirección para reducir error.
$\mathcal{L}$	Función de pérdida.	Error en respuestas esperadas.

Esto puede funcionar muy bien, pero tiene costes:

VENTAJA	COSTE
Gran capacidad de adaptación.	Mucha memoria y cómputo.
Puede modificar profundamente el comportamiento.	Riesgo de olvidar capacidades útiles si los datos son estrechos.
Produce un único modelo ajustado.	Guardar y servir muchas variantes puede ser caro.

En equipos pequeños, fine-tuning completo no suele ser la primera opción. Antes se prueba prompt, RAG, evaluación, datos mejores y métodos eficientes como LoRA.

El olvido catastrófico: adaptar puede borrar lo aprendido

Hay un riesgo concreto detrás de «el modelo puede olvidar capacidades útiles». Tiene nombre propio: olvido catastrófico. Cuando mueves muchos pesos con un conjunto de datos estrecho, el modelo mejora en esa tarea y a la vez degrada habilidades que ya tenía, porque reescribe parámetros que servían para otras cosas. Es un fenómeno conocido desde las primeras redes conexionistas.¹⁹

Imagina un asistente que respondía bien sobre muchos temas. Lo afinas con 2 000 ejemplos de un único formato jurídico. Después escribe contratos impecables, pero contesta peor preguntas generales que antes acertaba. No es magia ni mala suerte: los gradientes empujaron sus pesos hacia un rincón del espacio.

DEFENSA	IDEA	COSTE
Congelar la base	PEFT (LoRA, adapters, prompt tuning) deja W intacto y aprende una pieza aparte.	Menos capacidad de cambio profundo.
Mezclar datos	Incluir ejemplos generales junto a los del dominio (<i>data replay</i>).	Necesitas mantener un conjunto general.
Regularizar hacia el origen	Penalizar alejarse de los pesos base, como en <i>Elastic Weight Consolidation</i> .	Más configuración y ajuste.
Evaluar dentro y fuera	Medir la tarea nueva y también las capacidades viejas.	Exige un conjunto de regresión.

La regularización elástica es una de las defensas clásicas: protege los parámetros importantes para tareas previas mientras aprende la nueva.²⁰ Por eso PEFT no solo ahorra memoria: al congelar la base, reduce de raíz el riesgo de olvidar. Es una de las razones por las que LoRA se volvió la opción por defecto en muchos equipos.

La pérdida y los gradientes: qué cambia de verdad

Cuando decimos «ajustar pesos», estamos diciendo algo muy concreto: calcular una pérdida, derivarla y mover parámetros para reducirla.

Supongamos que queremos enseñar al modelo a continuar esta frase:

El cliente quiere cambiar unos auriculares porque llegaron...

Y el token correcto en nuestro ejemplo es:

rotos

El modelo produce logits para tres candidatos:

TOKEN CANDIDATO	LOGIT
factura	2,0
rotos	1,0
saludo	0,0

Aplicamos softmax:

$$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

TOKEN	PROBABILIDAD P_I	ETIQUETA Y_I	GRADIENTE $P_I - Y_I$
factura	0,665	0	0,665
rotos	0,245	1	-0,755
saludo	0,090	0	0,090

La pérdida de entropía cruzada para el token correcto es:

$$\mathcal{L} = -\log(p_{\text{rotos}})$$

$$\mathcal{L} \approx -\log(0,245) \approx 1,408$$

La derivada respecto a cada logit, cuando usamos softmax y entropía cruzada, queda así:

$$\frac{\partial \mathcal{L}}{\partial z_i} = p_i - y_i$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
z_i	Logit del token candidato i .	$z_{\text{factura}} = 2,0$.
p_i	Probabilidad tras softmax.	$p_{\text{rotos}} = 0,245$.
y_i	Etiqueta real en formato one-hot.	Para «rotos», $y = 1$.
$p_i - y_i$	Gradiente respecto al logit.	Para «rotos», $-0,755$.

Lectura práctica:

TOKEN	QUÉ LE DICE EL GRADIENTE AL ENTRENAMIENTO
factura	«Te estás pasando con este token; baja su puntuación».
rotos	«Este era el bueno; sube su puntuación».
saludo	«También está algo alto; bájalo un poco».

Si la última capa calcula:

$$z = hW$$

entonces el gradiente llega a W así:

$$\frac{\partial \mathcal{L}}{\partial W} = h^T (p - y)$$

En la práctica no entrenamos con una frase, sino con lotes de ejemplos. Si tenemos B ejemplos, una pérdida media sería:

$$\mathcal{L}_{\text{lote}} = -\frac{1}{B} \sum_{b=1}^B \log p(y^{(b)} | x^{(b)})$$

El optimizador no ve «atención al cliente», «factura» o «tono educado». Ve números. Si el lote contiene muchos ejemplos donde una devolución debe terminar en un formato concreto, los gradientes empujan al modelo hacia ese patrón. Si los ejemplos son pobres, repetidos o demasiado estrechos, el modelo puede mejorar esa plantilla y empeorar fuera de ella.

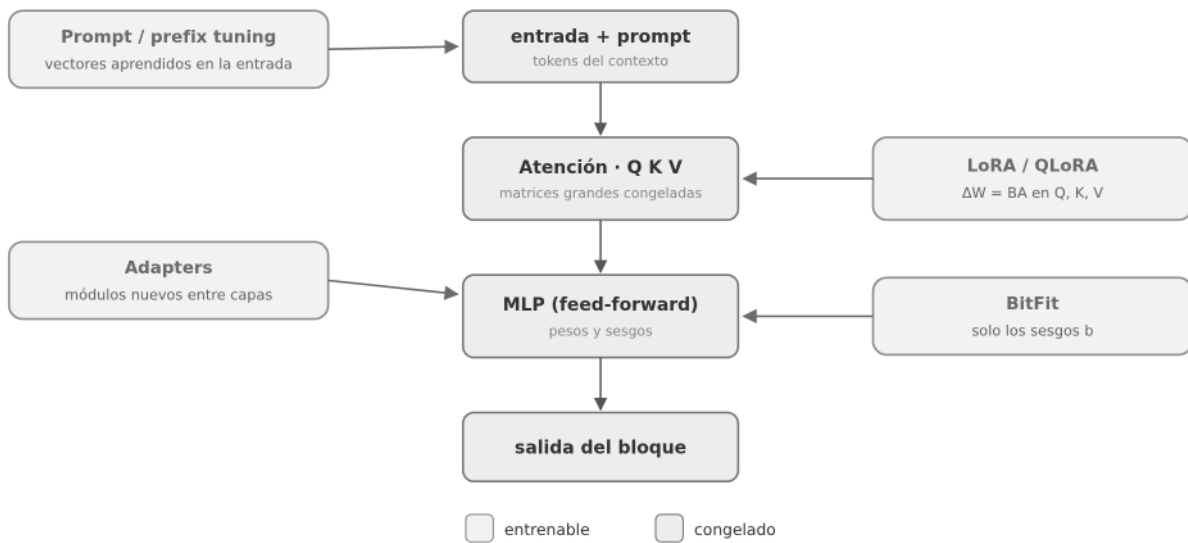
TÉCNICA	POR DÓNDE DEJAMOS PASAR EL GRADIENTE	QUÉ QUEDA CONGELADO	LECTURA PRÁCTICA
Fine-tuning completo	Casi todo el modelo.	Poco o nada.	Máxima capacidad de cambio, máximo coste y más riesgo de tocar cosas que funcionaban.
LoRA / QLoRA	Matrices pequeñas A y B .	Matriz grande W y la mayor parte del modelo.	Adaptas el comportamiento con una pieza pequeña que se puede guardar aparte.
Adapters	Módulos añadidos entre capas.	La red base.	Útil si quieres muchas tareas sobre una base común.
Prompt tuning / prefix tuning	Vectores continuos aprendidos.	Los pesos del modelo.	Cambia la entrada interna, no el modelo base.
IA3	Vectores que escalan activaciones.	Casi todos los pesos.	Ajuste muy ligero: más fino, pero también menos expresivo.
DPO	Pesos o adaptadores a partir de preferencias.	Depende de la implementación.	Enseña qué respuesta preferimos cuando hay varias plausibles.
Destilación	El estudiante.	El profesor.	El grande enseña señales; el pequeño aprende a imitar lo suficiente.

Esa es la diferencia real entre técnicas: **por dónde dejamos pasar el gradiente**. Y por eso la pérdida no basta como única brújula. Puede bajar la pérdida en tus ejemplos y aun así empeorar una tarea real si el conjunto de evaluación no representa el uso diario.

Por dónde pasa el gradiente: el mapa de PEFT

Cada técnica entrena una parte distinta del bloque. Lo verde recibe gradiente; lo gris queda congelado.

Fine-tuning completo mueve todo el bloque; PEFT congela la base y entrena solo una pieza



IA para gente curiosa / Facsímil 03 / Capítulo 06 / 686f6c61

No todo es LoRA: mapa de técnicas de adaptación

LoRA y QLoRA son importantes, pero no son las únicas opciones. Esta tabla sirve como brújula rápida. No hace falta memorizar todos los nombres; lo importante es saber qué parte se entrena, qué problema resuelve y qué puede salir mal.

TÉCNICA	QUÉ SE ENTRENA	CUÁNDO ENCAJA	CUIDADO
Prompting	Nada. Solo contexto escrito.	Primer intento, prototipos, tareas simples.	Si el contexto cambia mucho, puede volverse frágil.
RAG	Normalmente nada del modelo; se entrena o configura recuperación.	Conocimiento privado o cambiante.	Si recupera mal, el modelo contesta con mala base.
Fine-tuning completo	Todos o casi todos los pesos.	Cambio profundo de dominio o comportamiento.	Caro; riesgo de olvidar capacidades útiles.
SFT / instruction tuning	Pesos del modelo, con pares instrucción-respuesta.	Enseñar formato, estilo y seguimiento de tareas.	La calidad de ejemplos manda más que el nombre de la técnica.
Adapters	Módulos pequeños insertados en capas.	Muchas tareas con una base compartida.	Añaden algo de latencia y complejidad. ²¹
Prompt tuning	Vectores de prompt continuos.	Modelos grandes, tareas concretas y bajo coste.	No es texto legible; es un prompt aprendido. ²²
Prefix tuning	Vectores de prefijo para condicionar capas.	Generación controlada con pocos parámetros.	Requiere saber dónde insertar el prefijo. ²³
P-tuning v2	Prompts continuos profundos, a menudo en varias capas.	Cuando quieres prompt tuning más expresivo en tareas de comprensión.	Sigue siendo menos interpretable que ejemplos escritos. ²⁴
BitFit	Principalmente sesgos del modelo.	Prueba barata cuando tienes pocos datos.	Puede quedarse corto si la tarea exige cambiar representaciones profundas. ²⁵
LoRA	Matrices pequeñas A y B .	Ajuste eficiente y práctico en LLMs.	Depende de rango, capas elegidas y datos.
QLoRA	LoRA sobre modelo base cuantizado.	Ajustar modelos grandes con poca memoria.	Cuantización y configuración importan mucho.
IA3	Vectores que escalan activaciones.	Ajuste muy ligero con pocos ejemplos.	Menos flexible que métodos con más parámetros. ²⁶
AdaLoRA	Bajo rango con presupuesto adaptativo.	Cuando no quieres repartir el mismo rango a todas las matrices.	Más configuración; no siempre merece la complejidad. ²⁷
DoRA	Magnitud y dirección del peso, usando bajo rango para la dirección.	Cuando LoRA se queda algo corto y quieres más capacidad.	Es una variante más especializada; conviene validar contra LoRA simple. ²⁸

TÉCNICA	QUÉ SE ENTRENA	CUÁNDO ENCAJA	CUIDADO
DPO / preferencias	Pesos o adaptadores usando pares preferido/rechazado.	Alinear estilo y preferencias tras SFT.	No sustituye datos correctos ni evaluación. ²⁹
Destilación	Un estudiante que imita al profesor.	Reducir tamaño o coste.	Puede perder capacidades fuera de la tarea.
Cuantización	A veces nada; a veces calibración.	Reducir memoria e inferencia.	Puede bajar calidad si se aprieta demasiado.

Una regla práctica: si no tienes evaluación, no elijas técnica todavía. Primero crea diez, cincuenta o cien casos reales que puedas revisar. Sin eso, fine-tuning, LoRA o DPO son formas sofisticadas de adivinar.

Más allá de DPO: alinear con preferencias hoy

DPO fue un punto de inflexión porque optimiza preferencias directamente, sin entrenar un modelo de recompensa aparte como en RLHF clásico. Pero no se quedó solo. A **10 de junio de 2026** conviene conocer la familia, no para usarlas todas, sino para no creer que «alinear» es una sola técnica.

MÉTODO	QUÉ NECESITA	QUÉ APORTA
RLHF	Modelo de recompensa entrenado más un bucle de refuerzo.	El esquema original; potente pero con más piezas que ajustar.
DPO	Pares preferido/rechazado.	Optimiza la preferencia directamente, sin modelo de recompensa. ³⁰
KTO	Solo una etiqueta buena/mala por respuesta, sin pares.	Útil cuando no tienes comparaciones, solo señales sueltas. ³¹
ORPO	Pares de preferencia, pero sin modelo de referencia.	Une SFT y preferencias en un solo paso, más simple de operar. ³²
SimPO	Pares de preferencia, sin modelo de referencia.	Usa la probabilidad media de la secuencia como recompensa; más ligero en memoria. ³³

La lectura útil no es «cuál es la mejor», sino qué datos tienes. Si tienes pares claros, DPO u ORPO encajan. Si solo tienes señales sueltas de bueno o malo, KTO. Y, como siempre, la preferencia no arregla una base de datos pobre: ordena estilo y criterio sobre algo que ya funciona.

Casos cercanos: elegir sin ponerse solemne

La tienda online. El catálogo cambia cada día, pero el tono de atención al cliente debe ser siempre claro y amable. Para el catálogo, RAG: recuperar precio, stock, plazo y política vigente. Para el tono y el formato, quizá basten buenos ejemplos; si se repite mucho, LoRA o adapters pueden fijar esa forma de responder.

El departamento de compras. Recibe presupuestos en PDF y quiere extraer proveedor, importes, plazos y condiciones. Si el problema es leer documentos nuevos, empieza por extracción, validación y RAG. Si el problema es que el modelo devuelve JSON desordenado, SFT o LoRA con ejemplos muy buenos puede ayudar. Si las respuestas son largas y caras, destilar un modelo pequeño para esa tarea empieza a tener sentido.

La app que debe funcionar en un portátil normal. Aquí la pregunta no es solo «qué modelo sabe más», sino «qué modelo cabe, responde rápido y no quema presupuesto». Un modelo open weights cuantizado puede ser suficiente. Si además quieres una tarea muy concreta, QLoRA para adaptar y cuantización para servir son dos piezas que se hablan entre sí.

El equipo docente. Quiere un asistente para generar ejercicios de práctica con el estilo del curso. No necesita memorizar todos los materiales: necesita seguir una plantilla, respetar nivel y producir soluciones revisables. Puedes empezar con prompting y evaluación; si funciona pero se repite cada semana, una adaptación ligera puede ahorrar trabajo.

LoRA: adaptar sin tocar todo

LoRA, *Low-Rank Adaptation*, parte de una observación práctica: quizá no necesitamos actualizar toda una matriz grande para adaptar un modelo. Podemos congelar el peso original W y aprender una actualización de bajo rango.³⁴

En una capa lineal:

$$y = Wx$$

LoRA usa:

$$y = Wx + \frac{\alpha}{r}BAx$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
W	Matriz original congelada.	No se entrena durante LoRA.
A	Matriz pequeña entrenable.	Reduce dimensión hacia rango r .
B	Matriz pequeña entrenable.	Vuelve a la dimensión original.
r	Rango de la adaptación.	$r = 8, r = 16, r = 64$.
α	Escala de la adaptación.	Controla fuerza de LoRA.
BA	Actualización de bajo rango.	Aproxima el cambio necesario en W .

Si W mide 4096×4096 , tiene:

$$4096 \cdot 4096 = 16\,777\,216$$

parámetros.

Con LoRA de rango $r = 16$:

$$16(4096 + 4096) = 131\,072$$

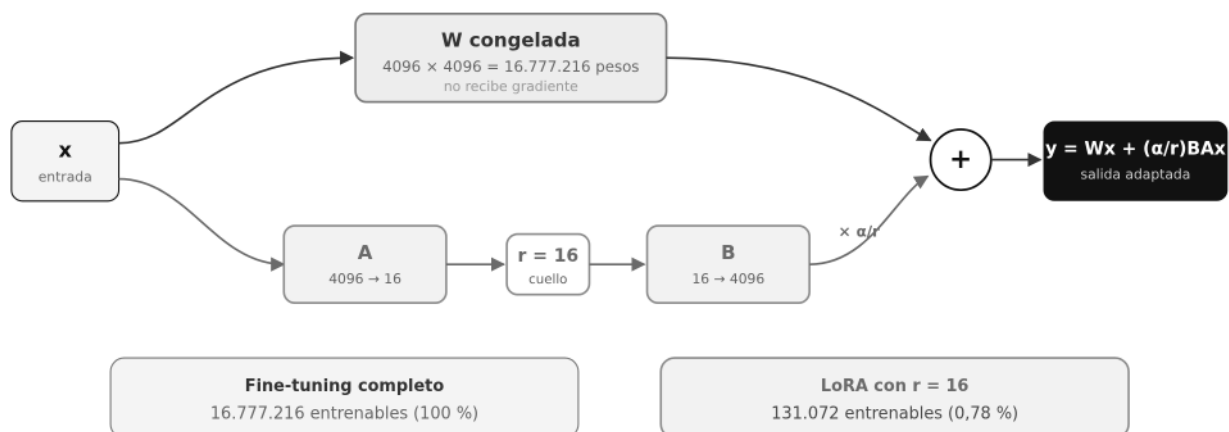
parámetros entrenables.

MÉTODO	PARÁMETROS ENTRENABLES EN LA MATRIZ DEL EJEMPLO
Fine-tuning completo	16 777 216
LoRA con $r = 16$	131 072

No es gratis ni siempre suficiente, pero cambia la escala del problema. Puedes adaptar un modelo grande entrenando una fracción pequeña de sus parámetros.

LoRA: congelar la matriz grande y aprender una pequeña

El gradiente solo pasa por A y B. La matriz W no se toca, así que la base no se olvida.



IA para gente curiosa / Facsímil 03 / Capítulo 06 / 686f6c61

Una ventaja práctica aparece al servir el modelo. Como BA tiene la misma forma que W , puedes **fusionar** el adaptador sumándolo a los pesos base, $W' = W + \frac{\alpha}{r}BA$, y obtener un único modelo sin coste extra en inferencia. O puedes hacer lo contrario: mantener W compartida y **conmutar adaptadores** según la petición, sirviendo muchas variantes de tarea con una sola copia de la base en memoria. Esa flexibilidad, guardar un adaptador de pocos megabytes en lugar de un modelo entero, es parte de por qué LoRA se volvió tan común en producción.

QLoRA: adaptar con menos memoria

QLoRA combina cuantización y LoRA: mantiene el modelo base cuantizado y entrena adaptadores LoRA encima.³⁵

La idea:

pesos base cuantizados y congelados + adaptadores LoRA entrenables

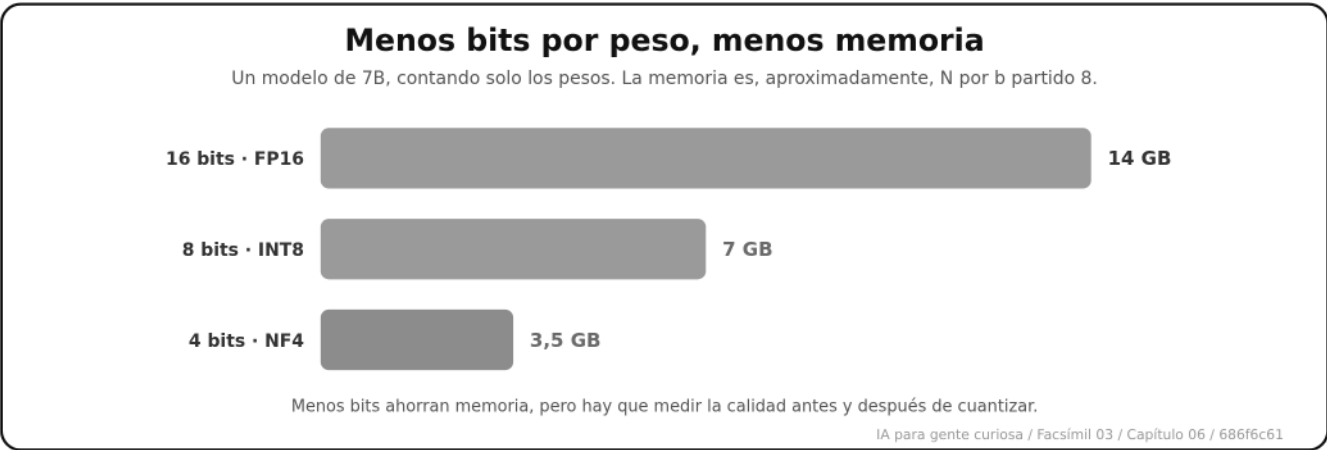
Si un modelo tiene N parámetros y cada parámetro usa b bits, la memoria aproximada solo para pesos es:

$$\text{memoria} \approx \frac{N \cdot b}{8}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
N	Número de parámetros.	7×10^9 .
b	Bits por parámetro.	16, 8, 4.
8	Bits por byte.	Conversión a bytes.

Para un modelo de 7B:

PRECISIÓN	CÁLCULO APROXIMADO	MEMORIA APROXIMADA
16 bits	$7B \cdot 16/8$	14 GB
8 bits	$7B \cdot 8/8$	7 GB
4 bits	$7B \cdot 4/8$	3,5 GB



La realidad añade metadatos, activaciones, KV cache y sobrecostos del runtime, pero la intuición sirve: menos bits reducen memoria. En el capítulo 07 conectaremos esto con inferencia, hardware y ejecución local.

«Cuantizar a 4 bits» no es una sola receta. Hay métodos con nombre, y conviene reconocerlos porque tienen compromisos distintos. Tómalos como estado a **10 de junio de 2026**: los nombres y versiones cambian, pero la idea de cada familia es estable.

MÉTODO	IDEA CENTRAL	DÓNDE ENCAJA
GPTQ	Cuantización post-entrenamiento que ajusta peso a peso para minimizar el error de la capa. ³⁶	Servir modelos a 3-4 bits con poca caída de calidad.
AWQ	Protege los pesos más importantes mirando qué activaciones pesan más. ³⁷	Inferencia rápida a 4 bits en GPU.
NF4 (bitsandbytes)	Tipo de dato de 4 bits «normal-float» diseñado para pesos; es el que usa QLoRA. ³⁸	Entrenar adaptadores con poca memoria.
GGUF / llama.cpp	Formato de pesos cuantizados pensado para ejecutar en CPU y portátiles.	Edge, local y prototipos sin GPU.
FP8	Coma flotante de 8 bits con soporte en hardware reciente.	Entrenamiento e inferencia a gran escala.

