CSC6203/CIE6021: Large Language Model

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

Lecture 5: Efficiency in LLMs

Winter 2023
Benyou Wang
School of Data Science

Recap

Overview

- LLM training
  - LLM Pretraining (including Word Tokenization)
  - Instruction Finetuning
  - Reinforcement Learning from Human Feedback
- LLM Evaluation

Tokenization

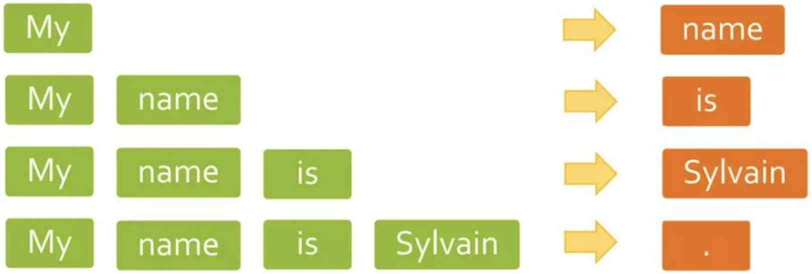
- Before Tokenization : **This is tokenizing**
- After Tokenization : **This is token izing**

Character-level tokenization?
Word-level tokenization?
Why Subword tokenization?

Pre-training

Example **plain** text （**do not need supervised data, e.g. web and books**）

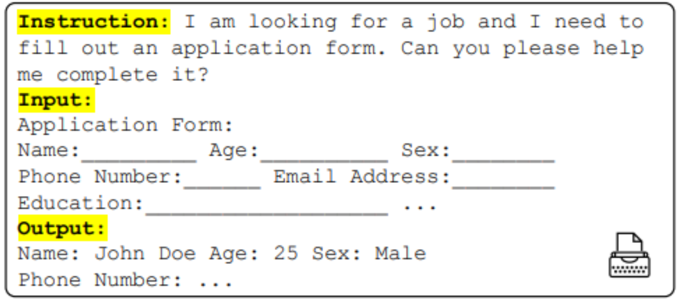
The Large Language Model (LLM) represents a cutting-edge innovation in the field of artificial intelligence, harnessing vast amounts of textual data to provide nuanced responses and generate coherent narratives. As a descendant of OpenAI's renowned GPT series, the LLM showcases the rapid evolution of machine learning capabilities, embodying an unparalleled ability to comprehend, generate, and assist in myriad linguistic tasks. This technological marvel encapsulates the collective knowledge of countless sources and offers a tantalizing glimpse into the future of human-computer symbiosis, where the boundaries between natural and artificial intelligence become increasingly blurred.



Training with purely next word (token) prediction

Instruction fine-tuning (supervised fine-tuning)

Usually a triplet **(instruction, input, output) -- need supervised data**



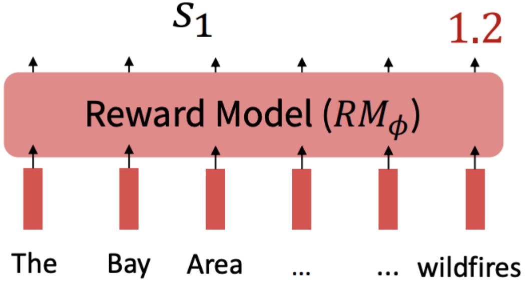
```
Instruction: I am looking for a job and I need to
fill out an application form. Can you please help
me complete it?
Input:
Application Form:
Name:_________ Age:__________ Sex:________
Phone Number:______ Email Address:________
Education:___________________ ...
Output:
Name: John Doe Age: 25 Sex: Male
Phone Number: ...
```

Training with next word (token) prediction but usually only for the output part

Reinforcement Learning from ~~Human~~ AI Feedback

An earthquake hit San Francisco. There was minor property damage, but no injuries. > A 4.2 magnitude earthquake hit San Francisco, resulting in massive damage. > The Bay Area has good weather but is prone to earthquakes and wildfires.

s_1 s_3 s_2



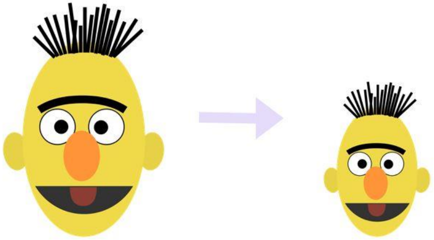
Bradley-Terry [1952] paired comparison model

$$J_{RM}(\phi) = -\mathbb{E}_{(s^w,s^l)\sim D}[\log \sigma(RM_\phi(s^w) - RM_\phi(s^l))]$$

"winning" sample

"losing" sample

s^w should score higher than s^l

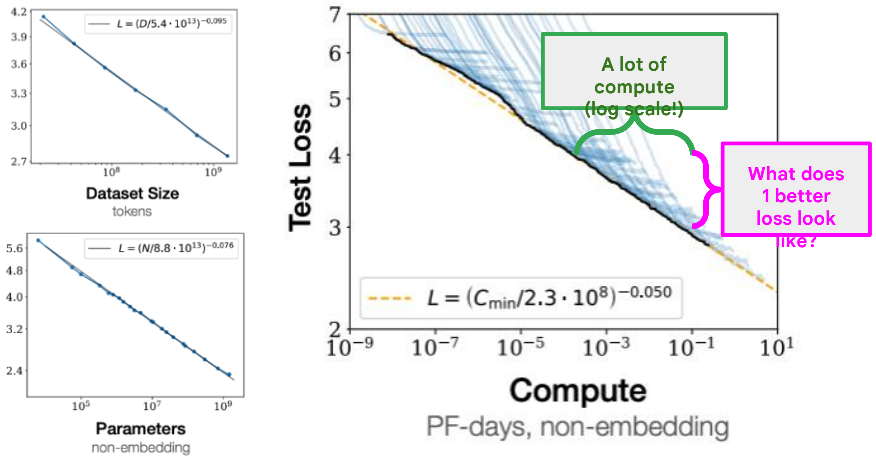


Today’s main course： efficiency

又想马儿不吃草，也想马儿跑得快！

Why do we need efficiency?

LLMs follow Scaling Laws



A lot of compute (log scale!)

What does 1 better loss look like?

[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.*

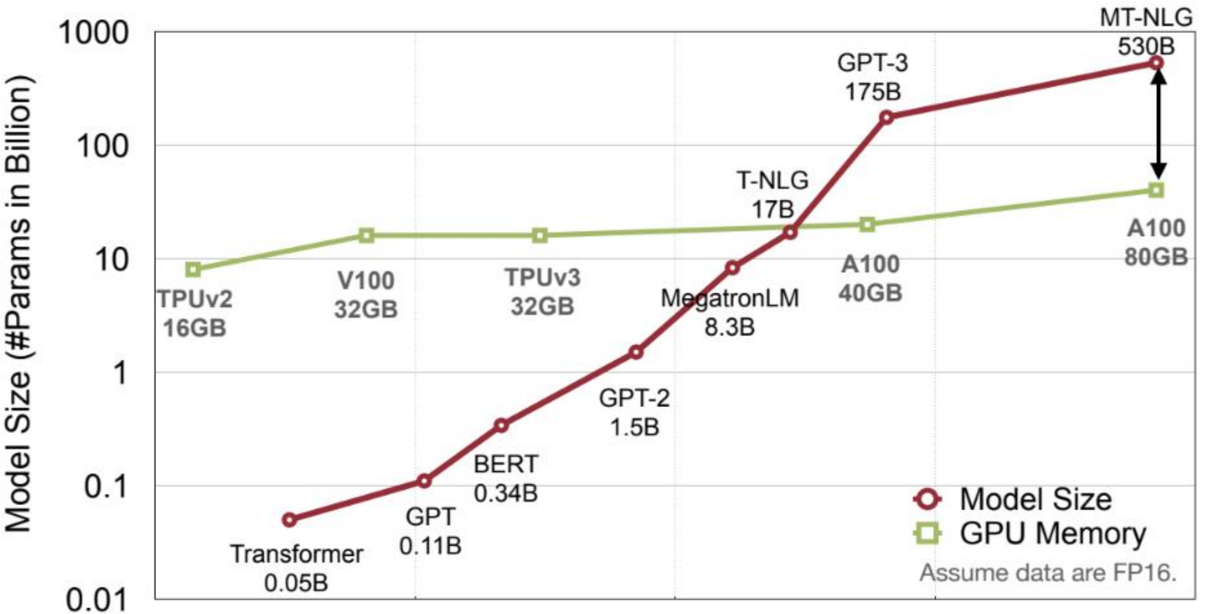
Jason's rephrase: You should expect to get a better language model if you scale up model size, dataset size, and amount of compute.

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

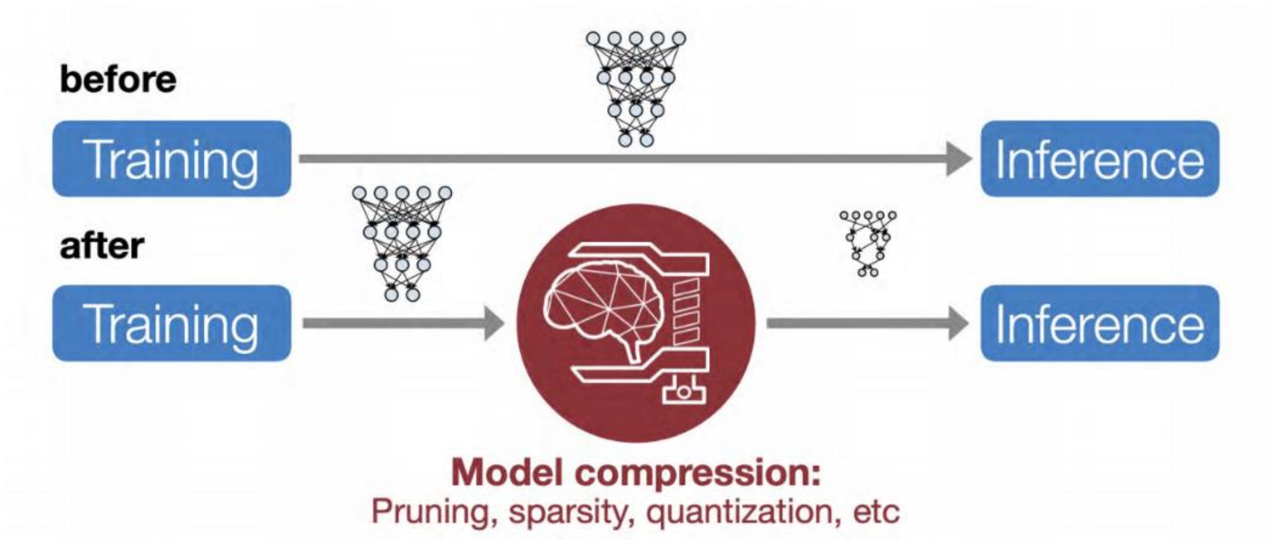
Note 2: By the way, data is unlabeled (self-supervised) data from the internet (e.g., common crawl).

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



Outline

1. Overview of Efficiency in LLMs
2. Efficiency before the LLM Era
    - Efficiency **beyond** Transformer (Quantization, Pruning, Knowledge distillation etc.)
    - Efficiency **within** Transformer (*Spars*ity e.g., Mixture of Expert and efficiency in *long context*)
    - **Parameter-efficient** finetuning (Lora, adapter, prompt, moe) and modularization
3. Efficiency after the LLM Era
    a. **Memory-efficient** Training: ZeRO, LOMO, Distributed Training, Flash-Attention, Lora/QLORA, Language model Inference: Early existing and Speculative decoding
4. Future direction

Efficiency before LLMs

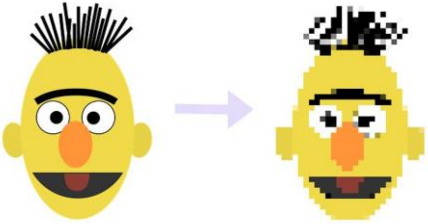
- Efficiency within Transformer
  - Sparsity (e.g., Mixture of Expert)
  - Long Attention (e.g., quadratic computing -> leaning computing, w.r.t. the sequence length)

- Efficiency beyond Transformer
  - Quantization
  - Pruning
  - Knowledge distillation

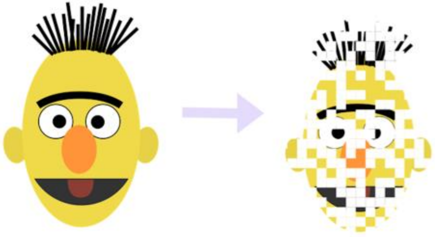
- Parameter-efficient Finetuning
  - Save space for fine-tuning.

- ~~Others (e.g., parameter compression and parameter sharing)~~
  - Parameter compression does not necessarily lead to faster inference.

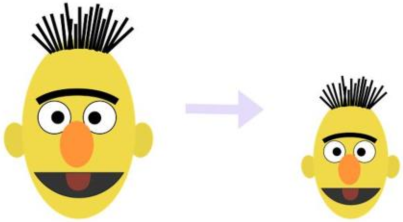
Efficiency beyond Transformer



Quantization
“Low resolution”



Pruning
Removing weight connections

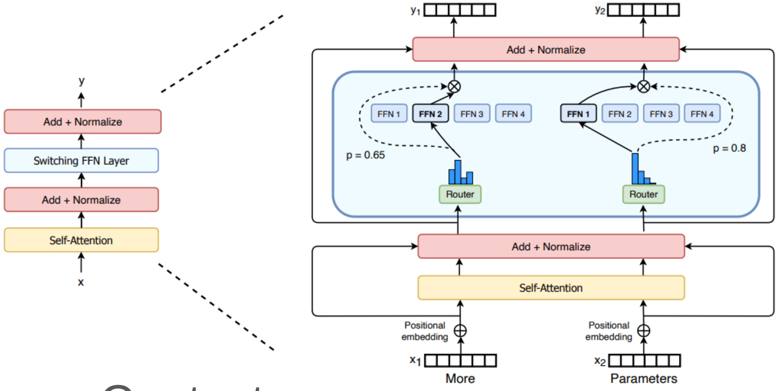


Knowledge distillation
Learning smaller models from big ones

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

Efficiency within Transformer

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



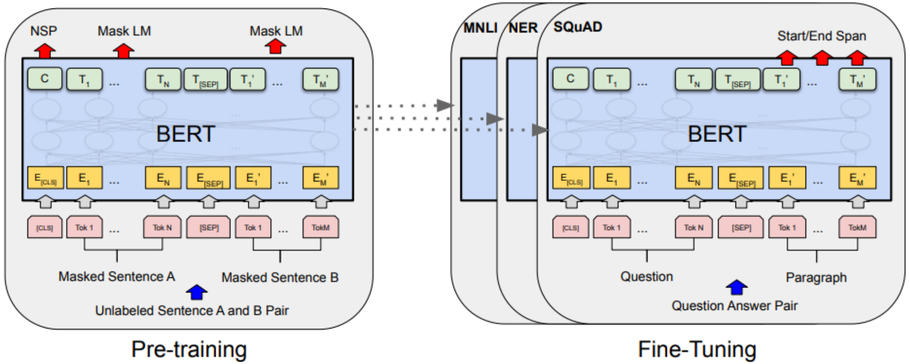
- Efficiency for Long Context
  - Computing complexity of is O($N^2 D$), which is quadratic to the sequence length

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

From time-efficiency vs. space-efficiency

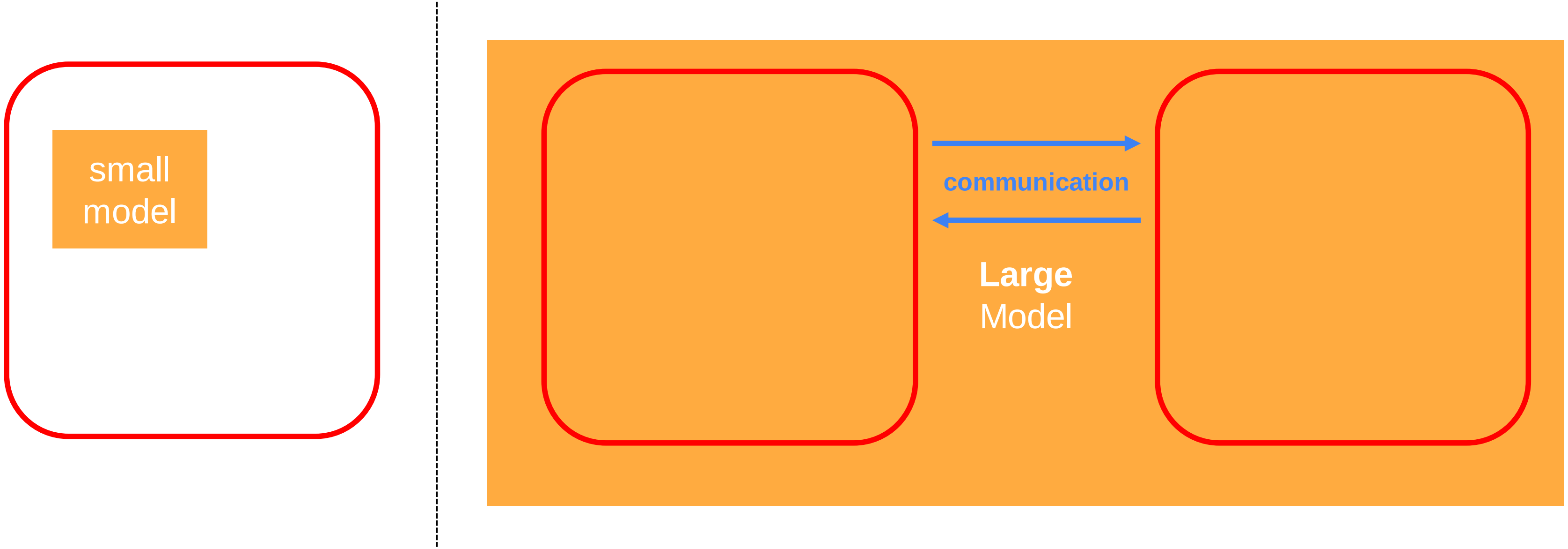
Each task has a full finetuned model

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



Parameter-efficient finetuning

Space-efficient: from parameter to memory



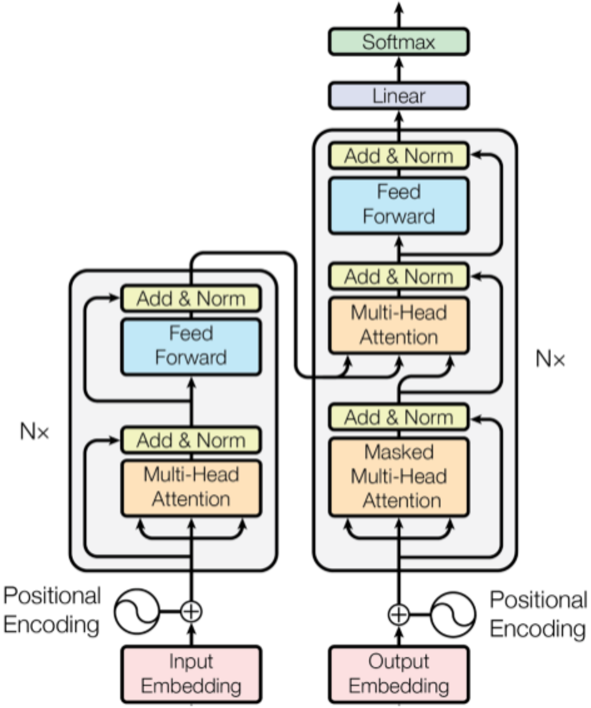
In LLM era, a whole model might not be stored in a GPU memory, **communication costs** may be the bottleneck!

