LoLCATs: On Low-Rank Linearizing of Large Language Models

Anonymous authors
Paper under double-blind review

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 show that the subquadratic analogs used in prior work struggle to approximate the original softmax attention layer. 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: (1) the "*attention transfer*" training step uses our new linear attention architecture, designed to improve the approximation fidelity, and minimizes the MSE between the original and new layer's attention outputs, (2) we adjust for any approximation errors by simply using *low-rank* adaptation (LoRA). LoLCATs significantly improves linearizing quality, training efficiency, and scalability. LoLCATs produces 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, 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 methods 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 78.7% and 77.4% 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 (Gu & Dao, 2023; Peng et al., 2023; Yang et al., 2023; Arora et al., 2024; Beck et al., 2024). In a complementary direction, linearizing aims to start with openly available LLMs—*e.g.,* those with 7B+ parameters trained on trillions of tokens (AI, 2024; Jiang et al., 2023)—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 in modern LLMs.

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.,* recovering 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 the various 70B+ LLMs available today (Touvron et al., 2023a;b).

Existing methods present opportunities to improve all three criteria. On quality, despite using motivated subquadratic analogs such as RetNet-inspired linear attentions (Sun et al., 2023; Mercat et al., 2024) or state-space model (SSM)-based Mamba layers (Gu & Dao, 2023; Yang et al., 2024; Wang et al., 2024), prior works significantly reduce performance on popular LM Evaluation Harness tasks (LM Eval) (Gao et al., 2023) (up to 23.4-28.2 pts on 5-shot MMLU (Hendrycks et al., 2020)). On parameter and token efficiency, to adjust for architectural differences, prior methods update *all* model parameters in at least one stage of training (Mercat et al., 2024; Wang et al., 2024; Yang et al.,

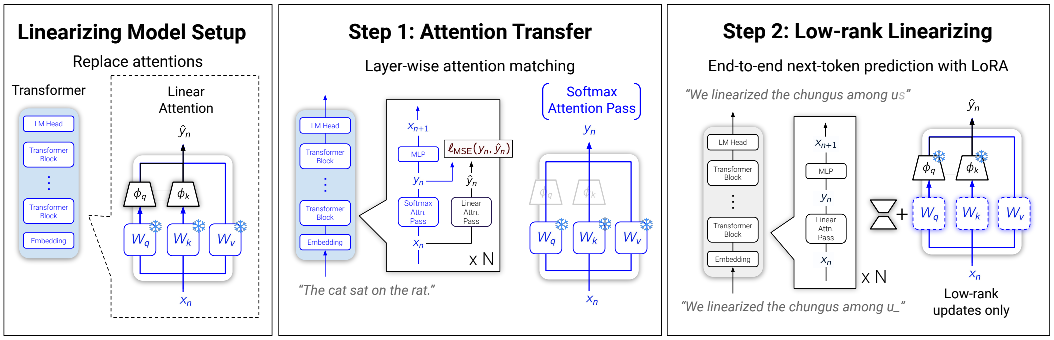


Figure 1: **LoLCATs framework**. We linearize LLMs by (1) training attention analogs to approximate softmax attentions (attention transfer), before swapping attentions and (2) minimally adjusting (with LoRA).

2024), and use 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 (Dubey et al., 2024).

In this work, we thus propose **LoLCATs** (**Lo**w-rank **L**inear **C**onversion with **A**ttention **T**ran**s**fer), 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) (Hu et al., 2021) ("**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 (Keles et al., 2023; Qin et al., 2022) or need full-model updates to recover linearizing quality (Kasai et al., 2021; Mercat et al., 2024). 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 (Zhang et al., 2024), we also find that later layers can result in 200$\times$ the MSE of earlier ones (Figure 7). We later find this issue aggravated by larger LLMs; joint training for Llama 3.1 405B's 126 attention layers fails to linearize LLMs.

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 0.2$-$8.0 points (pts) on zero-shot LM Eval tasks; +17.8 pts on 5-shot MMLU)). With Llama 3 8B, LoLCATs for the first time closes the zero-shot LM Eval gap between linearized and Transformer models (73.1 vs 73.7 pts), while supporting 3$\times$ higher throughput and 64$\times$ larger batch sizes vs. popular FlashAttention-2 (Dao, 2023) implementations (generating 4096 token samples on an 80GB H100). We further validate

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	✓	✓	✓✓	✓✓

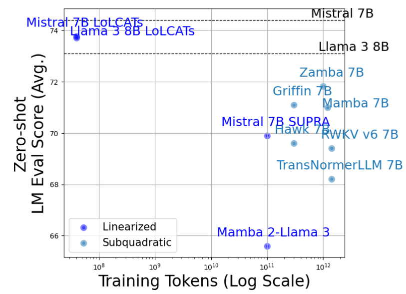


Figure 2: **Linearizing comparison**. LoLCATs significantly improves LLM linearizing quality and training efficiency.

