# State Space Models for Event Cameras

Supplementary Material

## Contents:

## 1. Explanation of S4(D) and S5 models

The Structured State Spaces (S4 and its diagonal variant S4D [7, 9]) and Simplified State Space (S5) [12] models are advanced approaches in sequence modeling, each with its unique characteristics.

S4 introduces a sequence model based on the Structured State Space Model (SSM) [4, 5, 8]. It addresses computational bottlenecks in previous work [5] by reparameterizing structured state matrices to maintain a hidden state that encodes the long history of input. S4D [9] is a variant of the S4 model, simplifying it by using a fully diagonal state matrix. This adaptation preserves the performance of the original model but with a simpler implementation. Gupta [9] observed that removing the low-rank part of S4's HiPPO-LegS matrix [4] results in a diagonal matrix (referred to as the normal-HiPPO matrix) with comparable performance to the original S4. S5 [12] is an evolution of S4, using a multi-input, multi-output (MIMO) single SSM instead of multiple single-input, single-output (SISO) SSMs used in S4 [7].

The HiPPO matrix [4], a non-normal matrix, is decomposed as a sum of a normal and a low-rank matrix. S4 applies new techniques to overcome computational limitations associated with this decomposition. The S5 model simplifies the S4 layer by replacing the bank of many independent SISO SSMs with one MIMO SSM and implementing efficient parallel scans [1]. This shift eliminates the need for convolutional and frequency-domain approaches, making the model purely recurrent and time-domain based.

S4 equates the diagonal matrix case to the computation of a Cauchy kernel, applying to any matrix that can be decomposed as Normal Plus Low-Rank (NPLR). S5 utilizes a diagonal state matrix for efficient computation using parallel scans. It inherits HiPPO initialization schemes from S4, using a diagonal approximation to the HiPPO matrix for comparable performance.

S5 matches the computational complexity of S4 for both online generation and offline recurrence. It handles time-varying SSMs and irregularly sampled observations, which are challenges for the convolutional implementation in S4. S5 has been shown to match or outperform S4 in various long-range sequence modeling tasks, including speech classification and 1-D image classification [12]. This design opens up new possibilities in deep sequence modeling, including handling time-varying SSMs and combining state space layers with attention mechanisms for enhanced performance, which is our contribution along with specific problem and architecture design.

In summary, while S4 introduced a novel reparameterization of state space models for efficient long sequence modeling, S5 builds upon this by simplifying the model structure and computation, leading to a more usable and potentially more flexible approach for various sequence modeling tasks. Both S4 and S5 models can be seen on Figure 1 and Figure 2 respectively.

## 2. Initialization of continuous-time matrices

### 2.1. State Matrix Initialization

This subsection elaborates on the initialization process of continuous-time matrices which are crucial since they allow us to discretize them with different time steps to deploy our model at higher inference frequencies. As described by [4], S4's capacity to handle long-range dependencies stems from employing the HiPPO-LegS matrix, which decomposes the input considering an infinitely long, exponentially diminishing measure [5, 7]. The HiPPO-LegS matrix and its corresponding single-input-single-output (SISO) vector are defined as:

$$(\mathbf{A}_{\text{LegS}})_{nk} = -\begin{cases} (2n+1)^{1/2}(2k+1)^{1/2}, & n > k \\ n+1, & n = k \\ 0, & n < k \end{cases}. \tag{1}$$

$$(\mathbf{b}_{\text{LegS}})_n = (2n+1)^{\frac{1}{2}}. \tag{2}$$

Here, the input matrix $\mathbf{B}_{\text{LegS}} \in \mathbb{R}^{N\times H}$ is constructed by concatenating $\mathbf{b}_{\text{LegS}} \in \mathbb{R}^{N}$ $H$ times.

Theorem 1 of Gu et al. [5] establishes that HiPPO matrices from [4], $\mathbf{A}_{\text{HiPPO}} \in \mathbb{R}^{N\times N}$, can be represented in a normal plus low-rank (NPLR) format, comprising a normal matrix, $\mathbf{A}^{\text{Normal}}_{\text{HiPPO}} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^* \in \mathbb{R}^{N\times N}$, and a low-rank component:

$$\mathbf{A}_{\text{HiPPO}} = \mathbf{A}^{\text{Normal}}_{\text{HiPPO}} - \mathbf{P}\mathbf{Q}^\top = \tag{3}$$

$$\mathbf{V}\left(\mathbf{\Lambda} - (\mathbf{V}^*\mathbf{P})(\mathbf{V}^*\mathbf{Q})^*\right)\mathbf{V}^* \tag{4}$$

