CLT-Forge: A Scalable Library for Cross-Layer Transcoders and Attribution Graphs

Florent Draye[1], Vedant Palit[1,2,4], Abir Harrasse[2], Tung-Yu Wu[2], Jiarui Liu[2,3], Punya Syon Pandey[2,4], Roderick Wu[2], Chih-Hao Hsu[2], Terry Jingchen Zhang[2,4] Zhijing Jin[1,2,4], Bernhard Schölkopf[1,5]

[1]Max Planck Institute for Intelligent Systems, Tübingen, Germany
[2]Jinesis AI Lab, University of Toronto & Vector Institute [3]CMU
[4]EuroSafeAI [5]ELLIS Institute Tübingen

fdraye@tue.mpg.de

Abstract

Mechanistic interpretability seeks to understand how Large Language Models (LLMs) represent and process information. Recent approaches based on dictionary learning and transcoders enable representing model computation in terms of sparse, interpretable features and their interactions, giving rise to feature attribution graphs. However, these graphs are often large and redundant, limiting their interpretability in practice. Cross-Layer Transcoders (CLTs) address this issue by sharing features across layers while preserving layer-specific decoding, yielding more compact representations, but remain difficult to train and analyze at scale. We introduce an open-source library for end-to-end training and interpretability of CLTs. Our framework integrates scalable distributed training with model sharding and compressed activation caching, a unified automated interpretability pipeline for feature analysis and explanation, attribution graph computation using Circuit-Tracer, and a flexible visualization interface. This provides a practical and unified solution for scaling CLT-based mechanistic interpretability. Our code is available at https://github.com/LLM-Interp/CLT-Forge.

1 Introduction

Understanding how Large Language Models (LLMs) represent and process information is a central goal of mechanistic interpretability. A key assumption in this area is the *linear representation hypothesis*, which posits that semantic features are encoded as directions in the activation space of neural networks (Park et al., 2023). Building on this idea, *transcoders* approximate the computation of MLP layers by replacing their dense nonlinear transformations with sparse, linear combinations of interpretable features computed from the same inputs (Dunefsky et al., 2024). Since MLPs constitute a primary source of nonlinearity in transformer architectures, this substitution enables a structured view of model computation in terms of feature interactions. In particular, computation can be represented as a feature *attribution graph*, where nodes correspond to features and edges capture causal influence through linear pathways (Ameisen et al., 2025; Lindsey et al., 2025b). Such graphs can be efficiently computed, pruned, analyzed, and intervened upon to yield mechanistic explanations to the internal working mechanism of LLMs (Hanna et al., 2025).

However, a major limitation of this approach is that the size of attribution graphs can quickly grow to hundreds of nodes, making them difficult to analyze in practice. This redundancy stems in part from the fact that similar features often emerge across multiple layers, leading to duplicated nodes that represent the same underlying concept in different forms.

Cross-Layer Transcoders (CLTs) address this issue by sharing features across layers while allowing layer-specific decoding (Ameisen et al., 2025). This reduces redundancy and produces much more compact and interpretable attribution graphs (Lindsey et al., 2025a). However, CLTs introduce significant computational overhead in training due to their large parameter count.

Despite recent progress, there are limited open-source frameworks that support end-to-end training of CLTs at scale, where industry labs such as Anthropic do not release their proprietary framework. Most open-source codebases largely lack essential components such as efficient distributed training with optimal model sharding, or activation caching with quantization and compression for memory efficiency. Furthermore, post-hoc analysis remains a major bottleneck: interpreting millions of features requires retrieving activating data, computing statistics, and generating explanations (e.g., via automated interpretability), yet existing tools are fragmented and poorly integrated. Visualization

Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pages 850–858
October 24-29, 2026 ©2026 Association for Computational Linguistics

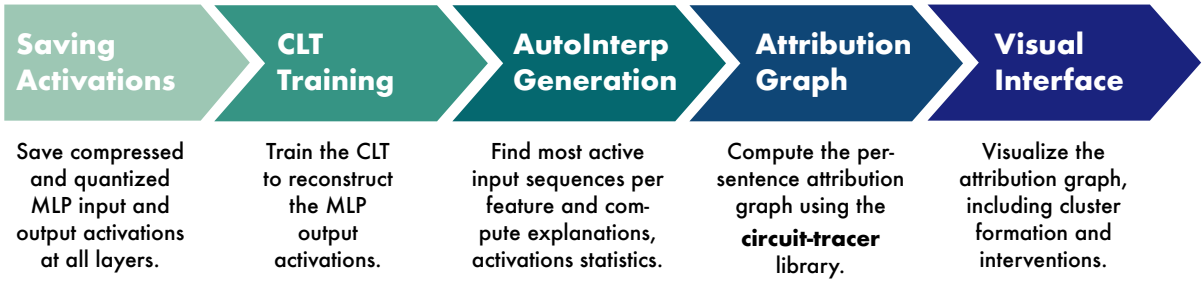

Figure 1: Overview of the CLT-Forge framework.

is similarly limited; while tools such as Neuronpedia (Lin and Bloom) provide very useful interfaces, they remain difficult to extend and integrate into research workflows.

