

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

CSC5051/MDS5110/CSC6052 : Natural Language Processing

Lecture 11: Efficiency in LLMs

Fall 2026

Benyou Wang
School of Data Science

Contents

Foundations — *why it matters*

1 Why Efficiency?
Scaling laws · the compute gap · a map of efficiency methods

Smaller and sparser models — *change the model*

2 Efficient Architectures
Sparse attention · GQA / MLA · Mixture of Experts

3 Model Compression
Quantization · pruning · knowledge distillation

Faster systems — *change how we run it*

4 Efficient Training
LoRA and PEFT · modular adapters · data/model parallelism · ZeRO

5 Efficient Inference
KV cache · PagedAttention · speculative decoding · FlashAttention · vLLM

Looking ahead

6 Outlook
2026 trends · discussion

One question for the whole lecture

Where does the time and memory go?

FLOPs
how much arithmetic

Bytes moved
memory traffic — usually the real bottleneck

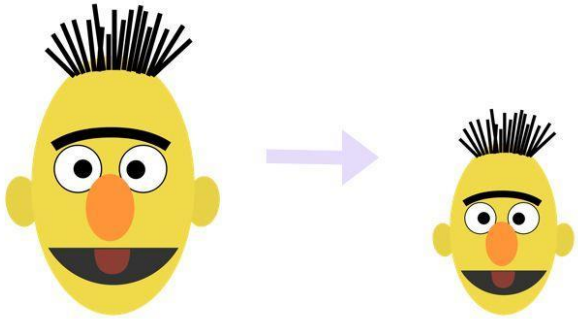
Bytes stored
weights, activations, optimizer states, KV cache

Communication
between GPUs and nodes

1 Why Efficiency?

Model size, data and usage grow much faster than hardware. Efficiency decides who can train, serve and afford LLMs.

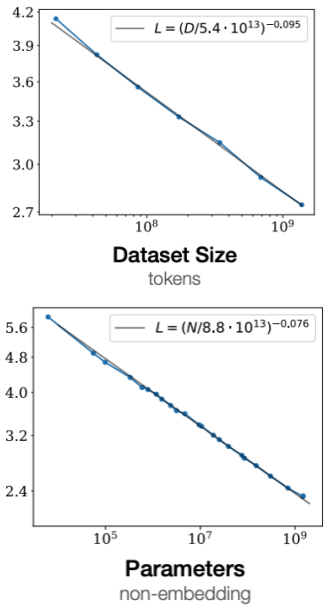
Today's Main Course: Efficiency

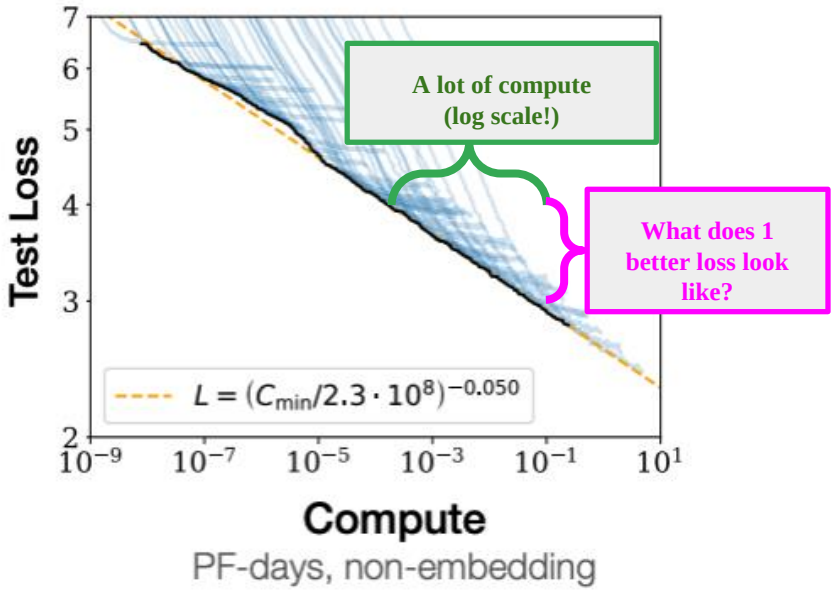


We want the horse to run fast — without feeding it!
(faster AND cheaper)

Why do we need efficiency?

LLMs follow Scaling Laws





Scaling laws for neural language models (2020).

Kaplan et al., 2020:
Language modeling performance improves smoothly as we increase the model size, dataset size, and amount of compute for training.

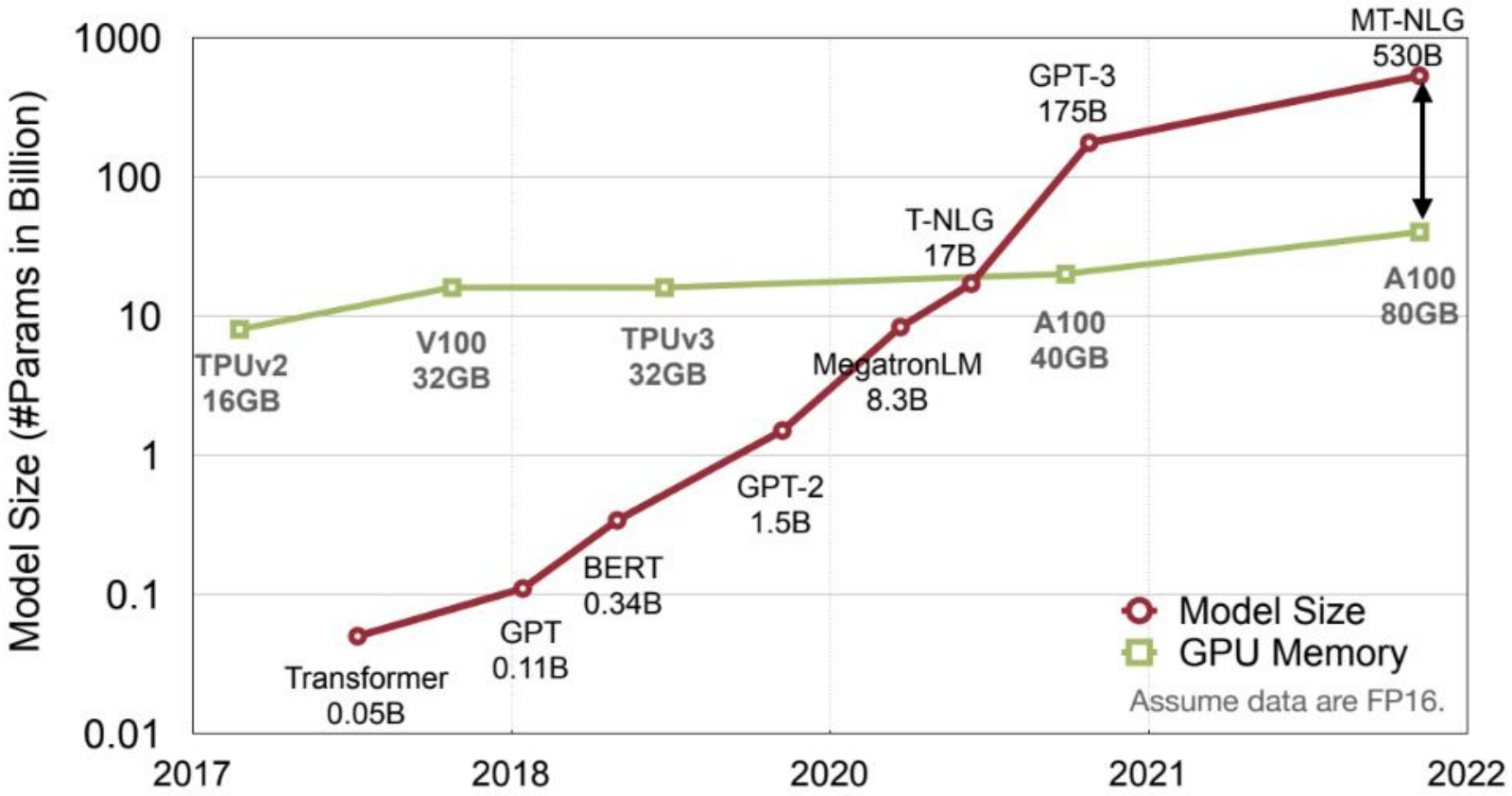
Jason's rephrase: You should expect to get a better language model if you scale up model size, dataset size, and amount of compute.

Note 1: "Within reasonable limits, performance depends very weakly on architectural hyperparameters such as depth and width."

Note 2: By the way, data is unlabeled (self-supervised) data from the internet (e.g., common crawl).

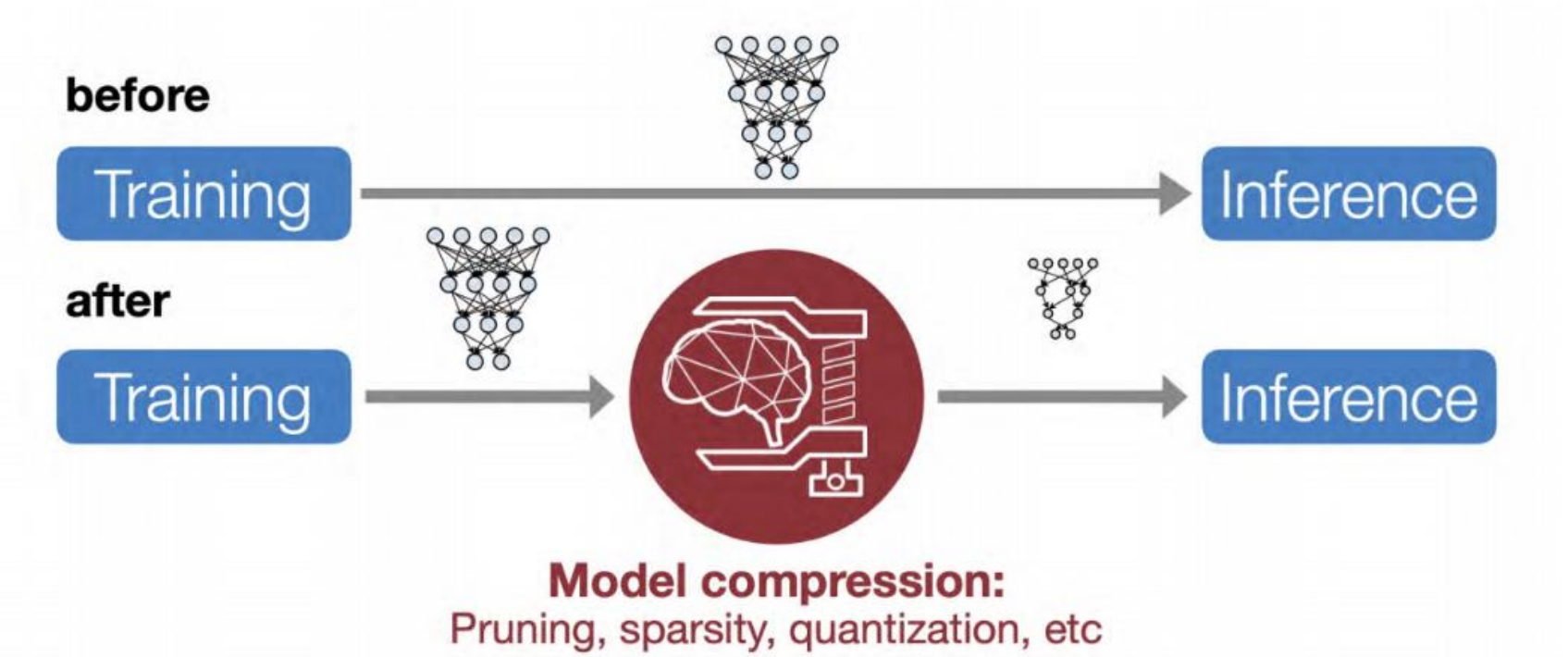
Suggested further reading:
Scaling laws for neural language models (2020).
Training compute-optimal large language models (2022).

Model size of LMs is growing exponentially, yet the hardware…





Bridge the gap between the supply and demand of AI computing



The Guessed Sizes for Anthropic Claude

Sonnet (1T parameters) Opus (3-4T parameters) So much powerful and so much expensive
Mythos (10T parameters) Would be super expensive!

Using (expensive) AI largely improves working efficiency

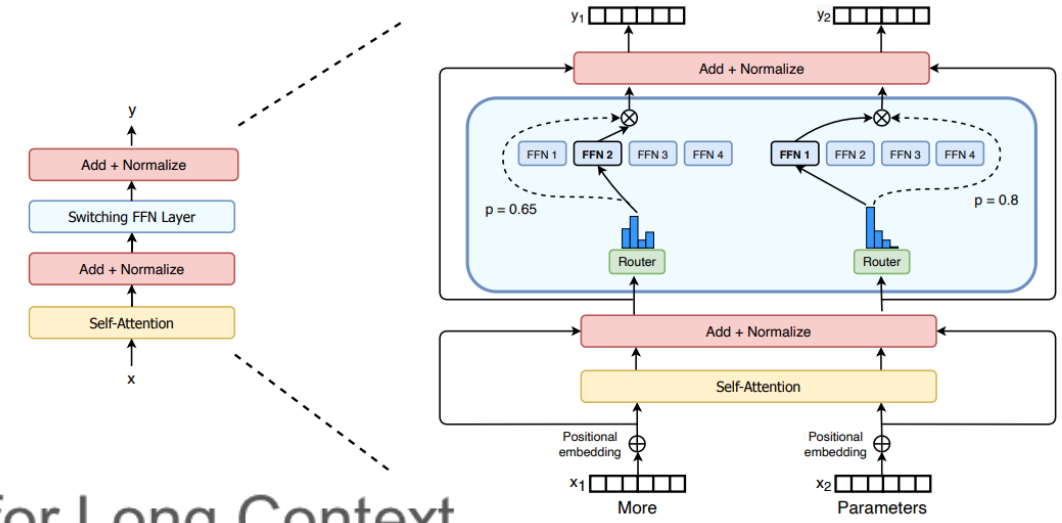
- Shuyu Yao (0.1B+ RMB annual salary)
- Daya Guo (0.1B RMB annual salary)

- 10M RMB annual salary (A PhD graduate in CUHKSZ)
- 10M Dollar annual salary (an HKU AP moved to Meta)

- AI has **widened the productivity gap** between people.

Type I: Efficiency within Transformer

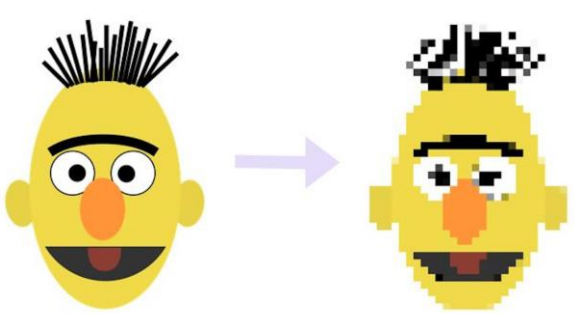
- Sparsity (e.g., Mixture of Expert [1])



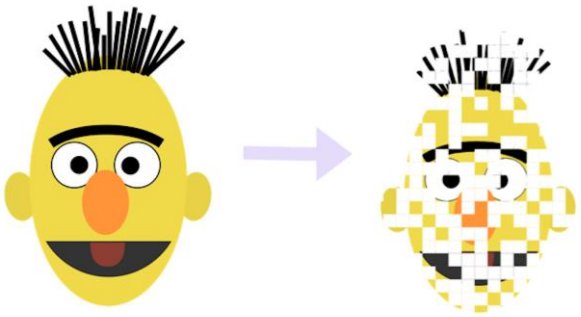
- Efficiency for Long Context
  - Computing complexity of is $O(N^2D)$, which is quadratic to the sequence length

William Fedus, Barret Zoph, Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. https://arxiv.org/pdf/2101.03961.pdf

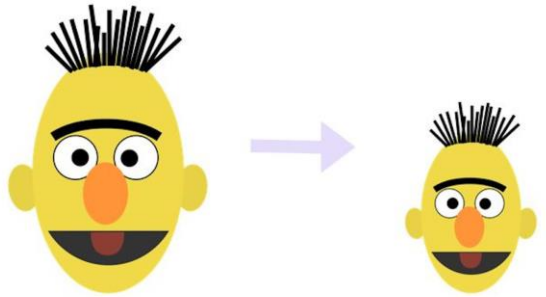
Type II: Efficiency beyond Transformer



Quantization
“Low resolution”



Pruning
Removing weight connections



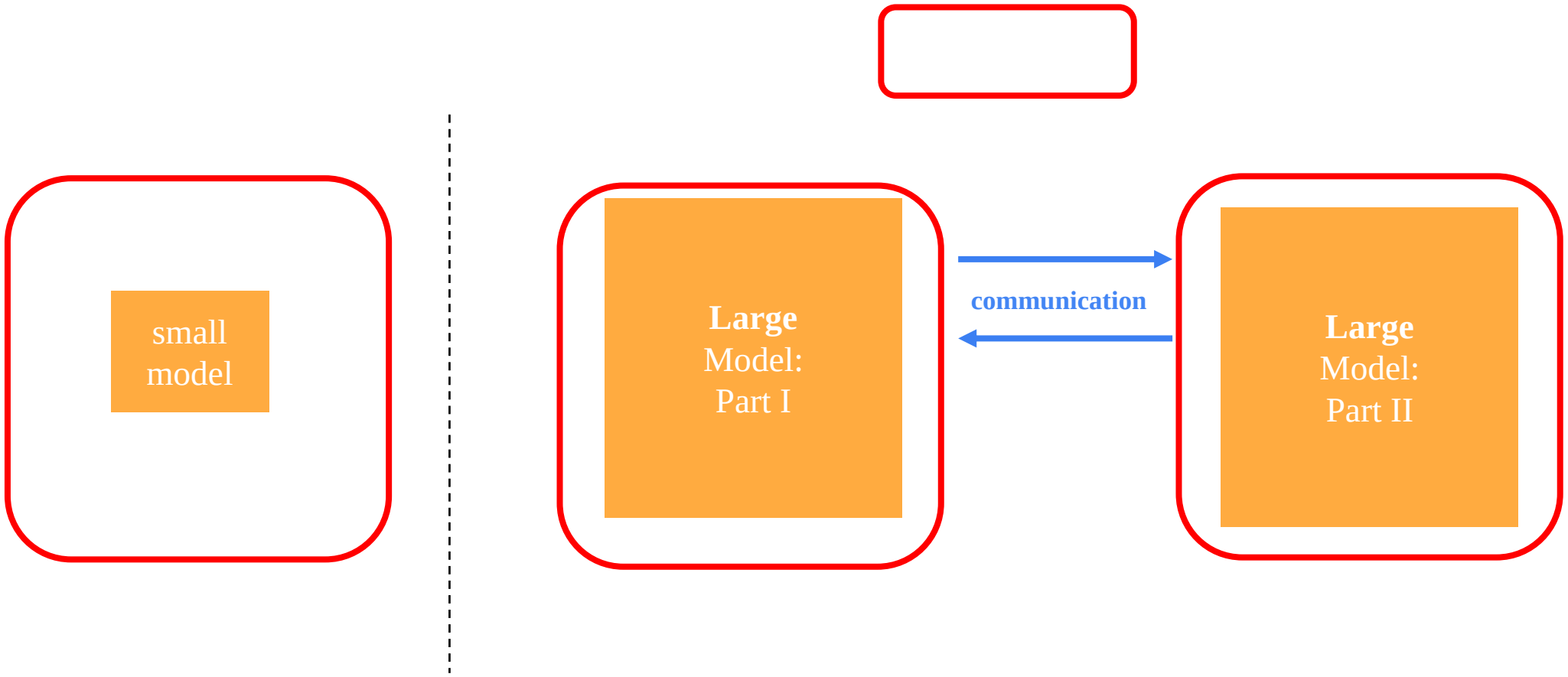
Knowledge distillation
Learning smaller models from big ones

https://rasa.com/blog/compressing-bert-for-faster-prediction-2/

Efficiency after LLMs

All Efficiency methods before LLMs (within/beyond transformer) + Communication costs + Memory saving + Distributed Training Speculative decoding (only for autor

Space Efficiency: From Parameters to Memory

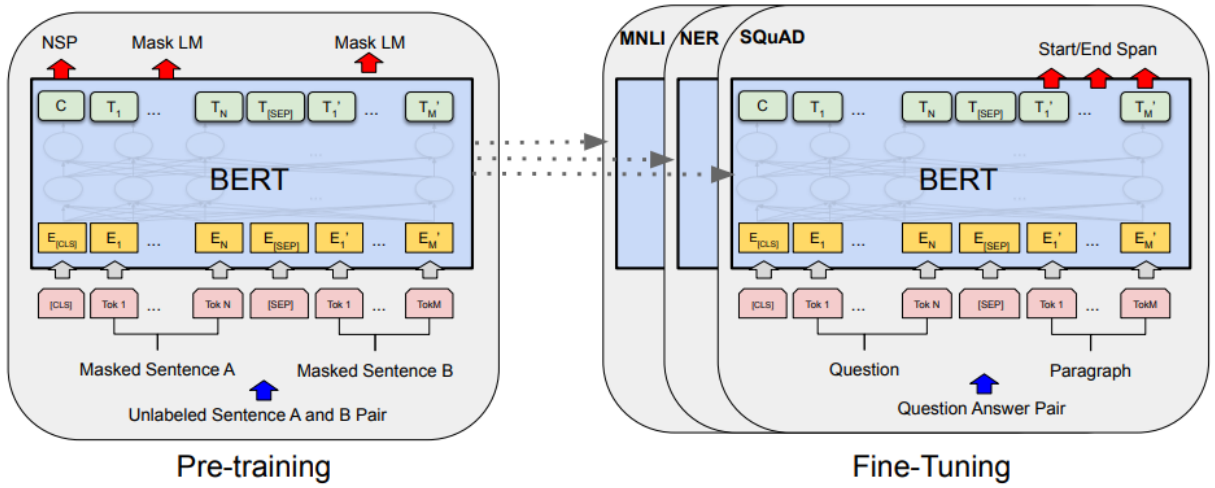


In LLM era, a whole model might not be stored in a GPU memory, **communication costs** may be the bottleneck!

Time Efficiency vs. Space Efficiency

Each task has a full fine-tuned model

Can we just finetune a few (partial) parameters for a new task?



Parameter-efficient fine-tuning

2 Efficient Architectures

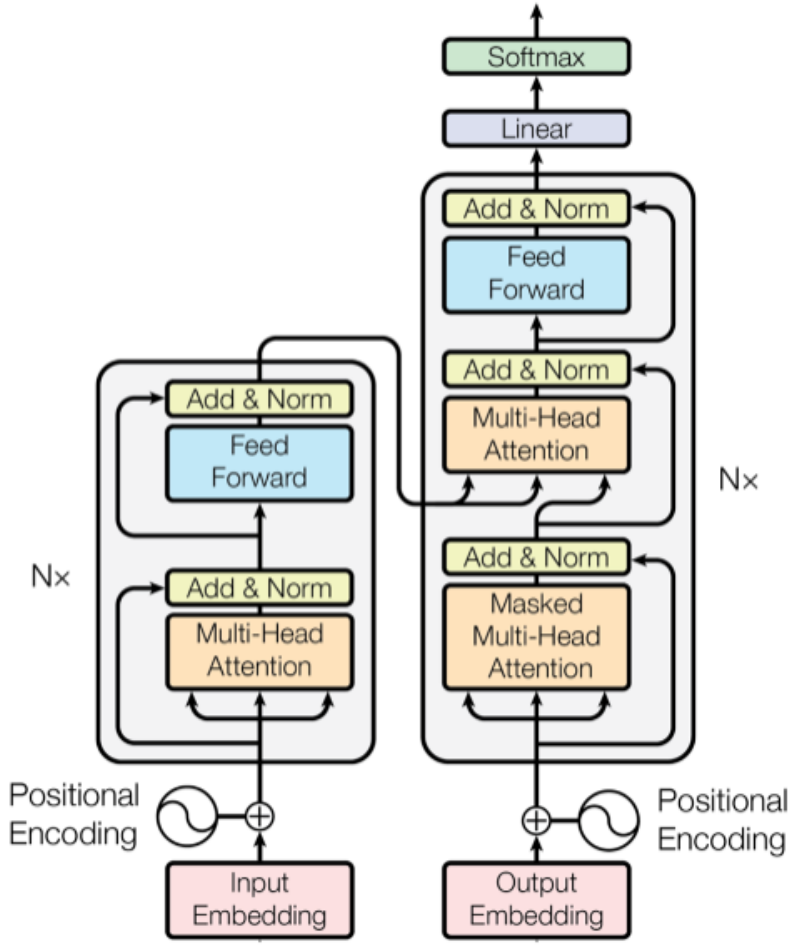
Attention is quadratic in length and the FFN holds most parameters. Make each of them do less work per token.

In this part

2.1 Efficient Attention

2.2 Mixture of Experts

Recap: Transformers



- Each encoder block has two sub-layers:
  - The first is a **multi-head self-attention** mechanism.
  - The second is a position-wise **fully connected feed-forward** network.
- Each decoder block has an additional third sub-layer:
  - The third is a multi-head attention over the output of the encoder stack.
- A residual connection is added around each of the two sub-layers, followed by layer normalization:

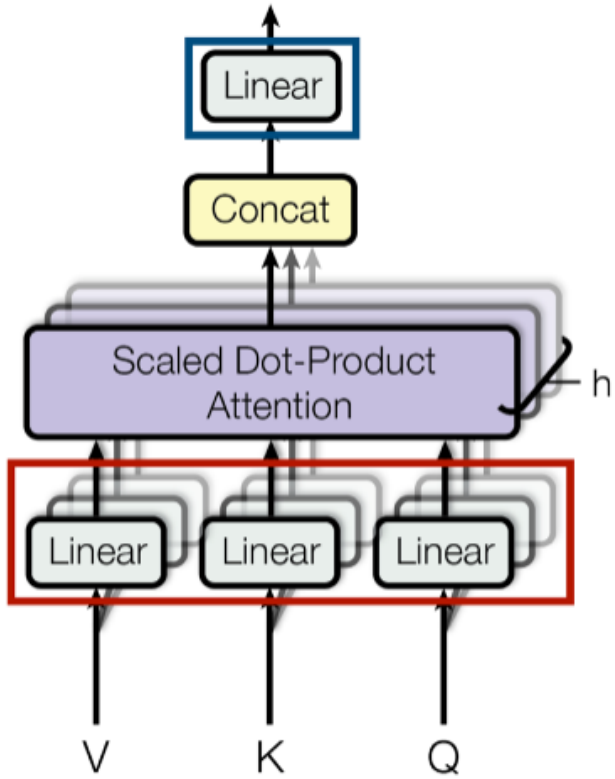
$$\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$$

- The decoder generates the output sequence of symbols one element at a time in an **auto-regressive** manner.

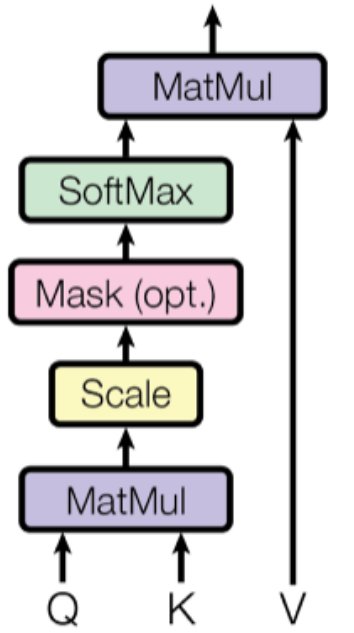
Recap: Transformers - Multi-head Self-Attention (MHSA)

- **Project** Q, K and V with h **different**, learned linear projections.
- Perform the **scaled dot-product attention** function on each of these projected versions of Q, K and V **in parallel**.
- **Concatenate** the output values.
- **Project** the output values again, resulting in the final values.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



Scaled Dot-Product Attention



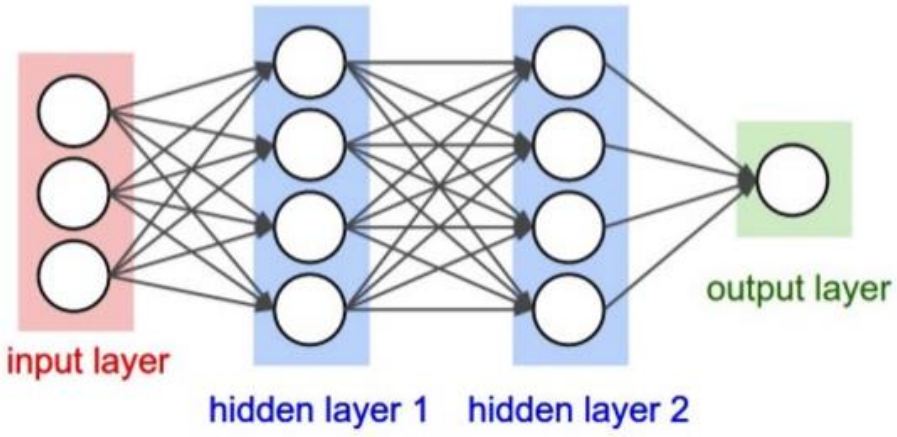
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Recap: Transformers - Feed-Forward Network (FFN)

- Each block in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position **separately** and **identically**.
- This consists of two linear transformations with a ReLU activation in between.

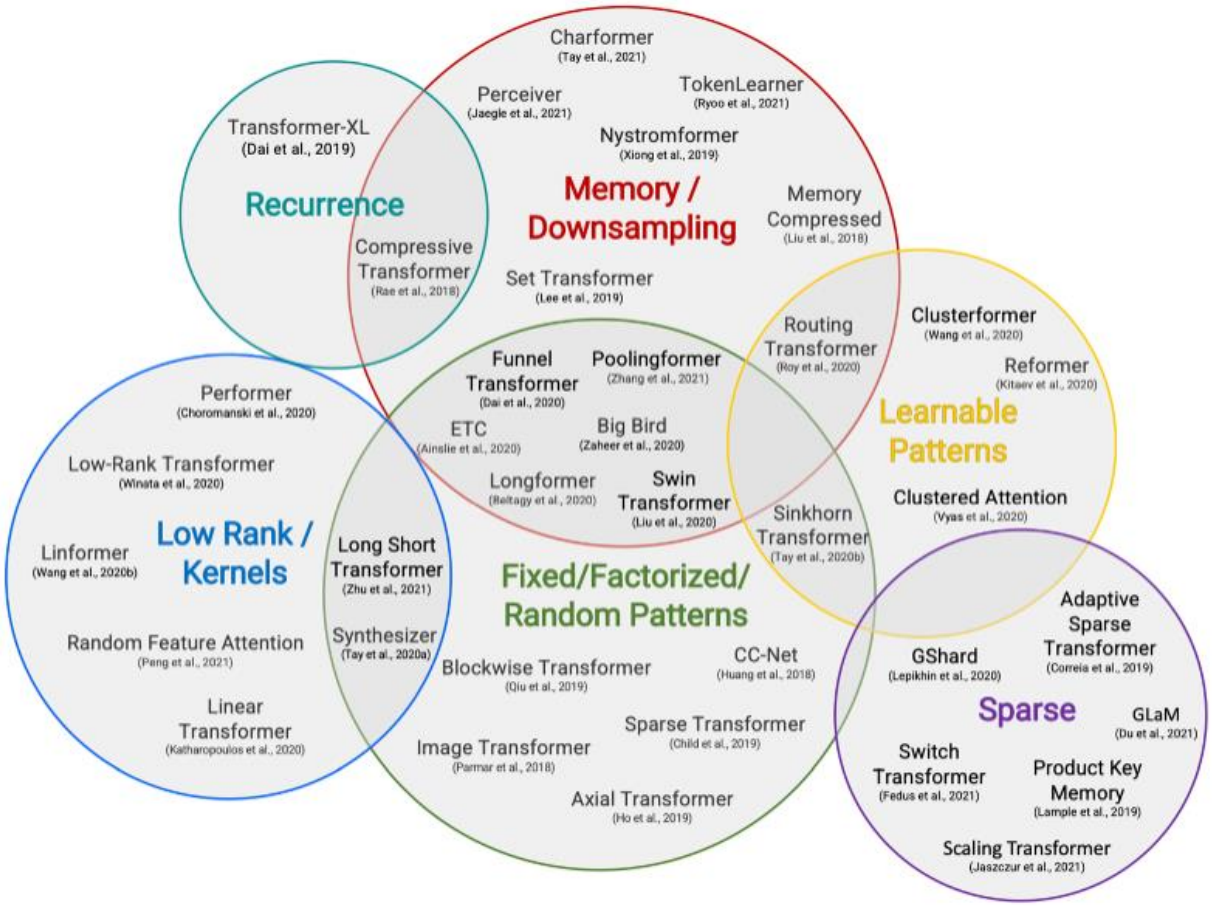
$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- The middle hidden size is usually larger than and input and output size (**inverted bottleneck**).



Model	#L	#H	d_{model}	LR	Batch
125M	12	12	768	$6.0e{-4}$	0.5M
350M	24	16	1024	$3.0e{-4}$	0.5M
1.3B	24	32	2048	$2.0e{-4}$	1M
2.7B	32	32	2560	$1.6e{-4}$	1M
6.7B	32	32	4096	$1.2e{-4}$	2M
13B	40	40	5120	$1.0e{-4}$	4M
30B	48	56	7168	$1.0e{-4}$	4M
66B	64	72	9216	$0.8e{-4}$	2M
175B	96	96	12288	$1.2e{-4}$	2M

Efficient Transformers

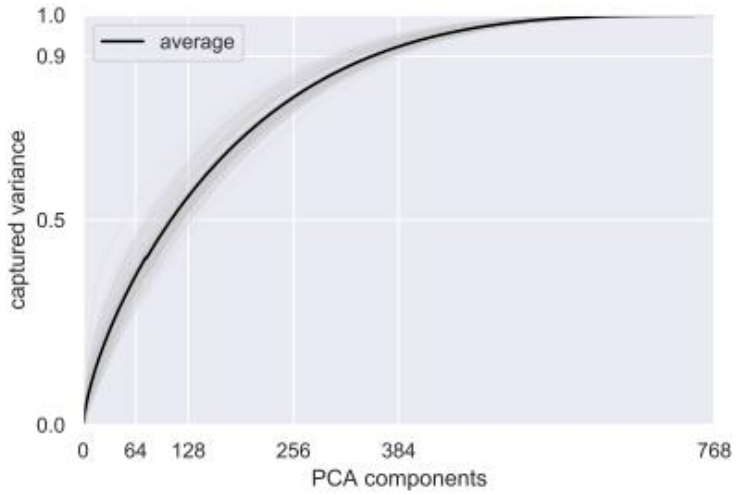


Efficient Transformers: A Survey [Tay *et al.*, 2020]

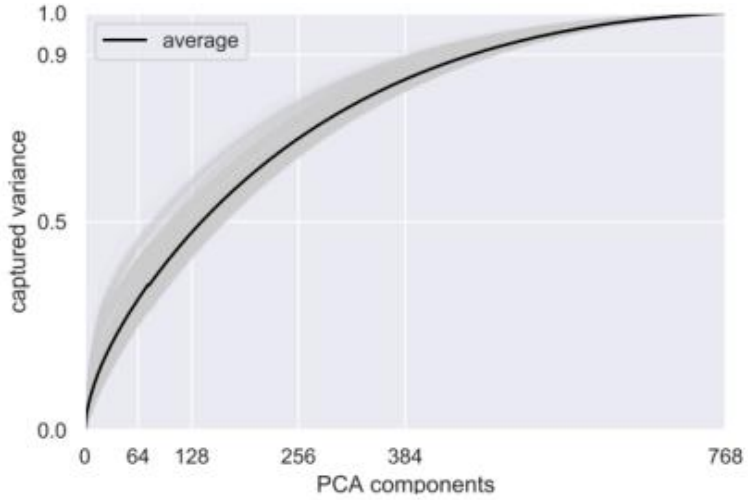
Could we make it efficient and also maintain the performance?

Can the horse run fast without being fed?

Motivation: Parameter redundancy existed



(a) PCA for each single weight matrix



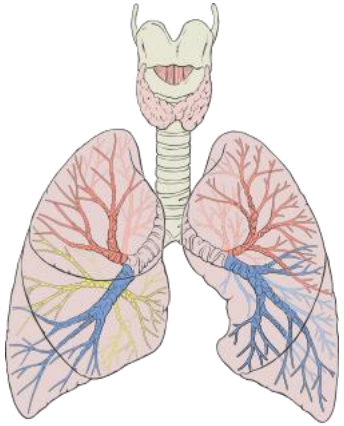
(b) PCA for a pair of matrices along columns

Figure 1: PCA for existing weight block matrices in BERT-base. We got nearly similar results in Fig. 5 for paired matrices along rows and columns, as shown in App. C.

Benyou Wang, Yuxin Ren, Lifeng Shang , Xin Jiang, Qun Liu. Exploring extreme parameter compression for pre-trained language models. ICLR 2022.

Decomposability

- A computing module f is **decomposable** if its sub-components {g1, g2,⋯gH} could be independently calculated without interactions: $f(x) = \delta(g1(x), g2(x),\cdots, gH(x))$. Usually, δ is a simple operation that has negligible computing cost compared to **g**



Decomposability might lead to **redundancy**, as it has backup modules.

Self-attention is decomposable

- As there exist multiple heads, each head could be independently calculated.

$$\text{Att}_h(\mathbf{X}) = \text{Softmax}(\frac{1}{\sqrt{d}}\mathbf{X}\boldsymbol{W}_h^Q \boldsymbol{W}_h^{K^T}\mathbf{X}^T)\mathbf{X}\boldsymbol{W}_h^V \boldsymbol{W}_h^{O^T}$$

Feed-forward network is also decomposable

- It performs like a multi-head mechanism

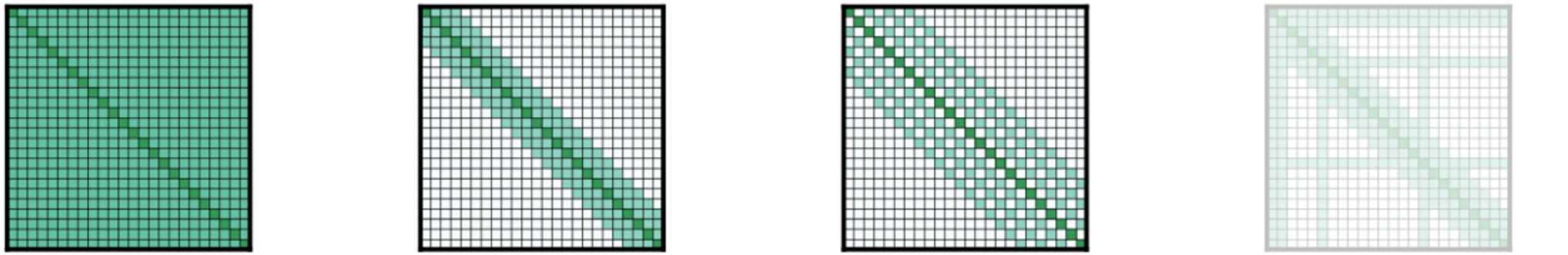
$$\mathrm{FFN}(\mathbf{X}) = \sum_{h=1}^{4D} \mathrm{GeLU}(\mathbf{X}\boldsymbol{W}_{\cdot,h}^{In} + b_h^{In})\boldsymbol{W}_{h,\cdot}^{Out} + b^{Out}$$

2.1 Efficient Attention

Full attention compares every token with every other token: $O(N^2)$. Sparse patterns and shared KV heads cut the cost.

