# UNLOCKING STATE-TRACKING IN LINEAR RNNS THROUGH NEGATIVE EIGENVALUES

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Linear Recurrent Neural Networks (LRNNs), such as Mamba, RWKV, GLA, mLSTM, and DeltaNet have emerged as efficient alternatives to transformers in large language modeling, offering linear scaling with sequence length and improved training efficiency. However, LRNNs struggle with state-tracking which is important for, e.g., code comprehension or tracking chess pieces across a board. Even parity, the simplest state-tracking task, which non-linear RNNs like LSTMs handle effectively, cannot be solved by current LRNNs. Recently, Sarrof et al. (2024) demonstrated that the failure of LRNNs like Mamba to solve parity stems from restricting the eigenvalue range of their diagonal state-transition matrices to $[0, 1]$, and that incorporating negative eigenvalues can resolve this issue. We generalize this result to full matrix LRNNs, which have recently shown promise in models such as DeltaNet. We prove that no finite-precision LRNN with state-transition matrices having only positive eigenvalues can solve parity, while complex eigenvalues are needed to count modulo 3. Notably, we also prove that LRNNs can learn any regular language when their state-transition matrices are products of identity plus vector outer product matrices with eigenvalues in the range $[-1, 1]$. Our empirical results confirm that extending the eigenvalue range of models like Mamba and DeltaNet to include negative values not only enables them to solve parity but consistently improves their performance on state-tracking tasks. Furthermore, pre-training LRNNs with an extended eigenvalue range for language modeling achieves comparable performance and stability while showing promise for coding tasks. Our work enhances the expressivity of modern LRNNs, broadening their applicability without changing the cost of training or inference.

## 1 INTRODUCTION

Transformer architectures (Vaswani et al., 2017) have revolutionized NLP but scale quadratically in sequence length, posing computational challenges for long sequences. To address this, Linear Recurrent Neural Networks (LRNNs) have emerged as promising alternatives that offer linear scaling while maintaining competitive performance (Gu & Dao, 2023; Dao & Gu, 2024; Yang et al., 2024a; Peng et al., 2023; Deletang et al., 2023; Sun et al., 2024; Beck et al., 2024). LRNNs update their state via matrix-vector products with structured, learnable state-transition matrices, enabling parallelization techniques like the associative scan (Blelloch, 1990) for efficient training. The structure of state-transition matrices largely determines LRNN capabilities. While successful models like Mamba (Gu & Dao, 2023), GLA (Yang et al., 2024a), and mLSTM (Beck et al., 2024) use diagonal matrices (diagonal LRNN), recent work explores more complex forms. Notably, non-diagonal matrices using generalized Householder (GH) transformations, defined as $\boldsymbol{I} + \boldsymbol{u}\boldsymbol{u}^\top$ where $\boldsymbol{u}$ is a learnable vector and $\boldsymbol{I}$ is the identity, show promise in models like DeltaNet (Yang et al., 2024b) and TTT-Linear (Sun et al., 2024), potentially enhancing expressiveness while preserving efficiency.

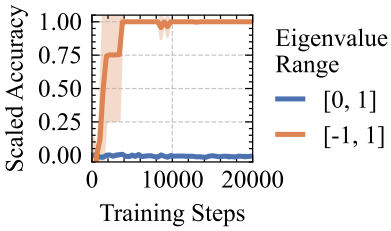



Figure 1: Extending the eigenvalue range of the state transition matrices of diagonal LRNNs improves performance from random guessing (range $[0, 1]$), to perfect score (range $[-1, 1]$) on learning parity. Trained on sequences up to length 40; Tested on lengths 40–256 (3 seeds).

Despite these successes, both transformers and current LRNNs face a fundamental limitation: they struggle to learn and track the state of even simple finite-state machines from sequences of state-transitions (Deletang et al., 2023). This limitation may impair performance on tasks such as entity tracking in narratives, handling nested structures in code, and other reasoning tasks that can benefit from maintaining and updating an internal state over time (Merrill et al., 2024). Even the simplest state-tracking task, computing the parity of a sequence of bits, cannot be solved by current LRNNs, while non-linear RNNs like LSTMs (Hochreiter & Schmidhuber, 1997) and sLSTM (Beck et al., 2024) can solve parity easily (Merrill et al., 2024). However, in-contrast to modern linear RNNs, non-linear RNNs lack an efficient method for parallelizing the training across sequence length.

Recently, Sarrof et al. (2024) demonstrated that the inability of diagonal LRNNs to solve parity problems stems from the constraining the eigenvalues of their state-transition matrices to be positive. Specifically, they proved that diagonal LRNNs, when implemented with finite precision and exclusively positive real eigenvalues, cannot solve the *parity* problem for sequences of arbitrary length. However, their work did not provide empirical evidence showing that diagonal LRNNs with negative eigenvalues can be successfully trained to overcome this limitation. We prove that the same limitation also affects LRNNs with non-diagonal state-transition matrices, and further prove that complex eigenvalues are necessary to solve the more challenging task of modular counting when the modulus is a power of two. Our findings also apply to the GH matrices employed by DeltaNet, as they share the same eigenvalue limitations. To overcome this, we propose a simple yet powerful solution: extend the range of possible eigenvalues from $[0, 1]$ to $[-1, 1]$. This change enables state-tracking and significantly improves LRNNs' expressivity without compromising their efficiency and training stability. We illustrate in Figure 1 that this change allows diagonal LRNNs to learn parity. We open-source the code for our experiments in this anonymous repository.

We make the following *contributions:*

1. We prove that any finite-precision LRNN with only positive real eigenvalues in the state-transition matrices (most LRNNs used in practice) cannot solve parity at arbitrary sequence lengths (Theorem 1), while complex eigenvalues are required to learn counting modulo 3 (Theorem 2).
2. By extending the eigenvalue range to $[-1, 1]$, we significantly improve LRNN state-tracking capabilities. We prove that LRNNs with state-transition matrices formed by products of generalized Householder (GH) matrices in the range $[-1, 1]$ can learn any regular language (Theorem 4), in some cases with just a single layer (Theorem 3). Importantly, such an extension allows even practical LRNNs, using just one GH matrix (like DeltaNet), to learn substantially harder tasks, such as the composition of permutations of two elements (swaps), compared to diagonal LRNNs.
3. We show that the eigenvalue range of Mamba and DeltaNet can be extended to $[-1, 1]$ without compromising efficiency or training stability. We test the modified methods on parity, modular arithmetic, and permutation composition, demonstrating improved state-tracking performance.
4. We pre-train modified versions of DeltaNet (340M parameters) and Mamba (370M parameters) and show that they reach performance comparable to the original models on generative language modeling tasks, while DeltaNet shows improved perplexity on coding and math datasets. Notably, we gain 2 points of perplexity on CodeParrot (Tunstall et al., 2022).

## 2 Related Work

**Linear RNNs.** Linear RNNs encompass state-space models and causal, linear attention mechanisms. State-space models, originally used for continuous dynamical systems, inspired LRNN variants like S4 (Gu et al., 2022) and H4 (Fu et al., 2021). Recent advancements, such as Mamba (Gu & Dao, 2023; Dao & Gu, 2024), introduced input-dependent gating of the hidden state, significantly improving language modeling performance. Concurrently, linear attention emerged as an alternative to classical softmax attention, with Katharopoulos et al. (2020) demonstrating that causal, linear attention transformers can be reformulated as RNNs with linear scaling in sequence length. Building on this, Yang et al. (2024a) proposed Gated Linear Attention (GLA), adding a gating mechanism similar to Mamba, while DeltaNet (Yang et al., 2024b) and TTT-Linear (Sun et al., 2024) explored more expressive gating with non-diagonal state-transition matrices. Recent work has combined non-linear and linear RNNs, as seen in xLSTM (Beck et al., 2024), a successor to the traditional LSTM (Hochreiter & Schmidhuber, 1997).

**Expressivity Results.** Several studies have explored the expressive power of transformers. Liu et al. (2023) demonstrated that transformers can learn shortcut solutions for *solvable* finite state automata, though these solutions lack generalizability to arbitrary sequence lengths and perform poorly out-of-distribution. Unlike RNNs, transformers' high parallelizability prevents them from learning *unsolvable* finite state automata Merrill & Sabharwal (2023). These findings typically use algebraic formal language theory and circuit complexity (we refer to Liu et al. (2023), for a tutorial on these topics), using the *log-precision assumption* and a number of layers scaling linearly or logarithmically with sequence length. While earlier research established transformers' Turing completeness, it relied on either arbitrary precision (Deletang et al., 2023) or infinite depth (Pérez et al., 2021).
Diagonal LRNNs can simulate any RNN with infinite depth Gu & Dao (2023) and approximate regular enough functions when the state dimension grows linearly with sequence length (Orvieto et al., 2024). However, things change when depth and state size are fixed. Merrill et al. (2024) proved that finite-depth diagonal LRNNs, like transformers, cannot learn unsolvable finite state automata when restricted to log-precision arithmetic. In a finite-precision setting, Sarrof et al. (2024) showed that diagonal LRNNs with positive values in the state-transition matrix, while capable of learning all star-free languages, cannot solve even the simple *parity* problem, a non-star-free language recognizable by a solvable automaton with two states. However, their analysis was limited to the diagonal case and they did not test the benefit of negative eigenvalues in practice. Differently from these works, we also study non-diagonal LRNNs that can still be trained efficiently at large scale.

# 3 Background

## 3.1 Linear Recurrent Neural Networks (LRNNs)

We describe LRNNs using notation inspired by Sarrof et al. (2024), focusing on the core linear recurrences while abstracting away non-linear computations for each token. LRNNs are, in fact, stacks of layers with common structure but distinct learnable parameters. Each layer takes input vectors $\boldsymbol{x}_1, \dots, \boldsymbol{x}_t \in \mathbb{R}^l$ and outputs $\hat{\boldsymbol{y}}_1, \dots, \hat{\boldsymbol{y}}_t \in \mathbb{R}^p$ as:

$$\begin{gathered} \boldsymbol{H}_t = \boldsymbol{A}(\boldsymbol{x}_t)\boldsymbol{H}_{t-1} + \boldsymbol{B}(\boldsymbol{x}_t), \quad \hat{\boldsymbol{y}}_t = \mathrm{dec}(\boldsymbol{H}_t, \boldsymbol{x}_t) \\ \boldsymbol{H}_0 \in \mathbb{C}^{n\times d}, \quad \boldsymbol{A} : \mathbb{R}^l \to \mathbb{C}^{n\times n}, \quad \boldsymbol{B} : \mathbb{R}^l \to \mathbb{C}^{n\times d}, \quad \mathrm{dec} : \mathbb{C}^{n\times d} \times \mathbb{R}^l \to \mathbb{R}^p \end{gathered} \tag{1}$$

Here, $\boldsymbol{A}, \boldsymbol{B}$ and $\mathrm{dec}$ are learnable, generally non-linear functions, with $\mathrm{dec}$ usually expressed by a feed-forward neural network. This definition encompasses most LRNN variants, which differ in the form of $\boldsymbol{A}$ and $\boldsymbol{B}$, $\mathrm{dec}$ parameterization, and use of positional encoding. Table 1 illustrates how three popular LRNNs fit this framework. For other architectures, see (Yang et al., 2024b, Table 4).

| | $\boldsymbol{A}(\boldsymbol{x}_t)$ | $\boldsymbol{B}(\boldsymbol{x}_t)$ | $\hat{\boldsymbol{y}}_t$ |
|---|---|---|---|
| **Mamba** | $\mathrm{Diag}\left(\exp\left(-\Delta_{t,i}\exp(\boldsymbol{w}_{1,i})\right)\right)$ | $\Delta_{t,i}x_{t,i}\boldsymbol{k}_t$ | $\psi(\boldsymbol{q}_t^\top \boldsymbol{H}_t^\top + \boldsymbol{w}_2 \odot \boldsymbol{x}_t)$ |
| **GLA** | $\mathrm{Diag}\left(\boldsymbol{\alpha}_t\right)$ | $\boldsymbol{k}_t\boldsymbol{v}_t^\top$ | $\psi(\boldsymbol{q}_t^\top \boldsymbol{H}_t^\top)$ |
| **DeltaNet** | $\boldsymbol{I} - \beta_t\boldsymbol{k}_t\boldsymbol{k}_t^\top$ | $\boldsymbol{k}_t\boldsymbol{v}_t^\top$ | $\psi(\boldsymbol{q}_t^\top \boldsymbol{H}_t^\top)$ |

Table 1: Different LRNNs layers as instances of (1), where $\boldsymbol{\alpha}_t = \mathrm{sigmoid}(\boldsymbol{W}_\alpha \boldsymbol{x}_t)$, $\boldsymbol{\Delta}_t = \mathrm{softplus}(\boldsymbol{W}_\Delta \boldsymbol{x}_t)$, $\beta_t = \mathrm{sigmoid}(\boldsymbol{w}_\beta \boldsymbol{x}_t)$, while $\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t$ are the output of learnable and possibly non-linear functions of $\boldsymbol{x}_t$. Also $\psi : \mathbb{R}^d \to \mathbb{R}^d$ is another learnable function usually containing an MLP and a normalization, while $\boldsymbol{W}_1 \in \mathbb{R}^d$, $\boldsymbol{W}_\Delta \in \mathbb{R}^{d\times l}$, $\boldsymbol{W}_\alpha \in \mathbb{R}^{n\times l}$, $\boldsymbol{w}_\beta \in \mathbb{R}^l$ and $\boldsymbol{w}_2 \in \mathbb{R}^d$ are learnable parameters. For simplicity, for Mamba, we wrote the matrices for the recursion of each row of the state $\boldsymbol{H}_t$ and set $\boldsymbol{x}_t = (x_{t,1}, \dots, x_{t,d})$, $\boldsymbol{W}_1 = (\boldsymbol{w}_{1,1}, \dots, \boldsymbol{w}_{1,d})$ and $\boldsymbol{\Delta}_t = (\Delta_{t,1}, \dots, \Delta_{t,d})$.

The *state-transition matrices* $\boldsymbol{A}(\boldsymbol{x}_t)$ are typically diagonal or generalized Householder (GH), i.e., identity plus vector outer product, as shown in Table 1, to enable efficient matrix-vector products on modern hardware. These matrices consistently have eigenvalues (and norm) in the range $[0, 1]$.

## 3.2 Formal Language Theory

**Finite State Automata and Regular Languages.** A (deterministic) finite state automaton (FSA) is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta)$ where $\Sigma$ is a finite set of letters called alphabet, $Q$ is a finite set of states, $q_0 \in Q$ is the starting state and $\delta : Q \times \Sigma \to Q$ is the state-transition function (see Hopcroft,

2001, for an introduction). We define the set $\Sigma^*$, whose elements are sequences called words, as the smallest superset of $\Sigma$ that contains the empty word $\varepsilon$ and is closed under word concatenation. We extend the state-transition function to $\delta : Q \times \Sigma^* \to Q$ by defining $\delta(q, \varepsilon) = q$ and $\delta(q, \boldsymbol{w}) = \delta(\delta(q, w_1 \dots w_{i-1}), w_i)$ for any $\boldsymbol{w} = w_1 \dots w_i \in \Sigma^*$ with $i \geq 2$. We say that $\delta(q_0, \boldsymbol{w})$ is the state that $\mathcal{A}$ reaches after reading the word $\boldsymbol{w} \in \Sigma^*$. A *language* $L \subseteq \Sigma^*$ is said to be recognized by $\mathcal{A}$ if there exists a recognizing set $R \subseteq Q$ such that $L = \{\boldsymbol{w} \in \Sigma^* : \delta(q_0, \boldsymbol{w}) \in R\}$. Regular languages are the ones that can be recognized by an FSA. Given an FSA $\mathcal{A}$, the set $\mathcal{T}(\mathcal{A}) = \{\delta(\cdot, \boldsymbol{w}) : \boldsymbol{w} \in \Sigma^*\}$ of functions $\rho : Q \to Q$, together with the function composition operation forms a *monoid* called *transition monoid*, i.e. it is associative, closed and contains the identity $\delta(\cdot, \varepsilon)$. Such monoid has a finite number of elements, since $|Q| < \infty$. Moreover, if $\delta(\cdot, w)$ is bijective for every $w \in \Sigma$, then $\mathcal{T}(\mathcal{A})$ forms a *group*, i.e. it contains the inverse for each element.

