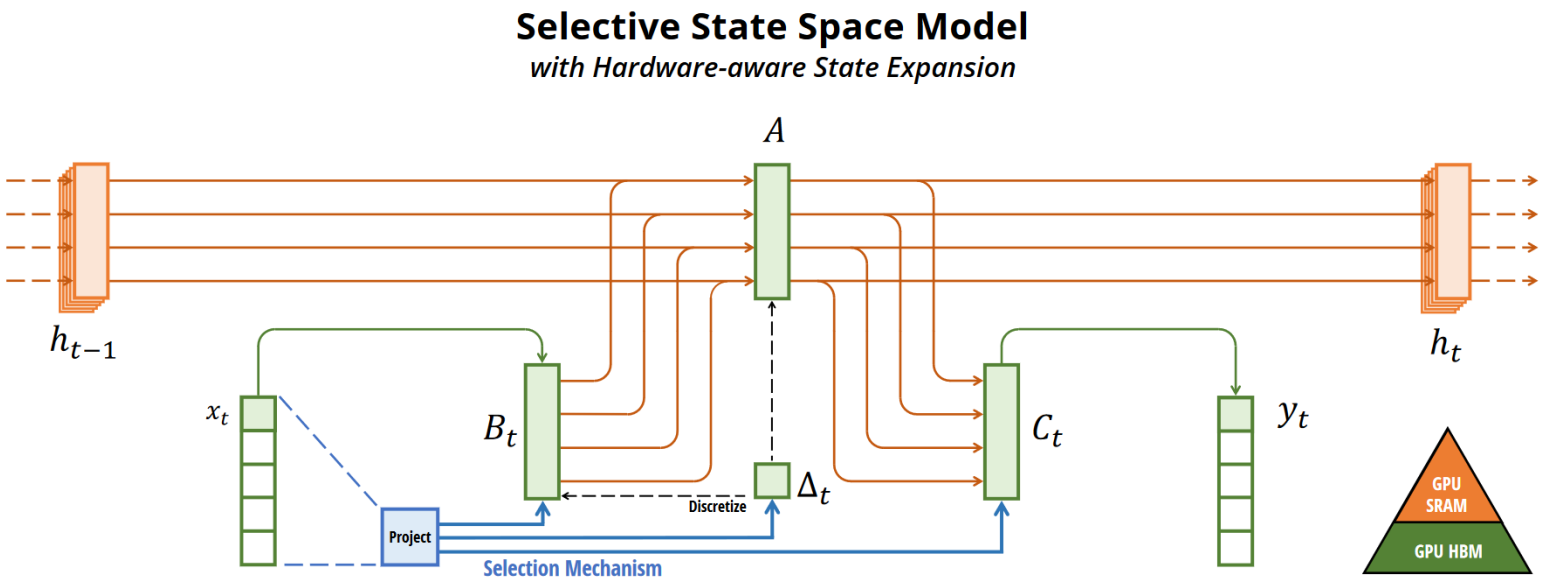


Mamba: Linear-Time Sequence Modeling with Selective State Spaces

Mamba/S4 explained

Umar Jamil

Topics

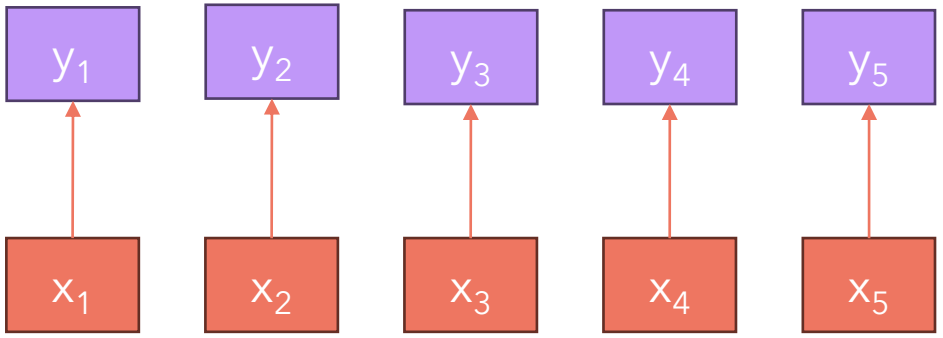
- Introduction to sequence models
- State space models
  - Fast track math course: differential equations
  - State space models
  - Discretization
  - Recurrent computation
  - Convolutional computation
  - From 1 dimension to multiple dimensions
  - The importance of the A matrix
  - The HIPPO matrix
- Mamba
  - Motivation
  - Selective Scan
    - What is the scan operation?
    - Parallel Scan
    - Kernel Fusion
    - Recomputation (of the activations)
  - Model's architecture
  - Performance of the model

Prerequisites

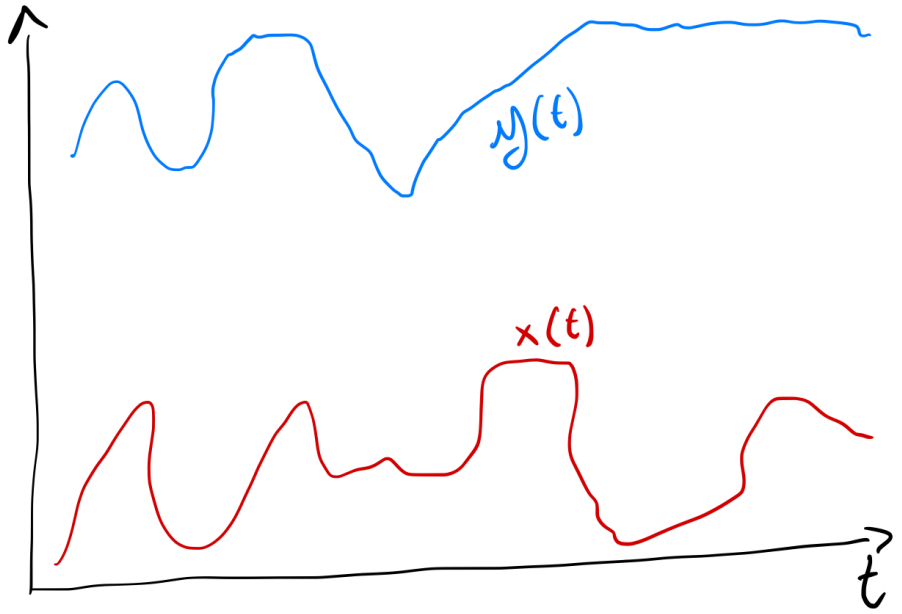
- Basics of calculus. High school mathematics will be enough.
- Basic understanding of the Transformer model (and neural networks in general)

Sequence modeling

The goal of a sequence model is to map an input sequence, to an output sequence. We can map a continuous input signal $x(t)$ to an output signal $y(t)$ or a discrete input sequence to a discrete output sequence.



Discrete signal

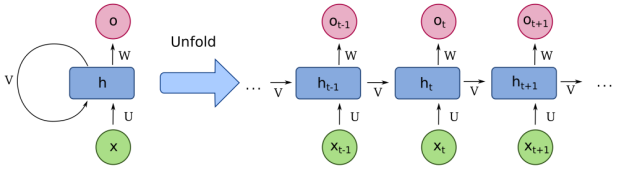


Continuous signal

We can choose among many models to do sequence modeling, let's review them.

Sequence modeling: models

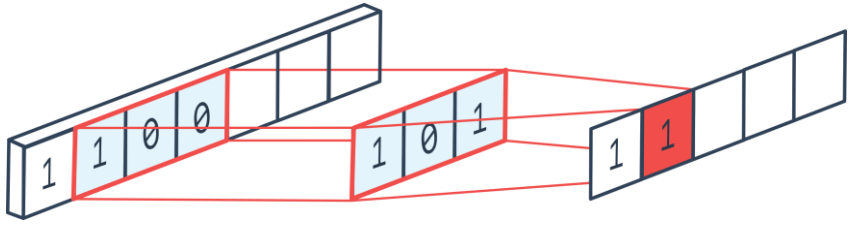
Theoretically with infinite context window
Practically suffer from vanishing/exploding gradient
Training: $O(N)$, **not parallelizable**
Inference: constant time for each token.



Source: Wikipedia

RNN

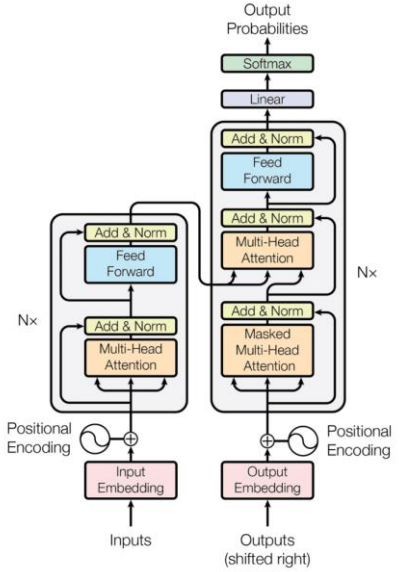
Finite context window (depending on kernel size)
Need to materialize the kernel before using it.
Training and inference depend on kernel size.
Easily parallelizable



Source: https://ai.stackexchange.com/questions/28767/what-does-channel-mean-in-the-case-of-an-1d-convolution

CNN

Finite context window
Training: $O(N^2)$, **easily parallelizable**
Inference: $O(N)$ when using KV-Cache, for each token. This means that if we want to generate the 10th token, we need to perform 10 dot product. To generate the 11th token, we will need to perform 11 dot products, etc.



Transformer

Sequence modeling: models

In an ideal world, we would like a model for which we can:

1. Parallelize the training (like the Transformer) and can scale linearly to long sequences (with a computation/memory cost of O(N) like the RNN)
2. Can inference each token with a constant computation/memory cost (O(1) like the RNN)…

Let's explore State Space Models (SSM)… but first, we need to review some maths.

