Multi-stream Sequence Learning

Mohamed Elsayed [1 2] **A. Rupam Mahmood** [1 2 3]

Abstract

We re-evaluate the suitability of the independent and identically distributed (IID) training paradigm for sequence learning, where long data streams are segmented into shorter and shuffled chunks, thereby breaking their natural continuity and undermining long-range credit assignment. This paper offers *multi-stream sequence learning*, a training framework that presents multiple data streams in their natural order. To support this framework, we propose *Memora*, a recurrent-only architecture whose persistent hidden states make it more suitable for sequence learning than Transformers. Memora builds on *Gated Linear Recurrent Unit* (GLRU)—a new lightweight recurrent unit designed for efficient parallel training and robust temporal reasoning—and achieves effective learning on long byte-level sequences. Our experiments on structured and byte-level benchmarks demonstrate that models trained under the multi-stream sequence learning framework consistently outperform standard recurrent and state-space models trained with IID training setting, underscoring the importance of preserving continuity in sequence learning.

1. Introduction

Modern sequence models, particularly Transformers, have achieved remarkable performance across diverse domains by adopting the independent-and-identically-distributed (IID) training paradigm where long continuous data streams are partitioned into randomized, fixed-length segments. This segmentation strategy fully exploits modern hardware, achieves high-throughput parallel processing, and underpins state-of-the-art results in language, vision, and beyond (Brown et al. 2020, Hoffmann et al. 2022, Touvron et al. 2023, Guo et al. 2025). Yet this very convenience comes at the expense of severing the temporal continuity intrinsic to data streams. Natural streams (e.g., linguistic text, audio waveforms, videos, or genomic code) may rely on dependencies that span far beyond individual chunks. As a stopgap, practitioners extend context windows (Ding et al. 2024, Pal et al. 2023, Wang et al. 2024b), an approach that rapidly escalates computational cost and becomes impractical for those with limited resources (Huang et al. 2024). This tension invites a pivotal question: does the IID regime inherently constrain the capacity to learn genuine long-range dependencies?

The temporal segmentation not only disrupts the natural continuity of streams but also hinders the model's ability to capture dependencies that extend beyond segment boundaries. Transformers, when trained on arbitrarily chunked windows, lack mechanisms to relate patterns across segments, leaving long-range structure unmodeled. While recurrent architectures (Hochreiter and Schmidhuber 1997, Arjovsky et al. 2016, Orvieto et al. 2023) are, in principle, better suited for such dependencies by maintaining hidden state over time, modern variants typically conform to the same IID regime. In practice, they reset hidden states at segment boundaries (e.g., Gu and Dao 2024, De et al. 2024), discarding accumulated memory and undermining their temporal expressivity. A recurrent framework that fully leverages persistent state across long sequences remains largely unrealized.

Motivated by the limitations of the IID training paradigm and the untapped capacity of recurrent architectures, we propose the *multi-stream sequence learning* paradigm, a training framework that maintains temporal coherence by presenting multiple parallel data streams in their natural order and resetting states only at stream boundaries (e.g., end of document). Unlike IID-based chunking, this approach makes learning updates sequentially, enabling the model to preserve contextual memory and build representations that extend beyond each update block. This paradigm aligns naturally with real-world applications like online inference and real-time decision-making, where the assumption of independent samples does not hold. Under this setting, a recurrent architecture can maintain a persistent state across the entire stream to perform long-range credit assignment, unlocking modeling capabilities that IID training inherently disrupts and revitalizing the strength of recurrence.

[1]Department of Computing Science, University of Alberta, Edmonton, Canada [2]Alberta Machine Intelligence Institute [3]CIFAR AI Chair. Correspondence to: Mohamed Elsayed <mohamedelsayed@ualberta.ca>.

Efficient Systems for Foundation Models Workshop at the International Conference on Machine Learning (ICML). Copyright 2025 by the author(s).

Realizing multi-stream sequence learning at scale requires an architecture that balances the efficiency of parallel training with the expressivity of recurrence—we propose *Memora*, a lightweight recurrent-only backbone built on the *Gated Linear Recurrent Unit* (GLRU). GLRU employs gating mechanisms with parallelizable formulation, narrowing the throughput gap with Transformers. GLRU enables stable performance under various update strides, including a stride of 1, making Memora adaptable to diverse training scenarios, from offline pretraining to real-time learning.[1]

Through extensive evaluations on byte-level sequence modeling, we demonstrate that Memora trained under the multi-stream paradigm consistently surpasses both standard recurrent and state-space models trained with the IID paradigm, and rivals Transformer models. Our work contributes (1) a formalization of multi-stream sequence learning paradigm, (2) the Memora architecture with the GLRU cell for efficient, scalable recurrence supporting both truncated backpropagation through-time and real-time recurrent learning, and (3) empirical evidence that continuity-aware training unlocks performance gains on long-sequence tasks for recurrent-based models.

2. A new learning framework

We propose multi-stream sequence learning, a new training framework designed to preserve temporal continuity across update blocks by presenting multiple data streams in their natural order. Unlike IID training, where update blocks are randomized chunked segments of the original stream, our approach treats each stream temporally in a continuous manner and only presents a new stream at semantically meaningful boundaries (e.g., end of document or video).

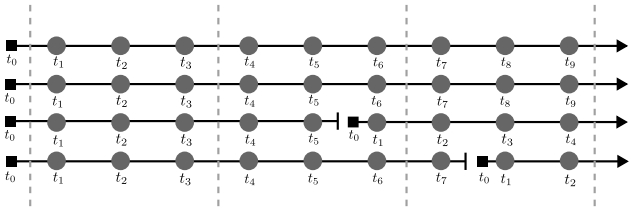


Figure 1. Overview of multi-stream sequence learning. Four streams (B=4) each yield blocks of length T=3; when an episode ends it is immediately replaced by a new one starting at t_0.

Figure 1 illustrates the workflow of the multi-stream sequence learning. The learner is presented with B parallel streams, each containing a series of episodes (e.g., an article, a video, or a complete conversation). When one episode ends, we dynamically swap it with a new episode that starts from t_0. Training is done by collecting a block of stride T from each stream, giving an effective batch size of $B \times T$ for an update. This setup generalizes several regimes, for example, when $B = 1$ and $T = 1$, the model reduces to what is known as streaming learning (e.g., Elsayed et al. 2024). We refer to T as the update stride, which is known as the sequence length for IID training. However, we reserve the latter for IID training, where—unlike in multi-stream training—the sequence is broken at the block boundary.

While each episode is randomly selected from the corpus, the temporal coherence within each episode, which can be as long as a whole book, is preserved. This enables the model to maintain and evolve hidden states across update blocks, supporting long-range credit assignment and structured memory accumulation. Recurrent architectures naturally lend themselves to this paradigm, in contrast to Transformer-based models. Yet, current recurrent learning paradigms do not evolve hidden states across update blocks, as they pick the blocks IID. In contrast, update blocks in multi-stream learning are contiguous and from the same episode unless the episode ends before a block completes.

Multi-stream sequence learning aligns closely with real-world deployment scenarios, such as online prediction (Carreira et al. 2024), continual learning (Elsayed and Mahmood 2024), and real-time control (Vasan et al. 2024), where the assumption of independent samples breaks down. Furthermore, it retains compatibility with modern hardware by enabling efficient parallelism across streams, while unlocking new capabilities through temporal continuity. We provide a comprehensive background on recurrent learning in Appendix A and the related works in Appendix D.

3. The Memora Architecture

To study the multi-stream learning framework, we introduce the Memora architecture, a simplified recurrent-only architecture that is compatible with the multi-stream learning paradigm. The Memora architecture depends on the Gated Linear Recurrent Unit (GLRU), which is a gated recurrent architecture. We start by describing GLRU and explain how it can be used with T-BPTT and RTRL; then we describe the components of the Memora architectural design.

3.1. Learning Temporal Dependencies with GLRU

Recent advancements in recurrent learning (e.g., Gu and Dao 2024, De et al. 2024) demonstrated the importance of gated recurrence in solving complex tasks such as language. In addition, gated recurrent has been shown to be able to implement the linear attention (Katharopoulos et al. 2020) operation (Zucchet et al. 2024, Huang et al. 2023, Dao and Gu 2024). Our design builds on previous works and relies on gating both the input and the state. Specifically, GLRU builds on LRU (Orvieto et al. 2023), which uses complex-

[1] We provide a minimal easy-to-follow implementation of Memora under the IID and multi-stream settings through this Colab Notebook.

valued, non-gated recurrence, but incorporates a real-valued, gated recurrence instead. The GLRU recurrence formulation is given by

$$\boldsymbol{h}_t = \boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma}_t \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t), \tag{1}$$

where $\boldsymbol{\gamma}_t = \sqrt{1-\boldsymbol{r}_t^2}$, $\boldsymbol{g}(\boldsymbol{x}_t) = \boldsymbol{G}\boldsymbol{x}_t$, $\boldsymbol{r}(\boldsymbol{x}_t) = e^{-ce^{\boldsymbol{\nu}} \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t)}$, $\boldsymbol{x}_t \in \mathbb{R}^d$, $\boldsymbol{h}_t \in \mathbb{R}^n$, and $\boldsymbol{G}, \boldsymbol{B} \in \mathbb{R}^{n \times d}$. We set c to 3. We do not apply gating on the output since the Memora architecture provides that gating, similar to the Hawk architecture (De et al. 2024). The output of the recurrence is given by $\boldsymbol{y}_t = \boldsymbol{h}_t$. Learning with T-BPTT is straightforward and can be done using Algorithm 2. However, learning with RTRL (see Algorithm 1) requires deriving the sensitivity matrices for each learnable parameter in the recurrence equation, namely: $\boldsymbol{\nu}, \boldsymbol{B}, \boldsymbol{R}, \boldsymbol{G}$. We provide the update equations here and defer the full derivation to Appendix F. The RTRL sensitivity update equations are given by

$$\begin{aligned}
\mathbf{S}_t^{\boldsymbol{\nu}} &= \boldsymbol{r}(\boldsymbol{x}_t) \circ \mathbf{S}_{t-1}^{\boldsymbol{\nu}} - ce^{\boldsymbol{\nu}} \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t) \circ \boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} \\
&\quad + c\frac{\boldsymbol{r}(\boldsymbol{x}_t)^2}{\boldsymbol{\gamma}} \circ e^{\boldsymbol{\nu}} \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t) \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t), \\
\mathbf{S}_t^{\boldsymbol{B}} &= \operatorname{Diag}(\boldsymbol{r}(\boldsymbol{x}_t))\, \mathbf{S}_{t-1}^{\boldsymbol{B}} + (\boldsymbol{\gamma} \circ \boldsymbol{g}(\boldsymbol{x}_t))\boldsymbol{x}_t^\top, \\
\mathbf{S}_t^{\boldsymbol{G}} &= \operatorname{Diag}(\boldsymbol{r}(\boldsymbol{x}_t))\, \mathbf{S}_{t-1}^{\boldsymbol{G}} + (\boldsymbol{\gamma} \circ (\boldsymbol{B}\boldsymbol{x}_t))\boldsymbol{x}_t^\top, \\
\mathbf{S}_t^{\boldsymbol{R}} &= \operatorname{Diag}(\boldsymbol{r}(\boldsymbol{x}_t))\boldsymbol{S}_r^{\boldsymbol{R}} \\
&\quad + \left(\boldsymbol{d} \circ \left(\boldsymbol{h}_{t-1} - \frac{\boldsymbol{r}(\boldsymbol{x}_t)}{\boldsymbol{\gamma}} \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t)\right)\right)\boldsymbol{x}_t^\top,
\end{aligned}$$

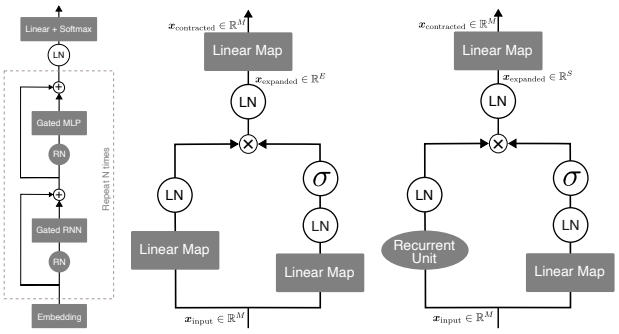
where $\boldsymbol{d} = c\boldsymbol{r}(\boldsymbol{x}_t) \circ e^{\boldsymbol{\nu}} \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t) \circ (1 - \sigma'(\boldsymbol{R}\boldsymbol{x}_t))$, and the division is performed element-wise.

Learning with RTRL allows learning in real-time from the samples as they arrive. It is compatible with our multi-stream learning paradigm and can achieve efficient learning with $T = 1$. One fundamental limitation of RTRL is that its parallelization with parallel scan (when $T > 1$) is expensive since the sensitivity equations are based on matrices. Therefore, we limit the usage to the case where $T = 1$ with B parallel streams.

Finally, we place GLRU in the landscape of recurrent units in Table 1. Notably, our GLRU allows for state expansion since the input $\boldsymbol{x}_t$ can be expanded to larger space using $\boldsymbol{B}$, similar to the LRU unit and unlike RG-LRU. Additionally, our design allows for efficient RTRL implementation, which is facilitated by the element-wise input and output gating. RG-LRU design, in principle, also allow for efficient RTRL. In contrast, the RTRL mode of GRU (Chung et al. 2014) is intractable and requires computational complexity of $O(n^4)$. The RTRL mode of RG-LRU is not introduced in the literature, so we derive its RTRL mode in Appendix F to compare it with GLRU in the experiments and skip comparing with RTRL-mode of S6 due to its derivation and implementation complexity.

