AI VIET NAM – 2024

Mamba: When Transformer is Not All You Need

Minh-Duc Bui và Quang-Vinh Dinh
PR-Team: Đăng-Nhã Nguyễn, Minh-Châu Phạm và Hoàng-Nguyên Vũ

Ngày 3 tháng 2 năm 2024

1 Introduction

Trong những năm gần đây, Transformer trở thành "kẻ thống trị" trong hầu hết các loại data, xuất phát từ language, lan dần ra đến image, audio, tabular,... và luôn là backbone của các state-of-the-art (SoTA) model trên hầu hết các benchmark. Large Language Model (LLM), trong những năm gần đây, trở thành chủ đề nóng hổi trong AI, vô vàn những model mới, những phương pháp mới đã được đề xuất để cải tiến LLM, nhưng có một thứ vẫn không thay đổi nhiều, chính là backbone của các LLM này hầu hết là Transformer.

Transformer xuất hiện (từ bài paper Attention Is All You Need) vào năm 2017 đã làm lu mờ hướng phát triển chính của language vào thời điểm bấy giờ là Recurrent Neural Network (RNN), khiến RNN dần ít được sự quan tâm của các nhóm nghiên cứu. Transformer trở thành chủ đề nóng hổi khắp mọi nơi vì khả năng tính toán attention mạnh mẽ, vượt trội hoàn toàn so với các kiến trúc RNN bấy giờ. Điểm yếu của Transformer là độ phức tạp $O(n^2)$ so với độ dài sequence, còn RNN chỉ là $O(n)$. Khi chiều dài input tăng gấp đôi thì độ phức tạp của Transformer tăng gấp 4 lần trong khi RNN tăng chỉ 2 lần. Tuy nhiên vì khả năng quá mạnh mẽ nên mọi người hầu hết chỉ tập trung vào độ chính xác.

Từ khi Transformer ra đời thì vẫn còn một số nhóm tác giả vẫn tiếp tục phát triển RNN để có thể tiến gần hơn performance của Transformer. Một trong những hướng đó là State Space Model (SSM) bắt nguồn từ **Control Theory**. Ngày 1 tháng 12 năm 2023, Mamba: Linear-Time Sequence Modeling with Selective State Spaces của 2 tác giả Albert Gu và Tri Dao xuất hiện và đem SSM model trở lại đường đua và vượt mặt cả Transformer trên mọi phương diện: tốc độ nhanh hơn, memory ít hơn, và độ chính xác cao hơn.

Trong bài viết này ta sẽ tìm hiểu điểm mạnh, điểm yếu của RNN và Transformer. Từ đó, ta sẽ xây dựng một model deep learning lý tưởng hội tụ đủ các điểm mạnh này, và cuối cùng hãy cùng xem các nhà nghiên cứu đã làm gì để đạt được điều này nhé!

2 Pros and Cons of RNNs and Transformers

Trong phần này, ta sẽ tìm hiểu kỹ hơn về điểm mạnh và điểm yếu của RNN và Transformer. Ta có thể quan sát bảng 1 để hiểu rõ hơn.

	RNN	**Transformer**	**Ideal Model**
Parallelizable	no	yes	yes
Training Stage	$O(n)$	$O(n^2)$	$O(n)$
Inference Stage	$O(n)$	$O(n^2)$	$O(n)$
Inference per Token	$O(1)$	$O(n)$	$O(1)$

Bảng 1: Bảng so sánh điểm mạnh và yếu của RNN, Transformer và model lý tưởng (ideal model).

- **Parallelizable**: RNN không thể tính toán parallel khi training vì tính chất **Linear-Time Variance** (để tính toán được output ở thời điểm t thì cần có hidden state vào thời điểm $t-1$ trước đó). Ngoài ra, RNN còn gặp phải vấn đề vanishing/exploding gradient do thông tin của các token trước đó được lưu trữ ngầm định (**implicitly**) thông qua hidden state làm RNN không thể "nhớ" được nhiều khi chiều dài sequence tăng.

  Ngược lại, Transformer vượt trội nhờ vào khả năng tính toán parallel khi training vì thông tin từ các token trước đó được lưu trữ rõ ràng (**explicitly**). Transformer không còn gặp phải các vấn đề vanishing/exploding gradient nhờ vào những phương pháp hiện đại (Skip-connection, LayerNorm,...).

  Suy ra, ta cần thiết kế một model có khả năng parallel khi training.

- **Training Stage**: Để training RNN ta cần đưa vào model từng token để tính toán và trả về hidden state rồi dùng hidden state đó và token tiếp theo để đưa vào model. Điều này tạo nên độ phức

tạp $O(n)$, với n là chiều dài sequence.

Transformer sở hữu độ phức tạp là $O(n^2)$ khi training vì token ở thời điểm t sẽ được tính toán attention đối với toàn bộ $t-1$ token phía trước đó (RNN thì chỉ quan tâm đến 1 token duy nhất trước đó là token thứ $t-1$), đây vốn là thách thức lớn nhất đối đối với Transformer, các nghiên cứu sau Transformer sẽ tập trung giải quyết vấn đề $O(n^2)$ này.

Suy ra, model lý tưởng phải có độ phức tạp O(n) khi training.

- **Inference Stage**: Cả RNN và Transformer đều cho độ phức tạp khi inference giống với khi training lần lượt là $O(n)$ và $O(n^2)$.

Ta cần thiết kế model lý tưởng có khả năng inference là O(n).

- **Inference per Token**: Hãy cùng phân tích kỹ hơn về thời gian để tạo ra 1 token khi inference.

Đối với RNN, để có thể tạo ra 1 token ở thời điểm t bất kỳ ta cần 2 input là hidden state $t-1$ và token t. Ở những thời điểm khác nhau thì thời gian để tạo ra 1 token đều như nhau (vì chỉ cần 2 input). Nói cách khác, thời gian để tạo ra token mới đối với RNN là $O(1)$ ở bất kỳ thời điểm nào.

