

## A Limitations of Non-Adaptive/Partially Adaptive SSMs

The expressivity of different classes of SSMs is defined by the types of dynamical systems and formal languages they are able to simulate. Appendix D analyzes the formal language expressivity and limitations of different classes of SSMs. In this section, we analyze expressivity related to different kinds of dynamical systems. First, we show that in line with Figure 1, SSM expressivity can be arranged in the order  $\text{LTI real spectra} \subset \text{LTI complex} \subset \text{LTV partial} \subset \text{LTV}$ . The models that have higher expressivity can simulate the models lower in the expressivity scale. Since  $\text{LTV w unitary spectra}$  cannot be arranged precisely in this scale, we show an example of a class of multitimescale process where a partial LTV model like Mamba cannot simulate in a fixed hidden state and layer limits.

**Expressivity of Single Block SSMs** The dynamical systems that can be simulated by single block SSMs without non-linearities can be arranged in the order  $\text{LTI real spectra} \subset \text{LTI complex} \subset \text{LTV partial} \subset \text{LTV}$ .

*Proof:* For the proof, we start with the most general LTV SSM and show that the next lower class SSM is a special case. We do the same for all the subsequent SSM classes in the expressivity chain.

The single block LTV SSM has the following discrete form:

$$\begin{cases} \frac{dx(t)}{dt} = \exp(\Delta_t A_t) x(t) + \Delta_t B_t u(t), \\ y(t) = C_t x(t). \end{cases}$$

The next lower class of ssm:  $\text{LTV partial}$  has the following form

$$\begin{cases} \frac{dx(t)}{dt} = \exp(\Delta_t A) x(t) + \Delta_t B_t u(t), \\ y(t) = C_t x(t). \end{cases}$$

Note here that the  $\Delta_t$  is a scalar that varies with time, but  $A$  is a fixed matrix. This can be derived as an instance of the LTV with  $A_t = \Delta_t A$  where the equivalence between the two hold only in the case where the dimensionality of the SSM is 1. Similarly, the next lower class  $\text{LTI complex}$  has the following form

$$\begin{cases} \frac{dx(t)}{dt} = \exp(\Delta A) x(t) + \Delta B u(t), \\ y(t) = C x(t). \end{cases}$$

This is an instance of the LTV  $\text{partial}$  with  $\Delta_t = \Delta$ ,  $B_t = B$  and  $C_t = C$ , which means all the matrices are time invariant. The final class  $\text{LTI real spectra}$  is an instance of  $\text{LTI complex}$  where the eigenvalues are further restricted to have 0 angle in the imaginary plane.

LTV is the most general class, but it is computationally infeasible to simulate the most general case. The non-diagonalizability of general matrix classes require performing a full  $O(n^3)$  matrix computation at each time-step. Hence LTV w unitary spectra with simultaneously diagonalizable unitary matrices is chosen as a principled middle ground. It is however not instantly apparent how LTV w unitary spectra compares against LTV partial. To illustrate the difference, we introduce an example of a multi-timescale process.

**Multi-timescale features:** A time-series  $u(t) \in \mathbb{R}$  is said to have multi-timescale features if the hidden state can be factorized into the following form <sup>3</sup>

$$x(t+1) = \begin{pmatrix} f(t) & 0 \\ 0 & g(t) \end{pmatrix} x(t).$$

Where  $f(t) \in \mathbb{C}$ ,  $g(t) \in \mathbb{C}$  are general complex valued time varying functions and  $f(t) \neq cg(t)$  for some constant  $c$ . That is, the timeseries exhibits at least two *independent* features denoting two different timescales.

**partial LTV SSMs in multi-timescale timeseries:** partial LTV SSMs are not able to represent multi-timescale features in data.

<sup>3</sup>the results trivially extends to systems with more than 2 dimensions

460 *Proof:* The proof is by contradiction. If partial LTV SSMs are able to solve multi-timescale  
 461 timeseries, the following  $\begin{pmatrix} f(t) & 0 \\ 0 & g(t) \end{pmatrix} = \Delta_t A$  is true. Solving the system for  $A$  leads to a  
 462 constraint on  $f(t) = cg(t)$  where  $c$  is some constant. This is true only when one of the functions is a  
 463 constant multiple of the other, that is *the two functions are dependent and have the same timescale*  
 464 *(with a possible constant factor difference).*

465 LTV w unitary spectra **SSMs in multi-timescale timeseries:** LTV w unitary spectra SSMs  
 466 can represent multi-timescale features in data as long as the  $f(t), g(t) \in \exp(i\theta)$  where  $\theta \in [-\pi, \pi]$ .  
 467  $f(t), g(t)$  can be independent.

468 *Proof:* We first substitute  $f(t) = \exp(i\theta^f(t))$  and  $g(t) = \exp(i\theta^g(t))$ . The resulting dynamical  
 469 system has  $f(t)$  and  $g(t)$  has eigenvalues and are unit magnitude themselves. This is the definition of  
 470 LTV w unitary spectra.

471 To summarize, if  $f$  and  $g$  are independent (e.g.  $f(t) = t^2, g(t) = t$ ), then the partial LTV system  
 472 cannot represent the multi-timescale features in the hidden state. On the other hand, LTV w unitary  
 473 spectra imposes a weaker constraint where the only requirement is that  $f(t)$  and  $g(t)$  are constrained  
 474 to the unit circle in the imaginary plane; the timescales of the two variables can be independent.

## 475 B AUSSM Derivation

476 We derive the AUSSM from a controlled and adaptive version of the skew-symmetric ODE used in  
 477 the jPCA procedure in computational neuroscience given below. The skew-symmetric ODE is first  
 478 discretized using the Zero Order Hold procedure and then parameterized in polar coordinates. The  
 479 steps to obtain the final AUSSM formulation are provided below.

$$\begin{cases} \frac{dx(t)}{dt} = A_t x(t) + B u(t), \\ y(t) = C x(t). \end{cases} \quad (9)$$

480 The above ODE is discretized following the Zero Order Hold procedure with a step size of  $\Delta_t$  (note  
 481 that the step size is also time varying like the recurrent matrix)

$$\begin{cases} x(t) = \exp(\Delta_t A_t) x(t-1) + \Delta_t B u(t), \\ y(t) = C x(t). \end{cases} \quad (10)$$

482 The convolution form of the above system can be derived from this recurrence as shown below  
 483 (assuming  $x(0) = 0$ )

$$y(1) = C \Delta_1 C B u(1) \quad (11)$$

$$y(2) = C \exp(\Delta_2 A_2) \Delta_1 B u(1) + \Delta_2 C B u(2) \quad (12)$$

$$\vdots \quad (13)$$

$$y(t) = C \sum_{k=1}^{t-1} \left( \prod_{l=k+1}^t \exp(\Delta_l A_l) \right) \Delta_k B u(k) + \Delta_t C B u(t) \quad (14)$$

484 Note that without additional assumptions on  $A$ , the matrix exponential and the repeated products  
 485 cannot be simplified further which can result in computationally inefficient approaches to compute the  
 486 output. We take motivation from the use of structured matrices used in efficient SSM implementations  
 487 and propose that  $A_t$  is in a class of matrices that are simultaneously diagonalizable with the same  
 488 basis. Let this diagonalizable basis be  $P$ .

