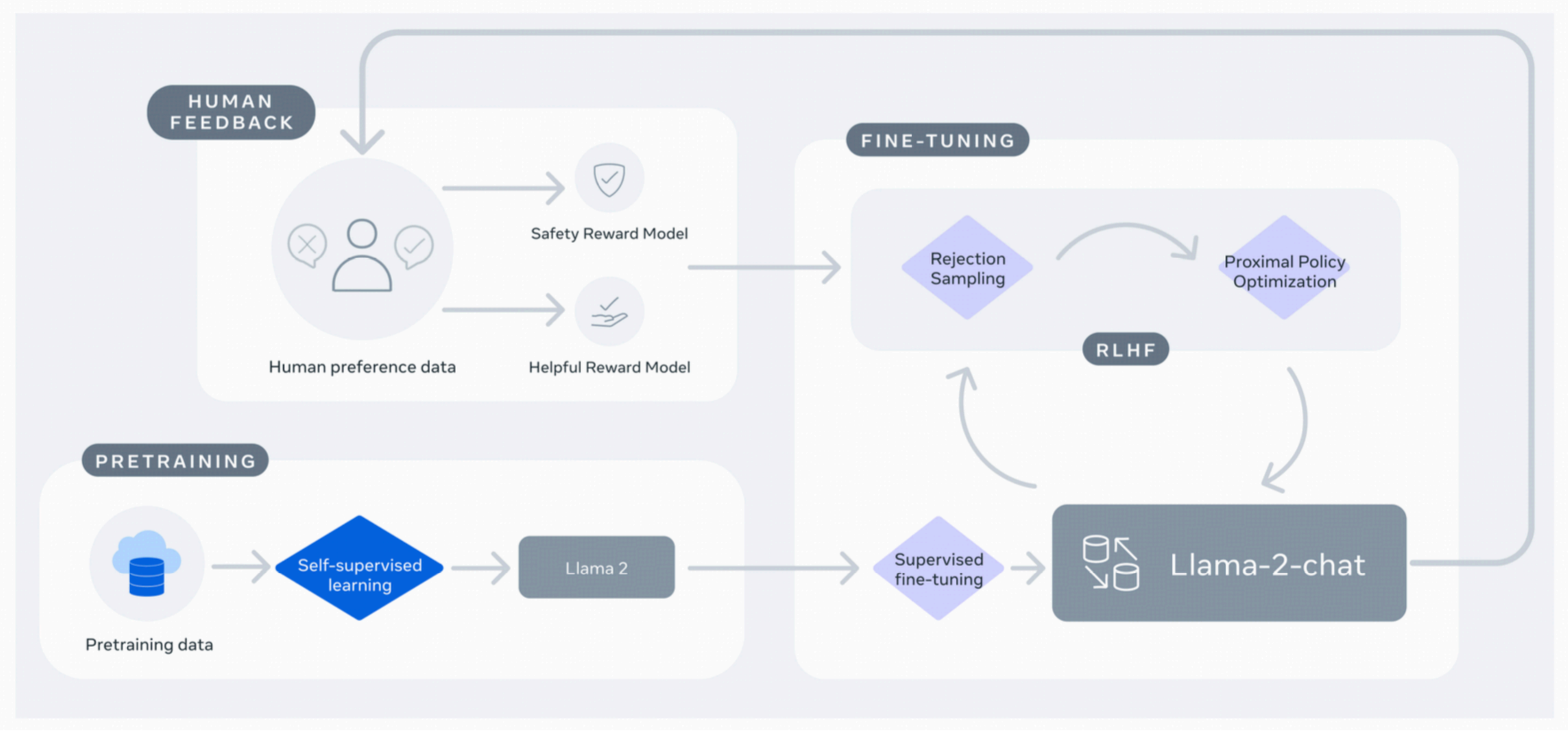
ECE7115 ~~Multimodal VLM~~ LLM

4. Modern LLM Architecture

Spring 2026

Namhyuk Ahn, Inha University

Last Week: LLM Training



Last Week: LLM Training

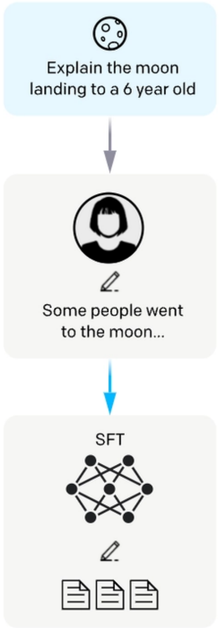
Step 1

Collect demonstration data, and train a supervised policy.

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.

This data is used to fine-tune GPT-3 with supervised learning.



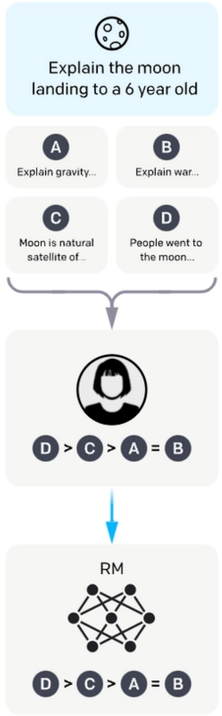
Step 2

Collect comparison data, and train a reward model.

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.



Step 3

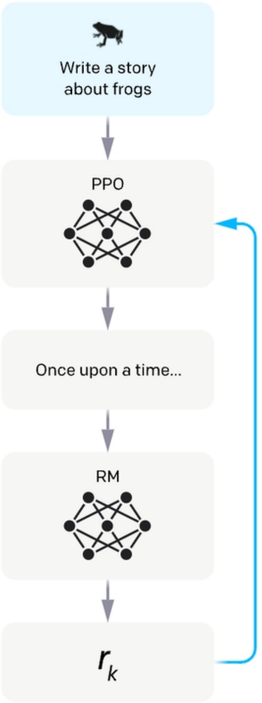
Optimize a policy against the reward model using reinforcement learning.

A new prompt is sampled from the dataset.

The policy generates an output.

The reward model calculates a reward for the output.

The reward is used to update the policy using PPO.



Last Week: Fine-Tuning

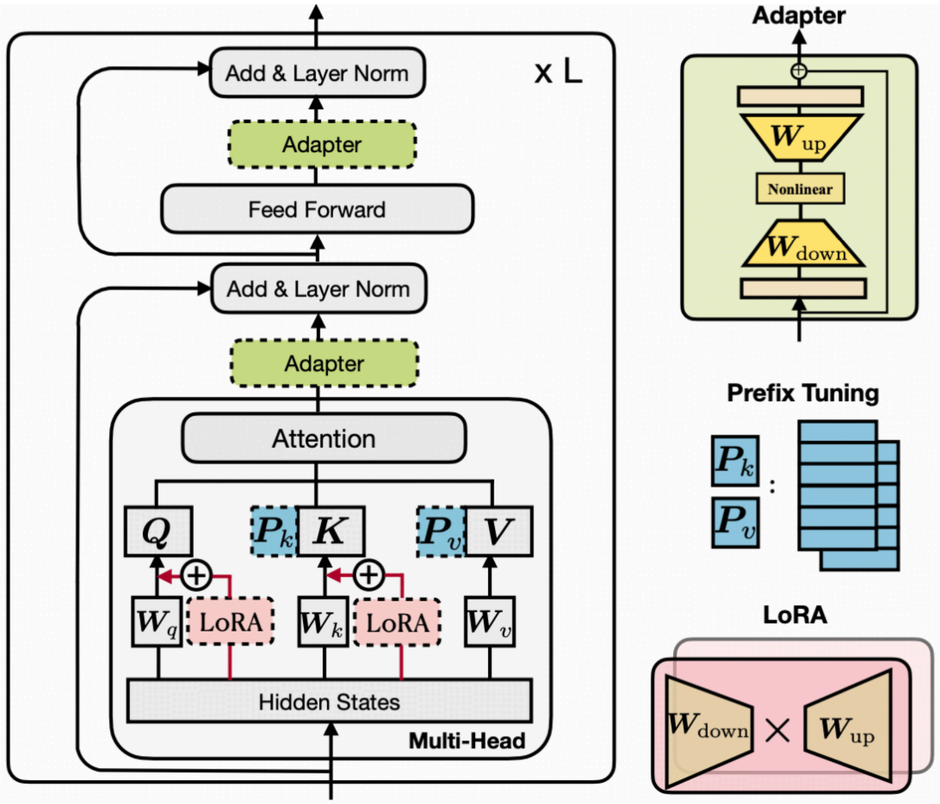


Figure 1: Illustration of the transformer architecture and several state-of-the-art parameter-efficient tuning methods. We use blocks with dashed borderlines to represent the added modules by those methods.

Figure 2: Performance of different methods on the XSum (Narayan et al., 2018) summarization task. The number of fine-tuned parameters is relative to the tuned parameters in full fine-tuning.

Last Week: Prompting

- In-context learning: all the magic starts

Instruction

```
Please classify movie reviews as 'positive' or ‘negative’.
```

Examples

```
Input: I really don’t like this movie.
Output: negative

Input: This movie is great!
Output: positive
```

- Chain-of-thought
  - Provided "chain-of-thought" example to let the model think itself

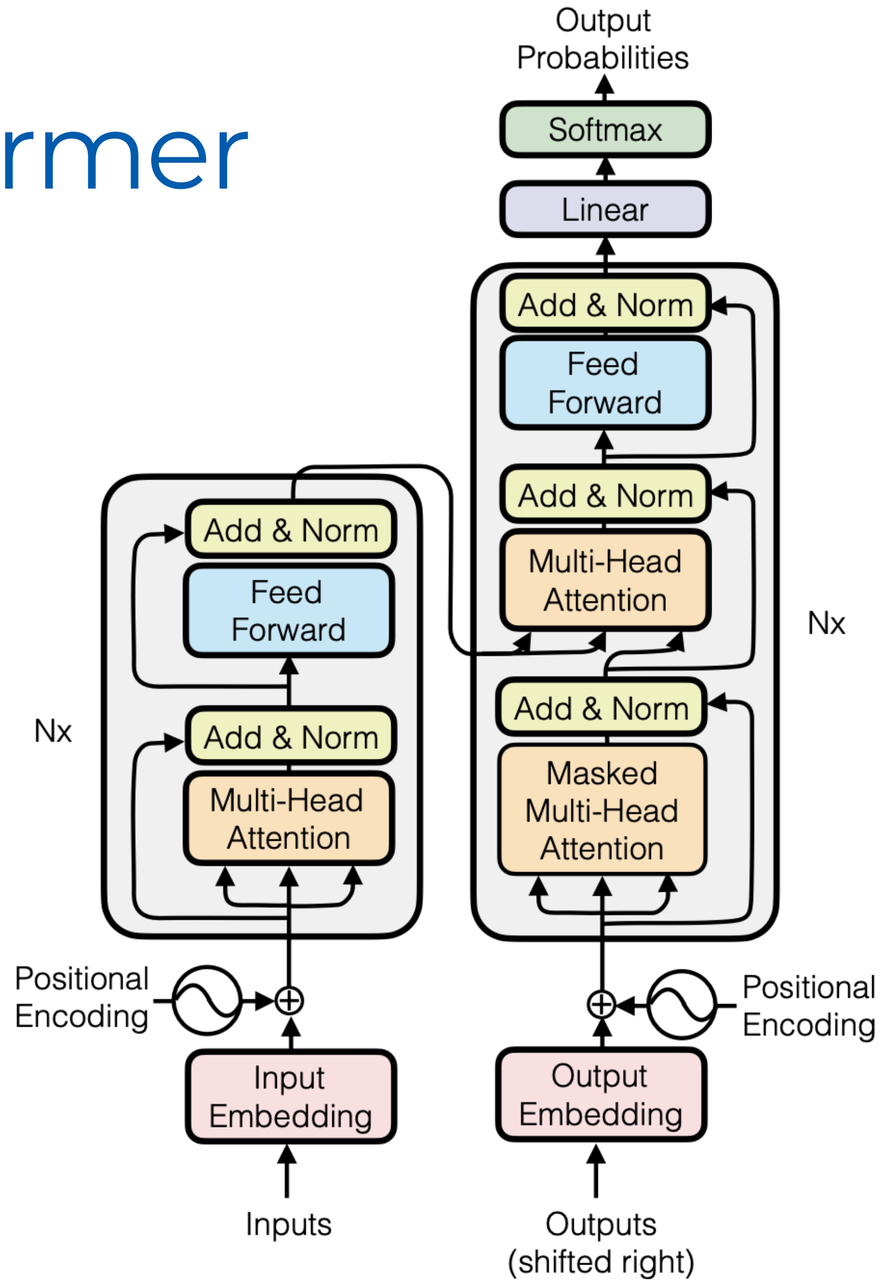
Starting Point: Vanilla Transformer

- Choices in the standard transformer
  - Sinusoidal positional encoding

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

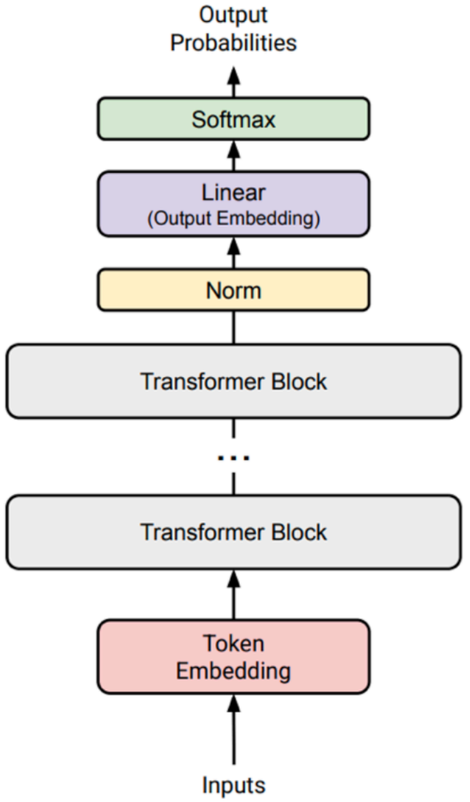
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

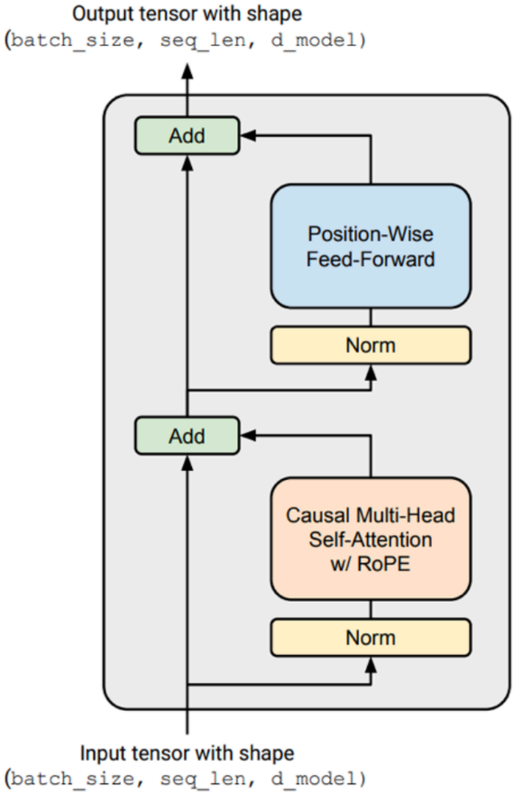
  - FFN with ReLU
  - LayerNorm
  - Residual connection (add ops)



TL;DR: Simple, Modern Variants

- Differences:
  - RMSNorm not LayerNorm
  - Norm is in front of the block
    - aka Pre-Norm
  - Rotary position embeddings (RoPE)
  - FFN uses SwiGLU, not ReLU
    - No bias





Modern LLM Architecture Variants

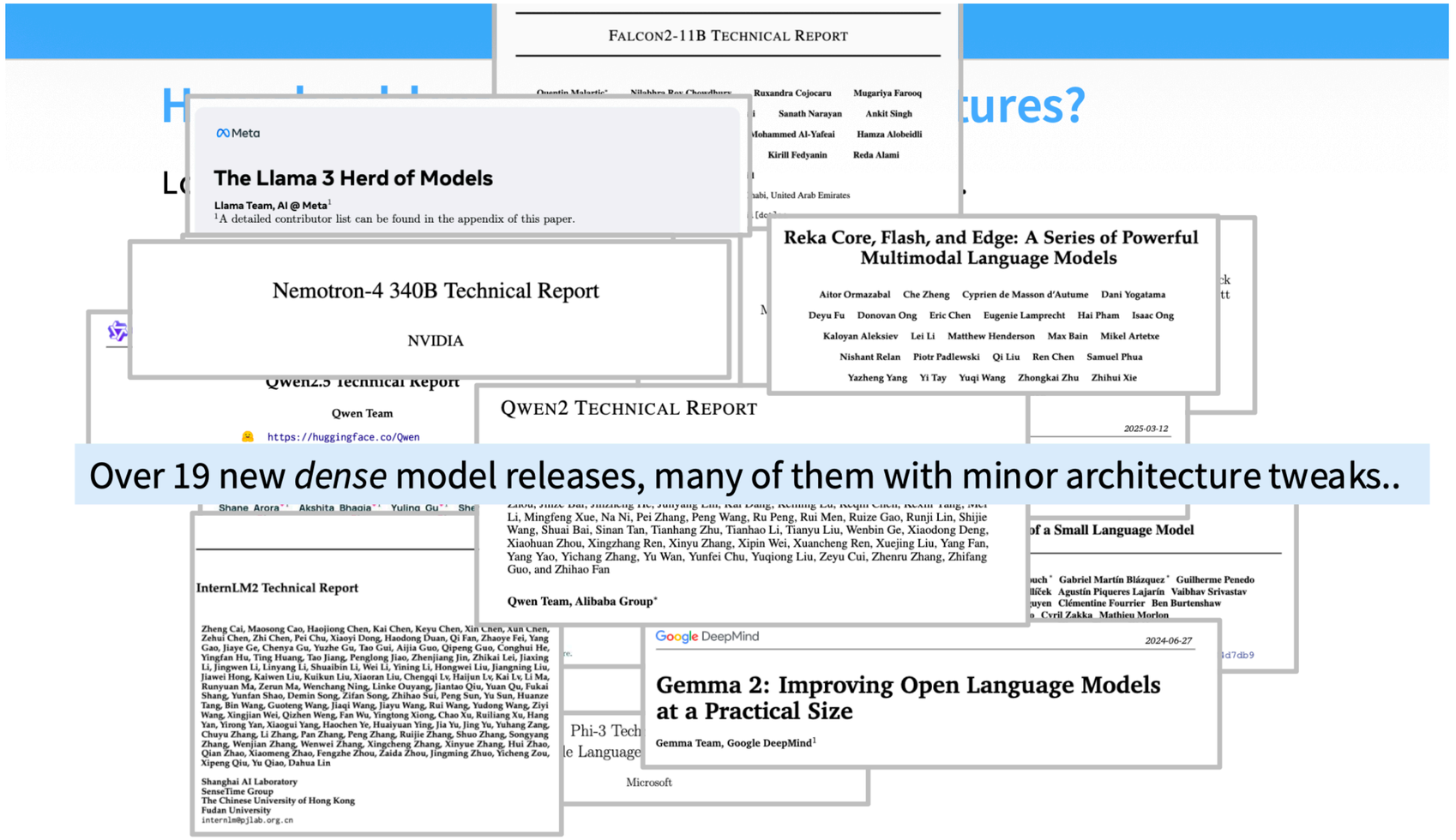


image credit: Stanford CS336

Modern LLM Architecture Variants

- In this lecture, we will talk through many major architecture and hyperparameter variants

Name	Year	Norm	Parallel Layer	Pre-norm	Position embedding	Activations	Stability tricks
Original transformer	2017	LayerNorm	Serial	☐	Sine	ReLU	
GPT	2018	LayerNorm	Serial	☐	Absolute	GeLU	
T5 (11B)	2019	RMSNorm	Serial	☑	Relative	ReLU	
GPT2	2019	LayerNorm	Serial	☑	Absolute	GeLU	
T5 (XXL 11B) v1.1	2020	RMSNorm	Serial	☑	Relative	GeGLU	
mT5	2020	RMSNorm	Serial	☑	Relative	GeGLU	
GPT3 (175B)	2020	LayerNorm	Serial	☑	Absolute	GeLU	
GPTJ	2021	LayerNorm	Parallel	☑	RoPE	GeLU	
LaMDA	2021			☑	Relative	GeGLU	
Anthropic LM (not claude)	2021			☑			
Gopher (280B)	2021	RMSNorm	Serial	☑	Relative	ReLU	
GPT-NeoX	2022	LayerNorm	Parallel	☑	RoPE	GeLU	
BLOOM (175B)	2022	LayerNorm	Serial	☑	AliBi	GeLU	
OPT (175B)	2022	LayerNorm	Serial	☑	Absolute	ReLU	
PaLM (540B)	2022	RMSNorm	Parallel	☑	RoPE	SwiGLU	Z-loss
Chinchilla	2022	RMSNorm	Serial	☑	Relative	ReLU	
Mistral (7B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU	
LLaMA2 (70B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU	
LLaMA (65B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU	
GPT4	2023			☐			
Olmo 2	2024	RMSNorm	Serial	☐	RoPE	SwiGLU	Z-loss QK-norm
Gemma 2 (27B)	2024	RMSNorm	Serial	☑	RoPE	GeGLU	Logit soft capping Pre+post norm
Nemotron-4 (340B)	2024	LayerNorm	Serial	☑	RoPE	SqRelu	
Qwen 2 (72b) - same for 2.5	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Falcon 2 11B	2024	LayerNorm	Parallel	☑	RoPE	GeLU	Z-loss
Phi3 (small) - same for phi4	2024	RMSNorm	Serial	☑	RoPE	GeGLU	
Llama 3 (70B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Reka Flash	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Command R+	2024	LayerNorm	Parallel	☑	RoPE	SwiGLU	
OLMo	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Qwen (14B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
DeepSeek (67B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Yi (34B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Mixtral of Experts	2024			☐			
Command A	2025	LayerNorm	Parallel	☑	Hybrid (RoPE+NoPE)	SwiGLU	
Gemma 3	2025	RMSNorm	Serial	☐	RoPE	GeGLU	Pre+post norm QK-norm
SmolLM2 (1.7B)	2025	RMSNorm	Serial	☑	RoPE	SwiGLU	

Lecture Overview

- Modern LLM Architecture Beyond Vanilla Transformer
  - Activations, FFN
  - Positional Embedding (Encoding)
  - Hyperparameters
  - Stability Tricks

- Attention Variants
  - Multi-Head Attention
  - Multi-Query Attention
  - Group-Query Attention
  - Multi-Head Latent Attention

Modern LLM Architecture

image credit: Stanford CS336

Architecture Variations

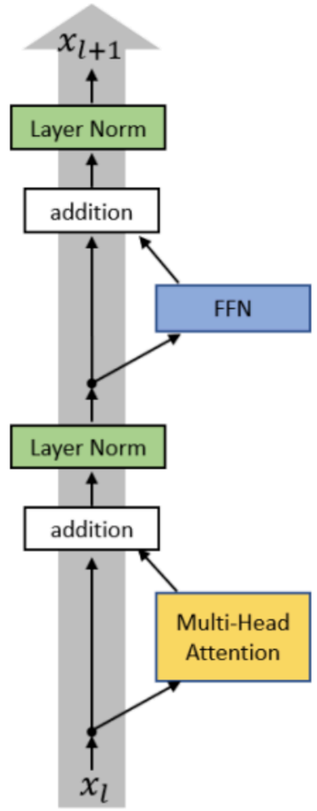
- Low consensus before LLaMA (2023)
- Trends toward LLaMA-style architecture

Name	Year	Norm	Parallel Layer	Pre-norm	Position embedding	Activations
Original transformer	2017	LayerNorm	Serial	☐	Sine	ReLU
GPT	2018	LayerNorm	Serial	☐	Absolute	GeLU
T5 (11B)	2019	RMSNorm	Serial	☑	Relative	ReLU
GPT2	2019	LayerNorm	Serial	☑	Sine	GeLU
T5 (XXL 11B) v1.1	2020	RMSNorm	Serial	☑	Relative	GeGLU
mT5	2020	RMSNorm	Serial	☑	Relative	GeGLU
GPT3 (175B)	2020	LayerNorm	Serial	☑	Sine	GeLU
GPTJ	2021	LayerNorm	Parallel	☑	RoPE	GeLU
LaMDA	2021			☑	Relative	GeGLU
Gopher (280B)	2021	RMSNorm	Serial	☑	Relative	ReLU
GPT-NeoX	2022	LayerNorm	Parallel	☑	RoPE	GeLU
BLOOM (175B)	2022	LayerNorm	Serial	☑	AliBi	GeLU
OPT (175B)	2022	LayerNorm	Serial	☑	Absolute	ReLU
PaLM (540B)	2022	RMSNorm	Parallel	☑	RoPE	SwiGLU
Chinchilla	2022	RMSNorm	Serial	☑	Relative	ReLU
Mistral (7B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU
LLaMA2 (70B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU
LLaMA (65B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU
Qwen (14B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
DeepSeek (67B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
Yi (34B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU

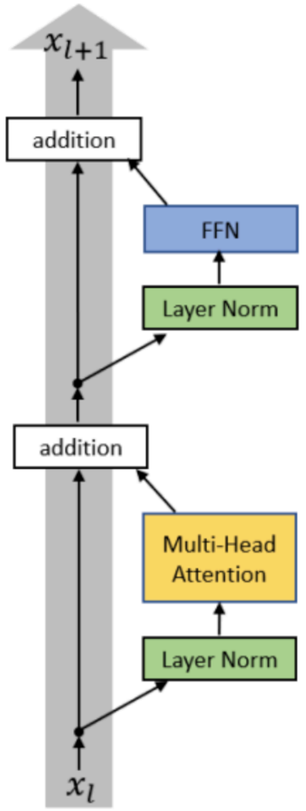
Pre-Norm vs. Post-Norm [Xiong+2020]

- Set up LayerNorm so that it doesn't affect the main residual signal path
- Almost all modern LLMs use pre-norm

Post-LN Transformer	Pre-LN Transformer
$x_{l,i}^{post,1} = \text{MultiHeadAtt}(x_{l,i}^{post}, [x_{l,1}^{post}, \cdots, x_{l,n}^{post}])$	$x_{l,i}^{pre,1} = \text{LayerNorm}(x_{l,i}^{pre})$
$x_{l,i}^{post,2} = x_{l,i}^{post} + x_{l,i}^{post,1}$	$x_{l,i}^{pre,2} = \text{MultiHeadAtt}(x_{l,i}^{pre,1}, [x_{l,1}^{pre,1}, \cdots, x_{l,n}^{pre,1}])$
$x_{l,i}^{post,3} = \text{LayerNorm}(x_{l,i}^{post,2})$	$x_{l,i}^{pre,3} = x_{l,i}^{pre} + x_{l,i}^{pre,2}$
$x_{l,i}^{post,4} = \text{ReLU}(x_{l,i}^{post,3} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$	$x_{l,i}^{pre,4} = \text{LayerNorm}(x_{l,i}^{pre,3})$
$x_{l,i}^{post,5} = x_{l,i}^{post,3} + x_{l,i}^{post,4}$	$x_{l,i}^{pre,5} = \text{ReLU}(x_{l,i}^{pre,4} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$
$x_{l+1,i}^{post} = \text{LayerNorm}(x_{l,i}^{post,5})$	$x_{l+1,i}^{pre} = x_{l,i}^{pre,5} + x_{l,i}^{pre,3}$
	Final LayerNorm: $x_{Final,i}^{pre} \leftarrow \text{LayerNorm}(x_{L+1,i}^{pre})$

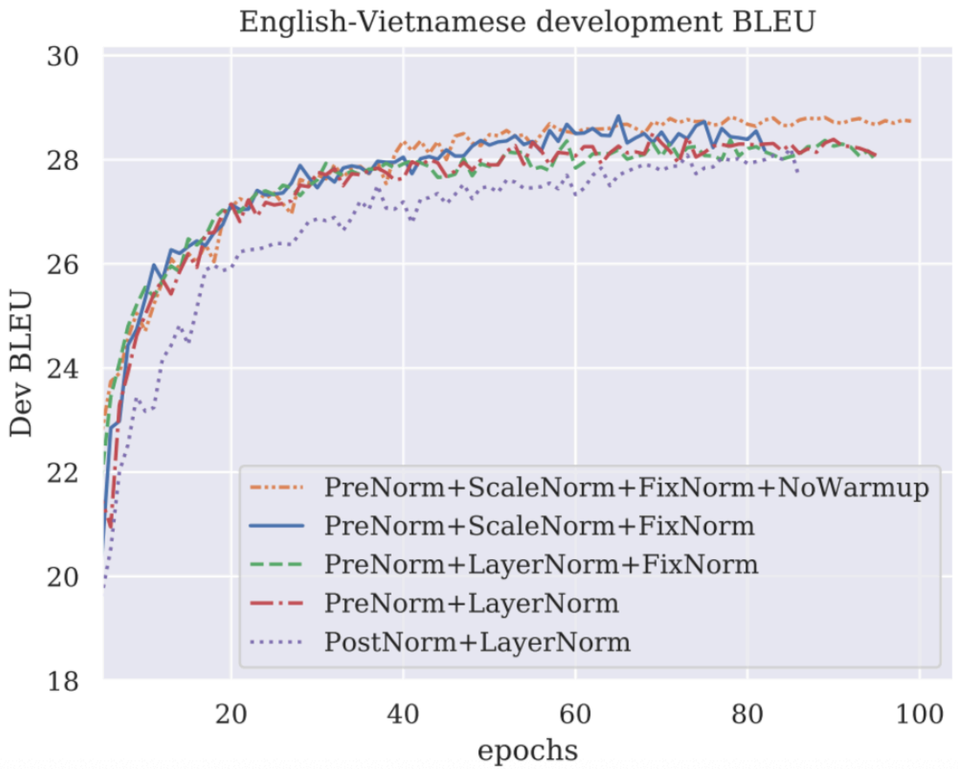


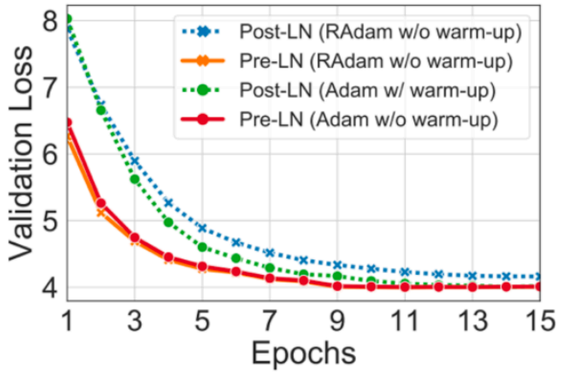
post-LayerNorm



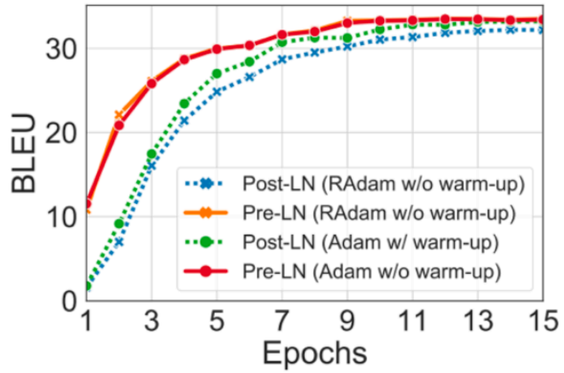
pre-LayerNorm

Pre-Norm vs. Post-Norm [Salazar+ 2019, Xiong+2020]

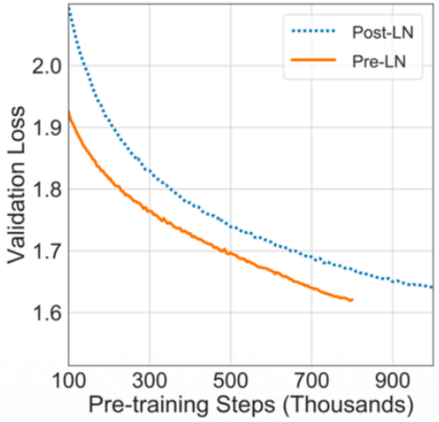




(a) Validation Loss (IWSLT)



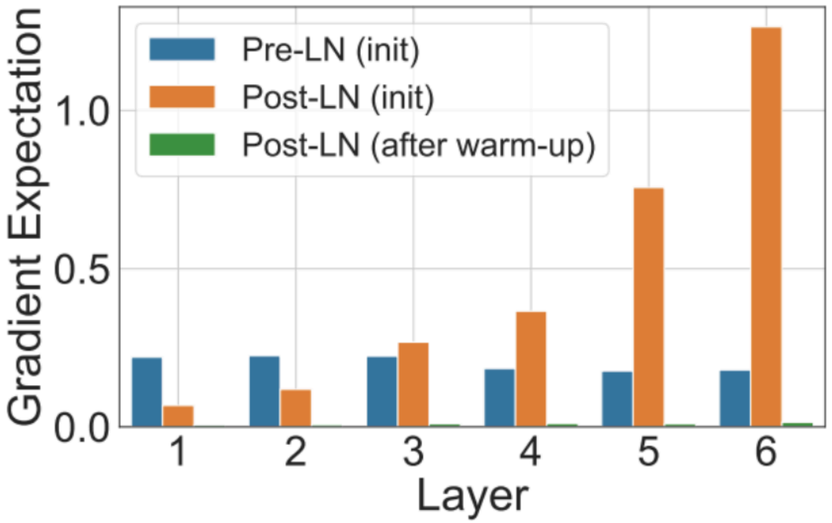
(b) BLEU (IWSLT)



(a) Validation Loss on BERT

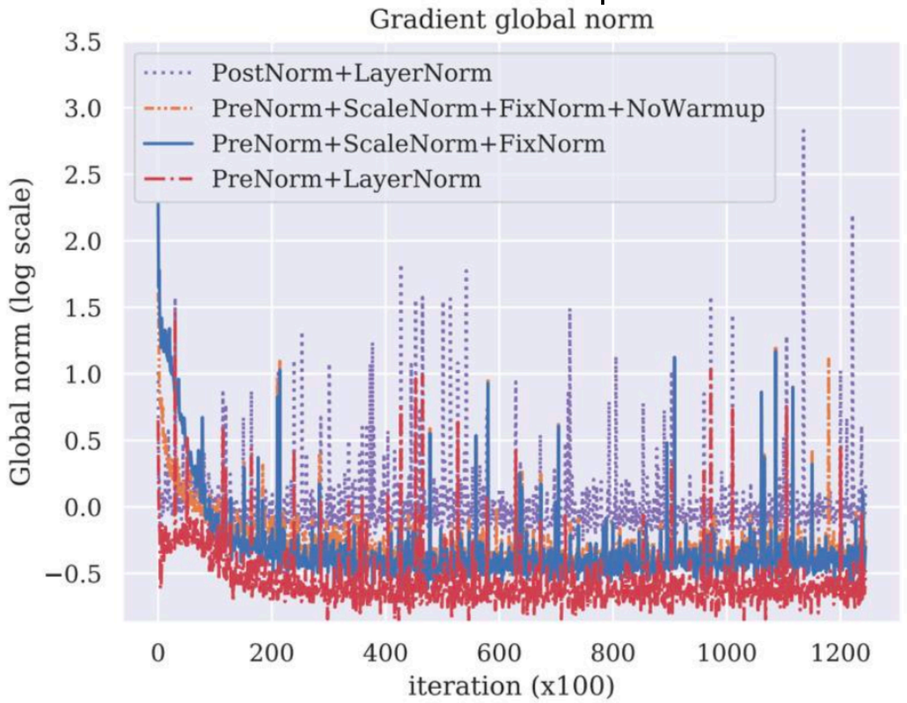
Pre-Norm vs. Post-Norm [Salazar+ 2019, Xiong+2020]

Gradient attenuation



(a) W^1 in the FFN sub-layers

Gradient spikes



- Pre-Norm increases training stability and enables larger LRs for large networks

LayerNorm vs. RMSNorm [Zhang+ 2019]

- Original Transformer: LayerNorm
  - Normalizes the mean/variance across d_{model}
  - Usage: GPT-1/2/3, OPT, GPT-J, etc

gain

bias

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

vector stddev

vector mean

- Many modern LLMs: RMSNorm
  - Does not subtract mean or add a bias term
  - Usage: LLaMA-family, Chinchilla, T5, etc

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\text{RMS}(\mathbf{x})} \cdot \mathbf{g}$$

Why RMSNorm?

