MODULAR TTT: RETHINKING TEST-TIME TRAINING AS COMPOSABLE MODULES

Anonymous authors
Paper under double-blind review

ABSTRACT

Test-time training (TTT) views sequence modeling as an online learning problem in which fast weights are updated by an internal learning rule. Despite the growing number of TTT variants, existing approaches typically hard-code each variant separately, which makes it difficult to design new TTT methods and to isolate the role of each component. To address this, we propose Modular TTT, a framework that represents the inner learner as a directed acyclic graph and exposes the fast-weight network, loss function, learning rate, weight decay, and normalization as explicit design dimensions. Modular TTT automatically composes primitive-level train-view forward, train-view backward, and causal query-view rules into the full graph-level TTT computation, including the fast-weight state transition. Using Modular TTT, we systematically ablate the components of TTT and find that small learning-rate initialization, weight decay, and a single-layer nonlinearity improve performance, while MSE and inner-product losses perform similarly. Deeper fast-weight networks and normalization tend to hurt performance because they induce excessively large activations, while residual connections and gating provide little measurable benefit. Guided by these findings, we train the best resulting variant as 410M- and 1.45B-parameter models on 100B tokens, and observe training loss and benchmark performance comparable to Gated DeltaNet.

1 INTRODUCTION

Sequence modeling is a fundamental problem across many domains, and a broad range of architectures have been developed for it, including attention (Vaswani et al., 2017), recurrent neural networks (RNNs) (Hochreiter & Schmidhuber, 1997; Chung et al., 2014), structured state space models (SSMs) (Gu et al., 2022; Gu & Dao, 2024), and linear attention (Katharopoulos et al., 2020). Test-time training (TTT) (Sun et al., 2024) offers a different perspective by casting sequence modeling as an online learning process, in which fast weights (Schmidhuber, 1992; Ba et al., 2016) are updated by an inner learning rule as the model processes a sequence. In contrast to methods that restrict state updates to fixed recurrences or explicit caches, TTT treats state update as inner learning, expanding the design space of sequence models.

Despite the growing number of TTT variants, the design space of TTT remains poorly understood. Existing approaches typically hard-code each variant separately, which creates two practical difficulties. First, it makes it difficult to develop new TTT methods in a systematic way, since modifying a variant often changes several components at once. Second, it obscures the role of individual components. In practice, the fast-weight network, loss function, learning rate, weight decay, and normalization are often changed together, making it difficult to determine which design choices are responsible for the observed behavior. As a result, while many TTT variants have been proposed (Sun et al., 2024; Zhang et al., 2025; Behrouz et al., 2025d; Tandon et al., 2025), their common structure and the contribution of each module remain unclear.

In this work, we present Modular TTT, a framework that represents the inner learner as a directed acyclic graph and factorizes TTT into explicit, controllable design dimensions. In Modular TTT, the fast-weight network, loss function, learning rate, weight decay, and normalization are treated as modular components that can be specified and varied independently. Given a learner DAG built from registered primitives, Modular TTT automatically composes their train-view forward, train-view backward, and causal query-view rules into the full graph-level TTT computation, including the fast-weight state transition. Automatic differentiation provides the local backward signals of the

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

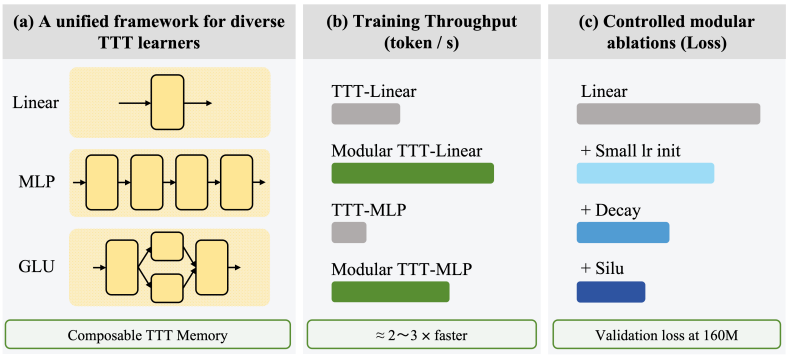

Figure 1: **(a)** Linear, MLP, and gated learners viewed as composable TTT memory forms. **(b)** Training throughput vs. official TTT at 160M scale. **(c)** Ablations over key design choices: small lr init, decay, and SiLU each reduce validation loss.

train-view loss; the causal query readout and fast-weight state transition are defined by the registered primitive rules. This removes the need to hand-derive a new global update rule for every TTT variant, and turns TTT design from a collection of custom implementations into a modular space that can be explored and analyzed systematically. Its optimized implementation also achieves multi-fold training-throughput improvements over the official implementations of the same TTT topologies. Figure 1 gives an overview of the modular formulation, speedup, and ablation results.

Building on Modular TTT, we conduct systematic ablations over the key components of TTT and obtain a clearer empirical picture of its design space. We find that small learning-rate initialization, weight decay, and a single-layer nonlinearity provide consistent gains, while MSE and inner-product losses perform similarly. Deeper fast-weight networks and normalization tend to degrade performance, likely due to excessively large activations, and residual connections and gating yield little measurable benefit. Guided by these findings, we train the selected variant at 410M/1.45B scale for 100B tokens and observe loss and benchmark performance comparable to Gated DeltaNet (GDN) (Yang et al., 2025).

To summarize, our contributions are threefold. First, we introduce Modular TTT, a unified framework that represents the TTT inner learner as a directed acyclic graph and treats the fast-weight network, loss function, learning rate, weight decay, and normalization as modular design dimensions. Second, we use this framework to systematically analyze the role of major TTT components, identifying which design choices provide consistent gains and which offer little benefit under the evaluated settings. Third, guided by this analysis, we train the best resulting variant at 410M and 1.45B scale, and show that it achieves training loss and benchmark performance comparable to GDN.

2 Related Work

Existing work on efficient sequence modeling can be broadly grouped into four families: recurrent models, linear attention, state space models, and test-time training. All of these approaches aim to preserve long-context modeling ability while reducing the computation and memory costs that grow with sequence length, although they differ substantially in their implementations.

RNN Recurrent models compress historical context into a fixed-size hidden state, and enhance expressiveness through mechanisms such as gating and nonlinear transformations (Hochreiter & Schmidhuber, 1997; Chung et al., 2014). The core idea is to accumulate contextual information through explicit state transition rules in a constant-size state, so that the cost of each inference step is decoupled from the history length. Recent work shows that removing nonlinearities from RNNs can improve efficiency while still achieving competitive performance (Martin & Cundy, 2018; Orvieto et al., 2023; Qin et al., 2023; Peng et al., 2023).

Linear Attention Linear attention rewrites attention as inner products of feature maps and exploits the associativity of matrix multiplication to aggregate key-value pairs first, thereby avoiding the explicit construction of a quadratic attention matrix (Katharopoulos et al., 2020). In the causal setting, reformulating token-wise recurrences into chunkwise recurrent computation (Yang et al., 2024a) enables more efficient GPU utilization while preserving linear complexity with respect to sequence length. Subsequent work further extends this direction along several axes, including feature maps

(Choromanski et al., 2021), decay mechanisms (Sun et al., 2023; Qin et al., 2024a; Yang et al., 2024a; Qin et al., 2024b; Yang et al., 2024b; 2025; Peng et al., 2025) and normalization schemes.

State Space Models State space models start from continuous- or discrete-time state equations and model long-range dependencies through structured state transitions. Representative works such as S4 (Gu et al., 2022) show that, with specialized initialization and structured parameterization, state space models can achieve strong empirical performance. More recent methods (Gu & Dao, 2024; Dao & Gu, 2024) further emphasize content-dependent decay and hardware-friendly parallel scan algorithms, making this line a major direction for efficient sequence modeling.

Test-Time Training Unlike the approaches above, test-time training (Krause et al., 2018; Sun et al., 2024) formulates sequence modeling as an online learning process, in which the hidden state is no longer a fixed-form vector state, but the fast weights (Schmidhuber, 1992; Ba et al., 2016; Schlag et al., 2021; Irie et al., 2021; 2022) of a learnable model, and the corresponding state update is given by one or more optimization steps on a self-supervised objective. With suitable choices of the online learning loss and other components, the TTT perspective can instantiate a variety of Linear Attention, Linear RNN, and SSM variants. Recent work from this perspective has introduced instances such as TTT-Linear and TTT-MLP (Sun et al., 2024), LaCT (Zhang et al., 2025), and Titans (Behrouz et al., 2025d), demonstrating the potential of internal learners for long-context modeling. Subsequent work has further explored neural memory optimization (Behrouz et al., 2025a), broader connections between TTT, retention, and online optimization (Behrouz et al., 2025b;c), and extensions to vision (Han et al., 2025). However, naive TTT implementations typically rely on token-wise recurrent updates and therefore cannot fully exploit GPU parallelism. As in linear attention and SSMs, existing work often adopts chunkwise TTT implementations (Zhang et al., 2025; Li et al., 2025) to achieve a better balance between computational efficiency and empirical performance.

3 METHOD

3.1 ANALYZING THE COMPUTATION OF TTT

Previous TTT methods typically hard-code both the forward and backward computations. However, we observe that this process can be automated. Consider an m-layer TTT module with nonlinear activation functions $f^{(i)}$, where $i \in \{1, \dots, m\}$ indexes layers inside the TTT inner learner. Given an input $\mathbf{K} \in \mathbb{R}^{n \times d}$, where n denotes the sequence length and d denotes the feature dimension, we first perform a forward pass, which we refer to as the *train-view forward*:

$$\begin{aligned} &\mathbf{K}^{(0)} = \mathbf{K}, \quad d^{(0)} = d, \qquad \mathbf{U}^{(i)} = \mathbf{K}^{(i-1)}\mathbf{W}^{(i)}, \quad \mathbf{W}^{(i)} \in \mathbb{R}^{d^{(i-1)} \times d^{(i)}}, \\ &\mathbf{K}^{(i)} = f^{(i)}(\mathbf{U}^{(i)}) \in \mathbb{R}^{n \times d^{(i)}}, \qquad \hat{\mathbf{V}} = \mathbf{K}^{(m)}. \end{aligned} \tag{1}$$

Given a target $\mathbf{V} \in \mathbb{R}^{n \times d^{(m)}}$ and a loss function $\mathcal{L}$, we then perform a backward pass, which we refer to as the *train-view backward*:

$$\begin{aligned} l = \mathcal{L}(\hat{\mathbf{V}}, \mathbf{V}), \quad \mathbf{dK}^{(m)} = \frac{\partial l}{\partial \hat{\mathbf{V}}}, \qquad \mathbf{dU}^{(i)} = \mathbf{dK}^{(i)} \odot \left[f^{(i)}(\mathbf{U}^{(i)}) \right]', \\ \mathbf{dK}^{(i-1)} = \mathbf{dU}^{(i)}[\mathbf{W}^{(i)}]^\top, \qquad \mathbf{dW}^{(i)} = [\mathbf{K}^{(i-1)}]^\top \mathbf{dU}^{(i)}. \end{aligned} \tag{2}$$

Finally, we perform another forward pass using a query $\mathbf{Q} \in \mathbb{R}^{n \times d}$, which we refer to as the *query-view forward*, where $\mathrm{Tril}(\mathbf{X})$ denotes the lower-triangular part of $\mathbf{X}$ (i.e., $[\mathrm{Tril}(\mathbf{X})]_{ij} = X_{ij}$ for $i \geq j$ and 0 otherwise):

$$\begin{aligned} &\mathbf{Q}^{(0)} = \mathbf{Q}, \qquad &&\mathbf{R}^{(i)} = \mathbf{Q}^{(i-1)}\mathbf{W}^{(i)} - \mathrm{Tril}\Big(\mathbf{Q}^{(i-1)}[\mathbf{K}^{(i-1)}]^\top\Big)\,\mathbf{dU}^{(i)}, \\ &\mathbf{W}^{(i)} = \mathbf{W}^{(i)} - \mathbf{dW}^{(i)}, \quad \mathbf{Q}^{(i)} = f^{(i)}(\mathbf{R}^{(i)}), \qquad &&\mathbf{O} = \mathbf{Q}^{(m)}. \end{aligned} \tag{3}$$

This query-view form follows the causal dual update used in TTT (Sun et al., 2024). This example highlights several key observations:

- The train-view forward defines the computation performed by the TTT module.
- The train-view backward provides the activations required by the query-view forward.
- The query-view forward performs the actual forward computation of the TTT module.
- By defining the train-view and query-view operations for a single layer, we can derive the corresponding train-view and query-view operations for a multi-layer network.

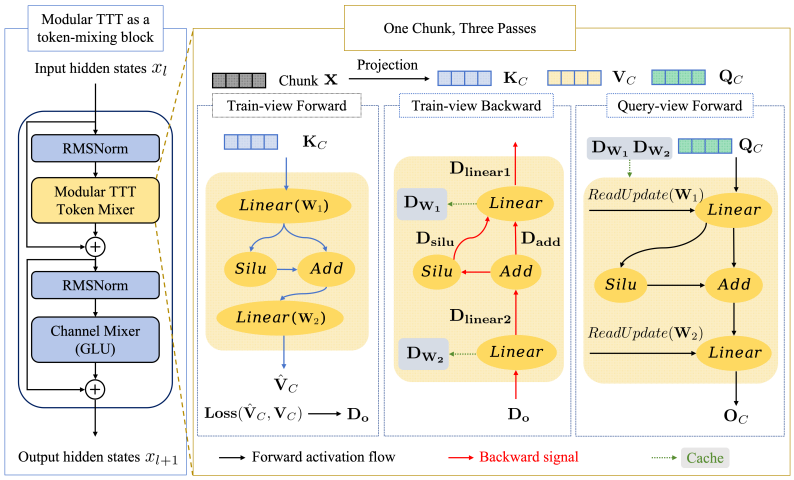

Figure 2: Modular TTT: graph-structured memory with a shared three-pass computation. **Left:** the Modular TTT layer within a standard pre-norm backbone. **Right:** one chunk processed by train-view forward, train-view backward, and query-view forward over the graph memory.

3.2 MODULAR TTT

As in the example above, a general TTT module can be viewed as a computation graph. By defining the `train-view forward`, `train-view backward`, and `query-view forward` rules for each primitive in the graph, we can automatically compose the full graph-level TTT computation, without hand-deriving a new global update rule for every specific variant.

More concretely, we represent a TTT module as a directed acyclic graph $G = (\mathcal{V}, \mathcal{E})$. Each node in the graph corresponds to a primitive operation, such as a linear map, an elementwise nonlinearity, a residual addition, or a normalization layer, while each edge represents a tensor dependency.

