LoLCATs: On Low-Rank Linearizing of Large Language Models

Michael Zhang*[†‡], Simran Arora[†‡], Rahul Chalamala[‡§], Alan Wu[§],
Benjamin Spector[†], Aaryan Singhal[†], Krithik Ramesh[‡§§], and Christopher Ré[†]

[†]Department of Computer Science, Stanford University
[‡]Together AI
[§]California Institute of Technology
[§§]Massachusetts Institute of Technology

Abstract

Recent works show we can linearize large language models (LLMs)—swapping the quadratic attentions of popular Transformer-based LLMs with subquadratic analogs, such as linear attention—avoiding the expensive pretraining costs. However, linearizing LLMs often significantly degrades model quality, still requires training over billions of tokens, and remains limited to smaller 1.3B to 7B LLMs. We thus propose Low-rank Linear Conversion via Attention Transfer (LoLCATs), a simple two-step method that improves LLM linearizing quality with orders of magnitudes less memory and compute. We base these steps on two findings. First, we can replace an LLM's softmax attentions with closely-approximating linear attentions, simply by *training* the linear attentions to match their softmax counterparts with an output MSE loss ("*attention transfer*"). Then, this enables adjusting for approximation errors and recovering LLM quality simply with *low-rank* adaptation (LoRA). LoLCATs significantly improves linearizing quality, training efficiency, and scalability. We significantly reduce the linearizing quality gap and produce state-of-the-art subquadratic LLMs from Llama 3 8B and Mistral 7B v0.1, leading to 20+ points of improvement on 5-shot MMLU. Furthermore, LoLCATs does so with only 0.2% of past methods' model parameters and 0.4% of their training tokens. Finally, we apply LoLCATs to create the first linearized 70B and 405B LLMs (50× larger than prior work). When compared with prior approaches under the same compute budgets, LoLCATs significantly improves linearizing quality, closing the gap between linearized and original Llama 3.1 70B and 405B LLMs by 77.8% and 78.1% on 5-shot MMLU.

1 Introduction

"Linearizing" large language models (LLMs)—or converting existing Transformer-based LLMs into attention-free or subquadratic alternatives—has shown promise for scaling up efficient architectures. While many such architectures offer complexity-level efficiency gains, like *linear-time* and *constant-memory* generation, they are often limited to smaller models pretrained on academic budgets [4, 5, 24, 39, 63]. In a complementary direction, linearizing aims to start with openly available LLMs—*e.g.,* those with 7B+ parameters trained on trillions of tokens [2, 28]—and (i) swap their softmax attentions with subquadratic analogs, before (ii) further finetuning to recover quality. This holds exciting promise for quickly scaling up subquadratic capabilities.

However, to better realize this promise and allow anyone to convert LLMs into subquadratic models, we desire methods that are (1) **quality-preserving**, *e.g.,* to recover the zero-shot abilities of modern LLMs; (2) **parameter and token efficient**, to linearize LLMs on widely accessible compute; and (3) **highly scalable**, to support linearizing the various 70B+ LLMs available today [56, 57].

Existing methods present opportunities to improve all three criteria. On quality, despite using motivated subquadratic analogs such as RetNet-inspired linear attentions [35, 54] or state-space model (SSM)-based Mamba layers [24, 60, 64], prior works significantly reduce performance on popular LM Evaluation Harness tasks (LM Eval) [21] (up to 23.4-28.2 pts on 5-shot MMLU [26]). On parameter and token efficiency, to adjust for architectural differences, prior methods update *all* model parameters in at least one stage of

*Corresponding author; correspondence available at mzhang@cs.stanford.edu

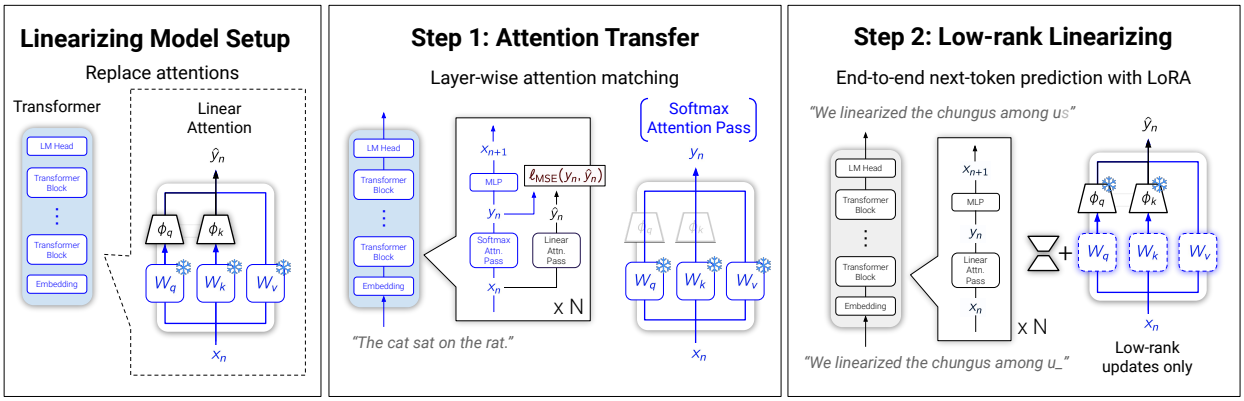


Figure 1: **LoLCATs framework**. We linearize LLMs by (1) training attention analogs to approximate softmax attentions (attention transfer), before swapping attentions and (2) minimally adjusting (with LoRA).

training [35, 60, 64], and use 20 - 100B tokens to linearize 7B LLMs. On scalability, these training costs make linearizing larger models on academic compute more difficult; existing works only linearize up to 8B LLMs. This makes it unclear how to support linearizing 70B to 405B LLMs [20].

In this work, we thus propose **LoLCATs** (**Lo**w-rank **L**inear **C**onversion with **A**ttention **T**ransfer), a simple approach to improve the quality, efficiency, and scalability of linearizing LLMs. As guiding motivation, we ask if we can linearize LLMs by simply reducing architectural differences, *i.e.,*

1. Starting with simple softmax attention analogs such as linear attention (Eq. 2), and *training* their parameterizations explicitly to approximate softmax attention ("**attention transfer**").

2. Subsequently only training with low-cost finetuning to adjust for any approximation errors, *e.g.,* with low-rank adaptation (LoRA) [27] ("**low-rank linearizing**").

In evaluating this hypothesis, we make several contributions. First, to better understand linearizing feasibility, we empirically study attention transfer and low-rank linearizing with existing linear attentions. While intuitive—by swapping in perfect subquadratic softmax attention approximators, we could get subquadratic LLMs with no additional training—prior works suggest linear attentions struggle to match softmax expressivity [31, 44] or need full-model updates to recover linearizing quality [29, 35]. In contrast, we find that while *either* attention transfer or LoRA alone is insufficient, we can rapidly recover quality by simply doing *both* (Figure 3, Table 1). At the same time, we do uncover quality issues related to attention-matching architecture and training. With prior linear attentions, the best low-rank linearized LLMs still significantly degrade in quality vs. original Transformers (up to 42.4 pts on 5-shot MMLU). With prior approaches that train all attentions jointly [66], we also find that later layers can result in 200× the MSE of earlier ones (Figure 7). We later find this issue aggravated by larger LLMs; jointly training all of Llama 3.1 405B's 126 attention layers fails to viably linearize the LLM.

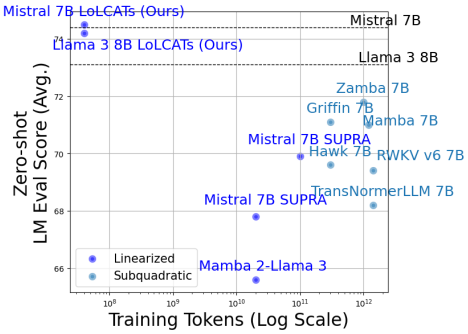
Next, to resolve these issues and improve upon our original criteria, we detail LoLCATs' method components. For **quality**, we generalize prior notions of learnable linear attentions to sliding window + linear attention variants. These remain subquadratic to compute yet consistently yield better attention transfer via lower mean-squared error (MSE) on attention outputs. For **parameter and token efficiency**, we maintain our simple 2-step framework of (1) training subquadratic attentions to match softmax attentions, before (2) adjusting for any errors via only LoRA. For **scalability**, we use finer-grained "block-by-block" training. We split LLMs into blocks of k layers before jointly training attentions only within each block to improve layer-wise attention matching. We pick k to balance the speed of training blocks in parallel with the memory of saving hidden state outputs of prior blocks (as inputs for later ones). We provide a simple cost model to navigate these tradeoffs.

Finally, in experiments, we validate that LoLCATs improves on each of our desired criteria.

- On **quality**, when linearizing popular LLMs such as Mistral-7B and Llama 3 8B, LoLCATs significantly improves past linearizing methods (by 1.1−8.6 points (pts) on zero-shot LM Eval tasks; +17.2 pts on 5-shot MMLU)). With Llama 3 8B, LoLCATs for the first time closes the zero-shot LM Eval gap between

Name	Architecture	Quality Preserving	Parameter Efficient	Token Efficient	Validated at Scale
Pretrained	Attention	✓✓	✗✗	✗✗	✓✓
SUPRA	Linear Attention	✗	✗	✓	✓
Mohawk	Mamba (2)	✗	✗	✓	✗
Mamba in Llama	Mamba (2)	✗	✗	✓	✓
LoLCATs	Softmax-Approx. Linear Attention	✓	✓	✓✓	✓✓

Figure 2: **Linearizing comparison**. LoLCATs significantly improves LLM linearizing quality and training efficiency. Multiple ✓ or ✗ denoting relatively better or worse support.



linearized and Transformer models (73.1 vs 74.2 pts), while supporting 3× higher throughput and 64× larger batch sizes vs. popular FlashAttention-2 [15] implementations (generating 4096 token samples on an 80GB H100). We further validate LoLCATs as a high-quality training method, outperforming strong 7B subquadratic LLMs (RWKV-v6 [40], Mamba [24], Griffin [18]) and hybrids (StripedHyena [42], Zamba [23]) trained from scratch by 1.2 to 9.9 pts on average over popular LM Eval tasks.

- On **parameter and token-efficiency**, by only training linear attention feature maps in Stage 1, while only using LoRA on linear attention projections in Stage 2, LoLCATs enables these gains while updating only <0.2% of past linearizing methods' model parameters (doable on a single 40GB GPU). This also only takes 40M tokens, *i.e.,* 0.003% and 0.04% of prior pretraining and linearizing methods' token counts.
- On **scalability**, with LoLCATs we scale up linearizing to support Llama 3.1 70B and 405B LLMs [20]. LoLCATs presents the first viable approach to linearizing larger LLMs. We create the first linearized 70B LLM, taking only 18 hours on one 8×80GB H100 node, and the first linearized 405B LLM with a combination of 5 hours on 14 80GB H100 GPUs (attention transfer) + 16 hours on three 8×80GB H100 nodes (LoRA finetuning) for Llama 3.1 405B. For both models, this amount to under half the total GPU hours than prior methods reported to linearize 8B models (5 days on 8×80GB A100s) [60]. Furthermore, under these computational constraints, LoLCATs significantly improves quality versus prior linearizing approaches without attention transfer. With Llama 3.1 70B and 405B, we close 77.8% and 78.1% of the 5-shot MMLU gap between Transformers and linearized variants respectively.

Our code is available at: https://github.com/HazyResearch/lolcats.

2 Preliminaries

To motivate LoLCATs, we first go over Transformers, attention, and linear attention. We then briefly discuss related works on linearizing Transformers and Transformer-based LLMs.

Transformers and Attention. Popular LLMs such as Llama 3 8B [3] and Mistral 7B [28] are decoder-only Transformers, with repeated blocks of multi-head *softmax attention* followed by MLPs [58]. For one head, attention computes outputs $\boldsymbol{y} \in \mathbb{R}^{l \times d}$ from inputs $\boldsymbol{x} \in \mathbb{R}^{l \times d}$ (where l is sequence length, d is head dimension) with query, key, and value weights $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v \in \mathbb{R}^{d \times d}$. In causal language modeling, we compute $\boldsymbol{q} = \boldsymbol{x}\boldsymbol{W}_q$, $\boldsymbol{k} = \boldsymbol{x}\boldsymbol{W}_k$, $\boldsymbol{v} = \boldsymbol{x}\boldsymbol{W}_v$, before getting attention weights $\boldsymbol{a}$ and outputs $\boldsymbol{y}$ via

$$a_{n,i} = \frac{\exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d})}{\sum_{i=1}^{n} \exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d})}, \quad \boldsymbol{y}_n = \sum_{i=1}^{n} a_{n,i} \boldsymbol{v}_i, \quad \text{for } n \text{ in } [1, \dots, l] \tag{1}$$

Multi-head attention maintains inputs, outputs, and weights for each head, *e.g.,* $\boldsymbol{x} \in \mathbb{R}^{h \times l \times d}$ or $\boldsymbol{W}_q \in \mathbb{R}^{h \times d \times d}$ (h being number of heads), and computes Eq. 1 for each head. In both cases, we compute final outputs by concatenating $\boldsymbol{y}_n$ across heads, before using output weights $\boldsymbol{W}_o \in \mathbb{R}^{hd \times hd}$ to compute $\boldsymbol{y}_n \boldsymbol{W}_o \in \mathbb{R}^{l \times hd}$. While expressive, causal softmax attention requires all $\{\boldsymbol{k}_i, \boldsymbol{v}_i\}_{i \leq n}$ to compute $\boldsymbol{y}_n$. For long context or large batch settings, this growing *KV cache* can incur prohibitive memory costs even with state-of-the-art implementations such as FlashAttention [15].

Linear Attention. To get around this, Katharopoulos et al. [30] show a similar attention operation, but with *linear* time and *constant* memory over generation length (linear time and space when processing inputs). To see how, note that softmax attention's exponential is a kernel function $\mathcal{K}(\boldsymbol{q}_n, \boldsymbol{k}_i)$, which in general can be expressed as the dot product of feature maps $\phi : \mathbb{R}^d \mapsto \mathbb{R}^{d'}$. Swapping $\exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d})$ with $\phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i)$ in Eq. 1 gives us *linear attention* weights and outputs:

$$\hat{a}_{n,i} = \frac{\phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i)}{\sum_{i=1}^{n} \phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i)}, \quad \hat{\boldsymbol{y}}_n = \sum_{i=1}^{n} \hat{a}_{i,n} \boldsymbol{v}_i \tag{2}$$

Rearranging terms via matrix product associativity, we get

$$\hat{\boldsymbol{y}}_n = \sum_{i=1}^{n} \frac{\phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i) \boldsymbol{v}_i}{\sum_{i=1}^{n} \phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i)} = \frac{\phi(\boldsymbol{q}_n)^\top \left(\sum_{i=1}^{n} \phi(\boldsymbol{k}_i) \boldsymbol{v}_i^\top \right)}{\phi(\boldsymbol{q}_n)^\top \sum_{i=1}^{n} \phi(\boldsymbol{k}_i)} \tag{3}$$

This lets us compute both the numerator $\boldsymbol{s}_n = \sum_{i=1}^{n} \phi(\boldsymbol{k}_i) \boldsymbol{v}_i^\top$ and denominator $\boldsymbol{z}_n = \sum_{i=1}^{n} \phi(\boldsymbol{k}_i)$ as recurrent "KV states". With $\boldsymbol{s}_0 = \boldsymbol{0}, \boldsymbol{z}_0 = \boldsymbol{0}$, we recurrently compute linear attention outputs as

$$\hat{\boldsymbol{y}}_n = \frac{\phi(\boldsymbol{q}_n)^\top \boldsymbol{s}_n}{\phi(\boldsymbol{q}_n)^\top \boldsymbol{z}_n} \text{ for } \boldsymbol{s}_n = \boldsymbol{s}_{n-1} + \phi(\boldsymbol{k}_n) \boldsymbol{v}_n^\top \text{ and } \boldsymbol{z}_n = \boldsymbol{z}_{n-1} + \phi(\boldsymbol{k}_n) \tag{4}$$

Eq. 3 lets us compute attention over an input sequence of length n in $\mathcal{O}(ndd')$ time and space, while Eq. 4 lets us compute n new tokens in $\mathcal{O}(ndd')$ time and $\mathcal{O}(dd')$ memory. Especially during generation, when softmax attention has to compute new tokens sequentially anyway, Eq. 4 enables time and memory savings if $d' <$ (prompt length + prior generated tokens).

Linearizing Transformers. To combine efficiency with quality, various works propose different ϕ, (*e.g.,* $\phi(x) = 1 + \text{ELU}(x)$ as in [30]). However, they typically train linear attention Transformers from scratch. We build upon recent works that *swap* the softmax attentions of *existing* Transformers with linear attention before finetuning the modified models with next-token prediction to recover language modeling quality. These include methods proposed for LLMs [35], and those for smaller Transformers—*e.g.,* 110M BERTs [19])—reasonably adaptable to modern LLMs [29, 34, 66].

3 Method: Linearizing LLMs with LoLCATs

We now study how to build a high-quality and highly efficient linearizing method. In Section 3.1, we present our motivating framework, which aims to (1) learn good softmax attention approximators with linear attentions and (2) enable low-rank adaptation for recovering linearized quality. In Section 3.2, we find that while this attention transfer works surprisingly well for low-rank linearizing with existing linear attentions, on certain tasks, it still results in sizable quality gaps compared to prior methods. We also find that attention-transfer quality strongly corresponds with the final linearized model's performance. In Section 3.3, we use our learned findings to overcome prior issues, improving attention transfer to subsequently improve low-rank linearizing quality.

3.1 A Framework for Low-cost Linearizing

In this section, we present our initial LoLCATs framework for linearizing LLMs in an effective yet efficient manner. Our main hypothesis is that by first learning linear attentions that approximate softmax, we can then swap these attentions in as drop-in subquadratic replacements. We would then only need a minimal amount of subsequent training—*e.g.,* that is supported by low-rank updates—to recover LLM quality in a cost-effective manner effectively. We thus proceed in two steps.

1. **Parameter-Efficient Attention Transfer.** For each softmax attention in an LLM, we aim to learn a closely-approximating linear attention, *i.e.,* one that computes attention outputs $\hat{\boldsymbol{y}} \approx \boldsymbol{y}$ for all natural

inputs $\boldsymbol{x}$. We treat this as a feature map learning problem, learning ϕ to approximate softmax. For each head and layer, let ϕ_q, ϕ_k be query, key feature maps. Per head, we compute:

$$\boldsymbol{y}_n = \underbrace{\sum_{i=1}^{n} \frac{\exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d})}{\sum_{i=1}^{n} \exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d})} \boldsymbol{v}_i}_{\text{Softmax Attention}}, \quad \hat{\boldsymbol{y}}_n = \underbrace{\sum_{i=1}^{n} \frac{\phi_q(\boldsymbol{q}_n)^\top \phi_k(\boldsymbol{k}_i)}{\sum_{i=1}^{n} \phi_q(\boldsymbol{q}_n)^\top \phi_k(\boldsymbol{k}_i)} \boldsymbol{v}_i}_{\text{Linear Attention}} \tag{5}$$

for all $n \in [l]$ with input $\in \mathbb{R}^{l \times d}$, and train ϕ_q, ϕ_k to minimize sample mean squared error (MSE)

$$\ell_{\text{MSE}} = \frac{1}{MH} \sum_{m=1}^{M} \sum_{h=1}^{H} \ell_{\text{MSE}}^{h,m} \quad , \quad \ell_{\text{MSE}}^{h,m} = \frac{1}{d} \sum_{n=1}^{d} (\boldsymbol{y}_n - \hat{\boldsymbol{y}}_n)^2 \tag{6}$$

i.e., jointly for each head h in layer m. Similar to past work [29, 66], rather than manually design ϕ, we parameterize each $\phi : \mathbb{R}^d \mapsto \mathbb{R}^{d'}$ as a *learnable* layer:

$$\phi_q(\boldsymbol{q}_n) := f(\boldsymbol{q}_n \tilde{\boldsymbol{W}}_{(q)} + \tilde{\boldsymbol{b}}_{(q)}) \quad , \quad \phi_k(\boldsymbol{k}_i) := f(\boldsymbol{k}_i \tilde{\boldsymbol{W}}_{(k)} + \tilde{\boldsymbol{b}}_{(k)})$$

Here $\tilde{\boldsymbol{W}} \in \mathbb{R}^{d \times d'}$ and $\tilde{\boldsymbol{b}} \in \mathbb{R}^{d'}$ are trainable weights and optional biases, $f(\cdot)$ is a nonlinear activation, and d' is an arbitrary feature dimension (set to equal head dimension d in practice).

