Structured State Space Models - An overview

Lukas Bierling

August 2024

Introduction

The transformer model, based on attention mechanisms, currently stands as the leading approach in language modeling. Over the past few years, various versions of the base model, including GPT and BERT, have been introduced. The transformer architecture encompasses both encoder and decoder components, tailored for discriminative and generative tasks, respectively. It has effectively replaced recurrent neural networks (RNNs) by overcoming the vanishing gradient problem that limits RNNs' ability to process long sequences. Moreover, unlike RNNs, transformers can be trained in parallel, boosting their training efficiency. However, despite their impressive training performance, transformers face challenges in inference efficiency. The requirement to execute multiple attention computations, which scale quadratically, leads to significant computational costs during inference. Conversely, RNNs are more efficient in inference, utilizing constant space to generate output tokens. This highlights a key trade-off in modern language models: balancing training parallelism with inference efficiency. At present, no model excels in both areas while maintaining strong evaluation metrics. For example, linear transformers, which employ only linear operations, support parallel training and maintain constant space complexity for output token generation. However, they fall short in performance compared to traditional non-linear transformers and RNNs. This raises the question of whether any model architecture can achieve both parallel training and inference efficiency. The key to constant space complexity lies in the ability to formulate output generation in a recurrent manner. If a model's internal computations that lead to output can be restructured to follow recurrent principles, inference efficiency can be ensured. This idea has inspired the development of state-space models in NLP, which offer both parallelizable convolutional and recurrent formulations for output generation. However, as highlighted by the authors of the Mamba paper, these models often lack the general reasoning capabilities of transformers. This article will provide an introduction to state-space models in general and their application in NLP, and will conclude by discussing how the new architecture of Mamba, also known as selective state-space models, aims to address the limitations of traditional state-space models.

State Space Models

State space models, originating from control theory, provide a mathematical framework for describing and analyzing systems that evolve over time. These models are composed of several key components:

- x_t: The input vector at time step t, representing external factors influencing the system.
- h_t: The hidden state vector at time step t, encapsulating all the information necessary to describe the system's status at that particular time.
- y_t: The observation (or measurement) vector at time step t, representing the system's outputs or measurements.
- A: The state transition matrix, which defines how the hidden state evolves from one time step to the next.
- B: The input matrix, which determines the influence of the input vector on the state.
- C: The observation matrix, which maps the hidden state to the observations.
- D: The feedthrough (or direct transmission) matrix, which captures the direct influence of the inputs on the observations.

The system's dynamic state and measurement equation are then as follows:

$$h'(t) = Ah(t) + Bx(t) \tag{1}$$

$$y(t) = Ch(t) + Dx(t) \tag{2}$$

State space models use first-order differential equations to describe the evolution of the hidden state over time. This hidden state is then utilized to compute the system's response or output, $y(t)$, based on the input $x(t)$. Hence, being able to compute $y(t)$ requires solving the differential equation to find $h(t)$, which is often hard to analytically. Important to note is that the matrix D is often left out of equation 2 when the SSM is used within neural networks, because the addition $D(x(t))$ is simply a skip connection which can be left out in network architectures. Equation 2 would then be:

$$y(t) = Ch(t) \tag{3}$$

Equations 1 and 2 describe the state space model in its natural, continuous form. However, to apply it on discrete data such as token sequences, we first have to discretize it. In general, there exist various methods to discretize the model. This is also where many state space model differ. Examples are the

bilinear method or using the matrix exponential. For the scope of this article we will only discuss the bilinear method in detail. We know that the continuous hidden state equation of the SSM is given by:

$$\frac{dh(t)}{t} = h'(t) = Ah(t) + Bx(t)$$

Setting $f(t) = Ah(t) + Bx(t)$ we assume that over a small interval $[t_n, t_{n+1}]$ the function $f(t)$ can be approximated by a linear interpolation. With $\delta = t_{n+1} - t_n$ the area under $f(t)$ is then

$$\text{Area} = \Delta \frac{f(t_n) + f(t_{n+1})}{2} \tag{4}$$

Using this linear approximation the change in state h over one time step is then:

$$h_{n+1} - h_n = \frac{\Delta}{2}(f(t_n) + f(t_{n+1}) \tag{5}$$

When we substitute $f(x_n) = Ah_n + Bx_n$ and $f(x_{n+1}) = h_{n+1} + Bx_{n+1}$ we get:

$$h_{n+1} - h_n = \frac{\Delta}{2}\left(Ah_n + Bx_n + Ah_{n+1} + Bx_{n+1}\right) \tag{6}$$

By rearranging, we arrive at:

$$h_{n+1} - \frac{\Delta}{2}Ah_{n+1} = h_n + \frac{\Delta}{2}(Ah_n + Bx_n + Bx_{n+1}) \tag{7}$$

Rewriting this, get us to:

$$\left(I - \frac{\Delta}{2}A\right)h_{n+1} = \left(I + \frac{\Delta}{2}A\right)h_n + \frac{\Delta}{2}B\left(x_n + x_{n+1}\right) \tag{8}$$

where I is the identity matrix. We can now solve for h_{n+1} which results in the final equation:

$$h_{n+1} = \left(I - \frac{\Delta}{2}A\right)^{-1}\left(\left(I + \frac{\Delta}{2}A\right)h_n + \frac{\Delta}{2}B\left(x_n + x_{n+1}\right)\right) \tag{9}$$