- Modern explanation: it's faster (and just as good).
  - Fewer operations (no mean calculation)
  - Fewer parameters (no bias term to store)

$$\mathrm{RMS}(\mathbf{x}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

$$\mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\mathrm{RMS}(\mathbf{x})} \cdot \mathbf{g}$$

- Does this explanation make sense?
  - Matmuls are the vast majority of FLOPs (and memory)

Operator class	% flop
△ Tensor contraction	99.80
□ Stat. normalization	0.17
○ Element-wise	0.03

[Ivanov+ 2021]

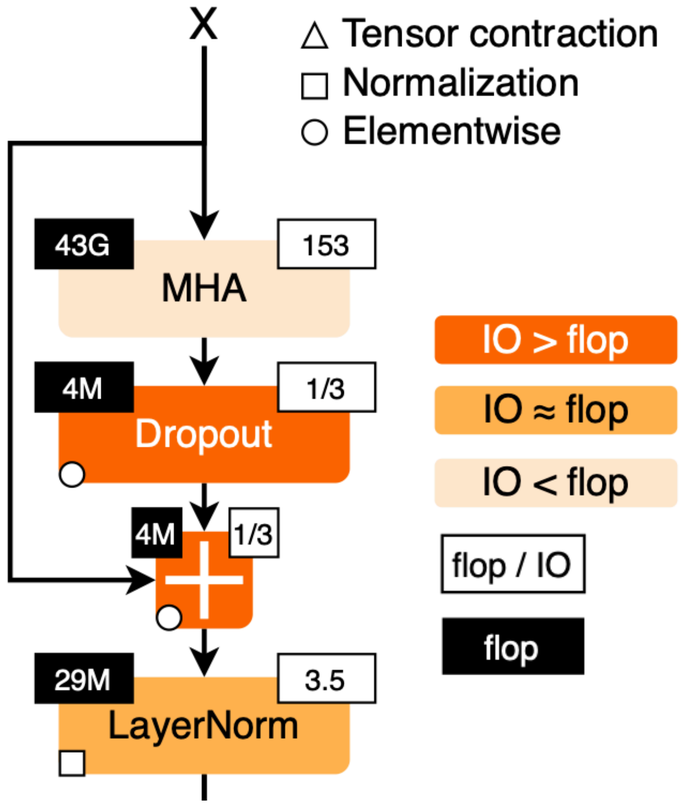
Why RMSNorm?

- Modern explanation: it's faster (and just as good).
  - Fewer operations (no mean calculation)
  - Fewer parameters (no bias term to store)

- Important lesson: FLOPs are not runtime!
  (we will discuss later)

Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

[Ivanov+ 2021]



Why RMSNorm?

- Runtime (and surprisingly, performance) gains have been seen in papers

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	$223M$	$11.1T$	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02	26.62
RMS Norm	$223M$	$11.1T$	3.68	2.167 ± 0.008	**1.821**	**75.45**	**17.94**	**24.07**	**27.14**
Rezero	$223M$	$11.1T$	3.51	2.262 ± 0.003	1.939	61.69	15.64	20.90	26.37
Rezero + LayerNorm	$223M$	$11.1T$	3.26	2.223 ± 0.006	1.858	70.42	17.58	23.02	26.29
Rezero + RMS Norm	$223M$	$11.1T$	3.34	2.221 ± 0.009	1.875	70.33	17.32	23.02	26.19
Fixup	$223M$	$11.1T$	2.95	2.382 ± 0.012	2.067	58.56	14.42	23.02	26.31

[Narang+ 2020]

More Generally: No Bias Terms

- Most modern transformers don't have bias terms

- Original Transformer $\mathbf{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$

- Most modern implementations (if they don't use SwiGLU)

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

  - Why? memory (similar to RMSNorm) and optimization stability

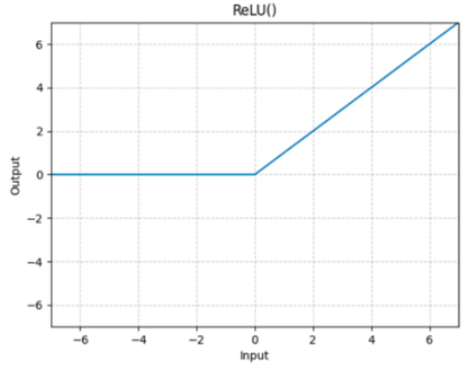
LayerNorm: Summary

- Basically everyone does pre-norm
  - It keeps the good parts of residual connections
  - Nicer gradient propagation, fewer spike

- Most people do RMSNorm
  - In practice, works as well as LayerNorm
  - But, has fewer params to move, which saves on wallclock time
  - Also drops bias since the compute/param tradeoffs are not good

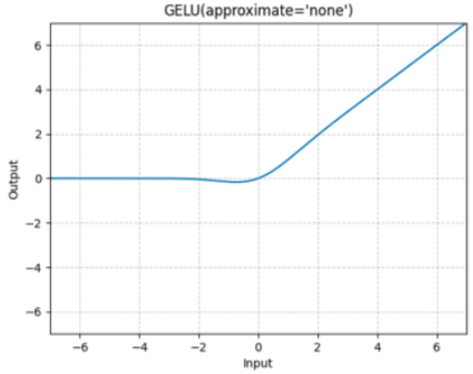
Activations

- ReLU $FF(x) = \max(0, xW_1)\, W_2$
  - Notable models: Transformer, T5, Chinchilla



$$FF(x) = \mathrm{GELU}(xW_1)W_2$$

- GeLU $GELU(x) \coloneqq x\Phi(x)$
  - Notable models: GPT-1/2/3



- SwiGLU / GeGLU
  - Notable models: LLaMA, most recent models
  - We will discuss these mostly

Gated Activations (*GLU)

- GLUs modify the 'first part' of a FF layer

$$FF(x) = \max(0, xW_1)\, W_2$$

- Instead of a linear+ReLU, augment the above with an (entrywise) linear

$$\max(0, xW_1) \rightarrow \max(0, xW_1) \otimes (xV)$$

- This gives the gated variant (ReGLU), note we have an extra param V

$$\mathrm{FF}_{\mathrm{ReGLU}}(x) = (\max(0, xW_1) \otimes xV)\, W_2$$

Gated Variants of Standard FF Layers

- GeGLU $\mathrm{FFN}_{\mathrm{GEGLU}}(x, W, V, W_2) = (\mathrm{GELU}(xW) \otimes xV)W_2$
  - Notable models: Phi3, Gemma

- SwiGLU $\mathrm{FFN}_{\mathrm{SwiGLU}}(x, W, V, W_2) = (\mathrm{Swish}_1(xW) \otimes xV)W_2$
  - Swish is x * sigmoid(x)
  - Notable models: most recent models

- Note: Gated models use smaller dimensions (d_{ff}) by 2/3
  - To match # params to the ReLU-based FFN

Do Gated Linear Units Work? [Shazeer 2020]

- Yes, fairly consistently so

	Score Average	CoLA MCC	SST-2 Acc
FFN_{ReLU}	83.80	51.32	94.04
FFN_{GELU}	83.86	53.48	94.04
$\text{FFN}_{\text{Swish}}$	83.60	49.79	93.69
FFN_{GLU}	84.20	49.16	94.27
$\text{FFN}_{\text{GEGLU}}$	84.12	53.65	93.92
$\text{FFN}_{\text{Bilinear}}$	83.79	51.02	**94.38**
$\text{FFN}_{\text{SwiGLU}}$	84.36	51.59	93.92
$\text{FFN}_{\text{ReGLU}}$	**84.67**	**56.16**	**94.38**
[Raffel et al., 2019]	83.28	53.84	92.68
ibid. stddev.	0.235	1.111	0.569

- In conclusion, it said

> We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

Do Gated Linear Units Work? [Narang+ 2021]

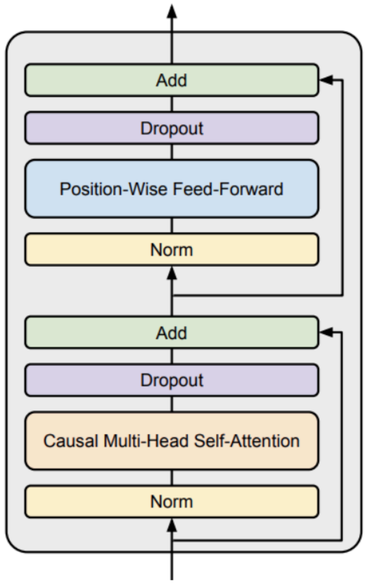
- Yes, fairly consistently so

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ
Vanilla Transformer	$223M$	$11.1T$	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02
GeLU	$223M$	$11.1T$	3.58	2.179 ± 0.003	1.838	**75.79**	**17.86**	**25.13**
Swish	$223M$	$11.1T$	3.62	2.186 ± 0.003	1.847	**73.77**	17.74	**24.34**
ELU	$223M$	$11.1T$	3.56	2.270 ± 0.007	1.932	67.83	16.73	23.02
GLU	$223M$	$11.1T$	3.59	2.174 ± 0.003	**1.814**	**74.20**	**17.42**	24.34
GeGLU	$223M$	$11.1T$	3.55	2.130 ± 0.006	**1.792**	**75.96**	**18.27**	**24.87**
ReGLU	$223M$	$11.1T$	3.57	2.145 ± 0.004	**1.803**	**76.17**	**18.36**	**24.87**
SeLU	$223M$	$11.1T$	3.55	2.315 ± 0.004	1.948	68.76	16.76	22.75
SwiGLU	$223M$	$11.1T$	3.53	2.127 ± 0.003	**1.789**	**76.00**	**18.20**	**24.34**
LiGLU	$223M$	$11.1T$	3.59	2.149 ± 0.005	**1.798**	**75.34**	**17.97**	**24.34**
Sigmoid	$223M$	$11.1T$	3.63	2.291 ± 0.019	1.867	**74.31**	17.51	23.02
Softplus	$223M$	$11.1T$	3.47	2.207 ± 0.011	1.850	**72.45**	17.65	**24.34**

Gating and Activations: Summary

- Many variations (ReLU, GeLU, *GLU) across models
- *GLU isn’t necessary for a good model, but it’s probably helpful
- But evidence points towards consistent gains from Swi/GeGLU

Serial vs. Parallel Layers



- Normal transformer blocks are serial
  - In other words, they compute attention, then the MLP

- Can we parallelize the transformer block?
  - A few models do parallel layers (GPT-J, PaLM, Cohere models)

Parallel Layers – We use a "parallel" formulation in each Transformer block (Wang & Komatsuzaki, 2021), rather than the standard "serialized" formulation. Specifically, the standard formulation can be written as:

$$y = x + \text{MLP}(\text{LayerNorm}(x + \text{Attention}(\text{LayerNorm}(x))))$$

Whereas the parallel formulation can be written as:

$$y = x + \text{MLP}(\text{LayerNorm}(x)) + \text{Attention}(\text{LayerNorm}(x))$$

The parallel formulation results in roughly 15% faster training speed at large scales, since the MLP and Attention input matrix multiplications can be fused. Ablation experiments showed a small quality degradation at 8B scale but no quality degradation at 62B scale, so we extrapolated that the effect of parallel layers should be quality neutral at the 540B scale.

image credit: Stanford CS336

Recap

- Pre-vs-post norm
  - Everyone does pre-norm
- Layer vs RMSnorm
  - RMSnorm has clear compute wins, sometimes even performance
- Gating
  - GLUs seem generally better, though differences are small
- Serial vs parallel layers
  - No extremely serious ablations

Name	Year	Norm	Parallel Layer	Pre-norm	Position embedding	Activations
Original transformer	2017	LayerNorm	Serial	☐	Sine	ReLU
GPT	2018	LayerNorm	Serial	☐	Absolute	GeLU
T5 (11B)	2019	RMSNorm	Serial	☑	Relative	ReLU
GPT2	2019	LayerNorm	Serial	☑	Absolute	GeLU
T5 (XXL 11B) v1.1	2020	RMSNorm	Serial	☑	Relative	GeGLU
mT5	2020	RMSNorm	Serial	☑	Relative	GeGLU
GPT3 (175B)	2020	LayerNorm	Serial	☑	Absolute	GeLU
GPTJ	2021	LayerNorm	Parallel	☑	RoPE	GeLU
LaMDA	2021			☑	Relative	GeGLU
Anthropic LM (not claude)	2021			☑		
Gopher (280B)	2021	RMSNorm	Serial	☑	Relative	ReLU
GPT-NeoX	2022	LayerNorm	Parallel	☑	RoPE	GeLU
BLOOM (175B)	2022	LayerNorm	Serial	☑	AliBi	GeLU
OPT (175B)	2022	LayerNorm	Serial	☑	Absolute	ReLU
PaLM (540B)	2022	RMSNorm	Parallel	☑	RoPE	SwiGLU
Chinchilla	2022	RMSNorm	Serial	☑	Relative	ReLU
Mistral (7B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU
LLaMA2 (70B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU
LLaMA (65B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU
GPT4	2023			☐		
Baichuan 2	2023	RMSNorm	Serial	☑	AliBi	SwiGLU
Olmo 2	2024	RMSNorm	Serial	☐	RoPE	SwiGLU
Gemma 2 (27B)	2024	RMSNorm	Serial	☑	RoPE	GeGLU
Nemotron-4 (340B)	2024	LayerNorm	Serial	☑	RoPE	SqRelu
Qwen 2 (72b) - same for 2.5	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
Falcon 2 11B	2024	LayerNorm	Parallel	☑	RoPE	GeLU
Phi3 (small) - same for phi4	2024	RMSNorm	Serial	☑	RoPE	GeGLU
Llama 3 (70B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
Reka Flash	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
Command R+	2024	LayerNorm	Parallel	☑	RoPE	SwiGLU
OLMo	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
Qwen (14B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
DeepSeek (67B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
Yi (34B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
Mixtral of Experts	2024			☐		
Command A	2025	LayerNorm	Parallel	☑	Hybrid (RoPE+NoPE)	SwiGLU
Gemma 3	2025	RMSNorm	Serial	☐	RoPE	GeGLU
SmolLM2 (1.7B)	2025	RMSNorm	Serial	☑	RoPE	SwiGLU

Positional Embedding

Positional Embeddings Variants

- Sine Embedding: add sines/cosines
  - Notable models: Transformer

$$Embed(x, i) = v_x + PE_{pos}$$

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

- Absolute Embedding: add a position vector to the word embedding
  - $Embed(x, i) = v_x + u_i$ u_i is a looked-up vector by the token's position
  - Notable models: GPT-1/2/3

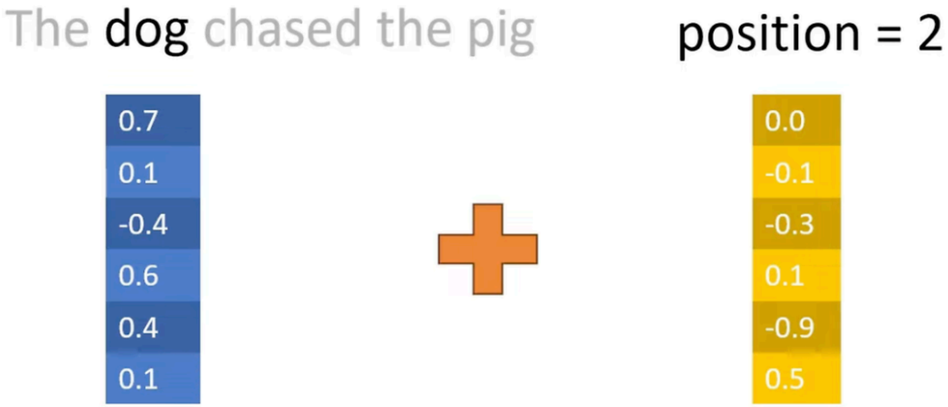
- Relative Embedding: add a vector to the attention computation
  - No PE addition to the input
  - Notable models: T5, Gopher

$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}$$

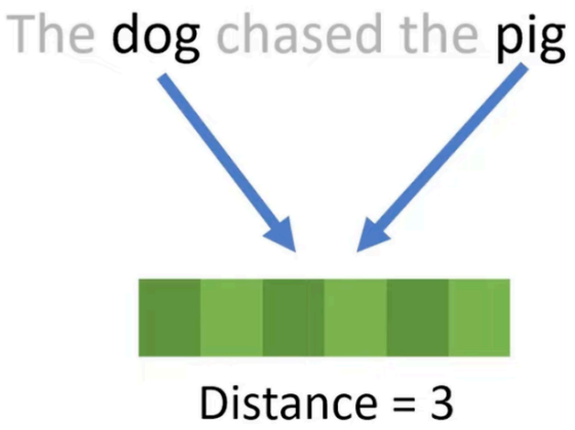
relative distance between two tokens

Absolute vs. Relative PE

- Absolute PEs add an encoding to the input in hope that relative position will be captured
- Relative PEs explicitly encode relative position



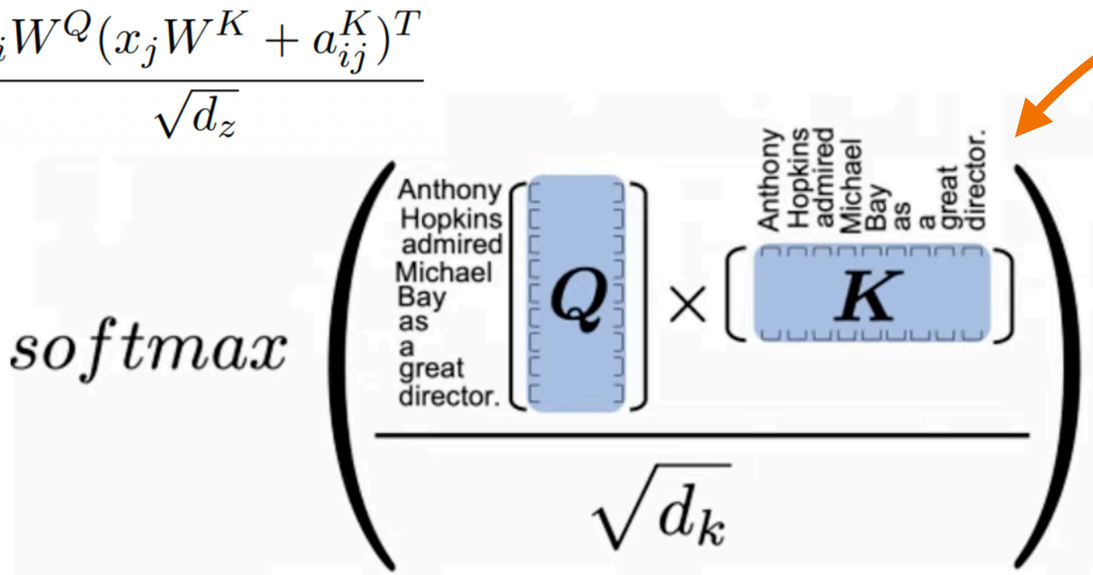
Absolute PE

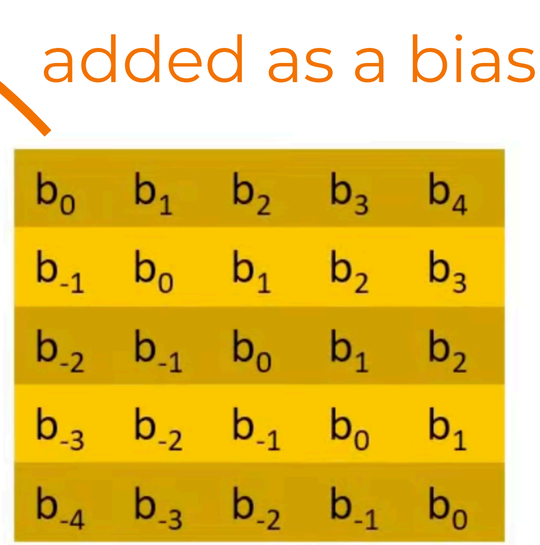


Relative PE

Relative Embedding [Shaw+ 2018, Huang+ 2018, Raffel+ 2019]

$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}$$

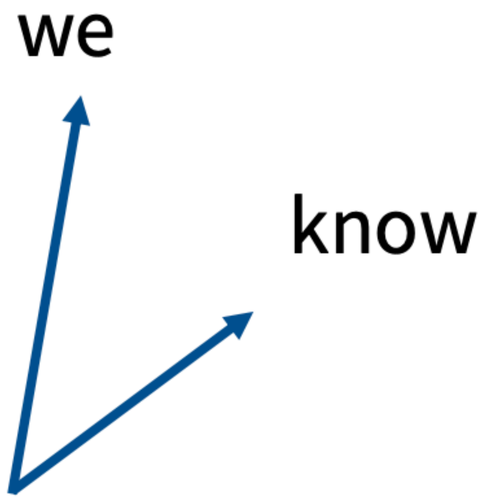




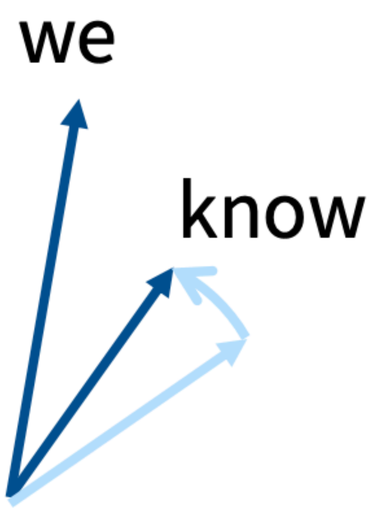
- Good at capturing relative distance, but inefficient at inference
  - Extra step in self-attention layer
  - Difficult to do KV caching (we will cover this later)

Rotary Positional Embeddings (RoPE) [Su+ 2021]

- Intuitive idea: rotate embeddings by their position



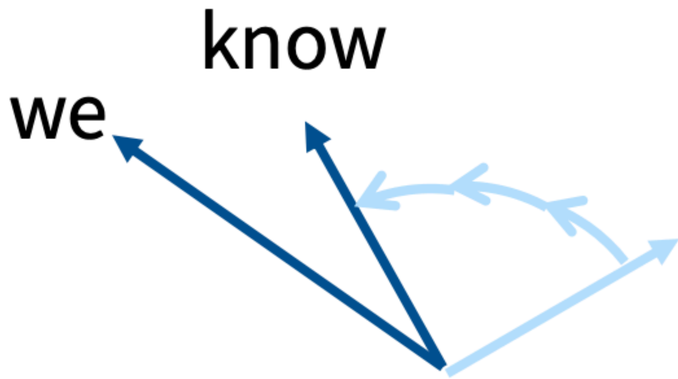
Position independent word embedding



Embedding
"we know that"

Rotate "we" by '0 positions'

Rotate "know" by '1 positions'

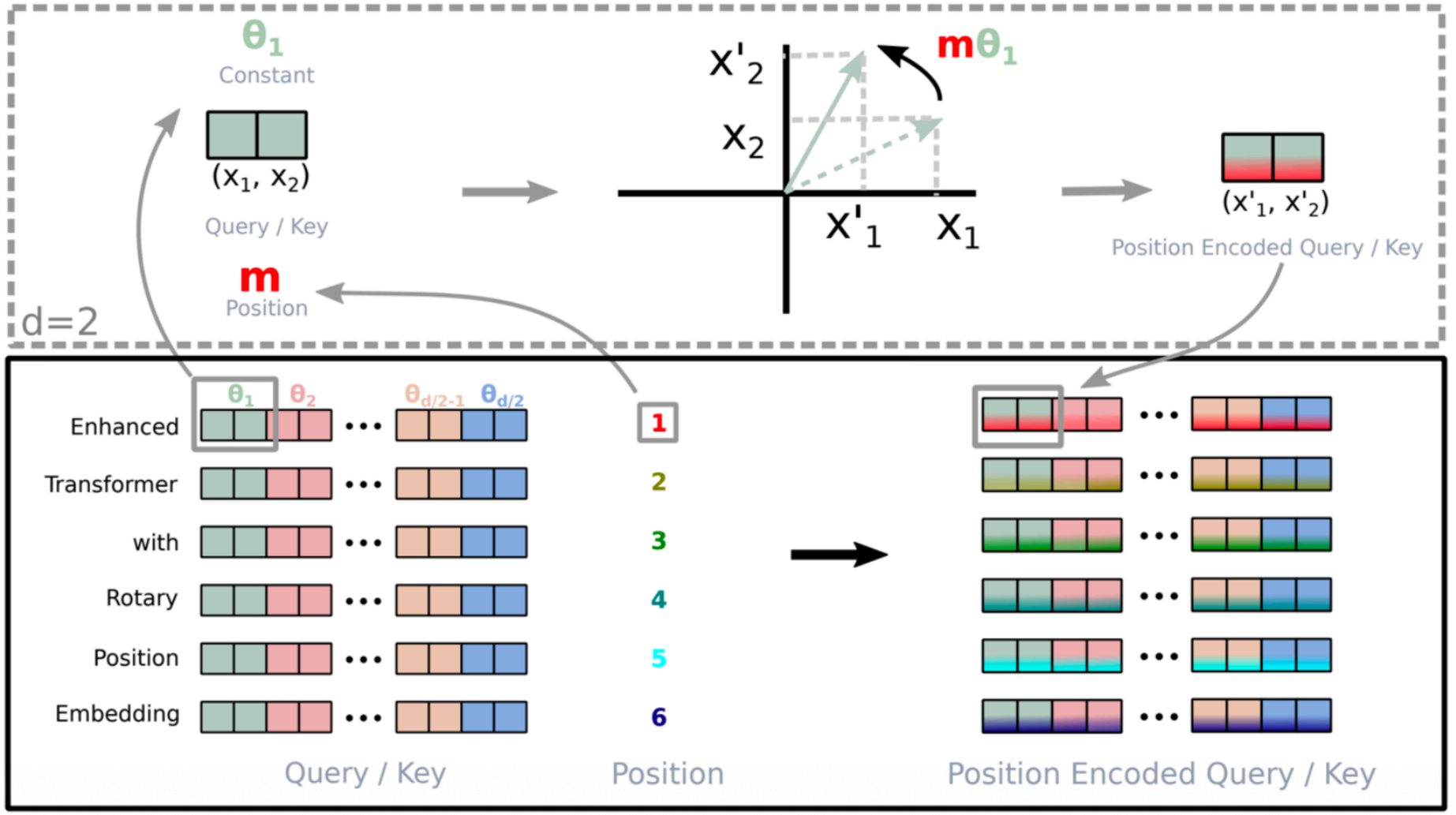


Embedding
"of course we know"

Rotate "we" by '2 positions'

Rotate "know" by '3 positions'

RoPE


θ1
Constant
(x1, x2)
Query / Key
m
Position
d=2
x'2
x2
x'1
x1
mθ1
(x'1, x'2)
Position Encoded Query / Key
θ1
θ2
θd/2-1
θd/2
Enhanced
Transformer
with
Rotary
Position
Embedding
1
2
3
4
5
6
Query / Key
Position
Position Encoded Query / Key

RoPE: Math

- For the 2d vector

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} W^{(11)}_{\{q,k\}} & W^{(12)}_{\{q,k\}} \\ W^{(21)}_{\{q,k\}} & W^{(22)}_{\{q,k\}} \end{pmatrix} \begin{pmatrix} x^{(1)}_m \\ x^{(2)}_m \end{pmatrix}$$

2D rotation matrix Q, K (not V)

RoPE: Math

- In general, Nd vector case

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}^d_{\Theta,m}\boldsymbol{W}_{\{q,k\}}\boldsymbol{x}_m$$

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

RoPE: Implementation

Usual attention stuff

```
query_states = self.q_proj(hidden_states)
key_states = self.k_proj(hidden_states)
value_states = self.v_proj(hidden_states)

# Flash attention requires the input to have the shape
# batch_size x seq_length x head_dim x hidden_dim
# therefore we just need to keep the original shape
query_states = query_states.view(bsz, q_len, self.num_heads, self.head_dim).transpose(1, 2)
key_states = key_states.view(bsz, q_len, self.num_key_value_heads, self.head_dim).transpose(1, 2)
value_states = value_states.view(bsz, q_len, self.num_key_value_heads, self.head_dim).transpose(1, 2)
```

Get the RoPE matrix cos/sin

Multiply query/key inputs

```
cos, sin = self.rotary_emb(value_states, position_ids)
query_states, key_states = apply_rotary_pos_emb(query_states, key_states, cos, sin)
```

...

Same stuff as the usual multi-head self attention below

Hyperparameters

Hyperparameters

- In Transformer, for example,
  - How much bigger should the FFN size be compared to hidden size?
  - How many heads, and should # heads always divide hidden size?
  - What should my vocab size be?

- These are also hyperparameters
  - Learning rate scheduling
  - Do people even regularize these huge LMs?
  - How do people scale these models: very deep or very wide?

1. FFN Size

- Model dimension ratio in FFN $\mathbf{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$

- There are two hyperparameters that conventionally called
  - The feedforward dim (d_{ff}) and model dim (d_{model})
  - i.e. `W1 = Linear(d_model, d_ff); W2 = Linear(d_ff, d_model)`

  - So, what should their relationship be?

$$d_{ff} = 4 \times d_{model}$$

    - This is almost always true with just a few exceptions

Exception #1: *GLU

- Remember that GLU variants scale down d_{ff} by 2/3
  - This means most GLU variants have $d_{ff} = 8/3 \times d_{model}$
  - Some notable such examples

Model	d_{ff}/d_{model}
PaLM	4
Mistral 7B	3.5
LLaMA-2 70B	3.5
LLaMA 70B	2.68
Qwen 14B	2.67
DeepSeek 67B	2.68
Yi 34B	2.85
T5 v1.1	2.5

Exception #2: T5

- As we have (and will) see, most LMs have boring hyperparameters

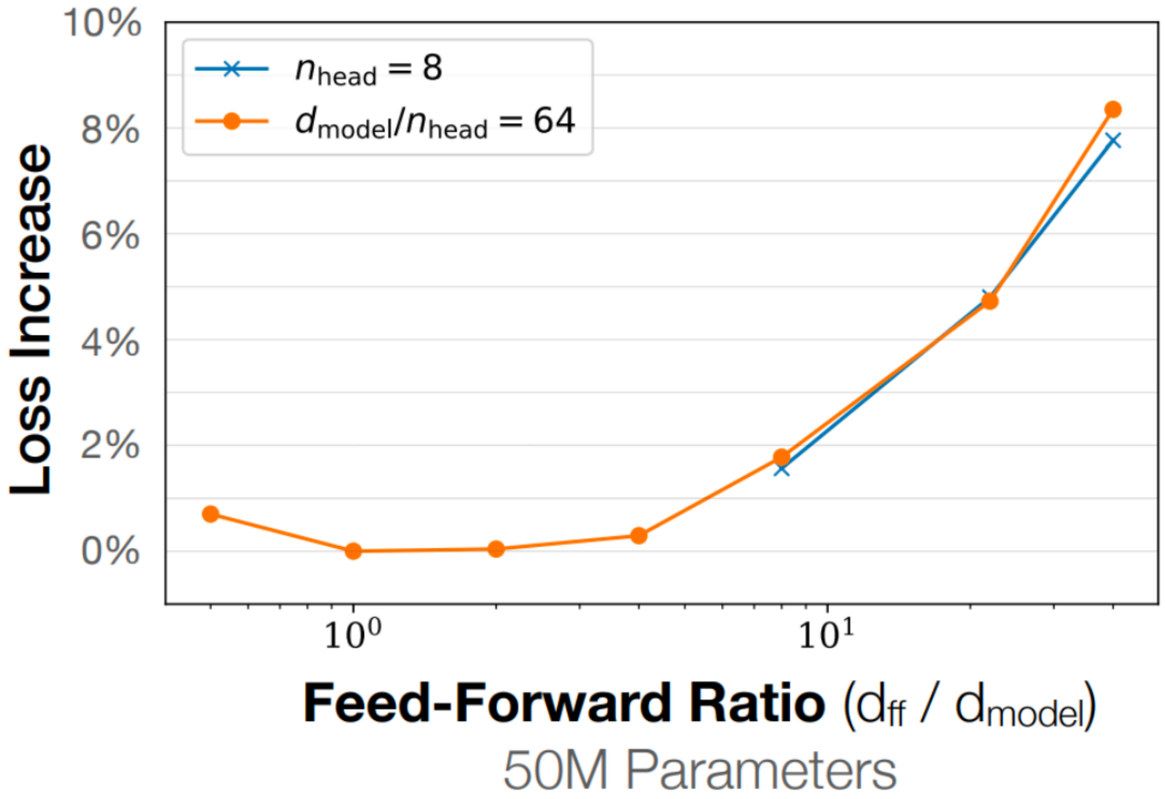
- One exception is T5 which has some very bold settings
  - In particular, for the 11B model, they set
  - `d_ff = 65,536, d_model = 1024; d_ff / d_model = 64`

> for "11B" we use $d_{\text{ff}} = 65{,}536$ with 128-headed attention producing a model with about 11 billion parameters. We chose to scale up d_{ff} specifically because modern accelerators (such as the TPUs we train our models on) are most efficient for large dense matrix multiplications like those in the Transformer's feed-forward networks.