(Simple) introduction to **differential** equations

Let's talk about differential equations with a very simple example. Imagine you have some bunnies, and the population grows at a constant rate of λ proportional to the number of bunnies, which means that every bunny will give birth to λ baby bunnies. So, we can say that the rate of change of the population of bunnies is the following:

$$(\text{number of baby bunnies born on a particular time step } t) = \lambda \times (\text{bunnies on the same time step } t)$$

$$(\text{rate of change w.r.t time}) = \lambda \times (\text{bunnies at time t})$$

$$\frac{db}{dt} = \lambda b(t)$$

$\frac{db}{dt}$ indicates the rate of change of the population of bunnies w.r.t time

$b(t)$ indicates how many bunnies we have at time t

How can we find the population at time $t = 100$, knowing that the population is made of 5 bunnies at $t = 0$? We need to find the $b(t)$ that describes the population of our bunnies over time. Solving a differential equation means to find a function $b(t)$ that makes the expression above true for all values of t. We can clearly verify that the solution is $b(t) = ke^{\lambda t}$, where $k = b(0) = 5$, our initial population of bunnies.

Usually, we represent a differential equation by omitting the variable t by writing it as follows: $\dot{b} = \lambda b$

We usually use differential equations to model the state of a system over time, with the goal of finding a function that gives us the state of the system at any time step, given the initial state of the system at time 0.

State space models

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

$$h'(t) = \boldsymbol{A}h(t) + \boldsymbol{B}x(t)$$
$$y(t) = \boldsymbol{C}h(t) + \boldsymbol{D}x(t)$$

This state space model is **linear** and **time invariant**. Linear because the relationships in the expressions above are linear, and time invariant because the parameter matrices **A**, **B**, **C**, **D** do not depend on time (they are fixed).

Note: for now consider A, B, C, D, x(t), h(t) and y(t) to be numbers, not vectors. Later we will extend our analysis to vectors.

To find the output signal $y(t)$ at time t, we first need to find a function $h(t)$ that describes the state of the system for all time steps. But that can be hard to solve analytically.
Usually we never work with continuous signals, but always with discrete ones (because we sample them), so how can we produce outputs $y(t)$ for a discrete signal? **We first need to discretize our system!**

Discretization

To solve a differential equation we need to find the function h(t) that makes the two hand sides of the equation equal, but most of the time it's hard to find the analytical solution of a differential equation, that's why we can approximate the solution of a differential equation. To find the approximate solution of a differential equation means to find a sequence of h(0), h(1), h(2), h(3), etc. that describe the evolution of our system over time. So instead of finding $h(t)$ we want to find $h(t_k) = h(k\Delta)$ where Δ is our step size.

Remember the bunnies problem? Let's try to find the approximate solution - which I remind you is a function $b(t)$ - using **Euler's method**!

1. First let's re-write our bunnies population model: $b'(t) = \boldsymbol{\lambda}\mathrm{b}(\mathrm{t})$
2. The derivative of a function is the rate of change of the function, that is: $\lim_{\Delta \to 0} \frac{b(t+\Delta)-b(t)}{\Delta} = b'(t)$. So, by choosing a small step size Δ we can get rid of the limit: $\frac{b(t+\Delta)-b(t)}{\Delta} \cong b'(t)$. By multiplying with Δ and moving terms around we can further write: $b(t+\Delta) \cong b'(t)\Delta + b(t)$
3. We can then plug the bunnies population model into the previous expression to obtain: $b(t+\Delta) \cong \boldsymbol{\lambda}\mathrm{b}(\mathrm{t})\Delta + b(t)$
4. We obtained a **recurrent** formulation!

Discretization

Wonderful! Let's use our recurrent formulation to approximate the state of the bunnies population over time: $b(t+\Delta) \cong \boldsymbol{\lambda} \mathrm{b}(\mathrm{t})\Delta + b(t)$

We set $\boldsymbol{\lambda} = 2, \Delta = 1$.

For example, if we started with a population of 5 bunnies at time $t = 0$, we can calculate the evolution of the population as follows:

- Knowing the population at time $t = 0$, we can calculate the population at time $t = 1$: $b(1) = \Delta \boldsymbol{\lambda} \mathrm{b}(0) + b(0) = 1 \times 2 \times 5 + 5 = 15$
- Knowing the population at time $t = 1$, we can calculate the population at time $t = 2$: $b(2) = \Delta \boldsymbol{\lambda} \mathrm{b}(1) + b(1) = 1 \times 2 \times 15 + 15 = 45$
- Knowing the population at time $t = 2$, we can calculate the population at time $t = 3$: $b(3) = \Delta \boldsymbol{\lambda} \mathrm{b}(2) + b(2) = 1 \times 2 \times 45 + 45 = 135$
- The smaller the step size Δ, the better the approximation w.r.t the analytical solution $b(t) = 5e^{\lambda t}$

Discretization

By using a similar reasoning to the bunny scenario, we can also discretize our state space model, so that we can calculate the evolution of the state over time by using a **recurrent** formulation.

1. By using the definition of derivative we know that: $h(t+\Delta) \cong \Delta h'(t) + h(t)$
2. This is the (continuous) state space model: $h'(t) = \boldsymbol{A}h(t) + \boldsymbol{B}x(t)$
3. We can substitute the state space model into the first expression to get the following

$$
\begin{aligned}
h(t+\Delta) &\cong \Delta\big(\boldsymbol{A}h(t) + \boldsymbol{B}x(t)\big) + h(t) \\
&= \Delta\boldsymbol{A}h(t) + \Delta\boldsymbol{B}x(t) + h(t) \\
&= (I + \Delta\boldsymbol{A})h(t) + \Delta\boldsymbol{B}x(t) \\
&= \overline{\boldsymbol{A}}h(t) + \overline{\boldsymbol{B}}x(t)
\end{aligned}
$$

Discretization

Wonderful! Now we have a recurrent formula that allows us to iteratively calculate the state of the system one step at a time, knowing the state at the previous time step. The matrices $\overline{\boldsymbol{A}}$ and $\overline{\boldsymbol{B}}$ are the **discretized** parameters of the model. This allows us to also calculate the output $y(t)$ of the system for discrete time steps.

$$h'(t) = \boldsymbol{A}h(t) + \boldsymbol{B}x(t) \quad (1a) \qquad h_t = \overline{\boldsymbol{A}}h_{t-1} + \overline{\boldsymbol{B}}x_t \quad (2a)$$

$$y(t) = \boldsymbol{C}h(t) \quad (1b) \qquad y_t = \boldsymbol{C}h_t \quad (2b)$$

In the paper instead of the Euler method they use the **Zero-Order Hold** (ZOH) rule to discretize the system.

> **Discretization.** The first stage transforms the "continuous parameters" $(\Delta, \boldsymbol{A}, \boldsymbol{B})$ to "discrete parameters" $(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}})$ through fixed formulas $\overline{\boldsymbol{A}} = f_A(\Delta, \boldsymbol{A})$ and $\overline{\boldsymbol{B}} = f_B(\Delta, \boldsymbol{A}, \boldsymbol{B})$, where the pair (f_A, f_B) is called a discretization rule. Various rules can be used such as the zero-order hold (ZOH) defined in equation (4).
>
> $$\overline{\boldsymbol{A}} = \exp(\Delta \boldsymbol{A}) \qquad \overline{\boldsymbol{B}} = (\Delta \boldsymbol{A})^{-1}(\exp(\Delta \boldsymbol{A}) - \boldsymbol{I}) \cdot \Delta \boldsymbol{B} \quad (4)$$

Note: in practice, we do not choose the discretization step Δ, but we make it a parameter of the model so that it can be learnt with gradient descent.

Recurrent computation

Now that we have our recurrent formula, how can we use it to calculate the output of the system for various time steps?

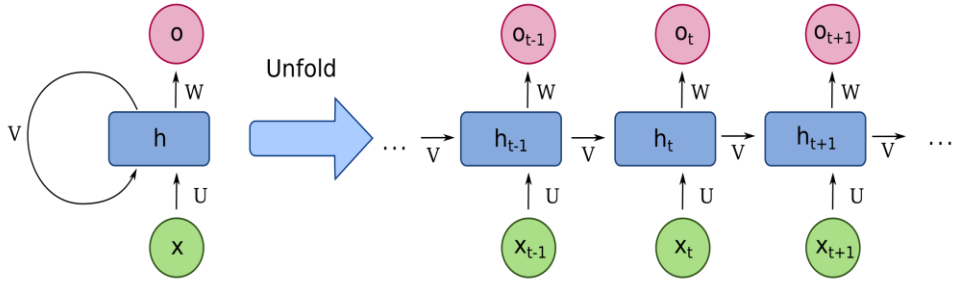
Suppose that, for simplicity, the initial state of the system is 0.

$$h_t = \overline{A}h_{t-1} + \overline{B}x_t$$
$$y_t = Ch_t$$

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

$$h_1 = \bar{A}h_0 + \bar{B}x_1$$
$$y_1 = Ch_1$$

$$h_2 = \bar{A}h_1 + \bar{B}x_2$$
$$y_2 = Ch_2$$



Source: Wikipedia

Recurrent computation: the problem

The recurrent formulation is great for inference, because we can compute one token at a time with a constant memory/computation requirement. This makes it suitable during inference in a Large Language Model, in which we want to generate one token at a time given the prompt and the previously generated tokens.

The recurrent formulation is not good for training, because during training we already have all the tokens of the input and the target, so we want to parallelize the computation as much as possible, just like the Transformer does!

Thankfully, State Space Models provide a convolutional mode as well, let's see how it works!

Convolutional computation

Let's expand the output we built before for every time step

$$h_t = \overline{A}h_{t-1} + \overline{B}x_t$$
$$y_t = Ch_t$$

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

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

$$h_2 = \bar{A}h_1 + \bar{B}x_2 = \bar{A}(\bar{A}\bar{B}x_0 + \bar{B}x_1) + \bar{B}x_2 = \bar{A}^2\bar{B}x_0 + \bar{A}\bar{B}x_1 + \bar{B}x_2$$
$$y_2 = Ch_2 = C(\bar{A}^2\bar{B}x_0 + \bar{A}\bar{B}x_1 + \bar{B}x_2) = C\bar{A}^2\bar{B}x_0 + C\bar{A}\bar{B}x_1 + C\bar{B}x_2$$

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

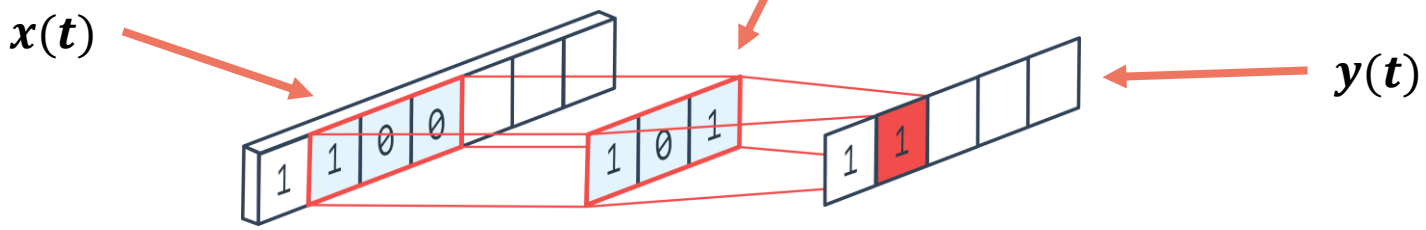
Convolutional computation

By using the formula we derived, we note something interesting: the output of the system can be calculated using a convolution of a kernel $\bar{K}$ with the input $x(t)$.

