

Gated KalmaNet: A Fading Memory Layer Through Test-Time Ridge Regression

Liangzu Peng^{1,*} Aditya Chattopadhyay^{2,†} Luca Zancato² Elvis Nunez² Wei Xia² Stefano Soatto²

University of Pennsylvania¹ AWS Agentic AI²

lpenn@seas.upenn.edu

{achatto, zancato, elvisnun, wxia, soattos}@amazon.com

Abstract

As efficient alternatives to softmax Attention, linear state-space models (SSMs) achieve constant memory and linear compute, but maintain only a lossy, fading summary of the past, often leading to inferior performance in recall oriented tasks. We propose Gated KalmaNet (GKA), a layer that reduces this gap by accounting for the full past when predicting the next token, while maintaining SSM-style efficiency. GKA achieves this by solving an online ridge regression problem at test time, with constant memory and linear compute cost in the sequence length. Drawing inspiration from the Kalman Filter, we iteratively solve the online ridge regression problem. However, a critical insight is that standard Kalman filter equations are numerically unstable in low-precision environments (like bfloat16) and difficult to parallelize in modern hardware. We address both challenges through two key innovations: (1) an adaptive regularization strategy with input-dependent gating that controls the condition number of the ridge regression, ensuring numerical stability while balancing memory retention. And (2) the use of Chebyshev Iteration instead of other conventional iterative solvers, which we demonstrate to be more stable in low-precision settings. To further improve scalability, we develop a hardware-aware chunk-wise implementation of Chebyshev Iteration along with custom kernels for backpropagating through our adaptive regularization and gating mechanisms. Empirically, GKA shows strong language understanding capabilities on short-context tasks outperforming existing SSM layers (like Mamba2, GLA and Gated DeltaNet). On long-context, GKA excels at real-world RAG and LongQA tasks up to 128k tokens, achieving more than 10% relative improvement over other fading memory baselines.

1. Introduction

Large Language Models (LLMs) powered by (softmax) Attention mechanisms [57] have revolutionized sequence modeling through their ability to form rich associations within their context window. However, a fundamental challenge that LLMs face is that their time complexity scales quadratically and storage grows linearly with their input length.

Recent years have seen intense efforts to develop Attention alternatives. Among them, memory layers based on linear State-Space models (SSMs) have grown popular for their linear-time computation and constant storage cost in the sequence length [9, 64]. These SSMs find inspirations from classic techniques in adaptive signal processing, and integrating them into modern SSMs leads to principled layer design and enhanced performance [34, 66, 71]. However, pure SSM models still underperform Attention in many settings, especially on long-context tasks. This gap is a consequence of their different memory mechanisms: SSMs have a *fading* fixed dimensional *lossy* state of the past, while Attention has an *eidetic* ever increasing KV-cache state [71].

To bridge this gap, we aim at designing a memory layer that enjoys the efficiency of linear SSMs while performing computation conditioned on the exact past. Towards this goal, we first draw insights from the *Kalman filter* (KF) [28]. In signal processing terms, KF computes the most recent state conditioned on all data seen thus far, and, under mild assumptions, KF is optimal in the *Maximum A-Posteriori* (MAP) sense. In the LLM context, we use KF to update the state of an SSM layer and predict its output based on all past inputs. However, integrating KF into such a layer is non-trivial and faces two challenges:

- *Parallelizable Training.* KF is an online algorithm and needs to be parallelized to fully utilize modern hardware that is highly optimized for large-scale LLM training.
- *Numerical Stability.* KF involves matrix inversion, which can be numerically unstable in low precision arithmetic.

In this work, we propose *Gated KalmaNet* (GKA), a memory layer that incorporates KF into its design and is both numerically stable and trainable on highly parallelizable hardware. We start by observing that the KF recursion solves a test-time

*Work done during an internship at AWS Agentic AI.

†Correspondence to achatto@amazon.com

ridge regression problem. Then, to solve such a regularized problem stably, we make the following choices:

- At the modeling level, we adaptively choose the regularization strength of our test-time objective function based on the Frobenius norm of the regularized data covariance. With this choice we can easily upper bound the condition number of the optimization problem.
- At the algorithmic level, we note that exact solvers (e.g., `torch.linalg.solve`) are hard to parallelize (in a chunk-wise manner), so we resort to the classic *Chebyshev Iteration* (CH), which we show has high numerical accuracy and fast convergence compared with alternatives such as (accelerated) gradient descent and conjugate gradient.

To make GKA scalable and efficient, we implement CH with adaptive regularization in Triton in a hardware-aware, chunk-wise manner. Our technical novelty here includes deriving a chunk-wise implementation that back-propagates through the Frobenius norm, for which the difficulty is the presence of a *nested* recurrence. Furthermore, we combine CH with a *gating mechanism* that decides the regression residual weights in an input-aware and time-varying fashion, enhancing the contribution of recent inputs and smoothly fading out distant contexts. Overall, to the best of our knowledge, this is a first adoption of the CH method for training sequence modeling layers in LLMs stably at scale.

Finally, we demonstrate the efficacy of GKA on numerous LLM benchmarks. For example, on synthetic recall tasks (MQAR) [1], our method achieves the highest recall accuracy among other state-of-the-art linear SSMs including Mamba2 [9] and (Gated) DeltaNet [66, 67]. Also, GKA outperforms existing SSMs on several (short-context tasks from LM-Harness [15]) and long-context tasks (from RULER [25] and HELMET [68]). Specifically, GKA improves upon SSM baselines by at least 10% on real-world long-context tasks like Retrieval-Augmented Generation and Long Question-Answering tasks up to 128k tokens.

2. Prior Work and Preliminaries

In this section we briefly review prior work and preliminaries that will set the stage for motivating our choice of designing an SSM layer based on the Kalman Filter. For a more detailed exposition of related work refer Section A.

(Softmax) Attention. At each time t , Attention [57] linearly projects the t -th input token to obtain three vectors, named *query* q_t , *key* k_t , *value* v_t respectively. Then, it outputs a vector $y_t \in \mathbb{R}^D$ as a convex combination of all values seen so far, with coefficients c_1, \dots, c_t given by inner products of the current query q_t with all seen keys and a softmax mapping:

$$y_t = \sum_{i=1}^t c_i v_i, \quad c_i := \frac{\exp(\frac{k_i^\top q_t}{\sqrt{D}})}{\sum_{i=1}^t \exp(\frac{k_i^\top q_t}{\sqrt{D}})}. \quad (\text{Attn})$$

From an optimization perspective, (Attn) can be viewed as solving the following *regression* objective¹,

$$y_t = \underset{v}{\operatorname{argmin}} \sum_{i=1}^t \exp\left(\frac{k_i^\top q_t}{\sqrt{D}}\right) \cdot \|v - v_i\|_2^2. \quad (1)$$

The success of (Attn) is often attributed to its ability to perform verbatim retrieval of relevant context from the entire past. Here, the past refers to the entire key-value pairs observed thus far, also known as the *KV-cache*, which grows linearly with time t . Moreover, the computation is also linear at each time t , and doing so for all t results in a quadratic time complexity. This high computation and storage cost of Attention makes its use prohibitive in long context scenarios.

Linear State-Space Models (SSMs). The high computation cost of (Attn) has motivated a flurry of work developing new LLM layers, like SSMs, with linear rather than quadratic cost. Most SSMs maintain a *state matrix* $S_t \in \mathbb{R}^{D \times D}$ and update it at each time step via a linear recursion of the form

$$S_t = \gamma_t \cdot S_{t-1} + \beta_t \cdot v_t k_t^\top, \quad y_t = S_t q_t, \quad (\text{Linear-SSM})$$

where γ_t, β_t are typically in $[0, 1]$. Unlike the verbatim lookup of (Attn), here (Linear-SSM) essentially compresses the entire KV-cache into a fixed-dimensional representation S_t . Subsequent computation of the output y_t relies on S_t and no longer on the exact past. This results in a constant cost of storage and computation at every timestep.

In many linear SSMs (e.g., RetNet [54], Mamba2 [9]), the use of γ_t and β_t is often heuristic and finds inspirations from nonlinear recurrent neural networks [23]; in that light, γ_t and β_t are called *forgetting* and *input* gates, respectively. This basic form of (Linear-SSM) has been generalized by replacing γ_t with a diagonal matrix (GLA [64], RWKV-6 [44], Longhorn [34]) or *low-rank-plus-identity* matrix (Gated DeltaNet [51, 66, 67], DeltaProduct [53], RWKV-7 [45]).

Similarly to that of (Attn) the case with *low-rank-plus-identity* matrices can often be justified from an optimization perspective. For example, Gated DeltaNet [51, 66] updates the state via (I is the $D \times D$ identity matrix)

$$S_t = \gamma_t \cdot S_{t-1} (I - \beta_t k_t k_t^\top) + \beta_t \cdot v_t k_t^\top, \quad (\text{GDN})$$

which can be viewed as applying one gradient descent step with stepsize β_t and initialization $\gamma_t S_{t-1}$ to the objective

$$\min_S \|S k_t - v_t\|_2^2. \quad (2)$$

The objectives of (GDN) (2) and (Attn) (1) are prime examples that expose a general distinction between linear SSMs and (Attn): The former updates its state based on a regression objective that considers only the previous *lossy* state

¹Concretely, for keys and queries of unit norm, Attention is precisely the *Nadaraya-Watson estimator* [39, 62] with the Gaussian kernel to approximate the conditional expectation of the value given a query; cf. [4, 58].

and the current time step, whereas the latter uses the entire, exact KV-cache to solve its regression objective (1).

We hypothesize this myopic view of SSM objectives results in their lower performance and limited long-context abilities. We then ask: *What is an objective or, equivalently, a recursion that considers the entire past as (Attn) while still being solvable in linear time as in (Linear-SSM)?*

3. A Linear SSM Inspired by the Kalman Filter

In Section 3.1 we show how the Kalman Filter (KF) gives insights into a new linear SSM layer that takes all past time instants into account. In Section 3.2 we explain the numerical and efficiency challenges of building such a layer.

3.1. Motivation from Kalman Filter

KF is an established online approach that takes the exact past into account to optimally solve a *weighted ridge regression* objective (e.g., see [46, Proposition 2 & Lemma 3]). In our context, this means that the *optimal* state

$$S_t = \underset{S \in \mathbb{R}^{D \times D}}{\operatorname{argmin}} \lambda \cdot \|S\|_F^2 + \sum_{i=1}^t \eta_i \cdot \|S k_i - v_i\|_2^2 \quad (3)$$

can be computed by the KF recursion

$$S_t = S_{t-1} - \frac{(S_{t-1} k_t - v_t) k_t^\top \Phi_{t-1}}{1/\eta_t + k_t^\top \Phi_{t-1} k_t}, \quad (\text{KF})$$

where η_t is the weight for the t -th key-value pair, and Φ_{t-1} is the Hessian inverse of (3) at time $t-1$ (Φ_{t-1} itself can be continually updated via the *Woodbury matrix identity*). It is now clear that objective (3) takes the entire KV-cache into account, similarly to (Attn). It is also clear that (KF) is an efficient update scheme similarly to (Linear-SSM); indeed, (KF) is also a *low-rank-plus-identity* form (cf. (GDN)).

A key difference from (Linear-SSM) is that (KF) leverages second-order information from Φ_{t-1} to solve (3) optimally, whereas (Linear-SSM) relies on instantaneous objectives akin to (2) (cf. [66, Table 2]). It is in this sense that we say (KF) is more expressive than other (Linear-SSM) or (GDN). We now detail the differences in the objectives of (KF) and (Attn):

- (KF) computes a parametric linear estimator that enables a constant-sized memory, while (1) computes a non-parametric point estimate that entails storing the full cache.
- In (1), the weights of the same residual vary over time as the queries differ at each time, while in (3) i -th weight η_i is constant once observed at time i . The former results in quadratically many weights—thus a quadratic time complexity—and the latter linearly many.
- In (3), the regularizer $\lambda \cdot \|S\|_F^2$ prevents overfitting our state to key-value pairs, as only a finite amount of “information” can be stored in a constant-sized memory beyond which will result in “fuzzy” recall. In this light, λ can be thought of as controlling the memorization “capacity” of the state.

3.2. Hurdles Towards Scalable Kalman Filter SSMs

Despite its optimality and (sequential) computational efficiency the (KF) recursion lacks a hardware-aware implementation that leverages parallelism in modern Tensors Cores. Moreover, for long sequences it can lose numerical precision due to division (and more significantly due to how the Hessian inverse Φ_t is updated). The final hurdle is conceptual: Fixing weights η_i and regularization λ over time as in (3) might make a layer less expressive.

We are aware of the use of (KF) or (3) in neural network training three decades ago [52] or in deep continual learning recently [36, 47, 72]. We are also aware of the recent mentioning of (3) or efforts towards solving it, which go by the name *test-time optimization* [59, 61]. However, to the best of our knowledge, none of the prior work has fully addressed the above hurdles that need to be solved to design an SSM layer that is trainable in parallel, numerical well-behaved, and sufficiently expressive. In particular, both [59] and [61] have overlooked a basic numerical concern: The worst-case numerical error in solving (3) can be $\epsilon \cdot \kappa$ [19], where κ is the condition number of the Hessian in (3) and ϵ the machine precision; since $\epsilon \approx 0.007$ (bf16), (3) has to be regularized *strongly* for κ and the worst-case error to be small, regardless of algorithmic choices to solve (3). Indeed, the regularization enforced in [59] sets λ to be lower bounded by 0.25, but this is not sufficient: Their κ is as large as 500 [59, Fig. 13], implying a worst-case error of 3.5 (The implementation of [59] available on GitHub is numerically vulnerable; we failed to train it without NaNs in various settings.). Also, the regression objective in [61] has no regularization, which makes it numerically ill-posed for low-precision training.

4. Gated KalmaNet (GKA)

We propose *Gated KalmaNet* (GKA) to address the above hurdles: We enhance numerical stability via adaptive regularization and the classic *Chebyshev Iteration* (CH), increase expressivity of KF via a standard gating mechanism, and improve parallelism via a hardware-friendly implementation.

4.1. CH with Adaptive Regularization & Weighting

Motivation. As alluded earlier, solving (3) via (KF) is sequential in nature, and here we consider alternatives amenable to parallelizable training. Our first step towards this is to write down a closed form solution to (3) and compute the output

$$y_t = S_t q_t = \left(\sum_{i=1}^t \eta_i v_i k_i^\top \right) \left(\sum_{i=1}^t \eta_i k_i k_i^\top + \lambda I \right)^{-1} q_t.$$

With the weighted covariances $U_t := \sum_{i=1}^t \eta_i v_i k_i^\top$ and $H_t := \sum_{i=1}^t \eta_i k_i k_i^\top$, we note that y_t can be computed via first solving $(H_t + \lambda I)x = q_t$ for x and then left-multiplying

Algorithm 1: Chebyshev Iteration to solve $H\xi = q$

1 Input: $H \in \mathbb{R}^{D \times D}$, $q \in \mathbb{R}^D$, eigenvalue bounds L, μ with $L \geq \mu > 0$, number of iterations r ;
2 Initialize: $\rho \leftarrow \frac{L-\mu}{L+\mu}$; $\omega_0 = 0$; the first two iterates $\xi_{-1} \leftarrow 0, \xi_0 \leftarrow \frac{2q}{L+\mu}$;
3 For Loop ($i = 1, \dots, r$):
$$\omega_i \leftarrow \frac{4}{4 - \rho^2 \omega_{i-1}} \quad (\text{weight schedule})$$
$$\xi_i \leftarrow \xi_{i-1} - \frac{2 \cdot \omega_i}{L + \mu} (H\xi_{i-1} - q) \quad (\text{grad descent})$$
$$\xi_i \leftarrow \xi_i + (\omega_i - 1)(\xi_{i-1} - \xi_{i-2}) \quad (\text{momentum})$$

4 Output: ξ_r

U_t . An exact solver (e.g., `torch.linalg.solve`) can do so with high accuracy, by parallelizing over the batch dimension. However, it is inefficient here for two reasons. First, it takes $O(D^3)$ time for every t . Second, it requires explicitly forming and materializing all H_t 's, which would entail a large I/O cost. In light of this, we resort to first-order iterative methods that admit chunk-wise implementation without materializing all H_t 's, enabling parallelism over chunks and batches. Furthermore, they often take $O(D^2)$ time complexity per iteration and can converge quickly in a few iterations. The iterative method we choose is the *Chebyshev Iteration* (CH); we proceed to describe its basic idea, with a justification of using CH deferred to Section 4.2.3.