$$y(t) = C \sum_{k=2}^{t-1} P \left( \prod_{l=k+1}^{t-1} \exp(\Delta_l \Lambda(A_l)) \right) P^{-1} \Delta_k B u(k) + \Delta_t C B u(t), \quad (15)$$

489 where  $\Lambda(A_l)$  is the diagonal matrix with the eigenvalues of  $A_l$  on the diagonal. Now, the repeated  
 490 matrix product has a simplified form as shown below.

$$y(t) = C P \sum_{k=2}^{t-1} \left( \exp \left( \sum_{l=k+1}^{t-1} \Delta_l \Lambda(A_l) \right) \right) P^{-1} \Delta_k B u(k) + \Delta_t C B u(t), \quad (16)$$

491 For a new set of  $B'$  and  $C'$  such that  $C' = CP$  and  $B' = P^{-1}B$ , we get

$$y(t) = C' \sum_{k=2}^{t-1} \left( \exp \left( \sum_{l=k+1}^{t-1} \Delta_l \Lambda(A_l) \right) \right) \Delta_k B' u(k) + \Delta_t C' B' u(t), \quad (17)$$

492 The above equation has one more simplification that shows the unitarity of the discrete dynamical  
 493 system. Since  $A_l$  is a skew-symmetric matrix, the eigenvalues  $\Lambda(A_l)$  is purely imaginary, meaning  
 494 the above equation simplifies further in the polar form of  $A_l$ .

$$y(t) = C' \sum_{k=2}^{t-1} \left( \exp \left( \mathbf{i} \sum_{l=k+1}^{t-1} \Delta_l \Im(\Lambda(A_l)) \right) \right) \Delta_k B' u(k) + \Delta_t C' B' u(t), \quad (18)$$

where  $\mathbf{i}^2 = -1$  is the complex iota and  $\Im(\cdot)$  is the function that obtains the imaginary component of a complex number. Since  $C'$  and  $B'$  are also complex due to the multiplication with  $P$ , we use polar forms for them too to finally obtain

$$y(t) = R_C \exp(\mathbf{i} \theta_C) \sum_{k=2}^{t-1} \left( \exp \left( \mathbf{i} \sum_{l=k+1}^{t-1} \Delta_l \Im(\Lambda(A_l)) \right) \right) \Delta_k R_B \exp(\mathbf{i} \theta_B) u(k) + \Delta_t R_C \exp(\mathbf{i} \theta_C) R_B \exp(\mathbf{i} \theta_B) u(t). \quad (19)$$

495 To handle a  $d$ -dimensional input, this formulation is replicated  $d$  times for each input dimension.  
 496 For adaptivity, we use where  $\Lambda(\Delta_l A_l)_j = \sum_r \chi_{jr} u_r(l) + \chi_j^{\text{bias}}$  and  $\Delta_{lj} = \sum_r \chi_{jr}^\Delta u_r(l) + \chi_j^{\Delta \text{bias}}$ ,  
 497 We use the above formulation in our experiments and parameterize the following for learning:  
 498  $R_C, \theta_C, R_B, \theta_B, \chi_{jr}, \chi_j^{\text{bias}}, \chi_j^{\Delta \text{bias}}, \chi_{jr}^\Delta$ .

## 499 C Eigenvalue Analysis

500 **Lemma 3** (Exponential of a Skew-Symmetric Matrix is Orthogonal). *Let  $A \in \mathbb{R}^{n \times n}$  be a real*  
 501 *skew-symmetric matrix, i.e.,  $A^\top = -A$ . Then the matrix exponential  $\exp(\Delta A)$  is orthogonal for any*  
 502  *$\Delta \in \mathbb{R}$ , i.e.,*

$$\exp(\Delta A)^\top \exp(\Delta A) = I.$$

503 *Proof.* Let  $U = \exp(\Delta A)$ . Then,

$$U^\top = (\exp(\Delta A))^\top = \exp(\Delta A^\top) = \exp(-\Delta A),$$

504 since  $A^\top = -A$ . Therefore,

$$U^\top U = \exp(-\Delta A) \exp(\Delta A) = \exp(0) = I,$$

505 which shows that  $U$  is orthogonal. □

506 **Lemma 4** (Marginal Stability of Discrete-Time Dynamics). *Let  $A \in \mathbb{R}^{n \times n}$  be a real skew-symmetric*  
 507 *matrix and define  $\Phi = \exp(\Delta A)$  for some  $\Delta > 0$ . Then all eigenvalues of  $\Phi$  lie on the complex unit*  
 508 *circle. In particular, the discrete-time linear system*

$$x(t) = \Phi x(t-1)$$

509 *is marginally stable.*

510 *Proof.* The eigenvalues of a real skew-symmetric matrix  $A$  are purely imaginary, i.e.,  $\lambda_j = i\omega_j \in i\mathbb{R}$ .  
 511 The eigenvalues of  $\Phi = \exp(\Delta A)$  are then

$$\mu_j = \exp(\Delta \lambda_j) = \exp(i\Delta \omega_j),$$

512 which all lie on the complex unit circle since  $|\exp(i\theta)| = 1$  for all  $\theta \in \mathbb{R}$ . Hence, the system exhibits  
 513 marginal stability. □

514 **Lemma 5** (Norm Preservation under Skew-Symmetric Dynamics). *Let  $A \in \mathbb{R}^{n \times n}$  be a real skew-*  
515 *symmetric matrix, and let  $\Phi = \exp(\Delta A)$ . Then for any  $x \in \mathbb{R}^n$ ,*

$$\|\Phi x\|_2 = \|x\|_2.$$

516 *Hence, the transformation does not amplify or diminish the norm of the state vector, preventing both*  
517 *gradient explosion and vanishing during backpropagation through time.*

518 *Proof.* Since  $\Phi$  is orthogonal by Lemma 1, we have:

$$\|\Phi x\|_2^2 = (\Phi x)^\top (\Phi x) = x^\top \Phi^\top \Phi x = x^\top x = \|x\|_2^2.$$

519 Taking the square root yields  $\|\Phi x\|_2 = \|x\|_2$ . □

520 **Lemma 6** (Input-Modulated Rotation Frequencies via Skew-Symmetric Generator). *Let  $A : \mathbb{R} \rightarrow$*   
521  *$\mathbb{R}^{n \times n}$  be a smooth function such that  $A(u)$  is skew-symmetric for all  $u \in \mathbb{R}$ . Then for each  $u \in \mathbb{R}$ ,*  
522 *all eigenvalues of  $A(u)$  lie on the imaginary axis, and the eigenvalues of the discrete-time transition*  
523 *matrix  $\Phi(u) = \exp(\Delta A(u))$  lie on the complex unit circle.*

524 *Furthermore, the eigenvalues of  $A(u)$  depend continuously on  $u$ , and thus the angular frequency of*  
525 *state-space rotation is smoothly and directly modulated by the input.*

526 *Proof.* Let  $A(u) \in \mathbb{R}^{n \times n}$  be skew-symmetric for all  $u \in \mathbb{R}$ , i.e.,  $A(u)^\top = -A(u)$ . It is a well-known  
527 result from linear algebra that real skew-symmetric matrices have purely imaginary eigenvalues or  
528 zero.