Sparse Attention - LongFormer

Local Attention + Global Attention



(a) Full n^2 attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window

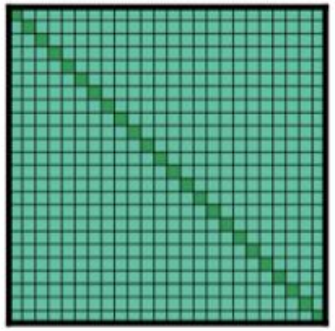
- Attention with **sliding window** (analogous to CNNs):
  - A fixed-size window attention surrounding each token.
  - The complexity is reduced from $O(N^2)$ to **O(N x W)**, where W is the window size.

- Attention with **dilated sliding window** (analogous to dilated CNNs):
  - Dilate the sliding window with gaps of size dilation D.
  - The receptive field is enlarged from W to **W x D**, with the same complexity.

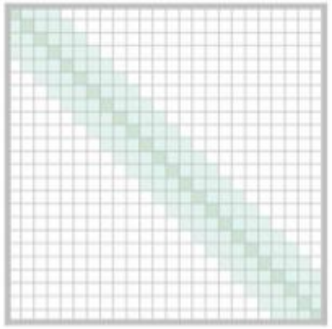
Longformer: The Long-Document Transformer [Beltagy *et al.*, 2020]

Sparse Attention - LongFormer

Local Attention + Global Attention



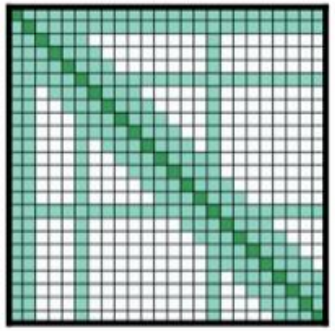
(a) Full n^2 attention



(b) Sliding window attention



(c) Dilated sliding window



(d) Global+sliding window

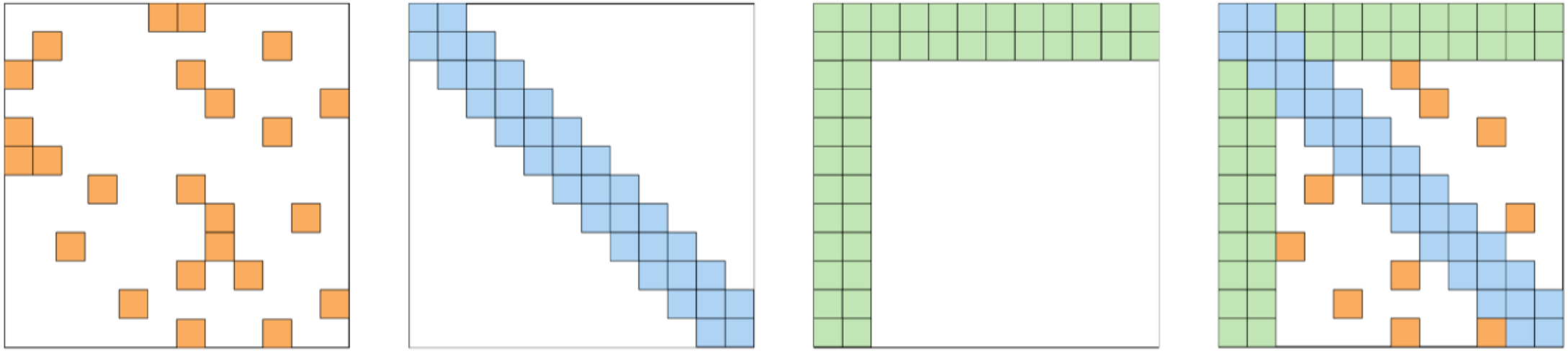
- **Global attention** added on a few **pre-selected** input locations:
  - Classification: The special token ([CLS]), aggregating the whole sequence.
  - QA: All question tokens, allowing the model to compare the question with the document.

- Global attention is applied **symmetrically**:
  - A token with a global attention attends to all tokens across the sequence, and all tokens in the sequence attend to it.

Longformer: The Long-Document Transformer [Beltagy *et al.*, 2020]

Sparse Attention - Big Bird

Random Attention + Local Attention + Global Attention



(a) Random attention (b) Window attention (c) Global Attention (d) BIGBIRD

- Random sparse attention:
  - Each query attends over r random number of keys: *i.e.* A(i, ·) = 1 for r randomly chosen keys.
  - Information can flow fast between any pair of nodes (rapid mixing time for random walks).

Big Bird: Transformers for Longer Sequences [Zaheer *et al.*, 2021]

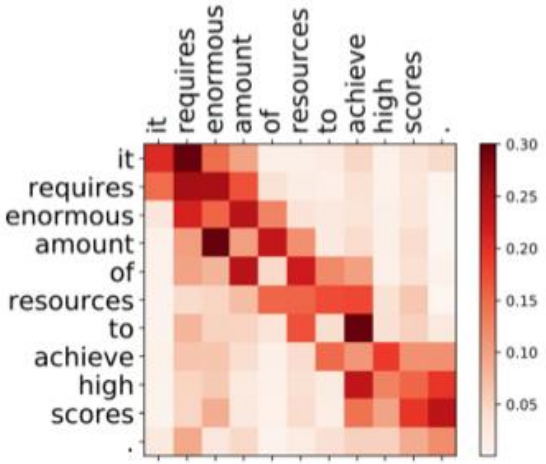
Sparse Attention - Lite Transformer

Local Convolution + Global Attention

- Long-Short Range Attention (LSRA):
  - **Convolution**: Efficiently extract the **local** features.
  - **Attention**: Tailored for **global** feature extraction.

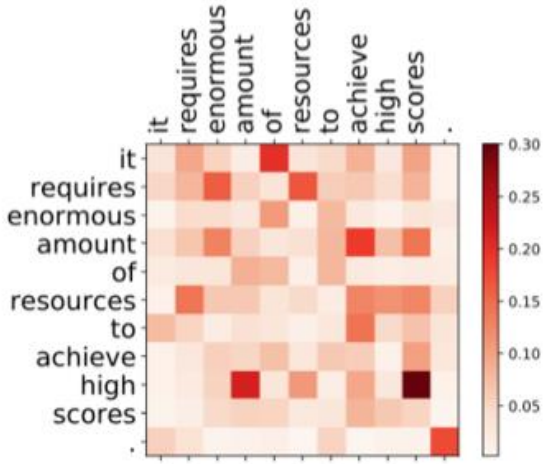
Original Attention

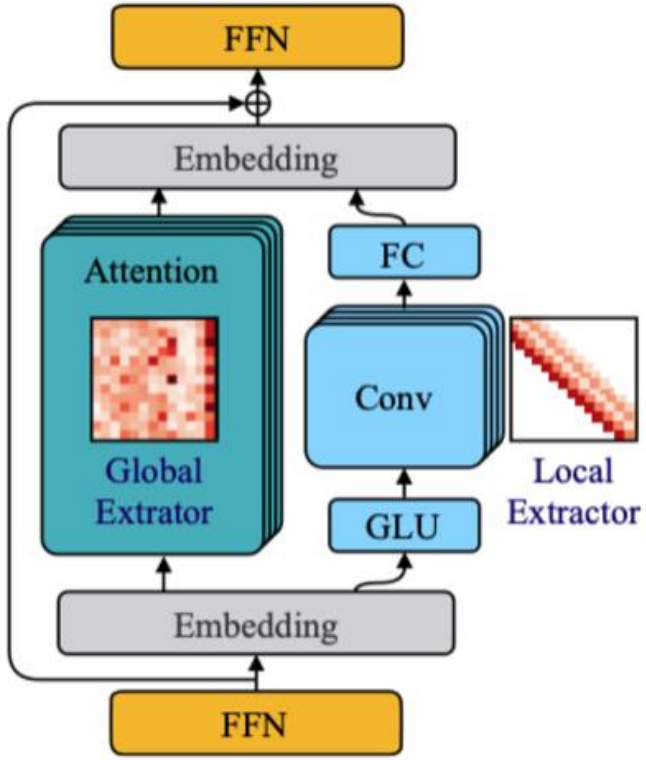
(Too much emphasize on local feature extraction)



Attention in LSRA

(Dedicated for global feature extraction)





Lite Transformer with Long-Short Range Attention [Wu *et al.*, 2020]

Sharing KV Heads: MQA and GQA

Grouped query attention (GQA) is a method to increase the efficiency of the attention mechanism in transformer models, often used to enable faster inference from large language models (LLMs).

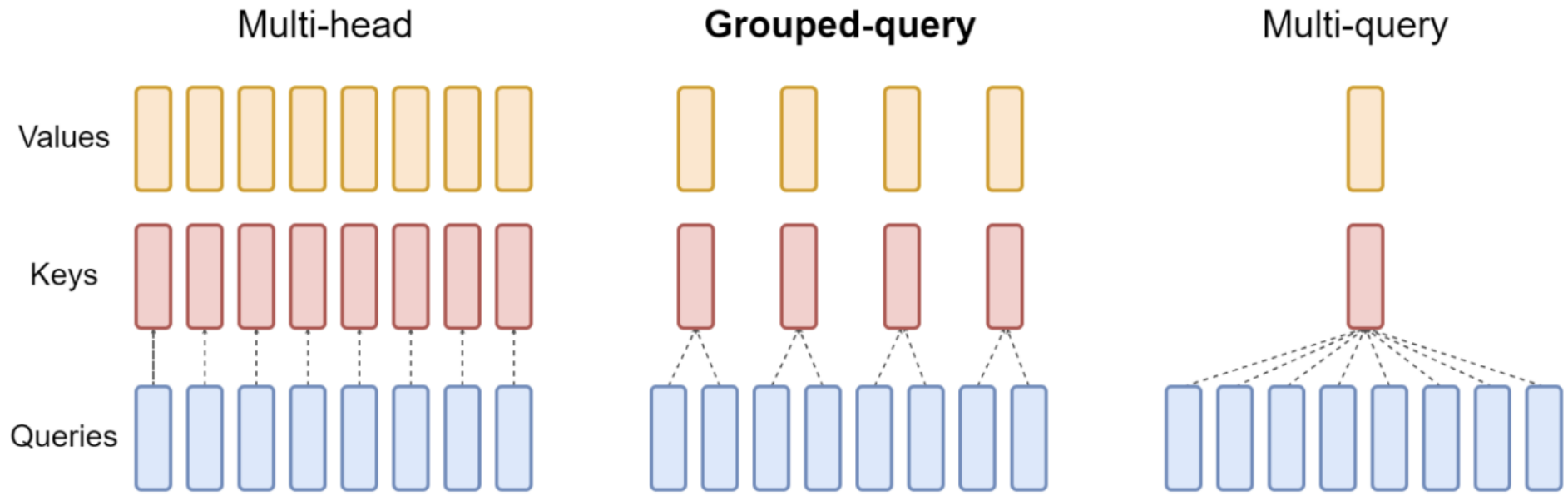


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

Multi-Head Latent Attention (DeepSeek-V2)

Multi-head Latent Attention (MLA) equipped with low-rank key-value joint compression, and achieves better performance than MHA, but requires a significantly smaller amount of KV cache.

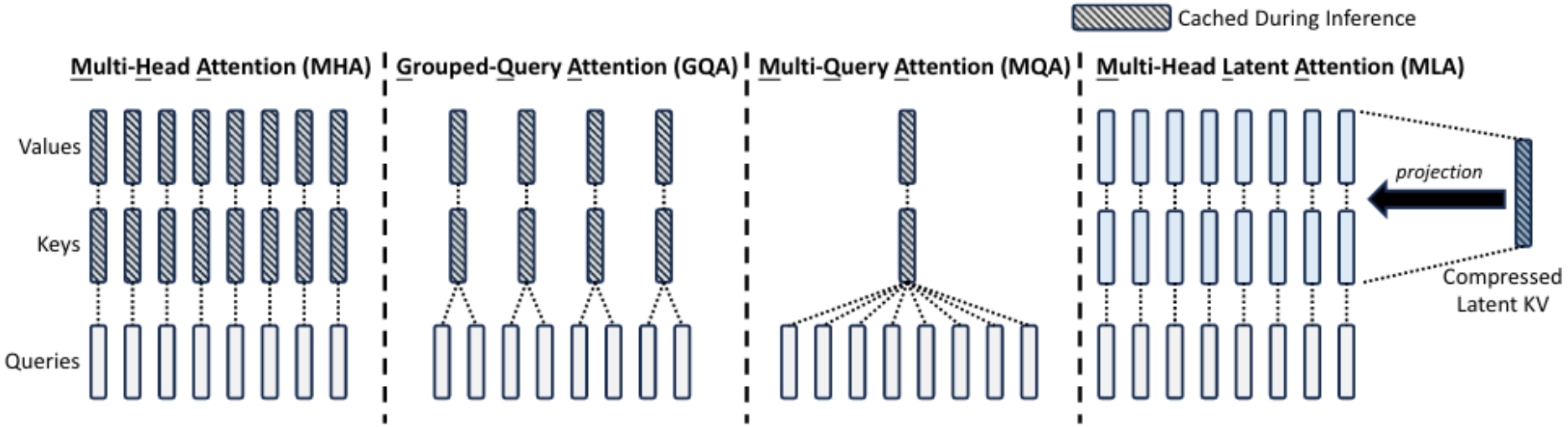


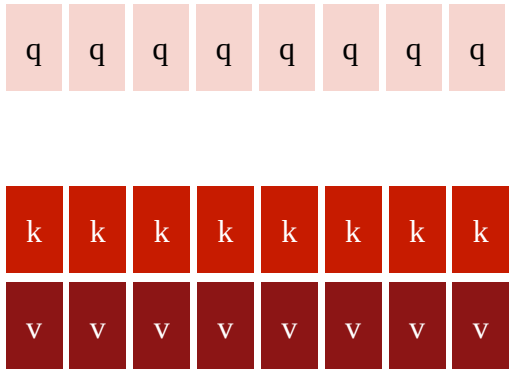
Figure 3 | Simplified illustration of Multi-Head Attention (MHA), Grouped-Query Attention (GQA), Multi-Query Attention (MQA), and Multi-head Latent Attention (MLA). Through jointly compressing the keys and values into a latent vector, MLA significantly reduces the KV cache during inference.

Keep the expressiveness (approaching MHA) but also save KV caches than GQA and MQA

Sharing Keys and Values: MHA → GQA → MQA → MLA

MHA

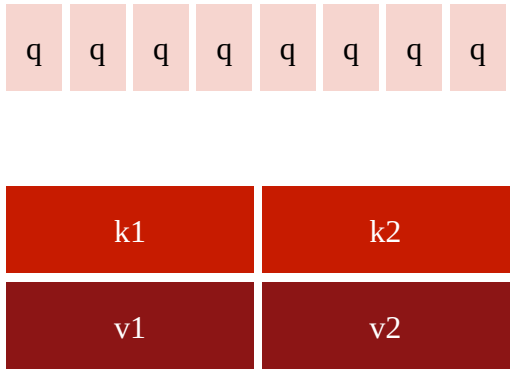
8 query heads, 8 KV heads



KV per token: 16 vectors

GQA

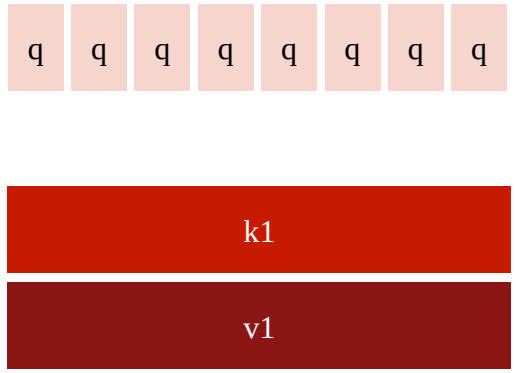
8 query heads share 2 KV heads



4× smaller (Llama 2/3, Qwen)

MQA

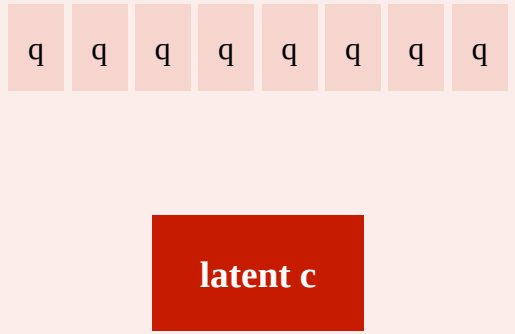
all 8 query heads share 1 KV head



8× smaller, some quality loss

MLA

compress K and V into one small latent



up-project to K, V on the fly

≈ GQA cost, MHA quality (DeepSeek-V2/V3)

Decoding is bound by reading the KV cache — fewer KV bytes per token means faster decoding and longer contexts.

Shazeer (2019) MQA · Ainslie et al. GQA (EMNLP 2023) · DeepSeek-AI. DeepSeek-V2 (2024): MLA.

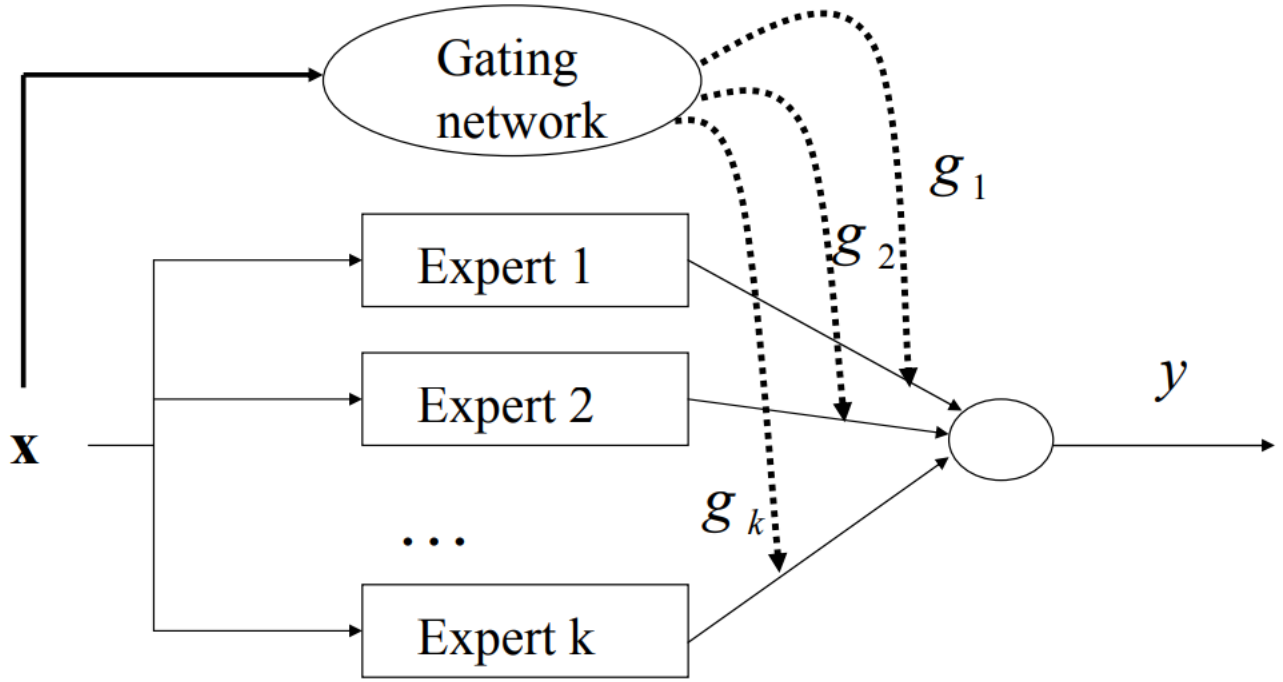
2.2 Mixture of Experts

Make the FFN sparse: many experts, but each token only visits a few of them.

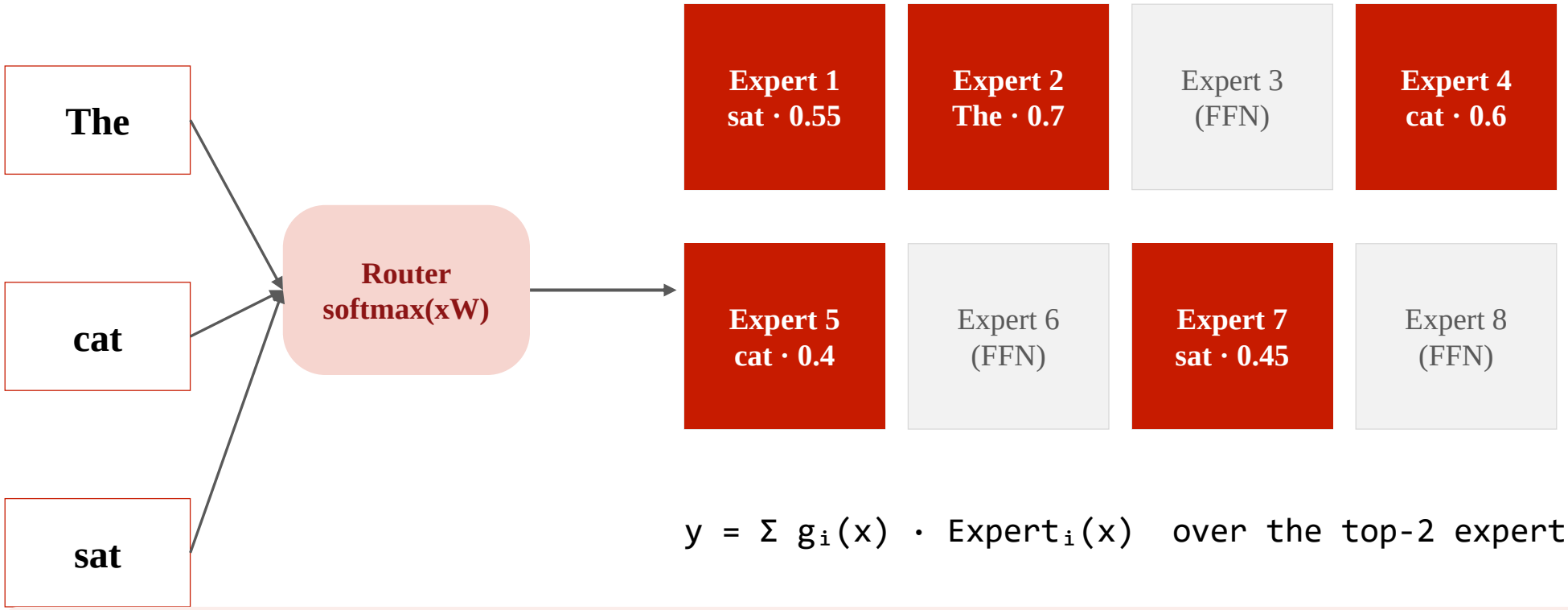
Mixture of Experts

- **Gating network** : decides what expert to use

  $g_1, g_2, \ldots g_k$ - gating functions



Mixture of Experts: Each Token Visits Only a Few Experts



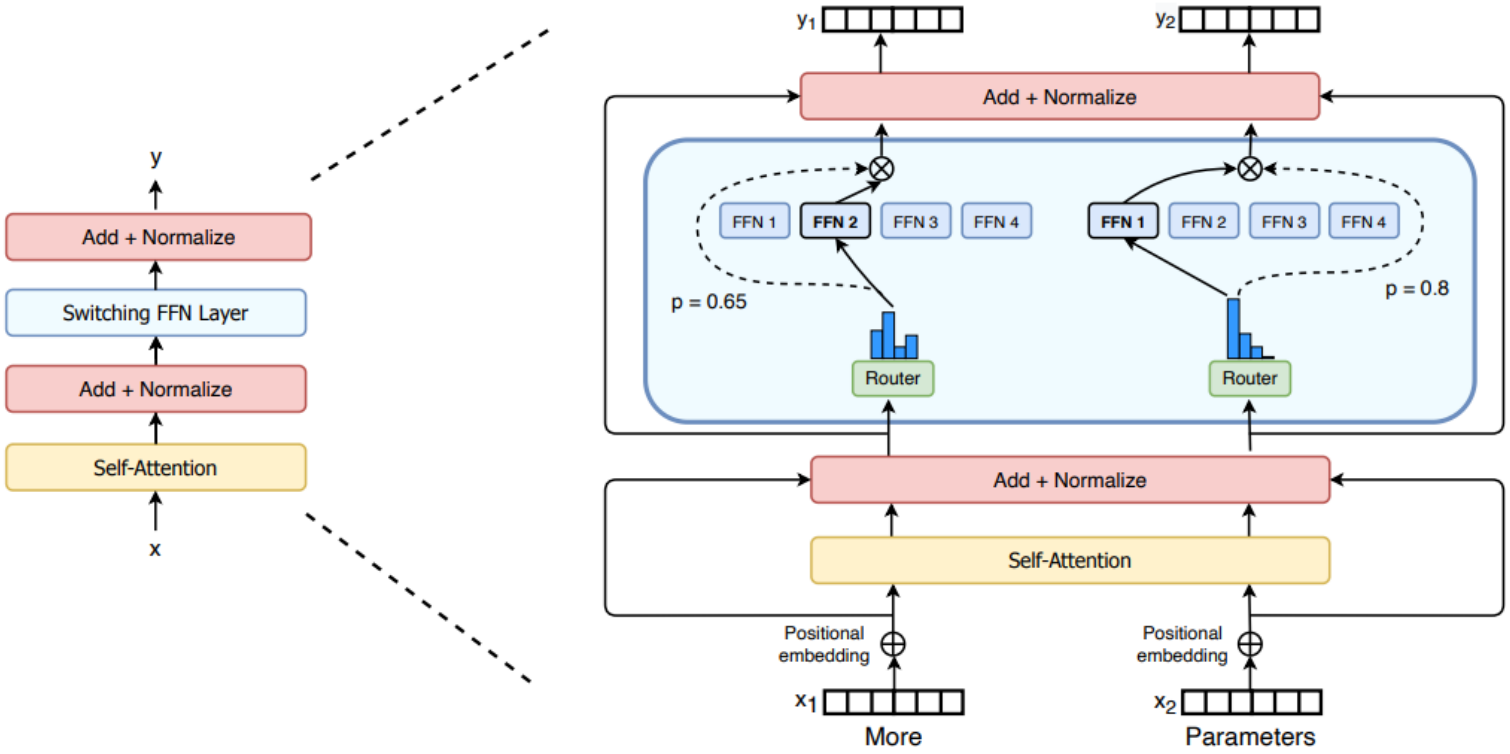
$$y = \Sigma\ g_i(x) \cdot \text{Expert}_i(x) \quad \text{over the top-2 experts}$$

Capacity of 8 experts, compute of 2. DeepSeek-V3: 671B parameters, only 37B active per token.

A load-balancing loss (or bias) keeps the router from sending every token to the same expert.

Shazeer et al. (2017) · Fedus et al. Switch Transformers (2021) · DeepSeek-AI. DeepSeek-V3 Technical Report (2024).

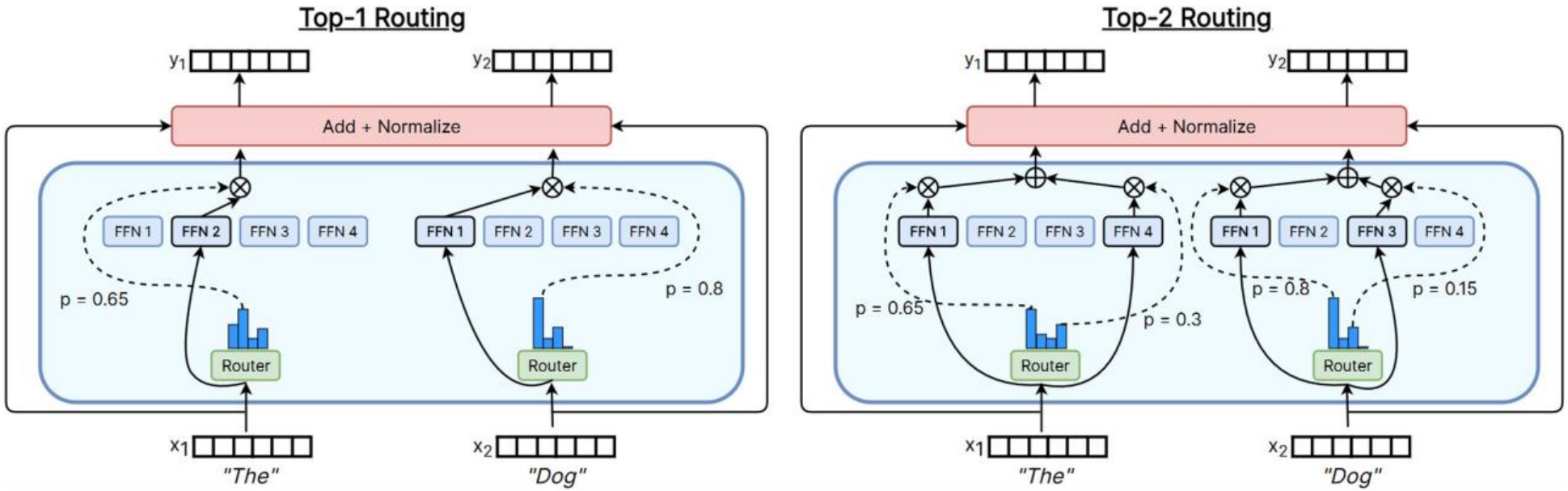
MoE in Transformer



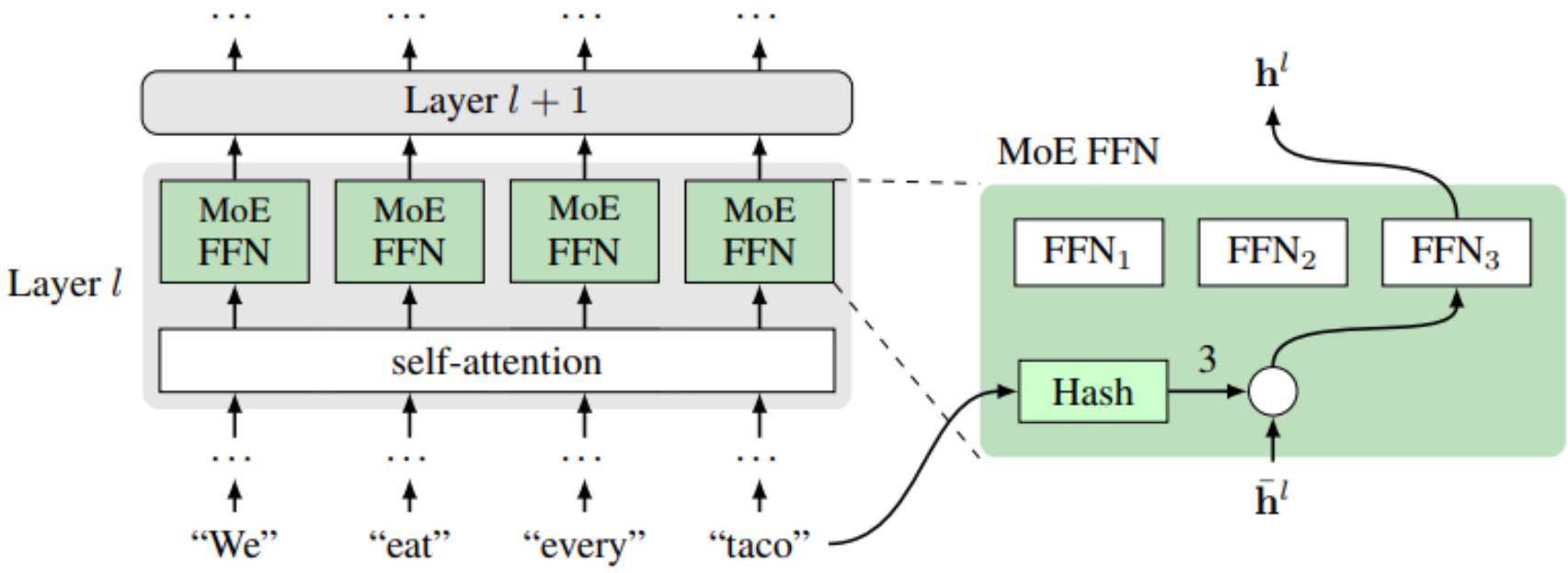
Token-specific routing

William Fedus, Barret Zoph, Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. https://arxiv.org/pdf/2101.03961.pdf

Routing Algorithms



An Example of Hash Routing



token-based hashing

Stephen Roller, Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston. Hash Layers For Large Sparse Models. https://arxiv.org/pdf/2106.04426.pdf

Random hash also works

Table 3: **Different Hash Layering Methods** on pushshift.io Reddit.

Model	Hashing Type	Valid PPL	Test PPL
Baseline Transformer	-	24.90	24.96
Hash Layer 1x64	Balanced assignment	23.16	23.23
Hash Layer 1x64	Fixed random assignment	23.22	23.27
Hash Layer 1x64	Token clustering (using Baseline Transformer)	23.90	23.99
Hash Layer 1x64	Dispersed Hash (within token clusters)	23.17	23.22
Hash Layer 1x64	Hash on position	25.07	25.14
Hash Layer 1x64	Bigrams	24.19	24.28
Hash Layer 1x64	Previous token	24.16	24.22
Hash Layer 1x64	Future token predictions (using Transformer Baseline)	25.02	25.09
Hash Layer 1x64	Future token (Oracle)	1.97	1.97
Hash Layer 5x16	Same hash per layer (balance assignment)	23.74	23.81
Hash Layer 5x16	Different Hash per layer	23.21	23.27

Stephen Roller, Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston. Hash Layers For Large Sparse Models. https://arxiv.org/pdf/2106.04426.pdf

MoE Models

- Open-source (above the arrow).
- Private models (under the arrow).



MoE Design Choices

What should we care when designing a MoE?

Network types	FFN, Attention
Fine-grained experts	64 experts/128 experts/…
Shared experts	Isolated experts
Activation Function	ReLU/GEGLU/SwiGLU
MoE frequency	Every two layer/Each layer/…
Training auxiliary loss	Auxiliary loss/Z-loss/…

Fine-grained and Shared Experts

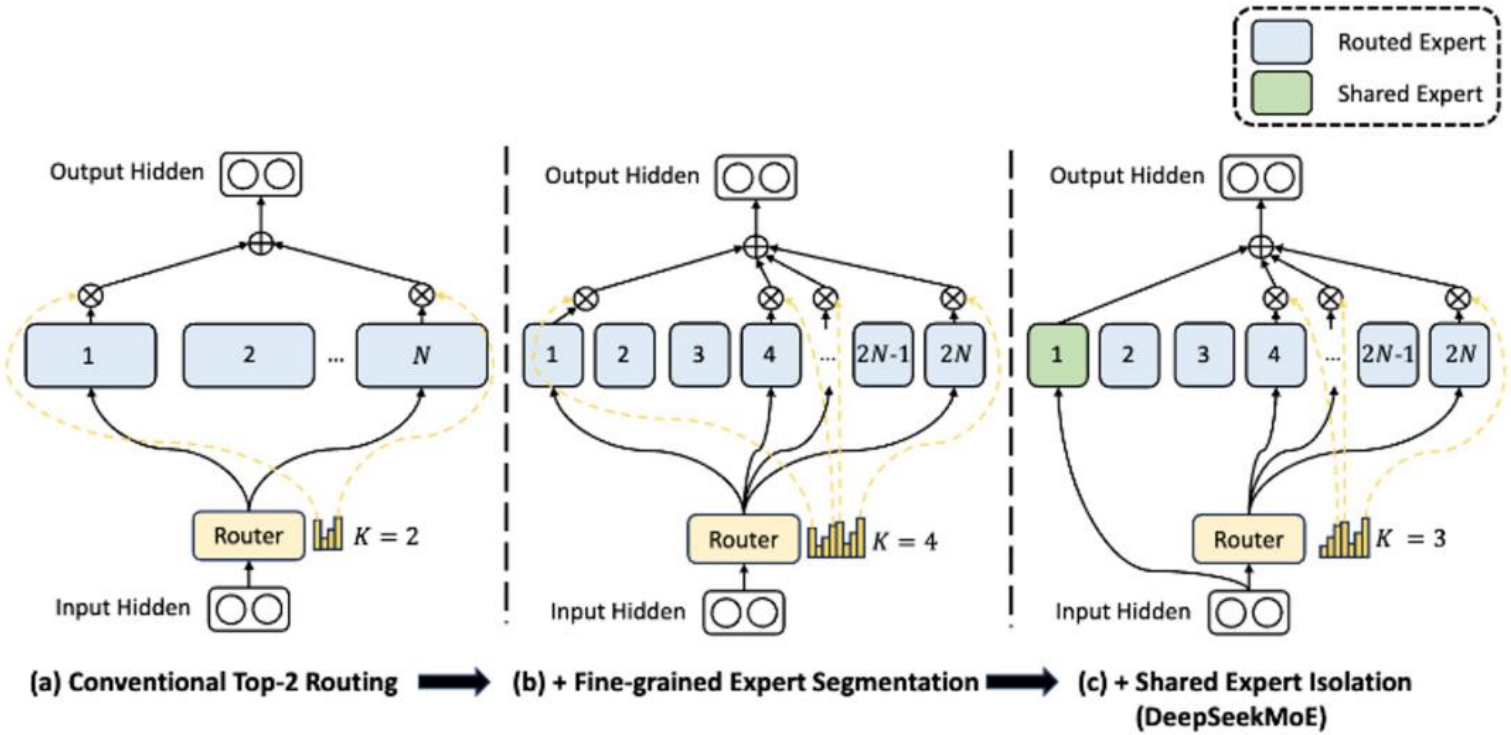


Figure 2 | Illustration of DeepSeekMoE. Subfigure (a) showcases an MoE layer with the conventional top-2 routing strategy. Subfigure (b) illustrates the fine-grained expert segmentation strategy. Subsequently, subfigure (c) demonstrates the integration of the shared expert isolation strategy, constituting the complete DeepSeekMoE architecture. It is noteworthy that across these three architectures, the number of expert parameters and computational costs remain constant.