**State-Tracking and Monoid Word Problems.** State-tracking is the problem of determining the state of a system only by observing a sequence of updates applied to it. Formally, it can be expressed as a *monoid word problem* (Merrill et al., 2024), where given a monoid $(M, \cdot)$ ($M$ is the set and $\cdot$ is the associative operation), we want to send words $m_1 \dots m_t \in M^*$, describing the sequence of updates, to their product $m_1 \cdot m_2 \cdot \ldots \cdot m_t \in M$, representing the state of the system after the updates. If $M$ is finite there is a corresponding FSA $(M, M, e, \delta)$ that solves the word problem, where the starting state is $e$ (the identity element), and the transition function is $\delta(m_1, m_2) = m_2 \cdot m_1$ for $m_1, m_2 \in M$. In this work, we focus on group word problems, i.e. problems where the monoid is also a group. In particular, on the cyclic group $\mathbb{Z}_m$, i.e. addition modulo $m$ and symmetric groups $S_m$, i.e. the group of permutations on $m$ elements. Parity is equivalent to the $S_2$ word problem, while many state-tracking problems such as tracking chess moves or code evaluation, can be shown to be harder than the $S_5$ word problem, which cannot be solved by transformers and diagonal LRNNs even in log-precision (Merrill et al., 2024; Merrill & Sabharwal, 2023).

**One LRNN Layer is an FSA.** Given an alphabet $\Sigma \subset \mathbb{N}$, we can view one layer of an LRNN in (1) as the FSA $\mathcal{A}_{\text{lin}} = (\Sigma, \mathcal{H}, \boldsymbol{H}_0, \delta_{\text{lin}})$, where $\delta_{\text{lin}}(\boldsymbol{H}, w) = \boldsymbol{A}(w)\boldsymbol{H} + \boldsymbol{B}(w)$, which is extended as we saw previously[1], and $\mathcal{H} = \{\delta_{\text{lin}}(\boldsymbol{H}_0, \boldsymbol{w}) : \boldsymbol{w} \in \Sigma^*\}$. We say that a LRNN layer in (1) *implements* the FSA $\mathcal{A} = (\Sigma, Q, q_0, \delta)$ if $\mathcal{A}_{\text{lin}}$ can mimic the state transitions of $\mathcal{A}$[2]. Formally, if there exists a surjective function $g : \mathcal{H} \to Q$, such that for any $\boldsymbol{H} \in \mathcal{H}$, $w \in \Sigma$

$$\delta(g(\boldsymbol{H}), w) = g(\delta_{\text{lin}}(\boldsymbol{H}, w)) = g(\boldsymbol{A}(w)\boldsymbol{H} + \boldsymbol{B}(w))$$

Every language $L$ recognized by $\mathcal{A}$ can also be recognized by this LRNN layer with a sufficiently powerful $\text{dec}$. In particular if $R \subseteq Q$ is the recognizing set for $L$ and $\boldsymbol{H}_0$ is such that $q_0 = g(\boldsymbol{H}_0)$, then the decoder $\text{dec}(\boldsymbol{H}_t, w_t) = \mathbf{1}\{g(\boldsymbol{H}_t) \in R\}$, will correctly determine if $\boldsymbol{w} \in L$. Therefore, implementing $\mathcal{A}$ is harder than recognizing $L$. A principal goal of this work is to show that current LRNNs cannot recognize simple languages such as parity (negative results) while appropriate modifications to the state-transition matrices, enable LRNNs to recognize broader classes of FSA (positive results), with certain classes of FSA requiring a single layer. Notice, that while LRNNs with one layer can recognize any regular language, the state transition matrices might not fit into the structure imposed by current LRNNs, such as those in Table 1 (see Appendix A.2 for more details).

## 4 Theoretical Analysis

We begin by highlighting the limitations of current LRNNs, demonstrating that they fail to meet a necessary condition for solving parity and modular counting problems: the eigenvalues of their state-transition matrices are restricted to the range $[0, 1]$. Subsequently, we illustrate how extending this eigenvalue range to $[-1, 1]$ significantly enhances the expressive power of LRNNs.

### 4.1 Limitations of Current LRNNs

The parity $y_t \in \{0, 1\}$ of a sequence of ones and zeros $x_1 \dots x_t \in \{0, 1\}^t$ is 1 if the total number of ones in the sequence is odd, and 0 if it's even. Equivalent to addition modulo 2, it can be computed by summing the values in the input sequence and then applying the modulo 2 function: $y_t = (\sum_{i=1}^{t} x_i) \bmod 2$. We can also express this as the linear recursion

$$h_t = h_{t-1} + x_t, \quad h_0 = 0, \quad y_t = h_t \bmod 2 \tag{2}$$

[1] We let $\delta_{\text{lin}} : \mathbb{R}^{n \times d} \times \Sigma \to \mathbb{R}^{n \times d}$ and extend it to $\delta_{\text{lin}} : \mathbb{R}^{n \times d} \times \Sigma^* \to \mathbb{R}^{n \times d}$ since we didn't define $\mathcal{H}$ yet

[2] This definition is equivalent to that of FSA homomorphism, see (Maler & Pnueli, 1994, Definition 3)

where $h_t$ contains the total number of ones. This solution can be implemented by an LRNN with one layer and scalar states by setting $\boldsymbol{A}(x_t) = 1$, $\boldsymbol{B}(x_t) = x_t$, $\boldsymbol{H}_0 = 0$, and $\mathrm{dec}(\boldsymbol{H}_t, x_t) = \boldsymbol{H}_t \bmod 2$ in Equation (1). However, implementing such a solution with finite precision presents an issue: the state $h_t$ can grow indefinitely with $t$, eventually reaching the limit of our precision range. Indeed, $h_t \in \{0, \ldots, t\}$, requiring $\log_2(t+1)$ bits for storage. Such solutions, referred to as *shortcut solutions*, are the only ones learnable by transformers when allowing $O(\log(t))$ bits of precision and either depth $O(\log(t))$ or width $O(t)$ (Liu et al., 2023). Moreover, the MLP in $\mathrm{dec}$ must approximate the modulus 2 function, which is challenging to learn due to its discontinuous and periodic nature.

A more efficient solution, which implements the two-state FSA solving this problem, can still be realized by a finite-precision LRNN with one layer and scalar states (and consequently with vector states and diagonal state-transition matrices) using the recursion:

$$h_t = a(x_t)h_{t-1} + b(x_t), \quad h_0 = 0, \quad b(1) = a(0) = 1, a(1) = -1, \quad y_t = h_t. \tag{3}$$

Note that the state-transition scalar $a(1)$ is negative. However, current diagonal LRNNs do not allow negative values, and so are unable to learn parity. This raises the question: can non-diagonal LRNNs, such as DeltaNet, solve parity?

The following result gives an answer to this question by providing necessary condition for a LRNN to solve parity. It generalizes (Sarrof et al., 2024, Theorem 2) to non-diagonal matrices, showing that there must be at least one eigenvalue which is not real and positive. This eigenvalue could simply have a nonzero imaginary part without necessarily being negative and real.

**Theorem 1** (Parity)**.** *A finite-precision LRNN with finitely many layers of the form (1) can solve parity for arbitrary input lengths, in particular it can recognize the language* $(11)^*$*, only if at every layer,* $\boldsymbol{A}(\boldsymbol{x}_t)$ *admits at least one eigenvalue* $\lambda$ *with* $|\lambda| \geq 1$ *and that is not real and positive.*

Notice that Mamba, mLSTM, GLA and even non-diagonal LRNNs such as DeltaNet do not satisfy such requirement. The proof in Appendix B.1 uses the same core idea in the one of (Sarrof et al., 2024, Theorem 2). For one layer, we show that when $\boldsymbol{x} = 1^k$ and the conditions for the eigenvalues of $\boldsymbol{A}(1)$ are not met, each element of state $\boldsymbol{H}_k$ in finite precision will be constant for large enough $k$. Thus, $\hat{\boldsymbol{y}}_k$ cannot be equal to $y_k$ (for $k$ large enough) no matter the choice of $\mathrm{dec}$. To show this, we use the expression for the powers of the Jordan Canonical form of $\boldsymbol{A}(1)$, to prove that each element of $\boldsymbol{A}(1)^k$ either converges or diverges to a point in the complex infinity when $k \to \infty$.

We now study the problem of counting modulo $m$, which can be seen as an easier version of addition modulo $m$. For this problem the input of length $k$ never changes and is equal to $\boldsymbol{x} = 1^k$, while the correct output is $y_k = \sum_{i=1}^{k} x_i \bmod m$. The following theorem establishes that to solve this problem, the state-transition matrices of the LRNN must have at least one eigenvalue with a nonzero imaginary part and modulus greater than or equal to one.

**Theorem 2** (Modular Counting)**.** *A finite-precision LRNN with finitely many layers can solve modular counting with modulus greater than 2 for arbitrary input lengths, in particular it can recognize the language* $(1^m)^*$ *with* $m$ *not a power of two, only if at every layer* $\boldsymbol{A}(\boldsymbol{x}_t)$ *admits at least one eigenvalue* $\lambda$ *with nonzero imaginary part and such that* $|\lambda| \geq 1$*.*

Note that all LRNNs allowing only symmetric or triangular state-transition matrices with real entries do not satisfy the assumptions of Theorem 2. For one layer, the proof in Appendix B.2 is similar that of Theorem 1 but we consider the two sequences $\boldsymbol{H}_{2k}$ and $\boldsymbol{H}_{2k+1}$, showing that they have a defined limit when $k \to \infty$, even when $\boldsymbol{A}(1)$ admits negative eigenvalues less or equal than $-1$.

Theorems 1 and 2 identify a fundamental limitation of current design choices on the structure of the state-transition matrices of LRNNs. Specifically, the most popular approaches outlined in Table 1 are incapable of solving parity problems, as the eigenvalues of their state-transition matrices are confined to the interval $[0, 1]$. Further, even if we allow negative eigenvalues that are still real, we cannot solve counting modulus $m$. However, as we will see in the next section, LRNNs with state-transition matrices that are products of generalized Householder (GH) matrices, each with eigenvalues in $[-1, 1]$, are more powerful than LRNNs with diagonal state-transition matrices.

### 4.2 Allowing Negative Eigenvalues

In this section, we explore the implications of extending the eigenvalue range to include negative values. We focus on two classes of LRNNs: those with diagonal state-transition matrices, which are

currently the most prevalent (including GLA, Mamba, and Mamba2), and those with generalized Householder (GH) state-transition matrices, as recently proposed in the DeltaNet architecture. In particular, if we let $\boldsymbol{s} : \mathbb{R}^l \to [0,1]^n$ and $\phi : \mathbb{R}^l \to [0,1]$, $\boldsymbol{v} : \mathbb{R}^l \to \mathbb{R}^n$, being learnable functions such that $\|\boldsymbol{v}(\boldsymbol{x})\| = 1$ for every $\boldsymbol{x} \in \mathbb{R}^l$, then the state transition matrices of each layer of many LRNNs, such as those in Table 1, can be written as either

$$\boldsymbol{A}_{\text{diag}}(\boldsymbol{x}) := \text{Diag}(\boldsymbol{s}(\boldsymbol{x})), \quad \text{or} \quad \boldsymbol{A}_{\text{GH}}(\boldsymbol{x}) := \boldsymbol{I} - \phi(\boldsymbol{x})\boldsymbol{v}(\boldsymbol{x})\boldsymbol{v}(\boldsymbol{x})^\top,$$

where $\boldsymbol{A}_{\text{diag}}(\boldsymbol{x})$ is diagonal and has every eigenvalue $s(\boldsymbol{x})_i \in [0,1]$ and $\boldsymbol{A}_{\text{GH}}(\boldsymbol{x})$ is GH and has all eigenvalues equal to one except for the one associated to the eigenvector $\boldsymbol{v}(\boldsymbol{x})$, which is equal to $1 - \phi(\boldsymbol{x}) \in [0,1]$. To address the limitations discussed in the previous section, we propose the following modification that can be easily applied to any LRNN belonging to either class.

$$\boldsymbol{A}^-_{\text{diag}}(\boldsymbol{x}) := \text{Diag}(2\boldsymbol{s}(\boldsymbol{x}) - 1), \quad \boldsymbol{A}^-_{\text{GH}}(\boldsymbol{x}) := \boldsymbol{I} - 2\phi(\boldsymbol{x})\boldsymbol{v}(\boldsymbol{x})\boldsymbol{v}(\boldsymbol{x})^\top. \tag{4}$$

Note that $\boldsymbol{A}^-_{\text{diag}}(\boldsymbol{x})$ has eigenvalues $2s(\boldsymbol{x})_i - 1 \in [-1,1]$ and $\boldsymbol{A}^-_{\text{GH}}(\boldsymbol{x})$ has all eigenvalues equal to one, except for one that is equal to $1 - 2\phi(\boldsymbol{x}) \in [-1,1]$. Thus, we have extended the range of eigenvalues from $[0,1]$ to $[-1,1]$.

We know from the previous section, that LRNNs with the modified state transition matrices can implement the solution to the parity problem by setting $s(1) = 0$ and $\phi(1) = 1$ so that if we consider a scalar recursion, then $\boldsymbol{A}^-_{\text{diag}}(1) = \boldsymbol{A}^-_{\text{GH}}(1) = -1$. However, we have also shown that since the eigenvalues are real, we cannot solve counting modulo $m$ with $m \geq 3$. Despite this, we note that counting modulo $m$ is linked to rotation by an angle of $2\pi/m$ radians in $\mathbb{R}^2$, and we can express a rotation as a product of two reflections, each of which can be written as a GH matrix. In other words, for any integer $m \geq 3$ there exist unit norm vectors $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathbb{R}^2$ such that

$$\boldsymbol{R}(\theta) := \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \left(\boldsymbol{I} - 2\boldsymbol{v}_1\boldsymbol{v}_1^\top\right)\left(\boldsymbol{I} - 2\boldsymbol{v}_2\boldsymbol{v}_2^\top\right), \quad \theta = \frac{2\pi}{m}.$$

Interestingly, this construction can be done only with GH matrices having one eigenvalue equal to $-1$. If we set the state-transition matrix in eq. (1) to $\boldsymbol{A}(1) = \boldsymbol{R}(\theta)$, an LRNN with one layer can count modulo $m$, since if we also set $\boldsymbol{H}_0 = (1,0)^\top$, $\boldsymbol{M} = (\boldsymbol{H}_0, \boldsymbol{R}(\theta)\boldsymbol{H}_0, \dots, \boldsymbol{R}((m-1)\theta)\boldsymbol{H}_0)^\top \in \mathbb{R}^{m\times 2}$, $\boldsymbol{B}(1) = 0$ and $\text{dec}(\boldsymbol{H}, x) = \arg\max_i \boldsymbol{M}_i^\top \boldsymbol{H} - 1$, then for the input $\boldsymbol{x} = 1^t$ we get

$$\hat{y}_t = \text{dec}(\boldsymbol{H}_t, 1) = \text{dec}(\boldsymbol{A}(1)^t\boldsymbol{H}_0, 1) = \text{dec}(\boldsymbol{R}(t\theta)\boldsymbol{H}_0, 1) = t \bmod m.$$

Therefore, we examine the impact of our change to the eigenvalue range on state-transition matrices constructed as repeated products of GH matrices.

### 4.3 Products of Generalized Householder Matrices

We define the set of all matrices in $\mathbb{R}^{n\times n}$ that can be expressed as a product of $k$ GH matrices with a given range $\Omega \subseteq \mathbb{R}$ of allowed eigenvalues:

$$\mathcal{M}_k(\Omega) := \left\{\boldsymbol{C}_1\boldsymbol{C}_2\cdots\boldsymbol{C}_k \;:\; \boldsymbol{C}_i = \boldsymbol{I} - \beta_i\boldsymbol{v}_i\boldsymbol{v}_i^\top, \quad (1-\beta_i) \in \Omega, \quad \boldsymbol{v}_i \in \mathbb{R}^n, \|\boldsymbol{v}_i\| = 1\right\}. \tag{5}$$

We first observe that if $\boldsymbol{M}_{\text{H}} \in \mathcal{M}_1(\{-1\})$, then $\boldsymbol{M}_{\text{H}}$ is a reflection, or Householder matrix, and that for any $\boldsymbol{x} \in \mathbb{R}^l$, $\boldsymbol{A}_{\text{GH}}(\boldsymbol{x}) \in \mathcal{M}_1([0,1])$ and $\boldsymbol{A}^-_{\text{GH}}(\boldsymbol{x}) \in \mathcal{M}_1([-1,1])$ so that with our change we also include reflections. Note also that $\mathcal{M}_k(\Omega) \subseteq \mathcal{M}_{k'}(\Omega')$ if either $k' \geq k$ and $1 \in \Omega$, or if $\Omega \subseteq \Omega'$. Repeated products of diagonal matrices with values in the range $[-1,1]$ remain diagonal, with eigenvalues in the same range.