Chebyshev Iteration (CH). CH can be seen as an *accelerated gradient descent* method (AGD) that applies ([grad descent](#)) and ([momentum](#)) to the strongly convex objective $\frac{1}{2}\xi^\top H\xi - \xi^\top q$, that is to solve the optimality condition $H\xi = q$ (Algorithm 1). Different from AGD, CH incorporates a ([weight schedule](#)) and makes specific choices of different parameters; these choices makes CH optimal with the fastest convergence among first-order methods [43].

We now replace the above exact solver with CH:

$$\hat{x}_t = \text{CH}(H_t + \lambda I, q_t, r), \quad y_t = U_t \hat{x}_t.$$

Here, $\text{CH}(H_t + \lambda I, q_t, r)$ means r iterations of CH to approximately solve $(H_t + \lambda_t I)x = q_t$. To improve stability and expressivity, next we allow regularization λ and weight η_i to be time-varying and chosen adaptively. We write λ_t and $\eta_{t,i}$ to make their dependency in time t explicit, with $\eta_{t,i}$ being the weight of the i -th token at time t .

Adaptive Regularization. As mentioned, the condition number κ_t of $H_t + \lambda_t I$ has to be controlled for any method to be numerically stable. We choose λ_t to be proportional to the Frobenius norm $\|H_t\|_F$, that is to set $\lambda_t = a \cdot \|H_t\|_F$ for

some constant $a > 0$. An upper bound on κ_t now ensures:

$$\kappa_t = \frac{\lambda_{\max}(H_t) + \lambda_t}{\lambda_{\min}(H_t) + \lambda_t} \leq \frac{\|H_t\|_F + \lambda_t}{\lambda_t} = \frac{a + 1}{a}. \quad (4)$$

Here $\lambda_{\max}(H_t), \lambda_{\min}(H_t)$ are the maximum and minimum eigenvalues of H_t , respectively. Given this choice of λ_t , we set $L = \|H_t\|_F + \lambda_t$ and $\mu = \lambda_t$ for Algorithm 1.

Adaptive Weighting (Gating). We use weights $\eta_{t,i}$ that are exponentially decaying in time: For all $t \geq i$, we parameterize $\eta_{t,i} = \prod_{j=i+1}^t \gamma_j$, with each $\gamma_j \in [0, 1]$ learnable. The fading weights encode the “prior” of *recency bias* that has been shown to exist in LLMs [13] without even explicitly computing the weights from the query-key dot products as in ([Attn](#)). Similarly to ([Attn](#)), the weights on the residuals are now time-varying, but differently, the exponentially decay parameterization allows for linear-time implementation.

Forward Recurrence. We now summarize our recurrence which arms CH with adaptive regularization and weighting:

$$H_t = \gamma_t \cdot H_{t-1} + k_t k_t^\top, \quad U_t = \gamma_t \cdot U_{t-1} + v_t k_t^\top, \quad (\text{CH})$$
$$y_t = U_t \hat{x}_t, \quad \hat{x}_t = \text{CH}(H_t + \lambda_t I, q_t, r).$$

4.2. Chunk-wise Implementation

In this subsection, we describe our hardware-aware implementation for the forward + backward passes for ([CH](#)). More details can be found in Section B.

4.2.1. Forward Pass

Similarly to prior work [9, 64, 66], we now describe a *chunk-wise* implementation for ([CH](#)). In ([CH](#)), given U_t and \hat{x}_t , computing $y_t = U_t \hat{x}_t$ in a chunk-wise fashion is similar to that of ([Linear-SSM](#)); also similar is the calculation of $H_t \xi_{i-1}$ as needed in ([grad descent](#)). For these we refer the reader to [64, 66] for details. Our algorithmic novelty here is a chunk-wise computational formula for $\|H_t\|_F$, presented next.

Let T be the sequence length and C the chunk size such that $N := T/C$ is an integer. For $t = 0, \dots, N - 1$, write $[t] := tC$. The core idea of a chunk-wise implementation is as follows. First, we compute and store the *initial state* $H_{[t]}$ of every chunk. This gives us implicit access to $H_{[t]+c}$ via unrolling the recurrence of H_t for c steps and therefore allows us to carry out computation with $H_{[t]+c}$; for example, we can compute the matrix-vector product $H_{[t]+1}\xi$ via $H_{[t]}\xi + \gamma_1 k_{[t]+1} k_{[t]+1}^\top \xi$. This is without forming $H_{[t]+1}$ explicitly, thereby reducing the number of states to materialize on chip. To implement such a scheme, we need to precompute all $H_{[t]}$'s sequentially, and then do the computation with parallelism over chunks and within each chunk.

We now make this idea precise for computing all $\|H_t\|_F$'s within a chunk. Since the computation of each chunk is the same, we simplify by working with the first one where we have access to initial state H_0 , gates $\gamma_1, \dots, \gamma_C$, keys

$K_C = [k_1, \dots, k_C] \in \mathbb{R}^{D \times C}$, and we aim to compute $\|H_1\|_F, \|H_2\|_F, \dots, \|H_C\|_F$. With these notations, we first compute the C -dimensional vector $\zeta = [\zeta_1, \dots, \zeta_C]^\top$ of cumulative products of γ_i 's, with $\zeta_c = \prod_{i=1}^c \gamma_i$. Then, form the $C \times C$ upper triangular matrix M whose (i, j) -th entry $M_{j,c}$ is ζ_c / ζ_j ($\forall c \geq j$). Now, unroll the recurrence of H_c :

$$H_c = \zeta_c H_0 + \sum_{j=1}^c M_{j,c} k_j k_j^\top = \zeta_c H_0 + \sum_{j=1}^c M_{j,c} k_j k_j^\top.$$

Expanding $\|H_c\|_F^2$ gives the following sum of three terms:

$$\zeta_c^2 \|H_0\|_F^2 + 2\zeta_c \sum_{j=1}^c M_{j,c} k_j^\top H_0 k_j + \left\| \sum_{j=1}^c M_{j,c} k_j k_j^\top \right\|_F^2.$$

With ζ , the first term $\zeta_c^2 \|H_0\|_F^2$ is easily computed in parallel for all c . For the second term, we first compute the vector of quadratic forms $k_j^\top H_0 k_j$ for all j in parallel, broadcast it and multiply it with M element-wise, sum over each column, and multiply the result with 2ζ element-wise. Finally, with Gram matrix $G_C := K_C^\top K_C$, one verifies the third term can be computed in parallel for all c via the following pseudocode:

$$\text{column-sum}(((G_C \odot G_C)M) \odot M). \quad (5)$$

Here \odot denotes element-wise multiplication and the sum is over each column. Summing the three terms and taking the square root, we obtain $\|H_1\|_F, \dots, \|H_C\|_F$, as desired.

4.2.2. Backward Pass

Motivation. Typically, the backward pass is done automatically via `torch.autograd`. However, for iterative methods such as CH (Algorithm 1), `torch.autograd` would store some *activations* or intermediate iterates, entailing large storage cost. While in principle we can back-propagate through CH without storing any intermediate activations or iterates (by our trick of *reverting the CH iterations*, cf. Section C.1), under this trick it is difficult to compute all the gradients in a chunk-wise fashion. Therefore, we resort to the *implicit differentiation* trick, which is practically efficient and chunk-wise implementable, for backpropagation through the linear equations that CH approximately solves.

Implicit Differentiation. We derive the backward pass for our method with the standard implicit differentiation trick. It assumes we find an exact solution x_t^* to the equations $(H_t + \lambda_t I)x = q_t$. In the backward pass, we are given the gradient $dx_t^* := \frac{d\mathcal{L}}{dx_t^*}$ of some loss function \mathcal{L} , and need to compute the corresponding gradients at q_t, k_t, γ_t . For example, via the chain rule we obtain $dq_t := \frac{d\mathcal{L}}{dq_t}$ via

$$dq_t = (H_t + \lambda_t I)^{-1} dx_t^*, \quad (6)$$

that is to solve linear equations similarly to the forward pass. Since the forward pass computes an approximate solution \hat{x}_t via CH, we receive an approximate up stream gradient $d\hat{x}_t$ (not exactly dx_t^*). Thus we employ CH to obtain an approximate gradient $d\hat{q}_t = \text{CH}(H_t + \lambda_t I, d\hat{x}_t, r)$; cf. Table 1.

Backward Recurrence. Besides dq_t , we need to compute dH_t from which we obtain dk_t and $d\gamma_t$ via the chain rule. We describe $d\gamma_t$ in the Appendix. Here we analyze dk_t :

Lemma 1. With $\lambda_t = a\|H_t\|_F$, $w_t = \frac{2a \cdot (x_t^*)^\top dq_t}{\|H_t\|_F}$, we have

$$dk_i = \sum_{t \geq i} M_{i,t} (-dq_t (x_t^*)^\top - x_t^* dq_t^\top - w_t H_t) k_i. \quad (7)$$

With $A_i := \sum_{t \geq i} M_{i,t} \cdot dq_t (x_t^*)^\top$, we can compute the first two terms $-A_i k_i$ and $-A_i^\top k_i$ in (7), similarly to (Linear-SSM). Specifically, A_i satisfies the recursion

$$A_i = \gamma_{i+1} A_{i+1} + dq_i (x_i^*)^\top, \quad (8)$$

thus calculating $A_i k_i$ amounts to calculating $U_t q_t$ in (Linear-SSM); a difference is that the recursion here runs backwards.

Similarly, with $B_i = \sum_{t \geq i} M_{i,t} w_t H_t$, the third term in (7) can be written recursively as

$$B_i = \gamma_{i+1} B_{i+1} + w_i H_i, \quad o_i = B_i k_i. \quad (9)$$

Chunk-wise Recurrence. As indicated, a chunk-wise implementation for computing $A_i k_i$ is known. On the other hand, computing $B_i k_i$ is more challenging than $A_i k_i$, as the additive term $w_i H_i$ in the backward recursion (9) is not necessarily rank-1; rather, H_i itself is defined via the forward recursion in (CH). Our contribution here is a derivation for computing $B_i k_i$ efficiently in a chunk-wise manner.

We begin by unrolling B_i to B_{C+1} :

$$B_i = M_{i,C} \cdot \gamma_{C+1} B_{C+1} + B_i^{\text{intra}} \\ B_i^{\text{intra}} := \sum_{c=i}^C M_{i,c} \cdot w_c H_c \quad (10)$$

We next discuss the *intra-chunk* term $B_i^{\text{intra}} k_i$ and *cross-chunk* term $M_{i,C} \cdot \gamma_{C+1} B_{C+1}$ in succession.

Intra-chunk Computation. We now unroll H_c and obtain an expression B_i^{intra} more amenable to parallelism:

$$B_i^{\text{intra}} = \sum_{c=i}^C M_{i,c} \cdot w_c H_c \\ = \sum_{c=1}^C M_{i,c} w_c \left(\zeta_c H_0 + \sum_{j=1}^c M_{j,c} k_j k_j^\top \right) \\ = H_0 \sum_{c=1}^C M_{i,c} w_c \zeta_c + \sum_{j=1}^C k_j k_j^\top \sum_{c=1}^C M_{i,c} w_c M_{j,c}.$$

Table 1. **Implicit differentiation for computing dq_t .**

	forward pass	backward pass
exact	$x_t^* = (H_t + \lambda_t I)^{-1} q_t$	$dq_t^* = (H_t + \lambda_t I)^{-1} dx_t^*$
CG	$\hat{x}_t = \text{CG}(H_t + \lambda_t I, q_t, r)$	$d\hat{q}_t = \text{CG}(H_t + \lambda_t I, d\hat{x}_t, r)$
CH	$\hat{x}_t = \text{CH}(H_t + \lambda_t I, q_t, r)$	$d\hat{q}_t = \text{CH}(H_t + \lambda_t I, d\hat{x}_t, r)$

The coefficients of H_0 , written as b_i , are easily computed in parallel for all i via element-wise operations, broadcasting, and summing. The coefficient of $k_j k_j^\top$ is precisely the (i, j) -th entry of the matrix $M_w := M \text{diag}(w_1, \dots, w_C) M^\top$. Thus $[B_1^{\text{intra}} k_1, \dots, B_C^{\text{intra}} k_C]$ is equal to

$$H_0(\text{diag}(b_1, \dots, b_C) K_C) + K_C ((K_C^\top K_C) \odot M_w).$$

Here the *mask* M_w is in general a full matrix with no zero entries, as opposed to the triangular matrix in the case of (Linear-SSM). While the triangular mask in the backward pass allows the error feedback from future tokens to be leveraged for learning past tokens, here our full mask M_w allows all tokens to interact with all other tokens in the backward pass, which facilitates the information flow and learning.

Cross-chunk Computation. In (10), both γ_{C+1} and B_{C+1} are from the future chunks, thus we revise (10) into the cross chunk recursion of $\tilde{B}_{C+1} := \gamma_{C+1} B_{C+1}$ which allows us to maintain a single term \tilde{B}_{C+1} from the future:

$$\tilde{B}_1 = \zeta_C \cdot \tilde{B}_{C+1} + \tilde{B}_1^{\text{intra}}, \quad \tilde{B}_1^{\text{intra}} := \sum_{c=1}^C \zeta_c \cdot w_c H_c.$$

In our intra-chunk computation, we store the intra-chunk term $\tilde{B}_1^{\text{intra}}$ of all chunks, implement the above with a simple for loop, and collect the terms $\zeta_C \cdot \tilde{B}_{C+1} k_i$.

4.2.3. Comparison to Other Iterative Solvers

Here we validate our choice of Chebyshev Iteration (CH) by benchmarking it against other iterative methods.

Convergence in the Forward Pass. We generate random regression problems, which we solve via CH and 3 other baselines: gradient descent (GD), accelerated GD with Nesterov’s momentum (AGD), conjugate gradient (CG). GD and AGD are run with stepsizes that are optimal for regression problems. Fig. 1a shows CG converges the fastest within a few iterations, while CH reaches the same accuracy as CG at iteration 10 and eventually attains the smallest errors.

Stability of the Backward Pass. We then proceed and measure the gradient stability of CG and CH, whose backward passes are implemented either via implicit differentiation as per Table 1 (*impl*), or via `torch.autograd` (*auto*).

In Fig. 1b, CG (*impl*) as a standalone layer has its gradient close to that of the exact solver up to a 10^{-3} relative difference. In Fig. 1c, this difference is amplified to almost 1

in a 5-layer LLAMA where (*Attn*) is replaced with (3). This indicates CG (*impl*) completely deviates from the reference gradient (exact), defeating its purpose of training the network from the regression feedback. In contrast, the gradients of CH (*impl*) and CH (*auto*) are eventually close to that of the exact solver either as a single layer (Fig. 1c) or within multiple layers (Fig. 1d), up to a 10^{-6} difference. Moreover, the curves for CH (*impl*) and CH (*auto*) nearly overlap, suggesting that their gradients may be close. The following lemma confirms and formalizes this intuition (see Section C for a proof), thereby justifying our choice of CH over the alternatives:

Lemma 2. *Let dq_t be the exact gradient of q_t for CH, e.g., computed by CH (*auto*). Let $d\hat{q}_t$ be the gradient of CH (*impl*), computed as per Table 1. We have $dq_t = d\hat{q}_t$.*

4.3. Architectural Consideration

Our GKA layer in Fig. 2 includes two components (in green) on top of established practices (in blue). The CH component is described in Section 4.1, thus here we introduce the α -connection. First, the sigmoid activation ensures $\alpha_t \in [0, 1]$, so the output of the α -connection is a convex combination of the original query q_t and the output \hat{x}_t of CH. Second, it plays a similar role to residual connection, which establishes a direct path that facilitates the gradient flow and improves training; we show this is indeed the case in Section F.3. Finally, the full architecture for GKA is the standard Transformer, with its attention layer replaced by the GKA layer.

5. Experiments

In this section, we empirically validate the efficacy of our approach. We first evaluate memorization ability on synthetic associative recall tasks (Section 5.1). We then report training throughput of GKA (Section 5.2). Finally, we examine performance on short-context language understanding benchmarks such as commonsense reasoning and long-context modeling abilities in Section 5.3. The Appendix details our experimental settings (Section D) and ablations of various modeling choices (Section F, Section G).

Baselines. All experiments consider the following state-of-the-art linear SSM-based fading memory layers as baselines: Mamba2 [9], DeltaNet [66], Gated DeltaNet (GDN) [67], and Gated Linear Attention (GLA) [64]. Each of these layers rely on instantaneous objectives that depend on the previous *lossy* state and current tokens (e.g., (2)), as opposed to the entire history of tokens observed so far as in GKA. Finally, we contrast our results with (Softmax) Attention, which serves as our paragon. For our Attention-based model, we adopt the architecture proposed in Qwen3 models [63].

