Unveiling the Power of Attention: A Deep Dive into Attention Mechanisms

Bilal FAYE

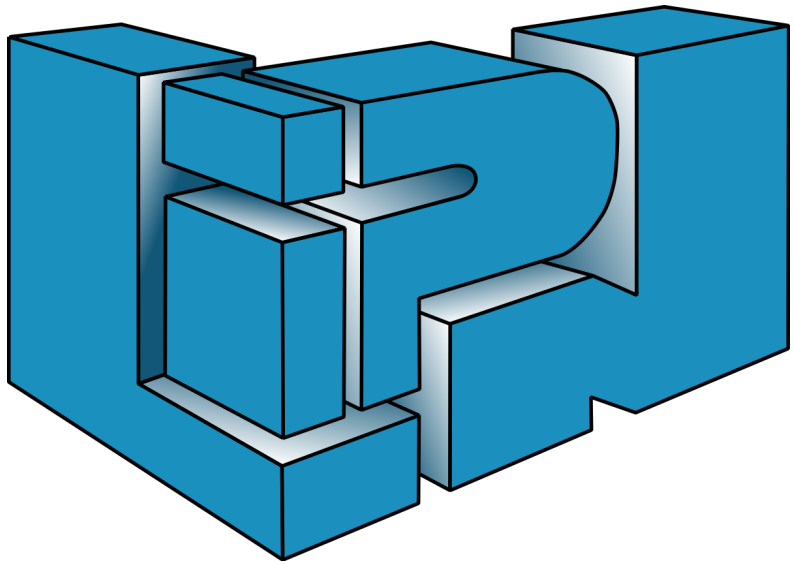


Table of contents

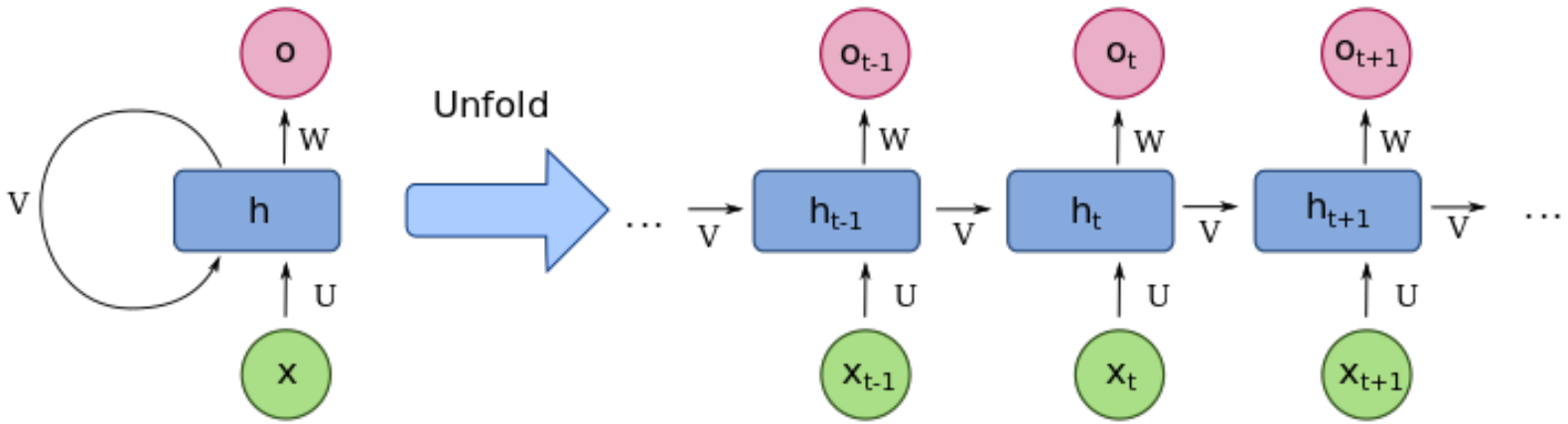
Introduction to sequence models

Sequence models map input sequence to output sequence. The input sequence can be continuous (audio) or discrete (text) signal and output either can be continuous or discrete signal.

There are several type of sequence models :

- Recurrent Neural Networks (RNNs)
- Convolutional Neural Networks (CNNs)
- Transformers
- State Space Models (SSMs)

Recrrent Neural Networks (RNNs)


o
W
V
h
U
x
Unfold
...
o_{t-1}
o_t
o_{t+1}
h_{t-1}
h_t
h_{t+1}
x_{t-1}
x_t
x_{t+1}

RNNs properties :

- Infinite context window (theorically)
- Training $O(N)$ but not parallelizable
- Inference $O(1)$: constant time for each token generation

Convolutional Neural Networks (CNNs)

