Facsímil 01

Los cimientos

Coleccionable completo · 12 capítulos

686f6c61 · 2026

Índice

Capítulo 01: Qué es y qué no es la inteligencia artificial

Entrando en el tema

Abres el navegador, escribes «explícame cómo funciona un motor de combustión» y pulsas Intro. En menos de tres segundos tienes delante un texto articulado, con ejemplos, párrafos bien estructurados y un tono didáctico impecable. Tu cerebro, entrenado durante décadas para asociar lenguaje coherente con inteligencia, llega a una conclusión instantánea: aquí dentro hay alguien que sabe de motores.

Es una reacción comprensible, humana y equivocada.

Este capítulo nace exactamente de esa reacción. Vamos a desmontarla pieza a pieza. Porque entender qué es, y sobre todo qué no es, la inteligencia artificial es la primera decisión de ingeniería que tomarás al trabajar con ella. Y posiblemente la más importante.

Qué no es la inteligencia artificial

Empecemos por lo que no es. La lista de malentendidos es larga, pero cinco negaciones bastan para despejar el terreno.

No es magia ni ciencia ficción. No hay una entidad misteriosa dentro de la máquina. Es *software*, ejecutándose en servidores, con electricidad y silicio.[1] Impresionante, sí. Pero explicable, medible y auditable.

No es una mente consciente. No tiene deseos, emociones ni intenciones. No «quiere» nada. Procesa tokens y calcula probabilidades. Punto. Si alguna vez has sentido que el modelo «te entiende», has experimentado el efecto Eliza: nuestro cerebro antropomorfiza cualquier cosa que produzca lenguaje coherente.[2]

No es un buscador glorificado. Un buscador recupera páginas o documentos que ya existen. Un LLM (*Large Language Model*, modelo grande de lenguaje) genera texto a partir de patrones aprendidos durante el entrenamiento y no consulta fuentes por defecto. Son mecanismos radicalmente distintos.

No «piensa» como un humano. Puede producir pasos que parecen razonamiento, pero su mecanismo base es predecir el siguiente token a partir de patrones estadísticos. No hay deliberación, no hay duda, no hay «déjame pensarlo». Hay una multiplicación de matrices a escala industrial.

No es infalible ni objetiva. Puede alucinar (generar información falsa con total confianza), heredar sesgos de los datos de entrenamiento y carecer de introspección fiable sobre sus propios errores. No te dirá «no sé» a menos que se lo hayas enseñado explícitamente.

Qué sí es la inteligencia artificial

Ahora la definición positiva. Si la IA no es conciencia ni un oráculo, ¿qué es?

Modelos matemáticos masivos. Redes neuronales con miles de millones o billones de parámetros (números) ajustados para reconocer patrones en texto, imágenes, código y audio.

Reconocimiento de patrones a escala. Lo que una persona tarda horas en analizar, el modelo lo procesa en segundos. No «entiende» en el sentido humano de saber qué está diciendo, contrastarlo con el mundo y hacerse responsable de ello: aprende regularidades estadísticas y las usa para producir una salida plausible. Esa diferencia parece filosófica, pero en ingeniería es muy concreta: una salida plausible necesita verificación.

Predicción estadística del siguiente token. Dada una secuencia de tokens, ¿cuál es el más probable a continuación? Esta operación, repetida miles de veces, produce párrafos coherentes, código funcional y respuestas que parecen razonadas. Es el mecanismo central de todo LLM moderno.

Un amplificador de capacidades. Si sabes programar, te hace más rápido. Si no sabes, te da una falsa sensación de que funciona… hasta que deja de hacerlo. La IA no sustituye el criterio humano: lo amplifica. Si le das contexto claro y restricciones precisas, el resultado es impresionante. Si le das ambigüedad, el resultado es impredecible.

Para situarnos en el mapa, estos son los hitos que nos han traído hasta aquí. La parte histórica es estable; la parte de herramientas actuales no lo es. **Fecha de corte: 10 de junio de 2026.** Ese día, las fuentes consultadas para la fila de sistemas con herramientas y agentes fueron la documentación pública de OpenAI Agents SDK, Claude Agent SDK, Google ADK y Model Context Protocol.[3]

FECHA	HITO	POR QUÉ IMPORTA
1950	Test de Turing	Turing propone un criterio para evaluar si una máquina exhibe comportamiento inteligente. No es un test técnico, pero inaugura la pregunta filosófica que aún nos hacemos.[4]
1956	Conferencia de Dartmouth	McCarthy, Minsky, Shannon y Rochester organizan un taller de verano en Dartmouth College. Acuñan el término «inteligencia artificial».[5] Es el nacimiento del campo como disciplina.
1986	Retropropagación	Rumelhart, Hinton y Williams publican el algoritmo que permite entrenar redes de más de dos capas. Sin esto no existiría el *deep learning*.[6] Lo veremos en detalle en el capítulo 6.
2012	AlexNet gana ImageNet	Una red neuronal convolucional arrasa en el concurso de reconocimiento de imágenes. Comienza la era del *deep learning* moderno.[7]
2017	Transformer	El artículo «Attention Is All You Need» presenta una arquitectura que procesa todas las palabras simultáneamente en vez de secuencialmente. Es la base de todo lo que vino después.[8] Lo desarrollaremos con todo detalle en el facsímil 3.
2020	GPT-3	175 000 millones de parámetros. Demostró que escalar funciona: más datos y más parámetros producen más capacidad.[9]
2022	ChatGPT	RLHF (aprendizaje por refuerzo con *feedback* humano) más una interfaz de chat. La IA se hace accesible al público general.
2023	GPT-4	Multimodal y con razonamiento avanzado. Salto cualitativo en capacidades.[10]
2024	Claude 3 / Gemini	Competencia real. Contextos de 200 000 o más tokens. La IA se convierte en herramienta de trabajo diaria.
2025-2026	Sistemas con herramientas y agentes de software	El chat deja de ser la única interfaz. Aparecen SDKs y protocolos que combinan modelo, herramientas, estado, trazas, aprobaciones y ejecución sobre entornos reales. Lo exploraremos a fondo en el facsímil 5.

Cómo funciona por dentro

Este capítulo es conceptual: no vamos a entrenar modelos ni a derivar nada. Aparecerán solo dos expresiones matemáticas, las justas para ver el mecanismo con precisión. Antes de cerrarlo necesitas entender ese mecanismo, porque está debajo de todo lo demás. Cada capítulo futuro (la neurona artificial, la retropropagación, el Transformer, los agentes) se construye sobre esta pieza.

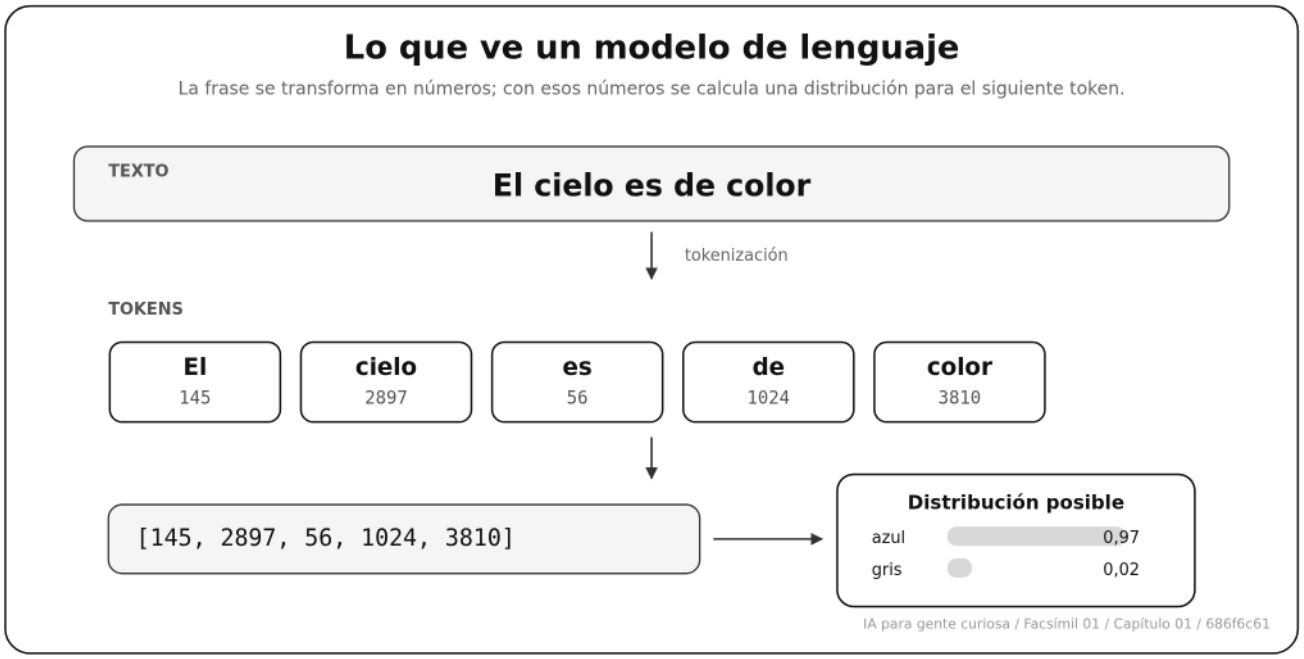
Se llama **predicción del siguiente token** y funciona así.

Imagina que escribes:

> El cielo es de color

El modelo recibe esta secuencia y ejecuta tres pasos:

Paso 1: tokenización. Convierte cada palabra (o fragmento de palabra) en un número que la identifica. «El» podría ser el token 145, «cielo» el 2897, «es» el 56, «de» el 1024 y «color» el 3810. El modelo no ve palabras: ve secuencias de números.



El modelo no ve palabras: ve secuencias de números. En el capítulo 9 exploraremos la tokenización en profundidad y entenderemos por qué «cielo» puede ser un token y «extraordinariamente» pueden ser tres.

EJEMPLO, NO NOTACIÓN OFICIAL.

$$\text{tok}(\text{El cielo es de color}) = [145,\ 2897,\ 56,\ 1024,\ 3810]$$

> No es una fórmula del campo: la tokenización es un algoritmo, y aquí ponemos un ejemplo concreto para verla como una función que recibe un texto y devuelve la lista de identificadores que ese texto tiene en el vocabulario del modelo. El algoritmo real (por ejemplo, *byte-pair encoding*) lo veremos en el capítulo 9.

Paso 2: transformación por pesos aprendidos. Esos números atraviesan las capas de la red neuronal. En cada capa, los miles de millones de parámetros, los pesos aprendidos durante el entrenamiento, transforman la representación. Conviene no llamarlo «memoria» sin matiz: el modelo no abre una ficha interna sobre el cielo ni recupera una frase guardada. Aplica pesos que codifican regularidades observadas durante el entrenamiento y produce una representación nueva de la secuencia.

Paso 3: muestreo. El modelo produce una distribución de probabilidad sobre todos los tokens posibles que podrían seguir. No elige «la correcta». Asigna probabilidades (en este ejemplo cada candidato es un token de una sola pieza):

- `azul` : 97 %
- `gris` : 2 %
- `rojo` : 0,5 %
- `verde` : 0,3 %
- `naranja` : 0,1 %
- ...y una probabilidad minúscula para todos los demás tokens del vocabulario, que son decenas de miles.

¿De dónde salen esos porcentajes? El modelo no produce probabilidades directamente, sino una puntuación sin normalizar para cada token, su *logit*. La función *softmax* convierte ese vector de puntuaciones en una distribución de probabilidad que suma 1:

$$P(t_{n+1} = v \mid t_1, \ldots, t_n) = \frac{e^{z_v}}{\sum_{u \in V} e^{z_u}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$P(t_{n+1} = v \mid t_1, \ldots, t_n)$	Probabilidad de que el siguiente token sea v, dado el contexto anterior	0,97 para `azul`
v	Un token candidato del vocabulario	`azul` , `gris` , `rojo` …
z_v	Logit de v: la puntuación sin normalizar que el modelo asigna al token	más alta cuanto más probable es el token
e^{z_v}	Exponencial del logit. Convierte cualquier puntuación en un número positivo	nunca negativo
V	El vocabulario completo, todos los tokens posibles	decenas de miles
$\sum_{u \in V} e^{z_u}$	Suma de las exponenciales de todos los tokens. Es lo que normaliza el resultado	el total que reparte el 100 %

En palabras: el modelo exponencia la puntuación de cada token, así ninguna probabilidad es negativa, y divide por la suma de todas, así el conjunto suma 1. El token con el logit más alto se lleva la mayor probabilidad.[11]

Luego **muestrea** de esa distribución. Normalmente elegirá `azul`, pero no siempre. Esa pequeña probabilidad de elegir `gris` o `rojo` es lo que hace que el texto generado no sea idéntico en cada ejecución. Es lo que le da variabilidad, creatividad... y también impredecibilidad.

El proceso se repite. El token elegido, digamos `azul`, se añade a la secuencia de entrada:

```
El cielo es de color azul
```

Y el modelo predice el siguiente:

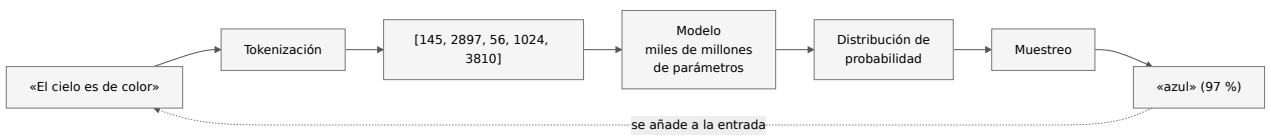
```
El cielo es de color azul ,
```

Y el siguiente:

El cielo es de color azul, **aunque**

Y así, token a token, durante cientos o miles de iteraciones, hasta que el modelo genera un token especial que significa «terminé». Lo que empezó como una pregunta de cinco palabras se convierte en una respuesta de tres párrafos.

No hay comprensión humana, intención ni conciencia. Hay una operación estadística, predecir el siguiente elemento probable de una secuencia, ejecutada a una escala que produce resultados que *parecen* inteligentes. La precisión importa: no estamos diciendo que el sistema sea inútil; estamos diciendo que su utilidad no lo convierte en sujeto, oráculo ni fuente fiable por sí misma.



En el día a día

Entender qué es y qué no es la IA no es un ejercicio filosófico. Determina decisiones de ingeniería concretas. Veamos tres situaciones reales.

Situación 1: ¿LLM o SQL? Tienes una base de datos con millones de registros de ventas y necesitas saber el importe total del último trimestre. Si crees que el LLM «sabe cosas», le pasarás el *prompt* «¿cuánto vendimos el último trimestre?» y obtendrás un número. El problema es que ese número puede ser correcto, aproximado o completamente inventado, y no tendrás forma de saberlo sin verificarlo manualmente. Si entiendes que el LLM predice tokens, usarás SQL para la consulta exacta y el LLM para resumir el resultado o redactar el informe. Cada herramienta para lo que sirve.

Situación 2: integrar IA generativa en un producto. Tu equipo quiere añadir un asistente que responda preguntas de los usuarios sobre la documentación. Si crees que el modelo es infalible, desplegarás sin *guardrails* ni evaluaciones y el primer usuario que reciba una respuesta incorrecta, con la confianza y elocuencia características de un LLM, actuará sobre información falsa. Si entiendes que el modelo alucina, diseñarás el sistema con verificación humana para decisiones de impacto, recuperación de documentos fuente para respuestas auditables y evaluaciones automáticas que midan la tasa de alucinación antes de cada despliegue.

Situación 3: depurar código generado. Le pides al modelo que escriba una función de autenticación y te devuelve treinta líneas impecables. Si crees que «piensa», confiarás y desplegarás. Si entiendes el mecanismo, revisarás cada línea como harías con el código de un compañero junior muy rápido pero sin criterio: ¿valida todos los casos límite?, ¿usa una biblioteca actualizada?, ¿hay algún error sutil de lógica que solo se manifiesta con ciertos valores de entrada? La elocuencia del código no garantiza su corrección.

Por qué debería importarte

La decisión técnica más importante al trabajar con IA no es qué modelo usar ni qué proveedor elegir. Es **entender qué estás usando realmente**.

De esa comprensión, o de su ausencia, se derivan todas las decisiones posteriores: cuándo desplegar un agente autónomo y cuándo un simple *prompt*, cómo diseñar las evaluaciones, qué nivel de supervisión humana exigir, cuánto presupuesto asignar a la revisión de salidas, qué arquitectura de *software* elegir para integrar el modelo.

Si entiendes que la IA no piensa, no entiende y no razona como un humano, puedes usarla mejor: le darás instrucciones más precisas, diseñarás sistemas que no dependan de su infalibilidad y confiarás menos ciegamente en sus respuestas.

El resto de este facsímil asume que has interiorizado esta idea. Cada capítulo vuelve a ella desde un ángulo distinto.

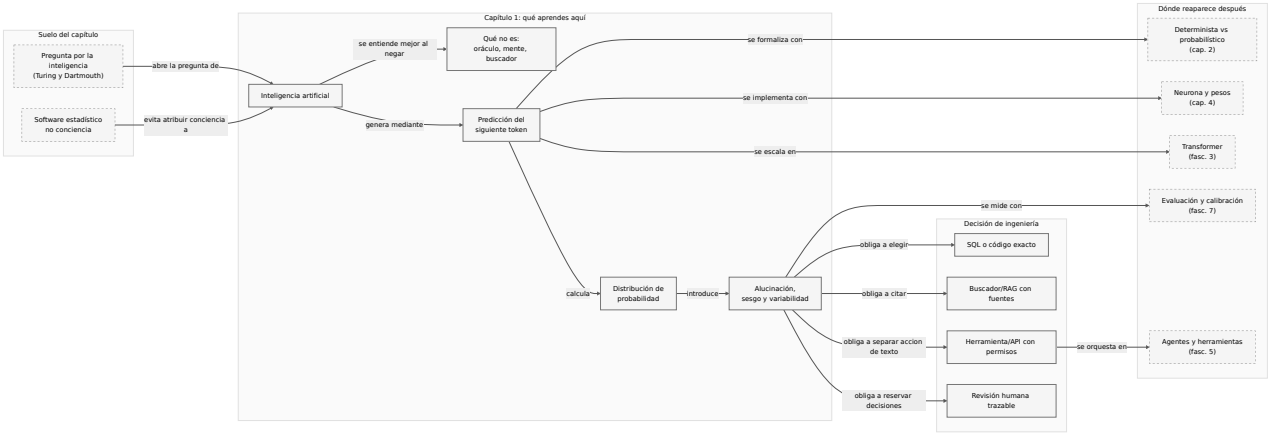
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Antropomorfizar («la IA cree que...», «el modelo piensa que...»)	Atribuir estados mentales a un sistema estadístico lleva a confusiones graves sobre sus capacidades y limitaciones. El modelo no cree nada: asigna probabilidades.	Sustituir mentalmente «la IA cree» por «el modelo asigna alta probabilidad a». Obligarse a usar el vocabulario técnico hasta que se convierta en hábito.
Tratarla como un oráculo infalible	Desplegar sin verificación humana en contextos donde el error tiene consecuencias económicas, legales o de seguridad. El modelo siempre responde con confianza, incluso cuando la respuesta es falsa.	Diseñar siempre con supervisión humana para decisiones de impacto. Si la decisión mueve dinero, afecta a personas o tiene consecuencias legales, un humano debe revisarla.
Confundir elocuencia con corrección	Un modelo puede generar texto perfectamente articulado, con estructura impecable y tono profesional, y ser completamente falso. La fluidez no es garantía de veracidad.	Verificar hechos contra fuentes externas. Las evaluaciones miden corrección, no estilo. Un texto bien escrito y equivocado es más peligroso que uno mal escrito y correcto.
Subestimar el no determinismo	Esperar la misma salida para el mismo *prompt*. El modelo muestrea de una distribución de probabilidad; dos ejecuciones pueden producir respuestas distintas. Esto rompe los tests unitarios tradicionales.	Diseñar tests que validen estructura y propiedades (¿la respuesta contiene un número?, ¿menciona los tres conceptos clave?), no texto exacto.

Cómo encaja todo

Este mapa se lee de izquierda a derecha. Primero aparece el suelo que hereda el capítulo: la pregunta histórica por la inteligencia y la realidad material de que hablamos de software estadístico. En el centro está la idea que debes llevarte: un modelo de lenguaje predice tokens, no consulta la verdad. A la derecha aparece la decisión de ingeniería que nace de esa idea: elegir entre consulta exacta, recuperación documental, generación, herramientas y supervisión.

Si el mapa solo dijera «IA conecta con Transformer», sería demasiado pobre. Lo importante es ver que este capítulo te prepara para distinguir mecanismos. Esa distinción reaparece en sistemas deterministas, neuronas, entrenamiento, RAG, agentes, evaluación y operación.


Suelo del capítulo
Pregunta por la inteligencia (Turing y Dartmouth)
Software estadístico no conciencia
abre la pregunta de
evita atribuir conciencia a
Inteligencia artificial
se entiende mejor al negar
Qué no es: oráculo, mente, buscador
genera mediante
Predicción del siguiente token
Capítulo 1: qué aprendes aquí
se formaliza con
se implementa con
se escala en
calcula
Distribución de probabilidad
introduce
Alucinación, sesgo y variabilidad
se mide con
Decisión de ingeniería
obliga a elegir
SQL o código exacto
obliga a citar
Buscador/RAG con fuentes
obliga a separar accion de texto
Herramienta/API con permisos
obliga a reservar decisiones
Revisión humana trazable
se orquesta en
Dónde reaparece después
Determinista vs probabilístico (cap. 2)
Neurona y pesos (cap. 4)
Transformer (fasc. 3)
Evaluación y calibración (fasc. 7)
Agentes y herramientas (fasc. 5)

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Inteligencia artificial	Rama de la informática que construye sistemas capaces de realizar tareas que normalmente requieren inteligencia humana, mediante modelos matemáticos que aprenden patrones a partir de datos.
LLM (*Large Language Model*)	Modelo de lenguaje de gran escala. Red neuronal con miles de millones de parámetros entrenada sobre cantidades masivas de texto para predecir y generar lenguaje.
Token	Unidad mínima de texto que el modelo procesa. Puede ser una palabra, parte de una palabra o un carácter. El modelo opera con tokens, no con palabras completas como las entendemos las personas.
Predicción del siguiente token	Mecanismo central de todo LLM: dada una secuencia de tokens, el modelo calcula qué token es más probable a continuación y, repitiendo la operación, genera texto coherente.
Softmax	Función que convierte las puntuaciones sin normalizar (logits) de cada token en una distribución de probabilidad que suma 1, de modo que ninguna probabilidad sea negativa.
Parámetro	Cada uno de los números que el modelo ajusta durante el entrenamiento. Un modelo de 175 000 millones de parámetros tiene 175 000 millones de números. Codifican patrones aprendidos, pero no son una base de datos consultable ni una memoria de hechos.
Alucinación	Fenómeno por el cual un LLM genera información sintácticamente correcta pero factualmente falsa, expresada con total seguridad. No es un *bug*: es una propiedad emergente de la predicción estadística.
RLHF (*Reinforcement Learning from Human Feedback*)	Técnica de post-entrenamiento donde personas evalúan respuestas del modelo y esa señal se usa para alinear su comportamiento con preferencias humanas.
Transformer	Arquitectura de red neuronal presentada en 2017 que sustituye el procesamiento secuencial por un mecanismo de atención paralela. Es la base de todos los LLMs modernos.
RAG (*Retrieval-Augmented Generation*)	Arquitectura que recupera documentos o datos externos antes de generar una respuesta, para que el modelo no dependa solo de sus patrones internos.
Sistema híbrido	Solución que combina varias piezas: consulta exacta, recuperación documental, generación, herramientas, permisos y revisión humana según el caso.

Antes de pasar página

- [ ] ¿Puedo explicar con mis propias palabras por qué un LLM no «piensa» ni «sabe» cosas como una persona? (Si no, vuelve a «Qué no es la inteligencia artificial».)
- [ ] ¿Entiendo el mecanismo de predicción del siguiente token? ¿Podría explicárselo a alguien sin conocimientos técnicos? (Si no, vuelve a «Cómo funciona por dentro».)
- [ ] ¿Puedo enumerar al menos tres cosas que la IA **no** es? (Si no, vuelve a «Qué no es la inteligencia artificial».)
- [ ] ¿Puedo enumerar al menos tres cosas que la IA **sí** es? (Si no, vuelve a «Qué sí es la inteligencia artificial».)
- [ ] ¿Conozco los hitos clave de la IA desde 1950 hasta hoy? (Si no, vuelve a la tabla en «Qué sí es la inteligencia artificial».)
- [ ] ¿Sé diferenciar entre elocuencia y corrección en una respuesta generada por IA? (Si no, vuelve a «Dónde solía tropezar yo», error «Confundir elocuencia con corrección».)
- [ ] ¿Podría decidir si un caso necesita SQL, RAG, LLM, herramienta/API, revisión humana o un sistema híbrido? (Si no, vuelve a «Cómo funciona por dentro».)

En resumen

IDEA FUERZA	DETALLE
La IA no piensa, no entiende y no es consciente.	Es un sistema de predicción estadística de tokens entrenado sobre cantidades masivas de datos. Tratarla como un oráculo es el error más caro que puedes cometer al diseñar sistemas con IA.
El mecanismo fundamental es la predicción del siguiente token.	Dada una secuencia, el modelo calcula qué palabra es más probable a continuación. Repetido miles de veces, produce texto coherente. Es álgebra lineal, probabilidad y muestreo.
La IA es un amplificador, no un sustituto del criterio humano.	Multiplica la velocidad de quien ya sabe. A quien no sabe, le da una falsa sensación de competencia. Diseña siempre con verificación humana para decisiones de impacto.
La elocuencia no es corrección.	Un texto bien escrito y completamente falso es la alucinación por excelencia. Verifica siempre contra fuentes. La confianza con la que el modelo afirma algo no guarda relación con la veracidad de lo que afirma.

Para saber más

Anthropic. (2026). *Agent SDK overview*. https://code.claude.com/docs/en/agent-sdk/overview

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

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

Google. (2026). *Agent Development Kit*. https://adk.dev/

Krizhevsky, A., Sutskever, I. y Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. En *Advances in Neural Information Processing Systems 25* (pp. 1097-1105). https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

McCarthy, J., Minsky, M. L., Rochester, N. y Shannon, C. E. (1956). A proposal for the Dartmouth summer research project on artificial intelligence. http://jmc.stanford.edu/articles/dartmouth.html

Model Context Protocol. (2026). *Specification*. https://modelcontextprotocol.io/specification

OpenAI. (2026). *Agents SDK*. https://developers.openai.com/api/docs/guides/agents

OpenAI. (2023). GPT-4 technical report. https://arxiv.org/abs/2303.08774

Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0

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

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236), 433-460. https://doi.org/10.1093/mind/LIX.236.433

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

Wolfram, S. (2023). What Is ChatGPT doing… and why does it work? *Stephen Wolfram Writings*. https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work/

Notas

1. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. Los autores definen la IA como el estudio de agentes que reciben percepciones del entorno y ejecutan acciones, sin atribuir conciencia o intencionalidad al sistema. ↩

2. 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. Las autoras acuñaron el término «stochastic parrots» para describir el riesgo de atribuir comprensión a modelos que solo reproducen patrones estadísticos. ↩

3. OpenAI. (2026). *Agents SDK*. https://developers.openai.com/api/docs/guides/agents; Anthropic. (2026). *Agent SDK overview*. https://code.claude.com/docs/en/agent-sdk/overview; Google. (2026). *Agent Development Kit*. https://adk.dev/; Model Context Protocol. (2026). *Specification*. https://modelcontextprotocol.io/specification. Fuentes consultadas el 10 de junio de 2026. No se citan como verdad permanente del mercado, sino como fotografía técnica del momento. ↩

4. Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236), 433-460. https://doi.org/10.1093/mind/LIX.236.433 ↩

5. McCarthy, J., Minsky, M. L., Rochester, N. y Shannon, C. E. (1956). A proposal for the Dartmouth summer research project on artificial intelligence. http://jmc.stanford.edu/articles/dartmouth.html ↩

6. Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0 ↩

7. Krizhevsky, A., Sutskever, I. y Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. En *Advances in Neural Information Processing Systems 25* (pp. 1097-1105). https://papers.nips.cc/paper/4824 ↩

8. 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 ↩

9. 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 ↩

10. OpenAI. (2023). GPT-4 technical report. https://arxiv.org/abs/2303.08774 ↩

1. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep Learning*. MIT Press. La función softmax es la forma estándar de convertir un vector de puntuaciones en una distribución de probabilidad en la capa de salida de un modelo. https://www.deeplearningbook.org ↩

Capítulo 02: Sistemas deterministas frente a sistemas probabilísticos

Entrando en el tema

Acabas de integrar un LLM en tu aplicación. Has escrito un *prompt* cuidadoso, has probado varias veces y la respuesta es buena. Como buena ingeniera, escribes un test:

```
assert respuesta == "Rust es un lenguaje de programación de sistemas..."
```

El test pasa en tu máquina. Haces *push*. El CI lo ejecuta. Falla. Lo vuelves a ejecutar en local. Pasa. Lo ejecutas otra vez. Falla.

No es un *bug*. Es una colisión entre dos formas de pensar el *software*. Y este capítulo existe para que esa colisión no te pille por sorpresa.

Qué es un sistema determinista

Un sistema determinista cumple una propiedad sencilla: **misma entrada, misma salida. Siempre.**

Llevas décadas programando en este mundo sin saber que tenía nombre. Cada vez que escribes `sumar(2, 3)` y esperas `5`, estás confiando en el determinismo. Cada compilador que traduce tu código a binario, cada consulta SQL que devuelve las mismas filas para la misma cláusula `WHERE`, cada función pura que no depende de nada externo.

El determinismo es el contrato silencioso de la ingeniería de *software* clásica: `f(x) = y`, y punto.[1] Si alguna vez rompe este contrato, lo llamamos *bug* y lo arreglamos.

```
function sumar(a, b) {
  return a + b;
}
// sumar(2, 3) = 5, siempre.
// sumar(2, 3) = 5, también mañana.
// sumar(2, 3) = 5, en cualquier servidor del mundo.
```

Este contrato es tan fundamental que nuestras herramientas (tests unitarios, *debuggers*, integración continua) están construidas sobre él. Un test que pasa y luego falla sin cambiar el código es, en el mundo determinista, una señal de alarma. Algo está roto.

Dónde aparece el no determinismo en IA

Aquí conviene ser muy precisos. No toda IA es no determinista. Un árbol de decisión entrenado, una regresión logística con pesos fijos o una red neuronal en modo inferencia pueden comportarse como funciones deterministas: misma entrada, mismos pesos, misma configuración, misma salida. Incluso dentro de un LLM, el *forward pass* que calcula los *logits* es una transformación matemática sobre números.

El no determinismo suele entrar después o alrededor del modelo: en el **muestreo** de tokens, en semillas aleatorias, en diferencias de precisión numérica, en *batching*, en kernels de GPU, en documentos recuperados que cambian o en herramientas externas que devuelven estados distintos. Por eso una integración real con IA no se prueba solo como una función pura. Se prueba como un sistema con capas.

En generación de texto, un LLM no devuelve «la respuesta correcta». Calcula una **distribución de probabilidad** sobre todos los tokens posibles y luego, según la configuración, puede **muestrear** de ella.

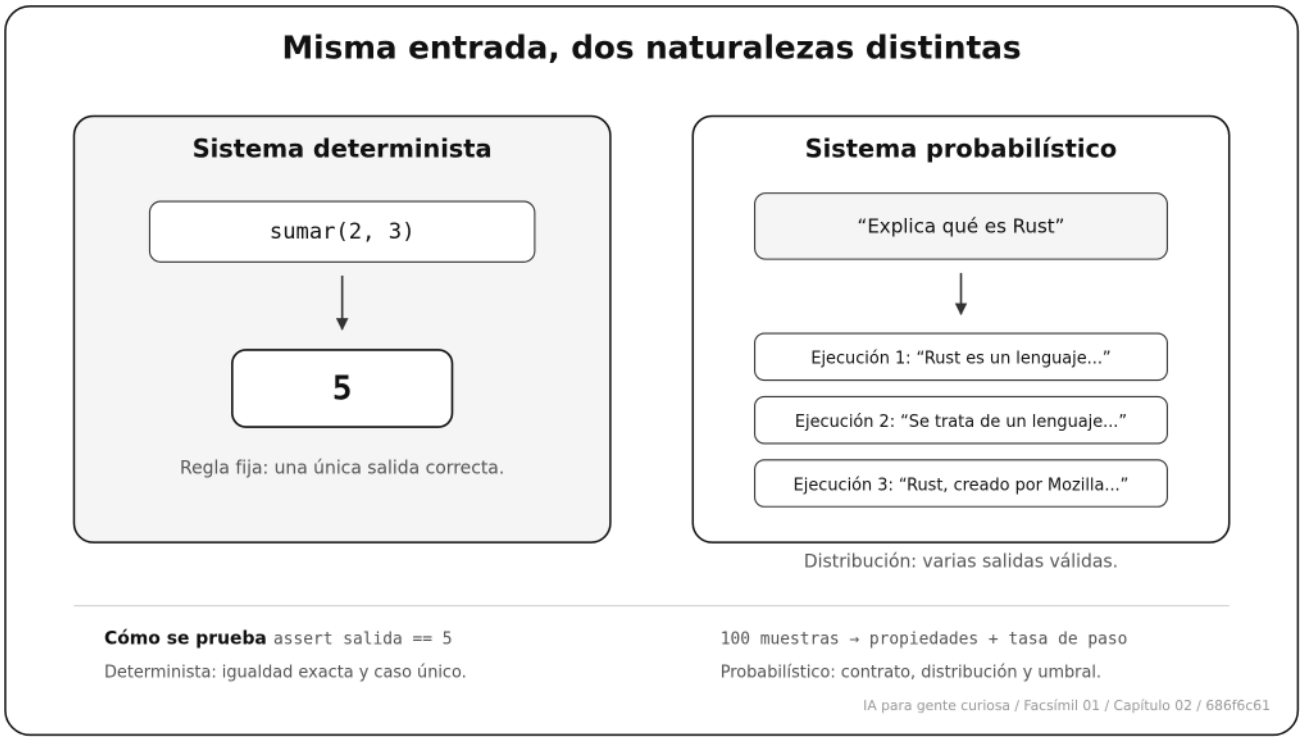
Esa palabra, muestrear, es la clave. No elige el token con mayor probabilidad (aunque puede configurarse para que lo haga). Elige un token al azar, pero con más probabilidad de elegir los que tienen puntuación alta. Es como lanzar un dado cargado: el 6 sale más a menudo, pero de vez en cuando sale un 3.

Por eso el mismo *prompt* puede producir respuestas distintas:

```
prompt: "Explica qué es Rust"

Ejecución 1: "Rust es un lenguaje de programación de sistemas..."
Ejecución 2: "Se trata de un lenguaje de programación enfocado en..."
Ejecución 3: "Rust, creado por Mozilla Research, es un lenguaje..."
```

Las tres son correctas. Las tres son razonables. Pero no son idénticas. Y si tu test espera una frase exacta, dos de cada tres ejecuciones fallarán sin que nada esté roto.



Cómo funciona por dentro

El viaje desde el *prompt* hasta el token generado tiene cinco estaciones:

```
Texto de entrada → Tokenización → Distribución de probabilidades → Muestreo → Token elegido
```

Estación 1: texto de entrada. Escribes tu *prompt*. Digamos: «Explica qué es Rust».

Estación 2: tokenización. El *prompt* se convierte en una secuencia de números. Este paso es determinista: el mismo texto siempre produce los mismos tokens. Lo vimos en el capítulo anterior y lo exploraremos en profundidad en el capítulo 9.

Estación 3: distribución de probabilidades. Los tokens atraviesan las capas del modelo. Al final, el modelo produce una lista de puntuaciones, llamadas *logits*, para cada token de su vocabulario. Esas puntuaciones se convierten en probabilidades mediante la función *softmax*.[2] El token con mayor probabilidad es el que el modelo asigna como continuación más probable, pero no es el único posible.

La idea se puede escribir así:

$$P(t_i \mid c) = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
t_i	Token candidato que podría venir después	`azul`
c	Contexto ya visto por el modelo	`El cielo es de color`
z_i	*Logit*: puntuación cruda asignada al token t_i	4,0 para `azul`
T	Temperatura usada antes de convertir logits en probabilidades	1,0
$\sum_j$	Suma sobre todos los tokens candidatos del vocabulario	`azul`, `gris`, `rojo`, …
$P(t_i \mid c)$	Probabilidad de elegir t_i dado el contexto	0,84 para `azul`

Con tres tokens candidatos y logits sencillos, la temperatura cambia la forma de la distribución:

TOKEN	LOGIT	PROBABILIDAD CON $T = 0{,}5$	PROBABILIDAD CON $T = 1$	PROBABILIDAD CON $T = 2$
`azul`	4,0	0,98	0,84	0,63
`gris`	2,0	0,018	0,11	0,23
`rojo`	1,0	0,002	0,04	0,14

No memorices los números. Quédate con el mecanismo: al bajar la temperatura, la distribución se concentra en el candidato más probable; al subirla, se reparte más probabilidad entre alternativas. Esto no convierte al modelo en «más inteligente» o «menos inteligente». Cambia la política de muestreo.

Hay una forma precisa de nombrar lo que cambia: la temperatura ajusta la *entropía* de la distribución, la medida de incertidumbre que formalizó Claude Shannon en 1948.[3] Con temperatura baja hay poca entropía: casi toda la probabilidad se concentra en un token y el modelo se vuelve predecible. Con temperatura alta hay mucha entropía: la probabilidad se reparte y varias continuaciones compiten.

Estación 4: muestreo. Aquí puede entrar el no determinismo. Tres parámetros controlan cuánta aleatoriedad se introduce:

- **`temperature`**: controla lo plano o picuda que es la distribución. Con temperatura baja (cercana a 0), el modelo casi siempre elige el token más probable. Con temperatura alta (cercana a 2), incluso tokens con probabilidad baja tienen opciones.[4] Lo veremos en detalle en el capítulo sobre parámetros de configuración.

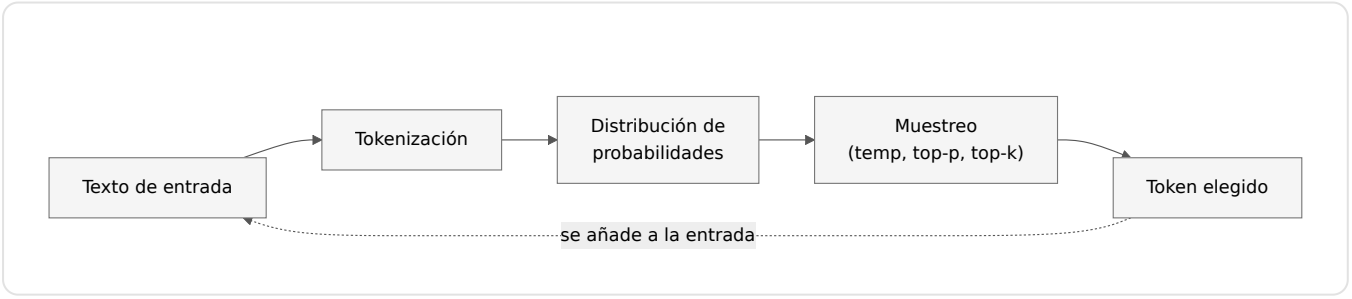
- `top-p`: en lugar de considerar todos los tokens, el modelo solo considera los más probables hasta que sus probabilidades suman `p`. Si `top-p = 0.9`, el modelo elige entre el conjunto mínimo de tokens que suman el 90 % de la probabilidad total. Con la distribución de antes (`azul` 0,84, `gris` 0,11, `rojo` 0,04) y `top-p = 0.9`, el modelo ordena de mayor a menor y va sumando: `azul` llega a 0,84, añade `gris` y alcanza 0,95, ahí corta. El conjunto elegible queda en {`azul`, `gris`} y `rojo` se descarta: por improbable que fuera, ya no puede salir.
- `top-k`: el modelo solo considera los `k` tokens más probables y descarta el resto. Con `top-k = 2` sobre esa misma distribución, solo `azul` y `gris` siguen en juego, sin mirar su probabilidad acumulada.

Con `temperature = 0`, el modelo se acerca al determinismo: casi siempre elige el token más probable. Pero no lo garantiza del todo: pequeñas diferencias en precisión numérica, *batching* o *hardware* pueden producir variaciones.

¿Por qué no lo garantiza? Porque la suma en coma flotante no es asociativa: sumar los mismos números en distinto orden puede cambiar el resultado en los últimos decimales. Cuando el modelo corre en una GPU, el orden en que se acumulan millones de productos depende del paralelismo, del tamaño del lote (*batching*) y del *hardware* concreto. Esas diferencias minúsculas, cuando dos tokens tienen puntuaciones muy parecidas, bastan para que de vez en cuando gane el otro.

Si necesitas acercarte a la reproducibilidad, tienes dos palancas. La primera es la **semilla** (*seed*): algunos proveedores permiten fijarla para que, con la misma entrada y configuración, el muestreo siga el mismo camino. La segunda, fácil de pasar por alto, es **fijar la versión del modelo**: cuando un proveedor actualiza el modelo por debajo del mismo nombre, la misma pregunta puede devolver otra respuesta semanas después sin que tú hayas cambiado una sola línea. Para sistemas que dependen de la estabilidad, fija la versión y registra cuál usaste.

Estación 5: token elegido. El modelo devuelve un token. Ese token se añade a la secuencia de entrada y el ciclo se repite hasta generar un token de fin.



En el día a día

La variabilidad de los LLM en productos reales tiene consecuencias prácticas inmediatas. Veamos tres.

Escribir tests para código que usa LLMs. No puedes hacer `assert respuesta == "texto exacto"`. En su lugar, validas propiedades: ¿la respuesta contiene los tres puntos clave que pediste?, ¿tiene la estructura esperada (JSON, Markdown, lista)?, ¿menciona los conceptos correctos?, ¿está en el idioma solicitado?[5] Es un cambio de mentalidad: de validar igualdad exacta a validar propiedades semánticas.

```
# Frágil: una redacción distinta y válida rompe el test.
# assert respuesta == "Rust es un lenguaje de programación de sistemas..."

# Robusto: validamos propiedades, no texto exacto.
assert "Rust" in respuesta
assert any(t in respuesta.lower() for t in ["lenguaje", "programación"])
assert 20 < len(respuesta.split()) < 300
```

Depurar comportamientos inconsistentes. Un usuario reporta que el asistente «a veces responde mal». En un sistema determinista, reproducirías la entrada y obtendrías el mismo error. Con un LLM, necesitas ejecutar varias veces, observar patrones y usar evaluaciones automáticas que midan la tasa de error en lugar de comprobar ejecuciones individuales.

Diseñar experiencias de usuario. Si tu producto muestra respuestas generadas por IA, el usuario esperará que la misma pregunta reciba una respuesta consistente. Si cada vez que pregunta «¿cuál es mi saldo?» obtiene una redacción distinta, puede percibirlo como un error. La solución no es forzar el determinismo: es diseñar la experiencia para que la variabilidad sea una ventaja (creatividad, adaptación al contexto) y no una fuente de confusión.

Por qué debería importarte

El paso más importante que darás al trabajar con IA no es técnico: es mental.

Llevas años (décadas, probablemente) entrenando tu cerebro para pensar en sistemas deterministas. Un *bug* es una desviación del comportamiento esperado. Un test que falla es una alarma. La reproducibilidad es sagrada.

Trabajar con IA exige un modelo mental distinto. No preguntas «¿esto devuelve X?», sino «¿esto devuelve algo razonable dentro de un rango?». No validas igualdad, validas propiedades. No depuras ejecuciones individuales, observas distribuciones de resultados.

Este cambio de mentalidad no es opcional. Si diseñas sistemas con IA generativa como si siempre fueran funciones puras de texto a texto, construirás sistemas frágiles que fallarán en producción por razones que tus tests no pueden detectar.

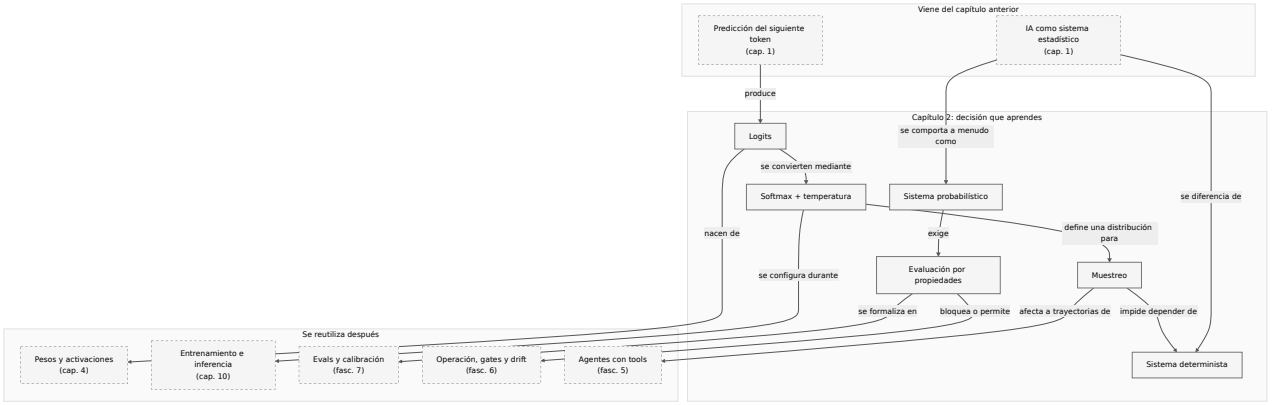
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Testear texto exacto	Escribir `assert respuesta == "Rust es un lenguaje..."` condena tu test a fallar aleatoriamente. La misma pregunta puede recibir respuestas correctas pero textualmente distintas.	Valida propiedades: ¿contiene las palabras clave?, ¿respeta el formato pedido?, ¿tiene una longitud razonable?
Decir "la IA no es determinista" sin matices	Mezcla cosas distintas: modelo, muestreo, runtime, proveedor, RAG y herramientas. Una parte del sistema puede ser determinista y otra no.	Dibuja la frontera: qué pieza es función pura, qué pieza muestrea, qué pieza consulta estado externo y qué pieza depende de infraestructura.
Confiar en que `temperature=0` es totalmente determinista	Incluso con temperatura cero, pequeñas diferencias en precisión numérica, *batching* o *hardware* pueden producir variaciones. No es un interruptor de determinismo: es un atenuador de variabilidad.	Si necesitas determinismo absoluto, usa modelos clásicos. Si usas un LLM, diseña para tolerar variabilidad.
Depurar como si fuera un *bug* determinista	Ante una respuesta incorrecta de un LLM, repetir el mismo *prompt* para «reproducir el error» puede no funcionar. El error puede ser estadístico, no sistemático.	Ejecuta múltiples veces, mide la tasa de error y busca patrones. Usa evaluaciones automáticas.
Ignorar los parámetros de muestreo	Usar los valores por defecto sin entender qué hace cada uno. Una temperatura alta en un contexto donde necesitas respuestas consistentes genera frustración en el usuario.	Ajusta `temperature`, `top-p` y `top-k` según el caso de uso. Creatividad alta para *brainstorming*, baja para respuestas factuales.

Cómo encaja todo

Este mapa se lee de arriba abajo. El capítulo hereda del capítulo 1 la idea de predicción del siguiente token. Aquí la vuelve operativa: si hay distribución y muestreo, ya no puedes probar, depurar ni diseñar experiencia de usuario como si todo fuera una función pura.

La decisión central del capítulo es cambiar de mentalidad: de salida única a contrato probabilístico. Esa decisión reaparece después en parámetros de inferencia, evaluación, operación, RAG y agentes, porque todos esos sistemas necesitan medir comportamiento en muchas ejecuciones, no en una sola respuesta bonita.