Hay una diferencia importante que el capítulo 07 retomará: cuantizar **pesos** no es lo mismo que cuantizar **activaciones**. Algunos métodos tocan solo los pesos; otros, como SmoothQuant, reparten el rango entre pesos y activaciones para que el cálculo en enteros no se des controle.³⁹ La regla práctica no cambia: cuantizar siempre se mide antes y después, porque apretar de más puede costar justo la capacidad que necesitabas.

Destilación: enseñar a un modelo más pequeño

Destilación significa entrenar un modelo estudiante para imitar señales de un modelo profesor. Hinton, Vinyals y Dean popularizaron la idea de transferir conocimiento usando las probabilidades suaves del profesor, no solo etiquetas duras.⁴⁰

La intuición:

ETIQUETA DURA	DISTRIBUCIÓN SUAVE DEL PROFESOR
«La respuesta es París».	París 0,82; Lyon 0,10; Madrid 0,05; azul 0,03.

La segunda señal enseña más. No solo dice cuál es la respuesta más probable; también dice qué alternativas eran más plausibles.

Con temperatura T :

$$p_t^{(T)} = \text{softmax}\left(\frac{z_t}{T}\right)$$

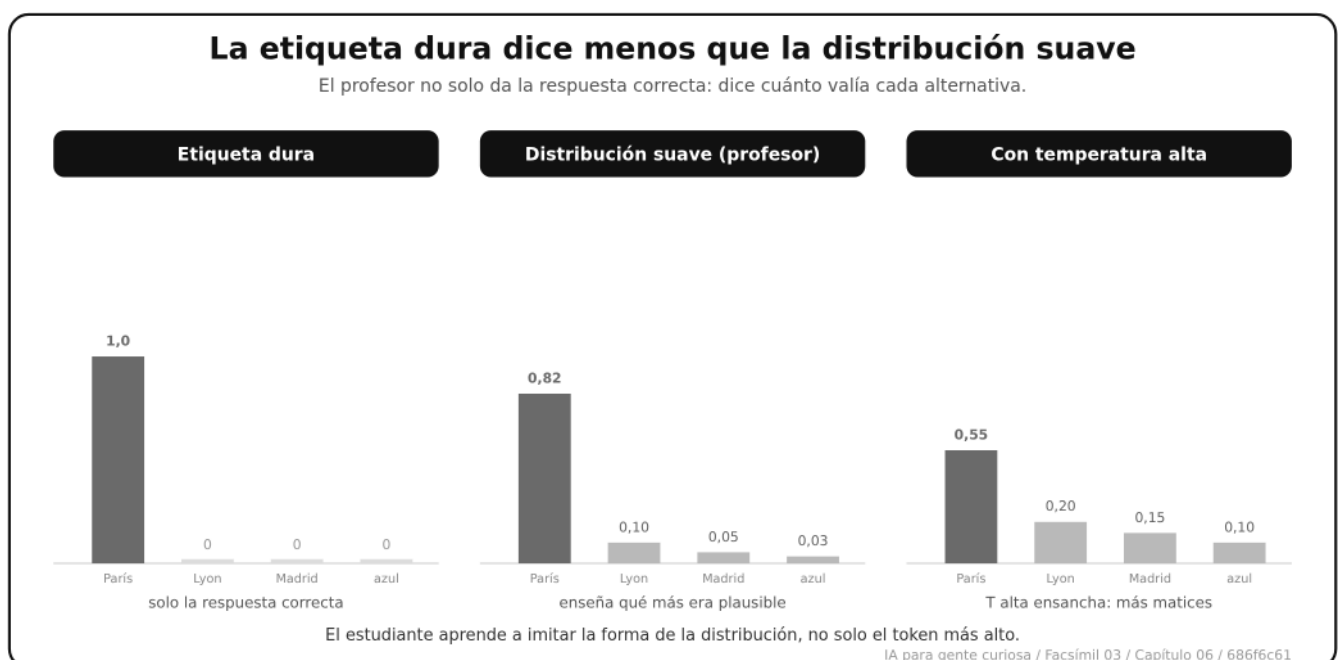
$$p_s^{(T)} = \text{softmax}\left(\frac{z_s}{T}\right)$$

Una pérdida habitual combina etiqueta real y distilación:

$$\mathcal{L} = (1 - \lambda) \text{CE}(y, p_s) + \lambda T^2 \text{KL}(p_t^{(T)} \parallel p_s^{(T)})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$p_t^{(T)}$	Distribución del profesor con temperatura.	Probabilidades suaves.
$p_s^{(T)}$	Distribución del estudiante con temperatura.	Lo que intenta imitar.
y	Etiqueta real o respuesta esperada.	Token correcto o clase correcta.
CE	Entropía cruzada.	Pérdida con etiqueta real.
KL	Divergencia KL.	Distancia entre distribuciones.
λ	Peso de la parte de distilación.	0,5, 0,7...
T	Temperatura.	Suaviza probabilidades.

DistilBERT es un ejemplo clásico: reduce tamaño y mejora velocidad manteniendo buena parte del rendimiento de BERT en tareas de comprensión.⁴¹



Un ejemplo entendible: la academia que prepara apuntes

Imagina una academia con una profesora experta y una ayudante nueva. La profesora corrige exámenes desde hace años. La ayudante todavía está aprendiendo.

Podemos enseñar a la ayudante de tres maneras:

MÉTODO	QUÉ RECIBE LA AYUDANTE	EQUIVALENTE EN MODELOS
Solo soluciones finales	«Esta respuesta está bien, está mal».	Etiquetas duras.
Explicaciones y matices	«Esta está bien; esta otra casi, pero confunde una fecha».	Distribuciones suaves o razonamiento.
Casos seleccionados	Ejercicios representativos del curso.	Datos curados para adaptación.

La destilación se parece al segundo caso. No obliga al estudiante a ser idéntico al profesor, pero intenta que aprenda su criterio de salida. En producto, esto puede servir para tener un modelo más barato o rápido que mantiene suficiente calidad en una tarea concreta.

Modo ingeniero: el presupuesto de adaptación en código

Las tres cuentas centrales del capítulo (parámetros entrenables de LoRA, gradiente de la pérdida y destilación con temperatura) caben en unas pocas líneas.

```

import numpy as np

def softmax(z):
    z = np.asarray(z, dtype=float)
    e = np.exp(z - z.max())
    return e / e.sum()

# 1) Parámetros entrenables: matriz 4096x4096
d_in = d_out = 4096
full = d_in * d_out          # 16.777.216
lora = 16 * (d_in + d_out)   # r=16 -> 131.072
adapter = 2 * d_in * 64      # cuello 64 -> 524.288
print(round(100 * lora / full, 3)) # 0.781 (%)
print(round(100 * adapter / full, 3)) # 3.125 (%)

# 2) Gradiente de la pérdida en los logits: p - y
logits = [2.0, 1.0, 0.0]      # factura, rotos, saludo
target = 1                    # el token correcto es "rotos"
p = softmax(logits)           # [0.665, 0.245, 0.090]
y = np.eye(len(logits))[target]
grad = p - y                  # [0.665, -0.755, 0.090]
loss = -np.log(p[target])     # 1.408
print(np.round(grad, 3), round(loss, 3))

# 3) Destilación: KL entre profesor y estudiante con temperatura T
T = 2.0
teacher = softmax(np.array([4.0, 2.0, 0.0]) / T)
student = softmax(np.array([3.0, 2.5, 0.5]) / T)
kl = float(np.sum(teacher * np.log(teacher / student)))
print(round(kl, 6))           # 0.066603

```

El bloque deja ver tres ideas a la vez. LoRA entrena el 0,78 % de la matriz, no el 100 %. El gradiente `p - y` solo es negativo en el token correcto: «sube este, baja los demás». Y la destilación mide con una KL cuánto se aleja el estudiante de la distribución suave del profesor. Cambia el rango, los logits o la temperatura y verás moverse cada número.

Modelos abiertos: abrir qué, exactamente

La palabra «abierto» se usa demasiado deprisa. Conviene separar piezas:

PIEZA	PREGUNTA QUE DEBES HACER
Pesos	¿Puedo descargar y ejecutar el checkpoint?
Licencia	¿Puedo usarlo comercialmente? ¿Puedo redistribuir adaptaciones?
Código	¿Está el código de entrenamiento o inferencia?
Tokenizer	¿Está disponible y versionado?
Configuración	¿Conozco arquitectura, contexto, formato de chat y parámetros?
Datos	¿Sé qué datos o mezcla de datos se usaron?
Receta de entrenamiento	¿Puedo reproducir el proceso?
Evaluación	¿Hay benchmarks, limitaciones y resultados verificables?
Model card	¿Hay descripción de uso previsto, límites y supuestos?



La Open Source Initiative publicó en 2024 la versión 1.0 de su definición de Open Source AI, centrada en libertades para usar, estudiar, modificar y compartir sistemas de IA.⁴² En la práctica, muchos modelos populares son **open weights**: los pesos están disponibles, pero no necesariamente todo el sistema cumple una definición estricta de open source.

Ejemplos de familias abiertas o de pesos disponibles, con matices:

FAMILIA	QUÉ ILUSTR	MATIZ
Llama 3	Pesos y documentación técnica ampliamente usados. ⁴³	Licencia propia; no equivale automáticamente a open source completo.
Mistral 7B	Modelo eficiente de 7B con publicación técnica. ⁴⁴	Hay que mirar licencia y versión concreta.
Qwen2.5	Familia de modelos con informe técnico amplio. ⁴⁵	Cada tamaño y variante puede tener condiciones propias.

La conclusión importante: abierto no es binario. Hay grados de apertura y permisos concretos. Para uso profesional, mirar solo si «se puede descargar» es insuficiente.

Qué elegir según el caso

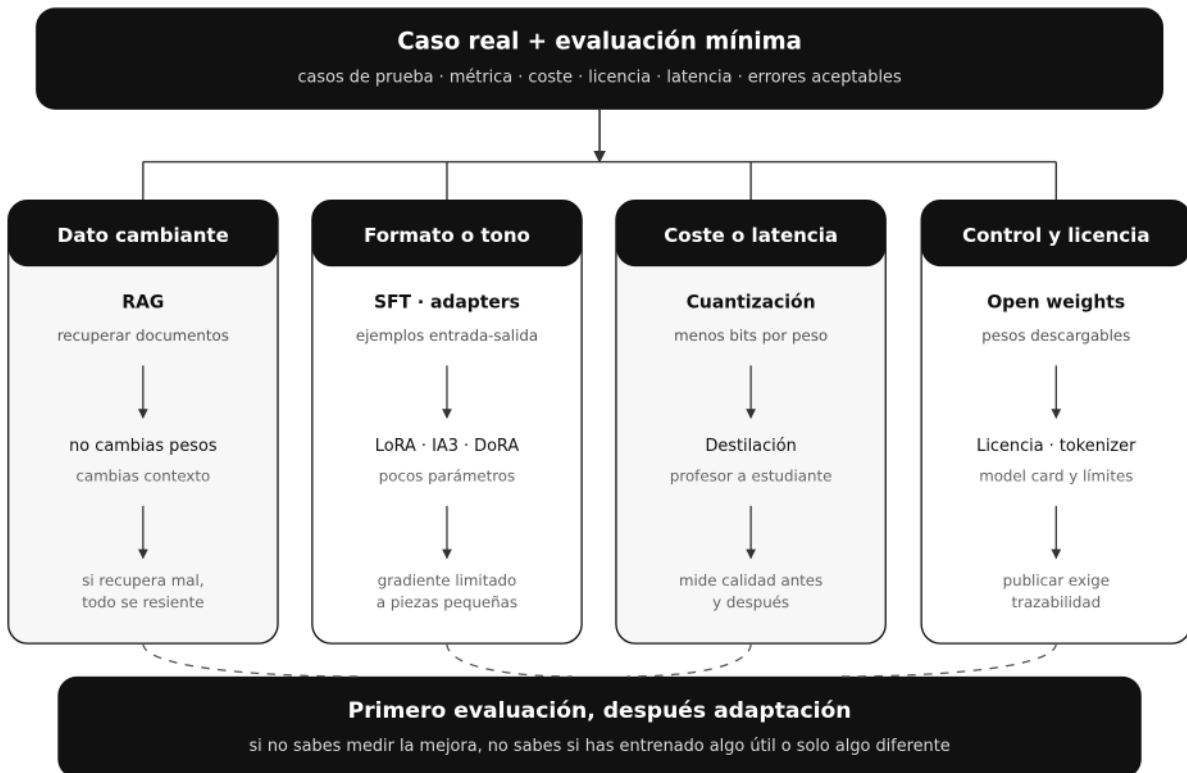
SITUACIÓN	PRIMERA OPCIÓN RAZONABLE	POR QUÉ
Necesitas datos actualizados o privados.	RAG antes que fine-tuning.	No quieres reentrenar para cada documento nuevo.
Necesitas formato muy específico.	Instruction tuning o ejemplos muy cuidados.	Enseñas patrón de salida.
Tienes pocos recursos de entrenamiento.	LoRA o QLoRA.	Entrenas pocos parámetros.
Necesitas baja latencia o edge.	Destilación, cuantización o modelo pequeño.	Reducen coste de inferencia.
Necesitas control local.	Modelo open weights con licencia compatible.	Puedes ejecutar y adaptar en tu infraestructura.
Necesitas máxima calidad general.	Modelo grande servido por API o modelo abierto fuerte bien evaluado.	A veces el coste compensa.

La decisión profesional no es «fine-tuning sí o no». Es una secuencia:

```
definir tarea -> crear evaluación -> probar base -> probar prompt/RAG -> adaptar si hace falta -> comprimir si compensa -> revisar licencia
```

Elegir técnica de adaptación

La pregunta no es "LoRA sí o no", sino qué problema estás intentando resolver.



IA para gente curiosa / Facsímil 03 / Capítulo 06 / 686f6c61

En el día a día

Un equipo de soporte. Tiene 3 000 respuestas históricas y quiere que el modelo use un tono concreto. Primero conviene crear evaluación y probar prompt/RAG. Si el problema es estilo y formato, LoRA puede bastar. Si el problema es conocimiento cambiante, RAG será más natural.

Un producto local. Necesita funcionar en una máquina pequeña. Aquí pesan cuantización y destilación. Tal vez un modelo de 7B cuantizado o un estudiante destilado sea más útil que un modelo enorme que no cabe en memoria.

Una empresa con restricciones de licencia. No basta con que el modelo esté en internet. Hay que revisar licencia, redistribución, uso comercial, obligación de atribución, datos, documentación y compatibilidad con el producto.

Un laboratorio universitario. Puede usar modelos abiertos para enseñar, reproducir experimentos y comparar adaptaciones. Pero debe enseñar también la diferencia entre pesos descargables y apertura completa.

Por qué debería importarte

Porque adaptar mal sale caro. Puedes gastar semanas afinando un modelo cuando bastaba con recuperar documentos. Puedes comprimir demasiado y perder justo la capacidad que necesitabas. Puedes usar un modelo con licencia incompatible con tu producto. O puedes entrenar LoRA sin evaluación y creer que ha mejorado porque responde con más confianza.

La buena noticia es que el mapa es manejable: primero define tarea y métricas; luego prueba una base; después decide si necesitas adaptación, destilación, cuantización o un modelo abierto específico. La arquitectura importa, pero el proceso de decisión importa igual.

También importa no enamorarse de una sigla. LoRA puede ser perfecta para un caso y excesiva para otro. Prompt tuning puede bastar si la tarea es estrecha. DPO puede ayudar cuando hay preferencias claras entre dos respuestas, pero no arregla una base de datos mala. Destilar puede ahorrar dinero, pero solo si el estudiante conserva lo que de verdad usas. La técnica correcta suele ser la más pequeña que resuelve el problema medido.

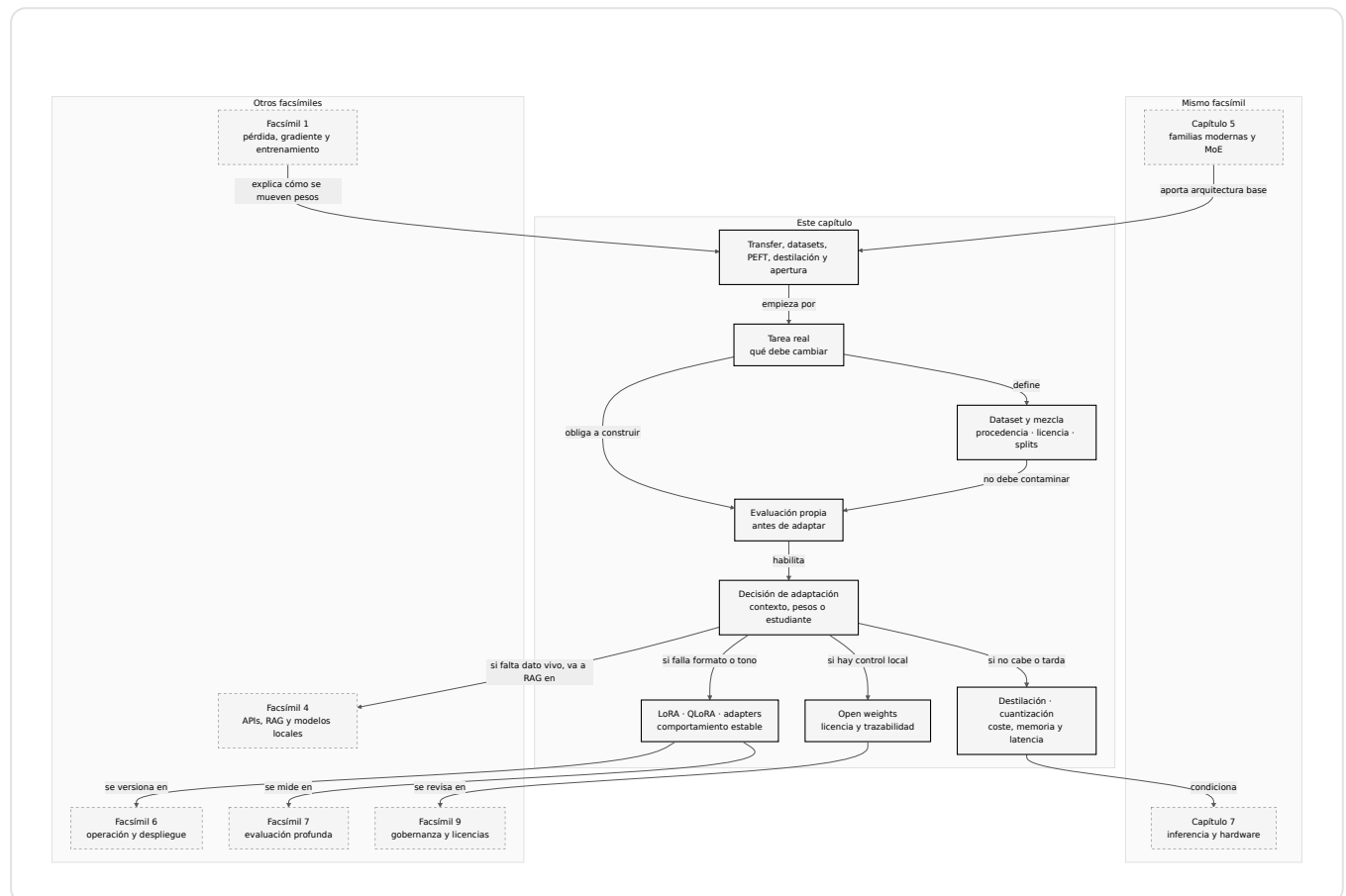
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Tratar un dataset como una lista de ejemplos	El dataset define formato, pérdida, cobertura, sesgos, permisos y trazabilidad. Es parte de la arquitectura.	Léelo como leerías una model card: procedencia, estructura, licencia, mezcla y versión.
Mezclar entrenamiento y evaluación	Si el modelo ve tus preguntas de test, puede mejorar la nota sin mejorar la capacidad real.	Mantén splits separados, guarda hashes y crea conjuntos de regresión aparte.
Usar fine-tuning para meter conocimiento cambiante	Si los datos cambian cada semana, reentrenar no es la forma más cómoda de mantenerlos vivos.	Pregunta primero si el problema es conocimiento, formato, estilo o razonamiento.
Confundir LoRA con modelo completo	LoRA suele ser una adaptación que necesita el modelo base compatible.	Guarda siempre base, adaptador, tokenizer, configuración y versión.
Pensar que destilar conserva todo	Un estudiante pequeño puede perder capacidades fuera del dominio usado para destilar.	Evalúa dentro y fuera de la tarea principal.
Decir open source cuando solo hay pesos	Descargar pesos no implica conocer datos, receta, licencia completa ni reproducibilidad.	Usa categorías: pesos abiertos, código abierto, datos abiertos, sistema abierto.
Cuantizar sin medir	Menos bits ahorran memoria, pero pueden afectar calidad en tareas sensibles.	Compara respuestas, latencia, memoria y errores antes y después.

Cómo encaja todo

Este mapa conecta adaptación con todo el temario. El capítulo 6 no vive aislado: usa pérdidas y gradientes del facsímil 1, familias de arquitectura del capítulo 5 y prepara inferencia, operación, evaluación y gobernanza.

La idea que debe quedar es operativa: antes de tocar pesos hay que definir tarea, dataset, evaluación y coste. Después se decide qué parte se mueve: contexto, recuperador, adaptador, estudiante, precisión o modelo base.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Transfer learning	Reutilizar un modelo entrenado como punto de partida para otra tarea.
Fine-tuning	Ajustar pesos de un modelo base con datos específicos.
PEFT	Familia de técnicas de ajuste eficiente de parámetros.
Gradiente de pérdida	Señal numérica que indica cómo cambiar parámetros para reducir el error.
Instruction tuning	Ajuste con ejemplos de instrucciones y respuestas deseadas.
Adapter	Módulos pequeños añadidos al modelo para adaptar tareas sin tocar toda la base.
Prompt tuning	Vectores de prompt continuos aprendidos que condicionan el modelo congelado.
Prefix tuning	Vectores de prefijo aprendidos para condicionar la generación sin actualizar todos los pesos.
P-tuning v2	Prompt tuning profundo con prompts continuos en varias capas.
BitFit	Ajuste de sesgos del modelo como técnica mínima de adaptación.
LoRA	Adaptación de bajo rango que entrena matrices pequeñas con el modelo base congelado.
QLoRA	LoRA sobre pesos base cuantizados para reducir memoria de entrenamiento.
IA3	Ajuste con vectores que escalan activaciones internas.
AdaLoRA	Variante que reparte el presupuesto de bajo rango según la importancia de cada matriz.
DoRA	Variante de LoRA que separa magnitud y dirección del peso para mejorar la adaptación.
DPO	Ajuste con pares de respuestas preferidas y no preferidas.
Destilación	Entrenar un modelo pequeño usando señales de un modelo mayor.
Cuantización	Reducir bits por peso o activación para ahorrar memoria y coste.
Open weights	Pesos descargables, no necesariamente apertura completa.

