Bricoman(IA) · Anthropic jlens

# Radiografiar un LLM capa a capa con la Jacobian lens

Taller técnico · revisión 2026-07-09

## Una reproducción crítica de jlens sobre GPT-2 y Gemma 4 E2B: qué revela la lens, qué no revela y cómo leer las capas sin exagerar conclusiones.

686f6c61 · 2026

**Criterio de lectura.** Cuando esta pieza usa expresiones como "aparece", "lee" o "verbaliza", se refiere a rankings de tokens producidos por la Jacobian lens o por la salida final del modelo. No debe leerse como consciencia, intención ni conocimiento garantizado: es una herramienta lineal de interpretabilidad sobre activaciones internas.

---

## 1. De qué va esto y por qué importa

Los grandes modelos de lenguaje (LLM) son cajas negras. Les metes texto y devuelven texto, pero lo que ocurre entre medias, qué representación interna se forma, en qué capa y para qué token, es opaco. Auditar qué pasa dentro no es una curiosidad académica: es la diferencia entre confiar ciegamente en una salida y poder auditarla. Si un modelo dice que la moneda de Italia es el "called", ¿es porque no aparece una representación útil o porque aparece en una capa intermedia y se pierde antes de la salida? Sin herramientas para mirar dentro, no puedes responder.

La **interpretabilidad mecanística** intenta abrir esas cajas. Una de sus herramientas más simples y potentes es la **lens** (lente): una transformación que proyecta activaciones de capas intermedias al espacio del vocabulario, de modo que podamos "leer" qué token llevaría cada capa si el modelo se detuviera ahí. Es como una radiografía del modelo: no te dice qué hace cada neurona, pero sí qué token queda legible en cada capa.

En junio de 2026, Anthropic publicó el código de acompañamiento del artículo *Verbalizable Representations Form a Global Workspace in Language Models* (Anthropic, 2026), bajo el nombre `jlens` (Jacobian lens). La tesis del paper es profunda: las representaciones que sobreviven al transporte lineal de la lens son las **verbalizables**, y forman un *global workspace* (espacio de trabajo global) en el sentido de Baars (1988), un cuello de botella por el que pasa la información que el modelo puede "decir". La lens, entonces, no es solo una herramienta de diagnóstico: es una forma de inspeccionar qué información queda verbalizable en el espacio interno del modelo. No implica consciencia, intención ni creencias: es una lectura lineal de activaciones.

Esta pieza de Bricoman(IA) reproduce, paso a paso y sobre hardware de consumo (una GPU de 16 GB), el pipeline completo:

1. instalar `jlens`,
2. aplicar una lens **preajustada** a `gpt2-small` y a **Gemma 4 E2B**,
3. **ajustar** una lens desde cero,
4. generar la **visualización interactiva** *slice-vis*,
5. interpretar los resultados, incluyendo el hallazgo de que en Gemma 4 E2B el token `euros` aparece en el top-5 de la lens en L28, aunque la salida final prioriza otros tokens,

6. crear **ejemplos propios** (no del repo de Anthropic) para mostrar la lens en acción.

Todo el código y las salidas que verás son **reales**, verificadas en la máquina del autor. Las figuras son nuestras, generadas con matplotlib y PIL, firmadas con `iaparagentecuriosa.686f6c61.dev`.

### 1.1. Por qué esto importa

La lens responde a una pregunta que el log final no puede responder: ¿aparece la respuesta correcta como representación legible en alguna capa intermedia? Esa pregunta cambia cómo diagnosticas un fallo, cómo comparas modelos y dónde intervienes si quieres modificar el comportamiento de un LLM.

Concretamente, la lens te da tres cosas que no tienes de otra forma:

- **Auditoría de conocimiento**: si un modelo falla, la lens ayuda a distinguir entre dos patrones: la representación correcta no aparece en ninguna capa observada, o aparece en una capa intermedia y luego se diluye. Sin la lens, solo ves la salida final y tienes que adivinar.
- **Detección de la capa de decisión**: en qué capa la respuesta empieza a aparecer como candidata fuerte te dice dónde intervenir si quieres modificar su comportamiento. Si la lens coloca el token correcto arriba en L28 pero la salida final no lo mantiene, sabes que la intervención debe ir entre la 28 y la 34. Esto es la base del *steering* y del *activation patching* (Meng et al., 2023).
- **Comparación de modelos**: dos modelos pueden dar la misma salida por razones internas muy distintas. Uno puede hacer aparecer el token correcto en la capa 15 y otro en la 28. Este sistema de Lens nos lo revela, y esa diferencia importa: el primer modelo es más eficiente en su procesamiento, el segundo más profundo.

Estas tres capacidades convierten a la lens en una herramienta de auditoría. No te dice qué hacer con el modelo, pero te dice dónde mirar.

### 1.2. Qué necesitas saber

La pieza asume que programas en Python, conoces PyTorch a nivel de usuario y entiendes la estructura básica de un transformer (capas, residual stream, attention, MLP, unembedding). Si no es así, la sección 2 te sitúa con un ejemplo concreto; si ya lo dominas, salta a la sección 3.

### 1.3. Hardware y software usado

| COMPONENTE | VALOR |
|---|---|
| GPU | NVIDIA GeForce RTX 5060 Ti (Blackwell, compute 12.0) |
| VRAM | 16 GB GDDR7 |
| RAM | 123 GB DDR5 |
| CPU | 32 núcleos |
| SO | Linux (Ubuntu), driver NVIDIA 580.159.03 |
| CUDA | 13.0 (vía PyTorch 2.13.0+cu130) |
| Python | 3.12.12 (gestionado con `uv`) |
| Gestor de paquetes | `uv` 0.11.8 |
| `torch` | 2.13.0+cu130 |
| `transformers` | 5.13.0 |
| `jlens` | 0.1.0 (clonado del repo oficial) |
| `accelerate` | 1.14.0 |

**Nota sobre Blackwell.** La RTX 5060 Ti pertenece a la arquitectura Blackwell (compute capability 12.0). Requiere CUDA 12.8 o superior; por eso usamos `torch 2.13.0+cu130`, que arrastra los paquetes `nvidia-*` de la rama CUDA 13.0. Con torches más antiguos (CUDA 12.1) el driver levanta la GPU pero los kernels caen en *no kernel image available for execution on the device*.

---

# 2. Contexto teórico

## 2.1. El residual stream y el unembedding

Un transformer decodificador (causal LM) procesa una secuencia de tokens $x = (x_1, \ldots, x_T)$ mediante $L$ bloques residuales. Tras la capa de embedding, cada posición $t$ lleva un vector $h_t^{(0)} \in \mathbb{R}^d$ (el *residual stream*). Cada bloque $l$ actualiza ese vector:

$$h_t^{(l)} = h_t^{(l-1)} + \text{Bloque}_l\left(h^{(l-1)}\right)_t, \qquad l = 1, \ldots, L.$$

La idea clave del residual stream es que la información fluye como un río: cada capa añade su contribución (atención + MLP) sin sobrescribir lo anterior. Esa estructura aditiva es lo que permite "leer" el estado en cualquier capa intermedia: $h_t^{(l)}$ contiene la suma de todo lo que las capas 1 a $l$ han añadido.

Tras la última capa, una norma final y una matriz de unembedding $W_U \in \mathbb{R}^{|V| \times d}$ producen los logits sobre el vocabulario $V$:

$$\text{logits}_t = W_U \,\text{Norm}\left(h_t^{(L)}\right), \qquad p(\cdot \mid x_{<t+1}) = \text{softmax}(\text{logits}_t).$$

La predicción del siguiente token es $\arg\max_{v \in V} \text{logits}_t$. El problema de la interpretabilidad es: **¿qué información lleva $h_t^{(l)}$ en capas intermedias, antes de llegar a $L$?**

### Un ejemplo concreto

Considera el prompt `"La capital de Francia es"` y un modelo con $d = 4$ (dimensiones ridículamente pequeñas, solo para poder dibujar) y vocabulario $V = \{$París, Madrid, Berlín, el, es, capital$\}$. La secuencia se tokeniza como 5 posiciones. Tras el embedding, cada posición $t$ tiene un vector $h_t^{(0)} \in \mathbb{R}^4$. La siguiente figura resume el ejemplo; los números son sintéticos y solo sirven para explicar la mecánica:

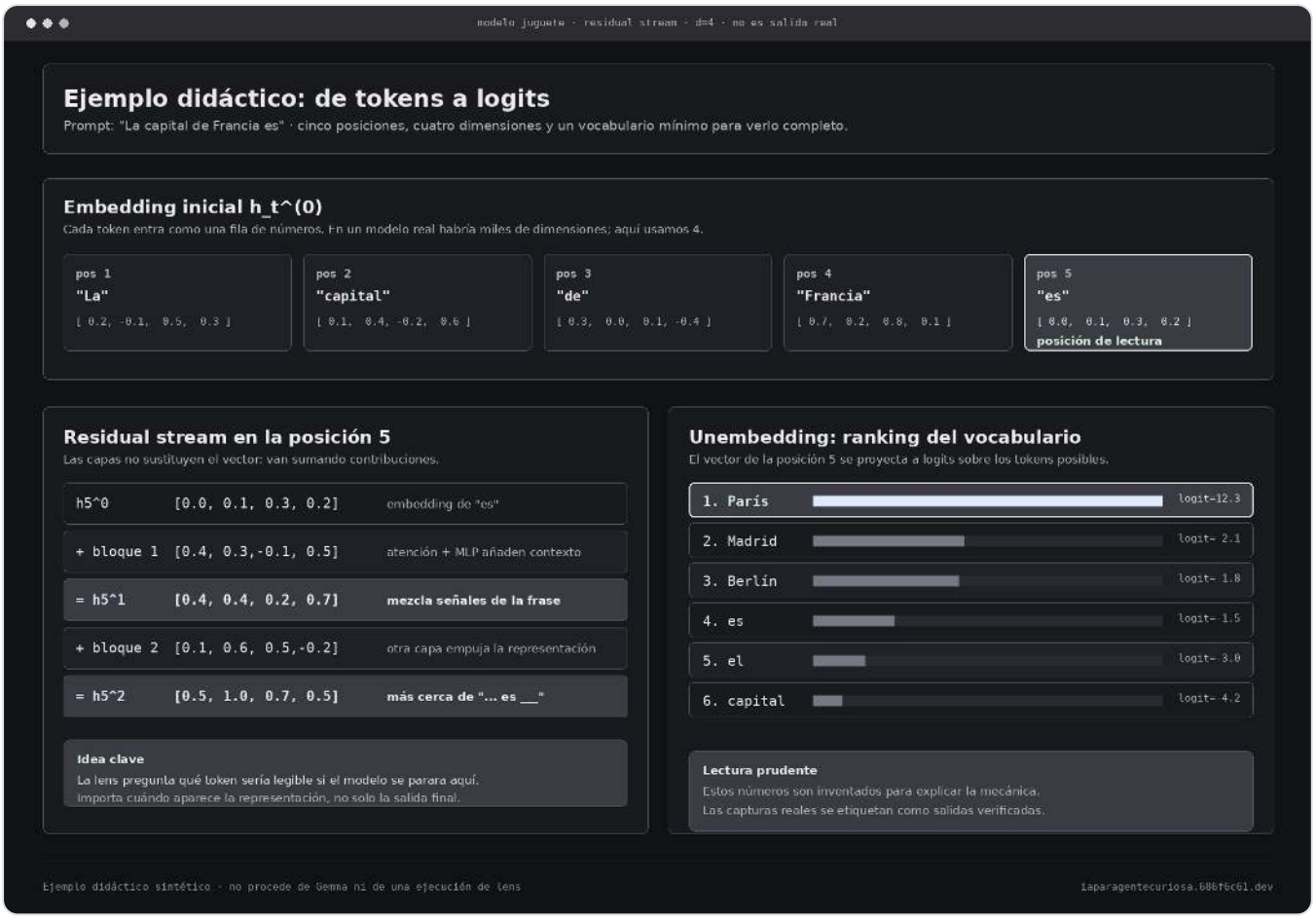



*Figura 0. Ejemplo didáctico sintético de residual stream y unembedding para el prompt "La capital de Francia es". No es una salida real de Gemma ni de la Jacobian lens: usa cinco tokens, cuatro dimensiones y un vocabulario mínimo para que el cálculo quepa en una página.*

Tras la capa 1, el bloque añade su contribución (atención sobre el contexto + MLP):

$$h_5^{(1)} = h_5^{(0)} + \text{Bloque}_1(h^{(0)})_5 = [0.0, 0.1, 0.3, 0.2] + [0.4, 0.3, -0.1, 0.5] = [0.4, 0.4, 0.2, 0.7]$$

La posición 5 ("es") ahora lleva también información de "Francia" (por la atención). Tras la capa 2, otro bloque añade más:

$$h_5^{(2)} = h_5^{(1)} + \text{Bloque}_2(h^{(1)})_5 = [0.4, 0.4, 0.2, 0.7] + [0.1, 0.6, 0.5, -0.2] = [0.5, 1.0, 0.7, 0.5]$$

Y así hasta la capa $L$. El residual stream en la posición 5 va acumulando, capa a capa, la representación de "la capital de Francia es ___". Al final, el unembedding $W_U \in \mathbb{R}^{6\times 4}$ (6 tokens en el vocabulario, 4 dimensiones) produce los logits:

$$\underset{\text{París}\quad\text{Madrid}\quad\text{Berlín}\quad\text{el}\quad\text{es}\quad\text{capital}}{\text{logits}_5 = W_U\,\text{Norm}(h_5^{(L)}) = [12.3\,,\ \ 2.1\ \ ,\ 1.8\ \ ,-3.0\,,-1.5\,,-4.2]}$$

El $\arg\max$ es "París". Pero, ¿en qué capa el residual $h_5^{(l)}$ empezó a hacer legible la respuesta París? La lens responde a eso: proyecta $h_5^{(l)}$ al espacio del vocabulario en cada capa y nos dice qué token llevaría si el modelo se detuviera ahí. Si en la capa 15 de 24 la lens ya lee "París",

sabemos que esa representación se forma a mitad del modelo, no solo al final.

Este ejemplo, en miniatura, es exactamente lo que haremos en la sección 9 con Gemma 4 E2B y el prompt de la moneda de Italia, donde veremos que `euros` aparece en la capa 28 aunque la salida final priorice otros tokens.

## 2.2. La familia de las *lens*

Una *lens* responde a esa pregunta proyectando $h_t^{(l)}$ al espacio del vocabulario. La diferencia entre lens es **qué transformación** aplican. El siguiente diagrama muestra el pipeline común:

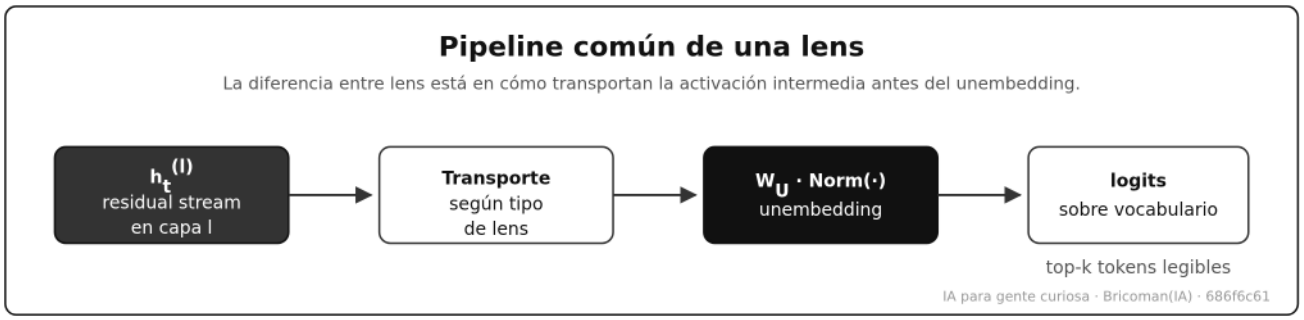



*Figura 1. Pipeline común a todas las lens. El residual stream $h^{(l)}$ en una capa intermedia se transforma (según el tipo de lens), se proyecta con el unembedding $W_U$ y produce logits sobre el vocabulario. La diferencia entre lens está solo en la transformación del medio. Figura propia.*

### 2.2.1. Logit lens (Nostalgebrista, 2020)

La más simple: aplica directamente el unembedding de la capa final a activaciones de capas previas. Asume que el residual stream es *estacionario*: que el espacio donde vive $h^{(l)}$ es el mismo donde vive $h^{(L)}$.

$$\text{lens}_l^{\text{logit}}(h) = W_U \,\text{Norm}(h), \qquad h = h_t^{(l)}.$$

**La analogía de la calle.** Imagina que el residual stream es una avenida larga con $L$ cruces. En cada cruce hay un puesto de fruta que añade algo a tu cesta. La logit lens es como si, al llegar al cruce 5, abrieras tu cesta y la llevaras directamente a la caja del supermercado del final de la avenida para ver qué te cobrarían. El problema es que el supermercado del final está calibrado para la cesta tal como llega al cruce $L$, no al cruce 5: las frutas del cruce 5 pueden no tener el mismo precio que las del cruce $L$. Si la avenida es corta (pocas capas), no pasa nada; si es larga (muchas capas), los precios ya no cuadran.

**Ejemplo concreto.** En GPT-2 small (12 capas), la logit lens puede funcionar razonablemente: si en la capa 8 el residual ya está alineado con "París", el unembedding directo lo lee correctamente. Pero en Gemma 4 E2B (35 capas), la logit lens en la capa 5 produce tokens

basura (HTML, puntuación), porque el espacio residual de la capa 5 no es el mismo que el de la capa 34. La deriva del espacio entre capas lo estropea.

Funciona sorprendentemente bien en GPT-2 y modelos pequeños, pero degrada en modelos modernos con muchas capas, porque el espacio residual deriva entre capas: lo que en la capa 5 significa "Italia" puede no significar lo mismo en la capa 30. La logit lens ignora esa deriva.

### 2.2.2. Tuned lens (Belrose et al., 2023)

Añade un *affine probe* entrenado por capa para corregir el sesgo de la logit lens:

$$\text{lens}_l^{\text{tuned}}(h) = W_U \,\text{Norm}(A_l h + b_l)\,,$$

donde $(A_l, b_l)$ se aprenden con supervisión sobre las logits finales.

**La analogía de la calle.** Siguiendo con la avenida: la tuned lens es como si, en cada cruce, contrataras a un traductor que sabe convertir la cesta del cruce 5 al formato que entiende el supermercado del final. Ese traductor se entrena mostrándole muchas cestas del cruce 5 y diciéndole qué cobró el supermercado al final. Aprende la conversión. El problema es que necesitas contratar (entrenar) un traductor distinto para cada cruce, y necesitas datos etiquetados (cestas con su precio final conocido).

**Ejemplo concreto.** Para entrenar la tuned lens de la capa 5 de un modelo, necesitas miles de prompts donde conoces los logits finales. Para cada uno, guardas $h^{(5)}$ y los logits finales, y entrenas $(A_5, b_5)$ para que $W_U\text{Norm}(A_5 h^{(5)} + b_5)$ se parezca a los logits finales. Es un problema de regresión estándar. El coste: un dataset curado y un optimizador por capa. La ventaja: corrige la deriva del espacio residual.

Es más precisa, pero requiere etiquetas y entrenamiento: necesitas un dataset de pares $(h^{(l)}, \text{logits finales})$ y un optimizador.

### 2.2.3. Jacobian lens (Anthropic, 2026)

La que nos ocupa. En vez de suponer estacionariedad (logit) ni aprender un probe (tuned), **transporta linealmente** $h^{(l)}$ al espacio de la capa final usando la **Jacobiana media entrada-salida** del modelo sobre un corpus de texto:

$$\boxed{\text{lens}_l^{\text{Jac}}(h) = W_U \,\text{Norm}(J_l\, h)\,, \qquad J_l = \mathbb{E}\left[\frac{\partial h^{(L)}}{\partial h^{(l)}}\right]}$$

La esperanza se promedia sobre prompts, posiciones de origen y posiciones destino (presentes y futuras) en un corpus tipo *web text*. La Jacobiana $J_l \in \mathbb{R}^{d\times d}$ es la derivada del residual final respecto al residual en la capa $l$: mide cómo una perturbación en $h^{(l)}$ se propaga, en promedio, hasta la salida. No necesita supervisión ni etiquetas: solo *forward* y *backward* sobre texto.

**La analogía de la calle.** En vez de contratar traductores (tuned) o fingir que no hace falta traducir (logit), la Jacobian lens es como si, en cada cruce, soltaras una pelota de prueba y midieras a dónde llega al final de la avenida. Lo haces muchas veces, con muchas pelotas y muchas avenidas (prompts), y promedias. El resultado es un mapa que te dice: "si la cesta en el cruce 5 cambia así, la cesta en el final cambia así". Ese mapa es $J_l$. No necesitas etiquetas ni traductores: el propio sistema (el modelo) te lo cuenta, porque la pelota sigue las leyes del sistema.