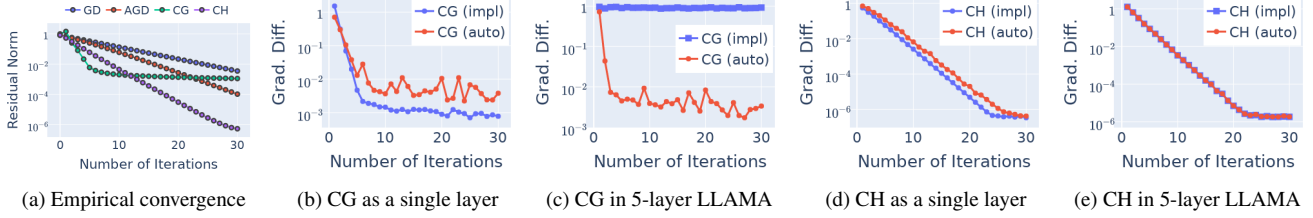


Figure 1. **CH converges with smaller errors than CG and is more numerically stable.** Convergence of different methods in residual norms during the forward pass with batch size 8, sequence length 2048, 8 heads, head dimension 128 (a), and relative gradient differences from the exact solver (`torch.linalg.solve`) to CG (b, c) or CH (d, e). The backward pass is via *implicit differentiation* (*impl*) or `torch.autograd` (*auto*); cf. Table 1. In (b, d) the gradients are those of $[q_t, k_t]$; in (c, e) the gradients are those of network weights.

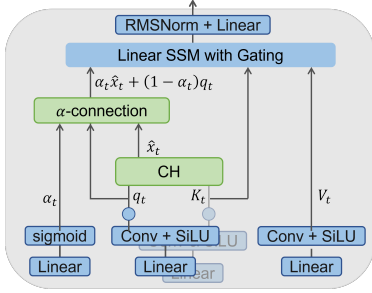


Figure 2. **Our GKA block.** Blue refers to established practices in the literature with the solid circles denote ℓ_2 normalization. Green components (CH and α -connection) are our proposals.

5.1. GKA on Synthetic Associative Recall Tasks

We first assess the capability of our models to recall information on the multi-Query Associative Recall (MQAR) task, a synthetic task introduced by Arora et al. [1]. This task presents the model with a sequence of key-value pairs to memorize, followed by a sequence of queries. For each query, the model must retrieve the corresponding key from memory and accurately recall its associated value. Attention based layers perform the best in this task, while SSM-based memory layers are known to struggle as their memory fades away as the context length grows.

We compare GKA with Attention and other linear SSM baselines on this task. For each memory layer type, we train 2-layer models on MQAR training data and evaluate on a held-out test set. We repeat this experiment for 4 different learning rates spanning from 10^{-4} to 10^{-2} . As shown in Fig. 3a, GKA improves upon every other linear SSM baseline at all sequence lengths and model dimensions considered. Note, the complexity of the task increases with increasing sequence length and number of key-value pairs, while larger model dimensions improve memorization capacity through increased state size. The success of our layer can be attributed to our modeling choice: unlike other fading memory designs (like GDN or Mamba2), we construct states based on the optimal MAP estimate conditioned on the entire history, enabling better retention of remote information.

5.2. Training Throughput of GKA

In Fig. 3b we measure the running time (forward + backward) of a single GKA layer and compare it with FlashAttention [18], DeltaNet, and GDN. Our layer achieves comparable running time to GDN, a state-of-the-art SSM layer, despite having a more computationally expensive state update equation (CH) than (GDN). This demonstrates that our chunk-wise parallelization strategy effectively compensates for the additional computational cost.

5.3. GKA on Language Modeling

5.3.1. Short-context Tasks

Setup. For this set of experiments, we construct 2.8B LLM models for each memory layers (GKA and baselines described in Section 5) by cascading blocks of mem + Multi-Layer Perceptron (MLP) blocks.² Hereby, we refer to the 2.8B models with the same name as the layer used to construct them. We then train each model on DCLM [32], a generic pre-training dataset for 100B tokens at 4K context length using the AdamW optimizer with a peak Learning Rate (LR) of 10^{-3} and gradient clipping of 1.0. We used the cosine LR scheduler with a warmup period of 5B tokens with a global batch size of 2M tokens. All models employ the GPT2 tokenizer with a vocabulary size of 50K tokens.

Tasks. Following prior works [64, 67, 71], to consider language modeling capabilities of our model we perform zero-shot evaluation on the following eight common-sense reasoning tasks from LM-Harness [15]: Arc-E, Arc-C, BoolQ, COPA, HellaSWAG, PIQA, SciQ, Winogrande. We also evaluate models on FDA and SWDE, real-world recall-intensive tasks which focus on extracting structured information like tagged content from raw text (for example, HTML files). All these tasks are relatively short ($< 2K$ tokens).

Results. We report our results in Table 2. GKA outperforms all fading memory baselines on average across all tasks owing to its ability to better manage its state via solving (3). In particular, GKA outperforms both GDN and Mamba2 on recall-intensive tasks (FDA and SWDE) by about 10%

²For Mamba2 baseline, we consider cascading blocks of Mamba2 layer alone since a single Mamba2 layer has the Mamba2 SSM and MLP.

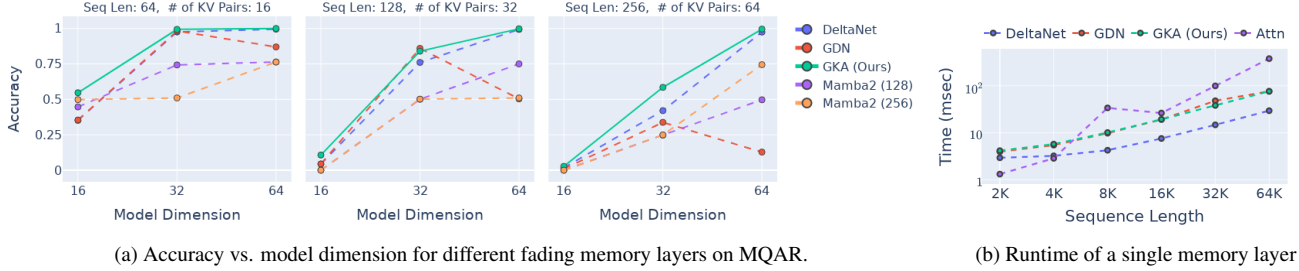


Figure 3. **MQAR results** (a) Each plot corresponds to a particular sequence length and number of key-value pairs for the model to memorize. **Runtime** (b) Runtimes are for a single forward + backward pass (8 heads, head dim 128, batch size 4, averaged over 20 runs).

Table 2. **On average GKA improves upon all fading memory baselines across all tasks.** We report results for zero-shot evaluation of 2.8B language models for short-context tasks. For each task, bold indicates highest value followed by underlined.

Model	ARC-C acc_n ↑	ARC-E acc_n ↑	BoolQ acc ↑	COPA acc ↑	HellaSWAG acc_n ↑	PIQA acc_n ↑	SciQ acc_n ↑	Winogrande acc ↑	FDA contains ↑	SWDE contains ↑	Avg
Transformer	32.25	56.10	64.28	80.00	60.96	73.56	79.50	61.72	58.53	72.28	63.92
Gated Linear Attention	27.82	50.80	52.57	78.00	48.83	70.13	69.60	54.54	2.81	20.43	47.55
DeltaNet	32.85	58.16	42.51	81.00	61.13	73.78	43.90	61.72	11.80	46.08	51.29
Mamba2	32.24	59.64	58.72	82.00	62.23	73.78	79.80	62.19	7.71	41.13	55.94
Gated DeltaNet	<u>32.59</u>	60.02	<u>62.75</u>	<u>82.00</u>	<u>62.80</u>	<u>74.32</u>	<u>80.60</u>	<u>62.35</u>	8.26	44.28	57.00
Gated KalmaNet (Ours)	32.51	<u>59.89</u>	61.68	85.00	63.84	74.81	83.20	64.17	<u>12.89</u>	<u>50.95</u>	<u>58.89</u>

(rel. improvement). We note that although GKA improves upon existing SSM layers there is still a gap with Attention-based Transformer especially on recall-tasks owing to the eidetic capabilities of Attention. Nevertheless, as discussed in Section 1 this improvement comes at a quadratic cost at training time, whereas our layer’s computational complexity is still comparable to existing SSM layers (cf. Section 5.2). In Section I we extend our results to Hybrid models (stack of SSM and Attention layers) and show that the gap with full Transformer models becomes negligible (while still benefiting the SSM’s computational advantages). Finally, in Section E we show that GKA exhibits stronger scaling with compute than other SSM baseline models.

5.3.2. Long-context Tasks

Setup. To enable long-context capabilities of our models, as is common practice, we perform continued pre-training of our 2.8B models obtained in Section 5.3.1 on 25B tokens of long documents at 128K context length (cf. Appendix). To the best of our knowledge we are the first to train and evaluate SSM models up to 128K context (e.g., previous work [67] only considered up to 4K/8K context).

Tasks. For long-context, we refrain from using perplexity as it is known to have limitations at assessing long-context performance of LLMs [14, 16, 40]. Instead, we turn to recently proposed benchmarks that mix synthetic and real datasets comprising several long-context tasks: synthetic Recall, Retrieval-Augmented Generation (RAG), Many shot In-Context Learning (ICL) and Long Question-Answering (LongQA). For synthetic Recall and LongQA we consider tasks from the RULER benchmark [25]. For RAG and ICL

we consider tasks from HELMET [68].

Results. Fig. 4 reports our results. GKA shows strong RAG and LongQA capabilities, outperforming all fading memory baselines by at least 10% (rel. improvement). Interestingly, on synthetic Recall tasks from RULER, GKA is competitive only at 4K context length and starts to fall behind afterwards. We attribute this divergence to the fundamental differences between these task types. While both RAG and LongQA can be thought of as finding relevant information in long streams of text, they involve more realistic linguistic patterns and semantic relationships that align with natural text distributions seen during pretraining. In contrast, synthetic Recall tasks require models to find specific words, numbers, or UUIDs verbatim from long contexts filled with random distractors. This artificial setting does not reflect natural text distributions. Since GKA computes MAP estimates of the latent state based on learned representations of observed tokens, it relies on its pretrained weights to determine which information is important to retain. The synthetic and unnatural structure of Recall tasks makes it difficult for the model to identify what should be retained, as pretrained knowledge provides little signal about importance in these artificial contexts. This suggests that our approach excels in realistic scenarios where pretrained knowledge about natural language structure can guide information selection, but struggles when the signal-to-noise distinction is purely artificial.

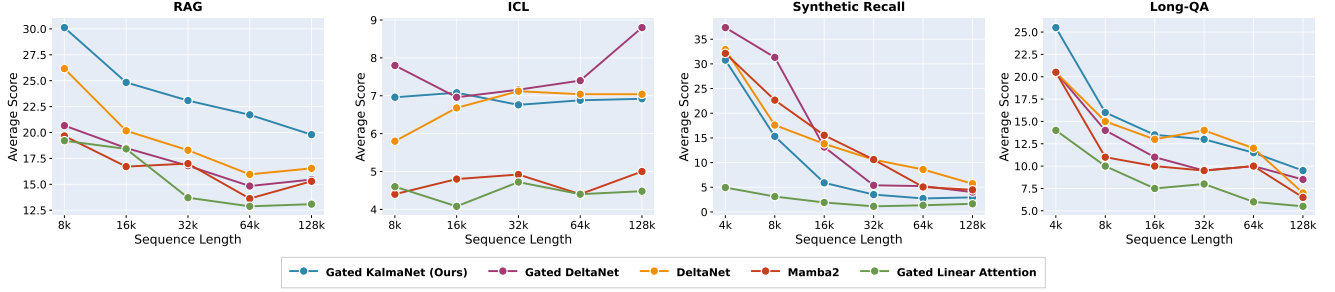


Figure 4. **Long Context Performance up to 128k tokens.** GKA achieves strong RAG and LongQA capabilities, outperforming all baselines by 10% in relative improvement. Interestingly, we observe that there is no clear winner Synthetic Recall. All models struggle to perform better than random chance on ICL.

6. Kalman Filter for *Optimally* Modelling Fading Memory

In this section, we show how the Kalman Filter (KF) provides a principled solution for constructing an optimal fading memory that accounts for the entire history. We begin by describing the standard Kalman Filter recurrence in the context of memory modeling. However, the KF has a fundamental limitation: its inherently sequential nature makes it impractical for large-scale training on modern hardware accelerators Section 3.2. To address this, we make simplifying assumptions that makes KF amenable to parallelization on modern hardware accelerators. We then demonstrate that several recent state-space models (DeltaNet, Gated DeltaNet, and Kimi Delta Attention) can be viewed as approximations to the KF recurrence. Specifically, these methods approximate the “optimal” Kalman gain matrix while ignoring dependencies on the past. In contrast, GKA computes the exact Kalman gain by considering the full history. This theoretical advantage translates to improved empirical performance, as we demonstrate in Section 5.

6.1. A Dynamical System for Fading Memory

The Kalman filter is a classical algorithm for online optimal inference in linear Gaussian State-Space Models. It gives a principled way to maintain and update a compact state as new noisy observations arrive. The latent state serves as a compressed “memory” of the past. More formally, it is a minimal sufficient statistic that makes past observations conditionally independent of future ones given the state.

We begin by describing a linear Gaussian model for fading memory.

$$\begin{aligned} s_t &= A_t s_{t-1} + B_t u_t + w_t, & w_t &\sim \mathcal{N}(0, Q_t) \\ v_t &= k_t^\top s_t + \mu_t, & \mu_t &\sim \mathcal{N}(0, r_t), \end{aligned} \quad (\text{LGM})$$

where $s_t \in \mathbb{R}^n$ is a latent state that summarizes the past, $u_t \in \mathbb{R}^n$ is the control input that updates the state and v_t is the scalar measurement observed at time t . $A_t, B_t \in \mathbb{R}^{n \times n}$ are the state transition and input selection matrices, and

$k_t \in \mathbb{R}^n$ is the emission (readout) vector. Finally, w_t and μ_t are Gaussian process and measurement noise, respectively.

Parameter interpretation. A_t and B_t control the forgetting (fading of the remote past) and input selectivity rates respectively, determining how the state evolves over time. The measurement noise μ_t naturally gives rise to gating mechanisms commonly used in modern SSM layers, as we will show in Section 6.4.

Extension to multi-channel measurements. In attention mechanisms, the memory consists of verbatim key-value pairs that can be queried to retrieve past information [71]. Similarly, we want our state to reconstruct past values from their corresponding keys. To achieve this, we extend to a matrix-valued state $S_t \in \mathbb{R}^{n \times d}$, where each column independently follows the dynamics in (LGM).

Specifically, for the i^{th} channel:

$$\begin{aligned} s_{t,i} &= A_{t,i} s_{t-1,i} + B_{t,i} u_{t,i} + w_{t,i}, & w_t &\sim \mathcal{N}(0, Q_{t,i}) \\ v_{t,i} &= k_{t,i}^\top s_{t,i} + \mu_{t,i}, & \mu_t &\sim \mathcal{N}(0, r_{t,i}), \end{aligned}$$

where (k_t, v_t) is the key-value pair at time t and $v_{t,i}$ is the i^{th} element of v_t . In what follows, we focus on a single channel and drop the subscript i from the state for notational clarity.

6.2. Kalman Filter for Optimal Inference

Given the model in (LGM) and a sequence of measurements $\{v_1, v_2, \dots, v_t\}$, the Kalman Filter computes the *Maximum A-Posteriori* (MAP) estimate of the latent state at time t :

$$\hat{s}_t = \arg \max_s p(s \mid v_1, v_2, \dots, v_t), \quad (11)$$

where p is the probability density function. The MAP estimate is optimal in the sense that it minimizes the expected squared error between the true state and its estimation given all measurements up to time t .

The KF recursion. The Kalman Filter updates the state estimate recursively as new measurements arrive. At time t ,

the update is:

$$\hat{s}_t = \underbrace{A_t \hat{s}_{t-1} + B_t u_t}_{\text{Predicted state}} + \underbrace{G_t (v_{t,i} - k_t^\top [A_t \hat{s}_{t-1} + B_t u_t])}_{\text{Innovation}}, \quad (12)$$

where the *innovation* measures the discrepancy between the actual measurement v_t and the predicted measurement based on the predicted state estimate.

