TOASTER: NONLINEAR MEMORY FROM CASCADED LINEAR MEMORIES AND TARGET PROPAGATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Recurrent memories let language models process long sequences with a fixed-size state. Their design must balance the richness of the stored representation with the cost of updating it. Linear memories such as DeltaNet and KDA store one matrix and take one delta-rule step per token. A chunk of such steps has an exact WY form, so they train fast. On the other hand, deep memories such as TTT and Titans store a small MLP, which can hold maps that no matrix can. However, their update has no WY form. TOASTER bridges the two with one fact: a gradient step on a linear layer is a delta-rule write. It builds a two-layer memory from two linear memories with an activation between them. Target propagation gives the hidden memory its values by projecting the output error back through the output memory. Given a chunk's cross-layer signals, each layer can then run on an unmodified delta-rule kernel. Our default runs this kernel on the output layer and updates the hidden layer once per chunk. Running it on both layers gives the best model in our ablation. At matched backbones and training-token budgets, TOASTER has lower loss than KDA at all five tested sizes (0.2B–1.6B non-embedding parameters) on both C4 and PG19. Its C4 perplexity is also slightly lower than the attention baseline at every size. At the largest backbone, its mean accuracy on 13 three-shot tasks is 51.84%, compared with 50.52% for KDA. In length extrapolation, its loss stays below KDA's up to 20K tokens ($5\times$ the training length). On the 145M backbone, at a hidden width matched to Titans' memory, TOASTER takes at most approximately $1.33\times$ KDA's training-step time and is 2.8–$5.5\times$ faster than the tested Titans configurations at 32K context.

1 INTRODUCTION

Softmax attention (Vaswani et al., 2017) pays for its exactness with a key–value cache that grows with the context, and a large body of work now replaces some or all of it with a *recurrent memory*: a fixed-size state that is written to and read from as the sequence is processed (Behrouz et al., 2025b; Katharopoulos et al., 2020; Gu & Dao, 2024; Yang et al., 2024a; 2025). Because the state is bounded, two questions separate one memory from another: how much of the past it can pack into a given number of state parameters, and how cheaply it can be trained.

Linear memory trains fast but stores a linear map. The memories that have used in large production models (e.g., Gated DeltaNet in Qwen3-Next (Qwen Team, 2025), Kimi Delta Attention (KDA) in Kimi Linear and the 2.8T-parameter Kimi K3 (Kimi Team et al., 2025; Kimi Team, 2026)) are all *linear*: the state is a single matrix mapping keys to values, and each token applies one rank-one correction, a generalized Householder step (Schlag et al., 2021; Yang et al., 2024b). Products of such factors admit the compact *WY representation* (Bischof & Van Loan, 1987), which turns a whole chunk of sequential updates into a few dense matrix multiplications while remaining *exact*; this is why these models train at throughput competitive with attention at long context (Yang et al., 2024b; Yang & Zhang, 2024). Howevre, their memory at the end of the day is limited to linear mappings and hence its capacity is limited.

Nonlinear memory stores more but cannot use WY. A second line of work, rooted in test-time memorization (Sun et al., 2025; Behrouz et al., 2025b; 2026c;b), replaces the matrix with a small neural network whose weights are updated online by gradient descent on a reconstruction loss.

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.

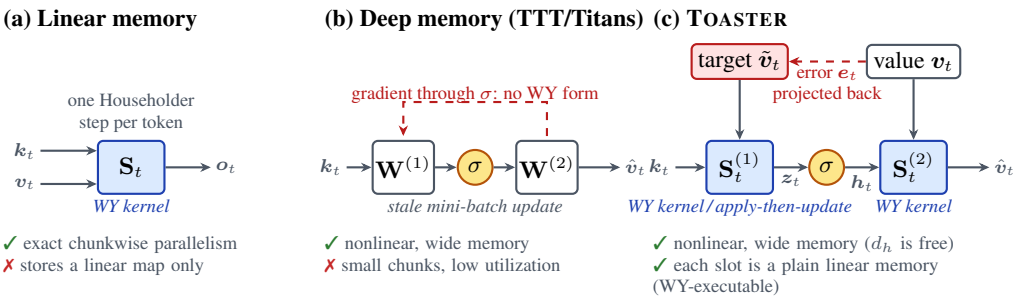


Figure 1: **TOASTER at a glance.** (a) A linear memory applies one Householder step per token, so a chunk of updates has an exact WY form. (b) A deep memory is more expressive, but its gradient update through the nonlinearity has no such form. (c) TOASTER cascades two linear memories around a pointwise activation σ. The output error $\boldsymbol{e}_t$ is projected back through the second memory into a target $\tilde{\boldsymbol{v}}_t$ for the first; both memories are then plain linear memories that the same WY kernel can advance (by default the first takes the cheaper apply-then-update step of Algorithm 1 instead), with the targets and hidden keys of a chunk computed once, in parallel, from the chunk-initial states.

Such a memory can store associations that no linear map can, but the nonlinearity destroys the Householder structure: no WY form exists, and training falls back to mini-batch gradient descent within a chunk, evaluated at a stale state, which is only accurate for small chunks and, at such chunks, leaves accelerators mostly idle (Zhang et al., 2026; Li et al., 2026). Recent work reduces the cost with very large chunks (Zhang et al., 2026) or hierarchical memories (Li et al., 2026) but keeps the stale mini-batch update. Table 1 summarizes the landscape: existing memories are WY-parallelizable as a whole or nonlinear; the closest two-pass slot memories (ABC, GSA; Appendix B) write their two memories independently.

This work. The two properties turn out not to be in conflict, for a reason that lies in the delta rule itself. A delta-rule write with key $\boldsymbol{k}_t$ and value $\boldsymbol{v}_t$ changes the state by $-\beta_t \boldsymbol{k}_t(\mathbf{S}_{t-1}^\top \boldsymbol{k}_t - \boldsymbol{v}_t)^\top$: a rank-one update whose left factor is the key and whose right factor is whatever residual is presented as the value. The gradient of any loss with respect to a linear layer with input $\boldsymbol{k}_t$ is also rank-one, $\boldsymbol{k}_t \boldsymbol{g}_t^\top$, so such a layer can be trained by an unmodified delta-rule kernel by handing it the value $\boldsymbol{z}_t - \boldsymbol{g}_t$, where $\boldsymbol{z}_t$ is its current output and $\boldsymbol{g}_t$ the error arriving there. TOASTER applies this to a two-layer memory network $\boldsymbol{k} \mapsto \mathbf{S}^{(2)\top}\sigma(\mathbf{S}^{(1)\top}\boldsymbol{k})$, i.e., two linear memories with a pointwise activation between them (Figure 1). The *output* memory $\mathbf{S}^{(2)}$ is already an ordinary linear memory, with key $\boldsymbol{h}_t = \sigma(\mathbf{S}^{(1)\top}\boldsymbol{k}_t)$ and value $\boldsymbol{v}_t$; the *hidden* memory $\mathbf{S}^{(1)}$ needs a value, and projecting the output error back through $\mathbf{S}^{(2)}$ supplies one—the construction known as target propagation (Lee et al., 2015). Once the hidden keys and targets of a chunk are computed, each layer is a standard linear memory that an unmodified WY kernel can advance: nonlinearity enters only through *what* is written, never through *how* each state evolves.[1]

Memory	Exact chunkwise (WY) form	Nonlinear state
Linear (delta rule): DeltaNet, Gated DeltaNet, KDA, Gated DeltaNet-2, Comba, DeltaProduct	✓	✗
Deep (test-time training): TTT, Titans, Atlas, LaCT, TNT	✗	✓
TOASTER (this work)	✓ per layer†	✓

Table 1: **Comparison between linear memory, deep memory, and TOASTER.** †Given the chunk's cross-layer signals; by default the output memory uses the kernel and the hidden memory a summed chunk step (Section 3.4).

Contributions. *(i)* We introduce TOASTER, a recurrent memory that cascades linear memories through pointwise activations and trains the hidden layers by target propagation (Section 3). With the

[1]Hence the name: two slots (linear memories) with heat (a nonlinearity) between them; in principle nothing prevents more slots.

exact target its per-token dynamics coincide with gradient descent on a two-layer memory network; given the cross-layer signals of a chunk, each layer can be advanced exactly by an existing WY kernel, so a chunk costs at most two kernel calls plus a few dense matrix multiplications. *(ii)* We describe three design choices (Section 3.4): two that stabilized training across our scaling ladder—a bias through homogeneous coordinates, and a bounded, rescaled back-projected error that also gives the hidden layer its smaller step—with the failure each addressed, and one made for throughput—a cheaper apply-then-update step for the hidden memory in place of its kernel call—and ablate all three at 145M (Table 12). *(iii)* Across a scaling ladder of five model sizes (0.2B–1.6B non-embedding parameters) and two corpora, TOASTER reaches lower language-modeling loss than KDA, the linear memory it is built on, in all ten settings, matches or slightly improves on a softmax-attention Transformer in-domain, and has the highest average of the three on a 13-task three-shot downstream suite (a 2.6% relative improvement over KDA). In length extrapolation, its loss stays below KDA's up to 20K tokens, five times the training length (Section 4). *(iv)* At a state matched to Titans ($d_h = d_k$), TOASTER's training step is within $1.33\times$ of KDA's and 2.8–$5.5\times$ faster than Titans at 32K (Figure 4). Related work is discussed in Appendix B.

2 BACKGROUND: TWO KINDS OF RECURRENT MEMORY

Given a token sequence with projections $\boldsymbol{q}_t, \boldsymbol{k}_t \in \mathbb{R}^{d_k}$ and $\boldsymbol{v}_t \in \mathbb{R}^{d_v}$, a recurrent memory maintains a state that is *written* with $(\boldsymbol{k}_t, \boldsymbol{v}_t)$ and *read* with $\boldsymbol{q}_t$; a unifying view (Sun et al., 2025; Behrouz et al., 2025b; Wang et al., 2025) treats writing as one step of online optimization of a reconstruction loss inside the forward pass. The two families below differ only in what the state is.

Linear memory and the WY representation. A linear memory stores a matrix $\mathbf{S}_t \in \mathbb{R}^{d_k \times d_v}$, reads with $\boldsymbol{o}_t = \mathbf{S}_t^{\mathsf{T}} \boldsymbol{q}_t$, and writes by a gradient step on $\mathcal{L}_t(\mathbf{S}) = \frac{1}{2}\|\mathbf{S}^{\mathsf{T}}\boldsymbol{k}_t - \boldsymbol{v}_t\|^2$ with step size $\beta_t \in (0, 1]$, optionally preceded by a forget gate α_t:

$$\mathbf{S}_t \;=\; \alpha_t\big(\mathbf{I} - \beta_t \boldsymbol{k}_t \boldsymbol{k}_t^{\mathsf{T}}\big)\,\mathbf{S}_{t-1} \;+\; \beta_t\,\boldsymbol{k}_t \boldsymbol{v}_t^{\mathsf{T}}. \tag{1}$$

With $\alpha_t \equiv 1$ this is DeltaNet (Schlag et al., 2021); a scalar data-dependent α_t gives Gated DeltaNet (Yang et al., 2025), and a channel-wise gate $\mathrm{Diag}(\boldsymbol{\alpha}_t)$, applied before the Householder factor, gives KDA (Kimi Team et al., 2025) (Appendix E). What matters for training is that Eq. (1) is *affine in the state*: the factor $\mathbf{I} - \beta_t \boldsymbol{k}_t \boldsymbol{k}_t^{\mathsf{T}}$ is a generalized Householder matrix, and over a chunk of C tokens the product of the C factors has the compact WY form $\mathbf{I} - \sum_{r=1}^{C} \boldsymbol{k}_r \boldsymbol{w}_r^{\mathsf{T}}$, with auxiliary vectors $\boldsymbol{w}_r$ obtained from a small triangular solve (Bischof & Van Loan, 1987; Joffrain et al., 2006). As a result, the state after the chunk and all C outputs can be computed *exactly* from the state before the chunk with a few dense matrix multiplications (Yang et al., 2024b); the recurrence is only sequential across chunks. We write this chunkwise kernel as

$$\big(\mathbf{S}_{[j+1]},\ \mathbf{O}_{[j]}\big) \;=\; \textsc{LinMem}\big(\mathbf{S}_{[j]};\ \mathbf{K}_{[j]}, \mathbf{V}_{[j]}, \mathbf{Q}_{[j]}\big), \tag{2}$$

where $\mathbf{K}_{[j]}, \mathbf{Q}_{[j]} \in \mathbb{R}^{C \times d_k}$ and $\mathbf{V}_{[j]}, \mathbf{O}_{[j]} \in \mathbb{R}^{C \times d_v}$ stack the keys, queries, values, and outputs of chunk j, $\mathbf{S}_{[j]}$ is the state entering the chunk, and the gates are passed alongside the keys. Efficient implementations of LINMEM exist for all the gated variants above (Yang & Zhang, 2024); Appendix E gives a self-contained derivation. The limitation is capacity: whatever it has seen, a linear memory represents a single linear map $\boldsymbol{k} \mapsto \mathbf{S}^{\mathsf{T}}\boldsymbol{k}$ with $d_k d_v$ numbers.

Deep memory and why the WY form is lost. Test-time-training memories (Sun et al., 2025; Behrouz et al., 2025b; 2026b) replace the matrix by a neural network $\mathcal{M}(\cdot\,; \mathcal{W}_t)$, typically a two-layer MLP, updated by gradient descent on the same reconstruction loss and read by a forward pass, $\boldsymbol{o}_t = \mathcal{M}(\boldsymbol{q}_t; \mathcal{W}_t)$. The stored map is no longer linear, and the capacity is set by the width of the network rather than by $d_k d_v$. But the gradient with respect to the first-layer weights passes through the activation, so the update is not affine in the state and no WY form exists. The standard remedy evaluates all gradients of a chunk at the *frozen* chunk-initial state (Sun et al., 2025; Behrouz et al., 2025b),

$$\mathcal{W}_{[j+1]} \;=\; \alpha_{[j]}\,\mathcal{W}_{[j]} \;-\; \sum_{r=1}^{C} \beta_{[j],r}\,\nabla_{\mathcal{W}} \mathcal{L}_{[j],r}\big(\mathcal{W}_{[j]}\big), \tag{3}$$

with per-token step sizes $\beta_{[j],r}$ and a chunk-level decay $\alpha_{[j]}$. This is parallel within the chunk but is a mini-batch approximation whose error grows with C. Implementations therefore use small chunks

(C between 16 and 64), pay a nonlinear forward and backward pass per gradient, and run at a small fraction of peak throughput (Zhang et al., 2026; Li et al., 2026). Equation (3) is also cruder than Eq. (2) in kind: the update is additive, so within a chunk no token's write can erase what an earlier token wrote. The next section shows that a deep memory can be rearranged so that *every layer* is a linear memory in its own right, recovering the exact WY form layer by layer at the price of computing the signals that pass between layers once per chunk.

3 TOASTER

TOASTER is a deep memory each of whose layers is a linear memory. We define the cascade (Section 3.1), show how target propagation turns the hidden layer into a linear-memory problem of its own (Section 3.2), give the chunkwise algorithm (Section 3.3), and describe the design choices that stabilized its training and make it fast at scale (Section 3.4). We present two layers; Appendix C treats N.

3.1 A CASCADE OF LINEAR MEMORIES

TOASTER keeps two matrix states, $\mathbf{S}_t^{(1)} \in \mathbb{R}^{d_k \times d_h}$ and $\mathbf{S}_t^{(2)} \in \mathbb{R}^{d_h \times d_v}$, each governed by a gated delta-rule recurrence of the form Eq. (1). The hidden width d_h is a free hyperparameter: it sets the capacity of the memory independently of the projection widths d_k and d_v. A key $\boldsymbol{k}_t$ is passed through the cascade,

$$\boldsymbol{z}_t = \mathbf{S}_{t-1}^{(1)\top} \boldsymbol{k}_t, \qquad \boldsymbol{h}_t = \sigma(\boldsymbol{z}_t), \qquad \hat{\boldsymbol{v}}_t = \mathbf{S}_{t-1}^{(2)\top} \boldsymbol{h}_t, \tag{4}$$

where σ is a pointwise activation. The composite map $\boldsymbol{k} \mapsto \mathbf{S}^{(2)\top}\sigma(\mathbf{S}^{(1)\top}\boldsymbol{k})$ is a two-layer MLP with weights $\{\mathbf{S}_t^{(1)}, \mathbf{S}_t^{(2)}\}$, i.e., the deep-memory parameterization of TTT and Titans. Reading uses the same cascade with the query and the updated states, $\boldsymbol{o}_t = \mathbf{S}_t^{(2)\top}\sigma(\mathbf{S}_t^{(1)\top}\boldsymbol{q}_t)$. The only question is how to *write*: how to update the two states so that the cascade as a whole learns to map $\boldsymbol{k}_t$ to $\boldsymbol{v}_t$.

