Machine Discovery

Parallelizing RNN with DEER

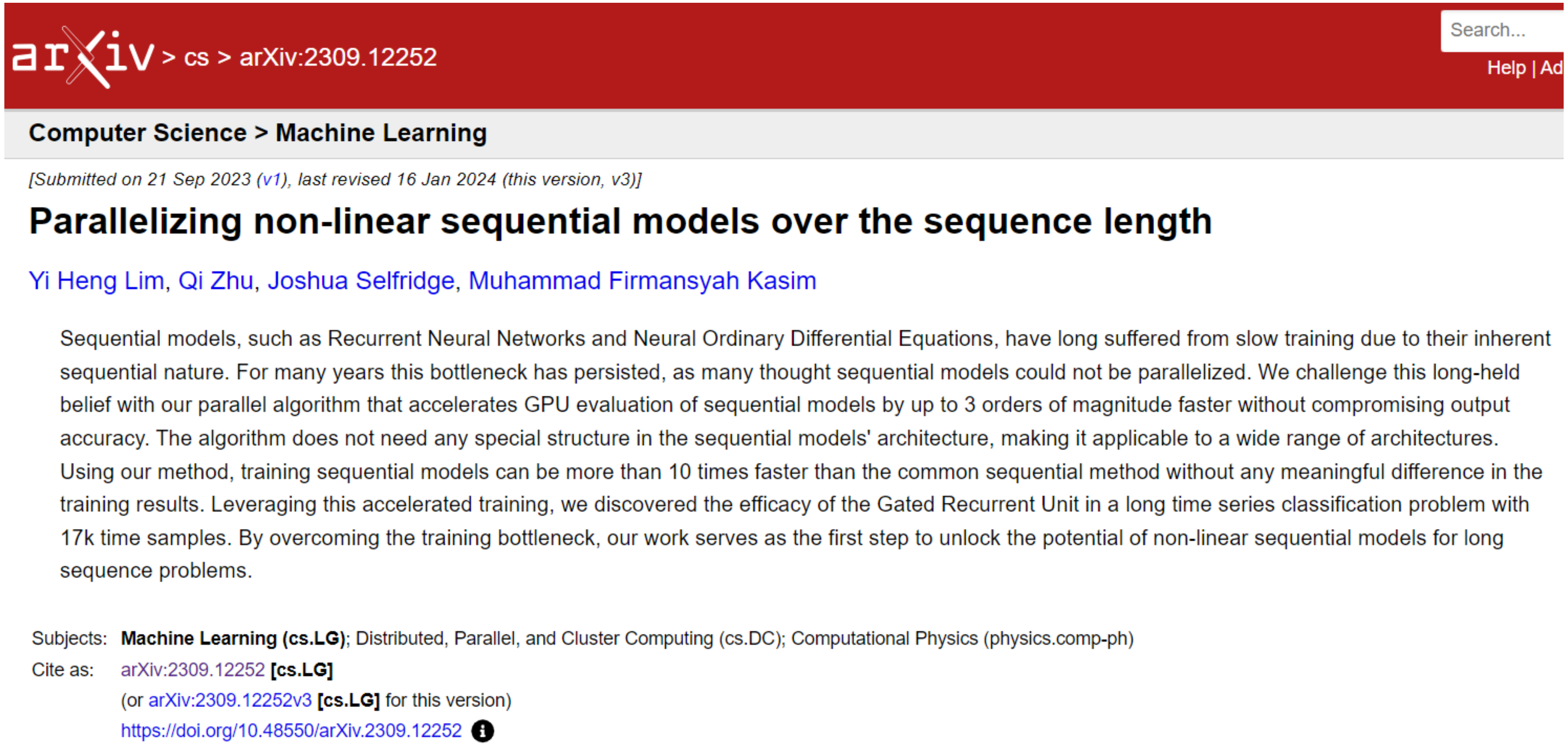
M. F. Kasim

21 February 2024

On behalf of other authors:

Y. H. Lim, Q. Zhu, J. Selfridge

The paper



> cs > arXiv:2309.12252

Search...

Help | Ad

Computer Science > Machine Learning

[Submitted on 21 Sep 2023 (v1), last revised 16 Jan 2024 (this version, v3)]

Parallelizing non-linear sequential models over the sequence length

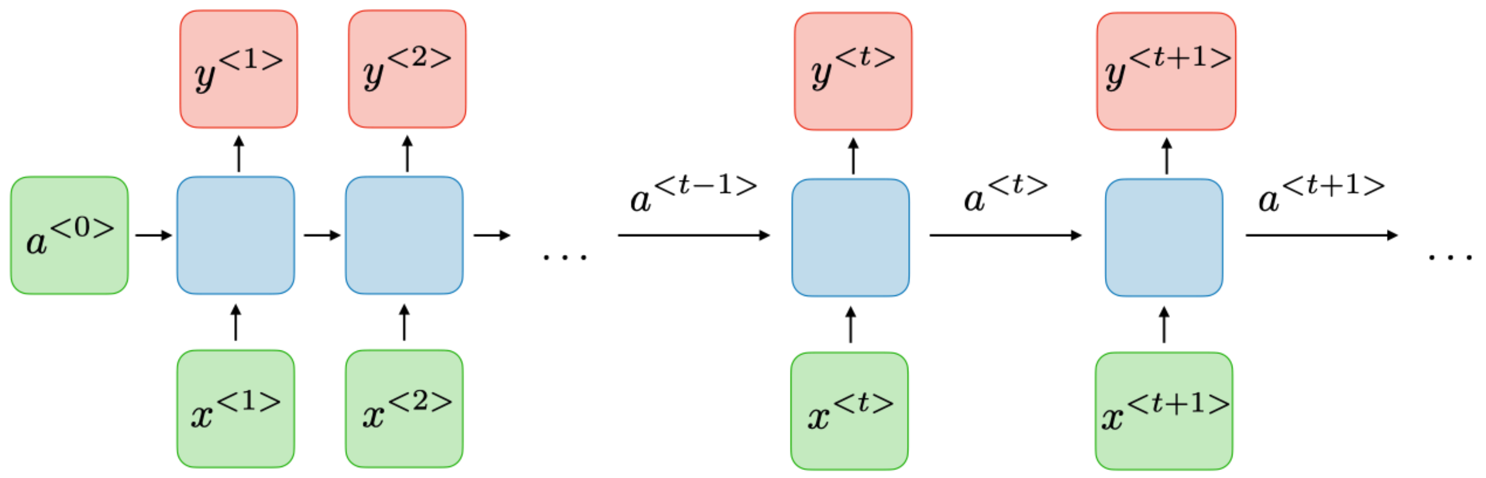
Yi Heng Lim, Qi Zhu, Joshua Selfridge, Muhammad Firmansyah Kasim

Sequential models, such as Recurrent Neural Networks and Neural Ordinary Differential Equations, have long suffered from slow training due to their inherent sequential nature. For many years this bottleneck has persisted, as many thought sequential models could not be parallelized. We challenge this long-held belief with our parallel algorithm that accelerates GPU evaluation of sequential models by up to 3 orders of magnitude faster without compromising output accuracy. The algorithm does not need any special structure in the sequential models' architecture, making it applicable to a wide range of architectures. Using our method, training sequential models can be more than 10 times faster than the common sequential method without any meaningful difference in the training results. Leveraging this accelerated training, we discovered the efficacy of the Gated Recurrent Unit in a long time series classification problem with 17k time samples. By overcoming the training bottleneck, our work serves as the first step to unlock the potential of non-linear sequential models for long sequence problems.

