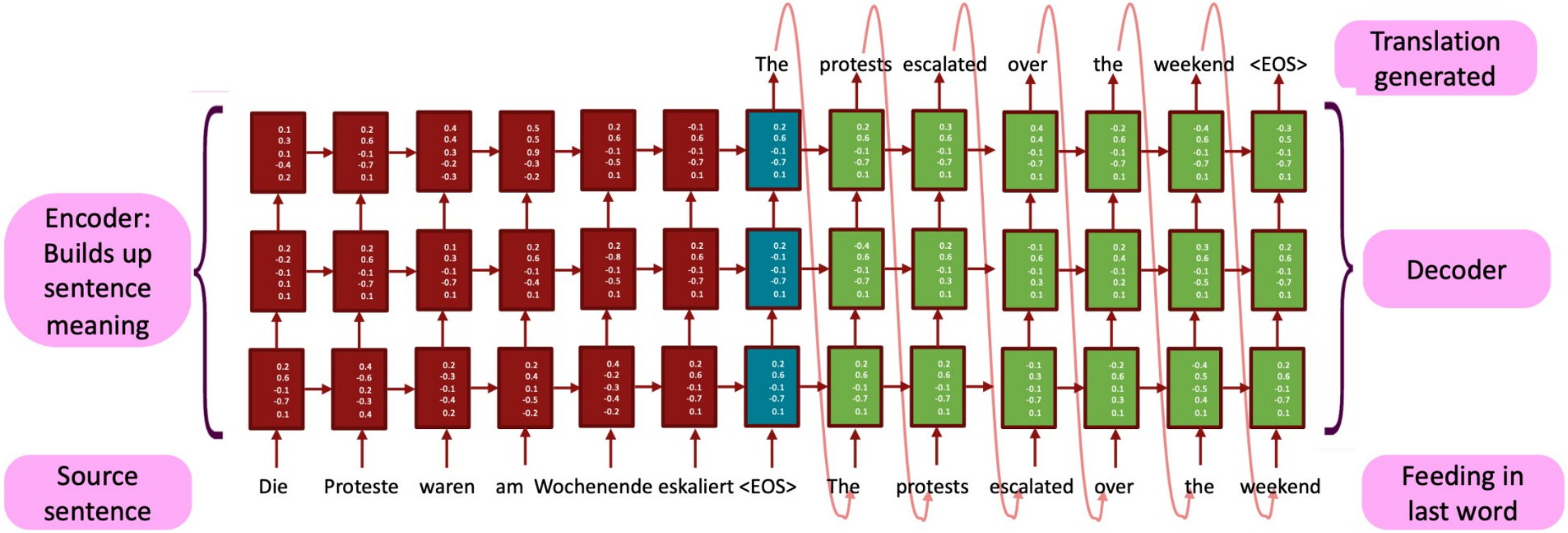
Attention

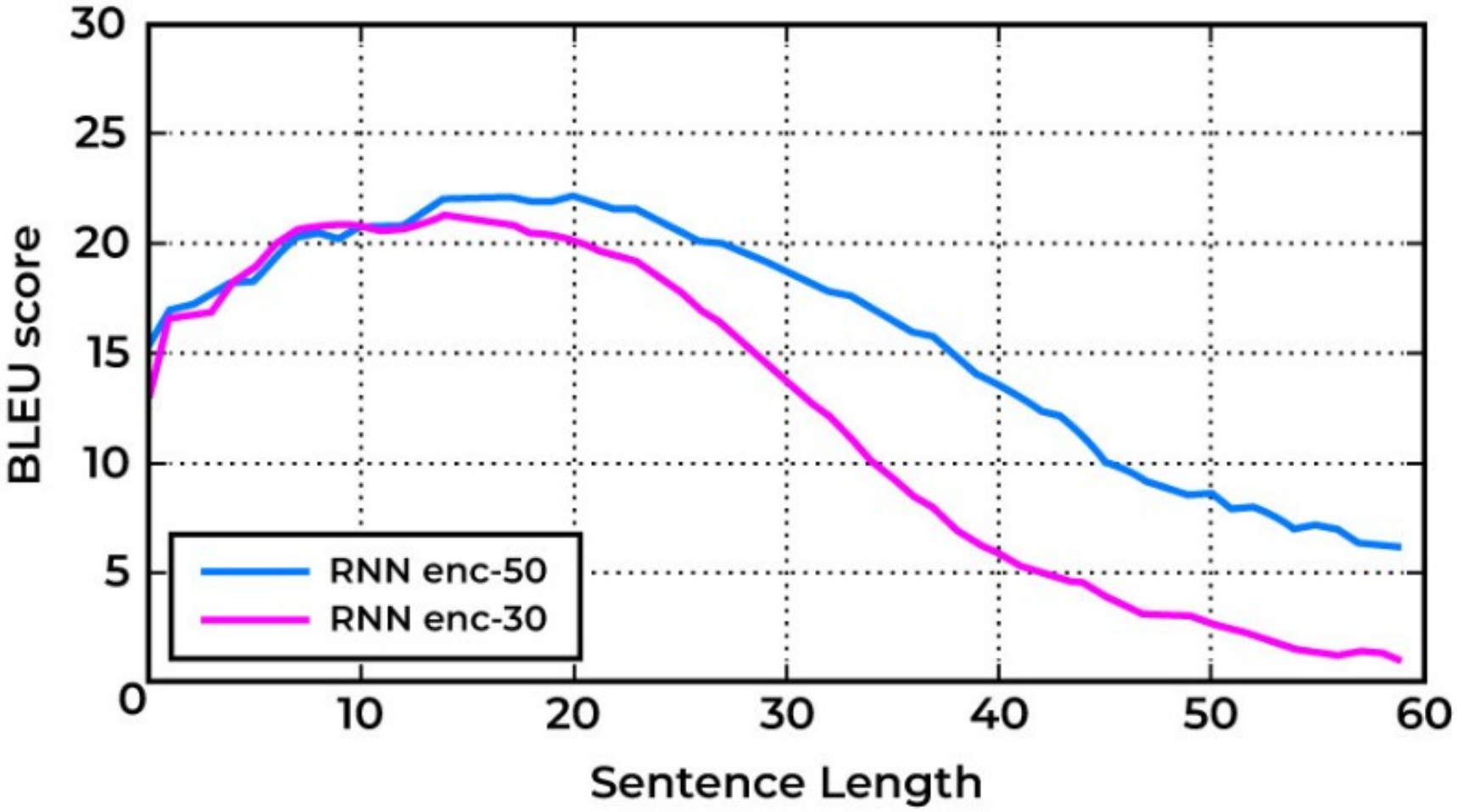
seq2seq

multi-layer deep encoder-decoder

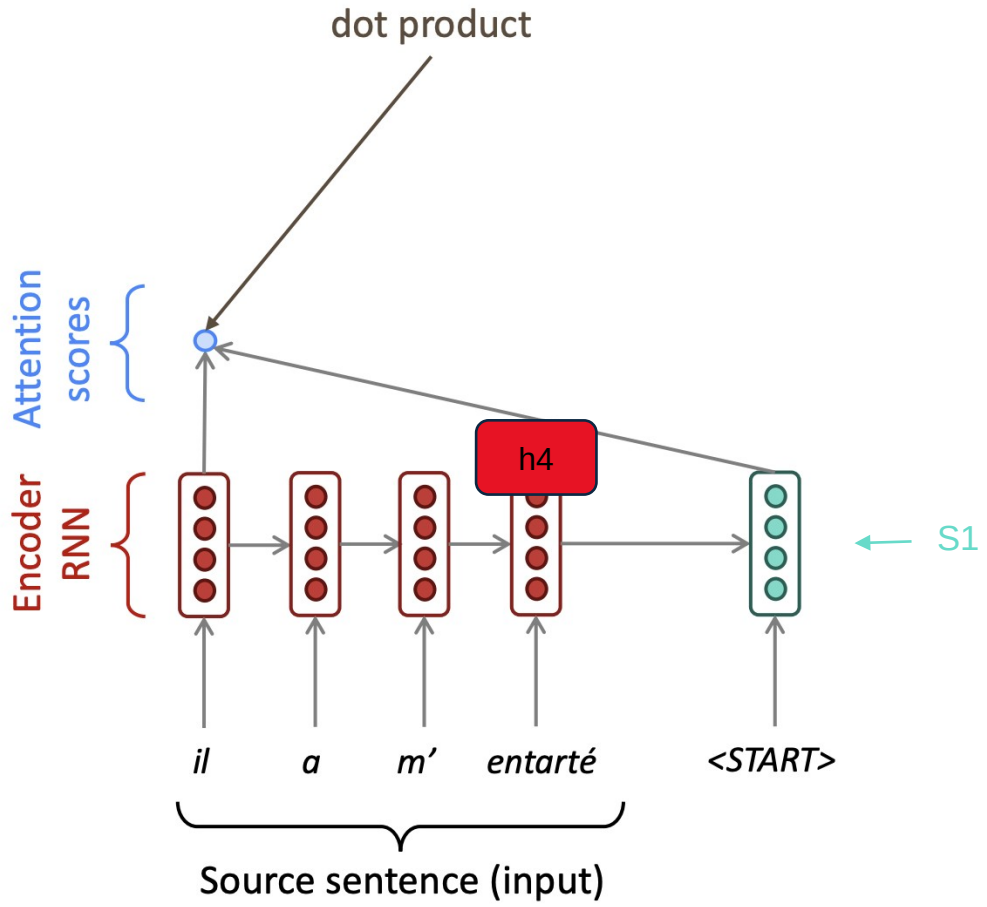


Why attention?

bottleneck problem



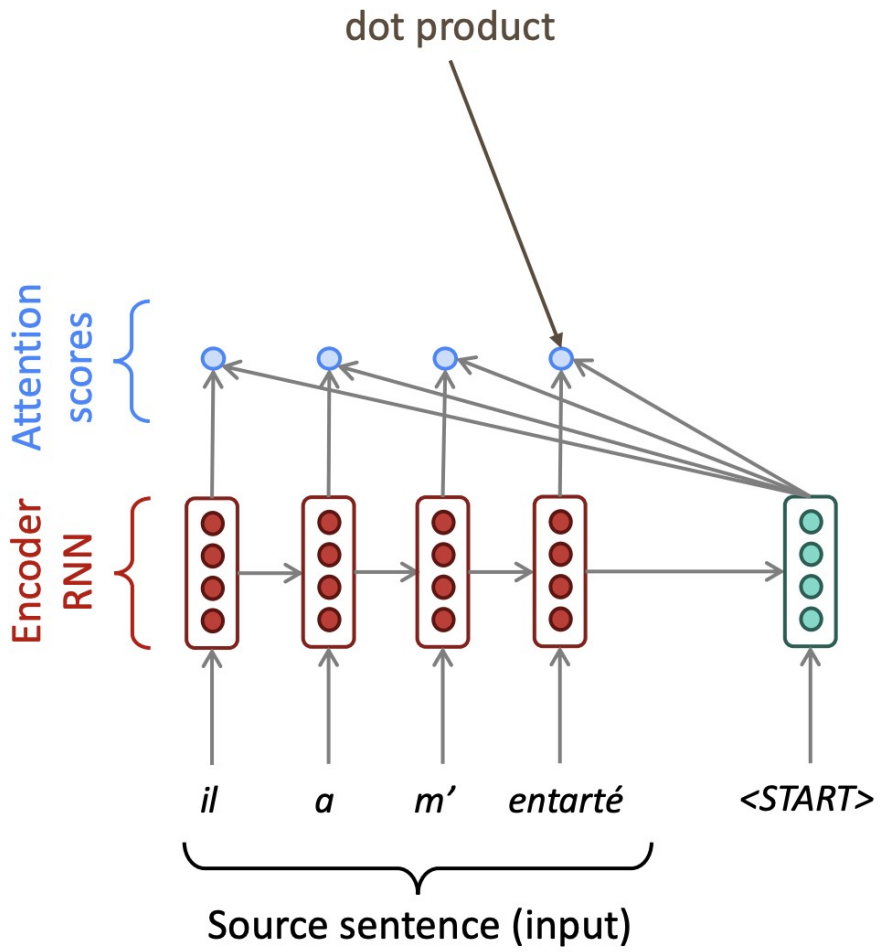
Attention



s1 = DecoderRNN(<START>, h4)

e1 = s1 · h1

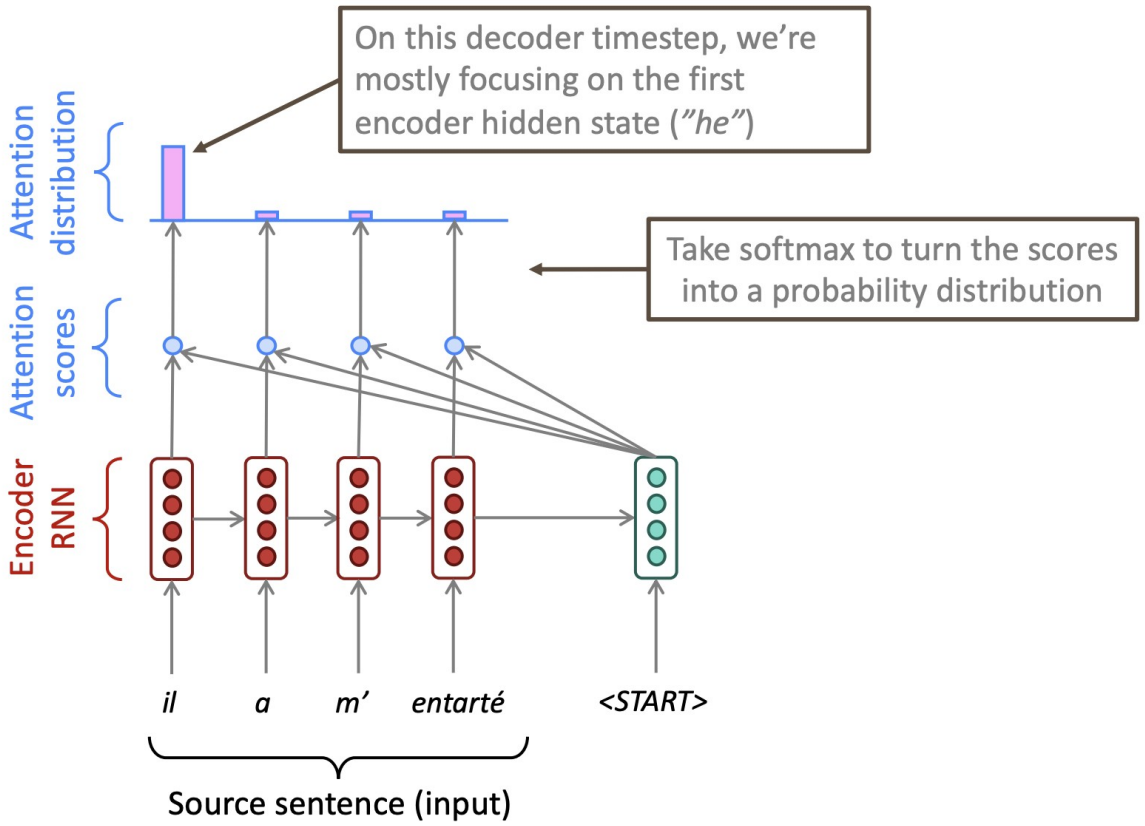
Attention



s1 = DecoderRNN(<START>, h4)

e1 = s1 · h1
e2 = s1 · h2
e3 = s1 · h3
e4 = s1 · h4

Attention

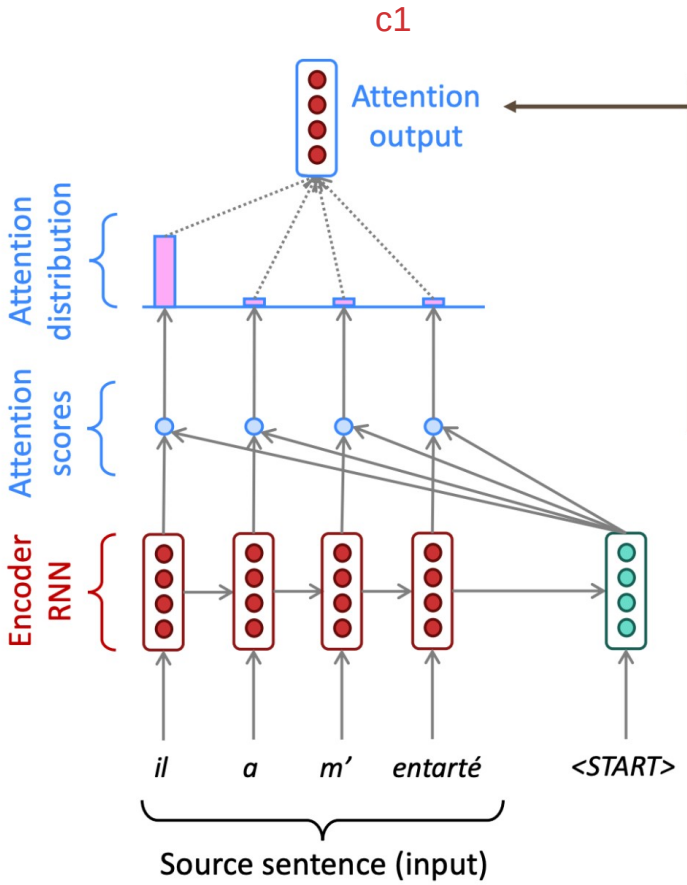


s1 = DecoderRNN(<START>, h4)

e1 = s1 · h1
e2 = s1 · h2
e3 = s1 · h3
e4 = s1 · h4

α = softmax(e)

Attention



Use the attention distribution to take a **weighted sum** of the encoder hidden states.

