CROSS-LAYER TOPK: DYNAMICALLY ALLOCATING SPARSITY ACROSS LAYERS IN TRANSCODERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Cross-Layer Transcoders (CLTs) decompose the activations of large language models into human-interpretable features and trace how these features interact across layers to form circuits, making them a key tool for mechanistic interpretability. However, training CLTs effectively and efficiently is hard. Beyond the usual trade-off between sparsity and reconstruction quality, a CLT has to model all layers at once, which adds a further challenge. The common JumpReLU recipe controls sparsity only indirectly through a penalty, while the direct extension of TopK selects a fixed number of latents in every layer. Yet reconstruction difficulty differs widely across layers, and intermediate layers are much harder than others, so giving every layer the same budget is suboptimal. We propose Cross-Layer TopK, which sets a single L_0 budget for the whole model and dynamically allocates it across layers during training. In a comparison with JumpReLU and other TopK variants on Qwen3-8B-Base, this gives lower FVU while using 20% fewer active latents than JumpReLU. The better reconstruction carries over to circuit tracing, yielding more complete circuits with higher replacement and completeness scores. The fixed per-token budget of Cross-Layer TopK also makes it easy to pair with sparse decoding, which cuts both computation and communication and gives a $2.9\times$ speedup per training step. We further show that Cross-Layer TopK follows scaling laws in both latent size and sparsity.

1 INTRODUCTION

Mechanistic interpretability aims to explain the computation of LLMs in terms of human-interpretable features and the interactions among them, commonly referred to as circuits. A widely used approach is dictionary learning, in which an encoder maps activations to a sparse, overcomplete set of features and a decoder reconstructs a target from these features under a sparsity constraint. Sparse autoencoders (SAEs), transcoders, and cross-layer transcoders (CLTs) share this encoder–decoder structure and differ only in their inputs and outputs. SAEs reconstruct an activation from itself and thus extract the features it encodes (Gao et al., 2025; Lieberum et al., 2024). Transcoders instead reconstruct the output of an MLP block from its input, yielding a sparse replacement for the MLP (Dunefsky et al., 2024). CLTs further allow a feature read from the residual stream at layer l to contribute to the MLP outputs of all layers $l' \geq l$, so that computation distributed across many layers can be captured by a single feature. Compared with training one transcoder per layer (per-layer transcoders, PLTs), CLTs have been shown to produce sparser and more interpretable circuits, and have revealed the mechanisms behind real-world model behaviors such as planning, multilingual processing, and jailbreaks (Ameisen et al., 2025; Lindsey et al., 2025b).

Training dictionaries effectively and efficiently is challenging, since it requires optimizing reconstruction and sparsity at the same time. Sparsity (typically measured by L_0, the number of active features) is enforced by the activation function that follows the encoder. Gated and JumpReLU activations control the sparsity level through a penalty term, whereas TopK (Gao et al., 2025) directly keeps the k largest pre-activations and sets the rest to zero. CLTs add a further difficulty: because a feature at layer l contributes to the MLP outputs of all later layers, all layers must be modeled jointly, and the reconstruction losses of different layers are coupled.

The activation function that enforces this sparsity is therefore a central design choice. The existing CLT recipe (Ameisen et al., 2025) uses JumpReLU, which has practical drawbacks. Its sparsity is

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

obtained indirectly through a sparsity penalty, and the level reached at the end of training cannot be predicted from the penalty in advance. Hitting a target sparsity therefore requires repeated searches over the penalty coefficient, at a computational cost that is substantial for large CLTs. Even a penalty designed around a target sparsity (Rajamanoharan et al., 2024b) cannot control it exactly, and in our experiments JumpReLU also leaves far more latents dead than TopK (Section 4.2).

A natural remedy is to replace the activation function with TopK (Gao et al., 2025), which keeps the k most strongly activated latents within each layer and thus makes the sparsity exact by construction. This sacrifices a degree of freedom that JumpReLU enjoys, because layers are not equally hard to reconstruct. The early layers and the last few layers of an LLM are typically easier, whereas the middle layers are harder, and activating more latents at a layer buys a better reconstruction there. Because its penalty is summed over all layers, JumpReLU is free to redistribute sparsity across layers on its own, spending more activations where reconstruction is hard. This per-layer variant, which we call Per-Layer TopK, fixes the number of active latents at every layer ahead of time, leaving the model no such freedom during training.

Motivated by these observations, we propose Cross-Layer TopK. Rather than selecting the top-k latents in each layer separately, we pool the latents of all L layers and select the $L \cdot k$ with the largest scores for each token. It thus keeps the total sparsity exact, as TopK does, while leaving the allocation across layers to the model, as JumpReLU does. Further analysis shows that Cross-Layer TopK learns to allocate a larger share of the L_0 budget to intermediate layers during training, which leads to its better overall reconstruction.

We compare Cross-Layer TopK against two TopK variants in ablations. Per-Layer TopK, the direct extension of TopK SAEs to CLTs described above, gives every layer the same budget regardless of how hard it is to reconstruct. Cross-Layer BatchTopK, which combines BatchTopK, originally proposed for SAEs (Bussmann et al., 2024), with our cross-layer selection, selects the top latents over all tokens in a training batch, which relies on the batch being large and randomly mixed; since no such batch is available at inference, a per-feature activation threshold must be calibrated post hoc on training data. This train–inference mismatch leads to good reconstruction on the training set but a marked drop on the validation set.

In summary, we propose Cross-Layer TopK, a simple sparsity mechanism for CLTs, compare it systematically with JumpReLU and other TopK variants, and show that it offers three advantages:

- **Better reconstruction.** By allocating the L_0 budget across layers dynamically, Cross-Layer TopK achieves a lower fraction of variance unexplained (FVU) than all compared methods, even with 20% fewer active latents than JumpReLU, and exhibits scaling laws in both latent size and sparsity.
- **More complete circuits.** The improved reconstruction carries over to circuit tracing, where Cross-Layer TopK attains higher replacement and completeness scores, indicating better circuit-level interpretability.
- **Simpler sparse decoding.** Because the number of active latents per token is fixed, Cross-Layer TopK can use sparse decoding directly, without monitoring L_0 or handling variable shapes; in our tensor-parallel setup this gives a $2.9\times$ speedup per training step.

The rest of the paper is organized as follows. Section 2 reviews prior work on sparse dictionary learning and circuit analysis. Section 3 defines Cross-Layer TopK formally. Section 4 evaluates it against three other CLT architectures on reconstruction quality (Section 4.2), analyzes how it allocates the budget across layers (Section 4.3), evaluates circuit completeness on attribution graphs (Section 4.4), measures the training speedup that a fixed L_0 makes possible (Section 4.5), and studies scaling laws in latent size and sparsity (Section 4.6). Section 5 concludes.