3.2 TARGET PROPAGATION MAKES EVERY LAYER A LINEAR MEMORY

Both states should reduce the reconstruction loss $\mathcal{L}_t = \frac{1}{2}\|\hat{\boldsymbol{v}}_t - \boldsymbol{v}_t\|^2$. For the *output* layer this is already a linear-memory problem: $\mathbf{S}^{(2)}$ maps $\boldsymbol{h}_t$ to $\hat{\boldsymbol{v}}_t$, and the delta rule of Eq. (1) with key $\boldsymbol{h}_t$ and value $\boldsymbol{v}_t$ is exactly a gradient step on $\mathcal{L}_t$ with respect to $\mathbf{S}^{(2)}$.

The *hidden* layer has no value of its own. Backpropagating $\mathcal{L}_t$ through σ and $\mathbf{S}^{(2)}$ gives a gradient for $\mathbf{S}^{(1)}$, but as a function of the two states that update is nonlinear, so the WY machinery does not apply to it directly. Target propagation (Bengio, 2014; Lee et al., 2015) asks a different question: *what value should $\boldsymbol{z}_t$ have taken* for the loss to be smaller? We introduce a target $\tilde{\boldsymbol{v}}_t \in \mathbb{R}^{d_h}$ and give the hidden layer the local loss $\mathcal{L}_t^{(1)} = \frac{1}{2}\|\boldsymbol{z}_t - \tilde{\boldsymbol{v}}_t\|^2$, choosing $\tilde{\boldsymbol{v}}_t$ so that the local gradient equals the true one, $\nabla_{\boldsymbol{z}_t}\mathcal{L}_t^{(1)} = \nabla_{\boldsymbol{z}_t}\mathcal{L}_t$. A one-line chain rule (Appendix D) gives

$$\boxed{\tilde{\boldsymbol{v}}_t \;=\; \boldsymbol{z}_t \;-\; \big(\mathbf{S}_{t-1}^{(2)}\,\boldsymbol{e}_t\big) \odot \sigma'(\boldsymbol{z}_t), \qquad \boldsymbol{e}_t = \hat{\boldsymbol{v}}_t - \boldsymbol{v}_t.} \tag{5}$$

The target corrects the current pre-activation by the output error projected back through the output memory and gated by the activation slope. With $\tilde{\boldsymbol{v}}_t$ in hand, the hidden layer is updated by the delta rule with key $\boldsymbol{k}_t$ and value $\tilde{\boldsymbol{v}}_t$—the same rule as the output layer, and the same rule as any linear memory.

Proposition 1 (Per-token equivalence)**.** *Let both layers use scalar step sizes $\beta_t^{(1)}, \beta_t^{(2)}$, no forgetting, and the exact target (5), with $\boldsymbol{h}_t$ and $\tilde{\boldsymbol{v}}_t$ computed from the previous states. Updating $\mathbf{S}^{(1)}$ by Eq. (1) with $(\boldsymbol{k}_t, \tilde{\boldsymbol{v}}_t)$ and $\mathbf{S}^{(2)}$ by Eq. (1) with $(\boldsymbol{h}_t, \boldsymbol{v}_t)$ is identical to one step of gradient descent on $\mathcal{L}_t(\mathbf{S}^{(1)}, \mathbf{S}^{(2)})$ evaluated at $(\mathbf{S}_{t-1}^{(1)}, \mathbf{S}_{t-1}^{(2)})$.*

The proof is two lines (Appendix D): since $\nabla_{\mathbf{S}^{(1)}}\mathcal{L}_t = \boldsymbol{k}_t(\boldsymbol{z}_t - \tilde{\boldsymbol{v}}_t)^\top$, the delta-rule step toward $(\boldsymbol{k}_t, \tilde{\boldsymbol{v}}_t)$ *is* the gradient step, so token by token TOASTER with the exact target is a two-layer TTT/Titans memory trained by gradient descent. The joint transition of the two states is still nonlinear; what the rewriting buys is that $\boldsymbol{h}_t$ and $\tilde{\boldsymbol{v}}_t$ become *keys and values that the kernel consumes*, so that, conditioned

Algorithm 1 TOASTER, one chunk j. $\mathcal{N}$ is the bounded normalization of Section 3.4 (identity for the exact target) and $\boldsymbol{\beta}^{(1)}$ the hidden-layer step sizes; gates are omitted (passed to LINMEM with the keys, Appendix E; averaged over the chunk on line 6).

Require: chunk-initial states $\mathbf{S}^{(1)}_{[j]}, \mathbf{S}^{(2)}_{[j]}$; chunk inputs $\mathbf{K}_{[j]}, \mathbf{Q}_{[j]} \in \mathbb{R}^{C\times d_k}$, $\mathbf{V}_{[j]} \in \mathbb{R}^{C\times d_v}$

1: **Phase I (cross-layer signals, all rows in parallel):**

2: $\quad \mathbf{Z}_{[j]} \leftarrow \mathbf{K}_{[j]}\mathbf{S}^{(1)}_{[j]}$; $\quad \mathbf{H}_{[j]} \leftarrow \sigma(\mathbf{Z}_{[j]})$; $\quad \mathbf{E}_{[j]} \leftarrow \mathbf{H}_{[j]}\mathbf{S}^{(2)}_{[j]} - \mathbf{V}_{[j]}$

3: $\quad \mathbf{G}_{[j]} \leftarrow \mathcal{N}\big(\mathbf{E}_{[j]}\mathbf{S}^{(2)\mathsf{T}}_{[j]}\big) \odot \sigma'(\mathbf{Z}_{[j]})$; $\quad \widetilde{\mathbf{V}}_{[j]} \leftarrow \mathbf{Z}_{[j]} - \mathbf{G}_{[j]}$ $\quad \triangleright$ hidden errors and targets

4: **Phase II (hidden memory), one of:** $\quad \triangleright$ (b) is the default

5: $\quad$ (a) WY kernel: $\quad \big(\mathbf{S}^{(1)}_{[j+1]}, \mathbf{Z}^{q}_{[j]}\big) \leftarrow \text{LINMEM}\big(\mathbf{S}^{(1)}_{[j]};\ \mathbf{K}_{[j]}, \widetilde{\mathbf{V}}_{[j]}, \mathbf{Q}_{[j]}\big)$

6: $\quad$ (b) apply-then-update: $\quad \mathbf{Z}^{q}_{[j]} \leftarrow \mathbf{Q}_{[j]}\,\mathbf{S}^{(1)}_{[j]}$; $\quad \mathbf{S}^{(1)}_{[j+1]} \leftarrow \mathbf{S}^{(1)}_{[j]} - \mathbf{K}_{[j]}{}^{\mathsf{T}}\,\mathrm{Diag}(\boldsymbol{\beta}^{(1)})\,\mathbf{G}_{[j]}$

7: **Phase III (output memory):** $\big(\mathbf{S}^{(2)}_{[j+1]}, \mathbf{O}_{[j]}\big) \leftarrow \text{LINMEM}\big(\mathbf{S}^{(2)}_{[j]};\ \mathbf{H}_{[j]}, \mathbf{V}_{[j]}, \sigma(\mathbf{Z}^{q}_{[j]})\big)$

8: **return** $\mathbf{S}^{(1)}_{[j+1]}, \mathbf{S}^{(2)}_{[j+1]}, \mathbf{O}_{[j]}$

on them, each state's transition is a Householder product. This is the property the chunkwise algorithm exploits.

3.3 CHUNKWISE-PARALLEL TRAINING

Within a chunk of C tokens (Algorithm 1), Phase I computes the hidden keys $\mathbf{H}_{[j]}$ and targets $\widetilde{\mathbf{V}}_{[j]}$ of every token from the frozen chunk-initial states with three dense products; Phase II advances the hidden memory, by default with the apply-then-update step of line 6 (Section 3.4); and Phase III advances the output memory with one call of the chunkwise delta-rule kernel LINMEM. Given its keys and values, a call of LINMEM is exact for any chunk size. The approximations are that the cross-layer signals are evaluated once, at the chunk-initial states, and, by default, that the hidden layer sums its C writes as in Eq. (3) rather than applying them in sequence.

With no forgetting and the exact target, both update approximations vanish at $C = 1$; and the updates of Algorithm 1 reduce to Proposition 1, and both are first order in the within-chunk change of the states (Appendix D, which also gives the chunk update in closed form).

We use $C = 32$, as for the linear baseline. With $d_h = 4d_k$, a chunk costs about seven times the arithmetic of the underlying kernel (nine times with LINMEM on both layers), yet the memory kernels remain a small part of a block: 8–9% of its multiply–accumulate operations at 1.2B, against 1% for KDA (Appendix F). No new kernel is required: our JAX implementation shares one chunkwise routine (Appendix E) with the KDA baseline.

3.4 DESIGN CHOICES FOR STABILITY AND SPEED

The target of Eq. (5) is exact but assumes ideal conditions. Two modifications stabilized the training of TOASTER across our scaling ladder, and a third choice, made for throughput rather than stability, selects the update rule of the hidden memory. We describe each with the failure or cost it addressed in development, ablate all three at 145M (Table 12), and give the exact settings in Appendix G.

A bias through homogeneous coordinates. Linear memories are bias-free outer products, and a bias adds nothing to them. In the cascade, without a bias the pre-activations $\boldsymbol{z}_t = \mathbf{S}^{(1)\mathsf{T}}\boldsymbol{k}_t$ pass through the origin, and under ReLU-family activations many hidden units settle at zero—the dying-unit phenomenon (Lu et al., 2020). Because the state is rewritten at every token, the collapse is self-reinforcing: in early runs without a bias, the effective rank of $\mathbf{S}^{(1)}$ collapsed within a few hundred tokens, leaving a nearly linear cascade. We therefore absorb a bias into both states through homogeneous coordinates (Hartley & Zisserman, 2003), $\bar{\boldsymbol{k}}_t = [\boldsymbol{k}_t; 1]$ and $\bar{\boldsymbol{h}}_t = [\boldsymbol{h}_t; 1]$, so that $\mathbf{S}^{(1)}_t \in \mathbb{R}^{(d_k+1)\times d_h}$ and $\mathbf{S}^{(2)}_t \in \mathbb{R}^{(d_h+1)\times d_v}$. Each layer remains an unmodified linear memory with one extra input coordinate.

A bounded, rescaled back-projected error. In backpropagation the weights that transport the error change slowly; in TOASTER the transporting matrix $\mathbf{S}^{(2)}_{t-1}$ is rewritten at every token and its norm can compound over a long sequence, so the raw back-projection $\mathbf{S}^{(2)}_{t-1}\boldsymbol{e}_t$ produces targets whose scale drifts, and the hidden memory diverges. We pass the back-projected error through a bounded pointwise normalization before gating,

$$\tilde{\boldsymbol{v}}_t \;=\; \boldsymbol{z}_t \;-\; \mathcal{N}\big(\mathbf{S}^{(2)}_{t-1}\,\boldsymbol{e}_t\big) \odot \sigma'(\boldsymbol{z}_t), \qquad \mathcal{N}(x) = \gamma\ \mathrm{softsign}(x) = \frac{\gamma\, x}{1+|x|}, \tag{6}$$

with a fixed scale $\gamma = 0.01$. Each coordinate of the hidden correction is thus bounded by $\gamma \max_z |\sigma'(z)|$ (about 1.13γ for GELU), so γ also sets the effective step of the hidden layer relative to the output layer, which is written with ground-truth values; the hidden layer needs a much smaller step because its targets are back-projected through a d_h-wide state. Since softsign preserves the sign of each coordinate, the hidden update remains, per token and without forgetting, a descent direction for $\mathcal{L}_t$ although it is no longer the gradient (Appendix D). In preliminary runs, hard clipping and RMS or ℓ_2 normalization of the whole vector were less reliable across model sizes. Both memories otherwise inherit the input-dependent gates and step sizes of the underlying KDA recurrence, with scales set separately from the baseline's (Appendix G). The initial states $\mathbf{S}^{(1)}_0$ and $\mathbf{S}^{(2)}_0$ are learned, the recurrent mixers use no positional encoding, and inference runs the same chunked recurrence as training.

Apply-then-update for the hidden memory. The kernel on both layers is the form analyzed in Appendix D, and it is the better model: at 145M it lowers C4 perplexity by 0.05 (15.64 against 15.69, Table 12). It is also the slower one: in preliminary runs, training with the kernel on both layers was 1.3–1.4× slower per step than with the two dense products of line 6, which need no triangular solve. We use the same apply-then-update ordering as LaCT (Zhang et al., 2026) and TNT (Li et al., 2026): every query of a chunk reads the chunk-initial state, and the chunk's writes take effect at its end. We trade that difference for throughput: apply-then-update is the default for the hidden memory in every experiment unless stated otherwise, with the kernel for the output memory; Figure 4 uses these update rules at a smaller, state-matched hidden width (Appendix G.8).

4 EXPERIMENTS

Setup. We compare TOASTER with softmax attention (**Attention**), the linear memory **KDA** (Kimi Team et al., 2025), and the deep test-time-training memory **Titans** (Behrouz et al., 2025b). We do not include Gated DeltaNet (Yang et al., 2025), since KDA is a more expressive variant of it (channel-wise rather than scalar forget gates) that consistently outperforms it (Kimi Team et al., 2025). For a fair comparison, all models use the decoder-only backbone of NanoDO (Liu et al., 2024), with the attention block replaced by the corresponding sequence mixer, and are trained under identical conditions on C4 (Raffel et al., 2020) with a 4K-token context at five model sizes, labeled 145M to 1.2B (0.20B–1.58B non-embedding parameters, Table 5). Titans enters only the throughput comparison. Model sizes and token budgets, optimizer settings, and all architecture and mixer hyperparameters are given in Appendix G; evaluation settings are in Appendix G.5.

Language modeling across scales. Figure 2 summarizes language-modeling quality along the ladder. TOASTER has the lowest perplexity of the three architectures at every size on C4, up to and including 1.2B, and it is below KDA at every size on both corpora. The differences at 1.2B are small in absolute terms (0.02 perplexity over Attention and 0.06 over KDA on C4) but not trivial at this scale, where the three architectures follow nearly parallel power laws in compute (middle panel) and a few hundredths of perplexity separate them. The right panel makes this concrete: on C4, Attention needs 12% more compute than TOASTER to reach the same loss at 145M and 1% more at 1.2B, whereas KDA needs 2–4% more compute than Attention at every size. On PG19 the gain shrinks from 6% at 145M to −6% at 1.2B. We attribute this to capacity: attention keeps the entire context available, whereas the memory compresses it into a state whose size per head is fixed by d_h; as the models grow and the rest of the network improves, this fixed-size state is expected to become the limiting factor on long documents, so the memory capacity should be scaled together with the model, a direction we leave to future work. Compute is $6ND$, as in Kaplan et al. (2020): it leaves out the sequence mixers' own operations, whose share of a block falls as $1/D$ with width

Perplexity	145M	300M	550M	850M	1.2B
C4 (in-domain)					
Attention	15.92	13.52	11.94	10.78	9.96
KDA	16.00	13.57	11.97	10.83	10.00
TOASTER	**15.69**	**13.40**	**11.85**	**10.76**	**9.94**
PG19					
Attention	22.41	18.73	16.40	**14.52**	**13.19**
KDA	22.64	18.98	16.56	14.66	13.37
TOASTER	**22.21**	**18.68**	**16.37**	14.57	13.29

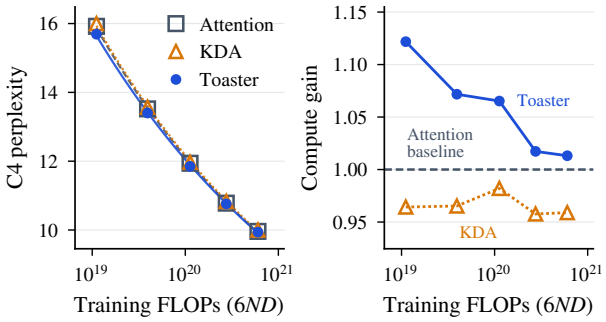


Figure 2: **Language modeling across the scaling ladder.** Left: perplexity on C4 and PG19 (best per column in bold). Middle: C4 perplexity against training compute ($6ND$ FLOPs). Right: compute-efficiency gain relative to Attention at matched $6ND$ compute. Details and all numbers are in Appendix G.2 and Appendix H.

Table 2: **Validation perplexity and downstream accuracy (%) at 1.2B.** Per-category averages and task details are in Appendix G.5 and Appendix H.