Viene del capítulo anterior
Predicción del siguiente token (cap. 1)
IA como sistema estadístico (cap. 1)
produce
Capítulo 2: decisión que aprendes
Logits
se comporta a menudo como
se convierten mediante
Softmax + temperatura
Sistema probabilístico
se diferencia de
nacen de
exige
define una distribución para
se configura durante
Evaluación por propiedades
Muestreo
se formaliza en
bloquea o permite
afecta a trayectorias de
impide depender de
Sistema determinista
Se reutiliza después
Pesos y activaciones (cap. 4)
Entrenamiento e inferencia (cap. 10)
Evals y calibración (fasc. 7)
Operación, gates y drift (fasc. 6)
Agentes con tools (fasc. 5)

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Sistema determinista	Sistema donde la misma entrada produce siempre la misma salida. Un compilador, una consulta SQL o una función pura son ejemplos.
Sistema estocástico	Sistema donde la misma entrada puede producir salidas diferentes porque incorpora elementos de probabilidad o muestreo.
Muestreo	Proceso de elegir un valor concreto a partir de una distribución de probabilidad. En un LLM, se muestrea cuál será el siguiente token.
Temperatura	Parámetro que controla cuánta aleatoriedad se introduce al muestrear. Valores bajos producen respuestas más predecibles; valores altos, más creativas.
Logit	Puntuación cruda que asigna el modelo a cada token antes de convertirse en probabilidad.
Softmax	Función que convierte puntuaciones crudas en probabilidades que suman 1.
Distribución de probabilidad	Asignación de una probabilidad a cada resultado posible. En un LLM, a cada token del vocabulario.
top-p	Estrategia de muestreo que considera solo el conjunto mínimo de tokens cuya probabilidad acumulada llega al valor `p`.
top-k	Estrategia de muestreo que considera solo los `k` tokens más probables y descarta el resto.
Entropía	Medida de la incertidumbre de una distribución de probabilidad. La temperatura baja la reduce y la temperatura alta la aumenta.
Semilla	Valor que fija el punto de partida del muestreo para que la misma entrada y configuración sigan el mismo camino.
Evaluación probabilística	Forma de probar sistemas variables midiendo propiedades, tasas de paso y distribuciones, no igualdad exacta de texto.

Antes de pasar página

- [ ] ¿Puedo explicar con mis propias palabras la diferencia entre un sistema determinista y uno estocástico? (Si no, vuelve a «Qué es un sistema determinista» y «Dónde aparece el no determinismo en IA».)

- [ ] ¿Entiendo por qué un LLM puede dar respuestas distintas al mismo *prompt*? (Si no, vuelve a «Cómo funciona por dentro».)
- [ ] ¿Sé qué hace el parámetro `temperature`? (Si no, vuelve a la estación 4 en «Cómo funciona por dentro».)
- [ ] ¿Puedo nombrar al menos dos estrategias para testear sistemas que usan LLMs? (Si no, vuelve a «En el día a día».)
- [ ] ¿Entiendo por qué `temperature = 0` no garantiza determinismo absoluto? (Si no, vuelve a «Dónde solía tropezar yo», error «Confiar en que temperature = 0 es totalmente determinista».)
- [ ] ¿Sé explicar la diferencia entre comprobar una coincidencia exacta y comprobar una propiedad al testear un sistema con LLMs? (Si no, vuelve a «En el día a día».)

En resumen

IDEA FUERZA	DETALLE
En IA generativa, la variabilidad suele aparecer en el muestreo y en el sistema que rodea al modelo.	El mismo *prompt* puede producir respuestas distintas porque el modelo puede muestrear de una distribución de probabilidad, porque el runtime no siempre es idéntico o porque el contexto externo cambia.
Programar con IA exige un cambio de mentalidad.	Pasas de validar igualdad exacta a validar propiedades. De preguntar «¿esto devuelve X?» a preguntar «¿esto devuelve algo razonable?».
Los parámetros de muestreo controlan la aleatoriedad, no la eliminan.	`temperature`, `top-p` y `top-k` ajustan cuánta variabilidad hay, pero ni siquiera `temperature = 0` garantiza determinismo absoluto.
Testear sistemas con IA requiere herramientas distintas.	No tests de igualdad textual. Sí evaluaciones automáticas, validación de estructura, *asserts* sobre propiedades semánticas.

Para saber más

Anthropic. (2025). Develop tests for LLM applications. https://platform.claude.com/docs/en/build-with-claude/develop-tests

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

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

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

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., 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. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. Los autores dedican el capítulo 2 al concepto de agente racional, estableciendo la diferencia entre entornos deterministas (el siguiente estado está completamente determinado por el estado actual y la acción) y entornos estocásticos. ↩

2. Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer. El capítulo 4 aborda los modelos lineales para clasificación y la transformación de puntuaciones en probabilidades mediante la función *softmax*. ↩

3. 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 definió la entropía como la cantidad de incertidumbre de una distribución de probabilidad, y es la base de toda la teoría de la información. ↩

4. 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. Este artículo introdujo el muestreo *top-p* (*nucleus sampling*) y analiza en profundidad por qué los métodos de muestreo ingenuos producen texto degenerado. ↩

5. Anthropic. (2025). Develop tests for LLM applications. https://platform.claude.com/docs/en/build-with-claude/develop-tests ↩

Capítulo 03: Principios fundamentales de la inteligencia artificial

Entrando en el tema

Has oído las palabras. Aprendizaje supervisado. *Transformers*. RLHF. *Scaling laws*. Suenan a conceptos avanzados que solo entiende quien tiene un doctorado. Pero no lo son. Son cinco ideas, sorprendentemente accesibles, que explican por qué la IA moderna funciona como funciona.

Este capítulo no te va a convertir en experto en ninguna de ellas. Para eso están el resto de capítulos del facsímil. Pero sí te va a dar el mapa. Cuando termines, sabrás nombrar las cinco patas de la mesa y, sobre todo, sabrás qué pata buscar cuando algo falle.

Aprendizaje supervisado: aprender con ejemplos

Imagina que quieres enseñarle a alguien a distinguir facturas falsas de facturas verdaderas. Le enseñas cien facturas. Cada una lleva una etiqueta: «verdadera» o «falsa». La persona observa, compara, detecta patrones (el logotipo borroso, el NIF que no cuadra, el formato de fecha extraño) y, tras revisar suficientes ejemplos, es capaz de clasificar facturas nuevas que no había visto antes.

Eso es el aprendizaje supervisado.

En términos técnicos: le das al modelo un conjunto de ejemplos etiquetados (cada entrada con su salida esperada) y el modelo ajusta sus parámetros para minimizar el error entre lo que predice y la etiqueta real.[1] Es la base del *fine-tuning*: coges un modelo pre-entrenado y lo ajustas con ejemplos específicos de tu dominio. El modelo ya sabe lenguaje; tú le enseñas tu jerga.

El aprendizaje supervisado es el paradigma más intuitivo porque se parece a como aprendemos muchas cosas: practicando con ejemplos cuya respuesta correcta conocemos.

Aprendizaje no supervisado: encontrar patrones sin pistas

Ahora imagina que tienes un montón de facturas sin etiquetar. No sabes cuáles son verdaderas ni falsas. Pero aun así, al observarlas, detectas que algunas se parecen mucho entre sí (mismo formato, mismos proveedores, mismos importes redondos) y otras son

completamente distintas. Sin que nadie te haya dicho qué es cada cosa, has encontrado estructura.

Eso es el aprendizaje no supervisado.

El modelo recibe datos sin etiquetar y busca patrones, agrupaciones o representaciones. No hay una «respuesta correcta» contra la que comparar. El modelo descubre la estructura por sí mismo.

Este paradigma es pariente cercano del responsable silencioso del éxito de los LLMs. Cuando un modelo como GPT se pre-entrena sobre miles de millones de documentos de internet, no hay un humano etiquetando cada frase. El modelo se entrena con una tarea **auto-supervisada** (*self-supervised*): predecir la siguiente palabra.[2] Conviene precisarlo, porque es un matiz que el campo distingue: no es exactamente aprendizaje no supervisado. En el no supervisado no hay respuesta correcta y solo se busca estructura. En el auto-supervisado sí hay etiqueta, pero sale de los propios datos: la siguiente palabra del texto es la respuesta. Esa diferencia es justo lo que permite aprovechar todo internet como ejemplos etiquetados sin que nadie etiquete nada a mano.

Post-training: enseñar preferencias, no solo hechos

Un modelo pre-entrenado sabe completar frases. Pero no sabe que es mejor decir «no tengo esa información» que inventársela. No sabe que ciertos temas requieren un tono más cuidadoso. No sabe cuándo callar.

El *post-training* resuelve esto. Tras el pre-entrenamiento masivo (aprendizaje no supervisado), se aplican técnicas que enseñan al modelo **preferencias**:

- **SFT** (*Supervised Fine-Tuning*): se entrena al modelo con ejemplos de conversaciones de alta calidad escritas por personas. El modelo aprende el formato y el tono deseados.
- **RLHF** (*Reinforcement Learning from Human Feedback*): personas comparan pares de respuestas del modelo y eligen cuál es mejor. Esa señal de preferencia entrena un modelo de recompensa que guía al modelo principal. Es aprendizaje por refuerzo, pero la recompensa la define un humano, no el entorno.[3]
- **DPO** (*Direct Preference Optimization*): una alternativa a RLHF que aprende directamente de las preferencias humanas sin necesidad de entrenar un modelo de recompensa separado.[4]

RLHF y DPO conectan con el tercer gran paradigma del aprendizaje automático, el que completa la tríada junto al supervisado y el no supervisado: el **aprendizaje por refuerzo**, donde un agente aprende por ensayo y error guiado por una señal de recompensa, no por ejemplos etiquetados. Lo distintivo en RLHF es que esa recompensa la define la preferencia humana, no el entorno. El facsímil 10 está dedicado por completo a este paradigma.

El *post-training* es la capa que convierte un modelo que «sabe cosas» en un modelo con el que puedes hablar. Lo exploraremos en detalle en el facsímil 4.

Atención: mirar todo a la vez

Piensa en cómo lees esta frase: «El gato que perseguía al ratón que robó el queso que estaba en la cocina es negro». Para saber que «negro» se refiere a «gato», tu cerebro ha tenido que conectar dos palabras separadas por una larga cláusula subordinada. Ha tenido que prestar atención a la relación entre «gato» y «negro» ignorando temporalmente el ratón, el queso y la cocina.

Los modelos anteriores a 2017 no podían hacer esto eficientemente. Procesaban el texto palabra por palabra, en orden, y para cuando llegaban a «negro» ya se habían olvidado de «gato».

El mecanismo de **atención**, presentado en el artículo «Attention Is All You Need»,[5] cambió esto radicalmente. Permite al modelo mirar **todas las palabras de la entrada simultáneamente** y decidir, para cada palabra que va a generar, cuáles de las palabras de entrada son relevantes y en qué medida.

Es el mecanismo que hizo posibles los LLMs modernos. Sin atención, los modelos no podrían manejar contextos largos ni capturar dependencias entre palabras separadas por decenas de miles de tokens. El facsímil 3 está dedicado íntegramente a abrir esta caja negra y entenderla por dentro.

Scaling laws: escala, coste y límites

En 2020, un grupo de investigadores de OpenAI se preguntó algo aparentemente simple: ¿qué pasa si entrenamos modelos cada vez más grandes con más datos y más computación? La respuesta, publicada en el artículo «Scaling Laws for Neural Language Models»,[6] fue sorprendentemente ordenada: el rendimiento mejora de forma **predecible** con la escala.

La relación simplificada es: **más datos + más parámetros + más computación = modelo más capaz**. Pero escrita así puede engañar. No significa que cualquier modelo grande sea mejor en cualquier tarea, ni que puedas ignorar la calidad de los datos, la arquitectura, el entrenamiento o la evaluación. Significa que, bajo ciertas condiciones experimentales, la pérdida baja siguiendo una relación bastante regular cuando aumentan parámetros, datos y cómputo.

Esto no es una analogía: hay una fórmula real detrás. Una forma de escribir una ley de escala, en la línea de los trabajos de Kaplan y de Chinchilla, es esta:[7]

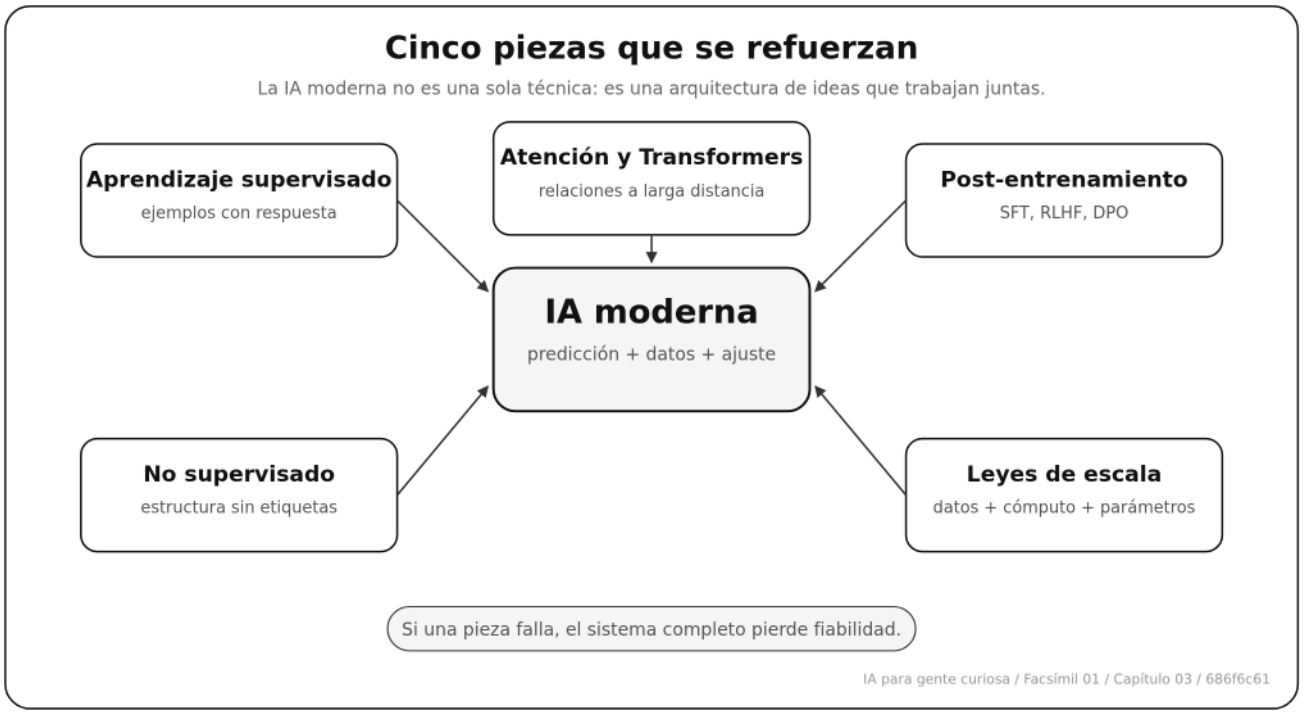
$$L(N, D) \approx L_\infty + \frac{A}{N^\alpha} + \frac{B}{D^\beta}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$L(N, D)$	Pérdida esperada del modelo entrenado con N parámetros y D tokens	1,72
L_∞	Límite inferior aproximado: pérdida que no desaparece aunque escales	1,55
N	Número de parámetros del modelo	7B, 70B, 405B
D	Cantidad de datos de entrenamiento, normalmente tokens	1T tokens
A, B	Constantes ajustadas empíricamente	dependen del experimento
α, β	Exponentes que dicen cuánto ayuda escalar parámetros o datos	valores pequeños, no milagrosos

Esta fórmula es una forma pedagógica de leer la idea, no una receta para presupuestar tu modelo ni una garantía de rendimiento. Las constantes y exponentes se estiman empíricamente en un experimento concreto; cambian con arquitectura, datos, régimen de entrenamiento y métrica. La intuición importante es que hay **rendimientos decrecientes**. Añadir diez veces más parámetros no suele darte diez veces más calidad. Además, Hoffmann y coautores mostraron con Chinchilla que muchos modelos grandes estaban infraentrenados: para un presupuesto de cómputo fijo, no basta con aumentar parámetros; también hay que aumentar datos de entrenamiento en proporción razonable.[8]

Esto explica por qué la industria invierte miles de millones en clústeres de GPUs, pero también explica por qué un equipo serio no compra «el modelo más grande» sin medir. Las *scaling laws* sirven para razonar sobre tendencias de entrenamiento; tu decisión de producto se toma con evals, latencia, coste por token, privacidad, facilidad de operación y calidad en tu tarea.

La lectura menos optimista es clara: mejorar el rendimiento requiere inversiones crecientes. Pasar de un modelo bueno a uno excelente cuesta mucho más que pasar de uno malo a uno aceptable. Entender esta dinámica es clave para tomar decisiones realistas sobre qué modelos usar y cuándo.



Cómo funciona por dentro

Cuando uno mira los cinco principios desde lejos, parece que cada uno exige una maquinaria distinta. No es así. Bajo el aprendizaje supervisado, el no supervisado y el post-entrenamiento late un único motor común: el descenso de gradiente (*gradient descent*), el mismo procedimiento que se desarrolla en los capítulos 6 y 7. La idea es siempre la misma. El modelo tiene millones (o miles de millones) de parámetros ajustables. Se mide cómo de mal lo hace con una función de pérdida (*loss*). Se calcula en qué dirección habría que mover cada parámetro para que esa pérdida baje un poco. Y se da un pequeño paso en esa dirección. Repetido millones de veces, ese gesto humilde es lo que convierte una red de números aleatorios en un modelo capaz.

Si la maquinaria es la misma, ¿qué distingue entonces a un paradigma de otro? La respuesta es más sencilla y más elegante de lo que suele contarse: lo único que cambia es de dónde sale la señal de error, es decir, contra qué se compara la predicción del modelo para saber si ha acertado o ha fallado. El motor no cambia. Cambia la fuente de la verdad que alimenta ese motor.

En el aprendizaje supervisado, la señal sale de una etiqueta puesta por una persona. El modelo ve una factura, predice «verdadera», y la pérdida mide la distancia entre esa predicción y la etiqueta humana («falsa»). Esa distancia es el error, y de ese error nace el gradiente que corrige los parámetros. En el aprendizaje no supervisado, y muy en particular en su variante auto-supervisada (*self-supervised*), no hay nadie etiquetando: la señal sale de la propia estructura de los datos. Cuando un modelo de lenguaje predice la siguiente palabra de un texto, la «etiqueta» es simplemente la palabra que de verdad venía a continuación. El

texto se etiqueta a sí mismo. Por eso se puede entrenar con internet entero sin pagar a nadie por anotar nada. En el post-entrenamiento (RLHF y DPO), la señal sale de las preferencias humanas: a una persona se le muestran dos respuestas del modelo y elige cuál prefiere. Esa preferencia (esta es mejor que aquella) se convierte en la fuente de error que ajusta los parámetros para que el modelo tienda a producir lo que la gente prefiere.

Conviene ser honestos aquí, porque de los cinco principios del capítulo solo tres son de verdad paradigmas de aprendizaje. La atención no lo es: no es una forma de extraer señal de error, sino un mecanismo arquitectónico, una pieza del diseño interno del modelo que decide cómo este mira la entrada y pondera la relevancia de unas palabras respecto a otras. La atención no entrena al modelo; es parte de lo que el motor de gradiente ajusta. Y las *scaling laws* tampoco son un mecanismo: son una ley empírica, una observación regular sobre cómo mejora el rendimiento cuando aumentan parámetros, datos y cómputo. Describen el comportamiento del sistema visto desde fuera, no una pieza que puedas tocar. Tanto la atención como las leyes de escala operan sobre el mismo motor de descenso de gradiente, pero en otra capa: una en la arquitectura, la otra en la observación del conjunto.

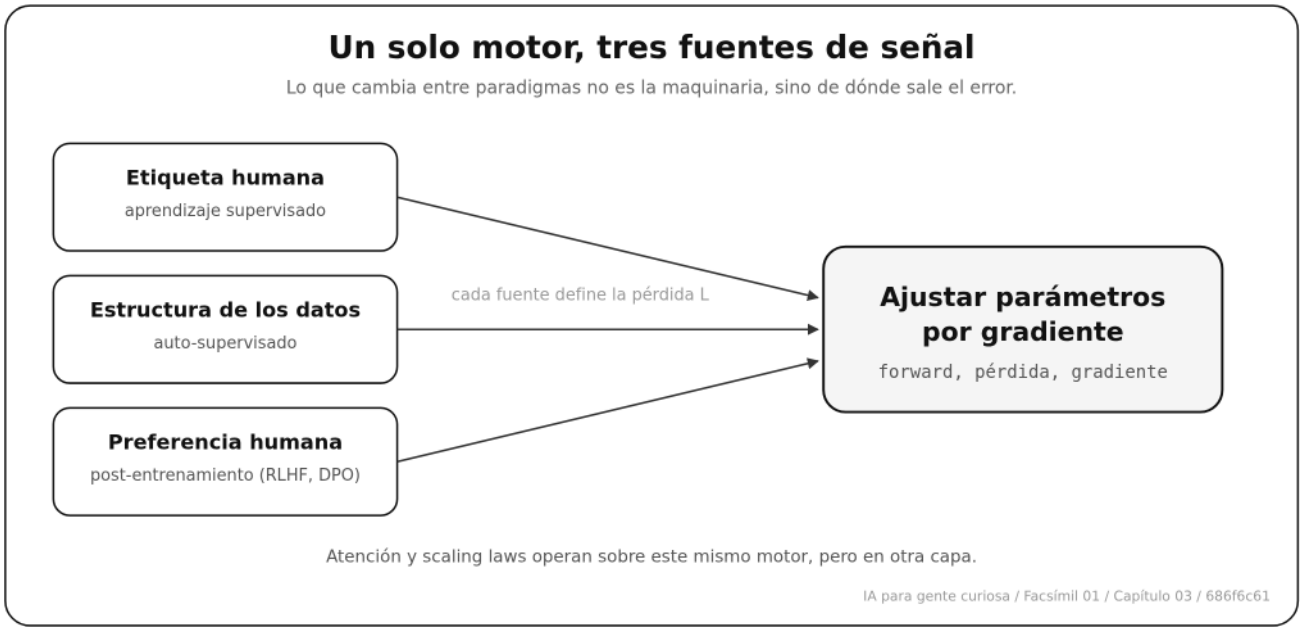
Vale la pena bajar al detalle del mecanismo común, porque entenderlo de una vez sirve para los tres paradigmas a la vez. El ciclo tiene cuatro pasos. Primero, el paso hacia delante (*forward pass*): los datos de entrada atraviesan el modelo y este produce una predicción. Segundo, el cálculo de la pérdida: se compara esa predicción con la señal de referencia, y aquí es donde entra la única diferencia entre paradigmas, porque la referencia es una etiqueta humana, o la siguiente palabra del propio texto, o una preferencia entre dos respuestas. Tercero, el gradiente: mediante retropropagación (*backpropagation*, el tema del capítulo 7) se calcula, para cada parámetro, cuánto contribuye al error y en qué dirección debería moverse para reducirlo. Cuarto, la actualización: cada parámetro se desplaza un poquito en la dirección que reduce la pérdida, regulado por un factor llamado tasa de aprendizaje (*learning rate*).

Una forma compacta de escribir ese cuarto paso, la actualización, es esta:

$$\theta_{t+1} = \theta_t - \eta \, \nabla_\theta L$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
θ_t	Los parámetros del modelo en el paso actual	una red de millones de pesos
θ_{t+1}	Los parámetros ya corregidos para el paso siguiente	la red, un poquito mejor
η	Tasa de aprendizaje: tamaño del paso que se da	0,0001
$\nabla_\theta L$	Gradiente de la pérdida: dirección en la que el error sube	un vector con un valor por parámetro
L	Pérdida, calculada según la fuente de señal del paradigma	etiqueta, texto o preferencia

En palabras: coge los parámetros actuales, mira en qué dirección crece el error y muévete un paso pequeño en sentido contrario. Lo notable es que esta fórmula es idéntica en los tres paradigmas. Lo único que cambia entre ellos está escondido dentro de L: qué se usó como referencia para medir el error. Cambia la fuente de la señal, no el motor que la consume. Esa es, quizá, la idea más unificadora de todo el aprendizaje automático moderno, y la que conviene llevarse de este capítulo antes de pasar a las decisiones prácticas.



En el día a día

Cada uno de estos principios se traduce en decisiones de ingeniería.

Si tu problema tiene datos etiquetados, el aprendizaje supervisado, posiblemente mediante *fine-tuning*, es tu primer candidato. Clasificar tickets de soporte, detectar fraude, predecir abandono de clientes: todo esto es supervisado.

Si solo tienes datos sin etiquetar y necesitas encontrar estructura (agrupar clientes, detectar anomalías, reducir dimensiones), el aprendizaje no supervisado es la herramienta. Pero cuidado: que el modelo encuentre grupos no significa que esos grupos signifiquen algo útil para tu negocio. Lo veremos en el capítulo sobre *clustering*.

Si estás integrando un LLM en un producto, primero separa qué problema tienes. Si falta conocimiento que cambia (políticas internas, catálogo, normativa, documentación) normalmente necesitas recuperación de información, no tocar pesos. Si falta formato, tono, criterios de abstención o una conducta repetida y medible, entonces puede tener sentido *post-training*, *fine-tuning* o adaptadores. No basta con un buen *prompt*, pero tampoco se responde a todo con *fine-tuning*: necesitas saber qué pieza del sistema está fallando.

Si necesitas que el modelo maneje contexto largo, la atención es tu aliada y tu enemiga. Permite procesar documentos extensos, pero su coste computacional crece cuadráticamente con la longitud de la secuencia. Cada vez que duplicas el contexto, el coste se multiplica por cuatro. Diseñar estrategias de ventana de contexto (cuánto texto incluir, cuánto resumir, cuánto recuperar) es una decisión de arquitectura que depende de entender cómo funciona la atención.

Si estás decidiendo qué modelo usar, las *scaling laws* te dicen que más grande no siempre es mejor *para ti*. Un modelo de 7 000 millones de parámetros puede ser suficiente para tu caso de uso, y uno de 70 000 millones puede ser excesivamente caro y lento sin aportar una mejora proporcional. La decisión correcta depende de tus requisitos de calidad, latencia y coste.

Por qué debería importarte

Estos cinco principios no son trivia académica. Son el vocabulario con el que se toman todas las decisiones técnicas en IA.

Cuando alguien te diga «vamos a hacer *fine-tuning*», necesitas saber que está hablando de aprendizaje supervisado sobre un modelo pre-entrenado. Cuando escuches «este modelo tiene mejor atención», necesitas entender qué significa eso para tu caso de uso. Cuando un proveedor te diga «nuestro modelo es más grande, por tanto mejor», las *scaling laws* te dan el marco para evaluar si ese «mejor» justifica el coste.

Cada capítulo a partir de ahora asume que conoces este mapa. La neurona artificial es la pieza mínima del aprendizaje supervisado. La retropropagación es cómo se ajustan los parámetros. El Transformer es la atención llevada a la práctica. El *fine-tuning* es aprendizaje supervisado aplicado. Todo encaja.

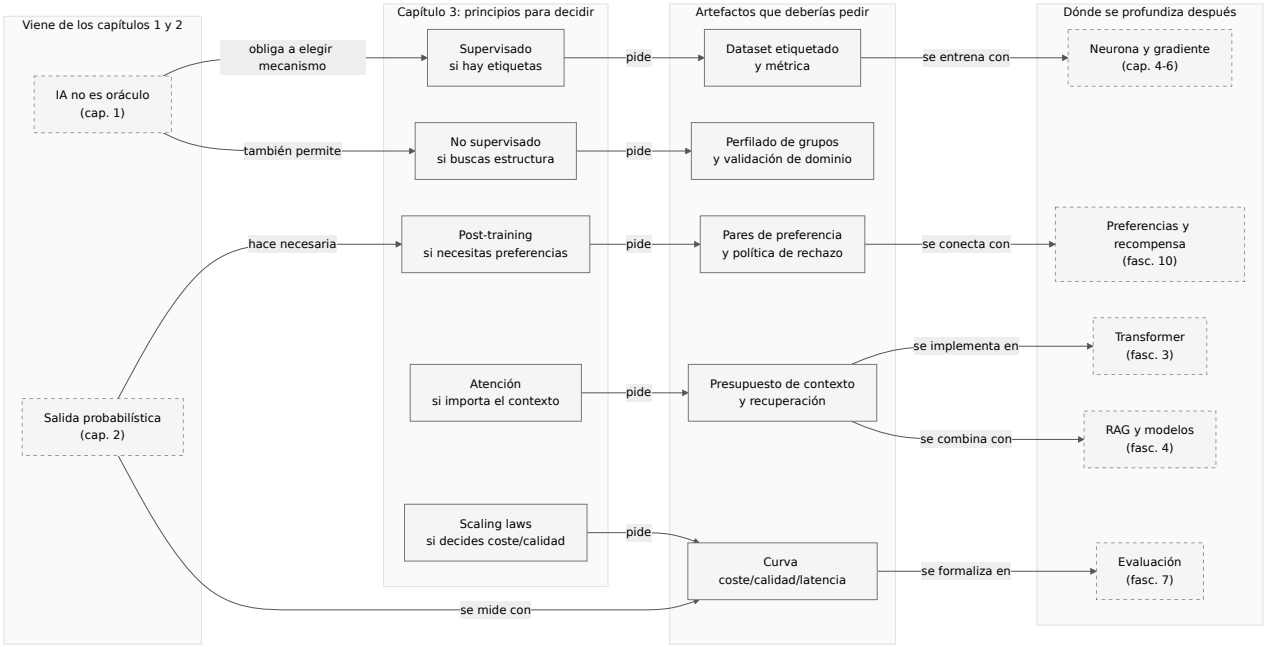
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Creer que el aprendizaje no supervisado no necesita validación	Que un modelo encuentre patrones no significa que esos patrones sean útiles, reales o accionables. Un *cluster* puede ser un artefacto estadístico sin significado de negocio.	Valida siempre los resultados no supervisados contra conocimiento del dominio. Un grupo de clientes debe tener sentido para el equipo de negocio, no solo para el algoritmo.
Asumir que más parámetros siempre es mejor	Un modelo más grande es más caro, más lento y no necesariamente mejor para tu caso de uso concreto. Las *scaling laws* predicen rendimiento agregado, no rendimiento en tu tarea específica.	Evalúa el modelo en tus datos y tu tarea. No delegues la decisión en el número de parámetros.
Responder "fine-tuning" antes de diagnosticar	Un dominio especializado puede fallar por muchas razones: falta contexto, datos obsoletos, formato de salida inestable, vocabulario técnico, criterios de abstención o evaluación pobre. Ajustar pesos no arregla documentos desactualizados ni permisos mal diseñados.	Decide por causa: RAG para conocimiento vivo, reglas o herramientas para acciones verificables, *fine-tuning* o LoRA para comportamiento repetido y estable, y evaluación antes de cambiar nada.
Ignorar el coste cuadrático de la atención	Duplicar la longitud del contexto multiplica por cuatro el coste computacional. Diseñar sin tener esto en cuenta lleva a sistemas lentos y caros en producción.	Mide el coste real con tu volumen de datos y tu latencia objetivo. Considera estrategias de ventana deslizante, resumen previo o recuperación selectiva.

Cómo encaja todo

Este mapa se lee de izquierda a derecha. Hereda de los dos capítulos anteriores dos ideas: la IA no es una mente y sus salidas son probabilísticas. El capítulo 3 añade el primer vocabulario de diseño: qué tipo de aprendizaje o arquitectura estás invocando cuando dices «vamos a usar IA».

La decisión que enseña es práctica: no todos los problemas piden el mismo principio. Un caso con etiquetas pide supervisado; uno sin etiquetas pide exploración; uno con preferencias pide post-training; uno con contexto largo pide entender atención; uno limitado por coste pide mirar escala y evals. Ese mapa reaparece en casi todo lo que viene después.


Viene de los capítulos 1 y 2
IA no es oráculo
(cap. 1)
Salida probabilística
(cap. 2)
Capítulo 3: principios para decidir
obliga a elegir
mecanismo
también permite
hace necesaria
Supervisado
si hay etiquetas
No supervisado
si buscas estructura
Post-training
si necesitas preferencias
Atención
si importa el contexto
Scaling laws
si decides coste/calidad
se mide con
Artefactos que deberías pedir
pide
pide
pide
pide
pide
Dataset etiquetado
y métrica
Perfilado de grupos
y validación de dominio
Pares de preferencia
y política de rechazo
Presupuesto de contexto
y recuperación
Curva
coste/calidad/latencia
Dónde se profundiza después
se entrena con
se conecta con
se implementa en
se combina con
se formaliza en
Neurona y gradiente
(cap. 4-6)
Preferencias y
recompensa
(fasc. 10)
Transformer
(fasc. 3)
RAG y modelos
(fasc. 4)
Evaluación
(fasc. 7)

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Aprendizaje supervisado	Paradigma donde el modelo aprende a partir de ejemplos etiquetados con la respuesta correcta, ajustando sus parámetros para minimizar el error de predicción.
Aprendizaje no supervisado	Paradigma donde el modelo encuentra patrones en datos sin etiquetar, sin una respuesta correcta predefinida contra la que comparar.
Aprendizaje auto-supervisado	Variante donde la etiqueta sale de los propios datos (por ejemplo, predecir la siguiente palabra), lo que permite entrenar con texto sin etiquetar a mano.
Pre-entrenamiento	Fase inicial masiva de entrenamiento no supervisado donde el modelo aprende patrones generales del lenguaje a partir de cantidades inmensas de texto.
Post-training	Conjunto de técnicas aplicadas tras el pre-entrenamiento para enseñar al modelo preferencias de tono, formato y conducta, no solo hechos.
SFT	Ajuste supervisado del modelo con ejemplos de conversaciones de alta calidad escritas por personas para fijar formato y tono.
Aprendizaje por refuerzo	Paradigma donde un agente aprende por ensayo y error guiado por una señal de recompensa, no por ejemplos etiquetados.
RLHF	Técnica de post-entrenamiento donde la señal de aprendizaje proviene de personas que comparan y puntúan respuestas del modelo.
DPO	Técnica de optimización por preferencias que aprende de pares de respuestas elegidas y rechazadas sin entrenar un modelo de recompensa separado.
Fine-tuning	Proceso de tomar un modelo pre-entrenado y ajustarlo con ejemplos etiquetados de un dominio específico. Es aprendizaje supervisado sobre una base pre-entrenada.
Atención	Mecanismo que permite al modelo evaluar la relevancia de cada palabra de entrada para cada palabra de salida, procesando todas las relaciones simultáneamente.
Scaling law	Relación predecible entre los recursos invertidos en entrenamiento y el rendimiento del modelo.

Antes de pasar página

- [ ] ¿Puedo explicar la diferencia entre aprendizaje supervisado y no supervisado con un ejemplo concreto? (Si no, vuelve a las dos primeras secciones.)

- [ ] ¿Entiendo qué aporta el *post-training* que no aporta el pre-entrenamiento? (Si no, vuelve a «Post-training: enseñar preferencias».)
- [ ] ¿Sé qué hace el mecanismo de atención y por qué fue revolucionario? (Si no, vuelve a «Atención: mirar todo a la vez».)
- [ ] ¿Puedo explicar por qué las *scaling laws* son importantes para decidir qué modelo usar? (Si no, vuelve a «Scaling laws» y a «En el día a día».)
- [ ] ¿Entiendo por qué más parámetros no siempre es la respuesta correcta? (Si no, vuelve a «Dónde solía tropezar yo», error «Asumir que más parámetros siempre es mejor».)
- [ ] ¿Sé justificar qué principio domina un caso propio y cómo lo comprobaría? (Si no, vuelve a «Cómo funciona por dentro».)

En resumen

IDEA FUERZA	DETALLE
La IA moderna se apoya en cinco principios.	Aprendizaje supervisado, no supervisado, post-training, atención y *scaling laws*. Entenderlos es tener el mapa antes de adentrarse en el territorio.
El aprendizaje supervisado entrena con ejemplos etiquetados; el no supervisado encuentra patrones sin etiquetas.	El primero es la base del *fine-tuning*; el segundo, del pre-entrenamiento de los LLMs.
El mecanismo de atención permite procesar todas las palabras a la vez.	Es la innovación arquitectónica que hizo posibles los modelos de lenguaje modernos. Sin ella, no hay Transformer.
Las *scaling laws* demuestran que el rendimiento mejora de forma predecible con la escala.	Pero también advierten de que la mejora tiene un coste creciente. Más grande no siempre es la respuesta correcta para tu caso.

Para saber más

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. arXiv:2203.15556. https://doi.org/10.48550/arXiv.2203.15556

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

Ouyang, L. y otros (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems 35*, 27730-27744. https://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C. D. y Finn, C. (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

Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0

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

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

---

Notas

1. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. Los capítulos 18 a 21 cubren en profundidad los paradigmas de aprendizaje supervisado, no supervisado y por refuerzo. ↩

2. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org. El capítulo 14 aborda los autoencoders y las representaciones aprendidas sin etiquetas humanas, fundamentos del pre-entrenamiento moderno. ↩

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://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html ↩

4. Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C. D. y Finn, C. (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 ↩

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://papers.nips.cc/paper/7181-attention-is-all-you-need ↩

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

7. La idea de las leyes de escala procede de Kaplan, J. y otros (2020), arXiv:2001.08361. La forma con un término para los parámetros y otro para los datos, más una pérdida irreducible, sigue a Hoffmann, J. y otros (2022), arXiv:2203.15556, el trabajo conocido como Chinchilla. Aquí se escribe L_∞ donde el artículo original usa E. ↩

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

Capítulo 04: La neurona artificial: el átomo del aprendizaje

Entrando en el tema

Has oído hablar de modelos con miles de millones de parámetros. De redes que escriben código, generan imágenes y mantienen conversaciones. Es fácil imaginar que dentro de todo eso hay algo increíblemente complejo, inabarcable.

Y lo hay. Pero también es verdad que todo empieza con una pieza que cabe en tres líneas de código.

Esa pieza es la neurona artificial. No es una metáfora biológica vaga: es una función matemática concreta, con entradas, pesos, un sesgo y una salida. Si entiendes una neurona, entiendes el átomo del aprendizaje profundo. El resto (capas, redes, arquitecturas, Transformers) es composición.

Este capítulo es el primero con fórmulas. No te asustes. Vamos a desmenuzar cada símbolo, a ponerle números concretos y a escribir el código que lo implementa. Si al final entiendes `y = max(0, w₁x₁ + w₂x₂ + b)`, has entendido lo esencial.

Qué es una neurona artificial

Una neurona artificial es una **función matemática** que recibe números, los multiplica por pesos, suma un sesgo y aplica una función de activación.[1] Punto. No hay biología. Hay aritmética.

Lo de «neurona» es un nombre prestado de la biología, pero la semejanza es superficial. Una neurona biológica es una célula con dendritas, axones y neurotransmisores. Una neurona artificial es una operación de multiplicación, suma y comparación. El nombre ayuda a visualizar, pero no te lleves a engaño: estás ante álgebra lineal, no ante neurociencia.

La capacidad del sistema no está en una neurona individual. Está en los **miles de millones de parámetros** (pesos, sesgos, matrices de proyección, normalizaciones y otras piezas aprendidas) ajustados durante el entrenamiento. Una neurona sola no aprende nada útil. Muchas operaciones paramétricas organizadas en una arquitectura moderna pueden generar código, traducir idiomas y responder preguntas.

La fórmula

En 1943, Warren McCulloch y Walter Pitts publicaron el primer modelo matemático de una neurona artificial.[2] Su neurona era binaria (encendida o apagada) y no aprendía: los pesos eran fijos. Quince años después, Frank Rosenblatt dio el paso decisivo: el **perceptrón**, una neurona cuyos pesos se ajustaban automáticamente a partir de ejemplos.[3] Era la primera neurona que aprendía.

Hoy, ochenta años después, esta idea sigue viva dentro de los modelos modernos: tomar números de entrada, multiplicarlos por parámetros aprendidos, combinarlos y aplicar no linealidades. Pero un LLM no es simplemente una pila gigante de perceptrones: añade embeddings, atención, normalización, conexiones residuales y bloques MLP. La fórmula de la neurona no explica todo el Transformer, pero sí la operación mínima que veremos repetida dentro de sus bloques:

$$y = f\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
y	Salida de la neurona	0,76
f	Función de activación	ReLU (la más común)
n	Número de entradas	3
x_i	Entrada i (un número)	$x_1 = 0{,}5$, $x_2 = 0{,}3$, $x_3 = 0{,}8$
w_i	Peso de la conexión i	$w_1 = 0{,}2$, $w_2 = -0{,}4$, $w_3 = 0{,}7$
b	Sesgo (bias)	0,1

En palabras: multiplicas cada entrada por su peso, sumas todo, añades el sesgo, y pasas el resultado por la función de activación. Eso es todo.

En ingeniería se suele escribir la misma idea en forma vectorial:

$$z = \mathbf{w}^T \mathbf{x} + b$$

$$y = f(z)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\mathbf{x}$	Vector de entradas	$[0{,}5, 0{,}3, 0{,}8]$
$\mathbf{w}$	Vector de pesos	$[0{,}2, -0{,}4, 0{,}7]$
$\mathbf{w}^T\mathbf{x}$	Producto escalar entre pesos y entradas	$0{,}54$
z	Suma ponderada antes de activar	$0{,}64$
y	Salida final de la neurona	$0{,}64$ con ReLU

Esta forma vectorial te obliga a mirar algo que en producción rompe más cosas de las que parece: **las dimensiones**. Si $\mathbf{x}$ tiene 3 valores, $\mathbf{w}$ debe tener 3 pesos. Si una entrada tiene cuatro columnas y el modelo espera tres, no tienes un problema filosófico: tienes un contrato de datos roto.

Con los números del ejemplo:

$$\text{suma} = (0{,}5 \times 0{,}2) + (0{,}3 \times -0{,}4) + (0{,}8 \times 0{,}7) + 0{,}1$$

$$\text{suma} = 0{,}10 + (-0{,}12) + 0{,}56 + 0{,}1 = 0{,}64$$

Si usamos ReLU como activación: $y = \max(0, 0{,}64) = 0{,}64$.

Si usamos sigmoide: $y = \frac{1}{1+e^{-0{,}64}} \approx 0{,}65$.

La diferencia entre 0,64 y 0,65 parece pequeña, pero la elección de la función de activación determina qué puede aprender la red. Lo veremos a continuación.

Cómo funciona por dentro

En código, una neurona es literalmente esto:

```
function neurona(inputs, weights, bias) {
  const sum = inputs.reduce((acc, x, i) => acc + x * weights[i], 0);
  return activation(sum + bias);
}

// Ejemplo: neurona con 3 entradas
const entradas = [0.5, 0.3, 0.8];
const pesos    = [0.2, -0.4, 0.7];
const sesgo    = 0.1;

neurona(entradas, pesos, sesgo);
// sum = 0.5*0.2 + 0.3*(-0.4) + 0.8*0.7 + 0.1 = 0.64
// output = activation(0.64)
```

Observa tres cosas:

Una neurona es una función pura. Dadas las mismas entradas, pesos y sesgo, siempre produce la misma salida. El no determinismo del que hablamos en el capítulo 2 no está en la neurona individual: está en el muestreo que ocurre muchas capas después, al final del modelo.

Los pesos pueden ser negativos. Fíjate en $w_2 = -0{,}4$. Un peso negativo significa que la entrada correspondiente reduce la activación de la neurona. Es análogo a las sinapsis inhibitorias en biología, pero aquí es simplemente un número negativo multiplicando.

El sesgo desplaza todo. Sin sesgo ($b = 0$), nuestra neurona produciría $y = 0{,}54$ con ReLU. Con sesgo ($b = 0{,}1$), produce $y = 0{,}64$. El sesgo permite a la neurona activarse incluso cuando todas las entradas son cero. Es un parámetro más que se aprende durante el entrenamiento.

Funciones de activación

La función de activación f es lo que hace que una red neuronal no sea simplemente una combinación lineal de sus entradas. Sin ella, diez capas de neuronas serían equivalentes a una sola operación lineal: no podrían aprender patrones complejos. La no linealidad es la que permite que las capas profundas capturen relaciones que una sola capa no podría.[4]

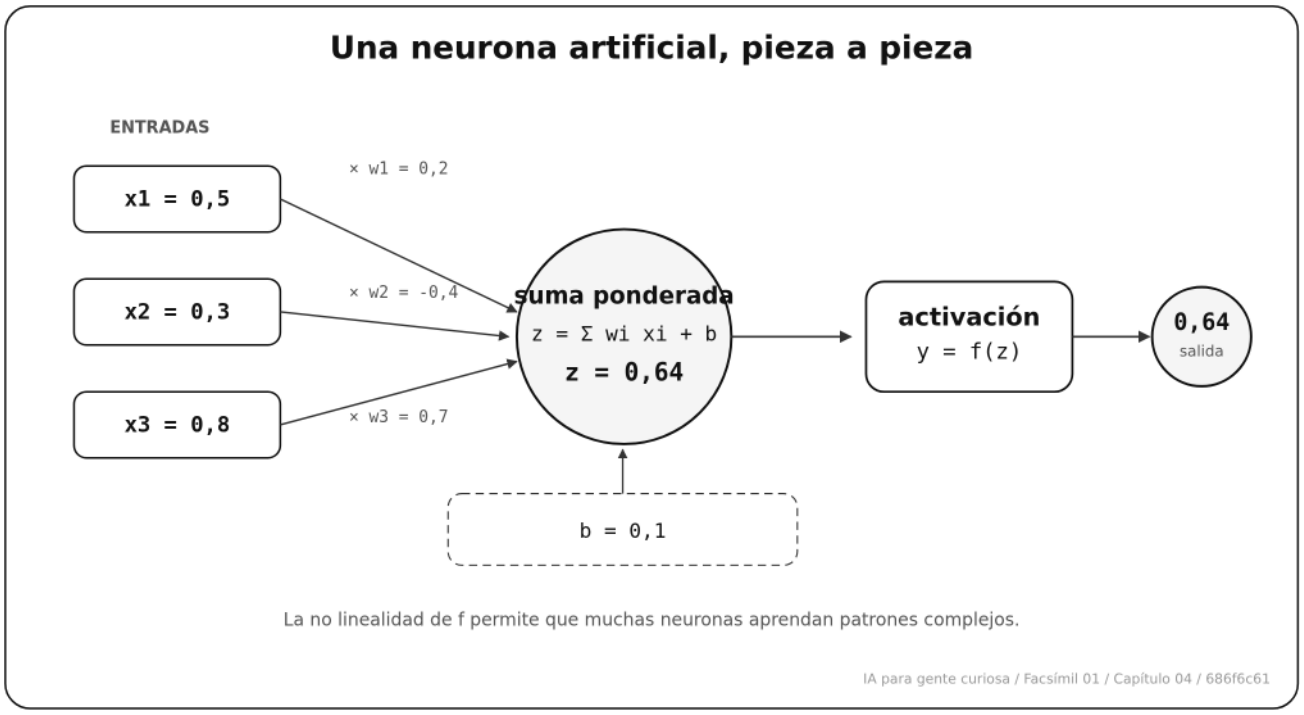
Conviene verlo, no solo afirmarlo. Si la activación fuera la identidad (es decir, sin ninguna no linealidad), encadenar dos neuronas daría:

$$y = w_2(w_1 x + b_1) + b_2 = (w_2 w_1)\, x + (w_2 b_1 + b_2)$$

que vuelve a tener la forma $w'x + b'$, con $w' = w_2 w_1$ y $b' = w_2 b_1 + b_2$: una sola neurona lineal. En palabras: por muchas capas que apiles, sin una función de activación no lineal todo colapsa en una única operación lineal. La activación es justo lo que rompe ese colapso y deja que cada

capa añada capacidad nueva.

