Myosotis: structured computation for attention like layer

Evgenii Egorov [*]
University of Amsterdam
egorov.evgenyy@ya.ru

Hanno Ackermann **Markus Nagel** **Hong Cai**
Qualcomm AI Research[†]
hackerma, markusn, hongcai @qti.qualcomm.com

Abstract

Attention layers apply a sequence-to-sequence mapping whose parameters depend on the pairwise interactions of the input elements. However, without any structural assumptions, memory and compute scale quadratically with the sequence length. The two main ways to mitigate this are to introduce sparsity by ignoring a sufficient amount of pairwise interactions or to introduce recurrent dependence along them, as SSM does. Although both approaches are reasonable, they both have disadvantages. We propose a novel algorithm that combines the advantages of both concepts. Our idea is based on the efficient inversion of tree-structured matrices.

1 Introduction

Modeling interactions in high-dimensional objects efficiently has been a long-standing challenge in machine learning, particularly when short-range dependencies are insufficient and long-range interactions must be captured. The Transformer architecture [1] was a breakthrough in addressing long-range dependencies, but it suffers from quadratic computational complexity with respect to input length. To mitigate this, numerous efficient Transformer variants have been proposed, exploiting sparsity or hierarchical structure. Examples include BigBird [2], Longformer [3], Performer [4], and Swin Transformer [5], each designed to handle longer sequences or high-dimensional inputs.

An alternative approach is offered by State-Space Models (SSMs), which recast sequence modeling as linear dynamical systems, allowing exact or approximate recurrent computation over long sequences [6, 7, 8, 9]. SSMs have been extended to multidimensional signals, audio and video tasks, demonstrating strong performance while maintaining linear complexity [10, 11]. Variants like S4 [12] and S5[13] further improve parameterization and initialization, leading to robust training and effective long-range dependency modeling.

Building on advances in graphical models and deep learning architectures, we introduce a new structural state-space layer, Myo, illustrated in Fig. 2. Myo retains the linear complexity of existing SSM layers but explicitly leverages structural assumptions about token interactions, connecting sparsity patterns with recurrence constraints via matrix inversion. This framework allows SSM layers to emerge as a special case and enables direct inheritance of initialization schemes from the SSM literature [14, 9]. The layer is straightforward to implement (see Sec. A) and can incorporate domain knowledge in the form of graphs.

2 Background

In this section, we review two related sequence-to-sequence layers: the state space model layer (SSM) and self-attention. As our goal is computational complexity, we focus on the core components of the layer and ignore mapping from input sequence to the parameters of the layer. We identify a gap between two approaches that motivates our solution.

[*]Work done during an internship in Qualcomm AI Research. Alt. email: email.evgenii.egorov@gmail.com
[†]Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.

We denote tuples of length L by $s_{1:L}$ or s, when the length is clear. If all elements of the sequence belong to the same space V, then we write $s \in V^L$, otherwise specify per element $s_k \in V_k$. The first index of sequence is 1. We denote the input sequence of the layer as a tuple $u \in \mathbb{R}^{L\times M}$ and the output sequence as a tuple $x \in \mathbb{R}^{L\times N}$. We index elements of a tuple with lower index, for example, $u_t \in \mathbb{R}^M, x_k \in \mathbb{R}^N$ are the t-th and the k-th elements of the sequences x and u. Both self-attention and state space layers are parametrized of the sequence-to-sequence mapping $K : \mathbb{R}^{L\times M} \to \mathbb{R}^{L\times N}$, which itself can be input dependent. We refer to its matrix as the kernel matrix.

2.1 State Space Model (SSM) Layer

Given an input sequence $u_{1:L}$ and parameter sequences $A_{1:L}$ and $B_{1:L}$, with $A_k \in \mathbb{R}^{N\times N}$, $B_k \in \mathbb{R}^{N\times M}$. A state space layer maps an input sequence to the output $x_{1:L}$ by the following recurrence:

$$x_1 = B_1 u_1, \quad x_k = A_{k-1}x_{k-1} + B_k u_k \quad \text{for } k \in \{2, \dots, L\}. \tag{1}$$

As matrix multiplication is an associative operation $((AB)(CD) = ((A(B(C(D))))$, the above recurrence can be computed in parallel by reusing intermediate computations using *associative scan* operations. As differentiation is a linear operation, the same is applicable for differentiation through the associative scan on a backward pass. The time complexity on T processors is $O((L/T + \log_2 T)$ and the space complexity is linear over the sequence length L.

2.2 Attention Layer

We are given the input sequence $u_{1:L}$ and parameter sequences $q_{1:L}, k_{1:L}$, $q_k \in \mathbb{R}^N$, $k_k \in \mathbb{R}^N$ and a nonlinearity σ. We call the composition of the non-linearity and inner product the kernel $k(x,y) = \sigma(\langle x, y\rangle)$ of the attention. The attention layer maps the input sequence $u_{1:L}$ to the output sequence $x_{1:L}$ as follows:

$$k \in \{1, \dots, L\}, \quad x_k = \sum_{n=1}^{L} u_n \cdot \frac{\sigma(\langle q_k, k_n\rangle)}{\sum_{n' \in \{1,\dots,L\}} \sigma(\langle q_k, k_{n'}\rangle)}. \tag{2}$$

Hence, the complexity over both time and space of the self-attention layer is quadratic with respect to the length of the sequence L. To reduce it, additional assumptions about a function $k(x, y)$ should be made. Two common approaches are the separability and sparsity assumptions.