More interestingly, our next result shows that products of GH matrices can represent any matrix with Euclidean norm less than or equal to 1. Without our modification, however, or even just by restricting to $\mathcal{M}_k((-1,1])$, they are limited to matrices where any eigenvalue is either equal to 1 or has modulus strictly smaller than 1.

**Proposition 1** (Expressivity of products of GH matrices)**.** *The following hold for $\mathcal{M}_k$ in eq.* (5)*:*

1. *For any $k \in \mathbb{N}$ and $\boldsymbol{N} \in \mathcal{M}_k([-1,1])$, $\|\boldsymbol{N}\| \leq 1$.*
2. *For any $\boldsymbol{M} \in \mathbb{R}^{n\times n}$ with $\|\boldsymbol{M}\| \leq 1$, then $\boldsymbol{M} \in \mathcal{M}_{3n}([-1,1])$ and if $\boldsymbol{M}$ is orthogonal then $\boldsymbol{M} \in \mathcal{M}_n(\{-1,1\})$, while $\boldsymbol{M} \in \mathcal{M}_{n-1}(\{-1,1\})$ when $\boldsymbol{M}$ is a permutation matrix.*

3. *Any eigenvalue $\lambda$ of $\boldsymbol{N} \in \mathcal{M}_k((-1,1])$ is either $1$ or satisfies $|\lambda| < 1$ and if in addition $\boldsymbol{N} \in \mathcal{M}_k([0,1])$, then $\lambda \in \mathbb{R}$.*

The proof in Appendix C.1 uses linear algebra arguments, the SVD decomposition, and the fact that that every $n \times n$ orthogonal matrix can be written as a product of $n$ reflections.

A consequence of Proposition 1 is that if for every layer of an LRNN there exists $k \in \mathbb{N}$ such that the map $\boldsymbol{A}$ from inputs to state transition matrix is such that $\boldsymbol{A} : \mathbb{R}^l \to \mathcal{M}_k((-1,1]) \subset \mathbb{R}^{n\times n}$, then the LRNN cannot learn to count modulo $m$, with $m \geq 3$, due to Theorem 2. On the contrary, if we allow $\boldsymbol{A} : \mathbb{R}^l \to \mathcal{M}_k([-1,1])$ and $k$ is large enough, we can show that an LRNN with one layer can implement any FSA whose transition monoid is a group and that with multiple layers the LRNN can recognize any regular language. This is the content of the following two theorems.

**Theorem 3.** *Every FSA $\mathcal{A} = (\Sigma, Q, q_0, \delta)$ whose transition monoid $\mathcal{T}(\mathcal{A})$ is also a group, can be implemented by a finite-precision LRNN with one layer and $\boldsymbol{A} : \Sigma \to \mathcal{M}_{k-1}(\{-1,1\}) \subset \mathbb{R}^{n\times n}$, where $n$ is the smallest natural number such that $\mathcal{T}(\mathcal{A})$ is isomorphic to a subgroup of $S_n$, and $k = \max_{w\in\Sigma} \sum_{q\in Q} \mathbf{1}\{\delta(q,w) \neq q\}$ is the maximum number of changed states after applying a single transition. Moreover, if $\mathcal{T}(\mathcal{A})$ is isomorphic to the cyclic group $\mathbb{Z}_m$, then we can set $\boldsymbol{A} : \Sigma \to \mathcal{M}_2([-1,1]) \subset \mathbb{R}^{2\times 2}$. If $m = 2$ (parity) we can set $\boldsymbol{A} : \Sigma \to \{-1,1\}$.*

The proof in Appendix C.2 uses matrix representations of groups to map each state-transition function to the corresponding matrix representation. This can always be done using permutation matrices, but for cyclic groups we can also use $2 \times 2$ rotation matrices. In the case of permutations, if every state-transitions permutes at most $k$ states then the corresponding permutation matrix will be in $\mathcal{M}_{k-1}(\{-1,1\})$, since if it is not the identity, it can be written as a product of at most $k-1$ permutations of two elements (swaps), each in $\mathcal{M}_1(\{-1\})$.
A consequence of Theorem 3 is that if every transition function of the FSA has a permutation representation corresponding to a swap or to the identity, then a LRNN layer with $\boldsymbol{A} = \boldsymbol{A}^-_{\mathrm{GH}}$, can implement it. This is useful in practice because the cost of the recursion increases $k$-fold if we use a product of $k$ GH matrices compared to just one. Also, for many problems, state-transition for the FSA might either be simple or be encoded using multiple letters. For example, for addition modulo 5, a word may look like "3+2+6=1" (two letters per addition operations). This makes it possible even for an LRNN with state-transition matrices in $\mathcal{M}_1([-1,1])$ to model complex transitions. Indeed if each transition uses $k$ letters, then if we set $\boldsymbol{B} \equiv 0$ and $\boldsymbol{A} : \mathbb{R}^l \to \mathcal{M}_1([-1,1])$ in eq. (1), then

$$\boldsymbol{H}_t = \boldsymbol{C}(x_t, \ldots, x_{t-k})\boldsymbol{H}_{t-k}, \quad \boldsymbol{C}(x_t, \ldots, x_{t-k}) := \boldsymbol{A}(x_1)\boldsymbol{A}(x_2)\cdots\boldsymbol{A}(x_{t-k}) \in \mathcal{M}_k([-1,1]),$$

which allows to model permutations that change up to $k+1$ elements.

**Theorem 4.** *Every FSA $\mathcal{A} = (\Sigma, Q, q_0, \delta)$ can be implemented by a finite-precision LRNN with $s \leq 2^{|Q|}$ layers, each of the form 1, where $n \leq |Q|$, $p \leq s$, $d = 1$, $\boldsymbol{A} : \mathbb{R}^l \to \mathcal{M}_n([-1,1]) \subset \mathbb{R}^{n\times n}$ and $\boldsymbol{B} : \mathbb{R}^l \to \mathbb{N}^n$. Therefore, LRNNs with state transition matrices that are repeated products of GH matrices each with eigenvalues in the range $[-1,1]$ can recognize any regular languages.*

The proof in Appendix C.4 exploits the landmark Theorem by Krohn & Rhodes (1965), which states that every FSA can be decomposed as a *cascade* of simpler FSAs whose state-transition functions are either one-to-one or constant. Each layer of the LRNN will implement each FSA of the cascade using $n \times n$ permutation matrices, with $n$ being the number of states of the FSA, which are in $\mathcal{M}_{n-1}(\{-1,1\})$, for the one-to-one transitions, while for constant (state-independent) transitions it will set the state-transition matrix to $0 \in \mathcal{M}_n(\{0\})$ and set $\boldsymbol{B}$ in (1) accordingly.
Note that we can obtain the matrix $0 \in \mathbb{R}^{n\times n}$ only inefficiently as a product of $n$ GH matrices, while it can also be obtained with a single diagonal matrix. This points towards LRNNs hybrids using a mix of GH and diagonal matrices, whose exploration we leave for future work.

**Discussion** The results in Theorems 3 and 4 for LRNNs are in sharp contrast with the ones for transformers (Liu et al., 2023; Merrill & Sabharwal, 2023) and diagonal LRNNs (Merrill et al., 2024), which always require either the number of layers or the precision growing with the sequence length, and in most cases can only solve group word problems where the group is *solvable*, i.e. excluding $S_n$ with $n \geq 5$. Moreover, we note that compared to LRNNs without any restriction to the norm of the state-transition matrices, which need only one layer to recognize any regular language, our result requires both the number of layers and the width of the LRNN to be (in the worst case) exponential in the number of states of the FSA, although we conjecture that the number of layers can probably be reduced to at most linear using a more refined decomposition.

# 5 EXPERIMENTS

We investigate the effects of expanding the eigenvalue range of state-transition matrices from $[0, 1]$ to $[-1, 1]$, as explained in Section 4.2, on both synthetic tasks and language modeling. Our experiments involve Mamba, and DeltaNet, with variants trained using both the original and extended eigenvalue ranges, as shown in Table 2. We label these variants accordingly. Note that the changes increase the expressivity of Mamba and DeltaNet while coming at no additional computational cost. Detailed information on the implementation can be found in Appendix D.4.

Table 2: Summary of modifications to the state-transition matrices $\boldsymbol{A}(\boldsymbol{x}_t)$ to extend the eigenvalue range from $[0, 1]$ (Table 1) to $[-1, 1]$. We set $\boldsymbol{s}(\boldsymbol{x}_t) = \exp\left(-\Delta_{t,i} \exp(\boldsymbol{w}_{1,i})\right)$.

| | $[0, 1]$ | $[-1, 1]$ |
|---|---|---|
| Mamba | $\mathrm{Diag}(\boldsymbol{s}(\boldsymbol{x}_t))$ | $\mathrm{Diag}(2\boldsymbol{s}(\boldsymbol{x}_t) - 1)$ |
| DeltaNet | $\boldsymbol{I} - \beta_t \boldsymbol{k}_t \boldsymbol{k}_t^\top$ | $\boldsymbol{I} - 2\beta_t \boldsymbol{k}_t \boldsymbol{k}_t^\top$ |

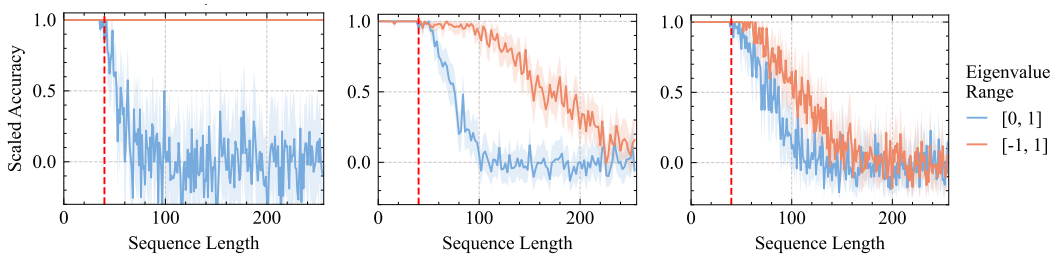
## 5.1 CHOMSKY HIERARCHY

We conducted experiments with some of the formal language tasks proposed by Deletang et al. (2023) and similarly used to benchmark xLSTM (Beck et al., 2024). Our focus was on tasks where mLSTM (a linear RNN) previously underperformed while sLSTM (a non-linear RNN) succeeded, specifically parity, modular arithmetic (both regular languages), and modular arithmetic with brackets (context-free language). As in Beck et al. (2024), we trained each model with sequence lengths ranging from 3 to 40 and evaluated on lengths from 40 to 256, to assess length generalization. Note that our theoretical results cover just regular languages, excluding modular arithmetic with brackets.

Table 3: Performance comparison of various recurrent models on formal language tasks. We report the best of 3 runs (Table 5 in the Appendix reports the median). Scores are scaled accuracy, with 1.0 indicating perfect performance and 0.0 random guessing. The positive impact of allowing negative eigenvalues ($[-1, 1]$ range) versus restricting to positive eigenvalues ($[0, 1]$ range) is evident for both Mamba and DeltaNet. Results in parenthesis are as reported in Beck et al. (2024).

| | **Parity** | **Mod. Arithm. (w/o brackets)** | **Mod. Arithm. (w/ brackets)** |
|---|---|---|---|
| mLSTM | 0.087 (0.04) | 0.122 (0.04) | 0.120 (0.03) |
| sLSTM | **1.000** (1.00) | **0.135** (1.00) | **0.133** (0.57) |
| Mamba $[0, 1]$ | 0.000 | 0.003 | 0.018 |
| Mamba $[-1, 1]$ | **1.000** | **0.111** | **0.036** |
| DeltaNet $[0, 1]$ | 0.017 | 0.187 | 0.182 |
| DeltaNet $[-1, 1]$ | **1.000** | **0.612** | **0.339** |

We compared mLSTM and sLSTM with two models: Mamba and DeltaNet. Our findings, presented in Table 3, demonstrate that expanding the range of eigenvalues from [0, 1] to [-1, 1] enables all examined models to fully solve the parity task, confirming Theorem 1. For modular arithmetic, this expansion led to substantial performance improvements for Mamba and especially DeltaNet, since the latter has non-diagonal state-transition matrices, more suited for this task (see Theorem 3). In Figure 2, we also report performance vs sequence length for DeltaNet. Note that we were unable to replicate the sLSTM results reported by Beck et al. (2024) for the modular arithmetic tasks. Additional experiments and details on the tasks in Appendix D.1.




(a) Parity (b) Mod. Arithm. w/o brackets (c) Mod. Arithm. w/ brackets

Figure 2: *DeltaNet* performance (scaled accuracy) on formal language tasks across sequence lengths. The models were trained on sequences up to length 40 (red dashed line). We test on 8192 randomly sampled sequences of lengths between 40 and 256. The curves show mean and 95% confidence interval.

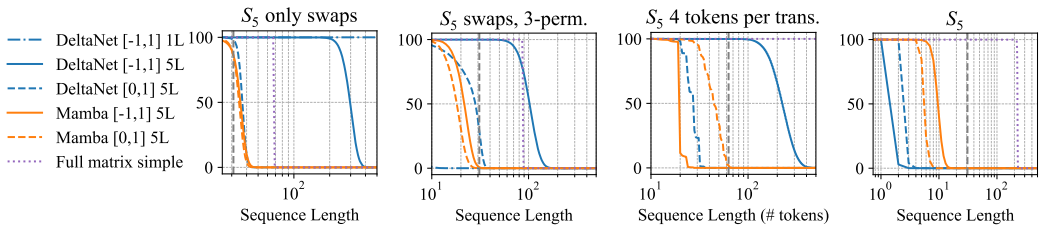



Figure 3: Validation sequence accuracy for different lengths on $S_5$ after 30 (2 leftmost plots) and 90 epochs of training (1 seed). The dashed vertical line indicates the sequence length used during training (32 except for the third plot from the left where it is 64 since we use 4 tokens per permutation). Each method is labeled with name, eigenvalue range, and number of layers. The dashed vertical line indicates the sequence length used during training. Full matrix simple is a simple one-layer baseline where the state update matrices are unstructured and have no control on the eigenvalues.

### 5.2 State-Tracking

We perform experiments on group word problems, relying on the code provided by Merrill et al., 2024. In particular, we focus on the $S_5$ group, which is the first *unsolvable* symmetric group where current LRNN and transformers are known to perform poorly. We also report results for the addition modulo 60, i.e. the cyclic group $\mathbb{Z}_{60}$, in Appendix D.2.2. We note that parity is $S_2$. In these experiments, the input to the model is a sequence of group elements, while the supervision is given by another sequence of group elements, each being the product of the previous ones in the input. Since solving $S_5$ would require LRNNs with state-transition matrices that are repeated products of 4 GH matrices (see Theorem 3), each with eigenvalues $[-1, 1]$, we also consider three simplified setups: (i) allowing as inputs only permutations up to 2 elements (identity and swaps), (ii) allowing only permutations up to 3 elements, (iii) using 4 tokens for each permutation. Additional details are in Appendix D.2. We stress that, even when restricting the inputs, possible outputs remain the same, since swaps are generators of the group.

**Results** Figure 3 shows that, as predicted by Theorem 3, restricting the inputs to only swap permutations allows DeltaNet $[-1, 1]$ with one layer to fully learn the task (since its state transition matrix can model a swap), while DeltaNet $[0, 1]$ only manages to fit the training length, even with 5 layers. On the contrary, just by including also permutations of 3 elements, we notice a substantial decrease in the performance of all models, although interestingly extending the range is still advantageous and DeltaNet $[-1, 1]$ with 5 layers reaches a good length generalization. Moreover, using 4 tokens per group element seems also beneficial compared to standard $S_5$, since DeltaNet $[-1, 1]$ with 5 layers manages to extrapolate very well until around length 120, which corresponds to 30 group elements, while all models trained on standard $S_5$ have 0 sequence accuracy prior to sequence length 20. We also report that Mamba, being a diagonal model, performs poorly on all setups, with and without increased eigenvalue range.