TÉRMINO	DEFINICIÓN
Modelo abierto	Modelo con permisos, documentación y artefactos suficientes para uso, estudio y adaptación.
Dataset de preentrenamiento	Corpus grande usado para aprender patrones generales antes de adaptar.
Dataset SFT	Pares de instrucción-respuesta para enseñar formato, tono y seguimiento de tareas.
Dataset de preferencias	Comparaciones entre respuestas para enseñar criterios de elección.
Dataset de razonamiento	Problemas con solución, pasos o verificador para entrenar o medir resolución estructurada.
Dataset multimodal	Datos que combinan texto con imágenes, audio, vídeo u otras señales.
Data mixture	Proporción de familias de datos en entrenamiento.
Data contamination	Presencia de ejemplos de evaluación dentro del entrenamiento.
Olvido catastrófico	Pérdida de capacidades previas cuando se adapta un modelo con datos estrechos.
Fusión de adaptadores	Sumar un adaptador LoRA a los pesos base para servir un único modelo sin coste extra en inferencia.
Cuantización post-entrenamiento	Reducir bits de un modelo ya entrenado con métodos como GPTQ, AWQ o NF4.

Antes de pasar página

- ☐ ¿Puedo explicar la diferencia entre prompt, RAG y fine-tuning? (Si no, vuelve a «Qué no es adaptar un modelo».)
- ☐ ¿Puedo explicar qué dataset usaría para preentrenamiento, SFT, preferencias, razonamiento, código y multimodalidad? (Si no, vuelve a «Datasets».)
- ☐ ¿Sé distinguir dataset de entrenamiento y dataset de evaluación? (Si no, vuelve a «Tipos de dataset con ejemplos».)
- ☐ ¿Sé leer un dataset como una decisión de arquitectura: procedencia, licencia, formato, filtros, mezcla y trazabilidad? (Si no, vuelve a «Qué debe mirar un ingeniero».)
- ☐ ¿Entiendo qué significa $p_i - y_i$ en el gradiente de una pérdida? (Si no, vuelve a «La pérdida y los gradientes».)

- ☐ ¿Puedo distinguir LoRA, QLoRA, adapters, prompt tuning, IA3, AdaLoRA, DoRA y DPO sin quedarme solo con siglas? (Si no, vuelve a «No todo es LoRA: mapa de técnicas de adaptación».)
- ☐ ¿Sé cuándo probar LoRA antes que fine-tuning completo? (Si no, vuelve a «LoRA».)
- ☐ ¿Puedo calcular cuántos parámetros entrena LoRA en una matriz pequeña? (Si no, vuelve a «LoRA: adaptar sin tocar todo».)
- ☐ ¿Entiendo por qué QLoRA reduce memoria? (Si no, vuelve a «QLoRA».)
- ☐ ¿Sé qué es el olvido catastrófico y por qué congelar la base con PEFT lo reduce? (Si no, vuelve a «El olvido catastrófico».)
- ☐ ¿Distingo GPTQ, AWQ, NF4 y GGUF y para qué sirve cada uno? (Si no, vuelve a «Cuantización por dentro».)
- ☐ ¿Conozco alternativas a DPO como KTO, ORPO o SimPO según los datos que tengo? (Si no, vuelve a «Más allá de DPO».)
- ☐ ¿He reproducido en código los parámetros de LoRA, el gradiente y la KL de destilación? (Si no, vuelve a «Modo ingeniero».)
- ☐ ¿Puedo explicar qué aprende un estudiante en destilación? (Si no, vuelve a «Destilación».)
- ☐ ¿Sé distinguir open weights de open source completo? (Si no, vuelve a «Modelos abiertos».)
- ☐ ¿Sé qué cambia al variar el rango de LoRA, los bits de cuantización o la temperatura de destilación? (Si no, vuelve a «LoRA» y «Destilación».)
- ☐ ¿Puedo conectar adaptación, compresión y apertura con inferencia y evaluación? (Si no, vuelve a «Cómo encaja todo».)

En resumen

IDEA FUERZA	DETALLE
Casi nadie entrena un LLM grande desde cero.	Lo habitual es reutilizar una base y adaptarla con datos, instrucciones o herramientas.
Los datasets son arquitectura.	Preentrenamiento, SFT, preferencias, razonamiento, código y multimodalidad tienen formatos, pérdidas y riesgos distintos.
La pérdida baja moviendo gradientes.	Cada técnica decide qué partes reciben esa señal: todo el modelo, adaptadores, matrices LoRA, prompts continuos o un estudiante.
LoRA y QLoRA cambian la escala del ajuste, pero no están solas.	Adapters, prompt tuning, prefix tuning, BitFit, IA3, AdaLoRA, DoRA y DPO son piezas distintas del mismo mapa.
Destilación y cuantización comprimen, pero hay que medir.	Ahorran coste y memoria, aunque pueden perder capacidades.
Abierto no significa solo descargable.	Pesos, licencia, tokenizer, datos, receta, evaluación y model card importan.

Para saber más

Dettmers, T., Pagnoni, A., Holtzman, A. y Zettlemoyer, L. (2023). QLoRA: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems* 36.

<https://arxiv.org/abs/2305.14314>

Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, 4171-4186. <https://doi.org/10.18653/v1/N19-1423>

Gao, L. y otros (2020). *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*. <https://arxiv.org/abs/2101.00027>

Longpre, S. y otros (2023). *The Flan Collection: Designing Data and Methods for Effective Instruction Tuning*. <https://arxiv.org/abs/2301.13688>

Databricks. (2023). *databricks-dolly-15k*. <https://huggingface.co/datasets/databricks/databricks-dolly-15k>

Bai, Y. y otros (2022). *Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback*. <https://arxiv.org/abs/2204.05862>

- Cui, G. y otros (2023). *UltraFeedback: Boosting Language Models with High-quality Feedback*. <https://arxiv.org/abs/2310.01377>
- Cobbe, K. y otros (2021). *Training Verifiers to Solve Math Word Problems*. <https://arxiv.org/abs/2110.14168>
- Kocetkov, D. y otros (2022). *The Stack: 3 TB of permissively licensed source code*. <https://arxiv.org/abs/2211.15533>
- Schuhmann, C. y otros (2022). *LAION-5B: An open large-scale dataset for training next generation image-text models*. <https://arxiv.org/abs/2210.08402>
- Liu, H. y otros (2023). *Visual Instruction Tuning*. <https://arxiv.org/abs/2304.08485>
- Hendrycks, D. y otros (2021). *Measuring Massive Multitask Language Understanding*. <https://arxiv.org/abs/2009.03300>
- Thakur, N. y otros (2021). *BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models*. <https://arxiv.org/abs/2104.08663>
- Muennighoff, N. y otros (2023). *MTEB: Massive Text Embedding Benchmark*. <https://arxiv.org/abs/2210.07316>
- Dubey, A. y otros (2024). *The Llama 3 herd of models*. <https://arxiv.org/abs/2407.21783>
- Hinton, G., Vinyals, O. y Dean, J. (2015). *Distilling the knowledge in a neural network*. <https://arxiv.org/abs/1503.02531>
- Houlsby, N. y otros (2019). Parameter-efficient transfer learning for NLP. *International Conference on Machine Learning*. <https://arxiv.org/abs/1902.00751>
- Hu, E. J. y otros (2022). LoRA: Low-rank adaptation of large language models. *International Conference on Learning Representations*. <https://arxiv.org/abs/2106.09685>
- Jacob, B. y otros (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2704-2713. <https://doi.org/10.1109/CVPR.2018.00286>
- Jiang, A. Q. y otros (2023). *Mistral 7B*. <https://arxiv.org/abs/2310.06825>
- Lester, B., Al-Rfou, R. y Constant, N. (2021). The power of scale for parameter-efficient prompt tuning. *Proceedings of EMNLP*, 3045-3059. <https://doi.org/10.18653/v1/2021.emnlp-main.243>
- Li, X. L. y Liang, P. (2021). Prefix-tuning: Optimizing continuous prompts for generation. *Proceedings of ACL*, 4582-4597. <https://doi.org/10.18653/v1/2021.acl-long.353>

Liu, H. y otros (2022). *Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning*. <https://arxiv.org/abs/2205.05638>

Liu, S.-Y. y otros (2024). DoRA: Weight-decomposed low-rank adaptation. *International Conference on Machine Learning*. <https://arxiv.org/abs/2402.09353>

Liu, X. y otros (2022). P-Tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *Proceedings of ACL*. <https://arxiv.org/abs/2110.07602>

Open Source Initiative. (2024). *The Open Source AI Definition -- 1.0*. <https://opensource.org/ai/open-source-ai-definition>

Ouyang, L. y otros (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35, 27730-27744. <https://arxiv.org/abs/2203.02155>

Rafailov, R. y otros (2023). Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems* 36. <https://arxiv.org/abs/2305.18290>

Raffel, C. y otros (2020). Exploring the limits of transfer learning with a unified text-to-text Transformer. *Journal of Machine Learning Research*, 21(140), 1-67. <https://www.jmlr.org/papers/v21/20-074.html>

Sanh, V., Debut, L., Chaumond, J. y Wolf, T. (2019). *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. <https://arxiv.org/abs/1910.01108>

Yang, A. y otros (2024). *Qwen2.5 technical report*. <https://arxiv.org/abs/2412.15115>

Ben-Zaken, E., Ravfogel, S. y Goldberg, Y. (2022). BitFit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *Proceedings of ACL*. <https://arxiv.org/abs/2106.10199>

Zhang, Q. y otros (2023). AdaLoRA: Adaptive budget allocation for parameter-efficient fine-tuning. *International Conference on Learning Representations*. <https://arxiv.org/abs/2303.10512>

Notas

1. Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, 4171-4186. <https://doi.org/10.18653/v1/N19-1423>. BERT popularizó el esquema de preentrenar representaciones de lenguaje y ajustarlas en tareas posteriores. ↩

2. Raffel, C. y otros (2020). Exploring the limits of transfer learning with a unified text-to-text Transformer. *Journal of Machine Learning Research*, 21(140), 1-67.
<https://www.jmlr.org/papers/v21/20-074.html>. T5 estudia transferencia en un marco unificado donde muchas tareas se expresan como entrada de texto y salida de texto. ↵
3. Ouyang, L. y otros (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems 35*, 27730-27744.
<https://arxiv.org/abs/2203.02155>. El trabajo muestra cómo el ajuste con instrucciones y preferencias humanas mejora la capacidad de seguir peticiones. ↵
4. Raffel y otros (2020) crearon C4 para T5 como corpus web filtrado. Gao, L. y otros (2020). *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*.
<https://arxiv.org/abs/2101.00027>. ↵
5. Kocetkov, D. y otros (2022). *The Stack: 3 TB of permissively licensed source code*.
<https://arxiv.org/abs/2211.15533>. ↵
6. Longpre, S. y otros (2023). *The Flan Collection*. <https://arxiv.org/abs/2301.13688>. Databricks. (2023). *databricks-dolly-15k*. <https://huggingface.co/datasets/databricks/databricks-dolly-15k>. ↵
7. Bai, Y. y otros (2022). *Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback*. <https://arxiv.org/abs/2204.05862>. Cui, G. y otros (2023). *UltraFeedback*. <https://arxiv.org/abs/2310.01377>. ↵
8. Cobbe, K. y otros (2021). *Training Verifiers to Solve Math Word Problems*.
<https://arxiv.org/abs/2110.14168>. ↵
9. Schuhmann, C. y otros (2022). *LAION-5B*. <https://arxiv.org/abs/2210.08402>. Liu, H. y otros (2023). *Visual Instruction Tuning*. <https://arxiv.org/abs/2304.08485>. ↵
10. Hendrycks, D. y otros (2021). *Measuring Massive Multitask Language Understanding*.
<https://arxiv.org/abs/2009.03300>. BEIR y MTEB aparecen en los capítulos de embeddings y evaluación. ↵
11. Hugging Face. (2026). *DataTrove*. <https://github.com/huggingface/datatrove>. La documentación lo presenta como una librería para procesar, filtrar y deduplicar texto a gran escala, con bloques de estadísticas, filtros, tokens y deduplicación. ↵
12. NVIDIA. (2026). *Overview of NeMo Curator*. <https://docs.nvidia.com/nemo/curator/latest/about>. La documentación describe una plataforma abierta para curation de datos escalable y consciente de privacidad para datasets de entrenamiento de IA generativa. ↵
13. Cleanlab. (2026). *Discover Bad Data/Responses in any LLM/Human-Written Dataset*.
https://help.cleanlab.ai/tlm/use-cases/instruction_tuning_data/. El tutorial muestra cómo puntuar pares prompt-respuesta y localizar ejemplos de baja calidad en datasets de instruction tuning o evals. ↵

- .4. Deepchecks. (2026). *Welcome to Deepchecks*. <https://docs.deepchecks.com/0.8/getting-started/welcome.html>. La documentación lo define como herramienta para validar datos y modelos: integridad, distribuciones, splits y comparación entre modelos. ↵
- .5. Hugging Face. (2026). *Evaluate*. <https://huggingface.co/docs/evaluate/index>. La documentación lo presenta como librería para evaluar modelos y datasets con métodos de evaluación de NLP, visión, RL y otros dominios. ↵
- .6. Hugging Face. (2026). *Lighteval*. <https://huggingface.co/docs/lighteval/index>. La documentación lo describe como toolkit para evaluar LLMs con distintos backends, tareas, métricas y resultados detallados. ↵
- .7. EleutherAI. (2026). *Language Model Evaluation Harness*. <https://github.com/EleutherAI/lm-evaluation-harness>. El repositorio lo describe como framework unificado para probar modelos generativos en muchas tareas, con benchmarks públicos y soporte para modelos locales, APIs y adaptadores. ↵
- .8. Liang, P. y otros (2022). *Holistic Evaluation of Language Models*. <https://arxiv.org/abs/2211.09110>. HELM propone evaluar modelos con múltiples escenarios y métricas para evitar una lectura estrecha del rendimiento. ↵
- .9. McCloskey, M. y Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation*, 24, 109-165. El trabajo describe cómo entrenar una red en una tarea nueva puede borrar lo aprendido en una anterior. ↵
- .10. Kirkpatrick, J. y otros (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13), 3521-3526. <https://doi.org/10.1073/pnas.1611835114>. El método estima la importancia de cada peso y penaliza moverlo si era relevante para tareas anteriores. ↵
- .11. Houlsby, N. y otros (2019). Parameter-efficient transfer learning for NLP. *International Conference on Machine Learning*. <https://arxiv.org/abs/1902.00751>. El trabajo propone adaptar modelos de lenguaje con módulos pequeños manteniendo la mayoría de parámetros congelados. ↵
- .12. Lester, B., Al-Rfou, R. y Constant, N. (2021). The power of scale for parameter-efficient prompt tuning. *Proceedings of EMNLP*, 3045-3059. <https://doi.org/10.18653/v1/2021.emnlp-main.243>. El método aprende soft prompts manteniendo congelado el modelo base. ↵
- .13. Li, X. L. y Liang, P. (2021). Prefix-tuning: Optimizing continuous prompts for generation. *Proceedings of ACL*, 4582-4597. <https://doi.org/10.18653/v1/2021.acl-long.353>. Prefix tuning optimiza vectores continuos de prefijo con el modelo base congelado. ↵
- .14. Liu, X. y otros (2022). P-Tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *Proceedings of ACL*. <https://arxiv.org/abs/2110.07602>. P-tuning v2 muestra que prompts continuos bien optimizados pueden acercarse al fine-tuning con pocos parámetros. ↵

- !5. Ben-Zaken, E., Ravfogel, S. y Goldberg, Y. (2022). BitFit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *Proceedings of ACL*.
<https://arxiv.org/abs/2106.10199>. BitFit ajusta solo términos de sesgo y aun así puede competir en algunos escenarios. ↵
- !6. Liu, H. y otros (2022). *Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning*. <https://arxiv.org/abs/2205.05638>. El trabajo introduce IA3, que aprende vectores de escala para adaptar activaciones con pocos parámetros. ↵
- !7. Zhang, Q. y otros (2023). AdaLoRA: Adaptive budget allocation for parameter-efficient fine-tuning. *International Conference on Learning Representations*.
<https://arxiv.org/abs/2303.10512>. AdaLoRA reparte el presupuesto entrenable según la importancia estimada de cada matriz. ↵
- !8. Liu, S.-Y. y otros (2024). DoRA: Weight-decomposed low-rank adaptation. *International Conference on Machine Learning*. <https://arxiv.org/abs/2402.09353>. DoRA separa magnitud y dirección para acercarse más al fine-tuning completo con pocos parámetros. ↵
- !9. Rafailov, R. y otros (2023). Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems 36*.
<https://arxiv.org/abs/2305.18290>. DPO optimiza preferencias directamente sin entrenar un modelo de recompensa separado. ↵
- !0. Rafailov, R. y otros (2023). Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems 36*.
<https://arxiv.org/abs/2305.18290>. ↵
- !1. Ethayarajh, K. y otros (2024). *KTO: Model alignment as prospect theoretic optimization*.
<https://arxiv.org/abs/2402.01306>. ↵
- !2. Hong, J., Lee, N. y Thorne, J. (2024). *ORPO: Monolithic preference optimization without reference model*. <https://arxiv.org/abs/2403.07691>. ↵
- !3. Meng, Y., Xia, M. y Chen, D. (2024). *SimPO: Simple preference optimization with a reference-free reward*. <https://arxiv.org/abs/2405.14734>. ↵
- !4. Hu, E. J. y otros (2022). LoRA: Low-rank adaptation of large language models. *International Conference on Learning Representations*. <https://arxiv.org/abs/2106.09685>. LoRA congela los pesos preentrenados e introduce matrices entrenables de bajo rango para adaptar modelos grandes con muchos menos parámetros. ↵
- !5. Dettmers, T., Pagnoni, A., Holtzman, A. y Zettlemoyer, L. (2023). QLoRA: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems 36*.
<https://arxiv.org/abs/2305.14314>. QLoRA reduce memoria al ajustar modelos grandes cuantizados en 4 bits mientras entrena adaptadores de bajo rango. ↵

16. Frantar, E., Ashkboos, S., Hoefler, T. y Alistarh, D. (2022). *GPTQ: Accurate post-training quantization for generative pre-trained transformers*. <https://arxiv.org/abs/2210.17323>. El método cuantiza con información de segundo orden para reducir la pérdida de calidad a 3-4 bits. ↵
17. Lin, J. y otros (2023). *AWQ: Activation-aware weight quantization for LLM compression and acceleration*. <https://arxiv.org/abs/2306.00978>. AWQ preserva un pequeño porcentaje de pesos salientes según las activaciones para mantener calidad. ↵
18. Dettmers, T. y otros (2023). *QLoRA: Efficient finetuning of quantized LLMs*. <https://arxiv.org/abs/2305.14314>. Introduce NF4 y la doble cuantización para entrenar adaptadores sobre una base de 4 bits. ↵
19. Xiao, G. y otros (2023). *SmoothQuant: Accurate and efficient post-training quantization for large language models*. <https://arxiv.org/abs/2211.10438>. Desplaza dificultad de las activaciones a los pesos para habilitar cuantización entera de ambos. ↵
20. Hinton, G., Vinyals, O. y Dean, J. (2015). *Distilling the knowledge in a neural network*. <https://arxiv.org/abs/1503.02531>. El trabajo propone entrenar modelos más pequeños usando la distribución de salida de modelos más grandes o conjuntos de modelos. ↵
21. Sanh, V., Debut, L., Chaumond, J. y Wolf, T. (2019). *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. <https://arxiv.org/abs/1910.01108>. DistilBERT usa destilación durante preentrenamiento para obtener un modelo más pequeño y rápido que conserva gran parte de las capacidades de BERT. ↵
22. Open Source Initiative. (2024). *The Open Source AI Definition -- 1.0*. <https://opensource.org/ai/open-source-ai-definition>. La definición busca aclarar qué condiciones debería cumplir un sistema de IA para considerarse open source. ↵
23. Dubey, A. y otros (2024). *The Llama 3 herd of models*. <https://arxiv.org/abs/2407.21783>. El informe presenta modelos Llama 3 preentrenados y postentrenados, incluyendo tamaños grandes y variantes para distintos usos. ↵
24. Jiang, A. Q. y otros (2023). *Mistral 7B*. <https://arxiv.org/abs/2310.06825>. El trabajo presenta un modelo de 7B con decisiones de eficiencia como sliding window attention y grouped-query attention. ↵
25. Yang, A. y otros (2024). *Qwen2.5 technical report*. <https://arxiv.org/abs/2412.15115>. El informe describe la familia Qwen2.5 y resultados en múltiples tareas. ↵

Capítulo 07: Inferencia optimizada, edge AI y hardware

La IA también es esperar

Hay una escena muy común: eliges un modelo, haces una demo, todo parece brillante... y luego alguien lo prueba con diez usuarios a la vez. El primer token tarda demasiado. La respuesta completa llega tarde. La GPU se llena aunque el modelo «cabía». El portátil se calienta. La factura sube. El problema ya no es si el modelo sabe contestar; es si puede contestar **a tiempo, con memoria suficiente y a un coste razonable**.

Este capítulo trata de esa parte menos vistosa y absolutamente decisiva: la inferencia. En el capítulo anterior hablamos de adaptar, destilar, cuantizar y elegir modelos abiertos. Ahora preguntamos qué pasa cuando ese modelo se usa de verdad: cuánta memoria consume, qué limita la velocidad, qué pinta la caché KV, por qué el hardware importa y cuándo tiene sentido llevar IA al dispositivo.

La idea de fondo es sencilla: un modelo útil no es solo el que responde bien. Es el que responde bien **dentro de tus restricciones**.

Estado del arte con fecha de corte

Fecha de corte: 10 de junio de 2026.

Fuentes consultadas ese día: papers de FlashAttention, PagedAttention/vLLM, speculative decoding, GQA y cuantización; documentación vigente de vLLM, TensorRT-LLM, llama.cpp y MLPerf Inference; y una lectura pedagógica de Turing Post sobre QKV y KV cache.

Esta sección no pretende congelar el mercado. Los nombres de runtimes, formatos y aceleradores cambian. Lo que sí es más estable son las tensiones técnicas: memoria, ancho de banda, lotes, caché KV, latencia, calidad y coste.

CAPA DEL PROBLEMA	ESTADO OBSERVADO A 10 DE JUNIO DE 2026	QUÉ DEBES MIRAR
Kernels de atención	FlashAttention y variantes IO-aware son referencia para reducir movimiento de memoria en atención exacta. ¹	Si tu runtime usa kernels modernos para tu GPU y tu longitud de contexto.
KV cache	PagedAttention y cachés paginadas reducen desperdicio y fragmentación al servir muchas peticiones. ²	Memoria real con batch, contexto largo y usuarios simultáneos.
Planificación de peticiones	Continuous batching, chunked prefill, prefix caching y separación prefill/decode aparecen en runtimes modernos como vLLM y TensorRT-LLM. ³	Latencia p50/p95, no solo tokens/s máximos.
Decodificación	Speculative decoding acelera cuando un modelo pequeño propone tokens que el grande puede aceptar. ⁴	Si la tasa de aceptación compensa el coste del modelo auxiliar.
Arquitectura del modelo	MQA y GQA reducen el número de cabezas de clave/valor y, por tanto, la memoria de KV cache. ⁵	No solo tamaño en parámetros: también número de capas, contexto y cabezas KV.
Cuantización	INT8, FP8, INT4, GPTQ, AWQ, SmoothQuant y GGUF conviven según hardware, runtime y objetivo. ⁶	Calidad tras cuantizar, memoria, velocidad y soporte real del runtime.
Edge AI	llama.cpp/GGUF, MLX, ONNX Runtime, NPUs y GPUs integradas permiten casos locales, pero con límites térmicos y de memoria. ⁷	Si necesitas privacidad, offline, baja latencia local o menor coste por uso.
Benchmarking	MLPerf Inference sigue siendo una referencia neutral para medir sistemas completos, no solo modelos. ⁸	Comparar sistema completo: modelo, runtime, hardware, batch, precisión y latencia.