MoE Experts Design

Reference	Models	Expert Count (Activ./Total)	d_{model}	d_{ffn}	d_{expert}	#L	#H	d_{head}	Placement Frequency	Activation Function	Share Expert Count
GShard [86] (2020)	600B	2/2048	1024	8192	d_{ffn}	36	16	128	1/2	ReLU	0
	200B	2/2048	1024	8192	d_{ffn}	12	16	128	1/2	ReLU	0
	150B	2/512	1024	8192	d_{ffn}	36	16	128	1/2	ReLU	0
	37B	2/128	1024	8192	d_{ffn}	36	16	128	1/2	ReLU	0
Switch [49] (2021)	7B	1/128	768	2048	d_{ffn}	12	12	64	1/2	GEGLU	0
	26B	1/128	1024	2816	d_{ffn}	24	16	64	1/2	GEGLU	0
	395B	1/64	4096	10240	d_{ffn}	24	64	64	1/2	GEGLU	0
	1571B	1/2048	2080	6144	d_{ffn}	15	32	64	1	ReLU	0
GLaM [44] (2021)	0.1B/1.9B	2/64	768	3072	d_{ffn}	12	12	64	1/2	GEGLU	0
	1.7B/27B	2/64	2048	8192	d_{ffn}	24	16	128	1/2	GEGLU	0
	8B/143B	2/64	4096	16384	d_{ffn}	32	32	128	1/2	GEGLU	0
	64B/1.2T	2/64	8192	32768	d_{ffn}	64	128	128	1/2	GEGLU	0
DeepSpeed-MoE [121] (2022)	350M/13B	2/128	1024	$4d_{model}$	d_{ffn}	24	16	64	1/2	GeLU	0
	1.3B/52B	2/128	2048	$4d_{model}$	d_{ffn}	24	16	128	1/2	GeLU	0
	PR-350M/4B	2/32-2/64	1024	$4d_{model}$	d_{ffn}	24	16	64	1/2, 10L-32E, 2L-64E	GeLU	1
	PR-1.3B/31B	2/64-2/128	2048	$4d_{model}$	d_{ffn}	24	16	128	1/2, 10L-64E, 2L-128E	GeLU	1
ST-MoE [197] (2022)	0.8B/4.1B	2/32	1024	2816	d_{ffn}	27	16	64	1/4, add extra FFN	GEGLU	0
	32B/269B	2/64	5120	20480	d_{ffn}	27	64	128	1/4, add extra FFN	GEGLU	0
Mixtral [74] (2023)	13B/47B	2/8	4096	14336	d_{ffn}	32	32	128	1	SwiGLU	0
	39B/141B	2/8	6144	16384	d_{ffn}	56	48	128	1	SwiGLU	0
LLAMA-MoE [149] (2023)	3.0B/6.7B	2/16	4096	11008	688	32	32	128	1	SwiGLU	0
	3.5B/6.7B	4/16	4096	11008	688	32	32	128	1	SwiGLU	0
	3.5B/6.7B	2/8	4096	11008	1376	32	32	128	1	SwiGLU	0
DeepSeekMoE [30] (2024)	0.24B/1.89B	8/64	1280	-	$\frac{1}{4}d_{ffn}$	9	10	128	1	SwiGLU	1
	2.8B/16.4B	8/66	2048	10944	1408	28	16	128	1, except 1st layer	SwiGLU	2
	22B/145B	16/132	4096	-	$\frac{1}{8}d_{ffn}$	62	32	128	1, except 1st layer	SwiGLU	4
OpenMoE [172] (2024)	339M/650M	2/16	768	3072	d_{ffn}	12	12	64	1/4	SwiGLU	1
	2.6B/8.7B	2/32	2048	8192	d_{ffn}	24	24	128	1/6	SwiGLU	1
	6.8B/34B	2/32	3072	12288	d_{ffn}	32	24	128	1/4	SwiGLU	1
Qwen1.5-MoE [151] (2024)	2.7B/14.3B	8/64	2048	5632	1408	24	16	128	1	SwiGLU	4
DBRX [34] (2024)	36B/132B	4/16	6144	10752	d_{ffn}	40	48	128	1	SwiGLU	0
Jamba [94] (2024)	12B/52B	2/16	4096	14336	d_{ffn}	32	32	128	1/2, 1:7 Attention:Mamba	SwiGLU	0
Skywork-MoE [154] (2024)	22B/146B	2/16	4608	12288	d_{ffn}	52	36	128	1	SwiGLU	0
Yuan 2.0-M32 [166] (2024)	3.7B/40B	2/32	2048	8192	d_{ffn}	24	16	256	1	SwiGLU	0

- Most recent models place MoE each layer.
- Some of recent models apply Shared experts.

Auxiliary Loss

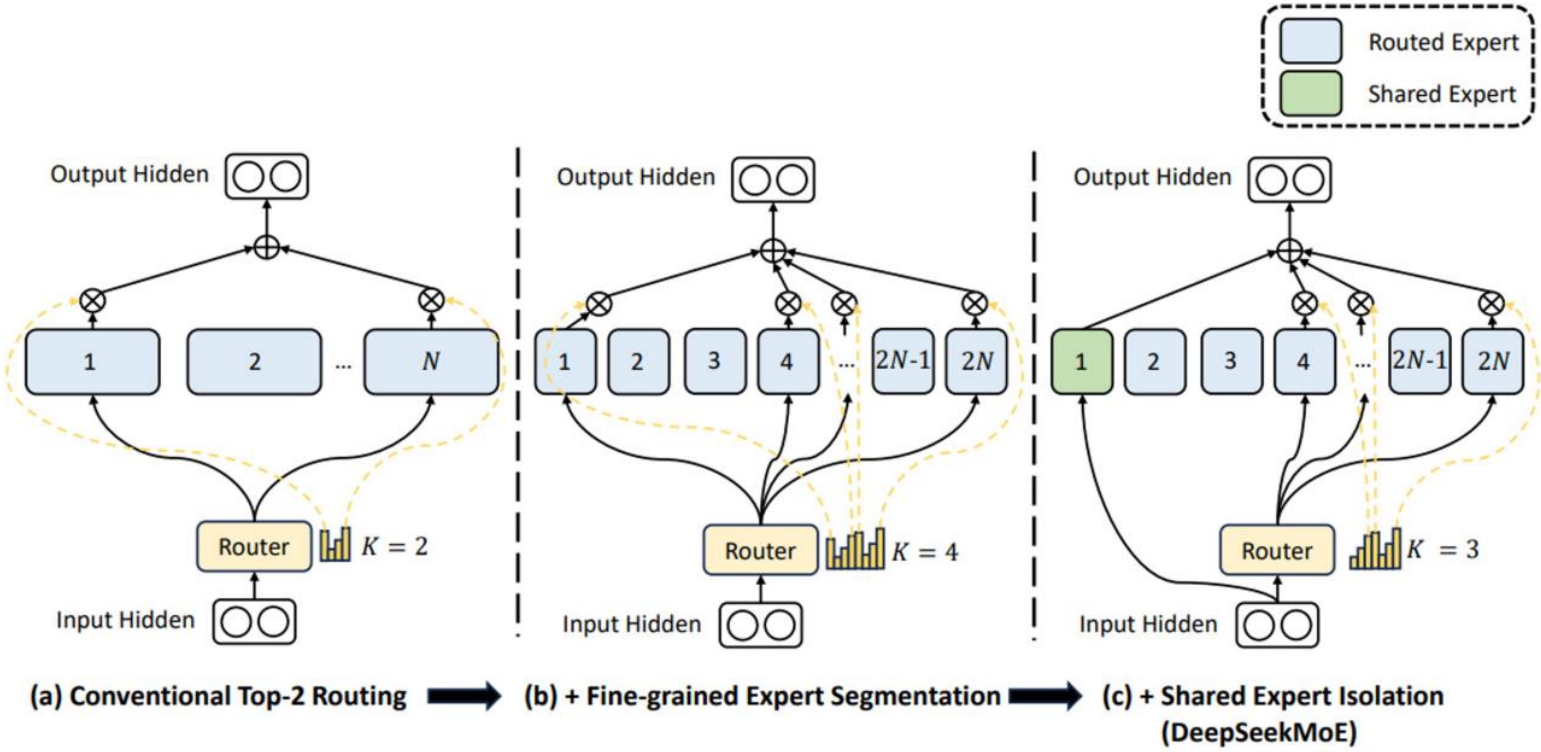
Training with different auxiliary loss:

Reference	Auxiliary Loss	Coefficient
Shazeer et al.[135], V-MoE[128]	$L_{importance} + L_{load}$	$w_{importance} = 0.1, w_{load} = 0.1$
GShard[86], **Switch-T**[49], GLaM[44], Mixtral-8x7B[74], DBRX[34], Jamba[94], DeepSeekMoE[30], DeepSeek-V2[36], Skywork-MoE[154]	L_{aux}	$w_{aux} = 0.01$
ST-MoE[197], OpenMoE[172], MoA[182], JetMoE [139]	$L_{aux} + L_z$	$w_{aux} = 0.01, w_z = 0.001$
Mod-Squad[21], Moduleformer[140], DS-MoE[117]	L_{MI}	$w_{MI} = 0.001$

- Importance loss: encourages all experts to have equal importance
- Load loss: ensure balanced loads
- Auxiliary loss: mitigating load balance losses
- Z-loss: improving training stability by penalizing large logits
- MI-loss: mutual information (MI) between experts and tasks to build task-expert alignment

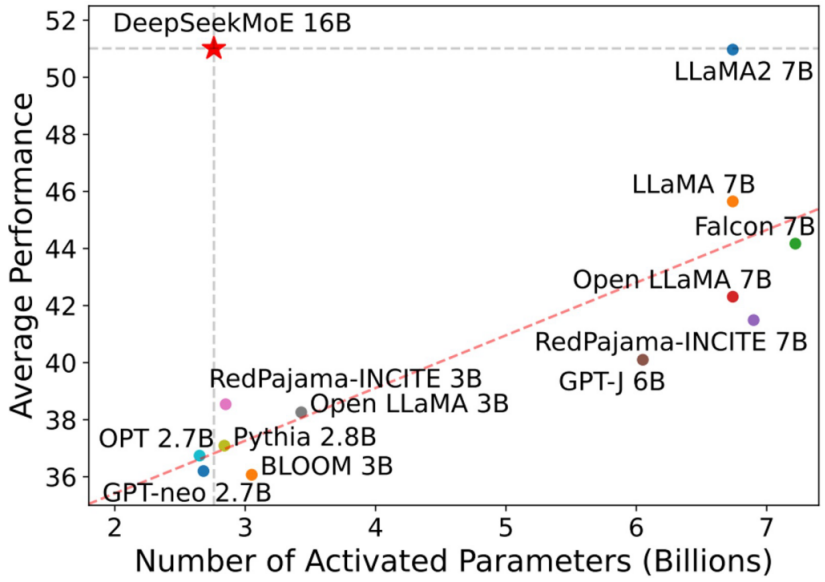
Training MoE - Deepseek (example)

Deepseek-MoE 16B, total 16.4B parameters, 2.8B activated parameters.
Each MoE layer consists of 2 shared experts and 64 routed experts (select 6 experts).



- Most recent models place MoE each layer.
- Some of recent models apply Shared experts.

Training MoE - Deepseek (example)



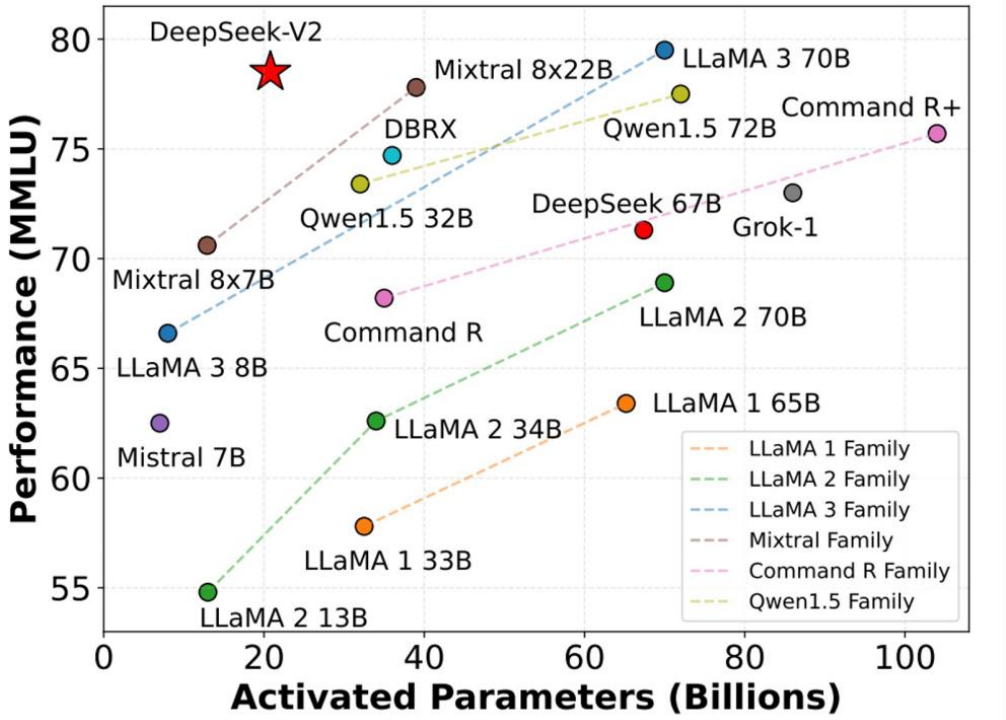
Metric	# Shot	DeepSeek 7B (Dense)	DeepSeekMoE 16B
# Total Params	N/A	6.9B	16.4B
# Activated Params	N/A	6.9B	2.8B
FLOPs per 4K Tokens	N/A	183.5T	74.4T
# Training Tokens	N/A	2T	2T
Pile (BPB)	N/A	0.75	**0.74**
HellaSwag (Acc.)	0-shot	75.4	**77.1**
PIQA (Acc.)	0-shot	79.2	**80.2**
ARC-easy (Acc.)	0-shot	**67.9**	**68.1**
ARC-challenge (Acc.)	0-shot	48.1	**49.8**
RACE-middle (Acc.)	5-shot	**63.2**	61.9
RACE-high (Acc.)	5-shot	**46.5**	**46.4**
DROP (EM)	1-shot	**34.9**	32.9
GSM8K (EM)	8-shot	17.4	**18.8**
MATH (EM)	4-shot	3.3	**4.3**
HumanEval (Pass@1)	0-shot	26.2	**26.8**
MBPP (Pass@1)	3-shot	**39.0**	**39.2**
TriviaQA (EM)	5-shot	59.7	**64.8**
NaturalQuestions (EM)	5-shot	22.2	**25.5**
MMLU (Acc.)	5-shot	**48.2**	45.0
WinoGrande (Acc.)	0-shot	**70.5**	**70.2**
CLUEWSC (EM)	5-shot	**73.1**	72.1
CEval (Acc.)	5-shot	**45.0**	40.6
CMMLU (Acc.)	5-shot	**47.2**	42.5
CHID (Acc.)	0-shot	**89.3**	**89.4**

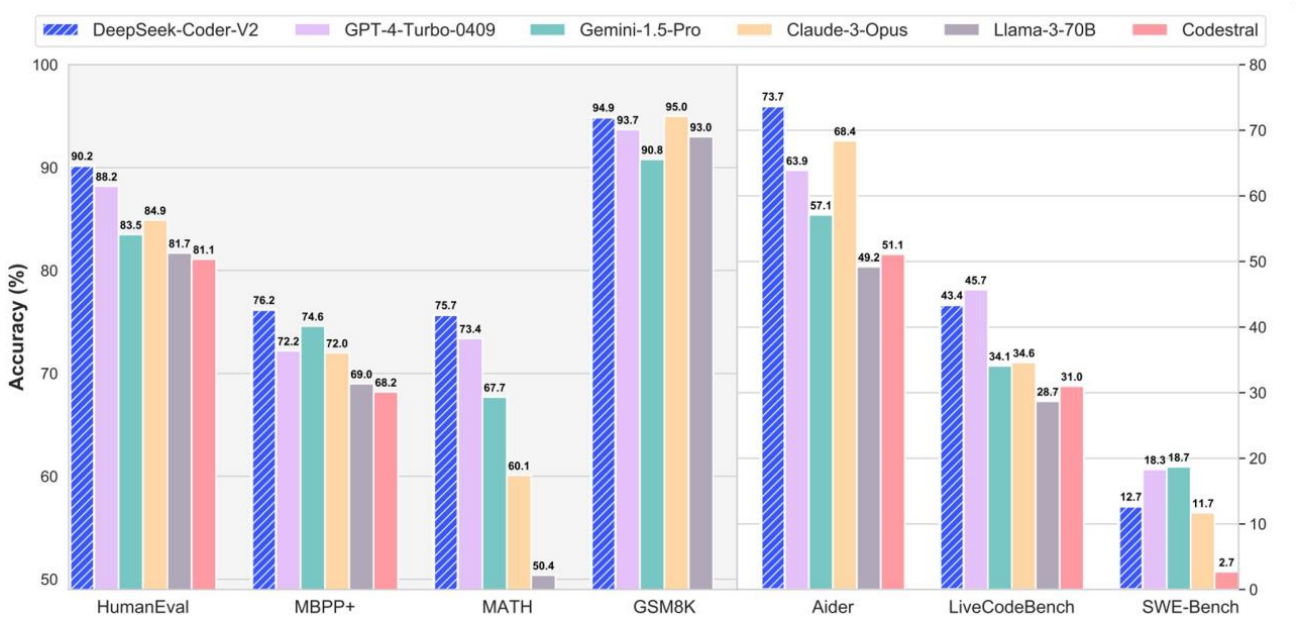
DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models

Training MoE - Deepseek (example)

Deepseek-V2
236B total parameters, 21B are activated.
2 shared experts and 160 routed experts (6 select).

Deepseek-Coder-V2
Continue pretraining from an intermediate checkpoint of Deepseek-V2 (4.2T) and further train 6T. Total 10.2T tokens.





Sparse Upcycling - Qwen-MoE

Qwen1.5-MoE-A2.7B (Mar, 2024)
Upcycled from Qwen-1.8B, 14.3B parameters in total and 2.7B activated parameters.

- Fine-grained experts (total 64 experts)
- use shared (4 experts) and routing experts (60 experts, choose 4)

Model	MMLU	GSM8K	HumanEval	Multilingual	MT-Bench
Mistral-7B	64.1	47.5	27.4	40.0	7.60
Gemma-7B	64.6	50.9	32.3	-	-
Qwen1.5-7B	61.0	62.5	36.0	45.2	7.60
DeepSeekMoE 16B	45.0	18.8	26.8	-	6.93
Qwen1.5-MoE-A2.7B	62.5	61.5	34.2	40.8	7.17

A remarkable reduction of 75% in training

3 Model Compression

A trained model is redundant. Store it in fewer bits, remove what is unused, or teach a smaller model to imitate it.

In this part

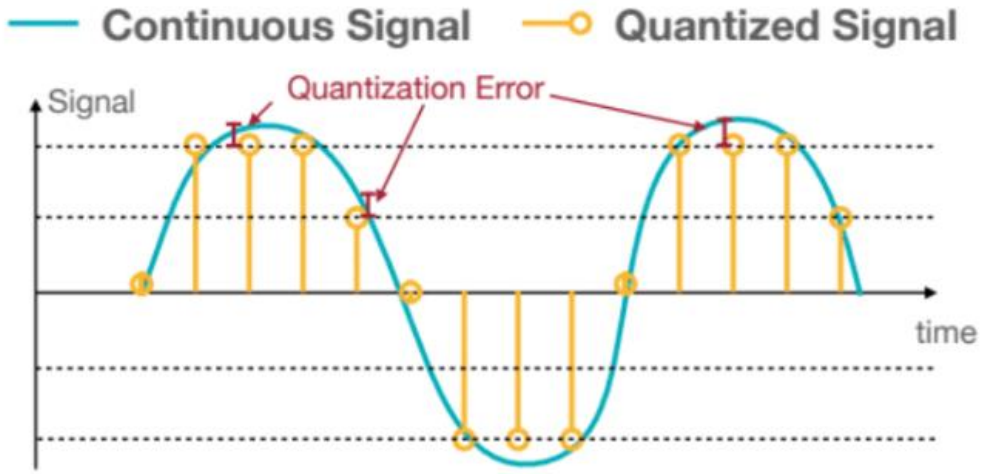
3.1 Quantization

3.2 Pruning

3.3 Knowledge Distillation

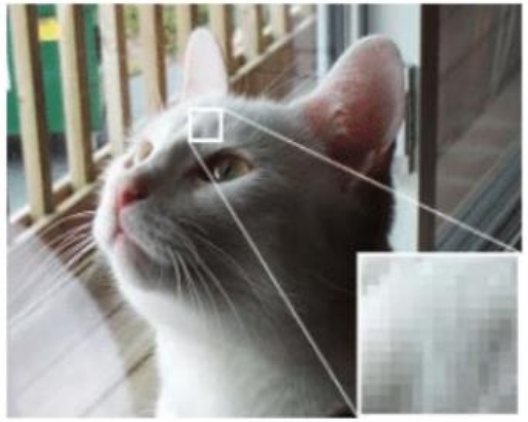
Quantization - What is Quantization?

Quantization is the process of constraining an input from a continuous or otherwise large set of values to a discrete set.

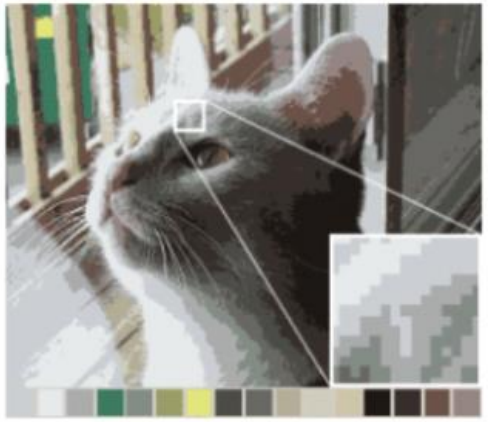


The difference between an input value and its quantized value is referred to as quantization error.

Original Image



16-Color Image

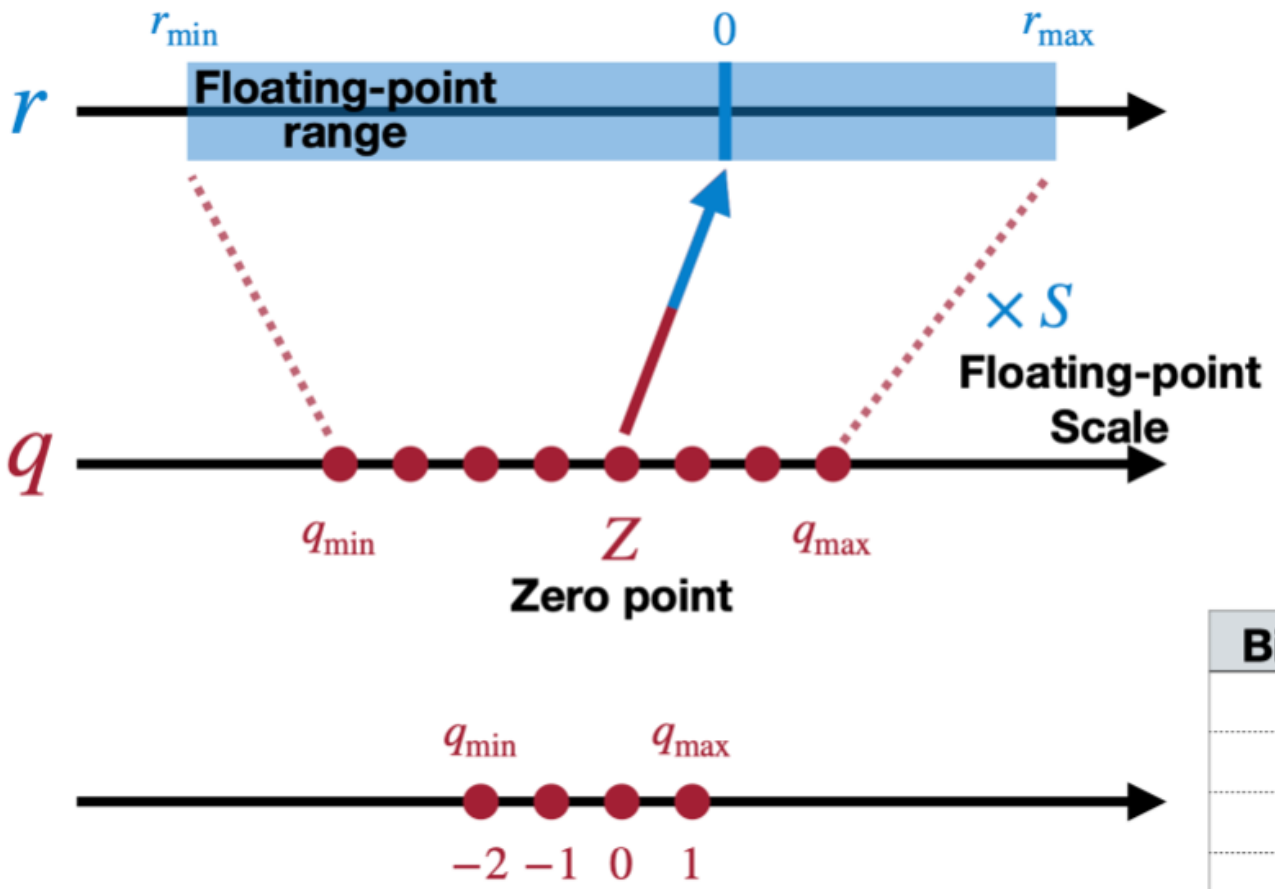


Images are in the public domain.

"Palettization"

Quantization [Wikipedia]

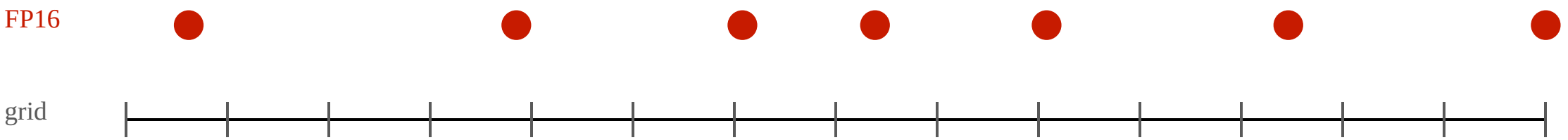
Quantization



Binary	Decimal
01	1
00	0
11	-1
10	-2

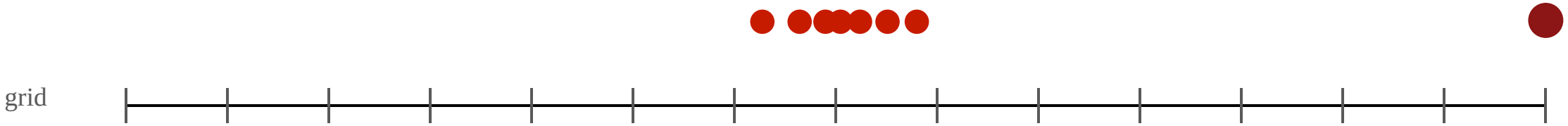
Quantization: Snap Every Weight to a Small Grid

INT4 example: 16 levels. scale = max|w| / 7, q = round(w / scale)



each weight moves by at most half a step: small rounding error, 4× fewer bytes than FP16

One outlier (w = 8.0) stretches the scale: all normal weights collapse onto 1–2 levels

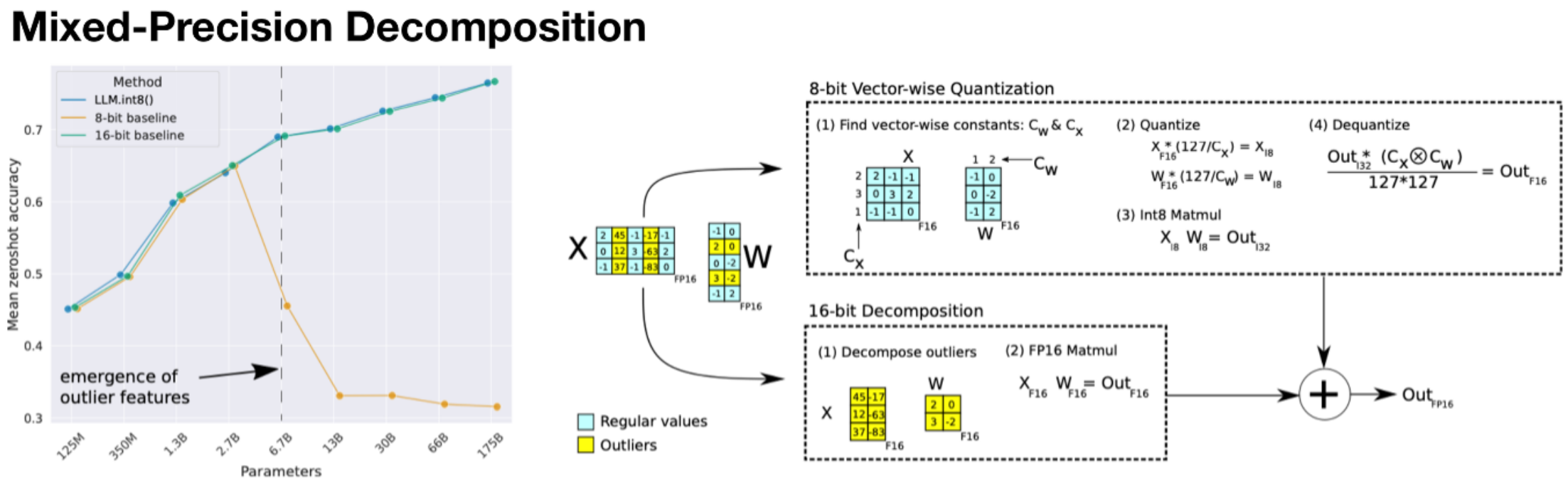


Fixes: keep outlier channels in FP16 (LLM.int8()) · smaller groups with their own scale · protect salient weights (AWQ) · compensate errors (GPTQ)

Today: INT4 weights for local models; FP8 training (DeepSeek-V3); FP4 (NVFP4) inference on new GPUs.

Dettmers et al. LLM.int8() (2022) · Frantar et al. GPTQ (2023) · Lin et al. AWQ (2024) · DeepSeek-V3 (2024).

Quantization - LLM.int8()



- **Motivation**: Transformers have outlier features that have **large values** (especially large models).
  - They occur in particular hidden dimensions, leading to large quantization error.
- **Key idea**: Separate outlier features into a **separate FP16 MM**, quantize the other values to Int8.
  - Outlier: At least one feature dimension with a magnitude larger than the threshold (6).
  - Token-wise scale factor (for X) and (output) channel-wise scale factor (for W).

LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale [Dettmers *et al.*, 2022]

Mixed Precision Training

- Use lower-precision floating point to reduce memory and speed up computation

Common formats:

- FP32 (32-bit): full precision, baseline
- FP16 (16-bit): 2x memory savings, but overflow/underflow issues
- BF16 (16-bit): same range as FP32 with reduced precision, stable training
- FP8 (8-bit): emerging for H100/B200, used in DeepSeek-V3 training

Mixed precision training (Micikevicius et al., 2018):

- - Maintain FP32 master weights, compute forward/backward in FP16/BF16
- - Loss scaling to prevent gradient underflow

Almost all modern LLM training uses BF16 by default

FP8 training is becoming standard on latest hardware (H100, B200), e.g. DeepSeek R1

Post-Training Quantization for LLMs

GPTQ (Frantar et al., 2023):

- - One-shot weight quantization using approximate second-order information
- - INT4/INT3 with minimal accuracy loss
- - Widely used: AutoGPTQ, ExLlama

AWQ (Lin et al., 2024): Activation-Aware Weight Quantization

- - Key insight: not all weights are equally important
- - Protect salient weight channels (determined by activation magnitude)
- - Better accuracy than GPTQ at same bit-width

GGUF (llama.cpp):

- - CPU-friendly quantization format (Q4_K_M, Q5_K_M, etc.)
- - Powers local LLM inference on consumer hardware
- - Supports mixed quantization across layers

Quantization Methods Comparison

- Method | Bits | Speed | Quality | Use Case
- ---|---|---|---|---
- FP16/BF16 | 16 | 1x | Baseline | Training & serving
- LLM.int8() | 8 | ~1x | Lossless | GPU inference
- GPTQ | 4/3 | 1.5-2x | Good | GPU inference
- AWQ | 4 | 1.5-2x | Better | GPU inference
- GGUF (Q4_K_M) | 4-5 | Varies | Good | CPU/local inference

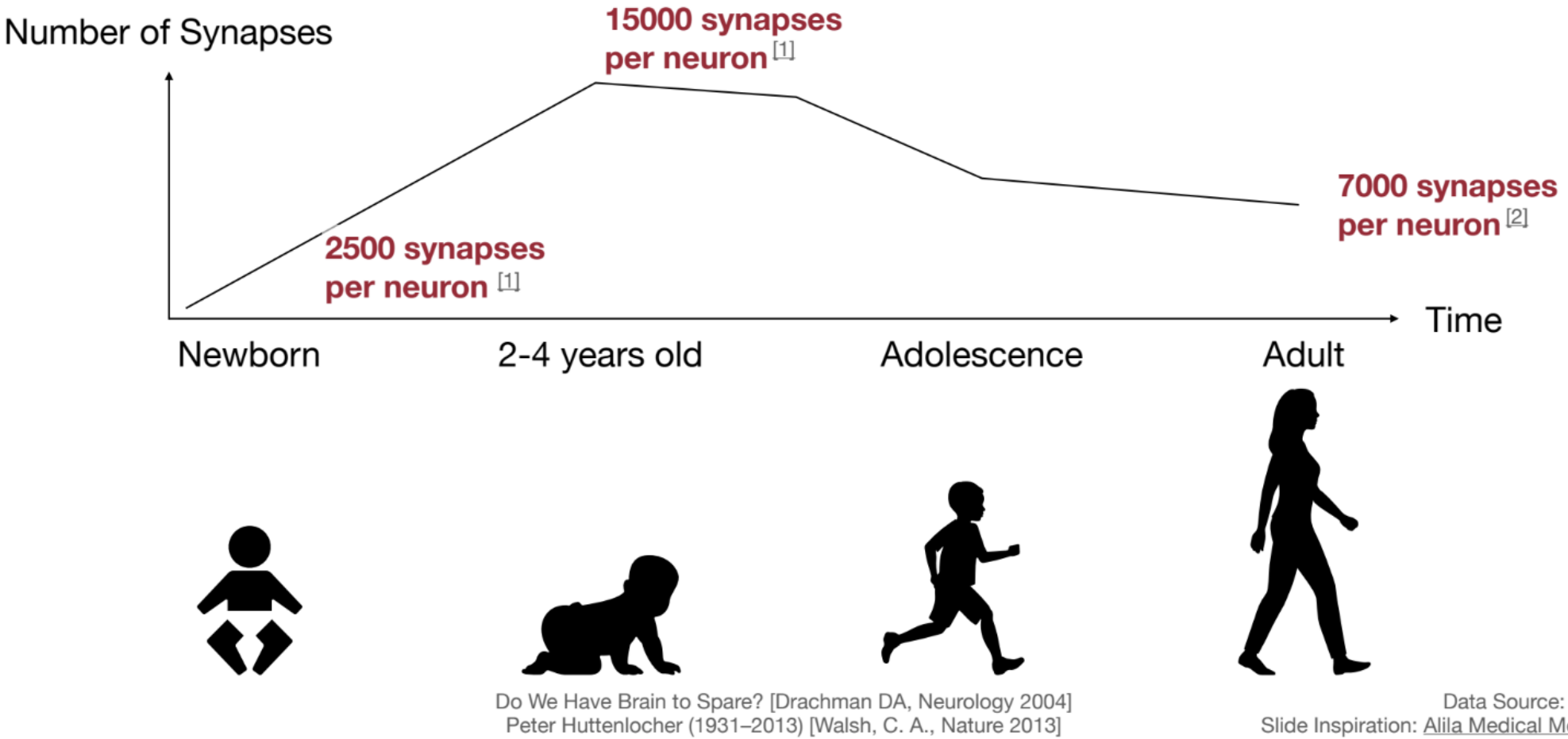
- Rule of thumb:
- - INT8: almost no quality loss, 2x memory savings
- - INT4: small quality loss, 4x memory savings
- - Below INT4: significant degradation for most models

3.2 Pruning

Many weights barely matter. Remove them — one by one, or whole heads, neurons and layers.

Neural Network Pruning - What is Pruning?

Pruning happens in human brains (synapses vs. neurons)

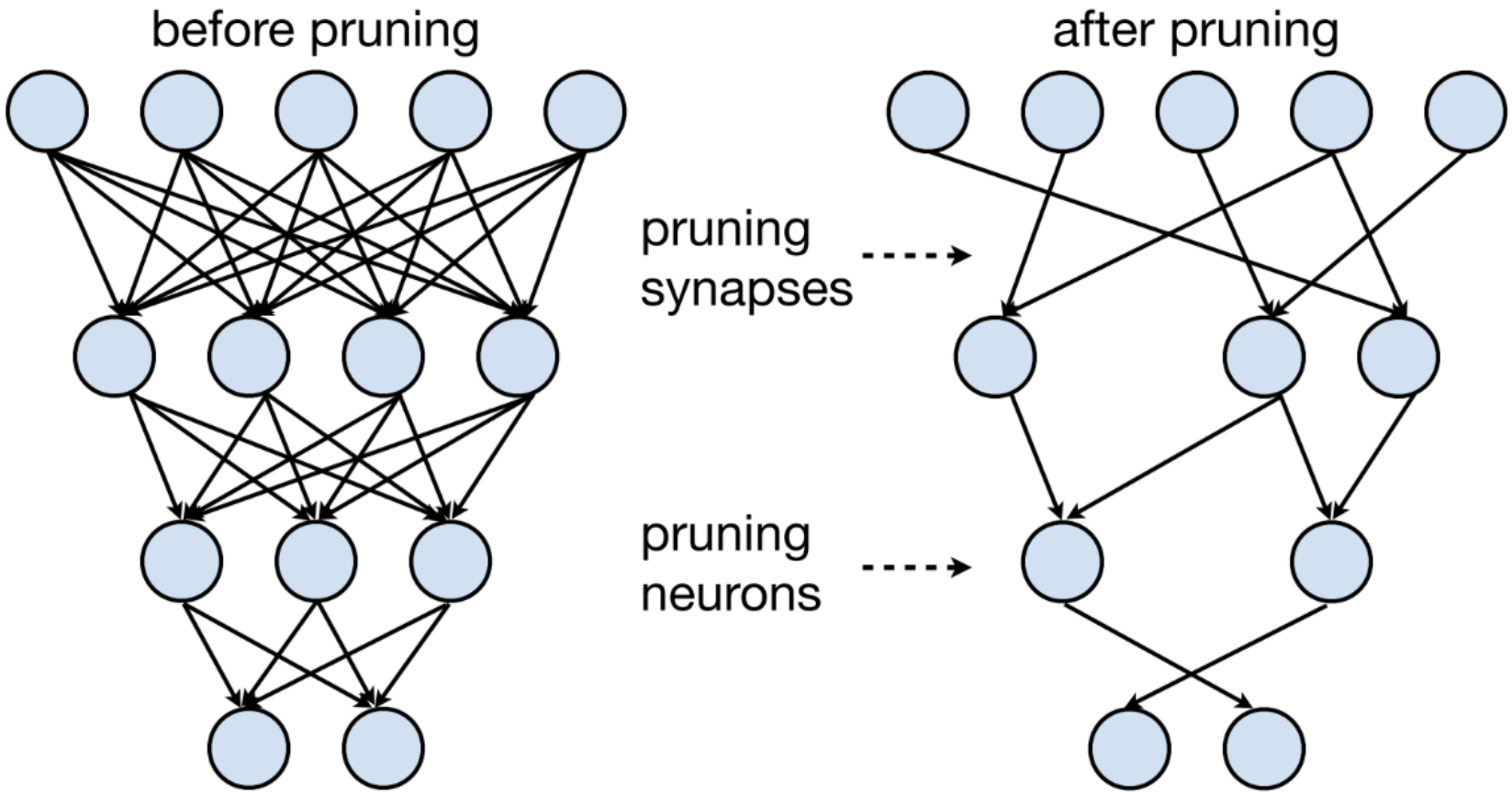


Do We Have Brain to Spare? [Drachman DA, Neurology 2004]
Peter Huttenlocher (1931–2013) [Walsh, C. A., Nature 2013]

Data Source: 1, 2
Slide Inspiration: Alila Medical Media

Neural Network Pruning - What is Pruning?