The attention output mostly contains information from the hidden states that received high attention.

Decoder RNN

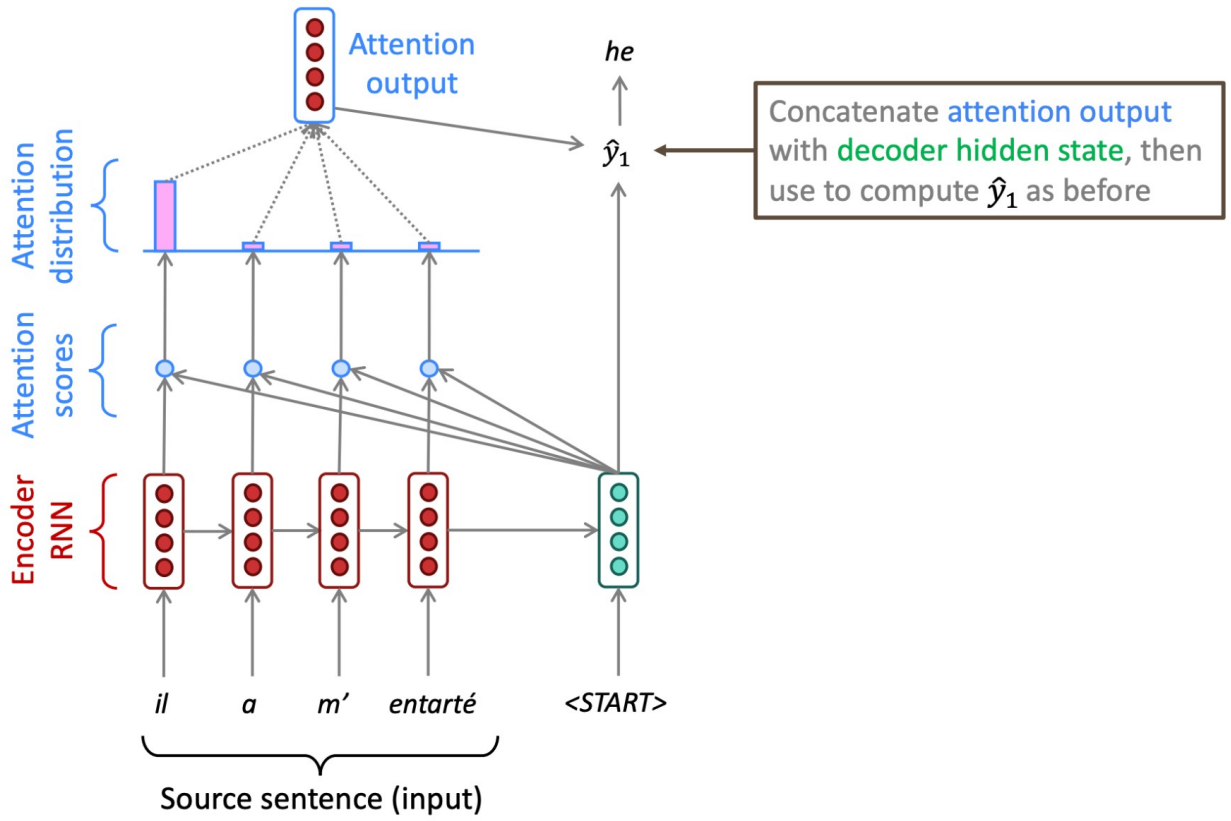
s1 = DecoderRNN(<START>, h4)

e1 = s1 · h1
e2 = s1 · h2
e3 = s1 · h3
e4 = s1 · h4

α = softmax(e)

c1 = α1 · h1 + α2 · h2 + α3 · h3 + α4 · h4

Attention



s1 = DecoderRNN(<START>, h4)

e1 = s1 · h1
e2 = s1 · h2
e3 = s1 · h3
e4 = s1 · h4

α = softmax(e)

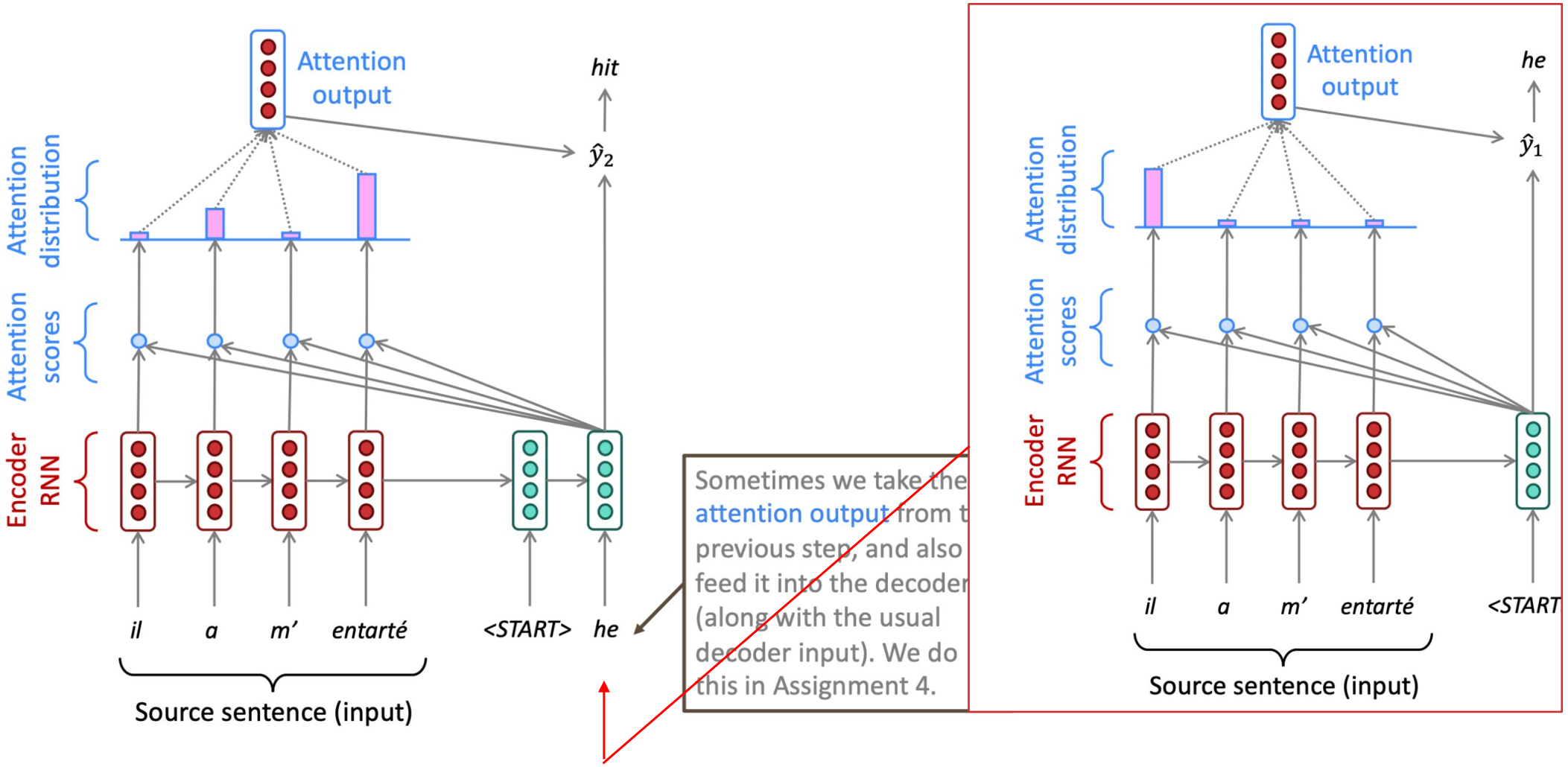
c1 = α1 · h1 + α2 · h2 + α3 · h3 + α4 · h4

y1 = tanh(W1 · [s1 ; c1] + b1)

z = W2 · y1 + b2

softmax(z)

Attention


Attention output
hit
$\hat{y}_2$
Attention distribution
Attention scores
Encoder RNN
il
a
m'
entarté
<START>
he
Source sentence (input)
Sometimes we take the attention output from t previous step, and also feed it into the decoder (along with the usual decoder input). We do this in Assignment 4.
Attention output
he
$\hat{y}_1$
Attention distribution
Attention scores
Encoder RNN
il
a
m'
entarté
<START
Source sentence (input)

Attention

Attention: in equations

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores e^t for this step:

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

- We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \text{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output $\boldsymbol{a}_t$ with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

22

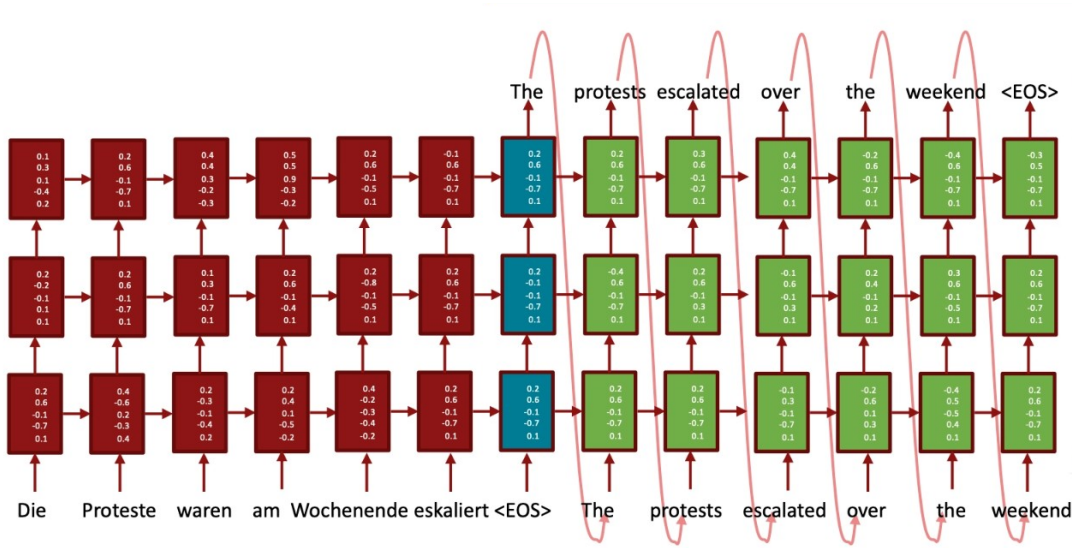
Attention

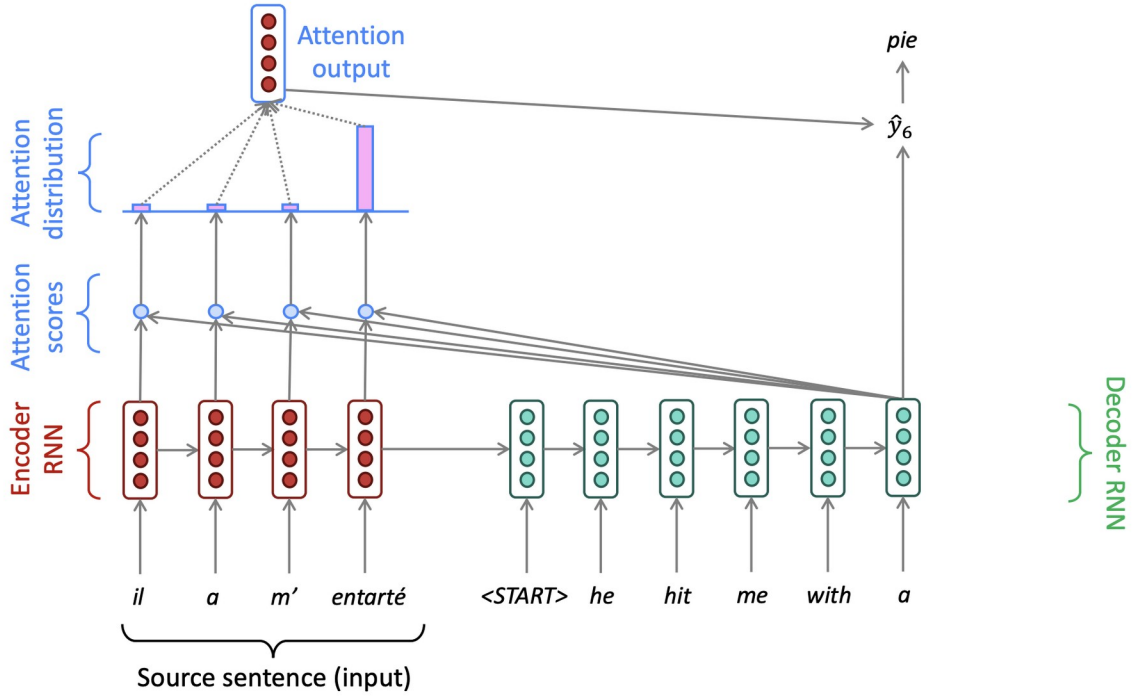
Attention's pros

- significantly improves NMT performance
- more "human-like" model of the MT process
- solves bottleneck problem
- helps with vanishing gradient problem
- provides some interpretability

Attention

helps with vanishing gradient problem





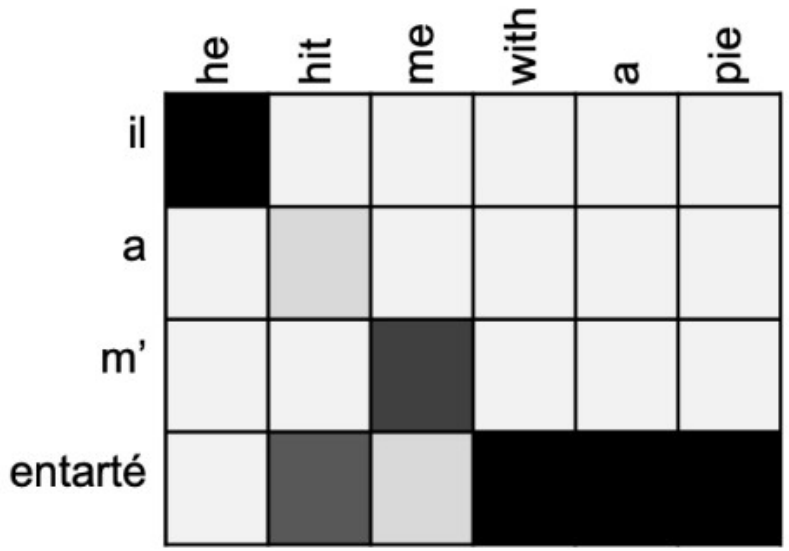
$$\frac{\partial E}{\partial h_1} = \frac{\partial E}{\partial s_t} \times \left(\prod_{j=2}^{t} \frac{\partial s_j}{\partial s_{j-1}} \right) \times \frac{\partial \mathbf{s_0}}{\partial \mathbf{h_N}} \times \left(\prod_{k=2}^{N} \frac{\partial h_k}{\partial h_{k-1}} \right)$$

$$\frac{\partial E}{\partial h_1} = \frac{\partial E}{\partial z_t} \frac{\partial z_t}{\partial c_t} \cdot \boldsymbol{\alpha_{t1}} + (\text{기존 디코더 체인을 거쳐가는 경로})$$

Attention

provides some interpretability

We get (soft) alignment for free!



