LensLapse: In-Browser Logit Lens for Interactive Exploration of Language Model Training Dynamics

Tatsuki Okada **Yusuke Sakai** **Taro Watanabe**
Nara Institute of Science and Technology (NAIST), Japan
okada.tatsuki.ou0@naist.ac.jp, {sakai.yusuke.sr9, taro}@is.naist.jp

Abstract

The logit lens is a standard mechanistic interpretability tool for inspecting a Transformer's predictions across layers. However, existing logit lens tools typically focus on individual checkpoints rather than interactive exploration of a model's *training history*. In this study, we introduce **LensLapse**, an open-source web application for interactive exploration of a model's training history through token-level interpretability. Users can scrub a slider across public checkpoints from five model families and three multilingual checkpoint suites and observe how per-layer, per-position logit lens predictions evolve from frequency bias to factual knowledge. For browser-compatible models, arbitrary user prompts are probed *live in the browser* via local inference over per-checkpoint model pairs. No server, no API key, no hosting cost, and prompts never leave the user's device. Moreover, **LensLapse** supports local models. A single `pip install` enables users to launch the application and explore their own models and training checkpoints without any additional web setup. LensLapse is MIT-licensed and permanently hosted at https://iamtatsuki05.github.io/lenslapse/.[1,2,3,4]

1 Introduction

Interpretability tools, e.g., the logit lens (nostalgebraist, 2020), almost exclusively analyze *trained* models, even though the phenomena they reveal, such as factual recall, frequency bias, and layer specialization, emerge *during training*. Checkpoint suites were introduced to facilitate this type of investigation: Pythia ships 154 intermediate checkpoints for each of 16 models trained on identically ordered data (Biderman et al., 2023), and OLMo (Groeneveld et al., 2024) and LLM360 (Liu et al., 2024) follow the same philosophy. However, in practice, publicly available tools for these checkpoint suites are typically limited to scalar loss curves and static output galleries, while inspecting token-level internal dynamics still requires custom analysis code. At the same time, interactive interpretability tools are particularly valuable for live demonstrations, exploratory analysis, and communicating model behavior. Hosted lens tools use server-side inference (Belrose et al., 2023; Geva et al., 2022; Tufanov et al., 2024), making their availability dependent on backend maintenance. Client-side execution avoids this requirement for browser-compatible models.

In this study, we introduce **LensLapse**, an open-source web application for interactive exploration of a model's training history through token-level interpretability. LensLapse addresses both challenges by combining browser-based inference for browser-compatible models with interactive exploration across training checkpoints. Figure 1 shows the interface: a log-scale slider navigates training checkpoints, a grid displays the top-1 lens prediction for every layer-position pair, and clicking a cell visualizes its probability trajectory throughout training. Curated prompts are rendered instantly from precomputed shards, while arbitrary user prompts are processed live on the user's own hardware. Moreover, it can be easily deployed on user-trained models, facilitating interactive analysis of model behavior throughout training.

The main features of **LensLapse** are as follows:

- An interactive hosted system for exploring training checkpoints through token-level interpretability over a real public checkpoint suite (three Pythia sizes; 38 precomputed and 20 live-probeable checkpoints each for 14M/70M).
- A fully client-side logit lens implementation, exact by construction because the lens over the

[1] : iamtatsuki05/lenslapse
[2] : pip install lenslapse
[3] : iamtatsuki05.github.io/lenslapse/docs
[4] : https://youtu.be/OBnvG9Ru3j8

Proceedings of the 15th International Joint Conference on Natural Language Processing and the 5th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, pages 66–73
November 6-10, 2026 ©2026 Association for Computational Linguistics