529 Let  $\lambda_j(u) \in \mathbb{C}$  be an eigenvalue of  $A(u)$ . Since  $A(u)$  is real and skew-symmetric,  $\lambda_j(u) = i\omega_j(u)$   
530 for some  $\omega_j(u) \in \mathbb{R}$ , and the eigenvalues come in complex-conjugate pairs if nonzero.

531 Now consider the discrete-time transition matrix:

$$\Phi(u) := \exp(\Delta A(u)).$$

532 Because the exponential of a skew-symmetric matrix is orthogonal (by Lemma 1),  $\Phi(u)$  is an  
533 orthogonal matrix. The eigenvalues of an orthogonal matrix with determinant 1 lie on the complex  
534 unit circle, i.e.,

$$|\mu_j(u)| = 1 \quad \text{for all eigenvalues } \mu_j(u) \text{ of } \Phi(u).$$

535 Furthermore, the eigenvalues of  $\Phi(u)$  are given by

$$\mu_j(u) = \exp(\Delta \lambda_j(u)) = \exp(i\Delta \omega_j(u)),$$

536 so their arguments (i.e., angular velocities) are precisely modulated by the real-valued frequencies  
537  $\omega_j(u)$ , which in turn depend on the input  $u$ .

538 To show that the rotational frequencies vary continuously with  $u$ , recall that the eigenvalues of a  
539 smooth matrix function  $A(u)$  depend continuously on  $u$ , provided that  $A(u)$  has distinct eigenvalues  
540 or that perturbations are small (which holds generically due to the structure of skew-symmetric  
541 matrices). Since  $A(u)$  is assumed to be smooth, all  $\omega_j(u)$  vary continuously with  $u$ , and therefore so  
542 do the corresponding angles  $\Delta \omega_j(u)$  of the discrete-time rotation matrix.

543 □

## 544 D Formal Language Expressivity

545 Our formal expressivity analysis uses the setting and proofs of [11] as a starting point. That is, we  
546 abstract away architectural details without loss of generality, and directly work with the already  
547 discretized form of the SSM. We assume fixed-precision floating-point arithmetic. Here, we briefly  
548 reiterate a somewhat abstract definition of our SSM to simplify the expressivity proofs.

549 **Definition 1** (SSM layer). *A single SSM layer is a sequence-to-sequence map  $\mathbb{R}^d \rightarrow \mathbb{R}^d$ ,  $(u_t) \mapsto (y_t)$*   
550 *for  $t \in [T]$  for sequence length  $T$ . It is defined recurrently by*

$$x_t = A_t \odot x_{t-1} + B_t \tag{20}$$

551 *where  $\odot$  is elementwise multiplication,  $x_0 \in \mathbb{C}^m$  with  $m = n \cdot d$ , and  $A, B : \mathbb{R}^d \rightarrow \mathbb{C}^m$  are smooth,*  
552 *input-dependent maps with  $A_t = A(u_t)$  and  $B_t = B(u_t)$ . Note that  $A$  already subsumes the*

553 discretization variable  $\Delta$ , which is itself a function of the input, as introduced in [9]. The output of  
 554 the layer is computed as

$$y_t = \phi(x_t, u_t) \quad (21)$$

555 where

$$\phi: \mathbb{C}^m \times \mathbb{R}^d \rightarrow \mathbb{R}^d, \quad (x_t, u_t) \mapsto \text{Mix}_1(\Re(\text{Mix}_2(x_t, u_t)), u_t) \quad (22)$$

556  $\text{Mix}_1$  and  $\text{Mix}_2$  contain linear maps and a non-linearity (either silu or softplus).<sup>4</sup> Note that in our  
 557 implementation, unlike [9], we do not apply normalization between the two Mix blocks but before  
 558 the input enters the layer (see Def. 4). For ease of notation, we subsume  $C_t$  into  $\text{Mix}_2$  without loss of  
 559 generality.  $\text{Mix}_2$  also usually contains a convolution of the input before the SSM recurrence, which  
 560 we ignore in expressivity analyses following [11, Remark 18].

561 **Definition 2** (Mamba layer). A Mamba layer is an SSM layer where  $A_t$  and  $B_t$ , are input-dependent  
 562 and real-valued,<sup>5</sup> and, additionally,  $A_t \in \mathbb{R}^+$  is non-negative.

**Definition 3** (AUSSM layer). An AUSSM layer is an SSM layer where  $B_t$  and  $C_t$  are fixed constant  
 functions (not input dependent) and  $A_t$  is input dependent, complex valued, and each entry has unit  
 magnitude, i.e.,

$$\forall j \in [d], \quad |A_{t,j}| = \sqrt{\Re(A_{t,j})^2 + \Im(A_{t,j})^2} = 1$$

563 **Definition 4** (Full SSM). For a full SSM we usually stack multiple layers  $(1, \dots, L)$  on top of each  
 564 other, and indicate the layer we mean by a superscript, e.g.,  $x_t^{(\ell)}$  is the hidden state at time  $t$  in layer  
 565  $\ell$ . The input to the first layer  $u_t^{(1)}$  is some embedding of the input of the full SSM computed by some  
 566 injective embedding function  $e: \Sigma \rightarrow \mathbb{R}^d$ , where  $\Sigma$  is the alphabet of possible input values at a single  
 567 timestep, and the input to layer  $\ell \in [L]$  for  $\ell > 1$  is the normalized output of the previous layer  $\ell - 1$ :

$$u_t^{(\ell)} = \text{Norm}(y_t^{(\ell-1)}) \quad (23)$$

568 We use RMSNorm for the Norm, defined by

$$\text{RMSNorm}(x) = \frac{g \odot x}{\sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2}} \quad (24)$$

569 where  $x \in \mathbb{R}^n$  and  $g \in \mathbb{R}^d$  is a learned gain parameter. Importantly, like [9], our implementation  
 570 uses skip connections between consecutive layers, i.e., for

$$y^{(\ell)} = \phi(x_t, u_t) + y^{(\ell-1)} \quad (25)$$

571 The final layer applies another RMSNorm and then a final output function.

572 We now introduce some notions from automata theory that are necessary for our expressivity results.

**Definition 5.** A deterministic finite-state automaton (FSA)  $\mathcal{A}$  is a tuple  $(\Sigma, Q, q_I, \delta)$  where  $\Sigma$  is  
 an alphabet (finite, non-empty set),  $Q$  is a finite set of states,  $q_I \in Q$  is an initial state, and  
 $\delta: Q \times \Sigma \rightarrow Q$  is a transition function. The transition function can be lifted from symbols to symbol  
 sequences as

$$\delta: Q \times \Sigma^* \rightarrow Q, \quad \delta(q, \varepsilon) = q, \quad \delta(q, \sigma_{\leq t}) = (\delta(q, \sigma_{< t}), \sigma_t)$$

573 where  $\varepsilon$  is the empty string,  $\Sigma^*$  is the Kleene closure over  $\Sigma$ , and we use boldface to mark sequences  
 574 of zero or more symbols from  $\Sigma^*$ .