Separability assumption We consider the following parameterization of a fixed or learnable dictionary of functions $\{\phi_p\}_{p=1}^P, \phi_p : \mathbb{R}^N \to \mathbb{R}^{N_p}$. Using it, we define a separable kernel:

$$k(x,y) = \sum_{p=1}^{P} \langle \phi_p(x), \phi_p(y)\rangle, x, y \in \mathbb{R}^N. \tag{3}$$

As $\sum_{p=1}^P \langle \phi_p(x), \phi_p(y)\rangle = \langle(\phi_1(x), \dots, \phi_P(x)), (\phi_1(y), \dots, \phi_P(y))\rangle$, we consider the feature stacking map $\psi : \mathbb{R}^N \to \mathbb{R}^{\sum_{n=1}^P N_p}, \psi(x) = (\phi_1(x), \dots, \phi_P(x)))$ and hence the "linear" self-attention map:

$$Q_k = \psi(q_k) \in \mathbb{R}^D, \;\; K_k = \psi(k_k) \in \mathbb{R}^D, \;\; D = \sum_{n \in \{1,\dots,P\}} N_p \;\text{ for }\; k \in \{1, \dots, L\} \tag{4}$$

$$x_k = \sum_{n=1}^{L} u_n \cdot \frac{\langle Q_k, K_n\rangle}{\sum_{n'=1}^{L} \langle Q_k, K_{n'}\rangle} = (\langle Q_k, \sum_{n' \in \{1,\dots,L\}} K_{n'}\rangle)^{-1} \sum_{p=1}^{D} \left(\sum_{n=1}^{L} u_n \cdot (K_n)_p \right) (Q_k)_p. \tag{5}$$

As a result, for fixed $P \ll L$, we obtain a desirable linear complexity solution over the length of the sequence. If we keep only the lower triangular part ($n \leq k$), the relation to the state-space model can be made more explicit. Consider x_k before scaling and summating over p and denote it by x_k^p:

$$x_k^p = q_k^p \sum_{n \leq k} u_n \cdot k_n^p = \frac{q_k^P}{q_{k-1}^p} x_{k-1}^p + (k_k^p q_k^p) u_k, \tag{6}$$

which reassembles the recurrence.

Note that the separability assumption on the kernel reduces the number of free parameters of the kernel matrix K from L^2 to $L \times D$. Although the kernel matrix K is generally dense, this reduces computational costs from quadratic to linear over the sequence length.

Sparsity assumptions Another way to reduce complexity is to introduce a structural sparsity pattern in the attention kernel computation. It is convenient to represent a sparsity assumption by a graph. Consider a directed graph $G(V, E)$, where $V = \{1, \dots, L\}$ is a set of vertices, and E is a set of directed edges over V. Vertexes of the graph correspond to the sequence elements, and the directed edges are present if we assume a nonzero attention value for a general input. For each pair $(n, m) \in V \times V$, we have the following matrix element:

$$k_{nm} = \sigma(q_n, k_m), \quad \text{if } (n, m) \in E, \text{ otherwise } 0. \tag{7}$$

If the graph has structured sparsity, it can be used to reduce the computation cost. A simple example is a pattern of edges present in sliding windows $E = \{(n, m) : \ |n - m| \le T\}$, for some constant T. The kernel matrix K will be a band matrix, with block size T and therefore the computational and memory complexity will be $O(TL)$. We illustrate some sparsity patterns and the corresponding attention matrices in Figure 1.

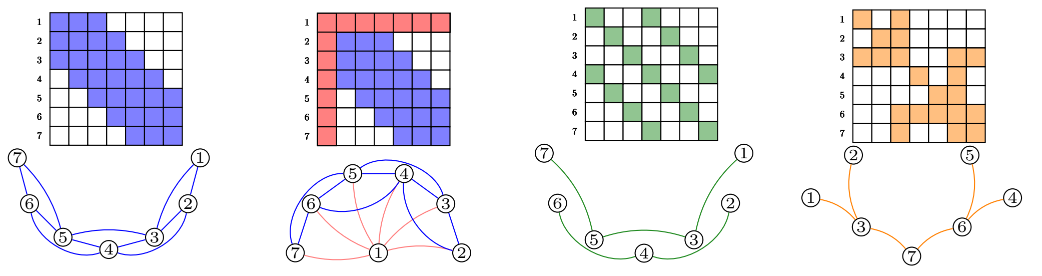
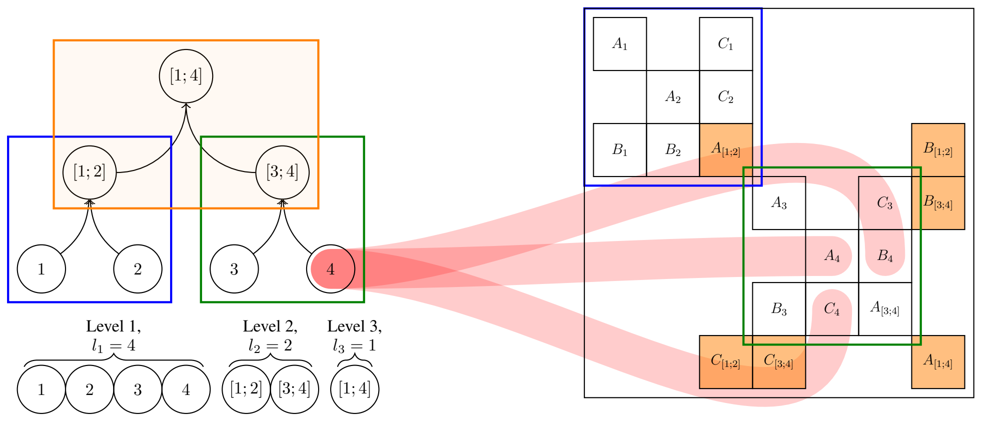

(a) Sliding window (b) Sliding + global (c) Dilated (d) Binary tree

Figure 1: Graphs of sparsity patterns (without self-loop edges) and corresponding attention matrices. Labels of nodes correspond to the usual enumeration: the top row is 1 and the last is 7.

3 The Myosotis (Myo) Layer


[1; 4]
[1; 2]
[3; 4]
1
2
3
4
Level 1, $l_1 = 4$
Level 2, $l_2 = 2$
Level 3, $l_3 = 1$
A_1 C_1 A_2 C_2 B_1 B_2 $A_{[1;2]}$ $B_{[1;2]}$
A_3 C_3 $B_{[3;4]}$ A_4 B_4 B_3 C_4 $A_{[3;4]}$
$C_{[1;2]}$ $C_{[3;4]}$ $A_{[1;4]}$

(a) Rooted tree G and its BFS flattening (b) Corresponding block T_G matrix

Figure 2: Correspondence between layer representation as a tree structure and block matrix representation. For element 4 red arrows highlight a tuple (A_4, C_4, B_4) in a graph node and its places in a matrix.