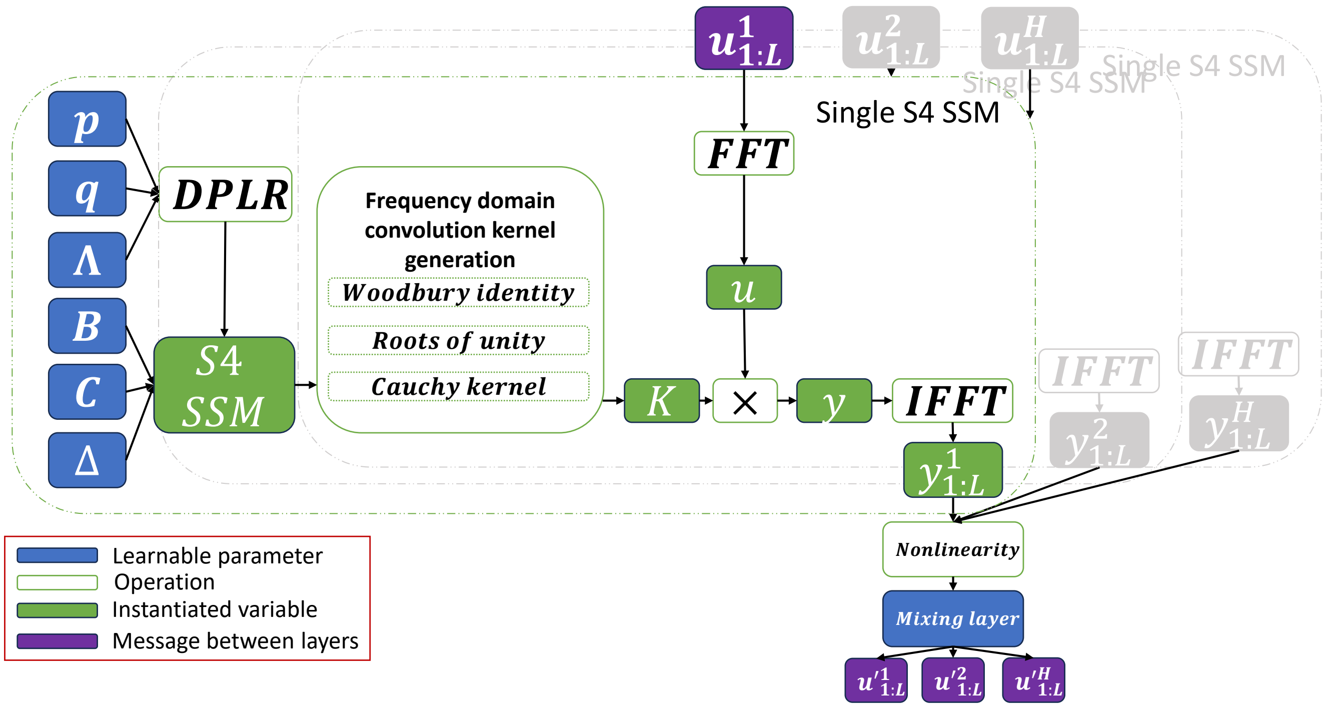


Figure 1. In the S4 layer, each dimension of the input sequence $u_{1:L} \in \mathbb{R}^{L \times H}$ is processed by a separate SSM. This process involves using a Cauchy kernel to determine the coefficients for frequency domain convolutions. The convolutions, done via FFTs, generate the output $y_{1:L} \in \mathbb{R}^{L \times H}$ for each SSM. The outputs then go through a nonlinear activation function, which includes a layer that mixes them to produce the final output of the layer.

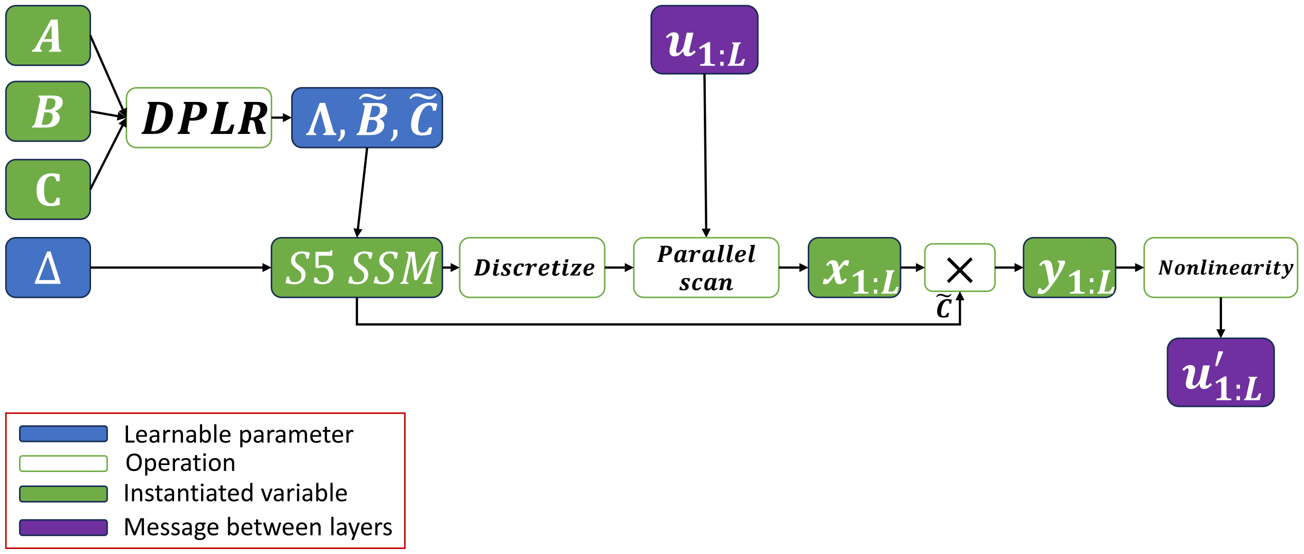


Figure 2. For the S5 layer, a parallel scan technique is employed on a diagonal linear SSM to get the SSM outputs. This approach bypasses the need for frequency domain operations and convolution kernel computations required by S4, resulting in a model that functions in a purely time-domain, recurrent manner. Recurrence is parallelized with the employment of parallel scans [1].