Efficiency after LLMs

- All Efficiency methods before LLMs (within/beyond transformer)
- + **Communication** costs
- **+ Memory** saving
- Speculative decoding (only for language model)

Efficiency within Transformers

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:

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

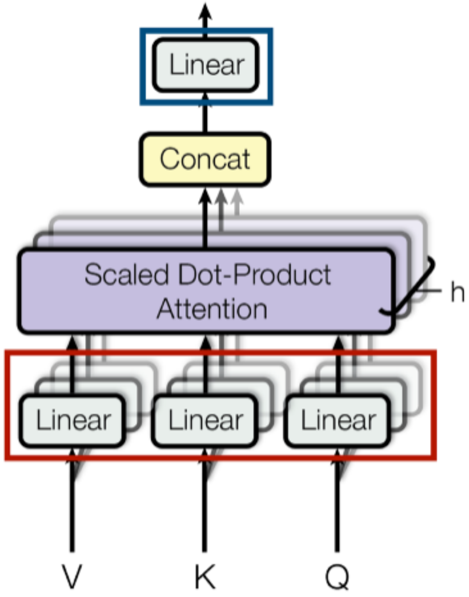
- The decoder generates the output sequence of symbols one element at a time in an **auto-regressive** manner.

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

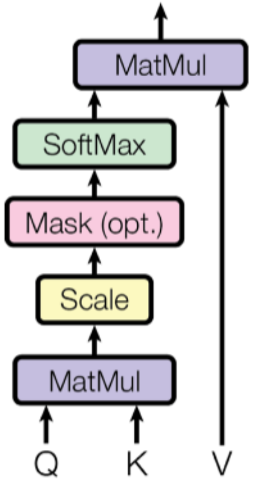
- **Project** Q, K and V with h **different**, learned linear projections.
- Perform the **scaled dot-product attention** function on each of these projected versions of Q, K and V **in parallel**.
- **Concatenate** the output values.
- **Project** the output values again, resulting in the final values.

$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})\boxed{W^O}$$

$$\text{where } \mathrm{head_i} = \mathrm{Attention}(Q\boxed{W_i^Q}, K\boxed{W_i^K}, V\boxed{W_i^V})$$



Scaled Dot-Product Attention



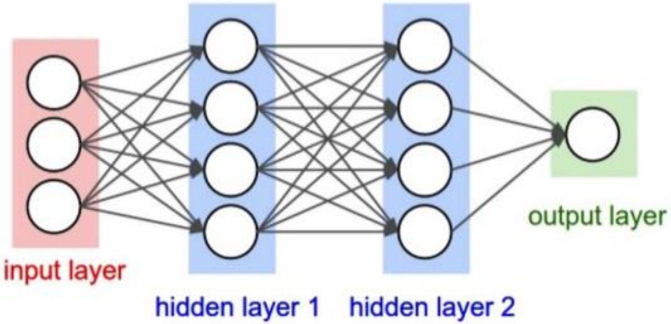
$$\mathrm{Attention}(Q,K,V)$$
$$= \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

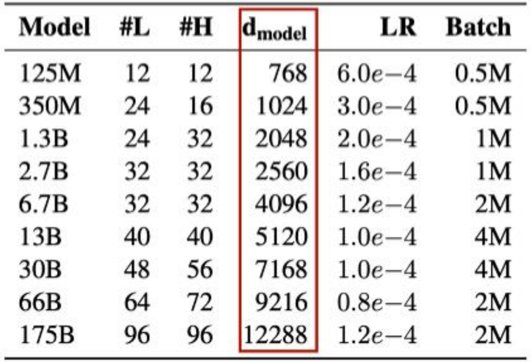
Recap: Transformers - Feed-Forward Network (FFN)

- 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.

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

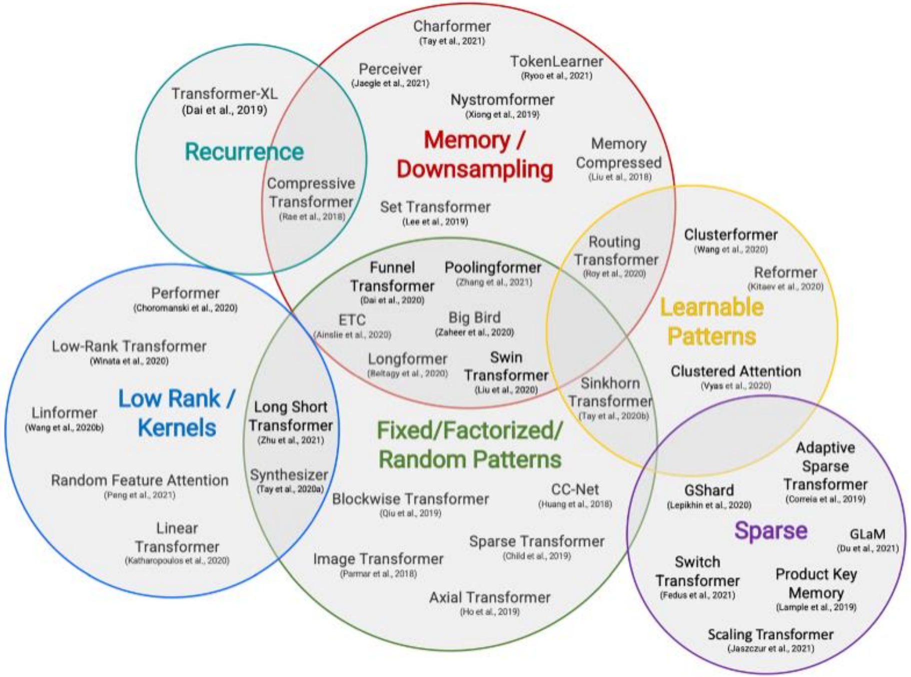
- The middle hidden size is usually larger than and input and output size (**inverted bottleneck**).





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



Recurrence

Transformer-XL (Dai et al., 2019)

Compressive Transformer (Rae et al., 2018)

Memory / Downsampling

Charformer (Tay et al., 2021)

Perceiver (Jaegle et al., 2021)

TokenLearner (Ryoo et al., 2021)

Nystromformer (Xiong et al., 2019)

Memory Compressed (Liu et al., 2018)

Set Transformer (Lee et al., 2019)

Routing Transformer (Roy et al., 2020)

Funnel Transformer (Dai et al., 2020)

Poolingformer (Zhang et al., 2021)

ETC (Ainslie et al., 2020)

Big Bird (Zaheer et al., 2020)

Longformer (Beltagy et al., 2020)

Swin Transformer (Liu et al., 2020)

Learnable Patterns

Clusterformer (Wang et al., 2020)

Reformer (Kitaev et al., 2020)

Clustered Attention (Vyas et al., 2020)

Sinkhorn Transformer (Tay et al., 2020b)

Low Rank / Kernels

Performer (Choromanski et al., 2020)

Low-Rank Transformer (Winata et al., 2020)

Linformer (Wang et al., 2020b)

Random Feature Attention (Peng et al., 2021)

Linear Transformer (Katharopoulos et al., 2020)

Long Short Transformer (Zhu et al., 2021)

Synthesizer (Tay et al., 2020a)

Fixed/Factorized/ Random Patterns

Blockwise Transformer (Qiu et al., 2019)

CC-Net (Huang et al., 2018)

Sparse Transformer (Child et al., 2019)

Image Transformer (Parmar et al., 2018)

Axial Transformer (Ho et al., 2019)

Sparse

GShard (Lepikhin et al., 2020)

Adaptive Sparse Transformer (Correia et al., 2019)

GLaM (Du et al., 2021)

Switch Transformer (Fedus et al., 2021)

Product Key Memory (Lample et al., 2019)

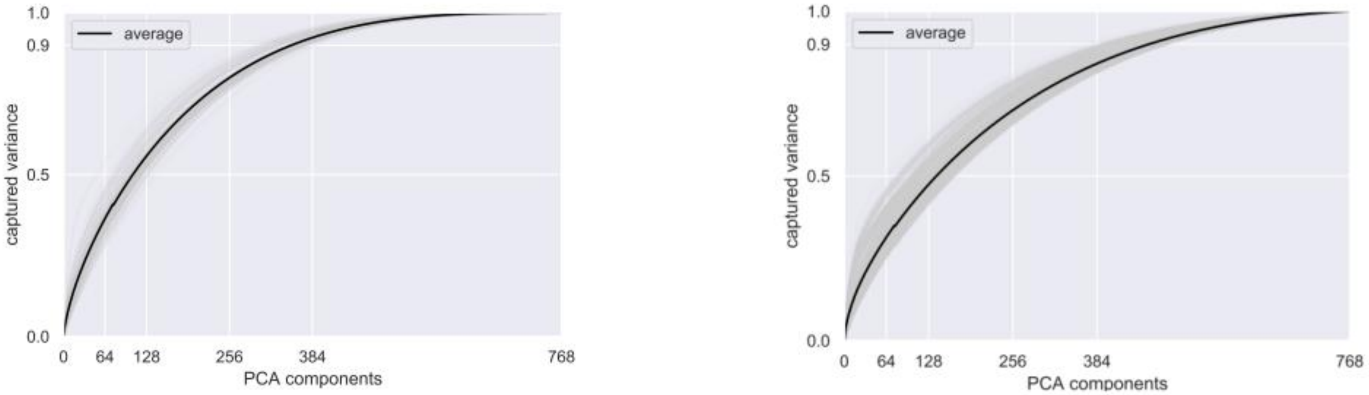
Scaling Transformer (Jaszczur et al., 2021)

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

Why could we achieve efficiency within Transformer but maintaining the performance ?

马儿可以不吃草，也跑得快吗?

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 (可分解性)

- A computing module f is **decomposable** if its sub-components {g1, g2,···gH} could be independently calculated without interactions: f(x) = δ(g1(x), g2(x),···, 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

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

Feed-forward network is also decomposable

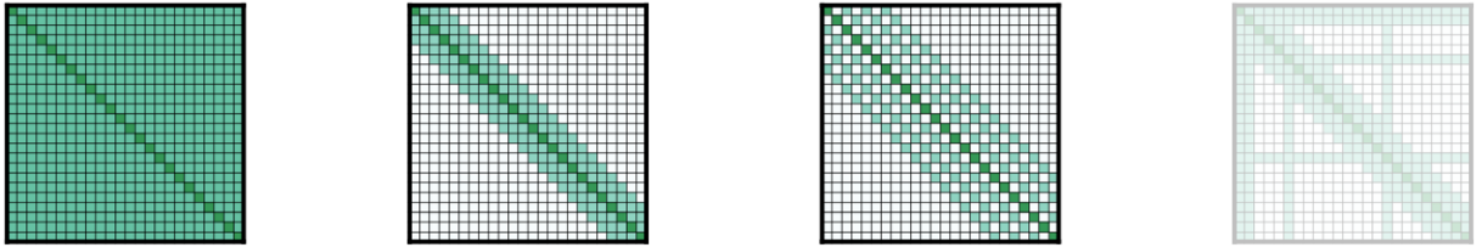
- It performs like a multi-head mechanism

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

Efficient Transformers - efficient Attention

Sparse Attention - LongFormer

Local Attention + Global Attention



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

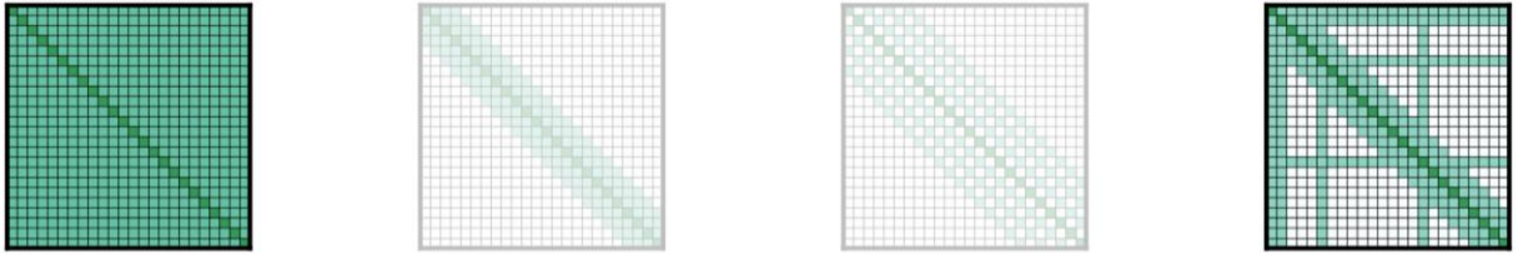
- Attention with **sliding window** (analogous to CNNs):
  - A fixed-size window attention surrounding each token.
  - The complexity is reduced from $O(N^2)$ to **O(N x W)**, where W is the window size.

- 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

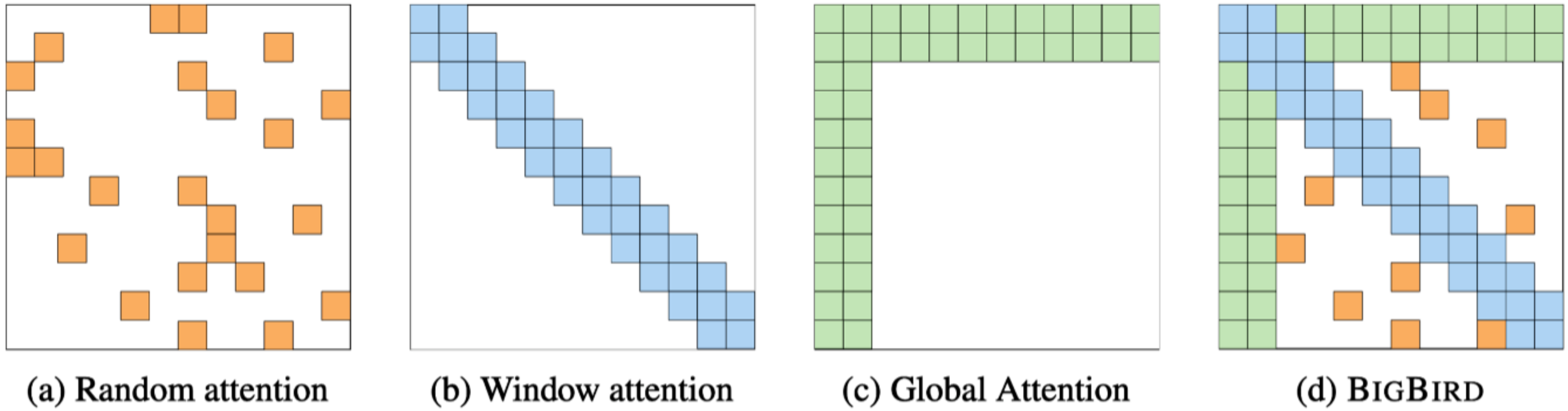
- **Global attention** added on a few **pre-selected** input locations:
  - Classification: The special token ([CLS]), aggregating the whole sequence.
  - QA: All question tokens, allowing the model to compare the question with the document.

- 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]

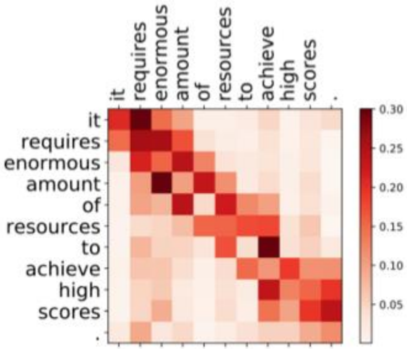
Sparse Attention - Lite Transformer

Local Convolution + Global Attention

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

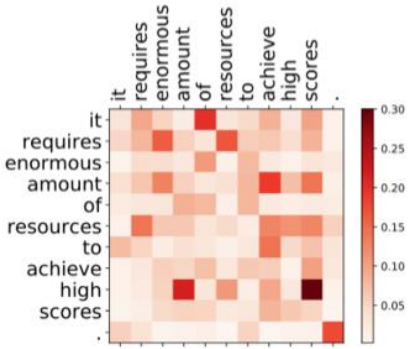
Original Attention

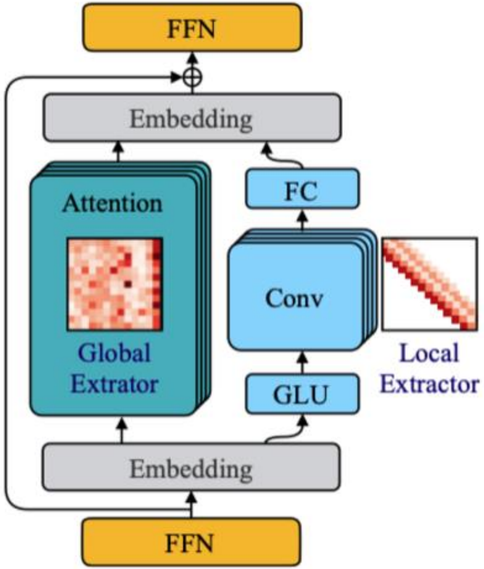
(Too much emphasize on local feature extraction)



Attention in LSRA

(Dedicated for global feature extraction)





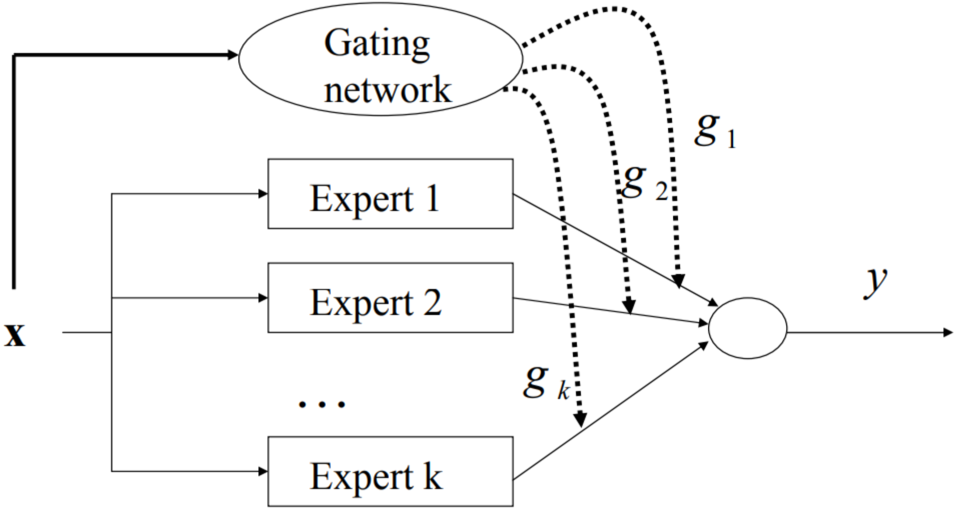
Lite Transformer with Long-Short Range Attention [Wu *et al.*, 2020]