In this work, we introduce the first unified library for CLTs that enables scalable training and post-hoc interpretability analysis. Our contributions are:

- **Scalable training infrastructure**: Open-source support for large-scale CLT training with efficient GPU sharding and optimized activation caching with compression and quantization.
- **Efficient attribution computation**: Native integration with Circuit Tracer (Hanna et al., 2025) to efficiently compute and prune feature-level attribution graphs.
- **Built-in workflow for automated interpretability (autointerp) with flexible visual interface**: A unified and scalable autointerp pipeline with a visual interface for exploring features, visualizing attribution graphs, and performing interventions.

The code is released under the MIT License and a demonstration video is available at `https://youtu.be/6ptrrLawTl8`.

2 Preliminary

2.1 Cross-Layer Transcoders

Let $h_\ell \in \mathbb{R}^{d_{\text{model}}}$ denote the input to the MLP at layer ℓ for a single token position. In both standard and cross-layer transcoders, this representation is first projected into a sparse feature space via an encoder:

$$z_\ell = \sigma\left(W_{\text{enc}}^{\ell} h_\ell + b_{\text{enc}}^{\ell}\right) \in \mathbb{R}^{d_{\text{features}}}, \qquad (1)$$

where $W_{\text{enc}}^{\ell} \in \mathbb{R}^{d_{\text{features}} \times d_{\text{model}}}$ and $b_{\text{enc}}^{\ell} \in \mathbb{R}^{d_{\text{features}}}$ are layer-specific parameters, and σ denotes a sparsifying nonlinearity. In practice, we use JumpReLU as σ. In a standard transcoder, the MLP output at layer ℓ is reconstructed by decoding features from the same layer:

$$\hat{m}_\ell = W_{\text{dec}}^{\ell} z_\ell + b_{\text{dec}}^{\ell}, \qquad (2)$$

where $W_{\text{dec}}^{\ell} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{features}}}$ and $b_{\text{dec}}^{\ell} \in \mathbb{R}^{d_{\text{model}}}$.

Cross-Layer Transcoders (CLTs) extend this formulation by allowing features extracted at earlier layers to contribute to the reconstruction at later layers. The MLP output at a target layer ℓ' is reconstructed as:

$$\hat{m}_{\ell'} = \sum_{\ell \leq \ell'} W_{\text{dec}}^{\ell \to \ell'} z_\ell + b_{\text{dec}}^{\ell'}, \qquad (3)$$

where $W_{\text{dec}}^{\ell \to \ell'} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{features}}}$ maps features from layer ℓ to layer ℓ'.

This formulation reflects the intuition that semantic features can persist across layers while undergoing transformations. For a detailed comparison between cross-layer transcoders and standard transcoders, we refer the reader to (Lindsey et al., 2025a).

Cross-Layer Transcoders were introduced by Anthropic (Ameisen et al., 2025; Lindsey et al., 2025b), together with their training, attribution, intervention, and visualization frameworks. More recently, open-source efforts have focused on reproducing and systematically evaluating CLTs against standard transcoders (Lindsey et al., 2025a). Despite these advances, applications of CLTs in the literature remain limited. Existing studies include a replication of the "greater-than" mechanism (Merullo et al., 2025) and a multilingual analysis (Harrasse et al., 2025), highlighting both the promise of CLTs and the need for more accessible and scalable infrastructure.

One existing open-source library for large-scale CLT training is Eleuther AI's *CLT-Training* fork of Sparsify (EleutherAI, 2024), which relies on on-the-fly activation computation. Additionally, their