Subjects: **Machine Learning (cs.LG)**; Distributed, Parallel, and Cluster Computing (cs.DC); Computational Physics (physics.comp-ph)

Cite as: arXiv:2309.12252 **[cs.LG]**
(or arXiv:2309.12252v3 **[cs.LG]** for this version)
https://doi.org/10.48550/arXiv.2309.12252

Sequential nature of RNN

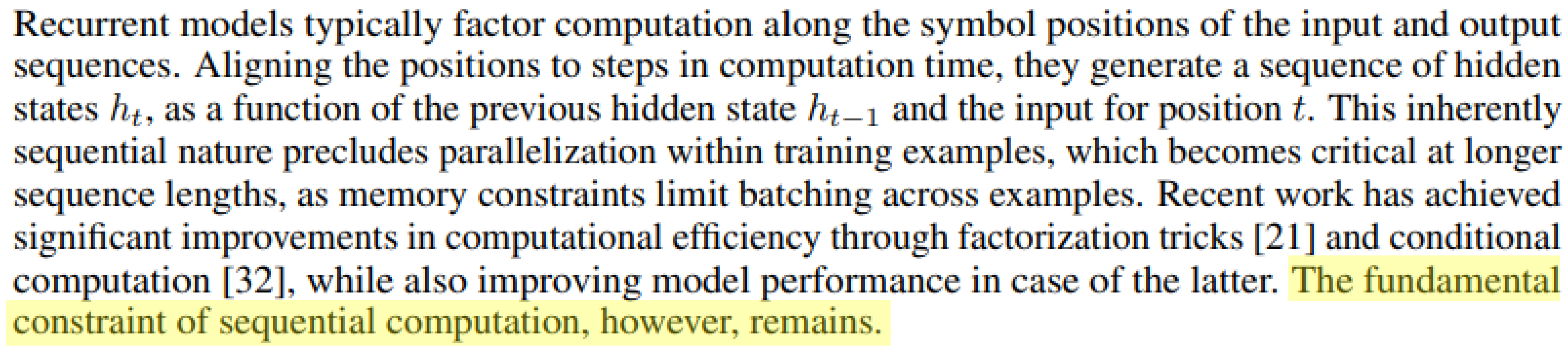


- The next state depends on the previous states: can’t be parallelized

$$a_{t+1} = f(a_t)$$

- Sequential models are slow in GPU/TPU

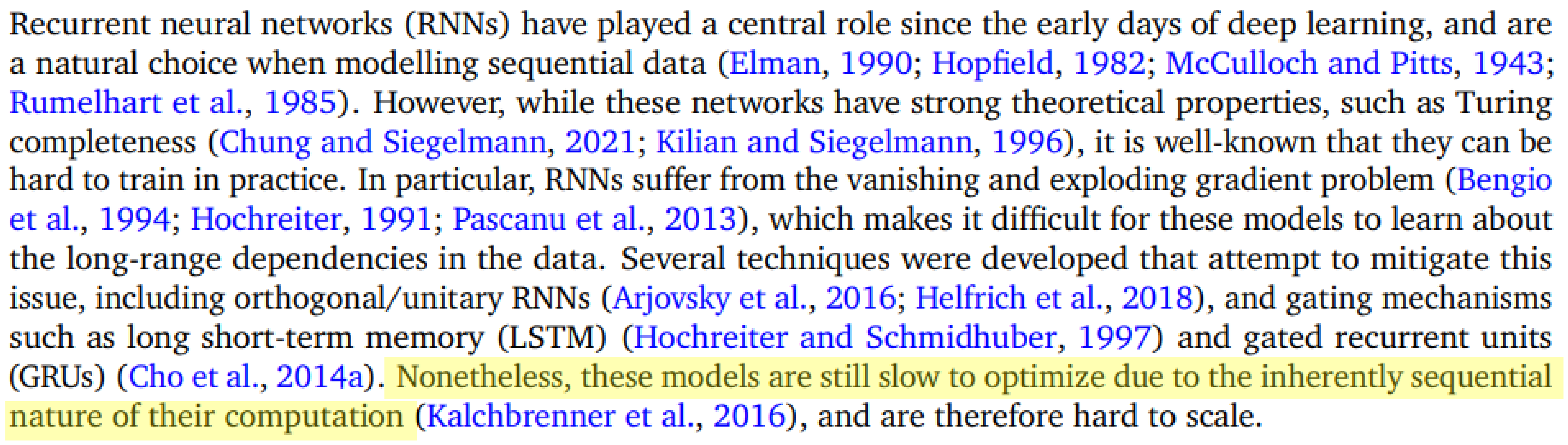
Many says RNN is not parallelizable



Recurrent models typically factor computation along the symbol positions of the input and output sequences. Aligning the positions to steps in computation time, they generate a sequence of hidden states h_t, as a function of the previous hidden state h_{t-1} and the input for position t. This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples. Recent work has achieved significant improvements in computational efficiency through factorization tricks [21] and conditional computation [32], while also improving model performance in case of the latter. The fundamental constraint of sequential computation, however, remains.