The *Kalman gain* G_t determines how much to trust the new measurement versus the predicted state. It is computed as follows:

$$G_t = \frac{[A_t \Sigma_{t-1} A_t^\top + Q_t] k_t}{k_t^\top [A_t \Sigma_{t-1} A_t^\top + Q_t] k_t + r_t}. \quad (13)$$

The *error covariance* Σ_t quantifies the uncertainty in the state estimate. It represents the covariance of the estimation error ($s_t - \hat{s}_t$) conditioned on all measurements up to time t . The covariance is updated as:

$$\Sigma_t = (I - G_t k_t^\top) (A_t \Sigma_{t-1} A_t^\top + Q_t) \quad (14)$$

Equations (12), (13) and (14) constitute the KF recursion. We initialize with $\hat{s}_0 = 0$ and $\Sigma_0 = \sigma I_n$, where I_n is the $n \times n$ identity matrix and σ represents our prior uncertainty about the state before observing any measurements.

6.3. Gated KalmaNet: A Steady-State Dynamical System for Large-Scale Training

Despite its optimality, the KF recursion in its most general form is inherently sequential; each update depends on the previous state estimate. This sequential dependency prevents the parallelization necessary for efficient large-scale training on modern hardware.

To enable parallelization, we make a key simplifying assumption: the underlying state remains static over time. This reduces the problem from *tracking* a dynamic state to *estimating* a fixed but unknown parameter from sequential noisy measurements. Formally, we assume a steady-state model:

$$\begin{aligned} s_t &= s_{t-1} \\ v_{t,i} &= k_t^\top s_t + \mu_t, \quad \mu_t \sim \mathcal{N}(0, r_t), \end{aligned} \quad (15)$$

where $A_t = I_n$, $B_t = 0$, and $w_t = 0$ (i.e., no state evolution, no control input, and no process noise).

Adapting to evolving context. While the steady-state assumption may initially seem restrictive, contexts naturally evolve as topics change, GKA addresses this through adaptive weighting (Section 4.1). By assigning higher weights to recent measurements, older observations are naturally faded

out over time, allowing the model to track shifting context despite the static formulation.

Under this simplification, the KF recursion reduces to:

$$\begin{aligned} \hat{s}_t &= \hat{s}_{t-1} + G_t (v_{t,i} - k_t^\top \hat{s}_{t-1}). \\ G_t &= \frac{\Sigma_{t-1} k_t}{k_t^\top \Sigma_{t-1} k_t + r_t}. \\ \Sigma_t &= (I - G_t k_t^\top) \Sigma_{t-1}. \end{aligned} \quad (16)$$

Collecting all channels, these equations can be written compactly in matrix form as shown in (KF).³ A key insight of this work is that the KF recursion for the steady-state model admits an efficient parallel implementation via chunked processing (detailed in Section 4) that results in Gated KalmaNet.

Critically, the KF recursion accounts for the entire history when computing state estimates. The Kalman gain G_t at each step depends on all previous measurements through Σ_{t-1} . This contrasts with most existing SSMs, which we show next can be viewed as approximations that ignore historical dependencies when computing their gain matrices. This principled treatment of the full history is a key advantage of our approach.

6.4. Connection with Existing SSM Layers

DeltaNet [65] approximates the KF recursion in (16) by assuming fixed error covariance: $\Sigma_t = I_n$ for all t . This simplifies the Kalman gain to:

$$G_t = \frac{k_t}{k_t^\top k_t + r_t} = \frac{k_t}{1 + r_t}, \quad (17)$$

where the second equality assumes unit-normalized keys, a common assumption in practical instantiations of DeltaNet. Substituting (17) into the state update (16) and defining $\beta_t = (1 + r_t)^{-1}$ yields:

$$\hat{s}_t = (I - \beta_t k_t k_t^\top) \hat{s}_{t-1} + \beta_t k_t v_{t,i}, \quad (\text{DeltaNet})$$

which is the DeltaNet recurrence. By fixing Σ_t , DeltaNet avoids tracking the evolving uncertainty in the state estimate, a key simplification that sacrifices optimality for computational efficiency. In contrast, GKA maintains the full error covariance Σ_t , allowing it to optimally weight measurements based on the entire history.

Gated DeltaNet (GDN) [67] extends DeltaNet by incorporating explicit forgetting through a time-dependent decay factor α_t . Like DeltaNet, GDN can be viewed as fixing $\Sigma_t = I_n$, but applying this approximation to the KF recursion for a fading dynamical system where the state decays over time.

³with columns of S_t transposed to being rows of S_t to be consistent with the notation in (KF) and taking the noise variance $r_t = \frac{1}{\eta_t}$.

Specifically, GDN assumes

$$\begin{aligned} s_t &= \alpha_t s_{t-1} + w_t & w_t &\sim \mathcal{N}(0, I_n) \\ v_{t,i} &= k_t^\top s_t + \mu_t, & \mu_t &\sim \mathcal{N}(0, r_t), \end{aligned} \quad (18)$$

where $\alpha_t \in [0, 1]$ is a learned decay factor controlling how much past information to retain. This corresponds to setting $A_t = \alpha_t I_n$ in (LGM). When $\alpha_t \rightarrow 0$, the state "forgets" the past completely; when $\alpha_t \rightarrow 1$, the state is fully retained.

Under the identity covariance assumption $\Sigma_t = I_n$, the Kalman gain becomes:

$$G_t = \frac{(\alpha_t^2 + 1)k_t}{(\alpha_t^2 + 1)k_t^\top k_t + r_t} = \frac{k_t}{1 + r_t/(\alpha_t^2 + 1)}, \quad (19)$$

where the second equality again assumed unit-normalized keys (as in DeltaNet). Defining $\beta_t = (1 + \frac{r_t}{\alpha_t^2 + 1})^{-1}$ and substituting into the state update (12) yields:

$$\begin{aligned} \hat{s}_t &= \alpha_t \hat{s}_{t-1} + \beta_t k_t (v_{t,i} - k_t^\top [\alpha_t \hat{s}_{t-1}]), \\ &= [I_n - \beta_t k_t k_t^\top] \alpha_t \hat{s}_{t-1} + \beta_t k_t v_{t,i}, \end{aligned} \quad (\text{GDN})$$

which recovers the GDN recurrence. In practice, β_t is an input-dependent learnable parameter.

Like DeltaNet, GDN avoids tracking the evolving uncertainty Σ_t , trading optimality for computational simplicity. The key difference is that GDN's explicit forgetting factor α_t provides additional control over the memory horizon. However, by fixing $\Sigma_t = I_n$, GDN still ignores how measurement history should optimally influence the Kalman gain, leading to suboptimal performance compared to GKA (see Section 5).

Kimi Delta Attention (KDA) [56] further extends GDN by using channel-specific decay factors $\alpha_{t,i}$ in place of the global α_t . This allows different channels to have independent memory horizons. In the KF framework, this corresponds to:

$$s_{t,i} = \alpha_{t,i} s_{t-1,i} + w_{t,i} \quad w_{t,i} \sim \mathcal{N}(0, I_n), \quad (20)$$

for each channel i . While this added flexibility can improve expressiveness, KDA still assumes $\Sigma_t = I_n$ and therefore does not optimally consider the entire past when computing its state update. Like DeltaNet and GDN, KDA sacrifices optimality for computational simplicity.

7. Discussions and Limitations

Thanks to its expressive test-time ridge regression objective, Gated KalmaNet extends previous fading memory layers like Mamba2, LongHorn and Gated DeltaNet, all of which only depend on an instantaneous test-time objective. However, GKA is only optimal among linear memory layers, solving our test-time objective leveraging non-linear updates

while still maintaining hardware efficiency and numerical stability is an interesting area for future research. Despite the efficient kernels we implemented, we believe even faster implementations of our idea are possible, e.g., via *sketching* (see Section H for preliminary results). Finally, while we have showed promising results in combining GKA with Attention layers into Hybrid models (Section I), further scaling beyond 3B parameters models is required to validate GKA on more challenging real world problems.

References

- [1] Simran Arora, Sabri Eyuboglu, Aman Timalsina, Isys Johnson, Michael Poli, James Zou, Atri Rudra, and Christopher Ré. Zoology: Measuring and improving recall in efficient language models. *arXiv preprint arXiv:2312.04927*, 2023. 2, 7, 15, 25
- [2] Sangmin Bae, Bilge Acun, Haroun Habeeb, Seungyeon Kim, Chien-Yu Lin, Liang Luo, Junjie Wang, and Carole-Jean Wu. Hybrid architectures for language models: Systematic analysis and design insights. *arXiv preprint arXiv:2510.04800*, 2025. 15
- [3] Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xlstm: Extended long short-term memory. *Advances in Neural Information Processing Systems*, 37:107547–107603, 2024. 15
- [4] Sneha Chaudhari, Varun Mithal, Gungor Polatkan, and Rohan Ramanath. An attentive survey of attention models. *ACM Transactions on Intelligent Systems and Technology*, 12(5): 1–32, 2021. 2
- [5] Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, et al. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025. 15
- [6] Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, et al. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. *arXiv preprint arXiv:2410.05080*, 2024. 15
- [7] Hao Cui, Zahra Shamsi, Gowoon Cheon, Xuejian Ma, Shutong Li, Maria Tikhonovskaya, Peter Norgaard, Nayantara Mudur, Martyna Plomecka, Paul Raccuglia, et al. Curie: Evaluating llms on multitask scientific long context understanding and reasoning. *arXiv preprint arXiv:2503.13517*, 2025. 15
- [8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pages 2978–2988, 2019. 15
- [9] Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning*, 2024. 1, 2, 4, 6, 15
- [10] Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024. 15
- [11] Soham De, Samuel L Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, et al. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*, 2024. 15
- [12] Xin Dong, Yonggan Fu, Shizhe Diao, Wonmin Byeon, Zijia Chen, Ameya Sunil Mahabaleshwarkar, Shih-Yang Liu, Matthijs Van Keirsbilck, Min-Hung Chen, Yoshi Suhara, et al. Hymba: A hybrid-head architecture for small language models. *arXiv preprint arXiv:2411.13676*, 2024. 15
- [13] Hanpei Fang, Sijie Tao, Nuo Chen, Kai-Xin Chang, and Tet-suya Sakai. Do large language models favor recent content? a study on recency bias in llm-based reranking. *arXiv preprint arXiv:2509.11353*, 2025. 4
- [14] Lizhe Fang, Yifei Wang, Zhaoyang Liu, Chenheng Zhang, Stefanie Jegelka, Jinyang Gao, Bolin Ding, and Yisen Wang. What is wrong with perplexity for long-context language modeling? *arXiv preprint arXiv:2410.23771*, 2024. 8
- [15] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac’h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 2024. 2, 7
- [16] Tianyu Gao, Alexander Wettig, Howard Yen, and Danqi Chen. How to train long-context language models (effectively). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7376–7399, 2025. 8
- [17] Paolo Gloriosio, Quentin Anthony, Yury Tokpanov, James Whittington, Jonathan Pilault, Adam Ibrahim, and Beren Millidge. Zamba: A compact 7b ssm hybrid model. *arXiv preprint arXiv:2405.16712*, 2024. 15
- [18] Alicia Golden, Samuel Hsieh, Fei Sun, Bilge Acun, Basil Hosmer, Yejin Lee, Zachary DeVito, Jeff Johnson, Gu-Yeon Wei, David Brooks, et al. Is flash attention stable? *arXiv preprint arXiv:2405.02803*, 2024. 7
- [19] Gene H Golub and Charles F Van Loan. *Matrix Computations (4th ed.)*. The Johns Hopkins University Press, 2013. 3
- [20] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023. 15
- [21] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *CoRR*, abs/2111.00396, 2021. 15
- [22] Albert Gu, Isys Johnson, Karan Goel, Khaled Saab, Tri Dao, Atri Rudra, and Christopher Ré. Combining recurrent, convolutional, and continuous-time models with linear state space layers. *Advances in neural information processing systems*, 34:572–585, 2021. 15
- [23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. 2
- [24] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. An empirical analysis of compute-optimal large language model training. *Advances in neural information processing systems*, 35:30016–30030, 2022. 26
- [25] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekish, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What’s the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024. 2, 8, 25

- [26] Samy Jelassi, David Brandfonbrener, Sham M Kakade, and Eran Malach. Repeat after me: Transformers are better than state space models at copying. *arXiv preprint arXiv:2402.01032*, 2024. 15
- [27] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023. 15
- [28] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 1960. 1, 15
- [29] Gregory Kamradt. Needle in a haystack - pressure testing llms. https://github.com/gkamradt/LLMTest_NeedleInAHaystack, 2023. 25
- [30] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020. 15
- [31] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023. 15
- [32] Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Yitzhak Gadre, Hritik Bansal, Etash Guha, Sedrick Scott Keh, Kushal Arora, et al. Datacomp-lm: In search of the next generation of training sets for language models. *Advances in Neural Information Processing Systems*, 37:14200–14282, 2024. 7
- [33] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, et al. Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*, 2024. 15
- [34] Bo Liu, Rui Wang, Lemeng Wu, Yihao Feng, Peter Stone, and Qiang Liu. Longhorn: State space models are amortized online learners. In *International Conference on Learning Representations*, 2025. 1, 2
- [35] Stefan Ljung and Lennart Ljung. Error propagation properties of recursive least-squares adaptation algorithms. *Automatica*, 21(2):157–167, 1985. 15
- [36] Mark D McDonnell, Dong Gong, Amin Parvaneh, Ehsan Abbasnejad, and Anton van den Hengel. RanPAC: Random projections and pre-trained models for continual learning. *Advances in Neural Information Processing Systems*, 2023. 3
- [37] Amirkeivan Mohtashami and Martin Jaggi. Landmark attention: Random-access infinite context length for transformers. *arXiv preprint arXiv:2305.16300*, 2023. 15
- [38] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*, 101, 2024. 15
- [39] Elizbar A Nadaraya. On estimating regression. *Theory of Probability & Its Applications*, 9(1):141–142, 1964. 2
- [40] Elvis Nunez, Luca Zancato, Benjamin Bowman, Aditya Gotlatkar, Wei Xia, and Stefano Soatto. Expansion span: Combining fading memory and retrieval in hybrid state space models. *arXiv preprint arXiv:2412.13328*, 2024. 8, 15
- [41] Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. In *International Conference on Machine Learning*, pages 26670–26698. PMLR, 2023. 15
- [42] Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. Marconi: Prefix caching for the era of hybrid llms. *arXiv preprint arXiv:2411.19379*, 2024. 15
- [43] Fabian Pedregosa. Residual polynomials and the Chebyshev method. <http://fa.bianp.net/blog/2020/polyopt/>, 2020. 4
- [44] Bo Peng, Daniel Goldstein, Quentin Gregory Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Teddy Ferdinand, Kranthi Kiran GV, Haowen Hou, Satyapriya Krishna, Ronald McClelland Jr., Niklas Muennighoff, Fares Obeid, Atsushi Saito, Guangyu Song, Haoqin Tu, Ruichong Zhang, Bingchen Zhao, Qihang Zhao, Jian Zhu, and Rui-Jie Zhu. Eagle and finch: RWKV with matrix-valued states and dynamic recurrence. In *Conference on Language Modeling*, 2024. 2
- [45] Bo Peng, Ruichong Zhang, Daniel Goldstein, Eric Alcaide, Xingjian Du, Haowen Hou, Jiaju Lin, Jiaxing Liu, Janna Lu, William Merrill, et al. Rwkv-7" goose" with expressive dynamic state evolution. Technical report, arXiv preprint arXiv:2503.14456, 2025. 2
- [46] Liangzu Peng and René Vidal. Mathematics of continual learning. Technical report, arXiv:2504.17963 [cs.LG], 2025. 3
- [47] Liangzu Peng, Juan Elenter, Joshua Agterberg, Alejandro Ribeiro, and Rene Vidal. TSVD: Bridging theory and practice in continual learning with pre-trained models. In *International Conference on Learning Representations*, 2025. 3
- [48] Fabio Petroni, Aleksandra Piktus, Angela Fan, Patrick Lewis, Majid Yazdani, Nicola De Cao, James Thorne, Yacine Jernite, Vladimir Karpukhin, Jean Maillard, et al. Kilt: a benchmark for knowledge intensive language tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2523–2544, 2021. 25
- [49] A.H. Sayed. *Fundamentals of Adaptive Filtering*. Wiley, 2003. 15
- [50] A.H. Sayed. *Adaptive Filters*. Wiley, 2011. 15
- [51] Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International conference on machine learning*, 2021. 2
- [52] Samir Shah, Francesco Palmieri, and Michael Datum. Optimal filtering algorithms for fast learning in feedforward neural networks. *Neural Networks*, 5(5):779–787, 1992. 3
- [53] Julien Siems, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazi. Deltaproduct: Improving state-tracking in linear rnns via householder products. Technical report, arXiv:2502.10297v6 [cs.LG], 2025. 2
- [54] 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. Technical report, arXiv:2307.08621v4 [cs.CL], 2023. 2