2 Related Work

Sparse dictionary learning. Dictionary learning is motivated by superposition: a layer represents more features than it has neurons (Elhage et al., 2022). Sparse autoencoders (SAEs) address this by learning an over-complete dictionary in which each activation is a sparse combination of interpretable directions (Bricken et al., 2023), and the recipe has since been scaled to frontier models and entire model families (Gao et al., 2025; Lieberum et al., 2024). The mechanism that enforces

sparsity has been the main axis of progress. Early SAEs pair a ReLU encoder with an L_1 penalty, which shrinks the magnitudes of the latents it selects. Gated SAEs remove that bias by separating the choice of which latents to activate from their magnitude (Rajamanoharan et al., 2024a), and JumpReLU SAEs replace the penalty with a learned per-latent threshold trained through a straight-through estimator (Rajamanoharan et al., 2024b). TopK instead fixes the number of active latents per token, removing both the penalty coefficient and the threshold (Gao et al., 2025). BatchTopK SAEs relax the fixed per-token count into a single budget shared across a batch, so that the quota is allocated dynamically among tokens (Bussmann et al., 2024), and Mutual Choice SAEs make a similar batch-level allocation (Ayonrinde, 2024). The selection then depends on which inputs share a batch, so a threshold has to be estimated after training before the SAE can be applied to a single input. Cross-Layer TopK applies the same dynamic allocation along the layer axis, an axis that exists only in cross-layer models. The per-token budget stays fixed, so no post-hoc threshold is needed.

Transcoders and circuit analysis. Transcoders learn a sparse map from the input of an MLP to its output, so that the MLP can be replaced rather than merely described (Dunefsky et al., 2024), and their features have been found more interpretable than those of SAEs (Paulo et al., 2025). A natural way to trace circuits is to train one transcoder per layer (per-layer transcoders, PLTs) and chain them. Cross-layer transcoders instead let a feature write to all later layers, yielding a replacement model whose attribution graphs follow directly from the feature activations; Ameisen et al. (2025) show that CLTs produce sparser and more interpretable circuits than PLTs, and this has supported circuit-level accounts of frontier models (Lindsey et al., 2025b). Existing CLTs use the JumpReLU architecture (Ameisen et al., 2025), and Lindsey et al. (2025a) discuss how to train them efficiently at scale, which our sparse decoder builds on. Draye et al. (2026) release an open-source library for training CLTs and computing attribution graphs. We evaluate circuits with the replacement and completeness scores of Ameisen et al. (2025), which measure how much of an attribution graph is carried by features rather than by reconstruction error.

3 METHOD

3.1 CROSS-LAYER TRANSCODERS

Consider an LLM with L layers. All quantities below are defined for a single token, and we omit the token index. Let $\mathbf{x}^l \in \mathbb{R}^n$ and $\mathbf{y}^l \in \mathbb{R}^n$ denote the input and output of the MLP block at layer l, where n is the residual stream dimension. A cross-layer transcoder (CLT) (Ameisen et al., 2025) encodes $\mathbf{x}^l$ into a sparse feature activation $\mathbf{a}^l \in \mathbb{R}^m$ at each layer, where m is the latent size, and reconstructs $\mathbf{y}^l$ from the feature activations of layer l and all preceding layers.

Encoder. For each layer $l \in \{1, \dots, L\}$, the pre-activation $\mathbf{z}^l \in \mathbb{R}^m$ is

$$\mathbf{z}^l = \mathbf{W}^l_{\text{enc}}\, \mathbf{x}^l,$$

where $\mathbf{W}^l_{\text{enc}} \in \mathbb{R}^{m \times n}$ is the encoder weight matrix of layer l. Stacking all layers gives $\mathbf{Z} = [\mathbf{z}^1, \dots, \mathbf{z}^L] \in \mathbb{R}^{m \times L}$. The latent activations of all layers, $\mathbf{A} = [\mathbf{a}^1, \dots, \mathbf{a}^L] \in \mathbb{R}^{m \times L}$, are obtained by a sparse activation function σ:

$$\mathbf{A} = \sigma(\mathbf{Z}).$$

We write σ as acting on the full matrix $\mathbf{Z}$ because, in general, the activation at one layer may depend on the pre-activations of other layers. For JumpReLU and Per-Layer TopK, σ simply acts on each column $\mathbf{z}^l$ independently. In all mechanisms we consider, σ first rectifies $\mathbf{Z}$ with ReLU, so feature activations are non-negative. The dictionary is over-complete yet each activation is sparse, i.e. $m \gg n$.

Decoder. The CLT reconstructs $\mathbf{y}^l$ as

$$\hat{\mathbf{y}}^l = \sum_{l' \le l} \mathbf{W}^{l' \to l}_{\text{dec}}\, \mathbf{a}^{l'},$$

where $\mathbf{W}^{l' \to l}_{\text{dec}} \in \mathbb{R}^{n \times m}$ is the decoder weight matrix from the features of layer l' to the output of layer l. Following Ameisen et al. (2025), we use no bias terms in either the encoder or the decoder.

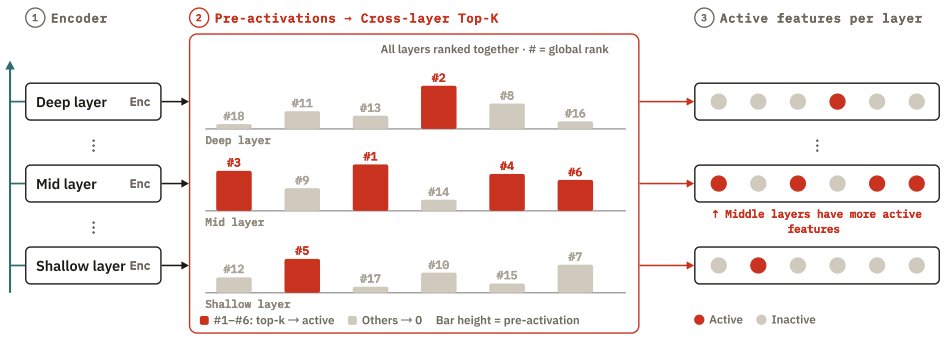


Figure 1: Overview of Cross-Layer TopK. Each layer's encoder maps the MLP input $\mathbf{x}^l$ to pre-activations $\mathbf{z}^l$. The latents of all L layers are pooled and a single TopK selects the $K = L \cdot k$ with the highest scores, so the number of active latents per layer is decided by the model rather than fixed in advance. Each active latent at layer l then contributes to the reconstruction of layers l through L via its decoder weights.

3.2 Cross-Layer TopK

Cross-Layer TopK (Figure 1) is a particular choice of σ: it ranks all latents of a token across all layers by a scalar score and activates the K largest. The total number of active latents per token is therefore fixed, while their distribution over layers is left to the model. We set $K = L \cdot k$ so that Cross-Layer TopK has the same total budget as Per-Layer TopK with k latents per layer.

Formally, let s_i^l denote the score of latent i at layer l, and let $\mathcal{S}$ be the set of the K index pairs (i, l) with the largest scores. Cross-Layer TopK selects positions by the scores but keeps the rectified values of $\mathbf{Z}$:

$$a_i^l = \begin{cases} \max(z_i^l, 0), & (i,l) \in \mathcal{S}, \\ 0, & \text{otherwise}, \end{cases}$$

where a_i^l and z_i^l denote the i-th entries of $\mathbf{a}^l$ and $\mathbf{z}^l$, indicating the activation and pre-activation of the i-th latent in layer l, respectively.