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

$$\overline{\boldsymbol{K}} = (C\overline{\boldsymbol{B}}, C\overline{\boldsymbol{AB}}, \dots, C\overline{\boldsymbol{A}}^{\boldsymbol{k}}\overline{\boldsymbol{B}}, \dots) \quad (3a)$$

$$y = x * \overline{\boldsymbol{K}} \quad (3b)$$



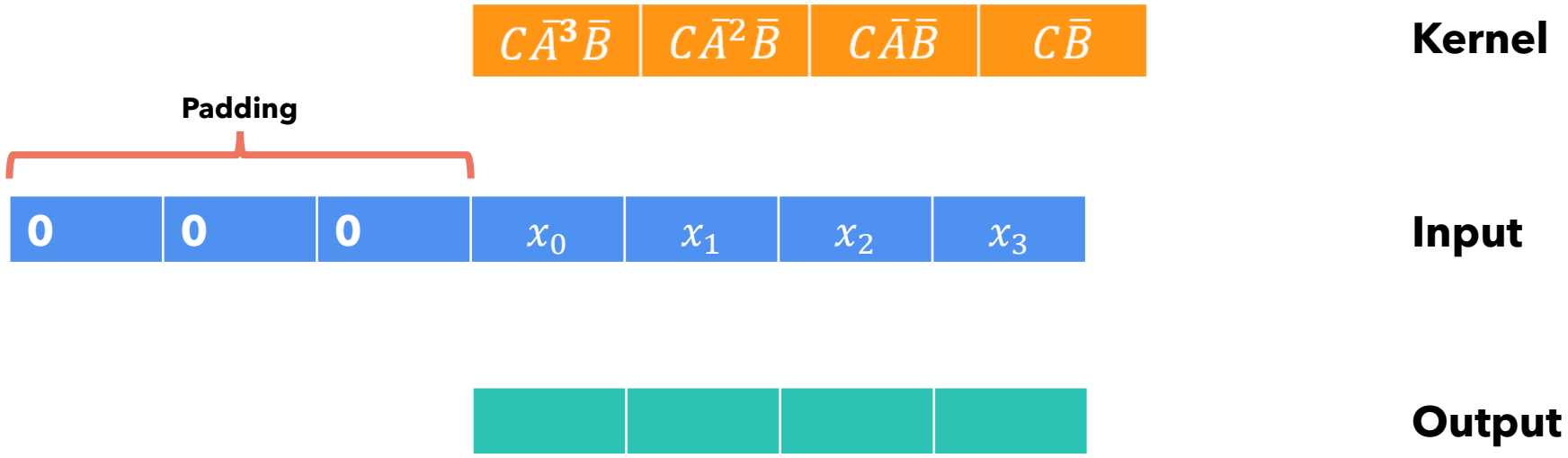
Let's prove it!

Convolutional formulation: create the kernel

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

$$\overline{\boldsymbol{K}} = (C\overline{\boldsymbol{B}}, C\overline{\boldsymbol{AB}}, \ldots, C\overline{\boldsymbol{A}}^{\boldsymbol{k}}\overline{\boldsymbol{B}}, \ldots) \quad (3a)$$

$$y = x * \overline{\boldsymbol{K}} \quad (3b)$$



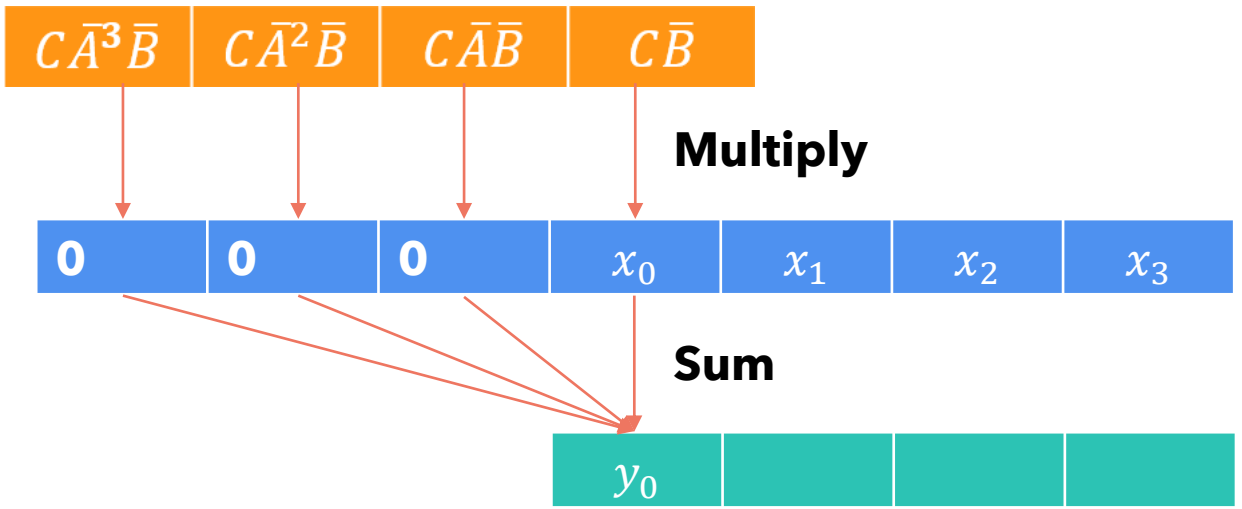
I am inverting the kernel just because it makes it easier to visualize, but the computation doesn't change.

Convolutional formulation: step 1

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

$$\overline{\boldsymbol{K}} = (C\overline{\boldsymbol{B}}, C\overline{\boldsymbol{AB}}, \ldots, C\overline{\boldsymbol{A}}^{\boldsymbol{k}}\overline{\boldsymbol{B}}, \ldots) \quad (3a)$$

$$y = x * \overline{\boldsymbol{K}} \quad (3b)$$



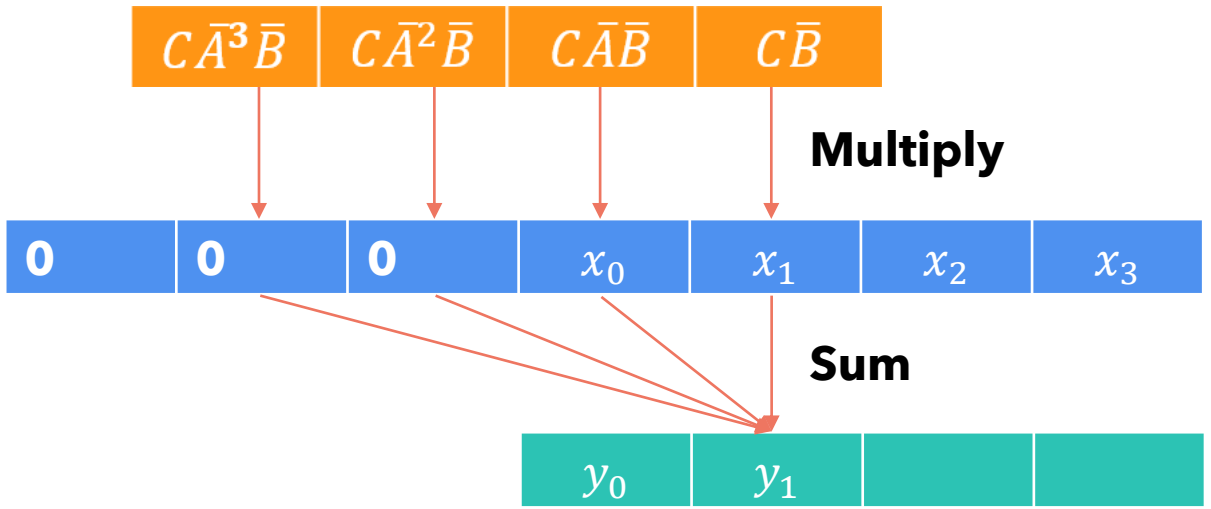
$$y_0 = C\bar{B}x_0$$

Convolutional formulation: step 2

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

$$\overline{\boldsymbol{K}} = (C\overline{\boldsymbol{B}}, C\overline{\boldsymbol{AB}}, \ldots, C\overline{\boldsymbol{A}}^{\boldsymbol{k}}\overline{\boldsymbol{B}}, \ldots) \quad (3a)$$