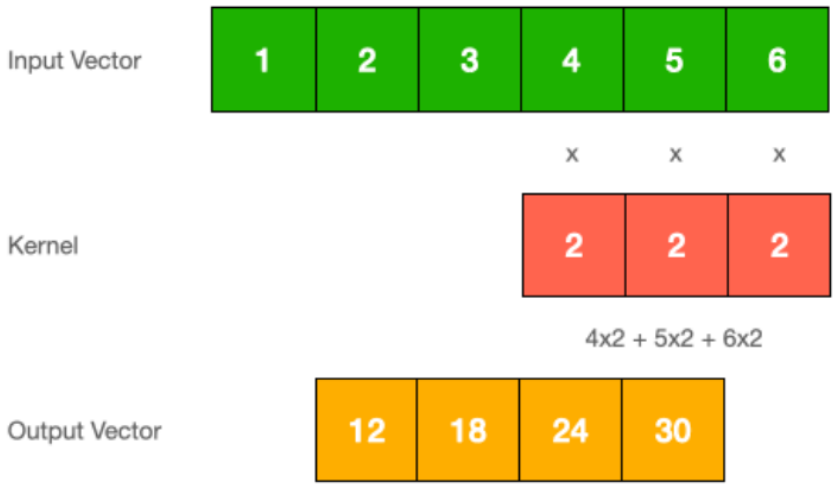
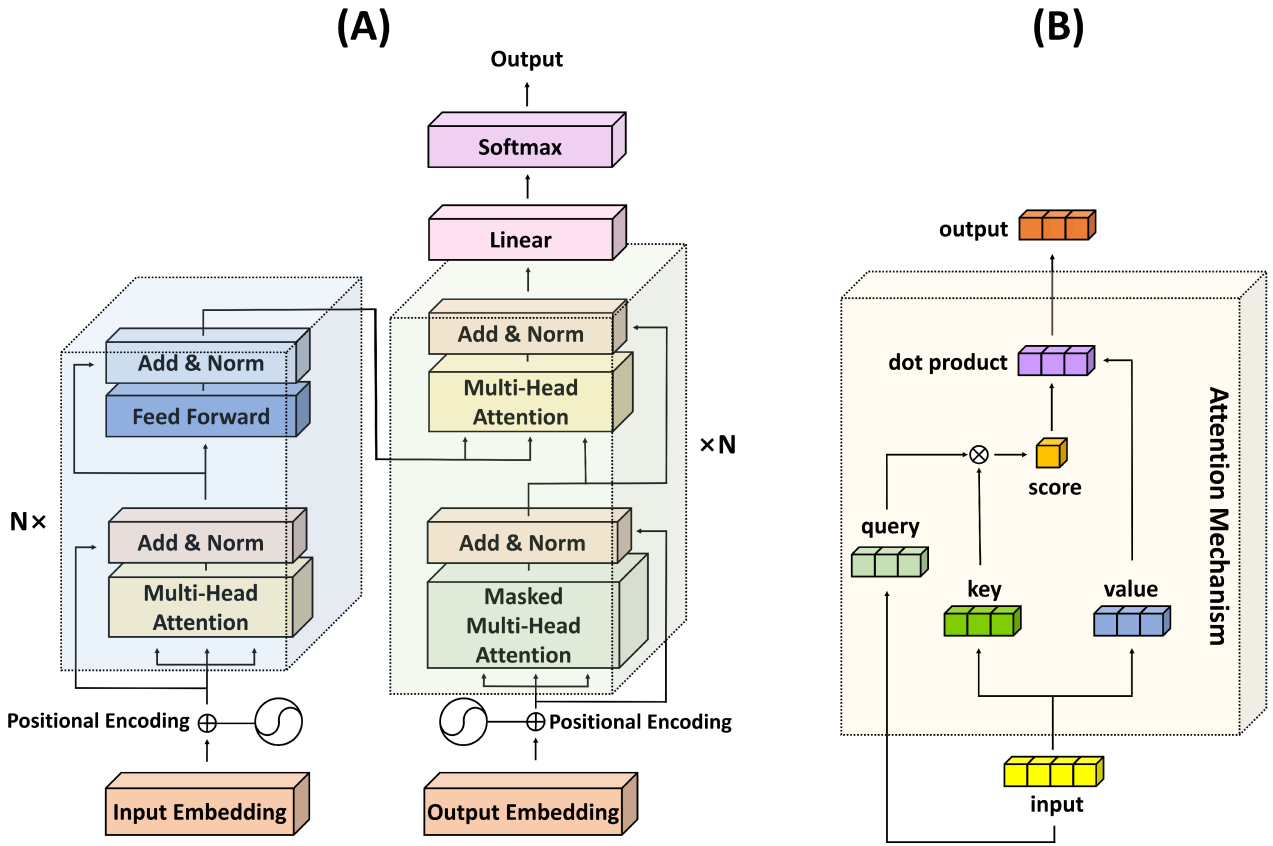


Figure – Conv1D

CNNs properties :

- Finite context window (depending on kernel size)
- Training and inference depend on kernel size
- Easily parallelizable

Transformers


(A)
Output
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
×N
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
(B)
output
dot product
score
query
key
value
input
Attention Mechanism

Transformers properties :

- Finite context window
- Training : $O(N^2)$, easily parallelize
- Inference : $O(N)$ when using KV-cache $((Q.K).V = Q.(K.V))$ for each token. To generate the 10^{th} token we need to perform 10 dot product, 11 to generate the 11^{th} etc.

State Space Models (SSMs)

SSMs propeties :

- Share notable similarities with RNNs
- They parallelize training as Transformers (O(N))
- Can inference each token with a constant computation cost (O(1)) like the RNN.

State Space Models (SSMs)

A state space model allows to map input signal $x(t)$ to output signal $y(t)$ by means of state representation $h(t)$ as follow :

$$\begin{aligned} h'(t) &= Ah(t) + Bx(t) \\ y(t) &= Ch(t) + Dx(t) \end{aligned} \tag{1}$$

where :
$h(t)$: State
$h'(t)$: Derivate of the state (differential equation)
$x(t)$: Input vector
$y(t)$: Output vector
A, C, D : Fixed values

The SSM is **linear** and **time-invariant** :

- Linear : relationships in Equation 1 are linear
- Time-invariant : parameters A, B, C, D remain constant over time, independent of t. The process to generate $y(t)$ is $O(1)$ regardless of t.

The primary objective of the State Space Model (SSM) is to model $h(t)$ based on the differential equation 1.

Finding an analytical solution to the differential equation 1 is challenging.

However, we can approximate the solution using methods such as Euler's method and zero-order hold.

Euler method illustration

Typically, we don't deal directly with continuous signals $x(t)$; instead, we discretize them due to sampling. How do we generate $y(t)$ for a discrete signal ?

Let's consider $b'(t) = \lambda b(t)$.

The derivative of a function represents its rate of change :
$b'(t) = \lim_{\Delta \to 0} \frac{b(t+\Delta)-b(t)}{\Delta}$.

If Δ is very small,
$b'(t) \approx \frac{b(t+\Delta)-b(t)}{\Delta} \implies \Delta b'(t) + b(\Delta) \approx b(t+\Delta)$.

$\lambda b(t)\Delta + b(t) \approx b(t+\Delta) \implies b(t+\Delta) \approx \lambda b(t)\Delta + b(t)$.

This yields a recurrent formulation : $b(t+\Delta) \approx \lambda b(t)\Delta + b(t)$.

When we fix λ and Δ, we can compute the system's evolution when $b(0)$ is known.

Using the derivative, we approximate $h(t+\Delta)$ as $\Delta h'(t) + h(t)$. Substituting $h'(t)$ from equation 1 :

$$\begin{aligned} h(t+\Delta) &= \Delta(Ah(t) + Bx(t)) + h(t) \\ &= \Delta Ah(t) + \Delta Bx(t) + h(t) \\ &= (\Delta A + I)Ah(t) + \Delta Bx(t) \\ &= \bar{A}h(t) + \bar{B}x(t) \end{aligned}$$

This allows us to express $h(t+\Delta)$ in terms of $h(t)$ and $x(t)$, where $\bar{A} = \Delta A + I$ and $\bar{B} = \Delta B$.