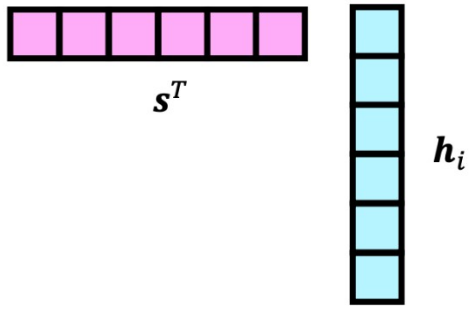
,

Attention

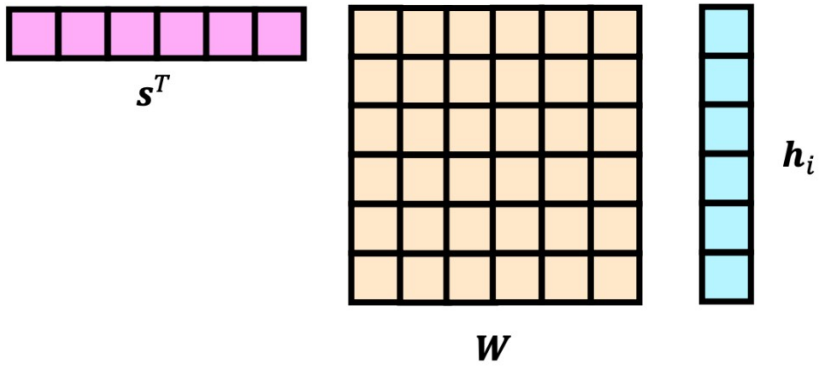
Attention variants

There are several ways you can compute $\boldsymbol{e} \in \mathbb{R}^N$ from $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_1}$ and $\boldsymbol{s} \in \mathbb{R}^{d_2}$:

- Basic dot-product attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{h}_i \in \mathbb{R}$
  - This assumes $d_1 = d_2$. This is the version we saw earlier.



- Multiplicative attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{W} \boldsymbol{h}_i \in \mathbb{R}$ [Luong, Pham, and Manning 2015]
  - Where $\boldsymbol{W} \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix. Perhaps better called "bilinear attention"



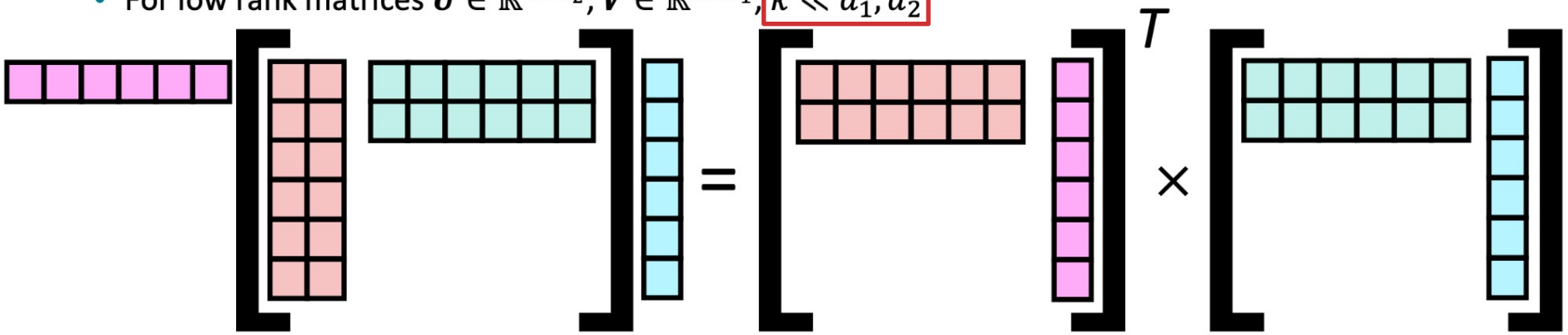
d1 d2

W ?

Attention

bottleneck problem

- Reduced-rank multiplicative attention: $e_i = s^T(\boldsymbol{U}^T\boldsymbol{V})h_i = (\boldsymbol{U}s)^T(\boldsymbol{V}h_i)$
  - For low rank matrices $\boldsymbol{U} \in \mathbb{R}^{k \times d_2}$, $\boldsymbol{V} \in \mathbb{R}^{k \times d_1}$, $k \ll d_1, d_2$



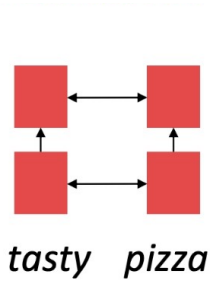
- Additive attention: $e_i = \boldsymbol{v}^T \tanh(\boldsymbol{W}_1 \boldsymbol{h}_i + \boldsymbol{W}_2 \boldsymbol{s}) \in \mathbb{R}$ [Bahdanau, Cho, and Bengio 2014]
  - Where $\boldsymbol{W}_1 \in \mathbb{R}^{d_3 \times d_1}$, $\boldsymbol{W}_2 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $\boldsymbol{v} \in \mathbb{R}^{d_3}$ is a weight vector.
  - d_3 (the attention dimensionality) is a hyperparameter
  - "Additive" is a weird/bad name. It's really using a feed-forward neural net layer.

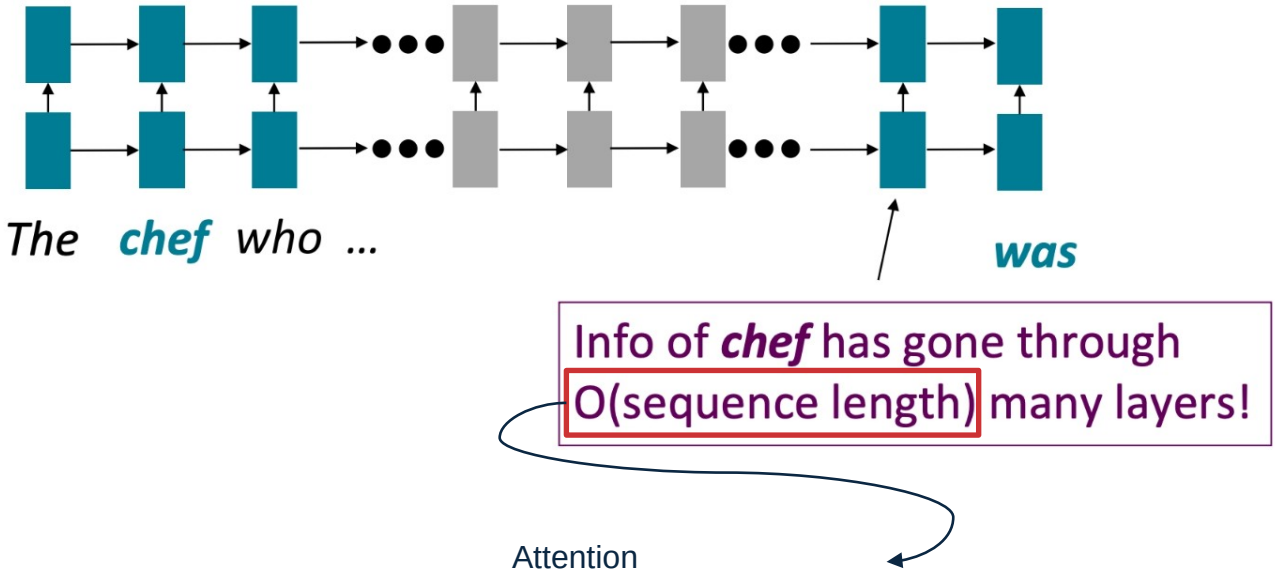
RNN

Linear interaction distance

RNN ,

.





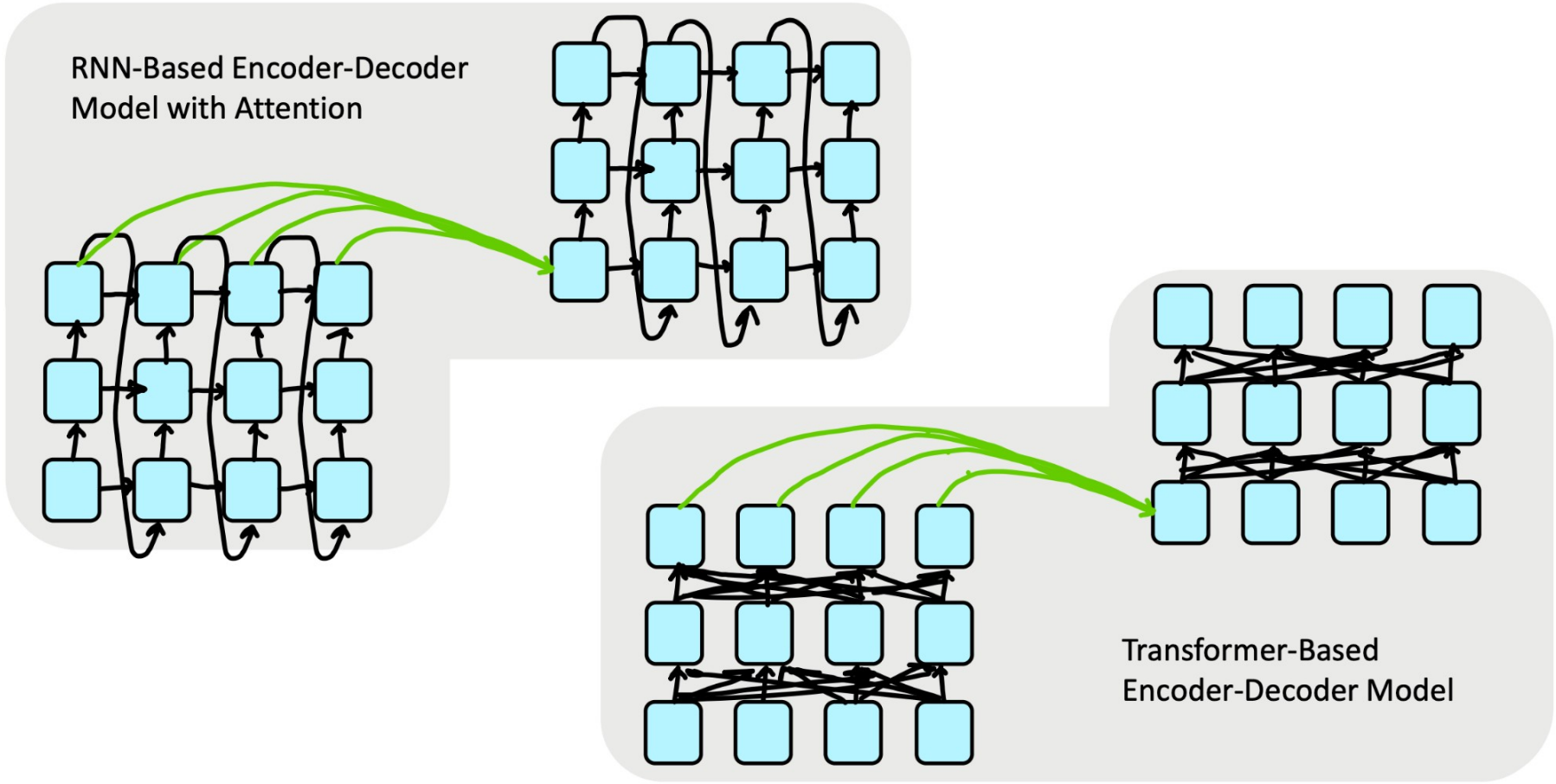
RNN

Lack of parallelizability

RNN

,

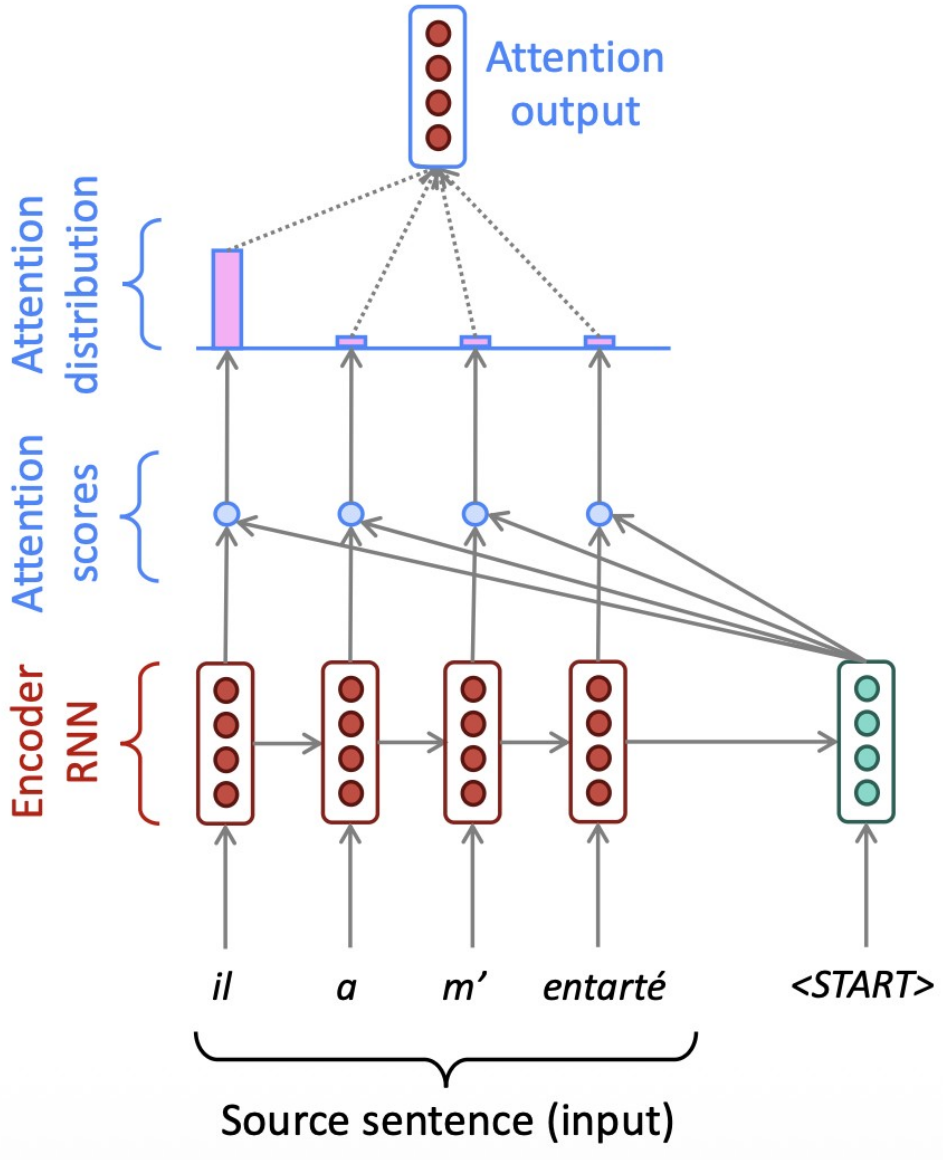
.



The Transformer Architecture

From Cross-Attention to Self-Attention

Attention between two sequences



- **Cross-Attention**
  - ✓ Query <- Decoder
  - ✓ Keys & Values <- Encoder

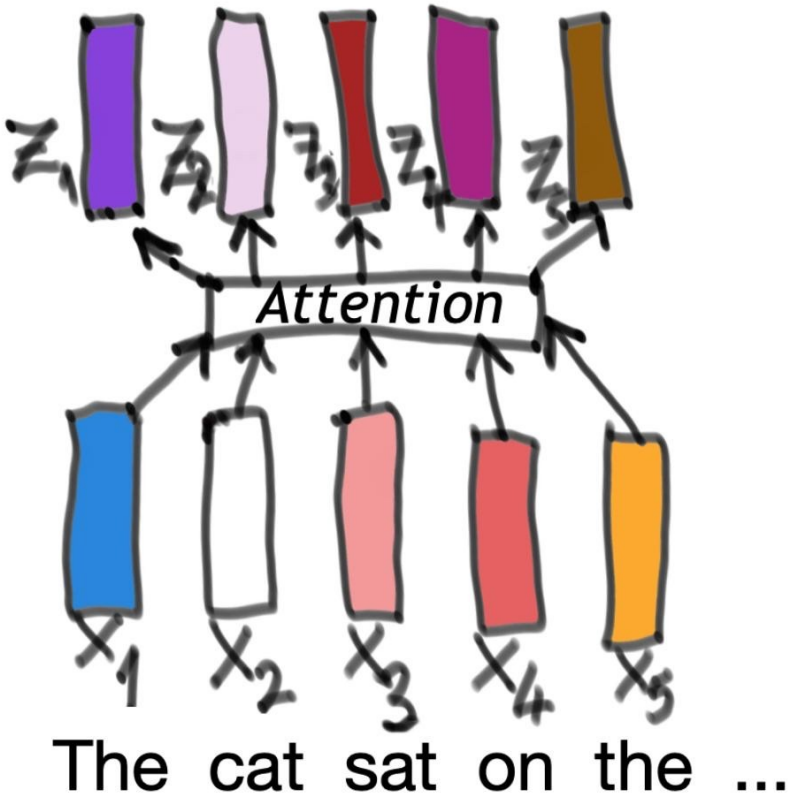
⇩

Different source of Q and K,V

Self Attention

Q,K and V from the same sequence

- **How Self-Attention Works**
  - In full self-attnetion, every token attends to every other token