3.2. Architectural Design

Memora is a recurrent-only architecture with two main components: 1) residual normalized gated recurrent block followed by 2) residual normalized MLP block. This design follows the general Transformer architecture outline where the first block captures the temporal relations and the other learns representations (see Touvron et al. 2023). We apply pre-normalization using RMSNorm (Zhang and Sennrich 2019) on each block with learnable parameters. Each block employs the gated linear unit design (Shazeer 2020) where the input is expanded then contracted using the block linear maps. This design is common across many architectures (e.g., De et al. 2024, (Gu and Dao 2024)). We propose a new variation where we use layer norm (Ba et al. 2016) without learnable parameters after each linear mapping and also after each element-wise multiplication. Our gated recurrent block is very close to the gated MLP and instead of using a linear map in one of the GLU branches, we replace it with GLRU. Lastly, the activation σ we use is GeLU (Hendrycks and Gimpel 2016). Figure 2 outlines the Memora architecture and its two main components.



(a) Memora (b) Gated MLP (c) Gated Recurrence

Figure 2. The Memora architecture. Filled blocks represent components with learnable parameters. LN/RN denotes LayerNorm/RMSNorm. M is the model dimension and S is the state dimension.

4. Experiments

In this section, we start by studying long-range memory capabilities and selective attention under high memory demand using the selective copying task (Gu and Dao 2024). Then we study the effect of varying sequence lengths for IID or update strides for multi-stream on the quality of byte-level language modeling with TinyStories (Eldan and Li 2023). After that, we study how Memora scales with the number of parameters using byte-level language modeling with FineWebEdu (Penedo et al. 2024) and DNA modeling with the human genome dataset (HG38, Schneider et al. 2017). We then study learning from one sample at a time with RTRL and 1-step BPTT. Finally, we consider GLRU alternatives to show the performance gain with GLRU when

	Gated LRU (Ours)	**RG-LRU** (De et al. 2024)	**LRU** (Orvieto et al. 2023)	**GRU** (Chung et al. 2014)
State expansion	Yes	No	Yes	Yes
RTRL mode	Efficient	Efficient	Efficient	Intractable
Gated/Selective	Yes	Yes	No	Yes
T-BPTT Scan	Efficient	Efficient	Efficient	Intractable

Table 1. Comparison of Gated LRU, RG-LRU, LRU, and GRU. Gated LRU is the first gated recurrent unit that supports different modes of training: multi-stream learning with T-BPTT, RTRL, and parallelization for learning with RTRL.

used with Memora compared to other baselines in Appendix G. Here, we focus on the key results and experimental details and provide the full experimental details and configurations in Appendix H.

4.1. Selective Copying

The selective copying task is a variation of the original copying task (Arjovsky et al. 2016), where the learner must memorize tokens at varying positions within a sequence. This task demands context-aware reasoning and effectively differentiates models that use gating mechanisms from those that do not. Gating enables models to selectively retain or discard information, making this task easier to solve. Our experimental setup follows Gu and Dao (2024) but with a reduced training budget of $50,000$ iterations instead of $200,000$, allowing us to study training efficiency under limited compute. For simplicity, we let each model see the full episode—full BPTT is used instead of truncated BPTT.

Table 2 reports validation performance for Llama2 Transformer, Hawk, Mamba2 models, and Memora variants, each with two layers. Both Memora with GLRU and Hawk quickly solve the task with relatively few parameters, using model dimensionality of 64 and state dimensionality of 256. In contrast, Transformer and Mamba require increased model dimensionality to achieve comparable results within the same number of iterations. Notably, Memora with GLRU attains the highest accuracy with the fewest parameters, relying solely on its recurrent architecture without temporal convolutions. Replacing GLRU with another gated unit (MinGRU) or a non-gated unit (LRU) leads to a significant performance drop, underscoring the superior memory capacity of GLRU.

4.2. Byte-level language modeling with varying sequence lengths or update strides

We study byte-level language modeling on the TinyStories dataset (Eldan and Li 2023), focusing on how performance scales with sequence length for IID and with update stride for multi-stream, while keeping the effective batch size BT fixed. In the multi-stream setup, sequences are presented in natural order, which we hypothesize favors recurrent models. In contrast, we do not expect Transformers to benefit from this structure due to their lack of persistent memory, but we still add multi-stream Transformer (Llama2) experiments.

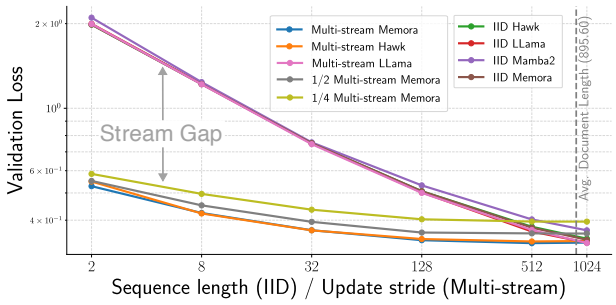


Figure 3. Performance scaling with sequence length (for the IID setting) or update stride (for the multi-stream setting). We show the final performance of different baselines trained on TinyStories using the IID and multi-stream settings. The average document size is 895.60, which means models with sequence lengths of 1024 may contain an entire document in an update block.

Figure 3 shows the performance of Mamba, Llama2, Hawk, and Memora in the IID setting, and compares them with Memora in the multi-stream setting. We also adapt Hawk and Llama2 to the multi-stream setup, representing existing recurrent-based and Transformer-based models, respectively. Sequence length (or update stride) is varied from 2 to 1024. In the IID setting, all models perform similarly but degrade significantly at shorter sequence lengths—as expected, because shorter contexts reduce temporal credit assignment. In contrast, both Memora and Hawk perform well in the multi-stream setting even with an update stride of 2. Note that multi-stream Llama2 showed no advantage over IID training due to lack of states. This suggests that maintaining a persistent state across update blocks helps recurrent models recover long-range dependencies lost in the IID setting. We further evaluate Memora in the multi-stream setting with reduced training budgets (half and quarter). Even with limited training, Memora consistently outperforms IID-trained models across nearly all sequence lengths, further emphasizing its superior memory capabilities in multi-stream.

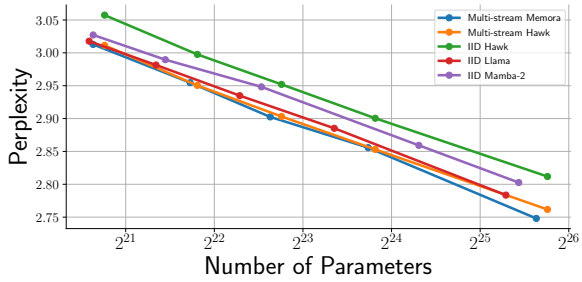
4.3. DNA modeling

DNA sequences are naturally long, presenting a challenge for modern sequence models. For instance, human chromosome 2 contains approximately 250 million base pairs (Hillier et al. 2005). We use the human genome dataset

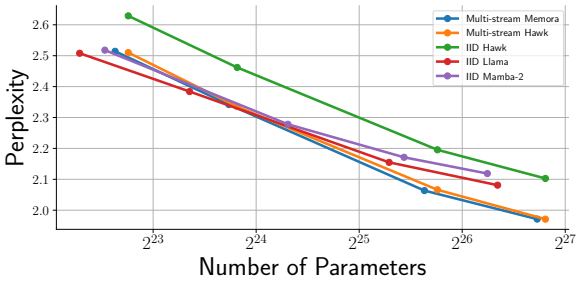
Model	Arch. Type	Accuracy	Parameter Count
Llama2	Transformer	$98.5200\% \pm 0.8701$	888,768
Hawk	Recurrent + Temp. Conv.	$99.0120\% \pm 0.4391$	438,080
Mamba2	Recurrent + Temp. Conv.	$99.4280\% \pm 0.3646$	553,828
Memora w/ MinGRU	Recurrent-only	$53.8280\% \pm 12.5294$	206,080
Memora w/ LRU	Recurrent-only	$11.0160\% \pm 0.6581$	304,896
Memora w/ GLRU	Recurrent-only	$99.5620\% \pm 0.1918$	239,360

Table 2. Validation Accuracy on the Selective Copying Task. Each model is trained for 50K iterations with batch size of 64. Each episode has a length of 4096 with only 16 numbers to remember with varying positions. We show the average over 5 runs with its standard error.

(HG38; Schneider et al. 2017) with the train-validation splits defined by Avsec et al. (2021). The sequences span up to 131,072 base pairs, making this a strong benchmark for evaluating long-range dependency modeling.



(a) HG38 DNA



(b) Deduplicated FineWebEdu

Figure 4. (A) Performance scaling with number of parameters on the HG38 DNA dataset. (b) Performance scaling with number of parameters on the de-duplicated FineWebEdu dataset. The average document size is 4667.55 and the sequence length or the update stride is 512.

In Figure 4(a), we compare the performance of Llama2, Hawk, and Mamba2 in the IID setting, and contrast them with Memora and Hawk in the multi-stream setting. In IID training, Llama2 outperforms both Hawk and Mamba2, demonstrating the strength of Transformer architectures. However, in the multi-stream setting, where a persistent state is maintained, Hawk surpasses Llama2 at most scales. Memora achieves the best overall performance, despite its simpler, recurrent-only design and lack of temporal convolutions, highlighting its strong memory capacity for capturing long-range structure.

4.4. Byte-level language modeling with FineWebEdu

Next, we next study byte-level language modeling using a deduplicated version of the FineWebEdu dataset (Penedo et al. 2024), released as part of the SmolLM dataset (Allal et al. 2025). Byte-level modeling enables flexible, modality-agnostic sequence learning across domains like text, audio, and genomics by operating directly on raw bytes, removing the need for tokenization or domain-specific preprocessing. This approach can also improve generalization by avoiding biases introduced by methods like subword tokenization (Wang et al. 2024a). However, they usually require more updates. In Figure 4(b), we present performance scaling with model size for multi-stream Memora, multi-stream Hawk, IID Llama2, IID Mamba2, and IID Hawk. We observe trends consistent with the previous task: (1) IID Llama2 outperforms both IID Hawk and IID Mamba2, and (2) multi-stream Hawk and Memora outperform IID Llama2 in most scales, with Memora slightly outperforming Hawk.

4.5. Learning from one sample at a time

Next, we study the challenging problem of learning language models using one sample (e.g., a character) at a time in an online fashion, the canonical learning mode of RTRL. Specifically, we consider a multi-stream setting in which the model receives one data point per time step from each of several parallel streams. Learning from continuous, unshuffled data streams enables a range of practical applications, including autonomous vehicles (Verma et al. 2023), video generation (Carreira et al. 2024), adaptive chatbots (Dai et al. 2025), and robotics (Vasan et al. 2024).

In Figure 5, we evaluate language modeling performance in this setting using a single update stride with 1024 parallel streams. We compare Hawk and Memora, each tested under two modes: RTRL and 1-step BPTT. We find that both modes of Memora outperform their Hawk counterparts. Additionally, Memora performs better with RTRL than with 1-step BPTT presumably because RTRL is able to assign credit over longer temporal dependencies, making it more effective for real-time learning.

Lastly, we investigate streaming sequence learning, where the model is updated at every time step with an update

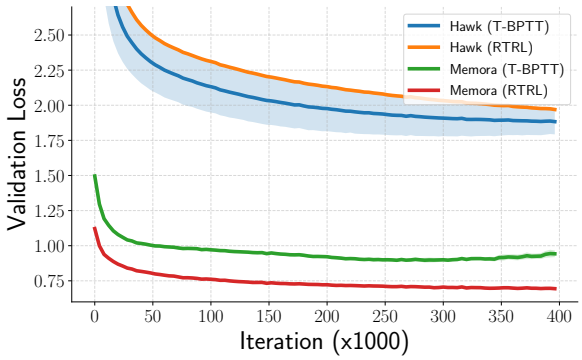


Figure 5. Learning with an update stride of 1 using 1024 parallel streams from TinyStories. We compare Hawk and Memora under the multi-stream setting with RTRL against 1-step BPTT.

stride of 1 and using a single continuous stream of data. This streaming setting—commonly explored in prior work (e.g., Goyal et al. 2009, Elsayed et al. 2024)—is particularly suited for on-device learning and real-time adaptation to non-stationary inputs, such as fine-tuning a pretrained model during deployment. Here, we tackle the challenging setting of streaming learning from scratch using byte-level inputs.

Figure 6 compares the performance of Memora trained with RTRL against 1-step BPTT over 6M iterations. Both methods reduce the validation loss, but RTRL consistently outperforms 1-step BPTT. We note here that we train both systems for 6M bytes where the dataset is about 2B bytes, which means both system are severely under trained. Nonetheless, to our knowledge, this is the first successful demonstration of deep sequence learning under strict streaming constraints. This result underscores the strong capacity of Memora to learn under stringent online learning constraints.

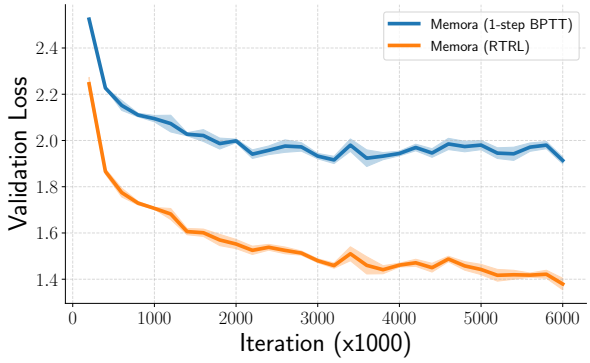


Figure 6. Streaming sequence learning (using an update stride of 1 and a single stream of data) on TinyStories. We compare RTRL against 1-step BPTT both using the Memora architecture.