Đối với Transformer, để tạo ra 1 token ở thời điểm t thì ta cần toàn bộ $t-1$ token trước đó để tính toán attention. Ví dụ để tạo ra token thứ 5 ta cần 4 token trước đó, tính toán 1 cách tương đối ta sẽ có 4 phép attention. Khi tạo ra token thứ 100 thì ta cần 99 token trước đó, tương với với 99 phép attention. Như vậy, đối với Transfomer, khi token tạo ra càng nhiều thì thời gian tạo ra từng token sẽ tăng lên, tương ứng với thời gian tạo ra từng token là $O(n)$.

Ta cần một model lý tưởng có thời gian tạo ra token khi inference là O(1) ở mọi thời điểm.

Model lý tưởng mà ta suy luận ra chính là Ideal Model của cột cuối cùng ở bảng 1, đây là model hội tụ điểm mạnh của cả RNN và Transformer. Phần tiếp theo ta sẽ tìm hiểu lịch sử của State Space Model - Ideal Model.

Part I: The History of State Space Models

Trong phần này ta sẽ tìm hiểu quá trình xây dựng model lý tưởng như ta đã đề cập ở phần trên xuất phát từ State Space Model (SSM) từ Control Theory để tạo ra model có khả năng tính toán parallel khi training với $O(n)$ và inference cũng với $O(n)$. Tuy nhiên SSM vẫn gặp phải các vấn đề về vanishing/exploding gradient đối với long sequence. Structured State Space Model (S4) ra đời nhằm giải quyết vấn đề này của SSM bằng một thay đổi đơn giản trong cách khởi tạo trọng số.

1 State Space Models

State Space Model (SSM) hoạt động bằng cách map (ánh xạ) input $x \in \mathbb{R}^{1\times D}$ sang không gian latent space $h \in \mathbb{R}^{1\times N}$, $(N > D)$ để tính toán sau đó map h về output $y \in \mathbb{R}^{1\times D}$. SSM được định nghĩa thông qua công thức sau:

$$h'(t) = \mathbf{A}h(t) + \mathbf{B}x(t), \tag{1a}$$
$$y(t) = \mathbf{C}h(t) + \mathbf{D}x(t), \tag{1b}$$

trong đó, $\mathbf{A} \in \mathbb{R}^{N\times N}, \mathbf{B} \in \mathbb{R}^{N\times 1}, \mathbf{C} \in \mathbb{R}^{1\times N}, \mathbf{D} \in \mathbb{R}^{N\times 1}$ là các ma trận parameter được model học. $\mathbf{D}x(t)$ đóng vai trò như skip-connection trong các model deep learning, ta có thể tạm bỏ đi phần này để tiện biểu diễn các công thức sau này. Tuy nhiên ta vẫn chưa thể áp dụng công thức này vào các model deep learning được vì công thức vẫn đang ở dạng function-to-function, tức ở dạng continuous. Ta cần discrete (rời rạc hóa) công thức trên bằng **step size** Δ (còn được gọi là **timescale**) và để model học giá trị Δ này. Ta bắt đầu từ công thức đạo hàm tại một điểm:

$$h'(t) = \lim_{\Delta\to 0} \frac{h(t+\Delta) - h(t)}{\Delta}. \tag{2}$$

Khi Δ rất nhỏ thì ta có thể xấp xỉ công thức (2) thành:

$$\begin{aligned} h'(t) &\approx \frac{h(t+\Delta) - h(t)}{\Delta} \\ h'(t)\Delta &\approx h(t+\Delta) - h(t) \\ h(t+\Delta) &\approx h'(t)\Delta + h(t). \end{aligned} \tag{3}$$

Khi thay (1a) vào (3) thì ta đã chuyển thành dạng discrete:

$$\begin{aligned} h_{t+\Delta} &\approx (\mathbf{A}h_t + \mathbf{B}x_t)\Delta + h_t \\ h_{t+\Delta} &\approx \Delta\mathbf{A}h_t + \Delta\mathbf{B}x_t + h_t \\ h_{t+\Delta} &\approx (\Delta\mathbf{A} + \mathbf{I})h_t + \Delta\mathbf{B}x_t \\ h_{t+\Delta} &\approx \overline{\mathbf{A}}h_t + \overline{\mathbf{B}}x_t, \end{aligned}$$

với $\overline{\mathbf{A}} = (\Delta\mathbf{A} + \mathbf{I})$ và $\overline{\mathbf{B}} = \Delta\mathbf{B}$.

Như vậy, ta vừa chuyển đổi từ dạng continuous sang dạng discrese nhờ vào phương pháp dùng định nghĩa đạo hàm tại một điểm. Trong thực tế, ta có nhiều phương pháp với các cách biến đổi khác nhau nhưng ý nghĩa vẫn là discrete công thức (1a) và (1b). Các công thức phổ biến thường được sử dụng là **Bilinear** hoặc **Zero-Order Hold (ZOH)**:

$$\textbf{(Bilinear)} \quad \begin{aligned} \overline{\mathbf{A}} &= (\mathbf{I} - \Delta/2\mathbf{A})^{-1}(\mathbf{I} + \Delta/2\mathbf{A}) \\ \overline{\mathbf{B}} &= (\mathbf{I} - \Delta/2\mathbf{A})^{-1} \cdot \Delta\mathbf{B} \end{aligned} \qquad \textbf{(ZOH)} \quad \begin{aligned} \overline{\mathbf{A}} &= \exp(\Delta\mathbf{A}) \\ \overline{\mathbf{B}} &= (\Delta\mathbf{A})^{-1}(\exp(\Delta \cdot \mathbf{A}) - \mathbf{I}) \cdot \Delta\mathbf{B}. \end{aligned}$$

Dạng discrete tổng quát sau khi ta biến đổi sẽ có công thức nhau sau:

$$h_t = \overline{\mathbf{A}}h_{t-1} + \overline{\mathbf{B}}x_t, \tag{4a}$$
$$y_t = \mathbf{C}h_t + \mathbf{D}x_t, \tag{4b}$$

$$\overline{\mathbf{A}}, \overline{\mathbf{B}} = \text{Bilinear}(\mathbf{A}, \mathbf{B}, \Delta) \quad \text{hoặc} \quad \overline{\mathbf{A}}, \overline{\mathbf{B}} = \text{ZOH}(\mathbf{A}, \mathbf{B}, \Delta),$$

trong đó $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$, và Δ là các parameter được model học, $\overline{\mathbf{A}}$ và $\overline{\mathbf{B}}$ được tính toán với mỗi step dựa vào công thức ZOH hoặc Bilinear, $t = 0...(L-1)$ là thứ tự của các token với chiều dài sequence L.