Following common practice (Ameisen et al., 2025), we score each latent by its rectified pre-activation scaled by the norm of its decoder weights. Let $\mathbf{w}_i^{l \to l'}$ denote the i-th column of $\mathbf{W}_{\text{dec}}^{l \to l'}$. Then

$$s_i^l = \max(z_i^l, 0) \cdot \left\| \left[\mathbf{w}_i^{l \to l}; \mathbf{w}_i^{l \to l+1}; \ldots; \mathbf{w}_i^{l \to L} \right] \right\|_2,$$

which measures the ℓ_2 norm of the total output that latent i would write to layers l through L.

3.3 Training Objective

The CLT is trained to reconstruct the MLP output of every layer. The reconstruction loss sums the squared error over all L layers,

$$\mathcal{L}_{\text{recon}} = \sum_{l=1}^{L} \|\mathbf{y}^l - \hat{\mathbf{y}}^l\|_2^2,$$

where the MLP inputs and outputs of each layer are normalized in advance to an expected norm of $\sqrt{n}$, so that the per-layer errors are comparable in scale. To prevent dead latents, we add the auxiliary loss following Gao et al. (2025), adapted to cross-layer selection (Appendix A). The total loss is $\mathcal{L} = \mathcal{L}_{\text{recon}} + \alpha \mathcal{L}_{\text{aux}}$ with coefficient α.

4 Experiments

We evaluate Cross-Layer TopK on reconstruction quality, circuit completeness, and training efficiency. Section 4.1 describes the experimental setup. Section 4.2 compares reconstruction against

three other CLT architectures: JumpReLU, Per-Layer TopK, and Cross-Layer BatchTopK. Section 4.3 analyzes how Cross-Layer TopK allocates the L_0 budget across layers. Section 4.4 tests whether better reconstruction leads to more complete circuits, and Section 4.5 measures training efficiency. Finally, Section 4.6 shows how reconstruction scales with latent size and sparsity.

4.1 Experimental Setup

We train CLTs on the activations of Qwen3-8B-Base (Yang et al., 2025). Activations are collected from the Olmo 3 corpus (Olmo et al., 2025), mixing pre-training text, reasoning chain-of-thought data, and instruction-following SFT data in a 5:2:1 ratio. Activations are stored in a buffer and drawn uniformly at random into micro-batches so that each batch mixes tokens from many sequences. Tokens whose activation norm at any layer exceeds $20\times$ the batch mean are discarded as outliers, both in training and in evaluation. Almost all of the discarded tokens are the first token of a sequence.

4.2 Reconstruction Quality

We measure reconstruction by the fraction of variance unexplained (FVU). For layer l,

$$\mathrm{FVU}^l = \frac{\mathbb{E}\big[\|\mathbf{y}^l - \hat{\mathbf{y}}^l\|_2^2\big]}{\mathbb{E}\big[\|\mathbf{y}^l - \bar{\mathbf{y}}^l\|_2^2\big]},$$

where $\bar{\mathbf{y}}^l = \mathbb{E}[\mathbf{y}^l]$. The overall FVU is the average over layers, $\mathrm{FVU} = \frac{1}{L}\sum_{l=1}^{L} \mathrm{FVU}^l$. All FVUs are measured on a held-out validation set drawn from the same distribution as the training data.

Baselines. We compare Cross-Layer TopK against three other CLT architectures, all with latent size $m = 32{,}768$. The three TopK architectures share identical hyperparameters and activate $K = 360$ latents per token.

- **Per-Layer TopK** activates k latents in every layer, with $L \cdot k = K$ so that the total budget matches Cross-Layer TopK.
- **Cross-Layer BatchTopK** extends BatchTopK (Bussmann et al., 2024) to CLTs, spending a single budget over all layers and all tokens of a training batch. At inference, a per-latent threshold is estimated from training batches following the original procedure.
- **JumpReLU** is trained with the quadratic L_0 penalty of McDougall et al. (2025), $\lambda \frac{2}{L_0^{\text{target}}}\big(\|\mathbf{a}\|_0 - L_0^{\text{target}}\big)^2$, together with the auxiliary loss of Conerly et al. (2025). Because the penalty vanishes as L_0 approaches the target, sparsity stalls above it, ending at an average of 452 active latents per token.

Full hyperparameters are given in Appendix B.

Results. Table 1 summarizes the results. Cross-Layer TopK attains the lowest FVU of the four architectures. Relative to Per-Layer TopK, which differs only in selecting latents within each layer rather than across layers, it lowers FVU from 27.64% to 27.07%. This gap is an average over all 36 layers and understates the gain in the layers that matter most. Cross-Layer TopK shifts budget from the last layers to the hardest intermediate layers. In layers 15–19, where the FVU of Per-Layer TopK exceeds 40%, Cross-Layer TopK lowers it by 1.3–1.6 points (Figure 2). Cross-Layer BatchTopK reaches 26.98% during training, marginally below Cross-Layer TopK, but rises to 29.04% at inference once thresholds replace batch-wise selection. JumpReLU reaches only 32.17% despite using more active latents than the other architectures. Since a larger budget works in its favor, this comparison understates the gap. The three TopK architectures leave 3.0–3.2% of latents dead, whereas JumpReLU leaves 41.2%, so its sparsity budget is concentrated on a much smaller part of the dictionary.

Cross-Layer BatchTopK degrades at inference. The threshold of each latent is estimated by averaging, over training batches, the smallest activation of that latent that survived selection. Batches in which the latent never fires contribute nothing, and latent activations are bursty, so the estimate is taken from exactly the batches in which the latent was most active. The threshold is therefore biased upwards, most severely for rare latents, and Cross-Layer BatchTopK activates fewer latents at inference than in training.

Table 1: Reconstruction with $m = 32{,}768$. L_0 is the average number of active latents per token. All rows are measured on the validation set, except the Cross-Layer BatchTopK training row, which reports the FVU during training with batch-wise selection. Dead latents are those that did not fire in the last 250 optimizer steps (about 8.2M tokens).

Architecture	L_0	**FVU** ↓	**Dead latents** ↓
JumpReLU	452	32.17%	41.2%
Cross-Layer BatchTopK (threshold, inference)	289	29.04%	—
Per-Layer TopK	360	27.64%	3.0%
Cross-Layer TopK (ours)	360	**27.07%**	3.2%
Cross-Layer BatchTopK (batch selection, training)	360	<u>26.98%</u>	3.0%

JumpReLU cannot reach the target sparsity. The quadratic penalty decays to zero as sparsity approaches the target, so near the target it is dominated by the reconstruction loss and sparsity stalls above the target. We also find JumpReLU sensitive to the penalty coefficient λ: if sparsity decreases too fast, a large fraction of latents die, so λ has to be tuned so that sparsity approaches the target gradually, which requires repeated training runs.

4.3 Cross-Layer Allocation

The comparison above shows that freeing the allocation of latents across layers lowers the FVU. We now examine how Cross-Layer TopK uses this freedom. Figure 2 reports, for every layer, the average number of active latents it receives (left) and the FVU it attains (right).