LoLCATs-linearizing as a high-quality training method, outperforming strong 7B subquadratic LLMs (RWKV-v6 (Peng et al., 2024), Mamba (Gu & Dao, 2023), Griffin (De et al., 2024)) and hybrids (StripedHyena (Poli et al., 2023a), Zamba (Glorioso et al., 2024)) trained from scratch by 1.8 to 4.7 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. 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 the Llama 3.1 70B and 405B parameter LLMs (Dubey et al., 2024). LoLCATs presents the first viable approach to linearizing larger LLMs, creating the first linearized 70B and 405B LLMs with only 9.5 hours on an 8×80GB H100 node for Llama 3.1 70B and 16 hours across 24 80GB H100s for Llama 3.1 405B. This is in total less than half the compute reported by prior methods to linearize 8B models (5 days on 8×80GB H100s) (Wang et al., 2024). Furthermore, under these computational constraints, LoLCATs significantly improves quality vs. prior linearizing approaches (Kasai et al., 2021; Mercat et al., 2024). With Llama 3.1 70B and 405B, we close 78.7% and 77.4% of the 5-shot MMLU gap between Transformers and linearized variants respectively.

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 (AI@Meta, 2024) and Mistral 7B (Jiang et al., 2023) are decoder-only Transformers, with repeated blocks of multi-head *softmax attention* followed by MLPs (Vaswani et al., 2017). For one head, attention models 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

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

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 (Dao, 2023).

Linear Attention. To get around this, Katharopoulos et al. (2020) 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 $\mathcal{K}(\boldsymbol{q}, \boldsymbol{k})$, which in general can be expressed as the dot product of feature maps $\phi : \mathbb{R}^d \mapsto \mathbb{R}^{d'}$.

Swapping $\exp(\boldsymbol{q}^\top \boldsymbol{k}/\sqrt{d})$ with $\phi(\boldsymbol{q})^\top \phi(\boldsymbol{k})$ 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 Katharopoulos et al. (2020)). 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 (Mercat et al., 2024), and those for smaller Transformers—*e.g.,* 110M BERTs (Devlin et al., 2018))—reasonably adaptable to modern LLMs (Kasai et al., 2021; Mao, 2022; Zhang et al., 2024).

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:

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

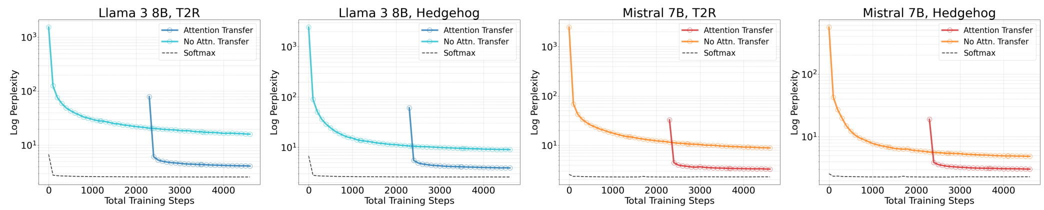


Figure 3: **Attention transfer training efficiency**. Even accounting for initial training steps, low-rank linearizing with attention transfer still consistently achieves lower perplexity faster across feature maps and LLMs.

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 (Kasai et al., 2021; Zhang et al., 2024), rather than manually design ϕ, we parameterize each $\phi : \mathbb{R}^d \mapsto \mathbb{R}^{d'}$ as a *learnable* layer:

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

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) (Hu et al., 2021). 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 parameter counts.

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 (Mercat et al., 2024; Yang et al., 2024; Wang et al., 2024). They use linearizing layers featuring GroupNorms (Wu & He, 2018) and decay factors (Sun et al., 2023), or alternate SSM-based architectures (Gu & Dao, 2023; Dao & Gu, 2024). 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 (AI, 2024) and Mistral 7B v0.1 (Jiang et al., 2023). For linearizing layers, we study two feature maps used in prior work (Table 2). To support the rotary positional embeddings (RoPE) (Su

Attention	Llama 3 8B				Mistral 7B			
	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).

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 4. Acc. norm: ARC-C, HS. Acc. otherwise. 5-shot MMLU. 0-shot otherwise.

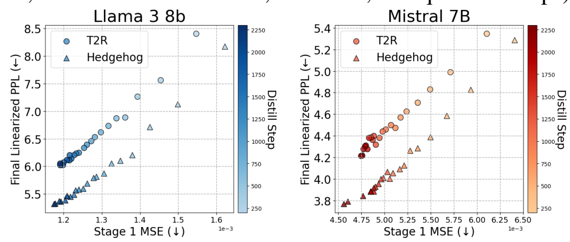


Figure 5: **Attention MSE vs. PPL**. Across feature maps, LLMs; lower MSE coincides with better linearized quality.

et al., 2024) 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 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 (Taori et al., 2023). Following Zhang et al. (2024), we train all feature maps jointly. We include training code and implementation details in App. **??**).

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 Kasai et al. (2021), Hedgehog from Zhang et al. (2024), both $\oplus$ (concat) and SM_d (softmax) apply over feature dimension.

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 (Hu et al., 2021), which results in training 0.2% of LLM parameter counts.

