# ATTENTION AND COMPRESSION IS ALL YOU NEED FOR CONTROLLABLY EFFICIENT LANGUAGE MODELS

**Jatin Prakash** **Aahlad Puli** **Rajesh Ranganath**
New York University
jatin.prakash@nyu.edu

## ABSTRACT

The quadratic cost of attention in transformers motivated the development of efficient approaches: namely sparse and sliding window attention, convolutions and linear attention. Although these approaches result in impressive reductions in compute and memory, they often trade-off with quality, specifically in-context recall performance. Moreover, apriori fixing this quality-compute tradeoff in an architecture means being suboptimal from the get-go: some downstream applications require more memory for in-context recall, while others require lower latency and memory. Further, these approaches rely on heuristic choices that artificially restrict attention, or require handcrafted and complex recurrent state update rules, or they must be carefully composed with attention at specific layers to form a hybrid architecture that complicates the design process, especially at scale.

To address above issues, we propose **C**ompress & **A**ttend **T**ransformer (CAT), a conceptually simple architecture employing two simple ingredients only: dense attention and compression. CAT decodes chunks of tokens by attending to compressed chunks of the sequence so far. Compression results in decoding from a reduced sequence length that yields compute and memory savings, while choosing a particular chunk size trades-off quality for efficiency. Moreover, CAT can be trained with multiple chunk sizes at once, unlocking control of quality-compute trade-offs directly at test-time without any retraining, all in a single adaptive architecture.

In exhaustive evaluations on language modeling and common-sense reasoning, in-context recall, and long-context understanding, a single adaptive CAT model outperforms many existing efficient baselines, including hybrid architectures, across different compute-memory budgets. Further, a single CAT matches dense transformer in language modeling across different model scales while being $1.4 - 3\times$ faster and requiring $2 - 9\times$ lower total memory usage.

Play with CATs at: github.com/rajesh-lab/cat-transformer

## 1 INTRODUCTION

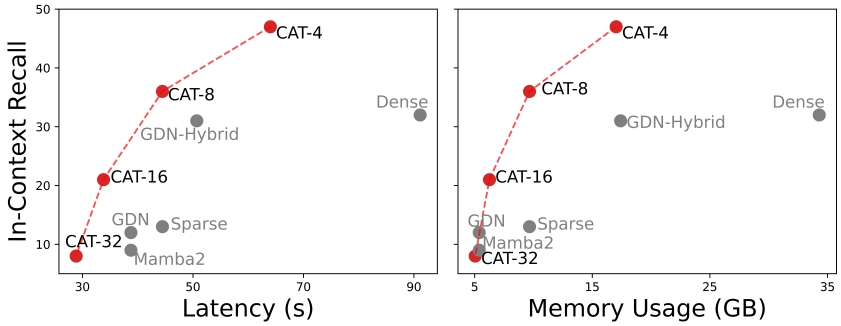


Figure 1: **CAT unlocks test-time control of quality-efficiency trade-offs**, where a **single adaptive CAT model** (all **red** dots come from a single model) outperforms nearly every popular efficient architecture on real-world in-context recall tasks across varying compute-memory budgets.

Transformers (Vaswani et al., 2017) are the default architectures for large language models (LLMs) and rely on the powerful self-attention mechanism (Bahdanau et al., 2014). However, the compute required for decoding with self-attention grows quadratically with the sequence length, with memory costs growing linearly, making transformers expensive to deploy.

Given the cost of self-attention, there has been interest in designing efficient alternatives: while approaches like sparse and sliding window attention (Child et al., 2019; Zaheer et al., 2020; Jiang et al., 2023) heuristically restricts the tokens being attended to; approaches like linear attention (Katharopoulos et al., 2020; Arora et al., 2024a; Dao & Gu, 2024; Yang et al., 2025b) rely on fixed-size recurrent states with complex state update rules, to enable constant compute and memory costs. However, restricting attention to tokens apriori or using fixed-size recurrent states, that have problems managing information over long sequences, hurt in-context recall performance (Arora et al., 2024a; Jelassi et al., 2024; Wen et al., 2024). To make these approaches performant, they require careful composition with dense attention at specific layers, making the design process cumbersome especially at scale (Waleffe et al., 2024; Wang et al., 2025). Enabling efficiency by recursively compressing the sequence can avoid fixed-memory bottlenecks and heuristic restrictions (Rae et al., 2020; Chevalier et al., 2023), but sequential computations in these approaches makes the training slow and optimization difficult (Geiping et al., 2025).

Additionally, existing approaches do not account for the differences in compute and memory requirements across diverse downstream tasks. For example, writing short email replies does not require strong in-context recall performance and usage of linear attention can be sufficient; but code auto-completion demands accurate recall of function names from the entire code repository in the context, where more memory and compute of dense attention may be preferred. The existing approaches for efficiency *fix* the compute-memory usage before training. This means if at test time a problem demands a higher budget for better performance, a whole new model needs to be trained. Training multiple models with different tradeoffs is one way to tackle this problem but repeating this for every downstream task can become quickly prohibitive. Even if such models were available, routing between these models based on the context requires holding all of them in memory.

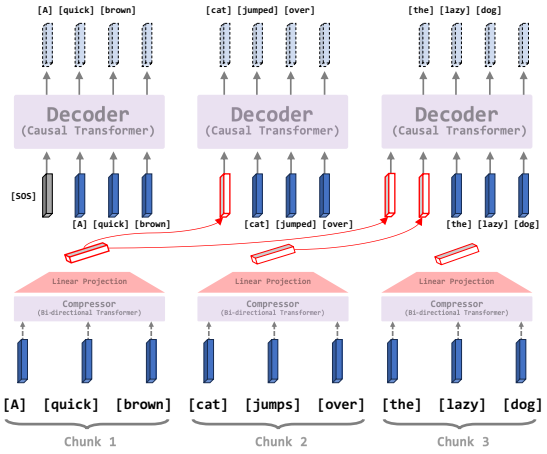


Figure 2: **The Compress and Attend Transformer (CAT) architecture.** CAT chunks up a sequence of length $N$ into $N/C$ chunks of $C$ tokens (illustrated for $C = 3$). Each chunk is parallelly compressed into a chunk representation. CAT then decodes each chunk by attending to past chunk representations. Compression results in a reduced sequence length enabling compute and memory savings during decoding. Chunk size in CAT acts as knob, offering test-time control of quality-efficiency trade-offs, where higher chunk sizes result in increased efficiency.

To address the issues raised above, we propose a conceptually simple architecture: **C**ompress & **A**ttend **T**ransformer (CAT) that employs two simple well-known ingredients, namely dense attention and compression. CAT compresses chunks of tokens in parallel into a shorter sequence using a *compressor*, which a *decoder* then attends to while autoregressively modeling the tokens in the latest chunk; both compressor and decoder are simple dense transformers themselves (see Figure 2). With the compression and decoding being parallel over tokens during training, there is no recurrence along the sequence dimension, which enables end-to-end **scalable training**. Decoding from the reduced sequence length due to compression enables **compute and total memory savings**. This reduction allows the use of more parameters in CAT, thereby improving model quality.[1] Choosing a particular chunk size in CAT trades-off quality for compute and memory (see Figure 1). At the same

[1]similar in vein to flagship models (Yang et al., 2025a; Agarwal et al., 2025) that are all parameterized as Mixture-of-Experts (MoEs) (Shazeer et al., 2017), where additional model parameters does not mean more compute, and brings improvements in quality.

time, the **memory grows *gracefully***, linearly with sequence length but at a significantly slower rate, to enable in-context recall performance at long sequence lengths. Importantly, training CATs across multiple chunk sizes at once **unlocks control of quality-compute trade-offs directly at test-time** without any retraining, all in a single adaptive architecture.

To summarize, this paper:

- Introduces the CAT architecture to efficiently model sequences by decoding each chunk of tokens given parallelly compressed representations of the past chunks. Adjusting a single knob (chunk size) at test-time controls quality-efficiency trade-offs, allowing a single CAT model to *interpolate* between the dense transformer and efficient alternatives without any retraining.
- Provides a parallel and scalable implementation for training CATs (we scale from 90M to 1B parameters) and an efficient pure PyTorch implementation for generation that does not require any custom CUDA or Triton kernels, unlike most efficient baselines.

  We release code at: `github.com/rajesh-lab/cat-transformer`
- Demonstrates that a **single adaptive CAT model**
  - outperforms many popular efficient baselines including hybrid architectures on language modeling, common-sense reasoning, long-context understanding, in-context recall, and needle-in-haystack tasks, across different compute and memory budgets.
  - matches or outperforms the dense transformer on language modeling at multiple model scales while being $\mathbf{1.4-3}\times$ faster and using a $\mathbf{2-9}\times$ smaller total memory footprint.
  - surpasses, interestingly even the dense transformer on real-world in-context recall tasks using the least efficient setting (CAT-4) while still being atleast $1.5\times$ faster and $2\times$ memory efficient, akin to MoEs (Shazeer et al., 2017).

*Brief outline of the paper:* Section 2 first lays out CATs, including the overall architecture design, training objective and implementation details regarding scalable training and efficient generation. This is followed by a small discussion with the related work, highlighting conceptual differences between CATs and other efficient architectures in Section 3 and Table 1. Then, Section 4 exhaustively compares CATs with efficient baselines on plethora of downstream tasks. Finally, Section 5 ends with a discussion on the practical utility of CATs and some future work.

## 2 COMPRESS AND ATTEND TRANSFORMERS (CATS)

In this section, we describe the components of the CAT architecture and how it is trained for test-time control of trade-offs between quality and compute. Next, we discuss efficient implementation for CATs and the resulting compute and memory savings.

**Compression and decoding.** Given a sequence $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ of $N$ tokens, we split the sequence into chunks $(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_{N_C})$ containing $C$ tokens each, such that $\mathbf{c}_i = (x_{C\cdot i+1}, \ldots, x_{C\cdot i+C}) = (\mathbf{x}_{i,1}, \ldots \mathbf{x}_{i,C}) = \mathbf{x}_{i,:}$, where $\mathbf{x}_{i,:}$ indexes the $i$-th chunk of $C$ consecutive tokens (`numpy` array slicing).

Next, CAT compresses each chunk $\mathbf{c}_i$ using the *compressor* $f_\theta$ into chunk representations. The *compressor* $f_\theta$ is a dense bidirectional transformer with hidden size $D_f$, followed by a linear projection to $D_g$. This leads to a *compressed* chunk representation $f_\theta(\mathbf{c}_i) \in \mathcal{R}^{D_g}$. That is:

$$\mathbf{x} = \{x_1, \cdots x_N\} \xrightarrow{\text{chunking}} \{\mathbf{c}_1, \cdots \mathbf{c}_{N_c}\} \xrightarrow{\text{compress}} \{f_\theta(\mathbf{c}_1), \cdots f_\theta(\mathbf{c}_{N_c})\}.$$

After compression, CAT decodes the original sequence $\mathbf{x}$ from the compressed chunk representations $\{f_\theta(\mathbf{c}_i)\}_{i=1}^{N_C}$ using a *decoder* $g_\theta$, which is a causal dense transformer having hidden size $D_g$, matching the linear projection from the *compressor*. CAT decodes chunks autoregressively, where to decode each token $\mathbf{x}_{i,j}$ in a chunk $\mathbf{c}_i$, the decoder takes as input the previous tokens $\{\mathbf{x}_{i,<j}\}$ in chunk $\mathbf{c}_i$ and the past chunk representations $\{f_\theta(\mathbf{c}_1), \ldots, f_\theta(\mathbf{c}_{i-1})\}$. Formally, the predictive

distribution $p_\theta$ for the tokens in chunk $\mathbf{c}_i$ is defined as:

$$p_\theta(\mathbf{c}_i \mid \mathbf{c}_{i-1} \cdots \mathbf{c}_1) = \prod_{j=1}^{C} g_\theta\big( \underbrace{\mathbf{x}_{i,j}}_{j^{\text{th}} \text{ token in chunk } \mathbf{c}_i} \mid \underbrace{\mathbf{x}_{i,j-1}, \ldots \mathbf{x}_{i,1}}_{\text{previous tokens in chunk } \mathbf{c}_i}, \underbrace{f_\theta(\mathbf{c}_{i-1}) \cdots f_\theta(\mathbf{c}_1)}_{\text{past chunk representations}} \big) \quad (1)$$

By using compressed chunk representations, CAT reduces the amount of compute and memory required for decoding; the larger the chunk size the larger the reduction in memory and compute.