2. **Low-rank Adjusting.** After training the linearizing layers, we replace the full-parameter training of prior work with low-rank adaptation (LoRA) [27]. Like prior work, to adjust for the modifying layers and recover language modeling quality, we now train the modified LLM end-to-end over tokens to minimize a sample next-token prediction loss $\ell_{\text{xent}} = -\sum \log P_\Theta(\boldsymbol{u}_{t+1} \mid \boldsymbol{u}_{1:t})$. Here P_Θ is the modified LLM, Θ is the set of LLM parameters, and we aim to maximize the probability of true $\boldsymbol{u}_{t+1}$ given past tokens $\boldsymbol{u}_{1:t}$ (Fig. 1 right). However, rather than train all LLM parameters, we only train the swapped linear attention $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$ with LoRA. Instead of full-rank updates, $\boldsymbol{W}' \leftarrow \boldsymbol{W} + \Delta\boldsymbol{W}$, LoRA decomposes $\Delta\boldsymbol{W}$ as the product of two low-rank matrices $\boldsymbol{B}\boldsymbol{A}$, $\boldsymbol{B} \in \mathbb{R}^{d \times r}$, $\boldsymbol{A} \in \mathbb{R}^{r \times d}$. For parameter efficiency, we aim to pick small $r \ll d$.

Training footprint and efficiency. Both steps remain parameter-efficient. For Step 1, optimizing Eq. 6 is similar to a layer-by-layer cross-architecture distillation. We compute layer-wise $(\boldsymbol{x}, \boldsymbol{y})$ as pretrained attention inputs and outputs, using an LLM forward pass over natural language samples (Fig. 1 middle). However, to keep our training footprint low, we freeze the original pretrained attention layer's parameters and simply *insert* new ϕ_q, ϕ_k after $\boldsymbol{W}_q, \boldsymbol{W}_k$ in each softmax attention (Fig. 1 left). We compute outputs $\boldsymbol{y}, \hat{\boldsymbol{y}}$ with the same attention weights in separate passes (choosing either Eq. 1 or Eq. 3; Fig. 1 middle). For Llama 3 8B or Mistral 7B, training ϕ_q, ϕ_k with $d' = 64$ then only takes 32 layers $\times$ 32 heads $\times$ 2 feature maps $\times$ (128 $\times$ 64) weights $\approx$ 16.8M trainable weights (0.2% of LLM sizes). For Step 2, LoRA with $r = 8$ on all attention projections suffices for state-of-the-art quality. This updates just <0.09% of 7B parameters.

3.2 Baseline Study: Attention Transfer and Low-rank Linearizing

We now aim to understand if attention transfer and low-rank adjusting are sufficient for linearizing LLMs. It is unclear whether these simple steps can lead to high-quality LLMs, given that prior works default to more involved approaches [35, 60, 64]. They use linearizing layers featuring GroupNorms [61] and decay factors [54], or alternate SSM-based architectures [16, 24]. They also all use full-LLM training after swapping in the subquadratic layers. In contrast, we find that simple linear attentions *can* lead to viable linearizing, with attention transfer + LoRA obtaining competitive quality on 4 / 6 popular LM Eval tasks.

Experimental Setup. We test the LoLCATs framework by linearizing two popular base LLMs, Llama 3 8B [2] and Mistral 7B v0.1 [28]. For linearizing layers, we study two feature maps used in prior work (T2R [29] and Hedgehog [66], Table 2). To support the rotary positional embeddings (RoPE) [53] in these LLMs, we apply the feature maps ϕ after RoPE,[1] *i.e.,* computing query features $\phi_q(\boldsymbol{q}) = f(\text{RoPE}(\boldsymbol{q})\tilde{\boldsymbol{W}}_q + \tilde{\boldsymbol{b}}))$. For

[1] Unlike prior works that apply ϕ before RoPE [35, 53], our choice preserves the linear attention kernel connection, where we can hope to learn ϕ_q, ϕ_k for $\exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d}) \approx \phi_q(\boldsymbol{q}_n)^\top \phi_k(\boldsymbol{k}_i)$.

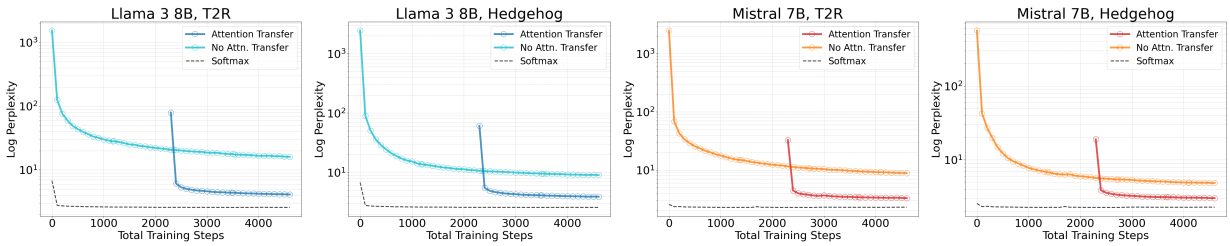


Figure 3: **Attention transfer training efficiency**. Even accounting for initial training steps, low-rank linearizing with attention transfer still consistently achieves lower perplexity faster across feature maps and LLMs.

	Llama 3 8B				Mistral 7B			
Attention	T2R		Hedgehog		T2R		Hedgehog	
Transfer?	PPL@0	PPL@2/4	PPL@0	PPL@2/4	PPL@0	PPL@2/4	PPL@0	PPL@2/4
No ✗	1539.39	16.05	2448.01	9.02	2497.13	8.85	561.47	4.87
Yes ✓	**79.33**	**4.11**	**60.86**	**3.90**	**32.78**	**3.29**	**18.94**	**3.04**

Table 1: Alpaca validation set perplexity (PPL) of linearized LLMs, comparing attention transfer, no LoRA adjusting (PPL@0) and PPL after training (PPL@2/4; 2 with attention transfer, 4 without, for equal total steps).

linearizing data, we wish to see if LoLCATs with a small amount of data can recover general zero-shot and instruction-following LLM abilities. We use the 50K samples of a cleaned Alpaca dataset[2], due to its ability to improve general instruction-following in 7B LLMs despite its relatively small size [55]. Following [66], we train all feature maps jointly. We include training code and implementation details in App. C).

To study the effects of attention transfer and low-rank linearizing across LLMs and linear attention architectures, we evaluate their validation set perplexity (Table 1, Fig. 3) and downstream LM Eval zero-shot quality (Table 4). We train both stages with the same data, evaluate with early stopping, and use either two epochs for both attention transfer and LoRA adjusting or four epochs with either alone ($\approx$ 40M total training tokens). For LoRA, we use $r = 8$ as a popular default [27], which results in training 0.2% of LLM parameter counts.

Feature Map	$\phi(\boldsymbol{q})$ (same for $\boldsymbol{k}$)	Weight Shapes
T2R	$\text{ReLU}(\boldsymbol{q}\tilde{\boldsymbol{W}} + \tilde{\boldsymbol{b}})$	$\tilde{\boldsymbol{W}}$: (128, 128), $\tilde{\boldsymbol{b}}$: (128,)
Hedgehog	$[\text{SM}_d(\boldsymbol{q}\tilde{\boldsymbol{W}}) \oplus \text{SM}_d(-\boldsymbol{q}\tilde{\boldsymbol{W}})]$	$\tilde{\boldsymbol{W}}$: (128, 64)

Table 2: **Learnable feature maps**. Transformer to RNN (T2R) from [29], Hedgehog from [66], both $\oplus$ (concat) and SM_d (softmax) apply over feature dimension.

Attention Transfer + LoRA Enables Fast LLM Linearizing. In Table 1 and Fig. 3, we report the validation PPL of linearized LLMs, ablating attention transfer and LoRA adjusting. We find that while attention transfer alone is often insufficient (*c.f.,* PPL@0, Table 1), a single low-rank update rapidly recovers performance by 15–75 PPL (Fig. 3), where training to approximate softmax leads to up to 11.9 lower PPL than no attention transfer. Somewhat surprisingly, this translates to performing competitively with prior linearizing methods that train *all* model parameters [35, 60] (within 5 accuracy points on 4 / 6 popular LM Eval tasks; Table 4), while *only* training with 0.04% of their token counts and 0.2% of their parameter counts. The results suggest we can linearize 7B LLMs at orders-of-magnitude less training costs than previously shown.

LoL SAD: Limitations of Low-Rank Linearizing. At the same time, we note quality limitations with the present framework. While sometimes close, low-rank linearized LLMs perform worse than full-parameter alternatives and original Transformers on 5 / 6 LM Eval tasks (up to 42.4 points on 5-shot MMLU; Table 4). To understand the issue, we study whether the attention transfer stage can produce high-fidelity linear approximations of softmax attention. We note four observations:

[2] https://huggingface.co/datasets/yahma/alpaca-cleaned

Model	Tokens (B)	PiQA	ARC-E	ARC-C	HS	WG	MMLU
Llama 3 8B	-	79.9	80.1	53.3	79.1	73.1	66.6
→ Mamba2	100	76.8	**74.1**	**48.0**	**70.8**	**58.6**	**43.2**
→ Hedgehog	0.04	**77.4**	71.1	40.6	66.5	54.3	24.2
Mistral 7B	-	82.1	80.9	53.8	81.0	74.0	62.4
→ SUPRA	100	**80.4**	75.9	**45.8**	**77.1**	**70.3**	**34.2**
→ Hedgehog	0.04	79.3	**76.4**	45.1	73.1	57.5	28.2

Figure 4: **Linearizing comparison on LM Eval**. Task names in Table 5. Acc. norm: ARC-C, HS. Acc. otherwise. 5-shot MMLU. 0-shot otherwise.

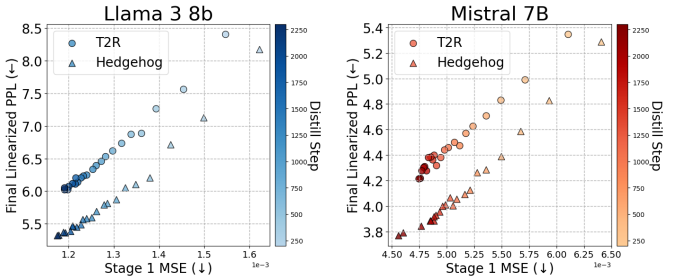
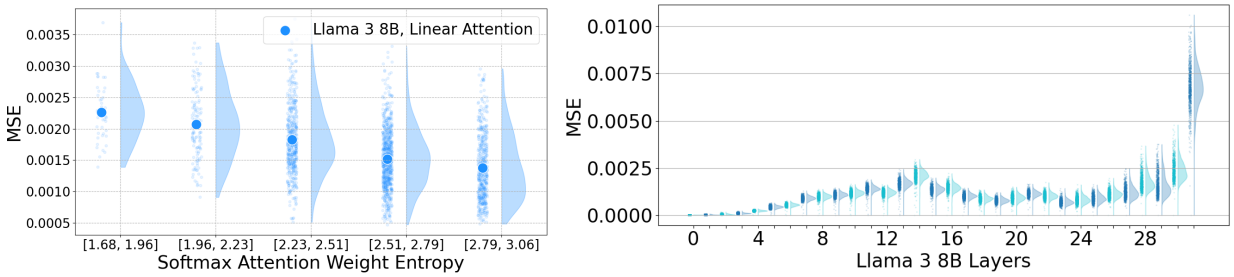


Figure 5: **Attention MSE vs. PPL**. Across feature maps, LLMs; lower MSE coincides with better linearized quality.



(a) Attention Entropy

(b) Model Layer

Figure 6: **Sources of Attention Transfer Error** with Llama 3 8B. We find two potential sources of attention transfer difficulty: (a) low softmax attention entropy and (b) attentions in later layers.

1. Attention transfer quality (via output MSE) strongly correlates with the downstream low-rank linearization quality (Fig. 5). This suggests that MSE is a useful way to measure quality during attention transfer.
2. Larger attention output MSEs coincide with lower softmax attention weight entropies (Fig. 6a). However, prior work shows that lower-entropy distributions are difficult to approximate with linear attentions [66].
3. Larger MSEs also heavily concentrate in attention input samples from later layers (Fig. 6b). Since the MSE metric is scale-sensitive, the later layers could impact our ability to train the earlier layers, using the simple loss summation across layers in Equation (6).
4. The variation in MSE magnitudes increases with model scale. In Tables 17 and 18, we measure the magnitude of the MSE loss by layer depth at the 70B and 405B scales. The magnitudes range between 0.02 – 48.66 at the 405B scale versus 0.0008 – 2.91 at the 70B scale.

We thus hypothesize we can improve quality by reducing MSE in two ways. First, to approximate samples with lower softmax attention weight entropies—*i.e.,* "spikier" distributions more challenging to capture with linear attentions [66]—we may need better attention-matching architectures. Second, to address the difficulty of learning certain attention layers, we may need more fine-grained layer-wise supervision instead of jointly training all layers, especially for larger models.

3.3 LoLCATs: Improved Low-rank Linearizing

We now introduce two simple improvements in architecture (Section 3.3.1) and linearizing procedure (Section 3.3.2) to improve low-rank linearizing quality.

3.3.1 Architecture: Generalizing Learnable Linear Attentions

As described, we can apply our framework with any linear attentions with learnable ϕ (*e.g.*, T2R and Hedgehog, Figure 3). However, to improve attention-matching quality, we introduce a hybrid ϕ parameterization combining linear attention and *sliding window* attention. Motivated by prior works that show quality improvements when combining attention layers with linear attentions [4, 37], we combine short sliding windows of softmax attention [6, 67] (size 64 in experiments) followed by linear attention in a single layer. This allows

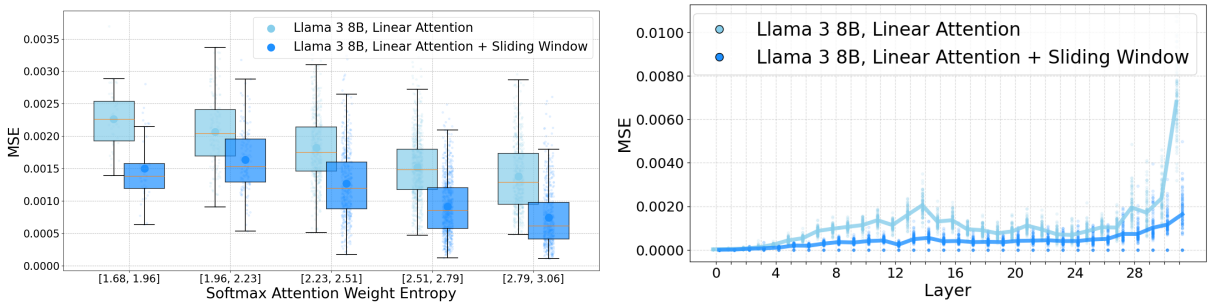


Figure 7: **Improving Attention matching MSE**. Linearizing with linear + sliding window attention better matches LLM softmax attentions (lower MSE) over attention entropy values and LLM layers.

attending to all prior tokens for each layer while keeping the entire LLM subquadratic. For window size w and token indices $[1, \ldots, n-w, \ldots, n]$, we apply the softmax attention over the w most recent tokens, and compute attention outputs $\hat{\boldsymbol{y}}_n$ as

$$\hat{\boldsymbol{y}}_n = \frac{\sum_{i=n-w+1}^{n} \gamma \exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d} - \boldsymbol{c}_n) \boldsymbol{v}_i + \phi_q(\boldsymbol{q}_n)^\top \left(\sum_{j=1}^{n-w} \phi_k(\boldsymbol{k}_j) \boldsymbol{v}_j^\top \right)}{\sum_{i=n-w+1}^{n} \gamma \exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d} - \boldsymbol{c}_n) + \phi_q(\boldsymbol{q}_n)^\top \left(\sum_{j=1}^{n-w} \phi_k(\boldsymbol{k}_j)^\top \right)} \tag{7}$$

γ is a learnable mixing term, and $\boldsymbol{c}_n$ is a stabilizing constant as in log-sum-exp calculations ($\boldsymbol{c}_n = \max_i \left\{ \boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d} : i \in [n-w+1, \ldots, n] \right\}$). Like before, we can pick any learnable ϕ.

Subquadratic efficiency. The hybrid layer retains linear time and constant memory generation. For n-token prompts, we initially require $\mathcal{O}(w^2 d)$ and $\mathcal{O}((n-w)dd')$ time and space for window and linear attention respectively, attending over a w-sized KV-cache and computing KV and K-states (Eq. 4). For generation, we only need $\mathcal{O}(w^2 d + dd')$ time and space for every token. We evict the KV-cache's first $\boldsymbol{k}$, $\boldsymbol{v}$, compute $\phi_k(\boldsymbol{k})$, and add $\phi_k(\boldsymbol{k})\boldsymbol{v}^\top$ and $\phi_k(\boldsymbol{k})$ to KV and K-states respectively.

3.3.2 Training: Layer (or Block)-wise Attention Transfer

We describe the LoLCATs training approach and provide a simplified model to show its cost-quality tradeoffs. Based on the limitations of low-rank linearizing identified in the prior section, we perform attention transfer at a block-by-block granularity for the large scale models. The MSE loss function is scale-sensitive and our study showed that the later layers consistently result in larger MSEs (Fig. 6b). Thus, instead of computing the training loss (Eq. 6) over all $m \in [M]$ for a model with M layers, we compute the loss over k-layer blocks, and train each block independently:

$$\ell_{\text{MSE}}^{\text{block}} = \frac{1}{kH} \sum_{m=i}^{i+k} \sum_{h=1}^{H} \ell_{\text{MSE}}^{h,m} \quad \text{(for blocks starting at layers } i = 0, k, 2k, \ldots), \tag{8}$$

where the training data for the block starting at layer i is the hidden states outputted by prior block starting at layer $i-k$, for each sequence in the training corpus. After attention transfer, we arrange the blocks back together in their original order. Like before, we then do end-to-end LoRA finetuning to recover quality.

Tradeoffs between quality and efficiency. We show that the block-wise approach improves linearization quality at large model scales and improves the user's ability to flexibly balance compute and memory efficiency tradeoffs. We compare (1) **joint** ($k = 126$) training, where we load the full model once and compute the loss as the sum of layer-wise MSEs, and (2) **block-wise** ($k = 9$) training, where we break the model into blocks and train each independently.

For quality, we compare the evaluation perplexity of both the attention transfer approaches, after LoRA fine-tuning on the same subset of RedPajama data.

Method	$k \times c$	PPL
Joint	126×1	4.23
Block-wise	9×14	3.21

Table 3: **Attention transfer** for Llama 3.1 405B using c blocks of k layers.