La lectura corta: a fecha de corte, optimizar inferencia no significa tocar un botón. Significa coordinar modelo, caché, kernel, cuantización, scheduler, hardware y evaluación.

Qué no es optimizar inferencia

Optimizar inferencia no es solo elegir un modelo más pequeño. Un modelo pequeño puede ser más rápido, sí, pero quizá no tenga la calidad mínima. Un modelo grande puede ser aceptable si usa buen batching, buena caché y cuantización adecuada. La pregunta no es «pequeño o grande», sino «qué combinación cumple calidad, latencia, memoria y coste».

Tampoco es medir una única petición. Si pruebas un prompt aislado en local, quizá veas 80 tokens/s. En producción, con varias peticiones simultáneas, contexto largo y respuestas de distinta longitud, la historia cambia. Importan la cola, los percentiles, el tamaño de lote, el tiempo hasta primer token y el comportamiento cuando el sistema se llena.

Y optimizar no es perseguir el número más alto de tokens por segundo. A veces un sistema con menos throughput bruto da mejor experiencia porque entrega antes el primer token. O porque no se cae cuando llegan peticiones largas. O porque mantiene calidad tras cuantizar.

El bucle real: prefill y decode

Cuando escribes una petición a un LLM, desde fuera parece una sola acción: envías texto y el modelo responde. Por dentro no ocurre así. La inferencia tiene dos momentos con personalidad muy distinta.

Primero está el **prefill**. El modelo lee todo lo que ya existe: instrucciones del sistema, conversación previa, documentos recuperados, pregunta del usuario y cualquier otro token de contexto. En esta fase todavía no está «escribiendo» la respuesta. Está construyendo una representación interna de ese contexto y llenando la caché que usará después. Si el prompt es largo, el prefill pesa mucho. Si metes veinte páginas de documentación en contexto, el coste no aparece solo al final: aparece aquí.

Después viene el **decode**. Ahora el modelo ya ha leído el contexto y empieza a generar. Pero un LLM autoregresivo genera de forma secuencial: produce un token, lo añade al contexto, usa ese nuevo contexto para producir el siguiente, y así sucesivamente. Por eso generar 300 tokens no es como calcular una tabla de 300 celdas independientes; es una cadena donde cada eslabón depende del anterior.

Una forma humana de verlo: el prefill es leer el enunciado completo antes de contestar; el decode es dictar la respuesta palabra a palabra. Leer mucho puede retrasar el arranque. Dictar mucho puede alargar el final.

También hay una diferencia técnica importante: el prefill suele aprovechar mejor el paralelismo porque procesa muchos tokens de entrada juntos; el decode tiene menos margen, porque el token $t + 1$ depende de haber generado antes el token t . Por eso algunas mejoras optimizan el arranque de la respuesta y otras optimizan la velocidad de escritura. Si mezclamos ambas fases en una sola métrica, dejamos de ver dónde duele de verdad.

FASE	QUÉ HACE	QUÉ SUELE LIMITAR
Prefill	Procesa todos los tokens de entrada y construye la KV cache.	Cómputo y longitud del contexto.
Decode	Genera tokens nuevos uno a uno usando la KV cache.	Memoria, ancho de banda y planificación.

Esta separación explica por qué dos usuarios pueden sentir experiencias muy distintas con el mismo modelo. Un usuario con un prompt corto y una respuesta larga puede ver el primer token pronto, pero esperar bastante hasta el final. Otro usuario con un prompt enorme puede esperar mucho al principio aunque luego la respuesta salga a buen ritmo.

Piénsalo con situaciones concretas. En una asesoría, alguien pega un contrato de treinta páginas y pregunta «¿qué cláusulas debo revisar primero?». Ahí el prefill manda: el sistema tiene que leer mucho antes de empezar. En una app de escritura, alguien pide «redáctame una propuesta completa de dos páginas». El contexto puede ser corto, pero el decode pesa: hay que producir muchos tokens seguidos. En un panel de soporte, veinte personas preguntan cosas parecidas con las mismas instrucciones del sistema. Ahí no basta con un modelo rápido; importa reutilizar prefijos, organizar lotes y no desperdiciar KV cache.

Si el usuario manda una entrada de n_{entrada} tokens y queremos generar n_{salida} tokens, conviene separar lectura de contexto y generación.

Ejemplo de fórmula. Como calculadora de primer orden, no como benchmark universal, podemos escribir:

$$T_{\text{total}} \approx T_{\text{prefill}} + n_{\text{salida}} \cdot T_{\text{decode}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
T_{total}	Tiempo total de respuesta.	7,5 segundos.
T_{prefill}	Tiempo de procesar el contexto inicial.	0,85 segundos.
n_{salida}	Tokens generados.	300 tokens.
T_{decode}	Tiempo medio por token generado.	0,022 segundos/token.

En producto solemos mirar dos métricas porque responden a preguntas distintas. La primera mide cuándo empieza la sensación de respuesta; la segunda mide cuánto tarda en completarse.

MÉTRICA	PREGUNTA QUE RESPONDE
TTFT (<i>time to first token</i>)	¿Cuánto tarda el usuario en ver que algo empieza?
Tokens/s	¿A qué velocidad se completa la respuesta una vez empieza?

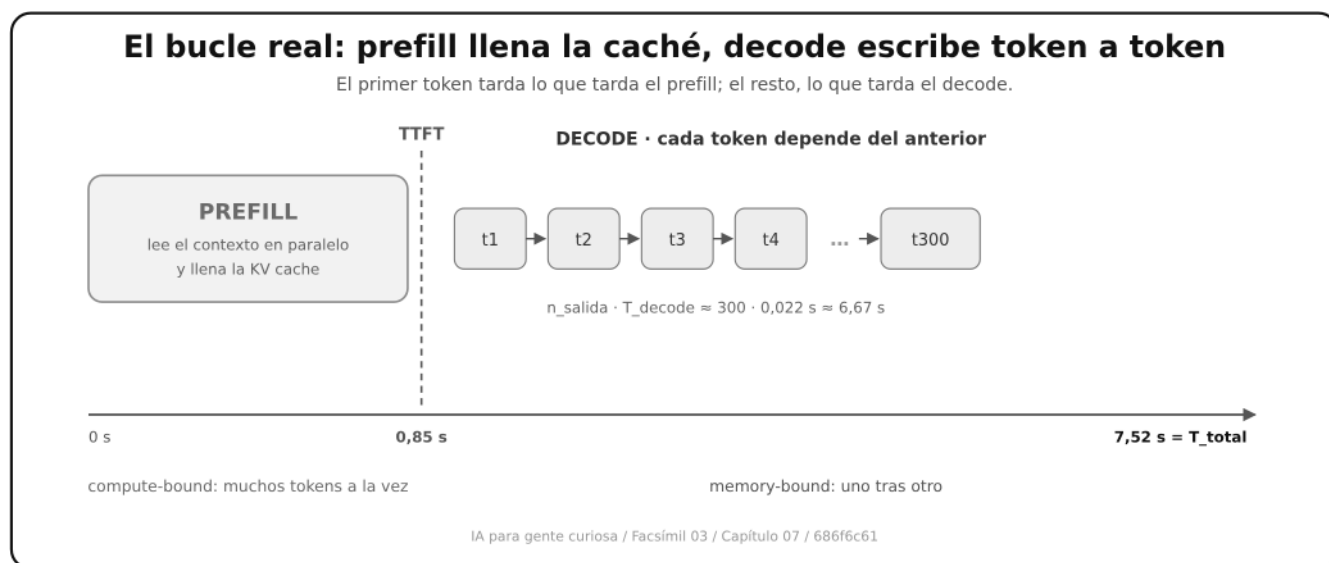
Con tokens por segundo, la misma idea se escribe así:

$$T_{\text{total}} \approx \frac{\text{TTFT}_{ms}}{1000} + \frac{n_{\text{salida}}}{\text{tokens/s}}$$

Ejemplo:

$$T_{\text{total}} \approx 0,85 + \frac{300}{45} \approx 7,52 \text{ s}$$

Esto explica una sensación habitual: el sistema puede «empezar rápido» pero terminar lento, o empezar lento pero luego escribir deprisa. Ambas cosas importan.



La memoria invisible: KV cache

En los capítulos de atención vimos Q , K y V . Durante inferencia, las claves y valores del contexto pasado se vuelven especialmente importantes. Si el modelo ya procesó los primeros 4096 tokens, no queremos recalcularlos enteros cada vez que genera un token nuevo. Sería como releer todo el libro antes de escribir cada palabra de un resumen.

La solución es guardar parte de ese trabajo en una **KV cache**. Esa caché permite que el decode consulte el pasado sin recomputarlo desde cero. A cambio, consume memoria. Y esa memoria crece con cuatro cosas que suelen crecer justo cuando el producto tiene éxito: más

usuarios simultáneos, más contexto, más capas y más cabezas de clave/valor.

Alyona Vert lo explica con una imagen muy clara: durante generación autoregresiva, las claves y valores de tokens anteriores se guardan para que el modelo no tenga que recalcular todo el pasado en cada nuevo token.⁹ Esa lectura no sustituye los papers de inferencia, pero ayuda a que la intuición técnica aterrice.

Ejemplo de fórmula. Una aproximación de memoria para KV cache es:

$$M_{KV} \approx 2 \cdot L \cdot B \cdot S \cdot H_{KV} \cdot d_h \cdot \text{bytes}$$

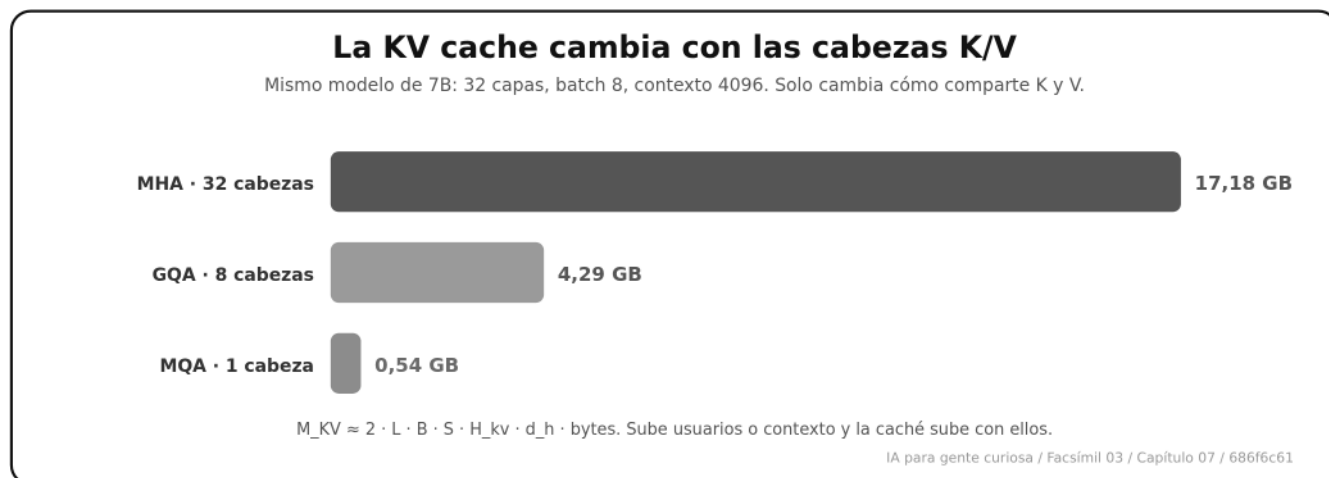
El factor 2 aparece porque guardamos claves y valores. L son capas, B es batch o conversaciones simultáneas, S es longitud de contexto, H_{KV} son cabezas de clave/valor, d_h es dimensión por cabeza y `bytes` depende de la precisión usada. Esta aproximación ayuda a detectar órdenes de magnitud; el consumo real depende del runtime, paginación de caché, fragmentación, buffers, activaciones, paralelismo y margen operativo.

SÍMBOLO	SIGNIFICADO	EJEMPLO
2	Guardamos claves y valores.	K y V .
L	Número de capas.	32 capas.
B	Batch o peticiones activas.	8 peticiones.
S	Tokens de contexto mantenidos.	4096 tokens.
H_{KV}	Cabezas de clave/valor.	32 en MHA, 8 en GQA, 1 en MQA.
d_h	Dimensión por cabeza.	128.
bytes	Bytes por valor.	2 para FP16/BF16.

Con $L = 32$, $B = 8$, $S = 4096$, $d_h = 128$ y 2 bytes por valor, el mismo modelo puede tener memorias de caché muy distintas según cómo organice sus cabezas K/V :

ATENCIÓN	H_{KV}	KV CACHE APROXIMADA
Multi-head attention	32	17,18 GB
Grouped-query attention	8	4,29 GB
Multi-query attention	1	0,54 GB

Aquí aparece una lección importante: dos modelos con los mismos parámetros pueden tener costes de inferencia distintos si gestionan de forma distinta K y V . Por eso GQA o MQA no son detalles académicos: afectan directamente a cuántas conversaciones caben a la vez.



Otro ejemplo sencillo: imagina un asistente de una universidad que responde dudas de matrícula. El modelo cabe en memoria cuando lo prueba una persona. Pero el primer lunes de septiembre entran cien estudiantes con conversaciones largas, documentos de normativa y respuestas a medio generar. Lo que antes era «el modelo cabe» se convierte en «el modelo más la caché de todas las conversaciones no cabe». La KV cache es ese coste que no se ve en la ficha del modelo, pero aparece justo cuando el sistema empieza a usarse de verdad.

Por qué el decode va más lento: memory-bound y compute-bound

Hay una pregunta que da una intuición muy potente: cuando la GPU genera un token, ¿qué la frena, hacer cuentas o mover datos? La respuesta separa dos regímenes que el modelo *roofline* hizo famosos.¹⁰

En el **prefill**, la GPU procesa muchos tokens de entrada a la vez. Reutiliza cada peso del modelo para muchos tokens, así que hace muchas operaciones por cada byte que lee de memoria. Está **compute-bound**: el límite es la potencia de cálculo. La GPU trabaja a tope.

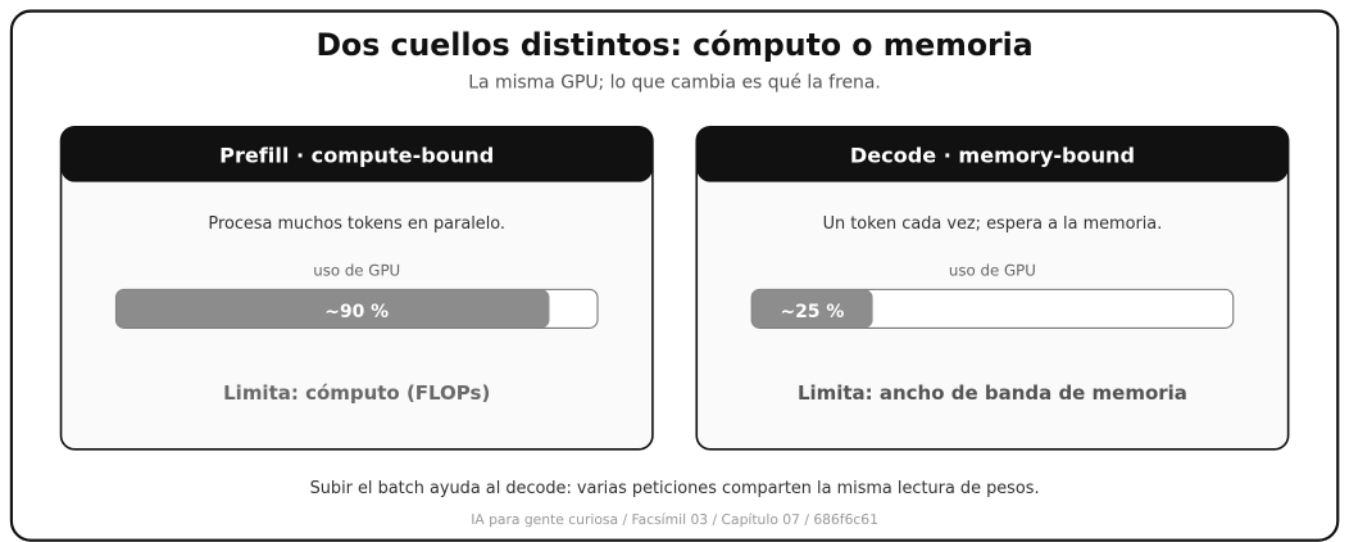
En el **decode**, la GPU genera un solo token cada vez. Para producirlo tiene que leer todos los pesos del modelo y toda la KV cache de memoria, y con esa lectura enorme hace muy pocas operaciones. Está **memory-bound**: el límite es el ancho de banda de memoria, no el cálculo. La GPU pasa buena parte del tiempo esperando datos, medio vacía.

FASE	RÉGIMEN	QUÉ LA FRENA	POR QUÉ
Prefill	Compute-bound	Cómputo (FLOPs)	Muchos tokens por lectura de pesos: mucha cuenta por byte.
Decode	Memory-bound	Ancho de banda de memoria	Un token por lectura de todos los pesos: poca cuenta por byte.

Esta idea reordena toda la lista de técnicas. Si el decode está limitado por mover datos, entonces:

- **Subir el batch** ayuda mucho: varias peticiones comparten la misma lectura de pesos, así que cada byte leído sirve para más tokens.
- **Cuantizar** ayuda al decode aunque no cambie la calidad: menos bits por peso son menos bytes que mover.
- **Speculative decoding** ayuda: el modelo grande verifica varios tokens propuestos en una sola pasada de lectura, en vez de uno por pasada.
- **GQA y MQA** ayudan: menos cabezas K/V son menos KV cache que leer en cada token.

No es casualidad que casi todas las optimizaciones de decode ataquen la memoria, no el cálculo. Saber en qué régimen estás te dice qué palanca mover.

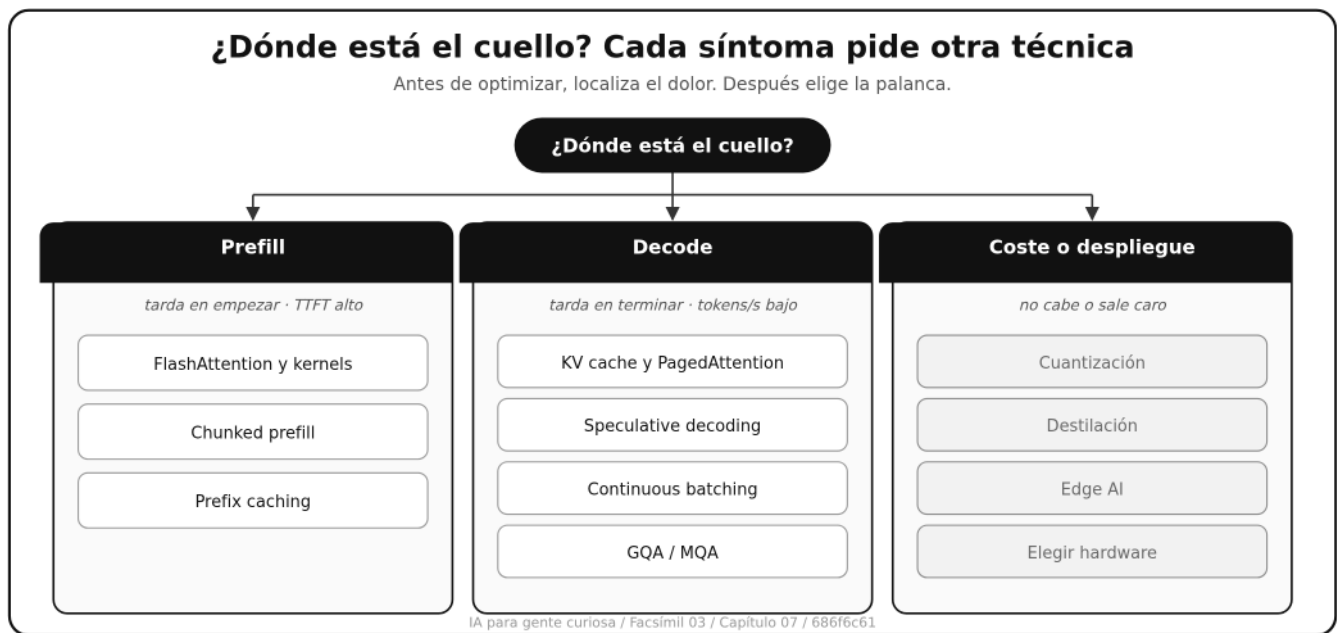


veces el decode es lento porque cada token depende del anterior. Y a veces el modelo simplemente no cabe.

Por eso las técnicas de optimización no son intercambiables. Cada una toca una parte distinta del sistema. Antes de elegir una, conviene preguntar: ¿mi problema está en leer el contexto, en generar tokens, en servir muchos usuarios, en memoria, en coste o en calidad?

TÉCNICA	QUÉ MEJORA	INTUICIÓN
FlashAttention	Atención más rápida y con menos memoria intermedia.	No cambia la fórmula; cambia cómo se mueven los datos.
PagedAttention	Gestión de KV cache con menos desperdicio.	Trata la caché como páginas, no como bloques rígidos enormes.
Continuous batching	Throughput con muchas peticiones.	Entra y sale gente del lote sin esperar a que todas las respuestas terminen.
Prefix caching	Peticiones con prefijos repetidos.	Si muchas consultas comparten sistema o contexto, no recalculas todo.
Speculative decoding	Menos tiempo por tokens generados.	Un modelo pequeño propone; el grande verifica.
Cuantización	Memoria y a veces velocidad.	Menos bits por peso o activación.
GQA/MQA	Memoria de KV cache.	Menos cabezas K/V compartidas entre cabezas de consulta.
Distilación	Tamaño y coste.	Un estudiante más pequeño imita una parte útil del profesor.

Si el cuello está en el **prefill**, suelen importar kernels de atención, longitud de contexto, chunked prefill y reutilización de prefijos. Si el cuello está en el **decode**, suelen importar KV cache, speculative decoding, batching y ancho de banda de memoria. Si el cuello está en **coste o despliegue**, entran cuantización, destilación, edge y elección de hardware.



No todas se suman limpiamente. Una técnica puede mejorar throughput y empeorar TTFT. Otra puede funcionar bien en una GPU y no tener soporte real en otra. Por eso el estado del arte no se adopta leyendo una lista: se adopta midiendo tu caso.

En una empresa pequeña, por ejemplo, puede parecer natural pasar de una API a un modelo local para ahorrar. Si cada consulta usa un prompt enorme con políticas internas, quizá el ahorro se pierde porque el prefill consume mucho. Si las consultas comparten el mismo contexto base, prefix caching puede cambiar la película. Si el problema es que cada respuesta debe ser larga, speculative decoding puede ayudar más que seguir recortando bits. La técnica correcta depende de dónde está el dolor.

