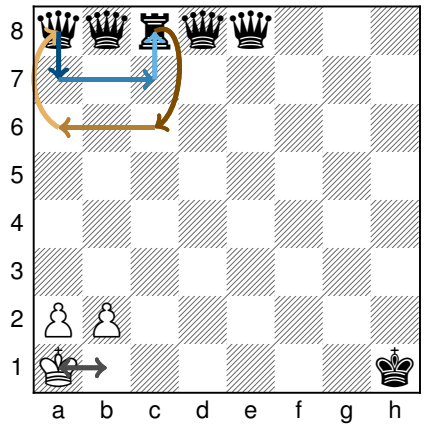
arXiv:2404.08819v2 [cs.LG] 4 Jun 2024

The Illusion of State in State-Space Models

William Merrill [1] **Jackson Petty** [1] **Ashish Sabharwal** [2]

Abstract

State-space models (SSMs) have emerged as a potential alternative to transformers. One theoretical weakness of transformers is that they cannot express certain kinds of sequential computation and state tracking (Merrill & Sabharwal, 2023a), which SSMs are explicitly designed to address via their close architectural similarity to recurrent neural networks. *But do SSMs truly have an advantage (over transformers) in expressive power for state tracking?* Surprisingly, the answer is no. Our analysis reveals that the expressive power of S4, Mamba, and related SSMs is limited very similarly to transformers (within TC^0), meaning these SSMs cannot solve simple state-tracking problems like permutation composition and consequently are provably unable to accurately track chess moves with certain notation, evaluate code, or track entities in a long narrative. To supplement our formal analysis, we report experiments showing that S4 and Mamba indeed struggle with state tracking. Thus, despite their recurrent formulation, the "state" in common SSMs is an illusion: S4, Mamba, and related models have similar expressiveness limitations to non-recurrent models like transformers, which may fundamentally limit their ability to solve real-world state-tracking problems. Moreover, we show that only a minimal change allows SSMs to express and learn state tracking, motivating the development of new, more expressive SSM architectures.




```
x = [0, 0, 1, 0, 0]
x[1], x[3] = x[3], x[1] # Swap 1, 3
```

Alice, Bob, Carl, Dan, and Emma each have a coin. All are dimes except Carl's. Alice and Carl trade coins.

Figure 1: We prove that SSMs, like transformers, cannot solve inherently sequential problems like permutation composition (S_5), which lies at the heart of state-tracking problems like tracking chess moves in source-target notation (see Section 3.2), evaluating Python code, or entity tracking. Thus, SSMs cannot, in general, solve these problems either.

Code: `http://jpetty.org/ssm-illusion`

1. Introduction

Recent theoretical work has shown that transformer architecture based models are incapable of expressing inherently sequential computation (Merrill & Sabharwal, 2023a). These results reveal a surprising limitation of transformers: they cannot express simple kinds of *state tracking* problems, such as composing sequences of permutations, which even simple recurrent neural networks (RNNs) can naturally express. In a different line of work, state space model (SSM) architectures (Gu et al., 2021; 2022a; Fu et al., 2023; Gu & Dao, 2023; Wang et al., 2024) have been introduced as an alternative to transformers, with the goal of achieving RNN-like expressive power for handling problems that are naturally stateful and sequential (Gu et al., 2021; 2022b). *But does the seemingly stateful design of SSMs truly enable them to solve sequential and state-tracking problems that transformers cannot?* If so, this would be a promising property of SSMs because state tracking is at the heart of large language model (LLM) capabilities such as tracking entities in a narrative

[1]New York University [2]Allen Institute for AI. Correspondence to: William Merrill <willm@nyu.edu>, Jackson Petty <petty@nyu.edu>, Ashish Sabharwal <ashishs@allenai.org>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

(Heim, 1983; Kim & Schuster, 2023), playing chess under certain notation[1], or evaluating code. This would motivate further research on SSM architectures and their deployment in the next generation of LLMs.

In this work, we show that the apparent stateful design of SSMs is an *illusion* as far as their expressive power is concerned. In contrast to the suggestion by Gu et al. (2021; 2022b) (and, perhaps, a broader belief in the community) that SSMs have expressive power for state tracking similar to RNNs, we prove theoretically that linear and Mamba-style SSMs, like transformers, cannot express inherently sequential problems, including state-tracking problems like composing permutations that RNNs can easily express. Further, our experiments confirm this prediction: both transformers and these SSMs cannot learn to compose permutations with a fixed number of layers, whereas RNNs can compose permutations with just a single layer. Our results imply that arguments that current SSMs have an advantage over transformers due to being "more recurrent" or capable of tracking state are misguided. In fact, the SSM architectures we consider are just as theoretically unequipped for state tracking and recurrent computation as transformers are.

We first establish the theoretical weakness of linear SSMs and near generalizations by proving they are in the complexity class L-uniform TC^0, which has been previously shown for transformers (Merrill & Sabharwal, 2023a). This implies these SSMs cannot solve inherently sequential problems (formally, problems that are NC^1-hard), including state-tracking problems like permutation composition (Liu et al., 2023). Permutation composition is a fundamental problem at the heart of many real-world state-tracking problems such as playing chess, evaluating code, or tracking entities in a narrative (Figure 1), implying solutions to these problems, too, cannot be expressed by SSMs, at least in the worst case.

At first glance, our results may appear to contradict Gu et al. (2021)'s claim that linear SSMs can simulate general recurrent models, which can express permutation composition. But the contradiction is resolved by a difference in assumptions: Gu et al. (2021) relied on *infinite depth* (number of layers) to show that SSMs could simulate RNNs. We, on the other hand, analyze the realistic setting with a bounded number of layers, under which we find that SSMs cannot simulate the recurrent state of an RNN and, in fact, suffer from similar limitations as transformers for state tracking.

Empirically, we find that S4 (Gu et al., 2022a) and S6 (Gu & Dao, 2023) SSMs, as well as transformers, do *not* learn to solve the permutation composition state-tracking problem with a fixed number of layers, while simple RNNs can do so with just one layer. This provides empirical support for our theoretical separation in expressive power for state tracking between SSMs and true recurrent models. We also find that both transformers and SSMs struggle compared to RNNs on state-tracking problems less complex than permutation composition where it is not known whether they can express a solution. Thus, in practice, SSMs may struggle not just on the hardest state-tracking problems like permutation composition but also on easier variants.

[1]The hardness of chess state tracking holds with (source, target) notation, but standard notation may make state tracking easier.

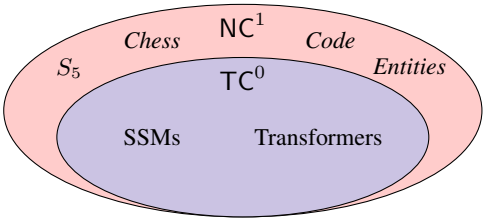


Figure 2: Complexity hierarchy within NC^1. Transformers can only recognize languages within TC^0 (Merrill & Sabharwal, 2023a), and we show the same for SSMs (Theorems 4.2 and 4.4). Thus, both architectures cannot express the "hard state tracking" captured by NC^1-complete problems like S_5, which *can* be straightforwardly expressed by RNNs. The figure assumes the widely held conjecture $\mathsf{TC}^0 \neq \mathsf{NC}^1$.

Finally, we consider a minimal extension of a linear SSM which makes the transition matrix input dependent, similar to Liquid S4 (Hasani et al., 2023). We show that this extension has sufficient expressive power for state tracking and permutation composition. Empirically, we show that our implementation of this extension learns to solve permutation composition with a single layer, just like an RNN, while being similarly parallelizable to other SSMs. It is an open question whether such SSM architectures with greater expressivity for state tracking are practically viable for large-scale language modeling.

2. Background

We first present the SSM architectures we will analyze (Section 2.1). Our analysis of the state tracking capabilities of SSMs borrows deeply from the circuit complexity and algebraic formal language theory literature. We thus review how circuit complexity can be used to analyze the power of neural networks (Section 2.3) and how state-tracking problems can be captured algebraically and analyzed within the circuit complexity framework (Section 3).

2.1. Architecture of State-Space Models

SSMs are a neural network architecture for processing sequences similar in design to RNNs or linear dynamical systems. SSMs have been suggested to have two potential advantages compared to transformers owing to their recur-

rent formulation: faster inference and, possibly, the ability to better express inherently sequential or stateful problems (Gu et al., 2021; 2022b). Several architectural variants of SSMs have been proposed, including S4 (Gu et al., 2022a) and Mamba (Gu & Dao, 2023). Recently, SSMs have been shown to achieve strong empirical performance compared to transformers in certain settings, particularly those involving a long context (Gu & Dao, 2023; Wang et al., 2024).

SSMs consist of *SSM layers*, which can be thought of as simplified RNN layers. We define a *generalized linear SSM layer* that encapsulates both S4 (Gu et al., 2022a) and the S6 layer used by Mamba (Gu & Dao, 2023) as special cases.

Definition 2.1 (Generalized linear SSM layer)**.** Given a sequence[2] $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^k$, the *recurrent form* of a linear SSM layer defines a new sequence of states $\mathbf{h}_1, \ldots, \mathbf{h}_n \in \mathbb{R}^d$ using projections $\bar{\mathbf{A}}_i \in \mathbb{R}^{d \times d}$ and $\bar{\mathbf{B}}_i \in \mathbb{R}^{d \times k}$, which can themselves depend on $\mathbf{x}_i$. For each $1 \leq i \leq n$,

$$\mathbf{h}_i = \bar{\mathbf{A}}_i \mathbf{h}_{i-1} + \bar{\mathbf{B}}_i \mathbf{x}_i. \qquad \text{(Recurrent form)}$$

The *convolutional form* of the SSM layer defines the same[3] $\mathbf{h}_1, \ldots, \mathbf{h}_n$ computed differently as a summation:

$$\mathbf{h}_i = \sum_{j=1}^{i} \left(\prod_{k=j+1}^{i} \bar{\mathbf{A}}_k \right) \bar{\mathbf{B}}_j \mathbf{x}_j. \qquad \text{(Convolutional form)}$$

The layer outputs $\mathbf{y}_i = \mathbf{C}_i \mathbf{h}_i + \mathbf{D}_i \mathbf{x}_i \in \mathbb{R}^k$, where $\mathbf{C}_i \in \mathbb{R}^{k \times d}$ and $\mathbf{D}_i \in \mathbb{R}^{k \times k}$ depend on $\mathbf{x}_i$.

Two common cases of this layer are when $\bar{\mathbf{A}}_i$ does not depend on the input ("non-gated"; Section 4.2) and when $\bar{\mathbf{A}}_i$ is diagonal (Section 4.3). In both of these cases, we will show that the SSM can be simulated in TC^0.

A **generalized linear SSM** is made up of multiple such layers, with a linear projection and a non-linearity applied after every layer (Rush & Karamcheti, 2022). Layer-norm can also be applied, either before or after the layer.

Practical Details. In S4 and related SSMs, Definition 2.1 is applied elementwise ($k = 1$) across all m elements of the previous layer output (Gu et al., 2022a). In practice, the weight matrix initialization is crucial for training. Our expressivity results (Theorems 4.2 and 4.4) apply for any generalized linear SSM (including S4 and S6), independent of initialization. In contrast to S4 and S6, H3 (Fu et al., 2023) does not meet Definition 2.1 because the context is not represented by a single vector. Rather, it resembles a transformer with SSM components.

[2] In practice, this sequence is often a vector $\mathbf{x}_1, \ldots \mathbf{x}_n \in \mathbb{R}^m$ and the SSM is applied elementwise on each feature.

[3] The two forms express the same function over $\mathbb{R}$ or any other distributive datatype. Over floating points (Section 2.2), they are not guaranteed to be the same, but we must assume the error is negligible for them to be well-defined and usable in practice.

2.2. Numeric Datatype

Circuit-complexity analysis of neural networks depends to some degree on low-level details about arithmetic and the underlying datatype $\mathbb{D}$ used in the network's computation graph. We can think of $\mathbb{D}$ as parameterized by the number of bits available to represent a number in $\mathbb{D}$. For instance, non-negative integers in $[0, 2^p]$ use p bits, signed integers in $[-2^p, 2^p]$ use $p + 1$ bits, FP16 uses 16 bits, etc.

Our main results (Theorems 4.2 and 4.4) will go through for any datatype $\mathbb{D}$ for which the following operations are efficiently parallel-computable (i.e., are in the complexity class L-uniform TC^0, to be defined shortly in Section 2.3):

1. Iterated addition, i.e., summing n numbers in $\mathbb{D}$
2. Iterated product, i.e., multiplying n numbers in $\mathbb{D}$
3. Matrix powering, i.e., computing the n-th power of a fixed-size $d \times d$ matrix over $\mathbb{D}$

When $\mathbb{D}$ is any finite-precision datatype, i.e., has a fixed number of bits available (e.g., 16 or 64), then these operations are easily seen to be in L-uniform TC^0. As Merrill & Sabharwal (2023b) argue, however, finite-precision datatypes severely limit the expressivity of neural architectures from a formal perspective (e.g., finite-precision transformers cannot represent uniform attention), motivating the use of parameterized datatypes that can (approximately) represent any number with a sufficiently large parameter. Interestingly, when $\mathbb{D}$ is the datatype of n-bit integers, all of the above operations are known to be in L-uniform TC^0 (Hesse, 2001; Mereghetti & Palano, 2000). Realistically, however, neural model implementations use floating point numbers with much fewer than n bits. Following Merrill & Sabharwal (2023b), we use the **log-precision floating point** model, i.e., $c \log n$ bit floats where c is some fixed constant (see Appendix A for a formal definition). Merrill & Sabharwal (2023a) showed that iterated addition over log-precision floats is in L-uniform TC^0. We extend the arguments of Hesse (2001) and Mereghetti & Palano (2000) to show that iterated product and matrix powering over log-precision floats are also in L-uniform TC^0 (see Appendix A).

2.3. Limits of Transformers via Circuit Complexity

A line of recent work has used circuit complexity and logic formalisms to identify expressivity limitations of transformers on reasoning problems (Angluin et al., 2023; Merrill & Sabharwal, 2023a; Liu et al., 2023; Chiang et al., 2023; Merrill & Sabharwal, 2023b; Hao et al., 2022); see Strobl et al., 2024 for a survey. In particular, Merrill & Sabharwal (2023a) showed transformers can only solve problems in the complexity class TC^0, which is the set of problems that can be recognized by constant-depth, polynomial-size threshold circuit families. Such circuits, in addition to having standard

AND, OR, and NOT gates (of arbitrary fan-in), can also use threshold gates that output 1 iff at least k of the inputs are 1, where k is a parameter of the gate. Informally, TC^0 can be thought of as the class of problems that can be solved with extremely parallel (constant-depth) computation.[4]

Problems outside TC^0, corresponding to problems that are inherently sequential and thus cannot be parallelized, cannot be solved by transformers. No problems in polynomial time are known unconditionally to be outside TC^0, but unless the widely held conjecture that $\mathsf{TC}^0 \neq \mathsf{NC}^1$ is false, many simple NC^1-hard problems are outside TC^0. In particular, this includes simulating finite automata (NC^1-complete), evaluating boolean formulas (NC^1-complete), determining graph connectivity (L-complete), and solving linear equations (P-complete). These problems have already been shown to be inexpressible by transformers (Merrill & Sabharwal, 2023a). By showing that SSMs can be simulated in TC^0, we will establish that they also cannot be solved by SSMs.

3. State Tracking

Informally, a state-tracking problem is a problem where the text specifies some sequence of updates to the state of the world, and the goal of the problem is to determine what the world state is after the updates have been applied in sequence. The circuit complexity view on the power of neural networks can be combined with other insights from algebraic formal language theory to analyze the kinds of state tracking that SSMs can express. In particular, this theory reveals which kinds of state-tracking problems are (likely) not in TC^0. This will, in turn, allow us to find examples of hard state tracking that models like SSMs cannot express.

3.1. State Tracking as a Monoid Word Problem

From the perspective of algebraic formal language theory, state tracking over a finite world can be captured as a *word problem* on a *finite monoid* (Liu et al., 2023).[5] Different updates to the world become different elements in the monoid, and resolving the final world state after all the updates have been applied is equivalent to computing the product of a sequence of elements (also called a "word").

Definition 3.1 (Word problem)**.** Let M be a finite set, and $(M, \cdot)$ a finite monoid (i.e., M with identity and associative multiplication). The word problem for M is to reduce sequences in M^* under multiplication; that is, send $m_0 m_1 \cdots m_k$ to $m_0 \cdot m_1 \cdot \ldots \cdot m_k \in M$. Solving the word problem requires reducing sequences of arbitrary length.

Example 3.2. Consider the monoid $\{0, 1\}$ where $\cdot$ is addition modulo 2. The word problem is to compute the parity of a string, e.g., $0011 \mapsto 0$. From a state-tracking perspective, this monoid captures a world with a single light switch. Identity 0 corresponds to no action, and 1 flips the switch.

Modeling state tracking with word problems lets us draw connections between circuit complexity and algebra to understand which word problems are hard to solve. Krohn & Rhodes (1965) established that not all word problems are created equal: some, like Example 3.2, are in TC^0, while others are NC^1-complete, requiring recurrent processing to solve (Immerman & Landau, 1989; Barrington, 1989). Because we will show SSMs can be simulated in TC^0, it follows that NC^1-complete state-tracking problems cannot be expressed by SSMs (cf. Figure 2).

Whether or not a word problem is NC^1-complete depends on the algebraic structure of the underlying monoid. Barrington (1989) showed that the word problem of every finite non-solvable[6] group is NC^1-complete. That non-solvable groups have NC^1-complete word problems is notable because of the ubiquity with which non-solvable groups show up in tasks involving state tracking. The canonical example of an NC^1-complete word problem is that of S_5, the symmetric group on five elements that encodes the permutations over five objects. As an immediate instantiation of this, consider a document describing a sequence of transpositions: *"swap ball 1 and 3, swap ball 3 and 5, swap ball 4 and 2, ..."*. Being able to answer the question *"where does ball 5 end up?"* for all possible swap sequences requires solving the S_5 word problem.[7] Beyond permutations, Figure 1 shows how many natural state-tracking problems like tracking chess moves, evaluating code, or tracking entities also encode the structure of S_5, meaning these state-tracking problems also cannot be expressed by a model in TC^0. Rather, in order to solve these problems, the depth of the model would have to be expanded to accommodate longer inputs.

Although the S_5 word problem is canonical, in this paper we will consider the word problem on a closely related group A_5: the *alternating* group on five elements. We do this for simplicity: A_5 is a subgroup of S_5 containing only even permutations, and is the smallest non-solvable subgroup. We will compare the word problem on A_5 to two other baseline groups: $A_4 \times \mathbb{Z}_5$, a non-abelian but solvable group;

[4] We use TC^0 to mean L-uniform TC^0, meaning the circuit family is constructible by a Turing machine that runs in space logarithmic in the size of the input (cf. Merrill & Sabharwal, 2023a; Strobl et al., 2024). We believe our results could be extended from L-uniform TC^0 to $\mathsf{DLOGTIME}$-uniform TC^0 using techniques similar to Merrill & Sabharwal (2023b) for composing TC^0 circuits in a way that preserves $\mathsf{DLOGTIME}$ uniformity.

[5] We consider finite monoids for simplicity, but the approach may be extendable to infinite (e.g., finitely generated) monoids.

[6] We focus on word problems on groups, which are monoids with inverses. Formally, a group G is solvable exactly when there is a series of subgroups $1 = G_0 < G_1 < \cdots < G_k = G$ such that G_{i-1} is normal in G_i and G_i / G_{i-1} is abelian.

[7] W.l.o.g., any permutation can be factored into a sequence of transpositions, or swaps.

and $\mathbb{Z}_{60}$, an abelian group encoding mod-60 addition. We choose these groups as points of comparison because they all have 60 distinct elements, meaning that the difficulty in learning their word problems will come only from the complexity of learning the group multiplication operation.

3.2. Encoding S_5 in Chess State Tracking

Figure 1 already gives some intuition into how state-tracking problems encode S_5. Out of these examples, the most intricated case is chess. We now give a proper reduction from S_5 to tracking chess moves, showing formally that not just S_5, but chess state tracking as well, is NC^1-complete. We define the chess state-tracking problem as follows:

- **Input:** A **chessboard state** and **sequence of chess moves**, where each move is written in UCI notation as a tuple (source square, target square). This differs from the standard SAN notation that represents other information like piece type (Toshniwal et al., 2021).
- **Output:** The resulting board state after starting in the initial board state and applying the sequence of moves one after another, ignoring draws. If any move is illegal given the previous board state, a null state is returned.

We show that S_5 can be reduced to chess state tracking, establishing its NC^1-completeness:

Proposition 3.3. *S_5 can be reduced to chess state tracking in UCI notation via NC^0 reductions.*

Proof. Without loss of generality, we consider the variant of S_5 where the output is true if and only if the original first element returns to the first position after the given sequence of permutations has been applied.

The idea, as illustrated in Figure 1, is to map each element of S_5 to a fixed sequence of chess moves that permutes five pieces accordingly on the chessboard. Given an instance of the S_5 word problem, we will construct an initial board state and a sequence of moves such that the final chessboard state encodes the output of that S_5 problem instance.

Let M denote the set of chess moves in the UCI, i.e., (source square, target square), notation.

Initial Board State. We construct a chessboard similar to Figure 1 but with a black rook at a8 and black queens at b8 to e8.

Chess Move Sequence. We then construct a finite function $f : S_5 \to M^*$ that encodes a permutation π as a sequence of chess moves. We first factor each permutation π to a sequence of transpositions $\tau_1(\pi) \cdots \tau_{m_\pi}(\pi)$. Each transposition $\tau \in T$ can in turn be expressed as a sequence of chess moves analogously to Figure 1. For example, transposing items 1 and 3 can be expressed as the move sequence: (a8, a7), (a1, b1), (c8, c6), (b1, a1), (a7, c7), (a1, b1), (c6, a6), (b1, a1), (c7, c8), (a1, b1), (a6, a8), (b1, a1), which has the crucial property that it transposes a8 with c8. We denote the mapping from transpositions to chess move sequences as $f : T \to M^*$. Putting it all together, we have

$$f(\pi) = \bigcirc_{j=1}^{m_\pi} f(\tau_j(\pi)).$$

To reduce a sequence of permutations $w \in S_5^*$, we let

$$f(w) = \bigcirc_{i=1}^{n} f(w_i).$$

Putting It All Together. We call our oracle for chess state tracking with the constructed initial board state and $f(w)$ as the sequence of chess moves. By construction, we can then return true if and only if the rook is at a8. The reduction can be implemented in NC^0 because it is a simple elementwise mapping of the input tokens, and decoding from the output chessboard is a finite table lookup. □

As a fun aside, we note that the chess board constructed in the above proof is reachable in a standard chess game. The chess sequences encoding permutation sequences are all valid in the game of chess, except that they ignore the fact that repeated board states in chess technically lead to a draw.

Since S_5 is NC^1-complete under AC^0 reductions and $\mathsf{NC}^0 \subseteq \mathsf{AC}^0$, we have:

Corollary 3.4. *The chess state-tracking problem is NC^1-complete under AC^0 reductions.*

Theorem 3.2 of Feng et al. (2023) uses a similar reduction to prove formula evaluation is NC^1-complete. Reductions can be constructed for evaluating Python or tracking entities in a dialog, as suggested by Figure 1. As for chess, the task formatting for entity tracking affects its hardness. For instance, the formatting used by Kim & Schuster (2023) in their Figure 1 is not NC^1-complete, whereas the variant shown in our Figure 1 is. This underscores the value of theory for constructing examples of hard state tracking.

4. SSMs Can be Simulated in TC^0

In this section, we show that the convolutional form of common variants of SSM can be simulated in TC^0. Assuming the convolutional form of the model computes the same function as the recurrent form, this implies such SSMs cannot solve inherently sequential problems, despite their appearance of recurrence and statefulness. We first show containment in TC^0 for non-gated SSMs (Theorem 4.2), and then show the same holds for diagonal SSMs (Theorem 4.4).

4.1. Conditions for Linear SSMs in TC^0

Before characterizing specific SSM architectures, we first show that the complexity of computing transition matrix products essentially determines the complexity of simulating an SSM with a circuit family.

Lemma 4.1. *Let M be a log-precision generalized linear SSM. Then there exists an L-uniform TC^0 circuit family that computes M's convolutional form if:*

1. *For any integer interval $[j, k]$, the matrix product $\prod_{i=j}^{k} \bar{\mathbf{A}}_i$ can be computed in L-uniform TC^0 as a function of $\bar{\mathbf{A}}_j, \ldots, \bar{\mathbf{A}}_k$ (to $c \log n$ precision for any $c > 0$).*
2. *For $1 \leq i \leq n$, $\bar{\mathbf{A}}_i, \bar{\mathbf{B}}_i, \mathbf{C}_i$, and $\mathbf{D}_i$ can be computed in L-uniform TC^0 as a function of $\mathbf{x}_i$.*

Proof. Following the proof structure of Merrill & Sabharwal (2023a), we describe how to construct a log-space bounded Turing machine T_M that, given $\mathbf{x}_1, \ldots, \mathbf{x}_n$ as input, prints a circuit that simulates M on this input. We first note that for all processing done before or after an SSM layer (projection, non-linearity, layer norm, etc.), T_M can follow known simulations of such operations for transformers (Merrill & Sabharwal, 2023a; 2024) to output a TC^0 circuit simulating this processing. We thus focus on simulating an individual SSM layer.

Recall from Definition 2.1 that M's convolutional form requires computing $\mathbf{h}_i = \sum_{j=1}^{i} \left(\prod_{k=j+1}^{i} \bar{\mathbf{A}}_k \right) \bar{\mathbf{B}}_j \mathbf{x}_j$ and $\mathbf{y}_i = \mathbf{C}_i \mathbf{h}_i + \mathbf{D}_i \mathbf{x}_i$. By the second precondition, T_M can print a TC^0 circuit that computes all matrices involved here. Further, by the first precondition, T_M can also print a TC^0 circuit that computes the innermost product in the computation of each hidden state $\mathbf{h}_i$, namely $\prod_{k=j+1}^{i} \bar{\mathbf{A}}_k$. It can now print a TC^0 circuit to multiply the resulting product[8] with $\bar{\mathbf{B}}_j$ and $\mathbf{x}_j$, and then print a circuit to compute an iterated sum over the i resulting vectors to compute $\mathbf{h}_i$ (cf. iterated addition in Appendix A). It can similarly print a (simpler) circuit to compute $\mathbf{y}_i$. Thus, the entire SSM layer can be simulated by an L-uniform TC^0 circuit. □

We will use Lemma 4.1 to show that any non-gated or diagonal generalized linear SSM can be simulated in TC^0.

4.2. Non-Gated SSMs are in TC^0

Theorem 4.2 (Non-gated SSM)**.** *Let M be a log-precision generalized linear SSM such that, for any i,*

$$\bar{\mathbf{A}}_i = \bar{\mathbf{A}}, \quad \bar{\mathbf{B}}_i = \bar{\mathbf{B}}, \quad \mathbf{C}_i = \mathbf{C}, \quad \mathbf{D}_i = \mathbf{D}.$$

Then there exists an L-uniform TC^0 circuit family that computes M's convolutional form.

Proof. We prove this by showing that both conditions from Lemma 4.1 are satisfied. Computing the matrix product reduces to powering $\bar{\mathbf{A}}^{k-j}$. Crucially, we can use the fact that matrix powering over floats is in L-uniform TC^0 (Lemma A.8, extending Mereghetti & Palano, 2000). Finally, $\bar{\mathbf{A}}_i, \bar{\mathbf{B}}_i, \mathbf{C}_i$, and $\mathbf{D}_i$ can be computed in L-uniform TC^0 because they are constants. □

As S4 satisfies the premises of Theorem 4.2, we obtain:

Corollary 4.3. *There exists an L-uniform TC^0 circuit family that computes S4's convolutional form.*

4.3. Diagonal SSMs are in TC^0

Theorem 4.4 (Diagonal SSM)**.** *Let M be a log-precision generalized linear SSM where for $1 \leq i \leq n$:*

1. *the transition matrix $\bar{\mathbf{A}}_i$ is diagonal, denoted $\mathrm{diag}(\bar{\mathbf{a}}_i)$ where $\bar{\mathbf{a}}_i \in \mathbb{R}^d$;*
2. *each of $\bar{\mathbf{a}}_i, \bar{\mathbf{B}}_i, \mathbf{C}_i$ and $\mathbf{D}_i$ can be computed in L-uniform TC^0 as a function of $\mathbf{x}_i$.*

Then there exists an L-uniform TC^0 circuit family that computes M's convolutional form.

Proof. By the first condition, $\prod_i \bar{\mathbf{A}}_i = \prod_i \mathrm{diag}(\bar{\mathbf{a}}_i)$. Iterated multiplication of diagonal matrices is reducible to several iterated scalar multiplications, placing this product in L-uniform TC^0 (Lemma A.5). The second condition from Lemma 4.1 is satisfied by assumption. Thus, M's convolutional form is computable in L-uniform TC^0. □

Since S6 satisfies the premises of Theorem 4.4, we have:

Corollary 4.5. *There exists an L-uniform TC^0 circuit family that computes S6's convolutional form (used by Mamba).*

Proof. For the first condition, note that S6's transition matrix $\bar{\mathbf{A}}_i$ is defined as $\exp(\delta_i \mathbf{A})$ for a fixed diagonal $\mathbf{A}$. The set of diagonal matrices is closed under scalar multiplication and matrix exponentiation, so $\bar{\mathbf{A}}_i$ is also diagonal. See Appendix B for a proof that the second condition is satisfied by the S6 parameterization. □

Appendix C extends Theorem 4.4 to hold even when $\{\bar{\mathbf{A}}_i\}$ are **simultaneously diagonalizable**, rather than just diagonal. Specifically, we prove the following generalization:

Theorem 4.6 (Simultaneously diagonalizable SSM)**.** *Let $\mathbf{W}$ be a fixed matrix. Let M be a log-precision generalized linear SSM such that, for $1 \leq i \leq n$,*

[8] Let $c \log n$ be the SSM's precision. We compute $\prod_k \bar{\mathbf{A}}_k$ to $c' \log n$ precision for a large enough c' (similar to the proof of Lemma A.5) such that the full product $\left(\prod_k \bar{\mathbf{A}}_k \right) \bar{\mathbf{B}}_j \mathbf{x}_j$ is correct to at least $c \log n$ bits, as technically required by Definition A.3.

1. *the transition matrix $\bar{\mathbf{A}}_i$ is computable to log precision by the expression $\mathbf{W}\,\mathrm{diag}(\bar{\mathbf{a}}_i)\mathbf{W}^{-1}$, where $\bar{\mathbf{a}}_i \in \mathbb{R}^d$;*
2. *each of $\bar{\mathbf{a}}_i, \bar{\mathbf{B}}_i, \mathbf{C}_i$ and $\mathbf{D}_i$ can be computed in L-uniform TC^0 as a function of $\mathbf{x}_i$.*

Then there exists an L-uniform TC^0 circuit family that computes M's convolutional form.

This, in turn, allows us to prove that a simultaneously diagonalizable transition matrix generalization of the S6 layer is also in L-uniform TC^0 (Corollary C.7).

4.4. Discussion

Theorems 4.2 and 4.4 establish that common SSM variants, like transformers, can only express solutions to problems in the class TC^0. This means these SSMs cannot solve NC^1-hard problems like evaluating boolean formulas or graph connectivity. In particular, it shows that they are limited as far as their state tracking capabilities as they are unable to compose permutations (solve the S_5 word problem):

Corollary 4.7. *Assuming $\mathsf{TC}^0 \neq \mathsf{NC}^1$, no log-precision SSM with the S4 or S6 architecture can solve the word problem for S_5 or any other NC^1-hard problem.*

In contrast, RNNs can easily express S_5 via standard constructions that encode finite-state transitions into an RNN (Minsky, 1954; Merrill, 2019). This shows that SSMs cannot express some kinds of state tracking and recurrence that RNNs can. This tempers the claim from Gu et al. (2021, Lemma 3.2) that SSMs have the expressive power to simulate RNNs, which relied on the assumption that SSMs can have *infinite depth*. In a more realistic setting with a bounded number of layers, our results show SSMs cannot express many state-tracking problems, including those which can be solved by fixed-depth RNNs.

5. Extending the Expressive Power of SSMs

We have shown that S4 and S6, despite their seemingly "stateful" design, cannot express problems outside TC^0, which includes state tracking like S_5. We show how SSMs can be extended to close the gap in expressive power with RNNs, allowing them to express S_5. Two simple extensions can bring about this increase in expressive power, assuming layer input dimension $k > 1$. First, adding a nonlinearity makes the SSM into an RNN, adding expressive power but degrading parallelism. On the other hand, allowing $\bar{\mathbf{A}}_i$ to be input-dependent makes the SSM more like a weighted finite automaton (WFA; Mohri, 2009), adding expressive power while remaining parallelizable.

5.1. Via Nonlinearities

One extension to the SSM is to add a nonlinearity, effectively making it an RNN. We call this an RNN-SSM layer:

$$\mathbf{h}_i = \mathrm{sgn}\left(\bar{\mathbf{A}}\mathbf{h}_{i-1} + \bar{\mathbf{B}}x_i\right).$$

A model with this architecture can solve the S_5 word problem when the input dimension $k > 1$:

Theorem 5.1. *For any regular language $L \subseteq \Sigma^*$ (including the word problem for S_5), there exists a one-layer log-precision RNN-SSM with $k = |\Sigma|$ that recognizes L.*

Proof. The standard constructions for simulating automata with RNNs (cf. Minsky, 1954; Merrill, 2019) apply. The condition $k = |\Sigma|$ comes from needing to represent token types with linearly independent vectors. □

Adding a nonlinearity to the output of an SSM layer (as in Mamba) is not the same thing as an RNN-SSM. Rather, an RNN-SSM applies the nonlinearity at each recurrent update. A downside of this approach is that it becomes nonlinear to parallelize the RNN-SSM computation graph with the SCAN algorithm used by linear SSMs (Blelloch, 1990).

5.2. Via Input-Dependent Transition Matrices

Another way to get greater expressive power is to let the transition matrix $\bar{\mathbf{A}}_i$ be fully input-dependent, as explored by Liquid S4 (Hasani et al., 2023). To illustrate this, we define a minimally different SSM called Input-Dependent S4 (IDS4) that achieves greater expressive power for state tracking. Let $\pi_{\mathbf{A}} : \mathbb{R}^k \to \mathbb{R}^{d\times d}$ be some affine transformation where the output vector is interpreted as a $d \times d$ matrix, and let $\bar{\mathbf{A}}_i = \pi_{\mathbf{A}}(\mathbf{x}_i)$. Let $\bar{\mathbf{B}}, \mathbf{C}, \mathbf{D}$ be fixed (w.r.t. i). By Definition 2.1, the IDS4 convolutional form computes an *iterated product* of non-diagonal, input-dependent matrices:

$$\mathbf{h}_i = \sum_{j=1}^{i} \left(\prod_{k=j+1}^{i} \pi_{\mathbf{A}}(\mathbf{x}_i) \right) \bar{\mathbf{B}}\mathbf{x}_j.$$

In contrast to matrix powers or iterated products of diagonal matrices, iterated products of *general* matrices cannot be computed in TC^0 (Mereghetti & Palano, 2000). This means that the arguments from Theorems 4.2 and 4.4 will not go through for IDS4. In fact, we can show IDS4 gains expressive power beyond TC^0:

Theorem 5.2. *For any regular language $L \subseteq \Sigma^*$ (including the word problem for S_5), there exists a one-layer log-precision IDS4 SSM with $k = |\Sigma|$ that recognizes $\$L$, where $\$ \notin \Sigma$ is a special beginning-of-string symbol.*

Proof. It suffices to show that IDS4 can simulate a deterministic finite automaton (DFA). We do this via a transition

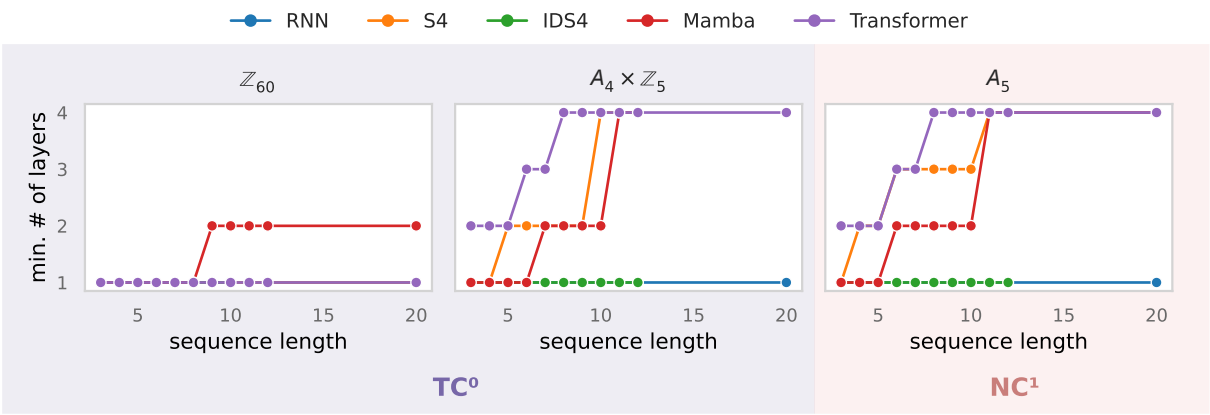


Figure 3: Minimum number of layers (lower is better) required to attain $> 90\%$ validation accuracy on group multiplication problems by sequence length and group. RNN and IDS4 models of constant depth can solve arbitrarily long sequences, while transformer, S4, and Mamba models require depths monotonically increasing in sequence length.

monoid construction. For any $w \in \Sigma^*$, let $\delta_w : Q \to Q$ be the function mapping a state to its eventual destination state after w is read from that state. For any DFA, this set of functions forms a finite monoid (the *transition monoid*) under composition, following from the Myhill-Nerode theorem (Hopcroft et al., 2001). Further, each monoid element δ_w can be represented as a boolean transition matrix, making matrix multiplication isomorphic to monoid composition. Computing the transition monoid of a DFA allows recognizing valid words: compute the monoid element for a word by multiplying the elements for its tokens and then check whether the initial state maps to an accepting state.

Fix a DFA and its transition monoid δ. To complete the proof, we show there exists an SSM that, for all $w \in \Sigma^*$, computes δ_w given input $x = \$w$. Let $\bar{\mathbf{A}}_i$ be the transition matrix representation of δ_{x_i}. Matrix multiplication is isomorphic to composition of transition monoid elements. We view indices in $\mathbf{h}_i$ as states and define $\bar{\mathbf{B}}\$$ as 1 at the initial state q_0 and 0 elsewhere. For other σ, let $\bar{\mathbf{B}}\sigma = \vec{0}$. This yields the following convolutional form:

$$\mathbf{h}_i = \left(\prod_{k=2}^{i} \bar{\mathbf{A}}_i\right) \mathbf{B}\$ \equiv \left(\bigcirc_{k=2}^{i} \delta_{x_k}\right)(q_0).$$

Since $x = \$w$, we conclude that $\mathbf{h}_{|x|} \equiv \delta_w(q_0)$. □

5.3. Discussion

Theorems 5.1 and 5.2 show that two minimal extensions of the SSM enable expressive power outside TC^0, allowing the model to solve hard state-tracking problems:

Corollary 5.3. *There exist a one-layer log-precision RNN-SSM and WFA-SSM that express the word problem for S_5 (with a beginning-of-string symbol), and these these SSMs cannot be simulated in TC^0.*

But would these variants of SSMs be feasible to use in practice? Besides expressive power, there are two competing practical concerns that might make these extensions problematic: parallelism and the impact on learning dynamics.

Parallelism. To be used effectively in an LLM, a model architecture must be parallelizable on practical hardware. Architectures in TC^0 are parallelizable by design (Merrill & Sabharwal, 2023a), but architectures in NC^1 may still be parallelizable to log depth even if they cannot be parallelized to constant depth. For IDS4, the bottleneck would be computing iterated matrix product with a log-depth computation graph. This could be achieved with the SCAN algorithm (Blelloch, 1990) similar to S4 and S6. In contrast, it is less clear how to parallelize a model with a nonlinearity.

Learning Dynamics. Another potential concern for IDS4 is that learning dynamics could be degraded. In particular, an iterated product of matrices may lead to vanishing or exploding gradients. However, this is already potentially an issue for the S6 architecture, where the selective gating involves computing an iterated product of scalars.

6. Can SSMs Learn Permutations in Practice?

Having established theoretical limitations of SSMs for state tracking, we empirically test how well SSMs can learn such tasks, focusing on the A_5 word problem. Since this problem is NC^1-complete and transformers, S4, and Mamba can only express functions in TC^0, these models should require a depth that grows with the input length to solve this problem.

Task. We model word problems (see Section 3.1) as a token-tagging task. Models are given as input a sequence $g_0 g_1 g_2 \cdots g_n$ drawn from one of A_5, $A_4 \times \mathbb{Z}_5$, or $\mathbb{Z}_{60}$. At each step i, the label is the product of the first i elements of

the sequence. Modeling the problem as a tagging task rather than as a sequence classification task provides the models with more supervision during training, making it as easy as possible to learn the correct function. We tokenize inputs such that each element gets a unique token.

Models. We train a transformer as a TC^0 baseline, an RNN that we expect can perform state tracking, and three SSMs: S4 (Gu et al., 2022a), Mamba (Gu & Dao, 2023), and IDS4 (Section 5.2). For IDS4, we initialize the affine projection α as a random normal centered around the identity: $\alpha(\mathbf{x}_i) \sim \mathbf{I} + \mathcal{N}(0, \sigma^2)$. This ensures that, at initialization, input-dependent transitions tend to propagate the previous state, which we expect to aid learning efficiency.

Experimental Setup. We train models on sequences of length n for successively larger values of n and report full-sequence accuracy on a test set.[9] To validate the prediction that SSMs and transformers require growing depth to solve longer A_5 word problems, we plot the minimum depth with 90% test accuracy as a function of input sequence length.

Results. Figure 3 shows single-layer RNN and IDS4 models learn the word problem for arbitrarily long sequences for all three groups. In contrast, transformer, S4, and Mamba models require depth monotonically increasing in sequence length to attain good test accuracy for the non-commutative groups. We draw three conclusions from this:

1. As expected, S4 and Mamba show the same limitations as transformers on the A_5 word problem. Longer A_5 sequences require deeper models, consistent with these models being in TC^0. In contrast, RNNs (Theorem 5.1) and IDS4 (Theorem 5.2) can efficiently solve the A_5 word problem.

2. Transformers, S4, and Mamba require greater depth even for $A_4 \times \mathbb{Z}_5$, which can be theoretically expressed by TC^0 circuits. Although transformer and Mamba models of a given depth perform as good or better on $A_4 \times \mathbb{Z}_5$ as they on A_5, they still require increasingly many layers to handle proportionally longer sequences. There are two possible interpretations of this. First, it could be that while these word problems are expressible in TC^0, they cannot be expressed by S4, Mamba, or transformers (which can each likely recognize only a proper subset of TC^0). On the other hand, it is possible that these word problems *are* expressible by transformers, S4, and Mamba but that effectively learning a constant-depth solution is difficult.

3. Despite this limitation, S4 and Mamba appear *empirically better* than transformer at approximate state tracking on the non-commutative tasks. For length-n sequences from $A_4 \times \mathbb{Z}_5$ or A_5, the transformer requires at least as many (and frequently more) layers as S4 or Mamba.

[9] We always include all 3600 pairwise sequences of length 2 in the training data along with the training split of length-n sequences.

7. Conclusion

We formally analyzed a family of generalized linear SSMs and showed that, like transformers, common SSM variants including S4 and Mamba can only express computation within the complexity class L-uniform TC^0 of highly parallel computations. This means they cannot solve inherently sequential problems like graph connectivity, boolean formula evaluation, and—of particular interest for state tracking—the permutation composition problem S_5. S_5 can be naturally expressed by true recurrent models like RNNs and captures the essence of hard state tracking due to its NC^1-completeness. In practice, one-layer RNNs can easily learn a task capturing S_5 while linear SSMs require depth growing with the sequence length. These results reveal that S4, Mamba, and related SSMs cannot truly track state: rather, they can only solve simple state-tracking problems for which shallow shortcuts exist (Liu et al., 2023).

On the other hand, we showed that an input-dependent SSM similar to Hasani et al.'s (2023) Liquid S4 can both express and learn the S_5 word problem, providing evidence that the expressiveness limitations of current SSMs can be overcome. Ultimately, this line of work could unlock new neural architectures that balance the parallelism of transformers and SSMs with full expressive power for state tracking, enabling LLMs that can benefit from scale while enjoying a greater capacity to reason about games, code, and language.

Impact Statement

This paper aims to advance the foundational understanding of state-space architectures for deep learning. Such work can affect the development and deployment of deep learning models in a variety of ways, which in turn can have societal impacts. However, we find it difficult to meaningfully speculate about or anticipate these downstream impacts here.

Acknowledgments

This work benefited from discussions with and valuable feedback from Chris Barker, Stefano Ermon, and Charles Foster. It was supported in part through the NYU IT High Performance Computing resources, services, and staff expertise. It was funded by NSF award 1922658, and WM was supported by an NSF graduate research fellowship, AI2, and Two Sigma.

References

Angluin, D., Chiang, D., and Yang, A. Masked hard-attention transformers and Boolean RASP recognize exactly the star-free languages, 2023. arXiv:2310.13897.

Barrington, D. A. Bounded-width polynomial-

size branching programs recognize exactly those languages in nc1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989. URL https://www.sciencedirect.com/science/article/pii/0022000089900378.

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

Chiang, D., Cholak, P., and Pillay, A. Tighter bounds on the expressivity of transformer encoders. In *ICML*, 2023.

Feng, G., Zhang, B., Gu, Y., Ye, H., He, D., and Wang, L. Towards revealing the mystery behind chain of thought: A theoretical perspective. In *NeurIPS*, 2023.

Fu, D. Y., Dao, T., Saab, K. K., Thomas, A. W., Rudra, A., and Re, C. Hungry hungry hippos: Towards language modeling with state space models. In *ICLR*, 2023.

Gu, A. and Dao, T. Mamba: Linear-time sequence modeling with selective state spaces, 2023. arXiv:2312.00752.

Gu, A., Johnson, I., Goel, K., Saab, K. K., Dao, T., Rudra, A., and Re, C. Combining recurrent, convolutional, and continuous-time models with linear state space layers. In *NeurIPS*, 2021.

Gu, A., Goel, K., and Re, C. Efficiently modeling long sequences with structured state spaces. In *ICLR*, 2022a.

Gu, A., Goel, K., Saab, K., and Ré, C. Structured state spaces: Combining continuous-time, recurrent, and convolutional models, January 2022b. URL https://hazyresearch.stanford.edu/blog/2022-01-14-s4-3. Blog post accessed January 31, 2024.

Hao, S., Angluin, D., and Frank, R. Formal language recognition by hard attention transformers: Perspectives from circuit complexity. *TACL*, 10:800–810, 2022.

Hasani, R., Lechner, M., Wang, T.-H., Chahine, M., Amini, A., and Rus, D. Liquid structural state-space models. In *ICLR*, 2023.

Heim, I. File change semantics and the familiarity theory of definiteness. *Semantics Critical Concepts in Linguistics*, pp. 108–135, 1983.

Hesse, W. Division is in uniform TC^0. In *International Colloquium on Automata, Languages, and Programming*, pp. 104–114, 2001.

Hesse, W., Allender, E., and Barrington, D. A. M. Uniform constant-depth threshold circuits for division and iterated multiplication. *J. Comput. Syst. Sci.*, 65:695–716, 2002.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. Introduction to automata theory, languages, and computation. *ACM SIGACT News*, 32(1):60–65, 2001.

Immerman, N. and Landau, S. The complexity of iterated multiplication. In *[1989] Proceedings. Structure in Complexity Theory Fourth Annual Conference*, pp. 104–111, 1989. doi: 10.1109/SCT.1989.41816.

Kim, N. and Schuster, S. Entity tracking in language models. In Rogers, A., Boyd-Graber, J., and Okazaki, N. (eds.), *ACL*, July 2023.

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

Liu, B., Ash, J. T., Goel, S., Krishnamurthy, A., and Zhang, C. Transformers learn shortcuts to automata. In *ICLR*, 2023.

Mereghetti, C. and Palano, B. Threshold circuits for iterated matrix product and powering. *RAIRO-Theor. Inf. Appl.*, 34(1):39–46, 2000. doi: 10.1051/ita:2000105. URL https://doi.org/10.1051/ita:2000105.

Merrill, W. Sequential neural networks as automata. In Eisner, J., Gallé, M., Heinz, J., Quattoni, A., and Rabusseau, G. (eds.), *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, Florence, August 2019. ACL.

Merrill, W. and Sabharwal, A. The parallelism tradeoff: Limitations of log-precision transformers. *TACL*, 11, 2023a.

Merrill, W. and Sabharwal, A. A logic for expressing log-precision transformers. In *NeurIPS*, 2023b.

Merrill, W. and Sabharwal, A. The expressive power of transformers with chain of thought. In *ICLR*, 2024.

Minsky, M. Neural nets and the brain-model problem. *Unpublished doctoral dissertation, Princeton University, NJ*, 1954.

Mohri, M. *Weighted Automata Algorithms*, pp. 213–254. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-01492-5. doi: 10.1007/978-3-642-01492-5_6. URL https://doi.org/10.1007/978-3-642-01492-5_6.

Reif, J. H. and Tate, S. R. On threshold circuits and polynomial computation. *SIAM Journal on Computing*, 21(5):896–908, 1992. doi: 10.1137/0221053. URL https://doi.org/10.1137/0221053.

Rush, S. and Karamcheti, S. The annotated S4. In *Blog Track at ICLR 2022*, 2022. URL https://openreview.net/forum?id=xDaLPsMBZv-.

Strobl, L., Merrill, W., Weiss, G., Chiang, D., and Angluin, D. What formal languages can transformers express? A survey. *TACL*, 12, 2024.

Toshniwal, S., Wiseman, S., Livescu, K., and Gimpel, K. Chess as a testbed for language model state tracking. In *AAAI*, 2021.

Wang, J., Gangavarapu, T., Yan, J. N., and Rush, A. M. Mambabyte: Token-free selective state space model, 2024. arXiv:2401.13660.

A. Floating-Point Arithmetic

Our results use the **log-precision floating point** model used by Merrill & Sabharwal (2023b) to analyze transformers. For some fixed constant $c \in \mathbb{Z}^+$, a $c \log n$ precision float is a tuple $\langle m, e \rangle$ where m, e are signed integers together taking $c \log n$ bits. Using $|x|$ to mean the number of bits used to represent integer x, this float represents the value $m \cdot 2^{e-|m|+1}$.

Unlike for integers, arithmetic operations over log-precision floats are not closed. That is, the product $\phi_1 \times \phi_2$ of two p-precision floats is a well-defined number but may not be exactly representable as a p-precision float. It is thus necessary to define approximate versions of these operations when formalizing log-precision floating-point arithmetic. To this end, Merrill & Sabharwal (2023a) define a natural notion of approximate iterated addition over log-precision floats and show that it is computable in L-uniform TC^0. We can naturally apply their definition of iterated addition for floats to matrices of floating points, defining iterated summation over matrices of datatype $\mathbb{D}$ as the result of treating the numbers as reals, performing exact arithmetic, and casting the exact output ϕ back to $\mathbb{D}$, denoted $\mathsf{cast}_{\mathbb{D}}(\phi)$. Formally:

Definition A.1 (Iterated $\mathbb{D}$-matrix sum; Merrill & Sabharwal, 2023a)**.** For matrices $\mathbf{M}_1, \ldots, \mathbf{M}_n$ over $\mathbb{D}$ with the same size, their *iterated* $\mathbb{D}$*-sum* is

$$\bigoplus_{i=1}^{z} \mathbf{M}_i \triangleq \mathsf{cast}_{\mathbb{D}} \left(\sum_{i=1}^{z} \mathsf{cast}_{\mathbb{R}}(\mathbf{M}_i) \right).$$

Here $\mathsf{cast}_{\mathbb{R}}$ converts a number in $\mathbb{D}$ to the corresponding real number. $\mathbb{D}$ is implicit in the notations $\mathsf{cast}_{\mathbb{R}}$ and $\bigoplus$. Integer addition can be obtained as a special case for 1-dimensional matrices. We can also analogously defined iterated summation, which will be necessary for formalizing SSMs:

Definition A.2 (Iterated $\mathbb{D}$-matrix product)**.** For square matrices $\mathbf{M}_1, \ldots, \mathbf{M}_z$ over $\mathbb{D}$, their *iterated* $\mathbb{D}$*-product* is

$$\bigotimes_{i=1}^{z} \mathbf{M}_i \triangleq \mathsf{cast}_{\mathbb{D}} \left(\prod_{i=1}^{z} \mathsf{cast}_{\mathbb{R}}(\mathbf{M}_i) \right).$$

Merrill & Sabharwal (2023a) showed that iterated addition from for log-precision floats is in L-uniform TC^0. It naturally follows from their argument that **iterated addition** over log-precision float matrices is also in L-uniform TC^0. In general, iterated matrix products are not necessarily computable in TC^0. However, we extend the arguments of Hesse (2001) and Mereghetti & Palano (2000) for integers to show that two special cases (iterated scalar multiplication and matrix powering) over log-precision floats are also computable in L-uniform TC^0.

Finally, we define a canonical value for a compositional arithmetic expression over floats that enjoys the associative property.

Definition A.3 (Flattened expression evaluation)**.** Let ϕ be a compositional expression over floats, which may contain alternating sums and products as well as other operations like $\exp$. We define the *canonical value* of ϕ as the value returned by the computation graph obtained by flattening all adjacent sums into a single sum (and analogously for products).

Definition A.3 has the nice effect of making Definition A.2 associative. The only results that rely on this assumption are our analysis of diagonalizable SSMs in Appendix C. We also deal with the details of this assumption in Lemma 4.1, though the proof there also goes through directly without handling these details.

A.1. Complexity of Iterated Scalar Multiplication

The first special case of iterated matrix products we analyze is when the matrices are simply scalars (or, w.l.o.g., diagonal matrices). In this case, the iterated product can be computed in L-uniform TC^0.

Lemma A.4 (Iterated $\mathbb{D}$-product)**.** *Let $\phi_1, \ldots, \phi_z \in \mathbb{D}$ be such that $z \leq n$ and each ϕ_i can be represented as an n-bit integer. If operators $\mathsf{cast}_{\mathbb{D}}$ and $\mathsf{cast}_{\mathbb{R}}$ are in L-uniform TC^0, then the iterated $\mathbb{D}$-product $\bigotimes_{i=1}^{z} \phi_i$ can be computed in L-uniform TC^0.*

Proof. By preconditions of the lemma, we can compute $y_i = \mathsf{cast}_{\mathbb{R}}(\phi_i)$ for each i in L-uniform TC^0. Since each ϕ_i is equivalent to an n-bit integer, y_i can be viewed as an n-bit integer. The iterated integer product $y = \prod_{i=1}^{z} y_i$ can be computed with an L-uniform TC^0 circuit (Hesse, 2001). Finally, by a precondition of the lemma, we can cast the result back to $\mathbb{D}$, i.e., compute $\mathsf{cast}_{\mathbb{D}}(y)$ which equals

the iterated $\mathbb{D}$-product $\bigotimes_{i=1}^{z} \phi_i$, with an L-uniform TC^0 circuit. □

Lemma A.5 (Iterated float product)**.** *Let $\phi_1, \ldots, \phi_z$ be $c \log n$ precision floats and $z \leq n$. Then the iterated float product $\bigotimes_{i=1}^{z} \phi_i$ can be computed in L-uniform TC^0.*

Proof. The idea is to convert (by scaling up) the sequence of ϕ_i to another sequence of floats that are all representable as integers, apply Lemma A.4, reverse the scaling, and cast the result back to a $c \log n$ precision float.

Let e be the smallest exponent across all ϕ_i and $q = \max\{0, -e\}$. Construct re-scaled floats $\psi_i = \phi_i 2^q$ by adding q to the exponent of ϕ_i, using up to $c \log n$ additional bits in the exponent if necessary to keep the computation exact. Note that e, q, and all ψ_i can easily be computed exactly by an L-uniform TC^0 circuit as they involve fixed-arity arithmetic operations. Further, by construction, every ψ_i has a non-negative exponent and thus represents an integer.

The maximum number representable by each $c \log n$ precision float ϕ_i is upper bounded by 2^{n^c}. Thus, the maximum number representable by each entry ψ_i is $2^{n^c} \times 2^q = 2^{n^c+q}$. Let $m = n^c + q$. It follows that each ψ_i can be equivalently represented as an m-bit integer. Further, this integer can be computed by left-shifting the mantissa of ψ_i by a number of bits equal to the value of the exponent of ψ_i (which is non-negative). Finally, this left-shift, and thus the $\mathsf{cast}_{\mathbb{R}}$ operation over m-precision floats, can be easily computed by an L-uniform threshold circuit of size $\mathrm{poly}(m)$. In the other direction, casting from reals to m-precision floats can also be easily accomplished by an L-uniform threshold circuit of size $\mathrm{poly}(m)$.

Observing that $\psi_1, \ldots, \psi_z$ is a sequence of floats each representable as an m-bit integer, we now apply Lemma A.4 with $\mathbb{D}$ being 'float' to conclude that iterated float product $\tau = \bigotimes_{i=1}^{z} \psi_i$ can be computed by an L-uniform threshold circuit of size $\mathrm{poly}(m)$. Since $m \leq 2n^c$, this circuit is also of size $\mathrm{poly}(n)$.

Finally, to compute the original iterated float product $\bigotimes_{i=1}^{z} \phi_i$, we divide τ by 2^{qz}. This can be accomplished by subtracting qz from the exponent of τ; again, we do this computation exactly, using up to $(c+1) \log n$ additional bits in the exponent if necessary. We then cast the resulting float back to a $c \log n$ precision float. All this can be done in L-uniform TC^0, finishing the proof that $\bigotimes_{i=1}^{z} \phi_i$ can be computed in L-uniform TC^0. □

A.2. Complexity of Matrix Powering

The second special case we analyze is matrix powering: i.e., a matrix product where all the matrices being powered are the same. Mereghetti & Palano (2000) showed that when the datatype $\mathbb{D}$ is n-bit integers, one can compute $\mathbf{M}^n$ in TC^0. We note that their construction also works for computing $\mathbf{M}^z$ for any $z \leq n, z \in \mathbb{Z}^+$. Further, as they remark, their construction can, in fact, be done in *uniform* TC^0. Specifically, we observe most of their construction involves sums and products of constantly many n-bit integers, which can be done in L-uniform TC^0. The only involved step is dividing a polynomial of degree (up to) n by a polynomial of degree (up to) $d-1$ and returning the remainder. It turns out that this "polynomial division with remainder" operation can also be performed in L-uniform TC^0 (see Corollary 6.5 of Hesse et al., 2002 and an explanation in Appendix A.3). We thus have the following extension of Mereghetti & Palano's result:

Lemma A.6 (Integer matrix power, derived from Mereghetti & Palano, 2000)**.** *Let $d \in \mathbb{Z}^+$ be a fixed constant. Let $\mathbf{M}$ be a $d \times d$ matrix over n-bit integers and $z \leq n, z \in \mathbb{Z}^+$. Then integer matrix power $\mathbf{M}^z$ can be computed in L-uniform TC^0.*

We extend this to matrix powers over $\mathbb{D}$ rather than integers:

Lemma A.7 ($\mathbb{D}$-matrix power)**.** *Let $d \in \mathbb{Z}^+$ be a fixed constant. Let $\mathbf{M}$ be a $d \times d$ matrix over a datatype $\mathbb{D}$ with entries equivalently representable as n-bit integers. Let $z \leq n, z \in \mathbb{Z}^+$. If operators $\mathsf{cast}_{\mathbb{D}}$ and $\mathsf{cast}_{\mathbb{R}}$ are in L-uniform TC^0, then $\mathbb{D}$-matrix power $\mathbf{M}^z$ can be computed in L-uniform TC^0.*

Proof. By preconditions of the lemma, we can compute $\mathsf{cast}_{\mathbb{R}}(\mathbf{M})$ in L-uniform TC^0. Since the entries of $\mathbf{M}$ are equivalent to n-bit integers, $\mathsf{cast}_R(\mathbf{M})$ can be viewed as a $d \times d$ integer matrix of n-bit integers. By Lemma A.6, we can compute $\mathsf{cast}_R(\mathbf{M})^z$ using an L-uniform TC^0 circuit. Finally, by a precondition of the lemma, we can cast the result back to $\mathbb{D}$, i.e., compute $\mathsf{cast}_{\mathbb{D}}(\mathsf{cast}_{\mathbb{R}}(\mathbf{M})^z)$ which equals $\mathbf{M}^z$, with an L-uniform TC^0 circuit. □

Lemma A.8 (Float matrix power)**.** *Let $d, c \in \mathbb{Z}^+$ be fixed constants. Let $\mathbf{M}$ be a $d \times d$ matrix over $c \log n$ precision floats. Let $z \leq n, z \in \mathbb{Z}^+$. Then float matrix power $\mathbf{M}^z$ can be computed in L-uniform TC^0.*

Proof. The idea is to convert (by scaling up) $\mathbf{M}$ to another float matrix all whose entries are representable as integers, apply Lemma A.7, reverse the scaling, and cast the result back to $c \log n$ precision floats.

Let e be the smallest exponent across all float entries of $\mathbf{M}$ and $q = \max\{0, -e\}$. Construct a re-scaled float matrix $\tilde{\mathbf{M}} = \mathbf{M} 2^q$ by adding q to the exponent of every entry of $\mathbf{M}$, using up to $c \log n$ additional bits in the exponent if necessary to keep the computation exact. Note that e, q, and $\tilde{\mathbf{M}}$ can easily be computed exactly by an L-uniform TC^0 circuit as they involve fixed-arity arithmetic operations. Further, by construction, $\tilde{\mathbf{M}}$ has non-negative exponents in

all its float entries. Thus, every entry of $\tilde{\mathbf{M}}$ represents an integer.

The maximum number representable by each $c \log n$ precision float in $\mathbf{M}$ is upper bounded by 2^{n^c}. Thus, the maximum number representable by each entry of $\tilde{\mathbf{M}}$ is $2^{n^c} \times 2^q = 2^{n^c+q}$. Let $m = n^c + q$. It follows that each entry ϕ of $\tilde{\mathbf{M}}$ can be equivalently represented as an m-bit integer. Further, this integer can be computed by left-shifting the mantissa of ϕ by a number of bits equal to the value of the exponent of ϕ (which is non-negative). Finally, this left-shift, and thus the $\mathsf{cast}_{\mathbb{R}}$ operation over m-precision floats, can be easily computed by an L-uniform threshold circuit of size $\mathrm{poly}(m)$. In the other direction, casting from reals to m-precision floats can also be easily accomplished by an L-uniform threshold circuit of size $\mathrm{poly}(m)$.

Note that $2^q \in [0, n^c]$ and hence $m \in [n^c, 2n^c]$. In particular, $m \geq n$. Thus $z \leq n$ (a precision) implies $z \leq m$. Observing that $\tilde{\mathbf{M}}$ is a matrix of floats each representable as an m-bit integer, we now apply Lemma A.7 with $\mathbb{D}$ being 'float' to conclude that float matrix power $\tilde{\mathbf{M}}^z$ can be computed by an L-uniform threshold circuit of size $\mathrm{poly}(m)$. Since $m \leq 2n^c$, this circuit is also of size $\mathrm{poly}(n)$.

Finally, to compute $\mathbf{M}^z$, we first divide each entry of $\tilde{\mathbf{M}}^z$ by 2^{qz}. This can be accomplished by subtracting qz from the exponent of each entry of $\tilde{\mathbf{M}}$; again, we do this computation exactly, using up to $(c+1)\log n$ additional bits in the exponent if necessary. We then cast all entries of the resulting matrix back to $c \log n$ precision floats. All this can be done in L-uniform TC^0, finishing the proof that $\mathbf{M}^z$ can be computed in L-uniform TC^0. □

A.3. L-Uniformity of Polynomial Division in TC^0

Hesse et al. (2002) state that polynomial division is in L-uniform TC^0 in Corollary 6.5. For historical reasons, this claim is preceded by weaker claims in older papers. We briefly clarify this situation to help understand why the stronger claim is valid.

Reif & Tate (1992) establish that polynomial division can be performed in P-uniform TC^0, whereas we state our results for L-uniform TC^0, which is a smaller class. However, the only issue preventing the polynomial division result from originally going through in the L-uniform case is that, at the time of Reif & Tate's publication, it was not known whether integer division and iterated integer multiplication are computable in L-uniform TC^0. However, Hesse (2001) later proved exactly this. Combining the two results, Theorem 3.2 of Reif & Tate (1992) goes through even with L-uniformity (not just P-uniformity). Its Corollary 3.3 then allows us to conclude that integer polynomial division can be solved by L-uniform TC^0 circuits because the output of integer polynomial division is an analytic function whose Taylor expansion has a finite number of terms (Reif & Tate, 1992).

B. S6 Parameterization

To justify that the S6 architecture used by Mamba is computable in TC^0, we justify that $\bar{\mathbf{A}}_i, \bar{\mathbf{B}}_i, \mathbf{C}_i, \mathbf{D}_i$ can be computed as a function of $\mathbf{x}_i$ in TC^0.

We begin by summarizing how exactly is S6 parameterized. S6 first defines continuous-time parameters:

1. $\mathbf{A}$ is a fixed, diagonal matrix that is invertible (each $a_{ii} \neq 0$);
2. $\mathbf{B}_i = \pi_{\mathbf{B}}(\mathbf{x}_i)$ is computed via a projection;
3. $\mathbf{C}_i = \pi_{\mathbf{C}}(\mathbf{x}_i)$ is computed via a projection;
4. $\mathbf{D}_i = \mathbf{I}$.

Next, we need to discretize the matrices $\mathbf{A}$ and $\mathbf{B}$. S6 does this using an input-dependent discretization factor δ_i:

$$\delta_i = \mathrm{softplus}(\delta + \pi_\delta(\mathbf{x}_i)).$$

The discretized matrices are then defined as:

$$\bar{\mathbf{A}}_i = \exp(\delta_i \mathbf{A})$$
$$\bar{\mathbf{B}}_i = (\delta_i \mathbf{A})^{-1} \left(\bar{\mathbf{A}}_i - \mathbf{I}\right) \delta_i \mathbf{B}_i.$$

It is clear to see that the diagonalizability condition of Theorem 4.4 is satisfied because $\bar{\mathbf{A}}_i$ itself is diagonal. Additionally, all the relevant matrices can be computed in TC^0.

Proposition B.1. *$\bar{\mathbf{A}}_i, \bar{\mathbf{B}}_i, \mathbf{C}_i$, and $\mathbf{D}_i$ can all be computed as functions of $\mathbf{x}_i$ in L-uniform TC^0.*

To prove this, observe that $\mathbf{A}, \mathbf{B}_i, \mathbf{C}_i, \mathbf{D}_i$ can all be computed in L-uniform TC^0 because they are either constants or linear transformations of $\mathbf{x}_i$. To justify that $\bar{\mathbf{A}}_i$ and $\bar{\mathbf{B}}_i$ can be computed in L-uniform TC^0, we just need to justify that we can invert diagonal matrices and compute $\mathrm{softplus}$ and $\exp$ in L-uniform TC^0.

Lemma B.2. *Diagonal matrices over log-precision floats can be inverted in L-uniform TC^0.*

Proof. Inverting a diagonal matrix just involves forming the reciprocal along the diagonal. Scalar reciprocals can be approximated to error at most 2^{-n^c} (for any c) in TC^0 (Hesse et al., 2002). This means we can compute the reciprocal of a log-precision float (cf. Appendix A) exactly up to log precision. □

In Appendix D, we show that we can compute the nonlinearities $\exp$ and $\mathrm{softplus}$ over a bounded domain in TC^0.

C. Diagonalizable SSMs

We extend Theorem 4.4 to cover the case when the SSMs transition matrices are *simultaneously diagonalizable*, rather than just diagonal. This requires us to note that when working with log-precision floating point representations of matrices, a diagonal matrix $\mathbf{A}$ and its diagonalized decomposition $\mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}$ are numerically substitutable.

Theorem 4.6 (Simultaneously diagonalizable SSM)**.** *Let $\mathbf{W}$ be a fixed matrix. Let M be a log-precision generalized linear SSM such that, for $1 \leq i \leq n$,*

1. *the transition matrix $\bar{\mathbf{A}}_i$ is computable to log precision by the expression $\mathbf{W}\,\mathrm{diag}(\bar{\mathbf{a}}_i)\mathbf{W}^{-1}$, where $\bar{\mathbf{a}}_i \in \mathbb{R}^d$;*
2. *each of $\bar{\mathbf{a}}_i, \bar{\mathbf{B}}_i, \mathbf{C}_i$ and $\mathbf{D}_i$ can be computed in L-uniform TC^0 as a function of $\mathbf{x}_i$.*

Then there exists an L-uniform TC^0 circuit family that computes M's convolutional form.

Proof. When the first condition is satisfied, the following equality holds over log-precision floats:

$$\prod_i \bar{\mathbf{A}}_i = \prod_i \left(\mathbf{W}\,\mathrm{diag}(\bar{\mathbf{a}}_i)\mathbf{W}^{-1}\right).$$

By the associativity of $\mathbb{D}$-matrix products, we can remove the parentheses to get

$$\begin{aligned}\prod_i \bar{\mathbf{A}}_i &= \prod_i \mathbf{W}\,\mathrm{diag}(\bar{\mathbf{a}}_i)\mathbf{W}^{-1} \\ &= \mathbf{W}\left[\prod_i \mathrm{diag}(\bar{\mathbf{a}}_i)\right]\mathbf{W}^{-1}.\end{aligned}$$

Iterated multiplication of diagonal matrices is reducible to several iterated scalar multiplications, which is in L-uniform TC^0 (Lemma A.5). Then the product of all $\bar{\mathbf{A}}_i$ is the product of three L-uniform TC^0-computable matrices, so is itself L-uniform TC^0-computable. The second condition from Lemma 4.1 is satisfied by assumption. Thus, the convolutional form for M can be computed in L-uniform TC^0. □

C.1. Diagonalizable S6

We can define an extension of S6 which satisfies these conditions to show that it is also in L-uniform TC^0.

Definition C.1. Diagonalizable S6 has continuous-time parameters:

1. $\mathbf{A}$ is a fixed matrix diagonalizable as $\mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}$ that is invertible (each $a_{ii} \neq 0$);
2. $\mathbf{B}_i = \pi_{\mathbf{B}}(\mathbf{x}_i)$ is computed via a projection;
3. $\mathbf{C}_i = \pi_{\mathbf{C}}(\mathbf{x}_i)$ is computed via a projection;
4. $\mathbf{D} = \mathbf{I}$.

As in the standard S6, the discretization of $\mathbf{A}$ and $\mathbf{B}$ is done by an input-dependent discretization factor δ_i:

$$\delta_i = \mathrm{softplus}(\delta + \pi_\delta(\mathbf{x}_i)).$$

The discretized matrices are then defined as

$$\begin{aligned}\bar{\mathbf{A}}_i &= \exp(\delta_i \mathbf{A}), \\ \bar{\mathbf{B}}_i &= (\delta_i \mathbf{A})^{-1}(\bar{\mathbf{A}}_i - \mathbf{I})\delta_i \mathbf{B}_i.\end{aligned}$$

To prove that $\bar{\mathbf{A}}_i$ and $\bar{\mathbf{B}}_i$ have the necessary properties, we first introduce some lemmas dealing with matrix-valued functions of diagonalizable matrices.

Lemma C.2. *If a matrix $\mathbf{A}$ is diagonalizable, then we can substitute its diagonalized decomposition $\mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}$ in a computation graph over log-precision floats involving $\mathbf{A}$ without incurring any meaningful error.*

Proof. Let $\mathbf{A}$ be diagonalizable. Then there exists invertible $\mathbf{W}$ and diagonal $\mathrm{diag}(\mathbf{a})$ such that $\mathbf{A} = \mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}$. Note that the product of a fixed number of matrices is in L-uniform TC^0, and so the first $c \log n$ bits of $\mathbf{A}$ and $\mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}$ are identical. □

Lemma C.3. *Let $\mathbf{A}$ be diagonalizable as $\mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}$, where $\mathbf{a} \in \mathbb{R}^d$. Then $c \cdot \mathbf{A}$ is simultaneously diagonalizable with $\mathbf{A}$ via $c \cdot \mathbf{A} = \mathbf{W}c \cdot \mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}$.*

Proof. Scalar multiplication commutes around matrix multiplication. □

Lemma C.4. *Let $\mathbf{A}$ be diagonalizable as $\mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}$, where $\mathbf{a} \in \mathbb{R}^d$. Then $\exp(\mathbf{A}) = \mathbf{W}\exp(\mathrm{diag}(\mathbf{a}))\mathbf{W}^{-1}$.*

Proof. The matrix exponential is defined as a power series, so for diagonalizable $\mathbf{A}$ it follows that

$$\begin{aligned}\exp(\mathbf{A}) &= \exp(\mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}) \\ &= \sum_{k=0}^{\infty} \frac{1}{k!}(\mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1})^k \\ &= \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{W}\,\mathrm{diag}(\mathbf{a})^k\mathbf{W}^{-1} \\ &= \mathbf{W}\left(\sum_{k=0}^{\infty} \frac{1}{k!}\,\mathrm{diag}(\mathbf{a})^k\right)\mathbf{W}^{-1} \\ &= \mathbf{W}\exp(\mathrm{diag}(\mathbf{a}))\mathbf{W}^{-1}.\end{aligned}$$

□

The expressions in Lemma C.4 are equivalent not just over real numbers but also over log-precision floats. This is because we know both expressions can be approximated in TC^0 with error at most 2^{-n^c}, which means the $c \log n$ bits of the approximation must be equivalent.

Lemma C.5. *Diagonalizable matrices over log-precision floats can be inverted in* L*-uniform* TC^0.

Proof. Let $\mathbf{A} = \mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}$. Then $\mathbf{A}^{-1} = \mathbf{W}^{-1}\,\mathrm{diag}(\mathbf{a})^{-1}\mathbf{W}$. We are guaranteed that each of these matrices exists, and furthermore by Lemma B.2 we know that $\mathrm{diag}(\mathbf{a})^{-1}$ is computable in L-uniform TC^0. Their product, involving a finite number of additions and multiplies, is also computable in L-uniform TC^0. □

Proposition C.6. $\bar{\mathbf{A}}_i$ *and* $\bar{\mathbf{B}}_i$ *can be computed as functions of* $\mathbf{x}_i$ *in* L*-uniform* TC^0.

Proof. We first show that $\bar{\mathbf{A}}_i$ is L-uniform TC^0 computable. By definition,

$$\bar{\mathbf{A}}_i = \exp(\delta_i \mathbf{A}).$$

By Corollary D.2, δ_i is computable in L-uniform TC^0. The product $\delta_i\mathbf{A}$ is simultaneously diagonalizable with $\mathbf{A}$ so

$$\begin{aligned}\bar{\mathbf{A}}_i &= \exp(\mathbf{W}\delta_i\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}) && \text{(Lemma C.3)}\\ &= \mathbf{W}\exp(\mathrm{diag}(\mathbf{a}))\mathbf{W}^{-1}. && \text{(Lemma C.4)}\end{aligned}$$

Since the exponential of scalars is L-uniform TC^0 computable by Corollary D.2, then $\bar{\mathbf{A}}_i$ is as well.

Turning to $\bar{\mathbf{B}}_i$, note that the term $(\delta_i\mathbf{A})^{-1}$ is L-uniform TC^0 computable by Lemma C.5 since $\delta_i\mathbf{A}$ is diagonalizable. Since $\bar{\mathbf{A}}_i$ is L-uniform TC^0 computable, the difference $\bar{\mathbf{A}}_i - \mathbf{I}$ is as well. Then every term in

$$\bar{\mathbf{B}}_i = (\delta_i\mathbf{A})^{-1}(\bar{\mathbf{A}}_i - \mathbf{I})\delta_i\mathbf{B}_i$$

is L-uniform TC^0 computable, and so their product is as well. □

Remark. Since $\mathbf{C}_i$ and $\mathbf{D}_i$ are unchanged between the standard and diagonalizable versions of S6, the proofs of their computability as functions of $\mathbf{x}_i$ in L-uniform TC^0 pass through from Appendix B.

Corollary C.7. *There exists an* L*-uniform* TC^0 *circuit family that computes Diagonalizable S6's convolutional form.*

Proof. Note that since $\mathbf{A} = \mathbf{W}\,\mathrm{diag}(\mathbf{a})\mathbf{W}^{-1}$ is fixed the set of transition matrices $\{\bar{\mathbf{A}}_i\}$ is simultaneously diagonalizable via $\mathbf{W}$ for all i.

Then Diagonalizable S6 meets the conditions for Theorem 4.6. □

D. Nonlinearities in L-Uniform TC^0

.

The parameterization of SSMs (and transformers) involves computing nonlinearities like exp and softplus. We leverage existing circuit complexity results (Reif & Tate, 1992) to show that, in general, any well-behaved nonlinearity should be computable in L-uniform TC^0 when used in conjunction with pre- or post-layer norm.

Lemma D.1 (Adapts Corollary 3.3, Reif & Tate, 1992)**.** *Let* $X = (-B, B)$ *be a bounded interval. Let* f *be a function over* X *with a convergent Taylor series:*

$$f(x) = \sum_{n=0}^{\infty} \frac{a_n}{b_n}(x - x_0)^n,$$

where a_n, b_n *are integers with magnitude at most* $2^{n^{O(1)}}$ *computable in* L*-uniform* TC^0*. Then* f *can be approximated over* X *by* L*-uniform* TC^0 *circuits to log precision (error at most* 2^{-n^c} *for any* $c \geq 1$*).*

Proof. Reif & Tate (1992) give a proof when $X = (-1, 1)$. We generalize to $X = (-B, B)$, assuming w.l.o.g. $B = 2^k$. The idea is to transform f to have domain $(-1, 1)$ via

$$g(x) = f(Bx).$$

Then, we can apply Corollary 3.3 of Reif & Tate (1992) to approximate g with error at most 2^{-n^c}. Reif & Tate (1992) state their result for P-uniform TC^0, but through advances in circuit complexity since the time of publication (Appendix A.3), their construction naturally applies for L-uniform TC^0 as well.

To approximate f, compute $z = x/B$, which can be done exactly since $B = 2^k$. We conclude by computing $g(z) = f(x)$, which, as established, has error at most 2^{-n^c}. □

Because of pre- and post-norm layers, the elements of $\mathbf{x}_i$ in an SSM will remain in a bounded domain $(-B, B)$. Thus, the following lemma shows we can compute them:

Corollary D.2. *The pointwise nonlinearities* exp, log, *and* softplus *are computable over* $(-B, B)$ *in* L*-uniform* TC^0.

Proof. By Reif & Tate (1992, Corollary 3.3) know that the Taylor series for exp and log is convergent with a_n, b_n computable in L-uniform TC^0. Then exp and log are themselves computable in L-uniform TC^0.

Since $\mathrm{softplus}(x) = \log(1 + \exp(x))$ is a fixed composition of L-uniform TC^0-computable functions, it too is computable in L-uniform TC^0. □