SAiDL Assignment 2025
State-Space Models

Soham Kalburgi

CS&IS, BITS Pilani, KK Birla Goa Campus
f20230460@goa.bits-pilani.ac.in

1 Introduction

Transformers have revolutionized sequence modeling but suffer from quadratic $\mathcal{O}(L^2)$ complexity, making them inefficient for long sequences. State-Space Models (SSMs) offer a promising alternative by leveraging signal processing principles to achieve subquadratic complexity while preserving expressive power.

The Selective State-Space (S4) model, introduced in *Efficiently Modeling Long Sequences with Structured State Spaces* [3], addresses this challenge with $\mathcal{O}(L \log L)$ complexity. In this report, a simplified S4 implementation in PyTorch is presented and evaluated on the sequential CIFAR-10 dataset from the *Long Range Arena (LRA)* benchmark. Additionally, alternative discretization schemes are explored to assess their impact on model performance. Most of this report follows *The Annotated S4* [4] blog.

2 Theoretical Understanding

2.1 State-Space Models

A linear time-invariant state-space model (SSM) in continuous time is given by the linear system:

$$\begin{aligned} \dot{\boldsymbol{x}}(t) &= \boldsymbol{A}x(t) + \boldsymbol{B}u(t) \\ \boldsymbol{y}(t) &= \boldsymbol{C}x(t) + \boldsymbol{D}u(t) \end{aligned} \tag{1}$$

where $\boldsymbol{x} \in \mathbb{R}^H$ is the state-space vector and $\dot{\boldsymbol{x}}$ is its time derivative, $\boldsymbol{y} \in \mathbb{R}^N$ is the output vector, $\boldsymbol{u} \in \mathbb{R}^N$ is the input signal, and $\boldsymbol{A} \in \mathbb{R}^{H\times H}$, $\boldsymbol{B} \in \mathbb{R}^{H\times N}$, $\boldsymbol{C} \in \mathbb{R}^{N\times H}$, $\boldsymbol{D} \in \mathbb{R}^{N\times N}$ are the learnable parameters of the system. The term $\boldsymbol{D}u$ can be viewed as a skip connection and can be omitted.

2.2 Discrete-time SSM

To be applied on a discrete input sequence $(u_0, u_1, \dots)$ instead of a continuous function $u(t)$, the SSM must be discretized by a step size Δ that represents the resolution of the input. Conceptually, the inputs u_k can be viewed as sampling an implicit underlying continuous signal $u(t)$, where $u_k = u(k\Delta)$.

Discretizing the continuous-time SSM using the *bilinear* method, which converts the state matrix $\boldsymbol{A}$ into an approximation $\overline{\boldsymbol{A}}$ gives:

$$\begin{aligned} \overline{\boldsymbol{A}} &= (\boldsymbol{I} - \Delta/2 \cdot \boldsymbol{A})^{-1}(\boldsymbol{I} + \Delta/2 \cdot \boldsymbol{A}) \\ \overline{\boldsymbol{B}} &= (\boldsymbol{I} - \Delta/2 \cdot \boldsymbol{A})^{-1}\Delta\boldsymbol{B} \\ \overline{\boldsymbol{C}} &= \boldsymbol{C} \end{aligned} \tag{2}$$

The discrete SSM is:

$$\begin{aligned} x_k &= \overline{\boldsymbol{A}}x_{k-1} + \overline{\boldsymbol{B}}u_k \\ y_k &= \overline{\boldsymbol{C}}x_k \end{aligned} \tag{3}$$

This equation is now a sequence-to-sequence map $u_k \mapsto y_k$ instead of function-to-function. The state equation is now a recurrence in x_k, where $x_k \in \mathbb{R}^N$ can be viewed as a *hidden state* with transition matrix $\overline{\boldsymbol{A}}$.