Efficient Transformers - efficient FFNs

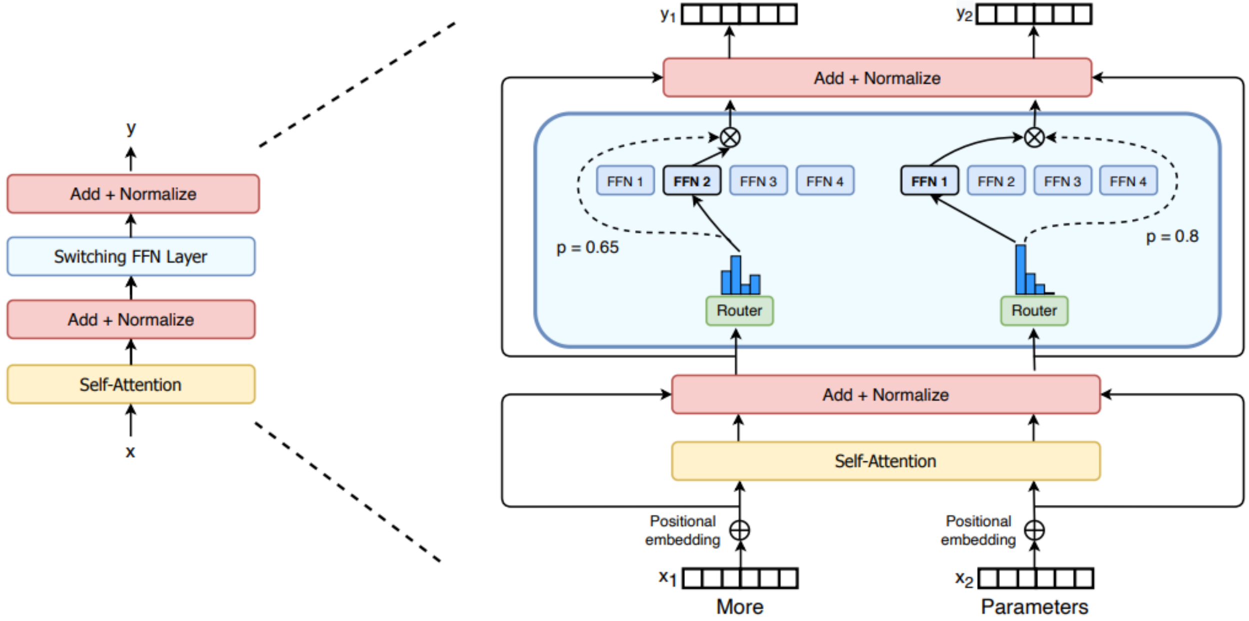
Mixture of Expert

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

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

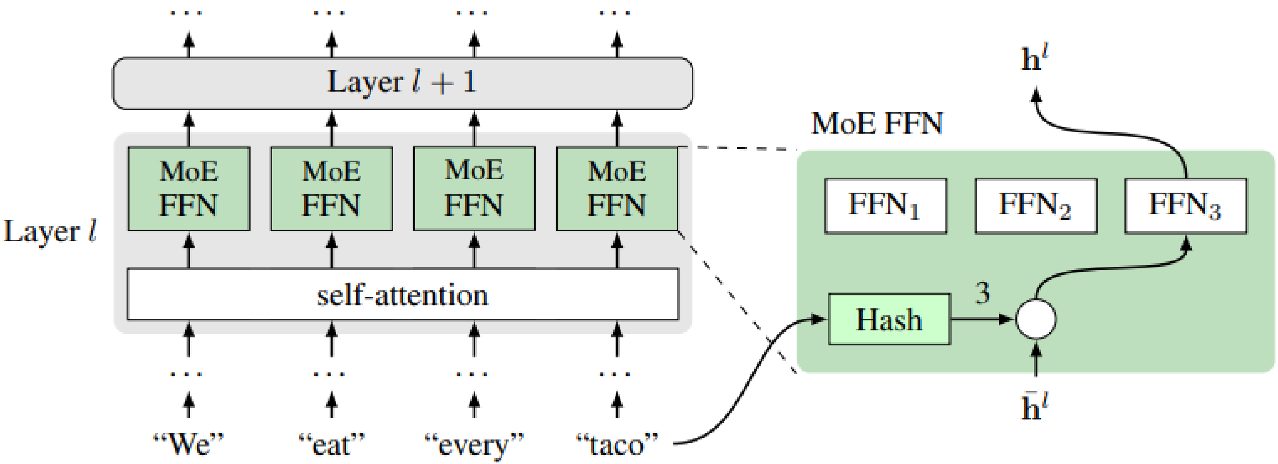


MOE in Transformer



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

How to gate? An example of Hash



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

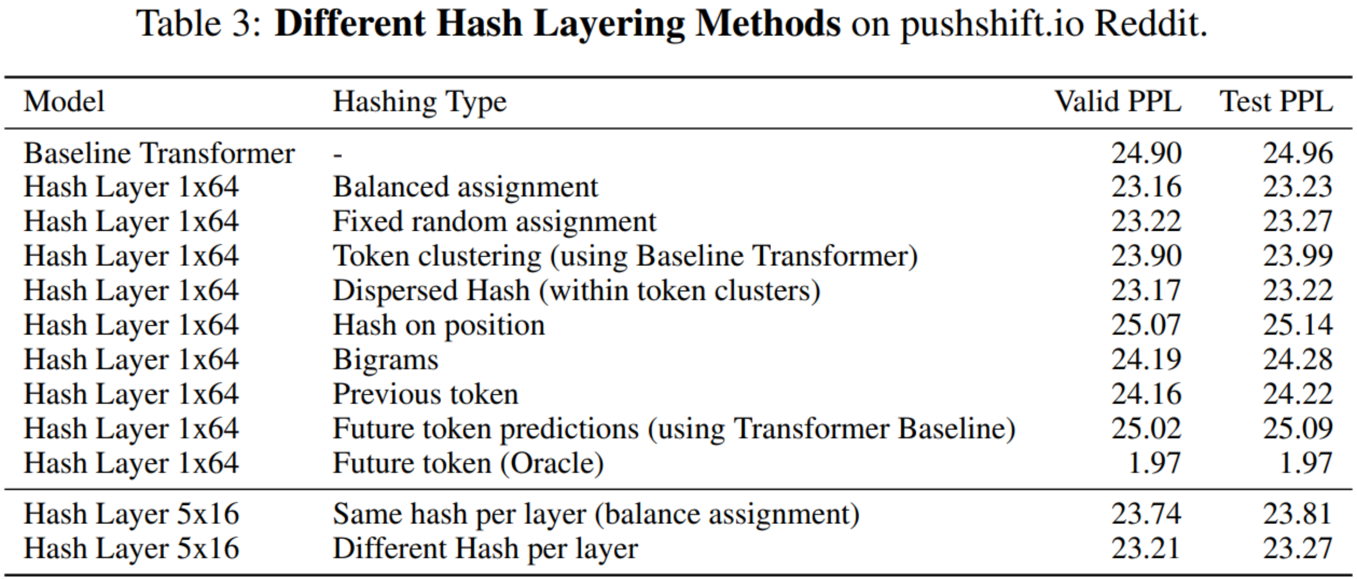


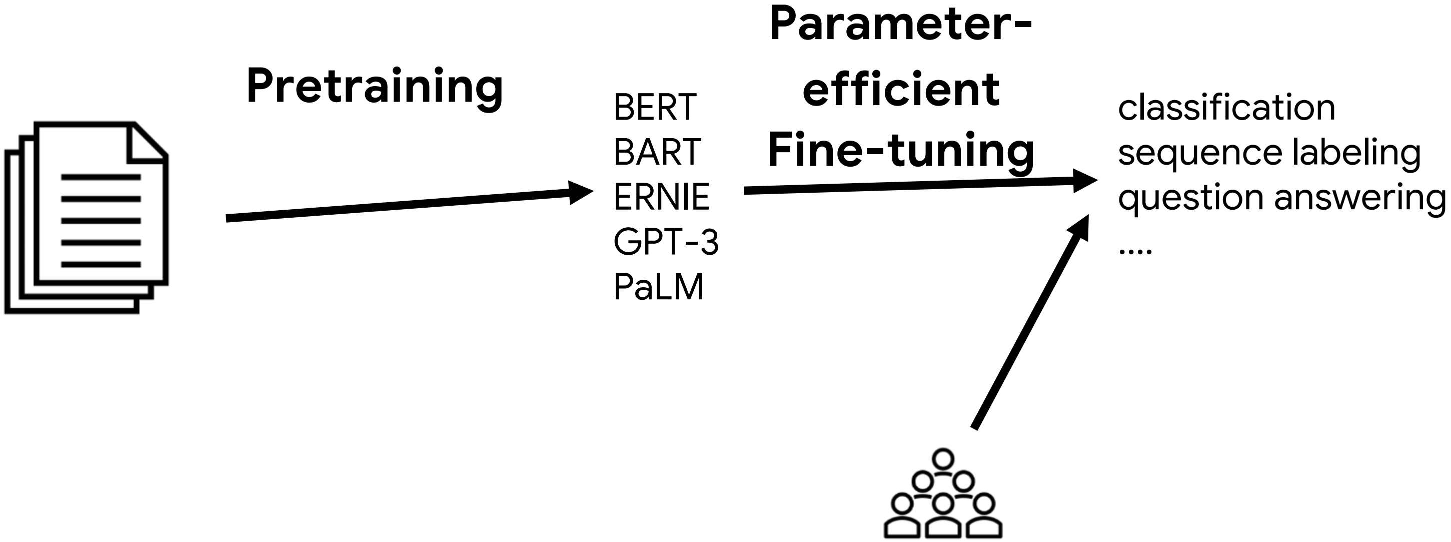
Table 3: **Different Hash Layering Methods** on pushshift.io Reddit.

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

Parameter-efficient Fine-tuning

From Fine-tuning to Parameter-efficient Fine-tuning



Pretraining

BERT
BART
ERNIE
GPT-3
PaLM

Parameter-efficient Fine-tuning

classification
sequence labeling
question answering
....

Which to implement the efficient finetuning

- **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.

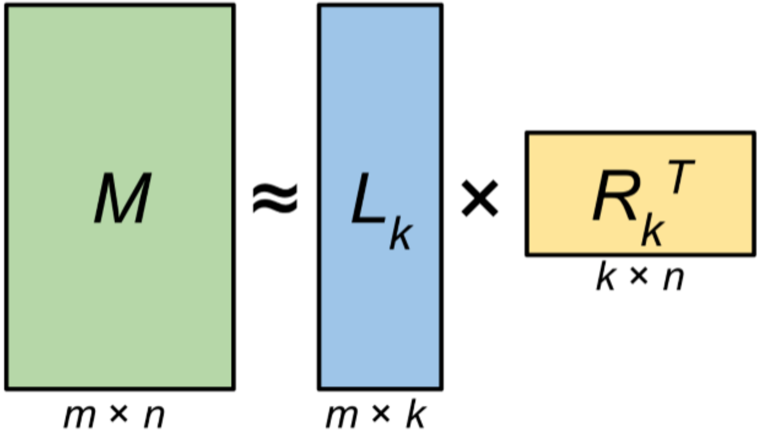
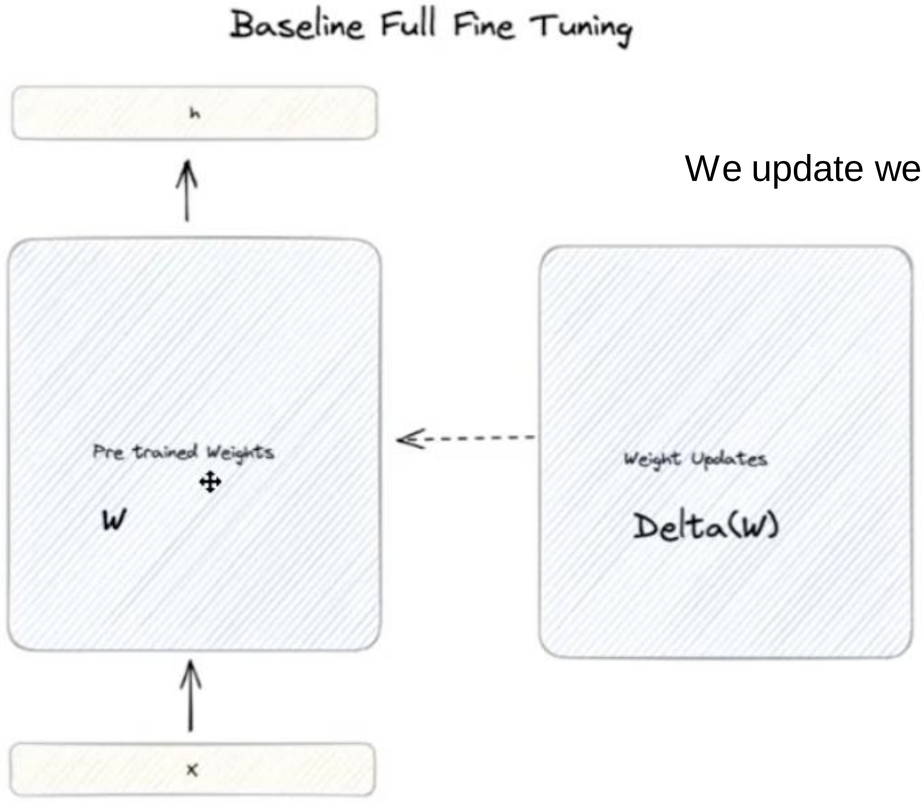


Image credit: https://dustinstansbury.github.io/theclevermachine/assets/images/svd-data-compression/low-rank-approximation.png

Baseline Full Fine-tuning

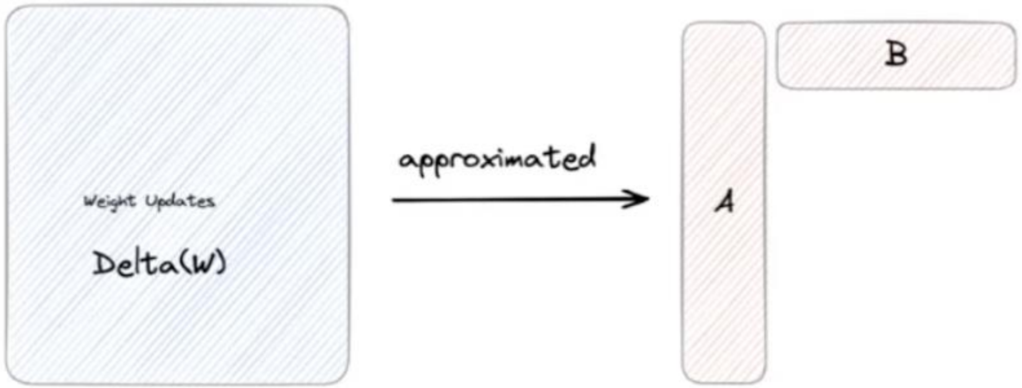


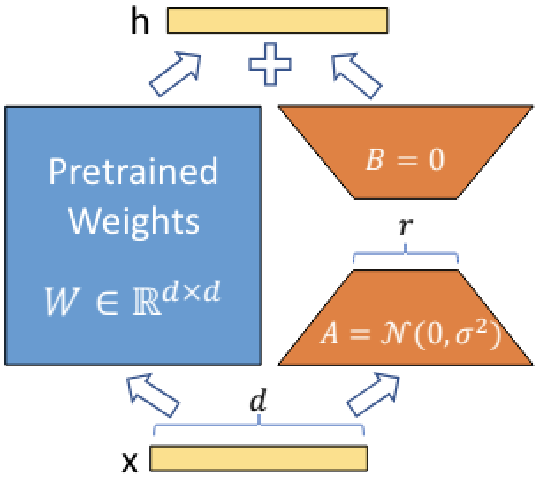
We update weights: W = W + Delta(W)

Problem

Delta(W) is huge

LoRA Tuning




h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

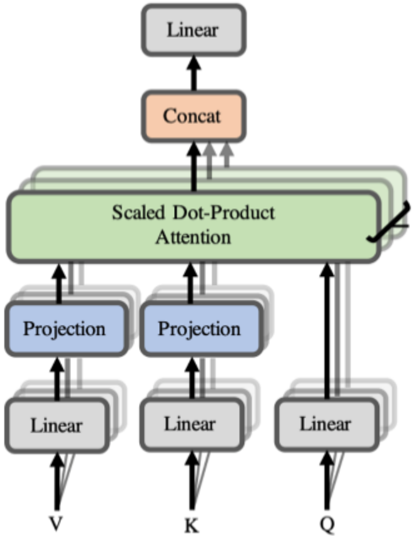
For a pre-trained weights W_0, we approx Delta(w) by B and A:

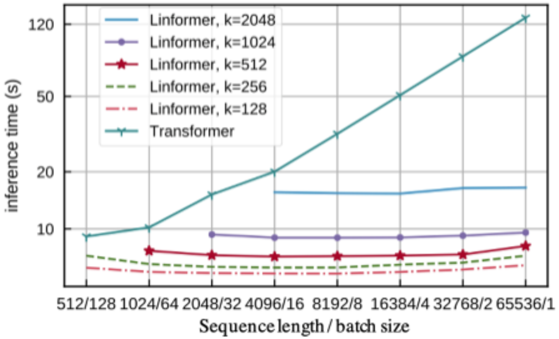
$h = W_0x + \Delta W x = W_0x + BAx$, where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and the rank $r \ll \min(d, k)$

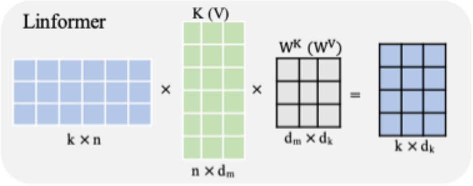
During Training, we only compute gradient w.r.t. Delta(W)

Low-Rank Approximations - Linformer

Approximate Self-Attention with Low-Rank Matrix







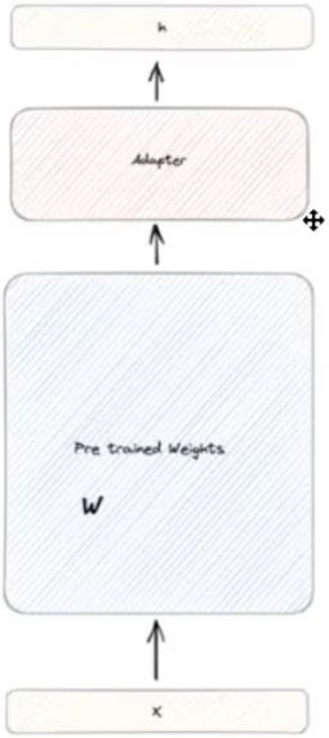
$$\overline{\text{head}_i} = \text{Attention}(QW_i^Q, \boxed{E_i}KW_i^K, \boxed{F_i}VW_i^V)$$

$$= \underbrace{\text{softmax}\left(\frac{QW_i^Q(E_iKW_i^K)^T}{\sqrt{d_k}}\right)}_{\bar{P}:n\times k} \cdot \underbrace{F_iVW_i^V}_{k\times d},$$

- It projects the **length dimension** (**not the feature dimension**) of keys and values to a lower-dimensional representation (from N to k).
- Low-rank method reduces the memory complexity problem of self-attention (from N N to N×k).