FUNCIÓN	FÓRMULA	SALIDA	SE USA EN
ReLU	$\max(0, x)$	$[0, +\infty)$	Capas ocultas. La más usada. Simple, rápida y evita que los gradientes se desvanezcan.[5]
Leaky ReLU	$\max(0{,}01x,\ x)$	$(-\infty, +\infty)$	Variante de ReLU que deja pasar algo de señal en la zona negativa en vez de aplastarla a cero, evitando las «neuronas muertas».
GELU	$x \cdot \Phi(x)$	$\approx [-0{,}17, +\infty)$	La activación por defecto en los Transformers modernos (GPT, BERT). Suaviza la ReLU usando Φ, la distribución acumulada de la normal.[6]
Sigmoide	$\frac{1}{1+e^{-x}}$	$(0, 1)$	Capa de salida para clasificación binaria. Convierte cualquier número en una probabilidad entre 0 y 1.
Tanh	$\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$(-1, 1)$	RNNs y LSTMs. Similar a la sigmoide pero centrada en cero, lo que ayuda al entrenamiento.
Softmax	$\frac{e^{x_i}}{\sum_j e^{x_j}}$	Probabilidades que suman 1	Capa final de clasificación multiclase. Convierte un vector de puntuaciones en una distribución de probabilidad.



En el día a día

La neurona artificial no es un concepto académico que aparece una vez en el capítulo 1 y no vuelves a ver. Aparece en cada capa de cada red neuronal que uses. Cuando llamas a una API de OpenAI, miles de millones de estas operaciones se ejecutan en paralelo en las GPUs del proveedor. Cuando haces *fine-tuning* de un modelo, estás ajustando los pesos y sesgos de neuronas como esta.

A nivel de código, aunque uses bibliotecas como PyTorch o TensorFlow que abstraen la neurona individual, entender qué hay debajo te permite depurar problemas de entrenamiento, elegir funciones de activación adecuadas y entender por qué ciertas arquitecturas funcionan mejor para ciertos problemas.

Por ejemplo: si tu red no aprende, una de las primeras cosas que comprobarás es si tus funciones de activación son adecuadas. Una ReLU en la capa de salida de un problema de clasificación binaria no tiene sentido: necesitas una sigmoide. Si todas tus activaciones son cero, puede que tengas «neuronas muertas» por ReLU en un rango donde todas las entradas son negativas. La cura habitual es cambiar a Leaky ReLU o GELU, que dejan pasar algo de señal en la zona negativa en lugar de aplastarla a cero. Estos diagnósticos solo son posibles si entiendes qué hace cada pieza.[7]

Por qué debería importarte

La neurona artificial es uno de los conceptos más importantes del *deep learning*. No porque sea compleja (es lo más simple que verás en este facsímil), sino porque te enseña la operación elemental: pesos aprendidos, combinación lineal y no linealidad.

Una capa básica es un conjunto de neuronas que reciben las mismas entradas. Una red es una composición de capas. Un Transformer es una arquitectura más rica: combina atención, proyecciones, MLPs, normalizaciones y conexiones residuales. Pero en el fondo matemático seguimos viendo el mismo patrón: multiplicar matrices, sumar parámetros aprendidos, aplicar funciones y encadenar transformaciones.

Si entiendes `y = f(w₁x₁ + w₂x₂ + b)`, estás a una capa de distancia de entender una red neuronal completa. Y a unas pocas capas más de entender un Transformer.[8]

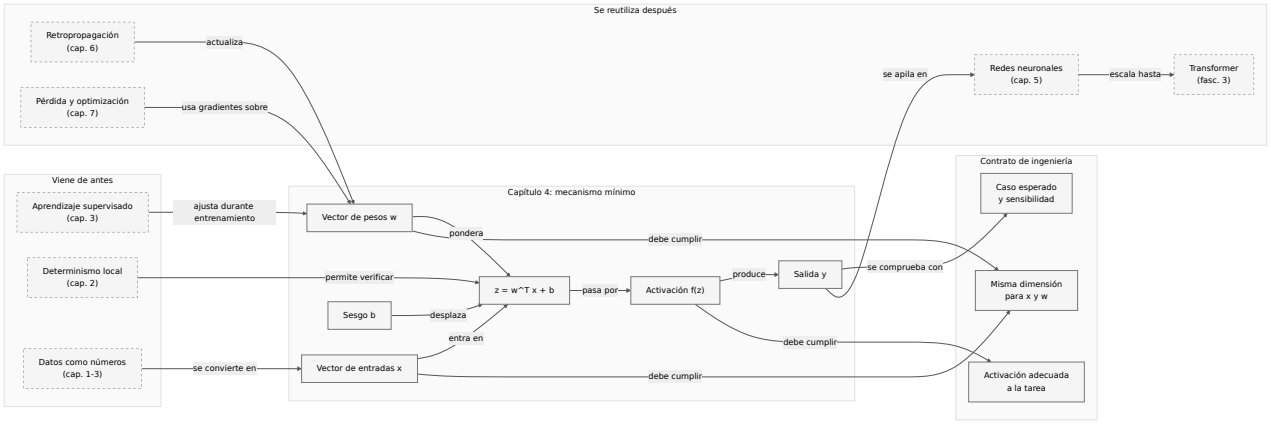
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Olvidar la función de activación	Sin activación no lineal, una red de cien capas es matemáticamente equivalente a una sola capa. No puede aprender patrones complejos.	Comprueba siempre que cada capa (excepto la de salida en algunos casos) tenga una función de activación no lineal.
Usar la activación equivocada para la tarea	Usar ReLU en la capa de salida para clasificación binaria produce valores sin sentido. Usar sigmoide en capas ocultas profundas causa desvanecimiento del gradiente.	ReLU para capas ocultas, sigmoide para salida binaria, softmax para salida multiclase, tanh para RNNs.
Ignorar el sesgo	Poner el sesgo a cero o eliminarlo limita lo que la neurona puede aprender. Sin sesgo, la neurona solo puede representar funciones que pasan por el origen.	Incluye siempre el sesgo. Es un parámetro más y se aprende igual que los pesos.
Pensar que cada neurona «entiende» algo concreto	Una neurona individual no codifica un concepto reconocible. La representación emerge de la combinación de muchas neuronas. Intentar interpretar neuronas individuales suele llevar a conclusiones erróneas.	No busques significado en una neurona. Búscalo en los patrones de activación del conjunto.

Cómo encaja todo

Este mapa se lee desde la fórmula hacia fuera. Los capítulos anteriores explican por qué necesitamos modelos que aprendan y por qué sus salidas finales pueden ser probabilísticas. Este capítulo baja al nivel mínimo: una neurona individual sí es determinista; lo que se aprende son sus pesos y su sesgo.

La decisión práctica que aparece aquí es nueva: antes de hablar de redes grandes, hay que respetar contratos pequeños. Dimensiones correctas, activación adecuada y cálculo reproducible. Si eso falla en una neurona, fallará multiplicado por millones en una red.


Se reutiliza después
Retropropagación (cap. 6)
actualiza
Pérdida y optimización (cap. 7)
usa gradientes sobre
se apila en
Redes neuronales (cap. 5)
escala hasta
Transformer (fasc. 3)
Viene de antes
Aprendizaje supervisado (cap. 3)
ajusta durante entrenamiento
Determinismo local (cap. 2)
permite verificar
Datos como números (cap. 1-3)
se convierte en
Capítulo 4: mecanismo mínimo
Vector de pesos w
pondera
Sesgo b
desplaza
entra en
Vector de entradas x
z = w^T x + b
pasa por
Activación f(z)
produce
Salida y
debe cumplir
debe cumplir
debe cumplir
se comprueba con
Contrato de ingeniería
Caso esperado y sensibilidad
Misma dimensión para x y w
Activación adecuada a la tarea

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Neurona artificial	Función matemática que recibe entradas numéricas, las multiplica por pesos, suma un sesgo y aplica una función de activación para producir una salida.
Peso	Número que multiplica una entrada de la neurona. Representa la importancia de esa entrada para la salida. Se ajusta durante el entrenamiento.
Sesgo (*bias*)	Valor que se suma a la suma ponderada para desplazar la salida. Permite a la neurona activarse incluso con entradas nulas.
Función de activación	Función no lineal aplicada a la suma ponderada más el sesgo. Sin ella, la red no podría aprender patrones complejos.
Producto escalar	Operación que multiplica dos vectores componente a componente y suma los resultados. En la neurona es $\mathbf{w}^T\mathbf{x}$.
Dimensión	Número de componentes de un vector. Entradas y pesos deben tener la misma dimensión para poder calcular la neurona.
Perceptrón	Primera neurona artificial capaz de aprender, propuesta por Rosenblatt en 1958, cuyos pesos se ajustan automáticamente a partir de ejemplos.
ReLU	Función de activación que devuelve max(0, x). La más usada en capas ocultas.
Leaky ReLU	Variante de ReLU que deja pasar algo de señal en la zona negativa en vez de aplastarla a cero, evitando las neuronas muertas.
GELU	Función de activación que suaviza la ReLU con la distribución acumulada de la normal. Es la activación por defecto en los Transformers modernos.
Sigmoide	Función de activación que comprime cualquier entrada a un valor entre 0 y 1. Útil para clasificación binaria.
Softmax	Función que convierte un vector de puntuaciones en una distribución de probabilidad que suma 1. Se usa en la capa final de clasificación multiclase.

Antes de pasar página

- [ ] ¿Puedo escribir de memoria la fórmula de una neurona artificial? (Si no, vuelve a «La fórmula».)
- [ ] ¿Puedo calcular a mano la salida para unas entradas, pesos y sesgo dados? (Si no, vuelve al ejemplo numérico en «La fórmula».)

- [ ] ¿Entiendo para qué sirve la función de activación y qué pasaría sin ella? (Si no, vuelve a «Funciones de activación».)
- [ ] ¿Sé qué función de activación usar en cada situación? (Si no, vuelve a la tabla en «Funciones de activación».)
- [ ] ¿Sé explicar por qué una neurona falla si el número de entradas y de pesos no coincide? (Si no, vuelve a «La fórmula».)

En resumen

IDEA FUERZA	DETALLE
Una neurona artificial es $y = f(\Sigma w_i x_i + b)$.	Entradas por pesos, más sesgo, pasado por una función de activación. Tres líneas de código.
La función de activación es lo que da poder a la red.	Sin ella, cien capas equivalen a una. Con ella, cada capa captura patrones cada vez más abstractos.
La inteligencia no está en una neurona: está en miles de millones de ellas.	Una neurona sola es una calculadora simple. Conectadas en capas y entrenadas con cantidades masivas de datos, producen los modelos que usas cada día.
Todo lo que viene se construye sobre esto.	Una capa es un conjunto de neuronas. Una red es una pila de capas. Un LLM es una arquitectura específica de red. Todo empieza aquí.

Para saber más

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). arXiv:1606.08415. https://arxiv.org/abs/1606.08415

LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539

McCulloch, W. S. y Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), 115-133. https://doi.org/10.1007/BF02478259

Nair, V. y Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. En *Proceedings of the 27th International Conference on Machine Learning* (pp. 807-814). https://www.cs.toronto.edu/~hinton/absps/reluICML.pdf

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386-408. https://doi.org/10.1037/h0042519

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

Notas

1. Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386-408. https://doi.org/10.1037/h0042519. Rosenblatt introdujo el perceptrón, la primera neurona artificial implementada en hardware, sentando las bases del aprendizaje supervisado. ↩

2. McCulloch, W. S. y Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), 115-133. https://doi.org/10.1007/BF02478259. Este artículo fundacional propuso que las neuronas podían modelarse como dispositivos lógicos de umbral, estableciendo el puente conceptual entre biología y computación. ↩

3. Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386-408. https://doi.org/10.1037/h0042519. Rosenblatt implementó el perceptrón en hardware (la máquina Mark I) y demostró que podía aprender a clasificar patrones visuales simples, inaugurando el campo del aprendizaje automático. ↩

4. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org. El capítulo 6 aborda en profundidad las funciones de activación y su papel en la capacidad representacional de las redes profundas. ↩

5. Nair, V. y Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. En *Proceedings of the 27th International Conference on Machine Learning* (pp. 807-814). https://www.cs.toronto.edu/~hinton/absps/reluICML.pdf. Nair e Hinton introdujeron las unidades lineales rectificadas (ReLU) demostrando que mejoraban significativamente el entrenamiento de redes profundas frente a las funciones sigmoidales. ↩

6. Hendrycks, D. y Gimpel, K. (2016). Gaussian error linear units (GELUs). arXiv:1606.08415. https://arxiv.org/abs/1606.08415. GELU es la activación adoptada por BERT, GPT y la mayoría de Transformers posteriores. ↩

7. 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. He y sus colegas demostraron que arquitecturas con conexiones residuales permiten entrenar redes de cientos de capas sin degradación del rendimiento, un avance que depende críticamente de una correcta elección de funciones de activación en cada bloque. ↩

8. LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539. Esta revisión de los tres pioneros del *deep learning* explica cómo las redes profundas con muchas capas aprenden representaciones jerárquicas, desde patrones simples en las primeras capas hasta conceptos abstractos en las profundas. ↩

Capítulo 05: Redes neuronales: capas, arquitectura y flujo

Entrando en el tema

En el capítulo anterior construiste una neurona. Una. Tres entradas, tres pesos, un sesgo y una salida. Funciona. Pero una neurona sola no llega muy lejos: puede trazar una línea recta entre dos grupos de puntos y poco más.

El salto empieza cuando conectas neuronas entre sí. Miles, millones, miles de millones de ellas. Organizadas en capas, cada una transformando los datos un poco más, pasando el resultado a la siguiente. Como una cadena de montaje donde cada estación añade una capa de abstracción.

Eso es una red neuronal. Y este capítulo explica cómo se organiza.

De la neurona a la capa

Una capa es un conjunto de neuronas que reciben las mismas entradas pero tienen sus propios pesos y sesgos.[1] Si la capa anterior tiene 3 neuronas y esta capa tiene 5, hay 3 × 5 = 15 conexiones, cada una con su propio peso. Más 5 sesgos, uno por neurona.

Cada neurona de la capa hace exactamente lo que viste en el capítulo 4: multiplica entradas por pesos, suma, suma el sesgo, aplica activación. La diferencia es que ahora sus entradas no son los datos originales (como «horas de estudio» o «temperatura»), sino las salidas de las neuronas de la capa anterior.

Y así, capa tras capa, los datos se transforman. Lo que empezó como una fila de números crudos termina como una predicción.

En una red real no solemos escribir neurona por neurona. Escribimos capas con matrices. Si la capa anterior produce un vector de tamaño d_{l-1} y la capa actual tiene d_l neuronas, la capa se calcula así:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = f^{(l)}(z^{(l)})$$

SÍMBOLO	FORMA	QUÉ SIGNIFICA
$a^{(l-1)}$	d_{l-1}	Activaciones que llegan desde la capa anterior. En la primera capa, es el vector de entrada x.
$W^{(l)}$	$d_l \times d_{l-1}$	Matriz de pesos. Una fila por neurona de la capa actual y una columna por entrada recibida.
$b^{(l)}$	d_l	Vector de sesgos. Uno por neurona de la capa actual.
$z^{(l)}$	d_l	Puntuaciones antes de aplicar activación.
$a^{(l)}$	d_l	Salida de la capa después de la activación.

La regla de ingeniería es simple: **la salida de una capa debe tener la misma dimensión que la entrada esperada por la siguiente**. Si una capa entrega 64 valores, la siguiente necesita una matriz de pesos con 64 columnas. Este contrato parece una minucia, pero es una de las primeras cosas que se comprueban cuando una arquitectura no arranca.

También podemos contar parámetros antes de entrenar:

$$\text{parámetros de la capa } l = d_l \cdot d_{l-1} + d_l$$

El primer término son pesos. El segundo, sesgos. Para una red con capas $[4, 32, 16, 3]$, el conteo sería:

CONEXIÓN	CÁLCULO	PARÁMETROS
Entrada 4 → Oculta 32	$32 \cdot 4 + 32$	160
Oculta 32 → Oculta 16	$16 \cdot 32 + 16$	528
Oculta 16 → Salida 3	$3 \cdot 16 + 3$	51
Total	160 + 528 + 51	**739**

Este número no mide inteligencia. Mide coste y capacidad potencial. Si tienes 200 filas de datos y una red con 3 millones de parámetros, probablemente no estás haciendo ciencia: estás dando al modelo demasiadas formas de memorizar.

Para situar la escala: esta red de juguete tiene 739 parámetros. Un modelo de lenguaje de los que usas a diario tiene entre 7.000 y 70.000 millones. La operación de fondo es exactamente la misma (multiplicar por pesos, sumar el sesgo, activar), repetida miles de millones de veces

y organizada en bloques especializados. Lo que cambia no es la pieza, sino cuántas hay y cómo se conectan.

La arquitectura de una red

La estructura más básica se llama **perceptrón multicapa** (*MLP*, *multilayer perceptron*). Tiene tres tipos de capas:

```
Capa de entrada → Capa(s) oculta(s) → Capa de salida
```

Capa de entrada. No hace cálculos. Simplemente recibe los datos y los distribuye a la primera capa oculta. Si tu *dataset* tiene 10 columnas, tu capa de entrada tiene 10 neuronas. Una por *feature*.

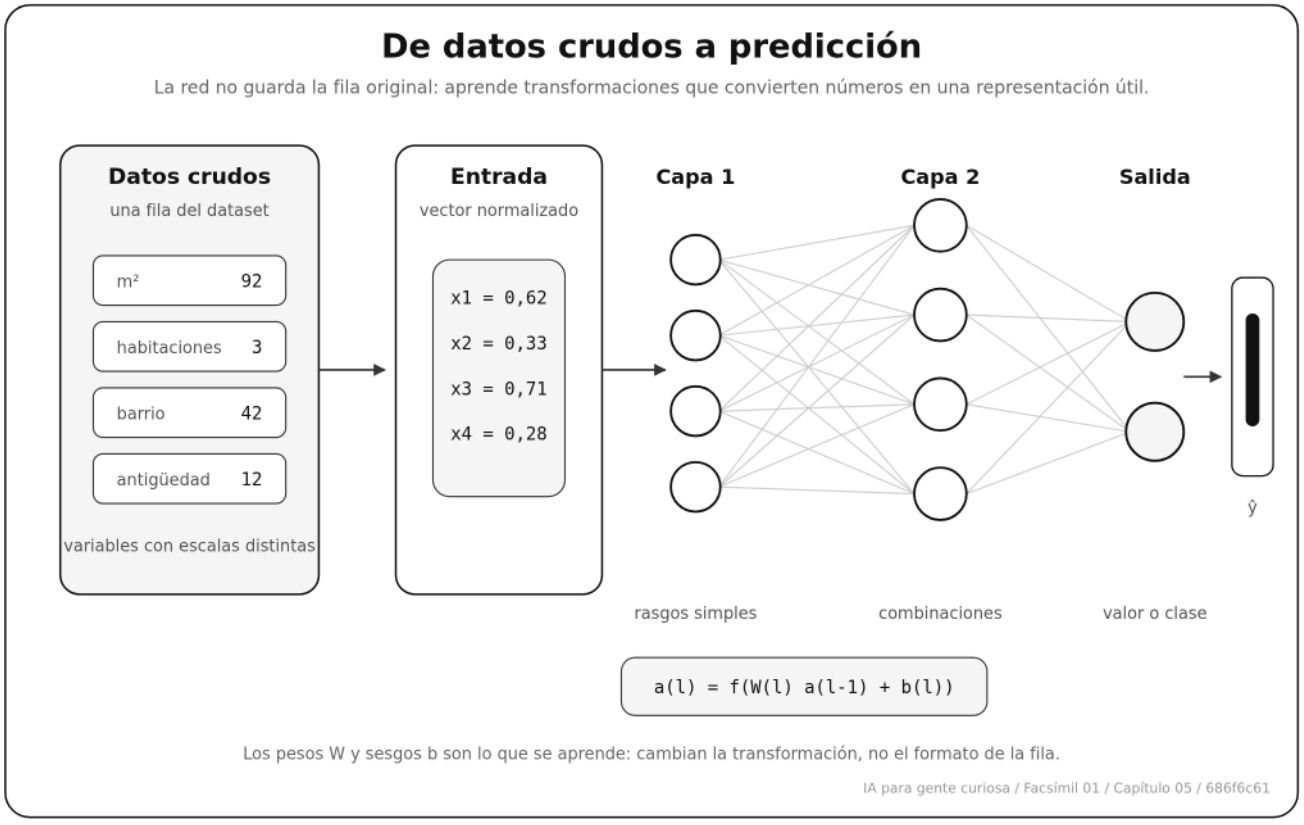
Capas ocultas. Aquí ocurre el aprendizaje. Una red puede tener una, diez o cien capas ocultas. Cada una transforma la representación de los datos en algo ligeramente más abstracto. Son «ocultas» porque no las ves desde fuera: solo ves lo que entra y lo que sale.

Capa de salida. Produce la predicción final. Si estás clasificando imágenes de dígitos (0-9), tu capa de salida tendrá 10 neuronas, una por clase, con softmax como activación. Si estás prediciendo el precio de una casa, tendrá 1 neurona, sin activación (o con activación lineal).

Para diseñarla con criterio, no empieces preguntando «¿cuántas capas pongo?». Empieza por el contrato del problema:

PREGUNTA DE DISEÑO	DECISIÓN TÉCNICA	EJEMPLO
¿Cuántas variables entran?	Tamaño de la capa de entrada.	24 columnas numéricas y categóricas codificadas → entrada de 24 dimensiones.
¿Qué debe salir?	Tamaño y activación de salida.	Binario → 1 salida + sigmoide. Multiclase excluyente → k salidas + softmax. Regresión → 1 salida lineal.
¿Cuánta no linealidad necesito?	Número de capas ocultas.	Datos tabulares sencillos: 1-3 capas. Imagen, audio o lenguaje: arquitecturas especializadas.
¿Cuánta capacidad puedo permitirme?	Ancho de cada capa y número total de parámetros.	64 neuronas pueden bastar para un clasificador interno; millones de parámetros exigen más datos y GPU.
¿Cómo voy a medir si funciona?	Métrica y conjunto de validación antes de entrenar.	Accuracy sola no basta si las clases están desbalanceadas; mira F1, recall, calibración o error absoluto según tarea.

Esta tabla evita una trampa común: diseñar una red como quien elige muebles. En ingeniería, arquitectura no es estética; es una hipótesis medible sobre datos, tarea, capacidad y coste.



Qué aprende cada capa

Una de las intuiciones más poderosas del *deep learning* es que las capas aprenden representaciones jerárquicas. No es una metáfora: es lo que se observa empíricamente al visualizar las activaciones de redes entrenadas.[2]

Capas tempranas. Detectan los patrones más elementales. En visión: bordes, colores, orientaciones. En texto: combinaciones de letras, prefijos y sufijos frecuentes. Son patrones que aparecen en cualquier imagen o texto, independientemente del contenido.

Capas intermedias. Combinan los patrones simples en conceptos más complejos. En visión: formas, texturas, partes de objetos (una rueda, un ojo, una esquina). En texto: frases hechas, relaciones sintácticas entre palabras.

Capas profundas. Abstracciones de alto nivel. En visión: objetos completos, rostros, escenas. En texto: significado semántico, intención del hablante, estructura argumental.

Esta jerarquía explica por qué el *transfer learning* funciona: las primeras capas de una red entrenada para reconocer imágenes sirven para casi cualquier tarea visual. Los bordes son bordes en todas partes. Son las capas profundas las que necesitan especializarse.[3]

Qué significa «deep» en *deep learning*

«*Deep*» significa «muchas capas». Una red con dos o tres capas es *shallow* (superficial). Una con decenas o cientos es *deep*. No hay un umbral oficial: la profundidad es una cuestión de grado.

Más capas permiten más abstracción, pero también traen problemas:

- **Más parámetros que entrenar**: más memoria, más computación, más datos necesarios.
- **Gradientes que se desvanecen**: en redes muy profundas, el gradiente puede hacerse tan pequeño en las primeras capas que dejan de aprender. Las funciones de activación como ReLU y arquitecturas como ResNet mitigan esto.[4]

La decisión de cuántas capas usar no es arbitraria: depende del problema, de los datos disponibles y del presupuesto computacional. Más capas no siempre es mejor. Pero cuando tienes datos masivos y tareas complejas (como procesar lenguaje natural), la profundidad marca la diferencia.

Lo que hace entrenable una red profunda

Apilar capas no basta. Si solo encadenas matrices y activaciones, una red muy profunda se entrena fatal: los gradientes se desvanecen o explotan, las activaciones se desestabilizan capa a capa y el modelo memoriza en vez de generalizar. Cuatro piezas, todas presentes dentro de un Transformer moderno, son las que arreglan esto.

Inicialización de pesos. Una red no arranca con los pesos a cero, sino con valores aleatorios pequeños y bien escalados. Si arrancan demasiado grandes, las activaciones explotan; si arrancan demasiado pequeños, se desvanecen ya en la primera pasada. Hay recetas para escalarlos según la activación: *He* para ReLU, *Xavier* (o *Glorot*) para tanh y sigmoide. No es un detalle menor: una mala inicialización puede impedir que una red profunda aprenda nada.

Normalización. Tras una capa, las activaciones pueden tomar cualquier escala, y eso desestabiliza el entrenamiento. La normalización las recentra y reescala para mantenerlas en un rango estable. Hay dos variantes habituales: *Batch Normalization* normaliza usando las estadísticas del lote, y *Layer Normalization* normaliza cada ejemplo por separado.[5] **LayerNorm es la que usan los Transformers**, y por tanto la que hay dentro de los LLMs.

Conexiones residuales. En lugar de pasar la salida de una capa tal cual, se le suma su propia entrada:

$$\text{salida} = x + f(x)$$

Es decir, la capa solo tiene que aprender la *diferencia* respecto a la entrada, no toda la transformación. Y, sobre todo, el `+ x` crea un atajo por el que el gradiente fluye directo hacia atrás sin desvanecerse. Esta idea simple es la que permitió entrenar redes de cientos de capas (*ResNet*) y está en cada bloque de cada Transformer.

Dropout y regularización. Son la respuesta directa al sobreajuste que aparece tantas veces en este capítulo. El *dropout* apaga al azar un porcentaje de neuronas en cada paso de entrenamiento, forzando a la red a no depender de ninguna en exclusiva y a aprender representaciones redundantes y robustas. El *weight decay* (regularización L2) penaliza los pesos grandes para que el modelo no se ajuste al ruido. Ninguna de las dos cambia la arquitectura: se activan al entrenar y se apagan al usar el modelo.

Cómo fluyen los datos

El viaje de los datos a través de una red se llama **propagación hacia delante** (*forward pass*). Es determinista: dados los mismos datos de entrada y los mismos pesos, la salida es siempre la misma.

En el *forward pass*, cada capa:

1. Recibe un vector de números de la capa anterior.
2. Multiplica cada entrada por su peso correspondiente.
3. Suma todo y añade el sesgo.
4. Aplica la función de activación.
5. Pasa el resultado a la capa siguiente.

Al final de la última capa, tienes una predicción. En el siguiente capítulo veremos cómo, a partir del error de esa predicción, la red ajusta todos sus pesos hacia atrás. Pero por ahora, quédate con esto: una red neuronal es simplemente una función matemática compuesta, donde la salida de cada capa es la entrada de la siguiente.

En código de producción suele aparecer una dimensión adicional: el **batch**. No procesas una sola fila, sino muchas a la vez. Si tienes 128 ejemplos y cada uno tiene 24 variables, la entrada puede tener forma 128×24. La capa no cambia su contrato interno: sigue esperando 24 columnas. Lo que cambia es que calcula 128 casos en paralelo.

OBJETO	FORMA TÍPICA	LECTURA
Entrada de un ejemplo	24	Una fila con 24 variables.
Batch de entrada	128×24	128 filas a la vez.
Pesos de la primera capa	64×24	64 neuronas, cada una mira las 24 variables.
Salida de la primera capa	128×64	128 filas, ahora representadas con 64 activaciones.
Salida final binaria	128×1	Una probabilidad por ejemplo.

Cuando una librería te devuelve un error de dimensiones, normalmente te está diciendo que una de estas piezas no encaja. Un ingeniero no lo resuelve probando números al azar: imprime formas, revisa contrato y comprueba dónde se rompió la cadena.

Presupuesto de una arquitectura

Antes de entrenar, puedes hacer una revisión fría de la red: formas, parámetros y memoria aproximada. No predice la calidad final, pero evita diseños absurdos.

Imagina un clasificador para priorizar tickets internos. Cada ticket ya está convertido a 48 variables numéricas: señales del usuario, categoría, antigüedad, canal, idioma y métricas históricas. La salida tiene 3 clases: baja, media y alta.

ARQUITECTURA	CAPAS	PARÁMETROS	LECTURA
MLP pequeño	$48 \to 32 \to 3$	1.667	Buen primer modelo. Barato, fácil de depurar y suficiente si las señales son fuertes.
MLP medio	$48 \to 128 \to 64 \to 3$	14.723	Más capacidad. Tiene sentido si hay bastantes ejemplos y relaciones no lineales.
MLP excesivo	$48 \to 1024 \to 1024 \to 3$	1.102.851	Puede funcionar, pero con pocos datos memorizará. También aumenta latencia, memoria y coste.

La comparación importante no es «qué red parece más profesional», sino qué red puedes justificar con datos. Si tienes 5.000 ejemplos limpios, empezar por un MLP pequeño o medio es razonable. Si tienes 200 ejemplos y clases desbalanceadas, el problema no se arregla con más capas: se arregla revisando datos, particiones, métrica y quizá usando un modelo más simple.

En el día a día

Cuando usas un LLM, no eliges cuántas capas tiene. El proveedor ya lo decidió por ti. Pero entender la arquitectura te permite tomar mejores decisiones:

Elegir el tamaño de modelo adecuado. Un modelo de 7B parámetros tiene menos capas (o capas más pequeñas) que uno de 70B. Más capas significan más capacidad de abstracción, pero también más latencia y más coste. Si tu tarea es clasificar el sentimiento de reseñas de producto, un modelo pequeño probablemente baste. Si necesitas análisis jurídico sobre documentos de cien páginas, necesitas profundidad, contexto y evaluación específica.

Hacer *fine-tuning* con criterio. Cuando ajustas un modelo pre-entrenado, la pregunta no es solo «qué capas entreno». En visión clásica sí era habitual congelar capas tempranas y entrenar capas finales. En LLMs modernos muchas veces se usan adaptadores, LoRA, QLoRA o entrenamiento parcial de módulos concretos, porque tocar todos los pesos es caro y puede degradar capacidades previas. La idea de fondo es la misma: decidir qué parte de la arquitectura permites modificar y cómo comprobarás que no empeora lo que ya funcionaba.

Depurar problemas de entrenamiento. Si tu *fine-tuning* no converge, entender la arquitectura te ayuda a diagnosticar: ¿tienes demasiadas capas para los datos que tienes? ¿El gradiente se está desvaneciendo en las capas tempranas? ¿La función de activación de la última capa es la adecuada para tu tarea?

Por qué debería importarte

La arquitectura de una red neuronal no es un detalle de implementación. Es la decisión de diseño más importante después de elegir los datos.

La diferencia entre un MLP de 3 capas y un Transformer de 96 capas no es solo de escala: es cualitativa. El MLP puede clasificar dígitos escritos a mano. El Transformer puede mantener una conversación larga, escribir código y trabajar con documentos extensos dentro de su ventana de contexto. La arquitectura determina qué patrones puede aprender y qué coste tendrá usarlos.

Y sin embargo, ambos se construyen con transformaciones paramétricas diferenciables: matrices, sesgos, no linealidades, normalizaciones y conexiones entre bloques. Lo que cambia es la organización. Por eso este capítulo importa: es el puente entre entender una operación elemental y entender por qué una arquitectura grande es ingeniería de formas, coste y flujo de información.

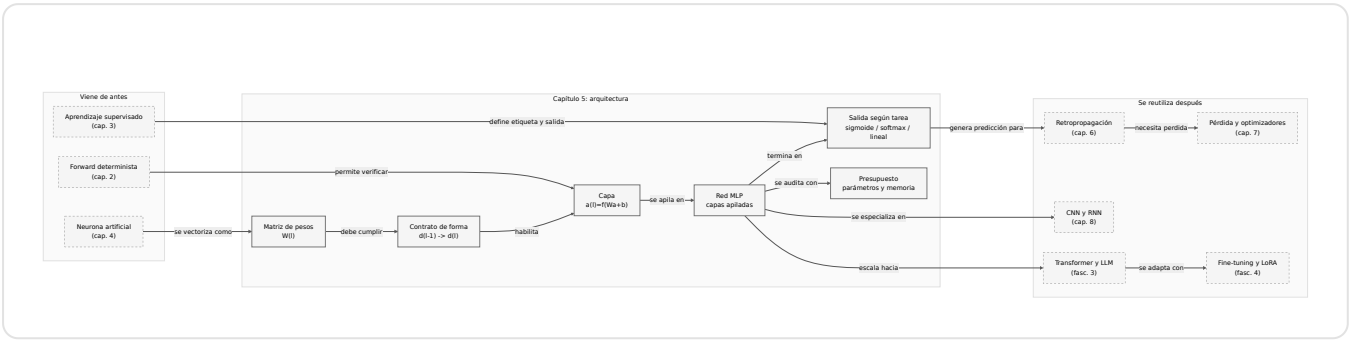
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Añadir capas sin criterio	Más capas no siempre mejoran el rendimiento. Pueden causar *overfitting* si no tienes suficientes datos, o *underfitting* si el gradiente se desvanece.	Empieza con una arquitectura sencilla y añade complejidad solo cuando los datos y la tarea lo justifiquen. Mide el rendimiento en validación, no en entrenamiento.
Ignorar la función de activación de la capa de salida	Usar ReLU en la salida de un clasificador binario produce valores sin sentido. Usar softmax para regresión fuerza una distribución de probabilidad donde no aplica.	Elige la activación según la tarea: sigmoide para binario, softmax para multiclase, lineal para regresión, ninguna para valores sin restricción.
Creer que las capas profundas «entienden» conceptos humanos	Una capa profunda puede activarse fuertemente ante «gatos», pero no «sabe» qué es un gato. Reconoce un patrón estadístico, no un concepto.	No interpretes las activaciones como comprensión. Son correlaciones aprendidas. La interpretabilidad requiere técnicas específicas (facsímil 7).
Olvidar que el *forward pass* es determinista	Si obtienes resultados distintos con los mismos datos y pesos, el problema no está en la red. Está en el preprocesamiento, en el *batch* o en el muestreo posterior.	Aísla cada componente. El *forward pass* de una red con pesos fijos es determinista. Si hay variabilidad, búscala fuera.
No mirar las formas de los tensores	Muchos errores no son conceptuales: una capa entrega 128 valores y la siguiente espera 64, o la salida tiene 10 clases y la etiqueta viene como binaria.	Antes de entrenar, escribe la secuencia de dimensiones y cuenta parámetros. Si no puedes hacerlo en papel, tampoco lo depurarás bien en código.
Confundir más parámetros con más rigor	Una red grande puede parecer más seria, pero si el conjunto de datos es pequeño solo aumenta la capacidad de memorizar ruido.	Compara parámetros con ejemplos disponibles, mira validación y mantén una arquitectura base sencilla como control.

Cómo encaja todo

Este capítulo es el puente entre una neurona aislada y una red entrenable. Hereda del capítulo 4 la operación mínima $y = f(w^T x + b)$, pero la convierte en una arquitectura: matrices, capas, formas, activaciones y salida. Esa arquitectura no aprende todavía; solo calcula. Aprenderá en el capítulo 6, cuando la pérdida viaje hacia atrás y modifique pesos.

La idea que conviene guardar es esta: una red neuronal es una función compuesta con contratos de forma. Si respetas esos contratos, puedes apilar capas, medir coste y elegir salidas coherentes con la tarea. Si no los respetas, el modelo no falla por misterioso: falla porque la ingeniería básica no encaja.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Red neuronal	Grafo dirigido de neuronas artificiales organizadas en capas que transforman datos secuencialmente.
Capa	Conjunto de neuronas que reciben las mismas entradas y producen salidas que alimentan la capa siguiente.
Capa oculta	Capa intermedia entre la entrada y la salida. Transforma los datos en representaciones progresivamente más abstractas.
MLP (*multilayer perceptron*)	Arquitectura donde cada neurona de una capa se conecta con todas las neuronas de la capa siguiente.
Deep learning	Entrenamiento de redes con muchas capas (decenas o cientos) que aprenden jerarquías de representaciones.
Forward pass	Recorrido de los datos desde la entrada hasta la salida. Es determinista: mismos pesos, misma salida.
Matriz de pesos	Conjunto de pesos de una capa escrito como matriz W. Su forma $d_l \times d_{l-1}$ define cuántas entradas recibe cada neurona y cuántas neuronas tiene la capa.
Ancho	Número de neuronas de una capa. Aumenta capacidad y coste.
Profundidad	Número de capas con parámetros. Permite composiciones más complejas, pero complica entrenamiento y depuración.
Contrato de forma	Requisito de que la salida de una capa tenga la dimensión que espera la siguiente. Es una validación básica antes de entrenar.
Presupuesto de parámetros	Conteo de pesos y sesgos de la arquitectura. Sirve para comparar capacidad, memoria aproximada y riesgo de sobreajuste.
Normalización (*LayerNorm*)	Recentrado y reescalado de las activaciones para mantenerlas en un rango estable. *LayerNorm* normaliza cada ejemplo por separado y es la variante que usan los Transformers.
Conexión residual	Atajo que suma la entrada a la salida de una capa ($x + f(x)$) para que el gradiente fluya hacia atrás sin desvanecerse.
Dropout	Apagado aleatorio de un porcentaje de neuronas en cada paso de entrenamiento para reducir el sobreajuste y forzar representaciones robustas.

Antes de pasar página

- [ ] ¿Puedo dibujar (mentalmente o en papel) la estructura de una red neuronal con capa de entrada, dos ocultas y una de salida? (Si no, vuelve al diagrama en «La arquitectura de una red».)
- [ ] ¿Puedo escribir la fórmula de una capa $a^{(l)} = f(W^{(l)}a^{(l-1)} + b^{(l)})$ y explicar la forma de W? (Si no, vuelve a «De la neurona a la capa».)
- [ ] ¿Entiendo qué aprende cada capa y por qué las capas tempranas son más genéricas? (Si no, vuelve a «Qué aprende cada capa».)
- [ ] ¿Sé qué significa «deep» en *deep learning* y qué problemas trae? (Si no, vuelve a «Qué significa deep».)
- [ ] ¿Puedo explicar el *forward pass*? (Si no, vuelve a «Cómo fluyen los datos».)
- [ ] ¿Sé defender una arquitectura calculando sus parámetros, formas y salida? (Si no, vuelve a «Presupuesto de una arquitectura».)

En resumen

IDEA FUERZA	DETALLE
Una red neuronal es una composición de capas.	Cada capa aplica una transformación $Wa + b$, una activación y entrega una nueva representación.
Las dimensiones son contrato, no decoración.	Si una capa produce 64 valores, la siguiente debe aceptar 64 entradas. Muchos errores reales empiezan ahí.
Los parámetros se pueden contar antes de entrenar.	$d_l \cdot d_{l-1} + d_l$ por capa. Ese conteo ayuda a razonar sobre memoria, datos necesarios y riesgo de sobreajuste.
Profundidad y ancho son decisiones de ingeniería.	Más capas o más neuronas pueden dar más capacidad, pero también más coste, más latencia y más superficie de fallo.
El capítulo 6 necesita este capítulo.	La retropropagación solo tiene sentido cuando sabes qué pesos, sesgos y capas debe actualizar.

Para saber más

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

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

Krizhevsky, A., Sutskever, I. y Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. En *Advances in Neural Information Processing Systems 25* (pp. 1097-1105). https://papers.nips.cc/paper/4824

LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539

Nielsen, M. (2015). *Neural networks and deep learning*. http://neuralnetworksanddeeplearning.com

Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0

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

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. y Polosukhin, I. (2017). Attention is all you need. En *Advances in Neural Information Processing Systems 30*. https://arxiv.org/abs/1706.03762

Notas

1. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org. El capítulo 6 aborda en profundidad las arquitecturas de redes *feedforward*, desde el perceptrón simple hasta las redes profundas modernas. ↩

2. LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539. Los autores describen cómo las redes profundas aprenden representaciones jerárquicas: desde bordes y texturas en las primeras capas hasta partes de objetos y conceptos completos en las capas profundas. ↩

3. Krizhevsky, A., Sutskever, I. y Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. En *Advances in Neural Information Processing Systems 25* (pp. 1097-1105). https://papers.nips.cc/paper/4824. AlexNet demostró que las características aprendidas por las capas tempranas de una CNN son transferibles entre tareas visuales, sentando las bases del *transfer learning* moderno. ↩

4. 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 permiten que el gradiente fluya directamente a través de las capas, resolviendo el problema del desvanecimiento en redes de más de cien capas. ↩

5. Ba, J. L., Kiros, J. R. y Hinton, G. E. (2016). Layer normalization. arXiv:1607.06450. https://arxiv.org/abs/1607.06450. LayerNorm normaliza sobre las características de cada ejemplo y es la variante usada en los Transformers, donde el tamaño de lote y la longitud de secuencia varían. ↩

6. TensorFlow Playground. https://playground.tensorflow.org. Una herramienta educativa que visualiza en tiempo real el entrenamiento de redes neuronales sobre conjuntos de datos sintéticos. ↩

Capítulo 06: Cómo aprende una red: retropropagación

Entrando en el tema

Imagina que estás en una montaña con niebla cerrada. No ves el valle. No ves el camino. Pero puedes sentir la pendiente bajo tus pies. Sabes, en cada punto, hacia dónde baja el terreno. Das un paso en esa dirección. Vuelves a tantear. Otro paso. Así, a ciegas pero con información local, acabas llegando al punto más bajo.

Eso es exactamente lo que hace una red neuronal cuando aprende. El «valle» es el mínimo de la función de pérdida (el punto donde el error es más pequeño). La «pendiente» es el gradiente. Y los «pasos» son las actualizaciones de los pesos.

El algoritmo que calcula esa pendiente se llama **retropropagación** (*backpropagation*).[1] Es el motor del aprendizaje profundo. Sin él, las redes de más de dos capas no podrían entrenarse. Con él, una red de cien capas ajusta todos sus parámetros en cada iteración.

Este capítulo es denso. Respira hondo. Vamos a desmenuzarlo pieza a pieza.

El bucle de entrenamiento

Entrenar una red neuronal es un bucle de cuatro pasos que se repite millones de veces:

```
1. Forward pass → 2. Calcular pérdida → 3. Backward pass → 4. Actualizar pesos → (repetir)
```

En pseudocódigo:

```
for epoch in range(num_epochs):
    for batch in dataloader:
        optimizer.zero_grad()                    # Reset antes de acumular
        prediction = model.forward(batch.input)  # 1. Forward
        loss = loss_fn(prediction, batch.target)  # 2. Pérdida
        loss.backward()                          # 3. Gradientes
        optimizer.step()                         # 4. Actualizar
```

Cuatro líneas de código. Miles de millones de ejecuciones. Así se entrenan los LLMs.

Antes de seguir, tres palabras de ese pseudocódigo que conviene tener muy claras, porque reaparecen en todo el *deep learning*:

- **Iteración** (o *step*). Una vuelta del bucle interior: un *forward*, una pérdida, un *backward* y una actualización de pesos. Es la unidad mínima de aprendizaje. Cada iteración mueve los parámetros un pasito.
- ***Mini-batch*** (lote). En cada iteración no se procesa un solo ejemplo ni todo el *dataset* de golpe, sino un grupo pequeño, típicamente de 16, 32 o 256 ejemplos. Se calcula el gradiente medio del lote y se actualiza una vez. ¿Por qué un término medio? Procesar el *dataset* entero en cada paso (descenso de gradiente *batch* completo) es carísimo y solo da un paso por pasada; procesar un ejemplo cada vez (modo *online*) es ruidoso y desaprovecha el paralelismo de la GPU. El *mini-batch* coge lo mejor de ambos: aprovecha el hardware y, además, el pequeño ruido del promedio ayuda a escapar de mínimos malos. A entrenar así, con lotes y un poco de azar, se le llama **descenso de gradiente estocástico** (*SGD*, *stochastic gradient descent*), y es la base de casi todos los optimizadores que verás en el capítulo 7.
- **Época** (*epoch*). Una pasada completa por todos los ejemplos del *dataset*. Si tienes 10.000 ejemplos y un *mini-batch* de 100, una época son 100 iteraciones. Entrenar de verdad suele requerir muchas épocas: el modelo ve los mismos datos una y otra vez, ajustando un poco los pesos cada vez.

Cada paso merece su propia explicación:

1. Forward pass. Los datos atraviesan la red de izquierda a derecha. Capa a capa, neurona a neurona. Al final, obtienes una predicción. Es determinista: mismos pesos, misma salida. Lo vimos en el capítulo 5.

2. Calcular la pérdida. Comparas la predicción con la respuesta real. ¿Cuánto se ha equivocado el modelo? La función de pérdida te da un número. Si la predicción es perfecta, la pérdida es cero. Cuanto mayor es el error, mayor es la pérdida.[2]

3. Backward pass. Aquí está la parte importante. El error viaja hacia atrás, de derecha a izquierda, desde la salida hasta la entrada. En cada neurona, en cada peso, el algoritmo calcula cuánto contribuyó ese parámetro al error total. Esto se llama **gradiente**. La herramienta matemática que lo permite es la **regla de la cadena** del cálculo.

4. Actualizar pesos. Conoces el gradiente. Sabes en qué dirección modificar cada peso para reducir el error. Aplicas un pequeño ajuste en esa dirección. La magnitud del ajuste la controla el *learning rate*: un paso demasiado grande y te pasas del mínimo; demasiado pequeño y tardas una eternidad en llegar.

La cadena completa: qué significa cada símbolo

Antes de lanzarnos a las derivadas, necesitamos ponernos de acuerdo en el vocabulario. Vamos a seguir el viaje de un solo dato a través de una neurona con activación sigmoide.[3] Es el ejemplo más simple posible, pero contiene toda la mecánica que se repite en redes de miles de millones de parámetros.

SÍMBOLO	NOMBRE	SIGNIFICADO	EN EL EJEMPLO
x	Entrada	El dato que llega a la neurona. Puede ser una *feature* del *dataset* o la salida de una capa anterior.	$x = 2$
w	Peso	Número aprendido que pondera la entrada. Si cambia, cambia la predicción.	$w = 0{,}40$
b	Sesgo	Ajuste aprendido independiente de la entrada. Permite desplazar la salida.	$b = -0{,}10$
z	Preactivación	Suma lineal antes de aplicar la activación. $z = wx + b$. Es la puntuación cruda.	$z = 0{,}40 \times 2 - 0{,}10 = 0{,}70$
σ	Sigmoide	Función que comprime z al intervalo $(0, 1)$. $\sigma(z) = \frac{1}{1+e^{-z}}$.	función, no un valor
a	Activación	Salida de la neurona tras aplicar la activación. $a = \sigma(z)$.	$a \approx 0{,}668$
y	Etiqueta real	La respuesta correcta del ejemplo de entrenamiento.	$y = 1$
E	Error (pérdida)	Medida numérica del fallo. Aquí usamos MSE: $E = \frac{1}{2}(a - y)^2$.	$E \approx 0{,}055$
η	*Learning rate*	Tamaño del paso al actualizar. La dirección la marca el gradiente.	$\eta = 0{,}1$

Dos aclaraciones importantes:

No confundas z y a. z es la puntuación cruda antes de la activación. Puede ser negativa, grande, pequeña. a es la salida ya transformada por la sigmoide, siempre entre 0 y 1. El error se calcula con a, no con z: comparamos la predicción final con la realidad.