Let the primitive at node j be denoted by ϕ_j, with inputs from its parent set $\mathrm{pa}(j)$, parameters denoted by θ_j and $\mathbf{h}_j^{\text{train}}$ denotes the input and output of a node. In the train-view, we execute the forward pass of each node in topological order:

$$\mathbf{h}_j^{\text{train}} = \phi_j^{\text{train}}\big(\{\mathbf{h}_i^{\text{train}} : i \in \mathrm{pa}(j)\}; \theta_j\big), \tag{4}$$

which produces the train-view output $\hat{\mathbf{V}}$, and we compute the inner learning loss $l = \mathcal{L}(\hat{\mathbf{V}}, \mathbf{V})$. Next, we execute the train-view backward in reverse topological order to obtain intermediate gradients, local parameter updates, and backward signals to parent nodes. Denoting $\bar{\mathbf{h}}_j^{\text{train}} = \partial l / \partial \mathbf{h}_j^{\text{train}}$, the backward computation at node j can be written as

$$\Big(\{\bar{\mathbf{h}}_i^{\text{train}}\}_{i\in\mathrm{pa}(j)}, \Delta\theta_j\Big) = \phi_j^{\text{train-bwd}}\big(\bar{\mathbf{h}}_j^{\text{train}}, \mathbf{h}_j^{\text{train}}, \{\mathbf{h}_i^{\text{train}}\}_{i\in\mathrm{pa}(j)}, \theta_j\big), \tag{5}$$

where $\Delta\theta_j$ denotes the parameter update induced by the inner learning rule. The train-view activations and gradients, represented by $\{\mathbf{h}_i^{\text{train}}\}_{i\in\mathrm{pa}(j)}$ and $\bar{\mathbf{h}}_j^{\text{train}}$, are retained for the subsequent query-view computation. The multi-layer MLP example above is precisely a special case of this general procedure on a computation graph.

After obtaining the train-view activations, backpropagated gradients, and parameter updates, we perform the query-view forward on a query input $\mathbf{Q}$. Unlike an ordinary forward pass, the query-view depends not only on the current input, but also on the train-view activations, gradients, and parameter updates produced by the train-view. For each node, we define the query-view computation as

$$\mathbf{h}_j^{\text{query}} = \phi_j^{\text{query}}\big(\{\mathbf{h}_i^{\text{query}} : i \in \mathrm{pa}(j)\}, \{\mathbf{h}_i^{\text{train}} : i \in \mathrm{pa}(j)\}, \bar{\mathbf{h}}_j^{\text{train}}, \theta_j, \Delta\theta_j\big). \tag{6}$$

Automatic differentiation operates on an already specified computation graph. Applied to the train-view loss, it produces $\mathbf{d}\hat{\mathbf{V}}$ and the local backward signals propagated through the learner graph. The causal query-view readout and the fast-weight state transition require separate primitive-level specifications. Modular TTT registers these rules and composes them over the learner DAG. Executing the query-view forward in topological order then produces the final output $\hat{\mathbf{O}}$ of the TTT module.

This modular formulation provides two direct benefits. First, it significantly reduces the cost of constructing new TTT variants. Rather than manually deriving custom forward and backward rules for every new architecture, we only need to reorganize or replace local primitives in the graph.

Algorithm 1 Modular TTT

Input: key $\mathbf{K}$, query $\mathbf{Q}$, target $\mathbf{V}$, topological order τ, output node o, loss function $\mathcal{L}$, train-view operators $\{\phi_j^{\text{train}}\}_{j\in\mathcal{V}}$, train-view backward operators $\{\phi_j^{\text{train}-\text{bwd}}\}_{j\in\mathcal{V}}$, query-view operators $\{\phi_j^{\text{query}}\}_{j\in\mathcal{V}}$, parameters $\{\theta_j\}_{j\in\mathcal{V}}$

Output: $\hat{\mathbf{O}}$

Pass 1: train-view forward

$\mathbf{h}_{\text{input}}^{\text{train}} \leftarrow \mathbf{K}$
for $j \in \tau$ **do**
  $\mathbf{h}_j^{\text{train}} \leftarrow \phi_j^{\text{train}}(\{\mathbf{h}_i^{\text{train}} : i \in \text{Par}(j)\}; \theta_j)$
end for
$\hat{\mathbf{V}} \leftarrow \mathbf{h}_o^{\text{train}}$

Pass 2: train-view backward

$l \leftarrow \mathcal{L}(\hat{\mathbf{V}}, \mathbf{V})$
Initialize $\bar{\mathbf{h}}_j^{\text{train}} \leftarrow \mathbf{0}$ for all $j \in \mathcal{V}$, and set $\bar{\mathbf{h}}_o^{\text{train}} \leftarrow \partial l/\partial \hat{\mathbf{V}}$
for $j \in \text{reverse}(\tau)$ **do**
  $(\{\boldsymbol{\delta}_{i\leftarrow j}^{\text{train}}\}_{i\in\text{Par}(j)}, \Delta\theta_j) \leftarrow \phi_j^{\text{train}-\text{bwd}}(\bar{\mathbf{h}}_j^{\text{train}}, \mathbf{h}_j^{\text{train}}, \{\mathbf{h}_i^{\text{train}} : i \in \text{Par}(j)\}, \theta_j)$
  for $i \in \text{Par}(j)$ **do**
    $\bar{\mathbf{h}}_i^{\text{train}} \leftarrow \bar{\mathbf{h}}_i^{\text{train}} + \boldsymbol{\delta}_{i\leftarrow j}^{\text{train}}$
  end for
end for

Pass 3: query-view forward

$\mathbf{h}_{\text{input}}^{\text{query}} \leftarrow \mathbf{Q}$
for $j \in \tau$ **do**
  $\mathbf{h}_j^{\text{query}} \leftarrow \phi_j^{\text{query}}(\{\mathbf{h}_i^{\text{query}} : i \in \text{Par}(j)\}, \{\mathbf{h}_i^{\text{train}} : i \in \text{Par}(j)\}, \bar{\mathbf{h}}_j^{\text{train}}, \theta_j, \Delta\theta_j)$
end for
$\mathbf{O} \leftarrow \mathbf{h}_o^{\text{query}}$
return $\mathbf{O}$

Second, it enables systematic ablations. Since different design factors correspond to different nodes or subgraphs, we can vary them in a more controlled way and directly analyze how each factor affects training loss and downstream performance. Therefore, Modular TTT should not be viewed as yet another specific TTT variant, but rather as a unified framework for expressing, implementing, and analyzing the design space of TTT. The full algorithmic procedure is given in Algorithm 1. As summarized in Table 1, we define a set of primitives in Modular TTT and specify for each primitive the corresponding ϕ^{train}, $\phi^{\text{train-bwd}}$, and ϕ^{query}. Similar to TTT and linear attention, Modular TTT also incorporates learning-rate and decay mechanisms, and supports multiple choices of loss functions. Additional method details are provided in Appendix A.

4 Experiments

We implement Modular TTT in PyTorch, conduct language modeling experiments within the Flame framework (Zhang & Yang, 2025), and evaluate zero-shot downstream performance using `lm-eval-harness` (Sutawika et al., 2025). We use Modular TTT to isolate the effects of existing and proposed TTT design choices under matched backbone, data, training budget, implementation, and evaluation settings. Our experiments consist of two stages. The first stage systematically ablates the major components of TTT, including the loss function, learning-rate initialization, decay, and nonlinearity. All experiments in this stage are conducted at 160M and 410M scale, with models trained on a large-scale English pretraining corpus for 10B tokens.

Based on the configurations identified in this ablation stage, we then perform a larger-scale study. In this stage, we train representative shortlisted variants on the same pretraining corpus for 100B tokens and compare them against representative baselines, including LLaMA, GDN (Yang et al., 2025), and LaCT (Zhang et al., 2025) where available. We use the official TTT implementation (Sun et al., 2024) for throughput and reproduction comparisons. We use the OpenAI GPT-2 BPE tokenizer throughout. The sequence length is set to 2K in the ablation stage and 4K in the large-scale stage. For TTT variants, we use a chunk size of 256 to balance efficiency and empirical performance. Full architecture, training, and evaluation details are provided in Appendix B.1.

4.1 Ablation

Loss Ablation We first ablate the choice of loss function. Table 2 reveals a clear two-level split: **MSE and inner product are the only competitive losses in the current TTT setting, while L1 and RMSE consistently underperform.** Across five paired runs at each scale, we observe only

Table 1: Primitive operators and loss functions used in Modular TTT. Here, $\boldsymbol{\eta}$ denotes the learning rate and $\mathbf{M}$ denotes the decay matrix. For the GATE operator, $\mathrm{sum}(\cdot)$ and $\mathrm{cumsum}(\cdot)$ are taken along the sequence dimension. Here, s denotes a normalization constant. When $s = n$, we average the loss over samples; otherwise, we use the unnormalized loss.

Weight Operators

Primitive	ϕ^{train}	$\phi^{\text{train-bwd}}$	ϕ^{query}
Linear	$\hat{\mathbf{V}} = \mathbf{K}\mathbf{W}$	$\mathbf{dK} = \mathbf{d}\hat{\mathbf{V}}\mathbf{W}^\top$ $\hat{\mathbf{K}} = \mathbf{K} \odot \boldsymbol{\eta},\ \mathbf{dW} = \hat{\mathbf{K}}^\top \mathbf{d}\hat{\mathbf{V}}$	$\mathbf{O} = \mathbf{Q}\mathbf{W} - \mathrm{Tril}\left(\mathbf{Q}\hat{\mathbf{K}}^\top \odot \mathbf{M}\right)\mathbf{d}\hat{\mathbf{V}}$ $\mathbf{W} = \mathbf{W} - \mathbf{dW}$
Gate	$\hat{\mathbf{V}} = \mathbf{K}\,\mathrm{diag}(\mathbf{W})$	$\mathbf{dK} = \mathbf{d}\hat{\mathbf{V}}\,\mathrm{diag}(\mathbf{W})$ $\hat{\mathbf{K}} = \mathbf{K} \odot \boldsymbol{\eta}$ $\mathbf{dW} = \mathrm{sum}(\hat{\mathbf{K}} \odot \mathbf{d}\hat{\mathbf{V}})$	$\mathbf{O} = \mathbf{Q}\,\mathrm{diag}(\mathbf{W}) - \mathbf{Q} \odot \mathrm{cumsum}(\hat{\mathbf{K}} \odot \mathbf{d}\hat{\mathbf{V}})$ $\mathbf{W} = \mathbf{W} - \mathbf{dW}$

Non-weight Operators and Loss Functions

Primitive	$\phi^{\text{train}}/\phi^{\text{query}}$	$\phi^{\text{train-bwd}}$
	Non-weight Operators	
Norm	$\hat{\mathbf{v}} = \mathbf{k}/\sigma$ $\sigma = \sqrt{\left(\sum_{i=1}^{d} k_i^2\right)/d + \epsilon}$	$c = \left(\sum_{i=1}^{d} \hat{v}_i\, d\hat{v}_i\right)/d$ $\mathbf{dk} = (\mathbf{d}\hat{\mathbf{v}} - c\hat{\mathbf{v}})/\sigma$
Act	$\hat{\mathbf{V}} = f(\mathbf{K})$	$\mathbf{dK} = \mathbf{d}\hat{\mathbf{V}} \odot f'(\mathbf{K})$
Add	$\hat{\mathbf{V}} = \mathbf{K}_1 + \mathbf{K}_2$	$\mathbf{dK}_1 = \mathbf{d}\hat{\mathbf{V}},\ \mathbf{dK}_2 = \mathbf{d}\hat{\mathbf{V}}$
Mul	$\hat{\mathbf{V}} = \mathbf{K}_1 \odot \mathbf{K}_2$	$\mathbf{dK}_1 = \mathbf{d}\hat{\mathbf{V}} \odot \mathbf{K}_2,\ \mathbf{dK}_2 = \mathbf{d}\hat{\mathbf{V}} \odot \mathbf{K}_1$
	Loss Functions	
Inner Product	$l = -\left(\sum_{i=1}^{n} \hat{\mathbf{v}}_i^\top \mathbf{v}_i\right)/s$	$\mathbf{d}\hat{\mathbf{V}} = -\mathbf{V}/s$
MSE	$l = \left(\sum_{i=1}^{n} \lVert\hat{\mathbf{v}}_i - \mathbf{v}_i\rVert_2^2\right)/(2s)$	$\mathbf{d}\hat{\mathbf{V}} = (\hat{\mathbf{V}} - \mathbf{V})/s$
L1	$l = \left(\sum_{i=1}^{n} \lVert\hat{\mathbf{v}}_i - \mathbf{v}_i\rVert_1\right)/s$	$\mathbf{d}\hat{\mathbf{V}} = \mathrm{sign}(\hat{\mathbf{V}} - \mathbf{V})/s$
RMSE	$l = \sqrt{\left(\sum_{i=1}^{n} \lVert\hat{\mathbf{v}}_i - \mathbf{v}_i\rVert_2^2\right)/s}$	$\mathbf{d}\hat{\mathbf{V}} = (\hat{\mathbf{V}} - \mathbf{V})/\sqrt{s\lVert\hat{\mathbf{V}} - \mathbf{V}\rVert_F^2}$

a marginal difference between MSE and inner-product loss (Appendix Table 15). This behavior can be partly explained by the fact that the loss only enters the fast update through the gradients $\Delta\mathbf{W} = \hat{\mathbf{K}}^\top \mathbf{d}\hat{\mathbf{V}}, \mathbf{d}\hat{\mathbf{V}} = \frac{\partial \mathcal{L}(\hat{\mathbf{V}}, \mathbf{V})}{\partial \hat{\mathbf{V}}}$.

MSE preserves the residual magnitude in $\mathbf{d}\hat{\mathbf{V}}$, while inner product directly uses the target value as the write signal; both maintain an informative update scale. By contrast, L1 keeps only the sign of the residual, and RMSE normalizes the residual scale across the chunk, which weakens the memory write. A detailed analysis of the output gradients for these losses is provided in Appendix C.3.

Table 2: Experimental results for TTT loss functions. All methods use small-lr init and scalar decay. Only MSE and inner product are competitive choices, while L1 and RMSE perform substantially worse.

Loss	160M	410M
L1	3.2665	2.9727
Inner-product	3.0383	**2.7938**
MSE	**3.0380**	2.7949
RMSE	3.0658	2.8041

Table 3: Experimental results for TTT learning-rate initialization. Both MSE and inner-product losses are reported. Standard refers to initialization near 1; small-lr init sets $\eta_0 = 10^{-3}$. Small-lr init consistently achieves lower loss. Lower is better.

Loss	Decay	160M		410M	
		standard	small-lr init	standard	small-lr init
MSE	none	3.6012	3.2005	3.4036	2.9343
	scalar	3.3035	**3.0380**	3.0820	2.7949
Inner-product	none	3.2407	3.2028	2.9572	2.9273
	scalar	3.0538	3.0383	2.8054	**2.7938**

Table 4: Experimental results for TTT decay ablation. All methods use small-lr init and both MSE and inner-product losses are reported. Scalar decay provides the best efficiency–performance trade-off, while vector decay achieves the lowest loss at lower throughput and higher memory cost. Lower loss is better; higher Tgs is better. Tgs denotes training tokens per accelerator per second, and Mem is measured in GB.