- Other exceptions: Gemma 2 (8x), SmolLM/Gemma 3 (4x, GLU)

Evidence for 4 or 8/3?

- Empirically, between 1-10 are near-optimal



[Kaplan+ 2020]

2. # Heads in Multi-Head Attention

- Normally, `d_model = num_heads * d_head`, but some exceptions exist

	Num heads	Head dim	Model dim	Ratio
GPT3	96	128	12288	1
T5	128	128	1024	16
T5 v1.1	64	64	4096	1
LaMDA	128	128	8192	2
PaLM	48	258	18432	1.48
LLaMA2	64	128	8192	1

3. Aspect Ratio

- Should my model be deep or wide? How deep and how wide?
- Most models are surprisingly consistent on this one too!

Sweet Spot?

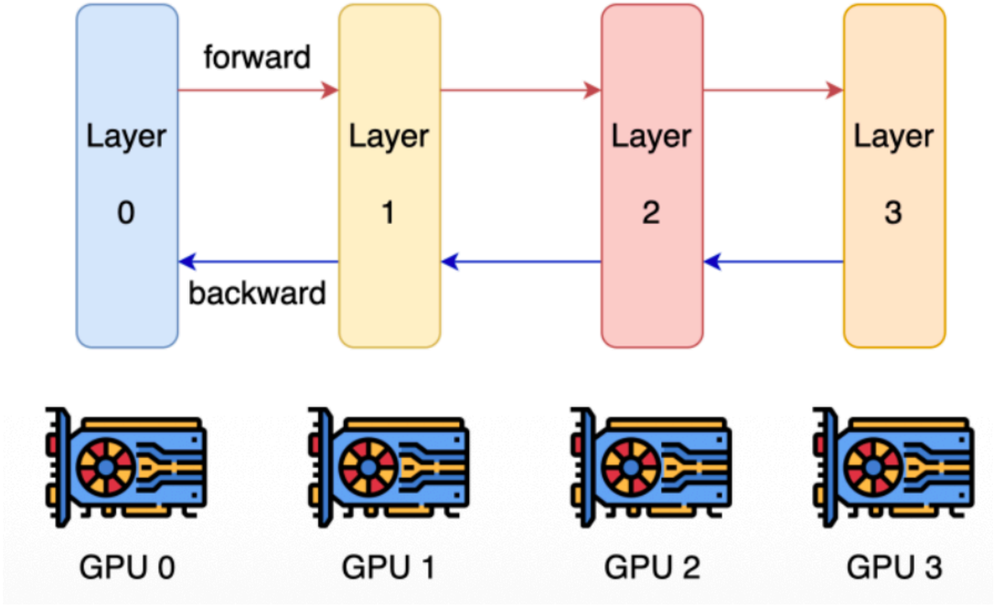
Model	d_{model}/n_{layer}
BLOOM	205
T5 v1.1	171
PaLM (540B)	156
GPT3/OPT/Mistral/Qwen	128
LLaMA / LLaMA2 / Chinchila	102
T5 (11B)	43
GPT2	33

Considerations About Aspect Ratio

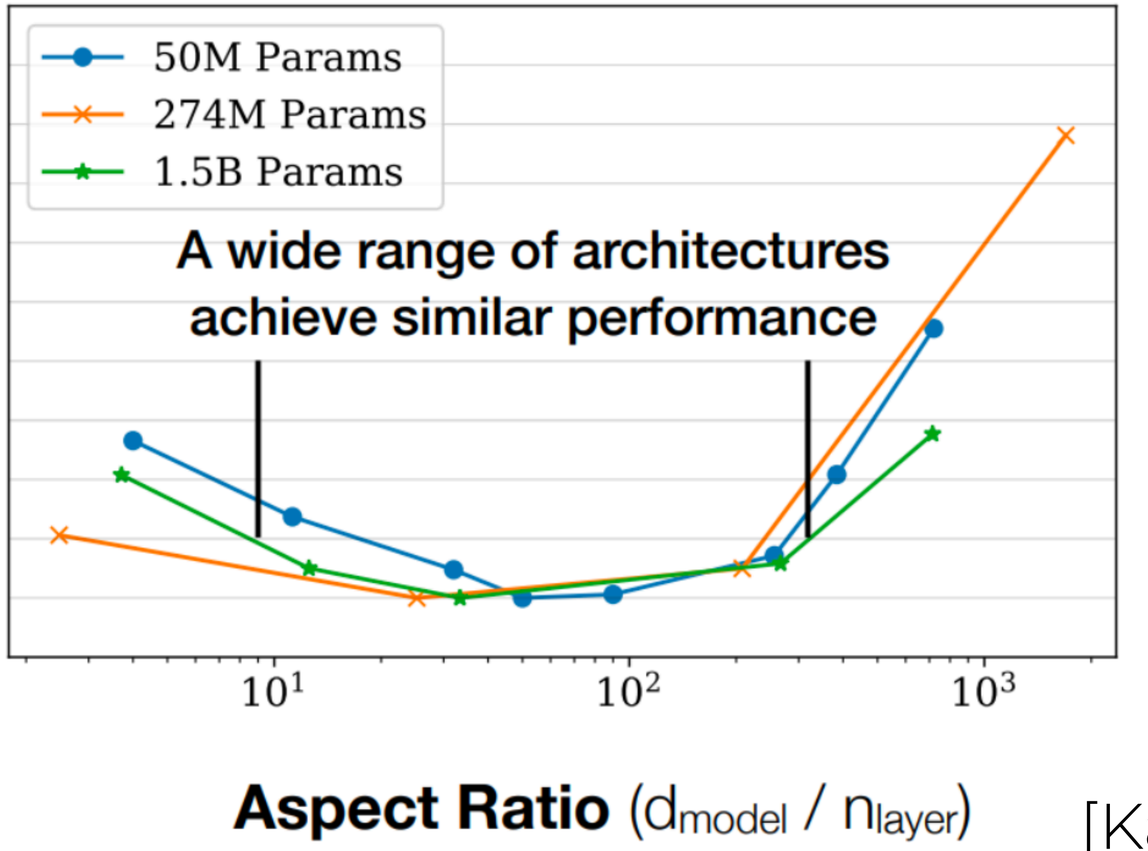
- Extremely deep models are harder to parallelize and have higher latency

> **The Limits of Depth vs Width** We note an obvious limitation with our advice. Scaling depth has an obvious limiter, i.e., they are non-parallelizable across different machines or devices and every computation has to always wait for the previous layer. This is unlike width, which can be easily parallelizable over thousands or hundreds of thousands of devices. Within the limitation of scaling

[Tay+ 2021]



Evidence on Aspect Ratio



[Kaplan+ 2020]

4. Vocabulary Size

Monolingual: 30-50k vocab

Model	Token count
Original transformer	37000
GPT	40257
GPT2/3	50257
T5/T5v1.1	32128
LLaMA	32000

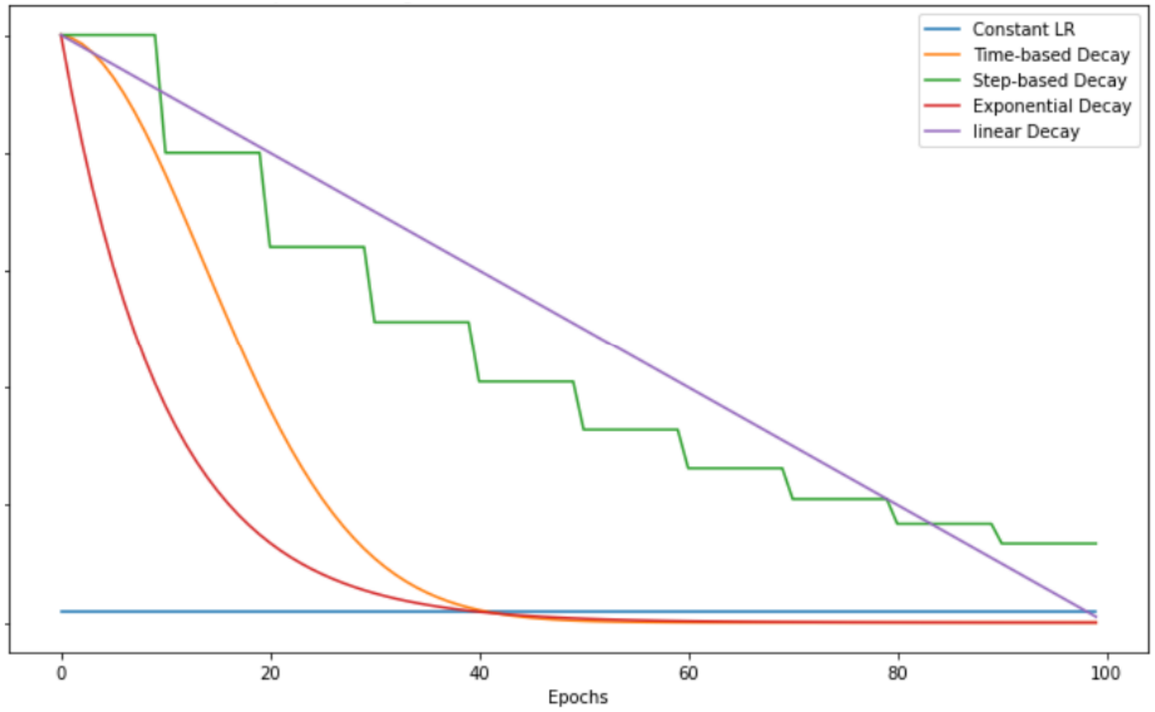
Multilingual / production systems: 100-250k

Model	Token count
mT5	250000
PaLM	256000
GPT4	100276
Command A	255000
DeepSeek	100000
Qwen 15B	152064
Yi	64000

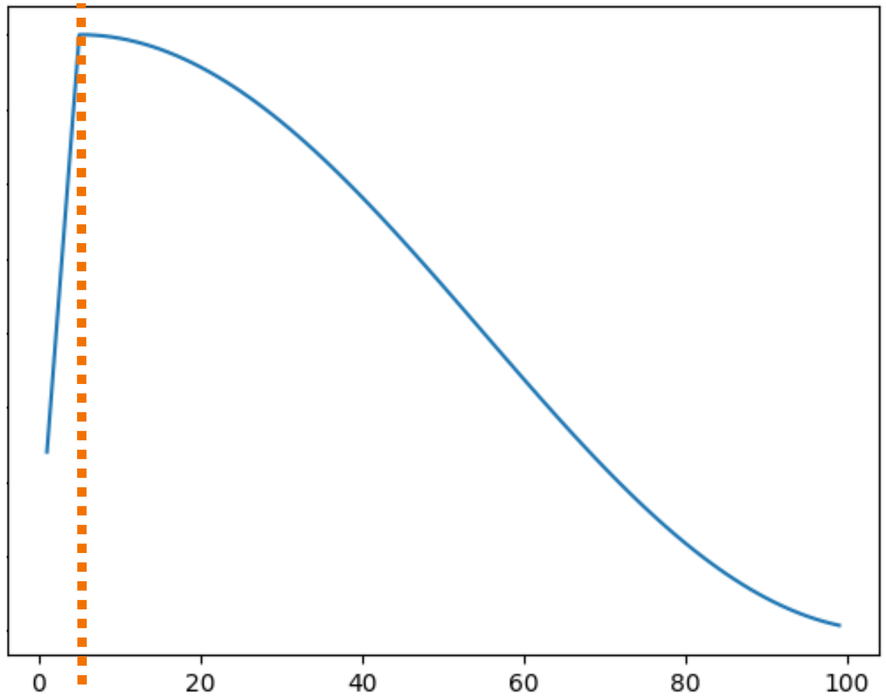
5. Learning Rate Scheduling

- Normally, cosine schedule with linear warmup is used
  - We will see modern scheduling (WSD) later

Scheduler comparison



Cosine schedule + linear warmup



6. Regularization

- Question: Do we need regularization during pre-training?

- 🤔🤔🤔
  - There is a lot of data (trillions of tokens), more than parameters
  - Normally, we do 1-epoch training on a corpus (hard to memorize)

- This is all quite reasonable… but what do people do in practice?

Dropout and Weight Decay

- In practice,

Model	Dropout*	Weight decay
Original transformer	0.1	0
GPT2	0.1	0.1
T5	0.1	0
GPT3	0.1	0.1
T5 v1.1	0	0
PaLM	0	(variable)
OPT	0.1	0.1
LLaMA	0	0.1
Qwen 14B	0.1	0.1

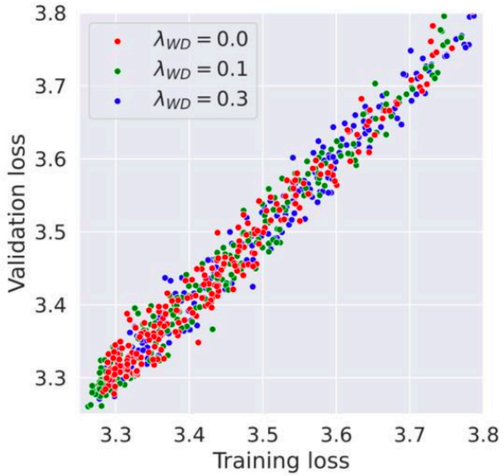
Many older models used dropout during pre-training

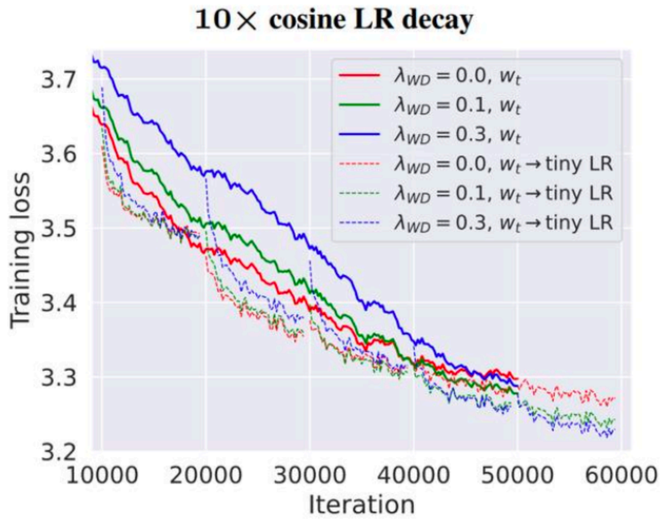
Recent models (except Qwen) rely only on weight decay

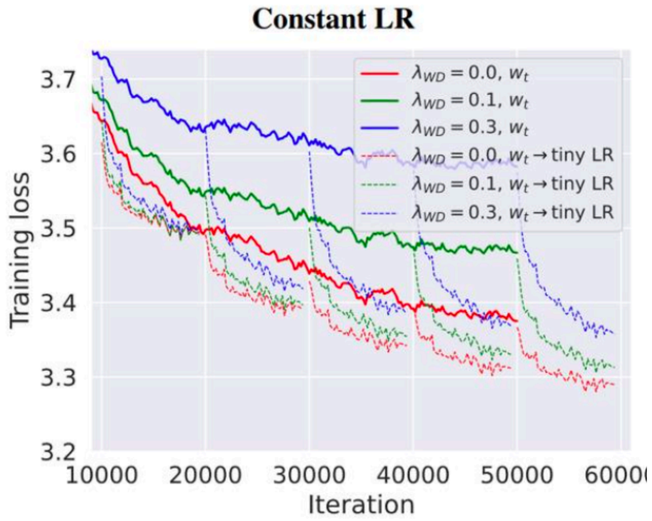
* Most of the time papers just don't discuss dropout. On open models, this closely matches not doing dropout. This may not be true of closed models.

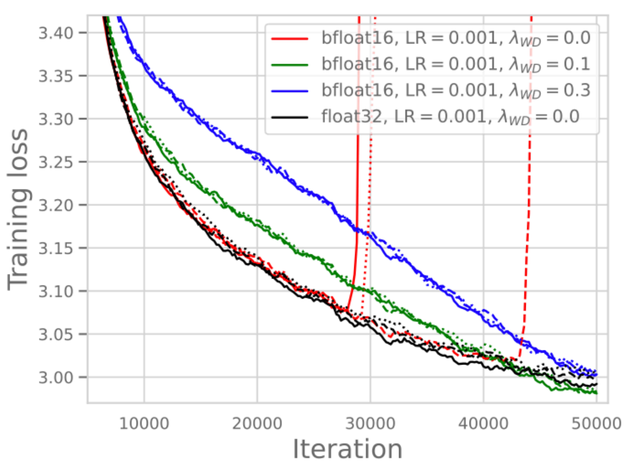
Why Weight Decaying LLMs?

- [Andriushchenko+ 2023] has interesting observations
  - For the under-training regime (like 1-epoch training of LLM)









WD doesn't control overfitting

WD interacts with LR (cosine schedule)

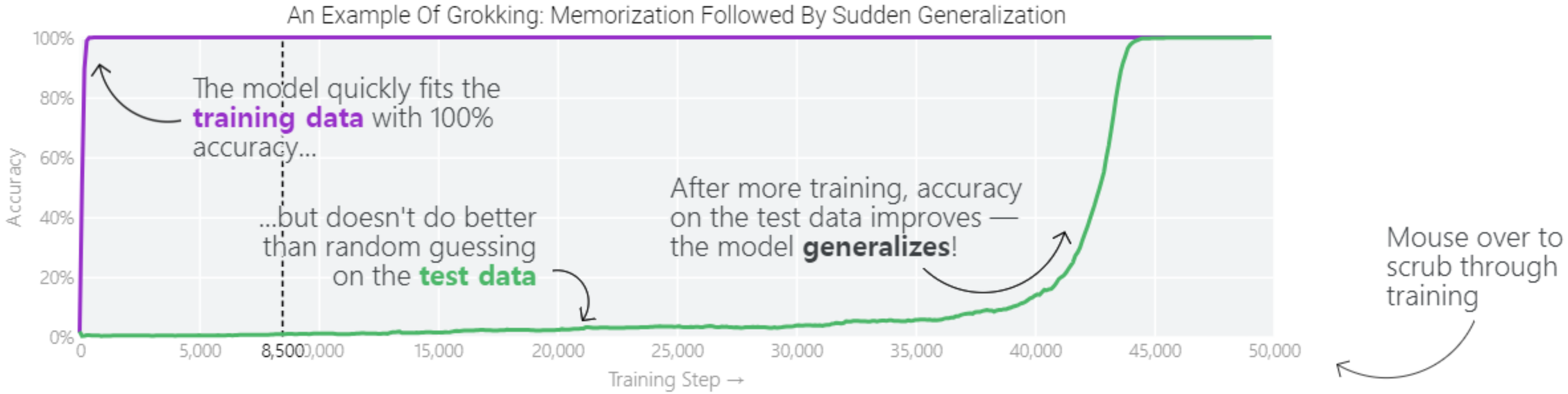
WD increases stability when using BF16

[Power+ 2022, Liu+ 2022, Zhong+ 2023]

Why Weight Decaying LLMs?

- Weight decay is effective at "grokking"

- Grokking (delayed generalization): Generalization after overfitting



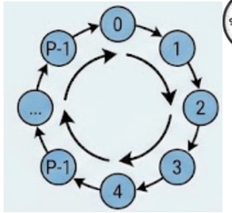
[Power+ 2022, Liu+ 2022, Zhong+ 2023]

Why Weight Decaying LLMs?

- Grokking: Pizza and clock algorithms of modulo operation

Step 1: Embed token a and b to a circle where $w_k = 2\pi k/p$ for some $k \in [1,2\cdots,p-1]$

$$a \to \mathbf{E}_a \equiv (\mathbf{E}_{a,x}, \mathbf{E}_{a,y}) = (\cos(w_k a), \sin(w_k a)), b \to \mathbf{E}_b \equiv (\mathbf{E}_{b,x}, \mathbf{E}_{b,y}) = (\cos(w_k b), \sin(w_k b))$$



Clock Algorithm

Step 2: compute the angle sum using multiplication.

$$\mathbf{E}_{ab} \equiv \begin{pmatrix} \mathbf{E}_{ab,x} \\ \mathbf{E}_{ab,y} \end{pmatrix} = \begin{pmatrix} \mathbf{E}_{a,x}\mathbf{E}_{b,x} - \mathbf{E}_{a,y}\mathbf{E}_{b,y} \\ \mathbf{E}_{a,x}\mathbf{E}_{b,y} + \mathbf{E}_{a,y}\mathbf{E}_{b,x} \end{pmatrix} = \begin{pmatrix} \cos(w_k(a+b)) \\ \sin(w_k(a+b)) \end{pmatrix}$$

$$\mathbf{H}_{ab} = \mathbf{E}_{ab}$$

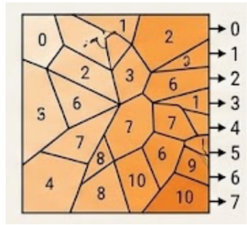
Pizza Algorithm

Step 2.1: compute the vector mean.

$$\mathbf{E}_{ab} = (\mathbf{E}_a + \mathbf{E}_b)/2 = (\cos(w_k a) + \cos(w_k b), \sin(w_k a) + \sin(w_k b))/2$$

Step 2.2: using $\mathbf{E}_{ab}$ and nonlinearities to compute $\mathbf{H}_{ab}$

$$\mathbf{H}_{ab} = |\cos(w_k(a-b)/2)|(\cos(w_k(a+b)), \sin(w_k(a+b)))$$



Generalization

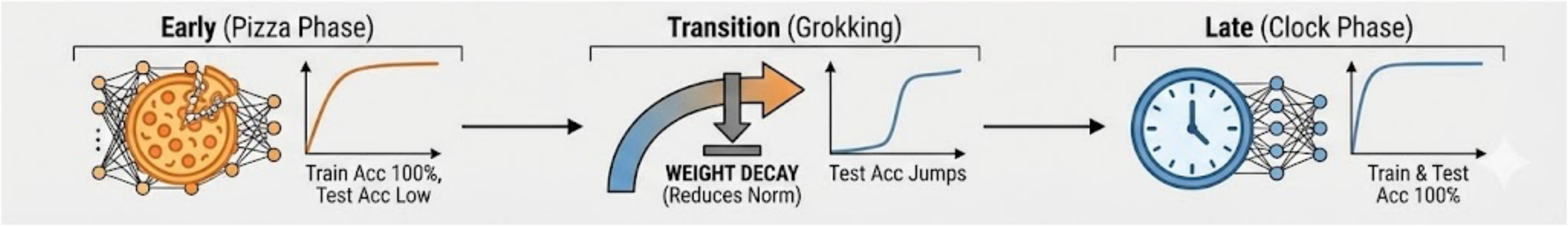
Memorization

Step 3: score possible outputs c using a dot product.

$$Q_{abc} = \mathbf{U}_c \cdot \mathbf{H}_{ab}, \quad \mathbf{U}_c \equiv (\mathbf{E}_{c,x}, \mathbf{E}_{c,y}) = (\cos(w_k c), \sin(w_k c))$$

$$Q_{abc}(\text{Clock}) = \cos(w_k(a+b-c))$$

$$Q_{abc}(\text{Pizza}) = \left|\cos(w_k(a-b)/2)\right| \cos(w_k(a+b-c))$$



But, grokking does not always happen
(e.g. model is so strong so that memorization can solve all the problems)

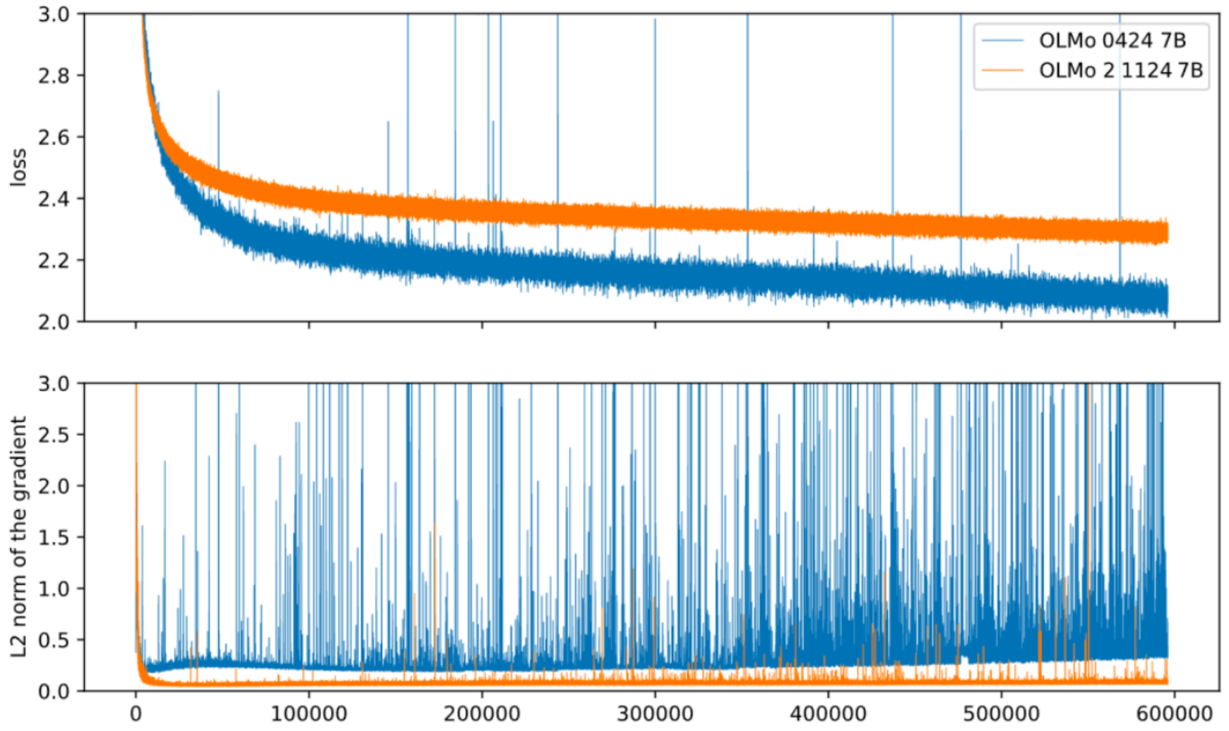
Recap

- FFN Size: Factor-of-4 rule of thumb (8/3 for GLUs) is standard
- Head Dim: `d_head * num_head = d_model` is standard
- Aspect Ratio: Wide range of 'good' values (around 100-200)
- Regularization: You still 'regularize' LMs but its effects are primarily on optimization dynamics

Stability Tricks

Training Stability

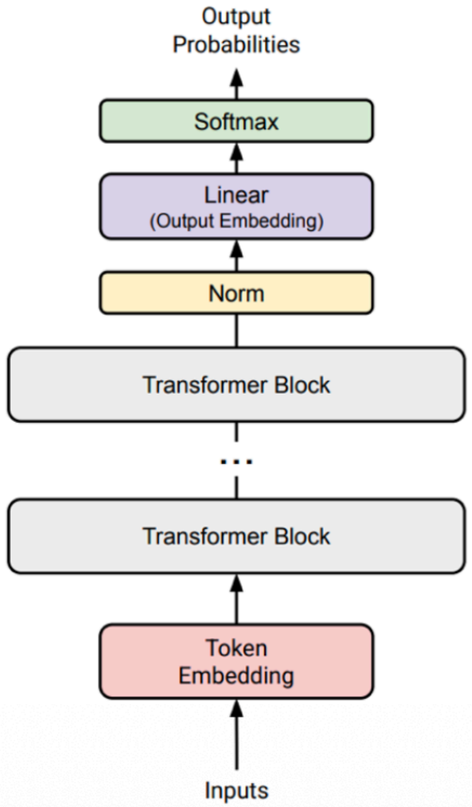
- How to stably train the model is very important

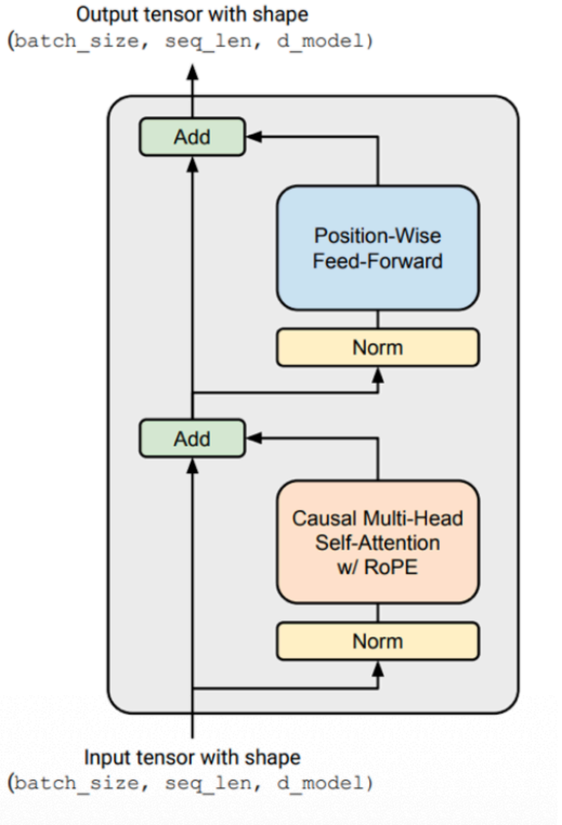


- A lot of modules can reduce stability

Beware of Softmax

- Softmax can be ill-behaved due to exponentials and divide-by-zero
- But in Transformer, Softmaxes are everywhere!