No entrenamos contra z. La pérdida E compara a con y. Pero para ajustar w y b necesitamos saber cómo afectan a E. Y w y b afectan a z, no directamente a a. Por eso necesitamos la regla de la cadena: para propagar el error desde a hasta z, y desde z hasta w y b.

Forward pass: la predicción

Con los números del ejemplo, el *forward pass* es:

$$x = 2, \quad w = 0{,}40, \quad b = -0{,}10, \quad y = 1$$

$$z = w \cdot x + b = 0{,}40 \times 2 + (-0{,}10) = 0{,}70$$

$$a = \sigma(z) = \frac{1}{1 + e^{-0{,}70}} \approx 0{,}668$$

$$E = \frac{1}{2}(a - y)^2 = \frac{1}{2}(0{,}668 - 1)^2 \approx 0{,}055$$

La neurona predice 0,668. La respuesta correcta es 1. La predicción se queda corta. Queremos empujar la salida hacia arriba: necesitamos que a se acerque a 1.

Backward pass: cómo viaja el error hacia atrás

El objetivo del *backward pass* es calcular cuatro gradientes:

$$\frac{\partial E}{\partial a}, \quad \frac{\partial E}{\partial z}, \quad \frac{\partial E}{\partial w}, \quad \frac{\partial E}{\partial b}$$

Cada uno responde a una pregunta: «si cambio un poco esto, ¿cuánto cambia el error?». Se leen como derivadas parciales: la letra ∂ indica que solo estamos variando una variable y manteniendo el resto constantes.

Vamos paso a paso, de derecha a izquierda.

Paso 1: error respecto a la salida

$$\frac{\partial E}{\partial a}$$

Usamos la pérdida MSE: $E = \frac{1}{2}(a - y)^2$. Su derivada respecto a a es directa:

$$\frac{\partial E}{\partial a} = a - y = 0{,}668 - 1 = -0{,}332$$

Lectura: como la predicción es menor que la realidad, el gradiente es negativo. «Si subo a, el error E baja». El signo negativo nos dice la dirección correcta. La magnitud (0,332) nos dice cuánto impacto tiene.

Paso 2: error antes de la activación

$$\frac{\partial E}{\partial z}$$

Aquí entra la regla de la cadena. E depende de a, y a depende de z a través de la sigmoide. Por tanto:

$$\frac{\partial E}{\partial z} = \frac{\partial E}{\partial a} \cdot \sigma'(z)$$

La derivada de la sigmoide tiene una forma elegante: $\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) = a(1 - a)$.[4] En nuestro caso:

$$\sigma'(z) = a(1 - a) = 0{,}668 \times 0{,}332 \approx 0{,}222$$

$$\frac{\partial E}{\partial z} = -0{,}332 \times 0{,}222 \approx -0{,}074$$

Lectura: la sigmoide ha atenuado el gradiente. De -0,332 hemos pasado a -0,074. Es la sigmoide diciendo: «en esta zona de la curva, un cambio en z produce un cambio pequeño en a». Esto es el germen del problema del desvanecimiento del gradiente que mencionamos en el capítulo 5.

Paso 3: culpa asignada al peso

$$\frac{\partial E}{\partial w}$$

E depende de z, y $z = wx + b$. La derivada de z respecto a w es simplemente x. Regla de la cadena:

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial w} = \frac{\partial E}{\partial z} \cdot x$$

$$\frac{\partial E}{\partial w} \approx -0{,}074 \times 2 = -0{,}148$$

Lectura: el gradiente es negativo. Si aumentamos w, el error disminuye. Tiene sentido: la predicción era demasiado baja, y aumentar w aumenta z, lo que aumenta a, lo que reduce E. Además, el gradiente respecto a w es proporcional a x: las entradas más grandes reciben actualizaciones más grandes.

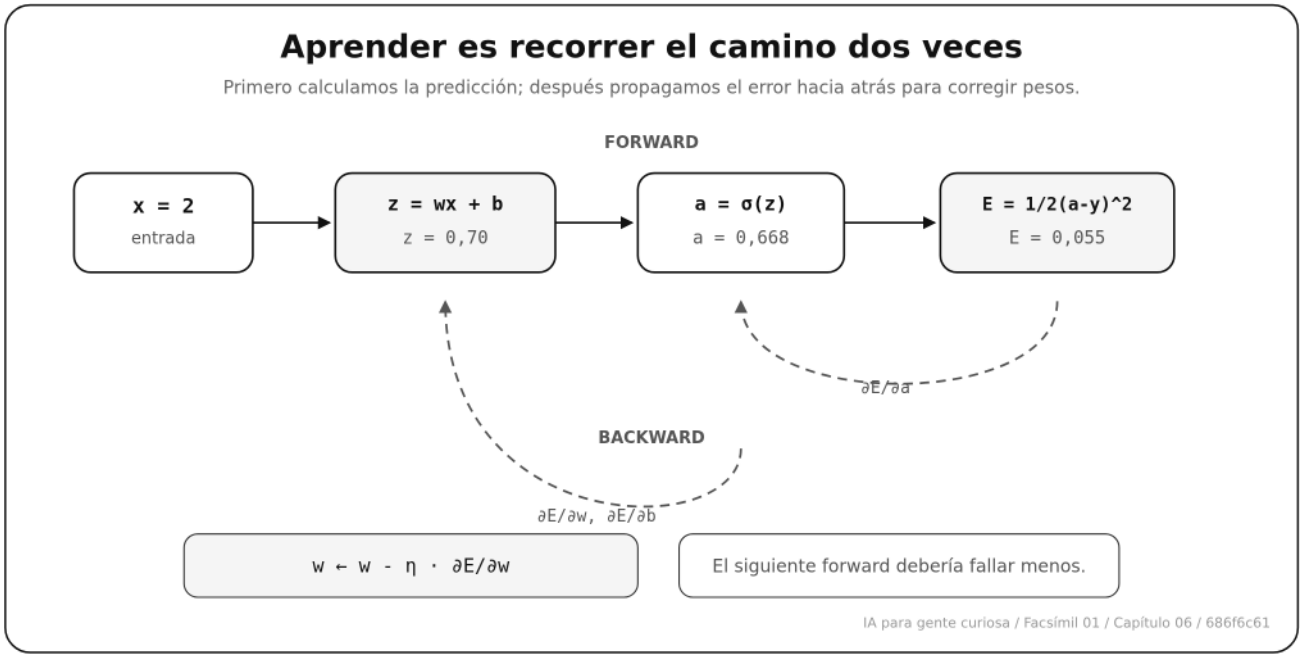
Paso 4: culpa asignada al sesgo

$$\frac{\partial E}{\partial b}$$

$z = wx + b$. La derivada de z respecto a b es 1. Por tanto:

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial b} = \frac{\partial E}{\partial z} \cdot 1 \approx -0{,}074$$

Lectura: el gradiente respecto al sesgo es igual al gradiente respecto a z. El sesgo se ajusta directamente con la señal de error que llega a la neurona, sin la mediación de la entrada.



La actualización final

Conocidos los gradientes, actualizamos los parámetros. La regla es siempre la misma:

$$w \leftarrow w - \eta \cdot \frac{\partial E}{\partial w}$$

$$b \leftarrow b - \eta \cdot \frac{\partial E}{\partial b}$$

El signo **menos** es crucial: nos movemos en la dirección que **reduce** la pérdida, no en la que la aumenta. Con $\eta = 0{,}1$:

$$w_{\text{nuevo}} = 0{,}40 - 0{,}1 \times (-0{,}148) \approx 0{,}415$$

$$b_{\text{nuevo}} = -0{,}10 - 0{,}1 \times (-0{,}074) \approx -0{,}093$$

Ambos parámetros han subido. La próxima vez que esta misma entrada pase por la neurona, z será mayor, a se acercará más a 1, y el error E bajará. Una iteración. Quedan miles de millones.

Resumen de gradientes

GRADIENTE	LECTURA	FÓRMULA	VALOR
$\partial E/\partial a$	Error respecto a la salida final	$a - y$ (con MSE)	-0,332
$\partial E/\partial z$	Error traducido al espacio preactivación	$\partial E/\partial a \cdot \sigma'(z)$	-0,074
$\partial E/\partial w$	Culpa asignada al peso	$\partial E/\partial z \cdot x$	-0,148
$\partial E/\partial b$	Culpa asignada al sesgo	$\partial E/\partial z$	-0,074

Observa el patrón: el gradiente fluye hacia atrás, atenuándose en cada paso (la sigmoide reduce -0,332 a -0,074). En redes profundas, esta atenuación se acumula capa tras capa. Es el problema del desvanecimiento del gradiente, y es la razón por la que las arquitecturas modernas usan ReLU en lugar de sigmoide en las capas ocultas.

En el día a día

En la práctica, nunca calculas gradientes a mano. PyTorch, TensorFlow y JAX lo hacen automáticamente mediante **diferenciación automática** (*autograd*).[5] Tú escribes el *forward pass* y la biblioteca construye el grafo de operaciones y calcula los gradientes por ti.

Lo que esas librerías hacen al ejecutar `loss.backward()` es exactamente la retropropagación, y tiene un nombre técnico: **diferenciación automática en modo inverso** (*reverse-mode autodiff*). «Modo inverso» quiere decir que se recorre el grafo de operaciones desde la salida hacia la entrada, justo como el *backward pass* de este capítulo. ¿Y por qué inverso, y no directo? Porque en una red hay millones de parámetros de entrada pero una sola pérdida escalar de salida. El modo inverso calcula en una única pasada cómo afecta esa salida a todos los parámetros a la vez; el modo directo necesitaría una pasada por cada parámetro. La regla práctica es: si tienes muchas entradas y pocas salidas (como aquí: millones de pesos, una pérdida), el modo inverso es el eficiente; si tuvieras pocas entradas y muchas salidas, convendría el directo. Por eso todo el *deep learning* usa el inverso.

Pero entender qué hay debajo cambia las decisiones de ingeniería que tomas cuando un entrenamiento no va bien. Estos son los casos típicos.

Depurar gradientes que se desvanecen o explotan. Cuando una red no aprende, lo primero que haces no es cambiar de modelo sino registrar la norma del gradiente capa por capa. Si observas que en las primeras capas es casi cero mientras en las últimas es razonable, decides que tienes **desvanecimiento**: el error se atenúa tanto al retroceder que las capas iniciales apenas se mueven. Lo viste en el ejemplo, la sigmoide convirtió $-0{,}332$ en $-0{,}074$, y esa pérdida se acumula capa tras capa. La decisión entonces es estructural: sustituyes la sigmoide por ReLU en las capas ocultas, añades **conexiones residuales** para que el `+ x` del capítulo 5 le dé al gradiente un atajo hacia atrás sin atenuarse, o metes normalización. El problema gemelo es la **explosión**: si la norma del gradiente crece sin control y la pérdida acaba en NaN, algo típico en redes recurrentes y secuencias largas, decides aplicar ***gradient clipping***. Es una sola línea que reescala el gradiente cuando su norma supera un umbral, y su valor real es que evita que un único lote raro arruine horas de entrenamiento. Importa porque la diferencia entre observar la norma y no observarla es la diferencia entre corregir la causa y cambiar parámetros al azar.

Elegir la función de activación correcta. Aquí la decisión es directa y se sigue de la matemática que ya calculaste. La sigmoide multiplica el gradiente por $a(1-a)$, un número que nunca pasa de 0,25 y que tiende a cero en los extremos, así que cada capa con sigmoide encoge la señal que viaja hacia atrás. ReLU no hace eso: su derivada es exactamente 1 para entradas positivas, de modo que el gradiente pasa intacto. Por eso, salvo motivo concreto, decides poner ReLU en las capas ocultas y reservar la sigmoide para la salida cuando necesitas una probabilidad. Importa porque esta única elección determina si una red profunda puede entrenarse o se queda congelada en sus primeras capas.

Entender por qué el *learning rate* importa. El gradiente solo te da dirección y sensibilidad; el tamaño del paso lo pones tú con η, y esa elección decide si el entrenamiento converge. Si lo pones demasiado grande, los pesos saltan de un lado a otro del mínimo y la pérdida oscila o diverge; si lo pones demasiado pequeño, el entrenamiento avanza tan despacio que desperdicias cómputo. Lo que observas para decidir es la curva de pérdida en las primeras iteraciones: si sube o serpentea, bajas η; si baja con una lentitud exasperante, lo subes. Importa porque la magnitud del gradiente, 0,148 en nuestro ejemplo, se multiplica por η para fijar el paso real, así que el mismo modelo con dos *learning rates* distintos puede converger o no aprender nada.

Cómo se depura en ingeniería

Cuando una red no aprende, una respuesta floja sería «prueba otro modelo». La respuesta de ingeniería es más aburrida y más útil: mirar señales. Antes de cambiar arquitectura, conviene saber si el *backward pass* está calculando algo coherente.

La comprobación clásica se llama **gradient check** o comprobación por diferencias finitas. Si quieres estimar el gradiente respecto a un peso w, perturbas ese peso un poco hacia arriba y hacia abajo, calculas la pérdida en ambos casos y comparas:

$$\frac{\partial E}{\partial w} \approx \frac{E(w+\epsilon) - E(w-\epsilon)}{2\epsilon}$$

Aquí ϵ es un número pequeño, por ejemplo 10^{-5}. Si el gradiente analítico que calculas con backpropagation y el gradiente numérico por diferencias finitas no se parecen, algo está mal: signo, derivada, activación, pérdida, broadcasting o acumulación.

No se usa este método para entrenar porque sería lentísimo: por cada parámetro tienes que ejecutar al menos dos *forward passes*. Se usa para **comprobar** implementaciones pequeñas, depurar capas nuevas y explicar por qué confiar ciegamente en `loss.backward()` no basta cuando tú has escrito parte de la función.

SEÑAL	QUÉ MIRAR	QUÉ PUEDE INDICAR
Pérdida antes/después	La pérdida debería bajar tras una actualización razonable.	Si sube siempre, revisa signo, *learning rate* o etiqueta.
Norma del gradiente	Tamaño global de los gradientes.	Casi cero: gradiente desvanecido. Enorme: gradiente explosivo.
Ratio de actualización	$	\Delta w
NaN o infinito	Valores no numéricos en pérdida o gradientes.	Activaciones saturadas, división por cero, *learning rate* excesivo o datos mal escalados.
Diferencias finitas	Gradiente numérico frente a gradiente analítico.	Detecta errores de implementación en derivadas o signo.

Esta tabla es pequeña, pero cambia la forma de trabajar. Dejas de mirar el entrenamiento como una caja negra y empiezas a tratarlo como un sistema observable.

Por qué debería importarte

La retropropagación no es un detalle de implementación. Es el algoritmo que hace posible el *deep learning*. Sin él, solo podríamos entrenar redes de una capa.

Cuando un modelo mejora durante entrenamiento, *post-training* o *fine-tuning*, ocurre una variante de este mismo proceso: se mide un error o una preferencia, se calcula una señal de gradiente y se ajustan parámetros. En una conversación normal con un LLM, en cambio, el

modelo no aprende por el simple hecho de recibir *feedback* del usuario; responde con pesos ya fijados. Esa distinción importa: una cosa es usar un modelo y otra es cambiarlo.

Entender la retropropagación es entender **cómo aprende una máquina**. No hay atajos. Pero una vez que lo entiendes, todo lo demás (arquitecturas, optimizadores, *fine-tuning*, RLHF) son variaciones sobre este mismo tema.

Dónde solía tropezar yo

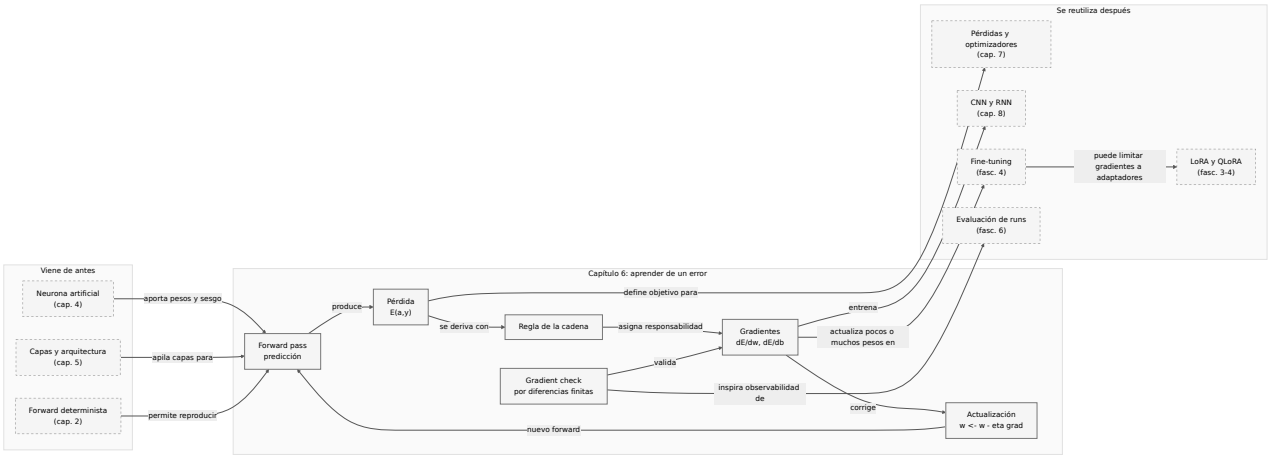
ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
No resetear los gradientes entre iteraciones	Los gradientes se acumulan por defecto en PyTorch. Si no llamas a `zero_grad()`, cada iteración suma su gradiente al anterior, produciendo actualizaciones incorrectas.	`optimizer.zero_grad()` al inicio de cada iteración, antes del `forward` y del `backward`. Así sabes que el gradiente que miras pertenece a ese lote.
Confundir la dirección del gradiente	El gradiente apunta en la dirección de **máximo crecimiento** del error. Por eso actualizamos con signo **menos**: para ir en dirección contraria, hacia el mínimo.	Grábate: `w = w - η * grad`. El signo menos no es negociable.
Interpretar el gradiente como magnitud absoluta de error	Un gradiente de -0,148 no significa que el peso esté «un 14,8 % mal». Significa que un cambio unitario en el peso cambia el error en -0,148 unidades. Es una sensibilidad, no un porcentaje de error.	Lee el gradiente como «si aumento esto en 1, el error cambia en X». No como «esto está X % equivocado».
Olvidar que el *forward pass* es determinista pero el resultado del entrenamiento no	Con los mismos datos iniciales y los mismos hiperparámetros, el entrenamiento puede converger a soluciones distintas. El paisaje de la función de pérdida tiene muchos mínimos locales.	No esperes reproducibilidad exacta en el entrenamiento. Fija semillas aleatorias si necesitas comparar experimentos.
Confiar en autograd sin comprobar nada	Que una biblioteca calcule gradientes no significa que tu pérdida, tu salida, tus etiquetas o tu capa personalizada estén bien planteadas.	En problemas pequeños, compara con diferencias finitas. En problemas reales, registra norma del gradiente, pérdida y ratio de actualización.

Cómo encaja todo

La retropropagación conecta dos mundos: el cálculo hacia delante, que produce una predicción, y la optimización, que decide cómo corregir pesos. En capítulos anteriores ya teníamos neuronas, capas y salida; aquí aparece la pregunta nueva: **qué parámetro tuvo**

qué responsabilidad en el error.

En ingeniería, este capítulo también introduce una costumbre que seguirá apareciendo: no basta con que algo entrene; hay que observarlo. Pérdida, gradientes, ratios de actualización y comprobaciones numéricas son señales de salud del sistema.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Retropropagación	Algoritmo que propaga el error desde la salida hacia las capas anteriores usando la regla de la cadena para calcular cómo ajustar cada parámetro.
Gradiente	Vector de derivadas parciales que indica cuánto cambia el error al modificar cada parámetro. Apunta en la dirección de máximo crecimiento.
Regla de la cadena	Propiedad del cálculo que permite derivar funciones compuestas: $\frac{\partial E}{\partial w} = \frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial w}$.
Learning rate (η)	Tamaño del paso en la actualización de pesos. La dirección la marca el gradiente; η controla la magnitud.
Pérdida (loss)	Medida numérica del error entre la predicción y la realidad. El objetivo del entrenamiento es minimizarla.
Derivada de la sigmoide	$\sigma'(z) = \sigma(z)(1-\sigma(z)) = a(1-a)$. Tiende a 0 en los extremos, causando desvanecimiento del gradiente.
Diferenciación automática	Técnica que calcula gradientes a partir del grafo de operaciones. PyTorch, TensorFlow y JAX la usan para implementar `backward()`.
Diferencias finitas	Aproximación numérica del gradiente perturbando un parámetro: $(E(w+\epsilon) - E(w-\epsilon))/(2\epsilon)$. Sirve para comprobar derivadas.
Norma del gradiente	Medida del tamaño global de los gradientes. Ayuda a detectar gradientes demasiado pequeños, demasiado grandes o inestables.
Iteración (*step*)	Una vuelta del bucle interior: un *forward*, una pérdida, un *backward* y una actualización de pesos sobre un lote.
Mini-batch (lote)	Grupo pequeño de ejemplos que se procesa junto en cada iteración para calcular un gradiente medio.
Época (*epoch*)	Una pasada completa por todos los ejemplos del *dataset*. Entrenar suele requerir muchas épocas.
SGD (descenso de gradiente estocástico)	Entrenar con lotes pequeños y algo de azar en lugar de todo el *dataset*. Es la base de casi todos los optimizadores.
Gradient clipping (recorte de gradiente)	Reescalar el gradiente cuando su norma supera un umbral, para evitar la explosión y los valores no numéricos.
Diferenciación automática en modo inverso (*reverse-mode autodiff*)	Recorrido del grafo de la salida a la entrada que calcula en una sola pasada el gradiente de la pérdida respecto a todos los parámetros. Es lo que hace `loss.backward()`.

TÉRMINO	DEFINICIÓN
Conexión residual	Atajo que suma la entrada a la salida de un bloque (`+ x`) y deja pasar el gradiente hacia atrás sin atenuarse.

Antes de pasar página

- [ ] ¿Puedo explicar los cuatro pasos del bucle de entrenamiento? (Si no, vuelve a «El bucle de entrenamiento».)
- [ ] ¿Entiendo qué significa cada símbolo de la tabla (x, w, b, z, a, y, E, η)? (Si no, vuelve a «La cadena completa».)
- [ ] ¿Puedo calcular a mano el *forward pass* y el *backward pass* para un ejemplo sencillo? (Si no, vuelve a «Forward pass» y «Backward pass».)
- [ ] ¿Sé por qué actualizamos con signo menos? (Si no, vuelve a «La actualización final».)
- [ ] ¿Puedo explicar qué es un gradient check por diferencias finitas? (Si no, vuelve a «Cómo se depura en ingeniería».)
- [ ] ¿Sé defender qué *learning rate* usaría y cómo comprobaría un gradiente? (Si no, vuelve a «Cómo se depura en ingeniería».)

En resumen

IDEA FUERZA	DETALLE
El entrenamiento es un bucle: predecir, medir el error, calcular gradientes, actualizar pesos.	Cuatro pasos que se repiten millones de veces. La retropropagación es el paso 3: cómo calcular los gradientes.
La regla de la cadena es la herramienta matemática que propaga el error hacia atrás.	$\partial E/\partial w = \partial E/\partial a \cdot \partial a/\partial z \cdot \partial z/\partial w$. El error fluye de derecha a izquierda, atenuándose en cada activación.
El gradiente dice «dirección y sensibilidad». El *learning rate* dice «tamaño del paso».	$w \leftarrow w - \eta \cdot \partial E/\partial w$. El signo menos va hacia el mínimo; η controla la velocidad.
Un *backward pass* sano se puede comprobar.	Diferencias finitas, norma del gradiente y pérdida antes/después ayudan a distinguir fallo matemático, fallo de escala y fallo de configuración.
Sin retropropagación no hay *deep learning*.	Las redes de más de dos capas dependen de este algoritmo. Todo lo demás (arquitecturas, optimizadores, *fine-tuning*) son variaciones sobre este mismo tema.

Para saber más

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

LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539

McCulloch, W. S. y Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), 115-133. https://doi.org/10.1007/BF02478259

Nielsen, M. (2015). *Neural networks and deep learning*. http://neuralnetworksanddeeplearning.com

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386-408. https://doi.org/10.1037/h0042519

Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0

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

Notas

1. Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0. Este artículo demostró que la retropropagación permitía entrenar redes con capas ocultas, resolviendo el problema que Minsky y Papert habían señalado en 1969 y abriendo la puerta al *deep learning* moderno. ↩

2. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org. El capítulo 6.2 aborda en detalle cómo las funciones de pérdida cuantifican el error y guían el aprendizaje mediante el gradiente. ↩

3. McCulloch, W. S. y Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), 115-133. https://doi.org/10.1007/BF02478259. Este artículo fundacional estableció el modelo de neurona como unidad de cómputo con entradas ponderadas y umbral de activación, el ancestro conceptual de la neurona artificial moderna. ↩

4. LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539. Los autores explican cómo la derivada de la sigmoide $\sigma'(z) = a(1-a)$ tiende a cero en los extremos, causando el desvanecimiento del gradiente en redes profundas, y cómo ReLU mitiga este problema. ↩

5. Nielsen, M. (2015). *Neural networks and deep learning*. http://neuralnetworksanddeeplearning.com. El capítulo 2 explica la retropropagación desde cero con ejemplos detallados y código Python, incluyendo una implementación completa sin bibliotecas de *deep learning*. ↩

Capítulo 07: Funciones de pérdida y optimizadores

Entrando en el tema

En el capítulo anterior construiste el motor: la retropropagación. Sabe calcular gradientes. Sabe en qué dirección ajustar cada peso para reducir el error. Pero le faltan dos piezas para funcionar: **qué error medir** y **cómo aplicar los ajustes**.

La primera pieza es la función de pérdida: la regla que decide qué cuenta como «error». La segunda es el optimizador: el algoritmo que decide cómo traducir los gradientes en actualizaciones concretas de los pesos.

Elegir mal cualquiera de las dos es como tener un coche con un motor excelente pero sin volante o sin frenos. El motor hace su trabajo, pero el coche no va a ninguna parte.

Funciones de pérdida: cómo medir el error

Una función de pérdida toma dos cosas (la predicción del modelo y la respuesta real) y devuelve un número: el error.[1] Cuanto mayor es el número, peor es la predicción. El objetivo del entrenamiento es hacer ese número lo más pequeño posible.

Hay tres funciones que cubren la inmensa mayoría de casos prácticos:

FUNCIÓN	SE USA EN	QUÉ MIDE
Cross-entropy	Clasificación, LLMs	Diferencia entre la distribución de probabilidad predicha y la real. Penaliza fuertemente las predicciones confiadas pero equivocadas.
MSE (error cuadrático medio)	Regresión	Media de los errores al cuadrado. Penaliza más los errores grandes que los pequeños (por el cuadrado).
Contrastive loss	*Embeddings*, búsqueda semántica	Acerca representaciones de ejemplos similares y aleja las de ejemplos diferentes.

Las fórmulas importan porque te dicen qué está castigando exactamente el entrenamiento:

PÉRDIDA	FÓRMULA SIMPLIFICADA	LECTURA
Binary cross-entropy	$-[y \log(p) + (1 - y) \log(1 - p)]$	Para una salida binaria p. Castiga mucho estar seguro y equivocarse.
Cross-entropy multiclase	$-\log(p_y)$	Para k clases. Solo mira la probabilidad asignada a la clase correcta.
MSE	$\frac{1}{n}\sum_i (\hat{y}_i - y_i)^2$	Para valores continuos. Los errores grandes pesan más por el cuadrado.
Triplet loss (contrastive)	$\max(0,\ d(a,p) - d(a,n) + m)$	Para representaciones. Acerca el ancla a un ejemplo similar y la aleja de uno distinto, con un margen m de separación.

Los símbolos que aparecen en esas fórmulas:

SÍMBOLO	SIGNIFICADO	EJEMPLO
y	Etiqueta real. En clasificación binaria vale 0 o 1	1 (la clase positiva)
p	Probabilidad que el modelo predice para la clase positiva	0,90
p_y	Probabilidad que el modelo asigna a la clase correcta (multiclase)	0,75 para la clase `alta`
n	Número de ejemplos sobre los que se promedia	128
$\hat{y}_i$	Predicción del modelo para el ejemplo i	19,4 (precio estimado)
y_i	Valor real del ejemplo i	21,0 (precio real)
a	Ancla: el ejemplo de referencia	una foto de un gato
$d(a,p)$	Distancia entre el ancla y un ejemplo similar (positivo)	pequeña si el modelo va bien
$d(a,n)$	Distancia entre el ancla y un ejemplo distinto (negativo)	grande si el modelo va bien
m	Margen: separación mínima exigida entre ambas distancias	0,2

En palabras: las tres primeras comparan lo que el modelo predice con lo que debería salir. La binaria y la multiclase trabajan con probabilidades y castigan asignar poca probabilidad a la respuesta correcta; la MSE trabaja con números y castiga la distancia al cuadrado. La *triplet*

loss no compara con una etiqueta: empuja a que el ancla quede más cerca de lo similar que de lo distinto, al menos por un margen m.

Ejemplo concreto: si la clase correcta es `alta` y el modelo le da probabilidad $p_y = 0{,}01$, la cross-entropy es $-\log(0{,}01) \approx 4{,}61$. Si le da $p_y = 0{,}90$, baja a $-\log(0{,}90) \approx 0{,}105$. No es una penalización lineal: fallar con mucha confianza duele mucho más. Por eso esta pérdida encaja tan bien con clasificación y predicción de siguiente token.

Cross-entropy es la reina indiscutible de la IA moderna. Cada vez que un LLM predice el siguiente token, está minimizando una *cross-entropy* entre la distribución que predice y la distribución real (que asigna probabilidad 1 al token correcto y 0 al resto).[2] Funciona especialmente bien porque su gradiente tiene una forma sorprendentemente simple. Para softmax seguido de cross-entropy, el gradiente respecto a la puntuación z_i de cada clase es exactamente la probabilidad predicha menos la real:

$$\frac{\partial E}{\partial z_i} = p_i - y_i$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
z_i	Puntuación (logit) que el modelo da a la clase i antes de softmax	3,1
p_i	Probabilidad que el modelo asigna a la clase i tras softmax	0,99
y_i	Valor real: 1 para la clase correcta, 0 para el resto	1

En palabras: el gradiente es el error directo, lo que el modelo predijo menos lo que debía predecir. Si el modelo asigna un 99 % de probabilidad al token correcto, $p_i - y_i$ es casi cero (ya lo hace bien). Si asigna un 1 %, el gradiente es grande (tiene mucho que aprender). Esa simplicidad, que el gradiente sea literalmente el error, es una de las razones de su éxito.

Esta pérdida tiene un nombre cuando se usa para evaluar modelos de lenguaje: la **perplejidad** (*perplexity*). Es la exponencial de la cross-entropy media:

$$\mathrm{PPL} = e^{H}, \qquad H = -\frac{1}{N}\sum_{i=1}^{N} \log p(\mathrm{token}_i)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
PPL	Perplejidad: la cross-entropy llevada a una escala intuitiva	12,5
H	Cross-entropy media sobre todos los tokens evaluados	2,53
N	Número de tokens evaluados	1.000.000
$p(\mathrm{token}_i)$	Probabilidad que el modelo asignó al token correcto en la posición i	0,21

Se interpreta como «entre cuántas opciones, de media, duda el modelo en cada token». Una perplejidad de 1 sería un modelo perfecto, que siempre acierta el siguiente token; una de 50 significa que, de media, está tan indeciso como si eligiera entre 50 opciones igual de probables. Cuando leas que un modelo de lenguaje tiene una perplejidad de X, estás leyendo su cross-entropy traducida a una escala más intuitiva.

Un matiz práctico: obligar al modelo a predecir exactamente 1 para la clase correcta y 0 para el resto puede volverlo demasiado confiado. El **label smoothing** suaviza un poco esas etiquetas, por ejemplo 0,9 para la correcta y un pequeño resto repartido entre las demás. Esto regulariza, mejora la calibración (que la confianza del modelo se parezca a su acierto real) y es habitual al entrenar Transformers.

MSE es la elección natural cuando predices valores numéricos: el precio de una casa, la temperatura de mañana, los ingresos del próximo trimestre. El cuadrado en la fórmula (`(predicción - realidad)²`) hace que los errores grandes pesen mucho más que los pequeños: equivocarse por 100 es cien veces peor que equivocarse por 10, pero la penalización es diez mil veces mayor. Esto empuja al modelo a evitar errores graves a toda costa.

Contrastive loss no se usa para predecir, sino para **representar**. Su objetivo es que dos imágenes del mismo objeto tengan vectores de *embedding* cercanos, y dos imágenes de objetos distintos los tengan lejanos. Es la base de los sistemas de búsqueda por similitud y del aprendizaje de representaciones.

El margen y las pérdidas de clasificación

Las tres pérdidas anteriores son las que más vas a ver en la IA moderna, pero esconden una idea más antigua y muy iluminadora que conviene conocer, porque organiza casi todo el aprendizaje supervisado clásico y reaparece, disfrazada, dentro de la *cross-entropy*. La idea es el **margen**, y nace de una pregunta sencilla: en una clasificación, ¿cuándo una predicción es buena, no solo correcta?

Para verlo, coloquemos la clasificación binaria en su forma más cruda. El modelo calcula una **puntuación** (en inglés *score*), un número que es positivo cuando apuesta por la clase positiva y negativo cuando apuesta por la negativa; cuanto más lejos de cero, más confiado está.[3] Si

codificamos la etiqueta real como $+1$ para la clase positiva y -1 para la negativa, el **margen** es el producto de la puntuación por la etiqueta:

$$\text{margen} = \text{puntuación} \cdot y$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
puntuación	Número que produce el modelo: positivo apuesta por la clase $+1$, negativo por la -1	2,3
y	Etiqueta real codificada como $+1$ o -1	$+1$
margen	Puntuación por etiqueta: positivo si acierta, negativo si falla; su tamaño es la confianza	2,3

En palabras: el margen junta en un solo número las dos cosas que importan de una predicción. Su **signo** dice si el modelo acierta (margen positivo, la puntuación va en el sentido de la etiqueta) o falla (margen negativo). Su **tamaño** dice con cuánta holgura, es decir, la confianza. Un margen de $+2{,}3$ es un acierto cómodo; uno de $+0{,}1$ es un acierto por los pelos, a punto de equivocarse; uno de $-1{,}5$ es un error claro y confiado, el peor caso. Geométricamente, si los pesos están normalizados, el margen es la distancia con signo del ejemplo a la **frontera de decisión**, la línea (o el plano) que separa una clase de la otra.

Con el margen en la mano, las pérdidas de clasificación se vuelven fáciles de comparar, porque todas son funciones de ese único número: castigan los márgenes negativos y premian los positivos, y solo se diferencian en *cómo* de severas son en la zona de transición. Las tres clásicas son:

PÉRDIDA	FÓRMULA (EN FUNCIÓN DEL MARGEN M)	IDEA
0-1 (*zero-one*)	$1[m \leq 0]$	Vale 1 si hay error, 0 si hay acierto. Es lo que de verdad quieres minimizar.
Bisagra (*hinge*, SVM)	$\max(0,\ 1 - m)$	Cero en cuanto el margen supera 1; si no, crece de forma lineal. Exige un acierto con holgura.
Logística	$\log_2(1 + e^{-m})$	Nunca llega a cero del todo: siempre empuja a aumentar el margen, cada vez con menos fuerza.

El problema de la pérdida **0-1**, la que mide lo que realmente nos importa (cuántas veces nos equivocamos), es que es inservible para entrenar con gradientes: es plana a ambos lados y da un escalón vertical en cero, así que su derivada es cero en casi todas partes y no señala ninguna dirección de mejora. El motor que montaste en el capítulo anterior se quedaría

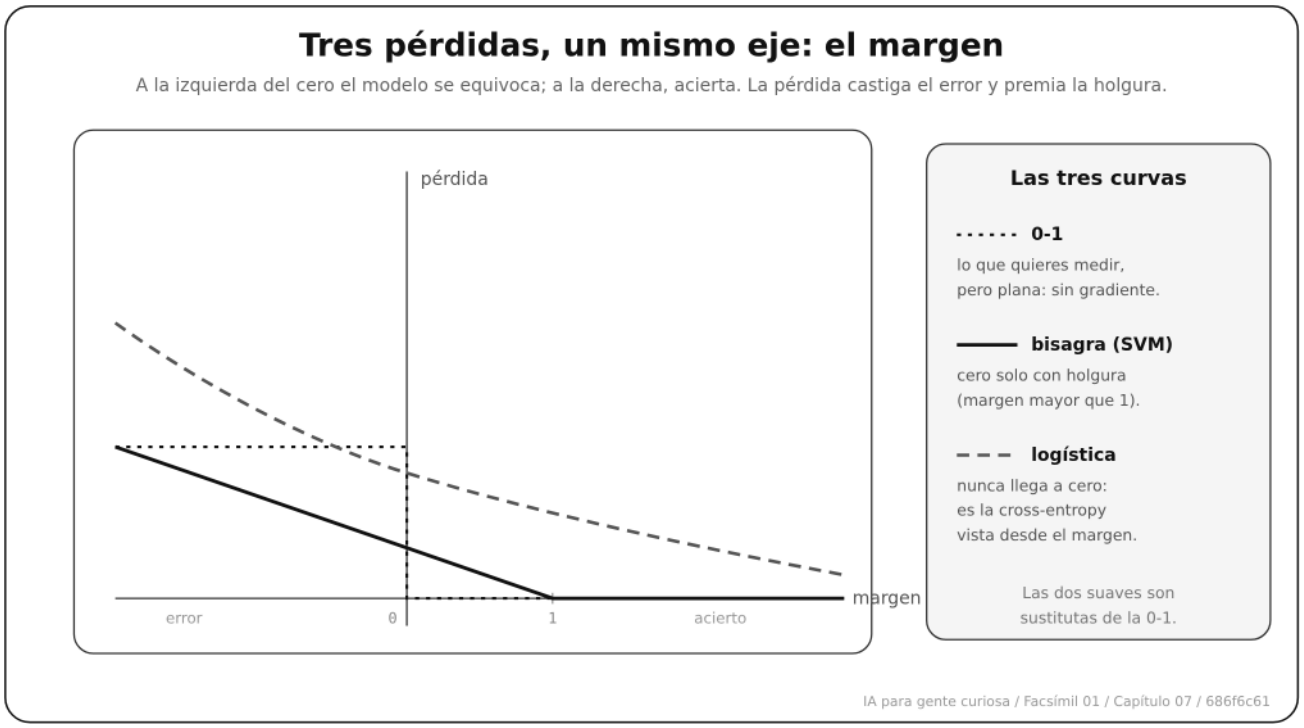
parado. Por eso, en la práctica, se sustituye por una pérdida **sustituta** (en inglés *surrogate*) que sí tenga pendiente: una que se parezca a la 0-1 pero baje de forma suave, de modo que el descenso de gradiente tenga siempre hacia dónde tirar.

$$\text{Bisagra}(m) = \max(0,\ 1 - m) \qquad \text{Logística}(m) = \log_2(1 + e^{-m})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
m	Margen del ejemplo (puntuación por etiqueta)	0,4
$\max(0,\ \cdot)$	Recorta a cero los valores negativos: si ya hay holgura, no penaliza	$\max(0,\ 0{,}6) = 0{,}6$
e^{-m}	Exponencial del margen cambiado de signo: enorme si el margen es muy negativo	$e^{-0{,}4} \approx 0{,}67$

En palabras: la pérdida **bisagra** dibuja una rampa que toca el suelo justo cuando el margen llega a 1, no a 0. Esa exigencia de pasar de largo, de acertar con un colchón de seguridad, es la idea central de las **máquinas de vectores de soporte** (SVM), que estudiarás como modelo en el capítulo de *machine learning* clásico: no basta con caer del lado correcto de la frontera, hay que hacerlo con margen.[4] La pérdida **logística**, en cambio, nunca se conforma: aunque el margen sea ya grande y positivo, sigue empujando un poquito más, con una fuerza que se desvanece poco a poco. Es la pérdida de la **regresión logística**, y no es una desconocida: es exactamente la *binary cross-entropy* de antes, reescrita en función del margen en lugar de la probabilidad. La reina de la IA moderna y este clásico de los años cincuenta son, por debajo, la misma pérdida.

Un ejemplo numérico fija las diferencias. Para un acierto justo, con margen $m = 0{,}4$: la pérdida 0-1 vale 0 (acertó, punto), la bisagra vale $\max(0,\ 1 - 0{,}4) = 0{,}6$ (acertó, pero sin la holgura que exige, así que sigue penalizando), y la logística vale $\log_2(1 + e^{-0{,}4}) \approx 0{,}76$ (también insiste en mejorar). Para un error confiado, con $m = -1{,}5$: la 0-1 vale 1, la bisagra $\max(0,\ 1 + 1{,}5) = 2{,}5$ y la logística $\log_2(1 + e^{1{,}5}) \approx 1{,}82$. Las dos sustitutas castigan mucho más el error confiado que el acierto ajustado, que es justo el comportamiento que se busca.



La lección que deja el margen va más allá de estas tres curvas. Casi cualquier pérdida de clasificación se entiende como una forma distinta de penalizar márgenes negativos, y elegir una u otra es elegir cuánto castigas la duda frente al error franco: la bisagra deja de insistir en cuanto hay holgura suficiente, lo que produce fronteras que dependen solo de los ejemplos difíciles, los más cercanos al borde; la logística nunca suelta, lo que da probabilidades mejor calibradas a cambio de no ignorar nunca a los ejemplos fáciles. No hay una mejor en abstracto; hay una mejor para lo que necesitas.

Optimizadores: cómo aplicar los ajustes

Una vez que la retropropagación ha calculado los gradientes, el optimizador decide **cómo** actualizar los pesos. La regla básica es siempre la misma:

$$w \leftarrow w - \eta \cdot \frac{\partial E}{\partial w}$$

Pero los optimizadores modernos añaden memoria estadística y adaptación a este proceso.[5]

OPTIMIZADOR	IDEA CLAVE	CUÁNDO USARLO
SGD	Gradiente descendente con mini-lotes. El más simple.	Cuando necesitas control total sobre el *learning rate* y el *momentum*.
Adam	*Learning rate* adaptativo por parámetro. Funciona bien sin ajuste.	El estándar para la mayoría de tareas. Buen punto de partida.
AdamW	Adam + *weight decay* correcto (desacoplado).	El estándar para entrenar LLMs y Transformers.
Muon	Ortogonaliza el *momentum* de las matrices de las capas ocultas. Más eficiente en cómputo.	Modelos grandes donde el cómputo aprieta; solo en capas ocultas, con Adam para el resto.

SGD (*Stochastic Gradient Descent*) es el abuelo de todos los optimizadores. Actualiza los pesos en la dirección del gradiente, con un tamaño de paso fijo (η). Es simple, funciona, pero requiere ajustar el *learning rate* con cuidado: si es muy alto, el entrenamiento diverge; si es muy bajo, tarda una eternidad.

Antes de llegar a Adam hace falta una idea intermedia: el **momentum** (momento). El SGD básico da cada paso mirando solo el gradiente actual, así que en zonas ruidosas va dando bandazos. El momentum le añade memoria: acumula una media de los gradientes recientes y se mueve en esa dirección acumulada, como una bola que rueda cuesta abajo y gana velocidad donde la pendiente es consistente, sin frenarse en cada pequeño bache. Acelera el avance en las direcciones estables y amortigua las oscilaciones. En la práctica casi nunca se usa SGD sin momentum.

Adam (*Adaptive Moment Estimation*) mantiene una estimación del *momentum* (media móvil de los gradientes pasados) y una estimación de la varianza. Con estas dos, adapta el *learning rate* para cada parámetro individualmente. Un parámetro que recibe gradientes grandes y consistentes recibe pasos grandes. Uno que recibe gradientes pequeños o ruidosos recibe pasos pequeños. Esto hace que Adam funcione sorprendentemente bien sin necesidad de ajustar el *learning rate*.

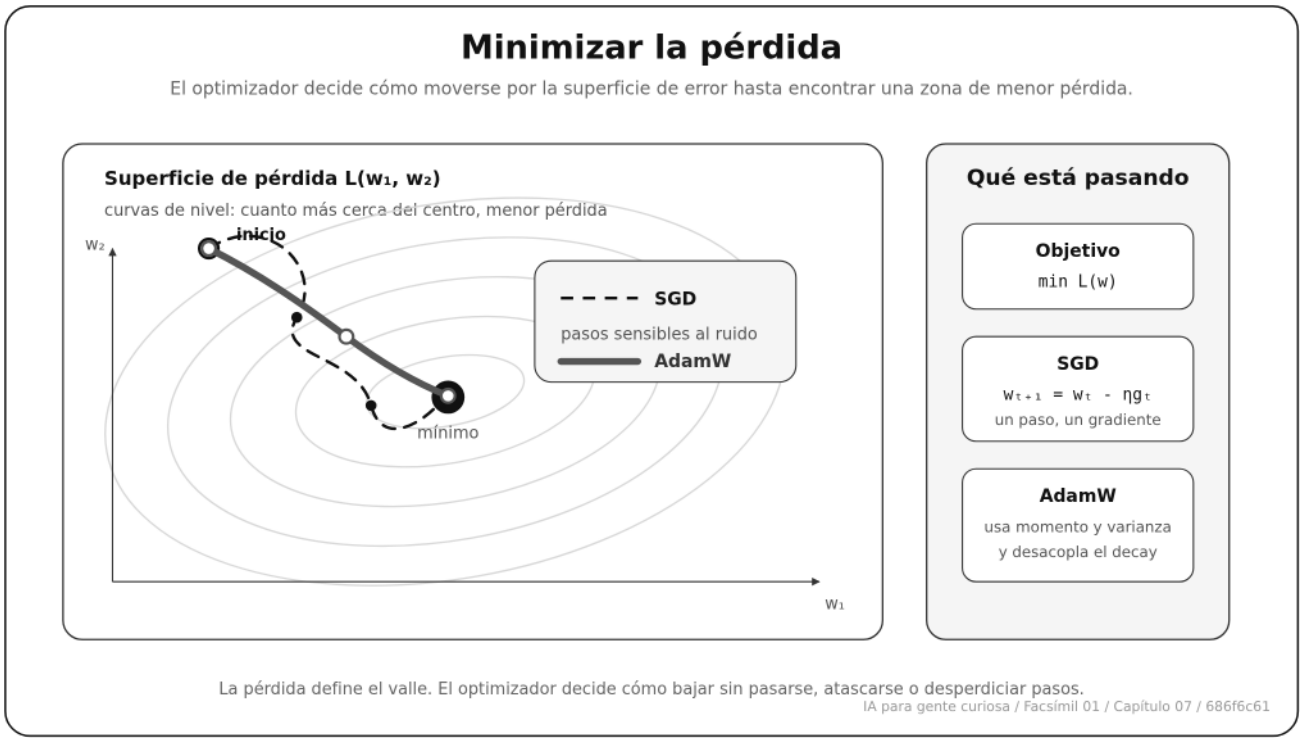
AdamW corrige un error sutil en la implementación original de Adam: la forma en que aplica la regularización *weight decay*. En Adam clásico, la *weight decay* está acoplada a la tasa de aprendizaje adaptativa, lo que reduce su efectividad.[6] AdamW la desacopla, aplicando la regularización directamente a los pesos. Es el optimizador estándar para entrenar LLMs y Transformers.

