Adjoint sharding for very long context training of state space models

Anonymous authors
Paper under double-blind review

Abstract

Despite fast progress, efficiently training large language models (LLMs) in extremely long contexts remains challenging. Existing methods fall back to training LLMs with short contexts (up to a few thousand tokens) and use inference time techniques when evaluating on very long contexts (above 1M tokens). Training on very long contexts is limited by GPU memory availability and the prohibitively long training times it requires on state-of-the-art hardware. Meanwhile, many real-life applications require training/fine-tuning with long context on specific tasks. Such applications include, for example, augmenting the context with various sources of raw reference information for extraction, summarization, or fact reconciliation tasks. We propose adjoint sharding, a novel technique that comprises sharding gradient calculation during training to reduce memory requirements by orders of magnitude, making training on very long contexts computationally tractable. At the core of our adjoint sharding algorithm lies the adjoint method, which efficiently computes gradients that are provably equivalent to the gradients computed using standard backpropagation. We also propose truncated adjoint sharding to accelerate the algorithm while maintaining performance. We provide a distributed and a parallel-computing version of adjoint sharding to speed up training and to show that adjoint sharding is compatible with these standard memory-reduction techniques. Empirical results show the proposed adjoint sharding algorithm reduces memory usage by up to 3× on a large language model with 1.27B parameters on 1M context length training. This reduction in memory usage allows increasing the maximum context length of training a 1.27B parameter model from 35K tokens to above 100K tokens on a training infrastructure composed of five AWS P4 instances.

1 Introduction

Foundation models are a new paradigm in artificial intelligence research focused on building large, general-purpose models that adapt to different tasks OpenAI et al. (2024); Meta et al. (2024); Cai et al. (2024); Pióro et al. (2024). Extensive training on large datasets equips foundation models with broad capabilities. We then fine-tune the foundation models on smaller datasets for specific applications. Foundation models commonly employ the transformer architecture Vaswani et al. (2023). Despite the immense success, training transformer-based models requires memory growing quadratically with the context length L, limiting their applications on long context tasks Li et al. (2024). Researchers developed various techniques to conquer this problem, ranging from inference time context window expansion (Ding et al., 2024b;a), IO-aware algorithms (Dao et al., 2022; Dao, 2023; Shah et al., 2024), and various linearly scaling language model architectures (Gu & Dao, 2024b; Dao & Gu, 2024b; Peng et al., 2023a; Beltagy et al., 2020). On another note, distributed learning enables training large models with many GPUs, and efficient training methods like activation checkpointing, model/gradient sharding, and mixed-precision computing have further reduced the memory requirement of training a large model Verbraeken et al. (2020); Zhao et al. (2023); Rajbhandari et al. (2020b); Micikevicius et al. (2018); Herrmann et al. (2019). However, current methodologies are entirely based on backpropagation and compute the gradient as a whole, inevitably requiring a memory growing rapidly with model size and context length (Damadi et al., 2023). Current sharding methods ignore the activations and only consider the model weights and optimizer states, constituting only a minor amount of the total memory

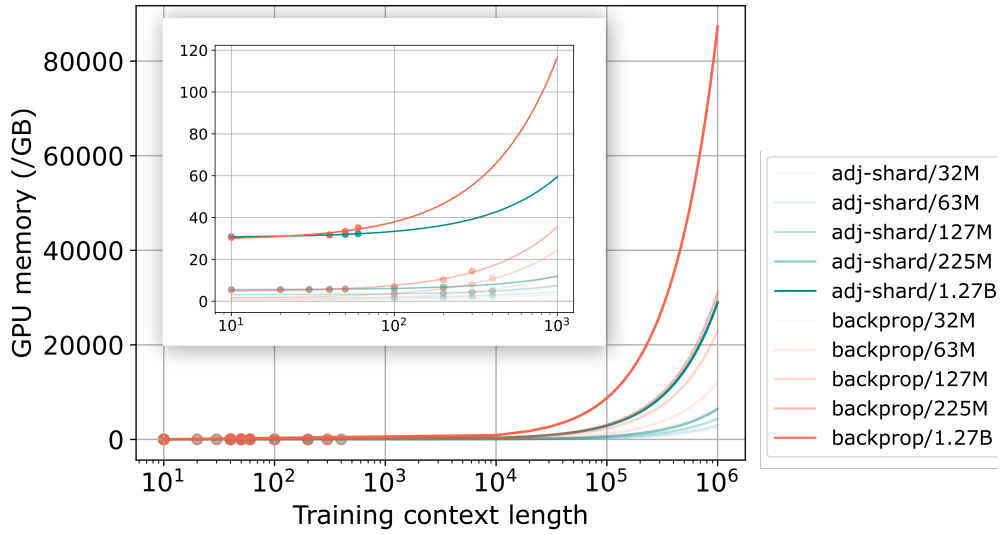

Figure 1: Compared to backpropagation (red lines), adjoint sharding (blue lines) significantly reduces memory requirements at training. Showing memory cost to train 32M, 63M, 127M, 225M, and 1.27B parameter State Space Model (SSM) with batch size 2 and Adam optimizer on one GPU.

cost (Sohoni et al., 2022). Activation checkpointing is among the limited techniques that consider activation values. Activation checkpointing offloads necessary intermediate states to the CPU and recomputes them on the fly, trading compute time for memory reduction (Sohoni et al., 2022; Rajbhandari et al., 2020a). The substantial time required for offloading to the CPU hinders the effectiveness of activation checkpointing. We propose adjoint sharding to disassemble the gradient computation of residual and/or recurrent-based models to achieve orders of magnitude lower memory usage during training.

Adjoint method The adjoint sharding method roots in the adjoint method for recurrent models (Cao et al., 2002; Johnson, 2007). Given an optimization problem of a parametric recurrent forward process, the adjoint method is concerned with the computation of the gradients regarding the process's parameters. Backpropagation saves intermediate states to calculate gradients, whereas the adjoint method relies on a backward adjoint process to compute gradients. The adjoint method is a constant-memory optimization technique for dynamical systems Chen et al. (2019); Xu et al. (2022). In this paper, we are only concerned with the adjoint method for recurrent relations.

Vector-Jacobian product Adjoint sharding disassembles the gradient computation of a large language model (LLM) into independent vector-Jacobian product (VJP) computations. By multiplying the Jacobian on the left with a vector, it becomes unnecessary to compute the expensive Jacobian. Modern VJPs are as fast as a forward function call of the model, and can be thousands of times faster than Jacobian computations Balestriero & Baraniuk (2021). We speed up adjoint sharding by employing the VJPs.

Truncated adjoint sharding Sharding the gradient computation allows us to prioritize the essential gradients and disregard the rest, resulting in faster computation. We term this novel method truncated adjoint sharding, and empirically showcase its performance.

Distributed and parallel computation In addition, we have developed a distributed multi-GPU variant of adjoint sharding to improve the scalability of LLM training further. We also analyze the memory cost of parallel computation of adjoint sharding, opening up directions for massive speedups.

State-space models and residual networks Residual networks (ResNets) are a commonly applied neural network structure. We illustrate adjoint sharding assuming a ResNet structure (He et al., 2015). State-space models (Mamba) have achieved performances on par with attention-based models while possessing a linear scaling regarding the context length L, a polynomial speedup compared to the L^2 scaling of transformers Vaswani et al. (2023); Gu & Dao (2024a).

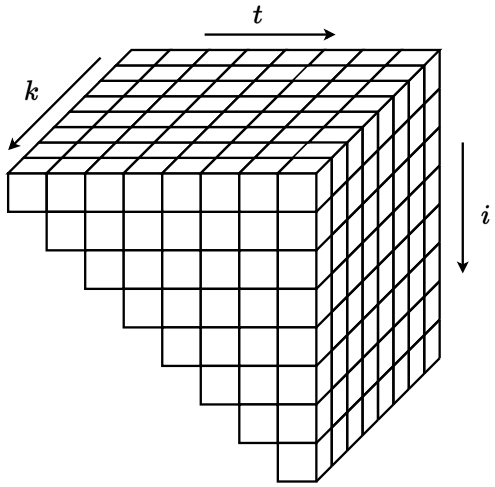

Figure 2: Adjoint sharding disassembles large models' gradient computations along the sequence dimension t and the layer dimension k. When evaluating the gradient at time t, we perform t vector-Jacobian products along the adjoint dimension i for every layer index k.

2 Related works

Linear LLMs (De et al., 2024; Beck et al., 2024; Peng et al., 2023a) proposed LLM architectures with a linear inference time complexity. We form each linear LLM by stacking K residual layers together, where each layer has a recurrent relation. However, their temporal relationships are nonlinear, which limits the application of adjoint sharding to disassemble the gradients into independent vector-Jacobian products.

Backpropagation through time Applying the adjoint method for recurrent models leads to backpropagation through time (BPTT) (Werbos, 1990). BPTT is a training algorithm developed for recurrent neural networks (RNNs). RNN models suffer from the exploding and vanishing gradient because of the $\prod_{j=i+1}^{t} \partial \mathbf{f}(\mathbf{x}^j, \mathbf{h}^{j-1}, \mathbf{W_h})/\partial \mathbf{h}^{j-1}$ term (Pascanu et al., 2013). SSMs provide remedies with careful parameterization of the recurrent dynamics inspired by classical SSM theory (Fu et al., 2023; Gu et al., 2021; 2022; Gupta et al., 2023; Orvieto et al., 2023; Kaul, 2020). Linear temporal relations allow efficient evaluations of the model, while preserving universal approximation capabilities (Wang & Xue, 2023). By a similar token, truncated adjoint sharding can be seen as a more general version of the truncated backpropagation through time (Jaeger, 2005; Tallec & Ollivier, 2017).

Neural ordinary differential equations The adjoint method also applies to the optimization of continuous systems, especially the ordinary differential equations (ODEs) (Chen et al., 2019; Dupont et al., 2019). Optimizing neural ODEs with autograd requires backpropagating through numerical solvers along every step, using an unrealistic amount of memory. The adjoint method does not backpropagate through the operations of the solver and uses a constant amount of memory. However, applying the adjoint method for continuous systems requires solving a costly ODE initial value problem with dimensionality equal to the number of parameters.

Low memory training methods Researchers proposed various low memory training techniques to train big models in very long contexts. ZERO provides data- and model-parallel training while retaining low communication volume, while eliminating memory redundancies (Rajbhandari et al., 2020b). PyTorch FSDP streamlines model, gradient, and data parallelization (Zhao et al., 2023). Activation checkpointing discards intermediate values during the forward step, and recomputes on the fly during the training phase (Sohoni et al., 2022). CPU offloading scales large model training by offloading data and computations to the CPU, trading computing time for memory reduction (Ren et al., 2021). Ring attention leverages the blockwise computation of self-attention and feedforward to distribute long sequences across multiple devices while fully overlapping the communication of key-value blocks with blockwise attention computations, enabling

very-long context training of attention-based methods (Liu et al., 2023; 2024). The proposed adjoint sharding distributes state-space model computations across multiple devices and multiple multi-GPU instances (MIG) to enable very-long context training of state-space models.

Context length extension methods The Existing context length extension method separates into two classes. The first type is fine-tuning free methods, including Positional Interpolation (PI) (Chen et al., 2023), the NTKAware Scale ROPE (NTK) (users, 2023), and StreamingLLM (Xiao et al., 2024). The second type is fine-tuning methods, including LongChat (Li* et al., 2023), LongAlpaca (Chen et al., 2024), YaRN (Peng et al., 2023b), and LongLlama (Chen et al., 2024). Additional methods like activation beacon tune a network separate from the LLM (Zhang et al., 2024). As shown in Figure 3, fine-tuning methods achieve better performances than fine-tuning-free methods at lengths where we perform fine-tuning. However, fine-tuning methods suffer from a high computational cost and require a potentially intractable amount of GPU memory during fine-tuning.

