Were RNNs All We Needed?

Leo Feng
Mila – Université de Montréal & Borealis AI
leo.feng@mila.quebec

Frederick Tung
Borealis AI
frederick.tung@borealisai.com

Mohamed Osama Ahmed
Borealis AI
mohamed.o.ahmed@borealisai.com

Yoshua Bengio
Mila – Université de Montréal
yoshua.bengio@mila.quebec

Hossein Hajimirsadeghi
Borealis AI
hossein.hajimirsadeghi@borealisai.com

Abstract

The scalability limitations of Transformers regarding sequence length have renewed interest in recurrent sequence models that are parallelizable during training. As a result, many novel recurrent architectures, such as S4, Mamba, and Aaren, have been proposed that achieve comparable performance. In this work, we revisit traditional recurrent neural networks (RNNs) from over a decade ago: LSTMs (1997) and GRUs (2014). While these models were slow due to requiring to backpropagate through time (BPTT), we show that by removing their hidden state dependencies from their input, forget, and update gates, LSTMs and GRUs no longer need to BPTT and can be efficiently trained in parallel. Building on this, we introduce minimal versions (minLSTMs and minGRUs) that (1) use significantly fewer parameters than their traditional counterparts and (2) are fully parallelizable during training (175× faster for a sequence of length 512). Lastly, we show that these stripped-down versions of decade-old RNNs match the empirical performance of recent sequence models.

1 Introduction

Over the past few years, Transformers (Vaswani et al., 2017) have been the dominant architecture in many areas, leading to advancements in tasks like machine translation (Devlin et al., 2019), text generation (Brown et al., 2020), and more. However, Transformers have a quadratic computational complexity in the sequence length, making them prohibitively expensive for long sequences, especially in low-resource settings. As such, numerous works have investigated the design of more efficient alternatives that achieve competitive performance with that of Transformers. Recently, there has been a renewed interest in recurrent sequence models that can be trained efficiently processing their context in parallel. These models (1) during training require only linear memory in the sequence length and (2) at inference time are rolled out recurrently token-by-token, requiring only constant memory. As a result, these models can scale to significantly longer sequences than Transformers[1].

A family of efficiently trainable recurrent sequence models that has recently gained much traction is that of state-space models, specifically the recently proposed Mamba (Gu & Dao, 2024). Mamba

[1] The title of this paper pays tribute to the original Transformers paper, “Attention is All You Need”.

Preprint. Under review.

(S6) is a state-space model that differentiates itself from prior works by leveraging input-dependent transitions. The recent success of Mamba and the proposals of many new variants of state-space models has led to several survey papers (Wang et al., 2024; Patro & Agneeswaran, 2024; Qu et al., 2024). Another extensively explored group of methods is those based on attention. Peng et al. (2023) proposed a linear attention model that can be written recurrently while being trained in parallel. Feng et al. (2024) showed that softmax attention (and Transformers) can be viewed as a recurrent neural network (RNN). Building on their RNN formulation of attention, they proposed Aaren, a softmax attention model, that can be computed in parallel for efficient training or unrolled sequentially as an RNN for efficient inference. Although many recurrent models have been proposed with vastly different architectures, these recent state-of-the-art methods are all efficiently trainable using the same algorithm – the parallel prefix scan algorithm (Blelloch, 1990).

Inspired by the striking algorithmic similarities between the numerous recently proposed sequence models, we revisit LSTMs (Hochreiter & Schmidhuber, 1997) and GRUs (Cho et al., 2014) from a modern lens. As traditional RNNs from over a decade ago, LSTMs and GRUs are only computable sequentially and require to backpropagate through time (BPTT) during training. As such, LSTMs and GRUs were far too slow to scale beyond a few hundred tokens, resulting in their deprecation. Revisiting these models, we show that by removing hidden state dependencies from their input, forget, and update gates, LSTMs and GRUs no longer need to BPTT and can be trained efficiently using the parallel scan algorithm. Building on this, we simplify LSTMs and GRUs further by removing their constraints on output range, (i.e., their use of $\tanh$) and ensuring their output is time-independent in scale. These steps result in minimal versions (minLSTMs and minGRUs) that (1) use significantly fewer parameters than their traditional counterpart and (2) are trainable in parallel ($175\times$ faster for a context length of 512). Finally, we show that these stripped-down versions of decade-old RNNs match the empirical performance of recent sequence models.

2 Background

In this section, we review recurrent neural networks (RNNs). RNNs are recurrent sequence models that maintain a hidden state across time steps, capturing temporal dependencies. As such, RNNs are particularly suitable for sequence modelling settings such as those involving time series, natural language processing, and other sequential tasks where context from previous steps informs the current prediction. Vanilla RNNs (Elman, 1990), however, struggle with issues of vanishing and exploding gradients, limiting their ability to learn long-term dependencies.

2.1 LSTM

Addressing this limitation, Hochreiter & Schmidhuber (1997) introduced Long Short-Term Memory (LSTM) networks. LSTMs are enhanced RNNs designed to mitigate the vanishing gradient problem, allowing the model to learn long-term dependencies. LSTMs are computed as follows:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\boldsymbol{i}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\tilde{\boldsymbol{c}}_t &= \tanh(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\boldsymbol{o}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)
\end{aligned}$$

where $\odot$ represents an element-wise multiplication of vectors, t is the current timestep, $\boldsymbol{h}_t$ is the outputted hidden state, $[\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]$ represents the concatenation of $\boldsymbol{x}_t$ with $\boldsymbol{h}_{t-1}$, d_h is the size of the hidden state, $\boldsymbol{c}_t$ is a cell state that maintains information over the sequence, and $\tilde{\boldsymbol{c}}_t$ is the candidate cell state to be added, $\boldsymbol{i}_t$, $\boldsymbol{f}_t$, and $\boldsymbol{o}_t$ are gating mechanisms. The input gate $\boldsymbol{i}_t$ controls how much new information from the candidate cell state is added. The forget gate $\boldsymbol{f}_t$ determines the proportion of information in the cell gate to discard. The output gate $\boldsymbol{o}_t$ decides what information from the cell state should be outputted. The σ and $\tanh$ are used for scaling to ensure that the output does not explode/vanish. An LSTM module maintains both a cell and a hidden state and, in total, contains $O(4d_h(d_x + d_h))$ parameters.

2.2 GRU

Simplifying LSTM, Cho et al. (2014) introduced Gated Recurrent Unit (GRU) which only uses two gates and a single state instead of LSTM's three gates and two states (hidden and cell state). GRU's reduced complexity leads to faster training and inference times while achieving competitive performance in many tasks. GRUs are computed as follows:

$$\begin{aligned}
\boldsymbol{z}_t &= \sigma(\text{Linear}_{d}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\boldsymbol{r}_t &= \sigma(\text{Linear}_{d}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\tilde{\boldsymbol{h}}_t &= \tanh(\text{Linear}_{d}([\boldsymbol{x}_t, \boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}])) \\
\boldsymbol{h}_t &= (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t
\end{aligned}$$

where $\tilde{\boldsymbol{h}}_t$ is the candidate hidden state that represents a potential new value for the hidden state. GRU combines LSTM's forget and input gates into a single update gate $\boldsymbol{z}_t \in (0, 1)$ which decides how much of the past information to carry forward (i.e., $1 - \boldsymbol{z}_t$) and how much new information from the candidate hidden state to add (i.e., $\boldsymbol{z}_t$). Additionally, LSTM's output gate is removed and instead, a reset gate $\boldsymbol{r}_t$ is added that controls how much past information is used in computing the candidate hidden state. GRU reduces the total number of parameters and computations, requiring only $O(3d_h(d_x + d_h))$ parameters. However, GRUs and LSTMs are only computable sequentially. As a result, during training they require backpropagating their gradients through time (BPTT), requiring linear training time and greatly limiting their ability to scale to long contexts.

2.3 Parallel Scan

Due to this limitation, Transformers replaced LSTMs and GRUs as the defacto sequence modelling method for years by leveraging parallelization during training. However, Transformers have a quadratic complexity in the sequence length, limiting their ability to scale to long contexts. Recently, a resurgence of many new recurrent models have been proposed as replacements for Transformers that achieve comparable performance and are trainable in parallel, while avoiding the BPTT issue that traditional RNNs (e.g., LSTMs and GRUs) faced. Although many different architectures have been proposed, many of these models are efficiently trained using the parallel prefix scan algorithm (Blelloch, 1990).