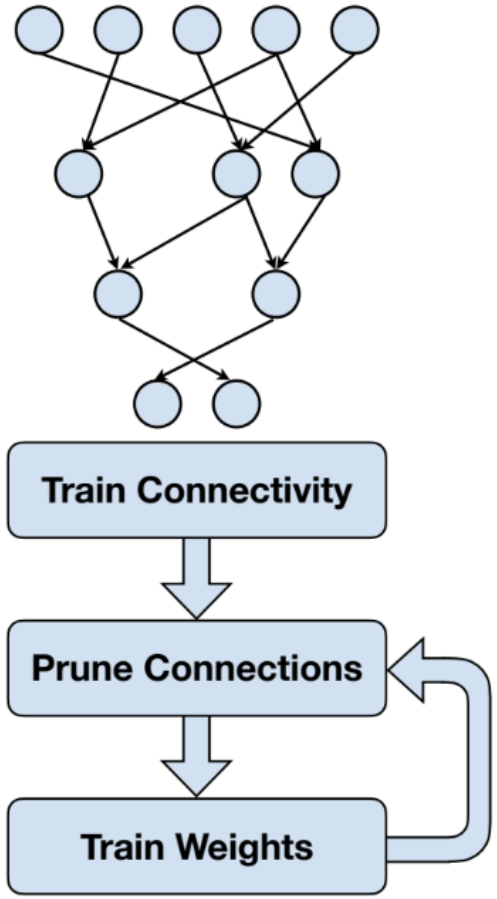
Make neural network smaller by removing synapses and neurons

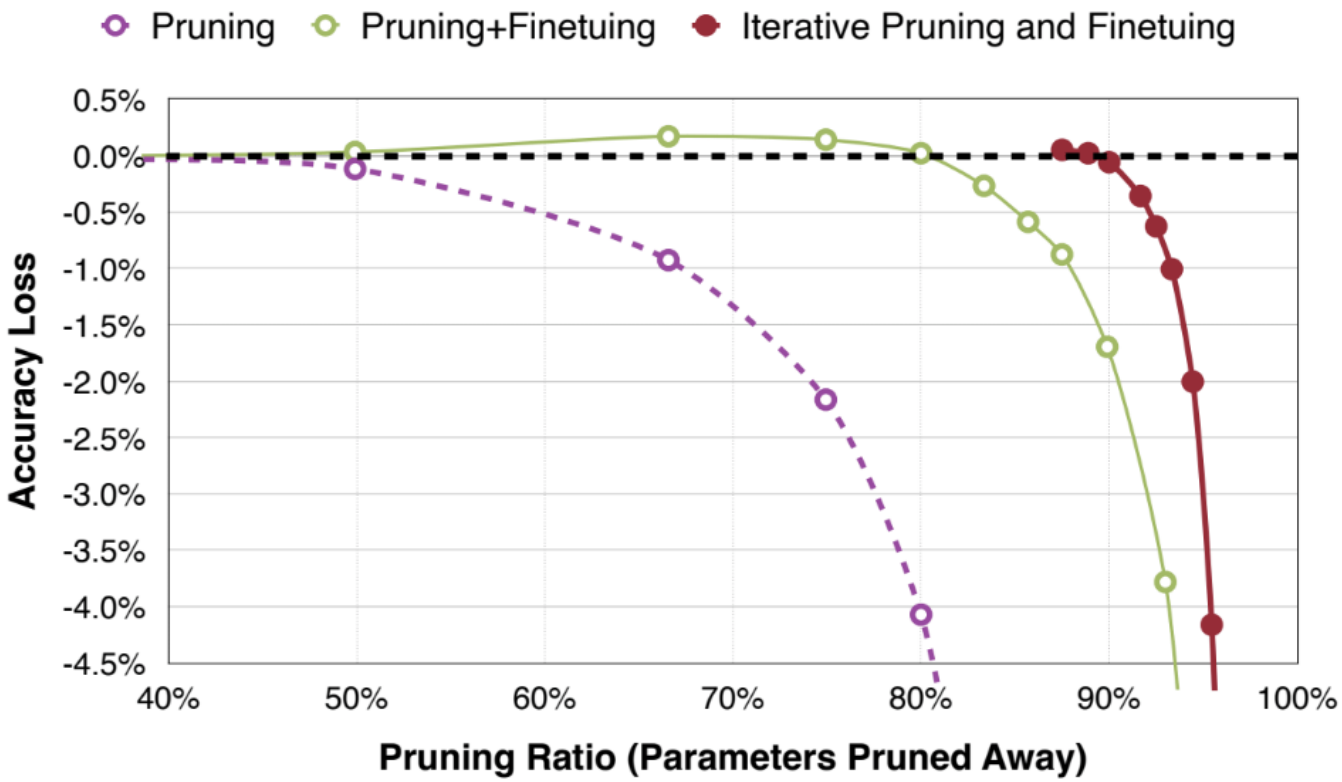


Optimal Brain Damage [LeCun *et al.*, NeurIPS 1989]
Learning Both Weights and Connections for Efficient Neural Network [Han *et al.*, NeurIPS 2015]

Neural Network Pruning - What is Pruning?

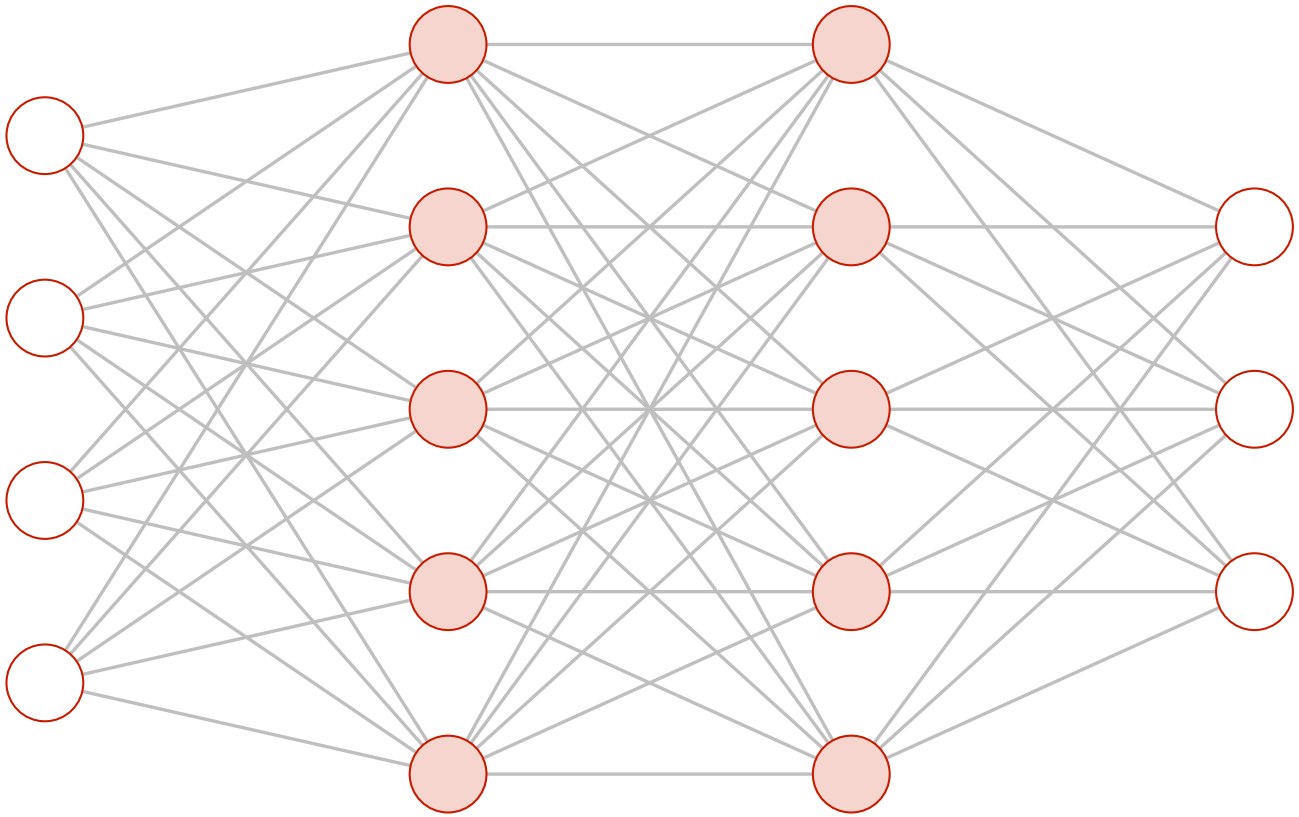
Make neural network smaller by removing synapses and neurons





Learning Both Weights and Connections for Efficient Neural Network [Han *et al*., NeurIPS 2015]

Pruning: Remove What Does Not Matter



① Unstructured
drop individual small-magnitude weights → sparse matrices

② Structured
drop whole neurons, heads, layers → a smaller dense model, real speedup

③ Recover
briefly continue training (e.g. LLM-Shearing: prune then continue pre-training)

Han et al. Deep Compression (2016) · Frankle & Carbin. Lottery Ticket Hypothesis (2019) · Xia et al. Sheared LLaMA (2024).

Neural Network Pruning - How should we formulate pruning

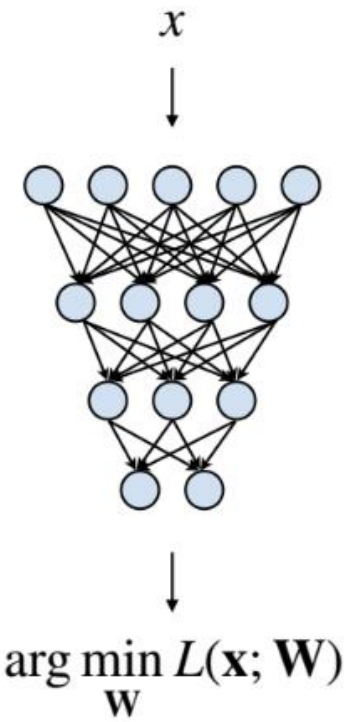
- In general, we could formulate the pruning as follows:

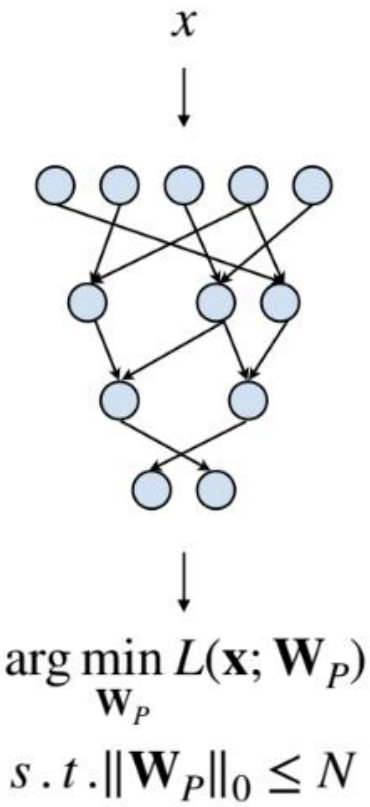
$$\arg\min_{\mathbf{W}_P} L(\mathbf{x}; \mathbf{W}_P)$$

subject to

$$\|\mathbf{W}_p\|_0 < N$$

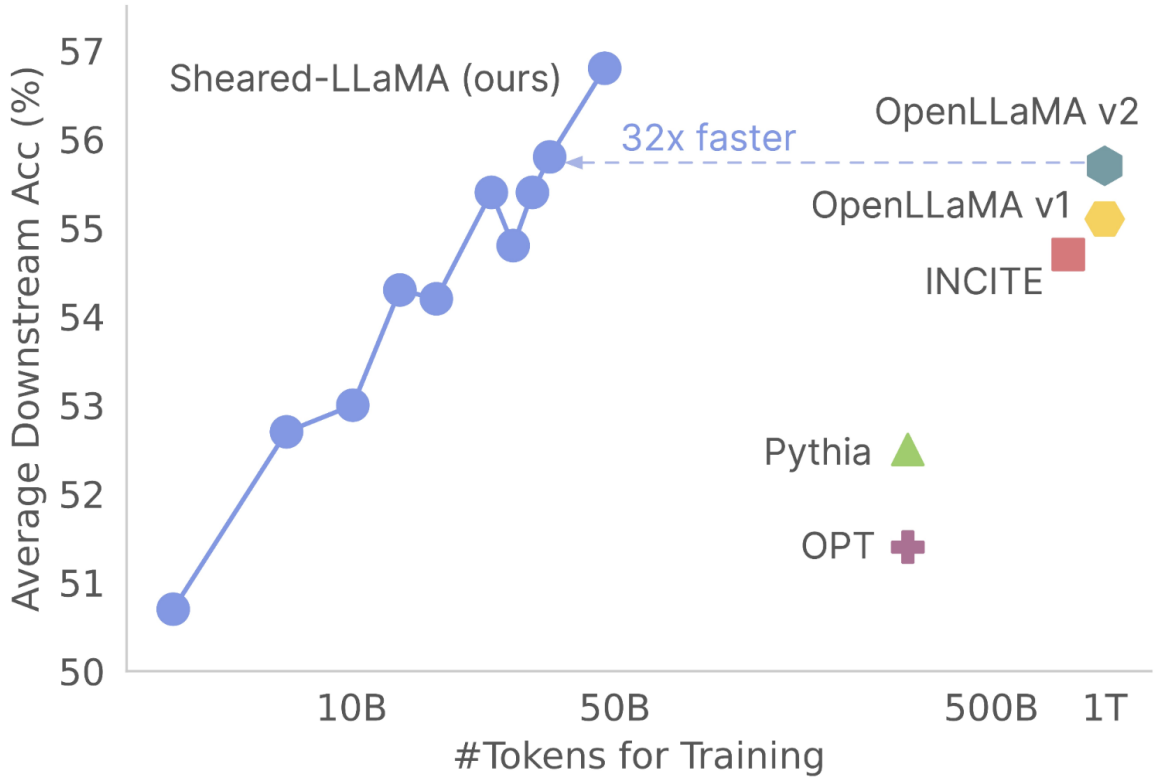
  - L represents the objective function for neural network training;
  - $\mathbf{x}$ is input, $\mathbf{W}$ is original weights, $\mathbf{W}_P$ is pruned weights;
  - $\|\mathbf{W}_p\|_0$ calculates the #nonzeros in W_P, and N is the target #nonzeros.





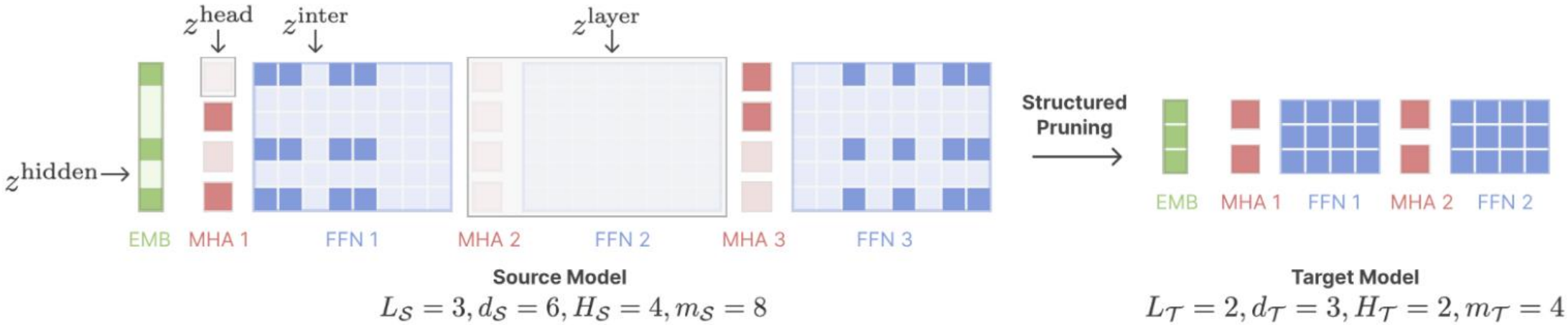
LLM-Shearing: Accelerating via Structured Pruning

An efficient method of constructing LLMs by first pruning a larger existing model and then continually pre-training it.



- Sheared-LLaMA-2.7B achieves better performance than existing open-source models of the same scale with 3% (1/32) of the compute.
- The trajectory shows a compelling case that if we invest more tokens and compute, the capability of Sheared-LLaMA can be further improved.

LLM-Shearing: Accelerating via Structured Pruning



1. Target Structure Pruning: prune a source model to to a pre-specified target architecture (e.g., an existing model's config), and meanwhile maximizing the pruned model’s performance

LLM-Shearing: Accelerating via Structured Pruning

Algorithm 1: Dynamic Batch Loading

Require: Training data of k domains $D_1, D_2, \cdots, D_k$, validation data $D_1^{\text{val}}, D_2^{\text{val}}, \cdots, D_k^{\text{val}}$, initial data loading weights $w_0 \in \mathbb{R}^k$, reference loss $\ell_{\text{ref}} \in \mathbb{R}^k$, LM loss function $\mathcal{L}$ or pruning loss $\mathcal{L}_{\text{prune}}$, training steps T, evaluation interval m, model parameters θ (θ, z, ϕ, λ for pruning)

for $t = 1, \cdots, T$ **do**
  if $t \mod m = 0$ **then**
    $\ell_t[i] \leftarrow \mathcal{L}(\theta, z, D_i^{\text{val}})$ if *pruning* else $\mathcal{L}(\theta, D_i^{\text{val}})$
    $\Delta_t[i] \leftarrow \max\{\ell_t[i] - \ell_{\text{ref}}[i], 0\}$ ▷ Calculate loss difference
    $w_t \leftarrow \texttt{UpdateWeight}(w_{t-m}, \Delta_t)$ ▷ Update data loading proportion
  end
  Sample a batch of data $\mathcal{B}$ from $D_1, D_2, \cdots, D_k$ with proportion w_t;
  if *pruning* **then**
    Update θ, z, ϕ, λ with $\mathcal{L}_{\text{prune}}(\theta, z, \phi, \lambda)$ on $\mathcal{B}$
  else
    Update θ with $\mathcal{L}(\theta, \mathcal{B})$
  end
end

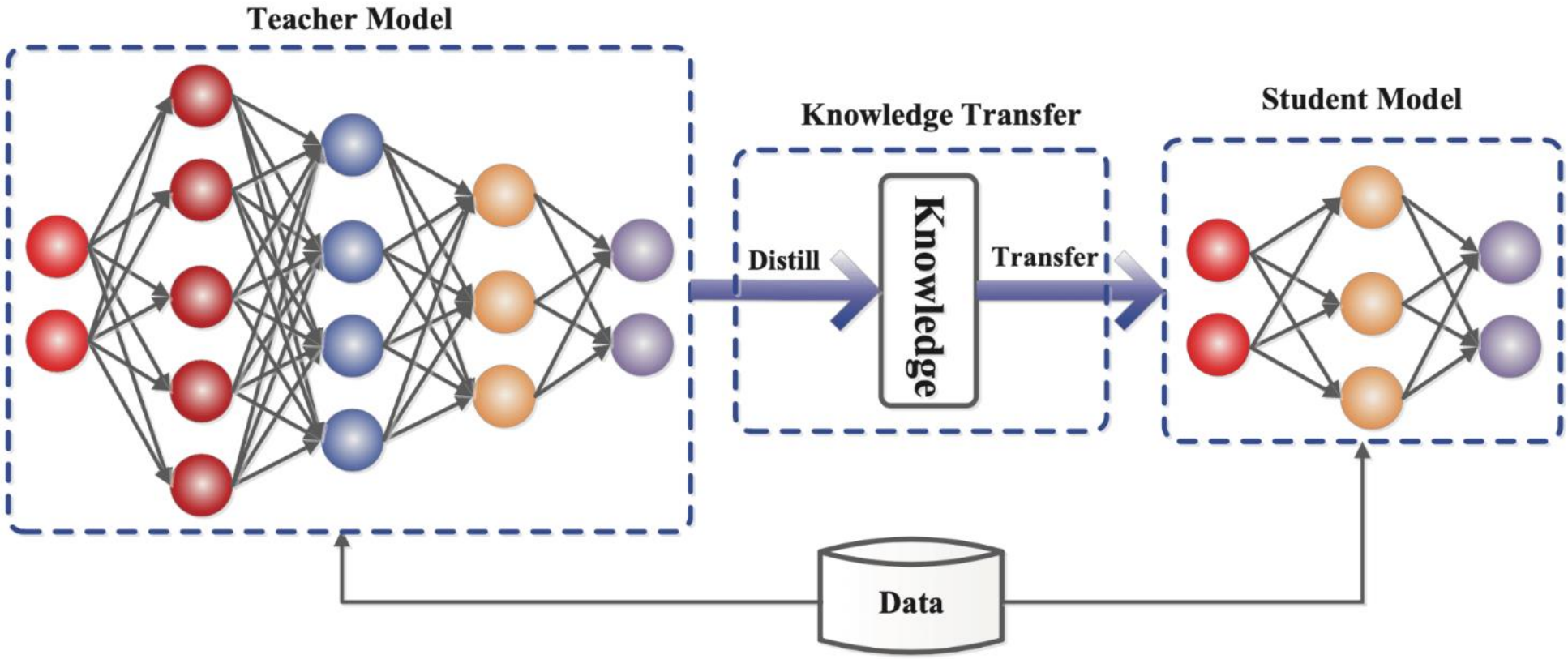
Subroutine `UpdateWeight`(w, Δ)
  $\alpha \leftarrow w \cdot \exp(\Delta)$ ▷ Calculate the unnormalized weights
  $w \leftarrow \frac{\alpha}{\sum_i \alpha[i]}$ **return** w ▷ Renormalize the data loading proportion
return θ

2. Dynamic batch loading: Pruning results in varying information retainment across domains. Concretely, they load more data for domains that recover slow, and the loading proportion is dynamically decided on the fly.

3.3 Knowledge Distillation

A big teacher already knows the task. Let a small student learn from the teacher's outputs.

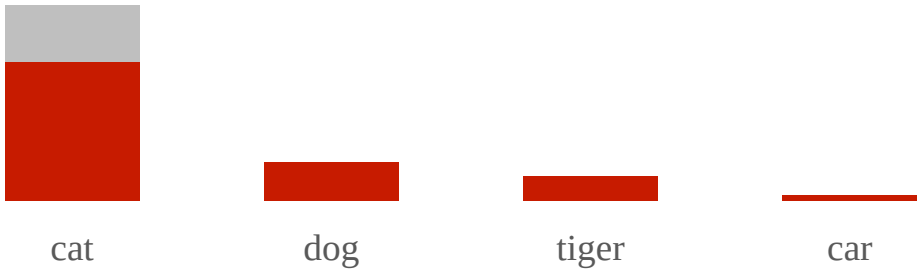
Framework of knowledge distillation



Knowledge Distillation: Learn from the Teacher's Soft Labels

Input: a photo of a cat

Teacher (large)



Hard label: "cat"

Soft labels: "mostly cat, a bit like a dog or a tiger"

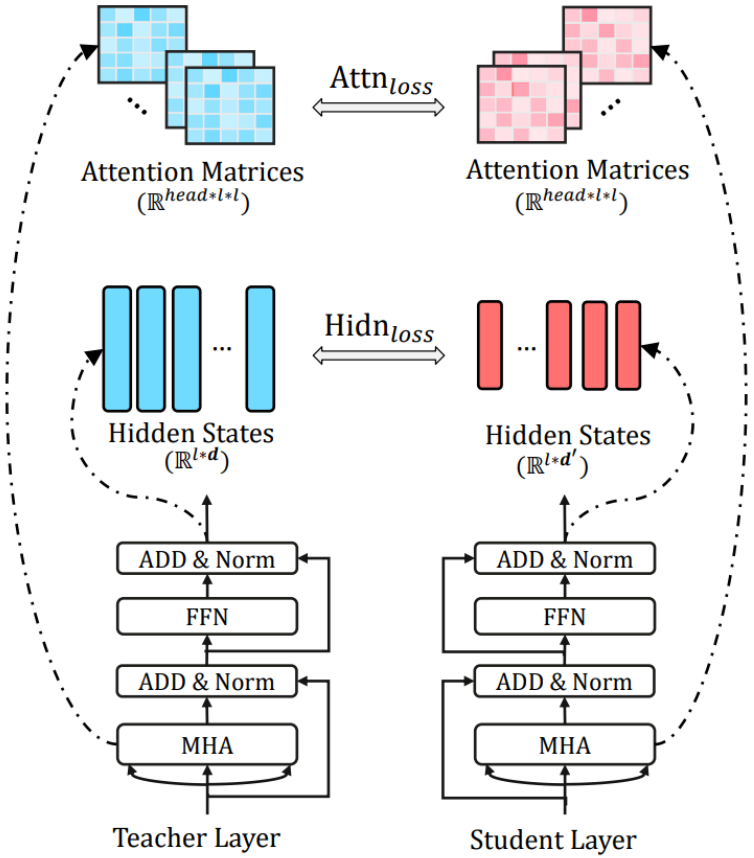
Student (small)

Loss = CE(student, hard label) + λ · KL(teacher_T ‖ student_T) (temperature T softens both)

For LLMs: match logits (MiniLLM), hidden states (TinyBERT) — or simply train on the teacher's generated text: data distillation (Alpaca, DeepSeek-R1-Distill).

Hinton et al. Distilling the Knowledge in a Neural Network (2015) · Jiao et al. TinyBERT (2020) · Gu et al. MiniLLM (2024).

TinyBERT



In transformer, it would be nice to learn attentions from teacher model.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li , Fang Wang and Qun Liu. TinyBERT: Distilling BERT for Natural Language Understanding.
https://arxiv.org/pdf/1909.10351.pdf

Benefits of KD compared to directly training

More fine-grained supervision learn from teachers' hidden states in every layers Making use of unannotated data (teacher model provide supervision) Data augmen

A few work of KD in LLMs

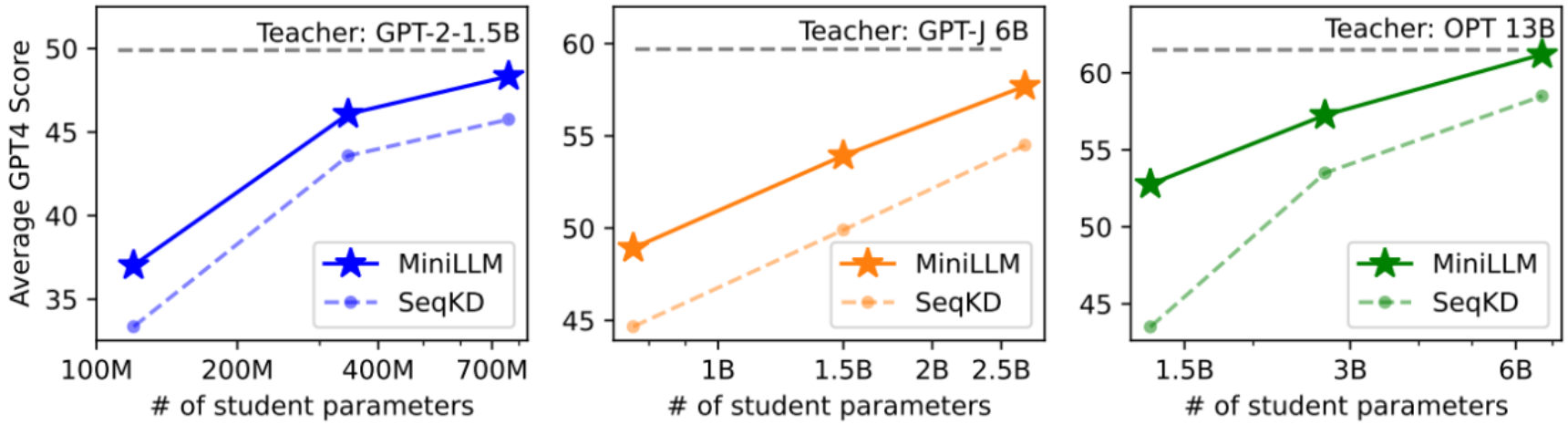


Figure 1: The comparison of MINILLM with the sequence-level KD (SeqKD) in terms of the average GPT-4 feedback score on our evaluation sets. **Left**: GPT-2-1.5B as the teacher and GPT-2 125M, 340M, 760M as the students. **Middle**: GPT-J 6B as the teacher and GPT-2 760M, 1.5B, GPT-*Neo* 2.7B as the students. **Right**: OPT 13B as the teacher and OPT 1.3B, 2.7B, 6.7B as the students.

Traditional KD has limited success for LLMs, but data distillation (using large models to generate training data for smaller ones) has proven highly effective [See GLM, KIMI, Minimax]

Yuxian Gu, Li Dong , Furu Wei , Minlie Huang. Knowledge Distillation of Large Language Models. https://arxiv.org/pdf/2306.08543.pdf

Data Distillation: A New Paradigm of KD for LLMs

Key idea: use a powerful teacher LLM to generate training data for smaller student models

- Stanford Alpaca (2023): GPT-3.5 generates 52K instruction-following examples to train LLaMA-7B
- Phi Series (Microsoft): GPT-4 generates "textbook-quality" data to train small models (1.3B-14B)
- Phi-4 (14B) rivals much larger models on reasoning benchmarks
- Orca (Microsoft): learns step-by-step reasoning from GPT-4 explanations

Why it works: teacher provides richer supervision than human labels alone

- - Reasoning chains, intermediate steps, diverse examples
- - Effectively transfers "knowledge" without matching architectures

More on Knowledge Distillation

- Beyond traditional KD: the teacher-student boundary is blurring

Self-distillation: model improves by training on its own refined outputs

- Constitutional AI (Anthropic): model critiques and revises its own responses
- Self-play / SPIN: model generates training signal from itself

Cross-model distillation at scale:

- DeepSeek-R1-Distill: distill reasoning from R1 (671B) to Qwen/LLaMA (1.5B-70B)
- NVIDIA Nemotron: synthetic data pipeline for instruction tuning

Takeaway: in the LLM era, KD is more about data than architecture matching

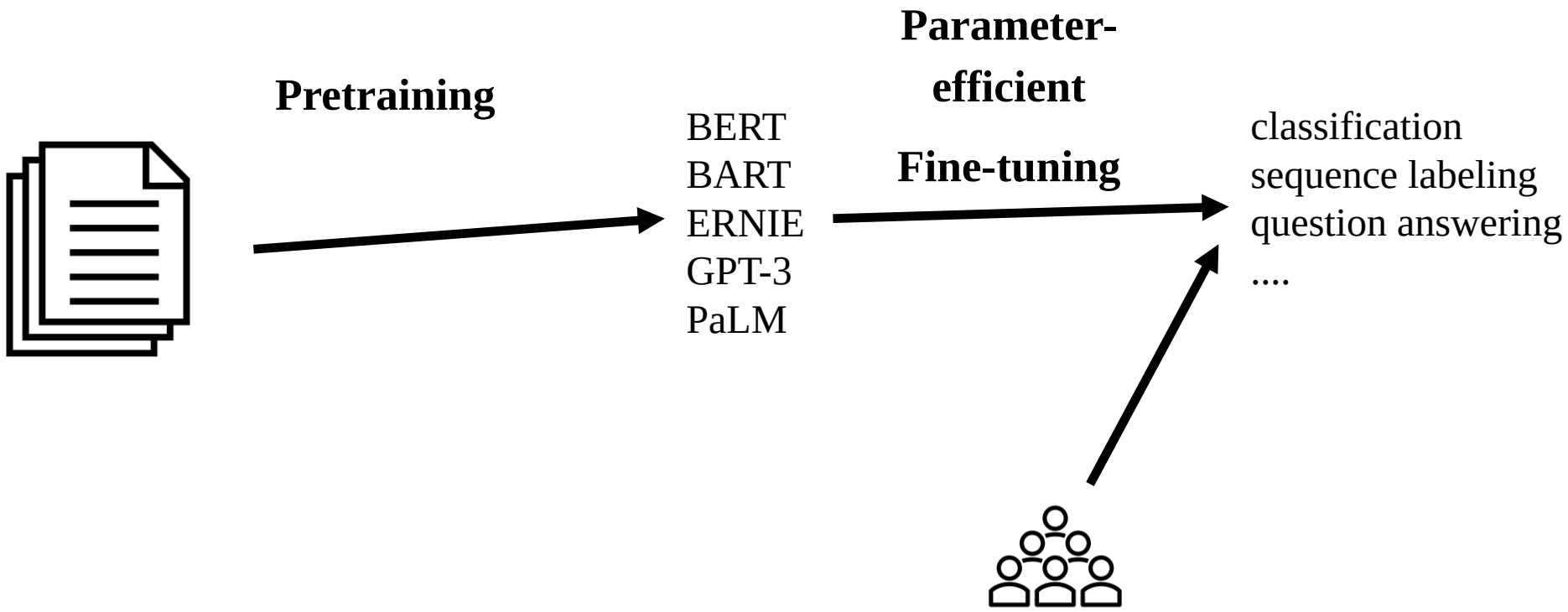
4 Efficient Training

Full fine-tuning and full replication do not fit on GPUs. Train fewer parameters, and split the rest across devices.

In this part

4.1 Parameter-Efficient Fine-Tuning
4.2 Modularity and Composition
4.3 Distributed Training

From Fine-tuning to Parameter-efficient Fine-tuning



Where to Insert the Trainable Parameters

Globally LoRA : Low-rank matrix Locally Adapter: a newly-added later Soft Prompt : “some newly-added fake tokens” Expert in MOE ….

Low-Rank Approximations (LoRA)

- Improve efficiency by leveraging **low-rank** approximations of the self-attention matrix.
- The key idea is to assume **low-rank structure** in the N×N matrix.