With the help of the assumption that the external input $x_t \approx x_{t+1}$ for the small interval Δ, we can even simplify the equation to:

$$h_{n+1} = \left(I - \frac{\Delta}{2}A\right)^{-1}\left(\left(I + \frac{\Delta}{2}A\right)h_n + \Delta B x_n\right) \tag{10}$$

This yields the discretized SSM with:

$$\begin{aligned}
\bar{A} &= \left(I - \frac{\Delta}{2}A\right)^{-1}\left(I + \frac{\Delta}{2}A\right) \\
\bar{B} &= \left(I - \frac{\Delta}{2}A\right)^{-1}\Delta B \\
\bar{C} &= C
\end{aligned}$$

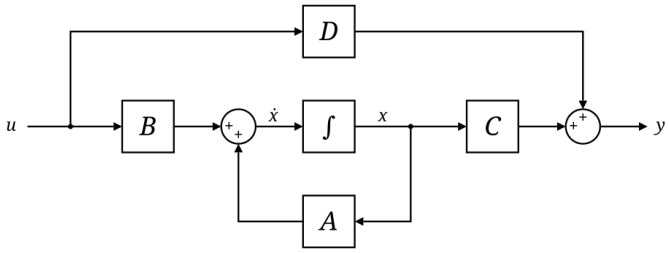


Figure 1: Source: https://en.wikipedia.org/wiki/State-space_representation

and results in the final discrete representation of the SSM:

$$h_{t+1} = \bar{A}h_t + \bar{B}x_{t+1} \tag{11}$$
$$y_{t+1} = \bar{C}h_{t+1} \tag{12}$$

This is actually now just a recurrent formulation which lets us generate output tokens in constant time and space! So the theoretical need of discretizing the SSM led to an useful reformulation which can be now developed further.

As mentioned earlier, the transformer training can be fully parallelized. The recurrent formulation of the SSM cannot. Another widely known model architecture, the one of convlutional neural networks (CNN) consisting of convolutional operations, can also be parallelized [3]. Keeping this in mind, it is possible to reformulate the recurrent SSM to a convolutional one. This is done by unrolling the SSm for multiple time steps to see how inputs and previous hidden states influence updated hidden states. Using equation 11 we can unroll it as:

Step 0: $h_0 = \bar{B}x_0$

Step 1: $h_1 = \bar{A}h_0 + \bar{B}x_1 = \bar{A}\bar{B}x_0 + \bar{B}x_1$

Step 2: $h_2 = \bar{A}h_1 + \bar{B}x_2 = \bar{A}^2\bar{B}x_0 + \bar{A}\bar{B}x_1 + \bar{B}x_2$

...

Step k: $h_k = \bar{A}h_{k-1} + \bar{B}x_k = \bar{A}^k\bar{B}x_0 + \bar{A}^{k-1}\bar{B}x_1 + ... + \bar{A}\bar{B}x_{k-1} + \bar{B}x_k = \sum_{i=0}^{k} \bar{A}^{k-i}\bar{B}x_i$

Consequently, equation 12 can be handled similarly:

Step 0: $y_0 = \bar{C}h_0 = \bar{C}\bar{B}x_0$

Step 1: $y_1 = \bar{C}h_1 = \bar{C}\left(\bar{A}\bar{B}x_0 + \bar{B}x_1\right) = \bar{C}\bar{A}\bar{B}x_0 + \bar{C}\bar{B}x_1$

Step 2: $y_2 = \bar{C}h_2 = \bar{C}\left(\bar{A}^2\bar{B}x_0 + \bar{A}\bar{B}x_1 + \bar{B}x_2\right) = \bar{C}\bar{A}^2\bar{B}x_0 + \bar{C}\bar{A}\bar{B}x_1 + \bar{C}\bar{B}x_2$

...

Step k: $y_k = \bar{C}h_k = \bar{C}\left(\bar{A}^k\bar{B}x_0 + \bar{A}^{k-1}\bar{B}x_1 + ... + \bar{A}\bar{B}x_{k-1} + \bar{B}x_k\right) = \sum_{i=0}^{k} \bar{C}\bar{A}^{k-i}\bar{B}x_i$

This recursive unrolling demonstrates that we can construct a convolutional kernel $\bar{\mathbf{K}}_k = (\bar{C}\bar{B}, \bar{C}\bar{B}\bar{A}, \bar{C}\bar{B}\bar{A}^2, ..., \bar{C}\bar{B}\bar{A}^k)$, such that we can write:

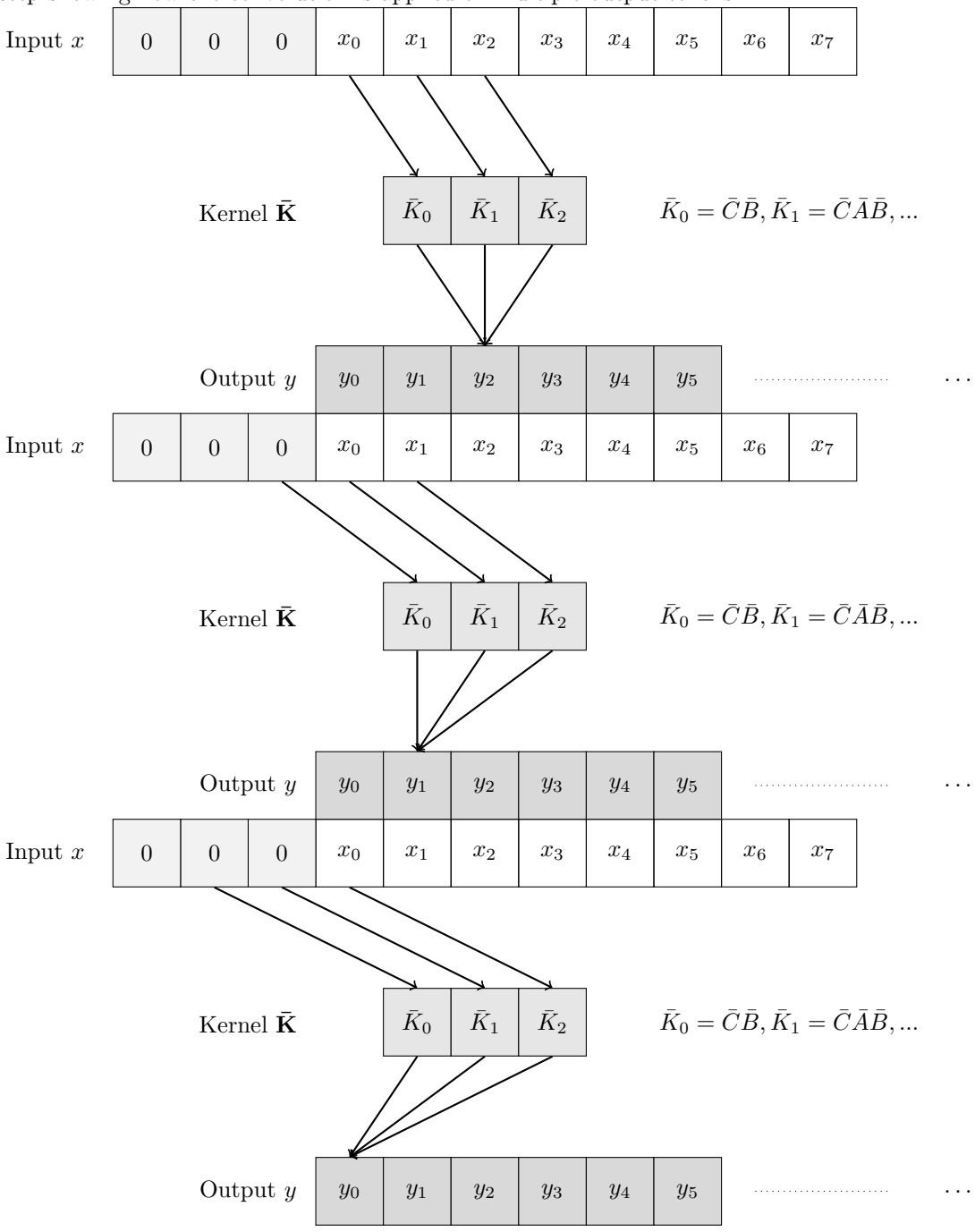
$$y_k = \bar{\mathbf{K}}_k * x \tag{13}$$

where $x = (x_0, x_1, ..., x_k)$. If we apply the correct padding to x we can compute every output $y_i \forall i \in [k]$ with the convolutional operator $\bar{\mathbf{K}}_k$. This can be exemplified by figures 2 to 4. The padding allows us to perform the convolution for each output y_i. This accentuates the flexibility of the convolutional formulation of the SSM.

Usually $\bar{\mathbf{K}}_k$ is called the SSM convolution kernel. We now have a flexible formulation of the SSM. A recurrent one for inference and a parallelizable convolutional one for training. As as summary, the benefits and downsides of both views are:

- The recurrent view:
  - **Positive**: Natural inductive bias for sequential data, and in principle unbounded context.
  - **Positive**: Efficient inference (constant-time state updates).
  - **Negative**: Slow learning (lack of parallelism).
  - **Negative**: Gradient disappearance or explosion when training too-long sequences.
- The convolutional view:
  - **Positive**: Local, interpretable features.
  - **Positive**: Efficient (parallelizable) training.
  - **Negative**: Slowness in online or autoregressive contexts (must recalculate entire input for each new data point).
  - **Negative**: Fixed context size.

Figure 2: Applying $\bar{\mathbf{K}}_k$ on padded input for y_0. It is simply doing $y_0 = \bar{C}\bar{B}x_0$ and the other operations for y_1 and y_2. The depiction demonstrates how it works for padded inputs. There are three different tokens are computed at each step showing how the convolution is applied on multiple output tokens.



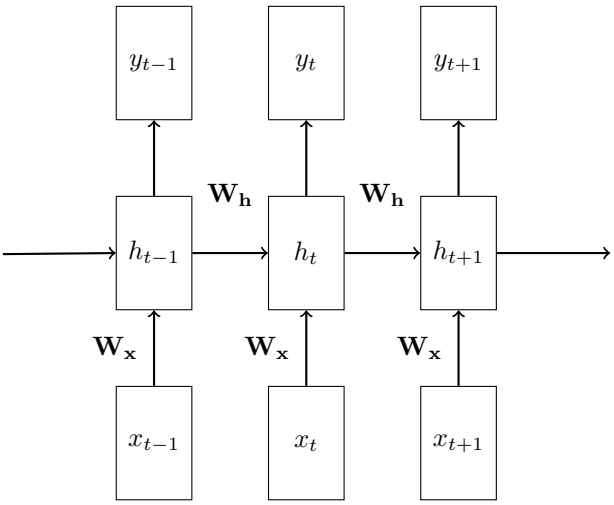


Figure 3: Unrolled RNN depiction

Depending on the stage of the process (training or inference) and the type of data available, we can switch between different perspectives to optimize the model's performance. We favor the convolutional view for fast training through parallelization, the recursive view for efficient inference, and the continuous view for managing continuous data.