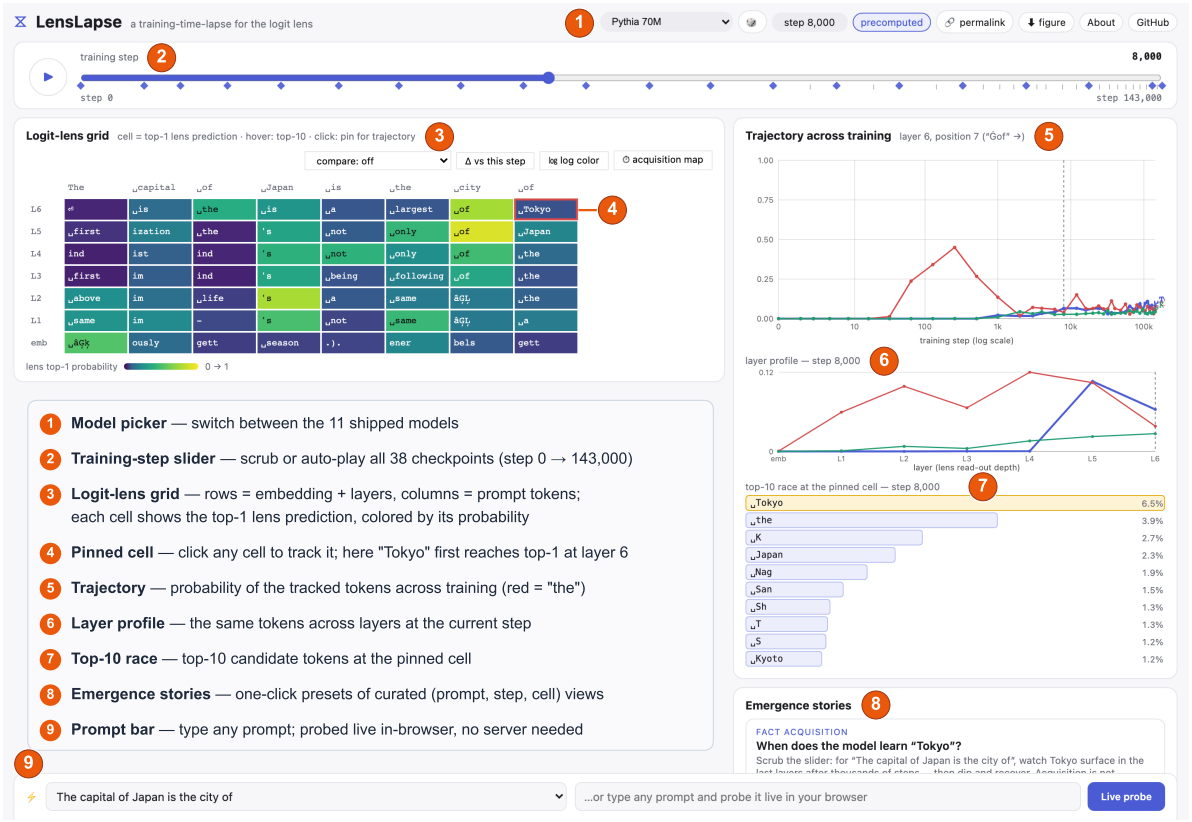


Figure 1: The LensLapse interface at training step 8,000 of Pythia-70M for the prompt *"The capital of Japan is the city of"*. Components: (1) model picker over the eleven shipped models; (2) training-step slider scrubbing the 38 checkpoints (step 0 → 143,000); (3) logit lens grid (rows = embedding + 6 layers, columns = token positions; each cell = top-1 lens prediction colored by probability); (4) pinned cell, where "Tokyo" first reaches top-1 at layer 6; (5) trajectory panel plotting exact probabilities across all 38 checkpoints, where the red curve is the frequency-bias token "the" spiking mid-training and decaying (dashed rule = current step); (6) layer profile at this step; (7) top-10 race at the pinned cell; (8) "emergence stories" preset views; and (9) prompt bar probing free text live in-browser.

last block's output reproduces the model's logits, asserted per checkpoint at export; the shipped fp16 format preserves 100% top-1 agreement with fp32 PyTorch whereas int8 does not (Section 4.1).

- A zero-backend architecture based on a static site, versioned weights on the Hugging Face Hub, and client-side computation, enabling zero hosting cost, no inference-server concurrency limit, no server maintenance, and on-device prompt processing.
- An open, architecture-agnostic conversion recipe (five decoder-norm families all pass its parity check) that can be applied to arbitrary public checkpoint suites, demonstrated on seven additional public models alongside Pythia and GPT-2: three multilingual suites (MAP-Neo, BAAI Aquila, BLOOM) and four architecture-coverage examples (SmolLM2, Qwen3, OPT, Gemma 3).

LensLapse targets interpretability and training-dynamics researchers browsing hypotheses before writing analysis code, instructors and students without GPU access, and checkpoint-suite maintainers seeking a reusable front-end for hosting their own models and building interactive demonstrations.