Safe Softmax

$$\text{Softmax} = P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$$

- Problem of Softmax: if x_i is very large, it produces overflow
  - We want to increase numerical stability

- Safe Softmax

Safe Softmax

Softmax = $P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$

- Problem of Softmax: if x_i is very large, it produces overflow
  - We want to increase numerical stability

- Safe Softmax $P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$

Safe Softmax

Softmax = $P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$

- Problem of Softmax: if x_i is very large, it produces overflow
  - We want to increase numerical stability

- Safe Softmax $P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$

$$= \frac{e^{x_i} \cdot e^{-m}}{\left(\sum_{j=1}^{V} e^{x_j}\right) \cdot e^{-m}} \quad \text{where, } m = \max(x)$$

Safe Softmax

Softmax = $P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$

- Problem of Softmax: if x_i is very large, it produces overflow
  - We want to increase numerical stability

- Safe Softmax

$$P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$$

$$= \frac{e^{x_i} \cdot e^{-m}}{\left(\sum_{j=1}^{V} e^{x_j}\right) \cdot e^{-m}} \quad \text{where, } m = \max(x)$$

$$= \boxed{\frac{e^{x_i - m}}{\sum_{j=1}^{V} e^{x_j - m}}}$$

LogSumExp Softmax

Softmax = $P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$

- Softmax with LogSumExp Trick
  - nn.CrossEntropyLoss uses nn.LogSoftmax

LogSumExp Softmax

Softmax = $P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$

- Softmax with LogSumExp Trick
  - nn.CrossEntropyLoss uses nn.LogSoftmax

$$\log P(y_i) = \log\left(\frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}\right)$$

LogSumExp Softmax

Softmax = $P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$

- Softmax with LogSumExp Trick
  - nn.CrossEntropyLoss uses nn.LogSoftmax

$$\log P(y_i) = \log\left(\frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}\right)$$

$$= x_i - \log\left(\sum_{j=1}^{V} e^{x_j}\right)$$

LogSumExp Softmax

Softmax = $P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$

- Softmax with LogSumExp Trick
  - nn.CrossEntropyLoss uses nn.LogSoftmax

$$\log P(y_i) = \log\left(\frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}\right)$$

$$= x_i - \log\left(\sum_{j=1}^{V} e^{x_j}\right)$$

$$= x_i - \log\left(e^{m} \cdot \sum_{j=1}^{V} e^{x_j - m}\right)$$

LogSumExp Softmax

Softmax = $P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$

- Softmax with LogSumExp Trick
  - nn.CrossEntropyLoss uses nn.LogSoftmax

$$\log P(y_i) = \log\left(\frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}\right)$$

$$= x_i - \log\left(\sum_{j=1}^{V} e^{x_j}\right)$$

$$= x_i - \log\left(e^{m} \cdot \sum_{j=1}^{V} e^{x_j - m}\right)$$

$$= x_i - \left(m + \log \sum_{j=1}^{V} e^{x_j - m}\right)$$

Output Softmax: z-Loss Trick [Chowdhery+ 2022]

- Google PaLM uses an auxiliary loss to encourage training stability

Loss function – The model is trained with the standard language modeling loss function, which is the average log probability of all tokens without label smoothing. We additionally use an auxiliary loss of $z\_loss = 10^{-4} \cdot \log^2 Z$ to encourage the softmax normalizer $\log(Z)$ to be close to 0, which we found increases the stability of training.

$$P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}} \leftarrow Z$$

Attention Softmax: QK Norm [Henry+ 2020]

- Q, K are layer (RMS) normed before going into the softmax operation
- Vanilla Attention: $\mathbf{softmax}(\frac{QK^T}{\sqrt{d}})V$
- QK Norm Attention: $\mathbf{softmax}(g * \hat{Q}\hat{K}^T)V$
  - $\hat{Q}, \hat{K}$: normalized QK
  - g: learnable scalar param to compensate the representation power
  - Note: $1/\sqrt{d}$ can be removed

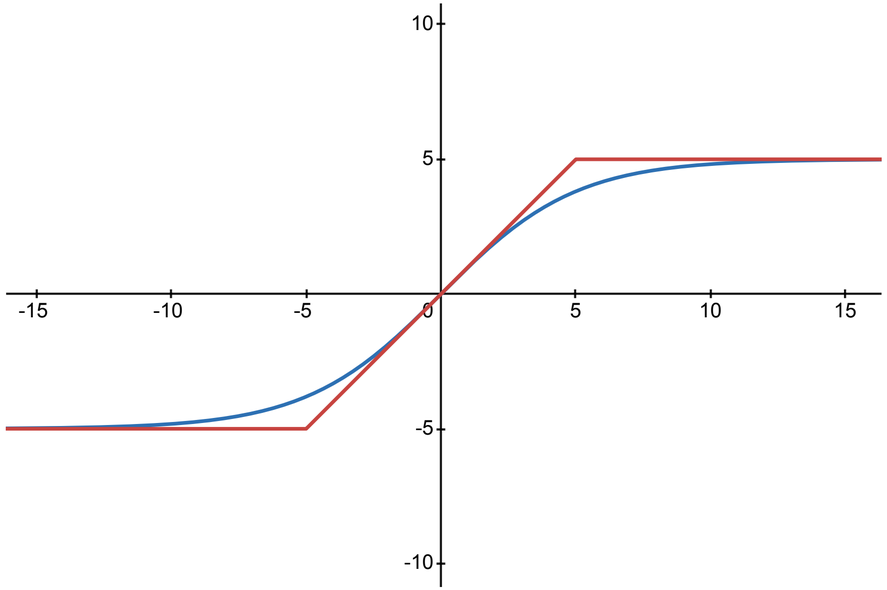
Logit Soft-Capping [Gemma Team 2024]

- Soft-capping the logits to some maximum value via Tanh
  - Softcap:

$$\text{softcap}(x) = t * \tanh(\frac{x}{t})$$

  - Hardcap: clip x in [-t, t]

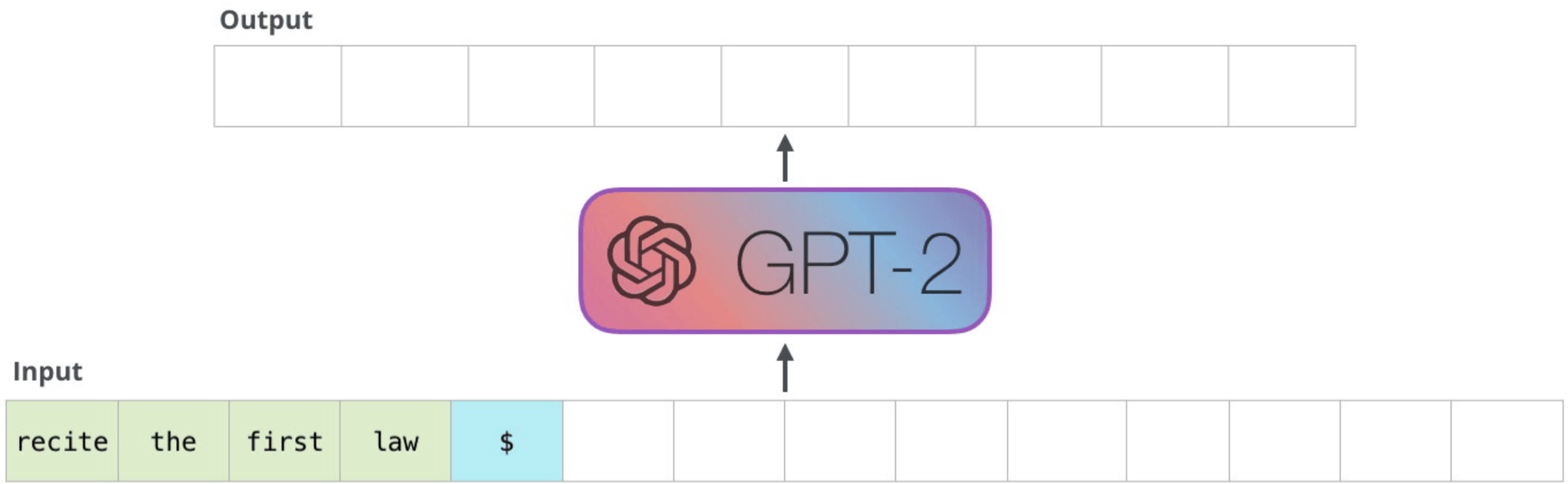
- Gemma 2 uses t = 30.0 for output softmax, and t = 50.0 for attention softmax



Attention Variants

Recap: Auto-Regressive Generation

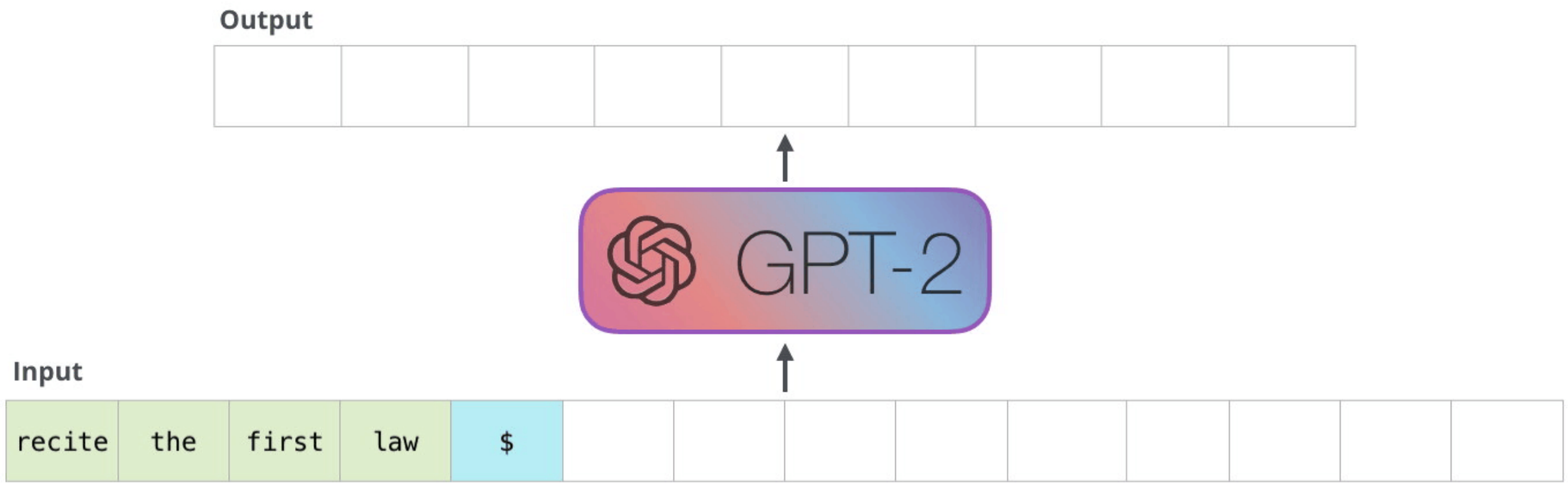
- Transformer decoder predicts the next token one-by-one



https://jalammar.github.io/illustrated-gpt2/

Recap: Auto-Regressive Generation

- Transformer decoder predicts the next token one-by-one



https://jalammar.github.io/illustrated-gpt2/

Recap: Self-Attention

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

I had a warm and cozy morning today

$$q = W_q x_i$$

$$K = W_k x$$

$$V = W_v x$$

Recap: Self-Attention

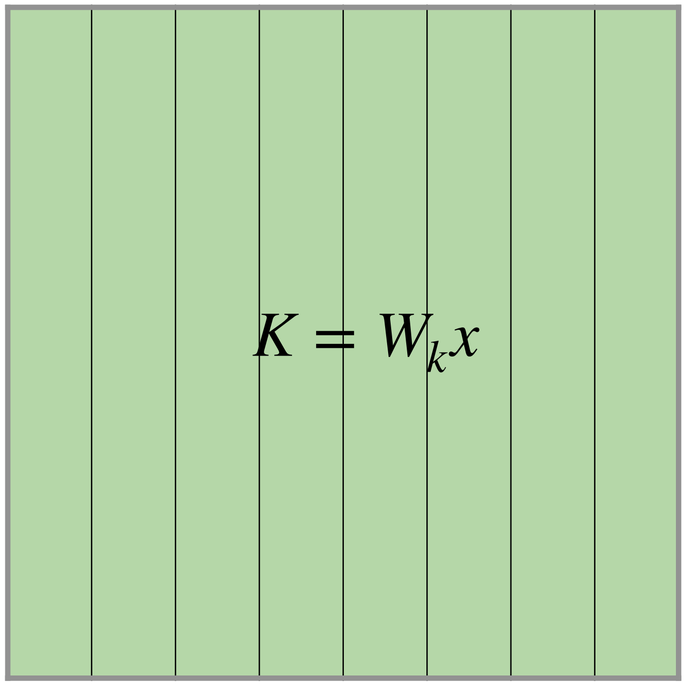
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

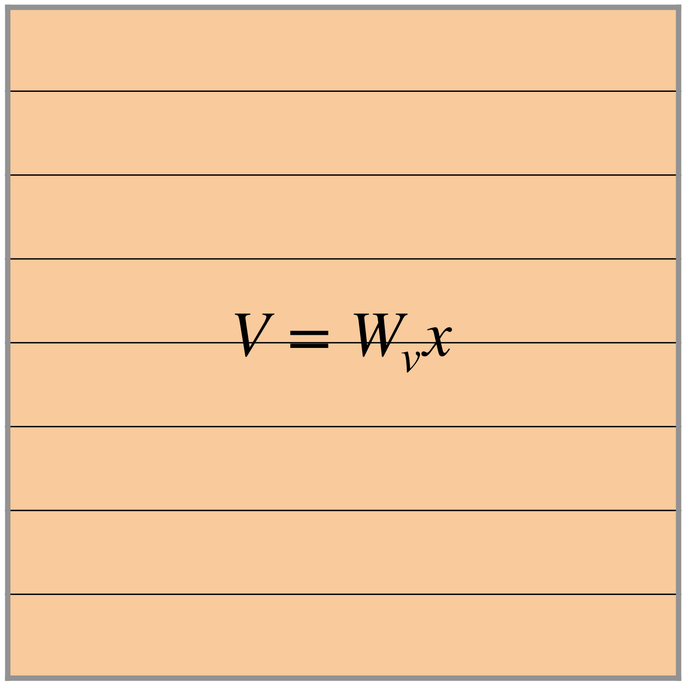
I had a warm and cozy morning today

Value
Weighted sum over previous context

$$q = W_q x_i$$

Query
New token in this decoder step

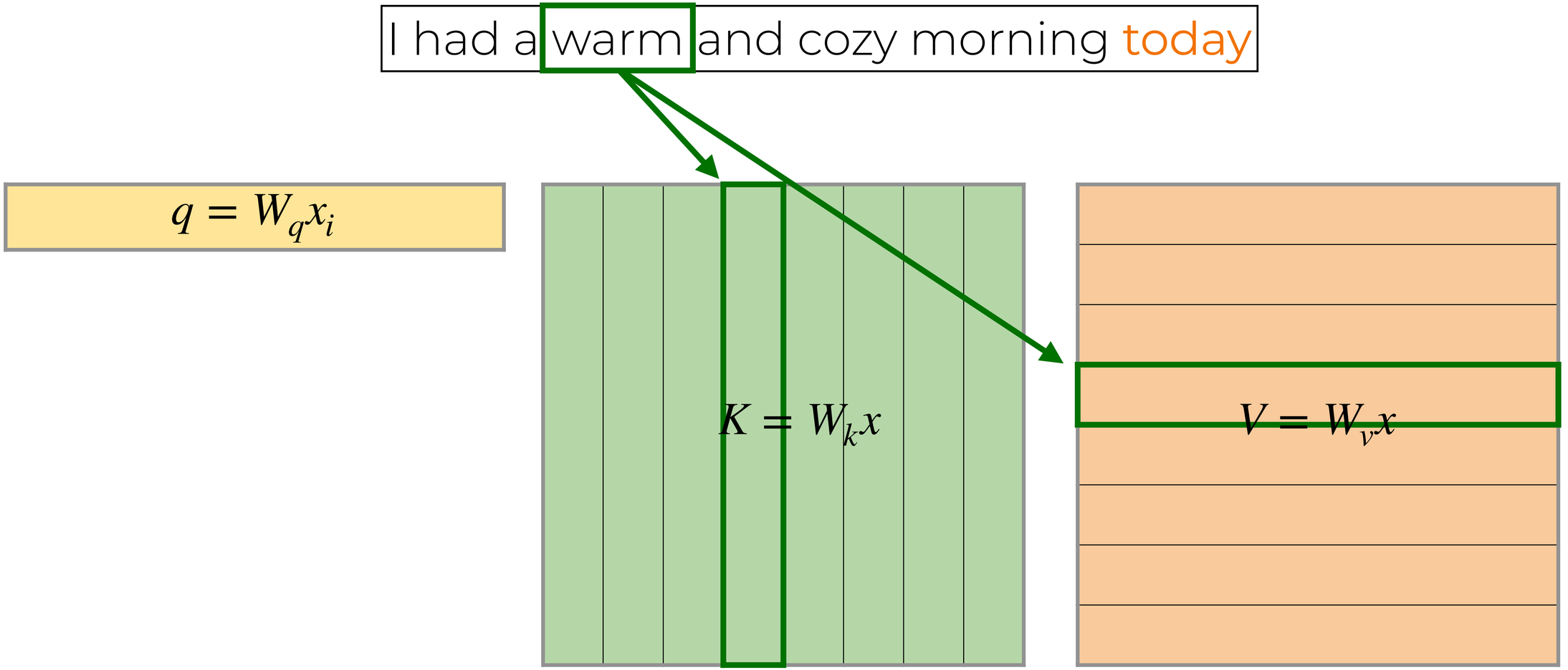




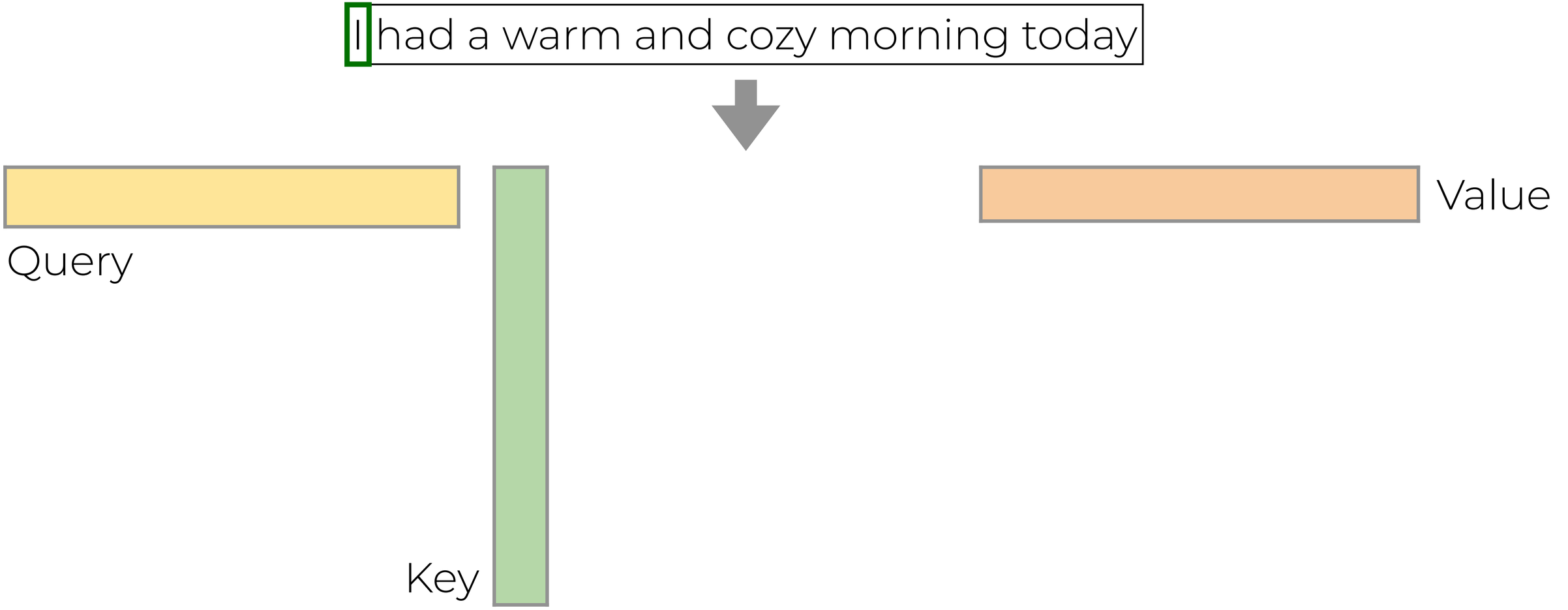
Key
Previous context that model should attend

Recap: Self-Attention

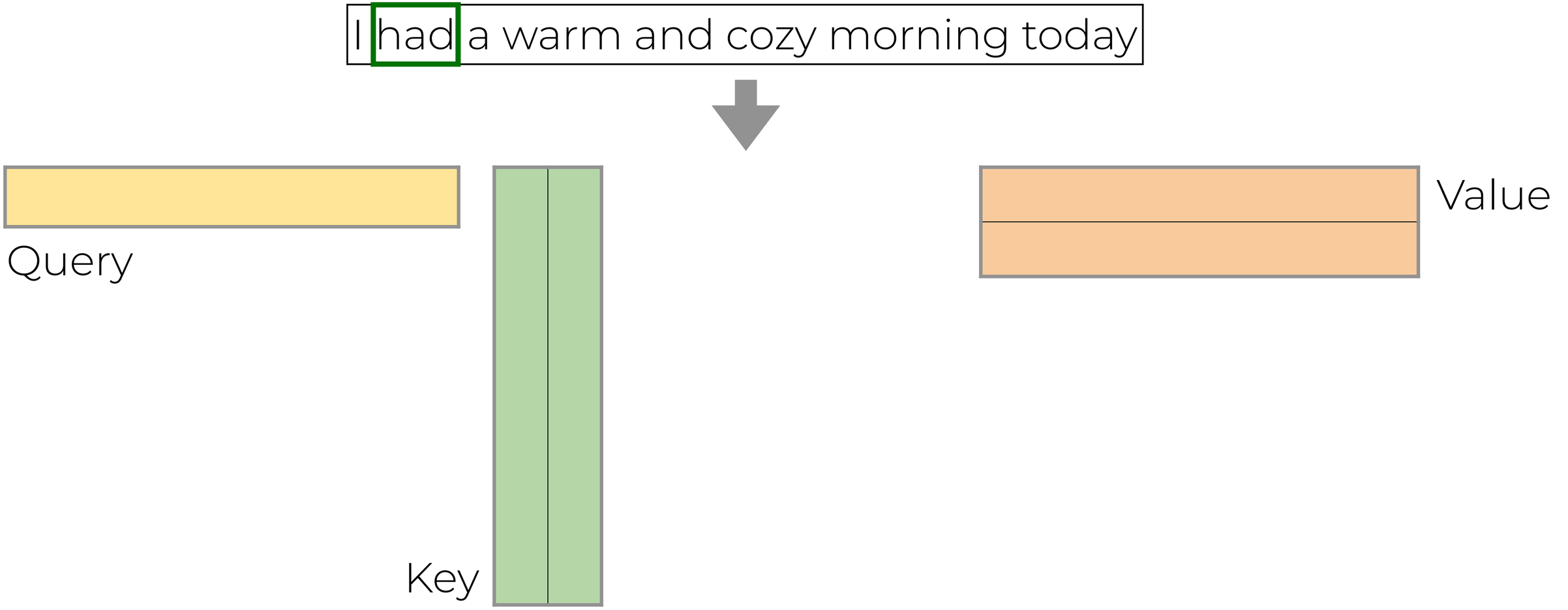
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$



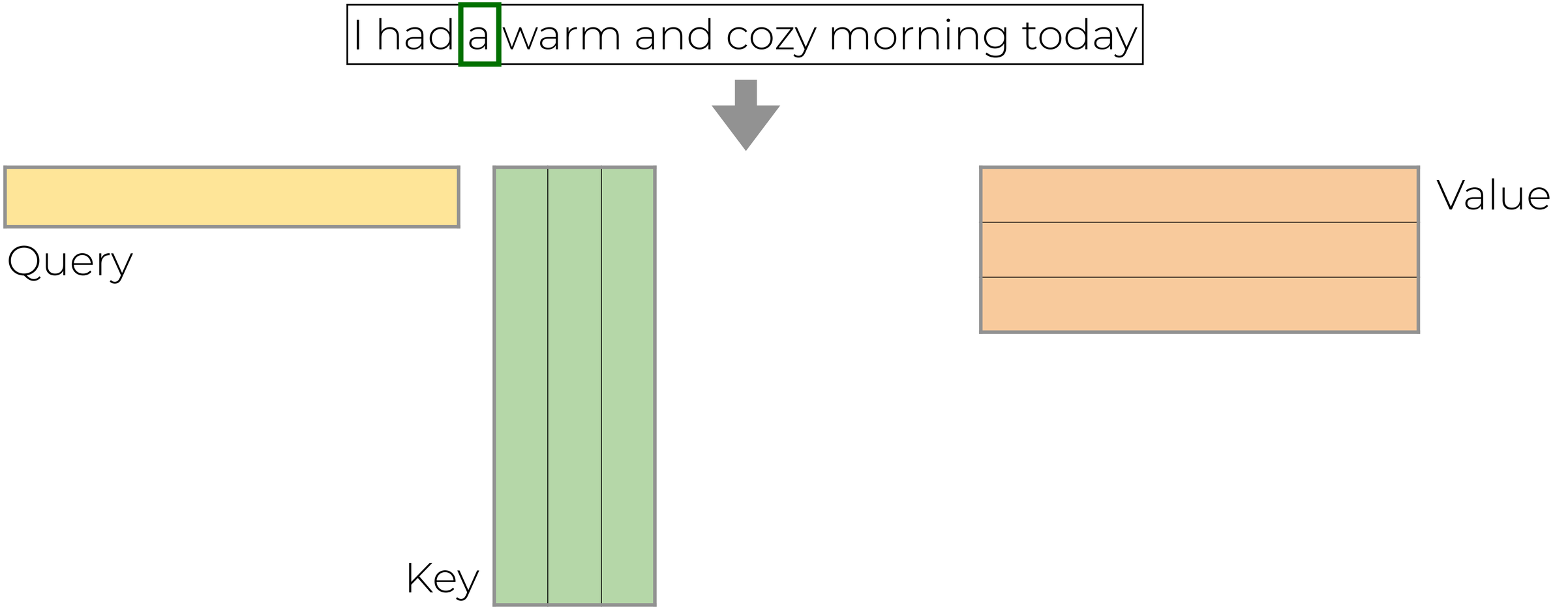
Inference: Naive Sampling



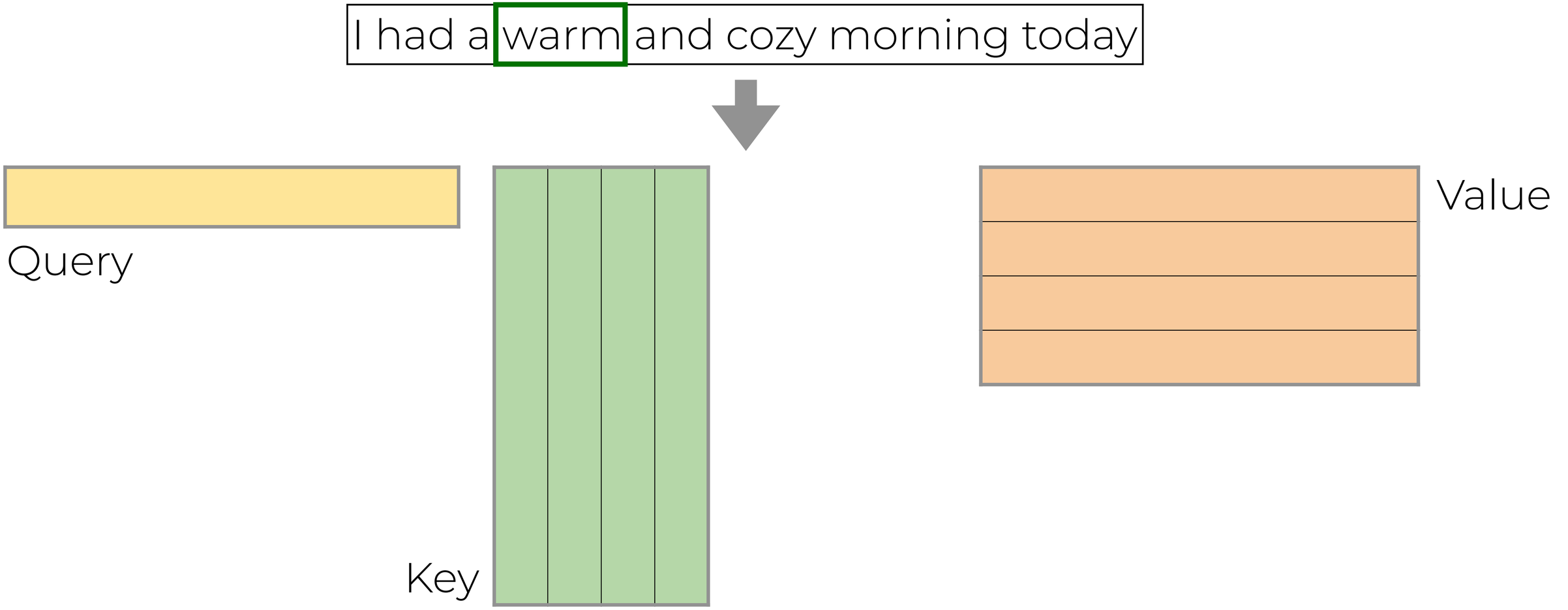
Inference: Naive Sampling



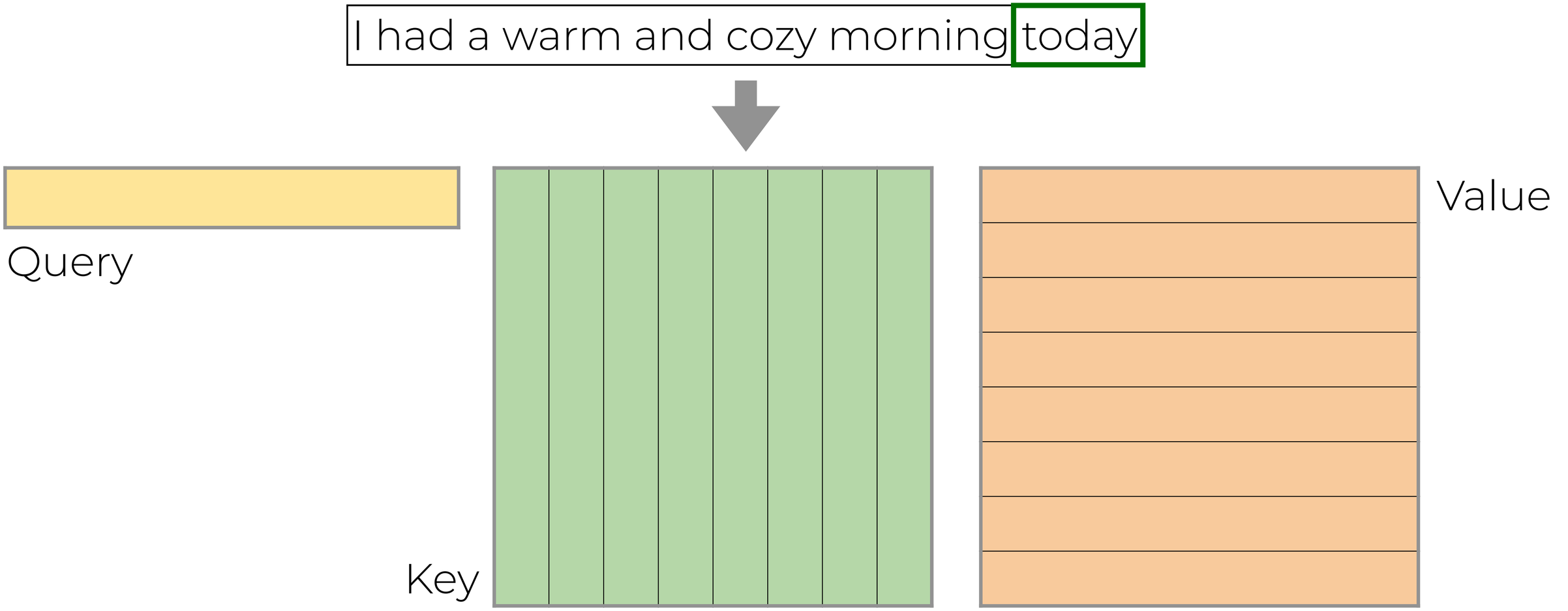
Inference: Naive Sampling



Inference: Naive Sampling

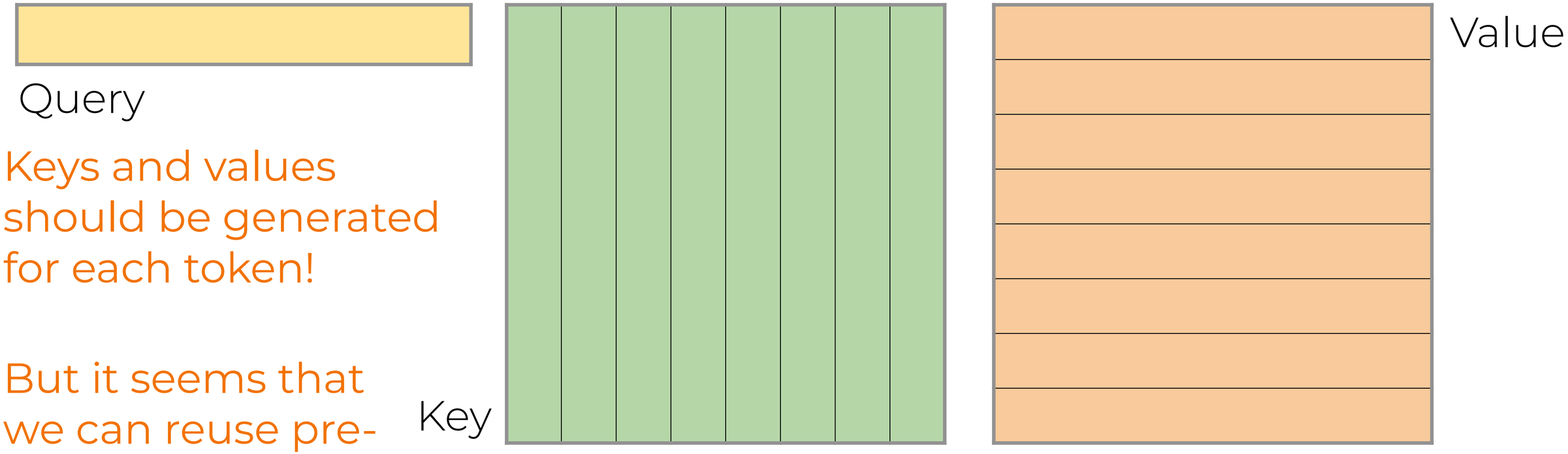


Inference: Naive Sampling



Inference: Naive Sampling

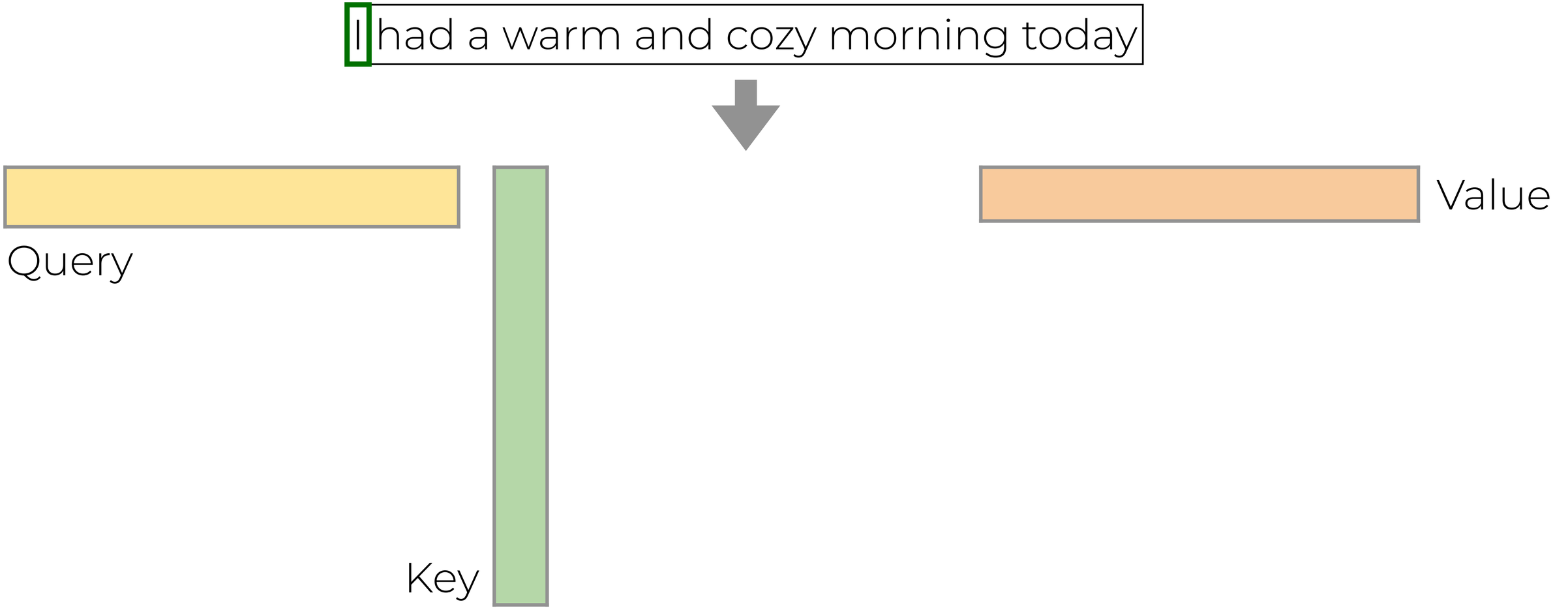
I had a warm and cozy morning today



Keys and values should be generated for each token!