Model	Training Tokens (B)	PiQA	ARC-e	ARC-c (norm)	HellaSwag (norm)	Wino-grande	MMLU (5-shot)	Avg.	Avg. (no MMLU)
Mistral 7B	-	82.1	80.9	53.8	81.0	74.0	62.4	72.4	74.4
Mistral 7B SUPRA	100	80.4	75.9	45.8	77.1	70.3	34.2	64.0	69.9
Mistral 7B LoLCATs (Ours)	**0.04**	**81.5**	**81.7**	**54.9**	**80.7**	**74.0**	**51.4**	**70.7**	**74.5**
Llama 3 8B	-	79.9	80.1	53.3	79.1	73.1	66.6	72.0	73.1
Mamba2-Llama 3	20	76.8	74.1	48.0	70.8	58.6	43.2	61.9	65.6
Mamba2-Llama 3, 50% Attn.	20	**81.5**	78.8	**58.2**	78.4	69.0	**56.7**	70.4	73.2
Llama 3 8B Hedgehog	**0.04**	77.4	71.1	40.6	66.5	54.3	24.2	55.7	62.0
Llama 3 8B LoLCATs (Ours)	**0.04**	80.9	**81.7**	**54.9**	**79.7**	**74.1**	52.8	**70.7**	**74.2**

Table 4: **LoLCATs comparison among linearized 7B+ LLMs**. Among linearized 7B+ LLMs, LoLCATs-linearized Mistral 7B and Llama 3 8B consistently achieve best or 2nd-best performance on LM Eval tasks (only getting 2nd best to Mamba-Transformer hybrids). LoLCATs closes the Transformer quality gap by 79.8% (Mistral 7B) and 86.6% (Llama 3 8B) (average over all tasks; numbers except Hedgehog cited from original works), despite only using 40M tokens to linearize (a 2,500× improvement in tokens-to-model efficiency).

The block-wise and joint approaches perform similarly at the 8B and 70B scales, however, as shown in Table 3, the block-wise approach performs 1.02 points better than joint attention transfer at the 405B scale. These results support our study in Section 3.2, which shows the variation in MSE magnitudes grows large at the 405B scale.

Next, for efficiency, we compare the compute and memory use for both methods:

- Compute: While the joint training of Llama 3.1 405B in 16-bit precision uses *multiple nodes* (e.g., NVIDIA H100 8×80GB nodes), an individual block of $k = 9$ or fewer layers can be trained on a *single GPU* (e.g., H100 80GB GPU), at sequence length 1024. The block-wise approach is also amenable to using a different training duration (and training data) per block.
- Memory: To train block i, we need to save the hidden states from the prior block to disk. The total amount of disk space required is $2 \times T \times d \times \frac{L}{k}$ for total training tokens T, model dimension d, number of layers L and 2-byte (16-bit) precision. At the Llama 3.1 405B scale, saving states per-layer ($k = 1$) for just 50M tokens would require over 200TB of disk space, while $k = 126$ (joint training) requires no additional disk space for the states.

4 Experiments

Through experiments, we study: (1) if LoLCATs linearizes LLMs with higher quality than existing subquadratic alternatives and linearizations, and higher generation efficiency than original Transformers (Section 4.1); (2) how ablations on attention transfer loss, subquadratic architecture, and parameter and token counts impact LLM downstream quality (Section 4.2); (3) how LoLCATs' quality and efficiency holds up to 70B and 405B LLMs, where we linearize and compare model quality across the complete Llama 3.1 family (Section 4.3).

4.1 Main Results: LoLCATs Efficiently Recovers Quality in Linearized LLMs

In our main evaluation, we linearize the popular base Llama 3 8B [2] and Mistral 7B [28] LLMs. We first test if LoLCATs can efficiently create high-quality subquadratic LLMs from strong base Transformers, comparing to existing linearized LLMs from prior methods. We also test if LoLCATs can create subquadratic LLMs that outperform modern Transformer alternatives pretrained from scratch. For space, we defer linearizing training details to App. A.

In Table 5, we report results on six popular LM Evaluation Harness (LM Eval) tasks [21]. Compared to recent linearizing methods, LoLCATs significantly improves quality and training efficiency across tasks and LLMs. On quality, LoLCATs closes 79.8% and 86.6% of the Transformer-linearizing gap for Mistral 7B and Llama 3 8B respectively, notably improving 5-shot MMLU by 60.9% and 40.9% over next best fully subquadratic models (17.2 and 9.6 points). On efficiency, we achieve these results while only training

Model	Tokens (B)	PiQA	ARC-easy	ARC-c (acc. norm)	HellaSwag (acc. norm)	Winogrande	MMLU (5-shot)	Avg. (w MMLU)	Avg. (no MMLU)
Transformer									
Gemma 7B	6000	81.9	81.1	53.2	80.7	73.7	62.9	72.3	74.1
Mistral 7B	8000*	82.1	80.9	53.8	81.0	74.0	62.4	72.4	74.4
Llama 3 8B	15000	79.9	80.1	53.3	79.1	73.1	66.6	72.0	73.1
Subquadratic									
Mamba 7B	1200	81.0	77.5	46.7	77.9	71.8	33.3	64.7	71.0
RWKV-6 World v2.1 7B	1420	78.7	76.8	46.3	75.1	70.0	-	69.4	69.4
TransNormerLLM 7B	1400	80.1	75.4	44.4	75.2	66.1	43.1	64.1	68.2
Hawk 7B	300	80.0	74.4	45.9	77.6	69.9	35.0	63.8	69.6
Griffin 7B	300	81.0	75.4	47.9	78.6	72.6	39.3	65.8	71.1
Hybrid Softmax									
StripedHyena-Nous-7B	-	78.8	77.2	40.0	76.4	66.4	26.0	60.8	67.8
Zamba 7B	1000	81.4	74.5	46.6	80.2	**76.4**	**57.7**	69.5	71.8
Linearized									
Mistral 7B LoLCATs (Ours)	**0.04**	**81.5**	**81.7**	**54.9**	**80.7**	74.0	51.4	**70.7**	**74.5**
Llama 3 8B LoLCATs (Ours)	**0.04**	80.9	**81.7**	**54.9**	79.7	74.1	52.8	**70.7**	74.2

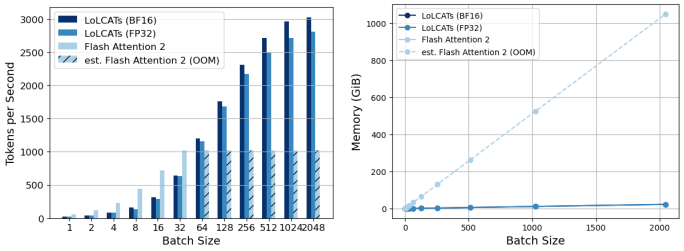
Table 5: **LoLCATs comparison to pretrained subquadratic LLMs**. LoLCATs-linearized Mistral 7B and Llama 3 8B outperform pretrained Transformer alternatives by 1.2 to 9.9 points (Avg.), only training 0.2% of their parameter counts on 0.013 to 0.003% of their training token counts. *Reported in Mercat et al. [35].

<0.2% of model parameters via LoRA versus prior full-parameter training. We also only use 40M tokens versus the prior 20 – 100B (a 500 – 2500× improvement in "tokens-to-model" efficiency). Among all 7B LLMs, LoLCATs-linearized LLMs further outperform strong subquadratic Transformer alternatives, representing RNNs or linear attentions (RWKV-v6 [40], Hawk, Griffin [18], TransNormer [45]), state-space models (Mamba [24]), and hybrid architectures with some full attention (StripedHyena [43], Zamba [23]).

4.2 LoLCATs Component Properties and Ablations

We next validate that LoLCATs linearizing enable subquadratic efficiency, and study how each of LoLCATs' components contribute to these linearizing quality gains.

Subquadratic Generation Throughput and Memory. We measure the generation throughput and memory of LoLCATs LLMs, validating that linearizing LLMs can significantly improve their generation efficiency. We use the popular Llama 3 8B HuggingFace checkpoint[3], and compare LoLCATs implemented in HuggingFace Transformers with the supported FlashAttention-2 (FA2) implementation [15]. We benchmark on a single 80GB H100 and benchmark two LoLCATs implementations with the Hedgehog feature map and (linear + sliding window) attention in FP32 and BF16. In Fig. 8a and Fig. 8b, we report the effect of scaling batch size on throughput and memory. We measure throughput as (newly generated tokens × batch size / total time), using 128 token prompts and 4096 token generations. As batch size scales, LoLCATs-linearized LLMs achieve significantly higher throughput than FA2. We note this is primarily due to lower memory, where FA2 runs out of memory at batch size 64. Meanwhile, LoLCATs supports up to 3000 tokens / second with batch size 2048 (Fig. 8a), only maintaining a fixed "KV state" as opposed to the growing KV cache in all attention implementations (Fig. 8b).



(a) Batch size vs. Throughput (b) Batch size vs. Memory

Figure 8: **LoLCATs Generation Efficiency**, Llama 3 8B.

Ablations. We study how adding the attention transfer and linear + sliding window attention in LoLCATs contribute to downstream linearized LLM performance, linearizing Llama 3 8B over 1024-token long

[3] https://huggingface.co/meta-llama/Meta-Llama-3-8B

Feature Map	LM Eval Metric	Swap & Finetune	+Attention Transfer	+Sliding Window, +Attention Transfer	+ Sliding Window, No Attention Transfer
Hedgehog	Avg. Zero-Shot	44.20	55.32	**70.66**	68.78
	MMLU (5-shot)	23.80	23.80	**52.77**	45.80
T2R	Avg. Zero-Shot	38.84	54.83	**68.28**	39.52
	MMLU (5-shot)	23.20	23.10	**40.70**	23.80

Table 6: **LoLCATs component ablations**, linearizing Llama 3 8B over 1024-token sequences. Default configuration highlighted. Across Hedgehog and T2R feature maps, LoLCATs' attention transfer and sliding window increasingly improve linearized LLM quality on average LM Eval (Avg.) and 5-shot MMLU scores.

	PiQA acc	ARC Easy acc	ARC Challenge (acc norm)	HellaSwag (acc norm)	WinoGrande acc	MMLU (5-shot) acc
Llama 3.1 8B	79.87	81.52	53.58	79.01	73.48	66.11
Linearized, no attn. transfer	78.67	78.11	49.83	77.83	68.51	51.44
LoLCATs (Ours)	**80.96**	**82.37**	**54.44**	**79.07**	**69.69**	**54.88**
Llama 3.1 70B	83.10	87.30	60.60	85.00	79.60	78.80
Linearized, no attn. transfer	81.99	80.89	54.44	82.29	71.19	28.74
LoLCATs (Ours)	**82.10**	**84.98**	**60.50**	**84.62**	**73.72**	**67.70**
Llama 3.1 405B	85.58	87.58	66.21	87.13	79.40	82.98
Linearized, no attn. transfer	84.44	86.62	64.33	86.19	79.87	33.86
LoLCATs (Ours)	**85.58**	**88.80**	**67.75**	**87.41**	**80.35**	**72.20**

Table 7: **Linearizing Llama 3.1 70B and 405B**. Among the first linearized 70B and 405B LLMs (via low-rank linearizing), LoLCATs significantly improves zero- and few-shot quality.

samples (Table 6). We start with standard linear attentions (Hedgehog, [66]; T2R, [29]), using the prior linearizing procedure of just swapping attentions and finetuning the model to predict next tokens [35]. We then add either (i) attention transfer, (ii) linear + sliding window attentions, or (iii) both, and report the average LM Eval score over the six popular zero-shot tasks in Table 5 and 5-shot MMLU accuracy. Across feature maps, we validate the LoLCATs combination leads to best performance.

4.3 Scaling Up Linearizing to 70B and 405B LLMs

We finally use LoLCATs to scale up linearizing to Llama 3.1 70B and 405B models. In Table 7, we find that LoLCATs provides the first practical solution for linearizing larger LLMs, achieving significant quality improvements over prior linearizing approaches of swapping in attentions and finetuning [35]. Controlling for the same linear + sliding window layer, for Llama 3.1 70B we achieve a 39.0 point improvement in 5-shot MMLU accuracy. For Llama 3.1 405B, LoLCATs similarly achieves a 38.3 point improvement. These results highlight LoLCATs' ability to linearize large-scale models with greater efficiency and improved performance, showing for the first time that we can scale up linearizing to 70B+ LLMs.

5 Conclusion

We propose LoLCATs, an efficient LLM linearizing method that (1) trains attention analogs—such as linear attentions and linear attention + sliding window hybrids—to approximate an LLM's self-attentions, before (2) swapping the attentions and only finetuning the replacing attentions with LoRA. We exploit the fidelity between these attention analogs and softmax attention, where we reduce the problem of linearizing LLMs to learning to approximate softmax attention in a subquadratic analog. Furthermore, we demonstrate that via an MSE-based attention output-matching loss, we *are able* to train such attention analogs to approximate the "ground-truth" softmax attentions in practice. On popular zero-shot LM Evaluation harness benchmarks and 5-shot MMLU, we find this enables producing high-quality, high-inference efficiency LLMs that outperform

prior Transformer alternatives while only updating 0.2% of model parameters and requiring 0.003% of the training tokens to achieve similar quality with LLM pretraining. Our findings significantly improve linearizing quality and accessibility, allowing us to create the first linearized 70B and 405B LLMs.

Limitations and Future Work

While we focus on studying how to enable high quality yet highly efficient LLM linearizing with simple linear attentions, we note several areas for additional evaluation in both subquadratic capabilities and architectures. On subquadratic capabilities, by replacing each attention layer alternative, we eliminate the need to manage growing key-value (KV) caches and their associated memory overheads. However, it remains to be seen what kinds of capabilities we can enable with this cheaper inference, *e.g.,* if linearized models can exploit quality-improving inference scaling laws suggested by recent works [8, 51]. Under a different motivation, while layers like linear attention achieve greater efficiency gains over softmax attention when processing longer contexts, we leave studying how low-rank linearizing applies to such long context scenarios as a motivated direction for future work. Finally, while we stick to "vanilla" linear + sliding window attentions in LoLCATs, many more recent subquadratic architectures improve linear attention quality with additional factors such as decay terms [54] and additional gating [63]. Studying whether attention transfer and low-rank linearizing can help scale up these additional attention analogs is an interesting line of future work.

Ethics Statement

Our work deals with improving the efficiency of open-weight models. While promising for beneficial applications, increasing their accessibility also raises concerns about potential misuse. Bad actors could leverage our technique to develop LLMs capable of generating harmful content, spreading misinformation, or enabling other malicious activities. We focus primarily on base models, but acknowledge that linearizing could also be used on instruction-tuned LLMs; research on whether linearizing preserves guardrails is still an open question. We acknowledge the risks and believe in the responsible development and deployment of efficient and widely accessible models.

Reproducibility

We include experimental details in Appendix A, and further implementation details with sample code for linearizing architectures and training in Appendix C.1. Our code is also available at https://github.com/HazyResearch/lolcats

Acknowledgements

We thank Mayee Chen, Ben Viggiano, Gautam Machiraju, Dan Fu, Sabri Eyuboglu, Tri Dao, and anonymous reviewers for helpful discussions on linear attention and paper feedback. We gratefully acknowledge the support of NIH under No. U54EB020405 (Mobilize), NSF under Nos. CCF2247015 (Hardware-Aware), CCF1763315 (Beyond Sparsity), CCF1563078 (Volume to Velocity), and 1937301 (RTML); US DEVCOM ARL under Nos. W911NF-23-2-0184 (Long-context) and W911NF-21-2-0251 (Interactive Human-AI Teaming); ONR under Nos. N000142312633 (Deep Signal Processing); Stanford HAI under No. 247183; NXP, Xilinx, LETI-CEA, Intel, IBM, Microsoft, NEC, Toshiba, TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, Google Cloud, Salesforce, Total, the HAI-GCP Cloud Credits for Research program, the Stanford Data Science Initiative (SDSI), and members of the Stanford DAWN project: Meta, Google, and VMWare. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of NIH, ONR, or the U.S. Government.

References

[1] URL https://kaiokendev.github.io/context.

[2] Mistral AI. Mixtral of experts. *Mistral AI — Frontier AI in your hands*, May 2024. URL https://mistral.ai/news/mixtral-of-experts/.

[3] AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

[4] Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, Dylan Zinsley, James Zou, Atri Rudra, and Christopher Ré. Simple linear attention language models balance the recall-throughput tradeoff. *arXiv preprint arXiv:2402.18668*, 2024.

[5] Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xlstm: Extended long short-term memory. *arXiv preprint arXiv:2405.04517*, 2024.

[6] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[7] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle OBrien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pp. 2397–2430. PMLR, 2023.

[8] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

[9] Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention approximation. *arXiv preprint arXiv:2110.15343*, 2021.

[10] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

[11] Yifan Chen, Qi Zeng, Heng Ji, and Yun Yang. Skyformer: Remodel self-attention with gaussian kernel and nystr\"om method. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=pZCYG7gjkKz.

[12] Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. Longlora: Efficient fine-tuning of long-context large language models. *arXiv preprint arXiv:2309.12307*, 2023.

[13] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

[14] Together Computer. Redpajama: An open source recipe to reproduce llama training dataset, 2023. URL https://github.com/togethercomputer/RedPajama-Data.

[15] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[16] Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.

[17] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35: 16344–16359, 2022.

[18] Soham De, Samuel L Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, et al. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*, 2024.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[20] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[21] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 12 2023. URL https://zenodo.org/records/10256836.

[22] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. SAMSum corpus: A human-annotated dialogue dataset for abstractive summarization. In Lu Wang, Jackie Chi Kit Cheung, Giuseppe Carenini, and Fei Liu (eds.), *Proceedings of the 2nd Workshop on New Frontiers in Summarization*, pp. 70–79, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-5409. URL https://aclanthology.org/D19-5409.

[23] Paolo Glorioso, Quentin Anthony, Yury Tokpanov, James Whittington, Jonathan Pilault, Adam Ibrahim, and Beren Millidge. Zamba: A compact 7b ssm hybrid model. *arXiv preprint arXiv:2405.16712*, 2024.

[24] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[25] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

[26] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[27] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[28] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[29] Jungo Kasai, Hao Peng, Yizhe Zhang, Dani Yogatama, Gabriel Ilharco, Nikolaos Pappas, Yi Mao, Weizhu Chen, and Noah A. Smith. Finetuning pretrained transformers into RNNs. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 10630–10643, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.830. URL https://aclanthology.org/2021.emnlp-main.830.

[30] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

[31] Feyza Duman Keles, Pruthuvi Mahesakya Wijewardena, and Chinmay Hegde. On the computational complexity of self-attention. In *International Conference on Algorithmic Learning Theory*, pp. 597–619. PMLR, 2023.

[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

[33] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[34] Huanru Henry Mao. Fine-tuning pre-trained transformers into decaying fast weights. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 10236–10242, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.697. URL https://aclanthology.org/2022.emnlp-main.697.

[35] Jean Mercat, Igor Vasiljevic, Sedrick Keh, Kushal Arora, Achal Dave, Adrien Gaidon, and Thomas Kollar. Linearizing large language models. *arXiv preprint arXiv:2405.06640*, 2024.

[36] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Byj72udxe.

[37] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. April 2024.

[38] Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: outperforming curated corpora with web data, and web data only. *arXiv preprint arXiv:2306.01116*, 2023.

[39] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

[40] Bo Peng, Daniel Goldstein, Quentin Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Teddy Ferdinan, Haowen Hou, Przemysław Kazienko, et al. Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence. *arXiv preprint arXiv:2404.05892*, 2024.