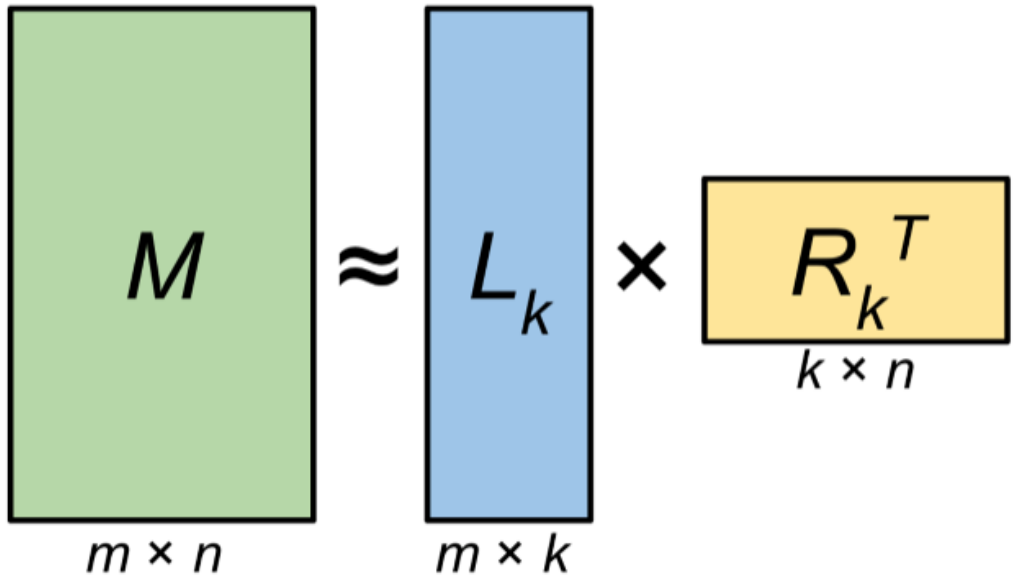
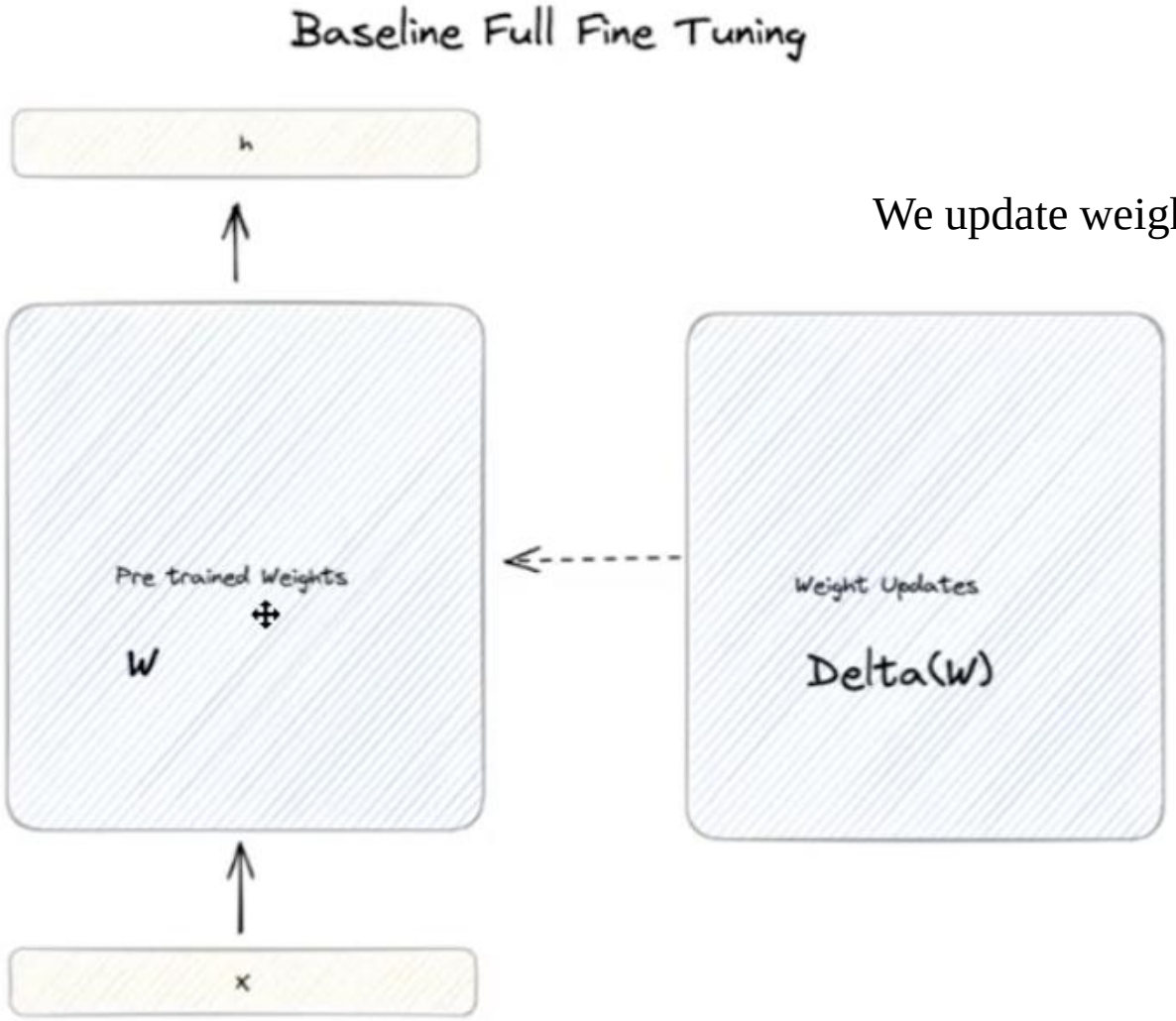


Image credit: https://dustinstansbury.github.io/theclevermachine/assets/images/svd-data-compression/low-rank-approximation.png

Baseline Full Fine-tuning

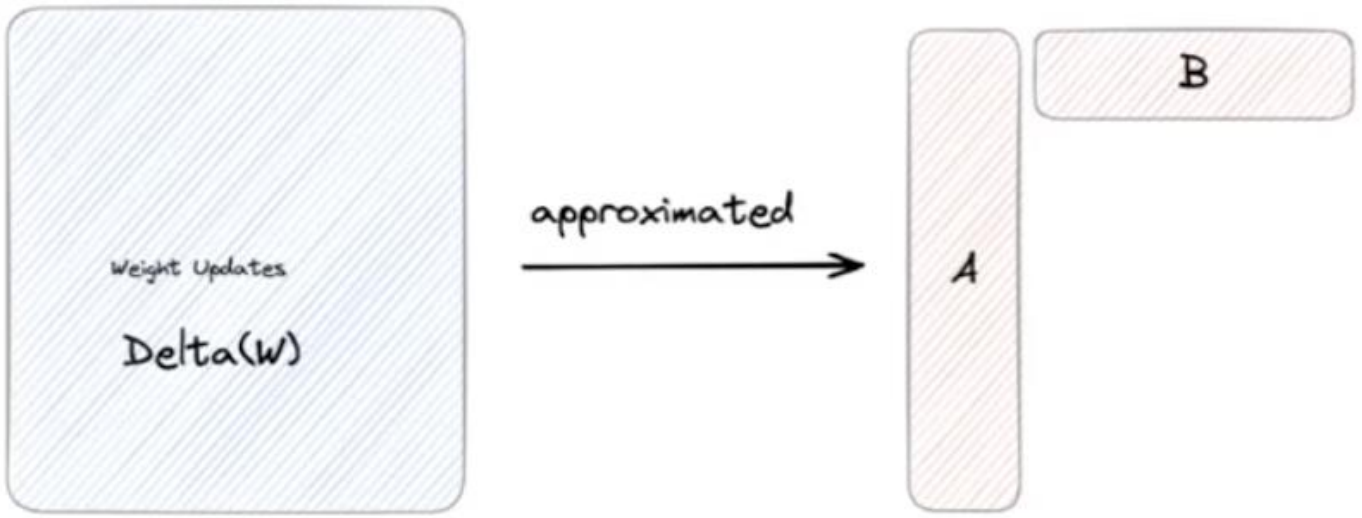


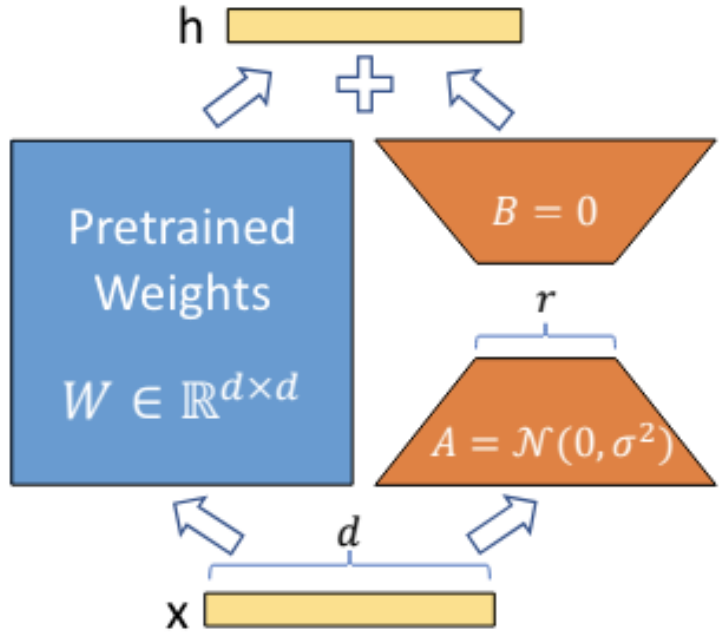
We update weights: W = W + Delta(W)

Problem

Delta(W) is huge

LoRA Tuning



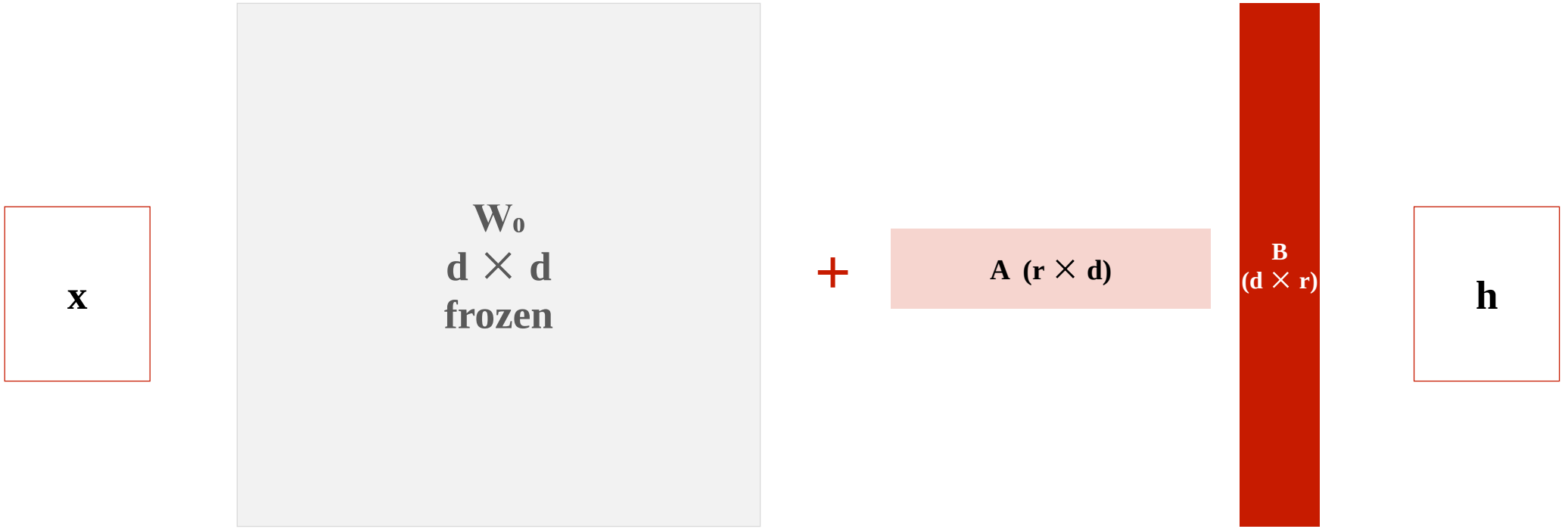


For a pre-trained weights W_0, we approx Delta(w) by B and A:

$h = W_0 x + \Delta W x = W_0 x + BAx$, where $B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}$, and the rank $r \ll \min(d, k)$

During Training, we only compute gradient w.r.t. Delta(W)

LoRA: Freeze W, Learn a Thin Update B·A



$$h = W_0 x + B A x$$

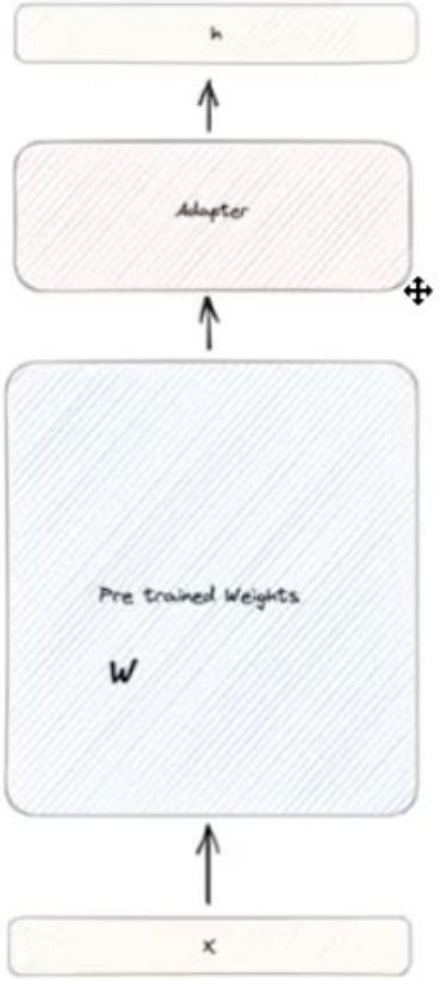
After training: $W = W_0 + BA$ — zero extra inference latency; swap B·A per task.

d = 4096, r = 8: 2dr = 65K vs. d^2 = 16.8M

→ train 0.4% of the weights

Hu et al. LoRA: Low-Rank Adaptation of Large Language Models (ICLR 2022) · Dettmers et al. QLoRA (2023).

Other Parameter-efficient Tunings: Adapter Tuning



We add an adapter module after pre-trained weights W. And during training, we only compute gradient w.r.t. the adapter

Other Parameter-efficient Tunings: Adapter Tuning

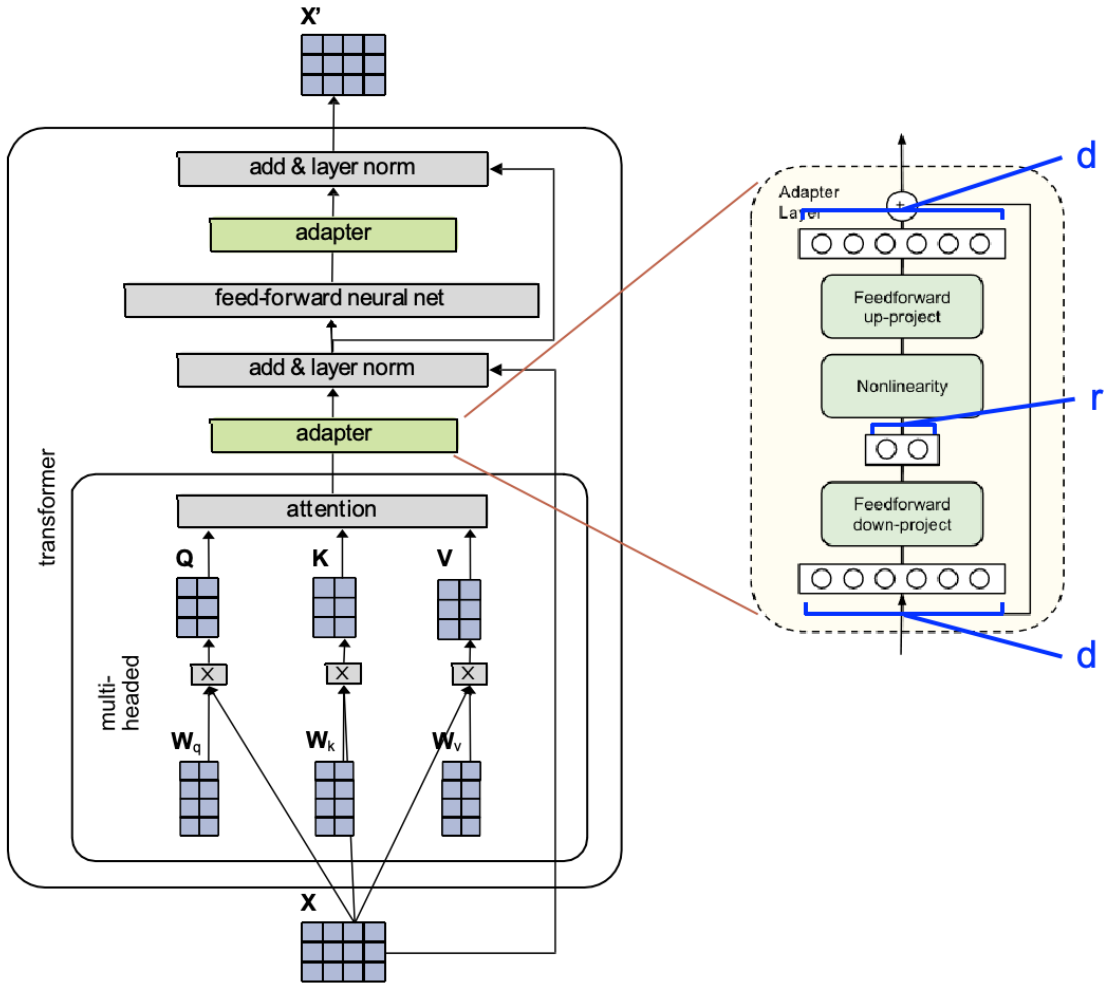


Figure inspired by He et al. (2022) https://arxiv.org/pdf/2110.04366

- An adapter layer is simply a feed- forward neural network with one hidden layer, and a residual connection.
- For input dimension, d, the adapter layer also has output dimension d, but bottlenecks to a lower dimension m in the middle.

Other Parameter-efficient Tunings: Adapter Tuning

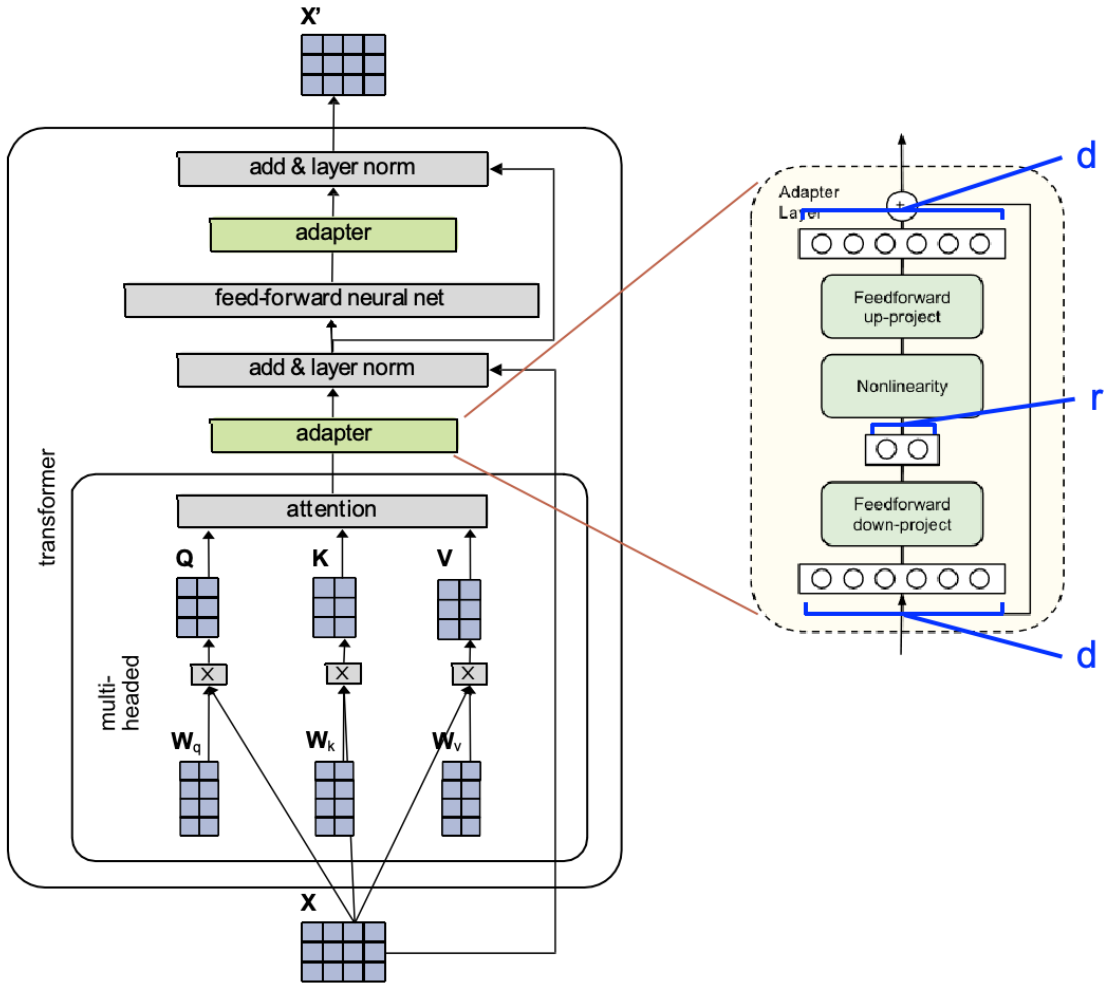
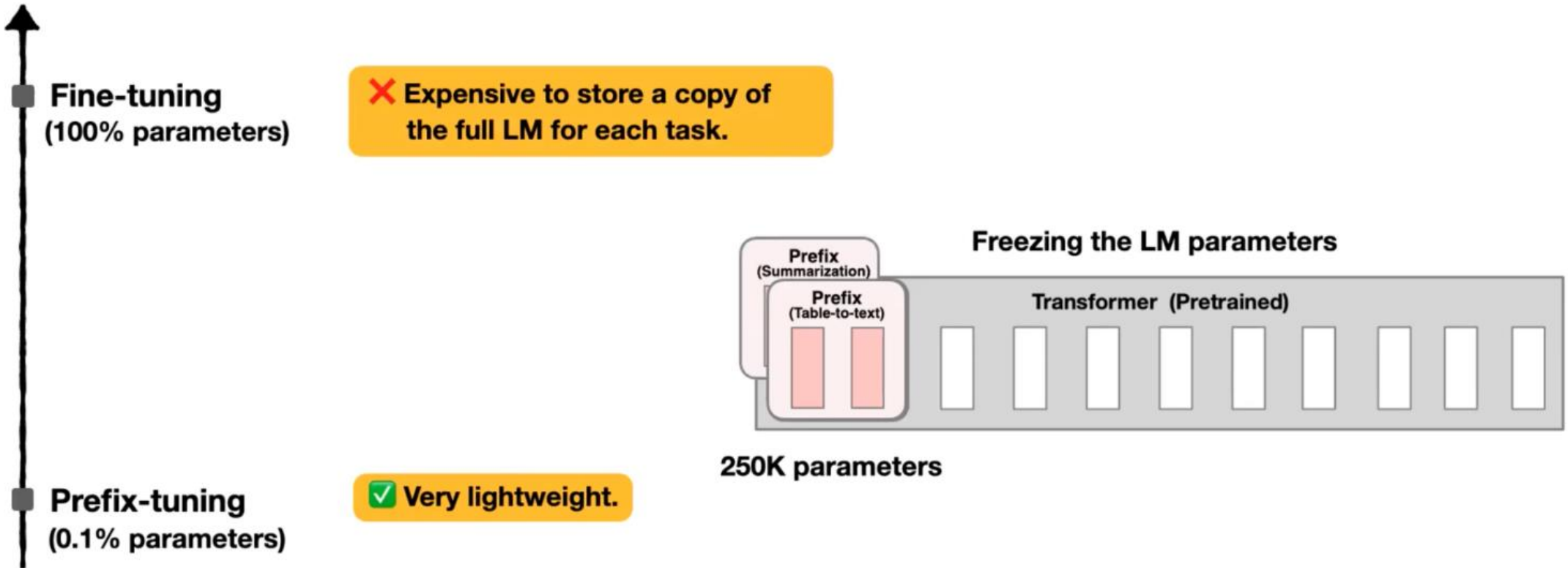


Figure inspired by He et al. (2022) https://arxiv.org/pdf/2110.04366

- In practice, r is chosen s.t. r << d and the adapter layers contain **only 0.5% – 8% of the total parameters**.
- When added to a deep neural network (e.g. Transformer) all the other parameters of the pretrained model are kept fixed, and only the adapter layer parameters are fine-tuned.

Other Parameter-efficient Tunings: Prefix-Tuning

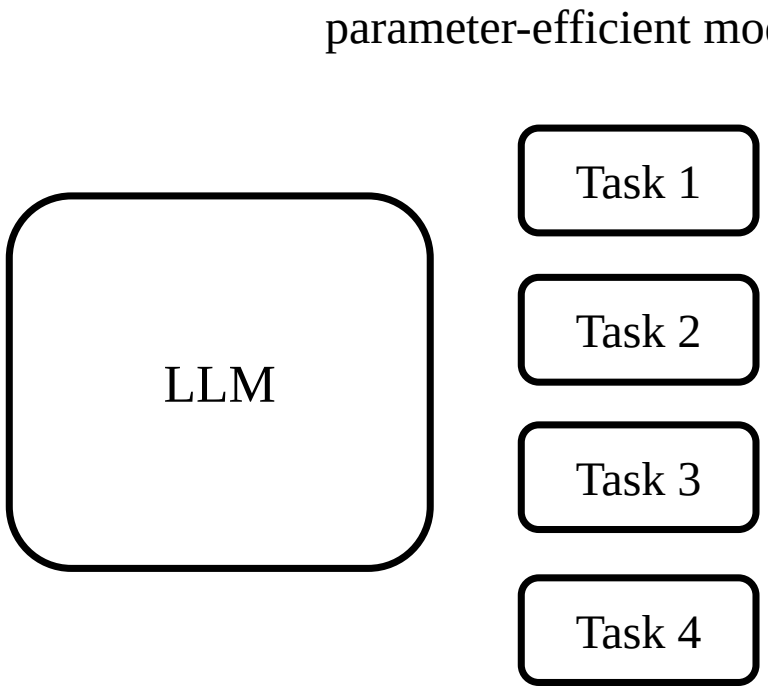


We prepend a prefix matrix in each transformer layer. And during Training, we only compute gradient w.r.t. Prefix parameters

4.2 Modularity and Composition

If each task is a small module, can modules be combined like Lego bricks?

Parameter-efficient Fine-tuning

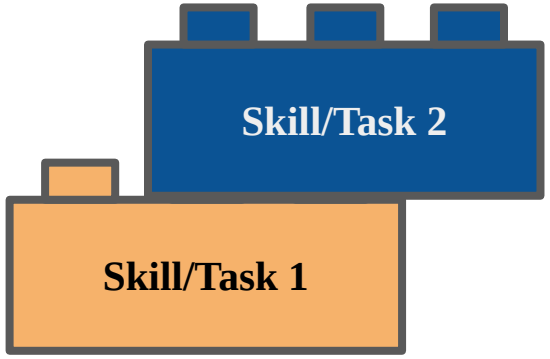


One task, one module

Modularity and Compositionality?



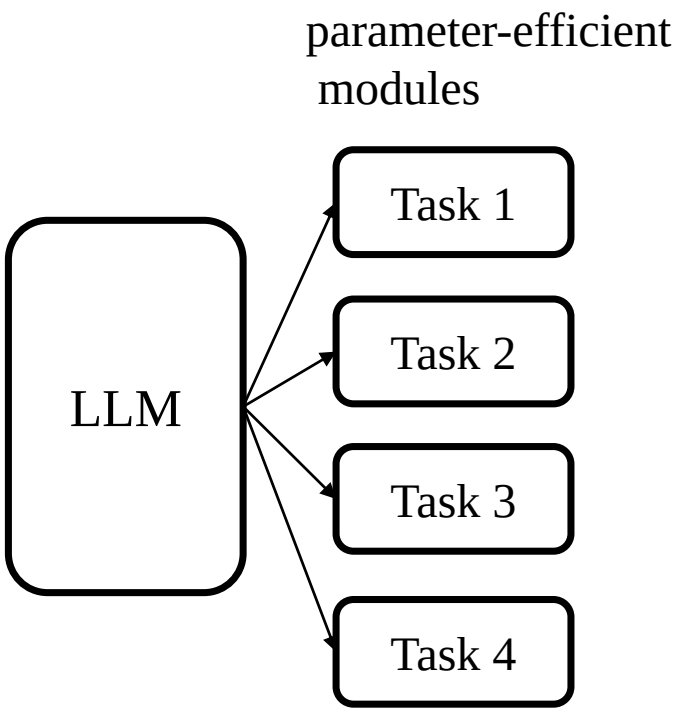
Modularity and Compositionality?



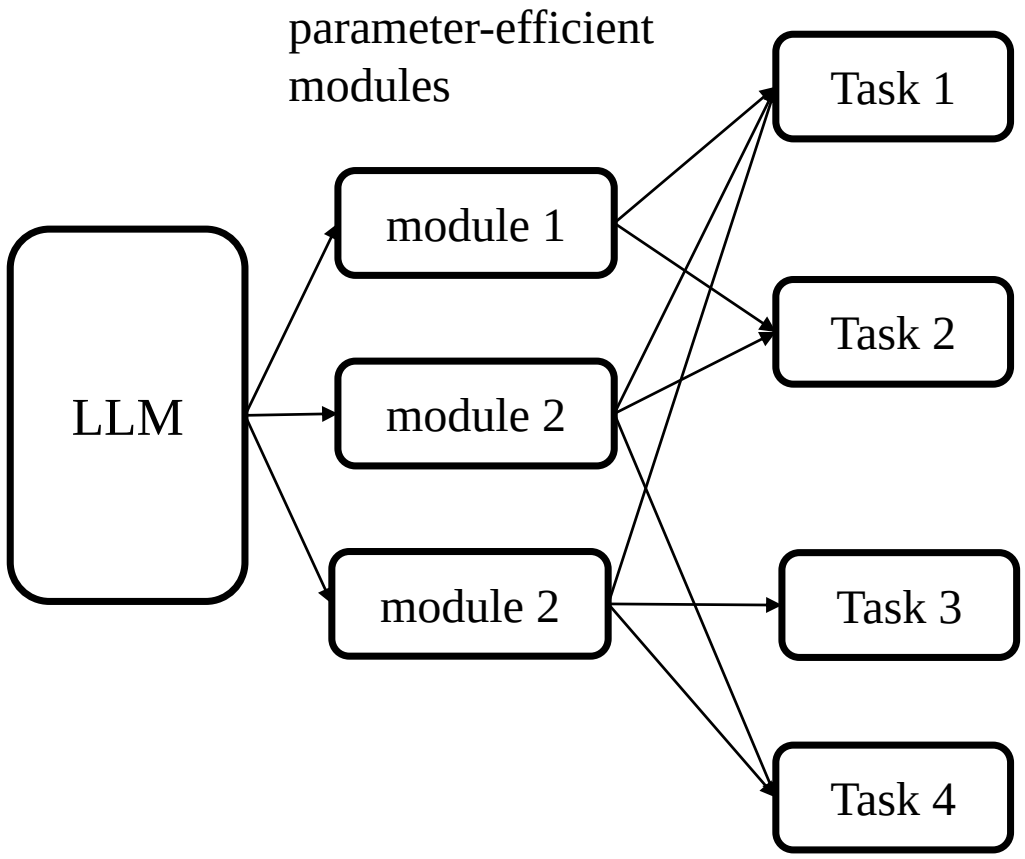
Modularity and Compositionality?



Parameter-efficient Fine-tuning

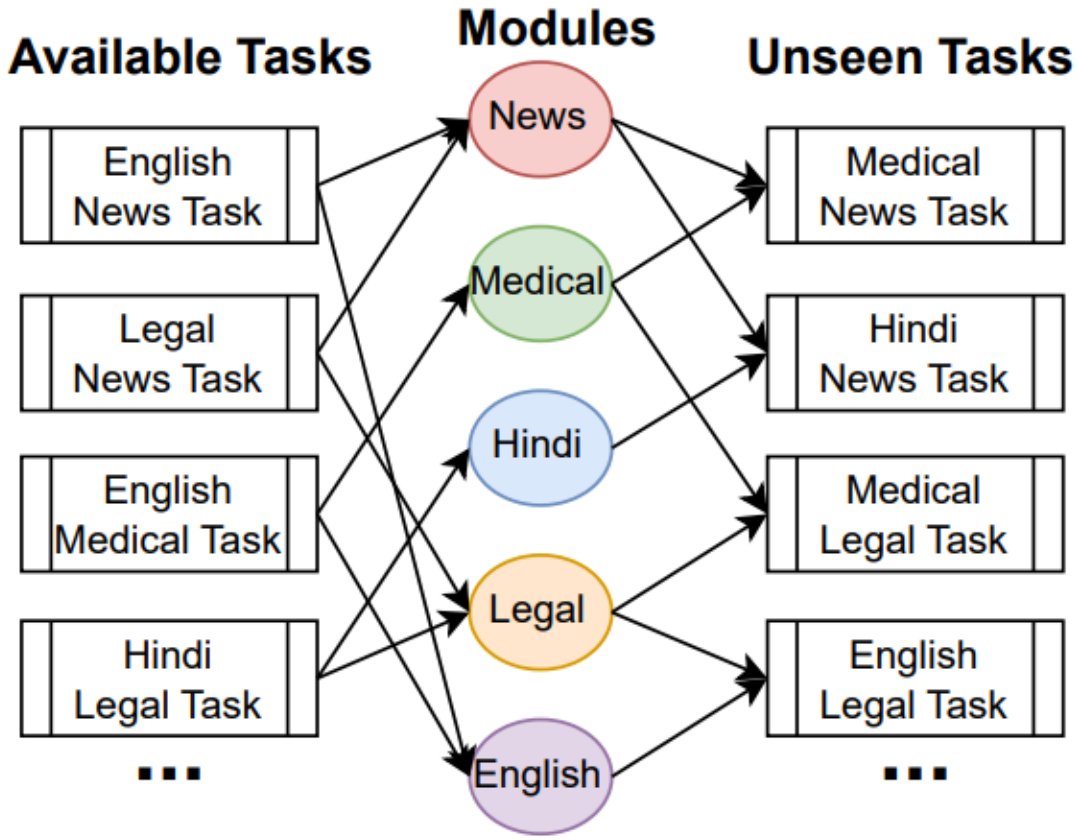


One task, one module



One task, composed modules

Modular Retrieval



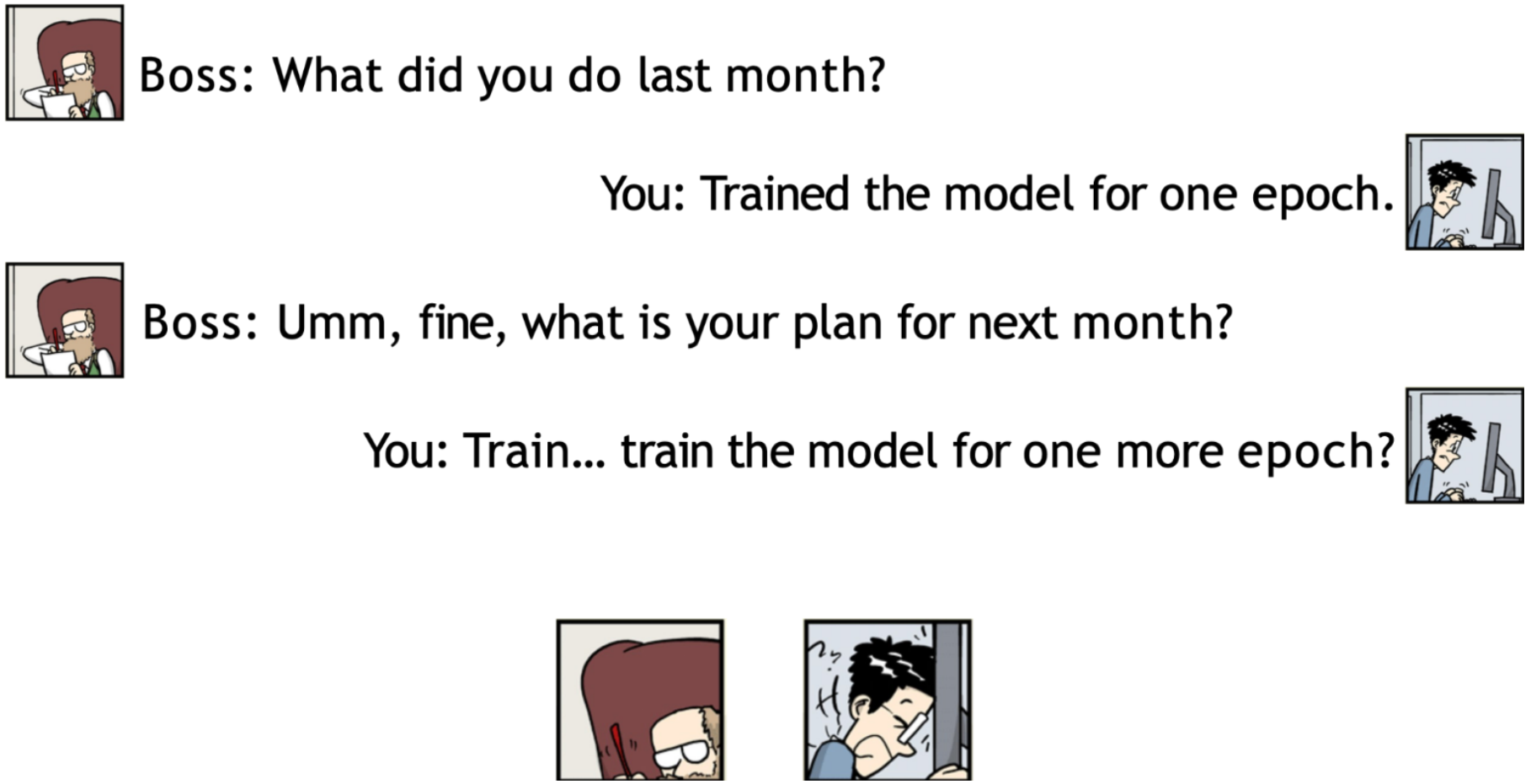
Juhao Liang , Chen Zhang , Zhengyang Tang , Jie Fu , Dawei Song , **Benyou Wang**. Modular Retrieval for Generalization and Interpretation. https://arxiv.org/pdf/2303.13419.pdf

4.3 Distributed Training

A 70B model does not fit on one GPU. Split data, layers, tensors — and optimizer states.

Models are getting larger and larger

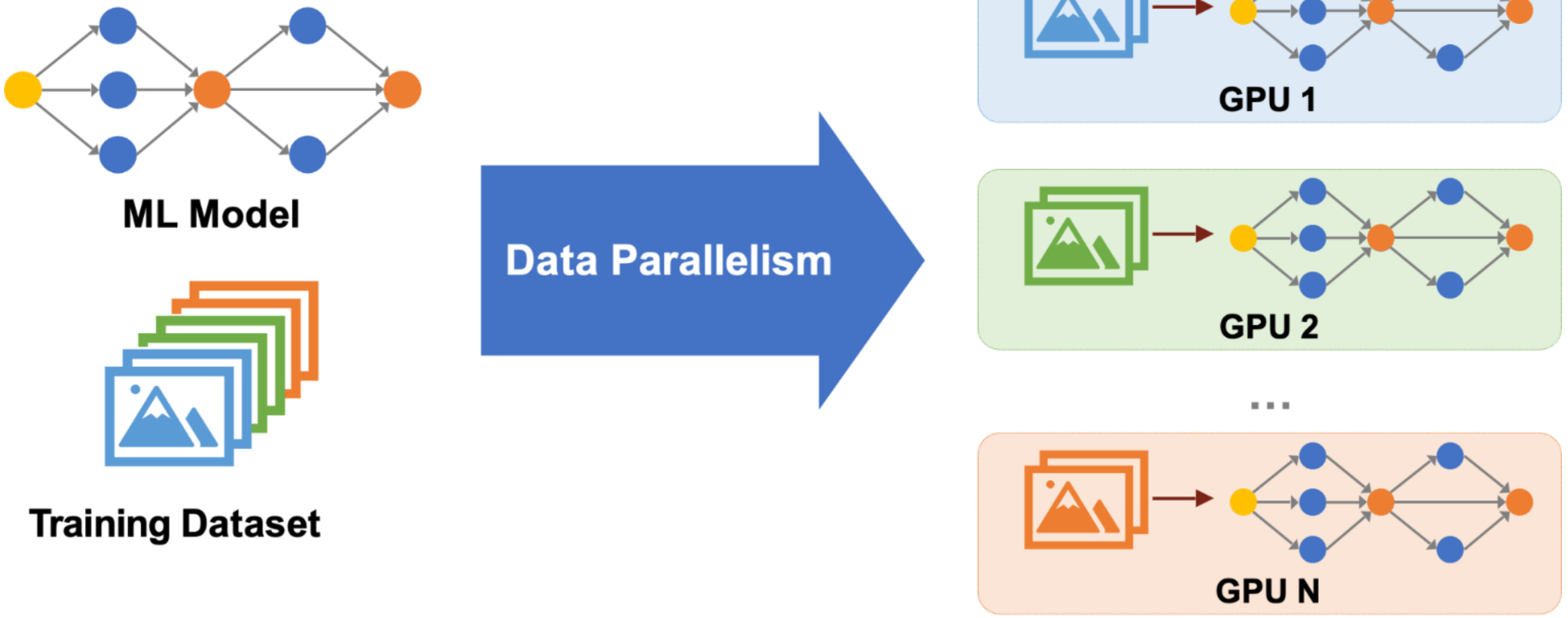
LLMs take much longer time to train!