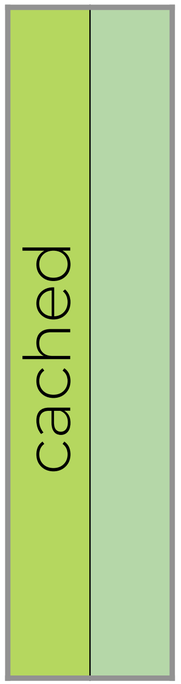
But it seems that we can reuse pre-computed KV

Inference: Sampling with KV-Cache



Inference: Sampling with KV-Cache

Query



Key

Value

Caching KV!

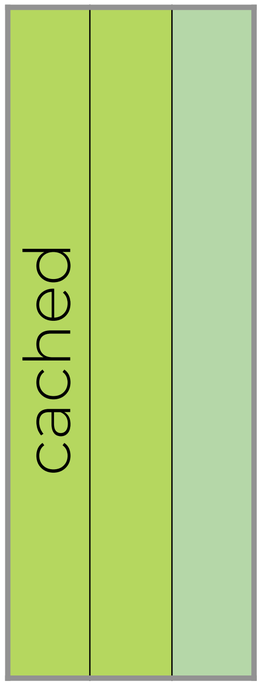
Store KV in memory
and no re-compute

Inference: Sampling with KV-Cache

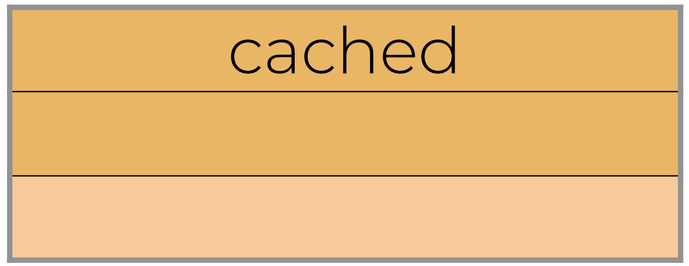
Query

Caching KV!

Store KV in memory
and no re-compute

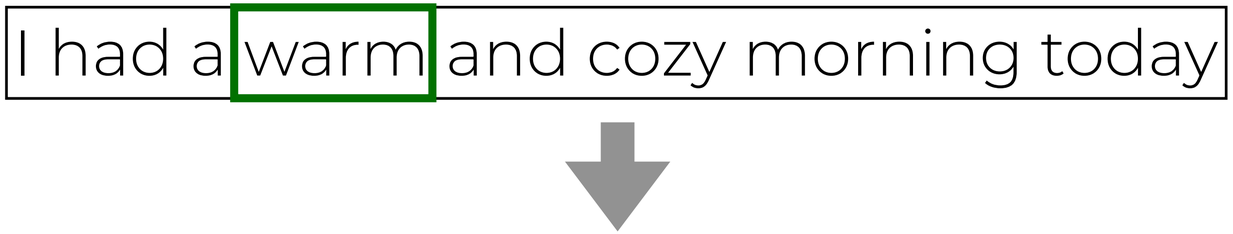


Key



Value

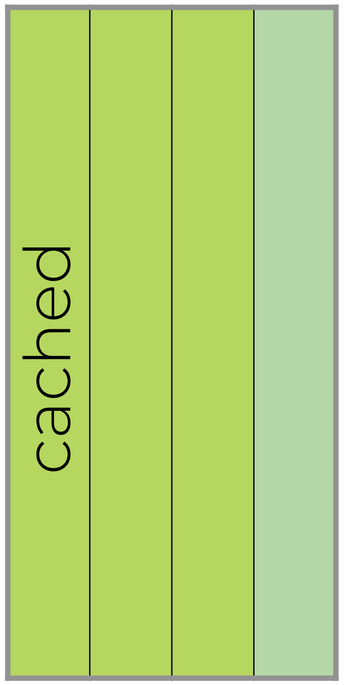
Inference: Sampling with KV-Cache



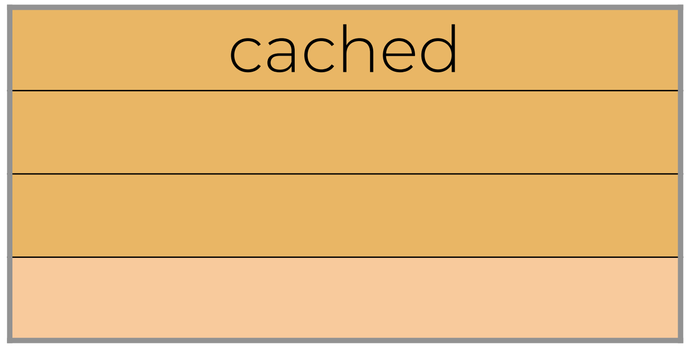
Query

Caching KV!

Store KV in memory
and no re-compute



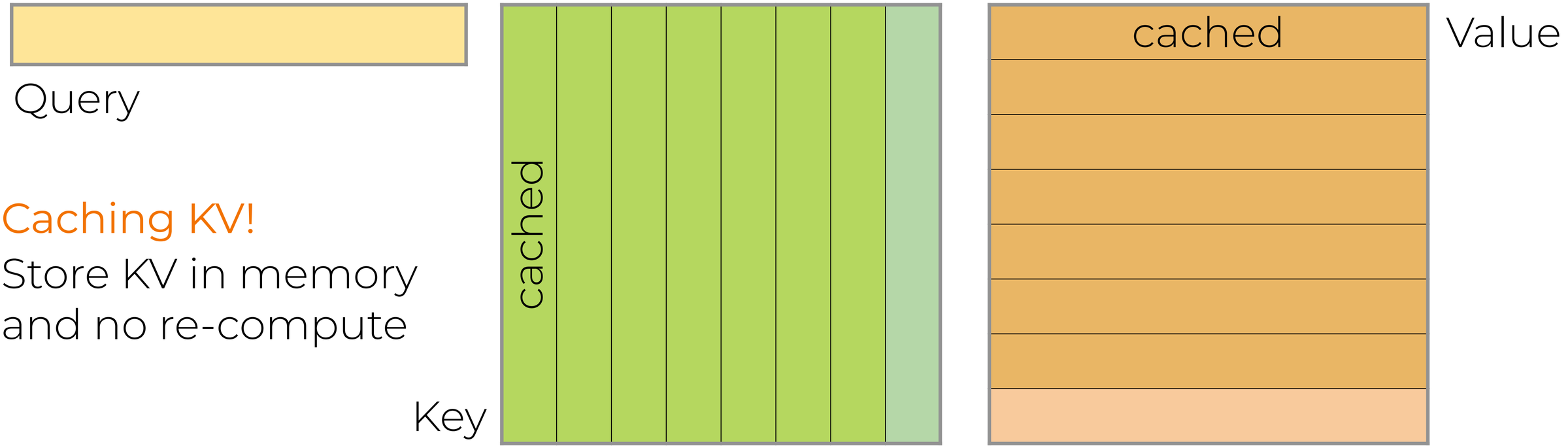
Key



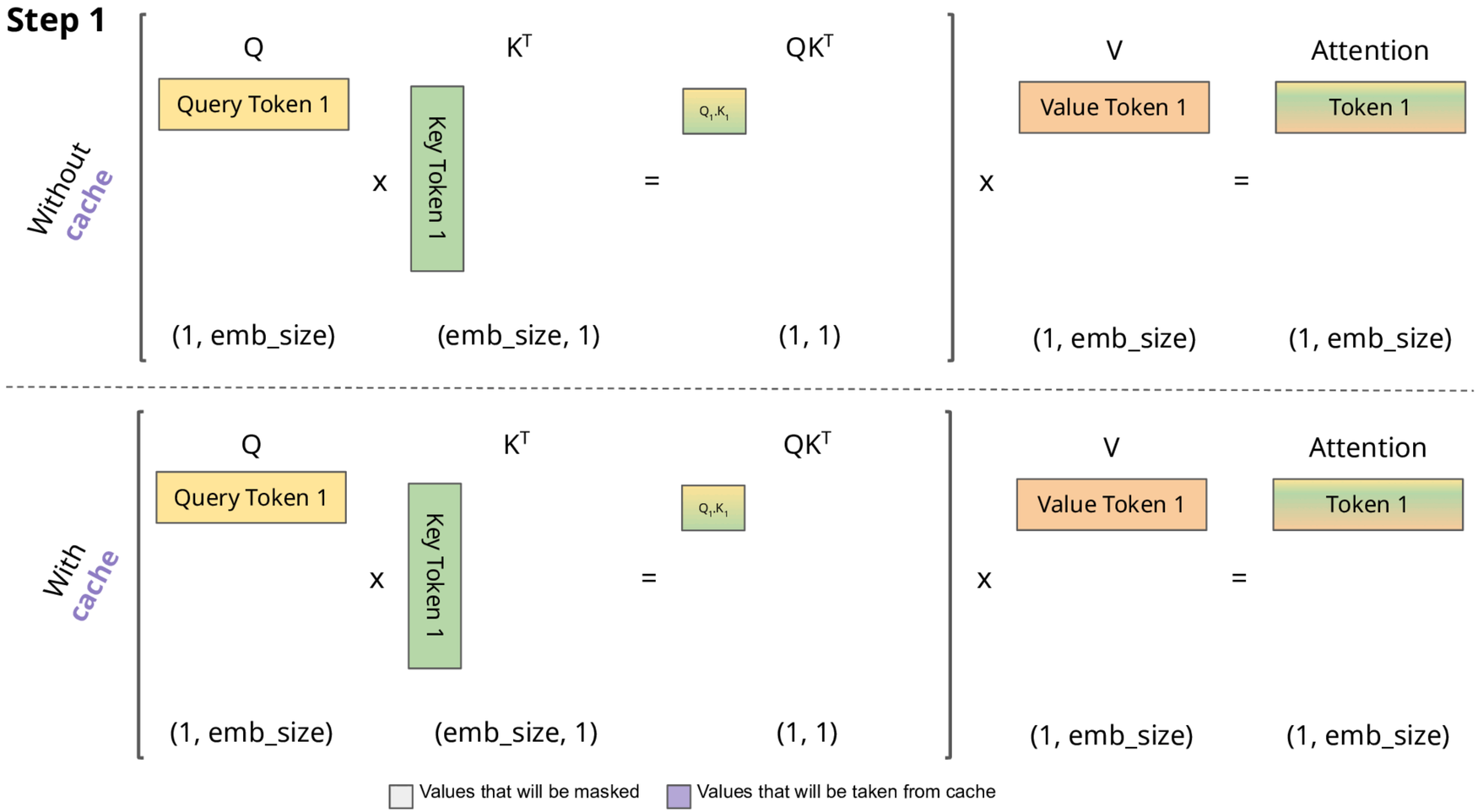
Value

Inference: Sampling with KV-Cache

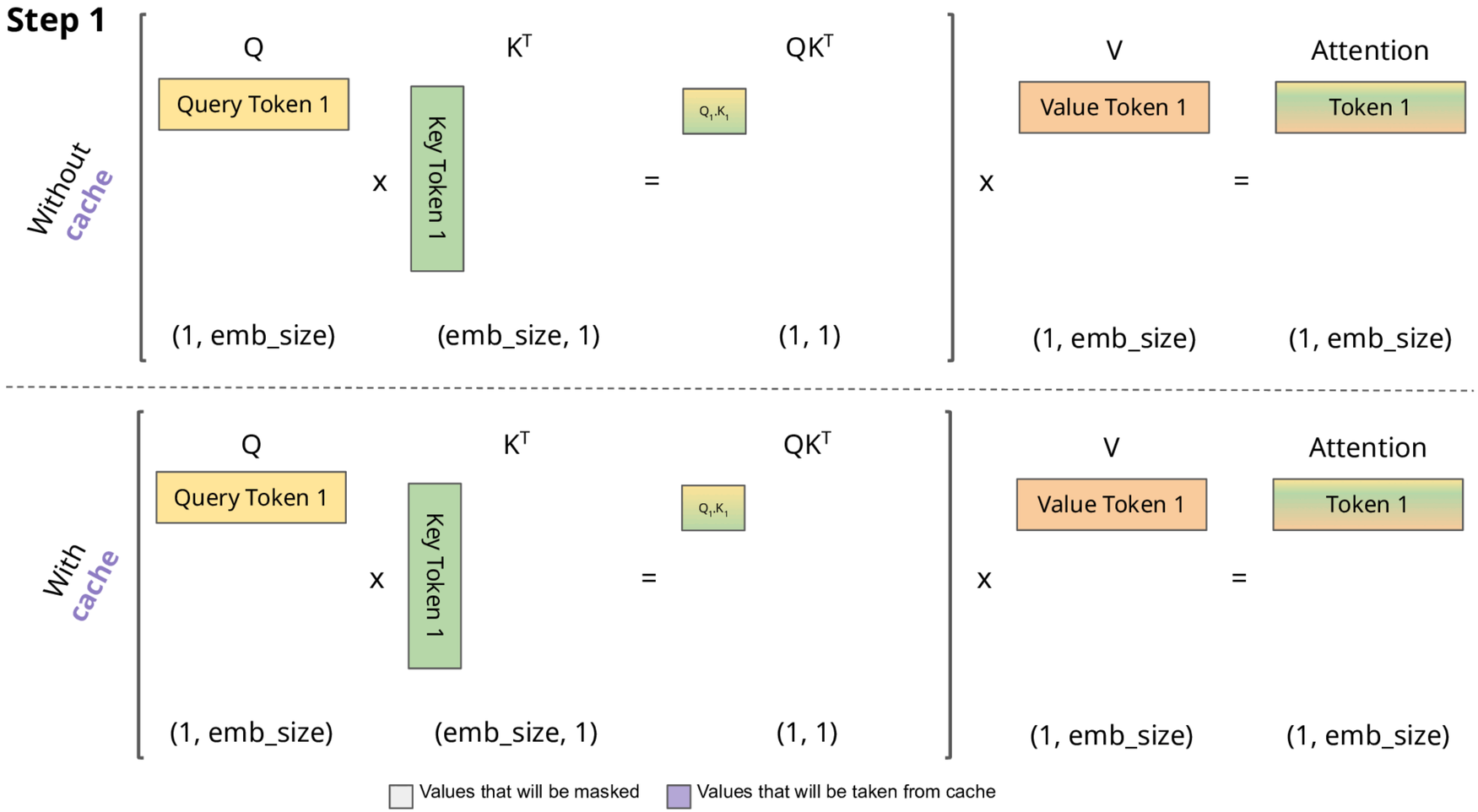
I had a warm and cozy morning today



Inference: Sampling with KV-Cache



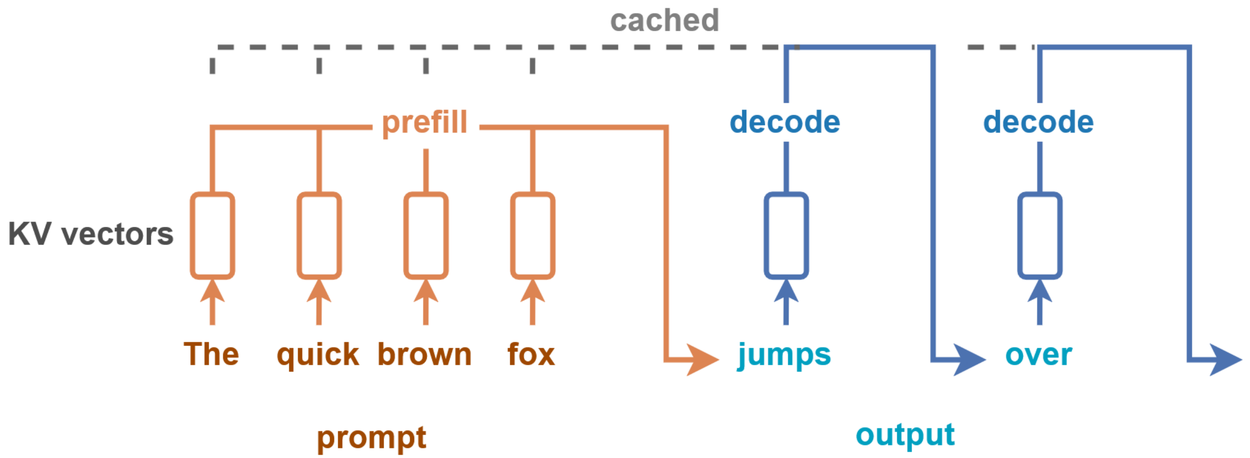
Inference: Sampling with KV-Cache



Inference: Prefil and Decode Phases

- Prefil phase: Parallel processing of the entire input sequence
  - Transforms all input tokens into KV pairs to populate the KV cache

- Decode phase: Sequential generation of tokens, one at a time
  - Reuses KV cache; generates QKV for the current token only
  - Append the new KV pair to the KV cache



KV-Caching: Memory Usage

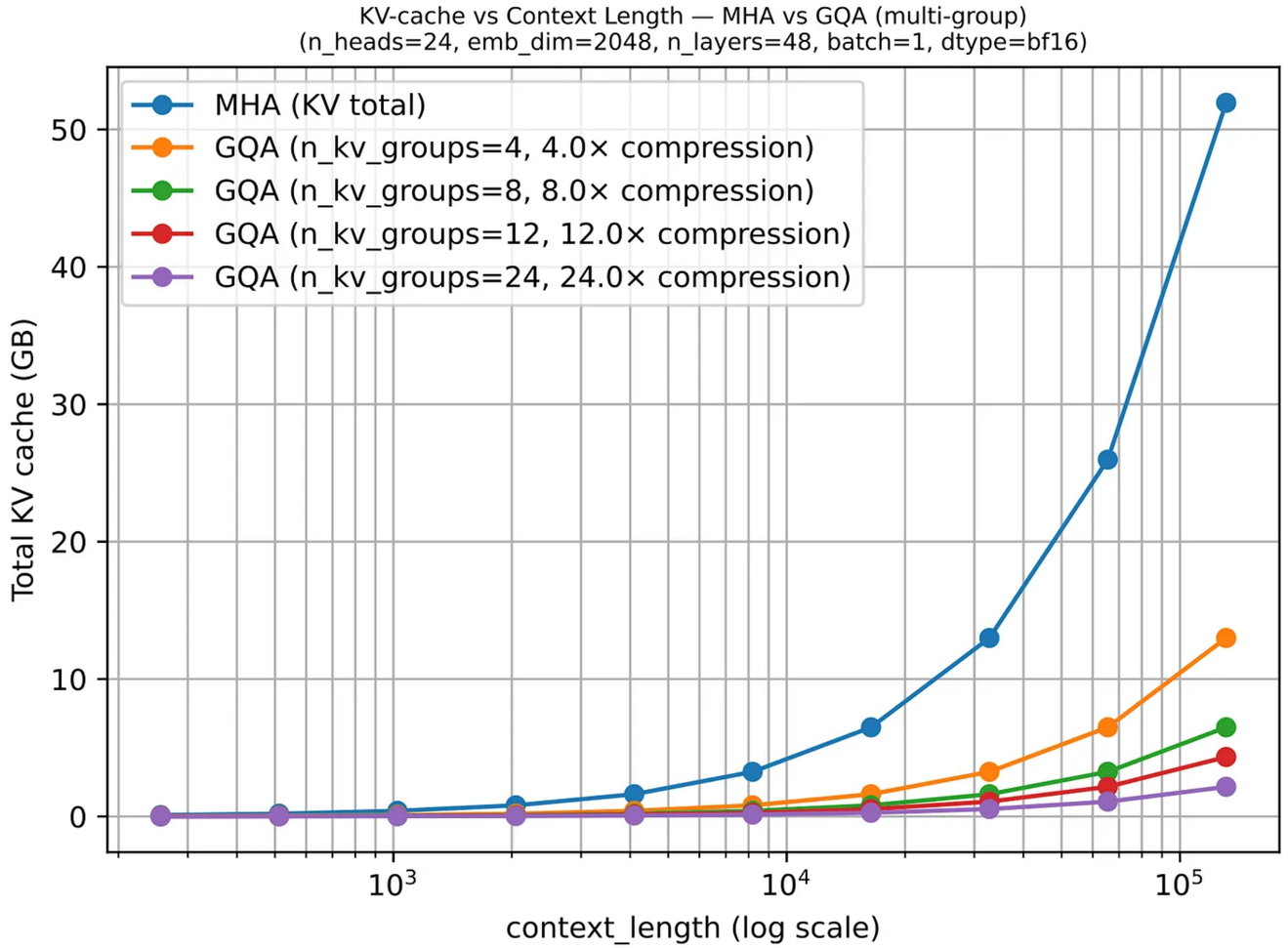
2 x precision x KV dim x # heads x # layers x # seq. length

- For example, DeepSeekR1 / V3
  - 2 = key/value
  - precision = 2 bytes per elems
  - KV dim = d_k, d_v = 128
  - # heads = 128
  - # layers = 61
  - # seq. length = 32,768 tokens

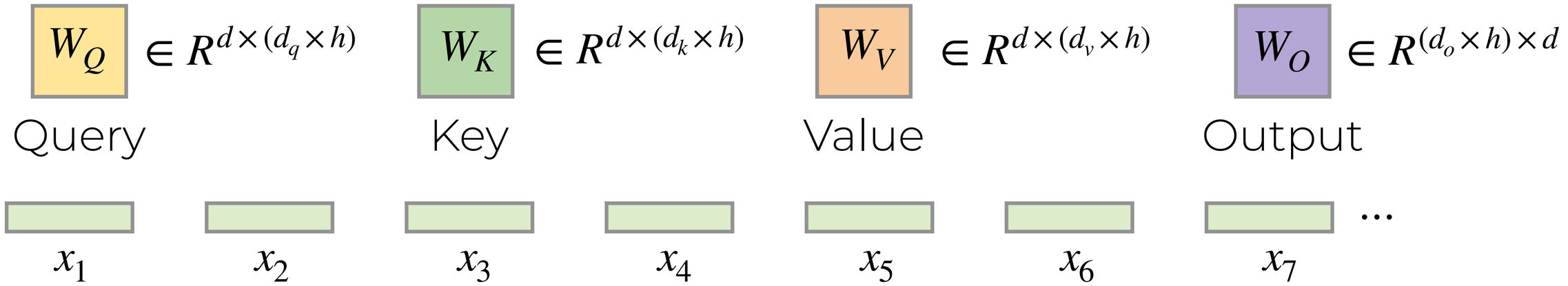
  - Total required memory for just KV-cache is 131 GB!

https://github.com/rasbt/LLMs-from-scratch/tree/main/ch04/04_gqa

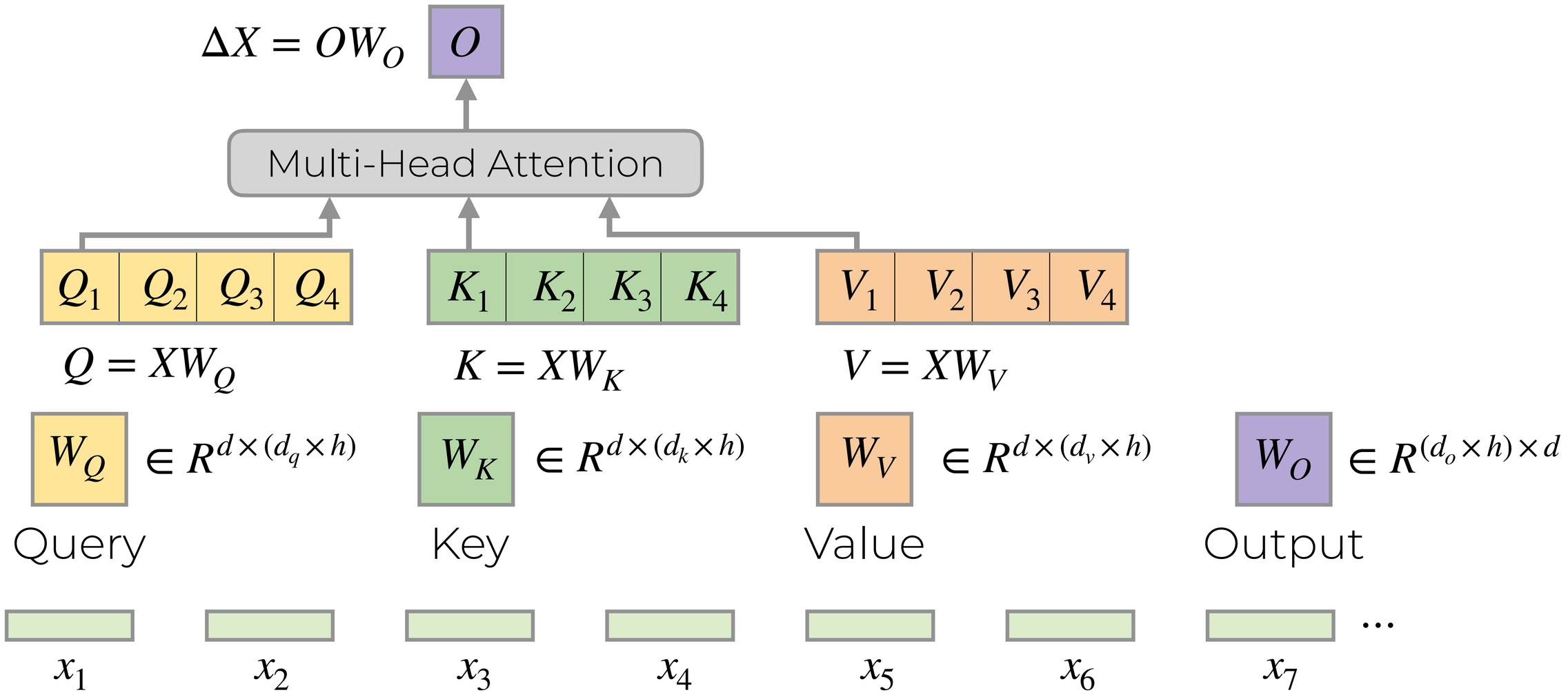
KV-Caching: Memory Usage



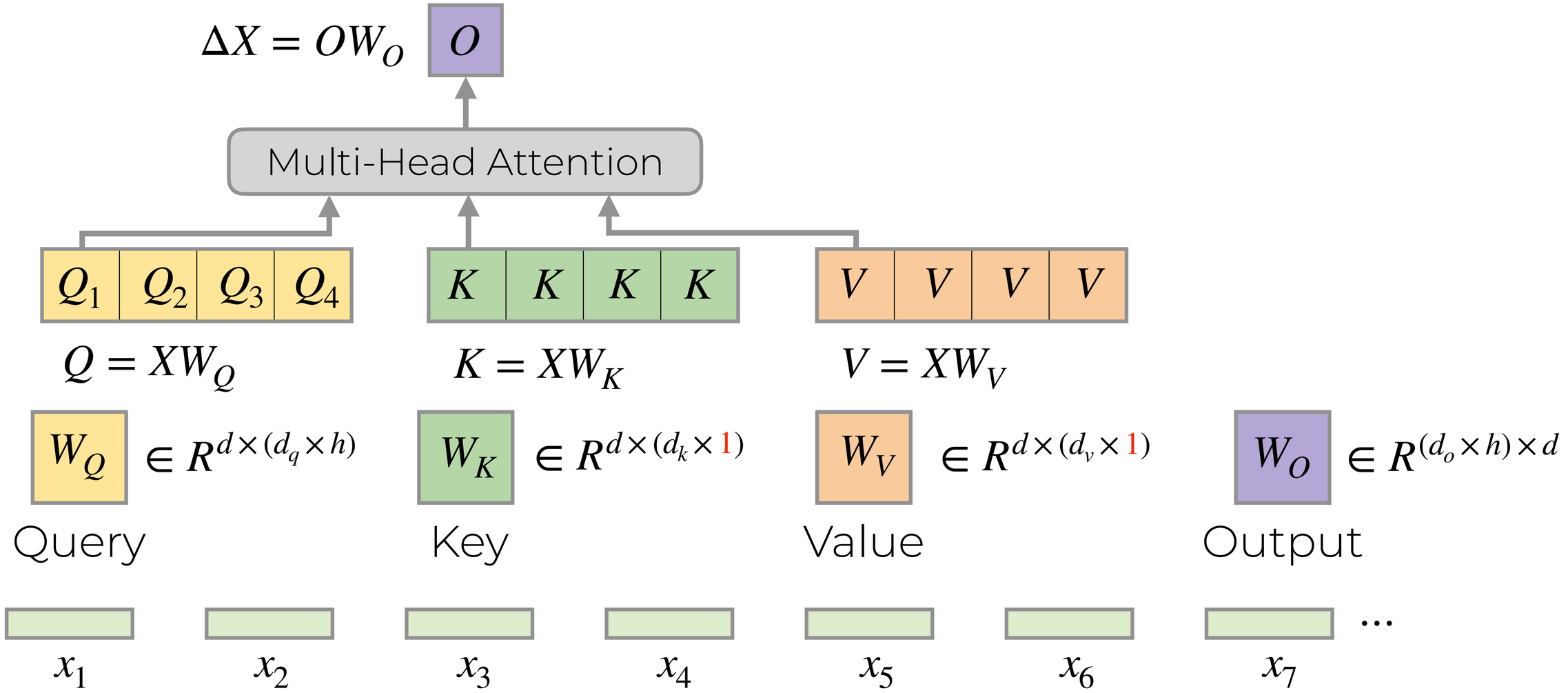
Multi-Head Attention (MHA)



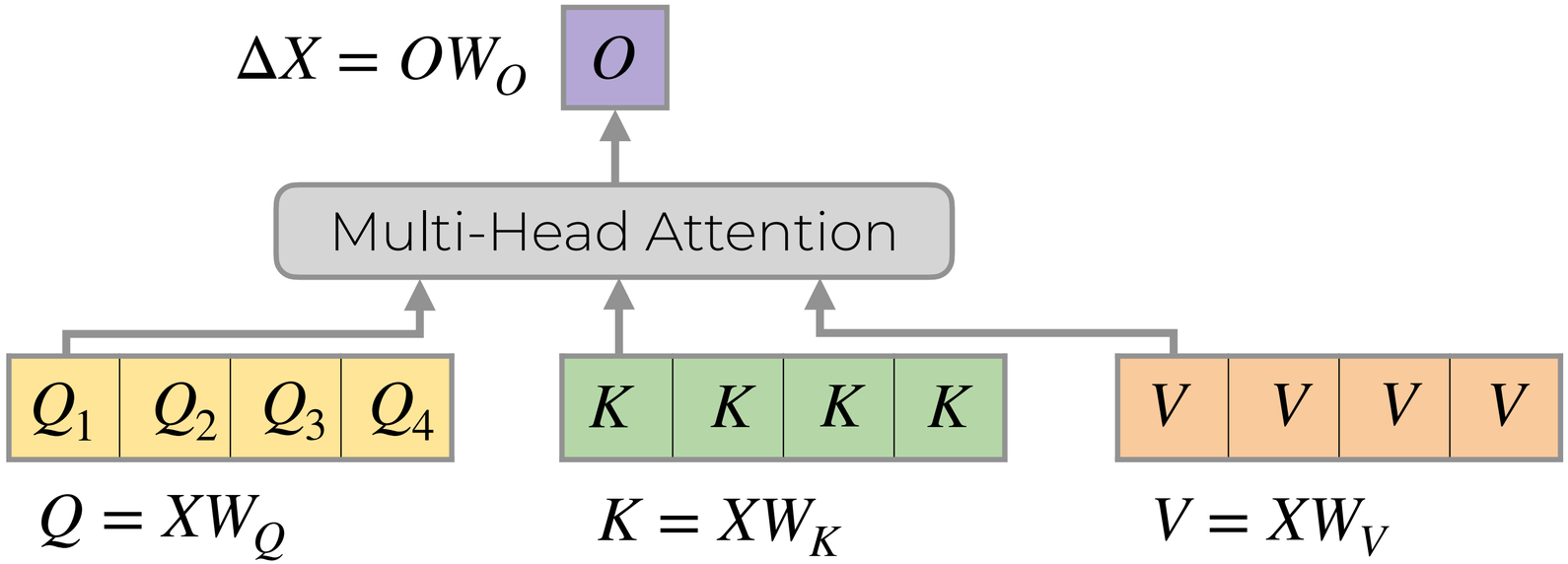
Multi-Head Attention (MHA)



Multi-Query Attention (MQA) [Shazeer+ 2019]



Multi-Query Attention (MQA) [Shazeer+ 2019]