It becomes evident that the matrix A and then also $\bar{A}$ is very important to make the SSM expressive and enable it to capture dependencies in the data. Looking at the Kernel $\bar{\mathbf{K}}_k$, one can see how $\bar{A}$ influences the computation of hidden states h_t. While SSMs get trained in the usual neural network way, i.e. via backpropagation, it was shown that the performance of SSM was way worse when A was initialized randomly, i.e. with random weights. Authors of the original paper about state space models (also called structured state space models (S4) [2]) therefore used a so-called HiPPO (High-Order Polynomial Projection Operator) as weight initilization. The matrix A consisting of n rows and k columns can then be written at the start as:

$$A_{nk} = \begin{cases} (-1)^{n-k}(2k+1), & \text{if } n > k \\ k+1, & \text{if } n = k \\ 0, & \text{if } n < k \end{cases} \tag{14}$$

This initialization alone improved model performance enormously when used for language modeling tasks.

By examining Figures 1 and 3, as well as equations 11 and 12, one can observe clear similarities between SSMs and RNNs, particularly when the residual

connection of the matrix D is excluded. This is not surprising, as the basic discretized SSM is essentially a linear RNN, where non-linear activations before computing state h_t are omitted. This observation is significant because it positions SSMs within the context of language models. Additionally, it highlights that a fully linear RNN can be reinterpreted using the convolutional kernel of SSMs, which allows for parallel training. In summary, SSMs can be assigned to the space of RNNs. However, as they are linear they can be parallelize d during training. Additionally, they make use of the discretization process which directly influences the values of the weights matrices $\bar{A}$ and $\bar{B}$. SSMs used for NLP in the context of neural networks can be described by the following algorithm [1]:

Algorithm 1 SSM (S4)

Input: $\mathbf{x} : (B, L, D)$
Output: $\mathbf{y} : (B, L, D)$
$\mathbf{A} : (D, N) \leftarrow$ Parameter ▷ Represents structured $N \times N$ matrix
$\mathbf{B} : (D, N) \leftarrow$ Parameter
$\mathbf{C} : (D, N) \leftarrow$ Parameter
$\Delta : (D) \leftarrow \tau\Delta$(Parameter)
$\mathbf{A}, \mathbf{B} : (D, N) \leftarrow$ discretize$(\Delta, \mathbf{A}, \mathbf{B})$
$\mathbf{y} \leftarrow \text{SSM}(\mathbf{A}, \mathbf{B}, \mathbf{C})(\mathbf{x})$ ▷ Time-invariant: recurrence or convolution
return $\mathbf{y}$

A, B and Δ are trainable parameters accentuating the difference to RNNs where Δ is not part of the trainable parameters. This defines one SSM block that can be used analogously to attention blocks in the Transformer. Its input and output shapes are the same making the stacking of them simple to build deep multi-layer SSM models.

Selective State Space Models

Introduction and Intuition

While SSMs provide notable efficiency improvements over classic transformer or RNN models, they still lack the performance and expressive power needed for certain tasks. One key reason for this, and also why the transformer architecture has been so successful, is the absence of input dependency in the hidden states h_t. The matrices A, B, and C are the same for all input tokens x_t. The objective of the learning process in an SSM is, therefore, to construct universal matrices that track past hidden states and inputs regardless of the specific input. This rigidity makes them inflexible when information about token positions in a sequence is required.
For instance, consider the following three sequences:

(1) The grass is green. The grass is green. The grass is green. The rose is red. The grass is green. The grass is green. The grass is green.

(2) The grass is green. The grass is green. The grass is green. The rose is white. The grass is green. The grass is green. The grass is green.

(3) The grass is green. The rose is white. The grass is green. The grass is green. The grass is green. The grass is green. The grass is green.

Suppose the model is tasked with answering the question: *What color is the rose?* Further, suppose the matrix A is designed to emphasize more recent tokens because it was trained that way. It may capture the color of the rose in the hidden state for the first two sequences. However, because A does not adjust based on the input, it fails to capture the information about the rose in the last sequence. This scenario underscores the limitations of static matrices in SSMs. They are unable to dynamically adjust the weights to capture relevant information but must instead rely on a general matrix A that applies universal rules to remember past tokens in the sequence.

To better understand the differences, let's compare this to the transformer approach. A transformer model utilizes attention blocks to map an input sequence $X \in R^{n \times d}$ to an output sequence $Y \in R^{n \times d}$, using an attention mechanism denoted as $\mathbf{A}$. This results in input-output shapes that are identical to those found in the SSM block described in Algorithm 1. However, the key distinction lies in how the attention mechanism $\mathbf{A}$ operates.