**Ejemplo concreto.** Para estimar $J_5$ en Gemma 4 E2B, tomas 1000 prompts de wikitext. Para cada uno, haces un *forward* y, para cada dimensión del residual final, haces un *backward* inyectando una cotangente *one-hot*. El gradiente que llega a la capa 5 te dice cómo cambia $h^{(L)}$ si cambias $h^{(5)}$. Promedias sobre los 1000 prompts y obtienes $J_5$. El coste: 1 *forward* + 96 *backwards* por prompt (para $d = 1536$, $B = 16$). Sin etiquetas, sin dataset curado: solo texto.

El siguiente diagrama compara las tres:

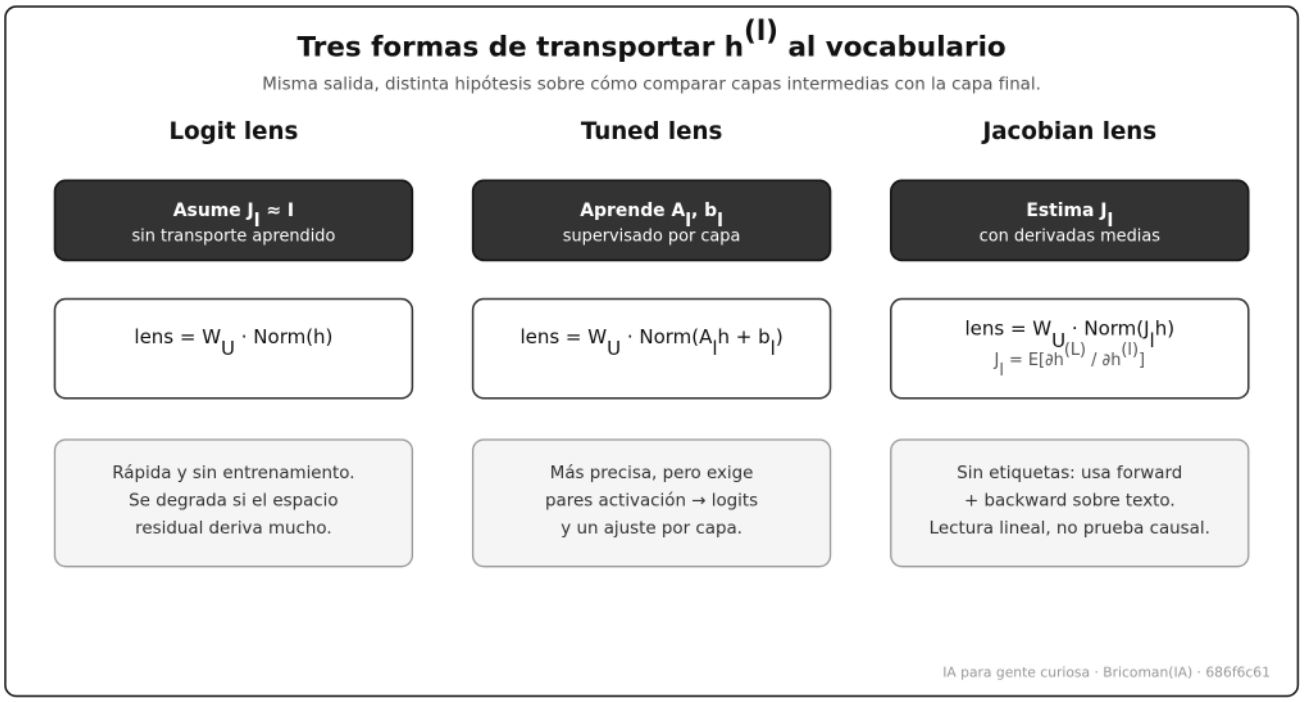



*Figura 2. Las tres lens comparadas. La logit lens asume estacionariedad ($J_l \approx I$); la tuned lens aprende un probe affine supervisado; la jacobian lens estima $J_l$ sin etiquetas, promediando derivadas del modelo sobre un corpus. Figura propia.*

## 2.3. Por qué la Jacobiana es la opción "honesta"

La logit lens asume $J_l \approx I$ (identidad). La tuned lens aprende $J_l$ a partir de etiquetas. La Jacobian lens **estima** $J_l$ directamente del modelo, sin etiquetas, promediando sobre la distribución de datos. Es, en cierto modo, el *mejor predictor lineal* de $h^{(L)}$ a partir de $h^{(l)}$ bajo la distribución del corpus.

¿Por qué "mejor predictor lineal"? Por el teorema de la proyección ortogonal: la mejor aproximación lineal de $h^{(L)}$ dado $h^{(l)}$ (en sentido de error cuadrático medio) es $\mathbb{E}[h^{(L)}] + \mathrm{Cov}(h^{(L)}, h^{(l)})\,\mathrm{Cov}(h^{(l)})^{-1}(h^{(l)} - \mathbb{E}[h^{(l)}])$. La Jacobiana media captura exactamente la parte lineal de esa proyección, sin necesidad de estimar covarianzas empíricas: la derivada del modelo *es* la respuesta lineal del sistema.

**La analogía de la calle.** Imagina que quieres predecir cuánto pesará una cesta al final de la avenida sabiendo lo que pesa en el cruce 5. La logit lens dice: "pesan lo mismo" (asume identidad). La tuned lens dice: "déjame ver muchas cestas y te entreno un predictor" (necesita etiquetas). La Jacobian lens dice: "voy a empujar la cesta un poquito en el cruce 5 y mido cuánto cambia al final; lo hago muchas veces y promedio" (sin etiquetas, solo el sistema). El empujón es la derivada. El promedio de empujones es la Jacobiana. Y el teorema de proyección garantiza que ese promedio de empujones es el mejor predictor lineal que puedes construir sin etiquetas.

**Ejemplo concreto.** Si en la capa 5 el residual "enciende" la dirección de "Italia", la Jacobiana $J_5$ te dice cuánto se "enciende" la dirección de "euro" en la capa final. Si la dirección de "Italia" en la capa 5 se propaga fuertemente a "euro" en la capa final, $J_5$ tendrá un valor alto en esa entrada. Si no se propaga, será cercano a cero. La lens usa eso para transportar: multiplica $h^{(5)}$ por $J_5$ y obtiene una aproximación de $h^{(L)}$ que el unembedding sí sabe leer.

La ventaja práctica: no necesitas etiquetas ni un dataset curado. Solo texto. El modelo mismo te dice, vía sus derivadas, cómo se propaga la información.

## 2.4. El estimador práctico

Calcular $J_l$ exigiendo derivar $h^{(L)}$ respecto a cada componente de $h^{(l)}$ es caro. El estimador del módulo `jlens.fitting` (Anthropic, 2026) lo resuelve así: para cada dimensión de salida $i$, inyecta una cotangente *one-hot* en **todas las posiciones destino válidas a la vez** y hace *backward*. El gradiente en la posición de origen $p$ es entonces

$$g_i(p) = \sum_{p' \geq p} \frac{\partial h^{(L)}_{p'}}{\partial h^{(l)}_{p}}[i, :],$$

y se promedia sobre posiciones de origen $p$. El coste es **un *forward* +** $\lceil d/B \rceil$ ***backwards*** por prompt, donde $B$ es el tamaño de lote de dimensiones ( `dim_batch` ). Las primeras 16 posiciones se excluyen (son *attention sinks* con estadísticas atípicas), y la última posición no tiene *target*.

**La analogía de la calle.** Imagina que la avenida tiene 35 cruces y en cada cruce hay 1536 direcciones posibles (las dimensiones del residual). Para mapear cómo un empujón en el cruce 5 llega al final, no puedes empujar una a una las 1536 direcciones (serían 1536 *backwards* por cruce). El truco es empujarlas en lotes: 16 direcciones a la vez. Así, en vez de 1536 *backwards*, haces 96 (1536 / 16). Y en cada *backward*, no empujas solo en una posición del prompt, sino en todas las posiciones destino a la vez, porque el modelo es causal y lo que pasa en una posición no afecta a las anteriores.

**Ejemplo concreto.** Para Gemma 4 E2B ($d = 1536$, $B = 16$), cada prompt cuesta 1 *forward* + 96 *backwards*. Con 455 prompts, son 43.680 *backwards* en total. En una RTX 5060 Ti, el *fit* completo tardaría varias horas; por eso en esta pieza usamos la lens preajustada para Gemma y reservamos el ajuste desde cero para `gpt2-small`.

¿Por qué se excluyen las 16 primeras posiciones? Porque en los transformers modernos, las primeras posiciones actúan como *attention sinks* (Xiao et al., 2024): acumulan atención residual que no refleja el contenido del prompt, y sus estadísticas son atípicas. Incluirlas sesgaría la media.

**La analogía de la calle.** Los primeros 16 puestos de la avenida son como puestos de "basura" donde la gente deja cosas raras que no tienen que ver con el contenido real de la cesta. Si incluyes esos puestos en el promedio, te sesgan el mapa. Por eso se excluyen: no representan el comportamiento típico del modelo.

## 2.5. El *global workspace* y por qué es importante

La hipótesis del *global workspace* (Baars, 1988; Dehaene, 2014) propone que en el cerebro existe un espacio compartido al que convergen los módulos especializados y desde el que se difunde la información consciente. Esa teoría habla de consciencia humana; aquí la usamos solo como analogía técnica para entender un espacio compartido y verbalizable en un modelo, no para atribuir consciencia al modelo.

**La analogía de la calle.** Imagina el cerebro como una ciudad con muchos talleres especializados: uno ve, otro oye, otro recuerda, otro decide. Cada taller trabaja en su rincón. Pero hay una plaza central (el *global workspace*) a la que todos pueden enviar su resultado, y desde la que se difunde a toda la ciudad. Lo que llega a la plaza es lo que puede difundirse y reportarse. Lo que se queda en el taller no.

Anthropic (2026) sugiere que, en un LLM, el residual stream final cumple ese papel: las representaciones que la lens puede leer son las que el modelo "mantiene verbalizables" en ese momento. La evidencia es que la lens, que es un transporte lineal sin entrenamiento supervisado, consigue leer representaciones coherentes en capas intermedias, eso solo es posible si esas representaciones viven en un espacio "global" que se mantiene estable a lo largo del modelo.

**Ejemplo concreto.** En la sección 9 veremos que, en Gemma 4 E2B, la lens lee "euros" en la capa 28. Eso significa que la representación de "euros" llegó a la "plaza" (el residual stream) en esa capa. Pero la salida final del modelo es "called": la plaza tenía la respuesta correcta, pero algo entre la capa 28 y la 34 la redirigió. La lens nos deja ver qué había en la plaza en cada momento, no solo lo que sale por la puerta al final.

La implicación práctica: si una representación es legible por la lens en la capa $l$, entonces el modelo *podría* verbalizarla en ese punto. Si la salida final difiere, algo entre $l$ y $L$ la pierde, diluye o redirige. Ese tramo de capas es donde tiene sentido mirar si quieres intervenir.

---

## 3. Preparación del entorno

Todo el código se ejecuta en un clon del repo oficial. Crea un directorio de trabajo y clona el repo ahí:

### 3.1. Clonar el repo

```
git clone --depth 1 https://github.com/anthropics/jacobian-lens.git jlens-repo
cd jlens-repo
```

Salida real:

```
Clonando en 'jlens-repo'...
```

### 3.2. Requisitos del `pyproject.toml`

El fichero `pyproject.toml` declara:

```
requires-python = ">=3.10"
dependencies = [
    "torch",
    "huggingface_hub",
    "transformers>=5.5",
    "numpy",
]
```

Dos observaciones importantes:

- **`transformers>=5.5`**: Gemma 4 se incorporó en transformers 5.x. La versión 5.13.0 que instalaremos lo soporta nativamente (clases `Gemma4ForCausalLM`, `Gemma4ForConditionalGeneration`, etc.).
- **`torch`**: sin pin de versión. Para Blackwell necesitamos `torch>=2.10` con CUDA 12.8+. `uv` resuelve automáticamente la rama `cu130`.

### 3.3. Crear el entorno con `uv`

```
uv python install 3.12          # si no la tienes ya
uv venv --python 3.12 .venv
source .venv/bin/activate
uv pip install -e .
uv pip install accelerate matplotlib pillow
```

Salida real (fragmento):

```
Using CPython 3.12.12
Creating virtual environment at: .venv
 + nvidia-nccl-cu13==2.29.7
 + nvidia-nvjitlink==13.3.33
 + torch==2.13.0
 + tokenizers==0.22.2
 + transformers==5.13.0
 + triton==3.7.1
 ...
```

**Por qué Python 3.12 y no 3.14.** El Python del sistema puede ser 3.14, pero la cadena torch/triton aún no tiene wheels estables para 3.14 en el momento de escribir esto. Python 3.12 es el punto dulce: soporte maduro de torch 2.13 y de transformers 5.13.

### 3.4. Verificar CUDA en la GPU

```
python -c "
import torch
print('torch:', torch.__version__)
print('cuda available:', torch.cuda.is_available())
print('cuda version:', torch.version.cuda)
print('device name:', torch.cuda.get_device_name(0))
print('compute capability:', torch.cuda.get_device_capability(0))
x = torch.randn(1000,1000, device='cuda')
y = x @ x.T
torch.cuda.synchronize()
print('matmul en GPU OK, shape:', tuple(y.shape))
"
```

Salida real:

```
torch: 2.13.0+cu130
cuda available: True
cuda version: 13.0
device name: NVIDIA GeForce RTX 5060 Ti
compute capability: (12, 0)
matmul en GPU OK, shape: (1000, 1000)
```

`compute capability (12, 0)` confirma que es Blackwell. Los kernels de torch 2.13+cu130 se ejecutan sin el temido *no kernel image available*.

### 3.5. Token de HuggingFace para Gemma 4

Gemma 4 es un modelo **gated**: hay que aceptar la licencia en la página del modelo y esperar la aprobación de Google. Sin este paso, no puedes descargar los pesos y el resto del tutorial no funciona. Es el único paso del pipeline que depende de un tercero (Google) y que no es automatizable.

#### Paso 1: crear una cuenta en HuggingFace

Si no tienes cuenta, regístrate en https://huggingface.co/join. Es gratis y te da acceso al Hub, donde viven los modelos y las lens.

#### Paso 2: generar un token de acceso

Una vez dentro, ve a **Settings → Access Tokens** (https://huggingface.co/settings/tokens) y crea un token nuevo:

1. Pulsa **New token**.
2. Ponle un nombre (por ejemplo, `jlens`).
3. Elige tipo **Read** (solo lectura, suficiente para descargar modelos).
4. Pulsa **Create**.
5. Copia el token: empieza por `hf_` y tiene unos 37 caracteres. Guárdalo en un sitio seguro, no lo vas a ver otra vez en la web.

#### Paso 3: solicitar acceso a Gemma 4

Gemma 4 es *gated*, lo que significa que Google exige aceptar su licencia antes de dejarte descargar los pesos. Para cada variante que quieras usar:

1. Ve a la página del modelo en HuggingFace:
   - E2B: https://huggingface.co/google/gemma-4-E2B
   - E4B: https://huggingface.co/google/gemma-4-E4B
   - 31B: https://huggingface.co/google/gemma-4-31B

2. Si no has solicitado acceso, verás un banner con el texto *"You need to agree to share your contact information to access this model"*. Pulsa el botón **Agree and access repository**.
3. Rellena el formulario (nombre, email, organización si aplica) y acepta las condiciones de uso.
4. Espera. Google revisa las solicitudes y, en la mayoría de los casos, aprueba en cuestión de minutos u horas. Recibirás un email cuando el acceso esté concedido.

> **Nota.** La aprobación es por cuenta de HuggingFace, no por token. Una vez aprobada, cualquier token de tu cuenta funciona. Si pides acceso a E2B, también te suele dar acceso a E4B y 31B, pero conviene solicitarlo en cada página por separado para evitar sorpresas.

**Paso 4: guardar el token en local**

El token se guarda en `~/.cache/huggingface/token` con permisos `600` (solo tu usuario puede leerlo). HuggingFace lo busca ahí automáticamente:

```
mkdir -p ~/.cache/huggingface
printf 'hf_xxx_tu_token' > ~/.cache/huggingface/token
chmod 600 ~/.cache/huggingface/token
```

También puedes exportarlo como variable de entorno para la sesión actual:

```
export HF_TOKEN=hf_xxx_tu_token
```

Ambas formas funcionan. La primera es persistente; la segunda es temporal.

**Paso 5: verificar que tienes acceso**

Para confirmar que el token funciona y que Google te ha aprobado, intenta descargar el `config.json` del modelo. Si devuelve 200 con contenido JSON real, tienes acceso. Si devuelve 401 o 403, el token es inválido o no tienes acceso aprobado:

```
export HF_TOKEN=hf_xxx_tu_token
curl -s -o /tmp/cfg.json -w "HTTP %{http_code} | size=%{size_download}\n" \
  -H "Authorization: Bearer $HF_TOKEN" \
  https://huggingface.co/google/gemma-4-E2B/resolve/main/config.json
head -c 200 /tmp/cfg.json
```

Salida real en la máquina del autor (acceso aprobado):

```
HTTP 200 | size=4914 bytes
{
  "architectures": ["Gemma4ForConditionalGeneration"],
  "audio_config": { ... },
  "model_type": "gemma4",
  "text_config": {
    "hidden_size": 1536,
    "num_hidden_layers": 35,
    ...
  }
}
```

Si ves `HTTP 401` o `HTTP 403`, revisa el token y la aprobación. Si ves una página HTML en vez de JSON, es la página de *gated* pidiéndote que aceptes la licencia: vuelve al paso 3.

El token del autor **tenía acceso aprobado** a las tres variantes (E2B, E4B, 31B).

---

## 4. Dónde conseguir las lens preajustadas

Anthropic **no publica** lens en su organización de HuggingFace. El repo de código `jlens` es solo la implementación; las lens pre-calculadas (las matrices $J_l$ por capa) las publica la comunidad en dos repos de HuggingFace. Esto tiene una ventaja: no necesitas hacer el *fit* tú (que tarda horas) si alguien ya lo hizo para tu modelo.

### 4.1. Los dos repos de lens

| REPO HF | CARACTERÍSTICA | TAMAÑO TOTAL | CUÁNDO USARLO |
|---|---|---|---|
| `neuronpedia/jacobian-lens` | Lens en bf16, la colección más completa | 56.8 GB | Modelo en bf16 o fp16 cargado en GPU |
| `mcfadyeni/jacobian-lenses` | Lens ajustadas sobre modelos **cuantizados en 4-bit** | sin datos | Modelo cargado en 4-bit (bitsandbytes) |

**Cómo elegir.** Si cargas el modelo en bf16 (lo normal para una GPU de 16 GB), usa `neuronpedia/jacobian-lens`. Si lo cargas en 4-bit (para una GPU más pequeña), usa `mcfadyeni/jacobian-lenses`, porque la lens tiene que estar ajustada sobre el modelo en el mismo formato en que lo vas a usar: una lens ajustada en bf16 no encaja bien con un modelo en 4-bit, porque las activaciones tienen escalas distintas.

### 4.2. Modelos disponibles en `neuronpedia/jacobian-lens`

La colección de neuronpedia cubre las familias principales de modelos abiertos:

| FAMILIA | MODELOS CON LENS PUBLICADA |
|---|---|
| **Gemma** | gemma-2-2b/9b/27b, gemma-3-1b/4b/12b/27b, **gemma-4-e2b/e4b/31b** |
| **Qwen** | qwen2.5-7b-it, qwen3-1.7b/4b/8b/14b/32b, qwen3.5-0.8b/2b/4b/9b/27b |
| **Llama** | llama3.1-8b (+ it), llama3.3-70b-it |
| **OLMo** | olmo-3-1025-7b, olmo-3-1125-32b |
| **Otros** | gpt-oss-20b, gpt2-small, pythia-70m-deduped |

Si tu modelo no está en la lista, tienes dos opciones: (1) ajustar la lens tú mismo con `jlens.fit` (sección 7), o (2) usar la lens de un modelo similar como aproximación (con peor calidad).

### 4.3. Estructura del repo

La estructura dentro de `neuronpedia/jacobian-lens` es:

```
neuronpedia/jacobian-lens/
├── gpt2-small/jlens/Salesforce-wikitext/gpt2_jacobian_lens.pt   (13 MB)
├── gemma-4-e2b/jlens/Salesforce-wikitext/gemma-4-E2B_jacobian_lens.pt  (160 MB)
├── gemma-4-e4b/jlens/Salesforce-wikitext/gemma-4-E4B_jacobian_lens.pt  (537 MB)
├── gemma-4-31b/jlens/Salesforce-wikitext/gemma-4-31B_jacobian_lens.pt  (3.4 GB)
├── qwen3-*/jlens/...
├── llama3.1-*/jlens/...
└── ...
```

Cada modelo tiene su propia carpeta, y dentro de ella un subdirectorio por corpus de ajuste (de momento todos usan `Salesforce-wikitext`). El archivo `.pt` es un checkpoint de PyTorch con las Jacobianas $J_l$ por capa.