2 Convolutional Representation of State Space Model

Công thức (4a) và (4b) có độ phức tạp là $O(n)$ và không thể tính toán parallel, chỉ phù hợp cho quá trình inference, không phù hợp khi training model. Để có thể training model parallel ta cần biến đổi công thức này.

Để đơn giản, ta đặt $h_{-1} = \mathbf{0}$ tương ứng với input đầu tiên không có hidden state và tính toán các giá trị y khi $t = 0, 1, 2$:

$$\begin{aligned} h_0 &= \overline{\mathbf{B}}x_0 \\ y_0 &= \mathbf{C}h_0 = \mathbf{C}\overline{\mathbf{B}}x_0 \end{aligned}$$

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

$$\begin{aligned} h_2 &= \overline{\mathbf{A}}h_1 + \overline{\mathbf{B}}x_2 = \overline{\mathbf{A}}(\overline{\mathbf{A}\mathbf{B}}x_0 + \overline{\mathbf{B}}x_1) + \overline{\mathbf{B}}x_2 = \overline{\mathbf{A}}^2\overline{\mathbf{B}}x_0 + \overline{\mathbf{A}\mathbf{B}}x_1 + \overline{\mathbf{B}}x_2 \\ y_2 &= \mathbf{C}h_2 = \mathbf{C}(\overline{\mathbf{A}}^2\overline{\mathbf{B}}x_0 + \overline{\mathbf{A}\mathbf{B}}x_1 + \overline{\mathbf{B}}x_2) = \mathbf{C}\overline{\mathbf{A}}^2\overline{\mathbf{B}}x_0 + \mathbf{C}\overline{\mathbf{A}\mathbf{B}}x_1 + \mathbf{C}\overline{\mathbf{B}}x_2 \end{aligned}$$

Sau khi khai triển ta có thể nhận thấy một pattern được lặp lại, giá trị y_k tại thời điểm k bất kỳ có công thức tổng quát như sau:

$$y_k = \mathbf{C}\overline{\mathbf{A}}^k\overline{\mathbf{B}}x_0 + \mathbf{C}\overline{\mathbf{A}}^{k-1}\overline{\mathbf{B}}x_1 + \ldots + \mathbf{C}\overline{\mathbf{A}\mathbf{B}}x_{k-1} + \mathbf{C}\overline{\mathbf{B}}x_k. \tag{5}$$

Nếu ta đặt:

$$\overline{\mathbf{K}} = (\mathbf{C}\overline{\mathbf{B}}, \mathbf{C}\overline{\mathbf{A}\mathbf{B}}, \ldots, \mathbf{C}\overline{\mathbf{A}}^{L-1}\overline{\mathbf{B}}), \tag{6a}$$
$$x = (x_0, x_1, \ldots, x_{L-1}), \tag{6b}$$

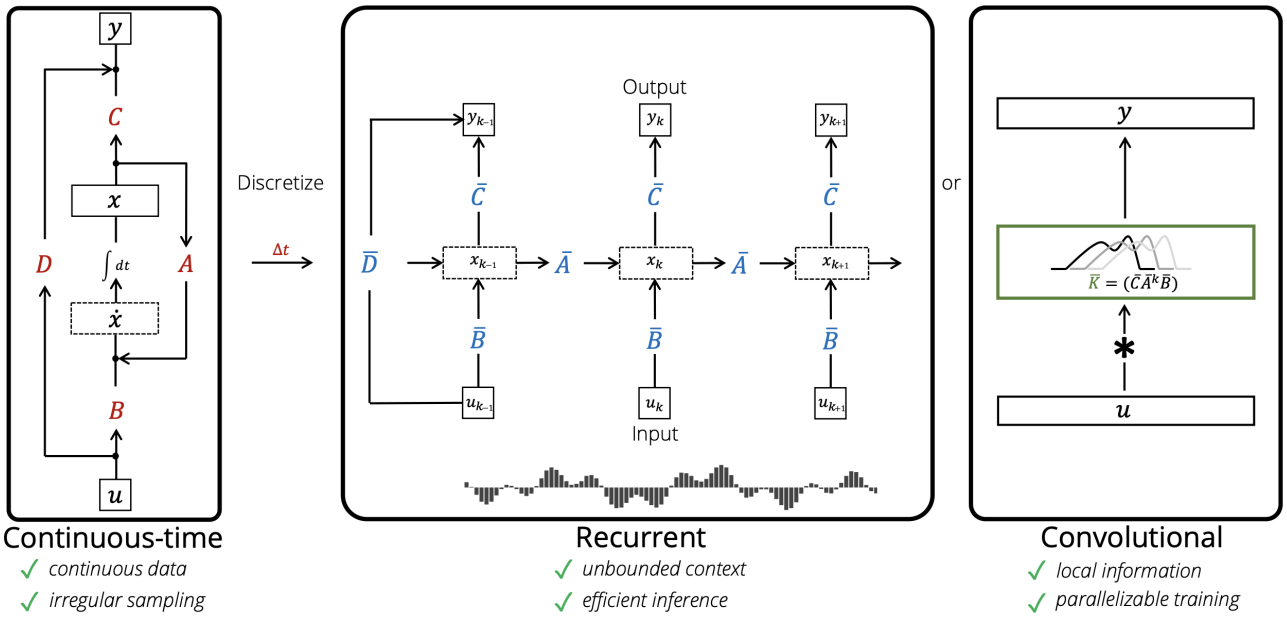
thì công thức (5) trở thành:

$$y = \overline{\mathbf{K}} * x. \tag{7}$$

Nhìn vào công thức (7) ta thấy, $\overline{\mathbf{K}}$ được tạo thành từ việc nhân các ma trận $\overline{\mathbf{A}}, \overline{\mathbf{B}}, \mathbf{C}$ với nhau, và việc nhân các ma trận này không phụ thuộc vào input. Hay nói cách khác công thức (7) có thể được tính toán parallel.