The attention mechanism relies on three components: keys, queries, and values. These are generated by applying separate linear transformations to the input X. Specifically, the keys K and queries Q have dimensions d_k, while the values V have dimensions d_v.
The attention process begins with the computation of the dot product between queries and keys, followed by the application of a softmax function (equation 15) to derive the attention scores $\mathbf{S}$. These scores are then used to weight the value vectors, producing the final weighted matrix $\mathbf{A}$, as expressed formally by the following equations:

$$\mathbf{S}(Q, K) = \text{softmax}(QK^{\text{T}}) \tag{15}$$

$$\mathbf{A}(Q, K, V) = \mathbf{S}(Q, K)V \tag{16}$$

The attention block $\mathbf{A}$ integrates this mechanism as follows:

$Q, K, V = W_Q X, W_K X, W_V X$
$A = W_{proj} X + \mathbf{A}(Q, K, V)$ (with a residual connection to the input)
$Y = \mathbf{A}(X)$

In this context, W_Q, W_K, W_V, and W_{proj} represent linear transformations. A crucial aspect of this process is that the attention mechanism $\mathbf{A}$ is inherently dependent on the input X, as the keys, queries, and values are directly derived from it. This input-dependent nature allows the attention mechanism to dynamically construct a representation of the sequence, assigning higher importance to tokens that are more relevant within the context of the input sequence.

This dynamic weighting is primarily achieved through the matrix $\mathbf{S} \in R^{n \times n}$. This matrix contains normalized scores that reflect the relative importance of tokens. For example, $\mathbf{S}_{ij}$ represents a score $s \in [0, 1]$ indicating how important the token at position j is relative to the token at position i. This mechanism can be compared to the hidden state h_t in earlier models, which is intended to capture all prior token information in the sequence. However, $\mathbf{S}$ is evidently more expressive as it captures the relative importance of each token independently. Consequently, the values in $\mathbf{S}$ vary for each input sequence, since different parts of the sequence may hold more significance depending on the context. This is where the hidden state h_t in SSMs or RNNs falls short, as it cannot capture token-specific information but instead acts as a universal hidden state tracker, represented by the matrix A (which is distinct from the attention block $\mathbf{A}$). One could argue, that the transformer does explicitly not use a hidden state h_t that compresses the past context. This accentuates the classic trade-off: Not using h_t adds more expressivity but also requires the transformer to store $\mathbf{S}$ in memory when doing inference for one token. Hence, it is running in quadratic time during inference compared to the constant time of RNNs or SSMs.

Using the rose example form before undermines how the context-dependent matrix $\mathbf{S}$ overcomes the SMM-rigidity in the form of matrix A. The attention mechanism in Transformers works like a smart focus that can adjust itself depending on what the model needs to understand. When the Transformer reads the sequence, it doesn't treat every word equally. Instead, it learns to pay attention to the parts of the sequence that are most relevant to the question at hand.
For example, when asked "What color is the rose?", the Transformer will start by identifying where the word "rose" appears in the sequence. Then, it will focus its attention on the words that are nearby and likely describe the rose, such as "red" or "white." This is done through the attention scores matrix $S\mathbf{S}$, which assigns higher values to the words that are more closely related to "rose" in the context of the sentence. In sequence (1):

The rose is red.

The attention mechanism will highlight the word "red" because it appears next to "rose." This focus ensures that the model understands that "red" is describing

the rose.
Similarly, in sequence (2):

The rose is white.

The attention shifts to "white" as the relevant descriptor for "rose," even though the rest of the sequence may contain repeated or less relevant information.
Even in sequence (3):

The rose is white. The grass is green. The grass is green. The grass is green.

The Transformer can still correctly identify "white" as the color of the rose, despite the presence of distracting information. This is because the attention mechanism dynamically adjusts its focus based on the importance of the words relative to the question being asked.
In contrast, an SSM would struggle in this situation because its static matrix A would treat the entire sequence uniformly, unable to hone in on the critical information about the rose's color.

The dynamic nature of the Transformer's attention mechanism allows it to understand and prioritize different parts of the sequence based on the context, making it far more effective in handling complex language tasks where relationships between words and their meanings are crucial. This ability to adjust focus based on input is why Transformers excel in tasks that require nuanced understanding of sequence structure and contextual meaning.

This dynamic, context-sensitive processing allows transformers to effectively capture and emphasize relationships between tokens, making them highly effective in tasks that require an understanding of sequence structure and contextual meaning.

Selection in SSM

To overcome the lack of expressivity of the hidden state in classic SSMs, the authors of *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*Gu and Dao [1] addressed this by introducing selective state space models. They mention a common issue, or problem set, the traditional SSM cannot solve, the selective copying problem. The usual copying algorithm requires the model to shift sequential tokens at positions $i - k$ to positions $i + j - k + j$. In selective copying the model has now to find, i.e. select, the tokens to be copied instead using the sequence starting at position i. Figures 4 and 5 display the problem. It requires content-aware reasoning to be able to memorize the relevant tokens (colored) and filter out the irrelevant ones (white).

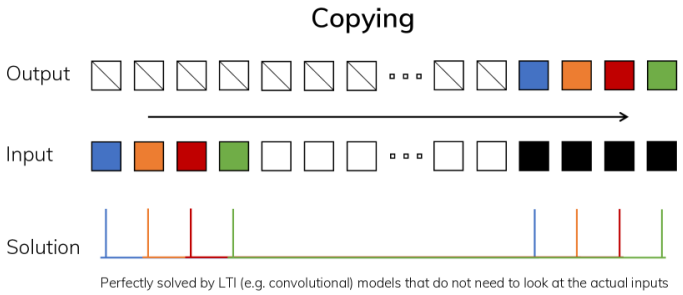


Figure 4: Classic copying problem. Shift sequential colored tokens to the black positions

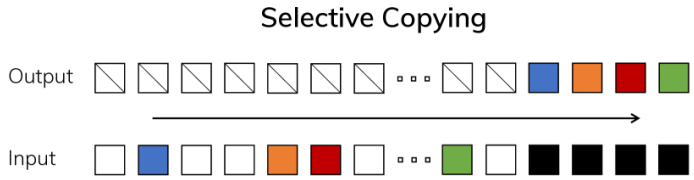


Figure 5: Selective copying problem. Shift arbitrarily placed colored tokens to the black positions

These tasks highlight the limitations of SSMs. From a recurrent perspective, their fixed dynamics (such as the constant transitions in (11)) prevent them from selecting the appropriate information from the context or modifying the hidden state based on input. In the convolutional context, while global convolutions can address simple tasks like vanilla Copying due to their time-awareness, they struggle with the Selective Copying task due to their lack of content-awareness (Figure 2). Specifically, the variable spacing between inputs and outputs cannot be effectively modeled by static convolutional kernels [1].