with unitary $\mathbf{V} \in \mathbb{C}^{N \times N}$, diagonal $\mathbf{\Lambda} \in \mathbb{C}^{N \times N}$, and low-rank factors $\mathbf{P}, \mathbf{Q} \in \mathbb{R}^{N \times r}$. This equation reveals the conjugation of HiPPO matrices into a diagonal plus low-rank (DPLR) structure. Therefore, the HiPPO-LegS matrix can be expressed through the normal HiPPO-N matrix and the low-rank term $\mathbf{P}_{\text{LegS}} \in \mathbb{R}^N$, as suggested by Goel et al. [3]:

$$\mathbf{A}_{\text{LegS}} = \mathbf{A}_{\text{LegS}}^{\text{Normal}} - \mathbf{P}_{\text{Legs}}\mathbf{P}_{\text{Legs}}^{\top} \tag{5}$$

where

$$\mathbf{A}_{\text{LegS}_{nk}}^{\text{Normal}} = -\begin{cases} (n+\frac{1}{2})^{1/2}(k+\frac{1}{2})^{1/2}, & n > k \\ \frac{1}{2}, & n = k \\ (n+\frac{1}{2})^{1/2}(k+\frac{1}{2})^{1/2}, & n < k \end{cases}. \tag{6}$$

$$\mathbf{P}_{\text{Legs}_n} = (n+\frac{1}{2})^{\frac{1}{2}} \tag{7}$$

We default to initializing the S5 layer state matrix $\mathbf{A}$ as $\mathbf{A}_{\text{LegS}}^{\text{Normal}} \in \mathbb{R}^{P\times P}$ and then decompose it to obtain the initial $\mathbf{\Lambda}$. We often find benefits in initializing $\tilde{\mathbf{B}}$ and $\tilde{\mathbf{C}}$ using $\mathbf{V}$ and its inverse, as detailed below.

Performance improvements on various tasks were noted with the S5 state matrix initialized as block-diagonal [12], with each diagonal block equaling $\mathbf{A}_{\text{LegS}}^{\text{Normal}} \in \mathbb{R}^{R\times R}$ (where $R < P$). This initialization process also involves decomposing the matrix to obtain $\mathbf{\Lambda}$, as well as $\tilde{\mathbf{B}}$ and $\tilde{\mathbf{C}}$. Even in this case, $\tilde{\mathbf{B}}$ and $\tilde{\mathbf{C}}$ remain densely initialized without constraints to keep $\mathbf{A}$ block-diagonal during learning. The hyperparameter $J$ [12] denotes the number of HiPPO-N blocks on the diagonal for initialization, with $J = 1$ indicating the default single HiPPO-N matrix initialization. Further discussion on the rationale behind this block-diagonal approach is in Appendix 6.3.

### 2.2. Input, Output, and Feed-through Matrices Initialization

The input matrix $\tilde{\mathbf{B}}$ and output matrix $\tilde{\mathbf{C}}$ are explicitly initialized using the eigenvectors from the diagonalization of the initial state matrix. We start by sampling $\mathbf{B}$ and $\mathbf{C}$ and then set the complex learnable parameters $\tilde{\mathbf{B}} = \mathbf{V}^{-1}\mathbf{B}$ and $\tilde{\mathbf{C}} = \mathbf{C}\mathbf{V}$.

The feed-through matrix $\mathbf{D} \in \mathbb{R}^H$ is initialized by sampling each element independently from a standard normal distribution.

### 2.3. Timescale Initialization

Prior research, notably [9] and [8], has underscored the significance of initializing timescale parameters. Explored in detail by Gu et al. [8], we align with S4 practices and initialize each element of $\log \mathbf{\Delta} \in \mathbb{R}^P$ from a uniform distribution over the interval $[\log \delta_{\min}, \log \delta_{\max})$, with default values set to $\delta_{\min} = 0.001$ and $\delta_{\max} = 0.1$.

## 3. Background on Parallel Scans for Linear Recurrences

This section provides a concise introduction to parallel scans, which are central element for the parallelization of the S5 method implemented in 5.1. For an in-depth exploration, refer to [1].

In essence, a scan operation takes a binary associative operator $\bullet$, adhering to the property $(a\bullet b)\bullet c = a\bullet(b\bullet c)$, and a sequence of $L$ elements $[a_1, a_2, ..., a_L]$. The output is the cumulative sequence: $[a_1,\ (a_1 \bullet a_2),\ ...,\ (a_1 \bullet a_2 \bullet ... \bullet a_L)]$.