Decay	MSE						Inner-product					
	160M			410M			160M			410M		
	Loss	tgs	Mem	Loss	tgs	Mem	Loss	tgs	Mem	Loss	tgs	Mem
none	3.2005	109,399	27.54	2.9343	36,956	37.25	3.2028	110,990	26.88	2.9273	37,668	35.56
scalar	3.0380	105,118	27.57	2.7949	34,978	37.30	3.0383	105,878	26.91	2.7938	35,709	35.61
vector	3.0038	79,024	30.74	2.7821	27,074	40.81	3.0048	80,156	29.54	2.7822	27,450	38.74

Learning-rate initialization We parameterize the learning rate following (Grazzi et al., 2024) as $\eta_t = 2\,\mathrm{Sigmoid}(\beta_t + b)$, where $\beta_t = \mathbf{x}_t^\top \mathbf{w} \in \mathbb{R}$. Under the default initialization, β_t is typically close to 0 at the beginning of training, so $\eta_t \approx 2\,\mathrm{Sigmoid}(b)$. We refer to the setting with $b = 0$

Table 5: Experimental results for nonlinear operators. All methods use small-lr init and scalar decay, without mean scaling. GELU and SiLU consistently improve performance and provide the best trade-off between throughput and final loss.

Method	MSE						Inner-product					
	160M			410M			160M			410M		
	Loss	tgs	Mem	Loss	tgs	Mem	Loss	tgs	Mem	Loss	tgs	Mem
Linear	3.0380	105,118	27.57	2.7949	34,978	37.30	3.0383	105,878	26.91	2.7938	35,709	35.61
Linear + GELU	3.0232	97,808	30.98	2.7844	32,114	42.83	3.0225	98,405	30.97	**2.7832**	32,456	43.07
Linear + SiLU	**3.0205**	98,293	30.98	2.7849	32,129	42.83	3.0242	98,858	30.97	2.7838	32,453	43.07
Linear + Norm	3.0300	93,226	32.15	2.8192	30,501	44.76	3.0335	94,625	31.01	2.7899	31,139	43.11

and thus $\eta_t \approx 1$ as the *default initialization*, and to the setting with $b = \log \frac{0.5\times10^{-3}}{1-0.5\times10^{-3}}, \eta_t \approx 10^{-3}$, as *small-lr init*. A similar initialization strategy is also adopted in the official TTT implementation (Sun et al., 2024) and in LaCT (Zhang et al., 2025).

TTT is highly sensitive to learning-rate initialization, and small-lr init consistently improves stability and final performance. Under both MSE and inner product losses, small-lr init yields substantially lower loss, with a more pronounced improvement under MSE.

To understand this behavior, consider the parameter update under MSE:

$$\mathbf{W} = \mathbf{W} - \mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})(\mathbf{K}\mathbf{W} - \mathbf{V}) = \left(\mathbf{I} - \mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})\mathbf{K}\right)\mathbf{W} + \mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})\mathbf{V}. \tag{7}$$

If the learning rate is too large, the matrix $\mathbf{I} - \mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})\mathbf{K}$ may have eigenvalues with magnitude larger than 1, which leads to instability. Therefore, small-lr init is beneficial for stable learning. A detailed stability analysis of the update spectrum and chunk-scale effect is provided in Appendix C.2.

Decay Ablation Motivated by prior work on linear attention, we also study decay in the TTT setting, considering three variants: no decay, scalar decay, and vector decay. Consistent with earlier findings, Table 4 shows the same ordering under both losses. Without decay, past contributions to the fast weights accumulate without attenuation, which limits the model's ability to forget stale context. Scalar decay applies a global forgetting factor to past contributions at each step, multiplicatively contracting them and recovering most of the quality gain with almost no additional computational overhead. Vector decay extends this mechanism to feature-selective forgetting and achieves the best loss, but at the cost of roughly a 25% drop in throughput and about 3 GB higher peak memory at both model scales. **Scalar decay recovers most of the quality gain from vector decay at negligible efficiency cost; no decay is clearly weaker.** To balance efficiency and performance, we use scalar decay throughout the remaining discussion.

Role of Non-linearity To study the role of nonlinearity, we append non-fast-weight activations, including GELU and SiLU, after the linear module, and separately test Norm as a normalization operator. As shown in Table 5, **simple non-fast-weight activations consistently improve performance, while Norm is mixed; GELU/SiLU provide the best trade-off between quality and efficiency.** Linear-SiLU outperforms Linear in all five paired runs at both the 160M and 410M scales (Appendix Table 15).

For a linear node followed by an activation σ, the write to the preceding fast weight becomes

$$\Delta \boldsymbol{W} = \mathbf{K}^\top \left(\mathbf{D}_o \odot \sigma'(\mathbf{U})\right), \tag{8}$$

where $\mathbf{U} = \mathbf{K}\boldsymbol{W}$ and $\mathbf{D}_o$ is the gradient propagated from the activation output. Thus the nonlinearity increases expressivity by gating the write coordinatewise. Moreover, because the derivative is bounded, it does not induce excessively large activations.

Norm exhibits less stable results and can occasionally degrade performance. We attribute this instability in part to the fact that normalization can lead to larger activations. Note that the gradient of Norm is:

$$\mathbf{dz} = \frac{1}{\sigma(\mathbf{z})}\left(\mathbf{dy} - \mathrm{mean}(\mathbf{y} \odot \mathbf{dy})\mathbf{y}\right), \mathbf{y} = \mathbf{z}/\sigma(\mathbf{z}), \sigma(\mathbf{z}) = \sqrt{\|\mathbf{z}\|_2^2/d + \epsilon}. \tag{9}$$

The $1/\sigma(\mathbf{z})$ factor can amplify gradient when $\sigma(\mathbf{z})$ is small. This helps explain why Norm is less robust than pointwise activations in Table 5.

Deep Memory ablation We next study whether replacing a single linear layer with an MLP brings additional gains. Under our controlled language-modeling setting and stabilization sweeps, Table 6 gives a negative answer:

Table 6: Deep TTT variants at 160M. Residual/gated variants and full stabilized sweeps are in Appendix D.4. Lower is better; × denotes divergence.

Shallow / Product		Activation-only		Norm-containing	
Method	Loss	Method	Loss	Method	Loss
Linear	3.0380	Linear-Linear-SiLU	3.1223	Linear-Linear-Norm	×
Linear-SiLU	**3.0205**	Linear-SiLU-Linear	3.1240	Linear-Norm-Linear	3.1594
Linear-Norm	3.0300	Linear-SiLU-Linear-SiLU	3.1156	Linear-Norm-Linear-Norm	×
Linear-Linear	3.1265			Linear-SiLU-Linear-Norm	3.1144

Deeper graph learners do not surpass the shallow frontier, and shallow linear variants remain the strongest among the tested configurations. This difficulty already appears in the two-layer linear case, and the same barrier remains after adding activations or normalization. Table 6 keeps the main-text comparison focused on chain-structured deep learners. Across these variants, added depth changes the number of layers but does not change the qualitative conclusion: activation placement gives only limited gains, and norm-containing variants either diverge or remain behind the shallow frontier. Residual and gated variants are reported in Appendix D.4, where they are analyzed together with their stabilization sweeps. Appendix C.4 provides the corresponding factor-coupling analysis for deeper fast learners.

Why Is Deep TTT Memory more difficult to optimize?
We argue that the difficulty of optimizing deep TTT memory is not merely an optimization accident, but a structural consequence of how one-step TTT updates interact with factorized fast weights. Consider a TTT memory of the form $\mathbf{Y} = f(\mathbf{X}) = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)}$. We compare two parameterizations. First, treat the product $\mathbf{W}^{(1)}\mathbf{W}^{(2)} \triangleq \mathbf{W}$ as a single fast weight.
In this case, the update is given by $\Delta = \mathbf{X}^\top \mathbf{dY}$ and $\mathbf{W} = \mathbf{W} - \Delta$.
Now consider the two-factor memory. Define

$$\mathbf{X}^{(1)} \triangleq \mathbf{X}\mathbf{W}^{(1)}, \qquad \mathbf{X}^{(2)} = \mathbf{Y} \triangleq \mathbf{X}^{(1)}\mathbf{W}^{(2)}. \tag{10}$$

The corresponding updates are

$$\begin{aligned} \Delta^{(2)} &= [\mathbf{X}^{(1)}]^\top \mathbf{dY} = [\mathbf{X}\mathbf{W}^{(1)}]^\top \mathbf{dY} = [\mathbf{W}^{(1)}]^\top \mathbf{X}^\top \mathbf{dY} = [\mathbf{W}^{(1)}]^\top \Delta, \\ \mathbf{dX}^{(1)} &= \mathbf{dX}^{(2)}[\mathbf{W}^{(2)}]^\top = \mathbf{dY}[\mathbf{W}^{(2)}]^\top, \Delta^{(1)} = \mathbf{X}^\top \mathbf{dX}^{(1)} = \mathbf{X}^\top \mathbf{dY}[\mathbf{W}^{(2)}]^\top = \Delta[\mathbf{W}^{(2)}]^\top. \end{aligned} \tag{11}$$

Therefore, the induced update on the effective fast weight becomes

$$\begin{aligned} &(\mathbf{W}^{(1)} - \Delta[\mathbf{W}^{(2)}]^\top)(\mathbf{W}^{(2)} - [\mathbf{W}^{(1)}]^\top \Delta) \\ &= \mathbf{W}^{(1)}\mathbf{W}^{(2)} - \Delta[\mathbf{W}^{(2)}]^\top \mathbf{W}^{(2)} - \mathbf{W}^{(1)}[\mathbf{W}^{(1)}]^\top \Delta + \Delta[\mathbf{W}^{(2)}]^\top [\mathbf{W}^{(1)}]^\top \Delta. \end{aligned} \tag{12}$$

For any nonzero scalar c, we have $\mathbf{W}^{(1)}\mathbf{W}^{(2)} = (c\mathbf{W}^{(1)})(c^{-1}\mathbf{W}^{(2)})$.
Thus the represented memory function $f(\mathbf{X}) = \mathbf{X}\mathbf{W}$ remains unchanged.
However, under this rescaling, the two-layer update becomes

$$\Delta[\mathbf{W}^{(2)}]^\top [\mathbf{W}^{(1)}]^\top \Delta - c^2 \mathbf{W}^{(1)}\mathbf{W}^{(1)\top}\Delta - c^{-2}\Delta\mathbf{W}^{(2)\top}\mathbf{W}^{(2)}. \tag{13}$$

Therefore, two factorizations that represent the same effective fast weight can induce substantially different TTT update directions.
When c is large, the dominant terms are $\Delta[\mathbf{W}^{(2)}]^\top[\mathbf{W}^{(1)}]^\top\Delta - c^2\mathbf{W}^{(1)}\mathbf{W}^{(1)\top}\Delta$.
When c is small, the dominant terms are $\Delta[\mathbf{W}^{(2)}]^\top[\mathbf{W}^{(1)}]^\top\Delta - c^{-2}\Delta\mathbf{W}^{(2)\top}\mathbf{W}^{(2)}$.
Therefore, optimization is not only required to learn the correct effective fast weight $\mathbf{W}$, but also to discover a factorization that induces favorable update dynamics. This additional degree of freedom makes deep TTT Memory substantially more difficult to optimize.

4.2 Efficiency Benchmark

Primitive-level backward computation. The analytic Linear and Norm operators yield $1.65\times$ and $2.62\times$ speedups, respectively, over `torch.autograd.grad` references under identical inputs. Full settings and results are reported in Appendix D.8 (Table 31).

End-to-end throughput. Modular TTT achieves a $2.2\times$–$3.3\times$ throughput improvement over the official TTT implementation (Sun et al., 2024) in two representative 160M norm-containing settings; see Appendix D.9 (Table 32) for details.

4.3 Scale Up

Based on the preceding ablations, we compare linear and linear-SiLU learners with MSE or inner-product loss and scalar decay, balancing validation loss, throughput, and simplicity rather than selecting the lowest-loss decay option.

Table 7: Large-scale downstream evaluation at 410M and 1.45B. Arrows indicate the direction of improvement. External baselines: LLaMA (Touvron et al., 2023), GDN (Yang et al., 2025), and LaCT (Zhang et al., 2025) where available.

Method	Params	Perplexity				Multiple-choice									Containment			
		loss↓	Wiki. ppl↓	LMB. ppl↓	Avg ppl↓	BoolQ acc↑	PIQA acc↑	Hella. acc-n↑	Wino. acc↑	ARC-e acc↑	ARC-c acc-n↑	OBQA acc-n↑	SIQA acc↑	Avg acc↑	SWDE acc↑	SQuAD acc↑	FDA acc↑	Avg acc↑
External baselines, 410M																		
LLaMA	410M	2.5479	20.63	25.64	23.14	59.51	69.15	47.47	55.80	65.28	31.83	35.40	40.63	50.63	58.96	40.42	62.43	53.94
GDN	410M	2.5559	21.00	22.62	21.81	59.69	71.00	47.64	52.88	65.66	31.83	34.80	40.07	50.45	22.68	31.80	16.33	23.60
LaCT	410M	2.5582	21.28	25.94	23.61	60.86	70.13	47.10	52.49	65.03	30.03	35.80	40.02	50.18	33.75	30.16	18.60	27.50
Modular TTT MSE Loss, 410M																		
M	410M	2.5648	21.28	25.10	23.19	53.76	69.04	47.92	52.33	67.21	31.74	35.40	41.10	49.81	25.02	32.14	8.53	21.90
M-SiLU	410M	2.5582	21.07	23.80	22.43	61.44	70.40	48.10	52.88	65.66	33.87	35.00	40.53	50.99	28.08	31.77	8.80	22.88
Modular TTT Inner-Product Loss, 410M																		
M	410M	2.5621	21.38	24.81	23.10	56.30	69.37	47.71	53.04	64.52	30.89	37.40	39.41	49.83	26.28	33.91	9.62	23.27
M-SiLU	410M	2.5569	20.98	23.22	22.10	60.03	70.62	48.14	53.20	66.50	31.91	34.40	39.92	50.59	25.38	31.84	12.89	23.37
External baselines, 1.45B																		
LLaMA	1.45B	2.3041	15.26	12.78	14.02	62.91	73.83	59.37	57.22	73.48	38.31	39.80	40.43	55.67	75.34	44.50	75.77	65.20
GDN	1.45B	2.3042	15.53	11.87	13.70	60.70	74.37	59.16	60.54	73.61	39.68	41.00	41.30	56.30	52.39	38.37	20.15	36.97
Modular TTT MSE Loss, 1.45B																		
M	1.45B	2.3193	15.84	12.71	14.28	60.37	74.43	58.96	58.41	72.94	40.61	41.80	41.71	56.15	41.40	38.71	24.77	34.96
M-SiLU	1.45B	2.3197	15.85	12.90	14.37	59.91	72.91	58.92	58.64	73.23	40.02	40.60	41.15	55.67	44.19	37.77	24.41	35.46
Modular TTT Inner-Product Loss, 1.45B																		
M	1.45B	2.3150	15.92	12.19	14.05	62.35	73.18	58.70	59.98	73.48	40.78	41.00	42.07	56.44	43.65	37.90	17.42	32.99
M-SiLU	1.45B	2.3156	15.84	12.31	14.07	56.97	74.21	58.80	58.64	73.74	39.93	39.20	42.32	55.48	42.30	38.00	17.79	32.70

Table 7 reports downstream benchmarks at 410M and 1.45B, covering perplexity, multiple-choice, and containment-style tasks evaluated with lm-evaluation-harness (Sutawika et al., 2025). The short-listed Modular TTT variants remain competitive with strong recurrent baselines on training loss and multiple-choice accuracy, while containment-style tasks remain more challenging. At 410M, the SiLU variant improves the multiple-choice average over the external baselines; at 1.45B, the inner-product linear variant gives the strongest Modular TTT result and is close to GDN on average multiple-choice accuracy. Appendix D.5 reports additional external-baseline checks, including the LaCT comparison.