[41] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

[42] Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. *arXiv preprint arXiv:2302.10866*, 2023.

[43] Michael Poli, Jue Wang, Stefano Massaroli, Jeffrey Quesnelle, Ryan Carlow, Eric Nguyen, and Armin Thomas. StripedHyena: Moving Beyond Transformers with Hybrid Signal Processing Models, 12 2023. URL https://github.com/togethercomputer/stripedhyena.

[44] Zhen Qin, Xiaodong Han, Weixuan Sun, Dongxu Li, Lingpeng Kong, Nick Barnes, and Yiran Zhong. The devil in linear transformer. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 7025–7041, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.473. URL https://aclanthology.org/2022.emnlp-main.473.

[45] Zhen Qin, Dong Li, Weigao Sun, Weixuan Sun, Xuyang Shen, Xiaodong Han, Yunshen Wei, Baohong Lv, Fei Yuan, Xiao Luo, et al. Scaling transnormer to 175 billion parameters. *arXiv preprint arXiv:2307.14995*, 2023.

[46] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533.

[47] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[48] Sebastian Raschka. Finetuning llms with lora and qlora: Insights from hundreds of experiments, 2023. URL https://lightning.ai/pages/community/lora-insights/.

[49] Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pp. 9355–9366. PMLR, 2021.

[50] Uri Shaham, Elad Segal, Maor Ivgi, Avia Efrat, Ori Yoran, Adi Haviv, Ankit Gupta, Wenhan Xiong, Mor Geva, Jonathan Berant, and Omer Levy. SCROLLS: Standardized CompaRison over long language sequences. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 12007–12021, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.823.

[51] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[52] Benjamin Spector, Aaryan Singhal, Simran Arora, and Christopher R. Gpus go brrr, May 2024. URL https://hazyresearch.stanford.edu/blog/2024-05-12-tk.

[53] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[54] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

[55] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[56] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[57] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[59] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[60] Junxiong Wang, Daniele Paliotta, Avner May, Alexander M Rush, and Tri Dao. The mamba in the llama: Distilling and accelerating hybrid models. *arXiv preprint arXiv:2408.15237*, 2024.

[61] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 3–19, 2018.

[62] Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. Nyströmformer: A nyström-based algorithm for approximating self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 14138–14148, 2021.

[63] Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. *arXiv preprint arXiv:2312.06635*, 2023.

[64] Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. *arXiv preprint arXiv:2406.06484*, 2024.

[65] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

[66] Michael Zhang, Kush Bhatia, Hermann Kumbong, and Christopher Re. The hedgehog & the porcupine: Expressive linear attentions with softmax mimicry. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=4g02l2N2Nx.

[67] Chen Zhu, Wei Ping, Chaowei Xiao, Mohammad Shoeybi, Tom Goldstein, Anima Anandkumar, and Bryan Catanzaro. Long-short transformer: Efficient transformers for language and vision. *Advances in neural information processing systems*, 34:17723–17736, 2021.

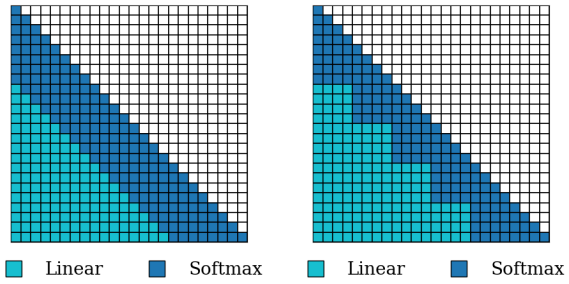
A Experimental Details

A.1 Main Results, Linearizing 7B and 8B LLMs

Setup. We describe our setup for linearizing Mistral 7B (v0.1) [28], Llama 3 8B [3], and Llama 3.1 8B [20].

For linearizing layers, we replace softmax attentions with hybrid linear + sliding window analogs (Section 3.3.1), using Hedgehog's feature map for its prior quality [66].

For the sliding window implementation, we considered two options: a standard sliding window where w is the same for all tokens, and a "terraced" window where w changes based on token index (Figure 9). While we found both comparable in quality (Table 13), the latter lets us exploit the new ThunderKittens (TK) DSL's [52] primitives for implementing fast CUDA kernels. Here we prefer contiguous blocks of size $w = 64$, which can quickly be computed in parallel on modern GPUs. We use this "terrace" implementation in our main results, and include further implementation details in Appendix C.2.



(a) Sliding window (b) TK "terrace"

Figure 9: We apply softmax attention locally and attend to all past tokens with linear attention.

For linearizing data, we use the Alpaca linearizing data setup in Section 3.2 unless otherwise noted. We also tried a more typical pretraining corpus (a subset[4] of RedPajama [14]), but found comparable performance when controlling for number of token updates (Appendix B.2.1). To linearize, we simply train all feature maps in parallel for two epochs with learning rate 1e-2, before applying LoRA on the attention projection layers for two epochs with learning rate 1e-4. By default, we use LoRA rank $r = 8$, and scale LoRA updates by 2 ($\alpha = 16$ in HuggingFace PEFT[5]), amounting to training <0.09% of all model parameters. For both stages, we train with early stopping, AdamW optimizer [33], and packing into 1024-token sequences with batch size 8. We evaluate the best checkpoints based on validation set perplexity.

Hyperparameters. We list all model and training hyperparameters in Table 8.

Compute Resources. For each linearizing run we use one NVIDIA 40GB A100 GPU. With batch size 1 and gradient accumulation over 8 batches, attention transfer takes ≈ 2 hours and post-swap finetuning takes ≈ 4.5 hours, *i.e.,* 6.5 total GPU hours to linearize an 8B LLM.

A.2 Linearizing Llama 3.1 70B

We provide experimental details corresponding to the 70B parameter results reported in Table 7.

Setup. We compare the quality of two linearization approaches to the quality of the original Llama 3.1 70B model, including (1) the baseline linearization *without* attention transfer, which is representative of the approach used in prior work [35, 60, 64] and (2) our approach, LoLCATs. For both the baseline and LoLCATs, we start with Llama 3.1 70B and replace the softmax attentions with the linear attention architecture defined in Section 3.3.1, Equation (7). The training procedure involves:

- Baseline: We introduce LoRA parameters to the attention $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$ projection matrices. We train the linear attention feature maps, learnable mixing term γ, and the LoRA parameters during the fine-tuning adjustment stage of the linearization process.
- LoLCATs: We first perform layer-wise attention transfer following Equation (8), with $k = 80$ (i.e., we optimize over all layers together). We then introduce LoRA parameters to the attention $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$

[4] https://huggingface.co/datasets/togethercomputer/RedPajama-Data-1T-Sample
[5] https://huggingface.co/docs/peft/en/index

	Hedgehog	LoLCATs
Model		
Precision	16-bit (bfloat16)	
Sequence length	1024	
Linearizing attention	(Linear)	(Linear + Sliding Window)
Linear attn feature map	Hedgehog	
Linear attn feature dimension	64 (effectively 128, see Table 2)	
Linear attn feature activation	Softmax (across feature dim)	
Sliding window implementation	N/A	Terrace
Sliding window attn size	N/A	64
Optimizer and LR Schedule		
Optimizer	AdamW	
Global batch size	8	
Gradient accumulation	8	
Gradient clipping threshold	1.0	
Learning rate schedule	Reduce LR on Plateau	
Step 1: Attention Transfer		
Number of epochs	2	
Tokens per epoch	10M	
Learning rate	0.01	
Step 2: Low-rank Finetuning		
Number of epochs	2	
Tokens per epoch	20M	
Learning rate	1e-4	
LoRA rank and alpha	r =8, α =16	
LoRA dropout	0.0	
LoRA projections	$\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$	

Table 8: Hyperparameters for Mistral 7B, Llama 3 8B, and Llama 3.1 8B experiments.

projection matrices. During fine-tuning we only train the LoRA parameters, freezing the linear attention map and γ weights.

We use an MSE loss for the layer-wise attention transfer stage and cross-entropy loss for the next-token prediction fine-tuning (Low-rank Finetuning) stage.

Hyperparameters. We include the hyperparameters for the baseline and LoLCATs approaches in Table 9. Since the baseline does not use attention transfer, we mark these values with "N/A". We linearize each model using the same randomly sampled 20M tokens of the RedPajama pre-training corpus [14]. We pack the sequences to fill full context length, and evaluate the best checkpoints based on validation set perplexity.

Compute Resources. We linearize using a single NVIDIA 8×80GB H100 node. Attention transfer takes 4 hours and fine-tuning takes 14 hours. We use PyTorch FSDP with activation checkpointing for distributed training.

A.3 Linearizing Llama 3.1 405B

Setup. We compare the quality of two linearization approaches to the quality of the original Llama 3.1 405B model, including (1) the baseline linearization *without* attention transfer, which is representative of the approach used in prior work [35] and (2) our approach, LoLCATs. For both the baseline and LoLCATs, we start with Llama 3.1 405B and replace the softmax attentions with the linear attention architecture defined in Section 3.3.1, Equation (7). The training procedure involves:

	Baseline	LoLCATs
Model		
Precision	16-bit (bfloat16)	
Sequence length	1024	
Linearizing attention	Linear + Sliding Window	
Linear attn feature map	Hedgehog	
Linear attn feature dimension	64 (effectively 128, see Table 2)	
Linear attn feature activation	Softmax (across feature dim)	
Sliding window implementation	Terrace	
Sliding window attn size	64	
Optimizer and LR Schedule		
Optimizer	AdamW	
Global batch size	8	
Gradient accumulation	8	
Gradient clipping threshold	1.0	
Learning rate schedule	Reduce LR on Plateau	
Stage 1: Attention Transfer		
Number of epochs	N/A	1
Tokens per epoch	N/A	20M
Learning rate	N/A	0.01
Stage 2: Low-rank Finetuning		
Number of epochs	1	
Tokens per epoch	20M	20M
Learning rate	1e-4	
LoRA rank and alpha	r =8, α =16	
LoRA dropout	0.0	
LoRA projections	$\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$	

Table 9: Hyperparameters for Llama 3.1 70B experiments.

- Baseline: We introduce LoRA parameters to the attention $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$ projection matrices. We train the linear attention feature maps, learnable mixing term γ, and the LoRA parameters during the fine-tuning adjustment stage of the linearization process.
- LoLCATs: We first perform block-wise attention transfer following Equation (8), with $k = 9$ as the block size. To perform attention transfer for block i, we save the hidden states outputted by block $i - 1$ to disk and then use this as training data for block i. We then introduce LoRA parameters to the attention $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$ projection matrices. During fine-tuning we only train the LoRA parameters, freezing the linear attention map and γ weights.

For the reported checkpoints, we train the layer-wise attention transfer stage using use a weighted combination of the MSE loss on attention outputs plus a cross-entropy loss between the softmax and linear attention maps. We use cross-entropy loss for the next-token prediction fine-tuning (Low-rank Finetuning) stage.

Hyperparameters. We include the hyperparameters for the baseline and LoLCATs approaches in Table 10. Since the baseline does not use attention transfer, we mark it with "N/A". We linearize each model using the same randomly sampled 20M tokens of the RedPajama pre-training corpus [14]. We pack the sequences to fill the full context length, and evaluate the best checkpoints based on validation set perplexity.

Compute Resources. We linearize the baseline using three NVIDIA 8 × 80GB H100 nodes, evaluating the best validation checkpoint after 19.5 hours. For LoLCATs, we perform attention transfer using a single

	Baseline	LoLCATs
Model		
Precision	16-bit (FP16)	
Sequence length	1024	
Linearizing attention	Linear + Sliding Window	
Linear attn feature map	Hedgehog	
Linear attn feature dimension	64 (effectively 128, see Table 2)	
Linear attn feature activation	Softmax (across feature dim)	
Sliding window implementation	Terrace	
Sliding window attn size	64	
Optimizer and LR Schedule		
Optimizer	AdamW	
Global batch size	8	
Gradient accumulation	8	
Gradient clipping threshold	1.0	
Learning rate schedule	Reduce LR on Plateau	
Stage 1: Attention Transfer		
Number of epochs	N/A	1
Tokens per epoch	N/A	20M
Learning rate	N/A	0.01
MSE and X-ent weights	N/A	1000, 1
Stage 2: Low-rank Finetuning		
Number of epochs	1	
Tokens per epoch	20M	20M
Learning rate	1e-4	
LoRA rank and alpha	$r = 4$, $\alpha = 8$	
LoRA dropout	0.5	
LoRA projections	$\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$	

Table 10: Hyperparameters for Llama 3.1 405B experiments.

80GB H100 GPU for 5 hours per block, and we finetune using three NVIDIA 8 × 80GB H100 nodes for 16 hours. We use PyTorch FSDP with activation checkpointing for distributed training.

B Additional Experiments

To better understand LoLCATs' properties and performance for high quality yet parameter and training-efficient linearizing, we now study how different amounts of parameter training, different data sources, and different amounts of training data affect LoLCATs quality.

B.1 Study on Parameter-Efficient Training

In our main results, we found that simple default initializations (*e.g.,* rank 8, applied to all attention projection layers) could recover high quality linearizing while only updating <0.2% of model parameters. In this section, we study how changing different aspects of low-rank adaptation and sliding window size impact linearizing performance.

B.1.1 Effect of LoRA Rank

We study the effect of LoRA rank in post-swap finetuning for zero-shot linearized LLM performance. Following standard implementations [27], we consider two factors: rank r, which determines the rank of the

LoRA Rank	PiQA	ARC-e	ARC-c (norm)	HellaSwag (norm)	Wino-grande	MMLU (5-shot)	Avg.	Avg. (no MMLU)
4	80.7	**82.5**	**56.0**	79.6	73.6	50.1	**71.3**	**74.5**
8	80.9	81.7	54.9	**79.7**	**74.1**	52.8	70.7	74.2
16	80.7	81.9	54.5	**79.7**	73.8	48.9	69.9	74.1
32	**81.1**	81.5	54.5	**79.7**	72.8	51.0	70.1	73.9
64	80.9	81.9	54.5	79.3	72.1	51.7	71.1	73.8
128	80.5	80.9	52.8	78.4	72.2	**53.4**	69.7	73.0
256	80.7	80.2	52.1	78.8	71.4	52.1	69.2	72.7

Table 11: **LoRA rank r comparison**. Evaluation on LM Evaluation Harness tasks. When adapting all projections $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$ after attention transfer, we find smaller ranks $r = 4, 8$ are surprisingly sufficient. Larger ranks with more parameter-heavy updates do not necessarily improve downstream performance.

low-rank matrices $\boldsymbol{A}, \boldsymbol{B}$ we decompose the weight deltas into; and alpha α, where α/r is a scaling factor that controls the degree to which $\boldsymbol{BA}$ affect the output (*i.e.,* LoRA output $y = \boldsymbol{W}x + \frac{\alpha}{r}\boldsymbol{BA}x$).

Setup. We sweep over ranks {4, 8, 16, 32, 64, 128, 256}, and adjust α such that $\alpha/r = 2$ as a default scaling factor [48]. For all runs, we linearize Llama 3 8B, and use the same default experimental setup as our main results. We start with the linear + "terrace" window attentions, training the feature maps via attention transfer over 20M tokens over Alpaca (2 full epochs). We then freeze feature maps and apply LoRA with the above ranks on all attention weight projections (freezing all other parameters such as those in MLPs or GroupNorms), finetuning for two more epochs over Alpaca using the hyperparameters in Table 8. We evaluate with LM Eval tasks.

Results. We report results in Table 11. We find that when applying LoRA to adjust in linearizing after attention transfer, larger rank updates, *e.g.,* $r = 128$ or 256, do not necessarily lead to improved zero or few-shot downstream performance. Somewhat surprisingly, using just $r = 4$ leads to overall best performance, while $r = 128$ and $r = 8$ achieve best and second-best 5-shot MMLU accuracy.

B.1.2 Effect of LoRA Projection Layer

We next compare performance when applying LoRA to different weight matrices of the linear attention layers. With the same training and evaluation setup as Appendix B.1.1, but fixing $r = 8, \alpha = 16$, we now apply LoRA to different combinations of $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$ weights after swapping in attention-transferred linear attentions. We use the same combination for each layer.

We report results in Table 12. Interestingly, when isolating for projections updated, LoRA on projections *not involved* in computing layer-wise attention weights (value $\boldsymbol{W}_v$ and output $\boldsymbol{W}_o$ projections) improves quality compared to query $\boldsymbol{W}_q$ or key projections $\boldsymbol{W}_k$. Somewhat surprisingly, updating just $\boldsymbol{W}_v$ or $\boldsymbol{W}_o$ achieves comparable performance to updating all projections (*c.f.*, average zeros-shot accuracy of 74.19% when updating just $\{\boldsymbol{W}_v\}$, 74.09% for $\{\boldsymbol{W}_o\}$ versus 74.24% updating $\{\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o\}$). Meanwhile, updating just $\boldsymbol{W}_q$ or $\boldsymbol{W}_k$ performs significantly worse (72.68%, 72.29%; versus 74.24%). This suggests much of the quality recovery in Stage-2 low-rank finetuning comes from adjusting values and outputs to the learned attention weights—as opposed to further refining attention weight computation. While best results do come from a combination of adapting either $\boldsymbol{W}_q$ or $\boldsymbol{W}_k$ with value and output projections, we may be able to achieve even more parameter-efficient linearizing—with comparable quality—by focusing on subsets with the latter.

B.1.3 Effect of Window Size

We now compare model performance using different window sizes in the LoLCATs linear + sliding window attention layer. With the standard sliding window implementation (Fig 9), we compare LM Eval performance

LoRA Projection	PiQA	ARC-e	ARC-c (norm)	HellaSwag (norm)	Winogrande	Average
$\boldsymbol{W}_q$	79.49	81.06	51.45	79.18	72.22	72.68
$\boldsymbol{W}_k$	79.82	79.80	50.68	78.54	72.61	72.29
$\boldsymbol{W}_v$	80.69	82.49	56.66	79.28	71.82	74.19
$\boldsymbol{W}_o$	80.09	81.65	55.63	79.35	73.72	74.09
$\boldsymbol{W}_q, \boldsymbol{W}_k$	79.76	81.19	51.02	79.29	72.22	72.70
$\boldsymbol{W}_q, \boldsymbol{W}_v$	80.96	81.90	54.35	79.01	72.30	73.70
$\boldsymbol{W}_q, \boldsymbol{W}_o$	**81.28**	81.86	54.78	79.20	73.72	74.17
$\boldsymbol{W}_k, \boldsymbol{W}_v$	80.74	82.62	**56.83**	79.26	71.98	74.28
$\boldsymbol{W}_k, \boldsymbol{W}_o$	80.69	82.15	55.46	79.53	73.01	69.33
$\boldsymbol{W}_v, \boldsymbol{W}_o$	80.69	**82.79**	55.89	79.57	71.59	74.10
$\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v$	80.90	82.20	56.48	79.13	73.95	**74.53**
$\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_o$	80.41	80.89	54.18	79.18	**74.35**	73.80
$\boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$	80.36	82.32	54.61	79.13	73.56	74.00
$\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$	80.85	81.73	54.86	**79.65**	74.11	74.24

Table 12: **LoRA projection comparison**. Evaluation on zero-shot LM Evaluation Harness tasks. We apply LoRA with the same rank $r = 8$ to different combinations of the attention projections, shading scores by increasing quality (darker is better). When isolating for projections updated, LoRA on projections *not involved* in layer-wise attention weight computations (value $\boldsymbol{W}_v$ and output $\boldsymbol{W}_o$ projections) achieves higher quality over query $\boldsymbol{W}_q$ or key projections $\boldsymbol{W}_k$. This suggests $\boldsymbol{W}_v, \boldsymbol{W}_o$ may be more important to adapt after attention transfer, although best results involve a combination across attention weight and output projections.

after linearizing with window sizes $w \in \{4, 16, 64, 256\}$. We also compare against the ThunderKittens-motivated "terraced" implementation used in our main experiments with window size 64. In Table 13, we find that in each of these settings, having more softmax attention generally improves performance (*c.f.*, $w =$ 16,64 versus $w =$4). However, more softmax attention does not always lead to better quality. Window size 256 results in up to an 8.8 point drop in 5-shot MMLU accuracy, suggesting we may not necessarily trade-off more softmax attention for higher quality. Comparing the standard sliding window and terracing implementations, we find similar performance (70.3 versus 70.7 average accuracy across tasks).