3.1 Identifying a gap

As we can see in Figure 1, there is a wide variety of sparsity patterns, possibly reflecting domain knowledge. Although this is an efficient way to reduce computational cost, hard-zeroing out elements is a restrictive choice. In contrast, SSM-like layers invoke separability, which keeps the attention matrix dense, yet has an efficient way of computation via recurrence. Intuitively, it is clear that the recurrence in 1 corresponds to the chain graph. Motivated by this, we make the connection precise and propose a Myosotis[3] layer.

The sketch of the approach is the following. We construct a tree graph G from a sequence and consider the corresponding block tree matrix T_G. The output of the proposed layer is the application of T_G^{-1} to the input u. As the matrix is tree-structured (block), therefore solving $T_G x = u$ is an efficient recurrence-like computation. Hence, we fuse both ideas discussed above: we use the sparse T_G, but apply the dense T_G^{-1}. Next, we discuss the algorithm in detail.

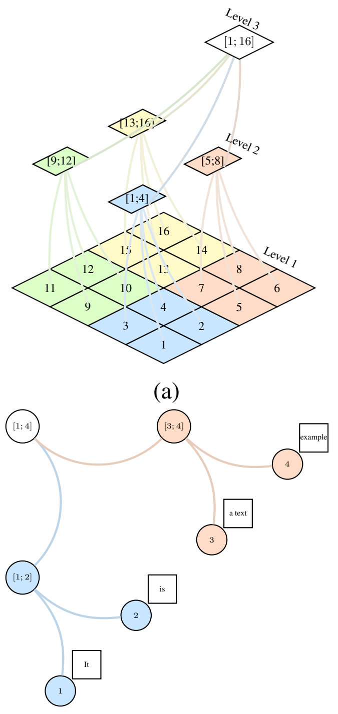
3.2 Myo Layer: Structure

For explanation purposes in this section, we will construct the matrix associated with the layer explicitly; however, as we discuss further application of the layer, it is matrix-free. We parameterize the layer with a rooted tree graph $G(V, E)$. We consider a (reversed) breadth-first search traverse (BFS) of the graph G, with D levels, where the bottom (leaf) level is the first level and the root node is the last. The number of nodes at each level is denoted by $l_k, k \in \{1, \ldots, D\}$, with $L = \sum_{k=1}^{D} l_k$. We label the leaf vertices from 1 to l_1 according to the BFS order. The non-leaf node is labeled by its subtree with segment $[n, m]$ of the most left leaf n and the most right leaf m. For example, the root label is labeled $[1; l_1]$ as it covers all leaves.

Layer input and output We consider a BFS-traverse ordered sequence $u_{1:L}$ as input and $x_{1:L}$ as output, with k-th element $x_k, u_k \in \mathbb{R}^{d_k}$. Hence, the first 1 to l_1 elements constitute the first level of the tree, the next $l_1 + 1$ to $l_1 + l_2 + 1$ the second level of the tree, and the last element with index L is the root. We denote by π a permutation mapping a BFS index to the post-order depth-first search (DFS) traverse.

In this paper, we focus on text and image domains; hence we need to introduce a graph structure on top. A natural choice is a hierarchical structure. For the image domain, we consider a quad tree structure, where the first l_1 elements of the sequence correspond to the flattering of the z order (morton) of the image, and all the next levels are virtual nodes which cover sequentially larger segments of the image. For the text domain, n-nary tree-cover chunks of tokens. In Figure 3 we provide an illustration.

For classification tasks, there are two common strategies for information aggregation before MLP classification head. The first approach is to add a classification token at the end or in the middle of the sequence. In the Myo layer, the hidden state of the root node can serve as the state of the classification token naturally. Alternatively, one can average the hidden state over tokens. We propose to average the top k layers of BFS (counting from the root), which interpolate between both common strategies, where $k = 1$ corresponds to using only the hidden state of the root.



(b)

Figure 3: Constructing a tree graph for image and text inputs.

Layer parameters With each vertex of the tree $v \in V$, we associate three matrices (A_v, B_v, C_v) as layer parameters. As the graph is a rooted tree, the association of B_v and C_v with the vertex v is the same as the association with an edge $\{v, \partial^* v\}$. The matrices B_v, C_v are responsible for the interaction between the vertex v with its parent $\partial^* v$, and the matrix A_v defines a self-interaction block. Using π ordering of labels, we construct a block matrix T_G, where A_v is a diagonal block at the index $(\pi(v), \pi(v))$ and off-diagonal blocks B_v and C_v are in the positions $(\pi(v), \pi(\partial^* v))$ and $(\pi(\partial^* v), \pi(v))$. In Figure 2, we provide an illustration of this process. The output of the Myo layer is the solution to the linear equation (with depth first

[3]The forget-me-not flower's scientific name: Myosotis

post-order traverse ordering π of matrix rows):

$$T_G(\{A_v, B_v, C_v\}_{v=1}^{L}) \underbrace{x_\pi}_{\text{Output of the layer}} = \underbrace{u_\pi}_{\text{Input to the layer}} . \tag{8}$$

For the general left part T_G a direct solver has complexity $O(L^3)$, while an iterative solver will require several iterations with complexity $O(L^2)$ per iteration. We will take advantage of the tree structure and provide linear (in a single processor) and parallelized solutions.

3.3 Myo: Efficient Computation

Now we introduce an algorithm for solving Equation 8 in $O(L)$ memory and $O(\log_k L)$ time complexity, when G is k-nary perfect tree. To build up the approach, we start with a 1-level rooted tree. We label leafs with $c \in \{1, \dots, k\}$ and the root $[1;k]$. We traverse from leaf to root (upward iteration) and backward (downward iteration), resembling Gaussian eliminations and substitutions steps. The upward iteration in block-matrix notation is the following:

$$\left[\begin{array}{cccc|c} A_1 & O & O & B_1 & u_1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ O & O & A_k & B_k & u_k \\ C_1 & \dots & C_k & A_{[1;k]} & u_{[1;k]} \end{array}\right] \Rightarrow \left[\begin{array}{cccc|c} I & \dots & O & A_1^{-1}B_1 & A_1^{-1}u_1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ O & \dots & I & A_k^{-1}B_k & A_k^{-1}u_k \\ O & \dots & O & A_{[1;k]} - \sum_{c=1}^{k} C_c A_c^{-1} B_c & u_{[1;k]} - \sum_{c=1}^{k} C_c A_c^{-1} u_c \end{array}\right] \tag{9}$$