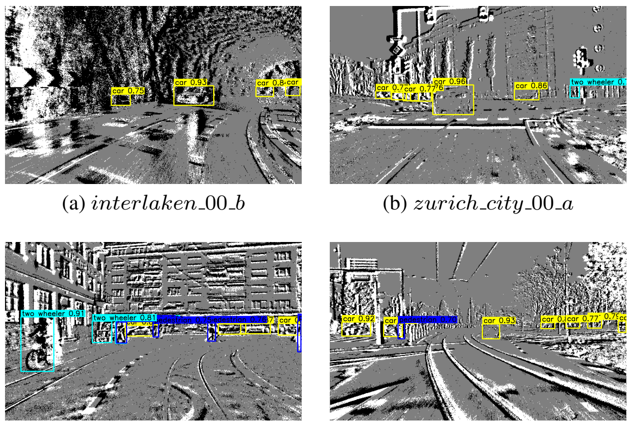
The principle behind parallel scans is leveraging the flexible computation order of associative operators. For linear recurrences, parallel scans are applicable on the following state update equations:

$$\mathbf{x}_k = \overline{\mathbf{A}}\mathbf{x}_{k-1} + \overline{\mathbf{B}}\mathbf{u}_k, \qquad \mathbf{y}_k = \overline{\mathbf{C}}\mathbf{x}_k + \overline{\mathbf{D}}\mathbf{u}_k \tag{8}$$

We initiate the process with scan tuples $c_k = (c_{k,a}, c_{k,b}) := (\overline{\mathbf{A}},\ \overline{\mathbf{B}}\mathbf{u}_k)$. The binary associative operator, applied to tuples $q_i, q_j$ (initial or intermediate), generates a new tuple $q_i \bullet q_j := (q_{j,a} \odot q_{i,a},\ q_{j,a} \otimes q_{i,b} + q_{j,b})$, where $\odot$ and $\otimes$ represent matrix-matrix and matrix-vector multiplication, respectively. With adequate processing power, this parallel scan approach can compute the linear recurrence of (8) in $O(\log L)$ sequential steps, thereby significantly enhancing computational efficiency [1].

## 4. DSEC dataset evaluation

To test the generalizability of our model, we perform inference on the DSEC dataset [2] with a model trained on 1 Mpx dataset [11]. Some results are shown below:



(a) *interlaken_00_b*

(b) *zurich_city_00_a*

(c) *zurich_city_00_b*

(d) *zurich_city_07_a*

Figure 3. Detections on DSEC dataset [2] with model trained on 1 Mpx dataset [11]. Names of the specific DSEC scenes are in the subcaptions.

## 5. PyTorch implementation of Parallel Scan and S5 model

In the following subsections, we give implementations of parallel scans and S5 layer in PyTorch.

### 5.1. Parallel Scan Operation

Listing 1. Implementation of efficient parallel scan used in S5 model.

```python
from typing import Callable
import torch
from torch.utils._pytree import tree_flatten, tree_unflatten
from functools import partial

def safe_map(f, *args):
    args = list(map(list, args))
    n = len(args[0])
    return list(map(f, *args))

def combine(tree, operator, a_flat, b_flat):
    a = tree_unflatten(a_flat, tree)
    b = tree_unflatten(b_flat, tree)
    c = operator(a, b)
    c_flat, _ = tree_flatten(c)
    return c_flat

def _scan(tree, operator, elems, axis: int):
    num_elems = elems[0].shape[axis]

    if num_elems < 2:
        return elems

    reduced_elems = combine(tree, operator,
        [torch.ops.aten.slice(elem, axis, 0, -1, 2) for elem in elems],
        [torch.ops.aten.slice(elem, axis, 1, None, 2) for elem in elems],
    )

    # Recursively compute scan for partially reduced tensors.
    odd_elems = _scan(tree, operator, reduced_elems, axis)

    if num_elems % 2 == 0:
        even_elems = combine(tree, operator,
            [torch.ops.aten.slice(e, axis, 0, -1) for e in odd_elems],
            [torch.ops.aten.slice(e, axis, 2, None, 2) for e in elems],)
    else:
        even_elems = combine(tree, operator, odd_elems,
            [torch.ops.aten.slice(e, axis, 2, None, 2) for e in elems],)

    # The first element of a scan is the same as the first element of the original 'elems'.
    even_elems = [
        torch.cat([torch.ops.aten.slice(elem, axis, 0, 1), result], dim=axis)
        if result.shape.numel() > 0 and elem.shape[axis] > 0
        else result
        if result.shape.numel() > 0
        else torch.ops.aten.slice(
            elem, axis, 0, 1
        )  # Jax allows/ignores concat with 0-dim, Pytorch does not
        for (elem, result) in zip(elems, even_elems)
    ]
    return list(safe_map(partial(_interleave, axis=axis), even_elems, odd_elems))

def associative_scan(operator: Callable, elems, axis: int = 0, reverse: bool = False):
    elems_flat, tree = tree_flatten(elems)

    if reverse:
        elems_flat = [torch.flip(elem, [axis]) for elem in elems_flat]
    num_elems = int(elems_flat[0].shape[axis])
    scans = _scan(tree, operator, elems_flat, axis)

    if reverse:
        scans = [torch.flip(scanned, [axis]) for scanned in scans]

    return tree_unflatten(scans, tree)
```