### 4.4. El `config.yaml`: qué parámetros usó el ajuste

Cada carpeta incluye además un `config.yaml` con los parámetros exactos del ajuste. Esto es útil para saber en qué condiciones se calculó la lens y si puedes fiarte de ella. Por ejemplo, el de `gemma-4-31b`:

```yaml
np_model_id: "gemma-4-31b"
hf_model_name: "google/gemma-4-31B"
dataset:
  name: "Salesforce/wikitext"
  config: "wikitext-103-raw-v1"
fit:
  n_prompts: 1000
  dim_batch: 64
  max_seq_len: 128
  dtype: "bfloat16"
results:
  prompts_fitted: 861
  final_mean_rel_change: 0.00112414
```

**Qué mirar en el `config.yaml`:**

- `n_prompts`: cuántos prompts se usaron. Más prompts suelen estabilizar la estimación; por debajo de unas decenas conviene desconfiar y validar con más cuidado.
- `max_seq_len`: longitud máxima de los prompts. 128 es estándar; si tu prompt es más largo, la lens sigue funcionando pero puede ser menos precisa en posiciones lejanas.
- `dtype`: el tipo de dato del ajuste. Debe coincidir con el tipo en que cargas el modelo (bf16 o fp16).
- `final_mean_rel_change`: si es menor que 0.01, la lens convergió bien. Si es mayor, el ajuste no terminó de estabilizarse.

### 4.5. Cómo cargar una lens en Python

La lens se carga con `JacobianLens.from_pretrained`, que descarga el archivo del Hub automáticamente y lo cachea en `~/.cache/huggingface/hub`:

```
import jlens

lens = jlens.JacobianLens.from_pretrained(
    "neuronpedia/jacobian-lens",
    filename="gemma-4-e2b/jlens/Salesforce-wikitext/gemma-4-E2B_jacobian_lens.pt",
)
print(repr(lens))
# JacobianLens(d_model=1536, n_prompts=455, source_layers=[0..33] (34 layers))
```

El primer argumento es el `repo_id` del repo de HuggingFace, y `filename` es la ruta al archivo `.pt` dentro del repo. La descarga ocurre solo la primera vez; las siguientes cargas usan la caché.

> **Convención de nombres.** Ojo: el archivo de gpt2-small se llama `gpt2_jacobian_lens.pt` (sin el `-small`), mientras que el de Gemma 4 sigue el patrón `gemma-4-E2B_jacobian_lens.pt`. Si falla la carga con `FileNotFoundError`, revisa el nombre exacto en el árbol del repo con `curl https://huggingface.co/api/models/neuronpedia/jacobian-lens/tree/main/gpt2-small?recursive=true`.

---

## 5. El pipeline completo

Antes de entrar en cada paso, este diagrama muestra el flujo completo de trabajo con `jlens`:

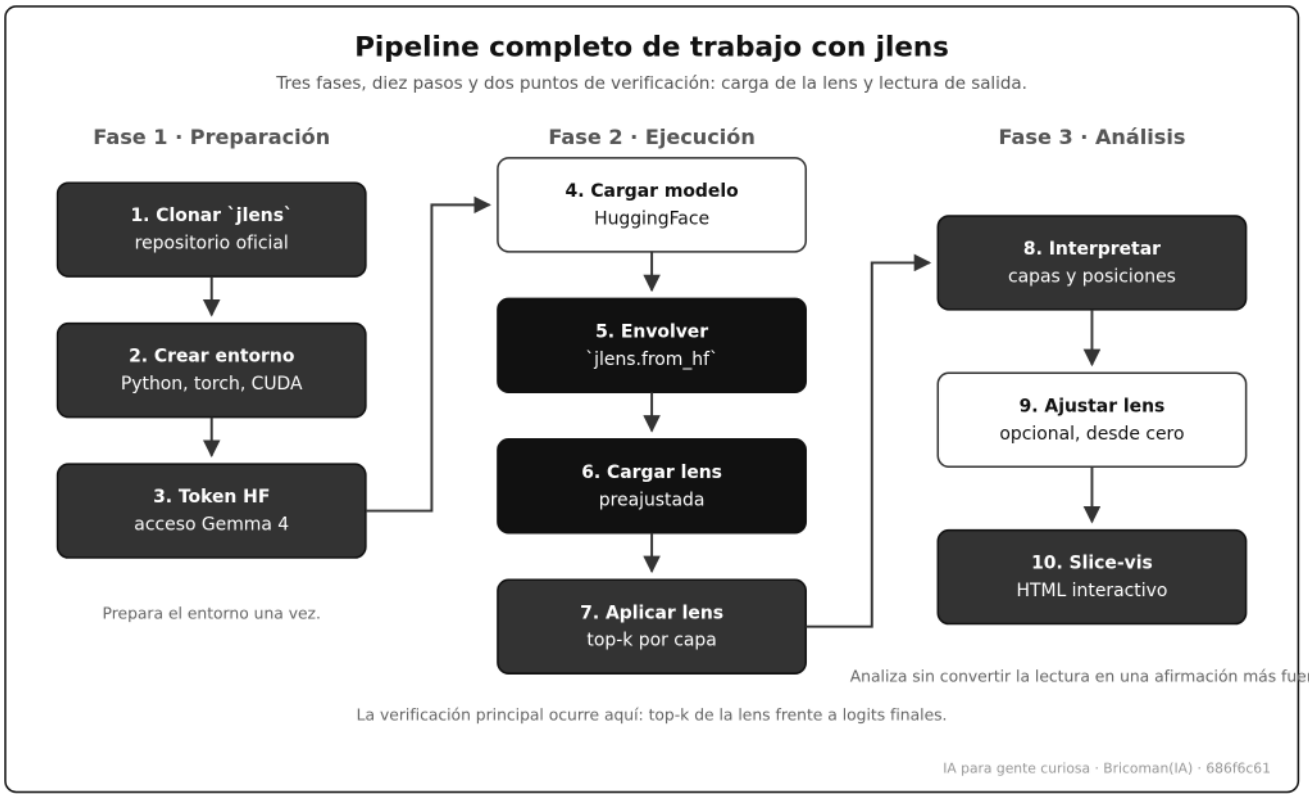



*Figura 3. Pipeline completo de trabajo con `jlens`. Los pasos 1-3 son preparación (clonar, entorno, token), 4-7 son el núcleo (cargar modelo, envolver, cargar lens, aplicar) y 8-10 son análisis e interpretación. Figura propia.*

---

## 6. Hands-on 1: aplicar la lens preajustada a gpt2-small

Empezamos con `gpt2-small` (124 M params, 12 capas, $d = 768$) porque es diminuto, carga en segundos y nos permite validar el pipeline completo antes de meter a Gemma 4. La idea es: si algo falla, que falle en un modelo de 124 millones de parámetros, no en uno de 5 mil millones.

### 6.1. Qué hace el script, paso a paso

El script hace cuatro cosas, en orden:

1. **Cargar el modelo** desde HuggingFace con `AutoModelForCausalLM.from_pretrained`. Esto descarga los pesos (la primera vez) y los coloca en memoria. Para gpt2-small son 124 M de parámetros en float32, unos 500 MB.

2. **Envolver el modelo con `jlens.from_hf`**. Esta función no copia el modelo: registra *hooks* en cada capa residual para poder capturar las activaciones intermedias, y localiza el unembedding. El resultado es un `HFLensModel` que expone `n_layers`, `d_model`, `layers` (la

lista de bloques), y los métodos `encode`, `forward` y `unembed`.

3. **Cargar la lens preajustada** con `JacobianLens.from_pretrained`. Esto descarga el archivo `.pt` del Hub (13 MB para gpt2-small) y carga las Jacobianas $J_l$ pre-calculadas para cada capa. La lens es un objeto que contiene un diccionario `jacobians: dict[int, Tensor]` con una matriz $d \times d$ por capa.

4. **Aplicar la lens** con `lens.apply(model, prompt, positions=[-2])`. Esto ejecuta un *forward* del modelo, captura las activaciones en cada capa, las transporta con $J_l$ y las proyecta con el unembedding. Devuelve `lens_logits` (logits por capa), `model_logits` (logits finales reales) e `input_ids` (tokens del prompt).

### 6.2. El script

```
import transformers, torch, jlens

MODEL_ID = "openai-community/gpt2"
LENS_REPO = "neuronpedia/jacobian-lens"
LENS_FILE = "gpt2-small/jlens/Salesforce-wikitext/gpt2_jacobian_lens.pt"
PROMPT = "Fact: The currency used in the country shaped like a boot is"

# 1. Cargar modelo HuggingFace
hf = transformers.AutoModelForCausalLM.from_pretrained(MODEL_ID, torch_dtype=torch.float32)
tok = transformers.AutoTokenizer.from_pretrained(MODEL_ID)

# 2. Envolver con jlens.from_hf
model = jlens.from_hf(hf, tok)

# 3. Cargar la lens preajustada (se descarga sola del Hub)
lens = jlens.JacobianLens.from_pretrained(LENS_REPO, filename=LENS_FILE)

# 4. Aplicar la lens en la posicion -2 (el token "is")
lens_logits, model_logits, input_ids = lens.apply(model, PROMPT, positions=[-2])

for layer, logits in sorted(lens_logits.items()):
    top5 = logits[0].topk(5)
    tokens = [tok.decode([t]) for t in top5.indices]
    print(f"L{layer:>2}  {tokens}")
```

### 6.3. Salida real

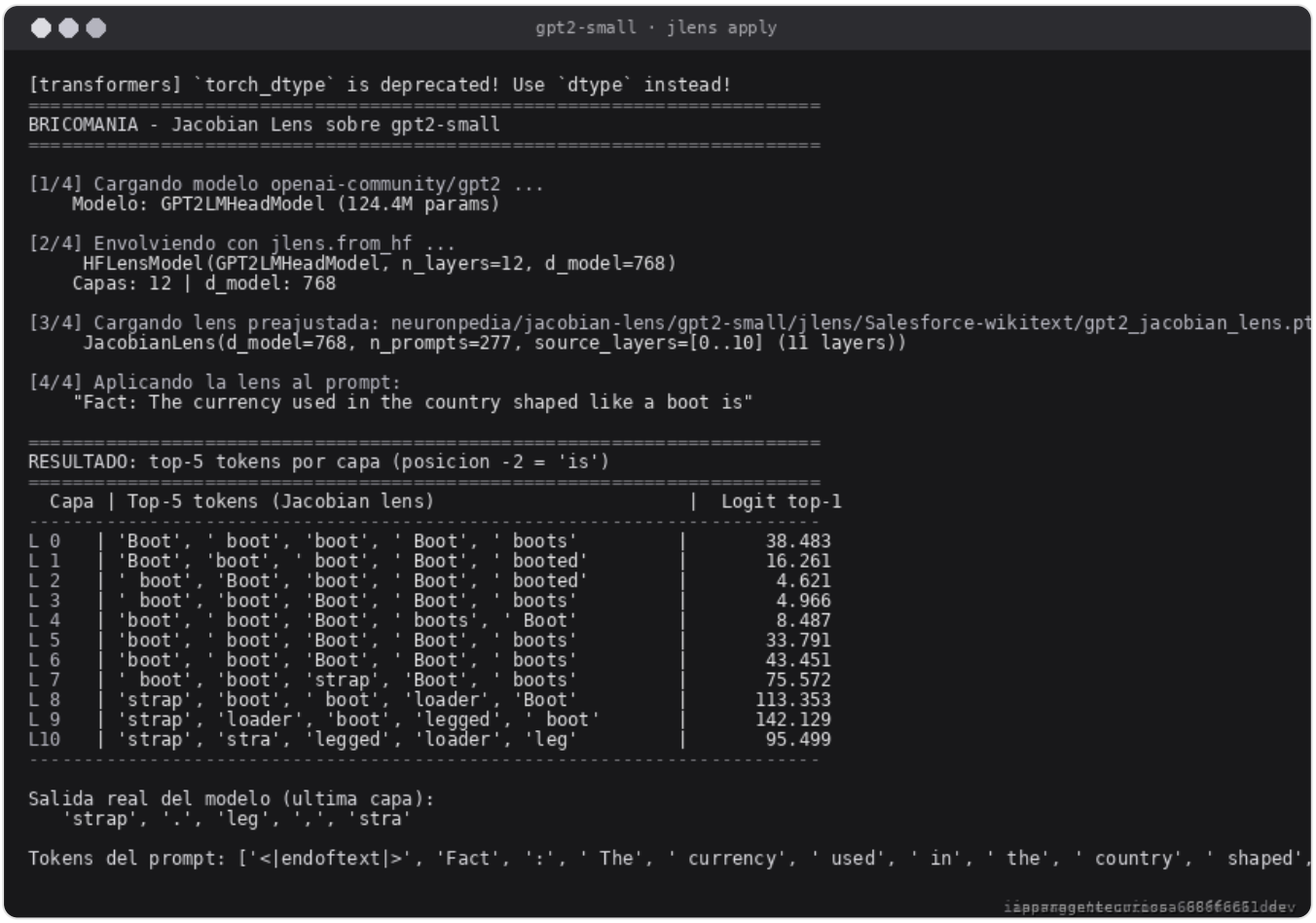



*Figura 4. Salida real verificada al aplicar la Jacobian lens preajustada a gpt2-small. Se aprecia cómo las capas 0-6 leen "boot" (el token anterior) y las capas 8-10 saltan a "strap"/"loader" (completando "bootstrap"), sin llegar nunca a la respuesta factual. Renderizada para publicación desde la salida de la máquina del autor.*

### 6.4. Interpretación: por qué gpt2-small falla

`gpt2-small` es un modelo débil, entrenado en 2020. La lens revela algo instructivo:

- En las capas 0-6, la posición de "is" lee **"boot"**: el modelo aún no ha procesado la pregunta, solo "repite" el token más reciente del contexto. Esto es esperable: las capas tempranas hacen procesamiento local. El residual stream en la posición de "is" aún no ha recibido información de las capas que procesan el significado de la frase.
- En las capas 7-10, salta a **"strap"**, "loader", "legged": está completando la palabra compuesta "bootstrap"/"bootlegged", no respondiendo a la pregunta factual. El modelo ha pasado de procesamiento local a asociación léxica, pero no a recuperación de conocimiento. La atención está mirando "boot" y asociándolo con "bootstrap", que es la continuación más frecuente de "boot" en su corpus de entrenamiento.

- En esta reproducción, la salida final ( `strap` ) y las capas intermedias no hacen aparecer `euro` / `euros` como candidato relevante. La lectura no prueba ausencia absoluta de conocimiento, pero sí que este prompt no recupera esa respuesta en la lens observada.

**Por qué esto importa:** en esta lectura de `gpt2-small` , `euro` / `euros` no aparece como candidato relevante. Las capas se quedan en procesamiento local y asociaciones léxicas ( `boot` → `strap` / `loader` ), no en la recuperación del hecho factual. Es una conclusión acotada al prompt, la posición y la lens usada.

---

## 7. Hands-on 2: ajustar (fit) una lens desde cero

Para entender qué hay dentro de la lens, la ajustamos nosotros sobre gpt2-small con un corpus diminuto (20 prompts). El paper usa 1000; ~100 ya es utilizable. La pregunta que responderemos: ¿se puede ajustar una lens útil con muy pocos datos, o hace falta el corpus completo?

### 7.1. Qué hace el fit, paso a paso

El *fit* calcula, para cada capa $l$, la Jacobiana media $J_l = \mathbb{E}[\partial h^{(L)} / \partial h^{(l)}]$ sobre un corpus de prompts. El proceso, para cada prompt:

1. **Forward**: se ejecuta el modelo sobre el prompt, construyendo el grafo de autograd desde cada capa hasta la salida.
2. **Backward por dimensión**: para cada dimensión de salida $i$ (de las $d$ totales), se inyecta una cotangente *one-hot* en todas las posiciones destino válidas y se hace *backward*. El gradiente que llega a la capa $l$ es una fila de la Jacobiana.
3. **Promedio**: se acumula el gradiente en un sumador y se promedia sobre todos los prompts y posiciones.

El coste es **1 forward +** $\lceil d/B \rceil$ **backwards** por prompt, donde $B$ es `dim_batch` (cuántas dimensiones se procesan a la vez). Para gpt2-small ($d = 768$, $B = 8$), son 96 backwards por prompt. Con 20 prompts, son 1920 backwards en total, que tardan ~115 segundos.

### 7.2. El script

```
import transformers, torch, jlens, time

hf = transformers.AutoModelForCausalLM.from_pretrained("openai-community/gpt2",
torch_dtype=torch.float32)
tok = transformers.AutoTokenizer.from_pretrained("openai-community/gpt2")
model = jlens.from_hf(hf, tok)

prompts = [
    "The Eiffel Tower is a wrought-iron lattice tower on the Champ de Mars in Paris,
France.",
    "In economics, inflation is a general increase in the prices of goods and services...",
    "The Pythagorean theorem states that in a right triangle, the square of the
hypotenuse...",
    # ... 20 prompts en total (estilo wikitext)
]

t0 = time.perf_counter()
lens = jlens.fit(
    model,
    prompts=prompts,
    checkpoint_path="runs/my_gpt2_lens_ckpt.pt",
    max_seq_len=128,
    dim_batch=8,
)
print(f"Fit en {time.perf_counter()-t0:.1f}s")
lens.save("runs/my_gpt2_lens.pt")
```

## 7.3. Salida real

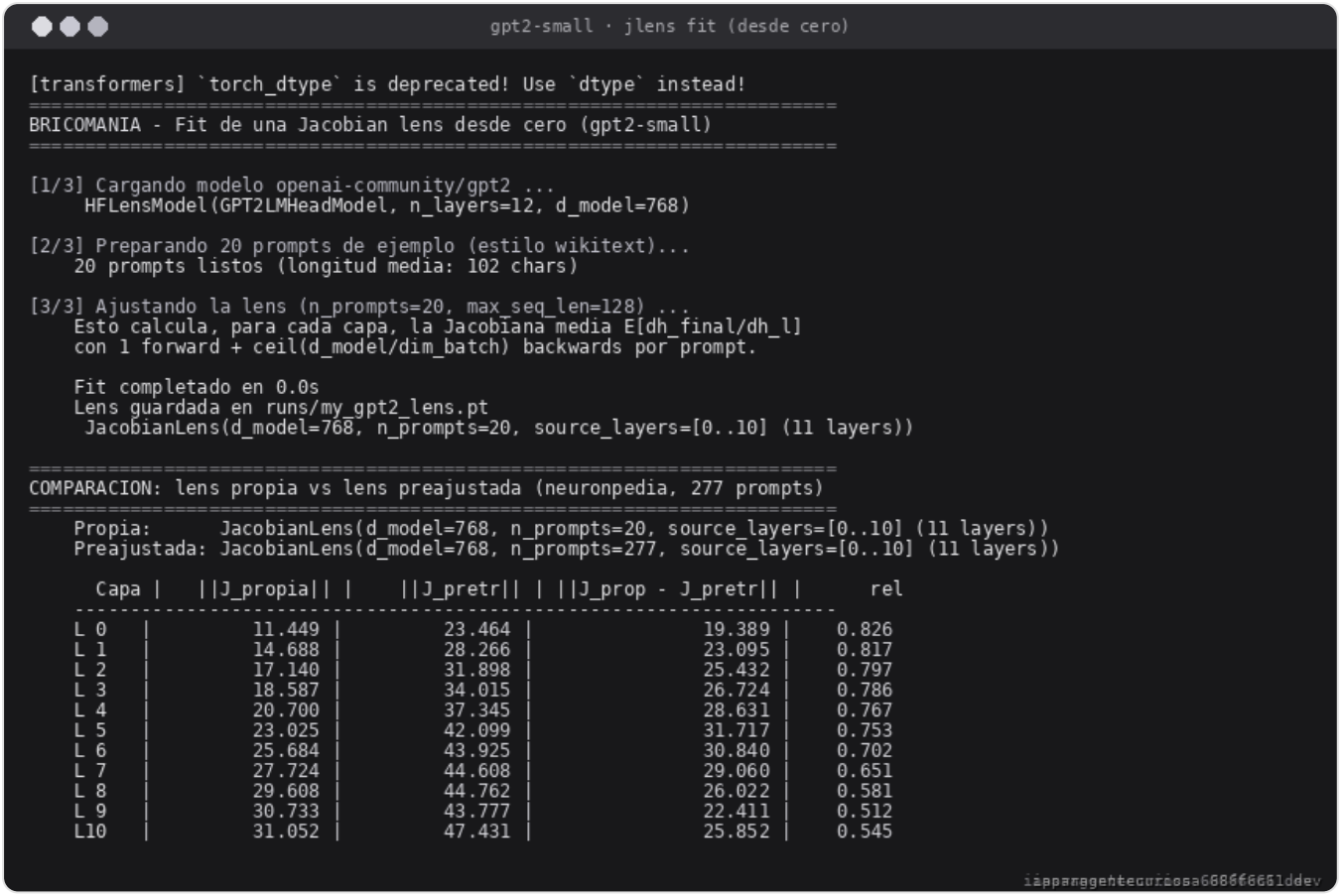



*Figura 5. Salida real verificada del ajuste de una Jacobian lens desde cero sobre gpt2-small (20 prompts, 115.6 s). Se aprecia la comparación final entre la lens propia y la preajustada de neuronpedia: la diferencia relativa cae del 83 % en L0 al 51 % en L9. Renderizada para publicación desde la salida de la máquina del autor.*