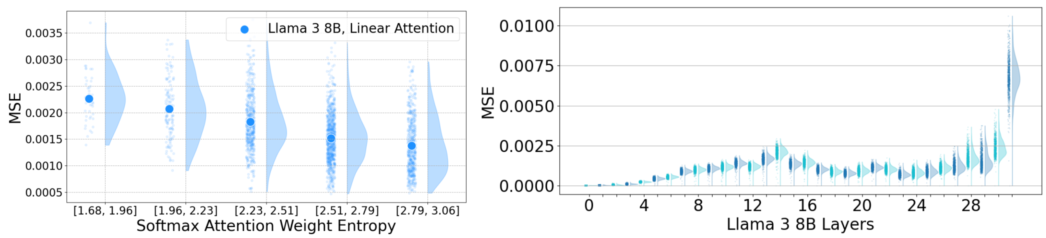
Attention Transfer + LoRA Enables Fast LLM Linearizing. In Table 1 and Fig. 3, we report the validation PPL of linearized LLMs, ablating attention transfer and LoRA adjusting. We find that while attention transfer alone is often insufficient (*c.f.,* PPL@0, Table 1), a single low-rank update rapidly 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 (Mercat et al., 2024; Wang et al., 2024) (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 three observations:

1. Attention transfer quality (via output MSE) strongly ties to low-rank linearized quality (Fig. 5).

[1] Unlike prior works that apply ϕ before RoPE (Mercat et al., 2024; Su et al., 2024), our choice preserves the linear attention kernel connection, where we can hope to learn ϕ_q, ϕ_k for $\exp(\boldsymbol{q}^\top \boldsymbol{k}'/\sqrt{d}) \approx \phi_q(\boldsymbol{q})^\top \phi_k(\boldsymbol{k}')$.

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



(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.

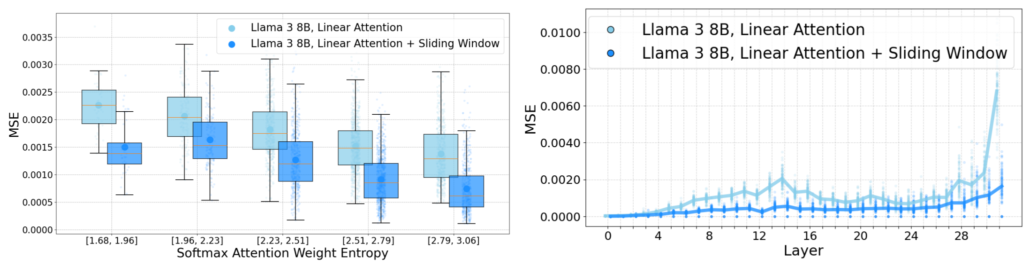


Figure 7: **Improving Attention matching MSE**. Linearizing with linear + sliding window attention better matches LLM softmax attentions (lower MSE) over attention entropy values and LLM layers.

2. Larger attention output MSEs coincide with lower softmax attention weight entropies (Fig. 6a).
3. Larger MSEs also heavily concentrate in attention input samples from later layers (Fig. 6b).

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 (Zhang et al., 2024)—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.

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 (Arora et al., 2024; Munkhdalai et al., 2024), we combine short sliding windows of softmax attention (Beltagy et al., 2020; Zhu et al., 2021) (size 64 in experiments) followed by linear attention in a single layer. This allows attending to all prior tokens for each layer while keeping the entire LLM subquadratic. For window size w and token indices $[1, \dots, n-w, \dots, n]$, 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 ($\boldsymbol{c}_n = \max_i \left\{ \boldsymbol{q}_n^\top \boldsymbol{k}_i / \sqrt{d} : i \in [n-w+1, \dots, n] \right\}$). 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

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 Hedgehog	**0.04**	79.3	76.4	45.1	73.1	57.5	28.2	59.9	66.3
Mistral 7B LoLCATs (Ours)	**0.04**	**81.1**	**81.1**	**52.9**	**80.5**	**73.2**	**52.2**	**70.2**	**73.8**
Llama 3 8B	-	79.9	80.1	53.3	79.1	73.1	66.6	72.0	73.1
Mamba2-Llama 3	100	76.8	74.1	48.0	70.8	58.6	43.2	61.9	65.6
Mamba2-Llama 3, 50% Attn.	100	**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**	80.6	**81.8**	53.5	**79.1**	**73.4**	54.9	**70.6**	**73.7**

Table 3: **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 73.8% (Mistral 7B) and 86.1% (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).

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 training approach and provide a simplified model to show its cost-quality tradeoffs. To improve layer-wise quality, 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:

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

We can choose k to balance cost and quality. There are several approaches for training block-wise, including via joint training with separate optimizer groups per block or by sequentially training separate blocks. The primary costs-tradeoffs between these two approaches are:

- **Compute.** Increasing k increases the compute required per block. While the joint training of Llama 3.1 405B in 16-bit precision uses multiple NVIDIA H100 8×80GB nodes, separate blocks of $k = 9$ or fewer layers fits on a *single* H100 80GB GPU, at sequence length 1024.
- **Memory and training time.** The total amount of memory 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 outputs per-layer ($k = 1$) for 40M tokens would require 165TB of disk space. Sequentially saving the outputs and training the blocks increases total training time.