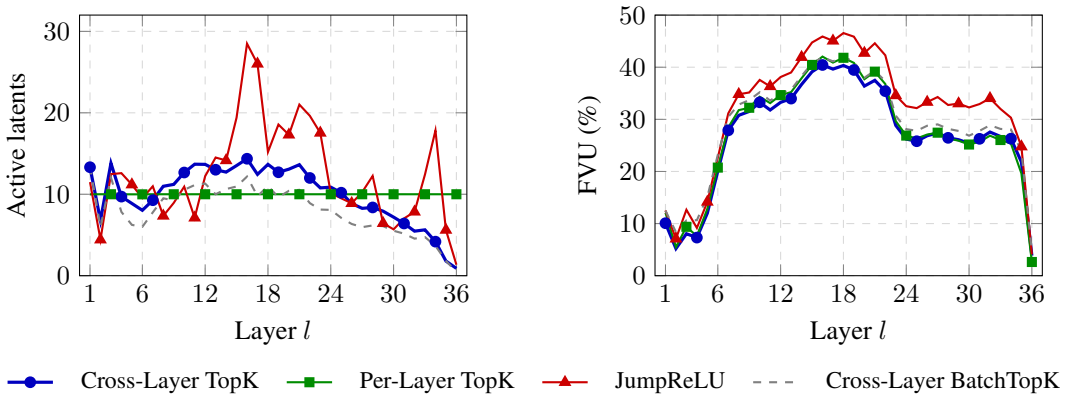


Figure 2: Per-layer allocation (left) and FVU (right). Per-Layer TopK is flat by construction; the architectures that can redistribute concentrate their budget on the intermediate layers, where FVU is highest, and withdraw it from the last layers.

Intermediate layers are harder to reconstruct. Under Per-Layer TopK, which gives every layer the same 10 latents, the FVU rises from about 5–12% in the first few layers to over 40% in layers 15–19, then falls back to under 4% in the last layer. Reconstruction difficulty is thus far from uniform across depth, and a uniform budget is poorly matched to it.

Cross-Layer TopK moves the budget toward the hard layers. With the same total of 360 latents per token, Cross-Layer TopK allocates 12–14 latents to most of layers 10–22, peaking at 14.36 in layer 16, and progressively withdraws from the last layers, down to 0.90 in layer 36. Over the intermediate layers, the budget follows the FVU profile of the fixed-budget baseline and concentrates where reconstruction is hardest: across layers 5–27, the number of latents a layer receives under Cross-Layer TopK correlates with its FVU under Per-Layer TopK at $r = 0.85$. The first layers deviate from this pattern and receive more latents than their low FVU would suggest, e.g. 13.8 in

layer 3, consistent with the observation of Lange et al. (2026) that early-layer features of a CLT can write to all later layers at little extra cost. For the same reason, the outputs of the last layers are partly reconstructed by latents from earlier layers, so these layers receive few latents of their own even though their FVU under Per-Layer TopK remains around 25%. JumpReLU, which is also free to redistribute, arrives at a similar profile with a much larger swing (from 1.35 to 28.49 latents) and peaks in the same layer. Cross-Layer BatchTopK follows the same profile but shifted downward, reflecting its reduced budget of about 289 latents at inference.

The reallocation is a trade. Cross-Layer TopK reconstructs 27 of the 36 layers better than Per-Layer TopK, with the largest gain of 2.0 points in layer 1. The nine exceptions are exactly the last nine layers, from which it withdrew most of its budget, with the largest loss of 2.2 points in layer 35. Averaged over layers the trade is favorable, as the overall FVU in Table 1 shows. JumpReLU is worse than Cross-Layer TopK in every layer despite its larger budget.

4.4 Circuit Completeness

CLTs are built for circuit tracing (Ameisen et al., 2025). Circuit tracing substitutes the CLT for the MLP blocks of the model and expresses the computation behind a prediction as an *attribution graph*, whose nodes are the token embeddings, the active CLT features, and the target logit, and whose edges carry the direct linear effect of one node on another. Since the CLT does not reconstruct the MLP outputs exactly, an error node is added at every layer and token position to absorb the residual. Paths through feature nodes can be inspected feature by feature, whereas error nodes are opaque. The fraction of influence routed through features therefore measures how much of the computation the attribution graph explains in terms of features, and Ameisen et al. (2025) quantify it at two granularities. The *completeness score* is local: for each node, the fraction of its incoming influence that comes from non-error nodes, averaged over nodes weighted by their influence on the target logit. The *replacement score* is global: the fraction of path influence from the token embeddings to the target logit that passes only through non-error nodes. Both range from 0 to 1, higher being better; precise definitions are given in Appendix C.

Table 2: Replacement and completeness scores, averaged over the same 100 prompts for all architectures. L_0 is the average number of active latents per token at inference. Higher is better.

Architecture	L_0	**Replacement** $\uparrow$	**Completeness** $\uparrow$
JumpReLU	452	0.589	0.897
Per-Layer TopK	360	0.577	0.894
Cross-Layer BatchTopK	289	0.578	0.889
Cross-Layer TopK (ours)	360	$\mathbf{0.617}$	$\mathbf{0.902}$

We sample 100 prompts from a held-out validation set drawn from the same distribution as the training data. Following Ameisen et al. (2025), we exclude prompts whose next token is trivial to predict, keeping a prompt only if Qwen3-8B-Base predicts the ground-truth token under greedy decoding, the token's unigram frequency in the pre-training corpus is below 10^{-4}, the token does not appear earlier in the prompt, and the cross-entropy of the prediction exceeds 0.2. The same 100 prompts are used for all four architectures from Section 4.2 ($m = 32{,}768$). Building and scoring an attribution graph requires a full forward pass of the replacement model followed by attribution over every active feature at every token position, so 100 prompts is a substantial evaluation.

Table 2 reports the scores averaged over the 100 prompts. Cross-Layer TopK attains the highest replacement and completeness scores of the four architectures. At the same budget of 360 latents per token, it exceeds Per-Layer TopK by 0.04 on replacement score and 0.008 on completeness score. Cross-Layer BatchTopK, whose post-hoc thresholds leave it with only 289 active latents at inference, scores about the same as Per-Layer TopK on replacement and lowest on completeness. JumpReLU ranks second on replacement score despite the worst FVU; this is explained by its larger budget of 452 active latents per token, which gives the graph more feature paths to route influence through. That Cross-Layer TopK still exceeds it on both scores with 20% fewer active latents indicates that the gain comes from better reconstruction rather than from a larger budget.

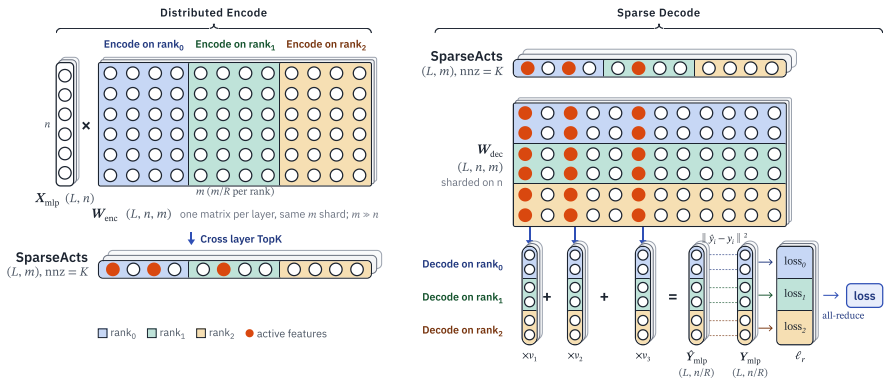


Figure 3: Sparse decoding. The encoder is sharded along the latent axis, the decoder along the model dimension, and only the K active latents of each token are exchanged across ranks, avoiding the dense reconstruction all-reduce. The batch dimension is omitted for clarity.