2 Background and Related Work

Table 1 compares LensLapse with similar interactive interpretability systems. Hosted logit lens tools inspect a single trained model using server-side inference (Belrose et al., 2023; Geva et al., 2022; Tufanov et al., 2024), while Neuronpedia (Lin, 2023) provides a rich hosted platform for Sparse Autoencoder (SAE) features and circuits for trained models. Transformer Explainer (Cho et al., 2026) introduced in-browser GPT-2 inference for education but does not expose layer-wise vocabulary predictions. BuildAnLLM (Bhalgat, 2025) provides a local checkpoint viewer for user-trained models. OLMoTrace (Liu et al., 2025) traces outputs back

System	Lens-style layer view	Training-time axis	Client-side inference
Tuned Lens Space (Belrose et al., 2023)	✓	✗	✗
LM-Debugger (Geva et al., 2022)	✓	✗	✗
LM Transparency Tool (Tufanov et al., 2024)	✓	✗	✗
Neuronpedia (Lin, 2023)	SAE features	✗	✗
Transformer Explainer (Cho et al., 2026)	✗	✗	✓
BuildAnLLM (Bhalgat, 2025)	✓	✓[a]	✗
LLM360 K2 gallery (Liu et al., 2024)	✗	✓	✗
OLMoTrace (Liu et al., 2025)	✗	✗[b]	✗
LensLapse (ours)	✓	✓	✓

Table 1: Comparison with interactive interpretability systems. Note that this feature comparison is not a runtime or usability benchmark. Neuronpedia provides SAE feature and circuit views. [a]Supports only local checkpoints. [b]Traces outputs to training *data*, rather than across training *time*.

to training *data*, complementing our focus on training *time*. Recent work tracks representation formation throughout pretraining using crosscoders (Bayazit et al., 2026; Ge et al., 2026). LensLapse complements these methods by providing browser-based logit-lens visualization across checkpoints, without implementing crosscoder analysis.

3 LensLapse: System Description

3.1 Interface

As shown in Figure 1, LensLapse provides a single-page interface organized with three coupled views:

Lens grid renders the top-1 lens prediction of the selected checkpoint for every (layer, position) cell on a canvas, colored by probability; hovering reveals top-10 distribution, and clicking pins a cell.

Training-step slider is log-scaled with one tick for each precomputed checkpoint; diamonds indicate live-probeable checkpoints. Scrubbing the slider, or using one-click playback, animates the grid, with cells flashing as their top-1 predictions change. An *acquisition-map* view recolors the grid according to when each cell first predicts its final answer, while a *diff* view highlights changes relative to a fixed reference checkpoint.

Trajectory panel plots, for the pinned cell, the exact probabilities across every checkpoint for the tokens to which the final model converges (the final top-3 plus the gold continuation; typed tokens can be added to the live-traced set). It also displays the current checkpoint as a movable rule, together with an animated top-10 bar race and the classic p-vs-layer profile. Curated *emergence stories* jump to preset (prompt, checkpoint, pinned-cell) configurations for first-time users. The full application state is encoded in the URL hash, making every view a shareable permalink, while one-click export saves the current view as a publication-ready PNG or PDF figure. A model picker switches among all eleven shipped models, each labeled with its documented language support. Grid depth, checkpoint list, and live-probeable checkpoints adapt automatically, and a compare mode renders a second model's grid in lockstep for side-by-side comparison across model scales.

3.2 Efficient Hosting and Data Flow

We support two complementary execution modes. For common use cases, we provide precomputed prompts and hosted models optimized for efficient browser-based execution. For larger models or user-trained checkpoints, we also provide an easy deployment pipeline, enabling the same interactive interface through a lightweight local backend.

Precomputed path. For 16 curated prompts, or 26 for the multilingual-capable models, we precompute the full lens output using fp32 PyTorch over 38 checkpoints (20 for the 160M model) and serve the results as static JSON shards. Each shard contains the top-10 tokens for every cell, together with exact probability and rank trajectories for position-wise target tokens. This enables instant visualization without downloading model weights; the grid renders in under one second on the first visit.

Live path. Any free-text prompt triggers a forward pass, without autoregressive generation or KV caching, through every registered checkpoint rather than only the selected one, allowing the same time-lapse playback available for curated prompts. Model files are fetched once from a public Hugging Face repository, cached via the browser Cache API, and executed with ONNX Runtime Web (ONNX Runtime developers, 2021), preferring WebGPU and falling back to WASM or, if model files are un-

available, to a precomputed-only mode with a visible status badge. Tokenization uses Transformers.js (Hugging Face, 2024). All (layer, position) hidden states are passed through the lens head using a single batched matrix multiplication per checkpoint, while softmax and top-10 extraction are performed in JavaScript. Because inference runs entirely on the client, user prompts never leave the device. We use ONNX and JavaScript rather than PyTorch-based TransformerLens, nnsight, or tuned-lens to avoid requiring a Python backend in this path.