	Perplexity ↓		Downstream accuracy (%) ↑													
Model	C4	PG19	ARC-c	ARC-e	COPA	Hella.	OBQA	PIQA	SWAG	MultiRC	MathQA	CB	MNLI	RTE	SST-2	Avg.
Attention	9.96	**13.19**	30.64	60.68	67.00	**62.03**	**38.60**	75.14	61.20	57.16	21.70	50.00	31.93	53.79	49.08	50.69
KDA	10.00	13.37	31.16	61.06	70.00	61.33	38.40	75.19	**62.47**	57.12	21.68	50.00	31.99	47.29	49.08	50.52
TOASTER	**9.94**	13.29	**32.62**	**61.55**	**72.00**	61.23	38.40	**76.16**	62.17	**57.20**	**21.72**	**53.57**	**33.73**	**54.51**	49.08	**51.84**

(for TOASTER from 15% at 145M to 8% at 1.2B). Counting them raises the gains over Attention and offsets TOASTER's gain over KDA at these sizes (Appendix G.2).

Commonsense reasoning. Table 2 reports three-shot accuracy at 1.2B on 13 commonsense-reasoning and language-understanding benchmarks (Appendix G.5). TOASTER has the **highest** average (51.8, against 50.7 for Attention and 50.5 for KDA) and is at or above both baselines on 10 of the 13 tasks, with every exception below one point. Six tasks are near chance for all three models at this scale; on the other seven the averages are 57.7 for TOASTER, 57.1 for KDA, and 56.5 for Attention (Appendix G.5).

Length extrapolation on long sequences. All models are trained with a 4K context. Figure 3 evaluates them on 20K-token sequences from the long-book question-answering split of ∞Bench (Zhang et al., 2024a), five times the training length, reporting perplexity by token position averaged over 100 sequences (Appendix G.6). Attention is evaluated directly at the longer length, without any length-extrapolation technique such as position interpolation (Chen et al., 2023), YaRN (Peng et al., 2024), or ALiBi (Press et al., 2022); it degrades sharply beyond its training context, whereas both recurrent models stay flat. TOASTER's advantage over KDA is maintained over the entire range: beyond the first thousand positions its loss stays below KDA's at every position, by 0.008–0.014 nats averaged over each 4K window, including the final window (16–20K, 0.009 nats).

Table 3: **Single needle-in-a-haystack accuracy (%) at 1.2B**, averaged over 15 needle positions, by context length (Appendix G.7).

	S-NIAH-1 (passkey)				S-NIAH-2 (number)				S-NIAH-3 (UUID)			
Model	512	1K	2K	4K	512	1K	2K	4K	512	1K	2K	4K
Attention	100.0	100.0	100.0	96.3	100.0	**100.0**	**100.0**	**79.0**	**100.0**	**100.0**	**100.0**	**85.7**
KDA	100.0	100.0	100.0	**100.0**	100.0	99.0	88.7	37.3	98.0	96.0	77.7	28.0
TOASTER	100.0	100.0	100.0	**100.0**	100.0	99.3	90.3	62.7	98.0	97.0	73.7	27.3

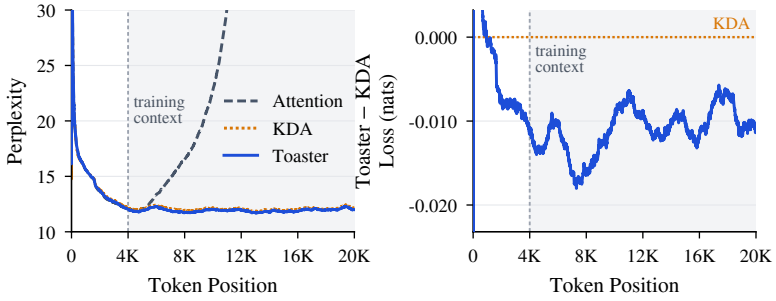


Figure 3: **Length extrapolation at 1.2B.** Left: perplexity by position. Right: TOASTER's loss minus KDA's (negative favors TOASTER). Curves are running averages over positions (Appendix G.6).

Needle in a haystack. We use the single-needle retrieval tasks of RULER (Hsieh et al., 2024), adapted and verified for our context lengths and tokenizer (Appendix G.7): a key–value pair (the needle) is hidden in a long context (the haystack) and the model must return the value given the key. S-NIAH-1 places a passkey in repeated synthetic text, and S-NIAH-2 and S-NIAH-3 place a number or a UUID in essay text, which tests whether the memory can retain the needle while storing irrelevant context. Table 3 reports accuracy averaged over 15 needle depths; the full pressure test, by depth and length, is in Appendix H. The recurrent models retrieve every passkey, and Attention retrieves at least 96%. In the essay haystacks, the recurrent models lose needles that are far from the query at 4K, whereas Attention, which keeps every token, retains 79–86%. TOASTER retains more than KDA on S-NIAH-2 (63% against 37% at 4K) and matches it on the UUID task (27% against 28% at 4K), whose random strings are the hardest content to keep in a fixed-size state.

Throughput comparison. TOASTER trains at close to the speed of the linear memory it is built from and, at long context, several times faster than a deep memory of the same size. Figure 4 reports the wall-clock time of one training step at a fixed budget of 512K tokens per step as the context grows from 2K to 32K (Appendix G.8). To compare training throughput at equal memory size, we match the state of the two deep memories: TOASTER and Titans are both timed at hidden width $d_h = d_k = 64$, and all three recurrent models use 64-dimensional heads. TOASTER is 1.2–1.8× faster than Titans at 2K and 2.8–5.5× faster at 32K, and the gap keeps widening with the context. Both memories evaluate a chunk's errors with one forward and backward pass at the chunk-initial state (Phase I of Algorithm 1 for TOASTER). TOASTER then advances its hidden memory with two dense products and its output memory with one call of the same chunkwise kernel as KDA (Section 3.3). Titans' step time grows several-fold from 2K to 32K although every step processes the same number of tokens, whereas TOASTER's stays within 1.33× of KDA's at every context length and is 1.27–1.55× Flash Attention's, which KDA itself trails up to 8K. This overhead is measured without any specialized kernel: TOASTER reuses the KDA kernel unchanged for its output memory and advances the hidden memory by the two dense products of apply-then-update (Section 3.4); a fused kernel for the cascade is left to future work.

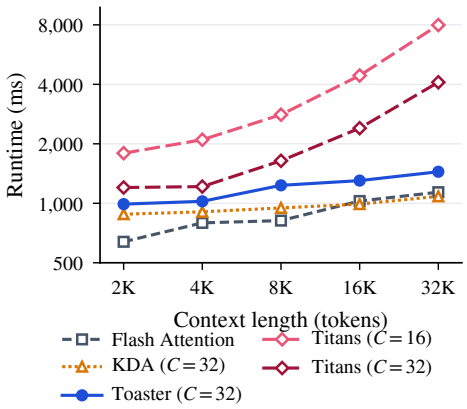


Figure 4: **Training-step time against context length** (145M, 512K tokens per step; state-matched hidden width $d_h = d_k = 64$; Appendix G.8).

Perplexity versus training speed. Figure 5 compares perplexity and per-step runtime at 145M (timed as in Figure 4). Titans evaluates all gradients of a chunk at the chunk-initial state (Eq. (3)), so its chunk size C trades quality for speed: larger chunks train faster but reach higher perplexity. At its best ($C = 16$), Titans is both slower and worse than TOASTER, and the chunk sizes that make it as fast as TOASTER leave it worse than KDA. TOASTER has the lowest perplexity of the three

Table 4: **Ablations at 145M**: C4 perplexity and its difference Δ to the default TOASTER (red: worse, blue: better); each column changes one component. Full results in Table 12 (Appendix G.9).

	Reference		State width		Activation σ		Hidden-layer learning		Bias
	KDA	TOASTER	$d_h = d_k$	$d_h = 2d_k$	identity	leaky ReLU	$\gamma = 0$	WY kernel	w/o
C4 ppl (↓)	16.00	15.69	15.90	15.76	15.80	15.71	15.71	15.64	15.86
Δ	+0.31	–	+0.22	+0.07	+0.11	+0.02	+0.02	−0.05	+0.17

architectures at a speed comparable to KDA's: nonlinear-memory quality at close to linear-memory speed.

Ablation study. Table 12 varies one component of TOASTER at a time at 145M (Appendix G.9); apart from the slower WY-kernel update of the hidden memory, no change lowers perplexity by more than 0.01. The 0.31 gain over KDA has three sources (Table 4): a larger state, from which a linear cascade with the same state (identity activation) already gains 0.20; the nonlinearity, which adds the remaining 0.11 (GELU is the best of five activations); and target propagation, which lets the hidden memory learn in context and helps most when that memory is advanced exactly by the WY kernel. Most of the gain thus comes from a wide nonlinear memory that runs on linear-memory kernels (full discussion in Appendix G.9).

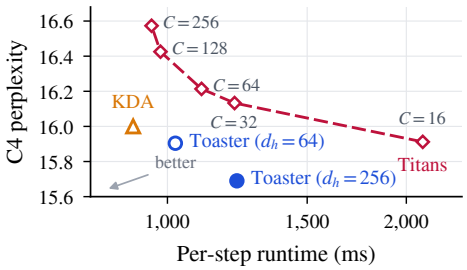


Figure 5: **Perplexity against per-step runtime** (145M, 4K context; C: chunk size).

5 CONCLUSION AND LIMITATIONS

TOASTER resolves an apparent conflict between two families of recurrent memory. By writing a two-layer memory as two linear memories with an activation in between, and giving the hidden memory its own targets through target propagation, it lets every layer keep a linear memory's Householder transition (by default only the output memory keeps it; the hidden memory takes the cheaper apply-then-update step) and confines the nonlinearity to the choice of what is written: with the exact target and no forgetting it is, token by token, gradient descent on a two-layer memory network, and chunk by chunk it is at most two calls to an unmodified linear-memory kernel. On a five-point scaling ladder it improves on the linear memory it is built from in every setting, by a margin that narrows with scale, and is on par with softmax attention in-domain. The recipe—keep every layer's transition affine, move the nonlinearity into the targets—applies in principle to any linear recurrence with a chunkwise kernel.

Limitations. Our evidence has clear boundaries. First, the efficiency evidence is an operation count (Appendix F) and one throughput profile (Figure 4), measured at a single model size on one platform, with no specialized kernel for TOASTER and no LaCT baseline; with $d_h = 4d_k$ TOASTER has about eight times the state and seven to nine times the kernel arithmetic of KDA, and the compute-efficiency gains of Figure 2 are per training token: by the operation counts of Appendix F, TOASTER costs 6–13% more per token than KDA (up to 16% with the kernel on both layers), which offsets its gain over KDA at equal compute (Table 7). Second, all training runs are single runs with one seed, the study stops at 1.6B non-embedding parameters, 63.7B tokens, and a 4,096-token context, and the KDA baseline uses 64-dimensional heads; larger scales, wider-head or state-matched linear baselines, and recall beyond 4K are untested. Third, the downstream evidence is thin: on the 13-task three-shot suite TOASTER has the best average by about a point, but we ran no per-task significance tests and six tasks are at or near chance for all three models. Fourth, the design choices of Section 3.4 are ablated only at 145M (Table 12), KDA has not been retuned with TOASTER's gate scales, and the effect of the chunk size on the approximation of Section 3.3 has not been isolated. Finally, we evaluated depth two only; deeper cascades (Appendix C) are untested.

AI USE STATEMENT

In this work, we used generative AI tools for paper drafting and editing, for literature summarization and topic discovery, for figure creation and the code that generates tables and plots from experiment logs, and for typesetting the derivations in Appendix D and Appendix E. We also used generative AI tools to help interpret the experimental results and to draft the corresponding text, to provide feedback on the experimental design, including which controls and ablations to run, to draft and check the statement and proof of Proposition 2, and for code editing in the evaluation pipeline. We have not used generative AI tools for research ideation, for the design of the method, for the mathematical formulation of the target-propagation update, for the implementation of the training and inference code, for synthetic data generation, or to run experiments; translation, data cleaning, and qualitative analysis are not applicable to this work. We have reviewed all AI-assisted work: every derivation was checked by hand by the authors, the numbers in the tables and figures are generated from the experiment logs by scripts that the authors inspected and the numbers quoted in the text were checked against them, all related-work claims were verified against the cited papers, and all AI-assisted text was revised by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Ehsan Amid, Rohan Anil, and Manfred K. Warmuth. LocoProp: Enhancing BackProp via local loss optimization. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2022. arXiv:2106.06199.

Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. MathQA: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, 2019. arXiv:1905.13319.

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 (NeurIPS)*, 2016. arXiv:1610.06258.

Ali Behrouz, Meisam Razaviyayn, Peilin Zhong, and Vahab Mirrokni. Nested learning: The illusion of deep learning architectures. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025a. arXiv:2512.24695.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025b. arXiv:2501.00663.

Ali Behrouz, Zeman Li, Yuan Deng, Peilin Zhong, Meisam Razaviyayn, and Vahab Mirrokni. Memory caching: RNNs with growing memory. In *International Conference on Machine Learning (ICML)*, 2026a. arXiv:2602.24281.

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. In *International Conference on Machine Learning (ICML)*, 2026b. arXiv:2505.23735.

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. In *International Conference on Learning Representations (ICLR)*, 2026c. arXiv:2504.13173.

Yoshua Bengio. How auto-encoders could provide credit assignment in deep networks via target propagation. *arXiv preprint arXiv:1407.7906*, 2014.

Christian Bischof and Charles Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13, 1987. doi: 10.1137/0908009.

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. arXiv:1911.11641.

Tommaso Cerruti, Tim Rieder, George Rowlands, Lingfeng Jin, and Imanol Schlag. Linear attention architectures: Mechanisms, trade-offs, and cross-layer routing. *arXiv preprint arXiv:2607.07953*, 2026.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

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.

Ido Dagan, Oren Glickman, and Bernardo Magnini. The PASCAL recognising textual entailment challenge. In *Machine Learning Challenges Workshop (MLCW)*, volume 3944 of *Lecture Notes in Computer Science*, pp. 177–190. Springer, 2005.

Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning (ICML)*, 2024. arXiv:2405.21060.

Marie-Catherine de Marneffe, Mandy Simons, and Judith Tonhauser. The CommitmentBank: Investigating projection in naturally occurring discourse. In *Proceedings of Sinn und Bedeutung*, volume 23, 2019.

Maxence Ernoult, Fabrice Normandin, Abhinav Moudgil, Sean Spinney, Eugene Belilovsky, Irina Rish, Blake Richards, and Yoshua Bengio. Towards scaling difference target propagation by learning backprop targets. In *International Conference on Machine Learning (ICML)*, 2022. arXiv:2201.13415.

Guhao Feng, Shengjie Luo, Kai Hua, Ge Zhang, Di He, Wenhao Huang, and Tianle Cai. In-place test-time training. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2604.06169.

Thomas Frerix, Thomas Möllenhoff, Michael Moeller, and Daniel Cremers. Proximal backpropagation. In *International Conference on Learning Representations (ICLR)*, 2018. arXiv:1706.04638.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling (COLM)*, 2024. arXiv:2312.00752.

Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2nd edition, 2003. doi: 10.1017/CBO9780511811685.

Ali Hatamizadeh, Yejin Choi, and Jan Kautz. Gated DeltaNet-2: Decoupling erase and write in linear attention. *arXiv preprint arXiv:2605.22791*, 2026.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. arXiv:2203.15556.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling (COLM)*, 2024. arXiv:2404.06654.

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

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 (NeurIPS)*, 2021. arXiv:2106.06295.

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 *International Conference on Machine Learning (ICML)*, 2022. arXiv:2202.05798.

Thierry Joffrain, Tze Meng Low, Enrique S. Quintana-Ortí, Robert van de Geijn, and Field G. Van Zee. Accumulating Householder transformations, revisited. *ACM Transactions on Mathematical Software*, 32(2):169–179, 2006. doi: 10.1145/1141885.1141886.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning (ICML)*, 2020. arXiv:2006.16236.

Daniel Khashabi, Snigdha Chaturvedi, Michael Roth, Shyam Upadhyay, and Dan Roth. Looking beyond the surface: A challenge set for reading comprehension over multiple sentences. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, 2018.

Kimi Team. Kimi K3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026. Technical report.

Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, et al. Kimi Linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025.

Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP): System Demonstrations*, 2018. arXiv:1808.06226.

Aakash Lahoti, Kevin Y. Li, Berlin Chen, Caitlin Wang, Aviv Bick, J. Zico Kolter, Tri Dao, and Albert Gu. Mamba-3: Improved sequence modeling using state space principles. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2603.15569.

Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, volume 9284 of *Lecture Notes in Computer Science*, pp. 498–515. Springer, 2015. doi: 10.1007/978-3-319-23528-8_31. arXiv:1412.7525.

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. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2511.07343.

Junchen Liu, Sven Elflein, Or Litany, Zan Gojcic, and Ruilong Li. Test-time training with KV binding is secretly linear attention. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2602.21204.