B.2 Study on Linearizing Data

While most of our work focuses on architecture and training procedure for improving LLM linearizing quality, we now study how data selection affects LoLCATs performance.

B.2.1 Data Source: Alpaca versus RedPajama

We study the effect of linearizing data for downstream LLM performance. While we initially found that just using the $\sim$ 50K samples of a cleaned Alpaca dataset[6] [55] could lead to surprisingly high performance on popular zero-shot LM Eval tasks, prior linearizing works [35] use more typical pretraining datasets to linearize such as RefinedWeb [38]. We thus also try linearizing with a random subset of RedPajama [14] to evaluate how LoLCATs works with pretraining data, albeit without any special curation. For both setups, we pack samples into 1024 token sequences and randomly subsample the RedPajama data so that we use the same number of training tokens (20M) for both attention transfer and finetune stages (40M tokens overall). We use the setup as described in Appendix A.1 for all other hyperparameters.

In Table 14, we find that across Mistral 7B (v0.1) and Llama 3 8B, using the Alpaca cleaned dataset actually leads to better downstream task quality for all tasks except for 5-shot MMLU, where linearizing with RedPajama consistently leads to $\sim$ 2 percentage point improvements. LoLCATs with both of these datasets leads to comparable or higher performance than prior methods trained on 2500$\times$ the data (*c.f.*,

[6] https://huggingface.co/datasets/yahma/alpaca-cleaned

Window Size	PiQA	ARC-e	ARC-c (norm)	HellaSwag (norm)	Winogrande	MMLU (5-shot)	Avg.	Avg. (no MMLU)
4	80.7	81.4	55.8	76.6	72.1	40.8	67.9	73.3
16	80.5	**82.2**	**56.0**	78.2	73.9	**50.3**	70.2	74.1
64	**80.7**	81.7	54.7	**79.1**	**75.3**	**50.3**	**70.3**	**74.3**
256	80.6	81.8	55.0	75.6	74.9	41.5	68.3	73.6
TK 64	80.9	81.7	54.9	79.7	74.1	52.8	70.7	74.2

Table 13: **Window size comparison**. We ablate the window size in LoLCATs linear + sliding window attention. For each window size w, the layer applies softmax attention to the w-most recent positions, combined with Hedgehog linear attention applied for all prior positions (Eq. 7, Figure 9). We compare $w \in \{4, 16, 64, 256\}$ using the standard sliding window implementation with our default $w = 64$ terraced window setup motivated by ThunderKittens (TK 64). Window size 64 performs best, where both implementations perform comparably.

Model	Linearizing Dataset	PiQA	ARC-e	ARC-c (acc. norm)	HellaSwag (acc. norm)	Wino-grande	MMLU (5-shot)	Avg.	Avg. (no MMLU)
Mistral 7B (v0.1)	-	82.1	80.9	53.8	81.0	74.0	62.4	72.4	74.4
→ LoLCATs (Ours)	Alpaca Clean	**81.5**	**81.7**	**54.9**	**80.7**	**74.0**	51.4	**70.7**	**74.5**
→ LoLCATs (Ours)	RedPajama	80.1	77.6	49.0	80.3	71.7	**53.2**	68.6	71.7
Llama 3 8B	-	79.9	80.1	53.3	79.1	73.1	66.6	72.0	73.1
→ LoLCATs (Ours)	Alpaca Clean	**80.9**	**81.7**	**54.9**	**79.7**	**74.1**	52.8	**70.7**	**74.2**
→ LoLCATs (Ours)	RedPajama	78.9	79.0	52.0	78.1	72.6	**55.2**	69.3	72.1

Table 14: **Linearizing data comparison**. For linearizing Mistral 7B (v0.1) and Llama 3 8B, LoLCATs with Alpaca and RedPajama subsets perform comparably (c.f. prior methods, Table 4), though we find that Alpaca actually leads to higher accuracy for most tasks other than 5-shot MMLU.

Table 4; SUPRA trained on 100B tokens gets Avg. accuracy of 64.0%), suggesting that LoLCATs can robustly improve linearizing quality over different data sources.

B.2.2 Sample Lengths: Effect of Effective Sequence Lengths

We further study the impact of sample sequence length for linearizing quality. By default, for linearizing data we pack original data samples into sequences of consistent length, *e.g.,* 1024 tokens. As done in prior work [47], this allows us to pack multiple short data samples together into longer training sequences, improving training efficiency and removing any padding tokens. However, it may also introduce situations where our linearizing sequences only carry short-context dependencies, *i.e.,* because we pack together many samples with few tokens, or split longer samples into multiple sequences. Especially with attention transfer, linearized LLMs may model longer samples less well (*e.g.,* the 5-example in-context samples in 5-shot MMLU) because we never learn to approximate attentions over "long enough" sequence lengths.

Effective sequence length. To study this data effect, we define an "effective sequence length" (ESL) metric. This roughly captures for each query how far back a layer needs to attend to capture all non-zero softmax attention weights. For query at position i, we define the ESL per query as

$$\text{ESL}(\boldsymbol{q}_i) := \sum_{j=1}^{i} (i-j) \frac{\exp(\boldsymbol{q}_i^\top \boldsymbol{k}_j / \sqrt{d})}{\sum_{m \leq j} \exp(\boldsymbol{q}_i^\top \boldsymbol{k}_m / \sqrt{d})} \tag{9}$$

We compute a sample's ESL per head as the sum over all query ESLs, *i.e.,* $\sum_{i=1}^{n} \text{ESL}(\boldsymbol{q}_i)$ for a sample with n tokens. We average this over all heads and layers to measure a sample's overall ESL.

We hypothesize that if our linearizing data only has samples with shorter ESL than those encountered at test time, then we would poorly model these test samples. Conversely, we may be able to improve linearizing

Linearizing Data	PiQA	ARC-easy	ARC-challenge	HellaSwag (acc. norm)	Winogrande (acc. norm)	MMLU (5-shot)	Avg. (no MMLU)
Alpaca	**80.9**	**81.7**	**54.9**	**79.7**	**74.1**	52.8	**74.2**
RedPajama	78.9	79.0	52.0	78.1	72.6	55.2	72.1
RedPajama (Sample Top ESL)	78.4	77.0	49.8	78.0	71.4	**56.5**	70.9

Table 15: **Effect of ESL on linearized LLM quality**, Llama 3 8B. While linearizing with longer ESLs—*e.g.*, RedPajama samples or specifically filtering for top ESLs in RedPajama (Sample Top ESL)—improves 5-shot MMLU accuracy up to 3.7 points, it reduces quality on all other evaluated LM Eval tasks by 1.7 to 5.0 points.

quality by specifically filtering for samples with longer ESL. We report two findings next.

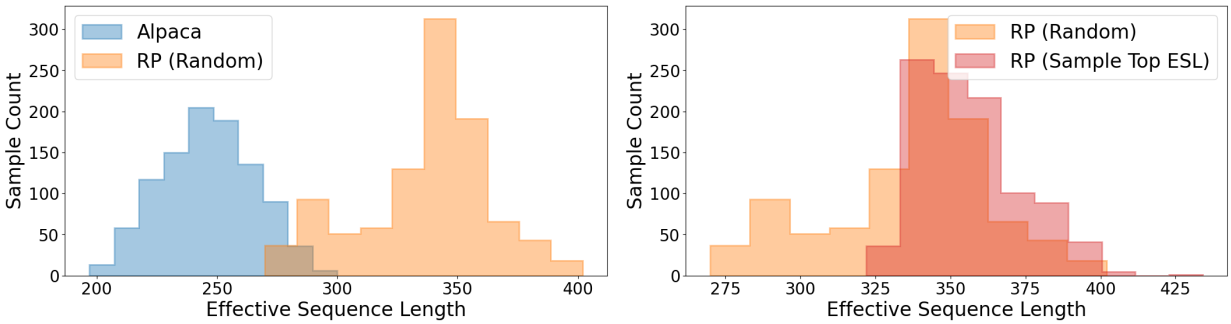


Figure 10: **Effective sequence length distributions**. Although we pack data samples into 1024-token training sequences, different data sources can vary in effective sequence lengths (left). Furthermore, we can selectively filter for longer ESL samples (right). We find linearizing with longer ESLs coincides with improved MMLU scores, albeit at the cost of other LM Eval tasks (Table 15).

Finding 1: ESL corresponds with RedPajama versus Alpaca performance. In Figure 10, we first plot the distribution of sample ESLs computed with Llama 3 8B on Alpaca and RedPajama linearizing data subsets. We find RedPajama samples on average display longer ESLs, which conincides with improved MMLU score (*c.f.*, Table B.2.1).

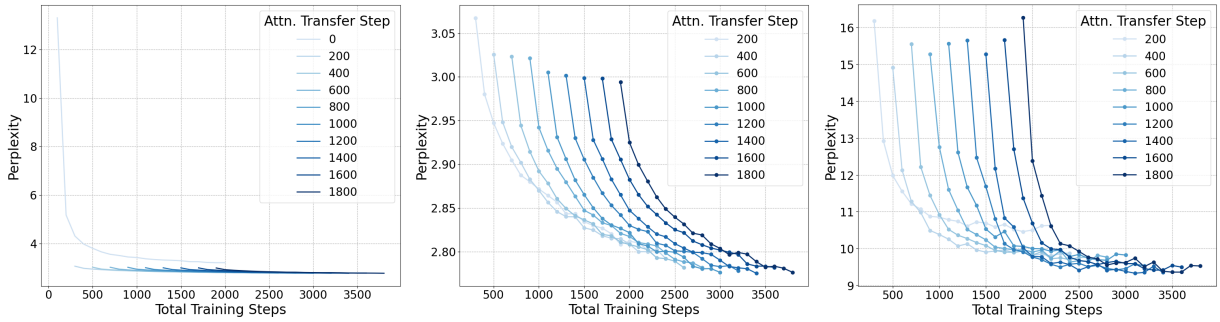
Finding 2: Filtering for higher ESL improves MMLU. Furthermore, we can increase the ESLs in linearizing data by actively filtering for high ESL samples (Figure 10 right). Here we actively filter for the top 20,000 packed RedPajama samples with the highest ESLs, amounting to 20M tokens. When doing one epoch of attention transfer and low-rank linearizing with this subset, we further improve MMLU accuracy by 1.3 points (Table 15). However, this comes at a cost for all other LM Eval tasks, dropping quality compared to random RedPajama packing by 0.2 to 2.1 points.

B.3 Study on Linearizing Token Budget

We further study how varying the number of tokens used for both attention transfer and low-rank adaptation impacts LoLCATs linearizing quality.

Impact of minimal tokens. To first test how efficient we can be with attention transfer, we linearize Llama 3 8B with varying numbers of attention transfer steps (0 - 1800), before low-rank adjusting for up to 2000 steps. We use the Alpaca dataset and the same packed random sampling as our main experiments, and measure evaluation perplexity on validation samples both in-distribution (held-out Alpaca samples) and out-of-distribution (RedPajama validation samples) over different combinations of steps (Figure 12). Without attention transfer, low-rank adaptation converges significantly higher on in-distribution samples (Figure 11a), suggesting poorer quality linearizing. However, we find similar held-out perplexities after relatively few attention transfer steps (*c.f.*, 1000 - 1800 updates, the former amounting to just 8 million tokens for attention transfer), where all runs improve in-distribution PPL by ∼0.23 points after LoRA finetuning for 2000 steps.

In Table 16, we report the numerical values for held-out perplexities at the end of linearizing (1800 attention transfer steps + 2000 low-rank adaptation steps), as well as the average LM Eval score over zero-shot tasks. We similarly find competitive generalized zero-shot quality with relatively few attention transfer steps (200 steps), all achieving 7.70–8.16 higher points than the next best Mamba-Llama model (0.16–0.62 higher points than the 50% softmax attention variant, *c.f.*, Table 4). Without any attention transfer, linearized LLMs perform drastically worse on out-of-distribution samples (Table 16, RedPajama and LM Eval metrics).



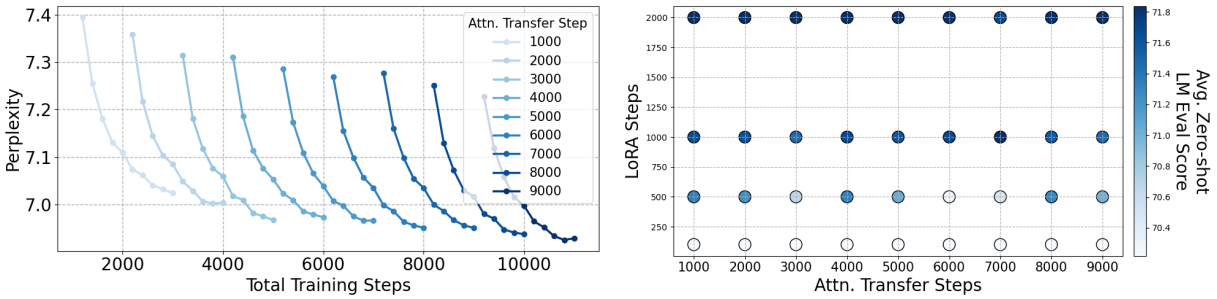
(a) Alpaca PPL (b) Alpaca Val. PPL (Zoomed) (c) RedPajama Val. PPL (Zoomed)

Figure 11: **Evaluation curves over number of training steps**. Llama 3 8B linearized with Alpaca. We report the impact of steps allotted to attention transfer versus LoRA linearizing, using validation set perplexity (PPL) over both in-distribution (held-out Alpaca samples) and out-of-distribution (RedPajama validation samples). We first run linearizing with 0 - 1800 attention transfer steps, before LoRA-finetuning for up to 2000 steps. Without any attention transfer (0 steps), linearized LLMs get much higher perplexity (Fig 11a). On the other hand, we observe similar convergence after attention transfer over only 1000 steps.

Attn. Transfer Steps	0	200	400	600	800	1000	1200	1400	1600	1800
Alpaca Eval PPL	3.211	2.802	2.792	2.782	2.782	2.776	2.778	**2.775**	2.782	2.776
RedPajama Eval PPL	61.305	10.459	9.799	9.699	9.628	9.679	9.420	**9.328**	9.413	9.358
Avg. Zero-shot LM Eval Acc.	56.86	73.68	73.34	73.70	73.66	73.74	73.47	73.70	**73.80**	73.66

Table 16: **Effect of attention transfer steps**. With Llama 3 8B linearized on Alpaca data, we report the final evaluation perplexities after 2000 LoRA steps in Fig 12, as well as downstream LM Eval performance averaged over zero-shot tasks. We again find competitive quality with relatively few attention transfer steps.

Impact of more pretraining data. We finally study how linearizing over larger amounts of pretraining data impacts quality. We randomly sample a larger set of unique RedPajama training sequences (1024-token packed; 72,000 such samples overall), allowing us to linearize Llama 3 8B with different combinations of up to 9,000 attention transfer updates and 2,000 low-rank linearizing updates. To test language modeling recovery, we report both held-out validation sample perplexity (Table 12a) and general zero-shot LM Evaluation Harness quality (Table 12b). Increasing both Stage 1 attention transfer steps and Stage 2 low-rank adjusting steps notably improves validation perplexity. However, we similarly find competitive zero-shot LM scores across all evaluated attention transfer steps. Across checkpoints at different numbers of low-rank updates, attention transfer with up to 9× more unique tokens does not seem to monotonically improve downstream quality (Table 12b). Meanwhile, we do find that across various amounts of attention transfer steps, subsequent low-rank adaptation consistently improves average zero-shot LM score by >1 points.



(a) RedPajama Validation Perplexity

(b) LM Eval Score

Figure 12: **Evaluation quality from ablating RedPajama linearizing updates**.

B.4 Study on Block-wise Attention Transfer

This section studies how the properties of attention transfer change with model scale, motivating LoLCATs's block-wise training approach (Section 3.3.2).

Attention transfer involves training the LoLCATs linear attentions to match the outputs of softmax attention at each layer by minimizing the MSE between the softmax and linear attention outputs. Since the MSE loss is scale-sensitive and the magnitude of the MSE loss varies across layers (Figure 6b, we hypothesize that jointly training all 126 Transformer layers in Llama 3.1 405B – by summing the MSE losses across all layers – may be difficult. Correspondingly, in Table 3, we find that block-wise attention transfer leads to lower language modeling perplexity for 405B linearized LLMs compared to joint training. However, we find that joint training is sufficient and performs similarly to block-wise training at the smaller scales (8B, 70B).

We study attention transfer across model scales and find that the variation in MSE magnitudes correlates with the benefit of block-wise training. In Tables 17 and 18, we report magnitude of the MSE loss as the depth of the layers in the block increases. We find that the variation in MSE magnitudes increases with the model size: while the largest MSEs at the 70B scale are between $2.5 - 3$, the MSE of the final block at the 405b scale is 48.66 (over $13\times$ that of the next largest MSE by block, 3.62).

Block (Layer range)	Eval MSE
0-4	$8e-4$
5-9	0.03
10-14	0.06
15-19	0.10
20-24	0.28
25-29	0.73
30-34	0.28
35-39	0.28
40-44	0.25
45-49	1.08
50-54	0.26
55-59	0.18
60-64	0.45
65-69	0.50
70-74	2.91
75-79	2.56

Table 17: **Attention transfer block-wise MSE** We report the eval MSE by 5-layer block for each of the 16 blocks in the 80 Transformer layer Llama 3.1 70B model. Each block is trained on the exact same set of RedPajama data at sequence length 1024.

Block (Layer range)	Eval MSE
0-8	0.03
9-17	0.02
18-26	0.09
27-35	0.04
36-44	0.36
45-53	0.42
54-62	1.59
63-71	2.75
72-80	3.62
81-89	2.49
90-98	0.29
99-107	1.11
108-116	3.42
117-126	48.66

Table 18: **Attention transfer block-wise MSE** We report the eval MSE by 9-layers block for the 14 blocks in the 126 Transformer layer Llama 3.1 405B model. Each block is trained on the exact same set of RedPajama data at sequence length 1024.

C Implementation Details

C.1 Code Implementation

Below we provide further details on implementing LoLCATs with PyTorch-like code and example demonstrations from the HuggingFace Transformers library.