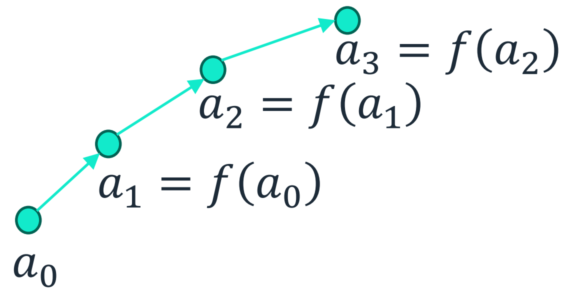
Vaswani *et al.*, Attention is all you need, 2017

Many says RNN is not parallelizable

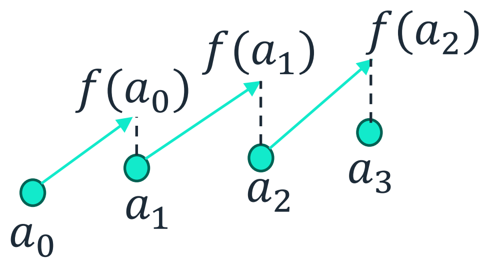

Recurrent neural networks (RNNs) have played a central role since the early days of deep learning, and are a natural choice when modelling sequential data (Elman, 1990; Hopfield, 1982; McCulloch and Pitts, 1943; Rumelhart et al., 1985). However, while these networks have strong theoretical properties, such as Turing completeness (Chung and Siegelmann, 2021; Kilian and Siegelmann, 1996), it is well-known that they can be hard to train in practice. In particular, RNNs suffer from the vanishing and exploding gradient problem (Bengio et al., 1994; Hochreiter, 1991; Pascanu et al., 2013), which makes it difficult for these models to learn about the long-range dependencies in the data. Several techniques were developed that attempt to mitigate this issue, including orthogonal/unitary RNNs (Arjovsky et al., 2016; Helfrich et al., 2018), and gating mechanisms such as long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) and gated recurrent units (GRUs) (Cho et al., 2014a). Nonetheless, these models are still slow to optimize due to the inherently sequential nature of their computation (Kalchbrenner et al., 2016), and are therefore hard to scale.

Orvieto *et al.*, Resurrecting Recurrent Neural Networks for Long Sequences, 2023

Simplest parallel algorithm for sequence: Multi-shoot algorithm with Picard iteration



Sequential
(non-parallelizable)



Multi-shoot
(parallelizable)

Multi-shoot algorithm:

1. Get some initial guess of a_i
2. Evaluate the next function $f(a_i)$
3. **Update the value of a_i** and repeat step #2 until $a_{i+1} = f(a_i)$

Simplest update algorithm (Picard iteration):

$$a_i^{(k+1)} \leftarrow f\left(a_{i-1}^{(k)}\right)$$

Picard iteration is really bad: rarely converge
Alternative: Newton's method

Newton’s method for root finder

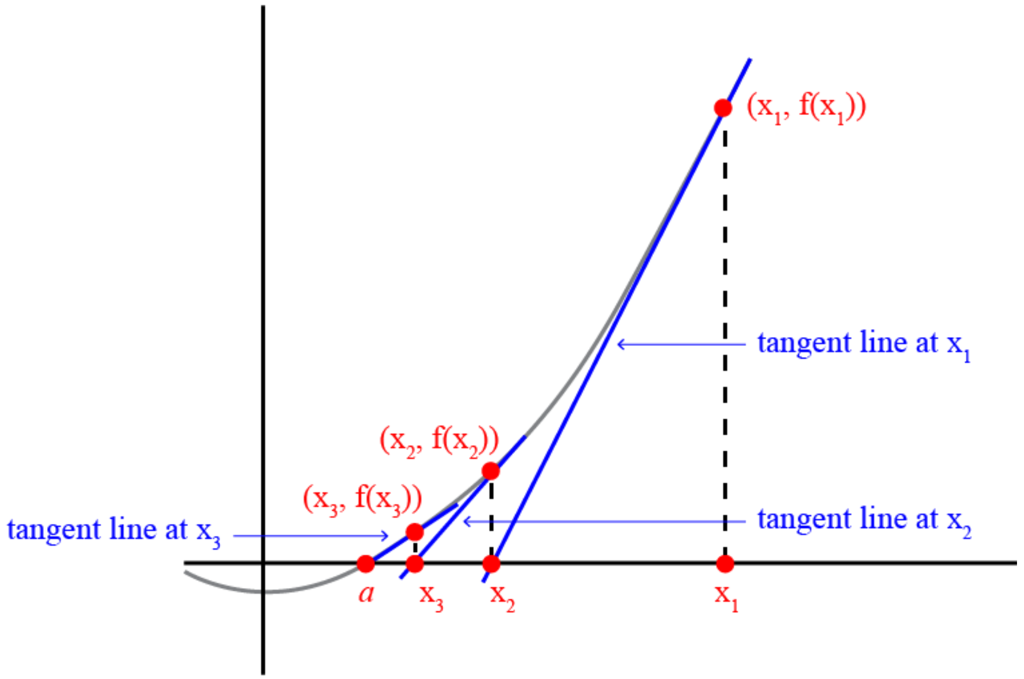
- Finding y so that $g(y) = 0$
- For 1-dimension, iterate:

$$y^{(k+1)} \leftarrow y^{(k)} - \frac{g\left(y^{(k)}\right)}{g'\left(y^{(k)}\right)}$$

- For n-dimensions, iterate:

$$\mathbf{y}^{(k+1)} \leftarrow \mathbf{y}^{(k)} - \left[\mathbf{g}'\left(\mathbf{y}^{(k)}\right)\right]^{-1} \mathbf{g}\left(\mathbf{y}^{(k)}\right)$$

- Faster and more robust convergence than Picard iteration



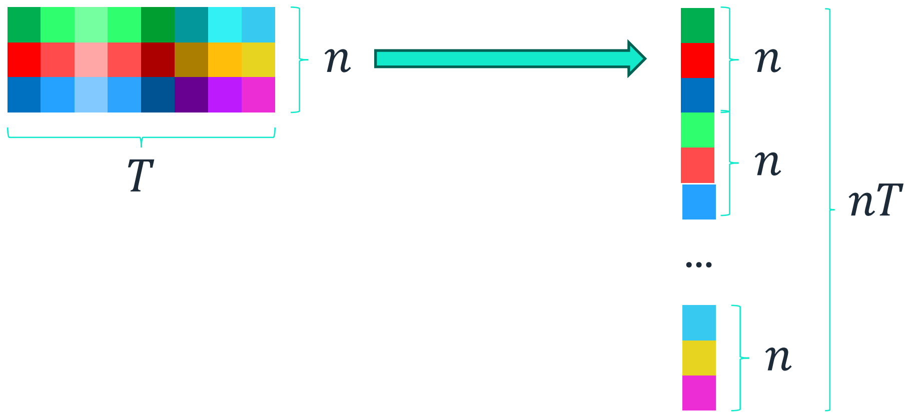
Newton's method for RNN: problem description