Peter J. Liu, Roman Novak, Jaehoon Lee, Mitchell Wortsman, Lechao Xiao, Katie Everett, Alexander A. Alemi, Mark Kurzeja, Pierre Marcenac, Izzeddin Gur, Simon Kornblith, Kelvin Xu, Gamaleldin Elsayed, Ian Fischer, Jeffrey Pennington, Ben Adlam, and Jascha Sohl-Dickstein. NanoDO: A minimal Transformer decoder-only language model implementation in JAX. https://github.com/google-deepmind/nanodo, 2024. Version 0.1.0.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019. arXiv:1711.05101.

Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying ReLU and initialization: Theory and numerical examples. *Communications in Computational Physics*, 28(5):1671–1706, 2020. doi: 10.4208/cicp.OA-2020-0165. arXiv:1903.06733.

Alexander Meulemans, Francesco S. Carzaniga, Johan A. K. Suykens, João Sacramento, and Benjamin F. Grewe. A theoretical framework for target propagation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. arXiv:2006.14331.

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 (EMNLP)*, 2018. arXiv:1809.02789.

Tsendsuren Munkhdalai, Alessandro Sordoni, Tong Wang, and Adam Trischler. Metalearned neural memory. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. arXiv:1907.09720.

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. In *Second Conference on Language Modeling (COLM)*, 2025. arXiv:2503.14456.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window extension of large language models. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2309.00071.

Hao Peng, Jungo Kasai, Nikolaos Pappas, Dani Yogatama, Zhaofeng Wu, Lingpeng Kong, Roy Schwartz, and Noah A. Smith. ABC: Attention with bounded-memory control. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2022. arXiv:2110.02488.

Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations (ICLR)*, 2022. arXiv:2108.12409.

Qwen Team. Qwen3-Next: Towards ultimate training & inference efficiency. https://huggingface.co/Qwen/Qwen3-Next-80B-A3B-Instruct, September 2025. Model card and blog post.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations (ICLR)*, 2020. arXiv:1911.05507.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. arXiv:1910.10683.

Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S. Gordon. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, 2011.

Vincent Roulet and Zaid Harchaoui. Target propagation via regularized inversion for recurrent neural networks. *Transactions on Machine Learning Research*, 2023.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning (ICML)*, 2021. arXiv:2102.11174.

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

Tatsukichi Shibuya, Nakamasa Inoue, Rei Kawakami, and Ikuro Sato. Fixed-weight difference target propagation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 9811–9819, 2023. doi: 10.1609/aaai.v37i8.26171. arXiv:2212.10352.

Julien Siems, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. DeltaProduct: Improving state-tracking in linear RNNs via Householder products. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2502.10297.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. doi: 10.1016/j.neucom.2023.127063. arXiv:2104.09864.

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. In *International Conference on Machine Learning (ICML)*, 2025. arXiv:2407.04620.

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.

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.

Bohao Tang, Zhen Qin, Yuqi Pan, Zheng Li, Pengfei Liu, and Ya Zhang. Modular TTT: Rethinking test-time training as composable modules. *arXiv preprint arXiv:2608.07110*, 2026.

Neehal Tumma, Noel Loo, and Daniela Rus. Preconditioned DeltaNet: Curvature-aware sequence modeling for linear recurrences. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2604.21100.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. arXiv:1706.03762.

Johannes von Oswald, Nino Scherrer, Seijin Kobayashi, Luca Versari, Songlin Yang, Sarthak Mittal, Maximilian Schlegel, Kaitlin Maile, Yanick Schimpf, Oliver Sieberling, Alexander Meulemans, Rif A. Saurous, Guillaume Lajoie, Charlotte Frenkel, Razvan Pascanu, Blaise Agüera y Arcas, and João Sacramento. MesaNet: Sequence modeling by locally optimal test-time training. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2506.05233.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. SuperGLUE: A stickier benchmark for general-purpose language understanding systems. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019a. arXiv:1905.00537.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations (ICLR)*, 2019b. arXiv:1804.07461.

Ke Alexander Wang, Jiaxin Shi, and Emily B. Fox. Test-time regression: A unifying framework for designing sequence models with associative memory. *arXiv preprint arXiv:2501.12352*, 2025.

Adina Williams, Nikita Nangia, and Samuel R. Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, 2018. arXiv:1704.05426.

Songlin Yang and Yu Zhang. FLA: A Triton-based library for hardware-efficient implementations of linear attention mechanism. https://github.com/fla-org/flash-linear-attention, 2024.

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 (ICML)*, 2024a. arXiv:2312.06635.

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 (NeurIPS)*, 2024b. arXiv:2406.06484.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated Delta Networks: Improving Mamba2 with delta rule. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2412.06464.

Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. SWAG: A large-scale adversarial dataset for grounded commonsense inference. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018. arXiv:1808.05326.

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 (ACL)*, 2019. arXiv:1905.07830.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. arXiv:1910.07467.

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. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2505.23884.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Khai Hao, Xu Han, Zhen Leng Thai, Shuo Wang, Zhiyuan Liu, and Maosong Sun. ∞Bench: Extending long context evaluation beyond 100K tokens. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 15262–15277, 2024a. arXiv:2402.13718.

Yu Zhang, Songlin Yang, Ruijie Zhu, Yue Zhang, Leyang Cui, Yiqiao Wang, Bolun Wang, Freda Shi, Bailin Wang, Wei Bi, Peng Zhou, and Guohong Fu. Gated slot attention for efficient linear-time sequence modeling. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024b. arXiv:2409.07146.

Zeyun Zhong, Joya Chen, Manuel Martin, Frederik Diederichs, Juergen Gall, and Juergen Beyerer. Rethinking expressivity and efficiency in test-time training. *arXiv preprint arXiv:2608.21308*, 2026.

A STATEMENTS

A.1 REPRODUCIBILITY STATEMENT

All architectural details needed to implement TOASTER are given in Section 3 (forward pass, target, and chunkwise algorithm) and Appendix C (deeper cascades), with the target derivation and the per-token equivalence proof in Appendix D and the WY algorithm behind the kernel in Appendix E. Appendix G lists the complete scaling ladder (depth, width, heads, batch size, learning rate, schedule, and token budget per size), the optimizer settings, the shared backbone, the mixer-specific settings of KDA and TOASTER, and the tokenizer; Appendix G.5 describes the three-shot downstream evaluation protocol; and Appendix H reports every number behind the figures and summary tables of the main text. Source code for the TOASTER layer and the evaluation configuration will be released upon publication.

A.2 ETHICS STATEMENT

This work studies the architecture of language-model memory, trains models on the public web corpus C4, and evaluates them on C4, PG19, ∞Bench, and public benchmark suites. It involves no human subjects, and it neither creates nor releases new datasets. The risks and benefits of the resulting models are those of language models trained on public web text in general; more efficient long-context memory lowers the cost of such models and may broaden both beneficial and harmful uses in equal measure. We are not aware of ethical concerns specific to this work.

B RELATED WORK

Linear recurrent memories. Linear attention (Katharopoulos et al., 2020) and its gated and state-space descendants (Sun et al., 2023; Yang et al., 2024a; Gu & Dao, 2024; Dao & Gu, 2024; Peng et al., 2025; Lahoti et al., 2026) maintain a matrix state updated by an outer-product write. The delta rule replaces the pure write with an erase-and-write Householder step (Schlag et al., 2021), and Yang et al. (2024b) showed that the WY representation (Bischof & Van Loan, 1987; Joffrain et al., 2006) yields an exact, matmul-rich chunkwise algorithm; Gated DeltaNet (Yang et al., 2025), KDA (Kimi Team et al., 2025), Gated DeltaNet-2 (Hatamizadeh et al., 2026), Comba (Hu et al., 2025), DeltaProduct (Siems et al., 2025), preconditioned variants (Tumma et al., 2026), and MesaNet (von Oswald et al., 2026) refine the gate, rank, geometry, or optimality of this step while keeping a chunkwise form. These memories now run in production hybrids that interleave three linear layers with one softmax-attention layer: Qwen3-Next with Gated DeltaNet (Qwen Team, 2025), and Kimi Linear and the 2.8T-parameter Kimi K3 with KDA (Kimi Team et al., 2025; Kimi Team, 2026). Notably, all of K3's recurrent layers are linear delta-rule memories computed with WY/UT-transform kernels and the report contains no nonlinear memory.

Two-pass slot memories and deep fast-weight memories. ABC (Peng et al., 2022) bounds attention's memory to a fixed number of slots, storing two matrices that are read through a softmax over the slot keys, and GSA (Zhang et al., 2024b) gates the slot updates so that, in the authors' words, it "comprises a two-layer GLA linked via softmax" and is trained as two chunkwise passes. These are the closest precedents for placing a nonlinearity between two linear memories that each keep a chunkwise kernel; the two memories are written independently, from the input alone, and GSA notes that the analogous two-pass DeltaNet was not explored. Delta MLP (Irie et al., 2021) writes every layer of a deep fast-weight network with the delta rule, from keys and values produced by the slow network, and the metalearned neural memory (Munkhdalai et al., 2019) updates each layer toward targets produced by a learned feedback network; in neither is a layer's target derived from the output error of the memory itself. TOASTER differs from all four in that its two writes are coupled: the output memory is keyed by the hidden activations, and the hidden memory is written toward a target derived from the output error, so that with the exact target the pair is one gradient step on the composite memory (Proposition 1).

Nonlinear (test-time-training) memories. Fast-weight memories written at test time go back to Ba et al. (2016), deep and recurrent fast-weight programmers to Irie et al. (2021), and deep memories with locally computed targets to the metalearned neural memory of Munkhdalai et al. (2019). TTT (Sun

et al., 2025) and Titans (Behrouz et al., 2025b) store an MLP updated by gradient descent at test time; Miras (Behrouz et al., 2026c), Atlas (Behrouz et al., 2026b), Nested Learning (Behrouz et al., 2025a), memory caching (Behrouz et al., 2026a), and end-to-end or in-place variants (Tandon et al., 2025; Feng et al., 2026) broaden the objectives, optimizers, and placement of such memories. Their training cost has been attacked from two directions: LaCT (Zhang et al., 2026) uses very large chunks so that each update is a large matmul, and TNT (Li et al., 2026) splits the memory into a global module updated on large chunks and local modules updated in parallel. Both keep the stale mini-batch update of Eq. (3). Concurrently, E^2-TTT (Zhong et al., 2026) derives closed-form chunk transitions that reproduce the exact chunk-end weights and momentum of an MLP memory, with every per-token gradient still evaluated at the chunk-start weights; TOASTER instead removes the obstacle to exact chunkwise computation within each layer, by making every layer a linear memory, and, with the kernel on both layers, freezes only the cross-layer signals at the chunk start (Proposition 2); its default advances the hidden layer by the summed chunk-start step of Eq. (3), like these methods, and the output layer exactly. Modular TTT (Tang et al., 2026) ablates the components of TTT and finds that deeper fast-weight networks tend to hurt at the scales tested, which it attributes to excessively large activations; TOASTER writes its hidden layer with a bounded, separately scaled correction rather than a raw gradient (Section 3.4), and the cascade improves on the one-layer memory at every size of our ladder. Complementary to ours, Wang et al. (2025) frame linear and test-time-training memories as test-time regression, and Liu et al. (2026) prove that TTT with key–value binding is linear attention with a learned, state-dependent feature map; TOASTER's read, $\mathbf{S}^{(2)\mathsf{T}}\sigma(\mathbf{S}^{(1)\mathsf{T}}\boldsymbol{q})$, is of this form, so the question is whether writing the hidden memory in context adds anything beyond a learned nonlinear key map, which the identity and $\gamma = 0$ rows of Table 12 test.

Target propagation. Target propagation (Bengio, 2014; Lee et al., 2015) trains deep networks by giving each layer a local target instead of a backpropagated gradient; later work analyzed it as approximate Gauss–Newton (Meulemans et al., 2020), applied it to recurrent networks (Roulet & Harchaoui, 2023), and learned or fixed the feedback pathways (Ernoult et al., 2022; Shibuya et al., 2023). Our purpose differs: we seek not a backprop-free learning rule but a *form* of the exact gradient step in which each layer's update is a standard linear-memory write, because that is what the WY kernel can execute. The observation behind Eq. (5), that a gradient step on a linear layer is a key–value write with the input as key and the error as value, is the dual form of gradient-trained layers of Irie et al. (2022); and the target $\boldsymbol{z}_t - \boldsymbol{g}_t$, a pre-activation moved by one gradient step and then fitted with a local squared loss, is the local-target construction of ProxProp (Frerix et al., 2018) and LocoProp (Amid et al., 2022), which TOASTER applies inside a recurrent memory, where the local fit is executed by a delta-rule kernel. Related in spirit, Cerruti et al. (2026) forward a lower layer's delta-rule write error or write value into the next layer of a network; TOASTER routes the analogous signal inside a single memory, between its two slots.

C DEEPER CASCADES

The construction of Section 3 extends to a cascade of N linear memories $\mathbf{S}_t^{(n)} \in \mathbb{R}^{d_{n-1}\times d_n}$, $n = 1,\dots,N$, with $d_0 = d_k$ and $d_N = d_v$. Writing $\boldsymbol{h}_t^{(0)} = \boldsymbol{k}_t$, the forward pass is

$$\boldsymbol{z}_t^{(n)} = \mathbf{S}_{t-1}^{(n)\mathsf{T}}\boldsymbol{h}_t^{(n-1)}, \qquad \boldsymbol{h}_t^{(n)} = \sigma(\boldsymbol{z}_t^{(n)}) \ \ (n < N), \qquad \hat{\boldsymbol{v}}_t = \boldsymbol{z}_t^{(N)}, \tag{7}$$

so that the output layer has no activation. Let $\mathcal{L}_t = \frac{1}{2}\|\hat{\boldsymbol{v}}_t - \boldsymbol{v}_t\|^2$ and define, for every layer, the target $\tilde{\boldsymbol{v}}_t^{(n)}$ through $\nabla_{\boldsymbol{z}_t^{(n)}}\mathcal{L}_t = \boldsymbol{z}_t^{(n)} - \tilde{\boldsymbol{v}}_t^{(n)}$. The chain rule gives the backward recursion

$$\tilde{\boldsymbol{v}}_t^{(N)} = \boldsymbol{v}_t, \qquad \tilde{\boldsymbol{v}}_t^{(n)} = \boldsymbol{z}_t^{(n)} - \Big(\mathbf{S}_{t-1}^{(n+1)}\big(\boldsymbol{z}_t^{(n+1)} - \tilde{\boldsymbol{v}}_t^{(n+1)}\big)\Big) \odot \sigma'(\boldsymbol{z}_t^{(n)}) \quad (n = N-1,\dots,1), \tag{8}$$

which for $N = 2$ is Eq. (5). Layer n is then written with key $\boldsymbol{h}_t^{(n-1)}$ and value $\tilde{\boldsymbol{v}}_t^{(n)}$, and, exactly as in Proposition 1, the N delta-rule steps coincide with one gradient step on $\mathcal{L}_t$ with respect to all N states. The chunkwise algorithm generalizes accordingly: Phase I runs the forward recursion and then Eq. (8) on the frozen chunk-initial states, producing all keys and targets of the chunk; Phases II–$(N{+}1)$ advance the layers in order $n = 1,\dots,N$, each hidden layer by either rule of Algorithm 1 and the output layer by LINMEM, each phase taking as queries the activated per-query outputs of the previous one. The cost is at most N sequentially dependent kernel calls and $O(N)$

dense products per chunk. In practice the bounded normalization $\mathcal{N}$ of Eq. (6) is applied to each back-projected error in Eq. (8), and the frozen-target approximation of Section 3.3 occurs at every hidden layer. We have not trained cascades with $N > 2$.

D TARGET DERIVATION AND PROOF OF PROPOSITION 1

Derivation of Eq. (5). With the forward pass Eq. (4) and $\boldsymbol{e}_t = \hat{\boldsymbol{v}}_t - \boldsymbol{v}_t$, the chain rule through $\hat{\boldsymbol{v}}_t = \mathbf{S}_{t-1}^{(2)\top}\boldsymbol{h}_t$ and $\boldsymbol{h}_t = \sigma(\boldsymbol{z}_t)$ gives

$$\nabla_{\boldsymbol{z}_t}\mathcal{L}_t = \Big(\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{z}_t}\Big)^{\top}\Big(\frac{\partial \hat{\boldsymbol{v}}_t}{\partial \boldsymbol{h}_t}\Big)^{\top}\nabla_{\hat{\boldsymbol{v}}_t}\mathcal{L}_t = \mathrm{Diag}\big(\sigma'(\boldsymbol{z}_t)\big)\,\mathbf{S}_{t-1}^{(2)}\,\boldsymbol{e}_t = \big(\mathbf{S}_{t-1}^{(2)}\,\boldsymbol{e}_t\big)\odot\sigma'(\boldsymbol{z}_t), \tag{9}$$

where we used the Jacobians $\partial \boldsymbol{h}_t/\partial \boldsymbol{z}_t = \mathrm{Diag}(\sigma'(\boldsymbol{z}_t))$ and $\partial \hat{\boldsymbol{v}}_t/\partial \boldsymbol{h}_t = \mathbf{S}_{t-1}^{(2)\top}$. The local loss $\mathcal{L}_t^{(1)} = \frac{1}{2}\|\boldsymbol{z}_t - \tilde{\boldsymbol{v}}_t\|^2$ has gradient $\boldsymbol{z}_t - \tilde{\boldsymbol{v}}_t$; equating the two gradients and solving for $\tilde{\boldsymbol{v}}_t$ yields Eq. (5).