Heavy models and reproducible probes. We benchmark browser inference only for Pythia-14M–160M. We currently provide off-the-shelf support for models up to Qwen3-0.6B. The maximum browser-compatible size and the first limiting factor (download size, memory, or latency) are not considered. When browser resources are insufficient, precomputed views or a probe server are needed. MAP-Neo and Aquila also need the server for free-text tokenization. For local use, a single `pip install lenslapse` provides the application and probe API through a single local port, without CORS configuration, and supports offline use once models are cached. The interface registers Hub models and local runs persistently without export, supports background ONNX conversion, and detects a local server when available. The server uses Hugging Face Transformers (Wolf et al., 2020) and the precomputed pipeline's hooked forward pass to obtain all layers' residual-stream states. A *trace-across-training* action probes every registered checkpoint for trajectory visualization. UI actions are scriptable through the `lenslapse` CLI (`probe`, `trace`, and `models`). Every live result is cached in IndexedDB (browser) and a keyed JSON cache (server), enabling byte-for-byte replay across sessions. Appendix A provides usage examples.

3.3 Zero-Backend Deployment

The site is a static bundle on GitHub Pages[5] and model weights live in a public Hugging Face repository and are fetched directly by the browser. Nothing executes server-side, so hosting is free, capacity scales with the number of visitors' devices, and there is no backend to maintain. Therefore, once deployed, the application can remain available without server maintenance.

[5]The site is up to ∼250MB, with tokenizer files for the multilingual models and precomputed shards accounting for most of the size, well below the GitHub Pages 1GB limit.

Bring your own suite. Adding a model requires only data preparation rather than code modification. A single onboarding command (`lenslapse add-model`) exports, parity-checks, precomputes, and registers a Hugging Face checkpoint suite, a single Hub model, or a local directory of Trainer checkpoints, including user-trained models. The resulting ONNX models can then be deployed on any static hosting service. The included GitHub Actions workflow enables end-to-end deployment with no hosting cost (see `docs/deployment.md`). The full pipeline, e.g., conversion, precomputation, fidelity evaluation, and the benchmark harness, is released under MIT license in the same repository.

Blog-style interactive documents. Because precomputed results are stored as static JSON files, the same framework can be readily adapted to arbitrary datasets and models. Even for models too large for browser inference, interactive visualizations can be embedded as ***precomputed path***. This makes it practical to publish interactive analyses alongside papers, blogs, documentation, or entire datasets, improving accessibility, reproducibility, and long-term availability of research artifacts.

4 Evaluation

4.1 Numerical Fidelity

Table 2 compares different weight formats against fp32 PyTorch ground truth over all (layer, position) cells for 16 curated prompts at three checkpoints. Dynamic int8 quantization, a common choice for browser deployment, is unsuitable in this setting: final-layer top-1 agreement drops to 52% (per-tensor) or 71% (per-channel) at step 143k, and the degradation increases throughout training as weight outliers emerge. Consequently, quantization errors are largest at later checkpoints, where inspection is often most informative. In contrast, storing weights in fp16 while computing in fp32 preserves 100% top-1 agreement on this benchmark while reducing storage by half relative to fp32, and is therefore the default configuration in LensLapse.

In the production browser implementation (WebGPU), live probing matches the fp32 PyTorch precomputed results on all 56 evaluated cells, with a maximum absolute probability difference of 5.5×10^{-4}. The ONNX-free probe server (fp32 by default) also achieves identical top-1 predictions on all 56 cells. By comparison, an fp16 server-side forward pass already flips 5 of the 56 predictions at step 143k, and lower-precision execution

Format	Top-1 agreement (all / final layer) step 8k	step 64k	step 143k	MB
fp32 ONNX	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00	283
int8 (tensor)	.90 / .88	.73 / .69	.61 / **.52**	71
int8 (channel)	.93 / .95	.78 / .85	.74 / .71	72
fp16 (ours)	**1.00 / 1.00**	**1.00 / 1.00**	**1.00 / 1.00**	142

Table 2: Lens fidelity vs. fp32 PyTorch on Pythia-70M (16 prompts; 861 cells per checkpoint). Mean final-layer KL at step 143k: 2×10^{-5} (fp32, fp16), 0.18 (int8 channel), 0.60 (int8 tensor). The fp32 size includes the backbone's external weight file.