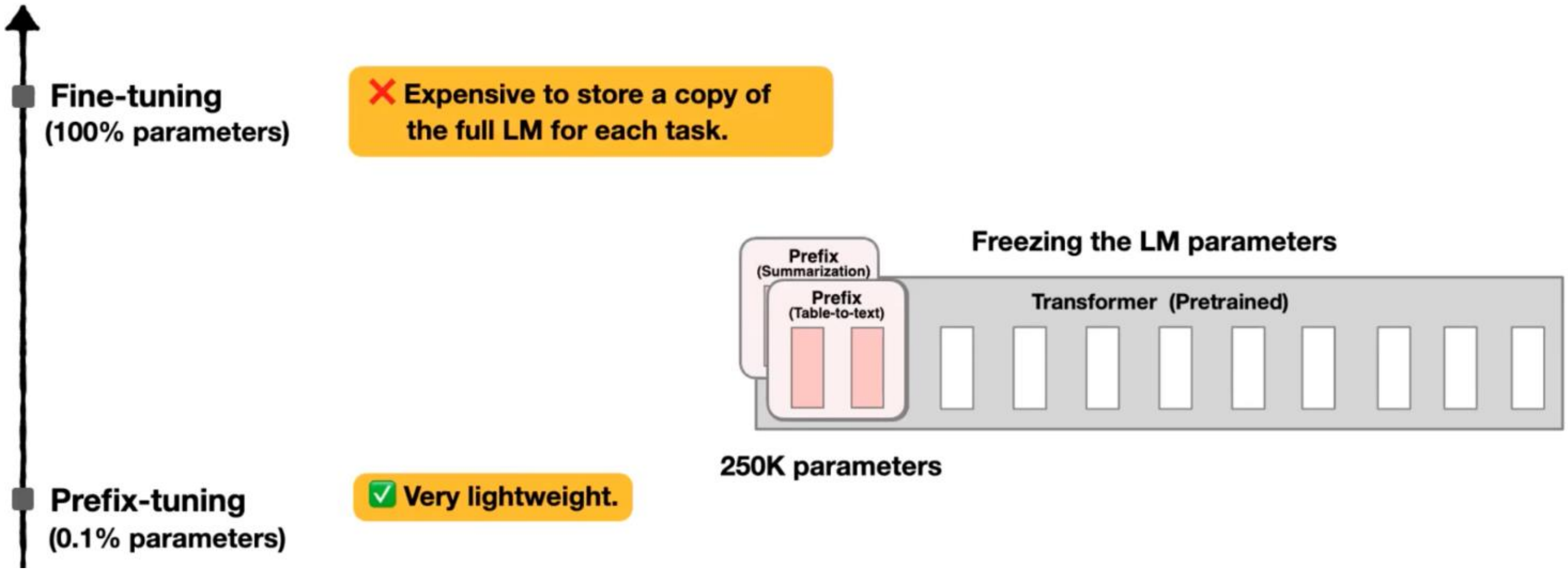
Linformer: Self-Attention with Linear Complexity [Wang *et al.*, 2020]

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: Prefix-Tuning

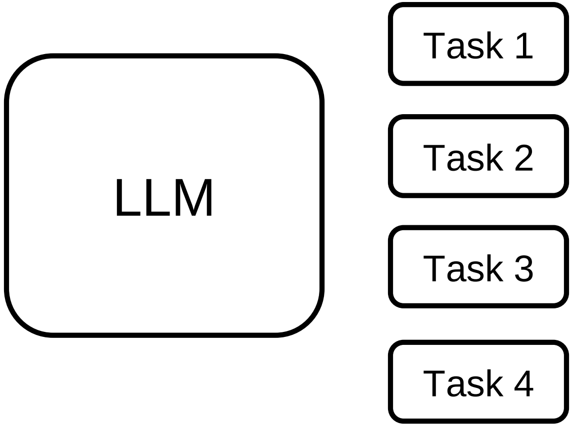


We prepend a prefix matrix in each transformer layer. And during Training, we only compute gradient w.r.t. Prefix parameters

Parameter-efficient Fine-tuning - **modulization**

Parameter-efficient Fine-tuning

parameter-efficient modules



LLM

Task 1

Task 2

Task 3

Task 4

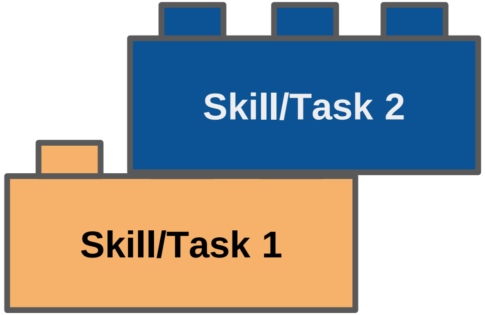
One task, one module

Modularity and Compositionality?

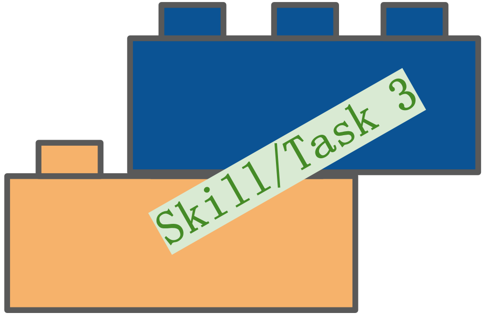
Skill/Task 1

Skill/Task 2

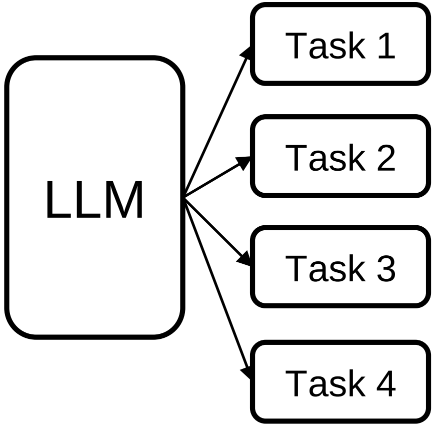
Modularity and Compositionality?



Modularity and Compositionality?



Parameter-efficient Fine-tuning



parameter-efficient modules

LLM

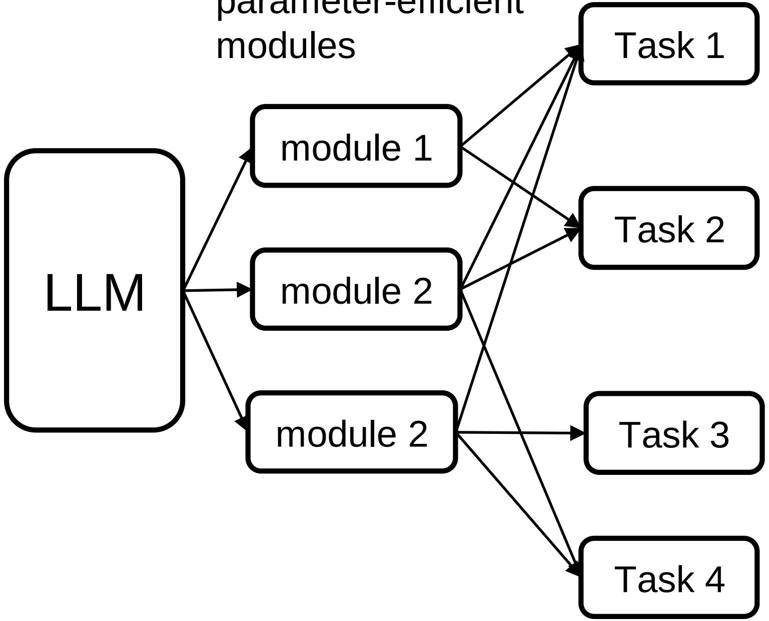
Task 1

Task 2

Task 3

Task 4

One task, one module



parameter-efficient modules

LLM

module 1

module 2

module 2

Task 1

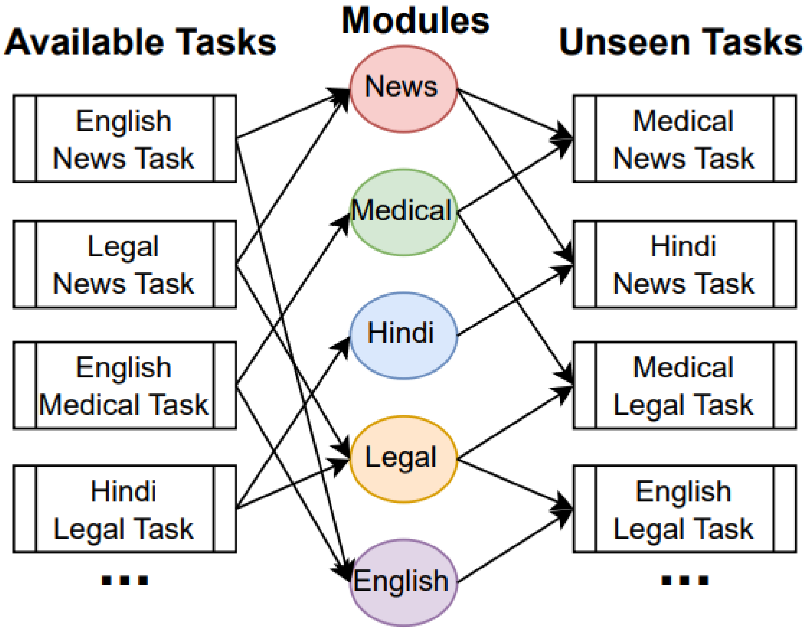
Task 2

Task 3

Task 4

One task, composed modules

Modular Retrieval

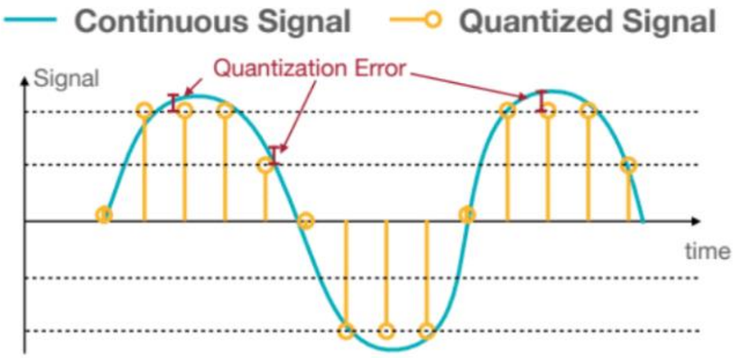


Juhao Liang , Chen Zhang , Zhengyang Tang , Jie Fu , Dawei Song , **Benyou Wang**. Modular Retrieval for Generalization and Interpretation. https://arxiv.org/pdf/2303.13419.pdf

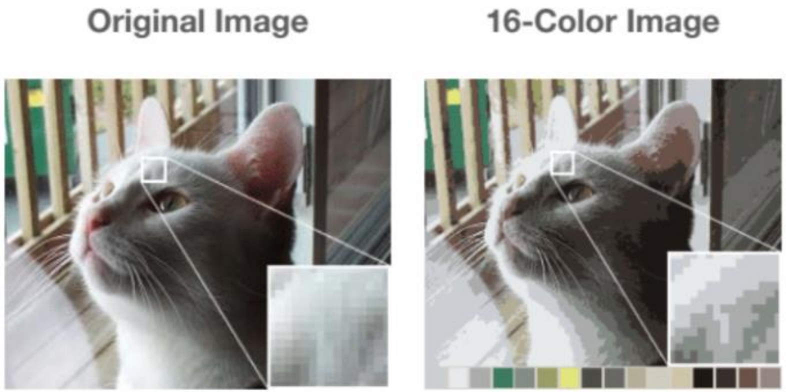
Efficiency Beyond Transformers - Quantization

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.

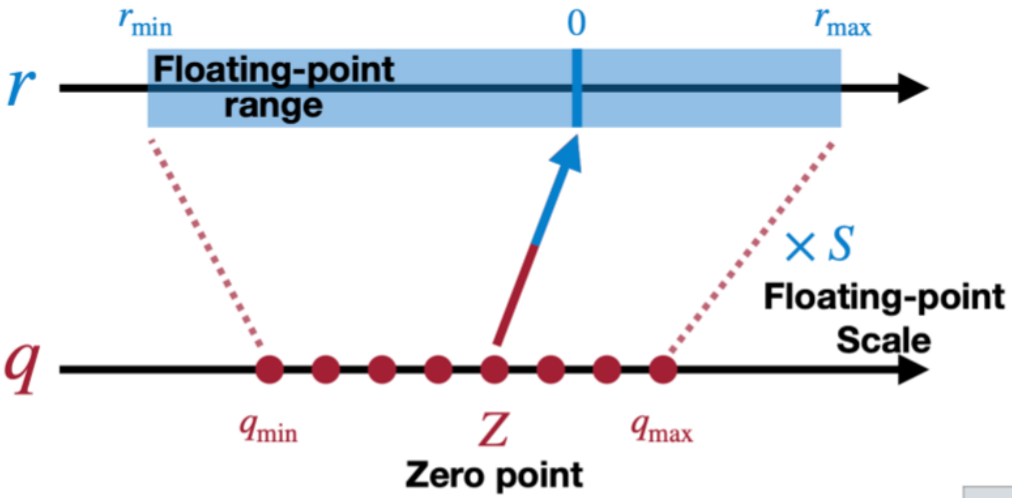


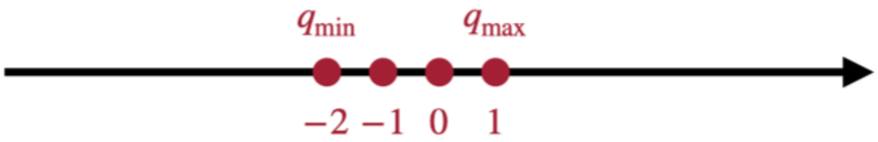
Images are in the public domain.

"Palettization"

Quantization [Wikipedia]

Quantization

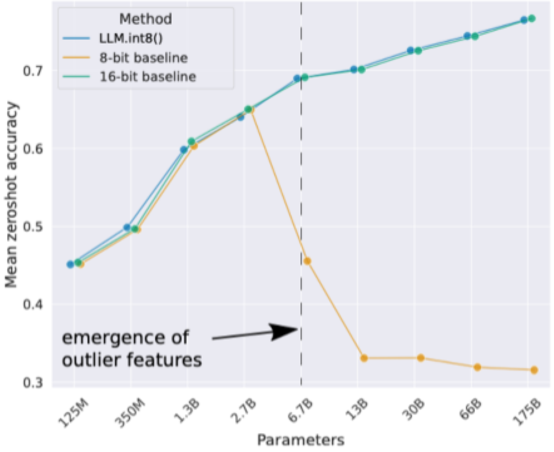


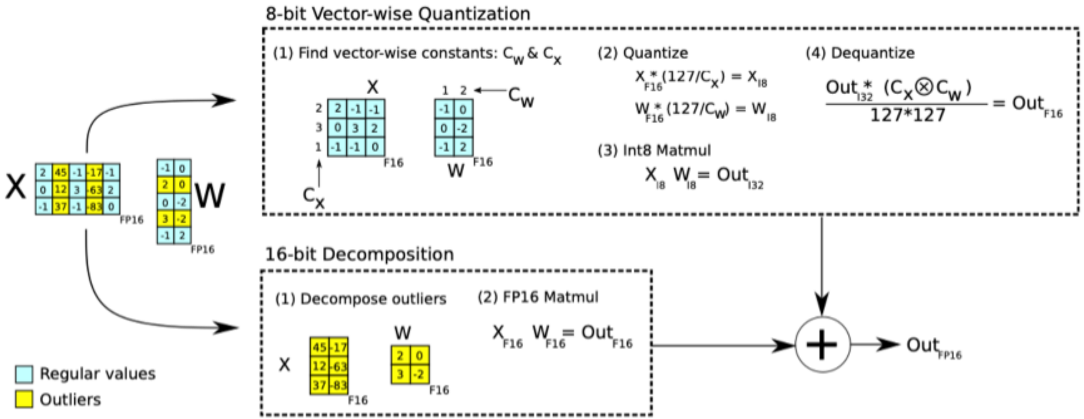


Binary	Decimal
01	1
00	0
11	-1
10	-2

Quantization - LLM.int8()

Mixed-Precision Decomposition





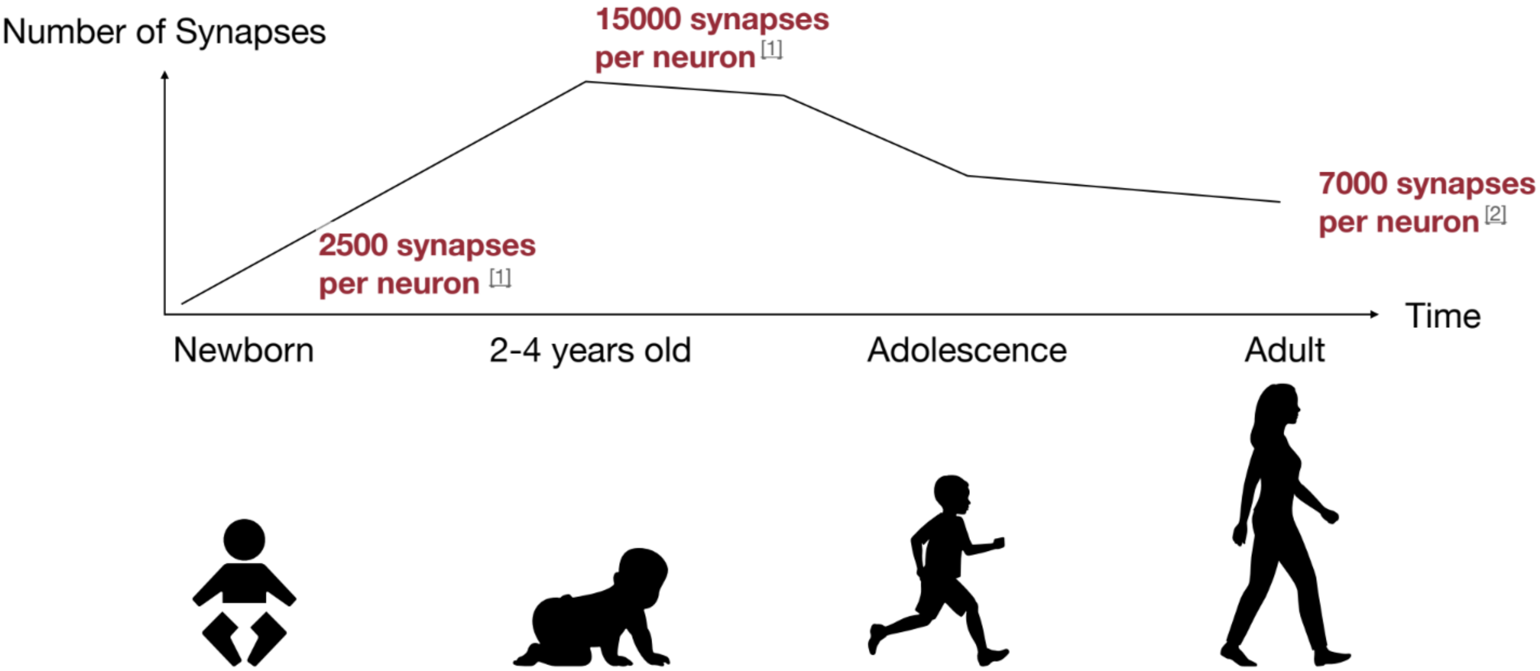
- **Motivation**: Transformers have outlier features that have **large values** (especially large models).
  - They occur in particular hidden dimensions, leading to large quantization error.
- **Key idea**: Separate outlier features into a **separate FP16 MM**, quantize the other values to Int8.
  - 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]

Efficiency Beyond Transformers - Pruning

Neural Network Pruning - What is Pruning?

Pruning happens in human brains (synapse 突触 vs. neuron神经元)

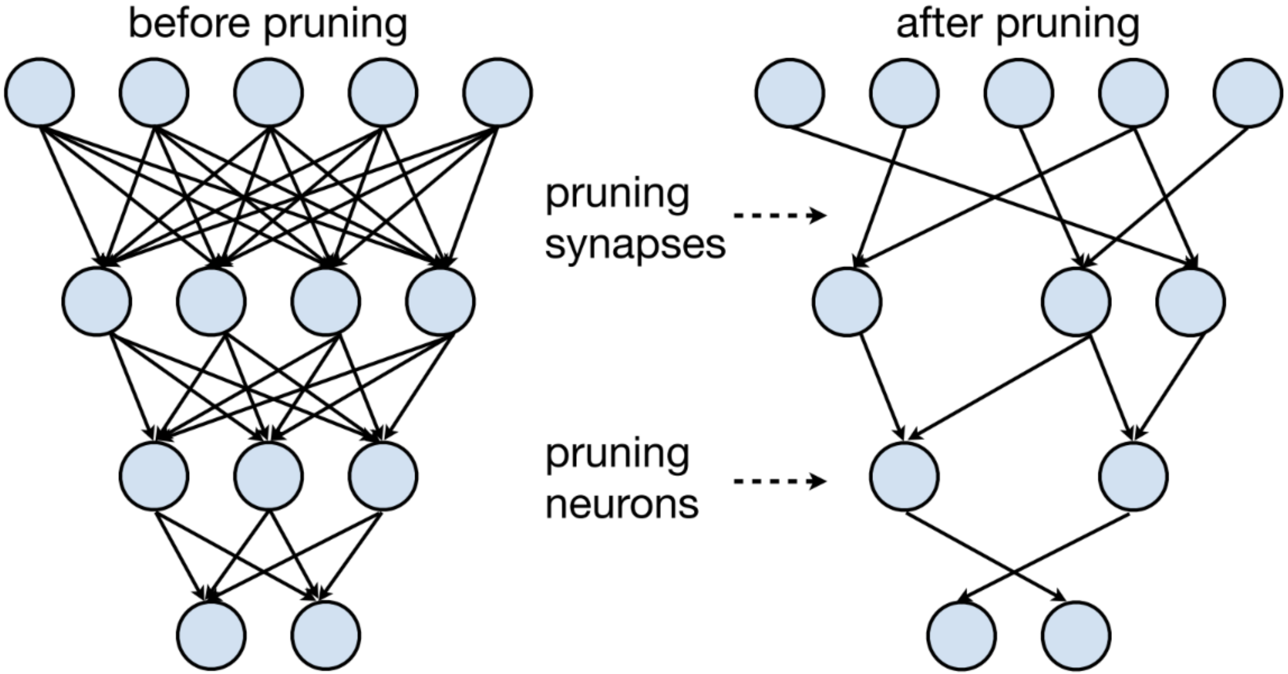


Do We Have Brain to Spare? [Drachman DA, Neurology 2004]
Peter Huttenlocher (1931–2013) [Walsh, C. A., Nature 2013]

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

Neural Network Pruning - What is Pruning?