- RNN equation for $\mathbf{y} \in \mathrm{R}^n$:

$$\mathbf{y}_i = \mathbf{f}(\mathbf{y}_{i-1}, \mathbf{x}_i)$$

for $i = \{1, \dots, T\}$, given $\mathbf{y}_0$

- $\mathbf{y}_{1\dots T}$



Newton's method for RNN: Jacobian

- Root-finding equation for RNN

$$\mathbf{g}(\mathbf{y}_{1\ldots T}) = \mathbf{y}_{1\ldots T} - \mathbf{f}(\mathbf{y}_{0\ldots T-1}, \mathbf{x}_{1\ldots T})$$

- The Jacobian

$$\frac{\partial \mathbf{g}}{\partial \mathbf{y}_{1\ldots T}} = \mathbf{I} - \frac{\partial \mathbf{f}(\mathbf{y}_{0\ldots T-1}, \mathbf{x}_{1\ldots T})}{\partial \mathbf{y}_{1\ldots T}}$$

$$\begin{bmatrix} \mathbf{I} & & \\ & \mathbf{I} & \\ & & \mathbf{I} \end{bmatrix} - \begin{bmatrix} \frac{\partial \mathbf{f}(y_0)}{\partial \mathbf{y}_1} & \frac{\partial \mathbf{f}(y_0)}{\partial \mathbf{y}_2} & \frac{\partial \mathbf{f}(y_0)}{\partial \mathbf{y}_3} \\ \frac{\partial \mathbf{f}(y_1)}{\partial \mathbf{y}_1} & \frac{\partial \mathbf{f}(y_1)}{\partial \mathbf{y}_2} & \frac{\partial \mathbf{f}(y_1)}{\partial \mathbf{y}_3} \\ \frac{\partial \mathbf{f}(y_2)}{\partial \mathbf{y}_1} & \frac{\partial \mathbf{f}(y_2)}{\partial \mathbf{y}_2} & \frac{\partial \mathbf{f}(y_2)}{\partial \mathbf{y}_3} \end{bmatrix}$$

Newton’s method for RNN: Jacobian

- The Jacobian inverse $\mathbf{z} = \mathbf{g}'^{-1}\mathbf{a}$:
    - Find $\mathbf{z}$ such that $\mathbf{g}'\mathbf{z} = \mathbf{a}$ or

$$\mathbf{z}_i - \frac{\partial \mathbf{f}(\mathbf{y}_{i-1})}{\partial \mathbf{y}_{i-1}} \mathbf{z}_{i-1} = \mathbf{a}_i$$

    - Can be solved by **parallel scan**

$$\underset{\mathbf{g}'}{\begin{bmatrix} \mathbf{I} & & \\ -\frac{\partial \mathbf{f}(\mathbf{y}_1)}{\partial \mathbf{y}_1} & \mathbf{I} & \\ & -\frac{\partial \mathbf{f}(\mathbf{y}_2)}{\partial \mathbf{y}_2} & \mathbf{I} \end{bmatrix}} \underset{\mathbf{z}}{\begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \\ \mathbf{z}_3 \end{bmatrix}} \underset{=}{=} \underset{\mathbf{a}}{\begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 - \frac{\partial \mathbf{f}(\mathbf{y}_1)}{\partial \mathbf{y}_1} \mathbf{z}_1 \\ \mathbf{z}_3 - \frac{\partial \mathbf{f}(\mathbf{y}_2)}{\partial \mathbf{y}_2} \mathbf{z}_2 \end{bmatrix}}$$

Newton's method for RNN: putting them all together

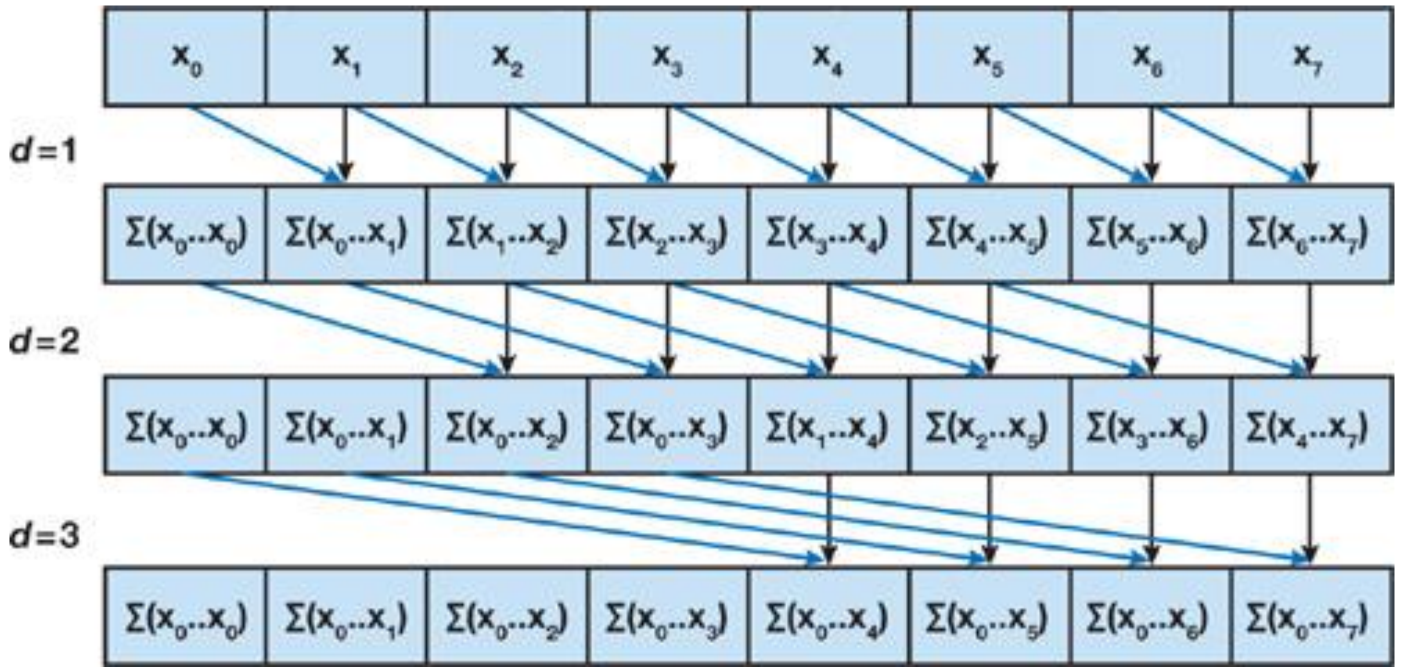
- Newton's method for multidimensional signal:

$$\mathbf{y}^{(k+1)} \leftarrow \mathbf{y}^{(k)} - \left[\mathbf{g}'\left(\mathbf{y}^{(k)}\right)\right]^{-1}\mathbf{g}\left(\mathbf{y}^{(k)}\right)$$