During training, the compression and the decoding happens in parallel for all tokens in the sequence because compression of a chunk does not depend on earlier chunks. This choice allows the entire CAT model to be efficiently trained end-to-end with the standard next-token prediction loss. The end-to-end training ensures that CATs *learn what to retain* in their compressed chunk representations rather than relying on fixed attention patterns, or complex state update rules.

**Training for test-time control in compute and memory.** Varying the chunk size in CATs trades-off quality for compute and memory efficiency. Training CAT with multiple chunk sizes renders a single adaptive model whose compute-memory budget can be adjusted directly at test-time without any retraining.

To build such a controllably efficient CAT model, we uniformly sample a chunk size $C$ at each training iteration, and pass in a *learnable* indicator token to CAT to indicate which chunk size it is currently operating at. The compressed tokens are separated from the uncompressed ones in the decoder using a marker token shared across different chunk sizes. After training, one can use the same CAT model at different compute/memory budget at test-time by just changing the indicator token. Appendix B.4 provides further detail.

### 2.1 How to implement fast and scalable CATs

As both components of CAT are transformers, CAT admits a pure PyTorch implementation for scalable training and fast generation, without requiring custom CUDA or Triton kernels. We describe the approach here.

**Fast and Parallel Compression.** Compression of chunks of tokens is efficient and can be executed in parallel, for instance by using `torch.vmap` (Horace He, 2021), to produce $\{f_\theta(\mathbf{c}_i)\}$ for all chunks $\mathbf{c}_i$. This costs a total of $O(\frac{N}{C} \cdot C^2) = O(NC)$ in self-attention compute, rather than $O(N^2)$.

**Naive and Slow Training.** For training the decoder, a naive implementation can lead to slower training. To compute logits for tokens in chunk $\mathbf{c}_i$, that is computing $g_\theta(\mathbf{c}_i \mid f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1}))$ in parallel can be non-trivial. Since, for chunk $\mathbf{c}_i$, the number of past chunk varies, making shapes variable and as a result, harder to parallelize the computation of logits. One could employ a python loop and compute logits for every chunk sequentially, but that would be slow and won't scale. Padding to make shapes constant to allow parallelism would make things worse by increasing wasteful computations. In fact, even if one bypasses varying shapes problem and manages to compute logits for every chunk in parallel, the total self-attention operations in the decoder would scale as $O(\sum_{i=1}^{N_c}(i + C)^2) = O((\frac{N}{C})^3)$, that is cubic in sequence length. Thus, even the perfect parallel approach for training will not scale, despite CATs being a simple architecture.

**Fast and Scalable Training.** To make training scalable in CATs, we observe that in computing logits for every chunk $\mathbf{c}_i$, one calculates exactly the same key-value vectors for the representation $f_\theta(\mathbf{c}_j)$ in the decoder transformer, where $j < i$. This means that computation is duplicated. We exploit this observation in training CATs.

On a high-level, we implement this observation by modifying the original chunked sequence $\mathbf{x} = \{\mathbf{c}_1, \ldots \mathbf{c}_i \ldots\}$ to $\{\mathbf{c}_1, f_\theta(\mathbf{c}_1), \mathbf{c}_2, f_\theta(\mathbf{c}_2), \ldots \mathbf{c}_i, f_\theta(\mathbf{c}_i) \ldots\}$, that is we insert compressed representations of the chunk after the chunk of tokens itself. Now, we pass this sequence into the decoder during training, with a custom attention mask (App. Figure 8) that allows a token in chunk $\mathbf{c}_i$ to attend to previous tokens within that chunk and *only* to previous chunk representations, which would be $f_\theta(\mathbf{c}_{i-1}), f_\theta(\mathbf{c}_{i-2}) \ldots f_\theta(\mathbf{c}_1)$. Any token in chunk $\mathbf{c}_i$ does not attend to raw tokens outside this chunk. This implementation allows re-use of key-values for chunk representations $f_\theta(\mathbf{c}_i)$ in decoder for computing logits of a future chunk $\mathbf{c}_j$, where $j > i$. This way of computing logits is quadratic in sequence length, in fact it is a constant times better: $O(\frac{N^2}{C})$ vs. the $O(N^2)$ complexity

of the dense transformer, allowing for a potentially faster pre-training (see Section 5 and appendix D for a discussion).

**Fast and Efficient Generation.** Due to compression, CATs can throwaway past chunks of tokens, and only keep their compressed chunk representations in memory. This straightaway results in a big reduction of memory; the KV cache is slashed by a factor of $C$, even for a modest chunk size of 4 (see Figure 3). This slash in memory is accompanied by reduced memory accesses the decoder makes in CATs, which is the major bottleneck during generation. The decoder attends to atmost $N_c + C$ tokens during generation, reducing compute required in self-attention significantly.

Implementing generation is simpler than training and very similar to how it occurs for a dense transformer. In fact, a pure PyTorch implementation[2] for CATs is on-par with efficient architectures that utilize custom kernels. Given a sequence, CATs first compute representations for each chunk in parallel and use them to prefill the decoder's KV cache. Then generation proceeds chunk by chunk: each new chunk is decoded token by token in the decoder, and once a chunk is complete, the chunk is compressed and its representation is prefilled in the KV cache for the generation of the next chunk. This loop continues until the sequence is fully generated.

The full implementation details along with a PyTorch style pseudo-code are in Appendices B and D.3.

## 3 Related Work

We provide a brief summary of the most relevant related work in this section. Table 1 highlights key properties and conceptual differences between CATs and other methods. For an extended related work, refer to Appendix E.

| Method | Unrestricted Access to Memory? | Flexible memory? | Scalable training? | Both compute & memory efficient? | Adaptive? |
|---|---|---|---|---|---|
| ***Dense Attention***: Vaswani et al. (2017) | ✓ | ✓ | ✓ | ✗ | ✗ |
| ***Sparse Attention***: Child et al. (2019) | ✗ | ✓ | ✓ | ✓ | ✗ |
| ***NSA***: Yuan et al. (2025) | ✓ | ✓ | ✓ | ✗ | ✗ |
| ***Sliding window Attn.***: Jiang et al. (2023) | ✗ | ✗ | ✓ | ✓ | ✗ |
| ***Linear Attention***: Dao & Gu (2024) | ✓ | ✗ | ✓ | ✓ | ✗ |
| ***Recursive compression***: Chevalier et al. (2023) | ✓ | ✓ | ✗ | ✓ | ✗ |
| ***MegaByte/Block Transformer***: Ho et al. (2024); Yu et al. (2023) | ✓ | ✗ | ✓ | ✓ | ✗ |
| ***CATs*** | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 1: We categorize the existing related work into key properties that are desirable for an efficient architecture. *"Both compute and memory efficient?"* signifies savings during inference; *"Unrestricted Access to Memory"* signifies whether an architecture can freely access any part of the memory in the past, without any artificial restrictions.

**Efficient self-attention using custom masks:** These techniques include *heuristically* defined *fixed* sparse or stratified attention masks (Child et al., 2019; Zaheer et al., 2020) or local sliding window masks (Jiang et al., 2023) that artificially restricts the tokens being attended to in self-attention. The compute required (and in some attention masks, memory) for attention goes down during generation, but if the *wrong* attention mask is chosen for the task, these methods will be less performant or would

[2]Our implementation is inspired from: github.com/meta-pytorch/gpt-fast

require more depth (Arora et al., 2024a). To match quality of a dense transformer, these models either require big window sizes (making their memory costs large again) or need to be composed with dense attention again at specific layers (Arora et al., 2024a; Agarwal et al., 2025).

**Compressing past context:** Rae et al. (2020); Chevalier et al. (2023) explored recurrent formulations of a transformer to enable generation of longer sequences on limited compute and memory by compressing past context. But sequential training is slow and memory intensive, making these approaches hard to scale on modern hardware that favors parallel computations. Moreover, training models in a recurrent fashion has optimization challenges, back-propagation through time (BPTT) being the most important one. More recently Geiping et al. (2025) had to use very careful recipe to train a large recurrent architecture in a stable manner and prevent optimization collapse.

Alternatively, Native Sparse Attention (NSA) (Yuan et al., 2025) reduce attention compute by attending to compressed chunks of tokens as well as to specific chunks of uncompressed tokens in the past. These past tokens are compressed in parallel in every layer. This is similar in spirit to our work, however there are no memory savings during inference since the KV cache needs to be retained for the entire past context; there are only compute savings.

**Linear attention:** Arora et al. (2024a); Katharopoulos et al. (2020) linearize self-attention that replace softmax-based attention with kernelized dot-product-based linear attention, that further admits a linear recurrence form. Recent enhancements incorporate data-dependent gating mechanism in the recurrence (Dao & Gu, 2024; Yang et al., 2025b); all require handcrafted and complicated recurrent state update rules. These architectures show impressive reductions in compute and memory, but the fixed-size recurrent state struggles to manage information over long sequences, that hurts in-context recall performance (Arora et al., 2024a; Jelassi et al., 2024; Wen et al., 2024). To make these mixers competitive, they are usually composed with long sliding window attention at specific layers (Yang et al., 2025b). Performing such a composition is unclear and requires careful *trial-and-error* (Waleffe et al., 2024; Qwen, 2025) making the design process for an efficient architecture cumbersome, especially at scale (Wang et al., 2025).

**Hierarchical transformers:** Nawrot et al. (2021; 2022); Slagle (2024) explored downsample-then-upsample approach (*hour-glass* like structure), where the sequence is downsampled into *coarse* tokens followed by upsampling into *fine-grained* tokens before being decoded. Due to the *hour-glass* structure, there are compute savings during training; but the architecture must maintain a cache for all the past tokens leading to significant memory accesses (especially for *fine-grained* ones) which is the main bottleneck during generation.

Unlike above, Ho et al. (2024); Yu et al. (2023) break up the modeling of a sequence into independent chunks/patches, given a single compressed representation of the entire past. While compression helps in efficiency, the requirement to decode each chunk from a single, fixed size, compressed representation results in poor in-context recall even on simple toy tasks (Fig. 7). Further, unlike the original encoder-decoder architectures that attend directly to past tokens (Raffel et al., 2020; Vaswani et al., 2017), decoder in CAT attends to the compressed representations of chunks of tokens in the past.

CATs sidestep many limitations of existing efficient baselines described above. Firstly, CATs do not require any heuristic choices for attention masks, complex state update rules, or careful composition with attention layers to have competitive performance; CATs directly build on simple dense transformer abstractions. Secondly, CATs alleviate the fixed memory bottleneck by having flexible but efficient memory usage: it grows *gracefully* as sequence length increases, resulting in superior in-context recall performance, despite using similar memory overall compared to fixed memory baselines (Table 3). Thirdly, CATs admit scalable training where compression and decoding can happen in parallel. Finally, CATs enable control of quality-compute trade-offs at test-time, allowing them to cater to diverse downstream tasks with different compute-memory budgets. This is similar in spirit to Kusupati et al. (2022); Devvrit et al. (2023); Beyer et al. (2023). Table 1 provides a brief summary.

Refer to Appendix E for an extended related work.

## 4 EXPERIMENTS

**Baselines:** Our experiments provide a comprehensive comparison of recent architectures, including (i) attention-based baselines: standard Dense Transformer (Touvron et al., 2023) and Sparse Transformer (Child et al., 2019), (ii) Linear Transformers such as Mamba2 (Dao & Gu, 2024) and GatedDeltaNet (GDN) (Yang et al., 2025b), as well as (iii) hybrid architectures such as the hybrid variant of GDN having alternate layers as long sliding windows, GDN-Hybrid.