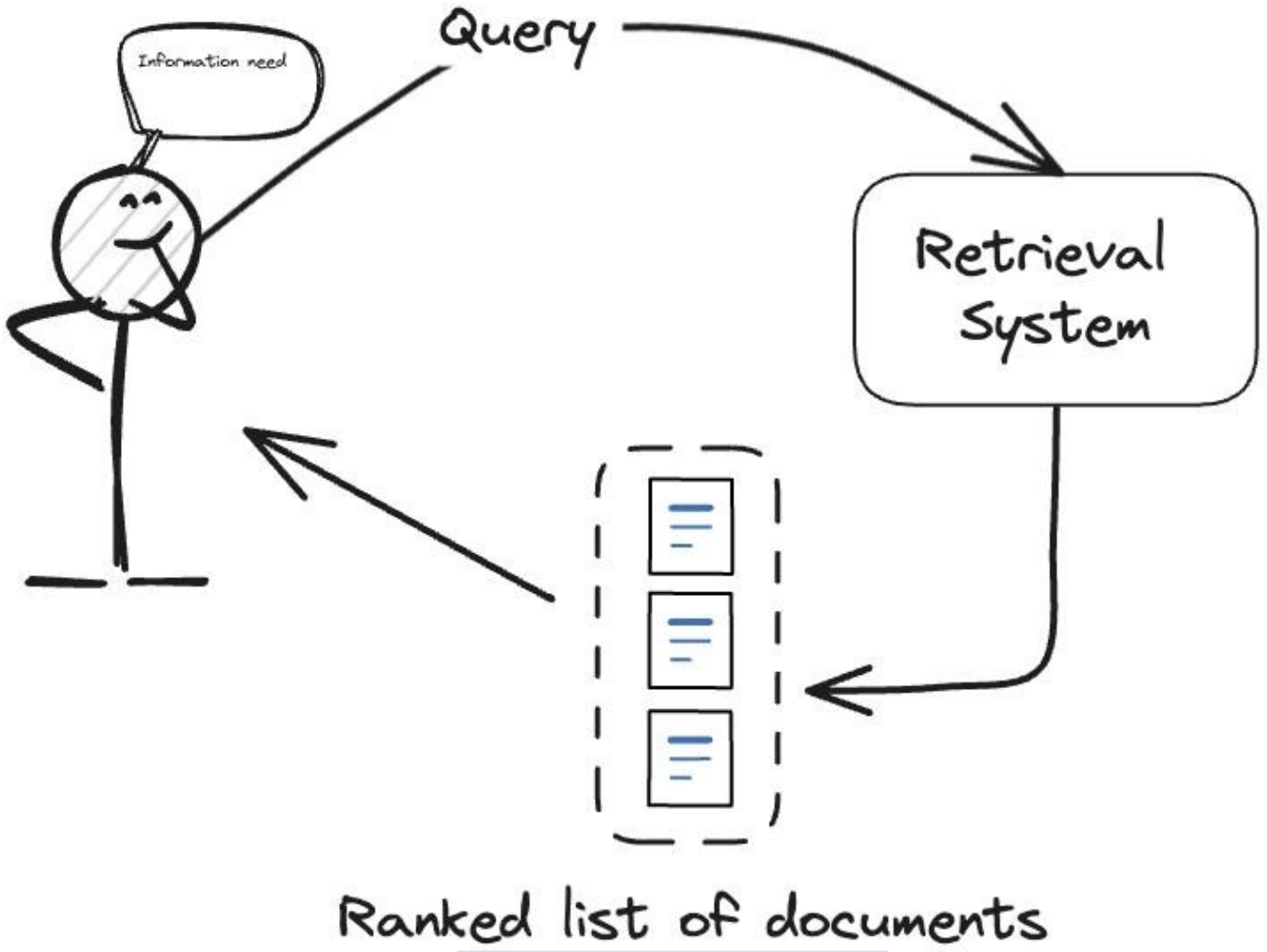
How does a token decide where to attend?

Each token gathers information from the entire sequence

Query(Q), Key(K), Value(V)

How a token decides where to attend

- **Query**
  - What information am I looking for?
- **Key**
  - What information do I match with?
- **Value**
  - What information do I provide?



Weighted values

Scaled Dot-Product Attention

From similarity scores to weighted information

① Query-Key Matching

: compute similarity scores

$$A = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)$$

② Scale and Normalize

: convert scores into attention weights

③ Weighted Sum of Values

: retrieve and combine information

$$O = AV$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

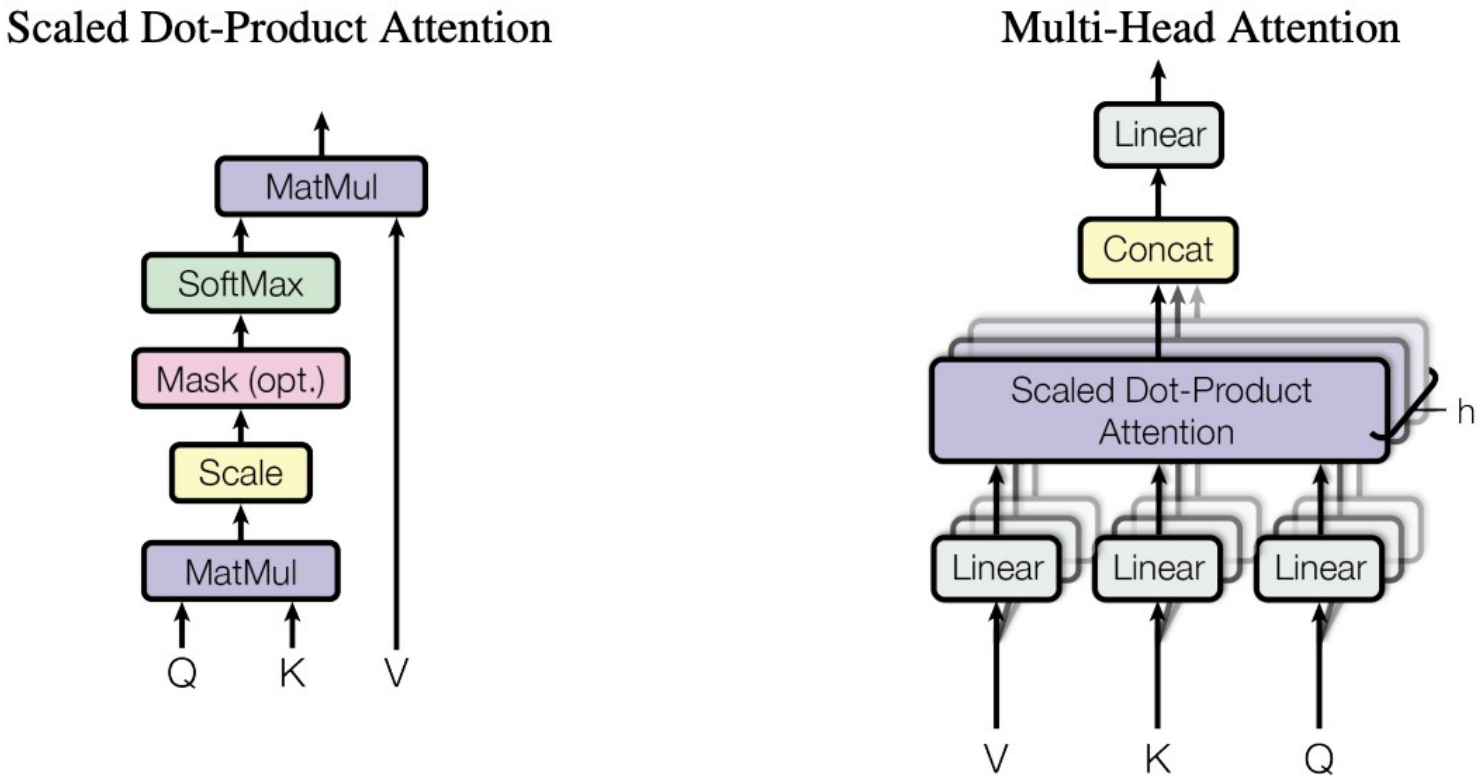
Agenda

Core Transformer Concepts

1. (Scaled Dot Product Self Attention)
2. Multi-Head Attention + Masking
3. Residual + Normalization
4. Feed Forward Layer
5. Positional Encoding

Scaled Dot-Product Attention to Multi-Head Attention

Multi-Head Attention

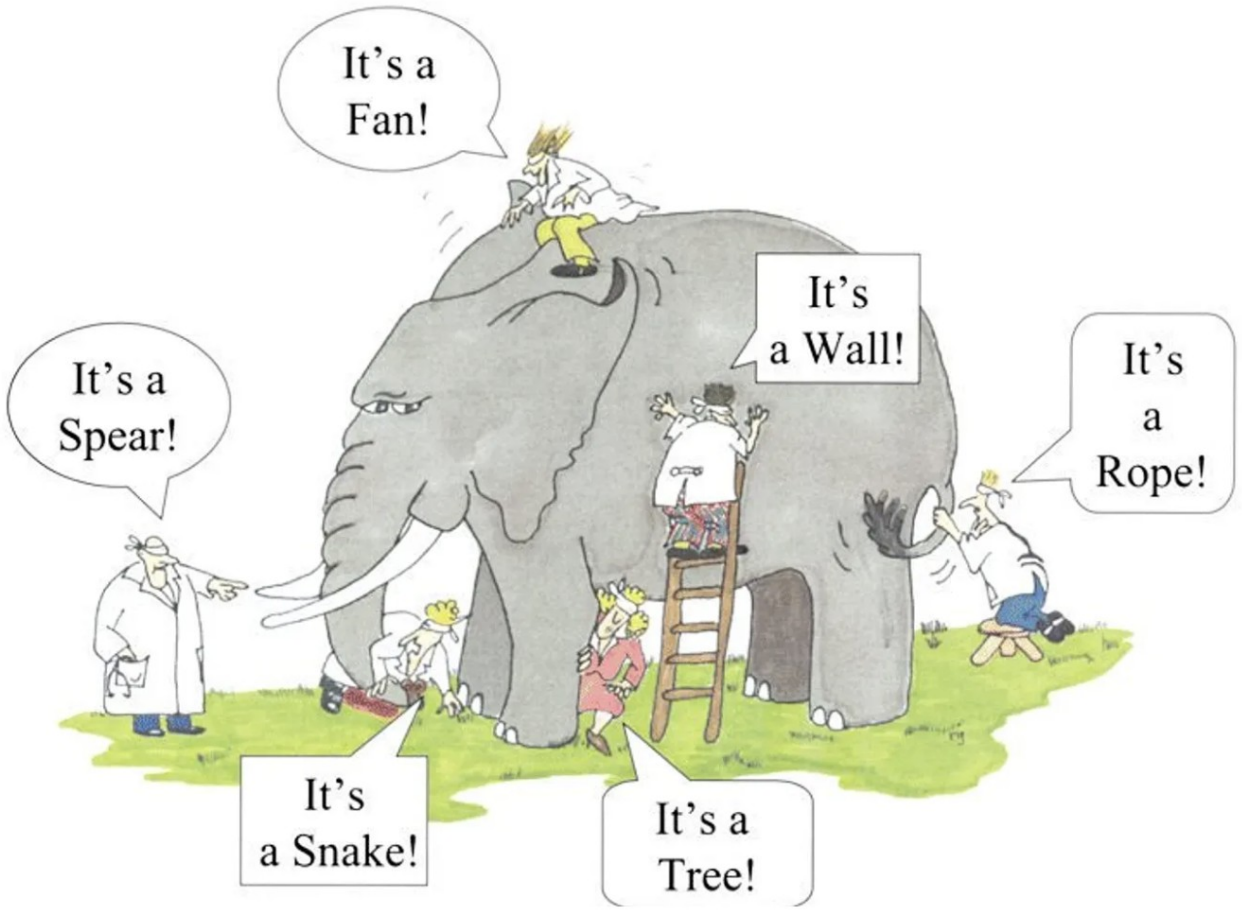


Vaswani, Ashish, et al. "Attention is all you need." *NeurIPS 2017*

Scaled Dot-Product Attention to Multi-Head Attention

Multi-Head Attention

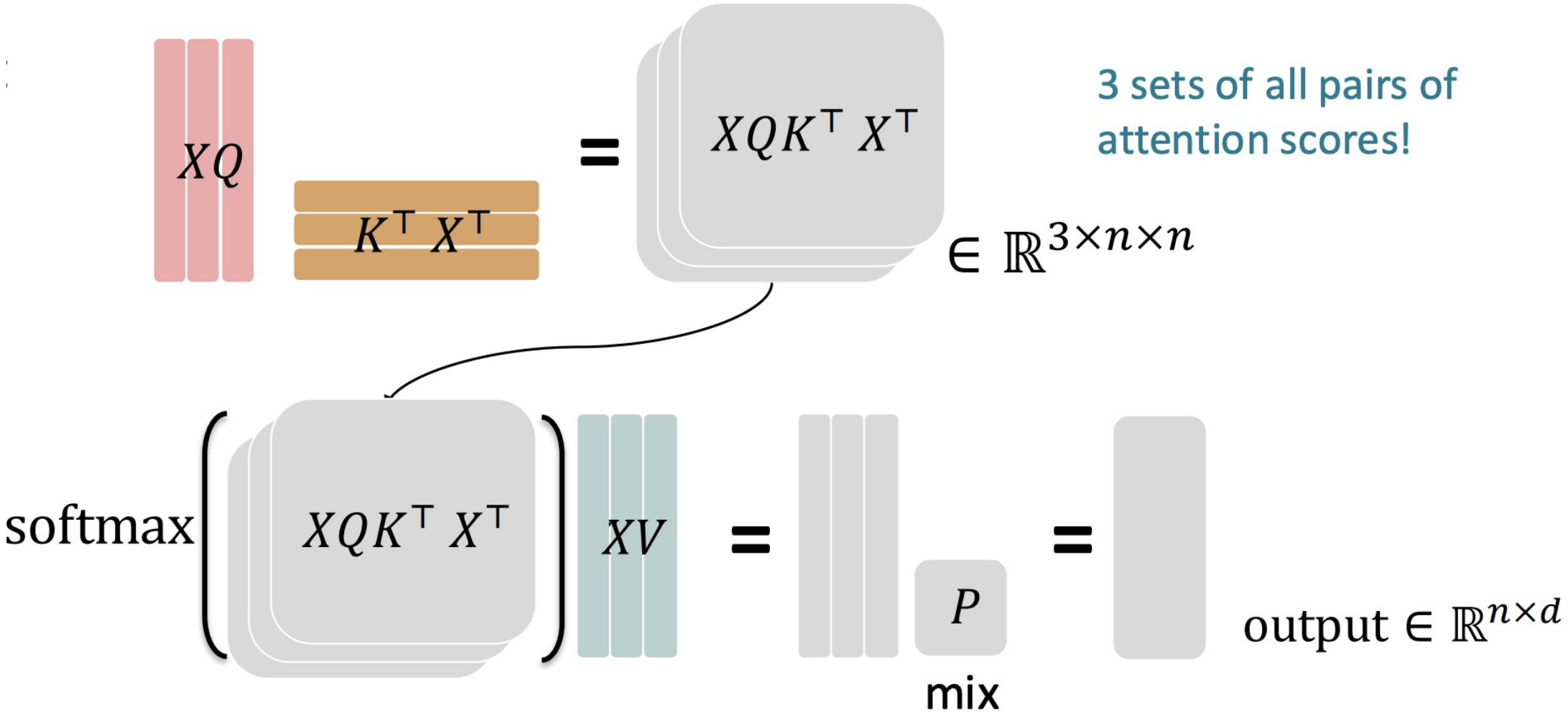
- **Why Multi-Head Attention?**



Song, "Mastering Llama Multi-Head Attention " blog, 2023.

