State-Space Models

SAiDL Assignment 2025

Rajat Soni

1 Introduction

State-space models (SSMs) have long been a staple in fields such as control theory and time series analysis. Recently, there has been a surge of interest in applying these models to deep learning tasks that require handling long-range dependencies. In this paper, we review the mathematical underpinnings of SSMs and introduce recent advances that render them both computationally efficient and capable of modeling long sequences.

2 Background and Related Work

Traditional models like RNNs, CNNs, and Transformers have been adapted for long-range dependencies but often at the cost of computational efficiency. Early attempts to use continuous SSMs faced challenges in numerical stability and memory usage. The HiPPO framework, which introduces a specialized matrix parameterization, significantly improves the memory capabilities of SSMs for long sequences. The recent S4 model further refines these ideas by reparameterizing the state matrices, allowing for efficient computations in both recurrent and convolutional forms.

3 Theoretical Foundations

3.1 Continuous-Time State Space Models

The continuous-time state space model is defined as:

$$x'(t) = Ax(t) + Bu(t), \quad y(t) = Cx(t) + Du(t). \tag{1}$$

Here, $x(t) \in \mathbb{R}^N$ is the latent state, $u(t)$ the input signal, and $y(t)$ the output signal. Equation (1) forms the backbone of SSMs used in various scientific disciplines.

3.2 HiPPO Matrix for Continuous-Time Memorization

To effectively capture long-range dependencies, the HiPPO framework modifies the state matrix A. A common choice is the HiPPO matrix:

$$A_{nk} = \begin{cases} -\sqrt{(2n+1)(2k+1)} & \text{if } n > k, \\ n+1 & \text{if } n = k, \\ 0 & \text{if } n < k, \end{cases} \tag{2}$$

which helps the system to "memorize" the history of the input $u(t)$, as demonstrated by improved performance on sequential tasks.

3.3 Discretization of the Continuous Model

For practical applications, the continuous model must be discretized. Using a bilinear (Tustin) transformation, the discrete state space model becomes:

$$\begin{aligned} x_k &= \bar{A}x_{k-1} + \bar{B}u_k, \quad \text{where} \quad \bar{A} = \left(I - \frac{\Delta}{2}A\right)^{-1}\left(I + \frac{\Delta}{2}A\right), \\ y_k &= Cx_k, \end{aligned} \tag{3}$$

with $\bar{B} = \left(I - \frac{\Delta}{2}A\right)^{-1} \Delta B$ and Δ representing the discretization step size.

3.4 Convolutional Representation

Unrolling the recurrence in Equation (3) reveals that the output can be written as a convolution:

$$y_k = \sum_{i=0}^{k} C\bar{A}^{k-i}\bar{B}\,u_i, \tag{4}$$

or more compactly,

$$y = K * u, \tag{5}$$

where the convolution kernel K is given by

$$K = \left(C\bar{B},\ C\bar{A}\bar{B},\ C\bar{A}^2\bar{B},\ \ldots,\ C\bar{A}^{L-1}\bar{B}\right). \tag{6}$$

Efficient computation of this kernel is key to applying SSMs to long sequences.

4 Methodology: Structured State Spaces (S4)

The S4 model introduces a structured parameterization to overcome the computational challenges inherent in SSMs. The core idea is to reparameterize the state matrix A using a decomposition that separates it into a normal component and a low-rank correction:

$$A = V\,(\Lambda - (V^*P)(V^*Q)^*)\,V^*, \tag{7}$$

where V is unitary, Λ is diagonal, and $P, Q \in \mathbb{C}^{N\times r}$ are low-rank factors. This decomposition (Normal Plus Low-Rank, or NPLR) allows for:

- **Efficient Recurrence:** With the reparameterization, each time-step update can be computed in $\mathcal{O}(N)$ operations.
- **Fast Convolution:** The convolution kernel in Equation (6) can be computed in nearly linear time with respect to the sequence length.