All baselines use $L = 12$ layers with hidden size of $D = 1024$, making their parameters count not more than $\sim$ 300M, except Sparse Transformer that uses $\sim 800M$ parameters due to hidden size of $2D = 2048$ for a fair comparison with CATs (as we will see below). GDN-Hybrid employs a sliding window of 2K, following Yang et al. (2025b). Refer to Appendix D for more details regarding hyperparameters used for each baseline.

**Training setup:** All models were trained on 15B tokens of FineWeb-Edu Penedo et al. (2024) which is $2.5\times$ the Chinchilla optimal, with a context length of 4K following Behrouz et al. (2024); Yang et al. (2025b). We use the AdamW optimizer (Loshchilov & Hutter, 2017) with a peak learning rate of 8e-4, weight decay of 0.1, gradient clipping of 1.0, batch-size of 0.5M tokens, employing the GPT2 tokenizer (see Appendix D for more details).

| Model | LMB↓ | Wiki↓ | FW↓ | HS↑ | PQ↑ | AE↑ | AC↑ | WG↑ | OQA↑ | Avg.↑ |
|---|---|---|---|---|---|---|---|---|---|---|
| Dense | 38.7 | 19.6 | 17.1 | 34.8 | 65.6 | 56.7 | 24.4 | 51.1 | 20.0 | 42.1 |
| Sparse | 37.2 | 18.5 | 16.0 | 35.6 | 66.8 | 57.3 | 25.4 | 51.1 | 22.8 | 43.2 |
| Mamba2 | 36.1 | 19.5 | 16.7 | 36.1 | 67.0 | 59.2 | 26.5 | 51.9 | 21.6 | 43.7 |
| GDN | **35.7** | 18.8 | 16.3 | 36.1 | 66.8 | 58.7 | 25.2 | 51.6 | 22.8 | 43.5 |
| GDN-H | 36.6 | 18.5 | 16.2 | **36.8** | 66.3 | 56.4 | 25.8 | 52.1 | 20.4 | 43.0 |
| CAT-4 | 38.0 | **18.1** | 16.0 | 35.6 | 66.4 | 59.5 | **27.1** | 51.5 | 23.4 | 43.9 |
| CAT-8 | 37.2 | **18.1** | **15.8** | 35.4 | 66.8 | 60.1 | **27.4** | 51.3 | 23.6 | 44.1 |
| CAT-16 | 36.8 | 18.4 | 16.0 | 35.5 | 67.3 | 60.2 | 27.0 | 52.0 | 23.8 | 44.3 |
| CAT-32 | 36.8 | 19.1 | 16.4 | 35.9 | **68.2** | **61.0** | 27.0 | **53.6** | **25.0** | **45.1** |

Table 2: Zero-shot perplexity and accuracy on language modeling and common-sense reasoning benchmarks. Note that CAT-4/8/16/32 reported here is a single model.

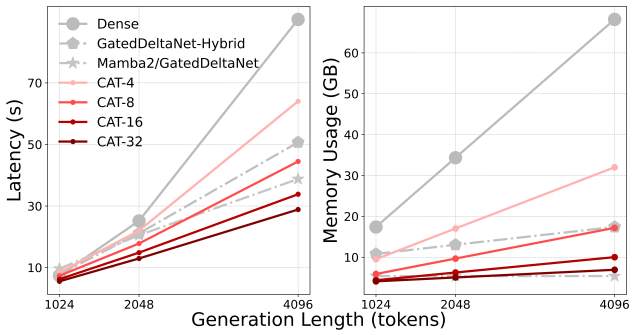


Figure 3: **A single CAT model generates $1.4-3.2\times$ faster than the dense transformer while showcasing upto $2.2-9.5\times$ lower memory usage**. Per Table 3, CAT-8 outperforms GDN-Hybrid in real-world recall tasks while being faster and requiring similar memory; CAT-16 outperforms Mamba2 and GDN and is $1.15\times$ faster but costs slightly ($\sim 15\%$) more memory.

| Model | SWDE | FDA | Avg. |
|---|---|---|---|
| Dense | 43.4 | 19.7 | 32.0 |
| Sparse | 20.9 | 6.0 | 13.0 |
| Mamba2 | 13.5 | 4.5 | 9.0 |
| GDN | 18.0 | 6.8 | 12.0 |
| GDN-Hybrid | 44.0 | 17.8 | 31.0 |
| CAT-4 | **49.1** | **45.1** | **47.1** |
| CAT-8 | 38.2 | 34.8 | 36.5 |
| CAT-16 | 27.5 | 15.4 | 21.5 |
| CAT-32 | 13.2 | 3.2 | 8.2 |

Table 3: Zero-shot performance on real-world in-context recall tasks measured at 2K sequence lengths. We report results on SWDE and FDA here, which have longer sequences among the datasets in the suite (others have an average length of $\leq$ 300 tokens (Arora et al., 2024b)). Appendix A shows evaluations on all datasets. All CATs reported here is a single model.

**What makes CATs purr?** To match dense-transformer perplexity, we empirically find a more *expressive* decoder helps: that is, decoder uses $2\times$ hidden size. *This suggests accurate decoding from compressed representations needs extra compute*, with similar observations in recent works (Ho et al., 2024; Yu et al., 2023). Refer to App. C.2 for a comparison. Further, we find depth of compressor does not have major effect on perplexity (App. C). Given these findings, to instantiate CATs that compete with dense transformer of depth $L$ and hidden size $D$: CATs use a decoder of depth $L$ and hidden size $2D$, and a compressor

of depth $L/4$ and hidden size $D$. While this increases parameters, CATs are still significantly faster and memory efficient (see Figure 3) compared to the corresponding dense transformer. Thus, for CATs we use $L = 12$ layers, same as baselines, but a wider hidden size of $D_g = 2D = 2048$ for the decoder. The compressor uses $L = 3$ layers and hidden size of $D_f = D = 1024$. This makes the parameter count for CATs close to $1B$. We train CATs simultaneously on chunk sizes $C = \{4, 8, 16, 32\}$. Note that this CAT is a single model that can work with different chunk sizes at once, offering different compute-quality trade-offs at test-time.

**Language modeling and understanding benchmarks:** Table 2 reports the zero-shot perplexity against LAMBADA (LMB) (Paperno et al., 2016), WikiText (Wiki) (Merity et al., 2016), and on a held-out test set of FineWeb-Edu (FW), and the zero-shot accuracies on key common-sense reasoning benchmarks; Appendix D.2 expands the acronyms in Table 2. All CAT variants outperform existing efficient baselines on common-sense reasoning benchmarks on average. CATs-4/8/16 match or outperform all the baselines on the language modeling tasks except LMB. Note that CAT-32 outperforms other CAT variants on common-sense reasoning and language understanding benchmarks since these evaluations only consider short sequences ($\leq$ 30 tokens on average) – this means the decoder in case of CAT-32 directly consumes the raw sequence without any compression. We test language understanding on longer contexts in Table 4 on a suite of tasks from LongBench (Bai et al., 2023) where CATs-4/8/16 outperform all the baselines, and we observe the expected trends between different CATs.

| | Single-doc QA | | Multi-doc QA | | Few Shot | | Avg. |
|---|---|---|---|---|---|---|---|
| **Model** | QAS | MQA | HQA | 2WMQ | TQA | TREC | |
| Dense | 3.9 | 12.2 | 6.9 | **10.8** | 11.2 | 10.6 | 9.3 |
| Sparse | 5.1 | 11.0 | 7.0 | 10.6 | 10.5 | 5.6 | 9.3 |
| Mamba2 | 4.1 | 11.9 | **7.6** | 7.6 | 9.0 | 7.6 | 8.0 |
| GDN | **8.3** | **15.5** | 6.0 | 7.9 | 7.4 | 8.3 | 8.9 |
| GDN-Hybrid | 4.2 | 13.3 | 6.6 | 11.6 | 11.8 | 6.5 | 9.0 |
| CAT-4 | 5.6 | 12.7 | 7.4 | 9.9 | 12.1 | **35.6** | **13.9** |
| CAT-8 | 5.5 | 11.0 | 6.1 | 8.0 | **12.4** | 29.5 | 12.1 |
| CAT-16 | 4.3 | 14.1 | 6.1 | 5.6 | 10.5 | 16.6 | 9.5 |
| CAT-32 | 4.7 | 11.0 | 7.0 | 6.6 | 10.0 | 8.3 | 7.9 |

Table 4: Zero-shot evaluation of baselines on suite of tasks from LongBench Bai et al. (2023) up to 4K tokens. Refer to Appendix D.2. CAT-4/8/16/32 are a single model.

**Real world in-context recall:** Table 3 reports results on real-world in-context recall tasks from Arora et al. (2024a). Linear models (Mamba2, GatedDeltaNet) lag far behind dense attention, while GDN-Hybrid reduces the gap. CAT surpasses nearly all efficient baselines, benefiting from the gracefully growing memory. Interestingly, CAT outperforms even the dense transformer at moderate chunk sizes ($C = 4, 8$), while being at least $1.4\times$ faster and $2.2\times$ more memory efficient. Figure 1 reports these results, with more details in App. D.4. More importantly, CAT achieves these strong results at varying compute-memory budgets using a single adaptive model.

Table 5: Accuracy on RULER Hsieh et al. (2024) and BabiLong Kuratov et al. (2024) benchmarks. All CATs reported are a single model.

| | S-NIAH-N | | | S-NIAH-U | | | BabiLong | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | **1K** | **2K** | **4K** | **1K** | **2K** | **4K** | **0K** | **1K** | **2K** | **4K** |
| Dense | 96.0 | 92.0 | 43.0 | **93.6** | 55.7 | 19.8 | **49.0** | 14.0 | 12.0 | 1.0 |
| Sparse | 51.2 | 46.2 | 5.0 | 12.8 | 1.4 | 0.8 | 29.0 | 22.0 | 6.0 | 4.0 |
| Mamba2 | 97.7 | 81.1 | 18.6 | 46.7 | 4.6 | 1.0 | 30.0 | 18.0 | 19.0 | 0.0 |
| GDN | 84.7 | 69.1 | 13.6 | 38.9 | 2.6 | 2.0 | 48.0 | **36.0** | **31.0** | **6.0** |
| GDN-Hybrid | **99.0** | **97.0** | 44.0 | 50.9 | 5.6 | 2.6 | 35.0 | 10.0 | 2.0 | 1.0 |
| CAT-4 | 96.0 | **97.0** | **96.0** | 79.6 | **59.3** | 46.5 | 46.0 | 22.0 | 9.0 | 1.0 |
| CAT-8 | 90.0 | 93.0 | 91.0 | 68.1 | 57.5 | **47.3** | 46.0 | 19.0 | 9.0 | 5.0 |
| CAT-16 | 76.0 | 72.0 | 70.0 | 10.0 | 6.6 | 3.8 | 31.0 | 5.0 | 8.0 | 5.0 |
| CAT-32 | 60.0 | 37.0 | 31.0 | 0.0 | 0.0 | 0.0 | 17.0 | 10.0 | 7.0 | 5.0 |

**Synthetic needle-in-haystack & State-tracking:** Table 5 reports results on RULER (Hsieh et al., 2024) synthetic single-needle tasks: S-NIAH-N (recall number worth 7 tokens from the long context) and the harder variant S-NIAH-U (recall a long alpha-numeric string or UUID containing 32 tokens from the long context).

Linear recurrent models (Mamba2, GDN) struggle at longer contexts, and while GDN-Hybrid narrows the gap with dense transformers, performance still drops at longer contexts. CATs-4/8/16 outperform

the efficient baselines as context length increases, showing slower degradation with length, even compared to the dense transformer. This slow degradation can possibly be attributed to reduced sequence length in CAT that leads to fewer distractions for attention (Barbero et al., 2024; Chiang & Cholak, 2022; Golovneva et al., 2025). Further, large-chunk CAT underperforms at short contexts but interestingly surpass baselines at long ones. One reason why large-chunk CAT underperforms could be due to ineffective compression – due to larger chunks, the compressor in CAT is not always able to surface the *right* information in the chunk representation for accurate retrieval. More pretraining or finetuning on specific task data alleviates this problem for large chunk CAT (see App. A.6). That being said, note that there is an upper limit to how much information fixed sized chunk representations can practically learn to hold for large token chunks. On the BabiLong state-tracking and retrieval task (`qa1` subset), all models decline as context grows, although linear recurrent models (Mamba2, GDN) perform better, in accordance with observations in Kuratov et al. (2024).