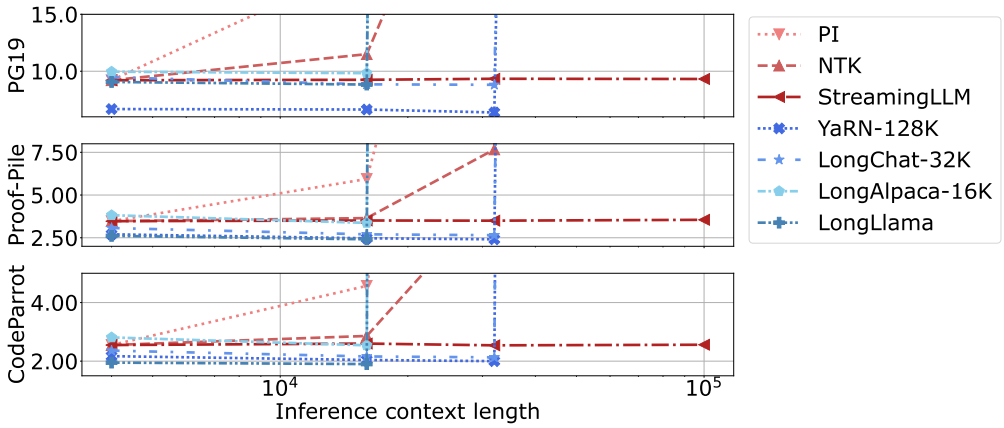

Figure 3: Lines in red and in blue are fine-tuning free and fine-tuning methods. Fine-tuning methods achieve better performances than the fine-tuning-free method but often suffer from out-of-memory issues (Chen et al., 2023; users, 2023; Xiao et al., 2024; Li* et al., 2023; Chen et al., 2024; Peng et al., 2023b; Zhang et al., 2024; Tworkowski et al., 2023). Lower values are better across all three tasks.

3 Background

We first give a concise introduction to the state-space models, the residual networks, and the adjoint method.

3.1 State-space models

While our method generally applies to all recurrent models, we illustrate the idea using state-space models (SSMs), which have shown performances at least on par with transformers at small to medium scale (Dao & Gu, 2024a). Given an input token sequence $\{\mathbf{x}_t\}_{t=1}^{T}$, the SSMs first calculate the corresponding matrices $\mathbf{A}^t$, $\mathbf{B}^t$, and $\mathbf{C}^t$ to evolve the dynamics as follows:

$$\mathbf{A}^t = \boldsymbol{\mathcal{A}}(\mathbf{x}^t);\ \mathbf{B}^t = \boldsymbol{\mathcal{B}}(\mathbf{x}^t);\ \mathbf{C}^t = \boldsymbol{\mathcal{C}}(\mathbf{x}^t).$$

The SSMs evolve a latent dynamics $\mathbf{h}^t$, whose initial condition $\mathbf{h}^0$ is often assumed to be zero. With $\mathbf{h}^0$ and $\mathbf{A}^t$, $\mathbf{B}^t$ defined, the dynamics evolves as:

$$\mathbf{h}^t = \mathbf{A}^t\mathbf{h}^{t-1} + \mathbf{B}^t\mathbf{x}^t.$$

The matrices $\mathbf{C}^t$ then maps the latent dynamics $\mathbf{h}^t$ back to token space as $\mathbf{y}^t = \mathbf{C}^t\mathbf{h}^t$, with $\mathbf{y}^t$ being the predicted token at t. For a sequence of T tokens, we denote:

$$\begin{aligned}
&\mathbf{A} = (\mathbf{A}^1, \mathbf{A}^2, \ldots, \mathbf{A}^T),\ \mathbf{B} = (\mathbf{B}^1, \mathbf{B}^2, \ldots, \mathbf{B}^T),\\
&\mathbf{C} = (\mathbf{C}^1, \mathbf{C}^2, \ldots, \mathbf{C}^T),\ \mathbf{H} = (\mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^T),\\
&\mathbf{X} = (\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^T),\ \mathbf{Y} = (\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^T).
\end{aligned}$$

In the most general case, we have $\mathbf{H} \in \mathbb{R}^{T\times N}, \mathbf{A} \in \mathbb{R}^{T\times N\times N}, \mathbf{B} \in \mathbb{R}^{T\times N\times P}, \mathbf{C} \in \mathbb{R}^{T\times P\times N}, \mathbf{X} \in \mathbb{R}^{T\times P}, \mathbf{Y} \in \mathbb{R}^{T\times P}$, where N is the hidden state dimension, and P is the input/output dimension. We evolve the dynamics for $t = 1, \ldots, T$, and assume that $\mathbf{h}^0$ is a fixed and predefined constant. The input to an SSM is $\mathbf{X}$ and $\mathbf{h}^0$, and the output is $\mathbf{Y}$. We define SSM$(\cdot)$ as performing the following five steps:

1. $\{\mathbf{A}^t\}_{t=1}^T = \{\boldsymbol{\mathcal{A}}(\mathbf{x}^t)\}_{t=1}^T$,
2. $\{\mathbf{B}^t\}_{t=1}^T = \{\boldsymbol{\mathcal{B}}(\mathbf{x}^t)\}_{t=1}^T$,
3. $\{\mathbf{C}^t\}_{t=1}^T = \{\boldsymbol{\mathcal{C}}(\mathbf{x}^t)\}_{t=1}^T$,
4. $\{\mathbf{h}^t\}_{t=1}^T = \{\mathbf{A}^t\mathbf{h}^{t-1} + \mathbf{B}^t\mathbf{x}^t\}_{t=1}^T$;
5. $\{\mathbf{y}^t\}_{t=1}^T = \{\mathbf{C}^t\mathbf{h}^t\}_{t=1}^T$.

The input to the five steps is $\mathbf{X}$, and the output is $\mathbf{Y}$. We can then write SSM$(\mathbf{X}) = \mathbf{Y}$. SSMs reduce the quadratic computational complexity with sequence length on transformers to linear, and the inference-time memory requirements from the key-value cache. SSM-based models at small to medium scales have shown performances on par with or better than transformer-based models. For instance, (Pióro et al., 2024; Anthony et al., 2024) shows that the SSM-based mixture-of-experts (MOE) model outperforms the baseline transformer-based MOE model on model sizes as big as 2400M parameters. (Waleffe et al., 2024) performed an extensive empirical study and found that while SSMs outperform transformers on various tasks, they underperform on tasks that require stellar copying, in-context learning, or long-context reasoning abilities. (Waleffe et al., 2024) also experimented with an SSM-transformer hybrid model, which outperforms transformers and is up to eight times faster when generating tokens at inference time. (Lieber et al., 2024) trained a 52B parameter model and further affirmed the hybrid model's performance.

3.2 Residual Networks

In practice, we have K SSMs stacked together, and we have a large language head (LLH) $\Omega \in \mathbb{R}^{\mathbb{T}\times P}$, where $\mathbb{T}$ is the number of all possible tokens. To predict a token, we have $\mathbf{o}^t = \Omega\hat{\mathbf{y}}_K^t$. Define $(\mathbf{y}_K^1, \ldots, \mathbf{y}_K^T) = \mathbf{Y}_K$, a ResNet computes $\mathbf{Y}_K$ as follows:

$$\begin{aligned}
(\mathbf{y}_K^1, \ldots, \mathbf{y}_K^T) &= \mathbf{Y}_{K-1} + \mathrm{SSM}_K(\hat{\mathbf{Y}}_{K-1})\\
&= \mathbf{Y}_0 + \mathrm{SSM}_1(\hat{\mathbf{Y}}_0) + \cdots + \mathrm{SSM}_K(\hat{\mathbf{Y}}_{K-1})\\
&= \mathbf{Y}_0 + \sum_{k=1}^K \mathrm{SSM}_k(\hat{\mathbf{Y}}_{k-1}) = \mathbf{Y}_0 + \sum_{k=1}^K \tilde{\mathbf{Y}}_k,
\end{aligned}$$

where

$$\hat{\mathbf{Y}}_k = (\hat{\mathbf{y}}_k^1, \ldots, \hat{\mathbf{y}}_k^T) = (\mathrm{Norm}(\mathbf{y}_k^1), \ldots, \mathrm{Norm}(\mathbf{y}_k^T)),$$

and $\mathrm{SSM}_k(\hat{\mathbf{Y}}_{k-1}) = \tilde{\mathbf{Y}}_k$. Therefore, for a latent state at time t we have $\mathbf{y}_K^t = \mathbf{y}_0^t + \sum_{k=1}^K \tilde{\mathbf{y}}_k^t$.

ResNet has been the foundation of numerous modern networks, including the transformers, diffusion models, segmentation models, SSMs, and more (He et al., 2016; Guo et al., 2022; Kirillov et al., 2023; Peebles & Xie, 2023). ResNet's residual structure allows for a separation between gradients of each layer by applying differentiation on summations.

3.3 Adjoint method

The adjoint method is concerned with optimizing $\mathbf{y}(\mathbf{h}(\boldsymbol{\theta}), \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$, where $\mathbf{h}(\boldsymbol{\theta}) \in \mathbb{R}^P$ is the solution to $\mathbf{f}(\mathbf{h}(\boldsymbol{\theta}), \boldsymbol{\theta}) = 0$ (Cao et al., 2002). To employ gradient-based algorithms like the stochastic gradient descent (SGD) or the Adam, we compute the derivative of $\mathbf{y}$ regarding $\boldsymbol{\theta} \in \mathbb{R}^{|\boldsymbol{\theta}|}$:

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}\boldsymbol{\theta}} = \frac{\partial \mathbf{y}}{\partial \boldsymbol{\theta}} + \frac{\partial \mathbf{y}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \boldsymbol{\theta}}, \tag{1}$$

with d being the total derivative, and ∂ being the partial derivative. The adjoint method converts computing $\mathrm{d}\mathbf{y}/\mathrm{d}\boldsymbol{\theta}$ to solving an adjoint equation. In our case, we need the adjoint method for recurrence relations, where $\mathbf{y}$ is given by $\mathbf{y} = \mathbf{y}^t \equiv \mathbf{y}(\mathbf{h}^t(\boldsymbol{\theta}), \boldsymbol{\theta})$, and $\mathbf{h}$ is given by

$$\begin{cases} \mathbf{h}^0 &= \mathbf{b}(\boldsymbol{\theta}), \\ \mathbf{h}^t &= \mathbf{f}(t, \mathbf{h}^{t-1}, \boldsymbol{\theta}). \end{cases} \tag{2}$$

We have

$$\frac{\mathrm{d}\mathbf{f}(t, \mathbf{h}^{t-1}, \boldsymbol{\theta})}{\mathrm{d}\boldsymbol{\theta}} = \frac{\partial \mathbf{f}(t, \mathbf{h}^{t-1}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} + \frac{\partial \mathbf{f}(t, \mathbf{h}^{t-1}, \boldsymbol{\theta})}{\partial \mathbf{h}^{t-1}} \frac{\partial \mathbf{h}^{t-1}}{\partial \boldsymbol{\theta}}. \tag{3}$$

Proposition 3.1. *(Cao et al., 2002) When the states* $\mathbf{h}$ *are defined as Equation 2, the gradient of* $\mathbf{y}$ *with respect to* $\boldsymbol{\theta}$ *is given as:*

$$\begin{cases} \frac{\mathrm{d}\mathbf{y}^t}{\mathrm{d}\boldsymbol{\theta}} &= \frac{\partial \mathbf{y}^t}{\partial \boldsymbol{\theta}} + \boldsymbol{\lambda}^0 \mathbf{b}(\boldsymbol{\theta}) + \sum_{i=1}^t \boldsymbol{\lambda}^i \frac{\partial \mathbf{f}(i, \mathbf{h}^{i-1}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}, \\ \boldsymbol{\lambda}^t &= \partial \mathbf{y}^t / \partial \mathbf{h}^t, \\ \boldsymbol{\lambda}^{i-1} &= \boldsymbol{\lambda}^i \left(\partial \mathbf{f}(i, \mathbf{h}^{i-1}, \boldsymbol{\theta}) / \partial \mathbf{h}^{i-1} \right). \end{cases} \tag{4}$$

Equivalently, we have

$$\boldsymbol{\lambda}^i = (\partial \mathbf{y}^t / \partial \mathbf{h}^t) \left(\prod_{j=t}^{i+1} \left(\partial \mathbf{f}(j, \mathbf{h}^{j-1}, \boldsymbol{\theta}) / \partial \mathbf{h}^{j-1} \right) \right)$$

(Johnson, 2007).