After discretization, the system is modeled as follows :

$$\begin{aligned} h_t &= \bar{A}h(t-1) + \bar{B}x(t) \\ y(t) &= Ch(t) + Dx(t) \end{aligned} \tag{2}$$

In practice, the Zero-order Hold method provides a better approximation than the Euler method :

$$\bar{A} = \exp(\Delta A)$$

$$\bar{B} = (\Delta A)^{-1}(\exp(\Delta A) - I) \cdot \Delta B$$

Here, A, B, C, and Δ are treated as learnable parameters.

Computation of $h(t)$ can be approached using two distinct methods :

- Recurrent computation
- Convolutional computation

Recurrent computation

With the recurrent formula $h_t = \bar{A}h_{t-1} + \bar{B}x_t$ and $y_t = Ch_t$ (where $Dx(t)$ is not utilized and can be considered as a skip connection), we are able to compute the system's output for various time steps.
Let's suppose the initial state of the system is $h_0 = \bar{B}x_0$ and $y_0 = Ch_0$.
At step 1 :

$$h_1 = \bar{A}h_0 + \bar{B}x_1$$

$$y_1 = Ch_1$$

At step 2 :

$$h_2 = \bar{A}h_1 + \bar{B}x_2$$

$$y_2 = Ch_2$$

This mechanism operates similarly to Recurrent Neural Networks.

- Limits :
  - The recurrent formulation is not suitable for training time.
- Advantages :
  - The recurrent formulation excels at inference time, with constant time complexity $O(1)$ for generating each token.

Convolutional computation

$$h_0 = \bar{B}x_0$$
$$y_0 = Ch_0 = C\bar{B}h_0$$

$$\begin{aligned} h_1 &= \bar{A}h_0 + \bar{B}x_1 \\ y_1 &= Ch_1 = C(\bar{A}h_0 + \bar{B}x_1) \\ &= C(\bar{A}(\bar{B}x_0) + \bar{B}x_1) \\ &= C\bar{A}\bar{B}x_0 + C\bar{B}x_1 \end{aligned}$$

$$y_k = C\bar{A}^k\bar{B}y_0 + C\bar{A}^{k-1}\bar{B}y_1 + C\bar{A}^{k-2}\bar{B}y_2 + \ldots + C\bar{B}y_k$$

By utilizing the derived formula, we observe an interesting pattern : the output of the system can be computed using a convolution operation between a kernel $\bar{K}$ and the input $x(t)$:

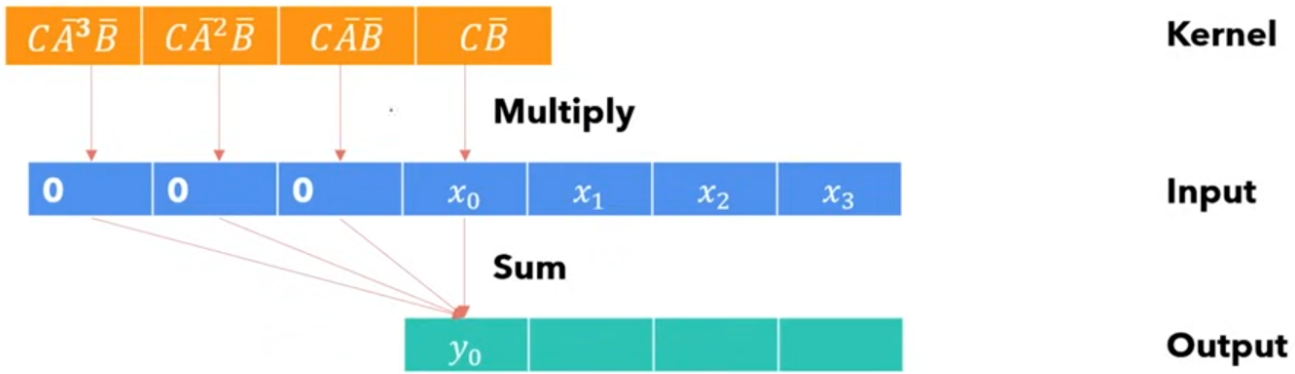
The kernel $\bar{K}$ can be regarded as a filter. If we treat A, B, C, x_t, and y_t as numerical values, we can apply $\bar{K}$ to the input x_t to compute y_t as follows :

Convolutional formulation: step 1

$$y_k = C\bar{A}^k\bar{B}x_0 + C\bar{A}^{k-1}\bar{B}x_1 + \cdots + C\bar{A}\bar{B}x_{k-1} + C\bar{B}x_k$$

$$\overline{\boldsymbol{K}} = (C\overline{\boldsymbol{B}}, C\overline{\boldsymbol{AB}}, \ldots, C\overline{\boldsymbol{A}}^{\boldsymbol{k}}\overline{\boldsymbol{B}}, \ldots)$$

$$y = x * \overline{\boldsymbol{K}}$$



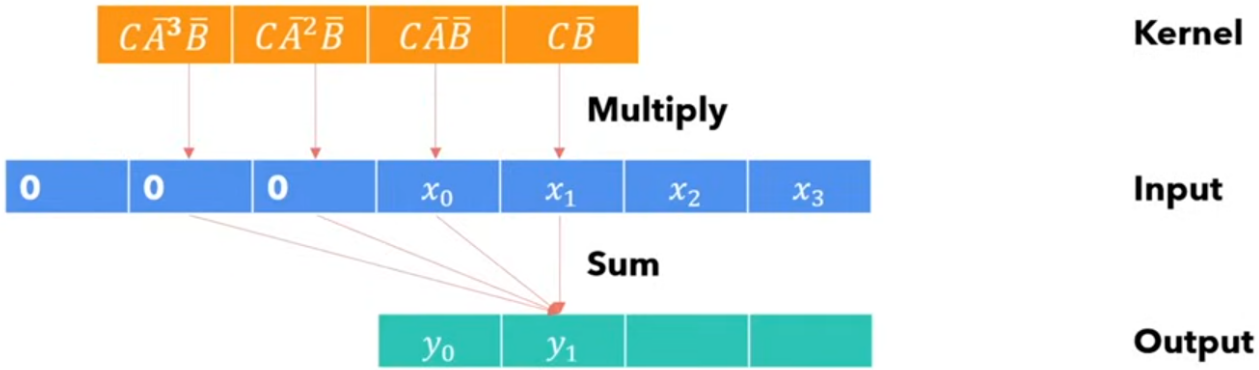
$$y_0 = C\bar{B}x_0$$

Convolutional formulation: step 2

$$y_k = C\bar{A}^k\bar{B}x_0 + C\bar{A}^{k-1}\bar{B}x_1 + \cdots + C\bar{A}\bar{B}x_{k-1} + C\bar{B}x_k$$