**Benchmarking generation:** Figure 3 compares architectures as one scales the sequence length, with a fixed batch-size of 320. CAT generates sequences $1.4 - 3.2\times$ **faster** than the dense transformer while showcasing **upto** $2.2 - 9.5\times$ **lower total memory usage** as one increases chunk sizes, despite using more parameters due to wider decoder and the additional compressor. This is not surprising since the major bottlenecks during generation are: (a) KV cache size that drives the main memory requirement during generation and not the parameter count (see discussion in Section 5), (b) memory accesses required for a token, and (c) FLOPs used per token determined by the past tokens being attended to. CATs reduce all the above factors significantly despite carrying more parameters overall. Refer to appendix D.3 for benchmark details.

**CATs scale similar to their dense counterparts:** Figure 5 demonstrates that CATs scale similar to their dense transformer equivalents. We evaluate against three dense transformer scales $\{31M, 92M, 260M\}$, with their CAT equivalents containing parameters $\{95M, 326M, 1000M\}$. All models were trained for 15B tokens, under the identical setup in described in Section 4.

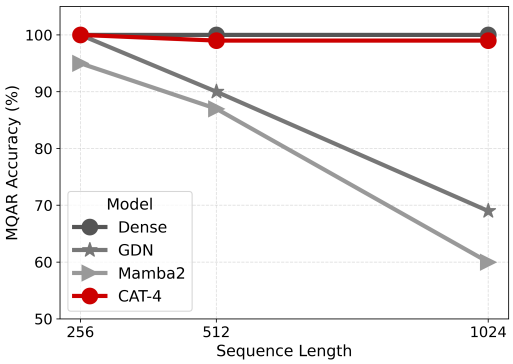


Figure 4: Comparison of architectures on MQAR task (up to $4\times$ standard length). All models are memory-matched in bytes at every point (except dense transformer); CAT outperforms baselines especially at longer sequences, while still using same memory.

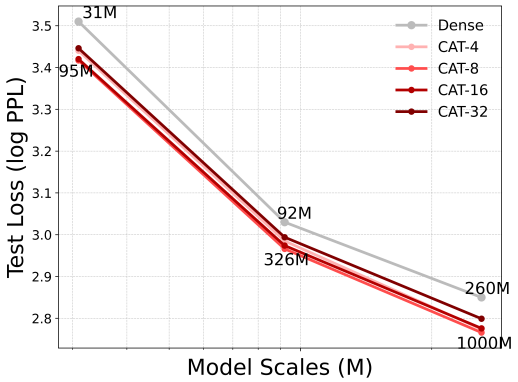


Figure 5: CATs scale like their dense transformer counterparts while being up to $3\times$ faster and $9\times$ more memory-efficient. All CAT points come from a single model at a particular scale, evaluated at different chunk sizes.

**Ablations:** We investigate how different choices affect performance of CATs in App. C. We further provide performance of CATs when trained independently for a single chunk size in App. A.3.

**Instantiating CAT as a layer:** While CAT presented in the paper is a separate architecture, one can take the core concepts and instantiate CAT as a layer that can be swapped in any sequence model, replacing dense attention. This can unlock lots of interesting possibilities starting with creating hybrid as well as adaptive architectures that mixes CAT layers alongside dense attention, or perhaps even linear attention. We leave this open for future work. App. A.5 reports results when instantiating CAT as a layer on MQAR task along with more details. We provide a scalable and efficient implementation for CAT as a layer in the released code.

### 4.1 More comparisons with baselines

**CATs outperform baselines when memory matched at every sequence length:** To rule out any memory discrepancy (Fig. 3), we evaluate CAT and baselines on the challenging MQAR task (Arora et al., 2023a), matching memory budgets down to the level of bytes, and stress-test up to 1K sequence length ($4\times$ standard); Figure 4 reports these results. Baselines are grid-searched over learning rates. Linear models show degradation at longer contexts, while CATs remain near-perfect, thanks to the flexible yet efficient memory. We use the setup described in App. A.4.

**MegaByte/Block Transformer struggle at in-context recall:** The MegaByte/Block Transformer (Ho et al., 2024; Yu et al., 2023) has elements similar to CAT but fails to solve a simple in-context recall task in fig. 7 across different hyperparameters and architecture configurations due to the fixed memory bottleneck. In fact, the block transformer overfits on the task. CATs alleviate the memory bottleneck, allowing it to solve the task, with even lower memory requirements. See appendix A.2 for details.

## 5 Discussion and Conclusion

We introduce **C**ompress & **A**ttend **T**ransformers (CATs), a simple *controllably* efficient alternative to the standard transformer architecture. On language modeling tasks, common-sense reasoning, in-context recall and long-context understanding, CAT outperforms various existing efficient baselines, across different compute-memory budgets. Notably, CAT-4 (the least efficient setting) outperforms the dense transformer in both language modeling and recall tasks while being $1.5\times$ faster and requiring $2\times$ less memory. We discuss a possible explanation for this observation, followed by the practical utility of CATs and some future directions.

**Parameters and Efficiency.** Despite the larger parameter count in CATs, working with compressed sequences ensures that CATs are faster and memory efficient than their dense transformer equivalent. In spirit, CATs are similar to flagship models that are all parameterized as Mixture-of-Experts (MoEs) (Shazeer et al., 2017), where increased parameter counts in MoEs (upto $10\times$ more[3]) does not mean higher computational costs during inference. In fact, more parameters brings improvement in quality while using the same compute (Muennighoff et al., 2024). Due to compression, CATs utilize their increased model parameters *smartly*, bringing improvements in quality (in-context recall) while still being efficient.

**Are CATs adoptable?** Current implementation for training CATs costs twice as much time due to inefficient compilation (using recent PyTorch FlexAttention API (Dong et al., 2024)) of attention mask employed in CATs (see Appendix B.5 for a discussion); custom kernels can be developed in future to mitigate this difference and potentially realize compute savings during pre-training discussed in Section 2.1. However, training is a one time cost, and the service life of models dictates profits, making *serving costs the more important consideration*. Deploying language models at scale is often constrained *not* by model weights but by the memory footprint of their KV cache. For instance, `Qwen3-14B` at a modest batch size of 16, which is common in chat/code completion, requires an *order of magnitude more memory* for the KV cache than the model weights themselves: 28GB for the weights vs. $\sim$ 670GB for the KV cache at maximum context length. In contrast, a CAT variant of the same model could reduce total memory usage upto $\sim 4\times$ despite having more model parameters overall[4], and lead to higher generation throughput. As most modern GPU workloads are increasingly memory-bound rather than compute-bound, memory reductions play an even more critical role (Gholami et al., 2024). The reduction in memory and increase in throughput is more pronounced at larger batch sizes for CATs, which are critical for workloads such as synthetic data generation (Maini et al., 2025) and large-scale rollouts in reinforcement learning (RL) post-training pipelines for math/code reasoning or alignment for faster RL training (Noukhovitch et al., 2024) or better RL optimization (Zhang & Ranganath, 2025). Further, note that CATs serve as multiple models in one, enabling reduced compute during high traffic, longer shelf-life under smaller budgets, and deployment on cheaper hardware – **all from a single training run**.

[3] `Qwen3-30B-A3B`: only 3B parameters are *active* during inference out of 30B parameters in total

[4] Total memory usage for CATs: $28 \cdot (4 + \frac{1}{4}) + \frac{670 \cdot 2}{32} = 160$GB, which is $\sim 4\times$ better at chunk size $C = 32$

**Future work:** CATs currently rely on dense transformer abstractions, but the architecture is general and could incorporate other sequence mixers directly; for e.g. linear attention as *compressor* with dense attention *decoders* for long-range interactions between the compressed sequence. A different direction is data-dependent adaptivity. CATs, as they stand, require users to choose a chunk size appropriate for their compute and memory budgets. Instead, one could post-train with RL to allow CATs to learn to allocate budget themselves based on the context and the task. Such post-training would enable truly adaptive efficiency. Further, as illustrated before, CATs could possibly be instantiated as a layer instead as a full architecture, where every layer has a separate compressor and decoder. This could enable interesting hybrid and adaptive architectures combining dense attention layers mixed with CAT layers. Additionally, one can train a CAT where compression only kicks in only after 1K tokens (say), resulting in parallel compression of older context only. Finally, scaling up CATs to larger model scales and longer training would enable further insights and better comparisons.

## 6 Reproducibility Statement

We release code for CATs at: `github.com/rajesh-lab/cat-transformer`. We provide exhaustive implementation details for CATs in Section 2.1 and pseudo-code in Appendix B. Further, we provide training details and hyperparameters for baselines in Appendix D. We directly use the official code for implementing and benchmarking baselines.

## 7 Acknowledgments

We would like to thank Neelabh Madan, Saksham Rastogi (`pogs`), Aastha Jain, Raghav Singhal, Zhixuan Lin, Mark Goldstein, Ethan Barron, Anirudh Buvanesh, Atharv Sonwane, Daman Arora and Anshuk Uppal for super useful discussions and feedback. This work was partly supported by the NIH/NHLBI Award R01HL148248, NSF Award 1922658 NRT-HDR: FUTURE Foundations, Translation, and Responsibility for Data Science, NSF CAREER Award 2145542, NSF Award 2404476, ONR N00014-23-1-2634, Optum, and Apple. We would also like to thank the support by IITP with a grant funded by the MSIT of the Republic of Korea in connection with the Global AI Frontier Lab International Collaborative Research.

## References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Simran Arora, Sabri Eyuboglu, Aman Timalsina, Isys Johnson, Michael Poli, James Zou, Atri Rudra, and Christopher Ré. Zoology: Measuring and improving recall in efficient language models. *arXiv preprint arXiv:2312.04927*, 2023a.

Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. Language models enable simple systems for generating structured views of heterogeneous data lakes. *arXiv preprint arXiv:2304.09433*, 2023b.

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*, 2024a.

Simran Arora, Aman Timalsina, Aaryan Singhal, Benjamin Spector, Sabri Eyuboglu, Xinyi Zhao, Ashish Rao, Atri Rudra, and Christopher Ré. Just read twice: closing the recall gap for recurrent language models. *arXiv preprint arXiv:2407.05483*, 2024b.

Jacob Austin, Sholto Douglas, Roy Frostig, Anselm Levskaya, Charlie Chen, Sharad Vikram, Federico Lebron, Peter Choy, Vinay Ramasesh, Albert Webson, and Reiner Pope. How to scale your model. 2025. Retrieved from https://jax-ml.github.io/scaling-book/.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Federico Barbero, Andrea Banino, Steven Kapturowski, Dharshan Kumaran, João Madeira Araújo, Oleksandr Vitvitskyi, Razvan Pascanu, and Petar Veličković. Transformers need glasses! information over-squashing in language tasks. *Advances in Neural Information Processing Systems*, 37:98111–98142, 2024.

Loïc Barrault, Paul-Ambroise Duquenne, Maha Elbayad, Artyom Kozhevnikov, Belen Alastruey, Pierre Andrews, Mariano Coria, Guillaume Couairon, Marta R Costa-jussà, David Dale, et al. Large concept models: Language modeling in a sentence representation space. *arXiv preprint arXiv:2412.08821*, 2024.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663*, 2024.

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

Lucas Beyer, Pavel Izmailov, Alexander Kolesnikov, Mathilde Caron, Simon Kornblith, Xiaohua Zhai, Matthias Minderer, Michael Tschannen, Ibrahim Alabdulmohsin, and Filip Pavetic. Flexivit: One model for all patch sizes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14496–14506, 2023.

Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 7432–7439, 2020.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://openreview.net/forum?id=kp1U6wBPXq.

David Chiang and Peter Cholak. Overcoming a theoretical limitation of self-attention. *arXiv preprint arXiv:2202.12172*, 2022.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Zihang Dai, Guokun Lai, Yiming Yang, and Quoc Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. *Advances in neural information processing systems*, 33:4271–4282, 2020.

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

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.

