Efficient Long-Context Inference via H2O KV Cache Compression on Pythia-2.8B

Haiyang Duan
Shanghai Jiao Tong University
oceanwave@sjtu.edu.cn

Abstract

Large Language Models (LLMs) suffer from high memory consumption during inference due to the linear growth of Key-Value (KV) cache. In this paper, we reproduce and evaluate the **H2O (Heavy Hitter Oracle)** algorithm on the **Pythia-2.8B** model. H2O leverages the observation that a small set of "Heavy Hitter" tokens contributes disproportionately to attention scores. Our experiments on Wikitext-2 demonstrate that H2O can reduce KV cache memory by **95%** (budget of 128 tokens) while significantly outperforming local window attention (PPL 14.24 vs. 40.25). Furthermore, our sensitivity analysis reveals a critical "safety margin" for local context, showing that while Heavy Hitters are crucial, a minimum local window size is necessary to prevent performance collapse.

1 Introduction

The deployment of Large Language Models (LLMs) is constrained by memory bandwidth and capacity. While model parameters are static, the KV cache grows linearly with sequence length, becoming the primary bottleneck for long-context inference [2]. Recent works such as H2O [1] have proposed eviction-based strategies to compress the KV cache by exploiting the sparsity of attention matrices.

In this work, we focus on the reproduction and evaluation of H2O. Unlike previous studies that often utilize smaller scale or different architectures, we scale our experiments to **Pythia-2.8B** [3] to observe distinct attention patterns. We systematically compare H2O with Full Cache and Local Attention baselines across metrics including Perplexity (PPL), Effective Memory Usage, and Time Per Output Token (TPOT).

2 Methodology

H2O Algorithm. The H2O algorithm relies on the observation that attention matrices follow a power-law distribution. It maintains a fixed budget K of KV pairs based on a hybrid score:

$$S_t = \text{Recent}(t) + \text{HeavyHitter}(t) \tag{1}$$

The policy retains the most recent k_{recent} tokens to preserve local syntactic information, and the top-k_{heavy} tokens with the highest accumulated attention scores S to capture long-range dependencies.

Implementation. We implemented H2O on Pythia-2.8B using a training-free **monkey-patching** approach. We dynamically replaced the ‘forward‘ method of the ‘GPTNeoXAttention‘ module in the Hugging Face Transformers library. To accurately measure memory savings, we monitored the *Effective KV Size*, ensuring that evicted tensors are physically released.

3 Experiments

3.1 Setup

We evaluate on **Pythia-2.8B** (FP16) using the **Wikitext-2** (Test Split) dataset. We compare three methods: 1) **Full Cache** (Upper Bound); 2) **Local Attention** (Retains only recent tokens); and 3) **H2O** (Hybrid policy). We measure PPL, memory usage (GB), and throughput.

Note that we evaluate Perplexity (PPL) in token-by-token generation mode, rather than standard parallel teacher-forcing. This is strictly necessary to enforce the dynamic KV cache eviction policy at every step, ensuring the measured PPL reflects the actual inference quality of the compressed model.

3.2 Main Results

Table 1 presents the quantitative comparison. H2O significantly outperforms Local Attention. At a strict budget of 128, H2O achieves a **PPL of 14.24**, far superior to Local Attention (40.25).

Table 1: Performance comparison on Pythia-2.8B. H2O achieves significantly lower PPL than Local Attention under the same memory budget. (r=recent, h=heavy hitters)

Method	Budget	Eff. KV Size (GB)	TPOT (ms)	Throughput (tok/s)	PPL (↓)
Full Cache	∞	1.44	83.90	11.92	**8.21**
Local Attention	128	0.04	**62.38**	**16.03**	40.25
H2O (64r + 64h)	128	0.04	82.80	12.08	14.24
Local Attention	256	0.08	62.72	15.94	30.33
H2O (128r + 128h)	256	0.08	72.25	13.84	11.97
H2O (64r + 192h)	256	0.08	83.64	11.96	**11.88**
H2O (256r + 256h)	512	0.16	69.70	14.35	10.08
H2O (64r + 448h)	512	0.16	74.08	13.50	**9.41**

Performance Analysis. At budget 256, H2O (64r+192h) achieves a PPL of 11.88, effectively recovering the model's capability compared to the Local baseline (30.33). As the budget increases to 512, the best H2O configuration (64r+448h) reaches a PPL of **9.41**, approaching the Full Cache baseline (8.21) while using only $\sim$11% of the memory.