2.3 Convolutional Representation

The recurrent SSM in (3) can be written as a discrete convolution. Let the initial state $x_{-1} = 0$. Expanding gives:

$$\begin{aligned} x_0 &= \overline{\boldsymbol{B}}u_0 \quad x_1 = \overline{\boldsymbol{AB}}u_0 + \overline{\boldsymbol{B}}u_1 \quad x_2 = \overline{\boldsymbol{A}}^2\overline{\boldsymbol{B}}u_0 + \overline{\boldsymbol{AB}}u_1 + \overline{\boldsymbol{B}}u_2 \quad \dots \\ y_0 &= \overline{\boldsymbol{CB}}u_0 \quad y_1 = \overline{\boldsymbol{CAB}}u_0 + \overline{\boldsymbol{CB}}u_1 \quad y_2 = \overline{\boldsymbol{CA}}^2\overline{\boldsymbol{CB}}u_0 + \overline{\boldsymbol{CAB}}u_1 + \overline{\boldsymbol{CB}}u_2 \quad \dots \end{aligned} \tag{4}$$

This can be vectorized into a convolution.

$$\begin{aligned} y_k &= \overline{\boldsymbol{CA}}^k\overline{\boldsymbol{B}}u_0 + \overline{\boldsymbol{CA}}^{k-1}\overline{\boldsymbol{B}}u_1 + \dots + \overline{\boldsymbol{CAB}}u_{k-1} + \overline{\boldsymbol{CB}}u_k \\ y &= \overline{\boldsymbol{K}} * u \end{aligned} \tag{5}$$

$$\overline{\boldsymbol{K}} \in \mathbb{R}^L = (\overline{\boldsymbol{CB}}, \overline{\boldsymbol{CAB}}, \dots, \overline{\boldsymbol{CA}}^{L-1}\overline{\boldsymbol{B}}) \tag{6}$$

$\overline{\boldsymbol{K}}$ is called the SSM convolutional kernel. This filter is the size of the entire sequence thus, computing it naively, $\mathcal{O}(N^2L)$ time, $\mathcal{O}(NL)$ space, is slow and memory inefficient.

2.4 Long-Range Dependencies

A basic SSM with random initialization performs poorly in practice. The HiPPO [2] theory of continuous-time memorization specifies a certain class of matrices $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ that when incorporated into (1), allows the state $x(t)$ to memorize the history of input $u(t)$. The HiPPO matrix is defined as:

$$\boldsymbol{A}_{nk} = -\begin{cases} (2n+1)^{1/2}(2k+1)^{1/2} & \text{if } n > k \\ n+1 & \text{if } n = k \\ 0 & \text{if } n < k \end{cases} \tag{7}$$

where, the negative sign is to ensure stability via *Hurwitz condition* [1], $\mathrm{Re}(\lambda_i) < 0 \quad \forall \lambda_i \in \mathrm{eig}(A)$, i.e. all eigenvalues have negative real parts. This prevents state dynamics from diverging over time. The main takeaway is that this matrix aims to compress the past history into a state that has enough information to approximately reconstruct the history. It also specifies $\boldsymbol{B}_n = (2n+1)^{1/2}$.

2.5 Structured State-Space: S4

S4 has two main differences from a basic SSM.

1. It addresses a *modeling challenge* - long-range dependencies - by using a special formula for the $\boldsymbol{A}$ matrix, i.e., HiPPO (7).
2. It solves a *computational challenge* of SSMs by introducing a special representation and algorithm to be able to work with this matrix.

Diagonal Plus Low-Rank (DPLR) SSM A DPLR SSM is $(\boldsymbol{\Lambda} - \boldsymbol{PQ}^*, \boldsymbol{B}, \boldsymbol{C})$ for some diagonal $\boldsymbol{\Lambda}$ and matrices $\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{C}^{N\times 1}$. WLOG, rank is assumed to be 1, i.e., the matrices are vectors.