Fnu Devvrit, Sneha Kudugunta, Aditya Kusupati, Tim Dettmers, Kaifeng Chen, Inderjit Dhillon, Yulia Tsvetkov, Hannaneh Hajishirzi, Sham Kakade, Ali Farhadi, and Prateek Jain. Matformer: Nested transformer for elastic inference. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@NeurIPS 2023)*, 2023. URL https://openreview.net/forum?id=93BaEweoRg.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*, 2024.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. *arXiv preprint arXiv:1903.00161*, 2019.

Daniel Y Fu, Tri Dao, Khaled K Saab, Armin W Thomas, Atri Rudra, and Christopher Ré. Hungry hungry hippos: Towards language modeling with state space models. *arXiv preprint arXiv:2212.14052*, 2022.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *arXiv preprint arXiv:2502.05171*, 2025.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

Olga Golovneva, Tianlu Wang, Jason Weston, and Sainbayar Sukhbaatar. Multi-token attention. *arXiv preprint arXiv:2504.00927*, 2025.

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

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

Kai Han, An Xiao, Enhua Wu, Jianyuan Guo, Chunjing Xu, and Yunhe Wang. Transformer in transformer. *Advances in neural information processing systems*, 34:15908–15919, 2021.

Namgyu Ho, Sangmin Bae, Taehyeon Kim, Hyunjik Jo, Yireun Kim, Tal Schuster, Adam Fisch, James Thorne, and Se-Young Yun. Block transformer: Global-to-local language modeling for fast inference. *Advances in Neural Information Processing Systems*, 37:48740–48783, 2024.

Richard Zou Horace He. functorch: Jax-like composable function transforms for pytorch. https://github.com/pytorch/functorch, 2021.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

Sukjun Hwang, Brandon Wang, and Albert Gu. Dynamic chunking for end-to-end hierarchical sequence modeling. *arXiv preprint arXiv:2507.07955*, 2025.

Samy Jelassi, David Brandfonbrener, Sham M Kakade, and Eran Malach. Repeat after me: Transformers are better than state space models at copying. *arXiv preprint arXiv:2402.01032*, 2024.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

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.

Yuri Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. Babilong: Testing the limits of llms with long context reasoning-in-a-haystack, 2024.

Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, et al. Matryoshka representation learning. *Advances in Neural Information Processing Systems*, 35:30233–30249, 2022.

Adrian Łańcucki, Konrad Staniszewski, Piotr Nawrot, and Edoardo M Ponti. Inference-time hyper-scaling with kv cache compression. *arXiv preprint arXiv:2506.05345*, 2025.

Heejun Lee, Geon Park, Youngwan Lee, Jaduk Suh, Jina Kim, Wonyoung Jeong, Bumsik Kim, Hyemin Lee, Myeongjae Jeon, and Sung Ju Hwang. A training-free sub-quadratic cost transformer model serving framework with hierarchically pruned attention. *arXiv preprint arXiv:2406.09827*, 2024.

Heejun Lee, Geon Park, Jaduk Suh, and Sung Ju Hwang. Infinitehip: Extending language model context up to 3 million tokens on a single gpu, 2025. URL https://arxiv.org/abs/2502.08910.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.

Colin Lockard, Prashant Shiralkar, and Xin Luna Dong. Openceres: When open information extraction meets the semi-structured web. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 3047–3056, 2019.

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

Pratyush Maini, Vineeth Dorna, Parth Doshi, Aldo Carranza, Fan Pan, Jack Urbanek, Paul Burstein, Alex Fang, Alvin Deng, Amro Abbas, et al. Beyondweb: Lessons from scaling synthetic data for trillion-scale pretraining. *arXiv preprint arXiv:2508.10975*, 2025.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

Niklas Muennighoff, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Jacob Morrison, Sewon Min, Weijia Shi, Pete Walsh, Oyvind Tafjord, Nathan Lambert, et al. Olmoe: Open mixture-of-experts language models. *arXiv preprint arXiv:2409.02060*, 2024.

Piotr Nawrot, Szymon Tworkowski, Michał Tyrolski, Łukasz Kaiser, Yuhuai Wu, Christian Szegedy, and Henryk Michalewski. Hierarchical transformers are more efficient language models. *arXiv preprint arXiv:2110.13711*, 2021.

Piotr Nawrot, Jan Chorowski, Adrian Łańcucki, and Edoardo M Ponti. Efficient transformers with dynamic token pooling. *arXiv preprint arXiv:2211.09761*, 2022.

Piotr Nawrot, Adrian Łańcucki, Marcin Chochowski, David Tarjan, and Edoardo M Ponti. Dynamic memory compression: Retrofitting llms for accelerated inference. *arXiv preprint arXiv:2403.09636*, 2024.

Piotr Nawrot, Robert Li, Renjie Huang, Sebastian Ruder, Kelly Marchisio, and Edoardo M Ponti. The sparse frontier: Sparse attention trade-offs in transformer llms. *arXiv preprint arXiv:2504.17768*, 2025.

Michael Noukhovitch, Shengyi Huang, Sophie Xhonneux, Arian Hosseini, Rishabh Agarwal, and Aaron Courville. Asynchronous rlhf: Faster and more efficient off-policy rl for language models. *arXiv preprint arXiv:2410.18252*, 2024.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.

Raghavendra Pappagari, Piotr Zelasko, Jesús Villalba, Yishay Carmiel, and Najim Dehak. Hierarchical transformers for long document classification. In *2019 IEEE automatic speech recognition and understanding workshop (ASRU)*, pp. 838–844. ieee, 2019.

Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International conference on machine learning*, pp. 4055–4064. PMLR, 2018.

Guilherme Penedo, Hynek Kydlíček, Anton Lozhkov, Margaret Mitchell, Colin A Raffel, Leandro Von Werra, Thomas Wolf, et al. The fineweb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37:30811–30849, 2024.

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

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. In *International Conference on Machine Learning*, pp. 28043–28078. PMLR, 2023.

Qwen. Qwen3-next: Towards ultimate training & inference efficiency, 2025. URL https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd&from=research.latest-advancements-list. Accessed: 2025-09-18.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SylKikSYDH.

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.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *arXiv preprint arXiv:1806.03822*, 2018.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Kevin Slagle. Spacebyte: Towards deleting tokenization from large language modeling. *Advances in Neural Information Processing Systems*, 37:124925–124950, 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.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

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.

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.

Roger Waleffe, Wonmin Byeon, Duncan Riach, Brandon Norick, Vijay Korthikanti, Tri Dao, Albert Gu, Ali Hatamizadeh, Sudhakar Singh, Deepak Narayanan, et al. An empirical study of mamba-based language models. *arXiv preprint arXiv:2406.07887*, 2024.

Dustin Wang, Rui-Jie Zhu, Steven Abreu, Yong Shan, Taylor Kergan, Yuqi Pan, Yuhong Chou, Zheng Li, Ge Zhang, Wenhao Huang, et al. A systematic analysis of hybrid linear attention. *arXiv preprint arXiv:2507.06457*, 2025.

Kaiyue Wen, Xingyu Dang, and Kaifeng Lyu. Rnns are not transformers (yet): The key bottleneck on in-context retrieval, 2024. URL https://arxiv.org/abs/2402.18510.

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

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=r8H7xhYPwz.

Howard Yen. Long-context language modeling with parallel context encoding. Master's thesis, Princeton University, 2024.

Lili Yu, Dániel Simig, Colin Flaherty, Armen Aghajanyan, Luke Zettlemoyer, and Mike Lewis. Megabyte: Predicting million-byte sequences with multiscale transformers. *Advances in Neural Information Processing Systems*, 36:78808–78823, 2023.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025.

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.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

Lily H Zhang and Rajesh Ranganath. Preference learning made easy: Everything should be understood through win rate. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=UA8DESerfC.

## TABLE OF CONTENTS FOR THE APPENDICES

# A More experiments

## A.1 Recall evaluation

Here, we evaluate all baselines on all datasets from the EVAPORATE suite of tasks that tests for real-world in-context recall.

| Model | SWDE | FDA | Squad | TriviaQA | Drop | Avg. |
|---|---|---|---|---|---|---|
| Dense | 43.4 | 19.7 | 31.0 | 15.0 | 19.4 | 26.7 |
| Sparse | 20.9 | 6.0 | 20.7 | 15.2 | 19.3 | 16.4 |
| Mamba2 | 13.5 | 4.5 | 24.9 | 13.9 | 17.8 | 14.9 |
| GDN | 18.0 | 6.8 | 25.5 | **15.5** | 17.2 | 16.6 |
| GDN-H1 | 44.0 | 17.8 | **32.9** | 15.4 | **19.8** | 26.0 |
| CAT-4 | **49.1** | **45.1** | 28.3 | 15.0 | 17.9 | **31.1** |
| CAT-8 | 38.2 | 34.8 | 25.9 | 14.0 | 18.3 | 26.2 |
| CAT-16 | 27.5 | 15.4 | 20.4 | 14.8 | 16.9 | 18.9 |
| CAT-32 | 13.2 | 3.2 | 15.8 | 13.0 | 14.3 | 11.9 |

Table 6: Zero-shot performance on real-world in-context recall tasks from EVAPORATE suite, measured at 2K sequence lengths. Note that only SWDE and FDA have long token sequences among the datasets in the suite (others have an average length of $\leq 300$ tokens Arora et al. (2024b)).

## A.2 Comparison with MegaByte/Block Transformer

In figure 7, we evaluate in-context recall ability for Block Transformer architectures Ho et al. (2024); Yu et al. (2023), that model chunks of tokens similar to CATs but with a subtle but salient difference in the architecture circuit (that we explain below). For this experiment, we test on the MQAR task (a synthetic needle-in-haystack task Arora et al. (2023a)) on a modest sequence length of 256. We test the accuracy of retrieving just 4 needles. We parametrize components of Block Transformer that is: global model and local model using a transformer, the embedder is a look-up table or a transformer. We keep the patch size/chunk size as 4 – same as CAT. We keep the identical training setup for both architectures. We grid search for hyper-parameters (`lr`, `hidden_size`, and embedder parameterization), even **using more memory** than the CAT baseline, in its global decoder. Even in these simple settings and added advantage, Block Transformer Ho et al. (2024); Yu et al. (2023) fails to solve the task (fig. 7) – instead the model starts to memorize the train points, as seen from train loss and train accuracy – train metrics keep getting better, however, test metrics suffer.

CATs directly pass all the "local" patch/chunk representations directly to the decoder, unlike the block transformer that forces the history to be compressed into fixed dimensional representation. This design choice helps CAT *alleviate the memory bottleneck* that Ho et al. (2024) suffers from where the architecture must compress everything from the past into a single "global" representation to generate the next chunk. Note that this different design choice in CATs does not introduce any memory/compute overhead compared to Block Transformer Ho et al. (2024), it just changes the circuit of the architecture. In fact, CATs don't utilize three different components (embedder, global decoder, local decoder) – it only uses a compressor and a decoder, reducing the design space and (significant) parameter requirements further.

## A.3 CATs trained with fixed chunk size

Table 7 reports results for different CATs when trained with a fixed chunk size. We observe fixed chunk CATs exhibit better in-context recall performance compared to the single adaptive CAT model. This drop in performance in single adaptive CAT model could be attributed to the decoder losing some of its capacity to model multiple chunk sizes at once. More pre-training can alleviate some of this performance drop. Nevertheless, despite this drop, single adaptive CAT model still outperforms all baselines.

| Model | SWDE | FDA | Squad | TriviaQA | Drop | Avg. |
|---|---|---|---|---|---|---|
| CAT-4 | 49.1 | 45.1 | 28.3 | 15.0 | 17.9 | 31.1 |
| CAT-8 | 38.2 | 34.8 | 25.9 | 14.0 | 18.3 | 26.2 |
| CAT-16 | 27.5 | 15.4 | 20.4 | 14.8 | 16.9 | 18.9 |
| CAT-32 | 13.2 | 3.2 | 15.8 | 13.0 | 14.3 | 11.9 |
| CAT-4 (fixed) | **50.9** | **55.5** | **30.2** | 15.6 | 20.4 | **34.5** |
| CAT-8 (fixed) | 43.3 | 51.6 | 26.9 | 15.4 | **21.4** | 31.7 |
| CAT-16 (fixed) | 32.7 | 23.3 | 22.6 | **15.8** | 16.4 | 22.2 |
| CAT-32 (fixed) | 10.7 | 5.4 | 15.6 | 13.4 | 16.6 | 12.3 |

Table 7: In-context recall performance comparison of single adaptive CAT model and independently trained CATs with fixed chunk size. First four rows are a single CAT model, whereas the last four rows are separate CAT models, trained with a fixed chunk size. Fixed chunk size CATs outperform their adaptive versions.