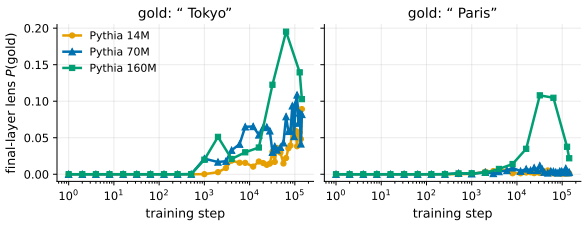


Figure 2: Scale × time in one view: exact final-layer lens probability of the gold continuation across training for all three shipped Pythia models, read from the precomputed shards (markers = checkpoints, log-scale steps). Left: "Tokyo" is acquired by every size, non-monotonically. Right: "Paris" is acquired only by 160M, a capability threshold, and later decays.

is therefore supported only as an explicit opt-in. In addition, capturing raw block outputs avoids double normalization of the final hidden state. Further details are provided in Appendix B. However, numerical fidelity does not establish interpretive reliability: intermediate-layer predictions can be brittle (Belrose et al., 2023). Tuned-lens support and cross-family validation of interpretive reliability remain future work.

4.2 Performance across Device Settings

Table 3 reports latency across browser engines, execution providers, emulated CPU throttling, model sizes, and prompt lengths, measured using our scripted Playwright harness. A *probe* measures the end-to-end latency from pressing "Live probe" to rendering the results, including a single forward pass, lens-head computation over all cells, and JavaScript softmax/top-10 extraction. Warm probes for 8-token prompts remain interactive ($\leq$320 ms) on unthrottled Chromium and WebKit across all three model sizes. Firefox using the WASM backend is the only notable outlier ($\sim$1.4 s, about 11× slower than Chromium-WASM), reflecting differences in ONNX Runtime's WASM implementation. WASM latency scales approximately linearly with model size (66/127/316 ms warm for 14M/70M/160M), whereas WebGPU shows sublinear scaling (63/70/134 ms). Under 4× CPU throttling (DevTools emulation of a low-end laptop), WASM slows by about 4.2× and WebGPU by about 3.3×. Because top-10 extraction is performed on the CPU, even the WebGPU pipeline retains a CPU-bound component.

Browser	EP	Model	Thr.	Load c/w	Probe f/w
Chromium	WebGPU	70M	1×	1052 / 295	157 / **70**
Chromium	WASM	70M	1×	935 / 230	155 / **127**
Chromium	WebGPU	70M	4×	3224 / 1066	488 / **233**
Chromium	WASM	70M	4×	2821 / 888	632 / **530**
Chromium	WebGPU	14M	1×	566 / 106	149 / **63**
Chromium	WASM	14M	1×	509 / 84	94 / **66**
Chromium	WebGPU	160M	1×	1799 / 590	237 / **134**
Chromium	WASM	160M	1×	1610 / 489	352 / **316**
Firefox	WASM	70M	1×	2404 / 2850	2320 / **1399**
WebKit	WASM	70M	1×	644 / 203	160 / **125**
Chromium	WebGPU	70M†	1×	1045 / 279	310 / **205**
Chromium	WASM	70M†	1×	916 / 231	509 / **453**

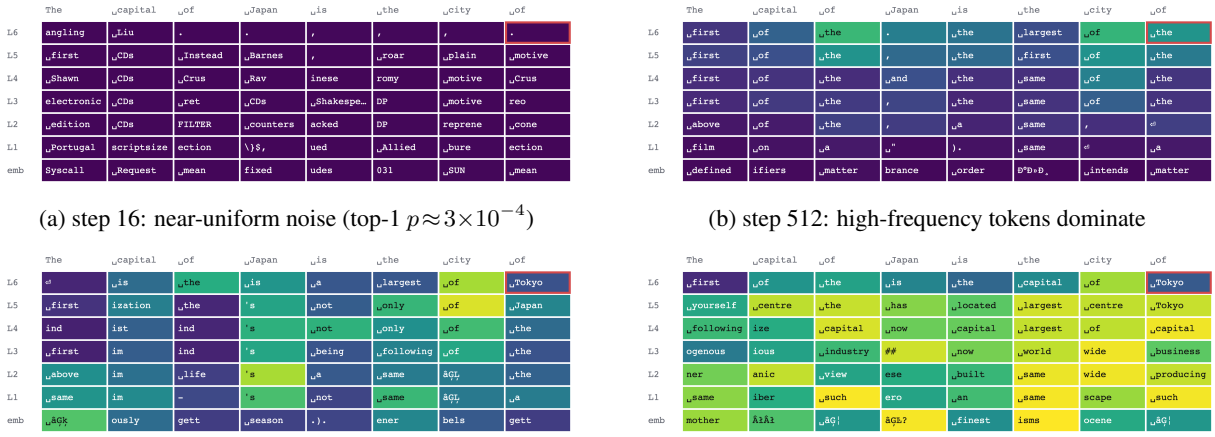
Table 3: Latency in ms on an M1 Max (2026-07-15). Load = checkpoint fetch (local host) + session init, cold / Cache-API-warm. Probe = forward + lens + top-10, first / warm mean of 5. Thr. = emulated CPU throttling. †with a 30-token prompt (all other rows use 8-token prompts).