575 The extended transition function  $\delta$  forms a transformation monoid under composition, called the  
 576 transition monoid of the FSA.

**Definition 6.** A set-reset automaton is an FSA whose transition function maps all states to a single  
 state for each input symbol, that is,  $\forall \sigma \in \Sigma, \exists p \in Q$  s.t.

$$\delta(q, \sigma) = p, \quad \forall q \in Q$$

577 Note that the transition monoid of a set-reset automaton is aperiodic [34].

<sup>4</sup>Here,  $\text{silu}(x) = \frac{x}{1+\exp(-x)}$  and  $\text{softplus}(x) = x \log(1 + \exp(x))$

<sup>5</sup>Note that in Mamba,  $B_t$  is directly a function of the input while  $A_t$  is input dependent through  $\Delta$ , which is  
 itself a (non-linear) function of the input.

**Definition 7.** A cyclic permutation automaton is an automaton whose transitions are permutations over states, where every input symbol acts as some power of a fixed  $k$ -cycle with  $k = |Q|$ . That is, for every symbol  $\sigma \in \Sigma$ , the symbol-specific transition map  $\delta_\sigma : Q \rightarrow Q$  is a bijection, and at least one of the symbols forms a cycle of order exactly  $k$ , i.e. for some  $a \in \Sigma$ ,  $\delta_a^k = \text{id}$  and  $\delta_a^n \neq \text{id} \forall n \in [1, k-1]$ . All other symbol-transition matrices are powers of the same  $k$ -cycle, i.e.,  $\forall b \in \Sigma, \delta_b = \delta_a^n$  for some  $n \in [0, k-1]$ , where  $\delta_a^0 = \text{id}$ .

The transition monoid of a  $k$ -cyclic permutation automaton is the cyclic group  $C_k$  [35].

We start by showing that our AUSSM architecture overcomes the limitation of most SSMs pointed out in [36] by showing that it can perform modulo counting, and therefore, can simulate cyclic permutation automata.

**Lemma 1.** For any  $k \in \mathbb{Z}^+$ , one can construct a single-layer AUSSM that counts modulo  $k$ , which means AUSSMs can simulate arbitrary cyclic permutation automata.

*Proof.* Let  $\mathcal{A} = (\Sigma, Q, q_I, \delta)$  be a cyclic permutation automaton. Now we define the input alphabet of the AUSSM to be  $\Sigma$  and choose its hidden dimension to be  $d = |\Sigma|$ . Let  $a \in \Sigma$  be the symbol whose transition function  $\delta_a$  has order  $k$ . Then we set the parameters of the AUSSM as follows: Let  $B(u) = 0 \forall u = e(\sigma), \sigma \in \Sigma$ . Let  $A(e(a)) = \exp(2\pi i/k)$ . For each other symbol  $b \in \Sigma$ , we know there exists  $m \in [0, k-1]$  such that  $\delta_b = \delta_a^m$ , so we can set  $A(e(b)) = \exp(2\pi im/k)$ . Now, there is a trivial isomorphism  $\psi$  between the values of  $x$  and the states of the FSA  $\mathcal{A}$ : Just define  $\psi: \{\exp(2\pi in/k) \mid n \in [k]\} \rightarrow \mathbb{Z}/k\mathbb{Z}, \exp(2\pi in/k) \mapsto n$ , which maps every hidden state to the corresponding state of the automaton (arranged in the order of cycle traversal by  $\delta_a$ ). Now there are  $n$  distinct possible hidden states which can be read out at finite precision.  $\square$

**A note on precision.** In a realistic fixed-precision setting, floating-point operations introduce rounding errors when computing the exponential function and repeated products thereof. A single complex multiplication introduces an error of at most  $\sqrt{5}\epsilon$  [37], where  $\epsilon$  is the machine epsilon ( $\epsilon = 2^{-53} \approx 10^{-16}$  in 64-bit double floating point precision). This means after  $N$  multiplications, the accumulated error is  $\sqrt{5}\epsilon N$  in the worst case. Two adjacent  $k$ th roots of unity are separated by a  $2\pi/k$  segment of the unit circle, so by the chord theorem, the distance between them is  $\Delta = 2 \sin(\pi/k) \approx 2\pi/k$ . Hence, approximations start overlapping if the accumulated error surpasses  $\Delta/2$ , which happens after

$$N \geq \frac{\Delta}{2\sqrt{5}\epsilon} \approx \frac{\pi}{\sqrt{5}\epsilon k} \approx \frac{1}{k} \cdot 10^{16}$$

for 64-bit double floating point precision. Even for modulo counters of up to a million, it would take 10 billion tokens for counts to be indistinguishable, which exceeds the sequence length of most datasets currently used in practice.<sup>6</sup> If more precise counters are required, one could simply switch to 128-bit precision.

The second requirement for transcending the expressivity limits of common SSMs is the ability to implement cascade products of FSAs (see [23, 34, 38] for more details on cascade products):

**Definition 8.** Let  $\mathcal{A}_1 = (\Sigma_1, Q_1, q_{I,1}, \delta_1), \mathcal{A}_2 = (\Sigma_2, Q_2, q_{I,2}, \delta_2)$  be FSAs such that  $\Sigma_2 = Q_1 \times \Sigma_1$ . Then the cascade product  $\mathcal{A}_1 \circ \mathcal{A}_2$  is the FSA  $\mathcal{C} = (\Sigma_1, Q_1 \times Q_2, (q_{I,1}, q_{I,2}), \delta_c)$  with  $\delta_c$  defined as

$$\delta_c((q_1, q_2), \sigma) = (\delta_2(q_1, (q_2, \sigma)), \delta_1(q_1, \sigma)) \quad (26)$$

Here, we use tuples of states taken from the state sets of the component FSAs to denote the state of the cascade. Intuitively, the state of the cascade at any given time is the combination of the states that the component FSAs are in at that point.

Note that for transitioning to the next state,  $\mathcal{A}_2$  requires access to the state  $\mathcal{A}_1$  was in before starting the current transition, meaning at time  $t$ , an FSA higher up in a cascade needs access to the state the lower-level FSAs were in at time  $t-1$ .

We will hence use the following crucial fact [11] used for constructing FSA cascade products in Mamba SSMs:

<sup>6</sup>For comparison, the human genome has around 3 billion base pairs.