Under this DPLR assumption, S4 overcomes the speed bottleneck as follows:

1. Instead of computing $\overline{\boldsymbol{K}}$ directly, S4 computes its spectrum by evaluating its *truncated generating function.* This now involves a matrix inverse instead of power.
2. S4 shows that the diagonal matrix matrix case is equivalent to the computation of a *Cauchy kernel* $\frac{1}{\omega_j - \zeta_k}$.
3. S4 shows that the low-rank term can be corrected by applying the *Woodbury identity* which reduces $(\boldsymbol{A} + \boldsymbol{PQ}^*)$ in terms of $\boldsymbol{A}^{-1}$, truly reducing to the diagonal case.

SSM Generating Functions The *truncated SSM generating function* at node z with truncation L is:

$$\hat{\mathcal{K}}_L(z, \overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \overline{\boldsymbol{C}}) \in \mathbb{C} := \sum_{i=0}^{L-1} \overline{\boldsymbol{C}}\,\overline{\boldsymbol{A}}^i \overline{\boldsymbol{B}} z^i \tag{8}$$

The generating function essentially converts the SSM convolution filter from the time domain to frequency domain. Once the z-transform of a discrete sequence known, the filters discrete fourier transform can be obtained from evaluations of its z-transform at the roots of unity $\Omega = \{e^{2\pi \frac{k}{L}} : k \in [L]\}$. Then, inverse fourier transformation can be applied, stably in $\mathcal{O}(L \log L)$ operations by applying an FFT, to recover the filter.

In the generating function, the matrix power can be replaced with an inverse.

$$\hat{K}_L(z) = \sum_{i=0}^{L-1} \overline{\boldsymbol{C}}\,\overline{\boldsymbol{A}}^i \overline{\boldsymbol{B}} z^i = \overline{\boldsymbol{C}}(\boldsymbol{I} - \overline{\boldsymbol{A}}^L z^L)(\boldsymbol{I} - \overline{\boldsymbol{A}}z)^{-1}\overline{\boldsymbol{B}} = \widetilde{\boldsymbol{C}}(\boldsymbol{I} - \overline{\boldsymbol{A}}z)^{-1}\overline{\boldsymbol{B}} \tag{9}$$

Diagonal Plus Low-Rank Case The appendix of the paper shows:

$$\widetilde{\boldsymbol{C}}\left(\boldsymbol{I} - \overline{\boldsymbol{A}}\right)^{-1} \overline{\boldsymbol{B}} = \frac{2\Delta}{1+z} \widetilde{\boldsymbol{C}} \left[2\frac{1-z}{1+z} - \Delta \boldsymbol{A} \right]^{-1} \overline{\boldsymbol{B}} \tag{10}$$

Let $\boldsymbol{A} = \boldsymbol{\Lambda} - \boldsymbol{P}\boldsymbol{Q}^*$, where $\boldsymbol{P}, \boldsymbol{Q} \in \mathbb{C}^{N \times 1}$ is a low-rank component. The *Woodbury identity* states:

$$(\boldsymbol{\Lambda} + \boldsymbol{P}\boldsymbol{Q}^*)^{-1} = \boldsymbol{\Lambda}^{-1} - \boldsymbol{\Lambda}^{-1}\boldsymbol{P}\left(1 + \boldsymbol{Q}^*\boldsymbol{\Lambda}^{-1}\boldsymbol{P}\right)^{-1} \boldsymbol{Q}^*\boldsymbol{\Lambda}^{-1} \tag{11}$$

Using (9), (10), (11) and with some algebra on $\boldsymbol{A}$ as above:

$$\hat{\boldsymbol{K}}_{\mathrm{DPLR}}(z) = c(z)\left[k_{z,\boldsymbol{\Lambda}}(\widetilde{\boldsymbol{C}}, \boldsymbol{B}) - k_{z,\boldsymbol{\Lambda}}(\widetilde{\boldsymbol{C}}, \boldsymbol{P})\left(1 + k_{z,\boldsymbol{\Lambda}}(\boldsymbol{Q}^*, \boldsymbol{P})\right)^{-1} k_{z,\boldsymbol{\Lambda}}(\boldsymbol{Q}^*, \boldsymbol{B})\right] \tag{12}$$

where, $k_{z,\boldsymbol{\Lambda}} = \sum_i \frac{\widetilde{\boldsymbol{C}}_i \boldsymbol{B}_i}{g(z) - \boldsymbol{\Lambda}_i}$ is a *Cauchy kernel.*