KV-cache per token

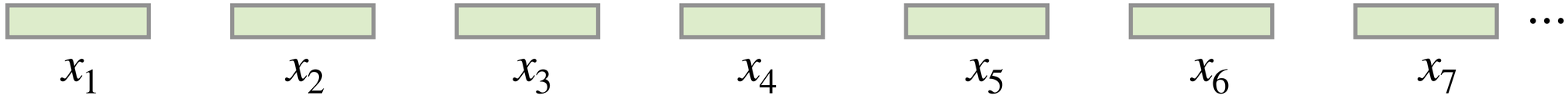
- MHA = 4MB
- MQA = 31KB

$W_Q \in R^{d \times (d_q \times h)}$ Query

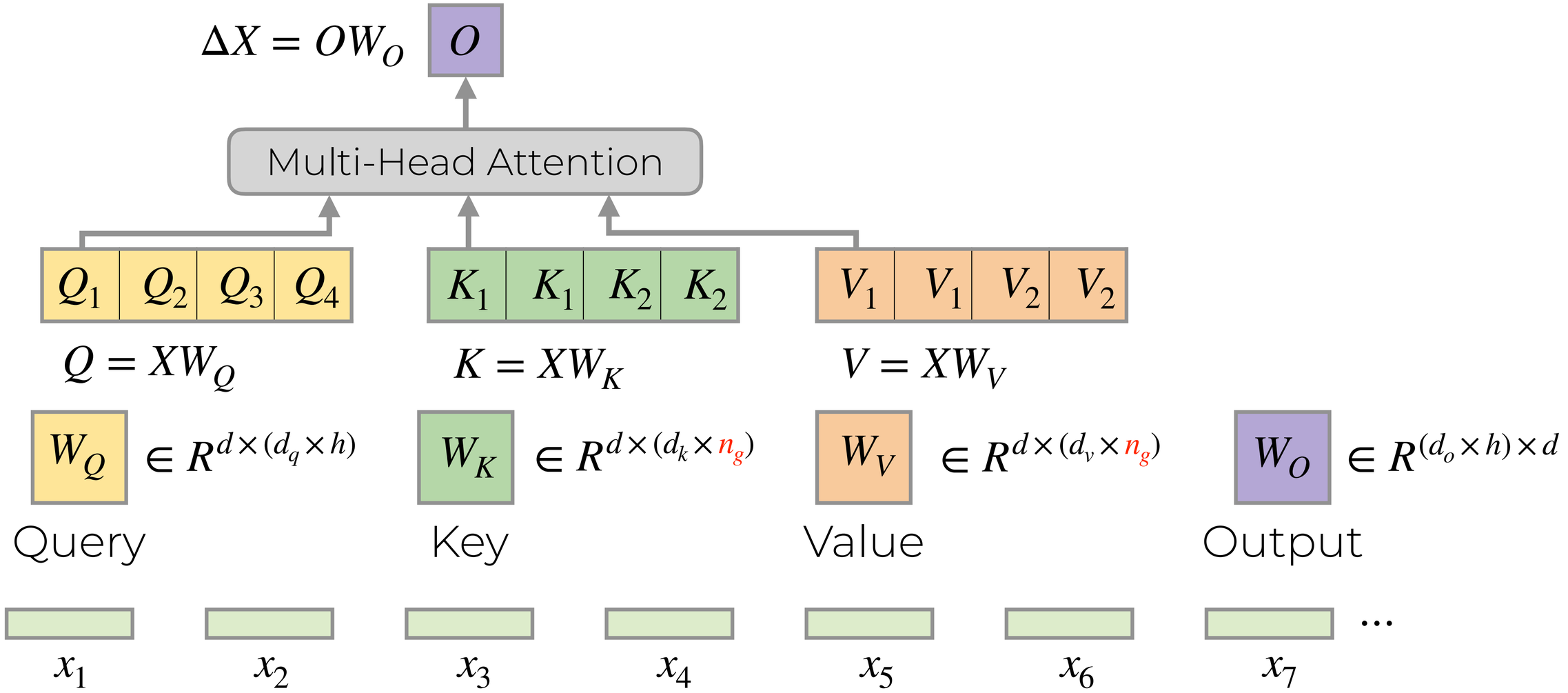
$W_K \in R^{d \times (d_k \times 1)}$ Key

$W_V \in R^{d \times (d_v \times 1)}$ Value

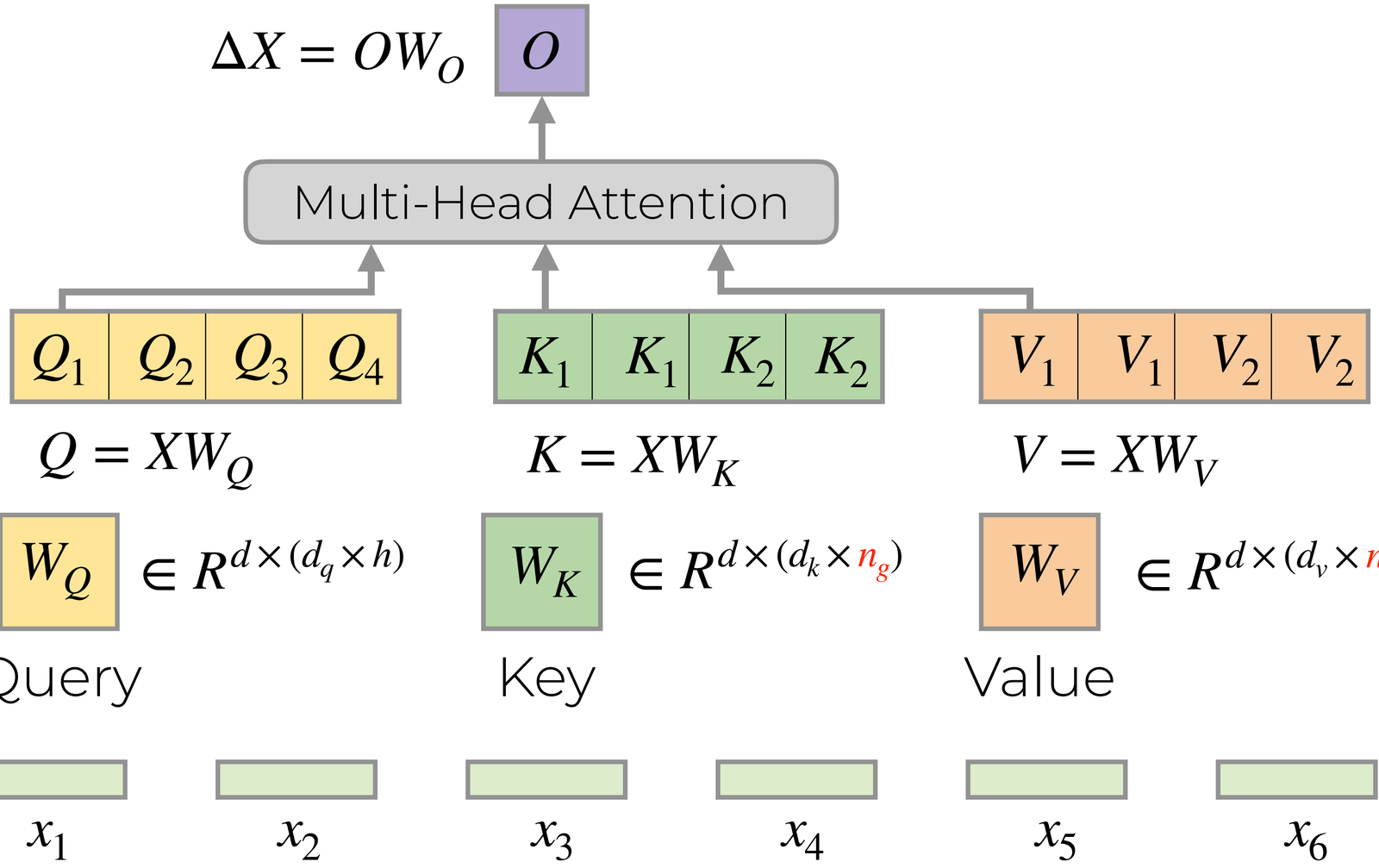
$W_O \in R^{(d_o \times h) \times d}$ Output



Grouped-Query Attention (GQA) [Ainslie+ 2023]



Grouped-Query Attention (GQA) [Ainslie+ 2023]

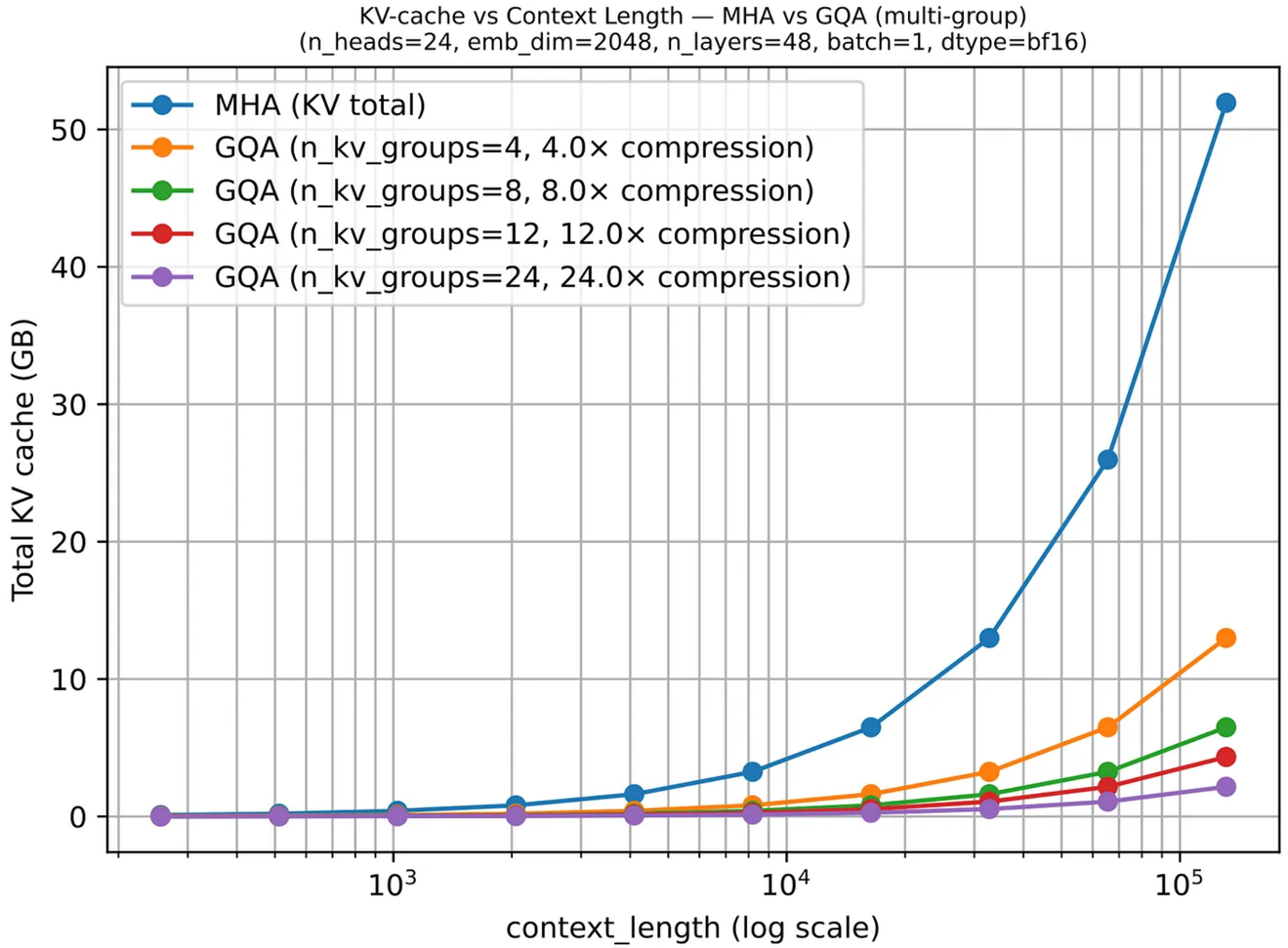


KV-cache per token

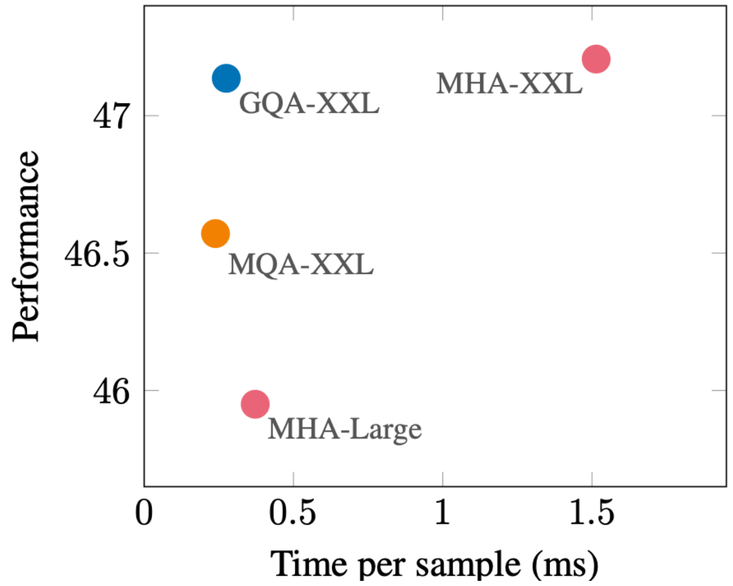
- MHA = 4MB
- MQA = 31KB
- GQA = 500KB

https://github.com/rasbt/LLMs-from-scratch/tree/main/ch04/04_gqa

MHA vs. GQA: Memory Usage

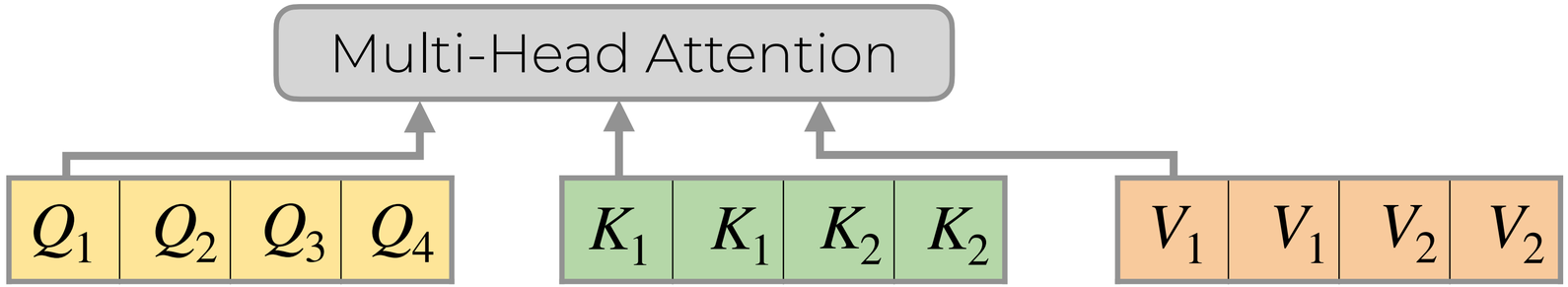


MHA vs. MQA vs. GQA [Ainslie+ 2023]



Model	T_{infer}	Average	CNN	arXiv	PubMed	MediaSum	MultiNews	WMT	TriviaQA
	s		R_1	R_1	R_1	R_1	R_1	BLEU	F1
MHA-Large	0.37	46.0	42.9	44.6	46.2	35.5	46.6	27.7	78.2
MHA-XXL	1.51	47.2	43.8	45.6	47.5	36.4	46.9	28.4	81.9
MQA-XXL	0.24	46.6	43.0	45.0	46.9	36.1	46.5	28.5	81.3
GQA-8-XXL	0.28	47.1	43.5	45.4	47.7	36.3	47.2	28.4	81.6

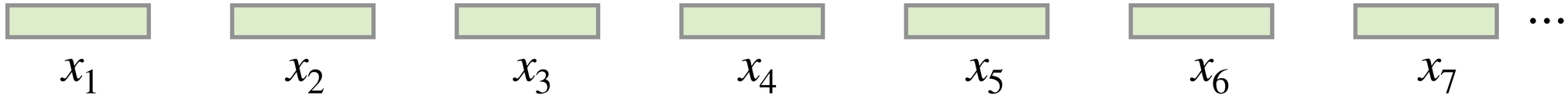
Revisit GQA



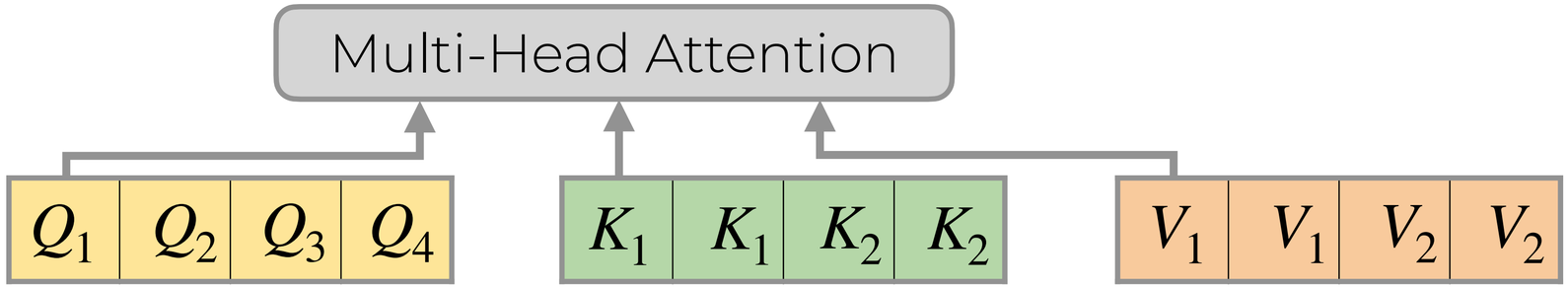
$$Q = XW_Q \qquad [K_1, K_2] = XW_K \qquad [V_1, V_2] = XW_V$$

$W_Q \in R^{d \times (d_q \times h)}$ $W_K \in R^{d \times (d_k \times n_g)}$ $W_V \in R^{d \times (d_v \times n_g)}$ $W_O \in R^{(d_o \times h) \times d}$

Query Key Value Output



Revisit GQA

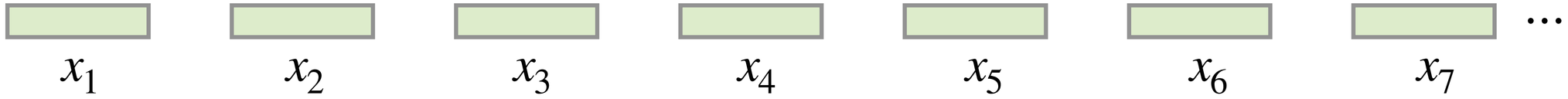


$$[K_1, K_1, K_2, K_2] = [K_1, K_2] \begin{bmatrix} I_{d_k} & I_{d_k} & 0 & 0 \\ 0 & 0 & I_{d_k} & I_{d_k} \end{bmatrix} = [K_1, K_2]\, W_K^{\uparrow}$$

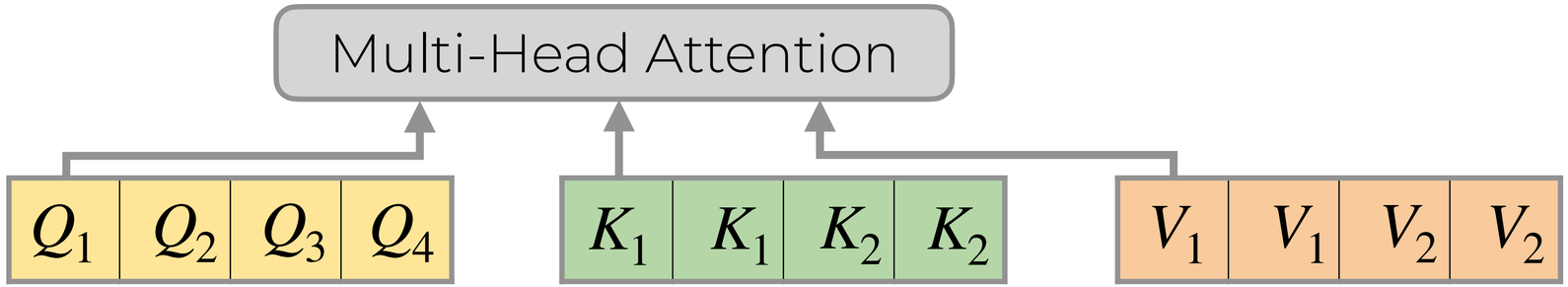
$$Q = XW_Q \qquad [K_1, K_2] = XW_K \qquad [V_1, V_2] = XW_V$$

$W_Q \in R^{d \times (d_q \times h)}$ $W_K \in R^{d \times (d_k \times n_g)}$ $W_V \in R^{d \times (d_v \times n_g)}$ $W_O \in R^{(d_o \times h) \times d}$

Query Key Value Output



Revisit GQA

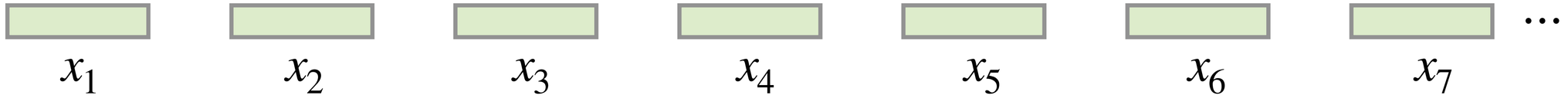


$$[V_1, V_1, V_2, V_2] = [V_1, V_2] \begin{bmatrix} I_{d_v} & I_{d_v} & 0 & 0 \\ 0 & 0 & I_{d_v} & I_{d_v} \end{bmatrix} = [V_1, V_2]\, W_V^{\uparrow}$$

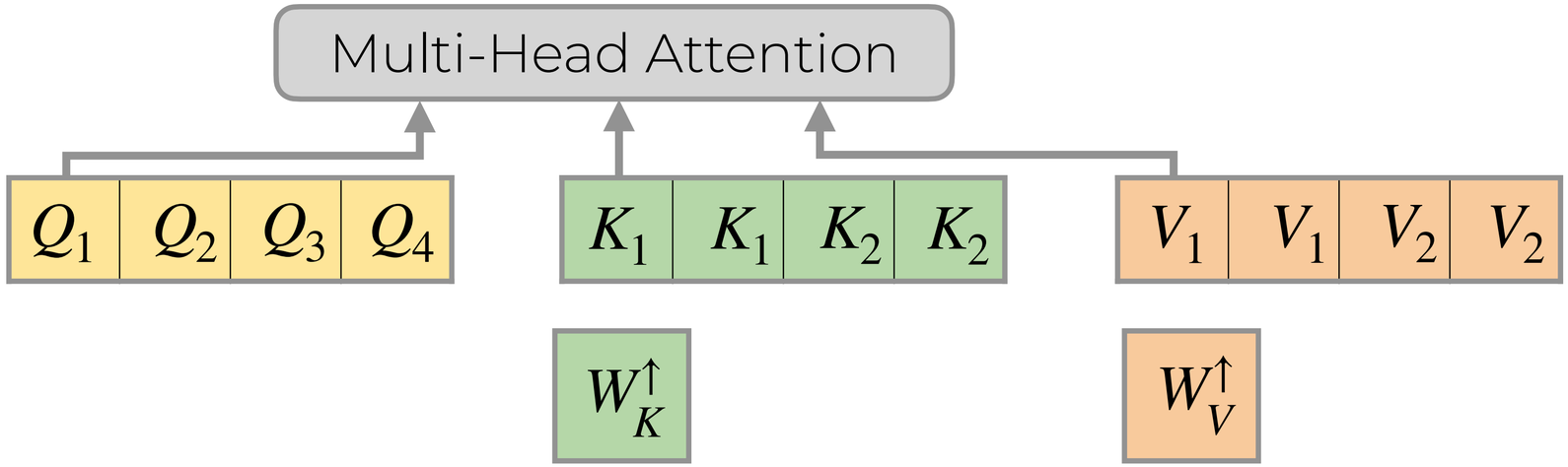
$$Q = XW_Q \qquad [K_1, K_2] = XW_K \qquad [V_1, V_2] = XW_V$$

$W_Q \in R^{d \times (d_q \times h)}$ $W_K \in R^{d \times (d_k \times n_g)}$ $W_V \in R^{d \times (d_v \times n_g)}$ $W_O \in R^{(d_o \times h) \times d}$

Query Key Value Output



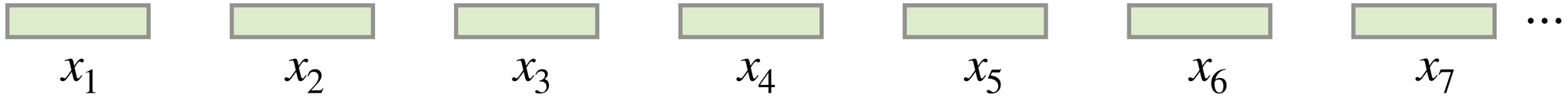
Revisit GQA



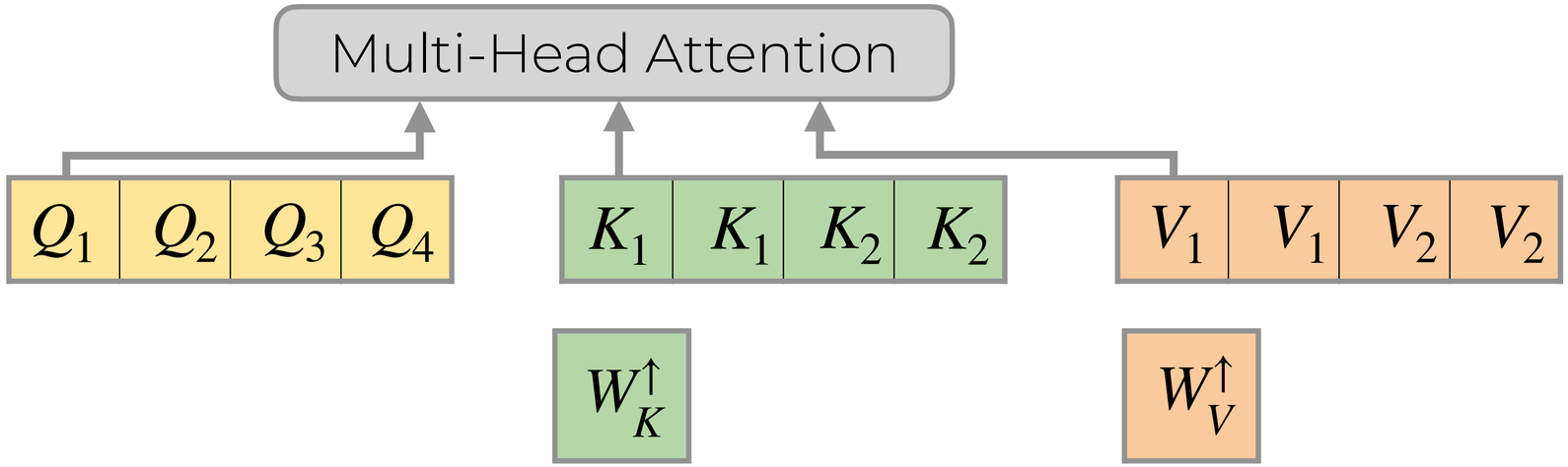
$$Q = XW_Q \qquad [K_1, K_2] = XW_K \qquad [V_1, V_2] = XW_V$$

$W_Q \in R^{d \times (d_q \times h)}$ $W_K \in R^{d \times (d_k \times n_g)}$ $W_V \in R^{d \times (d_v \times n_g)}$ $W_O \in R^{(d_o \times h) \times d}$

Query Key Value Output



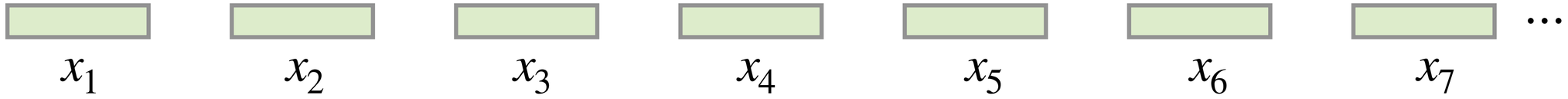
Revisit GQA



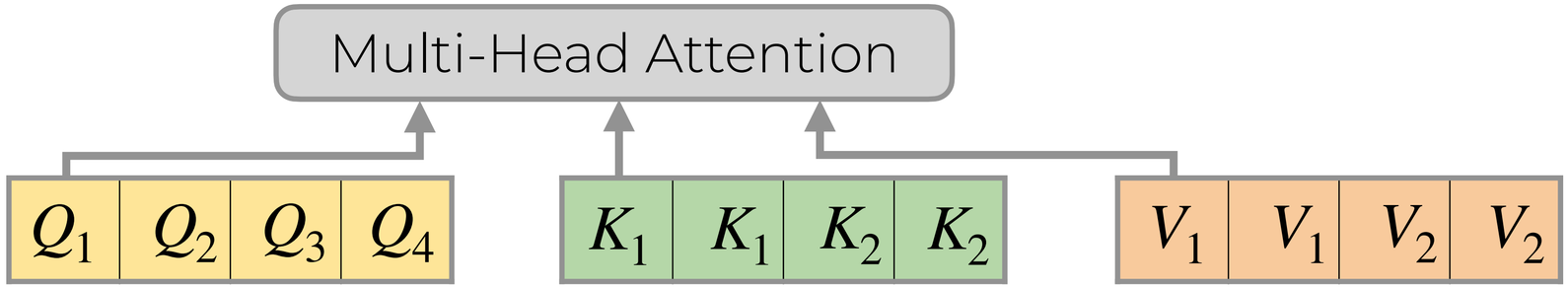
$$Q = XW_Q \qquad [K_1, K_2, V_1, V_2] = X[W_K, W_V] = XW_{KV}^{\downarrow}$$

$W_Q \in R^{d \times (d_q \times h)}$ $W_K \in R^{d \times (d_k \times n_g)}$ $W_V \in R^{d \times (d_v \times n_g)}$ $W_O \in R^{(d_o \times h) \times d}$

Query Key Value Output



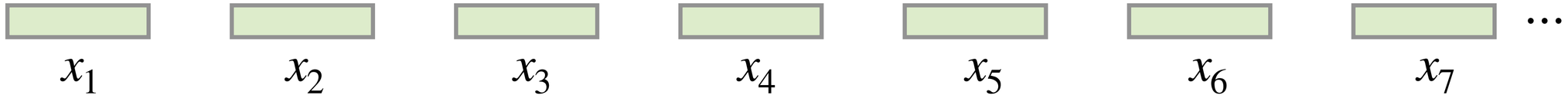
Revisit GQA



$$Q = XW_Q \qquad [K_1, K_2, V_1, V_2] = XW_{KV}^{\downarrow}$$

$W_Q \in R^{d \times (d_q \times h)}$ $W_{KV}^{\downarrow} \in R^{d \times (d_k n_g + d_v n_g)}$ $W_O \in R^{(d_o \times h) \times d}$

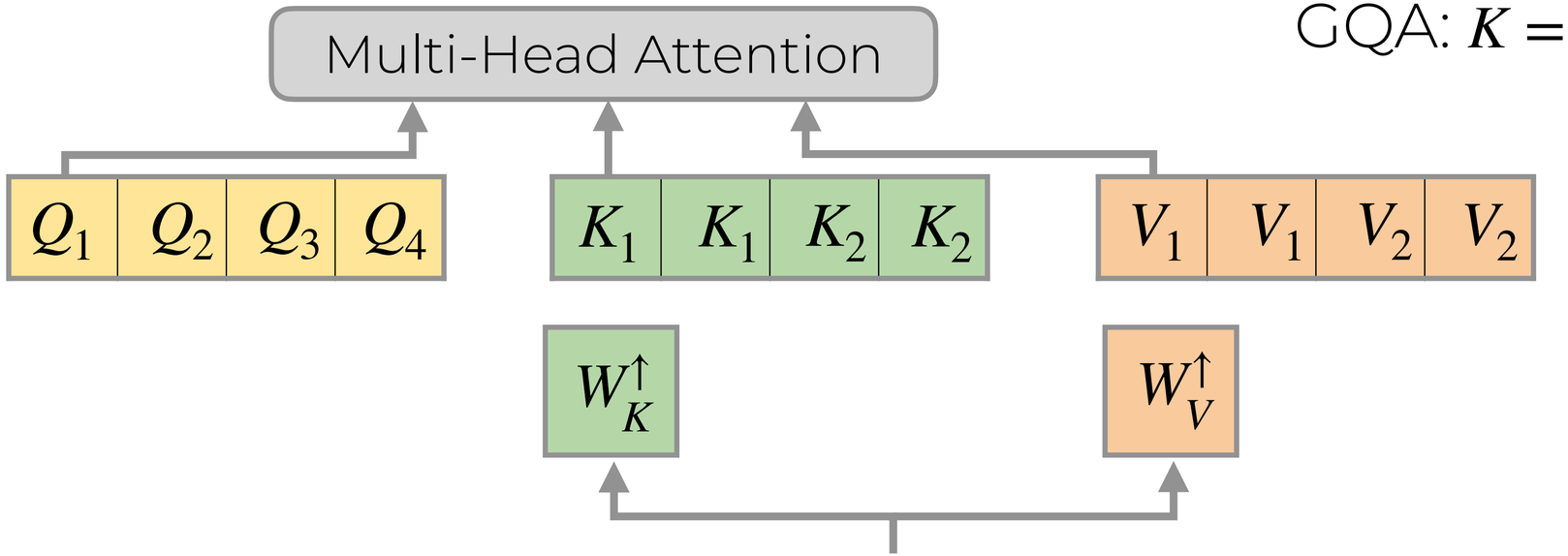
Query Key / Value Output



Revisit GQA

MHA: $K = XW_K;$ $V = XW_V$

GQA: $K = XW_{KV}^{\downarrow}W_K^{\uparrow};$ $V = XW_{KV}^{\downarrow}W_V^{\uparrow}$



