Facsímil 01

Los cimientos

Capítulo 02

Sistemas deterministas frente a sistemas probabilísticos

686f6c61 · 2026

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.

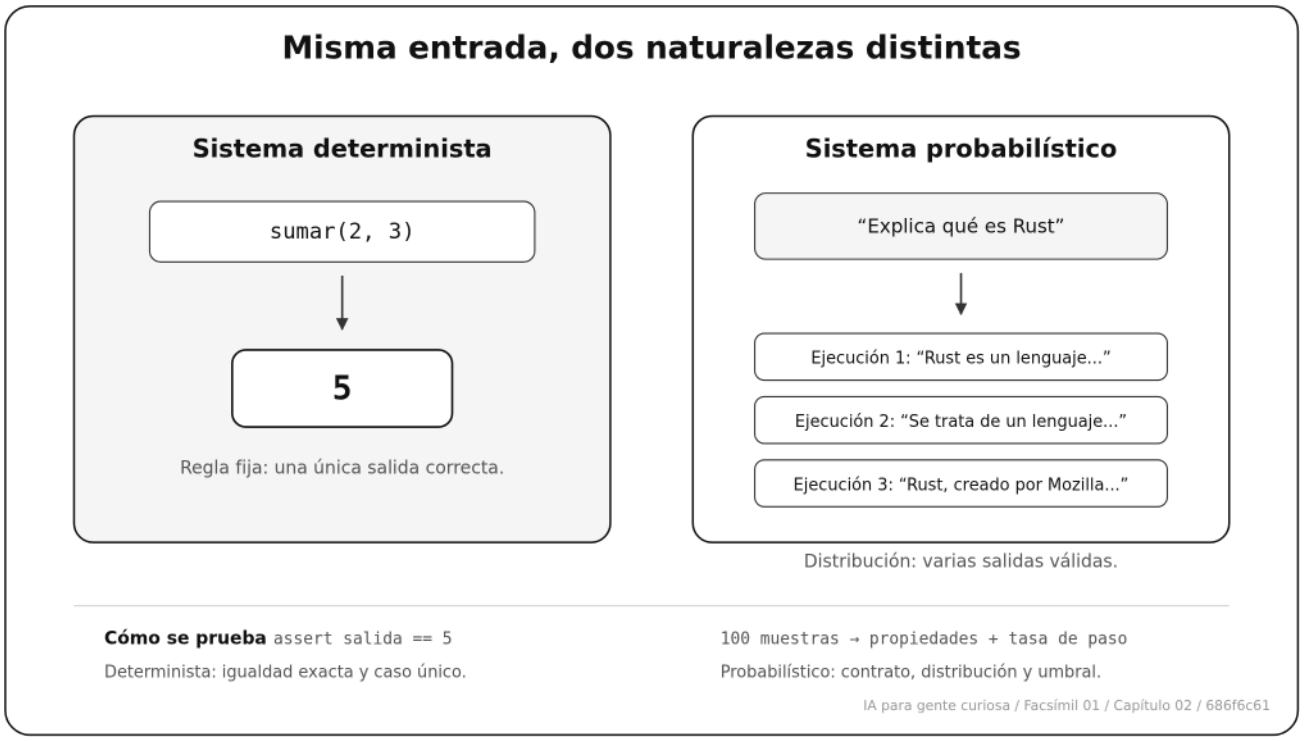
Esa palabra, muestrear, es la clave. No elige el token con mayor probabilidad (aunque puede configurarse para que lo haga). Elige un token al azar, pero con más probabilidad de elegir los que tienen puntuación alta. Es como lanzar un dado cargado: el 6 sale más a menudo, pero de vez en cuando sale un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

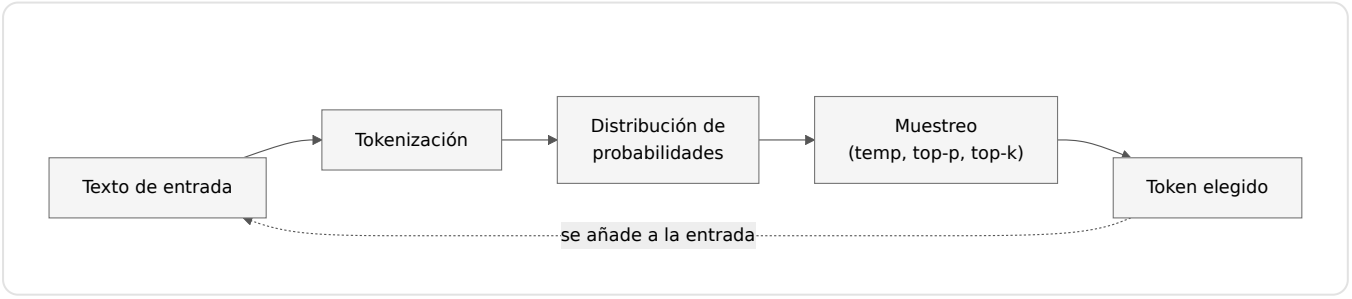
- `top-p`: en lugar de considerar todos los tokens, el modelo solo considera los más probables hasta que sus probabilidades suman `p`. Si `top-p = 0.9`, el modelo elige entre el conjunto mínimo de tokens que suman el 90 % de la probabilidad total. Con la distribución de antes (`azul` 0,84, `gris` 0,11, `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.

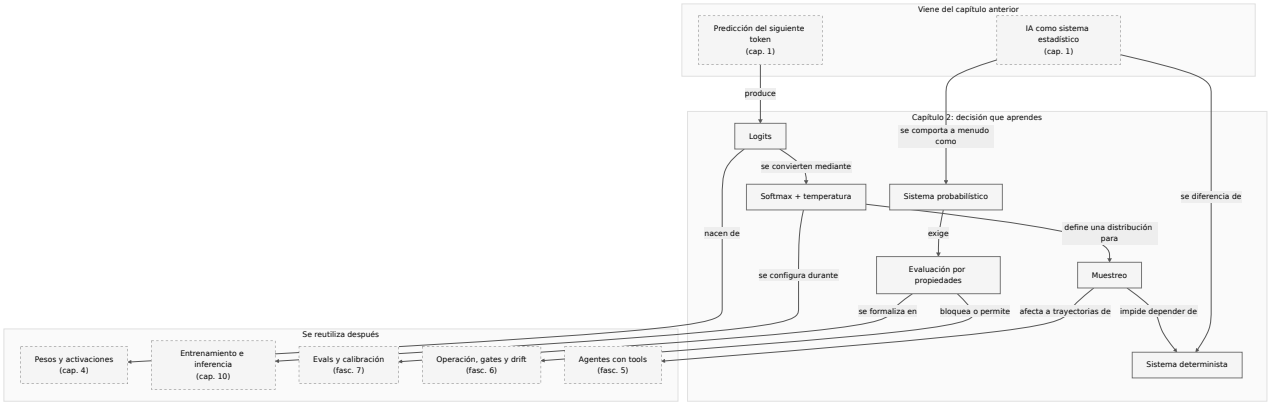
Dónde solía tropezar yo

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
Evaluación por propiedades
Muestreo
se configura durante
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 ↩