Converting HiPPO to DPLR HiPPO is Normal Plus Low-Rank (NPLR). The S4 techniques can apply to any matrix $\boldsymbol{A}$ that can be decomposed as NPLR.

$$\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^* - \boldsymbol{P}\boldsymbol{Q}^\top = \boldsymbol{V}\left(\boldsymbol{\Lambda} - \boldsymbol{V}^*\boldsymbol{P}(\boldsymbol{V}^*\boldsymbol{Q})^*\right)\boldsymbol{V}^* \tag{13}$$

for unitary $\boldsymbol{V} \in \mathbb{C}^{N \times N}$, diagonal $\boldsymbol{\Lambda}$, and low-rank factorization $\boldsymbol{P}, \boldsymbol{Q} \in \mathbb{R}^{N \times r}$. An NPLR SSM is therefore unitarily equivalent to some DPLR matrix. In this case, $\boldsymbol{A}$ is the HiPPO matrix. It is first written as a normal plus low-rank term, and then diagonalized to get $\boldsymbol{\Lambda}$.

3 Implementation Details

3.1 S4 Layer

Logarithmic Step Initializer

```python
def log_step_init(tensor, dt_min=0.001, dt_max=0.1):
    scale = torch.log(torch.tensor(dt_max)) - torch.log(torch.tensor(dt_min))
    return tensor * scale + torch.log(torch.tensor(dt_min))
```

Initializes the step size in logarithmic scale, with defaults set to 0.001 and 0.1 for minimum and maximum step sizes respectively.

HiPPO Matrix

```python
def hippo(N):
    P = torch.sqrt(1 + 2 * torch.arange(1, N+1, dtype=torch.float))
    A = torch.outer(P, P)
    A = torch.tril(A) - torch.diag(torch.arange(1, N+1, dtype=torch.float))
    return A
```

Implements the HiPPO-LegS matrix as per (7), without the negative sign. It follows 1-based indexing (i.e., first term is $\boldsymbol{A}_{11}$).

DPLR Decomposition

```python
def hippo_dplr(N):
    A = -1 * hippo(N)  # -ve sign here

    # low-rank correction
    p = 0.5 * torch.sqrt(2 * torch.arange(1, N+1, dtype=torch.float32) + 1.0)
    p = p.to(torch.complex64)

    # P = Q
    Ap = A.to(torch.complex64) + torch.outer(p, p)

    # eigenvalues, eigenvectors
    lambda_, V = torch.linalg.eig(Ap)

    return lambda_, p, V

def p_lambda(n):
    lambda_, p, V = hippo_dplr(n)
    Vc = V.conj().T
    p = Vc @ p
    return [p, lambda_]
```

Creates the DPLR decomposition of the HiPPO matrix (13). The `p_lambda` function projects the low-rank term into eigenbasis. $\boldsymbol{P} = \boldsymbol{Q}$ is important for stability.

Cauchy Kernel

```python
def cauchy_kernel(v, omega, lambda_):
    if v.ndim == 1:
        v = v.unsqueeze(0).unsqueeze(0)
    elif v.ndim == 2:
        v = v.unsqueeze(1)
    return (v/(omega-lambda_)).sum(dim=-1)
```

This is a helper function for $k_{z,\boldsymbol{\Lambda}}$ (12).