Per-token Loss. Appendix D.6 further examines per-token loss along 8k sequences, showing that the selected shallow Modular TTT variants remain stable around and beyond the 4k training context.

RULER. Detailed RULER (Hsieh et al., 2024) evaluations are reported in Appendix D.7, with family-level and template-level results in Tables 29–30. The inner-product linear variant is the strongest Modular TTT configuration on `niah_single` at 410M, while the MSE SiLU variant is strongest at 1.45B for contexts up to 4k. However, LLaMA remains substantially stronger, especially at 8k, indicating that precise long-context recall remains a limitation of the current shallow Modular TTT variants.

5 Conclusion

We presented Modular TTT, a framework that factorizes the TTT inner learner into explicit, independently controllable design dimensions. Modular TTT represents the fast-weight network as a DAG of registered primitives and automatically composes their train-view forward, train-view backward, and causal query-view rules into the full graph-level TTT computation, including the fast-weight state transition. This shared procedure reduces the cost of constructing and comparing TTT variants by avoiding topology-specific global update derivations for recombinations of existing primitives, and turns TTT design from a collection of custom implementations into a modular space that can be explored and analyzed systematically. Using this framework, we identified which components of TTT provide consistent gains and which do not, and trained the resulting best variant to performance comparable to GDN at 410M and 1.45B scales.

AI USE STATEMENT

Generative AI tools assisted with translating text, reviewing manuscript drafts to provide writing suggestions, and adjusting LaTeX formatting. LLM-based agents automated the submission of experiment jobs in batches. The authors take full responsibility for all claims and artifacts.

ETHICS STATEMENT

This work studies test-time training through language-model training and benchmark evaluation. The proposed framework supports systematic comparison of TTT components and does not target any specific downstream application.

REPRODUCIBILITY STATEMENT

Our implementation and experiment configurations are provided in the supplementary material. The appendices provide method and update-rule details, model and training configurations, evaluation protocols, theoretical analyses, and additional experimental results (Appendices A–D).

REFERENCES

Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 244–253, 2018.

Sanjeev Arora, Nadav Cohen, Wei Hu, and Yuping Luo. Implicit regularization in deep matrix factorization. In *Advances in Neural Information Processing Systems*, volume 32, pp. 7411–7422, 2019.

Jimmy Ba, Geoffrey E. Hinton, Volodymyr Mnih, Joel Z. Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

Pierre Baldi and Kurt Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural Networks*, 2(1):53–58, 1989. doi: 10.1016/0893-6080(89)90014-2.

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. *arXiv preprint arXiv:2504.13173*, 2025b.

Ali Behrouz, Meisam Razaviyayn, Peilin Zhong, and Vahab Mirrokni. Nested learning: The illusion of deep learning architectures. *arXiv preprint arXiv:2512.24695*, 2025c.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663*, 2025d.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2622–2632, 2019.

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

Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 10041–10071. PMLR, 2024.

Riccardo Grazzi, Julien Siems, Jörg K.H. Franke, Arber Zela, Frank Hutter, and Massimiliano Pontil. Unlocking state-tracking in linear RNNs through negative eigenvalues. In *NeurIPS 2024 Workshop on Mathematics of Modern Machine Learning*, 2024. URL `https://openreview.net/forum?id=EXGAodFkQX`.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*, 2022.

Dongchen Han, Yining Li, Tianyu Li, Zixuan Cao, Ziming Wang, Jun Song, Yu Cheng, Bo Zheng, and Gao Huang. Vit^3: Unlocking test-time training in vision. *arXiv preprint arXiv:2512.01643*, 2025.

Qiang Hao, Rui Cai, Yanwei Pang, and Lei Zhang. From one tree to a forest: A unified solution for structured web data extraction. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 775–784, 2011.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*, 2024. URL `https://openreview.net/forum?id=kIoBbc76Sy`.

Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. In *Advances in Neural Information Processing Systems*, volume 34, 2021.

Kazuki Irie, Róbert Csordás, and Jürgen Schmidhuber. The dual form of neural networks revisited: Connecting test time predictions to training patterns via spotlights of attention. In *Proceedings of the 39th International Conference on Machine Learning*, 2022.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

Kenji Kawaguchi. Deep learning without poor local minima. In *Advances in Neural Information Processing Systems*, volume 29, pp. 586–594, 2016.

Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of neural sequence models. In *Proceedings of the 35th International Conference on Machine Learning*, 2018.

Zeman Li, Ali Behrouz, Yuan Deng, Peilin Zhong, Praneeth Kacham, Mahdi Karami, Meisam Razaviyayn, and Vahab Mirrokni. TNT: Improving chunkwise training for test-time memorization. *arXiv preprint arXiv:2511.07343*, 2025.

Eric Martin and Chris Cundy. Parallelizing linear recurrent neural nets over sequence length. In *International Conference on Learning Representations*, 2018.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? A new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2618–2628, 2018.

Antonio Orvieto, Samuel L. Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pp. 1525–1534, 2016.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, Xingjian Du, Matteo Grella, Kranthi Gv, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartłomiej Koptyra, Hayden Lau, Jiaju Lin, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Johan Wind, Stanisław Woźniak, Zhenyuan Zhang, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. RWKV: Reinventing RNNs for the transformer era. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 14048–14077, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.936. URL https://aclanthology.org/2023.findings-emnlp.936/.

Bo Peng, Ruichong Zhang, Daniel Goldstein, Eric Alcaide, Xingjian Du, Haowen Hou, Jiaju Lin, Jiaxing Liu, Janna Lu, William Merrill, Guangyu Song, Kaifeng Tan, Saiteja Utpala, Nathan Wilce, Johan S. Wind, Tianyi Wu, Daniel Wuttke, and Christian Zhou-Zheng. RWKV-7 "goose" with expressive dynamic state evolution. *arXiv preprint arXiv:2503.14456*, 2025. doi: 10.48550/arXiv.2503.14456. URL https://arxiv.org/abs/2503.14456.

Zhen Qin, Songlin Yang, and Yiran Zhong. Hierarchically gated recurrent neural network for sequence modeling. In *Advances in Neural Information Processing Systems*, 2023.

Zhen Qin, Dong Li, Weigao Sun, Weixuan Sun, Xuyang Shen, Xiaodong Han, Yunshen Wei, Baohong Lv, Xiao Luo, Yu Qiao, and Yiran Zhong. Transnormerllm: A faster and better large language model with improved transnormer, 2024a.

Zhen Qin, Songlin Yang, Weixuan Sun, Xuyang Shen, Dong Li, Weigao Sun, and Yiran Zhong. HGRN2: Gated linear RNNs with state expansion. In *First Conference on Language Modeling*, 2024b.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 2383–2392, 2016.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: An adversarial winograd schema challenge at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. Social IQa: Commonsense reasoning about social interactions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pp. 4463–4473, 2019. doi: 10.18653/v1/D19-1454.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *Proceedings of the 38th International Conference on Machine Learning*, 2021.

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

Noam Shazeer. GLU variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, Tatsunori Hashimoto, and Carlos Guestrin. 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.

Lintang Sutawika, Hailey Schoelkopf, Leo Gao, Baber Abbasi, Stella Biderman, Jonathan Tow, ben fattori, Charles Lovering, farzanehnakhaee70, Jason Phang, Anish Thite, Fazz, Aflah, Niklas, Thomas Wang, sdtblck, nopperl, gakada, tttyuntian, researcher2, Julen Etxaniz, Chris, Hanwool Albert Lee, Leonid Sinev, Zdeněk Kasner, Kiersten Stokes, Khalid, KonradSzafer, Jeffrey Hsu, and Anjor Kanekar. EleutherAI/lm-evaluation-harness: v0.4.9.1, August 2025. URL `https://zenodo.org/records/16737642`.

Arnuv Tandon, Karan Dalal, Xinhao Li, Daniel Koceja, Marcel Rød, Sam Buchanan, Xiaolong Wang, Jure Leskovec, Sanmi Koyejo, Tatsunori Hashimoto, Carlos Guestrin, Jed McCaleb, Yejin Choi, and Yu Sun. End-to-end test-time training for long context. *arXiv preprint arXiv:2512.23675*, 2025.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

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*, 2024a.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. In *Advances in Neural Information Processing Systems*, volume 37, 2024b. doi: 10.52202/079017-3668. URL `https://proceedings.neurips.cc/paper_files/paper/2024/hash/d13a3eae72366e61dfdc7eea82eeb685-Abstract-Conference.html`.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule. In *International Conference on Learning Representations*, 2025.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

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 and Songlin Yang. Flame: Flash language modeling made easy, January 2025. URL `https://github.com/fla-org/flame`.

APPENDIX

APPENDIX CONTENTS

Appendix A Method details 14
Appendix B Experimental setup 16
Appendix C Analysis of Modular TTT Updates 18
Appendix D Additional ablation results 22
Appendix E Limitations 30

A METHOD DETAILS

A.1 DESIGN AXES

Table 8: Configuration axes in Modular TTT. The table summarizes the graph, loss, learning-rate, decay, normalization, and scheduling choices controlled by our implementation and experiments. Primitive and loss definitions follow Table 1.

Axis	Options	Component
Graph and loss choices		
Primitive	Linear, Gate, Norm, Act, Add, Mul	Node operator ϕ_j
Loss	Inner product, MSE, L1, RMSE	Loss $\mathcal{L}$
Update choices		
LR initialization	Standard init ($\eta_0 \approx 1$), small-lr init (10^{-3})	Initial scale η_0
Decay	None, scalar, vector	Weight decay

A.2 GRAPH MEMORY INSTANTIATIONS

The graph memory is specified by an ordered `graph_nodes` list and a designated `graph_output`. Each node is one of the primitives in Table 1; linear nodes carry fast weights, while activation, addition, multiplication, and normalization nodes only transform activations or gradients. The implementation executes the same graph in the train-view forward, train-view backward, and query-view forward.

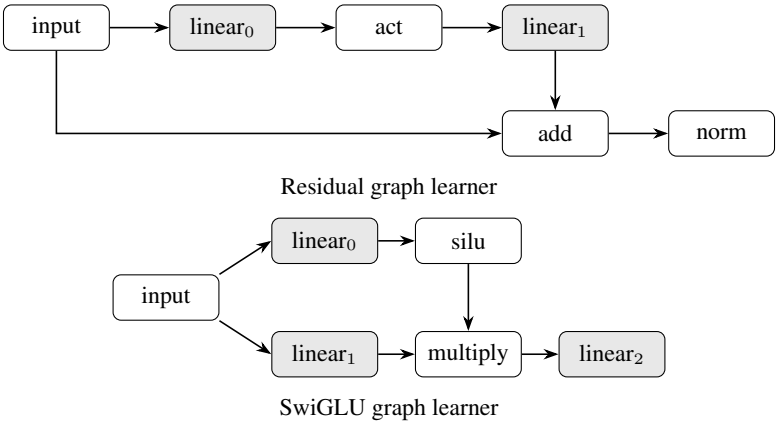

Figure 3: Two representative graph memories. Gray nodes contain fast weights, while white nodes do not. **Top:** Residual learner; the skip connection carries input directly to add, bypassing the two linear nodes. **Bottom:** SwiGLU learner (Shazeer, 2020; Zhang et al., 2025); two branches share the same input and are recombined by a multiplicative gate.

A.3 GRADIENT AND UPDATE RULES

Table 1 lists the train-view, train-view backward, and query-view rules for all primitives. This subsection records the implementation-specific details behind the train-view backward pass of Al-

gorithm 1: the loss gradient at the output node and the gradients used to propagate it through the graph.

Output gradient rules. All loss forms use the normalization constant s from Table 1. Within one chunk, the sample count in the table is $n = c$, so

$$s = \begin{cases} c & \text{if mean_loss = True,} \\ 1 & \text{otherwise,} \end{cases} \tag{A.1}$$

where c is the chunk size. The corresponding output gradients are

$$\text{MSE:} \quad \mathbf{d}\hat{\mathbf{V}} = \frac{\hat{\mathbf{V}} - \mathbf{V}}{s}, \tag{A.2}$$

$$\text{L1:} \quad \mathbf{d}\hat{\mathbf{V}} = \frac{\operatorname{sign}(\hat{\mathbf{V}} - \mathbf{V})}{s}, \tag{A.3}$$

$$\text{Inner product:} \quad \mathbf{d}\hat{\mathbf{V}} = -\frac{\mathbf{V}}{s}, \tag{A.4}$$

$$\text{RMSE:} \quad \mathbf{d}\hat{\mathbf{V}} = \frac{\hat{\mathbf{V}} - \mathbf{V}}{\sqrt{s\,\|\hat{\mathbf{V}} - \mathbf{V}\|_F^2}}. \tag{A.5}$$

These are the same rules used in the loss-choice analysis in Appendix C.3. MSE and inner product preserve chunk-level scale information; L1 and RMSE respectively remove magnitude or normalize it.

Primitive-level backward rules. For $y = x\mathbf{W}$,

$$\mathbf{d}x = \mathbf{d}y\,\mathbf{W}^\top. \tag{A.6}$$

The matrix is the incoming chunk state $\mathbf{W}_{\text{start}}$, because the train-view forward and train-view backward are evaluated before the query-view write. For an elementwise activation a,

$$\mathbf{d}x = \mathbf{d}y \odot a'(x), \tag{A.7}$$

and for root mean square normalization,

$$\sigma(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}, \qquad y = \frac{x}{\sigma(x)}, \tag{A.8}$$

the gradient propagated to the input is

$$\mathbf{d}x = \frac{1}{\sigma(x)}\left(\widetilde{\mathbf{d}y} - \operatorname{mean}(y \odot \widetilde{\mathbf{d}y})\, y\right). \tag{A.9}$$

Here $\widetilde{\mathbf{d}y} = \mathbf{w} \odot \mathbf{d}y$ when an affine scale $\mathbf{w}$ (RMSNorm's γ) is present, and $\widetilde{\mathbf{d}y} = \mathbf{d}y$ otherwise. For merge primitives,

$$\text{if } y = x_1 \odot x_2, \quad \mathbf{d}x_1 = \mathbf{d}y \odot x_2, \qquad \mathbf{d}x_2 = \mathbf{d}y \odot x_1; \qquad \text{if } y = x_1 + x_2, \quad \mathbf{d}x_1 = \mathbf{d}x_2 = \mathbf{d}y. \tag{A.10}$$

These local rules are the analytic backward operators used in the train-view backward without invoking nested automatic differentiation.