Proof of Proposition 1. The gradients of $\mathcal{L}_t$ with respect to the two states, evaluated at $(\mathbf{S}_{t-1}^{(1)}, \mathbf{S}_{t-1}^{(2)})$, are

$$\nabla_{\mathbf{S}^{(2)}}\mathcal{L}_t = \boldsymbol{h}_t\,\boldsymbol{e}_t^{\top}, \qquad \nabla_{\mathbf{S}^{(1)}}\mathcal{L}_t = \boldsymbol{k}_t\,\big(\nabla_{\boldsymbol{z}_t}\mathcal{L}_t\big)^{\top} = \boldsymbol{k}_t\,(\boldsymbol{z}_t - \tilde{\boldsymbol{v}}_t)^{\top}, \tag{10}$$

because $\hat{\boldsymbol{v}}_t$ is linear in $\mathbf{S}^{(2)}$ with input $\boldsymbol{h}_t$, and $\boldsymbol{z}_t$ is linear in $\mathbf{S}^{(1)}$ with input $\boldsymbol{k}_t$. The delta rule Eq. (1) without forgetting, applied to $\mathbf{S}^{(2)}$ with key $\boldsymbol{h}_t$ and value $\boldsymbol{v}_t$, reads

$$\mathbf{S}_t^{(2)} = \mathbf{S}_{t-1}^{(2)} - \beta_t^{(2)}\boldsymbol{h}_t\big(\mathbf{S}_{t-1}^{(2)\top}\boldsymbol{h}_t - \boldsymbol{v}_t\big)^{\top} = \mathbf{S}_{t-1}^{(2)} - \beta_t^{(2)}\,\boldsymbol{h}_t\,\boldsymbol{e}_t^{\top} = \mathbf{S}_{t-1}^{(2)} - \beta_t^{(2)}\,\nabla_{\mathbf{S}^{(2)}}\mathcal{L}_t, \tag{11}$$

and applied to $\mathbf{S}^{(1)}$ with key $\boldsymbol{k}_t$ and value $\tilde{\boldsymbol{v}}_t$,

$$\mathbf{S}_t^{(1)} = \mathbf{S}_{t-1}^{(1)} - \beta_t^{(1)}\boldsymbol{k}_t\big(\mathbf{S}_{t-1}^{(1)\top}\boldsymbol{k}_t - \tilde{\boldsymbol{v}}_t\big)^{\top} = \mathbf{S}_{t-1}^{(1)} - \beta_t^{(1)}\,\boldsymbol{k}_t\,(\boldsymbol{z}_t - \tilde{\boldsymbol{v}}_t)^{\top} = \mathbf{S}_{t-1}^{(1)} - \beta_t^{(1)}\,\nabla_{\mathbf{S}^{(1)}}\mathcal{L}_t. \tag{12}$$

Both updates are gradient steps on $\mathcal{L}_t$ evaluated at the previous states, which is the claim. With a forget gate α_t the same computation shows that the output-layer step is a gradient step taken after weight decay; for the hidden layer, whose target is computed from the undecayed state, the gated delta rule adds $\beta_t^{(1)}(1-\alpha_t)\,\boldsymbol{k}_t\boldsymbol{z}_t^{\top}$ to that step, partly undoing the decay along $\boldsymbol{k}_t$ (Titans, by contrast, evaluates the gradient before its decay and adds momentum). □

What the chunkwise algorithm computes. In Algorithm 1, Phase I is three dense matrix multiplications and a few pointwise operations, fully parallel across the chunk. Phase II advances the hidden memory, with keys $\mathbf{K}_{[j]}$ and values $\widetilde{\mathbf{V}}_{[j]}$, and returns the per-query pre-activations $\mathbf{Z}_{[j]}^{q}$, either by a call of LINMEM (line 5) or, by default, by apply-then-update (line 6): every query of the chunk reads the chunk-initial state, and the C delta-rule writes, whose residuals $\mathbf{G}_{[j]}$ Phase I has already evaluated at that state, are summed into one update at the end of the chunk. Phase III calls LINMEM for the output memory, with keys $\mathbf{H}_{[j]}$, values $\mathbf{V}_{[j]}$, and queries $\sigma(\mathbf{Z}_{[j]}^{q})$, returning the chunk's output. With the homogeneous coordinates of Section 3.4, keys, queries, and hidden activations carry an appended constant 1, and the bias coordinate of the back-projection is dropped. Given the hidden keys $\mathbf{H}_{[j]}$ and targets $\widetilde{\mathbf{V}}_{[j]}$, a kernel call computes its layer's delta-rule recurrence exactly, for any chunk size: within the chunk, later writes erase and overwrite earlier ones, and the layer's residual is evaluated at its running state rather than at the chunk-initial one, which Eq. (3) cannot do. The approximation is that the cross-layer signals are evaluated once, at the chunk-initial states. We do not claim that this approximation is uniformly smaller than that of Eq. (3); what a kernel call removes is the additive, erasure-free update within its layer. For the output layer the approximation only stales the keys $\boldsymbol{h}_r$; for the hidden layer its effect depends on the update rule, as the next paragraph shows.

The chunkwise hidden update. In Algorithm 1 with the kernel on the hidden layer (line 5), the target of token r in chunk j is $\tilde{\boldsymbol{v}}_r = \boldsymbol{z}_r^{[j]} - \boldsymbol{g}_r^{[j]}$, where $\boldsymbol{z}_r^{[j]} = \mathbf{S}_{[j]}^{(1)\top}\boldsymbol{k}_r$ and $\boldsymbol{g}_r^{[j]}$ are the pre-activation and back-projected error at the chunk-initial states. Writing $\mathbf{S}^{(1),r}$ for the hidden state after token r of the chunk, the kernel applies the delta rule with the running state, so the executed update is

$$\mathbf{S}^{(1),r} - \mathbf{S}^{(1),r-1} = -\beta_r\boldsymbol{k}_r\big(\mathbf{S}^{(1),r-1\top}\boldsymbol{k}_r - \tilde{\boldsymbol{v}}_r\big)^{\top} = -\beta_r\boldsymbol{k}_r\big(\boldsymbol{z}_r - \boldsymbol{z}_r^{[j]} + \boldsymbol{g}_r^{[j]}\big)^{\top}, \tag{13}$$

with $\boldsymbol{z}_r = \mathbf{S}^{(1),r-1\mathsf{T}}\boldsymbol{k}_r$ the running pre-activation. The term $\boldsymbol{z}_r - \boldsymbol{z}_r^{[j]} = (\mathbf{S}^{(1),r-1} - \mathbf{S}^{(1)}_{[j]})^{\mathsf{T}}\boldsymbol{k}_r$ is the change in the hidden layer's prediction on $\boldsymbol{k}_r$ produced by the earlier tokens of the chunk; it is small when those tokens' keys are nearly orthogonal to $\boldsymbol{k}_r$ (with the homogeneous coordinate the inner products are $\boldsymbol{k}_i^{\mathsf{T}}\boldsymbol{k}_r + 1$, so it does not vanish exactly), vanishes at $C = 1$, and is first order in the within-chunk change of the state (so its effect on the loss is second order). With apply-then-update (line 6) the residual of every token is evaluated at $\mathbf{S}^{(1)}_{[j]}$, where the target was computed, so the executed per-token update is exactly $-\beta_r \boldsymbol{k}_r(\boldsymbol{g}_r^{[j]})^{\mathsf{T}}$ and the C updates are summed, as in Eq. (3); the term $\boldsymbol{z}_r - \boldsymbol{z}_r^{[j]}$ does not arise, and the queries of the chunk read $\mathbf{S}^{(1)}_{[j]}$.

The chunk update in closed form. Proposition 2 states what Algorithm 1 computes over a whole chunk with the kernel on both layers, when the forget gates are off and the exact target Eq. (5) is used, and compares it with chunked gradient descent at the chunk-initial state, i.e., Eq. (3) without decay. For a matrix $\mathbf{X} \in \mathbb{R}^{C\times d}$ with rows $\boldsymbol{x}_r$ and step sizes $\boldsymbol{\beta}$, write $\mathbf{T}_{\mathbf{X}} = \big(\mathbf{I} + \mathbf{M}^{-} \odot (\operatorname{diag}(\boldsymbol{\beta})\,\mathbf{X}\mathbf{X}^{\mathsf{T}})\big)^{-1}\operatorname{diag}(\boldsymbol{\beta})$ for the WY matrix of Eq. (18).

Proposition 2 (Chunk update)**.** *Fix a chunk with keys $\bar{\mathbf{K}}$ (rows $\bar{\boldsymbol{k}}_r$), values $\mathbf{V}$, chunk-initial states $\mathbf{S}^{(1)}_{[j]}, \mathbf{S}^{(2)}_{[j]}$, and step sizes $\boldsymbol{\beta}^{(1)}, \boldsymbol{\beta}^{(2)}$. Let $\mathbf{Z} = \bar{\mathbf{K}}\,\mathbf{S}^{(1)}_{[j]}$, let $\bar{\mathbf{H}}$ have rows $[\sigma(\boldsymbol{z}_r); 1]$, let $\mathbf{E} = \bar{\mathbf{H}}\,\mathbf{S}^{(2)}_{[j]} - \mathbf{V}$, and let $\mathbf{G}$ have rows $\boldsymbol{g}_r = (\mathbf{S}^{(2)}_{[j]}\boldsymbol{e}_r) \odot \sigma'(\boldsymbol{z}_r)$ with the bias row dropped: the per-token errors of the two layers at the chunk-initial states. Without forgetting and with the exact targets $\tilde{\boldsymbol{v}}_r = \boldsymbol{z}_r - \boldsymbol{g}_r$, Algorithm 1 with the kernel on both layers (line 5) returns*

$$\mathbf{S}^{(1)}_{[j+1]} = \mathbf{S}^{(1)}_{[j]} - \bar{\mathbf{K}}^{\mathsf{T}}\,\mathbf{T}_{\bar{\mathbf{K}}}\,\mathbf{G}, \qquad \mathbf{S}^{(2)}_{[j+1]} = \mathbf{S}^{(2)}_{[j]} - \bar{\mathbf{H}}^{\mathsf{T}}\,\mathbf{T}_{\bar{\mathbf{H}}}\,\mathbf{E}, \tag{14}$$

whereas one step of chunked gradient descent on $\sum_r \mathcal{L}_r$ at the chunk-initial states, with per-token step sizes, gives the same expressions with $\mathbf{T}_{\bar{\mathbf{K}}}$ and $\mathbf{T}_{\bar{\mathbf{H}}}$ replaced by $\operatorname{diag}(\boldsymbol{\beta}^{(1)})$ and $\operatorname{diag}(\boldsymbol{\beta}^{(2)})$.

Proof of Proposition 2. By Eq. (18)–Eq. (19), the delta rule with keys $\mathbf{X}$ and values $\mathbf{U}$ advances a state $\mathbf{S}$ over the chunk to $\mathbf{S} + \mathbf{X}^{\mathsf{T}}\mathbf{T}_{\mathbf{X}}(\mathbf{U} - \mathbf{X}\mathbf{S})$. Phase II applies it to $\mathbf{S}^{(1)}_{[j]}$ with keys $\bar{\mathbf{K}}$ and values $\widetilde{\mathbf{V}} = \mathbf{Z} - \mathbf{G} = \bar{\mathbf{K}}\,\mathbf{S}^{(1)}_{[j]} - \mathbf{G}$, so $\mathbf{U} - \mathbf{X}\mathbf{S} = -\mathbf{G}$; Phase III applies it to $\mathbf{S}^{(2)}_{[j]}$ with keys $\bar{\mathbf{H}}$ and values $\mathbf{V}$, so $\mathbf{U} - \mathbf{X}\mathbf{S} = -\mathbf{E}$. For chunked gradient descent, the gradients of $\mathcal{L}_r$ at the chunk-initial states are $\bar{\boldsymbol{k}}_r\boldsymbol{g}_r^{\mathsf{T}}$ and $\bar{\boldsymbol{h}}_r\boldsymbol{e}_r^{\mathsf{T}}$ (proof of Proposition 1), and summing the C steps gives $-\bar{\mathbf{K}}^{\mathsf{T}}\operatorname{diag}(\boldsymbol{\beta}^{(1)})\mathbf{G}$ and $-\bar{\mathbf{H}}^{\mathsf{T}}\operatorname{diag}(\boldsymbol{\beta}^{(2)})\mathbf{E}$. □

Both updates are therefore driven by the same per-token errors and differ only in the $C \times C$ matrix that mixes them. Chunked gradient descent adds the C steps, each computed at the chunk-initial state, as in a Jacobi iteration; Algorithm 1 applies them in order, each residual re-evaluated against the running state, as in a Gauss–Seidel (Kaczmarz) sweep toward the fixed targets $\widetilde{\mathbf{V}}$ and $\mathbf{V}$. Since $\mathbf{L} = \mathbf{M}^{-} \odot (\operatorname{diag}(\boldsymbol{\beta})\mathbf{X}\mathbf{X}^{\mathsf{T}})$ is strictly lower triangular, $\mathbf{T}_{\mathbf{X}} - \operatorname{diag}(\boldsymbol{\beta}) = \sum_{m=1}^{C-1}(-\mathbf{L})^m \operatorname{diag}(\boldsymbol{\beta})$ exactly, so the two updates coincide only when $\beta_r \bar{\boldsymbol{x}}_r^{\mathsf{T}}\bar{\boldsymbol{x}}_i = 0$ for all $i < r$ in the chunk; with the homogeneous coordinate, $\bar{\boldsymbol{k}}_i^{\mathsf{T}}\bar{\boldsymbol{k}}_r = \boldsymbol{k}_i^{\mathsf{T}}\boldsymbol{k}_r + 1$, this never holds, and the bias coordinate is the main source of within-chunk coupling. Neither update is closer to per-token gradient descent in general: within a chunk, Algorithm 1 fits fixed targets, whereas chunked gradient descent adds fixed gradients, and the errors $\mathbf{G}$ are equally stale in both. With apply-then-update on the hidden layer (line 6, our default) the two forms are mixed: the hidden update is the chunked-gradient expression $\mathbf{S}^{(1)}_{[j]} - \bar{\mathbf{K}}^{\mathsf{T}}\operatorname{diag}(\boldsymbol{\beta}^{(1)})\mathbf{G}$, a Jacobi step toward the fixed targets, and the output update keeps $\mathbf{T}_{\bar{\mathbf{H}}}$, the Gauss–Seidel sweep.

The practical target is a descent direction. With the bounded target of Eq. (6), the hidden update becomes $-\beta_t^{(1)}\boldsymbol{k}_t\hat{\boldsymbol{g}}_t^{\mathsf{T}}$ with $\hat{\boldsymbol{g}}_t = \mathcal{N}(\mathbf{S}^{(2)}_{t-1}\boldsymbol{e}_t) \odot \sigma'(\boldsymbol{z}_t)$, whereas the gradient direction is $\boldsymbol{g}_t = (\mathbf{S}^{(2)}_{t-1}\boldsymbol{e}_t) \odot \sigma'(\boldsymbol{z}_t)$. Since $\mathcal{N}(x) = \gamma\,\operatorname{softsign}(x)$ has the sign of x for every coordinate, $\langle \boldsymbol{g}_t, \hat{\boldsymbol{g}}_t\rangle = \sum_i (\mathbf{S}^{(2)}_{t-1}\boldsymbol{e}_t)_i\,\mathcal{N}\big((\mathbf{S}^{(2)}_{t-1}\boldsymbol{e}_t)_i\big)\,\sigma'(\boldsymbol{z}_t)_i^2 \geq 0$, and therefore $\langle \nabla_{\mathbf{S}^{(1)}}\mathcal{L}_t,\ \boldsymbol{k}_t\hat{\boldsymbol{g}}_t^{\mathsf{T}}\rangle = \|\boldsymbol{k}_t\|^2\langle \boldsymbol{g}_t, \hat{\boldsymbol{g}}_t\rangle \geq 0$: the update is a descent direction for $\mathcal{L}_t$, although no longer the gradient.