$$\overline{K} = (C\overline{B}, C\overline{AB}, \ldots, C\overline{A}^k\overline{B}, \ldots)$$

$$y = x * \overline{K}$$



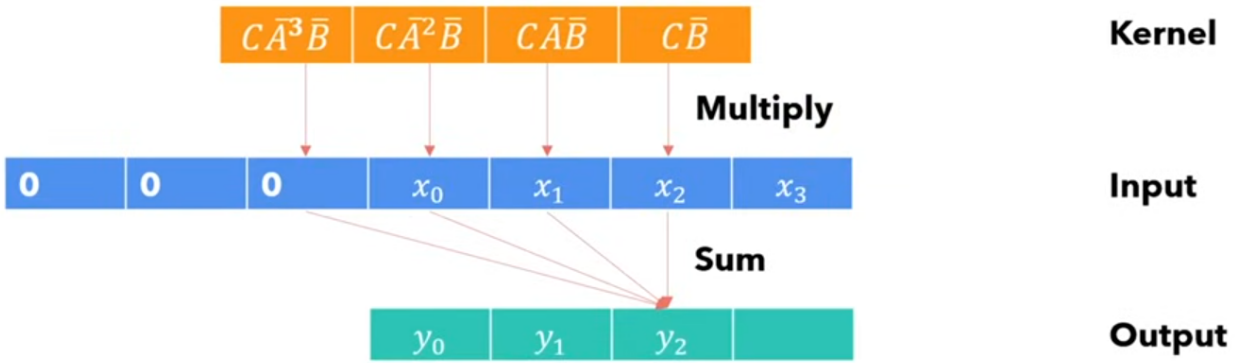
$$y_1 = C\bar{A}\bar{B}x_0 + C\bar{B}x_1$$

Convolutional formulation: step 3

$$y_k = C\bar{A}^k\bar{B}x_0 + C\bar{A}^{k-1}\bar{B}x_1 + \cdots + C\bar{A}\bar{B}x_{k-1} + C\bar{B}x_k$$

$$\overline{\boldsymbol{K}} = (C\overline{\boldsymbol{B}}, C\overline{\boldsymbol{AB}}, \ldots, C\overline{\boldsymbol{A}}^{\boldsymbol{k}}\overline{\boldsymbol{B}}, \ldots)$$

$$y = x * \overline{\boldsymbol{K}}$$



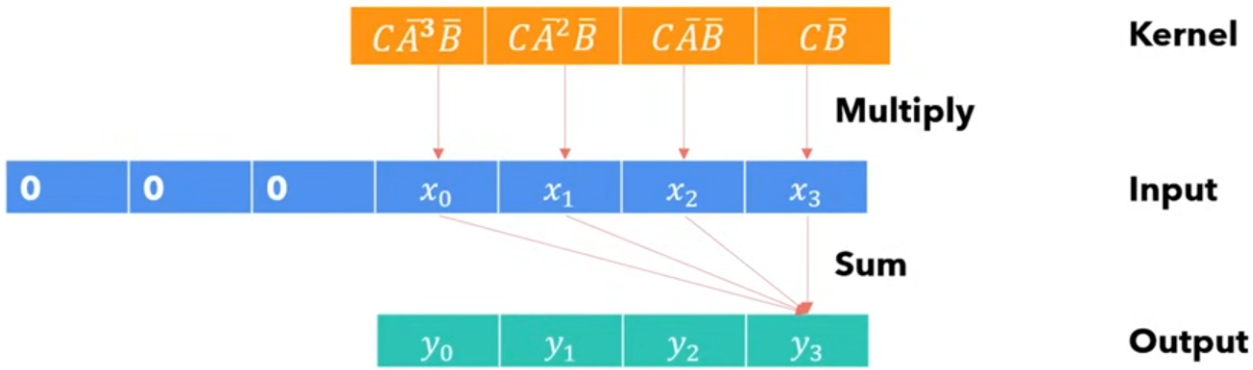
$$y_2 = C\bar{A}^2\bar{B}x_0 + C\bar{A}\bar{B}x_1 + C\bar{B}x_2$$

Convolutional formulation: step 4

$$y_k = C\bar{A}^k\bar{B}x_0 + C\bar{A}^{k-1}\bar{B}x_1 + \cdots + C\bar{A}\bar{B}x_{k-1} + C\bar{B}x_k$$