- [55] 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. [15](#)
- [56] Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiayi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, et al. Kimi linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025. [11](#)
- [57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017. [1](#), [2](#), [15](#)
- [58] René Vidal. Attention: Self-expression is all you need, 2022. [2](#)
- [59] Johannes von Oswald, Nino Scherrer, Seijin Kobayashi, Luca Versari, Songlin Yang, Maximilian Schlegel, Kaitlin Maile, Yanick Schimpf, Oliver Sieberling, Alexander Meulemans, et al. Mesanet: Sequence modeling by locally optimal test-time training. Technical report, arXiv:2506.05233 [cs.LG], 2025. [3](#), [23](#), [26](#)
- [60] Roger Waleffe, Wonmin Byeon, Duncan Riach, Brandon Norick, Vijay Korthikanti, Tri Dao, Albert Gu, Ali Hatamizadeh, Sudhakar Singh, Deepak Narayanan, et al. An empirical study of mamba-based language models. *arXiv preprint arXiv:2406.07887*, 2024. [15](#)
- [61] Ke Alexander Wang, Jiaxin Shi, and Emily B Fox. Test-time regression: a unifying framework for designing sequence models with associative memory. Technical report, arXiv:2501.12352v3 [cs.LG], 2025. [3](#)
- [62] Geoffrey S Watson. Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A*, pages 359–372, 1964. [2](#)
- [63] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. [6](#), [29](#)
- [64] 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*, 2024. [1](#), [2](#), [4](#), [6](#), [7](#), [15](#)
- [65] 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*, pages 115491–115522. Curran Associates, Inc., 2024. [10](#), [15](#)
- [66] Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. In *Neural Information Processing Systems*, 2024. [1](#), [2](#), [3](#), [4](#), [6](#)
- [67] Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *International Conference on Learning Representations*, 2025. [2](#), [6](#), [7](#), [8](#), [10](#), [15](#)
- [68] Howard Yen, Tianyu Gao, Minmin Hou, Ke Ding, Daniel Fleischer, Peter Izsak, Moshe Wasserblat, and Danqi Chen. HELMET: How to evaluate long-context language models effectively and thoroughly. In *International Conference on Learning Representations (ICLR)*, 2025. [2](#), [8](#), [25](#)
- [69] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23078–23097, 2025. [15](#)
- [70] Luca Zancato, Alessandro Achille, Giovanni Paolini, Alessandro Chiuso, and Stefano Soatto. Stacked residuals of dynamic layers for time series anomaly detection. *arXiv preprint arXiv:2202.12457*, 2022. [15](#)
- [71] Luca Zancato, Arjun Seshadri, Yonatan Dukler, Aditya Go-latkar, Yantao Shen, Benjamin Bowman, Matthew Trager, Alessandro Achille, and Stefano Soatto. B'mojo: Hybrid state space realizations of foundation models with eidetic and fading memory. In *Advances in Neural Information Processing Systems*, pages 130433–130462. Curran Associates, Inc., 2024. [1](#), [7](#), [9](#), [15](#)
- [72] Guanxiong Zeng, Yang Chen, Bo Cui, and Shan Yu. Continual learning of context-dependent processing in neural networks. *Nature Machine Intelligence*, 1(8):364–372, 2019. [3](#)

A. Related Work

Since the introduction of Self-Attention [57], significant research has been conducted to reduce its quadratic cost in processing long input sequences. As models and systems scale to million-token contexts, Attention’s bottlenecks have become critical blockers to frontier agentic applications in coding, information gathering, and scientific discovery [6, 7, 27]. Prior works have proposed various approximation schemes to overcome these limitations. For example, Reformer [30] uses locality-sensitive hashing to group tokens with similar embeddings. This enables the model to attend only to a subset of tokens rather than the entire sequence. Other works equip Transformer models with "compressed" memory tokens that are updated dynamically and causally over sliding windows on entire sequence chunks [8, 37, 38]. While a lot of prior work have focused on reducing the quadratic complexity of Attention with sparse approximations [40, 69], this work focuses on linear approximations of Attention.

A.1. Linear Attention

Linear Attention methods approximate the Attention mechanism with constant-size recurrent dynamical systems [3, 9, 64, 67]. Numerous State-Space Model (SSM) variations have been proposed, ranging from those closely resembling Linear Attention [55] or Linear Time-Invariant dynamical systems [22, 70], to those introducing novel adaptive or gated state updates [9, 41, 64].

Despite their differences, all SSMs follow the same basic working principle inspired by classical state-space models [28]: they process the input sequence by maintaining a *fixed-size* state that acts as a compressed (lossy) representation of all processed tokens. Moreover, when implemented in hardware, the state must have finite precision and “fades away the past” as more samples are processed. Successful SSM layers typically employ hardware-aware implementations that efficiently utilize modern matrix multiplication accelerators through highly parallelizable and scalable primitives, including associative scans [11, 20], chunking mechanisms [9, 64], and techniques that avoid materializing the entire state in slow high-bandwidth memory [20].

From a modeling perspective, most Linear Attention implementations introduce data-dependent gating factors to control the speed of their “fading” memory, balancing expressivity with scalability. For example, the transition from Mamba to Mamba2 replaced channel-wise data-dependent gating with head-wise gating for better scalability and Tensor Cores utilization. Input-dependent Gating has been shown to empirically improve training stability [1, 67] and has driven the development of Linear Attention models (e.g., from S4 [21] to Mamba [20] and from DeltaNet [65] to Gated DeltaNet [67]). In our work, we demonstrate that gating emerges naturally as a consequence of solving a weighted least squares objective function, establishing a connection to the favorable numerical properties classically described in the adaptive filtering literature [35, 49, 50].

A.2. Hybrid State Space Attention Models

While extending the recurrent state in SSM layers has yielded performant models, they typically underperform on tasks requiring recall of information from the distant past [26, 60]. Hybrid State-Space Models address this limitation by complementing SSMs’ “fading” state with Attention layers [10, 11, 17, 33]. Early architectures simply stacked SSMs and Attention layers with different blending ratios [9, 20, 60] or replaced full Attention layers with Sliding Window Attention [11]. More sophisticated designs have recently emerged [17, 71].

Notably, B’MOJO [71] complements SSMs’ fading state with "eidetic" memory by combining SSMs with Sliding Window Attention (SWA) in a single layer. Within the window, tokens can attend to a selected set of past tokens that were deemed difficult to predict using an asynchronous causal selection mechanism. B’MOJO was the first hybrid model to propose a parallel fusion of SSM and SWA at the layer level. Subsequent works [2, 12] have shown this parallel fusion approach to be more performant (at equivalent compute) than the stacked approach of earlier works.

Thanks to their lower memory footprint and test-time scalability over long sequences, Hybrid architectures are expanding into long-range agentic tasks and have recently been trained with Reinforcement Learning at scale [5]. When coupled with system-level optimizations like prefix caching [42] and specialized inference engines [31], Hybrid models can increase the number of rollouts (exploration), thereby improving end-to-end performance in Reinforcement Learning loops.

B. Forward and Backward Passes of Chebyshev Iteration (Details)

In Section 4.2 we described our chunk-wise implementation of the CH method with adaptive regularization and gating. We now give full details omitted there.

B.1. Forward Pass

CH in Detail. We begin with describing the CH method (Algorithm 1) in more detail. Assume we have a linear system of equations $H\xi = q$ where H is a $D \times D$ positive definite matrix. We assume H has its all eigenvalues lie in the interval $[\mu, L]$ and the values of μ and L is known. Note that solving this system is equivalent to solving the following quadratic problem:

$$\min_{\xi \in \mathbb{R}^D} \frac{1}{2} \xi^\top H \xi - \xi^\top q. \quad (21)$$

The classic Chebyshev Iteration in its standard form is presented in Algorithm 1. In the initialization phase, we set $\rho = \frac{L-\mu}{L+\mu}$, which is the typical convergence rate of gradient descent applied to the above quadratic problem with stepsize $\frac{2}{L+\mu}$; vaguely speaking, in this setting, this stepsize choice is optimal (e.g., that allows gradient descent to converge the fastest possible). Algorithm 1 initializes two points, ξ_{-1} and ξ_0 . Here ξ_{-1} is zero, and ξ_0 is a gradient step for (21) starting at ξ_{-1} and with stepsize $\frac{2}{L+\mu}$. The final component in initialization is the weight $\omega_0 = 2$. This is the starting point for the weight schedule recursion of ω_i in (weight schedule). Similarly, the initialization of ξ_{-1}, ξ_0 is where we start to compute ξ_i , whose update consists of (grad descent) and (momentum). Note that (grad descent) is with stepsize $2 \cdot \omega_i / (L + \mu)$. Since $\omega_i > 1$, this stepsize is strictly larger than $2 / (L + \mu)$, the latter being the optimal stepsize for vanilla gradient descent. Such a large stepsize alone might not guarantee convergence, but it is balanced by the (momentum) term $\xi_{i-1} - \xi_{i-2}$ with positive weight $\omega_i - 1$ so that the convergence of the Chebyshev iterative method is ensured.

Numerical Stability Considerations. Now we analyze the numerical properties of the Chebyshev Iteration. The major computation consists of matrix-vector multiplication; in a batched parallel implementation, this turns out to be matrix-matrix multiplication. For this, the numerical accuracy is well controlled (e.g., in Triton we could specify the accuracy in `tl.dot`). The update of ω_i in (weight schedule) might raise numerical concerns as it involves division. That said, we show this division operates in a numerically well-behaved range as ω_i is decreasing with i yet lower bounded by 1:

Lemma 3. *For any r , we have $2 = \omega_0 \geq \dots \geq \omega_r \geq \omega_1^* > 1$, where ω_1^* is defined as*

$$\omega_1^* := \frac{2(1 - \sqrt{1 - \rho^2})}{\rho^2}.$$

As a consequence, we have $4 - \rho^2 \omega_i \in [2, 4]$ for all $i = 0, \dots, r$.

Proof. If $L = \mu$, then H is a scaled identity matrix, and the algorithm is simplified a lot. So we assume $L > \mu$ in what follows. With $L > \mu > 0$ we have $\rho \in (0, 1)$. Since $\omega_0 = 2$, we have $4 - \rho^2 \omega_0 \geq 2$ and therefore $0 < \omega_1 \leq 2$. Repeating this argument and we see $\omega_i \in (0, 2]$ for all i . By the definition of ω_i , to show $\omega_i \leq \omega_{i-1}$ is to show

$$\frac{4}{4 - \rho^2 \omega_{i-1}} \leq \omega_{i-1} \Leftrightarrow g(\omega_i) \leq 0$$

where g is defined as $g(\omega) = \rho^2 \omega^2 - 4\omega + 4$. Note that $g(\omega)$ has two roots, ω_1^* , as defined earlier, and $\omega_2^* = \frac{2(1 + \sqrt{1 - \rho^2})}{\rho^2}$; ω_1^*, ω_2^* are the two fixed points of the update (weight schedule). Observing that $\omega_0 = 2$ lies in the interval (ω_1^*, ω_2^*) , and moreover, for any $i \geq 1$, if $\omega_{i-1} > \omega_1^*$ we must have

$$\omega_i = \frac{4}{4 - \rho^2 \omega_{i-1}} > \frac{4}{4 - \rho^2 \omega_1^*} = \omega_1^*.$$

This proves $\omega_i > \omega_1^*$ for all $i = 1, \dots, r$. Next, since $\omega_0 = 2$ lies in the interval (ω_1^*, ω_2^*) where $g(\omega)$ decreases, therefore we have $\omega_1 \leq \omega_0$. Thus ω_1 lies in (ω_1^*, ω_2^*) again. We could then conclude inductively that $\omega_1^* < \omega_i \leq \omega_{i-1}$ for all $i = 1, \dots, r$. \square

From Lemma 3 we know that the update of ω_i in (weight schedule) would not create much numerical concern in a forward pass, as we have $\omega_i \in [1, 2]$ for all i . Furthermore, we can bound the rate at which ω_i converges to ω_1^* :

Lemma 4. Define $\kappa := \frac{L}{\mu}$. For any $i = 1, \dots, r$, we have

$$(\omega_i - \omega_1^*) \leq R^i \cdot (\omega_0 - \omega_1^*),$$

where R is defined as

$$R := \frac{\kappa - 1}{\kappa + 1} \cdot \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}.$$

Proof. From the update rule of ω_i in (weight schedule) and the fixed point property of ω_1^* , we have

$$\begin{aligned} \omega_i - \omega_1^* &= \frac{4}{4 - \rho^2 \omega_{i-1}} - \frac{4}{4 - \rho^2 \omega_1^*} \\ &= \frac{4\rho^2}{(4 - \rho^2 \omega_{i-1})(4 - \rho^2 \omega_1^*)} \cdot (\omega_{i-1} - \omega_1^*) \\ &\stackrel{(i)}{=} \frac{\rho^2 \omega_1^*}{4 - \rho^2 \omega_{i-1}} \cdot (\omega_{i-1} - \omega_1^*) \\ &\stackrel{(ii)}{\leq} \frac{\rho^2 \omega_{i-1} \omega_1^*}{4} \cdot (\omega_{i-1} - \omega_1^*) \\ &\stackrel{(iii)}{\leq} \left(1 - \sqrt{1 - \rho^2}\right) \cdot (\omega_{i-1} - \omega_1^*) \\ &\stackrel{(iv)}{=} \left(\frac{\kappa - 1}{\kappa + 1} \cdot \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right) \cdot (\omega_{i-1} - \omega_1^*) \end{aligned}$$

Here, (i) follows from the fact that ω_1^* is a fixed point, (ii) follows from Lemma 3 that $\omega_i \leq \omega_{i-1}$, (iii) follows from the definition of ω_1^* and the fact $\omega_{i-1} \leq 2$, and (iv) follows from the definitions of κ and ρ . The proof is concluded by unrolling the above recurrence. \square

Remark 1. Here, we call R the linear convergence rate (or *contraction factor*) of ω_i to ω_1^* . First-order methods for solving $H\xi = q$ converge at most at a rate $R_a := \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$, and we see ω_i converges at an even faster rate. Numerically, assuming $\kappa = \frac{L}{\mu} = \frac{1.02}{0.02} = 51$, we then have:

$$\begin{aligned} R &\approx 0.7253, \quad R^5 \approx 0.2, \quad R^{10} \approx 0.04, \quad R^{20} \approx 0.0016, \quad R^{30} \approx 6 \times 10^{-5} \\ R_a &\approx 0.7543, \quad R_a^5 \approx 0.244, \quad R_a^{10} \approx 0.0597, \quad R_a^{20} \approx 0.0036, \quad R_a^{30} \approx 0.0002. \end{aligned}$$

Thus, with $\kappa = 51$, the update of ω_i in (weight schedule) converges in at most 20 iterations up to the bfloat16 precision.

B.2. Backward Pass

We now give details for backpropagation through the Chebyshev Iteration (Algorithm 1) via implicit differentiation.

Computing $\frac{dL}{dq_t}$ and $\frac{dL}{d\lambda_t}$. First, we follow Table 1 and Lemma 2, and compute dq_t . Then, given the equation $(H_t + \lambda_t I)dq_t = dx_t^*$, we have that

$$d(H_t + \lambda_t I) = -dq_t(x_t^*)^\top. \quad (22)$$

Therefore $d\lambda_t = \text{tr}(d(H_t + \lambda_t I)) = -(x_t^*)^\top dq_t$. Since we set $\lambda_t = a \cdot \|H_t\|_F$, this indicates

$$dH_t = -dq_t(x_t^*)^\top - a \cdot \frac{H_t}{\|H_t\|_F} \cdot ((x_t^*)^\top dq_t). \quad (23)$$

Note that this expression of dH_t is *partial*: It accounts for the upstream gradient from dq_t only and one might think of the subsequent states all depend on H_t . We will accumulate the gradients later when needed.

Now, the recursion of H_t in (CH) implies

$$dk_i = \sum_{t \geq i} (dH_t + (dH_t)^\top) k_i \cdot \frac{\zeta_t}{\zeta_i} \quad (24)$$

$$= \sum_{t \geq i} \frac{\zeta_t}{\zeta_i} (-dq_t \otimes (x_t^*)^\top - x_t^* \otimes (dq_t)^\top + w_t H_t) k_i, \quad (25)$$

which proves Lemma 1. We refer the reader to Section B.2.1 for more detailed derivations of dq_t and dk_t .