5. Conclusion, limitations, and future works

This paper challenged the IID training paradigm for sequence learning and introduced multi-stream sequence modeling, a framework that preserves temporal continuity by presenting data in natural order and resetting only at meaningful boundaries (e.g., end of document). To support this paradigm, we proposed Memora, a lightweight recurrent architecture designed to maintain a persistent state across long sequences. Our results demonstrated that Memora under the multi-stream setting effectively models long-range dependencies—even beyond the sequence length—and uniquely supports learning at extremely short update strides, including stride of 1. Within the sub-120M parameter regime, Memora consistently outperforms strong baselines, including Transformers, highlighting the potential of continuity-aware training and recurrence as a viable alternative to IID-based approaches.

While this work demonstrates the effectiveness of multi-stream sequence learning and the Memora architecture on long-range language modeling tasks, several limitations remain. Our experiments are limited to models with up to 120M parameters due to computational constraints typical in academic research settings, and it remains an open question whether the observed gains persist at larger scales where Transformers dominate. The scope of our evaluation is also restricted to byte-level language modeling; extending the paradigm to other modalities such as audio, video, or vision is an important direction for future work. Finally, we focus exclusively on next-token prediction with cross-entropy loss, leaving the application of multi-stream training to settings requiring long-range credit assignment, such as reinforcement learning, for future research.

References

Allal, L. B., Lozhkov, A., Bakouch, E., Blázquez, G. M., Penedo, G., Tunstall, L., Marafioti, A., Kydlíček, H., Lajarín, A. P., Srivastav, V., et al. (2025). Smollm2: When smol goes big–data-centric training of a small language model. *arXiv preprint arXiv:2502.02737*.

Arjovsky, M., Shah, A., and Bengio, Y. (2016). Unitary evolution recurrent neural networks. In *International conference on machine learning*, pages 1120–1128. PMLR.

Avsec, Ž., Agarwal, V., Visentin, D., Ledsam, J. R., Grabska-Barwinska, A., Taylor, K. R., Assael, Y., Jumper, J., Kohli, P., and Kelley, D. R. (2021). Effective gene expression prediction from sequence by integrating long-range interactions. *Nature methods*, 18(10):1196–1203.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Bai, S., Kolter, J. Z., and Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*.

Blelloch, G. E. (1990). Prefix sums and their applications.

Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Carreira, J., King, M., Patraucean, V., Gokay, D., Ionescu, C., Yang, Y., Zoran, D., Heyward, J., Doersch, C., Aytar, Y., et al. (2024). Learning from one continuous video stream. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 28751–28761.

Cho, K. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Dai, X., Xie, Y., Liu, M., Wang, X., Li, Z., Wang, H., and Lui, J. (2025). Multi-agent conversational online learning for adaptive llm response identification. *arXiv preprint arXiv:2501.01849*.

Dao, T. and Gu, A. (2024). Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *Forty-first International Conference on Machine Learning*.

De, S., Smith, S. L., Fernando, A., Botev, A., Cristian-Muraru, G., Gu, A., Haroun, R., Berrada, L., Chen, Y., Srinivasan, S., et al. (2024). Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*.

Ding, Y., Zhang, L. L., Zhang, C., Xu, Y., Shang, N., Xu, J., Yang, F., and Yang, M. (2024). Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*.

Eldan, R. and Li, Y. (2023). Tinystories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759*.

Elelimy, E., White, A., Bowling, M., and White, M. (2024). Real-time recurrent learning using trace units in reinforcement learning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Elsayed, M. and Mahmood, A. R. (2024). Addressing loss of plasticity and catastrophic forgetting in continual learning. In *The Twelfth International Conference on Learning Representations*.

Elsayed, M., Vasan, G., and Mahmood, A. R. (2024). Streaming deep reinforcement learning finally works. *arXiv preprint arXiv:2410.14606*.

Feng, L., Tung, F., Ahmed, M. O., Bengio, Y., and Hajimirsadegh, H. (2024). Were rnns all we needed? *arXiv preprint arXiv:2410.01201*.

Goyal, A., Daumé III, H., and Venkatasubramanian, S. (2009). Streaming for large scale nlp: Language modeling. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 512–520.

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

Gu, A., Goel, K., and Ré, C. (2021). Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. (2025). Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Han, T., Gokay, D., Heyward, J., Zhang, C., Zoran, D., Pătrăucean, V., Carreira, J., Damen, D., and Zisserman, A. (2025). Learning from streaming video with orthogonal gradients. *arXiv preprint arXiv:2504.01961*.

Hayes, T. L., Cahill, N. D., and Kanan, C. (2019). Memory efficient experience replay for streaming learning. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 9769–9776. IEEE.

Hayes, T. L. and Kanan, C. (2022). Online continual learning for embedded devices. *arXiv preprint arXiv:2203.10681*.

Hendrycks, D. and Gimpel, K. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Hillier, L. W., Graves, T. A., Fulton, R. S., Fulton, L. A., Pepin, K. H., Minx, P., Wagner-McPherson, C., Layman, D., Wylie, K., Sekhon, M., et al. (2005). Generation and annotation of the dna sequences of human chromosomes 2 and 4. *Nature*, 434(7034):724–731.

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

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. (2022). Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Huang, C., Zhu, G., Wang, X., Luo, Y., Ge, G., Chen, H., Yi, D., and Wang, J. (2024). Recurrent context compression: Efficiently expanding the context window of llm. *arXiv preprint arXiv:2406.06110*.

Huang, F., Lu, K., CAI, Y., Qin, Z., Fang, Y., Tian, G., and Li, G. (2023). Encoding recurrence into transformers. In *The Eleventh International Conference on Learning Representations*.

Irie, K., Gopalakrishnan, A., and Schmidhuber, J. (2024). Exploring the promise and limits of real-time recurrent learning. In *The Twelfth International Conference on Learning Representations*.

Javed, K., Shah, H., Sutton, R. S., and White, M. (2023). Scalable real-time recurrent learning using columnar-constructive networks. *The Journal of Machine Learning Research*, 24(1):12024–12057.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. (2020). Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR.

Kyrylov, V. (2024). Accelerated Scan.

LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (2012). *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg.

Lieber, O., Lenz, B., Bata, H., Cohen, G., Osin, J., Dalmedigos, I., Safahi, E., Meirom, S., Belinkov, Y., Shalev-Shwartz, S., et al. (2024). Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*.

Orvieto, A., Smith, S. L., Gu, A., Fernando, A., Gulcehre, C., Pascanu, R., and De, S. (2023). Resurrecting recurrent neural networks for long sequences. In *International Conference on Machine Learning*, pages 26670–26698. PMLR.

Pal, A., Karkhanis, D., Roberts, M., Dooley, S., Sundararajan, A., and Naidu, S. (2023). Giraffe: Adventures in expanding context lengths in llms. *arXiv preprint arXiv:2308.10882*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Penedo, G., Kydlíček, H., allal, L. B., Lozhkov, A., Mitchell, M., Raffel, C., Werra, L. V., and Wolf, T. (2024). The fineweb datasets: Decanting the web for the finest text data at scale. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Press, O. and Wolf, L. (2016). Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859*.

Qian, R., Dong, X., Zhang, P., Zang, Y., Ding, S., Lin, D., and Wang, J. (2024). Streaming long video understanding with large language models. *Advances in Neural Information Processing Systems*, 37:119336–119360.

Qin, Z., Yang, S., Sun, W., Shen, X., Li, D., Sun, W., and Zhong, Y. (2024). HGRN2: Gated linear RNNs with state expansion. In *First Conference on Language Modeling*.

Saran, A., Yousefi, S., Krishnamurthy, A., Langford, J., and Ash, J. T. (2023). Streaming active learning with deep neural networks. In *International Conference on Machine Learning*, pages 30005–30021. PMLR.

Schneider, V. A., Graves-Lindsay, T., Howe, K., Bouk, N., Chen, H.-C., Kitts, P. A., Murphy, T. D., Pruitt, K. D., Thibaud-Nissen, F., Albracht, D., et al. (2017). Evaluation of grch38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly. *Genome research*, 27(5):849–864.

Shazeer, N. (2020). Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Vasan, G., Elsayed, M., Azimi, A., He, J., Shariar, F., Bellinger, C., White, M., and Mahmood, A. R. (2024). Deep policy gradient methods without batch updates, target networks, or replay buffers. In Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J., and Zhang, C., editors, *Advances in Neural Information Processing Systems*, volume 37, pages 845–891. Curran Associates, Inc.

Vaswani, A., Shazeer, N. M., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Neural Information Processing Systems*.

Verma, V., Maimone, M. W., Gaines, D. M., Francis, R., Estlin, T. A., Kuhn, S. R., Rabideau, G. R., Chien, S. A., McHenry, M. M., Graser, E. J., et al. (2023). Autonomous robotics is driving perseverance rover's progress on mars. *Science Robotics*, 8(80):eadi3099.

Wang, J., Gangavarapu, T., Yan, J. N., and Rush, A. M. (2024a). Mambabyte: Token-free selective state space model. In *First Conference on Language Modeling*.

Wang, X., Salmani, M., Omidi, P., Ren, X., Rezagholizadeh, M., and Eshaghi, A. (2024b). Beyond the limits: A survey of techniques to extend the context length in large language models. *arXiv preprint arXiv:2402.02244*.

Williams, R. J. and Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2(4):490–501.

Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280.

Xue, L., Barua, A., Constant, N., Al-Rfou, R., Narang, S., Kale, M., Roberts, A., and Raffel, C. (2022). Byt5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306.

Yu, L., Simig, D., Flaherty, C., Aghajanyan, A., Zettlemoyer, L., and Lewis, M. (2023). Megabyte: Predicting million-byte sequences with multiscale transformers. *Advances in Neural Information Processing Systems*, 36:78808–78823.

Zhang, B. and Sennrich, R. (2019). Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32.

Zucchet, N., Kobayashi, S., Akram, Y., Oswald, J. V., Larcher, M., Steger, A., and Sacramento, J. (2024). Gated recurrent neural networks discover attention.

Zucchet, N., Meier, R., Schug, S., Mujika, A., and Sacramento, J. (2023). Online learning of long-range dependencies. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Zucchet, N. and Orvieto, A. (2024). Recurrent neural networks: vanishing and exploding gradients are not the end of the story. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

A. Background on Recurrent Learning

Let us consider a recurrent module with dynamics that can be written as $\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t, \boldsymbol{\theta})$, where $\boldsymbol{h}_t \in \mathbb{R}^n$ is the hidden-state vector, $\boldsymbol{x}_t \in \mathbb{R}^d$ is the input vector, and $\boldsymbol{\theta}$ is a set of learnable parameters of the recurrence function, containing the input-weight and recurrence-weight matrices. The output is given by $\boldsymbol{y}_t = g(\boldsymbol{h}_t, \boldsymbol{x}_t, \boldsymbol{\phi})$, where $\boldsymbol{y}_t \in \mathbb{R}^m$, and $\boldsymbol{\phi}$ is a set of learnable parameters of the output function. To learn the parameters, we need to compute $\nabla \mathcal{L}_{\boldsymbol{W}},\ \forall \boldsymbol{W} \in \boldsymbol{\theta}$ and $\nabla \mathcal{L}_{\boldsymbol{V}},\ \forall \boldsymbol{V} \in \boldsymbol{\phi}$. The gradient of the loss with respect to $\boldsymbol{W}$, treating it as a flattened vector, is given by

$$\frac{\partial \mathcal{L}(\boldsymbol{y}_t, \hat{\boldsymbol{y}}_t)}{\partial \boldsymbol{W}} = \left(\frac{\mathrm{d}\boldsymbol{h}_t}{\mathrm{d}\boldsymbol{W}}\right)^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t}, \quad \forall \boldsymbol{W} \in \boldsymbol{\theta}. \tag{2}$$

A.1. Backpropagation Through Time (BPTT)

Note that to obtain $\frac{\mathrm{d}\boldsymbol{h}_t}{\mathrm{d}\boldsymbol{W}}$, we need to consider both the direct and indirect effects of changing $\boldsymbol{W}$ on $\boldsymbol{h}_t$ since there are direct and indirect gradient paths at each time step. Since we can unroll the function $\boldsymbol{h}_t = f(f(f(\ldots f(\boldsymbol{h}_1, \boldsymbol{x}_0, \boldsymbol{\theta}), \ldots \boldsymbol{x}_{t-2}, \boldsymbol{\theta}), \boldsymbol{x}_{t-1}, \boldsymbol{\theta}), \boldsymbol{x}_t, \boldsymbol{\theta})$, we write the gradient $\frac{\mathrm{d}\boldsymbol{h}_t}{\mathrm{d}\boldsymbol{W}}$ as

$$\frac{\mathrm{d}\boldsymbol{h}_t}{\mathrm{d}\boldsymbol{W}} = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}} + \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} \frac{\mathrm{d}\boldsymbol{h}_{t-1}}{\mathrm{d}\boldsymbol{W}}, \quad \forall \boldsymbol{W} \in \boldsymbol{\theta}. \tag{3}$$

We need to keep unrolling further because $\boldsymbol{W}$ again affects $\boldsymbol{h}_{t-1}$ through two pathways. Let us define $\boldsymbol{I}_t \doteq \frac{\mathrm{d}\boldsymbol{h}_t}{\mathrm{d}\boldsymbol{W}}$, $\boldsymbol{J}_t \doteq \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}}$, and $\boldsymbol{K}_t \doteq \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}}$. We can write the recursive relationship as follows:

$$\begin{aligned}
\boldsymbol{I}_t &= \boldsymbol{J}_t + \boldsymbol{K}_t \boldsymbol{I}_{t-1} \\
&= \boldsymbol{J}_t + \boldsymbol{K}_t \boldsymbol{J}_{t-1} + \boldsymbol{K}_t \boldsymbol{K}_{t-1} \boldsymbol{I}_{t-2} \\
&= \boldsymbol{J}_t + \boldsymbol{K}_t \boldsymbol{J}_{t-1} + \boldsymbol{K}_t \boldsymbol{K}_{t-1} \boldsymbol{J}_{t-2} + \boldsymbol{K}_t \boldsymbol{K}_{t-1} \boldsymbol{K}_t \boldsymbol{I}_{t-3} \\
&= \sum_{j=1}^{t} \left(\prod_{i=j+1}^{t} \boldsymbol{K}_i \right) \boldsymbol{J}_j + \left(\prod_{i=1}^{t} \boldsymbol{K}_t \right) \boldsymbol{J}_0 \\
&= \sum_{j=1}^{t} \left(\prod_{i=j+1}^{t} \boldsymbol{K}_i \right) \boldsymbol{J}_j.
\end{aligned} \tag{4}$$

assuming that that $\boldsymbol{J}_0 = \boldsymbol{0}$. This relationship is utilized in BPTT by efficiently calculating the summation backward. Calculating this requires storing all previous inputs and states $\boldsymbol{x}_i, \boldsymbol{h}_i, \forall \{1, \ldots, t\}$. In other words, we need to backpropagate the gradient in time, starting from the current time step and going all the way back to the first time step. The computational and memory resource grows linearly with the number of steps because we need to go from the current step to the beginning of time for each update.