## 7.4. Comparación con la lens preajustada

Comparamos la norma de Frobenius $\|J_l\|_F$ de nuestra lens (20 prompts) con la de neuronpedia (277 prompts), y la diferencia relativa $\|J_l^{\text{propia}} - J_l^{\text{pre}}\|_F / \|J_l^{\text{pre}}\|_F$:

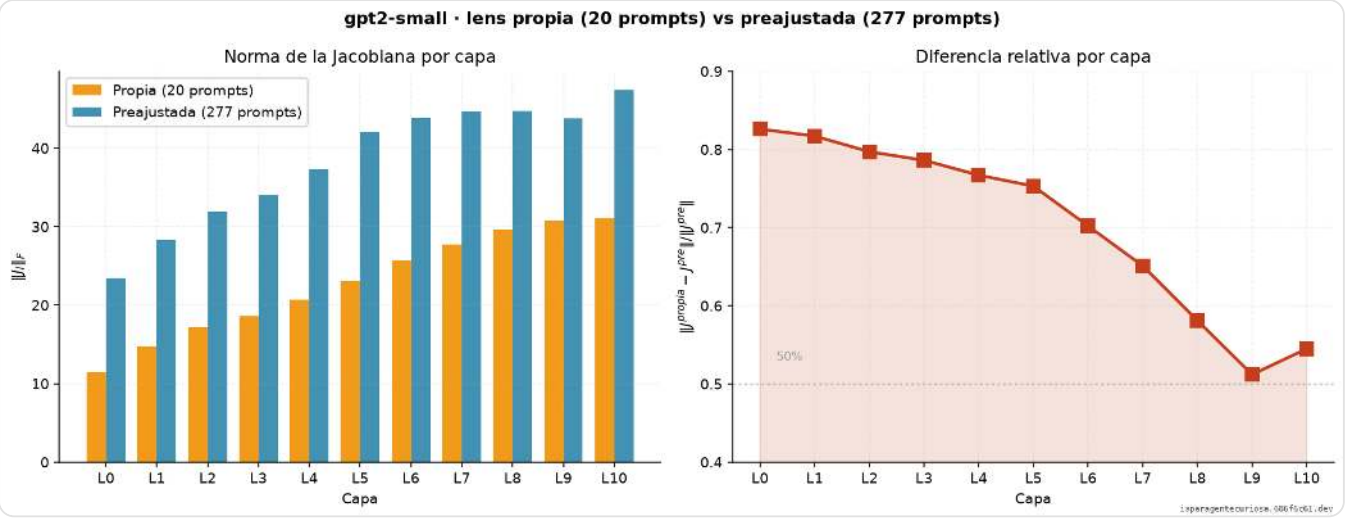



*Figura 6. Izquierda: norma $\|J_l\|_F$ por capa para la lens propia (20 prompts, naranja) y la preajustada de neuronpedia (277 prompts, azul). Derecha: diferencia relativa $\|J^{propia} - J^{pre}\|/\|J^{pre}\|$, que cae del 83 % en L0 al 51 % en L9. Las capas finales convergen antes; las iniciales necesitan más prompts. Figura propia con datos reales del fit.*

**Lectura:** con solo 20 prompts, la lens difiere entre un 51 % (capas finales) y un 83 % (capas iniciales) respecto a la de 277 prompts. La diferencia decrece con la profundidad de la capa: las capas finales son más estables (menos derivan el espacio residual), las iniciales necesitan más prompts para converger.

**Por qué las capas finales convergen antes:** las capas finales están más cerca de la salida, así que su Jacobiana $J_l$ es más parecida a la identidad (menos transformaciones intermedias que promediar). Las capas iniciales tienen que "resumir" el efecto de muchas capas posteriores, y eso requiere más datos para estimarlo bien. Esto cuadra con la sección 9.3 del paper, donde la calidad satura a partir de ~100 prompts.

La convergencia del ajuste se puede ver con más detalle usando el CSV de convergencia que publica neuronpedia para gpt2-small (277 prompts):

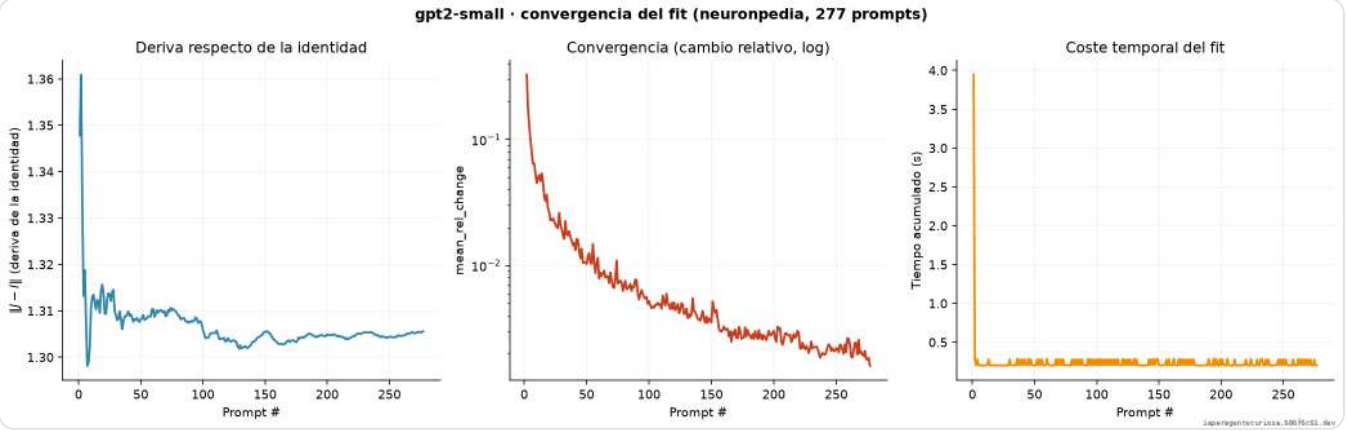



*Figura 7. Convergencia del fit de la Jacobian lens en gpt2-small (datos públicos de neuronpedia, 277 prompts). Izquierda: $\|J - I\|$ (deriva respecto de la identidad) crece con el número de prompts, la lens se aleja de la identidad a medida que captura la transformación*

*real del modelo. Centro: el cambio relativo de la media cae en escala logarítmica, señal de convergencia. Derecha: el coste temporal crece lineal con los prompts. Figura propia a partir del CSV público.*

**Por qué la deriva crece:** al principio, con pocos prompts, la Jacobiana media es casi la identidad porque no hay datos para estimar la transformación. A medida que se promedian más prompts, $J_l$ captura la transformación real del modelo y se aleja de $I$. El crecimiento se estabiliza cuando la media converge.

---

## 8. La visualización *slice-vis*

La lens se puede renderizar como una rejilla interactiva **posición × capa** donde cada celda muestra el token top-1 que la lens lee en esa posición y capa. El HTML es autocontenido (d3 embebido).

### 8.1. Generar la slice-vis

```
import transformers, jlens
from jlens.vis import compute_slice, build_page
from jlens.examples import EXAMPLES, resolve_prompt

hf = transformers.AutoModelForCausalLM.from_pretrained("openai-community/gpt2",
torch_dtype=torch.float32)
tok = transformers.AutoTokenizer.from_pretrained("openai-community/gpt2")
model = jlens.from_hf(hf, tok)
lens = jlens.JacobianLens.from_pretrained(
    "neuronpedia/jacobian-lens",
    filename="gpt2-small/jlens/Salesforce-wikitext/gpt2_jacobian_lens.pt",
)

# El ejemplo "ascii-face" del paper
ejemplo = [e for e in EXAMPLES if e.slug == "ascii-face"][0]
prompt = resolve_prompt(ejemplo, tok)

slice_data = compute_slice(model, lens, prompt, top_n=10, max_seq_len=512)
html, raw, payload = build_page(
    slice_data, prompt,
    title="gpt2-small | ASCII face",
    description=ejemplo.description,
    mode="embed",
)
with open("runs/slice_vis/ascii-face.html", "w") as f:
    f.write(html)
```

## 8.2. El ejemplo de la nariz ASCII: origen e importancia

El ejemplo `ascii-face` **no es nuestro**: viene incluido en el repo de Anthropic, en `jlens/examples.py` (Anthropic, 2026). Lo mantenemos aquí porque es el ejemplo canónico del paper y permite explicar por qué la lens es importante. En la sección 10 creamos ejemplos propios.

El prompt es una cara dibujada con caracteres ASCII, con un `^` que hace de nariz:

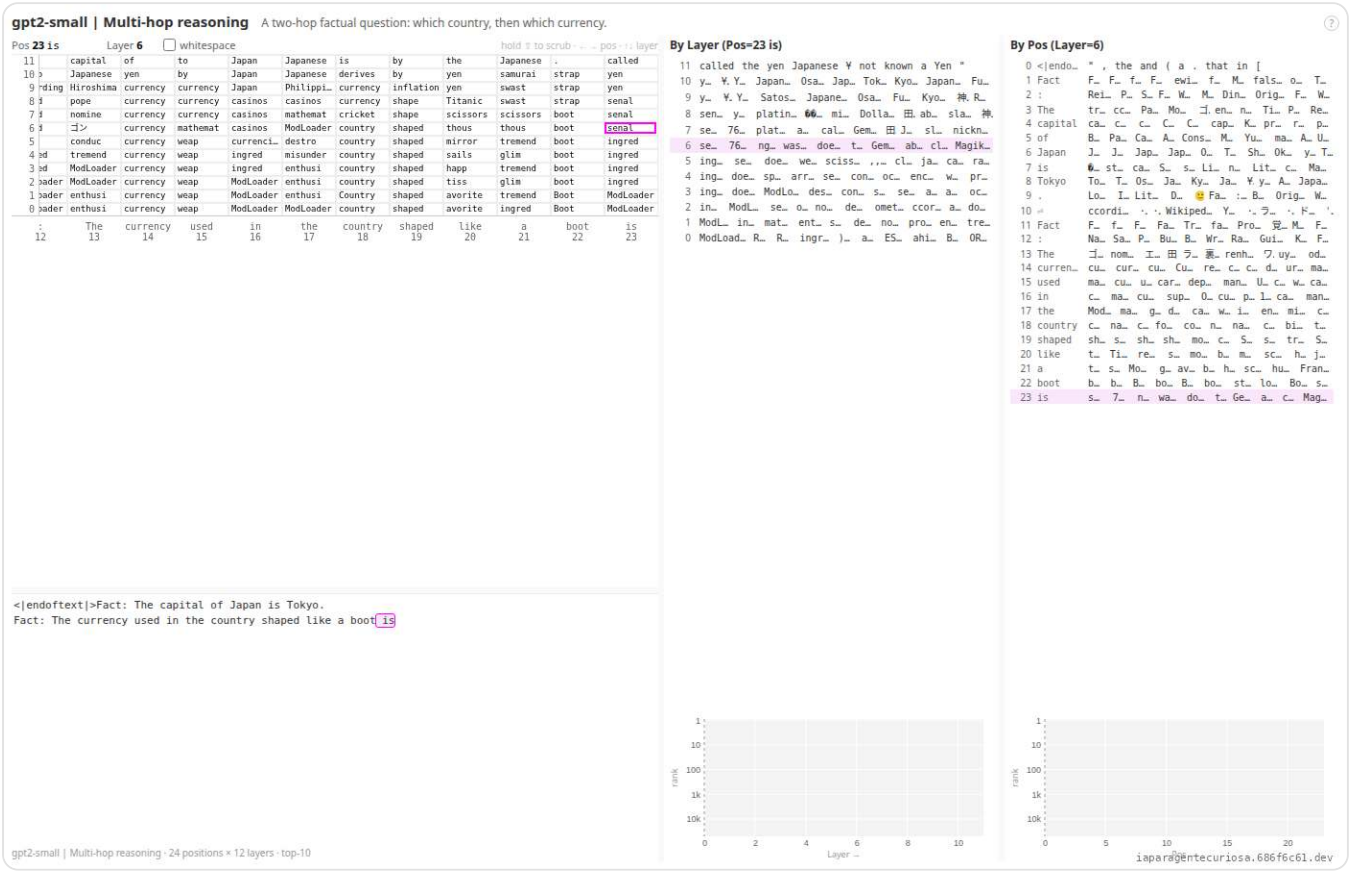
```
    _______
   /       \
  /  ~     ~  \
 (   o     o   )
 |      ^      |       <- la nariz
 |             |
 |   \_____/   |
  \           /
   \_________/
      |   |
What is this?
```

**Por qué este ejemplo es importante:** la nariz `^` no es una palabra, es un símbolo visual. Si la lens, aplicada en la posición del `^`, lee "nose" en alguna capa intermedia, eso significa que el modelo ha formado una representación **verbalizable** de un estímulo no verbal. Es la evidencia más directa del *global workspace*: el modelo convierte una representación visual (el carácter `^` en el contexto de una cara) en una representación lingüística ("nose") que podría verbalizar.

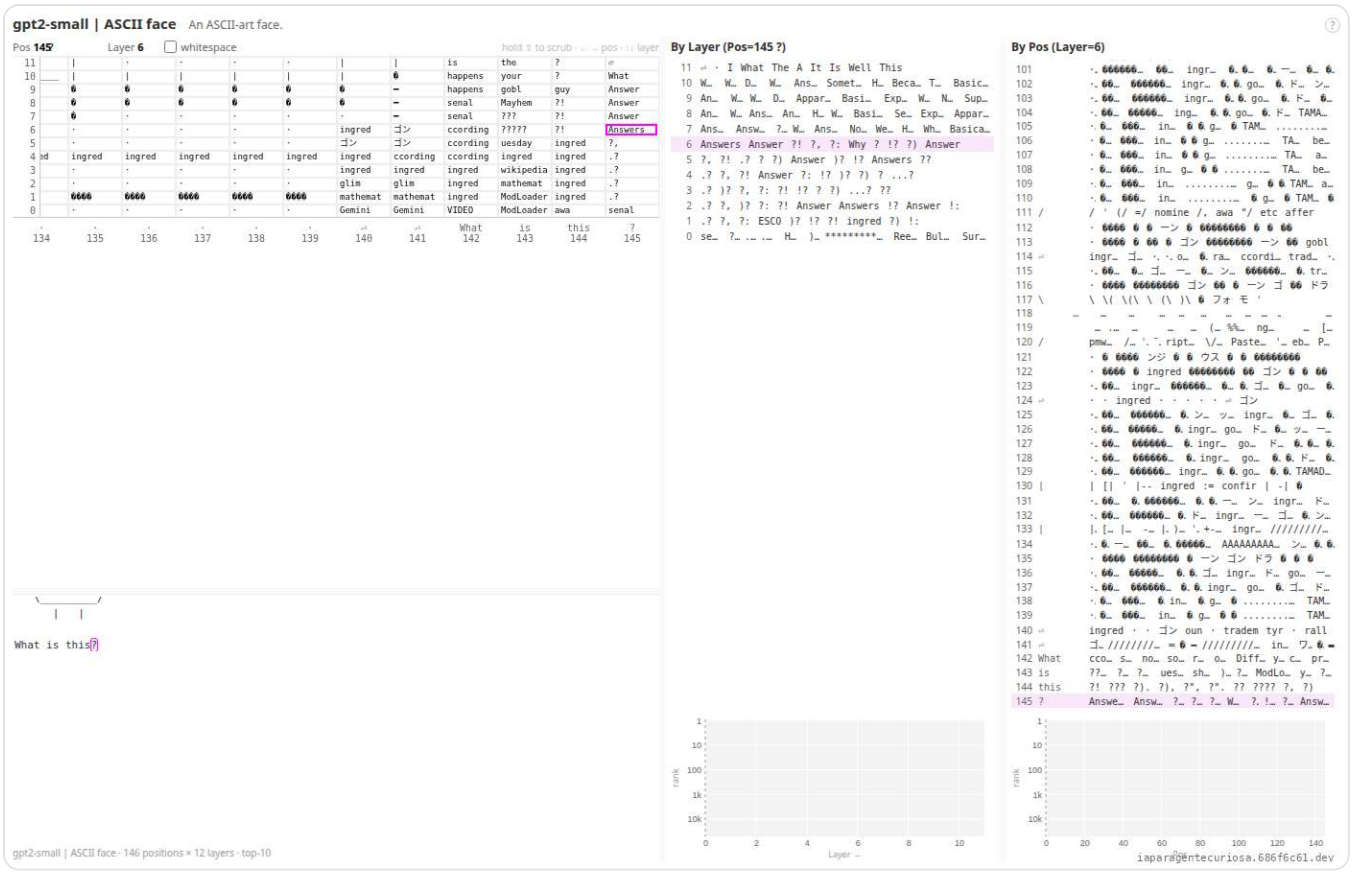
En el paper original (Anthropic, 2026), con Gemma 4 31B, la lens lee "nose" en la posición del `^` en capas medias. En nuestra reproducción con modelos más pequeños, el efecto no aparece con esa nitidez: gpt2-small lee sobre todo espacios y caracteres de control, y Gemma 4 E2B devuelve principalmente marcas estructurales del dibujo (`^`, guiones bajos, espacios y saltos de línea) en la posición de la nariz. Es un resultado negativo útil: la visualización funciona, pero este hardware y este modelo no reproducen el hallazgo canónico de 31B.

Los HTML generados en esta pieza están en `runs/slice_vis/` y se abren en cualquier navegador. Además se han renderizado a PNG con Chrome headless para incluirlos como figuras estáticas:

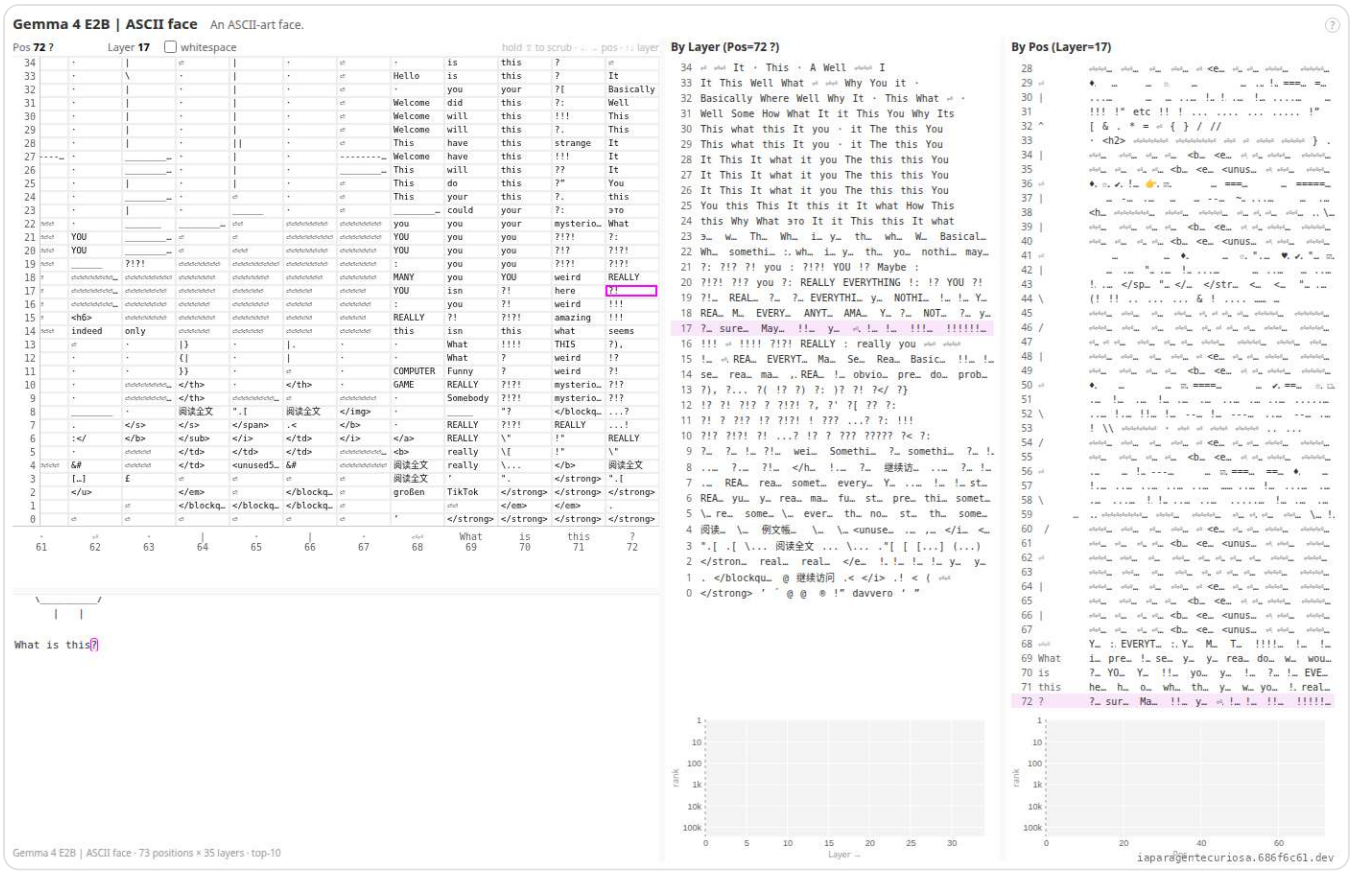
| ARCHIVO | MODELO | TAMAÑO HTML | RENDER PNG |
|---|---|---|---|
| `multihop.html` | gpt2-small | 1.2 MB | `multihop.png` (146 KB) |
| `ascii-face.html` | gpt2-small | 4.7 MB | `ascii-face.png` (155 KB) |
| `gemma4-e2b-ascii-face.html` | Gemma 4 E2B | 18.3 MB | `gemma4-e2b-ascii-face.png` (217 KB) |




*Slice-vis multihop. Render estático (Chrome headless) de la visualización interactiva de gpt2-small para el ejemplo multihop. La rejilla posición × capa muestra el token top-1 que la lens lee en cada celda. La versión HTML es interactiva: al hacer clic en una celda se fija el token y aparecen gráficas de rango. Render propio del HTML generado con* `jlens.vis.build_page`.