### 5.3 Language Modeling

**Experimental Setup** We train DeltaNet models with 340M parameters and Mamba models with 370M parameters, each using both original and extended eigenvalue ranges. The training is done on 32B tokens from the FineWeb-100B dataset (Penedo et al., 2024). For training details, we refer to Appendix D.3.1. Given our previous theoretical and experimental findings, we hypothesize that models (especially DeltaNet) with extended eigenvalue range will perform better on language modeling tasks that require state-tracking such as coding or mathematics, compared to unmodified models. To test this hypothesis, we evaluate the perplexity of these models in a length extrapolation setup using various datasets: CodeParrot (Tunstall et al., 2022) for coding, Math-Hard (Hendrycks et al., 2021) for mathematics, TriviaQA (Joshi et al., 2017), and SlimPajama (Soboleva et al., 2023).

**Results** Both models trained stably even with our modification and without changing the learning rate. The validation perplexity was comparable, albeit slightly lower, throughout training (See Figure 7 in the Appendix). The experiments in Figure 4 demonstrate that on coding and math datasets, DeltaNet with an eigenvalue range of $[-1, 1]$ achieves lower perplexity than the baseline with range $[0, 1]$. For TriviaQA, the perplexity of DeltaNet $[-1, 1]$ is slightly higher. Note that this is a task relying on memorization, not linked to state-tracking and hence we do not expect an improvement. On

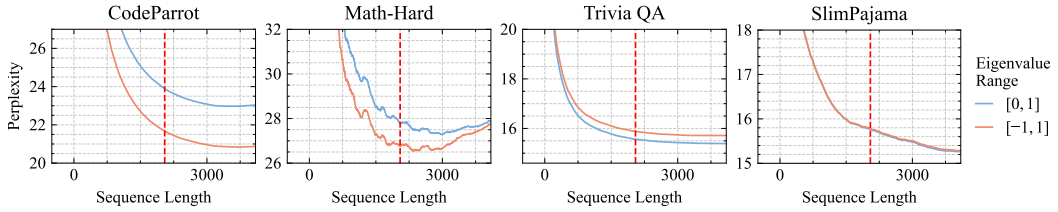



Figure 4: Performance vs sequence length of DeltaNet variants on different datasets. DeltaNet with eigenvalue range [-1, 1] shows improved perplexity on coding and math tasks compared to the [0, 1] baseline. The dashed vertical line indicates the training context length of 2048 tokens.

Table 4: Performance comparison using lm-harness benchmark (Gao et al., 2024) (SPJ reproduced from Yang et al. (2024b), Fine-Web ours). Results show for original and extended eigenvalue range. Our models show comparable performance across tasks.

| | Model | Wiki. ppl ↓ | LMB. ppl ↓ | LMB. acc ↑ | PIQA acc ↑ | Hella. acc_n ↑ | Wino. acc ↑ | ARC-e acc ↑ | ARC-c acc_n ↑ | Avg. ↑ | SWDE cont. ↑ | SQUAD cont. ↑ | FDA cont. ↑ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *15B tokens SPJ* | *340M params* | | | | | | | | | | | | |
| | Transformer++ | 28.39 | 42.69 | 31.0 | 63.3 | 34.0 | 50.4 | 44.5 | 24.2 | 41.2 | 42.2 | 22.1 | 21.4 |
| | Mamba $[0, 1]$ | 28.39 | 39.66 | 30.6 | 65.0 | 35.4 | 50.1 | 46.3 | 23.6 | 41.8 | 12.4 | 23.0 | 2.1 |
| | GLA $[0, 1]$ | 29.47 | 45.53 | 31.3 | 65.1 | 33.8 | 51.6 | 44.4 | 24.6 | 41.8 | 24.0 | 24.7 | 7.3 |
| | DeltaNet $[0, 1]$ | 28.24 | 37.37 | 32.1 | 64.8 | 34.3 | 52.2 | 45.8 | 23.5 | 42.1 | 26.4 | 28.9 | 12.8 |
| *32B tokens FineWeb* | *340M params* | | | | | | | | | | | | |
| | DeltaNet $[0, 1]$ | 28.71 | 42.63 | 28.5 | 67.5 | 40.4 | 51.3 | 46.8 | 24.5 | 43.2 | 34.3 | 30.0 | 10.5 |
| | DeltaNet $[-1, 1]$ | 29.01 | 46.38 | 28.3 | 68.0 | 39.1 | 51.0 | 48.4 | 23.4 | 43.0 | 31.3 | 26.3 | 9.6 |
| | *370M params* | | | | | | | | | | | | |
| | Mamba $[0, 1]$ | 32.04 | 42.82 | 29.1 | 67.4 | 39.6 | 52.7 | 47.0 | 24.4 | 43.4 | 14.2 | 20.1 | 1.3 |
| | Mamba $[-1, 1]$ | 32.41 | 55.09 | 26.5 | 67.6 | 39.2 | 53.0 | 46.8 | 24.0 | 42.9 | 12.5 | 18.2 | 1.3 |

SlimPajama, we observe no significant difference between the two DeltaNet variants. For Mamba instead, we see a general degradation of the performance on these tasks compared to the unmodified version (Figure 8 in the Appendix).

To ensure our models are comparable with those obtained by Yang et al. (2024b), we evaluate them on the same benchmark tasks from lm-harness (Gao et al., 2024) in Table 4. It is worth noting that we trained on 32B tokens of FineWeb, while Yang et al. (2024b) reported results from training on 15B tokens of SlimPajama. We find that our models perform worse in terms of perplexity on WikiText and LAMBADA, while achieving better average accuracy on classic benchmarks. Furthermore, we report that DeltaNet [0,1] performs better on recall-intensive tasks SWDE and SQuAD, where our eigenvalue extension slightly degrades performance.

## 6 Conclusion

In this work, we showed the substantial impact of extending the eigenvalue range of state-transition matrices in LRNNs from $[0, 1]$ to $[-1, 1]$. This modification provably enhances the expressivity of LRNNs in state-tracking tasks with no additional overhead in training or inference. While Mamba successfully solves the parity problem, its diagonal matrix structure inherently limits further performance gains. In contrast, DeltaNet, by leveraging its non-diagonal matrix structure, excels across a broader spectrum of tasks. Notably improving the perplexity on CodeParrot by 2 points. Our results underscore the critical role of non-diagonal state-transition matrices in augmenting state-tracking capabilities, highlighting a promising direction for future LRNN advancements.

**Limitations** In our language modeling experiments, we did not observe any performance gains with the Mamba model. Furthermore, diagonal models such as Mamba2 and GLA use the positivity of state transition matrices to compute repeated products in log space for numerical precision, a technique our modification does not directly support. This limitation may introduce potential instabilities in certain cases (refer to Appendix D.4 for more details).

**Future work** Further research is needed to assess the impact of training large-scale language models with state-tracking capabilities. To this end, we aim to understand the potential downsides of increased expressivity, which could guide hybrid model design. For example, we hypothesize a fundamental trade-off between state-tracking and memorization, which holds also theoretical interest.

## REFERENCES


Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. Language models enable simple systems for generating structured views of heterogeneous data lakes. *Proceedings of the VLDB Endowment*, 17(2):92–105, 2023.

Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xLSTM: Extended Long Short-Term Memory. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2024.

Yonatan Bisk, Rowan Zellers, Ronan Le bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):7432–7439, Apr. 2020.

Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

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

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

Gregoire Deletang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, et al. Neural Networks and the Chomsky Hierarchy. In *The Eleventh International Conference on Learning Representations*, 2023.

Daniel Y Fu, Tri Dao, Khaled Kamal Saab, Armin W Thomas, Atri Rudra, and Christopher Re. Hungry hungry hippos: Towards language modeling with state space models. In *The Eleventh International Conference on Learning Representations*, 2021.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 07 2024.

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

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

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural computation*, 9(8): 1735–1780, 1997.

JE Hopcroft. Introduction to automata theory, languages, and computation, 2001.

Roger A Horn and Charles R Johnson. *Matrix Analysis*. Cambridge University Press, 2012.

Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, 2017.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pp. 5156–5165. PMLR, 2020.

Kenneth Krohn and John Rhodes. Algebraic theory of machines. i. prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116: 450–464, 1965.

Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*, 2023.

Colin Lockard, Prashant Shiralkar, and Xin Luna Dong. When open information extraction meets the semi-structured web. *NAACL-HLT. Association for Computational Linguistics*, 2019.

Ilya Loshchilov and Frank Hutter. SGDR: Stochastic Gradient Descent with Warm Restarts. In *International Conference on Learning Representations*, 2017.

Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*, 2019.

Oded Maler and Amir Pnueli. On the cascaded decomposition of automata, its complexity and its application to logic. *ACTS Mobile Communication*, 48, 1994.

William Merrill and Ashish Sabharwal. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023.

William Merrill, Jackson Petty, and Ashish Sabharwal. The Illusion of State in State-Space Models. In *Forty-first International Conference on Machine Learning*, 2024.

Antonio Orvieto, Soham De, Caglar Gulcehre, Razvan Pascanu, and Samuel L Smith. Universality of linear recurrences followed by non-linear projections: Finite-width guarantees and benefits of complex eigenvalues. In *Forty-first International Conference on Machine Learning*, 2024.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc-Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1525–1534, 2016.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale, 2024.

Bo Peng, Eric Alcaide, Quentin Gregory Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Nguyen Chung, Leon Derczynski, et al. Rwkv: Reinventing rnns for the transformer era. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 784–789, 2018.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Yash Sarrof, Yana Veitsman, and Michael Hahn. The Expressive Capacity of State Space Models: A Formal Language Perspective. *Advances in Neural Information Processing Systems*, 2024.

Daria Soboleva, Faisal Al-Khateeb, Robert Myers, Jacob R Steeves, Joel Hestness, and Nolan Dey. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama, June 2023.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, et al. Learning to (learn at test time): RNNs with expressive hidden states. *arXiv preprint arXiv:2407.04620*, 2024.

Alexandre Torres. mamba.py: A simple, hackable and efficient Mamba implementation in pure PyTorch and MLX., 2024. URL `https://github.com/alxndrTL/mamba.py`.

Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

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

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated Linear Attention Transformers with Hardware-Efficient Training. In *Forty-first International Conference on Machine Learning*, 2024a.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing Linear Transformers with the Delta Rule over Sequence Length. *Advances in Neural Information Processing Systems*, 36, 2024b.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800, 2019.

# A Additional Background

In this section, we provide additional details on the notation used, the relationship between RNNs and regular languages, the assumption of finite-precision, and the decoder function.

## A.1 Notation

We dentoe with $\mathbb{C}, \mathbb{R}, \mathbb{N}$ the sets of complex, real and natural numbers respectively. We use lower case letters for scalar quantities (e.g. $x \in \mathbb{R}$), bold lower case letters for (column) vectors (e.g. $\boldsymbol{v} \in \mathbb{R}^n$), and bold upper case letters for matrices (e.g. $\boldsymbol{M} \in \mathbb{R}^{n \times d}$). Some functions with matrix (vector) outputs, such as $\boldsymbol{A}$ and $\boldsymbol{B}$ in Equation (1), are also bold upper (lower) case letters to put emphasis on the fact that they output matrices (vectors). We denote with $\|\boldsymbol{v}\|$ the euclidean norm of the vector $\boldsymbol{v} \in \mathbb{R}^n$. When $\boldsymbol{M} \in \mathbb{R}^{n \times d}$, $\|\boldsymbol{M}\|$ also refers to the euclidean norm, corresponding to the largest singular value. The vector $\boldsymbol{e}_i \in \mathbb{R}^n$ is the $i$-th vector of the canonical bases in $\mathbb{R}^n$, i.e. the one-hot vector with 1 only in the $i$-th component and 0 in the others.

We also define for a boolean $s$

$$\mathbf{1}\{s\} := \begin{cases} 1 \text{ if } s \text{ is true} \\ 0 \text{ if } s \text{ is false} \end{cases}.$$

We define $\mathrm{sigmoid}(x) := 1/(1+e^{-x})$ and $\mathrm{softplus}(x) := \ln(1+e^x)$.

We sometimes use regular expressions (see e.g. Hopcroft, 2001), to represent their corresponding regular language. So that e.g. $(11)^* = \{11\}^*$, where 11 is the set containing the word 11 and $*$ is the *Kleene star* operation, is the set of words containing the empty word $\epsilon$ and all the words with an even number of ones, while $1^m$ is the word containing 1 repeated $m$ times. A language is *star-free* if its regular expression does not contain the Kleene star.

## A.2 Regular Languages and Recurrent Neural Networks

**RNNs Can Recognize Any Regular Language** A layer of a general RNN can be formulated similarly to eq. (1) just by replacing the linear state update with a generic state-transition function $g$ as:

$$\boldsymbol{h}_t = g(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t), \quad \boldsymbol{h}_0 \in \mathbb{R}^n.$$

It is apparent that any FSA can be implemented by an RNN layer if $g$ is sufficiently expressive to model its state transition function.

**LRNNs Can Recognize Any Regular Language** As explained in (Liu et al., 2023, Appendix A.2) and in (Merrill et al., 2024, Theorem 5), we can always implement any FSA, and thus recognize any regular language, using matrix-vector multiplication and hence also a one layer LRNN by choosing $n = |Q|$, $\boldsymbol{H}_0 = (1, 0 \ldots, 0)^\top$ and by letting, for any $w \in \Sigma$, $\boldsymbol{B}(w) = 0$ and $\boldsymbol{A}(w) \in \mathbb{R}^{n \times n}$ the matrix with entries $\boldsymbol{A}(w)_{q',q} = \mathbf{1}\{\delta(w,q) = q'\}$. However, such construction cannot be implemented by modern LRNNs since in general $\boldsymbol{A}(w)$ can have norm greater than one and might not be symmetric. Both conditions are not allowed by the state-transition matrices in modern LRNNs (see e.g. the ones in Table 1).

## A.3 Finite Precision

For our positive results on LRNNs expressivity (Theorems 3 and 4), by finite precision we mean that since all quantities involved in the computations are a finite number, there exists a finite set $\mathbb{D} \subset \mathbb{R}$ that contains them and and thus we do not require computations to be done in the reals but we can use $\mathbb{D}$ as datatype. In particular, $\mathbb{D}$ does not depend on the length of the input sequence. In practice, such datatype is chosen beforehand, e.g. floating point numbers requiring a given number of bits, which may not capture all quantities in our constructions.

In our negative results of Theorems 1 and 2 instead, we can pick the finite set $\mathbb{D}$ arbitrarily, e.g. floating point numbers, and we also make the use of the function $\mathrm{cast} : \mathbb{R} \to \mathbb{D}$, that we extend to $\mathbb{C}$ by applying it separately to real and imaginary part and to vector and matrices by applying it element wise. The $\mathrm{cast}$ function is used because some computations of the state of the LRNN will

be allowed to be in infinite precision and then transformed to finite-precision using $\mathrm{cast}$ as specified in the proofs.

We believe that the finite precision setup is not only realistic but also allows a better focus on the drawbacks of modern LRNN. We note that for transformers, results usually rely instead on the notion of log-precision (Liu et al., 2023), meaning that the size of $\mathbb{D}$ grows logarithmically with the sequence length. This is mainly due to their limited expressivity compared to LRNNs. We also note that concerning the state-transition matrices of modern LRNNs (see Table 1), the values at the extremes of the eigenvalue range are technically not included (because of the use of the $\mathrm{sigmoid}$ and $\mathrm{softplus}$ functions). However, since we are working with finite precision, we can still include them by choosing the appropriate datatype $\mathbb{D}$, which also in practice includes key values such as $0$, $1$ and $-1$.

### A.3.1 The Decoder function

In (1), to compute the output $\hat{\boldsymbol{y}}_t$ from the state $\boldsymbol{H}_t$ and the vector $\boldsymbol{x}_t$, of an LRNN layer, we use the function $\mathrm{dec}$, to abstract away the computations that is done on $\boldsymbol{H}_t$ and $\boldsymbol{x}_t$, since they are not part of the recurrence. In this work, we do not consider the internal structure of $\mathrm{dec}$, but it usually contains a normalization and a feed-forward neural network and thus it can usually approximate any function.