4.3 Case Studies

Figure 3 shows four representative time-lapse snapshots. All observations below are obtained directly from LensLapse and are reproducible via permalinks. Unless otherwise noted, probabilities refer to the final-layer lens at the last prompt position of Pythia-70M.

Fact acquisition is visible and non-monotonic. For *"The capital of Japan is the city of"*, P("Tokyo") remains near zero until approximately step 1k and first reaches top-1 at step 8k. However, the trajectory is far from monotonic: across the 38 checkpoints, Tokyo is top-1 at only 10 checkpoints, losing the position throughout most of steps 12k–56k (bottoming at P=0.030 at step 32k) before recovering to P=0.082 at step 143k. In addition, Figure 4 shows the trajectory panel. It also shows P("the") peaking during mid-training before declining, indicating competition between frequency bias and factual knowledge.

Early frequency bias. At step 16 the lens distribution is nearly uniform (top-1 $p \approx 3\times10^{-4}$). By step 64, the five most frequent predictions (",",



(a) step 16: near-uniform noise (top-1 $p \approx 3 \times 10^{-4}$)

(b) step 512: high-frequency tokens dominate

(c) step 8,000: "Tokyo" first reaches top-1 (L6)

(d) step 143,000: facts settled in the deep layers

Figure 3: The time-lapse: the lens grid of Pythia-70M for *"The capital of Japan is the city of"* at four training steps (rows = embedding + layers bottom-up, columns = token positions, color = top-1 probability; the outlined cell is the final position at layer 6). Scrubbing the slider animates this progression continuously over 38 checkpoints.

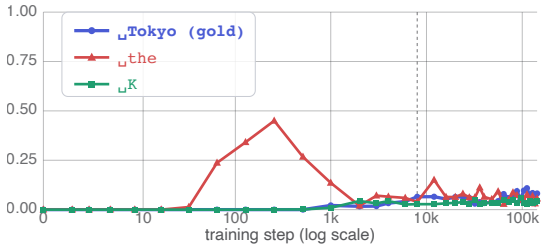


Figure 4: Exact lens-probability trajectories for the pinned cell of Figure 3 across all 38 checkpoints: "the" (red) spikes mid-training and decays, "Tokyo" rises late and non-monotonically (dashed rule = selected step).

"the", ".", newline, and "and") account for 67% of all 861 grid cells in the 16-prompt suite. By step 512, this top-5 share falls to 36%, before more position-specific predictions emerge.

Layer specialization emerges gradually. At step 143k, "Tokyo" is top-1 only at layers 5–6, while lower layers predict locally plausible yet incorrect continuations ("capital" at L4 and "business" at L3). Earlier checkpoints show even the deepest layers have not yet acquired the fact.

Browsing acquisition order. Reading first-acquisition steps from the gold continuations reveals a consistent progression: code completion first (*"return a +"* → "b": step 1k), followed by idioms and agreement ("much": 2k, "time": 3k), and finally factual knowledge ("Tokyo": 8k, "Washington": 32k). This progression from syntax to semantics to factual knowledge is observable within a single interface.

Learned, then lost. "oxygen" (*"Water is made of hydrogen and"*) becomes top-1 at step 8k but loses the position by step 143k. "Washington" exhibits a similar pattern, illustrating that knowledge acquisition is not monotonic. LensLapse makes such knowledge-drift patterns straightforward to inspect interactively.