Learnable Linear Attention. To start, we simply replace the softmax attentions in an LLM with a linear attention. We define such a class below.

```python
import copy
import torch.nn as nn

class LolcatsLlamaAttention(nn.Module):
    def __init__(self,
                 feature_dim: int,
                 base_attn: nn.Module,  # original Transformer attn.
                 ) -> None:
        super().__init__()

        # Inherit pretrained weights
        self.q_proj = base_attn.q_proj
        self.k_proj = base_attn.k_proj
        self.v_proj = base_attn.v_proj
        self.o_proj = base_attn.o_proj

        # Inherit other attention things
        self.rotary_emb = base_attn.rotary_emb
        self.base_attn  = base_attn  # keep for attention transfer
        self.num_heads  = base_attn.num_heads
        self.head_dim   = base_attn.head_dim

        # Initialize feature maps, see Hedgehog definition below
        self.feature_map_q = HedgehogFeatureMap(
            self.num_heads, self.head_dim, feature_dim
        )
        self.feature_map_k = copy.deepcopy(self.feature_map_q)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Compute linear attention (assume no GQA)
        (b: batch_size, h: num_heads, l: seq_len, d: head_dim)
        """
        q = self.q_proj(x)  # assume all are (b, h, l, d)
        k = self.k_proj(x)
        v = self.v_proj(x)

        # Apply rotary embeddings
        q = self.rotary_emb(q)
        k = self.rotary_emb(k)

        # Apply feature maps
        q = self.feature_map_q(q)  # (b, h, l, feature_dim)
        k = self.feature_map_k(k)  # (b, h, l, feature_dim)

        # Compute linear attention
        kv = torch.einsum('bhlf,bhld->bhfd', k, v)
        y  = torch.einsum('bhlf,bhfd->bhld', q, kv)
        y /= (torch.einsum('bhlf,bhf->bhl', q, k.sum(dim=2))
              + self.eps)[..., None]

        # Apply output projection
        return self.o_proj(rearrange('bhld -> blhd', y))
```

Listing 1: LoLCATs Linear Attention Class

Hedgehog Feature Map. We implement the Hedgehog feature map following Zhang et al. [66].

```python
import torch.nn as nn

class HedgehogFeatureMap(nn.Module):
    def __init__(self,
                 num_heads = 32: int,  # defaults for 8B LLMs
                 head_dim = 128: int,
                 feature_dim = 64: int,
                 ) -> None:
        super().__init__()
        self.num_heads = num_heads
        self.head_dim = head_dim
        self.feature_dim = feature_dim

        # Initialize trainable feature map weights
        self.weights = nn.Parameter(
            torch.zeros(self.num_heads, self.head_dim, self.feature_dim)
        )

    def self.activation(self: torch.Tensor) -> torch.Tensor:
        """Softmax across feature dims activation"""
        return torch.cat([
            torch.softmax(x, dim=-1), torch.softmax(-x, dim=-1)
        ])

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Assume x.shape is (b, h, l, d)
        (b: batch_size, h: num_heads, l: seq_len, d: head_dim)
        """
        x = torch.einsum('hdf,bhld->bhlf', self.weights, x)
        return self.activation(x)
```

Listing 2: Hedgehog Feature Map

Linearizing LLM Setup. To initialize an LLM for linearizing, we simply replace each softmax attention in the Transformer's layers with our LoLCATs linear attention class. We illustrate this with a Huggingface Transformer's class below.

```python
from transformers import AutoModelForCausalLM

def convert_model(model: AutoModelForCausalLM):
    """Setup linearizing attentions"""
    for layer in model.model.layers:
        layer.self_attn = LolcatsLlamaAttention(feature_dim=64,
                                                base_attn=layer.self_attn)
    return model
```

Listing 3: Linearizing LLM Setup

Attention Transfer Training. Finally, we can train LoLCATs layers in a simple end-to-end loop. Although doing this attention transfer is akin to a layer-by-layer cross-architecture distillation, due to architectural similarities we implement linearizing with the same footprint as finetuning a single model. Furthermore, as we freeze all parameters except for the newly introduced feature map weights, this amounts to *parameter-efficient* finetuning, training <0.2% of a 7B+ LLM's parameters.

```
"""
Example attention transfer training loop for Llama 3.1 8B
"""
import torch.nn as nn
from transformers import AutoModelForCausalLM

# Load Llama 3.1 8B
model_config = {
    'pretrained_model_name_or_path': 'meta-llama/Meta-Llama-3.1-8B'
}
model = AutoModelForCausalLM.from_pretrained(**model_config)

# Freeze all pretrained weights
for p in model.parameters():
    p.requires_grad = False

# Prepare LoLCATs linearizing layers
model = convert_model(model)

# Setup MSE loss criterion
mse_loss = nn.MSELoss()

# Get some linearizing data
train_loader = load_data(**data_kwargs)

# Train LoLCATs layers via attention transfer
for ix, input_ids in enumerate(train_loader):
    loss = 0                              # Attention transfer loss here
    x = model.embed_tokens(input_ids)  # Input embeddings from tokens

    for layer in model.layers:         # Forward pass thru model

        # Start Attention
        _x = layer.input_layernorm(x)  # Just Llama things

        ## Attention Transfer part ##
        with torch.no_grad():
            y_true = layer.self_attn.base_attn(_x)
        y_pred = layer.self_attn(_x)
        loss += mse_loss(y_pred, y_true)  # Add layer-wise MSE losses

        _x = y_true                           # Pass true attention outputs
                                              # thru to rest of model
        x = _x + x
        # End Attention

        # Start MLP
        _x = layer.post_attention_layernorm(x)
        _x = self.mlp(_x)
        x = _x + x
        # End MLP

    loss.backward() # End-to-end attention transfer
```

Listing 4: End-to-end attention transfer pseudocode.

C.2 Hardware-aware implementation of LoLCATs sliding window

Despite the theoretical efficiency of linear attention, existing implementations have long underperformed well-optimized attention implementations (e.g., FlashAttention) in wall clock speed [17]. To translate the benefits of LoLCATs to wall clock speedups, we develop a custom hardware-aware algorithm for LoLCATs **prefill** using the ThunderKittens CUDA framework.[7] We first briefly review the GPU execution model and then detail our algorithm.

C.2.1 GPU execution model

GPUs workloads are executed by independent streaming multiprocessors (SMs), which contain warps, groups of 32 threads, that operate in parallel.

Memory hierarchy. ML workloads involve moving large tensors (weights, activations) in and out of memory to perform computation. GPUs have a memory hierarchy, which includes global memory (HBM), shared memory (SRAM), and registers. Reading from and writing data to memory, referred to as I/O operations, takes time. There is a large amount of HBM, which has high I/O costs, and a small amount of SRAM and registers have much costs. All SMs access global memory, warps within an SM threadblock can access shared memory, and threads within a threadblock have independent register memory. To reduce the I/O costs, **locality** is key – kernels should perform as many operations as possible on data that has already been loaded into fast memory (*i.e.*, thread registers) before writing the results back to slower memory.

Compute units. GPUs have increasingly heterogeneous compute units on newer generations of hardware. Tensor cores—specialized compute units for matrix-matrix multiplications—are the fastest units, operating at 1.0 PetaFLOPS on Nvidia H100 GPUs in contrast to 67 TeraFLOPS for the general non Tensor core units. ML workloads should thus ideally **exploit the tensor cores**.

Cost model. Overall, workloads may either be compute or memory bound, depending on whether they are bottlenecked by the compute speed or I/O costs. To *hide* latencies from either expensive compute or I/O, a classic principle in systems is to **pipeline computation** among parallel workers.

C.2.2 ThunderKittens CUDA kernel for prefill

We describe our overall approach below and provide pseudocode in Algorithm 1, designed around the three principles above: memory locality, tensor core utilization, and pipelined execution.

The kernel fuses the entire LoLCATs layer, taking as input the attention queries, keys, and values, for $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v} \in \mathbb{R}^{N \times d}$ with sequence length N and head dimension d and outputting the result of the $\boldsymbol{y} \in \mathbb{R}^{N \times d}$. Following Llama 3 [3], we let $d = 128$ in the discussion below.

Pipeline execution overview. Each thread block handles a single batch and head element of size $N \times d$. The kernel loops over chunks of length 64 along the sequence dimension N, loading 64×128 tiles of $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}$, which we'll refer to as $\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t$, in each iteration t. We use 8 warps (workers) per thread block, splitting them into two groups of 4 workers that pipline the computation. One "warpgroup" is in charge of launching memory loads and stores and computing the relatively cheap terraced window attention, while the other focuses on computing the more expensive linear attention computation and recurrent state updates.

Warpgroup 1 (Window attention). For the diagonal 64×128 sized tiles, recall that the output is simply the window attention result. At iteration t, the warpgroup loads $\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t$ into thread registers, use the tensor cores to multiply queries and keys, apply a causal mask, apply the Softmax, and use the tensor cores to multiply the attention scores with the values. Note that we can use Nvidia's new waprgroup operations (e.g., WGMMA) introduced in the H100 architecture to perform these operations. We refer to the terraced window output as $\textbf{terrace}_{\mathbf{o}}$.

[7] https://github.com/HazyResearch/ThunderKittens

Because the window attention is relatively cheap, waprgroup 1 also helps handle loads and stores between HBM and SRAM for the entire kernel. We use tensor memory acceleration (TMA), a new H100 capability for asynchronous memory movement, to perform these loads and stores.

Warpgroup 2 (Linear attention). We briefly review the linear attention equation. The formulation on the left shows a **quadratic** view, wherein $\phi(\boldsymbol{q}_n)^\top$ and $\phi(\boldsymbol{k}_i)$ are multiplied first, while the right formulation shows a **linear** view, wherein $\boldsymbol{k}_i$ and $\boldsymbol{v}_i^\top$ are multiplied first.

$$\hat{\boldsymbol{y}}_n = \sum_{i=1}^{n} \frac{(\phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i))\boldsymbol{v}_i}{\sum_{i=1}^{n} \phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i)} = \frac{\phi(\boldsymbol{q}_n)^\top \left(\sum_{i=1}^{n} \phi(\boldsymbol{k}_i)\boldsymbol{v}_i^\top \right)}{\phi(\boldsymbol{q}_n)^\top \sum_{i=1}^{n} \phi(\boldsymbol{k}_i)} \tag{10}$$

Since LoLCATs uses *no linear attention* on the diagonal tiles, the linear attention contribution for tile t is as follows, where queries are multiplied by the cumulative KV state from the prior iterations up to $t-1$:

$$\hat{\boldsymbol{y}}_t \frac{\phi(\boldsymbol{q}_t)^\top \left(\sum_{i=1}^{t-1} \phi(\boldsymbol{k}_i)\boldsymbol{v}_i^\top \right)}{\phi(\boldsymbol{q}_n)^\top \sum_{i=1}^{t-1} \phi(\boldsymbol{k}_i)} \tag{11}$$

At $t = 0$, the KV state and K state are initialized to 0, maintained in waprgroup 2's registers.

At iteration t, warpgroup 2 loads in the learned feature maps into register and computes $\boldsymbol{qf}_t$. This result gets multiplied by the running KV state so far up until $t-1$ (again 0 at iteration 0), and the result, $\mathbf{linear_o}$ gets written to SMEM.

The warps then update the KV state to prepare for the next iteration by featurizing $\boldsymbol{k}_t$ to $\boldsymbol{kf}_t$ using the learned feature map, and multiplying by $\boldsymbol{v}_t$ with WGMMA operations. Note that because the KV state in linear attention is somewhat large ($d \times d$), we leave the state *in register* throughout the kernel execution to avoid I/O costs.

Combining the results. Warpgroup 1 loads the $\mathbf{linear_o}$ contribution from SMEM to its registers, adds the $\mathbf{terraced_o}$ component, normalizes the overall result, and stores it back to HBM using TMA asynchronous store operations. We provide pseudocode in Algorithm 1.

To recap, our overall algorithm uses three classical systems ideas to run efficiently: (1) pipelining the different attention and I/O operations, (2) keeping the fastest compute—the tensor cores—occupied, and (3) keeping the recurrent (KV) state local in fast memory (thread registers).

Algorithm 1 LoLCATs ThunderKittens prefill kernel

Input: Attention queries, keys, and values $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v} \in \mathbb{R}^{N \times d}$ for head dimension d and sequence length N
Output: LoLCATs attention output $\boldsymbol{o} \in \mathbb{R}^{N \times d}$

Let $\mathbf{local}_{KV}$ be the cumulative recurrent state ("KV-state") initialized to 0 in waprgroup 2's registers.

for $t \leftarrow 0$ to $\frac{N}{64}$ **do**

▷ Warpgroup 1, Terraced attention

Load $\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t \in \mathbb{R}^{64 \times d}$ from HBM to SMEM.

Compute the terraced attention output tile $\mathbf{terrace}_{\mathbf{o}t} \in \mathbb{R}^{64 \times d}$ in register using WGMMA operations.

▷ Warpgroup 2, Linear attention

Featurize $\boldsymbol{q}_t$ by multiplying with the learned feature map to obtain $\boldsymbol{qf}_t$

Compute the linear attention output tile $\mathbf{linear}_{\mathbf{o}t} \in \mathbb{R}^{64 \times d}$ in register, using $\boldsymbol{qf}_t$ and $\mathbf{local}_{KV}$.

Write $\mathbf{linear}_{\mathbf{o}t}$ from register to SMEM

Featurize $\boldsymbol{k}_t$ by multiplying with the learned feature map to obtain $\boldsymbol{kf}_t$

Update $\mathbf{local}_{KV}$ by multiplying $\boldsymbol{kf}_t$ and $\boldsymbol{v}_t$, and adding the result to $\mathbf{local}_{KV}$ in place, all in register

▷ Warpgroup 1, Combine results

Load $\mathbf{linear}_{\mathbf{o}t}$ from SMEM to register

Add $\boldsymbol{o}_t = \mathbf{linear}_{\mathbf{o}t} + \mathbf{terrace_o}$ in register

Write $\boldsymbol{o}_t$ to HBM

D Extended Related Work

D.1 Linearizing Transformers

In this work, we build upon both approaches explicitly proposed to linearize LLMs [35], as well as prior methods focusing on smaller Transformers reasonably adaptable to modern LLMs [29, 34, 66]. We highlight two approaches most related to LoLCATs and their extant limitations next.

Scalable UPtraining for Recurrent Attention (SUPRA). Mercat et al. [35] linearize LLMs by swapping softmax attentions with linear attentions similar to Retentive Network (RetNet) layers [54], before jointly training all model parameters on the RefinedWeb pretraining dataset [38]. In particular, they suggest that linearizing LLMs with the vanilla linear attention in Eq. 2 is unstable, and swap attentions with

$$\hat{\boldsymbol{y}}_n = \text{GroupNorm}\Big(\sum_{i=1}^{n} \gamma^{n-i} \phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i) \boldsymbol{v}_i\Big) \tag{12}$$

GroupNorm [61] is used as the normalization in place of the $\sum_{i=1}^{n} \phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i)$ denominator in Eq. 2, γ is a decay factor as in RetNet, and ϕ is a modified *learnable* feature map from Transformer-to-RNN (T2R) [29] with rotary embeddings [53]. In other words, $\phi(\boldsymbol{x}) = \text{RoPE}(\text{ReLU}(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}))$ with $\boldsymbol{W} \in \mathbb{R}^{d \times d}$ and $\boldsymbol{b} \in \mathbb{R}^d$ as trainable weights and biases. With this approach, they recover zero-shot capabilities in linearized Llama 2 7B [57] and Mistral 7B [28] models on popular LM Evaluation Harness [21] and SCROLLS [50] tasks.

Hedgehog. Zhang et al. [66] show we can train linear attentions to approximate softmax attentions, improving linearized model quality by swapping in the linear attentions as learned drop-in replacements. They use the standard linear attention (Eq. 2), where query, key, value, and output projections (the latter combining outputs in multi-head attention [58]) are first copied from an existing softmax attention. They then specify learnable feature maps $\phi(\boldsymbol{x}) = [\text{softmax}(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}) \oplus \text{softmax}(-\boldsymbol{x}\boldsymbol{W} - \boldsymbol{b})]$ (where $\oplus$ denotes concatenation, and both $\oplus$ and the softmax are applied over the *feature dimension*) for $\boldsymbol{q}$ and $\boldsymbol{k}$ in each head and layer, and train ϕ such that linear attention weights $\hat{\boldsymbol{a}}$ match a Transformer's original softmax weights $\boldsymbol{a}$. Given some sample data, they update ϕ with a cross-entropy-based distillation to minimize:

$$\mathcal{L}_n = -\sum_{i=1}^{n} \frac{\exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d})}{\sum_{i=1}^{n} \exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d})} \log \frac{\phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i)}{\sum_{i=1}^{n} \phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i)} \tag{13}$$

as the softmax and linear attention weights are both positive and sum to 1. As they focus on task-specific linearization (*e.g.,* GLUE classification [59] or WikiText-103 language modeling [36]), for both attention and model training they use task-specific training data. By doing this "attention distillation", they show significant linearized quality improvements over T2R mainly on smaller Transformers (*e.g.,* 110M parameter BERTs [19] and 125M GPT-2s [46]). They further show LLM linearizing by linearizing Llama 2 7B for a specific SAMSum summarization task [22].

D.2 Efficient Architectures

Subquadratic Attention Alternatives. Many prior works study more efficient sequence modeling modules compared to Transformer self-attention, commonly training proposed architectures from scratch. While our work is most similar to and compatible to linear attentions [11, 13, 30, 44, 49, 62, 63, 66] and sparse attentions [6, 9, 65], other efficient architectures include convolutional models [25, 42], recurrent neural nets (RNNs) (not strictly following the attention inductive bias) [5, 18, 39], and state-space model (SSM)-inspired neural net layers [16, 24, 25].

Improving LLMs for Long Contexts. We share motivation various works focus on improving context length quality [1, 10, 41]. They that the rotary embeddings [53] of popular LLMs [3, 7, 28, 57] struggle to handle contexts longer than those used in training. [12] notably study the training efficiency of this via LoRA. Our work is complementary, where we reduce the efficiency bottleneck by replacing self-attentions.

Further work lies in how to improve long context quality of linearized LLMs, *e.g.,* by integrating these methods. Other works also improve efficiency by optimizing the KV Cache from a systems standpoint [32]. These may present complementary optimizations to linear attention's algorithmic improvements.

E Additional Artifacts

E.1 Sample Generations

We include sample generations for various prompts, using Llama 3 8B models linearized with different LoLCATs-related linearizing setups on the Alpaca dataset. For reference, we also include responses from a standard (softmax attention) Llama 3 8B finetuned with LoRA (rank = 8, alpha = 16, applied to all attention projections) on the Alpaca training set. LoLCATs as presented often leads to greatest coherence among linearized LLMs.

Prompt

<|begin_of_text|> Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:
What is the difference between H&M and Balenciaga?

Response:

Hedgehog (Linear Only) (No LoRA)

Hindi: the difference is that in the first one is a little bit more complicated, but the other one is that is the same. <|end_of_text|>

Hedgehog (Linear Only) (Attention Transfer + LoRA)

H&M and Balenciaga are two of the most popular fashion brands in the world, but there are some differences between them.

H&M is a fast fashion retailer that sells trendy and affordable clothing. It was founded in Sweden in 1947 and is now available in over 60 countries worldwide. The brand is known for its fast fashion and affordable prices, making it a popular choice for those on a budget.

Balenciaga, on the other hand, is a luxury fashion house that was founded in 1977. It is known for its high-end ready-to-wear pieces, including leather jackets, dresses, and shoes. The brand is known for its use of luxurious fabrics and intricate designs, as well as its collaborations with other fashion houses such as Versace and Versace.

In summary, H&M and Balenci are two popular fashion brands that offer different styles and price points. While H&M is known for its fast fashion and affordable fashion, Balenciaga is known for its luxury and high-end ready-to-wear pieces.<|end_of_text|>

LoLCATs (Linear + Sliding Window) (No LoRA)

The difference between H&M and Balenciaga is that H&M is a high street brand while Balenciaga is a luxury brand. H&M is more affordable than Balenciaga, which is a luxury brand.<|end_of_text|>

LoLCATs (Linear + Sliding Window) (No Attention Transfer)

H&M and Balenciaga are both clothing brands, but there are some key differences between the two.