In our negative results on LRNNs expressivity in Theorems 1 and 2 our choice of arbitrary decoder guarantees the stronger results. For our positive results instead we either do not consider the decoder (Theorem 3) or we make use of a linear decoder (Theorem 4). We point our that to recognize regular languages efficiently and with a smaller LRNN state it is beneficial to have a more powerful (non-linear) decoder, as in the case of word problems for cyclic or permutation groups. However, such decoder may be hard to approximate.

## B Parity and Modular Counting – Proofs

We report the full proofs for the theorems in Section 4.1.

### B.1 Proof of Theorem 1

The language $(11)^*$ contains all sequences with an even number of ones. An FSA recognizing the language, for the sequence $1^k$ will output $y_k = 1$ if $k$ is even and $y_k = 0$ if $k$ is odd. Consider an LRNN with one layer as in (1). We will prove that if the assumptions on the eigenvalues of $\boldsymbol{A}(1)$ are not satisfied, then there exists a $\overline{k} > 0$ such that for every $k \geq \overline{k}$, the finite-precision version of the state $\boldsymbol{H}_k$ corresponding to the sequence $1^k$ does not depend on $k$ and is equal to $\overline{\boldsymbol{H}}$. Hence, no matter the choice of $\mathrm{dec}$, also the finite-precision version of $\hat{\boldsymbol{y}}_k$ will not vary with $k$ and thus for some $k' \geq \bar{k}$, $\hat{\boldsymbol{y}}_{k'} \neq k' \mod 2 = y_{k'}$. An inductive argument can then be used for the case of LRNNs with multiple (finitely many) layers, using the fact that the input of the next layer will be constant for $k$ large enough, as the input of the first layers.

By unrolling the recursion in 1 we obtain a closed form expression for the state

$$\boldsymbol{H}_k = \sum_{i=1}^{k-1} \left( \prod_{j=i+1}^{k-1} \boldsymbol{A}(\boldsymbol{x}_j) \right) \boldsymbol{B}(\boldsymbol{x}_i) + \left( \prod_{i=1}^{k} \boldsymbol{A}(\boldsymbol{x}_i) \right) \boldsymbol{H}_0,$$

where we set $\prod_{j=k}^{k-1} \boldsymbol{A}(x_j) = \boldsymbol{I}$ to avoid clutter. We follow Merrill et al. (2024) and make the simplifying assumption that $\boldsymbol{H}_k$ is computed using the above expression by first evaluating all products involving the matrices $\boldsymbol{A}(\boldsymbol{x}_j)$ separately and in infinite precision, then casting them into finite precision, and finally executing the sum also in infinite precision and casting the result in finite precision. Hence, if we set $\boldsymbol{x}_1 \dots \boldsymbol{x}_t = 1^k$, we get the following the exact and finite precision expressions for the state at time $k$.

$$\boldsymbol{H}_k = \sum_{i=0}^{k-1} \boldsymbol{A}(1)^i \boldsymbol{B}(1) + \boldsymbol{A}(1)^k \boldsymbol{H}_0, \quad \widehat{\boldsymbol{H}}_k = \mathrm{cast}\left( \sum_{i=0}^{k-1} \mathrm{cast}\left(\boldsymbol{A}(1)^i \boldsymbol{B}(1)\right) + \mathrm{cast}\left(\boldsymbol{A}(1)^k \boldsymbol{H}_0\right) \right),$$

where cast is an operation which rounds matrices with complex values elementwise into finite-precision. In particular, we consider the case where both the real and imaginary parts are casted separately.

Using the Jordan Canonical Form (e.g. Horn & Johnson, 2012) we can write $\boldsymbol{A}(1) = \boldsymbol{P}\boldsymbol{J}\boldsymbol{P}^{-1}$, where $\boldsymbol{J}$ is block diagonal made of the Jordan blocks $\boldsymbol{J}_1, \dots, \boldsymbol{J}_s$ with $s \leq n$, $\boldsymbol{J}_i \in \mathbb{R}^{k_i \times k_i}$ and with corresponding complex eigenvalues $\lambda_1 \dots \lambda_s$. Such decomposition is useful because it allows to write matrix powers as

$$\boldsymbol{A}(1)^k = \boldsymbol{P}\boldsymbol{J}^k\boldsymbol{P}^{-1}, \quad \boldsymbol{J}_i^k = \begin{bmatrix} \lambda_i^k & \binom{k}{1}\lambda_i^{k-1} & \binom{k}{2}\lambda_i^{k-2} & \cdots & \cdots & \binom{k}{k_i-1}\lambda_i^{k-k_i+1} \\ & \lambda_i^k & \binom{k}{1}\lambda_i^{k-1} & \cdots & \cdots & \binom{k}{k_i-2}\lambda_i^{k-k_i+2} \\ & & \ddots & \ddots & \vdots & \vdots \\ & & & \ddots & \ddots & \vdots \\ & & & & \lambda_i^k & \binom{k}{1}\lambda_i^{k-1} \\ & & & & & \lambda_i^k \end{bmatrix}.$$

Therefore, to study $\lim_{k\to\infty} \boldsymbol{A}(1)^k$, we can study the behaviour of the elements of the Jordan blocks when $k \to \infty$. If $|\lambda_i| < 1$ then all elements of $\boldsymbol{J}_i^k$ converge to zero, since the exponential is faster than the binomial $\binom{k}{j}$ with fixed $j$. Thus $\lim_{k\to\infty} \boldsymbol{J}_i^k = 0$. If instead $\lambda_i \in \mathbb{R}$ and $\lambda_i > 1$, then all nonzero elements of the Jordan block diverge to $+\infty$. Finally, when $\lambda_i \in \mathbb{R}$ and $\lambda_i = 1$, the diagonal elements are $\lambda_i^k = 1$, while the other nonzero elements diverge to $\infty$. Therefore we have that if $|\lambda_i| < 1$ or $\lambda_i$ is real and positive then there exists $\overline{\boldsymbol{J}}_i \in \{0, 1, \infty\}^{k_i \times k_i}$ such that $\lim_{k\to\infty} \boldsymbol{J}_i^k = \overline{\boldsymbol{J}}_i$. Now, assume that for every $i$ either $|\lambda_i| < 1$ or $\lambda_i \in \mathbb{R}$ with $\lambda_i \geq 1$. Then, from the structure of the Jordan decomposition, each element of the matrices $\boldsymbol{A}(1)^k\boldsymbol{B}(1)$ and $\boldsymbol{A}(1)^k\boldsymbol{H}_0$ will be a linear combination (with complex coefficients) of sequences of real numbers with well defined limits (either $0, 1$ or $+\infty$), and thus, when $k \to \infty$ either converges to a point in $\mathbb{C}$ or diverges to a specific point in the complex infinity.

Now let $\widehat{\boldsymbol{C}}_k = \mathrm{cast}(\boldsymbol{A}(1)^k\boldsymbol{B}(1))$ and $\widehat{\boldsymbol{D}}_k = \mathrm{cast}(\boldsymbol{A}(1)^k\boldsymbol{H}_0)$, then since cast operates elementwise and has bounded and finite range we have that there exists $\tau \in \mathbb{N}$, $\widehat{\boldsymbol{C}} \in \mathbb{C}^{n\times d}$ and $\widehat{\boldsymbol{D}} \in \mathbb{C}^{n\times d}$ such that for every $k \geq \tau$, $\widehat{\boldsymbol{C}}_k = \widehat{\boldsymbol{C}}$ and $\widehat{\boldsymbol{D}}_k = \widehat{\boldsymbol{D}}$ and

$$\widehat{\boldsymbol{H}}_k = \mathrm{cast}\left(\sum_{i=0}^{\overline{k}-1} \widehat{\boldsymbol{C}}_i + \widehat{\boldsymbol{D}} + (k - \overline{k} + 1)\widehat{\boldsymbol{C}}\right).$$

Note that only the last term inside cast varies with $k$ and in particular each element of the matrix inside cast converges to a point in $\overline{\mathbb{C}}$, that is the union of $\mathbb{C}$ and the complex infinity. Therefore, since we are applying again the cast operation we obtain that there exists $\overline{\boldsymbol{H}} \in \mathbb{C}^{n\times d}$ and $\bar{k} \geq \tau$ such that for every $k \geq \bar{k}$ we have $\widehat{\boldsymbol{H}}_k = \overline{\boldsymbol{H}}$, which concludes the proof. □

### B.2 Proof of Theorem 2

Let $\widehat{\boldsymbol{H}}_k$ and $\hat{y}_k := \mathrm{cast}(\mathrm{dec}(\widehat{\boldsymbol{H}}_k, x_k))$ be the finite-precision versions of the state $\boldsymbol{H}_k$ and (scalar) output of a one-layer LRNN on the input $\boldsymbol{x} = x_1 \dots x_k = 1^k$. Let also $y_k = \mathbf{1}\{k \mod m = 0\}$ be the correct output recognizing the word. We will show that if the assumptions on the eigenvalues are not satisfied, there exist $\overline{\boldsymbol{H}}_1, \overline{\boldsymbol{H}}_2 \in \mathbb{C}^{n\times n}$, $\bar{y}_1, \bar{y}_2 \in \mathbb{R}^p$ and $\tau \in \mathbb{N}$ such that for all $k \geq \tau$

$$\widehat{\boldsymbol{H}}_k := \begin{cases} \overline{\boldsymbol{H}}_1 & \text{if } k \mod 2 = 0 \\ \overline{\boldsymbol{H}}_2 & \text{otherwise} \end{cases}, \quad \hat{y}_k = \begin{cases} \bar{y}_1 & \text{if } k \mod 2 = 0 \\ \bar{y}_2 & \text{otherwise} \end{cases} \tag{6}$$

where without loss of generality we take $\bar{y}_1, \bar{y}_2 \in \{0, 1\}$. If $\bar{y}_1 = \bar{y}_2$, then, similarly to parity, $\hat{y}_k = \hat{y}_{k+1}$ for all $k > \tau$, while if $k \mod m = m - 1$, then $1 = y_{k+1} \neq y_k = 0$. Otherwise if $\bar{y}_1 \neq \bar{y}_2$ then if we assume that $k \mod d = 1$ and $\hat{y}_k = y_k = 0$, then $1 = \hat{y}_{k+1} \neq y_{k+1} = 0$ since $m > 2$. This will prove the result for a one-layer LRNN. Then, we will proceed with the proof of finitely many layers.

The proof can proceed similar to Theorem 1. Indeed, using the $k$-th power formula for the Jordan Decomposition of the matrix $\boldsymbol{A}(1)$ with eigenvalues $\lambda_1, \dots, \lambda_s$ we can prove that if $1 \leq i \leq s$,

$|\lambda_i| < 1$ or $\lambda_i \in \mathbb{R}$ and $\lambda_i \geq 1$, then when $k \to \infty$ each element of the corresponding Jordan block of $\boldsymbol{A}(1)^k$ either converges to a single value or diverges to $+\infty$. If instead $\lambda_i \in \mathbb{R}$ and $\lambda_i \leq -1$, the diagonal element of the corresponding Jordan block takes the form $c_k = (-1)^k |\lambda_i|^k$, while the ones in above the diagonal diagonal take the form $z_k = \binom{k}{j}(-1)^{k-t}|\lambda_i|^{k-t}$ with $t, j \leq n$. It can be shown that if we let $\bar{c} \in \{1, \infty\}$, then

$$\lim_{k\to\infty} c_{2k} = \bar{c}, \quad \lim_{k\to\infty} c_{2k+1} = -\bar{c}, \quad \lim_{k\to\infty} z_{2k} = \infty, \quad \lim_{k\to\infty} z_{2k+1} = -\infty.$$

Therefore we can apply the same reasoning of Theorem 1 using the finite-precision assumption to show that there exist $\bar{\tau} \in \mathbb{N}, \overline{\boldsymbol{C}}_1, \overline{\boldsymbol{C}}_2, \overline{\boldsymbol{D}}_1, \overline{\boldsymbol{D}}_2 \in \mathbb{C}^{n\times d}$ such that for every $k \geq \tau$ we have

$$\widehat{\boldsymbol{C}}_k := \text{cast}(\boldsymbol{A}(1)^k \boldsymbol{B}) = \begin{cases} \overline{\boldsymbol{C}}_1 \text{ if } k \bmod 2 = 1 \\ \overline{\boldsymbol{C}}_2 \text{ if } k \bmod 2 = 0 \end{cases} \quad \widehat{\boldsymbol{D}}_k := \text{cast}(\boldsymbol{A}(1)^k \boldsymbol{H}_0) = \begin{cases} \overline{\boldsymbol{D}}_1 \text{ if } k \bmod 2 = 1 \\ \overline{\boldsymbol{D}}_2 \text{ if } k \bmod 2 = 0 \end{cases}$$

Finally if for simplicity we consider $\tau \bmod 2 = 0$, we have that for $2k \geq \tau$

$$\widehat{\boldsymbol{H}}_{2k} = \text{cast}\left(\sum_{i=1}^{\tau-1} \widehat{\boldsymbol{C}}_i + \left(k - \frac{\tau}{2} + 1\right)\overline{\boldsymbol{C}}_2 + \left(k - \frac{\tau}{2}\right)\overline{\boldsymbol{C}}_1 + k\overline{\boldsymbol{D}}_2\right)$$

$$\widehat{\boldsymbol{H}}_{2k+1} = \text{cast}\left(\sum_{i=1}^{\tau-1} \widehat{\boldsymbol{C}}_i + \left(k - \frac{\tau}{2} + 1\right)(\overline{\boldsymbol{C}}_2 + \overline{\boldsymbol{C}}_1) + k\overline{\boldsymbol{D}}_1\right)$$

where we note that the limit for $k \to \infty$ of the term inside cast is well defined. Thus there exist $\overline{\boldsymbol{H}}_1, \overline{\boldsymbol{H}}_2 \in \mathbb{C}^{n\times d}$ and $\bar{k} \geq \tau$ such that eq. (6) is satisfied, concluding the proof for the case of a single layer.