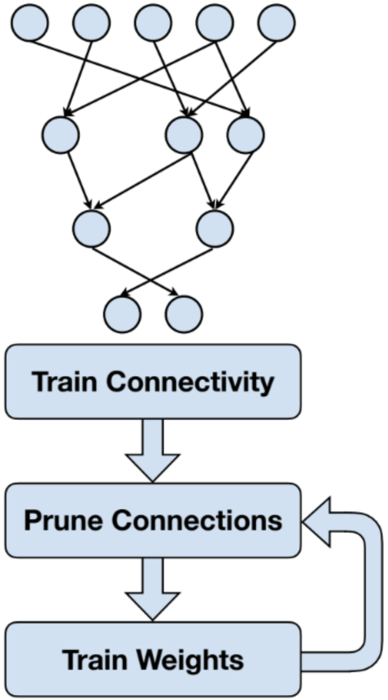
Make neural network smaller by removing synapses and neurons



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?

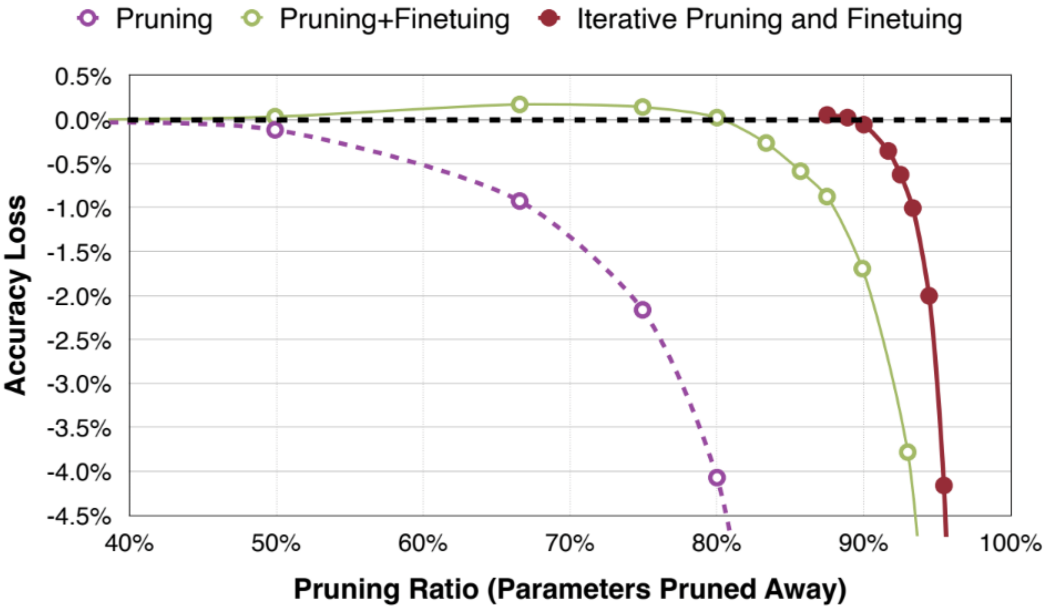
Make neural network smaller by removing synapses and neurons



Train Connectivity

Prune Connections

Train Weights



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

Neural Network Pruning - How should we formulate pruning

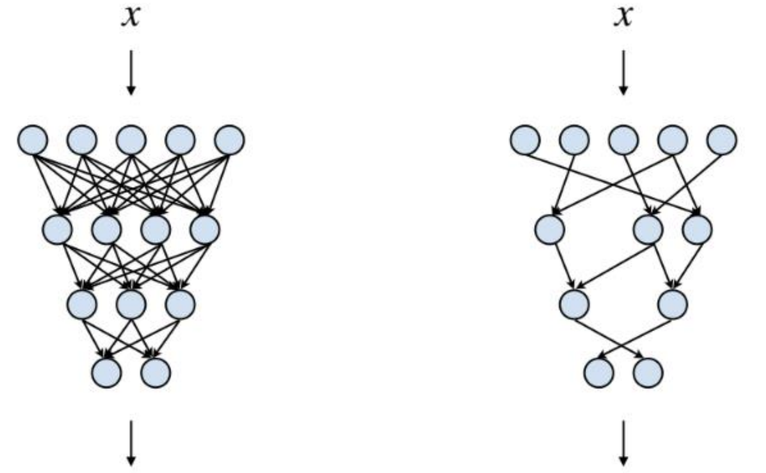
- In general, we could formulate the pruning as follows:

$$\underset{\mathbf{W}_P}{\arg\min}\, L(\mathbf{x}; \mathbf{W}_P)$$

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.

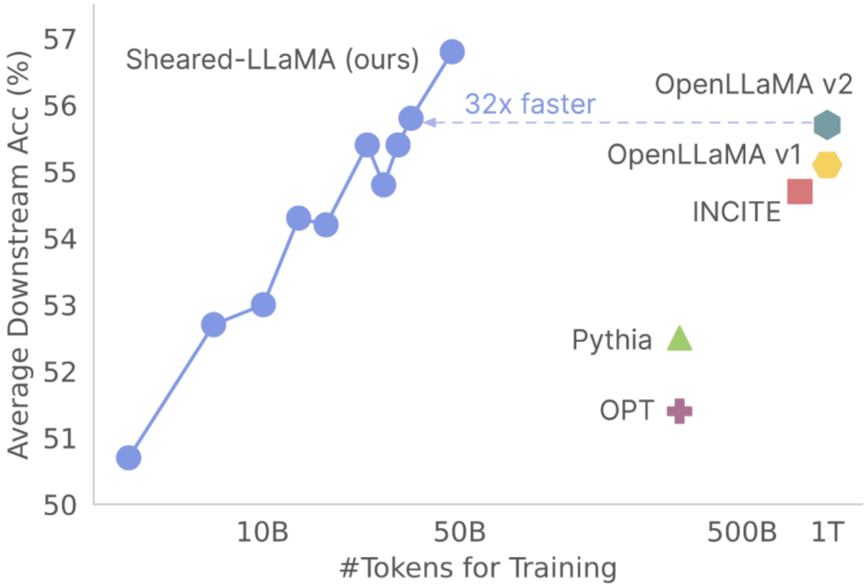


$$\underset{\mathbf{W}}{\arg\min}\, L(\mathbf{x}; \mathbf{W})$$

$$\underset{\mathbf{W}_P}{\arg\min}\, L(\mathbf{x}; \mathbf{W}_P)$$
$$s.t. \|\mathbf{W}_P\|_0 \le N$$

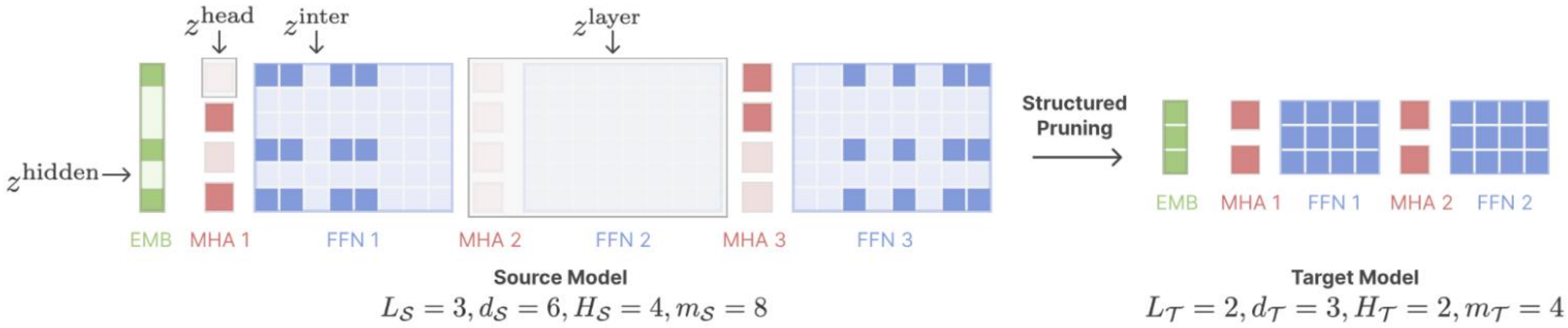
LLM-Shearing: Accelerating via Structured Pruning

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



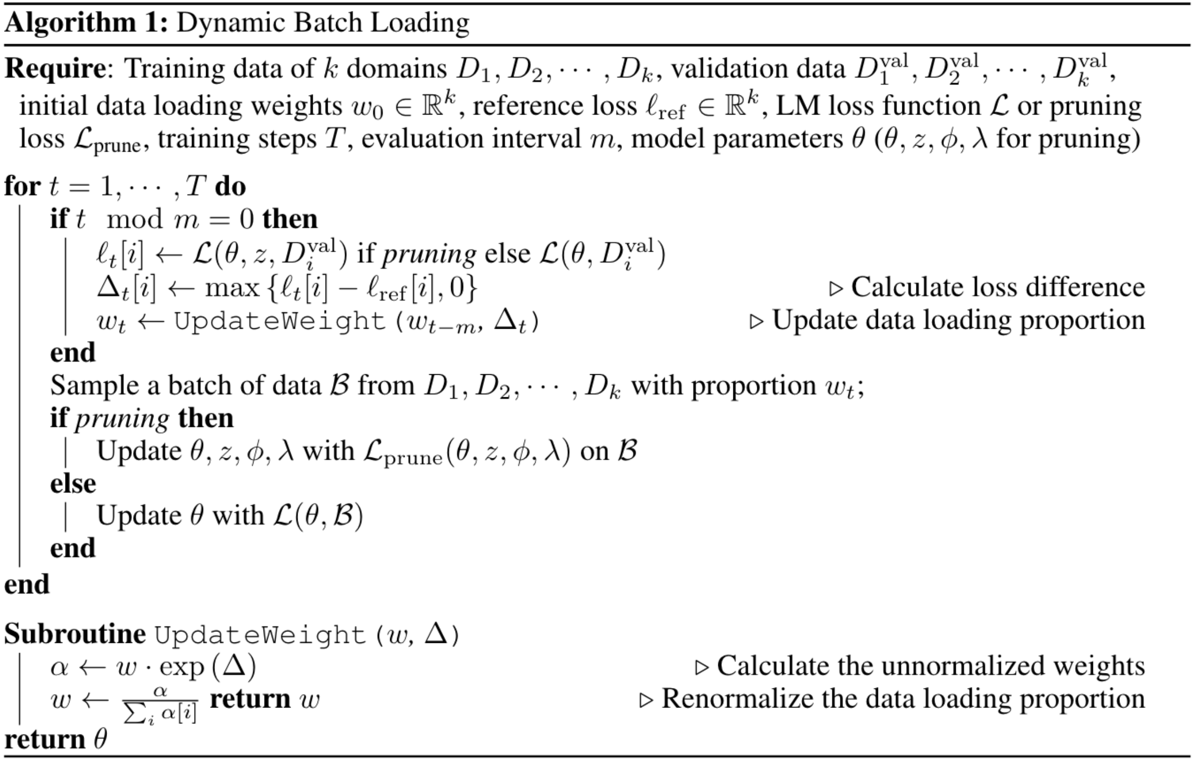
- Sheared-LLaMA-2.7B achieves better performance than existing open-source models of the same scale with 3% (1/32) of the compute.
- The trajectory shows a compelling case that if we invest more tokens and compute, the capability of Sheared-LLaMA can be further improved.

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



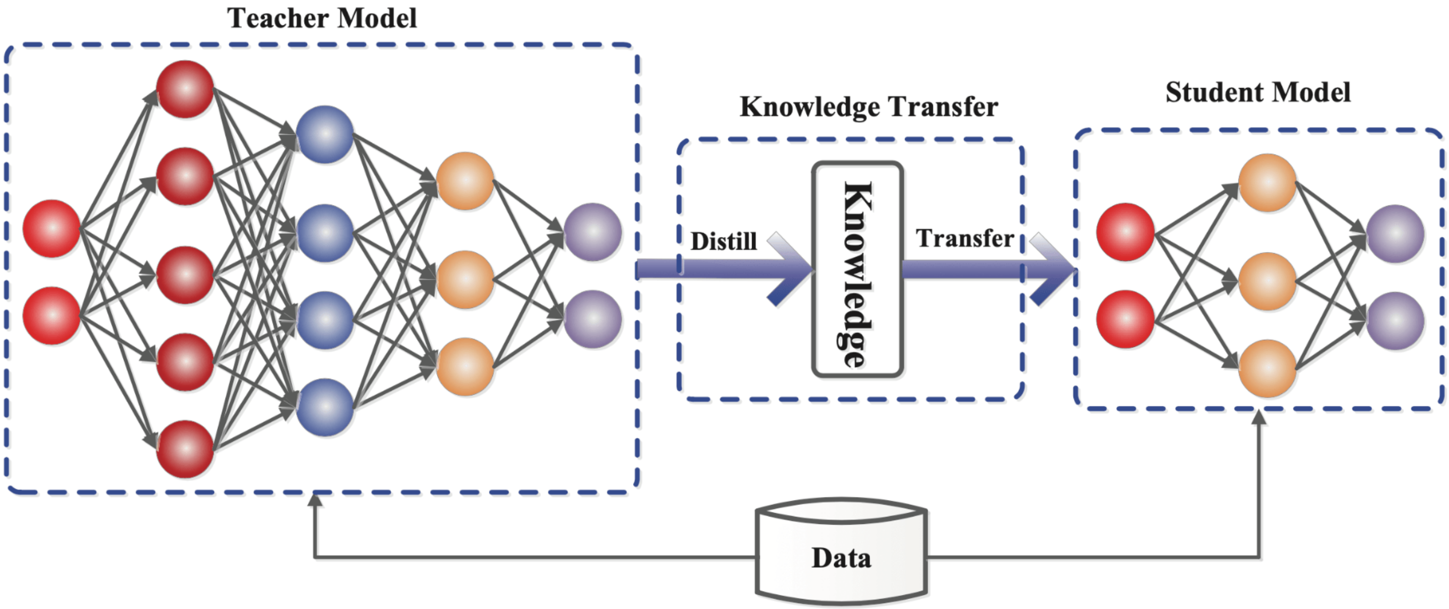
Algorithm 1: Dynamic Batch Loading

```
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 $\theta$ ($\theta, z, \phi, \lambda$ 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
        $w_t \leftarrow$ 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 $\theta, z, \phi, \lambda$ with $\mathcal{L}_{\text{prune}}(\theta, z, \phi, \lambda)$ on $\mathcal{B}$
    else
        Update $\theta$ with $\mathcal{L}(\theta, \mathcal{B})$
    end
end
Subroutine UpdateWeight($w$, $\Delta$)
    $\alpha \leftarrow w \cdot \exp(\Delta)$        ▷ Calculate the unnormalized weights
    $w \leftarrow \frac{\alpha}{\sum_i \alpha[i]}$ return $w$        ▷ Renormalize the data loading proportion
return $\theta$
```

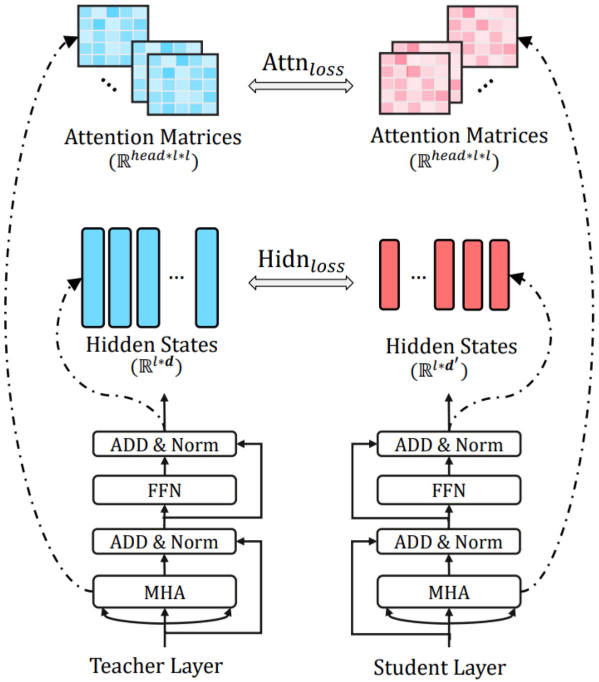
2. Dynamic batch loading: Pruning results in varying information retainment across domains. Concretely, they load more data for domains that recover slow, and the loading proportion is dynamically decided on the fly.

Efficiency Beyond Transformers - Distillation

Framework of knowledge distillation



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 on every layers)
- Making use of unannotated data (teacher model provide supervision)
  - Data augmentation is useful.