4.5 Training Efficiency

Cross-Layer TopK activates exactly K latents per token, so the sparsity pattern of the decoder input is known before decoding and has a fixed size. This makes it simple to pair with a sparse decoder: the K active latents can be exchanged across ranks and decoded directly, with no dynamic shapes or padding. JumpReLU offers no such guarantee, since its number of active latents varies per token, and Cross-Layer BatchTopK fixes it only at the batch level.

We pair Cross-Layer TopK with sparse decoding for tensor-parallel training (Figure 3), following the design notes of Lindsey et al. (2025a). The encoder is sharded along the latent axis, the decoder along the model dimension, and only the K active latents of each token are exchanged across the R tensor-parallel ranks. This removes the dense all-reduce of the reconstruction that dominates communication when both encoder and decoder are sharded along the latent axis. Sparse decoding has also been used for JumpReLU CLTs, where training monitors L_0 and switches from dense to sparse decoding once it falls below a crossover point (Draye et al., 2026). With Cross-Layer TopK the number of active latents per token is set by construction, so no monitoring or switching is needed. We compare three numerically equivalent kernels for building the reconstruction from the active latents:

- `dense` scatters the active latents back into a dense buffer and runs the original dense matmul. It is not sparsity-aware, but compared with sharding both encoder and decoder along the latent axis it reads only $1/R$ of the decoder weights per rank and avoids the dense all-reduce.
- `gather` gathers the selected rows of the decoder and accumulates them with `index_add`. It reads only the active rows, but materializes a temporary of size proportional to the number of active latents and relies on atomics.
- `embedding_bag` fuses the gather, the scaling, and the per-token sum into one kernel. It materializes no temporary and needs no atomics.

Appendix D gives the algorithm.

Table 3 compares the three kernels against the baseline that shards both encoder and decoder along the latent axis, on 128 GPUs with $B = 8192$ and $k = 10$. The `dense` kernel alone nearly halves the step time through reduced memory traffic and communication; the sparsity-aware kernels reduce it further, and `embedding_bag` is fastest, cutting compute time by 65% and communication time by 67% for a $2.9\times$ speedup per training step.

4.6 Scaling Laws

We study how Cross-Layer TopK scales along each of its two axes. Figure 4 (left) varies the latent size m from 2^{13} to 2^{15} at a fixed per-layer budget $k = 10$; Figure 4 (right) varies k from 5 to 20 at

Table 3: Step time on 128 GPUs at $k = 10$, $B = 8192$. Step time is split into compute (including memory access) and communication.

Scheme	Step (s)	Compute (s)	Comm. (s)	Speedup
Baseline (latent-sharded)	1.025	0.800	0.224	1.0×
Sparse decode, `dense`	0.552	0.451	0.101	1.9×
Sparse decode, `gather`	0.436	0.334	0.102	2.4×
Sparse decode, `embedding_bag`	**0.355**	**0.281**	**0.074**	**2.9×**

a fixed $m = 2^{15}$. All other hyperparameters are held fixed and every point is a separately trained model. In both cases the overall FVU follows a power law, FVU $\propto m^{-0.16}$ and FVU $\propto k^{-0.13}$, so doubling the latent size or the budget lowers FVU by about 11% and 9% respectively. These exponents are of the same order as those implied by the scaling law that Gao et al. (2025) fit for TopK SAEs over a comparable range. Ameisen et al. (2025) report Pareto improvements as CLT dictionaries grow, but because JumpReLU controls sparsity only through a penalty, L_0 shifts along with latent size and the two effects cannot be separated; Cross-Layer TopK fixes L_0 exactly, which lets us sweep each axis in isolation.

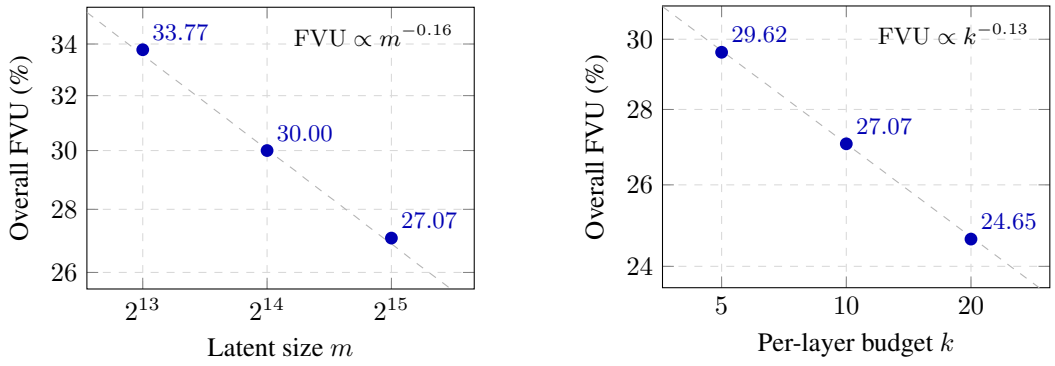


Figure 4: Scaling of Cross-Layer TopK. Left: overall FVU against latent size m at $k = 10$. Right: overall FVU against the per-layer budget k at $m = 2^{15}$, where each model activates $K = L \cdot k$ latents per token. Both axes are logarithmic, so a power law appears as a straight line; dashed lines are least-squares fits in log–log space.

5 Conclusion

We presented Cross-Layer TopK, a sparsity mechanism for cross-layer transcoders that selects the top-K latents over all layers of a token at once. The number of active latents per token is fixed by construction, while their allocation across layers is left to the model. The selection depends on the token alone, so no threshold has to be estimated after training. On Qwen3-8B-Base, Cross-Layer TopK learns to concentrate its budget on the intermediate layers that are hardest to reconstruct, attains lower FVU than Per-Layer TopK at the same sparsity and than JumpReLU at a looser one, and yields more complete circuits, with higher replacement and completeness scores. Its fixed budget also makes sparse decoding straightforward to apply, cutting training step time by 2.9× in our setup, and its reconstruction follows scaling laws in both latent size and sparsity.

Extending TopK from SAEs to CLTs is an intuitive step, and variants of it have appeared as implementation options in prior work and open-source libraries, but they have not been studied in depth or compared systematically. Our results show that the choice among them matters, and that selecting across layers is what makes TopK work well for CLTs. A CLT reconstructs every layer at once, and its decoder grows with the square of the number of layers, so each training run of a large CLT consumes substantial compute. This also kept us from scaling to larger size. More broadly, we hope this work helps make circuit tracing practical for frontier LLMs, advancing our understanding of how they compute and supporting the development of safer AI systems.