3 State Space Model is Ideal Model

Như vậy, thông qua phần 1 và 2 ta đã xây dựng được model SSM có độ phức tạp $O(n)$ khi training cùng với khả năng parallel, và inference với $O(n)$. Nói cách khác, đây chính là model lý tưởng mà ta đã xây dựng ở bảng 1. Hình sau miêu tả 3 dạng triển khai khác nhau của model SSM, ta không thể sử dụng dạng **Continuous-time** để xử lý các bài toán deep learning. Dạng **Recurrent** phù hợp với việc inference nhờ độ phức tạp $O(n)$ và dạng **Convolutional** phù hợp khi training nhờ khả năng tính toán parallel và độ phức tạp $O(n)$.



4 Structured State Space Models (S4)

Khi triển khai các model SSM này vào các bài toán deep learning thì ta vẫn gặp phải một số vấn đề tương tự RNN là vanishing/exploding gradient và hoạt động kém hiệu quả khi chiều dài sequence tăng. Dựa vào công thức (4a) ta thấy thông tin của các token quá khứ được lưu trữ vào hidden state h_{t-1} (tương tự hidden state của RNN). Và h_{t-1} được kiểm sát bởi ma trận tham số $\overline{\mathbf{A}}$, nói cách khác chính ma trận $\mathbf{A}$ (tạo nên $\overline{\mathbf{A}}$) là nguyên nhân dẫn đến tình trạng trên.

Để giải quyết vấn đề này, các tác giả của bài paper HiPPO: Recurrent Memory with Optimal Polynomial Projections thay vì sử dụng random để khởi tạo trọng số cho ma trận $\mathbf{A}$, họ sử dụng ma trận HiPPO (xuất phát từ HiPPO theory thuộc **continuous-time memorization**) làm ma trận khởi tạo, giúp tăng performance trên tập sequencial MNIST từ 60% lên 98%. Ma trận HiPPO được định nghĩa như sau:

$$(\textbf{HiPPO Matrix}) \quad A_{nk} = -\begin{cases} (2n+1)^{1/2}(2k+1)^{1/2} & \text{if } n > k \\ n+1 & \text{if } n = k \\ 0 & \text{if } n < k \end{cases}$$

Nhóm tác giả của paper Efficiently Modeling Long Sequences with Structured State Spaces tiếp tục cải tiến bằng một số phương pháp toán học như: **Diagonalization**, **truncated generating function**,

Woodbury identity, **Cauchy kernel**, **Normal Plus Low-Rank (NPLR)**, và **inverse FFT**. Ta sẽ không đi vào chi tiết trong bài viết này, mục tiêu của các phương pháp được rút gọn như sau:

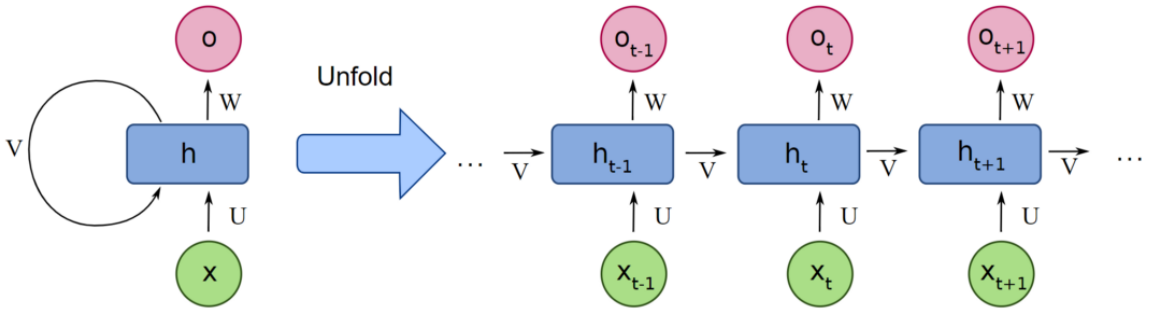
- Khởi tạo trọng số cho ma trận $\mathbf{A}$ để có thể lưu trữ thông tin hiệu quả hơn (so với việc random khởi tạo).
- Tăng tốc thời gian tính toán $\overline{\mathbf{K}}$ vì chứa $\overline{\mathbf{A}}^{L-1}$ (lũy thừa ma trận tốn rất nhiều thời gian tính toán, đặc biệt khi chiều dài sequence L lớn).

Part II: Mamba - State Space Models with Selective Scan (S6)

1 Motivation

Structured **S**tate **S**pace for **S**equence Modeling (S4) đã tạo nên một bước tiến lớn đối với hướng phát triển SSM. S4 giúp giảm độ phức tạp tính toán từ $O(N^2L)$ của các model SSM trước đó thành $O(N+L)$, từ $O(NL)$ thành $O(N+L)$ khi so sánh về memory (N và L lần lượt là state dimension và chiều dài sequence). So sánh với các model SSM trước đó, tốc độ tính toán đã tăng 30× và giảm memory hơn 400×. Tuy nhiên, khi xét về performance thì vẫn còn một khoảng cách lớn với Transformer.

Nếu nhìn vào công thức SSM ở dạng recurrent (4a) và (4b) hoặc dạng convolutional (7) thì ta thấy các ma trận tham số **A**, **B**, **C**, **D**, và Δ đều được áp dụng chung cho các token khác nhau trong cùng một sequence. Ta có thể tạm nhìn vào kiến trúc của RNN một cách tương đối từ hình 1 để có thể hiểu rõ hơn, vì SSM chính là 1 dạng cải tiến của RNN. Hình 1 cho thấy các trọng số của model được áp dụng vào tất cả các token khác nhau trong cùng một sequence. RNN (hay cả SSM) không có khả năng "attention" ít hoặc nhiều vào từng token khác nhau, nói cách khác, chúng đối xử tất cả các token trong một sequence như nhau. Đây là một trong những vấn đề chính khiến RNN hoạt động không hiệu quả. Ví dụ ta có câu "Tôi là học sinh", ta thấy từ "là" mang rất ít ý nghĩa, hay trong tiếng Anh là các từ như "a, the, is, are, was,...". Rõ ràng RNN và SSM đều đối xử các từ này và tất cả những từ khác theo cùng một cách (áp dụng cùng một bộ parameter).