Continuous batching: no dejar la GPU a medias

El batching clásico junta varias peticiones en un lote y las procesa a la vez. El problema aparece cuando tienen longitudes distintas: el lote no termina hasta que acaba la respuesta más larga, y mientras tanto las peticiones que ya terminaron dejan su sitio vacío. La GPU sigue ocupada, pero haciendo trabajo inútil.

El **continuous batching** (también llamado *in-flight batching*) rompe esa rigidez. En vez de esperar a que todo el lote termine, cuando una petición acaba, su hueco se rellena al instante con otra que estaba en cola. La GPU no espera: siempre tiene tokens útiles que generar. Por eso es una de las mayores ganancias de throughput en runtimes modernos como vLLM y TensorRT-LLM, y suele importar más que cambiar de modelo cuando el cuello es servir muchas peticiones a la vez.

Continuous batching: no dejar la GPU a medias

El estático espera a que todas terminen; el continuo rellena el hueco al instante.

Batching estático



Continuous batching



IA para gente curiosa / Facsímil 03 / Capítulo 07 / 686f6c61

El detalle fino: continuous batching mejora mucho el throughput, pero hay que vigilar el TTFT. Si el sistema mete demasiadas peticiones nuevas en mitad del decode de otras, el primer token de alguien puede retrasarse. Por eso los runtimes combinan esta idea con chunked prefill y límites de lote: el objetivo no es solo «más tokens por segundo», sino un buen equilibrio entre throughput y latencia percibida.

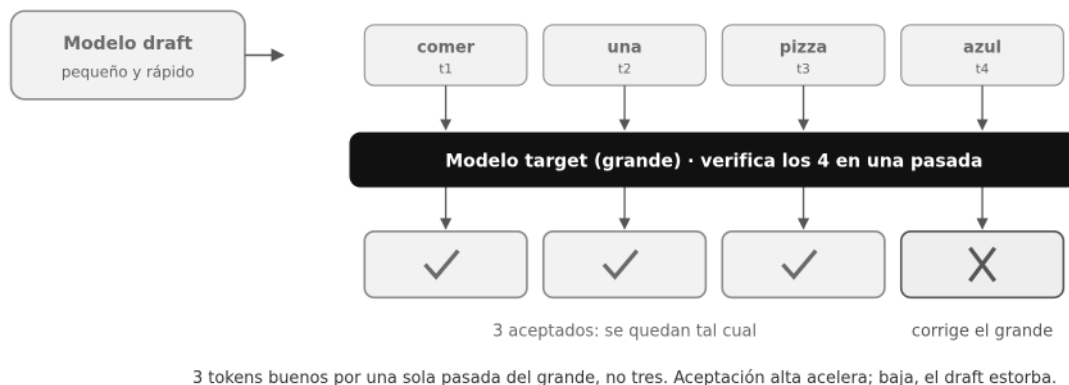
Speculative decoding: proponer y verificar

El decode es lento porque es secuencial: un token por pasada del modelo grande. La idea del **speculative decoding** es romper esa secuencialidad sin cambiar el resultado. Un modelo pequeño y barato (el *draft*) propone varios tokens de golpe; el modelo grande (el *target*) los verifica todos en una sola pasada, en paralelo. Acepta el tramo inicial que coincide con lo que él habría generado y descarta el resto, corrigiendo el primer fallo con su propio token.

La clave es que verificar varios tokens en paralelo cuesta casi lo mismo que generar uno, porque el decode es memory-bound: la lectura de pesos ya estaba pagada. Si el draft acierta tres de cada cuatro, avanzas tres tokens por pasada del grande en vez de uno. No mejora la calidad (la distribución final es la del target); mejora la velocidad, y solo si la **tasa de aceptación** compensa el coste de ejecutar el draft.

Speculative decoding: el pequeño propone, el grande verifica

Varios tokens aceptados en una sola pasada del modelo grande. Acelera sin cambiar la calidad.



IA para gente curiosa / Facsímil 03 / Capítulo 07 / 686f6c61

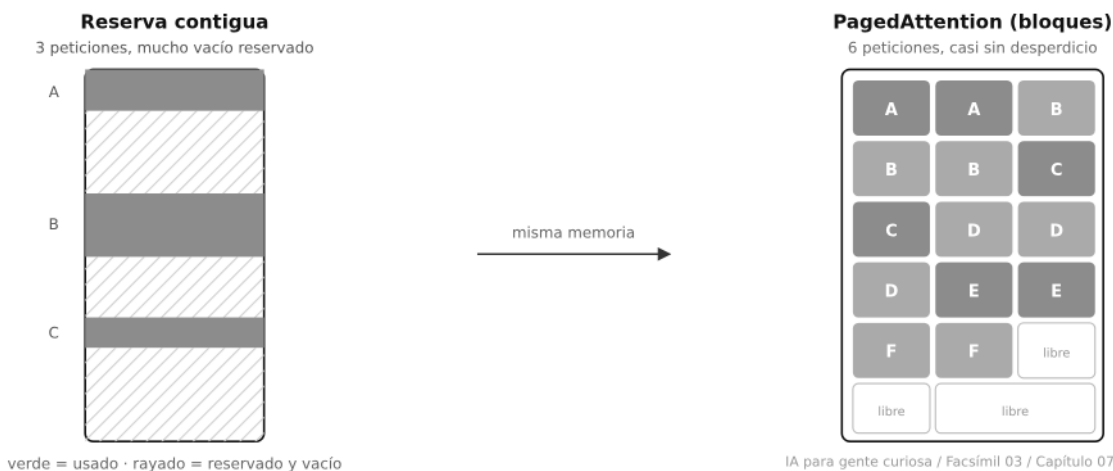
PagedAttention: la KV cache como páginas

La KV cache tiene un problema de gestión de memoria, no solo de tamaño. La forma ingenua de reservarla es dar a cada petición un bloque contiguo del tamaño de su contexto máximo, por si crece. Pero la mayoría de respuestas no llegan a ese máximo, así que queda mucha memoria reservada y vacía: fragmentación interna. Con esa reserva pesimista caben pocas conversaciones a la vez.

PagedAttention, la idea sobre la que se construyó vLLM, aplica algo que los sistemas operativos llevan décadas haciendo con la memoria virtual: dividir la KV cache en bloques pequeños de tamaño fijo (páginas) y asignarlos bajo demanda según crece cada secuencia. No hace falta memoria contigua ni reservar el máximo por adelantado. El desperdicio baja casi a cero y, en la misma GPU, caben muchas más peticiones simultáneas.

PagedAttention: la KV cache como páginas

Reservar el máximo por petición desperdicia memoria; paginar reserva solo lo usado.



IA para gente curiosa / Facsímil 03 / Capítulo 07 / 686f6c61

Modo ingeniero: el presupuesto de serving en código

Las tres cuentas del capítulo (memoria de pesos, KV cache y latencia con concurrencia) caben en unas líneas.

```
# 1) Memoria de pesos: N * bits / 8
N = 7_000_000_000
for bits in (16, 8, 4):
    gb = N * bits / 8 / 1e9
    print(bits, "bits ->", round(gb, 1), "GB")    # 14.0, 7.0, 3.5

# 2) KV cache: 2 * L * B * S * H_kv * d_h * bytes
L, B, S, d_h, byts = 32, 8, 4096, 128, 2
for nombre, h_kv in {"MHA": 32, "GQA": 8, "MQA": 1}.items():
    gb = 2 * L * B * S * h_kv * d_h * byts / 1e9
    print(nombre, round(gb, 2), "GB")            # 17.18, 4.29, 0.54

# 3) Latencia: TTFT + tokens_salida / (tokens_por_segundo)
ttft_s, n_out = 0.85, 300
solo = ttft_s + n_out / 45                      # 7.52 s, un usuario
capacidad, usuarios = 180, 12
compartido = ttft_s + n_out / (capacidad / usuarios) # 20.85 s, capacidad repartida
print(round(solo, 2), round(compartido, 2))
```

Tres lecciones en un vistazo. Los pesos bajan con los bits, sí, pero la KV cache de MHA (17 GB) puede pesar más que los propios pesos en 16 bits (14 GB). Y la latencia que siente un usuario aislado (7,5 s) casi se triplica (20,85 s) cuando doce comparten la misma capacidad. Cambia `h_kv`, `S`, `B` o `usuarios` y verás moverse justo lo que duele en producción.

Edge AI: cuando el modelo vive cerca

Edge AI significa ejecutar el modelo cerca del usuario, del sensor o del dispositivo: portátil, móvil, mini PC, servidor local, navegador, fábrica, aula o consulta. No siempre es mejor. Es una decisión de producto e infraestructura.

La pregunta no es «¿puedo correrlo local?», sino «¿qué gano al correrlo local y qué pierdo?». A veces ganas privacidad y respuesta sin red. A veces pierdes calidad, facilidad de actualización o estabilidad térmica. La tabla ayuda a separar motivos razonables de entusiasmo tecnológico.

MOTIVO	POR QUÉ PUEDE INTERESAR
Offline	El sistema sigue funcionando sin red.
Latencia local	Evitas ida y vuelta a un servidor lejano.
Control de datos	Algunos datos no salen del dispositivo o de la red local.
Coste marginal	Muchas inferencias locales pueden salir más baratas que llamadas remotas.
Personalización	El sistema puede adaptarse al entorno concreto.

Pero edge también trae límites:

LÍMITE	QUÉ IMPLICA
Memoria	El modelo debe caber junto con KV cache y aplicación.
Energía	Un portátil o móvil no puede sostener carga alta indefinidamente.
Temperatura	El rendimiento puede bajar si el dispositivo se calienta.
Runtimes	No todos los formatos corren igual en CPU, GPU, NPU o Apple Silicon.
Mantenimiento	Actualizar modelos locales puede ser más complejo que cambiar una API.

Una NPU no convierte cualquier modelo en viable. Una GPU no siempre gana. Una CPU moderna con un modelo cuantizado puede ser suficiente para una tarea pequeña. La pregunta buena es: ¿qué calidad necesito, con qué latencia, para cuántos usuarios, durante cuánto tiempo y con qué presupuesto térmico?

Un caso cercano: el asistente de almacén

Imagina una empresa con un almacén grande. Quiere un asistente que lea incidencias de pedidos y proponga la siguiente acción: reponer, avisar, revisar lote, generar etiqueta o pedir foto.

Hay tres diseños posibles:

DISEÑO	VENTAJA	COSTE
API externa potente	Calidad alta y mantenimiento simple.	Depende de red y coste por uso.
Servidor local con GPU	Control y buen rendimiento para muchos puestos.	Compra, operación y monitorización.
Modelo cuantizado en cada terminal	Funciona incluso con mala red.	Menor calidad y límites de memoria/temperatura.

No hay una respuesta universal. Si el almacén necesita funcionar sin conexión, edge pesa mucho. Si la tarea cambia cada semana y necesita consultar políticas vivas, RAG y servidor central pueden ser más naturales. Si hay muchas peticiones simultáneas, la clave quizá no sea el modelo, sino batching, KV cache y colas.

Ahora bajémoslo un poco más. Si cada terminal manda una foto convertida a descripción, el número de tokens de entrada puede ser pequeño y el decode de la acción recomendada domina poco. Un modelo local pequeño puede bastar. Si cada incidencia arrastra historial de cliente, condiciones de entrega, normativa interna y correos anteriores, el prefill se vuelve caro y conviene pensar en servidor, caché de prefijos y recuperación selectiva. Si el turno de mañana genera cientos de incidencias a la vez, el problema ya no es «qué modelo responde mejor», sino cómo se ordenan las peticiones para que ninguna persona espere demasiado.

En el día a día

En una aplicación de soporte. El usuario no mide tu media. Siente la espera. Si el primer token tarda cinco segundos, la experiencia parece rota aunque luego escriba rápido. TTFT importa tanto como tokens/s.

En un dashboard interno. Muchas personas lanzan consultas parecidas por la mañana. Prefix caching y batching pueden valer más que cambiar de modelo, porque el cuello está en repetir contexto y servir picos.

En un producto local. Un modelo GGUF de 4 bits puede ser perfecto para resumir notas privadas en un portátil, pero quizá no sirva para razonamiento largo. La decisión no es ideológica; es medición.

En una demo ejecutiva. Una sola petición luce bien. Lo profesional es enseñar también qué pasa con 20 peticiones, contexto largo y una respuesta de 500 tokens. Ahí aparece la verdad del sistema.

Por qué debería importarte

Porque la arquitectura se vuelve coste. El número de capas, la longitud de contexto, el tipo de atención, los bits de precisión, la KV cache y el runtime acaban convertidos en euros, segundos, vatios y temperatura.

También porque la optimización sin evaluación puede engañar. Si cuantizas y el modelo responde más rápido pero falla justo en los casos delicados, no has optimizado: has movido el problema. Si subes batch y mejora throughput pero empeora p95, quizá has hecho feliz al benchmark y triste al usuario.

Dónde solía tropezar yo

Estos son los tropiezos que más distorsionan decisiones reales. Casi todos tienen la misma raíz: mirar una métrica aislada y olvidarse del sistema completo.

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Mirar solo parámetros	Dos modelos de 7B pueden tener inferencias muy distintas por contexto, GQA, runtime y cuantización.	Mide pesos, KV cache, TTFT y tokens/s.
Confundir tokens/s con buena experiencia	El usuario nota mucho el primer token y los percentiles altos.	Mide TTFT, latencia total y p95/p99.
Cuantizar sin comparar calidad	Menos bits pueden romper justo los casos que importan.	Crea una evaluación antes y después.
Olvidar la KV cache	El modelo cabe, pero las conversaciones simultáneas no.	Calcula memoria de pesos y memoria de KV cache.
Creer que edge siempre es mejor	Local puede ser lento, caliente o difícil de actualizar.	Decide por caso: offline, datos, coste, latencia y mantenimiento.

Cómo encaja todo

Este mapa conecta inferencia con los capítulos anteriores y con la operación futura. QKV deja de ser una fórmula: se vuelve caché. Sampling deja de ser un parámetro: se vuelve experiencia de usuario. Cuantización deja de ser una tabla de bits: se vuelve memoria, calidad y coste.

La decisión que prepara el siguiente facsímil es clara: no basta con elegir modelo; hay que elegir dónde vive, cómo se sirve, qué latencia promete y cómo se mide.

```

graph TD
    subgraph Este_capitulo [Este capítulo]
        direction TB
        A[Edge AI  
CPU, GPU, NPU y límites  
términos] -- "elige dónde ejecutar" --> I[Inferencia, edge y  
hardware]
        B[condiciona memoria y  
runtime] --> I
        C[define tamaño y  
contexto] --> I
        I -- "empieza por" --> D[Perfil  
leer contexto y llenar  
caché]
        D -- "llena" --> E["KV cache  
memoria por capas,  
tokens y batch"]
        E -- "alimenta" --> F[Decode  
generar token a token]
        F -- "se percibe como" --> G["Métricas reales  
TTFT · tokens/s · p95 ·  
calidad"]
        G -- "afectan a" --> H["Facsímil 11  
producto y experiencia"]
        I -- "se integra en" --> J["Facsimil 4  
APIs, local, RAG y  
herramientas"]
        I -- "se optimiza con" --> K[Optimización  
kernels, batching,  
cache y especulación]
        K -- "se mejora con" --> F
        K -- "se mejora con" --> G
        F -- "controla salida durante" --> C
        F -- "produce K y V  
reutilizables" --> C
    end

    subgraph Mismo_facsímil [Mismo facsímil]
        direction TB
        I1[Capítulo 6  
modelos abiertos y  
cuantización]
        I2[Capítulo 1  
LLM, contexto y escala]
        I3[Capítulos 2-3  
atención, QKV y  
máscara]
        I4[Capítulo 4  
logits y sampling]
    end

    subgraph Otros_facsímiles [Otros facsímiles]
        direction TB
        J1["Facsimil 6  
operación, colas y  
observabilidad"]
        J2["Facsimil 7  
evaluación y calibración"]
    end

    C --> I1
    C --> I2
    C --> I3
    C --> I4
    C --> J1
    C --> J2
    
```

El diagrama ilustra el flujo de trabajo de un sistema de inferencia de LLM en el edge, dividido en tres secciones principales:

- Este capítulo:** Detalla la arquitectura central. Comienza con la selección de hardware (Edge AI: CPU, GPU, NPU y límites) y la configuración de memoria y runtime. La inferencia se ejecuta en el hardware, condicionada por la memoria y el runtime, y definida por el tamaño y el contexto. El proceso comienza leyendo el contexto y llenando el caché, lo que alimenta el KV cache (memoria por capas, tokens y batch). Este cache alimenta el decodificador (generar token a token), que produce K y V reutilizables y controla la salida durante la inferencia. El decodificador se percibe como métricas reales (TTFT, tokens/s, p95, calidad), que afectan al producto y la experiencia (Facsimil 11). La inferencia también se integra en APIs, local, RAG y herramientas (Facsimil 4) y se optimiza con kernels, batching, cache y especulación (Optimización).
- Mismo facsímil:** Muestra los componentes de capítulos anteriores que influyen en el proceso: Capítulo 6 (modelos abiertos y cuantización), Capítulo 1 (LLM, contexto y escala), Capítulos 2-3 (atención, QKV y máscara) y Capítulo 4 (logits y sampling).
- Otros facsímiles:** Muestra otros componentes que influyen en el proceso: Facsimil 6 (operación, colas y observabilidad) y Facsimil 7 (evaluación y calibración).

Vocabulario aprendido

Nos quedamos con las palabras que conviene tener a mano para leer documentación de runtimes, benchmarks y hardware sin perder el hilo.

TÉRMINO	DEFINICIÓN
Inferencia	Uso de un modelo entrenado para responder a una entrada nueva.
Prefill	Fase que procesa el contexto inicial antes de generar tokens.
Decode	Fase que genera tokens de salida uno a uno.
TTFT	Tiempo hasta recibir el primer token visible.
Throughput	Trabajo procesado por unidad de tiempo, por ejemplo tokens/s.
KV cache	Caché de claves y valores de atención para reutilizar el contexto pasado.
FlashAttention	Kernel de atención que reduce movimiento de memoria.
PagedAttention	Gestión paginada de KV cache para servir muchas peticiones con menos desperdicio.
Continuous batching	Técnica para mezclar peticiones activas y mejorar uso del hardware.
Speculative decoding	Método donde un modelo pequeño propone tokens y el grande los verifica.
Cuantización	Reducir precisión numérica para ahorrar memoria y, a veces, ganar velocidad.
Edge AI	Ejecutar modelos cerca del usuario o dispositivo.
NPU	Acelerador especializado para redes neuronales.
Compute-bound	Régimen limitado por el cálculo; típico del prefill.
Memory-bound	Régimen limitado por el ancho de banda de memoria; típico del decode.

Antes de pasar página

- ☐ ¿Puedo explicar la diferencia entre prefill y decode? (Si no, vuelve a «El bucle real».)
- ☐ ¿Entiendo por qué la KV cache puede ocupar varios GB? (Si no, vuelve a «La memoria invisible».)
- ☐ ¿Sé distinguir cuándo el decode es memory-bound y el prefill compute-bound? (Si no, vuelve a «Por qué el decode va más lento».)
- ☐ ¿Sé distinguir TTFT, tokens/s y throughput? (Si no, vuelve a «El bucle real».)

- ☐ ¿Entiendo por qué el continuous batching mantiene la GPU ocupada frente al batching estático? (Si no, vuelve a «Continuous batching».)
- ☐ ¿Sé cómo el speculative decoding acelera sin cambiar la calidad y por qué importa la tasa de aceptación? (Si no, vuelve a «Speculative decoding».)
- ☐ ¿Puedo explicar por qué PagedAttention desperdicia menos memoria que la reserva contigua? (Si no, vuelve a «PagedAttention».)
- ☐ ¿He reproducido en código la memoria de pesos, la KV cache y la latencia con concurrencia? (Si no, vuelve a «Modo ingeniero».)
- ☐ ¿Puedo explicar por qué GQA reduce memoria de inferencia? (Si no, vuelve a la tabla de KV cache.)
- ☐ ¿Sé por qué cuantizar no basta sin medir calidad? (Si no, vuelve a «Dónde solía tropezar yo».)
- ☐ ¿Puedo decidir cuándo edge AI tiene sentido? (Si no, vuelve a «Edge AI».)
- ☐ ¿Sé cómo cambian memoria y latencia al variar batch, longitud de secuencia y número de usuarios? (Si no, vuelve a «Modo ingeniero: el presupuesto de serving en código».)

En resumen

La versión corta del capítulo no es una lista de herramientas: es una forma de pensar. Primero entiende qué fase estás pagando, después mide y solo entonces optimiza.

IDEA FUERZA	DETALLE
Inferir no es una llamada simple.	Hay prefill, decode, KV cache, scheduler, kernels y hardware.
La memoria no son solo pesos.	La KV cache puede dominar cuando hay contexto largo y usuarios simultáneos.
Optimizar es medir compromisos.	TTFT, tokens/s, p95, calidad, coste y temperatura cuentan juntos.
Edge AI es una decisión de producto.	Sirve cuando offline, datos, latencia o coste local compensan sus límites.
El estado del arte tiene fecha.	A 10 de junio de 2026, las prácticas cambian rápido; el mecanismo importa más que la marca concreta.

Para saber más

Ainslie, J. y otros (2023). GQA: Training generalized multi-query Transformer models from multi-head checkpoints. *Proceedings of EMNLP*. <https://arxiv.org/abs/2305.13245>

Dao, T., Fu, D. Y., Ermon, S., Rudra, A. y Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems* 35. <https://arxiv.org/abs/2205.14135>

Dettmers, T., Lewis, M., Belkada, Y. y Zettlemoyer, L. (2022). LLM.int8(): 8-bit matrix multiplication for Transformers at scale. *Advances in Neural Information Processing Systems* 35. <https://arxiv.org/abs/2208.07339>

Frantar, E., Ashkboos, S., Hoefler, T. y Alistarh, D. (2022). GPTQ: Accurate post-training quantization for generative pre-trained Transformers. <https://arxiv.org/abs/2210.17323>

ggml-org. (2026). *llama.cpp: LLM inference in C/C++*. Consultado el 10 de junio de 2026. <https://github.com/ggml-org/llama.cpp>

Kwon, W. y otros (2023). Efficient memory management for large language model serving with PagedAttention. *Proceedings of SOSP*. <https://arxiv.org/abs/2309.06180>

Leviathan, Y., Kalman, M. y Matias, Y. (2023). Fast inference from Transformers via speculative decoding. *Proceedings of ICML*. <https://arxiv.org/abs/2211.17192>

Lin, J. y otros (2024). AWQ: Activation-aware weight quantization for LLM compression and acceleration. *Proceedings of Machine Learning and Systems*. <https://arxiv.org/abs/2306.00978>

MLCommons. (2026). *MLPerf Inference: Datacenter benchmark*. Consultado el 10 de junio de 2026. <https://mlcommons.org/benchmarks/inference-datacenter/>

NVIDIA. (2026). *TensorRT-LLM documentation*. Consultado el 10 de junio de 2026. <https://docs.nvidia.com/tensorrt-llm/index.html>

vLLM Project. (2026). *vLLM documentation*. Consultado el 10 de junio de 2026. <https://docs.vllm.ai/en/stable/>

Vert, A. (2026, 13 de mayo). *AI 101: Your Ultimate Guide to Attention: Mechanism, QKV, and KV Cache*. Turing Post. <https://www.turingpost.com/p/your-ultimate-guide-to-attention-mechanism-qkv-and-kv-cache>

Xiao, G. y otros (2023). SmoothQuant: Accurate and efficient post-training quantization for large language models. *Proceedings of ICML*. <https://arxiv.org/abs/2211.10438>

Notas

1. Dao, T., Fu, D. Y., Ermon, S., Rudra, A. y Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems* 35. <https://arxiv.org/abs/2205.14135>. La idea central es optimizar lecturas y escrituras entre memorias rápidas y lentas, no cambiar la matemática de la atención. ↵
2. Kwon, W. y otros (2023). Efficient memory management for large language model serving with PagedAttention. *Proceedings of SOSP*. <https://arxiv.org/abs/2309.06180>. vLLM se construye sobre esta idea para gestionar KV cache con menos memoria desperdiciada. ↵
3. vLLM Project. (2026). *vLLM documentation*. <https://docs.vllm.ai/en/stable/>. Consultado el 10 de junio de 2026. La documentación lista serving online/offline, prefix caching, speculative decoding, cuantización y métricas de producción. ↵
4. Leviathan, Y., Kalman, M. y Matias, Y. (2023). Fast inference from Transformers via speculative decoding. *Proceedings of ICML*. <https://arxiv.org/abs/2211.17192>. El método aprovecha un modelo auxiliar para proponer varios tokens y verificar en paralelo con el modelo principal. ↵
5. Ainslie, J. y otros (2023). GQA: Training generalized multi-query Transformer models from multi-head checkpoints. *Proceedings of EMNLP*. <https://arxiv.org/abs/2305.13245>. GQA ocupa un punto intermedio entre multi-head attention y multi-query attention. ↵
6. Dettmers, T., Lewis, M., Belkada, Y. y Zettlemoyer, L. (2022). LLM.int8(): 8-bit matrix multiplication for Transformers at scale. *Advances in Neural Information Processing Systems* 35. <https://arxiv.org/abs/2208.07339>. El trabajo muestra una ruta para inferencia 8-bit en modelos grandes cuidando valores atípicos. ↵
7. ggml-org. (2026). *llama.cpp: LLM inference in C/C++*. <https://github.com/ggml-org/llama.cpp>. Consultado el 10 de junio de 2026. El proyecto documenta GGUF, cuantización, servidor compatible con OpenAI y herramientas de conversión. ↵
8. MLCommons. (2026). *MLPerf Inference: Datacenter benchmark*. <https://mlcommons.org/benchmarks/inference-datacenter/>. Consultado el 10 de junio de 2026. MLPerf publica resultados, divisiones, categorías de disponibilidad y metadatos de sistemas completos. ↵
9. Vert, A. (2026, 13 de mayo). *AI 101: Your Ultimate Guide to Attention: Mechanism, QKV, and KV Cache*. Turing Post. <https://www.turingpost.com/p/your-ultimate-guide-to-attention-mechanism-qkv-and-kv-cache>. Consultado el 10 de junio de 2026. ↵
10. Williams, S., Waterman, A. y Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4), 65-76. <https://doi.org/10.1145/1498765.1498785>. El modelo relaciona rendimiento con intensidad aritmética y distingue cuándo limita el cómputo y cuándo limita el ancho de banda de memoria. ↵

Capítulo 08: Lo que deberías saber: arquitecturas y modelos

Entrando en el tema

Este facsímil empezó con una pregunta que parece sencilla: ¿qué hay dentro de un LLM? La respuesta no era “un chat”, ni “una API”, ni “un modelo grande”. Era una cadena de decisiones técnicas: tokens, embeddings, tensores, atención, MLP, normalización, logits, sampling, adaptación, memoria, inferencia y hardware.

Este capítulo es una revisión activa. No pretende que memorices nombres. Pretende que puedas mirar una ficha de modelo, una demo de producto o una discusión sobre hardware y preguntar con calma: qué arquitectura hay debajo, qué coste tiene usarla, qué se adaptó, qué se cuantizó, qué se está midiendo y qué parte pertenece al modelo o al sistema que lo sirve.

No es el final de la historia. Es el punto donde dejamos de hablar de modelos como cajas negras y empezamos a hablar de sistemas con piezas.

1. Qué es un LLM: modelo, parámetros y escala

Un LLM es un modelo entrenado para predecir tokens a partir de contexto. No guarda frases como una biblioteca. Ajusta parámetros para transformar una secuencia de entrada en una distribución de probabilidad sobre el siguiente token.¹

La idea que debes conservar es esta: los parámetros son memoria aprendida, el contexto es memoria temporal y la salida es una distribución. Si mezclas esas tres cosas, empiezan los malentendidos.

Caso cercano: una persona pega una política interna y pregunta “¿puedo aprobar este gasto?”. El modelo no “abre una carpeta” donde está la política. Procesa el texto pegado, lo mezcla con lo aprendido durante entrenamiento y produce una respuesta probable. Si falta un dato, puede sonar seguro igualmente. Por eso el contexto y la verificación no son decoración.

Vuelve al [capítulo 01](#) si no puedes explicar la diferencia entre parámetros, contexto, token, embedding y salida.

2. Transformer: de texto a tensores y atención

El Transformer convirtió texto en operaciones paralelizables sobre tensores.² El texto se tokeniza, cada token busca su vector, se añade posición y el bloque Transformer transforma esas representaciones capa tras capa.

La intuición importante: el modelo no lee palabras como las leemos nosotros. Trabaja con matrices. Cada capa reescribe la representación de cada posición usando información de otras posiciones.

Caso cercano: cuando escribes “el banco cerró la cuenta”, el token “banco” cambia de significado según “cuenta”. No basta con un diccionario. La atención permite que las posiciones se miren entre sí para construir significado contextual.

Vuelve al [capítulo 02](#) si no puedes dibujar el camino mínimo: texto → tokens → ids → embeddings → bloques Transformer → logits.

3. Q, K, V, máscara causal y softmax

La atención no es una metáfora bonita: es una operación concreta. Cada posición produce consultas Q , claves K y valores V . Las consultas se comparan con claves, se normalizan con softmax y se usan para mezclar valores.

La forma mínima de la atención escalada es:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{d_k}\right)V$$

SÍMBOL O	QUÉ SIGNIFICA	IMAGEN MENTAL
Q	Qué busca una posición.	“Necesito saber a qué se refiere esto”.
K	Qué ofrece cada posición para ser encontrada.	“Tengo información relevante para cierto tipo de pregunta”.
V	Qué contenido se mezcla si una posición importa.	“Esto es lo que apporto a la respuesta”.
d_k	Dimensión de las claves.	Escala para que softmax no se vuelva extremo demasiado pronto.

La máscara causal impide que una posición mire tokens futuros durante generación. Sin ella, entrenaríamos un modelo que ve la respuesta antes de producirla.

Caso cercano: en una frase como “Marta dejó el portátil en la mesa porque estaba cansada”, la atención ayuda a decidir si “estaba cansada” apunta a Marta o al portátil. No hay una regla universal simple: depende del contexto aprendido y de cómo cada cabeza atiende.

Vuelve al [capítulo 03](#) si no puedes explicar por qué softmax convierte puntuaciones en pesos que suman 1.

4. MLP, residual, LayerNorm, logits y sampling

La atención mezcla información entre posiciones. El MLP transforma cada posición por dentro. La conexión residual mantiene una vía de continuidad. LayerNorm estabiliza escalas. Los logits convierten el último vector en candidatos de token. Sampling decide qué token sale.

El modelo no “elige una palabra” directamente. Produce logits, los pasa por softmax y obtiene una distribución:

$$p_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

SÍMBOLO	QUÉ SIGNIFICA	QUÉ CONTROLA
z_i	Logit del token candidato i .	Preferencia bruta antes de normalizar.
T	Temperatura.	Cuánto se abre o cierra el abanico de opciones.
p_i	Probabilidad final del token i .	Opción que puede muestrearse.

Caso cercano: si un asistente escribe un email formal, temperatura baja puede mantener tono estable. Si le pides diez nombres creativos para un proyecto, una temperatura algo mayor puede abrir opciones. En ambos casos, el modelo no cambia de personalidad: cambia cómo se muestrea la distribución.

Vuelve al [capítulo 04](#) si no puedes explicar la diferencia entre logit, probabilidad, temperatura, top-k y top-p.

5. Arquitecturas modernas: familias, MoE, razonamiento y multimodalidad

El Transformer base no es el único diseño posible. Hay modelos encoder-only, decoder-only, encoder-decoder, MoE, multimodales, híbridos con recuperación, modelos de razonamiento con más cómputo en inferencia y arquitecturas pensadas para contexto largo. BERT popularizó encoder-only para comprensión.³ T5 mostró la potencia de formular muchas tareas como texto-a-texto.⁴

La idea importante no es coleccionar siglas. Es entender qué compromiso cambia cada familia: memoria, velocidad, calidad, entrenabilidad, multimodalidad, coste de servir y facilidad de adaptación.

MoE añade expertos y un router: no se activa todo el modelo para cada token. Esa idea permite aumentar capacidad manteniendo coste activo menor, aunque complica entrenamiento, balanceo y serving.⁵

Caso cercano: una plataforma que atiende dudas legales, técnicas y administrativas puede beneficiarse de rutas internas distintas. Pero si el router se equivoca, no basta con tener “muchos expertos”: hay que medir qué experto se activa, cuándo y con qué resultado.

Vuelve al [capítulo 05](#) si no puedes explicar qué gana y qué complica un modelo MoE.

6. Transfer learning, destilación y modelos abiertos

La mayoría de proyectos no entrena desde cero. Parte de un modelo base, lo adapta o lo rodea de herramientas. Fine-tuning cambia pesos. LoRA entrena matrices pequeñas sobre pesos congelados.⁶ QLoRA permite adaptar con menos memoria combinando cuantización y LoRA.⁷

La pregunta práctica es: ¿quiero cambiar comportamiento, aportar conocimiento, reducir coste o controlar despliegue? Cada objetivo apunta a una técnica distinta.

Caso cercano: una academia quiere que un modelo corrija entregas con una rúbrica concreta. Si el modelo ya entiende el tema pero no sigue bien el formato, quizá bastan ejemplos y validación. Si necesita tono y criterios muy específicos, LoRA puede tener sentido. Si falta información cambiante, RAG será mejor que ajustar pesos.

Vuelve al [capítulo 06](#) si no puedes distinguir fine-tuning, LoRA, QLoRA, destilación, cuantización y modelo abierto.

7. Inferencia optimizada, edge AI y hardware

Un modelo no se usa en abstracto. Se sirve. Y servirlo significa gestionar memoria, colas, caché, kernels, precisión, batch, latencia y coste. El capítulo anterior separó `prefill` y `decode`, que es una de las divisiones más útiles para pensar rendimiento.

Ejemplo de fórmula. Una aproximación sencilla de latencia, útil para discutir producto antes de medir con un runtime real, es:

$$T_{\text{total}} \approx T_{\text{prefill}} + n_{\text{salida}} \cdot T_{\text{decode}}$$

PIEZA	QUÉ PREGUNTA RESPONDE	POR QUÉ IMPORTA
T_{prefill}	¿Cuánto tarda en leer el contexto?	Contextos largos retrasan el primer token.
n_{salida}	¿Cuánto texto generará?	Respuestas largas encarecen decode.
T_{decode}	¿Cuánto cuesta cada token nuevo?	Generar es secuencial y depende de KV cache.

La KV cache puede ocupar varios GB con usuarios simultáneos. FlashAttention reduce movimiento de memoria en atención.⁸ PagedAttention organiza mejor esa caché en serving.⁹ GQA reduce memoria de claves y valores.¹⁰

Caso cercano: si tu asistente legal recibe contratos largos y genera respuestas cortas, el cuello puede estar en prefill. Si tu herramienta redacta informes largos desde un prompt breve, el cuello puede estar en decode. Si veinte personas lo usan a la vez, quizá el problema no es el modelo, sino la KV cache y el scheduler.

Vuelve al [capítulo 07](#) si no puedes explicar TTFT, tokens/s, KV cache y por qué un modelo que cabe solo puede no caber con varios usuarios.

Disecccionando un LLM: de una petición al siguiente token

Hasta ahora hemos repasado el facsímil capítulo a capítulo. Ahora hagamos lo que más se usa en la vida real: seguir una petición completa. No como dibujo bonito, sino como herramienta de depuración. Cuando una respuesta sale lenta, cara, repetitiva, truncada o falsa, necesitas saber en qué capa mirar.

La inspiración visual de esta sección viene de explicaciones interactivas como *The Anatomy of an LLM*, de Roy van Rijn, que recorre tokenización, embeddings, atención, logits, sampling, post-training, KV cache y cuantización en una cadena visual.¹¹ Aquí lo rehacemos con la mirada del facsímil: qué cambia, qué medimos y qué decisión tomaría un equipo.

La escena: una llamada que parece sencilla

Imagina una aplicación de soporte interno. El usuario pregunta:

```
Un alumno pide ampliar el plazo de entrega porque ha tenido una incidencia técnica documentada.
¿Qué categoría, riesgo y siguiente acción recomiendas?
```

La aplicación no manda solo esa frase. Normalmente ensambla una petición con varias capas:

CAPA DE ENTRADA	EJEMPLO	POR QUÉ IMPORTA
Instrucción de sistema	“Responde solo con JSON válido y cita la política usada.”	Define el contrato de comportamiento.
Documentos	Políticas internas sobre ampliaciones e incidencias.	Aportan conocimiento vivo que no debería inventarse.
Mensaje de usuario	La consulta concreta.	Es la tarea que cambia en cada llamada.
Contrato de salida	Campos <code>categoria</code> , <code>riesgo</code> , <code>accion_recomendada</code> , <code>fuentes</code> .	Permite validar automáticamente.
Parámetros	<code>temperature</code> , <code>top_p</code> , <code>max_tokens</code> , <code>stop</code> , <code>seed</code> .	Controlan variabilidad, longitud y parada.

La primera lección práctica es esta: **el prompt real no es el texto que escribe el usuario**. Es el paquete completo que la aplicación construye. Si no registras ese paquete, luego no sabes qué ha visto el modelo.

Fase 1: texto, tokens e IDs

El modelo no recibe letras como tú las ves. Recibe IDs de tokens. El tokenizer parte el texto en piezas y asigna a cada pieza un número. Esa conversión afecta a coste, contexto, truncamiento y rarezas: código, acentos, números largos, JSON y nombres propios pueden partirse de formas poco intuitivas.

LO QUE VE UNA PERSONA	LO QUE NECESITA EL MODELO	RIESGO DE INGENIERÍA
“incidencia técnica documentada”	IDs de tokens	Puede ocupar más tokens de lo esperado.
Un JSON de salida	Tokens con llaves, comillas y campos	Si el modelo añade texto extra, el parser falla.
Un documento largo	Miles de tokens	Puede desplazar instrucciones o fragmentos relevantes.
Código o logs	Subtokens raros	Puede encarecer y romper patrones.

No hay que memorizar IDs. Hay que entender que el presupuesto empieza aquí. Si el tokenizer transforma una petición en 12 000 tokens, eso condiciona latencia, coste y KV cache antes de que el modelo “piense” nada.

Fase 2: embeddings y posición

Un ID de token por sí solo no tiene geometría. El token 15339 no está cerca de 15340 por tener un número parecido. El embedding layer busca una fila en una matriz aprendida y devuelve un vector. Ese vector es el punto de partida del token dentro del modelo.

Una forma sencilla de escribirlo es:

$$x_i = E[t_i] + p_i$$

SÍMBOLO	QUÉ SIGNIFICA	EJEMPLO
t_i	ID del token en la posición i .	ID del token incidencia .
E	Matriz de embeddings aprendida.	Una tabla de tamaño vocabulario × dimensión.
$E[t_i]$	Fila de embedding seleccionada.	Vector de 4096 números en un modelo concreto.
p_i	Información posicional.	RoPE u otra codificación de posición.
x_i	Representación inicial del token.	Vector que entra al primer bloque.

La parte importante: el embedding inicial no es el significado final. El token “plazo” empieza con una representación, pero las capas siguientes la reescriben según “entrega”, “incidencia”, “política” y “ampliación”. La semántica útil aparece al procesar contexto.

Aquí aparece un detalle que en ingeniería se nota mucho: **posición no significa “pegar un número al token y olvidarse”**. En modelos modernos es frecuente usar RoPE, que codifica posición rotando las representaciones de consulta y clave para que la atención capture relaciones relativas entre posiciones.¹² Una forma intuitiva de verlo es:

$$q_i^{\text{rot}} = R(\theta_i)q_i, \quad k_j^{\text{rot}} = R(\theta_j)k_j$$

SÍMBOL O	QUÉ SIGNIFICA	POR QUÉ IMPORTA
q_i	Consulta del token en posición i .	Lo que esa posición busca en el contexto.
k_j	Clave del token en posición j .	Lo que otra posición ofrece para ser encontrada.
$R(\theta_i)$	Rotación dependiente de posición.	Mete orden sin convertir la posición en un simple contador.
$q_i^{\text{rot}}(k_j^{\text{rot}})^{\top}$	Comparación ya sensible a distancia relativa.	Ayuda a que “qué mira a qué” dependa también de dónde está.

RoPE: la posición rota las consultas y las claves

Cada posición gira el vector un ángulo distinto; la atención capta así la distancia relativa.



pos 0



pos 1



pos 2



pos 3



pos 4



pos 5

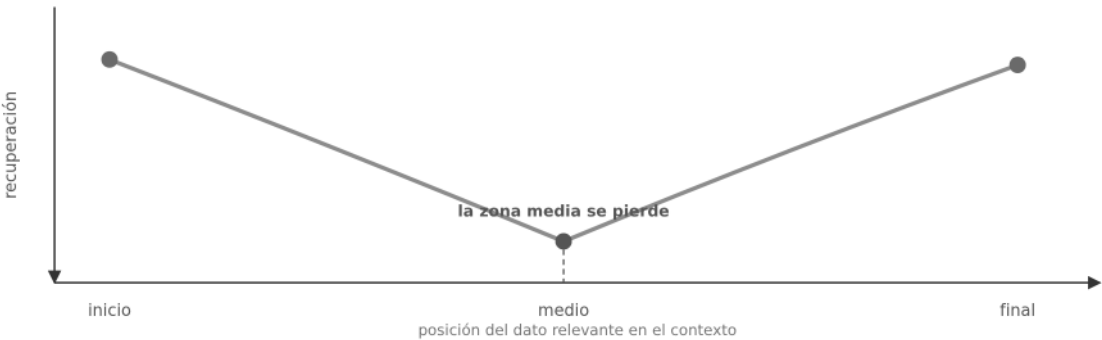
No es un contador pegado al token: es un giro. La comparación $q \cdot k$ depende de la distancia $i - j$, no de la posición absoluta, y por eso «qué mira a qué» tiene en cuenta cuán lejos están.

IA para gente curiosa / Facsímil 03 / Capítulo 08 / 686f6c61

Esto no convierte el contexto largo en memoria perfecta. El trabajo *Lost in the Middle* mostró que los modelos pueden recuperar peor información situada en zonas intermedias de contextos largos que información al principio o al final.¹³ Para una aplicación RAG esto es muy práctico: no basta con “meter más documentos”. Hay que ordenar evidencias, repetir señales críticas, chunkear con criterio y medir recuperación en la posición donde realmente cae el dato.

Lost in the Middle: el contexto largo no es memoria perfecta

La información en mitad de un contexto largo se recupera peor que al principio o al final.



IA para gente curiosa / Facsímil 03 / Capítulo 08 / 686f6c61

Fase 3: bloques Transformer

Un Transformer decoder-only repite un bloque muchas veces. Cada bloque suele tener atención, normalización, conexión residual y MLP. En lenguaje de ingeniería, cada capa toma un tensor de forma aproximada:

```
[batch, secuencia, d_model]
```

y devuelve otro tensor con la misma forma, pero con representaciones más contextualizadas.

PIEZA	QUÉ HACE	CÓMO SE ROMPE EN PRODUCCIÓN
Atención causal	Permite que cada posición use tokens anteriores.	Contexto largo aumenta coste y memoria.
Q, K, V	Calculan qué busca cada token, qué ofrecen los demás y qué contenido se mezcla.	Más heads y contexto implican más trabajo.
Residual	Mantiene una vía de continuidad entre capas.	Sin ella, entrenar redes profundas sería mucho más inestable.
LayerNorm/RMSNorm	Mantiene escalas numéricas manejables.	Cambios de precisión pueden afectar estabilidad.
MLP/SwiGLU/MoE	Reescribe cada posición por dentro.	En MoE aparece routing, balanceo y coste activo.

Si un alumno se queda solo con “la atención mira palabras”, pierde la ingeniería. La atención mueve información entre posiciones; el MLP transforma la representación de cada posición; la normalización estabiliza; la residual conserva señal; y repetir esto muchas capas produce el vector final desde el que saldrán logits.

Hay tres matices que conviene dejar claros porque aparecen mucho cuando se leen implementaciones reales, model cards y artículos técnicos como la guía visual de Kato.¹⁴

1. Las cabezas de atención no son trozos mágicos del texto. Cada head aprende proyecciones distintas:

$$Q_h = XW_h^Q, \quad K_h = XW_h^K, \quad V_h = XW_h^V$$

PIEZ A	QUÉ CAMBIA	LECTURA DE INGENIERÍA
X	Representaciones actuales de los tokens.	No son palabras originales; son vectores ya reescritos por capas previas.
W_h^Q	Proyección de consultas para la cabeza h .	Define qué tipo de relación puede buscar esa cabeza.
W_h^K	Proyección de claves.	Define cómo se ofrece cada posición a ser atendida.
W_h^V	Proyección de valores.	Define qué contenido se mezcla cuando una posición importa.

En producción esto importa porque `attention_heads`, `kv_heads`, `head_dim` y `d_model` no son números decorativos. Condicionan memoria, throughput, compatibilidad con kernels y tamaño de KV cache. GQA, por ejemplo, permite que varias cabezas de consulta compartan menos cabezas de claves y valores; por eso puede reducir memoria de KV cache sin eliminar todas las cabezas de atención.¹⁵

2. El residual stream es la autopista por la que viaja la señal. En muchos decoders modernos, una capa se parece más a esto que a un bloque lineal limpio:

$$x' = x + \text{Attention}(\text{RMSNorm}(x))$$

$$x_{\text{next}} = x' + \text{MLP}(\text{RMSNorm}(x'))$$

RMSNorm normaliza por raíz cuadrática media y prescinde del centrado de LayerNorm, lo que puede simplificar y acelerar ciertos diseños.¹⁶ La idea práctica: si cuantizas, cambias precisión o miras activaciones, no estás tocando “texto”; estás tocando el flujo numérico por

el que cada capa suma pequeñas correcciones. Por eso un modelo puede degradarse de manera rara: no siempre falla todo, a veces se vuelven frágiles ciertos formatos, idiomas, longitudes o tareas.