$$y = x * \overline{\boldsymbol{K}} \quad (3b)$$



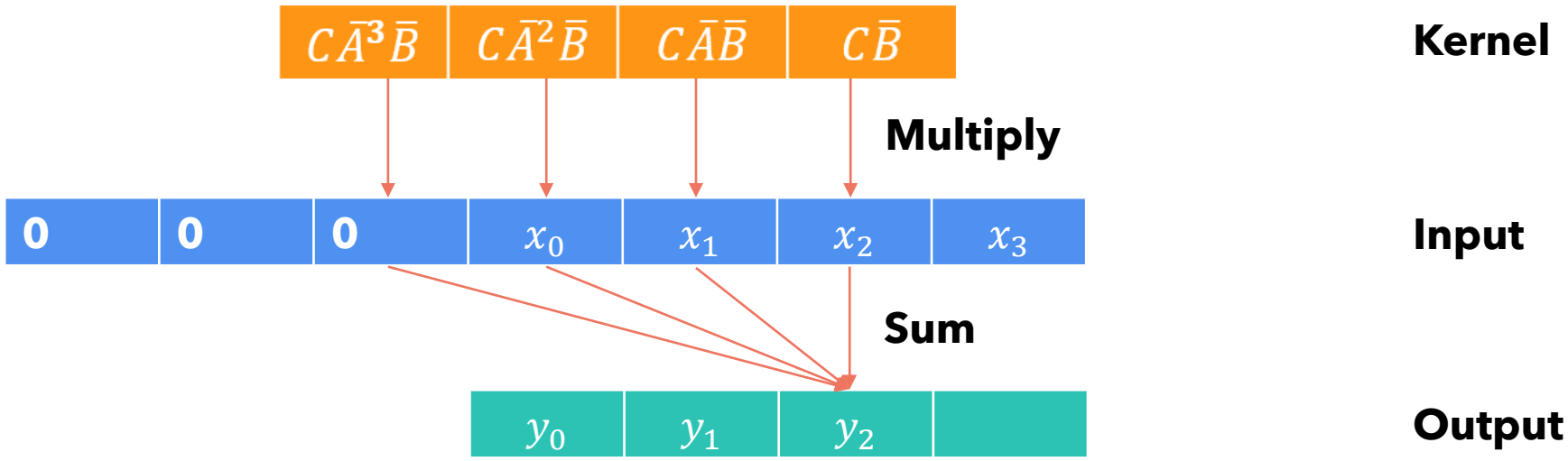
$$y_1 = C\bar{A}\bar{B}x_0 + C\bar{B}x_1$$

Convolutional formulation: step 3

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

$$\overline{\boldsymbol{K}} = (C\overline{\boldsymbol{B}}, C\overline{\boldsymbol{AB}}, \ldots, C\overline{\boldsymbol{A}}^{\boldsymbol{k}}\overline{\boldsymbol{B}}, \ldots) \quad (3a)$$

$$y = x * \overline{\boldsymbol{K}} \quad (3b)$$



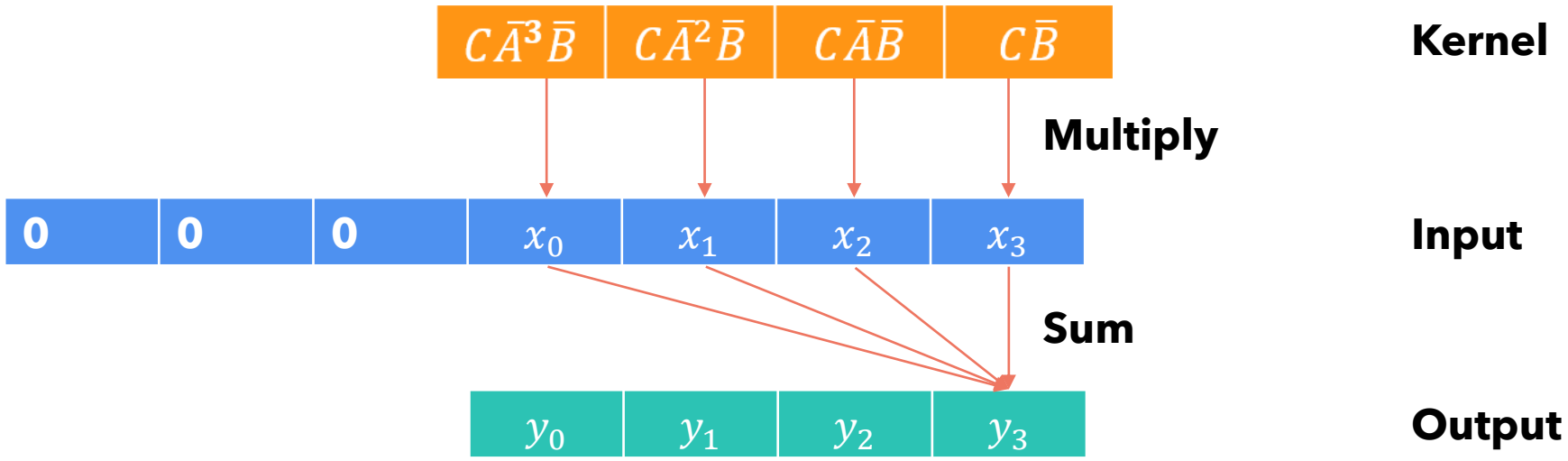
$$y_2 = C\bar{A}^2\bar{B}x_0 + C\bar{A}\bar{B}x_1 + C\bar{B}x_2$$

Convolutional formulation: step 4

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

$$\overline{\boldsymbol{K}} = (C\overline{\boldsymbol{B}}, C\overline{\boldsymbol{AB}}, \ldots, C\overline{\boldsymbol{A}}^{\boldsymbol{k}}\overline{\boldsymbol{B}}, \ldots) \quad (3a)$$

$$y = x * \overline{\boldsymbol{K}} \quad (3b)$$



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

Convolutional/Recurrent computation

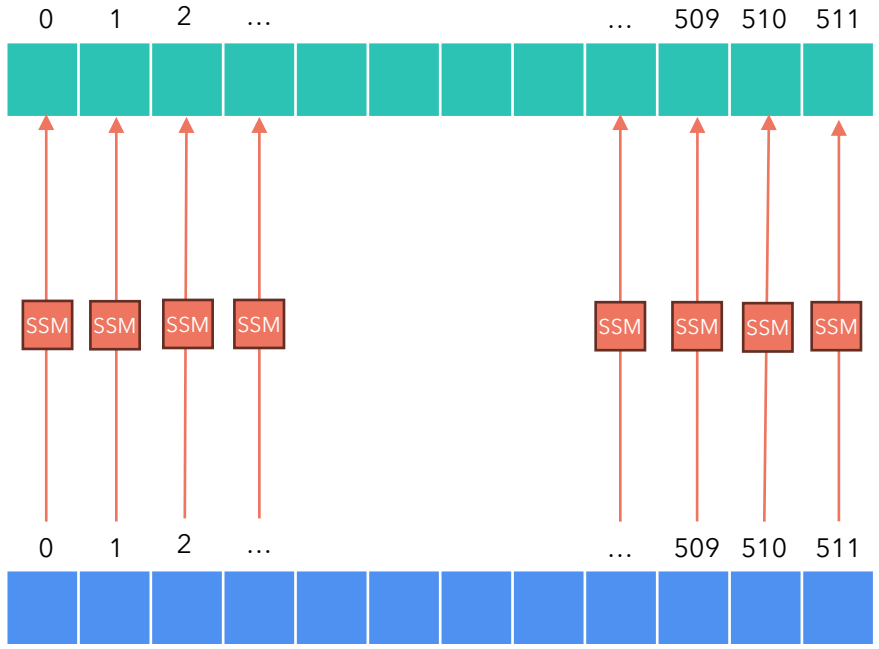
The best thing about the convolutional calculation is that it can be parallelized, because the output y_k does not depend on y_{k-1}, but only on the kernel and the input. However, materializing (building) the kernel can be computationally expensive, also from a memory point of view.

1. We can use the convolutional computation to perform training, because we already have all the input sequence of tokens, and it can be easily parallelized.
2. We can use the recurrent formulation to perform inference, one token at a time, using a constant amount of computation and memory for each token.

From 1 dimension to multiple dimensions

The state space model that we have studied so far, only computes one output (so only one number) for each input token (which is also represented by a single number). How can we work when the input/output signal is a vector?

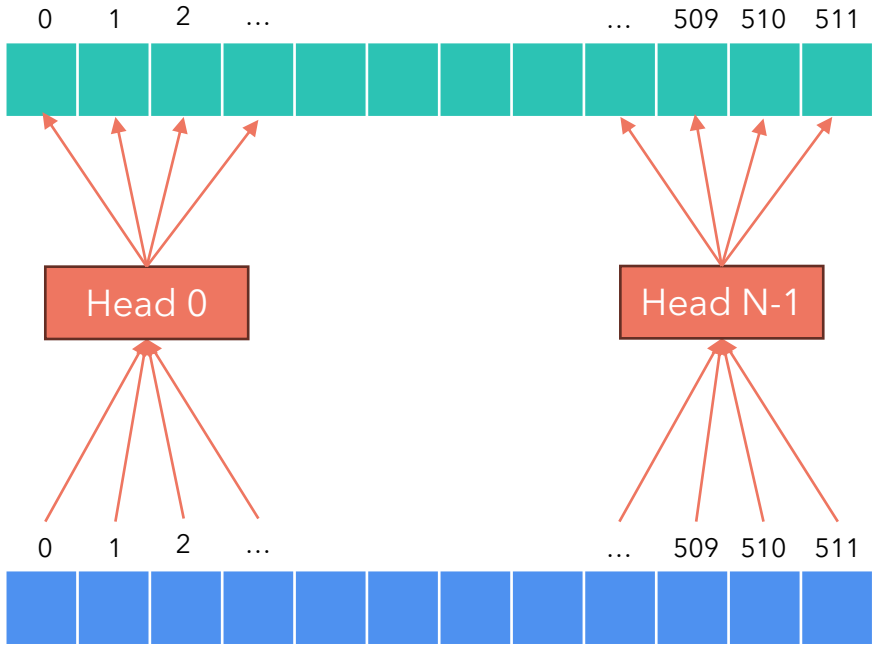
State Space Model: Each dimension is managed by an independent state space model!



Output

Input

Transformer: Each group of dimensions dimension is managed by a different head of the multi-head attention!



Of course we can parallelize all these operations by working on batches of input. This way, **the parameters A, B, C, D and the input x(t) and y(t) become vectors and matrices**. This way, the computation will be done in parallel for all the dimensions.

The importance of the A matrix

$$h_t = \overline{A}h_{t-1} + \overline{B}x_t$$
$$y_t = Ch_t$$

The **A** matrix in the state space model can be intuitively thought of as a matrix that "captures" information from the previous state to build the new state. It also decides how this information is copied forward in time.