A.2. Truncated-Backpropagation Through Time (T-BPTT)

We can simplify the intensive computation needed by BPTT if we truncate the backpropagation process. BPTT starts from time step t and goes all the way to time step 1. Instead, we can go back in time up to time step $t - T$, where $t > T$. This create the Truncated-Backpropagation Through Time (T-BPTT) update (Williams and Zipser 1989):

$$\frac{\mathrm{d}\boldsymbol{h}_t}{\mathrm{d}\boldsymbol{W}} \approx \sum_{j=t-T}^{t} \left(\prod_{i=j+1}^{t} \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{h}_{i-1}} \right) \frac{\partial \boldsymbol{h}_j}{\partial \boldsymbol{W}}, \forall t > T. \tag{5}$$

T-BPTT needs only to store last $T + 1$ inputs and states to approximate the gradient. Truncating the gradient drops any interactions beyond the truncation length, which may make the learner myopic.

A.3. Real-time Recurrent Learning (RTRL)

Instead of relying on the rolled-out equation (Eq. 4 or Eq. 5), we can instead compute $\boldsymbol{I}_t = \frac{\mathrm{d}\boldsymbol{h}_t}{\mathrm{d}\boldsymbol{W}}$, known also as the sensitivity matrix, incrementally using its recursive relationship. This process is known as Real-Time Recurrent Learning (RTRL)

(Williams and Peng 1990) given by

$$\boldsymbol{I}_t = \boldsymbol{J}_t + \boldsymbol{K}_t \boldsymbol{I}_{t-1}. \tag{6}$$

The quantities $\boldsymbol{J}_t$ and $\boldsymbol{K}_t$ use the current input, and they can be computed easily without BPTT but requiring to store $\boldsymbol{I}_{t-1}$ to compute $\boldsymbol{I}_t$ in an incremental fashion. RTRL computes the true gradient without any approximation in contrast to T-BPTT, given that the parameters are fixed.

A.4. Parallelization with Recurrent Learning

The T-BPTT algorithm is highly parallelizable if the truncation length is set to the update-segment length and the scan operation is performed on the input sequence to produce the outputs in parallel. This is what powers modern recurrent-based architectures such as Mamba (Gu and Dao 2024) and Hawk (De et al. 2024). In the IID paradigm, the data streams are split into fixed-length segments, and learning updates are done based on multiple segments are chosen randomly from the dataset in batches. This learning procedure makes recurrent-based networks compute the outputs in parallel, which is identical to the procedure done with the transformer architecture (Vaswani et al. 2017).

Unlike transformer, parallelization in recurrent-based networks is challenging because they have states evolving sequentially over time. Luckily, parallel scan (Blelloch 1990) can be used to parallelize diagonal linear recurrent-based models, which reduces the time complexity of training from $O(L)$ to $O(\log L)$, where L is the segment length, also known as the sequence length. We provide a primer on parallel scan in Appendix J and a primer on linear recurrent units in Appendix I.

Parallelizing RTRL is challenging because the computational cost of using parallel scan with the sensitivity equations is intractable, namely $O(n^3 \log L)$ for sensitivity matrices such as $\boldsymbol{S} \in \mathbb{R}^{n \times n}$. Thus, we only consider parallelization over the batch dimension and limit the segment length to 1.

B. The RTRL Algorithm under multi-stream sequence learning

Algorithm 1 RTRL with multi-stream sequence learning

Require: Number of streams B
Require: Recurrent function $f_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$
Require: Output function $g_{\boldsymbol{\phi}}$ with parameters $\boldsymbol{\phi}$
Require: Data streams $\mathcal{D}_k, \forall k \in \{1, \ldots, B\}$, step size α
Initialize: $\boldsymbol{\theta}, \boldsymbol{\phi}$, hidden state $\mathbf{h}_0 \leftarrow \mathbf{0}$
Initialize: Sensitivity matrix $\boldsymbol{I}_0^{\boldsymbol{W}} \leftarrow \mathbf{0}, \forall \boldsymbol{W} \in \boldsymbol{\theta}$
for $t = 1 \ldots \infty$ **do**
  $\delta_{\boldsymbol{V}} \leftarrow \mathbf{0}, \forall \boldsymbol{V} \in \boldsymbol{\phi}$, $\delta_{\boldsymbol{W}} \leftarrow \mathbf{0}, \forall \boldsymbol{V} \in \boldsymbol{\theta}$
  for $k = 1 \ldots B$ **do**
    $\boldsymbol{x}_t, \boldsymbol{y}_t, \texttt{reset} \leftarrow \mathcal{D}_{k,t}$
    if `reset` **then**
      $\boldsymbol{h}_{t-1} \leftarrow \mathbf{0}$
    end if
    $\mathbf{h}_t \leftarrow f_\theta(\mathbf{h}_{t-1}, \mathbf{x}_t)$
    Update sensitivity: $\boldsymbol{I}_t^{\boldsymbol{W}} \leftarrow \frac{\partial \mathbf{h}_t}{\partial \boldsymbol{W}} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \boldsymbol{I}_{t-1}^{\boldsymbol{W}}, \quad \forall \boldsymbol{W} \in \boldsymbol{\theta}$
    $\hat{\boldsymbol{y}}_t \leftarrow g_{\boldsymbol{\phi}}(\mathbf{h}_t, \mathbf{x}_t)$
    Compute $\mathcal{L}_t = \mathcal{L}(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t)$
    $\delta_{\boldsymbol{V}} \leftarrow \delta_{\boldsymbol{V}} + \frac{1}{B} \left(\frac{\partial \hat{\mathbf{y}}_t}{\partial \boldsymbol{V}} \right)^\top \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t}, \quad \forall \boldsymbol{V} \in \boldsymbol{\phi}$
    $\delta_{\boldsymbol{W}} \leftarrow \delta_{\boldsymbol{W}} + \frac{1}{B} \left(\boldsymbol{I}_t^{\boldsymbol{W}} \right)^\top \left(\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \right)^\top \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t}, \quad \forall \boldsymbol{W} \in \boldsymbol{\theta}$
  end for
  $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \, \delta_{\boldsymbol{W}}, \quad \forall \boldsymbol{W} \in \boldsymbol{\theta}$
  $\boldsymbol{V} \leftarrow \boldsymbol{V} - \alpha \, \delta_{\boldsymbol{V}}, \quad \forall \boldsymbol{V} \in \boldsymbol{\phi}$
end for

C. The T-BPTT Algorithm under multi-stream sequence learning

Algorithm 2 Truncated BPTT with multi-stream sequence learning

Require: Number of Streams B, Learning update stride T
Require: Recurrent function $f_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$
Require: Output function $g_{\boldsymbol{\phi}}$ with parameters $\boldsymbol{\phi}$
Require: Data streams $\mathcal{D}_k, \forall k \in \{1, \ldots, B\}$, step size α
Initialize: $\boldsymbol{\theta}, \boldsymbol{\phi}$, hidden state $\mathbf{h}_0 \leftarrow \mathbf{0}$
for $t_u = 1 \ldots \infty$ **do**
  for $k = 1 \ldots B$ **do**
    $\delta_{\boldsymbol{W}} \leftarrow \mathbf{0}, \forall \boldsymbol{W} \in \boldsymbol{\theta}, \quad \delta_{\boldsymbol{V}} \leftarrow \mathbf{0}, \forall \boldsymbol{V} \in \boldsymbol{\phi}, \quad \delta_{\boldsymbol{h}} \leftarrow \mathbf{0}$
    $\mathcal{B}_k \leftarrow \emptyset$
    for $t = 1$ **upto** T **do**
      $\boldsymbol{x}_t, \boldsymbol{y}_t, \texttt{reset} \leftarrow \mathcal{D}_k$
      if `reset` **then**
        $\boldsymbol{h}_{t-1} \leftarrow \mathbf{0}$
      end if
      $\mathbf{h}_t \leftarrow f_{\boldsymbol{\theta}}(\mathbf{h}_{t-1}, \boldsymbol{x}_t)$
      $\hat{\boldsymbol{y}}_t \leftarrow g_{\boldsymbol{\phi}}(\mathbf{h}_t, \boldsymbol{x}_t)$
      Compute $\mathcal{L}_t = \mathcal{L}(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t)$
      $\delta_{\boldsymbol{V}} \leftarrow \delta_{\boldsymbol{V}} + \left(\frac{\partial \hat{\boldsymbol{y}}_t}{\partial \boldsymbol{V}}\right)^\top \frac{\partial \mathcal{L}_t}{\partial \hat{\boldsymbol{y}}_t}, \quad \boldsymbol{V} \in \boldsymbol{\phi}$
      $\delta_{\mathbf{h}} \leftarrow \delta_{\mathbf{h}} + \left(\frac{\partial \hat{\boldsymbol{y}}_t}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}_t}{\partial \hat{\boldsymbol{y}}_t}$
      Append $(\mathbf{h}_{t-1}, \boldsymbol{x}_t, \mathbf{h}_t)$ to $\mathcal{B}_k$
    end for
    for $i = t$ **downto** 1 **do**
      Retrieve $(\mathbf{h}_{i-1}, \boldsymbol{x}_i, \mathbf{h}_i)$ from $\mathcal{B}_k$
      $\delta_{\boldsymbol{W}} \leftarrow \delta_{\boldsymbol{W}} + \left(\frac{\partial \mathbf{h}_i}{\partial \boldsymbol{W}}\right)^\top \delta_{\mathbf{h}} \cdot, \quad \boldsymbol{W} \in \boldsymbol{\theta}$
      $\delta_{\mathbf{h}} \leftarrow \left(\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}\right)^\top \delta_{\mathbf{h}}$
    end for
  end for
  $\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{1}{BT} \alpha \, \delta_{\boldsymbol{W}}, \quad \boldsymbol{W} \in \boldsymbol{\theta}$
  $\boldsymbol{V} \leftarrow \boldsymbol{V} - \frac{1}{BT} \alpha \, \delta_{\boldsymbol{V}}, \quad \boldsymbol{V} \in \boldsymbol{\phi}$
end for

D. Related Works

Streaming learning. Standard deep learning methods often assume access to the entire dataset; however, real-world applications require continuous data streams. There are a few supervised deep learning methods that work under the streaming learning setting (Hayes et al. 2019, Saran et al. 2023, Hayes and Kanan 2022), and additional efforts have adapted reinforcement learning (Elsayed et al. 2024, Vasan et al. 2024), language models (Goyal et al. 2009), and video predictors (Carreira et al. 2024, Qian et al. 2024, Han et al. 2025) to work under this setting. In our paper, we consider the multi-stream setting where there are more than one stream to process in parallel. Future work is needed to make Memora work with a single stream and a single update stride.

Attention-free models. To avoid the quadratic training cost and linear inference cost of attention on long histories, recent methods employ linear recurrence for fixed-size state memory, achieving linear training cost and constant inference cost. Mamba1 (Gu and Dao 2024), an approach that combines state-space approach (e.g., Gu et al. 2021) with temporal convolution (Bai et al. 2018), was the first method to be used in large-scale systems (Lieber et al. 2024) followed by Mamba2 (Dao and Gu 2024) and Hawk (De et al. 2024) that was based on the LRU recurrent unit (Orvieto et al. 2023). These architectures, however, still use IID chunking. In contrast, Memora is purely recurrent, preserving stream continuity to rival Transformers on long-sequence tasks.