A few work of KD in LLMs

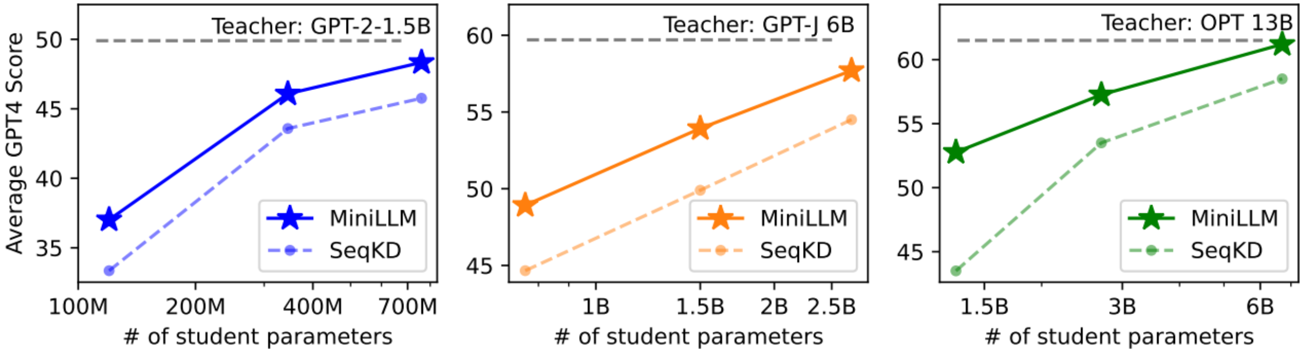


Figure 1: The comparison of MiniLLM with the sequence-level KD (SeqKD) in terms of the average GPT-4 feedback score on our evaluation sets. **Left**: GPT-2-1.5B as the teacher and GPT-2 125M, 340M, 760M as the students. **Middle**: GPT-J 6B as the teacher and GPT-2 760M, 1.5B, GPT-*Neo* 2.7B as the students.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

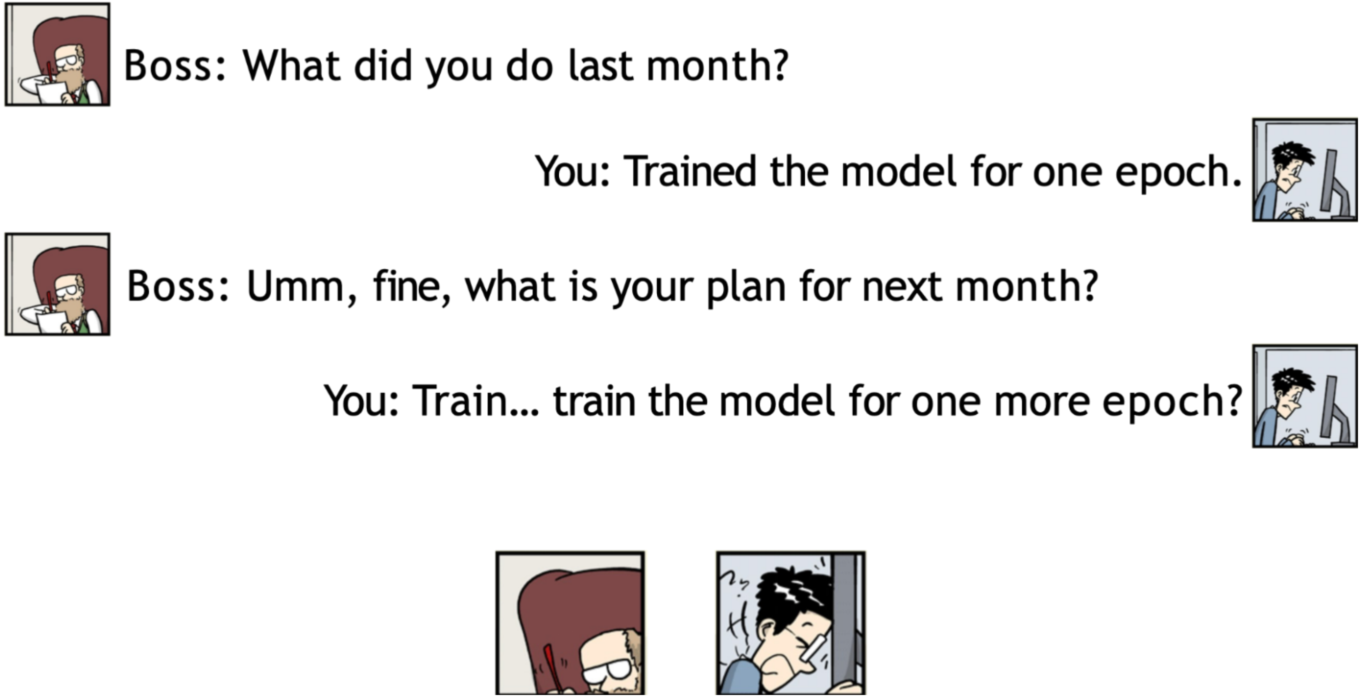
This seems not that working in LLMs. More investigation is needed

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

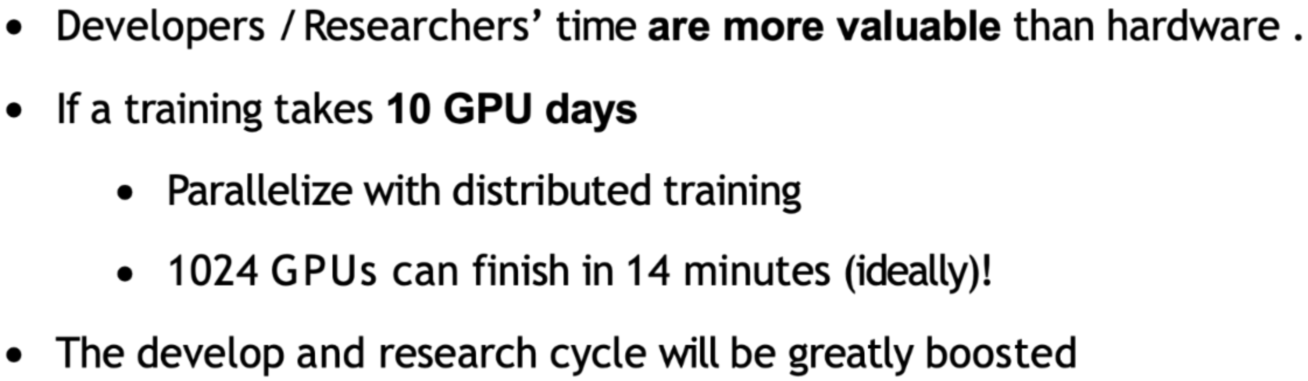
Memory-efficiency training

Models are getting larger and larger

LLMs take much longer time to train!

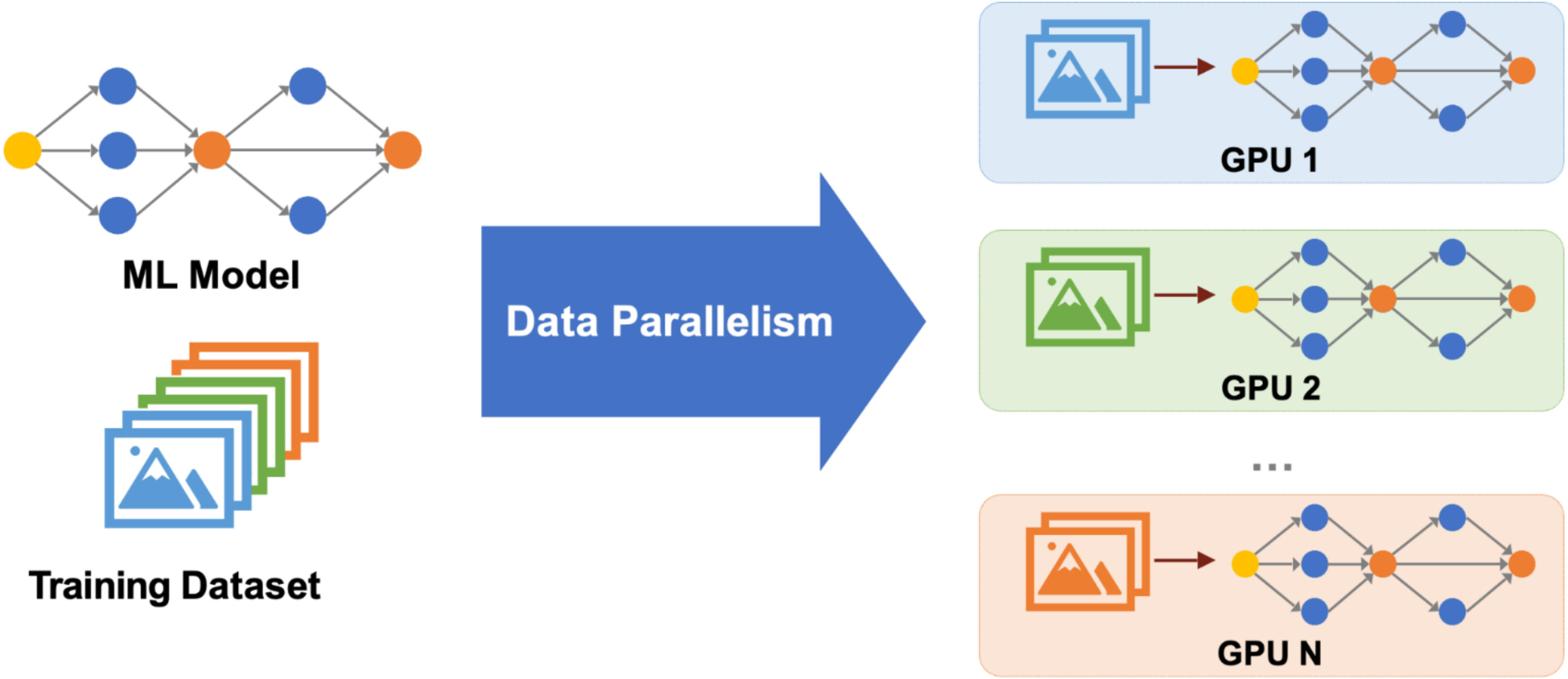

Boss: What did you do last month?
You: Trained the model for one epoch.
Boss: Umm, fine, what is your plan for next month?
You: Train... train the model for one more epoch?

Distributed Training is almost Necessary for every LLMs!

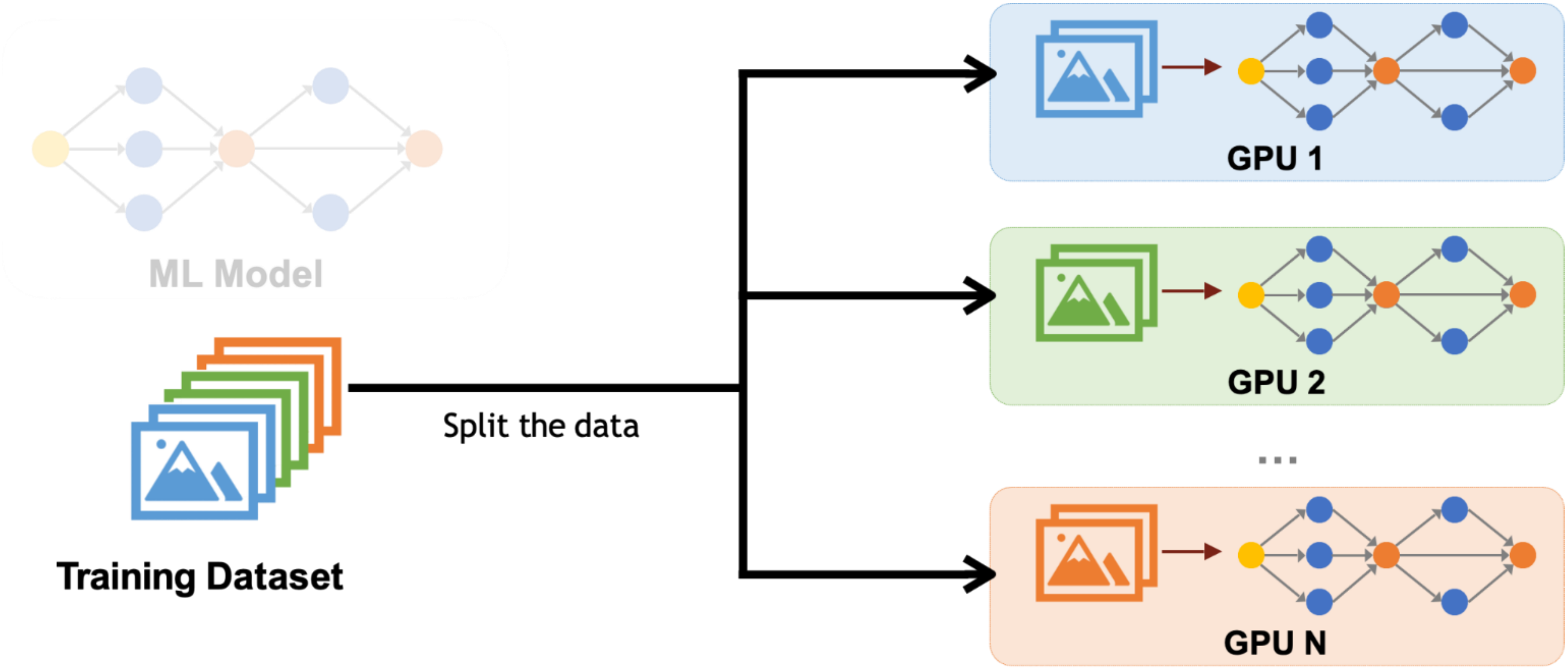


- Developers / Researchers’ time **are more valuable** than hardware .
- If a training takes **10 GPU days**
    - Parallelize with distributed training
    - 1024 GPUs can finish in 14 minutes (ideally)!
- The develop and research cycle will be greatly boosted

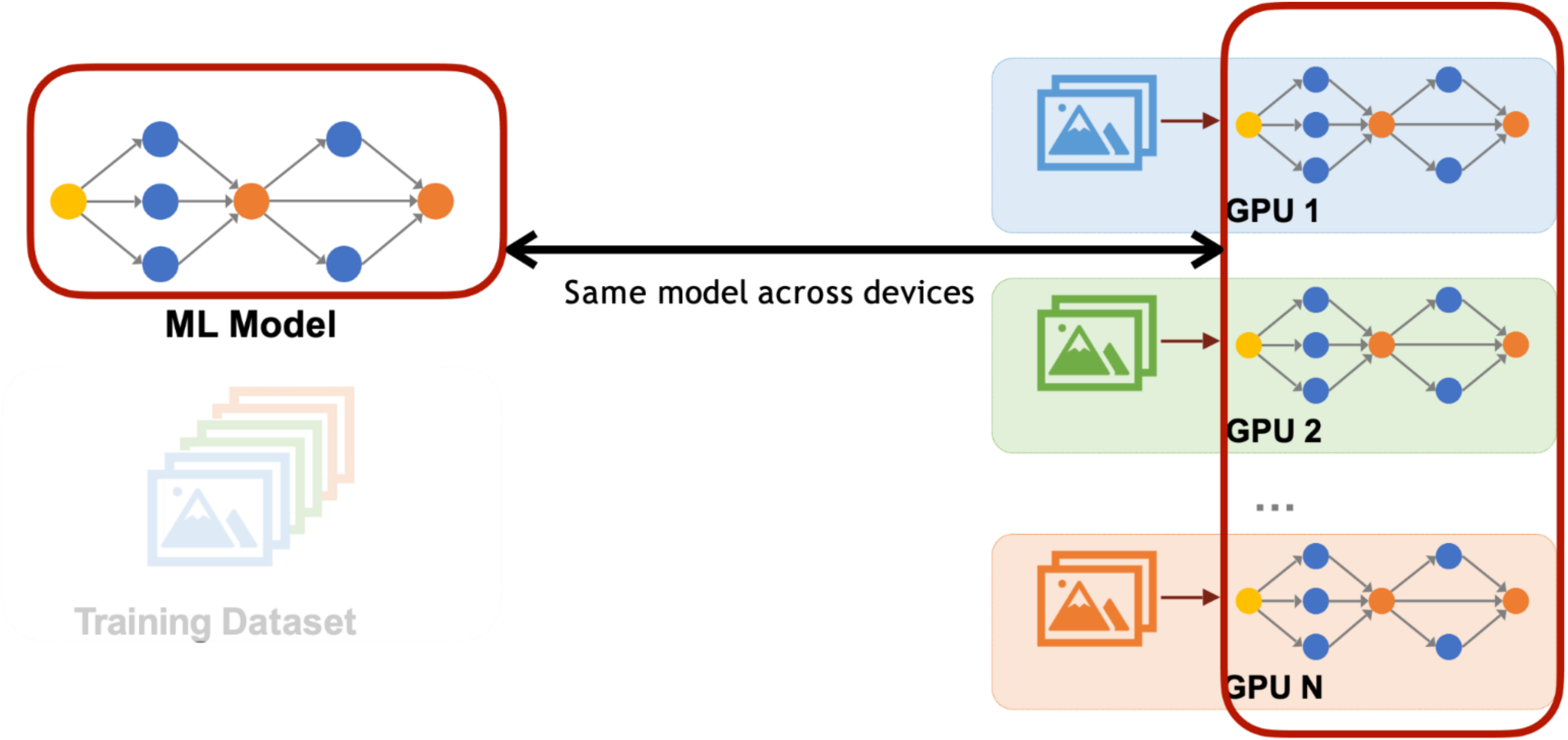
Parallelism in Distributed Training - Data Parallelism



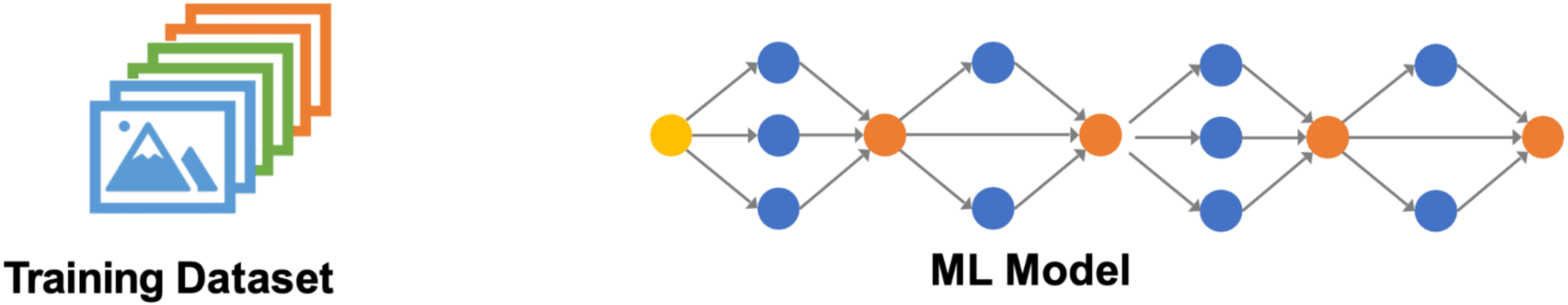
Parallelism in Distributed Training - Data Parallelism