**Multiple Layers** Note that since for one layer we have two sequences (even and odd) of outputs converging in finite time, there exist $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^p$ such that for all $k \geq \bar{k}$ we have

$$\hat{\boldsymbol{y}}_{2k} = \boldsymbol{a}, \quad \hat{\boldsymbol{y}}_{2k+1} = \boldsymbol{b}.$$

Therefore, consider an additional layer that takes as input $\boldsymbol{x}_1^{(2)}, \dots, \boldsymbol{x}_k^{(2)}$, with $\boldsymbol{x}_i^{(2)} = \hat{\boldsymbol{y}}_i$ and outputs $\hat{\boldsymbol{y}}_1^{(2)}, \dots, \hat{\boldsymbol{y}}_k^{(2)}$ as

$$\boldsymbol{H}_k^{(2)} = \boldsymbol{A}^{(2)}(\boldsymbol{x}_k^{(2)})\boldsymbol{H}_{k-1}^{(2)} + \boldsymbol{B}^{(2)}(\boldsymbol{x}_k^{(2)}), \quad \hat{\boldsymbol{y}}_k^{(2)} = \text{dec}^{(2)}(\boldsymbol{H}_k^{(2)}, \boldsymbol{x}_k^{(2)})$$

without loss of generality assume for simplicity that $\bar{k} = 1$ and that $\hat{\boldsymbol{y}}_{2k}^{(2)} = \boldsymbol{a}$ and $\hat{\boldsymbol{y}}_{2k+1}^{(2)} = \boldsymbol{b}$. We also set $\boldsymbol{A}_1 = \boldsymbol{A}^{(2)}(\boldsymbol{a})$, $\boldsymbol{A}_2 = \boldsymbol{A}^{(2)}(\boldsymbol{b})$ and $\boldsymbol{B}_1 = \boldsymbol{B}^{(2)}(\boldsymbol{a})$, $\boldsymbol{B}_2 = \boldsymbol{B}^{(2)}(\boldsymbol{b})$ and $\boldsymbol{C}_1 = \boldsymbol{A}_1\boldsymbol{A}_2$, $\boldsymbol{C}_2 = \boldsymbol{A}_1\boldsymbol{B}_2 + \boldsymbol{B}_1$. Then we can write

$$\begin{aligned} \boldsymbol{H}_{2k}^{(2)} &= \boldsymbol{A}_1\boldsymbol{H}_{2k-1}^{(2)} + \boldsymbol{B}_1 = \boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{H}_{2k-2}^{(2)} + \boldsymbol{A}_1\boldsymbol{B}_2 + \boldsymbol{B}_1 \\ \boldsymbol{H}_{2k}^{(2)} &= \boldsymbol{C}_1\boldsymbol{H}_{2(k-1)}^{(2)} + \boldsymbol{C}_2 = \sum_{i=0}^{k-1} \boldsymbol{C}_1^i\boldsymbol{C}_2 + \boldsymbol{C}_1^k\boldsymbol{H}_0 \end{aligned}$$

Furthermore for the odd sequences of states we have

$$\boldsymbol{H}_{2k+1}^{(2)} = \boldsymbol{A}_2\boldsymbol{H}_{2k}^{(2)} + \boldsymbol{B}_2 = \sum_{i=0}^{k-1} \boldsymbol{A}_2\boldsymbol{C}_1^i\boldsymbol{C}_2 + \boldsymbol{C}_1^k\boldsymbol{H}_0 + \boldsymbol{B}_2.$$

We notice that the sequences $\boldsymbol{H}_{2k}^{(2)}$ and $\boldsymbol{H}_{2k+1}^{(2)}$ are in a form similar to $\boldsymbol{H}_k$ of the first layer and when allowing for real but possibly negative eigenvalues we can use the same reasoning using the powers of the Jordan canonical form to show that if we let $\widehat{\boldsymbol{H}}_{2k}^{(2)}$ and $\widehat{\boldsymbol{H}}_{2k+1}^{(2)}$ being their finite-precision counterparts, then there exist $\overline{\boldsymbol{H}}_1^{(2)}, \overline{\boldsymbol{H}}_2^{(2)}, \overline{\boldsymbol{H}}_3^{(2)}, \overline{\boldsymbol{H}}_4^{(2)} \in \mathbb{R}^{n\times d}$, $\bar{k}_2 \geq 0$ such that for every $k \geq \bar{k}$

$$\widehat{\boldsymbol{H}}_{2k}^{(2)} = \begin{cases} \overline{\boldsymbol{H}}_1^{(2)} \text{ if } 2k \bmod 2 = 0 \\ \overline{\boldsymbol{H}}_2^{(2)} \text{ if } 2k \bmod 2 = 1 \end{cases}, \quad \widehat{\boldsymbol{H}}_{2k+1}^{(2)} = \begin{cases} \overline{\boldsymbol{H}}_3^{(2)} \text{ if } (2k+1) \bmod 2 = 0 \\ \overline{\boldsymbol{H}}_4^{(2)} \text{ if } (2k+1) \bmod 2 = 1 \end{cases}.$$

Therefore, for $k \geq \bar{k}_2$, the the function $k \mapsto \overline{\boldsymbol{H}}_k^{(2)}$ will be periodic with period four and hence no matter the choice of $\mathrm{dec}^{(2)}$, also the function $k \mapsto \hat{\boldsymbol{y}}_k^{(2)}$ will be periodic with period 4. Consequently, with two layers one can recognize the language $(1^m)^*$ only when $m = 1$, $m = 2$, $m = 4$, since that is the only case where $k \mapsto y_k$ has a period which is a divisor of 4. We can extend this argument inductively to the case of an LRNN with $L$ layers, to say that there exists $\bar{k}_L \geq 0$ such that for every $k \geq \bar{k}_L$, if we let $\boldsymbol{y}_k^{(L)}$ be the output of the last layer, the function $k \mapsto \hat{\boldsymbol{y}}_k^{(L)}$ is periodic with period $2^L$ and thus it can recognize the language $(1^m)^*$ only when $2^L \mod m = 0$, which happens only when there exists $p \leq L$ such that $m = 2^p$ and hence $m$ is a power of two, ending the proof. □

# C Products of Generalized Householder Matrices – Proofs

We provide proofs for the results stated in Section 4.3.

## C.1 Proof of Proposition 1

**First item** It can be shown by noting that if $\boldsymbol{C} \in \mathcal{M}_1([-1,1])$, then $\|\boldsymbol{C}\| \leq 1$ and using the sub-multiplicative property of the euclidean norm, i.e the fact that $\|\boldsymbol{AB}\| \leq \|\boldsymbol{A}\|\|\boldsymbol{B}\|$.

**Second item** Note that any real matrix has a singular value decomposition. Hence we can write

$$\boldsymbol{M} = \boldsymbol{USV}^\top$$

with $\boldsymbol{U}, \boldsymbol{V} \in \mathbb{R}^{n\times n}$ orthogonal and $\boldsymbol{S} = \mathrm{Diag}(\sigma_1, \dots, \sigma_n)$ with $\sigma_i \in [0,1]$, since $\|\boldsymbol{M}\| \leq 1$. It follows from the $n$-reflections theorem[3] that we can write $\boldsymbol{U}$ and $\boldsymbol{V}$ as either the identity $I \in \mathcal{M}_1(\{1\})$ or the product of at most $n$ reflections, each of which is in $\mathcal{M}_1(\{-1\})$. Hence $\boldsymbol{U}, \boldsymbol{V} \in \mathcal{M}_n(\{-1,1\})$. We can also write the matrix $\boldsymbol{S}$ as the product of $n$ GH matrices as

$$\boldsymbol{S} = \boldsymbol{S}_1\boldsymbol{S}_2 \dots \boldsymbol{S}_n, \quad \boldsymbol{S}_i = \boldsymbol{I} - (1-\sigma_i)\boldsymbol{e}_i\boldsymbol{e}_i^\top$$

where $\boldsymbol{e}_i$ is the $i$-th element of the canonical basis of $\mathbb{R}^n$. Hence, $\boldsymbol{S} \in \mathcal{M}_n([0,1])$. The proof of the first part is concluded since we wrote each of $\boldsymbol{U}, \boldsymbol{S}, \boldsymbol{V}$ as a product of at most $n$ GH matrices. If $\boldsymbol{M}$ is orthogonal we apply the $n$-reflections theorem directly. While we note that if $\boldsymbol{M} = \boldsymbol{P} \in \{0,1\}^{n\times n}$ with $\boldsymbol{P}$ being a permutation matrix different from the identity, it can be written as products of at most $n-1$ *swaps*, i.e. permutation matrices permuting only two elements. Therefore we have that there exists an integer $k \leq n-1$ and indices $i_1, \dots, i_k$ and $j_1, \dots, j_k$ such that $i_l \neq j_l$ and

$$\boldsymbol{P} = \prod_{l=1}^{k-1} \boldsymbol{P}_{i_l j_l}, \quad , \boldsymbol{P}_{ij} = (I - 2\boldsymbol{v}_{ij}\boldsymbol{v}_{ij}^\top) \qquad v_{ijl} = \begin{cases} 1/\sqrt{2} & \text{if } l = i \\ -1/\sqrt{2} & \text{if } l = j \\ 0 & \text{otherwise} \end{cases},$$

where we set $\boldsymbol{v}_{ij} = (v_{ij1}, \dots, v_{ijn})$. Note that since $\|\boldsymbol{v}_{ij}\| = 1$, $\boldsymbol{P}_{ij} \in \mathcal{M}_k(\{-1\})$ with $k \leq n$. For the the case where $M = I$ we can use the fact that $\boldsymbol{I} \in \mathcal{M}_1(\{1\})$

**Third item** Let $\boldsymbol{N} = \boldsymbol{C}_1\boldsymbol{C}_2 \cdots \boldsymbol{C}_k \in \mathcal{M}_k((-1,1])$, with $\boldsymbol{C}_i = \boldsymbol{I} - \beta_i \boldsymbol{k}_i\boldsymbol{k}_i^\top$ with $\|k_i\| = 1$ and $\beta_i \in [0,2)$. If $\boldsymbol{N} = \boldsymbol{I}$ the statement is satisfied, otherwise, let $\mathcal{V} = \mathrm{span}\{\boldsymbol{k}_i : i \in \{1,\dots,k\}, \beta_i > 0\}$. Any unit vector $\boldsymbol{v} \in \mathbb{R}^n$ can then be written as $\boldsymbol{v} = \boldsymbol{v}_1 + \boldsymbol{v}_2$ with $\boldsymbol{v}_1 \in \mathcal{V}$, $\boldsymbol{v}_2 \in \mathcal{V}^\top$ and $\|\boldsymbol{v}_1\|, \|\boldsymbol{v}_2\| \leq 1$. Now, if $\boldsymbol{v}_1 = 0$, then $\boldsymbol{N}\boldsymbol{v} = \boldsymbol{v}$, and hence $\boldsymbol{v}$ is an eigenvector with eigenvalue 1. Instead, if $\boldsymbol{v}_1 \neq 0$, then there exists $i'$ such that $\beta_{i'} \in (0,2)$ and $\boldsymbol{v}^\top\boldsymbol{k}_{i'} = \boldsymbol{v}_1^\top\boldsymbol{k}_{i'} \in (0,1]$ and if $i' < k$ either $\beta_j = 0$ or $\boldsymbol{v}^\top\boldsymbol{k}_j = 0$ for all $j \in \{i'+1, \dots, k\}$. Moreover, we have that

$$\|\boldsymbol{C}_{i'}\boldsymbol{v}\|^2 = \|\boldsymbol{v} - \beta_{i'}\boldsymbol{k}_{i'}\boldsymbol{k}_{i'}^\top\boldsymbol{v}\|^2 = 1 - \beta_{i'}(2-\beta_{i'})(\boldsymbol{v}^\top\boldsymbol{k}_i)^2 < 1,$$

where the last line comes from the fact that $\min_{x\in[0,2]} x(2-x) = 1$ and is only reached at 0 and 2 while $\beta_{i'} \in (0,2)$. Therefore, since for every $i$, $\|\boldsymbol{C}_i\| \leq 1$ and the euclidean norm is sub-multiplicative we have

$$\|\boldsymbol{N}\boldsymbol{v}\| = \|\boldsymbol{C}_1\boldsymbol{C}_2 \dots \boldsymbol{C}_k\boldsymbol{v}\| = \|\boldsymbol{C}_1\boldsymbol{C}_2 \dots \boldsymbol{C}_{i'}\boldsymbol{v}\| \leq \|\boldsymbol{C}_1\| \cdots \|\boldsymbol{C}_i\boldsymbol{v}\| < 1.$$

[3] This is a specialization of the Cartan–Dieudonné Theorem to $\mathbb{R}^n$, see Theorem 3 in `https://faculty.uml.edu/dklain/orthogonal.pdf` for a proof.

Therefore, if $\boldsymbol{v}$ is also an eigenvector with eigenvalue $\lambda \in \mathbb{C}$, then $\|\boldsymbol{N}\boldsymbol{v}\| = |\lambda| < 1$. Hence, we proved that for every eigenvector with eigenvalue $\lambda$ either $\lambda = 1$ or $|\lambda| < 1$. It remains to show that all eigenvalues of $\boldsymbol{N} \in \mathcal{M}_k([0,1])$ are real. For $k = 1$ it follows due to $\boldsymbol{N}$ being symmetric, for $k \geq 2$ let $\boldsymbol{D} = \boldsymbol{C}_1\boldsymbol{C}_2 \cdots \boldsymbol{C}_{k-1}$ so that $\boldsymbol{N} = \boldsymbol{D}\boldsymbol{C}_k$ and let $\boldsymbol{v}$ be any eigenvector of $\boldsymbol{N}$ with eigenvalue $\lambda$ and $\|\boldsymbol{v}\| = 1$. Then it holds that

$$\boldsymbol{v}^\top \boldsymbol{C}_k \boldsymbol{N} \boldsymbol{v} = \lambda \boldsymbol{v}^\top \boldsymbol{C}_k \boldsymbol{v}.$$

Therefore if $\boldsymbol{v}^\top \boldsymbol{C}_k \boldsymbol{v} \neq 0$, then $\lambda = \boldsymbol{v}^\top \boldsymbol{C}_k \boldsymbol{N} \boldsymbol{v} / \boldsymbol{v}^\top \boldsymbol{C}_k \boldsymbol{v} \in \mathbb{R}$. Otherwise when $\boldsymbol{v}^\top \boldsymbol{C}_k \boldsymbol{v} = 0$ it follows that

$$\boldsymbol{v}^\top \boldsymbol{C}_k \boldsymbol{v} = \|\boldsymbol{v}\|^2 - \beta_k (\boldsymbol{k}_k^\top \boldsymbol{v})^2 = 1 - \beta_k (\boldsymbol{k}_k^\top \boldsymbol{v})^2 = 0,$$

which is true only if $\beta_k = 1$ and either $\boldsymbol{v} = \boldsymbol{k}_k$ or $\boldsymbol{v} = -\boldsymbol{k}_k$ and thus $\boldsymbol{C}_k \boldsymbol{v} = \pm \boldsymbol{C}_k \boldsymbol{k}_k = 0$ and hence $\lambda = 0$, which concludes the proof. □

### C.2 Proof of Theorem 3

We first recall the notion of group isomorphism. Two groups $(G, *)$ and $(H, \cdot)$ where $G, H$ are the sets and $\star$ and $\cdot$ are the associative operations, are isomorphic, if there exist a bijective map $f : G \to H$ such that for every $g \in G$, $h \in H$

$$f(g * h) = f(g) \cdot f(h).$$

We view the LRNN layer in eq. (1) as the FSA $\mathcal{A}_{\text{lin}} = (\Sigma, \mathcal{H}, \boldsymbol{H}_0, \delta_{\text{lin}})$, where $\delta_{\text{lin}}(\boldsymbol{H}, w) = \boldsymbol{A}(w)\boldsymbol{H} + \boldsymbol{B}(w)$, which is extended in the usual way, and $\mathcal{H} = \{\delta_{\text{lin}}(\boldsymbol{H}_0, \boldsymbol{w}) : \boldsymbol{w} \in \Sigma^*\}$. Since $\mathcal{T}(\mathcal{A})$ is a group, from Cayley's theorem we have that it is isomorphic to a subgroup of $S_n$, which is the set of permutations on a set of a set of $n$ elements. Furthermore, each element in $S_n$ can be represented as an $n \times n$ permutation matrix. Since in general $n \neq |Q|$, we cannot let $\mathcal{H}$ to be a set of one hot vectors each corresponding to states in $Q$. Instead, we let $\boldsymbol{H}_0 = (1, \ldots, n)^\top$, $\mathcal{P} \subset \{0,1\}^{n \times n}$ be the set of permutation matrices and define $\boldsymbol{B} \equiv 0$ and $\boldsymbol{A} : \Sigma \to \mathcal{P}$ to be the function mapping each letter $w \in \Sigma$ to the permutation matrix corresponding to $\delta(\cdot, w)$. With this choice we can see that the function $f : \mathcal{T}(\mathcal{A}_{\text{lin}}) \to \mathcal{T}(\mathcal{A})$ such that $f(\delta_{\text{lin}}(\cdot, \boldsymbol{w})) = \delta(\cdot, \boldsymbol{w})$ for every $\boldsymbol{w} \in \Sigma^*$ is one-to-one (biejctive), and from our choice of $\boldsymbol{H}_0$, also the map $h : \mathcal{T}(\mathcal{A}_{\text{lin}}) \to \mathcal{H}$ such that for every $\boldsymbol{w} \in \Sigma^*$, $h(\delta_{\text{lin}}(\cdot, \boldsymbol{w})) = \delta_{\text{lin}}(\boldsymbol{H}_0, \boldsymbol{w})$ is also bijective. Moreover, the map $\phi : \mathcal{T}(\mathcal{A}) \to Q$ such that $\phi(\delta(\cdot, \boldsymbol{w})) = \delta(q_0, \boldsymbol{w})$ is surjective because we consider states that are only reachable from $q_0$, i.e. $Q = \{\delta(q_0, \boldsymbol{w}) : \boldsymbol{w} \in \Sigma^*\}$. Hence if we set $g = \phi \circ f \circ h^{-1}$, then $g : \mathcal{H} \to Q$ is surjective and for every $w \in \Sigma$ and $\boldsymbol{H} \in \mathcal{H}$ we have that

$$g(\delta_{\text{lin}}(\boldsymbol{H}, w)) = \delta(g(\boldsymbol{H}), w)$$

Thus, we have shown that such LRNN implements $\mathcal{A}$ and it does so with finite precision because the entries of all vectors and matrices are bounded integers. The proof is concluded by noting that permutation matrices have euclidean norm equal to one and real entries.