3. El MLP no es relleno entre atenciones. En muchos modelos, el MLP usa variantes tipo SwiGLU:

$$\text{SwiGLU}(x) = (\text{swish}(xW_{\text{gate}}) \odot xW_{\text{up}})W_{\text{down}}$$

PARTE	QUÉ HACE	EJEMPLO DE INTUICIÓN
W_{gate}	Abre o cierra rutas internas.	“Este patrón semántico sí aplica aquí”.
W_{up}	Expande dimensión intermedia.	Permite más capacidad de transformación.
$\odot$	Producto elemento a elemento.	Combina activación y contenido.
W_{down}	Devuelve al tamaño del modelo.	Reintegra la señal al residual stream.

SwiGLU y otras variantes de GLU se usan porque mejoran el bloque feed-forward frente a activaciones más simples en Transformers.¹⁷ Esta es una buena vacuna contra una frase peligrosa: “la atención lo hace todo”. No. La atención enruta información entre posiciones; el MLP transforma cada posición; y ambas piezas se alternan capa a capa.

Fase 4: logits, softmax y política de decoding

Cuando el modelo termina de procesar el contexto, todavía no ha escrito texto. Produce logits: una puntuación por token candidato del vocabulario. Después convertimos esas puntuaciones en probabilidades.

$$P(t_i \mid \text{contexto}) = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

SÍMBOLO	QUÉ SIGNIFICA	DECISIÓN PRÁCTICA
z_i	Logit del token candidato i .	Preferencia bruta antes de normalizar.
T	Temperatura.	Baja para JSON estricto; más alta para ideación.
j	Todos los tokens candidatos considerados.	En vocabularios reales pueden ser cientos de miles.
$P(t_i \mid \text{contexto})$	Probabilidad del token i dado el contexto.	No es verdad: es probabilidad de continuación.

Inteligencia artificial para gente curiosa · 686f6c61 · CC BY 4.0

Después llegan filtros y penalizaciones:

PARÁMETRO	QUÉ CAMBIA	CUÁNDO TOCARLO	QUÉ MEDIR
temperature	Aplana o concentra la distribución.	Bajar en salidas estructuradas; subir en generación creativa.	Entropía, tasa de JSON válido, diversidad.
top_k	Limita a los k tokens más probables.	Evitar candidatos raros sin cerrar demasiado.	Repetición, calidad, diversidad.
top_p	Conserva el conjunto mínimo con probabilidad acumulada p .	Generación abierta con control de cola.	Degeneración, diversidad, factualidad.
min_p	Elimina tokens demasiado pequeños frente al máximo.	Runtimes locales donde quieres cortar ruido extremo.	Tokens extraños, estabilidad.
logit_bias	Empuja o bloquea tokens concretos.	Forzar o evitar vocabulario técnico, formatos o palabras prohibidas.	Cumplimiento de formato y efectos secundarios.
max_tokens	Limita longitud de salida.	Control de coste y truncamiento.	Respuestas cortadas, coste por tarea.
stop	Para al encontrar secuencias concretas.	Evitar texto extra tras JSON o secciones delimitadas.	Tasa de salida limpia.

Aquí conviene repetirlo porque evita mucho humo: **los parámetros de decoding no hacen al modelo saber más**. Cambian cómo se selecciona el siguiente token a partir de lo que el modelo ya ha calculado.

También puede aparecer **speculative decoding**. No cambia la arquitectura mental de “siguiente token”, pero sí la ingeniería de serving: un modelo pequeño o una cabeza rápida propone varios tokens candidatos y el modelo grande verifica cuáles acepta.¹⁸ Si los acepta, ganas velocidad; si no, vuelves al modelo grande. No es una técnica para mejorar calidad: es una técnica para reducir latencia manteniendo la distribución objetivo bajo ciertas condiciones.

PIEZA	QUÉ HACE	CUÁNDO MERECE LA PENA
Modelo draft	Propone tokens baratos.	Cuando sus propuestas suelen coincidir con el modelo grande.
Modelo target	Verifica y corrige.	Cuando no quieres cambiar la calidad del modelo principal.
Ratio de aceptación	Porcentaje de tokens draft aceptados.	Si es bajo, el mecanismo añade complejidad sin acelerar.
Métrica útil	Tokens/s y p95 de latencia.	El objetivo es rendimiento, no “razonamiento mejor”.

Fase 5: prefill, decode y KV cache

La generación tiene dos fases operativas muy distintas:

FASE	QUÉ OCURRE	MÉTRICA TÍPICA	PROBLEMA TÍPICO
Prefill	El modelo procesa todo el contexto inicial.	TTFT, tiempo hasta primer token.	Contextos largos, documentos pegados, RAG excesivo.
Decode	El modelo genera token a token.	tokens/s, tiempo total, coste de salida.	Respuestas largas, batch, GPU saturada.

Durante decode, el modelo no quiere recalcular todo el contexto una y otra vez. Guarda claves y valores de atención ya calculados: la KV cache. Esa caché no es “memoria del usuario” ni “recuerdo semántico”. Es memoria de cálculo para acelerar generación autoregresiva.

Ejemplo de fórmula. Para estimar orden de magnitud de KV cache:

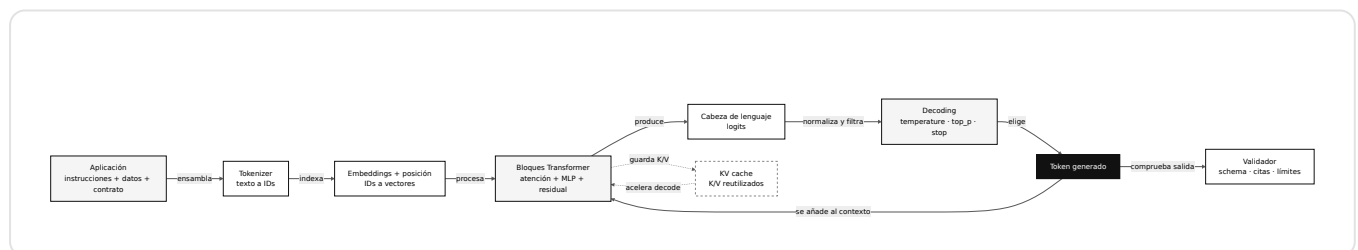
$$M_{KV} \approx 2 \cdot L \cdot B \cdot S \cdot H_{KV} \cdot d_h \cdot \text{bytes}$$

SÍMBOLO	QUÉ SIGNIFICA	EJEMPLO
2	Guardamos claves y valores.	K y V.
L	Número de capas.	32.
B	Batch o secuencias activas.	4 usuarios simultáneos.
S	Contexto reservado o usado.	4096 tokens.
H_{KV}	Cabezas KV.	8 si hay GQA.
d_h	Dimensión por cabeza.	128.
bytes	Bytes por valor.	2 en FP16/BF16.

Esta fórmula no sustituye a medir en vLLM, SGLang, llama.cpp, Ollama o el proveedor que uses. Sirve para no prometer imposibles. Si duplicas contexto o batch, la KV cache crece. Si el modelo “cabe” sin usuarios, quizá no cabe con veinte sesiones largas.

Fase 6: el token sale y el bucle vuelve a empezar

El token elegido se añade a la secuencia. Si hay streaming, el usuario puede verlo enseguida. Si hay salida estructurada, tu aplicación no debería confiar todavía: debe parsear, validar schema, comprobar campos obligatorios y decidir si acepta, reintenta, corrige o escala.



Si algo falla, dónde miro primero

Esta es la parte más útil para ingeniería. La disección no sirve para recitar arquitectura; sirve para depurar.

SÍNTOMA	PRIMERA CAPA QUE MIRARÍA	QUÉ COMPROBARÍA
Respuesta truncada	Contrato de salida	<code>max_tokens</code> , <code>stop</code> , timeout, longitud esperada.
JSON inválido	Decoding y contrato	Temperatura, salida estructurada, schema, texto extra.
Tarda mucho en empezar	Prefill	Tokens de entrada, documentos pegados, RAG demasiado amplio.
Empieza rápido pero tarda mucho en terminar	Decode	Tokens de salida, tokens/s, batch, GPU, streaming.
Se queda sin memoria	Runtime	Pesos + KV cache + batch + contexto + margen.
Repite frases	Sampling	<code>repeat_penalty</code> , <code>frequency_penalty</code> , temperatura, prompt.
Responde sin citar	Contexto y validación	Retrieval, contrato de salida, groundedness, abstención.
Parece seguro pero se equivoca	Evaluación	Dataset propio, slices, referencias, revisión humana.
Cambia demasiado entre ejecuciones	Reproducibilidad	<code>seed</code> , temperatura, modelo exacto, proveedor, cache, herramientas.
Usa mal una herramienta	Agente/tooling	Schema, permisos, argumentos, trazas, validadores.

Si falla, ¿dónde miro? Cada síntoma apunta a una capa

La salida es el final de la cadena, no donde empieza la explicación.

Formato y evidencia

Respuesta truncada

→ contrato de salida
max_tokens, stop, timeout

JSON inválido

→ decoding y schema
temperatura, salida estructurada

Responde sin citar

→ retrieval y contrato
groundedness, abstención

Velocidad y memoria

Tarda en empezar

→ prefill
tokens de entrada, RAG amplio

Tarda en terminar

→ decode
longitud de salida, tokens/s

Se queda sin memoria

→ pesos + KV cache
batch, contexto, margen

Estabilidad

Repite frases

→ sampling
repeat/frequency penalty

Seguro pero se equivoca

→ evaluación
dataset propio, slices, revisión

Cambia entre ejecuciones

→ reproducibilidad
seed, versión, proveedor

IA para gente curiosa / Facsímil 03 / Capítulo 08 / 686f6c61

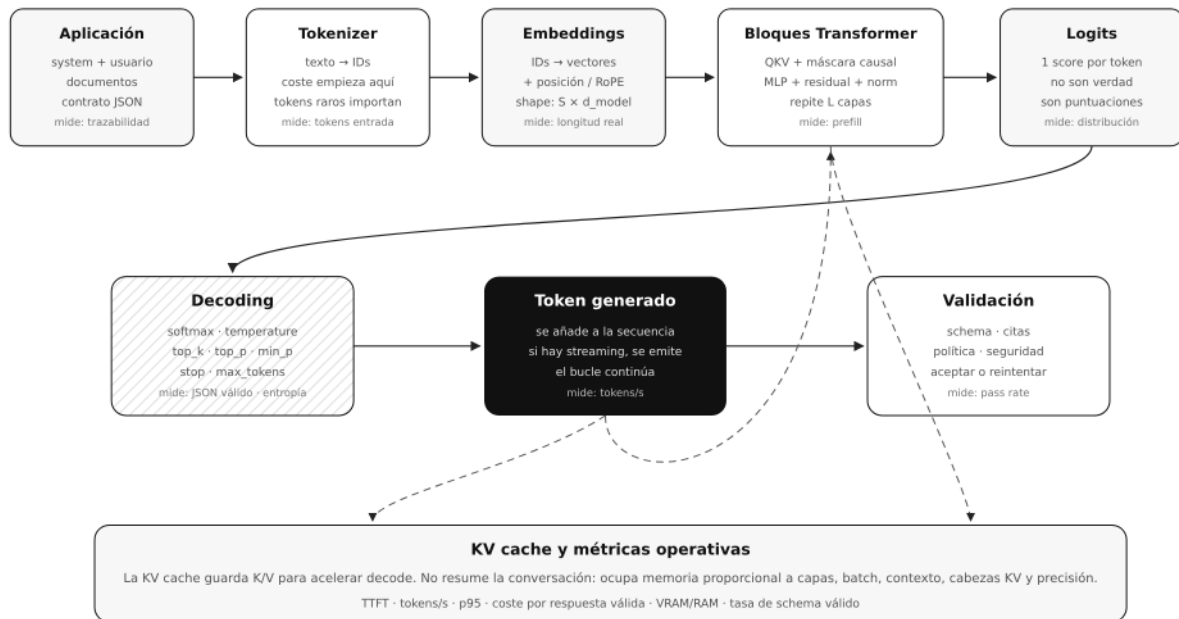
Una frase que debería quedarse: **la salida visible es el final de una cadena, no el lugar donde empieza la explicación**. Si el sistema falla, no preguntes solo “qué ha dicho el modelo”; pregunta qué texto entró, cuántos tokens ocupó, qué contexto recibió, qué logits se filtraron, qué parámetros estaban activos, qué memoria se reservó y qué validador aceptó la salida.

Corte anatómico de una generación

El siguiente esquema resume la disección. No pretende mostrar todos los detalles de un modelo real; pretende poner en una misma mesa las piezas que más se tocan cuando se construye una aplicación: entrada, tensores, bloques, logits, decoding, KV cache, salida y métricas.

Disección de una generación autoregresiva

No seguimos una marca: seguimos el camino técnico que cualquier LLM debe recorrer para producir el siguiente token.



IA para gente curiosa / Facsimil 03 / Capítulo 08 / 686f6c61

Qué debería llevarse un ingeniero

Después de esta disección, un ingeniero no debería decir “el modelo ha fallado” como si fuera una explicación suficiente. Debería poder formular hipótesis:

- si falla el formato, miro contrato, decoding y validador;
- si falla la evidencia, miro retrieval, contexto y citas;
- si falla la latencia inicial, miro prefill y tokens de entrada;
- si falla la latencia total, miro decode y longitud de salida;
- si falla memoria, miro pesos, KV cache, batch, contexto y precisión;
- si falla estabilidad, miro parámetros, versión de modelo, seed, proveedor y evaluación repetida.

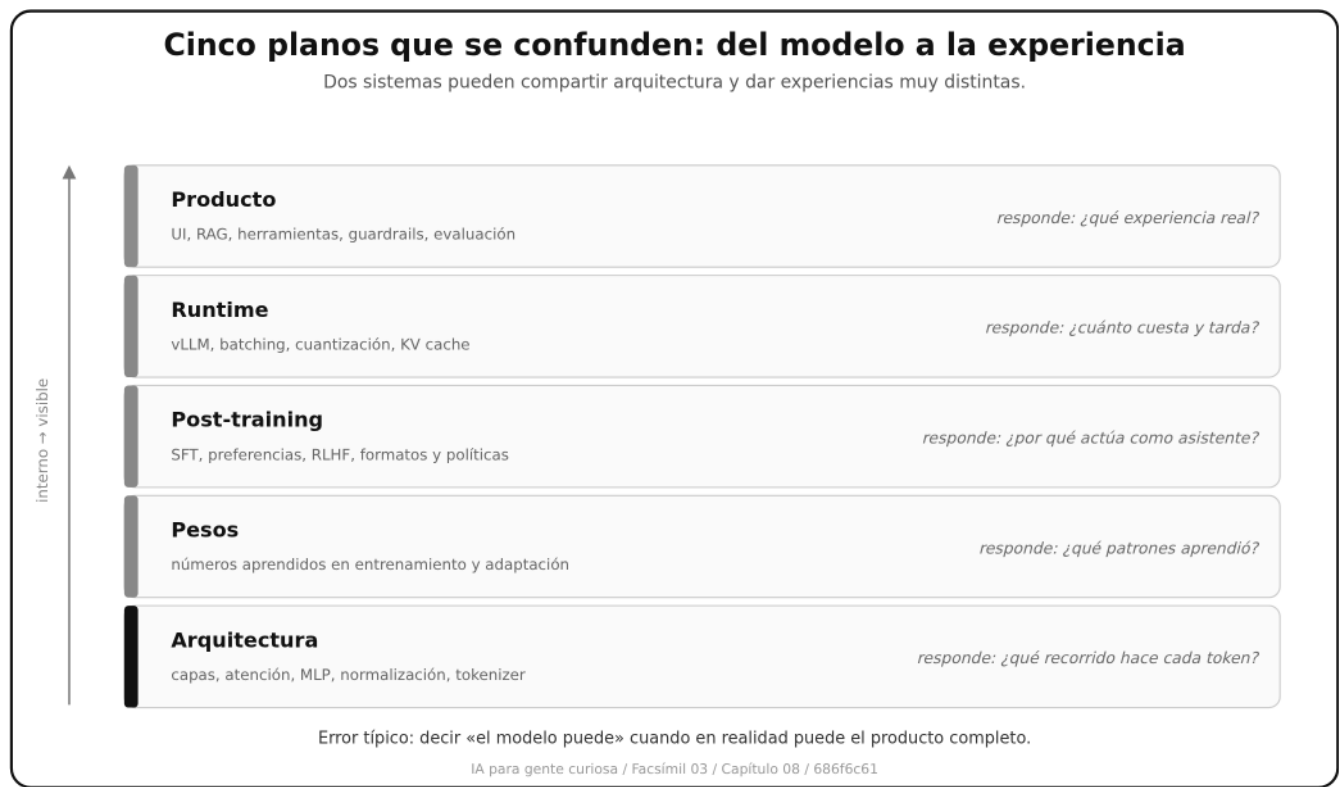
Esa mirada es la diferencia entre usar LLMs como una caja negra y trabajar con ellos como sistemas de ingeniería.

Arquitectura, pesos, post-training y producto no son lo mismo

Una de las ideas más útiles del artículo de Kato es separar piezas que en conversación se mezclan demasiado. Dos modelos pueden compartir “familia Transformer” y comportarse de forma muy distinta; dos despliegues pueden servir pesos parecidos y dar experiencias

distintas; una API puede esconder optimizaciones que no están en el paper del modelo.

PLANO	QUÉ ES	QUÉ PREGUNTA RESPONDE	ERROR TÍPICO
Arquitectura	Forma del modelo: capas, atención, MLP, normalización, tokenizer, contexto, MoE o no MoE.	¿Qué recorrido computacional hace cada token?	Creer que “Transformer” ya explica el comportamiento final.
Pesos	Números aprendidos durante entrenamiento y adaptación.	¿Qué patrones ha aprendido el modelo?	Llamar abierto a un modelo solo porque hay una demo pública.
Post-training	SFT, preferencias, RLHF/RLAIF/RLVR, herramientas, formatos y políticas.	¿Por qué responde como asistente y no como modelo base?	Atribuir al pretraining cosas que vienen de alineamiento o instrucciones.
Runtime	vLLM, SGLang, llama.cpp, servidor propietario, batching, cache, cuantización.	¿Cuánto cuesta, cuánto tarda y cuántos usuarios aguanta?	Comparar modelos sin comparar configuración de serving.
Producto	UI, memoria externa, RAG, herramientas, guardrails, evaluación, permisos.	¿Qué experiencia real tiene la persona?	Decir “el modelo puede” cuando realmente puede el sistema completo.



El mapa completo del facsímil

Facsímil 03: arquitecturas y modelos

La ruta completa: representar texto, transformar contexto, elegir arquitectura, adaptar y servir.

```
graph LR; 01[01 · Objeto LLM  
parámetros · contexto · escala  
qué sabe el modelo  
y qué le damos ahora] --> 02[02 · Transformer  
tokens · embeddings · capas  
texto convertido  
en tensores]; 02 --> 03[03 · Atención  
Q · K · V · softmax  
cada token decide  
qué contexto usar]; 03 --> 04[04 · Decisión  
MLP · logits · sampling  
de vector final  
a token elegido]; 04 --> 05[05 · Familias modernas  
decoder-only · encoder · MoE  
multimodal · razonamiento  
compromisos de arquitectura]; 05 --> 06[06 · Adaptar y comprimir  
fine-tuning · LoRA · QLoRA  
destilación · cuantización  
licencias y modelos abiertos]; 06 --> 07[07 · Servir el modelo  
prefill · decode · KV cache  
scheduler · hardware · edge  
latencia, memoria y coste]; 05 -.-> 08[08 · Criterio de elección  
No preguntes solo "qué modelo es mejor".  
Pregunta: tarea, datos, contexto, coste, latencia, licencia, evaluación y mantenimiento.]; 06 -.-> 08; 07 -.-> 08; 08 -.-> 09[09 · Preparar el modelo  
herramientas · RAG · API]; 09 -.-> 10[10 · Servir el modelo  
API · herramientas · RAG · API];
```

01 · Objeto LLM
parámetros · contexto · escala
qué sabe el modelo
y qué le damos ahora

02 · Transformer
tokens · embeddings · capas
texto convertido
en tensores

03 · Atención
Q · K · V · softmax
cada token decide
qué contexto usar

04 · Decisión
MLP · logits · sampling
de vector final
a token elegido

05 · Familias modernas
decoder-only · encoder · MoE
multimodal · razonamiento
compromisos de arquitectura

06 · Adaptar y comprimir
fine-tuning · LoRA · QLoRA
destilación · cuantización
licencias y modelos abiertos

07 · Servir el modelo
prefill · decode · KV cache
scheduler · hardware · edge
latencia, memoria y coste

08 · Criterio de elección
No preguntes solo "qué modelo es mejor". Pregunta: tarea, datos, contexto, coste, latencia, licencia, evaluación y mantenimiento.

09 · Preparar el modelo
herramientas · RAG · API

10 · Servir el modelo
API · herramientas · RAG · API

viene del facsímil 1: tokens, redes y métricas

prepara el facsímil 4: herramientas, RAG y API

IA para gente curiosa / Facsímil 03 / Capítulo 08 / 686f6c61

En el día a día

Inteligencia artificial para gente curiosa y b6b6b61 cc-b1-4.0

En producto, te ayuda a no elegir por fama. Un modelo enorme puede ser peor decisión si tu caso necesita baja latencia, ejecución local o una licencia concreta. Un modelo más pequeño puede ser suficiente si el dominio está bien acotado, hay recuperación externa y la evaluación es clara.

En ingeniería, te ayuda a separar problemas. Si la salida es mala, puede ser arquitectura, prompt, datos, decoding, herramienta, evaluación o expectativa. Si la latencia es mala, puede ser prefill, decode, KV cache, batch, hardware o red. Poner nombre a la pieza ahorra semanas de confusión.

En aprendizaje, te da una brújula: ya no estás leyendo “modelos” como si fueran marcas. Estás leyendo decisiones de diseño.

Por qué debería importarte

Porque el modelo que eliges condiciona todo lo demás: qué datos necesitas, cómo integras herramientas, cuánto cuesta servir, qué privacidad puedes prometer, qué evals debes construir y qué experiencia tendrá la persona que lo use.

También porque la mayoría de errores caros no nacen de no saber una sigla. Nacen de mezclar planos: pedirle a fine-tuning que actualice conocimiento cambiante, creer que cuantizar no afecta calidad, evaluar solo una demo, ignorar la KV cache o llamar “open source” a cualquier descarga de pesos.

Dónde solía tropezar yo

Estos tropiezos son útiles porque suenan razonables. Precisamente por eso conviene nombrarlos.

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Comparar modelos solo por tamaño	Los parámetros no dicen latencia, contexto útil, licencia, coste ni calidad en tu tarea.	Comparar con una tabla de criterios y pruebas propias.
Creer que Transformer equivale a chat	Transformer es arquitectura; el chat es una interfaz y un post-training.	Separar modelo base, modelo instruct y producto.
Meter conocimiento cambiante en fine-tuning	Ajustar pesos no es buena forma de actualizar datos vivos cada semana.	Usar recuperación o herramientas cuando el dato cambia.
Confundir cuantización con pérdida gratis	Menos bits pueden cambiar calidad, estabilidad o compatibilidad.	Medir antes/después con casos reales.
Ignorar serving	Una demo local no revela colas, KV cache, p95 ni coste por usuario.	Probar concurrencia, contexto largo y respuestas largas.
Depurar solo mirando la respuesta final	La salida puede fallar por tokenización, contexto, decoding, runtime, retrieval, schema o validador.	Diseccionar la llamada: entrada, tokens, logits, parámetros, KV cache y validación.
Recapitular como lista de siglas	Saber decir QKV, MoE o LoRA no significa entender el sistema.	Reexplicar cada pieza con entrada, salida, coste y ejemplo.