*Slice-vis ascii-face en gpt2-small. En esta reproducción pequeña dominan espacios, signos y fragmentos de estructura, no una lectura semántica limpia de "nose". Es precisamente el tipo de pantalla que conviene enseñar a los alumnos: no todas las visualizaciones bonitas son una confirmación fuerte.*




*Slice-vis ascii-face en Gemma 4 E2B. El HTML interactivo pesa más porque incluye más capas y más vocabulario rastreado. La posición del `^` no reproduce el "nose" de Gemma 4 31B; muestra sobre todo tokens estructurales del dibujo, lo que refuerza la lectura prudente de esta reproducción.*

### 8.3. Cómo leer una página *slice-vis*

- Cada celda muestra el token top-1 en esa (posición, capa); el superíndice es su rango sobre el vocabulario completo.
- Al hacer clic en una celda se fija ese token y aparecen gráficas de evolución de su rango y un mapa de calor.
- La fila inferior ($L = n_{\text{layers}} - 1$) es la salida real del modelo (con $J = I$).

---

## 9. Hands-on 3: Gemma 4 E2B, el plato fuerte

`gpt2-small` nos ha servido de juguete. Ahora vamos al modelo que nos interesa: **Gemma 4 E2B** (Google, 2026), un modelo de 5.100 millones de parámetros que cabe en una GPU de 16 GB y que, como veremos, puede hacer aparecer candidatos correctos en capas intermedias aunque la salida final priorice otra continuación. Esta sección es el corazón del artículo: aquí es donde la lens revela su valor real.

## 9.1. La trampa de los modelos multimodales

### 9.1.1. Qué significa "multimodal"

Un modelo **multimodal** es aquel que puede procesar varios tipos de entrada a la vez: texto, imagen, audio. Gemma 4 es trimodal (texto + imagen + audio). Esto no es un detalle menor para nosotros: la lens funciona sobre el **decoder de texto**, no sobre los encoders de imagen o audio. Si intentamos aplicar la lens al modelo entero (incluyendo las partes de audio e imagen), no funciona, porque la lens está diseñada para el residual stream del lenguaje, no para las activaciones de un encoder de audio.

**La analogía de la calle.** Imagina que Gemma 4 es un edificio con tres plantas: la planta baja procesa texto, el primer piso procesa imágenes y el segundo procesa audio. Cada planta tiene su propio equipo y sus propios pasillos. La lens es una herramienta que solo sabe leer los pasillos de la planta de texto. Si intentas usarla en los pasillos de audio o imagen, los números no significan nada. Por eso necesitamos aislar la planta de texto del resto del edificio.

### 9.1.2. `Gemma4ForConditionalGeneration` vs `*ForCausalLM`

En HuggingFace, los modelos de lenguaje siguen una convención de nombres:

- `*ForCausalLM`: modelos que solo generan texto (Llama, Qwen, Mistral). Tienen un decoder de texto puro.
- `*ForConditionalGeneration`: modelos que pueden condicionar su salida a múltiples entradas (texto + imagen + audio). Tienen un *wrapper* multimodal que envuelve al decoder de texto.

Gemma 4 carga como `Gemma4ForConditionalGeneration`, no como `Gemma4ForCausalLM`. La diferencia es estructural: el modelo que descargas de HuggingFace tiene, por dentro, tres submodelos (texto, imagen, audio) envueltos en un contenedor. Si llamamos a `from_pretrained` y obtenemos el contenedor, no podemos aplicar la lens directamente: hay que extraer el decoder de texto de dentro.

El `config.json` lo deja claro. Fíjate en que hay **tres configs anidadas**, una por modalidad:

```
{
  "architectures": ["Gemma4ForConditionalGeneration"],
  "audio_config": { "model_type": "gemma4_audio", "num_hidden_layers": 12, ... },
  "model_type": "gemma4",
  "text_config": {
    "hidden_size": 1536,
    "num_hidden_layers": 35,
    "vocab_size": 262144,
    "final_logit_softcapping": 30.0,
    ...
  }
}
```

El decoder de texto vive en `text_config`: 35 capas, $d = 1536$, vocabulario de 262.144 tokens. El de audio vive en `audio_config`: 12 capas. La lens solo trabaja con el de texto.

### 9.1.3. Cómo `jlens` extrae el decoder de texto: los *layouts*

La buena noticia: `jlens.from_hf` ya prevé esto. En `jlens/hf.py` hay una lista de *layouts* que prueba en orden. Un **layout** es, en este contexto, una receta que le dice a `jlens` dónde encontrar las piezas del modelo que necesita: las capas residuales, la norma final, la matriz de embedding y la matriz de unembedding. Cada familia de modelos organiza estas piezas de forma distinta, por eso hace falta un layout distinto para cada una.

```
_LAYOUTS = (
    Layout("model"),                      # decoder puro (Llama, Qwen, Mistral...)
    Layout("model.language_model"),        # wrapper multimodal
    Layout("language_model"),              # otro wrapper multimodal
    Layout("model", norm="final_layernorm"),  # Phi
    Layout("transformer", layers="h", norm="ln_f", embed="wte"),  # GPT-2
    Layout("gpt_neox", norm="final_layer_norm", embed="embed_in",
           lm_head="embed_out"),           # Pythia
)
```

Para Gemma 4, el layout ganador es `model.language_model`: extrae el decoder de texto del wrapper multimodal. ¿Cómo lo elige? `jlens` prueba cada layout en orden y se queda con el primero que funciona, es decir, el primero donde puede encontrar las capas, la norma y el unembedding sin error. Para un `*ForCausalLM` puro (como Llama), gana el primer layout (`"model"`). Para Gemma 4, que es `*ForConditionalGeneration`, el primer layout falla (porque `model` no tiene capas residuales directamente, las tiene dentro de `model.language_model`), y gana el segundo.

### 9.1.4. Los *hooks*: cómo `jlens` captura activaciones intermedias

Una vez extraído el decoder de texto, `jlens` necesita capturar las activaciones $h_t^{(l)}$ en cada capa intermedia. Para eso usa **hooks** de PyTorch.

Un **hook** es una función que se ejecuta automáticamente cuando PyTorch pasa por una capa. Hay dos tipos:

- **Forward hook**: se ejecuta después del *forward* de una capa. Recibe la entrada y la salida. `jlens` registra un forward hook en cada bloque residual para guardar $h_t^{(l)}$ en una lista.
- **Backward hook**: se ejecuta después del *backward* (durante el cálculo de gradientes). `jlens` lo usa durante el *fit* para inyectar cotangentes y calcular la Jacobiana.

El hook no modifica el modelo ni copia los pesos: solo "escucha" lo que pasa por cada capa y lo anota. Es como poner un micrófono en cada cruce de la avenida: no cambia el tráfico, solo lo graba.

### 9.1.5. `final_logit_softcapping`: por qué Gemma acota sus logits

Gemma 4 tiene un detalle técnico importante: **softcapping**. El `config.json` incluye `"final_logit_softcapping": 30.0`. Esto significa que, tras el unembedding, Gemma aplica una transformación no lineal a los logits para mantenerlos acotados:

$$\text{logits} = c \cdot \tanh\left(\frac{\text{logits}}{c}\right), \qquad c = 30.0.$$

**¿Qué es un logit?** Un logit es un número real (positivo o negativo) que un modelo asigna a cada token del vocabulario antes de convertirlo en probabilidad. El token con el logit más alto es el que el modelo "prefiere". La conversión a probabilidad se hace con *softmax*: $p_i = e^{z_i} / \sum_j e^{z_j}$, donde $z_i$ es el logit del token $i$. Los logits pueden ser cualquier número real, sin límite teórico.

**¿Por qué acotarlos?** Sin softcapping, los logits pueden crecer indefinidamente. Un logit de 50 hace que su token tenga probabilidad efectivamente 1.0 (porque $e^{50}$ es enormemente mayor que cualquier otro $e^{z_j}$), lo que hace que el modelo sea "demasiado seguro" y pierda capacidad de exploración. El softcapping con $\tanh$ comprime los logits al rango $[-30, +30]$: por muy grande que sea el logit crudo, $\tanh$ lo aplasta hacia 30 sin llegar nunca a 30. Así, la distribución de probabilidad nunca colapsa del todo en un solo token.

**La analogía de la calle.** Imagina que el logit es la "puntuación" que el modelo le da a cada token. Sin softcapping, un token puede tener 1000 puntos y los demás 0: el modelo dice "seguro que es este". Con softcapping, la puntuación máxima es 30: el modelo puede preferir un token, pero nunca con seguridad absoluta. Es como un examen donde la nota máxima es 30, no infinito: por bueno que seas, no puedes sacar más de 30.

`jlens` detecta `final_logit_softcapping` y aplica la misma transformación a los logits de la lens. Sin este detalle, los logits de la lens (que se calculan transportando $h^{(l)}$ y aplicando el unembedding) no serían comparables con los del modelo real, porque el modelo aplica el softcapping al final y la lens, si no lo aplica, produciría logits en otra escala. Por eso verás que en las tablas de salida los logits están siempre cerca de 30: es el techo del softcapping.

### 9.2. Cargar el modelo

```
import transformers, torch, jlens, os

os.environ["HF_TOKEN"] = "hf_xxx_tu_token"
hf = transformers.AutoModelForCausalLM.from_pretrained(
    "google/gemma-4-E2B",
    torch_dtype=torch.bfloat16,
    device_map="cuda",
    token=os.environ["HF_TOKEN"],
)
tok = transformers.AutoTokenizer.from_pretrained("google/gemma-4-E2B",
token=os.environ["HF_TOKEN"])
model = jlens.from_hf(hf, tok)
lens = jlens.JacobianLens.from_pretrained(
    "neuronpedia/jacobian-lens",
    filename="gemma-4-e2b/jlens/Salesforce-wikitext/gemma-4-E2B_jacobian_lens.pt",
)
```

#### 9.2.1. `torch.bfloat16`: qué es y por qué lo usamos

`torch_dtype=torch.bfloat16` le dice a PyTorch que cargue los pesos en **bfloat16** (bf16), un formato de número de 16 bits. Los dos formatos de 16 bits habituales son:

| FORMATO | BITS DE SIGNO | BITS DE EXPONENTE | BITS DE MANTISA | RANGO DINÁMICO | PRECISIÓN |
|---|---|---|---|---|---|
| float32 (fp32) | 1 | 8 | 23 | enorme | alta |
| float16 (fp16) | 1 | 5 | 10 | limitado | media |
| bfloat16 (bf16) | 1 | 8 | 7 | enorme (igual que fp32) | baja |

La clave de bf16 es que tiene el mismo exponente que fp32 (8 bits), por lo que puede representar números igual de grandes o pequeños, pero con menos mantisa (7 bits en vez de 23), por lo que pierde precisión. Para redes neuronales, esto es ideal: los pesos y activaciones no necesitan muchos decimales, pero sí necesitan no desbordarse (overflow). fp16, con solo 5 bits de exponente, se desborda con facilidad y requiere *gradient scaling*; bf16 no.

**La analogía de la calle.** Imagina que cada número es una cesta con frutas. fp32 es una cesta enorme con 23 frutas, muy precisa. fp16 es una cesta pequeña con 10 frutas, pero que se rompe si pones más de 65.000 frutas (overflow). bf16 es una cesta pequeña con 7 frutas, pero

que aguanta igual de peso que la cesta enorme. Para los pesos de un modelo, 7 frutas bastan: no necesitas precisión decimal, necesitas que no se rompa.

**Por qué bf16 y no fp32.** Gemma 4 E2B tiene 5.100 millones de parámetros. En fp32, cada parámetro ocupa 4 bytes, total 20.4 GB. En bf16, cada parámetro ocupa 2 bytes, total 10.2 GB. Una RTX 5060 Ti tiene 16 GB de VRAM: en fp32 no cabe, en bf16 sí. La pérdida de precisión de bf16 no afecta a la calidad de la lens: las Jacobianas se calculan en bf16 y los resultados son coherentes.

### 9.2.2. `device_map="cuda"`: qué hace

`device_map="cuda"` le dice a HuggingFace que coloque el modelo entero en la GPU (CUDA). La alternativa es `device_map="auto"`, que reparte el modelo entre GPU y CPU según la memoria disponible. Para un modelo que cabe entero en la GPU (como E2B en bf16), `"cuda"` es más eficiente: todo está en la GPU y no hay transferencias CPU-GPU en cada *forward*. Para un modelo que no cabe (como E4B multimodal en bf16), habría que usar `"auto"` o cargar solo la parte de texto.

### 9.2.3. Salida real

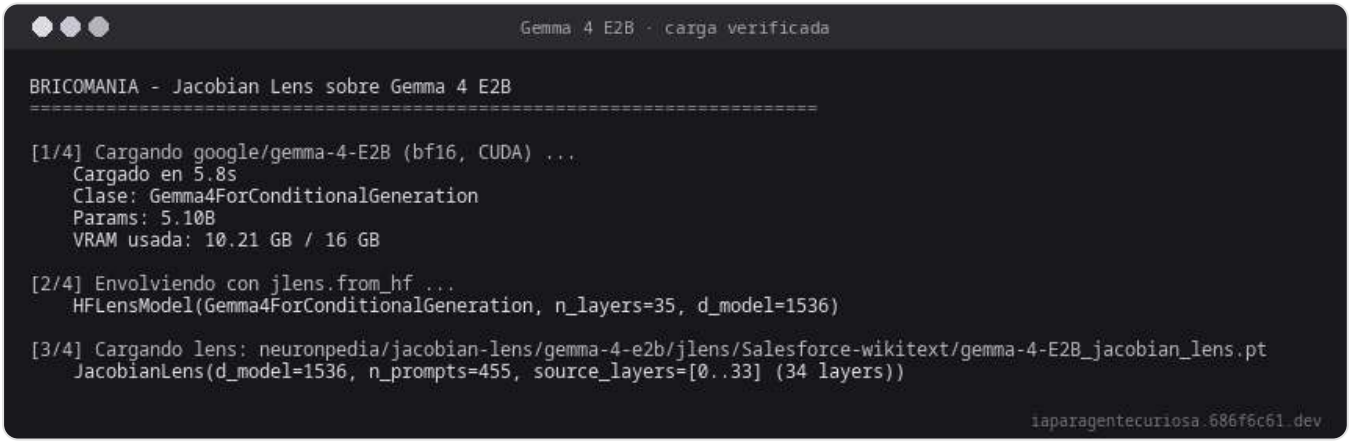



*Figura 8. Salida real verificada de la carga de Gemma 4 E2B (bf16, CUDA) y la lens preajustada de neuronpedia. El modelo ocupa 10.21 GB de VRAM y la lens queda cargada con 455 prompts de ajuste. Renderizada para publicación desde la salida de la máquina del autor.*

10.2 GB de VRAM: cabe con holgura en 16 GB. La segunda carga (caché caliente) baja a 6.0 s.

### 9.2.4. Qué significa `n_prompts=455` y `source_layers=[0..33]`

La línea `JacobianLens(d_model=1536, n_prompts=455, source_layers=[0..33] (34 layers))` merece explicación:

- **`d_model=1536`**: la dimensión del residual stream. Cada vector $h_t^{(l)}$ tiene 1536 componentes.
- **`n_prompts=455`**: la lens se ajustó con 455 prompts de wikitext. Es una muestra razonable para una lens preajustada, aunque no garantiza calidad por sí sola. Cada prompt contribuye con sus posiciones válidas al promedio de la Jacobiana.

- `source_layers=[0..33] (34 layers)`: la lens tiene Jacobianas para 34 capas, de la 0 a la 33. Pero el modelo tiene 35 capas (0 a 34). ¿Por qué 34 y no 35? Porque la última capa (L34) no necesita Jacobiana: es la capa final, donde $J_L = I$ (la identidad), ya que $h^{(L)}$ se transporta a sí misma. La lens no almacena la identidad porque no hace falta: aplicar la lens en la última capa es simplemente aplicar el unembedding directo, que es lo que hace el modelo real.

En las tablas de esta sección, por tanto, **L0-L33 son las capas fuente que trae la lens preajustada**. La salida final del modelo aparece como una fila separada y corresponde al readout final del decoder de texto (L34, más norma final, unembedding y softcapping).

### 9.3. Posiciones negativas: qué son y por qué importan

Antes de aplicar la lens, hay que entender qué posiciones miramos. En `jlens`, las **posiciones** se pueden dar como índices absolutos (0, 1, 2, ...) o como índices negativos (-1, -2, ...), como en Python:

- **Posición -1**: el último token del prompt. Es donde el modelo va a predecir el siguiente token. Si el prompt es *"Fact: The currency used in the country shaped like a boot is"*, la posición -1 es "is". La lens en la posición -1 nos dice qué candidatos quedan legibles justo antes de generar el siguiente token.
- **Posición -2**: el penúltimo token. En nuestro prompt, es "boot". La lens en la posición -2 nos dice qué candidatos quedan legibles en la posición de "boot", que ya ha sido procesada pero que el modelo podría seguir usando vía atención.

**Por qué miramos -2 y -1.** La posición -1 es la más interesante porque es donde el modelo predice; la -2 sirve como control. Si el token correcto apareciera en -2 y desapareciera en -1, el problema sería de mantenimiento de la representación. Si solo aparece en -1, el hallazgo está localizado en el momento de generación.

**La analogía de la calle.** Imagina que la avenida tiene 14 cruces (los 14 tokens del prompt). La posición -1 es el último cruce, donde decides qué decir. La posición -2 es el penúltimo, donde ya has oído toda la pregunta pero aún no te toca responder. Mirar las dos es como preguntar a alguien "¿qué sabías cuando oíste la pregunta?" (-2) y "¿qué sabías cuando te tocó responder?" (-1). Si sabías la respuesta en -2 pero la olvidaste en -1, el problema es de memoria a corto plazo. Si no la sabías en ninguna, el problema es de conocimiento.