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 (AI, 2024) and Mistral 7B (Jiang et al., 2023) 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.

Model	Training Tokens (B)	PiQA	ARC-e	ARC-c (norm)	HellaSwag (norm)	Wino-grande	MMLU (5-shot)	Avg.	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[3]	**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									
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**	81.1	81.1	52.9	80.5	73.2	52.2	70.2	73.8
Llama 3 8B LoLCATs (Ours)	**0.04**	80.6	**81.8**	53.5	**79.1**	**73.4**	54.9	**70.6**	**73.7**

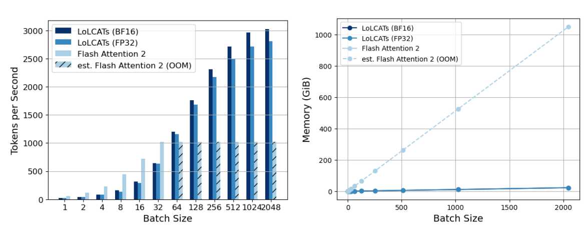
Table 4: **LoLCATs comparison to pretrained subquadratic LLMs**. LoLCATs-linearized Mistral 7B and Llama 3 8B further outperform pretrained subquadratic Transformer alternatives by 0.1 to 9.4 points (Avg.), while only training 0.2% of the model parameters on 0.013% to 0.003% of their pretraining token counts.

In Table 4, we report results on six popular LM Evaluation Harness (LM Eval) tasks (Gao et al., 2023). Compared to recent linearizing methods, LoLCATs significantly improves quality and training efficiency across tasks and LLMs. On quality, LoLCATs closes 73.8% and 81.1% of the Transformer-linearizing gap for Mistral 7B and Llama 3 8B respectively, notably improving 5-shot MMLU by 63.8% and 50% against next best fully subquadratic models. On efficiency, we achieve these results despite only training <0.2% of model parameters via LoRA versus prior full-parameter training and use 40M tokens versus the prior 100B (0.04% of the latter, a 2500$\times$ improvement in "tokens-to-model" efficiency). Among all 7B LLMs, LoLCATs-linearized LLMs further outperform strong subquadratic Transformer alternatives, including RNNs or linear attentions (RWKV-v6 (Peng et al., 2024), Hawk (De et al., 2024), Griffin (De et al., 2024), TransNormer (Qin et al., 2023)), state-space models (SSMs) (Mamba (Gu & Dao, 2023)), and hybrid architectures with some full attention (StripedHyena (Poli et al., 2023b), Zamba (Glorioso et al., 2024)).

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[4], and compare LoLCATs implemented in HuggingFace Transformers with the supported FlashAttention-2 (FA2) implementation (Dao, 2023). 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 $\times$ 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

[4] 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	**70.60**	68.78
	MMLU (5-shot)	23.80	23.80	**52.50**	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 5: **LoLCATs component ablations**, linearizing Llama 3 8B over 1024-token sequences. Default configuration highlighted. Across feature maps, LoLCATs' attention transfer and sliding window increasingly improve linearized LLM quality.

	PiQA acc	ARC Easy acc	ARC Challenge (acc norm)	HellaSwag (acc norm)	WinoGrande acc	MMLU (5-shot) acc
Llama 3.1 70B	83.10	87.30	60.60	85.00	79.60	78.80
Linearized, no attn. transfer	75.70	70.10	39.10	77.40	58.60	26.60
LolCATs (Ours)	**82.10**	**85.00**	**60.50**	**84.60**	**73.70**	**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**	**71.90**

Table 6: **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.

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 samples (Table 5). We start with standard linear attentions (Hedgehog, Zhang et al. (2024); T2R, Kasai et al. (2021)), using the prior linearizing procedure of just swapping attentions and finetuning the model to predict next tokens (Mercat et al., 2024). 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 4 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 6, we find that LoLCATs provides the first practical solution for linearizing larger LLMs, achieving significant quality improvements over prior linearizing approaches (Mercat et al., 2024). For Llama 3.1 70B, we achieve a 41.1 point improvement in 5-shot MMLU accuracy. For Llama 3.1 405B, LoLCATs similarly achieves a 38.0 point improvement over prior methods. These results highlight LoLCATss 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.

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, including sample code for the linearizing architecture and training (Appendix C).

References

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

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

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.

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.

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

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

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

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

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.

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.

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.

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.

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.

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.

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

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.

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.

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.

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.

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.

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.

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

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.

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.

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.

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

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.

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.

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.

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*, 2023a.

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 2023b. URL https://github.com/togethercomputer/stripedhyena.

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.

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.

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.

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.

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

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.

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.

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*, 2023a.

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*, 2023b.

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.

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.

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.

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

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.

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.

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.

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

For linearizing layers, we replace softmax attentions with hybrid linear + terraced window analogs (Section 3.3.1), using Hedgehog's feature map for its prior quality Zhang et al. (2024). For linearizing data, we report results using the Alpaca-linearized models. We also tried a more typical pretraining corpus (1B tokens[5] of RedPajama Computer (2023)), but found comparable performance when controlling for number of token updates. 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 $r = 8$, amounting to training <0.09% of all model parameters. For both stages, we train with early stopping, AdamW optimizer Loshchilov & Hutter (2017), and packing into 1024-token sequences with batch size 8.