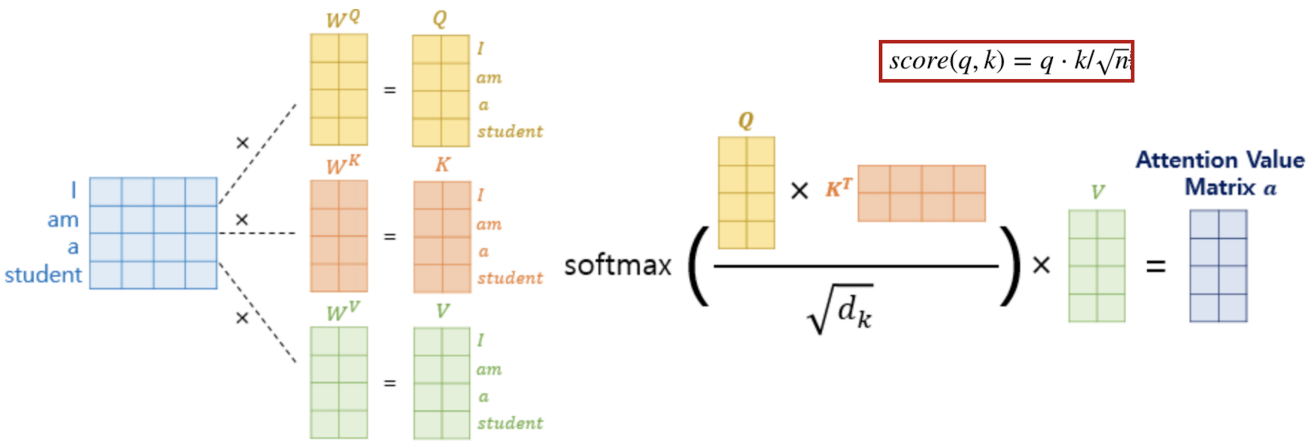


Hình 1: Minh họa kiến trúc RNN.

Hãy nhìn thử cách Transformer giải quyết vấn đề này, qua hình 2 ta thấy các ma trận **Q**, **K**, và **V** được tạo thành từ input x, nói cách khác, chúng có mối liên hệ với input (RNN và SSM thì không). Bên cạnh khả năng Attention mạnh mẽ, việc tạo ra các ma trận **Q**, **K**, và **V** từ input cũng đóng góp một phần lớn vào sự vượt trội của Transformer.

Motivation chính của Mamba chính là việc tạo ra các ma trận tham số phụ thuộc vào input. Sau đây là đoạn trích từ bài paper gốc.

> We identify a key limitation of prior models: the ability to efficiently select data in an input-dependent manner (i.e. focus on or ignore particular inputs).

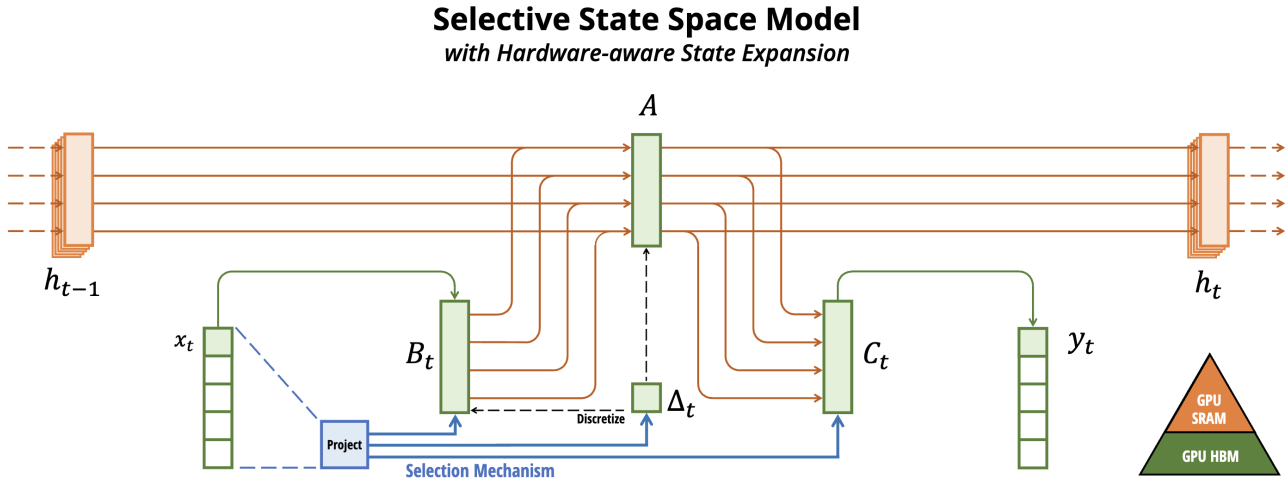


Hình 2: Minh họa cơ chế Attention trong Transformer.

2 Contribution

Mamba có 2 contribution chính là:

- **Selection Mechanism**: tạo ra parameter từ input.
- **Hardware-aware Algorithm**: gồm 3 phương pháp chính là kernel fusion, parallel scan, và recomputation nhằm mục đích tăng tốc độ tính toán cho các phép tính SSM.



Khi tạo ra các parameter từ input như Transformer tạo ra ma trận **QKV** thì ta không thể áp dụng dạng Convolutional (7) để tính toán nữa, nói cách khác ta không thể tính toán parallel khi training model (ta chỉ có thể sử dụng dạng recurrent). Để giải quyết vấn đề này, tác giả đã đưa thêm contribution thứ hai là Hardware-aware Algorithm gồm 3 phương pháp chính là kernel fusion, parallel scan, và recomputation để tăng tốc độ tính toán. Kết quả dù mất đi công thức ở dạng convolutional để tính toán parallel, nhưng ta đã có thể tính toán parallel từ dạng recurrent.