E THE WY REPRESENTATION AND THE CHUNKWISE KERNEL

This appendix derives the chunkwise algorithm behind LINMEM for the ungated delta rule, following Yang et al. (2024b), in the state convention $\mathbf{S}_t \in \mathbb{R}^{d_k \times d_v}$, $\boldsymbol{o}_t = \mathbf{S}_t^\top \boldsymbol{q}_t$ used throughout the paper. The gated variants used in our experiments follow the same pattern and are described at the end.

Unrolling a chunk. Consider chunk j with tokens $r = 1, \dots, C$, chunk-initial state $\mathbf{S}^0 = \mathbf{S}_{[j]}$, and $\mathbf{S}^r$ the state after token r. Unrolling $\mathbf{S}^r = (\mathbf{I} - \beta_r \boldsymbol{k}_r \boldsymbol{k}_r^\top)\mathbf{S}^{r-1} + \beta_r \boldsymbol{k}_r \boldsymbol{v}_r^\top$ gives

$$\mathbf{S}^r = \mathbf{P}^r \mathbf{S}^0 + \mathbf{H}^r, \qquad \mathbf{P}^r = \prod_{s=r}^{1} (\mathbf{I} - \beta_s \boldsymbol{k}_s \boldsymbol{k}_s^\top), \qquad \mathbf{H}^r = \sum_{i=1}^{r} \Big[\prod_{s=r}^{i+1} (\mathbf{I} - \beta_s \boldsymbol{k}_s \boldsymbol{k}_s^\top) \Big] \beta_i \boldsymbol{k}_i \boldsymbol{v}_i^\top, \tag{15}$$

where products are ordered with the largest index on the left.

WY form. Products of generalized Householder matrices admit the compact representation (Bischof & Van Loan, 1987)

$$\mathbf{P}^r = \mathbf{I} - \sum_{i=1}^{r} \boldsymbol{k}_i \boldsymbol{w}_i^\top, \qquad \mathbf{H}^r = \sum_{i=1}^{r} \boldsymbol{k}_i \boldsymbol{u}_i^\top, \tag{16}$$

with auxiliary vectors defined by the recursions

$$\boldsymbol{w}_r = \beta_r \Big(\boldsymbol{k}_r - \sum_{i<r} (\boldsymbol{k}_r^\top \boldsymbol{k}_i)\, \boldsymbol{w}_i \Big), \qquad \boldsymbol{u}_r = \beta_r \Big(\boldsymbol{v}_r - \sum_{i<r} (\boldsymbol{k}_r^\top \boldsymbol{k}_i)\, \boldsymbol{u}_i \Big). \tag{17}$$

Both follow by induction from $\mathbf{P}^r = (\mathbf{I} - \beta_r \boldsymbol{k}_r \boldsymbol{k}_r^\top)\mathbf{P}^{r-1}$ and $\mathbf{H}^r = (\mathbf{I} - \beta_r \boldsymbol{k}_r \boldsymbol{k}_r^\top)\mathbf{H}^{r-1} + \beta_r \boldsymbol{k}_r \boldsymbol{v}_r^\top$. Stacking the auxiliary vectors as rows of $\mathbf{W} \in \mathbb{R}^{C \times d_k}$ and $\mathbf{U} \in \mathbb{R}^{C \times d_v}$, the recursions Eq. (17) become a single triangular system (the UT transform (Joffrain et al., 2006)),

$$\mathbf{W} = \mathbf{T}\,\mathbf{K}, \qquad \mathbf{U} = \mathbf{T}\,\mathbf{V}, \qquad \mathbf{T} = \Big(\mathbf{I} + \mathbf{M}^- \odot \big(\mathrm{diag}(\boldsymbol{\beta})\, \mathbf{K}\mathbf{K}^\top \big) \Big)^{-1} \mathrm{diag}(\boldsymbol{\beta}), \tag{18}$$

where $\mathbf{M}^-$ is the strictly lower-triangular mask and the inverse is a $C \times C$ forward substitution.

Chunk state and outputs. Substituting Eq. (16) into Eq. (15) at $r = C$ and reading out $\boldsymbol{o}_r = (\mathbf{S}^r)^\top \boldsymbol{q}_r$ for every r gives

$$\mathbf{S}_{[j+1]} = \mathbf{S}_{[j]} + \mathbf{K}^\top \big(\mathbf{U} - \mathbf{W}\,\mathbf{S}_{[j]} \big), \qquad \mathbf{O} = \mathbf{Q}\,\mathbf{S}_{[j]} + \big(\mathbf{Q}\mathbf{K}^\top \odot \mathbf{M} \big) \big(\mathbf{U} - \mathbf{W}\,\mathbf{S}_{[j]} \big), \tag{19}$$

with $\mathbf{M}$ the lower-triangular mask including the diagonal. Every operation in Eq. (18)–Eq. (19) is a dense matrix product (or a small triangular solve), the state is propagated sequentially only across chunks, and the result is exact for any chunk size C. This is the kernel LINMEM of Eq. (2); TOASTER calls it once or twice per chunk with the arguments listed in Algorithm 1.

Gated variants. A scalar forget gate α_t (Gated DeltaNet) commutes with the Householder factors, so the cumulative decay $\gamma^r = \prod_{s \le r} \alpha_s$ can be pulled out of $\mathbf{P}^r$ and absorbed into decay-weighted keys and a decay-aware causal mask in Eq. (18)–Eq. (19) (Yang et al., 2025). A channel-wise gate $\mathrm{Diag}(\boldsymbol{\alpha}_t)$ (KDA) does not commute, and the cumulative diagonal decays instead enter the inner products $\boldsymbol{k}_r^\top \boldsymbol{k}_i$ of Eq. (17) and the corresponding entries of $\mathbf{T}$ (Kimi Team et al., 2025). In both cases the algorithm keeps the structure of Eq. (19), and TOASTER uses the KDA form of the kernel unchanged for every layer it advances with the kernel.

F OPERATION COUNT

We count multiply–accumulate operations per token and per head in the forward pass, ignoring the $O(C^3)$ triangular solve and pointwise work. For one call of LINMEM with input width d_{in}, output width d_{out}, and chunk size C, Eq. (18)–Eq. (19) cost approximately

$$3\, d_{\text{in}} d_{\text{out}} \;+\; 3\, C\, d_{\text{in}} \;+\; 2\, C\, d_{\text{out}} \tag{20}$$

per token (three products involving the state and five intra-chunk products of size C). KDA at $d_k = d_v = 64$, $C = 32$ therefore costs about 2.3×10^4 operations per token and head. TOASTER

makes two calls, $d_k \to d_h$ and $d_h \to d_v$, and adds the three Phase I products $\mathbf{K}_{[j]}\mathbf{S}_{[j]}^{(1)}$, $\mathbf{H}_{[j]}\mathbf{S}_{[j]}^{(2)}$, and $\mathbf{E}_{[j]}\mathbf{S}_{[j]}^{(2)\mathsf{T}}$, each $d_k d_h$ (or $d_h d_v$) per token; with $d_h = 256$ the total is about 2.0×10^5 operations per token and head, roughly nine times KDA. With the default apply-then-update hidden layer (line 6 of Algorithm 1) the first call is replaced by the two products $\mathbf{Q}_{[j]}\mathbf{S}_{[j]}^{(1)}$ and $\mathbf{K}_{[j]}{}^{\mathsf{T}}\,\mathrm{Diag}(\boldsymbol{\beta}^{(1)})\mathbf{G}_{[j]}$, $2d_k d_h$ per token, and the total falls to about 1.6×10^5, roughly seven times KDA. For comparison, the dense projections of a block at width $D = 2048$ (query, key, value, and output projections plus a SwiGLU feed-forward network of inner width $4D$) cost about $16D^2 \approx 6.7 \times 10^7$ operations per token. With 32 heads, the memory kernels thus account for roughly 1% of a block's operations for KDA and roughly 9% for TOASTER with two kernel calls (8% with the default) at the 1.2B scale, and proportionally less at larger widths, since the kernel cost grows with the number of heads while the dense cost grows with D^2. At 145M ($D = 1024$, 16 heads) the shares are about 2% for KDA and 15–19% for TOASTER. These are forward-pass operation counts (the backward pass scales all terms alike), excluding the $O(C^3)$ triangular solves and the gate projections; they are not measured throughput. Chunkwise kernels at $C = 32$ are often bound by state traffic rather than arithmetic, and TOASTER reads and writes about eight times KDA's state per chunk, with Phase III dependent on Phase II; we have not measured utilization, and the larger state traffic may lower it. The chunk loop also remains sequential: a chunk's hidden keys and targets depend on the chunk-initial states, so context parallelism across chunks does not apply.

State size. Per head, KDA stores $d_k d_v$ numbers and TOASTER stores $(d_k+1)d_h + (d_h+1)d_v$; with $d_k = d_v = 64$ and $d_h = 256$ this is 33,088 numbers, about eight times KDA. At the 1.2B scale (24 layers, 32 heads) the full TOASTER state is 2.5×10^7 numbers per sequence, constant in the sequence length; a softmax-attention key–value cache at the same scale holds $2 \cdot 24 \cdot (32 \cdot 64) = 9.8 \times 10^4$ numbers *per token* in the same precision, i.e., it exceeds the TOASTER state beyond a context of about 260 tokens.

G EXPERIMENTAL DETAILS

G.1 SCALING LADDER

Table 5: Scaling ladder. Depth, width, and number of heads grow together; the head dimension is 64, the feed-forward inner width is $4D$, and the context length is 4,096 tokens throughout. Parameter counts are those reported for the ladder configuration; the recurrent mixers add a small number of parameters for their gates and convolutions. The token budget is roughly proportional to model size (46 to 40 tokens per non-embedding parameter).

Size	Layers	Width D	Heads	Non-emb.	Total	Batch	Steps	Tokens	Peak LR	Weight decay
145M	12	1024	16	201M	234M	64	35,050	9.2B	1.95×10^{-3}	2.28×10^{-4}
300M	15	1280	20	390M	431M	96	42,708	16.8B	1.56×10^{-3}	1.87×10^{-4}
550M	18	1536	24	671M	721M	160	42,630	27.9B	1.30×10^{-3}	1.88×10^{-4}
850M	21	1792	28	1.06B	1.12B	192	55,095	43.3B	1.12×10^{-3}	1.45×10^{-4}
1.2B	24	2048	32	1.58B	1.65B	256	60,725	63.7B	9.77×10^{-4}	1.32×10^{-4}

Sizes and token budgets. Every architecture is trained at the five sizes of Table 5, from 12 layers of width 1024 to 24 layers of width 2048, scaling depth, width, and number of heads together. We refer to the sizes by the nominal labels 145M, 300M, 550M, 850M, and 1.2B; their non-embedding parameter counts are 0.20B, 0.39B, 0.67B, 1.06B, and 1.58B. The token budget, batch size, and peak learning rate are scaled with model size along a single scaling rule (9.2B to 63.7B tokens, i.e., 46 to 40 tokens per non-embedding parameter; Table 5), so that every point of the ladder is trained under the same regime. At a given size all architectures share every backbone, data, and optimizer setting; only the sequence mixer changes.

Data and tokenization. All models are pre-trained on C4 (Raffel et al., 2020) (version 3.1.0). Documents are concatenated and packed into sequences of 4,096 tokens, with beginning- and end-of-sequence tokens marking document boundaries. We use a unigram SentencePiece tokenizer (Kudo

& Richardson, 2018) with a vocabulary of 32,000 subwords plus 100 reserved tokens, shared by all architectures together with the embedding matrix. Validation loss is measured on the C4 validation split (in-domain) and, out of domain, on PG19 (Rae et al., 2020), full-length books that stress long-range coherence, each at the 4,096-token context. Perplexity is exp of the mean per-token loss.

Optimization. All models use AdamW (Loshchilov & Hutter, 2019) with moment coefficients $(0.9, 0.95)$ and decoupled weight decay applied to weight matrices only (not to biases or normalization scales). The learning rate warms up linearly from zero over the first 1,000 steps to the peak in Table 5 and decays linearly to zero at the last step; the global gradient norm is clipped to 1.0. Master weights, gradients, and the recurrent memory states of all recurrent models are kept in 32-bit floating point. All numbers in the paper are single training runs; we lacked the compute to repeat them and therefore rest the evidence on the consistency of the ordering across sizes and corpora.

G.2 Training compute, scaling fits, and compute-efficiency gain

Training compute. We measure the training compute of every ladder point as $C = 6ND$ FLOPs, where N is the number of non-embedding parameters of the ladder configuration (Table 5) and D the number of training tokens: the forward pass costs about $2N$ FLOPs per token and the backward pass about $4N$ (Kaplan et al., 2020; Hoffmann et al., 2022). The five points of the ladder correspond to 1.1×10^{19}, 3.9×10^{19}, 1.1×10^{20}, 2.8×10^{20}, and 6.0×10^{20} FLOPs. The approximation ignores the sequence-length-dependent cost of attention (quadratic in the 4,096-token context) and the memory kernels of the recurrent models (about 1% of a block's multiply–accumulates for KDA and 8–9% for Toaster at 1.2B; Appendix F), and it uses the same N for all three architectures, whose mixers differ only by a small number of gate and convolution parameters. The three models at a given size therefore share one compute budget, and the comparisons below are comparisons of loss at matched $6ND$ compute, not of wall-clock time (Figure 4).

Scaling fits. For each architecture and corpus we fit the power law $L(C) = (C_c/C)^{\alpha}$, i.e., $\log L = a - \alpha \log C$, by least squares in log–log space to the five points of the ladder, the form used by Kaplan et al. (2020) for the compute frontier. Table 6 lists the fitted exponents. Every fit has $R^2 \geq 0.9996$ and a maximum residual of 0.006 nats, and the exponents (0.045–0.047) are close to the frontier exponent of 0.050 reported by Kaplan et al. (2020). A fit with an irreducible term, $L(C) = E + AC^{-\alpha}$ (Hoffmann et al., 2022), is not identifiable from five points: on PG19 the fitted E is zero and on C4 it trades off against α ($E = 0.9$, $\alpha = 0.07$). We therefore report the two-parameter fit; the compute-efficiency gains below change by less than one percentage point under the three-parameter form, because they depend only on the local slope of Attention's curve.

Compute-efficiency gain. We convert a loss difference at matched compute into a compute ratio through Attention's fitted law. If Attention's loss falls as $C^{-\alpha_{\mathrm{A}}}$, lowering it from L_{A} to a model's loss L_{m} at the same compute requires multiplying Attention's compute by

$$g \;=\; \left(L_{\mathrm{A}}/L_{\mathrm{m}}\right)^{1/\alpha_{\mathrm{A}}}, \tag{21}$$

which we call the compute-efficiency gain of the model relative to Attention: $g > 1$ means the model reaches Attention's loss with $1/g$ of the compute, $g < 1$ that it needs more. The gain panels of Figure 2 and Figure 6 show g at the five measured sizes, joined by straight segments; no curve is fitted to the gains, and the fitted loss laws elsewhere are drawn over the measured compute range only, without extrapolation. Because $\alpha_{\mathrm{A}} \approx 0.046$, a loss difference of 0.01 nats (about 0.4%) corresponds to about 9% of compute, which is why the gain separates the architectures more visibly than the loss or perplexity curves.

Why $6ND$, and what the sequence mixers cost. We report compute as $6ND$ rather than as a count that includes each sequence mixer's own operations, for three reasons. First, it is the convention of compute scaling laws: Kaplan et al. (2020) likewise leave the context-dependent cost of attention out of C, since it is a small fraction of the total when $D \gg n_{\mathrm{ctx}}/12$ (on our ladder D is three to six times $n_{\mathrm{ctx}}/12$). Second, $6ND$ counts what the three architectures share at each size—the backbone, N, and the training tokens—so that the gain of Eq. (21) measures what the mixer contributes per parameter and per training token. Third, and most important for scaling, at a fixed context the mixer's own cost becomes a small part of the total as the model grows. The dense projections of a block

Table 6: Power-law fits $L = (C_c/C)^{\alpha}$ of validation loss against training compute (log–log least squares over the five sizes) and the compute-efficiency gain relative to Attention of Eq. (21) at the five measured sizes ($g > 1$: less compute than Attention for the same loss).

				Compute-efficiency gain vs. Attention				
Corpus	Model	α	R^2	145M	300M	550M	850M	1.2B
C4 (in-domain)	Attention	0.047	0.9997	1	1	1	1	1
	KDA	0.047	0.9996	0.96	0.97	0.98	0.96	0.96
	TOASTER	0.045	0.9998	1.12	1.07	1.07	1.02	1.01
PG19	Attention	0.047	0.9997	1	1	1	1	1
	KDA	0.046	0.9998	0.93	0.91	0.93	0.92	0.89
	TOASTER	0.045	0.9999	1.06	1.02	1.01	0.97	0.94

cost $16D^2$ operations per token. A recurrent mixer costs a fixed number of operations per head and token, set by its head dimensions and chunk size and independent of D (Appendix F), and causal attention costs about TD at context T. Relative to the dense projections, the mixers therefore shrink as $1/D$: about $22/D$ for KDA, $156/D$ for TOASTER ($194/D$ with the kernel on both layers), and $T/16D = 256/D$ for attention at $T = 4{,}096$. The top block of Table 7 lists these shares along the ladder; at width $4{,}096$ they would be about 0.5%, 4% (5%), and 6%.