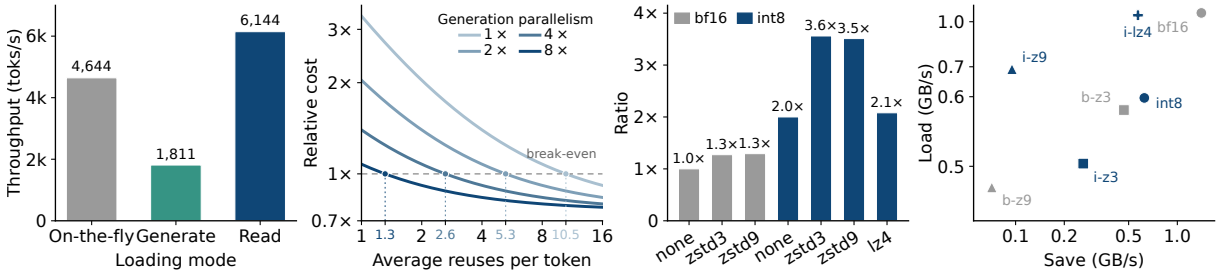

Figure 2: **Activation caching and compression trade-offs. Left pair:** Activation loading throughput for on-the-fly computation, cache generation, and cached reads. Cached reads achieve the highest throughput, while cache generation incurs a one-time serialization cost. Relative training time of caching compared to on-the-fly computation as a function of the average number of reuses per activation. Curves correspond to different levels of parallel cache generation since caching can easily be parallelized, while vertical markers indicate the break-even reuse count where caching becomes faster than recomputing activations. **Right pair:** Compression ratio achieved by different storage formats and compression methods, together with their corresponding save and load bandwidths. Int8 quantization reduces storage by up to $3.6\times$ while introducing a 12.5% reconstruction error.

work focuses on *TopK CLTs*, where only a small subset of features is active during decoding. In contrast, we do not impose activation sparsity, precluding the use of specialized kernels. Another library (Tigges, 2025) also supports feature sharding along the feature dimension with flexible activation functions. However, it does not provide a unified framework integrating scalable training with automated interpretability, attribution analysis, and interactive visualization. Other libraries support CLT training for smaller-scale models (e.g., GPT-2) using Distributed Data Parallel (DDP) (Tredal, 2025; Lange, 2025).

2.2 Attribution Graph

Following (Ameisen et al., 2025), the attribution score between feature n at layer ℓ and position k, and feature n' at layer ℓ' and position k' is:

$$a_{\ell,k,n}^{\ell',k',n'} = f_{k,n}^{\ell} \, J_{\ell,k}^{\ell',k'} \, g_{k',n'}^{\ell'}. \tag{4}$$

Here, $f_{k,n}^{\ell}$ denotes the decoder vector associated with feature n at layer ℓ and position k, while $g_{k',n'}^{\ell'}$ denotes the corresponding encoder vector for feature n' at layer ℓ' and position k'. The term $J_{\ell,k}^{\ell',k'}$ is the Jacobian mapping the MLP output at (ℓ, k) to the MLP input at (ℓ', k'). This Jacobian is computed during a forward pass of the model, where nonlinearities have been frozen with a stop-gradient operation. See (Ameisen et al., 2025) for more details. This workflow has been open-sourced in the Circuit-Tracer library (Hanna et al., 2025).

2.3 Autointerp Workflow and Visual Interface

Once a dictionary learning model, such as an SAE or a CLT, has been trained, interpreting its features typically requires identifying highly activating sequences, computing feature statistics, and generating natural language explanations, often using LLMs (Bricken et al., 2023).

Several libraries support parts of this workflow. Delphi (Paulo et al., 2024) focuses on automated interpretability, McDougall (2024) provides feature visualization, and Neuronpedia offers a comprehensive interface for feature exploration and attribution analysis (Lin and Bloom). However, these tools address different stages of the workflow and are not tightly integrated, often requiring practitioners to combine multiple libraries. This motivates a unified framework that integrates scalable automated interpretability, attribution analysis, and an extensible visualization interface.

3 CLT-Forge: Design and features

We describe in the following subsections the different components of the pipeline as described in Figure 1.

3.1 Activation Caching

Training a CLT requires reconstructing MLP outputs from their corresponding inputs, which in turn necessitates access to activations at all layers over large-scale datasets (e.g., hundreds of millions of tokens). We support two modes for handling activations. First, activations can be computed *on the fly*, which avoids storing large activation datasets and is useful in disk memory-constrained settings

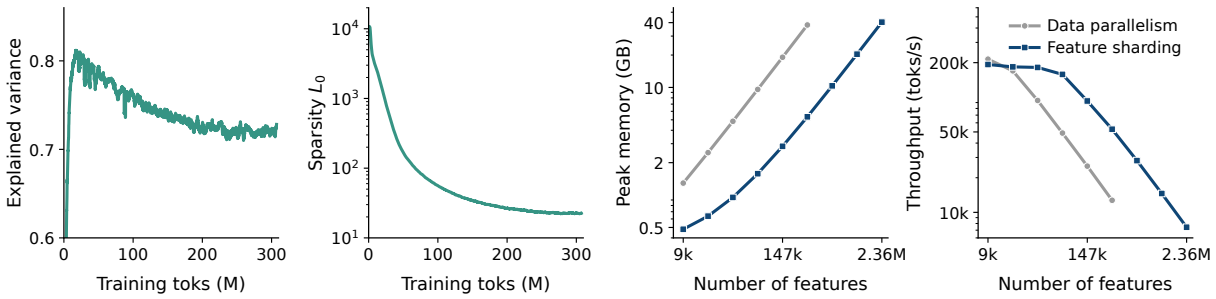

Figure 3: **CLT-Forge training dynamics and scalability. Left pair:** Training dynamics of a Cross-Layer Transcoder on Llama-3.2-1B activations with 65,536 features per layer (16 layers), trained on 300M OpenWebText tokens in 17 hours using a single 8×H100 node. To our knowledge, this is the first open-source implementation for training CLTs on a 1B-scale language model. **Right pair:** Comparison of feature sharding and data parallelism as the total number of features increases. Feature sharding dramatically reduces per-GPU memory while improving throughput, enabling substantially larger dictionaries than conventional data-parallel training.