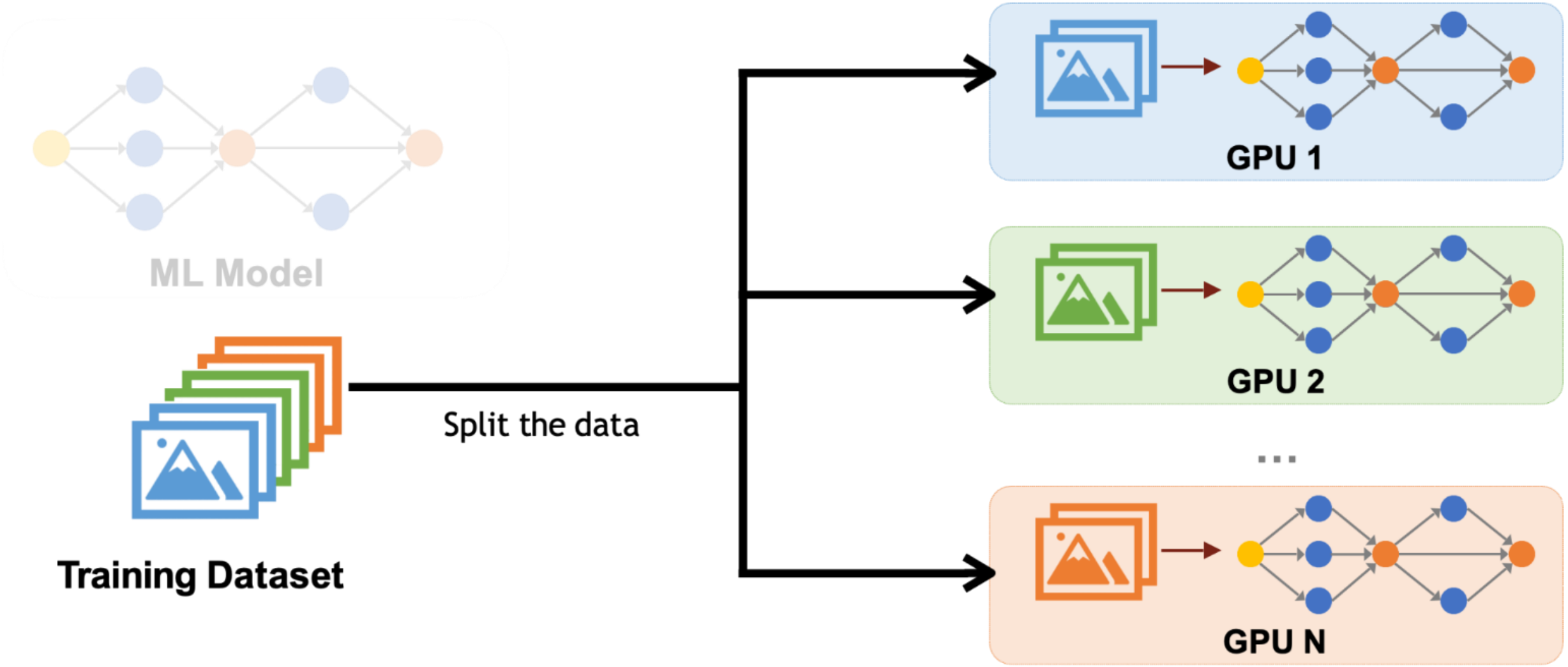
Distributed Training is almost Necessary for every LLMs!

- Developers / Researchers' time **are more valuable** than hardware .
- If a training takes **10 GPU days**
  - Parallelize with distributed training
  - 1024 GPUs can finish in 14 minutes (ideally)!
- The develop and research cycle will be greatly boosted

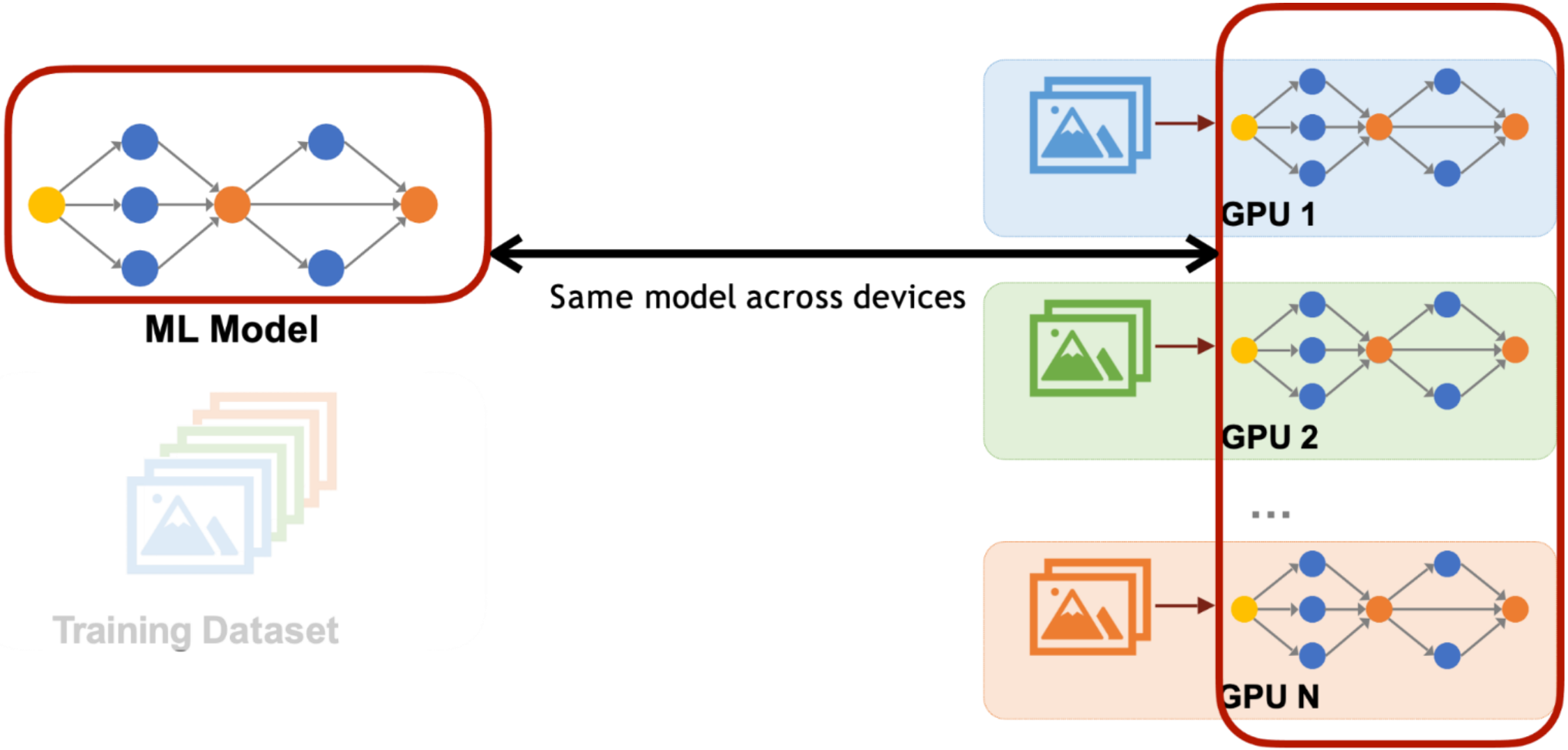
Parallelism in Distributed Training - Data Parallelism



Parallelism in Distributed Training - Data Parallelism



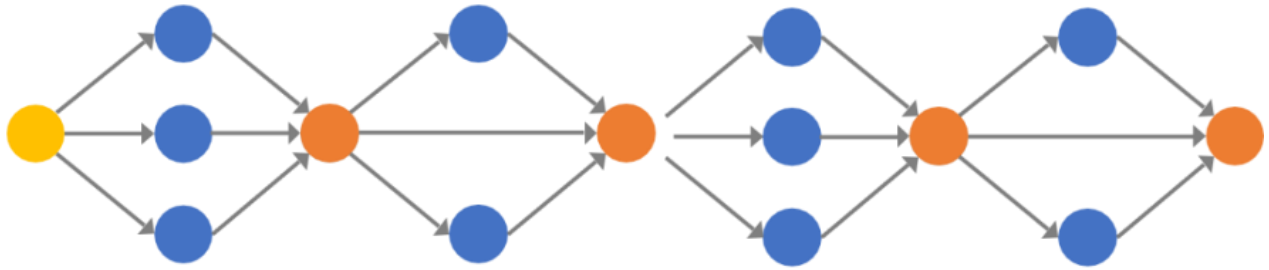
Parallelism in Distributed Training - Data Parallelism



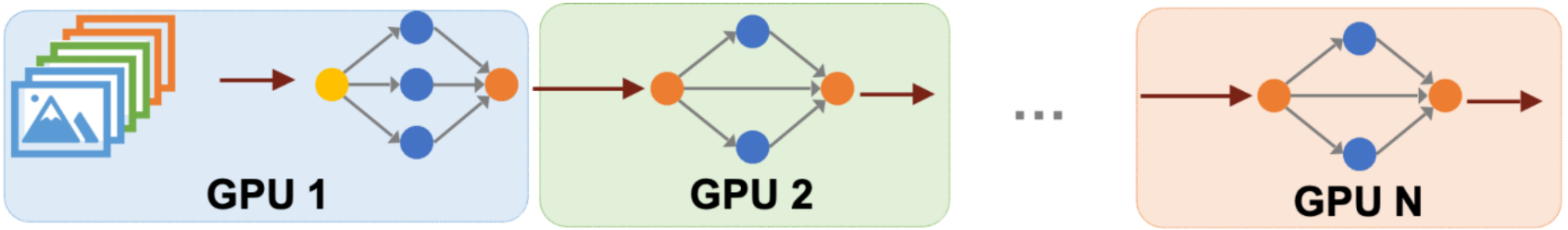
Parallelism in Distributed Training - Model Parallelism



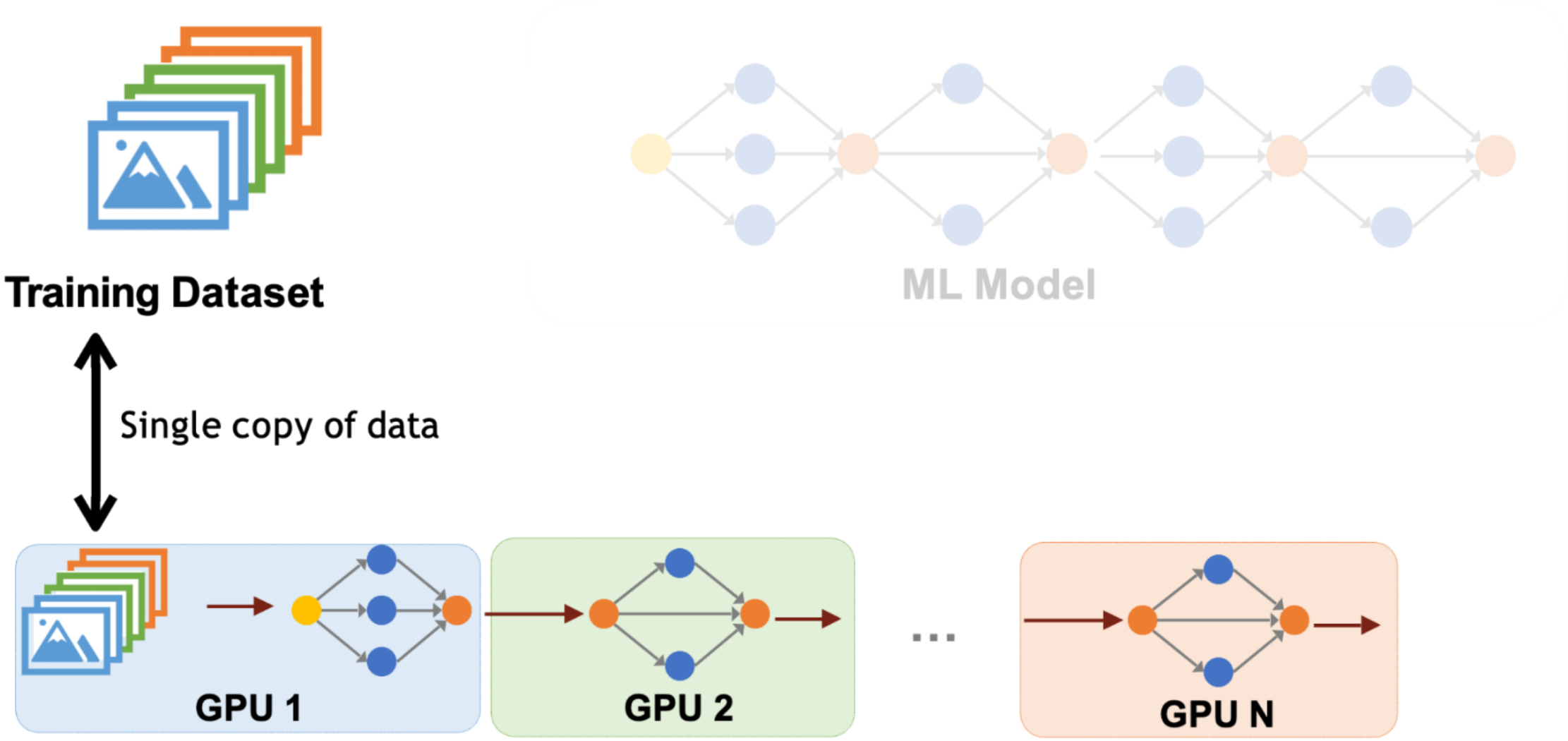
Training Dataset



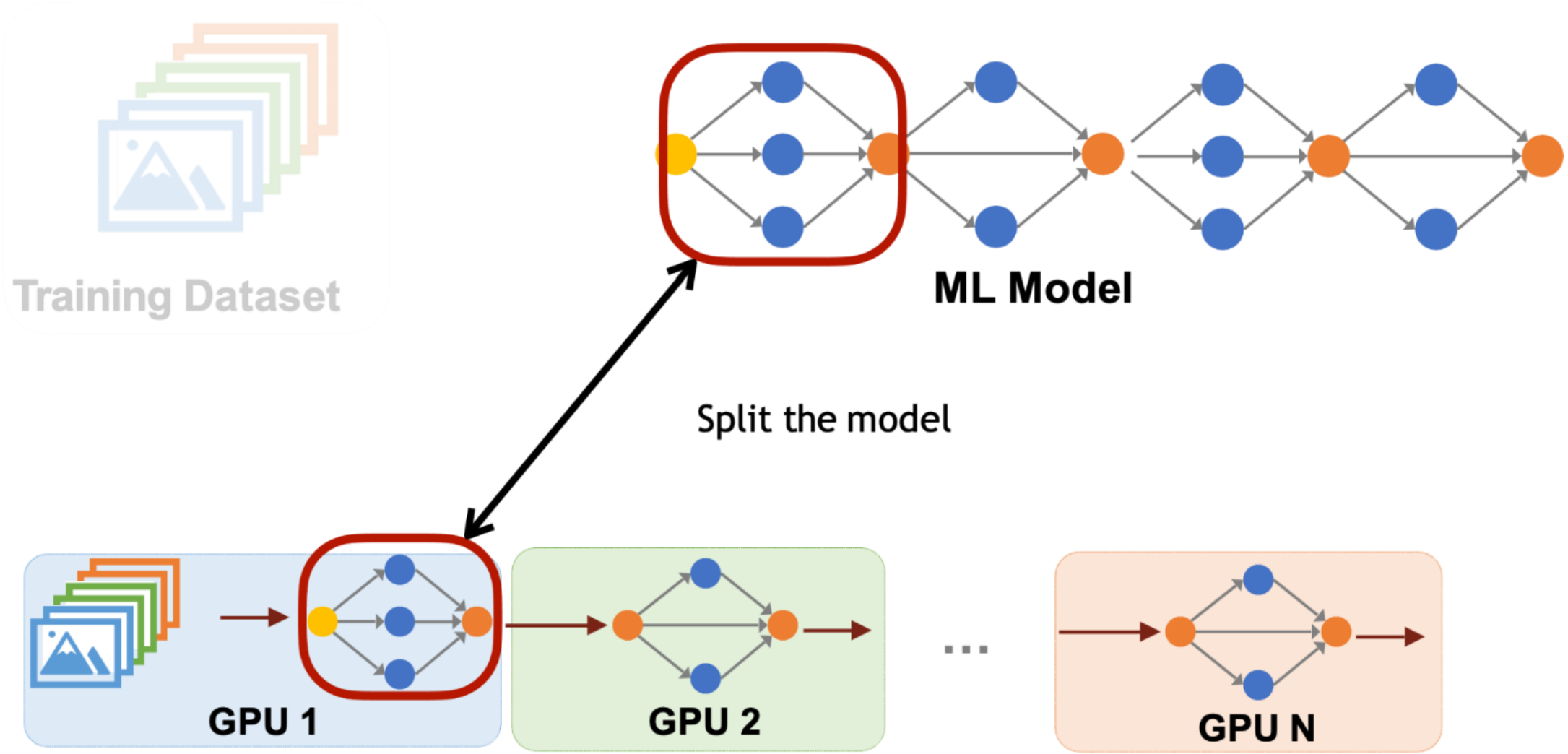
ML Model



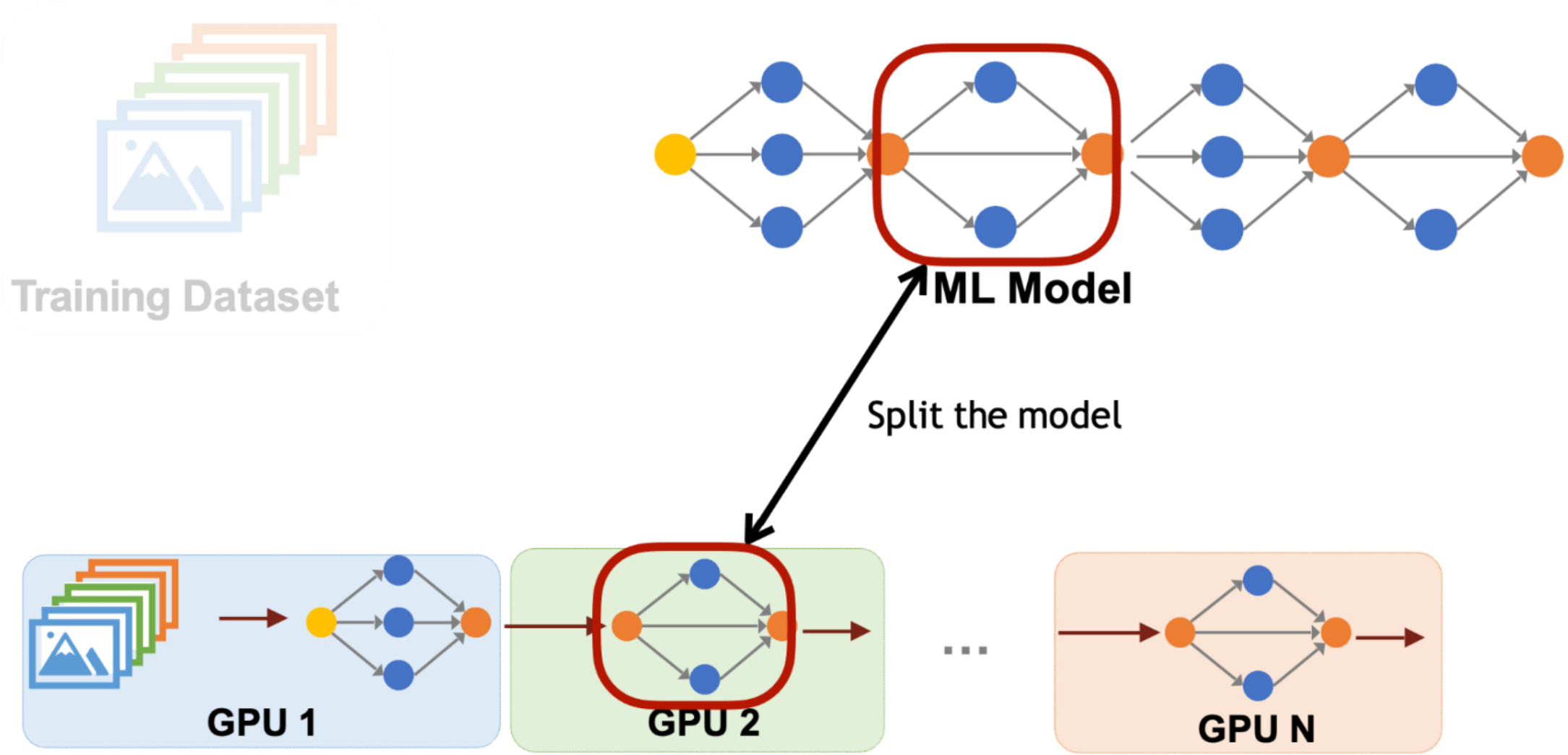
Parallelism in Distributed Training - Model Parallelism



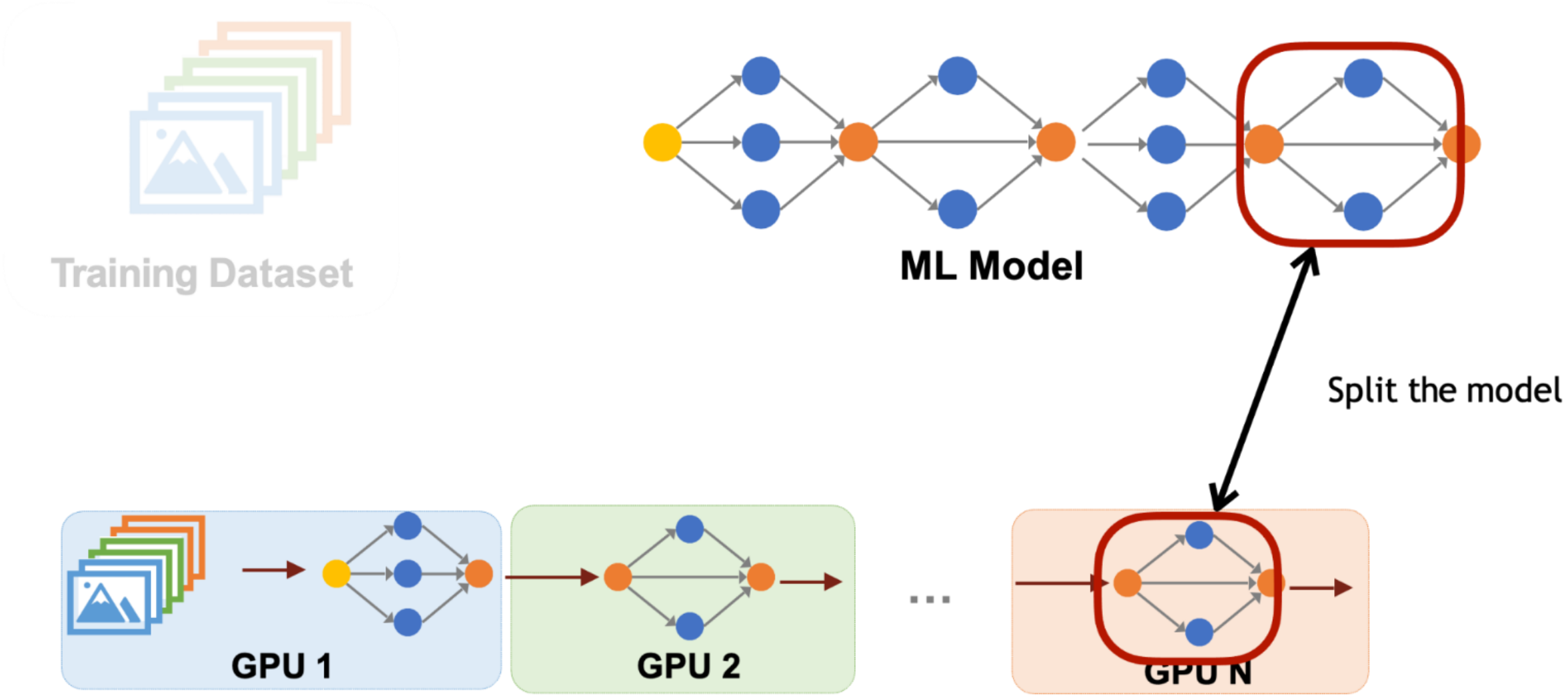
Parallelism in Distributed Training - Model Parallelism



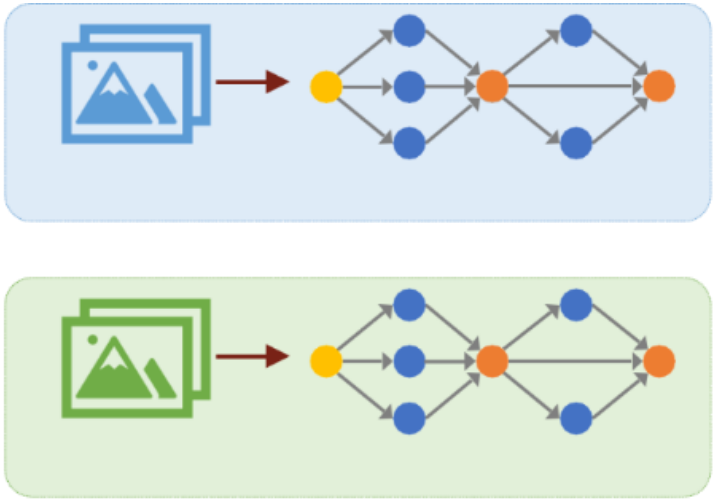
Parallelism in Distributed Training - Model Parallelism



Parallelism in Distributed Training - Model Parallelism

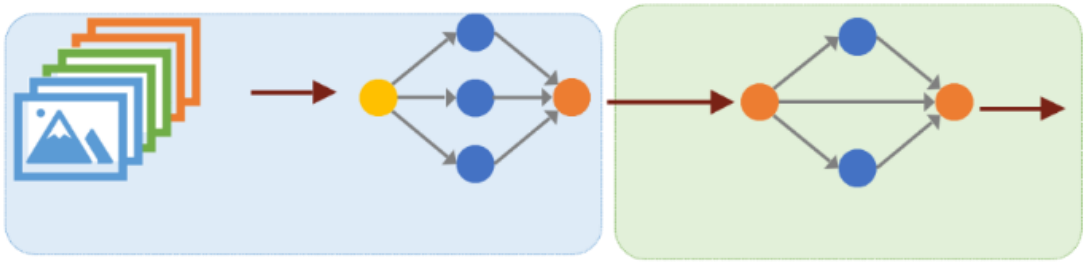


Parallelism in Distributed Training - DP vs MP



Data Parallelism:

- Split the data
- Same model across devices
- Easy to parallelize, high utilization
- N copies of model



Model Parallelism:

- Split the model
- Move activations through devices
- Hard to parallelize, load balancing issue
- Single copy of model

Distributed Training - ZeRO: Train Trillion-scale models

A Step Back: Training a Single Layer in Practice

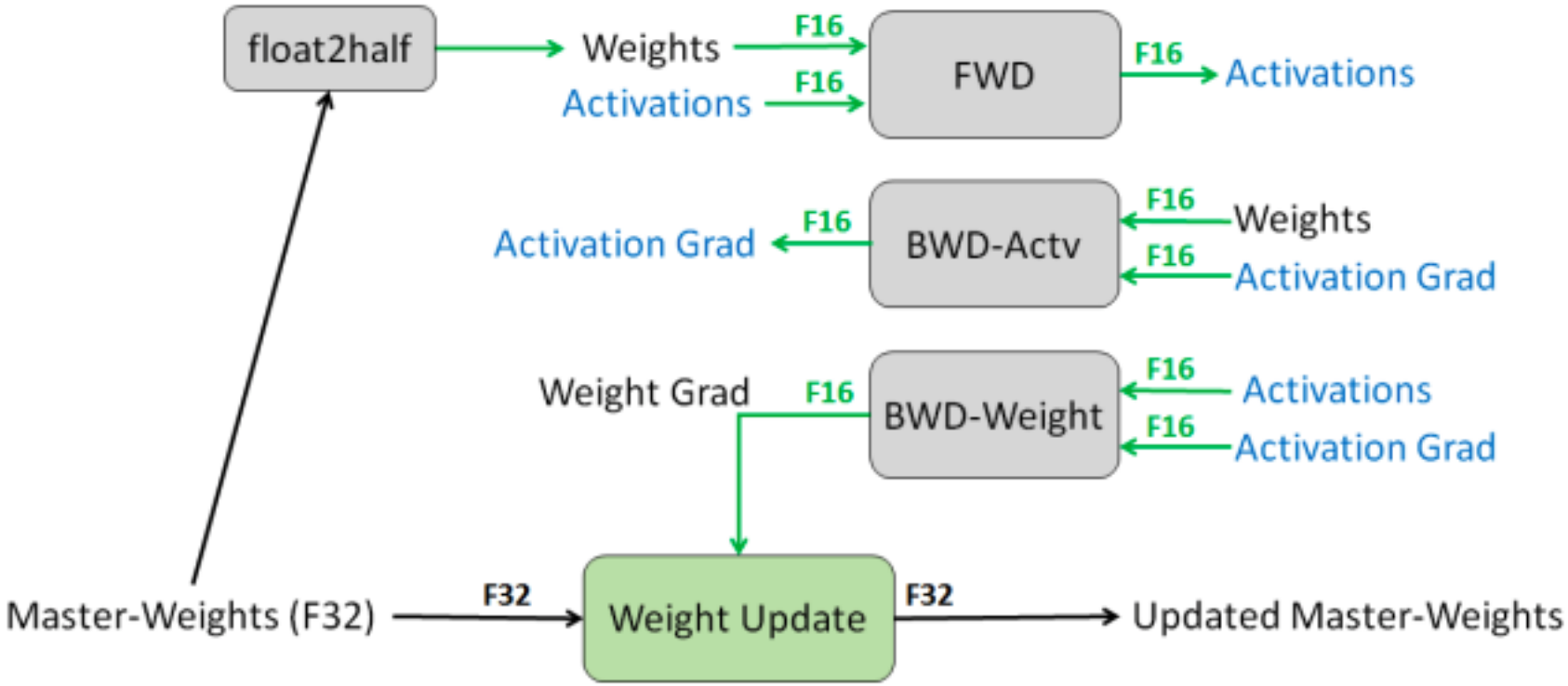


Figure 1: Mixed precision training iteration for a layer.

Memory Consumptions for this example:

Suppose the layer (or model) is trained using Adam Optimizer.
The number of parameters are **Φ**.

Then in a single training iteration, we have to save (corresponding memory consumption):

- Model parameters (fp16): 2**Φ**
- Model gradients (fp16): 2**Φ**
- Adam Optimizer states - copy of Parameters, Momentum and Variance (fp32): 4**Φ** + 4**Φ** + 4**Φ** = 12**Φ**
- Residual states, including activations, buffer, fragmentations

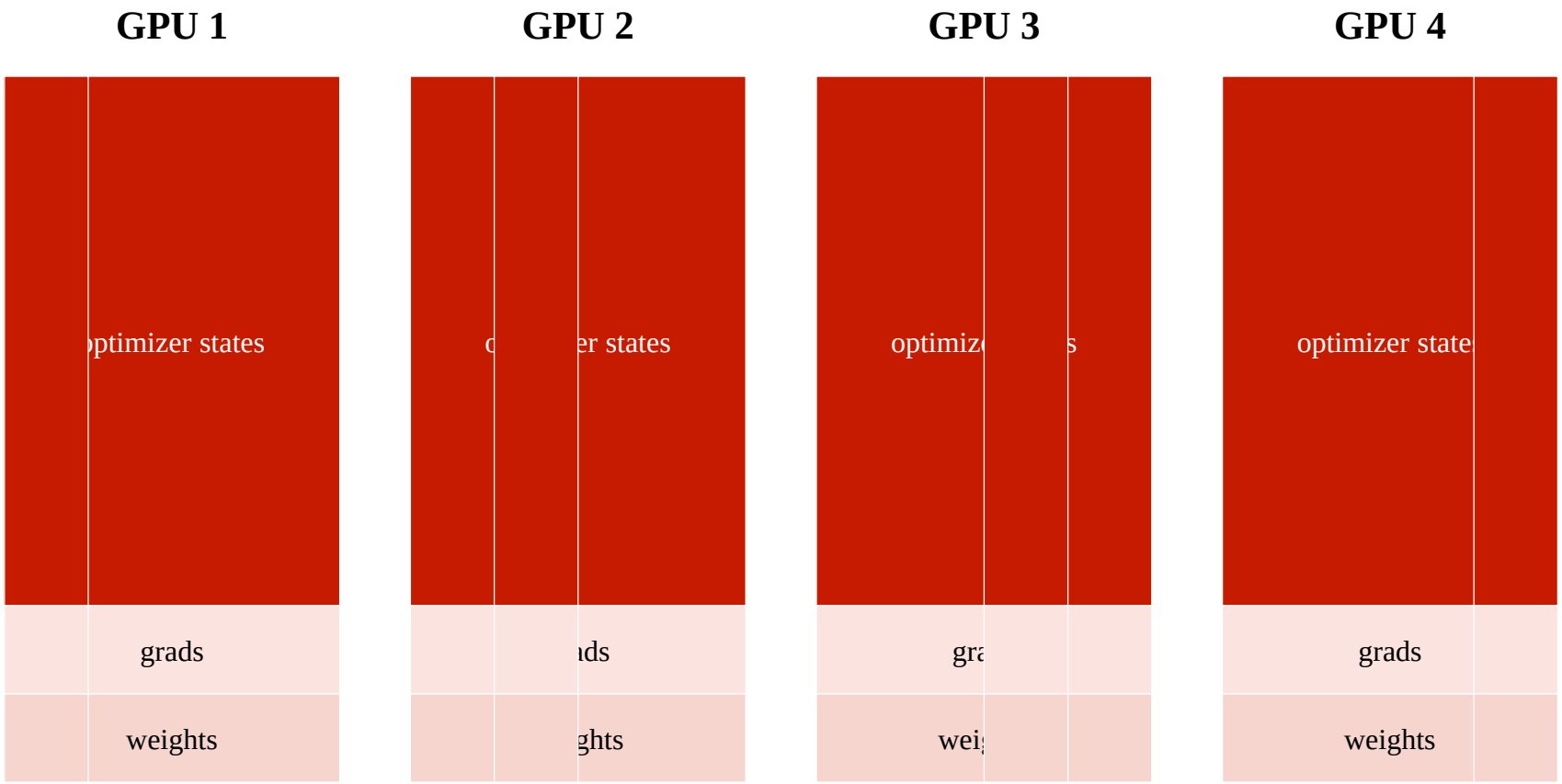
For a GPT-2 model, even it has only 1.5B model parameters (3GB memory is enough to hold it), training it would cost at least 24GB memory!

VRAM Estimation

Model: HuatuoGPT-7B Model Param(fp16): 7B*2=14GB Grad(fp16):7B*2=14GB Optimizer(AdamW) Master Weights(fp32): 7B*4=28GB Adam m(fp32): 7B*4=28GB Adam v(fp32): 7B*

ZeRO: Stop Storing the Same Thing on Every GPU

Mixed-precision Adam needs 16 bytes per parameter: weights 2 + gradients 2 + optimizer states 12



ZeRO-1: shard optimizer states → 4× less for the largest part

ZeRO-2: also shard gradients

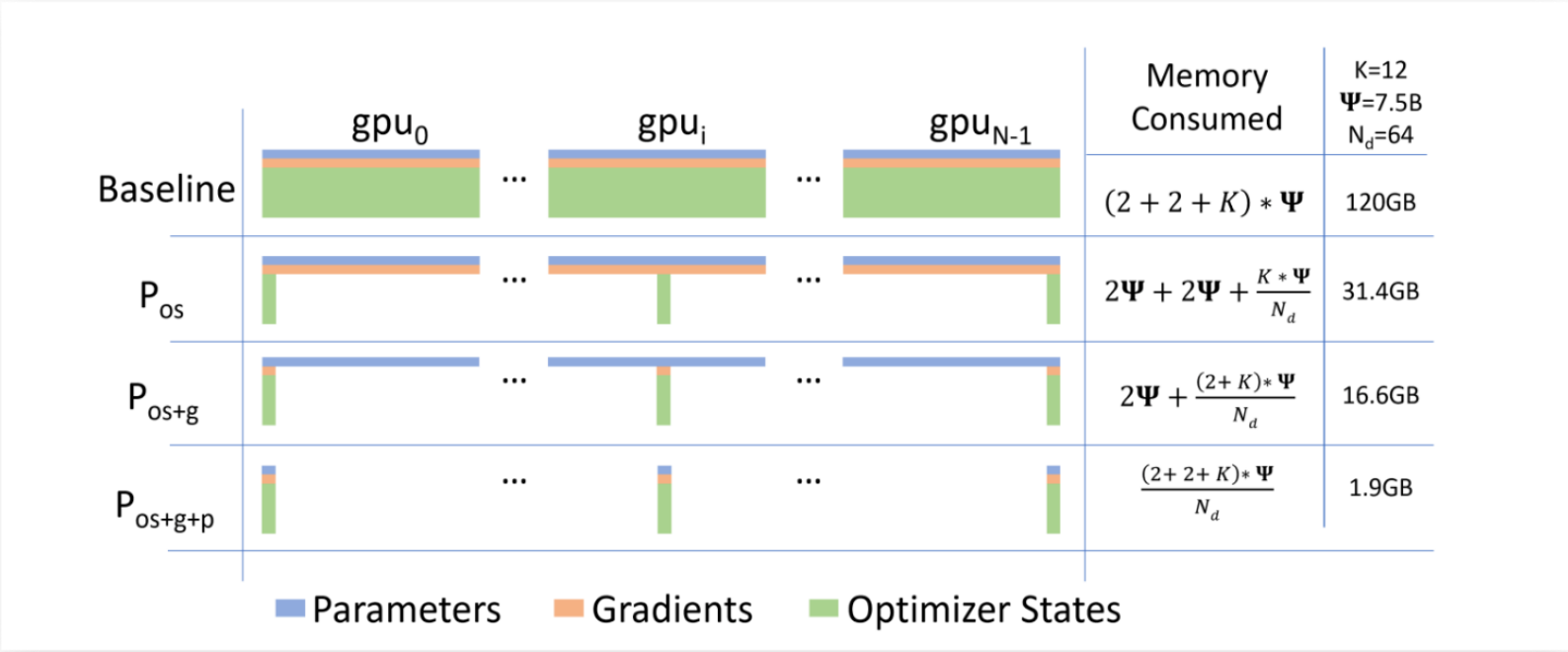
ZeRO-3: also shard weights → memory ∝ 1 / #GPUs

7.5B model on 64 GPUs: 120 GB → 31.4 GB (ZeRO-1) → 16.6 GB (ZeRO-2) → 1.9 GB per GPU (ZeRO-3)

Rajbhandari et al. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (SC 2020), Figure 1.

Parallel Strategy: ZeRO

1. ZeRO-DP: Shard the optimizer state
2. ZeRO-1&2: Same communication volume as DP
3. ZeRO-3: 1.5 communication volume as DP



K denotes the memory multiplier of optimizer states, and N denotes DP degree

[1] [1910.02054] ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (arxiv.org)

ZeRO: More GPUs for Larger Models!