A.4 Chunkwise query update

The Linear query-view rule in Table 1 is the chunkwise dual form of the token-wise recurrence. For a single fast linear primitive without decay,

$$\mathbf{W}_t = \mathbf{W}_{t-1} - \hat{\mathbf{k}}_t^\top \mathbf{d}\hat{\mathbf{v}}_t, \qquad \mathbf{o}_t = \mathbf{q}_t \mathbf{W}_t. \tag{A.11}$$

Telescoping from the incoming chunk state $\mathbf{W}_{\text{start}} = \mathbf{W}_0$ gives

$$\mathbf{W}_t = \mathbf{W}_{\text{start}} - \sum_{s=1}^{t} \hat{\mathbf{k}}_s^\top \mathbf{d}\hat{\mathbf{v}}_s. \tag{A.12}$$

The readout at position t is therefore

$$\mathbf{o}_t = \mathbf{q}_t \mathbf{W}_t = \mathbf{q}_t \mathbf{W}_{\text{start}} - \sum_{s=1}^{t} (\mathbf{q}_t \hat{\mathbf{k}}_s^\top)\, \mathbf{d}\hat{\mathbf{v}}_s. \tag{A.13}$$

In matrix form,

$$\mathbf{O} = \mathbf{Q}\mathbf{W}_{\text{start}} - \operatorname{Tril}(\mathbf{Q}\hat{\mathbf{K}}^\top)\mathbf{d}\hat{\mathbf{V}}, \qquad \mathbf{W}_{\text{end}} = \mathbf{W}_{\text{start}} - \hat{\mathbf{K}}^\top \mathbf{d}\hat{\mathbf{V}}, \tag{A.14}$$

where $\hat{\mathbf{K}}$ is the scaled key in Table 1, and $\operatorname{Tril}(\cdot)$ denotes the inclusive lower-triangular operator, retaining entries with $s \le t$ and zeroing entries with $s > t$ to enforce causality within the chunk.

A.5 Learning rate, decay, and fast-state variants

Learning-rate and decay factors are predicted by the outer token mixer and injected only at fast nodes. Following Table 1, constant, scalar, and vector learning rates form the scaled key $\hat{\mathbf{K}}$:

$$\hat{\mathbf{K}}^{(j)} = \boldsymbol{\eta}^{(j)} \odot \mathbf{K}^{(j)}, \quad \text{for constant, scalar, and vector learning rates.} \tag{A.15}$$

In the formulas below, $\hat{\mathbf{K}}^\top \mathbf{d}\hat{\mathbf{V}}$ denotes the effective update product after this injection. The gradient $\mathbf{d}\hat{\mathbf{V}}$ already carries the scaling factor $1/s$ from Appendix A.3. When mean scaling is enabled ($s = c$), this normalizes by chunk length; when $s = 1$, update scale is controlled by the learned learning-rate factor and its initialization.

When vector decay is enabled, let $\phi_{t,s} = \prod_{r=s+1}^{t} f_r$ and $\phi_{t,0} = \prod_{r=1}^{t} f_r$. This is the vector-decay analogue of the decay matrix $\mathbf{M}$ in Table 1. The chunkwise readout and end-of-chunk state update become

$$\mathbf{o}_t = \mathbf{q}_t\big(\phi_{t,0} \odot \mathbf{W}_{\text{start}}\big) - \sum_{s=1}^{t} \mathbf{q}_t\big(\phi_{t,s} \odot \hat{\mathbf{k}}_s^\top \mathbf{d}\hat{\mathbf{v}}_s\big), \tag{A.16}$$

$$\mathbf{W}_{\text{end}} = \phi_{c,0} \odot \mathbf{W}_{\text{start}} - \sum_{s=1}^{c} \phi_{c,s} \odot \hat{\mathbf{k}}_s^\top \mathbf{d}\hat{\mathbf{v}}_s. \tag{A.17}$$

The fused fast-update kernel accumulates these products in log space using the supplied $\log f_t$ tensors.

We implement RMSNorm by cascading Norm and Gate. When `rmsnorm_use_fast_weight` is disabled, the affine scale is not a fast weight. When it is enabled, the affine scale is promoted to a fast state and updated by the same causal mechanism as a linear fast node.

It also applies a persistent output gate after the graph output:

$$\mathbf{h}_{\text{gate}} = \text{Norm}\big(\mathbf{h}_o^{\text{query}}\big) \odot \sigma\big(\text{Gate}(\mathbf{h})\big). \tag{A.18}$$

Finally, fast linear nodes use either zero initialization (`flash`) or the Gaussian initialization of the original TTT recipe (Sun et al., 2024) (`official`). The initialization comparison confirms that zero initialization degrades performance across the tested configurations.

B Experimental setup

B.1 Model architecture

All models use a pre-normalization residual backbone. Each block applies RMSNorm, then the token mixer, then a residual connection, followed by RMSNorm and a bias-free GLU channel mixer with SiLU activation, following the LLaMA design (Touvron et al., 2023). The channel mixer has gate, up, and down projections. For Modular TTT, the token mixer uses graph memory with query, key, value, and output projections; the large-scale Modular TTT variants use an L2-normalized key side and leave the query side unnormalized.

Initialization. Unless otherwise stated, embeddings, the channel-mixer projections, and standard linear layers are initialized from a truncated normal distribution with standard deviation 0.02, with biases set to zero. In Modular TTT token mixers, the query, key, and value projections use Xavier uniform initialization with gain 0.01, while the output projection is zero-initialized by the residual rescaling rule. Fast weights use the official TTT initialization, namely Gaussian initialization with standard deviation 0.02. The scalar decay parameter is initialized so that the initial decay is approximately 0.99, and the small-lr initialization sets the initial inner learning rate to approximately 10^{-3}.

B.2 Training details

All language-model runs are trained on a large-scale English pretraining corpus with the OpenAI GPT-2 BPE tokenizer (`openai-community/gpt2`; vocabulary size 50,257) and the next-token prediction objective. Table 10 lists the shared optimization settings, and Table 11 gives the scale-specific settings.

Table 9: Architecture and Modular TTT hyperparameters. The scale names follow the paper notation. d_{model} is the hidden dimension, L the number of layers, H the number of heads, d_{head} the per-head query/key/value dimension, and d_{ffn} the GLU inner width.

Hyperparameter	160M	410M	1.45B
Backbone			
d_{model}	768	1024	2048
L (layers)	12	24	24
H (heads)	6	8	16
d_{head}	128	128	128
d_{ffn} (GLU inner)	2048	2816	5632
Vocabulary size	50,257	50,257	50,257
Normalization	RMSNorm	RMSNorm	RMSNorm
Channel mixer	GLU	GLU	GLU
Channel activation	SiLU	SiLU	SiLU
Linear bias	no	no	no
Residual layout	pre-norm	pre-norm	pre-norm
Modular TTT			
Chunk size	256	256	256
Mean scaling	disabled	disabled	disabled
Decay	scalar	scalar	scalar
Inner learning rate	scalar	scalar	scalar
Key normalization	L2	L2	L2
Query normalization	none	none	none
Fast-weight init.	Gaussian, std 0.02	Gaussian, std 0.02	Gaussian, std 0.02

We use the standard training split of this corpus. Documents are tokenized online, concatenated into a token stream, and packed into fixed-length training sequences of length 2048 for ablation runs and 4096 for scale-up runs. We do not insert an attention mask at document boundaries, so packed sequences may cross document boundaries under the standard causal next-token objective.

Table 10: Common training hyperparameters.

Hyperparameter	Value
Optimizer	AdamW
Learning rate	3×10^{-4}
AdamW betas	$(0.9, 0.95)$
AdamW ϵ	10^{-8}
LR schedule	warmup-stable-decay
Warmup	1000 steps
Minimum LR ratio	0.1
Precision	BF16
Gradient clipping	1.0
Seed	42

We use FSDP data-parallel sharding over the devices listed in Table 11.

Table 11: Training hyperparameters. Batch and eval batch are per-device batch sizes. The training token budget is computed as devices $\times$ batch $\times$ accumulation $\times$ context length $\times$ steps: the two ablation settings use $8 \times 32 \times 1 \times 2048 \times 20{,}000 = 10.5$B tokens, and the two scale-up settings use $16 \times 8 \times 2 \times 4096 \times 100{,}000 = 104.9$B tokens.

Scale	Tokens	Devices	Batch	Accum.	Ctx.	Steps	Eval batch
Ablation							
160M	10B	8	32	1	2048	20,000	192
410M	10B	8	16	2	2048	20,000	128
Scale-up							
410M	100B	16	8	2	4096	100,000	128
1.45B	100B	32	4	2	4096	100,000	64

We run downstream evaluation with `lm-eval-harness` v0.4.9.1. The evaluation covers perplexity on WikiText-103 (Merity et al., 2017) and LAMBADA (Paperno et al., 2016), multiple-choice accuracy on BoolQ (Clark et al., 2019), PIQA (Bisk et al., 2020), HellaSwag (Zellers et al., 2019), WinoGrande (Sakaguchi et al., 2020), ARC (Clark et al., 2018), OpenBookQA (Mihaylov et al.,

2018), and SIQA (Sap et al., 2019), and containment-style evaluation on SWDE (Hao et al., 2011) and SQuAD (Rajpurkar et al., 2016). Table 12 gives the exact harness task names and metrics.

Table 12: Downstream evaluation tasks. The table lists the exact `lm-eval-harness` task names and metrics used in Table 7. Metrics marked `acc_norm` use length-normalized multiple-choice scoring; metrics marked `acc` use unnormalized accuracy. Group averages are arithmetic means over the listed metrics within each group.

Group	Reported as	`lm-eval-harness` task	Metric
Perplexity	Wiki.	`wikitext`	`word_perplexity`
Perplexity	LMB.	`lambada_openai`	`perplexity`
Multiple-choice	BoolQ	`boolq`	`acc`
Multiple-choice	PIQA	`piqa`	`acc`
Multiple-choice	Hella.	`hellaswag`	`acc_norm`
Multiple-choice	Wino.	`winogrande`	`acc`
Multiple-choice	ARC-e	`arc_easy`	`acc`
Multiple-choice	ARC-c	`arc_challenge`	`acc_norm`
Multiple-choice	OBQA	`openbookqa`	`acc_norm`
Multiple-choice	SIQA	`social_iqa`	`acc`
Containment	SWDE	`swde`	`contains`
Containment	SQuAD	`squad_completion`	`contains`
Containment	FDA	`fda`	`contains`

For throughput comparisons, we report training tokens per second measured at the context length used by the corresponding run: 2048 for ablation and throughput comparisons and 4096 for scale-up runs. Throughput is averaged over 100 measured steps after a 10-step warm-up. Peak memory is the maximum reserved device memory reported by the training logger during a training step, after optimizer states have been initialized. Downstream evaluation uses the tasks and metrics listed in Table 12.

C Analysis of Modular TTT Updates

We analyze one chunk update under the output-gradient rules in Appendix A.3. The analysis tracks how each modular choice changes the scale, direction, or conditioning of the fast-weight write.

The order follows the main-text ablations. We first study update scale and small-lr init. We then compare loss functions, decay rules, shallow nonlinear modules, normalization, and multilayer fast learners.

The section stays with the operating point used in the main text; it does not introduce a separate one. When mean scaling appears below, it refers to the option $s = c$ in Table 1. Equivalently, this is the `mean_loss` option that divides the chunk-level output gradient by the chunk size.

C.1 Update scale

Proposition 1 (Chunk update scale can grow with chunk size)**.** *Consider one linear fast primitive over a chunk. The train-view gradient is evaluated at the incoming fast weight $\boldsymbol{W}_0$. The per-token update is*

$$\Delta \boldsymbol{W}_t = \eta\, \boldsymbol{k}_t^\top \boldsymbol{d}_t, \qquad \boldsymbol{d}_t = \frac{\partial l}{\partial \hat{\boldsymbol{v}}_t}/s, \qquad \hat{\boldsymbol{v}}_t = \boldsymbol{k}_t \boldsymbol{W}_0. \tag{A.19}$$

Let $\Delta \boldsymbol{W}$ denote the update subtracted from the incoming fast weight over one chunk, then $\|\Delta W\|_F = O(\eta c/s)$.

Proof. Since

$$\Delta \boldsymbol{W} = \eta/s \sum_{t=1}^{c} \boldsymbol{k}_t^\top \boldsymbol{d}_t, \tag{A.20}$$

Then

$$\|\Delta \boldsymbol{W}\|_F \;\le\; \eta/s \sum_{t=1}^{c} \|\boldsymbol{k}_t\|_2 \|\boldsymbol{d}_t\|_2 \le \eta c/s \max\{\|\boldsymbol{k}_t\|_2 \|\boldsymbol{d}_t\|_2\} = O(\eta c/s). \tag{A.21}$$

□

Remark 1 (Small-lr init as scale control)**.** *When $s = 1$, the output gradient is not divided. Instead, small-lr init sets $\eta_{\text{init}} = \eta_0$ so that $c\eta_0 = O(1)$. With the main-text chunk size $c = 256$ and $\eta_0 = 10^{-3}$, the product is $c\eta_0 = 0.256$. Thus the initial chunk update has controlled scale even without mean scaling.*

Mean scaling and small-lr init control the same initial quantity, but they differ after outer-loop training begins: mean scaling keeps the fixed factor $1/c$ in every output gradient, while small-lr init only sets the starting value of the learned learning-rate predictor.

C.2 SMALL-LR INIT ($\eta_0 = 10^{-3}$)

The learning rate η in Modular TTT is not a fixed scalar. It is predicted from the outer representation. Following the main text, we write it as

$$\eta_t = 2\,\sigma(\beta_t + b), \qquad \beta_t = \mathbf{x}_t^\top \mathbf{w}, \tag{A.22}$$

where σ is the sigmoid function and b is a fixed scalar offset.

Without any offset ($b = 0$), the initial prediction gives $\eta_t \approx 1.0$. This coefficient has the right order of magnitude for standard linear-attention-style accumulation, but it is too large for TTT updates without mean scaling. By Proposition 1, updates without mean scaling and with $\eta \sim 1$ have $O(c)$ initial chunk scale. This agrees with the main-text observation that standard init can be unstable and is empirically worse than small-lr init.