or during rapid experimentation (e.g., when exploring architectures or hyperparameters) (Bloom et al., 2024). The second approach is to precompute and store activations on disk prior to CLT training, which is more efficient in practice (Ameisen et al., 2025). We compare the trade-offs between on-the-fly activation computation and cached activations in Figure 2. Because activation caching can be generated independently across multiple workers, the one-time preprocessing cost is quickly amortized, making cached activations substantially more efficient once activations are reused multiple times during training.

However, activation caching typically requires several terabytes of storage, making naive caching impractical. CLT-Forge therefore supports a range of quantization and compression schemes, including symmetric per-layer int8 quantization together with optional zstd and LZ4 compression. Activations are stored in independently compressed fixed-size chunks together with their per-layer scale factors, enabling efficient streaming during training. Rather than providing a single compression mode, the framework exposes a trade-off between storage footprint and I/O performance, as shown in Figure 2. Lightweight compression (e.g., LZ4) prioritizes fast writes and reads, while stronger entropy coding (e.g., zstd) achieves higher compression ratios at the expense of slower cache generation. Since activation caches are typically written once but reused over many training epochs, this one-time write overhead is often amortized, making higher compression attractive in practice. In our experiments, int8 quantization with zstd reduces storage by up to 3.6× while maintaining high read throughput, reducing the activation cache of a 300M-token Llama-3.2-1B CLT training run from approximately 20 TB to around 4 TB.

During training, activation chunks are loaded into memory as needed. In a Distributed Data Parallel (DDP) setting, each worker processes different chunks. In contrast, when the CLT is sharded along the feature dimension (see Section 3.3), all workers operate on the same data but on different subsets of features. Finally, similar to (Bloom et al., 2024), we normalize activations to have approximately unit norm by dividing each activation by a normalization factor estimated from the first N batches.

3.2 Sparse Decode

We use JumpReLU as the activation and support multiple sparsity schedulers, following the initialization and training recommendations from (Anthropic, 2025). We also support TopK.

For cross-layer CLTs, the decoder $zW_{\text{dec}}^{\top}$ dominates the per-step FLOPs: each active latent writes to every target layer, so the cost scales with the full expansion K regardless of how sparse z actually is. This is wasted work as soon as training pushes activations into the sparse regime, for the Llama 1B CLT in Figure 3, L_0 has already dropped below 100 by a third of the way through training, and keeps falling.

We exploit this by adding a sparse-aware decode path built on `torch.nn.functional.embedding_bag`: for each token we look up only the active rows of W_{dec} and sum them in a single fused kernel, replacing the dense $\mathcal{O}(K\, d_{\text{in}})$ matmul with an $\mathcal{O}(k\, d_{\text{in}})$ gather where $k = \|z\|_0 \ll K$.

Figure 4 shows that the crossover k grows with

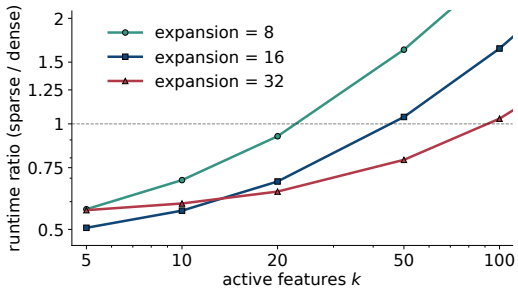

Figure 4: Sparse (`embedding_bag`) vs. dense decode runtime ratio at fixed k (bf16, $d_{\text{in}} = 2048$, cross-layer JumpReLU CLT). Below 1.0, sparse wins; the break-even k scales roughly linearly with expansion — $\sim$20, 50, 100 for $8\times$, $16\times$, $32\times$.

expansion, so wider dictionaries benefit even more from sparse decoding. At runtime we monitor per-batch L_0 and auto-switch from the dense to the sparse path once sparsity crosses the crossover threshold, which, in practice, happens well inside training and remains true at inference.

3.3 GPU Sharding

As model size scales, the corresponding CLTs also quickly become too large for a single GPU. While Distributed Data Parallel (DDP) could suffice for training CLTs on smaller models such as GPT-2, scaling to larger models would inevitably face memory limitations.