4.1 Algorithmic Sketch

An outline of the efficient convolution kernel computation in S4 is as follows:

1. Evaluate a truncated generating function associated with the SSM.
2. Apply the Woodbury identity to handle the low-rank correction efficiently.
3. Leverage fast Cauchy kernel algorithms to compute the spectrum of the convolution kernel.
4. Use an inverse FFT to obtain the final kernel.

These steps allow the S4 model to scale to sequences with tens of thousands of time steps.

5 Experiments

The S4 model has been benchmarked across a variety of tasks:

- **Long-Range Arena (LRA):** On tasks with sequences up to 16k steps, S4 outperforms Transformer-based models.
- **Raw Speech Classification:** S4 achieves high accuracy on raw audio signals without extensive pre-processing.
- **Image and Text Generation:** S4 demonstrates competitive performance and significantly faster generation times compared to conventional autoregressive models.

Empirical results indicate that the structured parameterization not only improves performance but also reduces memory usage and computational complexity.

6 Implementation Details and Proof-of-Concept

In this section, we describe two implementations that serve as practical illustrations of how to train and evaluate an S4 model on a benchmark dataset. Both implementations target the **sCIFAR-10** dataset and demonstrate how images can be reshaped into sequences for the S4 model. We do not show code here; instead, we outline the key steps and rationale behind each version.

6.1 Implementation 1: Simple Proof-of-Concept

Setup and Libraries. The process starts by importing standard Python libraries such as `torch`, `torchvision`, and `matplotlib`. A check for CUDA availability ensures that GPU acceleration is used if present. The path to the configuration files from the Long Range Arena repository (specifically for the CIFAR-10 benchmark) is added to the Python system path to facilitate the loading of baseline settings or other utilities.

Configuration. A simple configuration class is defined, specifying hyperparameters like the batch size, learning rate, and the number of training epochs. These hyperparameters guide the data loading process and the training loop.

Data Processing. The sCIFAR-10 dataset is downloaded via `torchvision`, providing a set of 3-channel, 32×32 images. Each image is flattened into a sequence of length 3072 (i.e., $3 \times 32 \times 32$), so the S4 model can treat each image as a one-dimensional sequence of tokens.

Model Architecture. A custom PyTorch `nn.Module` class (`CIFARS4Model`) is created:

- **Input Projection:** A linear layer projects each scalar token into a higher-dimensional embedding (dimension = `d_model`).
- **S4 Module:** The projected sequence is then passed to the S4 model (imported from `models.s4.s4d`), which handles long-range dependencies via its structured state-space parameterization.
- **Pooling:** Mean pooling is applied across the sequence length to collapse the time dimension into a single vector representation.
- **Classifier:** A final linear layer maps the pooled features to 10 output logits for the CIFAR-10 classes.

Training Loop. The model, loss function (cross-entropy), and optimizer (Adam) are instantiated. For each epoch:

1. Batches of images are reshaped and fed into the model.
2. The cross-entropy loss is computed, and gradients are backpropagated.
3. The optimizer updates the model parameters.
4. The model's performance is periodically evaluated on the test set, and metrics such as accuracy and average loss are logged.

This simple proof-of-concept demonstrates how to adapt an S4-based network to image classification tasks by viewing images as sequences.

6.2 Implementation 2: Discretization Flexibility

Extended Setup. Building on the first proof-of-concept, a second version of the code introduces additional flexibility in the discretization of the continuous state space model. Standard library imports and CUDA checks remain the same, but the path to the updated `models.s4.s4d_new` module is used to access extended functionality.

Discretization Methods. A new parameter (e.g., `disc`) is introduced in the S4 model class to specify the discretization technique. Common choices include `zoh` (zero-order hold), `bilinear`, `dirac`, and `async`. Each method alters how the continuous-time dynamics are discretized into their discrete-time counterparts.