After computing adjoint states $\{\boldsymbol{\lambda}^i\}_{i=0}^t$, the computation of the elements of $\boldsymbol{\lambda}^i(\partial \mathbf{f}(i, \mathbf{h}^{i-1}, \boldsymbol{\theta}) / \partial \boldsymbol{\theta})$ are independent, allowing parallelism. This computation is a vector-Jacobian product (vjp), with $\boldsymbol{\lambda}^i$ as the vector and $\partial \mathbf{f}(i, \mathbf{h}^{i-1}, \boldsymbol{\theta}) / \partial \boldsymbol{\theta}$ as the Jacobian. vjps can be evaluated with the reverse-mode automatic differentiation and initializing the reverse phase with $\boldsymbol{\lambda}^i$ Baydin et al. (2018a). As each vjp only requires saving its corresponding computation graph, and can be disposed of after the computation, we can compute vjps in parallel on modern GPUs. We will discuss this further in subsection 4.5. Adjoint sharding aims to use the adjoint method to replace backpropagation, which solves:

$$\frac{\mathrm{d}\mathbf{y}^t}{\mathrm{d}\boldsymbol{\theta}} = \frac{\partial \mathbf{y}^t}{\partial \boldsymbol{\theta}} + \frac{\partial \mathbf{y}^t}{\partial \mathbf{h}^t} \Bigg\{ \frac{\partial \mathbf{f}(t, \mathbf{h}^{t-1}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} + \frac{\partial \mathbf{f}(t, \mathbf{h}^{t-1}, \boldsymbol{\theta})}{\partial \boldsymbol{h}^{t-1}} \Bigg[\frac{\partial \mathbf{f}(t-1, \mathbf{h}^{t-2}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} + \frac{\partial \mathbf{f}(t-1, \mathbf{h}^{t-2}, \boldsymbol{\theta})}{\partial \boldsymbol{h}^{t-2}} \left(\frac{\partial \mathbf{f}(t-2, \mathbf{h}^{t-3}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} + \dots \right) \Bigg] \Bigg\}.$$

The backpropagation requires a sequential accumulation of the gradients, computing from the outermost layer inwards, therefore needs to save the computation graph for computations at all times t, and creates memory bottlenecks.

4 Adjoint sharding

We now introduce the adjoint sharding technique. We first illustrate the method assuming only one layer of SSM, and generalize to K layers.

4.1 Adjoint sharding for one SSM

Large-scale neural networks usually train with the autograd framework (Baydin et al., 2018b; Paszke et al., 2019). However, this framework suffers from a high memory cost when used with networks of recurrent nature (Baydin et al., 2018b). Although activation checkpointing proves a strong tool, which discards part of the intermediate values and recomputes them later on the fly, the memory cost is still high (Herrmann et al., 2019). We employ the adjoint method for recurrence relations to reduce the memory cost further, and more importantly, to break the temporal dependencies of activations and parallelize their computations.

Define $\theta = \langle \theta_{\boldsymbol{\mathcal{A}}}, \theta_{\boldsymbol{\mathcal{B}}}, \theta_{\boldsymbol{\mathcal{C}}} \rangle$ as $\boldsymbol{\mathcal{A}}$'s, $\boldsymbol{\mathcal{B}}$'s, and $\boldsymbol{\mathcal{C}}$'s parameters, for loss $l^t = l(\mathbf{y}^t)$, in the context of a single-layer SSM, we prove

Proposition 4.1. *The gradient* $\mathrm{d}l^t/\mathrm{d}\boldsymbol{\theta}$ *is given as*

$$\frac{\mathrm{d}l^t}{\mathrm{d}\boldsymbol{\theta}} = \left[\sum_{i=1}^{t} \mathrm{vjp}_{\boldsymbol{\mathcal{A}}^i}\left(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i} \otimes \mathbf{h}^{i-1}\right)\right] \oplus \left[\sum_{i=1}^{t} \mathrm{vjp}_{\boldsymbol{\mathcal{B}}^i}\left(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i} \otimes \hat{\mathbf{x}}^i\right)\right] \oplus \mathrm{vjp}_{\boldsymbol{\mathcal{C}}^t}\left(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t} \otimes \mathbf{h}^t\right), \tag{5}$$

where the adjoint state $\boldsymbol{\lambda}^{t,\tau} = \mathbf{C}^t(\prod_{i=1}^{t-\tau} \mathbf{A}^{t+1-i})$, $\mathrm{vjp}_{\mathrm{Net}^i}(v) = v \cdot \mathrm{Net}_{\boldsymbol{\theta}}(\mathrm{Input}^i)$, *with* $\boldsymbol{\theta}$ *being* Net*'s parameters and* i *being the index of* Input, $\otimes$ *is the vector outer product, and* $\oplus$ *is vector concatenation.*

The proof of proposition 4.1 is in section A.1. The gradient for parameters of $\boldsymbol{\mathcal{A}}$, and $\boldsymbol{\mathcal{B}}$ are each separated into $\{\mathrm{vjp}_{\boldsymbol{\mathcal{A}}^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i} \otimes \mathbf{h}^{i-1})\}_{i=1}^t$, $\{\mathrm{vjp}_{\boldsymbol{\mathcal{B}}^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i} \otimes \hat{\mathbf{x}}^i\}_{i=1}^t$, and the gradient for parameters of $\boldsymbol{\mathcal{C}}$ only depend on inputs at time t. After computing the adjoint states, these vjp computations are separated on both the network and the temporal level.

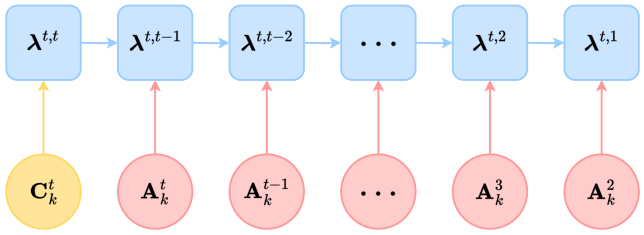

Figure 4: The adjoint states are computed sequentially backwards.

4.2 Adjoint sharding for multiple SSMs

We now generalize the results from subsection 4.1 to the general case of K SSMs concatenated. As introduced in subsection 3.2, the outputs of each SSM layer are added to the results of the last layer and normalized before feeding into the next layer. Define the loss over all token predictions $L = \sum_{t=1}^{T} l^t$, using the residual structure we have

$$\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} &= \sum_{t=1}^{T} \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \frac{\mathrm{d}\mathbf{y}_K^t}{\mathrm{d}\boldsymbol{\theta}} \\
&= \sum_{t=1}^{T} \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \frac{\mathrm{d}(\mathbf{y}_0^t + \sum_{k=1}^{K} \tilde{\mathbf{y}}_k^t)}{\mathrm{d}\boldsymbol{\theta}} \\
&= \sum_{t=1}^{T} \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \sum_{k=1}^{K} \frac{\mathrm{d}\tilde{\mathbf{y}}_k^t}{\mathrm{d}\boldsymbol{\theta}}.
\end{aligned}$$

Combining with proposition 4.1, we have

Proposition 4.2. *The gradient of the total loss L with respect to the* SSM *parameters $\boldsymbol{\theta}$ is given as*

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = &\left(\sum_{t=1}^{T}\sum_{k=1}^{K}\sum_{i=1}^{t}\mathrm{vjp}_{\boldsymbol{\mathcal{A}}_k^i}\left(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i}\otimes\mathbf{h}_k^{i-1}\right)\right)\\ &\oplus\left(\sum_{t=1}^{T}\sum_{k=1}^{K}\sum_{i=1}^{t}\mathrm{vjp}_{\boldsymbol{\mathcal{B}}_k^i}\left(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i}\otimes\hat{\mathbf{y}}_{k-1}^i\right)\right)\\ &\oplus\left(\sum_{t=1}^{T}\sum_{k=1}^{K}\mathrm{vjp}_{\boldsymbol{\mathcal{C}}_k^t}\left(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\otimes\mathbf{h}_k^t\right)\right),\end{aligned} \tag{6}$$

where the input to $\mathrm{vjp}_{\boldsymbol{\mathcal{C}}_k^t}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\otimes\mathbf{h}_k^t)$, $\mathrm{vjp}_{\boldsymbol{\mathcal{A}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i}\otimes\mathbf{h}_k^{i-1})$, and $\mathrm{vjp}_{\boldsymbol{\mathcal{B}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i}\otimes\hat{\mathbf{y}}_{k-1}^i)$ *are computed with the k-th* SSM *and the* $\hat{\mathbf{y}}_{k-1}^i = \mathrm{Norm}(\mathbf{y}_{k-2}^i + \mathrm{SSM}_{k-1}(\hat{\mathbf{Y}}_{k-2})^i)$ *(the normalized output sequence of the (k-1)-th* SSM*). The adjoint state at layer k is defined as* $\boldsymbol{\lambda}_k^{t,\tau} = \mathbf{C}_k^t(\prod_{i=1}^{t-\tau}\mathbf{A}_k^{t+1-i})$.

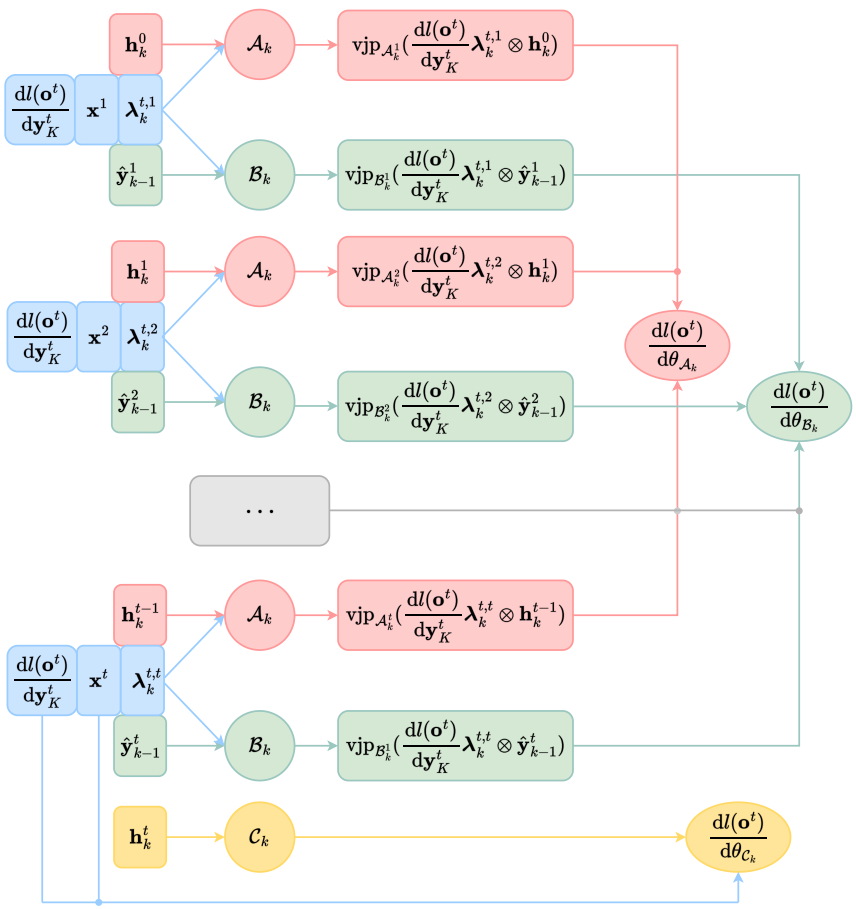

Figure 5: Computation schematic of $\mathrm{d}l^t/\mathrm{d}\boldsymbol{\theta}_{\boldsymbol{\mathcal{A}}_k}$, $\mathrm{d}l^t/\mathrm{d}\boldsymbol{\theta}_{\boldsymbol{\mathcal{B}}_k}$, and $\mathrm{d}l^t/\mathrm{d}\boldsymbol{\theta}_{\boldsymbol{\mathcal{C}}_k}$.

We provide the proof of proposition 4.2 in section A.2. Define $\boldsymbol{\Lambda}_k^t = \{\boldsymbol{\lambda}_k^{t,\tau}\}_{\tau=1}^t$, proposition 4.2 shows that the gradients of each network's parameters computed with each token only correlate through the adjoint states $\{\boldsymbol{\Lambda}_k^t\}_{k,t=1,1}^{K,T}$. We can efficiently compute the adjoint states after a forward pass. We can also compute the adjoint states on the fly in the gradient computation phase, as it only depends on $\mathbf{C}_k^t$ and $\mathbf{A}_k^t$ and has no dependencies on the network Jacobians regarding the network parameters. The adjoint sharding method breaks down the backpropagation computation from layer-wise and token-wise into foundational vjp computations that do not have any dependencies on each other.

We show a schematic of the computations to $\mathrm{d}l^t/\mathrm{d}\boldsymbol{\theta}_{\boldsymbol{\mathcal{A}}_k}$, $\mathrm{d}l^t/\mathrm{d}\boldsymbol{\theta}_{\boldsymbol{\mathcal{B}}_k}$, and $\mathrm{d}l^t/\mathrm{d}\boldsymbol{\theta}_{\boldsymbol{\mathcal{C}}_k}$ in Figure 5 and a schematic for computing the adjoint states in Figure 4.

4.3 Truncated adjoint sharding

One limitation of adjoint sharding is that the number of vjps performed increases polynomially with the number of tokens T. In particular, adjoint sharding computes the vjp for $\boldsymbol{\mathcal{A}_k}$ and $\boldsymbol{\mathcal{B}_k}$ $(1+T)T/2$ times, and for $\boldsymbol{\mathcal{C}_k}$ T times. When training large networks with many layers and a long context length T, applying adjoint sharding becomes computationally expensive. We propose truncated adjoint sharding, with which we argue that we can get similar results by computing a linearly growing number of vjps, and empirically showcase its performance.

Attention mechanisms have suffered from the $\mathcal{O}(T^2)$ complexities arising from the self-attention structure (Vaswani et al., 2023). To enable training with longer context lengths, global-local attention has been proposed, where we divide the contexts into sections, and compute the attention between sections rather than tokens (Yang et al., 2021). (Tallec & Ollivier, 2017) proposed truncated backpropagation through time (T-BPTT) to avoid gradient explosion/vanishing when training with very long contexts by only counting a fixed number of state transitions. Here, inspired by global-local attention and T-BPTT, instead of computing the full gradient given in Equation 11, we propose to train the SSMs to depend on up to $\bar{T}$ states:

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = &\left(\sum_{t=1}^{T}\sum_{k=1}^{K}\mathrm{vjp}_{\boldsymbol{\mathcal{C}}_k^t}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\otimes\mathbf{h}_k^t)\right)\oplus\left[\sum_{t=1}^{\bar{T}}\sum_{k=1}^{K}\sum_{i=1}^{t}\mathrm{vjp}_{\boldsymbol{\mathcal{A}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i}\otimes\mathbf{h}_k^{i-1})\right.\\ &+\sum_{t=\bar{T}+1}^{T}\sum_{k=1}^{K}\sum_{i=t+1-\bar{T}}^{t}\mathrm{vjp}_{\boldsymbol{\mathcal{A}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i}\otimes\mathbf{h}_k^{i-1})\Bigg]\oplus\left[\sum_{t=1}^{\bar{T}}\sum_{k=1}^{K}\sum_{i=1}^{t}\mathrm{vjp}_{\boldsymbol{\mathcal{B}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i}\otimes\hat{\mathbf{y}}_{k-1}^i)\right.\\ &\left.+\sum_{t=\bar{T}+1}^{T}\sum_{k=1}^{K}\sum_{i=t+1-\bar{T}}^{t}\mathrm{vjp}_{\boldsymbol{\mathcal{B}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i}\otimes\hat{\mathbf{y}}_{k-1}^i\right]\end{aligned}\tag{7}$$

As shown in Equation 7 above, we perform the same computations for $t=1,\ldots,\bar{T}$ as before, and only perform the vjps back to the last $\bar{T}$ states for $t>\bar{T}$. With truncated adjoint sharding, we perform $\bar{T}T+\bar{T}(\bar{T}-1)/2$ vjps, which grows linearly with T. We show the number of vjps performed with and without truncated adjoint sharding in Figure 6. When $\bar{T}=2000$, truncated adjoint sharding reduces 64% of the vjps when training with a context length of 10K. The essence of the truncated adjoint sharding method is that we only explicitly count gradients related to the last $\bar{T}$ states. As each state depends on its prior state, states still implicitly depend on all their prior states. We leave the investigation of $\bar{T}$'s impact on performances for future work.

We provide the algorithm for evaluating adjoint states for token index t and ResNet index k with truncated adjoint sharding $\bar{T}$ in algorithm 1, and the algorithm for evaluating the vjps for token index t and ResNet index k with truncated adjoint sharding $\bar{T}$ in algorithm 2.

Algorithm 1 Evaluating adjoint states for token index t and ResNet index k with truncated adjoint sharding $\bar{T}$

1: **Inputs:** t, k, $\bar{T}$, $\mathbf{C}_k^t$, $\{\mathbf{A}_k^i\}_{i=t+2-\bar{T}}^t$
2: Initialize adjoint state $\boldsymbol{\lambda}_k^{t,t}=\mathbf{C}_k^t$
3: Compute: intermediate values:
4: $\boldsymbol{\zeta}^{\bar{T}}=(\mathbf{A}_k^t\mathbf{A}_k^{t-1}\ldots\mathbf{A}_k^{t+2-\bar{T}},$
$\mathbf{A}_k^t\mathbf{A}_k^{t-1}\ldots\mathbf{A}_k^{t+3-\bar{T}},\ldots,\mathbf{A}_k^t\mathbf{A}_k^{t-1},\mathbf{A}_k^t,\mathbb{I})$.
5: Compute: adjoint states
6: $\bar{\boldsymbol{\Lambda}}_k^{\bar{T}}=(\boldsymbol{\lambda}_k^{t,t+1-\bar{T}},\boldsymbol{\lambda}_k^{t,t+2-\bar{T}},\ldots,\boldsymbol{\lambda}_k^{t,t})=\mathbf{C}_k^t\boldsymbol{\zeta}^{\bar{T}}$.
7: **Return:** $\bar{\boldsymbol{\Lambda}}_k^{\bar{T}}$.

Algorithm 2 Evaluating the vjp's for token index t and ResNet index k with truncated adjoint sharding $\bar{T}$

1: **Inputs:** t, k, $\bar{T}$, $\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}$, $\{\mathbf{h}_k^i\}_{i=t-\bar{T}}^t$, $\mathbf{C}_k^t$, $\{\mathbf{y}_{k-1}^i\}_{i=t+1-\bar{T}}^t$, $\{\mathbf{A}_k^i\}_{i=t+2-\bar{T}}^t$
2: **Call alg. 1 to compute** $\{\boldsymbol{\lambda}_k^{t,i}\}_{i=t+1-\bar{T}}^t$
3: **Compute:** $\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t} \otimes \mathbf{h}_k^t$, $\{\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \mathbf{h}_k^{i-1}\}_{i=t+1-\bar{T}}^t$, $\{\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \hat{\mathbf{y}}_{k-1}^i\}_{i=t+1-\bar{T}}^t$
4: **Compute:** $\left(\mathrm{vjp}_{\mathbf{C}_k^t}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t} \otimes \mathbf{h}_k^t), \sum_{i=t+1-\bar{T}}^t \mathrm{vjp}_{\mathbf{A}_k^i}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \mathbf{h}_k^{i-1}), \sum_{i=t+1-\bar{T}}^t \mathrm{vjp}_{\mathbf{B}_k^i}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \hat{\mathbf{y}}_{k-1}^i)\right)$
5: **Return:** $\left(\mathrm{vjp}_{\mathbf{C}_k^t}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t} \otimes \mathbf{h}_k^t), \sum_{i=t+1-\bar{T}}^t \mathrm{vjp}_{\mathbf{A}_k^i}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \mathbf{h}_k^{i-1}), \sum_{i=t+1-\bar{T}}^t \mathrm{vjp}_{\mathbf{B}_k^i}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \hat{\mathbf{y}}_{k-1}^i)\right)$

4.4 Distributed training

We now discuss how to distribute the storage and compute of the adjoint sharding method, assuming that we have Υ GPUs. Given the networks $\{\mathcal{A}_k, \mathcal{B}_k, \mathcal{C}_k\}_{k=1}^K$, initial tokens $\{\hat{\mathbf{y}}_0^t\}_{t=1}^T = \{\mathrm{Norm}(\mathbf{x}^t)\}_{t=1}^T$, and initial conditions $\{\mathbf{h}_k^0\}_{k=1}^K$ (usually set to $\mathbf{0}$), we can call algorithm 3 to get all necessary vectors for computing the gradient with adjoint sharding. As shown in algorithm 2, to compute the vjps' for token index t and ResNet

Algorithm 3 Forward step in evaluation mode on a distributed system

1: **Inputs:** $\{\hat{\mathbf{y}}_0^t\}_{t=1}^T$, $\{\mathbf{h}_k^0\}_{k=1}^K$, $\{\mathcal{A}_k, \mathcal{B}_k, \mathcal{C}_k\}_{k=1}^K$, Ω
2: On devices $\upsilon = 1, \ldots, \Upsilon$, in parallel **do**
3: **for** SSM model index $k = (\upsilon - 1)(K//\Upsilon) + 1, \ldots, \upsilon(K//\Upsilon)$ **do**
4: **for** Time step index $t = 1, \ldots, T$ **do**
5: Compute: $\mathbf{A}_k^t = \mathcal{A}_k(\hat{\mathbf{y}}_{k-1}^t)$; $\mathbf{B}_k^t = \mathcal{B}_k(\hat{\mathbf{y}}_{k-1}^t)$; $\mathbf{C}_k^t = \mathcal{C}_k(\hat{\mathbf{y}}_{k-1}^t)$; $\mathbf{h}_k^t = \mathbf{A}_k^t\mathbf{h}_k^{t-1} + \mathbf{B}_k^t\hat{\mathbf{y}}_{k-1}^t$; $\mathbf{y}_k^t = \mathbf{C}_k^t\mathbf{h}_k^t$.
6: Compute: $\mathbf{y}_k^t = \mathbf{y}_{k-1}^t + \hat{\mathbf{y}}_k^t$.
7: Compute: $\hat{\mathbf{y}}_k^t = \mathrm{Norm}(\mathbf{y}_k^t)$.
8: **end for**
9: **end for**
10: Store: $\{\mathbf{h}_k^t\}_{(t,k)=(1,(\upsilon-1)(K//\Upsilon)+1)}^{T,\upsilon(K//\Upsilon)}$, $\{\mathbf{C}_k^t\}_{(t,k)=(1,(\upsilon-1)(K//\Upsilon)+1)}^{T,\upsilon(K//\Upsilon)}$, $\{\hat{\mathbf{y}}_k^t\}_{(t,k)=(1,(\upsilon-1)(K//\Upsilon))}^{T,\upsilon(K//\Upsilon)-1}$, $\{\mathbf{A}_k^t\}_{(t,k)=(2,(\upsilon-1)(K//\Upsilon)+1)}^{T,\upsilon(K//\Upsilon)}$ on device υ.
11: Pass: $\{\mathbf{y}_{\upsilon(K//\Upsilon)-1}^t\}_{t=1}^T$, $\{\hat{\mathbf{y}}_{\upsilon(K//\Upsilon)-1}^t\}_{t=1}^T$ to device $\upsilon + 1$
12: **for** Time step index $t = 1, \ldots, T$ **do**
13: Compute: $\{\mathbf{o}^t = \Omega\mathbf{y}_K^t\}_{t=1}^T$, $\{l(\mathbf{o}^t)\}$, $\{\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\}_{t=1}^T$.
14: **end for**
15: Store: $\{\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\}_{t=1}^T$ on all Υ devices.

index k, we only need $t, k, \mathrm{d}l(\mathbf{o}^t)/\mathrm{d}\mathbf{y}_K^t, \{\mathbf{h}_k^i\}_{i=0}^t, \mathbf{C}_k^t, \{\hat{\mathbf{y}}_{k-1}^i\}_{i=1}^t, \{\mathbf{A}_k^i\}_{i=2}^t$. To compute all the gradients for layer k, we only need $\mathbf{A}$, $\mathbf{h}$, and $\mathbf{C}$ from the k-th layer, and $\hat{\mathbf{y}}$ from the $k-1$-th layer. Therefore, we can divide the K layers into Υ pieces, as shown in the appendix B. As the computations are fully independent and we compute the gradients using only data on local devices, we additionally distribute the model and the gradients, as shown in Table 6, where $\boldsymbol{\theta}_k$ represents the parameters of $\mathcal{A}_k$, $\mathcal{B}_k$, and $\mathcal{C}_k$, and $\mathrm{Gradient}_k$ represents the optimizer states for $\boldsymbol{\theta}_k$. The complete training streamline is provided in algorithm 4. We distribute the activations, computations, gradients, and optimization states across Υ devices. While the forward evaluation pass results across different devices, as shown in algorithm 3, the computation of gradients is parallel across the Υ devices. This parallelization will speed up the training as the gradient computation takes most of the computation budget. We will also get a memory per GPU close to Mem/Υ, with Mem being the memory cost if we only have a single GPU. If we have $\Upsilon > K$ devices, we can further speed up the forward evaluation by first evaluating $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ in parallel, and then adding them together on the distributed devices.

Algorithm 4 Evaluating $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}}$ with truncated adjoint sharding $\bar{T}$ on Υ devices

Inputs: $\{\mathbf{y}_0^t\}_{t=1}^T$, $\{\mathbf{h}_k^0\}_{k=1}^K$, $\{\mathcal{A}_k, \mathcal{B}_k, \mathcal{C}_k\}_{k=1}^K$, Ω, $\bar{T}$, Υ
Call alg. 3 for $\{\mathbf{A}_k^t, \mathbf{C}_k^t, \mathbf{h}_k^t, \hat{\mathbf{y}}_k^t\}_{(t,k)=(1,1)}^{(T,K)}, \{\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\}_{t=1}^T$ and saved on each GPU device.
On each device υ, in parallel **do**
Initialize gradient $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}}$
for Time step index $t = 1, \ldots, \bar{T}$, layer index $k = (\upsilon - 1)(K//\Upsilon) + 1, \ldots, \upsilon(K//\Upsilon)$ **do**
  Call alg. 2 for $\Xi = \left[\mathrm{vjp}_{\mathbf{C}_k^t}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t} \otimes \mathbf{h}_k^t), \sum_{i=1}^t \mathrm{vjp}_{\mathbf{A}_k^i}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \mathbf{h}_k^{i-1}), \sum_{i=1}^t \mathrm{vjp}_{\mathbf{B}_k^i}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \hat{\mathbf{y}}_{k-1}^i)\right]$
  Compute: $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} += \Xi$
end for
for Time step index $t = \bar{T} + 1, \ldots, T$, layer index $k = (\upsilon - 1)(K//\Upsilon) + 1, \ldots, \upsilon(K//\Upsilon)$ **do**
  Call alg. 2 for $\Xi = \left(\mathrm{vjp}_{\mathbf{C}_k^t}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t} \otimes \mathbf{h}_k^t), \sum_{i=t+1-\bar{T}}^t \mathrm{vjp}_{\mathbf{A}_k^i}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \mathbf{h}_k^{i-1}), \sum_{i=t+1-\bar{T}}^t \mathrm{vjp}_{\mathbf{B}_k^i}(\frac{\mathrm{d}l(\mathbf{o}^t)}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \hat{\mathbf{y}}_{k-1}^i)\right)$
  Compute: $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} += \Xi$
end for
Return: $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}}$

4.5 Parallel computing

Adjoint sharding converts the sequential process of backpropagation gradient computation into individual independent vjps, allowing for parallel computation. We analyze the time and memory cost of $\mathrm{vjp}_{\boldsymbol{\mathcal{A}}_k^i}((\mathrm{d}l^t/\mathrm{d}\mathbf{y}_K^t)\boldsymbol{\lambda}_k^{t,i} \otimes \mathbf{h}_k^{i-1})$, $\mathrm{vjp}_{\boldsymbol{\mathcal{B}}_k^i}((\mathrm{d}l^t/\mathrm{d}\mathbf{y}_K^t)\boldsymbol{\lambda}_k^{t,i} \otimes \hat{\mathbf{y}}_{k-1}^i)$, and $\mathrm{vjp}_{\boldsymbol{\mathcal{C}}_k^t}((\mathrm{d}l^t/\mathrm{d}\mathbf{y}_K^t) \otimes \mathbf{h}_k^t)$. vjp has a similar time complexity as a forward pass, and a memory complexity of $\mathrm{bs}(|\boldsymbol{\theta}| + \mathbb{O}) + |\boldsymbol{\theta}|$, where bs is the batch size, $\mathbb{O}$ is the number of elements in the network output, and $|\boldsymbol{\theta}|$ is the number of parameters (Novak et al., 2022). We provide the memory and FLOPs required to compute the vjps in Table 1 (NVIDIA, 2024).

We analyze training with a dataset containing contexts of lengths T, with Υ NVIDIA H100 GPUs, and performing computations in FP16. We use a selective diagonal SSM with K layers, and each $\boldsymbol{\mathcal{A}}_k$, $\boldsymbol{\mathcal{B}}_k$, and $\boldsymbol{\mathcal{C}}_k$ network is a single-layer multi-layer perceptron (MLP). For each data point $\{\mathbf{x}^t\}_{t=1}^T$, we store $\{\mathbf{A}_k^t, \mathbf{C}_k^t, \mathbf{h}_k^t, \mathbf{y}_k^t\}_{(t,k)=(1,1)}^{(T,K)}$ and $\{\mathrm{d}l(\mathbf{o}^t)/\mathrm{d}\mathbf{y}_K^t\}_{t=1}^T$, which is $TK(2N+P) + TP$ FP16 numbers. We also save $\boldsymbol{\theta}_{\boldsymbol{\mathcal{A}}}$, $\boldsymbol{\theta}_{\boldsymbol{\mathcal{B}}}$, and $\boldsymbol{\theta}_{\boldsymbol{\mathcal{C}}}$, each taking $PN + N$ FP16 numbers. We need to store $T(2NK + PK + P) + 3N(P+1)$ FP16 numbers before computing the vjp. As computing all adjoint state sequences takes up to $N(2P+1)(1+T)T/2$

		$\mathrm{vjp}_{\boldsymbol{\mathcal{A}}}$	$\mathrm{vjp}_{\boldsymbol{\mathcal{B}}}$	$\mathrm{vjp}_{\boldsymbol{\mathcal{C}}}$
Unstructured SSM	Memory	$\mathrm{bs}(N^2 + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{A}}}\rvert^*) + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{A}}}\rvert$	$\mathrm{bs}(NP + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{B}}}\rvert^*) + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{B}}}\rvert$	$\mathrm{bs}(NP + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{C}}}\rvert^*) + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{C}}}\rvert$
	FLOPs	$\mathrm{bs}(N^2(2P+1))$	$\mathrm{bs}(NP(2P+1))$	$\mathrm{bs}(NP \times (2P+1))$
Diagonal SSM	Memory	$\mathrm{bs}(N + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{A}}}\rvert^*) + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{A}}}\rvert$	$\mathrm{bs}(N + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{B}}}\rvert^*) + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{B}}}\rvert$	$\mathrm{bs}(N + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{C}}}\rvert^*) + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{C}}}\rvert$
	FLOPs	$\mathrm{bs}(N(2P+1))$	$\mathrm{bs}(N(2P+1))$	$\mathrm{bs}(N(2P+1))$
Scalar SSM	Memory	$\mathrm{bs}(1 + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{A}}}\rvert^*) + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{A}}}\rvert$	$\mathrm{bs}(N + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{B}}}\rvert^*) + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{B}}}\rvert$	$\mathrm{bs}(N + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{C}}}\rvert^*) + \lvert\boldsymbol{\theta}_{\boldsymbol{\mathcal{C}}}\rvert$
	FLOPs	$\mathrm{bs}(2P+1)$	$\mathrm{bs}((N(2P+1))$	$\mathrm{bs}(N(2P+1))$

Table 1: Memory and FLOPs required to compute the vjps. $|\boldsymbol{\theta}_{\boldsymbol{\mathcal{A}}}|^*$, $|\boldsymbol{\theta}_{\boldsymbol{\mathcal{B}}}|^*$, and $|\boldsymbol{\theta}_{\boldsymbol{\mathcal{C}}}|^*$ represents the number of elements of the biggest parameter vector of $\boldsymbol{\mathcal{A}}$, $\boldsymbol{\mathcal{B}}$, and $\boldsymbol{\mathcal{C}}$.

FLOPs, it takes $NP(1+T)/T$ FLOPs on average for each adjoint state. For T large enough, $(1+T)/T \approx 1$, we approximate the average FLOPs for each adjoint state with NP. Each vjp then takes $\mathrm{bs}(7NP + 3N)$ FLOPs of computation.

When computing with a selective diagonal SSM with $P = 128$, $N = 225$, and $\mathrm{bs} = 8$, while storing and performing computations in FP16, computing $\mathrm{vjp}_{\boldsymbol{\mathcal{A}}}$, $\mathrm{vjp}_{\boldsymbol{\mathcal{B}}}$, and $\mathrm{vjp}_{\boldsymbol{\mathcal{C}}}$ each takes around 0.6MB memory and 1798144 FLOPs. We characterize the capacity of a modern GPU with FLOPs/sec, which measures the computation speed; GPU memory bandwidth, which is the rate at which a GPU can move data between its memory and processing cores; GPU Memory, which is the amount of data a GPU can hold; and number of Multi-Instance GPU (MIG) instances, which is the number of fully isolated GPU instances with its own high-bandwidth memory, cache, and compute cores a GPU can host. An NVIDIA H100 Tensor Core GPU

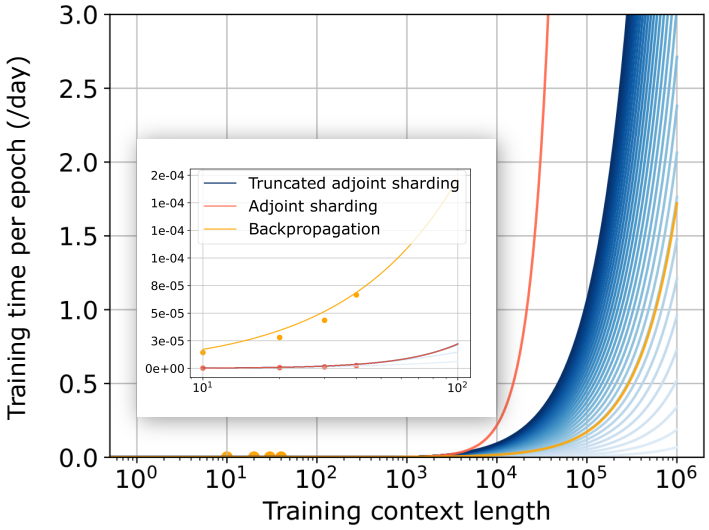

Figure 6: Training time (/day) per epoch comparison for adjoint sharding, truncated adjoint sharding, and backpropagation with different context lengths. Assumed a 100-layer SSM-ResNet model, a 280x acceleration for adjoint sharding from parallel computing (achievable with five Amazon P4 instances), and $\bar{T}$ from 15 to 2500.

has a GPU memory bandwidth 3.35TB/s and performs $1,979$ tera FP16 FLOPS per second. Therefore, the memory bandwidth allows computing (3.35TB/s)/0.6MB $= 5.58 \times 10$E6 batches of vjps per second, and the computing speed allows computing (1979tera/s)/1798144 $= 3.76 \times 1.1$E9 batches of vjps per second. At the same time, since the H100 GPU has 80GB memory, it can hold up to 80GB/(0.6MB/vjp) $= 133$ batches of vjps at the same time if we do not consider any memory overhead. As each H100 GPU can hold up to 7 instances in parallel, we perform the adjoint sharding algorithm with 7Υ instances, offering as much as a 56x speedup on one AWS P4 instance (8 H100 GPUs). We can not achieve such speedup for backpropagation because of its sequential nature.

Limitation The adjoint sharding method provides an alternative method of computing gradients to backpropagation. While we analytically proved that the gradients computed from adjoint sharding are equal to those from backpropagation, adjoint sharding suffers from a polynomial time complexity regarding the training context length when computing equivalent gradients. We provided the truncated adjoint sharding as a linear time complexity alternative, and leave the analysis of its convergence and further improvements on it for future work. We also provided a distributed and parallel computing algorithm for performing adjoint sharding. However, the overhead of naïve implementations of such an algorithm with multi-threading or multiprocessing overweights the speedups when the training context length is small. We leave the efficient implementation of the parallel algorithm on a CUDA kernel for future work.

Conclusion We introduced adjoint sharding, a distributed and parallel computing algorithm, to facilitate the training of LLMs on very long contexts. Unlike the sequential backpropagation, the adjoint sharding computes gradients of each LLM layer against each token independently through vector-Jacobian product, allowing for parallel computation. We propose truncated adjoint sharding to focus on essential gradients to avoid the limitation of vjps increasing polynomially regarding context length. We analyzed the memory and FLOP cost of each computation block in adjoint sharding and proposed a method to accelerate it through parallel computing. Empirical results suggest orders of magnitude of memory reduction in training while maintaining the same training results as backpropagation.

References

Quentin Anthony, Yury Tokpanov, Paolo Glorioso, and Beren Millidge. Blackmamba: Mixture of experts for state-space models, 2024. URL https://arxiv.org/abs/2402.01771.

Randall Balestriero and Richard Baraniuk. Fast jacobian-vector product for deep networks, 2021. URL https://arxiv.org/abs/2104.00219.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey, 2018a. URL https://arxiv.org/abs/1502.05767.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey, 2018b. URL https://arxiv.org/abs/1502.05767.

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, 2024. URL https://arxiv.org/abs/2405.04517.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020. URL https://arxiv.org/abs/2004.05150.

Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, et al. Internlm2 technical report, 2024. URL https://arxiv.org/abs/2403.17297.

Yang Cao, Shengtai Li, and Linda Petzold. Adjoint sensitivity analysis for differential-algebraic equations: algorithms and software. *Journal of Computational and Applied Mathematics*, 149(1):171–191, 2002. ISSN 0377-0427. doi: https://doi.org/10.1016/S0377-0427(02)00528-9. URL https://www.sciencedirect.com/science/article/pii/S0377042702005289. Scientific and Engineering Computations for the 21st Century - Me thodologies and Applications Proceedings of the 15th Toyota Conference.

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019. URL https://arxiv.org/abs/1806.07366.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation, 2023. URL https://arxiv.org/abs/2306.15595.

Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. Longlora: Efficient fine-tuning of long-context large language models, 2024. URL https://arxiv.org/abs/2309.12307.

Saeed Damadi, Golnaz Moharrer, and Mostafa Cham. The backpropagation algorithm for a math student, 2023. URL https://arxiv.org/abs/2301.09977.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024a. URL https://arxiv.org/abs/2405.21060.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024b. URL https://arxiv.org/abs/2405.21060.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/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. Griffin: Mixing gated linear recurrences with local attention for efficient language models, 2024. URL https://arxiv.org/abs/2402.19427.

Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024a.

Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens, 2024b. URL https://arxiv.org/abs/2402.13753.

Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural odes, 2019. URL https://arxiv.org/abs/1904.01681.

Daniel Y. Fu, Tri Dao, Khaled K. Saab, Armin W. Thomas, Atri Rudra, and Christopher Ré. Hungry hungry hippos: Towards language modeling with state space models, 2023. URL https://arxiv.org/abs/2212.14052.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024a. URL https://arxiv.org/abs/2312.00752.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024b. URL https://arxiv.org/abs/2312.00752.

Albert Gu, Isys Johnson, Karan Goel, Khaled Saab, Tri Dao, Atri Rudra, and Christopher Ré. Combining recurrent, convolutional, and continuous-time models with linear state-space layers, 2021. URL https://arxiv.org/abs/2110.13985.

Albert Gu, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Ré. How to train your hippo: State space models with generalized orthogonal basis projections, 2022. URL https://arxiv.org/abs/2206.12037.

Meng-Hao Guo, Tian-Xing Xu, Jiang-Jiang Liu, Zheng-Ning Liu, Peng-Tao Jiang, Tai-Jiang Mu, Song-Hai Zhang, Ralph R Martin, Ming-Ming Cheng, and Shi-Min Hu. Attention mechanisms in computer vision: A survey. *Computational visual media*, 8(3):331–368, 2022.

Ankit Gupta, Harsh Mehta, and Jonathan Berant. Simplifying and understanding state space models with diagonal linear rnns, 2023. URL https://arxiv.org/abs/2212.00768.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

Julien Herrmann, Olivier Beaumont, Lionel Eyraud-Dubois, Julien Hermann, Alexis Joly, and Alena Shilova. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory, 2019. URL https://arxiv.org/abs/1911.13214.

Herbert Jaeger. A tutorial on training recurrent neural networks , covering bppt , rtrl , ekf and the " echo state network " approach - semantic scholar. In *National Research Center for Information Technology, 2002*, 2005. URL https://api.semanticscholar.org/CorpusID:192593367.

Steven Johnson. Adjoint methods and sensitivity analysis for recurrence, 01 2007.

Shiva Kaul. Linear dynamical systems as a core computational primitive. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 16808–16820. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/c3581d2150ff68f3b33b22634b8adaea-Paper.pdf.

Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4015–4026, 2023.

Dacheng Li*, Rulin Shao*, Anze Xie, Ying Sheng, Lianmin Zheng, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. How long can open-source llms truly promise on context length?, June 2023. URL https://lmsys.org/blog/2023-06-29-longchat.

Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. Long-context llms struggle with long in-context learning, 2024. URL https://arxiv.org/abs/2404.02060.

Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A hybrid transformer-mamba language model, 2024. URL https://arxiv.org/abs/2403.19887.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023. URL https://arxiv.org/abs/2310.01889.

Hao Liu, Wilson Yan, Matei Zaharia, and Pieter Abbeel. World model on million-length video and language with blockwise ringattention, 2024. URL https://arxiv.org/abs/2402.08268.

Meta et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018. URL https://arxiv.org/abs/1710.03740.

Roman Novak, Jascha Sohl-Dickstein, and Samuel S. Schoenholz. Fast finite width neural tangent kernel, 2022. URL https://arxiv.org/abs/2206.08720.

NVIDIA. Matrix multiplication background user's guide, 2024. URL https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

OpenAI et al. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences, 2023. URL https://arxiv.org/abs/2303.06349.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013. URL https://arxiv.org/abs/1211.5063.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4195–4205, 2023.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Jiaju Lin, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Bolun Wang, Johan S. Wind, Stanislaw Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. Rwkv: Reinventing rnns for the transformer era, 2023a. URL https://arxiv.org/abs/2305.13048.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models, 2023b. URL https://arxiv.org/abs/2309.00071.

Maciej Pióro, Kamil Ciebiera, Krystian Król, Jan Ludziejewski, Michał Krutul, Jakub Krajewski, Szymon Antoniak, Piotr Miłoś, Marek Cygan, and Sebastian Jaszczur. Moe-mamba: Efficient selective state space models with mixture of experts, 2024. URL https://arxiv.org/abs/2401.04081.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020a. URL https://arxiv.org/abs/1910.02054.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020b. URL https://arxiv.org/abs/1910.02054.

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training, 2021. URL https://arxiv.org/abs/2101.06840.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://arxiv.org/abs/2407.08608.

Nimit S. Sohoni, Christopher R. Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report, 2022. URL https://arxiv.org/abs/1904.10631.

Corentin Tallec and Yann Ollivier. Unbiasing truncated backpropagation through time, 2017. URL https://arxiv.org/abs/1705.08209.

Szymon Tworkowski, Konrad Staniszewski, Mikołaj Pacek, Yuhuai Wu, Henryk Michalewski, and Piotr Miłoś. Focused transformer: Contrastive training for context scaling, 2023. URL https://arxiv.org/abs/2307.03170.

Reddit users. Ntk-aware scaled rope, 2023. URL https://www.reddit.com/r/LocalLLaMA/comments/14lz7j5/ntkaware_scaled_rope_allows_llama_models_to_have/.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning. *ACM Computing Surveys*, 53(2):1–33, March 2020. ISSN 1557-7341. doi: 10.1145/3377454. URL http://dx.doi.org/10.1145/3377454.

Roger Waleffe, Wonmin Byeon, Duncan Riach, Brandon Norick, Vijay Korthikanti, Tri Dao, Albert Gu, Ali Hatamizadeh, Sudhakar Singh, Deepak Narayanan, Garvit Kulshreshtha, Vartika Singh, Jared Casper, Jan Kautz, Mohammad Shoeybi, and Bryan Catanzaro. An empirical study of mamba-based language models, 2024. URL https://arxiv.org/abs/2406.07887.

Shida Wang and Beichen Xue. State-space models with layer-wise nonlinearity are universal approximators with exponential decaying memory, 2023. URL https://arxiv.org/abs/2309.13414.

P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10): 1550–1560, 1990. doi: 10.1109/5.58337.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024. URL https://arxiv.org/abs/2309.17453.

Xingzi Xu, Ali Hasan, Khalil Elkhalil, Jie Ding, and Vahid Tarokh. Characteristic neural ordinary differential equations, 2022. URL https://arxiv.org/abs/2111.13207.

Jianwei Yang, Chunyuan Li, Pengchuan Zhang, Xiyang Dai, Bin Xiao, Lu Yuan, and Jianfeng Gao. Focal self-attention for local-global interactions in vision transformers, 2021. URL https://arxiv.org/abs/2107.00641.

Peitian Zhang, Zheng Liu, Shitao Xiao, Ninglu Shao, Qiwei Ye, and Zhicheng Dou. Soaring from 4k to 400k: Extending llm's context with activation beacon, 2024. URL https://arxiv.org/abs/2401.03462.

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. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023. URL https://arxiv.org/abs/2304.11277.

A Proof

A.1 Proof for proposition 4.1

Proof. Define $\partial\tilde{\mathbf{y}}/\partial\mathbf{h}^t = \tilde{\mathbf{y}}^t_{\mathbf{h}^t}$, $\partial\tilde{\mathbf{h}}^t/\partial\mathbf{h}^{t-1} = \tilde{\mathbf{h}}^t_{\mathbf{h}^{t-1}}$, and $\partial\tilde{\mathbf{y}}/\partial\boldsymbol{\theta} = \tilde{\mathbf{y}}^t_{\boldsymbol{\theta}}$, $\partial\tilde{\mathbf{h}}^t/\partial\boldsymbol{\theta} = \tilde{\mathbf{h}}^t_{\boldsymbol{\theta}}$, by plugging in the expression for $\tilde{\mathbf{y}}^t$ from subsection 3.2, proposition 3.1 states that

$$\frac{\mathrm{d}\tilde{\mathbf{y}}^t}{\mathrm{d}\boldsymbol{\theta}} = \tilde{\mathbf{y}}^t_{\mathbf{h}^t}\left[(\prod_{i=1}^{t-1}\mathbf{h}^{t-i+1}_{\mathbf{h}^{t-i}})\mathbf{h}^1_{\boldsymbol{\theta}} + (\prod_{i=1}^{t-2}\mathbf{h}^{t-i+1}_{\mathbf{h}^{t-i}})\mathbf{h}^2_{\boldsymbol{\theta}} + \cdots + \mathbf{h}^t_{\mathbf{h}^{t-1}}\mathbf{h}^{t-1}_{\boldsymbol{\theta}} + \mathbf{h}^t_{\boldsymbol{\theta}}\right] + \tilde{\mathbf{y}}^t_{\boldsymbol{\theta}}.$$

In the context of SSM, we have:

$$\mathbf{h}^t = \mathbf{A}^t\mathbf{h}^{t-1} + \mathbf{B}^t\hat{\mathbf{x}}^t, \mathbf{h}^t_{\mathbf{h}^{t-1}} = \mathbf{A}^t, \mathbf{h}^t_{\boldsymbol{\theta}} = \mathbf{A}^t_{\boldsymbol{\theta}}\mathbf{h}^{t-1} + \mathbf{B}^t_{\boldsymbol{\theta}}\hat{\mathbf{x}}^t, \tilde{\mathbf{y}}^t = \mathbf{C}^t\mathbf{h}^t, \tilde{\mathbf{y}}^t_{\mathbf{h}^t} = \mathbf{C}^t, \tilde{\mathbf{y}}^t_{\boldsymbol{\theta}} = \mathbf{C}^t_{\boldsymbol{\theta}}\mathbf{h}^t. \tag{8}$$

Plugging in these relations, we get:

$$\frac{\mathrm{d}\tilde{\mathbf{y}}^t}{\mathrm{d}\boldsymbol{\theta}} = \mathbf{C}^t\left[(\prod_{i=1}^{t-1}\mathbf{A}^{t+1-i})\mathbf{h}^1_{\boldsymbol{\theta}} + (\prod_{i=1}^{t-2}\mathbf{A}^{t+1-i})\mathbf{h}^2_{\boldsymbol{\theta}} + \cdots + (\prod_{i=1}^{2}\mathbf{A}^{t+1-i})\mathbf{h}^{t-2}_{\boldsymbol{\theta}} + \mathbf{A}^t\mathbf{h}^{t-1}_{\boldsymbol{\theta}} + \mathbf{h}^t_{\boldsymbol{\theta}}\right] + \tilde{\mathbf{y}}^t_{\boldsymbol{\theta}}. \tag{9}$$

Define the adjoint state $\boldsymbol{\lambda}^{t,\tau} = \mathbf{C}^t(\prod_{i=1}^{t-\tau}\mathbf{A}^{t+1-i})$, we have

$$\frac{\mathrm{d}\tilde{\mathbf{y}}^t}{\mathrm{d}\boldsymbol{\theta}} = \boldsymbol{\lambda}^{t,1}\mathbf{h}^1_{\boldsymbol{\theta}} + \boldsymbol{\lambda}^{t,2}\mathbf{h}^2_{\boldsymbol{\theta}} + \cdots + \boldsymbol{\lambda}^{t,t-1}\mathbf{h}^{t-1}_{\boldsymbol{\theta}} + \boldsymbol{\lambda}^{t,t}\mathbf{h}^t_{\boldsymbol{\theta}} + \tilde{\mathbf{y}}^t_{\boldsymbol{\theta}}$$

Therefore, we have

$$\begin{aligned}\frac{\mathrm{d}l^t}{\mathrm{d}\boldsymbol{\theta}} &= \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\frac{\mathrm{d}(\tilde{\mathbf{y}}^t + \hat{\mathbf{x}}^t)}{\mathrm{d}\boldsymbol{\theta}} \\ &= \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\frac{\mathrm{d}\tilde{\mathbf{y}}^t}{\mathrm{d}\boldsymbol{\theta}} \\ &= \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}[\boldsymbol{\lambda}^{t,1}\mathbf{h}^1_{\boldsymbol{\theta}} + \boldsymbol{\lambda}^{t,2}\mathbf{h}^2_{\boldsymbol{\theta}} + \cdots + \boldsymbol{\lambda}^{t,t-1}\mathbf{h}^{t-1}_{\boldsymbol{\theta}} + \boldsymbol{\lambda}^{t,t}\mathbf{h}^t_{\boldsymbol{\theta}} + \tilde{\mathbf{y}}^t_{\boldsymbol{\theta}}]\end{aligned}$$

Plug in everything, we have

$$\begin{aligned}\frac{\mathrm{d}l^t}{\mathrm{d}\boldsymbol{\theta}} &= \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}[\boldsymbol{\lambda}^{t,1}(\mathbf{A}^1_{\boldsymbol{\theta}}\mathbf{h}^0 + \mathbf{B}^1_{\boldsymbol{\theta}}\hat{\mathbf{x}}^1) + \boldsymbol{\lambda}^{t,2}(\mathbf{A}^2_{\boldsymbol{\theta}}\mathbf{h}^1 + \mathbf{B}^2_{\boldsymbol{\theta}}\hat{\mathbf{x}}^2) + \cdots + \boldsymbol{\lambda}^{t,t}(\mathbf{A}^t_{\boldsymbol{\theta}}\mathbf{h}^{t-1} + \mathbf{B}^t_{\boldsymbol{\theta}}\hat{\mathbf{x}}^t) + \mathbf{C}^t_{\boldsymbol{\theta}}\mathbf{h}^t \\ &= \left[\sum_{i=1}^{t}\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i}(\mathbf{A}^i_{\boldsymbol{\theta}}\mathbf{h}^{i-1} + \mathbf{B}^i_{\boldsymbol{\theta}}\hat{\mathbf{x}}^i)\right] + \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\mathbf{C}^t_{\boldsymbol{\theta}}\mathbf{h}^t \\ &= \left[\sum_{i=1}^{t}\mathrm{vjp}_{\boldsymbol{\mathcal{A}}^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i}\otimes\mathbf{h}^{i-1}) + \mathrm{vjp}_{\boldsymbol{\mathcal{B}}^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i}\otimes\hat{\mathbf{x}}^i)\right] + \mathrm{vjp}_{\boldsymbol{\mathcal{C}}^t}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\otimes\mathbf{h}^t)\end{aligned}$$

where we define $\mathrm{vjp}_{NN^i}(v) = v\cdot NN_{\boldsymbol{\theta}}(\mathrm{Input}^i)$, with $\boldsymbol{\theta}$ being NN's parameters and i being the index of Input. Now, as $\mathrm{vjp}_{\boldsymbol{\mathcal{A}}^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i}\otimes\mathbf{h}^{i-1})$, $\mathrm{vjp}_{\boldsymbol{\mathcal{B}}^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i}\otimes\hat{\mathbf{x}}^i)$, and $\mathrm{vjp}_{\boldsymbol{\mathcal{C}}^t}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\otimes\mathbf{h}^t)$are separate, we have

$$\frac{\mathrm{d}l^t}{\mathrm{d}\boldsymbol{\theta}} = \left[\sum_{i=1}^{t}\mathrm{vjp}_{\boldsymbol{\mathcal{A}}^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i}\otimes\mathbf{h}^{i-1})\right] \oplus \left[\sum_{i=1}^{t}\mathrm{vjp}_{\boldsymbol{\mathcal{B}}^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i}\otimes\hat{\mathbf{x}}^i)\right] \oplus \mathrm{vjp}_{\boldsymbol{\mathcal{C}}^t}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\otimes\mathbf{h}^t), \tag{10}$$

where $\oplus$ is vector concatenation. □

A.2 Proof for proposition 4.2

Proof. First, using the structure of ResNet, we have

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} &= \sum_{t=1}^{T} \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \frac{\mathrm{d}\mathbf{y}_K^t}{\mathrm{d}\boldsymbol{\theta}} \\ &= \sum_{t=1}^{T} \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \frac{\mathrm{d}(\mathbf{y}_0^t + \sum_{k=1}^{K} \tilde{\mathbf{y}}_k^t)}{\mathrm{d}\boldsymbol{\theta}} \\ &= \sum_{t=1}^{T} \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \sum_{k=1}^{K} \frac{\mathrm{d}\tilde{\mathbf{y}}_k^t}{\mathrm{d}\boldsymbol{\theta}} \\ &= \sum_{t=1}^{T} \sum_{k=1}^{K} \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \frac{\mathrm{d}\tilde{\mathbf{y}}_k^t}{\mathrm{d}\boldsymbol{\theta}}\end{aligned}$$

from proposition 4.1, we have proven that for a single SSM model, we have

$$\frac{\mathrm{d}l^t}{\mathrm{d}\boldsymbol{\theta}} = \left[\sum_{i=1}^{t} \mathrm{vjp}_{\boldsymbol{\mathcal{A}}^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i} \otimes \mathbf{h}^{i-1})\right] \oplus \left[\sum_{i=1}^{t} \mathrm{vjp}_{\boldsymbol{\mathcal{B}}^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t}\boldsymbol{\lambda}^{t,i} \otimes \hat{\mathbf{x}}^i)\right] \oplus \mathrm{vjp}_{\boldsymbol{\mathcal{C}}^t}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}^t} \otimes \mathbf{h}^t),$$

so for the ResNet model, we have

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} &= \sum_{t=1}^{T} \sum_{k=1}^{K} \frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \frac{\mathrm{d}\tilde{\mathbf{y}}_k^t}{\mathrm{d}\boldsymbol{\theta}} \\ &= \sum_{t=1}^{T} \sum_{k=1}^{K} \left\{ \left[\sum_{i=1}^{t} \mathrm{vjp}_{\boldsymbol{\mathcal{A}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \mathbf{h}_k^{i-1})\right] \oplus \left[\sum_{i=1}^{t} \mathrm{vjp}_{\boldsymbol{\mathcal{B}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \hat{\mathbf{x}}_k^i)\right] \oplus \mathrm{vjp}_{\boldsymbol{\mathcal{C}}_k^t}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \otimes \mathbf{h}_k^t) \right\} \\ &= \left(\sum_{t=1}^{T} \sum_{k=1}^{K} \mathrm{vjp}_{\boldsymbol{\mathcal{C}}_k^t}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \otimes \mathbf{h}_k^t)\right) \\ &\oplus \left(\sum_{t=1}^{T} \sum_{k=1}^{K} \sum_{i=1}^{t} \mathrm{vjp}_{\boldsymbol{\mathcal{A}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \mathbf{h}_k^{i-1})\right) \\ &\oplus \left(\sum_{t=1}^{T} \sum_{k=1}^{K} \sum_{i=1}^{t} \mathrm{vjp}_{\boldsymbol{\mathcal{B}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \hat{\mathbf{x}}_k^i)\right) \\ &= \left(\sum_{t=1}^{T} \sum_{k=1}^{K} \mathrm{vjp}_{\boldsymbol{\mathcal{C}}_k^t}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \otimes \mathbf{h}_k^t)\right) \\ &\oplus \left(\sum_{t=1}^{T} \sum_{k=1}^{K} \sum_{i=1}^{t} \mathrm{vjp}_{\boldsymbol{\mathcal{A}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \mathbf{h}_k^{i-1})\right) \\ &\oplus \left(\sum_{t=1}^{T} \sum_{k=1}^{K} \sum_{i=1}^{t} \mathrm{vjp}_{\boldsymbol{\mathcal{B}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \hat{\mathbf{y}}_{k-1}^i)\right)\end{aligned} \tag{11}$$

where the input to $\mathrm{vjp}_{\boldsymbol{\mathcal{C}}_k^t}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t} \otimes \mathbf{h}_k^t)$, $\mathrm{vjp}_{\boldsymbol{\mathcal{A}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \mathbf{h}_k^{i-1})$, and $\mathrm{vjp}_{\boldsymbol{\mathcal{B}}_k^i}(\frac{\mathrm{d}l^t}{\mathrm{d}\mathbf{y}_K^t}\boldsymbol{\lambda}_k^{t,i} \otimes \hat{\mathbf{y}}_{k-1}^i)$ are computed with the k-th SSM and the $\hat{\mathbf{x}}_k^i = \hat{\mathbf{y}}_{k-1}^i = \mathrm{RMSNorm}(\mathbf{y}_{k-2}^i + \mathrm{SSM}_{k-1}(\hat{\mathbf{Y}}_{k-2})^i)$ (the normalized output sequence of the (k-1)-th SSM), and the adjoint state $\boldsymbol{\lambda}_k^{t,\tau} = \mathbf{C}_k^t(\prod_{i=1}^{t-\tau} \mathbf{A}_k^{t+1-i})$.

□

A.3 Proof of concept for VJP computation

As a proof of concept of why $(\mathrm{d}l^t/\mathrm{d}\mathbf{y}^t)\mathbf{C}^t_{\boldsymbol{\theta}}\mathbf{h}^t$ can computed with vjp, we present an explicit and simple example. We have $\mathbf{y} = [y_1, y_2]$, $\mathbf{h} = [h_1, h_2, h_3]$, $\boldsymbol{\theta} = \vec{\boldsymbol{\theta}}$. We then have

$$\frac{dl}{d\mathbf{y}} = \begin{bmatrix} l_{y_1} & l_{y_2} \end{bmatrix} \in \mathbb{R}^{1\times P}$$

$$\mathbf{C}_{\boldsymbol{\theta}} = \begin{bmatrix} C^{\vec{\boldsymbol{\theta}}}_{11} & C^{\vec{\boldsymbol{\theta}}}_{12} & C^{\vec{\boldsymbol{\theta}}}_{13} \\ C^{\vec{\boldsymbol{\theta}}}_{21} & C^{\vec{\boldsymbol{\theta}}}_{22} & C^{\vec{\boldsymbol{\theta}}}_{23} \end{bmatrix} \in \mathbb{R}^{P\times N\times|\boldsymbol{\theta}|}$$

$$\mathbf{h} = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} \in \mathbb{R}^{N\times 1}$$

With each $C^{\vec{\boldsymbol{\theta}}}_{ij} = [\partial C_{ij}/\partial\boldsymbol{\theta}_1, \ldots, \partial C_{ij}/\partial\boldsymbol{\theta}_{|\boldsymbol{\theta}|}] \in \mathbb{R}^{|\boldsymbol{\theta}|}$. We have

$$\begin{aligned}
\frac{\mathrm{d}l}{\mathrm{d}y}\mathbf{C}_{\boldsymbol{\theta}}\mathbf{h} &= C^{\vec{\boldsymbol{\theta}}}_{11} l_{y_1} h_1 + C^{\vec{\boldsymbol{\theta}}}_{21} l_{y_2} h_1 + C^{\vec{\boldsymbol{\theta}}}_{12} l_{y_1} h_2 + C^{\vec{\boldsymbol{\theta}}}_{22} l_{y_2} h_2 + C^{\vec{\boldsymbol{\theta}}}_{13} l_{y_1} h_3 + C^{\vec{\boldsymbol{\theta}}}_{23} l_{y_2} h_3 \\
&= [l_{y_1}h_1\ l_{y_1}h_2\ l_{y_1}h_3\ l_{y_2}h_1\ l_{y_2}h_2\ l_{y_2}h_3] \cdot [C^{\vec{\boldsymbol{\theta}}}_{11}\ C^{\vec{\boldsymbol{\theta}}}_{12}\ C^{\vec{\boldsymbol{\theta}}}_{13} C^{\vec{\boldsymbol{\theta}}}_{21}\ C^{\vec{\boldsymbol{\theta}}}_{22}\ C^{\vec{\boldsymbol{\theta}}}_{23}] \\
&= \mathrm{sum}\left(\left(\begin{bmatrix} l_{y_1} \\ l_{y_2} \end{bmatrix} \otimes \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \right) \circ \begin{bmatrix} C^{\vec{\boldsymbol{\theta}}}_{11} & C^{\vec{\boldsymbol{\theta}}}_{12} & C^{\vec{\boldsymbol{\theta}}}_{13} \\ C^{\vec{\boldsymbol{\theta}}}_{21} & C^{\vec{\boldsymbol{\theta}}}_{22} & C^{\vec{\boldsymbol{\theta}}}_{23} \end{bmatrix} \right)
\end{aligned}$$

where $\cdot$ is vector dot product, $\otimes$ is vector outer product, $\circ$ is element-wise product, and sum means summing all elements in a matrix.

B Distributed tensors' locations

We provide the specific location for each tensors in distributed training:

Table 2: Tensors stored on each GPU, part 1.

GPU index	$\mathrm{d}l(\mathbf{o}^t)/\mathrm{d}y^t_K$	h^t_k
$\upsilon = 1$	$t = 1, \ldots, T$	$t = 1, \ldots, T;\ k = 1, \ldots K//\Upsilon$
$\upsilon = 2$	$t = 1, \ldots, T$	$t = 1, \ldots, T;\ k = K//\Upsilon + 1, \ldots, 2(K//\Upsilon)$
$\ldots$	$\ldots$	$\ldots$
$\upsilon = \Upsilon - 1$	$t = 1, \ldots, T$	$t = 1, \ldots, T;\ k = (\Upsilon - 2)(K//\Upsilon) + 1, \ldots, (\Upsilon - 1)(K//\Upsilon)$
$\upsilon = \Upsilon$	$t = 1, \ldots, T$	$t = 1, \ldots, T;\ k = (\Upsilon - 1)(K//\Upsilon) + 1, \ldots, K$

Table 3: Tensors stored on each GPU, part 2.

GPU index	C^t_k
$\upsilon = 1$	$t = 1, \ldots, T;\ k = 1, \ldots K//\Upsilon$
$\upsilon = 2$	$t = 1, \ldots, T;\ k = K//\Upsilon + 1, \ldots, 2(K//\Upsilon)$
$\ldots$	$\ldots$
$\upsilon = \Upsilon - 1$	$t = 1, \ldots, T$
$\upsilon = \Upsilon$	$t = 1, \ldots, T;\ k = (\Upsilon - 1)(K//\Upsilon) + 1, \ldots, K$

Table 4: Tensors stored on each GPU, part 3.

GPU index	$\hat{y}_k^t$
$\upsilon = 1$	$t = 1, \dots, T$; $k = 0, \dots K//\Upsilon - 1$
$\upsilon = 2$	$t = 1, \dots, T$; $k = K//\Upsilon, \dots, 2(K//\Upsilon) - 1$
$\dots$	$\dots$
$\upsilon = \Upsilon - 1$	$t = 1, \dots, T$; $k = (\Upsilon - 2)(K//\Upsilon), \dots, (\Upsilon - 1)(K//\Upsilon) - 1$
$\upsilon = \Upsilon$	$t = 1, \dots, T$; $k = (\Upsilon - 1)(K//\Upsilon), \dots, K - 1$

Table 5: Tensors stored on each GPU, part 4.

GPU index	A_k^t
$\upsilon = 1$	$t = 2, \dots, T$; $k = 1, \dots K//\Upsilon$
$\upsilon = 2$	$t = 2, \dots, T$; $k = K//\Upsilon + 1, \dots, 2(K//\Upsilon)$
$\dots$	$\dots$
$\upsilon = \Upsilon - 1$	$t = 2, \dots, T$; $k = (\Upsilon - 2)(K//\Upsilon) + 1, \dots, (\Upsilon - 1)(K//\Upsilon)$
$\upsilon = \Upsilon$	$t = 2, \dots, T$; $k = (\Upsilon - 1)(K//\Upsilon) + 1, \dots, K$

Table 6: Tensors stored on each GPU, part 5.

GPU index	$\boldsymbol{\theta}_k$	Gradient$_k$
$\upsilon = 1$	$k = 1, \dots K//\Upsilon$	$k = 1, \dots K//\Upsilon$
$\upsilon = 2$	$k = K//\Upsilon + 1, \dots, 2(K//\Upsilon)$	$k = K//\Upsilon + 1, \dots, 2(K//\Upsilon)$
$\dots$	$\dots$	$\dots$
$\upsilon = \Upsilon - 1$	$k = (\Upsilon - 2)(K//\Upsilon) + 1, \dots, (\Upsilon - 1)(K//\Upsilon)$	$k = (\Upsilon - 2)(K//\Upsilon) + 1, \dots, (\Upsilon - 1)(K//\Upsilon)$
$\upsilon = \Upsilon$	$k = (\Upsilon - 1)(K//\Upsilon) + 1, \dots, K$	$k = (\Upsilon - 1)(K//\Upsilon) + 1, \dots, K$