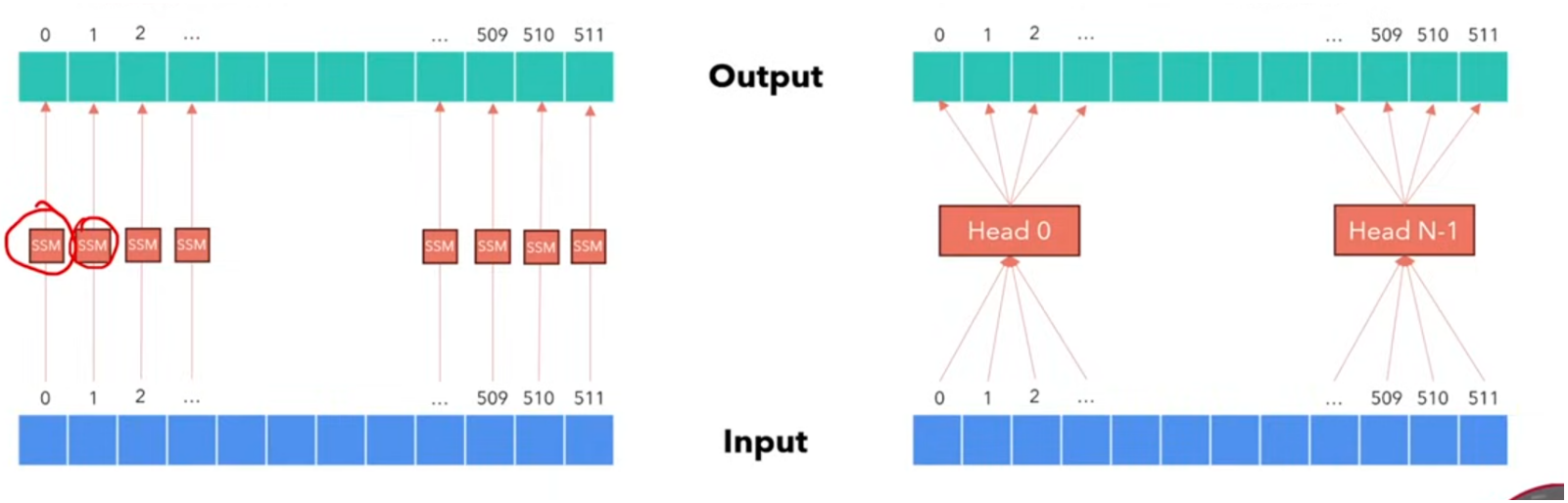
$$\overline{\boldsymbol{K}} = (C\overline{\boldsymbol{B}}, C\overline{\boldsymbol{AB}}, \ldots, C\overline{\boldsymbol{A}}^{\boldsymbol{k}}\overline{\boldsymbol{B}}, \ldots)$$

$$y = x * \overline{\boldsymbol{K}}$$



$$y_3 = C\bar{A}^3\bar{B}x_0 + C\bar{A}^2\bar{B}x_1 + C\bar{A}\bar{B}x_2 + C\bar{B}x_3$$

In scenarios where A, B, C, X_t, and y_t are vectors or matrices, each dimension is handled independently by its respective State Space Model.



Construction of matrix A

$\bar{A}$ encapsulates information from past states, essentially retaining the entire history of the input. Additionally, it influences the stability of the system. Consequently, the hidden state effectively captures local context, yet it may not preserve information from distant tokens, potentially leading to the loss of global context.

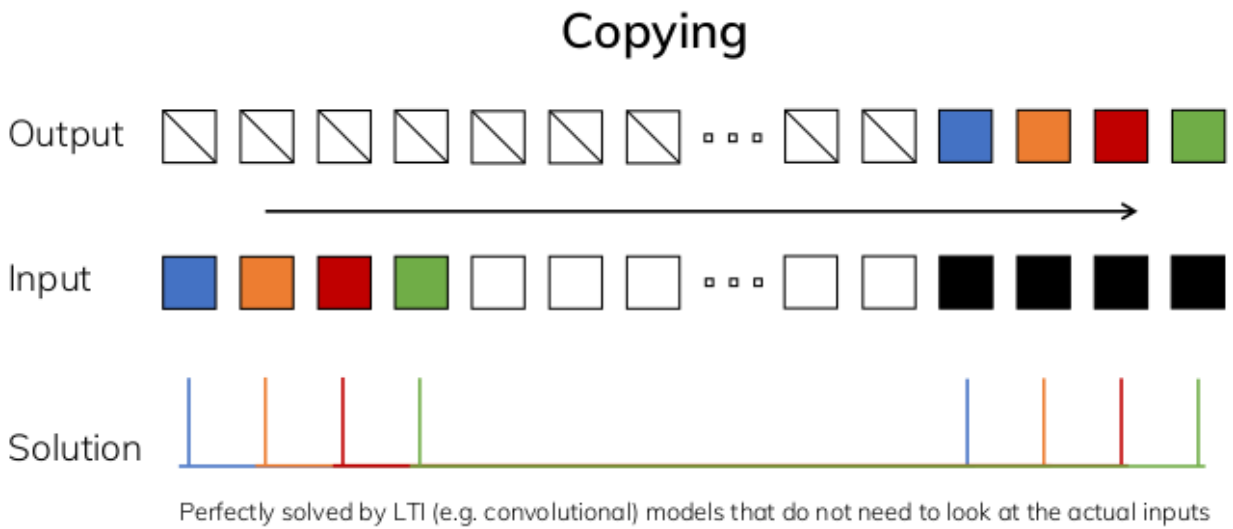
A matrix is constructed following the HIPPO theory, aiming to approximate the entirety of the input signal observed thus far into a vector of coefficients. These coefficients represent Legendre polynomials.

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

The distinction from Fourier Transformation lies in how the signal reconstruction is approached. With HIPPO theory in (SSM), instead of aiming to perfectly reconstruct the entire original signal, the emphasis is on accurately capturing the more recent signal while exponentially decaying the influence of older signals, akin to Exponential Moving Average (EMA). Consequently, the state $h(t)$ encapsulates more information about tokens observed more recently compared to older ones.

Existing SSMs limits

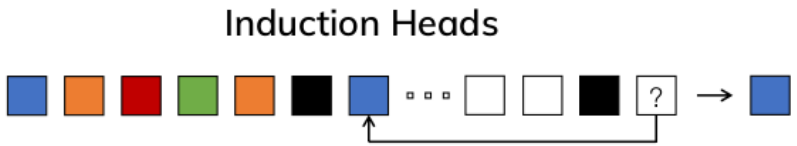
To replicate sequences, current SSMs (e.g. Vanilla SSM) rewrite the input token by token, albeit with a time-shifted approach (using copying methods). The time delay can be learned through convolution.



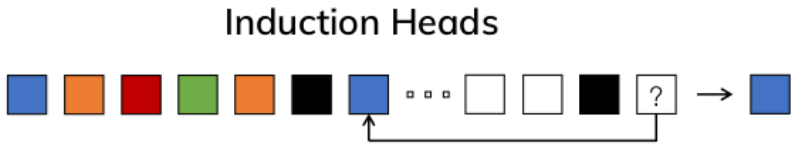
Current SSMs may exhibit deficiencies in :

- Selective copying
- Induction heads

Selective copying involves excluding non- relevant words (e.g., inappropriate words in a message).



Induction heads entail employing content-aware reasoning to generate the current token. For instance, during training, if the model encounters the color "green" after each occurrence of "red", during inference, when presented with the color "green", the model will predict the color "red" without explicit reasoning.



Mamba

Mamba is a selective SSM.