Causal Convolution

```python
def causal_convolution(u, K):
    l_max = u.shape[1]  # u.shape = [batch, seq_length, d_model]

    # pad with zeros
    ud = rfft(F.pad(u.float(), pad=(0, 0, 0, l_max, 0, 0)), dim=1)
    Kd = rfft(F.pad(K.float(), pad=(0, l_max)), dim=-1)

    # inverse fft
    return irfft(ud.transpose(-2, -1)*Kd)[..., :l_max].transpose(-2, -1).type_as(u)
```

Function to compute the non-circular convolution (5, 6) using the convolution theorem with fast fourier transform (FFT). The discrete convolution theorem for circular convolutions efficiently calculates the output of the convolution by first multiplying FFTs of the input sequences and then applying inverse FFT. To use this theorem for non-circular convolutions, the input sequences are padded with zeros, and the output sequence is then unpadded. As the length gets longer this FFT method is more efficient than the direct convolution.

Compute Frequencies

```python
def f_omega(l_max, dtype=torch.complex64):
    return torch.arange(l_max).type(dtype).mul(2j * torch.tensor(torch.pi) / l_max).exp()
```

Layer Class

```python
class S4Layer(nn.Module):
    def __init__(self, d_model, n, l_max):
        # ... init parameters...

        # buffers omega, ifft_order

        # init B as sqrt(2n+1)
        # init C with xavier normal
        # init D with ones

    # algorithm 1
    def roots(self):
        a0 = self.Ct.conj()
        a1 = self.p.conj()
        b0 = self.B
        b1 = self.p
        step = self.log_step.exp()

        # bilinear discretization
        g = torch.outer(2.0/step, (1.0-self.omega)/(1.0+self.omega))
        c = 2.0/(1.0+self.omega)

        k00 = cauchy_kernel(a0*b0, g.unsqueeze(-1), self.lambda_)
        k01 = cauchy_kernel(a0*b1, g.unsqueeze(-1), self.lambda_)
        k10 = cauchy_kernel(a1*b0, g.unsqueeze(-1), self.lambda_)
        k11 = cauchy_kernel(a1*b1, g.unsqueeze(-1), self.lambda_)
        return c*(k00-k01*(1.0/(1.0+k11))*k10)

    # ... properties ...

    @property
    def K(self):
        at_roots = self.roots()
        out = ifft(at_roots, n=self.l_max, dim=-1)
        conv = torch.stack([i[self.ifft_order] for i in out]).real
        return conv.unsqueeze(0)

    def forward(self, u):
        return causal_convolution(u, K=self.K) + (self.D * u)
```

The `roots` function computes $\hat{\boldsymbol{K}}$ as per line 3 of Algorithm 1 in the original paper (also (12)). It uses bilinear discretization. Other discretizations are explored in subsection 3.4. The layer also implements the skip connection $\boldsymbol{D}u$.

3.2 S4 Block

The `S4Block` class implements a residual architecture with the S4 Layer.

$$\text{LayerNorm} \to \text{S4Layer} \to \text{GELU} \to \text{Dropout} \to \text{Linear} \to \text{LayerNorm}$$

3.3 S4 Model

The `S4Model` class implements a stack of S4 Blocks for sequence processing tasks.

$$\text{Encoder} \to (n)\,\text{S4Block} \to \text{Decoder}$$

3.4 Discretization Schemes [5]

Bilinear Discretization Also called Tustin's method, already discussed in (2).

Zero-Order Hold (ZOH) Discretization

$$
\begin{aligned}
\overline{\boldsymbol{A}} &= e^{\boldsymbol{A}\Delta} \\
\overline{\boldsymbol{B}} &= \int_0^{\Delta} e^{\boldsymbol{A}\tau} \boldsymbol{B} d\tau \\
&= \boldsymbol{A}^{-1}(\overline{\boldsymbol{A}} - \boldsymbol{I})\boldsymbol{B} \\
\overline{\boldsymbol{C}} &= \boldsymbol{C}
\end{aligned}
\tag{14}
$$