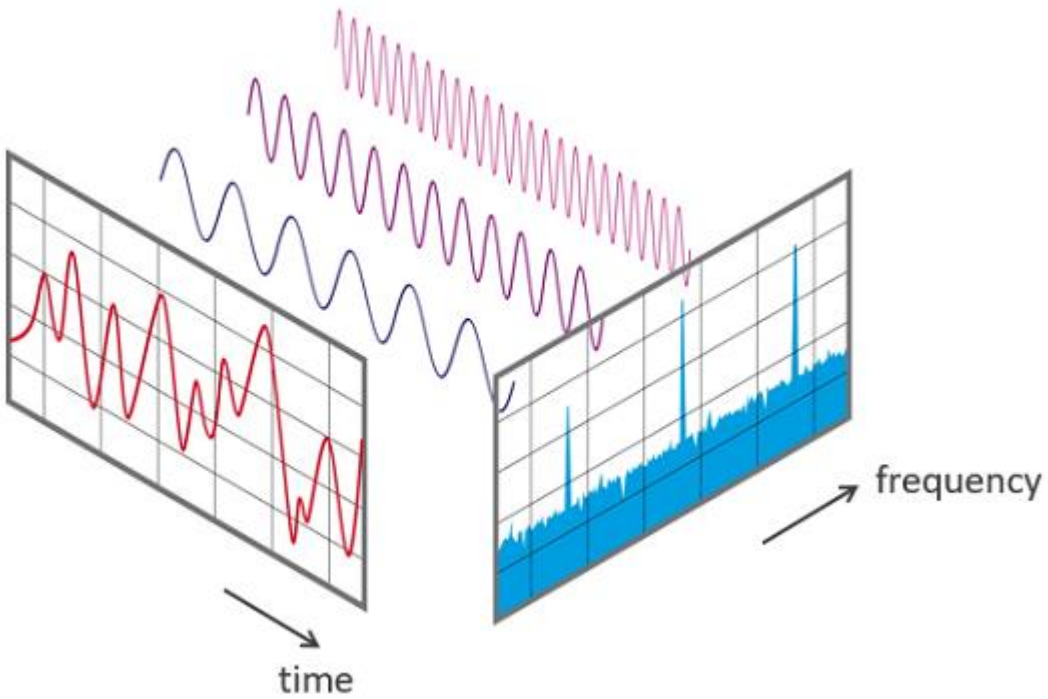
This means that we need to be careful about the structure of the A matrix, otherwise it may not capture well the history of all the inputs seen so far, which is needed to produce the next output. **This is very important for language models: the next token generated by a model should depend on the previous tokens (which constitute the prompt)!**

To make the A matrix behave well, the authors chose to use the HIPPO theory. **Let's see how it works!**

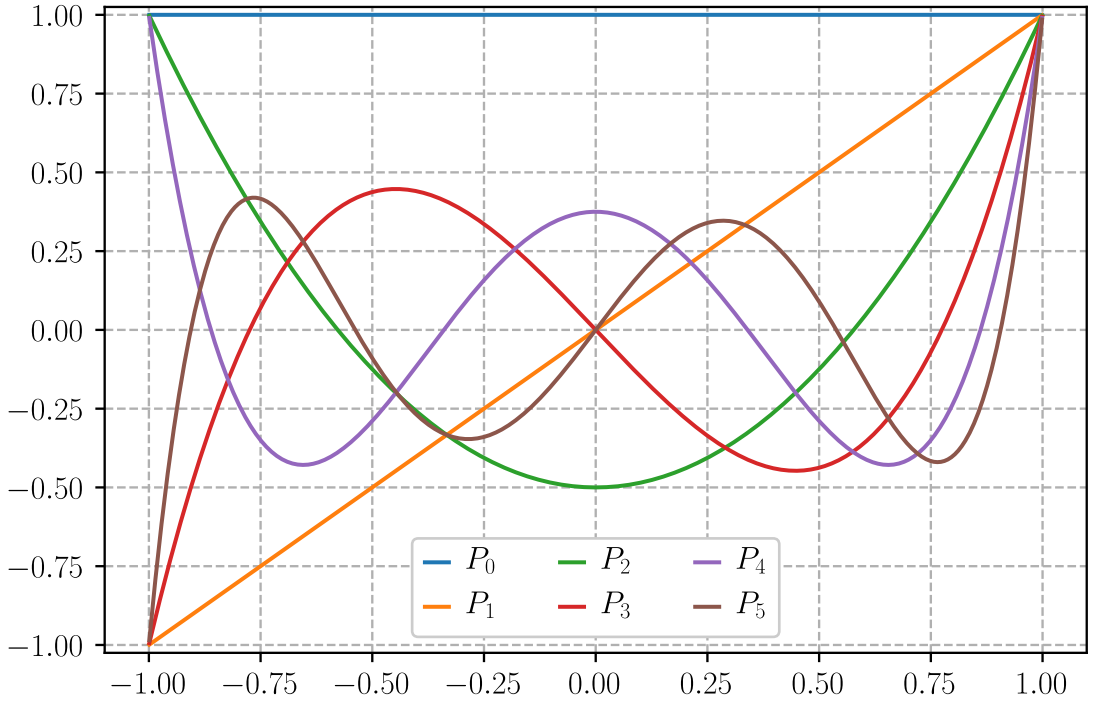
A little intuition from Fourier transformation

The Fourier transformation allows us to decompose a signal into sinusoidal functions, such that the sum of these functions approximates (well) the original signal.

With the HIPPO theory we do something similar, but instead of sinusoidal functions we use Legendre polynomials.



Source: https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft

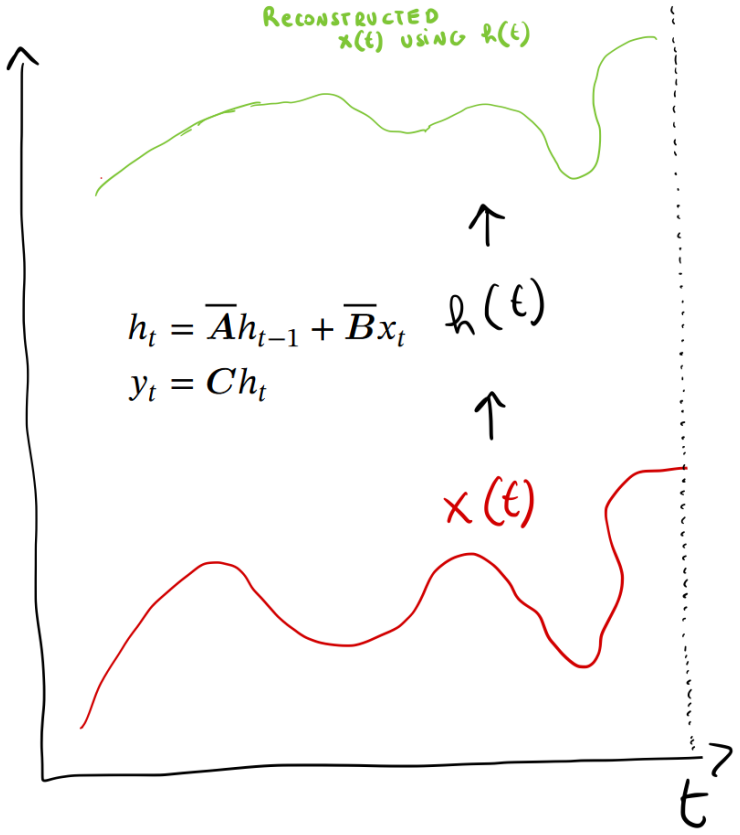


Source: Wikipedia

HIPPO in detail

With the HIPPO theory, we build the A matrix in such a way that it approximates all the input signal seen so far into a vector of coefficients (representing Legendre polynomials).

The difference with the Fourier transformation is that instead of building perfectly all the original signal so far, we build very precisely the more recent signal, while the older signal is decayed exponentially (like EMA). So, the state $h(t)$ captures more information about tokens seen more recently than more older ones.



2.2 Addressing Long-Range Dependencies with HiPPO

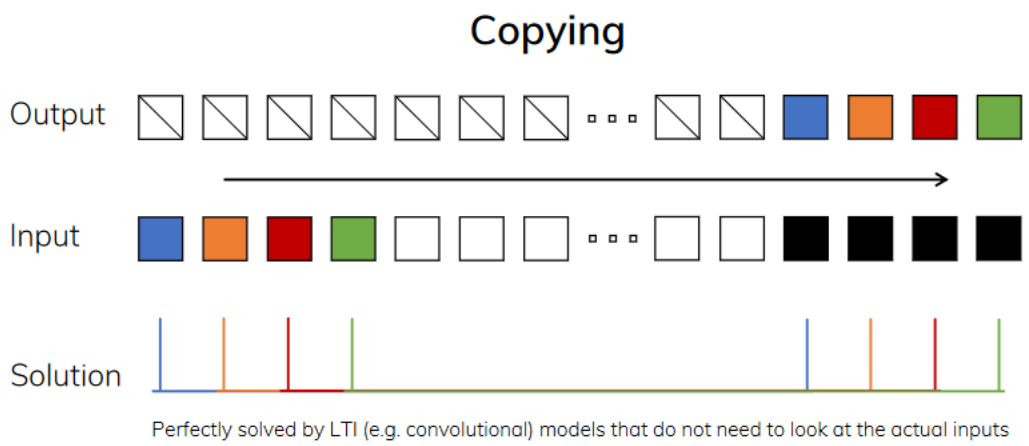
Prior work found that the basic SSM (1) actually performs very poorly in practice. Intuitively, one explanation is that linear first-order ODEs solve to an exponential function, and thus may suffer from gradients scaling exponentially in the sequence length (i.e., the vanishing/exploding gradients problem (Pascanu et al., 2013)). To address this problem, the LSSL leveraged the HiPPO theory of continuous-time memorization (Gu et al., 2020a). HiPPO specifies a class of certain matrices $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ that when incorporated into (1), allows the state $x(t)$ to memorize the history of the input $u(t)$. The most important matrix in this class is defined by equation (2), which we will call the HiPPO matrix. For example, the LSSL found that simply modifying an SSM from a random matrix $\boldsymbol{A}$ to equation (2) improved its performance on the sequential MNIST benchmark from 60% to 98%.