B Related Work

In this work, we build upon both approaches explicitly proposed to linearize LLMs Mercat et al. (2024), as well as prior methods focusing on smaller Transformers reasonably adaptable to modern LLMs Kasai et al. (2021); Mao (2022); Zhang et al. (2024). We highlight two approaches most related to LoLCATs and their extant limitations next.

Scalable UPtraining for Recurrent Attention (SUPRA). Mercat et al. (2024) linearize LLMs by swapping softmax attentions with linear attentions similar to Retentive Network (RetNet) layers Sun et al. (2023), before jointly training all model parameters on the RefinedWeb pretraining dataset Penedo et al. (2023). In particular, they suggest that linearizing LLMs with the vanilla linear attention in Eq. 2 is unstable, and swap attentions with

$$\hat{\boldsymbol{y}}_n = \text{GroupNorm}\Big(\sum_{i=1}^{n} \gamma^{n-i}\phi(\boldsymbol{q}_n)^\top\phi(\boldsymbol{k}_i)\boldsymbol{v}_i\Big) \tag{9}$$

GroupNorm Wu & He (2018) 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) Kasai et al. (2021) with rotary embeddings Su et al. (2024). 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 Touvron et al. (2023b) and Mistral 7B Jiang et al. (2023) models on popular LM Evaluation Harness Gao et al. (2023) and SCROLLS Shaham et al. (2022) tasks.

[5] https://huggingface.co/datasets/togethercomputer/RedPajama-Data-1T-Sample

Hedgehog. Zhang et al. (2024) 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 (Vaswani et al., 2017)) are first copied from an existing softmax attention. They then specify learnable feature maps $\phi(\boldsymbol{x}) = [\text{softmax}(\boldsymbol{xW} + \boldsymbol{b}) \oplus \text{softmax}(-\boldsymbol{xW} - \boldsymbol{b})]$ (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:

$$\mathcal{L}_n = -\sum_{i=1}^{n} \frac{\exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i/\sqrt{d})}{\sum_{i=1}^{n} \exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i/\sqrt{d})} \log \frac{\phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i)}{\sum_{i=1}^{n} \phi(\boldsymbol{q}_n)^\top \phi(\boldsymbol{k}_i)} \tag{10}$$

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 (Wang et al., 2018) or WikiText-103 language modeling (Merity et al., 2017)), 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 on both smaller Transformers (*e.g.,* 110M parameter BERTs Devlin et al. (2018) and 125M GPT-2s Radford et al. (2019)), and Llama 2 7B for a specific SAMSum summarization task Gliwa et al. (2019).

C Code Implementation

We include sample code for implementing LoLCATs with HuggingFace Transformers API.

```
def compute_loss(self, model: nn.Module, data: dict[torch.Tensor], **
    kwargs: any,):
    """
    Attention distillation ("attention transfer")
    - For each layer and head, get attentions and train to
      minimize some combo of MSE and cross-entropy loss
    """
    input_seq_len = data['input_ids'].shape[-1]
    inputs = {'input_ids': data['input_ids'].to(model.device)}

    # Get softmax attention outputs
    with torch.no_grad():
        # Set base_inference to True to use FlashAttention
        for layer in traverse_layers(model):
            layer.self_attn.base_inference = True
        # Get hidden states
        true_outputs = model(**inputs, output_attentions=True,
                             use_cache=False,)
        # Save attention layer inputs and outputs in outputs.attentions
        # attn_inputs  = [a[0] for a in true_outputs.get('attentions')]
        # attn_outputs = [a[1] for a in true_outputs.get('attentions')]
        true_attn_io = true_outputs.get('attentions')  # layer-wise attn
    inputs and outputs
        true_outputs = true_outputs.get('logits').cpu()
        for layer in traverse_layers(model):
            layer.self_attn.base_inference = False

    # Get trainable subquadratic attention outputs
    attention_type = getattr(layer.self_attn, 'attention_type', None)
    past_key_values = get_attention_cache(attention_type)

    total_seq_len = 0
    position_ids = torch.arange(input_seq_len).view(1, -1)

    loss_mse = 0
    for layer_idx, layer in enumerate(traverse_layers(model)):
        attn_input, attn_output = true_attn_io[layer_idx]
```

```
        attn_preds = layer.self_attn(attn_input.to(model.device),
                                     attention_mask=None,
                                     position_ids=position_ids,
                                     past_key_value=past_key_values)[1]
        # MSE on layer outputs
        loss_mse += criterion_mse(attn_preds, attn_output)
    loss_mse = loss_mse / (layer_idx + 1) * self.mse_factor
    loss = loss_mse
```