DP	**7.5B Model** (GB)			**128B Model** (GB)			**1T Model** (GB)		
	P_{os}	P_{os+g}	P_{os+g+p}	P_{os}	P_{os+g}	P_{os+g+p}	P_{os}	P_{os+g}	P_{os+g+p}
1	120	120	120	2048	2048	2048	16000	16000	16000
4	52.5	41.3	**30**	896	704	512	7000	5500	4000
16	35.6	**21.6**	7.5	608	368	128	4750	2875	1000
64	**31.4**	16.6	1.88	536	284	**32**	4187	2218	250
256	30.4	15.4	0.47	518	263	8	4046	2054	62.5
1024	30.1	15.1	0.12	513	257	2	4011	2013	**15.6**

- We can train a 7.5B model (like Llama2) using only 4 V100-32GB GPUs
- We can even train a 128B model using 64 V100-32GB GPUs

5 Efficient Inference

Training happens once; inference runs billions of times. Decoding is memory-bound, so we fight for memory and bandwidth.

In this part

5.1 Why Inference Is Hard

5.2 The KV Cache

5.3 Speculative Decoding

5.4 FlashAttention

5.5 Serving Systems: vLLM

Why Inference Efficiency Matters

Training is a one-time cost; inference runs millions of times

Key bottlenecks in LLM inference:

- Memory bandwidth: loading model weights from GPU memory
- KV Cache: grows linearly with sequence length and batch size
- Autoregressive decoding: generates one token at a time (sequential)

Goal: serve more users, with lower latency, at lower cost

Modern LLM serving systems: vLLM, TensorRT-LLM, SGLang

Two Phases of LLM Inference

Prefill Phase (prompt processing):

- - Process all input tokens in parallel
- - Compute KV cache for each token
- - Compute-bound (GPU cores are the bottleneck)

Decode Phase (token generation):

- - Generate one token at a time, autoregressively
- - Each step reads the entire KV cache but produces only 1 token
- - Memory-bandwidth-bound (reading KV cache is the bottleneck)

Key insight: decode phase is extremely memory-inefficient

- - GPU utilization can be as low as 1-5% during decoding!

Why is LLM Serving Hard?

Huge memory footprint

- - Model weights: LLaMA-70B needs ~140GB in FP16
- - KV Cache: grows with sequence length x batch size x layers x heads

Variable-length requests

- - Input lengths vary widely (10 tokens to 100K tokens)
- - Output lengths are unknown in advance
- - Hard to pre-allocate memory efficiently

Latency vs. Throughput tradeoff

- - Users want low latency (fast first token)
- - Providers want high throughput (serve more users)

Memory fragmentation

- - Naive allocation wastes 60-80% of KV cache memory!

5.2 The KV Cache

Caching keys and values avoids recomputation — but the cache itself becomes the largest memory consumer.

What is the KV Cache?

In Transformer attention: Attention(Q, K, V) = softmax(QK^T / sqrt(d)) V

During autoregressive generation:

- At each step, only the new token produces a new Q
- But it needs K and V from ALL previous tokens
- Recomputing K, V every step is wasteful

Solution: cache K and V tensors from previous steps

KV Cache size per token per layer: = 2 (K and V) x num_heads x head_dim x dtype_size

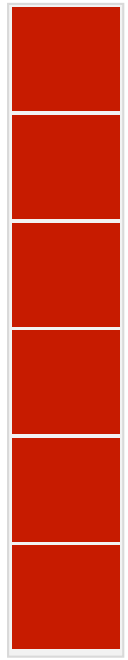
Example: LLaMA-2-70B, one sequence of 4096 tokens:

- = 2 x 80 layers x 64 heads x 128 dim x 2 bytes x 4096 tokens
- = ~5.2 GB for a SINGLE sequence!

The KV Cache Grows with Every Token

Each new token attends to all previous tokens — so we keep their keys and values in memory.

	The	cat	sat	on	the	mat
Keys	k1	k2	k3	k4	k5	k6
Values	v1	v2	v3	v4	v5	v6



GPU memory used by the KV cache

```
KV bytes per token = 2 × layers × KV heads × head dim × bytes
```

OPT-13B: ≈ 800 KB per token → a 2,048-token request needs ≈ 1.6 GB.

With dozens of requests, the cache — not the weights — limits the batch size.

Kwon et al. Efficient Memory Management for LLM Serving with PagedAttention (SOSP 2023).

KV Cache Optimization

In autoregressive generation, we cache Key and Value vectors to avoid recomputation

Problem: KV cache can consume massive GPU memory

- LLaMA-70B, 4K context, batch=32: ~40GB just for KV cache!

Solutions:

- Grouped Query Attention (GQA): share KV heads across query heads (in LLaMA-2/3, Mistral, Qwen2)
- Multi-Query Attention (MQA): extreme sharing (1 KV head)
- PagedAttention (vLLM): manage KV cache like virtual memory pages (Eliminates memory fragmentation, enables flexible batching)
- Multi-Head Latent Attention (MLA): DeepSeek-V2/V3 low-rank KV compression

PagedAttention: Virtual Memory for the KV Cache

Before: one contiguous chunk per request



reserved for the maximum length · mostly empty

60–80% of KV memory wasted by fragmentation

Logical blocks (request A)

block 0	Four	score	and	seven
block 1	years	ago	our	fathers
block 2	brought	forth		

Block table

0 → 7
1 → 1
2 → 3

Waste < 4% (only the last block) → 2–4× higher throughput

Physical KV blocks on the GPU

physical block 0	**A · block 1**
physical block 2	**A · block 2**
physical block 4	physical block 5
physical block 6	**A · block 0 + B (shared, ref count 2)**

Parallel samples / beams share the prompt's blocks; copy only on write.

Kwon et al. Efficient Memory Management for LLM Serving with PagedAttention (SOSP 2023) — the core of vLLM.

5.3 Speculative Decoding

Guess several tokens cheaply, then let the big model check them all in one pass — with no loss in quality.

Speculative Decoding

Leviathan et al., 2023; Chen et al., 2023

Problem: autoregressive decoding is slow (1 token per forward pass)

Key idea: use a small "draft" model to propose multiple tokens, then verify them in parallel with the large "target" model

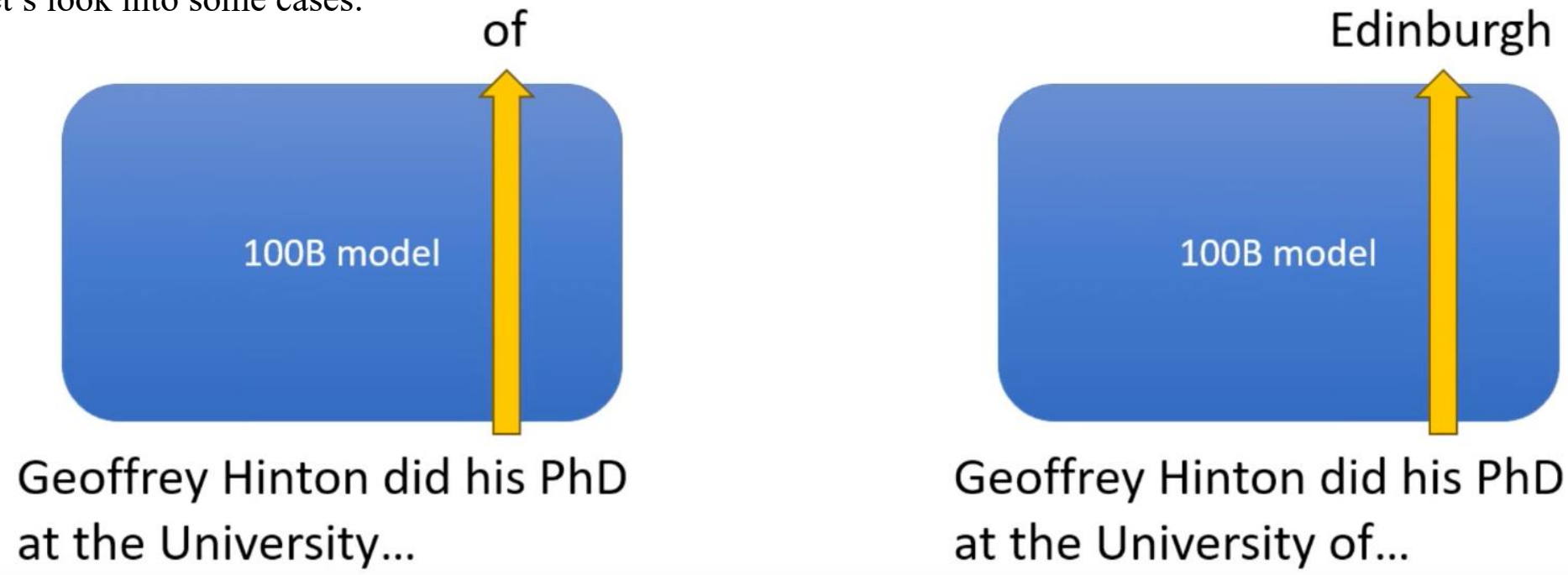
Algorithm:

- 1. Draft model generates K candidate tokens quickly
- 2. Target model verifies all K tokens in a single forward pass
- 3. Accept tokens that match; reject and resample otherwise

Result: 2-3x speedup with no quality loss (mathematically equivalent)
Variants: Medusa, EAGLE, Lookahead Decoding

Speculative Sampling: Easy vs. Hard Tokens

Let's look into some cases:



- Case 1: Predicting "of" is very easy, maybe we should use a 1B model which is enough
- Case 2: Predicting "Edinburgh" requires knowledge, which can be difficult, maybe we should use a 100B model
- This is key idea 1 behind: let small model deal with easy tokens, while large model deals with difficult tokens

Speculative Sampling: Use Parallel Verification



- We can give a transformer model multiple tokens at once, and let a large transformer model check them in parallel, while it does not increase compute time at well
- In this case, the probability for “Toronto” is low, cause the 100B model recognize it.
- This is key idea 2: let large transformer models check error tokens!

Speculative Decoding in Action

Prefix: “My”. p = small draft model, q = large target model.

Draft model p (small, fast)	dogs	love	chasing	**toys**
p(x)	p = 0.8	p = 0.7	p = 0.9	p = 0.8

Draft: 4 cheap forward passes, one token each

Target: ONE forward pass scores all 4 positions in parallel

Target model q (large, slow)	q = 0.9	q = 0.8	q = 0.8	q = 0.3
	✔ **q ≥ p → accept**	✔ **q ≥ p → accept**	✔ **accept w.p. q/p = 0.89**	✘ **accept w.p. q/p = 0.38**

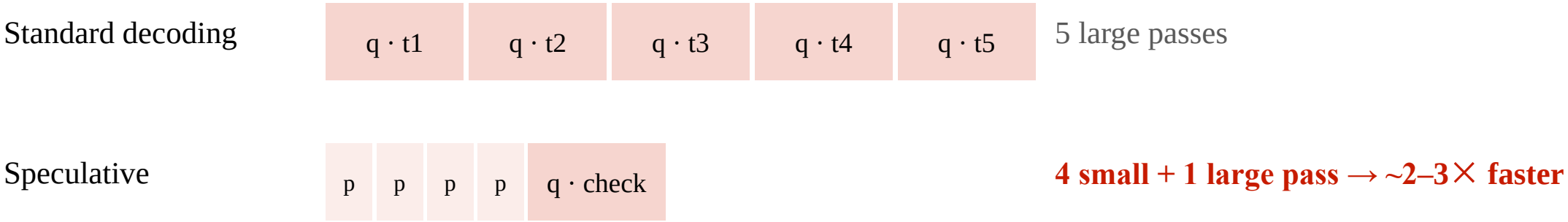
resample from norm(max(0, q − p))

Result: 4 new tokens (“dogs love chasing toys”) for 4 cheap passes + 1 big pass, instead of 4 big passes.

Leviathan et al. Fast Inference from Transformers via Speculative Decoding (ICML 2023) · Chen et al. Accelerating LLM Decoding with Speculative Sampling (2023).

Why It Is Faster — and Why It Is Exact

Wall-clock time for 5 tokens



Why faster?

- Decoding is memory-bound: one pass of the big model costs about the same for 1 or 5 tokens.
- Easy tokens (“of”, “the”) are guessed right by the small model.
- Speedup grows with the acceptance rate.

Why lossless?

- Accept token x with probability min(1, q(x)/p(x)).
- If rejected, sample from norm(max(0, q − p)).
- Together these give exactly q(x): the output distribution of the large model is unchanged.

Leviathan et al. (ICML 2023) · Chen et al. (2023). Variants: Medusa, EAGLE, lookahead decoding, multi-token prediction (DeepSeek-V3).

Speculative Sampling - Algorithm

M_p = draft model

M_q = target model

∞ meta-llama / **Llama-2-7b-chat-hf**

∞ meta-llama / **Llama-2-70b-chat-hf**

pf = prefix, $K = 5$ tokens

$p_1(x) = M_p(pf)$ → x_1

$p_2(x) = M_p(pf, x_1)$ → x_2

...

$p_5(x) = M_p(pf, x_1, x_2, x_3, x_4)$ → x_5

Speculative Sampling - Algorithm

$$p_1(x) = M_p(pf) \longrightarrow x_1$$

$$p_2(x) = M_p(pf, x_1) \longrightarrow x_2$$

$$\ldots$$

$$p_5(x) = M_p(pf, x_1, x_2, x_3, x_4) \longrightarrow x_5$$

Run draft model for K steps

$$q_1(x), q_2(x), q_3(x), q_4(x), q_5(x), q_6(x)$$

$$= M_q(pf, x_1, x_2, x_3, x_4, x_5)$$

Run target model once

Speculative Sampling - Algorithm

$$p_1(x) = M_p(pf) \longrightarrow x_1$$

$$p_2(x) = M_p(pf, x_1) \longrightarrow x_2$$

...

$$p_5(x) = M_p(pf, x_1, x_2, x_3, x_4) \longrightarrow x_5$$

Token	x1	x2	x3	x4	x5
	dogs	love	chasing	after	cars
p(x)	0.8	0.7	0.9	0.8	0.7
q(x)	0.9	0.8	0.8	0.3	0.8

$$q_1(x), q_2(x), q_3(x), q_4(x), q_5(x), q_6(x)$$

$$= M_q(pf, x_1, x_2, x_3, x_4, x_5)$$

Speculative Sampling - Rejection Sampling

Token	x1	x2	x3	x4	x5
	dogs	love	chasing	after	cars
p(x)	0.8	0.7	0.9	0.8	0.7
q(x)	0.9	0.8	0.8	0.3	0.8

Case 1: If $q(x) \geq p(x)$, then accept

Case 2: If $q(x) < p(x)$, then accept with probability $\frac{q(x)}{p(x)}$

In this case, we accept “dogs”, “love”, what about “chasing”? - we accept it with probability 0.8/0.9!

Speculative Sampling - Rejection Sampling

Token	x1	x2	x3	x4	x5
	dogs	love	chasing	after	cars
p(x)	0.8	0.7	0.9	0.8	0.7
q(x)	0.9	0.8	0.8	0.3	0.8

Case 1: If $q(x) \geq p(x)$, then accept

Case 2: If $q(x) < p(x)$, then accept with probability $\frac{q(x)}{p(x)}$

In this case, we accept “dogs”, “love”, what about “chasing”? - we accept it with probability 0.8/0.9, maybe we should accept it!

Speculative Sampling - Rejection Sampling

Token	x1	x2	x3	x4	x5
	dogs	love	chasing	after	cars
p(x)	0.8	0.7	0.9	0.8	0.7
q(x)	0.9	0.8	0.8	0.3	0.8

Case 1: If $q(x) \geq p(x)$, then accept

Case 2: If $q(x) < p(x)$, then accept with probability $\frac{q(x)}{p(x)}$

If we accept “chasing”, then what about “after”? The probability = 0.3/0.8, so maybe it should be rejected.

Speculative Sampling - Rejection Sampling

Token	x1	x2	x3	x4	x5
	dogs	love	chasing	after	cars
p(x)	0.8	0.7	0.9	0.8	0.7
q(x)	0.9	0.8	0.8	0.3	0.8

Case 1: If $q(x) \geq p(x)$, then accept

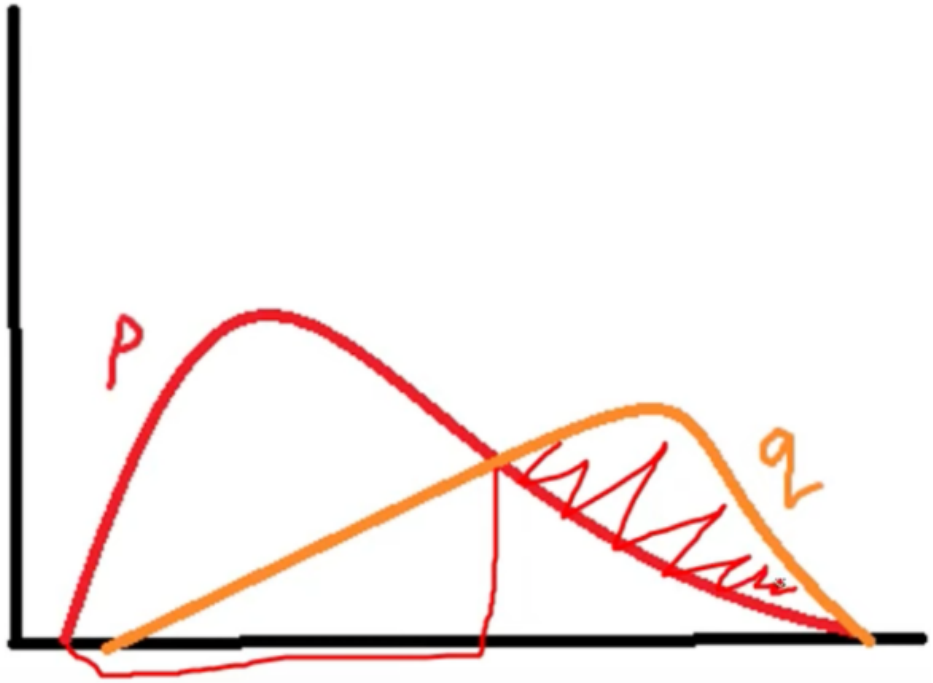
Case 2: If $q(x) < p(x)$, then accept with probability $\frac{q(x)}{p(x)}$

If we reject “after”, then we can sample a token from q(4) (based on the large model)!

Speculative Sampling - Rejection Sampling

Actually, don't sample $q(x)$

Adjusted distribution: $\left(q(x) - p(x)\right)_+$



We sample the 4th token by (q(4) - p(4))+!

Theoretically, we can ensure the token distribution is exactly q(x), so no loss in accuracy!

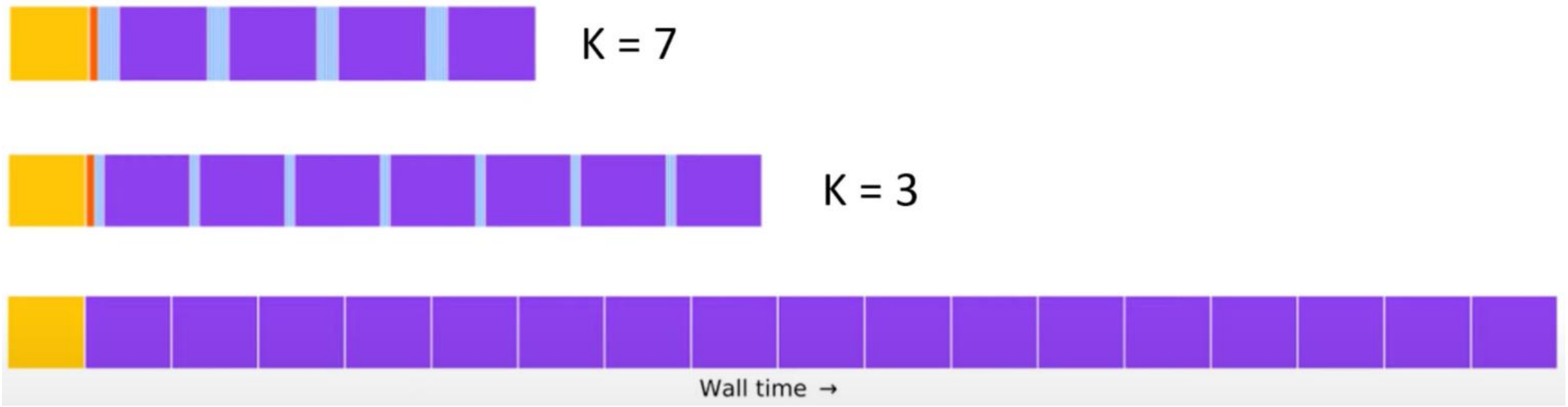
Speculative Sampling - #tokens generated in one pass

Token	x1	x2	x3	x4	x5
	dogs	love	chasing	after	cars
p(x)	0.8	0.7	0.9	0.8	0.7
q(x)	0.9	0.8	0.8	0.3	0.8

Worst case: first token is rejected -> 1 token

Best case: all tokens accepted -> K+1 tokens

Speculative Sampling - Wall Time



Speculative Sampling - Wall Time

Sampling Method	Benchmark	Result	Mean Token Time	Speed Up
ArS (Nucleus)	XSum (ROUGE-2)	0.112	14.1ms/Token	1×
SpS (Nucleus)		0.114	7.52ms/Token	1.92×
ArS (Greedy)	XSum (ROUGE-2)	0.157	14.1ms/Token	1×
SpS (Greedy)		0.156	7.00ms/Token	2.01×
ArS (Nucleus)	HumanEval (100 Shot)	45.1%	14.1ms/Token	1×
SpS (Nucleus)		47.0%	5.73ms/Token	2.46×

DeepMind

Recommends K = 3-4

Finds 2-2.5x speedup

Google Research

Recommends K = 3-7

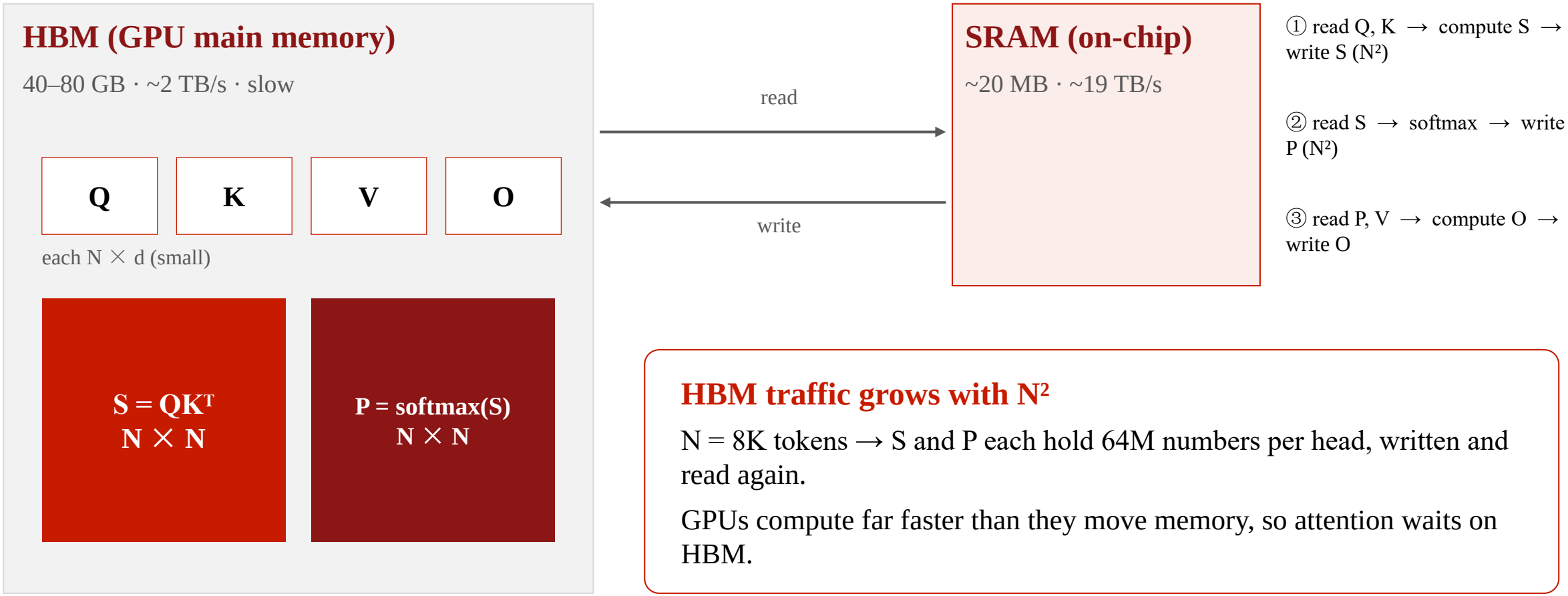
Finds 2-3.4x speedup

TASK	M_q	TEMP	γ	α	SPEED
ENDE	T5-SMALL ★	0	7	0.75	**3.4X**
ENDE	T5-BASE	0	7	0.8	2.8X
ENDE	T5-LARGE	0	7	0.82	1.7X
ENDE	T5-SMALL ★	1	7	0.62	**2.6X**
ENDE	T5-BASE	1	5	0.68	2.4X
ENDE	T5-LARGE	1	3	0.71	1.4X
CNNDM	T5-SMALL ★	0	5	0.65	**3.1X**
CNNDM	T5-BASE	0	5	0.73	3.0X
CNNDM	T5-LARGE	0	3	0.74	2.2X
CNNDM	T5-SMALL ★	1	5	0.53	**2.3X**
CNNDM	T5-BASE	1	3	0.55	2.2X
CNNDM	T5-LARGE	1	3	0.56	1.7X

5.4 FlashAttention

Attention is slow because of memory traffic, not arithmetic. Compute it tile by tile inside fast on-chip memory.

Standard Attention: The Problem Is Memory Traffic



HBM traffic grows with N^2

N = 8K tokens → S and P each hold 64M numbers per head, written and read again.

GPUs compute far faster than they move memory, so attention waits on HBM.

Click through: watch how often the huge N × N matrices travel between HBM and the chip.

Dao et al. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness (NeurIPS 2022). Bandwidth figures: NVIDIA A100.

FlashAttention: The Idea in One Slide

Dao et al., 2022: "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness"

Problem: standard attention is O(N^2) in both compute and memory
Key insight: the bottleneck is memory IO, not compute (GPU is fast, HBM is slow)

Solution: tiling + kernel fusion

- Compute attention in blocks (tiles) that fit in SRAM
- Never materialize the full N x N attention matrix in HBM
- Use online softmax trick to compute exact attention blockwise

Result: 2-4x speedup, memory from O(N^2) to O(N)
FlashAttention-2/3: further optimized for newer GPU architectures (H100)

FlashAttention: GPU Memory Hierarchy

Background Knowledge: GPU Structure
- SRAM: High-speed Cache Memory
  - High-speed, volatile, limited capacity
- HBM: High Bandwidth Memory
  - High speed, volatile, large capacity

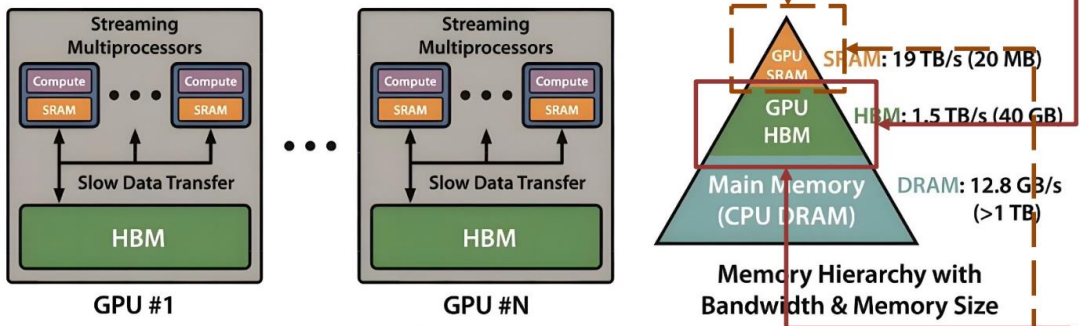
Key Idea: Utilize Characteristics of Attention
- Improve flops, optimize for SRAM storage
- Reduce IO, optimize the data bandwidth and efficiency

Implementation:
- Softmax: online softmax
  - Online softmax optimization, increasing computational efficiency
- Tiling: On-the-fly tiling (reducing computation)
  - Reduce recomputation (save time and resources)

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.



Attention	Standard	FlashAttention
Gflops	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtims (ms)	41.7	7.3

FlashAttention: Tile, Compute On-Chip, Never Store N × N



N × N scores: never stored

SRAM holds only one tile at a time

block 1: running max m, running sum ℓ, partial O

block 2: new max → rescale old ℓ and O, add new terms

block 3: rescale again → O1 is exact, written once

What changed?

- Memory: O(N) instead of O(N^2) — no S or P in HBM
- Traffic: each Q, K, V block read a few times; O written once
- Exact: same output as standard attention
- Backward: recompute S tiles instead of storing them

Dao et al. FlashAttention (NeurIPS 2022) · Dao. FlashAttention-2 (2023) · Shah et al. FlashAttention-3 (2024).

Online Softmax: Softmax Without Seeing the Whole Row

One row of scores, arriving in two blocks: block 1 = [2, 1], block 2 = [4, 3]

	running max	running sum	output
After block 1	m = 2	$\ell = e^{0} + e^{-1} = 1.368$	partial output uses weights e^(s − 2)
Block 2 arrives	m = max(2, 4) = 4	$\ell = 1.368 \cdot e^{(2-4)} + (e^{0} + e^{-1}) = 1.553$	rescale old output by e^(2 − 4), add new terms
Check	full row max = 4	$e^{-2} + e^{-3} + e^{0} + e^{-1} = 1.553$ ✔	identical to the one-shot softmax

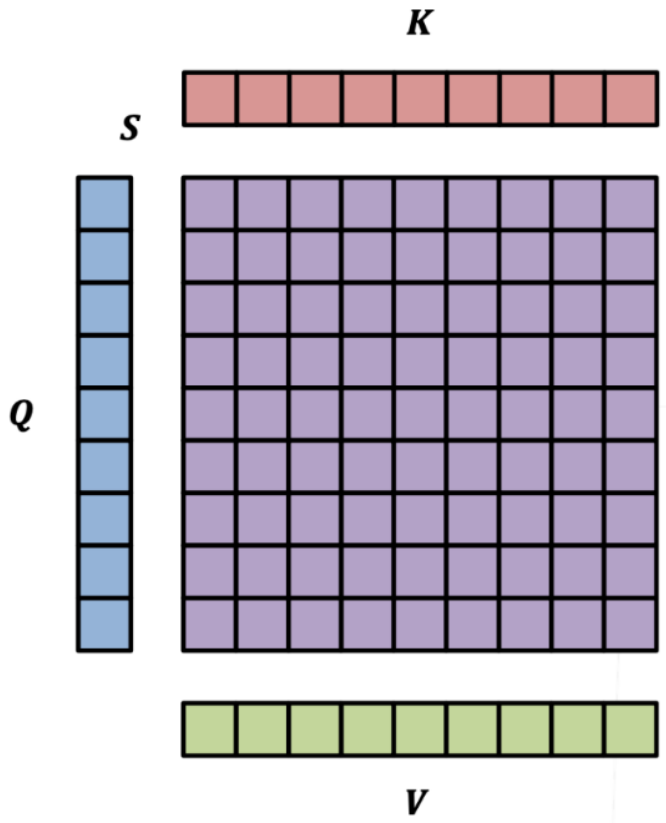
```
m_new = max(m_old, m_block)        ℓ_new = e^(m_old − m_new) · ℓ_old + Σ e^(s − m_new)
```

Two numbers per row are enough to merge blocks exactly — this is what lets FlashAttention work tile by tile.

Milakov & Gimelshein. Online Normalizer Calculation for Softmax (2018) · Dao et al. FlashAttention (2022).

FlashAttention Step by Step: Notation

Notations



Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.

2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.

5: **for** $1 \le j \le T_c$ **do**

6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

7: **for** $1 \le i \le T_r$ **do**

8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.

9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij})$ $_{r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}$ $_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} =$ $\text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij})$ $, \ell_i^{\text{new}} = e^{m_i - m_i^{\text{ne}}}$ $e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

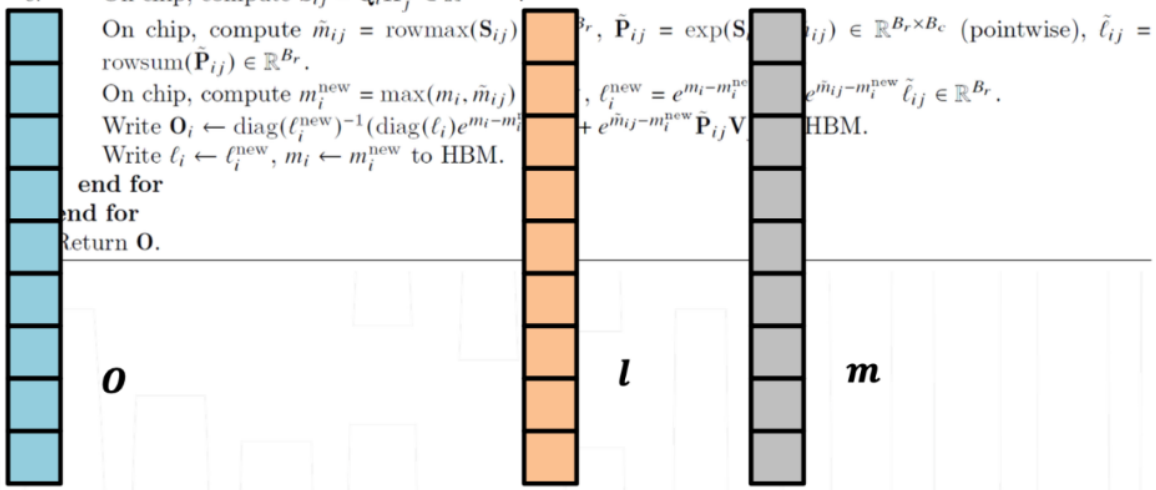
Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{}}}$ $+ e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}$ HBM.

Write $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$ to HBM.

end for

nd for

Return $\mathbf{O}$.



FlashAttention Step by Step: Split Blocks (1)

Split Blocks

Algorithm 1 FLASHATTENTION

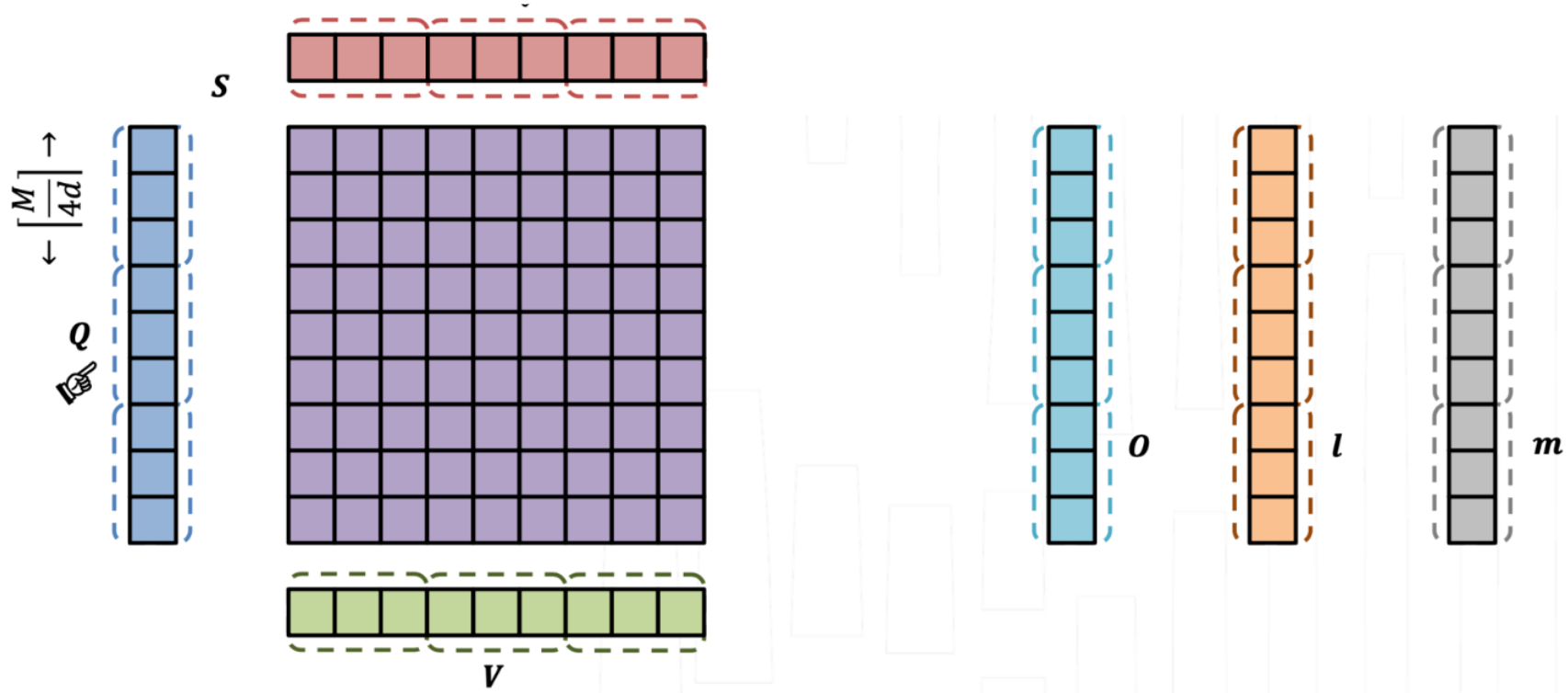
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.

2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.



FlashAttention Step by Step: Split Blocks (2)

Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

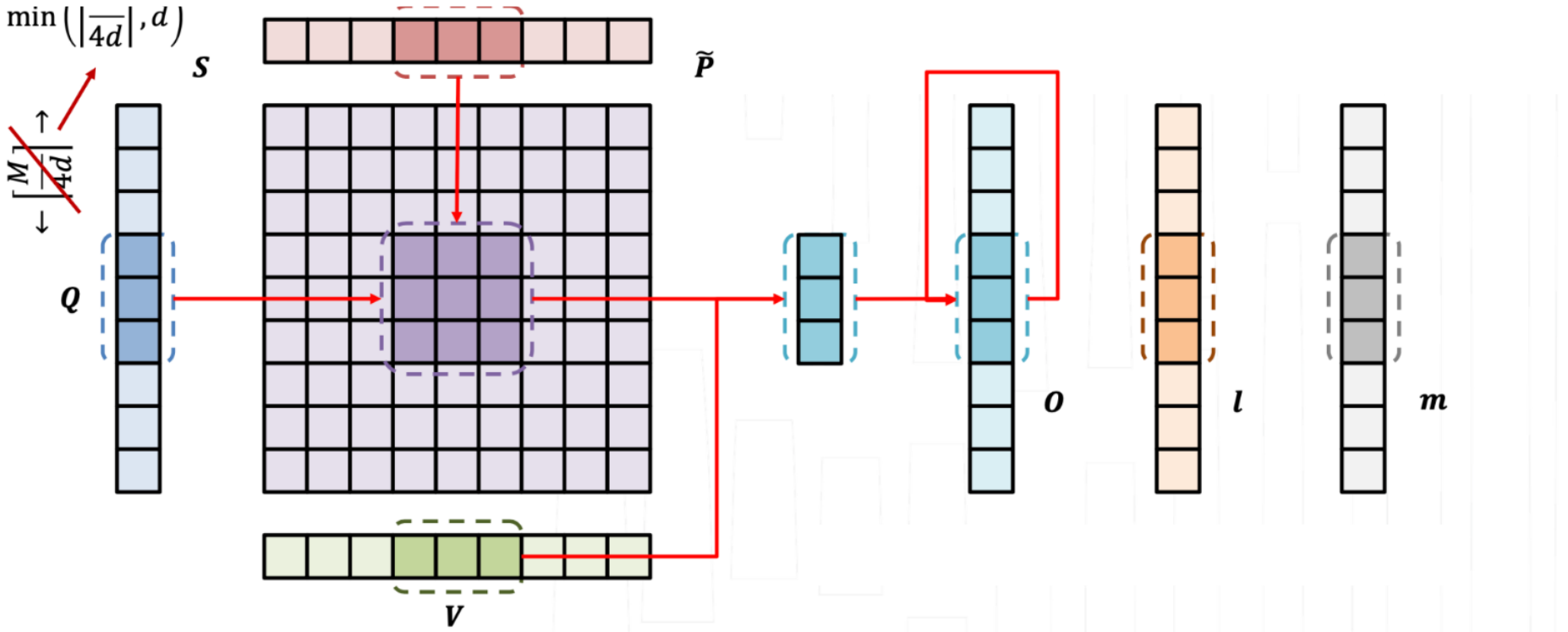
1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.

2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

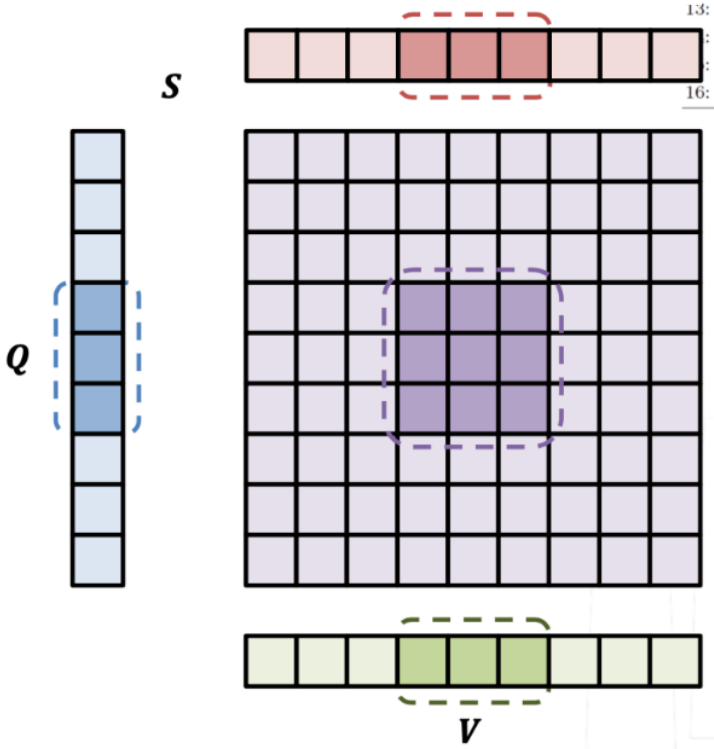
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.

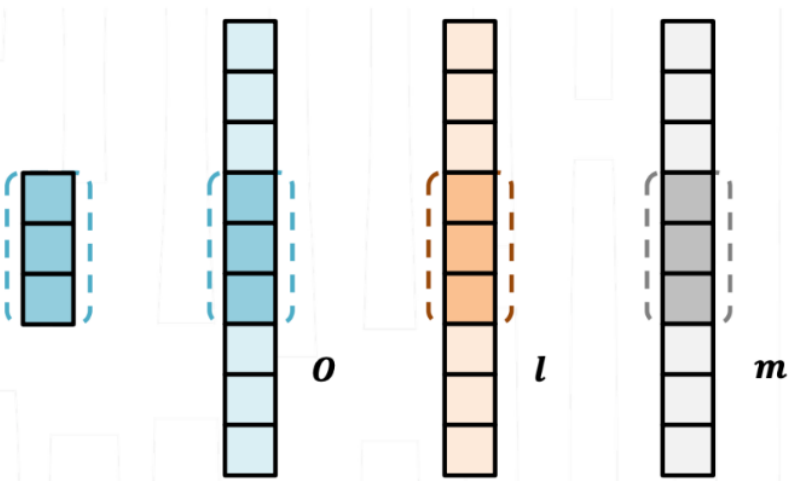
Split Blocks



FlashAttention Step by Step: Softmax Reduction (1)

Softmax Reduction



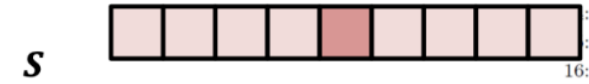


for $1 \le j \le T_c$ **do**
  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
  for $1 \le i \le T_r$ **do**
    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
    On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
    Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
    Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
  end for
end for
Return $\mathbf{O}$.

FlashAttention Step by Step: Softmax Reduction (2)

5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
end for
end for
16: Return $\mathbf{O}$.

Softmax Reduction



$$O_i = \frac{e^{Q_i K_1^T} \cdot V_1}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} + \cdots + \frac{e^{Q_i K_N^T} \cdot V_N}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}}$$



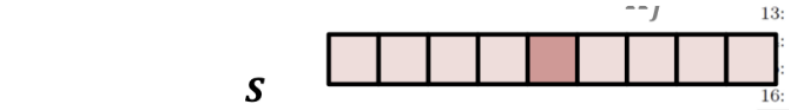
online softmax $O_1 = \frac{e^{Q_i K_1^T} \cdot V_1}{\sum_{j'}^{1} e^{Q_i K_{j'}^T}}$ $O_j = O_{j-1} \cdot \frac{\sum_{j'}^{j-1} e^{Q_i K_{j'}^T}}{\sum_{j'}^{j} e^{Q_i K_{j'}^T}} + \frac{e^{Q_i K_j^T} \cdot V_j}{\sum_{j'}^{j} e^{Q_i K_{j'}^T}}$



FlashAttention Step by Step: Softmax Reduction (3)

Softmax Reduction

5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
end for
end for
16: Return $\mathbf{O}$.

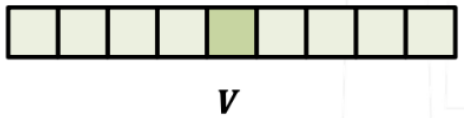


$$O_i = \frac{e^{Q_i K_1^T} \cdot V_1}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} + \cdots + \frac{e^{Q_i K_N^T} \cdot V_N}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} = \frac{e^{Q_i K_1^T} \cdot V_1}{\sum_{j'}^{1} e^{Q_i K_{j'}^T}} \cdot \frac{\sum_{j'}^{1} e^{Q_i K_{j'}^T}}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} + \cdots + \frac{e^{Q_i K_j^T} \cdot V_j}{\sum_{j'}^{j} e^{Q_i K_{j'}^T}} \cdot \frac{\sum_{j'}^{j} e^{Q_i K_{j'}^T}}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} + \cdots + \frac{e^{Q_i K_N^T} \cdot V_N}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} \cdot \frac{\sum_{j'}^{N} e^{Q_i K_{j'}^T}}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}}$$



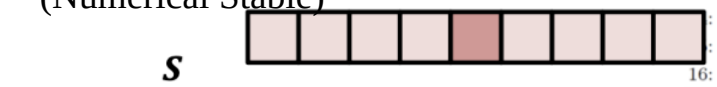
online softmax

$$O_1 = V_1 \quad O_j = O_{j-1} \cdot \frac{l_{ij-1}}{l_{ij}} + \frac{e^{S_{ij}} \cdot V_j}{l_{ij}}$$
$$l_{i1} = e^{S_{i1}} \qquad l_{ij} = l_{ij-1} + e^{S_{ij}}$$



FlashAttention Step by Step: Numerically Stable Softmax

Softmax Reduction
(Numerical Stable)



5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
0: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
1: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
2: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
3: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
end for
end for
16: Return $\mathbf{O}$.

$$O_i = \frac{e^{Q_i K_1^T} \cdot V_1}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} + \cdots + \frac{e^{Q_i K_N^T} \cdot V_N}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} = \frac{e^{Q_i K_1^T} \cdot V_1}{\sum_{j'}^{1} e^{Q_i K_{j'}^T}} \cdot \frac{\sum_{j'}^{1} e^{Q_i K_{j'}^T}}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} + \cdots + \frac{e^{Q_i K_j^T} \cdot V_j}{\sum_{j'}^{j} e^{Q_i K_{j'}^T}} \cdot \frac{\sum_{j'}^{j} e^{Q_i K_{j'}^T}}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} + \cdots + \frac{e^{Q_i K_N^T} \cdot V_N}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}} \cdot \frac{\sum_{j'}^{N} e^{Q_i K_{j'}^T}}{\sum_{j'}^{N} e^{Q_i K_{j'}^T}}$$



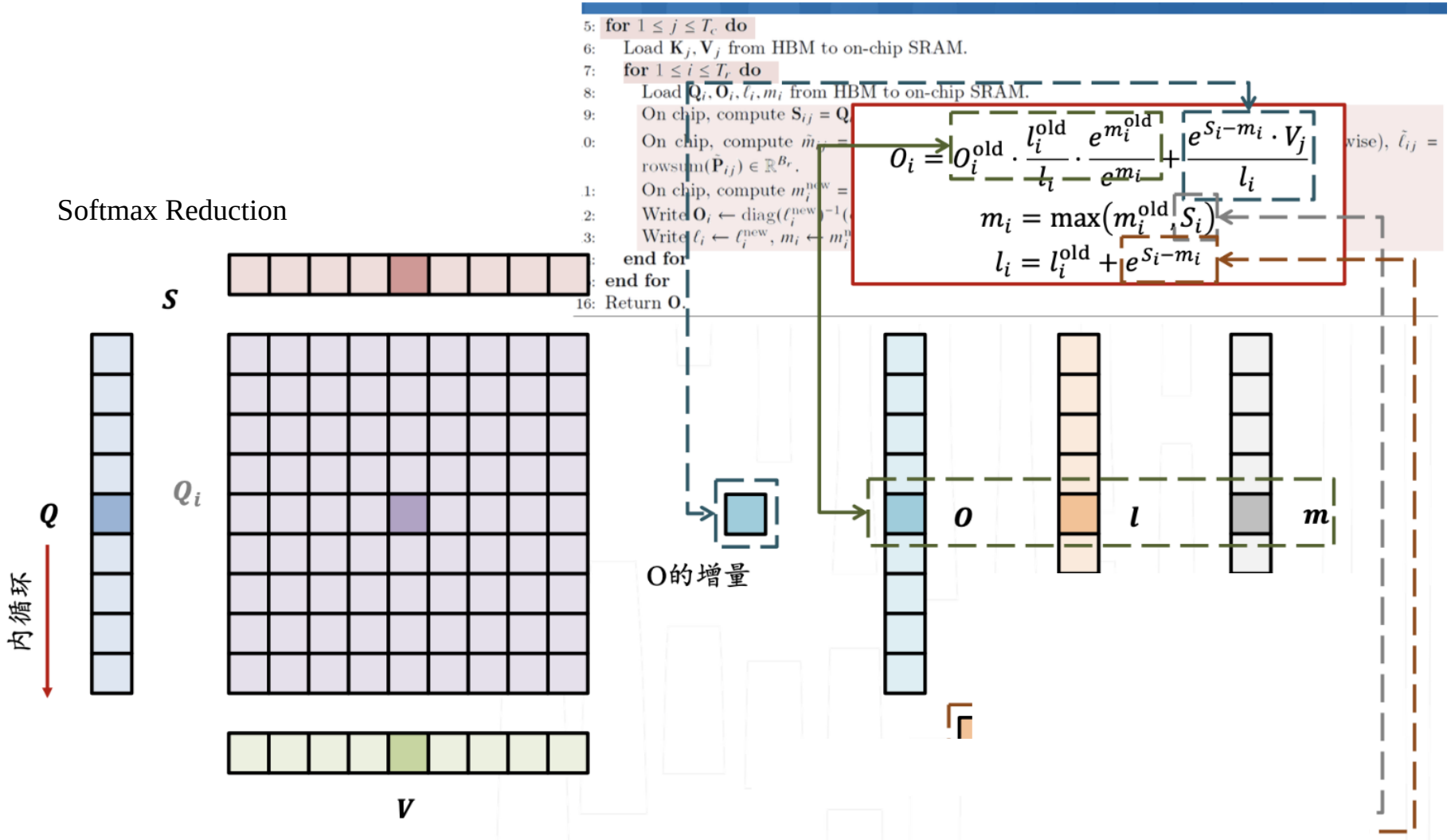
online softmax

$$O_1 = V_1$$
$$m_{i1} = S_{i1}$$
$$l_{i1} = e^{S_{i1} - m_{i1}}$$

$$O_i = O_i^{\text{old}} \cdot \frac{l_i^{\text{old}}}{l_i} \cdot \frac{e^{m_i^{\text{old}}}}{e^{m_i}} + \frac{e^{S_i - m_i} \cdot V_j}{l_i}$$
$$m_i = \max(m_i^{\text{old}}, S_i)$$
$$l_i = l_i^{\text{old}} + e^{S_i - m_i}$$

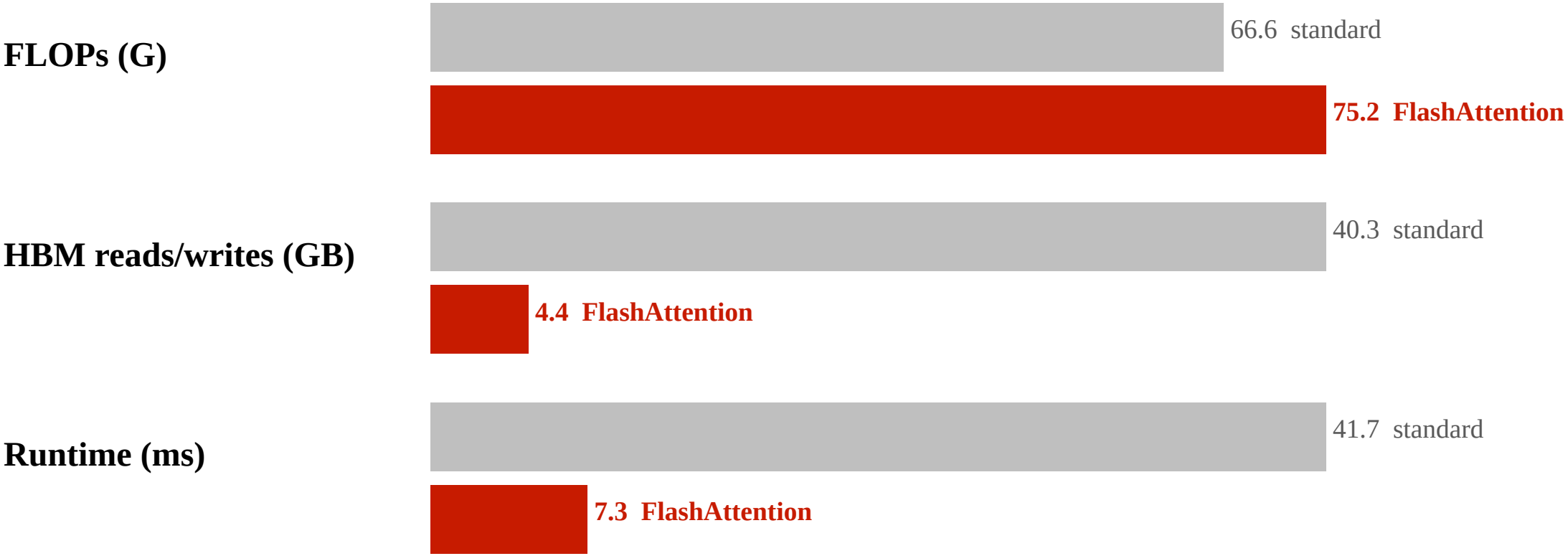


FlashAttention Step by Step: Putting It Together



Where the Speedup Comes From: IO, Not FLOPs

GPT-2 medium attention on an A100 (forward + backward)



FlashAttention does MORE arithmetic (recomputation) but moves 9× fewer bytes — and runs 5.7× faster.

Dao et al. FlashAttention (NeurIPS 2022), Figure 1 (left).

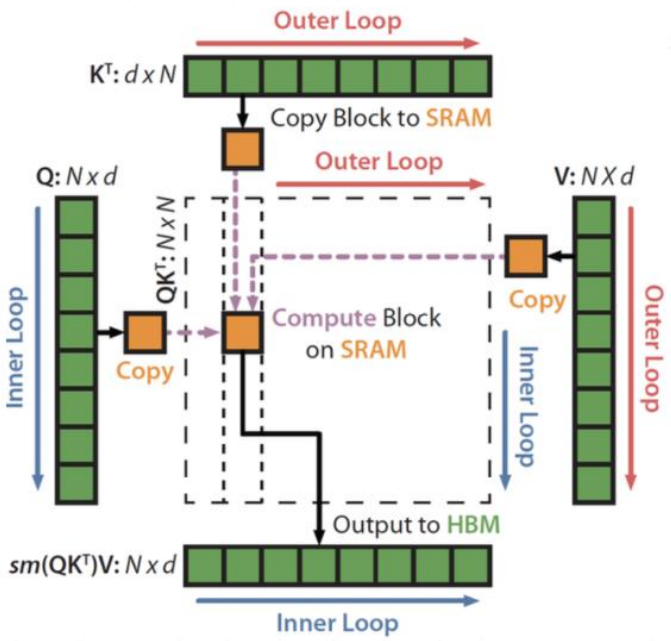
FlashAttention: Summary

Summary: Split blocks, Update Softmax, Complexity= $\mathcal{O}(Nd \cdot Nd/M) = \mathcal{O}(N^2d^2/M)$

Algorithm 1 FLASHATTENTION

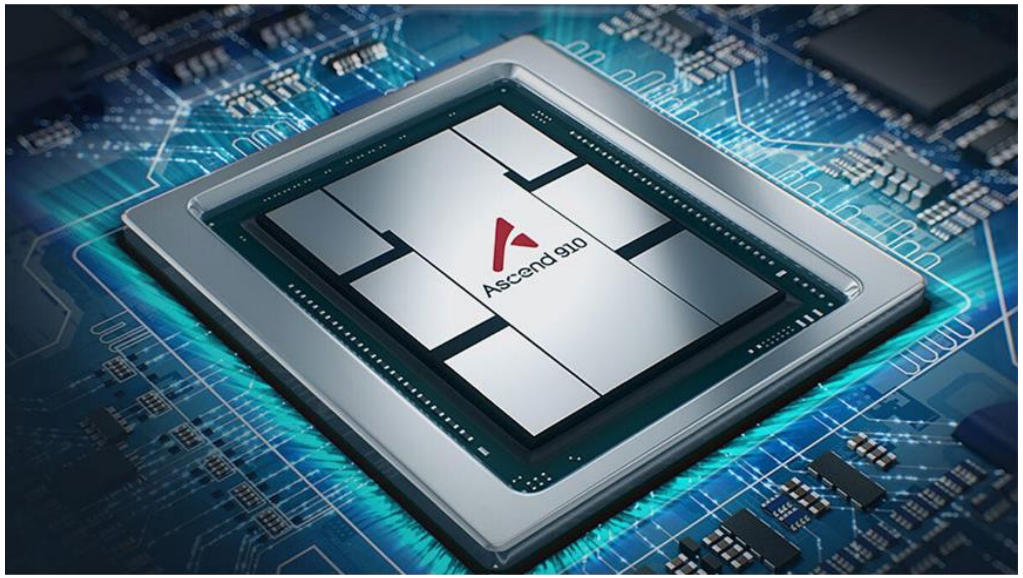
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.



Remark: Huawei Ascend Chips Are Bounded by Memory Bandwidth

- How to design **Ascend-friendly** Architecture?



Join us if you are interested (1k-10k Ascend NPU chips). wangbenyou@cuhk.edu.cn

5.5 Serving Systems: vLLM

Pages, batches and sharing: an operating-system view of LLM serving.

vLLM Architecture Overview

The four Components:

Scheduler

- Decides which requests to process in each iteration
- First-Come-First-Served (FCFS) with preemption

KV Cache Manager (Block Manager)

- Manages physical block allocation/deallocation
- Maintains block tables for all active sequences

Worker(s)

- Execute model forward pass on GPU(s)
- Run PagedAttention kernels

Tokenizer + Sampler

- Input tokenization and output token sampling

Static vs. Continuous Batching

Static batching: wait for the longest request

	t1	t2	t3	t4	t5	t6	t7	t8	t9
slot 1	A	A	A	idle	idle	idle	idle	E	
slot 2	B	B	B	B	B	B	B	F	
slot 3	C	C	idle	idle	idle	idle	idle	G	
slot 4	D	D	D	D	D	idle	idle		

Continuous batching: refill a slot as soon as it frees up

	t1	t2	t3	t4	t5	t6	t7	t8	t9
slot 1	A	A	A	E	E	E	E		
slot 2	B	B	B	B	B	B	B		
slot 3	C	C	F	F	F	F	F		
slot 4	D	D	D	D	D	G	G	G	

Schedule per iteration, not per batch.

No idle slots, new requests start immediately.

Up to 23× throughput reported vs. static batching.

Yu et al. Orca: A Distributed Serving System for Transformer-Based Generative Models (OSDI 2022) · Anyscale blog (2023).

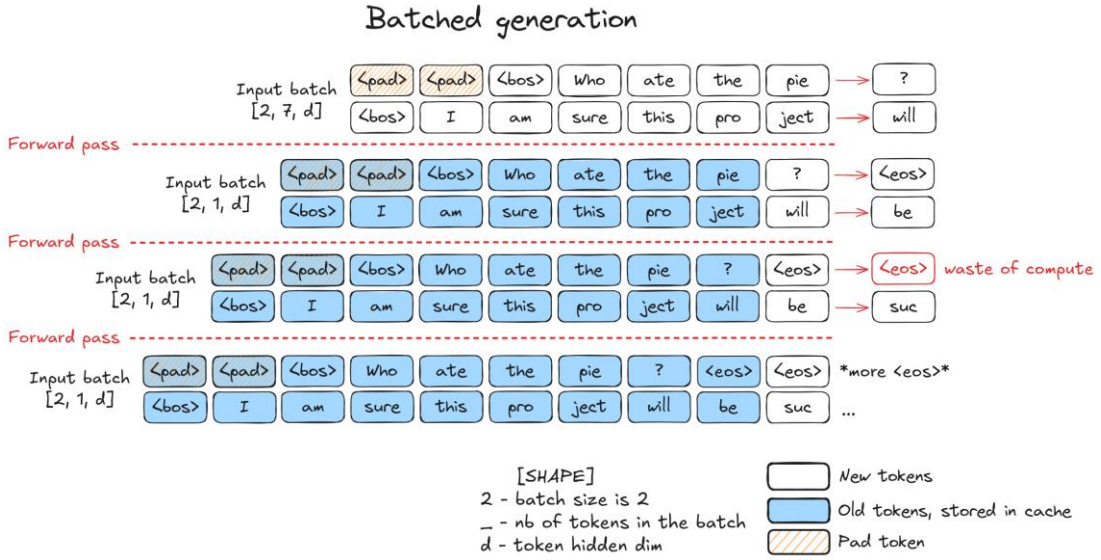
Continuous Batching

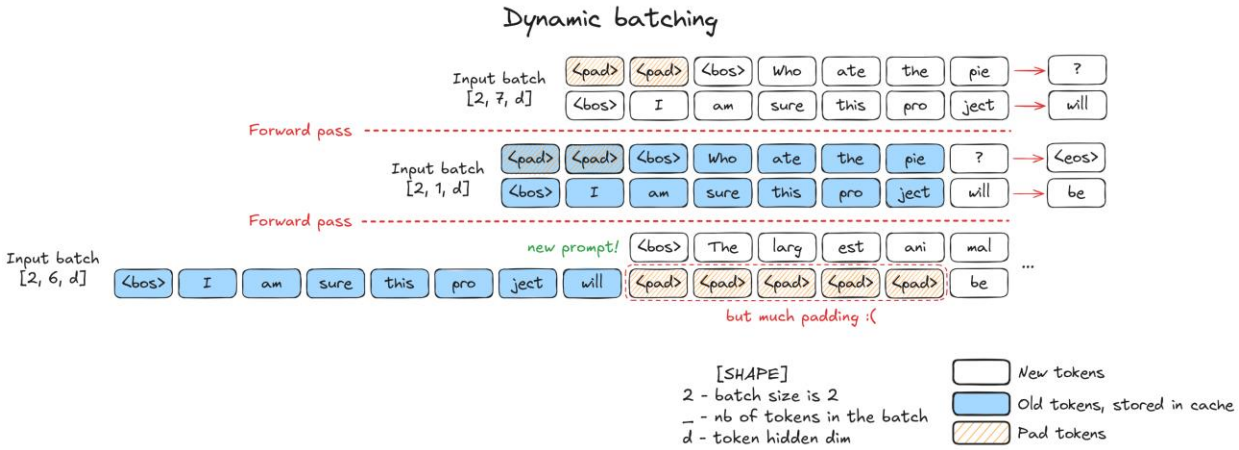
Traditional batching (static):

- - Wait for a batch of requests, process together
- - All requests must finish before new ones can start
- - Short requests wait for long ones -> wasted compute

Continuous batching (iteration-level):

- At each decode step, check if any request has finished
- Immediately insert new requests into the batch
- Each request independently enters and exits the batch





Benefits:

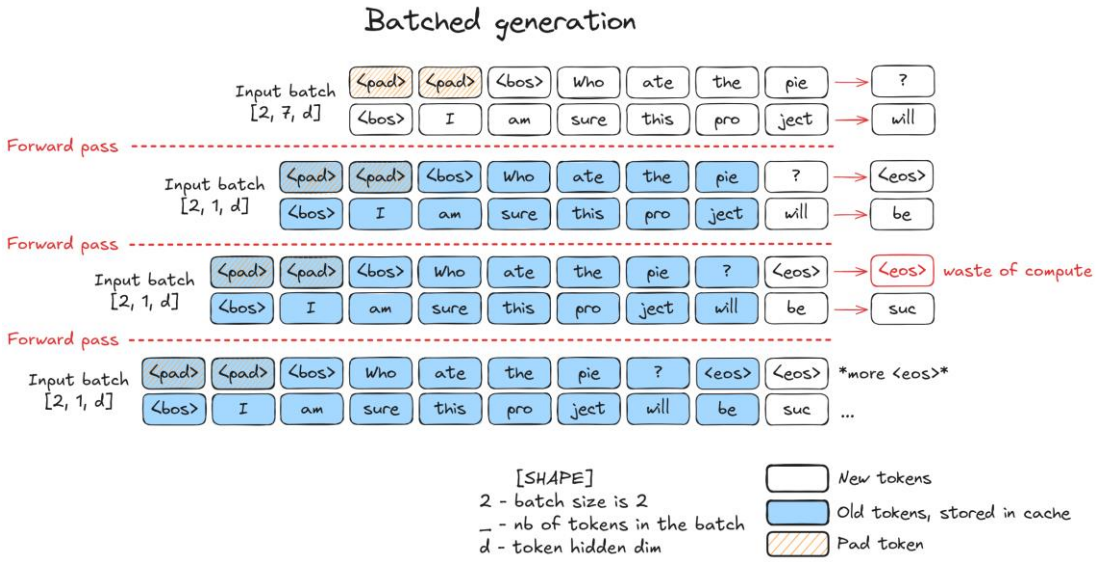
- No "batch bubble" from waiting for slowest request
- Higher GPU utilization
- Lower average latency

https://huggingface.co/blog/continuous_batching

Continuous Batching

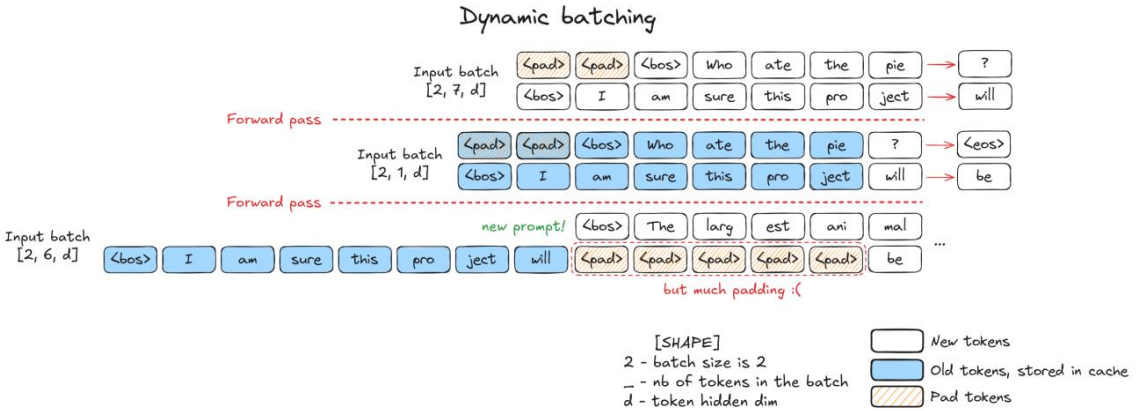
Traditional batching (static):

- - Wait for a batch of requests, process together
- - All requests must finish before new ones can start
- - Short requests wait for long ones -> wasted compute



Continuous batching (iteration-level):

- At each decode step, check if any request has finished
- Immediately insert new requests into the batch
- Each request independently enters and exits the batch

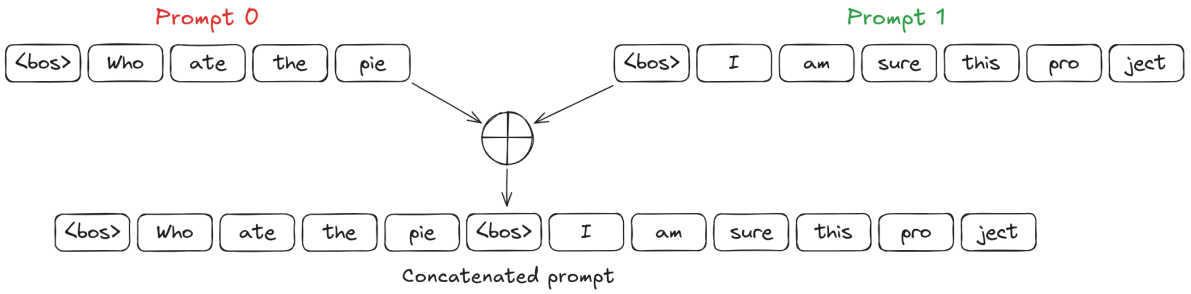


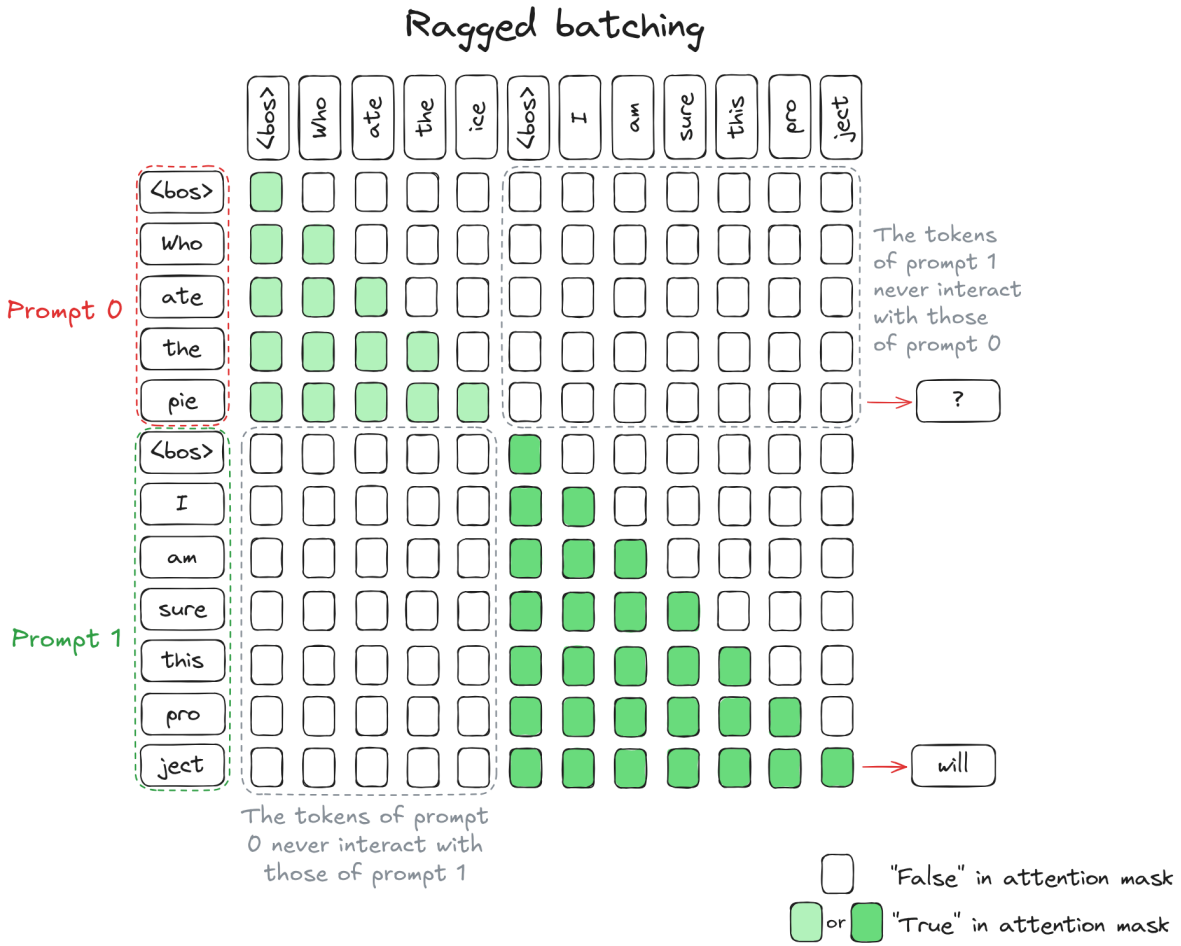
Benefits:

- No "batch bubble" from waiting for slowest request
- Higher GPU utilization
- Lower average latency

https://huggingface.co/blog/continuous_batching

Ragged Batching for Continuous Batching





Memory Sharing: Parallel Sampling

Use case: generate N completions for the same prompt (e.g., N=4)

Naive approach: duplicate the prompt KV cache N times

- - 4 copies of the same data -> 4x memory waste!

PagedAttention approach: Copy-on-Write (CoW)

- - All N sequences share the same physical blocks for the prompt
- - Block table entries point to the same physical blocks
- - Reference counter tracks how many sequences share each block

When sequences diverge:

- - Only copy the block that differs (like OS CoW fork)
- - Other blocks remain shared

Memory saving: up to Nx reduction for prompt KV cache

Memory Sharing: Beam Search

- Beam search: keep top-K candidates at each step

- Traditional: each beam maintains a full copy of KV cache
- - K beams x full sequence KV -> huge memory cost

- vLLM with PagedAttention:
- - Beams share blocks for common prefix
- - Only divergent tokens need new blocks
- - When a beam is pruned, just decrement reference count
- - Physical blocks freed only when ref_count reaches 0

- Analogy: garbage collection with reference counting

- Memory saving: typically 50-70% reduction for beam search

Preemption and Swapping

What if GPU memory runs out?

vLLM supports preemption (like OS process scheduling):

Option 1: Swapping

- - Move KV cache blocks from GPU memory to CPU memory
- - Resume later by swapping back
- - Like OS swapping pages to disk

Option 2: Recomputation

- - Discard KV cache blocks entirely
- - Recompute them when the request resumes
- - More expensive but no CPU memory needed

Scheduler uses FCFS policy:

- - Preempt the most recently arrived request first
- - Ensures fairness and avoids starvation

6 Outlook

Hardware, algorithms and systems are now co-designed. What will make tokens 10× cheaper next?

Outlook (2026): Where the Next 10× Comes From

Sparse attention in production	DeepSeek-V3.2 sparse attention (indexer + top-k tokens); DeepSeek-V4 mixes compressed and sparse attention for 1M-token context.
Linear / hybrid attention	Gated DeltaNet and Mamba layers mixed with full attention (Qwen3-Next, Kimi Linear, Nemotron-H): KV cache stops growing for most layers.
Low-bit everything	FP8 training is standard; NVFP4 inference on Blackwell and Rubin — vendors claim ~10× lower cost per token for large MoE models.
Disaggregated serving	Separate prefill and decode (and even attention and FFN) onto different GPUs; KV cache moves over fast networks (llm-d, Mooncake).
Efficient reasoning	Thinking tokens dominate cost: budgets, adaptive effort, multi-token prediction and speculative decoding for long chains of thought.

The frontier is co-design: model architecture, numerics, kernels and serving systems are now built together.

DeepSeek-V3.2 (2025) · SparDA, arXiv:2606.04511 (2026) · NVIDIA Vera Rubin platform (CES/GTC 2026) · Red Hat llm-d (2025). Vendor claims unverified.

Future Remarks: Hybrid Efficiency

Hybrid efficiency QLoRA LongLoRA QMOE Towards Adapting to LLMs

Group Discussions

Acknowledgement

- https://hanlab.mit.edu/courses/2023-fall-65940
- https://hanlab.mit.edu/courses/2022-fall-6s965
- https://docs.google.com/presentation/d/1EUV7W7X_w0BDrscDhPg7lMGzJCkeaPkGCJ3bN8dluXc/edit?resourcekey=0-7Nz5A7y8JozyVrnDtcEKJA
- https://www.youtube.com/watch?v=bQrdd3BI_fM
- https://github.com/princeton-nlp/LLM-Shearing
- https://www.youtube.com/watch?v=S-8yr_RibJ4
- https://www.youtube.com/watch?v=y9PHWGOa8HA
- https://www.youtube.com/watch?v=ZsompoMeIcI
- https://www.youtube.com/watch?v=D2DdEstvS30