Gains with the mixers counted. Counting the mixers multiplies each model's compute by $1 + o$, where o is its share in Table 7 (the backward pass scales all terms alike; gate projections and triangular solves are left out, as in Appendix F), and turns the gain of Eq. (21) into $g\,(1+o_{\mathrm{A}})/(1+o_{\mathrm{m}})$; refitting Attention's law on the mixer-inclusive compute changes these values by less than half a percentage point. Against Attention, whose quadratic term is the largest mixer cost at 4K, every gain rises: TOASTER's C4 gain becomes 1.22 at 145M and 1.06 at 1.2B, and KDA's 1.18 and 1.07. Against KDA, the extra kernel arithmetic of TOASTER offsets its gain at these sizes: the ratio of TOASTER's gain to KDA's is 1.06–1.16 on C4 and 1.05–1.14 on PG19 at matched $6ND$, and 0.99–1.03 and 0.98–1.02 at matched mixer-inclusive compute. On our ladder TOASTER's advantage over KDA is therefore one per parameter and per training token. Its extra cost and this advantage both shrink with scale (on C4, from 13% to 6% and from 16% to 6% of compute), and whether the advantage outlasts the cost at larger widths is untested. These are operation counts, not measured time; Figure 4 reports step time.

Table 7: Sequence-mixer cost and the compute-efficiency gain with it counted. Top: forward operations of the sequence mixer per token as a share of a block's dense projections ($16D^2$; Appendix F), for Attention at the 4,096-token context. Middle: C4 gain relative to Attention, Eq. (21), with each model's compute multiplied by one plus its share. Bottom: ratio of TOASTER's gain to KDA's at matched $6ND$ and at matched mixer-inclusive compute.

		145M	300M	550M	850M	1.2B
Mixer share	Attention	25.0%	20.0%	16.7%	14.3%	12.5%
	KDA	2.1%	1.7%	1.4%	1.2%	1.1%
	TOASTER	15.2%	12.2%	10.2%	8.7%	7.6%
	TOASTER, kernel on both layers	18.9%	15.2%	12.6%	10.8%	9.5%
C4 gain, mixer-inclusive	KDA	1.18	1.14	1.13	1.08	1.07
	TOASTER	1.22	1.15	1.13	1.07	1.06
TOASTER vs. KDA, C4	matched $6ND$	1.16	1.11	1.08	1.06	1.06
	mixer-inclusive	1.03	1.01	1.00	0.99	0.99
TOASTER vs. KDA, PG19	matched $6ND$	1.14	1.12	1.09	1.05	1.05
	mixer-inclusive	1.01	1.02	1.01	0.98	0.99

G.3 SHARED BACKBONE

All models are built on NanoDO (Liu et al., 2024), a minimal decoder-only Transformer implementation in JAX, and differ only in the sequence mixer: the attention block of every layer is replaced by

the mixer under comparison, and everything else—embeddings, normalization, feed-forward blocks, output head, data pipeline, and optimizer—is shared unchanged. Every model is a pre-normalized decoder: RMSNorm (Zhang & Sennrich, 2019) ($\epsilon = 10^{-6}$) precedes the sequence mixer and the feed-forward network of each block, and a final RMSNorm precedes the output projection. The feed-forward network is a SwiGLU (Shazeer, 2020) with inner width $4D$. In the attention baseline, queries and keys carry rotary position embeddings (Su et al., 2024) with base $10{,}000$ over the head dimension of 64; the recurrent mixers use no positional encoding (Section 3.4). Linear layers are initialized with Xavier-uniform weights and embeddings from a zero-mean normal distribution scaled by the fan-in. The recurrent mixers apply a depthwise causal convolution of width 4 to the projected queries, keys, and values before the memory, as in KDA.

G.4 Sequence mixers

Attention. Causal multi-head softmax attention with 64-dimensional heads and RMSNorm applied to queries and keys before the dot product.

KDA. The chunkwise gated delta rule of Kimi Team et al. (2025) with chunk size $C = 32$, ℓ_2-normalized keys and queries, input-dependent channel-wise forget gates and step sizes, a step-size scale of 1.0, and a decay scale of 0.1.

Titans. Used in the throughput comparison (Figure 4): a deep test-time-training memory (Behrouz et al., 2025b) whose two-layer MLP has hidden width $d_h = d_k = 64$, updated by gradient descent on the reconstruction loss with the stale chunkwise gradient of Eq. (3), momentum, and weight decay, and timed at chunk sizes $C = 16$ and $C = 32$ (Appendix G.8). Our Titans implementation builds on the public JAX code base of TTT (Sun et al., 2025).

Toaster. The configuration used in every main-text experiment except the throughput profile, which uses a smaller, state-matched hidden width (Appendix G.8): two KDA memories in cascade with hidden width $d_h = 4d_k = 256$, GELU activation, homogeneous-coordinate bias (Section 3.4), the hidden memory advanced by apply-then-update (line 6 of Algorithm 1, with the chunk-averaged forget gate) and the output memory by the KDA kernel, RMSNorm on the value stream, the bounded back-projected error of Eq. (6) with $\gamma = 0.01$, input-dependent channel-wise forget gates and step sizes produced by rank-64 projections of the input, a step-size scale of 0.13, a decay scale of 0.01, and chunk size $C = 32$. Unlike KDA, Toaster does not ℓ_2-normalize the keys $\bar{\boldsymbol{h}}_t$ or the queries of the output memory; nothing therefore bounds $\beta_t^{(2)} \|\bar{\boldsymbol{h}}_t\|^2$ by construction, but we have not found this to be an issue in practice. The memory states are accumulated in 32-bit floating point, as for KDA. The state per head is $(d_k + 1)d_h + (d_h + 1)d_v = 33{,}088$ numbers, about eight times KDA's $d_k d_v = 4{,}096$ (Appendix F).

G.5 Downstream evaluation

All downstream tasks are evaluated three-shot on their standard evaluation splits (the validation split where labeled, e.g., the SuperGLUE tasks): three in-context demonstrations of the task precede each query, and there is no task-specific fine-tuning or prompt engineering. For multiple-choice and classification tasks, each candidate completion is scored by its length-normalized conditional log-likelihood given the prompt and the highest-scoring candidate is selected. Evaluation runs use a batch size of 256 and the full 4,096-token context, data-parallel over four accelerators. The 13 tasks, grouped into the six skill categories of Table 10, are ARC-Easy and ARC-Challenge (Clark et al., 2018) (science QA); COPA (Roemmele et al., 2011), HellaSwag (Zellers et al., 2019), OpenBookQA (Mihaylov et al., 2018), PIQA (Bisk et al., 2020), and SWAG (Zellers et al., 2018) (commonsense reasoning); MultiRC (Khashabi et al., 2018) (reading comprehension); MathQA (Amini et al., 2019) (mathematical reasoning); CB (de Marneffe et al., 2019), MNLI (Williams et al., 2018), and RTE (Wang et al., 2019b; Dagan et al., 2005) (NLI and entailment); and SST-2 (Socher et al., 2013) (linguistic understanding), several of them through SuperGLUE (Wang et al., 2019a). Six of the tasks (SST-2, MultiRC, MNLI, MathQA, CB, RTE) are at or near the chance or majority-class baseline for every model at this scale; on the remaining seven (ARC-Easy, ARC-Challenge, COPA, HellaSwag, OpenBookQA, PIQA, SWAG) the averages are 56.5, 57.1, and 57.7 for Attention, KDA, and Toaster. The largest per-task margins (2–7 points on COPA, CB, and RTE) fall on the smallest evaluation

sets (100, 56, and 277 items, with binomial standard errors of 3–7 points), and no difference on an evaluation set of more than a thousand items exceeds two points.

G.6 Length extrapolation

Figure 3 evaluates the 1.2B models, trained with a 4,096-token context, on sequences five times longer than anything seen in training. We take the novels of the English long-book question-answering task (En.QA) of ∞Bench (Zhang et al., 2024a), in which the names of key entities are replaced throughout each book to prevent contamination, and draw one sequence of 20,480 tokens from each of 100 books. Each model scores every sequence with its final checkpoint (60,725 steps), and we record the per-token loss at every position, averaged over the 100 sequences; the perplexity at a position is $\exp$ of that mean loss. The curves in Figure 3 are a one-sided running average of the per-position mean loss over the preceding positions, which is why the first thousand positions reflect the warm-up of the average; the shaded region marks positions beyond the training context, and the perplexity axis is capped at 30, which Attention exceeds near position 10K. Attention is evaluated with the rotary base it was trained with and no position-extrapolation technique; the recurrent models need none, since their state does not grow with the context. Table 11 reports the mean loss of each model in 4K-token windows; beyond the first thousand positions, the running-average loss of Toaster is below KDA's at every position, and at the last position (20K) it is lower by 0.008 nats.

G.7 Needle in a haystack

We use the three single-needle tasks of RULER (Hsieh et al., 2024), in which a key–value pair (the needle) is inserted into a long context (the haystack) and the model is asked, after the context, to return the value associated with the key. S-NIAH-1 uses a passkey (a number) as the needle and repeated synthetic sentences as the haystack; S-NIAH-2 uses a number as the needle and essays as the haystack; S-NIAH-3 uses a UUID as the needle and the same essay haystack. We keep RULER's task definitions and prompt template but regenerate the haystacks for our setting: each prompt is built to fill exactly the target context length of 512, 1K, 2K, or 4K tokens under our tokenizer (Appendix G.1), and every generated prompt is verified to fit its length budget with the query intact. The needle is inserted at 15 depths spaced evenly from the start (0%, farthest from the query) to the end (100%) of the haystack. Accuracy is the fraction of examples per cell for which the value is retrieved exactly, with greedy decoding and no instruction tuning, using the final 1.2B checkpoints; Table 3 averages it over the 15 depths, and Figures 8–10 report every cell.

G.8 Throughput measurement

Figure 4 reports the wall-clock time of one training step (forward and backward pass with the optimizer update) for context lengths from 2K to 32K tokens, with the batch size reduced as the context grows so that every step processes 512K tokens. Each configuration is timed over three runs with different seeds and the mean is reported; the standard deviations are small compared with the differences between architectures. The measurements use the 145M configuration on eight TPU v6 chips (a $2 \times 2 \times 2$ slice). Attention uses Flash Attention, and KDA and Toaster use chunk size $C = 32$; Titans is timed at $C = 16$ and $C = 32$, since its stale chunkwise update becomes less accurate with C. The Toaster timed here is not the default configuration of our quality experiments. For a fair comparison of training throughput we match the state of the two deep memories: Toaster is timed at hidden width $d_h = d_k = 64$, the hidden width of the Titans memory MLP, instead of the default $d_h = 4d_k = 256$ (Appendix G.4), so that Toaster and Titans store the same state per head, and all three recurrent models use 64-dimensional heads. Only the update rules are those of the default: apply-then-update on the hidden memory and the KDA kernel on the output memory. We have not optimized the Toaster implementation for speed: it is written naively in JAX, calling the KDA chunkwise routine unchanged for the output memory and computing the hidden-memory update and the cross-layer signals with plain dense operations, with no fused or hardware-specific kernel. Improving its speed, for example with a fused kernel for the cascade, is left to future work. At 32K, one step takes 1,085 ms for KDA, 1,138 ms for Flash Attention, 1,444 ms for Toaster, and 4,097 and 7,980 ms for Titans at $C = 32$ and $C = 16$.

Default configuration. Figure 4 times TOASTER at the state-matched width $d_h = 64$. To connect the throughput to the quality results, we also time the default configuration used in every other experiment ($d_h = 4d_k = 256$; Appendix G.4), with the same update rules and the same protocol; Table 8 lists the step times of all configurations. Widening the hidden memory four-fold, which about quadruples the state per head (Appendix G.9), increases the step time by only 7–20% over the state-matched TOASTER. The default configuration takes 1.25–$1.48\times$ the step time of KDA ($1.42\times$ at 32K) and 1.36–$1.73\times$ that of Flash Attention, and its step time grows by $1.40\times$ from 2K to 32K, against $1.23\times$ for KDA and 3.4–$4.5\times$ for Titans. Although it stores about four times the state of the Titans memory, it is faster than Titans at every context length except 4K, where Titans at $C = 32$ is 0.7% faster (1,215 against 1,223 ms); at 32K it is $2.7\times$ and $5.2\times$ faster than Titans at $C = 32$ and $C = 16$.

Most of the overhead over KDA is thus present already at $d_h = 64$ (1.12–$1.33\times$ KDA) and does not come from the wider hidden memory. It is also larger than the operation counts suggest: at 145M the memory kernels account for about 15% of a block's operations for the default TOASTER and 2% for KDA (Table 7). We expect the remainder to come from the unfused implementation described above and from the larger state traffic noted in Appendix F, but we have not profiled these sources separately. At 1.2B the kernels' share falls to about 8% for TOASTER and 1% for KDA, so we expect a smaller relative overhead at the scale of our main results; we have timed only the 145M configuration.

Table 8: Training-step time (ms) at a fixed budget of 512K tokens per step, measured on the 145M configuration. The first TOASTER row is the state-matched configuration of Figure 4; the second is the default configuration of all quality experiments. Both advance the hidden memory by apply-then-update and the output memory by the KDA kernel. The last two rows give ratios of step times.

Model	Setting	2K	4K	8K	16K	32K
Flash Attention		639	796	817	1,027	1,138
KDA	$C = 32$	880	905	948	990	1,085
Titans	$d_h = 64$, $C = 16$	1,793	2,098	2,808	4,432	7,980
Titans	$d_h = 64$, $C = 32$	1,202	1,215	1,641	2,399	4,097
TOASTER	$d_h = 64$, $C = 32$ (state-matched)	990	1,023	1,233	1,303	1,444
TOASTER	$d_h = 256$, $C = 32$ (default)	1,105	1,223	1,325	1,463	1,544
Default / state-matched TOASTER		1.12	1.20	1.07	1.12	1.07
Default TOASTER / KDA		1.25	1.35	1.40	1.48	1.42

G.9 ABLATION PROTOCOL

All ablations use the 145M configuration of Table 5 (12 layers, width 1024, 16 heads) with the same data, token budget (9.2B tokens), optimizer, learning-rate schedule, and hardware configuration as the corresponding run of the ladder, and change one component of the default TOASTER configuration at a time: (i) the homogeneous-coordinate bias of Section 3.4, removed; (ii) the hidden width $d_h \in \{64, 128, 192, 256\}$, i.e., one to four times d_k; (iii) the activation $\sigma \in \{\text{identity}, \tanh, \text{SiLU}, \text{leaky ReLU}, \text{GELU}\}$; (iv) the error scale $\gamma \in \{0, 0.001, 0.005, 0.01, 0.05, 0.1\}$ of Eq. (6), which bounds each coordinate of the hidden correction and thus sets the effective step of the hidden layer; and (v) the update rule of the hidden memory, the WY kernel (line 5 of Algorithm 1) in place of the default apply-then-update step. Every row is compared with one run of the default configuration (C4 perplexity 15.687). The KDA reference is the 145M run of the scaling ladder (Figure 2). Each run is evaluated once, at the end of training, on the C4 validation split at the 4,096-token context. The state per head is $(d_k + 1)d_h + (d_h + 1)d_v$ with the bias (33,088 at the default width; 8,320, 16,576, and 24,832 at $d_h = 64$, 128, and 192) and $d_k d_h + d_h d_v = 32{,}768$ without it. With the identity activation the two memories compose to a single linear map of rank at most d_k, so that row isolates the effect of the nonlinearity from that of the larger state.

Discussion. The rows of Table 12 separate three sources of the 0.31 gain over KDA. The first is a larger state: KDA and TOASTER have nearly the same parameters (Appendix G.2), but TOASTER holds about eight times KDA's state, and perplexity improves monotonically as the hidden width

grows from d_k to $4d_k$ (15.90 to 15.69). A cascade with the identity activation, which has this state but, like KDA, reads through a linear map of rank at most d_k, recovers 0.20 of the gain (part of it may come from TOASTER's bias, learned initial states, and gate scales; Section 3.4, Appendix G.4). The second is the nonlinearity, which adds the remaining 0.11 by letting the read use the wider state; GELU is the best of the five activations tried. Most of this remains with $\gamma = 0$, where the hidden memory receives no error signal and changes only through its forget gate, so that TOASTER is a delta-rule memory over a learned nonlinear feature map of the key (Liu et al., 2026), and reaches 15.71. The third is target propagation, which lets the hidden memory learn in context as well: it improves on $\gamma = 0$ by 0.02 with the default apply-then-update step and by more than three times as much with the WY kernel (15.64, last row), which lets each write see the earlier ones. The error scale is otherwise flat over 0.001–0.01 and degrades only at larger values, and removing the homogeneous-coordinate bias costs more than halving the width. At 145M, most of the gain thus comes from a wide nonlinear memory that runs on linear-memory kernels, and learning in both layers adds to it, most when the hidden memory is advanced exactly; the WY kernel cost 1.3–$1.4\times$ the step time in preliminary runs, which is why the cheaper rule is the default (Section 3.4).

