Advancing Regular Language Reasoning in Linear Recurrent Neural Networks

Ting-Han Fan*
Independent Researcher
tinghanf@alumni.princeton.edu

Ta-Chung Chi*
Carnegie Mellon University
tachungc@andrew.cmu.edu

Alexander I. Rudnicky
Carnegie Mellon University
air@cs.cmu.edu

Abstract

In recent studies, linear recurrent neural networks (LRNNs) have achieved Transformer-level performance in natural language and long-range modeling, while offering rapid parallel training and constant inference cost. With the resurgence of interest in LRNNs, we study whether they can learn the hidden rules in training sequences, such as the grammatical structures of regular language. We theoretically analyze some existing LRNNs and discover their limitations in modeling regular language. Motivated by this analysis, we propose a new LRNN equipped with a block-diagonal and input-dependent transition matrix. Experiments suggest that the proposed model is the only LRNN capable of performing length extrapolation on regular language tasks such as Sum, Even Pair, and Modular Arithmetic. The code is released at https://github.com/tinghanf/RegluarLRNN.

1 Introduction

There is a recent surge in the use of LRNNs (Gu et al., 2022; Peng et al., 2023; Orvieto et al., 2023) as alternatives to the de-facto Transformer architecture (Vaswani et al., 2017; Radford et al., 2019), which is ingrained in the field of natural language processing. LRNNs depart from the inter-timestep non-linearity design principle of classic RNNs (Elman, 1990; Jordan, 1997; Hochreiter and Schmidhuber, 1997; Cho et al., 2014), while at the same time: 1. achieving Transformer-level performance on the task of natural language modeling (Fu et al., 2023; Poli et al., 2023) and even better performance on synthetic long-range modeling tasks (Gu et al., 2022; Gupta et al., 2022; Orvieto et al., 2023; Hasani et al., 2023; Smith et al., 2023). 2. having the added benefits of fast parallelizable training (Martin and Cundy, 2018) and constant inference cost.

In spite of the remarkable empirical performance on natural language tasks, there has been no research on LRNNs' ability to model regular language. Regular language is a type of language that strictly follows certain rules like grammar.[1] The successful modeling of a regular language is important since it implies a model's ability to learn the underlying rules of the data. For example, if the training data are arithmetic operations such as $1 + 2 \times 3$, a model should learn the rules of $a + b$, $a \times b$, and that $\times$ has a higher priority than $+$. Learning unambiguous rules behind the data is a critical step toward sequence modeling with regulated output.

In this paper, we aim to determine if existing LRNNs are competent to learn the correct grammar of regular language by testing their language transduction capability under the length extrapolation setting. Concretely, a model is trained only to predict the desired outputs on a set of short sequences of length L_{tr}. It then needs to predict the correct outputs for longer testing sequences of length $L_{ex} \gg L_{tr}$. Adopting the length extrapolation setting is essential to mitigate the risk of a model learning spurious shortcut solutions (Liu et al., 2023).

We theoretically show that some of the recently proposed LRNNs lack the expressiveness to encode certain arithmetic operations used in the tasks of regular language. In light of this observation, we propose a new LRNN equipped with a block-diagonal and input-dependent transition matrix, which enable the successful modeling of regular language. Experiments show that the proposed model is the only LRNN architecture that can extrapolate well on regular language tasks such as Sum, Even Pair, and Modular Arithmetic.

LRNNs in this work have the following general

*Equal contribution

[1]Formally speaking, the rules are defined/recognized by the underlying finite-state machine.

Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers), pages 45–53

formulation:

$$\begin{aligned} x_k &= A_k x_{k-1} + B u_k \\ y_k &= h(x_k). \end{aligned} \tag{1}$$

A_k is a matrix that defines the recurrence relation. A_k may or may not depend on the input u_k. When it is input-independent, A_k is reduced to A; otherwise, $A_k = g(u_k)$ for some function g. The first line encodes a linear recurrence in the state x_k. The second line is an output y_k that depends on x_k. To control the expressiveness, the function h may or may not be a linear operation. Since the existing LRNNs differ in their linear recurrence relations (Eq. (2), (3), and (4)), we mainly focus on analyzing these relations.