Byte-level data modeling. Byte-level sequence modeling offers a flexible, domain-agnostic approach across text, audio, and genomics by operating directly on raw bytes, removing the need for preprocessing or tokenization. This can improve generalization and robustness to morphological variations like typos or character-level reasoning (Xue et al. 2022). However, it also introduces challenges due to longer, noisier, and less structured sequences. While recent works have improved byte-level modeling through architectural innovations (Wang et al. 2024a, Yu et al. 2023), many still rely on static chunking and overlook the temporal dynamics of streaming data. In contrast, our approach enhances byte-level modeling by processing streams in their natural order, improving memory and temporal reasoning capabilities critical for long, unsegmented byte sequences.

E. Complex-valued Gated Linear Recurrent Unit (Complex GLRU)

Here, we describe the complex-valued GLRU unit. We replace our real-valued recurrent gating and input gating with their complex counterparts.

$$\begin{aligned}\boldsymbol{h}_t &= \boldsymbol{\lambda}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma}_t \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t)\\ \boldsymbol{y}_t &= \Re[\boldsymbol{C}\boldsymbol{h}_t],\end{aligned}$$

where $\boldsymbol{\lambda}(\boldsymbol{x}_t) \doteq \boldsymbol{r}(\boldsymbol{x}_t) \circ e^{i\boldsymbol{\theta}_t}$, $\boldsymbol{\gamma}_t = \sqrt{1-|\boldsymbol{\lambda}_t|}$, $\boldsymbol{g}(\boldsymbol{x}_t) = \boldsymbol{G}\boldsymbol{x}_t$, and $\boldsymbol{r}(\boldsymbol{x}_t) = e^{-ce^{\boldsymbol{\nu}} \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t)}$. The vector $\boldsymbol{\theta}_t \in \mathbb{R}^n$ contains the phase information of the complex-valued system. We need to learn complex-valued matrices, $\boldsymbol{B} \in \mathbb{C}^{n\times d}$, $\boldsymbol{G} \in \mathbb{C}^{n\times d}$, and $\boldsymbol{C} \in \mathbb{C}^{m\times n}$. Note that $\boldsymbol{R} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{\nu} \in \mathbb{R}^n$ are still real-valued. We have to use $\boldsymbol{y}_t = \Re[\boldsymbol{C}\boldsymbol{h}_t]$ instead of $\boldsymbol{y}_t = \Re[\boldsymbol{h}_t]$ to have no gradient bias. We refer the reader to (Elelimy et al. 2024) and (Orvieto et al. 2023) for more discussion about gradient bias. We implement complex-valued GLRU using the cosine representation with real-valued systems, and we refer the reader to Appendix I.4 for the details on how to convert the system from exponential representation form to cosine representation.

F. RTRL mode of GLRU and RG-LRU recurrent units

Here, we derive the RTRL sensitivity update equations for GLRU and RG-LRU units. We show that in both, their sensitivity tensors are diagonal and can be stored and computed efficiently.

F.1. GLRU RTRL Sensitivity Equations

The recurrence equation of GLRU is given by

$$\begin{aligned}\boldsymbol{h}_t &= \boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma}_t \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t)\\ \boldsymbol{y}_t &= \boldsymbol{h}_t\end{aligned}$$

where $\boldsymbol{\gamma}_t = \sqrt{1-\boldsymbol{r}_t^2}$, $\boldsymbol{g}(\boldsymbol{x}_t) = \boldsymbol{G}\boldsymbol{x}_t$, and $\boldsymbol{r}(\boldsymbol{x}_t) = e^{-ce^{\boldsymbol{\nu}} \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t)}$. Now, we can derive the the sensitivity update equation for the vector $\boldsymbol{\nu}$ using index notations as follows:

$$\begin{aligned}S^{\boldsymbol{\nu}}_{t,i,j} &= \frac{\partial h_{t,i}}{\partial \nu_j} = \frac{\partial}{\partial \nu_j}\left(r_i h_{t-1,i} + \gamma_i g_i \left(\sum_m B_{i,m} x_{t,m}\right)\right)\\ &= \frac{\partial r_i}{\partial \nu_j} h_{t-1,i} + r_i S^{\boldsymbol{\nu}}_{t-1,i,j} + \frac{\partial \gamma_i}{\partial \nu_j} g_i \left(\sum_m B_{i,m} x_{t,m}\right)\\ &= -c\delta_{i,j} e^{\nu_i} \sigma\left(\sum_k R_{i,m} x_{t,m}\right) r_i h_{t-1,i} + r_i S^{\boldsymbol{\nu}}_{t-1,i,j}\\ &+ c\delta_{i,j} \frac{r_i^2}{\gamma_i} e^{\nu_i} \sigma\left(\sum_k R_{i,m} x_{t,m}\right) g_i \left(\sum_m B_{i,m} x_{t,m}\right)\end{aligned}$$

Note how the structure coming from $\delta_{i,j}$ forces all off-diagonal elements where $i \neq j$ to be zero. Hence, we can store the sensitivity elements in a vector instead of a matrix.

We can also write the recursive relationship using the reduced sensitivity for $\boldsymbol{\nu}$ as follows:

$$\begin{aligned}\mathbf{S}_t^{\boldsymbol{\nu}} &= \frac{\partial}{\partial \boldsymbol{\nu}} \left(\boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma} \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t)\right) \\ &= \boldsymbol{r}(\boldsymbol{x}_t) \circ \mathbf{S}_{t-1}^{\boldsymbol{\nu}} - ce^{\boldsymbol{\nu}} \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t) \circ \boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + c\frac{\boldsymbol{r}(\boldsymbol{x}_t)^2}{\boldsymbol{\gamma}} \circ e^{\boldsymbol{\nu}} \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t) \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t)\end{aligned}$$

where $[\mathbf{S}_t^{\boldsymbol{\nu}}]_i \doteq \left[S_{t,i,j}^{\boldsymbol{\nu}}\right]_{i,j=1}$.

Next, we derive the sensitivity equation for the matrix $\boldsymbol{B}$ as follows:

$$\begin{aligned}S_{t,i,j,k}^{\boldsymbol{G}} = \frac{\partial h_{t,i}}{\partial G_{j,k}} &= \frac{\partial}{\partial G_{j,k}} \left(r_i h_{t-1,i} + \gamma_i b_i \sum_m G_{i,m} x_{t,m}\right) \\ &= r_i S_{t-1,i,j,k}^{\boldsymbol{G}} \delta_{i,j} + \gamma_i b_i \delta_{i,j} x_{t,k},\end{aligned}$$

where we use $b_i = \sum_k B_{i,k} x_{t,k}$. Note how the structure coming from $\delta_{i,j}$ forces all off-diagonal elements where $i \neq j$ to be zero. Hence, we can store the sensitivity elements in a matrix instead of a 3-tensor. We can also write the recursive relationship using the reduced sensitivity objects as follows:

$$\begin{aligned}\mathbf{S}_t^{\boldsymbol{G}} &= \frac{\partial}{\partial \boldsymbol{G}} \left(\boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma} \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{G}\boldsymbol{x}_t)\right) \\ &= \mathrm{Diag}(\boldsymbol{r}(\boldsymbol{x}_t)) \circ \mathbf{S}_{t-1}^{\boldsymbol{G}} + (\boldsymbol{\gamma} \circ \boldsymbol{B}\boldsymbol{x}_t)\boldsymbol{x}_t^\top,\end{aligned}$$

where $\left[\mathbf{S}_t^{\boldsymbol{G}}\right]_{i,j} \doteq \left[S_{t,i,j,k}^{\boldsymbol{G}}\right]_{i,j,k=1}$.

Next, we derive the sensitivity equation for the matrix $\boldsymbol{B}$ as follows:

$$\begin{aligned}S_{t,i,j,k}^{\boldsymbol{B}} = \frac{\partial h_{t,i}}{\partial B_{j,k}} &= \frac{\partial}{\partial B_{j,k}} \left(r_i h_{t-1,i} + \gamma_i g_i \sum_m B_{i,m} x_{t,m}\right) \\ &= r_i S_{t-1,i,j,k}^{\boldsymbol{B}} \delta_{i,j} + \gamma_i g_i \sum_m \delta_{i,j} \delta_{k,m} x_{t,m} \\ &= r_i S_{t-1,i,j,k}^{\boldsymbol{B}} \delta_{i,j} + \gamma_i g_i \delta_{i,j} x_{t,k}.\end{aligned}$$

Note how the structure coming from $\delta_{i,j}$ forces all off-diagonal elements where $i \neq j$ to be zero. Hence, we can store the sensitivity elements in a matrix instead of a 3-tensor. We can also write the recursive relationship using the reduced sensitivity objects as follows:

$$\begin{aligned}\mathbf{S}_t^{\boldsymbol{B}} &= \frac{\partial}{\partial \boldsymbol{B}} \left(\boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma} \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t)\right) \\ &= \mathrm{Diag}(\boldsymbol{r}(\boldsymbol{x}_t)) \circ \mathbf{S}_{t-1}^{\boldsymbol{B}} + (\boldsymbol{\gamma} \circ \boldsymbol{g}(\boldsymbol{x}_t))\boldsymbol{x}_t^\top,\end{aligned}$$

where $\left[\mathbf{S}_t^{\boldsymbol{B}}\right]_{i,j} \doteq \left[S_{t,i,j,k}^{\boldsymbol{B}}\right]_{i,j,k=1}$.

Finally, we derive the sensitivity equation for the matrix $\boldsymbol{R}$ as follows:

$$\begin{aligned}S_{t,i,j,k}^{\boldsymbol{R}} = \frac{\partial h_{t,i}}{\partial R_{j,k}} &= \frac{\partial}{\partial R_{j,k}} \left(r_i h_{t-1,i} + \gamma_i g_i \sum_m B_{i,m} x_{t,m}\right) \\ &= \delta_{i,j} r_i S_{t-1,i,j}^{\boldsymbol{R}} + \frac{\partial r_i}{\partial R_{j,k}} h_{t-1,i} \\ &= \delta_{i,j} r_i S_{t-1,i,j}^{\boldsymbol{R}} - c\delta_{i,j} r_i e^{\nu_i} \sigma\left(\sum_k R_{i,m} x_{t,m}\right)\left(1 - \sigma'\left(\sum_k R_{i,m} x_{t,m}\right)\right) h_{t-1,i} x_{t,k} \\ &- c\delta_{i,j} r_i e^{\nu_i} \sigma\left(\sum_k R_{i,m} x_{t,m}\right)\left(1 - \sigma'\left(\sum_k R_{i,m} x_{t,m}\right)\right) \frac{r_i}{\gamma_i} g_i \left(\sum_m B_{i,m} x_{t,m}\right) x_{t,k}\end{aligned}$$

$$= \delta_{i,j} r_i S^{\boldsymbol{R}}_{t-1,i,j} - \delta_{i,j} d_i \left(h_{t-1,i} - \frac{r_i}{\gamma_i} g_i \sum_m B_{i,m} x_{t,m} \right) x_{t,k},$$

where $d_i = c r_i e^{\nu_i} \sigma\left(\sum_k R_{i,m} x_{t,m}\right)\left(1 - \sigma'\left(\sum_k R_{i,m} x_{t,m}\right)\right)$. Note how the structure coming from $\delta_{i,j}$ forces all off-diagonal elements where $i \neq j$ to be zero. Hence, we can store the sensitivity elements in a matrix instead of a 3-tensor. We can also write the recursive relationship using the reduced sensitivity objects as follows:

$$\begin{aligned}
\mathbf{S}^{\boldsymbol{R}}_t &= \frac{\partial}{\partial \boldsymbol{R}} \left(\boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma} \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t) \right) \\
&= \mathrm{Diag}(\boldsymbol{r}(\boldsymbol{x}_t)) \boldsymbol{S}^{\boldsymbol{R}}_r + \left(\boldsymbol{d} \circ \left(\boldsymbol{h}_{t-1} - \frac{\boldsymbol{r}(\boldsymbol{x}_t)}{\boldsymbol{\gamma}} \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t) \right) \right) \boldsymbol{x}_t^\top.
\end{aligned}$$

where $\boldsymbol{d} = c\boldsymbol{r}(\boldsymbol{x}_t) \circ e^{\boldsymbol{\nu}} \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t) \circ (1 - \sigma'(\boldsymbol{R}\boldsymbol{x}_t))$ and $\left[\mathbf{S}^{\boldsymbol{R}}_t\right]_{i,j} \doteq \left[S^{\boldsymbol{R}}_{t,i,j,k}\right]_{i,j,k=1}$.

F.2. RG-LRU RTRL Sensitivity Equations

The RG-LRU unit was introduced by (De et al. 2024) and is typically used with T-BPTT. Here, we derive its RTRL mode. We start by writing the RG-LRU unit in the same notation we use in this paper. The RG-LRU unit is given by

$$\begin{aligned}
\boldsymbol{h}_t &= \boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma}_t \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ \boldsymbol{x}_t \\
\boldsymbol{y}_t &= \boldsymbol{h}_t
\end{aligned}$$

where $\boldsymbol{\gamma}_t = \sqrt{1 - \boldsymbol{r}_t^2}, \boldsymbol{g}(\boldsymbol{x}_t) = \sigma(\boldsymbol{G}\boldsymbol{x}_t), \boldsymbol{r}(\boldsymbol{x}_t) = e^{-c \log(1+e^{\boldsymbol{\nu}}) \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t)}$

The sensitivity update equation for the vector $\boldsymbol{\nu}$ are given by:

$$\begin{aligned}
S^{\boldsymbol{\nu}}_{t,i,j} &= \frac{\partial h_{t,i}}{\partial \nu_j} = \frac{\partial}{\partial \nu_j} \left(r_i h_{t-1,i} + \gamma_i g_i \left(\sum_m B_{i,m} x_{t,m} \right) \right) \\
&= \frac{\partial r_i}{\partial \nu_j} h_{t-1,i} + r_i S^{\boldsymbol{\nu}}_{t-1,i,j} + \frac{\partial \gamma_i}{\partial \nu_j} g_i \left(\sum_m B_{i,m} x_{t,m} \right) \\
&= -c\delta_{i,j} \sigma(\nu_i) \sigma\left(\sum_k R_{i,m} x_{t,m} \right) r_i h_{t-1,i} + r_i S^{\boldsymbol{\nu}}_{t-1,i,j} \\
&+ c\delta_{i,j} \frac{r_i^2}{\gamma_i} \sigma(\nu_i) \sigma\left(\sum_k R_{i,m} x_{t,m} \right) g_i x_{t,i}
\end{aligned}$$

Note how the structure coming from $\delta_{i,j}$ forces all off-diagonal elements where $i \neq j$ to be zero. Hence, we can store the sensitivity elements in a vector instead of a matrix.