Although Fully Sharded Data Parallel (FSDP) could help in this front, it remains suboptimal for CLT architectures and encounters memory constraints at larger scales. Following (Ameisen et al., 2025), we instead adopt feature sharding, where features are partitioned across GPUs and the partial outputs are aggregated at the end of the forward pass. As shown in Figure 3, feature sharding dramatically reduces per-GPU memory while improving throughput, enabling substantially larger Cross-Layer Transcoders than conventional data parallelism. Using this implementation, we train a Cross-Layer Transcoder on Llama-3.2-1B with 65,536 features per layer (16 layers) over 300M OpenWebText tokens in 17 hours on a single 8×H100 node. To our knowledge, this is the first open-source implementation supporting CLT training on a 1B-scale language model.

3.4 Automated Interpretability

For automated interpretability, we parallelize computation over the feature dimension. Each independent worker maintains a subset of encoder vectors on the GPU and computes the corresponding latent activations over large token streams (e.g., 10M tokens). During activation loading, we incrementally track and update the top-K activating sequences directly on the GPU, ensuring that memory usage remains minimal beyond storing these top activations. Within the same pass, we construct feature-level summaries, including top activating tokens, sequences, and optional metadata (e.g., language). If required by the user, these summaries are then used to automatically generate prompts for LLM-based explanation, providing representative examples and instructions to produce coherent feature descriptions. Results are finally stored in a per-worker database keyed by layer and feature index, which are merged into a single unified store upon completion, reducing the cost of writing millions of individual files. This design enables a fully integrated and parallelized, single-pass autointerp pipeline.

3.5 Attribution Graph

For the attribution graph computation, we map our CLT to Circuit-Tracer (Hanna et al., 2025), which allows us to compute, prune the attribution graph and perform interventions when needed.

3.6 Visual Interface

We developed a Python-based Dash interface (Figure 5) that enables interactive visualization and exploration of attribution graphs. The interface provides feature-level information, including top activating tokens, representative sequences, automatically generated explanations, and interventions.

In addition, similar to (Lin and Bloom), the interface includes tools for clustering features and constructing higher-level structures. Users can define clusters, connect them via edges, and perform interventions at the cluster level. These operations trigger background jobs that leverage Circuit Tracer to compute the corresponding effects efficiently (Hanna et al., 2025). The main objective of this interface is to provide a simplified and more easily extensible alternative to (Lin and Bloom), enabling rapid experimentation and iteration in academic settings.

4 Evaluation

Evaluation. As CLT-Forge is a systems library rather than a new modeling method, we evaluate it along two complementary axes: correctness and

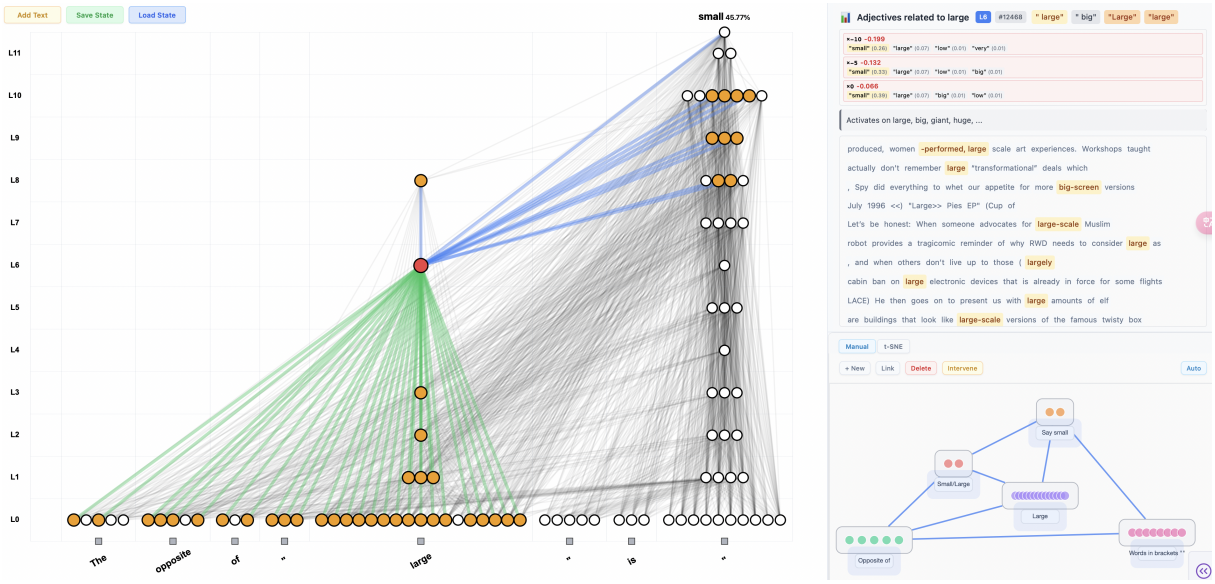

Figure 5: Example of a GPT-2 attribution-graph for the sentence 'The opposite of "large" is' with replacement-score 0.77.