### 9.4. Lectura en la posición -2 (el token "boot")

Aplicamos la lens en la posición -2 (el token "boot") del prompt *"Fact: The currency used in the country shaped like a boot is"*:

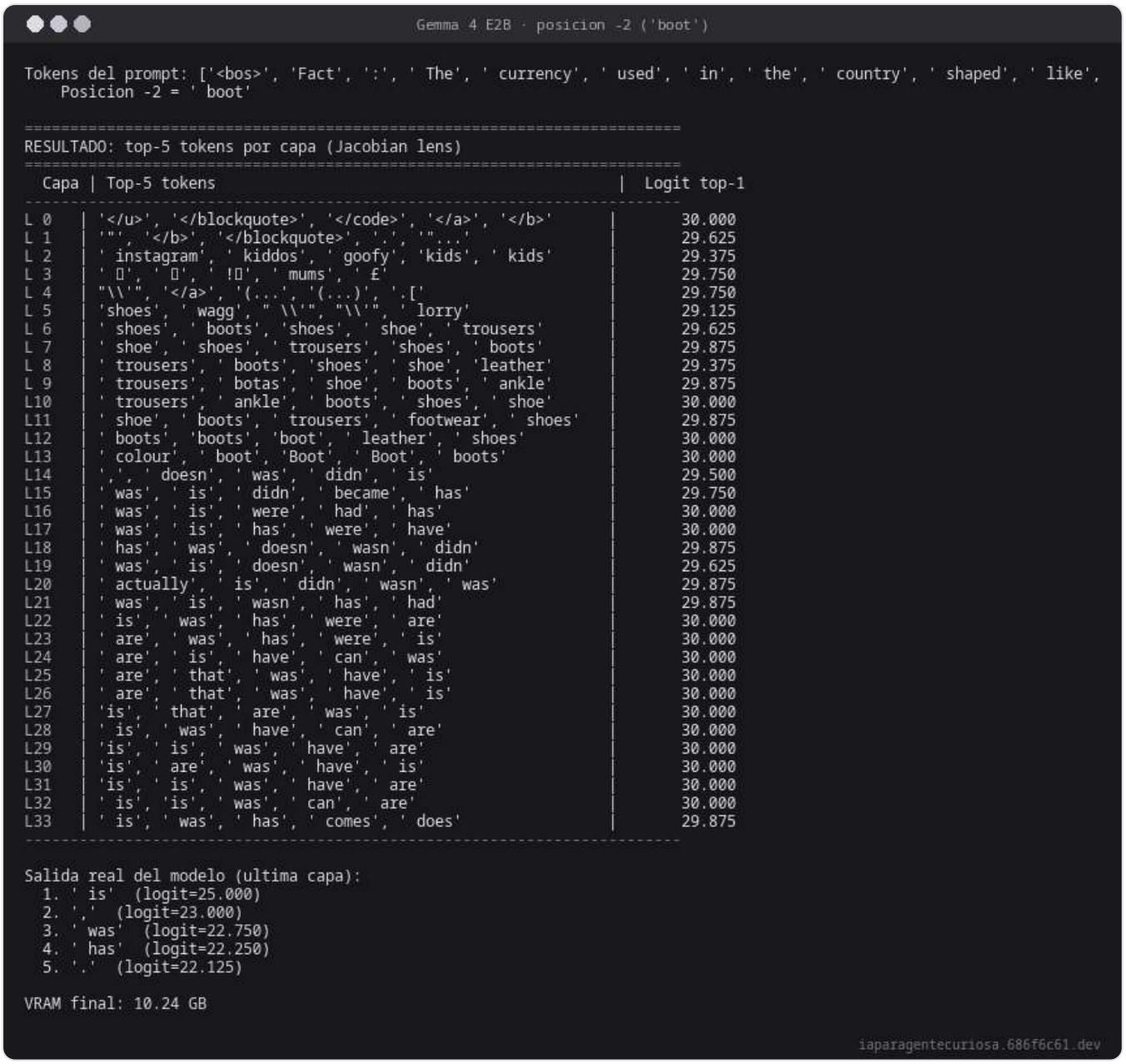



*Figura 8b. Salida real verificada al aplicar la Jacobian lens a Gemma 4 E2B en la posición -2 (token "boot"). Las capas 0-4 leen puntuación y HTML (ruido), L5-L13 leen calzado (shoes, boots, shoe), y L14-L33 estabilizan el verbo "is". Renderizada para publicación desde la salida de la máquina del autor.*

### 9.4.1. Qué es el "top-5" y el "logit top-1"

La tabla muestra, para cada capa, los **5 tokens con mayor logit** según la lens. El "top-5" es el ranking de los 5 tokens más probables según la lens en esa capa. El "logit top-1" es el logit (la puntuación cruda antes de softmax) del token que la lens pone en primer lugar.

**¿Por qué 5 y no 3 o 10?** 5 es un compromiso: con 3 tokens te pierdes candidatos interesantes (como "euros" en L28, que aparece en el top-4); con 10 tokens la tabla es ilegible. 5 es suficiente para ver el patrón dominante y los competidores cercanos.

**¿Por qué el logit y no la probabilidad?** La probabilidad (tras softmax) comprime los logits a $[0, 1]$ y los normaliza. Si el top-1 tiene logit 30 y el top-2 tiene logit 29, sus probabilidades son casi idénticas (0.999 y 0.0009), lo que no te dice nada. El logit crudo sí te dice cuánto "destaca" el top-1 sobre el resto. Por eso la tabla muestra el logit, no la probabilidad.

### 9.4.2. Lectura capa a capa

| RANGO DE CAPAS | QUÉ LEE LA LENS | INTERPRETACIÓN |
|---|---|---|
| L0-L4 | tokens HTML/puntuación ( `</u>` , `</blockquote>` , `'` , `£` ) | ruido residual, sin semántica. El modelo aún no ha procesado el contexto. |
| L5-L13 | **calzado**: `shoes` , `boots` , `shoe` , `botas` , `footwear` , `leather` | el modelo "ve" la palabra "boot" y la interpreta como *zapato*. Es procesamiento léxico: asocia el token con su categoría semántica. |
| L14-L22 | **verbos**: `was` , `is` , `has` , `were` , `are` | empieza a resolver la sintaxis: necesita un verbo. El modelo ha pasado de "qué es boot" a "qué verbo encaja aquí". |
| L22-L33 | **estabiliza en `is`** | la lens coloca `is` como candidato dominante para esa posición. La representación del verbo correcto se estabiliza. |

**Qué está pasando, en lenguaje llano.** En las primeras capas (L0-L4), el modelo no ha procesado nada: lee tokens de puntuación y HTML, que son los tokens más frecuentes en el corpus de entrenamiento. En las capas medias (L5-L13), el modelo "se da cuenta" de que la última palabra es "boot" y la asocia con calzado (shoes, boots, shoe): es procesamiento léxico puro, sin contexto. A partir de L14, el modelo empieza a procesar la sintaxis: necesita un verbo, y los verbos más probables son "was", "is", "has". Finalmente, estabiliza en "is", que es la respuesta correcta para esa posición.

Esto es exactamente lo que esperamos: Gemma 4 E2B resuelve la gramática ("the country ... is") pero, en la posición -2, aún no hace aparecer la moneda como candidata. La posición -2 es la de "boot": ahí la lectura se centra en el token anterior y en la estructura gramatical. La moneda aparece en la posición -1, donde el modelo ya tiene "is" y debe predecir qué viene después.

## 9.5. La posición -1: donde debería aparecer "euro"

Aquí está el hallazgo clave. Aplicamos la lens en la **posición -1** (el último token, "is", donde el modelo debe predecir el siguiente):

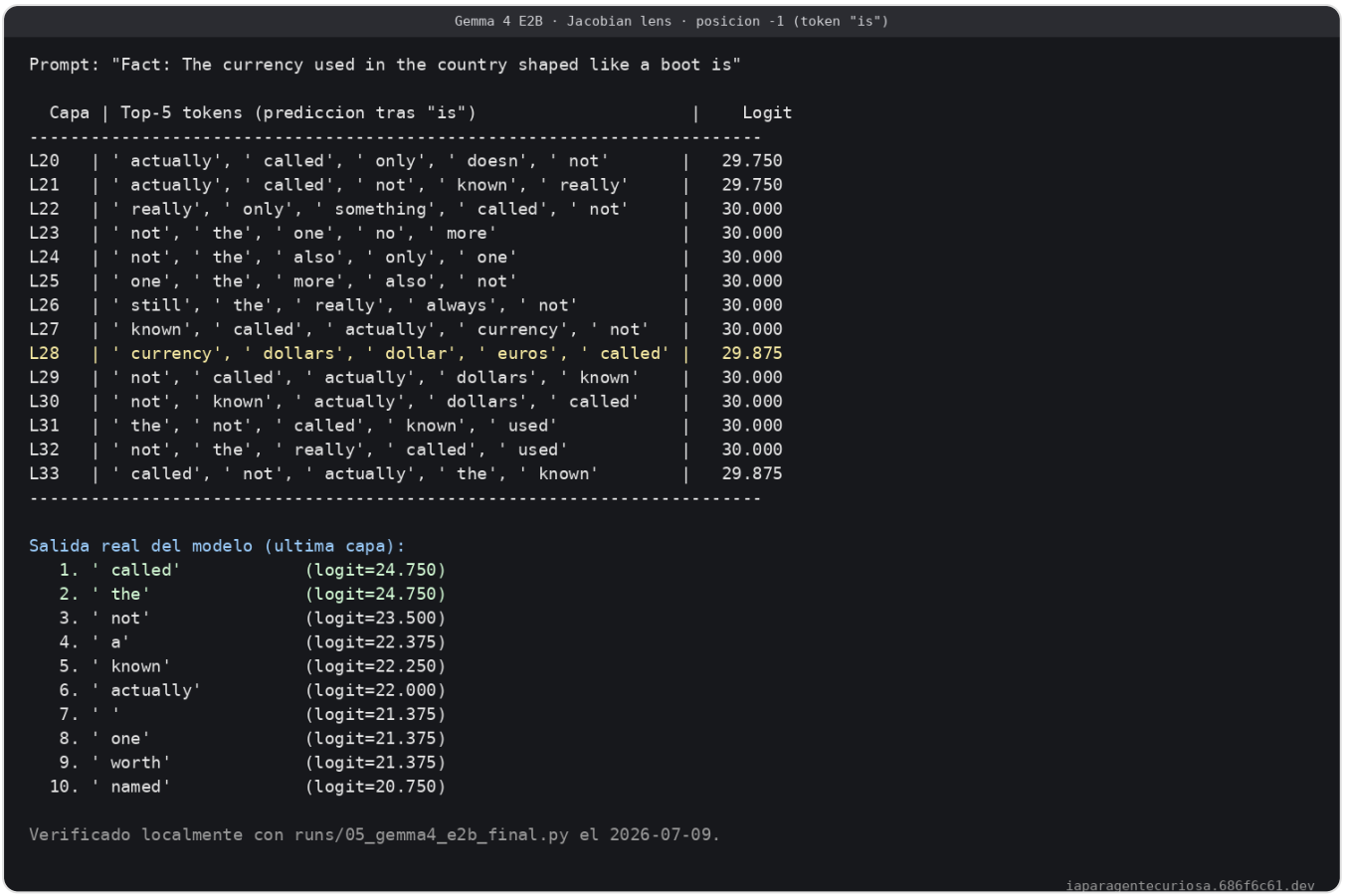



*Figura 9. Salida real verificada al aplicar la Jacobian lens a Gemma 4 E2B en la posición -1. La capa 28 lee `currency`, `dollars`, `dollar`, `euros`, la lens lee `euros` entre los candidatos de L28, pero la salida final prioriza `called` / `the`. Renderizada para publicación desde la salida de la máquina del autor.*

## 9.6. El hallazgo: `euros` aparece en L28

### 9.6.1. Lectura capa a capa de la posición -1

| CAPA | TOP-5 | QUÉ OCURRE |
|---|---|---|
| L0-L5 | puntuación, HTML | ruido. El residual stream aún no ha procesado el contexto. |
| L6-L8 | `smiley` , `chubby` , `mustache` , `snout` | artefactos tempranos de la lectura. No hay cara ASCII en este prompt; no conviene interpretarlos como evidencia semántica. |
| L9-L13 | `symbolizes` , `symbol` , `engraved` , `stamps` | interpreta "boot" como *símbolo* (la bota de Italia es un símbolo). |
| L14-L26 | `actually` , `not` , `called` , `known` , `one` , `the` | construye la sintaxis *"is called X"*. El modelo está preparando la estructura gramatical. |
| **L27** | `known` , `called` , `actually` , **`currency`** , `not` | **aparece "currency"** por primera vez en el top-5. |
| **L28** | **`currency`** , `dollars` , `dollar` , **`euros`** , `called` | **aparece "euros"** en el top-4. La respuesta correcta aparece como candidata fuerte en la lectura de la lens. |
| L29-L33 | `not` , `called` , `known` , `the` | la representación de "euros" se diluye. El modelo redirige hacia "called". |
| Salida final | `called` , `the` , `not` , `a` , `known` | la salida final empata arriba entre `called` y `the` ; el script lista `called` primero. |

### 9.6.2. Qué está pasando, en lenguaje llano

En las capas tempranas (L0-L5), la lectura está dominada por puntuación y HTML, como en la posición -2. En L6-L8 aparecen tokens como `smiley` , `chubby` , `mustache` o `snout` . No hay que sobrerreaccionar: este prompt no contiene una cara ASCII, así que esos tokens son mejor tratados como ruido o sesgos tempranos del transporte lineal, no como una interpretación fiable del contexto.

En L9-L13, el modelo interpreta "boot" como un **símbolo** ( `symbolizes` , `symbol` , `engraved` , `stamps` ): la bota de Italia es un símbolo geográfico. Es un paso intermedio: el modelo ya no ve calzado, ve un símbolo, pero aún no ha llegado a la moneda.

En L14-L26, el modelo construye la **sintaxis** de la respuesta: `actually` , `not` , `called` , `known` , `one` , `the` . Está preparando la estructura gramatical *"is called X"* o *"is actually X"*. Es como si el modelo estuviera montando el andamiaje de la frase antes de rellenar la X.

En **L27**, aparece `currency` por primera vez en el top-5. En **L28**, aparece `euros` en el top-4, junto con `currency`, `dollars`, `dollar`. **El token correcto aparece como candidato fuerte en la capa 28.** Ese es el dato clave: la representación queda verbalizable en esa capa.

Pero en L29-L33, la representación de "euros" se **diluye**: desaparece del top-5 y el modelo redirige hacia `called`, `not`, `known`, `the`. La salida final del modelo es `called`, no `euros`.

### 9.6.3. Disponibilidad frente a salida

**Esta es la distinción clave entre disponibilidad y salida.** En la capa 28, `euros` aparece junto con `currency`, `dollars` y `dollar`. Pero esa representación **no llega a la salida**: en la última capa, el ranking final prioriza `called` y `the` por delante de `euros`. La lens nos ha permitido ver que la respuesta correcta estuvo disponible como candidato de la lens, pero la verbalización final fue otra.

**La analogía de la calle.** Imagina que el residual stream es una cesta que pasa por puestos. En L28 alguien mete `euros`, pero los puestos finales inclinan la cesta hacia `called` y `the`, continuaciones frecuentes de una frase como *"is called X"* o *"is the..."*. La lens no lee pensamientos: enseña qué tokens resultan legibles en cada puesto.

### 9.6.4. La gráfica de rangos: cómo leerla

La evolución del rango de los tokens candidatos a lo largo de las capas lo muestra con claridad:

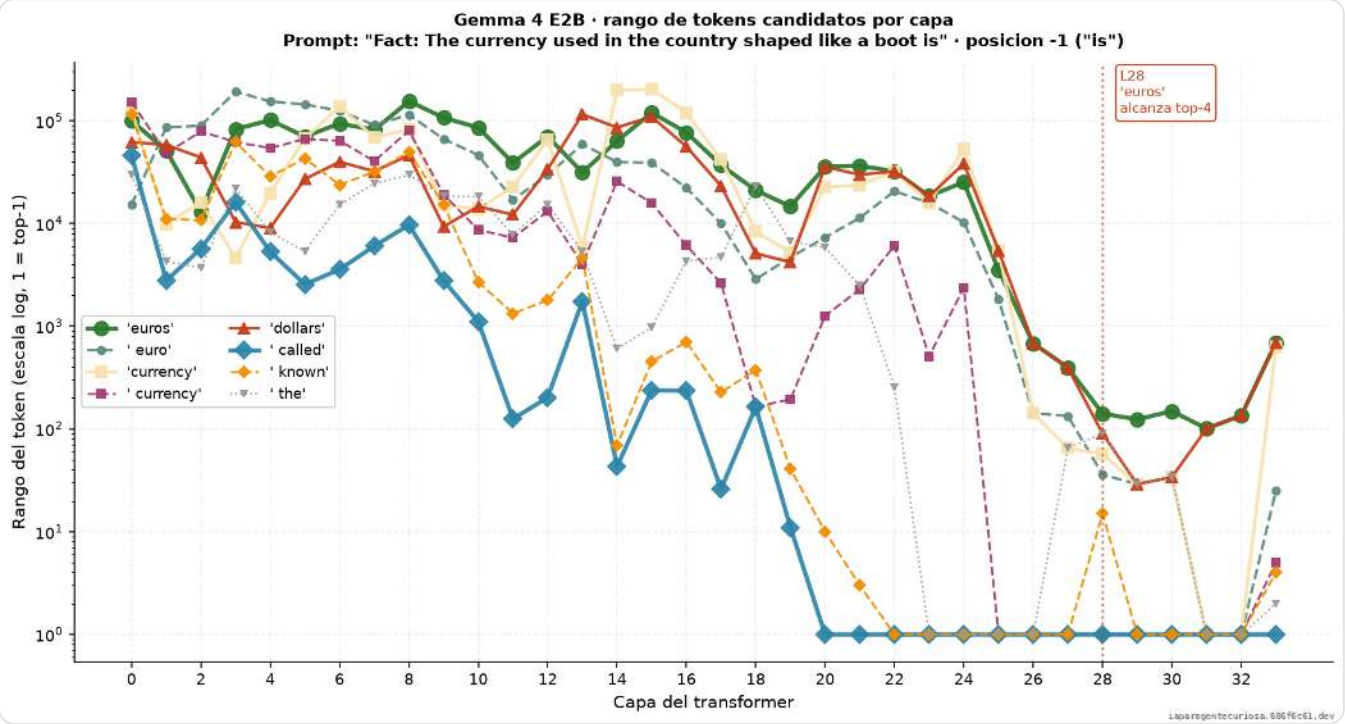



*Figura 11. Rango (escala log) de varios tokens candidatos en la posición -1, por capa, en Gemma 4 E2B. La línea roja vertical marca L28, donde "euros" alcanza su mejor rango (top-4 del vocabulario de 262 k tokens). Obsérvese cómo "called" (el token que el modelo acaba*

*diciendo) mejora en las capas finales, mientras que "euros" se diluye. El token aparece en L28, pero no llega como salida final. Figura propia con datos reales extraídos de la lens.*

**¿Qué es el "rango" de un token?** El rango de un token es su posición en el ranking de logits. Si el vocabulario tiene 262.144 tokens y "euros" tiene el logit número 4 más alto, su rango es 4. El rango 1 es el top-1 (el token que el modelo "diría"). Un rango bajo (1, 2, 3) significa que el token está muy arriba en la lista; un rango alto (100.000) significa que está muy abajo.

**¿Por qué escala log?** Porque el rango va de 1 a 262.144. Si dibujamos la escala lineal, los rangos bajos (1-100) se ven aplastados al fondo de la gráfica y no se distingue nada. La escala logarítmica expande la zona baja (donde están los tokens interesantes) y comprime la zona alta. Un rango de 4 y un rango de 40 se ven claramente separados en escala log, pero casi pegados en escala lineal.

**La línea roja vertical** marca L28, el momento en que "euros" alcanza su mejor rango (top-4). A la izquierda de la línea, "euros" tiene rangos altos (no aparece en el top-5). A la derecha, su rango empeora (sube) hasta desaparecer del top. Mientras tanto, "called" (el token que el modelo acaba diciendo) mejora su rango en las capas finales, bajando hacia el rango 1.

### 9.6.5. Por qué "euros" se diluye

**Por qué "euros" se diluye:** entre L28 y L33, las capas finales observadas por la lens redirigen la representación. El modelo "decide" que la respuesta segura es "called" (que encaja con la sintaxis *"is called X"*) en lugar de arriesgarse con "euros". Es un ejemplo de cómo el modelo puede tener el conocimiento pero preferir una verbalización más conservadora.

¿Qué significa "redirigir"? En un transformer, cada capa añade su contribución al residual stream. Si las capas L29-L33 añaden una contribución que "empuja" el residual hacia la dirección de "called" y "aleja" el residual de la dirección de "euros", el resultado neto es que "called" gana y "euros" pierde. La lens nos deja ver exactamente dónde ocurre ese giro: entre L28 y L33.

**La analogía de la calle.** Imagina que el residual stream es una cesta que va pasando por 35 puestos. En el puesto 28, alguien mete "euros" en la cesta. Pero en los puestos 29 al 33, otros puestos sacan "euros" y meten "called". Al llegar al final, la cesta tiene "called", no "euros". La lens te muestra qué había en la cesta en cada puesto, no solo al final. Si quieres que el modelo diga "euros", tendrías que intervenir entre el puesto 28 y el 33: o bien impedir que saquen "euros", o bien meter más "euros" para que no puedan sacarlo todo.

### 9.6.6. Por qué esto es importante

**Por qué esto es importante:** este patrón, una representación correcta que aparece y luego no llega a la salida, es exactamente el tipo de fenómeno que una lens ayuda a localizar. Sin la lens, solo verías que la salida prioriza `called` y `the`; con la lens, ves que `euros` estuvo en el top-5 en L28 y que el giro ocurre después.

La lectura es coherente con la tesis del paper (Anthropic, 2026): las representaciones verbalizables que la lens puede leer forman un *workspace* que no siempre coincide con la salida.