We denote results in an upward iteration by putting a hat on a symbol: $\hat{B}_v = A_v^{-1}B_v$, $\hat{u}_v = A_v^{-1}u_v$, $\hat{A}_v = A_v - \sum_{c\in\partial v} C_c\hat{B}_c$. We get the solution x by the substitution from root to each leaf $c \in \{1, \dots, k\}$ (backward iteration):

$$\hat{A}_{[1;k]} = A_{[1;k]} - \sum_{c=1}^{k} C_c A_c^{-1} B_c, \quad x_{[1;k]} = \hat{A}_{[1;k]}^{-1}\left(u_{[1;k]} - \sum_{c=1}^{k} C_c A_c^{-1} u_c\right), x_c = A_c^{-1}u_c - (A_c^{-1}B_c)x_{[1;k]}. \tag{10}$$

From this example, we can see a general recursive algorithm. As in a rooted tree the path between any node and the root is unique, the recursion is well defined. Hence, we have an upward and backward traverse, illustrated in Figure 4.

Upward traverse In upward traverse, each vertex v modifies its coefficients $A_v, B_v \rightarrow \hat{A}_v, \hat{B}_v$, its right-part $u_v \rightarrow \hat{u}_v$ and sends messages to its parent. Messages are aggregated across sibling nodes and update information in parent node: the on-diagonal block of coefficients and the right-part. Consider a parent p and its children $c \in \partial p$, the updates are as follows:

$$\hat{B}_c \leftarrow -A_c^{-1}B_c, \qquad \hat{u}_c \leftarrow A_c^{-1}u_c,$$
$$\hat{A}_p \leftarrow A_p + \underbrace{\sum_{c\in\partial p} C_c\hat{B}_c,}_{\text{Message from child nodes}} \qquad \hat{u}_p \leftarrow u_p - \underbrace{\sum_{c\in\partial p} C_c\hat{u}_c}_{\text{Message from child nodes}} . \tag{11}$$

As soon as a parent has received both messages, we consider it as a child of its own parent $\partial^* p$, and continue if the root has not been reached.

Backward traverse By construction, the upward traverse ends at the root level, with a single root node R and block $\hat{A}_R x_R = \hat{u}_R$. We obtain the solution $x_R = \hat{A}_R^{-1}u_R$ as an initial condition and start recurrence from root to leaf over the unique path. Given a parent p and its child nodes $c \in \partial^* p$, the solution in a child node is as follows:

$$x_c \leftarrow \hat{u}_c + \underbrace{\hat{B}_c x_{\partial c}}_{\text{Message from the parent node}} . \tag{12}$$

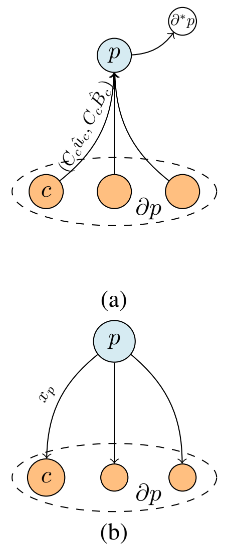


Figure 4: Message passing between parent and children nodes. A node p (a) receives messages from and (b) sends a message to children nodes ∂p.

These computations are independent across equidistant from root children-parent groups. Hence, we can process them in parallel. To this end, we consider the breadth-first levels of the tree. Initializing children and parents as the first and second levels, we scan over tree across pairs of consecutive levels and apply computations described in upward and backward traverse. We illustrate this in Figure 5 and provide details and pseudocode in Appendix A. This leads to efficient parallel computation in $O(\log_n L)$ for a n-nary perfect tree.

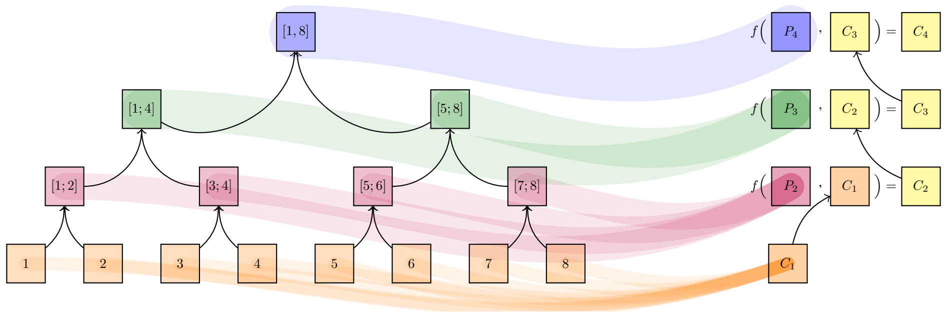


Figure 5: Illustration of the upwards traverse over BFS levels of the tree. Levels data is stored in arrays. The function f takes as input current value of carrying C (children), and current level data as parent P and output new C', which is passed up.

3.4 Myo: SSM layer as particular case

We show that a normal SSM layer is a particular case of the proposed layer, hence Myo is more general. Given a chain graph of length L, we consider the following system $T_c(\{(I, O, C_v)\}_{v=1}^{L})x_{1:L} = u_{1:L}$:

$$\begin{bmatrix} I & O & O & & \\ C_2 & I & O & & \\ O & C_3 & I & \ddots & \\ & \ddots & \ddots & \ddots & O \\ & & O & C_L & I \end{bmatrix} \begin{bmatrix} x_{\pi(1)} \\ \vdots \\ x_{\pi(L)} \end{bmatrix} = \begin{bmatrix} u_{\pi(1)} \\ \vdots \\ u_{\pi(L)} \end{bmatrix} \tag{13}$$

Matching the SSM layer As a basis, consider a 3-chain with two edges $G_c(V, E) : V_c = \{1, 2, 3\}, E_c = \{\{1, 2\}, \{2, 3\}\}$. As the matrix is lower triangular, the solution can be obtained by direct substitution:

$$\left[\begin{array}{ccc|c} I & O & O & u_1 \\ C_2 & I & O & u_2 \\ O & C_3 & I & u_3 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|cll} I & O & O & u_1 & & \\ O & I & O & u_2 & -C_2u_1 & \\ O & O & I & u_3 & -C_3u_2 & -C_3C_2u_1 \end{array}\right]. \tag{14}$$