## 5.2. S5 model

Listing 2. PyTorch implementation to apply a single S5 layer to a batch of input sequences.

```python
import torch
from typing import Tuple

def discretize_bilinear(Lambda, B_tilde, Delta):
    """ Discretize a diagonalized, continuous-time linear SSM
        using bilinear transform method.
        Args:
            Lambda (float32): diagonal state matrix              (P, 2)
            B_tilde (complex64): input matrix                    (P, H)
            Delta (float32): discretization step sizes           (P,)
        Returns:
            discretized Lambda_bar (float32), B_bar (float32)  (P, 2), (P, H, 2)
    """
    Lambda = torch.view_as_complex(Lambda)

    Identity = torch.ones(Lambda.shape[0], device=Lambda.device)
    BL = 1 / (Identity - (Delta / 2.0) * Lambda)
    Lambda_bar = BL * (Identity + (Delta / 2.0) * Lambda)
    B_bar = (BL * Delta)[..., None] * B_tilde

    Lambda_bar = torch.view_as_real(Lambda_bar)
    B_bar = torch.view_as_real(B_bar)

    return Lambda_bar, B_bar

@torch.jit.script
def binary_operator(
    q_i: Tuple[torch.Tensor, torch.Tensor], q_j: Tuple[torch.Tensor, torch.Tensor]
):
    """Binary operator for parallel scan of linear recurrence. Assumes a diagonal matrix A.
    Args:
        q_i: tuple containing A_i and Bu_i at position i       (P,), (P,)
        q_j: tuple containing A_j and Bu_j at position j       (P,), (P,)
    Returns:
        new element ( A_out, Bu_out )
    """
    A_i, b_i = q_i
    A_j, b_j = q_j

    return A_j * A_i, torch.addcmul(b_j, A_j, b_i)

def apply_ssm(
    Lambda_bars: torch.Tensor, B_bars, C_tilde, D, input_sequence, prev_state, bidir: bool = False
):
    """ Compute the LxH output of discretized SSM given an LxH input.
        Args:
            Lambda_bars (float32): discretized diagonal state matrix    (P, 2)
            B_bars      (float32): discretized input matrix             (P, H, 2)
            C_tilde    (float32): output matrix                         (H, P, 2)
            input_sequence (float32): input sequence of features        (L, H)
            prev_state (complex64): hidden state                        (H,)
        Returns:
            ys (float32): the SSM outputs (S5 layer preactivations)     (L, H)
            xs (complex64): hidden state                                (H,)
    """
    B_bars, C_tilde, Lambda_bars = as_complex(B_bars), as_complex(C_tilde), as_complex(Lambda_bars)

    cinput_sequence = input_sequence.type(Lambda_bars.dtype)  # Cast to correct complex type
    Bu_elements = torch.vmap(lambda u: B_bars @ u)(cinput_sequence)

    if Lambda_bars.ndim == 1:  # Repeat for associative_scan
        Lambda_bars = Lambda_bars.tile(input_sequence.shape[0], 1)

    Lambda_bars[0] = Lambda_bars[0] * prev_state
    _, xs = associative_scan(binary_operator, (Lambda_bars, Bu_elements))

    if bidir:
        _, xs2 = associative_scan(
            binary_operator, (Lambda_bars, Bu_elements), reverse=True
        )
        xs = torch.cat((xs, xs2), axis=-1)

    Du = torch.vmap(lambda u: D * u)(input_sequence)
    return torch.vmap(lambda x: (C_tilde @ x).real)(xs) + Du, xs[-1]
```

Listing 3. Raw S5 operator that can be instantiated and used as block in any architecture.

```python
import torch

class S5(torch.nn.Module):
    def __init__(...):
        self.seq = S5SSM(...)
        ...

    def forward(self, signal, prev_state, step_scale: float | torch.Tensor = 1.0):
        if not torch.is_tensor(step_scale):
            # Duplicate across batchdim
            step_scale = torch.ones(signal.shape[0], device=signal.device) * step_scale

        return torch.vmap(lambda s, ps, ss: self.seq(s, prev_state=ps, step_scale=ss))(signal, prev_state, step_scale)

class S5SSM(torch.nn.Module):
    def __init__(...):
        self.degree = degree
        self.discretize = self.discretize_bilinear
        ...

    def get_BC_tilde(self):
        match self.bcInit:
            case "dense_columns" | "dense" | "complex_normal":
                B_tilde = as_complex(self.B)
                C_tilde = self.C
            case "factorized":
                B_tilde = self.BP @ self.BH.T
                C_tilde = self.CH.T @ self.CP
        return B_tilde, C_tilde

    def forward(self, signal, prev_state, step_scale: float | torch.Tensor = 1.0):
        B_tilde, C_tilde = self.get_BC_tilde()
        if self.degree != 1:
            assert (
                B_bar.shape[-2] == B_bar.shape[-1]
            ), "higher-order␣input␣operators␣must␣be␣full-rank"
            B_bar **= self.degree

        step = step_scale * torch.exp(self.log_step)

        Lambda_bars, B_bars = self.discretize(self.Lambda, B_tilde, step)

        return apply_ssm(Lambda_bars, B_bars, C_tilde, self.D, signal, prev_state, bidir=self.bidir)
```