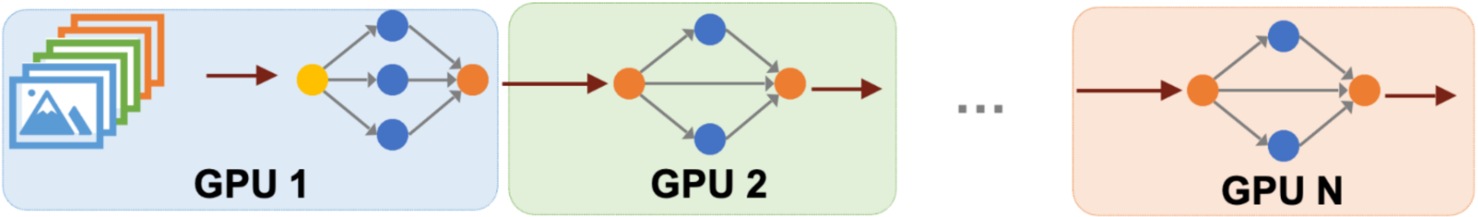


Parallelism in Distributed Training - Data Parallelism

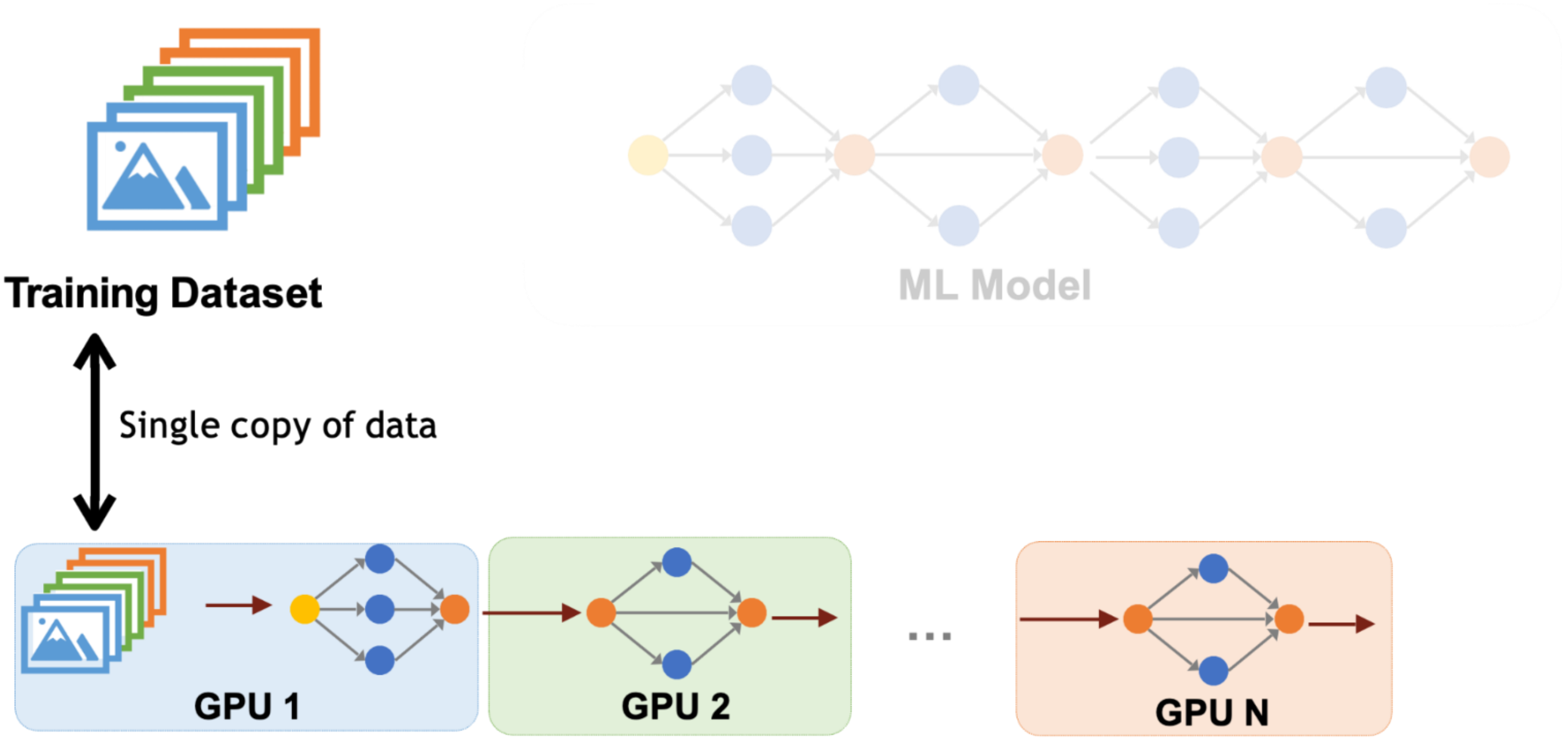


Parallelism in Distributed Training - Model Parallelism





Parallelism in Distributed Training - Model Parallelism



Training Dataset

Single copy of data

ML Model

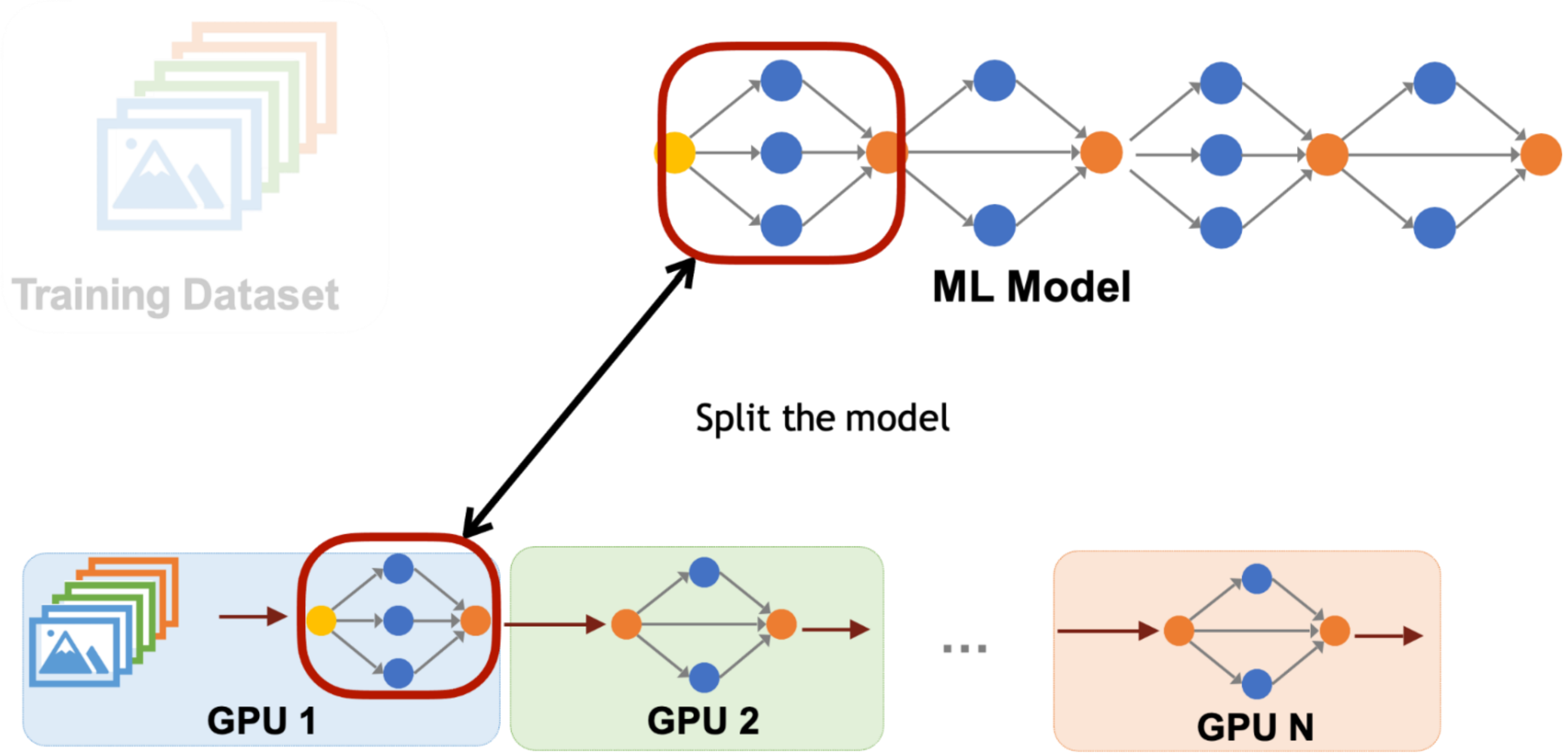
GPU 1

GPU 2

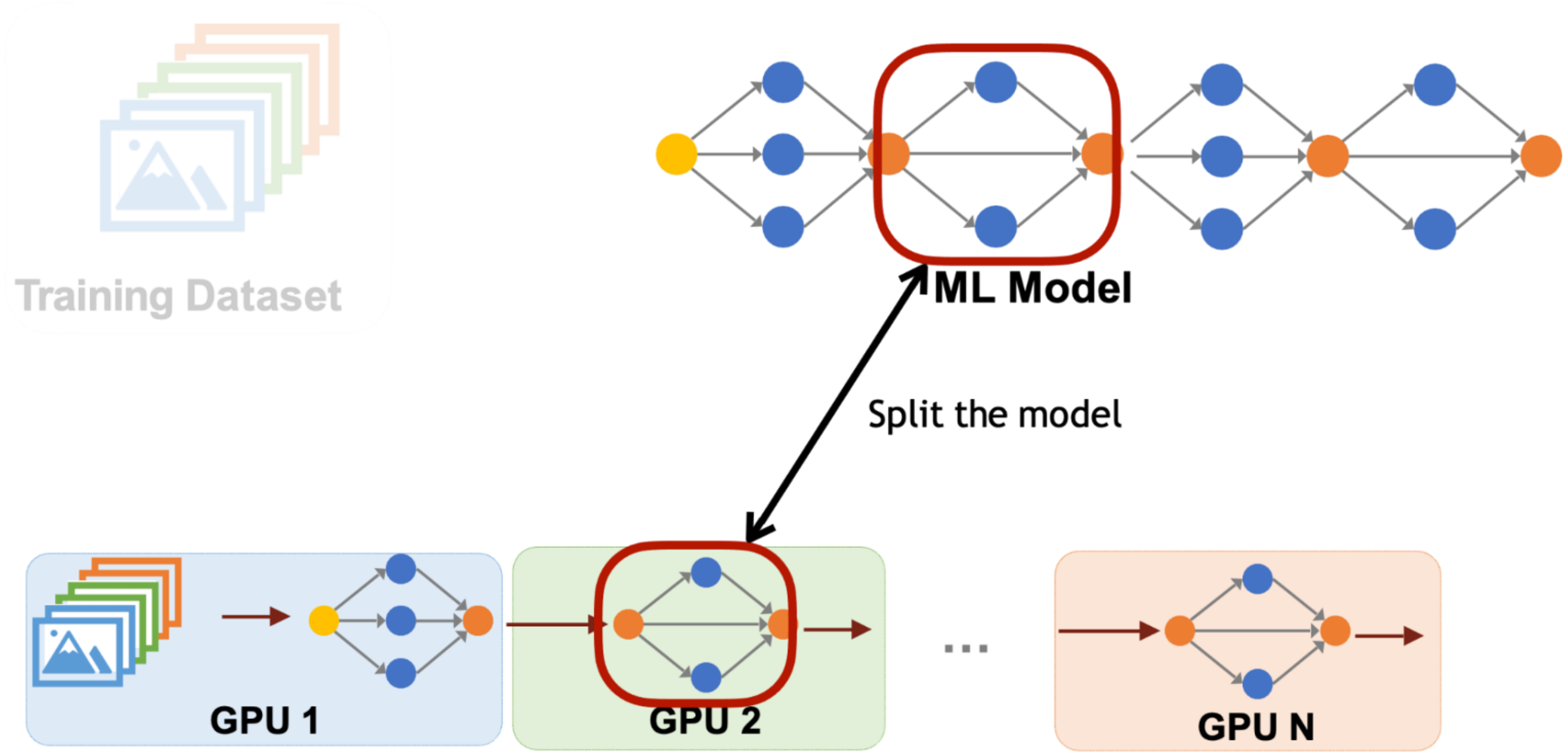
...

GPU N

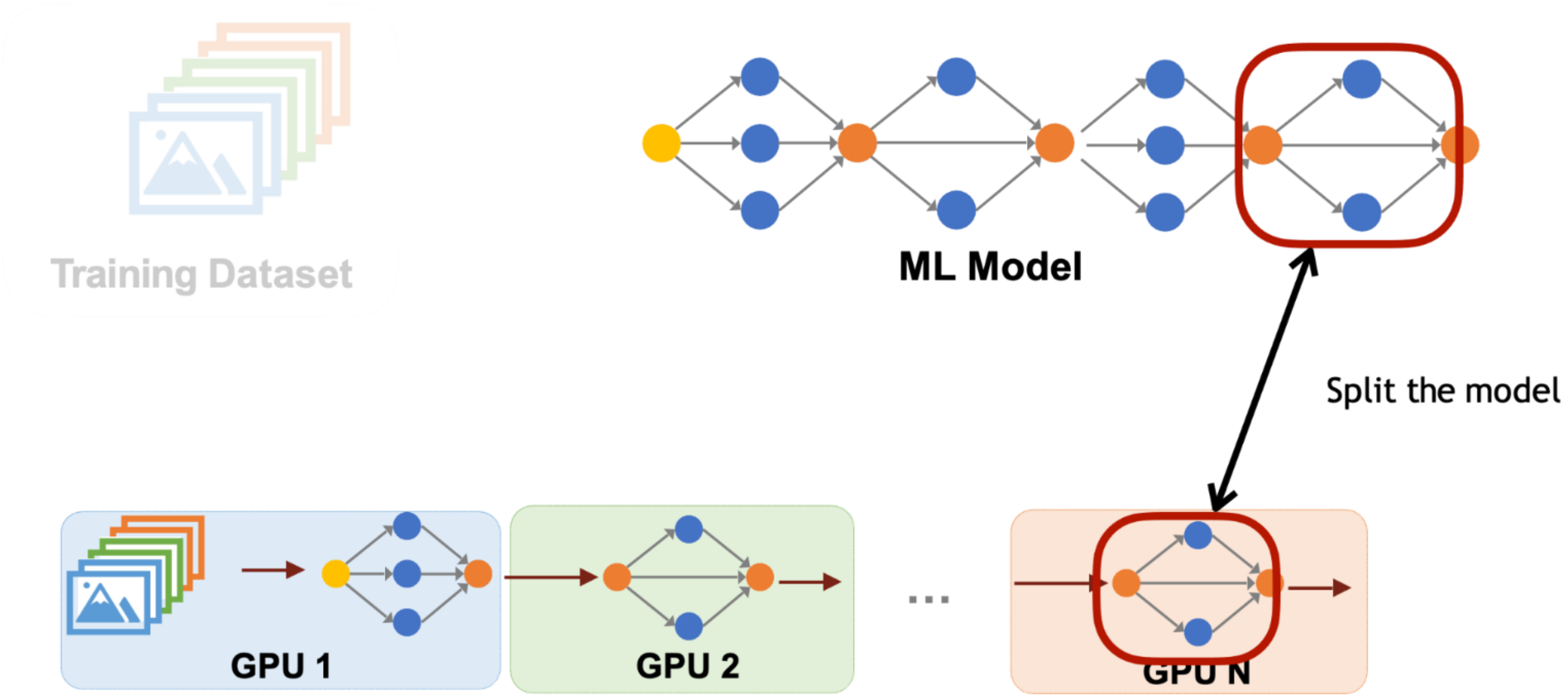
Parallelism in Distributed Training - Model Parallelism



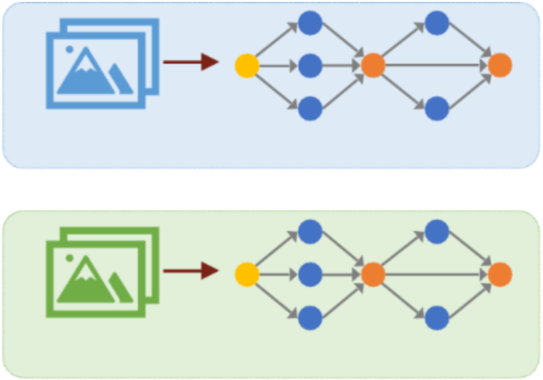
Parallelism in Distributed Training - Model Parallelism

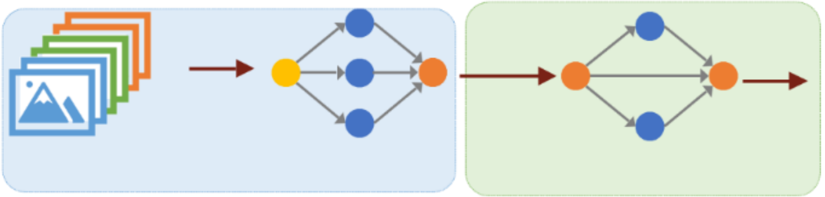


Parallelism in Distributed Training - Model Parallelism



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 and Memory Optimizations - ZeRO: Train Trillion-scale models

Let’s take a step back for training a singler layer in practice

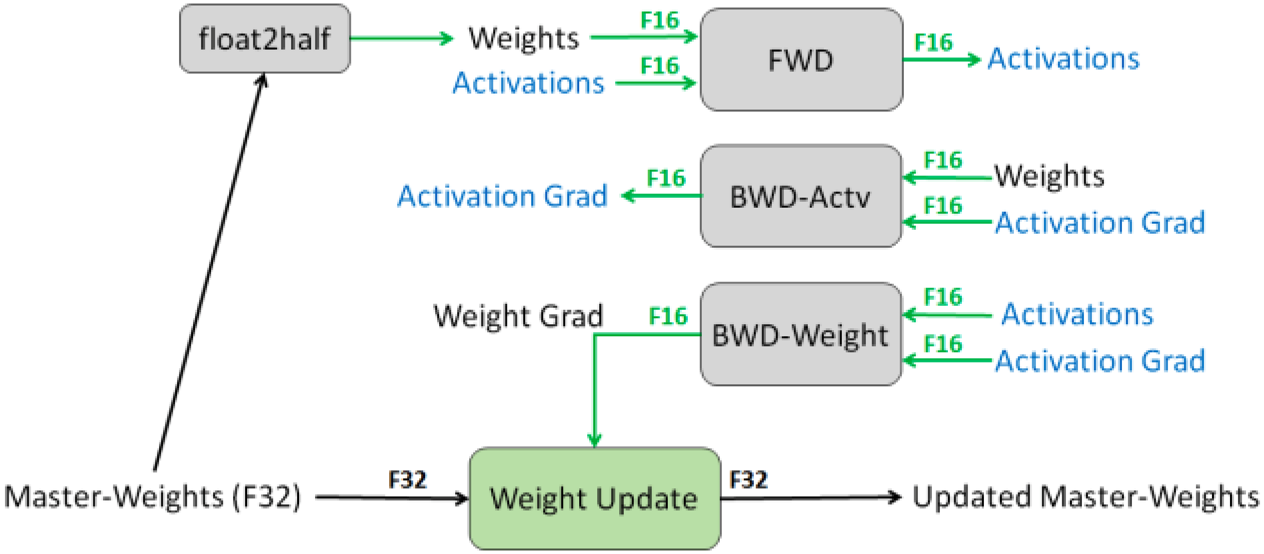


Figure 1: Mixed precision training iteration for a layer.

Memory Consumptions for this example:

Suppose the layer (or model) is trained using Adam Optimizer.
The number of parameters are $\mathbf{\Phi}$.

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

- Model parameters (fp16): $2\mathbf{\Phi}$
- Model gradients (fp16): $2\mathbf{\Phi}$
- Adam Optimizer states - copy of Parameters, Momentum and Variance (fp32): $4\mathbf{\Phi} + 4\mathbf{\Phi} + 4\mathbf{\Phi} = 12\mathbf{\Phi}$
- 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!