Dirac Discretization

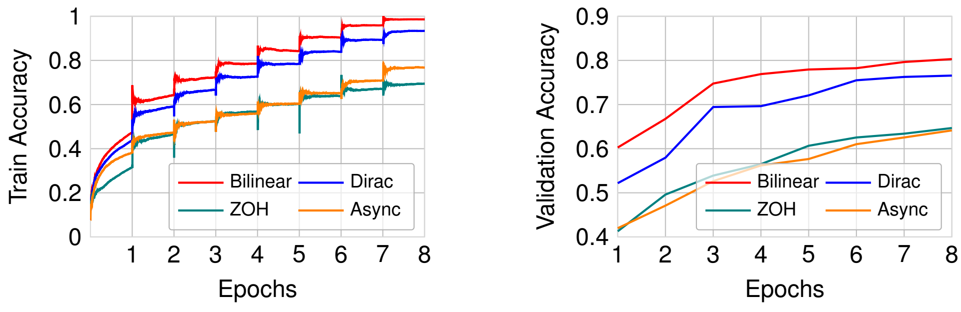
$$
\begin{aligned}
\overline{\boldsymbol{A}} &= e^{\boldsymbol{A}\Delta} \simeq \boldsymbol{I} + \boldsymbol{A}\Delta \\
\overline{\boldsymbol{B}} &= \boldsymbol{B}\Delta \\
\overline{\boldsymbol{C}} &= \boldsymbol{C}
\end{aligned}
\tag{15}
$$

Asynchronous Discretization

$$
\begin{aligned}
\overline{\boldsymbol{A}} &= e^{\boldsymbol{A}\Delta\delta_t} \\
\overline{\boldsymbol{B}} &= \int_0^{\Delta} e^{\boldsymbol{A}\tau} \boldsymbol{B} d\tau \\
&= \boldsymbol{A}^{-1}(\overline{\boldsymbol{A}} - \boldsymbol{I})\boldsymbol{B} \\
\overline{\boldsymbol{C}} &= \boldsymbol{C}
\end{aligned}
\tag{16}
$$

4 Results

All S4 models with different discretization schemes were tested on the sequential CIFAR-10 (sCIFAR-10) dataset. The training was conducted for 8 epochs, batch size 64, AdamW optimizer with weight decay 0.01, initial learning rate 0.001 with cosine learning rate annealing. For Async discretization, $\delta_t = 0.1$. The test accuracies are reported in Table 1 based on single-run evaluation.



(a) Train Accuracies (b) Validation Accuracies

Fig. 1. Training and validation accuracies of different S4 discretization schemes (Bilinear, ZOH, Dirac, Async) on the sequential CIFAR-10 dataset over 8 epochs.

Table 1. Test accuracies (%) of S4 Models with different discretization schemes.

Dataset	S4 Models			
	Bilinear	ZOH	Dirac	Async
sCIFAR-10	**80.20**	64.54	75.86	64.08

As seen in Fig 1, bilinear discretization outperforms other discretization schemes, followed by Dirac. ZOH and Async discretizations perform closely as expected.

References

1. Goel, K., Gu, A., Donahue, C., Re, C.: Its raw! Audio generation with state-space models. In: Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., Sabato, S. (eds.) Proceedings of the 39th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 162, pp. 7616–7633. PMLR (17–23 Jul 2022), https://proceedings.mlr.press/v162/goel22a.html
2. Gu, A., Dao, T., Ermon, S., Rudra, A., Ré, C.: Hippo: Recurrent memory with optimal polynomial projections. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 1474–1487. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper_files/paper/2020/file/102f0bb6efb3a6128a3c750dd16729be-Paper.pdf
3. Gu, A., Goel, K., Ré, C.: Efficiently modeling long sequences with structured state spaces (2022), https://arxiv.org/abs/2111.00396
4. Rush, A., Karamcheti, S.: The annotated s4. In: ICLR Blog Track (2022), https://iclr-blog-track.github.io/2022/03/25/annotated-s4/, https://iclr-blog-track.github.io/2022/03/25/annotated-s4/
5. Schöne, M., Sushma, N.M., Zhuge, J., Mayr, C., Subramoney, A., Kappel, D.: Scalable event-by-event processing of neuromorphic sensory signals with deep state-space models (2024), https://arxiv.org/abs/2404.18508