scalability. Across both GPT-2 and Llama-3.2-1B, CLT-Forge reproduces the behavior of existing closed-source Cross-Layer Transcoder implementations, achieving approximately 0.7-0.8 explained variance, 0.8 replacement score, and 0.95 graph completeness at low sparsity levels (Hanna et al., 2025; Merullo et al., 2025; Lindsey et al., 2025a; Harrasse et al., 2025). We then evaluate the systems components that enable large-scale training, including feature sharding, activation caching, and compression, and demonstrate training of a 65,536-feature-per-layer Cross-Layer Transcoder on Llama-3.2-1B over 300M OpenWebText tokens in 17 hours on a single 8×H100 node. Finally, we illustrate the complete workflow through end-to-end examples of automated interpretation, attribution analysis, and interactive visualization.

5 Limitations and Future Work

In this section, we outline several key directions for future work, many of which reflect open problems in mechanistic interpretability at large. First, attention maps are currently frozen when computing attribution graphs; incorporating attention attribution is an important next step (Kamath et al., 2025). We plan to use the approach of (Draye et al., 2025). Second, CLTs remain expensive to train and analyze due to their parameter count and memory footprint, motivating research on more compute- and memory-efficient architectures. Third, reconstruction error is optimized as a proxy for replacement score (Lindsey et al., 2025a). We also found that directly optimizing replacement score led to unstable and noisy results, suggesting the need for more robust objectives or optimization schemes.

6 Conclusion

We introduced CLT-Forge, the first unified open-source framework for scalable training and interpretability of Cross-Layer Transcoders. Our library integrates efficient feature-sharded training, compressed activation caching, automated interpretability, attribution graph computation, and an extensible visualization interface into a single workflow. By combining these components, CLT-Forge enables end-to-end analysis of CLTs at scales that were previously difficult to access in open-source settings. We hope that this framework lowers the barrier to studying mechanistic interpretability at scale, facilitates reproducible research, and supports rapid experimentation in academic environments. Future work will focus on improving efficiency, extending attribution methods (e.g., attention attribution), and open-sourcing CLTs for larger models.

Acknowledgement

This material is based in part upon work supported by the German Federal Ministry of Education and Research (BMBF): Tübingen AI Center, FKZ: 01IS18039B; by the Machine Learning Cluster of Excellence, EXC number 2064/1 – Project

number 390727645; by Schmidt Sciences SAFE-AI Grant; by Coefficient Giving; and by the Survival and Flourishing Fund. Resources used in preparing this research project were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies sponsoring the Vector Institute. F. D. acknowledges support through a fellowship from the Hector Fellow Academy.

References

Emmanuel Ameisen, Jack Lindsey, Adam Pearce, Wes Gurnee, Nicholas L. Turner, Brian Chen, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, and 8 others. 2025. Circuit tracing: Revealing computational graphs in language models. *Transformer Circuits Thread*.

Anthropic. 2025. Circuits updates — january 2025. https://transformer-circuits.pub/2025/january-update/index.html. Accessed: 2026-02-26.

Joseph Bloom, Curt Tigges, Anthony Duong, and David Chanin. 2024. Saelens. https://github.com/decoderesearch/SAELens. Accessed: 2026-02-27.

Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nick Turner, Cem Anil, Carson Denison, Amanda Askell, Robert Lasenby, Yifan Wu, Shauna Kravec, Nicholas Schiefer, Tim Maxwell, Nicholas Joseph, Zac Hatfield-Dodds, Alex Tamkin, Karina Nguyen, and 6 others. 2023. Towards monosemanticity: Decomposing language models with dictionary learning. *Transformer Circuits Thread*. Https://transformer-circuits.pub/2023/monosemantic-features/index.html.

Florent Draye, Anson Lei, Ingmar Posner, and Bernhard Schölkopf. 2025. Sparse attention post-training for mechanistic interpretability. *arXiv preprint arXiv:2512.05865*.

Jacob Dunefsky, Philippe Chlenski, and Neel Nanda. 2024. Transcoders find interpretable llm feature circuits. *Advances in Neural Information Processing Systems*, 37:24375–24410.

EleutherAI. 2024. Sparsify: Training sparse autoencoders and transcoders. https://github.com/EleutherAI/sparsify. Accessed: 2026-02-26.

Michael Hanna, Mateusz Piotrowski, Jack Lindsey, and Emmanuel Ameisen. 2025. Circuit-tracer: A new library for finding feature circuits. In *Proceedings of the 8th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pages 239–249.