H&M is a fast fashion brand known for its affordable and trendy clothing. The company was founded in Sweden and has stores in over 60 countries. It is known for its seasonal collections, which include clothing, accessories, and home decor. H&M has a reputation for its ethical and sustainable practices, and the company works with designers and brands to create its collections.

On the other hand, Burberry is a British luxury fashion house founded in 1856. It is known for its high-end fashion, including clothing, accessories, and fragrances. The company has a strong heritage, and its designs are inspired by British culture and history. Burberry is known for its classic trench coats, and its signature check pattern is instantly recognizable.

Overall, both brands are high-end fashion companies, but they have different target audiences and design styles. H&M is more accessible and affordable, while Burberry is more expensive and known for its classic, timeless designs. <|end_of_text|>

LoLCATs (Linear + Sliding Window) (Attention Transfer + LoRA)

H&M and Balenciaga are both fashion brands, but they have significant differences.

H&M is a fast-fashion brand that offers a wide range of clothing, accessories, and home decor at affordable prices. The brand is known for its trendy and stylish designs, and it often collaborates with high-profile designers to create limited-edition collections. H&M is a mass-market retailer, and its products are available in its own stores, as well as in department stores and online.

Balenciaga, on the other hand, is a luxury fashion house founded in 1917 by Spanish designer Cristobal Balenciaga. The brand is known for its avant-garde designs, high-quality materials, and attention to detail. Balenciaga's collections are often bold, experimental, and daring, and the brand has a strong following among fashion enthusiasts and celebrities. H&M, on the other hand, is a fast-fashion retailer that offers affordable clothing and accessories inspired by the latest trends. While H&M also produces its own clothing and accessories, the brand is known for its collaborations with high-end designers and its ability to produce fashion at a much lower price point than other luxury brands.<|end_of_text|>

Softmax Attention

H&M and Balenciaga are two fashion brands that are quite different from each other.

H&M is a fast fashion brand that offers affordable clothing and accessories for both men and women. The brand is known for its trendy and on-trend designs, and its collections often feature collaborations with high-profile designers and celebrities. H&M is a mass-market brand that aims to make fashion accessible to everyone.

Balenciaga, on the other hand, is a luxury fashion brand that is known for its avant-garde designs and high-end materials. The brand is associated with couture and haute couture, and its collections often feature bold and experimental designs. Balenciaga is a high-end brand that caters to a more exclusive clientele.

In summary, the main difference between H&M and Balenciaga is their target audience, price point, and design philosophy. H&M is a fast fashion brand that offers affordable and trendy designs, while Balenciaga is a luxury brand that focuses on high-end and avant-garde designs.<|end_of_text|>

Prompt

<|begin_of_text|> Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:
Explain why the following fraction is equivalent to 1/4. 4/16

Response:

Hedgehog (Linear Only) (No LoRA)

The number 4/4 is a fraction. This is a fraction that is equivalent to 4/4. The number 4/4 is the number 4/4/4/4/4.<|end_of_text|>

Hedgehog (Linear Only) (Attention Transfer + LoRA)

The fraction 1/4 is equivalent to 4/16. To see why, we can multiply both the numerator and denominator by 4 to obtain the equivalent fraction 16/16. This is because the multiplication of two equal numbers will not change the value of the fraction.

The number 4/16 can be rewritten as 16/4, which is equivalent to 4/4. Similarly, the numerator 4/4 can be rewritten as 4/4. Therefore, the fraction 4/4 is equivalent to 4/16, which is the same as 1/4. <|end_of_text|>

LoLCATs (Linear + Sliding Window) (No LoRA)

The fraction 4/16 is equivalent to 1/4 because the numerator and denominator are both divisible by 2. This is because 4/16 is equal to 4/16. This is because the numerator and denominator are both divisible by 2. This means that 4/16 is equal to 1/4. Therefore, 1/4 is equal to 1/4, which is 1/4. Therefore, 4/16 is equal to 1/4, and 1/4 is equal to 1/4.<|end_of_text|>

LoLCATs (Linear + Sliding Window) (No Attention Transfer)

1/4 = 4/16

To explain why the fraction 1/4 is equivalent to 4/16, we first need to understand what a fraction is. A fraction is a way to represent a part of a whole, where the whole is divided into equal parts. In this case, the whole is the whole number 4, which is divided into 4 equal parts, each of which is represented by the fraction 1/4. The symbol used to represent a fraction is a fraction bar, which is a horizontal line that goes between the numerator and the denominator. So, the fraction 1/4 can be written as 1/4 or 4/4.<|end_of_text|>

LoLCATs (Linear + Sliding Window) (Attention Transfer + LoRA)

The fraction 4/16 is equivalent to 1/4 because both fractions represent the same value. When simplifying 4/16, we can divide both the numerator and denominator by 4 to get 1/4. This is because 4 divided by 4 is equal to 1, and 16 divided by 4 is also equal to 4. So, we can cancel out the 4s to get 1/4.<|end_of_text|>

Softmax Attention

The fraction 4/16 is equivalent to 1/4 because both fractions represent the same value. To see why this is the case, we can simplify the fraction 4/16 by dividing both the numerator and the denominator by their greatest common factor.

The greatest common factor (GCF) of 4 and 16 is 4. We can simplify the fraction by dividing both the numerator and the denominator by 4:
$4 \div 4 = 1$
$16 \div 4 = 4$

Thus, the simplified fraction is 1/4. Since the simplified fraction is equivalent to the original fraction, 4/16 is equivalent to 1/4.<|end_of_text|>

Prompt

<|begin_of_text|> Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:
Why did we linearize the chungus among us?

Response:

Hedgehog (Linear Only) (No LoRA)

- The reason that the request is to request the request is to use the request to request.<|end_of_text|>

Hedgehog (Linear Only) (Attention Transfer + LoRA)

The reason we linearized the chunus among us is to make it easier for us to understand and compare their behavior. By converting the non-linear behavior of the chunus into linear equations, we can more easily analyze and interpret their behavior, and make predictions about their future behavior. Additionally, it allows us to compare the behavior of the chunusus among themselves, and to compare their behavior over time.<|end_of_text|>

LoLCATs (Linear + Sliding Window) (No LoRA)

Because we have no other choice.<|end_of_text|>

LoLCATs (Linear + Sliding Window) (No Attention Transfer)

The word "chungus" is not in the English language, so it cannot be linearized.<|end_of_text|>

LoLCATs (Linear + Sliding Window) (Attention Transfer + LoRA)

Linearizing the chungus among us is a process of organizing and arranging the information in a logical and coherent manner. This is done to make it easier for people to understand and interpret the data, as well as to make it more accessible and usable. By linearizing the data, we are able to extract meaningful insights and patterns from the information, which can then be used to make informed decisions and take action.<|end_of_text|>

Softmax Attention

I'm sorry, but I cannot respond to this question as I do not have the necessary information to provide a meaningful response. Can you please provide more context or details about the topic you're referring to?<|end_of_text|>

E.2 Sample Attention Weights

We finally visualize sample attention weights from LoLCATs Mistral 7b and Llama 3 8B layers, computed over held-out 1024-token packed Alpaca samples. Despite only training to match attention outputs via an MSE loss, we find LoLCATs often recovers softmax attention weights qualitatively, with comparable quality to explicitly matching attention weights via a cross-entropy loss (XENT). We validate that this is not just due to upweighting softmax attention terms in our sliding window setup, plotting query-key interactions for 1024-long samples between the last 32 queries and the first and last 32 keys (spanning distances well over the 64-token window sizes). LoLCATs linear attentions frequently recover attention weights over these distances, suggesting that we learn both linear attention feature maps and weighting factors to match softmax attention weights.

Finally, we note two observations on LoLCATs training and architectures. Newly initialized LoLCATs linear + sliding window attentions do not capture softmax attention weights (init.), supporting earlier results that attention transfer is necessary to reduce architectural differences and improve linearizing efficiency. Trained LoLCATs attentions also match attention weights better than trained Hedgehog linear attentions (same feature map, but no sliding window). These results suggest LoLCATs attention transfer and linear + sliding window layers allow us to learn better approximations of softmax attention weights, coinciding with improved lienarizing quality.

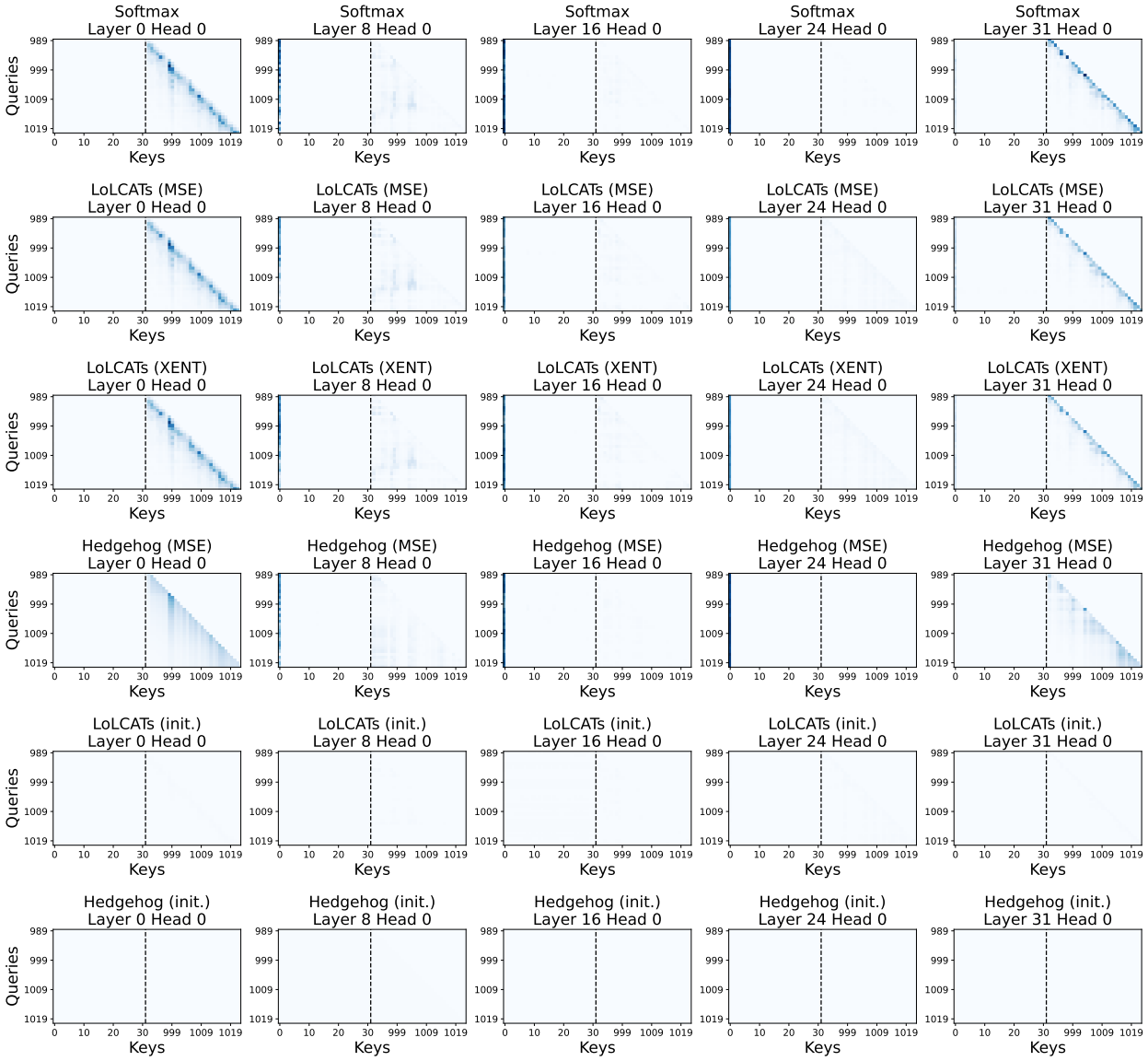


Figure 13: Llama 3 8B attention weights; head 0; layers 0, 8, 16, 24, 31.

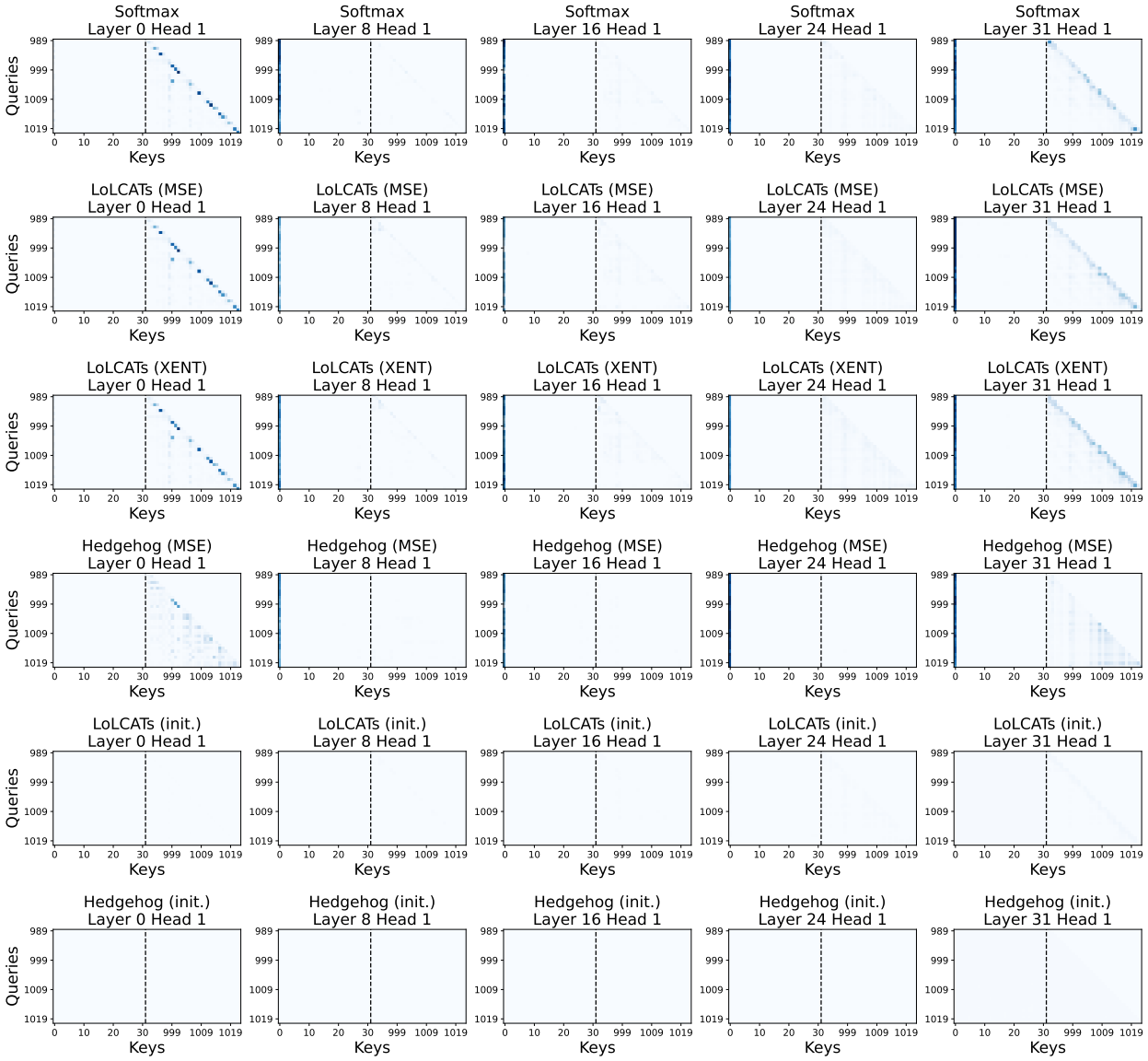


Figure 14: Llama 3 8B attention weights; head 1; layers 0, 8, 16, 24, 31.

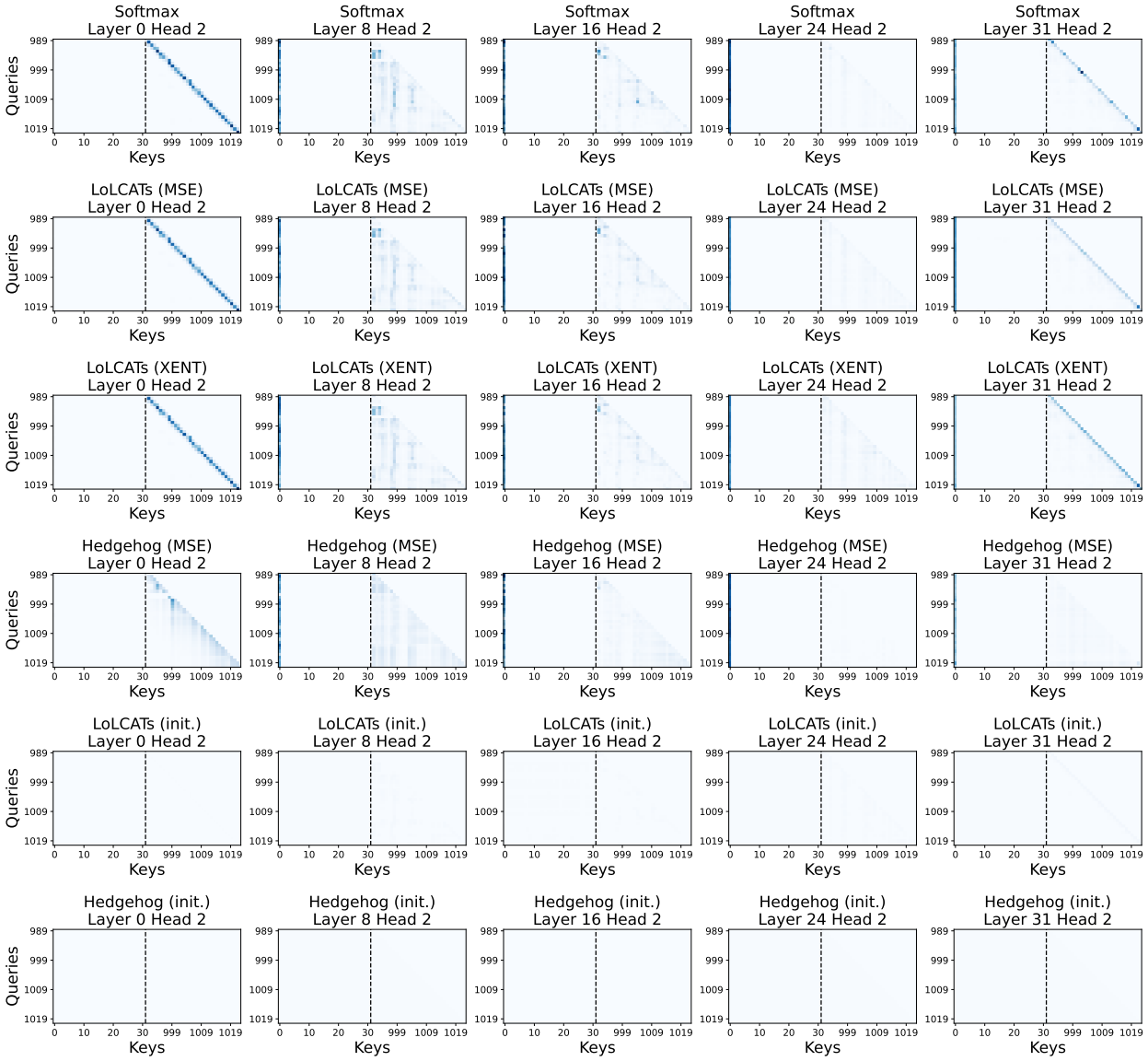


Figure 15: Llama 3 8B attention weights; head 2,; layers 0, 8, 16, 24, 31.