$Q = XW_Q$

$[K_1, K_2, V_1, V_2] = XW_{KV}^{\downarrow}$

$W_Q \in R^{d \times (d_q \times h)}$

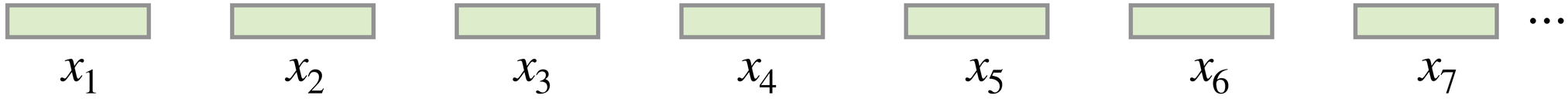
$W_{KV}^{\downarrow} \in R^{d \times (d_k n_g + d_v n_g)}$

$W_O \in R^{(d_o \times h) \times d}$

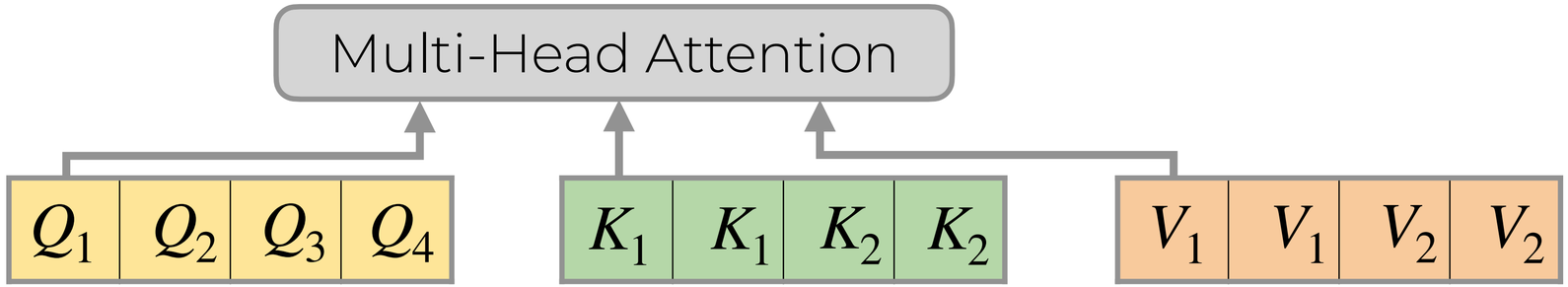
Query

Key / Value

Output



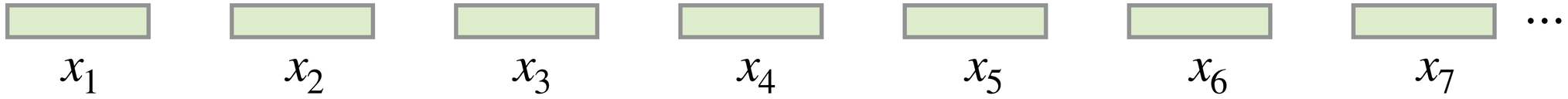
Multi-Head Latent Attention (MLA) [DeepSeek-AI 2024]



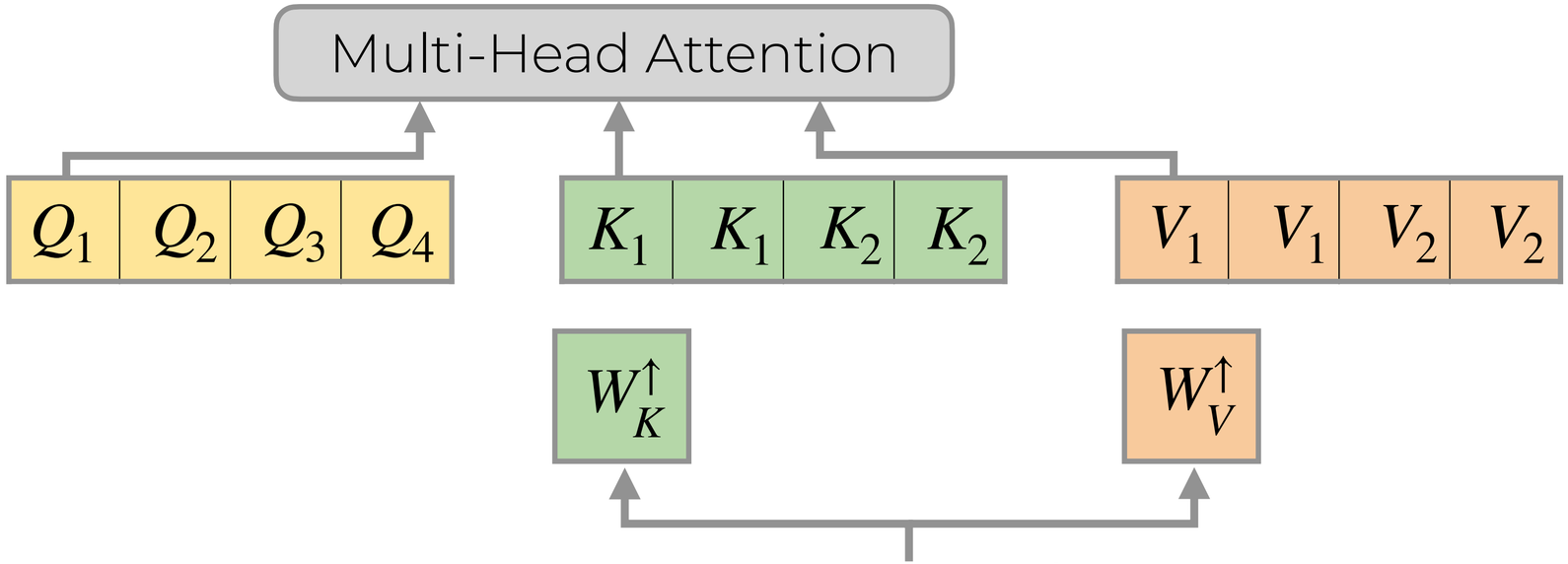
$$Q = XW_Q \qquad [K_1, K_2, V_1, V_2] = XW_{KV}^{\downarrow}$$

$W_Q \in R^{d \times (d_q \times h)}$ $W_{KV}^{\downarrow} \in R^{d \times (d_k n_g + d_v n_g)}$ $W_O \in R^{(d_o \times h) \times d}$

Query Key / Value Output



Multi-Head Latent Attention (MLA) [DeepSeek-AI 2024]



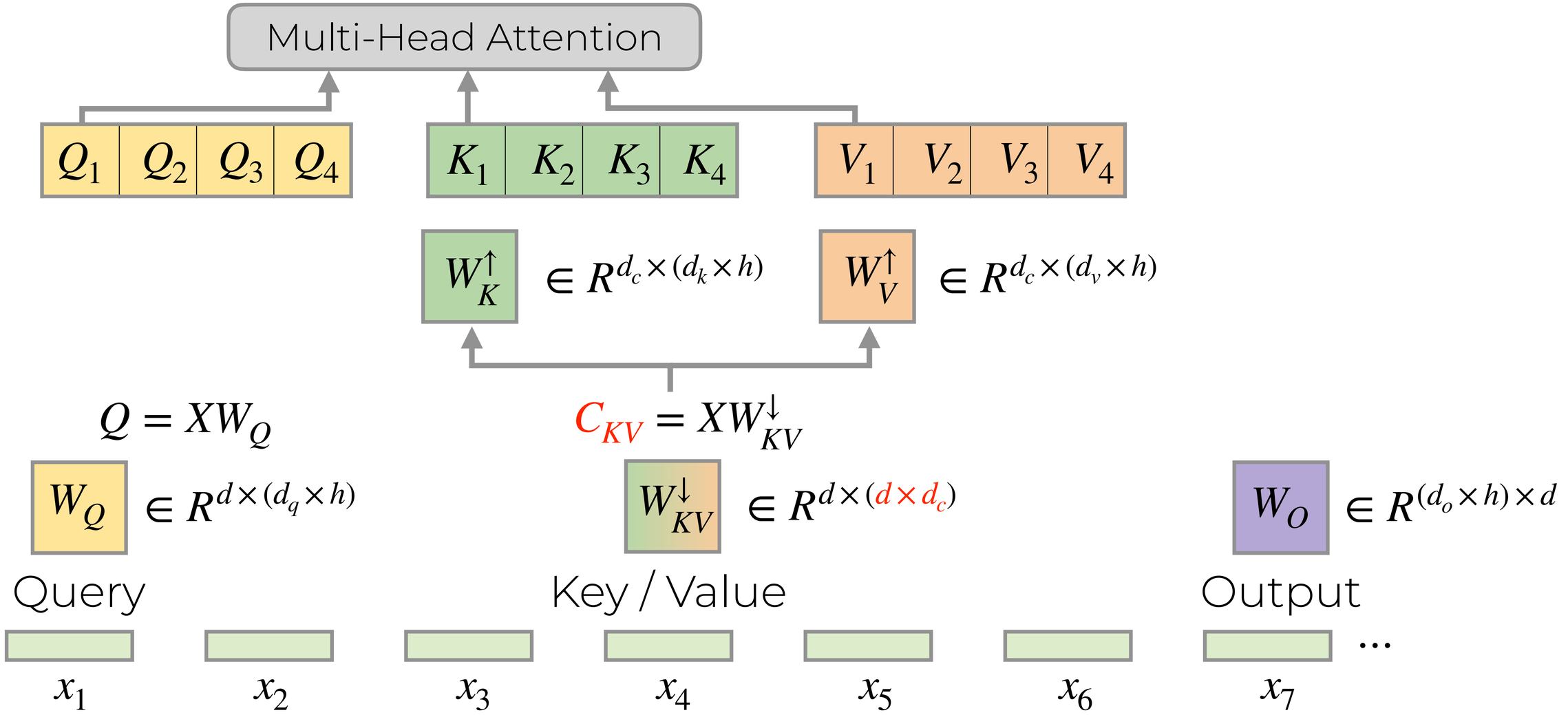
$$Q = XW_Q \qquad C_{KV} = XW_{KV}^{\downarrow}$$ (compress to cached latent code)

$W_Q \in R^{d \times (d_q \times h)}$ $W_{KV}^{\downarrow} \in R^{d \times (d \times d_c)}$ $W_O \in R^{(d_o \times h) \times d}$

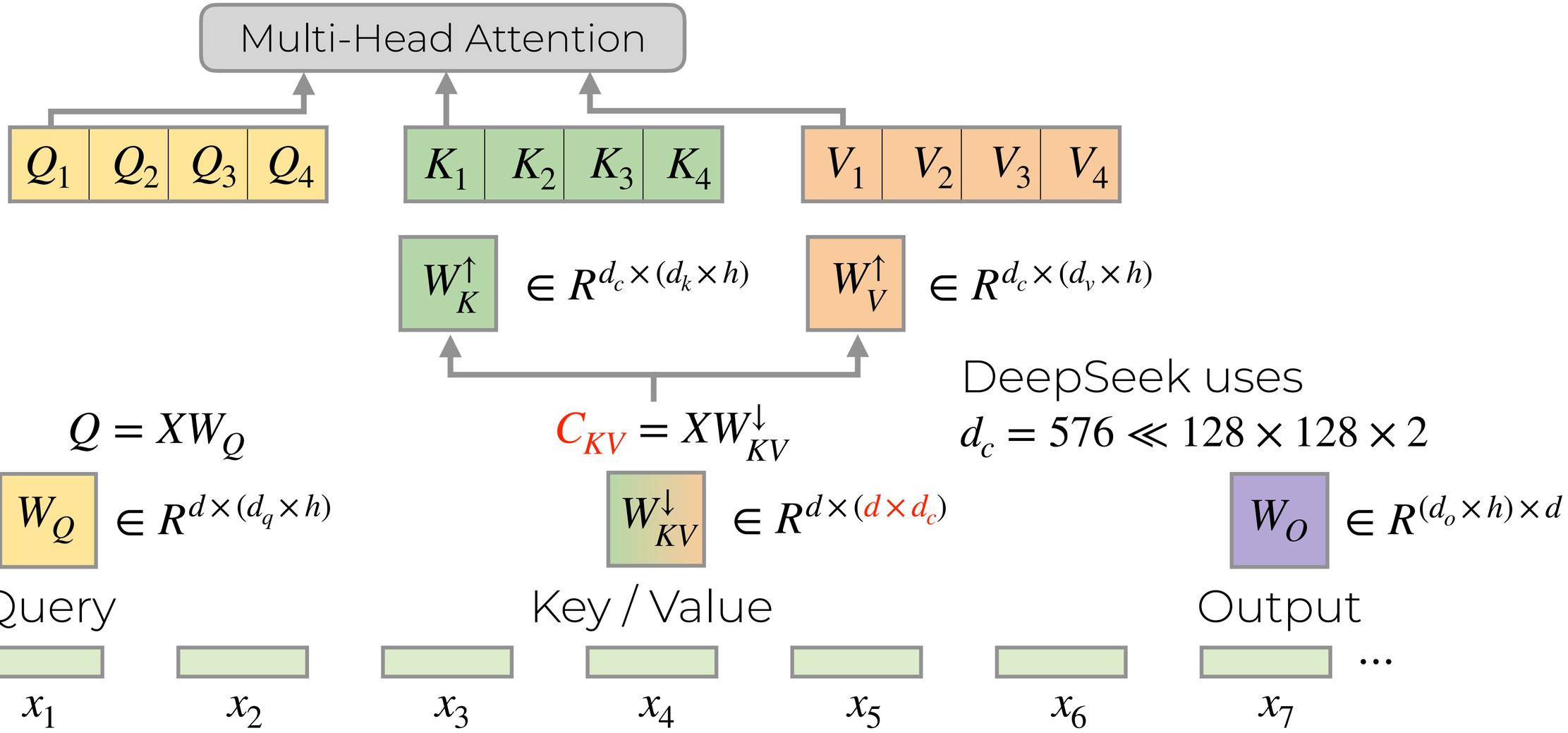
Query Key / Value Output

x_1 x_2 x_3 x_4 x_5 x_6 x_7 ...

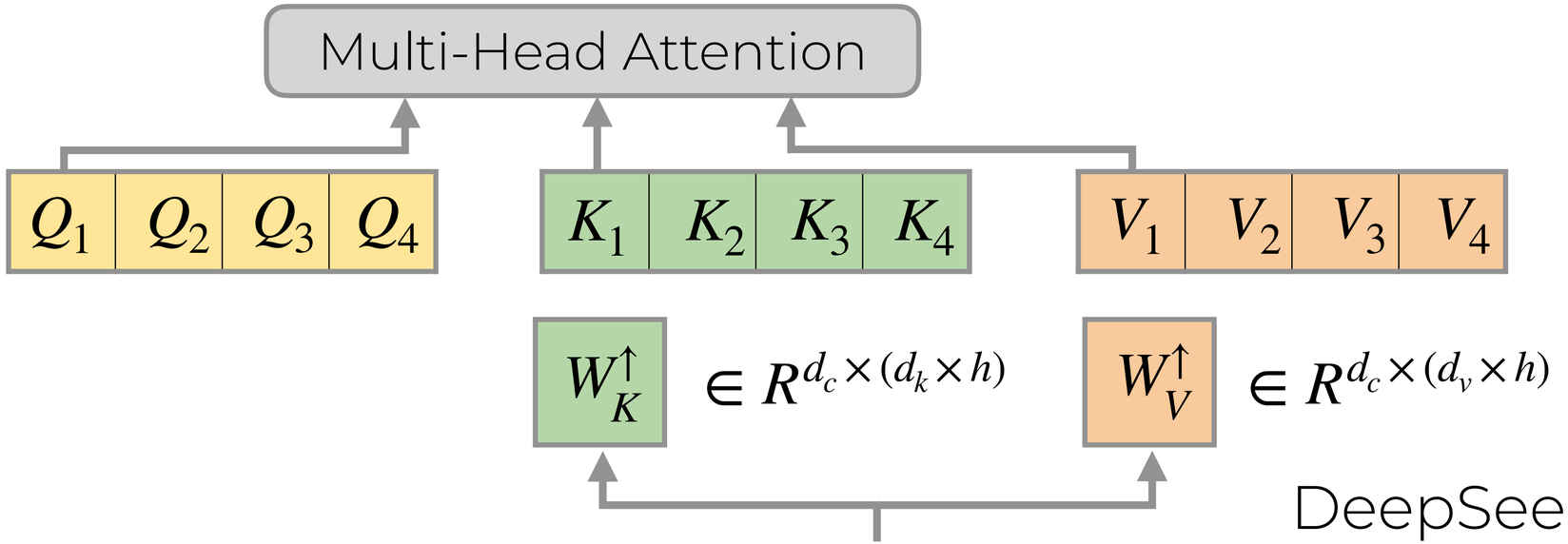
Multi-Head Latent Attention (MLA) [DeepSeek-AI 2024]



Multi-Head Latent Attention (MLA) [DeepSeek-AI 2024]



Multi-Head Latent Attention (MLA) [DeepSeek-AI 2024]



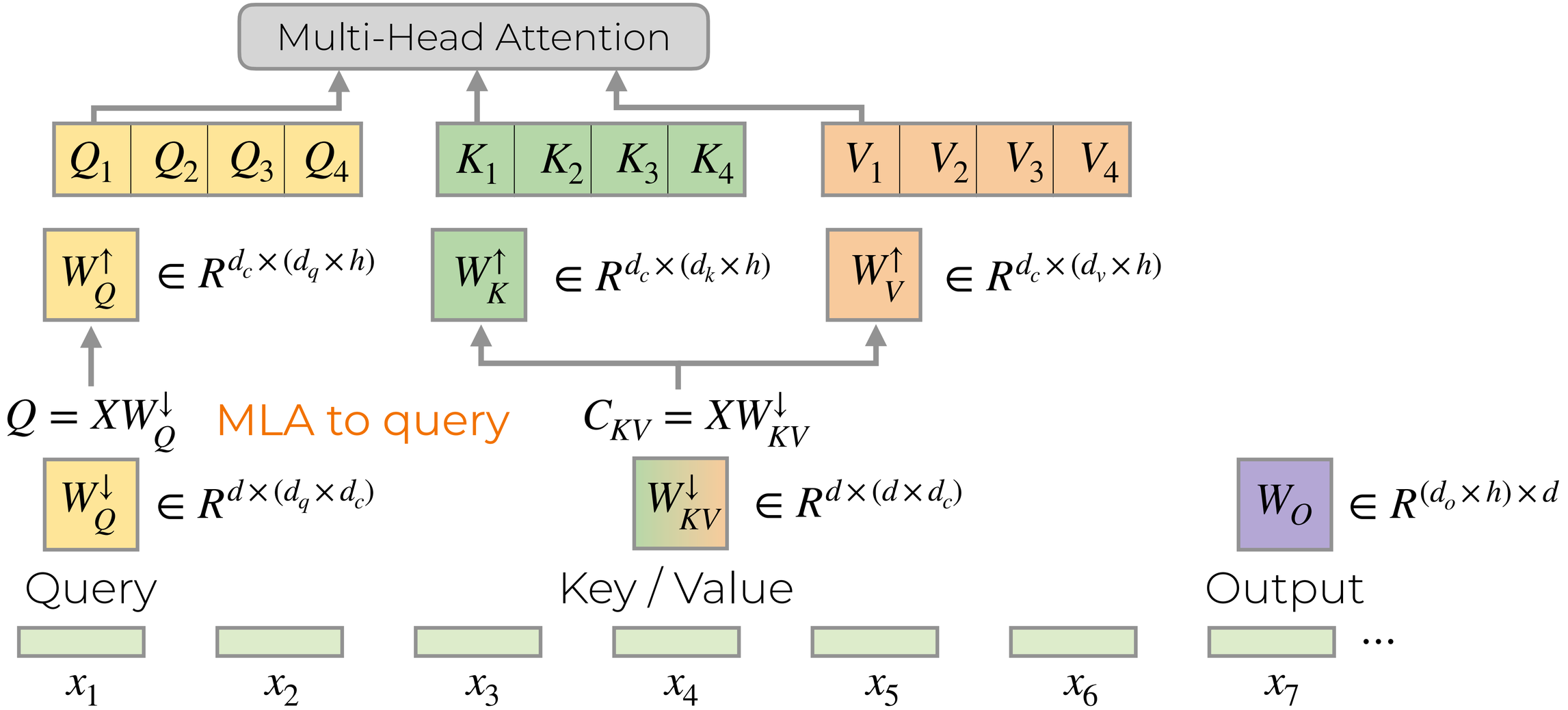
KV-cache per token

- MHA = 4MB
- MQA = 31KB
- GQA = 500KB
- MLA = 70KB

DeepSeek uses

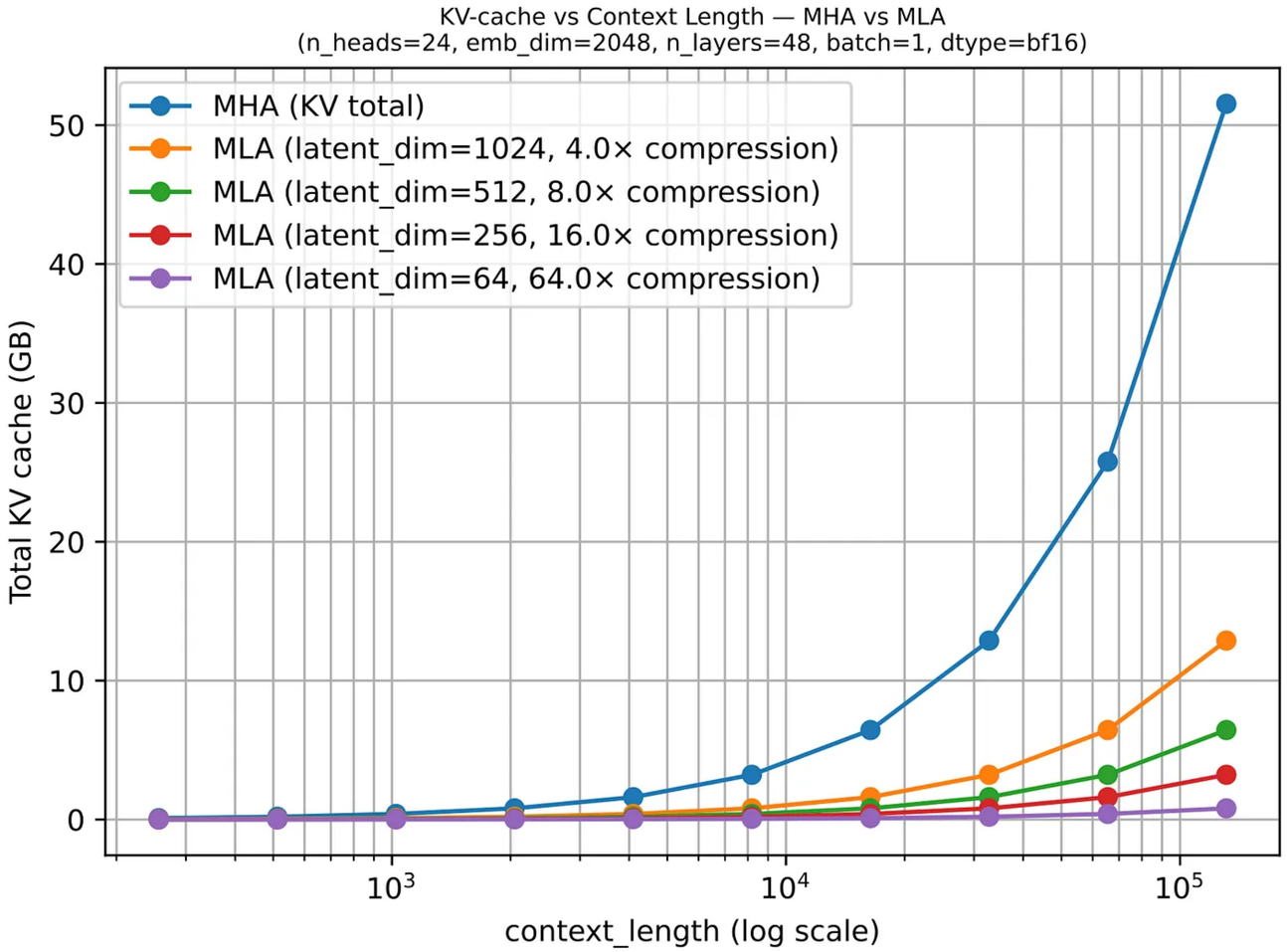
$$d_c = 576 \ll 128 \times 128 \times 2$$

Multi-Head Latent Attention (MLA) [DeepSeek-AI 2024]



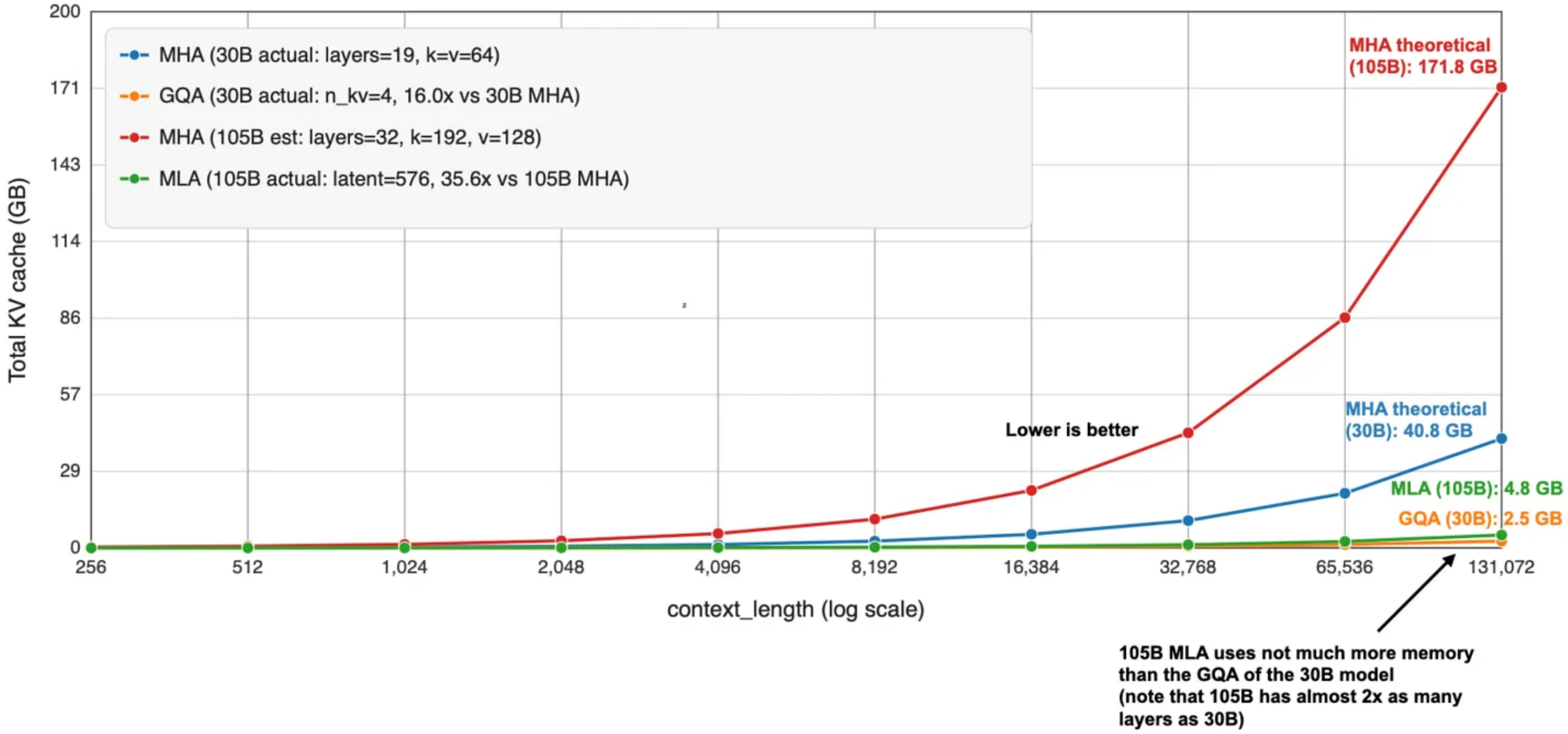
https://github.com/rasbt/LLMs-from-scratch/tree/main/ch04/05_mla

MHA vs. MLA: Memory Usage



https://magazine.sebastianraschka.com/p/visual-attention-variants

MHA vs. MQA vs. GQA vs. MLA



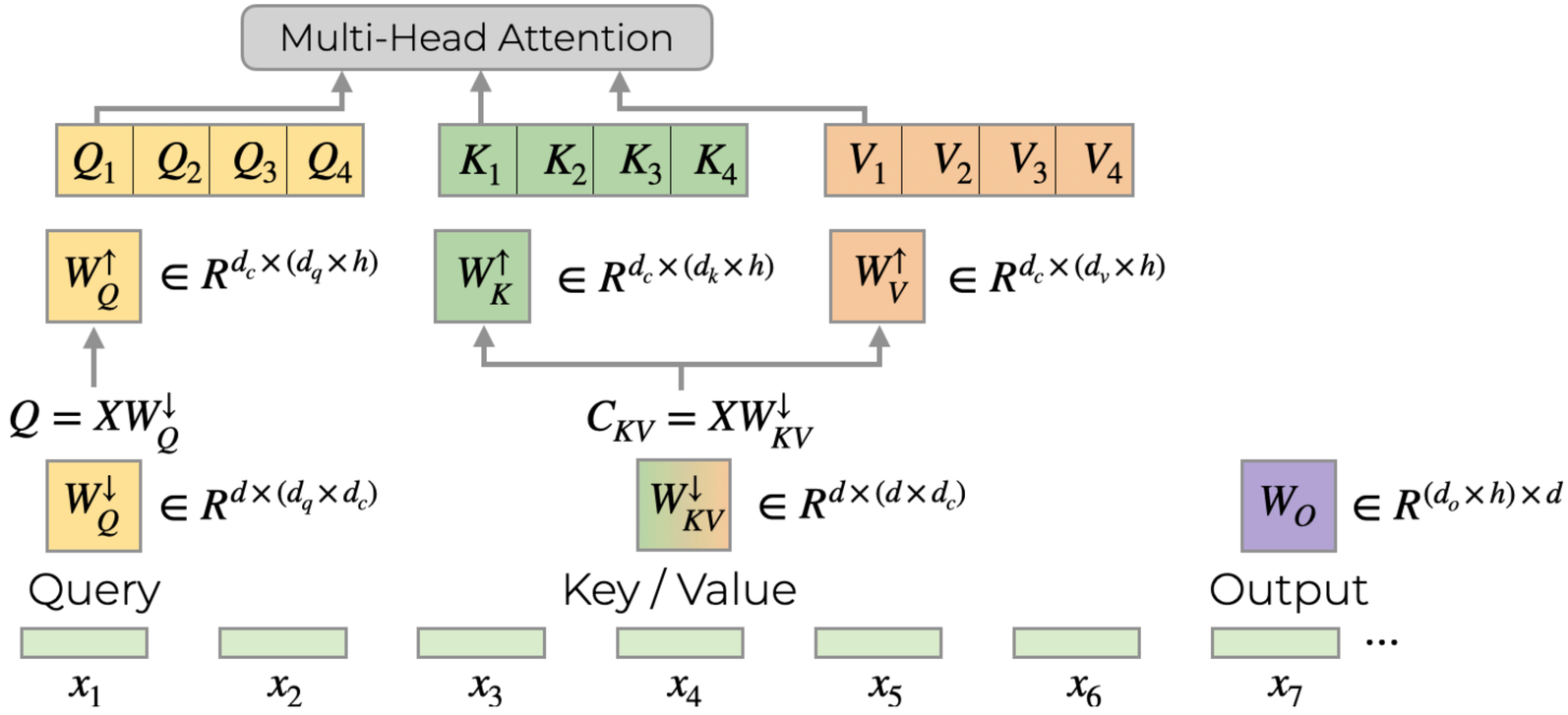
MHA vs. MQA vs. GQA vs. MLA [DeepSeek-AI 2024]

Attention Mechanism	KV Cache per Token (# Element)	Capability
Multi-Head Attention (MHA)	$2n_h d_h l$	Strong
Grouped-Query Attention (GQA)	$2n_g d_h l$	Moderate
Multi-Query Attention (MQA)	$2d_h l$	Weak
MLA (Ours)	$(d_c + d_h^R)l \approx \frac{9}{2} d_h l$	Stronger

Benchmark (Metric)	# Shots	Small MoE w/ MHA	Small MoE w/ MLA	Large MoE w/ MHA	Large MoE w/ MLA
# Activated Params	-	2.5B	2.4B	25.0B	21.5B
# Total Params	-	15.8B	15.7B	250.8B	247.4B
KV Cache per Token (# Element)	-	110.6K	15.6K	860.2K	34.6K
BBH (EM)	3-shot	37.9	**39.0**	46.6	**50.7**
MMLU (Acc.)	5-shot	48.7	**50.0**	57.5	**59.0**
C-Eval (Acc.)	5-shot	**51.6**	50.9	57.9	**59.2**
CMMLU (Acc.)	5-shot	52.3	**53.4**	60.7	**62.5**

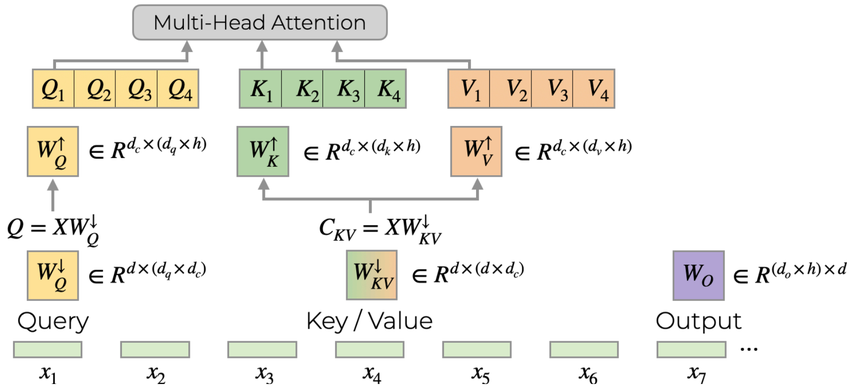
MLA at Inference

- It seems that we should compute down/up projection matrices
  - Good for memory reduction, but extra computation at inference
- Actually, we can avoid extra computation with some trick!