2.1 Selection Mechanism

Sau đây là hình vẽ so sánh sự khác biệt giữa algorithm của S4 và Mamba (S6):

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
$\Delta : (\mathtt{D}) \leftarrow \tau_\Delta$(Parameter)
$\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}} : (\mathtt{D}, \mathtt{N}) \leftarrow$ 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
$\boldsymbol{B} : (\mathtt{B}, \mathtt{L}, \mathtt{N}) \leftarrow s_B(x)$
$\boldsymbol{C} : (\mathtt{B}, \mathtt{L}, \mathtt{N}) \leftarrow s_C(x)$
$\Delta : (\mathtt{B}, \mathtt{L}, \mathtt{D}) \leftarrow \tau_\Delta$(Parameter$+s_\Delta(x)$)
$\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}} : (\mathtt{B}, \mathtt{L}, \mathtt{D}, \mathtt{N}) \leftarrow$ 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

Nhìn vào hình vẽ ta thấy ở S6, các ma trận tham số $\mathbf{B}$, $\mathbf{C}$, và Δ được tạo ra từ input x thông qua các lớp *Linear* được định nghĩa như sau:

$$s_B(x) = \text{Linear}_N(x), \quad s_N(x) = \text{Linear}_N(x), \quad s_A(x) = \text{Broadcast}_D(\text{Linear}_1(x)), \quad \text{và} \quad \tau_A = \text{softplus}$$

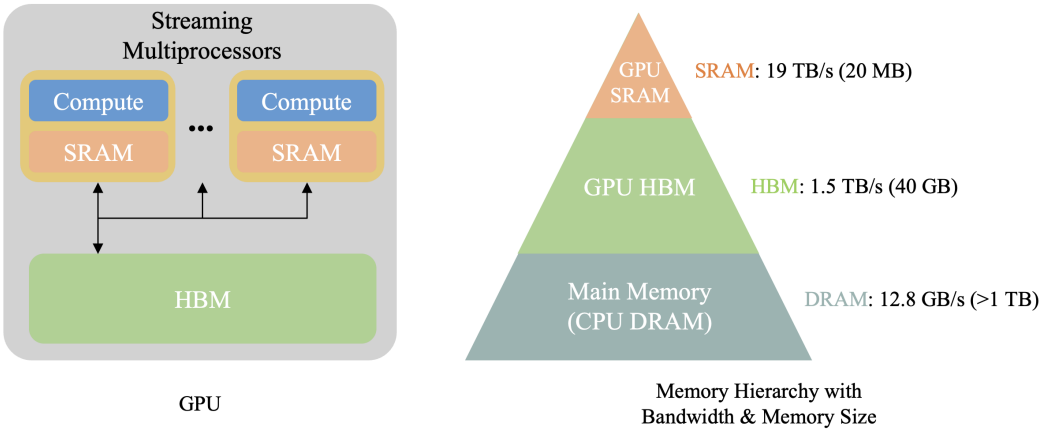
Ta thấy rằng, các ma trận $\mathbf{B}$, $\mathbf{C}$, và Δ sẽ có giá trị khác nhau ứng với từng input khác nhau, điều này giúp model có thể quyết định việc tập trung ít hay nhiều vào từng input khác nhau. Và các ma trận này sẽ có thêm một dimension về sequence L, vì thế, công thức đã chuyển đổi từ time-invariant sang time-varying.

2.2 Hardware-aware Algorithm

Để giải quyết vấn đề time-varing (phụ thuộc vào thời gian), hay không thể tính toán parallel khi training, các tác giả đã đề xuất 3 phương pháp giúp cải thiện điều này. Thế nhưng trước đó hãy tìm hiểu xem cách GPU thực sự hoạt động thế nào.

2.2.1 How does GPU work?

Kiến trúc GPU được mô tả như hình 3. Nhìn vào hình bên phải ta sẽ thấy phần lớn hơn 99% dung lượng của một GPU là Hight-Bandwidth Memory (HBM), còn lại một lượng rất nhỏ là SRAM, nơi thực hiện hiện các phép tính toán (nhân ma trận, softmax, cộng vector,...) trong model deep learning. HBM đóng vai trò là nơi lưu trữ, còn SRAM chính là nơi tính toán.



Hình 3: Hình vẽ mô tả kiến trúc GPU và Memory Hierarchy.

Hình bên trái mô tả quá trình tính toán của GPU, giả sử ta cần tính phép nhân 2 ma trận, quá trình tính toán sẽ được diễn ra như sau:

1. Di chuyển 2 ma trận cần tính toán từ HBM vào SRAM.
2. Thực hiện tính toán ở SRAM.
3. Di chuyển ma trận kết quả từ SRAM về HBM.

2.2.2 Kernel fusion

Tưởng tượng nếu ta thực hiện 3 phép tính nhân ma trận lên tiếp trên cùng 1 ma trận như sau: $\mathbf{A} = \mathbf{X_1} * \mathbf{Y}, \mathbf{B} = \mathbf{X_2} * \mathbf{Y}, \mathbf{C} = \mathbf{X_3} * \mathbf{Y}$). Quá trình tính toán sẽ diễn ra như sau:

1. Di chuyển $\mathbf{X_1}$ và $\mathbf{Y}$ từ HBM xuống SRAM để tính toán $\mathbf{A}$.
2. Di chuyển $\mathbf{A}$, $\mathbf{X_1}$, và $\mathbf{Y}$ từ SRAM về HBM.
3. Di chuyển $\mathbf{X_2}$, $\mathbf{Y}$ từ HBM xuống SRAM để tính toán $\mathbf{B}$
4. Di chuyển $\mathbf{B}$, $\mathbf{X_2}$, và $\mathbf{Y}$ từ SRAM về HBM.
5. Di chuyển $\mathbf{X_3}$, $\mathbf{Y}$ từ HBM xuống SRAM để tính toán $\mathbf{C}$
6. Di chuyển $\mathbf{C}$, $\mathbf{X_3}$, và $\mathbf{Y}$ từ SRAM về HBM.

Nhìn vào quá trình trên, chắc hẳn các bạn đã nhận ra một vấn đề làm tốn kém rất nhiều thời gian chính là di chuyển $\mathbf{Y}$ từ HBM xuống SRAM liên tục. GPU mạnh ở sức mạnh tính toán parallel, không phải tốc độ copy data từ HBM và SRAM, nói cách khác, quá trình copy data giữa HBM và SRAM tốn rất nhiều thời gian.

Chính vì vấn đề trên, tác giả đã đề xuất phương pháp **Kernel Fusion**, ý tưởng rất đơn giản là khi tính toán nhiều phép toán trên cùng một ma trận liên tiếp ($\mathbf{Y}$ trong ví dụ trên) thì sẽ giữ ma trận đó ở lại SRAM, tránh việc di chuyển giữa HBM và SRAM làm mất thời gian.