Moreover, Let $k = \max_{w \in \Sigma} \sum_{q \in Q} \mathbf{1}\{\delta(q, w) \neq q\} = \max_{w \in \Sigma} \sum_{i=1}^{n} \mathbf{1}\{(\boldsymbol{A}(w)\boldsymbol{H}_0)_i = \boldsymbol{H}_{0,i}\}$ be the maximum number of displaced element of the permutation associated with the alphabet $\Sigma$. Then, we know that every $\boldsymbol{A}(w) \in \mathcal{M}_{k-1}(\{-1, 1\})$.

If in addition there exists $m \in \mathbb{N}$ such that $\mathcal{T}(\mathcal{A})$ is isomorphic to a subgroup of the cyclic group $\mathbb{Z}_m$ with elements $\{0, \ldots, m-1\}$, we can modify the construction above to use a smaller dimension. If $m = 2$, then $\mathbb{Z}_2$ has elements $\{0, 1\}$, and $\mathcal{A}$ implements the parity automaton. Thus, we can set $\boldsymbol{H}_0 = -1$, $\boldsymbol{A}(0) = 1$, $\boldsymbol{A}(1) = -1$ and $g(1) = 1$ while $g(0) = 1$, which means that we can use a scalar recursion. Otherwise, if $m \geq 3$, we can modify the construction above by setting $\boldsymbol{H}_0 = (1, 0)^\top$ and, if for simplicity we assume $\Sigma \in \{0, \ldots, m-1\}$, for every $w \in \Sigma$ we let $\boldsymbol{A}(w)$ be the $2 \times 2$ rotation matrix corresponding to $\delta(\cdot, w)$:

$$\boldsymbol{A}(w) = \boldsymbol{R}(\theta) = \begin{bmatrix} \cos\theta_w & -\sin\theta_w \\ \sin\theta_w & \cos\theta_w \end{bmatrix}, \quad \theta_w = \frac{2\pi w}{m},$$

such that $\boldsymbol{R}(\theta) \in \mathcal{M}_2(\{-1\})$ (from Proposition 1). □

### C.3 KRHON-RHODES THEOREM

Before presenting the proof for Theorem 4, we provide the statement for the landmark result of Krohn-Rhodes (Krohn & Rhodes, 1965), after giving the definition of cascade product of two FSA.

**Definition 1** (Cascade product)**.** *Given two FSA $\mathcal{A} = (\Sigma, Q, q_0, \delta)$ and $\mathcal{B} = (Q \times \Sigma, Q', q_0', \delta')$, we define the cascade product FSA as $\mathcal{C} = \mathcal{B} \circ \mathcal{A} = (\Sigma, Q \times Q', (q_0, q_0'), \delta'')$ where for any $w \in \Sigma$*

$$\delta''((q, q'), w) := (\delta(q, w), \delta(q', (q, w)))$$

**Theorem 5** (Krohn-Rhodes, Theorem 4 in Maler & Pnueli (1994))**.** *For every FSA $\mathcal{A} = (\Sigma, Q, q_0, \delta)$ there exists $s \leq 2^{|Q|}$ and a cascade product FSA $\mathcal{C} = \mathcal{A}^{(s)} \circ \cdots \circ \mathcal{A}^{(1)} = (\Sigma, Q^\times, q_0^\times, \delta^\times)$, with $\mathcal{A}^{(i)} = \left(\Sigma^{(i)}, Q^{(i)}, q_0^{(i)}, \delta^{(i)}\right)$, with $|Q^{(i)}| \leq |Q|$, and a function $\mathcal{W} : Q^\times \to Q$ such that for any $\boldsymbol{w} \in \Sigma^*$, $\delta(q_0, \boldsymbol{w}) = \mathcal{W}(\delta^\times(q_0^\times, \boldsymbol{w}))$ and each $\mathcal{A}^{(i)}$ is permutation-reset automaton, which means that for every $w^{(i)} \in \Sigma^{(i)}$, $\delta^{(i)}(\cdot, w^{(i)})$ is either a bijection (i.e. a permutation over $Q$) or constant, ie. $\delta(\cdot, w^{(i)}) = q(w) \in Q^{(i)}$.*

### C.4 PROOF OF THEOREM 4

We apply the Krohn-Rhodes theorem (Theorem 5) to write $\mathcal{A}$ as the cascade product FSA $\mathcal{C} = \mathcal{A}^{(s)} \circ \cdots \circ \mathcal{A}^{(1)}$ with each FSA $\mathcal{A}^{(i)} = \left(\Sigma^{(i)}, Q^{(i)}, q_0^{(i)}, \delta^{(i)}\right)$ being permutation-reset and we show how the LRNN can implement $\mathcal{C}$.

We now show how the $i$-th layer of the LRNN with the structure in 1 can implement $\mathcal{A}^{(i)}$.

Let $n = |Q^{(i)}|$ and without loss of generality assume that $\Sigma = \{1, 2, \ldots, |\Sigma|\}$ and $Q^{(i)} = \{1, 2, \ldots, n\}$ with $q_0^{(i)} = 1$. For every $w \in \Sigma^{(i)}$ we set $\boldsymbol{A}^{(i)}(w) \in \{0,1\}^{n \times n}$, $\boldsymbol{B}^{(i)}(w) \in \{0,1\}^n$ such that $q, q' \in Q^{(i)}$

$$\begin{aligned}
&\boldsymbol{A}^{(i)}(w)_{q',q} = \mathbf{1}\{\delta(q, w) = q'\}, \quad \boldsymbol{B}^{(i)}(w)_{q'} = 0, && \text{if } \delta^{(i)}(\cdot, w) \text{ is bijective, or} \\
&\boldsymbol{A}^{(i)}(w)_{q',q} = 0, && \boldsymbol{B}^{(i)}(w)_{q'} = \mathbf{1}\{q' = q(w)\}, \quad \text{if } \delta^{(i)}(\cdot, w) \equiv q(w).
\end{aligned}$$

Then, for every word $\boldsymbol{w}^{(i)} = w_1^{(i)} \ldots w_t^{(i)} \in \Sigma^{(i)*}$, we set $g : \mathbb{R}^n \to \mathbb{R}$, such that $g(x) = (1, \ldots, n)^\top x$ and

$$\begin{aligned}
\boldsymbol{H}_t^{(i)} &= \boldsymbol{A}^{(i)}(w_t^{(i)})\boldsymbol{H}_{t-1}^{(i)} + \boldsymbol{B}^{(i)}(w_t^{(i)}), \qquad \boldsymbol{H}_0^{(i)} = (1, 0 \ldots, 0)^\top \in \mathbb{R}^n \\
y^{(i)} &= \mathrm{dec}^{(i)}(\boldsymbol{H}_t^{(i)}, w_t^{(i)}) = (g(\boldsymbol{H}_t^{(i)}), w_t^{(i)}) = (\delta^{(i)}(q_0^{(i)}, \boldsymbol{w}^{(i)}), w^{(i)})
\end{aligned}$$

So that such construction implements $\mathcal{A}^{(i)}$. In addition, by letting $\boldsymbol{w} = w_1 \ldots w_t \in \Sigma^*$ be the input to the LRNN, i.e. $w_j^{(1)} = w_j$, and and setting the output of each layer as the input to the next, i.e. $w_j^{(i)} = y_j^{(i-1)}$ for $i \geq 2$, for the output of the last layer we get

$$\begin{aligned}
y_t^{(s)} &= \mathrm{dec}^{(s)}(\boldsymbol{H}_t, w_t^{(s)}) \\
&= (\delta^{(s)}(q_0^{(s)}, \boldsymbol{w}^{(s)}), y_t^{(s-1)}) \\
&= (\delta^{(s)}(q_0^{(s)}, \boldsymbol{w}^{(s)}), \delta^{(s-1)}(q_0^{(s-1)}, \boldsymbol{w}^{(s-1)}), y_t^{(s-2)}) \\
&= (\delta^{(s)}(q_0^{(s)}, \boldsymbol{w}^{(s)}), \ldots, \delta^{(1)}(q_0^{(1)}, \boldsymbol{w}), w_t) \in \mathbb{N}^{s+1},
\end{aligned}$$

where we removed the nested parenthesis for simplicity. Hence, the first $s$ elements of $y_t^{(s)}$ are exactly the output of the cascade FSA $\mathcal{C}$. Note that our construction can be implemented in finite precision, since we only used matrices/vectors with entries either in $\{0, 1\}$, requiring only one bit, or in $Q^{(i)} \subset \mathbb{N}$, that can also be implemented using finite precision with $|Q^{(i)}|$ integers, requiring $\log_2(|Q^{(i)}|)$ bits. Note that we can exclude the last element of $y_t^{(s)}$ to get a dimension $\mathbb{N}^s$.

It is also the case that $\left\|\boldsymbol{A}^{(i)}(w)\right\| \leq 1$ for every $w \in \Sigma^{(i)}$ since $\boldsymbol{A}^{(i)}(w)$ is either a permutation matrix ($\left\|\boldsymbol{A}^{(i)}(w)\right\| = 1$ ) or the zero matrix ($\left\|\boldsymbol{A}^{(i)}(w)\right\| = 0$). Also, for every permutation matrix $\boldsymbol{P} \in \{0,1\}^{n \times n}$ which permutes only $k \leq n$ elements we have that $\boldsymbol{P} \in \mathcal{M}_{k-1}(\{-1, 1\})$.

Furthermore, for the zero matrix we have

$$0 = \prod_{i=1}^{n} (I - \boldsymbol{e}_i \boldsymbol{e}_i^\top) \in \mathcal{M}_n(\{0\})$$

It follows that $\mathcal{A}^{(i)}(w) \in \mathcal{M}_n([-1, 1])$ for $i \in \{1, \dots, s\}$. □

# D EXPERIMENTS

## D.1 CHOMSKY HIERARCHY

Here, we provide details on the formal language tasks and experimental protocol of Section 5.1.

### D.1.1 DETAILS ON THE EXPERIMENTAL SETUP

Like Beck et al. (2024), we trained each model with sequence lengths ranging from 3 to 40 and evaluated on lengths from 40 to 256, to understand the length generalization capabilities. We compared mLSTM and sLSTM with two models: Mamba (Gu & Dao, 2023) and DeltaNet (Yang et al., 2024b). All models contain 2 blocks, with 4 heads for the xLSTM and DeltaNet models. We set the embedding and heads' dimension to 128 across all experiments. For Mamba and DeltaNet, we also enable the 1-D depthwise-separable convolution layer with kernel size equal to 4 after the query/key/value projection. We train each model using AdamW (Loshchilov & Hutter, 2019) without gradient clipping, using 3 different learning rates (1e-2, 1e-3, 1e-4), with 3 different seeds each. We pick the best based on the median of the 3 seeds for every learning rate value. We use a batch size of 1024 (except for sLSTM, where we use 512) and a cosine annealing learning rate schedule (Loshchilov & Hutter, 2017) (mininum learning rate: 1e-6) after 10% warm-up steps. The weight decay is set to 0.1 during training. We train on every task for 100k steps in total. At each training step, we make sure to generate a valid random sample from the task at hand (see below).

### D.1.2 DETAILS ON THE EVALUATED TASKS

In Section 5.1 we conducted empirical evaluations on 3 tasks –namely parity, modular arithmetic without brackets and with brackets – from various levels of the Chomsky Hierarchy, as proposed by Deletang et al. (2023) and similarly used in xLSTM (Beck et al., 2024). Details for each task are given below, where $|\Sigma|$ is the vocabulary size and $Acc_{rand}$ is the accuracy of random guessing:

- **Parity ($|\Sigma| = 2$, $Acc_{rand} = 0.5$).** The parity $y_t \in \{0, 1\}$ of a sequence of ones and zeros $\boldsymbol{x} = x_1 \dots x_t \in \{0, 1\}^t$ is equal to 1 (resp. 0) if the total number of ones in the sequence is odd (resp. even). It is equivalent to addition modulo 2, it can be computed by summing all previous values and then using the modulo 2 function as $y_t = (\sum_{i=1}^{t} x_i) \bmod 2$.
- **Modular Arithmetic w/o Brackets ($|\Sigma| = 10$, $Acc_{rand} = 1/(|\Sigma| - 5)$).** Given a set of special tokens $\Sigma_s = \{+, -, *, =, [\texttt{PAD}]\}$ and a modulus $m \geq 1$, we compute the remainder $y_t = \boldsymbol{x}\ mod\ m$, where $\boldsymbol{x} = x_1 \dots x_t \in \Sigma^t$ and $y_t \in \{1, \dots, m-1\}$. Here, $\Sigma = \Sigma_s \cup \{0, \dots, m-1\}$. In our experiments $m = 5$. An example sequence is as follows:

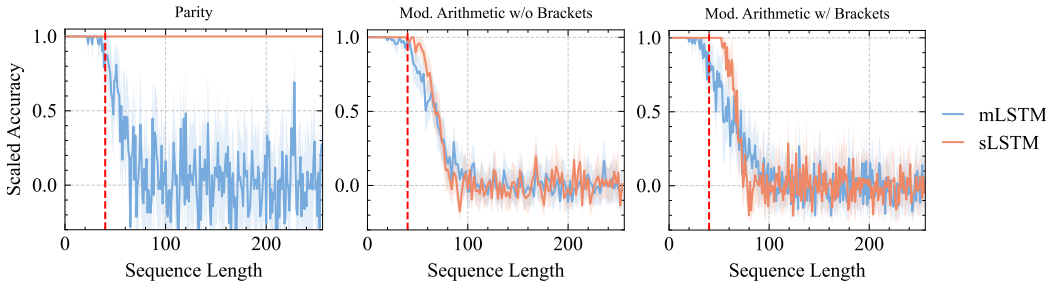
$$2 - 3 - 3 * 2 = 3\ [\texttt{PAD}]$$

- **Modular Arithmetic w/ Brackets ($|\Sigma| = 12$, $Acc_{rand} = 1/(|\Sigma| - 7)$).** Same definition as the modular arithmetic without brackets with a set of special tokens $\Sigma_s = \{+, -, *, =, ), (, [\texttt{PAD}]\}$. In our experiments $m = 5$. An example sequence is as follows:

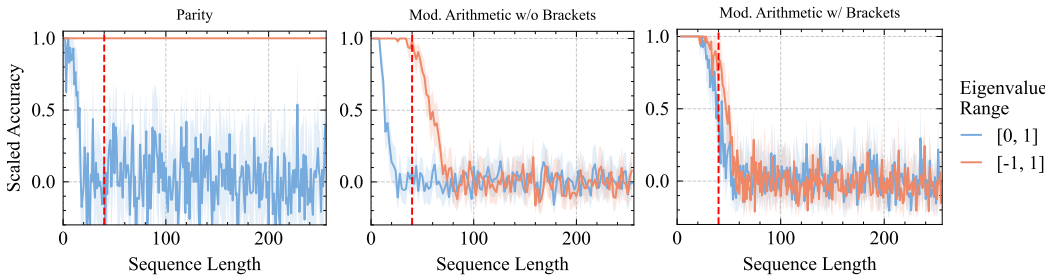
$$((((3 + 3) + -1) + -2) - ((3 - (-3)) + ((1) + 4))) = 2\ [\texttt{PAD}]$$

Table 5: Performance comparison of various recurrent models on regular and context-free language tasks. recurrent models on formal language tasks. We report the median ± median absolute deviation of 3 independent runs with different random seeds. Scores represent scaled accuracy, with 1.0 indicating perfect performance and 0.0 random guessing. The positive impact of allowing negative eigenvalues ($[-1, 1]$ range) versus restricting to positive eigenvalues ($[0, 1]$ range) is evident across different model architectures.