Model Architecture with Discretization. Similar to Implementation 1, the `CIFARS4Model` class includes:

- **Input Projection:** Flattened images are projected into a `d_model`-dimensional space.
- **S4 Module with `disc` Parameter:** The S4 block (now from `models.s4.s4d_new`) is initialized with the chosen discretization scheme, allowing experimentation with different transformations of the continuous dynamics.
- **Pooling and Classifier:** Mean pooling and a final linear layer for classification remain the same.

Training with Multiple Schemes. A list of discretization techniques (e.g., {`zoh`, `bilinear`, `dirac`, `async`}) is iterated over. For each technique:

1. A new model instance is created with the specified `disc` parameter.
2. The model is trained on the sCIFAR-10 dataset for a given number of epochs, using the same training loop structure as in Implementation 1.
3. After each epoch, validation metrics (loss and accuracy) are recorded for comparison across discretization schemes.

This extended version enables users to systematically compare the effects of different discretization methods on performance and convergence.

7 Results

7.1 Implementation 1: Training and Test Accuracy

Figure 1 shows the training and test accuracy curves over 100 epochs. We observe that both the training and test accuracy increase steadily, ultimately converging around 60%. This indicates that the model successfully learns to classify the sCIFAR-10 images when flattened into sequences.

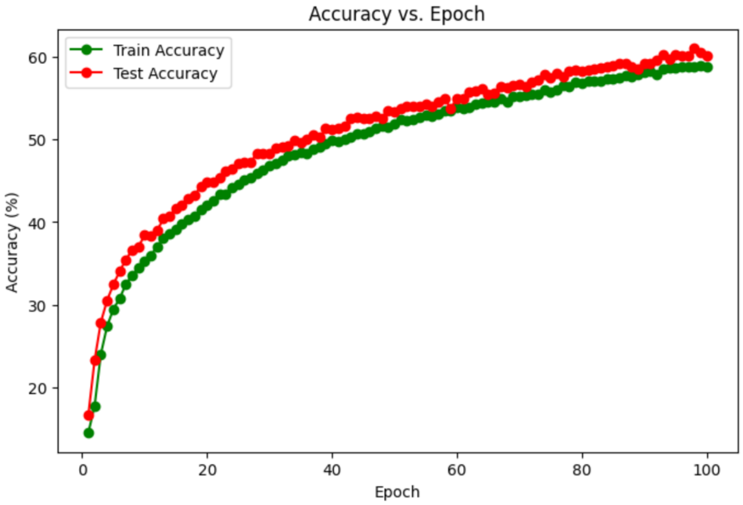

Figure 1: Training vs. Test Accuracy over 100 epochs for Implementation 1.

Figure 2 illustrates the training loss decreasing from approximately 2.2 to around 1.2. The downward trend in the loss curve reflects the model's increasing proficiency in mapping the input sequences to their correct labels.

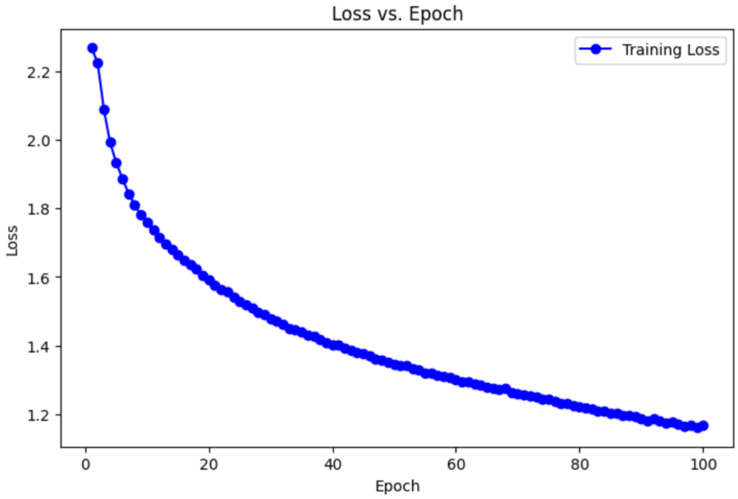

Figure 2: Training Loss vs. Epoch for Implementation 1.