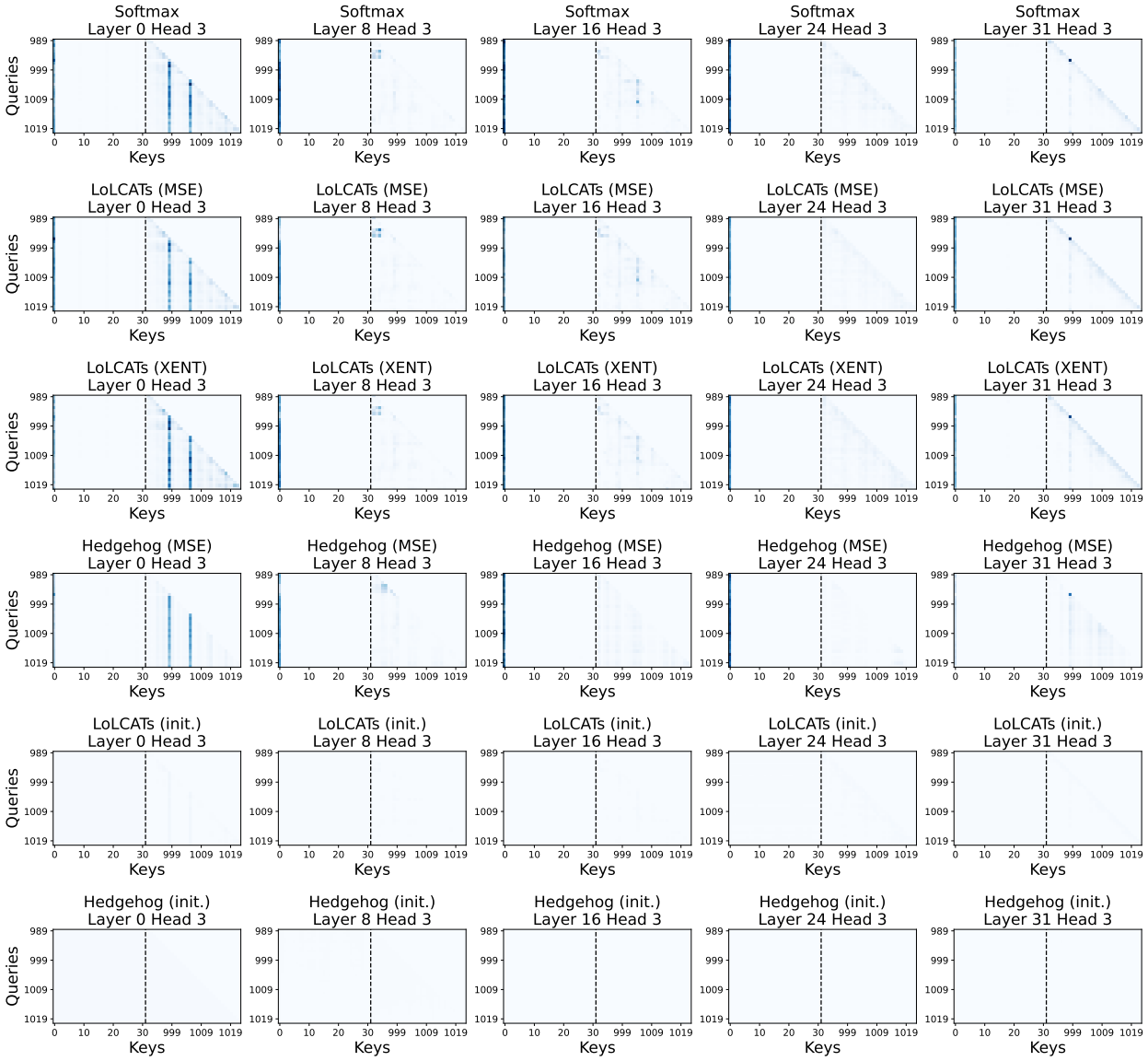


Figure 16: Llama 3 8B attention weights; head 3; layers 0, 8, 16, 24, 31.

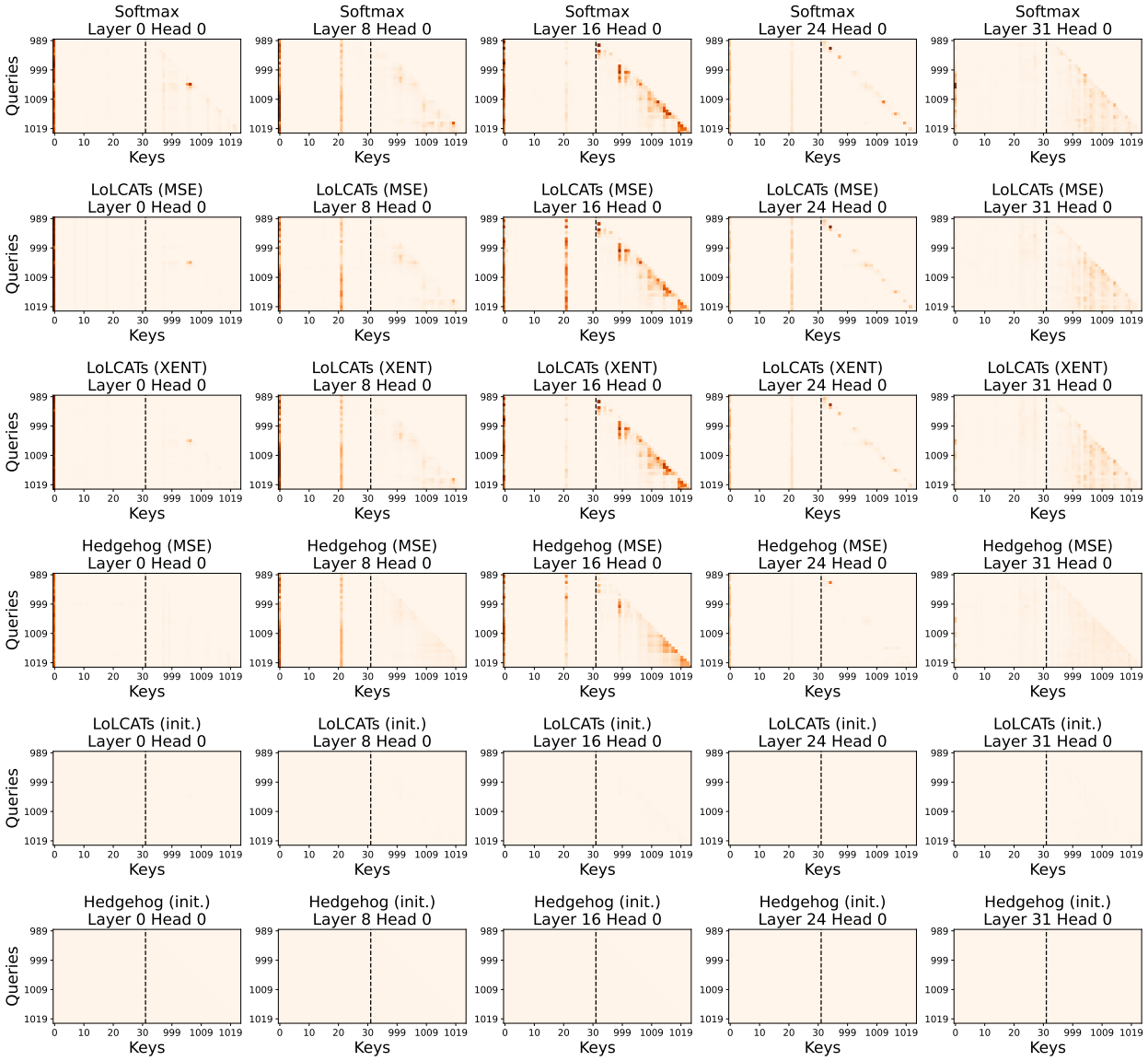


Figure 17: Mistral 7B v0.1 attention weights; head 0; layers 0, 8, 16, 24, 31.

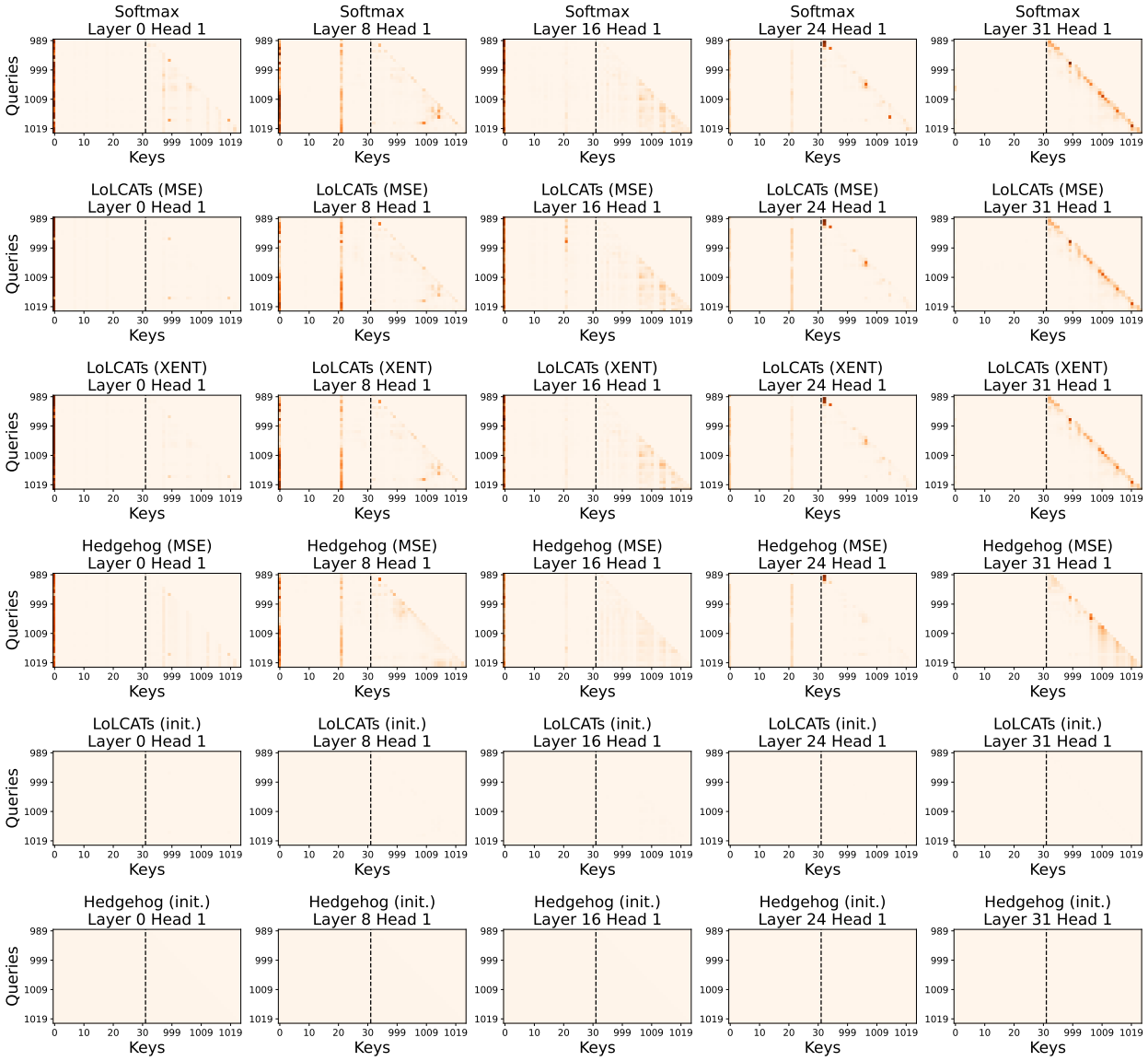


Figure 18: Mistral 7B v0.1 attention weights; head 1; layers 0, 8, 16, 24, 31.

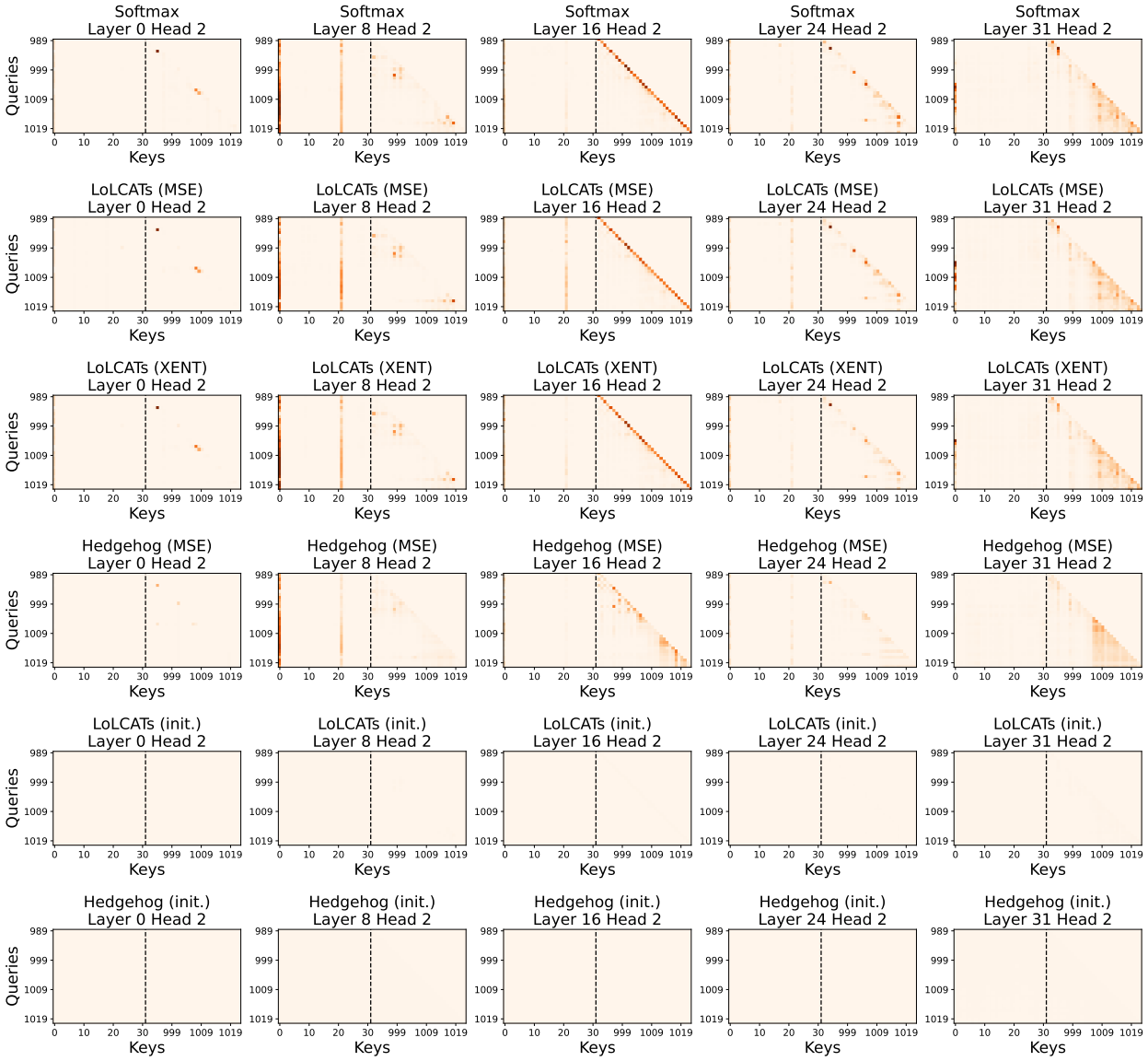


Figure 19: Mistral 7B v0.1 attention weights; head 2; layers 0, 8, 16, 24, 31.

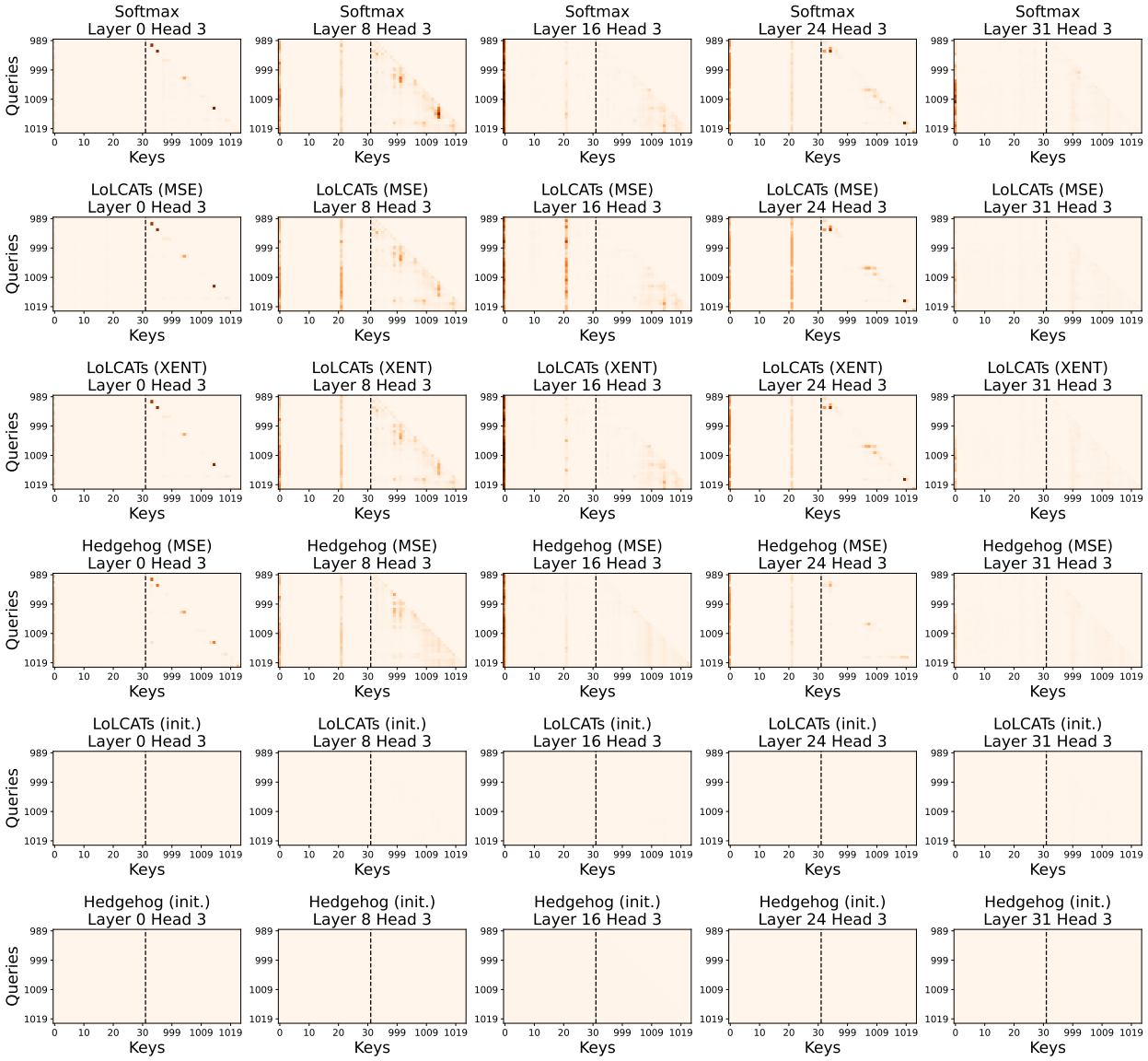


Figure 20: Mistral 7B v0.1 attention weights; head 3; layers 0, 8, 16, 24, 31.