## 6. Exploring the S4 and S5 Architectural Link

This section delves into the relationship between the S4 and S5 architectures, which is instrumental in the evolution of more efficient architectures and the expansion of theoretical insights from preceding research [4, 7, 12].

Our analysis is segmented into three distinct parts:

1. We utilize the linear nature of these systems to explain that the latent states generated by the S5 SSM are effectively a linear combination of those produced by the $H$ SISO S4 SSMs. Moreover, the outputs of the S5 SSM represent an additional linear transformation of these states (6.2).
2. For the SISO scenario, as $N$ becomes significantly large, the dynamics derived from a (non-diagonalizable) HiPPO-LegS matrix can be accurately approximated by the (diagonalizable) normal component of the HiPPO-LegS matrix. This is expanded to encompass the MIMO context, providing a rationale for initializing with the HiPPO-N matrix and consequently enabling efficient parallel scans (6.3).
3. We conclude with a discussion that S5, through strategic initialization of its state matrix, can emulate multiple independent S4 systems, thus easing previously held assumptions (6.3). We also discuss the set of timescale parameters, which have been observed to enhance performance (6.4).

It is important to note that many of these results are derived directly from the linearity of the recurrence.

### 6.1. Underlying Assumptions

The forthcoming sections are predicated on the following assumptions unless stated otherwise:

- We exclusively consider sequence maps that are $H$-dimensional to $H$-dimensional.
- The state matrix for each S4 SSM is the same, denoted as $\mathbf{A}^{(h)} = \mathbf{A} \in \mathbb{C}^{N\times N}$.
- We assert that the timescales for each S4 SSM are consistent, represented by $\Delta^{(h)} = \Delta \in \mathbb{R}_+$.
- S5 employs the identical state matrix $\mathbf{A}$ as in S4 [7], implying that the S5's latent size $P$ is such that $P = N$. Additionally, it is presumed that the S5 input matrix is a horizontal concatenation of the S4's column input vectors, expressed as $\mathbf{B} \triangleq \left[\mathbf{B}^{(1)} \mid \ldots \mid \mathbf{B}^{(H)}\right]$.

### 6.2. Distinct Output Projections from Equivalent Dynamics

In an S5 layer characterized by state matrix $\mathbf{A}$, input matrix $\mathbf{B}$, and an output matrix $\mathbf{C}$, and an S4 layer, where each of the $H$ individual S4 SSMs possesses a state matrix $\mathbf{A}$ and input vector $\mathbf{B}^{(h)}$, the outputs of the S5 SSM, $\mathbf{y}_k$, are equivalent to a linear combination of the latent states from the $H$ S4 SSMs, $\mathbf{y}_k = \mathbf{C}^{\text{equiv}}\mathbf{x}_k^{(1:H)}$, where $\mathbf{C}^{\text{equiv}} = [\,\mathbf{C} \cdots \mathbf{C}\,]$, provided that both layers are discretized using identical timescales.

*Proof.* Considering a singular S4 SSM, the discretized latent states in relation to the input sequence $\mathbf{u}_{1:L} \in \mathbb{R}^{L\times H}$ are described as:

$$\mathbf{x}_k^{(h)} = \sum\nolimits_{i=1}^{k} \overline{\mathbf{A}}^{k-i}\overline{\mathbf{B}}^{(h)} u_i^{(h)}. \tag{9}$$

For the S5 layer, the latent states are given by:

$$\mathbf{x}_k = \sum\nolimits_{i=1}^{k} \overline{\mathbf{A}}^{k-i}\overline{\mathbf{B}}\mathbf{u}_i, \tag{10}$$

where $\overline{\mathbf{B}}$ is defined as $\overline{\mathbf{B}} \triangleq \left[\overline{\mathbf{B}}^{(1)} \mid \ldots \mid \overline{\mathbf{B}}^{(H)}\right]$ and $\mathbf{u}_i$ is $\left[u_i^{(1)}, \ldots, u_i^{(H)}\right]^\top$.

Observation leads to:

$$\mathbf{x}_k = \sum\nolimits_{h=1}^{H} \mathbf{x}_k^{(h)}, \tag{11}$$

This outcome directly stems from the linearity of Equations (9) and (10), indicating that the MIMO S5 SSM states are equivalent to the sum of the states from the $H$ SISO S4 SSMs.

