lrnnx: A library for Linear RNNs

Karan Bania*[1], Soham Kalburgi*[2], Manit Tanwar*[2], Dhruthi Kiran*[2], Aditya Nagarsekar*[2], Harshvardhan Mestha*[2], Naman Chibber*[2], Anish Sathyanarayanan*[2], Aarush Rathore*[2], Raj Deshmukh*[2], Pratham Chheda*[2]

[1]Carnegie Mellon University, [2]BITS Pilani, K. K. Birla Goa Campus

https://github.com/SforAiDl/lrnnx

Abstract

Linear recurrent neural networks (LRNNs) provide a structured approach to sequence modeling that bridges classical linear dynamical systems and modern deep learning, offering both expressive power and theoretical guarantees on stability and trainability. In recent years, multiple LRNN-based architectures have been proposed, each introducing distinct parameterizations, discretization schemes, and implementation constraints. However, existing implementations are fragmented across different software frameworks, often rely on framework-specific optimizations, and in some cases require custom CUDA kernels or lack publicly available code altogether. As a result, using, comparing, or extending LRNNs requires substantial implementation effort. To address this, we introduce lrnnx, a unified software library that implements several modern LRNN architectures under a common interface. The library exposes multiple levels of control, allowing users to work directly with core components or higher-level model abstractions. lrnnx aims to improve accessibility, reproducibility, and extensibility of LRNN research and applications. We make our code available under a permissive MIT license.

1 Introduction

1.1 Context and Motivation

Recurrent neural networks (RNNs) are a classical approach to sequence modeling, which model context explicitly with a latent state. A conventional (non-linear) RNN can be described by eq. (1):

$$\begin{aligned} x_k &= \alpha(W_{xx}x_{k-1} + W_{xu}u_k), \\ y_k &= \beta(W_{yx}x_k), \end{aligned} \tag{1}$$

where α and β are non-linear activation functions. These non-linearities are largely responsible for the expressive power of RNNs, including results on Turing completeness (Siegelmann and Sontag, 1995). However, non-linear RNNs suffer from two well-known limitations: (i) the vanishing and exploding gradient problem (Hochreiter and Schmidhuber, 1997), which hinders both training stability and the learning of long-range dependencies, and (ii) the inherently sequential nature of training, which limits effective utilization of modern parallel hardware.

Despite these drawbacks, RNNs possess a highly desirable property: $\mathcal{O}(1)$ time complexity for inference. Transformers (Vaswani et al., 2017), which have become the dominant paradigm for sequence modeling, address both gradient instability and sequential training. However, they do so by abandoning the notion of an explicit latent state, resulting in $\mathcal{O}(n)$ time complexity for inference due to global attention, where n denotes the sequence length.

Linear recurrent neural networks (LRNNs) revisit the recurrent paradigm by restricting the state update to linear dynamics while carefully controlling stability through parameterization and discretization. This line of work has produced a family of models that combine efficient parallel training with $\mathcal{O}(1)$ inference-time complexity, while setting new records on long-range sequence modeling benchmarks. Moreover, LRNNs possess an inductive bias for signal data, enabling efficient end-to-end modeling of high-frequency modalities such as audio and sensor data streams.

1.2 Implementation Challenges

While the theoretical foundations and empirical performance of LRNNs have matured over time, their practical use remains hindered by the current fragmented implementation landscape. As illustrated in Table 1, existing LRNN architectures differ not only in modeling assumptions but also in software

*Equal contribution.

Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics
Volume 4: Student Research Workshop, pages 811–817

Layer	SISO	LTI	Public Implementation	Framework
S4 (Gu et al., 2022)	✓	✓	✓	PyTorch
S5 (Smith et al., 2023)	✗	✓	✓	JAX
LRU (Orvieto et al., 2023)	✗	✓	✗	N/A
Event-SSM (Schöne et al., 2024b)	✗	✓	✓	JAX
S6 (Gu and Dao, 2024)	✓	✗	✓	PyTorch
STREAM (Schöne et al., 2024a)	✓	✗	✓	PyTorch
RG-LRU (De et al., 2024)	✗	✗	✗	N/A
S7 (Soydan et al., 2024)	✗	✗	✗	N/A
Centaurus (Pei, 2025)	✗	✗	✓	PyTorch

Table 1: An overview of contemporary SSM architectures and their existing implementations (SISO: Single-Input Single-Output, LTI: Linear Time Invariant).

availability and framework choice. For example, comparing two conceptually similar models may require switching between PyTorch and JAX, adapting data pipelines, and re-implementing training utilities, while reproducing reported runtimes may further depend on custom CUDA kernels or unpublished low-level optimizations. In several cases, no public implementation is available at all, forcing researchers to re-implement entire models from scratch. This makes it difficult to reproduce results, benchmark models under consistent conditions, or integrate LRNNs into downstream applications. As a consequence, using LRNNs in practice or experimenting with them beyond a single architecture requires substantial engineering overhead.

1.3 The `lrnnx` Library

We address these challenges by introducing `lrnnx`, a unified library designed to make working with LRNNs comparable to working with standard neural network layers. The library provides consistent implementations of multiple LRNN architectures within a single unified framework, and abstracts away model-specific engineering details. As a result, switching between different LRNN formulations - such as changing the state-space parameterization or discretization scheme - amounts to instantiating a different class of the library, without needing to modify the surrounding training or evaluation code. `lrnnx` exposes both low-level building blocks (core recurrences) and higher-level modules (with activations and skip connections), supporting fine-grained research and experimentation as well as drop-in use in existing pipelines for direct application.

Our contributions in this work include the development of `lrnnx`, a unified framework that standardizes fragmented LRNN architectures into a single interface supported by high-performance custom CUDA kernels, thereby bridging the gap between research and deployment while significantly reducing the engineering overhead for cross-model benchmarking.

2 Related Work

Since the introduction of GPT-3 (Brown et al., 2020), a large body of research has focused on optimizing Transformer architectures and expanding their applications to diverse domains.

2.1 Speeding up Transformers

Efforts to mitigate the quadratic complexity of the Transformer's self-attention mechanism have yielded several approaches. LongFormer (Beltagy et al., 2020) replaces full attention with a combination of sliding window and global attention patterns. A broader class of *sub-quadratic methods* uses techniques like low-rank projections (Wang et al., 2020) or locality-sensitive hashing (Kitaev et al., 2020) to approximate attention more efficiently. A few hardware-aware techniques have also emerged. FlashAttention (Dao et al., 2022) reduces memory I/O without any approximations and vLLM (Kwon et al., 2023) introduces paged attention for efficient memory management. Recently, there has also been some work on pseudo distillation techniques like Matryoshka Embeddings (Kusupati et al., 2022) and Speculative Decoding (Leviathan et al., 2023). Most of these methods are transferrable to LRNNs.

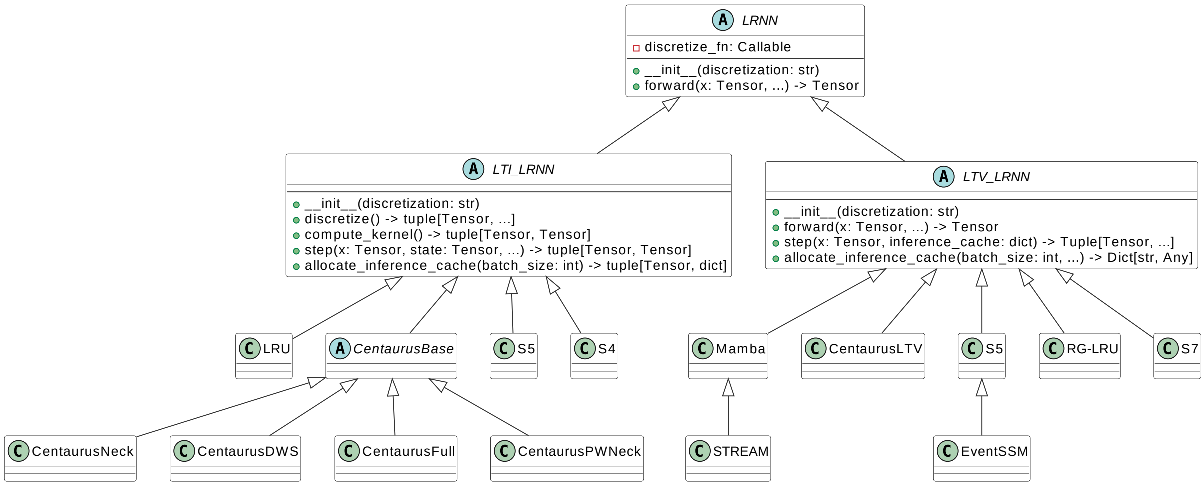

Figure 1: Class diagram describing `lrnnx`.

2.2 Linear RNNs

The central equation for LRNNs is described in eq. (2).

$$
\begin{aligned}
x_k &= A(k)x_{k-1} + B(k)u_k \\
y_k &= C(k)x_k \quad\ + D(k)u_k
\end{aligned}
\tag{2}
$$

Layer variants differ in how they parameterize the learnable matrices A, B and C. These layers can be broadly divided into two types: Linear Time Invariant (LTI) and Linear Time Varying (LTV).

2.2.1 LTI Layers

These layers maintain time-invariant matrices, i.e., $A(k) = A, \forall k$ (likewise for B and C). S4 (Gu et al., 2022) developed much of the theory required to train and compute this recurrence efficiently. These layers rely on the single-input single-output (SISO) framework, and use independent layers for each hidden dimension in the input. The S5 (Smith et al., 2023) layer extends S4 to train a multi-input multi-output (MIMO) model. LRU (Orvieto et al., 2023) re-formulates the problem from a deep learning perspective and develops methods to train a LRNN without the signal processing theory. The network is similar to S5 but makes no assumptions about the input signal u_k.

2.2.2 LTV Layers

These layers have time-varying matrices, and most have a direct LTI counterpart. S6 (Gu and Dao, 2024) is a time varying variant of S4 which makes it well suited for discrete modalities like text. S7 (Soydan et al., 2024) is a time-varying variant of S5, and the RG-LRU (De et al., 2024) is a time-varying alternative to LRU. STREAM (Schöne et al., 2024a) introduces a time-varying SISO state-space model that selectively updates state components to capture varying temporal frequencies in long sequences.

Finally, Centaurus (Pei, 2025) is in-between SISO and MIMO models.

2.3 Applications

Overall, these layers are a rich set of architectures which have been applied to several sequential and non-sequential domains from Audio (Text-to-speech (Goel et al., 2022), ASR (Pei, 2025), Enhancement (Pei, 2025; Pei et al., 2025)), RNA modeling (Ramesh et al., 2025), Vision (Liu et al., 2024), Event-streams (Schöne et al., 2024b) and even Point-clouds (Han et al., 2024). Furthermore, they have set new benchmarks on synthetic tasks in the long-range-arena (LRA) (Tay et al., 2021). Typically, transformers are hard to train for very long sequences ($\geq 2^{10}$), which is where these layers prove extremely useful.

3 Library Design

This section provides a high-level overview of `lrnnx`, describing its software architecture and core design principles.

Each layer in `lrnnx` follows a consistent interface derived from eq. (2). Model-specific details are abstracted behind a unified API for instantiation, training, and inference across all LRNN architectures. A summary of supported layer architectures is provided in Table 1.

We adopt a three-tier inheritance hierarchy. At the base, the `LRNN` class defines the forward interface and selects the discretization method. Layers are organized into LTI and LTV submodules corresponding to the variants described in section 2.2. LTI layers extend the `LTI_LRNN` class. For these layers, we implement optimal einsum contractions (Pei et al., 2025), which lead to efficiency gains. LTV layers extend the `LTV_LRNN` class. Each subclass defines its own parameterization of the matrices (A, B, C) from eq. (2), while preserving a shared programming interface. The broad layout of the library is as indicated in Figure 1.

Layer definition is decoupled from discretization. Supported schemes include ZOH, bilinear, dirac, and asynchronous (event-driven) discretization. Some models restrict supported methods (e.g., Centaurus uses only ZOH), and the design allows easy integration of custom schemes.

Layers follow a uniform constructor signature. For example, an S5 layer can be instantiated as:

```
layer = S5(
        d_model=512,
        d_state=64,
        discretization="zoh",
        **kwargs
)
```

For efficient autoregressive generation, all layers implement a `step` method.

For time-varying layers, `lrnnx` provides custom CUDA kernels, derived from the selective scan implementation in Mamba (Gu and Dao, 2024). These kernels integrate multiple discretization methods (ZOH, bilinear, dirac) and support asynchronous inputs within a fused scan and output projection, preserving memory efficiency while enabling flexible architectural choices. This is a benefit over some JAX implementations, which, while easy to implement, suffer from memory bottlenecks due to materialization of the hidden state.

To ensure correctness, we validate numerical equivalence between parallel, recurrent, and stepwise execution modes for every layer with an extensive and robust test suite, across sequence lengths, batch sizes, model dimensions, initializations, and discretizations. We further verify gradient consistency between custom CUDA kernels and reference PyTorch implementations.

3.1 Tutorials & Architectures

For end-to-end applications, the library provides components and tutorials for tasks such as language modeling, classification, and autoencoders. For example, `LRNNLMHeadModel` wraps an LRNN backbone with embeddings, stacked residual blocks, and a language modeling head:

```
lm = LRNNLMHeadModel(
    d_model=768, d_state=16, n_layer=12,
    vocab_size=50257,
    mixer_types=["S5", "S7", "attn", ...],
    mixer_kwargs={"S5": {..}, "S7": {..}, "attn": {..}, ...},
    d_intermediate=2048,
)
```

This design mirrors the head abstractions used in modern deep learning frameworks like Transformers (Wolf et al., 2020), enabling flexible adaptation to downstream tasks. The `mixer_types` argument allows mixing different LRNN backends and attention layers (De et al., 2024), while blocks, normalization, and MLP components remain fully configurable. All layers integrate with standard PyTorch workflows, including checkpointing, gradient checkpointing, mixed-precision training, and fused operations.

3.2 Inference support

JAX provides native support for such models with the `jax.lax.scan` operation which can remove CPU overheads entirely from the generation process. Analogues of this functionality do not exist in PyTorch, and a simple for-loop would give up all the benefits of fast inference. To mitigate this, similar to Gu and Dao (2024), we provide specialized inference capabilities using CUDA Graphs, to avoid CPU synchronization after each step. Our implementation is competitive at large sequence lengths and only adds a few ms at small ones.

4 Experiments

4.1 Setup

We run all of our GPU benchmarks on an NVIDIA A100 40GB GPU, using Python 3.12 and CUDA 12.9.

4.2 Benchmarks

We provide a performance analysis of our `lrnnx` implementations by comparing them to their original or alternative counterparts, on random tensors. We have evaluated our LRU implementation (PyTorch) against a popular public repository (Zucchet, 2023) (JAX). Our S5 implementation was compared against the original release (Smith et al., 2023), and similarly the Mamba implementation is evaluated relative to the official repository (Gu and Dao, 2024).

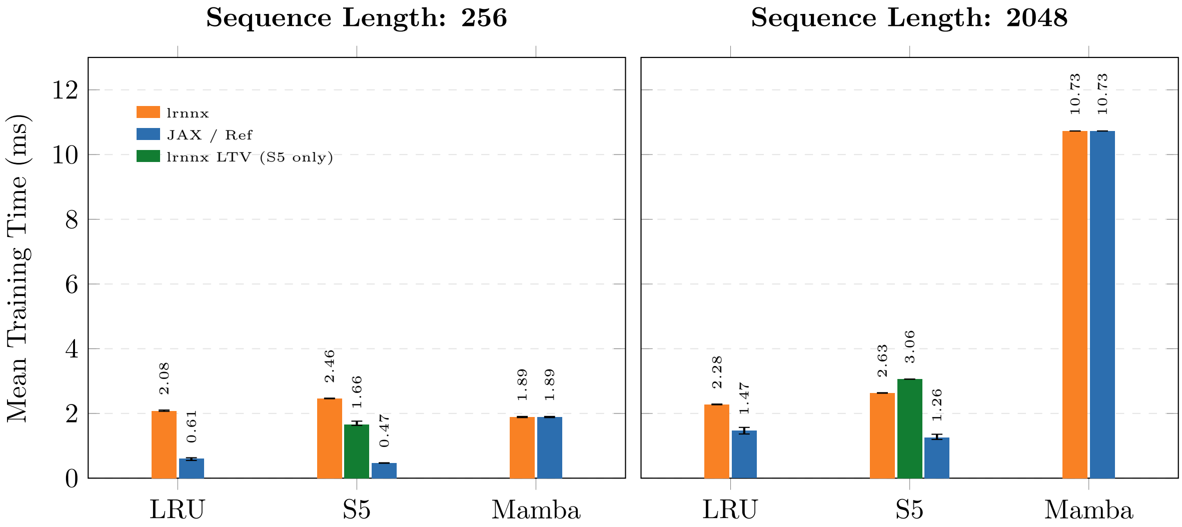

Figure 2: Training Time Comparison

We report average execution time (ms) for both training (forward plus backward pass) and autoregressive inference across models while varying batch size, sequence length, and model dimension for LRU, S5, and Mamba. For each configuration we run 10 warm-up passes, then time 90 forward passes; this is repeated for 5 experiments, and we report the mean and standard deviation across those 5 experiment means. We mirror the same sweep settings across all three models (batch sizes, sequence lengths, and model dimensions), and all plots use log scaling where specified. Wherever required, we set the state dimension to 16. Overall, our implementations are competitive to the public baselines – Figure 2. All benchmark results can be found in Appendix A.

5 Conclusion

In this work, we introduce `lrnnx`, a unified library consolidating SOTA linear RNN architectures into a single interface. By providing $O(1)$ inference complexity and strong inductive biases for signal-like data, the library facilitates efficient long-sequence modeling across diverse domains, including audio, vision, and event-streams (section 2.3). We expect `lrnnx` to empower the community with a scalable, easily extensible, and accessible alternative where Transformer-based methods encounter limitations.

Limitations

Despite its unified interface, `lrnnx` faces some constraints. Mirroring industry shifts toward single-framework specialization (Debut, 2025), our implementation is restricted to PyTorch, precluding direct use by researchers in the JAX or TensorFlow communities. Furthermore, the high-performance execution of several LTV layers relies on custom CUDA kernels, limiting optimal performance to NVIDIA hardware, and hindering accessibility for alternative backends.

We note that our models match other public implementations on training speed but are slightly slower for inference. We attribute this to known CPU overheads in PyTorch inference execution rather than to model-specific design choices. Though for production workloads, particularly in high batch size and long sequence length regimes, we expect inference performance to be very similar. Finally, the library lacks native wrappers for established ecosystem tools like Hugging Face (Wolf et al., 2020), DeepSpeed (Rasley et al., 2020), and FSDP (Zhao et al., 2023). Consequently, incorporating these models into large-scale distributed workflows requires the manual development of custom adapter layers. Beyond ecosystem integrations, there are architectural features we have not yet implemented. We do not yet provide bidirectional variants of LRNN layers, though the base interface is designed to support them.

Recently there has also been a resurgence in non-linear RNNs like xLSTM (Beck et al., 2024), while related in capabilities, these methods are orthogonal to our focus and thus have not been implemented.

References

Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. 2024. xlstm: Extended long short-term memory. In *Thirty-eighth Conference on Neural Information Processing Systems*.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, and 1 others. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems*.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *arXiv preprint arXiv:2205.14135*.

Soham De, Samuel L. Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, Guillaume Desjardins, Arnaud Doucet, David Budden, Yee Whye Teh, Razvan Pascanu, Nando De Freitas, and Caglar Gulcehre. 2024. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *Preprint*, arXiv:2402.19427.

Lysandre Debut. 2025. Linkedin post.

Karan Goel, Albert Gu, Chris Donahue, and Christopher Re. 2022. It's raw! Audio generation with state-space models. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 7616–7633. PMLR.

Albert Gu and Tri Dao. 2024. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*.

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

Xu Han, Yuan Tang, Zhaoxuan Wang, and Xianzhi Li. 2024. Mamba3d: Enhancing local features for 3d point cloud analysis via state space model. In *ACM Multimedia 2024*.

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

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*.

Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, and Ali Farhadi. 2022. Matryoshka representation learning. *arXiv preprint arXiv:2205.13147*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. *arXiv preprint arXiv:2211.17192*.

Yue Liu, Yunjie Tian, Yuzhong Zhao, Hongtian Yu, Lingxi Xie, Yaowei Wang, Qixiang Ye, Jianbin Jiao, and Yunfan Liu. 2024. VMamba: Visual state space model. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. 2023. Resurrecting recurrent neural networks for long sequences. *Preprint*, arXiv:2303.06349.

Yan Ru Pei. 2025. Let SSMs be convnets: State-space modeling with optimal tensor contractions. In *The Thirteenth International Conference on Learning Representations*.

Yan Ru Pei, Ritik Shrivastava, and Fnu Sidharth. 2025. Optimized Real-time Speech Enhancement with Deep SSMs on Raw Audio. In *Interspeech 2025*, pages 51–55.

Krithik Ramesh, Sameed M. Siddiqui, Albert Gu, Michael D. Mitzenmacher, and Pardis C. Sabeti. 2025. Lyra: An efficient and expressive subquadratic architecture for modeling biological sequences. *Preprint*, arXiv:2503.16351.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA. Association for Computing Machinery.

Mark Schöne, Karan Bania, Yash Bhisikar, Khaleelulla Khan Nazeer, Christian Mayr, Anand Subramoney, and David Kappel. 2024a. Stream: A universal state-space model for sparse geometric data. *Preprint*, arXiv:2411.12603.

Mark Schöne, Neeraj Mohan Sushma, Jingyue Zhuge, Christian Mayr, Anand Subramoney, and David Kappel. 2024b. Scalable event-by-event processing of neuromorphic sensory signals with deep state-space models. *Preprint*, arXiv:2404.18508.

Hava T. Siegelmann and Eduardo D. Sontag. 1995. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150.

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*.

Taylan Soydan, Nikola Zubić, Nico Messikommer, Siddhartha Mishra, and Davide Scaramuzza. 2024. S7: Selective and simplified state space layers for sequence modeling. *Preprint*, arXiv:2410.03464.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2021. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, and 3 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860.

Nicolas Zucchet. 2023. minimal-lru: Jax implementation of the linear recurrent unit. `https://github.com/NicolasZucchet/minimal-LRU`. GitHub repository.