Derivatives for Gating. In practice we often parameterize γ_t in the log space to ensure numerical stability. Thus, let us first revise our notations for this case. Let $g_t = \log \gamma_t$ and $G_t := \sum_{i=1}^t g_i = \log \left(\prod_{i=1}^t \gamma_i \right)$. Then the mask matrix M is

$$M_{i,j} = \begin{cases} \exp(G_j - G_i) & j \geq i; \\ 0 & \text{otherwise.} \end{cases} \quad (26)$$

Now, since for any $c = 1, \dots, C$ we have

$$H_c = \exp(G_c) \cdot H_0 + \sum_{j=1}^c \exp(G_c - G_j) \cdot k_j k_j^\top, \quad (27)$$

for any G_i we have the following basic derivatives:

$$\frac{dH_c}{dG_i} = \begin{cases} 0 & c < i; \\ \exp(G_i) \cdot H_0 + \sum_{j=1}^{i-1} \exp(G_i - G_j) \cdot k_j k_j^\top & c = i; \\ -\exp(G_c - G_i) k_i k_i^\top & c > i, \end{cases} \quad i = 1, \dots, C, \quad c = 1, \dots, C; \quad (28)$$

$$\frac{dH_{C+1}}{dG_i} = -\exp(G_{C+1} - G_i) k_i k_i^\top \quad i = 1, \dots, C \quad (29)$$

With dH_{C+1} being the aggregated gradient from the future, we have for $i = 1, \dots, C$ that

$$dG_i = \sum_{c=i}^{C+1} \langle dH_c, \frac{dH_c}{dG_i} \rangle \quad (30)$$

$$= e^{G_i} \langle dH_i, H_0 \rangle + \sum_{j=1}^{i-1} e^{G_i - G_j} \langle dH_i, k_j k_j^\top \rangle - \sum_{c=i+1}^C e^{G_c - G_i} \langle dH_c, k_i k_i^\top \rangle - e^{G_{C+1} - G_i} \langle dH_{C+1}, k_i k_i^\top \rangle \quad (31)$$

$$= e^{G_i} \langle dH_i, H_0 \rangle + \sum_{j=1}^i e^{G_i - G_j} \langle dH_i, k_j k_j^\top \rangle - \sum_{c=i}^C e^{G_c - G_i} \langle dH_c, k_i k_i^\top \rangle - e^{G_{C+1} - G_i} \langle dH_{C+1}, k_i k_i^\top \rangle \quad (32)$$

$$dG_{C+1} = e^{G_{C+1}} \langle dH_{C+1}, H_0 \rangle + \sum_{j=1}^C e^{G_{C+1} - G_j} \langle dH_{C+1}, k_j k_j^\top \rangle \quad (33)$$

Note that in one of the above equations we add and subtract the term $\langle dH_i, k_i k_i^\top \rangle$, which will simplify the implementation.

Recall that $dH_t = -dq_t(x_t^*)^\top - \frac{1}{2} \cdot w_t H_t$ with $w_t = \frac{2a \cdot (x_t^*)^\top dq_t}{\|H_t\|_F}$. In computing the derivatives of G_i the first term $dq_t(x_t^*)^\top$ is the standard term that arises in that of (Linear-SSM), which we omit here. We now focus on the second term $\frac{1}{2} \cdot w_t H_t$. This implies the gradients dG_i and dG_{C+1} are partly given respectively by (using the notations in Section 4.2.2 and omitting some algebraic operations)

$$\frac{1}{2} \cdot \langle w_i H_i, H_i \rangle - \frac{1}{2} \cdot k_i^\top B_i k_i \quad \text{and} \quad \frac{1}{2} \cdot e^{G_C} \langle \tilde{B}_{C+1}, H_0 \rangle + \frac{1}{2} \cdot \sum_{j=1}^C e^{G_C - G_j} \cdot k_j^\top \tilde{B}_{C+1} k_j. \quad (34)$$

Computing the first term $\langle w_i H_i, H_i \rangle$ in parallel is easy by invoking the definition of w_i and the Frobenius norm of H_i we stored during the forward pass. Computing the quadratic terms $k_i^\top B_i k_i$ and $k_j^\top \tilde{B}_{C+1} k_j$ in parallel is easy and follows from our computation of $B_i k_i$ and $\tilde{B}_{C+1} k_i$ for dk_i in Section 4.2.2. Computing $\langle \tilde{B}_{C+1}, H_0 \rangle$ is easy since we recompute the initial states H_0 of each chunk and have them available during the backward pass, while \tilde{B}_{C+1} is updated backwards in a for loop.

B.2.1. Computing $\frac{dL}{dq_t}$ and $\frac{dL}{dk_t}$.

In forward pass we solve

$$(H_t + \lambda_t I) x_t = q_t$$

$$\begin{aligned}
x_t &= (H_t + \lambda_t I)^{-1} q_t \\
\implies dx_t &= \underbrace{(H_t + \lambda_t I)^{-1} dq_t}_{J_{q_t \rightarrow x_t}}
\end{aligned} \tag{35}$$

Recall that the gradient is transpose of the Jacobian, thus we obtain

$$\frac{dL}{dq_t} = (H_t + \lambda_t I)^{-1} \frac{dL}{dx_t}. \tag{36}$$

Thus, we can obtain $\frac{dL}{dq_t}$ by running a Chebyshev iteration to solve (for z) the linear system of equations

$$(H_t + \lambda_t I)z = \frac{dL}{dx_t}.$$

Now we have

$$\begin{aligned}
dx_t &= d(H_t + \lambda_t I)^{-1} q_t \\
dx_t &= -(H_t + \lambda_t I)^{-1} d(H_t + \lambda_t I) (H_t + \lambda_t I)^{-1} q_t \\
dx_t &= -(H_t + \lambda_t I)^{-1} d(H_t + \lambda_t I) x_t \\
&= (x_t^\top \otimes -(H_t + \lambda_t I)^{-1}) \text{vec}(d(H_t + \lambda_t I))
\end{aligned} \tag{37}$$

In the last equality we have used the identity $\text{vec}(ABC) = (C^\top \otimes A) \text{vec}(B)$.

Now we will compute the Jacobian of λ with respect to H_t :

$$\begin{aligned}
\lambda_t &= a \|H_t\|_F \\
&= a \sqrt{\text{Tr}(H_t^\top H_t)} \\
\implies d\lambda &= ad \left(\sqrt{\text{Tr}(H_t^\top H_t)} \right) \\
&= a \frac{1}{2 \|H_t\|_F} \text{Tr}((dH_t)^\top H_t + H_t^\top (dH_t)) \\
&= a \frac{1}{\|H_t\|_F} \text{vec}(H_t)^\top d\text{vec}(H_t)
\end{aligned} \tag{38}$$

Substituting (38) in (37).

$$dx_t = (x_t^\top \otimes -(H_t + \lambda_t I)^{-1}) \left(\text{vec}(dH_t) + \frac{a}{\|H_t\|_F} \text{vec}(I) \text{vec}(H_t)^\top \text{vec}(dH_t) \right) \tag{39}$$

Thus, we can obtain $\text{vec}(\frac{dL}{dH_t})$ as,

$$\text{vec}(\frac{dL}{dH_t}) = (x_t \otimes -(H_t + \lambda_t I)^{-1}) \frac{dL}{dx_t} + \frac{a}{\|H_t\|_F} \text{vec}(H_t) \text{vec}(I)^\top (x_t \otimes -(H_t + \lambda_t I)^{-1}) \frac{dL}{dx_t} \tag{40}$$

Substituting from (36),

$$\begin{aligned}
\text{vec}(\frac{dL}{dH_t}) &= -(x_t \otimes \frac{dL}{dq_t}) - \frac{a}{\|H_t\|_F} \text{vec}(H_t) \text{vec}(I)^\top (x_t \otimes \frac{dL}{dq_t}) \\
&= -(x_t \otimes \frac{dL}{dq_t}) - \frac{a}{\|H_t\|_F} \text{vec}(H_t) \langle \frac{dL}{dq_t}, x_t \rangle
\end{aligned} \tag{41}$$

Now, with gating, we have $H_t = \gamma_t H_{t-1} + k_t k_t^\top$. Which can be unrolled as

$$H_l = \sum_{i=0}^l \left(\prod_{k=i}^{l-1} \gamma_k \right) k_i k_i^\top \tag{42}$$

We will compute $\frac{dL}{dk_l}$ for some $l \leq t$,

$$\frac{dL}{dk_l} = \sum_{t \geq l} \frac{d\text{vec}(H_t)}{dk_l} \text{vec}(\frac{dL}{dH_t}) \tag{43}$$

Computing $\frac{d\text{vec}(H_t)}{dk_l}$ for some $t \geq l$

$$H_t = \prod_{i=l}^{t-1} \gamma_i k_l k_l^\top + \text{terms indep. of } k_l. \quad (44)$$

Taking differentials on both sides,

$$\begin{aligned} dH_t &= \prod_{i=l}^{t-1} \gamma_i \left[dk_l k_l^\top + k_l (dk_l)^\top \right] \\ d(\text{vec}(H_t)) &= \prod_{i=l}^{t-1} \gamma_i \left[\text{vec}(dk_l k_l^\top) + \text{vec}(k_l (dk_l)^\top) \right] \\ &= \underbrace{\left(\prod_{i=l}^{t-1} \gamma_i \right) \left[(k_l \otimes I) + (I \otimes k_l) \right]}_{J_{k_l \rightarrow \text{vec}(H)}} dk_l \end{aligned} \quad (45)$$

where in the last equality we used the identity $\text{vec}(k_l dk_l^\top) = dk_l \otimes k_l = (I \otimes k_l) dk_l$.

Substituting the Jacobian (transposed for gradients) from (45) to (43) we obtain.

$$\frac{dL}{dk_l} = \sum_{t \geq l} \left(\prod_{i=l}^{t-1} \gamma_i \right) \left[(k_l^\top \otimes I) + (I \otimes k_l^\top) \right] \text{vec} \left(\frac{dL}{dH_t} \right) \quad (46)$$

Substituting the expression for $\text{vec}(\frac{dL}{dH_t})$ into equation (46) we get:

$$\begin{aligned} \frac{dL}{dk_l} &= - \sum_{t \geq l} \left(\prod_{i=l}^{t-1} \gamma_i \right) \left[(k_l^\top \otimes I) + (I \otimes k_l^\top) \right] \left[(x_t \otimes \frac{dL}{dq_t}) + \frac{a}{\|H_t\|_F} \text{vec}(H_t) \langle \frac{dL}{dq_t}, x_t \rangle \right] = \\ &= - \sum_{t \geq l} \left(\prod_{i=l}^{t-1} \gamma_i \right) \left[\left((k_l^\top \otimes I)(x_t \otimes \frac{dL}{dq_t}) + \frac{a}{\|H_t\|_F} (k_l^\top \otimes I) \text{vec}(H_t) \langle \frac{dL}{dq_t}, x_t \rangle \right) + \right. \\ &\quad \left. (I \otimes k_l^\top)(x_t \otimes \frac{dL}{dq_t}) + \frac{a}{\|H_t\|_F} (I \otimes k_l^\top) \text{vec}(H_t) \langle \frac{dL}{dq_t}, x_t \rangle \right] \end{aligned} \quad (47)$$

Note that the following equations hold:

$$\begin{aligned} (k_l^\top \otimes I)(x_t \otimes \frac{dL}{dq_t}) &= (k_l^\top x_t \otimes \frac{dL}{dq_t}) = \langle k_l, x_t \rangle \frac{dL}{dq_t} \\ (I \otimes k_l^\top)(x_t \otimes \frac{dL}{dq_t}) &= x_t \otimes k_l^\top \frac{dL}{dq_t} = \langle k_l, \frac{dL}{dq_t} \rangle x_t \end{aligned} \quad (48)$$

since $(A \otimes B)(C \otimes D) = AC \otimes BD$ and the fact that the Kronecker products after the simplification is a scalar times a vector.

For the other terms it holds:

$$\begin{aligned} (k_l^\top \otimes I) \text{vec}(H_t) &= \text{vec}(H_t k_l) \\ (I \otimes k_l^\top) \text{vec}(H_t) &= \text{vec}(k_l^\top H_t) = \text{vec}(H_t^\top k_l) \end{aligned} \quad (49)$$

where we used the fact $\text{vec}(AXB) = (B^\top \otimes A) \text{vec}(X)$ and the fact that the vec operator applied to a row vector returns the same result as applying it on its transpose (so we go from $k_l^\top H_t$ to $H_t^\top k_l$). Since H_t is symmetric we can sum both contributions and get twice that amount.

Eventually we get:

$$\boxed{\frac{dL}{dk_l} = - \sum_{t \geq l} \left(\prod_{i=l}^{t-1} \gamma_i \right) \left[\langle k_l, x_t \rangle \frac{dL}{dq_t} + \langle k_l, \frac{dL}{dq_t} \rangle x_t + \frac{2a}{\|H_t\|_F} \langle x_t, \frac{dL}{dq_t} \rangle H_t k_l \right]} \quad (50)$$

Or equivalently, collecting the terms that are linear in the gradient:

$$\frac{dL}{dk_l} = - \sum_{t \geq l} \left(\prod_{i=l}^{t-1} \gamma_i \right) \left[\langle k_l, x_t \rangle \frac{dL}{dq_t} + \langle k_l, \frac{dL}{dq_t} \rangle x_t + \frac{2a \langle x_t, \frac{dL}{dq_t} \rangle}{\|H_t\|_F} H_t k_l \right] \quad (51)$$

Note: The last term creates a dependence on k_l through $H_t k_l$, which is expected since the regularization λ_t couples the gradient computation.

C. Proof of Lemma 2

Lemma 2 describes an interesting phenomenon where, for the CH method (Algorithm 1), the gradient $d\hat{q}$ obtained from implicit differentiation coincides with the exact gradient dq obtained via backpropagation (chain rule). To prove this result, one way is to derive an analytic expression for dq (Section C.1) and then inspect the recursions. However, this can be algebraically involved. Here, we present a clear proof based on some simple observations.

First, note that the output ξ_r is linear in q and moreover there is a matrix function $p_r(H) \in \mathbb{R}^{D \times D}$ such that

$$\xi_r = p_r(H) \cdot q. \quad (52)$$

Here $p_r(H)$ is a polynomial function of H that encodes the Chebyshev iteration (Algorithm 1). Conversely, we understand that $p_r(H) \cdot q$ can be computed by applying the Chebyshev iteration with H, q for r iterations (together with other parameters such as μ, L). Then, given the output gradient $d\xi_r$, we have

$$dq = p_r(H)^\top \cdot d\xi_r = p_r(H) \cdot d\xi_r, \quad (53)$$

where the last equality follows, since H is symmetric, which implies $p_r(H)$ is symmetric. The proof is finished by observing that $p_r(H) \cdot d\xi_r$ can be computed via Algorithm 1 with $H = H, q = d\xi_r$ and other parameters, which gives us dq .

C.1. The Exact Backward Pass for dq and dH

Here we show how to obtain the exact gradients of dH and dq in Algorithm 1 given the output gradient $d\xi_r$, which might be of independent interests. The key insight here is that the Chebyshev iteration can be *reversed*.

Backward Pass for dq . Let I be the identity matrix of suitable size. To derive a backward pass of Algorithm 1, we first write down the update of ξ_i concisely in the following recursion

$$\xi_i = A_i \xi_{i-1} + b_i \xi_{i-2} + c_i q, \quad (54)$$

where A_i, b_i, c_i are defined as

$$A_i = \omega_i I - \frac{2 \cdot \omega_i}{L + \mu} H, \quad b_i = -(\omega_i - 1), \quad c_i = \frac{2 \cdot \omega_i}{L + \mu}. \quad (55)$$

Note that A_i is symmetric. Define $d\xi_i := \frac{d\mathcal{L}}{d\xi_i}$ for every i . With some loss function \mathcal{L} , assume we are now given $d\xi_r$, and our goal is to compute $dq := \frac{d\mathcal{L}}{dq}$. Since q appears in (54) for every i , we know $\xi_0, \xi_1, \dots, \xi_r$ all depend on q . Therefore, with $c_0 := \frac{2}{L + \mu}$, we have

$$dq = \sum_{i=0}^r c_i \cdot d\xi_i.$$

It remains to compute $d\xi_i$ for every i . Applying the chain rule to (54), we obtain

$$\begin{aligned} d\xi_{r-1} &= A_r \cdot d\xi_r \\ d\xi_{i-2} &= A_{i-1} \cdot d\xi_{i-1} + b_i \cdot d\xi_i, \quad \forall i = r, \dots, 2. \end{aligned} \quad (56)$$

Note that A_i, b_i, c_i depend on some constant terms and ω_i . Thus, to compute them backward we assume access to ω_r and these constants. By reversing (weight schedule) we derive the following recursion:

$$\begin{aligned} \nu_r &\leftarrow \omega_r \\ \nu_{i-1} &\leftarrow \frac{4}{\rho^2} \left(1 - \frac{1}{\nu_i} \right), \quad \forall i = r, \dots, 1. \end{aligned} \quad (57)$$

Similarly to how ω_i decreases with i and converges to ω_1^* , we may prove ν_i is convergent to the other fixed point, ω_2^* , as i decreases (and the iterate does not stop at $i = 1$).