We can also write the recursive relationship using the reduced sensitivity for $\boldsymbol{\nu}$ as follows:

$$\begin{aligned}
\mathbf{S}^{\boldsymbol{\nu}}_t &= \frac{\partial}{\partial \boldsymbol{\nu}} \left(\boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma} \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ \boldsymbol{x}_t \right) \\
&= \boldsymbol{r}(\boldsymbol{x}_t) \circ \mathbf{S}^{\boldsymbol{\nu}}_{t-1} - c\sigma(\boldsymbol{\nu}) \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t) \circ \boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + c\frac{\boldsymbol{r}(\boldsymbol{x}_t)^2}{\boldsymbol{\gamma}} \circ \sigma(\boldsymbol{\nu}) \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t) \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ \boldsymbol{x}_t,
\end{aligned}$$

where $[\mathbf{S}^{\boldsymbol{\nu}}_t]_i \doteq \left[S^{\boldsymbol{\nu}}_{t,i,j}\right]_{i,j=1}$.

Next, we derive the sensitivity equation for the matrix $\boldsymbol{G}$ as follows:

$$\begin{aligned}
S^{\boldsymbol{G}}_{t,i,j,k} &= \frac{\partial h_{t,i}}{\partial G_{j,k}} = \frac{\partial}{\partial G_{j,k}} \left(r_i h_{t-1,i} + \gamma_i x_i \sum_m G_{i,m} x_{t,m} \right) \\
&= r_i S^{\boldsymbol{G}}_{t-1,i,j,k} \delta_{i,j} + \gamma_i x_i \delta_{i,j} x_{t,k},
\end{aligned}$$

Note how the structure coming from $\delta_{i,j}$ forces all off-diagonal elements where $i \neq j$ to be zero. Hence, we can store the sensitivity elements in a matrix instead of a 3-tensor. We can also write the recursive relationship using the reduced sensitivity objects as follows:

$$\begin{aligned}\mathbf{S}_t^{\boldsymbol{G}} &= \frac{\partial}{\partial \boldsymbol{G}} \left(\boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma} \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ \boldsymbol{x}_t\right) \\ &= \mathrm{Diag}(\boldsymbol{r}(\boldsymbol{x}_t)) \circ \mathbf{S}_{t-1}^{\boldsymbol{G}} + (\boldsymbol{\gamma} \circ \boldsymbol{x}_t)\boldsymbol{x}_t^\top.\end{aligned}$$

where $\left[\mathbf{S}_t^{\boldsymbol{G}}\right]_{i,j} \doteq \left[S_{t,i,j,k}^{\boldsymbol{G}}\right]_{i,j,k=1}$.

Finally, we derive the sensitivity equation for the matrix $\boldsymbol{R}$ as follows:

$$\begin{aligned}S_{t,i,j,k}^{\boldsymbol{R}} &= \frac{\partial h_{t,i}}{\partial R_{j,k}} = \frac{\partial}{\partial R_{j,k}} \left(r_i h_{t-1,i} + \gamma_i g_i x_{t,i}\right) \\ &= \delta_{i,j} r_i S_{t-1,i,j}^{\boldsymbol{R}} + \frac{\partial r_i}{\partial R_{j,k}} h_{t-1,i} \\ &= \delta_{i,j} r_i S_{t-1,i,j}^{\boldsymbol{R}} - c\delta_{i,j} r_i \log(1+e^{\nu_i})\sigma\left(\sum_k R_{i,m}x_{t,m}\right)\left(1-\sigma'\left(\sum_k R_{i,m}x_{t,m}\right)\right) h_{t-1,i}x_{t,k} \\ &- c\delta_{i,j} r_i \log(1+e^{\nu_i})\sigma\left(\sum_k R_{i,m}x_{t,m}\right)\left(1-\sigma'\left(\sum_k R_{i,m}x_{t,m}\right)\right) \frac{r_i}{\gamma_i} g_i x_{t,i} x_{t,k} \\ &= \delta_{i,j} r_i S_{t-1,i,j}^{\boldsymbol{R}} - \delta_{i,j} d_i \left(h_{t-1,i} - \frac{r_i}{\gamma_i} g_i x_{t,i}\right) x_{t,k},\end{aligned}$$

where $d_i = cr_i \log(1+e^{\nu_i})\sigma\left(\sum_k R_{i,m}x_{t,m}\right)\left(1-\sigma'\left(\sum_k R_{i,m}x_{t,m}\right)\right)$. Note how the structure coming from $\delta_{i,j}$ forces all off-diagonal elements where $i \neq j$ to be zero. Hence, we can store the sensitivity elements in a matrix instead of a 3-tensor. We can also write the recursive relationship using the reduced sensitivity objects as follows:

$$\begin{aligned}\mathbf{S}_t^{\boldsymbol{R}} &= \frac{\partial}{\partial \boldsymbol{R}} \left(\boldsymbol{r}(\boldsymbol{x}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{\gamma} \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ (\boldsymbol{B}\boldsymbol{x}_t)\right) \\ &= \mathrm{Diag}(\boldsymbol{r}(\boldsymbol{x}_t))\boldsymbol{S}_r^{\boldsymbol{R}} + \left(\boldsymbol{d} \circ \left(\boldsymbol{h}_{t-1} - \frac{\boldsymbol{r}(\boldsymbol{x}_t)}{\boldsymbol{\gamma}} \circ \boldsymbol{g}(\boldsymbol{x}_t) \circ \boldsymbol{x}_t\right)\right)\boldsymbol{x}_t^\top.\end{aligned}$$

where $\boldsymbol{d} = c\boldsymbol{r}(\boldsymbol{x}_t) \circ \log(1+e^{\boldsymbol{\nu}}) \circ \sigma(\boldsymbol{R}\boldsymbol{x}_t) \circ (1-\sigma'(\boldsymbol{R}\boldsymbol{x}_t))$ and $\left[\mathbf{S}_t^{\boldsymbol{R}}\right]_{i,j} \doteq \left[S_{t,i,j,k}^{\boldsymbol{R}}\right]_{i,j,k=1}$.

G. Additionl experiment: Considering GLRU alternatives

Here, we conduct a study to assess the importance of our GLRU unit on a language modeling task using the TinyStories dataset with models of approximately 50M parameters. Since GLRU is a real-valued recurrent unit, we introduce a complex-valued variant by incorporating complex-valued gating in the recurrence, referred to as complex GLRU. We also compare against complex LRU (Orvieto et al. 2023) and its real-valued counterpart, real LRU, as well as MinGRU (Feng et al. 2024). We also evaluate restricted variants of complex GLRU and complex LRU, where the complex values are constrained as conjugate pairs (see Appendix E and I.5 for more details).

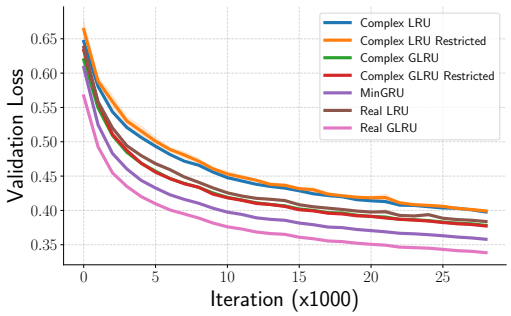


Figure 7. Performance comparison of Memora with different recurrent units.

Figure 7 presents validation performance across all baselines. Our model, real GLRU, consistently outperforms both gated (e.g., MinGRU) and non-gated (e.g., LRU) alternatives. We also find that constraining complex values to be conjugate pairs offers no performance benefit. Our findings align with observations by (Gu and Dao 2024), where real-valued gated units surpass complex-valued ones on language tasks, further validating our results in this setting.

H. Experimental Details

We use Python and Pytorch (Paszke et al. 2019) to implement our algorithms using automatic differentiation to backpropagate gradients with T-BPTT and RTRL. Additionally, we used the parallel scan implementation by (Kyrylov 2024).

We used LeCun initialization (LeCun et al. 2012) to initialize all weights except for the weights used for contracting the input (see Figure 2), which we initialize it $W_{i,j} \sim \mathcal{N}(0, 1/\sqrt{2 \times E \times N})$, where N is the number of layers in the model. Additionally, we use the ring initialization (Orvieto et al. 2023) in both RG-LRU and GLRU, using $r_{\text{min}} = 0.9$, $r_{\text{max}} = 0.999$, which is given as: $\boldsymbol{\nu}_{\text{init}} \leftarrow \log(-0.5 \log(\boldsymbol{u}(r_{\text{max}}^2 - r_{\text{min}}^2) + r_{\text{min}}^2)$, where $u_i \sim U[0,1], \forall i$.

In Figure 3, we list the common training configurations we used in all experiments except for the streaming learning experiment where we use RMSProp with $\beta_2 = 0.999$ and no weight decay. We then describe the specific details for each experiment in the next sections.

Configuration	Value
Optimizer	AdamW
Optimizer parameters	$\beta_1 = 0.9, \beta_2 = 0.95$
Weight decay	0.1
Bias	No
Dropout	No
Gradient Clipping	1.0
Floating-point precision	Bfloat16
GPU used	NVIDIA L40
Automatic mixed precision	Yes
Embedding Weight Tying (Press and Wolf 2016)	Yes

Table 3. The common training configuration shared in all experiments and baselines.

H.1. Selective Copying

We trained all models for $50,000$ iterations using a batch size of 64 and a constant learning rate of 3×10^{-4}, evaluating performance on $5,000$ randomly generated examples. Each model consists of two layers. The Memora variants—GLRU, MinGRU, and LRU—share the same configuration: a model dimension of 64, state dimension of 256, and a gated MLP with an expansion ratio of 3. Hawk uses a similar setup to Memora, with the addition of a convolution kernel size of 4. Llama2, by contrast, is configured with a larger model dimension of 192, three attention heads, and an MLP expansion ratio of 4. Mamba also adopts a model dimension of 192 but differs with a state dimension of 128, a head dimension of 64, an MLP expansion ratio of 2, and a convolution kernel size of 4. We report the average of 5 independent runs and report the standard error.

H.2. Byte-level language modeling with TinyStories

We use an effective batch size of $BT = 131{,}072$ and vary the sequence length in the IID setting or the update stride in multi-stream setting with values $T \in \{2, 8, 32, 128, 512, 1024\}$. All models are trained for 2 epochs using a constant learning rate of 3×10^{-4}, and are configured to have approximately 60 million parameters. The Mamba model uses a model dimension of 768, a state dimension of 128, a head dimension of 128, 16 layers, a convolutional kernel size of 4, and an MLP expansion ratio of 2, totaling $60{,}294{,}464$ parameters. The Hawk model is configured with a model dimension of 512, a state dimension of 768, 14 layers, a gated MLP expansion ratio of 3, and a convolutional kernel size of 4, totaling $66,259,968$ parameters. The Memora model uses a model dimension of 512, a state dimension of 768, 14 layers, and an MLP expansion ratio of 3, totaling $60{,}711{,}424$ parameters. Finally, the Llama2 baseline has a model dimension of 512, 8 attention heads, 18 layers, and an MLP expansion ratio of 4, totaling $61{,}491{,}712$ parameters. We report the average of 3 runs (the error bars are very small so we do not display it to reduce clutter and enhance visibility).

H.3. DNA modeling

We use an effective batch size of $BT = 524,288$ with a batch size of $B = 512$ and trained for 4 epochs. We used a learning rate warm-up for 10% of the total iterations followed by cosine annealing with a minimum of 10^{-5} and a maximum of 10^{-3}.

We compare Memora, Mamba, Hawk, and Llama2, each of which use 5 different model sizes in each method (sizes with prefixes S0, S1, S2, S3, S4). We list the model configurations in Table 4.

In Mamba2 and Hawk, the temporal convolutional kernel size is 4. In Hawk and Memora, we use MLP expansion factor of 3 in the gated MLP blocks (by setting $E = 3 \times M$ in Figure 2). In Mamba2, the expansion factor is set to 2. In Llama2, the MlP expansion factor is set to 4. We report the average of 3 runs (the error bars are very small to be visible in the figure).

H.4. Byte-level language modeling with FineWebEdu

We use an effective batch size of $BT = 262,144$ with a batch size of $B = 512$ and trained for $100,000$ iterations. We used a learning rate warm-up for 10% of the total iterations followed by cosine annealing with a minimum of 10^{-5} and a maximum of 10^{-3}. We compare Memora, Mamba, Hawk, and Llama2, each of which use 5 different model sizes in each method (sizes with prefixes S2, S3, S4, S5). We list the model configurations in Table 4. We use the same other model parameters mentioned in the previous section. We report the average of 2 runs.