Hence, since any chain graph with length $L > 3$ is a union of overlapping 3-chains, the solution is the following recurrence:

$$x_1 = u_1, \quad k \in \{2, \ldots, L\} : x_k = -C_kx_{k-1} + u_k = u_k - \sum_{1 \le k < l} \left(\prod_{n=l}^{k+1} C_n \right) u_l.^4 \tag{15}$$

It follows that the inverse of the lower bidiagonal matrix T_c is a dense low-triangular matrix. The value of $(T_c^{-1})_{ij}$ is the i-th coordinate of the solution $T_cx_{1:L} = u_{1:L}$ with the right part $u = (O, \ldots \underbrace{I}_{j\text{-th position}}, \ldots, O)$:

$$(T_c^{-1})_{ij} = \begin{cases} O + \delta_{ij}I, & \text{if } i \ge j, \\ -\prod_{n=j}^{i+1} C_n & \text{if } i < j. \end{cases} \tag{16}$$

To match the SSM layer, we need to also add a projection of the right part, for any left part T_G this is just a lock-diagonal rescaling with A_k. Recalling the definition of the SSM layer:

$$x_1 = S_1u_1, \quad k \in \{2, \ldots, L\} : x_k = I_{k-1}x_{k-1} + S_ku_k. \tag{17}$$

Hence, we have following correspondence:

[4] $\prod_{k=1}^{3} a_k = a_1a_2a_3, \prod_{k=3}^{1} a_k = a_3a_2a_1$

	SSM	Myo on the Chain graph
Self-term (named S in SSM, A in Myo)	$S : B_k$	$S : B_k^{-1}$
Interaction term (named I in SSM, B, C in Myo)	$I : A_k$	$I(B_k, C_k) : (O, -B_k^{-1} A_k)$

Table 1: Parameters correspondence for Myo on Chain graph in order to match given SSM layer.

Bidirectional SSM on a chain graph We note that an SSM layer has a preferable ordering, hence the i th element of the output sequence is influenced only by its predecessors and the information in elements after i is ignored. To mitigate this, a common approach is to reverse the input sequence $u_{1:L}$ and apply the same layer to both $u_{1:L}$ and $u_{L:1}$, stacking the output as channels. In contrast, Myo uses two interaction matrices (C_v, B_v) per edge, where B_v corresponds to the interaction of the vertex v with the parent $\partial^* v$ and C_v vice versa. In this case, the chain graph corresponds to the tri-diagonal matrix. Any tridiagonal matrix can be represented as the composition of lower bi-diagonal and upper bi-diagonal matrices. Hence, the application of a Myo layer on a chain graph corresponds to two consecutive applications of an SSM: first to the input $u_{1:L}$ and second to the reversed result.

Why beyond chains? Although the s chain is a tree, it is a very limited one, as any parent has exactly one child. As we shall see next, having several children leads to more interesting aggregation of information between nodes. Also, the tree structure allows us to more naturally map the neighborhoods, than flattening.

3.5 Myo: Parametrization

Partial Gauge Fixing Consider again a children-parent block: $\begin{bmatrix} A_1 & O & O & B_1 \\ \vdots & \ddots & \vdots & \vdots \\ O & O & A_k & B_k \\ C_1 & \dots & C_k & A_{[1;k]} \end{bmatrix}$. Transformation of the input vector $u_{1:L}$ with a block diagonal matrix D_v has the same effect on the output of the layer, as changing the layer parameters as follows: $A_v \to D_v^{-1} A_v, B_v \to D_{\partial v}^{-1} B_v$. We can partially fix redundancy by fixing all A_i equal to I. Note that it does not imply that diagonal elements will act trivially, i.e. diagonal blocks of inverted matrix are not identity. In order to show this, consider diagonal blocks during the upward pass:

$$\begin{array}{ccc} \text{for first (leaf) level node } c & \text{for second level node } v & \text{for third level node } w \\ \hat{A}_c = I, & \hat{A}_v = I + \sum_{c \in \partial^* v} C_c B_c, & \hat{A}_w = I + \sum_{v \in \partial^* w} C_v \left(I + \sum_{g \in \partial^* v} C_g B_g \right)^{-1} B_v \end{array} \tag{18}$$

For stability of training, we consider the matrix to be diagonally dominant and symmetric $C_v^\dagger = B_v$. We initialize the transition matrix blocks B_i as [13], diagonalizing the HIPPO-N matrix,

4 Experiments

For all experiments, we take the architecture from [13] and only change the SSM block to the Myo. The architecture consists of linear encoder, stacks of Myo layers with skip connection, and silu nonlinearity. For fairness of comparison, we did not use any augmentations, following a common experiment design on Long-Range Arena datasets. For all experiments, the block size was set to 1, that is, scalar, and the number of heads was selected to match the state dimension of the S4 and S5 models. All experiments were performed in a single NVIDIA V100 GPU accelerator.

Pixel-level 1d image classification We report results on classification tasks, including sequential CIFAR (3 channels) and sequential MNIST benchmarks. Both datasets flattened in common sequential versions: sMNIST, sCIFAR with snake order, and in quadtree aligned versions: zMNIST, zCIFAR wth morton ordering, see Figure 6. In Appendix A.4 we provide a description of the datasets and flattering procedure. For consistency, we used perfect trees with four children in Myo layers in both tasks. See Table 2 for results and the full Table 4 in Appendix A.5 (we omit some non-top scores for space considerations). The results in Table 2 suggest that without tree structure-aware flattering of an image, Myo performs on par with other architectures. However, if Morton ordering is used, which is aware of the quad tree structure, the results are slightly better. We were able to run only S5 model

Table 2: Test accuracy on image classification. We use the table from [13] and add our results.

Model (Input length)	sMNIST (784)	sCIFAR (1024)	zMNIST (784)	zCIFAR (1024)
Transformer [15, 1]	98.9	62.2	-	-
CCNN [16]	**99.72**	**93.08**	-	-
LSTM [17, 18]	98.9	63.01	-	-
r-LSTM [15]	98.4	72.2	-	-
HiPPO-RNN [6]	98.9	61.1	-	-
S4 [19, 12]	99.63	91.80	-	-
S4D [19]	-	89.92	-	-
Liquid-S4 [20]	-	92.02	-	-
S5	99.65	90.10	99.5 (our run)	89.9 (our run)
Myo (this work)	99.2	92.6	**99.7**	**93.4**

Table 3: Test accuracy on selected LRA benchmark tasks. ✗ indicates the model did not exceed random guessing. We used the table from [13] and add our results.