Listing 1: Attention Distillation Code

```
class LolcatsLlamaAttention(nn.Module):
    """
    Hedgehog attention implementation initialized from a
    `LlamaAttention` or `MistralAttention` object (base_attn)

    Most of the arguments are directly tied to argparse args

    Note that we don't currently support padding.
    """
    def __init__(self,
                 base_attn: nn.Module,  # like LlamaAttention
                 feature_map: str,
                 feature_map_kwargs: dict,
                 layer_idx: Optional[int] = None,
                 max_layer_idx: Optional[int] = None,
                 feature_map_mlp: Optional[str] = None,
                 feature_map_mlp_kwargs: Optional[dict] = None,
                 tie_qk_fmap: Optional[bool] = False,
                 rotary_config: Optional[dict] = None,
                 attention_type: Optional[str] = 'hedgehog_llama',
                 mask_value: int = 0,
                 eps: float = 1e-12,):
        super().__init__()

        self.mask_value = mask_value
        self.eps = eps
        self.layer_idx = (layer_idx if layer_idx is not None
                          else base_attn.layer_idx)
        self.max_layer_idx = max_layer_idx

        self.rotary_config = rotary_config

        self.tie_qk_fmap = tie_qk_fmap
        self.init_feature_map_(feature_map, feature_map_kwargs,
                               feature_map_mlp, feature_map_mlp_kwargs)
        self.init_weights_(base_attn)

    def init_feature_map_(self,
                          feature_map: str,
                          feature_map_kwargs: dict,
                          feature_map_mlp: str = None,
                          feature_map_mlp_kwargs: dict = None):
        """
        Initialize feature map
        """
        if feature_map_mlp is not None:
            feature_map_kwargs['num_heads'] = self.num_heads
            feature_map_kwargs['head_dim'] = self.head_dim
            feature_map_kwargs['dtype'] = self.q_proj.weight.dtype
            feature_map_kwargs['device'] = self.q_proj.weight.device
            feature_map_mlp = init_feature_map_mlp(feature_map_mlp,
                                                   feature_map_mlp_kwargs
    )
        self.feature_map_q = init_feature_map_act(name=feature_map,
```

```
                                                  mlp=feature_map_mlp,
                                                  **feature_map_kwargs)
        if self.tie_qk_fmap:  # tie mlp weights for query and key feature maps
            self.feature_map_k = self.feature_map_q
        else:
            self.feature_map_k = copy.deepcopy(self.feature_map_q)

    def init_weights_(self, base_attn: nn.Module):
        """
        Initialize module layers, weights, positional dependencies, etc.
        """
        self.attention_dropout = 0  # We don't use dropout
        self.hidden_size = base_attn.hidden_size
        self.num_heads = base_attn.num_heads
        self.head_dim = base_attn.head_dim
        self.num_key_value_heads = base_attn.num_key_value_heads
        self.num_key_value_groups = base_attn.num_key_value_groups

        self.q_shape = [self.num_heads, self.head_dim]
        self.k_shape = [self.num_key_value_heads, self.head_dim]
        self.v_shape = [self.num_key_value_heads, self.head_dim]

        self.max_position_embeddings = base_attn.max_position_embeddings
        device = base_attn.q_proj.weight.device
        scaling_factor = getattr(base_attn.rotary_emb, 'scaling_factor', 1.)
        if self.rotary_config is None:
            self.rotary_emb = get_rotary_embeddings(
                rope_scaling_type=None,
                head_dim=self.head_dim,
                max_position_embeddings=base_attn.rotary_emb.max_position_embeddings,
                rope_theta=base_attn.rotary_emb.base,
                rope_scaling_factor=scaling_factor,
                device=device,
            )
        else:
            if 'device' not in self.rotary_config:
                self.rotary_config['device'] = device
            self.rotary_emb = get_rotary_embeddings(**self.rotary_config)

        # Just initialize with original weights
        device = base_attn.q_proj.weight.device
        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
        del base_attn

    def linear_attention(self, q: torch.Tensor, k: torch.Tensor, v: torch.Tensor) -> Tuple[torch.Tensor, Optional[torch.Tensor], Optional[Tuple[torch.Tensor]]]:
        """
        Compute linear attention with CUDA kernel implementation from fast-transformers
        """
        dtype = q.dtype
        y = causal_dot_product(q.contiguous().to(dtype=torch.float32),
                               k.contiguous().to(dtype=torch.float32),
                               v.contiguous().to(dtype=torch.float32)).to(dtype=dtype)
        y = y / (torch.einsum("bhld,bhld->bhl", q, k.cumsum(dim=2)) + self.eps)[..., None]
        return y, None, None
```

```python

    def forward(self,
                hidden_states: torch.Tensor,
                attention_mask: Optional[torch.Tensor] = None,
                position_ids: Optional[torch.LongTensor] = None,
                past_key_value: Optional[Tuple[int, torch.Tensor, torch.Tensor]] = None,
                output_attentions: bool = False,
                use_cache: bool = False,
                **kwargs) -> Tuple[torch.Tensor, Optional[torch.Tensor], Optional[Tuple[torch.Tensor]]]:
        """
        Forward pass modified from transformers.models.mistral.modeling_mistral (v4.36)
        """
        b, l, _ = hidden_states.size()
        q = self.q_proj(hidden_states)
        k = self.k_proj(hidden_states)
        v = self.v_proj(hidden_states)
        kv_seq_len = k.shape[-2]

        q = q.view(b, l, *self.q_shape).transpose(1, 2)
        k = k.view(b, l, *self.k_shape).transpose(1, 2)
        v = v.view(b, l, *self.v_shape).transpose(1, 2)

        if past_key_value is not None:
            kv_seq_len += past_key_value[0].shape[-2]

        cos, sin = self.rotary_emb(k, seq_len=kv_seq_len)
        q, k = apply_rotary_pos_emb(q, k, cos, sin, position_ids)

        k = repeat_kv(k, self.num_key_value_groups)
        v = repeat_kv(v, self.num_key_value_groups)
        q, k = self.feature_map_q(q), self.feature_map_k(k)

        if attention_mask is not None and q.shape[2] > 1:
            lin_attn_mask = attention_mask[:, None, :, None]
            k = k.masked_fill(~lin_attn_mask, self.mask_value)

        if past_key_value is not None:
            kv_state = past_key_value.kv_states[self.layer_idx]
            k_state  = past_key_value.k_states[self.layer_idx]

            y_true, _, _ = self.linear_attention(q, k, v)
            past_key_value.update(k, v, self.layer_idx)
        else:
            y_true, _, _ = self.linear_attention(q, k, v)

        y_true = y_true.transpose(1, 2).contiguous().view(b, l, self.hidden_size)
        y_true = self.o_proj(y_true)
        attn_weights = None

        return y_true, attn_weights, past_key_value
```