615 **Fact 2** (Sarrof et al. [11], Lemma 17). *For any alphabet  $\Sigma$  there exists a single-layer Mamba SSM*  
 616 *such that the last-but-one input symbol can be read out from the hidden state at finite precision.*

617 We will also need the following fact about our hybrid architecture, allowing us to disregard the  
 618 particular alternating ordering of layer types:

619 **Note 1.** *We can always add an idempotent Mamba or AUSSM layer in the cascade without changing*  
 620 *the model's behavior. This can be done by setting the output projection of the SSM block in question*  
 621 *to map everything to zero. Then the input to the next layer will just be the output of the last but one*  
 622 *layer (via the skip connection). This means that for any Mamba or AUSSM with a specific behavior,*  
 623 *there is a hybrid AUSSM+Mamba with the same behavior.*

624 Now we have the necessary building blocks to show that our construction fulfills the main requirement  
 625 for increased expressivity, the ability to implement cascades of the two SSM layer types:

626 **Lemma 2.** *An SSM consisting of interleaved Mamba and AUSSM blocks (hybrid Mamba+AUSSM)*  
 627 *can implement cascade products of automata simulated by Mamba SSMs and AUSSMs.*

628 *Proof.* We want to show that the hybrid Mamba+AUSSM architecture with alternating Mamba  
 629 and AUSSM layers can implement cascade products of FSAs. In the following, we take a hybrid  
 630 Mamba+AUSSM to mean a stack of alternating Mamba and AUSSM layers, ignoring the initial  
 631 encoding and final normalization and output map. Without loss of generality, assume that the first  
 632 layer is always an AUSSM layer, and the last layer is always a Mamba layer (we can achieve this by  
 633 adding idempotent layers where necessary, see Note 1).

634 Also note that, by Note 1, any Mamba SSM and any AUSSM can be converted to an equivalent  
 635 hybrid Mamba+AUSSM.

636 It remains to be shown that a hybrid Mamba+AUSSM can simulate the cascade of two FSAs  
 637 simulated by hybrid Mamba+AUSSMs. This is simply an extension of [11, Lemma 19] to our hybrid  
 638 Mamba+AUSSM architecture.

639 Let  $\mathcal{A}_1 = (\Sigma_1, Q_1, q_{I,1}, \delta_1)$ ,  $\mathcal{A}_2 = (\Sigma_2, Q_2, q_{I,2}, \delta_2)$  be FSAs such that  $\Sigma_2 = Q_1 \times \Sigma_1$ . Assume  
 640 that there are hybrid Mamba+AUSSM models  $S_1, S_2$  that map input sequences  $(x_1, x_2, \dots, x_T)$  to  
 641 the sequences of states under  $A_1, A_2$ , at finite precision.<sup>7</sup>

642 Let  $S_c$  be the hybrid Mamba+AUSSM we want to simulate the cascade  $A_1 \circ A_2$ . The lower layers of  
 643  $S_c$  are just the layers of  $S_1$ . We add  $d$  dimensions that just copy the input via a skip connection. We  
 644 then add a Mamba layer (preceded by an idempotent AUSSM layer) that reads out the second-to-last  
 645 output of  $S_1$  in new dimensions (by Fact 2), while again forwarding the input via the skip connection.  
 646 Here, we also add a dummy dimension that is always 1, to avoid normalization making different  
 647 inputs indistinguishable. Now we have the input and the second-to-last output of  $S_1$ , corresponding  
 648 to the last state of  $A_1$ . Now the remaining layers of  $S_c$  are just those of  $S_2$ , which take this input and  
 649 compute the transition and state of  $A_2$ , again adding dimensions such that the state of  $A_2$  is separate  
 650 from the state of  $A_1$  and the input to the overall SSM. Now,  $S_c$  maps each  $w$  to the state sequence  
 651 under  $A_1 \circ A_2$ , again at finite precision. This can be inductively extended to a cascade product of  
 652 arbitrarily many FSAs.  $\square$

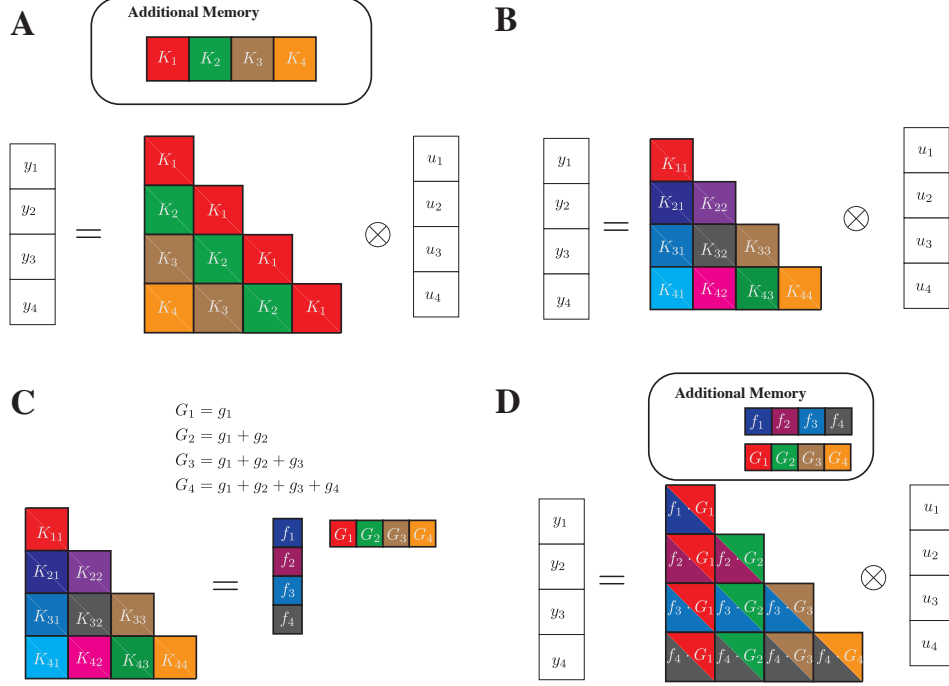
653 **Fact 3** (Consequence of Krohn-Rhodes Theorem [23] and the decomposition series of groups [27]).  
 654 *Any solvable language is recognized by a cascade of set-reset and cyclic permutation automata.*

655 **Corollary 1.** *Hybrid Mamba+AUSSM can recognize any solvable language, that is, any regular*  
 656 *language whose syntactic monoid does not contain non-solvable subgroups.*

657 *Proof.* By Lem. 1, an AUSSM can simulate cyclic permutation automata. By [11, Lem. 19], a  
 658 Mamba SSM can simulate set-reset automata. By Lem. 2, hybrid AUSSM+Mamba can simulate a  
 659 cascade of automata simulated by Mamba and AUSSM SSMs. Together with Fact 3, this means that  
 660 hybrid AUSSM+Mamba can recognize any solvable language.  $\square$

---

<sup>7</sup>Note here we implicitly assume a bijection exists between intervals on  $\mathbb{R}^d$  (the input at time  $t$ ,  $x_t$ ) and the alphabet symbols of the relevant FSA.



**Figure 3: Space Complexity of SSM formulations:** The figure illustrates an example convolution kernel for an SSM provided with four inputs at different timesteps ( $u_t$ ). The convolution is visualized as a matrix multiplication operation over the input sequence. **A.** In LTI SSMs, the convolution kernel ( $K_1, K_2, K_3, K_4$ ) is precomputed and applied to the input at different timesteps to obtain the output. **B.** In general LTV SSMs with time varying recurrence, the convolution kernel has  $O(L^2)$  elements each unique to the input and output being considered at each timestep. The use of convolution in this scenario leads to quadratic complexity in space (akin to the transformers). **C.** In the separable convolution case, the quadratic matrix of the general SSM can actually be obtained by the outer product between  $f_t$  for each timestep and the cumulative sums of a function  $g_k$  independent of  $t$ . **D.** Computing the convolution kernel can be achieved in just an additional  $O(2L)$  space.