Model (Input length)	ListOps (2048)	Pathfinder (1024)	Path-X (16384)
Transformer	36.37	71.40	✗
Luna-256	37.25	77.72	✗
H-Trans.-1D	49.53	68.78	✗
CCNN	43.60	91.51	✗
Mega ($\mathcal{O}(L^2)$)	**63.14**	**96.01**	97.98
Mega-chunk ($\mathcal{O}(L)$)	58.76	94.41	93.81
S4D-LegS	60.47	93.06	91.95
S4-LegS	59.60	94.20	96.35
Liquid-S4	62.75	94.80	96.66
S5	62.15	95.33	**98.58**
Myo (this work)	59.5	86.1	85.7

on Morton flattering, however, we do not expect that other architectures will improve their results with changing flattering order.

Subset of Long Range Arena benchmark We consider a binary classification task of flattened images (we keep the snake order, as a benchmark introduced) of PathX dataset and a 10-way classification task in Listops benchmark. See Appendix A.4 for a detailed description of the tasks. We consider both tasks as given text sequences and use a binary perfect tree. We present results in Table 3. Without tree-structured aware flattering, Myo performs comparable, but not better.

5 Conclusion

We present Myosotis, an SSM and attention-like layer based on an efficient recurrent inversion of the quad-tree-structured matrix. The benchmarks suggest that when the data align with the quad-tree structure, Myosotis is a superior choice, otherwise performing on par with SSM. We believe that this opens up new possibilities on a structured layer design.

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[2] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems*, 33, 2020.

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

[4] Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021.

[5] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[6] Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. Hippo: Recurrent memory with optimal polynomial projections. *Advances in Neural Information Processing Systems*, 33, 2020.

[7] 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. *Advances in Neural Information Processing Systems*, 34, 2021.

[8] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *The International Conference on Learning Representations (ICLR)*, 2022.

[9] Albert Gu, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Ré. How to train your hippo: State space models with generalized basis projections. In *The International Conference on Learning Representations (ICLR)*, 2023.

[10] Eric Nguyen, Karan Goel, Albert Gu, Gordon W. Downs, Preey Shah, Tri Dao, Stephen A. Baccus, and Christopher Ré. S4nd: Modeling images and videos as multidimensional signals using state spaces. *Advances in Neural Information Processing Systems*, 35, 2022.

[11] Karan Goel, Albert Gu, Chris Donahue, and Christopher Ré. It's raw! audio generation with state-space models. *International Conference on Machine Learning (ICML)*, 2022.

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

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

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

[15] Trieu Trinh, Andrew Dai, Thang Luong, and Quoc Le. Learning longer-term dependencies in RNNs with auxiliary losses. In *International Conference on Machine Learning*, pages 4965–4974. PMLR, 2018.

[16] David Romero, David Knigge, Albert Gu, Erik Bekkers, Efstratios Gavves, Jakub Tomczak, and Mark Hoogendoorn. Towards a general purpose CNN for long range dependencies in ND. *arXiv preprint arXiv:2206.03398*, 2022.

[17] Albert Gu, Caglar Gulcehre, Thomas Paine, Matt Hoffman, and Razvan Pascanu. Improving the gating mechanism of recurrent neural networks. In *International Conference on Machine Learning*, pages 3800–3809. PMLR, 2020.

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

[19] Albert Gu, Karan Goel, Ankit Gupta, and Christopher Ré. On the parameterization and initialization of diagonal state space models. In *Advances in Neural Information Processing Systems*, 2022.

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

[21] Alex Krizhevsky. Learning multiple layers of features from tiny images. *Master's thesis, University of Toronto*, 2009.

[22] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long Range Arena: A benchmark for efficient transformers. In *International Conference on Learning Representations*, 2021.

[23] Drew Linsley, Junkyung Kim, Vijay Veerabadran, Charles Windolf, and Thomas Serre. Learning long-range spatial dependencies with horizontal gated recurrent units. *Advances in Neural Information Processing Systems*, 31, 2018.

[24] Nikita Nangia and Samuel Bowman. ListOps: A diagnostic dataset for latent tree learning. *NAACL HLT 2018*, page 92, 2018.

[25] David Romero, Robert-Jan Bruintjes, Jakub Mikolaj Tomczak, Erik Bekkers, Mark Hoogendoorn, and Jan van Gemert. Flexconv: Continuous kernel convolutions with differentiable kernel sizes. In *International Conference on Learning Representations*, 2021.

[26] David Romero, Anna Kuzina, Erik Bekkers, Jakub Mikolaj Tomczak, and Mark Hoogendoorn. CKConv: Continuous kernel convolution for sequential data. In *International Conference on Learning Representations*, 2022.

[27] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. Trellis networks for sequence modeling. In *International Conference on Learning Representations*, 2019.

[28] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

[29] Shiyu Chang, Yang Zhang, Wei Han, Mo Yu, Xiaoxiao Guo, Wei Tan, Xiaodong Cui, Michael Witbrock, Mark A Hasegawa-Johnson, and Thomas S Huang. Dilated recurrent neural networks. *Advances in Neural Information Processing Systems*, 30, 2017.

[30] Shuai Li, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao. Independently recurrent neural network (INDRNN): Building a longer and deeper RNN. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5457–5466, 2018.

[31] Mario Lezcano-Casado and David Martınez-Rubio. Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group. In *International Conference on Machine Learning*, pages 3794–3803. PMLR, 2019.

[32] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. Legendre Memory Units: Continuous-time representation in recurrent neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[33] T. Konstantin Rusch and Siddhartha Mishra. Unicornn: A recurrent model for learning very long time dependencies. In *International Conference on Machine Learning*, pages 9168–9178. PMLR, 2021.

[34] Narsimha Reddy Chilkuri and Chris Eliasmith. Parallelizing Legendre memory unit training. In *International Conference on Machine Learning*, pages 1898–1907. PMLR, 2021.