3.3 Sensitivity Analysis: The Local Context Safety Margin

To understand the optimal balance between local context (k_{recent}) and global anchors (k_{heavy}), we conducted a fine-grained ratio sweep at Budget 256 and 512.

Configuration (Total=256)	Ratio (h/K)	PPL (↓)	TPOT (ms)
Local ($256r + 0h$)	0%	30.33	**62.72**
H2O ($128r + 128h$)	50%	11.97	72.25
H2O ($64r + 192h$)	75%	**11.88**	83.64
H2O ($32r + 224h$)	87.5%	13.12	80.61

Figure 1: **Sensitivity Analysis at Budget 256.** While increasing Heavy Hitters helps, reducing local context below 64 tokens (e.g., 32r+224h) causes performance regression.

The "Sweet Spot" of Local Context. As shown in Figure 1, increasing the Heavy Hitter ratio generally improves performance (11.97 $\rightarrow$ 11.88). However, we observe a distinct failure mode when the local window is compressed too aggressively. The configuration $32r + 224h$ suffers a regression to 13.12 PPL. This suggests a **"Safety Margin"** of approximately 64 tokens is required for Pythia-2.8B to maintain local syntactic coherence.

Scaling with Budget. This hypothesis is confirmed at Budget 512. Since the budget is ample, we can set $k_{recent} = 64$ (meeting the safety margin) and allocate the remaining huge budget to Heavy

Hitters ($448h$). This configuration achieves our best result of **9.41 PPL**, significantly outperforming the balanced split ($256r + 256h$, 10.08 PPL).

3.4 Overhead Analysis: Where does the time go?

While H2O significantly reduces memory usage (>90%) and maintains generation quality, our Python-based implementation observes a TPOT regression compared to the optimized Local Attention. To understand this trade-off, we profiled the inference latency per step. The breakdown is presented in Table 2.

Table 2: **Latency Breakdown per Generation Step (ms).** The theoretical speedup from reduced Attention Computation (0.72ms $\rightarrow$ 0.10ms) is overshadowed by the overhead of Python-level score calculation and eviction logic.

Component	Baseline	Local (256)	H2O (256)	Impact
Prep (Rotary Emb)	1.38	1.50	1.42	$\sim$
Attn Compute	**0.72**	**0.12**	**0.10**	**-86% (Gain)**
Score Calc	-	0.05	0.41	
Evict (TopK)	-	0.00	0.22	**+0.75 (Overhead)**
Gather/Update	-	0.00	0.12	
Total Latency	**2.10**	**1.67**	**2.27**	**Slower**

The Cost of Interpretability. As evidenced in Table 2, H2O successfully reduces the core Attention Computation time by approximately 86% (from 0.72ms to 0.10ms), confirming the algorithmic efficiency of the fixed-budget mechanism. However, this gain is completely negated by the **0.75ms overhead** introduced by the dynamic importance score calculation and ‘torch.topk‘ eviction policy.

Conclusion on Efficiency. The current bottleneck is not the algorithm itself but the implementation stack. The Local Attention baseline (1.67ms) represents the theoretical upper bound of speed for a budget of 256. To bridge the gap between H2O (2.27ms) and Local (1.67ms), future work must involve fusing the *Score-Evict-Update* pipeline into a custom CUDA kernel, which would eliminate the Python interpreter overhead and memory movement costs.

3.5 Visualization

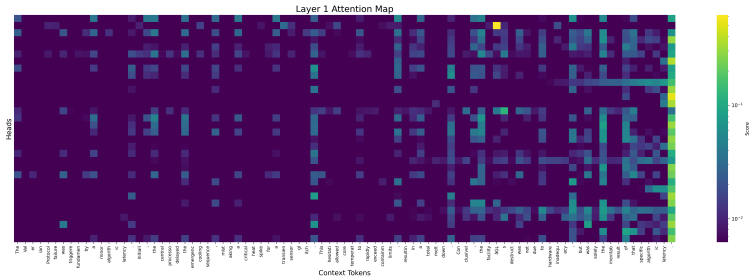
To verify the theoretical assumptions of H2O, we visualized the attention matrices of selected layers. Since attention scores often follow a power-law distribution where a few tokens dominate, we applied a **logarithmic normalization** to the heatmaps (Figure 2) to reveal patterns that might be invisible under a linear scale.