AI USE STATEMENT

In this work, generative AI tools were utilized to assist with software code writing, create and modify figures and images (such as Figure 1), edit the manuscript to improve readability, and source and search for research information. We also used generative AI tools to assist with translation, interpret research results and implement research methods (such as sparse decoding). We did not use generative AI tools to generate synthetic datasets, develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, clean and reformat datasets, support qualitative and thematic data analysis. All AI-assisted content, including figures, codes and manuscript texts, has been thoroughly reviewed, verified, and revised manually by the authors to ensure scientific accuracy, logical rigor, and consistency with the original research findings. The authors take full responsibility for the final content of this manuscript, including all text, claims, and research artifacts produced with the assistance of generative AI tools.

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, Tom Henighan, Adam Jermyn, Andy Jones, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. Circuit tracing: Revealing computational graphs in language models, 2025. URL https://transformer-circuits.pub/2025/attribution-graphs/methods.html. Transformer Circuits Thread.

Kola Ayonrinde. Adaptive sparse allocation with mutual choice & feature choice sparse autoencoders. *arXiv preprint arXiv:2411.02124*, 2024.

Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nicholas L. Turner, Cem Anil, Carson Denison, Amanda Askell, Robert Lasenby, Yifan Wu, Shauna Kravec, Nicholas Schiefer, Tim Maxwell, Nicholas Joseph, Alex Tamkin, Karina Nguyen, Brayden McLean, Josiah E. Burke, Tristan Hume, Shan Carter, Tom Henighan, and Chris Olah. Towards monosemanticity: Decomposing language models with dictionary learning. *Transformer Circuits Thread*, October 2023. URL https://transformer-circuits.pub/2023/monosemantic-features.

Bart Bussmann, Patrick Leask, and Neel Nanda. Batchtopk sparse autoencoders, 2024. URL https://arxiv.org/abs/2412.06410.

Tom Conerly, Hoagy Cunningham, Adly Templeton, Jack Lindsey, Basil Hosmer, and Adam Jermyn. Dictionary learning optimization techniques. Transformer Circuits Thread, Circuits Updates — January 2025, January 2025. URL https://transformer-circuits.pub/2025/january-update/index.html#DL. Accessed: 2026-07-08.

Florent Draye, Vedant Palit, Abir Harrasse, Tung-Yu Wu, Jiarui Liu, Punya Syon Pandey, Roderick Wu, Chih-Hao Hsu, Terry Jingchen Zhang, Zhijing Jin, and Bernhard Schölkopf. CLT-Forge: A scalable library for cross-layer transcoders and attribution graphs. *arXiv preprint arXiv:2603.21014*, 2026.

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

Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, Roger Grosse, Sam McCandlish, Jared Kaplan, Dario Amodei, Martin Wattenberg, and Christopher Olah. Toy models of superposition. *Transformer Circuits Thread*, 2022. URL https://transformer-circuits.pub/2022/toy_model/index.html.

Leo Gao, Tom Dupre la Tour, Henk Tillman, Gabriel Goh, Rajan Troll, Alec Radford, Ilya Sutskever, Jan Leike, and Jeffrey Wu. Scaling and evaluating sparse autoencoders. In *International Conference on Learning Representations*, volume 2025, pp. 26721–26754, 2025.

Georg Lange, RGRGRG, Kat Dearstyne, and Kamal Maher. Cross-layer transcoders are incentivized to learn unfaithful circuits. LessWrong, 2026. URL https://www.lesswrong.com/posts/6CS2NDmoLCFcEJMor/cross-layer-transcoders-are-incentivized-to-learn-unfaithful.

Tom Lieberum, Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Nicolas Sonnerat, Vikrant Varma, János Kramár, Anca Dragan, Rohin Shah, and Neel Nanda. Gemma scope: Open sparse autoencoders everywhere all at once on gemma 2. In *Proceedings of the 7th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pp. 278–300, November 2024.

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. The circuits research landscape: Results and perspectives. *Neuronpedia*, 2025a. URL https://neuronpedia.org/graph/info.

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, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. On the biology of a large language model. *Transformer Circuits Thread*, 2025b. URL https://transformer-circuits.pub/2025/attribution-graphs/biology.html.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Callum McDougall, Arthur Conmy, János Kramár, Tom Lieberum, Senthooran Rajamanoharan, and Neel Nanda. Gemma Scope 2: Technical paper. Technical report, Google DeepMind, 2025. URL https://storage.googleapis.com/deepmind-media/DeepMind.com/Blog/gemma-scope-2-helping-the-ai-safety-community-deepen-understanding-of-complex-language-model-behavior/Gemma_Scope_2_Technical_Paper.pdf.

Team Olmo, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, Jacob Morrison, Jake Poznanski, Kyle Lo, Luca Soldaini, Matt Jordan, Mayee Chen, Michael Noukhovitch, Nathan Lambert, Pete Walsh, Pradeep Dasigi, Robert Berry, Saumya Malik, Saurabh Shah, Scott Geng, Shane Arora, Shashank Gupta, Taira Anderson, Teng Xiao, Tyler Murray, Tyler Romero, Victoria Graf, Akari Asai, Akshita Bhagia, Alexander Wettig, Alisa Liu, Aman Rangapur, Chloe Anastasiades, Costa Huang, Dustin Schwenk, Harsh Trivedi, Ian Magnusson, Jaron Lochner, Jiacheng Liu, Lester James V. Miranda, Maarten Sap, Malia Morgan, Michael Schmitz, Michal Guerquin, Michael Wilson, Regan Huff, Ronan Le Bras, Rui Xin, Rulin Shao, Sam Skjonsberg, Shannon Zejiang Shen, Shuyue Stella Li, Tucker Wilde, Valentina Pyatkin, Will Merrill, Yapei Chang, Yuling Gu, Zhiyuan Zeng, Ashish Sabharwal, Luke Zettlemoyer, Pang Wei Koh, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. Olmo 3, 2025. URL https://arxiv.org/abs/2512.13961.

Gonçalo Paulo, Stepan Shabalin, and Nora Belrose. Transcoders beat sparse autoencoders for interpretability. *arXiv preprint arXiv:2501.18823*, 2025.

Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Tom Lieberum, Vikrant Varma, János Kramár, Rohin Shah, and Neel Nanda. Improving dictionary learning with gated sparse autoencoders. *arXiv preprint arXiv:2404.16014*, 2024a.

Senthooran Rajamanoharan, Tom Lieberum, Nicolas Sonnerat, Arthur Conmy, Vikrant Varma, János Kramár, and Neel Nanda. Jumping ahead: Improving reconstruction fidelity with jumprelu sparse autoencoders. *arXiv preprint arXiv:2407.14435*, 2024b.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

A Auxiliary Loss

The auxiliary loss follows Gao et al. (2025): dead latents are asked to reconstruct the error left by the live latents, which gives them a gradient signal and brings them back into use. We adapt it to cross-layer selection by applying the same pooled top-K to dead latents only.