The parallel scan algorithm is a parallel computation method for computing N prefix computations from N sequential data points via an associative operator $\oplus$ (e.g., $+$ and $\times$). The algorithm efficiently computes $\{\bigoplus_{i=1}^{k} u_i\}_{k=1}^{N}$ from $\{u_k\}_{k=1}^{N}$. In particular, we can apply the parallel scan method for efficiently computing a popular family of functions: $v_t = a_t v_{t-1} + b_t$ where $v_t, a_t, b_t \in \mathbb{R}$ and $v_0 \leftarrow b_0$ (Heinsen, 2023). The method takes as input $a_1, \ldots, a_n$ and $b_0, b_1, \ldots, b_n$ and computes via parallel scans $v_1, \ldots, v_n$.

3 Methodology

Naturally, the aforementioned algorithm also extends to vectors: $\boldsymbol{v}_t = \boldsymbol{a}_t \odot \boldsymbol{v}_{t-1} + \boldsymbol{b}_t$ where $\odot$ is the element-wise multiplication. Interestingly, we can see that the GRU and LSTM state recurrences resemble the vector formulation. In this section, we show that GRUs and LSTMs are trainable via parallel scan by simplifying and removing several hidden state dependencies from their various gates. Building on this, we further simplify these RNNs by removing their constraints on output range, (i.e., $\tanh$) and ensuring the outputs are time-independent in scale. Combining the steps, we describe minimal versions of GRUs and LSTMs (minGRUs and minLSTMs) that are trainable via parallel scan and perform comparably to Transformers and recently proposed sequence methods.

3.1 A Minimal GRU: minGRU

3.1.1 Step 1: Drop previous hidden state dependencies from gates

Revisiting GRU's hidden state recurrence which works as follows:

$$\boldsymbol{h}_t = (\boldsymbol{1} - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t$$

We can observe that the recurrence resembles the aforementioned parallel scan's formulation where $\boldsymbol{a}_t \leftarrow (\mathbf{1} - \boldsymbol{z}_t)$, $\boldsymbol{b}_t \leftarrow \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t$, and $\boldsymbol{v}_t \leftarrow \boldsymbol{h}_t$. However, $\boldsymbol{z}_t$ and $\tilde{\boldsymbol{h}}_t$ are dependent on previous hidden states $\boldsymbol{h}_{t-1}$, i.e., $\boldsymbol{z}_t = \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]))$ and $\tilde{\boldsymbol{h}}_t = \tanh(\text{Linear}_{d_h}([\boldsymbol{x}_t, r_t \odot \boldsymbol{h}_{t-1}]))$. As a result, it is not possible to apply the parallel scan as is since the algorithm's inputs $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ and $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n$ are conditional on already knowing its outputs $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_{n-1}$.

We can remedy this by simplifying GRUs, removing their previous hidden state (i.e., $\boldsymbol{h}_{t-1}$) dependencies. Specifically, the changes are as follows:

$$\begin{aligned} \boldsymbol{z}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\ \boldsymbol{r}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\ \tilde{\boldsymbol{h}}_t &= \tanh(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}])) \end{aligned} \quad \Rightarrow \quad \begin{aligned} \boldsymbol{z}_t &= \sigma(\text{Linear}_{\boldsymbol{d}_h}(\boldsymbol{x}_t)) \\ \tilde{\boldsymbol{h}}_t &= \tanh(\text{Linear}_{\boldsymbol{d}_h}(\boldsymbol{x}_t)) \end{aligned}$$

By removing the dependence on $\boldsymbol{h}_{t-1}$ from the candidate hidden state $\tilde{\boldsymbol{h}}_t$, the reset gate $\boldsymbol{r}_t$ that would control $\boldsymbol{h}_{t-1}$ weight is also no longer needed and is removed. Without the dependencies on previous hidden states, the inputs to the algorithm $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ and $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n$ are all easily computed in parallel and can thus be used to compute $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n$ efficiently via the parallel scan.

3.1.2 Step 2: Drop range restriction of candidate states

In GRU's hidden state recurrence, the proportion carried over from the previous hidden state $(\mathbf{1} - \boldsymbol{z}_t)$ and the amount added for the new candidate hidden state $(\boldsymbol{z}_t)$ sum to 1. As a result, the scale of GRU's hidden state value is time-independent. Instead, the scale of its hidden state depends on that of its candidate hidden states $\tilde{\boldsymbol{h}}_t$. The hyperbolic tangent function ($\tanh$) plays a crucial role in LSTMs and GRUs, restricting the range of (candidate) hidden states, i.e., $\tilde{\boldsymbol{h}}_t, \boldsymbol{h}_t \in (-1, 1)^{d_h}$. The $\tanh$ helps stabilize the training and mitigates vanishing gradients that result from applying sigmoid (σ) activations to linear transformations of the hidden state (e.g., $\boldsymbol{z}_t = \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]))$). In the previous step, these hidden state dependencies were removed. As such, we can simplify GRU further by removing the range restriction ($\tanh$) on the (candidate) hidden states as follows:

$$\tilde{\boldsymbol{h}}_t = \tanh(\text{Linear}_{d_h}(\boldsymbol{x}_t)) \quad \Rightarrow \quad \tilde{\boldsymbol{h}}_t = \text{Linear}_{d_h}(\boldsymbol{x}_t)$$

3.1.3 minGRU

Combining the two simplification steps results in a minimal version of GRU (minGRU):

GRU

$$\begin{aligned} \boldsymbol{h}_t &= (\mathbf{1} - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t \\ \boldsymbol{z}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\ \boldsymbol{r}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\ \tilde{\boldsymbol{h}}_t &= \tanh(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}])) \end{aligned}$$

$$\Rightarrow$$

minGRU

$$\begin{aligned} \boldsymbol{h}_t &= (\mathbf{1} - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t \\ \boldsymbol{z}_t &= \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t)) \\ \tilde{\boldsymbol{h}}_t &= \text{Linear}_{d_h}(\boldsymbol{x}_t) \end{aligned}$$

The resulting model is significantly more efficient than the original GRU (1) requiring only $O(2d_h d_x)$ parameters instead of GRU's $O(3d_h(d_x + d_h))$ parameters where d_x, d_h corresponds to the sizes of x_t and h_t respectively. In terms of training, minGRU (2) can be trained in parallel using the parallel scan algorithm, speeding up training significantly. In Section 4.1, we show that this corresponded to a $175\times$ speedup in training steps for a sequence length of 512 on a T4 GPU. The parameter efficiency gains are also significant. Typically, in RNNs, state expansion is performed (i.e., $d_h = \alpha d_x$ where $\alpha \geq 1$) allowing the models to more readily learn features from their inputs. minGRU uses approximately $33\%, 22\%, 17\%$, or 13% of parameters compared to GRU when $\alpha = 1, 2, 3$, or 4 respectively.

3.2 A Minimal LSTM: minLSTM

3.2.1 Step 1: Drop previous hidden state dependencies from gates

Revisiting LSTMs, we focus on their cell state recurrence which works as follows:

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$$

Similar to GRU's hidden state, we can see that LSTM's cell state recurrence resembles the aforementioned parallel scan's formulation $\boldsymbol{v}_t = \boldsymbol{a}_t \odot \boldsymbol{v}_{t-1} + \boldsymbol{b}_t$ where $\boldsymbol{a}_t \leftarrow \boldsymbol{f}_t$, $\boldsymbol{b}_t \leftarrow \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$, and $\boldsymbol{v}_t \leftarrow \boldsymbol{c}_t$. However, $\boldsymbol{f}_t$, $\boldsymbol{i}_t$ and $\tilde{\boldsymbol{c}}_t$ are dependent on the previous hidden state $\boldsymbol{h}_t$. As such, LSTM's cell state recurrence is unable to apply the parallel scan algorithm as is. We can address this in a similar fashion to GRU by removing their hidden state dependencies as follows:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\boldsymbol{i}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\tilde{\boldsymbol{c}}_t &= \tanh(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]))
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
\boldsymbol{f}_t &= \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t)) \\
\boldsymbol{i}_t &= \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t)) \\
\tilde{\boldsymbol{c}}_t &= \tanh(\text{Linear}_{d_h}(\boldsymbol{x}_t))
\end{aligned}$$

3.2.2 Step 2: Drop range restriction of candidate states

Similar to GRUs, LSTMs leverage the hyperbolic tangent function ($\tanh$) to restrict the range of its states between $(-1, 1)$. LSTMs apply the range restriction twice: once when computing the candidate cell state and once computing its hidden state. In this step, we drop both as follows:

$$\begin{aligned}
\tilde{\boldsymbol{c}}_t &= \tanh(\text{Linear}_{d_h}(\boldsymbol{x}_t)) \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
\tilde{\boldsymbol{c}}_t &= \text{Linear}_{d_h}(\boldsymbol{x}_t) \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \boldsymbol{c}_t
\end{aligned}$$

3.2.3 Step 3: Ensure output is time-independent in scale

In many sequence modelling settings (e.g., text generation), the optimization objective/target is time-independent in scale. Recall LSTM's cell state recurrence $\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$ where $\boldsymbol{i}_t, \boldsymbol{f}_t \in (0,1)^{d_h}$, and GRU's hidden state recurrence[2], $\boldsymbol{h}_t^{GRU} = (\mathbf{1} - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1}^{GRU} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t^{GRU}$ where $\boldsymbol{z}_t \in (0,1)^{d_h}$. GRUs retain $(\mathbf{1} - \boldsymbol{z}_t) \in (0,1)$ of the previous hidden state and add $\boldsymbol{z}_t$ of the new candidate state. Since these proportions sum to $\mathbf{1}$, the model ensures its outputs (i.e., hidden states) are *time-independent* in scale. In contrast, LSTM's forget and input gates are computed independently (e.g., $\boldsymbol{f}_t, \boldsymbol{i}_t \to \mathbf{1}$ or $\boldsymbol{f}_t, \boldsymbol{i}_t \to \mathbf{0}$), making its cell states *time-dependent* in scale[3] and optimization more difficult. As such, we ensure LSTM's output is time-independent in scale.

To do so, we can simply normalize the two gates, i.e., $\boldsymbol{f}'_t, \boldsymbol{i}'_t \leftarrow \frac{\boldsymbol{f}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}, \frac{\boldsymbol{i}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}$, ensuring that $\boldsymbol{f}'_t + \boldsymbol{i}'_t = \mathbf{1}$ and the scale of LSTM's cell state is time-independent. Ensuring that the hidden state is time-independent in scale, we also drop the output gate $\boldsymbol{o}_t$ which scales the hidden state. Without the output gate, the normalized hidden state is equal to the cell state, i.e., $\boldsymbol{h}_t = \boldsymbol{o}_t \odot \boldsymbol{c}_t \Rightarrow \boldsymbol{h}_t = \boldsymbol{c}_t$, making having both a hidden and cell state unnecessary. As such, we drop the cell state as well. In summary, the modifications are as follows:

$$\begin{aligned}
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \boldsymbol{c}_t \\
\boldsymbol{o}_t &= \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t)) \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \\
\tilde{\boldsymbol{c}}_t &= \text{Linear}_{d_h}(\boldsymbol{x}_t)
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
\boldsymbol{h}_t &= \boldsymbol{f}'_t \odot \boldsymbol{h}_{t-1} + \boldsymbol{i}'_t \odot \tilde{\boldsymbol{h}}_t \\
\tilde{\boldsymbol{h}}_t &= \text{Linear}_{d_h}(\boldsymbol{x}_t) \\
\boldsymbol{f}'_t, \boldsymbol{i}'_t &\leftarrow \frac{\boldsymbol{f}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}, \frac{\boldsymbol{i}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}
\end{aligned}$$

Notably, GRUs do not need this step as their outputs are already time-independent in scale.

3.2.4 minLSTM

Combining the three steps results in a minimal version of LSTM (minLSTM):

[2] A superscript is added to differentiate GRU's hidden state from LSTM's.

[3] For example, $\boldsymbol{c}_t \to \boldsymbol{c}_0 + \sum_{i=1}^{t} \tilde{\boldsymbol{c}}_t$ when $\boldsymbol{f}_{1:t}, \boldsymbol{i}_{1:t} \to 1$, growing in scale as the sequence length increases.

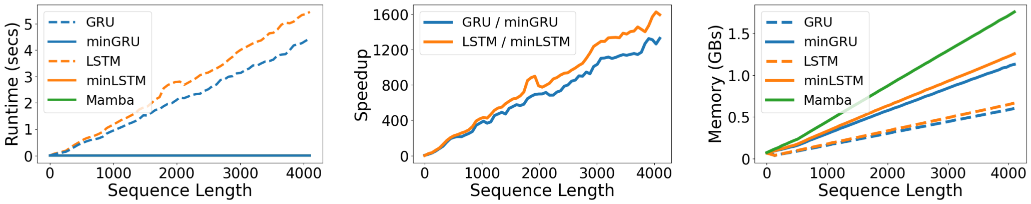

Figure 1: Training runtime (left), speedup (middle), and memory footprint (right) on a T4 GPU for a batch size of 64. In the training runtime plot (left), minGRU, minLSTM, and Mamba lines overlap. These methods are approximately the same in training runtime.

LSTM

$$
\begin{aligned}
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) \\
\boldsymbol{o}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \\
\boldsymbol{f}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\boldsymbol{i}_t &= \sigma(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \\
\tilde{\boldsymbol{c}}_t &= \tanh(\text{Linear}_{d_h}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]))
\end{aligned}
$$

$$\Rightarrow$$

minLSTM

$$
\begin{aligned}
\boldsymbol{h}_t &= \boldsymbol{f}'_t \odot \boldsymbol{h}_{t-1} + \boldsymbol{i}'_t \odot \tilde{\boldsymbol{h}}_t \\
\boldsymbol{f}_t &= \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t)) \\
\boldsymbol{i}_t &= \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t)) \\
\tilde{\boldsymbol{h}}_t &= \text{Linear}_{d_h}(\boldsymbol{x}_t) \\
\boldsymbol{f}'_t, \boldsymbol{i}'_t &\leftarrow \frac{\boldsymbol{f}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}, \frac{\boldsymbol{i}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}
\end{aligned}
$$

The minimal version (minLSTM) is significantly more efficient (1) requiring only $O(3d_h d_x)$ parameters compared to LSTM's $O(4d_h(d_x + d_h))$. Furthermore, minLSTM (2) can be trained in parallel using the parallel scan algorithm, speeding up training significantly. For example, in Section 4.1, we found that minLSTM corresponded to a $235\times$ speedup for a sequence of length 512 compared to LSTM on a T4 GPU. In terms of parameter efficiency, minLSTM uses only $38\%, 25\%, 19\%$, or 15% of parameters compared to LSTM when $\alpha = 1, 2, 3$, or 4 respectively where $d_h = \alpha d_x$.

4 Were RNNs All We Needed?

In this section, we compare the minimal versions (minLSTMs and minGRUs) with their traditional counterparts (LSTMs and GRUs) and modern sequence models. Pseudocode, PyTorch implementation, and detailed information regarding the experiment setup are available in the Appendix.

4.1 Minimal LSTMs and GRUs are very efficient

At test time, recurrent sequence models are rolled out sequentially, making their inferences efficient. Instead, the bottleneck of traditional RNNs is their training which requires linear training time (backpropagating through time) which resulted in their eventual deprecation. The renewed interest in recurrent sequence models is due to many new architectures being efficiently trained in parallel (Gu et al., 2021). In this section, we compare the resources required to train the traditional RNNs (LSTM and GRU), their minimal versions (minLSTM and minGRU), and a recent state-of-the-art sequence model. In particular, we focus on the comparison with Mamba (Gu & Dao, 2024) which has seen significant popularity recently. For these experiments, we consider a batch size of 64 and vary the sequence length. We measure the total runtime and memory complexity of performing a forward pass through the models, computing a loss, and computing gradients via a backward pass.

Runtime. In terms of runtime (see Figure 1 (left)), the simplified versions of LSTM and GRU (minLSTM and minGRU) Mamba achieve similar runtimes. Averaging over 100 runs, the runtime for sequence lengths of 512 for minLSTM, minGRU, and Mamba were 2.97, 2.72, and 2.71 milliseconds respectively. For a sequence with length 4096, the runtime were 3.41, 3.25, and 3.15 respectively. In contrast, the traditional RNN counterparts (LSTMs and GRUs) required a runtime that scaled linearly with respect to sequence length. For a sequence length of 512, minGRUs and minLSTMs were $175\times$ and $235\times$ faster per training step (see Figure 1 (middle)) than GRUs and

LSTMs on a T4 GPU. The improvement is even more significant as sequences grow in length with minGRUs and minLSTMs being $1324\times$ and $1361\times$ faster for a sequence length of 4096. As such, in a setting where minGRU would take a day to finish training for a fixed number of epochs, its traditional counterpart GRU could take over 3 years.

Memory. By leveraging a parallel scan algorithm to compute the outputs in parallel efficiently, minGRU, minLSTM, and Mamba create a larger computational graph, thus needing more memory compared to traditional RNNs (see Figure 1 (right)). The minimal variants (minGRU and minLSTM) use $\sim 88\%$ more memory compared to their traditional counterpart. Mamba uses 56% more memory compared to minGRU. In practice, however, runtime is the bottleneck when training RNNs.

Effect of removing $\boldsymbol{h}_{t-1}$. The original LSTM and GRU compute their various gates using their inputs $\boldsymbol{x}_t$ and previous hidden states $\boldsymbol{h}_{t-1}$. These models leverage their time-dependent gates to learn complex functions. However, minLSTM and minGRU's training efficiencies are achieved by dropping their gates' dependencies on the previous hidden states $\boldsymbol{h}_{t-1}$. As a result, minLSTM and minGRU's gates are dependent only on their inputs $\boldsymbol{x}_t$, resulting in a simpler recurrent module. As such, the gates of a model consisting of a single layer of minLSTM or minGRU are *time-independent* due to being conditioned on *time-independent* inputs $\boldsymbol{x}_{1:n}^{(1)}$.

However, in deep learning, models are constructed by stacking modules. Although the inputs to the first layer $\boldsymbol{x}_{1:n}^{(1)}$ is *time-independent*, its outputs $\boldsymbol{h}_{1:n}^{(1)}$ are *time-dependent* and are used as the inputs to the second layer, i.e., $\boldsymbol{x}_{1:n}^{(2)} \leftarrow \boldsymbol{h}_{1:n}^{(1)}$. As such, beginning from the second layer onwards, minLSTM and minGRU's gates will also be time-dependent, resulting in the modelling of more complex functions. In Table 1, we compare the performance of the models with varying numbers of layers on the Selective Copying Task from the Mamba paper (Gu & Dao, 2024). We can immediately see the impact of the time dependencies: increasing the number of layers to 2 or more drastically increases the model's performance.

Model	# Layers	Accuracy
MinLSTM	1	37.6 ± 2.0
	2	85.7 ± 5.8
	3	96.0 ± 2.8
MinGRU	1	37.0 ± 2.3
	2	96.8 ± 3.2
	3	99.5 ± 0.2

Table 1: Comparison of the number of layers on the Selective Copying Task (Gu & Dao, 2024).

Training Stability. Another effect of the number of layers is increased stability with decreased variance in the accuracy as the number of layers increases (see Table 1). Furthermore, although minLSTM and minGRU both solve the Selective Copying task, we can see that minGRU is an empirically more stable method than minLSTM, solving the task with more consistency and lower variance. minLSTM discards old information and adds new information, controlling the ratio with two sets of parameters (forget and input gate). During training, the two sets of parameters are tuned in different directions, making the ratio harder to control and optimize. In contrast, minGRU's discarding and adding of information is controlled by a single set of parameters (update gate), making it easier to optimize.

4.2 Minimal LSTMs and GRUs perform well

In the previous section, we showed the significant efficiency gains achieved by simplifying traditional RNNs. Here, we explore the empirical performance aspect of these minimal versions of LSTMs and GRUs compared to several popular sequence models.

Selective Copy. We consider the long-range Selective Copying task from the Mamba paper (Gu & Dao, 2024). Unlike the original Copying task (Arjovsky et al., 2016), the Selective Copying task's input elements are randomly spaced relative to their output, making the task harder. To solve the task, models are required to perform content-aware reasoning, memorizing relevant and filtering out irrelevant tokens.

In Table 2, we compare the simplified versions of LSTMs and GRUs (minLSTM and minGRU) against well-known recurrent sequence models that can trained in parallel: S4 (Gu et al., 2021), H3 (Fu et al., 2023), Hyena (Poli et al., 2023), and Mamba (S6) (Gu & Dao, 2024). The results for these baselines are quoted from the Mamba paper. Out of all of these baselines, only S6 from Mamba's paper is capable of solving this task. minGRU and minLSTM are also capable of solving

Dataset	DT	DS4	DAaren	DMamba	minLSTM	minGRU
HalfCheetah-M	42.6	42.5	42.2	42.8	42.7 ± 0.7	43.0 ± 0.4
Hopper-M	68.4	54.2	80.9	83.5	85.0 ± 4.4	79.4 ± 8.2
Walker-M	75.5	78.0	74.4	78.2	72.0 ± 7.5	73.3 ± 3.3
HalfCheetah-M-R	37.0	15.2	37.9	39.6	38.6 ± 1.1	38.5 ± 1.1
Hopper-M-R	85.6	49.6	77.9	82.6	88.5 ± 4.7	90.5 ± 0.9
Walker-M-R	71.2	69.0	71.4	70.9	69.7 ± 10.7	72.8 ± 8.9
HalfCheetah-M-E	88.8	92.7	75.7	91.9	85.4 ± 1.7	86.3 ± 0.5
Hopper-M-E	109.6	110.8	103.9	111.1	110.3 ± 1.6	109.7 ± 2.7
Walker-M-E	109.3	105.7	110.5	108.3	110.3 ± 0.5	110.3 ± 0.4
Average	76.4	68.6	75.0	78.8	78.1	78.2

Table 3: Reinforcement Learning results on the D4RL (Fu et al., 2020) datasets. We report the expert normalized returns (higher is better), following (Fu et al., 2020), averaged across five random seeds. The minimal versions of LSTM and GRU, minLSTM and minGRU outperform Decision S4 (David et al., 2023) and perform comparably with Decision Mamba (Ota, 2024), (Decision) Aaren (Feng et al., 2024) and Decision Transformer (Chen et al., 2021).

the Selective Copying task, achieving comparable performance to S6 and outperforming all other baselines. LSTMs and GRUs leverage content-aware gating mechanisms, making these minimal versions sufficient for solving this task that many popular sequence models fail to solve.

Reinforcement Learning. Next, we consider the MuJoCo locomotion tasks from the D4RL benchmark (Fu et al., 2020). Specifically, we consider the three environments: HalfCheetah, Hopper, and Walker. For each environment, the models are trained on three datasets of varying data quality: Medium (M), Medium-Replay (M-R), and Medium-Expert (M-E).

In Table 3, we compare minLSTM and minGRU with various Decision Transformer variants, including the original Decision Transformer (DT) (Chen et al., 2021), Decision S4 (DS4) (David et al., 2023), Decision Mamba (Ota, 2024), and (Decision) Aaren (Feng et al., 2024). The baseline results are retrieved from the Decision Mamba and Aaren papers. minLSTM and minGRU outperform Decision S4 and achieve performance competitive with Decision Transformer, Aaren, and Mamba. Unlike other recurrent methods, Decision S4 is a model whose recurrence transitions are not input-aware, affecting their performance. In terms of average score across the $3 \times 3 = 9$ datasets, minLSTM and minGRU outperform all the baselines except for Decision Mamba where the difference is marginal.

Model	Layer	Accuracy
H3	Hyena	30.1
Mamba	Hyena	28.4
S4	S4	18.3
H3	S4	57.0
Mamba	S4	56.4
S4	S6	97.0
H3	S6	99.7
Mamba	S6	99.8
minGRU	minGRU	99.5 ± 0.2
minLSTM	minLSTM	96.0 ± 2.8

Table 2: Selective Copy Task. minLSTM, minGRU, and Mamba's S6 (Gu & Dao, 2024) are capable of solving this task. Other methods such as S4, H3, and Hyena at best only partially solve the task.

Language Modelling. Finally, we consider a language modelling task. In this setting, we train a character-level GPT on the works of Shakespeare using the nanoGPT (Karpathy, 2022) framework. In Figure 2, we plot the learning curves with a cross-entropy loss comparing the proposed minimal LSTM and GRU (minLSTM and minGRU) with Mamba and Transformers. We found that minGRU, minLSTM, Mamba, and Transformers achieved comparable test losses of 1.548, 1.555, 1.575, and 1.547 respectively. Mamba performed slightly worse than the other models but trained faster, particularly in the early stages, achieving its best performance at 400 steps while minGRU and minLSTM continued training until 575 and 625 steps respectively. In contrast, Transformers trained significantly slower, requiring 2000 steps ($\sim 2.5\times$) more training steps than minGRU to achieve comparable performance, making it significantly slower and more resource-intensive to train (quadratic complexity compared to minGRU, minLSTM, and Mamba's linear complexity).

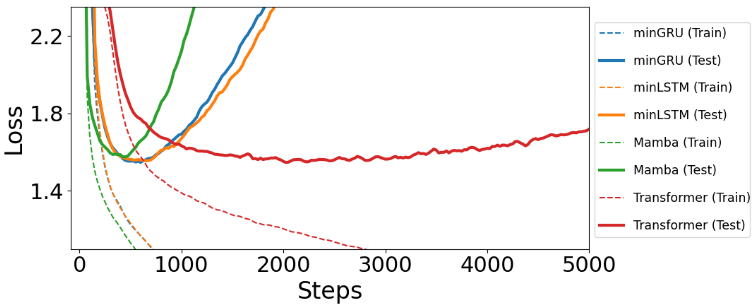

Figure 2: Language Modelling results on the Shakespeare dataset. Minimal versions of decade-old RNNs (LSTMs and GRUs) performed comparably to Mamba and Transformers. Transformers required $\sim 2.5\times$ more training steps to achieve comparable performance, overfitting eventually.

5 Related Work

In this section, we provide a discussion of the similarities and differences between existing recurrent sequence models and the simplified versions of LSTMs and GRUs (minLSTM and minGRU).

State-Space Models (SSMs). Although Mamba (Gu & Dao, 2024) and state-space models have gained significant popularity recently, the steps towards the recent success of Mamba began years ago. Gu et al. (2020) first proposed a discretized structured state-space model. Gu et al. (2021) scaled the idea up, introducing S4. The success of S4 became the basis for many future works (Gu et al., 2022; Gupta et al., 2022; Hasani et al., 2023; Smith et al., 2023) and state-space model applications in language (Mehta et al., 2023), audio (Goel et al., 2022), and more. Recently, Mamba was a significant breakthrough in SSM, outperforming previous methods and garnering substantial attention. A major novelty in Mamba was the proposal of S6, a state-space model whose transition matrices are input-dependent (i.e., A_t and B_t are functions of x_t). In contrast, earlier state-space model transition matrices were input-independent, limiting their expressivity. The success of Mamba and state-space models led to the writing of several survey papers (Wang et al., 2024; Patro & Agneeswaran, 2024; Qu et al., 2024).

Recurrent Versions of Attention. Another direction that proposed efficient recurrent sequence models is that of attention. Building on variations of linear attention (Katharopoulos et al., 2020), several papers have introduced recurrent versions that can be computed in parallel. Notably, Sun et al. (2023) and Qin et al. (2023) introduced variants that use an input-independent gating mechanism (decay factor). More recently, Katsch (2023) and Yang et al. (2024) proposed linear attention variants that use input-dependent gating. Feng et al. (2024) showed softmax attention can be viewed as an RNN and proposed a recurrent model based on their RNN formulation.

Parallelizable RNNs. Alternatively, several papers have proposed RNNs that can be trained efficiently in parallel. Orvieto et al. (2023) proposed an RNN that leverages complex diagonal recurrences and an exponential parameterization. Beck et al. (2024) proposed various enhancements to LSTM such as exponential gating, covariance update rule, and a normalizer state.

Although these three directions of designing efficient recurrent sequence models have proposed vastly different architectures, the core recurrent component of these models is remarkably similar. For example, although state-space models are typically written as $\boldsymbol{h}_t = \boldsymbol{A}_t\boldsymbol{h}_{t-1} + \boldsymbol{B}_t\boldsymbol{x}_t$, in practice, the transition matrices are typically diagonal for efficiency reasons. As such, Mamba's S6 (Gu & Dao, 2024) can be viewed as $\boldsymbol{h}_t = \mathrm{diag}(\boldsymbol{A}_t)\odot\boldsymbol{h}_{t-1} + \mathrm{diag}(\boldsymbol{B}_t)\odot\boldsymbol{x}_t$ where $\boldsymbol{A}_t$ and $\boldsymbol{B}_t$ are functions of $\boldsymbol{x}_t$. In contrast, consider the minimal version of GRU $\boldsymbol{h}_t = (\boldsymbol{1} - \boldsymbol{z}_t)\odot\boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \mathrm{Linear}_{d_h}(\boldsymbol{x}_t)$ and minimal version of LSTM $\boldsymbol{h}_t = \boldsymbol{f}'_t \odot \boldsymbol{h}_{t-1} + \boldsymbol{i}'_t \odot \mathrm{Linear}_{d_h}(\boldsymbol{x}_t)$. The recurrences of these models are similar. The major difference between these minimal RNNs, Mamba's S6, and other models is how their transitions (e.g., $\boldsymbol{z}_t$, $\boldsymbol{i}'_t$, $\boldsymbol{f}'_t$, $\boldsymbol{A}_t$, and $\boldsymbol{B}_t$) are computed from the input token $\boldsymbol{x}_t$.

Parallel Scan. Generalizing across the families of methods (including minLSTM and minGRU), these recent sequence models can be viewed as members of the same family of functions trainable via a parallel scan: $\boldsymbol{v}_t = \boldsymbol{a}_t \odot \boldsymbol{v}_{t-1} + \boldsymbol{b}_t$ (see Section 2.3) where $\boldsymbol{a}_t$ and $\boldsymbol{b}_t$ are functions of the input token $\boldsymbol{x}_t$. Improving upon the parallel scan algorithm, several models (Yang et al., 2024; Gu & Dao,

2024) such as Mamba have proposed specialized hardware-efficient methods that leverage GPU's memory hierarchy to reduce high I/O costs and speed up training. In our work, we implemented minLSTM and minGRU in plain PyTorch. However, due to the structural similarities in recurrences amongst the numerous methods that leverage parallel scan, many techniques such as chunking that apply to one work for speeding up training can also apply to others such as minGRU and minLSTM.

Parameter Initializations. Unrolling the recurrences of these new recurrent sequence models over time often results in their outputs and gradients vanishing/exploding (Wang et al., 2024) due to time dependency in their output's scale. To ensure model stability, the parameters of many models such as state-space models are initialized according to special distributions (Gu et al., 2020, 2022; Orvieto et al., 2023). In contrast, we found that minLSTM and minGRU are already stable using the default PyTorch initialization. Unlike SSMs, minLSTM and minGRU's outputs are time-independent in scale, avoiding potential instabilities.

6 Conclusion

In this work, we revisited RNNs from over a decade ago: LSTMs and GRUs. We show that these models are trainable via the parallel scan algorithm by removing their hidden state dependencies from their gates. Simplifying these models further, we removed their constraints on output range and ensured their output was time-independent in scale. These steps result in their minimal versions (minLSTM and minGRU). Empirically, we showed that minLSTM and minGRU (1) address the computational limitations of their traditional counterparts and (2) are as computationally efficient as Mamba, a popular recent state-of-the-art recurrent sequence model, and (3) are competitive in performance with recent sequence models. Considering the strong empirical performance of these simplified RNNs and their fundamental similarities with many recently proposed recurrent sequence methods, we question ”Were RNNs all we needed?”

Limitations

Our experiments were run on P100 (16 GBs) and T4 (16 GBs) GPUs. Due to computation limitations, our experiments are smaller in scale compared to works such as Mamba (Gu & Dao, 2024) which leveraged A100 80GB GPUs. To fit the selective copy task on the GPU, we leveraged gradient accumulation for training, splitting the standard batch size in half and slowing training significantly. Nonetheless, we hypothesize that these conclusions generalize to larger-scale settings due to the fundamental similarities between the minimal RNNs (minLSTM and minGRU) and many recent sequence methods.

References

Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *International conference on machine learning*, pp. 1120–1128. PMLR, 2016.

Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xlstm: Extended long short-term memory. *arXiv preprint arXiv:2405.04517*, 2024.

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

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

Kyunghyun Cho, Bart Van Merrienboer, Caglar Gulcehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

Shmuel Bar David, Itamar Zimerman, Eliya Nachmani, and Lior Wolf. Decision s4: Efficient sequence-based rl via state spaces layers. In *The Eleventh International Conference on Learning Representations*, 2023.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019.

Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. ISSN 0364-0213.

Leo Feng, Frederick Tung, Hossein Hajimirsadeghi, Mohamed Osama Ahmed, Yoshua Bengio, and Greg Mori. Attention as an rnn. *arXiv preprint arXiv:2405.13956*, 2024.

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

Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4rl: Datasets for deep data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020.

Karan Goel, Albert Gu, Chris Donahue, and Christopher Ré. It's raw! audio generation with state-space models. In *International Conference on Machine Learning*, pp. 7616–7633. PMLR, 2022.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2024.

Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. Hippo: Recurrent memory with optimal polynomial projections. *Advances in neural information processing systems*, 33: 1474–1487, 2020.

Albert Gu, Karan Goel, and Christopher Re. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*, 2021.

Albert Gu, Karan Goel, Ankit Gupta, and Christopher Ré. On the parameterization and initialization of diagonal state space models. *Advances in Neural Information Processing Systems*, 35:35971–35983, 2022.

Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces. *Advances in Neural Information Processing Systems*, 35:22982–22994, 2022.

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

Franz A Heinsen. Parallelization of an ubiquitous sequential computation. *arXiv preprint arXiv:2311.06281*, 2023.

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

Andrej Karpathy. NanoGPT. `https://github.com/karpathy/nanoGPT`, 2022.

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

Tobias Katsch. Gateloop: Fully data-controlled linear recurrence for sequence modeling. *arXiv preprint arXiv:2311.01927*, 2023.

Harsh Mehta, Ankit Gupta, Ashok Cutkosky, and Behnam Neyshabur. Long range language modeling via gated state spaces. In *The Eleventh International Conference on Learning Representations*, 2023.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. In *International Conference on Machine Learning*, pp. 26670–26698. PMLR, 2023.

Toshihiro Ota. Decision mamba: Reinforcement learning via sequence modeling with selective state spaces. *arXiv preprint arXiv:2403.19925*, 2024.

Badri Narayana Patro and Vijay Srinivas Agneeswaran. Mamba-360: Survey of state space models as transformer alternative for long sequence modelling: Methods, applications, and challenges. *arXiv preprint arXiv:2404.16112*, 2024.

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

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

Zhen Qin, Dong Li, Weigao Sun, Weixuan Sun, Xuyang Shen, Xiaodong Han, Yunshen Wei, Baohong Lv, Fei Yuan, Xiao Luo, et al. Scaling transnormer to 175 billion parameters. *arXiv preprint arXiv:2307.14995*, 2023.

Haohao Qu, Liangbo Ning, Rui An, Wenqi Fan, Tyler Derr, Xin Xu, and Qing Li. A survey of mamba. *arXiv preprint arXiv:2408.01129*, 2024.

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

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

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

Xiao Wang, Shiao Wang, Yuhe Ding, Yuehang Li, Wentao Wu, Yao Rong, Weizhe Kong, Ju Huang, Shihao Li, Haoxiang Yang, et al. State space model for new-generation network alternative to transformers: A survey. *arXiv preprint arXiv:2404.09516*, 2024.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. In *International Conference on Machine Learning*, 2024.

A Implementation Details: Vanilla Version

In this section, we provide the pseudocode and equivalent PyTorch code for minGRU and minLSTM. When performing repeated multiplications such as in many recurrent sequence models, numerical instabilities are common, especially during training. As such, we trained using a log-space implementation (see Section B) for improved numerical stability.

A.1 Pseudocode: Vanilla Version

A.1.1 minGRU: A Minimal GRU

Algorithm 1 Sequential Mode: Minimal Version of GRU (minGRU)

```
Input: $\boldsymbol{x}_t, \boldsymbol{h}_{t-1}$
Output: $\boldsymbol{h}_t$
  $\boldsymbol{z}_t \leftarrow \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t))$
  $\tilde{\boldsymbol{h}}_t \leftarrow \text{Linear}_{d_h}(\boldsymbol{x}_t)$
  $\boldsymbol{h}_t \leftarrow (\boldsymbol{1} - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t$
```

Algorithm 2 Parallel Mode: Minimal Version of GRU (minGRU)

```
Input: $\boldsymbol{x}_{1:t}, \boldsymbol{h}_0$
Output: $\boldsymbol{h}_{1:t}$
  $\boldsymbol{z}_{1:t} \leftarrow \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_{1:t}))$
  $\tilde{\boldsymbol{h}}_{1:t} \leftarrow \text{Linear}_{d_h}(\boldsymbol{x}_{1:t})$
  $\boldsymbol{h}_{1:t} \leftarrow \text{ParallelScan}((\boldsymbol{1} - \boldsymbol{z}_{1:t}), [\boldsymbol{h}_0, \boldsymbol{z}_{1:t} \odot \tilde{\boldsymbol{h}}_{1:t}])$
```

A.1.2 minLSTM: A Minimal LSTM

Algorithm 3 Sequential Mode: Minimal Version of LSTM (minLSTM)

```
Input: $\boldsymbol{x}_t, \boldsymbol{h}_{t-1}$
Output: $\boldsymbol{h}_t$
  $\boldsymbol{f}_t \leftarrow \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t))$
  $\boldsymbol{i}_t \leftarrow \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t))$
  $\boldsymbol{f}'_t, \boldsymbol{i}'_t \leftarrow \frac{\boldsymbol{f}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}, \frac{\boldsymbol{i}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}$
  $\tilde{\boldsymbol{h}}_t \leftarrow \text{Linear}_{d_h}(\boldsymbol{x}_t)$
  $\boldsymbol{h}_t \leftarrow \boldsymbol{f}'_t \odot \boldsymbol{h}_{t-1} + \boldsymbol{i}'_t \odot \tilde{\boldsymbol{h}}_t$
```

Algorithm 4 Parallel Mode: Minimal Version of LSTM (minLSTM)

```
Input: $\boldsymbol{x}_{1:t}, \boldsymbol{h}_0$
Output: $\boldsymbol{h}_{1:t}$
  $\boldsymbol{f}_{1:t} \leftarrow \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_{1:t}))$
  $\boldsymbol{i}_{1:t} \leftarrow \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_{1:t}))$
  $\boldsymbol{f}'_{1:t}, \boldsymbol{i}'_{1:t} \leftarrow \frac{\boldsymbol{f}_{1:t}}{\boldsymbol{f}_{1:t} + \boldsymbol{i}_{1:t}}, \frac{\boldsymbol{i}_{1:t}}{\boldsymbol{f}_{1:t} + \boldsymbol{i}_{1:t}}$
  $\tilde{\boldsymbol{h}}_{1:t} \leftarrow \text{Linear}_{d_h}(\boldsymbol{x}_{1:t})$
  $\boldsymbol{h}_{1:t} \leftarrow \text{ParallelScan}(\boldsymbol{f}'_{1:t}, [\boldsymbol{h}_0, \boldsymbol{i}'_{1:t} \odot \tilde{\boldsymbol{h}}_{1:t}])$
```

A.2 PyTorch Code: Vanilla Version

A.2.1 minGRU: A Minimal GRU

```
    def forward(self, x_t, h_prev):
        # x_t: (batch_size, input_size)
        # h_prev: (batch_size, hidden_size)

        z_t = torch.sigmoid(self.linear_z(x_t))
        h_tilde = self.linear_h(x_t)
        h_t = (1 - z_t) * h_prev + z_t * h_tilde
        return h_t
```

Listing 1: Sequential Mode: Minimal Version of GRU (minGRU)

```
    def forward(self, x, h_0):
        # x: (batch_size, seq_len, input_size)
        # h_0: (batch_size, 1, hidden_size)

        z = torch.sigmoid(self.linear_z(x))
        h_tilde = self.linear_h(x)
        h = parallel_scan((1 - z),
            torch.cat([h_0, z * tilde_h], dim=1))
        return h
```

Listing 2: Parallel Mode: Minimal Version of GRU (minGRU)

A.2.2 minLSTM: A Minimal LSTM

```
    def forward(self, x_t, h_prev):
        # x_t: (batch_size, input_size)
        # h_prev: (batch_size, hidden_size)

        f_t = torch.sigmoid(self.linear_f(x_t))
        i_t = torch.sigmoid(self.linear_i(x_t))
        tilde_h_t = self.linear_h(x_t)
        f_prime_t = f_t / (f_t + i_t)
        i_prime_t = i_t / (f_t + i_t)
        h_t = f_prime_t * h_prev + i_prime_t * tilde_h_t
        return h_t
```

Listing 3: Sequential Mode: Minimal Version of LSTM (minLSTM)

```
    def forward(self, x, h_0):
        # x: (batch_size, seq_len, input_size)
        # h_0: (batch_size, 1, hidden_size)

        f = torch.sigmoid(self.linear_f(x))
        i = torch.sigmoid(self.linear_i(x))
        tilde_h = self.linear_h(x)
        f_prime = f / (f + i)
        i_prime = i / (f + i)
        h = parallel_scan(f_prime,
            torch.cat([h_0, i_prime * tilde_h], dim=1))
        return h
```

Listing 4: Parallel Mode: Minimal Version of LSTM (minLSTM)

B Implementation Details: Log-Space Version (Additional Numerical Stability)

In this section, we detail the log-space version of minLSTM and minGRU for improved numerical stability. During training, the parallel modes are used to avoid backpropagation through time (BPTT), speeding up the training time significantly. At inference time, the sequential modes are used.

B.1 Parallel Scan: Log-Space Implementation

Recall that, the parallel scan's objective is to compute $\boldsymbol{h}_{1:t}$ where $\boldsymbol{h}_k = \boldsymbol{a}_k \odot \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$. In code, the vanilla parallel scan function would take as input: coefficients $\boldsymbol{a}_{1:t}$ and values $\boldsymbol{b}_{0:t}$. The function then outputs $\boldsymbol{h}_{1:t}$. For numerical stability, we consider a log-space implementation which takes as input $\log(\boldsymbol{a}_{1:t})$ and $\log(\boldsymbol{b}_{0:t})$ instead and outputs $\boldsymbol{h}_{1:t}$. The code for the parallel scan in log-space is included below and is based on the code by Heinsen (2023).

```
def parallel_scan_log(log_coeffs, log_values):
    # log_coeffs: (batch_size, seq_len, input_size)
    # log_values: (batch_size, seq_len + 1, input_size)
    a_star = F.pad(torch.cumsum(log_coeffs, dim=1), (0, 0, 1, 0))
    log_h0_plus_b_star = torch.logcumsumexp(
                            log_values - a_star, dim=1)
    log_h = a_star + log_h0_plus_b_star
    return torch.exp(log_h)[:, 1:]
```

Listing 5: Parallel scan based on Heinsen (2023). This function computes $\boldsymbol{h}_{1:t}$ given log coefficients $\log(\boldsymbol{a}_{1:t})$ and log values $\log(\boldsymbol{b}_{0:t})$.

B.2 Pseudocode: Log-Space Version

For maximal numerical stability, we rewrite the log-space versions of minGRU and minLSTM.

B.2.1 minGRU: A Minimal GRU

Recall minGRU's recurrence is as follows $\boldsymbol{h}_t \leftarrow (\boldsymbol{1} - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t$. As such, $\boldsymbol{a}_t \leftarrow (\boldsymbol{1} - \boldsymbol{z}_t)$ and $\boldsymbol{b}_t \leftarrow \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t$ where $\boldsymbol{z}_t = \sigma(\boldsymbol{k}_t)$ and $\boldsymbol{k}_t = \text{Linear}_{d_h}(x_t)$. As a result, $\log(\boldsymbol{a}_t) \leftarrow \log(\boldsymbol{1} - \boldsymbol{z}_t)$ and $\log(\boldsymbol{b}_t) \leftarrow \log(\boldsymbol{z}_t) + \log(\tilde{\boldsymbol{h}}_t)$. We can break down these down as follows:

$$\begin{aligned}
\log(\boldsymbol{z}_t) &= \log(\sigma(\boldsymbol{k}_t)) \\
&= \log\left(\frac{1}{1 + \exp(-\boldsymbol{k}_t)}\right) \\
&= -\text{Softplus}(-\boldsymbol{k}_t) \\
\log(\boldsymbol{1} - \boldsymbol{z}_t) &= \log\left(\frac{\exp(-\boldsymbol{k}_t)}{1 + \exp(-\boldsymbol{k}_t)}\right) \\
&= \log\left(\frac{1}{1 + \exp(\boldsymbol{k}_t)}\right) \\
&= -\text{Softplus}(\boldsymbol{k}_t)
\end{aligned}$$

where $\boldsymbol{k}_t = \text{Linear}_{d_h}(\boldsymbol{x}_t)$. However, we need to compute $\log(\tilde{\boldsymbol{h}})_t$ which is inconvenient if $\tilde{\boldsymbol{h}}_t$ has some negative values. We could use complex numbers and a complex number version of the parallel scan, but this would result in the parallel scan increasing in complexity. Instead, we propose to ensure that $\tilde{\boldsymbol{h}}_t > 0$. This be can done in a variety of ways. In our experiments, we added a continuous activation function g replacing $\tilde{\boldsymbol{h}}_t \leftarrow \text{Linear}_{d_h}(x_t)$ with $\tilde{\boldsymbol{h}}_t \leftarrow g(\text{Linear}_{d_h}(x_t))$ where $g(x) = \begin{cases} x + 0.5, & \text{if } x \geq 0 \\ \sigma(x), & \text{otherwise} \end{cases}$ and its log: $\log(g(x)) = \begin{cases} \log(x + 0.5), & \text{if } x \geq 0 \\ -\text{Softplus}(-x), & \text{otherwise} \end{cases}$.

At inference time, the sequential mode (Algorithm 5) is used. During training, the parallel mode (Algorithm 6) is used.

Algorithm 5 Sequential Mode: Minimal Version of GRU (minGRU) trained in log-space

```
Input: $\boldsymbol{x}_t, \boldsymbol{h}_{t-1}$
Output: $\boldsymbol{h}_t$
  $\boldsymbol{z}_t \leftarrow \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t))$
  $\tilde{\boldsymbol{h}}_t \leftarrow \text{g}(\text{Linear}_{d_h}(\boldsymbol{x}_t))$
  $\boldsymbol{h}_t \leftarrow (\boldsymbol{1} - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t$
```

Algorithm 6 Parallel Mode: Minimal Version of GRU (minGRU) for training in log-space

```
Input: $\boldsymbol{x}_{1:t}, \boldsymbol{h}_0$
Output: $\boldsymbol{h}_{1:t}$
  $\text{linear\_z} \leftarrow \text{Linear}_{d_h}$
  $\text{log\_}\boldsymbol{z}_{1:t} \leftarrow -\text{Softplus}(\text{linear\_z}(-\boldsymbol{x}_{1:t}))$
  $\text{log\_coeffs} \leftarrow -\text{Softplus}(\text{linear\_z}(\boldsymbol{x}_{1:t}))$
  $\text{log\_}\boldsymbol{h}_0 \leftarrow \text{log\_g}(h_0)$
  $\text{log\_}\tilde{\boldsymbol{h}}_{1:t} \leftarrow \text{log\_g}(\text{Linear}_{d_h}(\boldsymbol{x}_{1:t}))$
  $\boldsymbol{h}_{1:t} \leftarrow \text{ParallelScanLog}(\text{log\_coeffs}, [\text{log\_}\boldsymbol{h}_0, \text{log\_}\boldsymbol{z}_{1:t} + \text{log\_}\tilde{\boldsymbol{h}}_{1:t})$
```

B.2.2 minLSTM: A Minimal LSTM

We also derive minLSTM's log-space formulation as well. Recall minLSTM's recurrence is as follows $\boldsymbol{h}_t \leftarrow \boldsymbol{f}'_t \odot \boldsymbol{h}_{t-1} + \boldsymbol{i}'_t \odot \tilde{\boldsymbol{h}}_t$. As such, $\boldsymbol{a}_t \leftarrow \boldsymbol{f}'_t$ and $\boldsymbol{b}_t \leftarrow \boldsymbol{i}'_t \odot \tilde{\boldsymbol{h}}_t$. As a result, $\log(\boldsymbol{a}_t) \leftarrow \log(\boldsymbol{f}'_t)$ and $\log(\boldsymbol{b}_t) \leftarrow \log(\boldsymbol{i}'_t) + \log(\tilde{\boldsymbol{h}}_t)$.

$$\begin{aligned}
\log(\boldsymbol{f}'_t) &= \log\left(\frac{\boldsymbol{f}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}\right) \\
&= \log\left(\frac{1}{1 + \frac{\boldsymbol{i}_t}{\boldsymbol{f}_t}}\right) \\
&= -\log\left(1 + \frac{\boldsymbol{i}_t}{\boldsymbol{f}_t}\right) \\
&= -\log\left(1 + \exp\left(\log\left(\frac{\boldsymbol{i}_t}{\boldsymbol{f}_t}\right)\right)\right) \\
&= -\text{Softplus}\left(\log\left(\frac{\boldsymbol{i}_t}{\boldsymbol{f}_t}\right)\right) \\
&= -\text{Softplus}\left(\log(\boldsymbol{i}_t) - \log(\boldsymbol{f}_t)\right)
\end{aligned}$$

Recall that $\boldsymbol{i}_t$ and $\boldsymbol{f}_t$ are computed via sigmoid. In other words, $\boldsymbol{i}_t = \sigma(\boldsymbol{k}_t)$ and $\boldsymbol{f}_t = \sigma(\boldsymbol{p}_t)$ where $\boldsymbol{k}_t = \text{Linear}_{d_h}(\boldsymbol{x}_t)$ and $\boldsymbol{p}_t = \text{Linear}_{d_h}(\boldsymbol{x}_t)$. Furthermore, recall in minGRU's derivation we showed that $\log(\sigma(\boldsymbol{k}_t)) = -\text{Softplus}(-\boldsymbol{k}_t)$ Using this, we can simplify the computation as follows:

$$\begin{aligned}
\log(\boldsymbol{f}'_t) &= -\text{Softplus}\left(\log(\sigma(\boldsymbol{k}_t)) - \log(\sigma(\boldsymbol{p}_t))\right) \\
&= -\text{Softplus}\left(\text{Softplus}(-\boldsymbol{p}_t) - \text{Softplus}(-\boldsymbol{k}_t)\right)
\end{aligned}$$

Similarly, we also get that:

$$\log(\boldsymbol{i}'_t) = -\text{Softplus}\left(\text{Softplus}(-\boldsymbol{k}_t) - \text{Softplus}(-\boldsymbol{p}_t)\right)$$

Combining these derivations, we get the parallel mode (Algorithm 8) for efficient training.

Algorithm 7 Sequential Mode: Minimal Version of LSTM (minLSTM) trained in log-space

```
Input: $\boldsymbol{x}_t, \boldsymbol{h}_{t-1}$
Output: $\boldsymbol{h}_t$
    $\boldsymbol{f}_t \leftarrow \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t))$
    $\boldsymbol{i}_t \leftarrow \sigma(\text{Linear}_{d_h}(\boldsymbol{x}_t))$
    $\boldsymbol{f}'_t, \boldsymbol{i}'_t \leftarrow \frac{\boldsymbol{f}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}, \frac{\boldsymbol{i}_t}{\boldsymbol{f}_t + \boldsymbol{i}_t}$
    $\tilde{\boldsymbol{h}}_t \leftarrow g(\text{Linear}_{d_h}(\boldsymbol{x}_t))$
    $\boldsymbol{h}_t \leftarrow \boldsymbol{f}'_t \odot \boldsymbol{h}_{t-1} + \boldsymbol{i}'_t \odot \tilde{\boldsymbol{h}}_t$
```

Algorithm 8 Parallel Mode: Minimal Version of LSTM (minLSTM) for training in log-space

```
Input: $\boldsymbol{x}_{1:t}, \boldsymbol{h}_0$
Output: $\boldsymbol{h}_{1:t}$
    $\text{diff} \leftarrow \text{Softplus}(-\text{Linear}_{d_h}(\boldsymbol{x}_{1:t})) - \text{Softplus}(-\text{Linear}_{d_h}(\boldsymbol{x}_{1:t}))$
    $\text{log\_}\boldsymbol{f}'_{1:t} \leftarrow -\text{Softplus}(\text{diff})$
    $\text{log\_}\boldsymbol{i}'_{1:t} \leftarrow -\text{Softplus}(-\text{diff})$
    $\text{log\_}\boldsymbol{h}_0 \leftarrow \text{log\_g}(h_0)$
    $\text{log\_}\tilde{\boldsymbol{h}}_{1:t} \leftarrow \text{log\_g}(\text{Linear}_{d_h}(\boldsymbol{x}_{1:t}))$
    $\boldsymbol{h}_{1:t} \leftarrow \text{ParallelScanLog}(\text{log\_}\boldsymbol{f}'_{1:t}, [\text{log\_}\boldsymbol{h}_0, \text{log\_}\boldsymbol{i}'_{1:t} + \text{log\_}\tilde{\boldsymbol{h}}_{1:t})$
```

B.3 PyTorch Code: Log-Space Version

```python
def g(x):
    return torch.where(x >= 0, x+0.5, torch.sigmoid(x))
def log_g(x):
    return torch.where(x >= 0, (F.relu(x)+0.5).log(),
                        -F.softplus(-x))
```

Listing 6: The continuous function g ensures that $\tilde{\boldsymbol{h}}_t \leftarrow g(\text{Linear}_{d_h}(x_t))$ is positive.

B.3.1 minGRU: A Minimal GRU

```python
    def forward(self, x_t, h_prev):
        # x_t: (batch_size, input_size)
        # h_prev: (batch_size, hidden_size)

        z = torch.sigmoid(self.linear_z(x))
        h_tilde = g(self.linear_h(x))
        h_t = (1 - z) * h_prev + z * h_tilde
        return h_t
```

Listing 7: Sequential Mode: Minimal Version of GRU (minGRU) trained in log-space

```python
    def forward(self, x, h_0):
        # x: (batch_size, seq_len, input_size)
        # h_0: (batch_size, 1, hidden_size)

        k = self.linear_z(x)
        log_z = -F.softplus(-k)
        log_coeffs = -F.softplus(k)
        log_h_0 = log_g(h_0)
        log_tilde_h = log_g(self.linear_h(x))
        h = parallel_scan_log(log_coeffs,
                torch.cat([log_h_0, log_z + log_tilde_h], dim=1))
        return h
```

Listing 8: Parallel Mode: Minimal Version of GRU (minGRU) for training in log-space

B.3.2 minLSTM: A Minimal LSTM

```
    def forward(self, x_t, h_prev):
        # x_t: (batch_size, input_size)
        # h_prev: (batch_size, hidden_size)

        f_t = torch.sigmoid(self.linear_f(x_t))
        i_t = torch.sigmoid(self.linear_i(x_t))
        tilde_h_t = g(self.linear_h(x_t))
        f_prime_t = f_t / (f_t + i_t)
        i_prime_t = i_t / (f_t + i_t)
        h_t = f_prime_t * h_prev + i_prime_t * tilde_h_t
        return h_t
```

Listing 9: Sequential Mode: Minimal Version of LSTM (minLSTM) trained in log-space

```
    def forward(self, x, h_0):
        # x: (batch_size, seq_len, input_size)
        # h_0: (batch_size, 1, hidden_size)

        diff = F.softplus(-self.linear_f(x))  \
                        - F.softplus(-self.linear_i(x))
        log_f = -F.softplus(diff)
        log_i = -F.softplus(-diff)
        log_h_0 = log_g(h_0)
        log_tilde_h = log_g(self.linear_h(x))
        h = parallel_scan_log(log_f,
                torch.cat([log_h_0, log_i + log_tilde_h], dim=1))
        return h
```

Listing 10: Parallel Mode: Minimal Version of LSTM (minLSTM) for training in log-space

C Detailed Experiment Setup

In this section, we describe the experiment setup in detail.

C.1 Architecture

In all models, residual connections are added between layers and layer norms are applied before each layer.

Selective Copying. Each layer in the model consisted of (1) either a minLSTM or minGRU layer and (2) a linear layer.

Reinforcement Learning. In this work, we consider the Decision Transformer framework for (Offline) RL. Following prior works (Feng et al., 2024; Ota, 2024), we replace the Self-Attention module with our recurrent sequence modules: minLSTM and minGRU respectively.

Language Modelling. Prior works (Gu & Dao, 2024; Beck et al., 2024) apply a convolutional layer in addition to their recurrent sequence module. Following them, a layer of the model consists of an RNN (minLSTM and minGRU), a convolutional layer applied temporally with a kernel size of 4, and a two-layer MLP.

C.2 Hyperparameters and general experimental details

Selective Copying. Models are trained for $400,000$ steps with a batch size of 64, and using early stopping. The optimizer used is Adam with a learning rate of 3×10^{-4}. Due to GPU memory limitations, gradient accumulation is performed during training. Gradients for two batches of size 32 are accumulated for each gradient update and clipped to 1.0. Each model consists of 3 layers, an input dimension of 64, and a dropout ratio of 0.1. minLSTM and minGRU have an expansion factor of 6. Results for the baselines are referenced from the Mamba paper.

Reinforcement Learning. We follow the hyperparameter settings outlined by Ota (2024). For Hopper (Medium) and Hopper (Medium-Replay), an embedding dimension of 256 is used, while all other environments utilize an embedding dimension of 128. The learning rate is set to 1×10^{-4} for Hopper (Medium), Hopper (Medium-Replay), and Walker (Medium). For all other environments and datasets, the learning rate is 1×10^{-3}. The models are optimized using AdamW with a weight decay of 1×10^{-4} and a linear warmup for $10,000$ steps. Each model consists of 3 layers and has a dropout ratio of 0.1. The models are trained for $100,000$ steps with a batch size of 64. Gradients are clipped to 0.25. Results for the baselines are referenced from the Mamba and Aaren papers.

Language Modelling. The models are optimized using AdamW with a learning rate of 1×10^{-3}. Each model consists of three layers, a dropout ratio of 0.2, and an embedding dimension of 384. Training is done with 5000 steps using a batch size of 64 and evaluated every 25 steps. Gradients are clipped to 1.0. The Transformer is configured with 6 heads. Mamba uses an SSM state expansion factor of 16 and a block expansion factor of 2. Following Mamba, both minLSTM and minGRU utilize an expansion factor of 2 as well.

C.3 Datasets

Selective Copying. In this task, the model learns to extract data tokens from a sequence while disregarding noise tokens. Following Gu & Dao (2024), we consider a vocabulary of 16 and sequences of length 4096. Each sequence includes 16 randomly placed data tokens. The remainder of the tokens are noise.

Reinforcement Learning. In this setting, we consider continuous control tasks from the D4RL benchmark (Fu et al., 2020). These tasks based on MuJoCo comprise of three environments with dense rewards: HalfCheetah, Hopper, and Walker. For each environment, three different datasets are considered that have varying level represent varying levels of data quality:

- Medium (M): One million timesteps generated by a policy scoring about one-third of an expert policy's score.
- Medium-Replay (M-R): A replay buffer from an agent trained to perform like the Medium policy.

- Medium-Expert (M-E): One million timesteps from the Medium policy combined with one million from an expert policy.

Following Fu et al. (2020), reported scores are normalized such that 100 represents an expert policy performance.

Language Modelling. In this setting, we consider the Shakespeare dataset, comprising a collection of text data derived from the works of William Shakespeare. The training and testing data consists of $1,003,854$ and $111,540$ tokens respectively.