Un detalle que marca la diferencia al entrenar modelos grandes: el *learning rate* casi nunca es constante, sino que sigue un **calendario** (*schedule*). Lo habitual es un *warmup* al principio, empezar con un *learning rate* muy bajo y subirlo poco a poco durante las primeras iteraciones, para que el modelo no dé bandazos cuando los pesos todavía están desordenados,

seguido de un *decay,* bajarlo de forma progresiva a medida que el entrenamiento avanza, para afinar sin pasarse del mínimo. Todos los LLMs se entrenan con alguna variante de warmup más decay.

Una forma práctica de leer los optimizadores:

SEÑAL OBSERVADA	QUÉ PROBAR	POR QUÉ
La pérdida baja, pero muy lenta	Subir un poco el *learning rate* o usar Adam/AdamW.	El paso puede ser demasiado pequeño.
La pérdida oscila o explota	Bajar el *learning rate*, aplicar *gradient clipping* o revisar escalado de datos.	Los pasos son demasiado grandes o los gradientes son inestables.
Entrenamiento mejora y validación empeora	Añadir *weight decay*, *dropout*, más datos o *early stopping*.	Hay sobreajuste: el modelo memoriza.
Las clases minoritarias fallan	Ponderar la pérdida, cambiar métrica o revisar muestreo.	La pérdida media puede esconder que una clase casi no aprende.
La regresión castiga demasiado valores extremos	Probar MAE, Huber o revisar outliers.	MSE amplifica mucho errores grandes.



Cómo funciona por dentro

El capítulo ha dicho que Adam mantiene una estimación del *momentum* y otra de la varianza, y que con ellas adapta el paso de cada parámetro. Conviene abrir esa caja y ver el mecanismo concreto, porque entender cómo se calcula el paso explica por qué Adam funciona tan bien sin apenas ajuste manual.[7] Para cada peso del modelo, Adam guarda dos números que se actualizan en cada iteración: el primer momento, una media de los gradientes recientes que cumple el papel del *momentum*, y el segundo momento, una media de los gradientes al cuadrado que actúa como medida de la varianza. Ambos son medias móviles exponenciales, es decir, dan más peso a lo reciente y olvidan poco a poco lo antiguo.

El primer momento acumula la dirección reciente del gradiente. Se calcula así:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
m_t	Primer momento en el paso t: media móvil del gradiente	0,12
m_{t-1}	Primer momento del paso anterior	0,10
β_1	Factor de olvido del primer momento. Cuanto más cerca de 1, más memoria	0,9
g_t	Gradiente actual del peso, calculado por la retropropagación	0,30

En palabras: el nuevo m_t es casi todo lo que ya había acumulado (un 90 % con $\beta_1 = 0{,}9$) más una pizca del gradiente nuevo. Así la dirección se suaviza y no salta con cada gradiente ruidoso, igual que el *momentum* clásico.

El segundo momento acumula el tamaño típico del gradiente, no su dirección, porque eleva el gradiente al cuadrado y pierde el signo:

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
v_t	Segundo momento en el paso t: media móvil del gradiente al cuadrado	0,090
v_{t-1}	Segundo momento del paso anterior	0,089
β_2	Factor de olvido del segundo momento. Suele ser muy cercano a 1	0,999
g_t^2	Gradiente actual elevado al cuadrado. Siempre positivo	0,090

En palabras: v_t registra cómo de grandes han sido los gradientes últimamente, sin importar si empujaban hacia arriba o hacia abajo. Un peso con gradientes grandes o muy variables acumula un v_t alto; uno con gradientes pequeños y estables acumula un v_t bajo.

Con esos dos números, Adam actualiza el peso dividiendo el primer momento por la raíz del segundo. Antes aplica una corrección de sesgo, porque en los primeros pasos m_t y v_t arrancan en cero y quedan artificialmente bajos; los símbolos $\hat{m}_t$ y $\hat{v}_t$ son esas versiones corregidas que compensan el arranque frío.

$$\theta_t = \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\hat{v}_t + \epsilon}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
θ_t	Valor del peso tras la actualización	0,481
θ_{t-1}	Valor del peso antes de la actualización	0,500
η	*Learning rate*: tamaño base del paso	0,001
$\hat{m}_t$	Primer momento corregido de sesgo	0,12
$\hat{v}_t$	Segundo momento corregido de sesgo	0,090
ϵ	Número diminuto que evita dividir por cero	0,00000001

En palabras: el paso es proporcional a la dirección acumulada $\hat{m}_t$ dividida por cuánto se mueve ese peso de costumbre, $\hat{v}_t$. Aquí está la clave de la adaptación por parámetro. Un peso con gradientes consistentes y grandes tiene un $\hat{m}_t$ alto frente a un $\hat{v}_t$ moderado, así que el cociente es grande y recibe un paso amplio. Un peso con gradientes ruidosos o muy variables tiene un $\hat{v}_t$ alto, el denominador crece y el paso se encoge. Adam, en la práctica, le da a cada peso su propio *learning rate* efectivo sin que nadie lo configure peso a peso, y por eso funciona razonablemente bien con los valores por defecto.

AdamW añade una última pieza al mecanismo. El *weight decay*, esa regularización que empuja los pesos hacia valores pequeños, no entra dentro del cociente adaptativo, sino que se aplica aparte, restando directamente una fracción del propio peso en cada paso. Esa separación es lo que significa el término desacoplado: en Adam clásico el *weight decay* viajaba mezclado con la actualización adaptativa y su efecto quedaba diluido de forma desigual entre parámetros; en AdamW se resta limpio, con la misma fuerza para todos, y la regularización recupera el efecto que se espera de ella.[8] Ese matiz, aparentemente menor, es la razón de que AdamW se haya convertido en el estándar para entrenar Transformers y modelos de lenguaje.

Más allá de Adam: la nueva generación

AdamW lleva años siendo el estándar, pero desde 2023 ha aparecido una familia de optimizadores que le disputa el trono, y conviene conocerla porque ya se usa para entrenar modelos de primera línea.

El más comentado es **Muon**, de *MomentUm Orthogonalized by Newton-Schulz*, propuesto por Keller Jordan a finales de 2024.[9] La idea, en una frase: antes de dar el paso, Muon endereza el momento. Coge la dirección acumulada (el mismo *momentum* que ya conoces) y la ortogonaliza, es decir, la convierte en una matriz cuyas direcciones tienen todas una fuerza parecida, mediante una operación iterativa llamada Newton-Schulz que basta con repetir unas cinco veces. El efecto práctico es que ningún sentido del espacio de pesos se come a los demás, así que el entrenamiento avanza más equilibrado y llega a la misma calidad con menos pasos que AdamW. Muon se aplica solo a las matrices de pesos de las capas ocultas; los *embeddings*, la capa de salida y los parámetros sueltos (escalares y *bias*) se siguen entrenando con Adam, porque ahí la ortogonalización no aporta.

Que no es un experimento de laboratorio lo demostró Moonshot AI en 2025: entrenó **Kimi K2**, un modelo de un billón de parámetros, de los que se activan unos 32.000 millones por cada token porque es una mezcla de expertos, con una variante de Muon llamada **MuonClip**, y completó 15,5 billones de tokens de preentrenamiento sin un solo pico de inestabilidad.[10] El "Clip" tapa el talón de Aquiles de Muon a gran escala: cada cierto número de pasos reescala las matrices de consulta y clave de la atención para que sus valores no se disparen, que es justo lo que solía descarrilar estos entrenamientos.

Muon no está solo. **Lion**, de *EvoLved Sign Momentum*, lo descubrió Google en 2023 dejando que un buscador automático explorara algoritmos de optimización.[11] Lion se queda solo con el **signo** del momento, de modo que todos los pesos dan un paso del mismo tamaño y lo único que cambia es la dirección; a cambio de algo menos de finura, gasta la mitad de memoria que Adam, porque no necesita guardar el segundo momento. En esa misma búsqueda de ahorrar memoria está **Adafactor**, que en vez del segundo momento entero almacena una versión factorizada, y en el extremo opuesto está **Shampoo**, un método de segundo orden que estima la curvatura de la superficie por bloques para dar pasos mejor informados, a costa de más cálculo.

¿Quiere decir esto que AdamW está acabado? No. Sigue siendo la opción por defecto, la más probada y la que mejor se porta sin tocar nada, y para la inmensa mayoría de proyectos es la elección correcta. Pero cuando se entrenan modelos enormes y cada hora de GPU cuenta, esta nueva generación es la frontera donde hoy se pelea por exprimir el cómputo. La lección de fondo no cambia: el optimizador no aprende por ti, solo decide cómo moverte por la superficie de error, y elegirlo bien puede ahorrar semanas de entrenamiento.

Problemas comunes del entrenamiento

Incluso con la función de pérdida y el optimizador correctos, el entrenamiento puede fallar. Dos problemas aparecen una y otra vez:

Overfitting (sobreajuste). El modelo memoriza los datos de entrenamiento pero falla estrepitosamente con datos nuevos. Es como un estudiante que se aprende las respuestas del examen de memoria sin entender la materia: saca un 10 en el simulacro y suspende el examen real.[12] Soluciones: *dropout* (apagar neuronas aleatoriamente durante el entrenamiento), regularización (penalizar pesos grandes), más datos (la mejor medicina), *data augmentation* (generar variaciones de los datos existentes).

Vanishing gradients (desvanecimiento del gradiente). En redes profundas, el gradiente puede hacerse tan pequeño en las primeras capas que dejan de aprender. Es como si el profesor susurrara la corrección al alumno de la última fila, y el mensaje se fuera atenuando fila a fila hasta que los de delante no oyen nada.[13] Soluciones: ReLU en lugar de sigmoide (su gradiente es 1 para valores positivos), *skip connections* (atajos que permiten al gradiente saltar capas), *batch normalization* (normalizar las activaciones de cada capa).

El contrato mínimo de una run

Una *run* de entrenamiento no debería ser «he lanzado un modelo y ya veremos». Debería tener un contrato mínimo antes de empezar:

ELEMENTO	DECISIÓN	EJEMPLO
Tarea	Clasificación binaria, multiclase, regresión o representación.	Priorizar tickets: multiclase.
Salida	Número de salidas y activación final.	3 clases → softmax.
Pérdida	Función que coincide con la tarea.	Cross-entropy multiclase.
Métrica	Qué miras para decidir si sirve.	F1 macro si hay desbalance; accuracy si las clases están equilibradas.
Optimizador	Algoritmo y sus hiperparámetros.	AdamW, `lr=1e-3`, `weight_decay=1e-2`.
Validación	Partición que no actualiza pesos.	80/20 estratificado o validación temporal si hay fechas.
Criterio de parada	Cuándo dejar de entrenar.	Parar si validación no mejora en 5 épocas.
Observabilidad	Qué registras por época.	Train loss, val loss, métrica, norma del gradiente, learning rate.

Este contrato es una defensa contra el autoengaño. Si cambias pérdida, métrica y partición cada vez que algo sale mal, no estás optimizando: estás persiguiendo resultados. Una run seria deja claro qué cuenta como mejora antes de mirar el resultado.

En el día a día

En la práctica, casi nunca eliges la función de pérdida desde cero. Los *frameworks* ya tienen las implementaciones optimizadas. Pero sí tomas decisiones que dependen de entenderlas, y cada una es una decisión de ingeniería concreta, no una preferencia estética.

¿Cross-entropy o MSE? La pregunta no es de gusto, sino de qué supone cada pérdida sobre tu salida. La cross-entropy asume que el modelo emite una distribución de probabilidad y mide cuánta sorpresa le causa la respuesta correcta; encaja con clasificar (spam o no, gato o perro) y con predecir el siguiente token, donde la última capa es un softmax. La MSE asume una salida continua y penaliza la distancia al cuadrado, así que solo tiene sentido cuando predices un número real (un precio, una temperatura). Equivocarse aquí es más común de lo que parece y no siempre rompe el entrenamiento de forma visible: si usas MSE sobre probabilidades, el modelo aprende algo, pero su gradiente deja de ser el error limpio $p_i - y_i$ y la convergencia se vuelve lenta y peor calibrada. La regla operativa es mirar la activación final: softmax o sigmoide piden cross-entropy; una salida lineal pide MSE (o MAE/Huber si hay valores extremos que no quieres que dominen).

¿Qué optimizador y con qué calendario? Empieza con Adam o AdamW: su *learning rate* adaptativo por parámetro hace que funcionen bien sin afinar mucho, y cubren la inmensa mayoría de los casos. AdamW es preferible cuando vas a usar *weight decay*, porque lo desacopla de la actualización adaptativa y la regularización vuelve a tener el efecto esperado; por eso es el estándar al entrenar Transformers. La decisión no termina en el optimizador: en modelos grandes el *learning rate* casi nunca es constante. Un *warmup* corto al arranque evita que los pasos den bandazos cuando los pesos todavía están desordenados, y un *decay* posterior afina sin pasarse del mínimo. SGD con momentum sigue siendo una opción válida cuando quieres control total y estás dispuesto a buscar el *learning rate* a mano, algo habitual en visión por sus efectos de generalización.

¿Overfitting o underfitting? El diagnóstico se lee comparando dos curvas, no una. Si la pérdida de entrenamiento es baja pero la de validación es alta (y la brecha entre ambas crece con las épocas), tienes *overfitting*: el modelo memoriza en lugar de generalizar. La respuesta es regularizar (*weight decay*, *dropout*), conseguir más datos o aplicar *data augmentation*, y cortar con *early stopping* cuando la validación deja de mejorar. Si en cambio ambas pérdidas son altas y se estancan, tienes *underfitting*: el modelo es demasiado simple, el *learning rate* es inadecuado o no has entrenado lo suficiente; ahí toca subir capacidad, ajustar el optimizador o entrenar más. Confundir los dos lleva a aplicar el remedio contrario: regularizar un modelo que ya se queda corto solo empeora el resultado.

Por qué debería importarte

La función de pérdida es lo que le dices al modelo que quieres. No es un detalle técnico: es la especificación formal de tu objetivo.

Si usas MSE para clasificar, el modelo intentará minimizar la distancia numérica entre probabilidades, no maximizar la probabilidad de la clase correcta. Aprenderá algo, pero no lo que quieres. Si usas *cross-entropy* para regresión, el modelo asumirá que la salida es una distribución de probabilidad, no un valor continuo. Los resultados serán inconsistentes.

El optimizador, por su parte, determina si el modelo converge, a qué velocidad y a qué solución. Adam con un *learning rate* por defecto funciona en la mayoría de casos. Pero si tu modelo no converge, cambiar de optimizador o ajustar sus hiperparámetros puede ser la diferencia entre el éxito y el fracaso.

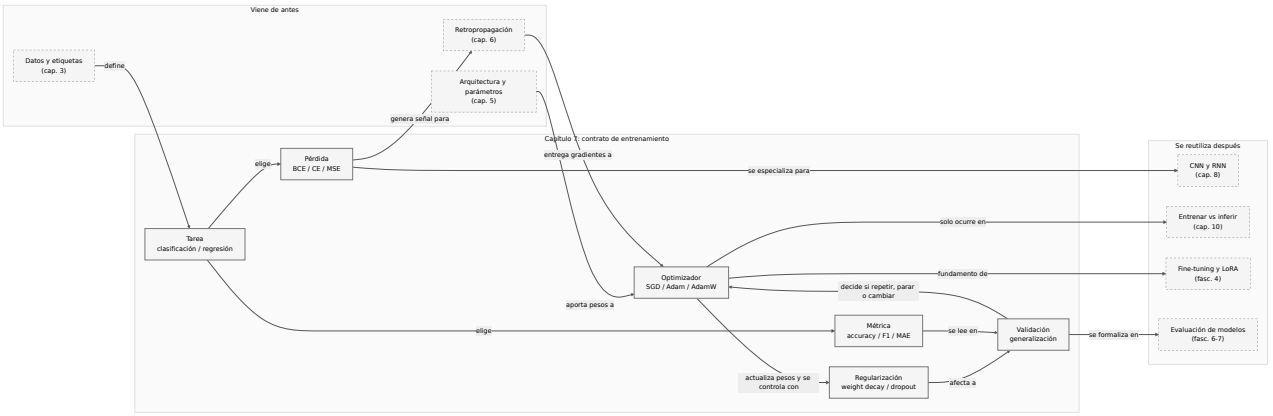
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Usar MSE para clasificación	MSE asume errores continuos y simétricos. La clasificación necesita comparar distribuciones de probabilidad. MSE penaliza igual una predicción de 0,4 cuando la respuesta es 1 que una predicción de 0,9 cuando la respuesta es 0.	*Cross-entropy* para clasificación. Siempre.
No monitorizar la pérdida de validación	Si solo miras la pérdida de entrenamiento, no detectas el *overfitting* hasta que es demasiado tarde. La pérdida de entrenamiento siempre baja; la de validación es la que importa.	Separa un conjunto de validación y monitoriza ambas curvas. Si divergen, estás sobreajustando.
Usar el *learning rate* por defecto sin comprobarlo	El valor por defecto de Adam (0,001) funciona en muchos casos, pero no en todos. Un *learning rate* demasiado alto hace que la pérdida oscile; uno demasiado bajo hace que el entrenamiento sea innecesariamente lento.	Prueba con 0,01, 0,001 y 0,0001. Observa la curva de pérdida: si oscila, baja el *learning rate*; si baja muy lentamente, súbelo.
Olvidar el *weight decay*	Sin regularización, los pesos pueden crecer descontroladamente, llevando a *overfitting*. AdamW incluye *weight decay* por defecto, pero SGD no.	Añade *weight decay* (típicamente 1e-4 a 1e-2). Observa si mejora la pérdida de validación.
Elegir métrica que contradice la pérdida	Puedes optimizar cross-entropy y celebrar accuracy mientras la clase minoritaria queda destrozada. La pérdida entrena; la métrica decide si te vale.	Define pérdida y métrica juntas. En desbalance, mira F1 macro, recall por clase o coste de error.
Comparar runs con particiones distintas	Si cada experimento usa otro split, no sabes si mejoró el modelo o si tuvo una validación más fácil.	Fija partición, semilla y protocolo antes de comparar optimizadores.

Cómo encaja todo

Este capítulo convierte el gradiente del capítulo 6 en una decisión de entrenamiento completa. La pérdida define qué significa fallar; el optimizador decide cómo moverse; la validación decide si esa mejora sirve fuera de los ejemplos que actualizan pesos.

También prepara una idea que volverá muchas veces: entrenar no es solo bajar una métrica interna. Es diseñar un contrato medible, ejecutarlo de forma reproducible y mirar si generaliza.


Viene de antes
Datos y etiquetas (cap. 3)
define
Retropropagación (cap. 6)
Arquitectura y parámetros (cap. 5)
genera señal para
Capítulo 7: contrato de entrenamiento
elige
Pérdida BCE / CE / MSE
entrega gradientes a
se especializa para
Tarea clasificación / regresión
Optimizador SGD / Adam / AdamW
aporta pesos a
solo ocurre en
fundamento de
decide si repetir, parar o cambiar
elige
Métrica accuracy / F1 / MAE
se lee en
Validación generalización
se formaliza en
actualiza pesos y se controla con
Regularización weight decay / dropout
afecta a
Se reutiliza después
CNN y RNN (cap. 8)
Entrenar vs inferir (cap. 10)
Fine-tuning y LoRA (fasc. 4)
Evaluación de modelos (fasc. 6-7)

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Función de pérdida	Función que mide el error entre la predicción y la realidad. El entrenamiento busca minimizarla.
Cross-entropy	Pérdida estándar para clasificación. Mide la diferencia entre la distribución predicha y la real.
MSE	Error cuadrático medio. Pérdida estándar para regresión. Penaliza fuertemente los errores grandes.
Margen	Puntuación por etiqueta ($+1$/-1). Su signo dice si se acierta; su tamaño, con cuánta confianza.
Pérdida 0-1	Vale 1 si hay error y 0 si hay acierto. Mide lo que importa, pero es plana: no sirve para entrenar con gradientes.
Pérdida bisagra (*hinge*)	Sustituta de la 0-1 que solo deja de penalizar con holgura (margen mayor que 1). Es la pérdida de las SVM.
Pérdida sustituta (*surrogate*)	Pérdida suave y con pendiente que reemplaza a la 0-1 para que el descenso de gradiente tenga dirección de mejora.
Optimizador	Algoritmo que actualiza los pesos a partir de los gradientes. Controla cómo y cuánto se ajusta cada parámetro.
Adam	Optimizador con *learning rate* adaptativo por parámetro. El más usado en la práctica.
AdamW	Variante de Adam con *weight decay* desacoplado. Es el estándar habitual para Transformers y muchos entrenamientos modernos.
Muon	Optimizador que ortogonaliza el *momentum* de las matrices de las capas ocultas (con iteración de Newton-Schulz). Más eficiente en cómputo; se combina con Adam para el resto de parámetros.
Lion	Optimizador que actualiza con el signo del momento: pasos de magnitud uniforme y menos memoria que Adam.
Momentum	Memoria que acumula una media de los gradientes recientes para acelerar las direcciones estables y amortiguar las oscilaciones del SGD.
Calendario de *learning rate*	Plan que varía la tasa de aprendizaje durante el entrenamiento, normalmente con un *warmup* inicial seguido de un *decay* progresivo.
Warmup	Fase inicial en la que el *learning rate* empieza muy bajo y sube poco a poco para evitar bandazos mientras los pesos están desordenados.

TÉRMINO	DEFINICIÓN
Perplejidad	Exponencial de la cross-entropy media. Mide entre cuántas opciones equivalentes duda de media un modelo de lenguaje en cada token.
Label smoothing	Suavizado de las etiquetas (la correcta deja de ser un 1 exacto) que regulariza y mejora la calibración, habitual al entrenar Transformers.
Weight decay	Regularización que penaliza pesos grandes y ayuda a reducir sobreajuste.
Overfitting	El modelo memoriza los datos de entrenamiento pero no generaliza a datos nuevos.
Curva de validación	Evolución de pérdida o métrica en datos que no actualizan pesos. Permite detectar generalización, sobreajuste y parada temprana.
Early stopping	Detener el entrenamiento cuando la validación deja de mejorar durante varias épocas.

Antes de pasar página

- [ ] ¿Puedo explicar cuándo usar *cross-entropy* y cuándo MSE? (Si no, vuelve a la tabla de funciones de pérdida.)
- [ ] ¿Entiendo la diferencia entre SGD, Adam y AdamW? (Si no, vuelve a la sección de optimizadores.)
- [ ] ¿Sé qué aporta Muon frente a AdamW y por qué empieza a usarse en los modelos grandes? (Si no, vuelve a «Más allá de Adam».)
- [ ] ¿Sé qué es el *overfitting* y cómo combatirlo? (Si no, vuelve a «Problemas comunes».)
- [ ] ¿Puedo escribir el contrato mínimo de una run antes de entrenar? (Si no, vuelve a «El contrato mínimo de una run».)
- [ ] ¿Sé comparar BCE, MSE, SGD y AdamW y elegir una combinación según el problema? (Si no, vuelve a «Optimizadores: cómo aplicar los ajustes».)
- [ ] ¿Entiendo por qué el *learning rate* no puede ser ni muy alto ni muy bajo? (Si no, vuelve a «Dónde solía tropezar yo».)

En resumen

IDEA FUERZA	DETALLE
La función de pérdida define qué significa «equivocarse».	*Cross-entropy* para clasificar, MSE para regresión, *contrastive* para representaciones. Elegir mal la pérdida es pedirle al modelo que optimice lo incorrecto.
El optimizador decide cómo aplicar los gradientes.	SGD es simple pero necesita ajuste. Adam es adaptativo y funciona en la mayoría de casos. AdamW es el estándar para Transformers, y una nueva generación (Muon, Lion) empieza a disputarle el sitio en los modelos más grandes.
El *overfitting* y el *vanishing gradient* son los dos grandes enemigos.	Monitoriza la pérdida de validación, usa *dropout* y regularización, y elige bien tus funciones de activación.
Una run necesita contrato antes de empezar.	Tarea, salida, pérdida, métrica, optimizador, validación y criterio de parada deben estar definidos antes de mirar resultados.

Para saber más

Chen, X. y otros (2023). Symbolic discovery of optimization algorithms. *arXiv*. https://arxiv.org/abs/2302.06675

Cortes, C. y Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273-297. https://doi.org/10.1007/BF00994018

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

Jordan, K. (2024). *Muon: an optimizer for hidden layers in neural networks* [entrada de blog]. https://kellerjordan.github.io/posts/muon/

Kimi Team (2025). *Kimi K2: open agentic intelligence*. *arXiv*. https://arxiv.org/abs/2507.20534

Kingma, D. P. y Ba, J. (2015). Adam: a method for stochastic optimization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1412.6980

LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539

Liang, P. y Sadigh, D. *Artificial intelligence: principles and techniques* (notas del curso CS221). Stanford University. https://stanford-cs221.github.io

Loshchilov, I. y Hutter, F. (2019). Decoupled weight decay regularization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1711.05101

Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0

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

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. y Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1), 1929-1958. http://jmlr.org/papers/v15/srivastava14a.html

---

Notas

1. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org. El capítulo 6.2 aborda en profundidad las funciones de pérdida, su relación con la estimación de máxima verosimilitud y cómo la elección de la función de pérdida determina lo que el modelo aprende. ↩

2. Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0. Aunque el artículo se centra en la retropropagación con MSE, establece el marco general donde cualquier función de pérdida diferenciable puede usarse para entrenar redes multicapa. ↩

3. Liang, P. y Sadigh, D. *Artificial intelligence: principles and techniques* (notas del curso CS221). Stanford University. https://stanford-cs221.github.io. Las notas presentan el armazón puntuación → margen → pérdida como base unificada del aprendizaje supervisado lineal, del que la pérdida *hinge* (SVM) y la logística son dos casos. ↩

4. Cortes, C. y Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273-297. https://doi.org/10.1007/BF00994018. El artículo introduce las máquinas de vectores de soporte, que buscan la frontera de máximo margen y cuya pérdida asociada es la *hinge*. ↩

5. Kingma, D. P. y Ba, J. (2015). Adam: a method for stochastic optimization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1412.6980. Kingma y Ba presentaron Adam, que combina las ventajas de AdaGrad (tasas de aprendizaje adaptativas) y RMSProp (media móvil del gradiente), convirtiéndose en el optimizador por defecto para la mayoría de aplicaciones de *deep learning*. ↩

6. Loshchilov, I. y Hutter, F. (2019). Decoupled weight decay regularization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1711.05101. Los autores demostraron que desacoplar la *weight decay* de la actualización del gradiente en Adam mejora significativamente la generalización. ↩

7. Kingma, D. P. y Ba, J. (2015). Adam: a method for stochastic optimization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1412.6980. El artículo deriva las dos medias móviles, la corrección de sesgo y la regla de actualización que se reproducen en esta sección. ↩

8. Loshchilov, I. y Hutter, F. (2019). Decoupled weight decay regularization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1711.05101. Los autores muestran que aplicar el *weight decay* fuera de la actualización adaptativa, en lugar de sumarlo al gradiente, mejora la generalización y vuelve más predecible el ajuste del hiperparámetro. ↩

9. Jordan, K. (2024). *Muon: An optimizer for hidden layers in neural networks* [entrada de blog]. https://kellerjordan.github.io/posts/muon/. Introduce Muon y la ortogonalización del momento mediante iteración de Newton-Schulz como vía para entrenar las capas ocultas con menos pasos que AdamW. ↩

.0. Kimi Team (2025). *Kimi K2: Open Agentic Intelligence*. arXiv:2507.20534. https://arxiv.org/abs/2507.20534. Describe MuonClip, que añade a Muon un reescalado periódico de las matrices de consulta y clave de la atención (*QK-clip*) para estabilizar el entrenamiento a gran escala, y reporta el preentrenamiento de Kimi K2 sin picos de pérdida. ↩

.1. Chen, X. y otros (2023). *Symbolic discovery of optimization algorithms*. arXiv:2302.06675. https://arxiv.org/abs/2302.06675. Lion, hallado por búsqueda simbólica de programas, actualiza con el signo del momento y, al no guardar el segundo momento, usa menos memoria que Adam manteniendo un rendimiento comparable. ↩

.2. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. y Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1), 1929-1958. http://jmlr.org/papers/v15/srivastava14a.html. El *dropout* (desactivar aleatoriamente neuronas durante el entrenamiento) es una de las técnicas más efectivas para prevenir el *overfitting*. ↩

.3. LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539. Los autores explican cómo las funciones de activación como ReLU y las arquitecturas con conexiones residuales mitigan el problema del desvanecimiento del gradiente. ↩

Capítulo 08: CNNs y RNNs: redes para visión y secuencias

Entrando en el tema

Hasta ahora hemos hablado de redes neuronales en abstracto: capas que se conectan con capas, neuronas que reciben entradas y producen salidas. Esa arquitectura genérica, el MLP, funciona para muchos problemas, pero tiene un punto débil: trata todos los datos como una lista plana de números, sin aprovechar su estructura.

Una imagen no es una lista plana. Es una rejilla de píxeles donde lo que importa son los patrones locales: un ojo está rodeado de ceja y párpado, no de píxeles aleatorios dispersos por la imagen. Un texto no es una lista plana. Es una secuencia donde el significado de una palabra depende de las que vinieron antes y de las que vendrán después.

Dos arquitecturas nacieron para resolver estos dos problemas. Las CNNs para la visión. Las RNNs para las secuencias. Y aunque los Transformers las han desplazado parcialmente, entenderlas es entender por qué el Transformer fue revolucionario.

CNNs: cómo ve una máquina

Una *Convolutional Neural Network* (CNN) no mira una imagen píxel a píxel. Aplica **filtros**: pequeñas matrices (3×3, 5×5) que se deslizan por la imagen detectando patrones locales.[1]

Piensa en un filtro como una plantilla. Un filtro que detecta bordes horizontales tiene valores positivos arriba y negativos abajo. Cuando se desliza sobre una zona de la imagen donde hay un borde horizontal, produce un valor alto. Cuando pasa por una zona uniforme, produce un valor cercano a cero.

El flujo de una CNN es:

```
Imagen → Convoluciones → Pooling → Más convoluciones → Pooling → Clasificación
```

Convolución. El corazón de la CNN. Un filtro pequeño se desliza por toda la imagen. En cada posición, multiplica sus valores por los píxeles correspondientes y suma el resultado. Una capa convolucional típica tiene decenas o cientos de filtros, cada uno especializado en detectar un patrón distinto: bordes verticales, esquinas, texturas, colores específicos.[2]

Veamos un ejemplo concreto. Imagina una imagen de 5×5 píxeles en escala de grises y un filtro de 3×3 que detecta bordes verticales:

```
Imagen 5x5:       Filtro 3x3 (borde vertical):
 1  1  1  0  0      -1   0   1
 1  1  1  0  0      -1   0   1
 1  1  1  0  0      -1   0   1
 1  1  1  0  0
 1  1  1  0  0
```

El filtro se coloca en la esquina superior izquierda y multiplica elemento a elemento: (-1×1 + 0×1 + 1×1) + (-1×1 + 0×1 + 1×1) + (-1×1 + 0×1 + 1×1) = 0 + 0 + 0 = 0. En esa zona no hay borde vertical: todo son unos.

Ahora deslizamos el filtro a la derecha, donde están los ceros. En la zona de transición entre unos y ceros: (-1×1 + 0×1 + 1×0) + (-1×1 + 0×1 + 1×0) + (-1×1 + 0×1 + 1×0) = (-1) + (-1) + (-1) = -3. El valor absoluto alto indica un borde vertical.

Esto es lo que hacen decenas de filtros en paralelo, cada uno especializado en un patrón distinto. La red **aprende** los valores del filtro durante el entrenamiento: no los programa una persona, los descubre la retropropagación.

En notación compacta, la convolución de una imagen I con un filtro K de tamaño $k \times k$ produce, en cada posición (i, j), esta suma:

$$(I * K)_{i,j} = \sum_{m=1}^{k} \sum_{n=1}^{k} I_{i+m,\, j+n} \cdot K_{m,n}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
I	Imagen de entrada (o mapa de la capa anterior)	rejilla 5×5
K	Filtro o *kernel* que se desliza	matriz 3×3
k	Tamaño del filtro (lado)	3
(i, j)	Posición donde se apoya el filtro	esquina superior izquierda
$(I * K)_{i,j}$	Valor de salida en esa posición	0 en zona uniforme, −3 en un borde

En palabras: en cada posición se multiplica el filtro por la zona de imagen que tiene debajo y se suman todos los productos. Eso da un punto del mapa de salida; deslizando el filtro por toda la imagen se obtiene el mapa completo.

Dos hiperparámetros controlan cómo se desliza el filtro:

- **Stride (paso).** Cuántos píxeles avanza el filtro en cada salto. Con *stride* 1 se solapa mucho y la salida casi conserva el tamaño; con *stride* 2 salta de dos en dos y reduce la resolución a la mitad.
- **Padding (relleno).** Cuántos píxeles de ceros se añaden alrededor del borde. Sin *padding* la salida encoge y los bordes se procesan menos veces; con *padding* se conserva el tamaño.

El tamaño del mapa de salida conviene tenerlo a mano para que las dimensiones cuadren:

$$\text{salida} = \left\lfloor \frac{N - K + 2P}{S} \right\rfloor + 1$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
N	Tamaño de la entrada (lado)	5
K	Tamaño del filtro (lado)	3
P	*Padding*: píxeles añadidos por lado	0
S	*Stride*: paso del filtro	1
salida	Tamaño del mapa resultante (lado)	$(5 - 3 + 0)/1 + 1 = 3$

En palabras: una imagen de 5×5 con un filtro de 3×3, sin relleno y paso 1, produce un mapa de 3×3.

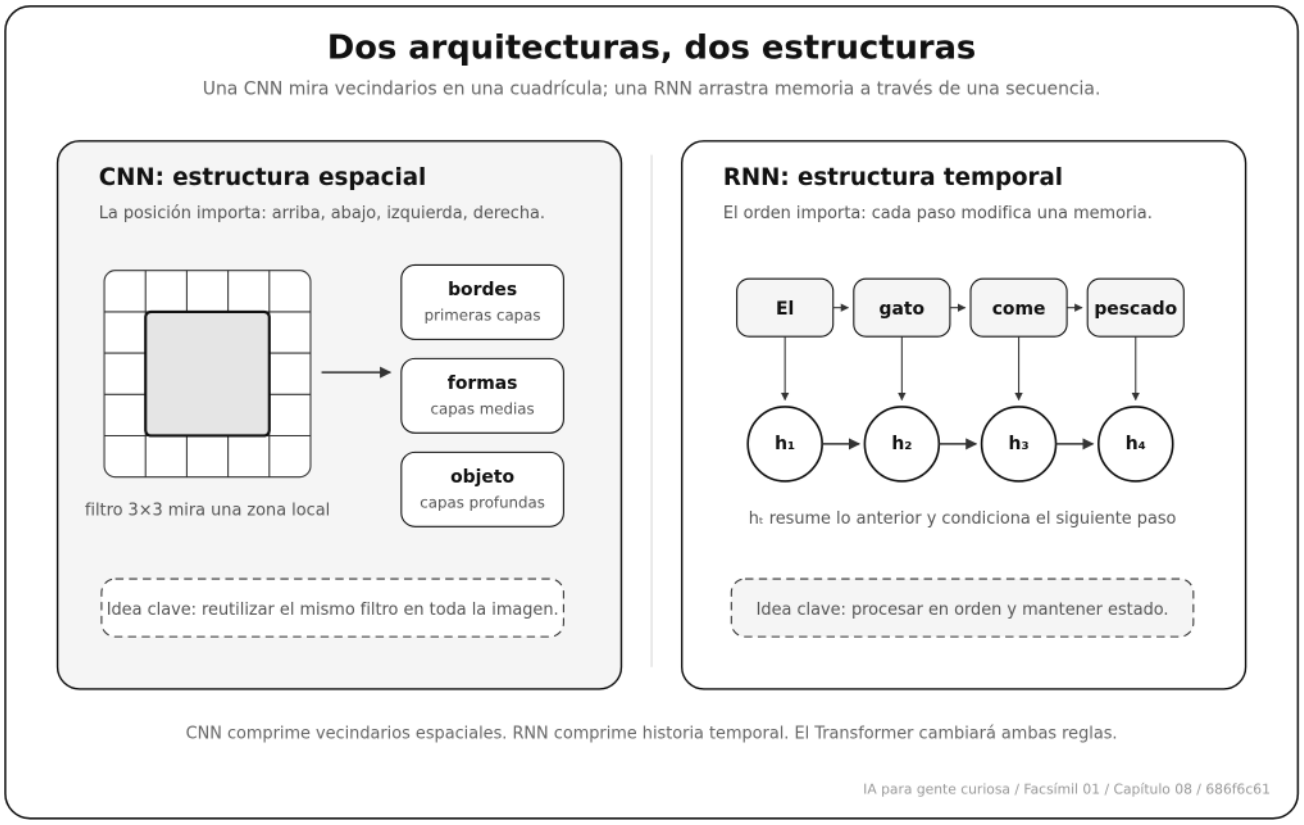
Pooling. Reduce la resolución espacial manteniendo la información importante. La operación más común es *max pooling*: divide la imagen en zonas (por ejemplo, 2×2) y se queda solo con el valor máximo de cada zona. Esto hace la red más eficiente (menos píxeles que procesar) y más robusta (invariante a pequeños desplazamientos: si el gato se mueve dos píxeles a la derecha, el *max pooling* probablemente siga detectándolo).

De patrones simples a conceptos. Las primeras capas detectan bordes y colores. Las capas intermedias combinan bordes en formas y texturas. Las capas profundas reconocen objetos: ojos, ruedas, letras. Es la misma jerarquía que vimos en el capítulo 5, pero especializada para datos visuales.

Canales y mapas de características. El ejemplo anterior usa una imagen en escala de grises, con un solo canal. Una imagen en color tiene **tres canales** (rojo, verde y azul), y el filtro opera sobre los tres a la vez: un filtro 3×3 sobre una imagen RGB es en realidad un volumen 3×3×3. Además, una capa convolucional no aplica un único filtro, sino decenas o cientos, y cada uno produce su propio **mapa de características** (*feature map*). Así, una capa

puede recibir 3 canales y entregar 64 mapas, que la capa siguiente tratará como sus 64 canales de entrada. Esa es la razón por la que las CNNs profundas tienen tantos parámetros pese a que cada filtro individual es diminuto.

Skip connections (ResNet). En 2015, ResNet introdujo una idea simple pero revolucionaria: atajos que permiten a la información saltarse capas.[3] En lugar de aprender la transformación completa, la capa aprende el «residuo»: la diferencia entre la entrada y la salida deseada. Si la transformación óptima es no hacer nada (la identidad), la capa puede aprender pesos cercanos a cero. Esto resolvió el problema de entrenar redes muy profundas y permitió arquitecturas de más de cien capas.



Relevancia actual. Las CNNs siguen siendo muy importantes en visión por computador porque aprovechan una estructura real de las imágenes: píxeles cercanos suelen estar relacionados. Ese sesgo inductivo las hace eficientes en detección, segmentación, clasificación y despliegues *edge*. Pero ya no son la única familia fuerte: los *Vision Transformers* tratan una imagen como parches y usan atención, y han demostrado que con suficientes datos y cómputo pueden competir o superar a las CNNs en muchas tareas.[4] En generación de imágenes, además, conviven U-Nets convolucionales y modelos de difusión basados en Transformers, como DiT.[5] La decisión de ingeniería no es “CNN vieja, Transformer nuevo”, sino qué estructura de datos, coste, volumen de entrenamiento y latencia tienes.

RNNs y LSTMs: cómo procesar secuencias

Antes de los Transformers, si querías que una red procesara texto, audio o series temporales, usabas una *Recurrent Neural Network* (RNN). Su idea es elegante: procesar la secuencia elemento a elemento, manteniendo un **estado oculto** que resume todo lo visto hasta ahora.[6]

```
Token 1 → RNN (estado oculto) → Token 2 + estado → RNN (actualiza estado) → Token 3 + estado...
```

Cuando la RNN lee la palabra «El», actualiza su estado oculto para reflejar que ha visto un artículo. Cuando lee «gato», el estado ahora codifica «El gato». Cuando llega a «es», el estado refleja «El gato es». Y así, token a token, el estado oculto acumula el contexto de toda la secuencia.

Esa actualización del estado tiene una fórmula, la ecuación que define una RNN:

$$h_t = \tanh(W_h\, h_{t-1} + W_x\, x_t + b)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x_t	Entrada en el paso t (el *embedding* del token actual)	el vector de «gato»
h_{t-1}	Estado oculto del paso anterior, la memoria acumulada	resumen de «El»
h_t	Nuevo estado oculto, ya actualizado con x_t	resumen de «El gato»
W_x, W_h	Matrices de pesos aprendidas (para la entrada y para el estado)	se ajustan al entrenar
b	Vector de sesgo	se ajusta al entrenar
$\tanh$	Activación que mantiene el estado acotado	salida entre −1 y 1

En palabras: el nuevo estado mezcla la entrada actual con el estado anterior, lo pasa por una activación y produce la memoria actualizada. Fíjate en que W_h y W_x son las mismas en todos los pasos: por eso, en el *backward*, el gradiente se multiplica una y otra vez por W_h, lo que explica el desvanecimiento o la explosión que veremos más abajo.

El problema es la memoria. Cuando llegas al token 500, el estado oculto ya ha sido sobrescrito cientos de veces. La información del token 1 se ha diluido hasta desaparecer. Es como el juego del teléfono: el mensaje se degrada con cada transmisión.[7]

LSTM: memoria con compuertas

La *Long Short-Term Memory* (LSTM) resuelve este problema con un mecanismo ingenioso: compuertas. En lugar de un solo estado oculto que se sobrescribe, la LSTM tiene:

- Una **compuerta de olvido**: decide qué información antigua descartar.
- Una **compuerta de entrada**: decide qué información nueva almacenar.
- Una **compuerta de salida**: decide qué parte del estado actual exponer como salida.

Estas compuertas son pequeñas redes neuronales que aprenden cuándo abrir y cuándo cerrar. Si el modelo encuentra una palabra clave al principio de un documento que será relevante quinientos tokens después, la compuerta de olvido puede decidir conservarla. Si encuentra información irrelevante, la descarta.

Gracias a las LSTMs, las RNNs pudieron capturar dependencias bastante más largas que una RNN simple. No conviene convertir esto en una cifra universal: que un modelo recuerde 100, 500 o más pasos depende de datos, arquitectura, tamaño del estado, entrenamiento y tarea. Lo importante es la forma del límite: la información tiene que pasar por una cadena temporal, y cada paso puede degradar la señal.

GRU: la hermana pequeña de la LSTM

La *Gated Recurrent Unit* (GRU), presentada en 2014, simplifica la LSTM: tiene solo dos compuertas (reinicio y actualización) en lugar de tres, y fusiona el estado oculto con la memoria de largo plazo en un solo vector.[8] Con menos parámetros que la LSTM, la GRU entrena más rápido y a menudo rinde igual de bien. Es la opción preferida cuando los recursos son limitados o cuando la LSTM no aporta una mejora significativa.

Bidireccional: mirar al pasado y al futuro

Una RNN estándar solo ve el pasado: cuando procesa la palabra 5, conoce las palabras 1 a 4 pero no tiene ni idea de lo que vendrá en la 6. Para muchas tareas (como etiquetar categorías gramaticales o traducir) el contexto futuro es tan importante como el pasado.

Las **RNNs bidireccionales** resuelven esto con dos RNNs en paralelo: una procesa la secuencia de izquierda a derecha y la otra de derecha a izquierda. Sus estados ocultos se concatenan, dando a cada posición acceso tanto al contexto anterior como al posterior. Son más caras computacionalmente pero notablemente más precisas en tareas donde el contexto completo importa.

El gradiente en las RNNs: un problema amplificado

El desvanecimiento del gradiente es especialmente dañino en las RNNs. En una red *feedforward* de 10 capas, el gradiente se atenúa 10 veces. En una RNN que procesa 100 tokens, el gradiente se atenúa 100 veces (porque la red se despliega en el tiempo y cada paso

temporal equivale a una capa). Es como si tuvieras una red de 100 capas, pero donde todas comparten los mismos pesos. Cada paso hacia atrás en el tiempo multiplica el gradiente por la misma matriz de pesos. Si los valores propios de esa matriz son menores que 1, el gradiente se desvanece exponencialmente. Si son mayores que 1, **explota**, produciendo actualizaciones caóticas.

Las LSTMs y GRUs mitigan esto con sus compuertas, que permiten que el gradiente fluya a través del estado de memoria con menos atenuación. Pero no eliminan el cuello de botella: la información sigue viajando paso a paso. Para secuencias muy largas, muchas tareas dejaron de intentar “mejorar la memoria recurrente” y pasaron a usar atención, recuperación o arquitecturas híbridas. Eso es exactamente lo que hizo el Transformer para lenguaje.

RNN vs LSTM vs Transformer

CARACTERÍSTICA	RNN	LSTM	TRANSFORMER
Procesamiento	Secuencial	Secuencial	Paralelo
Memoria/contexto práctico	Limitada por estado oculto y gradiente	Mejor que RNN simple, pero sigue siendo secuencial	Ventana explícita de atención; su tamaño depende del modelo y del coste
Paralelizable	No	No	Sí (aprovecha GPUs)
Velocidad	Lenta	Lenta	Rápida

El Transformer ganó por dos razones.[9] Primero, **paralelismo**: mira todos los tokens de una ventana a la vez. Una RNN tiene que procesar el token 1 antes que el 2, y el 2 antes que el 3. Un Transformer procesa la ventana simultáneamente, aprovechando GPUs masivamente paralelas. Segundo, **atención directa**: en lugar de comprimir todo el contexto en un estado oculto de tamaño fijo, cada token puede «mirar» directamente a cualquier otro token de la ventana. Si la palabra 500 necesita información de la palabra 1, la atención le da un camino directo. Ese camino no es gratis: la atención estándar crece de forma cuadrática con la longitud de la ventana, por eso los contextos largos requieren ingeniería adicional.

En el día a día

Aunque los Transformers dominan el procesamiento de lenguaje, las CNNs y RNNs siguen siendo relevantes. La decisión de ingeniería rara vez es elegir la familia más moderna; es elegir la que mejor encaja con la estructura del dato, el volumen disponible y el presupuesto de cómputo y latencia.

CNNs en visión. Cuando la tarea es clasificar imágenes, detectar objetos o segmentar y tienes pocos datos etiquetados, poco cómputo o un despliegue en dispositivo, una CNN sigue siendo la primera candidata razonable. El motivo es su sesgo inductivo: asume que los píxeles cercanos están relacionados y que el mismo patrón puede aparecer en cualquier zona, así que reutiliza filtros y necesita muchos menos ejemplos que un modelo sin esa suposición. Esa misma economía de parámetros la hace barata de entrenar y de servir. La restricción que manda aquí es el coste y el volumen de datos. Solo cuando dispones de muchísimos ejemplos, infraestructura de entrenamiento amplia y quieres atención global o unificar visión con lenguaje, un *Vision Transformer* entra en la conversación, porque sin ese volumen de datos su falta de sesgo inductivo se paga en sobreajuste.

CNNs, U-Nets y DiT en generación de imágenes. En generación no eliges una sola familia sino una combinación. Las U-Nets convolucionales han sido el bloque central de los modelos de difusión porque su estructura de codificador y decodificador con atajos preserva el detalle espacial fino que una imagen necesita. Al escalar a más datos y más cómputo, los Transformers de difusión como DiT se vuelven competitivos porque escalan mejor con el tamaño del modelo. La decisión depende del presupuesto de entrenamiento y de la resolución objetivo, así que conviene mirar la arquitectura concreta del modelo que usas en lugar de asumir una por defecto.