Multi-Head Attention

Multi-Head Attention

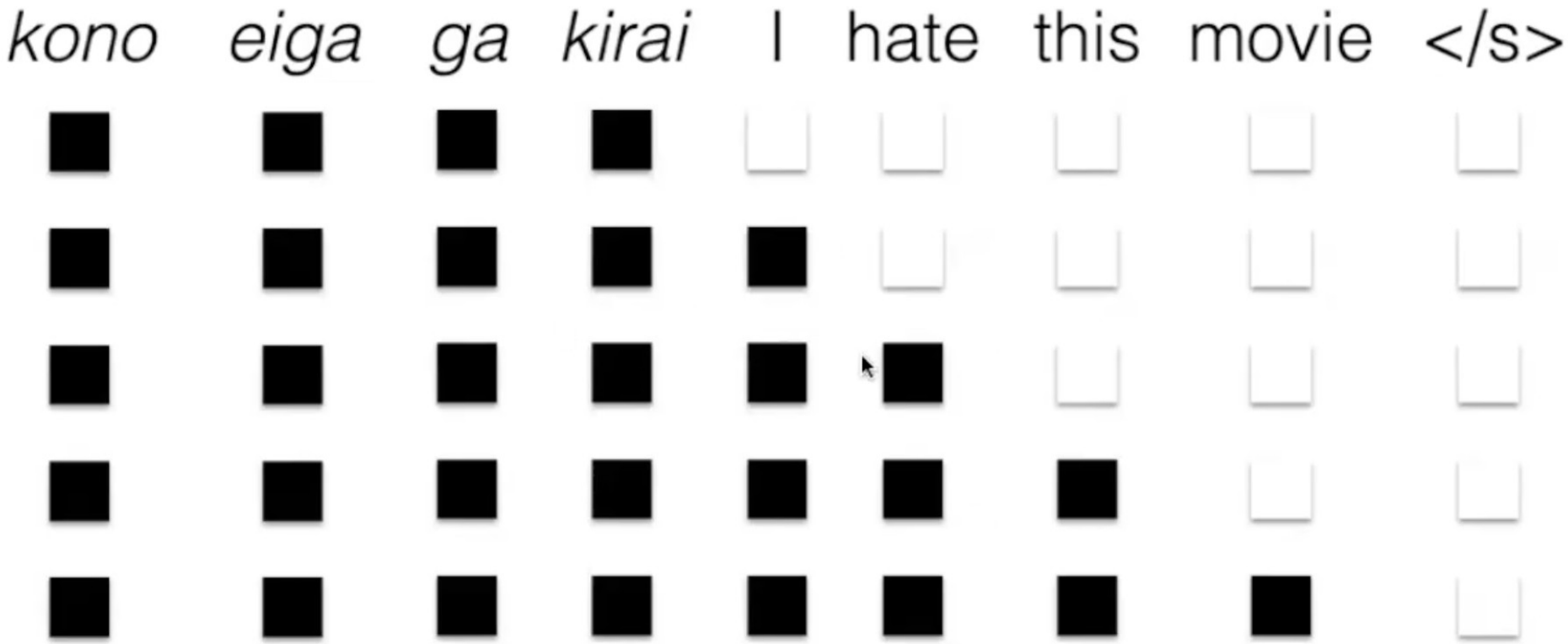


Masking

Multi-Head Attention

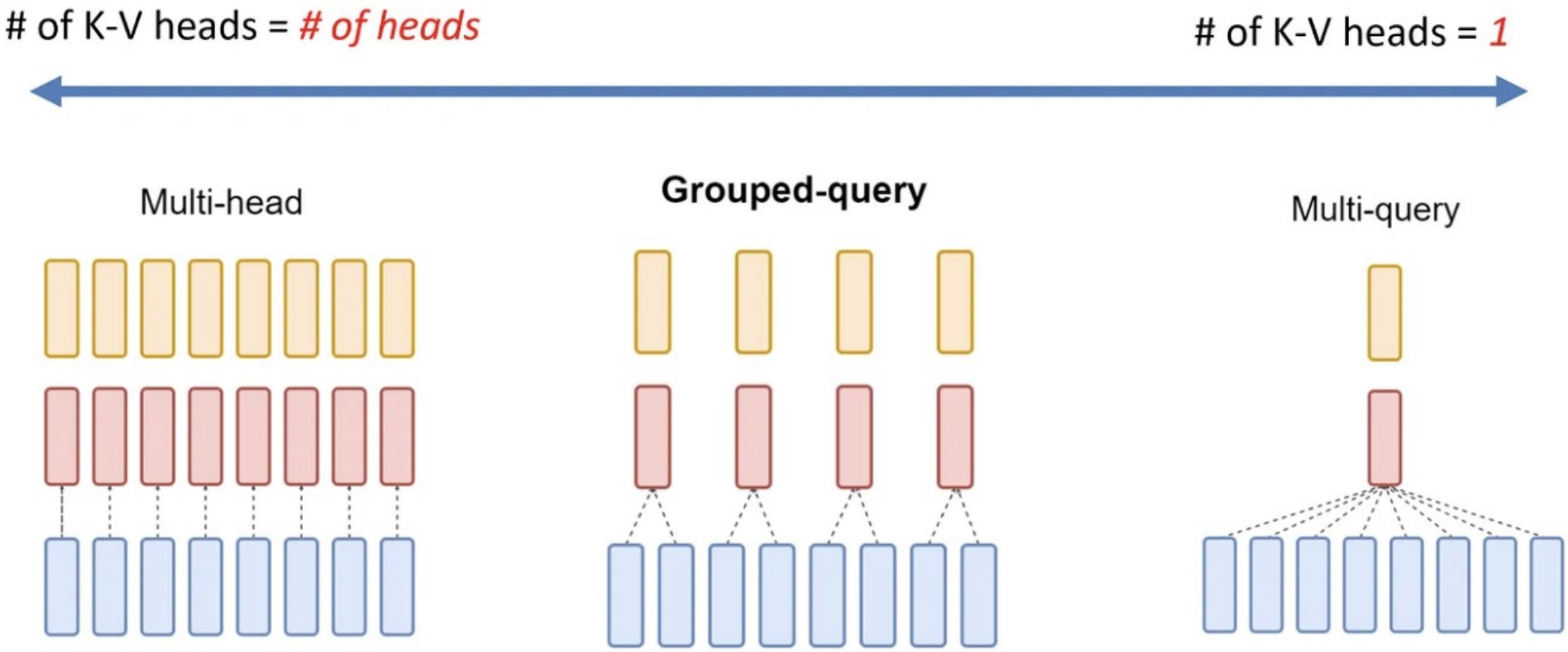
- **Goal: parallelize** operations while **not looking at the future**

$$e_{ij} = \begin{cases} q_i^\top k_j, j \leq i \\ -\infty, j > i \end{cases}$$



Grouped-Query Attention(GQA)

Multi-Head Attention - Modern Variants



Ainslie, Joshua, et al. "Gqa: Training generalized multi-query transformer models from multi-head checkpoints." *EMNLP*. 2023.

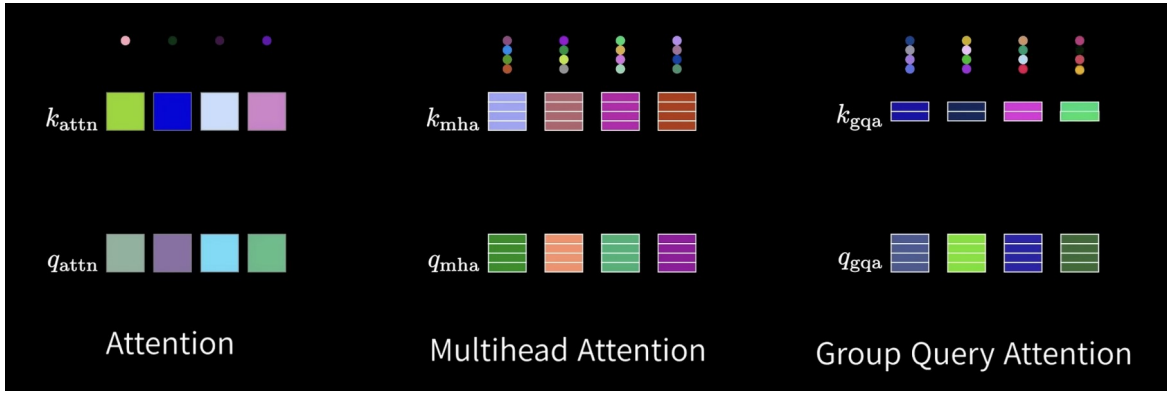
Grouped-Query Attention(GQA)

Multi-Head Attention - Modern Variants

- **Grouped-Query Attention**
  - Single-Head Attention: single perspective
  - Multi-Head Attention: multiple independent perspectives

Multi-Head Attention

$$Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$$



- **Method**
  - Multiple Query heads perform independent comparisons
  - against the same shared Key-Value group
- **Advantages**
  - Memory Efficiency: reduce the size of KV Cache
  - Performance: Maintains performance levels nearly identical to Multi-Head Attention.

	BoolQ	PIQA	SIQA	Hella-Swag	ARC-e	ARC-c	NQ	TQA	MMLU	GSM8K	Human-Eval
MHA	**71.0**	**79.3**	48.2	75.1	71.2	**43.0**	12.4	44.7	**28.0**	4.9	**7.9**
MQA	70.6	79.0	47.9	74.5	71.6	41.9	**14.5**	42.8	26.5	4.8	7.3
GQA	69.4	78.8	**48.6**	**75.4**	**72.1**	42.5	14.0	**46.2**	26.9	**5.3**	**7.9**

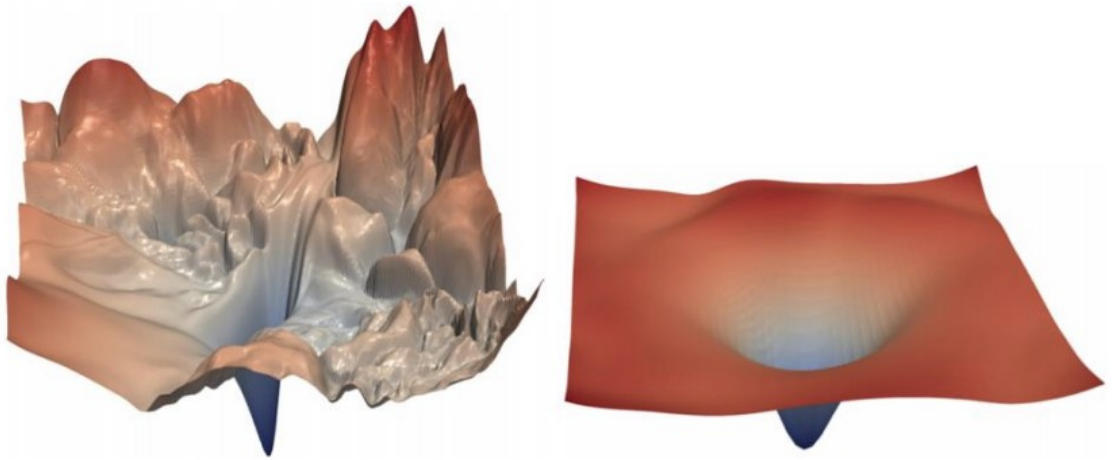
Touvron, Hugo, et al. "Llama 2: Open foundation and fine-tuned chat models." *arXiv* (2023).

Residual Connection

Residual Connection

- Prevents **vanishing gradients** and allows f to learn the difference from the input

$$\text{Residual}(\mathbf{x}, f) = f(\mathbf{x}) + \mathbf{x}$$



[no residuals] [residuals]

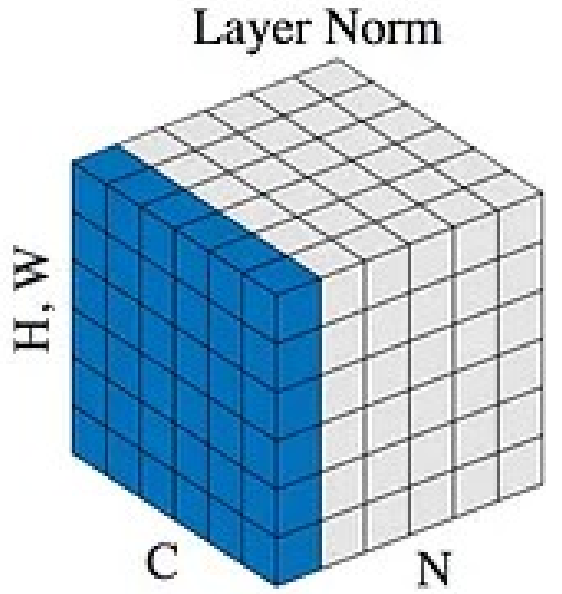
LI, Hao, et al. "Visualizing the loss landscape of neural nets." NeurIPS 2017.

Layer Normalization

Normalization

- **Limitations of Batch Normalization**
  - Batch size dependency: Sensitive to batch size

- **Layer Normalization**
  - Normalizes independently **for each sample**
  - Suitable for sequential data (Transformers or RNNs)



$$\text{LayerNorm}(\mathbf{x}; \mathbf{g}, \mathbf{b}) = \frac{\mathbf{g}}{\sigma(\mathbf{x})} \odot (\mathbf{x} - \mu(\mathbf{x})) + \mathbf{b}$$

bias

vector
stddev

vector
mean

RMS Normalization

Normalization - Modern Variants

- **RMS Normalization**
  - **Faster!** (**no mean calculation**, no bias term to store)

$$y = \frac{x}{\sqrt{||x||_2^2 + \varepsilon}} * \gamma$$

- **Layer Normalization**

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Pre-vs-post-norm

Normalization - Modern Variants

- **Better for gradient propagation!**
- Keep the residuals clean!(learn the identify function)

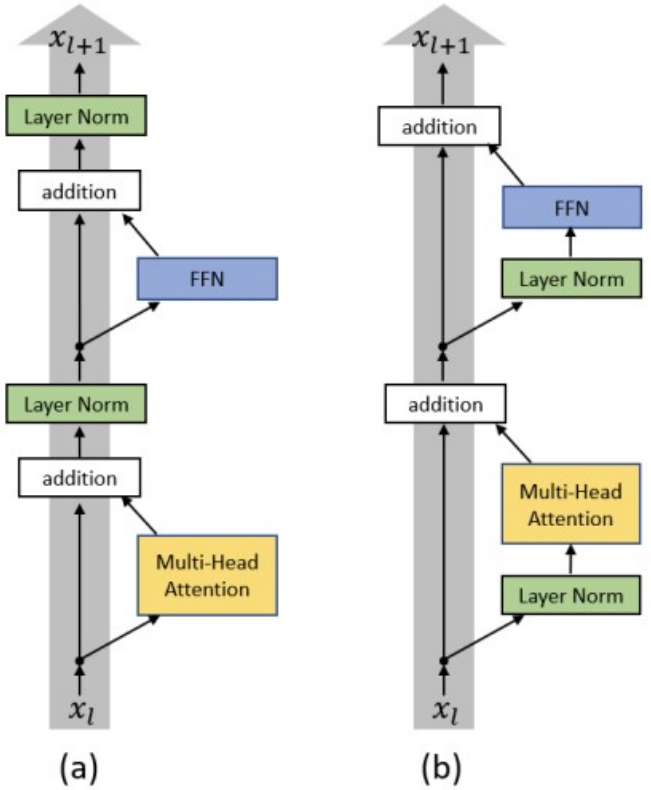


Figure 1. (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

Xiong, Ruibin, et al. "On layer normalization in the transformer architecture." *ICML*, 2020.

Activations

Feed Forward Network - Modern Variants

- **Original Transformer**

ReLU: $f(x) = \max(0, x)$

- **Gated Activations(GLU) - SwiGLU**
  - Swish (smooth, non-monotonic) $f(x) = x\sigma(x)$ Sigmoid: $\sigma(x) = \frac{1}{1+\exp(-x)}$
  - GLU (uses one linear projection to produce the "content" [left], and another to produce the gate [right])

$$\mathrm{GLU}(x) = (W_1 x) \odot \sigma(W_2 x)$$

  - SwiGLU(plugs Swish in as the activation function inside GLU)

$$\mathrm{FFN}_{\mathrm{SwiGLU}}(x, W, V, W_2) = (\mathrm{Swish}_1(xW) \otimes xV)W_2$$

gate!

Bias Terms

Feed Forward Network - Modern Variants

- **Original Transformer**

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- **Modern Variants don't have bias terms**
  - Memory (similar to RMS Normalization)
  - optimization stability

$$FFN(x) = \sigma(xW_1)W_2$$

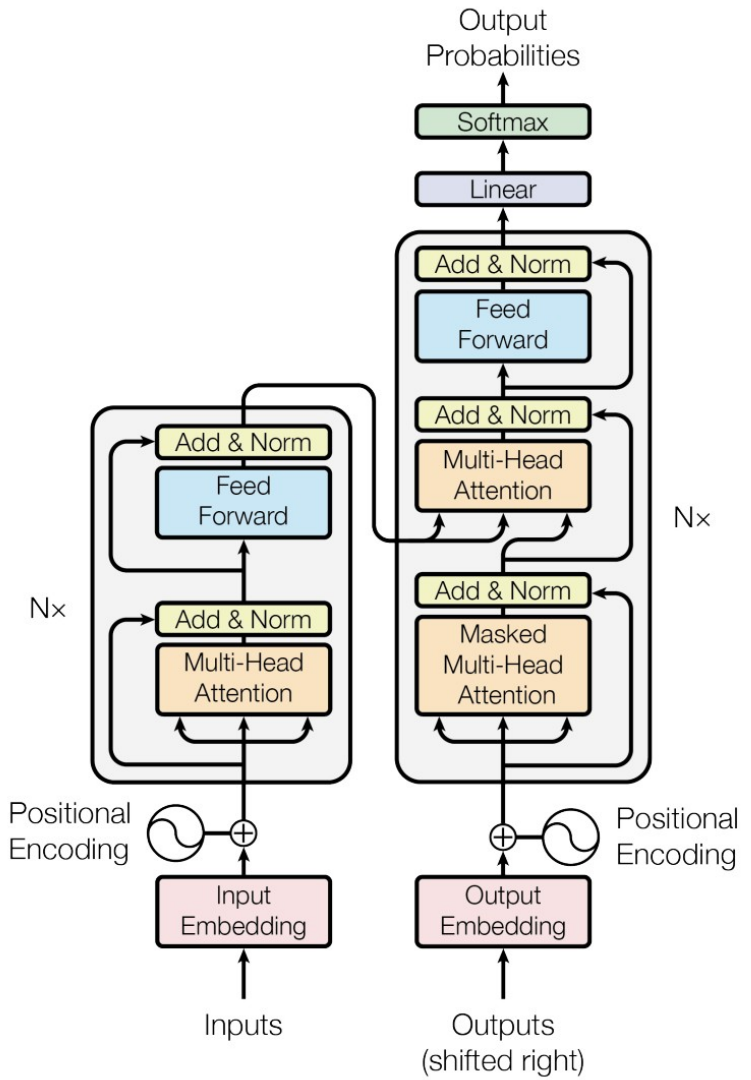


Figure 1: The Transformer - model architecture.