Vamos a programarlo

Ahora que hemos mirado una llamada por dentro, toca tomar una decisión de arquitectura. Esta segunda práctica convierte el criterio del facsímil en una matriz ponderada.

La práctica real está en **kit descargable**. No decide por ti, pero obliga a escribir qué pesa más: calidad, latencia, coste, privacidad, mantenimiento, adaptación y contexto externo.

ARCHIVO	QUÉ CONTIENE
<code>Makefile</code>	Atajos para ejecutar, probar y limpiar el kit.
<code>requirements.txt</code>	Declara que el ejercicio usa solo la biblioteca estándar de Python.
<code>data/model_strategy_case.json</code>	Opciones, criterios y pesos.
<code>contracts/model_strategy_policy.json</code>	Mínimos de privacidad/contexto y opción esperada.
<code>ops/score_model_strategy.py</code>	Scoring ponderado y razones de decisión.
<code>tests/test_lab_contract.py</code>	Tests que protegen las ideas del capítulo.
<code>templates/entrega.md</code>	Plantilla de entrega del alumno.
<code>output/model_strategy_report.json</code>	Puntuaciones estructuradas.
<code>output/model_strategy_decision.md</code>	Memo de decisión.

Ejecuta:

```
# Descomprime el ZIP del capítulo y ejecuta estos comandos dentro de esa carpeta
make run
cat output/model_strategy_decision.md
```

Como gate:

```
make test
```

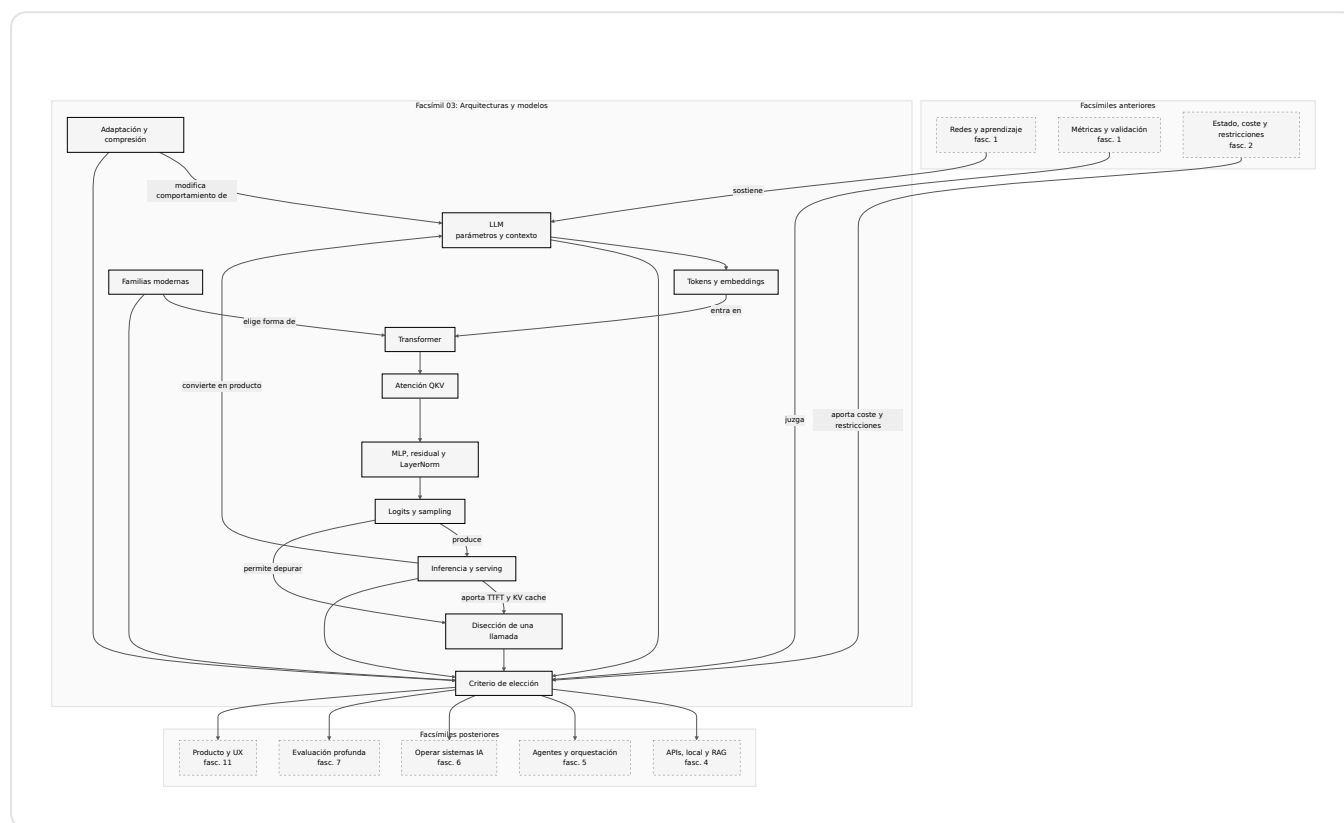
Qué deberías ver. Con el caso de partida, el memo declara un ganador (`api_con_rag`) y una tabla de puntuaciones muy ajustada: `api_con_rag` 3,75, `modelo_abierto_gpu` 3,65, `api_potente` 3,6 y `modelo_local_cuantizado` 3,5. La diferencia pequeña es el mensaje: la matriz no decide sola, obliga a escribir qué pesa más y deja una decisión revisable.

Qué entregaría un alumno. El Markdown generado, una opción nueva, pesos ajustados a su caso y una decisión defendible. Un caso más completo extendería esta misma decisión un paso más allá.

Cómo encaja todo

Este mapa no es un índice bonito: es la lectura operativa del facsímil. La primera fila responde a cómo un texto entra en un modelo y termina como token elegido. La segunda fila responde a qué decisiones aparecen cuando ese modelo deja de ser un dibujo y se convierte en una pieza que hay que adaptar, servir y mantener.

También sirve de puente. Lo aprendido en el facsímil 1 explica por qué existen pérdida, métricas y aprendizaje; el facsímil 2 enseña a pensar en estados, coste y restricciones; y el facsímil 4 tomará estas piezas para construir herramientas, RAG, APIs y sistemas locales.



Vocabulario aprendido

Antes de cerrar, conviene comprobar que estas palabras ya no son ruido. No hace falta recitarlas perfectas; hace falta poder usarlas en una conversación técnica sin perder el hilo.

TÉRMINO	DEFINICIÓN
LLM	Modelo de lenguaje grande entrenado para predecir tokens desde contexto.
Parámetro	Número aprendido durante entrenamiento.
Contexto	Tokens disponibles en una petición concreta.
Embedding	Vector que representa un token o fragmento de texto.
Transformer	Arquitectura basada en atención y bloques paralelizables.
Atención	Mecanismo que mezcla información entre posiciones.
Máscara causal	Restricción que evita mirar tokens futuros durante generación.
MLP	Subred que transforma cada posición dentro del bloque.
Residual	Conexión que conserva información entre capas.
LayerNorm	Normalización que estabiliza activaciones.
Logit	Puntuación previa a convertir candidatos en probabilidades.
Sampling	Elección del siguiente token desde una distribución.
MoE	Arquitectura con expertos y router que activa solo una parte del modelo.
Multimodalidad	Capacidad de trabajar con texto, imagen, audio u otros datos convertidos a representaciones compatibles.
LoRA	Técnica de adaptación con matrices de bajo rango.
QLoRA	LoRA sobre modelo cuantizado para reducir memoria.
Destilación	Entrenar un modelo pequeño para imitar uno mayor.
Cuantización	Representar números con menos bits para ahorrar memoria y coste.
KV cache	Memoria de claves y valores usada durante inferencia.
Serving	Sistema que ejecuta modelos para usuarios reales.
Dissección de LLM	Recorrido técnico de una petición desde texto y tokens hasta logits, sampling, KV cache y salida validada.
Prefill	Fase en la que el modelo procesa todo el contexto inicial antes de emitir el primer token.

TÉRMINO	DEFINICIÓN
Decode	Fase autoregresiva en la que el modelo genera un token, lo añade al contexto y repite.
TTFT	Tiempo hasta el primer token visible para el usuario.
Entropía de salida	Medida de dispersión de la distribución de candidatos; ayuda a comparar estabilidad y diversidad.
RoPE	Codificación posicional rotatoria que incorpora orden en consultas y claves de atención.
GQA	Atención agrupada donde varias cabezas de consulta comparten menos cabezas de claves y valores para reducir KV cache.
RMSNorm	Normalización basada en raíz cuadrática media usada en muchos decoders modernos.
Residual stream	Flujo principal de activaciones al que cada subcapa suma correcciones.
Speculative decoding	Técnica de inferencia donde un modelo rápido propone tokens y el modelo principal los verifica.

Antes de pasar página

Responde sin mirar los capítulos. Si dudas, el enlace te dice dónde volver.

- ☐ ¿Puedo explicar qué diferencia hay entre parámetros, contexto y salida? (Vuelve al [capítulo 01.](#))
- ☐ ¿Sé recorrer el camino texto → tokens → embeddings → Transformer → logits? (Vuelve al [capítulo 02.](#))
- ☐ ¿Puedo explicar Q, K, V y softmax sin decir solo “atención”? (Vuelve al [capítulo 03.](#))
- ☐ ¿Entiendo cómo temperatura, top-k y top-p cambian una respuesta? (Vuelve al [capítulo 04.](#))
- ☐ ¿Puedo distinguir encoder-only, decoder-only, encoder-decoder y MoE? (Vuelve al [capítulo 05.](#))
- ☐ ¿Sé cuándo elegir RAG, fine-tuning, LoRA, QLoRA o cuantización? (Vuelve al [capítulo 06.](#))
- ☐ ¿Puedo separar prefill, decode, TTFT, tokens/s y KV cache? (Vuelve al [capítulo 07.](#))
- ☐ ¿Puedo diseccionar una petición completa y decir dónde miraría si falla formato, latencia, memoria o evidencia? (Si no, vuelve a «Diseccionando un LLM: de una petición al siguiente token».)

- ☐ ¿Puedo justificar una elección de modelo con criterios, no con una marca? (Si no, vuelve a «El mapa completo del facsímil».)
- ☐ ¿Sé qué parámetro tocar para ver qué métrica se mueve al diseccionar un LLM? (Si no, vuelve a «Diseccionando un LLM: de una petición al siguiente token».)
- ☐ ¿Sé construir una matriz de decisión y ver cómo cambia el ganador al ajustar los pesos? (Si no, vuelve a «El mapa completo del facsímil».)

En resumen

La versión corta del facsímil no es “los LLMs son Transformers”. Es más concreta: un modelo es una arquitectura entrenada, adaptada y servida bajo restricciones.

IDEA FUERZA	DETALLE
Un LLM no es una API.	Es una arquitectura con parámetros, contexto, tokens, atención y sampling.
La atención no es una caja negra.	Q, K, V y softmax explican cómo una posición usa otras posiciones.
Generar texto es decidir tokens.	Logits, temperatura, top-k y top-p controlan el abanico de salida.
La arquitectura es compromiso.	MoE, multimodalidad, contexto largo y modelos híbridos cambian coste y capacidades.
Adaptar no siempre es entrenar.	RAG, fine-tuning, LoRA, QLoRA, destilación y cuantización resuelven problemas distintos.
Servir un modelo es ingeniería.	Prefill, decode, KV cache, hardware y p95 importan tanto como la calidad media.
Diseccionar una llamada ayuda a depurar.	Formato, latencia, memoria, evidencia y coste suelen fallar en capas distintas.
Elegir modelo es explicitar restricciones.	Tarea, datos, coste, privacidad, latencia, licencia, evaluación y mantenimiento.

Recursos para seguir: leer, construir y experimentar

Ya sabes qué hay dentro de un modelo de lenguaje: tokens, atención, capas, sampling y la familia de arquitecturas. Para que no se quede en teoría, aquí tienes recursos reales.

Para experimentar sin código. Lo has hecho en las cajas «Pruébalo en 5 minutos»: el visualizador de Brendan Bycroft (bbycroft.net/llm) para recorrer un transformer real en 3D y ver la atención por dentro; los playgrounds (platform.openai.com , aistudio.google.com , console.anthropic.com) para jugar con la temperatura y el top-p; Hugging Face (huggingface.co) para comparar modelos abiertos y leer sus fichas; y Ollama (ollama.com) para correr un modelo cuantizado en tu propio portátil.

Para construir. La librería de referencia para trabajar con estos modelos es Transformers de Hugging Face, sobre PyTorch; para inferencia local, Ollama y llama.cpp; para servir a escala, proyectos como vLLM. Casi todo es código abierto.

Para leer. El artículo fundacional es *Attention is All You Need* (Vaswani y otros, 2017), y para entenderlo paso a paso, *The Illustrated Transformer* de Jay Alammar es una de las mejores explicaciones visuales que existen. Cada capítulo cierra con su «Para saber más». Los cuadernos del facsímil son el puente entre el diagrama y el código.

Para saber más

Ainslie, J. y otros (2023). GQA: Training generalized multi-query Transformer models from multi-head checkpoints. *Proceedings of EMNLP*. <https://arxiv.org/abs/2305.13245>

Brown, T. B. y otros (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems 33*, 1877-1901. <https://papers.nips.cc/paper/2020/hash/1457c0d6bfc4967418bfb8ac142f64a-Abstract.html>

Dao, T. y otros (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems 35*. <https://arxiv.org/abs/2205.14135>

Dettmers, T. y otros (2023). QLoRA: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems 36*. <https://arxiv.org/abs/2305.14314>

Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, 4171-4186. <https://doi.org/10.18653/v1/N19-1423>

Fedus, W., Zoph, B. y Shazeer, N. (2022). Switch Transformers: scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120), 1-39. <https://www.jmlr.org/papers/v23/21-0998.html>

Hu, E. J. y otros (2022). LoRA: Low-rank adaptation of large language models. *International Conference on Learning Representations*. <https://arxiv.org/abs/2106.09685>

Kwon, W. y otros (2023). Efficient memory management for large language model serving with PagedAttention. *Proceedings of SOSP*. <https://arxiv.org/abs/2309.06180>

Kato. (2026). *How LLMs actually work*. <https://www.0xkato.xyz/how-llms-actually-work/>

Leviathan, Y., Kalman, M. y Matias, Y. (2023). Fast Inference from Transformers via Speculative Decoding. *Proceedings of the 40th International Conference on Machine Learning*. <https://arxiv.org/abs/2211.17192>

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F. y Liang, P. (2024). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12, 157-173. https://doi.org/10.1162/tacl_a_00638

Raffel, C. y otros (2020). Exploring the limits of transfer learning with a unified text-to-text Transformer. *Journal of Machine Learning Research*, 21(140), 1-67. <https://www.jmlr.org/papers/v21/20-074.html>

Shazeer, N. (2020). GLU Variants Improve Transformer. <https://arxiv.org/abs/2002.05202>

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B. y Liu, Y. (2024). RoFormer: Enhanced Transformer with Rotary Position Embedding. *Neurocomputing*, 568, 127063. <https://doi.org/10.1016/j.neucom.2023.127063>

Van Rijn, R. (2026). *The Anatomy of an LLM*. <https://www.royvanrijn.com/anatomy-of-an-llm/>

Vaswani, A. y otros (2017). Attention is all you need. *Advances in Neural Information Processing Systems 30*, 5998-6008. <https://papers.nips.cc/paper/7181-attention-is-all-you-need>

Zhang, B. y Sennrich, R. (2019). Root Mean Square Layer Normalization. *Advances in Neural Information Processing Systems 32*. <https://arxiv.org/abs/1910.07467>

Cuadernos para practicar

Has razonado sobre el modelo a lo largo del facsímil; estos cuadernos te dejan abrirlo y tocarlo. Son *notebooks* que se abren en Google Colab —gratis, en el navegador— o te puedes descargar. Cada uno, explicado paso a paso, con salidas reales, y anclado al capítulo del que sale.

Atención a mano: Q, K, V y la máscara causal

Qué prácticas: calcular con NumPy el mecanismo de atención que mueve a todos los LLM. **Dónde encaja:** capítulos 2 y 3 (el Transformer por dentro; *queries*, *keys*, *values* y máscara causal). **Qué necesitas:** un navegador. Corre en CPU; sin nada más.

Coges una frase de cinco palabras y calculas, a mano, cómo cada una «mira» a las demás: el parecido entre su pregunta (*query*) y las claves (*keys*), normalizado con *softmax*, da un mapa de atención que imprimes y dibujas. Luego aplicas la **máscara**

causal —poner menos infinito en el futuro— y ves el mapa volverse triangular: ninguna palabra puede mirar a las que vienen después, que es lo que distingue a un modelo que genera de uno que solo entiende. Terminas mezclando los valores (*values*) según esos pesos, que es justo lo que sale de la capa.

«Atención» deja de ser un eslogan: es un reparto de pesos que puedes ver casilla a casilla.

► [Abrir en Google Colab](#)

↓ [Descargar el cuaderno \(.ipynb\)](#)

Cómo elige palabras un LLM: temperatura, top-k y top-p

Qué prácticas: controlar la creatividad de un modelo moldeando su distribución de salida. **Dónde encaja:** capítulo 4 (MLP, residual, *logits* y *sampling*). **Qué necesitas:** un navegador. Corre en CPU; sin nada más.

Te pones en la última capa de un LLM: el modelo no «elige» una palabra, produce una puntuación para cada una del vocabulario y un proceso de muestreo decide. Juegas con los tres botones. La temperatura baja (0.5) agranda las diferencias y el modelo se vuelve seguro y repetitivo; la alta (1.5) las aplanas y aparecen palabras improbables — creatividad y disparates—. El *top-k* se queda con las mejores; el *top-p*, con las que acumulan un porcentaje de probabilidad, adaptándose a si el modelo lo tiene claro o duda. Muestras mil veces con cada estrategia y ves la diferencia entre un loro predecible y un poeta.

No hay un valor «correcto»: depende de si quieres un notario o un poeta. Y ese botón es tuyo, no del modelo.

► [Abrir en Google Colab](#)

↓ [Descargar el cuaderno \(.ipynb\)](#)

Cuantización: cómo cabe un modelo gigante en tu portátil

Qué prácticas: guardar los pesos de un modelo con 8 bits y medir qué ganas y qué pierdes. **Dónde encaja:** capítulos 6 y 7 (modelos abiertos; inferencia optimizada, *edge AI* y hardware). **Qué necesitas:** un navegador. Corre en CPU; sin nada más.

Tomas una matriz de pesos en coma flotante de 32 bits y la guardas en enteros de 8 bits: buscas el peso de mayor magnitud, partes el rango en 256 escalones y redondeas cada peso al más cercano. El trato tiene dos caras y mides las dos: la memoria cae **4**

veces y el error que introduces es minúsculo —alrededor del **1%** al multiplicar por un vector—. Compruebas que con valores atípicos o con menos bits el error se dispara, que es exactamente el filo en el que viven los formatos `q4` de los modelos GGUF.

No es magia: es redondear con cabeza. Y es la razón de que hoy corras modelos potentes en hardware modesto.

► **Abrir en Google Colab**

↓ **Descargar el cuaderno (.ipynb)**

Notas

1. Brown, T. B. y otros (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems 33*, 1877-1901.
<https://papers.nips.cc/paper/2020/hash/1457c0d6bfc4967418bfb8ac142f64a-Abstract.html>. GPT-3 popularizó la escala como variable central para capacidades emergentes de uso general. ↵
2. Vaswani, A. y otros (2017). Attention is all you need. *Advances in Neural Information Processing Systems 30*, 5998-6008. <https://papers.nips.cc/paper/7181-attention-is-all-you-need>. El paper introdujo una arquitectura basada en atención que evitaba recurrencia y facilitaba paralelismo. ↵
3. Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, 4171-4186.
<https://doi.org/10.18653/v1/N19-1423>. ↵
4. Raffel, C. y otros (2020). Exploring the limits of transfer learning with a unified text-to-text Transformer. *Journal of Machine Learning Research*, 21(140), 1-67.
<https://www.jmlr.org/papers/v21/20-074.html>. ↵
5. Fedus, W., Zoph, B. y Shazeer, N. (2022). Switch Transformers: scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120), 1-39. <https://www.jmlr.org/papers/v23/21-0998.html>. ↵
6. Hu, E. J. y otros (2022). LoRA: Low-rank adaptation of large language models. *International Conference on Learning Representations*. <https://arxiv.org/abs/2106.09685>. ↵
7. Dettmers, T. y otros (2023). QLoRA: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems 36*. <https://arxiv.org/abs/2305.14314>. ↵
8. Dao, T. y otros (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems 35*.
<https://arxiv.org/abs/2205.14135>. ↵

9. Kwon, W. y otros (2023). Efficient memory management for large language model serving with PagedAttention. *Proceedings of SOSP*. <https://arxiv.org/abs/2309.06180>. ↵
10. Ainslie, J. y otros (2023). GQA: Training generalized multi-query Transformer models from multi-head checkpoints. *Proceedings of EMNLP*. <https://arxiv.org/abs/2305.13245>. ↵
11. Van Rijn, R. (2026). *The Anatomy of an LLM*. <https://www.royvanrijn.com/anatomy-of-an-llm/>. Consultado el 14 de junio de 2026. El recurso se declara en progreso; aquí se usa como inspiración pedagógica, no como fuente única ni como especificación técnica. ↵
12. Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B. y Liu, Y. (2024). RoFormer: Enhanced Transformer with Rotary Position Embedding. *Neurocomputing*, 568, 127063. <https://doi.org/10.1016/j.neucom.2023.127063> ↵
13. Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F. y Liang, P. (2024). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12, 157-173. <https://doi.org/10.1162/tacl.a.00638> ↵
14. Kato. (2026). *How LLMs actually work*. <https://www.0xkato.xyz/how-llms-actually-work/>. Consultado el 14 de junio de 2026. ↵
15. Ainslie, J. y otros (2023). GQA: Training generalized multi-query Transformer models from multi-head checkpoints. *Proceedings of EMNLP*. <https://arxiv.org/abs/2305.13245> ↵
16. Zhang, B. y Sennrich, R. (2019). Root Mean Square Layer Normalization. *Advances in Neural Information Processing Systems* 32. <https://arxiv.org/abs/1910.07467> ↵
17. Shazeer, N. (2020). GLU Variants Improve Transformer. <https://arxiv.org/abs/2002.05202> ↵
18. Leviathan, Y., Kalman, M. y Matias, Y. (2023). Fast Inference from Transformers via Speculative Decoding. *Proceedings of the 40th International Conference on Machine Learning*. <https://arxiv.org/abs/2211.17192> ↵