**The importance of counting for other tasks.** Note that the ability to count modulo  $k$  does not just allow SSMs to model regular languages but also to approximate languages higher up on the Chomsky hierarchy. For example, it allows the recognition or generation of bounded Dyck languages, i.e., the correct parenthesization up to a certain depth (see [39] in the case of RNNs). Even context-sensitive language tasks can benefit from counting: For instance, sorting a sequence (the bucket sort task in §5) can be done by maintaining counters for all alphabet symbols and then outputting the symbols in order, according to their count (see counting sort and direct-address tables [40, Chapters 8 and 11]). Note that this works as long as the number of occurrences of any given symbol is smaller than the highest count expressible by the SSM, e.g.,  $k$  when using modulo  $k$  counting.

## E Complexity Analysis

SSMs leverage logarithmic complexity algorithms like FFT and parallel prefix sum to compute the convolution. Prior to this, the convolution kernel needs to be pre-computed and stored, which is the main bottleneck in computing the convolution. We will show below the space complexities for computing and storing the convolutions. Further, we show how the quadratic space complexity blowup of pure LTV systems can be managed using the separable convolution framework.



## 676 E.1 SSM Convolution

677 The convolution operation of a general SSM is given by the following

$$y(t) = \sum_{k \leq t} C_t (A_{t-1} \dots A_{k+2} A_{k+1}) B_k u(k) \quad (27)$$

678 There are two cases for the above convolution we consider:

679 **Linear Time Invariant (LTI)** : In the LTI case, the matrices in the SSM are constant over time and  
 680 the following holds

$$y(t) = \sum_{k \leq t} C A^{t-1-k} B u(k) \quad (28)$$

681 Now, the convolution kernel  $K(t, k) = C A^{t-1-k} B$  can be precomputed and since  $A^{t-k-1}$  is  
 682 common for many settings of  $t$  and  $k$  for which their difference is constant, the weights can be shared.  
 683 In fact, there are only  $O(L)$  unique entries in the convolution kernel (see Figure 3 A). The other entries  
 684 are duplicates of these entries. Once the convolution kernel is obtained, efficient algorithms like FFT  
 685 or Parallel Scan can be used to compute the convolution in  $O(\log L)$  time for each dimension, for a  
 686 total of  $O(L \log(L))$  time complexity. Therefore the total time complexity for computing the kernel  
 687 is  $O(L \log L)$  with a space complexity of  $O(L)$ .

688 **Linear Time Varying (LTV)** : In the LTV case, the matrices in the SSM can vary over time. This  
 689 introduces additional complexity in representing the convolution kernel in  $O(L^2)$  space matching  
 690 the quadratic complexity of computing self-attention in transformers. The reason for the quadratic  
 691 complexity is that the entries in the convolution kernel  $K(t, k)$  is unique for each setting of  $t, k$ . In  
 692 the case of separable convolution kernels (e.g the case of simultaneously diagonalizable matrices),  
 693 the resulting  $K(t, k)$  matrix has a further rank-1 factorization (this is discussed in detail in the main  
 694 text). This factorization enables the convolution kernel to be represented with only an additional  
 695  $O(2L)$  memory, where the 2 factor comes from each vector element in the outer product.

## 696 E.2 Parallel Scan

697 The reason for precomputing the convolution kernel is that we can apply one of the fast convolution  
 698 algorithms - FFT or parallel scan. In our case, we perform the parallel prefix sums for computing  
 699 cumulative sums. Here, we analyze the time and space complexity of the parallel prefix sum (scan)  
 700 algorithm, where the goal is to compute the prefix sums of an array  $A = [a_0, a_1, \dots, a_{L-1}]$  such that  
 701 the output array  $S$  satisfies

$$S_i = \sum_{j=0}^i a_j \quad \text{for } 0 \leq i, j < L. \quad (29)$$

702 We assume a parallel computation model such as the PRAM (Parallel Random Access Machine) or a  
 703 shared-memory model, and we are given  $P$  processors.

704 The parallel prefix sum algorithm typically consists of two main phases:

- 705 1. **Upsweep phase (Reduction):** Build a binary tree over the array and compute partial sums  
 706 from leaves to the root.
- 707 2. **Downsweep phase:** Propagate prefix sums from the root back down the tree to compute the  
 708 final result.

709 Both phases traverse a binary tree structure of height  $\log_2 L$ , assuming for simplicity that  $L$  is a  
 710 power of two. Each level of the tree can be processed in parallel.

711 **Work.** The total number of operations (work) in both phases is:

$$W(L) = \underbrace{(L-1)}_{\text{upsweep}} + \underbrace{(L-1)}_{\text{downsweep}} = 2L - 2 = \mathcal{O}(L). \quad (30)$$

712 This is the same amount of work as the sequential prefix sum algorithm, which confirms that the  
 713 parallel algorithm is work-efficient.

714 **Time Complexity with  $P$  Processors.** Using Brent’s Theorem (work-span model), the parallel  
715 time  $T_P$  on  $P$  processors is bounded by:

$$T_P \leq \frac{W(L)}{P} + S(L) = \mathcal{O}\left(\frac{L}{P} + \log L\right). \quad (31)$$

716 This means that when the number of parallel processors grow in the sequence length according to  
717  $P = \Theta(L/\log L)$ , the parallel prefix sum runs in optimal time  $\mathcal{O}(\log L)$ .

718 **Space Complexity** The space used by the algorithm includes:

- 719 • The original input array  $A$ , of size  $L$ .
- 720 • An auxiliary array to store intermediate results, typically of size  $L$ .
- 721 • Additional temporary variables per processor (constant per processor).

722 Hence, the total space complexity is:

$$\mathcal{O}(L + P) = \mathcal{O}(L) \quad (\text{since typically } P \leq L). \quad (32)$$

723 It is important to note that although the algorithm requires additional  $\mathcal{O}(L)$  space for the auxiliary  
724 variables, the CUDA kernel implementation hides these variables within the multiprocessor registers  
725 and shared memory. As a result, this complexity does not show up in the plots of either Mamba  
726 or auSSM. Existing GPU hardware for the 2080ti enables parallel processing of sequences up to  
727  $L = 2048$ . For longer sequences, the input is chunked into batches of  $L = 2048$ .

## 728 F Implementation

729 The theoretical analysis of the separable kernel formulation shows that the adaptive kernel can be  
730 implemented in only an additional linear space. However, the factor associated with the linear space  
731 is  $bdn$  where  $b$  is the batch size,  $d$  is the input dimension, and  $n$  is the hidden state dimension. In this  
732 section, we first show a Pytorch implementation of the AUSSM kernel and Mamba kernel that can  
733 be easily coded; with the higher cost of the constant factors. Next, we show how we implement the  
734 AUSSM kernel in practice so that the additional complexity is hidden within the computations of a  
735 CUDA kernel.