H.5. Learning from one sample at a time with RTRL and 1-step BPTT

For both Memora and Hawk, we use a model dimension of 384, a state dimension of 512, six layers, and a gated MLP expansion factor of 3. Training is conducted over 1024 parallel data streams with a single update stride for two epochs. We employ a constant learning rate and tune each method by selecting the best learning rate from the set $\{3 \times 10^{-4}, 3 \times 10^{-5}, 3 \times 10^{-6}, 3 \times 10^{-7}, 3 \times 10^{-8}\}$. The optimal learning rate was found to be 3×10^{-5} for Memora (both RTRL and 1-BPTT variants) and 3×10^{-7} for Hawk (for both RTRL and 1-BPTT). We report the average of 3 runs.

H.6. Memora with GLRU alternatives

We use an effective batch size of $BT = 131{,}072$, with a batch size of $B = 256$. We train for 2 epochs using a constant learning rate of 3×10^{-4}. We use a model dimension of 512, a state dimension of 768, 12 layers, and an MLP expansion factor of 3. We report the average of 3 runs.

H.7. Memora with streaming learning

We use an effective batch size of $BT = T = B = 1$ and train for 15M iterations with a constant learning rate of 3×10^{-5} for RTRL and 3×10^{-6} for 1-step BPTT which was selected based on searching in $\{3 \times 10^{-4}, 3 \times 10^{-5}, 3 \times 10^{-6}, 3 \times 10^{-7}\}$. We used a model dimension of 512, a state dimension of 768, a gated MLP expansion factor of 3, and 6 Memora layers. We used RMSProp with $\beta_2 = 0.999$ with no weight decay.

I. Primer on Linear Recurrent Units

The learner usually observes the environment partially; thus, it is required to construct its *learner state*, some internal representation of what the state of the environment might be. We denote the state construction function $f : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^n$ given by $\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)$, where $\boldsymbol{h}_t \in \mathbb{R}^n$ and $\boldsymbol{x}_t \in \mathbb{R}^d$ are the learner state and observation at time step t. The learner state is considered the learner's best ability to construct a compact history of the past. The output construction function $g : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^m$ maps the learner state into some usable output $\boldsymbol{y}_t \in \mathbb{R}^m$ for prediction and is given by $\boldsymbol{y}_t = g(\boldsymbol{h}_t, \boldsymbol{x}_t)$. The evolution of the system is fully described using the following:

$$\begin{aligned} \boldsymbol{h}_t &= f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t), \\ \boldsymbol{y}_t &= g(\boldsymbol{h}_t, \boldsymbol{x}_t). \end{aligned}$$

In the simple case of a linear system, the equations can be formulated as

$$\begin{aligned} \boldsymbol{h}_t &= \boldsymbol{A}\boldsymbol{h}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t \\ \boldsymbol{y}_t &= \boldsymbol{C}\boldsymbol{h}_t + \boldsymbol{D}\boldsymbol{x}_t, \end{aligned}$$

where $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, $\boldsymbol{B} \in \mathbb{R}^{n \times d}$, $\boldsymbol{C} \in \mathbb{R}^{n \times m}$, and $\boldsymbol{D} \in \mathbb{R}^{d \times m}$.

We can write the square matrix $\boldsymbol{A}$ using its eigenvalue decomposition as $\boldsymbol{A} = \boldsymbol{P}^{-1}\boldsymbol{\Lambda}\boldsymbol{P}$, where $\boldsymbol{P} \in \mathbb{C}^{n \times n}$ contains the eigenvectors and $\boldsymbol{\Lambda} \in \mathbb{C}^{n \times n}$ is a diagonal matrix containing the corresponding eigenvalues. (Orvieto et al. 2023) showed

Model	Params	Model Dim	State Dim	Heads	Head Dim	Layers (N)
Memora-S5	111,114,240	768	1024	–	–	12
Memora-S4	52,057,088	512	768	–	–	12
Memora-S3	13,966,848	384	512	–	–	6
Memora-S2	6,495,232	256	384	–	–	6
Memora-S1	3,470,400	192	256	–	–	6
Memora-S0	1,625,600	128	192	–	–	6
Hawk-S5	117,455,616	768	1024	–	–	12
Hawk-S4	56,813,056	512	768	–	–	12
Hawk-S3	14,765,952	384	512	–	–	6
Hawk-S2	7,094,528	256	384	–	–	6
Hawk-S1	3,673,344	192	256	–	–	6
Hawk-S0	1,777,792	128	192	–	–	6
Llama2-S5	85,150,464	768	–	12	–	12
Llama2-S4	41,038,336	512	–	8	–	12
Llama2-S3	10,720,128	384	–	6	–	6
Llama2-S2	5,116,928	256	–	4	–	6
Llama2-S1	2,658,048	192	–	4	–	6
Llama2-S0	1,575,424	128	–	4	–	6
Mamba2-S5	79,475,840	1024	128	–	64	12
Mamba2-S4	45,381,216	768	128	–	64	12
Mamba2-S3	20,772,928	512	128	–	64	12
Mamba2-S2	6,062,424	384	128	–	64	6
Mamba2-S1	2,858,384	256	128	–	64	6
Mamba2-S0	1,625,600	192	128	–	64	6

Table 4. Model configurations of Memora, Hawk, Llama2, and Mamba2 models used in the DNA modeling and de-duplicated FineWebEdu experiments

that we can rewrite the linear recurrent equation as:

$$\boldsymbol{h}_t = \boldsymbol{P}^{-1}\boldsymbol{\Lambda}\boldsymbol{P}\boldsymbol{h}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t \implies \boldsymbol{P}\boldsymbol{h}_t = \boldsymbol{\Lambda}\boldsymbol{P}\boldsymbol{h}_{t-1} + \boldsymbol{P}\boldsymbol{B}\boldsymbol{x}_t.$$

By defining $\tilde{\boldsymbol{h}} \doteq \boldsymbol{P}\boldsymbol{h}$ and $\tilde{\boldsymbol{B}} \doteq \boldsymbol{P}\boldsymbol{B}$, we can write the new recurrence equation as follows:

$$\tilde{\boldsymbol{h}}_t = \boldsymbol{\Lambda}\tilde{\boldsymbol{h}}_{t-1} + \tilde{\boldsymbol{B}}\boldsymbol{x}_t.$$

Since $\boldsymbol{\Lambda}$ is a diagonal matrix, we can utilize its diagonal $\boldsymbol{\lambda} \doteq \mathrm{diag}(\boldsymbol{\Lambda})$. The recurrent equation can be further simplified as:

$$\begin{aligned} \tilde{\boldsymbol{h}}_t &= \boldsymbol{\lambda} \circ \tilde{\boldsymbol{h}}_{t-1} + \tilde{\boldsymbol{B}}\boldsymbol{x}_t \\ \boldsymbol{y}_t &= \Re[\boldsymbol{C}\tilde{\boldsymbol{h}}_t] + \boldsymbol{D}\boldsymbol{x}_t \end{aligned} \tag{7}$$

which is referred to as an independent recurrent module (Zucchet et al. 2023) because each element in the new state does not depend on any interaction with the other elements. Some instantiation of independent recurrent modules with different details and assumptions are LRU (Orvieto et al. 2023), Online LRU (Zucchet et al. 2023), eLSTM (Irie et al. 2024), RTU (Elelimy et al. 2024), Hawk (De et al. 2024), HGRN (Qin et al. 2024), MinGRU/MinLSTM (Feng et al. 2024), and columnar networks (Javed et al. 2023). It is worth mentioning that, in our analysis, we have not made any assumptions so far other than the linearity of f and g. Thus, the equation with a complex-valued diagonal recurrence matrix is representationally equivalent to the original equation with a real-valued dense recurrence matrix.

I.1. Optimization issues with recurrent learning

One issue with the product term $\prod_{i=j+1}^{t} \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{h}_{i-1}}$ in Eq. 4 and Eq. 5 is that it can vanish if the magnitude of the eigenvalues are less than 1 and explode if they are greater than 1. T-BPTT is less sensitive to this issue than BPTT. Further efforts include gating mechanisms to prevent vanishing or exploding gradient from excessive multiplication like LSTM (Hochreiter and Schmidhuber 1997) and GRU (Cho 2014). Other efforts also restricted the eigenvalues of the product matrix to always

be close to but less than 1 (e.g., Arjovsky et al. 2016). Recently, it was noticed that if the recurrent unit is restricted to be linear, controlling its eigenvalues becomes much easier, and thus optimization becomes more efficient (Zucchet and Orvieto 2024), which is what powers modern large-scale recurrent learning methods (e.g., Gu and Dao 2024, De et al. 2024, Dao and Gu 2024).

I.2. Necessity of complex numbers learning

The nature of input signals can vary from discrete to continuous based on the application. For example, they can be discrete, like language text, or continuous, like audio. Empirically, prior research (Gu and Dao 2024) showed that the recurrent system benefits from having complex-valued states in cases where the input signal is continuous and with little to no gain in the case of the discrete input (e.g., language). Thus, the recent recurrent systems with language models usually assume real-valued state $\boldsymbol{h}_t \in \mathbb{R}^n, \forall t$. In this primer, we focus on the general case where the state is complex-valued.

I.3. Stability of Recurrent Learning

In Eq. 7, any entry in $\tilde{\boldsymbol{h}}_t$ can increase without bound if its corresponding eigenvalue is greater than or equal to 1, which makes the system unstable. To maintain stability, $\boldsymbol{\lambda}$ entries are restricted to have a magnitude less than 1. If rectangular representation, $a + ib$, is used, then $\sqrt{a^2 + b^2} < 1$ maintains stability. If trigonometric representation, $r(\cos(\theta) + i\sin(\theta))$, or exponential representation, $re^{i\theta}$, are used, then $r < 1$ maintains stability (Zucchet and Orvieto 2024, Elelimy et al. 2024). Further, if an entry i in $\boldsymbol{\lambda}$ has a magnitude close to one, $|\lambda_i| \approx 1$, this might cause instability if the input contribution, $\tilde{\boldsymbol{B}}, \boldsymbol{x}_t$ is large (Orvieto et al. 2023). Thus, we can dampen the contribution of the input proportionally to the eigenvalue magnitude via multiplication by $\sqrt{1 - |\boldsymbol{\lambda}|}$. The resultant recurrence equation becomes $\tilde{\boldsymbol{h}}_t = \boldsymbol{\lambda} \circ \tilde{\boldsymbol{h}}_{t-1} + \boldsymbol{\gamma} \circ \tilde{\boldsymbol{B}}\boldsymbol{x}_t$, where $\boldsymbol{\gamma} = \sqrt{1 - |\boldsymbol{\lambda}|}$. In the following, we skip these stability modifications to have simpler derivations.

I.4. Separating complex into real and imaginary components

The linear recurrent unit in Eq. 7 can be implemented and used with automatic differentiation libraries as shown in LRU (Orvieto et al. 2023) and online LRU (Zucchet et al. 2023). However, automatic differentiation with complex numbers in existing software libraries is tricky and might give unexpected results (Elelimy et al. 2024) due to lack of adoption and support. Thus, it is better to separate the complex numbers into their real and imaginary components so that we have real equations and imaginary equations. The values of both components are real-valued, and the automatic differentiation libraries can deal with them more easily.

Let us separate the complex recurrence equation into two equations: real and imaginary. We define $\tilde{\boldsymbol{h}}_t = \tilde{\boldsymbol{h}}_t^R + i\tilde{\boldsymbol{h}}_t^I$, where $\tilde{\boldsymbol{h}}_t^R$ is the real part of the state vector and $\tilde{\boldsymbol{h}}_t^I$ is the imaginary part. We do the same trick for $\boldsymbol{\lambda}$ and $\tilde{\boldsymbol{B}}$: $\boldsymbol{\lambda} = \boldsymbol{\lambda}^R + i\boldsymbol{\lambda}^I$ and $\tilde{\boldsymbol{B}} = \tilde{\boldsymbol{B}}^R + i\tilde{\boldsymbol{B}}^I$

The recurrence equation written as:

$$\begin{aligned}\tilde{\boldsymbol{h}}_t^R + i\tilde{\boldsymbol{h}}_t^I &= (\boldsymbol{\lambda}^R + i\boldsymbol{\lambda}^I) \circ (\tilde{\boldsymbol{h}}_{t-1}^R + i\tilde{\boldsymbol{h}}_{t-1}^I) + (\tilde{\boldsymbol{B}}^R + i\tilde{\boldsymbol{B}}^I)\boldsymbol{x}_t \\ &= \boldsymbol{\lambda}^R \circ \tilde{\boldsymbol{h}}_{t-1}^R + i\boldsymbol{\lambda}^R \circ \tilde{\boldsymbol{h}}_{t-1}^I + i\boldsymbol{\lambda}^I \circ \tilde{\boldsymbol{h}}_{t-1}^R - \boldsymbol{\lambda}^I \circ \tilde{\boldsymbol{h}}_{t-1}^I + \tilde{\boldsymbol{B}}^R\boldsymbol{x}_t + i\tilde{\boldsymbol{B}}^I\boldsymbol{x}_t\end{aligned}$$

Let us separate the real components from the imaginary ones:

$$\begin{aligned}\tilde{\boldsymbol{h}}_t^R &= \boldsymbol{\lambda}^R \circ \tilde{\boldsymbol{h}}_{t-1}^R - \boldsymbol{\lambda}^I \circ \tilde{\boldsymbol{h}}_{t-1}^I + \tilde{\boldsymbol{B}}^R\boldsymbol{x}_t \\ i\tilde{\boldsymbol{h}}_t^I &= i\boldsymbol{\lambda}^R \circ \tilde{\boldsymbol{h}}_{t-1}^I + i\boldsymbol{\lambda}^I \circ \tilde{\boldsymbol{h}}_{t-1}^R + i\tilde{\boldsymbol{B}}^I\boldsymbol{x}_t\end{aligned}$$

Note how we can drop i from both sides of the imaginary equations and retain a real-valued equation.