[35] N. Benjamin Erichson, Omri Azencot, Alejandro Queiruga, Liam Hodgkinson, and Michael Mahoney. Lipschitz recurrent neural networks. In *International Conference on Learning Representations*, 2021.

[36] 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. *Advances in Neural Information Processing Systems*, 34, 2021.

A Appendix: Details of forward pass computation

A.1 Myo: Shape Description

In this section, we elaborate on the tensor shapes in the Myo layer. For concreteness, we consider a basic block: a parent that covers the k child nodes, at some level l (the parent node is at level $l+1$)

$$\begin{bmatrix} A_1^l & O & O & B_1^l & \mid & u_1^l \\ \vdots & \ddots & \vdots & \vdots & \mid & \vdots \\ O & O & A_k^l & B_k^l & \mid & u_k^l \\ C_1^l & \dots & C_k^l & A_{[1;k]}^{l+1} & \mid & u_{[1;k]}^{l+1} \end{bmatrix}. \tag{19}$$

The choice of dimension of diagonal blocks fixes measurements of off-diagonal blocks. Given the input u^l with dimension $D_l = r \times d_l$, the size of the block can be equal to (D_l, D_l) or apply a layer with block size (d_l, d_l) to r different vectors u. On top of that, the layer can have a head dimension H, i.e. application of different parameters to the same input. Hence, the parameters of the layer $\mathcal{A}$ are given by a length list d (for each BFS layer), where each element A_l has dimensions (H, L_l, d_l, d_l) and input $\mathcal{U}$, with each element u_l with dimensions (H, L_l, d_l, r) (one can add as many leading dimensions as desired, for example batch). Next we describe the algorithm in pseudo-code both abstract and python-style.

A.2 Pseudo-code

Here we provide an abstract pseudocode without batch and head dimensions; see the next section A.3 for elaboration on them. We consider a tree T with L BFS layers, where 1 indicates the level of the leaf and L the level of the root. Let $\mathcal{A}_{1:L}, \mathcal{B}_{1:L}, \mathcal{C}_{1:L}, \mathcal{U}_{1:L}$ be tuples of length L. For each l in $\mathcal{A}_{1:L}$, shape(A_l) is (n_l, d_l, d_l), where n_l is the number of nodes at the level l, and d_l is the size of the block. Then shape(B_l) is (n_l, d_l, d_{l+1}) and shape(B_l) is (n_l, d_{l+1}, d_l). Finally, shape(u_l) is (n_l, d_l, r), where r is a number of right parts, that is, we solve $T[x^1_{1:L}, \dots, x^R_{1:L}] = [u^1_{1:L}, \dots, u^R_{1:L}]$. We enumerate dimensions as $(1, 2, 3)$ and refer to them as numbers in pseudo-code where the function is applicable and as ":", where the function is vectorized. We use *split* to denote splitting of the layer l array into chunks, where each chunk corresponds to the parent-child array at level $l+1$. If a tree is complete, then the BFS ordering guarantees that this holds; otherwise, it is always possible to sort branches not from the first level at most right of the tree.

Algorithm 1 Upward Traverse

```
procedure UpF($\{A_k, B_k, C_k, u_k\}_{k\in\{c,p\}}$)
    $\hat{B}_c \leftarrow -\text{Solve}_{:,2,3}(A_c, B_c)$
    $\hat{u}_c \leftarrow \text{Solve}_{:,2,:}(A_c, u_c)$
    for each $(b_{c,p}, c_{c,p})$ in split $(\hat{B}_c, C_c)$ do
        $\hat{A}_{p,\partial^* c} \leftarrow A_{p,\partial^* c} + \text{einsum}(lik, ljk)(c_{c,p}, b_{c,p})$
        $\hat{u}_{p,\partial^* c} \leftarrow u_{p,\partial^* c} - \text{einsum}(lik, ljk)(c_{c,p}, \hat{u}_{c,p})$
    end for
    return $(\hat{A}_p, B_p, C_p, \hat{u}_p), (\hat{u}_c, \hat{B}_c)$
end procedure
```

Algorithm 2 Downward Traverse

```
procedure SolveChP($B_c$, $\{u_k\}_{k\in\{c,p\}}$)
    for each $(b_{c,p}, u_p)$ in split $(B_c, u_p)$ do
        $b_c \leftarrow$ einsum(cik, kj)$(b_{c,p}, u_p)$
    end for
    $u_c \leftarrow u_c + b_c$
    return $u_c$
end procedure
```

Algorithm 3 Forward pass

```
procedure Forward pass($\mathcal{A}_{1:L}, \mathcal{B}_{1:L}, \mathcal{C}_{1:L}, \mathcal{U}_{1:L}$)
    $A_c, B_c, C_c, u_c \leftarrow \mathcal{A}_1, \mathcal{B}_1, \mathcal{C}_1, \mathcal{U}_1$
    $\mathcal{I}_{1:L}, \mathcal{X}_{1:L} \leftarrow$ List{empty}, List{empty}
    for each $p$ in $2 : L$ do
        $(A_c, B_c, C_c, u_c), \mathcal{I}_{1:L}[p] \leftarrow$
    UpF($\{A_k, B_k, C_k, u_k\}_{k\in\{c,p\}}$)
    end for
    $\mathcal{X}_{1:L}[1] \leftarrow \text{Solve}_{:,2:,}(A_c, u_c)$
    $x \leftarrow \mathcal{X}_{1:L}[1]$
    for each $u, b$ in reversed($\mathcal{I}_{1:L}$) do
        $x \leftarrow$ SolveChP$(u, b, x)$
        $\mathcal{X}_{1:L}$.appendleft($x$)
    end for
    return $\mathcal{X}_{1:L}$
end procedure
```