2 Limitations of Most LRNNs

In this section, we theoretically show that most LRNNs are unable to represent arithmetic operations. The analysis serves as a motivation to study input-dependent transition matrices with constraints on their column norm.

2.1 Input-independent LRNN

To begin with, state-space models (in discrete-time format) follow the standard LRNN recurrence relation:

$$x_k = A x_{k-1} + B u_k \tag{2}$$

Eq. (2) encapsulates the recurrence relation of S4 (Gu et al., 2022; Gupta et al., 2022), S5 (Smith et al., 2023), and Linear Recurrent Unit (Orvieto et al., 2023). For example, A represents the HiPPO matrix family (Gu et al., 2023) of S4 or a complex diagonal matrix of Linear Recurrent Unit. We show in Proposition 1 that such an input-independent matrix A cannot represent subtraction.

Proposition 1. *An input-independent LRNN is inconsistent in representing subtraction.*

Proof. Denote u_0, u_-, and u_1 as the input vector w.r.t. input characters 0, -, and 1. Denote z as the initial state vector. The sequences "0-1" and "1-0" are represented as

$$\begin{aligned} x_{0-1} &= A^3 z + A^2 u_0 + A u_- + u_1, \quad \text{for "0-1"} \\ x_{1-0} &= A^3 z + A^2 u_1 + A u_- + u_0, \quad \text{for "1-0"} \end{aligned}$$

Because $0 - 1 \neq 1 - 0$, by forcing $x_{0-1} \neq x_{1-0}$, we have

$$A^2 u_0 + A u_- + u_1 \neq A^2 u_1 + A u_- + u_0.$$

On the other hand, let $x_{0-} = A^2 z + A u_0 + u_-$ be the vector representation for "0-". The sequences "0-0-1" and "0-1-0" are represented as

$$\begin{aligned} x_{0-0-1} &= A^3 x_{0-} + A^2 u_0 + A u_- + u_1 \\ x_{0-1-0} &= A^3 x_{0-} + A^2 u_1 + A u_- + u_0. \end{aligned}$$

Notice x_{0-0-1} is for "0-0-1" while x_{0-1-0} for "0-1-0". Enforcing $x_{0-0-1} = x_{0-1-0}$, we have

$$A^2 u_0 + A u_- + u_1 = A^2 u_1 + A u_- + u_0,$$

which is a contradiction. □

The limitation described by Proposition 1 also applies to models adopting diagonal linear recurrence relations (Gupta et al., 2022; Smith et al., 2023; Orvieto et al., 2023). The failure to represent regular language will be corroborated by the inferior length extrapolation performance reported later in § 4.

3 Proposed Method

Now that input-independent LRNNs struggle with representing arithmetic operations, we review the paradigms known to model regular language, which is the type of formal language recognized by a Finite State Automata (FSA) (Chomsky, 1956). An FSA is described by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. Q and Σ are non-empty sets of states and input symbols. $q_0 \in Q$ is an initial state. $\delta : Q \times \Sigma \to Q$ is an input-dependent transition function; $F \subseteq Q$ is a set of final states.

We hypothesize that an LRNN could model regular language if it can simulate an FSA, whose transition function has the following two key properties:

- It is input-dependent.
- If represented in the matrix form, its column vectors all have unit norm (in $\|\cdot\|_1$).

3.1 Diagonal Input-dependent LRNN

Let us first examine the simplest input-dependent LRNN:

$$x_k = \text{diag}(v_k) x_{k-1} + B u_k, \tag{3}$$

where $v_k = f(u_k)$ is a vector that depends on u_k. Unfortunately, we show that a diagonal input-dependent LRNN still cannot represent subtraction in Proposition 2.

Proposition 2. *A diagonal input-dependent LRNN is inconsistent in representing subtraction.*

The proof is essentially a generalization of Proposition 1 and is deferred to Appendix A.1.

3.2 Improved Expressiveness: Liquid-S4

To improve the expressiveness of Eq. (3), we note that the recently proposed liquid-S4 (Hasani et al., 2023) model has the following recurrence relation:

$$\begin{aligned} x_k &= Ax_{k-1} + (Bu_k) \odot x_{k-1} + Bu_k \\ &= (A + \text{diag}(Bu_k))x_{k-1} + Bu_k, \end{aligned} \tag{4}$$

where $\odot$ denotes the Hadamard product and $\text{diag}(w)$ constructs a diagonal matrix from w. Although Liquid-S4 does not suffer from the limitation outlined in Proposition 2, our experiments in § 4.4 show that Liquid-S4 still cannot extrapolate on regular language tasks.

3.3 Block-diagonal Input-dependent LRNN

Finally, we decide to push the expressiveness of A_k to the limit and make it fully input-dependent:

$$x_k = A_k x_{k-1} + Bu_k, \tag{5}$$

where $A_k = g(u_k)$ is a block diagonal matrix in practice for the sake of efficiency. A_k depends on u_k but not previous timesteps. g is an arbitrary function with the output being the size of A_k.

Eq. (5) is numerically unstable because the product $\prod_{i=1}^{k} A_i$ could produce large numbers. The solution is to impose additional constraints on the norm of A_k:

$$\begin{aligned} &A_k = \text{diag}\left(A_k^{(1)}, ..., A_k^{(h)}\right) \in \mathbb{R}^{bh \times bh} \\ &A_k^{(i)} = \left[v_k^{(i,1)} \quad \dots \quad v_k^{(i,b)}\right] \in \mathbb{R}^{b \times b} \\ &\|v_k^{(i,j)}\|_p \leq 1, \quad i \in [1, ..., h], \quad j \in [1, ..., b], \end{aligned} \tag{6}$$

where $\|\cdot\|_p$ denotes the vector p-norm and $v_k^{(i,j)}$ is a column vector that depends on u_k. For any vector v, we can derive another vector v' to satisfy the p-norm constraint through $v' = v / \max(1, \|v\|_p)$. Because $\|v\|_p \geq \|v\|_q$ when $p \leq q$, a smaller p imposes a stronger constraint on the columns of $A_k^{(i)}$. In other words, we can stabilize Eq. (5) by selecting a sufficiently small p.

Take $p = 1$ as an example. Every block $A_k^{(i)}$ is a matrix that none of its column norm is greater than 1 in $\|\cdot\|_1$. This implies $A_{k+1}^{(i)} A_k^{(i)}$ is the same kind of matrix. Specifically, let $v^{(1)}, ..., v^{(b)}$ be the columns of $A_{k+1}^{(i)} A_k^{(i)}$. We have

$$\begin{aligned} &\left[\|v^{(1)}\|_1 \quad \dots \quad \|v^{(b)}\|_1\right] = \mathbb{1}^\top \left|A_{k+1}^{(i)} A_k^{(i)}\right| \\ &\leq \mathbb{1}^\top \left|A_{k+1}^{(i)}\right| \left|A_k^{(i)}\right| \leq \mathbb{1}^\top \left|A_k^{(i)}\right| \leq \mathbb{1}^\top. \end{aligned} \tag{7}$$

Note that $\mathbb{1}$ is a column vector of all ones. $|\cdot|$ and $\leq$ are element-wise absolute value and inequality operations. The last two inequalities holds since the column norm of $A_{k+1}^{(i)}$ and $A_k^{(i)}$'s are no greater than 1 in $\|\cdot\|_1$.

Eq. (7) demonstrates that $p = 1$ can stabilize the proposed block-diagonal recurrence, Eq. (5). However, a small p restricts a model's expressiveness. In § 4.4, we will show that $p = 1.2$ is small enough to yield good empirical performance.

3.4 Efficient Implementation via Parallel Scan

We implement LRNNs in the parallel scan (`PScan`) mode as shown in Fig. 1. The idea of `PScan` is to group *similar* operations together, run them in parallel, and deliver the same results as those in the sequential (`Sequential`) for loop mode. For example, to compute $x_3 = A_3A_2A_1u_0 + A_3A_2u_1 + A_3u_2 + u_3$, `Sequential` runs this in three steps. On the other hand, `PScan` decomposes the computation into two steps:

- Step 1: Compute $A_1u_0 + u_1$ and $A_3u_2 + u_3$. Because these two operations are similar, we can compute them in parallel.
- Step 2: $x_3 = A_3A_2(A_1u_0 + u_1) + (A_3u_2 + u_3)$.

Generally speaking, a length-L generation takes $\lceil \log_2 L \rceil$ steps using `PScan`. However, each step requires careful handling of the intermediate matrices. As illustrated in Fig. 1, for a length-8 generation, the first step requires $[A_1, A_3, A_5, A_7]$, the second step requires $[A_2, A_3A_2, A_6, A_7A_6]$, and the third step requires $[A_4, A_5A_4, A_6A_5A_4, A_7A_6A_5A_4]$. To this end, we present an algorithm to generate the intermediate matrices in Appendix A.2.1. We integrate these intermediate matrices in `PScan` and show that `PScan` is equivalent to `Sequential` in Appendix A.2.2.

The computational complexity of our model is $O(b^3 h \log(T))$, where b, h, and T represent the block size, number of blocks, and sequence length, respectively. With the embedding dimension held fixed as bh, the complexity scales quadratically w.r.t the block size.

4 Experiments

4.1 Regular Language Tasks

We evaluate the models using the regular language transduction tasks introduced in Deletang et al.

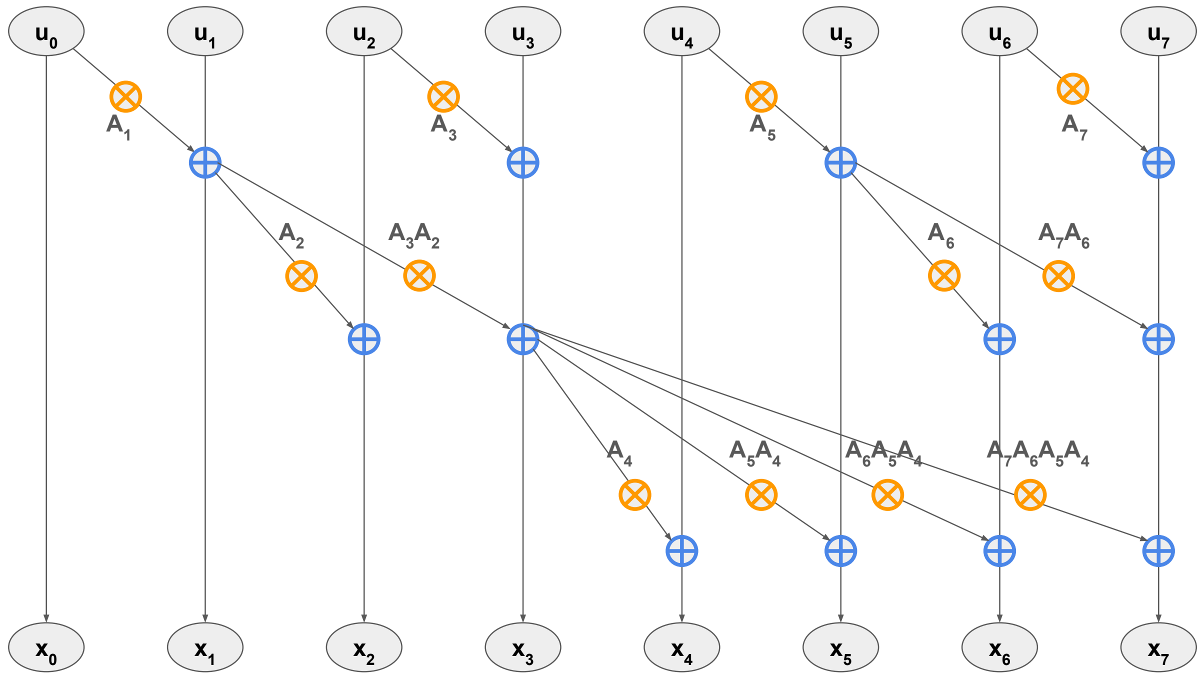

Figure 1: **Illustration of Parallel Scan for a length-8 generation.**

(2023). We prioritize language transduction over language recognition as the former can be more useful in practice Deletang et al. (2023). We are particularly interested in **Sum(5)**, **EvenPair(5)**, and **ModArith(5)**.

Sum(M) The input is a string $\{s_i\}_{i=0}^{n-1}$ of numbers in $[0, ..., M-1]$. The output is their sum modulo M: $\sum_{i=0}^{n-1} s_i \bmod M$. For example, when $M = 5$, the input $\texttt{0324}$ corresponds to the output $\texttt{4}$ because $0 + 3 + 2 + 4 \bmod 5 = 4$. Notably, **Sum(2)** is the famous PARITY problem that evaluates whether there is an odd number of $\texttt{1}$s in a bit string. Thus, **Sum(M)** is a generalization of PARITY and shares the same characteristic: If one error occurs during the summation, the output will be wrong.

EvenPair(M) The input is a string $\{s_i\}_{i=0}^{n-1}$ of numbers in $[0, ..., M-1]$. The output is 1 if $s_{n-1} = s_0$ and 0 otherwise. For example, when $M = 5$, the input $\texttt{0320}$ corresponds to the output $\texttt{1}$ because the first entry equals the last entry. Since **EvenPair(M)** only cares about the first and last entries, a model should learn to remember the first entry and forget the remaining ones $i \in [1, .., n-2]$.

ModArith(M) The input is a string $\{s_i\}_{i=0}^{n-1}$ of odd length (i.e., n is odd). The even entries ($i \in [0, 2, ...]$) are numbers in $[0, ..., M-1]$; The odd entries ($i \in [1, 3, ...]$) are symbols in $\{+, -, \times\}$. The output is the answer of a mathematical expression under modulo M. For example, when $M = 5$, the input $\texttt{1+2-3}\times\texttt{4}$ corresponds to the output $\texttt{1}$ because $1 + 2 - 3 \times 4 \bmod 5 = -9 \bmod 5 = 1$. **ModArith(M)** is much more complicated than **Sum(M)** and **EvenPair(M)** because a model should learn to prioritize multiplication over addition and subtraction.

4.2 Length Extrapolation

In our pilot experiments, we discovered that all models can achieve near-perfect same-length testing accuracy; i.e., testing with $L_{\text{ex}} = L_{\text{tr}}$. This is not impossible since a large enough model can memorize all training sequences in its parameters. To evaluate whether a model truly learns the underlying rules of a language, we first train a model on sequences of length L_{tr} generated by an FSA; It is then evaluated on sequences of length $L_{\text{ex}} > L_{\text{tr}}$ generated by the same FSA.

Table 1 summarizes the extrapolation setting. We mostly follow the requirements in Deletang et al. (2023), where the training and extrapolation lengths are 40 and 500. The lengths for **ModArith(5)** are 39 and 499 because this task requires odd-length inputs.

4.3 Baseline Models

We select baseline LRNNs such as S4 (Gu et al., 2022), S4D (Gupta et al., 2022), and Liquid-S4

	Sum(5)	EvenPair(5)	ModArith(5)
L_{tr}	40	40	39
L_{ex}	500	500	499

Table 1: **Training and Extrapolation Settings.** L_{tr} and L_{ex} represent the training and extrapolation sequence lengths, respectively.

	Ours	S4	S4D	Liquid-S4
Sum(5)	1.00	0.27	0.27	0.27
EvenPair(5)	0.99	0.81	0.82	0.72
ModArith(5)	1.00	0.27	0.27	0.27

Table 2: **Length Extrapolation Performance on Regular Language Tasks.** Each reported number is an average of five random trials. Each random trial returns the best testing accuracy over 40,000 gradient updates.

(Hasani et al., 2023) using the released codebase[2] under Apache-2.0 license. These models are chosen since they are the most stable and theoretically grounded LRNN design thanks to the careful parameterization of their state transition matrices. We also experiment with RWKV (Peng et al., 2023) and a vanilla LRNN without S4's parameterization. Unfortunately, their performance lags behind S4 on the reported tasks.

4.4 Experimental Results

For the proposed method, we set $p = 1.2$ in Eq. (6) and train the block-diagonal input-dependent LRNN with (b, h) = (8, 8). Because **ModArith** is more complicated than **Sum** and **EvenPair**, **ModArith** uses 3 layers while the others take 1 layer. Each layer is a full pass of LRNN as described in Eq. (1).

Table 2 compares the length extrapolation capability of our model with other LRNN baselines on regular language tasks. As we can see, the proposed model is the only LRNN that can extrapolate well on regular language. The inferior performance of S4 and S4D is expected since they cannot represent subtraction as illustrated in Prop. 1. As for Liquid-S4, despite the usage of input-dependent block matrices (discussed in § 3.2), it still cannot extrapolate well on regular language. We believe this can be explained by its low expressiveness (Eq. (4)) compared to the proposed model (Eq. (5) and (6)). Overall, we can see that the combination of input dependency and sufficient expressiveness plays an important role in terms of regular language modeling.

4.5 Speed Comparison

We conduct our experiments using a Quadro RTX 8000 GPU. To provide context for the aforementioned complexity analysis in § 3.4, we take the **Sum(5)** task and set $T = 40$ during the training stage. `Sequential` requires 0.033s per instance, while `PScan` completes the task in 0.021s. During the testing stage, we set $T = 500$, where both `Sequential` and `PScan` take 0.03s per instance. One might anticipate `PScan` to outperform `Sequential` during testing. However, in practice, this is not the case, as the complexity incurred by b^3 counteracts the speedup offered by $\log(T)$. To validate our hypothesis, we set $b = 1$ and reassess the speed. Subsequently, `PScan` achieves 0.0008s per instance, whereas `Sequential` takes 0.002s. Regarding why `PScan` demonstrates a notable speedup during the training stage, we hypothesize that it is due to the improved backpropagation path enabled by `PScan`.

[2]https://github.com/HazyResearch/state-spaces

5 Conclusion

In this work, we explored LRNNs in the realm of regular language modeling. We discovered that existing LRNNs cannot effectively represent subtraction. Consequently, we proposed a new LRNN equipped with a block-diagonal and input-dependent transition matrix. Our experiments confirmed the proposed model's capability to model various regular language tasks, including Sum, Even Pair, and Modular Arithmetic, under the challenging length extrapolation setting.

Limitations

The limitations of this work stem from several factors: (a) our evaluation is confined to only three regular language tasks; (b) the scope of our work excludes natural language; and (c) the proposed model introduces new hyperparameters such as the block size and the p-norm.

For (a), it is possible to discuss the average performance over randomly generated regular language, as demonstrated in Valvoda et al. (2022). Regarding (b), while natural language falls beyond the scope of our study, we believe the proposed model is at least as effective as prior linear RNN models on natural language, owing to its enhanced expressiveness. Concerning (c), the block size typically increases with the complexity of the problem. Nonetheless, it is feasible to maintain the same block size if more layers are employed (e.g., as described in § 4.4). Additionally, the p-norm parameter is chosen to be close to 1 to ensure stability; longer sequences correspond to smaller values of p.

Ethics Statement

Our work lays the groundwork for developing LRNNs in underexplored languages, such as regular language. Inappropriate usage of our technique might have negative societal impacts, including potential losses due to wrong predictions and ethical challenges regarding the improper use of the model. These implications apply to most language processing research and are not unique to this specific work.

References

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.

Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124.

Gregoire Deletang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A Ortega. 2023. Neural networks and the chomsky hierarchy. In *The Eleventh International Conference on Learning Representations*.

Jeffrey L Elman. 1990. Finding structure in time. *Cognitive science*, 14(2):179–211.

Daniel Y Fu, Tri Dao, Khaled Kamal Saab, Armin W Thomas, Atri Rudra, and Christopher Re. 2023. Hungry hungry hippos: Towards language modeling with state space models. In *The Eleventh International Conference on Learning Representations*.

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

Albert Gu, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Re. 2023. How to train your HIPPO: State space models with generalized orthogonal basis projections. In *International Conference on Learning Representations*.

Ankit Gupta, Albert Gu, and Jonathan Berant. 2022. Diagonal state spaces are as effective as structured state spaces. In *Advances in Neural Information Processing Systems*.

Ramin Hasani, Mathias Lechner, Tsun-Hsuan Wang, Makram Chahine, Alexander Amini, and Daniela Rus. 2023. Liquid structural state-space models. In *The Eleventh International Conference on Learning Representations*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Michael I Jordan. 1997. Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier.

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

Eric Martin and Chris Cundy. 2018. Parallelizing linear recurrent neural nets over sequence length. In *International Conference on Learning Representations*.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. 2023. Resurrecting recurrent neural networks for long sequences. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 26670–26698. PMLR.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, et al. 2023. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*.

Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. 2023. Hyena hierarchy: Towards larger convolutional language models. In *International Conference on Machine Learning*, pages 28043–28078. PMLR.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Jimmy T.H. Smith, Andrew Warrington, and Scott Linderman. 2023. Simplified state space layers for sequence modeling. In *The Eleventh International Conference on Learning Representations*.

Josef Valvoda, Naomi Saphra, Jonathan Rawski, Adina Williams, and Ryan Cotterell. 2022. Benchmarking compositionality with formal languages. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 6007–6018, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

A Additional Proofs

A.1 Proof of Proposition 2

Denote (A_0, u_0), (A_-, u_-), and (A_1, u_1) as the pairs of (transition matrix, input vector) w.r.t. input characters 0, $-$, and 1. Note that A_0, A_-, and A_1 are diagonal matrices by assumption.

Denote z as the initial state vector. The sequences `0-1` and `1-0` are represented as

$$x_{0-1} = A_1A_-A_0z + A_1A_-u_0 + A_1u_- + u_1$$
$$x_{1-0} = A_0A_-A_1z + A_0A_-u_1 + A_0u_- + u_0.$$

Note that x_{0-1} is `0-1` and x_{1-0} is `1-0`. Because the A matrices are diagonal, we know $A_1A_-A_0 = A_0A_-A_1$. Because $0 - 1 \neq 1 - 0$, by enforcing $x_{0-1} \neq x_{1-0}$, we have

$$A_1A_-u_0 + A_1u_- + u_1 \neq A_0A_-u_1 + A_0u_- + u_0. \tag{8}$$

On the other hand, let $x_{0-} = A_-A_0z + A_-u_0 + u_-$ be the vector representation for "0-". Consider two other sequences `0-0-1` and `0-1-0`, their vector representations are

$$x_{0-0-1} = A_1A_-A_0x_{0-} + A_1A_-u_0 + A_1u_- + u_1$$
$$x_{0-1-0} = A_0A_-A_1x_{0-} + A_0A_-u_1 + A_0u_- + u_0.$$

Note x_{0-0-1} is `0-0-1` and x_{0-1-0} is `0-1-0`. Similarly, because the A matrices are diagonal and $0 - 0 - 1 = 0 - 1 - 0$, by enforcing $x_{0-0-1} = x_{0-1-0}$, we have

$$A_1A_-u_0 + A_1u_- + u_1 = A_0A_-u_1 + A_0u_- + u_0. \tag{9}$$

Because Eq. (8) contradicts Eq. (9), the two relations $x_{0-1} \neq x_{1-0}$ and $x_{0-0-1} = x_{0-1-0}$ cannot co-exist. We hence conclude that an input-dependent diagonal linear RNN is inconsistent in representing subtraction.

A.2 Code for `PScan`

A.2.1 Illustration of Matrix Generation

```
import numpy as np
seq_len = 2**3 - 1
arr = np.array(['A' + str(i) for i in range(1,seq_len +1)]).reshape(-1,1)

def spt(x):
    assert len(x)%2 == 1, 'works when len(x)== 2**k -1 for k>=1'
    coef = x[::2]
    remain = x[1::2]

    coef_remain = np.core.defchararray.add(coef[1:], remain[:,-1:])
    remain = np.concatenate([remain, coef_remain], axis=1)
    return coef, remain

for i in range( int(np.ceil(np.log2(seq_len))) ):
    coef, arr = spt(arr)
    print(coef)
```

The below output shows the function `spt()` can generate the intermediate matrices during `PScan`.

```
[['A1']
 ['A3']
 ['A5']
 ['A7']]
[['A2' 'A3A2']
 ['A6' 'A7A6']]
[['A4' 'A5A4' 'A6A5A4' 'A7A6A5A4']]
```

A.2.2 Testing the Equivalence of `Sequential` and `PScan`

```
import numpy as np
import torch
import torch.nn as nn
torch.manual_seed(1)
emb_dim = 2
seq_len = 7
bs = 1
A = torch.randn(bs, seq_len, emb_dim, emb_dim)
u = torch.randn(bs, seq_len, emb_dim)
x0 = torch.randn(1, emb_dim)

# sequential
x = x0.expand(bs, emb_dim)
all_x = [x[:,None,:]]
for i in range(seq_len):
    x = torch.einsum('bij,bj->bi', A[:,i], x) + u[:,i]
    all_x.append(x[:,None,:])
all_x = torch.cat(all_x, dim=1)
print('sequential mode')
print(all_x)


# parallel scan
def scan(x, As):
    c = As.shape[2]*2
    x = x.view(bs, L//c, c, -1)
    x1, x2 = x[:,:,:c//2], x[:,:,c//2:]

    # x2.shape = (bs, group nums, group size, emb_dim)
    # As.shape = (bs, group nums*2-1, group size, emb_dim, emb_dim)

    assert As.shape[1]%2==1, 'works when As.shape[1]== 2**k -1 for k>=1'
    coef = As[:,::2]
    remain = As[:,1::2]
    prodd = torch.einsum('bncij,bnjk->bncik', coef[:,1:], remain[:,:,-1])
    remain = torch.cat([remain, prodd], dim=2)

    # coef.shape = (bs, group nums, group size, emb_dim, emb_dim)
    # apply a group of matrix (e.g., ['A2' 'A3A2']) to the last element of x2 in each group,
    # and add together
```

```
    x2 = x2 + torch.einsum('bncij,bnj->bnci', coef, x1[:,:,-1])
    x = torch.cat([x1, x2], dim=2)

    return x, remain

log2_L = int(np.ceil(np.log2(seq_len+1)))
L = 2**log2_L # the length after zero padding
n_zero = L - seq_len - 1
eu = torch.cat([x0.expand(bs,-1)[:,None,:], u], dim=1)
eu = nn.functional.pad(eu, (0,0,0, n_zero))


x = eu
As = nn.functional.pad(A, (0,0,0,0,0, n_zero))[:,:,None,:,:]

for i in range(log2_L):
    x, As = scan(x, As)
x = x.view(bs, L, emb_dim)[:,:seq_len+1,:]
print('parallel mode')
print(x)
```

The below shows that `Sequential` and `PScan` are equivalent as they generate the same outputs.

```
sequential mode
tensor([[[ 0.8310, -0.2477],
         [ 0.5167, -1.4218],
         [ 1.1399,  1.3024],
         [ 0.9628,  1.3150],
         [-1.5308, -1.6903],
         [-3.6631,  1.6082],
         [ 1.7805,  7.1659],
         [ 2.5068, -0.6256]]])
parallel mode
tensor([[[ 0.8310, -0.2477],
         [ 0.5167, -1.4218],
         [ 1.1399,  1.3024],
         [ 0.9628,  1.3150],
         [-1.5308, -1.6903],
         [-3.6631,  1.6082],
         [ 1.7805,  7.1659],
         [ 2.5068, -0.6256]]])
```