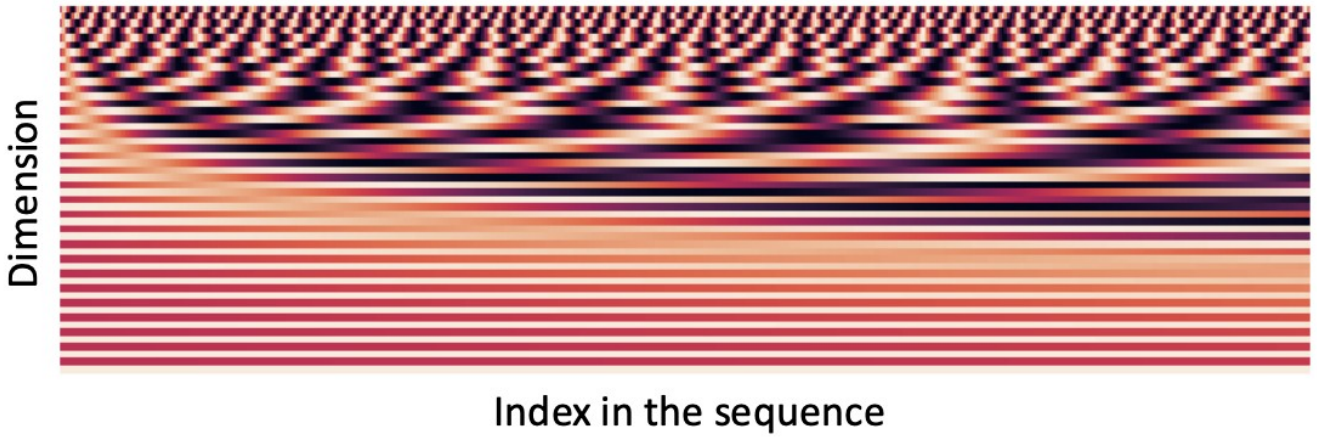
Vaswani, Ashish, et al. "Attention is all you need." *NeurIPS 2017*

Absolute Positional Encoding

Positional Encoding

$$\widetilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

- **Sine Positional Embeddings**



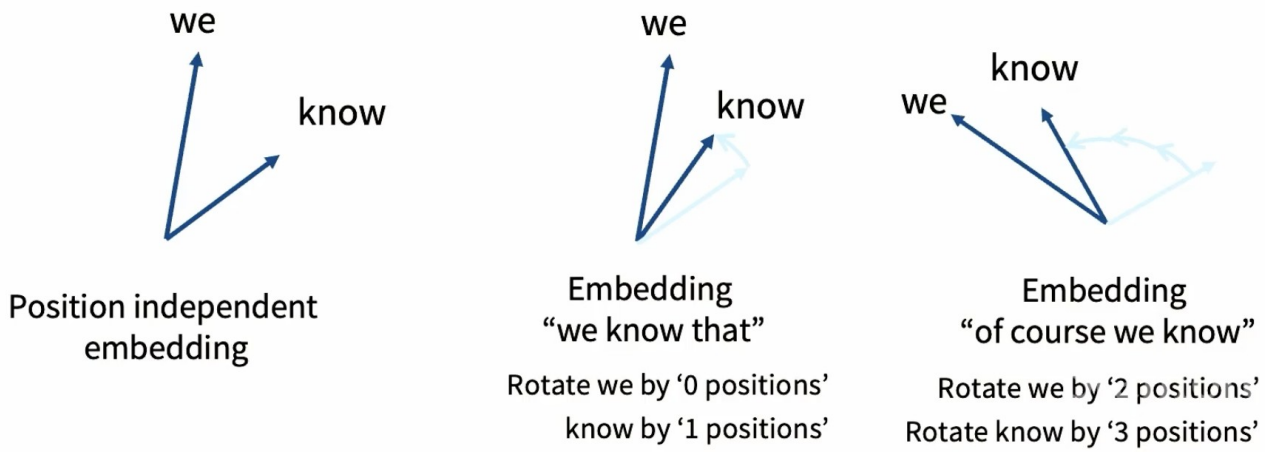
$$\boldsymbol{p}_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$

- **Learned absolute positional Embedding**

Rotary Positional Encodings (RoPE)

Positional Encoding - Modern Variants

- **Goal: the dot product to result in a function of relative position**
- Idea: Leverage nice properties of rotations



$$\langle f(x,i), f(y,j) \rangle = g(x,y,i-j)$$

Rotary Positional Encodings (RoPE)

Positional Encoding - Modern Variants

$$\langle f(x,i), f(y,j) \rangle = g(x,y,i-j)$$

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

$$(R_{\theta_1} q)^\top (R_{\theta_2} k) = q^\top R_{\theta_1}^\top R_{\theta_2} k$$

$$= q^\top R_{\theta_2 - \theta_1} k$$

- **Dot product only depends on the difference between θ_2 and θ_1 !!**

Original Transformer vs. Modern Variants

	Vaswani et al.	LLama	Llama 2
Norm Position	Post	**Pre**	**Pre**
Norm Type	LayerNorm	**RMSNorm**	**RMSNorm**
FFN/ Activation	ReLU	**SwiGLU**	**SwiGLU**
Positional Encoding	Sinusoidal	**RoPE**	**RoPE**
Attention	Multi-head	Multi-head	**Grouped-query**

Two Types of Transformers

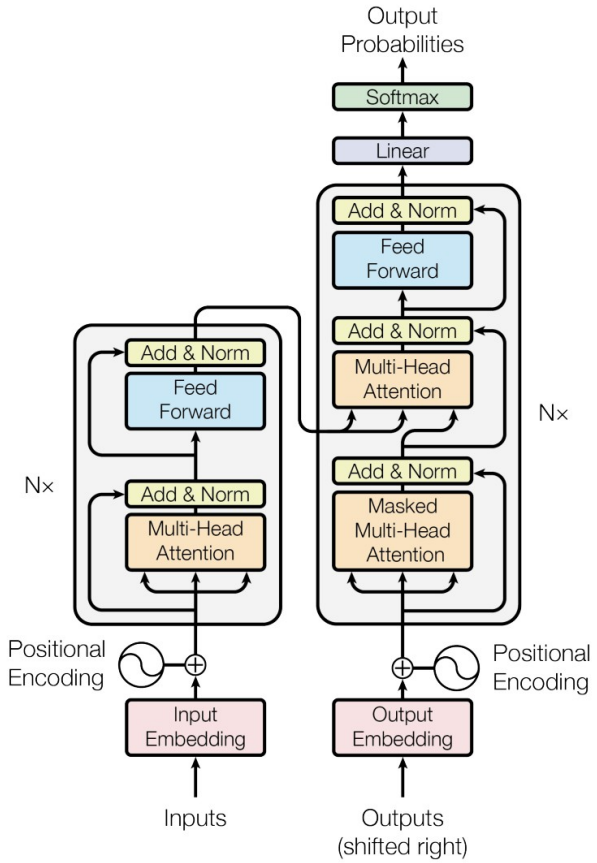
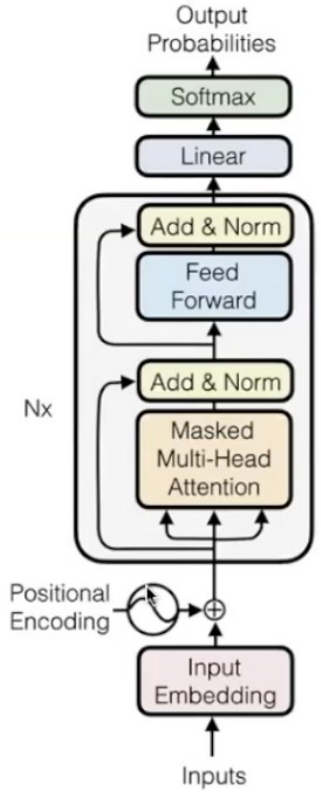


Figure 1: The Transformer - model architecture.

Encoder-Decoder Model



Decoder Only Model
(e.g. GPT, LLama)

Vaswani, Ashish, et al. "Attention is all you need." *NeurIPS 2017*

Cross Attention

Encoder–Decoder Cross Attention

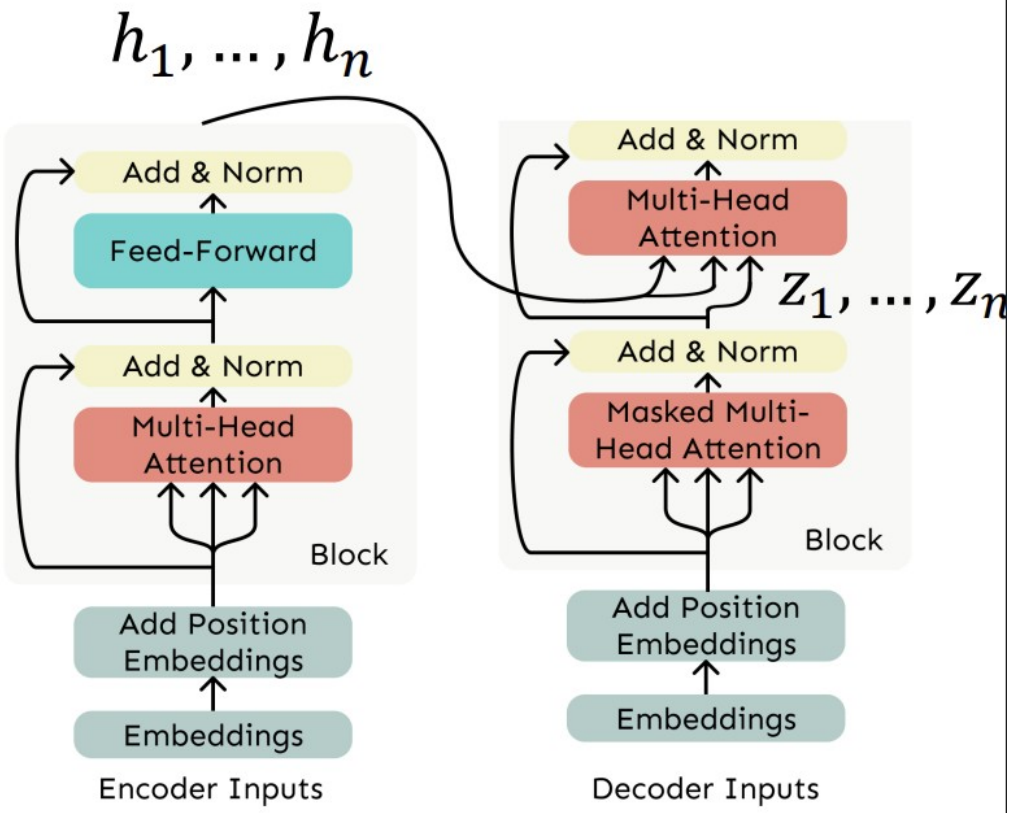
- Let h1, … , hn be output vectors from the Transformer encoder $x_i \in \mathbb{R}^d$
- Let z 1 , … , z n be input vectors from the Transformer decoder $z_i \in \mathbb{R}^d$

- Then keys and values are drawn from the **encoder** (like a memory)

$$k_i = Kh_i, v_i = Vh_i.$$

- And the queries are drawn from the **decoder**

$$q_i = Qz_i.$$



Limitations

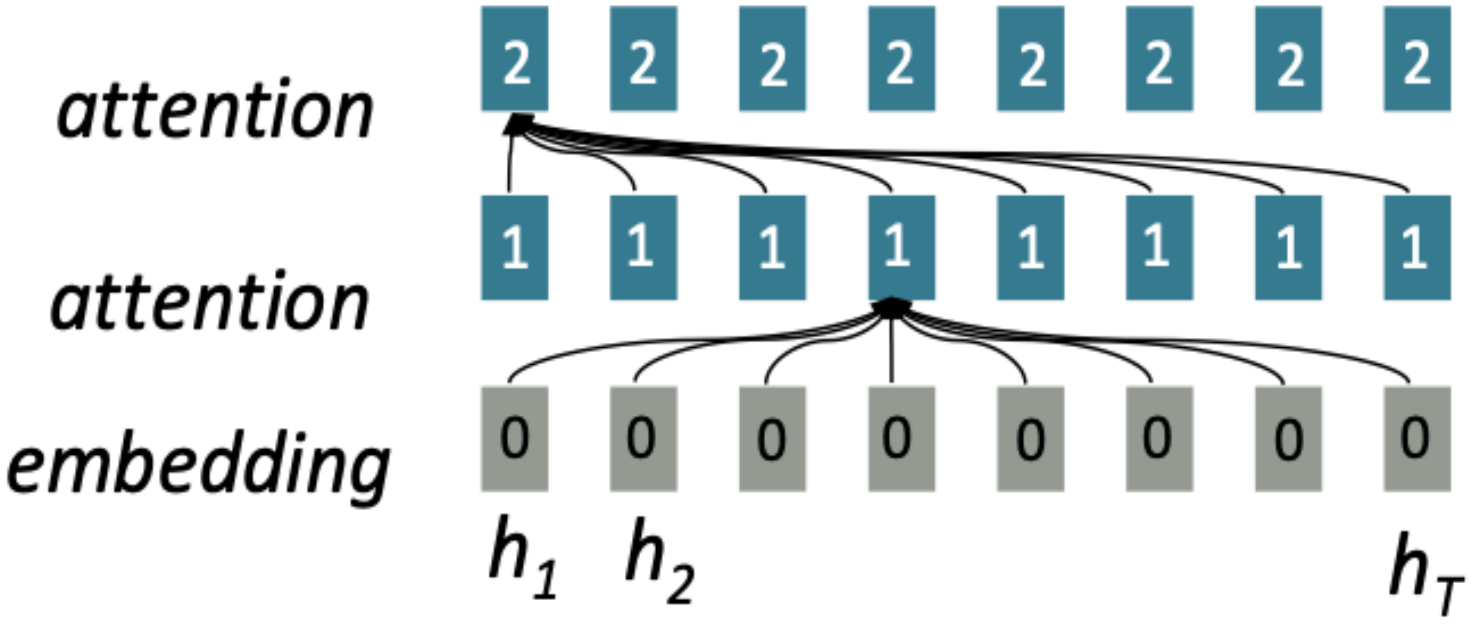
- **Quadratic Compute in Self Attention**
  - our computation grows quadratically with the sequence length

- **Positional Representations**

Evolution of Efficient Attention

The Quadratic Bottleneck

- **The Cost of Full Self-Attention**
  - Every token attends to every other token