Let $\mathcal{D}$ be the set of index pairs (i, l) whose latent has not fired in the last w optimizer steps. Using the scores s_i^l of Section 3.2, we restrict selection to dead latents,

$$\tilde{s}_i^l = \begin{cases} s_i^l, & (i,l) \in \mathcal{D}, \\ -\infty, & \text{otherwise}, \end{cases}$$

and let $\mathcal{S}'$ be the set of the K index pairs with the largest $\tilde{s}_i^l$. The auxiliary activations keep the pre-activations at these positions,

$$b_i^l = \begin{cases} z_i^l, & (i,l) \in \mathcal{S}', \\ 0, & \text{otherwise}, \end{cases}$$

and are decoded with the same decoder weights as the main reconstruction,

$$\hat{\mathbf{e}}^l = \sum_{l' \le l} \mathbf{W}_{\text{dec}}^{l' \to l} \, \mathbf{b}^{l'}.$$

With $\mathbf{e}^l = \mathbf{y}^l - \hat{\mathbf{y}}^l$ the reconstruction error of layer l, the auxiliary loss is

$$\mathcal{L}_{\text{aux}} = \sum_{l=1}^{L} \left\| \mathbf{e}^l - \hat{\mathbf{e}}^l \right\|_2^2.$$

B Training Settings

B.1 Activation Buffer

Activations are generated on the fly. For each training sequence we run Qwen3-8B-Base and record, for every token, the MLP input and output at all L layers; we call these $2L$ vectors an activation group. We sample 512 groups uniformly at random from each sequence (all groups if the sequence is shorter), discard the rest, and add them to a buffer of $163{,}840 = 10 \times 2^{14}$ groups. Each micro-batch draws 8192 groups uniformly at random from the buffer without replacement, and the buffer is refilled to capacity before the next micro-batch is drawn. Sampling a small subset of tokens per sequence and mixing them in a large buffer ensures that every micro-batch contains tokens from many different sequences.

B.2 Optimization

All models, across latent sizes and activation functions, are trained on 3.2B tokens with identical optimization hyperparameters; we chose this budget because the reconstruction loss had plateaued by then. We use AdamW (Loshchilov & Hutter, 2019) with $(\beta_1, \beta_2) = (0.9, 0.999)$ and a weight decay of 0.01. Gradients are accumulated over 4 micro-batches of 8192 tokens, giving an effective batch of $32{,}768$ tokens and about 9.8×10^4 optimizer steps in total. All schedules below are indexed by the micro-batch count t.

The learning rate follows a square-root warmup and then a linear decay. Over the first 100 micro-batches (0.8M tokens), it rises from $\eta_0 = 0.1\eta_{\max}$ to $\eta_{\max} = 8 \times 10^{-5}$ as $\eta(t) = \eta_0 + (\eta_{\max} - \eta_0)\sqrt{t/100}$; it then decays linearly back to η_0 over the remainder of training.

B.3 Initialization

Our initialization setup generally follows Conerly et al. (2025) and Ameisen et al. (2025). We separately normalize the activations of the input and output of each MLP layer to $\sqrt{n}$, i.e., $\mathbb{E}\left[\|X\|\right] = \sqrt{n}$, where X denotes all input or output activations of the MLP in a given layer.

After normalization, we filter out groups of activations where the norm of the input or output at any layer exceeds 20 times the batch-wise mean, as such outliers would destabilize training. We observe that this filtering strategy almost exclusively acts on groups corresponding to the first token of each training sample.

We initialized every entry of each $\mathbf{W}_{\text{enc}}^{l}$ from $\mathcal{U}(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}})$ and every entry of each $\mathbf{W}_{\text{dec}}^{l' \to l}$ from $\mathcal{U}(-\frac{1}{\sqrt{nL}}, \frac{1}{\sqrt{nL}})$.

For Cross-Layer TopK, we set the per-token budget to $K = 10L = 360$, where $L = 36$ for Qwen3-8B-Base. We applied a warmup schedule to K, linearly decaying it from 3600 to 360 over the first 10^5 micro-batches (0.82B tokens, about a quarter of training), after which it is held constant. The same schedule on the total budget is applied to all TopK variants. The auxiliary selection (Appendix A) always uses the same K as Cross-Layer TopK.

We set the coefficient of the auxiliary loss to $\alpha = 1/32$ throughout the entire training process. A latent counts as dead for the auxiliary loss if it has not fired in the last $w = 250$ optimizer steps, the same window used to report dead latents in Table 1.

B.4 Baseline Hyperparameters

Table 4 lists the settings specific to each baseline; all other settings are shared with Cross-Layer TopK. JumpReLU uses the sparsity penalty $\lambda \frac{2}{L_0^{\text{target}}} \left(\|\mathbf{a}\|_0 - L_0^{\text{target}} \right)^2$ described in Section 4.2.

Table 4: Hyperparameters specific to each baseline.

JumpReLU	
Target L_0^{target}	360
Penalty coefficient λ	$0 \to 5$, linear over the first 10^5 micro-batches
Straight-through estimator bandwidth	1.0
Threshold initialization	$e^{-3.5}$
Auxiliary loss coefficient (Conerly et al., 2025)	3×10^{-6}
Cross-Layer BatchTopK	
Batches used for threshold estimation	1,000 micro-batches

C Graph Score Definitions

This appendix gives the definitions of the replacement score and completeness score used in Section 4.4, following Ameisen et al. (2025).

Attribution graph. For a prompt and a target logit t, the attribution graph has node set $V = E \cup F \cup R \cup \{t\}$: the embedding nodes E (one per prompt token), the feature nodes F (one per active CLT feature at each token position), the error nodes R (one per layer and token position, absorbing the part of the MLP output not reconstructed by the CLT), and the target logit. Let $A \in \mathbb{R}^{|V| \times |V|}$ be the adjacency matrix, where $A_{v,u}$ is the direct linear effect of node u on node v in the replacement model. Embedding and error nodes have no incoming edges; the logit has no outgoing edges. We take t to be the logit of the top-1 predicted token, which by our prompt filter (Section 4.4) is the ground-truth next token.

Normalized influence. Let $\hat{A}$ be the unsigned, row-normalized adjacency matrix,

$$\hat{A}_{v,u} = \frac{|A_{v,u}|}{\sum_{u'} |A_{v,u'}|},$$

so that the incoming edge weights of every node with inputs sum to one. The indirect influence matrix

$$B = (I - \hat{A})^{-1} - I = \hat{A} + \hat{A}^2 + \hat{A}^3 + \cdots$$

sums the weights of all paths from u to v, so $B_{v,u}$ is the total influence of u on v through paths of any length.

Replacement score. Because every path ends at a source node, which is either an embedding or an error node, the influence on the logit splits as $\sum_{e\in E} B_{t,e} + \sum_{r\in R} B_{t,r} = 1$. The replacement score is the fraction that originates from embeddings,

$$\text{Replacement} = \sum_{e\in E} B_{t,e},$$

i.e. the fraction of end-to-end path weight from the prompt to the logit that passes only through feature nodes.