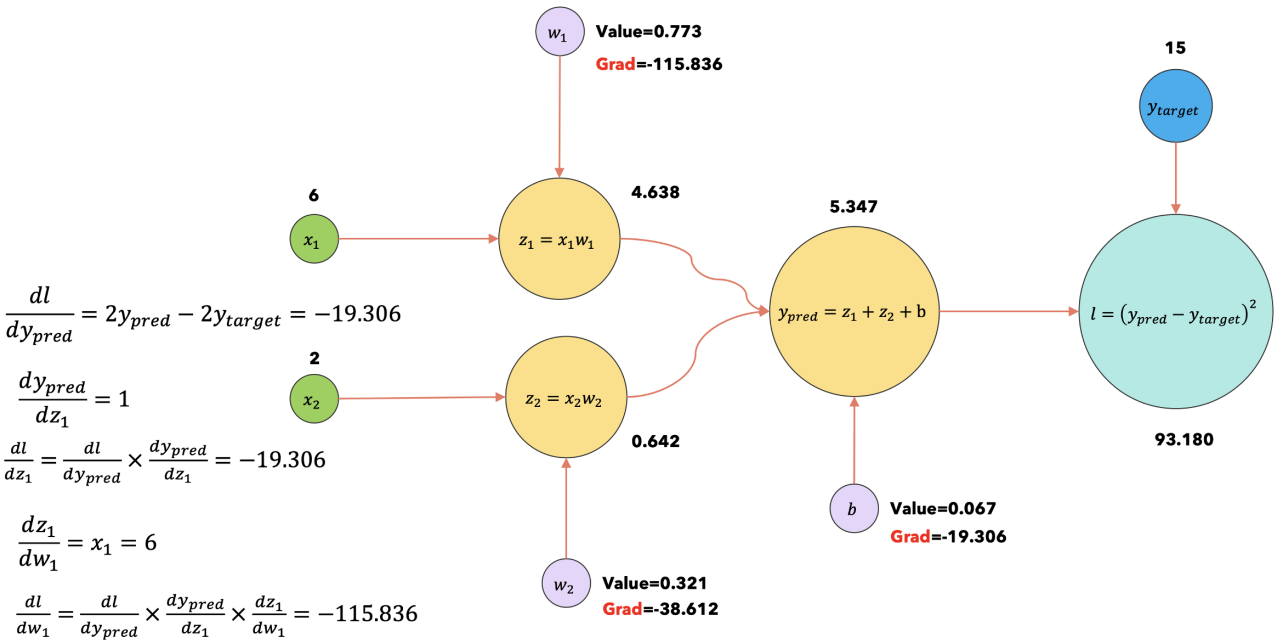
2.2.3 Recomputation

Tác giả đề xuất thêm một phương pháp thứ 2 để tăng tốc độ training model. Hãy xét ví dụ model đơn giản sau:

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

Sau đây là computational graph (đồ thị tính toán; ảnh từ đây) của model trên ứng với input $x_1 = 6$ và $x_2 = 2$. Ta thấy rằng khi tính toán model thì ta sẽ thu được các giá trị trung gian như z_1 và z_2, và ta không thể xóa các giá trị này đi được vì model cần lưu lại các giá trị này để tính toán đạo hàm theo Chain rule. Nói cách khác, toàn bộ các giá trị trung gian này sẽ được tạo ra ở SRAM rồi sau đó di chuyển về HBM để tiết kiệm bộ nhớ SRAM. Khi model với số lượng tham số lớn thì sẽ có hàng triệu giá trị trung gian được tính toán ở SRAM nhưng sau đó di chuyển về HBM, đến khi tính toán đạo hàm thì sẽ di chuyển ngược lại SRAM. Như ở phần trước ta đã thảo luận, quá trình copy data trong GPU là cực kỳ tốn kém thời gian.

Dựa vào motivation, tác giả đã đề xuất xóa bỏ những giá trị trung gian này thay vì di chuyển về HBM, đến khi tính đạo hàm thì ta sẽ recompute (tính toán lại) các giá trị này.



2.2.4 Parallel scan

Cuối cùng là phần quan trọng để giải quyết công thức ở dạng recurrent nhưng có thể chạy parallel. Ta sẽ bắt đầu từ bài toán **Pre-fix Sum** rất nổi tiếng trong lập trình, ví dụ như hình sau:

Initial array	9	6	7	10	8	7
Pre-fix sum	9	15	22	32	40	47

Pre-fix sum là một ma trận thỏa mãn: prefix_sum_array$_i$ = init_array$_i$ + prefix_sum_array$_{i-1}$. Ví dụ theo hình ta có: $15 = 9 + 6$, $22 = 15 + 7$, $32 = 22 + 10$,... Và ta hoàn toàn có thể tính toán ma trận prefix_sum_array này parallel, độ phức tạp lúc này sẽ giảm thành $O(N/T)$ với N là chiều dài array, T là số core trên compute unit (CPU hoặc GPU). Tham khảo thêm tại đây hoặc đây:

Ta thấy rằng dạng recurrent của Mamba cũng có tính chất tương tự, h_1 sẽ được tính toán từ x_1 và h_0, h_2 được tính toán từ x_2 và h_1,...

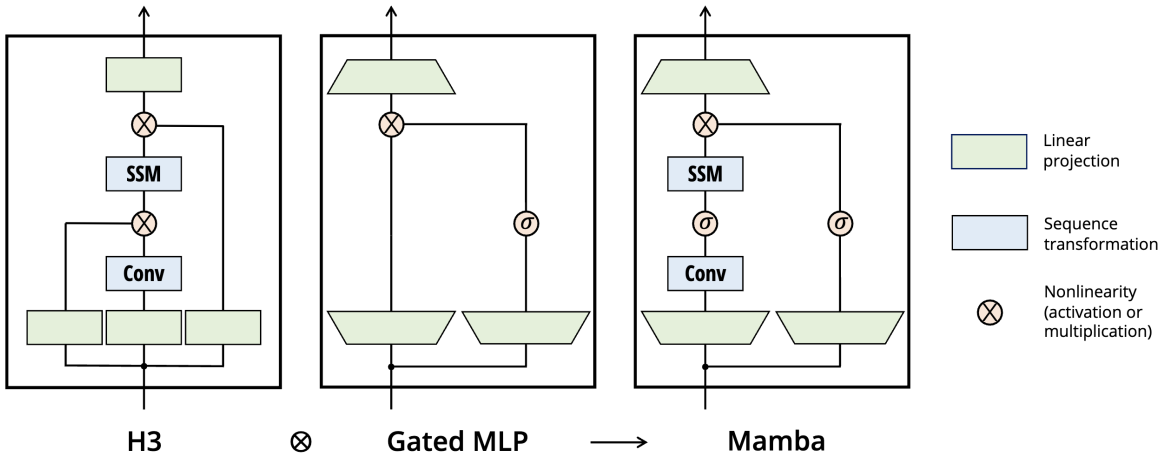
Model input	x_0	x_1	x_2	x_3	x_4	x_5
Scan output	h_0	h_1	h_2	h_3	h_4	h_5