Abir Harrasse, Florent Draye, Zhijing Jin, and Bernhard Schölkopf. 2025. Tracing multilingual representations in llms with cross-layer transcoders. *arXiv preprint arXiv:2511.10840*.

Harish Kamath, Emmanuel Ameisen, Isaac Kauvar, Rodrigo Luger, Wes Gurnee, Adam Pearce, Sam Zimmerman, Joshua Batson, Thomas Conerly, Chris Olah, and Jack Lindsey. 2025. Tracing attention computation through feature interactions. *Transformer Circuits Thread*.

Georg Lange. 2025. Cross-layer transcoder. https://github.com/Goreg12345/crosslayer-transcoder. GitHub repository.

Johnny Lin and Joseph Bloom. Neuronpedia: Interactive reference and tooling for analyzing neural networks, 2023. *URL https://www. neuronpedia. org. Software available from neuronpedia. org*.

Jack Lindsey, Emmanuel Ameisen, Neel Nanda, Stepan Shabalin, Mateusz Piotrowski, Tom McGrath, Michael Hanna, Owen Lewis, Curt Tigges, Jack Merullo, Connor Watts, Gonçalo Paulo, Joshua Batson, Liv Gorton, Elana Simon, Max Loeffler, Callum McDougall, and Johnny Lin. 2025a. The circuits research landscape: Results and perspectives. *Neuronpedia*.

Jack Lindsey, Wes Gurnee, Emmanuel Ameisen, Brian Chen, Adam Pearce, Nicholas L. Turner, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, and 8 others. 2025b. On the biology of a large language model. *Transformer Circuits Thread*.

Callum McDougall. 2024. SAE Visualizer. https://github.com/callummcdougall/sae_vis.

Jack Merullo, Connor Watts, Max Loeffler, Liv Gorton, Elana Simon, Tom McGrath, and Owen Lewis. 2025. Replicating circuit tracing for a simple known mechanism. https://www.goodfire.ai/research/replicating-circuit-tracing-for-a-simple-mechanism.

Kiho Park, Yo Joong Choe, and Victor Veitch. 2023. The linear representation hypothesis and the geometry of large language models. *arXiv preprint arXiv:2311.03658*.

Gonçalo Paulo, Alex Mallen, Caden Juang, and Nora Belrose. 2024. Automatically interpreting millions of features in large language models. *arXiv preprint arXiv:2410.13928*.

Curt Tigges. 2025. Cross-layer coding. https://github.com/curt-tigges/crosslayer-coding.

E. Tredal. 2025. Open cross-layer transcoder (openclt). https://github.com/etredal/openCLT. GitHub repository.

A Usage Examples

Minimal examples for the main CLT-Forge workflow are shown below. Complete configurations and executable scripts are available in the repository.

Activation caching.

```
from clt_forge import load_model, ActivationsStore
from clt_forge import CLTTrainingRunnerConfig

model = load_model("Llama-3.2-1B")
cfg = CLTTrainingRunnerConfig(expansion_factor=48)
store = ActivationsStore(model, cfg)
store.generate_and_save_activations()
```

Training.

```
from clt_forge import CLTTrainingRunner

CLTTrainingRunner(cfg).run()
```

Automated interpretability.

```
from clt_forge import AutoInterp, AutoInterpConfig

cfg = AutoInterpConfig(
    model_name="Llama-3.2-1B",
    clt_path="path/to/checkpoint",
)
AutoInterp(cfg).run("path/to/features")
```

Attribution graphs.

```
from clt_forge import AttributionRunner

runner = AttributionRunner(
    model_name="Llama-3.2-1B",
    clt_path="path/to/checkpoint",
)
graph = runner.run("The opposite of 'large' is")
```

Visualization.

```
from clt_forge import launch_interface

launch_interface(graph, "path/to/features")
```

B Additional Training Details

Training objective. Following prior work, we train the CLT to reconstruct the MLP outputs while encouraging sparse feature activations and penalizing dead features. The overall objective is

$$\mathcal{L} = \underbrace{\sum_{\ell'} \|\hat{m}_{\ell'} - m_{\ell'}\|_2^2}_{\text{reconstruction}} + \lambda_0 \underbrace{\sum_{\ell} \tanh\left(C\, z_\ell \odot \left\|\mathbf{W}_{\text{dec}}^{\ell}\right\|\right)}_{\text{sparsity}} + \lambda_1 \underbrace{\sum_{\ell} \mathrm{ReLU}\left(\exp(\tau) - h_\ell^{\text{pre}}\right) \left\|\mathbf{W}_{\text{dec}}^{\ell}\right\|}_{\text{dead-feature penalty}}$$

Here, $\mathbf{W}_{\text{dec}}^{\ell}$ denotes the concatenation of all decoder weights originating from layer ℓ, h_ℓ^{pre} denotes the feature pre-activations, τ is the dead-feature threshold, and C is a scaling constant. The coefficients λ_0 and λ_1 control the sparsity and dead-feature regularization terms, respectively.