### A.4 Sparse or Sliding Window Attention struggle at recall

We evaluate models on the synthetic multi-associate query recall (MQAR) task, proposed in Arora et al. (2023a) and further popularized in Arora et al. (2024a). All models use depth of 2 layers, and are trained and tested on sequence lengths upto 256 having varying number of key-value pairs. CAT models use a 1 layer compressor, followed by a 2 layer decoder, with a chunk size of 4, both using model dimension of $D = D_g = 64$ in this case. Note that the state size for CAT is $\frac{N}{C} \cdot D = 4096$ for this particular sequence length and model dimension. Sparse attention uses a chunk size of 4 (for fair comparison with CAT); Sliding window uses a window size of 64.

| Method | Solves? | State Size |
|---|---|---|
| Dense | ✓ | 16384 |
| Sparse | ✗ | 4096 |
| Sliding Window | ✗ | 4096 |
| CAT | ✓ | 4096 |

Table 8: For each method, we report the state size at which the particular method was trained for the MQAR task. Each method was grid searched for best possible hyper-parameters. We use the state size calculations provided in Arora et al. (2024a; 2023a).

In table 8, CAT is able to solve the MQAR task. Notably, we find the sparse attention as well as sliding window attention fail to solve the task at 2 layers, highlighting their dependence on depth.

### A.5 CAT as a layer

To instantiate CAT as a seperate layer in itself, we parameterize the *compressor* as a simple linear projection. We use the dense attention mechanism itself as the *decoder*. Before applying the compression and decoding from compressed chunk representations, we artificially up-project the input embeddings in the layer – this is done following the observation in the main paper that decoding from compressed representations requires more compute. Please find the actual implementation in the released code.

Table 9 reports MQAR accuracy when CAT is used as a layer. We use a fixed chunk size of 4 in this experiment. We use 2 layers of CAT. We follow the same setup described in Appendix A.4.

| Method | Solves? | State Size |
|---|---|---|
| Dense | ✓ | 16384 |
| CAT | ✓ | 4096 |
| CAT (layer) | ✓ | 4096 |

Table 9: CAT instantiated as a seperate layer solves the MQAR task again.

### A.6 FINETUNING CATS ON S-NIAH-U

S-NIAH-U is a task where model needs to recall 32 token long UUID strings from the long context. This section reports performance of CATs after task specific finetuning on samples from S-NIAH-U. We only apply the loss on tokens that appear in the answer span. Table 10 reports these results. This is accompanied by loss curves for different CATs depending on chunk size in Figure 6 on this task.

We observe two things: (i) after finetuning, performance goes up significantly for all chunk sizes. This signifies as chunk size increased, compressor in CATs, before finetuning, was not surfacing the *right* information in the chunk representation. (ii) the loss curves during finetuning indicate the same as well, however it still does not go completely to zero, especially for CAT-32. This indicates that there are limits to what information a fixed sized chunk representation can practically learn to surface, justifying its sub-par accuracy on the task.

This problem of not surfacing the *right* information in the chunk representation could be alleviated by more and longer pre-training, or choosing smaller chunk sizes for tasks that require accurate recall.

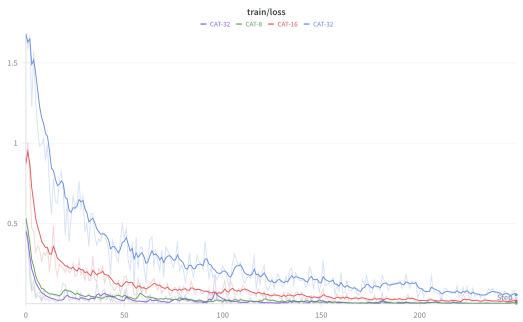


Figure 6: Loss curves when finetuning different CATs on samples from S-NIAH-U task.

| Model | Before | After |
|---|---|---|
| CAT-4 | 46.3 | 97.1 |
| CAT-8 | 47.3 | 97.0 |
| CAT-16 | 3.8 | 94.2 |
| CAT-32 | 0.0 | 64.3 |

Table 10: Performance on 4K sequence length before and after finetuning for different CAT variants.

### A.7 HOW ARE CATS DIFFERENT FROM BLOCK TRANSFORMER/MEGABYTE?

Works like Ho et al. (2024); Yu et al. (2023) break up the modeling of a sequence into chunks/-patches, where each chunk is modeled independently of each other given the previous "global" chunk embedding. An embedder first compresses each chunk independently, then these "local" chunk embeddings are passed to a "global" model where each "local" chunk embedding attends to past "local" chunk embeddings, forming a "global" chunk embedding. Each "global" chunk embedding is then passed to a decoder that is responsible for generating the next chunk.

On first glance, **CATs might appear similar to above works**, specifically Block Transformer Ho et al. (2024)/ MegaByte Yu et al. (2023), however the subtle but salient difference is: one directly feeds **all the previous "local" chunk/patch representations** directly to the decoder in CAT, whereas in works like Ho et al. (2024), one feeds in just the previous "global" chunk representation outputted by a "global" model to the decoder (refer to the comparative figure in **??**).

This architectural choice of passing *all* the compressed local chunks from the past directly to the decoder allows CATs to solve long-range recall tasks with ease while maintaining efficiency, whereas **Block Transformer/MegaByte is plagued by *learnability* problems** (even in toy recall tasks) due to constant size compression of history (see Figure 7). Additionally, CATs don't utilize three different

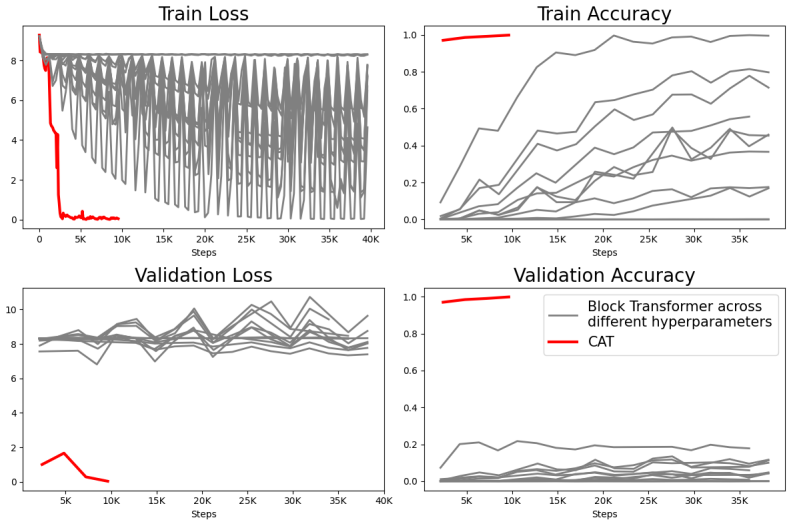


Figure 7: Block Transformer (Ho et al., 2024; Yu et al., 2023) (across different configurations and hyperparameters) fails to solve a simple MQAR task with only 4 key-value pairs, tested on modest sequence length of 256 tokens, possibly due to fixed memory. CAT solves the task with ease due to flexible yet efficient memory. Note that training of CAT stops when it solves the task perfectly.

components (embedder, global decoder, local decoder) – it only uses a compressor and a decoder, reducing the design space and (significant) parameter requirements further.

## B IMPLEMENTATION DETAILS

In this section, we discuss some implementation details regarding CATs. We repeat some text presented in the main paper to be self-contained below.

### B.1 TRAINING

**Training:** While CATs are simple and build on dense transformer abstractions, their naive PyTorch training implementation is very inefficient.

Note that compression of chunks of tokens is efficient since it can be done in parallel, specifically using `torch.vmap`$(f_\theta(\mathbf{c}_i))$ for all chunks $\mathbf{c}_i$. This costs a total of $O(\frac{N}{C} \cdot C^2) = O(NC)$ in self-attention compute, which is much better than $O(N^2)$.

But, computing logits for tokens in chunk $\mathbf{c}_i$, that is computing $g_\theta(\mathbf{c}_i \mid f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1}))$ can be non-trivial since for chunk $\mathbf{c}_i$, we have $i-1$ past chunk representations $\{f_\theta(\mathbf{c}_1), f_\theta(\mathbf{c}_2) \dots f_\theta(\mathbf{c}_{i-1})\}$. In other words, there are different number of past chunk representations for every chunk, making shapes variable and as a result, harder to parallelize computation of logits. One could employ a python loop and compute logits for every chunk sequentially, but that would be slow and won't scale. In fact, even if one manages to compute logits for every chunk in parallel, the total self-attention operations in the decoder would be $O(\sum_{i=1}^{\frac{N}{C}}(i+C)^2) = O((\frac{N}{C})^3)$, that is cubic in sequence length. Padding to make shapes constant would make things worse. Thus, naive techniques will not scale.

*With such difficulties in making the training scalable, it may not be surprising that despite the simplicity of* CATs*, it was not attempted in the community.* Note that unlike CATs, similar architectures Ho et al. (2024); Yu et al. (2023) do not have this problem: computing logits can be naively parallelized due to fixed shapes and self-attention operations scale quadratically due to a single compressed representation for the past.

In CATs, observe that in computing logits chunks $\mathbf{c}_i, \mathbf{c}_{i+1} \dots \mathbf{c}_{\frac{N}{C}}$, one calculates the same key-values for chunk representations $f_\theta(\mathbf{c}_j)$ in the decoder, where $j < i$. This points to repeated and identical computations. To exploit this observation, we take advantage of a custom attention mask in decoder to calculate logits for all chunks in parallel, and reuse computations done for a past chunk representation to be used for a computations for logits for a future chunk. To be concrete, once we calculate all chunk representations $f_\theta(\mathbf{c}_i)$ in parallel using `torch.vmap`, we insert $f_\theta(\mathbf{c}_i)$s at particular positions in the original sequence: after every chunk $\mathbf{c}_i$, we attach its chunk representation. That is, sequence would look like: $\{\mathbf{c}_1, f_\theta(\mathbf{c}_1), \mathbf{c}_2, f_\theta(\mathbf{c}_2), \dots \mathbf{c}_i, f_\theta(\mathbf{c}_i) \dots\}$. Now, we pass this sequence into the decoder during training, with a custom attention mask (see Figure 8) that allows a token in chunk $\mathbf{c}_i$ to attend to previous tokens within that chunk only as well as only to previous chunk representations, which would be $f_\theta(\mathbf{c}_{i-1}), f_\theta(\mathbf{c}_{i-2}) \dots f_\theta(\mathbf{c}_1)$ only. Any token in chunk $\mathbf{c}_i$ does not attend to raw tokens outside this chunk. This implementation allows re-use of key-values for chunk representations $f_\theta(\mathbf{c}_i)$ for calculation of logits of future chunks, in parallel, making the training of CATs efficient and scalable. We utilize the FlexAttention API Dong et al. (2024) to automatically create a custom kernel for the custom mask (Figure 8). Note that this way of computing logits is quadratic in sequence length but with a constant times better: concretely it is $O(\frac{N}{C} \cdot N + \frac{N}{C} \cdot C^2) = O(\frac{N^2}{C})$, **which is $C\times$ better than $O(N^2)$** (yellow dots in figure 8 provides a visual proof for this cost; number of yellow dots are significantly lower than $\frac{N^2}{2}$). Mathematically the cost of attention in CATs decoder is: $\sum_{i=1}^{N}[\frac{i}{C}] + (i \bmod C) + 1 = O(\frac{N^2}{C})$, where $[.]$ is the floor function, and $\bmod$ is modulo operator.