### 736 F.1 Pytorch

737 One of the most useful aspects of the theory of separable convolutions is that there is a relatively  
738 efficient Pytorch formulation for computing SSM kernels, even when the SSM is partially/fully  
739 time varying. However, the additional constant time penalty will be incurred. Nevertheless, the  
740 existence of such an implementation will still be interesting as it can enable fast prototyping of LTV  
741 SSMs, without dealing with the complexity of building a CUDA kernel. Here, we show two Pytorch  
742 implementations of the partial LTV Mamba kernel and the separable AUSSM kernel.

```

743 1 def mamba_ssm(u, dt, A, B, C, D, z):
744 2     """
745 3     params:
746 4         u: input Tensor (b,d,1)
747 5         dt: Delta Tensor (b,d,1)
748 6         A: Tensor (n)
749 7         B: Tensor (b,n,1)
750 8         C: Tensor (b,n,1)
751 9         D: Tensor (d)
752 10        z: Tensor (b,d,1)
753 11    Returns:
754 12        y: (b, d, 1)
755 13    """
756 14    A = einsum(A, dt, "n,bd1->bdn1")
757 15    G = torch.cumsum(axis=1)
758 16

```

```

759|7 g = einsum(exp(-G), dt, B, u, "bdn1,bd1,bn1,bd1->bdn1")
760|8 g = torch.cumsum(g, axis=-1)
761|9 f = einsum(C, exp(G), "bn1,bdn1->bdn1")
762|0
763|1 y = einsum(f, g, "bdn1,bdn1->bd1") + D * u
764|2 y = y * F.silu(z)
765|3
766|4 return y

```

767 The implementation of Mamba using the separable kernel formulation has less than 10 lines of  
768 Pytorch code. The Pytorch implementation of AUSSM is similar, except now we have to account for  
769 the time varying  $A$  matrix and  $B$  and  $C$  are relaxed.

```

770|1 def aussm(u, dt, chi, B, C, D, z):
771|2     """
772|3     params:
773|4         u: input Tensor (b,d,l)
774|5         dt: Delta Tensor (b,d,l)
775|6         chi: adaptivity matrix (d,n,d)
776|7         B: Tensor (n)
777|8         C: Tensor (n)
778|9         D: Tensor (d)
779|0         z: Tensor (b,d,l)
780|1     Returns:
781|2         y: (b,d,l)
782|3     """
783|4     A = einsum(chi, u, "dnr,blr->bldn")
784|5     A = einsum(dt, A, "bd1,bldn-bldn")
785|6     G = torch.cumsum(axis=1)
786|7
787|8     g = einsum(exp(-G), dt, B, u, "bdn1,bd1,n,bd1->bdn1")
788|9     g = torch.cumsum(g, axis=-1)
789|0     f = einsum(C, exp(G), "n,bdn1->bdn1")
790|1
791|2     y = einsum(f, g, "bdn1,bdn1->bd1") + D * u
792|3     y = y * F.silu(z)
793|4
794|5     return y

```

795 In this implementation, Mamba and Pytorch have the identical memory and time complexity as the  
796 hidden state is realized for both, albeit at only a fraction of the cost.

## 797 F.2 CUDA Kernel

798 In pure Pytorch, the additional complexity of realizing the hidden state is unavoidable, even though  
799 the computation does not have quadratic memory costs. The additional complexity of realizing  
800 the hidden state can be avoided by creating a CUDA kernel for the AUSSM equation. We use the  
801 following equation for the AUSSM, which we introduced in the main text:

$$y_{ti} = \Re \left[ \sum_{k \leq t} \sum_j C_j \exp \left( \mathbf{i} \sum_{l \leq t} \theta_{A_{lij}} \right) \frac{\Delta_{ki} B_j}{\exp \left( \mathbf{i} \sum_{l \leq k} \theta_{A_{lij}} \right)} u_i(k) \right]. \quad (33)$$

802 Each thread of the CUDA implementation computes the array inside the nested summation, which  
803 results in  $O(L)$  memory requirement for storing each of the variables ( $A, f, g$ ) for the forward pass.  
804 These variables are not realized at the same time in the GPU memory, but in registers within the  
805 streaming multiprocessors (SM) each processor holding 4 to 16 items of each array. For the 2080Ti  
806 GPU we ran the CUDA kernel on, the allowable maximum sequence length that can be processed by  
807 the kernel was 2048 after which the register and shared memory costs start to show up. We found  
808 that this sequence length is ideal for the hardware and tasks we tested on. The separable convolution  
809 trick is not restricted by the hardware, and can scale well for GPUs that can be released in the future  
810 with larger register and shared memory resources.

Table 4: **Best Hyperparameters.**

Task Group	Task	Layers	d	n	weight decay	learning rate
Algorithmic	repetition	ma	64	32	0.0	0.01
	bucket sort	am	64	32	0.0	0.01
	majority count	ma	64	32	0.1	0.01
	majority	ma	64	32	0.1	0.01
	solve equation	ma	64	32	0.0	0.01
	mod arith	am	16	8	0.0	0.01
	mod arith wo bra	ma	8	16	0.0	0.01
	cycle nav	ma	16	8	0.0	0.01
	parity	ma	16	8	0.0	0.01
Timeseries Classification	Heartbeat	ma	64	64	0.0	0.0001
	SCP1	amma	16	128	0.0	0.001
	SCP2	ma	16	128	0.0	0.0001
	Ethanol	ammama	16	64	0.001	0.00001
	Motor	ma	16	128	0.0	0.0001
	Worms	amma	16	16	0.0	0.001
Timeseries Regression	weather	ma	16	128	0.0	0.001

**Backpropagation:** For the CUDA kernel, we implemented a custom backpropagation operation. Implementing backpropagation requires the variables computed during the storage to be stored, which creates issues because the reason we are writing the CUDA kernel is so that we do not have to realize the memory-intensive hidden state. We therefore recompute the forward pass during backpropagation. The low complexity of implementing the AUSSM in CUDA means the recomputation does not incur a heavy penalty.

## G Experiments

We conduct three sets of experiments - (1) to evaluate the time/memory complexities of the different AUSSM implementations, (2) to evaluate the performance of AUSSM in algorithmic tasks enabling insights into the expressive power, and (3) to evaluate real-world performance implications in a range of long time series benchmarks. For each of the tasks involving training models (2 and 3), we perform two pipeline processes to obtain the final test accuracies. The first pipeline is the training and model selection pipeline with only the training and validation sets that are preselected based on the same criteria prior literature used. The second pipeline is the test pipeline and is entirely separate and performed starting 10 days prior to paper submission to avoid model selection based on the test results. The classification tasks are evaluated using the scaled test accuracy metric, where the obtained accuracy values are scaled with respect to baseline performance of a uniform random distribution as shown below.

$$\text{scaled accuracy score} = \frac{\text{test accuracy score} - \text{baseline accuracy score}}{1 - \text{baseline accuracy score}}$$

All the models were run in a supercomputing cluster, where we used 40 2080Ti GPUs for all except the dataset Eigenworms dataset that required higher memory. This is the lowest GPU available in the cluster with at least a CUDA compute of 7.5 required to run the mamba and AUSSM CUDA kernels. For larger memory Eigenworms workload, we used the L4 GPU which has a VRAM of 23GB. Higher VRAM GPUs were available in the cluster, but they were in high demand and unnecessary as our optimized CUDA kernel was able to handle even the large scale tasks in modest hardware.