Parameter scaling. A CLT associates each transformer layer with an encoder $\mathbb{R}^d \to \mathbb{R}^{ed}$ and each ordered layer pair (ℓ_s, ℓ_t) with $\ell_s < \ell_t$ with a decoder $\mathbb{R}^{ed} \to \mathbb{R}^d$, where $ed = d_{\text{features}}$. Ignoring biases, the total number of parameters is

$$N_{\text{CLT}}(L, d, e) = \left(\frac{L(L-1)}{2} + L\right) ed^2. \tag{5}$$

The parameter count therefore grows quadratically with both the number of layers and the model dimension. For Llama-3.2-1B, with $L = 16$, $d = 2048$, and $e = 48$, this gives approximately 27.4B parameters, motivating the feature-sharded training strategy used in CLT-Forge.

C Configuration

Table 1: Optimal training configurations for the GPT-2 and Llama-3.2-1B Cross-Layer Transcoders (CLTs).

Configuration	**GPT-2**	**Llama-3.2-1B**
Dictionary size, d_{latent}	24,576	65,536
Expansion factor	$32\times$	$32\times$
Training tokens	209M	307M
Token batch size	8192	8192
Adam (β_1, β_2)	$(0.9,\ 0.999)$	$(0.9,\ 0.999)$
Peak learning rate	2×10^{-4}	2×10^{-4}
Sparsity coefficient, λ_0	2	12
Numerical precision	FP32[a]	BF16
Hardware	$8\times$ H100 80GB	$8\times$ H100 80GB
Wall-clock training time	5h 04m	17h 35m
Final mean L_0	10.0	22.51
Mean explained variance	0.75	0.746

D Example of Training Dynamics

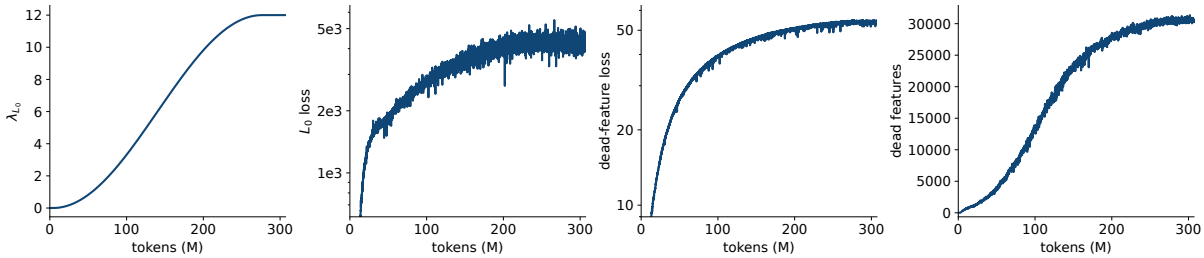

Figure 6: Training curves for the Llama-3.2-1B CLT over 307 M tokens. Left to right: L0-penalty coefficient λ_{L_0} (warmup schedule), L_0 loss, dead-feature auxiliary loss, and dead-feature count. Log-y axes are clipped to the post-warmup range (from 5% of tokens) so the initial spike is off-frame.

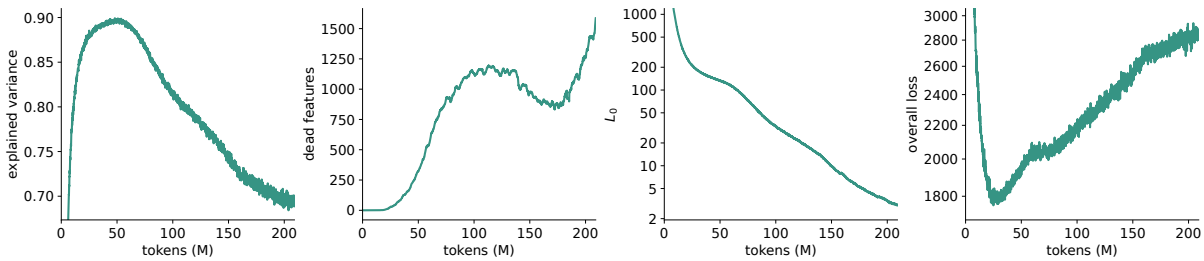

Figure 7: Training curves for the GPT-2 CLT over 209 M tokens. Left to right: explained variance on the reconstruction, dead-feature count, average L_0 per token, and overall training loss. As λ_{L_0} ramps up, L_0 collapses by three decades and dead features grow, while explained variance peaks early and then degrades as the sparsity penalty tightens.