Algorithm 1 SSM (S4)

Input: $x : (\mathtt{B}, \mathtt{L}, \mathtt{D})$
Output: $y : (\mathtt{B}, \mathtt{L}, \mathtt{D})$

$\boldsymbol{A} : (\mathtt{D}, \mathtt{N}) \leftarrow$ Parameter
▷ Represents structured $N \times N$ matrix
$\boldsymbol{B} : (\mathtt{D}, \mathtt{N}) \leftarrow$ Parameter
$\boldsymbol{C} : (\mathtt{D}, \mathtt{N}) \leftarrow$ Parameter
$\Delta : (\mathtt{D}) \leftarrow \tau_\Delta(\text{Parameter})$
$\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}} : (\mathtt{D}, \mathtt{N}) \leftarrow \text{discretize}(\Delta, \boldsymbol{A}, \boldsymbol{B})$
$y \leftarrow \text{SSM}(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C})(x)$
▷ Time-invariant: recurrence or convolution
return y

Algorithm 2 SSM + Selection (S6)

Input: $x : (\mathtt{B}, \mathtt{L}, \mathtt{D})$
Output: $y : (\mathtt{B}, \mathtt{L}, \mathtt{D})$

$\boldsymbol{A} : (\mathtt{D}, \mathtt{N}) \leftarrow$ Parameter
▷ Represents structured $N \times N$ matrix
$\boldsymbol{B} : (\mathtt{B}, \mathtt{L}, \mathtt{N}) \leftarrow s_B(x)$
$\boldsymbol{C} : (\mathtt{B}, \mathtt{L}, \mathtt{N}) \leftarrow s_C(x)$
$\Delta : (\mathtt{B}, \mathtt{L}, \mathtt{D}) \leftarrow \tau_\Delta(\text{Parameter} + s_\Delta(x))$
$\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}} : (\mathtt{B}, \mathtt{L}, \mathtt{D}, \mathtt{N}) \leftarrow \text{discretize}(\Delta, \boldsymbol{A}, \boldsymbol{B})$
$y \leftarrow \text{SSM}(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C})(x)$
▷ Time-varying: recurrence (*scan*) only
return y

Algorithms 1 and 2 illustrates the main selection mechanism that we use. The main difference is simply making several parameters $\Delta, \boldsymbol{B}, \boldsymbol{C}$ functions of the input, along with the associated changes to tensor shapes throughout. In particular, we highlight that these parameters now have a length dimension L, meaning that the model has changed from time-invariant to time-varying. (Note that shape annotations were described in Section 2). This loses the equivalence to convolutions (3) with implications for its efficiency, discussed next.

Mamba (Linear RNN) properties :

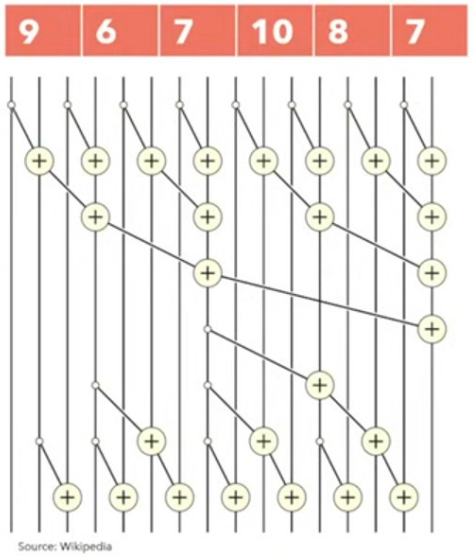
- The parameters B, C, and Δ depend on the input size
- Mamba cannot utilize convolutional computation due to the variable nature of its kernels (parameters), which are contingent upon the input size. The only viable option is to employ recurrent computation.
- Mamba utilizes **scan**, which resembles **Prefix-Sum**, where each state is the sum of the previous state and the current input.
- Parallel scan : Where operations using in scan are associative, the scan method can parallelized.

Parallel scan

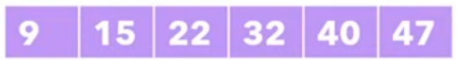
The parallel scan

What if I told you the scan operation can be parallelized? You would not believe me, but it can be! As long as the operations we are doing are associative (i.e. the operation benefits from the associative property). The associative property says simply that $A * B * C = (A * B) * C = A * (B * C)$, so the order in which we do operations does not matter.

Initial array



Prefix-Sum



We can spawn multiple threads to perform the binary operation in parallel, synchronizing at each step.
The time complexity instead of being $O(N)$ is reduced to $O(N/T)$ where T is the number of parallel threads.

Let's consider SSM recurrence formula. We will make the $\mathbf{x}_i$ and the $\mathbf{y}_i$ vectors, and $f_i(\mathbf{x}, \mathbf{y}) = A_i\mathbf{x}_i + B_i\mathbf{y}_i$ for index-dependent matrices A_i and B_i.

A priori, this is not an associative operation... But let's try anyway. We'll process the first half starting with some $\mathbf{y}_0$, and the second half starting with some $\mathbf{y}_{n/2-1}$. If we knew the real $\mathbf{y}_{n/2-1}$, it would be easy, but we don't. This doesn't really help us ; if we change $\mathbf{y}_{n/2-1}$, we have to recompute the scan for the second half.