For a discussion in training throughput, refer to a discussion in Appendix B.5.

```
1
2 def forward(input_ids, targets):
3
4     input_ids = einops.rearrange("b (k c) -> b k c", k=num_chunks, c=
          chunk_size)
5
6     # calculate f(x)
7     # shape of fx: (b, k, D_d)
```

```
    fx = torch.vmap(f)(input_ids)

    output_logits = list()
    for i in range(num_chunks): # note that this loop is done in parallel with the custom attention mask presented in the appendix
        # use the previous i+1 fx to predict the current chunk
        # shape of cur_chunk_logits: (b, 1, l, V)
        cur_chunk_logits = phi(input_ids[:, i, :], fx[:, :i+1, :])
        output_logits.append(cur_chunk_logits)
    output_logits = torch.cat(output_logits, dim=1) # shape: (b, k, c, V)
    output_logits = einops.rearrange(output_logits, "b k c v -> b (k c) v") # arrange all chunks logits together (or flatten)
    return torch.nn.functional.cross_entropy(output_logits, targets) # return the loss
```

Listing 1: Pseudocode for training step

### B.2 ATTENTION MASK

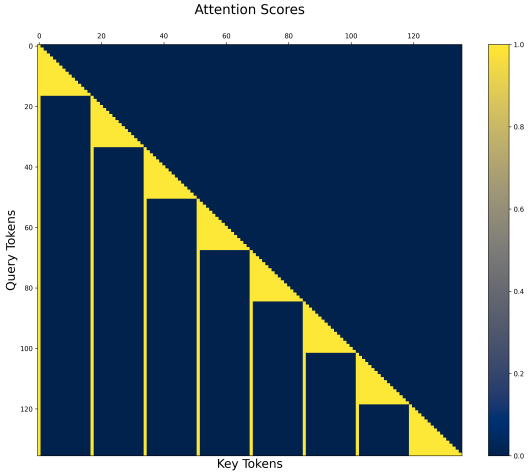


Figure 8: Sequence length is 128, and the chunk size that we use in this particular attention mask is $C = 16$.

Note that attention mask in figure 8 looks very similar to the attention mask as defined in Child et al. (2019), however, in CAT's case: (a) it is not heuristic choice, and (b), tokens in a particular chunk attend to the past $f_\theta(\mathbf{c}_i)$ representations obtained by the compressor, rather than the past token embeddings at that position as done in Child et al. (2019).

### B.3 GENERATION

The decoder during generation attends to atmost $\frac{N}{C} + C$ tokens. Due to compression, CATs can throwaway past chunks of tokens, and only keep their compressed chunk representations in memory. This straightaway results in a big reduction of memory; the KV cache is slashed by a factor of $C$. For even a moderate chunk size of 4, this results in big reductions in memory during generation (Figure 3) compared to a dense transformer. This slash in memory is accompanied by reduced memory accesses a decoder makes in CATs, which is the major bottleneck during generation.

Implementing generation is simpler than training and very similar to how it occurs for a dense transformer. In fact, a pure PyTorch implementation for CATs is on-par with efficient architectures that utilize custom kernels. We inspire our implementation from: https://github.com/meta-pytorch/gpt-fast. Given $i$ chunks of tokens: firstly, `torch.vmap` over chunks independently to calculate $f_\theta(\mathbf{c}_i)$ in parallel. Then prefill the decoder's KV cache in parallel with the obtained $f_\theta(\mathbf{c}_i)$s. Now generate the next chunk $\mathbf{c}_{i+1}$ autoregressively one token at a time. Note that this uses a simple causal mask since the previous positions are already prefilled with $f_\theta(\mathbf{c}_i)$s, which is required to decode chunk $\mathbf{c}_{i+1}$. Once all the tokens of the chunk $\mathbf{c}_{i+1}$ are generated, calculate $f_\theta(\mathbf{c}_{i+1})$ and prefill the decoder's KV cache just after the position where $f_\theta(\mathbf{c}_i)$ was cached. Now the KV cache is ready for generation of the next chunk $\mathbf{c}_{i+2}$ and this process will continue.

This simple implementation enables CATs to be $1.4 - 3.2\times$ **faster** than the dense transformer while showcasing **upto** $2.2 - 9.5\times$ **lower total memory usage** as one increases chunk sizes.

```

# https://github.com/pytorch-labs/gpt-fast/blob/7dd5661e2adf2edd6a1042a2732dcd3a94064ad8/generate.py#L154
def generate_chunk_by_chunk(
    input_ids
):
    # assume input_ids.shape == (batch_size, 1, chunk_size)

    # declare/reset static KV cache, shape: [batch_size, num_chunks + chunk_size, 2, D_d]

    input_pos = 0

    # compress the first chunk (batch_size, 1, chunk_size) -> (batch_size, 1, D_d)
    # get fx for the very first chunk
    fx = f(input_ids) # shape of fx: (batch_size, 1, D_d)
    next_token = prefill(fx, input_pos) # prefill at idx 0 with fx in g

    new_chunks = list()

    for i in range(num_chunks - 1):

        # generate entire chunk using fx that was prefilled earlier in g
        next_chunk = generate_chunk(next_token)
        new_chunks.append(next_chunk.clone())

        # get new fx
        # compress the new obtained chunk
        fx = f(next_chunk) # (batch_size, 1, chunk_size) -> (batch_size, 1, D_d)

        # prefill again at input_pos
        input_pos += 1
        next_token = prefill(fx, input_pos) # prefill fx at idx `input_pos` in g

    new_chunks = torch.cat(new_chunks)
    return new_chunks
```

Listing 2: Pseudocode for generation

### B.4 ADAPTIVE CATS

To enable training of adaptive CATs, we made some choices that we now describe. In every training iteration, we sample a chunk size uniformly at random and perform loss computation. Further, due to variable size of a chunk in every training iteration, one cannot keep a single projection matrix that projects processed token embeddings in the compressor to a single chunk representation (since shapes for projection matrix would be different for different chunk size). One could tackle this by keeping an independent projection matrix for every chunk size, but we found this didn't work well empirically, possibly due to reduced updates for every chunk size's projection weights (only one chunk size's projection weights are updated per iteration; this is not the case with compressor or the decoder, they are updated every iteration). Instead, we took inspiration from Beyer et al. (2023) where the authors declared a single projection matrix for all chunk sizes, and then linearly interpolated the matrix to the desired shape depending on the current chunk size. This means the linear interpolation is also under `torch.autograd` and is optimized so that the final linearly interpolated projection matrix gives a *good* chunk representation for every chunk size.

### B.5 TRAINING THROUGHPUT ANALYSIS

We make use of FlexAttention API to obtain a custom self-attention kernel specifically for the masking scheme Figure 8. This fused kernel gives a significant boost in training throughput in self-attention costs compared to using a naive PyTorch masked implementation.

That being said, an efficient training kernel can be developed in the future. In our experiments, using FlexAttention did not give significant boosts in training speeds compared to using Flash Attention on a dense transformer. This could be due to the fact that speeding up the attention maps (that we use in figure 8) may require different principles than Flash Attention like optimization that Flex Attention might be using under the hood.

As a result, due to the unavailability of an efficient training kernel, theoretical speed ups due to reduction in attention FLOPs in the CAT architecture don't appear in training wall-clock times. Additionally, MLPs in a transformer drive the majority of the FLOPs budget during training at smaller sequence lengths Austin et al. (2025). At a sequence length of 4096, CATs take $\leq 2.35\times$ to train compared to a dense transformer (measured on batch size of 8 with compressor depth of 3, decoder depth of 6, hidden size for compressor $D_f = 1024$ and hidden size for decoder $D_g = 2D = 2048$ for CAT, compared against dense transformer having depth of 6 and $D = 1024$, on a A100 80 GB PCIe.)

Developing an efficient attention kernel for training CATs is left as future work.

# C ABLATIONS

## C.1 ABLATION ON HIDDEN SIZE OF COMPRESSOR

With this ablation, we show that increasing hidden size of the compressor does not help in improving perplexity. We fix $D_g = 1536$ for these experiments. For this ablation, we use a smaller WikiText-103 dataset. Both compressor and decoder use the same depth $L = 6$.

| **Chunk Size** $C$ | **Size of** $D_f$ | **Perplexity** |
|---|---|---|
| 16 | 768 | 17.6 |
| | 1536 | 17.6 |

Table 11: Comparison of choices of hidden size of compressor on WikiText-103 perplexity.

There is no effect of increasing the hidden size of the compressor. The performance before and after remains the same.

## C.2 ABLATION ON HIDDEN SIZE OF DECODER

We ablate on different choices of $D_g$ along with different chunk sizes in CAT . In this setup, we fix $D_f$ in the compressor, and only vary $D_g$ or $C$ (chunk size). We use WikiText-103 for these experiments. In this setup, $D = 768$. Both compressor and decoder use the same depth of $L = 6$.

| **Chunk Size** $C$ | **Size of** $D_g$ | **Perplexity** |
|---|---|---|
| 4 | $D$ | 19.8 |
| | $2D$ | 17.4 |
| 8 | $D$ | 20.4 |
| | $2D$ | 17.7 |
| 16 | $D$ | 20.2 |
| | $2D$ | 17.6 |

Table 12: Comparison on choices of chunk sizes and sizes of $D_g$ on WikiText-103 perplexity.

We observe that we obtain the best perplexities when we $D_g = 2D$ for the particular chunk size we are using. Using this observation, we used this as our *default* configuration for the FineWeb-Edu experiments.

| **Model** | $D_f$ | $D_g$ | **Perplexity** | **Avg. recall** |
|---|---|---|---|---|
| Dense | $--$ | $D$ | 21.2 | 23.8 |
| CAT | $D$ | $D$ | 23.8 | 13.7 |
| CAT | $D$ | $2D$ | **20.7** | **19.8** |

Table 13: Impact on perplexity and average recall performance of CAT when varying $D_g$. For dense, $D_g$ implies hidden size for itself. Here, $D = 1024$. $D_g = 2D$ gives better perplexity and average recall. We train CAT only at chunk size $C = 8$ for these experiments. All models were trained for 5B tokens with 1K sequence length. Rest of the setup follows Sec. 4.

## C.3 ABLATION ON DEPTH OF THE COMPRESSOR

We ablate on the depth of the compressor. For a fixed chunk-size, $D_f = 768$ (compressor embedding size), $D_g = 1536$ (decoder hidden size), and a fixed depth of the decoder, we vary the compressor depth.

| Chunk Size $C$ | Depth of Compressor | Perplexity |
|---|---|---|
| 8 | 6 | 17.4 |
| | 3 | 17.4 |
| 16 | 6 | 17.8 |
| | 3 | 17.7 |

Table 14: Comparison on choices of depth of the compressor across different chunk sizes $C$ on WikiText-103.

We have an interesting observation that one can reduce the depth of the compressor without sacrificing on the downstream perplexity. This could mean one can compress small chunks of tokens without a requiring high capacity. In our generation benchmarks, we observed that compressor depth play less of a role in latency as compared to the decoder depth (since we compress tokens in parallel using one transformer call). That being said, compressor depth does play a significant role in training costs (due to the MLP training costs in the compressor). Therefore, reducing compressor depth goes into overall advantage for the CAT architecture.

However, what is the limit, and can one go to even a 1 layer of compressor is an interesting question to ask. There might be some lower bound on the compressor depth to start compressing chunks of tokens, but we leave this to future work.

## D MORE EXPERIMENT DETAILS

Here we provide more details about the experiments done in the main text.

### D.1 BASELINES

| Model | Total (M) | Embedding (M) | Non-Embedding (M) |
|---|---|---|---|
| Dense | 260 | 50 | 210 |
| Mamba2 | 260 | 50 | 210 |
| GDN | 310 | 50 | 260 |
| GDN-Hybrid | 280 | 50 | 230 |
| Sparse | 820 | 100 | 720 |
| CAT-4/8/16/32 | 150 + 820 | 50 + 100 | 100 + 720 |

Table 15: Model parameter sizes in millions, separated into embedding and non-embedding parameters. Parameters for CATs consists of cost of compressor + cost of decoder.

