Facsímil 03

Arquitecturas y modelos

Capítulo 01

Qué es un LLM: modelo, parámetros y escala

686f6c61 · 2026

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 facsímil 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 facsímil 4, cuando hablemos de APIs y RAG, estaremos construyendo parte del sistema completo. En el facsímil 5, cuando hablemos de agentes, estaremos añadiendo decisiones, herramientas y memoria alrededor del modelo. En el facsímil 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.

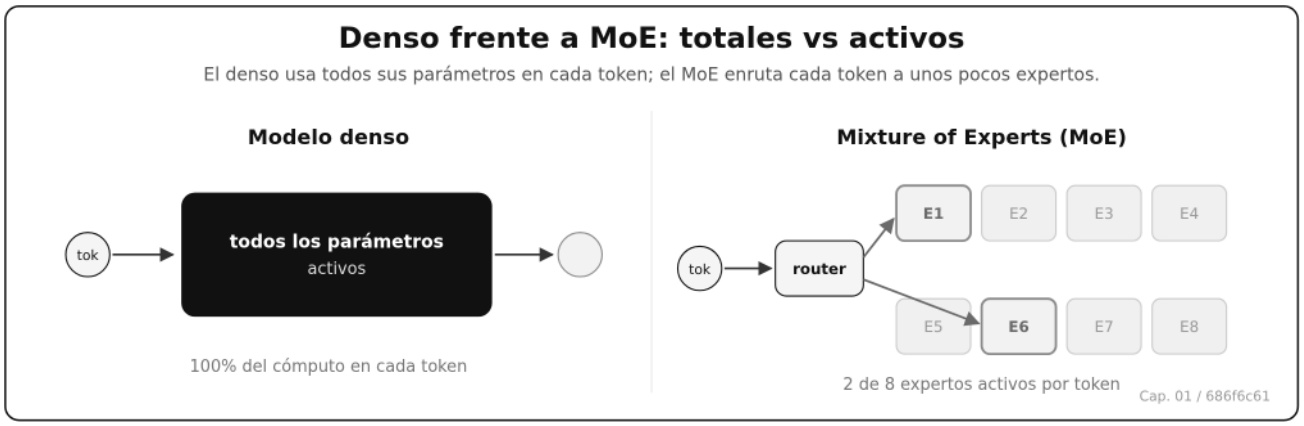
Qué tipos de LLM podrías encontrarte

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:

$$x_1, x_2, \ldots, x_T$$

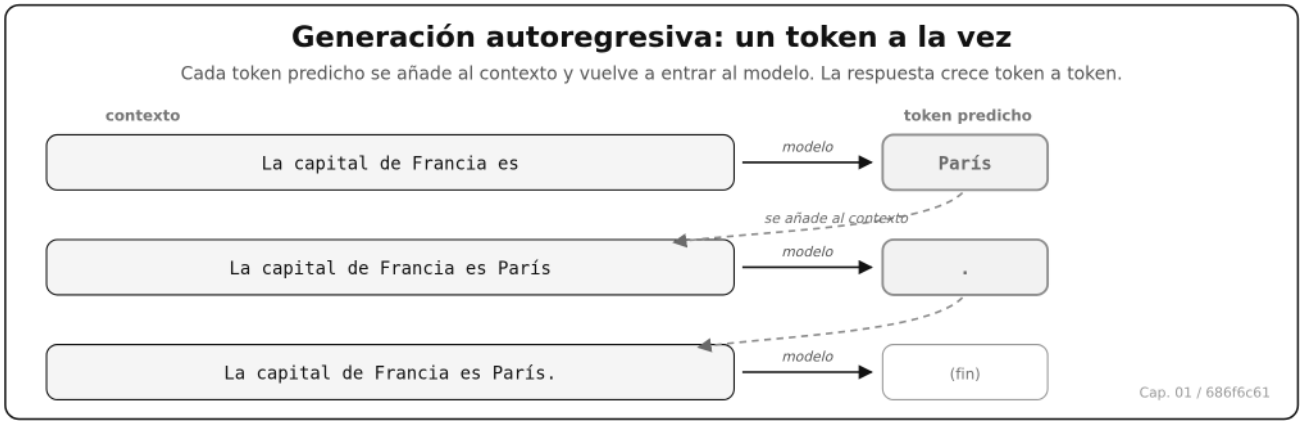
el modelo estima la probabilidad conjunta así:

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

SÍMBOLO	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.
$P(x_t \mid x_{<t}; \theta)$	Probabilidad de un token dado el contexto anterior y los parámetros.	$P(\text{azul} \mid \text{la casa})$.

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**:

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

SÍMBOLO	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:

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

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

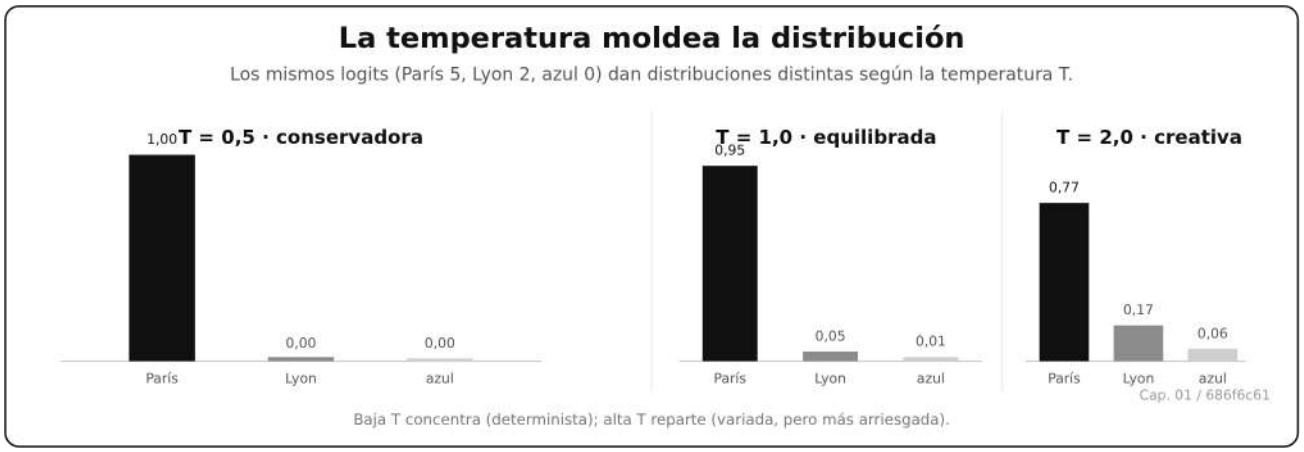
Aplicamos softmax:

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

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

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 aplana (reparte probabilidad, salidas más variadas y creativas, pero también más arriesgadas). Los mismos logits, distinta temperatura:



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

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

SÍMBOLO	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:

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

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:

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

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.

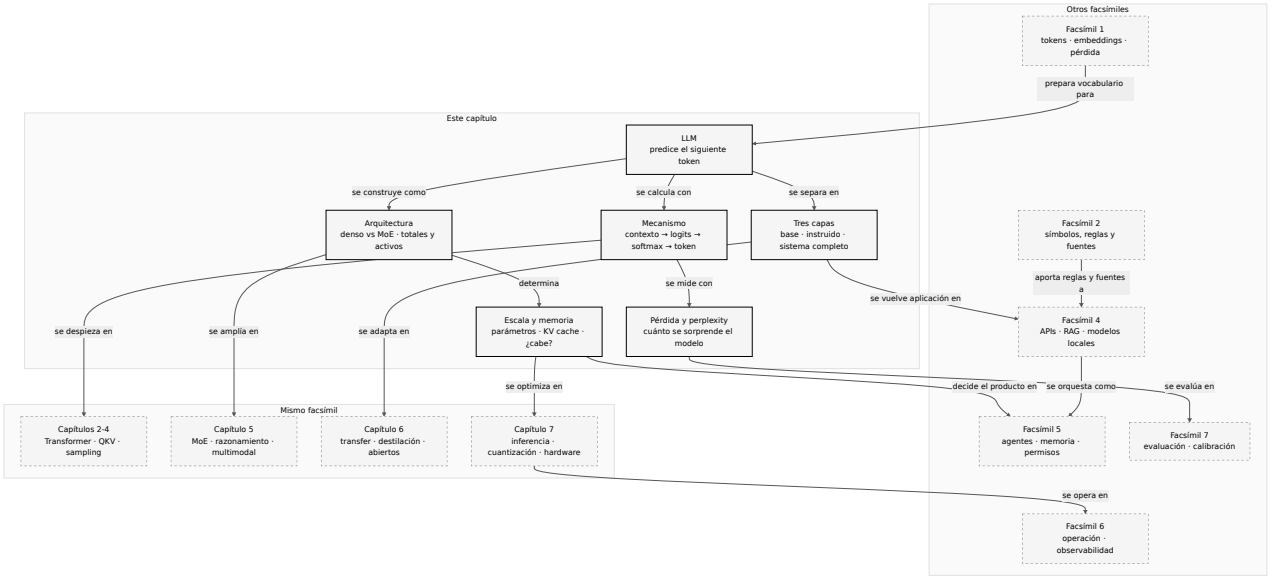
Dónde solía tropezar yo

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 conocimient 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.


Otros facsímiles
Facsímil 1
tokens · embeddings · pérdida
prepara vocabulario para
Este capítulo
LLM
predice el siguiente token
se construye como
se calcula con
se separa en
Arquitectura
denso vs MoE · totales y activos
Mecanismo
contexto → logits → softmax → token
Tres capas
base · instruido · sistema completo
Facsímil 2
símbolos, reglas y fuentes
aporta reglas y fuentes a
determina
se mide con
se vuelve aplicación en
se despieza en
se amplía en
se adapta en
Escala y memoria
parámetros · KV cache · ¿cabe?
Pérdida y perplexity
cuánto se sorprende el modelo
Facsímil 4
APIs · RAG · modelos locales
se optimiza en
decide el producto en
se orquesta como
se evalúa en
Mismo facsímil
Capítulos 2-4
Transformer · QKV · sampling
Capítulo 5
MoE · razonamiento · multimodal
Capítulo 6
transfer · destilación · abiertos
Capítulo 7
inferencia · cuantización · hardware
Facsímil 5
agentes · memoria · permisos
Facsímil 7
evaluación · calibración
se opera en
Facsímil 6
operación · observabilidad

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 aplana.
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 $P(x_1, \ldots, x_T) = \prod_t P(x_t \mid x_{<t}; \theta)$ 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 aplana 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/1457c0d6bfcb4967418bfb8ac142f64a-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/1457c0d6bfcb4967418bfb8ac142f64a-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. ↩

.0. 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. ↩

.1. 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. ↩