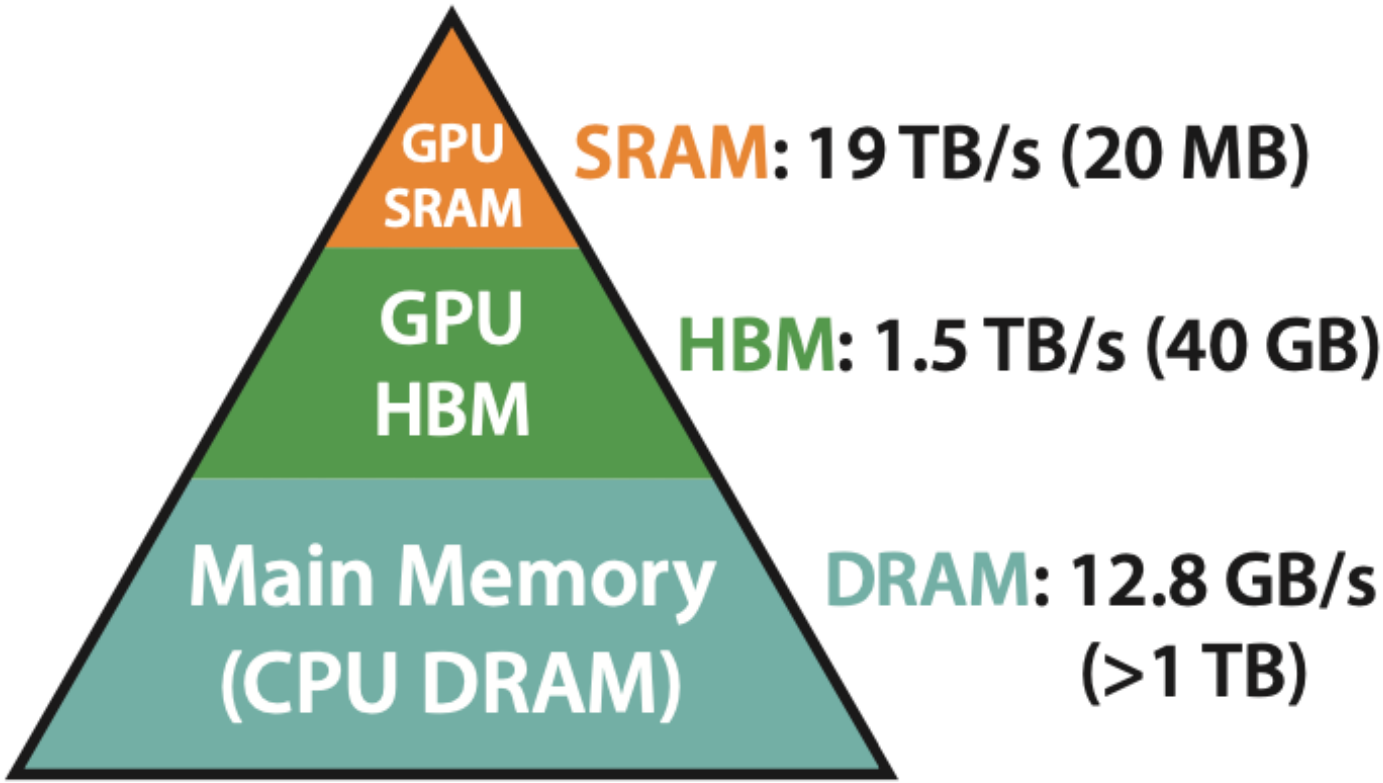
The trick is to lift the sequence $\mathbf{y}$. Notice that given the A_i, B_i, and $\mathbf{X}_i$, each $\mathbf{y}_i$ is a linear function of $\mathbf{y}_j$ for $j < i$. But we know how to represent linear functions between vectors : they are matrices. So if, instead of calculating the sequence of $\mathbf{y}_i$, we compute the sequence of $\mathbf{Y}_i$ which represents the function that gives $\mathbf{y}_i$ as a function of $\mathbf{y}_0$, then we are getting somewhere !

Indeed, we can compute the second half of the scan as $\mathbf{Y}_{n/2}, \mathbf{Y}_{n/2+1}, \ldots$ and, when the first half has completed, multiply those matrices by $\mathbf{y}_{n/2-1}$, which can be done in parallel.

GPU Memory hierarchy

The GPU comprises two types of memory :

- High Bandwidth Memory (HBM, in gigabytes) : Relatively slow
- SRAM (in megabytes) : Very fast



During matrix multiplication on the GPU, data is initially transferred from the DRAM to the SRAM. Subsequently, the GPU's core accesses the SRAM to perform computations, and the result is returned to the HBM. However, transferring data from the SRAM back to the HBM is not swift.

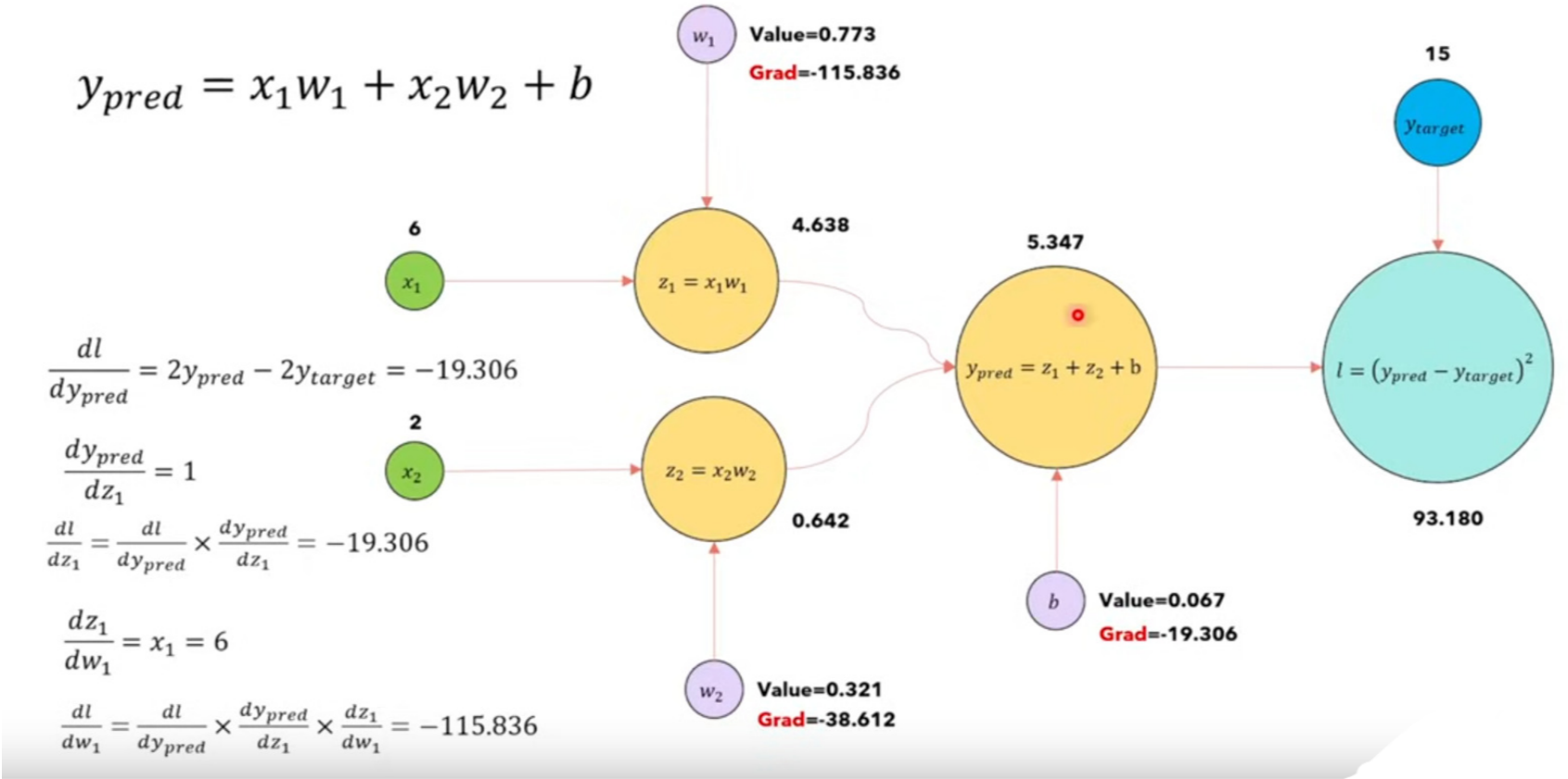
Instead of preparing the parallel scan input $(\bar{A}, \bar{B})$ of size (B, L, D, N) in HBM, Mamba loads parameters (Δ, A, B, C) from DRAM to SRAM, performs discretization and recurrence, and then writes the final output of size (B, L, D) back to DRAM.

Mamba kernel fusion

When a tensor sum is executed on a GPU, it first transfers the tensors from HBM to SRAM, performs the summation, and then transfers the result back to HBM. This process is repeated for subsequent summations with other tensors. Due to the relatively slow copying process, GPUs are inefficient in handling such operations. Mamba's kernel fusion technique circumvents the need for intermediate copies of results by conducting all operations within SRAM and only saving the final result in HBM.

Mamba recomputation

When training a deep learning model, it is converted into a computation graph. During backpropagation, to compute gradients at each node, we typically need to cache the output values from the forward step in HBM. However, this mechanism can consume a significant amount of memory. **Recomputing the values is faster than caching them in HBM and transferring them from SRAM afterward.**


$y_{pred} = x_1 w_1 + x_2 w_2 + b$
w_1
Value=0.773
Grad=-115.836
15
y_{target}
6
x_1
$z_1 = x_1 w_1$
4.638
5.347
$\frac{dl}{dy_{pred}} = 2y_{pred} - 2y_{target} = -19.306$
$y_{pred} = z_1 + z_2 + b$
$l = (y_{pred} - y_{target})^2$
2
x_2
$\frac{dy_{pred}}{dz_1} = 1$
$z_2 = x_2 w_2$
0.642
93.180
$\frac{dl}{dz_1} = \frac{dl}{dy_{pred}} \times \frac{dy_{pred}}{dz_1} = -19.306$
b
Value=0.067
Grad=-19.306
$\frac{dz_1}{dw_1} = x_1 = 6$
w_2
Value=0.321
Grad=-38.612
$\frac{dl}{dw_1} = \frac{dl}{dy_{pred}} \times \frac{dy_{pred}}{dz_1} \times \frac{dz_1}{dw_1} = -115.836$

The mambda block

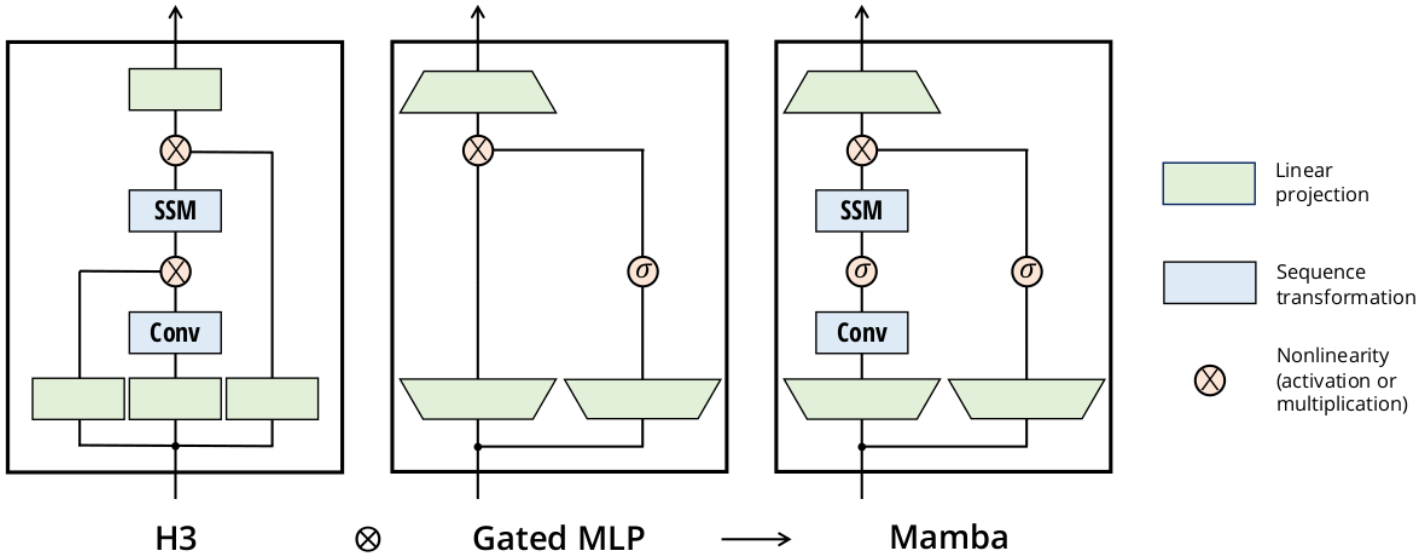


Figure 3: (**Architecture**.) Our simplified block design combines the H3 block, which is the basis of most SSM architectures, with the ubiquitous MLP block of modern neural networks. Instead of interleaving these two blocks, we simply repeat the Mamba block homogenously. Compared to the H3 block, Mamba replaces the first multiplicative gate with an activation function. Compared to the MLP block, Mamba adds an SSM to the main branch. For σ we use the SiLU / Swish activation (Hendrycks and Gimpel 2016; Ramachandran, Zoph, and Quoc V Le 2017).

Mamba limitations

- Mamba remains a recurrent neural network
- Its performance on large datasets is still unknown