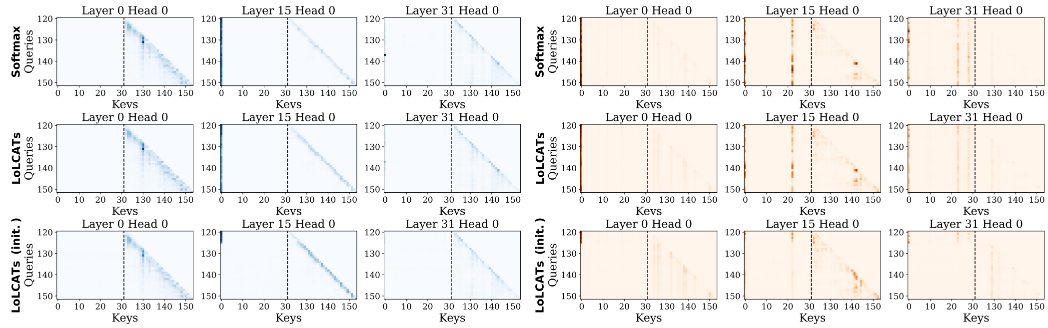
Listing 2: LoLCATs Attention Implementation

```python
class HedgehogFeatureMap(nn.Module):
    """
    Final 'activation' of feature map. Can probably be combined with
    `HedgehogFeatureMapMLP` below

    Full feature map is like f(xW + b)
    -> This is the `f` part
    """
```

```
    def __init__(self,
                 head_dim_idx: int = -1,
                 eps: float = 1e-12,
                 mlp: nn.Module = None,
                 halfspace: bool = False,
                ):
        super().__init__()
        self.head_dim_idx = head_dim_idx
        self.eps = eps
        self.mlp = mlp if mlp is not None else nn.Identity()
        self.activation = (self.halfspace_activation if halfspace
                           else self.fullspace_activation)

    def fullspace_activation(self, x: torch.Tensor):
        return torch.cat([
            torch.softmax( x, dim=self.head_dim_idx),
            torch.softmax(-x, dim=self.head_dim_idx)
        ], dim=self.head_dim_idx).clamp(min=self.eps)

    def halfspace_activation(self, x: torch.Tensor):
        return torch.softmax(x, dim=self.head_dim_idx).clamp(min=self.eps)

    def forward(self, x: torch.Tensor):
        """
        Assume x.shape is (batch_size, n_heads, seq_len, head_dim)
        """
        return self.activation(self.mlp(x))


class HedgehogFeatureMapMLP(nn.Module):
    """
    Learnable MLP in feature map.

    Full feature map is like f(xW + b)
    -> This is the `W` and (optional) `b` part
    """
    def __init__(self,
                 num_heads: int,
                 head_dim: int,     # input dim
                 feature_dim: int,  # output dim
                 dtype: torch.dtype,
                 device: torch.device,
                 skip_connection: bool = False,
                 bias: bool = False):
        super().__init__()
        self.num_heads = num_heads
        self.head_dim = head_dim
        self.feature_dim = feature_dim
        self.dtype = dtype
        self.device = device
        self.skip_connection = skip_connection
        self.bias = bias
        self.init_weights_()

    def init_weights_(self):
        """
        Initialize W and b
        """
        self.weight = nn.Parameter(torch.zeros(
            (self.num_heads, self.head_dim, self.feature_dim),
            dtype=self.dtype, device=self.device,
        ))
        nn.init.kaiming_uniform_(self.weight)