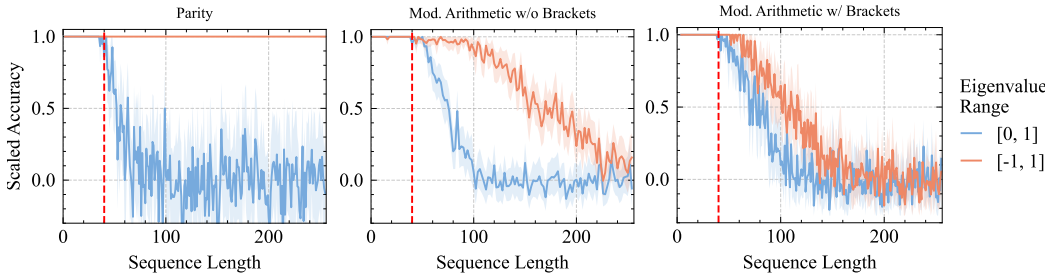
| | **Parity** | **Mod. Arithmetic (w/o brackets)** | **Mod. Arithmetic (w/ brackets)** |
|---|---|---|---|
| mLSTM | 0.018 ± 0.035 | 0.093 ± 0.028 | 0.097 ± 0.022 |
| sLSTM | 1.000 ± 0.000 | 0.130 ± 0.004 | 0.082 ± 0.003 |
| Mamba $[0, 1]$ | 0.000 ± 0.000 | 0.000 ± 0.005 | 0.016 ± 0.002 |
| Mamba $[-1, 1]$ | 1.000 ± 0.000 | 0.079 ± 0.032 | 0.029 ± 0.007 |
| DeltaNet $[0, 1]$ | 0.010 ± 0.005 | 0.126 ± 0.002 | 0.174 ± 0.008 |
| DeltaNet $[-1, 1]$ | 0.999 ± 0.006 | 0.422 ± 0.189 | 0.212 ± 0.008 |




(a) mLSTM and xLSTM




(b) Mamba




(c) DeltaNet

Figure 5: Performance (scaled accuracy) vs sequence length of *mLSTM*, *sLSTM*, *Mamba* and *DeltaNet* variants on different formal language tasks. Trained on sequences up to length 40 (dashed vertical red line). At test time, we sample uniformly at random 8192 sequences with lengths between 40 and 256. The curves show the mean and 95% CI.

### D.2 State-Tracking

#### D.2.1 Details of the Experiments

For the experiments in Section 5.2, we map each element of the group $S_5$ to an integer from $0$ to $119$, where $0$ corresponds to the identity permutation, and then construct inputs and output sequences of integers $x_1, \dots x_t$ and $y_1, \dots, y_t$ as follows

- $\mathbf{S_5}$ We sample $x_i$ uniformly at random from $\{0, \dots, 119\}$. $y_i$ is computed as the product of the permutations corresponding to $x_1, \dots, x_i$.
- $\mathbf{S_5}$ **only swaps** As $S_5$ but $x_i$ is sampled from the permutations that permute up to two elements (swaps and identity).
- $\mathbf{S_5}$ **swaps, 3-permutations** As $S_5$ but $x_i$ is sampled from the permutations that permute up to three elements.
- $\mathbf{S_5}$ **4 tokens per transition** If $i \mod 4 = 0$, then $x_i$ is sampled uniformly at random from $\{0, \dots, 119\}$, otherwise $x_i = 120$ (special token). For $i > 3$, $y_{i+3}$ is the product of the premutations corresponding to $x_1, \dots, x_i$, where 120 is treated as the identity permutation. $y_i = 0$ for $i \in \{1, 2, 3\}$.

For each input we also add a beginning of sequence token. For each setup we always sample $1.6$M examples for training and $40K$ examples of length $500$ for testing. We note that we are using a substantially larger training set compared to Merrill & Sabharwal (2023), to reduce the chances of overfitting.

We train all models using AdamW with weight decay $0.01$, learning rate $0.0001$, gradient clipping to $1.0$ and a batch size of $512$.

Both DeltaNet and Mamba models use an embedding dimension of $128$ and $4$ heads for DeltaNet. In the case of DeltaNet we do not use the 1-D convolutions for these experiments. Other parameters are kept as defaults.

**Full Matrix Baseline.** For the full matrix baseline we use a single layer and map directly each token $x_i$ to a learnable full state-transition matrix $\boldsymbol{A}(x_i) \in \mathbb{R}^{n \times n}$ via one-hot encoding. We then compute, for $i \in \{1, \dots, t\}$ the recursion

$$\boldsymbol{H}_i = \boldsymbol{A}(x_i)\boldsymbol{H}_{i-1}, \quad \boldsymbol{H}_0 = \boldsymbol{I} \in \mathbb{R}^{n \times n}$$

where $n$ is set to 32 for efficiency reason (memory and compute time grow quickly with $n$). After that we flatten each $\boldsymbol{H}_i$ into a vector and apply first a projection on the unit ball and then a linear decoder to get the final outputs. The projection was added to increase stability since we do not bound the norm of $\boldsymbol{A}(x_i)$. Since this model uses a full matrix, with $n \geq 5$ it should be fully able to learn $S_5$ without restricting the permutation in input. However in some situations the performance degrade quickly after some length, probably due to the fact that the norm of the learned $\boldsymbol{A}(x_i)$ is not close enough to one.

#### D.2.2 Cyclic Groups

We report in Figure 6 some experiments on group word problems with the group $\mathbb{Z}_{60}$. For this experiment we also consider the simplified version where each transition is encoded using 2 tokens. This is done as in the experiments of $S_5$ with 4 tokens, but using two tokens instead of $4$. Using one additional token should allow DeltaNet [-1,1] with more layers to learn the rotations needed to solve the task. However, using more tokens does not seem to help in this case. Extending the eigenvalue range seems to help in both settings, although surprizingly, Mamba [-1,1], even though it has a diagonal state-transition matrix, seems to perform best. We conjecture that in this case, the models might learn the shortcut solutions, also because they do not generalize very well to longer sequences.

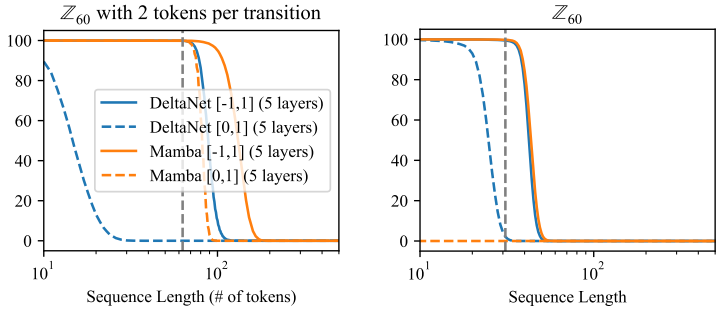



Figure 6: Validation sequence accuracy at different sequence lengths on the cyclic group $\mathbb{Z}_{60}$ (1 seed). Dashed vertical lines indicate the sequence length used for training (left 32, right 64). Using 2 tokens per transition seems to help only marginally in this case. Mamba [-1,1] is the best performing model. The variants with eigenvalues in [0,1] performed worse.

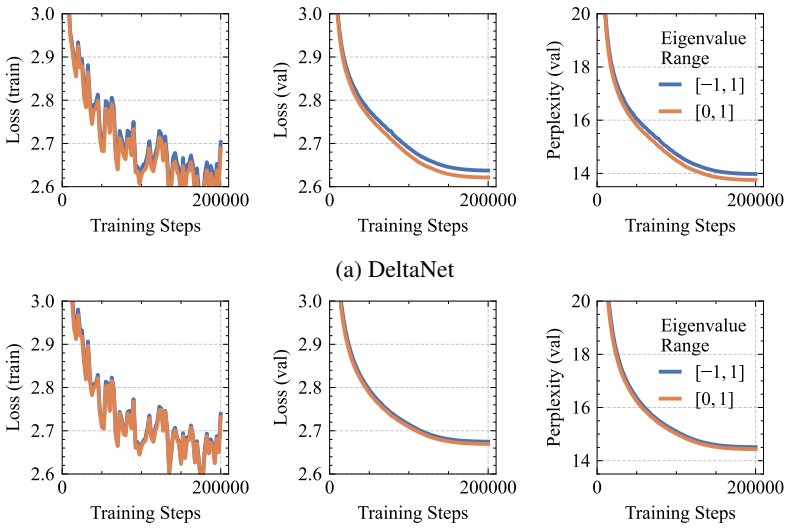
## D.3 Language Modeling

### D.3.1 Details on the experimental setup

Each model is trained for 200,000 steps with a per-GPU batch size of 10, distributed across 8 Nvidia A100 GPUs, using a context length of 2048. For optimization, we use AdamW (Loshchilov & Hutter, 2019) with learning rates of 3.1e-3 for DeltaNet and 5e-4 for Mamba (higher rates led to training instability). The learning rate was adjusted using cosine annealing (Loshchilov & Hutter, 2017) following a linear warm-up period of 5000 steps. We applied a weight decay of 0.1 throughout the training process.

### D.3.2 Details on the evaluated tasks

To produce the results in Table 4, we use the lm-harness benchmark (Gao et al., 2024), focusing on the same tasks as Yang et al. (2024b): LAMBADA (LMB) (Paperno et al., 2016), PIQA (Bisk et al., 2020), HellaSwag (Hella.) (Zellers et al., 2019), Winogrande (Wino.) (Sakaguchi et al., 2021), and ARC-easy (ARC-e) and ARC-challenge (ARC-c) (Clark et al., 2018). Additionally, we evaluate the performance on recall-intensive tasks (like Yang et al. (2024b)), including FDA (Arora et al., 2023), SWDE (Lockard et al., 2019), and SQUAD (Rajpurkar et al., 2018), to provide a comprehensive evaluation of our models' capabilities.




(a) DeltaNet

(b) Mamba

Figure 7: Learning curves of Mamba and DeltaNet when training on 32B tokens of Fine-Web 100B.

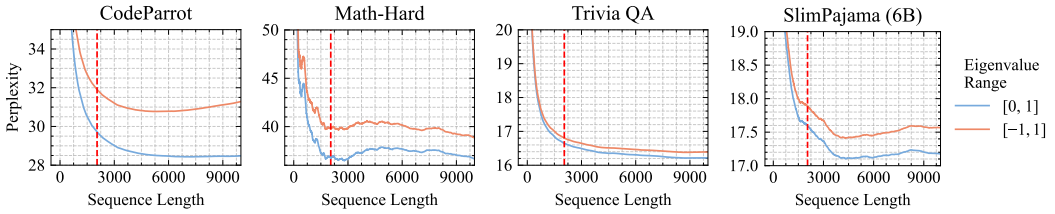



Figure 8: Length extrapolation performance of Mamba variants on different datasets. Mamba with eigenvalue range $[-1, 1]$ shows worse perplexity on coding and math tasks compared to the $[0, 1]$ baseline. The dashed, vertical line indicates the training context length of 2048 tokens.

### D.4 Implementation

We build on the original code for Mamba[4] and DeltaNet[5]. For DeltaNet, implementing the extended eigenvalue range is straightforward, since there is no need to modify the Triton kernel. However, Mamba requires modifications to the CUDA code of the associative scan for both forward and backward passes which however had no impact computational cost. We ensured the accuracy of the modifications by comparing the results with a naive implementation using a for-loop. For initial testing of the extended eigenvalue range, we used the pure pytorch implementation of Mamba by Torres (2024). We provide listings of the necessary code changes in Mamba and DeltaNet in Appendix D.4.1.

**Products in Log-space** We note that some diagonal models such as Mamba2 (Dao & Gu, 2024), GLA Yang et al. (2024a), mLSTM Beck et al. (2024) take advantage of the fact that all values of the state-transition matrices are positive to compute their repeated products in log-space. Our change would not allow to do this directly, and early tests on the chunkwise parallel form of GLA showed degraded performance. Therefore, for this work, we decided to focus on Mamba and DeltaNet since they do not compute the products in log-space. We mention however, that at the cost of increased computation time, it would be possible to do products in log-space by converting each value in the diagonal state-transition matrix to the product of its absolute value and sign. This way, absolute values can be multiplied in log space, while products of signs is coincidentally equivalent to addition modulo 2, i.e. parity, and hence can be done stably. We leave the investigation of this approach to future work. Furthermore, we also believe that our change may be less suited to method which use a normalized RNN state, such as mLSTM.

#### D.4.1 Implementation of Extended Eigenvalue Range

```
if constexpr (!kIsComplex) {
- thread_data[i] = make_float2(exp2f(delta_vals[r][i] * A_val[r]),
+ thread_data[i] = make_float2(2.0f * exp2f(delta_vals[r][i] * A_val[r]) - 1.0f,
                               !kIsVariableB ? delta_u_vals[r][i] : B_vals[i] * delta_u_vals[r][i]);
    if constexpr (!Ktraits::kIsEvenLen) {
        if (threadIdx.x * kNItems + i >= params.seqlen - chunk * kChunkSize) {
            thread_data[i] = make_float2(1.f, 0.f);
        }
    }
}
```

Figure 9: Modifications to the forward pass of the Mamba associative scan . These changes extend the eigenvalue range from $[0, 1]$ to $[-1, 1]$, enhancing the model's expressive capacity. Adapted from selective_scan_fwd_kernel.cuh.

[4] https://github.com/state-spaces/mamba

[5] https://github.com/sustcsonglin/flash-linear-attention

```
- const float delta_a_exp = exp2f(delta_vals[i] * A_scaled)
+ const float delta_a_exp = 2.0f * exp2f(delta_vals[i] * A_scaled) - 1.0f
```

```
- typename Ktraits::BlockScanT(smem_scan).InclusiveScan(
+ typename Ktraits::BlockScanT(smem_scan).ExclusiveScan(
                thread_data, thread_data, SSMScanOp<weight_t>(), prefix_op
            );
```

```
- const float a = thread_data[i].y - (!kIsVariableB ? delta_vals[i] * float(u_vals[i]) :
-        delta_vals[i] * float(u_vals[i]) * B_vals[i]);
+ float delta_a_exp = 2.0f * exp2f(delta_vals[i] * A_scaled) - 1.0f;
+ const float ddelta_a_exp = delta_a_exp + 1;
+ const float a = ddelta_a_exp * thread_data[i].y;
+ const float hi = delta_a_exp * thread_data[i].y + (!kIsVariableB ? delta_vals[i] *
+         float(u_vals[i]) :  delta_vals[i] * float(u_vals[i]) * B_vals[i]);
```

```
- const float a = thread_data[i].y - (!kIsVariableB ? delta_vals[i] * float(u_vals[i]) :
-        delta_vals[i] * float(u_vals[i]) * B_vals[i]);
+ float delta_a_exp = 2.0f * exp2f(delta_vals[i] * A_scaled) - 1.0f;
+ const float ddelta_a_exp = delta_a_exp + 1;
+ const float a = ddelta_a_exp * thread_data[i].y;
+ const float hi = delta_a_exp * thread_data[i].y + (!kIsVariableB ? delta_vals[i] *
+         float(u_vals[i]) :  delta_vals[i] * float(u_vals[i]) * B_vals[i]);
```

```
if constexpr (!kIsVariableB || !kIsVariableC) {
   if constexpr (!kIsVariableB) { // dBC\_val is dB\_val
- dBC_val += dout_vals[i] * (!kIsVariableC ? thread_data[i].y :  thread_data[i].y * C_vals[i]);
+ dBC_val += dout_vals[i] * (!kIsVariableC ? hi :  hi * C_vals[i]);
   } else { // dBC\_val is dC\_val
- dBC_val += dout_vals[i] * thread_data[i].y;
+ dBC_val += dout_vals[i] * thread_data[i].y;
   }
}
if constexpr (kIsVariableB) { dB_vals[i] = dx * delta_vals[i] * float(u_vals[i]); }
if constexpr (kIsVariableC) {
- dC_vals[i] = dout_vals[i] * (!kIsVariableB ? thread_data[i].y * B_val :  thread_data[i].y);
+ dC_vals[i] = dout_vals[i] * (!kIsVariableB ? hi * B_val :  hi);
}
```

Figure 10: Necessary changes to selective_scan_bwd_kernel.cuh

```
     if self.use_beta:
- beta = rearrange(self.b_proj(hidden_states), 'b l h -> b h l').sigmoid()
+ beta = 2 * rearrange(self.b_proj(hidden_states), 'b l h -> b h l').sigmoid()
     else:
        beta = q.new_ones(q.shape[0], q.shape[1], q.shape[2])
```

Figure 11: Simple modification to the beta calculation in DeltaNet (Source) allowing the extension of the eigenvalues to the range $[-1, 1]$ . The original implementation (in red) is replaced with an adjusted version (in green).