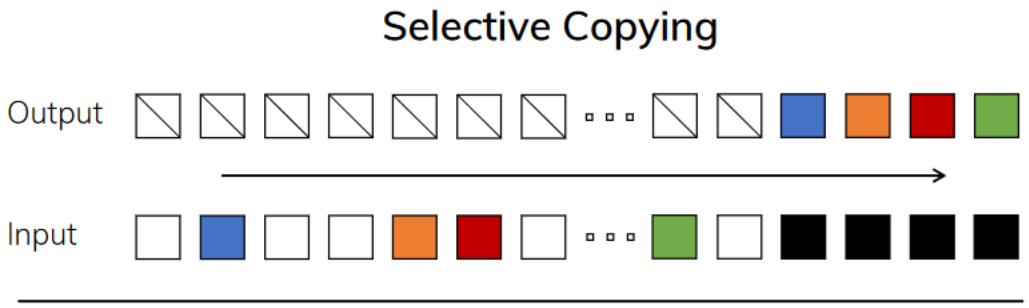
$$\textbf{(HiPPO Matrix)} \qquad \boldsymbol{A}_{nk} = -\begin{cases} (2n+1)^{1/2}(2k+1)^{1/2} & \text{if } n > k \\ n+1 & \text{if } n = k \\ 0 & \text{if } n < k \end{cases}. \tag{2}$$

Gu, A., Goel, K. and Ré, C., 2021. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*.

Mamba: the motivation (1)

The authors describe two tasks on which a vanilla SSM or even the S4 (Structured State Space Model) do not perform well, and this dictates the motivation behind mamba.





Intuition: rewrite the input one token at a time, but time-shifted.

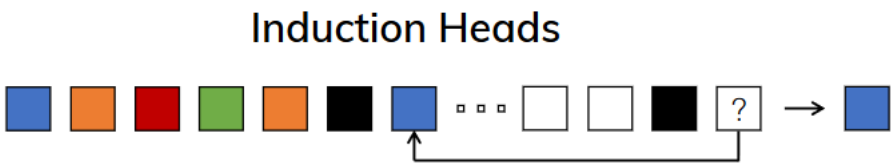
This **can** be performed by a vanilla SSM, and the time delay can be learnt by a convolution.

Intuition: given a comment on Twitter, rewrite the comment by removing all the bad words (the white tokens).

This **cannot** be performed by a vanilla SSM, because it requires content-aware reasoning, which vanilla SSM cannot do because they are time invariant (which means the parameters A, B, C, D are the same for every token it generates).

Mamba: the motivation (2)

The authors describe two tasks on which a vanilla SSM or even the S4 (Structured State Space Model) do not perform well, and this dictates the motivation behind mamba.



Intuition: for example with "few-shot" prompting we can "teach" LLMs new tasks and how to perform them. With Transformer-based model, this task can be done "easily", because Transformer-based models can attend to the previous tokens when generating the current one, so they can "recall previous history".

This task **cannot** be performed by a time-invariant SSM, because they cannot "select" which previous token to recall from their history.

Mamba: the motivation

The authors describe two tasks on which a vanilla SSM or even the S4 (Structured State Space Model) does not perform well, and this dictates the motivation behind mamba.

- The Selective Copying task modifies the popular Copying task (Arjovsky, Shah, and Bengio 2016) by varying the position of the tokens to memorize. It requires content-aware reasoning to be able to memorize the relevant tokens (colored) and filter out the irrelevant ones (white).
- The Induction Heads task is a well-known mechanism hypothesized to explain the majority of in-context learning abilities of LLMs (Olsson et al. 2022). It requires context-aware reasoning to know when to produce the correct output in the appropriate context (black).

These tasks reveal the failure mode of LTI models. From the recurrent view, their constant dynamics (e.g. the $(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}})$ transitions in (2)) cannot let them select the correct information from their context, or affect the hidden state passed along the sequence an in input-dependent way. From the convolutional view, it is known that global convolutions can solve the vanilla Copying task (Romero et al. 2021) because it only requires time-awareness, but that they have difficulty with the Selective Copying task because of lack of content-awareness (Figure 2). More concretely, the spacing between inputs-to-outputs is varying and cannot be modeled by static convolution kernels.

Mamba: a selective SSM

Algorithm 1 SSM (S4)

Input: $x : (\mathtt{B}, \mathtt{L}, \mathtt{D})$
Output: $y : (\mathtt{B}, \mathtt{L}, \mathtt{D})$
1: $\boldsymbol{A} : (\mathtt{D}, \mathtt{N}) \leftarrow$ Parameter
▷ Represents structured $N \times N$ matrix
2: $\boldsymbol{B} : (\mathtt{D}, \mathtt{N}) \leftarrow$ Parameter
3: $\boldsymbol{C} : (\mathtt{D}, \mathtt{N}) \leftarrow$ Parameter
4: $\Delta : (\mathtt{D}) \leftarrow \tau_\Delta(\text{Parameter})$
5: $\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}} : (\mathtt{D}, \mathtt{N}) \leftarrow \text{discretize}(\Delta, \boldsymbol{A}, \boldsymbol{B})$
6: $y \leftarrow \text{SSM}(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C})(x)$
▷ Time-invariant: recurrence or convolution
7: **return** y

$$h_t = \overline{A} h_{t-1} + \overline{B} x_t$$
$$y_t = C h_t$$

The $\bar{A}$ matrix also depends on the input, through Δ

Algorithm 2 SSM + Selection (S6)

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

$$s_B(x) = \text{Linear}_N(x),\ s_C(x) = \text{Linear}_N(x),\ s_\Delta(x) = \text{Broadcast}_D(\text{Linear}_1(x)),\ \text{and}\ \tau_\Delta = \text{softplus}$$

Mamba cannot be evaluated using a convolution, because the model's parameters differ for each input token, and even if we wanted to run a convolution, we would need to build L (sequence length) different convolutional kernels, which is just crazy from a memory/computation point of view.

Have you noticed that the authors talk about the "**scan**" operation when evaluating the recurrence? Let's talk about it!

B: Batch Size
L: Sequence Length
D: Size of the input vector (equivalent to *d_model* in the Transformer)
N: Size of the hidden state h.

The scan operation

If you have ever done competitive programming, you're familiar with the Prefix-Sum array, which is an array calculated sequentially, such that the value at each position indicates the sum of all the previous values. We can easily compute it with a for loop in linear time.

Initial array	9	6	7	10	8	7
Prefix-Sum	9	15	22	32	40	47

The scan operation refers to computing an array like the Prefix-Sum, in which each value can be computed using the previously computed value and the current input.

The recurrent formula of the SSM model can also be thought of as a scan operation, in which each state is the sum of the previous state and the current input.

Model input	x_0	x_1	x_2	x_3	x_4	x_5
Scan output	h_0	h_1	h_2	h_3	h_4	h_5

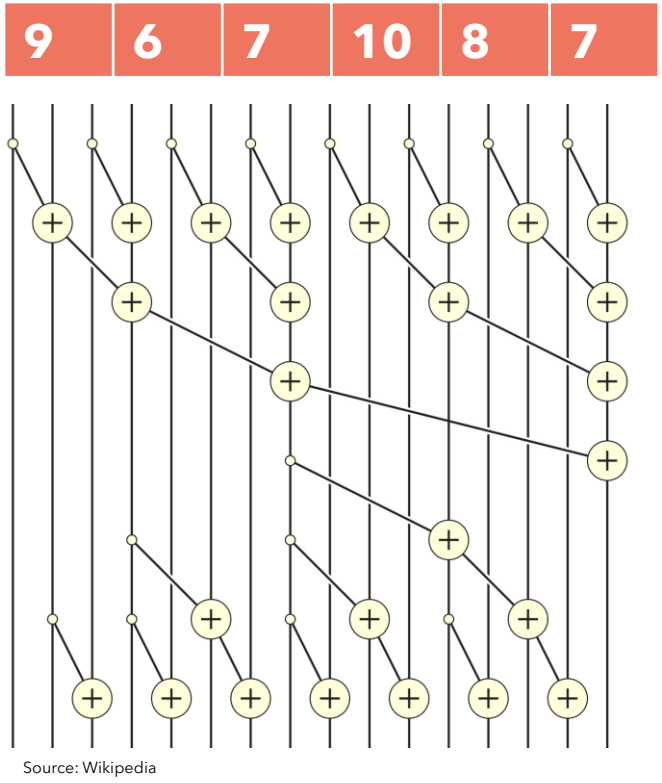
$$h_t = \overline{A}h_{t-1} + \overline{B}x_t$$
$$y_t = Ch_t$$

To generate the output, we just multiply each h_k with the matrix C to generate the output token y_k

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



Source: Wikipedia

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.

Mamba: Selective Scan

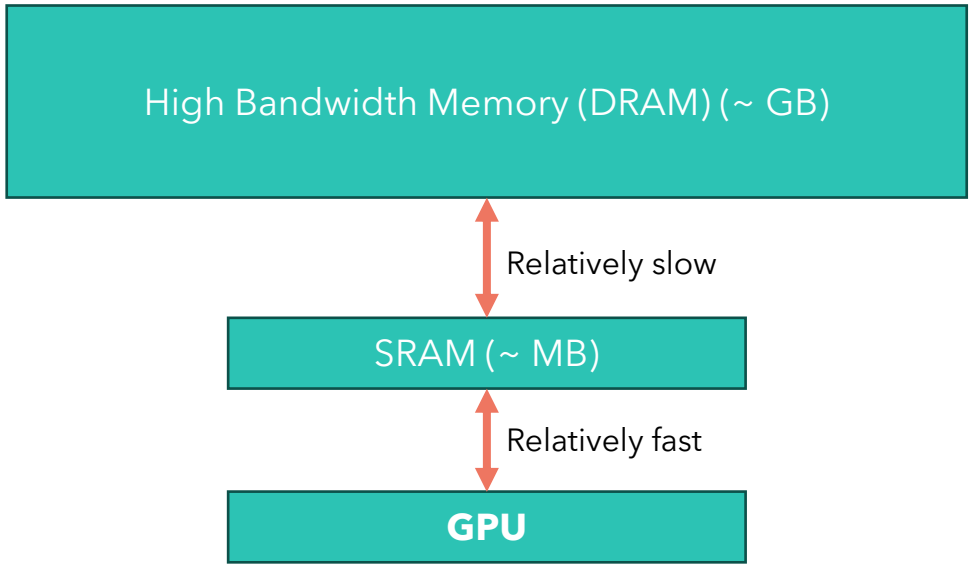
Since Mamba cannot be evaluated using a convolution (because it's time-varying), it cannot be parallelized. Our only way to calculate is to use the recurrent formulation, but thanks to the parallel scan algorithm it can be parallelized. The authors also indicate some techniques they use to make this algorithm faster

3.3.2 Overview of Selective Scan: Hardware-Aware State Expansion

The selection mechanism is designed to overcome the limitations of LTI models; at the same time, we therefore need to revisit the computation problem of SSMs. We address this with three classical techniques: kernel fusion, parallel scan, and recomputation. We make two main observations:

Let's see all these techniques one by one… but first let's see the memory hierarchy of the GPU

The GPU memory hierarchy



Because the GPU is not so fast at moving tensor around, but it's super fast at computing operations, sometimes, the problem in our algorithm may not be the number of computations we do, but how many tensors we move around in the different memory hierarchies. In this case, we say that the operation is IO-bound.

NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs		NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs	

* With sparsity

** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

Exploiting the memory hierarchy

The main idea is to leverage properties of modern accelerators (GPUs) to materialize the state h only in more efficient levels of the memory hierarchy. In particular, most operations (except matrix multiplication) are bounded by memory bandwidth (Dao, Fu, Ermon, et al. 2022; Ivanov et al. 2021; Williams, Waterman, and Patterson 2009). This includes our scan operation, and we use kernel fusion to reduce the amount of memory IOs, leading to a significant speedup compared to a standard implementation.

Concretely, instead of preparing the scan input $(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}})$ of size $(\mathtt{B}, \mathtt{L}, \mathtt{D}, \mathtt{N})$ in GPU HBM (high-bandwidth memory), we load the SSM parameters $(\Delta, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ directly from slow HBM to fast SRAM, perform the discretization and recurrence in SRAM, and then write the final outputs of size $(\mathtt{B}, \mathtt{L}, \mathtt{D})$ back to HBM.

Mamba: Kernel fusion

When we perform a tensor operation, our deep learning framework (e.g. PyTorch) loads the tensor in the fast memory (SRAM) of the GPU, performs the operation (e.g. matrix multiplication), and then saves back the result in the High-Bandwidth Memory of the GPU.

What if we do multiple operations on the same tensor (e.g. 3 operations)? Then the deep learning framework would perform the following sequence:

1. Load the input from HBM to SRAM, compute the first operation (CUDA kernel corresponding to the first operation) and then save back the result to HBM
2. Load the previous result from HBM to SRAM, compute the second operation (CUDA kernel corresponding to the second operation) and then save back the result to HBM
3. Load the previous result from HBM to SRAM, compute the third operation (CUDA kernel corresponding to the third operation) and then save back the result to SRAM.

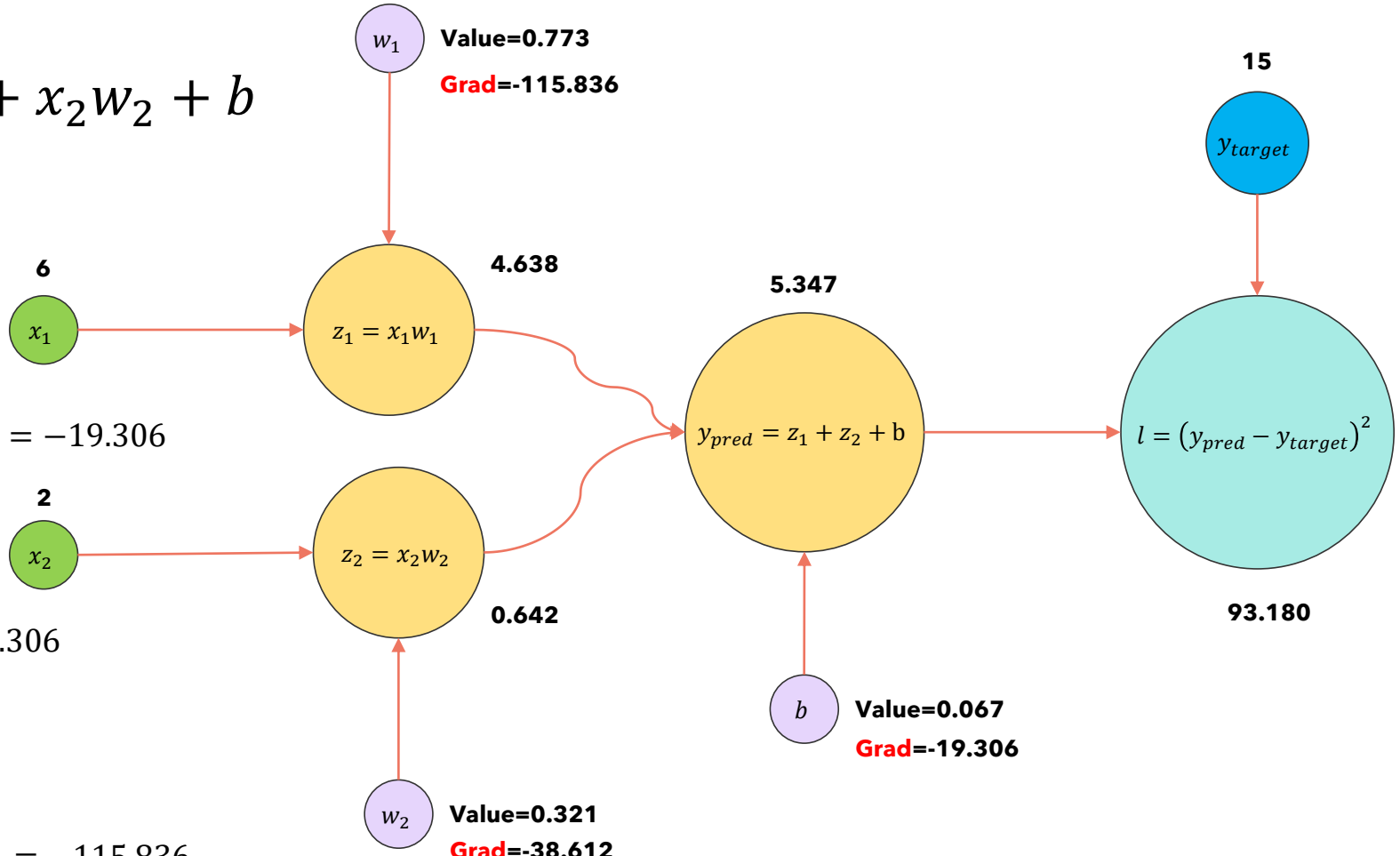
As you can see, the total time is occupied by the copying operations that we are performing, since we know that GPUs are relatively slow at copying data than they are at computing operations.

To make a sequence of operations faster, we can fuse the CUDA kernels to produce one custom CUDA kernel that does the three operations one after another without copying the intermediate results to the HBM and only copying the final results.

Mamba: Recomputation (1)

When we train a deep learning model, it gets converted into a computation graph. When we perform backpropagation, in order to calculate the gradients at each node, we need to cache the output values of the forward step as shown below

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



$$\frac{dl}{dy_{pred}} = 2y_{pred} - 2y_{target} = -19.306$$

$$\frac{dy_{pred}}{dz_1} = 1$$

$$\frac{dl}{dz_1} = \frac{dl}{dy_{pred}} \times \frac{dy_{pred}}{dz_1} = -19.306$$

$$\frac{dz_1}{dw_1} = x_1 = 6$$

$$\frac{dl}{dw_1} = \frac{dl}{dy_{pred}} \times \frac{dy_{pred}}{dz_1} \times \frac{dz_1}{dw_1} = -115.836$$

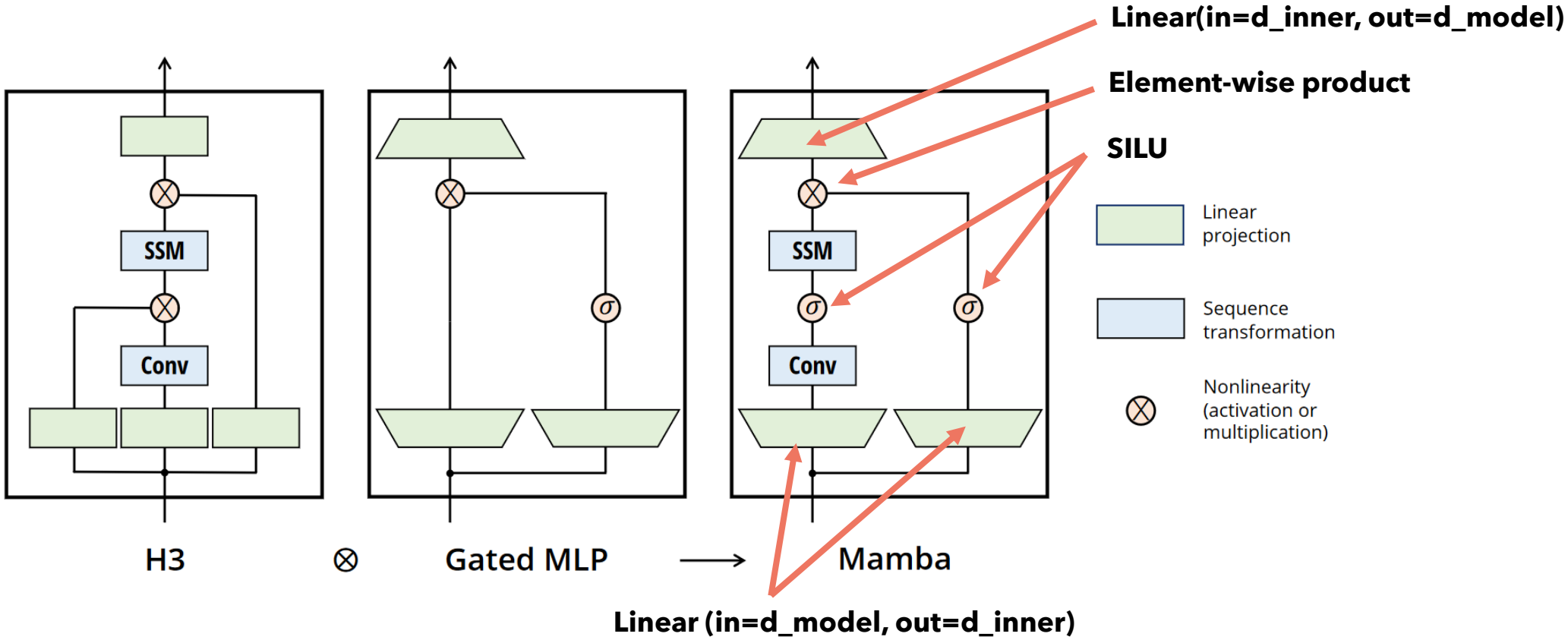
Mamba: Recomputation (2)

Since caching the activations and then reusing them during back-propagation means that we need to save them to the HBM and then copy them back from the HBM during backpropagation, it may be faster to just recompute them during backpropagation!