LSTMs en series temporales. Para predicción de ventas, mantenimiento predictivo o análisis de señales con secuencias cortas o medias, una LSTM sigue siendo muy competitiva y a menudo preferible. La restricción dominante es la relación entre coste y ganancia: la atención del Transformer crece de forma cuadrática con la longitud de la secuencia, y ese coste solo se justifica si de verdad necesitas relacionar posiciones muy lejanas. Para predecir la demanda de mañana a partir de unos cientos de puntos, un modelo recurrente entrena más rápido, es más barato de mantener y rinde igual o mejor que un Transformer enorme.

RNNs en dispositivos pequeños. Cuando la restricción que manda es la memoria y el consumo, una RNN simple o una GRU consume mucha menos RAM y cómputo que un Transformer, que debe guardar las activaciones de toda la ventana de atención. En un microcontrolador o un dispositivo *edge* con presupuesto de energía ajustado, procesar la señal paso a paso con un estado oculto pequeño puede ser la única opción que cabe en el hardware. Aquí la elegancia secuencial de la RNN, que en lenguaje era un lastre frente al paralelismo, se convierte en su mayor ventaja.

Por qué debería importarte

Estas arquitecturas no son reliquias históricas que estudias por cultura general. Son las piezas que explican **por qué** los Transformers son como son.

Cada decisión de diseño del Transformer es una respuesta a una limitación de las RNNs. La atención existe porque el estado oculto de una RNN no bastaba para contexto largo. El paralelismo existe porque el procesamiento secuencial era demasiado lento. La capacidad de

mirar posiciones lejanas dentro de una ventana explícita existe porque las LSTMs, incluso con compuertas, seguían obligando a que la información viajara paso a paso por la secuencia.

Entender de dónde venimos es entender hacia dónde vamos. Y en ingeniería, a veces la herramienta adecuada no es la más moderna, sino la que mejor se adapta a tus restricciones de recursos.

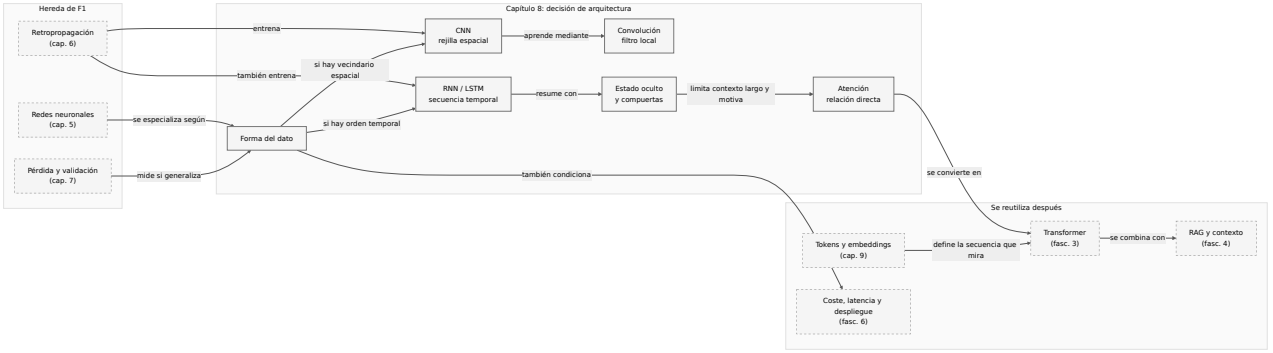
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Usar un Transformer para todo	Para una serie temporal de 100 puntos, una LSTM puede ser más rápida, más barata e igual de precisa que un Transformer. Usar la arquitectura más grande no siempre mejora la decisión.	Evalúa la complejidad de tu problema antes de elegir la arquitectura. Si es secuencial y corto, prueba RNN/LSTM primero.
Aplicar CNNs a datos no espaciales	Las CNNs asumen que los datos tienen estructura local (píxeles cercanos están relacionados). Aplicarlas a datos tabulares donde las columnas no tienen orden espacial no aporta ventajas sobre un MLP.	CNNs para imágenes y datos con estructura de rejilla. MLPs para datos tabulares. Transformers para secuencias largas.
Ignorar el preprocesamiento en CNNs	Una CNN espera imágenes de un tamaño fijo. Si le pasas imágenes de tamaños variados sin redimensionar, los resultados serán inconsistentes.	Normaliza el tamaño de las imágenes, escala los valores de píxel a [0,1] o [-1,1] y aplica *data augmentation* durante el entrenamiento.
Asumir que las RNNs «entienden» el orden	Las RNNs procesan en orden, pero eso no garantiza que capturen dependencias a largo plazo. Una RNN simple puede fallar en capturar que la primera palabra de un párrafo es el sujeto de la última oración.	Si necesitas dependencias a largo plazo, usa LSTMs, GRUs o, directamente, Transformers.

Cómo encaja todo

Este mapa se lee como una decisión de arquitectura, no como una línea histórica. Venimos de redes, gradientes y capas; aquí aprendemos que la forma del dato importa. Si la estructura es espacial, una CNN aprovecha vecindarios. Si la estructura es temporal, una RNN o LSTM arrastra estado. Si la secuencia es larga, necesitamos atención, recuperación o arquitecturas híbridas.

La decisión que enseña el capítulo no es memorizar siglas, sino justificar una familia por contrato de datos, coste y despliegue. Esa misma forma de pensar reaparece cuando elijas embeddings, modelos locales, RAG o arquitecturas de Transformer.


Hereda de F1
Retropropagación (cap. 6)
Redes neuronales (cap. 5)
Pérdida y validación (cap. 7)
Capítulo 8: decisión de arquitectura
entrena
también entrena
se especializa según
mide si generaliza
si hay vecindario espacial
si hay orden temporal
Forma del dato
CNN rejilla espacial
aprende mediante
Convolución filtro local
RNN / LSTM secuencia temporal
resume con
Estado oculto y compuertas
limita contexto largo y motiva
Atención relación directa
también condiciona
se convierte en
Se reutiliza después
Tokens y embeddings (cap. 9)
define la secuencia que mira
Transformer (fasc. 3)
se combina con
RAG y contexto (fasc. 4)
Coste, latencia y despliegue (fasc. 6)

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
CNN	Red neuronal convolucional. Usa filtros que se deslizan por la entrada para detectar patrones locales. Especializada en imágenes y datos con estructura de rejilla.
Convolución	Operación que aplica un filtro sobre una entrada, deslizándolo por todas las posiciones y calculando la similitud en cada punto.
Filtro (kernel)	Matriz pequeña de pesos aprendidos que se desliza por la entrada para detectar un patrón concreto como un borde o una textura.
Mapa de características	Salida que produce un filtro al recorrer toda la entrada; indica dónde aparece el patrón que ese filtro detecta.
Stride	Paso del filtro: cuántos píxeles avanza en cada salto. Un paso mayor reduce la resolución de la salida.
Padding	Relleno de ceros alrededor del borde de la entrada que sirve para conservar el tamaño de la salida tras la convolución.
Pooling	Operación que reduce la resolución espacial manteniendo la información relevante. *Max pooling* toma el valor máximo de cada zona.
RNN	Red neuronal recurrente. Procesa secuencias elemento a elemento, manteniendo un estado oculto que resume el contexto previo.
Estado oculto	Vector que resume la información de todos los elementos anteriores de una secuencia en una RNN.
LSTM	Variante de RNN con compuertas de olvido, entrada y salida. Permite aprender dependencias a más largo plazo que una RNN simple.
GRU	Variante de RNN más ligera que la LSTM, con solo dos compuertas. Entrena más rápido y suele rendir de forma comparable.
ViT (Vision Transformer)	Arquitectura que divide una imagen en parches y los procesa con atención en lugar de convoluciones.
Atención	Mecanismo que permite a cada elemento de una secuencia mirar directamente a cualquier otro dentro de una ventana, sin pasar por un estado recurrente.

Antes de pasar página

- [ ] ¿Puedo explicar cómo una CNN detecta patrones en una imagen? (Si no, vuelve a «CNNs: cómo ve una máquina».)
- [ ] ¿Entiendo qué hace el *pooling* y por qué es útil? (Si no, vuelve a la sección de CNNs.)
- [ ] ¿Sé cómo procesa una RNN una secuencia y cuál es su principal limitación? (Si no, vuelve a «RNNs y LSTMs».)
- [ ] ¿Puedo explicar por qué el Transformer superó a las RNNs? (Si no, vuelve a la tabla comparativa.)
- [ ] ¿Sé justificar una arquitectura por forma de datos, coste y despliegue? (Si no, vuelve a «En el día a día».)

En resumen

IDEA FUERZA	DETALLE
Las CNNs detectan patrones locales en imágenes mediante filtros que se deslizan por la entrada.	De bordes en las primeras capas a objetos completos en las profundas. *Pooling* reduce dimensionalidad y añade invarianza.
Las RNNs procesan secuencias manteniendo un estado oculto. Las LSTMs añaden compuertas para recordar a más largo plazo.	Mejoran el problema, pero siguen comprimiendo historia en estado recurrente y procesando paso a paso.
El Transformer ganó por paralelismo y atención directa.	Procesa una ventana completa a la vez y permite que cualquier token mire directamente a otro dentro de esa ventana, pagando coste de memoria y cómputo.

Para saber más

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. y Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. En *Proceedings of EMNLP* (pp. 1724-1734). https://doi.org/10.3115/v1/D14-1179

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J. y Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. En *International Conference on Learning Representations*. https://openreview.net/forum?id=YicbFdNTTy

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

Hochreiter, S. y Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735

Krizhevsky, A., Sutskever, I. y Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. En *Advances in Neural Information Processing Systems 25* (pp. 1097-1105). https://papers.nips.cc/paper/4824

LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539

Peebles, W. y Xie, S. (2023). Scalable diffusion models with Transformers. En *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 4195-4205). https://doi.org/10.1109/ICCV51070.2023.00387

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

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

Notas

1. Krizhevsky, A., Sutskever, I. y Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. En *Advances in Neural Information Processing Systems 25* (pp. 1097-1105). https://papers.nips.cc/paper/4824. AlexNet demostró que las CNNs profundas, entrenadas con GPUs, podían superar ampliamente a los métodos tradicionales de visión por computador, iniciando la revolución del *deep learning* en visión. ↩

2. LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539. Los autores describen cómo las capas convolucionales aprenden jerarquías de características visuales, desde bordes simples en las primeras capas hasta objetos complejos en las profundas. ↩

3. 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 permiten entrenar redes de más de cien capas sin degradación del rendimiento, resolviendo el problema del desvanecimiento del gradiente en arquitecturas muy profundas. ↩

4. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J. y Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. En *International Conference on Learning Representations*. https://openreview.net/forum?id=YicbFdNTTy. El trabajo popularizó ViT: dividir una imagen en parches y procesarlos con un Transformer. ↩

5. Peebles, W. y Xie, S. (2023). Scalable diffusion models with Transformers. En *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 4195-4205). https://doi.org/10.1109/ICCV51070.2023.00387. DiT muestra cómo sustituir bloques U-Net por Transformers escalables en modelos de difusión. ↩

6. Hochreiter, S. y Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735. Hochreiter y Schmidhuber introdujeron la LSTM para resolver el problema del desvanecimiento del gradiente en RNNs, permitiendo que las redes recurrentes aprendieran dependencias a largo plazo por primera vez. ↩

7. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org. El capítulo 10 aborda en detalle las RNNs, el problema del desvanecimiento del gradiente en secuencias largas y las arquitecturas con compuertas como LSTM y GRU. ↩

8. Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. y Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. En *Proceedings of EMNLP* (pp. 1724-1734). https://doi.org/10.3115/v1/D14-1179. La GRU fue introducida como una alternativa más simple y computacionalmente eficiente a la LSTM, con rendimiento comparable en muchas tareas. ↩

9. 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. Los autores demostraron que eliminar la recurrencia y usar solo atención permitía entrenar modelos más rápido (por el paralelismo) y con mejor calidad (por la capacidad de atender a cualquier posición de la secuencia). ↩

Capítulo 09: Del token al embedding: cómo el modelo representa lenguaje

Entrando en el tema

Escribes «Hola, ¿cómo estás?» y pulsas Enviar. Para ti son cuatro palabras, un signo de interrogación y una intención. Para el modelo, son números. Solo números.

Este capítulo explica el viaje que transforma tu texto en algo que una red neuronal puede procesar. Tiene dos etapas: la **tokenización** (trocear el texto en unidades procesables) y el **embedding** (convertir cada unidad en un vector de números que representa regularidades semánticas y de uso). Si el capítulo 4 fue el átomo (la neurona) y el capítulo 5 la molécula (la red), este capítulo es el sistema de coordenadas que permite operar con lenguaje sin que el modelo vea letras como las vemos nosotros.

Qué es un token

Un token es la **unidad mínima** que el modelo procesa. No es una palabra, aunque a veces coincide. No es un carácter, aunque a veces también. Es lo que el tokenizador (un programa que trocea texto) decide que es la unidad óptima para ese modelo.[1]

Algunos ejemplos con un tokenizador típico:

```
"Hola mundo"       → ["Hola", " mundo"]           (2 tokens)
"desarrolladores"  → ["des", "arrol", "ladores"]   (3 tokens)
"function get()"   → ["function", " get", "()"]   (3 tokens)
```

Observa tres cosas. Primero, los espacios importan: «mundo» y « mundo» son tokens distintos. El espacio indica que la palabra empieza tras otra. Segundo, las palabras largas o poco frecuentes se trocean en subtokens: «desarrolladores» se divide en tres piezas que el modelo puede recombinar. Tercero, el código se tokeniza de forma distinta al lenguaje natural: los paréntesis, llaves y operadores son tokens independientes.

El modelo no ve texto. Ve secuencias de números (IDs de token). Cada token tiene un identificador único en el vocabulario del modelo (típicamente entre 50 000 y 250 000 tokens distintos). Cuando escribes «Hola mundo», el modelo recibe algo como `[15496, 2159]`. Nunca ve las letras.

Todo se mide en tokens. La ventana de contexto (cuánto texto puede procesar el modelo de una vez), el precio de la API, el límite de respuesta: todo se mide en tokens. Es la moneda de la IA generativa.

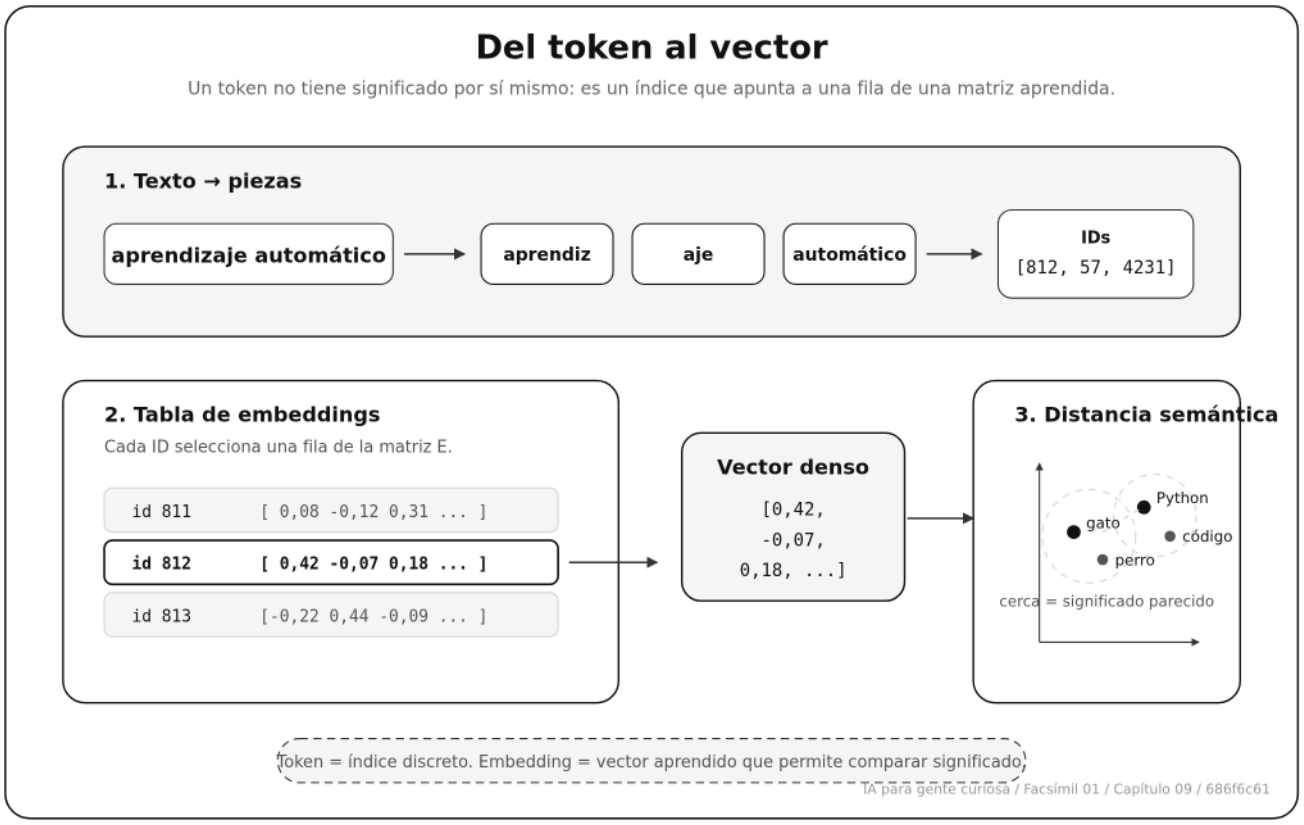
Cómo funciona la tokenización (BPE)

El algoritmo más usado se llama **Byte Pair Encoding** (BPE).[2] Funciona así:

1. Empieza con un vocabulario de caracteres individuales (a, b, c, ..., espacio, etc.).
2. Recorre todo el texto de entrenamiento y encuentra el **par de tokens consecutivos más frecuente**.
3. Fusiona ese par en un nuevo token y lo añade al vocabulario.
4. Repite miles de veces.

El resultado es un vocabulario donde las palabras muy frecuentes («el», «de», «y») son tokens únicos, mientras que las palabras raras o largas se descomponen en subtokens. «Desarrolladores» acaba siendo «des» + «arrol» + «ladores» porque esos fragmentos aparecen en muchas otras palabras («desarrollo», «arrollar», «desarrollado»). Es un equilibrio ingenioso: el vocabulario es manejable (50k-250k tokens) pero puede representar cualquier palabra.

El español gasta más tokens que el inglés. Como regla aproximada, un token equivale a 3-4 caracteres en texto latino. Pero el español, con sus tildes, sus conjugaciones verbales y sus palabras más largas, tiende a consumir más tokens que el inglés para expresar lo mismo. «Machine learning is great» son 4 tokens; «El aprendizaje automático es genial» son 7-8.



Qué es un embedding

Una vez que el texto se ha convertido en tokens, cada token necesita una representación numérica que la red pueda procesar. Esa representación es el **embedding**: un vector denso de números reales (típicamente entre 768 y 4096 dimensiones) que captura el significado del token en el contexto del modelo.[3]

«Denso» significa que casi todas las dimensiones tienen valores distintos de cero. Un vector *sparse* (disperso) tendría muchos ceros y unas pocas dimensiones activas. Un embedding es denso: cada dimensión contribuye un poco al significado global.

Las dimensiones no tienen etiquetas. No hay una dimensión que signifique «animalidad» y otra que signifique «tamaño». Cada dimensión es una coordenada en un espacio de alta dimensionalidad, y el significado emerge de la posición relativa del vector completo. Es como preguntar «¿qué significa la coordenada X = 0,23 en un mapa?». Nada por sí sola. Pero combinada con Y = -0,87 y las otras 1534 dimensiones, sitúa «gato» cerca de «felino» y lejos de «JavaScript».

```
"gato"       → [0.23, -0.87, 0.45, 0.12, ...]  (1536 dimensiones)
"felino"     → [0.21, -0.85, 0.48, 0.11, ...]  (muy cercano a "gato")
"JavaScript" → [-0.56, 0.33, -0.12, 0.78, ...] (completamente distinto)
```

Piensa en un mapa donde las palabras se colocan según su significado. «Perro» y «gato» están cerca (ambos son animales domésticos). «Python» y «JavaScript» están cerca (ambos son lenguajes de programación). Pero «perro» y «Python» están lejos. El embedding es la coordenada de cada palabra en ese mapa de miles de dimensiones.[4]

Embeddings en la práctica

Aritmética de significado

Una propiedad fascinante de los embeddings es que las relaciones semánticas a veces se expresan como operaciones vectoriales. Si el espacio de embeddings captura consistentemente las dimensiones de significado, las direcciones en ese espacio codifican relaciones:

```
rey - hombre + mujer ≈ reina
Madrid - España + Francia ≈ París
```

En notación vectorial, esto es literalmente sumar y restar los embeddings y buscar la palabra cuyo vector queda más cerca del resultado:

$$\mathbf{v}_{\text{rey}} - \mathbf{v}_{\text{hombre}} + \mathbf{v}_{\text{mujer}} \approx \mathbf{v}_{\text{reina}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\mathbf{v}_{\text{palabra}}$	El embedding (vector) de esa palabra	el vector de «rey»
$- \mathbf{v}_{\text{hombre}}$	Quitar la dirección «masculino»	restar el vector de «hombre»
$+ \mathbf{v}_{\text{mujer}}$	Añadir la dirección «femenino»	sumar el vector de «mujer»
$\approx$	El resultado cae cerca de	el vector de «reina»

En palabras: si a «rey» le restas «hombre» te queda algo parecido a la idea de «realeza», y al sumarle «mujer» llegas cerca de «reina». La dirección que separa «hombre» de «mujer» es parecida a la que separa «rey» de «reina».

Estas analogías, popularizadas por Word2Vec,[5] son ilustrativas, no garantizadas. Un embedding moderno no siempre produce estas operaciones limpiamente, pero el principio subyacente, que la dirección en el espacio codifica relaciones, es real y es la base de la búsqueda semántica.

Similitud semántica

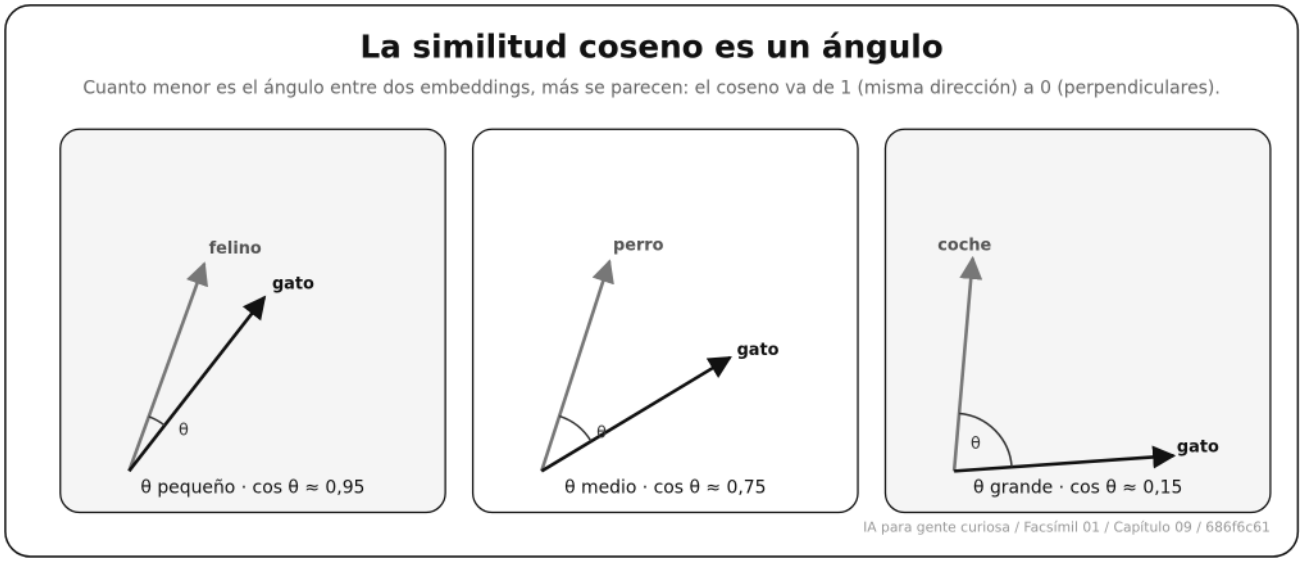
La medida estándar para comparar embeddings es la **similitud coseno**: el coseno del ángulo entre dos vectores. Vale 1 si apuntan exactamente en la misma dirección (idénticos), 0 si son perpendiculares (sin relación) y -1 si son opuestos. Su fórmula es el producto escalar de los dos vectores dividido por el producto de sus normas (sus longitudes):

$$\mathrm{sim}(\mathbf{a}, \mathbf{b}) = \cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\,\|\mathbf{b}\|} = \frac{\sum_i a_i b_i}{\sum_i a_i^2 \quad \sum_i b_i^2}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\mathbf{a}$, $\mathbf{b}$	Los dos embeddings que comparas	el vector de «gato» y el de «felino»
$\mathbf{a} \cdot \mathbf{b}$	Producto escalar: suma de los productos componente a componente	$\sum_i a_i b_i$
$\|\mathbf{a}\|$	Norma (longitud) del vector, $\sum_i a_i^2$	cuánto «mide» el vector
a_i, b_i	La componente i de cada vector	la dimensión 23 de cada embedding
θ	Ángulo entre los dos vectores	pequeño si apuntan parecido

En palabras: el coseno mide si dos vectores apuntan en la misma dirección, sin importar su longitud. Por eso se prefiere a la distancia euclídea cuando solo interesa el significado y no la magnitud. Un detalle práctico: si los vectores están normalizados (norma 1), el denominador vale 1 y la similitud coseno se reduce al producto escalar, más barato de calcular. Por eso muchos índices vectoriales normalizan los embeddings de antemano.

PAR DE PALABRAS	SIMILITUD	INTERPRETACIÓN
«gato» / «felino»	Alta (~0.95)	Cuasi-sinónimos
«gato» / «perro»	Media (~0.75)	Misma categoría
«gato» / «coche»	Baja (~0.15)	Sin relación
«deploy» / «desplegar»	Alta (~0.90)	Traducción (modelo multilingüe)



La clave: conceptos similares quedan **cerca** en el espacio vectorial. Esto permite buscar por significado, no por palabras exactas. «¿Cómo despliego en producción?» encuentra documentos sobre «deploy to prod» aunque no compartan ni una sola palabra.

Modelos de embedding (2026)

Fecha de corte: 10 de junio de 2026. Esta tabla no es un ranking universal: es una foto de familias útiles para entender decisiones de ingeniería. Antes de elegir proveedor hay que comprobar licencia, coste, límites, idioma, evaluación propia y si el índice vectorial ya existente obliga a re-embedar documentos.

MODELO	PROVEEDOR	DIMENSIONES DECLARADAS	QUÉ APORTA REALMENTE
`text-embedding-3-large`	OpenAI	3072 por defecto; reducible con `dimensions`	API madura y buen punto de partida general. Reducir dimensiones ahorra almacenamiento, pero hay que medir recuperación en tus datos.[6]
`voyage-4-large`	Voyage AI	1024 por defecto; 256, 512, 1024 o 2048	Familia orientada a recuperación y RAG multilingüe con dimensiones Matryoshka y compatibilidad dentro de la serie Voyage 4.[7]
`Qwen3-Embedding-8B`	Qwen (open weights)	hasta 4096; salida configurable de 32 a 4096	Opción autoalojable para equipos que necesitan control de pesos, idiomas y despliegue propio, a cambio de gestionar hardware y rendimiento.[8]
`BGE-M3`	BAAI (open weights)	1024	Modelo abierto multilingüe útil para búsqueda híbrida (densa, dispersa y multi-vector) y entornos donde se quiere controlar el índice y el runtime.[9] Conviene evaluarlo frente al dominio concreto antes de asumir que “open” significa suficiente.
`embed-v4`	Cohere	1536 por defecto; 256, 512, 1024 o 1536	Embeddings multimodales de texto e imagen, útiles para documentos visuales, PDFs, catálogos o búsqueda mixta.[10]

Importante: el modelo de embedding convierte entradas en vectores para comparar, ordenar o recuperar. No genera texto. Es distinto del LLM generativo. En un RAG típico, el embedding sirve para **buscar** fragmentos relevantes; el LLM sirve para **redactar** una respuesta usando esos fragmentos. Confundir ambas piezas lleva a arquitecturas caras y difíciles de depurar.

Cómo se entrenan los embeddings

El objetivo de entrenamiento es simple y brillante: dadas dos palabras que aparecen cerca en un texto, sus embeddings deben ser similares. Dadas dos palabras que no aparecen juntas, sus embeddings deben ser distintos.[11] El modelo aprende de millones de frases: si «gato» y «felino» aparecen en contextos similares («el _ _ _ _ duerme», «acaricié al _ _ _ _»), sus vectores se acercan. Si «gato» y «tornillo» nunca comparten contexto, sus vectores se alejan.

Este principio, «dime con qué palabras apareces y te diré qué significas», se llama **hipótesis distribucional** y es la base de casi todos los embeddings modernos. No hace falta que nadie etiquete datos: el propio texto proporciona la señal de aprendizaje. Es aprendizaje auto-supervisado a escala masiva.

Embeddings Matryoshka: menos dimensiones sin perder calidad

Una innovación reciente son los **embeddings Matryoshka** (MRL, *Matryoshka Representation Learning*).[1][2] La idea: entrenar el embedding para que funcione bien **a cualquier dimensionalidad**. Puedes tomar las primeras 256 dimensiones de un embedding de 1024 y obtener una representación de calidad razonable, o usar las 1024 completas para máxima precisión. Es como tener varios embeddings de distinto tamaño dentro del mismo vector, como las muñecas rusas que le dan nombre.

Esto resuelve un problema práctico enorme: no necesitas elegir entre precisión y coste de antemano. Usas 256 dimensiones para búsquedas rápidas y baratas, y 1024 para cuando necesitas la máxima calidad. Modelos como voyage-4-large, Qwen3-Embedding y Cohere embed-v4 ya lo implementan.

En el día a día

Búsqueda semántica y RAG. Es el uso más extendido y el que mejor enseña la cadena completa. Primero troceas tus documentos en fragmentos manejables porque el contexto del LLM es limitado y un párrafo enfocado recupera mejor que un manual entero. Pasas cada fragmento por el modelo de embedding y guardas el vector resultante junto al texto original en una base de datos vectorial como Pinecone, Chroma o pgvector. Cuando llega una pregunta, conviertes esa pregunta al mismo espacio vectorial con el mismo modelo y pides al índice los fragmentos cuya similitud coseno es mayor. Esos fragmentos se inyectan en el prompt para que el LLM redacte una respuesta apoyada en tu documentación y no en lo que recuerde de su entrenamiento. La decisión de ingeniería crítica es que la consulta y los documentos deben compartir modelo y normalización: si reindexas con otro modelo, tienes que recalcular todos los vectores. Conviene afinar el tamaño del fragmento, el número de resultados que recuperas y, si la precisión no basta, añadir un reordenador que repase los candidatos.

Clasificación. Puedes agrupar tickets de soporte, correos o reseñas por similitud semántica sin redactar reglas ni definir categorías a mano. Calculas el embedding de cada texto y agrupas los vectores cercanos, o comparas cada entrada nueva con vectores de ejemplo ya etiquetados y le asignas la etiqueta del más parecido. La pieza que interviene es siempre la misma medida de distancia en el espacio vectorial, pero la decisión está en elegir un modelo de embedding alineado con tu dominio e idioma, porque un modelo entrenado para inglés general clasificará mal jerga médica en español. Lo que hay que cuidar es validar con datos reales dónde poner el umbral que separa una categoría de otra, ya que ese corte determina cuántos casos quedan mal asignados.

Detección de duplicados. Comparas embeddings para encontrar textos que dicen lo mismo con palabras distintas, algo que una comparación literal o un hash nunca detectarían. Sirve para deduplicar una base de conocimiento, detectar reseñas falsas repetidas o evitar incidencias que ya existen. La mecánica es calcular la similitud coseno entre pares de vectores y marcar como sospechosos los que superan un umbral alto. Lo delicado es fijar ese

umbral con cuidado, porque demasiado bajo fusiona textos parecidos pero distintos y demasiado alto deja pasar duplicados reescritos. A escala grande no comparas todos los pares uno a uno sino que usas el índice vectorial para traer solo los vecinos más cercanos de cada texto.

Traducción y multilingüe. Los modelos multilingües colocan «deploy» cerca de «desplegar» y «perro» cerca de «dog» en el mismo espacio vectorial, porque aprendieron a representar el significado por encima del idioma. Esto permite buscar en español sobre una base de documentos en inglés sin traducir nada de antemano: la consulta y los documentos caen cerca si comparten significado aunque no compartan ni una palabra. La decisión de ingeniería es elegir un modelo entrenado de verdad para varios idiomas y no uno monolingüe, y comprobar que rinde en el par de idiomas que te importa. Hay que cuidar que la calidad del alineamiento varía según los idiomas, así que conviene evaluar con consultas reales antes de confiar en que la búsqueda cruzada funciona en tu caso.

Por qué debería importarte

Los embeddings son el pegamento que conecta el mundo del texto con el mundo de las matemáticas. Sin ellos, una red neuronal no podría procesar lenguaje: solo sabe multiplicar matrices de números.

Cada vez que usas un LLM, hay embeddings trabajando: los tokens de entrada se convierten en embeddings antes de entrar al Transformer. Cada vez que usas búsqueda semántica, hay embeddings trabajando: la consulta y los documentos se comparan en el espacio vectorial. Son ubicuos y silenciosos. Entenderlos te permite elegir el modelo adecuado, ajustar la dimensionalidad según tu presupuesto y depurar por qué tu búsqueda semántica devuelve resultados incoherentes.

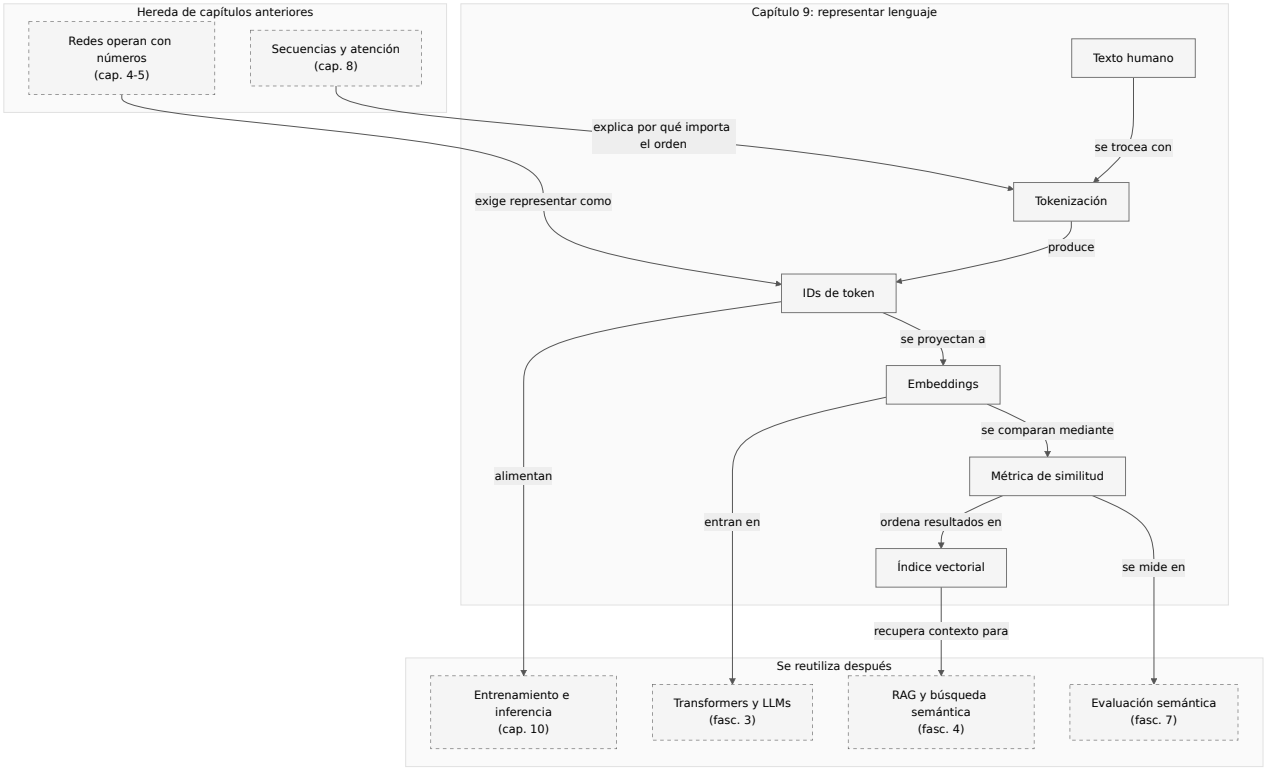
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Confundir token con palabra	«Hola mundo» son dos palabras, pero con BPE pueden ser 2-3 tokens. «Desarrolladores» es una palabra pero 3-4 tokens. Planificar costes en palabras te dará sorpresas en la factura.	Mide en tokens, no en palabras. Usa el tokenizador del proveedor para contar.
Tratar la métrica vectorial como una verdad universal	Coseno, producto escalar y distancia euclídea pueden comportarse distinto según normalización, modelo e índice. Decir “euclídea nunca sirve” es tan pobre como usarla sin pensar.	Lee la recomendación del modelo y de la base vectorial. Si los vectores están normalizados, coseno y producto escalar pueden ser equivalentes. Evalúa la métrica con consultas reales.
Usar el mismo modelo de embedding para todo	Un modelo optimizado para inglés puede rendir mal en español jurídico. Un modelo de código no sirve para reseñas de restaurantes.	Elige el modelo de embedding según tu dominio, idioma y tipo de contenido.
Asumir que más dimensiones siempre es mejor	Más dimensiones permiten más matices, pero también más memoria, más latencia y más coste de almacenamiento.	Evalúa con tus datos. A veces 256 dimensiones bastan; a veces necesitas 3072.

Cómo encaja todo

Este mapa se lee como una tubería de representación. Venimos de neuronas, capas y arquitecturas que solo operan con números; este capítulo explica cómo el texto entra en ese mundo numérico sin fingir que el modelo “lee” como una persona. La decisión técnica nueva es doble: elegir cómo troceas el texto y elegir qué espacio vectorial usarás para comparar significado.

La consecuencia aparece después en casi todo el facsímil. Entrenar e inferir necesitan tokens; RAG necesita embeddings y métricas de similitud; evaluar sistemas semánticos exige saber cuándo una búsqueda recupera por significado y cuándo solo recupera ruido con buena pinta.


Hereda de capítulos anteriores
Redes operan con números (cap. 4-5)
Secuencias y atención (cap. 8)
Capítulo 9: representar lenguaje
Texto humano
explica por qué importa el orden
se trocea con
exige representar como
Tokenización
produce
IDs de token
se proyectan a
Embeddings
se comparan mediante
alimentan
Métrica de similitud
entran en
ordena resultados en
Índice vectorial
se mide en
recupera contexto para
Se reutiliza después
Entrenamiento e inferencia (cap. 10)
Transformers y LLMs (fasc. 3)
RAG y búsqueda semántica (fasc. 4)
Evaluación semántica (fasc. 7)

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Token	Unidad mínima de texto procesada por el modelo. Puede ser palabra, subtoken o carácter.
Tokenizador	Programa que convierte texto en secuencias de tokens según un vocabulario predefinido.
BPE (Byte Pair Encoding)	Algoritmo de tokenización que parte de caracteres y fusiona los pares más frecuentes hasta formar un vocabulario de subpalabras.
Subtoken	Fragmento de palabra en que se descompone un término largo o poco frecuente para poder representarlo con piezas reutilizables.
Embedding	Vector denso de números reales que representa un token, frase o documento en un espacio semántico de alta dimensionalidad.
Similitud coseno	Medida de similitud entre vectores basada en el coseno de su ángulo. Es muy habitual en embeddings, pero debe validarse con la recomendación del modelo y del índice.
Producto escalar	Suma de los productos componente a componente de dos vectores, numerador de la similitud coseno.
Norma	Longitud de un vector, calculada como la raíz de la suma de sus componentes al cuadrado.
Hipótesis distribucional	Idea de que el significado de una palabra se deduce de las palabras con las que suele aparecer.
Aprendizaje auto-supervisado	Entrenamiento en el que la señal de aprendizaje sale del propio texto, sin necesidad de datos etiquetados a mano.
Embeddings Matryoshka	Embeddings entrenados para funcionar bien a varias dimensionalidades, permitiendo recortar el vector para equilibrar precisión y coste.
Base de datos vectorial	Sistema de almacenamiento optimizado para buscar vectores cercanos por similitud.

Antes de pasar página

- [ ] ¿Puedo explicar la diferencia entre token y palabra con un ejemplo? (Si no, vuelve a «Qué es un token».)
- [ ] ¿Entiendo qué es un embedding y por qué las dimensiones no tienen etiquetas? (Si no, vuelve a «Qué es un embedding».)

- [ ] ¿Sé qué mide la similitud coseno? (Si no, vuelve a «Similitud semántica».)
- [ ] ¿Puedo nombrar al menos tres usos prácticos de los embeddings? (Si no, vuelve a «En el día a día».)
- [ ] ¿Sé explicar por qué dos textos parecidos pueden tener tokens distintos? (Si no, vuelve a «Qué es un token».)

En resumen

IDEA FUERZA	DETALLE
Un token es la unidad mínima de texto para el modelo.	No es una palabra: «desarrolladores» son 3 tokens. Todo se mide en tokens: contexto, precio, límites.
Un embedding es un vector denso que representa regularidades semánticas y de uso.	Cientos o miles de números sitúan cada concepto en un espacio vectorial. Conceptos usados de forma parecida tienden a quedar cerca; conceptos distintos, lejos.
Los embeddings son el pegamento entre texto y matemáticas.	Sin ellos, las redes neuronales no podrían procesar lenguaje. Con ellos, puedes buscar por significado, no por palabras exactas.

Para saber más

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

Chen, J., Xiao, S., Zhang, P., Luo, K., Lian, D. y Liu, Z. (2024). BGE M3-Embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. arXiv:2402.03216. https://arxiv.org/abs/2402.03216

Cohere. (2025). *Announcing Embed Multimodal v4*. https://docs.cohere.com/changelog/embed-multimodal-v4

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

Kusupati, A., Bhatt, G., Rege, A., Wallingford, M., Sinha, A., Ramanujan, V., Howard-Snyder, W., Chen, K., Kakade, S., Jain, P. y Farhadi, A. (2022). Matryoshka representation learning. En *Advances in Neural Information Processing Systems 35*. https://arxiv.org/abs/2205.13147

Mikolov, T., Chen, K., Corrado, G. y Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv:1301.3781. https://arxiv.org/abs/1301.3781

OpenAI. (2026). *Vector embeddings*. https://developers.openai.com/api/docs/guides/embeddings

Qwen. (2025). *Qwen3-Embedding-8B*. Hugging Face. https://huggingface.co/Qwen/Qwen3-Embedding-8B

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

Sennrich, R., Haddow, B. y Birch, A. (2016). Neural machine translation of rare words with subword units. En *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* (pp. 1715-1725). https://doi.org/10.18653/v1/P16-1162

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

Voyage AI. (2026). *Text embeddings*. https://docs.voyageai.com/docs/embeddings

---

Notas

1. Brown, T. B. y otros (2020). Language models are few-shot learners. En *Advances in Neural Information Processing Systems 33* (pp. 1877-1901). https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html. GPT-3 usó tokenización por pares de bytes (BPE), que equilibra el tamaño del vocabulario con la capacidad de representar palabras raras como secuencias de subtokens. ↩

2. Sennrich, R., Haddow, B. y Birch, A. (2016). Neural machine translation of rare words with subword units. En *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* (pp. 1715-1725). https://doi.org/10.18653/v1/P16-1162. BPE se popularizó como método de tokenización por subpalabras, resolviendo el problema de las palabras fuera de vocabulario en traducción automática y siendo adoptado posteriormente por GPT y la mayoría de LLMs. ↩

3. Mikolov, T., Chen, K., Corrado, G. y Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv:1301.3781. https://arxiv.org/abs/1301.3781. Word2Vec demostró que las representaciones vectoriales densas de palabras capturan relaciones semánticas, permitiendo operaciones como «rey - hombre + mujer ≈ reina». ↩

4. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org. El capítulo 12 aborda las representaciones distribuidas y cómo los embeddings densos aprendidos permiten a los modelos capturar relaciones semánticas complejas. ↩

5. Mikolov, T., Chen, K., Corrado, G. y Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv:1301.3781. https://arxiv.org/abs/1301.3781. Las analogías semánticas vectoriales fueron una de las demostraciones más impactantes de Word2Vec, mostrando que los embeddings capturan relaciones sistemáticas entre conceptos. ↩

6. OpenAI. (2026). *Vector embeddings*. https://developers.openai.com/api/docs/guides/embeddings. La documentación indica 1536 dimensiones para `text-embedding-3-small`, 3072 para `text-embedding-3-large` y soporte para reducir dimensiones con el parámetro `dimensions`. ↩

7. Voyage AI. (2026). *Text embeddings*. https://docs.voyageai.com/docs/embeddings. La documentación de Voyage 4 declara contexto de 32 000 tokens y dimensiones 256, 512, 1024 y 2048 para `voyage-4-large`. ↩

8. Qwen. (2025). *Qwen3-Embedding-8B*. Hugging Face. https://huggingface.co/Qwen/Qwen3-Embedding-8B. La ficha declara 100+ idiomas, contexto de 32k y dimensiones configurables hasta 4096. ↩

9. Chen, J., Xiao, S., Zhang, P., Luo, K., Lian, D. y Liu, Z. (2024). BGE M3-Embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. arXiv:2402.03216. https://arxiv.org/abs/2402.03216. El artículo presenta BGE-M3 como modelo multilingüe, multifuncional y de granularidad variable, con entradas hasta 8192 tokens. ↩

10. Cohere. (2025). *Announcing Embed Multimodal v4*. https://docs.cohere.com/changelog/embed-multimodal-v4. Cohere declara dimensiones Matryoshka 256, 512, 1024 y 1536, contexto de 128k y soporte multimodal. ↩

11. Mikolov, T., Chen, K., Corrado, G. y Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv:1301.3781. https://arxiv.org/abs/1301.3781. Word2Vec popularizó dos arquitecturas: skip-gram (predecir contexto a partir de palabra) y CBOW (predecir palabra a partir de contexto), demostrando que ambas producen embeddings de alta calidad. ↩

12. Kusupati, A., Bhatt, G., Rege, A., Wallingford, M., Sinha, A., Ramanujan, V., Howard-Snyder, W., Chen, K., Kakade, S., Jain, P. y Farhadi, A. (2022). Matryoshka representation learning. En *Advances in Neural Information Processing Systems 35*. https://arxiv.org/abs/2205.13147. MRL

entrena embeddings para que sean útiles a múltiples dimensionalidades, permitiendo elegir entre precisión y coste sin reentrenar. ↩

Capítulo 10: Entrenamiento frente a inferencia: dos mundos distintos

Entrando en el tema

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

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

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

Entrenamiento vs inferencia

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

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

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

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

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

Las tres fases del entrenamiento

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

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

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

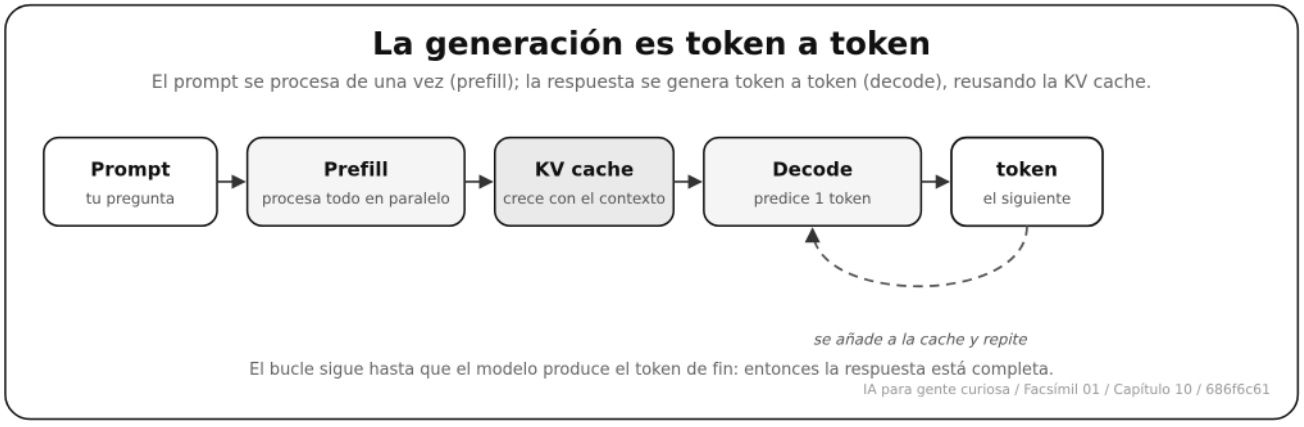
Fase 3: Alineamiento (RLHF/DPO). Personas comparan pares de respuestas del modelo y eligen cuál es mejor. Esa señal de preferencia se usa para refinar el comportamiento: ser útil pero no peligroso, admitir cuando no sabe algo, rechazar peticiones inapropiadas. Es la fase más barata y la que marca la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

Cómo funciona por dentro

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

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

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



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

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

Programación clásica vs machine learning

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

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

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

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

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

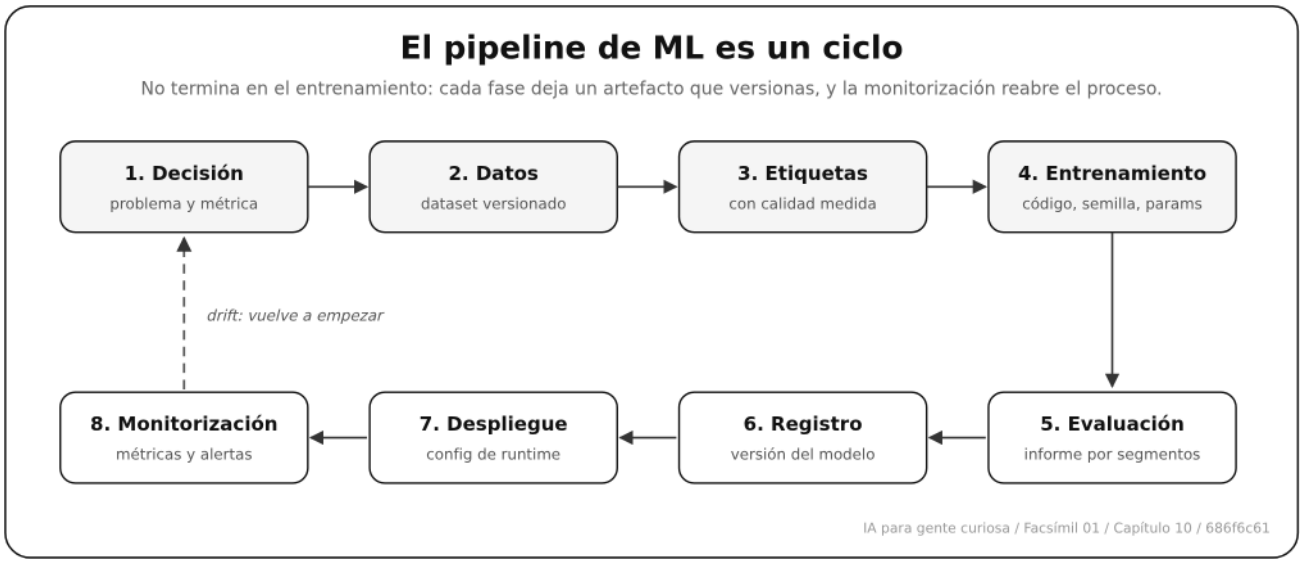
El riesgo principal: un *bug* en ML puede vivir en el *dataset,* no en el código. El modelo puede aprender atajos espurios (correlaciones que funcionan en entrenamiento pero fallan en producción) y tu código compilará perfectamente mientras el modelo toma decisiones incorrectas.