Scale and training time in one view. Figure 2 illustrates how model scale interacts with training dynamics. The more difficult fact *"The Eiffel Tower is located in the city of"* → "Paris" is never acquired as top-1 by the 14M model (final P=0.0012) or the 70M model (P=0.0028), whereas the 160M model reaches top-1 at step 32k before falling to rank 3 by step 143k (P=0.022), exhibiting another learned-then-lost trajectory. In contrast, "Tokyo" is acquired by all three model sizes, albeit at different and similarly non-monotonic stages of training (14M: 72k, 70M: 8k, 160M: 32k).

5 Conclusion

In this study, we introduced **LensLapse**, an open-source web application for interactive exploration of training dynamics through token-level interpretability. LensLapse provides high-fidelity browser-based visualization of logit lens predictions, supports user-trained models through a lightweight local probe server, and enables sustainable zero-backend deployment. Our experiments demonstrate that LensLapse enables interactive inspection of diverse training phenomena. We hope it will facilitate research, education, and interactive dissemination of language model interpretability.

References

Loubna Ben allal, Anton Lozhkov, Elie Bakouch, Gabriel Martin Blazquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Agustín Piqueres Lajarín, Hynek Kydlíček, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan Son NGUYEN, Ben Burtenshaw, Clémentine Fourrier, Haojun Zhao, Hugo Larcher, Mathieu Morlon, Cyril Zakka, and 3 others. 2025. SmolLM2: When smol goes big — data-centric training of a fully open small language model. In *Second Conference on Language Modeling*.

Deniz Bayazit, Aaron Mueller, and Antoine Bosselut. 2026. Crosscoding through time: Tracking emergence & consolidation of linguistic representations throughout LLM pretraining. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1353–1377, San Diego, California, United States. Association for Computational Linguistics.

Nora Belrose, Zach Furman, Logan Smith, Danny Halawi, Igor Ostrovsky, Lev McKinney, Stella Biderman, and Jacob Steinhardt. 2023. Eliciting latent predictions from transformers with the tuned lens. *CoRR*, abs/2303.08112.

Girish Bhalgat. 2025. BuildAnLLM: Build, train and inspect a language model locally. https://github.com/jammastergirish/BuildAnLLM.

Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar Van Der Wal. 2023. Pythia: a suite for analyzing large language models across training and scaling. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Aeree Cho, Grace C Kim, Alexander Karpekov, Seongmin Lee, Alec Helbling, Benjamin Hoover, Zijie J Wang, Minsuk Kahng, and Duen Horng Chau. 2026. Transformer explainer: Learning llm transformers with interactive visual explanation and experimentation. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, pages 1–21.

Xuyang Ge, Wentao Shu, Jiaxing Wu, Yunhua Zhou, Zhengfu He, and Xipeng Qiu. 2026. Evolution of concepts in language model pre-training. In *The Fourteenth International Conference on Learning Representations*.

Mor Geva, Avi Caciularu, Guy Dar, Paul Roit, Shoval Sadde, Micah Shlain, Bar Tamir, and Yoav Goldberg. 2022. LM-debugger: An interactive tool for inspection and intervention in transformer-based language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 12–21, Abu Dhabi, UAE. Association for Computational Linguistics.

Dirk Groeneveld, Iz Beltagy, Evan Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, and 24 others. 2024. OLMo: Accelerating the science of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15789–15809, Bangkok, Thailand. Association for Computational Linguistics.

Hugging Face. 2024. Transformers.js: State-of-the-art machine learning for the web. https://github.com/huggingface/transformers.js.

Johnny Lin. 2023. Neuronpedia: Interactive reference and tooling for analyzing neural networks. Software available from neuronpedia.org.

Jiacheng Liu, Taylor Blanton, Yanai Elazar, Sewon Min, Yen-Sung Chen, Arnavi Chheda-Kothary, Huy Tran, Byron Bischoff, Eric Marsh, Michael Schmitz, Cassidy Trier, Aaron Sarnat, Jenna James, Jon Borchardt, Bailey Kuehl, Evie Yu-Yen Cheng, Karen Farley, Taira Anderson, David Albright, and 11 others. 2025. OLMoTrace: Tracing language model outputs back to trillions of training tokens. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 178–188, Vienna, Austria. Association for Computational Linguistics.

Zhengzhong Liu, Aurick Qiao, Willie Neiswanger, Hongyi Wang, Bowen Tan, Tianhua Tao, Junbo Li, Yuqi Wang, Suqi Sun, Omkar Pangarkar, Richard Fan, Yi Gu, Victor Miller, Yonghao Zhuang, Guowei He, Haonan Li, Fajri Koto, Liping Tang, Nikhil Ranjan, and 8 others. 2024. LLM360: Towards fully transparent open-source LLMs. In *First Conference on Language Modeling*.

nostalgebraist. 2020. interpreting GPT: the logit lens. LessWrong blog post.

