From TLinFormer to TConstFormer: The Leap to Constant-Time Transformer Attention

Achieving $\mathcal{O}(1)$ Computation and $\mathcal{O}(1)$ KV Cache during Autoregressive Inference

Zhongpan Tang
tangzhongp@gmail.com

August 29, 2025

Abstract

Although the Transformer has become the cornerstone of modern AI, its autoregressive inference suffers from a linearly growing KV Cache and a computational complexity of $\mathcal{O}(N^2d)$, severely hindering its ability to process ultra-long sequences. To overcome this limitation, this paper introduces the TConstFormer architecture, building upon our previous work, TLinFormer. TConstFormer employs an innovative periodic state update mechanism to achieve a truly constant-size $\mathcal{O}(1)$ KV Cache. The computational complexity of this mechanism is also $\mathcal{O}(1)$ in an amortized sense: it performs purely constant-time computations for $k-1$ consecutive steps (e.g., $k = 256$) and executes a single linear-time global information synchronization only on the k-th step. Theoretical calculations and experimental results demonstrate that TConstFormer exhibits an overwhelming advantage over baseline models in terms of speed, memory efficiency, and overall performance on long-text inference tasks. This breakthrough paves the way for efficient and robust streaming language model applications.

1 Introduction

The Transformer [5] architecture has indisputably become the cornerstone of modern artificial intelligence, driving immense progress from Large Language Models (LLMs) to multimodal applications. However, behind this monumental success lies a fundamental scalability bottleneck in its core self-attention mechanism during autoregressive inference. Its computational complexity is $\mathcal{O}(N^2d)$, and for each new token generated, the model must append its Key and Value vectors to a growing cache (the KV Cache) and attend to the entire cache to maintain contextual coherence. This mechanism causes the memory footprint of the KV Cache to grow linearly with the sequence length N ($\mathcal{O}(N)$), which not only consumes precious GPU memory but also fundamentally impedes the Transformer's ability to handle ultra-long or even infinite sequences—a core requirement for streaming applications like real-time dialogue, long-document summarization, and video stream analysis.

To mitigate this issue, the community has proposed various approximation methods, with sliding window attention [2] being one of the most representative. By retaining the KV Cache of only the most recent W tokens, this method does limit memory and computational costs to a constant range. However, this "history truncation" strategy comes at a high price: it can lead to "catastrophic forgetting," severely compromising the model's performance and robustness on tasks requiring long-range dependencies. Therefore, achieving truly efficient long-sequence inference without sacrificing global information remains a critical, unsolved challenge.

In our prior work [4], we took a significant step toward addressing this challenge by returning to the first principles of connectionism, proposing TLinFormer, a linear attention architecture.

Building on this foundation, this paper introduces the TConstFormer architecture, aiming to completely overcome the streaming inference problem of Transformers. The core design of TConstFormer is an innovative periodic state update mechanism that completely decouples the Transformer's inference state from the growing sequence length. Specifically, we achieve two key breakthroughs:

1. **True $\mathcal{O}(1)$ Memory Footprint**: Across all time steps, TConstFormer's KV Cache is maintained at a strictly constant size, theoretically equipping it with the ability to process ultra-long data streams.

2. **Amortized $\mathcal{O}(1)$ Computational Complexity**: The model performs purely constant-time operations for $k-1$ consecutive steps and executes a global information synchronization at a linear cost only on the k-th step (e.g., $k = 256$). This makes the average single-step computational cost constant, ensuring sustained high throughput.

This periodic global synchronization design not only guarantees computational efficiency but, more importantly, plays the role of "memory consolidation," enabling the model to perform streaming inference without forgetting distant history, thus addressing the fundamental flaw of the sliding window mechanism.

The main contribution of this paper is the proposal of TConstFormer, a new Transformer architecture designed for efficient and robust streaming inference. We demonstrate through experiments that in long-sequence inference tasks, TConstFormer significantly outperforms existing baseline models in both inference latency and memory usage. This work paves the way for building next-generation language models capable of handling unbounded contexts.

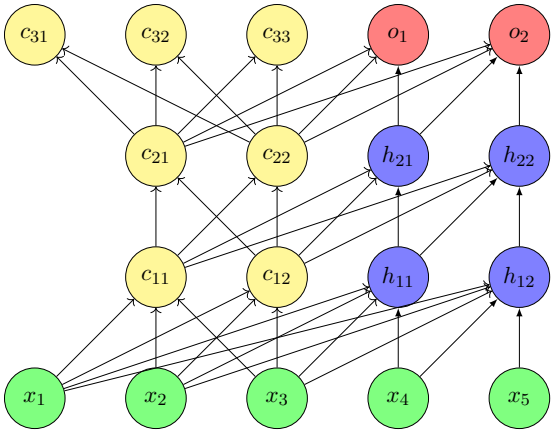
2 Revisiting TLinFormer from a Connectionist Perspective

2.1 The Root of Linear KV Cache Growth in TLinFormer

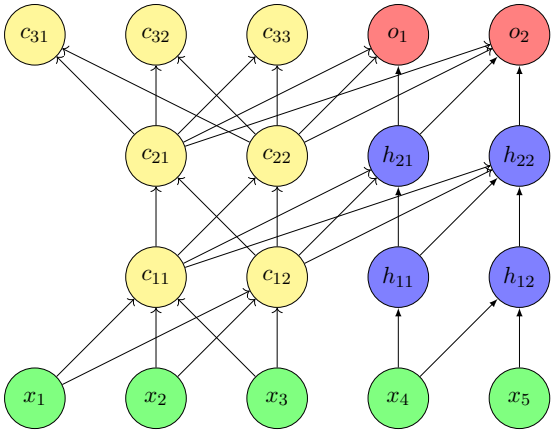
The design philosophy of our previous work, TLinFormer, was to return to the first principles of connectionism, preserving the integrity of information flow to the greatest extent possible while maintaining the causality of the Transformer. To this end, we only removed the connections in standard self-attention that do not conform to the law of causality. This design allowed TLinFormer to achieve a "Full Context-Aware" state, where information flow could reach the entire historical context, while significantly reducing computational complexity and KV cache consumption. However, this also meant it inherited a fundamental constraint related to sequence length: to maintain awareness of the entire history, the size of its KV Cache still inevitably grew linearly with the sequence length N ($\mathcal{O}(N)$).

Although TLinFormer has shown significant efficiency advantages over baseline models on sequences of finite length, the $\mathcal{O}(N)$ memory bottleneck prevents it from being suitable for the ideal scenario of processing longer data streams. As the sequence continues to expand, its memory consumption will eventually exceed the physical limits of the hardware. Therefore, we must make a trade-off between "information integrity" and "ultimate efficiency."

The core breakthrough of TConstFormer stems from a rethinking of this trade-off. We identified the key connections in TLinFormer responsible for long-term dependencies and selectively severed them. Specifically, as shown in Figure 1a, we interrupted the direct information pathways from historical inputs (e.g., x_1 to x_3) to the current computational units (e.g., h_{11}, h_{12}). This structural modification makes the model's inference state no longer dependent on the entire growing historical sequence, but on a fixed-size hidden state, thereby fundamentally solving the problem of the KV Cache growing linearly with sequence length.



(a) Information flow structure of TLinFormer.

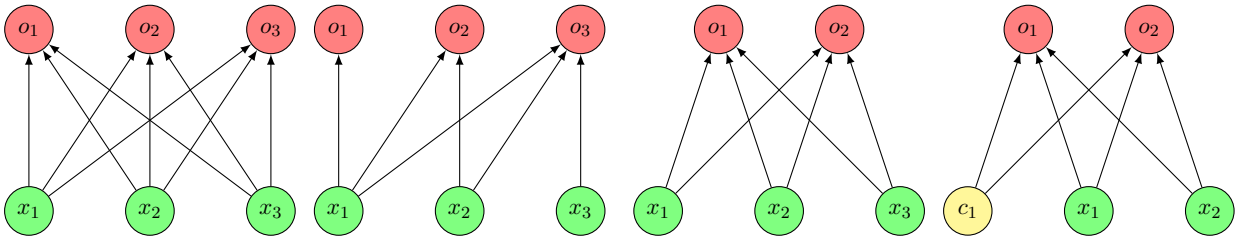


(b) Information flow structure of TConstFormer.

Figure 1: By removing the connections between x_1, x_2, x_3 and h_{11}, h_{12} in TLinFormer, we obtain the TConstFormer architecture.

3 TConstFormer Architecture

To implement the connection structure shown in Figure 1b, we still use the same attention components as in TLinFormer, as illustrated in Figure 2:



(a) Self Attention (b) Causal Attention (c) Focused Attention (d) Cross Attention

Figure 2: Connection diagrams for the 4 types of attention mechanisms required in this paper.

All the types above are special cases of the attention mechanism, calculated by the formula:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\boldsymbol{T}}}{\sqrt{d_k}}\right)\boldsymbol{V}$$

We no longer view it as the traditional "query-key-value" interaction, but rather examine the attention mechanism from the perspective of dimensionality transformation. Assuming Q has dimensions (B, L_Q, D), and K and V have dimensions (B, L_K, D), the result of attention has dimensions (B, L_Q, D). We can see this as the attention mechanism scaling the L dimension of K and V, i.e., scaling from the original L_K to L_Q. From an information flow perspective, attention completes a thorough fusion of the information in Q, K, and V. It is this picture that allows us to understand attention computation as a type of fully connected layer acting on the L dimension (like an MLP for the L dimension). Based on this understanding, we can precisely construct the information flow required by Figure 1b by designing and combining different patterns of attention.

A TConstFormer block operates on an input partitioned into a historical context window X_{hist} and a generation window X_{gen}. Its topological connections are constructed layer by layer as follows:

1. **Context Path Encoding**: The historical context X_{hist} is compressed in the L dimension in the first layer using the attention shown in Figure 2c. Subsequent intermediate layers are processed by self-attention layers. The final layer restores the L dimension using the attention shown in Figure 2d (of course, if stacking multiple TConstFormer blocks is not considered, the computation of the final layer can be omitted).

2. **Generation Path Computation**: At each layer i, the computation involves two information flows:

   - **Internal Cohesion (Causal Self-Attention)**: A causal self-attention mechanism is applied to the generation window representation from the previous layer (H_{i-1}, where $H_0 = X_{gen}$). This allows tokens within the generation window to interact with each other while adhering to causal constraints.
   - **Context Integration (Cross-Attention)**: Historical information is fused into the generation window using a cross-attention mechanism. The queries come from the generation path (H_{i-1}), while the keys and values come from (C_{i-1}).
   - The results of these two attention mechanisms are combined and passed through a feed-forward network (FFN) to produce the output of the current layer H_i.

A TConstFormer block can function as a standalone module or be stacked repeatedly to form a deep network. When used alone, the attention in the final layer of the historical window in Figure 1b (C_3) can be omitted. When multiple blocks are stacked, as shown in Figure 3, the output of each layer serves as the input for the next, thereby constructing a standard deep Transformer decoder structure.

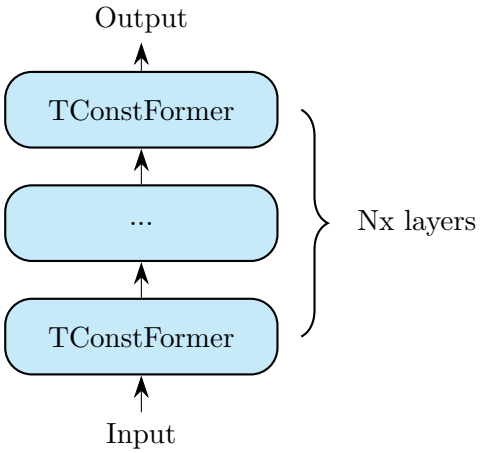


Figure 3: Schematic of a stacked TConstFormer network structure.

4 Attention Complexity Analysis

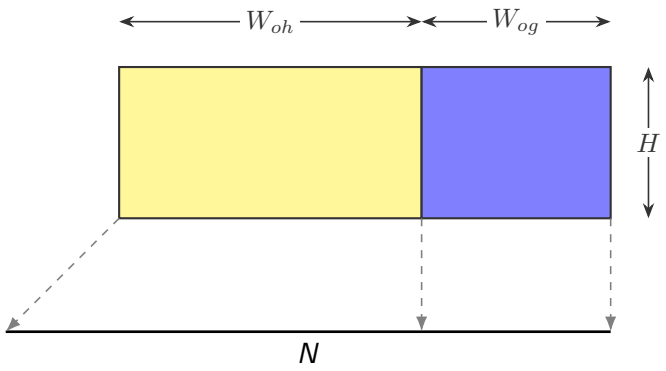


Figure 4: Windowed computation schematic.

Let the total input sequence length be N, the feature dimension be D, and the number of intermediate self-attention layers within a TConstFormer block be H. The model is partitioned into a window of length W_{oh} for processing historical context, which can observe a historical sequence of length $N - W_{og}$, and a generation window of length W_{og}.

A detailed derivation of the following discussion can be found in Appendix A.

4.1 Cache Miss

A Cache Miss refers to an event where pre-computed results cannot be reused, requiring a full computation from scratch. It primarily occurs in two scenarios:

1. **Training Phase**: Since the historical context is different for each training batch, the caching mechanism is not applicable, and every forward pass can be considered a Cache Miss.
2. **Inference Phase**:
   - When generating the initial token for a given context.
   - At the moment of generating the first new token after the historical context window is updated (i.e., slid).

The computational cost of a cache miss is equivalent to performing one full forward computation with the cache disabled.

The total computational cost is the sum of the costs for the context window and the generation window. Crucially, the total cost is a precise linear function of the total sequence length, of the form:

$$\text{Total Cost} = C_1 \cdot N + C_0 \tag{1}$$

where the slope and intercept are constants determined by the model's hyperparameters:

$$C_1 = D \cdot (2W_{oh}) \tag{2}$$

$$C_0 = D\left[H(W_{oh}^2 + W_{og}^2 + W_{og}W_{oh}) + 2W_{og}^2 - W_{og}W_{oh}\right] \tag{3}$$

$$\text{Total Cost} = D\left[N(2W_{oh}) + H(W_{oh}^2 + W_{og}^2 + W_{og}W_{oh}) + 2W_{og}^2 - W_{og}W_{oh}\right] \tag{4}$$

Since D, W_{oh}, W_{og}, and H are fixed or bounded after training, **the computational complexity of TConstFormer is strictly linear with respect to the sequence length** N.

4.2 Cache Hit

A Cache Hit occurs only during autoregressive inference. It refers to the event of generating any subsequent token other than the first one within a single generation cycle (i.e., when the historical context window remains static).

The total computational cost is a constant quantity:

$$\text{Total Cost} = (H+1)DW_{oh} + (H+2)DW_{og}^2 \tag{5}$$

4.3 Complexity Summary

The computational complexity of TConstFormer exhibits a dual-mode characteristic closely tied to the cache state, which is the core of its efficient inference capability.

- **On a Cache Miss**, such as during training or when generating the initial token, the model's computational cost is **strictly linear** with the total sequence length N, with a complexity of $\mathcal{O}(N)$. This constitutes the upper bound of the model's single computation overhead.
- **On a Cache Hit**, i.e., when generating subsequent tokens in autoregressive inference, the computational cost is **completely independent** of the total sequence length N, becoming a **constant** determined only by the window sizes (W_{oh}, W_{og}) and model depth (H), with a complexity of $\mathcal{O}(1)$.

This dynamic transition from linear to constant cost allows TConstFormer to maintain the overhead of the vast majority of generation steps at an extremely low level when processing long sequences, thereby achieving orders-of-magnitude inference acceleration.

5 Model Properties and Discussion

5.1 Training Process

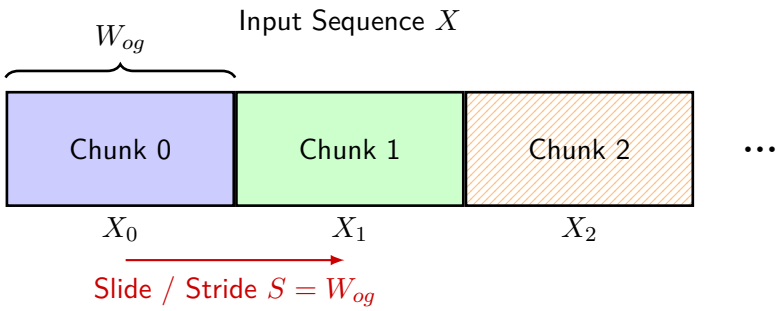


Figure 5: Sliding window information processing flow during training.

The model processes long sequences in chunks. During training, it employs a sliding window mechanism. First, it processes the interval $[0, W_{og}]$, where the historical context is empty. Then, the window slides by a distance of W_{og}, and the model processes the interval $[W_{og}, 2W_{og}]$, using the first chunk $[0, W_{og}]$ as its historical context. The window then slides again by W_{og}, and the model processes the interval $[2W_{og}, 3W_{og}]$, using the first two chunks $[0, 2W_{og}]$ as its historical context. This process continues until the entire sequence has been processed. The outputs of each generation window are concatenated to form the final output sequence for loss calculation.

5.2 Excellent Cache Efficiency during Inference

5.2.1 $\mathcal{O}(1)$ KV Cache Footprint

The KV cache of a standard Transformer scales linearly with the entire sequence length L, as shown in Equation (6), which often becomes a memory bottleneck.

$$M_{\text{transformer}} = 2 \cdot B \cdot L \cdot d_{\text{model}} \cdot P_{\text{bytes}} \cdot N_{\text{layers}} \tag{6}$$

Here, N_{layers} is the number of layers, B is the batch size, d_{model} is the hidden dimension, and P_{bytes} is the precision in bytes.

TConstFormer offers a significant advantage in autoregressive inference. The key-value (KV) cache associated with the historical context window (W_{oh}) remains static as long as the context itself does not change. A cache invalidation and re-computation event only occurs after W_{og} new tokens have been generated and the context window needs to be updated.

The cache overhead of TConstFormer comes from the KV Cache consumption within the historical and generation windows. The final $M_{\text{TConstFormer}}$ is:

$$M_{\text{TConstFormer}} = 2 \cdot B \cdot (H+1) \cdot W_{oh} \cdot d_{\text{model}} + 2 \cdot B \cdot (H+2) \cdot W_{og} \cdot d_{\text{model}} \tag{7}$$

The KV Cache footprint of TConstFormer is **constant and completely decoupled from the sequence length**, achieving a true $\mathcal{O}(1)$ space complexity.

5.2.2 Superior Time Acceleration

Standard autoregressive Transformers (Decoder-only) face a fundamental efficiency bottleneck during inference. In contrast, as discussed in Section 4, when a cache hit occurs, TConstFormer's inference time is constant.

6 Experiments

In this section, we will validate the effectiveness of TConstFormer through a series of experiments. We begin by detailing the experimental setup, including the baselines, model configurations, and evaluation metrics, to ensure fairness and reproducibility. Subsequently, we will present and thoroughly analyze the main results on the wikitext-103-v1 benchmark.

6.1 A Note on Long-Context Retrieval Tasks

The "Needle in a Haystack" benchmark is widely used to evaluate the long-context retrieval capabilities of Large Language Models (LLMs). However, this test primarily measures the complex instruction-following and long-range dependency abilities that emerge after pre-training on massive, diverse corpora. The TConstFormer architecture proposed in this paper focuses its core contribution on demonstrating a fundamental improvement in computational and memory efficiency. Due to the limitations of our model scale (41M parameters) and training data (Wikipedia), its design objective is not to replicate the full range of emergent abilities of large-scale LLMs. Therefore, we consider the "Needle in a Haystack" test to be orthogonal to the core goal of validating architectural efficiency that this paper aims to address, and thus have not included it in our primary evaluation scope. Extending the TConstFormer architecture to larger-scale models to explore its potential in complex retrieval tasks is a promising direction for future research.

6.2 Implementation Details

Hardware Environment: All our training, inference, and testing were conducted on a platform more aligned with consumer-grade hardware. The platform is configured as follows: one **NVIDIA GeForce RTX 4090 GPU** (24 GB VRAM), one **AMD EPYC 7543 CPU**, and 62 GB of system memory.

Software Stack: The software stack was consistent across environments: **Python 3.12.11**, **PyTorch 2.7.1**, **CUDA 12.6**, **cuDNN 9.5.1**, **Hugging Face Transformers (v4.55.2)**, and the operating system was **Ubuntu 22.04.5 LTS**.

6.2.1 Principle of Fair Comparison and Model Configuration

To ensure a fair and meaningful comparison between TConstFormer and the standard Transformer baseline, all experiments adhere to the principle of **parameter parity**. The core innovation of TConstFormer lies in the **reorganization** of information flow, rather than the introduction of new parameterized components. It is essentially a **topological reconstruction** of standard Transformer modules. Therefore, as long as the total computational depth of a stacked TConstFormer model matches the number of layers in a standard Transformer model, their **total parameter counts are identical**. This allows us to attribute any performance differences solely to the superiority of the architectural design.

In these experiments, we use a small-scale model configuration with approximately **41M** parameters. Both the baseline model and our TConstFormer use the same core hyperparameters:

- `vocab_size`: 50257 (same as GPT-2)
- `n_embd` (embedding dimension): 432
- `n_head` (number of attention heads): 12
- `n_transformer_block` (equivalent total depth): 8

For the baseline model, this is a standard 8-layer decoder-only Transformer. For TConstFormer, this equivalent depth of 8 is achieved by stacking 2 TConstFormer blocks, with each block having an internal depth hyperparameter of $H = 2$.

Training parameters, such as learning rate, were kept identical for all models. The equivalent batch size was set to 256 (achieved through gradient accumulation).

A Note on Hyperparameter Selection: The choice of core hyperparameters, such as the internal depth $H = 2$, was primarily guided by a pragmatic balance between computational overhead and the model's effective receptive field under limited computing resources. We believe that a comprehensive hyperparameter search tailored to different model scales is an essential step for future work, but it falls beyond the scope of this initial validation study.

6.2.2 Dataset and Evaluation Metrics

We use the wikitext-103-v1 dataset from the Hugging Face repository (Salesforce/wikitext) for all experiments. This dataset contains approximately 120 million tokens. Model performance is evaluated using Perplexity (PPL) on the validation set, where lower values indicate better performance.

6.2.3 Baselines and Model Variants

We compare TConstFormer against a standard decoder-only Transformer baseline and TLinFormer. To evaluate performance under different configurations, we trained multiple variants for each architecture. The naming convention for these variants is explained below:

`Base XXX`: Represents the standard Transformer baseline. The suffix `XXX` denotes the sequence length used during its training. For example, `Base 1K` refers to the baseline model trained with a 1K sequence length.

`TConstFormer/TLinFormer XXX-YYY-ZZZ`: Represents our TConstFormer/TLinFormer models, with the name composed of three parameters:

- `XXX`: The total sequence length used during training.
- `YYY`: The total length of the core observation window, i.e., $W_{total} = W_{oh} + W_{og}$.
- `ZZZ`: The ratio of the historical context observation window to the total observation window, i.e., W_{oh}/W_{total}.

For example, `TConstFormer 2K-512-0.5` represents a TConstFormer model trained with a 2K sequence length, a total observation window of 512, where the historical context window is set to half the total window size ($0.5 \times 512 = 256$).

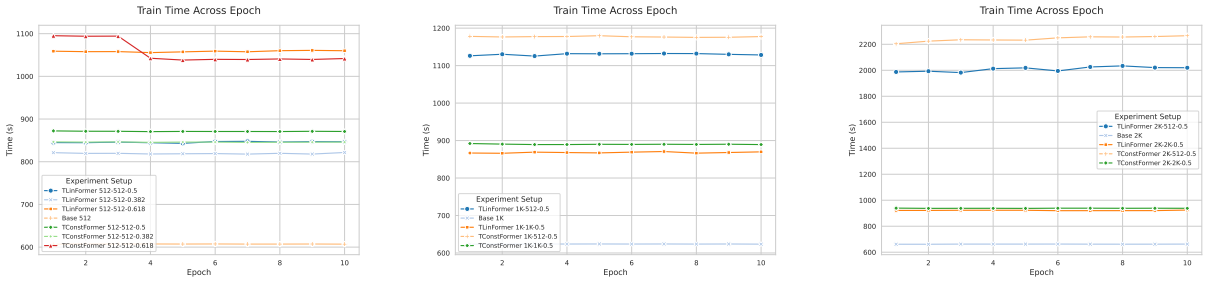
6.3 Training Results and Analysis

6.3.1 Analysis of Training Overhead and Trade-offs

We further evaluated the training efficiency of each architecture, as shown in Figure 6. It is important to note that to maximize hardware utilization across different sequence lengths, we adjusted the actual batch size and used gradient accumulation to ensure an equivalent batch size of 256 for all experiments. Therefore, a direct comparison of wall-clock time across different sequence lengths is not meaningful; we primarily focus on the relative efficiency at the same sequence length.

The results show that, under the same sequence length configuration, our new architectures exhibit a certain increase in training overhead compared to the baseline. This phenomenon is entirely consistent with our architectural design. For the `1K` sequence length, for instance, the baseline model processes the full 1K context in a single parallel pass. In contrast, our models (e.g., `TConstFormer 1K-1K-0.5`) use a chunked processing mechanism, for example, processing the first 512 tokens, then the next 512, and finally combining the results for loss calculation. While this chunked computation introduces additional scheduling overhead, it is precisely this design that forms the foundation for the model's efficient caching and significant performance gains during inference.

Quantitative Analysis of Training Overhead: To quantify this overhead more concretely, we analyzed the training time per epoch at a 1K sequence length. The baseline model, `Base 1K`, completed an epoch in approximately **620 seconds**, while our `TConstFormer 1K-1K-0.5` model took around **890 seconds**, representing an overhead of approximately **42%**. We argue that this **controllable, one-time training cost** is a highly valuable and pragmatic engineering trade-off for achieving **orders-of-magnitude acceleration across countless inference tasks** throughout the model's lifecycle.



(a) Training time per epoch for sequence length 512

(b) Training time per epoch for sequence length 1K

(c) Training time per epoch for sequence length 2K

Figure 6: **Training efficiency comparison at different sequence lengths.** The plots show the wall-clock time required for each model to complete a single epoch at training sequence lengths of 512, 1K, and 2K.

6.3.2 Validation Set PPL

Table 1 and Figure 7 show the perplexity of all model variants on the wikitext-103-v1 validation set as training progresses. We can observe several key findings from the results:

1. **Architectural reconstruction does not sacrifice base performance.** First, we verified that our proposed architectural reconstruction does not introduce significant performance loss under equivalent configurations. As shown in Table 1, when the observation window length is identical to the baseline model's context length (e.g., comparing `Base 1K` with `TLinFormer 1K-1K-0.5` and `TConstFormer 1K-1K-0.5`), the final perplexities (PPL) are almost identical (22.5, 22.7, and 22.7, respectively). This strongly demonstrates that our proposed information flow reorganization is achieved without compromising the model's fundamental expressive power.

2. **TConstFormer shows a performance advantage in equivalent configurations.** A key finding is that under identical training length and observation window configurations, TConstFormer consistently matches or outperforms TLinFormer, and generally reaches the performance of the baseline model. The most notable example is in the `512` context, where `TConstFormer 512-512-0.5` achieves a final PPL of **21.6**, matching `Base 512` and outperforming all TLinFormer variants (21.9). This indicates that TConstFormer's architectural optimization not only brings a leap in inference efficiency but also shows some advantage in model performance.

3. **Controllable performance trade-off from information compression.** We also investigated the effect of "forced compression" (i.e., using an observation window shorter than the total sequence length) on performance. The results show that both TLinFormer and TConstFormer exhibit a slight and expected performance gap compared to the full-size baseline when processing long sequences with short observation windows (e.g., comparing `Base 1K` vs. `1K-512-0.5`). We believe this performance trade-off is a direct manifestation of the model being forced to compress and abstract long historical information within a limited window, which is a key step towards higher efficiency and stronger generalization.

4. **High robustness to core hyperparameters.** Finally, we conducted an ablation study on the model's performance with different historical window ratios (W_{oh}/W_{total}). In the `512-512-X` configuration group, for both TLinFormer and TConstFormer, despite the hyperparameter changing from 0.382 to 0.618, the final PPL of all variants remained stable within a very small range. This proves that our architecture's performance advantage stems from its robust core design, rather than fine-tuning of specific hyperparameters, greatly enhancing its reliability and ease of use in practical applications.

Unexpected Performance Improvement from Architectural Simplification:

A noteworthy and counter-intuitive finding is that TConstFormer exhibited performance that matched or even surpassed TLinFormer in several configurations (see Table 1). We simplified TLinFormer architecturally by removing the direct connections between the generation window and the full historical sequence. Our initial intuition was that more information pathways should lead to greater expressive power. However, the experimental results suggest that this simplification turned out to be an advantage.

We speculate that this performance improvement stems from a stronger **Structured Inductive Bias** introduced by TConstFormer, which forces the network to achieve a clearer **Functional Specialization**:

- The **historical context window** is specifically shaped into an efficient **information encoding module**. Its sole responsibility is to distill the ever-growing sequence history into a bounded-size, high-information-density representation of the "world state."
- The **generation window**, in turn, focuses on its core task as a **language generation module**. It no longer needs to weigh between raw, unrefined historical details and a compressed summary, but can make decisions based on a more stable, abstract, and high-quality source of information.

We believe that this clear modular division of labor may simplify the model's optimization process and guide it to learn a more robust and generalizable internal representation.

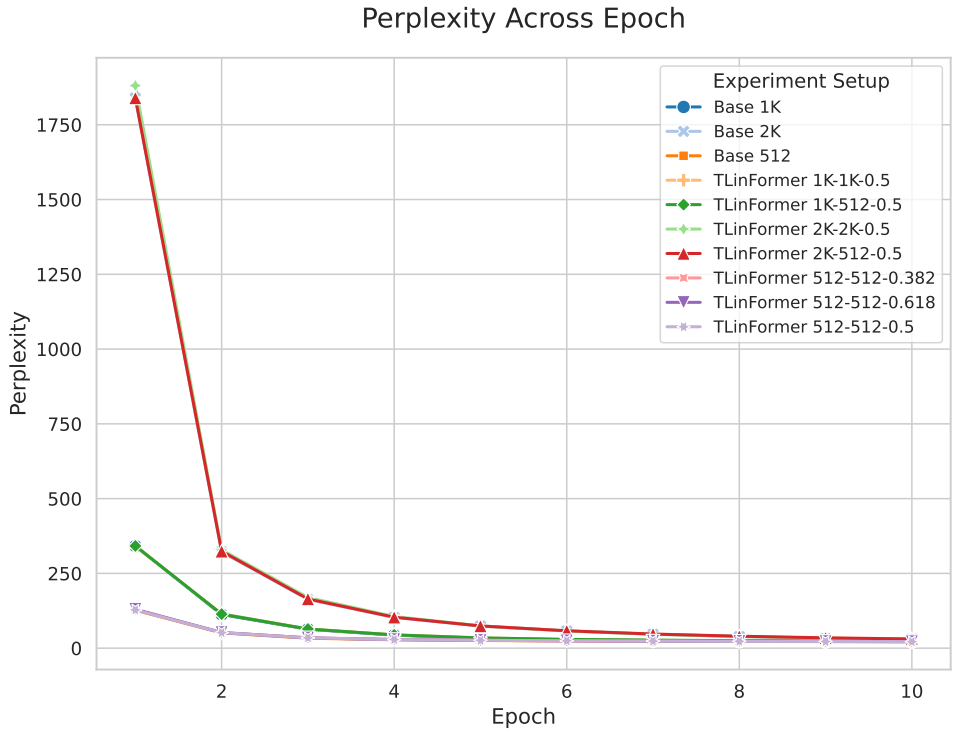


Figure 7: Perplexity (PPL) of each model over training epochs.

Table 1: Perplexity (PPL) on the wikitext-103-v1 validation set. Lower is better.

experiment	Epoch									
	1	2	3	4	5	6	7	8	9	10
Base 512	126.8	51.0	33.4	28.1	25.7	24.2	23.3	22.6	22.0	21.6
TLinFormer 512-512-0.382	129.1	51.9	34.4	28.7	26.2	24.6	23.6	22.8	22.4	21.9
TLinFormer 512-512-0.5	127.9	52.1	34.5	28.9	26.3	24.7	23.7	22.9	22.3	21.9
TLinFormer 512-512-0.618	130.1	52.4	34.4	28.8	26.2	24.8	23.6	23.0	22.4	21.9
TConstFormer 512-512-0.382	127.3	51.5	34.1	28.6	26.1	24.5	23.6	22.8	22.2	21.8
TConstFormer 512-512-0.5	124.3	51.0	33.7	28.2	25.7	24.1	23.2	22.5	22.0	21.6
TConstFormer 512-512-0.618	129.4	52.0	34.4	28.9	26.2	24.8	23.7	23.0	22.5	22.0
Base 1K	341.5	112.5	63.4	43.5	33.0	28.3	25.8	24.2	23.3	22.5
TLinFormer 1K-1K-0.5	340.7	115.0	64.3	44.6	34.2	29.0	26.3	24.6	23.5	22.7
TLinFormer 1K-512-0.5	341.8	114.0	64.0	44.5	34.3	29.4	26.8	25.0	23.8	23.0
TConstFormer 1K-1K-0.5	343.2	114.0	63.8	44.5	33.8	28.8	26.2	24.6	23.5	22.7
TConstFormer 1K-512-0.5	342.1	113.6	64.5	44.5	34.2	29.3	26.7	24.9	23.9	23.0
Base 2K	1839.4	321.8	162.9	102.9	73.9	57.5	46.6	38.8	32.7	29.5
TLinFormer 2K-2K-0.5	1881.0	328.5	169.4	106.0	74.6	57.7	46.8	38.9	33.7	29.8
TLinFormer 2K-512-0.5	1839.7	324.0	164.6	103.5	74.3	58.2	47.2	40.1	34.5	30.9
TConstFormer 2K-2K-0.5	1945.7	327.1	167.7	104.3	74.4	57.2	46.4	38.8	33.3	29.6
TConstFormer 2K-512-0.5	1852.0	326.5	165.7	103.9	74.5	58.4	46.9	39.4	33.9	30.3

6.4 Inference Results and Analysis

6.4.1 Testing Methodology

1. **Precondition:** Caching is always enabled.

2. **Environment Initialization:** Before each test run, we clear the GPU memory by calling `torch.cuda.empty_cache()` to ensure each test starts from a clean, consistent initial state, eliminating interference from caching.

3. **Incremental Sequence Length:** We start with a small initial sequence length (e.g., $N = 1$) and then incrementally increase the sequence length N by a fixed step (e.g., 10,000 tokens).

4. **Inference and Timing:** For each initial sequence length N, we generate a random integer tensor of shape $(1, N)$ as input. This tensor is fed into the model to generate 6 new tokens. We measure and record the time and cache consumption required to generate each token. We always select the first (cache miss, equivalent to cache off) and third token (cache hit) for detailed analysis.

5. **Determining Maximum Sequence Length:** After clearing the model's cache, we continuously increase N and repeat step 3 until the model fails to complete inference due to an Out of Memory (OOM) error.

6.4.2 Inference Time Complexity Analysis (Figures a, b, c)

The Baseline Model's Bottleneck: As shown in Figure 8(a), the inference latency of the standard Transformer baseline exhibits a near-quadratic $\mathcal{O}(N^2)$ trend with increasing sequence length N. This phenomenon reveals a gap between theory and practice: although the KV cache reduces the theoretical computational complexity to $\mathcal{O}(N)$, in practice, the performance bottleneck shifts from floating-point operations (FLOPs) to **Memory IO**, dominated by memory copy operations.

A note on pre-allocation strategies: While engineering tricks like pre-allocating a larger memory space can mitigate this issue, it fundamentally comes at the cost of higher static memory usage.

To ensure a fair algorithmic comparison with models like TConstFormer, the baseline model in this paper does not employ such additional engineering optimizations.

The Linear Advantage of TLinFormer and TConstFormer: In contrast, both of our proposed new architectures demonstrate exceptional scalability. TLinFormer (Figure 8(b)), with its unique dual-mode performance characteristics, successfully constrains both the **upper bound (cache miss)** and **lower bound (cache hit)** of inference latency to a low-slope linear growth, significantly outperforming the baseline.

TConstFormer (Figure 8(c)) achieves an even more fundamental breakthrough. Its performance upper bound also exhibits efficient linear growth, but its performance **lower bound** (the trough during a cache hit) is completely horizontal, no longer changing with sequence length N, demonstrating the ideal **constant-time** $\mathcal{O}(1)$ **characteristic**. This observation is perfectly consistent with our theoretical analysis (see Section 4) and fundamentally establishes TConstFormer's architectural advantage in long-sequence inference scenarios.

6.4.3 KV Cache Mechanism Efficiency Comparison (Figures d, e, f):

To quantify the actual benefits of caching, we compared the inference speedup ratio of each model during cache hits versus misses.

Cache Failure of the Baseline Model. As shown in Figure 8(d), the speedup ratio of the standard model peaks at only 1.26x and rapidly decays to 1.0 as the sequence grows, confirming that in long-sequence scenarios, its naive KV cache mechanism **completely fails** due to the memory bandwidth bottleneck.

Efficient Caching of TLinFormer and TConstFormer. As shown in Figures 8(e) and (f), the cache speedup ratios of both TLinFormer and TConstFormer show a strong positive correlation with sequence length, indicating that the longer the context, the more significant the performance gain from their caching mechanisms. TLinFormer achieves over a **10x** speedup on million-token sequences, while TConstFormer, with its constant-time cache hit cost, achieves an astonishing peak speedup of over **40x**, demonstrating a massive advantage.

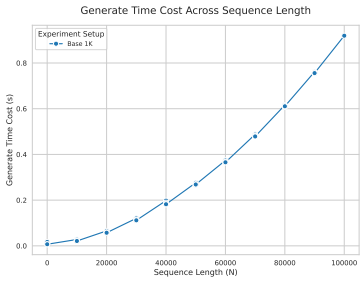
6.4.4 Memory Usage and Supported Sequence Length (Figure g)

Figure 8(g) provides a direct comparison of the cache memory usage of each model. The memory usage of both the baseline and TLinFormer grows linearly with sequence length N, although the latter has a gentler slope. TConstFormer once again demonstrates its fundamental architectural advantage, achieving **constant-level** $\mathcal{O}(1)$ **memory usage**, completely breaking free from the constraints of sequence length on memory.

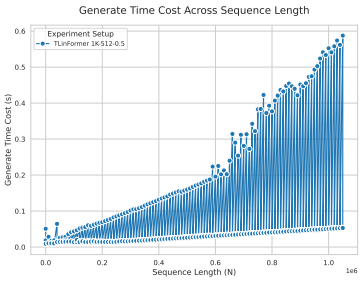
6.4.5 Overall Inference Speedup Ratio (Figures h, i)

Figures 8(h) and (i) quantify the enormous advantage of TConstFormer from an end-to-end perspective.

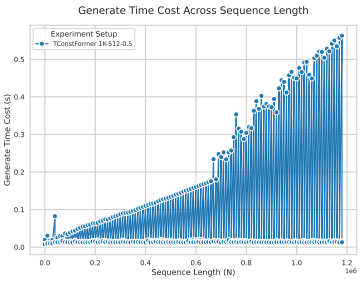
- **Compared to the Baseline Model (Figure h):** During a cache hit, TConstFormer achieves a speedup ratio that grows linearly with sequence length, reaching up to **tens of times** faster, representing an order-of-magnitude performance leap.
- **Compared to TLinFormer (Figure i):** Even when compared to the optimized TLinFormer, TConstFormer still achieves a significant and continuously growing performance advantage during a cache hit.



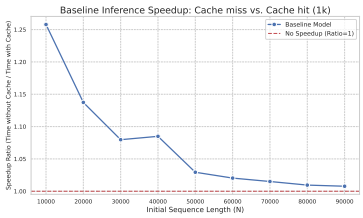
(a) Baseline model inference time



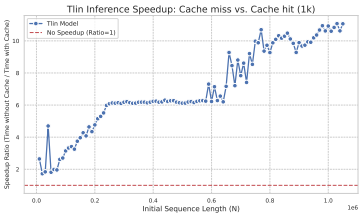
(b) TLinFormer inference time



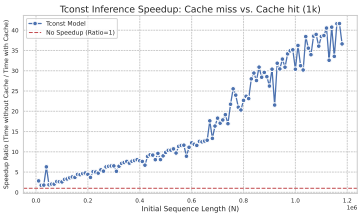
(c) TConstFormer inference time



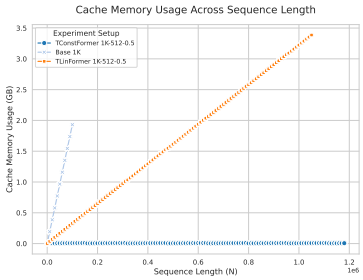
(d) Baseline cache hit speedup



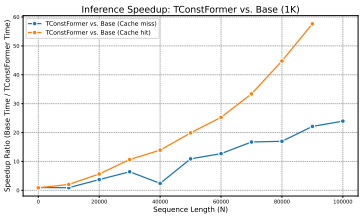
(e) TLinFormer cache hit speedup



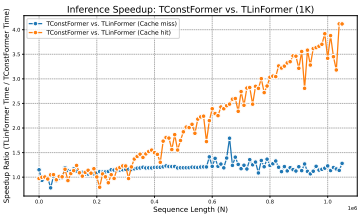
(f) TConstFormer cache hit speedup



(g) Cache memory usage of models



(h) TConstFormer vs. Baseline time ratio



(i) TConstFormer vs. TLinFormer time ratio

Figure 8: **Inference performance and cache efficiency comparison.** (**a**) Baseline model's latency grows super-linearly with sequence length. (**b**, **c**) Both TLinFormer and TConstFormer demonstrate excellent scalability. Their dual-mode performance (peaks for cache miss upper bound, troughs for cache hit lower bound) clearly validates the cache mechanism's effectiveness, with TConstFormer's lower bound being constant. (**d**) Due to a memory bandwidth bottleneck (caused by `torch.cat`), the baseline's cache speedup ratio rapidly decays towards 1 (ineffective) as sequence length increases. (**e**, **f**) In contrast, the cache speedup ratios of TLinFormer and TConstFormer grow steadily with sequence length, showing significant and sustained acceleration. (**g**) In terms of cache memory usage, both new architectures are far below the baseline, supporting longer sequences, with TConstFormer achieving O(1) cache consumption. (**h**, **i**) Overall inference time comparison shows that TConstFormer achieves orders-of-magnitude speedup over the baseline and outperforms TLinFormer.

6.5 Conclusion

In summary, the experimental results collectively validate that TConstFormer is not just an effective improvement, but a solution with an overwhelming advantage in both time and space efficiency for long-sequence inference tasks.

7 From Efficient Compression to Constant State: TConstFormer and Emergent Intelligence

First, let's consider the limitations of the standard auto-regressive architecture. When we view a decoder-only architecture from a fully-connected perspective, it can be understood as the model internally creating an observation window that expands equivalently as the sequence length increases. This processing logic is unreasonable because for an infinitely long sequence, the model would need to generate an infinitely long internal observation window. External hardware constraints mean it must have an upper limit on the sequence length it can handle. This leads to an inevitable conclusion: **for a truly efficient intelligent agent, its understanding of the world (i.e., its internal state) must be completely decoupled from the length of its history in terms of computational and storage resources**. Therefore, the observation window for this input sequence must be bounded.

Second, since compression is necessary, what is the constraint between the compressed length and the original length? This is what we will explore next. Foundational principles from information theory and compressed sensing, notably the empirical guideline $n > C \log N$, establish that high-dimensional signals often reside on a low-dimensional manifold [3, 1]. Here, n is the dimension of the compressed representation required to faithfully reconstruct a signal of original dimension N. This theoretical underpinning suggests that for processing long sequences, a remarkably small context window can be sufficient. For instance, to capture the essential information of a sequence with $N = 10^7$ tokens, a compressed representation of dimension $n \approx 134$ (for $C \approx 8.33$) could theoretically suffice.

Our TConstFormer architecture is precisely a solution to the aforementioned constraint. It not only inherits the "forced compression" philosophy of TLinFormer but elevates it to a new level. By achieving **constant-time** $\mathcal{O}(1)$ **cache updates and memory footprint**, TConstFormer implements architecturally what we call a **"Constant-State Representation"** mechanism.

This means that whether the historical sequence is one thousand, one million, or one billion tokens long, the "high-information-density state" that TConstFormer relies on to generate the next token is **completely invariant** in terms of storage cost and amortized computation. The model is thoroughly deprived of the ability to use "history length" as a shortcut. It is **forced** to learn a **scale-invariant knowledge distillation capability**—to indiscriminately refine historical information of any length into an internal state of fixed complexity that represents the core regularities of the world.

We speculate that this **Physical Constraint** on the complexity of the internal state may be a key prerequisite for the emergence of intelligence, especially general intelligence. Just as the human brain processes an infinite amount of information from the real world within a finite volume and energy budget, a truly general AI must also learn to model and predict an infinite stream of information with constant resource expenditure.

The constant-level efficiency of TConstFormer is more like a "golden hoop" imposed on the intelligent agent, one that aligns with the laws of the physical world. It is this very constraint that compels the model to move beyond its dependency on sequence length and learn deeper, more fundamental abstract principles.

Therefore, we believe that TConstFormer is not only a major step forward in the efficiency of long-sequence modeling but also a meaningful attempt to explore the **computational essence of intelligence**. Its "Constant-State" design philosophy gives us a clearer and more exciting glimpse of the path toward AGI.

8 Limitations and Discussion

Although TConstFormer demonstrates significant efficiency advantages both theoretically and experimentally, we must acknowledge that this study has several limitations, which also point toward directions for future work.

Model Scale and Task Complexity. The experimental validation in this study was primarily conducted on a small-scale model with approximately 41M parameters, largely due to the computational resource constraints of individual research. Consequently, the performance of TConstFormer on large language models at the scale of billions of parameters, as well as its ability to replicate the emergent capabilities in complex tasks (such as long-context instruction following), remains an open question. Scaling the TConstFormer architecture to larger models is a critical next step to validate its effectiveness in real-world, complex applications.

On the Capability for Precise Information Retrieval. The 'Constant-State' mechanism of TConstFormer essentially compresses and distills historical information. While this mechanism is advantageous for learning macroscopic semantics and structural patterns in text, its performance on tasks requiring **verbatim recall** (such as the 'Needle in a Haystack' test) is an area where we have not yet obtained definitive data, due to the limited scale of our model. This also represents a highly valuable direction for future research.

9 Conclusion and Future Work

In this paper, we departed from the mainstream paradigm of attention approximation and returned to the first principles of connectionism, proposing a constant-attention architecture—TConstFormer—from the perspective of information flow topology that achieves a fundamental breakthrough in efficiency. By identifying and reconstructing the performance-bottleneck pathways in TLinFormer, TConstFormer inherits its **computational precision** and **full-context accessibility** while, for the first time, reducing both the computational complexity and KV cache overhead of autoregressive inference to a **constant level ($\mathcal{O}(1)$)**. Experiments demonstrate that TConstFormer provides an extremely efficient, robust, and scalable solution for long-sequence modeling, thereby significantly lowering the hardware barrier for ultra-long sequence applications.

As discussed in Section 8, while this study has limitations regarding model scale, the "Constant-State" architecture of TConstFormer opens up several exciting directions for future AI model design:

- **Integration with Cutting-Edge Technologies:** TConstFormer's constant-level efficiency is orthogonal to and highly compatible with parameter-efficient optimization techniques such as Mixture-of-Experts (MoE). Combining them holds the promise of building next-generation foundation models that reach new heights in both parameter scale and performance, all within a limited computational budget.
- **Towards Infinite Context and Streaming Processing:** The $\mathcal{O}(1)$ inference cost means TConstFormer is naturally suited for **streaming processing of infinitely long sequences**, such as handling real-time video streams, unending dialogues, or continuous sensor data. Exploring its application in such open-ended tasks is a highly promising direction.
- **Exploring Higher-Dimensional Tensorial Attention:** The core idea of this paper leads to a more profound question: can we generalize this efficiency optimization, based on connection topology, from sequences (L) to higher-dimensional tensor data (e.g., video

data [T, H, W, C])? Developing computationally feasible High-dimensional Tensorial Attention could be a key step towards more general and powerful AI models.

Finally, in Section 7, we discussed the necessity of compression and pointed out that TConstFormer is precisely a solution that embodies this philosophy.

Code Availability

The source code for this paper is available at https://github.com/simonFelix-Ai/TConstFormer. The code is dual-licensed for academic and commercial use.

References

[1] Mohammed M. Abo-Zahhad, Aziza I. Hussein, and Abdelfatah M. Mohamed. Compressive Sensing Algorithms for Signal Processing Applications: A Survey. *International Journal of Communications, Network and System Sciences*, 08(06):197–216, 2015.

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The Long-Document Transformer, December 2020. arXiv:2004.05150 [cs].

[3] Emmanuel Candes, Justin Romberg, and Terence Tao. Robust Uncertainty Principles: Exact Signal Reconstruction from Highly Incomplete Frequency Information, September 2004. arXiv:math/0409186.

[4] Zhongpan Tang. Rethinking Transformer Connectivity: TLinFormer, A Path to Exact, Full Context-Aware Linear Attention, August 2025. arXiv:2508.20407 [cs].

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023. arXiv:1706.03762 [cs].

A Detailed Derivation of Computational Complexity

This appendix provides a detailed derivation of Equations (1) and (5) from the main text. We analyze the upper bound of the computational cost by considering a full computation cycle where both the context and generation windows are updated. The analysis is divided into costs associated with the left (context) and right (generation) windows.

A.1 Cache Miss

1. **Computational Cost of the Left Window (Historical Context)**:
    - **First Layer Cross-Attention**: The query sequence from the context window attends to the full history. Cost is $D \cdot (N - W_{og}) \cdot W_{oh}$.
    - **Intermediate Self-Attention Layers (H layers)**: Self-attention is performed within the context window of size W_{oh}. Cost is $H \cdot D \cdot W_{oh}^2$.
    - **Final Layer Cross-Attention (Dimension Restoration)**: The full history attends to the processed context window to restore the original sequence length. Cost is $D \cdot (N - W_{og}) \cdot W_{oh}$.
    - ***Total Cost of the Left Window (C_{left}):***

$$C_{\text{left}} = 2D(N - W_{og})W_{oh} + HDW_{oh}^2$$

2. **Computational Cost of the Right Window (Generation Area)**:

   - **Cross-Attention with Intermediate Context Layers ($H+1$ layers, including final output layer)**: The generation window attends to the processed context window. Cost is $(H+1) \cdot D \cdot W_{og} \cdot W_{oh}$.
   - **Causal Self-Attention (All $H+2$ layers, including final output layer)**: Causal self-attention is performed within the generation window. Cost is $(H+2) \cdot D \cdot W_{og}^2$.
   - ***Total Cost of the Right Window (C_{right}):***

$$C_{\text{right}} = (H+1)DW_{og}W_{oh} + (H+2)DW_{og}^2$$

3. **Derivation of Total Computational Cost (T):** The total cost is the sum of the costs of the two windows, $T = C_{\text{left}} + C_{\text{right}}$.

$$\begin{aligned} T = &\left[2D(N - W_{og})W_{oh} + HDW_{oh}^2\right] \\ &+ \left[(H+1)DW_{og}W_{oh} + (H+2)DW_{og}^2\right] \end{aligned}$$

Step 1: Expand all terms

$$\begin{aligned} = &\left(2DNW_{oh} - 2DW_{og}W_{oh} + HDW_{oh}^2\right) \\ &+ \left(HDW_{og}W_{oh} + DW_{og}W_{oh} + HDW_{og}^2 + 2DW_{og}^2\right) \end{aligned}$$

Step 2: Combine like terms

$$\begin{aligned} &= 2DNW_{oh} - 2DW_{og}W_{oh} + DW_{og}W_{oh} + HDW_{oh}^2 + HDW_{og}W_{oh} + HDW_{og}^2 + 2DW_{og}^2 \\ &= 2DNW_{oh} - DW_{og}W_{oh} + HDW_{oh}^2 + HDW_{og}W_{oh} + HDW_{og}^2 + 2DW_{og}^2 \end{aligned}$$

Step 3: Factor out common factor D and reorganize by variables N and H

$$= D\left[2NW_{oh} - W_{og}W_{oh} + HW_{oh}^2 + HW_{og}W_{oh} + HW_{og}^2 + 2W_{og}^2\right]$$

Step 4: Final organized form (grouping terms related to N and H)

$$= D\left[N(2W_{oh}) + H(W_{oh}^2 + W_{og}^2 + W_{og}W_{oh}) + 2W_{og}^2 - W_{og}W_{oh}\right]$$

Derivation complete.

A.2 Cache Hit

1. **Computational Cost of the Left Window (Historical Context)**:

$$C_{\text{left}} = 0$$

2. **Computational Cost of the Right Window (Generation Area)**:

   - **Cross-Attention with Intermediate Context Layers ($H+1$ layers, including final output layer)**: Only the last token of the generation window participates in the computation. Cost is $(H+1) \cdot D \cdot W_{oh}$.
   - **Causal Self-Attention (All $H+2$ layers, including final output layer)**: Causal self-attention is performed within the generation window. The upper bound on cost is $(H+2) \cdot D \cdot W_{og}^2$.

- ***Total Cost of the Right Window (C_{right}):***

$$C_{\text{right}} = (H+1)DW_{oh} + (H+2)DW_{og}^2$$

3. **Derivation of Total Computational Cost (T):** The total cost is the sum of the costs of the two windows, $T = C_{\text{left}} + C_{\text{right}}$.

$$T = (H+1)DW_{oh} + (H+2)DW_{og}^2$$

Derivation complete.