Weight Absorption in MLA

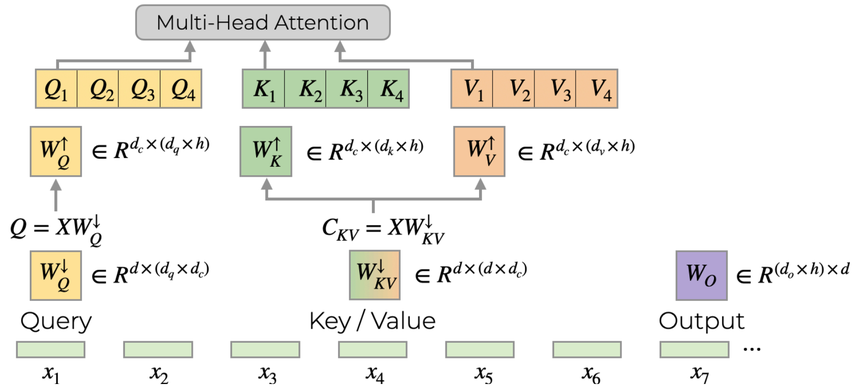
$$\Delta X = OW_O = [O_1, O_2, \ldots, O_h]\, W_O$$



Weight Absorption in MLA

$$\Delta X = OW_O = [O_1, O_2, \ldots, O_h]\, W_O$$

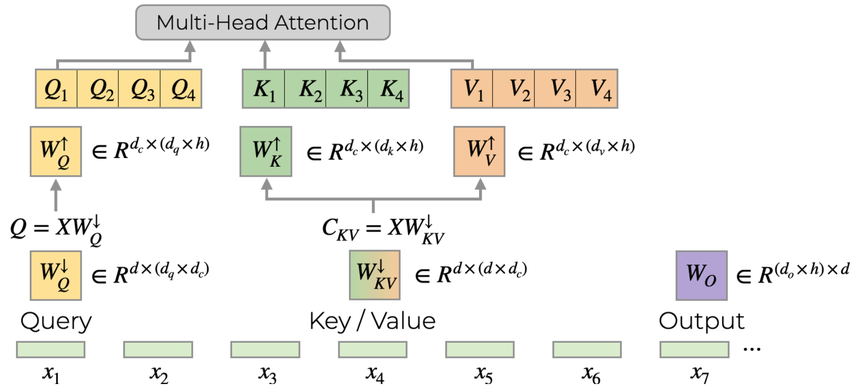
$$\Delta X = \left[\cdots, Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M \right) V_i, \cdots \right] W_O \quad Q_i = X W_Q^{\downarrow} W_Q^{\uparrow,i} \quad K_i = C_{KV} W_K^{\uparrow,i}$$



Weight Absorption in MLA

$$\Delta X = OW_O = [O_1, O_2, \ldots, O_h]\, W_O$$

$$\Delta X = \left[\cdots, Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M \right) V_i, \cdots \right] W_O$$



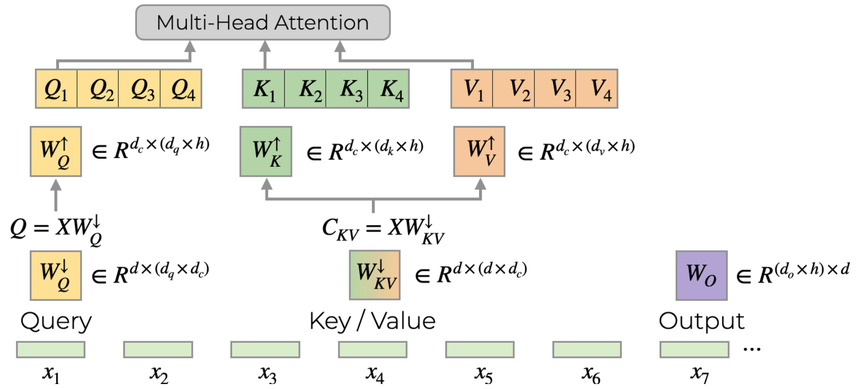
$$Q_i = XW_Q^{\downarrow} W_Q^{\uparrow,i} \qquad K_i = C_{KV} W_K^{\uparrow,i}$$

cached KV latent

Weight Absorption in MLA

$$\Delta X = OW_O = [O_1, O_2, \ldots, O_h]\, W_O$$

$$\Delta X = \left[\cdots, Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M \right) V_i, \cdots \right] W_O$$



$$Q_i = XW_Q^{\downarrow} W_Q^{\uparrow,i} \qquad K_i = C_{KV} W_K^{\uparrow,i}$$

cached KV latent

Weight Absorption in MLA

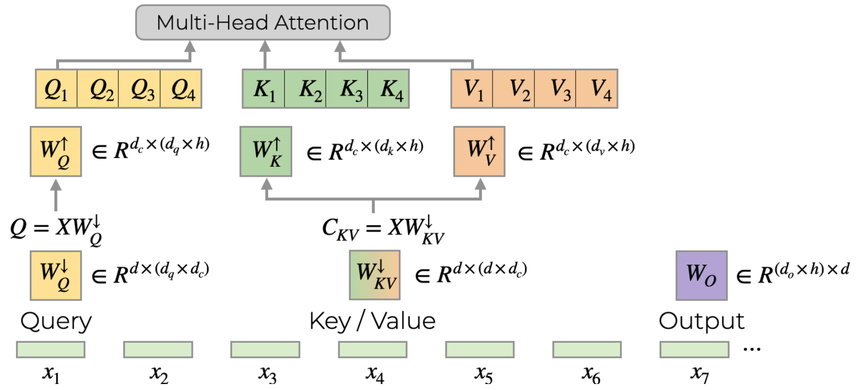
$$\Delta X = OW_O = [O_1, O_2, \ldots, O_h]\, W_O$$

$$\Delta X = \left[\cdots, Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M \right) V_i, \cdots \right] W_O$$

$$Q_i = X W_Q^{\downarrow} W_Q^{\uparrow,i} \qquad K_i = C_{KV} W_K^{\uparrow,i}$$

cached KV latent

$$Softmax\left(\frac{X\, W_Q^{\downarrow}\, W_Q^{\uparrow,i}\, W_K^{\uparrow,i\top}\, C_{KV}^{\top}}{\sqrt{d_k}} + M \right) V_i$$

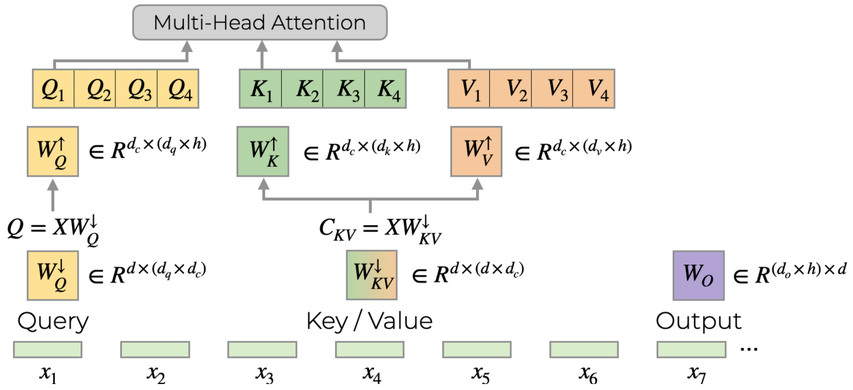


Weight Absorption in MLA

$$\Delta X = OW_O = [O_1, O_2, \ldots, O_h]\, W_O$$

$$\Delta X = \left[\cdots, Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M \right) V_i, \cdots \right] W_O$$

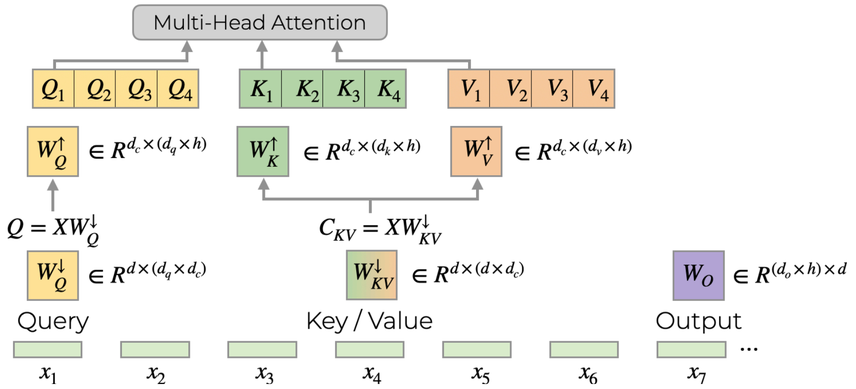
$$Q_i = X W_Q^{\downarrow} W_Q^{\uparrow,i} \qquad K_i = C_{KV} W_K^{\uparrow,i}$$



cached KV latent

$$Softmax\left(\frac{X \boxed{W_Q^{\downarrow}\ W_Q^{\uparrow,i}\ W_K^{\uparrow,i\top}} C_{KV}^{\top}}{\sqrt{d_k}} + M \right) V_i$$

Weight Absorption in MLA



$$\Delta X = OW_O = [O_1, O_2, \ldots, O_h]\, W_O$$

$$\Delta X = \left[\cdots, Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M \right) V_i, \cdots \right] W_O \qquad Q_i = XW_Q^{\downarrow} W_Q^{\uparrow,i} \qquad K_i = C_{KV} W_K^{\uparrow,i}$$

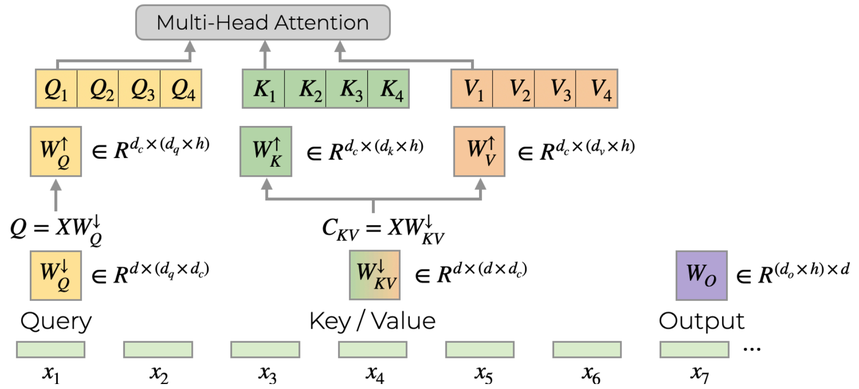
cached KV latent

$$Softmax\left(\frac{X \left[W_Q^{\downarrow}\ W_Q^{\uparrow,i}\ W_K^{\uparrow,i\top} \right] C_{KV}^{\top}}{\sqrt{d_k}} + M \right) V_i$$

$$= Softmax\left(\frac{X\ W_{QK}\ C_{KV}^{\top}}{\sqrt{d_k}} + M \right) V_i$$

Weight Absorption in MLA

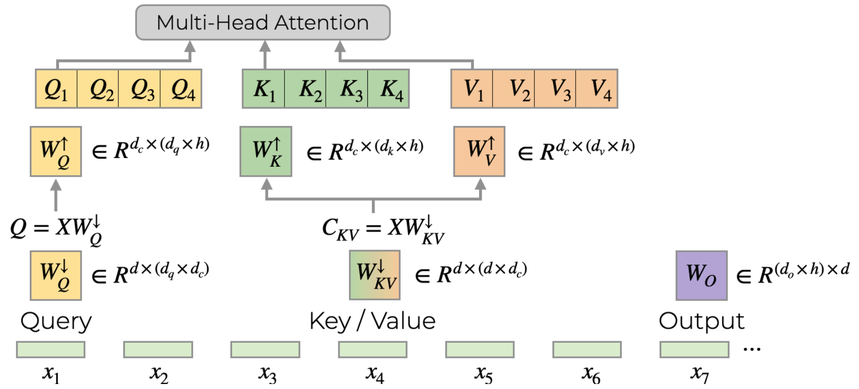
$$\Delta X = [\cdots, A_i V_i, \cdots] W_O$$



Weight Absorption in MLA

$$\Delta X = [\cdots, A_i V_i, \cdots] W_O$$

$$= [A_1 V_1,\ A_2 V_2,\ A_3 V_3, \cdots] W_O \qquad V_i = C_{KV} W_V^{\uparrow, i}$$



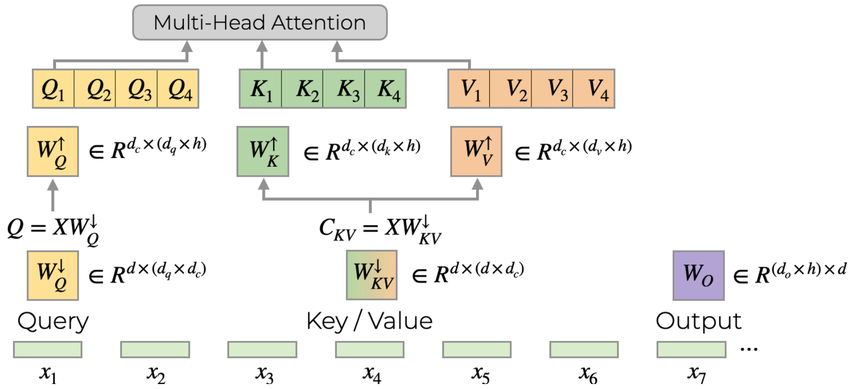
Weight Absorption in MLA

$$\Delta X = [\cdots, A_i V_i, \cdots] W_O$$

$$= [A_1 V_1,\ A_2 V_2,\ A_3 V_3, \cdots] W_O$$

cached KV latent

$$V_i = C_{KV} W_V^{\uparrow, i}$$



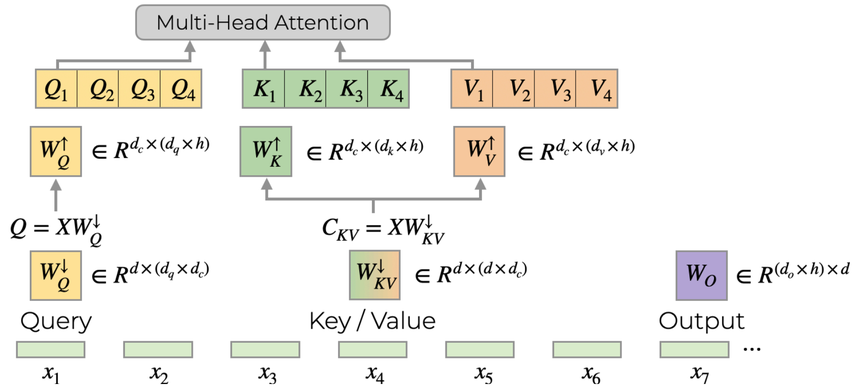
Weight Absorption in MLA

$$\Delta X = [\cdots, A_i V_i, \cdots] W_O$$

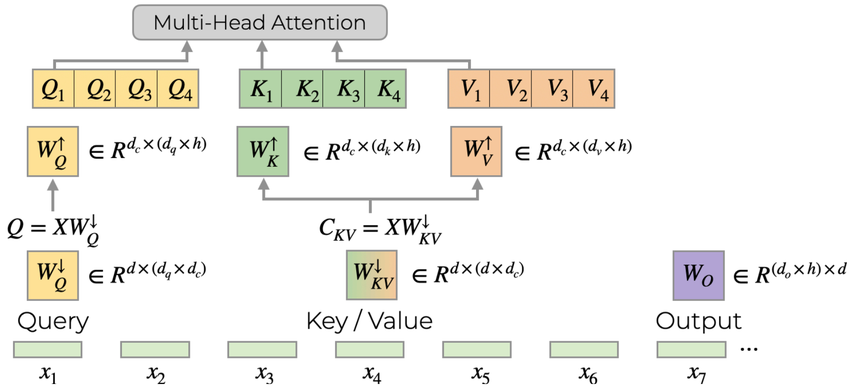
$$= [A_1 V_1,\ A_2 V_2,\ A_3 V_3, \cdots] W_O \qquad V_i = C_{KV} W_V^{\uparrow, i}$$

$$= [A_1 C_{KV} W_V^{\uparrow,1},\ A_2 C_{KV} W_V^{\uparrow,2},\ A_3 C_{KV} W_V^{\uparrow,3}, \cdots] W_O$$

cached KV latent



Weight Absorption in MLA



cached KV latent

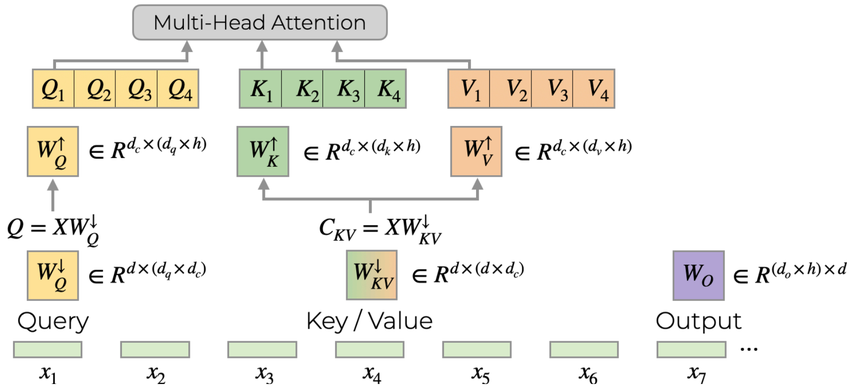
$$\Delta X = [\cdots, A_i V_i, \cdots] W_O$$

$$= [A_1 V_1, A_2 V_2, A_3 V_3, \cdots] W_O \qquad V_i = C_{KV} W_V^{\uparrow,i}$$

$$= [A_1 C_{KV} W_V^{\uparrow,1}, A_2 C_{KV} W_V^{\uparrow,2}, A_3 C_{KV} W_V^{\uparrow,3}, \cdots] W_O$$

$$= [A_1 C_{KV}, A_2 C_{KV}, A_3 C_{KV}, \cdots] \begin{bmatrix} W_V^{\uparrow,1} & 0 & 0 & \\ 0 & W_V^{\uparrow,i} & 0 & \cdots \\ 0 & 0 & W_V^{\uparrow,3} & \\ & \cdots & & \end{bmatrix} W_O$$

Weight Absorption in MLA



cached KV latent

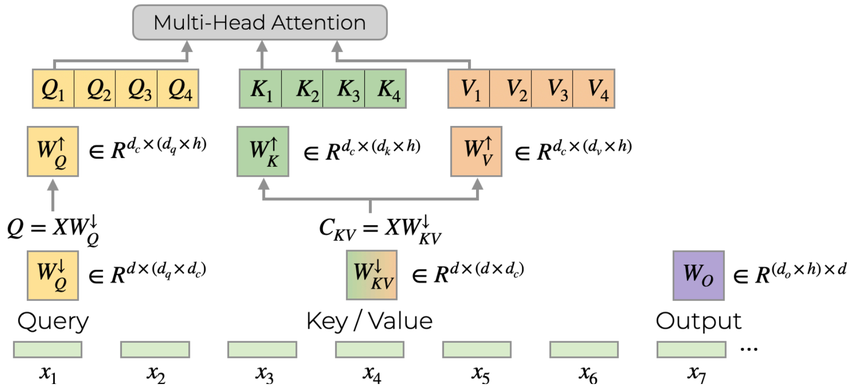
$$\Delta X = [\cdots, A_i V_i, \cdots] W_O$$

$$= [A_1 V_1, A_2 V_2, A_3 V_3, \cdots] W_O \qquad V_i = C_{KV} W_V^{\uparrow,i}$$

$$= [A_1 C_{KV} W_V^{\uparrow,1}, A_2 C_{KV} W_V^{\uparrow,2}, A_3 C_{KV} W_V^{\uparrow,3}, \cdots] W_O$$

$$= [A_1 C_{KV}, A_2 C_{KV}, A_3 C_{KV}, \cdots] \begin{bmatrix} W_V^{\uparrow,1} & 0 & 0 & \\ 0 & W_V^{\uparrow,i} & 0 & \cdots \\ 0 & 0 & W_V^{\uparrow,3} & \\ & \cdots & & \end{bmatrix} W_O$$

Weight Absorption in MLA



cached KV latent

$$\Delta X = [\cdots, A_i V_i, \cdots] W_O$$

$$= [A_1 V_1, A_2 V_2, A_3 V_3, \cdots] W_O \qquad V_i = C_{KV} W_V^{\uparrow,i}$$

$$= [A_1 C_{KV} W_V^{\uparrow,1}, A_2 C_{KV} W_V^{\uparrow,2}, A_3 C_{KV} W_V^{\uparrow,3}, \cdots] W_O$$

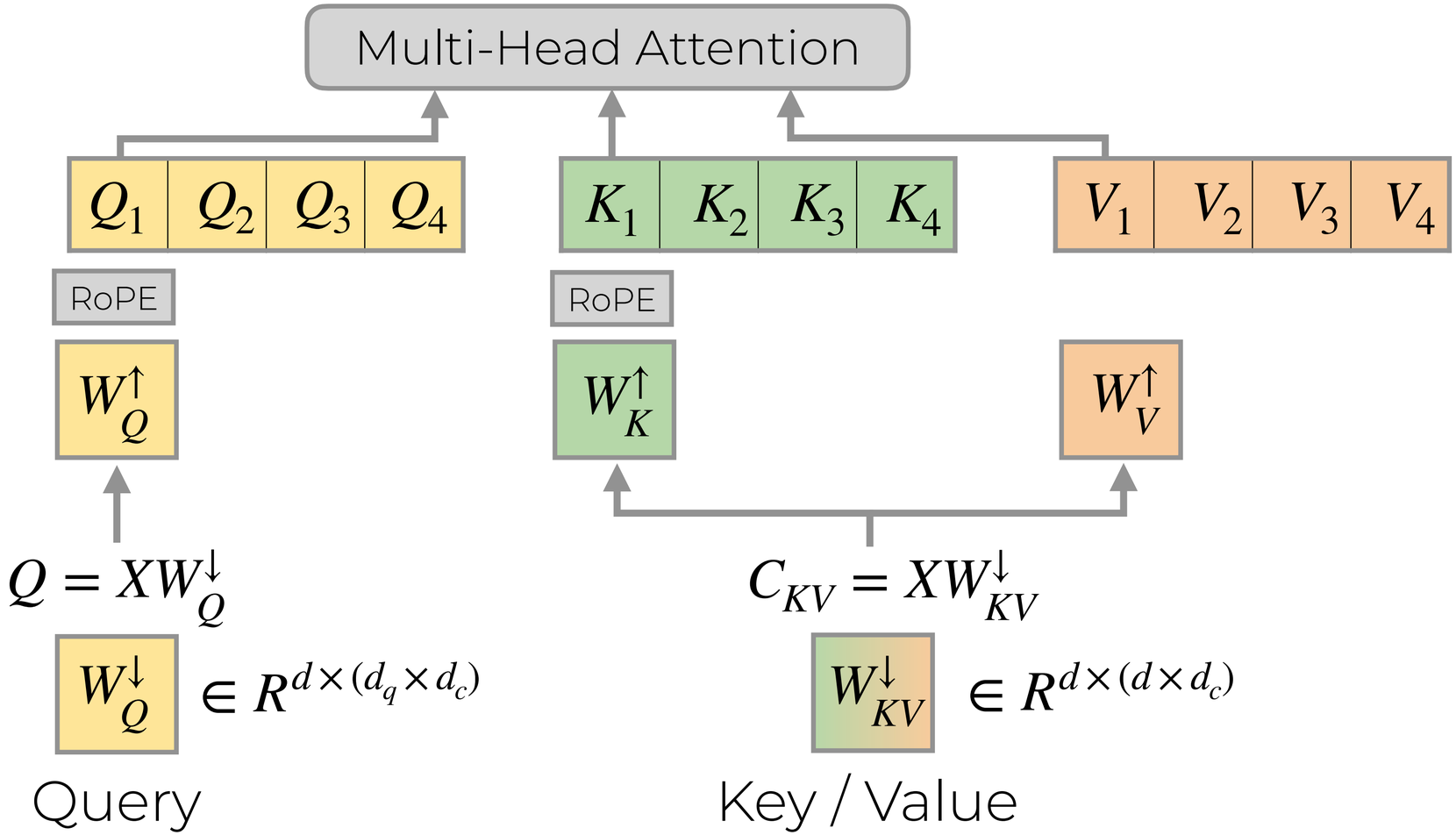
$$= [A_1 C_{KV}, A_2 C_{KV}, A_3 C_{KV}, \cdots] \begin{bmatrix} W_V^{\uparrow,1} & 0 & 0 & \\ 0 & W_V^{\uparrow,i} & 0 & \cdots \\ 0 & 0 & W_V^{\uparrow,3} & \\ & \cdots & & \end{bmatrix} W_O$$

$$= [A_1 C_{KV}, A_2 C_{KV}, A_3 C_{KV}, \cdots] W_O$$

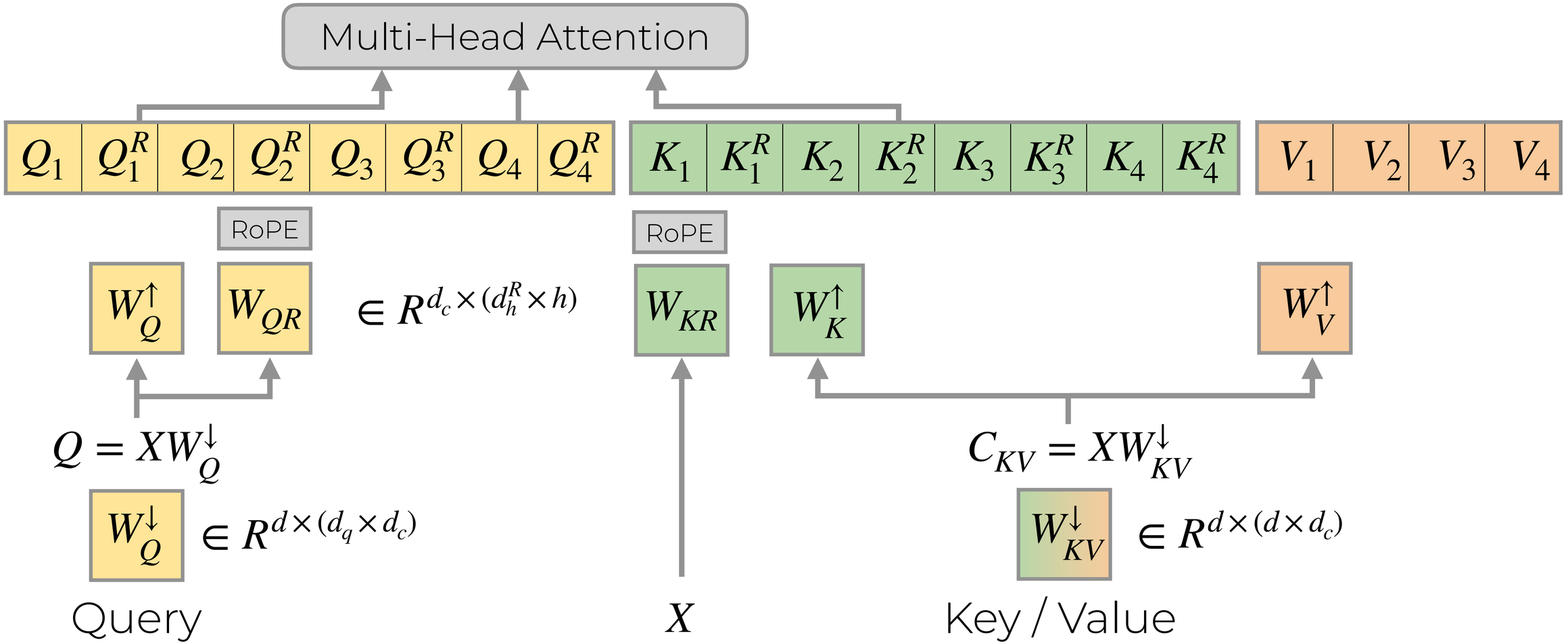
MLA with RoPE

- RoPE: $O_i = Softmax\left(\frac{RoPE(XW_Q^i)\ RoPE(W_K^i X)^\top}{\sqrt{d_k}} + M\right) V_i \qquad RoPE(X) = XR_{\theta,m}$
- MLA with RoPE: $O_i = Softmax\left(\frac{RoPE(XW_Q^{\downarrow} W_Q^{\uparrow,i})\ RoPE(C_{KV} W_K^{\uparrow,i})^\top}{\sqrt{d_k}} + M\right) V_i$
  - With RoPE, we cannot use weight absorption trick!

MLA with Decoupled RoPE



MLA with Decoupled RoPE



MLA with Decoupled RoPE

- RoPE: $O_i = Softmax\left(\frac{RoPE(XW_Q^i)\ RoPE(W_K^i X)^\top}{\sqrt{d_k}} + M\right) V_i \qquad RoPE(X) = XR_{\theta,m}$
- MLA with RoPE: $O_i = Softmax\left(\frac{RoPE(XW_Q^{\downarrow} W_Q^{\uparrow,i})\ RoPE(C_{KV} W_K^{\uparrow,i})^\top}{\sqrt{d_k}} + M\right) V_i$
- MLA with decoupled RoPE:

$$O_i = Softmax\left(\frac{XW_{QK}C_{KV}^\top + RoPE(XW_Q^{\downarrow} W_{QR})\ RoPE(XW_{KR})^\top}{\sqrt{d_k}} + M\right) V_i$$

MLA with Decoupled RoPE

- RoPE: $O_i = Softmax\left(\frac{RoPE(XW_Q^i)\ RoPE(W_K^i X)^\top}{\sqrt{d_k}} + M\right) V_i \qquad RoPE(X) = XR_{\theta,m}$

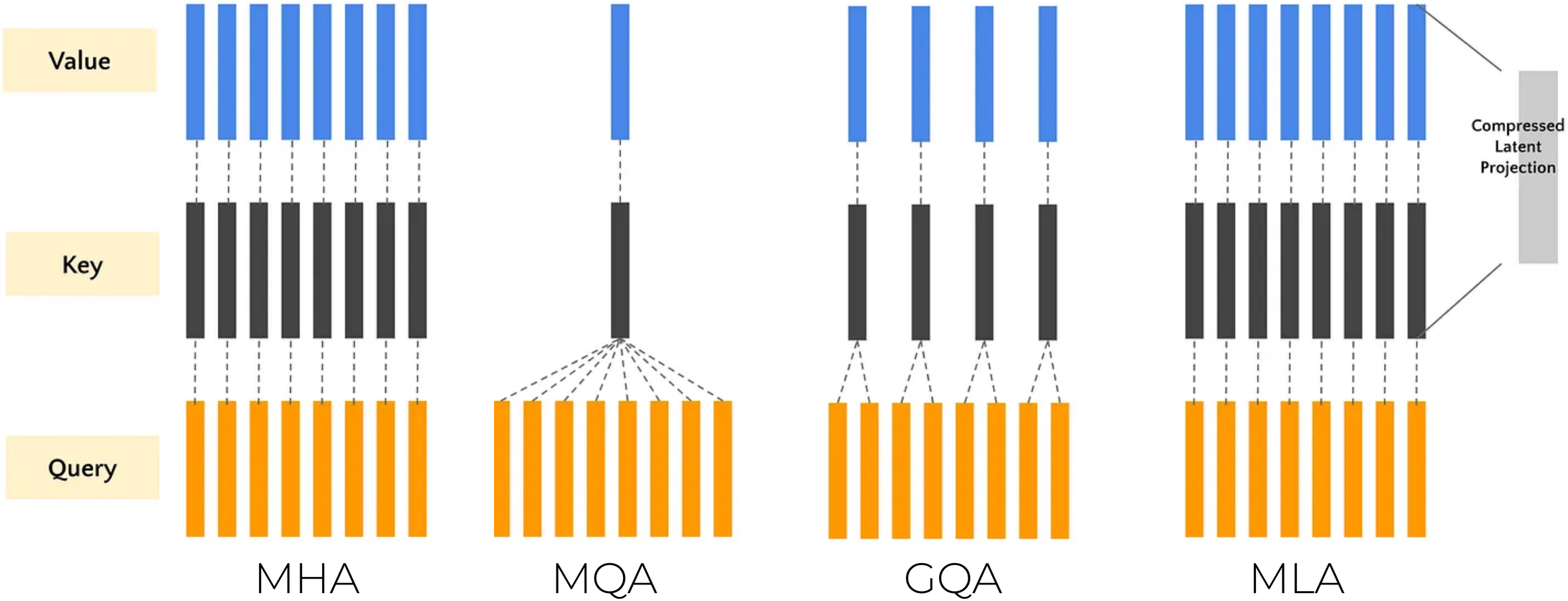
- MLA with RoPE: $O_i = Softmax\left(\frac{RoPE(XW_Q^{\downarrow} W_Q^{\uparrow,i})\ RoPE(C_{KV} W_K^{\uparrow,i})^\top}{\sqrt{d_k}} + M\right) V_i$

- MLA with decoupled RoPE:

weight absorption RoPE key cache

$$O_i = Softmax\left(\frac{XW_{QK}C_{KV}^\top + RoPE(XW_Q^{\downarrow} W_{QR})\ RoPE(XW_{KR})^\top}{\sqrt{d_k}} + M\right) V_i$$

Summary


Value
Key
Query
Compressed
Latent
Projection
MHA
MQA
GQA
MLA