A.3 Python style pseudo-code

Here, "b" is a batch dimension, "h" is a head dimension, "p", "c" is a dimension corresponding to the nodes over the layer, and the last two dimensions are either block dimensions for coefficients or block dimension and number of right parts for right part input.

```python

CarryLeaf2Root: TypeAlias = Tuple[Float[Array, "b h p m m"], Float[Array, "b h p m k"],
                                  Float[Array, "b h p k m"], Float[Array, "b h p m r"],
                                  Tuple[int, ...]]

YLeaf2Root: TypeAlias = Tuple[Float[Array, "b h c m r"],
                              Float[Array, "b h c m k"],
                              Tuple[int, ...]]

def solve_leaf2root_scan_f(carry: CarryLeaf2Root,
                           x: CarryLeaf2Root) -> Tuple[CarryLeaf2Root, YLeaf2Root]:
    # carry
    # a b c y num
    # 0 1 2 3 4
    Ap, Bp, Cp, Yp, nump = x

    B = update_B_leafs(carry[0], carry[1])
    Yl = update_right_part_leaf(carry[0], carry[3])
    MC = make_leaf2parent_coeff_message(carry[2], B, carry[4])
    A = sum_update_coeff_parent(Ap, MC)
    MY = make_leaf2parent_rp_message(carry[2], Yl, carry[4])
    Y = update_right_part_parent(Yp, MY)

    return (A, Bp, Cp, Y, nump), (Yl, B, carry[4])

def solve_leaf_given_parent(Yl: Float[Array, "b h c m r"],
                            B: Float[Array, "b h c m n"],
                            Yp: Float[Array, "b h p n r"],
                            split: Tuple[int, ...]) -> Float[Array, "b h c m r"]:
    b = split(B, split, axis=2)
    Ypp = split(Yp, axis=2)
    b = concatenate([einsum('...cik, ...kj -> ...cij', bb, yy) for bb, yy in zip(b, Ypp)
 ], axis=2)
    return Yl + b

def solve_schur_scan(a: List[Float[Array, "b h l m m"]],
                     b: List[Float[Array, "b h l m k"]],
                     c: List[Float[Array, "b h l k m"]],
                     y: List[Float[Array, "b h l m r"]],
                     split: Tuple[Tuple[int, ...], ...]) -> List[Float[Array, "b h l m r"
 ]]:

    parents = zip(a[1:], b[1:], c[1:], y[1:], split[1:])
    carry = a[0], b[0], c[0], y[0], split[0]
    ys = []

    for p in parents:
        carry, y = solve_leaf2root_scan_f(carry, p)
        ys.append(y)

    yr = deque([dense_solve(carry[0], carry[3])])
    carry = yr[-1]

    for Yc, Bc, numc in reversed(ys):
        carry = solve_leaf_given_parent(Yc, Bc, carry, numc)
        yr.appendleft(carry)

    return list(yr)



```

Listing 1: Python style implementation

A.4 Datasets description

- `MNIST`: 10-way (0-9 digits) classification of a 28×28 grayscale image of a handwritten digit. The input image is flattened into a 784-length scalar sequence.

- `CIFAR` [21]: 10-way image classification using the CIFAR-10 dataset The input is flattened into a sequence of inputs of 1024-length and three-channel triple (R,G,B). There are $45,000$ training examples, $5,000$ validation examples, and $10,000$ test examples.

Both datasets are flattened in common sequential versions: sMNIST, sCIFAR with snake order, and in quad-tree aligned versions: zMNIST, zCIFAR wth morton order, see Figure 6

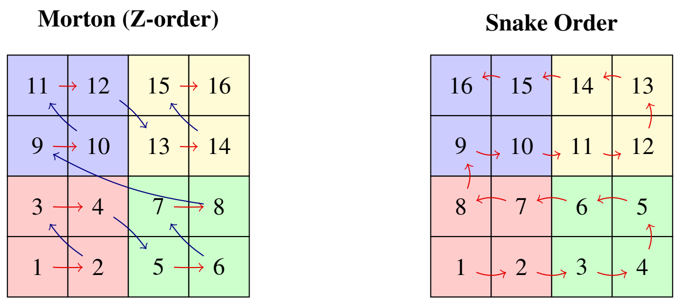


Figure 6: Example of flattening a 4×4 grid: Morton (Z-order) and Snake ordering.

`Long Range Arena` (LRA) [22] datasets:

- `Pathfinder` [23]: a binary classification task. A 32 $\times$ 32 grayscale image image shows a start and an end point as a small circle. There are a number of dashed lines in the image. The task is to classify whether there is a dashed line (or path) joining the start and end point. Sequences are all of the same length $(1,024)$. There are $160'000$ training examples, $20'000$ validation examples, and $20'000$ test examples.
- `Path-X`: Identical to the `Pathfinder` challenge, with the images are 128×128 pixels.
- `ListOps` [24]: 10-way(0-90) digits classification task, representing the integer result of the expression. Given a nested set of mathematical operations (such as `min` and `max`) and integer operands in $\{0, \ldots 9\}$, compute the integer result of the mathematical expression. Characters are encoded as one-hot vectors, with 17 unique values possible (opening brackets and operators are grouped into a single token). The sequences are padded to a maximum length of $2,000$ with a fixed indicator value. There are $96'000$ training sequences, $2,000$ validation sequences, and 2000 test sequences.

A.5 Full table

Table 4: Test accuracy on image classification. We used the table from [13] and add our results.

Model (Input length)	sMNIST (784)	sCIFAR (1024)	zMNIST (784)	zCIFAR (1024)
Transformer [15, 1]	98.9	62.2	-	-
CCNN [16]	**99.72**	**93.08**	-	-
FlexTCN [25]	99.62	80.82	-	-
CKConv [26]	99.32	63.74	-	-
TrellisNet [27]	99.20	73.42	-	-
TCN [28]	99.0	-	-	-
LSTM [17, 18]	98.9	63.01	-	-
r-LSTM [15]	98.4	72.2	-	-
Dilated GRU [29]	99.0	-	-	-
Dilated RNN [29]	98.0	-	-	-
IndRNN [30]	99.0	-	-	-
expRNN [31]	98.7	-	-	-
UR-LSTM [17]	99.28	71.00	-	-
UR-GRU [17]	99.27	74.4	-	-
LMU [32]	-	-	-	-
HiPPO-RNN [6]	98.9	61.1	-	-
UNIcoRNN [33]	-	-	-	-
LMU-FFT [34]	-	-	-	-
LipschitzRNN [35]	99.4	64.2	-	-
LSSL [36]	99.53	84.65	-	-
S4 [19, 12]	99.63	91.80	-	-
S4D [19]	-	89.92	-	-
Liquid-S4 [20]	-	92.02	-	-
S5	99.65	90.10	99.5 (our run)	89.9 (our run)
Myo (this work)	99.2	92.6	**99.7**	**93.4**