The output matrix $\mathbf{C}$ for S5 is a singular dense matrix:

$$\mathbf{y}_k = \mathbf{C}\mathbf{x}_k. \tag{12}$$

Substituting the relationship from (11) into (12) enables expressing the outputs of the MIMO S5 SSM in terms of the states of the $H$ SISO S4 SSMs:

$$\mathbf{y}_k = \mathbf{C}\sum\nolimits_{h=1}^{H} \mathbf{x}_k^{(h)} = \sum\nolimits_{h=1}^{H} \mathbf{C}\mathbf{x}_k^{(h)}. \tag{13}$$

Defining the vertical concatenation of the $H$ S4 SSM state vectors as $\mathbf{x}_k^{(1:H)} = \left[\mathbf{x}_k^{(1)\top}, \ldots, \mathbf{x}_k^{(H)\top}\right]^\top$, we ascertain that the S5 SSM outputs can be written as:

$$\mathbf{y}_k = \mathbf{C}^{\text{equiv}}\mathbf{x}_k^{(1:H)}, \quad \text{with} \quad \mathbf{C}^{\text{equiv}} = [\,\mathbf{C} \mid \cdots \mid \mathbf{C}\,], \tag{14}$$

thereby confirming their equivalence to a linear combination of the $HN$ states computed by the $H$ S4 SSMs. $\square$

This proof demonstrates that the outputs of the S5 SSM, under the specified constraints, can be interpreted as a linear combination of the latent states generated by $H$ similarly constrained S4 SSMs, sharing identical state matrices and timescale parameters. However, it does not imply that the outputs of the S5 SSM are directly identical to those of the effective block-diagonal S4 SSM; indeed, they differ, as will be further clarified in the analysis of the S4 layer.

Assuming the output vector for each S4 SSM corresponds to a row in the S5 output matrix, i.e., $\mathbf{C} =$

$\left[\mathbf{C}^{(1)\top} \mid \ldots \mid \mathbf{C}^{(H)\top}\right]^\top$, the output of each S4 SSM can be expressed as:

$$y_k^{(h)} = \mathbf{C}^{(h)} \mathbf{x}_k^{(h)}, \tag{15}$$

where $y_k^{(h)} \in \mathbb{R}$. The effective output matrix operating on the entire latent space in S4 is:

$$y_k^{(h)} = \left(\mathbf{C}^{\mathrm{S4}} \mathbf{x}_k\right)^{(h)} \tag{16}$$

By comparing (14) and (16), the distinction in the equivalent output matrices employed by both layers becomes clear:

$$\mathbf{C}^{\mathrm{S4}} = \begin{bmatrix} \mathbf{C}^{(1)} & \cdots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \mathbf{C}^{(H)} \end{bmatrix}, \tag{17}$$

$$\mathbf{C}^{\mathrm{equiv}} = \begin{bmatrix} \mathbf{C}^{(1)} & \cdots & \mathbf{C}^{(1)} \\ \vdots & \ddots & \vdots \\ \mathbf{C}^{(H)} & \cdots & \mathbf{C}^{(H)} \end{bmatrix} = [\mathbf{C} \mid \cdots \mid \mathbf{C}]. \tag{18}$$

In S4, the effective output matrix comprises independent vectors on the diagonal, while in S5, it uniformly connects dense output matrices across the $H$ S4 SSMs. Thus, S5 can be seen as defining a different projection of the $H$ independent SISO SSMs than S4 does. Both projection matrices possess an identical parameter count.

Despite variations in projection, the interpretability of the latent dynamics in S5 as a linear projection from S4's latent dynamics suggests a promising approach. This observation leads to the hypothesis that initializing the state dynamics in S5 with the HiPPO-LegS matrix, the same as in the method employed in S4 [7], may yield similarly effective results.

It remains an open question whether one method consistently surpasses the other in terms of expressiveness. It's also important to underscore that practical implementation of S4 and S5 would not directly utilize the block diagonal matrix and repeated matrix, respectively, as described in Equation 18. These matrices serve primarily as theoretical tools to explain the conceptual equivalence between S4 and S5 models.

## 6.3. Relaxing the Assumptions

Consider an instance where the S5 SSM state matrix is configured in a block-diagonal form. In such a scenario, an S5 SSM with a latent size of $JN = \mathcal{O}(H)$ would employ a block-diagonal matrix $\mathbf{A} \in \mathbb{R}^{JN \times JN}$, complemented by dense matrices $\mathbf{B} \in \mathbb{R}^{JN \times H}$ and $\mathbf{C} \in \mathbb{R}^{H \times JN}$, and $J$ distinct timescale parameters $\mathbf{\Delta} \in \mathbb{R}^J$. The latent state $\mathbf{x}_k \in \mathbb{R}^{JN}$ of this system can be divided into $J$ distinct states $\mathbf{x}_k^{(j)} \in \mathbb{R}^N$. Consequently, this allows the decomposition of the system into $J$ individual subsystems, with each subsystem discretized using a respective $\Delta^{(j)}$. The discretization can be represented as:

$$\overline{\mathbf{A}} = \begin{bmatrix} \overline{\mathbf{A}}^{(1)} & & \\ & \ddots & \\ & & \overline{\mathbf{A}}^{(J)} \end{bmatrix}, \quad \overline{\mathbf{B}} = \begin{bmatrix} \overline{\mathbf{B}}^{(1)} \\ \vdots \\ \overline{\mathbf{B}}^{(J)} \end{bmatrix}, \tag{19}$$