El pipeline: de los datos a producción

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

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

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



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

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

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

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

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

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

Entrenar no es desplegar

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

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

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

Cuantización: modelos que caben en tu portátil

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

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

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

EJEMPLO, NO NOTACIÓN OFICIAL.

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

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

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

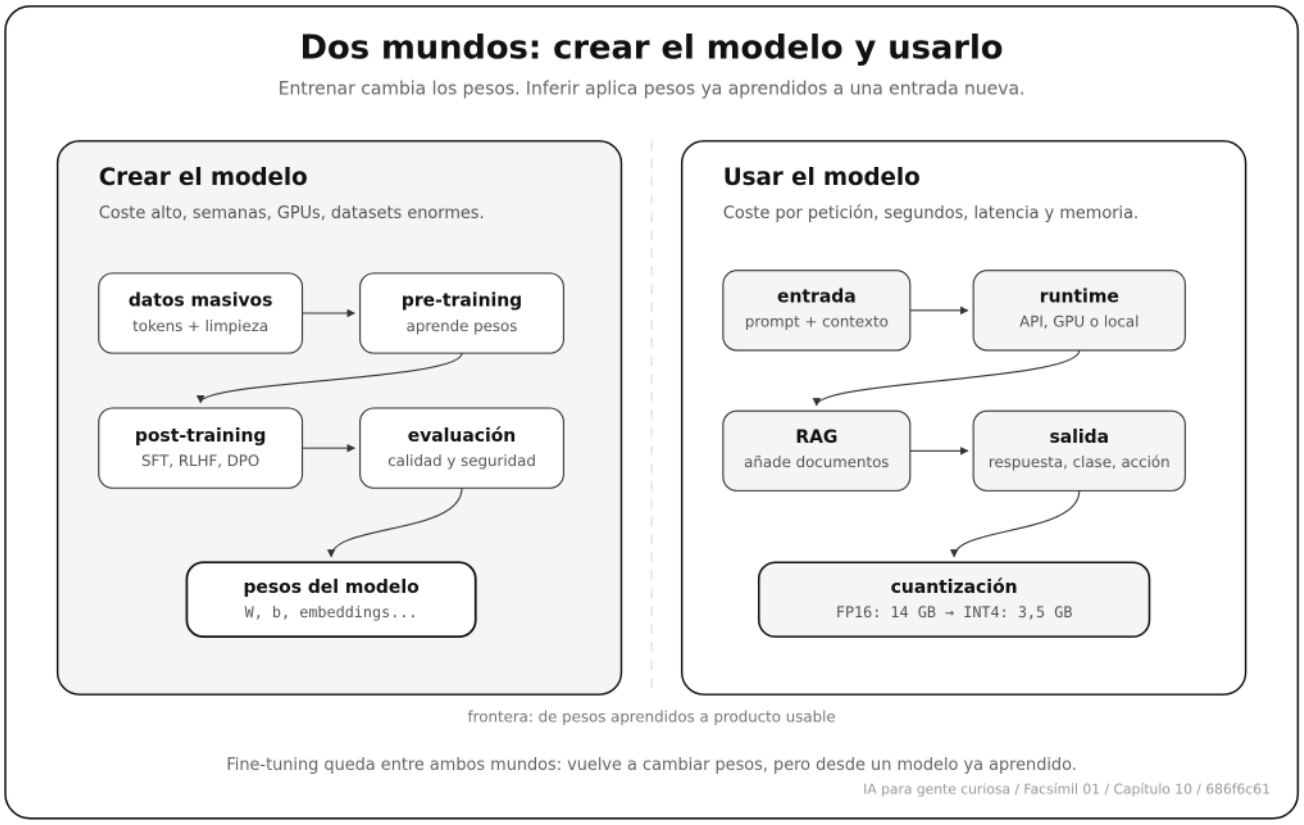
En palabras: multiplicas el número de parámetros por los bytes de cada uno. Un modelo de 7B en FP16 son 7.000 millones × 2 bytes = 14 GB; a 4 bits (medio byte) bajan a 3,5 GB.

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

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

Formatos comunes:

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



En el día a día

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

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

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

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

producción, tampoco podrás depurarlo cuando falle.

Por qué debería importarte

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

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

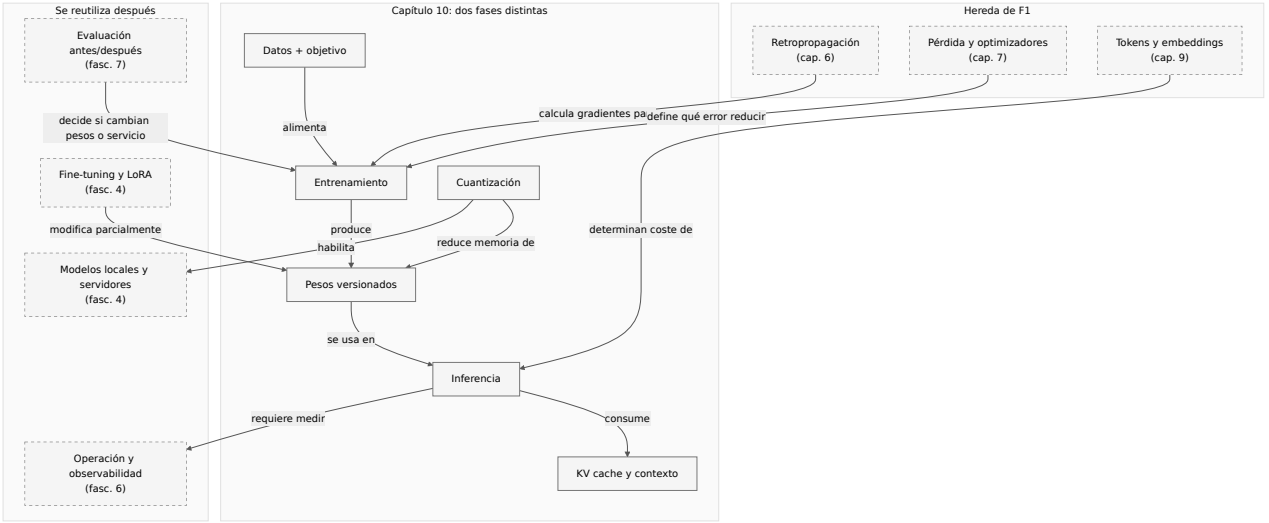
Dónde solía tropezar yo

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

Cómo encaja todo

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

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



Vocabulario aprendido

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

Antes de pasar página

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

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

En resumen

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

Para saber más

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

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

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

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

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

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

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

Notas

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

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

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

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

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

Cuadernos para practicar

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

Entrenamiento vs inferencia: el sobreajuste, en directo

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

Capítulo 11: Machine learning clásico: el mapa antes de los LLM

Entrando en el tema

Hemos pasado diez capítulos construyendo los cimientos: neuronas, redes, retropropagación, embeddings. Todo eso es *deep learning*. Pero antes del *deep learning*, y todavía hoy para la mayoría de problemas del mundo real, existe el *machine learning* clásico.

No es un primo pobre. Es un conjunto de herramientas más simples, más rápidas y más interpretables que resuelven la mayoría de problemas que te encontrarás. Clasificar correos, predecir ventas, detectar fraude, agrupar clientes: todo esto se hacía con ML clásico décadas antes de que existiera ChatGPT. Y se sigue haciendo.

Este capítulo es tu mapa. No te convertirá en experto en cada técnica, pero te dará el vocabulario para saber qué buscar cuando te enfrentes a un problema real.

El paisaje del ML clásico

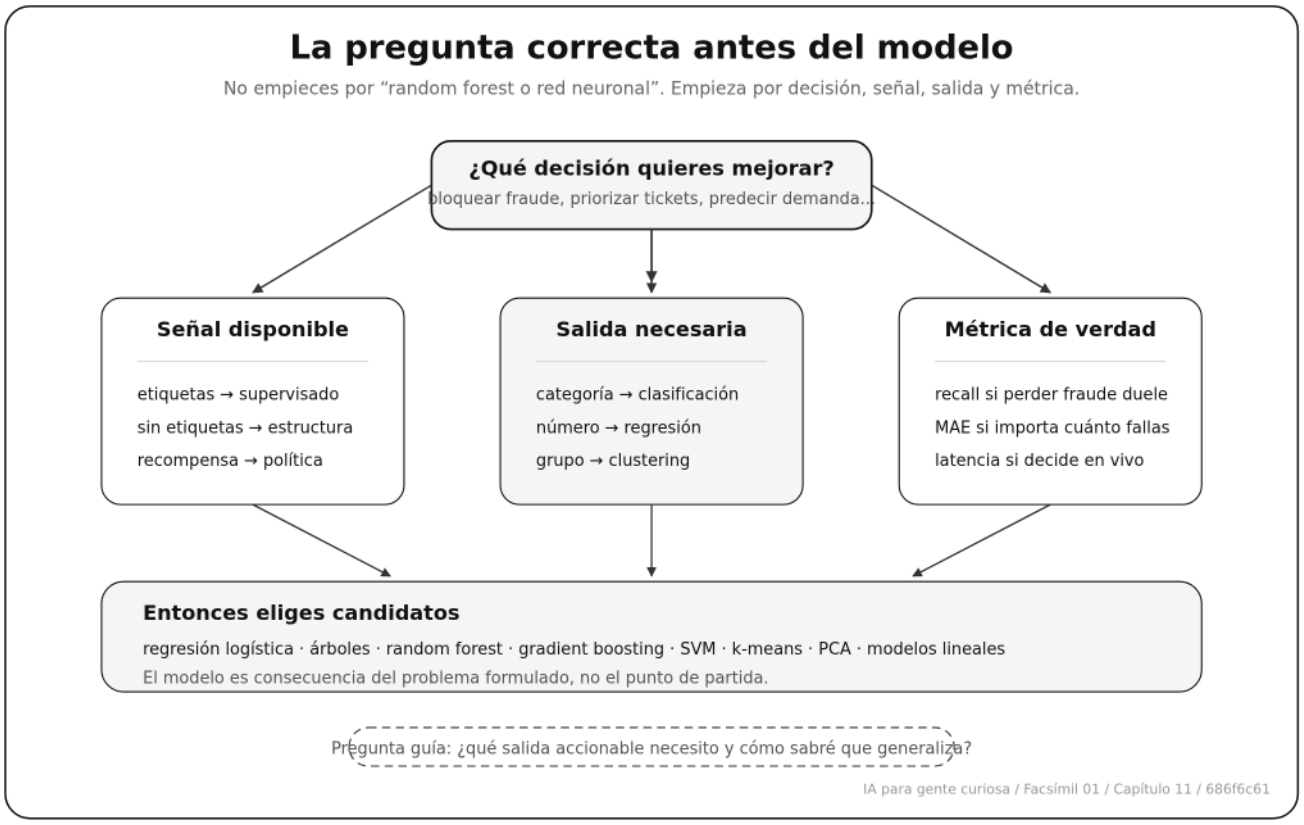
El *machine learning* se organiza alrededor de una pregunta fundamental: ¿qué tipo de señal de aprendizaje tienes?[1]

Supervisado. Tienes ejemplos etiquetados: cada entrada viene con su respuesta correcta. «Este *email* es *spam*», «esta imagen contiene un gato», «esta transacción es fraudulenta». El modelo aprende a imitar esas etiquetas. Es el paradigma más común y el que mejor funciona cuando tienes datos etiquetados de calidad.

No supervisado. No tienes etiquetas. Buscas estructura oculta: agrupar clientes por comportamiento, detectar anomalías, reducir la dimensionalidad de tus datos. El modelo encuentra patrones, pero tú tienes que interpretarlos. Que el algoritmo encuentre tres grupos no significa que esos tres grupos signifiquen algo útil para tu negocio.

Refuerzo. No tienes ejemplos etiquetados, pero sí una señal de recompensa. Un agente actúa en un entorno, recibe *feedback* y aprende a maximizar la recompensa acumulada. Es el paradigma de los juegos, la robótica y, cada vez más, del post-entrenamiento de LLMs (RLHF).

Semi-supervisado y auto-supervisado. Variantes híbridas. El semi-supervisado combina unos pocos ejemplos etiquetados con muchos sin etiquetar. El auto-supervisado genera sus propias etiquetas a partir de los datos: predecir la siguiente palabra en un texto, predecir la zona recortada de una imagen. Es el paradigma que hizo posibles los LLMs.



Anatomía de un dataset

Antes de elegir algoritmo, necesitas entender tus datos.[2]

- **Instancia (fila):** un ejemplo concreto. Un cliente, una transacción, una imagen, un *email*.
- **Feature (columna, variable, atributo):** una característica medible de la instancia. Edad, importe, número de caracteres, color promedio.
- **Label / target (etiqueta):** lo que quieres predecir. «Fraude / No fraude», «150 000 €», «Categoría A».
- **Dataset:** el conjunto de instancias con sus *features* y, si es supervisado, sus etiquetas.

Regla práctica: si no puedes explicar qué representa cada fila, qué significa cada columna y qué decisión habilita la predicción, no tienes un problema de ML. Tienes datos sueltos. Resolver esa ambigüedad es el primer paso de cualquier proyecto.

Clasificación vs regresión

La primera decisión técnica es el tipo de salida.[3]

Clasificación. La salida es una categoría discreta. «*Spam* / No *spam*», «Gato / Perro / Pájaro», «Fraude / No fraude». El modelo predice una clase o una distribución de probabilidad sobre las clases. Con esa probabilidad, tú decides el umbral: ¿bloqueo toda transacción con probabilidad de fraude > 0.3, o solo las que superan 0.9?

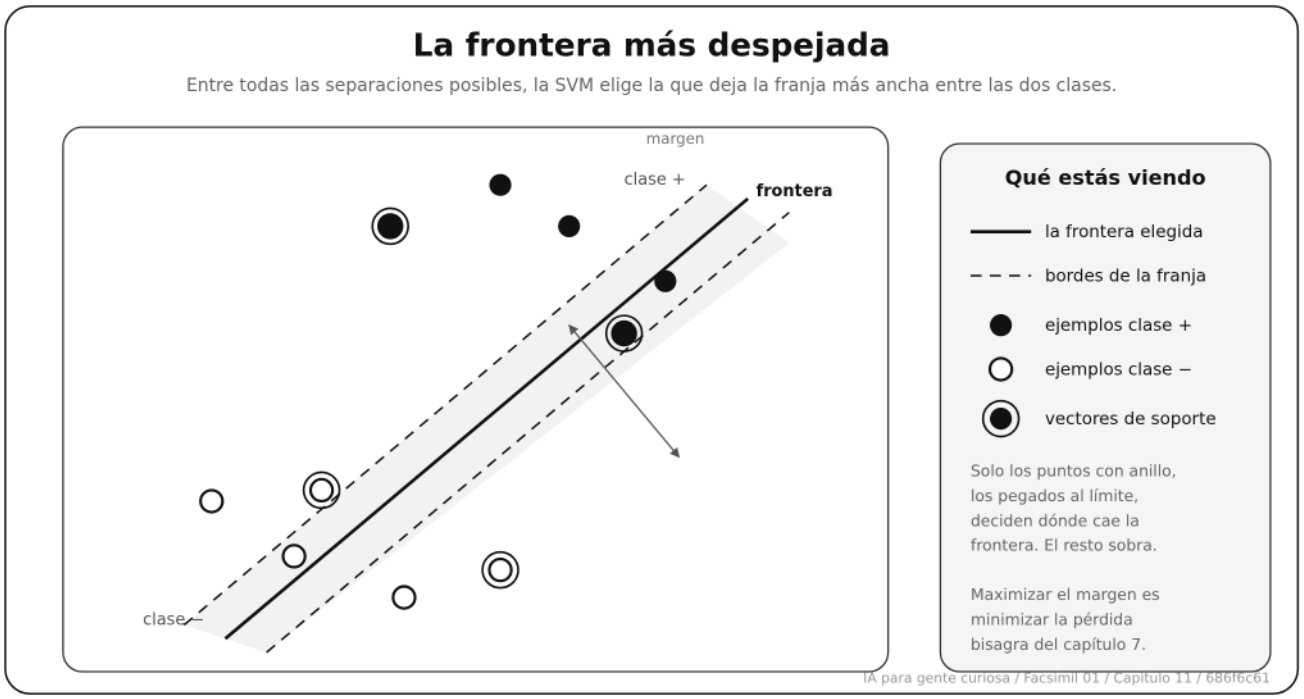
Regresión. La salida es un valor continuo. El precio de una casa, la temperatura de mañana, los ingresos del próximo trimestre. Aquí no existe «acertar o fallar» de forma binaria: importa cuánto te alejas del valor real. Un error de 1 000 € en una predicción de 300 000 € es aceptable; un error de 100 000 € no lo es.

Error común: convertir todo en clasificación porque es más cómodo. Si el valor numérico importa para decidir, usa regresión. Si solo importa el orden relativo, quizá necesitas *ranking*. Si una probabilidad dispara una acción, necesitas calibración, no solo clasificación.

Un modelo con criterio: las máquinas de vectores de soporte

Entre los modelos supervisados clásicos hay uno que merece una parada propia, porque encarna una idea elegante que sigue viva mucho después de su época dorada: la **máquina de vectores de soporte** (en inglés *support vector machine*, SVM).[4] Para una clasificación binaria con dos clases bien separadas, hay infinitas rectas (o planos) que las dividen sin error. La pregunta que se hace la SVM es cuál de todas elegir, y su respuesta tiene sentido: la que deja **la franja más ancha posible** entre las dos clases, es decir, la frontera de **margen máximo**.

La intuición es la de un camino. Si tienes que trazar una raya entre dos grupos de casas, la más robusta no es la que pasa rozando una de ellas, sino la que deja el mayor espacio libre a ambos lados; así, un dato nuevo que caiga un poco descolocado todavía quedará del lado correcto. De ese planteamiento sale el nombre del modelo: la frontera no depende de todos los ejemplos, sino solo de los pocos que tocan los bordes de la franja, los más difíciles, los pegados al límite. Esos ejemplos son los **vectores de soporte**, y son los únicos que sostienen la frontera; podrías borrar todos los demás y la solución no se movería.



Entrenar una SVM es, por debajo, exactamente lo que viste en el capítulo de funciones de pérdida: minimizar la **pérdida bisagra** (*hinge*), la que no se conforma con caer del lado correcto y exige hacerlo con holgura. Esa conexión cierra el círculo entre el modelo y su pérdida: el deseo geométrico de «la franja más ancha» y la fórmula $\max(0,\, 1 - m)$ son la misma idea contada de dos maneras.

Dos piezas más completan el modelo y explican por qué fue tan dominante en los años dos mil. La primera es el **margen blando** (*soft margin*): los datos reales no se separan limpiamente, sus clases se solapan, así que la SVM permite que algunos ejemplos invadan la franja o caigan del lado equivocado, a cambio de un coste. Un parámetro, llamado habitualmente C, regula ese equilibrio entre dejar la franja ancha y tolerar pocos errores; es la palanca de regularización del modelo. La segunda es el **truco del núcleo** (*kernel trick*): cuando ninguna recta separa bien las clases, la SVM puede medir la similitud entre ejemplos como si los hubiera proyectado a un espacio de muchas más dimensiones, donde sí existe una frontera plana, sin llegar a construir ese espacio de forma explícita. Con un núcleo adecuado, como el RBF, una frontera recta en ese espacio invisible se ve como una curva flexible en el original. Eso le dio a las SVM una potencia notable con pocos datos, antes de que el *deep learning* tomara el relevo.

¿Cuándo elegirla hoy? Las SVM rinden bien con conjuntos de datos pequeños o medianos y con muchas *features*, como la clasificación de texto, y cuando quieres una frontera robusta y bien fundamentada. En datos tabulares grandes, los *ensembles* de árboles (*random forest*, *gradient boosting*) suelen ganarles en la práctica, y cuando hay muchísimos datos y señal compleja, las redes neuronales se imponen. Pero la idea del margen máximo no se ha jubilado: sigue latiendo en la pérdida bisagra, en el modo en que pensamos la robustez de una frontera, y en la intuición de que los ejemplos difíciles, los del borde, son los que de verdad definen un clasificador.

Validación: cómo saber si generaliza

Entrenar un modelo es fácil. Lo difícil es saber si funcionará con datos que no ha visto nunca.[5]

Train / validation / test. Divide tus datos en tres conjuntos antes de tocar nada. Entrenas con *train*, ajustas hiperparámetros con *validation* y mides el rendimiento final con *test*. Si usas *test* para tomar decisiones, ya no es *test*: es *validation* y necesitas uno nuevo.

Cross-validation. Cuando tienes pocos datos, el *k-fold* entrena el modelo K veces, cada una con una partición distinta de train/validation. Te da una estimación más robusta del rendimiento y su variabilidad. No te fíes de una sola métrica con suerte.

Data leakage. El error más insidioso. Una columna que contiene información del futuro, un identificador que correlaciona con la etiqueta, un duplicado entre train y test. El modelo aprende una señal que no existirá en producción. Y tu métrica de validación será excelente... hasta que despliegues.

Data drift. El mundo cambia. Tus clientes cambian, los precios cambian, las campañas de *marketing* cambian. Un modelo entrenado con datos de 2023 puede fallar en 2026 aunque el código no haya cambiado. Monitoriza.

Overfitting y underfitting

Overfitting. El modelo memoriza los datos de entrenamiento. La pérdida de entrenamiento es mínima, pero la de validación es alta. Es el estudiante que se aprende las respuestas del examen de memoria: saca un 10 en el simulacro y suspende el examen real.[6] Solución: más datos, regularización, *dropout*, modelos más simples.

Underfitting. El modelo es demasiado simple para capturar los patrones de los datos. Tanto la pérdida de entrenamiento como la de validación son altas. Es el estudiante que ni siquiera ha abierto el libro. Solución: modelo más complejo, más *features*, más tiempo de entrenamiento.

El diagnóstico rápido: si ambas pérdidas son altas → *underfitting*. Si la de entrenamiento es baja pero la de validación es alta → *overfitting*. Si ambas son bajas y cercanas → buen trabajo.

Matriz de confusión, precision y recall

Cuando clasificas, la exactitud (*accuracy*) no cuenta toda la historia. Imagina un detector de fraude: el 99 % de las transacciones son legítimas. Un modelo que diga «no fraude» siempre tendrá un 99 % de *accuracy*. Y será completamente inútil.

La **matriz de confusión** desglosa lo que realmente ocurre:

	PREDICHO: POSITIVO	**PREDICHO: NEGATIVO**
Real: positivo	TP (verdadero positivo)	FN (falso negativo)
Real: negativo	FP (falso positivo)	TN (verdadero negativo)

De aquí salen las tres métricas que de verdad importan. La **precision** mide, de todo lo que el modelo marcó como positivo, qué parte lo era en realidad. Si predice fraude, responde cuántas de esas alertas eran fraudes reales:

$$\text{Precision} = \frac{TP}{TP + FP}$$

El **recall** (sensibilidad) mide, de todo lo que era realmente positivo, qué parte detectó el modelo. De todos los fraudes que ocurrieron, cuántos llegaste a capturar:

$$\text{Recall} = \frac{TP}{TP + FN}$$

El **F1** resume ambas en una sola cifra con su media armónica, útil cuando necesitas un equilibrio entre las dos:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
TP	Verdaderos positivos: casos positivos que el modelo acertó	fraudes detectados
FP	Falsos positivos: negativos que el modelo marcó como positivos	compras legítimas bloqueadas
FN	Falsos negativos: positivos que el modelo dejó escapar	fraudes no detectados
$\mathrm{Precision}$	Fiabilidad de las alertas positivas, entre 0 y 1	0,90 significa que nueve de cada diez alertas son fraude real
Recall	Cobertura de los positivos reales, entre 0 y 1	0,70 significa que capturas siete de cada diez fraudes
F_1	Media armónica de precision y recall, entre 0 y 1	equilibra ambas en un único número

En palabras: la precision responde «cuando aviso, ¿acierto?» y el recall responde «de todo lo que debía cazar, ¿cuánto cacé?». Subir una suele bajar la otra, porque bajar el umbral captura más positivos reales (más recall) a costa de más falsas alarmas (menos precision). El F1 resume ese equilibrio, y como la media armónica tira hacia el valor más bajo, basta con que precision o recall sea pequeño para que el F1 se hunda. Eso evita quedarte contento con un modelo que brilla en una métrica y fracasa en la otra.

La elección entre *precision* y *recall* depende del coste del error. En detección de fraude, un FN (dejar pasar un fraude) puede costar miles de euros; un FP (bloquear una compra legítima) puede costar un cliente. En diagnóstico médico, un FN (no detectar una enfermedad) puede costar una vida. No optimices *accuracy* a ciegas: optimiza la métrica que refleja el coste real del error en tu dominio.

Baseline, umbral y calibración

Un modelo no se evalúa en el vacío. Se evalúa contra una **baseline**: una regla simple que representa “lo mínimo que habría que superar”. Puede ser predecir siempre la clase mayoritaria, una regla de negocio escrita a mano o el sistema actual. Si tu modelo complejo no mejora claramente esa baseline en la métrica que importa, no has ganado ingeniería; has añadido complejidad.

Después viene el **umbral**. Muchos clasificadores no devuelven directamente una clase, sino una puntuación o probabilidad: “este ticket tiene 0,73 de probabilidad de ser urgente”. Convertir 0,73 en una acción exige una política: con umbral 0,5 quizá escalas demasiados tickets; con 0,9 quizá dejas pasar incidencias graves. El umbral no se elige por estética, sino por coste de FP y FN.

Y si vas a usar la probabilidad como probabilidad, necesitas **calibración**. Un modelo está bien calibrado si, entre todos los casos donde dice 0,8, aproximadamente el 80 % acaba siendo positivo.[7] Esto importa cuando una puntuación dispara una acción real: revisión manual, bloqueo, reembolso, alerta médica, priorización de SLA. Un modelo puede ordenar bien los casos y aun así mentir en la escala de sus probabilidades.

Clustering: agrupar sin etiquetas

A veces no tienes etiquetas, pero sospechas que tus datos tienen estructura. El *clustering* agrupa instancias por similitud. El algoritmo más sencillo y usado es **k-means**:[8]

1. Elige K (el número de grupos que quieres encontrar).
2. Coloca K centroides aleatoriamente.
3. Asigna cada punto al centroide más cercano.
4. Recalcula los centroides como la media de los puntos asignados.
5. Repite hasta que los centroides se estabilizan.

Es simple, rápido y funciona sorprendentemente bien. Pero tiene trampas:

- **Tienes que elegir K.** Si no sabes cuántos grupos hay, toca probar. Los métodos del codo (*elbow*) y la silueta (*silhouette*) ayudan, pero no sustituyen el criterio.
- **La distancia importa.** Normalmente se usa distancia euclídea. Si una *feature* tiene valores en millones y otra entre 0 y 1, la primera domina completamente la agrupación. Normaliza.
- **La semilla importa.** Distintas inicializaciones dan distintos resultados. No te cases con la primera agrupación que veas.

Cuándo no confiar en clusters. Que el algoritmo encuentre grupos no significa que esos grupos signifiquen algo. Un *cluster* puede ser un artefacto de la métrica de distancia, de una *feature* mal escalada o del azar. Valida siempre contra conocimiento del dominio: ¿tienen sentido esos grupos para quien conoce el negocio? Si no puedes explicar qué tienen en común los miembros de un *cluster*, probablemente no sea un hallazgo, sino ruido.

Más allá de k-means: jerárquico y por densidad

k-means es el martillo, pero no todo es un clavo. Arrastra dos supuestos fuertes: que sabes cuántos grupos hay (la K) y que esos grupos son más o menos esféricos y de tamaño parecido. Cuando alguna de esas dos cosas no se cumple, conviene tener a mano otras dos familias clásicas.

Clustering jerárquico (aglomerativo). En vez de fijar K de antemano, empieza tratando cada punto como su propio grupo y, paso a paso, funde los dos grupos más cercanos, hasta que queda uno solo. El resultado no es una partición, sino un árbol de fusiones —el **dendrograma**— que puedes cortar a la altura que quieras para obtener 2, 5 o 20 grupos. La decisión central es cómo medir la distancia entre dos grupos, lo que se llama el *enlace*: el mínimo entre sus puntos (*single*, que tiende a encadenar), el máximo (*complete*), la media, o el criterio de Ward, que funde los grupos que menos aumentan la varianza interna.[9] Su ventaja es doble: no te obliga a elegir K y te da una jerarquía legible. Su coste es que escala mal (cuadrático o peor en memoria y tiempo), así que no es para millones de puntos.

DBSCAN (por densidad). Define un *cluster* como una zona densa de puntos separada de otras por zonas vacías. Solo necesita dos parámetros: un radio `eps` y un mínimo de vecinos `minPts` para considerar un punto «interior». A partir de ahí, expande cada región densa hasta donde llega la densidad.[10] Tiene tres virtudes que a k-means le faltan: **descubre K por sí solo**, encuentra **grupos de forma arbitraria** (no solo esferas, también lunas o anillos) y **marca como ruido** los puntos aislados, en vez de forzarlos dentro de un grupo. Su talón de Aquiles es elegir `eps`: si los grupos tienen densidades muy distintas, un único radio no sirve para todos.

Evaluar agrupaciones sin etiquetas: la silueta

Sin etiquetas no existe el «acierto», así que ¿cómo sabes si una agrupación es buena? Una medida muy usada es el **coeficiente de silueta**. Para cada punto compara lo cerca que está de los suyos con lo cerca que está del grupo vecino más próximo. Si a es la distancia media a su propio grupo y b la distancia media al grupo ajeno más cercano, la silueta del punto es:

$$s = \frac{b - a}{\max(a, b)} \in [-1, 1]$$

Cerca de 1 significa que el punto está bien dentro de su grupo y lejos de los demás; cerca de 0, que cae justo en la frontera; negativa, que probablemente está en el grupo equivocado.[11] Promediar la silueta de todos los puntos da una nota global que, entre otras cosas, sirve para tantear cuántos grupos tiene tu *dataset*: pruebas varios valores de K y te quedas con el que mejor silueta da. Eso sí, sigue siendo una pista, no una verdad: una silueta alta sobre una métrica mal escalada también engaña.

Reducir dimensiones: PCA y la maldición de la dimensionalidad

Cuando cada instancia tiene cientos o miles de *features*, aparecen problemas que no se ven con dos o tres columnas: las distancias se vuelven poco informativas (casi todo queda «lejos» de casi todo), los modelos sobreajustan con facilidad y, además, no hay forma de dibujar los datos para mirarlos. Es la **maldición de la dimensionalidad**. La respuesta clásica es **reducir la dimensión**: describir cada punto con muchas menos coordenadas perdiendo lo mínimo.

El método de referencia es el **análisis de componentes principales (PCA)**. La idea geométrica es buscar las direcciones en las que los datos más varían y proyectar sobre ellas. Esas direcciones —las **componentes principales**— son los vectores propios de la matriz de covarianza de los datos, ordenados por su valor propio, que mide cuánta varianza captura cada uno.[12] [13] Si X son los datos centrados y $\Sigma = \frac{1}{n} X^\top X$ su covarianza, PCA resuelve $\Sigma v = \lambda v$: cada vector propio v es una componente y su λ dice qué fracción de la varianza explica. Quedándote con las primeras componentes —las de mayor λ— conservas casi toda la información con muchas menos columnas. El **gráfico de sedimentación** (*scree plot*), que dibuja la varianza explicada por componente, ayuda a decidir cuántas conservar.

PCA es una proyección **lineal**, así que cuando la estructura interesante es curva conviene otra herramienta solo para *visualizar*: **t-SNE** y **UMAP** colocan los puntos en dos dimensiones intentando preservar la vecindad local, y producen esos mapas donde los grupos saltan a la vista.[14] Una advertencia importante: estos mapas deforman las distancias globales, así que sirven para *ver* estructura, no para medirla ni para alimentar otro modelo con sus coordenadas. Esta reducción de dimensión es, además, el puente con el resto del facsímil: los *embeddings* del facsímil 4 y las representaciones del facsímil 8 son exactamente esto — describir cada cosa con un vector corto que conserva lo que importa.

Cómo funciona por dentro

Hasta aquí hemos hablado de clasificación, métricas y validación desde fuera. Conviene abrir uno de estos algoritmos y mirar su engranaje. Elijo el árbol de decisión porque conecta de forma directa con la clasificación y porque es el ejemplo canónico de modelo interpretable. La idea de partida es sencilla. Un árbol de decisión es una secuencia de preguntas sobre las *features* de cada instancia. Cada pregunta divide el grupo de ejemplos en dos partes y la meta es que esas partes queden lo más puras posible, es decir, que dentro de cada parte casi todos los ejemplos pertenezcan a la misma clase. Predecir consiste después en recorrer el árbol desde la raíz, respondiendo cada pregunta, hasta llegar a una hoja que asigna una clase.

El corazón del algoritmo es cómo elige cada pregunta. En cada nodo el árbol evalúa muchos cortes posibles del tipo «edad menor que 30», «importe mayor que 500», «tipo de cliente igual a premium» y se queda con el corte que más reduce la impureza de los grupos resultantes. La impureza se mide con un criterio numérico. El más usado es el índice de Gini, que cuantifica la probabilidad de etiquetar mal un ejemplo elegido al azar si le asignamos una clase según la proporción del grupo. Su fórmula es:

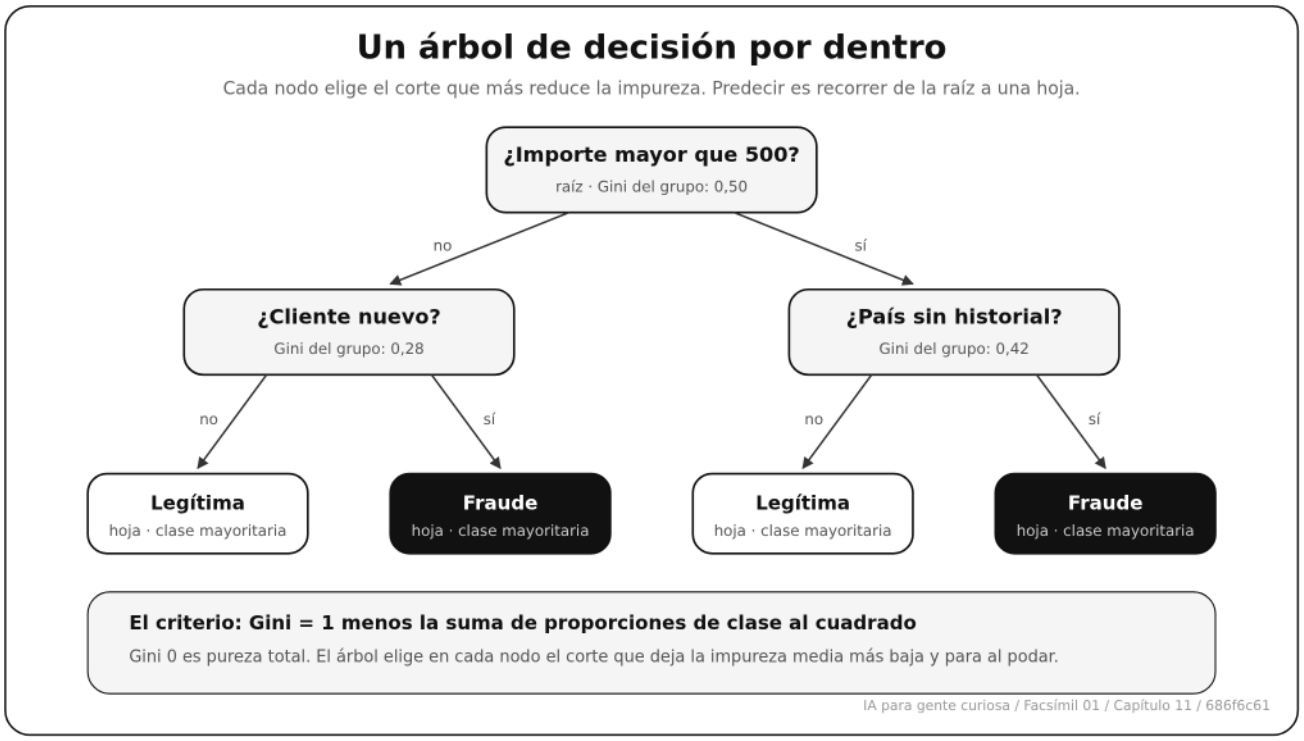
$$G = 1 - \sum_{k=1}^{C} p_k^2$$

SÍMBOLO	SIGNIFICADO
G	Índice de Gini del grupo, entre 0 y un valor máximo según el número de clases
C	Número de clases posibles
p_k	Proporción de ejemplos del grupo que pertenecen a la clase k
$\sum$	Suma sobre todas las clases, de la primera a la última

En palabras: si todos los ejemplos del grupo son de la misma clase, una proporción vale 1 y el resto 0, la suma de cuadrados vale 1 y el Gini queda en 0, que es la pureza total. Si las clases están muy mezcladas, las proporciones se reparten, la suma de cuadrados baja y el Gini sube. El árbol prueba cada corte candidato, calcula el Gini de los dos grupos que genera, los pondera por su tamaño y elige el corte que deja la impureza media más baja. Una alternativa equivalente en espíritu es la entropía, que mide la misma idea de mezcla con logaritmos. Ambas premian los cortes que separan limpiamente las clases.

El proceso es recursivo. Una vez elegido el mejor corte de la raíz, el árbol repite exactamente el mismo razonamiento dentro de cada rama, buscando ahora el mejor corte para el subgrupo que ha quedado en esa rama. Y vuelve a repetirlo en los subgrupos de los subgrupos. Este crecimiento continúa hasta una condición de parada: que el grupo sea puro, que queden muy pocos ejemplos para seguir dividiendo, que se alcance una profundidad máxima fijada de antemano o que ningún corte mejore la impureza. Aquí reaparece una idea de este capítulo. Si dejas crecer el árbol sin freno, acabará memorizando el conjunto de entrenamiento con ramas hechas a medida de cada ejemplo, que es justo el *overfitting* descrito antes. Por eso se podan los árboles o se limita su profundidad, lo que equivale a buscar el modelo más simple que aún explica los datos.

La razón por la que este algoritmo es interpretable está en su propia mecánica. La predicción de un árbol es un camino de preguntas legibles que cualquier persona del negocio puede seguir: «se marcó como fraude porque el importe superaba 500, el cliente era nuevo y la operación venía de un país sin historial». No hay nada oculto. Puedes imprimir el árbol entero y auditar cada decisión. Una red neuronal profunda, en cambio, distribuye su criterio entre millones de pesos sin un significado individual claro, de modo que sabes qué predice pero no puedes explicar fácilmente por qué. Esta transparencia es la que conecta con el espíritu del *machine learning* clásico que recorre todo el capítulo: modelos que no solo aciertan, sino que permiten justificar la decisión ante quien la sufre o se beneficia de ella. Por eso, cuando necesitas defender una clasificación delante de un cliente, un regulador o un equipo de negocio, un árbol de decisión sigue siendo una de las herramientas más valiosas del catálogo.



En el día a día

Cuando te enfrentes a un problema real, la primera pregunta es si tienes etiquetas. Si las tienes, es un problema supervisado: clasificación cuando la salida es una categoría y regresión cuando es un número. En ambos casos conviene empezar por lo simple, una regresión logística o un árbol de decisión, antes de saltar a una red neuronal. Si el modelo sencillo ya resuelve el problema, te has ahorrado complejidad, coste y un sistema difícil de explicar.

Si no tienes etiquetas, el terreno cambia. El clustering sirve para explorar y ver si los datos esconden grupos, la detección de anomalías para encontrar lo raro y la reducción de dimensionalidad para visualizar y comprimir. Son herramientas de descubrimiento, no de predicción: te muestran estructura, pero la interpretación corre de tu cuenta.

El volumen de datos también manda. Con cientos o miles de ejemplos, el ML clásico es tu mejor apuesta, porque los árboles, las SVMs y los *ensembles* rinden bien en ese rango. El *deep learning* solo empieza a brillar cuando hay decenas de miles de ejemplos o más. Y si necesitas justificar tus decisiones ante un cliente, un regulador o un compañero, un árbol de decisión te dice exactamente por qué clasificó algo como lo hizo, mientras que una red de cien capas no te da esa cuenta.

Por qué debería importarte

El *deep learning* no ha dejado obsoleto al ML clásico. Lo ha complementado. Para muchos problemas, en especial con datos tabulares y pocos ejemplos, un *random forest* o una regresión logística siguen siendo más rápidos, más baratos y más interpretables que cualquier red neuronal.

Además, el ML clásico te da el vocabulario para medir. *Precision*, *recall*, *overfitting*, *cross-validation*: estos conceptos no desaparecen cuando usas LLMs. Se vuelven más importantes. Un agente de IA que clasifica tickets de soporte necesita una matriz de confusión igual que un clasificador de *spam* de 2005.

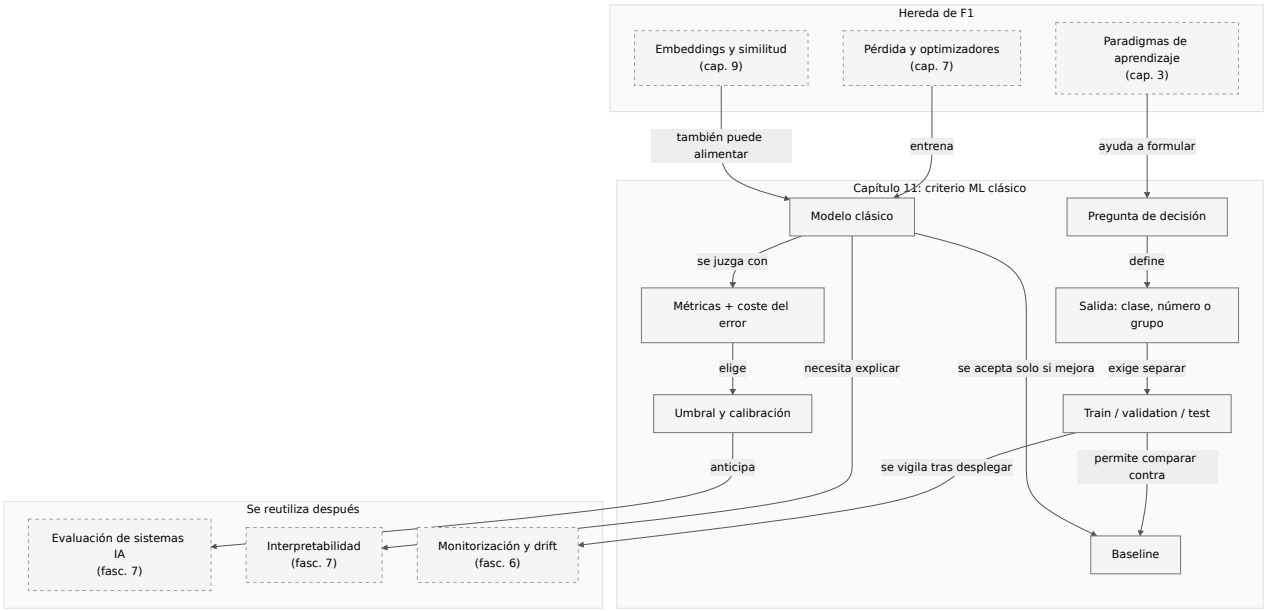
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Medir solo *accuracy*	En conjuntos desbalanceados (1 % de fraude, 99 % legítimo), un modelo que siempre dice «legítimo» tiene 99 % de *accuracy* y es completamente inútil.	Mira *precision*, *recall* y F1. Si las clases están desbalanceadas, *accuracy* no te dice nada.
No separar train/validation /test	Si usas los mismos datos para entrenar y evaluar, el modelo parece perfecto... hasta que lo pones en producción.	Separa antes de tocar nada. *Test* se toca una sola vez, al final.
Ignorar la escala de las features	En k-means (y en cualquier algoritmo basado en distancia), una *feature* con valores grandes domina a las demás.	Normaliza o estandariza tus *features* antes de entrenar.
Interpretar clusters como realidades objetivas	El algoritmo siempre encuentra grupos, aunque los datos sean ruido puro.	Valida contra conocimiento del dominio. Si los grupos no tienen sentido para el negocio, no los uses para decidir.