Khi áp dụng parallel scan vào Mamba giúp ta có thể tính toán parallel dạng recurrent của Mamba.

3 Mamba Architecture

Sau khi đã tìm hiểu về Mamba (S6), giờ ta sẽ xây dựng Mamba block. Hình 4 mô tả mamba block được lấy ý tưởng từ H3, Gated MLP. Ta thấy trong block mamba, SSM là S6 với một số activation function

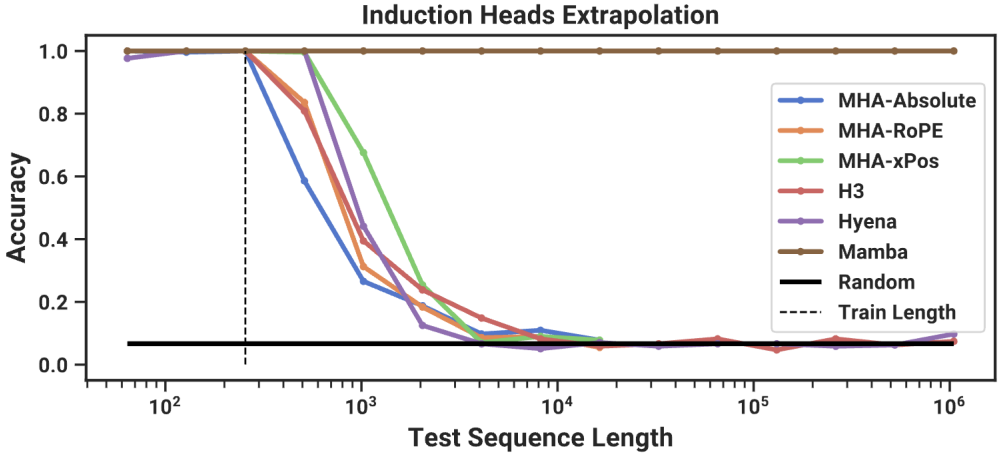
và linear projection xung quanh. Bằng cách stack nhiều mamba block lên nhau ta sẽ thu được model chính (tương tự Transformer).



Hình 4: Kiến trúc của Mamba block

4 Results

Mamba đạt tuyệt đối 100% accuracy đối với task Induction Heads khi số lượng token lên đến 1M. Tất cả các model còn khác đều giảm accuracy từ token thứ khoảng 1K trở đi, và giảm về <10% ở khoảng 10K token.



Model	Params	Test Accuracy (%) at Sequence Length														
		2^6	2^7	$\mathbf{2^8}$	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
MHA-Abs	137K	✓	99.6	100.0	58.6	26.6	18.8	9.8	10.9	7.8	✗	✗	✗	✗	✗	✗
MHA-RoPE	137K	✓	✓	100.0	83.6	31.3	18.4	8.6	9.0	5.5	✗	✗	✗	✗	✗	✗
MHA-xPos	137K	✓	✓	100.0	99.6	67.6	25.4	7.0	9.0	7.8	✗	✗	✗	✗	✗	✗
H3	153K	✓	✓	100.0	80.9	39.5	23.8	14.8	8.2	5.9	6.6	8.2	4.7	8.2	6.3	7.4
Hyena	69M*	97.7	✓	100.0	✓	44.1	12.5	6.6	5.1	7.0	5.9	6.6	6.6	5.9	6.3	9.8
Mamba	74K	✓	✓	100.0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

* Most of the parameters are in learnable positional encodings.

Đối với Selective Copy task (model phải học để chỉ giữ lại thông tin quan trọng và loại bỏ các thông tin khác) cho thấy Mamba tiếp tục đạt accuracy gần như tuyệt đối. Các model còn lại đều cho thấy khả năng hoạt động kém hiệu quả.

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**

5 Conclusion

Như vậy trong bài viết này ta đã tìm hiểu về Mamba hay S6: **S**tructured **S**tate **S**pace for **S**equence Modeling with **S**elective **S**can - một kiến trúc mới với khả năng vượt trội hơn Transformer ở mọi phương diện: từ speed, memory đến accuracy. Mamba hứa hẹn sẽ trở thành backbone mới trong tương lai, thay thế các backbone Transformer hiện tại. Đặc biệt khi Mamba được thay thế cho các backbone LLM sẽ tạo ra vô số lợi ích cho cả nhà cung cấp và người dùng, vì giờ đây, họ đã có thể sử dụng một model LLM nhanh hơn, độ chính xác cao hơn, và tất nhiên chi phí cũng sẽ rẻ hơn rất nhiều so với Transfomer.

References

[1] Gu, Albert, and Tri Dao. "Mamba: Linear-time sequence modeling with selective state spaces."arXiv preprint arXiv:2312.00752 (2023)

[2] Gu, Albert, et al. "Combining recurrent, convolutional, and continuous-time models with linear state space layers."Advances in neural information processing systems 34 (2021): 572-585.

[3] Gu, Albert, Karan Goel, and Christopher Ré. "Efficiently modeling long sequences with structured state spaces."arXiv preprint arXiv:2111.00396 (2021).

[4] Tay, Yi, et al. "Long range arena: A benchmark for efficient transformers."arXiv preprint arXiv:2011.04006 (2020).

[5] Gu, Albert, et al. "Hippo: Recurrent memory with optimal polynomial projections."Advances in neural information processing systems 33 (2020): 1474-1487.