$$\mathbf{C} = \left[\ \mathbf{C}^{(1)} \mid \cdots \mid \mathbf{C}^{(J)}\ \right], \tag{20}$$

where $\overline{\mathbf{A}}^{(j)} \in \mathbb{R}^{N \times N}$, $\overline{\mathbf{B}}^{(j)} \in \mathbb{R}^{N \times H}$, and $\mathbf{C}^{(j)} \in \mathbb{R}^{H \times N}$. This division implies that the system can be interpreted as $J$ independent $N$-dimensional S5 SSM subsystems, with the total output being the sum of the outputs from these $J$ subsystems:

$$\mathbf{y}_k = \mathbf{C}\mathbf{x}_k = \sum_{j=1}^{J} \mathbf{C}^{(j)} \mathbf{x}_k^{(j)}. \tag{21}$$

The dynamics of each of these $J$ S5 SSM subsystems can be correlated to the dynamics of a distinct S4 system. Each of these S4 systems possesses its unique set of tied S4 SSMs, including separate state matrices, timescale parameters, and output matrices. Hence, the outputs of a $JN$-dimensional S5 SSM effectively correspond to the linear combination of the latent states from $J$ different S4 systems. This realization opens the possibility of initializing a block-diagonal S5 state matrix with multiple HiPPO-N matrices across its blocks, rather than a singular, larger HiPPO-N matrix.

## 6.4. Timescale Parameterization

This section delves into the parameterization nuances of the timescale parameters $\mathbf{\Delta}$. S4 possesses the capability to learn distinct timescale parameters for each S4 SSM [7], thereby accommodating various data timescales. Additionally, the initial setting of these timescales is crucial, as highlighted in the works of [6] and [9]. Relying solely on a single initial parameter might result in suboptimal initialization. The previous discussion in this paper proposes the learning of $J$ separate timescale parameters, corresponding to each of the $J$ subsystems. However, empirical evidence indicates superior performance when employing $P$ distinct timescale parameters [7, 12], one assigned to each state.

This approach can be interpreted in two ways. Firstly, it may be seen as assigning a unique scaling factor to each eigenvalue within the diagonalized system framework. Alternatively, it could be considered a strategy to increase the diversity of timescale parameters at the initialization phase, thereby mitigating the risks associated with inadequate initialization. It is noteworthy that the system has the potential to *learn* to operate with a singular timescale [12], achieved by equalizing all timescale values.

## 7. SSM-only and SSM-ConvNext models

To study the performance of SSM-only models, we replaced the attention block with a 2D-SSM block and trained it on Gen1 and 1 Mpx datasets. We obtain 46.1 *mAP* and 45.74 *mAP* which is lower than the original S5-ViT-B model's performance, as it can be seen in Table 1. To evaluate a CNN-based backbone, we replace the attention block with the ConvNext block [10] showing that ViT achieves better performance than the CNN structure.

| **Model** | **mAP**$_{Gen1}$ | **mAP**$_{1Mpx}$ |
|---|---|---|
| S5-ViT-B | 47.40 | 47.20 |
| S5-ConvNext-B | 45.92 | 45.66 |
| S5-SSM2D-B | 46.10 | 45.74 |

Table 1. **Comparison of mAP scores for Gen1 and 1 Mpx datasets across different base models.**

## References

[1] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990. 1, 2, 3

[2] Mathias Gehrig, Willem Aarents, Daniel Gehrig, and Davide Scaramuzza. DSEC: A stereo event camera dataset for driving scenarios. *IEEE RA-L*, 2021. 3

[3] Karan Goel, Albert Gu, Chris Donahue, and Christopher Ré. It's raw! audio generation with state-space models. *Int. Conf. Mach. Learn.*, 2022. 3

[4] Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. Hippo: Recurrent memory with optimal polynomial projections. *NeurIPS*, 33, 2020. 1, 7

[5] 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. *NeurIPS*, 34, 2021. 1

[6] Albert Gu, Karan Goel, Ankit Gupta, and Christopher Ré. On the parameterization and initialization of diagonal state space models. In *NeurIPS*, pages 35971–35983. Curran Associates, Inc., 2022. 8

[7] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *ICLR*, 2022. 1, 7, 8

[8] Albert Gu, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Ré. How to train your hippo: State space models with generalized basis projections. In *ICLR*, 2023. 1, 3

[9] Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces. In *NeurIPS*, 2022. 1, 3, 8

[10] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 1, 9

[11] Etienne Perot, Pierre de Tournemire, Davide Nitti, Jonathan Masci, and Amos Sironi. Learning to detect objects with a 1 megapixel event camera. In *NeurIPS*, pages 16639–16652. Curran Associates, Inc., 2020. 3

[12] Jimmy T.H. Smith, Andrew Warrington, and Scott Linderman. Simplified state space layers for sequence modeling. In *ICLR*, 2023. 1, 3, 7, 8