Completeness score. For each node v with incoming edges (the feature nodes and the logit), let

$$f_v = \sum_{u\in E\cup F} \hat{A}_{v,u}$$

be the fraction of its input that comes from non-error nodes. The completeness score averages f_v over nodes, weighting each node by its influence on the target logit,

$$\text{Completeness} = \frac{\sum_v B_{t,v}\, f_v}{\sum_v B_{t,v}},$$

with the logit itself included at weight one. Unlike the replacement score, which requires a complete error-free path, the completeness score gives partial credit to a graph whose important nodes are mostly, but not entirely, explained by features.

D Sparse Decode Details

This appendix gives the algorithm for the sparse decoding scheme of Section 4.5.

Notation. Following the convention of transcoder implementation notes (Lindsey et al., 2025a), we refer to the two sharded axes as the feature axis F and the model dimension D, which are the latent size m and the model dimension n of Section 3.1. B is the number of tokens in a micro-batch, R the number of ranks, L the number of layers, and K the number of active latents per token. $F_{\text{loc}} = F/R$ and $D_{\text{loc}} = D/R$ denote the shard held by one rank. Layers are indexed from 0 to $L-1$, and nnz_i is the number of active latents that layer i contributes to a micro-batch.

Algorithm. Algorithm 1 gives the pseudocode of one training step. The encoder is applied to the local feature shard, and each rank proposes its K highest-scoring latents per token as candidates. Since the global top-K is contained in the union of the local top-K sets, gathering the candidate scores (RK per token) suffices to find each token's global cutoff exactly; each rank then keeps its latents above the cutoff. The active latents of all ranks are gathered in COO format by an all-reduce over four tensors of length nnz (values, token index, feature index, layer index), so that every rank holds the full sparse activation. Each rank then decodes into its own slice of the model dimension, and the per-token loss is all-reduced at the end. To avoid performance cliffs if sparsity collapses, each layer falls back to `dense` when $\text{nnz}_i > \rho BF$, where ρ is a user-defined maximum density for the sparse kernels.

FLOP counts. We charge $2MNP$ for an $(M\times N)(N\times P)$ matmul, plus one such product in the backward pass for each input that requires a gradient. The encoder einsum is evaluated *once* per step—the auxiliary pre-activations $z^{\text{aux}} = z - a$ are elementwise—and its input is a received activation buffer that carries no gradient, so only ∇W_{enc} is charged:

$$C_{\text{enc}} = \underbrace{2BLF_{\text{loc}}D}_{\text{forward}} + \underbrace{2BLF_{\text{loc}}D}_{\nabla W_{\text{enc}}} = 4BLF_{\text{loc}}D, \qquad F_{\text{loc}} = F/W, \quad D_{\text{loc}} = D/W. \tag{1}$$

The decoder is evaluated *twice* per step, once for the main and once for the auxiliary reconstruction, and both its input and its weights carry gradients. Together these contribute a factor $\underbrace{2}_{\text{main+aux}} \times \underbrace{3}_{\text{fwd+bwd}} = 6$ over a single forward pass. A feature at source layer i writes to the $L-i$

Algorithm 1 Sparse decode for one CLT training step on R ranks

Require: MLP inputs X, encoder shard W_{enc} of shape (L, F_{loc}, D), decoder shards W_{dec} with $W_{\text{dec}}[i]$ of shape $(F, L-i, D_{\text{loc}})$, kernel `mode`, one of `dense`, `gather` or `embedding_bag`

procedure TRAINSTEP(X)
  pre $\leftarrow$ EINSUM(`bld, lfd->blf`, X, W_{enc}) ▷ (B, L, F_{loc})
  cand $\leftarrow$ TOPK(SCORE(pre), K) ▷ local candidates, (B, K)
  $\tau \leftarrow$ KTHLARGEST(ALLGATHER(cand), K) ▷ per-token global cutoff, (B)
  acts $\leftarrow$ pre $\odot \mathbb{1}[\text{SCORE}(\text{pre}) \geq \tau]$ ▷ exact global top-K
  sparse $\leftarrow$ ALLREDUCESUM(COO(acts, `feat_offset` $= \text{rank} \cdot F_{\text{loc}}$)) ▷ four tensors of length nnz
  return SPARSEDECODE(sparse, W_{dec}, `mode`) ▷ (B, L, D_{loc})
end procedure

procedure SPARSEDECODE(sparse, W_{dec}, `mode`)
  vals, rows, cols, slices $\leftarrow$ SORTBYLAYER(sparse) ▷ slices$[i]$ is the range of layer i
  recon $\leftarrow \mathbf{0}^{B \times L D_{\text{loc}}}$
  for $i = 0$ **to** $L-1$ **do**
    start, end $\leftarrow$ slices$[i]$
    row, col, val $\leftarrow$ rows[start : end], cols[start : end], vals[start : end]
    $V \leftarrow$ VIEW($W_{\text{dec}}[i]$, $(F, (L-i)D_{\text{loc}})$)
    out $\leftarrow$ recon$[:, iD_{\text{loc}}:]$ ▷ view of layers i to $L-1$
    if `mode` $=$ `dense` **then**
      out $\leftarrow$ out + SCATTER(row, col, val, B, F) V ▷ materialize A_i
    else if `mode` $=$ `gather` **then**
      INDEXADD(out, row, V[col] $\odot$ val) ▷ $(\text{nnz}_i, (L-i)D_{\text{loc}})$ temporary
    else
      out $\leftarrow$ out + EMBEDDINGBAG(col, V, OFFSETS(row), val, `sum`) ▷ fused
    end if
  end for
  return VIEW(recon, (B, L, D_{loc}))
end procedure

target layers $i, \ldots, L-1$, so with the `dense` kernel

$$C_{\text{dec}}^{\text{dense}} = 6 \sum_{i=0}^{L-1} 2BF(L-i)D_{\text{loc}} = 12\, BFD_{\text{loc}}\, \frac{L(L+1)}{2} = \frac{6BFDL(L+1)}{W},$$
$$C_{\text{step}}^{\text{dense}} = C_{\text{enc}} + C_{\text{dec}}^{\text{dense}} = \frac{2BLFD}{W}\,(3L+5), \tag{2}$$

the same decoder FLOP count as when sharding along F. Sharding along D does, however, narrow the reconstruction accumulator from D to D_{loc}, cutting its $BDL(L+1)$ words of traffic per decode by a factor of W, and it removes the reconstruction all-reduce, saving $2BLD$ words of communication per step. The `dense` kernel pays part of this back by re-densifying the gathered activations through a $2BFL$-word scratch buffer, so the *total* per-rank decoder traffic falls by $2.2\times$ at our configuration rather than by W. With the `gather` and `embedding_bag` kernels only the active rows are read. Writing nnz_i for the number of selected features at source layer i and $K = Lk$ for the per-token budget, the same prefactor 6 gives

$$C_{\text{dec}}^{\text{sparse}} = 6 \sum_{i=0}^{L-1} 2\,\text{nnz}_i\,(L-i)\,D_{\text{loc}} \;\approx\; 12\, BkD_{\text{loc}}\, \frac{L(L+1)}{2} = \frac{6BKD(L+1)}{W},$$
$$C_{\text{step}}^{\text{sparse}} = \frac{2BD}{W}\big(2LF + 3K(L+1)\big), \tag{3}$$

replacing the factor FL in the decoder term by K.