H FULL RESULTS

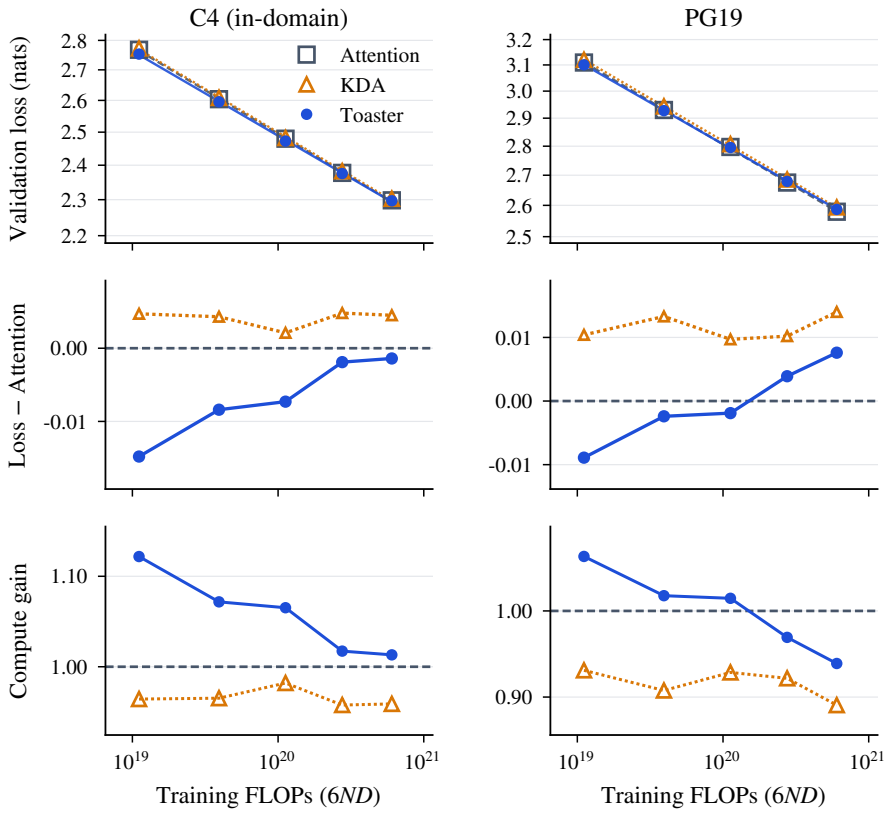


Figure 6: **Scaling ladder on C4 and PG19** (single runs). Top: validation loss against training compute ($6ND$ FLOPs) with the power-law fits of Appendix G.2. Middle: loss minus Attention's loss at the same compute (below zero is better than Attention). Bottom: compute-efficiency gain relative to Attention, Eq. (21), at the measured sizes.

Table 9: Validation loss (nats per token) and perplexity across the scaling ladder (lower is better; best per row within each block in bold), and the difference in loss between TOASTER and KDA and between TOASTER and Attention (negative favors TOASTER). Single runs.

	Loss (nats)			Perplexity			Δ loss of TOASTER vs.	
Corpus / size	Attention	KDA	TOASTER	Attention	KDA	TOASTER	KDA	Attention
C4 (in-domain)								
145M	2.7678	2.7725	**2.7530**	15.92	16.00	**15.69**	−0.0195	−0.0148
300M	2.6039	2.6082	**2.5955**	13.52	13.57	**13.40**	−0.0127	−0.0084
550M	2.4799	2.4820	**2.4726**	11.94	11.97	**11.85**	−0.0094	−0.0073
850M	2.3774	2.3822	**2.3755**	10.78	10.83	**10.76**	−0.0067	−0.0019
1.2B	2.2982	2.3027	**2.2968**	9.96	10.00	**9.94**	−0.0059	−0.0014
PG19								
145M	3.1094	3.1198	**3.1005**	22.41	22.64	**22.21**	−0.0193	−0.0089
300M	2.9299	2.9432	**2.9275**	18.73	18.98	**18.68**	−0.0157	−0.0024
550M	2.7973	2.8070	**2.7954**	16.40	16.56	**16.37**	−0.0116	−0.0019
850M	**2.6753**	2.6855	2.6792	**14.52**	14.66	14.57	−0.0063	+0.0039
1.2B	**2.5793**	2.5933	2.5869	**13.19**	13.37	13.29	−0.0064	+0.0076

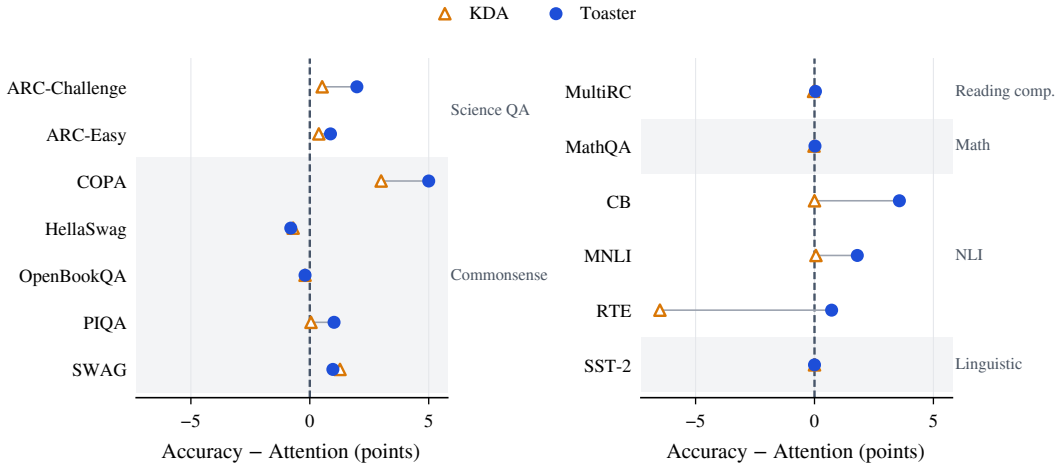


Figure 7: **Per-task three-shot accuracy relative to Attention at 1.2B** (points; right of the dashed line is better than Attention; single runs). TOASTER is above Attention on 10 of 13 tasks and tied on SST-2; its largest margins (COPA, CB, RTE) fall on the smallest evaluation sets, and most other differences are within one binomial standard error (Appendix G.5).

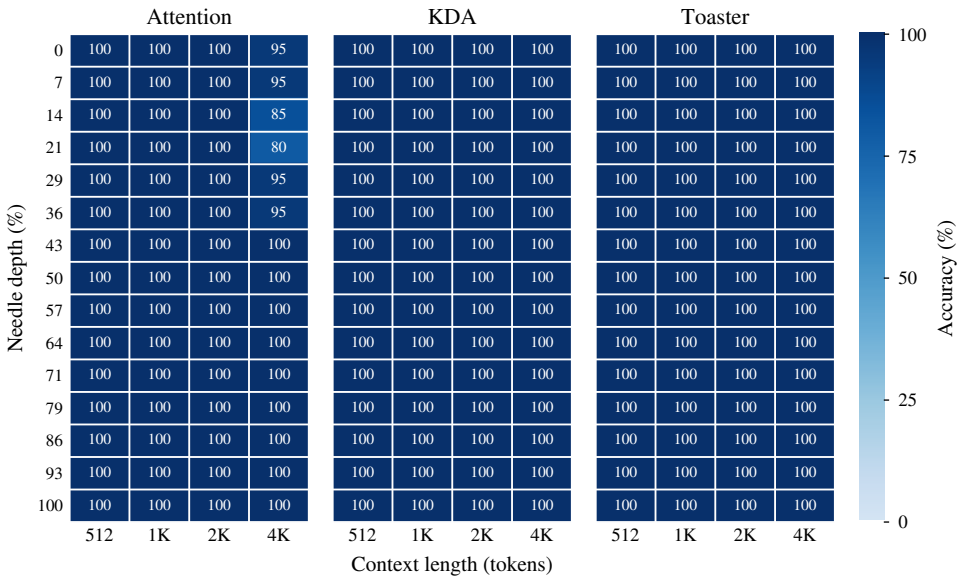


Figure 8: **Pressure test on S-NIAH-1 (passkey) at 1.2B.** Accuracy (%) for every needle depth (0%: needle at the start of the context, farthest from the query) and context length (Appendix G.7).

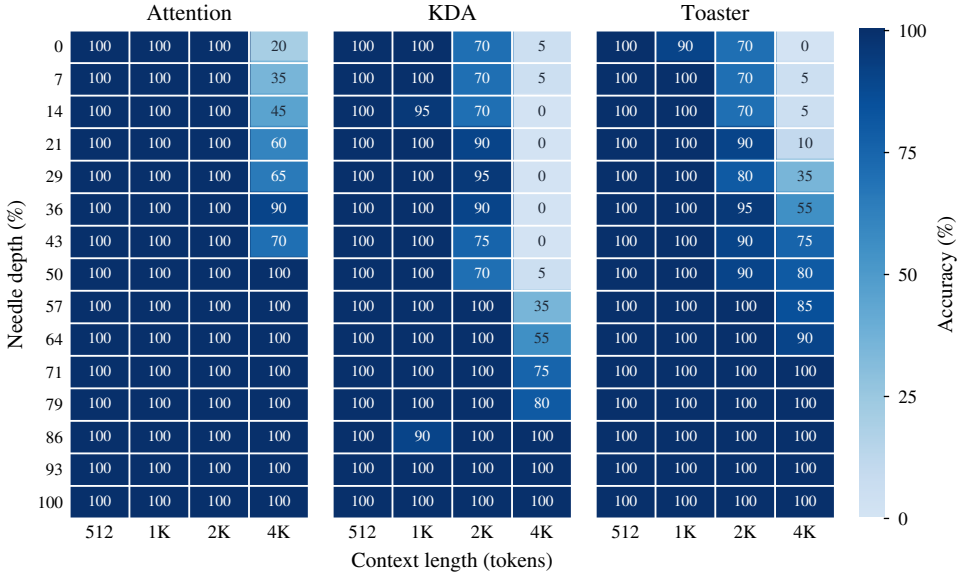


Figure 9: **Pressure test on S-NIAH-2 (number in essays) at 1.2B.** Same layout as Figure 8.

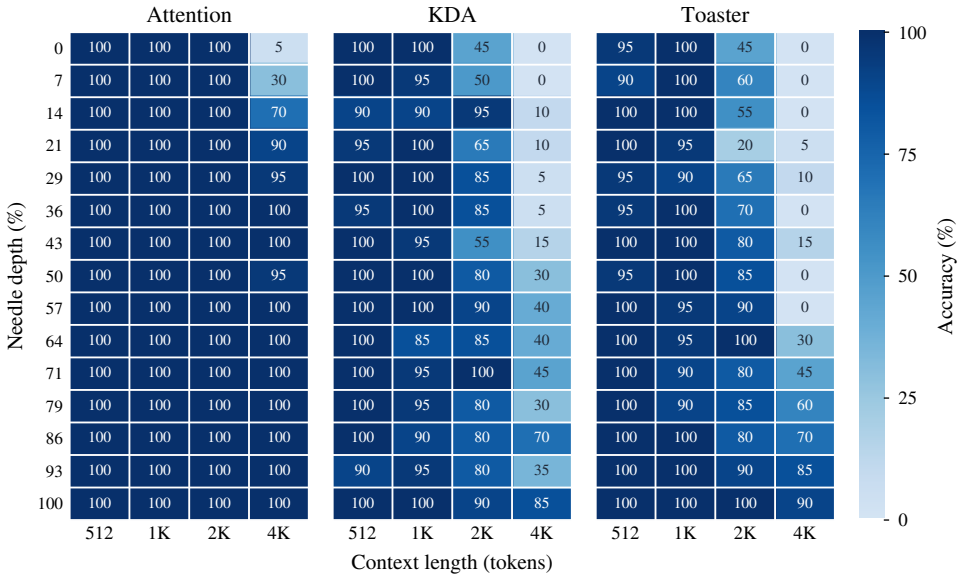


Figure 10: **Pressure test on S-NIAH-3 (UUID in essays) at 1.2B.** Same layout as Figure 8.

Pressure test. Figures 8–10 show every cell of the test. Up to 1K tokens every cell is at or near 100% for all models and tasks, and the passkey task (S-NIAH-1) stays at 100% at every length for the recurrent models. Where the haystack is real text, accuracy at 4K depends on the depth of the needle. On S-NIAH-2, TOASTER retrieves at least 75% of the needles placed at depth 43% or later and all of those from depth 71% on, whereas KDA reaches 75% only from depth 71% and 100% from 86%; on S-NIAH-3 both recurrent models fail for needles in the first half of the context and recover only towards its end (TOASTER above 60% from depth 79%, KDA above 60% at two depths). Attention loses only the needles in the first 15–30% of the context. At 2K the same ordering appears in milder form.

Table 10: Three-shot accuracy (%) on all 13 downstream tasks at 1.2B (nominal), with the average within each skill category. **Bold**: best per row; shaded: second best; three-way ties (SST-2) are left unmarked.

Task	Attention	KDA	TOASTER
Science QA			
ARC-Challenge	30.64	31.16	**32.62**
ARC-Easy	60.68	61.06	**61.55**
average	45.66	46.11	**47.08**
Commonsense reasoning			
COPA	67.00	70.00	**72.00**
HellaSwag	**62.03**	61.33	61.23
OpenBookQA	**38.60**	38.40	38.40
PIQA	75.14	75.19	**76.16**
SWAG	61.20	**62.47**	62.17
average	60.79	61.48	**61.99**
Reading comprehension			
MultiRC	57.16	57.12	**57.20**
average	57.16	57.12	**57.20**
Mathematical reasoning			
MathQA	21.70	21.68	**21.72**
average	21.70	21.68	**21.72**
NLI & entailment			
CB	50.00	50.00	**53.57**
MNLI	31.93	31.99	**33.73**
RTE	53.79	47.29	**54.51**
average	45.24	43.09	**47.27**
Linguistic understanding			
SST-2	49.08	49.08	49.08
average	49.08	49.08	49.08
Average over 13 tasks	50.69	50.52	**51.84**

Table 11: Length extrapolation at 1.2B: mean loss (nats per token) of each model in 4K-token windows of the 20K-token sequences of Figure 3, and the difference between TOASTER and KDA (negative favors TOASTER; the per-window standard error of the difference is about 0.001 nats). The training context is 4K tokens.

Window	Attention	KDA	TOASTER	TOASTER − KDA
0–4K	2.579	2.589	2.582	−0.008
4–8K	2.673	2.493	2.479	−0.014
8–12K	3.444	2.491	2.481	−0.010
12–16K	6.012	2.480	2.469	−0.011
16–20K	6.756	2.497	2.488	−0.009
4–20K	4.721	2.490	2.479	−0.011

Table 12: **Ablation study** at 145M (9.2B tokens; Appendix G.9). Each row changes one component of the default TOASTER (bias, $d_h = 4d_k$, GELU, $\gamma = 0.01$, apply-then-update) and reports C4 perplexity and its difference Δ to the default (red: worse, blue: better). The default row is the ablation's own run (15.687; the ladder's 145M run gives 15.690).

TOASTER ablations (145M)	C4 ppl (↓)	Δ
KDA (linear memory)	16.00	+0.31
TOASTER (default)	15.69	–
Bias		
w/o. homogeneous-coordinate bias	15.86	+0.17
Hidden width		
$d_h = d_k$ (64)	15.90	+0.22
$d_h = 2d_k$ (128)	15.76	+0.07
$d_h = 3d_k$ (192)	15.72	+0.04
Activation σ		
identity (linear cascade)	15.80	+0.11
tanh	15.82	+0.13
SiLU	15.79	+0.10
leaky ReLU	15.71	+0.02
Error scale γ (hidden-layer step)		
$\gamma = 0$ (no hidden-layer error)	15.71	+0.02
$\gamma = 0.001$	15.69	0.00
$\gamma = 0.005$	15.68	−0.01
$\gamma = 0.05$	15.70	+0.02
$\gamma = 0.1$	15.74	+0.05
Hidden-memory update		
WY kernel instead of apply-then-update	15.64	−0.05