```

```
        if self.bias:
            self.bias = nn.Parameter(torch.zeros(
                (1, self.num_heads, 1, self.feature_dim),
                dtype=self.dtype, device=self.device,
            ))
            nn.init.kaiming_uniform_(self.bias)
        else:
            self.bias = 0.  # hack

    def forward(self, x: torch.Tensor):
        """
        Assume x.shape is (batch_size, num_heads, seq_len, head_dim)
        """
        _x = torch.einsum('hdf,bhld->bhlf', self.layer, x) + self.bias
        return x + _x if self.skip_connection else _x
```

Listing 3: Hedgehog Learnable Feature Map Implementation



(a) Llama 3 8B attention weights (b) Mistral 7B (v0.1) attention weights

Figure 9: **Attention Transfer**. For both Llama 3 8B and Mistral 7B v0.1 LLMs, LoLCATs attention transfer trains subquadratic attentions that match original attention weights, despite only supervising based on attention layer outputs. They also learn to recover weights outside of the softmax windows, *c.f.* trained versus initialized (init.) attentions between queries at positions 130 - 150 and keys at positions 0 - 32.

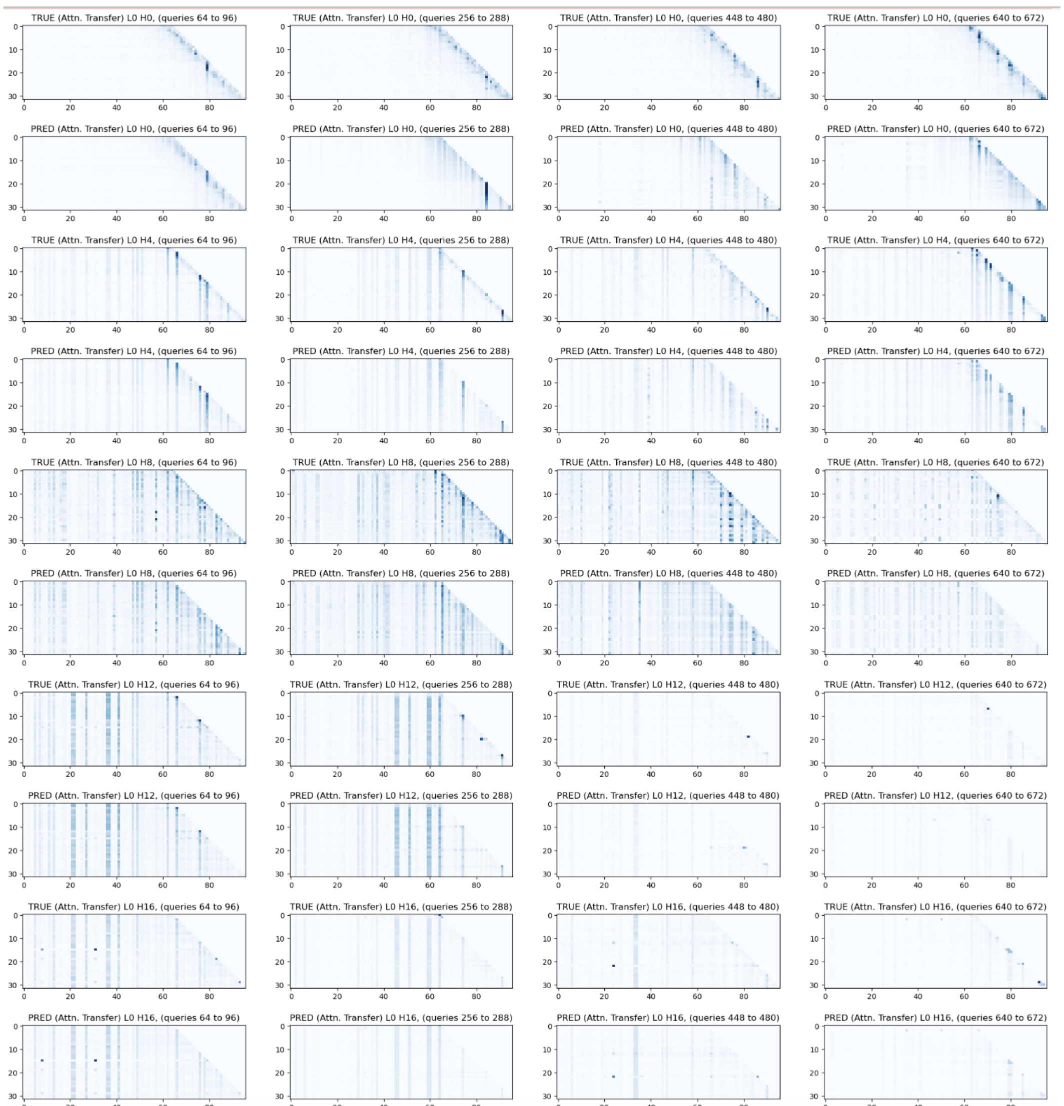


Figure 10: Linear attention (PRED) and softmax attention (TRUE) weights for hedgehog learned feature map, with attention transfer.

Figure 11: Linear attention (PRED) and softmax attention (TRUE) weights for hedgehog learned feature map, without attention transfer.