Finally, we must also avoid saving the intermediate states, which are necessary for backpropagation. We carefully apply the classic technique of recomputation to reduce the memory requirements: the intermediate states are not stored but recomputed in the backward pass when the inputs are loaded from HBM to SRAM. As a result, the fused selective scan layer has the same memory requirements as an optimized transformer implementation with FlashAttention.

Mamba: the Mamba Block(1)

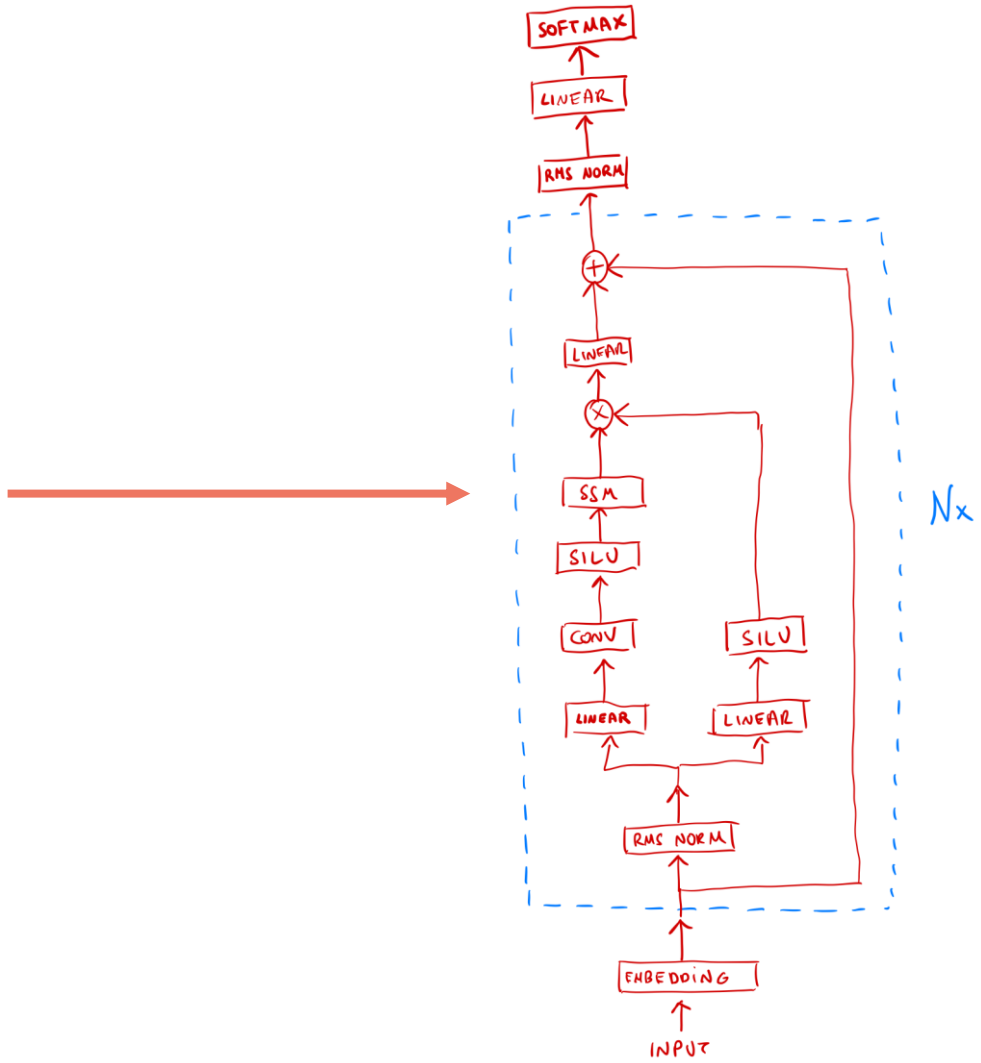
Mamba is built by stacking multiple layers of the Mamba block, shown below. This is very similar to the stacked layers of the Transformer model. The Mamba architecture derives from the *Hungry Hungry Hippo* (H3) architecture.



Now let's see the entire architecture of Mamba!

Mamba: the model architecture (2)

This is where we run the SSM



Mamba: the performance (1)

Model	Arch.	Layer	Acc.
S4	No gate	S4	18.3
-	No gate	S6	**97.0**
H3	H3	S4	57.0
Hyena	H3	Hyena	30.1
-	H3	S6	**99.7**
-	Mamba	S4	56.4
-	Mamba	Hyena	28.4
Mamba	Mamba	S6	**99.8**

Table 1: (**Selective Copying**.) Accuracy for combinations of architectures and inner sequence layers.

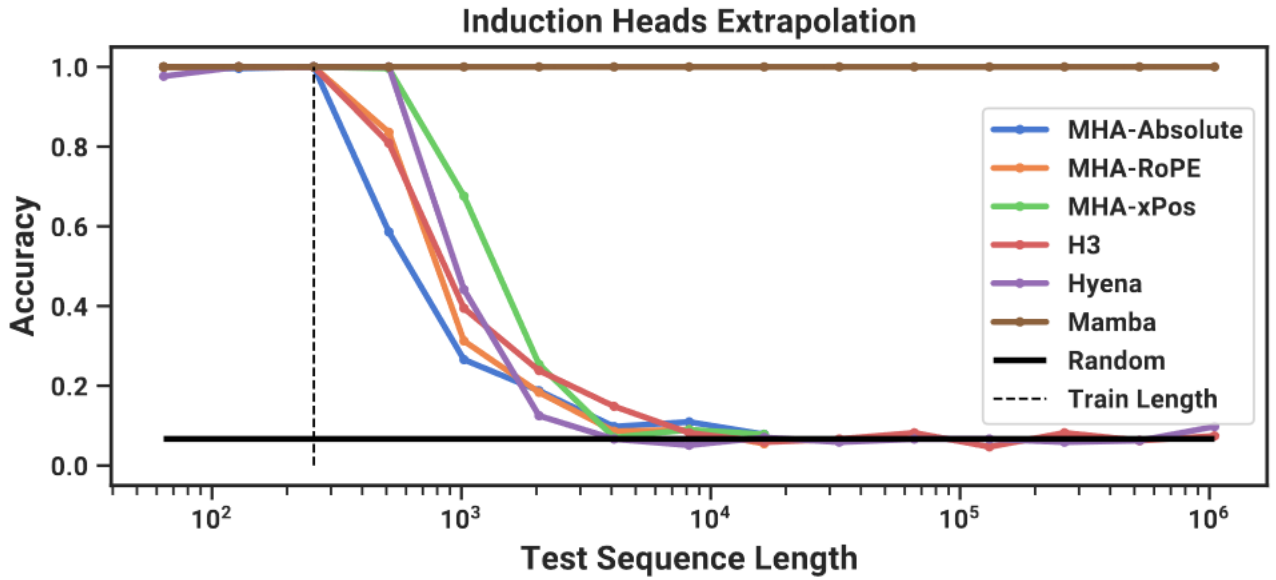


Table 2: (**Induction Heads**.) Models are trained on sequence length $2^8 = 256$, and tested on increasing sequence lengths of $2^6 = 64$ up to $2^{20} = 1048576$. Full numbers in Table 11.

Mamba: the performance (2)

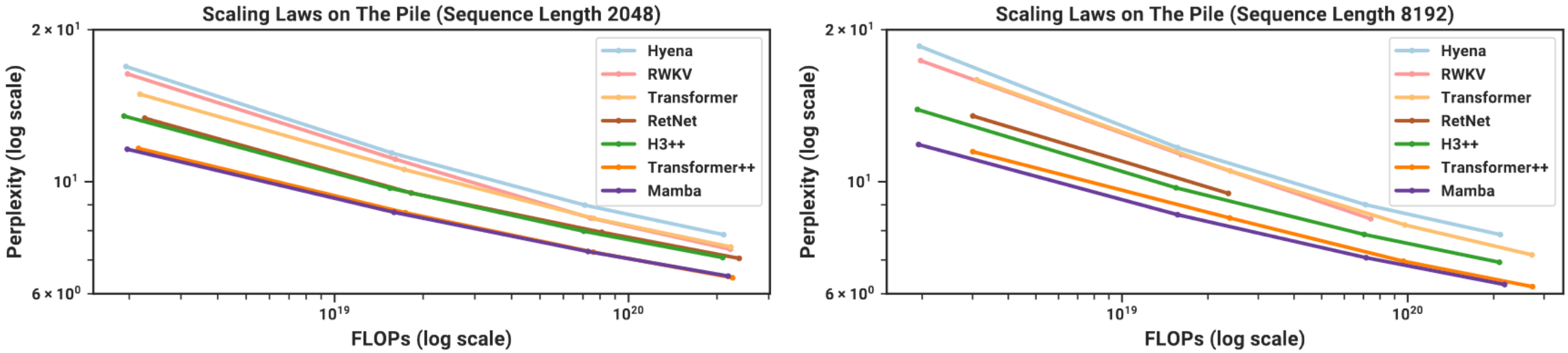


Figure 4: (**Scaling Laws**.) Models of size $\approx 125M$ to $\approx 1.3B$ parameters, trained on the Pile. Mamba scales better than all other attention-free models and is the first to match the performance of a very strong "Transformer++" recipe that has now become standard, particularly as the sequence length grows.

Thanks for watching!
Don’t forget to subscribe for more amazing content on AI and Machine Learning!