Backward Pass for dA . From (54) and (55) we see that

$$dA_i = d\xi_i \otimes \xi_{i-1}^\top, \quad dH = -\frac{2 \cdot \omega_i}{L + \mu} \cdot dA_i \quad (58)$$

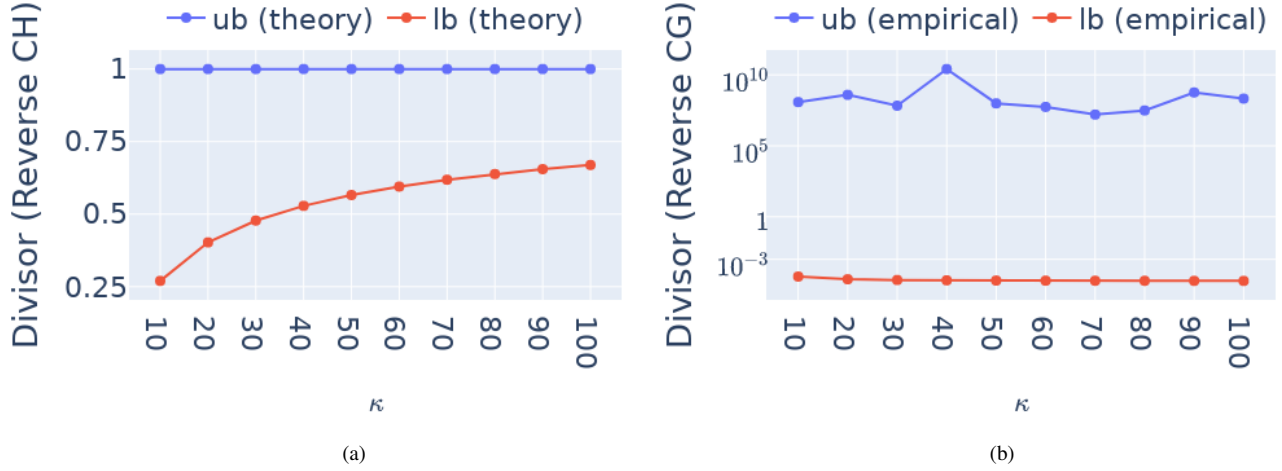


Figure 5. (a) The theoretical lower and upper bounds for the values of the divisor b_i that arise in reversing Chebyshev (59); (b) The empirical lower and upper bounds for the divisor that arises in reversing CG.

where \otimes denotes the Kronecker product; this is the out product of $d\xi_i$ and ξ_{i-1}^\top , as $d\xi_i$ and ξ_{i-1} are vectors.

Reverse Chebyshev Iteration. At first glance, computing $dA_i = d\xi_i \otimes \xi_{i-1}^\top$ requires storing ξ_{i-1} in the forward pass, and the actual calculation of dA_i is done after we run the backward pass for $d\xi_i$ in (56). However, storing all ξ_i 's would be memory-inefficient. To address this issue, a main insight here is that we can reverse (54) and write

$$\xi_{i-2} = \frac{1}{b_i}(\xi_i - A_i \xi_{i-1} + c_i q). \quad (59)$$

This implies that we can recover all the iterates ξ_r, \dots, ξ_0 as soon as we have access to the last two, ξ_r, ξ_{r-1} . Therefore, to obtain dA_i , we can run two iteration schemes in (56) and (59) simultaneously.

Remark 2. We find that being able to run the iterative update *backward* in a numerically stable fashion is a main feature of the Chebyshev iterative method (or more generally, gradient descent variants with momentum). Vanilla gradient descent can not efficiently reverse its iterate $\xi_i = \xi_{i-1} - \gamma_i(H\xi_{i-1} - q)$ with stepsize γ_i , as it requires inverting $(I - \gamma_i H)$. Moreover, reversing (59) can be done stably, as b_i is often in a good numerical range, which means division by b_i in (59) is not an issue. To see this, first note that by Lemma 3 we have

$$1 \geq -b_i = \omega_i - 1 \geq \omega_1^* - 1.$$

Note that ω_1^* defined in Lemma 3 is an increasing function of ρ and therefore of κ . We then have that $-b_i \in [0.25, 1]$ for any $\kappa \geq 10$ (we will not consider the case $\kappa < 10$ as this means we need to add a very large regularization strength which might harm the minimization of the regression loss). In comparison, if we were to reverse the CG iteration, we would need to divide a quantity that is often numerically as small as 10^{-3} or as large as 10^{10} (see Fig. 5). This is why it is numerically unstable to reverse CG.

D. Experimental Setup

Model Configurations. We consider models of 3 different sizes: 440M, 1B, and 2.8B. This is summarized in Table 3. All models are with the GPT2 tokenizer similarly to [59].

Training Configurations. All models are trained with the AdamW optimizer with initial learning rate 10^{-3} , 5% warm-up steps, cosine schedule, gradient clipping with maximum norm 1.

Models of the same scale use the same training configurations. Specifically (see also Table 4):

- For 440M models, we use sequence length 2048 and 8B DCLM tokens.
- For 1B models, we use sequence length 2048 and 20B DCLM tokens.
- For 2.8B models, we use sequence length 4096 and 100B DCLM tokens.

Algorithm 2: Backward Pass of Chebyshev Iteration

- 1 Input: $H, d\xi_r, L, \mu, ,$ number of iterations r , the final weight ω_r ;
- 2 Initialize $\rho \leftarrow \frac{L-\mu}{L+\mu}, d\xi_{r+1} \leftarrow 0, \nu_r \leftarrow \omega_r, \nu_{r+1} \leftarrow 0, \nu_0 \leftarrow 1, dq \leftarrow \frac{2\nu_r}{L+\mu} \cdot d\xi_r, dH \leftarrow -\frac{2\nu_r}{L+\mu} d\xi_r \otimes \xi_{r-1}^\top$;
- 3 For $i = r, \dots, 1$:

$$\xi_{i-2} \leftarrow -\frac{1}{\nu_i - 1} \left(\xi_i - \left(\nu_i I - \frac{2 \cdot \nu_i}{L + \mu} H \right) \xi_{i-1} + \frac{2 \cdot \nu_i}{L + \mu} q \right) \quad (60)$$

$$d\xi_{i-1} \leftarrow \left(\nu_i \cdot d\xi_i - \frac{2 \cdot \nu_i}{L + \mu} \cdot H \cdot d\xi_i \right) - (\nu_{i+1} - 1) \cdot d\xi_{i+1} \quad (61)$$

$$\nu_{i-1} \leftarrow \frac{4}{\rho^2} \left(1 - \frac{1}{\nu_i} \right) \quad (62)$$

$$dq \leftarrow dq + \frac{2\nu_{i-1}}{L + \mu} \cdot d\xi_{i-1} \quad (63)$$

$$dH \leftarrow dH - \frac{2\nu_{i-1}}{L + \mu} \cdot d\xi_{i-1} \otimes \xi_{i-2}^\top \quad (64)$$

Output: dq, dH ;

Table 3. Model sizes and the corresponding architectural configurations.

Model Size	Number of Layers	Number of Heads	Hidden Dimension
440M	28	8	1024
1B	28	12	1536
2.8B	32	20	2560

Table 4. Model sizes and the corresponding architectural configurations.

Model Size	Global Batch Size	Total Number of Training Tokens	Sequence Length
440M	1M	8B	2048
1B	2M	20B	2048
2.8B	2M	100B	4096

Model Hyperparameters. We use default parameters for all other models as given in the [Flash-Linear-Attention v0.4.0](#) library (except the ones mentioned in Table 3). For our approach, we use $\lambda_t = 0.02 \cdot \|H_t\|_F$, with gating and α -connection enabled by default, unless otherwise specified. We also run CH for 30 iterations for all experiments.

Individual Experiments. We now describe the setups for each individual experiment.

In Fig. 1a, we randomly generate tensors $k \in \mathbb{R}^{B \times T \times H \times D}$ and q and normalize them along the last dimension (D). Here B, T, H, D simulate the batch size, sequence length, number of heads, and head dimension, respectively. Then we compute the covariance matrices $H \in \mathbb{R}^{B \times T \times H \times D \times D}$ of k , normalize its every $D \times D$ slice by its Frobenius norm. The code to generate data is shown below.

```
k = torch.randn(B, T, H, D).to(dtype).to('cuda')
q = torch.randn(B, T, H, D).to(dtype).to('cuda')

q = q / torch.linalg.vector_norm(q, dim=-1, keepdim=True).to(q)
k = k / torch.linalg.vector_norm(k, dim=-1, keepdim=True).to(k)

kk = torch.einsum('...i,...j->...ij', k, k).cumsum(1)
kk = kk / torch.linalg.matrix_norm(kk, ord='fro')[..., None, None]
```



```
kk.diagonal(dim1=-2, dim2=-1).add_(ridge_strength)
```

For Fig. 1c and Fig. 1e, we generate random input ids with vocabulary size 5000, sequence length 2048 within a 5-layer LLAMA; we set 2 heads and head dimension 128 for this architecture.

In the MQAR experiments of Fig. 3a, we follow the standard experimental setting but consider a strictly harder setting with smaller model dimension (or hidden dimension). Indeed, in the setting of [1], the model dimension is always larger than or equal to the number of KV pairs, while in the setting here, in some cases the model dimension is smaller than the number of KV pairs, in which case linear SSMs could not perfectly memorize all KV pairs.

In the main paper, Fig. 3a is without any gating or α -connection.

In Fig. 4 we considered the following tasks for long context evaluations. Reported results for each task is average over the score obtained for individual datasets in that task.

- *Retrieval-Augmented Generation (RAG)*: These tasks consist of open-domain question answer where the model is given a gold passage (passage containing the answer) interspersed between many other retrieved passages from a corpus [48, Wikipedia dump split into 100-word passages]. The model is tasked with answering the question based on the obtained passages. We consider the following datasets from HELMET [68] for this task: Natural Questions, TriviaQA, PopQA, HotpotQA.
- *Many-shot In-Context Learning (ICL)*: ICL tests LLMs ability to learn new skills from a few examples. Here the task is to learn to classify between different concepts based on several in-context examples of the said concept. We consider the following datasets from HELMET [68] for this task: TREC Coarse, TREC Fine, NLU, BANKING77, CLINIC150.
- *Synthetic Recall*: These tasks are variations of the “Needle-in-a-Haystack” task [29] where the goal is to retrieve an important piece of information, the “needle” from a long context of distractor tokens, the “haystack”. These variations also test multi-hop tracing and aggregation capabilities of the model. We consider the following datasets from RULER [25] for this task: S-NIAH-1/2/3, MK-NIAH-1,2,3, MV-NIAH, MQ-NIAH, VT, CWE, FWE.
- *LongQA*: These are long document based question-answering tasks. The documents are typically made long by randomly sampling different paragraphs from the same dataset along with the paragraph that contains the answer. We consider the following datasets from RULER [25] for this task: SQuAD, HotpotQA.

E. How Does The Performance GKA Scale with Compute?

We consider models at three different scales: 440M, 1B and 2.8B. For training configurations and architecture refer to Section D. We use prototypical tasks from LM-Harness (see Section 5.3.1 for list of tasks) to evaluate language modeling capabilities of GKA and compare with baseline SSM/fading memory layers. Table 5 shows that at 440M scale, GKA is competitive with GDN and Deltanet. However, differences emerge at larger scales, with GKA showing increasing benefits. In particular, the retrieval capabilities of our model, as measured by FDA and SWDE consistently outperform all SSM baselines at 1B and 2.8B scale. We also report the results of equal-sized Transformer for completeness, which serves as a performance ceiling at each scale.

Table 5. **GKA shows stronger scaling with compute than other SSM baseline models.** LM-Harness results for models at different scales: 440M, 1B and 2.8B. All models were trained from scratch. 440M and 1B models were trained on 8B and 20B tokens respectively in accordance to the Chinchila scaling laws [24]. For the 2.8B model we trained on 100B tokens.

Model	ARC-C acc_n ↑	ARC-E acc_n ↑	BoolQ acc ↑	COPA acc ↑	HellaSWAG acc_n ↑	PIQA acc_n ↑	SciQ acc_n ↑	Winogrande acc ↑	FDA contains ↑	SWDE contains ↑	Avg
<i>440M Models</i>											
Transformer	24.40	<u>42.26</u>	<u>59.88</u>	70.00	36.19	64.15	61.50	51.70	5.17	35.64	45.09
Gated Linear Attention	24.06	40.28	56.57	<u>71.00</u>	32.70	62.24	57.80	50.67	1.00	9.18	40.55
Gated DeltaNet	25.17	41.96	58.23	72.00	36.96	64.69	<u>63.6</u>	51.7	1.91	11.88	<u>42.81</u>
DeltaNet	<u>25.09</u>	41.92	61.13	65.00	<u>37.20</u>	<u>64.47</u>	64.00	49.49	<u>2.81</u>	<u>14.31</u>	42.54
Gated KalmaNet (Ours)	24.57	43.22	56.94	<u>71.00</u>	37.22	<u>64.47</u>	62.8	50.83	1.45	14.04	42.65
<i>1B Models</i>											
Transformer	26.62	46.42	59.94	77.00	44.01	67.14	68.30	54.06	8.35	45.18	49.70
Mamba2	28.07	<u>46.63</u>	<u>60.21</u>	70.00	<u>44.57</u>	67.57	65.50	<u>54.30</u>	1.45	15.75	45.40
Gated Linear Attention	25.94	42.00	58.84	70.00	36.34	63.60	58.20	51.85	1.45	10.53	41.88
Gated DeltaNet	27.05	47.98	59.54	<u>74.00</u>	44.27	67.36	66.2	53.83	2.18	17.82	46.02
DeltaNet	<u>27.56</u>	46.25	59.97	71.00	43.18	<u>67.74</u>	65.90	55.41	3.09	20.61	46.07
Gated KalmaNet (Ours)	25.43	46.55	60.73	<u>74.00</u>	44.59	68.88	<u>67.60</u>	52.41	<u>6.17</u>	<u>21.87</u>	<u>46.82</u>
<i>2.8B Models</i>											
Transformer	32.25	56.10	64.28	80.00	60.96	73.56	79.50	61.72	58.53	72.28	63.92
Mamba2	32.24	59.64	58.72	<u>82.00</u>	62.23	73.78	79.80	62.19	7.71	41.13	55.94
Gated Linear Attention	27.82	50.80	52.57	78.00	48.83	70.13	69.60	54.54	2.81	20.43	47.55
Gated DeltaNet	<u>32.59</u>	60.02	<u>62.75</u>	<u>82.00</u>	<u>62.8</u>	<u>74.32</u>	<u>80.6</u>	<u>62.35</u>	8.26	44.28	57.00
DeltaNet	32.85	58.16	42.51	81.00	61.13	73.78	43.90	61.72	11.80	46.08	51.29
Gated KalmaNet (Ours)	32.51	<u>59.89</u>	61.68	85.00	63.84	74.81	83.2	64.17	<u>12.89</u>	<u>50.95</u>	<u>58.89</u>

F. Ablations

In this section we consider ablations for various modeling choices made in arriving at our final GKA model. For all ablations, we consider 2.8B models trained on 100B tokens on DCLM at 4K context length (unless mentioned otherwise). We use the same architecture and training configurations for these ablations as mentioned in Section D.

F.1. Does Adaptive Regularization Help?

As discussed in Section 4.1, we introduced adaptive regularization to control the condition number of $H_T + \lambda_t I$ for numerical stability. Here we ablate this choice, specifically we compare the following runs.

1. *Adaptive regularization.* We train a model with $\lambda_t = a||H_t||_F$. We report results for $a = 0.02$ for this run.
2. *Constant regularization* We train same model architecture (as above) with $\lambda_t = 0.25$ (a constant). This choice of 0.25 is motivated from concurrent work [59] which explored a similar ridge regression objective for LLM training.

As shown in Fig. 6, without strict condition number control, gradient norms spike during training, leading to increased cross entropy loss (compared to the run with adaptive regularization).

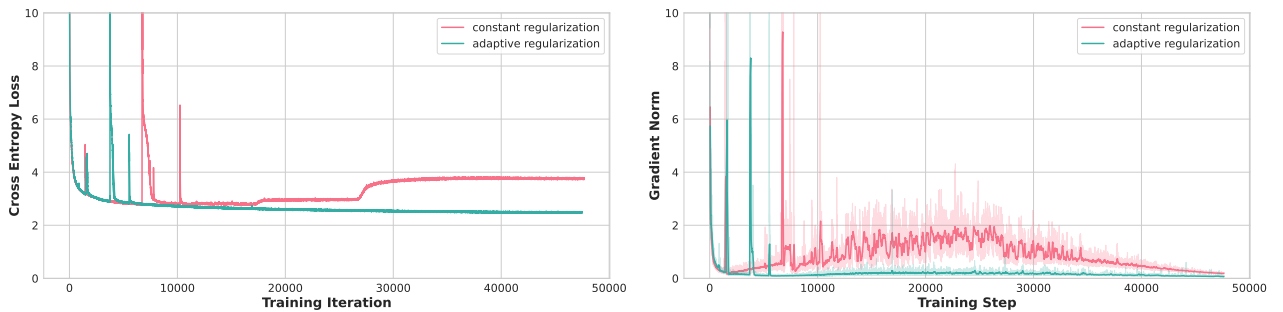


Figure 6. **Adaptive regularization results in smoother and better training curves.** (a) Plots the training curve for 2.8B models on 100B tokens from DCLM. (b) Plots the corresponding gradient norm. The model with constant regularization (red curve) results in a higher loss that can be attributed to its non-smooth trajectory over the course of its training run (spiky gradient norms).

F.2. Does Adaptive Weighting Help?

In Section 4.1, we discussed increasing the expressivity of our layer by introducing adaptive weights $\eta_{t,i}$ which re-weigh the past to be exponentially decaying in time. Given constant-sized memory, we hypothesize this adaptive weighting (gating) allows GKA to learn an effective representation by incorporating recency bias into its computation. In this subsection we test this hypothesis. We carry out the following runs.

1. *Adaptive weighting (gating)*. We train a model with adaptive weights. Specifically, for all $t \geq i$, we parameterize the weight for the i^{th} sample at time-step t as $\eta_{t,i} = \prod_{j=i+1}^t \gamma_j$, with each $\gamma_j \in [0, 1]$ learnable.
2. *No weighting*. We train the same model architecture as above, but with no weights. This essentially results in an unweighted ridge regression objective obtained by setting $\eta_i = 1$ in (3).

Table 6 shows clear benefits of adapting weighting with improvements across the board on all LM-Harness tasks considered, thereby validating our hypothesis.

Table 6. **Adaptive weighting outperforms across the board on LM-Harness tasks.** Results for 2.8B models trained on 100B tokens from DCLM with and without adaptive weights as introduced in Section 4.1.

Adaptive Weights	ARC-C acc_n \uparrow	ARC-E acc_n \uparrow	BoolQ acc \uparrow	COPA acc \uparrow	HellaSWAG acc_n \uparrow	PIQA acc_n \uparrow	SciQ acc_n \uparrow	Winogrande acc \uparrow	FDA contains \uparrow	SWDE contains \uparrow	Avg
\times	28.24	51.73	57.68	76	53.87	71.87	71.6	54.38	6.08	33.03	50.45
\checkmark	32.51	59.89	61.68	85	63.84	74.81	83.2	64.17	12.89	50.95	58.89

F.3. Does α -connection Improve Training of GKA?

In Section 4.3, we introduce the α -connection as a residual connection that establishes a direct path for gradient flow through the GLA solution, improving training stability. This allows the model to fall back on the GLA solution when CH produces poor-quality results due to non-convergence of the iterative solver within the fixed iteration budget. To validate this design choice, we perform two runs.

- R1. *with α -connection*. We train a model with the α -connection as shown in our GKA block in Fig. 2.
- R2. *without α -connection*. We train the same model architecture as above, but with no α connection. This can be simply understood as setting $\alpha_t = 1$ for all time-steps t in Fig. 2.

On LM-Harness, both models perform similarly, with R1 and R2 achieving aggregate scores of 58.89 and 58.39, respectively. However, clear differences emerge under long-context evaluation, where we trained both models on an additional 25B tokens from long documents at 128K context length. Fig. 7 shows that GKA without the α -connection exhibits inferior long-context performance on average, with Synthetic Recall and LongQA showing major degradation.

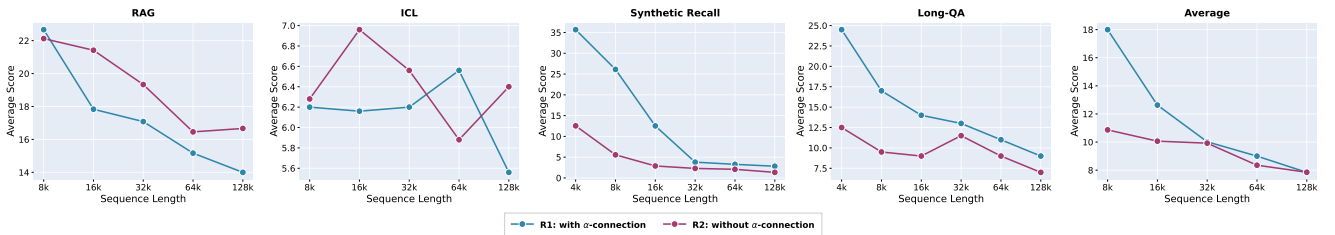


Figure 7. **GKA without the α -connection severely underperforms on Synthetic Recall and LongQA.** On ICL all SSMs struggle to perform better than random chance (see Fig. 4). Interestingly, although R2 exhibits poorer long-context abilities in aggregate, it outperforms R1 on RAG by a few points.

G. Effects of Different Regularization Strengths

Recall that we proposed setting adaptive regularization $\lambda_t = a \cdot \|H_t\|_F$. We now present experiments validating this choice. **Synthetic Experiments.** First, we generate data as per Fig. 1a, where the covariance matrix is normalized by its Frobenius norm. In this case we set $\lambda_t = a$ for a varying in $\{0.01, 0.02, 0.05, 0.1\}$. Fig. 8 shows that the *maximum regularized residual norm* (computed as the maximum of $\|(H_t + \lambda_t I)\xi_i - q\|_2$ over all dimensions where ξ_i is the estimate of CH at iteration i) decreases as we enlarge λ_t . This is because having a large λ_t reduces the condition number. The downside, though, with a large λ_t is that it reduces the memorization capacity, namely, it might enlarge $\|H_t \xi_i - q\|_2$, the true residual of interest.

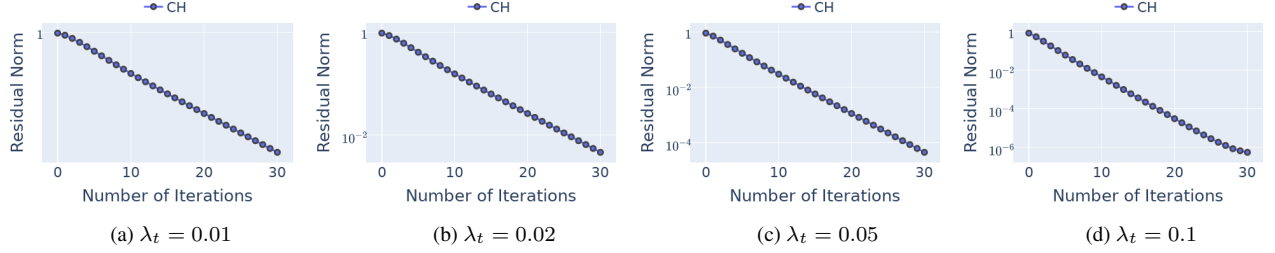


Figure 8. Convergence for varying regularization strengths (batch size 8, sequence length 2048, 8 heads, and head dimension 128).

GKA with different regularization strengths. We train several 2.8B models with varying regularization strength by choosing $a \in [0.01, 0.02, 0.05, 0.1]$. While performance on LM-Harness (Table 7) shows little discrepancy, we observe noticeable differences in long-context performance—where memorization capacity matters most—(Fig. 9). Specifically, the long-context performance of GKA improves initially as we decrease a from $0.1 \rightarrow 0.05$. This is expected since this increases the memorization capacity of the model. However, decreasing further from $0.05 \rightarrow 0.02 \rightarrow 0.01$ causes performance to decrease. This can be attributed to the increasing condition number of the problem, which reduces the quality of the solution computed by CH (Fig. 8).

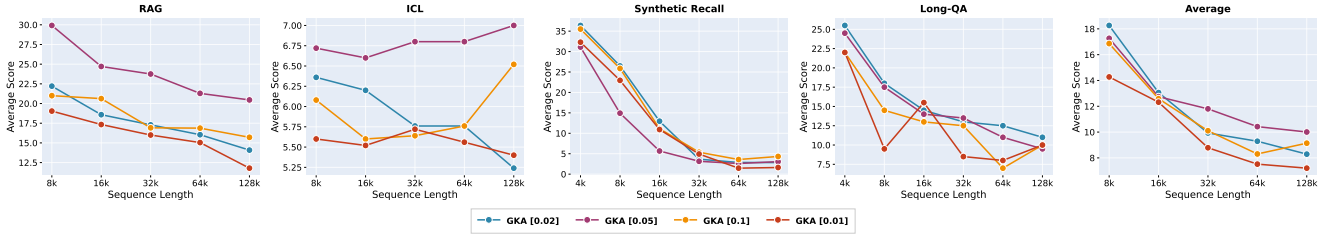


Figure 9. **Long context performance GKA for different regularization strengths.** The long-context performance of GKA improves initially as we decrease a from $0.1 \rightarrow 0.05$. This is expected since this increases the memorization capacity of the model. However, decreasing further from $0.05 \rightarrow 0.02 \rightarrow 0.01$ causes performance to decrease. This can be attributed to the increasing condition number of the problem, which reduces the quality of the solution computed by CH (Fig. 8)

Table 7. **Ablation over different choices of regularization strength** $\lambda_t = a \cdot \|H_t\|_F$. Short-context performance on LM-Harness shows little discrepancy with different regularization strengths.

a	ARC-C acc_n \uparrow	ARC-E acc_n \uparrow	BoolQ acc \uparrow	COPA acc \uparrow	HellaSWAG acc_n \uparrow	PIQA acc_n \uparrow	SciQ acc_n \uparrow	Winogrande acc \uparrow	FDA contains \uparrow	SWDE contains \uparrow	Avg
0.01	33.45	58.63	62.63	85.00	63.36	73.99	81.40	63.14	11.16	51.49	58.43
0.02	32.51	59.89	61.68	85.00	63.84	74.81	83.20	64.17	12.89	50.95	58.89
0.05	32.68	61.66	53.57	79.00	63.46	74.84	82.60	63.77	11.98	49.68	57.32
0.1	32.76	59.85	63.52	84.00	63.95	75.08	83.20	63.54	11.43	51.22	58.86

Table 8. **Latent sketching increases training throughput (by up to 10%) while marginally reducing accuracy (< 1%).** Training throughput is reported in # Billion tokens/day/node. It is measured on a single H200 GPU with a batch size of 1M tokens. Our results indicate minimal regression on LM-harness tasks but up to 10% improvement in training throughput (going from no-sketch to sketch dim 32). However, long context performance is adversely affected with sketching with up to 60% relative drop in performance. Future work will address this by exploring the use of sketching adaptively depending on the "complexity" of the task.

Sketch dimension	LM-Harness avg.	Training throughput
32	57.57	8.37
no-sketch	58.89	7.65

H. Latent Sketching for Approximate Solutions

We introduce the idea of sketching from random matrix theory to further control the amount of FLOPs vs accuracy in GKA. Sketching involves down projecting the normal equations into a low-dimensional subspace, solving the equations in this subspace and finally up-projecting the solution back to the original space. This reduces the worst-case computational complexity of our approach from $\mathcal{O}(D^2r)$ to $\mathcal{O}(d^2r)$, where $d \ll D$ and r is the number of iterations in Algorithm 1. To the best of our knowledge our work is the first one introducing sketching as a viable solution to increase efficiency of neural network layers that are defined implicitly by the solution to an optimization problem. Sketching can be thought of as an analogous to the Multi Latent Attention idea introduced by DeepSeek but applied to fading memory layers. Table 8 shows preliminary results of this idea applied to GKA. Both models (no-sketch and sketch dim 32) are trained from scratch at 2.8B scale on 100B tokens.

I. Hybrid Gated KalmaNet

As discussed in Section A.2, augmenting SSM models with Attention layers has proven to be an effective way of improving performance on tasks that require recalling information from the distant past. In this section, we show that our Gated KalmaNet layer can be interleaved with Attention layers to yield even stronger models. Our Hybrid GKA model is based on the Qwen3 architecture [63]. Namely, our Hybrid model consists of a stack of "decoder" blocks, each of which contains a sequence mixer—either Attention or GKA—followed by an MLP. Similar to Qwen3, our Attention layers use QK normalization layers. Our Hybrid model consists of 30 decoder blocks, 26 of which use GKA as the sequence mixer, and 4 that use Attention. The Attention decoder blocks are at indices 6, 14, 22, and 29. Our Hybrid models follow the same training procedure as our non-Hybrid models. Specifically, we pretrain our Hybrid model on 100B tokens with a 4K context size, followed by fine-tuning on 25B tokens at a 128K context size.

When evaluating our pretrained Hybrid model standard NLP benchmarks, we observe that it improves substantially on recall-oriented tasks (FDA & SWDE) compared to the non-Hybrid model⁴, as shown in Table 9. Further, when evaluating our fine-tuned long-context model on tasks that require effective modeling of long-range dependencies, we observe a significant improvement across all context lengths, as shown in Fig. 10.

Table 9. **Our Hybrid GKA + Attention model improves language modeling performance.** When interleaving Attention layers into our GKA models, we observe a significant improvement on recall-oriented tasks, such as FDA and SWDE, while preserving a similar performance on short-context tasks.

Model	ARC-C acc_n ↑	ARC-E acc_n ↑	BoolQ acc ↑	COPA acc ↑	HellaSWAG acc_n ↑	PIQA acc_n ↑	SciQ acc_n ↑	Winogrande acc ↑	FDA contains ↑	SWDE contains ↑	Avg
Gated KalmaNet (Hybrid)	33.02	59.47	64.07	80.00	62.74	74.59	81.40	64.64	53.18	72.46	64.56
Gated KalmaNet	32.51	59.89	61.68	85.00	63.84	74.81	83.20	64.17	12.89	50.95	58.89

⁴Note, our non-hybrid model shares the same architecture as the hybrid with the distinction that all 4 Attention layers are replaced with GKA layers.

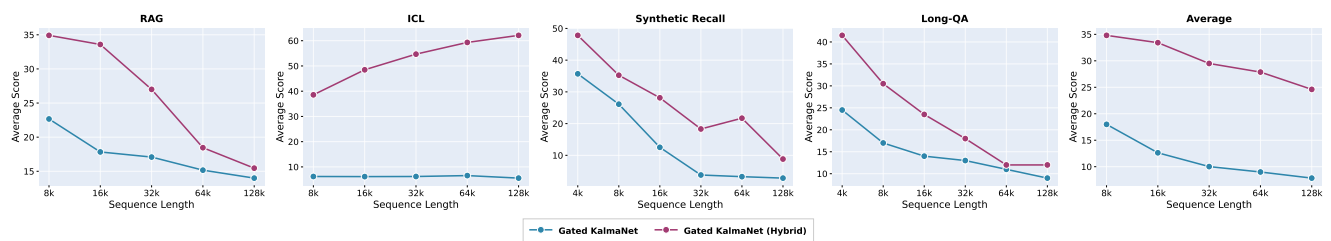


Figure 10. **Our Hybrid GKA + Attention model significantly improves performance across all long-context benchmarks compared to our non-Hybrid model.** Adding a few Attention layers to our GKA model improves long-range dependency modeling, improving performance across all sequence lengths on RAG, ICL, Synthetic Recall, and Long-QA.