7.2 Implementation 2: Discretization Comparison

Here are the implementations of the 4 different types of discretization techniques given in this paper - https://arxiv.org/abs/2404.18508. Implementations were found in this repo (click here)

```
def forward(self, L):

        dt = torch.exp(self.log_dt)  # (H)
        C = torch.view_as_complex(self.C)  # (H, N//2)
        A = -torch.exp(self.log_A_real) + 1j * self.A_imag  # (H, N//2)
        # Compute dtA = A * dt (broadcast dt over last dimension)
        dtA = A * dt.unsqueeze(-1)  # (H, N//2)
        # Create time indices tensor (assume unit time steps)
        t = torch.arange(L, device=A.device)  # (L)

        if self.disc == ’zoh’:
            # Zero-Order Hold discretization (default)
            T = dtA.unsqueeze(-1) * t.unsqueeze(0).unsqueeze(0)  # (H, N//2, L)
            C_scaled = C * (torch.exp(dtA) - 1.) / A
            K = 2 * torch.einsum(’hn, hnl -> hl’, C_scaled, torch.exp(T)).real
        elif self.disc == ’bilinear’:
            # Bilinear (Tustin) discretization
            C_scaled = C * (1. - dtA/2).reciprocal() * dt.unsqueeze(-1)
            dA = (1. + dtA/2) / (1. - dtA/2)
            E = torch.exp(dA.log().unsqueeze(-1) * t.unsqueeze(0).unsqueeze(0))
            K = 2 * torch.einsum(’hn, hnl -> hl’, C_scaled, E).real
        elif self.disc == ’dirac’:
            # Dirac discretization: use unit scaling (gamma_bar = 1)
            T = dtA.unsqueeze(-1) * t.unsqueeze(0).unsqueeze(0)
            # Do not apply the scaling factor (exp(dtA)-1)/A
            K = 2 * torch.einsum(’hn, hnl -> hl’, C, torch.exp(T)).real
        elif self.disc == ’async’:
            # Asynchronous discretization: here kept equivalent to ’zoh’
            T = dtA.unsqueeze(-1) * t.unsqueeze(0).unsqueeze(0)
            C_scaled = C * (torch.exp(dtA) - 1.) / A
            K = 2 * torch.einsum(’hn, hnl -> hl’, C_scaled, torch.exp(T)).real
        else:
            raise ValueError(f"Unknown discretization method: {self.disc}")

        return K
```

Figure 3 shows the test accuracy for four different discretization schemes (`zoh`, `bilinear`, `dirac`, and `async`) over 50 epochs. We can see that all schemes steadily improve accuracy, but `dirac` (green) tends to outperform the other three, eventually surpassing 55% test accuracy.

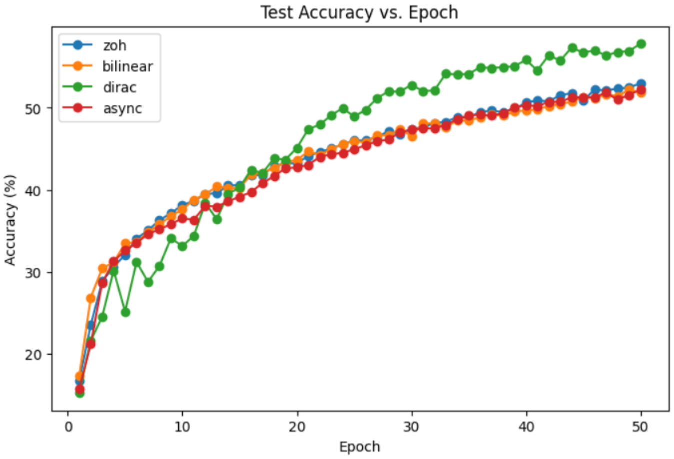

Figure 3: Test Accuracy vs. Epoch for different discretization methods in Implementation 2.