In summary, the tradeoff between efficiency and effectiveness in sequence models hinges on how well they compress their state: efficient models maintain a small state, whereas effective models retain all necessary context information within their state. Therefore, the authors propose that a key principle for developing sequence models is selectivity, which refers to the ability to focus on or filter inputs based on the context within a sequential state. This selection mechanism governs how information is propagated or interacts along the sequence dimensions [1].

This selection ability is highlighted with the Algorithm 2 for selective SSMs

(S6). To enable the model to focus on different parts of the inputs, we now introduced flexible B and C weight matrices, that depend on the input token. They are determined with their corresponding shapes as:

$$\mathbf{B} = s_B(x) = Linear_H(x) : (B, N, H)$$
$$\mathbf{C} = s_C(x) = Linear_H(x) : (B, N, H)$$
$$\Delta : (B, N, H) = \tau_\Delta(\text{Parameter} + s_\Delta(x)) = Linear_1(x) : (B, N, H)$$

where $s_B(x)$ and $s_C(x)$ are linear projections to dimension H that depend on the input x. That means, the linear projections are different when they are applied on different input tokens. Hence, the matrices $\mathbf{B}$ and $\mathbf{C}$ as well as the discretization parameter Δ are not static anymore as in Algorithm 1. While the matrix $\mathbf{A}$ remains static, the influence on $\mathbf{A}$, the hidden state and output through $\mathbf{B}$ and $\mathbf{C}$ is dynamic. This allows the model now, to selectively choose on what to focus in the hidden state h_t based on the input x_t. As we can see now, the shape of (B, N, H) indicates that we have separate weight matrices for each token in the sequences (through the additional second dimension N).

Algorithm 2 SSM + Selection (S6) [1]

1: **Input:** $\mathbf{x} : (B, N, D)$
2: **Output:** $\mathbf{y} : (B, N, D)$
3: $\mathbf{A} : (D, H) \leftarrow$ Parameter ▷ Represents structured $N \times N$ matrix
4: $\mathbf{B} : (B, N, H) \leftarrow s_B(x)$
5: $\mathbf{C} : (B, N, H) \leftarrow s_C(x)$
6: $\Delta : (B, N, H) \leftarrow \tau_\Delta(\text{Parameter} + s_\Delta(x))$
7: $\bar{\mathbf{A}}, \bar{\mathbf{B}} : (B, N, D, H) \leftarrow \text{discretize}(\Delta, \mathbf{A}, \mathbf{B})$
8: $y \leftarrow \text{SSM}(\bar{\mathbf{A}}, \bar{\mathbf{B}}, \mathbf{C})(x)$ ▷ Time-varying: recurrence (scan) only
9: **return** y

While reading the paper, I was confused how exactly the dependency on the input x is carried out in practice. Looking at the code implementation, I soon found out that steps 4 to 6 are actually just linear layers, i.e. separate linear projections based on the input.

Our discretized matrices $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$ now have the shape (B, N, D, H). This arises because the discretization involves multiplying Δ with $\mathbf{A}$, leading to this shape. Consequently, the hidden state now has the shape (B, D, H). This is somewhat counterintuitive, as hidden states in RNNs typically have the shape (B, H), i.e., they are a single vector compressing the sequence information. However, in S6, the hidden states (excluding the batch dimension) are matrices. This change is necessitated by the construction of $\bar{\mathbf{A}}$, where the only viable approach to make Equations 11 and 12 work is by aligning the shape of the hidden state accordingly, resulting in a two-dimensional tensor, i.e., a matrix when the batch dimension is ignored.

The selection process introduces a new formulation. As the state transformation matrices are now input dependent, i.e. time varying, the formulas change to:

$$\begin{aligned}
&\text{Step 0:} \quad h_0 = \bar{B}_0 x_0 \\
&\text{Step 1:} \quad h_1 = \bar{A}_1 h_0 + \bar{B}_1 x_1 \\
&\text{Step 2:} \quad h_2 = \bar{A}_2 h_1 + \bar{B}_2 x_2 \\
&\dots \\
&\text{Step k:} \quad h_k = \bar{A}_{k-1} h_k + \bar{B} k x_k
\end{aligned}$$

Analogously, the computation of y_t changes. Unlike Equation 13, where the formulas were compressed, the non-static and unknown transition matrices now prevent such compression. As a result, a convolutional formulation is no longer possible. This is a significant drawback since the main appeal of SSMs was their ability to offer constant-time inference and parallel training, making them a viable alternative to Transformers. Without the convolutional perspective, parallel training becomes infeasible, undermining the primary advantage of SSMs and making Transformers a more suitable option.
To address this issue, the authors introduced the *selective scan* algorithm, which enables the fully parallelized computation of hidden states on modern GPUs, even when input-dependent.