Cómo encaja todo

Este mapa se lee desde la decisión, no desde el algoritmo. Venimos de los principios del capítulo 3 y de las pérdidas del capítulo 7, pero aquí aparece la disciplina que seguirá en todo el facsímil: definir salida, baseline, validación y métrica antes de enamorarse de una arquitectura.

Lo que se aprende en este capítulo vuelve después en evaluación de LLMs, calibración, interpretabilidad y operación. Un sistema con agentes o RAG también necesita train/test mental: qué casos pruebo, qué métrica acepto, qué coste tiene cada error y cuándo prefiero una regla simple.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Clasificación	Tarea supervisada donde la salida es una categoría discreta entre un conjunto predefinido de clases.
Regresión	Tarea supervisada donde la salida es un valor numérico continuo.
Overfitting	El modelo memoriza los datos de entrenamiento y no generaliza a datos nuevos.
Underfitting	El modelo es demasiado simple para capturar los patrones de los datos.
Matriz de confusión	Tabla que cruza predicciones con valores reales, desglosando aciertos, falsos positivos y falsos negativos.
Precision	Proporción de predicciones positivas que son realmente correctas.
Recall	Proporción de casos positivos reales que el modelo detectó.
F1	Media armónica de precision y recall; solo es alta cuando ambas lo son.
Árbol de decisión	Modelo que clasifica mediante una secuencia de preguntas sobre las *features*, legible de la raíz a la hoja.
Índice de Gini	Medida de impureza de un grupo; vale 0 cuando todos los ejemplos son de la misma clase.
Máquina de vectores de soporte (SVM)	Clasificador que separa dos clases con la frontera de margen máximo. Se entrena minimizando la pérdida bisagra.
Margen máximo	La franja más ancha posible entre las clases; la SVM elige la frontera que la maximiza.
Vectores de soporte	Los ejemplos pegados al borde de la franja; los únicos que determinan dónde cae la frontera.
Kernel trick	Truco que permite trazar fronteras no lineales midiendo similitudes como en un espacio de más dimensiones, sin construirlo.
Baseline	Regla o sistema simple que el modelo debe superar para justificar su complejidad.
Calibración	Grado en que las probabilidades predichas se corresponden con frecuencias reales observadas.

TÉRMINO	DEFINICIÓN
Clustering	Técnica no supervisada que agrupa instancias por similitud sin etiquetas predefinidas.

Antes de pasar página

- [ ] ¿Puedo distinguir entre clasificación y regresión con ejemplos concretos? (Si no, vuelve a «Clasificación vs regresión».)
- [ ] ¿Sé interpretar una matriz de confusión? (Si no, vuelve a «Matriz de confusión».)
- [ ] ¿Entiendo la diferencia entre *precision* y *recall*? (Si no, vuelve a esa sección.)
- [ ] ¿Puedo explicar qué baseline usaría y qué umbral elegiría si el coste de FP y FN cambia? (Si no, vuelve a «Baseline, umbral y calibración».)
- [ ] ¿Puedo explicar qué es el *overfitting* y cómo detectarlo? (Si no, vuelve a «Overfitting y underfitting».)
- [ ] ¿Entiendo qué es el margen máximo de una SVM y por qué solo importan los vectores de soporte? (Si no, vuelve a «Un modelo con criterio».)
- [ ] ¿Sé interpretar precision, recall, F1 y falsos negativos? (Si no, vuelve a «Matriz de confusión, precision y recall».)

En resumen

IDEA FUERZA	DETALLE
El ML clásico no ha muerto.	Para datos tabulares, pocos ejemplos o necesidad de explicabilidad, los árboles y las regresiones siguen siendo la mejor opción.
La pregunta no es «qué modelo», sino «qué salida, qué señal, cómo mido».	Define el problema antes de elegir la herramienta. Clasificación, regresión, clustering: cada uno tiene sus métricas.
Las métricas correctas dependen del coste del error.	No optimices *accuracy* a ciegas. Un FN en diagnóstico médico no vale lo mismo que un FP en recomendaciones de películas.
Una probabilidad no es una decisión hasta que eliges baseline, umbral y calibración.	Si el modelo no supera una regla simple o sus probabilidades no son fiables, todavía no tienes un sistema defendible.
El *clustering* encuentra estructura, no significado.	Valida siempre contra conocimiento del dominio. Un grupo no es un hallazgo hasta que alguien que conoce el negocio dice «esto tiene sentido».

Para saber más

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

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5-32. https://doi.org/10.1023/A:1010933404324

Cortes, C. y Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273-297. https://doi.org/10.1007/BF00994018

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

Guo, C., Pleiss, G., Sun, Y. y Weinberger, K. Q. (2017). On calibration of modern neural networks. En *Proceedings of the 34th International Conference on Machine Learning* (pp. 1321-1330). PMLR. https://proceedings.mlr.press/v70/guo17a.html

Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The elements of statistical learning* (2.ª ed.). Springer. https://web.stanford.edu/~hastie/ElemStatLearn/

LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. En *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability* (Vol. 1, pp. 281-297). https://projecteuclid.org/euclid.bsmsp/1200512992

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

Notas

1. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. Los capítulos 18 a 21 cubren los paradigmas de aprendizaje y establecen el marco conceptual para clasificar cualquier problema de ML según el tipo de datos y retroalimentación disponible. ↩

2. Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer. El capítulo 1 establece los fundamentos de la teoría de la probabilidad y la decisión, y define la terminología básica de *datasets*, *features* y variables objetivo. ↩

3. Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The elements of statistical learning* (2.ª ed.). Springer. https://web.stanford.edu/~hastie/ElemStatLearn/. El capítulo 2 ofrece una visión general de aprendizaje supervisado y establece la distinción fundamental entre problemas de clasificación y regresión. ↩

4. Cortes, C. y Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273-297. https://doi.org/10.1007/BF00994018. El artículo introduce las SVM, que buscan la frontera de separación de máximo margen y popularizaron el uso de *kernels* para fronteras no lineales. ↩

5. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org. El capítulo 5.2 aborda las técnicas de validación y la importancia de separar correctamente los conjuntos de entrenamiento, validación y prueba para obtener estimaciones fiables del rendimiento. ↩

6. Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5-32. https://doi.org/10.1023/A:1010933404324. Breiman demostró que los *ensembles* de árboles de decisión, como los *random forests*, mitigan el *overfitting* al promediar múltiples modelos entrenados con subconjuntos distintos de datos. ↩

7. Guo, C., Pleiss, G., Sun, Y. y Weinberger, K. Q. (2017). On calibration of modern neural networks. En *Proceedings of the 34th International Conference on Machine Learning* (pp. 1321-1330). PMLR. https://proceedings.mlr.press/v70/guo17a.html. El artículo muestra que modelos modernos pueden tener alta exactitud y estar mal calibrados, por lo que sus probabilidades no deben interpretarse sin comprobación. ↩

8. MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. En *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability* (Vol. 1, pp. 281-297). https://projecteuclid.org/euclid.bsmsp/1200512992.

MacQueen introdujo el algoritmo k-means, que sigue siendo uno de los métodos de *clustering* más utilizados por su simplicidad y eficiencia computacional. ↩

9. Ward, J. H. (1963). Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301), 236-244. https://doi.org/10.1080/01621459.1963.10500845. Ward propuso el criterio de enlace que minimiza el incremento de varianza intragrupo al fundir, uno de los más usados en *clustering* aglomerativo. ↩

10. Ester, M., Kriegel, H.-P., Sander, J. y Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. En *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)* (pp. 226-231). AAAI Press. DBSCAN encuentra el número de grupos por sí solo, descubre formas arbitrarias y marca como ruido los puntos que no encajan en ninguna región densa. ↩

11. Rousseeuw, P. J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20, 53-65. https://doi.org/10.1016/0377-0427(87)90125-7. Rousseeuw introdujo la silueta como ayuda gráfica para interpretar y validar agrupaciones. ↩

12. Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(11), 559-572. https://doi.org/10.1080/14786440109462720. Pearson planteó por primera vez el ajuste de líneas y planos que mejor representan una nube de puntos, semilla de PCA. ↩

13. Jolliffe, I. T. (2002). *Principal Component Analysis* (2.ª ed.). Springer. El texto de referencia moderno sobre PCA, su álgebra, sus variantes y sus límites. ↩

14. van der Maaten, L. y Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9, 2579-2605. t-SNE preserva las distancias locales para revelar estructura de grupos en alta dimensión, pero deforma las distancias globales: úsalo para mirar, no para medir. ↩

Cuadernos para practicar

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

Machine learning clásico que decide de verdad

https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%C3%ADmil%2001/04-ml-clasico-que-decide.ipynb

Capítulo 12: Lo que deberías saber: recapitulación de fundamentos

Entrando en el tema

Has recorrido once capítulos convertidos en una lectura lenta. Has construido una neurona, entrenado una red, propagado errores hacia atrás, tokenizado texto, explorado embeddings y contrastado paradigmas de aprendizaje.

Este capítulo no es un resumen. Es una **revisión activa**: cada sección recupera un concepto nuclear, lo reformula desde otro ángulo, lo conecta con el resto y te confronta con una pregunta. Si algo no te suena, el número de capítulo te dice exactamente dónde volver.

No es un examen. Es un espejo. Si te reconoces en estas páginas, puedes avanzar al facsímil 2.

---

1. Qué es y qué no es la IA

El concepto. La inteligencia artificial no es consciente, no «piensa» y no es infalible. Es un sistema de predicción estadística de tokens que opera sobre patrones aprendidos de cantidades masivas de datos.[1] La confusión entre elocuencia y corrección es el error más caro que puedes cometer al diseñar sistemas con IA.[2]

Para recordar. Cada vez que un LLM te responde con frases impecables, recuerda: está generando tokens a partir de patrones aprendidos, no deliberando como una persona que contrasta el mundo. La confianza con la que afirma algo no guarda relación directa con la veracidad de lo que afirma.

Ejemplo fresco. Imagina que le pides a un LLM que calcule `1573 × 2849`. Si le obligas a descomponer el cálculo paso a paso, aumentas las probabilidades de que cada parte sea coherente y verificable. Si le pides la respuesta directa, puede inventarse un número con total seguridad. No es que «sepa multiplicar» o «no sepa»: es que el contexto y el procedimiento que le das cambian la distribución de tokens que generará.

Vuelve al capítulo 1 si: no puedes explicar por qué la IA no es un oráculo ni un buscador.

---

2. Sistemas deterministas frente a sistemas probabilísticos

El concepto. `sumar(2, 3)` siempre devuelve 5. `preguntar("¿Qué es Rust?")` puede devolver tres respuestas distintas, todas correctas. En IA generativa, la variabilidad suele aparecer en el muestreo, en la configuración de inferencia y en el sistema que rodea al modelo.[3] Esto cambia cómo diseñas, cómo testeas y cómo despliegas.

Para recordar. No puedes hacer `assert respuesta == "Rust es un lenguaje..."`. Validarás propiedades: ¿contiene las palabras clave?, ¿respeta el formato?, ¿menciona los conceptos correctos? El cambio de mentalidad, de igualdad exacta a validación de propiedades, es el paso más importante al trabajar con IA.

Ejemplo fresco. Un sistema de atención al cliente usa un LLM para clasificar la urgencia de tickets. Dos ejecuciones con el mismo ticket pueden dar «urgente» y «normal». Si tu sistema despacha automáticamente basándose en esa clasificación, necesitas un umbral de confianza y, probablemente, revisión humana para los casos límite. La variabilidad no es un *bug*: es una característica del sistema que debes diseñar.

Vuelve al capítulo 2 si: no puedes explicar por qué `temperature = 0` no garantiza determinismo absoluto.

---

3. Los cinco principios

El concepto. Cinco ideas sostienen la IA moderna: aprendizaje supervisado (ejemplos etiquetados), no supervisado (encontrar estructura sin etiquetas), post-training (enseñar preferencias), atención (mirar todo a la vez) y *scaling laws* (relaciones empíricas entre datos, parámetros, cómputo y pérdida).[4]

Para recordar. No son compartimentos estancos. Un LLM se pre-entrena con aprendizaje auto-supervisado, se post-entrena con SFT (supervisado) y RLHF (refuerzo), y su arquitectura se basa en la atención. Las *scaling laws* ayudan a razonar sobre tendencias de escala bajo condiciones concretas; no sustituyen la evaluación de tu tarea, tu coste y tus datos.

Ejemplo fresco. Cuando OpenAI entrena un nuevo modelo, no es un solo paso. Primero, pre-entrenamiento masivo sobre texto de internet (auto-supervisado). Luego, SFT con conversaciones de alta calidad escritas por personas (supervisado). Finalmente, RLHF donde personas comparan respuestas (refuerzo). Tres paradigmas encadenados. La arquitectura Transformer con atención es el soporte que hace posibles los tres.

Vuelve al capítulo 3 si: no puedes nombrar los cinco principios y dar un ejemplo de cada uno.

---

4. La neurona artificial

El concepto. Una neurona artificial es una función matemática: $y = f(\sum_i w_i x_i + b)$. Entradas multiplicadas por pesos, sumadas, más un sesgo, pasadas por una función de activación.[5] Tres líneas de código. El átomo del *deep learning*.[6]

Para recordar. La inteligencia no está en una neurona. Está en los miles de millones de parámetros ajustados durante el entrenamiento a lo largo de capas y capas de neuronas interconectadas. Pero la unidad mínima es siempre la misma.

Ejemplo fresco. ¿Cuántas neuronas se activan cuando escribes «Buenos días» en ChatGPT? Cada token de entrada atraviesa todas las capas del modelo. En un modelo de 70B parámetros, una sola predicción involucra miles de millones de operaciones $y = f(\sum_i w_i x_i + b)$. Tu «Buenos días» desencadena una cascada de multiplicaciones y sumas que recorre toda la red en milisegundos.

Vuelve al capítulo 4 si: no puedes escribir la fórmula de memoria y calcular la salida para `x₁=0.5, x₂=0.3, w₁=0.2, w₂=-0.4, b=0.1` con activación ReLU.

---

5. Redes neuronales: capas y arquitectura

El concepto. Una red neuronal es un grafo de neuronas organizadas en capas.[7] Cada capa transforma los datos y los pasa a la siguiente. Las primeras capas detectan patrones simples; las profundas, conceptos abstractos.[8]

Para recordar. «*Deep*» significa «muchas capas». Más capas permiten más abstracción, pero también requieren más datos, más computación y técnicas para que el entrenamiento no se rompa: buena inicialización de los pesos, normalización (BatchNorm, LayerNorm) y conexiones residuales que dan un atajo al gradiente. La arquitectura determina lo que la red puede aprender.

Ejemplo fresco. Un MLP de 3 capas puede clasificar dígitos escritos a mano (MNIST). Un Transformer de 96 capas puede mantener una conversación de una hora. La neurona es la misma en ambos. Lo que cambia es la organización. La arquitectura es la decisión de diseño más importante después de elegir los datos.

Vuelve al capítulo 5 si: no puedes dibujar la estructura de una red con capa de entrada, dos ocultas y una de salida, y explicar qué aprende cada capa.

---

6. Retropropagación

El concepto. La retropropagación aplica la regla de la cadena para propagar el error desde la salida hacia atrás.[9] Para cada peso, calcula $\partial E/\partial w$: cuánto cambia el error si modifico este peso. La fórmula completa:

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial a} \cdot \sigma'(z) \cdot x$$

SÍMBOLO	SIGNIFICADO
$\partial E/\partial w$	Cuánto cambia el error si muevo un poco este peso
$\partial E/\partial a$	Cuánto cambia el error con la salida de la neurona
$\sigma'(z)$	Pendiente de la activación en el punto z
x	Entrada que llegó por esa conexión

En palabras: el error se propaga hacia atrás multiplicando tres factores encadenados, que es la regla de la cadena, y el producto indica en qué dirección y con qué fuerza corregir cada peso.

Para recordar. El gradiente dice «dirección y sensibilidad». El *learning rate* dice «tamaño del paso». Actualizamos con signo menos porque queremos **reducir** el error, no aumentarlo. La regla $w \leftarrow w - \eta \cdot \partial E/\partial w$ es la ecuación más importante del *deep learning*.

Ejemplo fresco. En el capítulo 6 calculaste que con `x=2, w=0.40, b=-0.10, y=1, η=0.1`, el gradiente `∂E/∂w = -0.148`. Tras la actualización, `w = 0.415`. La nueva predicción se acerca más a 1. Una iteración. Millones por delante. Esto, multiplicado por miles de millones de parámetros y repetido durante semanas en clústeres de GPUs, es como se entrena un LLM.

Vuelve al capítulo 6 si: no puedes calcular a mano el gradiente para el ejemplo anterior.

---

7. Funciones de pérdida y optimizadores

El concepto. La función de pérdida define qué significa «equivocarse». El optimizador decide cómo corregirlo.[10] *Cross-entropy* para clasificar, MSE para regresión. SGD es simple, Adam es adaptativo (mantiene dos medias móviles por parámetro, la del gradiente y la de su cuadrado, para adaptar el paso de cada uno), y AdamW, que además desacopla el *weight decay*, es el estándar para Transformers.[11]

Para recordar. La elección de la función de pérdida es la especificación formal de tu objetivo. Si usas MSE para clasificar, el modelo optimizará la distancia numérica entre probabilidades, no la probabilidad de la clase correcta. Aprenderá algo, pero no lo que quieres.

Ejemplo fresco. Tienes que clasificar 10 000 tickets de soporte en 5 categorías. Si usas MSE, el modelo tratará la diferencia entre «categoría 1» y «categoría 2» como numéricamente equivalente a la diferencia entre «categoría 1» y «categoría 5», lo cual no tiene sentido para categorías nominales. *Cross-entropy* con softmax respeta que las categorías son discretas y no ordenadas.

Vuelve al capítulo 7 si: no puedes explicar cuándo usar *cross-entropy* y cuándo MSE.

---

8. CNNs y RNNs

El concepto. Las CNNs detectan patrones locales en imágenes mediante filtros convolucionales.[12] Las RNNs procesan secuencias manteniendo un estado oculto. Las LSTMs añaden compuertas para recordar más tiempo.[13] El Transformer las superó con paralelismo y atención directa.[14]

Para recordar. Cada limitación que el Transformer resolvió era una limitación real de las RNNs. El estado oculto no bastaba para contexto largo. El procesamiento secuencial era demasiado lento. La atención directa y el paralelismo fueron las respuestas.

Ejemplo fresco. Procesar «El gato que perseguía al ratón que robó el queso que estaba en la cocina es negro» con una RNN: cuando llega a «es negro», el estado oculto ya ha sido sobrescrito decenas de veces y probablemente ha olvidado que el sujeto era «gato». Con un Transformer, la atención conecta directamente «negro» con «gato» sin importar cuántas palabras haya en medio.

Vuelve al capítulo 8 si: no puedes comparar CNN, RNN, LSTM y Transformer en una frase cada una.

---

9. Tokens y embeddings

El concepto. Un token es la unidad mínima de texto que procesa el modelo.[15] Un embedding es un vector denso de números que representa el significado de un token en un espacio de alta dimensionalidad.[16] Los embeddings son el pegamento entre el mundo del texto y el mundo de las matemáticas.

Para recordar. El español gasta más tokens que el inglés. «Machine learning is great» son 4 tokens; «El aprendizaje automático es genial» son 7-8. Todo se mide en tokens: contexto, precio, límites. Y la similitud coseno es la medida estándar para comparar embeddings.

Ejemplo fresco. Construyes un buscador semántico para documentación interna. Conviertes 5 000 documentos a embeddings (modelo `text-embedding-3-small`, 1536 dimensiones). Cuando un empleado busca «cómo solicitar vacaciones», el sistema compara el embedding de la consulta con los 5 000 embeddings de documentos y devuelve los más cercanos por similitud coseno. Encuentra la política de vacaciones aunque la consulta no comparta ni una palabra con el título del documento.

Vuelve al capítulo 9 si: no puedes explicar la diferencia entre token y embedding, y para qué sirve cada uno.

---

10. Entrenamiento frente a inferencia

El concepto. Entrenar es cambiar pesos con datos y una función de pérdida. Pre-entrenar un modelo fundacional puede requerir semanas, clústeres enormes y presupuestos altísimos; entrenar modelos pequeños o ajustar adaptadores puede ser mucho más accesible.[17] Inferir es usar pesos ya entrenados para responder. La cuantización reduce la precisión numérica de los pesos (FP16 → INT4) para que un modelo de 14 GB quepa en 3,5 GB.

Para recordar. Casi siempre harás inferencia, evaluación, RAG, integración o ajuste parcial antes que pre-entrenamiento de frontera. El *fine-tuning* está a medio camino. Y la cuantización es una herramienta para mover modelos a hardware más modesto, siempre que evalúes la pérdida de calidad en tu tarea. En inferencia, la KV cache acelera la generación token a token reutilizando lo ya calculado, y los adaptadores como LoRA permiten especializar un modelo grande entrenando solo una fracción diminuta de sus pesos.

Ejemplo fresco. Descargas un modelo de 7B parámetros en formato GGUF (cuantizado a 4 bits). Ocupa 3,5 GB. Lo ejecutas en tu portátil con llama.cpp, sin GPU, sin conexión a internet. La calidad es ligeramente inferior al modelo original en FP16, pero para resumir documentos internos, clasificar correos o generar primeras versiones, funciona perfectamente. Has pasado de depender de una API a tener el modelo en local.

Vuelve al capítulo 10 si: no puedes explicar la diferencia entre entrenamiento, inferencia, *fine-tuning* y cuantización.

---

11. Machine learning clásico

El concepto. Antes del *deep learning*, y todavía hoy para la mayoría de problemas, existe el ML clásico.[18] Clasificación, regresión, clustering, validación, matriz de confusión, *precision*, *recall*: este vocabulario no desaparece con los LLMs, se vuelve más importante.

Para recordar. La pregunta no es «¿qué modelo uso?» sino «¿qué salida necesito, qué señal tengo y cómo sé que generaliza?». Antes de aceptar un modelo, compáralo con una baseline, fija el umbral según el coste de cada error y revisa la calibración de sus probabilidades. Si necesitas explicar cada decisión, un árbol de decisión, que parte los datos por el corte que más reduce la impureza (Gini), se lee de la raíz a la hoja. Un *random forest* con 500 árboles puede ser más rápido, más barato y más interpretable que una red neuronal para tu problema concreto.[19]

Ejemplo fresco. Tienes 2 000 transacciones etiquetadas (147 operaciones irregulares, 1 853 legítimas). Un *random forest* te da 94 % de *recall* en la clase irregular (detectas 138 de 147) con 3 % de falsos positivos. Entrenar tarda 3 segundos. No necesitas GPU. Puedes explicar exactamente qué *features* usa el modelo para decidir. ¿Necesitas un LLM para esto?

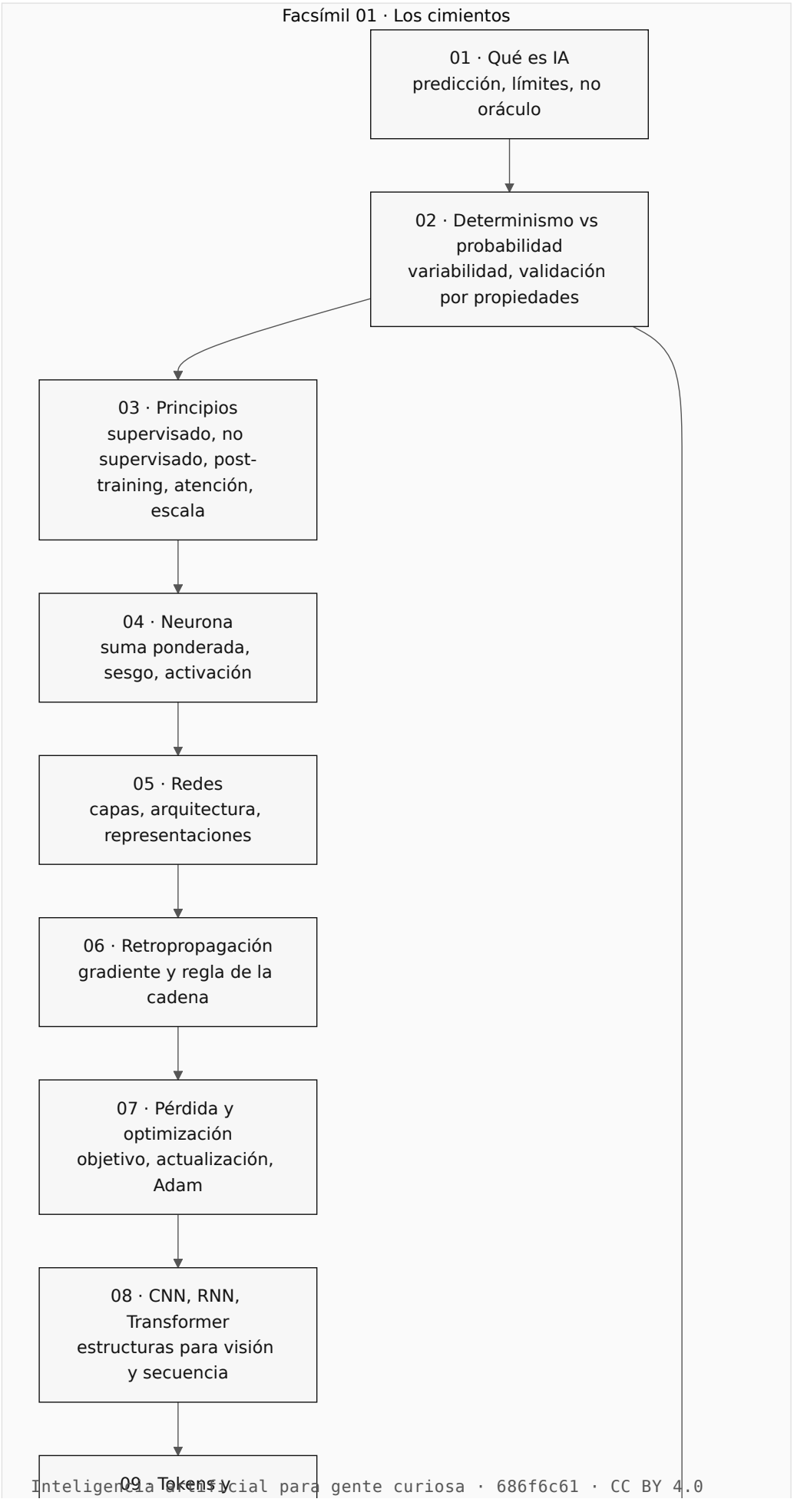
Vuelve al capítulo 11 si: no puedes definir *precision*, *recall* y explicar cuándo importa más uno que otro.

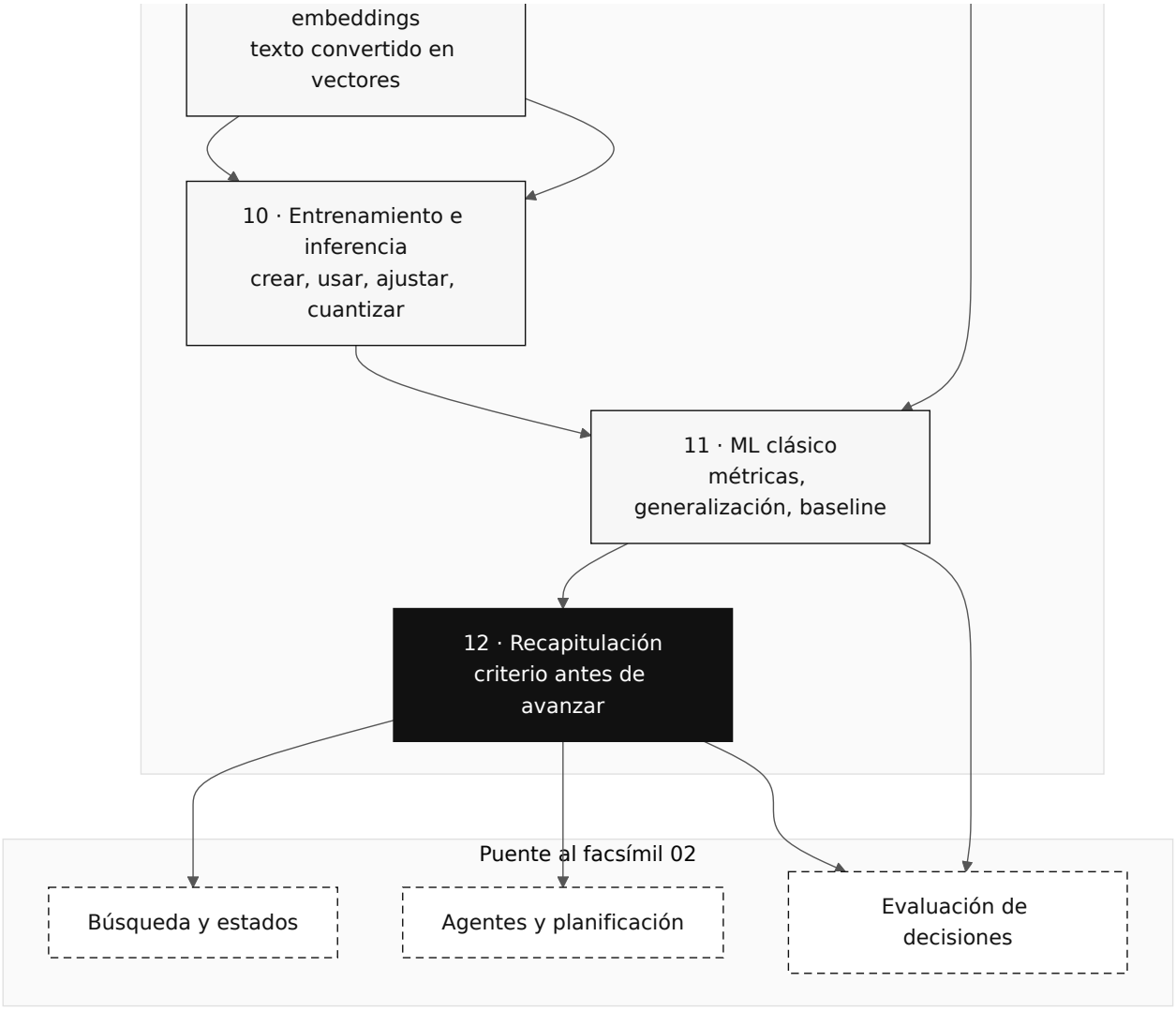
Cómo encaja todo

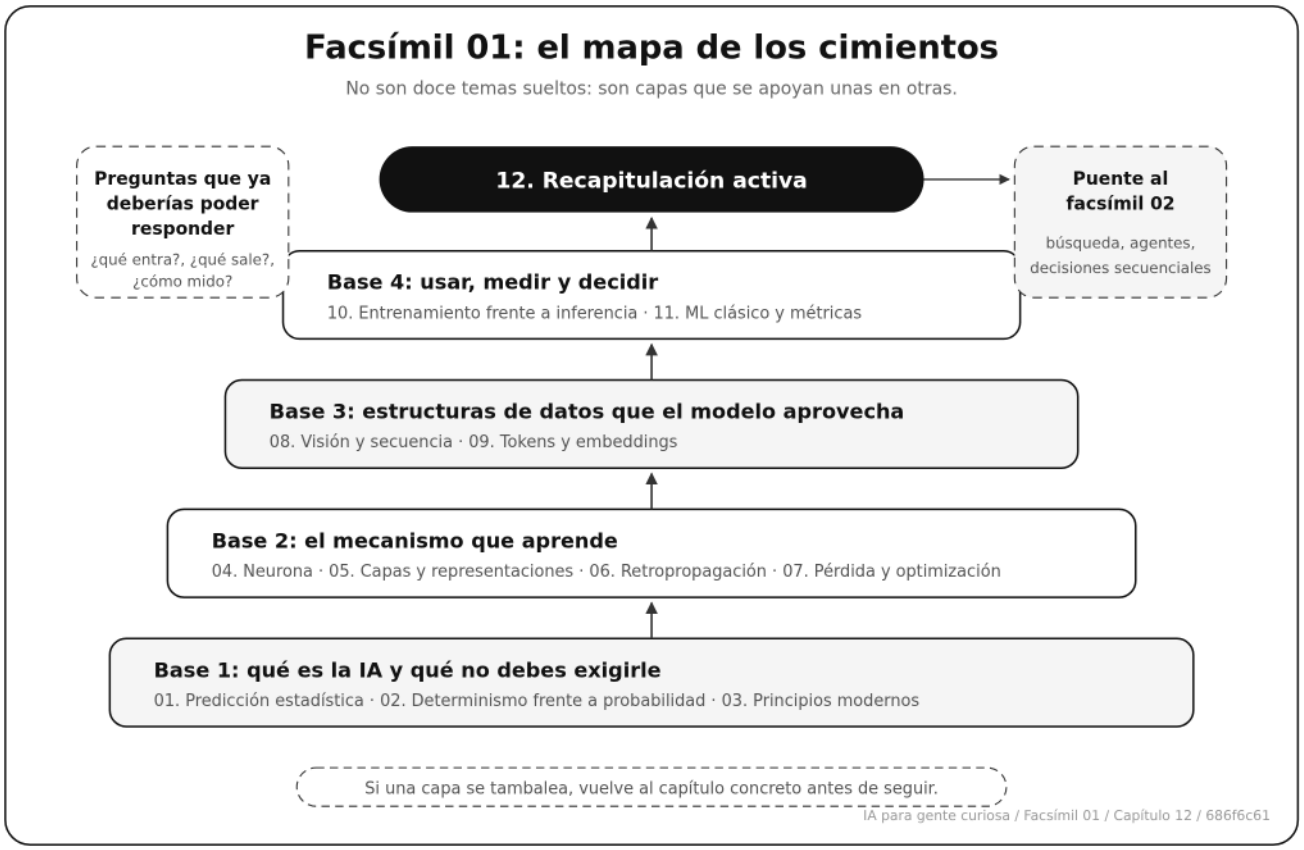
Este mapa no intenta repetir el facsímil entero. Léelo como una escalera: primero aclaramos qué puede y qué no puede hacer la IA, después construimos el mecanismo que aprende, luego convertimos texto e imágenes en representaciones matemáticas y, al final, aprendemos a usar, medir y comparar modelos.

La salida natural de este mapa es el facsímil 2. Allí la pregunta deja de ser «qué es un modelo» y pasa a ser «cómo busca, decide y planifica un sistema cuando tiene varias acciones posibles».

Facsímil 01 · Los cimientos




embeddings
texto convertido en vectores
10 · Entrenamiento e inferencia
crear, usar, ajustar, cuantizar
11 · ML clásico
métricas, generalización, baseline
12 · Recapitulación
criterio antes de avanzar
Puente al facsímil 02
Búsqueda y estados
Agentes y planificación
Evaluación de decisiones



Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Creer que recordar títulos es entender el facsímil	Puedes recitar “tokens, embeddings, backpropagation” y aun así no saber conectar las piezas.	Reexplica cada capítulo con un ejemplo propio y una fórmula mínima cuando exista.
Saltar al facsímil 2 con cimientos flojos	La búsqueda, los agentes y la planificación se apoyan en los conceptos de coste, error, representación y evaluación.	Si fallas una pregunta de la checklist, vuelve al capítulo concreto antes de seguir.
Confundir uso práctico con comprensión	Saber llamar a una API no implica entender inferencia, tokens, embeddings o evaluación.	Para cada herramienta que uses, escribe qué entrada transforma, qué salida produce y cómo medirías si funciona.
No medir	Sin métricas, cualquier demo parece buena hasta que falla en producción.	Antes de avanzar, identifica para cada sistema qué métrica usarías: pérdida, precisión, recall, coste o latencia.

Vocabulario aprendido

TÉRMINO	QUÉ DEBERÍAS PODER EXPLICAR YA
Inteligencia artificial	Sistema que transforma entradas en salidas útiles aprendiendo patrones, no una entidad consciente ni un oráculo.
Sistema probabilístico	Sistema cuya salida puede variar porque trabaja con distribuciones, muestreo e incertidumbre.
Neurona artificial	Función que combina entradas, pesos, sesgo y activación para producir una señal.
Red neuronal	Composición de muchas neuronas en capas que aprenden representaciones cada vez más útiles.
Gradiente	Dirección matemática que indica cómo cambiar un parámetro para modificar la pérdida.
Función de pérdida	Medida formal de error que define qué intenta mejorar el entrenamiento.
Token	Unidad mínima de texto que procesa un modelo de lenguaje.
Embedding	Vector numérico que representa texto, imagen u otra señal en un espacio matemático.
Inferencia	Uso de un modelo ya entrenado para producir una salida ante una entrada nueva.
Métrica	Medida que permite decidir si un modelo funciona para el objetivo real, no solo si parece convincente.

Antes de pasar página

Responde sin consultar los capítulos. Sé honesto con tu propio nivel: si fallas, el número te dice dónde volver.

- [ ] **1.** ¿Puedo explicar por qué un LLM no «piensa» ni «entiende», usando el concepto de predicción del siguiente token?
- [ ] **2.** ¿Sé por qué `temperature = 0` no garantiza que el modelo dé siempre la misma respuesta?
- [ ] **3.** ¿Puedo nombrar los cinco principios de la IA moderna y dar un ejemplo concreto de cada uno?
- [ ] **4.** ¿Puedo escribir la fórmula de una neurona artificial de memoria y calcular su salida para valores concretos?

- [ ] **5.** ¿Puedo explicar qué aprende cada capa de una red neuronal profunda?
- [ ] **6.** ¿Puedo calcular a mano el gradiente $\partial E/\partial w$ para una neurona con `x=2, w=0.4, b=-0.1, y=1` ?
- [ ] **7.** ¿Sé cuándo usar *cross-entropy* y cuándo MSE? ¿Y qué hace Adam que no hace SGD?
- [ ] **8.** ¿Puedo comparar CNN, RNN, LSTM y Transformer explicando la principal ventaja de cada una?
- [ ] **9.** ¿Entiendo la diferencia entre token y embedding? ¿Sé qué mide la similitud coseno?
- [ ] **10.** ¿Puedo explicar la diferencia entre entrenamiento, inferencia, *fine-tuning* y cuantización?
- [ ] **11.** ¿Sé qué mide la precisión y qué mide el *recall*? ¿Cuándo sacrificaría una por la otra?
- [ ] **12.** ¿Puedo explicar el *overfitting* y cómo detectarlo?

Diagnóstico: 10-12 respuestas sólidas: puedes avanzar al facsímil 2. 7-9: repasa los capítulos que fallaste. Menos de 7: vuelve a leer el facsímil con calma. No hay prisa. Los cimientos son eso: cimientos.

En resumen

IDEA FUERZA	DETALLE
La IA es predicción estadística, no conciencia.	Cada decisión de ingeniería depende de entender esto.
La neurona es el átomo, la red la molécula, la retropropagación el motor.	$y = f(\sum_i w_i x_i + b)$ → capas → $\partial E/\partial w$ → $w \leftarrow w - \eta \cdot \partial E/\partial w$. Todo se reduce a esto.
Los embeddings son el puente entre texto y matemáticas.	Sin ellos, las redes no podrían procesar lenguaje.
El ML clásico no ha muerto.	Para datos tabulares, pocos ejemplos o necesidad de explicabilidad, sigue siendo la mejor opción.
Si no sabes medir, no sabes si funciona.	*Precision*, *recall*, F1, validación, *overfitting*: el vocabulario para saber si tu modelo sirve.

Recursos para seguir: leer, construir y experimentar

Has recorrido los cimientos: la neurona, las redes, cómo aprenden, los tokens y los embeddings. Para que no se queden en teoría, aquí tienes recursos reales organizados por lo que quieras hacer.

Para experimentar sin escribir código. Casi todo lo de este facsículo se puede tocar en el navegador, y de hecho lo has hecho en las cajas «Pruébalo en 5 minutos» de cada capítulo: el TensorFlow Playground (`playground.tensorflow.org`) para ver una red aprender, ajustar capas y reventar el *learning rate*; CNN Explainer (`poloclub.github.io/cnn-explainer`) para mirar dentro de una red convolucional; el tokenizador de OpenAI (`platform.openai.com/tokenizer`) y el Embedding Projector (`projector.tensorflow.org`) para ver cómo el texto se vuelve números con significado; y Teachable Machine (`teachablemachine.withgoogle.com`) para entrenar tu propio clasificador con la webcam. Si quieres entenderlo en vídeo, la serie de 3Blue1Brown sobre redes neuronales es una de las mejores explicaciones visuales que existen.

Para construir. Cuando quieras pasar de jugar a programar, las librerías de referencia son PyTorch y TensorFlow/Keras para redes neuronales, y scikit-learn para el machine learning clásico (clasificadores, regresión, árboles). Para modelos ya entrenados que puedes usar o ajustar, Hugging Face (`huggingface.co`) es el punto de partida. No necesitas dominar las matemáticas para empezar: necesitas entender, y eso ya lo tienes.

Para leer. Cada capítulo cierra con su «Para saber más». Si tuvieras que quedarte con una base: entender bien la neurona y la retropropagación, porque todo lo demás (incluidos los grandes modelos de lenguaje de los siguientes facsículos) se construye encima de esas dos ideas. Los kits de este facsículo son el puente: son deterministas a propósito para que veas el mecanismo sin ruido, y te dejan a un paso de las librerías de verdad.

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

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5-32. https://doi.org/10.1023/A:1010933404324

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

Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The elements of statistical learning* (2.ª ed.). Springer. https://web.stanford.edu/~hastie/ElemStatLearn/

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

Kingma, D. P. y Ba, J. (2015). Adam: a method for stochastic optimization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1412.6980

LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539

McCulloch, W. S. y Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), 115-133. https://doi.org/10.1007/BF02478259

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386-408. https://doi.org/10.1037/h0042519

Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0

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

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

Cuadernos para practicar

Has decidido, calculado y medido sobre los cuatro cimientos del facsímil; estos cuadernos te dejan *tocar* lo que has aprendido. Son *notebooks* que se abren en Google Colab —gratis, en el navegador— o te puedes descargar para correr en tu ordenador. Cada uno está explicado paso a paso, con salidas reales, y dice de qué capítulo sale. Empieza por el que más te pique.

Una red neuronal a mano con NumPy

Qué practicas: construir y entrenar una red neuronal desde cero, sin librerías de *deep learning*. **Dónde encaja:** capítulos 4 a 7 (la neurona, las capas, la retropropagación y los optimizadores). **Qué necesitas:** un navegador. Corre en CPU; ni GPU ni claves.

Partes de un caso cotidiano —un riego que debe activarse «cuando los dos sensores discrepan»— que esconde el problema XOR: el que enseñó a la IA, en los años setenta, que una sola neurona no basta. Primero lo intentas con una neurona y la ves *encogerse de hombros*: devuelve 0.5 para todo y acierta 2 de 4. Luego añades una capa oculta de cuatro neuronas, escribes la retropropagación *a mano* (sin cajas negras) y la red pasa a acertar 4 de 4 con total seguridad. Por el camino ves la curva de pérdida bajando época a época y la frontera de decisión *curvándose*, que es justo lo que una recta no podía hacer. Cierra con cuatro experimentos para que rompas cosas: quita la capa oculta, sube el ritmo de aprendizaje, cambia la semilla.

No es un «hola mundo»: es ver, con números reales, por qué la profundidad existe.

► Abrir en Google Colab **↓ Descargar el cuaderno (.ipynb)**

Machine learning clásico que decide de verdad

Qué practicas: clasificar texto con un modelo clásico (TF-IDF + regresión logística) y decidir según el coste de cada error. **Dónde encaja:** capítulo 11 (*machine learning* clásico) y capítulo 2 (determinista frente a probabilístico). **Qué necesitas:** un navegador. Corre en CPU; sin GPU ni claves.

Montas un clasificador para una bandeja de soporte: leer cada mensaje y decidir si es una *incidencia* (algo roto, hay que actuar) o una *consulta* (una duda que espera). Lo entrenas con mensajes reales en español, lo mides con una **matriz de confusión** —que no te dice solo cuánto acierta, sino *cómo* se equivoca— y descubres que una incidencia tratada como consulta es el error caro: un sistema roto que nadie atiende. Entonces mueves el **umbral** de decisión: bajas el listón para atrapar la incidencia que se escapaba (de 1 a 0) a cambio de molestar a alguna consulta de más (de 1 a 7). Cierras leyendo qué palabras delatan a cada clase, porque un modelo clásico —a diferencia de una red— se puede abrir y entender.

Es el 80% de los problemas de empresa resueltos sin un solo gigavatio: saber cuándo basta con lo clásico es criterio, no pereza.

► Abrir en Google Colab **↓ Descargar el cuaderno (.ipynb)**

Entrenamiento vs inferencia: el sobreajuste en directo

Qué practicas: distinguir memorizar de aprender, y por qué un error de entrenamiento bajo puede engañarte. **Dónde encaja:** capítulo 10 (entrenamiento frente a inferencia). **Qué necesitas:** un navegador. Corre en CPU; sin GPU ni claves.

Ajustas curvas de complejidad creciente a unos pocos datos con ruido y ves, en una sola imagen, el error más caro del *machine learning*. Un polinomio de grado 1 se queda corto; uno de grado 15 pasa por todos los puntos de entrenamiento —incluido el ruido— y entre punto y punto se dispara. Luego dibujas las dos curvas que lo cuentan todo: el error de entrenamiento, que solo baja, y el de prueba (datos no vistos), que baja y vuelve a subir formando una **U**. El error de entrenamiento es mínimo justo en el grado más complejo, pero el modelo que generaliza está en el grado 4. Si solo miraras lo primero, elegirías el peor modelo con total confianza.

No es teoría: es la diferencia exacta entre una demo que deslumbra y un sistema que aguanta en producción.

► Abrir en Google Colab **↓ Descargar el cuaderno (.ipynb)**

Notas

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

2. 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 ↩

3. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. Los capítulos 13-14 abordan la incertidumbre y el razonamiento probabilístico. ↩

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

5. McCulloch, W. S. y Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), 115-133. https://doi.org/10.1007/BF02478259 ↩

6. Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386-408. https://doi.org/10.1037/h0042519 ↩

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

8. LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539 ↩

9. Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0 ↩

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

.1. Kingma, D. P. y Ba, J. (2015). Adam: a method for stochastic optimization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1412.6980 ↩

.2. Krizhevsky, A., Sutskever, I. y Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. En *Advances in Neural Information Processing Systems 25* (pp. 1097-1105). https://papers.nips.cc/paper/4824 ↩

.3. Hochreiter, S. y Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735 ↩

.4. 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 ↩

.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/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html ↩

.6. Mikolov, T., Chen, K., Corrado, G. y Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv:1301.3781. https://arxiv.org/abs/1301.3781 ↩

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

.8. Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The elements of statistical learning* (2.ª ed.). Springer. https://web.stanford.edu/~hastie/ElemStatLearn/ ↩

.9. Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5-32. https://doi.org/10.1023/A:1010933404324 ↩