---

## 10. Hands-on 4: ejemplos propios

Para mostrar que la lens no solo funciona con los ejemplos del repo de Anthropic, creamos tres ejemplos propios con distintos niveles de dificultad. Esto es importante porque demuestra que la lens es una herramienta general, no un truco que solo funciona en los prompts cuidadosamente elegidos del paper.

Los tres ejemplos cubren tres tipos de razonamiento:

- **Aritmética** (2+2=): un hecho elemental, sin ambigüedad, que un modelo de este tamaño debería saber.
- **Hecho geográfico de un salto** (capital de Japón): un hecho factual que requiere un solo paso de razonamiento (Japón → Tokio).
- **Razonamiento multi-salto** (Don Quijote → idioma): un hecho que requiere encadenar dos pasos (Don Quijote → Cervantes → español).

La pregunta, en los tres casos, es la misma: ¿aparece la respuesta como candidata interna y llega a la salida, aparece pero se pierde, o no aparece en las capas observadas? La lens permite distinguir estos casos en un modelo abierto sin depender solo de la salida final.

### 10.1. Ejemplo propio 1: aritmética elemental

**Prompt:** `"The result of the operation is: 2 + 2 ="`

El prompt se tokeniza como 14 tokens: `['<bos>', 'The', ' result', ' of', ' the', ' operation', ' is', ':', ' ', '2', ' +', ' ', '2', ' =']`. La posición -1 es `=`, y el modelo debería predecir `4` o `four`. Es un hecho elemental que cualquier modelo de este tamaño debería saber.

#### 10.1.1. Salida capa a capa de la lens

Aplicamos la lens en la posición -1 (`=`). Estos son los top-5 tokens por capa (datos reales):

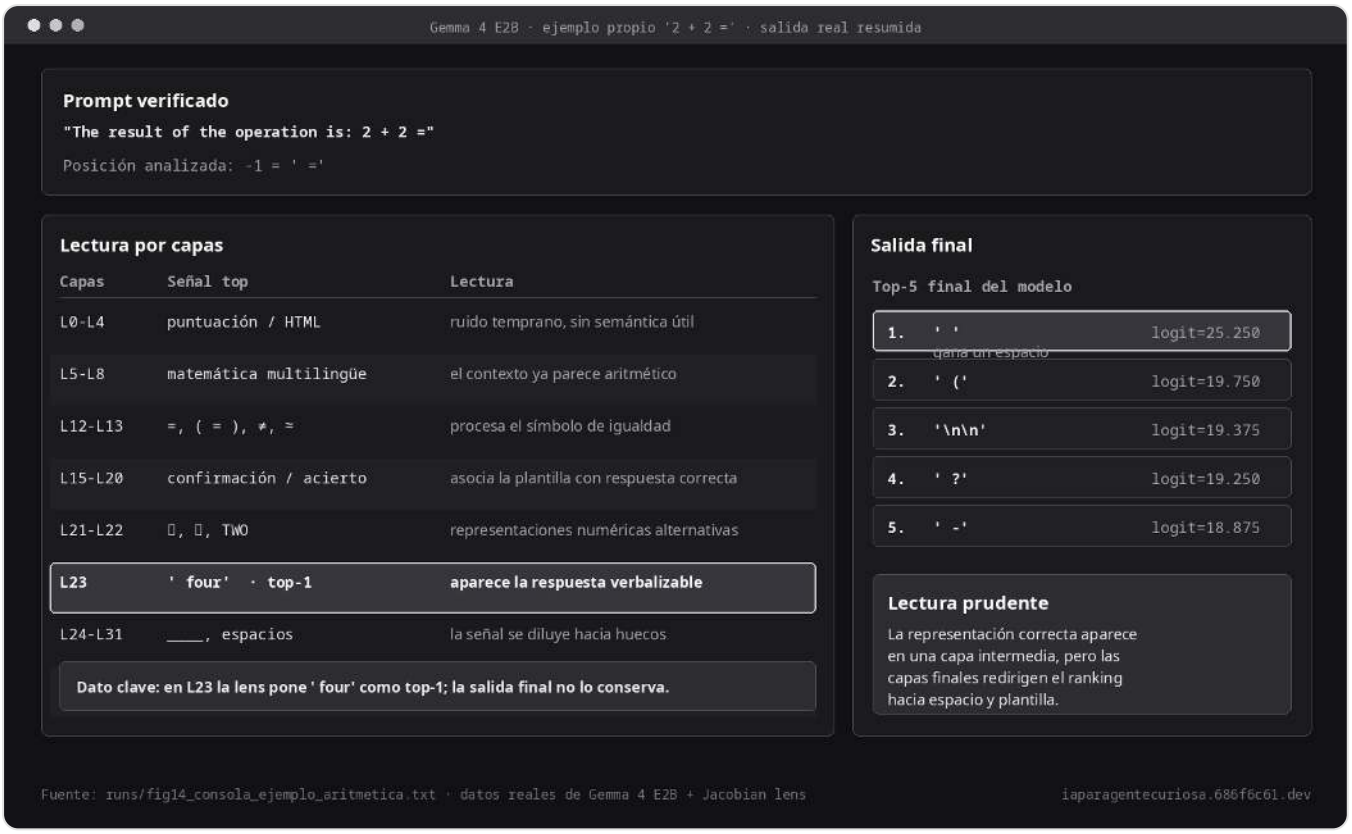



*Figura 12b. Extracto real resumido del ejemplo propio "2 + 2 =" en Gemma 4 E2B. La capa 23 lee "four" como top-1, pero la salida final del modelo es un espacio. Renderizado para publicación desde la salida verificada de la máquina del autor.*

### 10.1.2. Lectura capa a capa

| RANGO DE CAPAS | QUÉ LEE LA LENS | INTERPRETACIÓN |
|---|---|---|
| L0-L4 | puntuación, HTML, caracteres raros ( `’` , `</strong>` , `□□□□` ) | ruido residual, sin semántica. El modelo aún no ha procesado la operación. |
| L5-L8 | términos matemáticos en otros idiomas ( `математи` , `aritmética` , `matemat` , `math` , `алге` ) | el modelo "ve" que el contexto es matemático, pero aún no ha calculado. Asocia el contexto con la categoría "matemáticas". |
| L9-L13 | símbolos de igualdad y operadores ( `=)` , `=` , `(=` , `≠` , `=)` , `≈` ) | el modelo procesa el símbolo `=` y sus variantes. Es procesamiento sintáctico: reconoce el operador. |
| L14-L20 | **emojis** (□, □, ☑, □, 😂, 😉, □) | curioso: el modelo asocia "resultado correcto" con emojis de confirmación (□, ☑). Es un eco del entrenamiento en redes sociales, donde los resultados se marcan con emojis. |
| L21-L22 | números en otros alfabetos ( `١` , `٢` = 1, 2 en persa) y `TWO` | el modelo "ve" números, pero en representaciones alternativas. |
| **L23** | `four` , `=` , `five` , `_____` , `?` | **aparece "four" como top-1**. La respuesta correcta aparece como candidata fuerte en la lectura de la lens. |
| L24-L31 | guiones bajos ( `_____` , `_________` ) y espacios | la representación de "four" desaparece. El modelo redirige hacia marcadores de hueco ( `_____` ), como si esperara que alguien rellenara el resultado. |
| L32-L33 | ` ` , `4` , `XNUMX` , `_____` , `x` | "four" desaparece del top-5. Aparece `4` (el dígito) en el top-4 de L32 y top-2 de L33, pero el top-1 es un espacio. |
| Salida final | ` ` , `(` , `\n\n` , `?` , `-` | la salida prioriza un espacio. |

### 10.1.3. El hallazgo: aparece y se pierde

**Esto es sorprendente.** La operación 2+2=4 es el hecho más elemental que cabe imaginar. Y, sin embargo, la salida final es un espacio, no "four" ni "4". Pero la lens muestra que en la capa 23, "four" fue el top-1 del vocabulario de 262.144 tokens. La representación apareció en L23 y luego se perdió.

**La analogía de la calle.** En el cruce 23 alguien mete `four` en la cesta. Entre ese cruce y el final, las capas posteriores vuelven a empujar hacia espacios, huecos o continuaciones de plantilla. La lens te muestra el punto donde aparece `four` ; la salida final muestra que no

sobrevivió hasta el último ranking.

**Por qué la salida puede ser un espacio.** El prompt es `"The result of the operation is: 2 + 2 ="`. Esa forma se parece a plantillas de ejercicios, formularios o texto donde el resultado va después de un hueco. No podemos demostrar la causa solo con esta lens, pero el patrón observado es claro: `four` aparece en L23 y las capas finales priorizan un espacio.

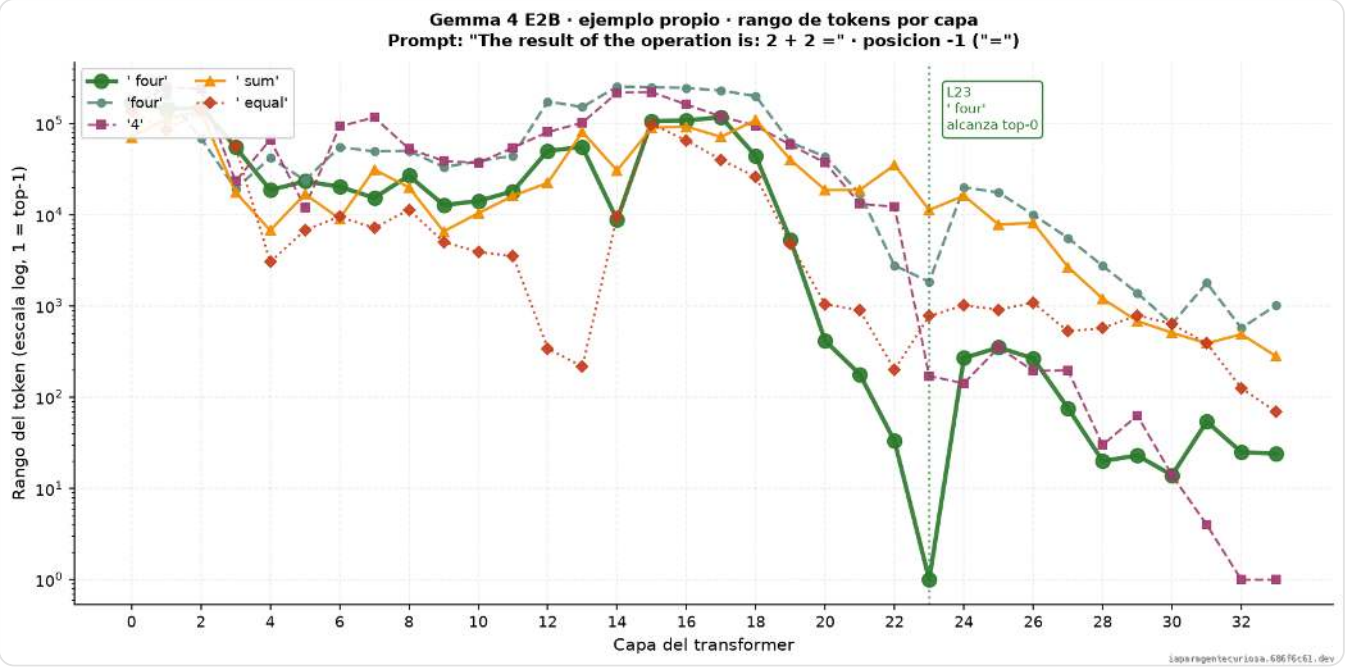



*Figura 12. Rango (escala log) de tokens candidatos en la posición -1 ( `=` ), por capa, en Gemma 4 E2B, para el prompt propio "The result of the operation is: 2 + 2 =". El token " four" alcanza el rango 0 (top-1) en la capa 23, pero la salida final del modelo es un espacio. Figura propia con datos reales.*

## 10.2. Ejemplo propio 2: hecho geográfico de un solo salto

**Prompt:** `"The capital of Japan is"`

El prompt se tokeniza como 6 tokens: `['<bos>', 'The', ' capital', ' of', ' Japan', ' is']`. La posición -1 es `is`, y el modelo debería predecir `Tokyo`. Es un hecho de un solo salto (Japón → Tokio), más simple que el de la moneda (que requiere dos saltos: bota → Italia → euro).

### 10.2.1. Salida capa a capa de la lens

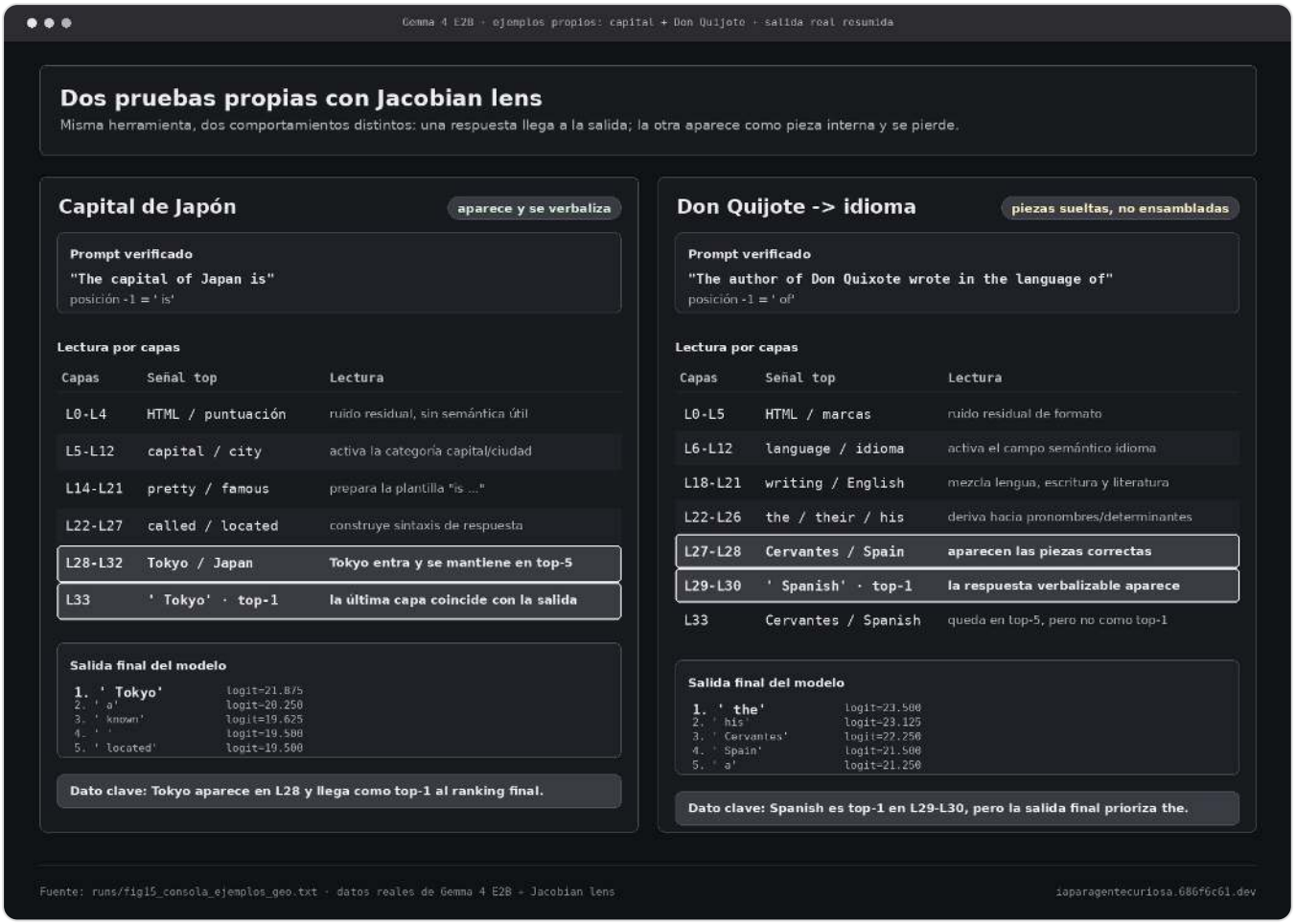



*Figura 13b. Extracto real resumido de los ejemplos propios "The capital of Japan is" y "The author of Don Quixote wrote in the language of" en Gemma 4 E2B. En el primero, "Tokyo" aparece en L28 y llega a la salida. En el segundo, "Spanish" es top-1 en L29-L30 pero la salida es "the" (piezas sueltas, no ensambladas). Renderizado para publicación desde la salida verificada de la máquina del autor.*

### 10.2.2. Lectura capa a capa

| RANGO DE CAPAS | QUÉ LEE LA LENS | INTERPRETACIÓN |
|---|---|---|
| L0-L4 | puntuación, HTML | ruido residual, sin semántica. |
| L5-L9 | "capital" en otros idiomas ( `столи` = ruso, `□□` = chino, `città` = italiano, `□□□□□□□` = hindi, `ciudad` = español) | el modelo "ve" que el contexto es sobre capitales y activa la categoría semántica en varios idiomas. Es procesamiento léxico multilingüe. |
| L10-L12 | `city` , `cities` , `Beijing` | el modelo asocia "capital" con "ciudad" y, curiosamente, con `Beijing` (la capital de otro país asiático). Es una activación por proximidad geográfica. |
| L14-L21 | adjetivos ( `pretty` , `very` , `amazing` , `incredibly` , `beautiful` , `famous` ) | el modelo prepara la sintaxis *"is [adjetivo] X"*, una estructura frecuente tras "is". |
| L22-L27 | `also` , `not` , `what` , `called` , `located` , `known` , `Japan` | el modelo construye la sintaxis de la respuesta. Aparece `Japan` (el propio país) y `called` / `located` / `known` (verbos de la estructura *"is called/located/known X"*). |
| **L28** | `Beijing` , **`Tokyo`** , `□□` , `Jakarta` , `Japan` | **aparece "Tokyo" por primera vez** en el top-5 (top-2), junto con `□□` (Tokio en japonés) y `Beijing` (otra capital asiática). La respuesta correcta aparece como candidata fuerte en la lectura de la lens. |
| L29-L32 | `located` , `known` , `Japan` , `Tokyo` , `□□` | "Tokyo" se mantiene en el top-5, alternando con `Japan` y `located` . La representación es estable. |
| **L33** | **`Tokyo`** , `located` , `known` , `Kyoto` , `Japan` | **"Tokyo" es el top-1** en el último `source_layer` de la lens. La lectura de la lens y la salida final coinciden en el top-1. |
| Salida final | `Tokyo` , `a` , `known` , `` , `located` | la salida prioriza Tokio. Correcto. |

### 10.2.3. El hallazgo: aparece y se verbaliza

**Este es el caso opuesto al de la moneda y al de la aritmética.** Tokyo aparece en L28 y llega a la salida (la salida es "Tokyo"). La lens y la salida coinciden.

**Por qué aquí sí y en los otros no.** La capital de Japón es un hecho de un solo salto, muy frecuente en el corpus de entrenamiento, sin ambigüedad práctica. La estructura *"The capital of Japan is X"* es una plantilla directa donde X es la capital, y `Tokyo` se mantiene hasta el

último ranking.

**La analogía de la calle.** En el cruce 28 entra `Tokyo` en la cesta. A diferencia de los otros ejemplos, los puestos finales no lo sustituyen: lo mantienen y llega a la salida.