The recurrent state of such a separated system can be seen as $\tilde{\boldsymbol{h}}_t^{\text{combined}} = [\tilde{\boldsymbol{h}}_t^R; \tilde{\boldsymbol{h}}_t^I] \in \mathbb{R}^{2n}$. One advantage of such a view is that we no longer need to learn a complex-valued $\boldsymbol{C}$ matrix. The output $\boldsymbol{y}_t$ is given by $\boldsymbol{y}_t^{\text{combined}} = \tilde{\boldsymbol{h}}_t^{\text{combined}}\boldsymbol{C}^{\text{combined}}$, where $\boldsymbol{C}^{\text{combined}} \in \mathbb{R}^{m \times 2n}$. This is representationally equivalent to learning a complex-valued state $\tilde{\boldsymbol{h}}_t \in \mathbb{R}^n$ and $\boldsymbol{C} \in \mathbb{C}^{m \times n}$ where the output is given as $\boldsymbol{y}_t = \Re[\boldsymbol{C}\tilde{\boldsymbol{h}}_t]$. This is because $\boldsymbol{y}^{\text{combined}} = \begin{bmatrix}\tilde{\boldsymbol{h}}_t^R \\ \tilde{\boldsymbol{h}}_t^I\end{bmatrix} \begin{bmatrix}\boldsymbol{C}_1 & \boldsymbol{C}_2\end{bmatrix} = \boldsymbol{C}_1\tilde{\boldsymbol{h}}_t^R + \boldsymbol{C}_2\tilde{\boldsymbol{h}}_t^I$, where $\boldsymbol{C}_1, \boldsymbol{C}_2 \in \mathbb{R}^{m \times n}$. On the other hand, $\boldsymbol{y}_t = \Re[\boldsymbol{C}\tilde{\boldsymbol{h}}_t] = \Re[(\boldsymbol{C}_R + i\boldsymbol{C}_I)(\tilde{\boldsymbol{h}}_t^R + i\tilde{\boldsymbol{h}}_t^I)] = \boldsymbol{C}_R\tilde{\boldsymbol{h}}_t^R - \boldsymbol{C}_I\tilde{\boldsymbol{h}}_t^I$. If we set $\boldsymbol{C}_R = \boldsymbol{C}_1$ and $\boldsymbol{C}_I = -\boldsymbol{C}_2$, then we get $\boldsymbol{y}_t = \boldsymbol{y}_t^{\text{combined}}$.

The cosine representation is better than the rectangular representation for learning (Elelimy et al. 2024) since we can directly restrict the magnitude of $\mathbf{\Lambda}$ eigenvalues. Utilizing the cosine representation of complex numbers: $a+ib = r\cos(\theta)+i\sin(\theta)$, we can write the two equations differently. We define $\boldsymbol{\lambda}^R = \boldsymbol{r}\cos(\boldsymbol{\theta})$, $\boldsymbol{\lambda}^I = \boldsymbol{r}\sin(\boldsymbol{\theta})$. The resultant recurrence equations are given by:

$$\tilde{\boldsymbol{h}}_t^R = \boldsymbol{r} \circ \cos(\boldsymbol{\theta}) \circ \tilde{\boldsymbol{h}}_{t-1}^R - \boldsymbol{r} \circ \sin(\boldsymbol{\theta}) \circ \tilde{\boldsymbol{h}}_{t-1}^I + \tilde{\boldsymbol{B}}^R \boldsymbol{x}_t,$$
$$\tilde{\boldsymbol{h}}_t^I = \boldsymbol{r} \circ \cos(\boldsymbol{\theta}) \circ \tilde{\boldsymbol{h}}_{t-1}^I + \boldsymbol{r} \circ \sin(\boldsymbol{\theta}) \circ \tilde{\boldsymbol{h}}_{t-1}^R + \tilde{\boldsymbol{B}}^I \boldsymbol{x}_t,$$

which recovers RTU unit (Elelimy et al. 2024).

I.5. Enforcing Conjugate Pairs Learning

Since the recurrent unit matrices are real, $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{B} \in \mathbb{R}^{n\times d}$, and $\boldsymbol{C} \in \mathbb{R}^{n\times m}$, then $\mathbf{\Lambda}$ must contain complex conjugate pairs. It might be beneficial to enforce conjugate pairs in the learning process. We tie the weights of $\mathbf{\Lambda} \in \mathbb{C}^{n\times n}$ enforce conjugate pairs and write it as

$$\mathbf{\Lambda}_{\text{restricted}} = \begin{bmatrix} \lambda_1 & 0 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \cdots & \vdots \\ 0 & 0 & 0 & \lambda_{2n-1}^* & 0 \\ 0 & 0 & 0 & \cdots & \lambda_{2n}^* \end{bmatrix} = \begin{bmatrix} \mathbf{\Lambda}_{\text{original}} \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{\Lambda}_{\text{conjugate}} \end{bmatrix},$$

where $\mathbf{\Lambda}_{\text{original}}, \mathbf{\Lambda}_{\text{conjugate}} \in \mathbb{C}^{\frac{n}{2}\times\frac{n}{2}}$. Here, $\lambda_k, \lambda_k^*, \forall k$ are conjugate pairs representing complex eigenvalues. We call recurrent units with this restriction as restricted complex-valued recurrence.

I.6. Learning with RTRL

To be able to learn with RTRL and the complex recurrent unit, $\tilde{\boldsymbol{h}}_t = \boldsymbol{\lambda} \circ \tilde{\boldsymbol{h}}_{t-1} + \tilde{\boldsymbol{B}}\boldsymbol{x}_t$, we need to compute the sensitivity matrices for $\boldsymbol{\lambda}$, and $\tilde{\boldsymbol{B}}$. We denote $\mathbf{S}^{\boldsymbol{\lambda}} \doteq \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{\lambda}}$ and $\mathbf{S}^{\tilde{\boldsymbol{B}}} \doteq \frac{\partial \boldsymbol{h}_t}{\partial \tilde{\boldsymbol{B}}}$ to the sensitivity matrices for $\boldsymbol{\lambda}$, and $\tilde{\boldsymbol{B}}$, respectively. We refer the reader to (Zucchet et al. 2023) for full derivation and analysis for RTRL for complex-valued LRU. In addition, we refer the reader to (Elelimy et al. 2024) for a derivation for complex-based LRU using real-valued system. Here, we provide the derivation here for completeness. The sensitivity matrices update equation is given as follows:

$$\begin{aligned} S_{t,i,j}^{\boldsymbol{\lambda}} &= \frac{\partial h_{t,i}}{\partial \lambda_j} = \frac{\partial}{\partial \lambda_j}\left(\lambda_i \tilde{h}_{t-1,i} + \sum_m \tilde{B}_{i,m} x_{t,m}\right) \\ &= \delta_{i,j}\tilde{h}_{t-1,i} + \delta_{i,j}\lambda_i S_{t-1,i,j}^{\boldsymbol{\lambda}} \\ S_{t,i,j,k}^{\tilde{\boldsymbol{B}}} &= \frac{\partial h_{t,i}}{\partial B_{j,k}} = \frac{\partial}{\partial B_{j,k}}\left(\lambda_i \tilde{h}_{t-1,i} + \sum_m \tilde{B}_{i,m} x_{t,m}\right) \\ &= \lambda_i S_{t-1,i,j,k}^{\tilde{\boldsymbol{B}}}\delta_{i,j} + \sum_m \delta_{i,j}\delta_{k,m}x_{t,m} \\ &= \lambda_i S_{t-1,i,j,k}^{\tilde{\boldsymbol{B}}}\delta_{i,j} + \delta_{i,j}x_{t,k}. \end{aligned}$$

We can also write the recursive relationship using the reduced sensitivity objects as follows:

$$\mathbf{S}_t^{\boldsymbol{\lambda}} = \frac{\partial}{\partial \boldsymbol{\lambda}}\left(\boldsymbol{\lambda} \circ \tilde{\boldsymbol{h}}_{t-1} + \tilde{\boldsymbol{B}}\boldsymbol{x}_t\right) = \boldsymbol{\lambda} \circ \mathbf{S}_{t-1}^{\boldsymbol{\lambda}} + \tilde{\boldsymbol{h}}_{t-1}$$
$$\mathbf{S}_t^{\tilde{\boldsymbol{B}}} = \frac{\partial}{\partial \tilde{\boldsymbol{B}}}\left(\boldsymbol{\lambda} \circ \tilde{\boldsymbol{h}}_{t-1} + \tilde{\boldsymbol{B}}\boldsymbol{x}_t\right) = \text{Diag}(\boldsymbol{\lambda})\mathbf{S}_{t-1}^{\tilde{\boldsymbol{B}}} + \mathbf{1}\boldsymbol{x}_t^\top$$

where $\circ$ denotes element-wise product. Note how the matrix $\mathbf{S}_t^{\boldsymbol{\lambda}}$ reduces to a vector since $\delta_{i,j} = 0, \forall i \neq j$. Similarly, the 3d tensor $\mathbf{S}_t^{\tilde{\boldsymbol{B}}}$ reduces to a 2d matrix. This structure in the sensitivity objects is a result of the structure of the independent recurrent module.

J. Primer on Parallel Scan

Parallel scan (Blelloch 1990) is an operation that applies a binary associative operator $\bullet$ on a number of elements L in a certain way. Let us consider the linear recurrence $\boldsymbol{h}_{k+1} = \boldsymbol{A}_k \boldsymbol{h}_k + \boldsymbol{B}_k \boldsymbol{x}_k$. For a sequence of length L, we can write the elements belonging to each step k as $c_k = (\boldsymbol{A}_k, \boldsymbol{B}_k \boldsymbol{x}_k)$. The elements $\{c_1, \ldots, c_L\}$ are precomputed before applying the parallel scan operator. The binary associative operator $\bullet$ of this recurrence is given by $q_i \bullet q_j \doteq (q_{j,1} \square q_{i,1}, q_{j,1} \diamondsuit q_{i,2} + q_{k,2})$, where $q_{i,1}$ is the 1st entry of the ith element, $q_{i,2}$ is the 2nd entry of the ith element, $\square$ denotes matrix-matrix multiplication, $\diamondsuit$ denotes matrix-vector multiplication, and $+$ denotes element-wise addition.

First, we perform the *upsweep*, where we recursively combine adjacent pairs of elements to build a binary tree. At the bottom level, we combine $(c_1, c_2), (c_3, c_4), \ldots, (c_{L-1}, c_L)$. Each pair is combined using $\bullet$, and the resulting values form the next level. This process continues until we reach the root. For example, a node covering c_3, c_4, c_5, c_6 will store $c_3 \bullet c_4 \bullet c_5 \bullet c_6$. These values are reused in the next phase. Second, we perform the *downsweep* to turn these tree values into the actual cumulative products. Starting from the root node then traversing the tree, we pass the value of the node to its children where the right child gets the value of the parent combined with the left child's value and the left child gets the same value as the parent. After this process is complete, each leaf node k contains the cumulative product $s_k = c_1 \bullet \cdots \bullet c_{k-1}$, containing all hidden states $\{h_1, \ldots, h_L\}$.

Because both upsweep and downsweep take $O(\log L)$ depth (by combining or distributing pairs in parallel), all hidden states are produced in $O(\log L)$ parallel time. The computational complexity is $O(M \log L)$, where M is the cost of matrix-matrix multiplication. Specifically, the cost is $O(n^3 \log L)$ using dense recurrence matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and $O(n \log L)$ for diagonal recurrence matrix $\boldsymbol{A}$.

K. Computational Resources

In all experiments, we used Nvidia L40 GPUs and Intel Xeon Gold CPUs to obtain the results. In the selective copying experiment, we ran 6 baselines each for 5 independent runs, resulting in a total of $6 \times 7 \times 5$ runs. Each run took around 6 hours to finish. In the second experiment where we varied the sequence length and used the TinyStories dataset, we ran models across 7 sequence lengths and 7 baselines each for 3 independent runs, resulting in a total of $7 \times 7 \times 3$ runs. Each run took about 24 hours of GPU time along with 4 CPUs and 16GB of RAM. In the DNA modeling experiment, we ran 5 models for each baselines with 4 sizes. We ran each model for 3 independent runs, resulting in a total of $5 \times 4 \times 3$ runs. The runs of the largest models took around 3 days of GPU time along with 4 CPUs and 16GB of RAM to finish. In the byte-level FineWebEdu modeling experiment, we ran 5 models for each baselines with 4 sizes. We ran each model for 2 independent runs, resulting in a total of $5 \times 4 \times 2$ runs. The runs of the largest models took around 4 days of GPU time along with 4 CPUs and 24GB of RAM to finish. Finally, we ran 7 baselines in the last experiment each for 3 independent runs, resulting in a total of 7×3 runs each taking 18 hours of GPU time and 4 CPUs and 16GB of RAM. Finally, there was considerable but hard-to-quantify amount of compute used for testing initial ideas, implementations, and preliminary experiments.