- For RNN:
  - $\mathbf{g}(\mathbf{y}_{1\ldots T}) = \mathbf{y}_{1\ldots T} - \mathbf{f}(\mathbf{y}_{0\ldots T-1}, \mathbf{x}_{1\ldots T})$
  - Jacobian inverse $\mathbf{z} = \mathbf{g}'^{-1}\mathbf{a}$: solving $\mathbf{z}_i - \frac{\partial \mathbf{f}(\mathbf{y}_{i-1})}{\partial \mathbf{y}_{i-1}}\mathbf{z}_{i-1} = \mathbf{a}_i$
- Algorithm (putting them together + a little bit of algebra):
  1. Get an initial guess $\mathbf{y}_{1\ldots T}^{(0)}$ and set $k = 0$
  2. Evaluate $\mathbf{a}_{1\ldots T} = \mathbf{f}\left(\mathbf{y}_{0\ldots T-1}^{(k)},\ \mathbf{x}_{1\ldots T}\right) - \frac{\partial \mathbf{f}}{\partial \mathbf{y}}\mathbf{y}_{0\ldots T-1}^{(k)}$ (parallelizable)
  3. Solve $\mathbf{y}_i^{(k+1)} = \mathbf{a}_i + \frac{\partial \mathbf{f}}{\partial \mathbf{y}}\mathbf{y}_{i-1}^{(k+1)}$, starting with $\mathbf{y}_0^{(k+1)} = \mathbf{y}_0$ (parallelizable with parallel scan)
  4. Repeat steps #2-#3 until convergence

Parallel prefix scan

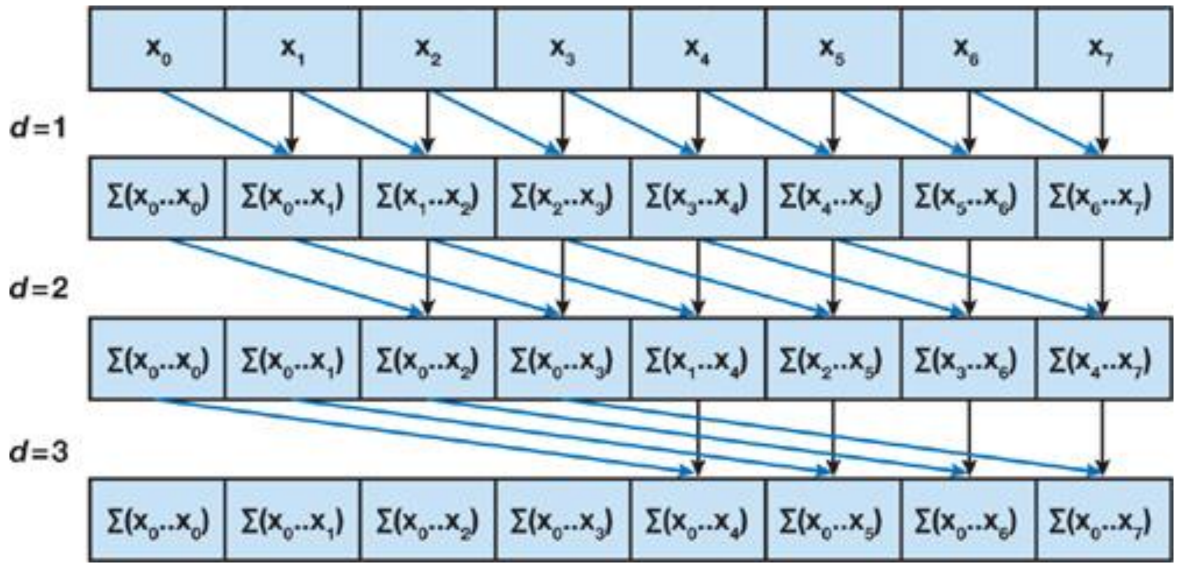
- Example: cumulative sum
  - $y_i = y_{i-1} + b_i$ where $y_{-1} = 0$
  - Operator: sum
  - [1, 4, 2, 5] → [1, 5, 7, 12]
- Example: cumulative product
  - $y_i = a_i y_{i-1}$ where $y_{-1} = 1$
  - Operator: multiply
  - [1, 4, 2, 5] → [1, 4, 8, 40]
- Our case:
  - $y_i = a_i + g_i y_{i-1}$
  - Can be parallelized using custom operator defined in Blelloch (1993)



x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
d=1
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_1..x_2)$ $\Sigma(x_2..x_3)$ $\Sigma(x_3..x_4)$ $\Sigma(x_4..x_5)$ $\Sigma(x_5..x_6)$ $\Sigma(x_6..x_7)$
d=2
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_1..x_4)$ $\Sigma(x_2..x_5)$ $\Sigma(x_3..x_6)$ $\Sigma(x_4..x_7)$
d=3
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_0..x_4)$ $\Sigma(x_0..x_5)$ $\Sigma(x_0..x_6)$ $\Sigma(x_0..x_7)$

The custom operator

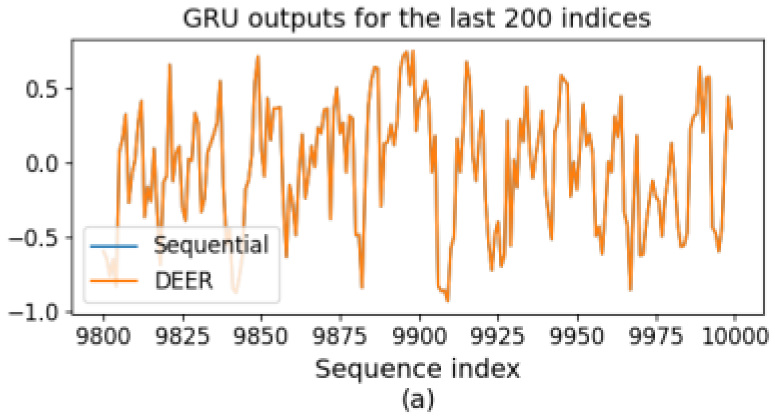
- Given a_i, g_i, y_0, compute $y_i = a_i + g_i y_{i-1}$
  - We define a variable $\mathbf{c}_i = (g_i, a_i)$ with associative operator
$$\mathbf{c}_i \cdot \mathbf{c}_j = \left(g_j g_i,\ g_j a_i + a_j\right)$$
- If we have a matrix-vector equation, let's say given $\mathbf{a}_i, \mathbf{G}_i, \mathbf{y}_0$, compute
$$\mathbf{y}_i = \mathbf{a}_i + \mathbf{G}_i \mathbf{y}_{i-1}$$
  - Then we can define $\mathbf{c}_i = (\mathbf{G}_i, \mathbf{a}_i)$ and reuse the equation above
$$\mathbf{c}_i \cdot \mathbf{c}_j = (\mathbf{G}_j \mathbf{G}_i,\ \mathbf{G}_j \mathbf{a}_i + \mathbf{a}_j)$$
- Bottleneck in matrix-matrix multiplication


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
$d=1$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_1..x_2)$ $\Sigma(x_2..x_3)$ $\Sigma(x_3..x_4)$ $\Sigma(x_4..x_5)$ $\Sigma(x_5..x_6)$ $\Sigma(x_6..x_7)$
$d=2$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_1..x_4)$ $\Sigma(x_2..x_5)$ $\Sigma(x_3..x_6)$ $\Sigma(x_4..x_7)$
$d=3$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_0..x_4)$ $\Sigma(x_0..x_5)$ $\Sigma(x_0..x_6)$ $\Sigma(x_0..x_7)$

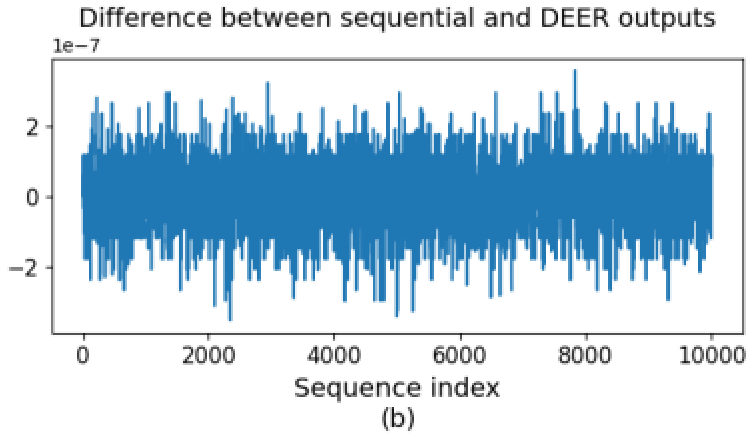
Recap of the algorithm

- Algorithm:
  1. Get an initial guess $\mathbf{y}_{1\ldots T}^{(0)}$ and set $k = 0$
  2. Evaluate $\mathbf{a}_{1\ldots T} = \mathbf{f}\left(\mathbf{y}_{0\ldots T-1}^{(k)},\ \mathbf{x}_{1\ldots T}\right) - \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \mathbf{y}_{0\ldots T-1}^{(k)}$ (parallelizable)
  3. Solve $\mathbf{y}_i^{(k+1)} = \mathbf{a}_i + \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \mathbf{y}_{i-1}^{(k+1)}$, starting with $\mathbf{y}_0^{(k+1)} = \mathbf{y}_0$ using parallel scan with custom operator
  4. Repeat steps #2-#3 until convergence
- Typically only requires <20 iterations to converge

Results: output comparison

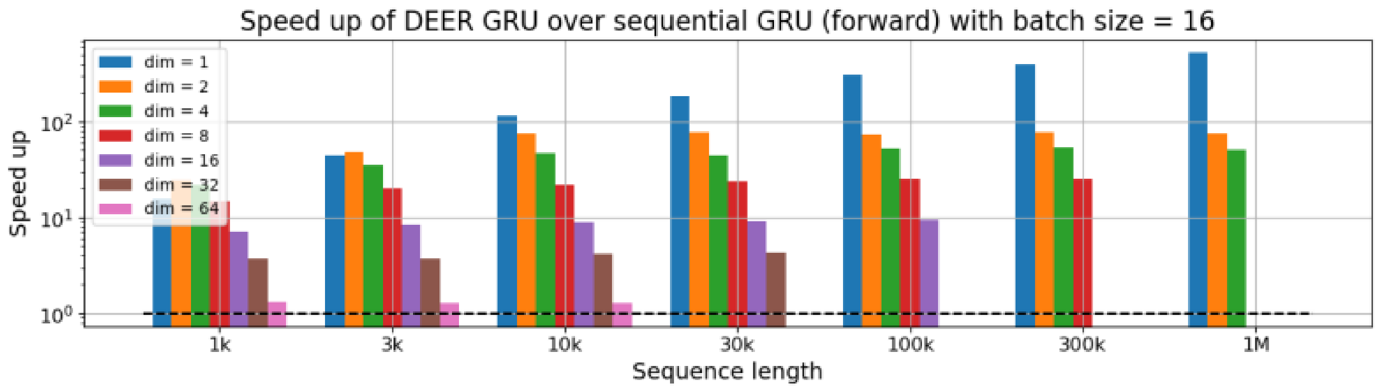


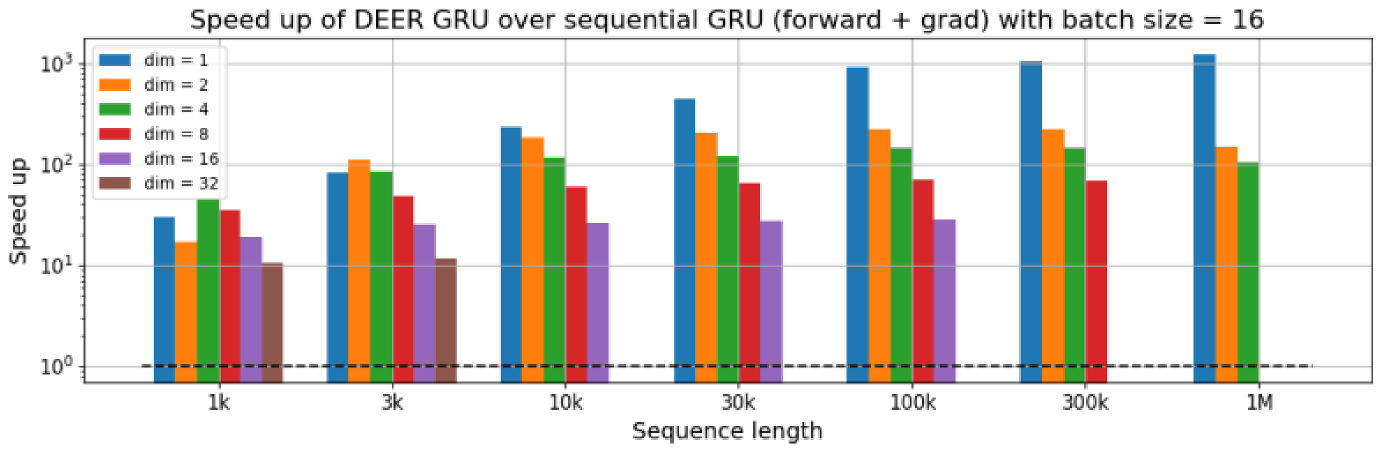
(a)



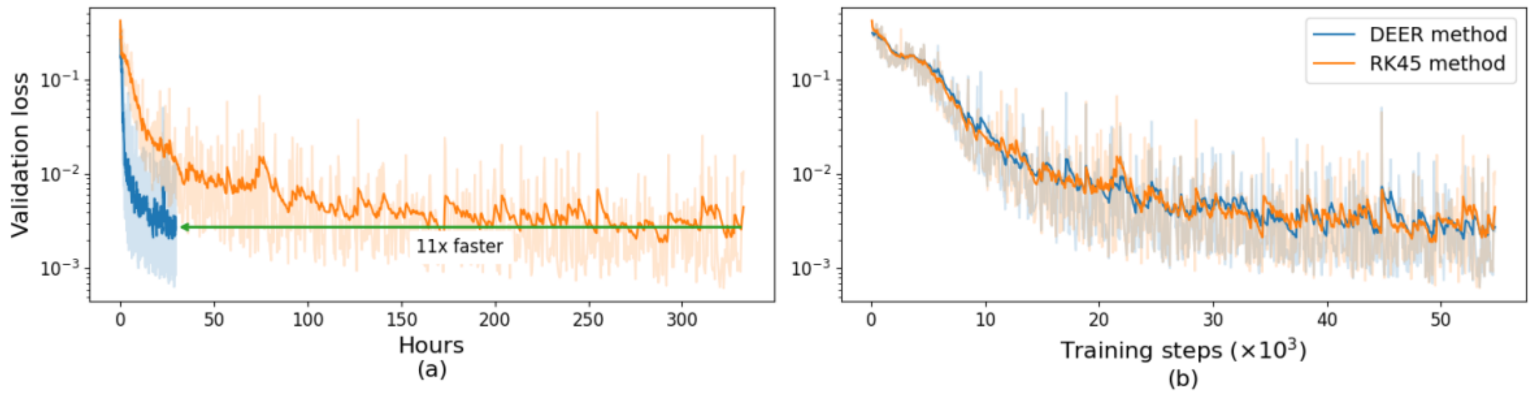
(b)

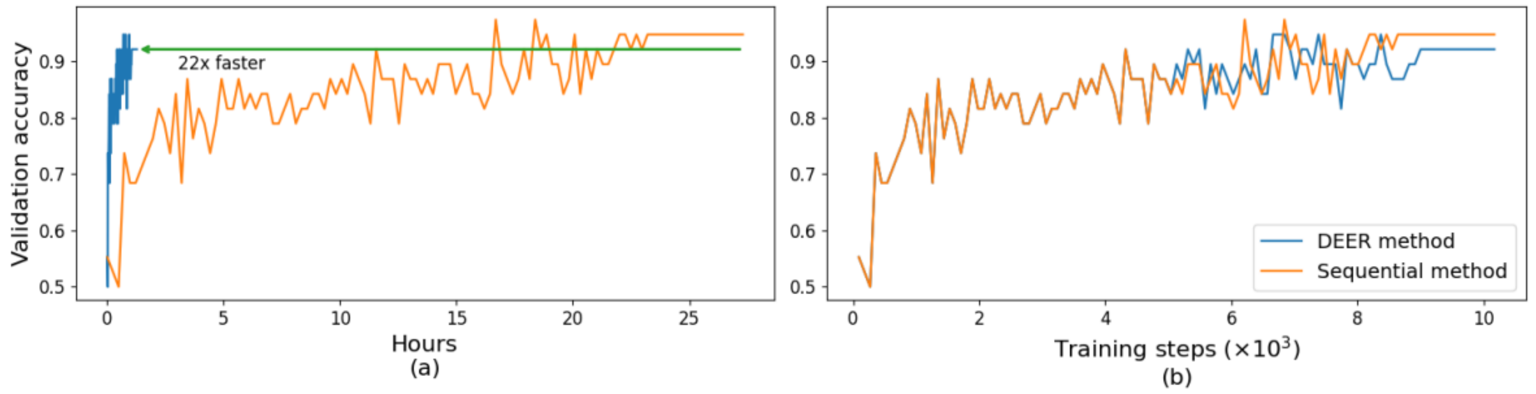
Results: speed up

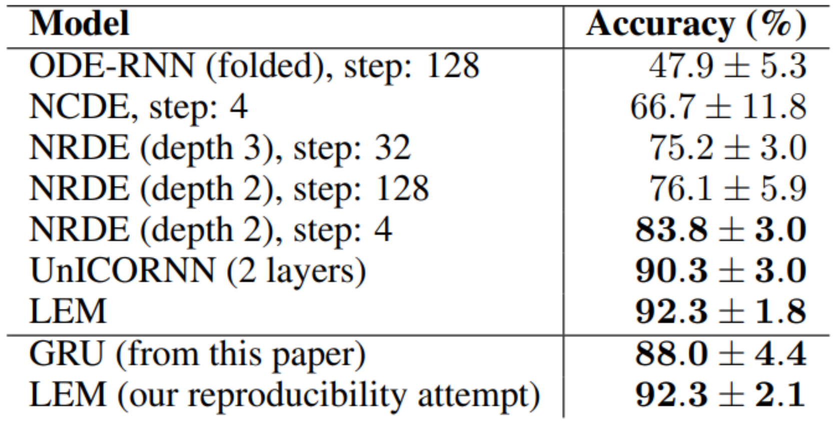




Results: usage in training







Model	**Accuracy (%)**
ODE-RNN (folded), step: 128	47.9 ± 5.3
NCDE, step: 4	66.7 ± 11.8
NRDE (depth 3), step: 32	75.2 ± 3.0
NRDE (depth 2), step: 128	76.1 ± 5.9
NRDE (depth 2), step: 4	$\mathbf{83.8 \pm 3.0}$
UnICORNN (2 layers)	$\mathbf{90.3 \pm 3.0}$
LEM	$\mathbf{92.3 \pm 1.8}$
GRU (from this paper)	$\mathbf{88.0 \pm 4.4}$
LEM (our reproducibility attempt)	$\mathbf{92.3 \pm 2.1}$

Drawbacks

- When evaluating the RNN: it may produce nan
  - Untrained RNN block typically converges, it becomes non-convergence after some training steps
- Bad scaling on large number of dimensions, $O(n^3)$, with n being the number of interdependent channels

Handling drawback #1: the process can produce nans

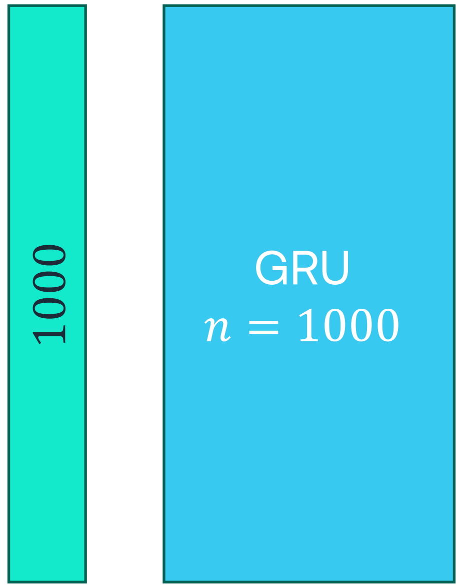
- When evaluating RNN with DEER:
  - Nans can be produced in the iterations
- This can be solved by simply substituting nans to 0 (or other values)
- The convergence is still guaranteed for RNN

Handling drawback #2: bad scaling w.r.t. number of interdependent channels

- This is due to the need to calculate the Jacobian matrix $\frac{\partial \mathbf{f}}{\partial \boldsymbol{s}}$ and perform explicit matrix-multiplications with complexity $O(n^3)$
- There is a trend in state-space where the states are detached and only a few states are interdependent (e.g., S4D has $n = 1$)
- Solution: detach states

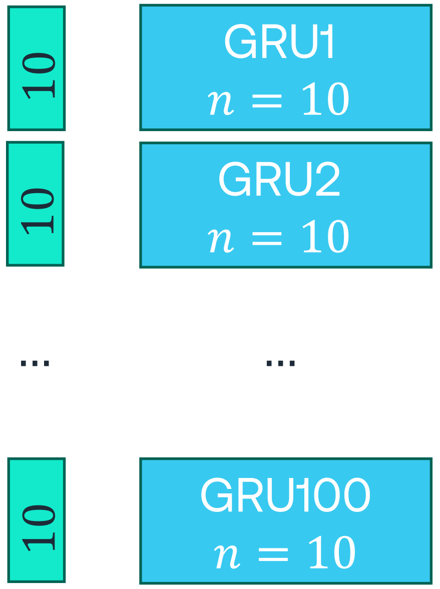
Multi-head GRU

Single head GRU



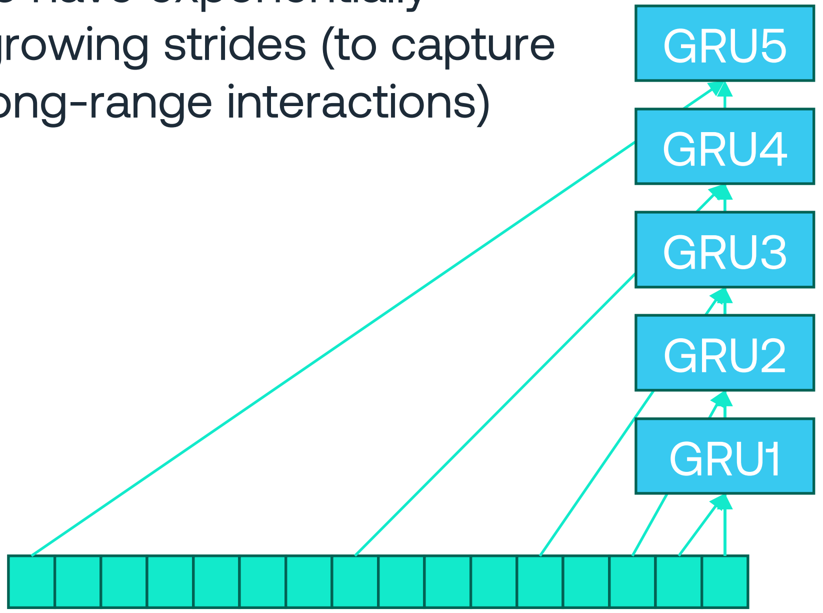
Typical use, slow with DEER

Multi-head GRU



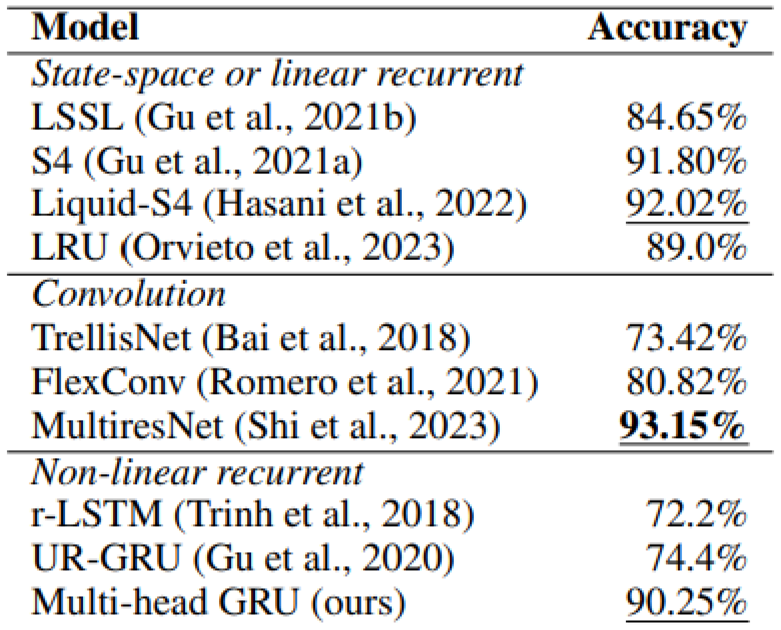
Modified use, fast with DEER

Multi-head GRU can be made to have exponentially growing strides (to capture long-range interactions)



Multi-head GRU results

- Classification on sequential CIFAR-10



Model	**Accuracy**
State-space or linear recurrent	
LSSL (Gu et al., 2021b)	84.65%
S4 (Gu et al., 2021a)	91.80%
Liquid-S4 (Hasani et al., 2022)	92.02%
LRU (Orvieto et al., 2023)	89.0%
Convolution	
TrellisNet (Bai et al., 2018)	73.42%
FlexConv (Romero et al., 2021)	80.82%
MultiresNet (Shi et al., 2023)	**93.15%**
Non-linear recurrent	
r-LSTM (Trinh et al., 2018)	72.2%
UR-GRU (Gu et al., 2020)	74.4%
Multi-head GRU (ours)	90.25%

Conclusion:
RNN is parallelizable!

Curious? See https://arxiv.org/abs/2309.12252