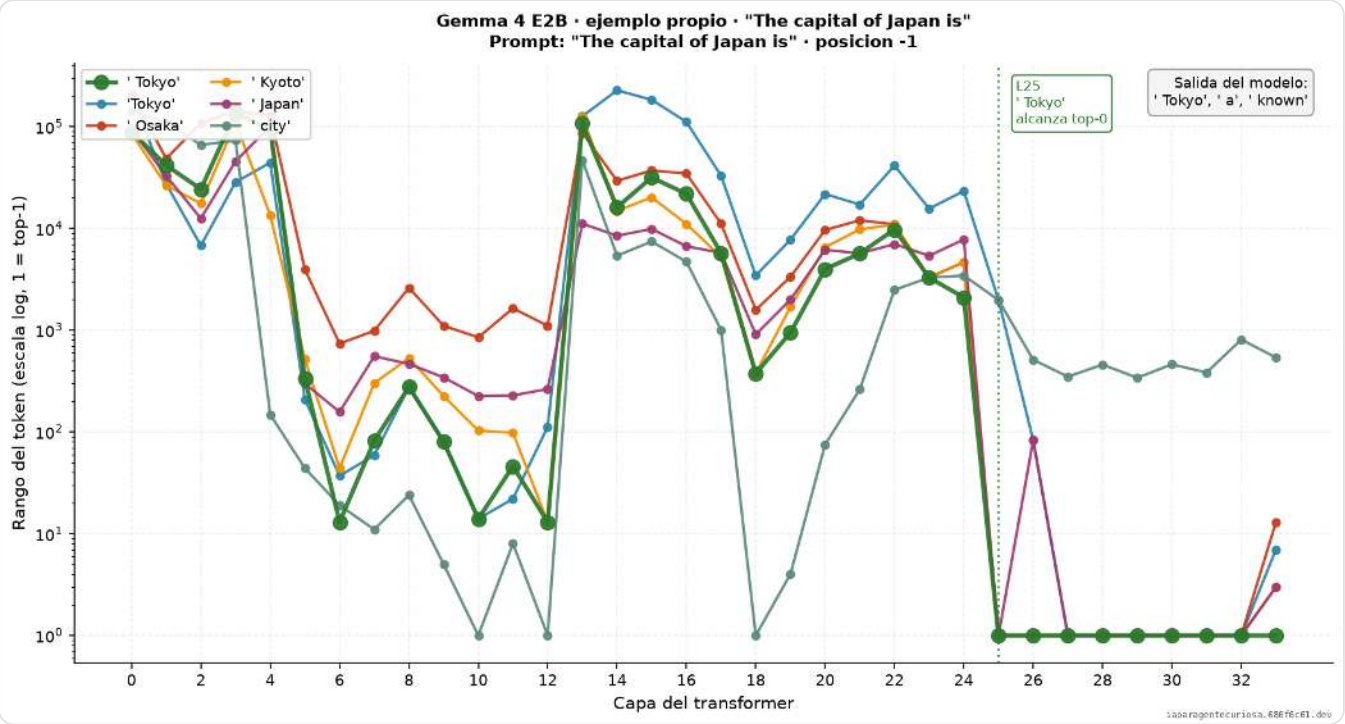



*Figura 13. Rango (escala log) de tokens candidatos en la posición -1 ( `is` ), por capa, en Gemma 4 E2B, para el prompt propio "The capital of Japan is". El token " Tokyo" aparece en el top-5 a partir de L28 y se mantiene hasta la salida, donde es el top-1. Figura propia con datos reales.*

## 10.3. Ejemplo propio 3: razonamiento multi-salto

**Prompt:** `"The author of Don Quixote wrote in the language of"`

El prompt se tokeniza como 13 tokens: `['<bos>', 'The', ' author', ' of', ' Don', ' Qu', 'ix', 'ote', ' wrote', ' in', ' the', ' language', ' of']`. Fíjate que "Quixote" se tokeniza en tres piezas ( `Qu` , `ix` , `ote` ), porque el tokenizador de Gemma 4 no lo tiene como token único. La posición -1 es `of` , y el modelo debería predecir `Spanish` . Esto requiere dos saltos: Don Quijote → Cervantes → idioma de Cervantes (español). Es el mismo patrón que el de la moneda, pero con conocimiento cultural en vez de geográfico.

### 10.3.1. Salida capa a capa de la lens

La salida completa de este ejemplo está en la segunda mitad de la captura anterior; debajo queda la lectura resumida por rangos de capas.

### 10.3.2. Lectura capa a capa

| RANGO DE CAPAS | QUÉ LEE LA LENS | INTERPRETACIÓN |
|---|---|---|
| L0-L4 | puntuación, HTML, palabras formales ( `defence` , `authorised` , `programme` ) | ruido residual, sin semántica. |
| L6-L12 | "lenguaje" en varios idiomas ( `языка` = ruso, □□ = chino, `linguagem` = portugués, `lenguaje` = español, `Language` , `languages` , `linguistic` ) | el modelo "ve" que el contexto es sobre lenguaje y activa la categoría semántica en varios idiomas. Curiosamente, aparece `Arabic` en L12: el modelo considera varios idiomas candidatos. |
| L18-L21 | `writing` , `language` , `literature` , `languages` , `English` , `popular` , `centuries` , `modern` | el modelo procesa el contexto literario: "escribió en el lenguaje de...". Aparece `English` como candidato, lo que muestra que el modelo está considerando qué idioma podría ser. |
| L22-L26 | `the` , `his` , `their` , `its` , `this` , `himself` | el modelo construye la sintaxis posesiva: *"wrote in the language of his..."*. Prepara la estructura gramatical. |
| **L27** | `his` , `Spaniards` , **`Cervantes`** , `Spain` , `poetry` | **aparece "Cervantes"** por primera vez en el top-5 (top-3), junto con `Spain` y `Spaniards` . El modelo ha identificado al autor y su país. |
| **L28** | **`Cervantes`** , `his` , **`Spanish`** , `Spain` , `Shakespeare` | **aparece "Spanish"** por primera vez (top-3), junto con `Cervantes` (top-1) y `Spain` . El modelo tiene las tres piezas del razonamiento: autor, país e idioma. |
| **L29-L30** | **`Spanish`** , `his` , `poetry` , `literary` , `Spain` | **"Spanish" es el top-1** en L29 y L30. El token correcto queda en la posición más alta del ranking de la lens. |
| L31-L32 | `Spain` , `his` , `Spanish` , `Madrid` , `literature` | "Spanish" baja al top-3/top-4. La representación se diluye ligeramente, pero sigue en el top-5. |
| **L33** | `Cervantes` , `Spain` , **`Spanish`** , `his` , `the` | en el último `source_layer` de la lens, "Spanish" sigue en el top-3. La representación no se ha perdido del todo. |
| Salida final | `the` , `his` , `Cervantes` , `Spain` , `a` | la salida prioriza `the` . |

### 10.3.3. El hallazgo: piezas sueltas, no ensambladas

**Este es el caso más rico de los tres.** La lens hace aparecer las tres piezas del razonamiento:

1. **El autor**: `Cervantes` aparece en el top-5 desde L27, y es el top-1 en L28 y L33.
2. **El país**: `Spain` aparece en el top-5 desde L27, y es el top-1 en L31-L32.
3. **El idioma**: `Spanish` aparece en el top-5 desde L28, y es el **top-1 en L29 y L30**.

Las tres piezas aparecen en capas intermedias. Y, sin embargo, la salida del modelo es `the`, no `Spanish`. La lectura sugiere que las piezas están disponibles, pero no se ensamblan en la respuesta final.

**La analogía de la calle.** En L28 entran `Cervantes`, `Spanish` y `Spain`; en L29-L30 domina `Spanish`. Las capas finales, sin embargo, vuelven hacia `the`, `his`, `Cervantes` y `Spain`, una continuación compatible con frases como *"the language of his..."*. La lens muestra que las piezas estuvieron disponibles, pero no que se integraran en una respuesta final.

**Por qué "Spanish" no llega a la salida.** El prompt termina en *"the language of"*. Esa estructura también admite continuaciones posesivas (*"the language of his country"*, *"the language of his people"*). No podemos afirmar la causa exacta solo con esta prueba, pero la salida final muestra que las capas finales prefieren una continuación gramatical ( `the` / `his` ) a verbalizar `Spanish`.

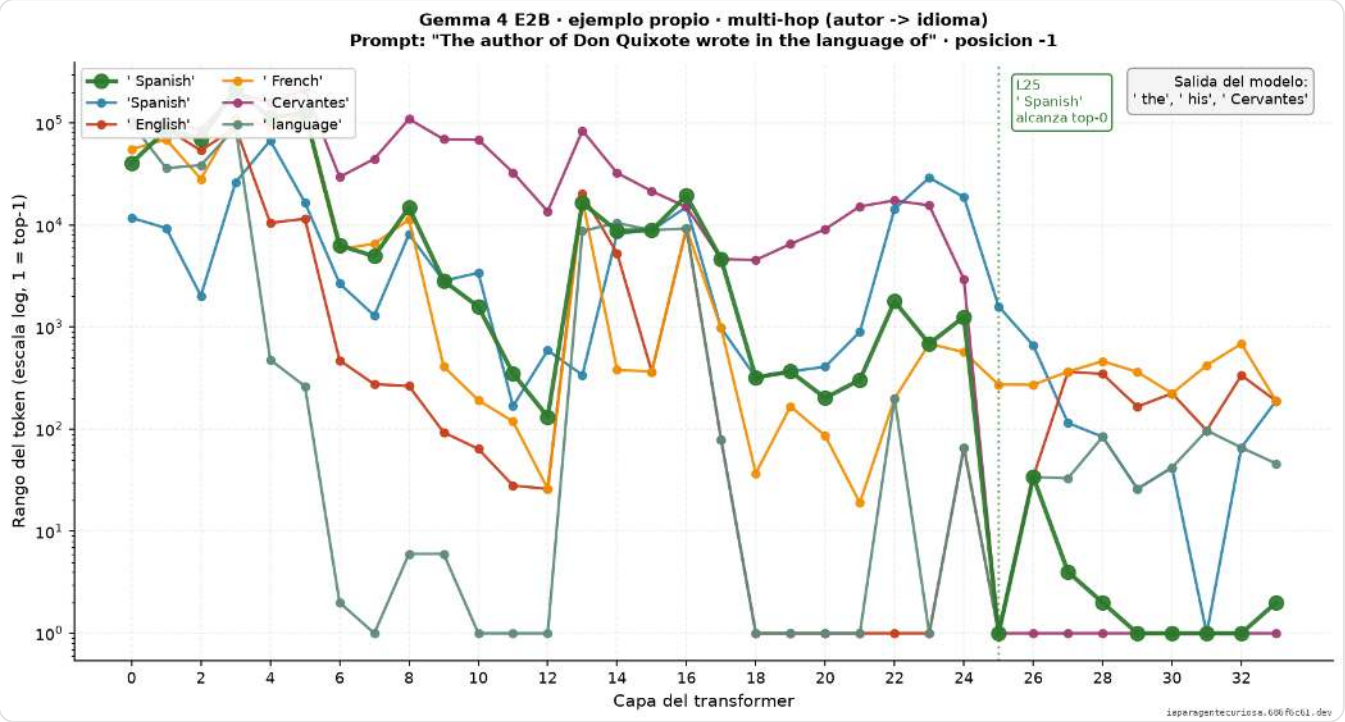



*Figura 14. Rango (escala log) de tokens candidatos en la posición -1 ( `of` ), por capa, en Gemma 4 E2B, para el prompt propio "The author of Don Quixote wrote in the language of". El token " Spanish" alcanza el top-1 (rango 0) en las capas 29 y 30, pero la salida final del*

*modelo es "the". Figura propia con datos reales.*

### 10.4. Los tres patrones, comparados

Los tres ejemplos juntos muestran los patrones posibles:

| PATRÓN OBSERVADO | EJEMPLO | SEÑAL INTERNA | CAPA | SALIDA TOP | COINCIDEN |
|---|---|---|---|---|---|
| Aparece y se verbaliza | Capital de Japón | `Tokyo` aparece | L28-L33 | `Tokyo` | sí |
| Aparece y no se verbaliza | Moneda de Italia | `euros` aparece en top-5 | L28 | `called` / `the` | no |
| Aparece y no se verbaliza | 2+2= | `four` aparece como top-1 | L23 | espacio | no |
| Piezas sueltas | Don Quijote | `Cervantes`, `Spain`, `Spanish` aparecen | L27-L33 | `the` | no |

**La regularidad es útil, pero hay que leerla con cuidado.** En los cuatro casos aparece una señal interna relevante en capas intermedias (L23-L30). La diferencia está en si esa señal llega o no a la salida. Solo en el caso de la capital de Japón la representación llega intacta. En los demás, las capas finales redirigen el ranking hacia una continuación más conservadora o más frecuente en la plantilla del prompt.

**La analogía de la calle.** Imagina que el residual stream es una cesta que va pasando por 35 puestos. En todos los casos, alguien mete la respuesta correcta en la cesta en algún puesto intermedio (L23-L30). La diferencia es qué pasa después: en el caso de Tokio, nadie saca la respuesta y llega al final; en los demás, los puestos finales la sacan y meten otra cosa (un espacio, "called", "the"). La lens te muestra qué había en la cesta en cada puesto, no solo al final.

La lens permite distinguir estos patrones sin adivinar solo desde la salida. Sin la lens, solo verías que el modelo prioriza "espacio", "called", "Tokyo" o "the"; con la lens, puedes localizar si un candidato correcto apareció antes y dónde se perdió.

---

## 11. Limitaciones y advertencias

La Jacobian lens es una herramienta potente, pero tiene limitaciones importantes que conviene conocer antes de usarla. Esta sección las repasa, porque usar la lens sin entender sus límites lleva a conclusiones equivocadas.

### 11.1. Es una implementación de referencia

El propio README de Anthropic lo dice: *"Not maintained and not accepting contributions"*. No está optimizada para velocidad; el tiempo de *fit* lo domina el *backward* del modelo. Esto significa dos cosas:

- **No la uses en producción.** El código es de investigación, no de producción. Puede tener bugs, no está optimizada, y nadie la mantiene. Si la usas para algo serio, haz tus propias pruebas y considera reimplementar las partes críticas.
- **El *fit* es lento.** Calcular la Jacobiana de un modelo de 5 B parámetros con 455 prompts tarda horas en una GPU de consumo. Si quieres ajustar una lens tú mismo en hardware de consumo, prepárate para esperar y mide memoria y tiempo antes de prometer resultados.

### 11.2. Solo funciona con decoders

La lens asume un **residual stream causal**: una arquitectura donde la información fluye de izquierda a derecha, capa a capa, sin bidireccionalidad. Esto excluye:

- **Encoders** (BERT, RoBERTa): son bidireccionales, cada posición ve todas las demás. La lens no sabe transportar en un espacio bidireccional.
- **Encoder-decoder** (T5, BART): tienen dos stacks de capas. La lens está diseñada para un solo stack.
- **Modelos con arquitecturas no estándar** (Mamba, RWKV): la lens asume capas residuales aditivas ($h^{(l)} = h^{(l-1)} + \text{Bloque}_l$). Otros mecanismos de recurrencia no encajan.

La lens funciona con Llama, Qwen, Mistral, Gemma, GPT-2, Pythia, OLMo: todos son decoders causales con residual stream aditivo.

### 11.3. No con APIs cerradas

Necesitas los pesos y el grafo de cálculo: no se puede aplicar a Claude, GPT-4 o Gemini vía API. Solo a modelos de pesos abiertos cargados localmente. Esto es porque la lens necesita:

- **Los pesos**: para hacer *forward* y *backward*.
- **El grafo**: para registrar *hooks* en cada capa residual y para inyectar cotangentes en el *backward*.
- **Acceso a las activaciones intermedias**: para capturar $h^{(l)}$ en cada capa.

Una API cerrada no te da nada de eso. Solo te da la salida final. Por eso la lens es una herramienta de **interpretabilidad mecanicista**, no de interpretabilidad de caja negra.

### 11.4. La lens es un promedio

$J_l$ es la **Jacobiana media** sobre un corpus (wikitext). Esto tiene una consecuencia importante: la lens es un promedio sobre la distribución de datos, no una garantía para cada prompt concreto.

Para prompts **dentro de distribución** (texto similar a wikitext: inglés, prosa, hechos enciclopédicos), la lens funciona bien. Para prompts **fuera de distribución** (código, otros idiomas, prompts muy cortos o muy largos), el transporte puede ser menos informativo. La Jacobiana media no captura bien cómo se propaga la información en regímenes que no vio durante el ajuste.

**La analogía de la calle.** La lens es como un mapa de la avenida hecho promediando muchas cestas en un día típico. Si tu cesta es típica (pan, fruta, leche), el mapa te dice bien a dónde llega. Si tu cesta es rara (un piano, un elefante), el mapa ya no sirve, porque los promedios se hicieron sin cestas raras.

### 11.5. "Leer" no es "decir"

Que la lens lea "euros" en L28 no significa que el modelo vaya a decir "euros": la salida final puede ser otra (como vimos en la sección 9 y en los ejemplos propios). La lens muestra **qué representación está disponible** en el residual stream de esa capa, no **qué se verbaliza** al final.

Esta distinción es crucial y es el hallazgo central del artículo:

- **Disponibilidad**: la representación "euros" está en el residual stream de L28. El modelo *podría* verbalizarla.
- **Verbalización**: la salida del modelo es "called". El modelo *no la verbaliza*.

La lens mide la disponibilidad, no la verbalización. Si confundes las dos, concludes que el modelo "dice euros" cuando en realidad "dice called". La lens indica disponibilidad de una representación, no garantiza que esa representación se convierta en salida.

### 11.6. El ejemplo ascii-face es de Anthropic

No es nuestro; viene en `jlens/examples.py`. Lo mantenemos por su valor pedagógico (es el ejemplo canónico del paper) y lo señalamos claramente cada vez que aparece. Los ejemplos de la sección 10 (aritmética, capital de Japón, Don Quijote) sí son nuestros, creados para esta pieza.

### 11.7. La lens es lineal

La lens transporta $h^{(l)}$ con una transformación **lineal** ($J_l\, h$). El modelo, en cambio, es profundamente no lineal (atención, MLP, normalizaciones). La lens es la **mejor aproximación lineal** de cómo se propaga la información, pero no captura las no linealidades.

Esto significa que la lens puede perderse interacciones no lineales: si la activación de "euros" en L28 depende de una interacción compleja entre varias dimensiones del residual stream, la lens (que es lineal) puede no verla. La lens ve la parte lineal de la historia, no toda la historia.

**La analogía de la calle.** Imagina que el residual stream es un río con curvas, rápidos y remolinos (no linealidades). La lens es como un mapa que solo muestra la corriente principal (la parte lineal): te dice por dónde va el agua en promedio, pero no te muestra los remolinos. Si la respuesta correcta está en un remolino, la lens no la ve.

### 11.8. La elección de la posición importa

La lens se aplica en una **posición** concreta del prompt. Si eliges la posición equivocada, puedes perderte el hallazgo. En nuestro ejemplo de la moneda, la posición -1 (donde el modelo predice) es la que revela "euros" en L28. La posición -2 (el token "boot") revela "shoes" y "is", no "euros". Si solo hubiéramos mirado la posición -2, habríamos concluido erróneamente que `euros` no aparece en la lectura.

**Consejo práctico.** Aplica siempre la lens en varias posiciones, no solo en una. Como mínimo, en la posición -1 (donde el modelo predice) y en la posición -2 (el token anterior). Si el prompt tiene una estructura compleja, considera posiciones intermedias donde el modelo podría estar procesando información clave.

---

## 12. Conclusiones

La Jacobian lens es una herramienta notable por su simplicidad: sin etiquetas, sin entrenamiento supervisado, solo *forward* y *backward* sobre texto, consigue leer el residual stream de un LLM capa a capa. En esta pieza la hemos aplicado a gpt2-small y a Gemma 4 E2B, y hemos visto:

- **gpt2-small**, en esta reproducción, no hace aparecer `euro` / `euros` como candidato relevante para el prompt de la bota de Italia; la lectura capa a capa queda atrapada en continuaciones léxicas como `strap` / `loader`.
- **Gemma 4 E2B** muestra el patrón central de la pieza: en L28 la lens lee `currency`, `dollars`, `dollar`, `euros`, `called`; en la salida final, `called` y `the` quedan arriba con el mismo logit y `euros` desaparece del top-10.
- **Los ejemplos propios** muestran tres patrones: aparece y se verbaliza (capital de Japón), aparece pero no se verbaliza (moneda de Italia y 2+2=), y piezas sueltas que no se ensamblan al final (Don Quijote). La lens permite distinguir estos patrones sin adivinar solo desde la salida.
- El *fit* desde cero con 20 prompts ya produce una lens inspeccionable, aunque con diferencias grandes respecto a la preajustada. Más prompts reducen la variación; esta pieza no intenta fijar una ley universal de saturación.

- La visualización *slice-vis* permite explorar posiciones y capas en el navegador, aunque las conclusiones cuantitativas deben apoyarse en los rankings y salidas verificadas.

El hallazgo de la capa 28 ilustra la tesis del paper de Anthropic (2026): algunas representaciones intermedias son verbalizables aunque no siempre coinciden con la salida final. Para un ingeniero, eso significa que puede auditar dónde aparece una respuesta candidata y en qué tramo de capas se pierde o se redirige.

---

## Referencias


- Anthropic. (2026). *Verbalizable representations form a global workspace in language models*. Transformer Circuits. https://transformer-circuits.pub/2026/workspace/index.html
- Anthropic. (2026). *jacobian-lens* [software de investigación]. GitHub. https://github.com/anthropics/jacobian-lens
- Baars, B. J. (1988). *A cognitive theory of consciousness*. Cambridge University Press.
- Belrose, N., Furman, Z., Smith, L., Halawi, D., Ostrovsky, I., McKinney, L., Biderman, S., & Steinhardt, J. (2023). *Eliciting latent predictions from transformers with the tuned lens*. arXiv:2303.08112. https://arxiv.org/abs/2303.08112
- Dehaene, S. (2014). *Consciousness and the brain: Deciphering how the brain codes our thoughts*. Viking.
- Google. (2026). *google/gemma-4-E2B* [model card]. Hugging Face. https://huggingface.co/google/gemma-4-E2B
- Meng, K., Bau, D. A., Andonian, A., & Belinkov, Y. (2022). *Locating and editing factual associations in GPT*. Advances in Neural Information Processing Systems, 35. https://arxiv.org/abs/2202.05262
- nostalgebraist. (2020). *Interpreting GPT: the logit lens*. LessWrong. https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens