TNT: IMPROVING CHUNKWISE TRAINING FOR TEST-TIME MEMORIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Recurrent neural networks (RNNs) with deep test-time memorization modules, such as Titans and TTT, represent a promising, linearly-scaling paradigm distinct from Transformers. While these expressive models do not yet match the peak performance of state-of-the-art Transformers, their potential has been largely untapped due to prohibitively slow training and low hardware utilization. Existing parallelization methods force a fundamental conflict governed by the chunksize hyperparameter: large chunks boost speed but degrade performance, necessitating a fixed, suboptimal compromise. To solve this challenge, we introduce TNT, a novel training paradigm that decouples training efficiency from inference performance through a two-stage process. Stage one is an efficiency-focused pre-training phase utilizing a hierarchical memory. A global module processes large, hardware-friendly chunks for long-range context, while multiple parallel local modules handle fine-grained details. Crucially, by periodically resetting local memory states, we break sequential dependencies to enable massive context parallelization. Stage two is a brief fine-tuning phase where only the local memory modules are adapted to a smaller, high-resolution chunksize, maximizing accuracy with minimal overhead. Evaluated on Titans and TTT models, TNT achieves a substantial acceleration in training speed—up to 17$\times$ faster than the most accurate baseline configuration—while simultaneously improving model accuracy. This improvement removes a critical scalability barrier, establishing a practical foundation for developing expressive RNNs and facilitating future work to close the performance gap with Transformers.

1 INTRODUCTION

The demand for modeling long sequences highlights a fundamental limitation of standard softmax attention (Vaswani et al., 2017): its quadratic complexity bottlenecks scaling. This has spurred extensive research into more efficient architectures.

Among these emerging paradigms, a particularly powerful approach is rooted in test-time memorization (Sun et al., 2024). Architectures leveraging this principle, which we refer to as **deep memory modules**, utilize a deep, online-adapted sub-network whose parameters are rapidly updated to encode in-context information. Prominent examples include Titans (Behrouz et al., 2025d) and Atlas (Behrouz et al., 2025a). This method stands in sharp contrast to **linear memory modules** (Yang et al., 2024a;b; Dao & Gu, 2024; Sun et al., 2023), which, despite their efficiency, are constrained by matrix-valued hidden states and linear state transitions. By leveraging expressive non-linear objectives and update rules, deep memory modules can theoretically overcome these limitations. While these methods generally do not yet achieve the state-of-the-art performance of Transformers, they represent a potentially promising paradigm for efficient sequence modeling, provided their training bottlenecks can be resolved.

Despite their expressive advantages, deep memory modules lack the efficient training algorithms of their linear counterparts, leading to low hardware utilization. Unlike linear memory modules, which utilize hardware-efficient parallelization, deep memory modules face challenges stemming from non-linear recurrences (e.g., LayerNorm between chunks) and the complexity of their deep structures. In practice, these challenges constrain their training to more frequent online updates on small data segments, resulting in poor computational throughput in training. This creates an in-

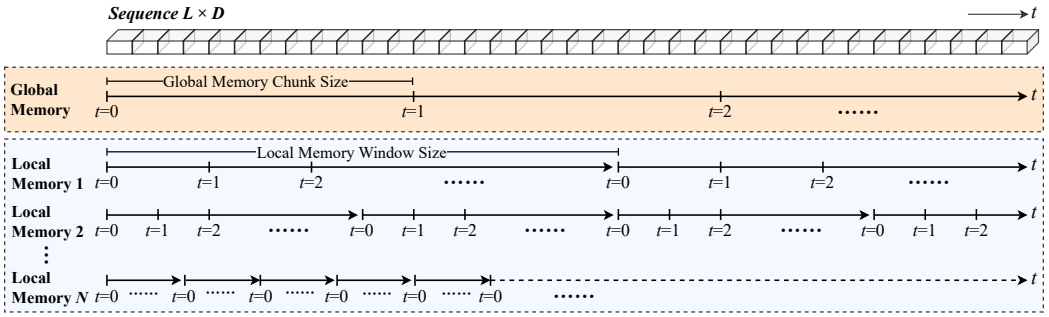

Figure 1: The basic diagram for illustraing TNT memory hierarchy. In each row, the updates at the same value of t ran at the same time (run in parrallel). $t = 0$ is the initialization of the memory.

herent tension, as these models typically rely on a fixed, small chunk size (e.g., 16 to 64 tokens) to balance memory layer expressiveness against training efficiency. Consequently, this trade-off between in-context learning capability and computational performance has become a critical bottleneck preventing the application of these models to truly long sequences in practice. Resolving this fundamental tension is the primary goal of this work.

Recent work attempts to mitigate this issue. Zhang et al. (2025) combines large chunks with local attention to enhance parallelism. However, this circumvents the inefficiency rather than solving it, complicates the analysis by mixing memory and attention, and neglects the need for small chunks (ideally 1) during inference. Concurrently, Guo et al. (2025) proposed a hierarchical memory system, but it is limited to linear memory modules and does not support short-term memories.

To resolve this tension, we introduce TNT[1], a novel training paradigm for deep memory modules. Our core insight is that different components of the model should process information at different granularities during distinct training stages. TNT begins with an **efficiency-focused pre-training stage** designed to maximize throughput. This is achieved via a hierarchical memory system: a **global memory** module operates on large, hardware-friendly chunks to capture long-range context, while multiple **local memory** modules handle fine-grained details in parallel. Crucially, we introduce a periodic reset mechanism for the local memory states. This breaks the sequential dependencies inherent even in non-linear RNNs (e.g., those with normalization between steps), enabling massive context parallelization. This is a key innovation, as efficiently parallelizing *non-linear* recurrences across the sequence length is a long-standing challenge, largely unsolved outside of Transformers and specialized linear RNNs (where parallel scans apply). Subsequently, a **performance-focused fine-tuning stage** adapts the model for optimal inference. During this stage, only the local memory modules are adjusted to use smaller chunk sizes, achieving high-resolution accuracy with minimal additional computational cost. This two-stage approach effectively decouples training efficiency from inference performance, significantly improving training scalability while addressing a key limitation of prior architectures. Furthermore, the local memory system itself can be hierarchical, employing multiple modules operating at different resolutions. This *multi-resolution* approach allows the model to capture *complex, multi-scale temporal dynamics* more effectively than a single fixed chunk size.

TNT is a general training paradigm applicable to any deep memory module rather than a specific architecture. By decoupling training throughput from inference accuracy, we resolve a fundamental tension constraining prior work. This removes dependency on hardware-specific optimizations for small chunks and enables flexible exploration of the architectural design space. We believe this paradigm will open new research avenues towards replacing softmax attention. Our main contributions are summarized as follows:

- We identify three fundamental challenges limiting the scalability and performance of deep memory modules: 1) domain mismatch between memory compression and retrieval; 2) tradeoff between memory performance and computational efficiency; 3) chunksize mismatch between training and inference (Section 3).

[1]TNT can be viewed as an abbreviation of *Two-stage Non-linear Training* or *TTT iNside TTT*. It also hints to its "explosive" impact on training efficiency.

- We introduce Q-K Projection, an efficient mechanism to resolve the domain mismatch between memory compression and retrieval (Section 4.1.2).
- We introduce a novel hierarchical memory architecture with periodic state resets, enabling context parallelism for non-linear deep memory modules (Section 4.1).
- We introduce an efficient fine-tuning mechanism to address chunksize mismatch between training and inference in deep memory modules (Section 4.2).
- Putting all above together, we introduce TNT, a general two-stage training paradigm that decouples training efficiency from inference performance by combining efficient pre-training with high-resolution fine-tuning (Figure 1, Figure 3, Section 4).
- We validate TNT on the Titans architecture, achieving up to a $17.37\times$ training speedup while improving accuracy, significantly advancing the practicality of expressive RNNs (Section 5).

Problem Definition and Notations We aim to train a neural network with parameters $\boldsymbol{\theta} \in \mathbb{R}^{d_m}$ to perform next-token prediction. Given a sequence $\mathbf{x} = (x_1, \ldots, x_L)$, the model's objective is to predict each token x_t using the context of its preceding tokens $(x_1, \ldots, x_{t-1})$. Following the attention formulation, each token x_t is represented by a d-dimensional vector. Each input token $\mathbf{x}_t$ is projected into query, key, and value vectors: $q_t, k_t, v_t \in \mathbb{R}^d$. For ease of notation in subsequent chunkwise operations, we define a function $\xi(i, j) := i - (i \mod j)$, which finds the beginning of the chunk containing index i for a chunk size j.

2 Preliminary

This section reviews preliminaries. Expanded related work is in Appendix B.

2.1 Deep Memory Modules via Test-Time Memorization

A powerful paradigm for sequence modeling is Test-Time Memorization (Sun et al., 2024), which enhances models by incorporating a secondary, rapidly adaptable neural network. Unlike the primary model parameters, or "slow weights" (θ) updated only during training, this approach introduces "fast weights" (Schlag et al., 2021). These fast weights, denoted by W, parameterize a sub-network, $f(W, \cdot) : \mathbb{R}^d \to \mathbb{R}^d$, that is updated online-during both training and inference-based on incoming tokens to dynamically store contextual information. While these modules do not yet achieve SOTA results compared to Transformers (Arora et al., 2024; Behrouz et al., 2025a), improving their training efficiency is crucial for enabling the wider experimentation needed to close this gap.

In this work, we focus on a similar/relevant principle: **deep memory modules** (Irie et al., 2021; Sun et al., 2024; Behrouz et al., 2025d;a;c). In contrast to **linear memory modules** (Sun et al., 2023; Yang et al., 2024b; Dao & Gu, 2024; Karami & Mirrokni, 2025; Hu et al., 2025), which are characterized by linear state transitions, deep memory modules employ non-linear recurrence rules and complex memory structures.

The core mechanism of a deep memory module can be distilled into two sequential operations for each input token: *1. Memory Compression* and *2. Memory Retrieval*. These are formally defined as:

$$\textit{Memory Compression:} \quad W_t \leftarrow W_{t-1} - \eta_t \nabla_W \mathcal{L}\big(f(W_{t-1}, k_t), v_t\big) \tag{1}$$

$$\textit{Memory Retrieval:} \quad o_t = f(W_t, q_t) \tag{2}$$

In *Memory Compression*, the fast weights W are updated via gradient descent, guided by a self-supervised loss $\mathcal{L}(\cdot, \cdot)$ (e.g., MSE) and a learned learning rate η_t. The objective associates a transformed key, $f(W_{t-1}, k_t)$, with its value, v_t, compressing information into the fixed-size neural memory (Wang et al., 2025; Behrouz et al., 2025b). In *Memory Retrieval*, the updated W_t processes a query q_t to produce o_t. These two operations are performed iteratively for each token.

2.2 Chunkwise Parallel Training

The sequential dependency (W_t depends on W_{t-1}) in Eqs. 1-2 prevents parallelization across the sequence length. To address this, deep memory modules adopt chunkwise parallel training (Hua et al., 2022; Sun et al., 2023) to enable hardware-efficient training.

The core principle is to divide the input sequence into non-overlapping chunks of size C. Within each chunk, an approximation of the gradient of the loss for every token is computed with respect to the fast-weight state from the beginning of that chunk. This formulation breaks the sequential token-to-token dependency for gradient calculation, which allows the updates for all tokens within the chunk to be computed in parallel. The formal operations for a token at time step t are as follows:

$$\textit{Chunkwise Memory Compression:} \quad W_t \leftarrow W_{\xi(t,C)} - \sum_{\tau=\xi(t,C)}^{t} \eta_\tau \nabla_W \mathcal{L}\big(f(W_{\xi(t,C)}, k_\tau), v_\tau\big) \tag{3}$$

$$\textit{Chunkwise Memory Retrieval:} \quad o_t = f(W_t, q_t) \tag{4}$$

Here, $W_{\xi(t,C)}$ denotes the state of the fast weights at the start of the chunk containing token t (See the definition of $\xi(\cdot,\cdot)$ at the end of Section 1). Although the update to obtain W_t still depends on prior tokens within its chunk, the summation of gradients can be implemented efficiently using parallel operations (e.g., cumulative summation), significantly improving hardware utilization during training. However, a sequential dependency remains: the final state of the fast weights from the k-th chunk, W_{kC}, is used as the initial state for the $(k+1)$-th chunk.

3 Challenges in Deep Memory Modules

While chunkwise parallelization enables deep memory modules to train on long sequences, this paradigm introduces significant challenges that limit their practical performance and scalability. In this section, we outline three fundamental challenges with deep memory modules.

Challenge 1: Lack of Efficient Training Implementations. A primary challenge for deep memory modules is the inefficiency of their training process, which leads to poor hardware utilization. While chunkwise parallelization theoretically enables sub-quadratic scaling, in practice, the training throughput lags significantly behind that of linear memory modules. This discrepancy arises from a fundamental tension between model expressiveness and computational efficiency.

To maintain a fine-grained learning signal, deep memory modules require small chunk sizes (e.g., 16-64 tokens) (Sun et al., 2024), which fail to saturate accelerators, making training memory-bound (rather than compute-bound). While linear memory modules use customized kernels (e.g., leveraging SRAM) (Sun et al., 2023; Gu & Dao, 2023; Qin et al., 2024; Yang et al., 2024a;c), this relies on linear state transitions and is incompatible with the large, non-linear states of deep memory modules.

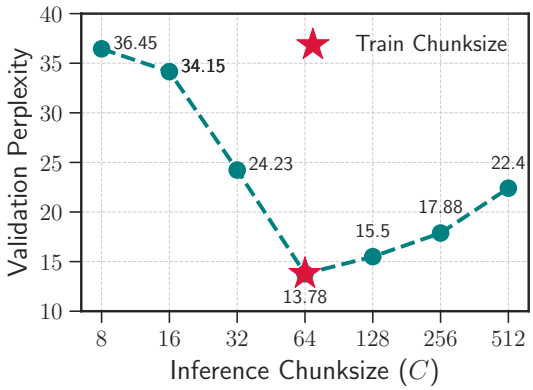

Figure 2: Sensitivity of inference chunk size on a 550M Titans model pre-trained with $C = 64$. Performance is optimal when the inference chunk size matches the training one.

The consequence is that deep memory modules suffer from extremely low FLOPs utilization, often falling below 5-10% of peak hardware performance (Zhang et al., 2025). This severe inefficiency makes pre-training prohibitively slow and costly, creating a major bottleneck that undermines the practical application of these expressive models to truly long sequences.

Challenge 2: Inconsistency Between Memory Compression and Retrieval. A fundamental inconsistency exists between how the memory sub-network is trained and how it is utilized. During Memory Compression (Eq. 1), the sub-network $f(W, \cdot)$ is optimized to learn a mapping from the key space to the value space by associating keys (k_t) with values (v_t). However, during Memory Retrieval (Eq. 2), the network is queried using a query vector (q_t) instead of a key. This substitution violates the intended input domain of the learned function, creating a discrepancy between the training objective and the retrieval task. This domain shift can degrade the integrity of the learned mapping and limit the model's retrieval performance. Our empirical validation can be found in Section 5.4

Challenge 3: Performance Sensitivity to a Fixed Pre-training Chunksize. The chunk size hyperparameter, C, governs the trade-off between training throughput and model expressiveness. Current practice for deep memory modules is to use the same fixed chunk size for both pre-training and inference. However, we observe that inference-time performance is highly sensitive to this pre-training choice. For example, as shown in Figure 2, a model pre-trained with a chunk size of 64 achieves optimal perplexity only when evaluated with that same chunk size.

This result reveals a critical train-test mismatch and contradicts the intuition that smaller chunks at inference should yield superior performance by capturing fine-grained dependencies with a "fresher" learning signal. Instead, the model becomes over-specialized to the specific chunk resolution seen during training. This inflexibility is a significant limitation; ideally, a model pre-trained with a large, hardwarefriendly chunk size should be adaptable enough to perform even better with smaller, more precise chunk sizes at inference. Current deep memory modules fail to achieve this adaptability.

4 TNT: An Improved Training Framework for Deep Memory

To address the challenges outlined in Section 3, we introduce TNT, an improved training paradigm for deep memory modules. Our framework is structured around a two-stage process designed to resolve the inherent tension between training efficiency and inference performance: an **Efficiency-focused Pre-training Stage** and a **Performance-focused Fine-tuning Stage**.

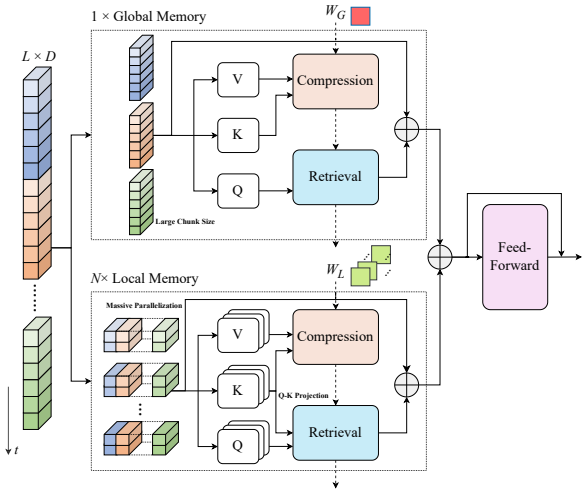

Figure 3: Architectural overview of TNT Stage 1.

The first stage maximizes training throughput by introducing a novel hierarchical memory architecture that enables unprecedented parallelism, directly addressing the challenges of low hardware utilization and inconsistent memory objectives (Challenges 1-2). The second stage employs an efficient fine-tuning strategy that adapts the model to high-resolution, small-chunk inference, resolving the sensitivity to the pre-training chunk size (Challenge 3). This two-stage approach effectively decouples training efficiency from inference performance, overcoming a key limitation of prior deep memory architectures.

4.1 TNT Stage 1: Efficient-focused Pre-training

4.1.1 TNT Memory Compression: Hierarchy Memory

Sequential state dependency prevents context parallelism (processing sequence shards in parallel across devices). To enable this, we propose that all parallel shards initialize their **local memory** with the same learned state, W_{init}. This breaks inter-chunk dependency, allowing massive parallelization. However, this causes local memory modules to lose the global context. To solve this, we introduce a **global memory** module, parameterized by V, that operates in parallel with the sharded local memories. The global memory processes the sequence with a relatively large chunk size (e.g., 2048 or greater), allowing it to efficiently capture long-range dependencies while maintaining high hardware utilization. This creates a hierarchical system where local memories handle fine-grained information within parallel shards, while the global memory provides the overarching context.

This hierarchical structure is flexible; a model can be designed with 1 global and N local memory modules, each operating at a different resolution. For clarity of illustration, we will assume the simplest case where $N = 1$. We defer the generalized formulation of TNT to Appendix F. We now formally define our memory compression mechanism.

TNT Memory Compression Rule. The hierarchical memory is updated as follows:

Global Memory Update. The global memory state V evolves sequentially across the input with a large chunk size C_G.

$$V_{(k+1)C_G} \leftarrow V_{kC_G} - \sum_{t=kC_G}^{(k+1)C_G} \eta_t \nabla_V \mathcal{L}\left(f(V_{kC_G}, k_t), v_t\right) \quad k = \{0, \ldots, L//C_G\} \tag{5}$$

Local Memory Update. The local memory W operates in parallel on sequence shards of length S_L. Within each shard, updates use a smaller chunk size C_L.

$$W_t \leftarrow \begin{cases} W_{\text{init}} & \text{if } 0 \equiv t \pmod{S_L} \\ W_{t-1} - \sum_{\tau=\xi(t,C_L)}^{t} \eta_\tau \nabla_W \mathcal{L}\left(f(W_{\xi(t,C_L)}, k_\tau), v_\tau\right) & \text{Otherwise} \end{cases} \tag{6}$$

The global memory update (Eq. 5) follows a standard chunkwise formulation where the state is carried over sequentially between large chunks. To maximize training throughput, the gradient for all tokens within a global chunk is computed with respect to the initial state of that chunk, allowing for a highly parallelized update.

In contrast, the local memory update (Eq. 6) introduces our key innovation: a periodic state reset. This rule enforces that the local memory state, W_t, is reset to a shared, learnable initial state W_{init} at the beginning of each segment of length S_L. This periodic reset is the critical mechanism that breaks the long-range sequential dependency across the input, thereby enabling true context parallelism for the fine-grained local memory modules.

The hierarchical design of deep memory modules boosts training efficiency through a two-pronged approach. Global modules create hardware-saturating, compute-intensive operations by processing large chunks. Concurrently, the local memory's reset mechanism enables context parallelism, where the sequence is processed as independent chunks that can be distributed across devices or stacked on a single accelerator to substantially increase training throughput.

4.1.2 TNT Memory Retrieval: Q-K Projection

As identified in Challenge 2, the memory compression step (Eq. 6) optimizes $f(W, \cdot)$ to map the key space to the value space. However, at retrieval, the network is queried using a query vector, q_t, which may lie outside the learned key domain, degrading performance. To resolve this, we propose *Q-K Projection*: projecting the query q_t onto the subspace spanned by previously observed keys. This ensures the input to the memory function is in the space memory was trained on. The final output combines retrieval from the global memory (raw query) and the local memory (projected query). We apply projection only locally as its fine-grained nature makes it more sensitive to the mismatch

TNT Memory Retrieval Rule. The hierarchical memory is retrieved as follows:

$$o_t = f(V_{\xi(t,C_G)}, q_t) + f\left(W_t, \sum_{\tau=\xi(t,C_L)}^{t} \frac{k_\tau k_\tau^\top}{\|k_\tau\|^2} q_t\right) \tag{7}$$

Crucially, this Q-K Projection does not require storing all past keys, which would be computationally and memory prohibitive. Instead, the projection matrix, $\sum_{\tau=1}^{t} \frac{k_\tau k_\tau^\top}{\|k_\tau\|^2} \in \mathbb{R}^{d\times d}$, can be maintained as a running sum. This results in a constant-size state that is updated efficiently in a chunkwise parallel manner. Since many modern deep memory modules normalize the query (q_t) and key (k_t) vectors by their L2 norm, the denominator in the Q-K projection can simplify to $\sum_{\tau=1}^{t} k_\tau k_\tau^\top$. We provide further details on this efficient implementation in Appendix D.

4.2 TNT Stage 2: Performance-focused Fine-tuning at Finer Resolution

Having addressed training efficiency in Stage 1, we now turn to optimizing for inference performance. An intuitive approach to enhance model resolution would be to evaluate the pre-trained model using a smaller chunk size. However, as established in Challenge 3, a direct mismatch between the pre-training and evaluation chunk sizes leads to significant performance degradation.

Table 1: TNT reaches the target training loss up to 17× faster than the baseline Titans. The table compares the time required for different 150M models to reach the same target loss 3.20.

Models	Implementation	C or C_L	Training Time (hrs)	Speedup
Titans	JAX	8	19.48	1.00×
Titans	JAX	16	10.79	1.81×
Titans	JAX	32	6.45	3.02×
Titans	JAX	64	4.18	4.67×
Titans	JAX	128	3.71	5.25×
Transformer (w/o gating)	JAX	-	1.74	11.18×
Transformer (w gating)	JAX	-	1.38	14.10×
Transformer (w/o gating)	FlashAttention (Pallas)	-	1.23	15.90×
Transformer (w gating)	FlashAttention (Pallas)	-	0.96	20.22×
TNT	JAX	{8}	2.54	7.68×
TNT	JAX	{16}	1.65	11.78×
TNT	JAX	{32}	1.22	15.92×
TNT	JAX	{64}	1.12	17.37×
TNT	JAX	{128}	1.16	16.75×

Our key insight is that this train-test discrepancy can be rectified with minimal computational overhead. We empirically observe that a brief fine-tuning phase, where the pre-trained model is updated for a small number of steps using a smaller local memory chunk size, not only recovers but often surpasses the original performance.

Based on this finding, we introduce Stage 2 of our TNT framework: a **Performance-focused Fine-tuning Stage**. In this stage, we continue training the efficiently pre-trained model with a smaller local chunk size ($C'_L < C_L$). This process adapts the model to the higher resolution required for optimal inference at a fraction of the cost of pre-training. By doing so, Stage 2 directly resolves Challenge 3, bridging the gap between the large chunk sizes required for efficient training and the small chunk sizes that yield the best performance at inference.

This two-stage process decouples pre-training efficiency from inference requirements. The bulk of training uses maximum throughput (large chunks), while the final model is produced with minimal overhead. Furthermore, fine-tuning specializes the model for the ideal inference scenario: a local chunk size of one ($C'_L = 1$). This aligns with the standard prefill-and-decode paradigm of autoregressive generation. The global memory handles the context prefill, and the optimized local memory handles iterative decoding.

5 Experiments

We empirically evaluate our two-stage training framework, TNT. While TNT is model-agnostic, we instantiate it with a strong deep memory model, Titans (Behrouz et al., 2025d), to demonstrate its effectiveness. We validate claims about training time and model accuracy in our experiments.

5.1 Experimental Setup

Baselines. We compare against several strong long-context architectures. Our primary comparison is Titans (Behrouz et al., 2025d), our base model. We also benchmark against vanilla Transformer (Vaswani et al., 2017), Gated Transformer (Qiu et al., 2025), and TTT (Sun et al., 2024).

Training and TNT Configuration. We train 150M parameter models following (Behrouz et al., 2025d), using a T5 tokenizer (32k vocab). We use the AdamW optimizer (Loshchilov & Hutter, 2017) with 0.1 weight decay and a cosine schedule (peak LR 1×10^{-3}). Experiments are conducted on a TPUv4 pod (2x2x2 topology, model parallelism 2). For TNT, the N local modules configuration is denoted by their chunksizes, $C_L = \{C_{L,1}, ..., C_{L,N}\}$. For instance, $C_L = \{8, 16\}$ indicates two local modules with chunksizes 8 and 16. The global memory uses $C_G = 2048$.

Experimental Configurations. For efficiency benchmarks (Sec. 5.2), we vary context length (2k-32k) with a 0.5M token batch size and local window $S_L = 2048$. For performance evaluation (Sec 5.3), we use a 16k context length, 1M token batch size, and $S_L = 4096$.

5.2 Faster Memory Training with TNT

Linear Runtime Scaling with Sequence Length. We first analyze single-step runtime performance by varying the sequence length while keeping the number of tokens per batch fixed. As shown in Figure 4, TNT's runtime grows linearly with sequence length, in contrast to the quadratic growth

of Titans and standard attention. This scaling advantage is significant at long contexts. At a 32K sequence length, TNT is **5.1× faster** than a comparable Titans model with the same memory chunksize ($C_L = C = 16$). We also observe that larger local chunk sizes consistently improve TNT's speed; with $C_L = \{128\}$, TNT is **1.3× faster** than the highly optimized FlashAttention kernel (Dao, 2024).

TNT's highly parallelizable architecture achieves a runtime that scales linearly with sequence length, a key advantage over the quadratic complexity of standard attention. Although models like Titans are also theoretically linear, their inherent sequential dependencies impede effective parallelization, resulting in poor hardware utilization and slower wall-clock times on long sequences. As sequence length increases, TNT's superior scalability creates a crossover point where it becomes significantly faster. This efficiency is most pronounced at very long contexts; for instance, at a sequence length of $32K$, a native JAX implementation of TNT ($C_L = 128$) outperforms even the highly optimized FlashAttention kernel, confirming its suitability for demanding long-context training scenarios.

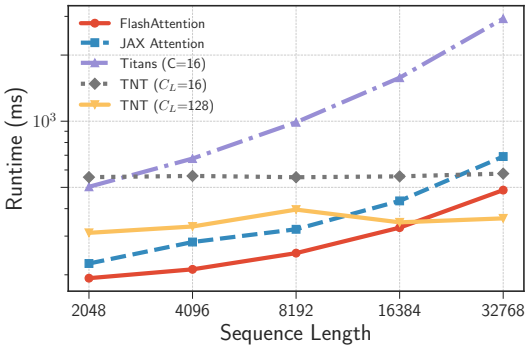

Figure 4: Runtime comparison of different models and implementations across varying sequence lengths, with the number of tokens per batch fixed at 0.5M. Additional results are presented in Figure 5.

Time-to-Quality Comparison. We next translate these single-step runtime gains into a practical time-to-quality setting. As shown in Table 1, our TNT framework significantly accelerates the total training time required to reach a target model quality. Our best configuration achieves this up to **17.4× faster** than the original Titans baseline. This efficiency gain is fundamental to the architecture; for instance, using an identical local memory chunksize of 8, TNT is already **7.7× faster** than its Titans counterpart. While competitive with standard vanilla Transformers in JAX, our implementation does not yet outperform highly optimized baselines like the Gated Transformer with FlashAttention (Dao et al., 2022). This is an expected result, as TNT currently lacks a custom kernel, which we leave for future work. Nonetheless, these results establish TNT as an efficient foundation for research on recurrent models, with a clear path toward matching the speed of state-of-the-art Transformers.

5.3 TNT Improves Model Quality

Our TNT framework significantly enhances model quality, outperforming strong RNN-based baselines and standard Transformer implementations. As detailed in Table 2, the initial **Stage 1 pre-training** is highly effective on its own. Our best Stage 1 model achieves an average perplexity of **23.13**, a marked improvement over the best-performing Titans model (25.07) and the vanilla Transformer (23.58). While TNT does not fully match the perplexity of the state-of-the-art Gated Transformer (22.39), it achieves a higher average accuracy on common-sense reasoning tasks (**41.0%** vs. 39.7%). At this scale, we consider perplexity a more stable metric for language modeling capability, as downstream task accuracy can be subject to higher variance.

Furthermore, the **Stage 2 fine-tuning** process offers an efficient method to further boost performance. This stage is computationally inexpensive, requiring only an additional 5% of the original pre-training compute (see Table 4), yet it consistently lowers the average perplexity to a final value of **23.09**. These results validate TNT as an effective framework for producing high-quality models that surpass the limitations of prior RNN-based architectures and stand as a strong alternative to standard Transformers.

5.4 Ablation Study

Table 2: Performance of TNT (150M parameters) and baselines on language modeling and common-sense reasoning tasks, trained on 10B tokens. For TNT models, we only use 1 global memory and use C_G to denote the global chunksize and the N local modules configuration is denoted by their chunksizes, $C_L = \{C_{L,1}, ..., C_{L,N}\}$. For instance, $C_L = \{8, 16\}$ indicates two local modules with chunksizes 8 and 16. The best results within a block are highlighted. The detailed training time is reported in Table 4

Model	C_G	C or C_L	C4 ppl ↓	FineWeb ppl ↓	PG19 ppl ↓	Avg. ppl ↓	PIQA acc ↑	Hella. acc ↑	ARC-e acc ↑	CSQA acc ↑	Avg. acc ↑
150M params / 10B tokens											
Transformer (w/o gating)	-	-	20.98	20.59	29.18	23.58	62.0	30.9	34.8	25.5	38.3
Transformer (w gating)	-	-	19.82	19.61	27.75	22.39	63.3	32.2	36.8	26.7	39.7
DeltaNet (2024c)	-	-	22.49	22.36	31.84	25.56	62.6	32.2	35.9	27.0	39.4
GatedDeltaNet (2025)	-	-	21.25	21.37	30.60	24.40	63.0	31.1	35.2	27.8	39.3
TTT (2024)	-	256	24.18	24.31	34.36	27.62	60.6	30.8	34.1	26.9	38.1
Titans (2025d)	-	256	23.53	24.13	33.73	27.13	61.3	30.8	35.1	27.8	38.8
Titans	-	8	22.25	22.07	30.90	25.07	60.8	32.0	35.5	27.8	39.0
TNT Stage 1: Efficiency-Focused Pre-training											
TNT Stage 1	2048	{8}	21.04	21.01	30.24	24.10	61.8	32.8	37.4	30.3	40.6
	2048	{8,16}	20.74	20.73	29.94	23.80	63.5	32.4	37.4	30.6	41.0
	2048	{4,8,16}	20.47	20.43	29.43	23.44	62.9	32.4	36.4	28.9	40.2
	2048	{4,8,16,32}	20.15	20.17	29.08	23.13	63.2	32.0	36.7	30.3	40.6
TNT Stage 2: Performance-Focused Fine-tuning on Stage 1 models											
TNT Stage 2	2048	{1}	20.86	20.91	30.21	23.99	63.2	32.8	37.4	30.1	40.9
	2048	{2,4}	20.65	20.70	29.97	23.77	63.4	32.5	37.3	30.2	40.9
	2048	{2,4,8}	20.32	20.35	29.42	23.36	64.0	32.0	36.9	28.1	40.3
	2048	{2,4,8,16}	20.10	20.13	29.05	23.09	63.5	32.3	37.4	30.2	40.9

We conducted an ablation study to validate TNT's key design choices, with results summarized in Table 3.

Table 3: Ablation study on TNT. The results show the contribution of each proposed change to the deep memory modules.

TNT	N	Language Modeling ppl ↓	C.S. Reasoning acc ↑
Base Model (Titans)	-	23.53	38.8
TNT Stage 1 (`1 Global Memory`)			
`+1 Local Memory`	1	21.04	40.6
`+2 Local Memory`	2	20.74	41.0
`+3 Local Memory`	3	20.47	40.2
`+4 Local Memory`	4	20.15	40.6
TNT Stage 1	1	21.04	40.6
w/o global memory	-	25.60	35.5
w/o Q-K projection	1	22.01	36.4
w Stage 2	1	20.86	40.9

Hierarchical Memory. The effectiveness of our hierarchical design is evident. Incrementally adding local memory modules consistently improves performance over the Titans baseline, reducing perplexity from 23.53 to 20.15 with four local modules. Conversely, removing the global memory is detrimental (PPL increases to 25.60), confirming its critical role in capturing long-range dependencies that are otherwise lost due to the local memories' reset mechanism.

Q-K Projection. The query-key projection proves essential for performance. Its removal incurs a substantial penalty (PPL increases from 21.04 to 22.01), validating our hypothesis that it is necessary to mitigate the compression-retrieval mismatch (Challenge 2).

Stage 2 Fine-tuning. Applying Stage 2 fine-tuning further enhances model capabilities, improving both language modeling (20.86 PPL) and common-sense reasoning (40.9%). This demonstrates its effectiveness in adapting the pre-trained models for high-resolution inference.

6 Conclusion

We introduce TNT, a two-stage training framework that resolves the fundamental conflict between training efficiency and inference performance in deep memory modules. By leveraging a hierarchical memory architecture with periodic state resets, TNT enables massive context parallelism during pre-training, followed by efficient fine-tuning for high-resolution inference. Our experiments demonstrate up to a 17× speedup compared to the most accurate RNN baselines while simultaneously improving performance. TNT removes a critical scalability bottleneck, significantly improving the practicality of deep memory modules and facilitating future research to close the performance gap with Transformers.

REFERENCES

Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, James Zou, Atri Rudra, and Christopher Re. Simple linear attention language models balance the recall-throughput tradeoff. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=e93ffDcpH3.

Ali Behrouz, Zeman Li, Praneeth Kacham, Majid Daliri, Yuan Deng, Peilin Zhong, Meisam Razaviyayn, and Vahab Mirrokni. Atlas: Learning to optimally memorize the context at test time. *arXiv preprint arXiv:2505.23735*, 2025a.

Ali Behrouz, Meisam Razaviyayn, Peilin Zhong, and Vahab Mirrokni. It's all connected: A journey through test-time memorization, attentional bias, retention, and online optimization, 2025b. URL https://arxiv.org/abs/2504.13173.

Ali Behrouz, Meisam Razaviyayn, Peilin Zhong, and Vahab Mirrokni. It's all connected: A journey through test-time memorization, attentional bias, retention, and online optimization. *arXiv preprint arXiv:2504.13173*, 2025c.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. *Advances in Neural Information Processing Systems*, 38, 2025d.

Róbert Csordás, Christopher Potts, Christopher D Manning, and Atticus Geiger. Recurrent neural networks learn to store and generate sequences using non-linear representations. In *Proceedings of the 7th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pp. 248–262, 2024.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

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.

Xavier Gonzalez, Andrew Warrington, Jimmy Smith, and Scott Linderman. Towards scalable and stable parallelization of nonlinear rnns. *Advances in Neural Information Processing Systems*, 37: 5817–5849, 2024.

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

Han Guo, Songlin Yang, Tarushii Goel, Eric P Xing, Tri Dao, and Yoon Kim. Log-linear attention. *arXiv preprint arXiv:2506.04761*, 2025.

Ramin Hasani, Mathias Lechner, Tsun-Hsuan Wang, Makram Chahine, Alexander Amini, and Daniela Rus. Liquid structural state-space models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=g4OTKRKfS7R.

Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

Jerry Yao-Chieh Hu, Dennis Wu, and Han Liu. Provably optimal memory capacity for modern hopfield models: Transformer-compatible dense associative memories as spherical codes. *arXiv preprint arXiv:2410.23126*, 2024.

Jiaxi Hu, Yongqi Pan, Jusen Du, Disen Lan, Xiaqiang Tang, Qingsong Wen, Yuxuan Liang, and Weigao Sun. Comba: Improving nonlinear rnns with closed-loop control. *arXiv preprint arXiv:2506.02475*, 2025.

Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc V. Le. Transformer quality in linear time, 2022. URL https://arxiv.org/abs/2202.10447.

Kazuki Irie, Imanol Schlag, Robert Csordas, and Jurgen Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. *Advances in neural information processing systems*, 34:7703–7717, 2021.

Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/.

M. Karami and V. Mirrokni. Lattice: Learning to efficiently compress the memory, 2025.

Mahdi Karami, Ali Behrouz, Praneeth Kacham, and Vahab Mirrokni. TRELLIS: Learning to compress key-value memory in attention models. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=r61s1FNYlj.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Dmitry Krotov. Hierarchical associative memory. *arXiv preprint arXiv:2107.06446*, 2021.

Dmitry Krotov and John J Hopfield. Dense associative memory for pattern recognition. *Advances in neural information processing systems*, 29, 2016.

Xiaoyu Li, Yuanpeng Li, Yingyu Liang, Zhenmei Shi, and Zhao Song. On the expressive power of modern hopfield networks. *arXiv preprint arXiv:2412.05562*, 2024.

Yi Heng Lim, Qi Zhu, Joshua Selfridge, and Muhammad Firmansyah Kasim. Parallelizing non-linear sequential models over the sequence length. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=E34AlVLN0v.

Bo Liu, Rui Wang, Lemeng Wu, Yihao Feng, Peter Stone, and Qiang Liu. Longhorn: State space models are amortized online learners. *arXiv preprint arXiv:2407.14207*, 2024.

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

Carlo Lucibello and Marc Mézard. Exponential capacity of dense associative memories. *Physical Review Letters*, 132(7):077301, 2024.

William Merrill, Jackson Petty, and Ashish Sabharwal. The illusion of state in state-space models. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=QZgo9JZpLq.

Tsendsuren Munkhdalai and Hong Yu. Neural semantic encoders. In *Proceedings of the conference. Association for Computational Linguistics. Meeting*, volume 1, pp. 397. NIH Public Access, 2017.

Tsendsuren Munkhdalai, Alessandro Sordoni, Tong Wang, and Adam Trischler. Metalearned neural memory. *Advances in Neural Information Processing Systems*, 32, 2019.

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.

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

Bo Peng, Ruichong Zhang, Daniel Goldstein, Eric Alcaide, Haowen Hou, Janna Lu, William Merrill, Guangyu Song, Kaifeng Tan, Saiteja Utpala, et al. Rwkv-7" goose" with expressive dynamic state evolution. *arXiv preprint arXiv:2503.14456*, 2025.

Hao Peng, Jungo Kasai, Nikolaos Pappas, Dani Yogatama, Zhaofeng Wu, Lingpeng Kong, Roy Schwartz, and Noah A. Smith. ABC: Attention with bounded-memory control. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7469–7483, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.515. URL https://aclanthology.org/2022.acl-long.515/.

DL Prados and SC Kak. Neural network capacity using delta rule. *Electronics Letters*, 25(3): 197–199, 1989.

Zhen Qin, Weigao Sun, Dong Li, Xuyang Shen, Weixuan Sun, and Yiran Zhong. Various lengths, constant speed: Efficient language modeling with lightning attention. In *Forty-first International Conference on Machine Learning*, 2024.

Zihan Qiu, Zekun Wang, Bo Zheng, Zeyu Huang, Kaiyue Wen, Songlin Yang, Rui Men, Le Yu, Fei Huang, Suozhi Huang, et al. Gated attention for large language models: Non-linearity, sparsity, and attention-sink-free. *arXiv preprint arXiv:2505.06708*, 2025.

Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Lukas Gruber, Markus Holzleitner, Thomas Adler, David Kreil, Michael K Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. Hopfield networks is all you need. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=tL89RnzIiCd.

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

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

Jürgen Schmidhuber. Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets. In *ICANN'93: Proceedings of the International Conference on Artificial Neural Networks Amsterdam, The Netherlands 13–16 September 1993 3*, pp. 460–463. Springer, 1993.

Mark Schöne, Babak Rahmani, Heiner Kremer, Fabian Falck, Hitesh Ballani, and Jannes Gladrow. Implicit language models are rnns: Balancing parallelization and expressivity. *arXiv preprint arXiv:2502.07827*, 2025.

Julien Siems, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. Deltaproduct: Increasing the expressivity of deltanet through products of householders. *arXiv preprint arXiv:2502.10297*, 2025.

Jimmy T.H. Smith, Andrew Warrington, and Scott Linderman. Simplified state space layers for sequence modeling. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=Ai8Hw3AXqks.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, et al. Learning to (learn at test time): Rnns with expressive hidden states. *arXiv preprint arXiv:2407.04620*, 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.

Matteo Tiezzi, Michele Casoni, Alessandro Betti, Tommaso Guidi, Marco Gori, and Stefano Melacci. On the resurgence of recurrent models for long sequences: Survey and research opportunities in the transformer era. *arXiv preprint arXiv:2402.08132*, 2024.

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.

Ke Alexander Wang, Jiaxin Shi, and Emily B. Fox. Test-time regression: a unifying framework for designing sequence models with associative memory, 2025. URL https://arxiv.org/abs/2501.12352.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. In *International Conference on Machine Learning*, pp. 56501–56523. PMLR, 2024a.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. *Advances in neural information processing systems*, 37:115491–115522, 2024b.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024c.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. *arXiv preprint arXiv:2412.06464*, 2025.

Tianyuan Zhang, Sai Bi, Yicong Hong, Kai Zhang, Fujun Luan, Songlin Yang, Kalyan Sunkavalli, William T Freeman, and Hao Tan. Test-time training done right. *arXiv preprint arXiv:2505.23884*, 2025.

Yu Zhang, Songlin Yang, Rui-Jie Zhu, Yue Zhang, Leyang Cui, Yiqiao Wang, Bolun Wang, Freda Shi, Bailin Wang, Wei Bi, et al. Gated slot attention for efficient linear-time sequence modeling. *Advances in Neural Information Processing Systems*, 37:116870–116898, 2024.

A LLM USAGE

We acknowledge the use of a large language model (LLM) solely for improving the linguistic quality and clarity of this manuscript. The model was not used for ideation, research methodology, or generating the scientific content presented in this work.

B ADDITIONAL RELATED WORK

Modern Linear Recurrent Neural Networks Due to the quadratic complexity of transformers, recently developing alternative architectures have gained attention, which led to the development of efficient recurrent alternatives (Tiezzi et al., 2024). Initial advancements in this domain, starts with models such as RetNet (Sun et al., 2023), RWKV (Peng et al., 2023), and S5 (Smith et al., 2023), which employed data-independent transition matrices coupled with Hebbian-like update mechanisms. Subsequently, a second generation of models emerged, incorporating input-dependent parameters within these linear architectures (e.g., linear RNNs (Hasani et al., 2023; Smith et al., 2023), RWKV6 (Peng et al., 2024)). These models also explored more expressive memory updating rules, notably those based on the delta rule (Peng et al., 2025; Schlag et al., 2021; Yang et al., 2024b;a; Liu et al., 2024). Further evolution in this line of research has extended these memory architectures to deeper models, while concurrently utilizing delta-rule-like update mechanisms (Sun et al., 2024) or data-dependent momentum-based update rules with forget gating (Behrouz et al., 2025d). More recently, to augment the performance of delta-rule-based sequential models, Siems et al. (2025) have proposed the application of multiple gradient descent updates per token, thereby yielding more expressive sequence models, particularly in state tracking tasks. In addition to the above fast linear recurrent sequence models, several studies have focused on RNNs with non-linear recurrence (Behrouz et al., 2025d;b;a; Csordás et al., 2024; Merrill et al., 2024; Lim et al., 2024; Schöne et al., 2025; Karami & Mirrokni, 2025; Gonzalez et al., 2024), and how their training can be faster (Gonzalez et al., 2024; Lim et al., 2024; Schöne et al., 2025).

Fast Weight Programs The conceptualization of linear layers as key-value associative memory systems can be traced back to Hopfield networks (Hopfield, 1982). This concept was subsequently developed in the context of fast weight programmers, wherein dynamic fast programs are integrated into recurrent neural networks to serve as writable memory stores (Schlag et al., 2021; Schmidhuber, 1992; 1993). Among the learning paradigms for such systems, Hebbian learning (Hebb, 2005) and the delta rule (Prados & Kak, 1989) have emerged as the most prominent. Both learning rules have been the subject of extensive investigation within the existing literature (Munkhdalai & Yu, 2017; Schmidhuber, 1992; Munkhdalai et al., 2019; Schlag et al., 2021; Irie et al., 2021; Yang et al., 2024b;a).

Hopfield Networks Our formulation is architecturally founded upon the broad concept of associative memory, wherein the primary objective is to learn an underlying mapping between keys and values. Seminal work by Hopfield (1982) on Hopfield Networks introduced one of the earliest neural architectures explicitly based on associative memory, defining it through the minimization of an energy function for storing key-value pairs. Although traditional Hopfield networks have seen diminished applicability in recent years, primarily due to constraints in vector-valued memory capacity and the nature of their energy function, several contemporary studies have focused on enhancing their capacity through various methodologies. These include efforts by Krotov (2021), Li et al. (2024), and Krotov & Hopfield (2016). Notably, extensions to the energy function of these models, often incorporating exponential kernels, have been explored (Krotov & Hopfield, 2016; Lucibello & Mézard, 2024). Furthermore, the relationship between these modernized Hopfield networks and Transformer architectures has been a subject of recent investigation (Ramsauer et al., 2021; Hu et al., 2024).

C CONNECTION OF QK-PROJECTION WITH MEMORY BOUNDED TRANSFORMERS

Revisiting our QK-Projection retrieval process, we first project the query vector into the space of stored keys and then retrieve its corresponding value by a forward pass on the memory module. In particular, given query q_t and stored keys of $\{k_i\}_{i=1}^{t}$, the output corresponds to query q_t can be

calcualted as:

$$o_t = f\left(W_t, \sum_{\tau=1}^{t} \frac{k_\tau k_\tau^\top}{\|k_\tau\|^2} q_t\right). \tag{8}$$

Given a normalization of keys, i.e., $\|k_\tau\|_2 = 1$, this formulation, can be re-written as:

$$o_t = f\left(W_t, \sum_{\tau=1}^{t} k_\tau k_\tau^\top q_t\right), \tag{9}$$

in which the second element, $\sum_{\tau=1}^{t} k_\tau k_\tau^\top q_t$, is equivalent to a simple forward pass for query q_t over a linear memory module of $\mathcal{M}'_t = \sum_{\tau=1}^{t} k_\tau k_\tau^\top$ with recurrence of:

$$\mathcal{M}'_t = \mathcal{M}'_{t-1} + k_t k_t^\top. \tag{10}$$

Such formulation of QK-Projection can remind us the two-pass process of memory bounded Transformers (Peng et al., 2022; Zhang et al., 2024; Karami et al., 2025), where in the simple linear attention form (Peng et al., 2022), the retrieval process can be written as:

$$W_t = W_{t-1} + \varphi_t v_t^\top, \tag{11}$$

$$o_t = W_t \, \texttt{softmax}\left(\left(\sum_{\tau=1}^{t} k_\varphi \varphi_\tau^\top\right) q_t\right). \tag{12}$$

Comparing with above two-pass process of ABC (Peng et al., 2022), our QK-projection method is applicable to both deep and linear memory. Furthermore, parameters of φ_t as well as k_t are tied and so the learning process is considerably easier, making the model more adaptable to new data/tasks. Moreover, when employ this projection in the local memory, we only do the summations starting from the "reset" state of the memory rather than starting from $\tau = 1$.

D Efficient Implementation of QK-Projection

This section details the efficient, parallelizable implementation of the QK-Projection mechanism. We demonstrate that this projection can be integrated into the chunkwise training paradigm without introducing sequential bottlenecks, thereby preserving the training efficiency of the TNT architecture.

The QK-Projection relies on a projection matrix, $\mathcal{M}_t$, which accumulates the outer products of keys within the current local memory shard (length S_L). Assuming normalized keys ($\|k_\tau\| = 1$), this matrix is defined by the following recurrence:

$$\mathcal{M}_t = \begin{cases} k_t k_t^\top & \text{if } t \equiv 1 \pmod{S_L} \\ \mathcal{M}_{t-1} + k_t k_t^\top & \text{otherwise} \end{cases}$$

This rule ensures that the projection state $\mathcal{M}_t$ is reset at the beginning of each shard, mirroring the reset of the local memory state W_t. The local memory retrieval is then computed as $f(W_t, \mathcal{M}_t q_t)$.

Chunkwise Parallel Computation. To maintain training efficiency, $\mathcal{M}_t$ is computed in a chunk-parallel manner. For any time step t within a chunk of size C_L, the projection matrix can be decomposed into two components:

$$\mathcal{M}_t = \underbrace{\mathcal{M}_{\xi(t,C_L)-1}}_{\text{Carry-over State}} + \underbrace{\sum_{\tau=\xi(t,C_L)}^{t} k_\tau k_\tau^\top}_{\text{Intra-chunk Sum}}$$

The first term is the final state from the previous chunk, which is carried over. The second term, the intra-chunk sum, is computed efficiently for all steps in the chunk simultaneously using a parallel prefix sum (scan) operation over the sequence of outer products $\{k_\tau k_\tau^\top\}$.

This implementation preserves end-to-end parallelism. The state passed between chunks is a single, constant-size matrix ($d \times d$), incurring minimal overhead. The periodic reset is handled by re-initializing this carry-over state at shard boundaries. Thus, QK-Projection enhances the model's retrieval mechanism without compromising the training efficiency fundamental to the TNT architecture.

E TNT APPLICABILITY

In this paper, we have focused on showing the effectiveness and efficiency of TNT and so for the sake of clarity, we use a simple memory module that optimizes its inner-loop with gradient descent. However, TNT recipes are applicable to different deep memory and non-linear architectures. For example, one can adapt the gating formulation in Titans (Behrouz et al., 2025d) or Mamba2 (Dao & Gu, 2024) for each of the local memories as well as the global memory. Another potential exploration is to incorporate closed feedback loop in the objective of the inner-loop as it has done in Hu et al. (2025). Similarly, one can employ more expressive optimizers as the inner-loop optimizers such as gradient descent with momentum, AdamW (Kingma & Ba, 2014), or muon (Jordan et al., 2024) as it has been done by Behrouz et al. (2025a); Zhang et al. (2025). While exploring all these combinations with TNT is a promising direction, for the sake of clarity and space constraint, we leave them for future studies.

F TNT GENERALIZATION FORMULATION

The TNT architecture can be generalized to a hierarchical system comprising one global memory and N parallel local memories. This allows the model to capture information at multiple timescales and resolutions simultaneously. Each local memory, denoted by $W^{(i)}$ for $i \in \{1, \dots, N\}$, operates with its own distinct chunk size C_{L_i}, shard length S_{L_i}, and learnable initial state $W^{(i)}_{\text{init}}$.

F.1 GENERALIZED MEMORY UPDATE

The update rules are extended as follows: the single global memory evolves sequentially, while the N local memories are updated in parallel, each with its independent schedule.

Global Memory Update. The global memory state V is updated sequentially with a large chunk size C_G, identical to the base case.

$$V_{(k+1)C_G} \leftarrow V_{kC_G} - \sum_{t=kC_G}^{(k+1)C_G} \eta_t \nabla_V \mathcal{L}\left(f(V_{kC_G}, k_t), v_t\right) \tag{13}$$

N-Local Memories Update. Each of the N local memories $W^{(i)}$ operates in parallel. The state of each memory is reset periodically according to its specific shard length S_{L_i}, enabling multi-resolution context parallelism.

$$W^{(i)}_t \leftarrow \begin{cases} W^{(i)}_{\text{init}} & \text{if } 0 \equiv t \pmod{S_{L_i}} \\ W^{(i)}_{t-1} - \sum_{\tau=\xi(t,C_{L_i})}^{t} \eta_\tau \nabla_{W^{(i)}} \mathcal{L}\left(f(W^{(i)}_{\xi(t,C_{L_i})}, k_\tau), v_\tau\right) & \text{Otherwise} \end{cases} \tag{14}$$

where $i = 1, \dots, N$.

F.2 GENERALIZED MEMORY RETRIEVAL

During retrieval, the final output is a composition of the outputs from the global memory and all N local memories. The global memory uses the raw query q_t, while each local memory uses a Q-K projection tailored to its specific context window, determined by its chunk size C_{L_i}.

TNT Generalized Retrieval Rule. The hierarchical memory is retrieved by summing the contributions from each memory module.

$$o_t = f(V_{\xi(t,C_G)}, q_t) + \sum_{i=1}^{N} f\left(W^{(i)}_t, \sum_{\tau=\xi(t,C_{L_i})}^{t} \frac{k_\tau k_\tau^\top}{\|k_\tau\|^2} q_t\right) \tag{15}$$

This formulation allows the network to integrate long-range dependencies from the global memory with fine-grained, parallel-processed information from a diverse set of local memories, each specializing in different temporal patterns.

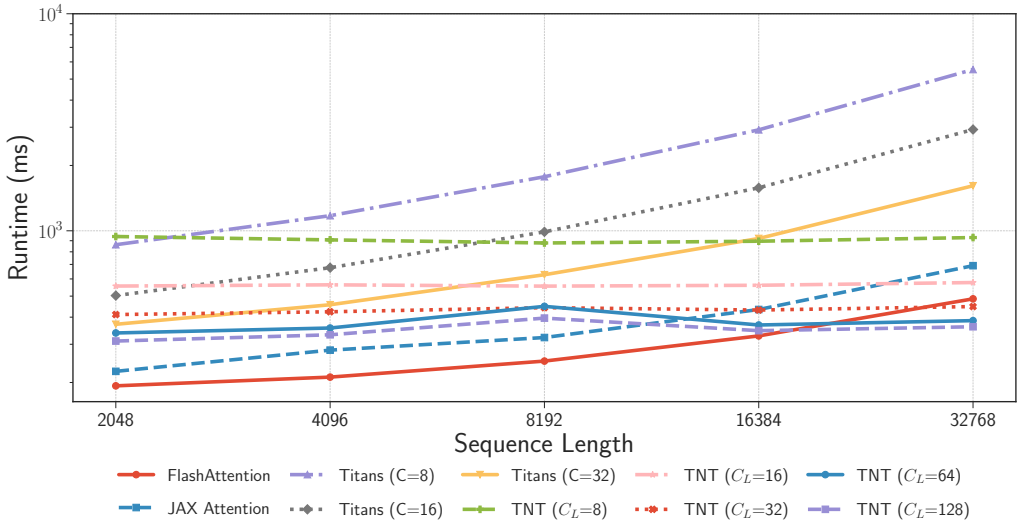

Figure 5: Runtime comparison of different models and implementations across varying sequence lengths, with the number of tokens per batch fixed at 0.5M.

Table 4: Training time for 150M parameter models trained on 10B tokens. For TNT models, the global chunksize is fixed at $C_G = 2048$, and C_L denotes the set of chunksizes for the local memory modules.

Model	C or C_L	**Training Time (hrs)**
150M params / 10B tokens		
Transformer (w/o gating)	-	0.80
Transformer (w gating)	-	0.82
TTT (2024)	256	1.69
Titans (2025d)	256	1.99
Titans	8	8.44
TNT Stage 1		
TNT Stage 1	{8}	3.06
	{8,16}	4.24
	{4,8,16}	5.00
	{4,8,16,32}	5.55
TNT Stage 2		
TNT Stage 2	{1}	0.15
	{2,4}	0.23
	{2,4,8}	0.26
	{2,4,8,16}	0.46

G Summary of Revisions and Additional Experiments

We thank the reviewers for their insightful feedback. In response to their comments, we have conducted several additional experiments and made revisions, which we summarize below.

1. **Scaling of Local Memory Depth (N).** In response to Reviewer g52o's query about the benefit of $N > 4$ local modules, we experimented with increasing the local memory depth from $N = 1$ to $N = 9$. As shown in Table 5, performance steadily improves with depth: average perplexity decreases from 24.10 to 22.97, and downstream accuracy increases from 40.6 to 41.6. This demonstrates that deeper local memory continuously improves performance, highlighting the potential of the TNT framework.
2. **Impact of Global Memory Chunk Size (C_G).** To address questions from Reviewers g52o and NKDU, we evaluated the effect of varying the global chunk size C_G from 32 to 8192. The results are in Table 6. We observed that while a smaller $C_G = 128$ yields a slightly better average perplexity (23.21) than our original $C_G = 2048$ (23.23), the improvement is marginal. We find this minor gain does not justify the additional computational cost, as smaller chunks reduce opportunities for parallelization. Therefore, our original setup provides a strong balance between performance and computational efficiency.
3. **Impact of Local Memory Chunk Size (C_L).** Per Reviewer g52o's request, we fixed the global memory ($C_G = 2048$) and varied the local memory chunk size C_L. As shown in Table 7, our results indicate that smaller local memory chunk sizes generally yield improved performance.
4. **Optimal Local Memory Configurations.** To address Reviewer g52o's question about selecting optimal chunk sizes for multi-local setups, we experimented with different memory hierarchies (Table 8). We found that the optimal configuration utilizes heterogeneous local memories with different resolutions. As a practical heuristic, we suggest starting with an exponential progression (e.g., $2^1, 2^2, \ldots, 2^N$ or $4^1, 4^2, \ldots, 4^N$ for N local memories). Based on our experience, this rule of thumb generally yields strong performance.
5. **Additional Baselines and Table Clarity.** In response to Reviewer 2mkd's concern about the number of baselines and clarity in Table 2, we have revised the table and added two modern RNN baselines: Deltanet (Yang et al., 2024c) and Gated Deltanet (Yang et al., 2025). As shown in the updated table, TNT (23.09 PPL) significantly outperforms both Deltanet (25.56 PPL) and Gated Deltanet (24.40 PPL), further demonstrating the effectiveness of our framework.

Table 5: The scaling behavior of local memory depth (N) on model performance. The depth is varied from $N = 1$ to $N = 9$. A clear improvement in performance is observed as the depth increases.

Model	C_G	C or C_L	C4 ppl ↓	FineWeb ppl ↓	PG19 ppl ↓	Avg. ppl ↓	PIQA acc ↑	Hella. acc ↑	ARC-e acc ↑	CSQA acc ↑	Avg. acc ↑
150M params / 10B tokens											
TNT Stage 1	2048	{8}	21.04	21.01	30.24	24.10	61.8	32.8	37.4	30.3	40.6
	2048	{8,16}	20.74	20.73	29.94	23.80	63.5	32.4	37.4	30.6	41.0
	2048	{4,8,16}	20.47	20.43	29.43	23.44	62.9	32.4	36.4	28.9	40.2
	2048	{4,8,16,32}	20.15	20.17	29.08	23.13	63.2	32.0	36.7	30.3	40.6
	2048	{4,8,16,32,64}	20.13	20.25	29.63	23.34	63.0	32.3	36.8	32.3	41.1
	2048	{4,8,16,32,64,128}	20.08	20.22	29.66	23.32	64.0	32.6	37.3	32.6	41.6
	2048	{4,8,16,32,64,128,256}	19.96	20.10	29.56	23.21	63.2	32.6	37.5	32.6	41.5
	2048	{4,8,16,32,64,128,256,512}	19.84	19.98	29.27	23.03	62.7	32.5	36.5	32.5	41.0
	2048	{4,8,16,32,64,128,256,512,1024}	19.74	19.92	29.24	22.97	63.7	32.9	37.0	32.9	41.6

Table 6: Impact of global memory chunk size on the TNT model's performance. This is evaluated at four fixed local memory sizes ($C_L = \{4, 8, 16, 32\}$). No clear correlation was observed between global chunk size and performance.

Model	C_G	C or C_L	C4 ppl ↓	FineWeb ppl ↓	PG19 ppl ↓	Avg. ppl ↓	PIQA acc ↑	Hella. acc ↑	ARC-e acc ↑	CSQA acc ↑	Avg. acc ↑
150M params / 10B tokens											
TNT Stage 1	32	{4,8,16,32}	20.28	20.31	29.26	23.28	63.1	32.8	37.6	29.6	40.8
	128	{4,8,16,32}	20.27	20.29	29.16	23.24	63.4	33.2	37.3	30.3	41.1
	256	{4,8,16,32}	20.22	20.25	29.17	23.21	62.9	32.9	38.4	30.0	41.0
	512	{4,8,16,32}	20.26	20.28	29.26	23.27	64.1	32.8	36.8	29.6	40.8
	1024	{4,8,16,32}	20.25	20.31	29.34	23.30	64.0	32.4	36.8	29.6	40.7
	2048	{4,8,16,32}	20.20	20.23	29.26	23.23	63.4	32.7	37.0	29.6	40.7
	4096	{4,8,16,32}	20.26	20.30	29.25	23.27	63.2	32.4	37.9	29.9	40.8
	8192	{4,8,16,32}	20.29	20.30	29.24	23.28	63.1	32.1	37.3	28.7	40.3

Table 7: Impact of local memory chunk size on the TNT model's performance. We use a fixed global memory ($C_G = 2048$) and vary the chunksize size C_L of a single local memory. Smaller local memory sizes are shown to yield improved performance.

Model	C_G	C or C_L	C4 ppl ↓	FineWeb ppl ↓	PG19 ppl ↓	Avg. ppl ↓	PIQA acc ↑	Hella. acc ↑	ARC-e acc ↑	CSQA acc ↑	Avg. acc ↑
150M params / 10B tokens											
TNT Stage 1	2048	{2}	20.86	20.90	30.16	23.97	63.3	32.4	36.2	28.3	40.1
	2048	{4}	21.02	21.04	30.39	24.15	63.9	32.5	37.5	29.9	41.0
	2048	{8}	21.04	21.01	30.24	24.10	61.8	32.8	37.4	30.3	40.6
	2048	{16}	21.08	21.03	30.28	24.13	63.3	31.5	36.4	30.3	40.4
	2048	{32}	21.06	21.04	30.24	24.11	62.8	32.0	36.4	27.6	39.7
	2048	{64}	21.17	21.15	30.48	24.26	62.8	32.8	36.5	29.3	40.4
	2048	{128}	21.39	21.39	30.80	24.53	63.5	31.8	35.6	28.7	39.9
	2048	{256}	21.53	21.57	31.12	24.74	62.6	31.9	38.0	30.1	40.6
	2048	{512}	21.74	21.81	31.45	25.00	62.3	31.4	36.1	28.0	39.4
	2048	{1024}	22.18	22.24	32.21	25.54	61.9	30.9	36.6	28.9	39.6
	2048	{2048}	22.79	22.83	33.06	26.23	62.2	31.5	35.8	26.7	39.1

Table 8: This analysis investigates different local memory configurations. We observe that the best performance is achieved with an optimal configuration of heterogeneous local memories (i.e., having different resolutions), which aligns with the core hypothesis of TNT.

Model	C_G	C or C_L ppl ↓	C4 ppl ↓	FineWeb ppl ↓	PG19 ppl ↓	Avg. acc ↑	PIQA acc ↑	Hella. acc ↑	ARC-e acc ↑	CSQA acc ↑	Avg.
150M params / 10B tokens											
TNT Stage 1	2048	{4,8,16}	20.47	20.43	29.43	23.44	62.9	32.4	36.4	28.9	40.2
	2048	{4,8,32}	20.36	20.33	29.37	23.35	62.6	32.6	36.9	28.2	40.1
	2048	{4,8,64}	20.38	20.39	29.45	23.40	63.1	32.2	36.8	29.4	40.4
	2048	{4,8,128}	20.46	20.50	29.51	23.49	62.7	31.7	35.5	27.4	39.3
	2048	{4,16,64}	20.37	20.36	29.35	23.36	62.7	32.9	37.3	29.5	40.6
	2048	{4,16,128}	20.43	20.44	29.44	23.44	63.5	32.5	38.1	29.7	41.0
	2048	{4,16,256}	20.54	20.56	29.67	23.59	63.2	32.5	36.2	28.1	40.0
	2048	{4,16,512}	20.60	20.62	29.79	23.67	62.7	32.0	35.8	27.3	39.5