Pairwise Comparisons

N Queries
×
N Keys
———————
N² Attention Scores

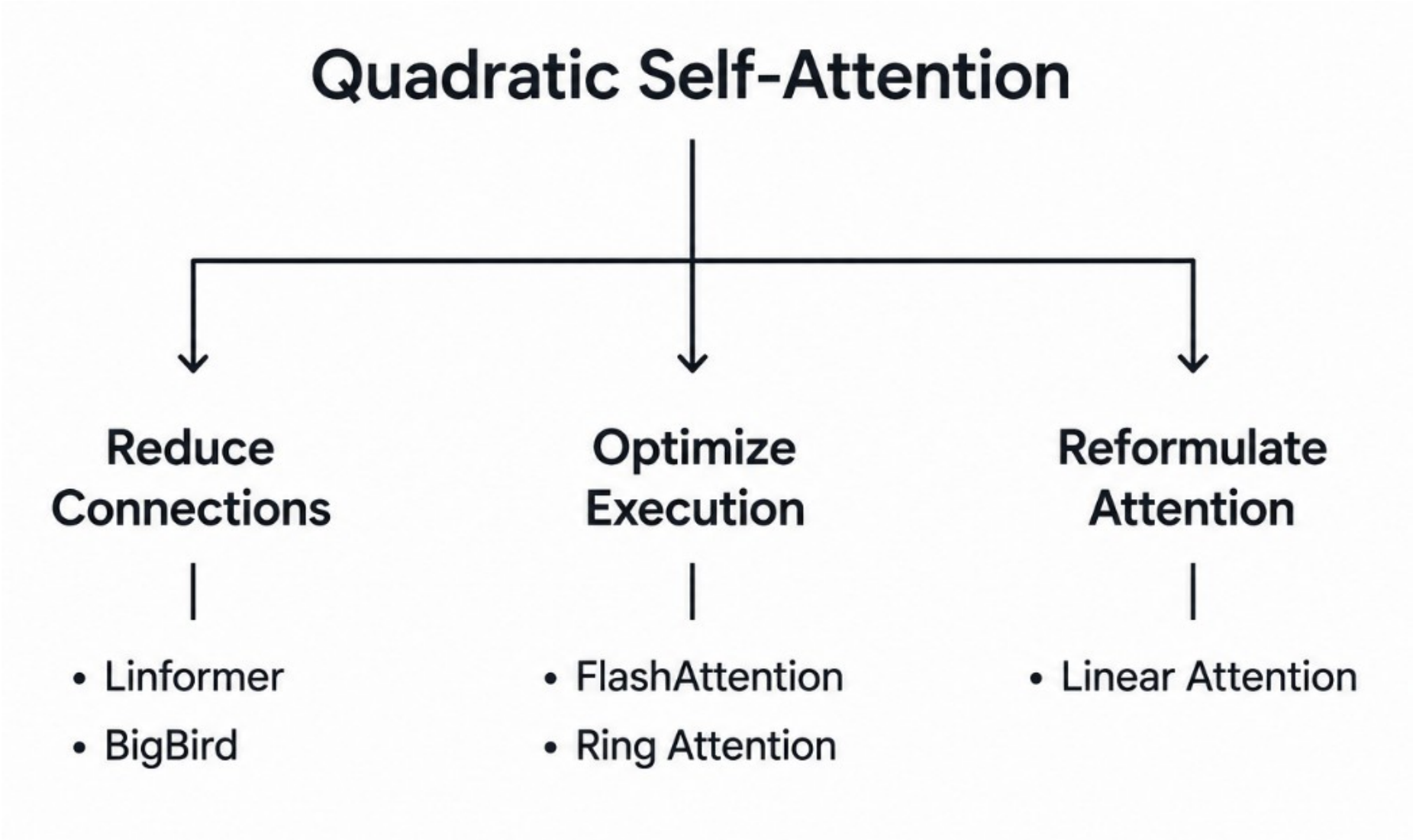
$$S = QK^T \in \mathbb{R}^{N \times N}$$

Quadratic Computation

Quadratic Memory

Can we reduce this cost without sacrificing performance?

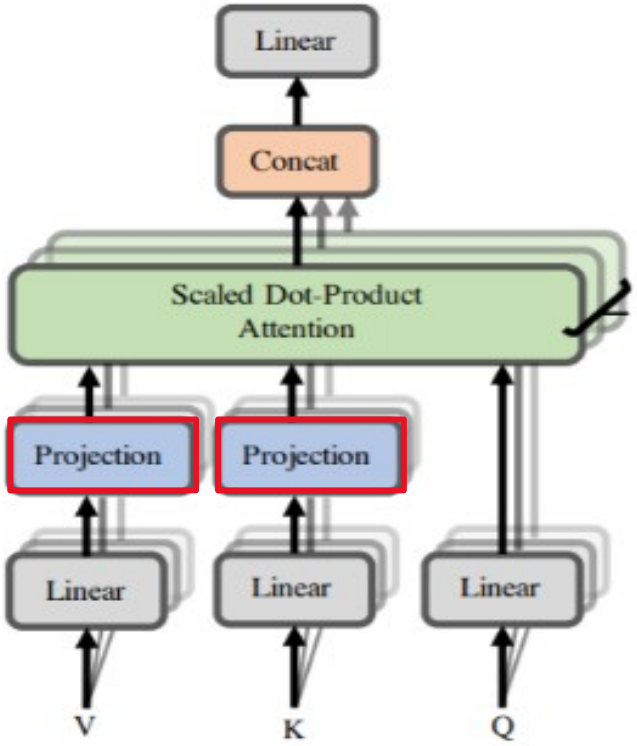
Three Directions for Efficient Long-Context Modeling



Reduce Connections

Reduce the number of token-to-token interactions

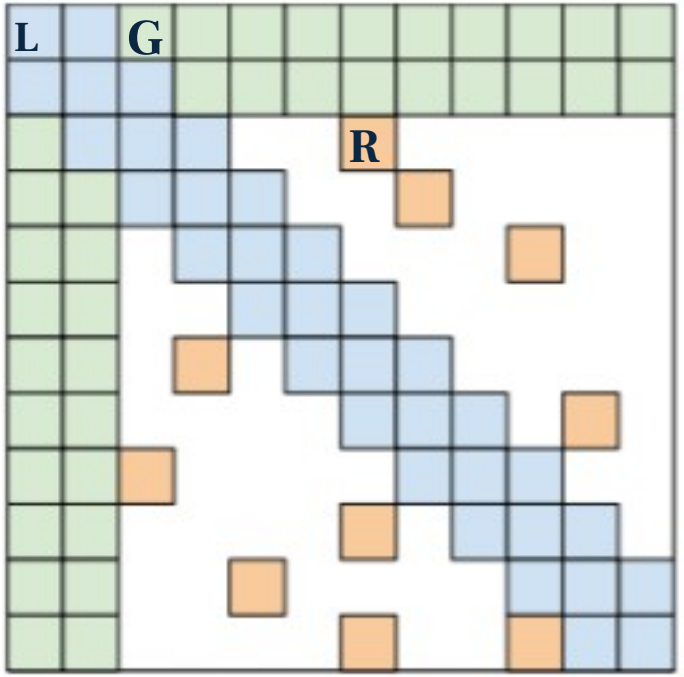
• Linformer



- Project K and V into a lower-dim space

✓ Lower computation

• BigBird



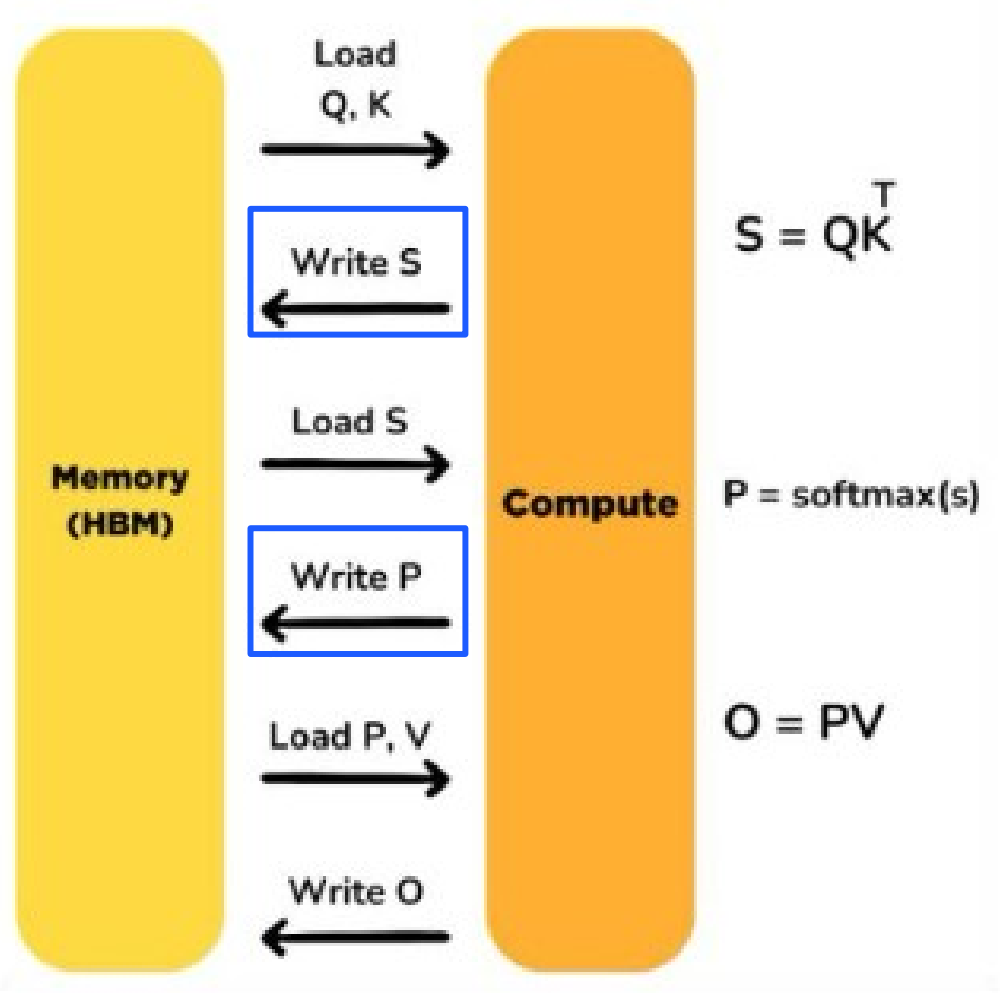
- Attend to only a subset of tokens

✓ Sparse computation

Optimize Execution (1)

Flash Attention : Change the execution, not the attention

- **Repeated HBM ↔ SRAM transfers**



Repeated HBM read/write

↓

Materialize intermediate matrics (S,P)

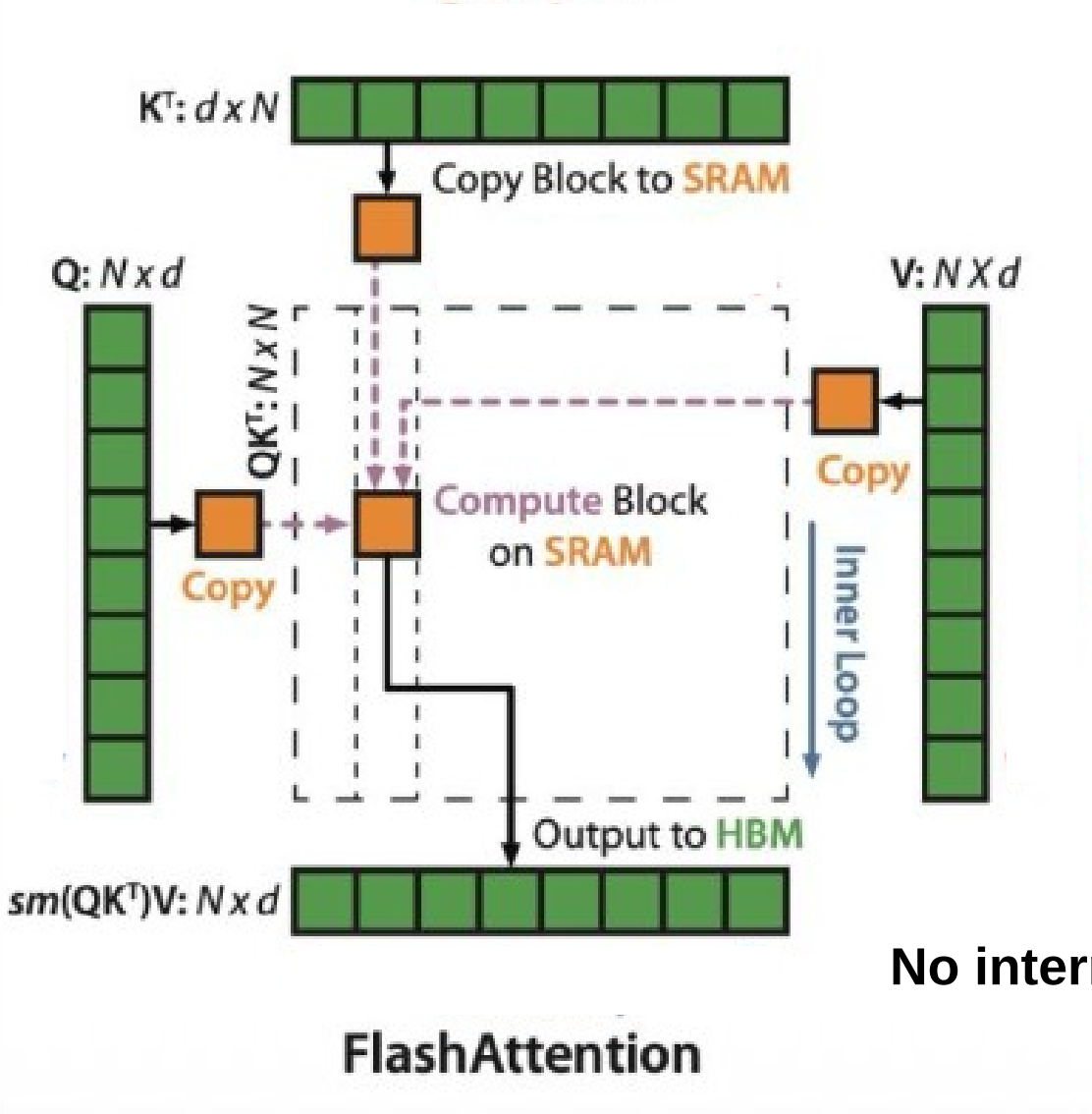
↓

Memory movement dominates execution

The bottleneck is **memory I/O**, not arithmetic

Flash Attention

Compute in SRAM, write once



① Load K/V block into SRAM

↓

② Compute attention inside SRAM

↓

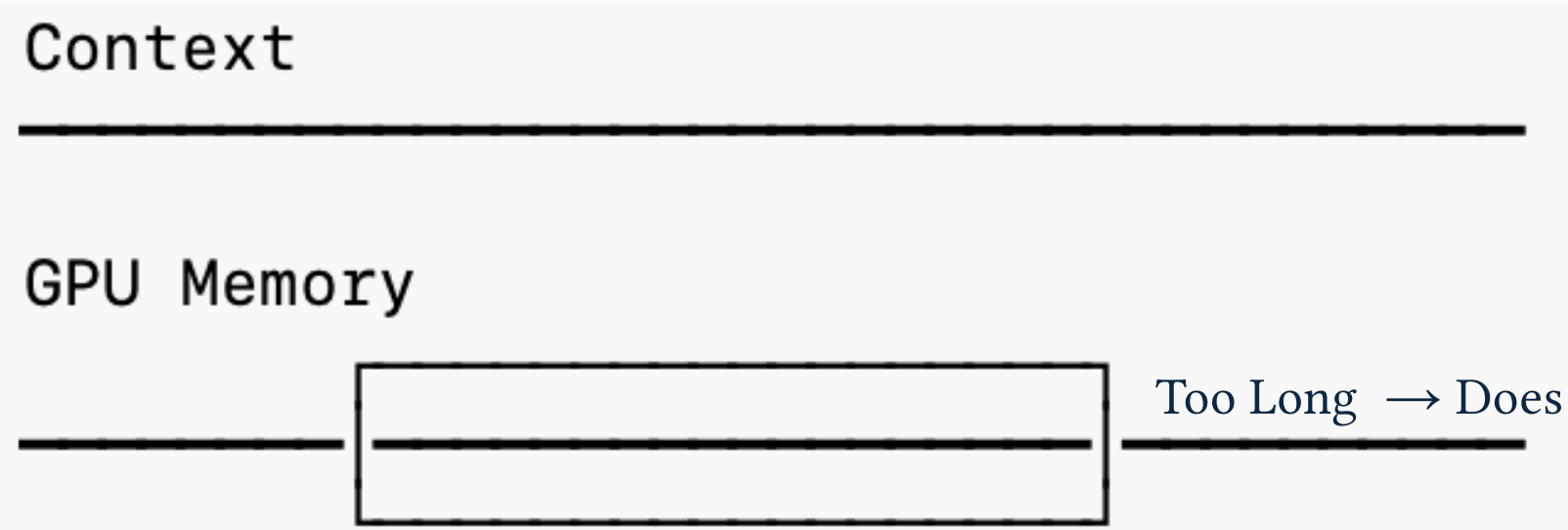
③ Write the output once

No intermediate attention matrices are written to HBM

Optimize Execution (2)

single GPU remains the bottleneck

- **Context grows beyond device memory**
  - A single GPU cannot store the full sequence and its intermediate states



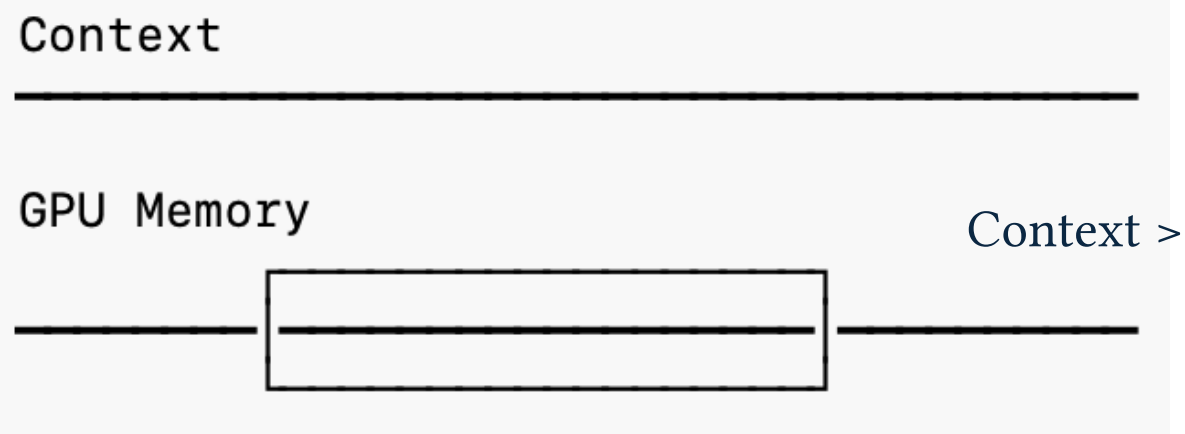
✗ Computation still runs on one GPU

✗ Context length is bounded by device memory

Faster attention does not remove the single-device memory limit.