Selective Scan

A common competitive programming problem is the one of the prefix sum. Given an array a of integers, find the cumulative sum of the array per entry.

Initial array a: [1, 2, 3, 4, 5]

1	2	3	4	5

Prefix Sum Array: [1, 3, 6, 10, 15]

1	3	6	10	15

Figure 6: Prefix sum problem

This logic is also applicable on the SSM state computation. Remember, each hidden state h is computed as:

$$h_t = \bar{A}_t h_{t-1} + \bar{B}_t x_t$$

Let $a = [x_0, x_1, x_2, ..., x_n]$ be the array of inputs, then the array of hidden states $b = [h_0, h_1, h_2, ..., h_n]$ is nothing else but the prefix sum array based on a with a slight modification. Instead of simply summing over the input entries, we perform the state update computation.

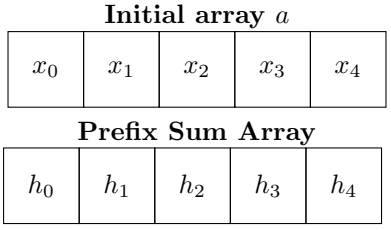


Figure 7: Prefix sum problem applied on SSM formula. Formally, $h_t = \bar{A}h_{t-1} + \bar{B}x_t$

These problem can be easily solved in $\mathcal{O}(n)$ time in sequential way. However, there exists also a parallel algorithm. Keeping this deduction in mind, we now introduce the parallel scan algorithm. It offers a new perspective on how to solve the prefix sum problem by speeding it up using parallelism concepts.

The parallel scan has a sweep up phase and a down sweep phase.

1. **Up-Sweep Phase:** For the first level, pair adjacent element and compute the sum of them. Store them in the array. For the second level, pair the sums from the previous level and store them in the array. This process is continued until we reach the root element, i.e. the last element. This operations have inherently created a binary tree with $\log(n)$ levels, where n is the number of elements in the array. Using the array [3,1,4,0,2,1,3,2] as input, Figure 8 displays how the binary tree is constructed.

2. **Sweep-Down Phase:** Replace the last element, i.e. the root, with a 0. For each level, replace the left child of an element with its parent value and replace the right child with the sum of the parent plus the (new) left child value. Continuing this until we reach the leaves, results in the desired prefix sum array. Figure 9 visualizes this.

As we can parallelize the operations per level on a GPU, the time complexity per level is $\mathcal{O}(1)$. A binary tree consisting of n elements has a depth of $\log(n)$. So the time complexity of the parallel scanfor each phase is $\mathcal{O}(log(n))$, for both phases in total $\mathcal{O}(2log(n))$. Since constants are ignored in complexity theory, the overall time complexity of the algorithm is $\mathcal{O}(log(n))$. This now lets us optimize the training of the selective SSM introducing a parallelizable training formulation trhough the parallel scan. The authors even extended this scan by introducing GPU hardware optimization concepts where the High-Bandwith Memory and SRAM are accessed in a highly efficient manner. Their selective scan algorithm runs then fully optimized on GPUs and is highly efficient for large inputs [1].

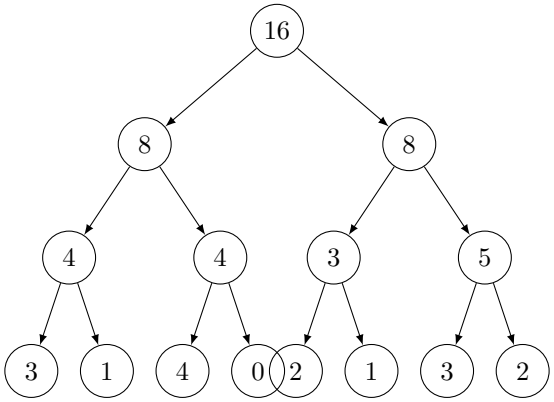


Figure 8: Binary tree construction for parallel scan based on array [3,1,4,0,2,1,3,2]

Mamba Model Architecture

The authors of Mamba now introduce the Mamba model as consisting of multiple Mamba blocks. Each block makes use of the selective SSM inside followed by some non-linearity operations. A block takes an input x and splits it into two through two linear projections. The first split is then passed through a convolutional layer followed by a non-linear SiLU activation function. Then, the selective SSM is applied and afterwards, the other splits get added back through a residual connection. The final output is projected back to a desired dimension. These blocks can now be stacked on top of each other just as in the Transformer. Refer to Figure 10 for a visual depiction. However, interestingly, the selective SSM performs a fully linear sequence modeling step through the SSM algorithm, and the non-linearity only comes into play later. Another interesting relation is the one about the size of the hidden state in different model architectures. As shown earlier, the Transformer makes use of the largest possible size of a hidden state by storing all relations between tokens in memory. The RNN only uses a very small hidden state which is not input-dependent. Mamba ranks in between the two, hoping to get the best of both worlds while staying efficient.