### G.1 Scalability Evaluation

To evaluate scalability in a fair manner, we report only the time spent in computations, ignoring the latencies associated with moving variables between GPU and CPU. This provides a fair evaluation of the algorithmic performance. 5 runs are used to warm up the GPU before starting the evaluation

828 to remove transient start-up effects. The run-time values are averaged over 50 runs, where each run  
829 computes a forward and backward pass for each of the implementations. The peak memory used  
830 during each run is also similarly recorded and averaged for each of the 50 runs.

## 831 **G.2 Algorithmic Tasks**

832 For algorithmic tasks, we used the results from [32] for comparing against baseline models. We used  
833 a grid search for hyperparameter tuning with a grid search over  $d \in \{8, 16, 32, 64\}$ ,  $n \in \{8, 16, 32\}$ ,  
834 weight decay  $\in \{0.0, 0.001, 0.01\}$ , learning rate in  $\{0.0001, 0.001, 0.01\}$  and 5 seeds. The batch  
835 size was fixed at 256. For pure AUSSM blocks, we tested networks with a depth of 2, 4, and 6. For  
836 hybrid AUSSM blocks, we tested all possible 2 block configurations of Mamba (represented as m)  
837 and AUSSM blocks (represented as a) -  $\{ma, am, mm, aa\}$ . For each of the evaluated algorithmic tasks,  
838 we randomly sampled 10000 samples from a train set up to 40 length sequences. The validation set  
839 is sampled independently from 40-256 sequence lengths and had 1,000 samples. The test set had  
840 10,000 samples from sequences of up to 256 sequence lengths.

841 The tasks use the same vocabulary size and configuration used in [32]. Some samples from the tasks  
842 are shown below as a timeline. Here, the mask is applied on the output to determine the output of  
843 interest for computing the loss and output.

844	1	Task: repetition											
845	2												
846	3	time	0	1	2	3	4	5	6	7	8	9	10
847	4												
848	5	input	3	5	0	7	3	ACT	3	5	0	7	3
849	6	output	5	0	7	3	ACT	3	5	0	7	3	PAD
850	7	mask	0	0	0	0	0	1	1	1	1	1	0
851	8												
852	9	Task: bucketsort											
853	0												
854	1												
855	2	time	0	1	2	3	4	5	6	7	8	9	10
856	3												
857	4	input	3	5	0	7	3	ACT	0	3	3	5	7
858	5	output	5	0	7	3	ACT	0	3	3	5	7	PAD
859	6	mask	0	0	0	0	0	1	1	1	1	1	0
860	7												
861	8	Task: modulararithmeticwobracess											
862	9												
863	0												
864	1	time	0	1	2	3	4	5	6	7	8	9	10
865	2												
866	3	input	0	*	2	-	6	-	7	-	0	=	5
867	4	output	*	2	-	6	-	7	-	0	=	5	PAD
868	5	mask	0	0	0	0	0	0	0	0	0	1	0
869	6												
870	7	Task: cyclenav											
871	8												
872	9												
873	0	time	0	1	2	3	4	5	6	7	8	9	10
874	1												
875	2	input	+1	STAY	+1	-1	+1	-1	-1	-1	+1	0	PAD
876	3	output	STAY	+1	-1	+1	-1	-1	-1	+1	0	PAD	PAD
877	4	mask	0	0	0	0	0	0	0	0	1	0	0
878	5												
879	6	Task: modulararithmetic											
880	7												
881	8												
882	9	time	0	1	2	3	4	5	6	7	8	9	10
883	0												
884	1	input	(	(	3	-	3	)	-	4	)	=	3
885	2	output	(	3	-	3	)	-	4	)	=	3	PAD
886	3	mask	0	0	0	0	0	0	0	0	0	1	0
887	4												
888	5	Task: solveequation											
889	6												
890	7												
891	8	time	0	1	2	3	4	5	6	7	8	9	10
892	9												
893	0	input	x	=	(	2	+	1	)	ACT	3	PAD	PAD
894	1	output	=	(	2	+	1	)	ACT	3	PAD	PAD	PAD
895	2	mask	0	0	0	0	0	0	0	1	0	0	0
896	3												
897	4	Task: parity											
898	5												
899	6												
900	7	time	0	1	2	3	4	5	6	7	8	9	10
901	8												
902	9	input	1	1	0	1	1	1	1	1	1	1	0
903	0	output	1	0	1	1	1	1	1	1	1	0	0
904	1	mask	0	0	0	0	0	0	0	0	0	0	1
905	2												
906	3	Task: majoritycount											
907	4												
908	5												
909	6	time	0	1	2	3	4	5	6	7	8	9	10
910	7												
911	8	input	45	56	51	43	51	34	10	46	54	44	56
912	9	output	56	51	43	51	34	10	46	54	44	56	2
913	0	mask	0	0	0	0	0	0	0	0	0	0	1
914	1												
915	2	Task: majority											
916	3												
917	4												
918	5	time	0	1	2	3	4	5	6	7	8	9	10
919	6												
920	7	input	45	56	51	43	51	34	10	46	54	44	56
921	8	output	56	51	43	51	34	10	46	54	44	56	51
922	9	mask	0	0	0	0	0	0	0	0	0	0	1
923	0												
924	1	Task: set											
925	2												
926	3												
927	4	time	0	1	2	3	4	5	6	7	8	9	10
928	5												
929	6	input	3	5	0	7	3	ACT	0	3	5	7	PAD
930	7	output	5	0	7	3	ACT	0	3	5	7	PAD	PAD
931	8	mask	0	0	0	0	0	1	1	1	1	0	0
932	9												

### 933 **G.3 Time Series benchmark**

934 For time series classification and regression benchmarks, we follow the train-validation protocol for  
935 model selection following prior works on the benchmark. For testing, we modified the procedure as  
936 the 5 arbitrary random seeds used to evaluate test performance in prior works may introduce unwanted  
937 biases due to the low number of random samples. Also, prior works used `jax` for implementations  
938 while we used `Pytorch` and the random seed does not create the same train-validation-test sets due to  
939 differences in the Pseudorandom number generators. We thus decided to evaluate on train-validation-  
940 test splits created with 20 different seeds. We anticipated that the higher samples help in providing  
941 a better estimation of the test accuracy than what the 5 arbitrary seeds provide. For each task, we  
942 performed a hyperparameter search over the following grid:  $d \in \{16, 64, 128\}$ ,  $n \in \{16, 64, 128\}$ ,  
943 learning rate  $\in \{0.00001, 0.0001, 0.001\}$  and 5 different seeds for model selection. The model  
944 hyperparameters with the highest mean validation accuracy is chosen for evaluation in the test set.