Figure 4 shows the training loss for four different discretization schemes (`zoh`, `bilinear`, `dirac`, and `async`) over 50 epochs. We can see that all schemes steadily decrease loss, but `dirac` (green) tends to have higher initial loss — maybe due to design.

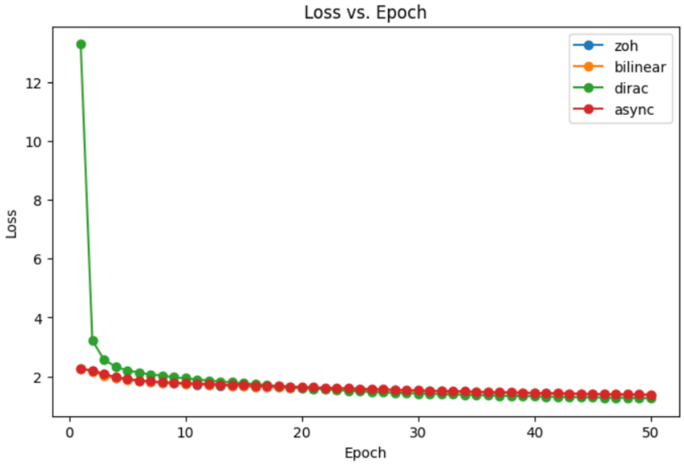

Figure 4: Training Loss vs. Epoch for different discretization methods in Implementation 2.

From these comparisons, it is evident that the choice of discretization can have a noticeable impact on final performance. Users interested in pushing the boundaries of S4-based models may find further improvements by tuning these discretization parameters, exploring other transformations, or employing more advanced hyperparameter optimization techniques.

Why might 'dirac' be outperforming the others The "dirac" discretization avoids applying any extra scaling factors – in particular, it omits the (exp(dtA)1)/A term that is present in the ZOH (and similarly in the asynchronous) version. This simpler "unit scaling" can be beneficial for several reasons:

1. Preservation of Dynamics: By not altering the inherent scale of the system dynamics, the Dirac method might better preserve the original continuous-time behavior, which could align more closely with the optimal kernel for the dataset.

2. Numerical Stability: Additional scaling factors in other methods can amplify numerical errors or lead to issues with gradient flow during training. The absence of these extra factors in the Dirac discretization might result in smoother gradients and more stable optimization.

3. Regularization Effect: The Dirac discretization might implicitly act as a regularizer by avoiding over-amplification of certain modes, which could help in scenarios like sequential CIFAR10 where over-parameterization or scaling issues might hurt performance.

In summary, the Dirac method's performance boost likely stems from its more straightforward handling of the dynamics – by not perturbing the scale, it potentially reduces numerical instability and better captures the essential features needed for the task.

8 Conclusion

We presented a comprehensive overview of state-space models and introduced the S4 model, which incorporates a novel structured parameterization to handle long-range dependencies in sequential data. Our first proof-of-concept implementation on the sCIFAR-10 dataset demonstrates that flattening images into sequences and applying S4 can achieve test accuracies around 60% within 100 epochs. The steady decline in training loss from 2.2 to 1.2 underscores the model's ability to learn effectively.

In the second implementation, we compared multiple discretization schemes (`zoh`, `bilinear`, `dirac`, and `async`). The results suggest that `dirac` generally provides better performance, surpassing 55% test accuracy in fewer than 50 epochs. This highlights how different discretization choices can affect the convergence and final accuracy of S4-based models.

Overall, these findings confirm that S4 can be adapted to image classification tasks, offering a promising avenue for handling long-range dependencies in flattened image sequences. Future work may include extending these ideas to higher-dimensional data, more complex architectures, or refined hyperparameter tuning to further improve accuracy and scalability.