Summary

The Mamba model offers a new perspective on language models through questioning the dominance of the Transformer. It makes use of SSMs and levels them up through selection. This process makes them highly effective. Their innovative use of the parallel scan also makes them highly efficient, enabling them to process longer sequences without having the quadratic complexity bottleneck of the transformer during inference. The architecture and concepts are

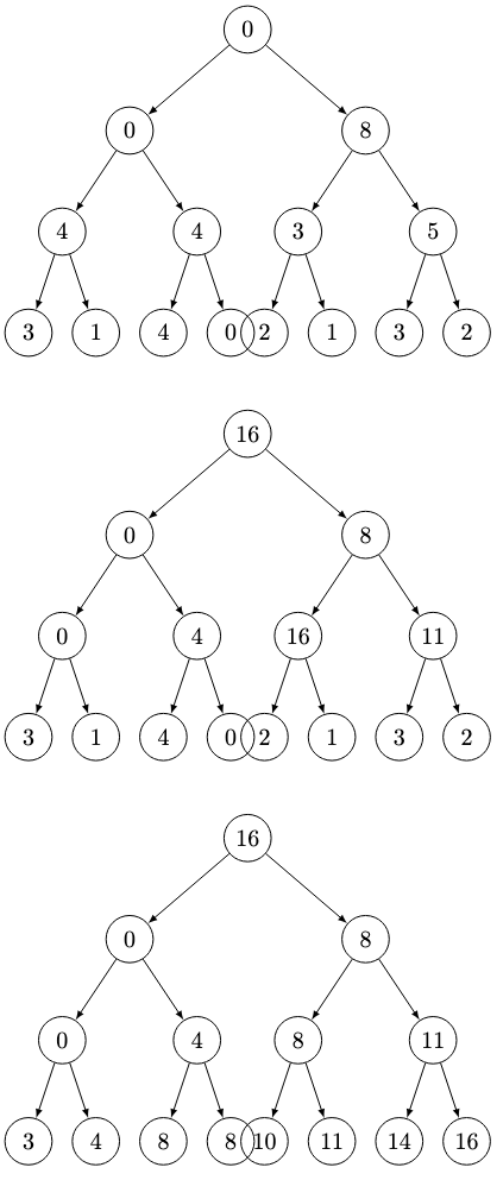


Figure 9: Down-Sweep phase of selective scan for levels 1-3 based on the array [3,1,4,0,2,1,3,2]. First level is skipped for simplicity reasons

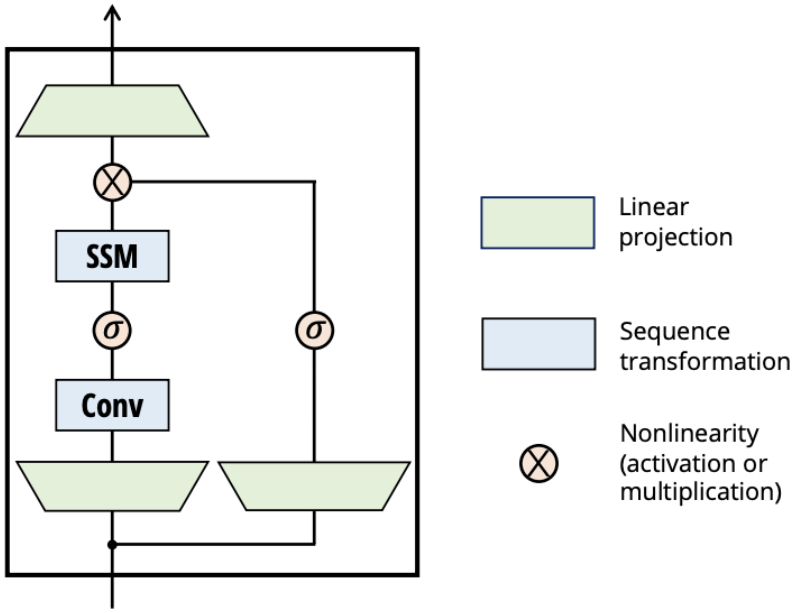


Figure 10: Mamba block [1]

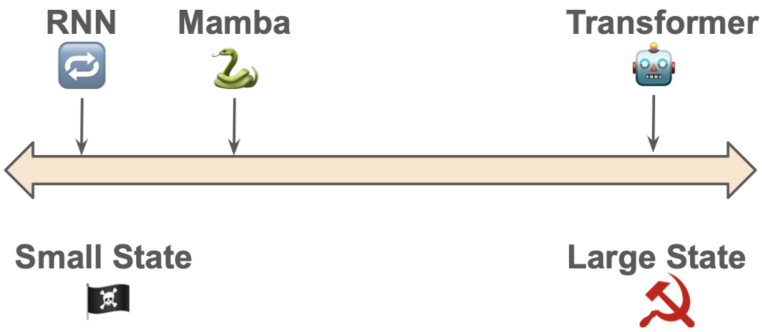


Figure 11: State size comparison of different models (https://thegradient.pub/mamba-explained/)

more advanced than in the Transformer and sometimes a bit counterintuitive. This may lead to less popularity in the first few years because researchers have to invest more resources in understanding the concept of SMMs and selectivity. Its first empirical experiments showed promising results for the realm of NLP. Whether the models get adapted in the future depends on the initiative of researchers, and whether they are willing to move away from the leading paradigm of Transformers.

References

[1] Albert Gu and Tri Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. 2024. arXiv: 2312.00752 [cs.LG]. URL: https://arxiv.org/abs/2312.00752.

[2] Albert Gu, Karan Goel, and Christopher Ré. *Efficiently Modeling Long Sequences with Structured State Spaces*. 2022. arXiv: 2111.00396 [cs.LG]. URL: https://arxiv.org/abs/2111.00396.

[3] *Introduction to State Space Models (SSM) — huggingface.co*. https://huggingface.co/blog/lbourdois/get-on-the-ssm-train. [Accessed 04-08-2024].