Proposition 2 (Small learning rates control the MSE update spectrum)**.** *Consider the MSE update without mean scaling used in the main-text stability discussion:*

$$\boldsymbol{W}^+ = \boldsymbol{W} - \mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})(\mathbf{K}\boldsymbol{W} - \mathbf{V}) = \mathbf{A}\boldsymbol{W} + \mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})\mathbf{V}, \tag{A.23}$$

where

$$\mathbf{A} = \mathbf{I} - \mathbf{H}, \qquad \mathbf{H} = \mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})\mathbf{K}. \tag{A.24}$$

If $\lambda_{\max}(\mathbf{H}) > 2$, then the homogeneous update $\boldsymbol{W} \mapsto \mathbf{A}\boldsymbol{W}$ amplifies some direction. If $0 \le \eta_t \le \eta_0$ for all tokens, then

$$\lambda_{\max}(\mathbf{H}) \le \eta_0 \|\mathbf{K}\|_2^2. \tag{A.25}$$

Thus a small initial learning rate reduces the spectral scale of $\mathbf{H}$ and lowers the risk that $\mathbf{A}$ has eigenvalues with magnitude greater than one.

Proof. **Step 1: Identify the homogeneous update.** The MSE update is affine in $\boldsymbol{W}$. Its homogeneous part is multiplication by

$$\mathbf{A} = \mathbf{I} - \mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})\mathbf{K}. \tag{A.26}$$

This is the matrix that appears in the main-text stability discussion.
Step 2: Relate eigenvalues of $\mathbf{A}$ and $\mathbf{H}$. Because $\eta_t \ge 0$, the matrix

$$\mathbf{H} = \mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})\mathbf{K} \tag{A.27}$$

is positive semidefinite. Thus all eigenvalues of $\mathbf{H}$ are nonnegative, so an eigenvalue of $\mathbf{A} = \mathbf{I} - \mathbf{H}$ can leave the unit disk only through the negative side, which occurs exactly when some $\lambda_i(\mathbf{H}) > 2$. If $\lambda_i(\mathbf{H})$ is an eigenvalue of $\mathbf{H}$, then $1 - \lambda_i(\mathbf{H})$ is the corresponding eigenvalue of $\mathbf{A}$. Therefore, if $\lambda_{\max}(\mathbf{H}) > 2$, then

$$|1 - \lambda_{\max}(\mathbf{H})| > 1, \tag{A.28}$$

so the homogeneous update amplifies the associated eigendirection.
Step 3: Bound the spectrum by the learning rate. If $0 \le \eta_t \le \eta_0$, then $\mathrm{diag}(\boldsymbol{\eta}) \preceq \eta_0 \mathbf{I}$. Hence

$$\mathbf{H} = \mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})\mathbf{K} \preceq \eta_0 \mathbf{K}^\top \mathbf{K}, \tag{A.29}$$

which gives

$$\lambda_{\max}(\mathbf{H}) \le \eta_0 \lambda_{\max}(\mathbf{K}^\top \mathbf{K}) = \eta_0 \|\mathbf{K}\|_2^2. \tag{A.30}$$

Small-lr init therefore lowers the initial spectral scale of $\mathbf{H}$, making it less likely that $\mathbf{A}$ has eigenvalues with magnitude greater than one. This is the spectral counterpart of the chunk-scale control in Proposition 1. □

Corollary 1 (Mean scaling and small η_0 control the initial update)**.** *Proposition 1 and Proposition 2 describe two related effects. At the start of outer-loop training, mean scaling with $\eta = O(1)$ and updates without mean scaling but with $\eta_0 = O(1/c)$ both produce an $O(1)$ chunk update scale.*

For MSE, small η_0 also reduces the spectral scale of $\mathbf{K}^\top \mathrm{diag}(\boldsymbol{\eta})\mathbf{K}$. This lowers the risk that the homogeneous update matrix has eigenvalues with magnitude greater than one. After training, mean scaling continues to divide every output gradient by c. In contrast, small η_0 only fixes the starting

value, so the predicted learning rate can grow or shrink per token as the outer representation learns. This matches the main-text setting, which uses small-lr init without mean scaling.

Small-lr init sets b to a negative value chosen so that the initial learning rate is η_0. Concretely,

$$b = \log\left(\frac{p}{1-p}\right), \qquad p = \frac{\eta_0}{2}, \tag{A.31}$$

so that $2\sigma(b) = \eta_0$. With $\eta_0 = 10^{-3}$, this gives $p = 5 \times 10^{-4}$ and $b \approx -7.60$.

Small-lr init has two effects. First, it gives an $O(1)$ effective per-chunk update at initialization for the same reason as mean scaling. Both reduce the initial write scale by a factor proportional to $1/c$. With $\eta_{\text{init}} = 10^{-3}$ and $c = 256$, the scale factor is $c \cdot 10^{-3} = 0.256$.

Second, small η_0 does not lock the learning rate to this value. The learned projection can adjust η_t during outer-loop training. In contrast, mean scaling is a fixed division by c. This distinction matches the main-text choice of small-lr init without mean scaling, while Appendix D.1 reports runs with mean scaling as a complementary regime.

C.3 Loss analysis

The loss ablation separates the four losses into two groups. MSE and inner-product loss are competitive, while L1 and RMSE are weaker. The output-gradient formulas in Appendix A.3 show a simple mechanism behind this split: the losses differ in how much scale information their output gradients carry.

Proposition 3 (L1 and RMSE remove residual scale)**.** *Let $\mathbf{R} = \hat{\mathbf{V}} - \mathbf{V}$ be a nonzero residual. For any $a > 0$, L1 gives the same sign pattern for $\mathbf{R}$ and $a\mathbf{R}$, while RMSE gives the same normalized gradient direction for $\mathbf{R}$ and $a\mathbf{R}$. Thus both losses remove the magnitude information carried by the residual.*

Proof. For L1,

$$\operatorname{sign}(a\mathbf{R}) = \operatorname{sign}(\mathbf{R}) \tag{A.32}$$

for any $a > 0$ away from zero entries. The gradient depends on the sign pattern but not on the size of the error.
For RMSE,

$$\frac{a\mathbf{R}}{\sqrt{s\|a\mathbf{R}\|_F^2}} = \frac{\mathbf{R}}{\sqrt{s\|\mathbf{R}\|_F^2}}, \tag{A.33}$$

again for $a > 0$ and $\mathbf{R} \neq 0$. The gradient direction is normalized, so its norm is decoupled from the reconstruction error. □

This mechanism matches the main-text split. MSE and inner product preserve a scale-bearing write signal, whereas L1 and RMSE collapse or normalize that scale. The latter two losses are useful controls, but they are not the default choices in the experiments.

C.4 Multilayer fast learners

This subsection supports the main-text deep-memory ablation. The central issue is not only model expressivity. In the one-step Modular TTT setting, depth changes the geometry of the inner update.

The zero-initialization arguments below are scope statements about a boundary case. They explain why the framework should not use all-zero fast factors for deeper product-form learners. The main experiments in Table 6 use the Gaussian fast-weight initialization described in Appendix B.1, so their failures should be read as failures under nonzero initialization and one-step Modular TTT updates, not as direct consequences of the zero-init trap.

The shallow linear learner has a simple update:

$$\hat{\mathbf{V}} = \mathbf{K}\boldsymbol{W}, \qquad \Delta\boldsymbol{W} = \mathbf{K}^\top\mathbf{D}. \tag{A.34}$$

Under MSE, this is the update of a convex quadratic in one fast weight. A deeper fast learner no longer has this geometry. Its factor updates depend on activations produced by upstream factors and gradients backpropagated through downstream factors.

We use this factor-coupling view throughout the subsection. In this reduced appendix version, we keep the zero-initialization trap as a clean boundary case: when all fast factors are initialized at zero,

product-form depth can prevent any inner update from being written. This is not the mechanism behind the Gaussian-initialized main experiments, but it illustrates that depth changes the update geometry in ways absent from a single fast weight.

These product-form effects are closely related to classical analyses of deep linear networks and deep matrix factorization (Baldi & Hornik, 1989; Kawaguchi, 2016; Arora et al., 2018; 2019).

C.4.1 CORE UPDATE IDENTITY

Let $\mathbf{K} \in \mathbb{R}^{c\times d}$ be the chunk key matrix, $\mathbf{V} \in \mathbb{R}^{c\times e}$ be the target, and let $\mathbf{D}$ denote the effective chunk-normalized signal used by the TTT update. For a depth-L product-form linear learner, the train-view output is

$$\hat{\mathbf{V}} = \mathbf{K}\boldsymbol{W}_1\boldsymbol{W}_2\cdots\boldsymbol{W}_L. \tag{A.35}$$

For $L = 2$, we write $\boldsymbol{W}_1 \in \mathbb{R}^{d\times m}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{m\times e}$.

Proposition 4 (Product-form updates are factor-coupled)**.** *For the depth-L product-form learner, define*

$$\mathbf{A}_{j-1} = \mathbf{K}\boldsymbol{W}_1\cdots\boldsymbol{W}_{j-1}, \qquad \mathbf{B}_{j+1} = \boldsymbol{W}_{j+1}\cdots\boldsymbol{W}_L, \tag{A.36}$$

with $\mathbf{A}_0 = \mathbf{K}$ and $\mathbf{B}_{L+1} = \mathbf{I}$. The train-view update of the j-th factor has the form

$$\Delta\boldsymbol{W}_j = \mathbf{A}_{j-1}^\top \mathbf{D}\, \mathbf{B}_{j+1}^\top. \tag{A.37}$$

Thus the update of each factor is conditioned by the other factors on its forward-backward path.

Proof. **Step 1: Perturb one factor.** Holding all other factors fixed, a perturbation $\delta\boldsymbol{W}_j$ changes the output by

$$\delta\hat{\mathbf{V}} = \mathbf{A}_{j-1}\,\delta\boldsymbol{W}_j\,\mathbf{B}_{j+1}. \tag{A.38}$$

Step 2: Pair with the gradient. Using the Frobenius inner product,

$$\langle \mathbf{D}, \delta\hat{\mathbf{V}}\rangle = \langle \mathbf{A}_{j-1}^\top \mathbf{D}\, \mathbf{B}_{j+1}^\top, \delta\boldsymbol{W}_j\rangle. \tag{A.39}$$

Hence the update is $\Delta\boldsymbol{W}_j = \mathbf{A}_{j-1}^\top \mathbf{D}\, \mathbf{B}_{j+1}^\top$.

Step 3: Compare with one layer. When $L = 1$, the formula reduces to $\Delta\boldsymbol{W} = \mathbf{K}^\top\mathbf{D}$. For $L \geq 2$, the update contains upstream activations, downstream factors, or both. This is the factor coupling absent from the shallow linear learner. □

C.4.2 A BOUNDARY CASE: ZERO INITIALIZATION

The factor-coupled identity exposes a clean boundary case that separates shallow and product-form fast learners: all-zero initialization. This case is useful because it shows that depth can change whether the TTT update is written at all.

Proposition 5 (Zero initialization gives zero gradients in product-form linear learners)**.** *Consider a product-form linear fast learner with depth $L \geq 2$. If all fast factors are initialized at zero at sequence start, then the train-view gradient of every fast factor is exactly zero. Since the update is zero, the zero state persists across chunks unless it is externally perturbed.*

By contrast, a single linear fast learner with $\boldsymbol{W} = 0$ has update $\Delta\boldsymbol{W} = \mathbf{K}^\top\mathbf{D}$, which is nonzero whenever $\mathbf{K}$ and $\mathbf{D}$ are nonzero.

Proof. **Step 1: Apply the factor-coupled update identity.** By Proposition 4,

$$\Delta\boldsymbol{W}_j = \mathbf{A}_{j-1}^\top \mathbf{D}\, \mathbf{B}_{j+1}^\top. \tag{A.40}$$

If $L \geq 2$ and all factors are zero, then for every j either $\mathbf{A}_{j-1}$ contains a zero factor or $\mathbf{B}_{j+1}$ contains a zero factor. Hence $\Delta\boldsymbol{W}_j = 0$ for all j.

Step 2: Show persistence of the zero state. Since every factor update is zero, the fast state remains at zero after the chunk update. The same argument therefore applies again at the next chunk unless another mechanism perturbs the fast weights.

Step 3: Compare with the single-layer case. For a single linear fast learner, there is no second zero factor on the update path. At $\boldsymbol{W} = 0$, the update is

$$\Delta\boldsymbol{W} = \mathbf{K}^\top\mathbf{D}. \tag{A.41}$$

This is nonzero for generic nonzero $\mathbf{K}$ and $\mathbf{D}$. Thus zero initialization does not trap a single linear factor, but traps a product-form learner with at least two zero factors. □

This boundary case gives one mechanism behind the main-text deep-memory ablation. Once the fast learner is parameterized as a product of multiple trainable maps, the one-step TTT update is no

longer governed by the single-layer geometry. Even when Gaussian initialization avoids the zero-gradient trap used in this proposition, the update remains factor-coupled, which is the mechanism emphasized in the main text.

D Additional ablation results

D.1 Operating point: mean scaling, small-lr init, and decay

Table 13: Update-scale ablation for the linear fast learner at 2048 context. We compare standard init and small-lr init with and without mean scaling. The small-lr columns without mean scaling match the main ablation setting. Lower is better.

Loss	Decay	160M				410M			
		mean scaling		no mean scaling		mean scaling		no mean scaling	
		standard	small-lr init	standard	small-lr init	standard	small-lr init	standard	small-lr init
MSE	none	3.1935	3.2298	3.6012	3.2005	2.9178	2.9561	3.4036	2.9343
	scalar	3.0766	3.0791	3.3035	3.0380	2.8230	2.8139	3.0820	2.7949
	vector	3.0419	3.0506	3.2405	3.0038	2.7948	2.7975	3.0548	2.7821
Inner	none	3.2111	3.2314	3.2407	3.2028	2.9362	2.9601	2.9572	2.9273
	scalar	3.0730	3.0774	3.0538	3.0383	2.8253	2.8164	2.8054	2.7938
	vector	3.0402	3.0474	3.0033	3.0048	2.7983	2.7986	2.7793	2.7822

Table 13 separates two ways of controlling the inner-update scale. Mean scaling fixes the gradient scale by dividing by the chunk length, while small-lr init controls the initial learned learning rate and then lets the outer model adapt it. The standard-init rows without mean scaling are consistently worse, especially under MSE, which is the instability regime discussed in the main text. Once the scale is controlled, the decay ordering is stable: no decay is weakest, scalar decay gives most of the gain, and vector decay gives the lowest loss when it is available. This supports the main-text operating point: small-lr init without mean scaling, with scalar decay used as the default efficiency–quality trade-off.

D.2 Shallow operators

Table 14: Single-layer operator ablation at 2048 context. The columns without mean scaling match the main nonlinear-operator sweep, while the mean-scaling columns give standard-init controls. All rows use scalar decay; Norm denotes RMSNorm. Lower is better.

Size	Method	MSE		Inner-product	
		Mean scaling standard	No mean scaling small-lr init	Mean scaling standard	No mean scaling small-lr init
160M	Linear	3.0766	3.0380	3.0730	3.0383
	Linear + GELU	3.0872	3.0232	3.0859	3.0225
	Linear + SiLU	3.0867	**3.0205**	3.0744	3.0242
	Linear + Norm	3.0929	3.0300	3.0814	3.0335
410M	Linear	2.8230	2.7949	2.8253	2.7938
	Linear + GELU	2.8155	2.7844	2.8150	**2.7832**
	Linear + SiLU	2.8139	2.7849	2.8148	2.7838
	Linear + Norm	2.8511	2.8192	2.9009	2.7899

Table 14 shows why the main text treats the activation gain as a shallow-operator effect tied to the chosen scale regime. With mean scaling and standard init, GELU and SiLU stay close to the Linear row. Without mean scaling and with small-lr init, both activations consistently improve over Linear at 160M and 410M. Norm is more sensitive to loss and scale, so it is not used as the default shallow extension.

Robustness. We compare MSE with inner-product loss for the Linear learner and compare Linear with Linear-SiLU under MSE using paired seeds $\{1, 24, 42, 48, 84\}$. All other training and evaluation settings are fixed. Table 15 reports the final validation loss at step 20,000 for each run, together with the five-seed mean and sample standard deviation.

Table 15: Multi-run results using seeds $\{1, 24, 42, 48, 84\}$. Entries are final validation losses at step 20,000; the last column reports the mean $\pm$ sample standard deviation. Lower is better.

Scale	Setting	Seed 1	Seed 24	Seed 42	Seed 48	Seed 84	Mean $\pm$ std
160M	Linear, MSE	3.0431	3.0371	3.0380	3.0384	3.0339	3.0381 ± 0.0033
160M	Linear, inner-product loss	3.0413	3.0374	3.0383	3.0386	3.0338	3.0379 ± 0.0027
160M	Linear-SiLU, MSE	3.0202	3.0246	3.0205	3.0219	3.0215	3.0217 ± 0.0017
410M	Linear, MSE	2.7981	2.7965	2.7949	2.7954	2.7919	2.7954 ± 0.0023
410M	Linear, inner-product loss	2.7923	2.7940	2.7938	2.7951	2.7917	2.7934 ± 0.0014
410M	Linear-SiLU, MSE	2.7839	2.7953	2.7849	2.7953	2.7906	2.7900 ± 0.0055

The difference between MSE and inner-product loss is small and does not establish a consistent ordering across scales. Linear-SiLU outperforms its paired Linear-MSE run for all five seeds at both scales, supporting a consistent gain from the SiLU activation.

D.3 Normalization diagnostics

Table 16: Normalization and ϵ at 160M for `Linear + Norm`. Lower is better.

ϵ	Mean scaling + MSE	No mean scaling + MSE	No mean scaling + Inner
10^{-5}	3.0929	3.0735	3.0787
10^{-3}	3.0744	3.0631	3.0447
10^{-2}	3.0733	3.0300	3.0335
0.05	3.0467	3.0341	3.0392
0.1	3.0475	3.0381	3.0387
0.5	3.0967	3.0371	3.0377
1.0	3.0884	3.0412	3.0419

Normalization and ϵ. Table 16 confirms that normalization is strongly ϵ-dependent. Increasing ϵ can reduce this gradient amplification, but this is a stabilization effect rather than a robust quality improvement 21, 22, and 24. The best Norm rows without mean scaling remain behind the GELU and SiLU rows in Table 14, so the main text attributes the shallow gain to simple pointwise activations rather than normalization.

Table 17: `Linear + SiLU + Norm` at 2048 context. Rows use $\epsilon = 10^{-2}$, scalar decay, and small-lr init, without mean scaling. Lower is better.

	160M	410M
MSE	3.1543	$\times$
Inner	$\times$	$\times$

SiLU followed by normalization. Table 17 tests whether Norm composes with the SiLU gain. In this setting it does not: the only completed row is worse than both `Linear + SiLU` and `Linear + Norm`, while the other rows either diverge or stay in failed high-loss regimes. This supports keeping the shallow recommendation simple: one linear fast map followed by a pointwise activation.

Fast-weight normalization. Table 18 isolates the stability factors for fast-weight normalization. Without mean scaling, the tested rows fail; with mean scaling, the runs become trainable, but they remain worse than the strongest shallow activation rows in Table 14. These runs therefore serve as diagnostics for the normalization mechanism rather than candidates for the final model family.

D.4 Multilayer and gated memories

This subsection reports the deeper and gated-memory sweeps behind Table 6. The goal is to test whether a more expressive inner learner improves the one-step Modular TTT setting after the update scale has been stabilized. All rows use the 160M scale, 2048-token context, MSE loss, scalar decay, small-lr init, and Gaussian fast-weight initialization, unless a table states otherwise. The shallow reference is `Linear-SiLU`, with validation loss 3.0205.

We organize the evidence in three steps. First, Table 19 lists the best completed row from each deeper family. Second, Tables 20 and 21 give same-setting sweeps over depth, activation placement, mean scaling, and normalization. Third, Tables 24 and 25 report stability and initialization checks for residual and gated memories.

Table 18: Fast-weight normalization at 160M. We vary mean scaling, small-lr init, and ϵ for the same topology. Lower is better.

Mean scaling	Small-lr init	ϵ	MSE	Inner
no	yes	default	×	×
		0.25	×	×
		1.0	×	×
yes	no	default	3.2719	3.2724
		0.25	3.1942	3.2007
		1.0	3.1989	3.2060
yes	yes	default	3.1835	3.2046
		0.25	3.1831	**3.1683**
		1.0	3.1737	3.1695

Table 19: Best completed rows for deeper and gated memories at 160M. Δ is the validation-loss gap to the shallow `Linear-SiLU` reference, 3.0205. Lower is better.

Family	Topology	160M	Δ
Shallow reference	Linear-SiLU	3.0205	–
Product-form linear	Linear-Linear	3.1265	+0.1060
Activation-only	Linear-SiLU-Linear-SiLU	3.1156	+0.0951
Norm-containing	Linear-SiLU-Linear-Norm ($\epsilon = 0.5$)	**3.1144**	+0.0939
Residual	$f_{\text{res-norm}}$ ($\epsilon = 0.5$)	3.1643	+0.1438
Gated	SwiGLU + Norm	3.2041	+0.1836

Best completed rows. Table 19 gives the main result. Some deeper memories can be made trainable, but none improves over the shallow `Linear-SiLU` reference. The best deeper row is `Linear-SiLU-Linear-Norm` at $\epsilon = 0.5$, and it is still worse by 0.0939 validation loss.

Table 20: Two-layer linear and activation-only memories at 160M. Rows use MSE loss, scalar decay, small-lr init, and Gaussian fast-weight initialization. The grid varies SiLU placement and mean scaling. Lower is better.

Topology	Mean scaling	No mean scaling
Linear-Linear	3.1888	3.1265
Linear-Linear-SiLU	3.1876	3.1223
Linear-SiLU-Linear	3.1845	3.1240
Linear-SiLU-Linear-SiLU	3.1767	**3.1156**

Matched depth sweeps. Table 20 varies where SiLU is placed in a two-linear-layer memory. Rows without mean scaling are better than rows with mean scaling in this grid. Adding SiLU between the two linear maps or after the second map helps slightly, but the gain is small compared with the gap to `Linear-SiLU`. All eight cells remain behind the shallow reference. This matches the product-form analysis in Appendix C.4: adding fast factors changes the inner-update geometry, and a pointwise activation does not remove this coupling.

Table 21: Norm-containing two-layer memories at 160M. Rows use MSE loss, scalar decay, small-lr init, and Gaussian fast-weight initialization. The default is $\epsilon = 10^{-5}$. Lower is better.

Topology	Mean scaling	No mean scaling
Linear-Linear-Norm	3.2409	×
Linear-Norm-Linear	3.1858	**3.1594**
Linear-SiLU-Linear-Norm	3.2792	×
Linear-SiLU-Linear-Norm, $\epsilon = 0.5$	3.1144	3.1939
Linear-Norm-Linear-Norm	×	×

Table 22: Mean-scaling epsilon sweep for Linear-Norm-Linear-Norm at 160M. Rows use MSE loss, scalar decay, small-lr init, and Gaussian fast-weight initialization. Lower is better; $\times$ denotes divergence.

Setting	Mean scaling
Linear-Norm-Linear-Norm, $\epsilon = 0.1$	$\times$
Linear-Norm-Linear-Norm, $\epsilon = 0.25$	3.4962
Linear-Norm-Linear-Norm, $\epsilon = 0.5$	3.4755
Linear-Norm-Linear-Norm, $\epsilon = 0.75$	3.1750
Linear-Norm-Linear-Norm, $\epsilon = 1.0$	3.1637

Table 21 gives the matched comparison for norm-containing depth. All trainable rows remain behind the shallow reference. Increasing ϵ above the default 10^{-5} stabilizes some rows without mean scaling, but it does not close the gap: `Linear-SiLU-Linear-Norm` reaches 3.1144 at $\epsilon = 0.5$. For the deeper `Linear-Norm-Linear-Norm` case, the mean-scaling sweep in Table 22 reaches 3.1637 at $\epsilon = 1.0$.

Chunk-size robustness. We sweep $c \in \{128, 256, 512\}$ at 160M and 410M, keeping all training parameters fixed and varying only c within each learner graph. Table 23 reports the final validation loss at step 20,000.

Table 23: Chunk-size robustness at 160M and 410M. Within each learner graph, all training parameters are fixed and only the chunk size c changes. Entries are final validation losses at step 20,000; lower is better.

Learner graph	160M			410M		
	$c = 128$	$c = 256$	$c = 512$	$c = 128$	$c = 256$	$c = 512$
Linear-GELU	3.0224	3.0232	3.0247	2.7834	**2.7844**	**2.7838**
Linear-SiLU	**3.0210**	**3.0205**	**3.0217**	**2.7833**	2.7849	2.7845
Linear-Norm	3.0375	3.0300	3.0285	2.7961	2.8192	2.8066
Linear-SiLU-Linear-Norm	3.1373	3.1144	3.1132	2.9295	2.9258	2.9128

Linear-GELU and Linear-SiLU form the strongest group in all six settings. Linear-Norm remains behind these activation-only shallow learners, and the deeper `Linear-SiLU-Linear-Norm` learner remains behind every single-layer learner across all evaluated chunk sizes and model scales.

For residual memories, we test two graph forms. Let σ denote SiLU. The first applies an activation after the residual addition,

$$f_{\text{res-act}}(\boldsymbol{k}) = \sigma(\boldsymbol{k} + \sigma(\boldsymbol{k}\boldsymbol{W}_0)\boldsymbol{W}_1)\,, \tag{A.42}$$

and the second applies output RMSNorm after the same residual addition,

$$f_{\text{res-norm}}(\boldsymbol{k}) = \mathrm{RMSNorm}_\epsilon(\boldsymbol{k} + \sigma(\boldsymbol{k}\boldsymbol{W}_0)\boldsymbol{W}_1)\,. \tag{A.43}$$

These expressions correspond directly to the graph `input -> linear -> SiLU -> linear -> add(input, linear) -> final operator`.

Table 24: Residual and gated memories at 160M. Rows use MSE loss, scalar decay, small-lr init, and Gaussian fast-weight initialization. The residual rows should be compared with their non-residual counterparts: $f_{\text{res-act}}$ with Linear-SiLU-Linear-SiLU, and $f_{\text{res-norm}}$ with Linear-SiLU-Linear-Norm at matched ϵ. Lower is better.

Family	Setting	Mean scaling	No mean scaling
Baseline for Residual	Linear-SiLU-Linear-SiLU	3.1767	3.1156
	Linear-SiLU-Linear-Norm, $\epsilon = 10^{-5}$	3.2792	$\times$
	Linear-SiLU-Linear-Norm, $\epsilon = 0.5$	3.1144	3.1939
Residual	$f_{\text{res-act}}$	3.2367	$\times$
	$f_{\text{res-norm}}$, $\epsilon = 10^{-5}$	$\times$	$\times$
	$f_{\text{res-norm}}$, $\epsilon = 0.5$	3.2068	**3.1643**
SwiGLU	$-$	$\times$	$\times$
SwiGLU + Norm	$\epsilon = 10^{-5}$	$\times$	$\times$
	$\epsilon = 0.25$	3.2097	$\times$
	$\epsilon = 0.5$	3.2126	$\times$
	$\epsilon = 1.0$	**3.2041**	$\times$

Residual and gated checks. According to Table 6, unlike the chain-structured rows kept in the main text, these memories introduce additive or multiplicative branch interactions, so their stability

depends more strongly on mean scaling and RMSNorm ϵ. Table 24 shows that these extensions do not improve the controlled one-step TTT setting. The residual rows remain worse than their non-residual counterparts in Tables 20 and 21, while SwiGLU requires additional stabilization and still stays below the shallow reference. Thus, the residual and gated checks support the same conclusion as the main deep-memory table: adding branch structure increases stabilization sensitivity but does not close the gap to the shallow Linear-SiLU frontier.

Table 25: Fast-weight initialization ablation at 160M with MSE loss. Zero init sets all fast weights to zero; Gaussian init follows the official TTT initialization. Lower is better.

Topology	Zero init	Gaussian init
No mean scaling		
Single Linear	3.0343	3.0380
Linear-SiLU	3.0247	**3.0205**
Linear-Linear	×	3.1265
Linear-SiLU-Linear-SiLU	×	3.1156
SwiGLU + Norm	×	×
Mean scaling		
SwiGLU + Norm	×	3.2041

Table 25 connects the initialization study to Appendix C.4. Zero initialization works for the single linear and shallow `Linear-SiLU` rows, but it fails for the product-form deeper and gated rows. Gaussian initialization avoids this zero-gradient failure mode. Even with Gaussian initialization, however, deeper and gated memories still do not beat the shallow `Linear-SiLU` reference.

Together, these sweeps support the main-text conclusion: in the tested one-step Modular TTT setting, increasing the fast-learner depth or adding residual and gated structure does not improve over the shallow `Linear-SiLU` memory.

D.5 Baselines and related model comparisons

This appendix gathers the external-system checks used to interpret the main comparisons. These checks should not be read as isolated graph-memory ablations. The baseline implementations package several choices together: the official TTT code has separate handwritten `linear` and `mlp` learner paths, while the LaCT code combines sliding-window softmax attention, a SwiGLU-style fast learner, optional TTT branching, normalization, learning-rate prediction, and fused kernels inside one layer. This is precisely why the main paper uses Modular TTT for controlled ablations: in our implementation, the graph topology, loss, learning-rate rule, decay rule, and update schedule are exposed as separate design dimensions rather than being tied to a model-specific update path.

D.5.1 Official TTT comparison

Table 26: Official TTT reproduction baselines at 160M and 2048 context. Lower is better.

Configuration	Source	160M
Linear, no norm	Official TTT	3.3953
TTT-Linear (Linear + norm)	Official TTT	3.3951
TTT-MLP (Linear-GELU-linear + norm)	Official TTT	3.4061

Table 26 records the completed reproduction runs using the official TTT implementation (Sun et al., 2024). The official implementation is organized around two fixed learner classes. The linear path maintains a single fast matrix and bias, uses a layer-normalized L2 reconstruction target, and implements separate dual-form and cache-aware update logic. The MLP path repeats the same structure with two fast matrices, a GELU nonlinearity, explicit backward computations for both factors, and topology-specific state dictionaries. As a result, changing the learner family also changes the update derivation, normalization placement, and cache state layout.

This differs from Modular TTT. In our implementation, a graph is built from primitive nodes such as Linear, Act, Norm, Add, and Multiply; the same three-pass procedure runs train-view forward, train-view backward, and query-view forward for any valid graph. Loss choice and decay are selected independently of the graph. The official runs in Table 26 are therefore useful as reproduction baselines, but they are not component-matched ablations against a single Modular TTT setting.

The relatively high losses in Table 26 are consistent with this recipe mismatch. The official path uses a fixed reconstruction-style TTT objective with residual target $\mathbf{V} - \mathbf{K}$, a learned learning-rate gate scaled by the head dimension, no scalar/vector decay in these reproduction configs, and topology-specific normalization/update code. In the main controlled ablations, each of these dimensions matters: no decay is substantially worse than scalar decay, normalization is mixed, and deeper product-form learners are fragile. Thus the official rows combine several weaker choices under our training setup, rather than isolating a single disadvantage. The gap should therefore be read as evidence that a modular implementation can match or improve quality while exposing the relevant design knobs, not as a claim that the official code is a controlled lower bound for every TTT recipe.

D.5.2 Inference efficiency

We evaluated inference efficiency using the 160M and 410M model configurations. The model architectures and training configurations match the corresponding main experiments, and all models follow the same inference measurement protocol. Table 27 reports prefill throughput, decode throughput, and peak allocated device memory.

During autoregressive decoding, Modular TTT updates the fast-weight state only at chunk boundaries. Tokens in the current incomplete chunk still use all updates accumulated up to the current position, so the associated computation grows within the chunk and introduces additional decode overhead relative to GDN's token-wise recurrent update. We leave optimization of this decode path to future work.

Table 27: Inference at 160M and 410M. `M` denotes Modular TTT-Linear, `M-SiLU` denotes Modular TTT-Linear-SiLU, and Inner denotes inner-product loss. Higher throughput and lower peak allocated memory are better.

Scale	Model	Prefill (tokens/s)	Decode (tokens/s)	Peak allocated (GiB)
160M	GDN	700,983.9	1,821.36	1.634
	LaCT	264,403.2	113.69	2.154
	M (MSE)	449,824.0	790.45	1.500
	M-SiLU (MSE)	404,772.1	700.55	1.500
	M (Inner)	448,892.3	789.93	1.500
	M-SiLU (Inner)	401,684.6	685.03	1.500
410M	GDN	227,305.4	400.70	1.779
	LaCT	86,888.0	37.46	2.043
	M (MSE)	130,244.3	219.67	1.751
	M-SiLU (MSE)	116,602.9	192.31	1.689
	M (Inner)	128,931.6	219.13	1.689
	M-SiLU (Inner)	112,546.9	201.49	1.751

GDN provides the highest prefill and decode throughput at both scales. Every evaluated Modular TTT configuration is faster than the current LaCT implementation and uses less peak allocated memory than GDN and LaCT. Within Modular TTT, the Linear variants provide the highest throughput, and the choice between MSE and inner-product loss has little effect on inference efficiency.

D.5.3 LaCT comparison

Table 28: External systems check in a LaCT-style setting with sliding-window size 512 and chunk size 512. The table reports final validation loss. Lower is better. Absolute loss is comparable only within tokenizer-matched groups.

Configuration	160M	930M (official in paper)
LaCT SwiGLU, official tokenizer, no TTT	2.7769	2.4012
LaCT SwiGLU, official tokenizer, TTT path	2.7978	2.4009

We also repeated the comparison in the LaCT setting (Zhang et al., 2025), using sliding-window size 512 together with chunk size 512. The LaCT implementation first computes sliding-window softmax attention and then optionally adds a TTT branch. When the branch is enabled, the code reuses or separately projects Q/K/V for the fast learner, applies SiLU and L2 normalization to the fast views, predicts per-branch learning rates, and dispatches to either a linear or SwiGLU update path. The SwiGLU path uses three fast weights and the form

$$f(x) = W_1\big(\mathrm{SiLU}(W_0 x) \odot W_2 x\big),$$

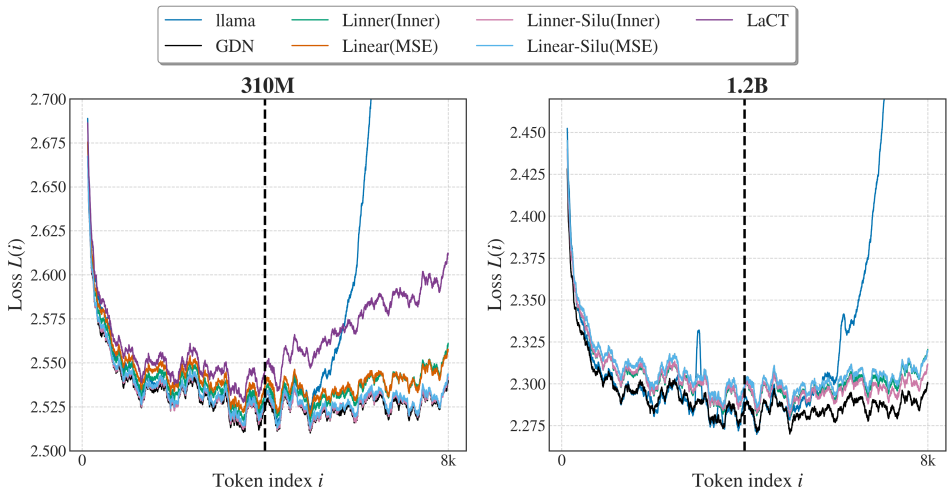

Figure 4: Per-token loss along an 8k sequence for representative 410M shallow Modular TTT variants. The dashed line marks the 4k training context used in the large-scale setting. Lower is better.

with optional momentum, Muon-style update normalization, fast-weight norm control, and fused Triton kernels. Thus LaCT changes the fast learner, local attention path, schedule, optimizer details, and normalization together.

Table 28 should therefore be interpreted as a boundary-regime systems check. With the official tokenizer and this strong sliding-window setting, disabling the TTT path is comparable to enabling it: the no-TTT variant is better at 160M, while the 930M official-scale rows are essentially tied. This does not contradict the main Modular TTT ablations. Rather, it shows that in this LaCT configuration the sliding-window softmax path carries much of the performance, so the result is not an isolated test of the SwiGLU graph learner. The controlled Modular TTT sweeps in the main text and Appendix D.4 are the relevant evidence for the claim that, under our one-step TTT operating point, deeper and gated graph learners do not improve over the shallow frontier.

D.6 Per-token loss diagnostics

Figure 4 examines how loss evolves along the sequence. The selected Modular TTT variants remain stable around and beyond the 4k training context, without the sharp post-context degradation observed in some baselines. This suggests that the shallow variants selected by the ablation are not only competitive in aggregate loss, but also behave robustly across token positions.

D.7 RULER evaluation details

We evaluate explicit retrieval with the needle-in-a-haystack (NIAH) tasks from RULER (Hsieh et al., 2024). The main text reports the `niah_single` family at 1k–8k as a compact view of direct single-key retrieval. This appendix gives the complete NIAH results used for that analysis, including `niah_multikey`, `niah_single`, `niah_multiquery`, and `niah_multivalue` at 1k–8k. For `niah_multikey` and `niah_single`, the family-level rows average the corresponding three templates; `niah_multiquery` and `niah_multivalue` are single task families.

The full results confirm the main-text conclusion. The selected Modular TTT variants are competitive with recurrent baselines on some short- and mid-context NIAH settings, but they do not match LLaMA on precise retrieval. The gap is largest at 4k–8k and on harder multi-key templates, where several models are near zero. Across Modular TTT variants, inner-product loss is often stronger for explicit retrieval at 410M, while the SiLU variants are less consistently beneficial than in language-modeling loss. These results suggest that the current shallow fast-weight memories improve efficient sequence modeling but are not yet reliable exact-recall mechanisms.

Table 29: Family-level `niah_single` RULER results at 410M and 1.45B. Entries are accuracies in percent. Higher is better.

Scale	Model	1k	2k	4k	8k
410M	LLaMA	97.93	93.33	80.47	51.87
	GDN	87.73	53.27	38.40	15.33
	LaCT	84.67	64.20	33.80	7.13
	M (MSE)	76.27	62.27	26.40	8.00
	M-SiLU (MSE)	74.33	50.67	25.27	13.60
	M (Inner)	81.67	55.73	28.93	14.07
	M-SiLU (Inner)	61.27	42.33	15.60	8.47
1.45B	LLaMA	99.00	94.13	99.20	92.13
	GDN	86.60	67.00	42.20	18.07
	M (MSE)	75.13	67.40	48.67	21.07
	M-SiLU (MSE)	88.60	64.93	52.87	15.00
	M (Inner)	88.93	79.87	50.00	18.20
	M-SiLU (Inner)	79.73	78.07	46.20	18.73

Table 30: Family-level RULER results for the three multi-NIAH families at 410M and 1.45B. Entries are accuracies in percent. Higher is better.

Scale	Family	Model	1k	2k	4k	8k
410M	`niah_multikey`	LLaMA	26.07	18.53	17.07	9.93
		GDN	7.80	7.67	7.27	3.73
		LaCT	14.13	9.53	6.60	3.73
		M (MSE)	8.33	7.20	6.93	3.33
		M-SiLU (MSE)	8.33	7.40	5.87	5.20
		M (Inner)	8.00	7.40	7.33	4.93
		M-SiLU (Inner)	9.53	7.53	7.20	4.67
	`niah_multiquery`	LLaMA	66.10	46.55	26.80	4.05
		GDN	22.35	20.25	14.10	6.05
		LaCT	25.35	20.95	10.80	3.50
		M (MSE)	24.25	21.55	17.25	6.40
		M-SiLU (MSE)	23.55	21.10	14.90	3.00
		M (Inner)	45.15	26.80	20.40	11.45
		M-SiLU (Inner)	28.45	23.85	18.75	7.65
	`niah_multivalue`	LLaMA	61.55	46.45	28.45	9.15
		GDN	30.60	20.00	17.15	9.00
		LaCT	29.70	21.60	12.05	6.40
		M (MSE)	33.55	24.90	16.70	5.30
		M-SiLU (MSE)	31.40	22.15	15.70	4.80
		M (Inner)	44.50	28.65	22.95	11.25
		M-SiLU (Inner)	33.75	22.40	18.65	11.95
1.45B	`niah_multikey`	LLaMA	43.47	34.80	30.80	20.13
		GDN	14.40	10.60	8.73	3.53
		M (MSE)	16.60	12.07	8.60	4.87
		M-SiLU (MSE)	12.53	9.67	7.40	3.07
		M (Inner)	15.20	12.33	8.20	3.93
		M-SiLU (Inner)	12.07	9.73	5.87	3.80
	`niah_multiquery`	LLaMA	89.20	85.25	65.40	21.95
		GDN	70.35	48.90	19.35	4.70
		M (MSE)	64.75	54.10	28.65	3.10
		M-SiLU (MSE)	50.20	30.20	17.15	4.75
		M (Inner)	73.50	49.05	20.75	8.00
		M-SiLU (Inner)	44.85	26.55	17.35	2.80
	`niah_multivalue`	LLaMA	85.10	72.70	52.20	22.35
		GDN	73.60	56.00	24.30	12.35
		M (MSE)	68.90	52.80	35.05	10.15
		M-SiLU (MSE)	51.85	30.05	18.10	5.90
		M (Inner)	76.60	52.15	20.05	6.95
		M-SiLU (Inner)	50.45	22.90	16.70	3.15

D.8 PRIMITIVE-LEVEL BACKWARD COMPUTATION

To isolate the cost of executing local backward rules, we compare the analytic backward operators for Linear and Norm with reference implementations based on `torch.autograd.grad` under identical inputs. The Linear and Norm input shapes are $(2, 128, 12, 128, 64)$ and $(2, 128, 12, 128)$, respectively. We use 20 warm-up iterations and 100 measured iterations.

Table 31: Microbenchmark of primitive-level backward computation. We report mean latency and maximum memory over 100 measured iterations after 20 warm-up iterations. Speedup is the autodiff-reference latency divided by the analytic-operator latency. Lower latency and memory are better.

Primitive	Implementation	Latency (ms)	Max mem. (MB)	Speedup
Linear	Analytic	**0.3127**	20.4	$1.65\times$
Linear	Autodiff ref.	0.5161	19.6	–
Norm	Analytic	**0.3146**	**19.3**	$2.62\times$
Norm	Autodiff ref.	0.8229	31.3	–

The analytic Linear and Norm operators yield $1.65\times$ and $2.62\times$ speedups over their autodiff references, respectively. Peak memory is comparable for Linear (20.4 versus 19.6 MB). For Norm, the analytic operator reduces peak memory from 31.3 to 19.3 MB. Under the evaluated software configuration, our inner-loop reference path based on `torch.autograd.grad` is incompatible with `torch.compile`. The custom analytic operators are included in the compiled graph.

D.9 END-TO-END THROUGHPUT

Table 32 compares Modular TTT with the official TTT implementation (Sun et al., 2024) in two representative 160M norm-containing settings. Modular TTT achieves a $2.2\times$–$3.3\times$ throughput improvement over the official implementation.

Table 32: Throughput validation in two representative 160M norm-containing settings on GPU. TTT-Linear and TTT-MLP follow the nomenclature of Sun et al. (2024). Modular TTT achieves about $2.2\times$–$3.3\times$ the training throughput of the official TTT implementation.

Setting	Official TTT	Modular TTT	Speedup
TTT-Linear (Linear + Norm)	43,397.5	93,366.9	$\approx 2.2\times$
TTT-MLP (Linear + GELU + Linear + Norm)	21,336.9	71,101.1	$\approx 3.3\times$

E LIMITATIONS

Limitations. Our study is limited to autoregressive language modeling under the training budgets and model scales reported in the experiments. Although Modular TTT exposes a broad set of graph, loss, learning-rate, decay, and normalization choices, it does not exhaust all possible inner learners or optimization schedules; in particular, deeper and gated learners may behave differently under different update rules, chunk sizes, optimizers, or hybrid attention designs. This scope also excludes system-specific update refinements used in LaCT, such as Muon-style update normalization and momentum variants, because they do not fit the current efficient fused graph implementation without adding specialized update paths; in our LaCT-style checks, these refinements did not produce a clear quality gain. Empirically, the selected shallow Modular TTT variants remain weaker than LLaMA on containment-style tasks and explicit long-context retrieval, indicating that fixed-state TTT still has limitations for precise recall. These limitations suggest that future work should study richer update schedules, better retrieval-oriented memory mechanisms, and broader evaluations beyond the language-modeling setting considered here.