The visualization reveals two critical distinct patterns:

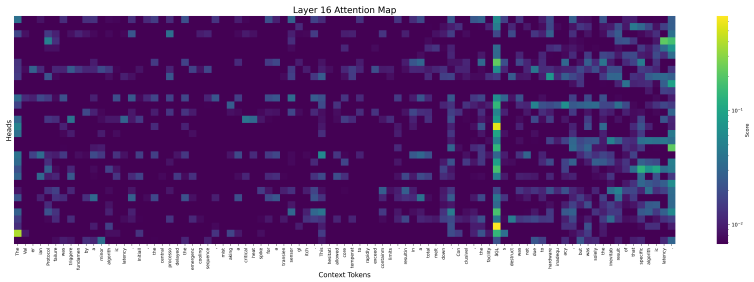
1. **Heavy Hitters (Vertical Stripes):** As seen in Layer 16 and 30, specific tokens (such as punctuation) form bright vertical lines, indicating they are attended to by almost all future tokens. This validates the H2O strategy of retaining tokens based on accumulated scores [1].
2. **Attention Sinks (First Column):** The logarithmic scale unveils that the very first token (index 0) consistently receives a stable amount of attention across deep layers, even when it is semantically irrelevant. This visual evidence supports the "Attention Sink" theory [2], explaining why our implementation (and H2O) implicitly benefits from retaining the start-of-sequence token as a high-scoring anchor.

4 Conclusion

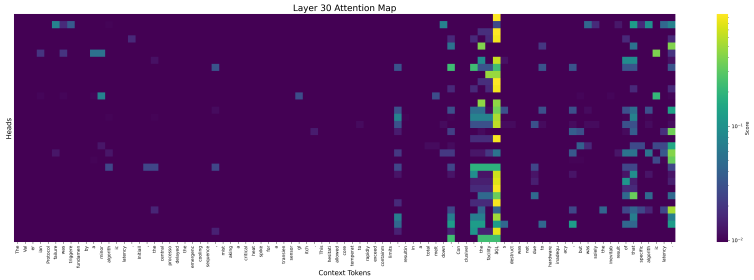
In this work, we reproduced and evaluated the H2O KV cache compression algorithm on Pythia-2.8B. Our experiments confirm that a small budget of tokens, composed of recent context and global "Heavy Hitters," can sustain generation quality comparable to a full cache. Specifically, at a budget of 512 tokens (only $\sim$11% of the full context), H2O achieves a perplexity of **9.41**, closely matching the dense baseline of 8.21.



(a) Layer 1 heatmap



(b) Layer 16 heatmap



(c) Layer 30 heatmap

Figure 2: **Evolution of Attention Patterns across layers (Log-scale Visualization).** We employ a logarithmic color scale to account for the heavy-tailed distribution of attention scores. (a) Shallow layers show diffuse attention. (b) Middle layers exhibit clear vertical stripes corresponding to **"Heavy Hitters"** (e.g., punctuation marks). (c) Deep layers are highly sparse. Note the consistent activation on the first column across heads, visually confirming the **"Attention Sink"** hypothesis [2].

Our study contributes two critical insights. First, we identified a **"Local Safety Margin"**: while allocating budget to Heavy Hitters improves long-range retrieval, a minimum local window of approximately 64 tokens is indispensable. Second, our latency analysis reveals that while H2O reduces Attention Computation by over 80%, Python-level overhead masks these gains. Future work must focus on optimized CUDA kernels to translate memory savings into wall-clock speedups.

References

[1] Zhang, Z., et al. (2023). H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. *NeurIPS 2023*.

[2] Xiao, G., et al. (2023). StreamingLLM: Efficient Streaming Language Models with Attention Sinks. *ICLR 2024*.

[3] Biderman, S., et al. (2023). Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling. *ICML 2023*.

A Appendix

A.1 Code Availability

To facilitate reproducibility and further research, we have open-sourced our full codebase, including the H2O implementation, benchmarking scripts, and analysis tools. The repository is available at:

`https://github.com/Ocean-Duan/H2O-Pythia-Reproduction`

A.2 Implementation Details

Our implementation diverges from modifying the Transformers library source code directly. Instead, we employed a **Dynamic Monkey-Patching** technique to inject the H2O logic at runtime.

Core Mechanism. The core logic is encapsulated in a custom forward function, 'h2o-gpt-neox-attention-forward'. We use Python's 'types.MethodType' to bind this function to each 'GPTNeoXLayer' instance.

KV Cache Compression Logic. Inside the patched forward function, we intercept the Key and Value tensors during the generation phase (when 'query-length == 1'). The process involves:

1. **Score Accumulation:** We maintain a persistent 'h2o-scores' buffer.
2. **Greedy Selection:** We combine the most recent window (Indices: $T - k_{recent} \dots T$) and the top-k_{heavy} indices from historical scores.
3. **Tensor Reduction:** We use 'torch.gather' to physically reduce tensors to the budget size.