ONNX Runtime developers. 2021. ONNX runtime. https://onnxruntime.ai/.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Igor Tufanov, Karen Hambardzumyan, Javier Ferrando, and Elena Voita. 2024. LM transparency tool: Interactive tool for analyzing transformer language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 51–60, Bangkok, Thailand. Association for Computational Linguistics.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara

Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, and 3 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

A Installation and Deployment

Listing 1 shows the complete local setup. A single `pip install` and one command launch both the web application and the live probe API on the same port, without requiring ONNX conversion or configuration files. Listing 2 shows how to publish a permanent, zero-cost deployment with user-trained models using the same `add-model` onboarding command described in Section 3.

B Conversion Recipe

Each checkpoint is exported as a pair of ONNX graphs. The **backbone** maps token ids to the residual stream after the embedding and after each transformer block, $[L{+}1, T, H]$. A subtlety makes correctness non-trivial: Hugging Face Transformers applies the final layer norm to the last entry of `output_hidden_states`, so naively lensing it double-normalizes (we measured a maximum logit error of 0.59). We instead capture each block's raw output with forward hooks, giving pre-norm states uniformly. The **lens head** is a separate three-node graph (final norm + unembedding) applicable to any layer's hidden state; splitting it from the backbone avoids duplicating the unembedding matrix. This yields an exactness identity, $\mathrm{lens}(h_L) \equiv$ model logits, which the export pipeline asserts per checkpoint: the difference is exactly 0.0 in PyTorch, and the exported models preserve top-1 at every probe position, with a maximum logit deviation of $1.1{\times}10^{-2}$ across the 60 Pythia fp16 checkpoints; the GPT-2 export passes the same assertions.

Weights are stored as fp16 initializers with a runtime cast that ONNX Runtime constant-folds at session load, halving file size (29/142/326 MB per checkpoint for 14M/70M/160M) while keeping computation in fp32. The recipe is architecture-generic: it introspects the decoder stack, the final norm (LayerNorm or RMSNorm), and the unembedding, and a parity checker asserts the lens identity per model. GPT-NeoX (Pythia, PyTorch diff 0.0), GPT-2-style LayerNorm (GPT-2 (Radford et al., 2019) with tied embeddings; BLOOM's ALiBi variant), Llama-style RMSNorm (SmolLM2-135M, 30 layers, allal et al., 2025; MAP-Neo-250M), Mistral-style RMSNorm (Aquila-135M; Qwen3-0.6B), and Gemma-style plus-one-weight RMSNorm (Gemma-3-270M) all pass; unsupported (e.g., post-norm) layouts fail loudly at export rather than shipping a wrong lens. A layout the generic attribute-path heuristic cannot reach, OPT's decoder nests two hops deeper than the others, registers an explicit override for that architecture's own `config.model_type` in one line, replacing the heuristic entirely for it rather than blending with it, so a wrong registration also fails loudly rather than silently guessing.

```
pip install lenslapse
lenslapse server
# Serving app + probe API at
  http://localhost:8017/
# Opening in your default browser...

# In the header: click the gear icon, enter
  a Hugging Face model id
# (or browse to a local training-checkpoint
  folder), and probe it live --
# no ONNX export, no config files.
```

Listing 1: Installing and probing your own model locally: one command serves the app and its probe API on the same port.

```
gh repo fork iamtatsuki05/lenslapse --clone
  && cd lenslapse

uv run lenslapse add-model --model
  <your-hf-model-or-local-checkpoint-dir> \
  --id my-model --label "My Model"
    --models-root ./onnx-out
# -> exports + parity-checks the ONNX pair,
  precomputes the lens shards,
#    and registers "my-model" in
  web/public/data/models.json
#    (a data change, not a code change).

hf upload-large-folder <you>/my-model-onnx
  --repo-type model ./onnx-out
# then point HF_DEFAULT in web/src/live.ts
  at <you>/my-model-onnx

git push
# One-time only: Settings -> Pages ->
  Source: GitHub Actions.
# -> deploy-pages.yml builds web/ (now
  including "my-model") and publishes
#    it live at
  https://<you>.github.io/lenslapse/ -- no
  server to run.
```

Listing 2: Deploying your own permanent, zero-cost copy of the site with your own model added.