Ring Attention

Scale exact attention across multiple GPUs



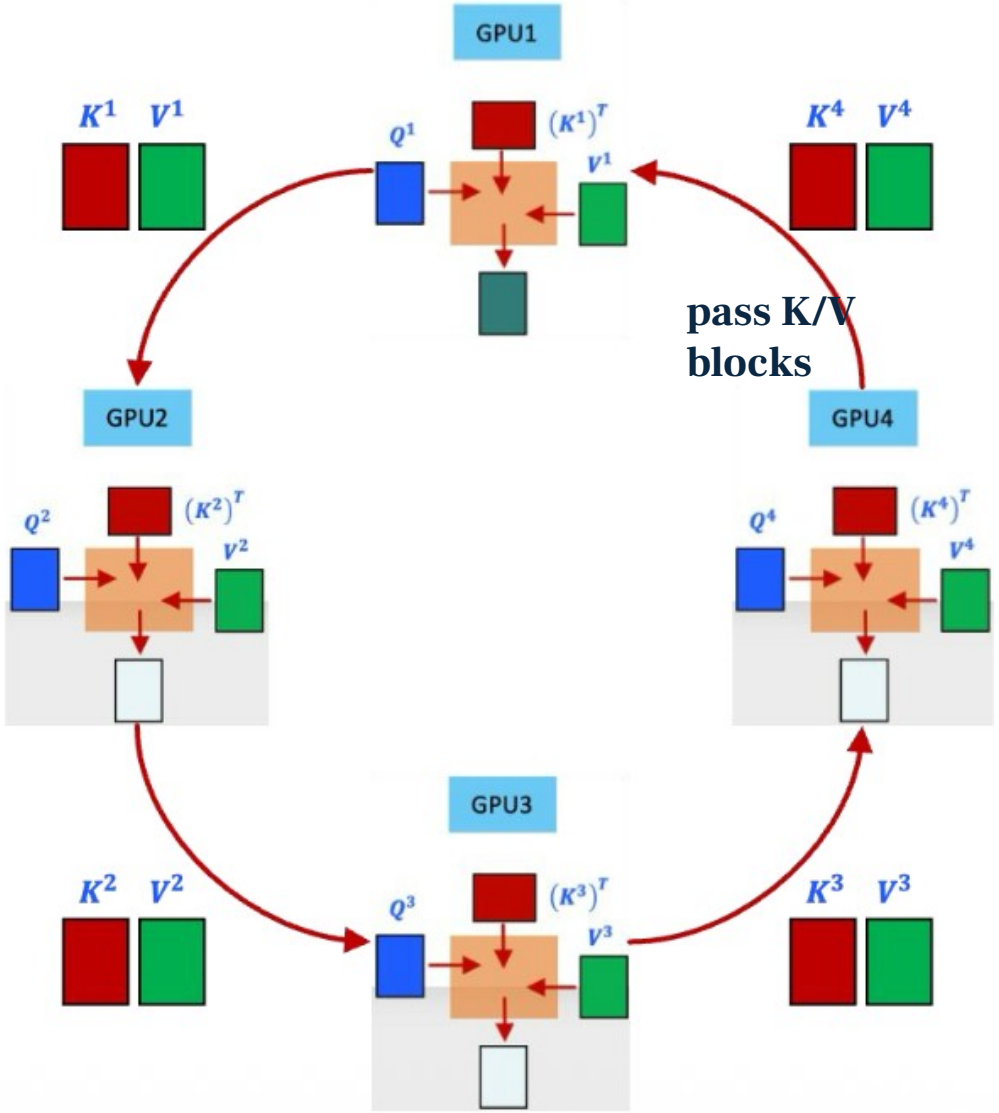
Context > GPU Memory

- **Split the context across multiple GPUs**
- **Exchange K/V blocks between GPUs**
- **Each GPU computes part of exact attention**

✔ **Exact attention without approximation**

Ring Attention

Scale exact attention across multiple GPUs



How it works

① Keep Query on each GPU

⇩

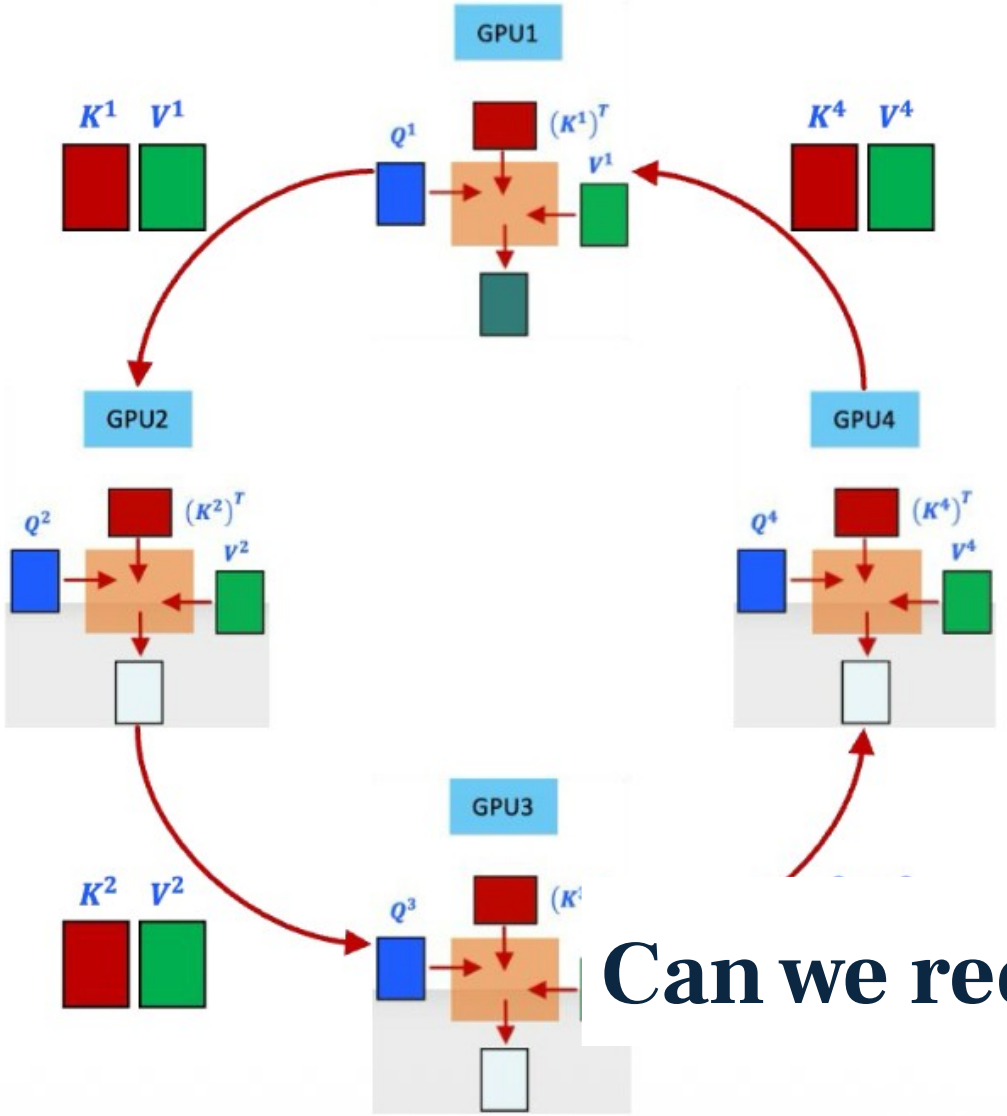
② Rotate K/V blocks

⇩

③ Repeat until all K/V blocks are processed

Ring Attention

Scale Exact Attention



✓ Exact Attention

↓

Still computes all QK^T interactions

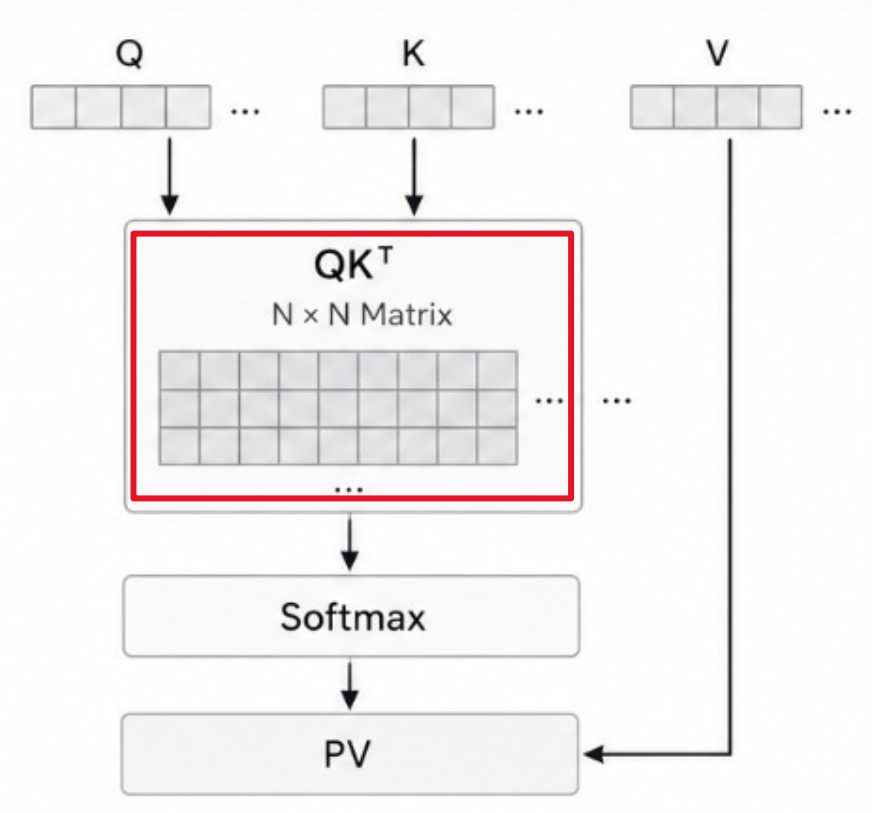
↓

$O(N^2)$ computation remains

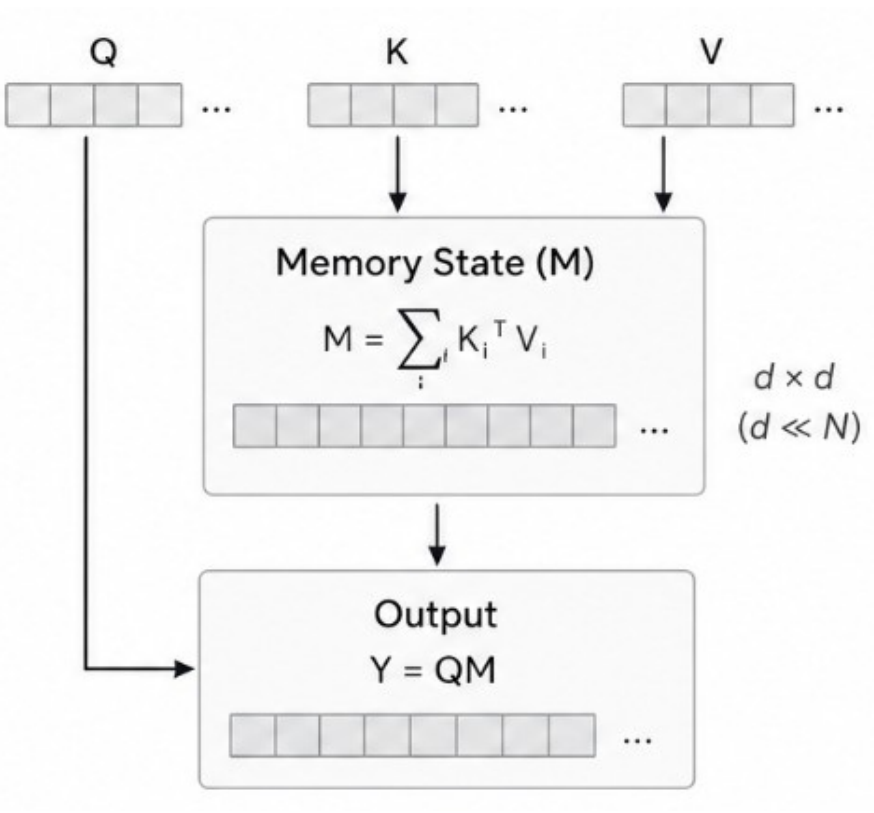
Can we reduce computation itself?

Reformulate Attention

Linear Attention : Reformulate the computation



Reformulate

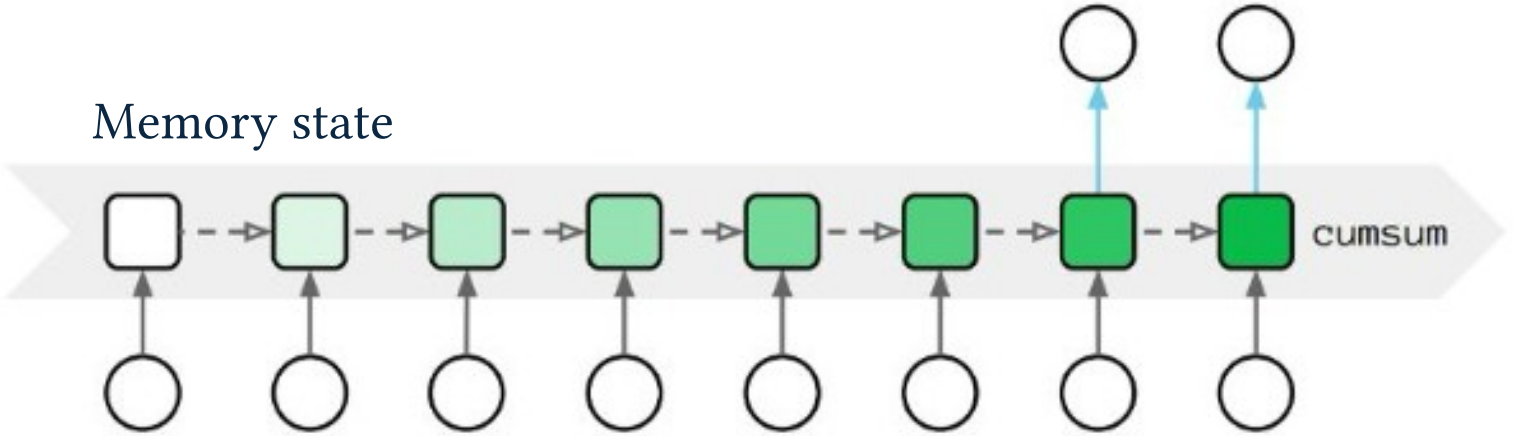


✓ Explicit QK^T Matrix
✓ N×N Attention Scores

✓ Memory State (M)
✓ No Explicit QK^T Matrix

Linear Attention

Attention with a fixed-size memory



① **Write :** Update the state

$M_t = M_{t-1} + K_t^T V_t$

② **Read :** Query reads memory

$y_t = Q_t M_t$

✔ **Constant Memory Size**

No explicit N x N attention matrix

Key Idea

Linear Attention replaces the attention matrix with a fixed-size memory state.

Comparison

Flash Attention & Ring Attention & Linear Attention

	Flash Attention	Ring Attention	Linear Attention
Idea	Optimize execution	Multi-GPU execution	Reformulate attention
Changes	Memory I/O	GPU parallelism	Attention computation
Exact Attention	✓	✓	× (Approximate)
Main Benefit	Faster on one GPU	Longer context	Linear complexity

Each method solves a different bottleneck

Thank You!