1. Dense transformer (or Transformer++) Vaswani et al. (2017); Touvron et al. (2023): We use rotary position embeddings along with the FlashAttention kernel to perform self-attention. The MLP is a SwiGLU MLP Touvron et al. (2023).
2. Sparse transformer Child et al. (2019): Follows the Dense transformer configuration, except the attention mask used. Moreover, we used $D = 2 \cdot 1024 = 2048$ for this baseline for a fair comparison with CATs. We used FlexAttention API to create optimized Flash Attention like kernel for this.
3. MAMBA2 Dao & Gu (2024): The model uses 2 Mamba mixer per layer. All layers use the MAMBA2 block without any mixing any attention. The `expand` is set to 2, $d_{state} = 128$, and convolution $k = 4$. Activations used are `SiLU`. We use the official codebase for MAMBA2 generation throughput and memory benchmarking: `https://github.com/state-spaces/mamba` and code from: `https://github.com/fla-org/flash-linear-attention` for training.
4. Gated Delta Net Yang et al. (2025b): We use the implementation provided at `https://github.com/fla-org/flash-linear-attention` for training. We use `head_dim` as 128 and `num_heads` as 8 (same as MAMBA2 above). For the hybrid version, we use sliding window layers at every other layer with a sliding window size of 2048.

### D.2 DATASETS

Following common practices done in Gu & Dao (2023); Dao & Gu (2024); Arora et al. (2024a); Yang et al. (2025b), we evaluate all models on multiple common sense reasoning benchmarks: PIQA Bisk et al. (2020), HellaSwag Zellers et al. (2019), ARC-challenge Clark et al. (2018), WinoGrande Sakaguchi et al. (2021) and measure perplexity on WikiText-103 Merity et al. (2016)and LAMBADA Paperno et al. (2016). In Table 2, HS denotes HellaSwag, PQ denotes PIQA, AE denotes ARC-Easy, AC denotes ARC-Challenge, WG denotes Winogrande, OQA denotes OpenBookQA, LMB denotes LAMBADA, Wiki denotes WikiText, and FW denotes FineWeb-Edu.

We evaluate on tasks from LongBench Bai et al. (2023) where each abbrevation in table 4 stands for: QAS: `qasper`, MQA: `multifieldqa_en`, HQA: `hotpotqa`, 2WMQ: `2wikimqa`, TQA: `triviaqa`, TREC: `trec` split of LongBench.

To measure real-world recall accuracy, we use datasets used in Arora et al. (2024a;b). Namely these consists of SWDE Lockard et al. (2019) for structured HTML relation extraction and several question answering datasets including SQuAD Rajpurkar et al. (2018), TriviQA Joshi et al. (2017), DROP Dua et al. (2019) and FDA Arora et al. (2023b). Since our pretrained models are small, we use the Cloze Completion Formatting prompts provided by Arora et al. (2024b).

We evaluate on tasks from the needle-in-haystack benchmark RULER Hsieh et al. (2024).

Additionally, we evaluate on datasets from the LongBench benchmark Bai et al. (2023) to evaluate long-context understanding.

Finally, to evaluate baselines on state-tracking tasks, we used the BabiLong benchmark Kuratov et al. (2024). Due to relatively small scale of our setup, we were only able to evaluate on `qa1` subset, since for other complex subsets, all baselines failed.

### D.3 Generation

Both dense transformer and CAT use FlexAttention API causal dot product kernels. We use the script provided in Dao & Gu (2024) to benchmark[5] Mamba2, GatedDeltaNet and GatedDeltaNet-Hybrid. All benchmarks used a prefill of 8 tokens. All benchmarks were run using a single NVIDIA A100 80GB PCIe, and use CUDA cache graphs for the next-token prediction.

### D.4 Main figure details

Figure 1 reports memory usage at 2K sequence length since both SWDE and FDA datasets have queries with context length $\leq$ 2K. The latencies are reported at maximum sequence length of 4K.

[5] github.com/state-spaces/mamba

## E Extended Related Work

**Reducing self-attention costs:** Reducing the cost of self-attention enables scaling transformers to large contexts and has been the focus of much work Child et al. (2019); Parmar et al. (2018); Beltagy et al. (2020); Jiang et al. (2023). Common techniques include *heuristically* defined sparse attention maps Child et al. (2019); Zaheer et al. (2020) or a sliding window Jiang et al. (2023) in order to reduce the tokens being attended to. The compute required (and in some cases, memory) for attention go down, however, compromising with the expressivity of the model. In turn, to achieve performance similar to that of full-attention, efficient models either require big window sizes (making their memory costs large again) (Arora et al., 2024a) or more layers (in case of sparse or sliding window attention, see App. A.4 and Tab. 3).

Shazeer (2019) proposes use of single or reduced key and value heads in the self-attention block, more commonly known as Grouped Query Attention (only one key/value head) or Multi Query Attention (reduced key/value heads). This results in reduction of memory with seemingly no loss in downstream performance, making this a popular choice in latest model releases Yang et al. (2025a). That being said, one could use the same technique inside CAT's decoder (and compressor) self-attention block, making it complimentary.

Concurrent works like Yuan et al. (2025) reduce attention compute by attending to compressed past tokens as well as to specific blocks of uncompressed tokens in the past. This is similar in spirit to our work, however, in the case of Yuan et al. (2025), there are no memory savings during inference.

Some works Rae et al. (2020); Chevalier et al. (2023) explored recurrent formulations of a transformer to enable processing of longer sequences on limited compute by compressing past context. However, training sequence models in a recurrent fashion has its own challenges, back-propagation through time (BPTT) being the most important one. More recently Geiping et al. (2025) had to use very careful weight initialization, truncated gradients, small learning rates and careful placement and tuning of norms to train a large-scale recurrent architecture in a stable manner and prevent optimization collapse. Nevertheless, these techniques are complementary to CAT.

Alternatively, one can optimize the computation of full-attention to directly reduce wall-clock time and memory by leveraging hardware advancements. For example, Dao et al. (2022) compute attention in blockwise manner and exploit the nature of online softmax Milakov & Gimelshein (2018) which removes the need to instantiate the entire $QK^T$ matrix and reduce calls to slow-read part of the GPU memory. As we utilize the attention mechanism as is, any reductions in cost due to hardware optimization that apply to the attention mechanism also proportionally reduce the cost of CAT models.

Finally, plethora of works have tackled reducing compute and memory requirements of a transformer in a *post-hoc* manner i.e. after it has been trained using full-attention (also called *training-free* sparse attention Nawrot et al. (2025)). Common techniques include prefill-time sparsification (vertical/s-lash/block; adaptive) and decode-time KV-cache selection/eviction (e.g. Li et al. (2024); Tang et al. (2024)). Further, Lee et al. (2024; 2025) explored offloading of key-value cache on the CPU along with sparsifying attention masks. However, because models are trained dense but run sparse, train-test mismatch can hurt downstream performance (Nawrot et al., 2025). Still, these works can be directly applied to the decoder in CAT, or any other efficient alternatives including hybrid architectures that make use of dense attention, making them complementary. Concurrently, Łańcucki et al. (2025) explored delayed token eviction strategies to enable effective KV cache compression (Nawrot et al., 2024).

**Linear attention and state-space models:** A different line of work reduces the generation cost of transformers by limiting the recurrent state, which is the vector required to decode each token. Self-attention keeps track of the entire context (or the KV cache) meaning that the recurrent state increases in size with each decoded token. Works like Arora et al. (2024a); Katharopoulos et al. (2020) linearize attention to make a fixed-size recurrent state that can be updated via simple averaging; the technique is to approximate self-attention with linear operations of query, key, and value vectors transformed through a feature map. The choice of the feature map falls to the user and approximating attention well requires the feature map to be large in size, which can counteract the gains in computational costs achieved by the linearization.

Alternatively, one can replace attention with linear or pseudo-linear sequence mixers such as state-space models (SSMs) Gu et al. (2021); Sun et al. (2023), gated convolutions Fu et al. (2022); Poli et al. (2023) and input-dependent recurrent Peng et al. (2023); Gu & Dao (2023) and more recently Yang et al. (2025b).

Typical implementations of linear attention and state-space models do achieve impressive reductions in generation costs and memory, but restrict the expressivity to the extent that these models do not solve in-context recall tasks without large recurrent state sizes Arora et al. (2024a; 2023a), or without composing with other sequence mixers, such as local sliding window attention (Arora et al., 2024a; Yang et al., 2025b). Choosing such a composition again falls back to the user, complicating the design process. Additionally, this process trades-off computation costs for performance because the attention layers that improve recall performance also come with larger time and memory costs.

Unlike the works discussed above, CATs require no complicated changes to the attention mechanism itself. Instead of relying manual approximations of history or utilizing any heuristic choice for feature maps, we let the model and optimization decide what the history should be using learned compression. Moreover, its unclear how much memory and compute a downstream task requires, making the adaptive property of CATs much desirable, which no other baselines provide.

**Hierarchical transformers:** Many previous works Pappagari et al. (2019); Han et al. (2021); Dai et al. (2020) have explored employing hierarchy in transformers for creating representations representations for documents/images, where a *local* encoder transformer processed parts of the document/image independently. Later works Nawrot et al. (2021; 2022); Slagle (2024) explored downsample-then-upsample approach (*hour-glass* like structure), where the sequence is downsampled into *coarse* tokens followed by upsampling into *fine-grained* tokens before being decoded. Due to the *hour-glass* structure, there are compute savings during training, but during generation, the architecture must maintain a cache for all the past tokens, leading to significant memory accesses. Concurrently, Hwang et al. (2025) explored a dynamic and end-to-end learned strategy for chunking in *hour-glass* like architectures.

Different from above, works like Ho et al. (2024); Yu et al. (2023) break up the modeling of a sequence into chunks/patches, where each chunk is modeled independently of each other given the previous "global" chunk embedding. An embedder first compresses each chunk independently, then these "local" chunk embeddings are passed to a "global" model where each "local" chunk embedding attends to past "local" chunk embeddings, forming a "global" chunk embedding. Each "global" chunk embedding is then passed to a decoder that is responsible for generating the next chunk.

On first glance, CATs might appear similar to above works, specifically Ho et al. (2024); Yu et al. (2023), however the subtle but salient difference is: one directly feeds **all the previous "local" chunk/patch representations** directly to the decoder in CAT, whereas in works like Ho et al. (2024), one feeds in just the previous "global" chunk representation outputted by a "global" model to the decoder. This architectural choice of passing *all* the compressed local chunks from the past directly to the decoder allows CATs to solve long-range recall tasks with ease while maintaining efficiency, whereas Ho et al. (2024) is plagued by *learnability* problems (even in toy recall tasks) due to constant size compression of history. Additionally, CATs don't utilize three different components (embedder, global decoder, local decoder) – it only uses a compressor and a decoder, reducing the design space and (significant) parameter requirements further.

Additionally, Yen (2024) extend the cache by using a modified encoder-decoder architecture, where decoder attends directly to final activations of a smaller frozen encoder, without any compression.

Finally, Barrault et al. (2024) suggest learning "concepts" instead of tokens by modeling the latent representation of language produced by pushing the token sequence through a large sentence embedder. The focus of this work is to decouple the modeling of the low-level details in each language, like tense and grammar, from the larger concept space that is shared across languages. In contrast, the goal with CAT is to reduce the cost of modeling sequences and can be used as a plug-and-play replacement to the latent concept model. Moreover, the encoder in Barrault et al. (2024) is an auto-encoder, that might keep irrelevant information in the chunk representation. Compressor in CATs only keeps information that is predictive of the future chunks.

**Adaptive architectures:** Kusupati et al. (2022); Devvrit et al. (2023) learns representations during training time that can work at different granularity during test-time, yielding adaptivity to the learned architecture. However, Devvrit et al. (2023) applies the *Matryoshka* technique on the feed-forward layer only, and not on the attention weights, that yields compute savings, but not memory savings. That being said, one could apply similar approaches to CATs making them complimentary. CATs use the same high-level approach described in Beyer et al. (2023): learn a single model that can work for various patch sizes at once depending on the downstream use-case at test-time. However, Beyer et al. (2023) worked with image tasks; CATs deal with language modeling and generation.