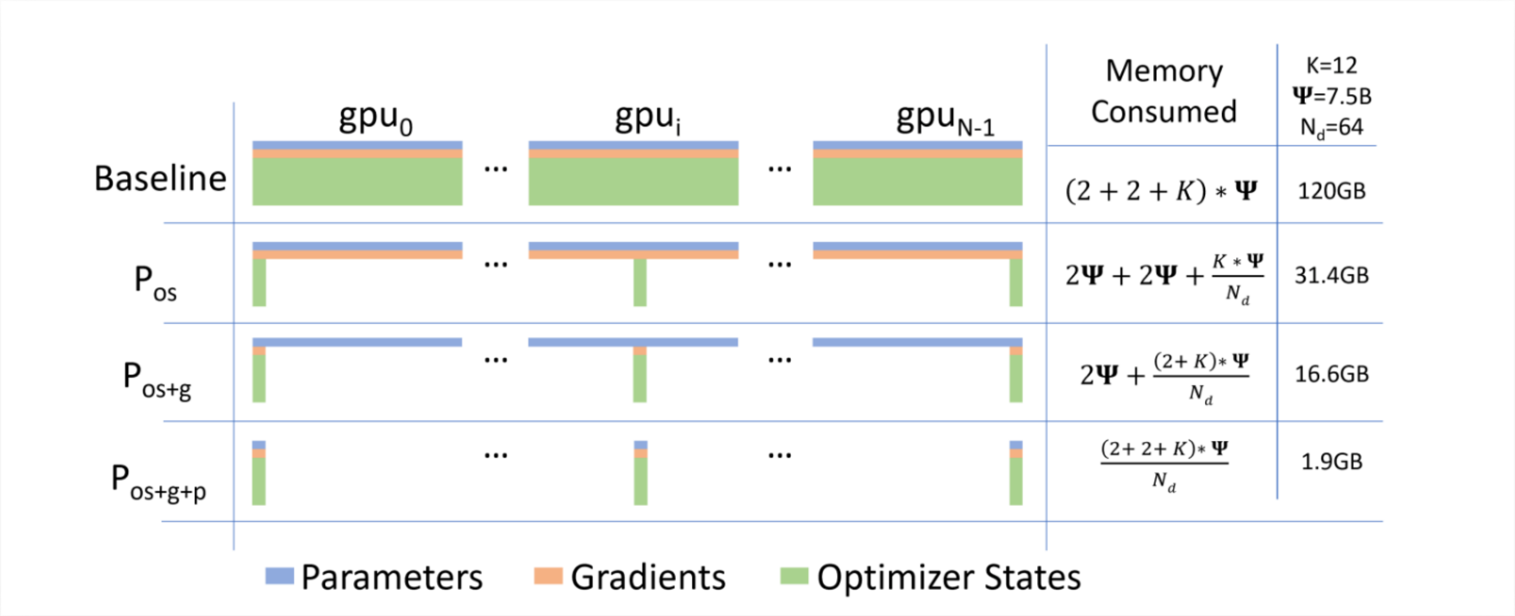
VRam Estimation

Model: HuatuoGPT-7B

1. Model
    a. Param(fp16): 7B*2=14GB
    b. Grad(fp16):7B*2=14GB
2. **Optimizer(AdamW)**
    a. Master Weights(fp32): 7B*4=28GB
    b. Adam m(fp32): 7B*4=28GB
    c. Adam v(fp32): 7B*4=28GB
3. Activation
4. Buffer&Fragmentation

Parallel Strategy: ZeRO

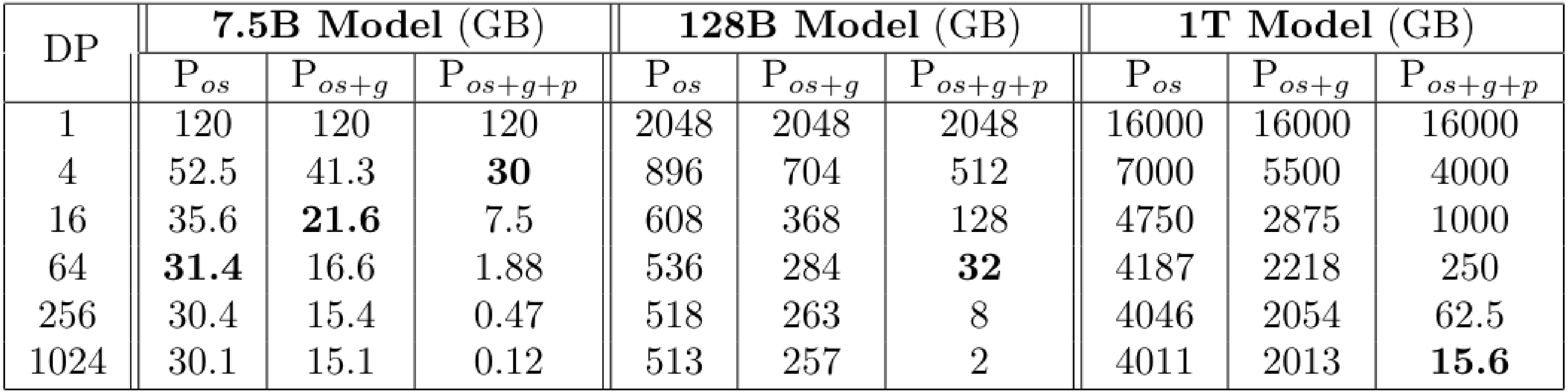
1. ZeRO-DP: Shard the optimizer state
2. ZeRO-1&2: Same communication volume as DP
3. ZeRO-3: 1.5 communication volume as DP



	Memory Consumed	K=12 Ψ=7.5B N_d=64
Baseline	$(2+2+K)*\Psi$	120GB
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB

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

ZeRO: the More GPUs, the Less Memory Consumption!



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

Future

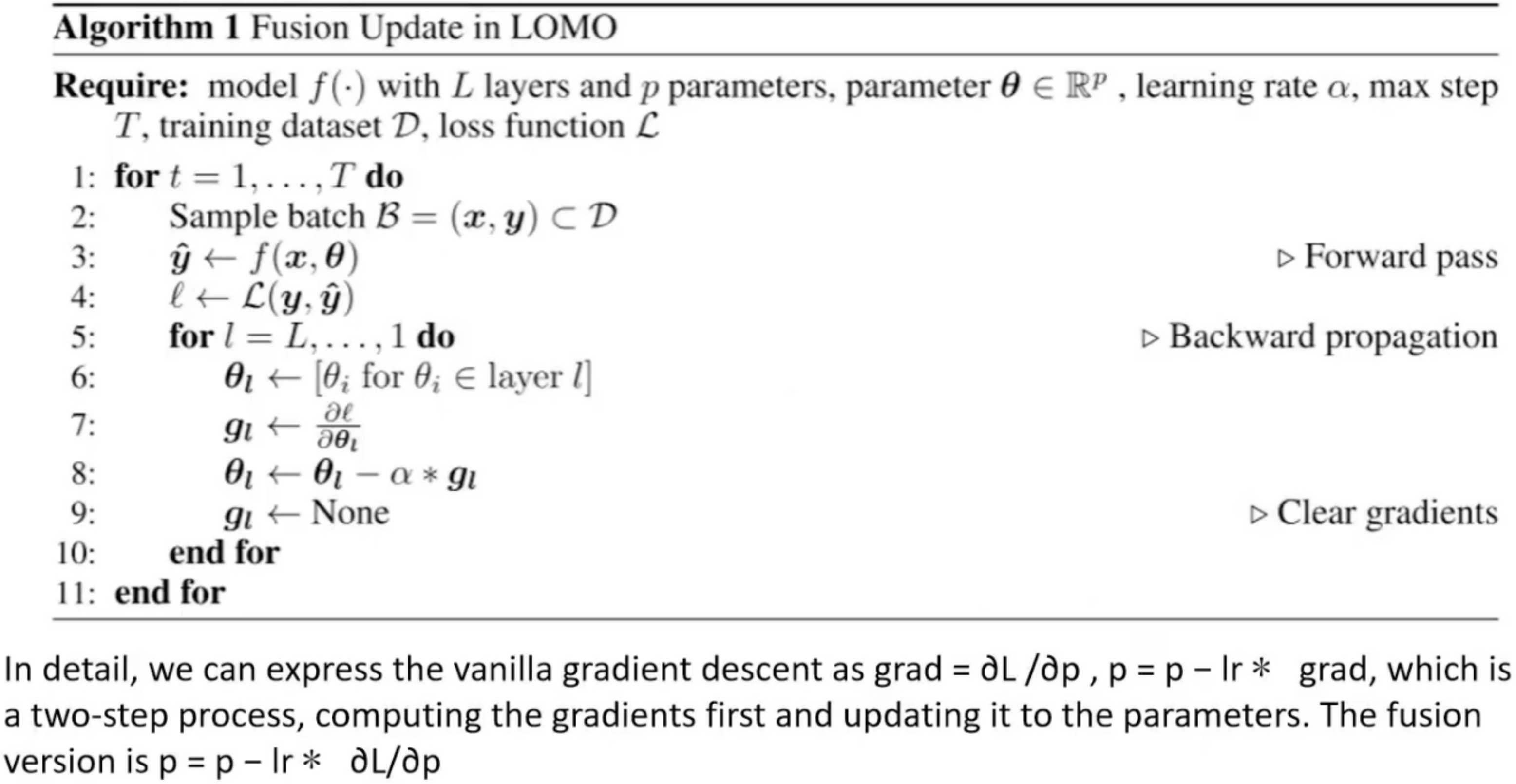
- Efficiency Long context for LLMs
- Hybrid efficiency
  - QLoRA
  - LongLoRA
  - QMOE
- MOE and modularization

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
- Shanghai AI Lab: 大型语言模型的技术原理

Distributed Training and Memory Optimizations - LOMO: Fuse gradient computation and gradient update!

LOMO: Compute Gradient and Update Parameter in SGD



Algorithm 1 Fusion Update in LOMO

```
Require: model $f(\cdot)$ with $L$ layers and $p$ parameters, parameter $\boldsymbol{\theta} \in \mathbb{R}^p$, learning rate $\alpha$, max step $T$, training dataset $\mathcal{D}$, loss function $\mathcal{L}$
for $t = 1, \ldots, T$ do
    Sample batch $\mathcal{B} = (\boldsymbol{x}, \boldsymbol{y}) \subset \mathcal{D}$
    $\hat{\boldsymbol{y}} \leftarrow f(\boldsymbol{x}, \boldsymbol{\theta})$                ▷ Forward pass
    $\ell \leftarrow \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$
    for $l = L, \ldots, 1$ do                        ▷ Backward propagation
        $\boldsymbol{\theta}_l \leftarrow [\theta_i \text{ for } \theta_i \in \text{layer } l]$
        $\boldsymbol{g}_l \leftarrow \frac{\partial \ell}{\partial \boldsymbol{\theta}_l}$
        $\boldsymbol{\theta}_l \leftarrow \boldsymbol{\theta}_l - \alpha * \boldsymbol{g}_l$
        $\boldsymbol{g}_l \leftarrow$ None                      ▷ Clear gradients
    end for
end for
```

In detail, we can express the vanilla gradient descent as grad = ∂L /∂p , p = p − lr ∗ grad, which is a two-step process, computing the gradients first and updating it to the parameters. The fusion version is p = p − lr ∗ ∂L/∂p

LOMO: Computation Graph and Space Complexity

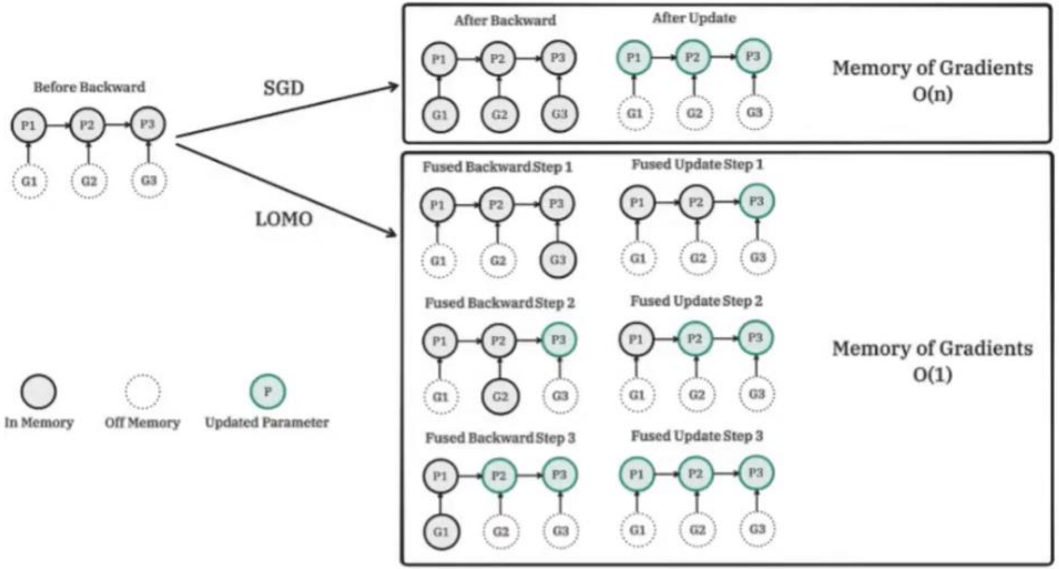
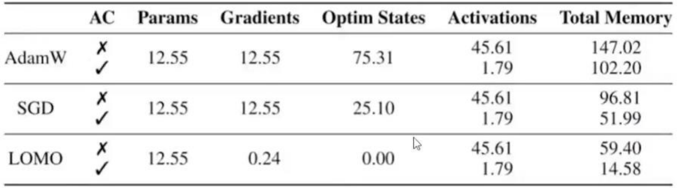


Figure : Comparison of SGD and LOMO in backpropagation and parameter update stages.
Pi refers to the parameter of the model and **Gi** refers to the gradient corresponding to **Pi**.
LOMO fused gradient computation and parameter update in one step to minimize the size of gradient tensors.

LOMO: Main results

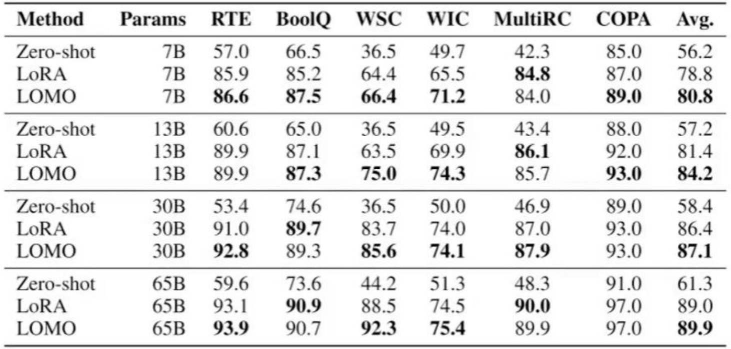
- **Memory Profile**



	AC	Params	Gradients	Optim States	Activations	Total Memory
AdamW	✗	12.55	12.55	75.31	45.61	147.02
	✓				1.79	102.20
SGD	✗	12.55	12.55	25.10	45.61	96.81
	✓				1.79	51.99
LOMO	✗	12.55	0.24	0.00	45.61	59.40
	✓				1.79	14.58

Table : Memory usage (GB) when training LLaMA-7B under different settings. AC refers toActivation Checkpointing.

- **Downstream performance**



Method	Params	RTE	BoolQ	WSC	WIC	MultiRC	COPA	Avg.
Zero-shot	7B	57.0	66.5	36.5	49.7	42.3	85.0	56.2
LoRA	7B	85.9	85.2	64.4	65.5	**84.8**	87.0	78.8
LOMO	7B	**86.6**	**87.5**	**66.4**	**71.2**	84.0	**89.0**	**80.8**
Zero-shot	13B	60.6	65.0	36.5	49.5	43.4	88.0	57.2
LoRA	13B	89.9	87.1	63.5	69.9	**86.1**	92.0	81.4
LOMO	13B	89.9	**87.3**	**75.0**	**74.3**	85.7	**93.0**	**84.2**
Zero-shot	30B	53.4	74.6	36.5	50.0	46.9	89.0	58.4
LoRA	30B	91.0	**89.7**	83.7	74.0	87.0	93.0	86.4
LOMO	30B	**92.8**	89.3	**85.6**	**74.1**	**87.9**	93.0	**87.1**
Zero-shot	65B	59.6	73.6	44.2	51.3	48.3	91.0	61.3
LoRA	65B	93.1	**90.9**	88.5	74.5	**90.0**	97.0	89.0
LOMO	65B	**93.9**	90.7	**92.3**	**75.4**	89.9	97.0	**89.9**

Table 3: Main results on SuperGLUE using LLaMA at all sizes (with 1,000 training examples).

QLoRA: Combine Quantization and Low-Rank Approximations

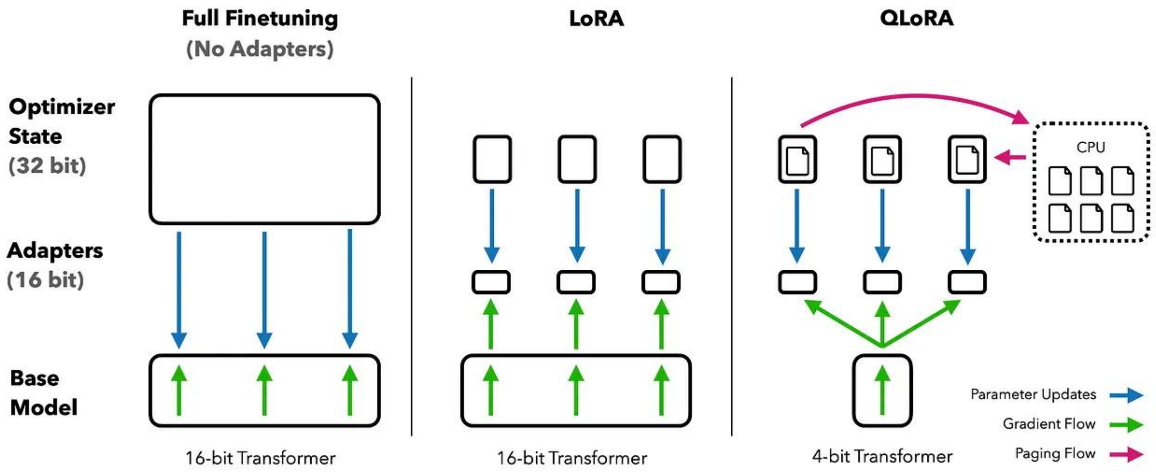
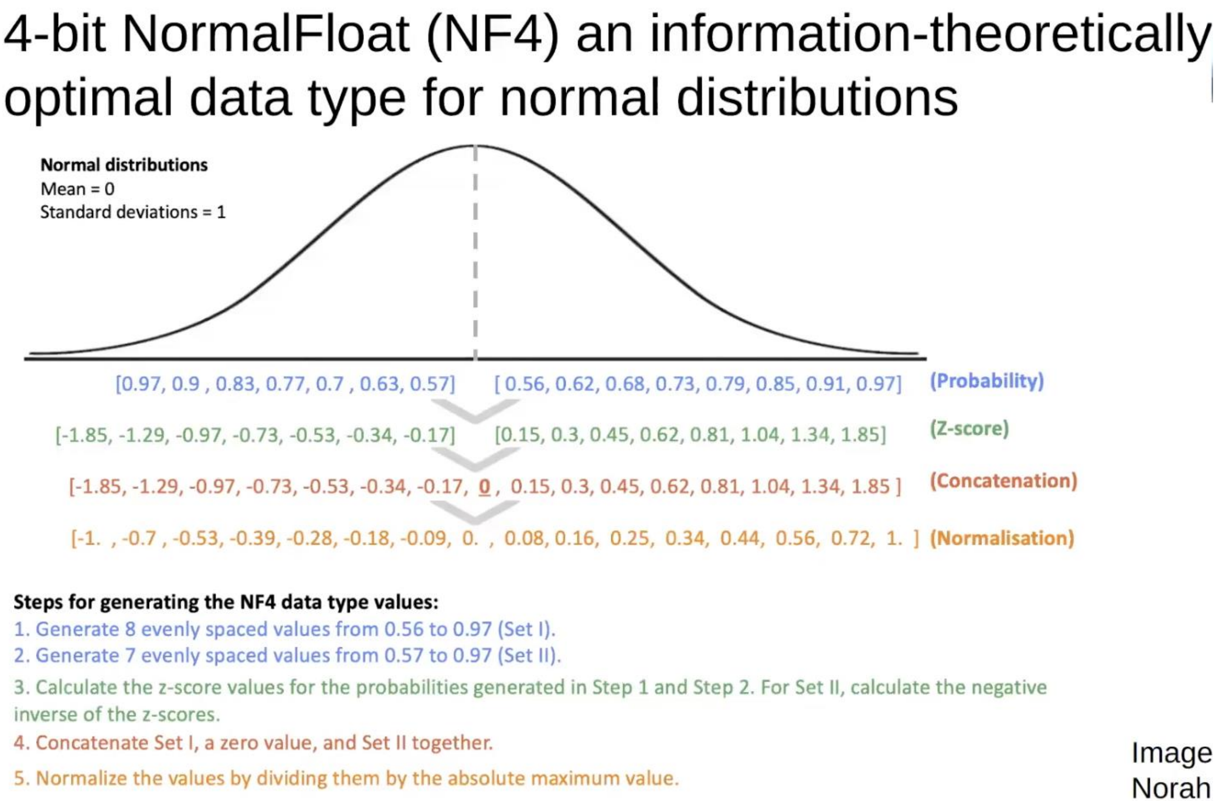


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

Key Idea: The LLM is loaded with 4bit. During training, the values are de-quantized to bf16 for processing. Utilizing the features of LoRA, the original model parameters can be locked and excluded from training. Only a small number of LoRA parameters are trained, which significantly reduces the required GPU memory.

QLoRA: 4-bit NormalFloat Quantization



4-bit NormalFloat (NF4) an information-theoretically optimal data type for normal distributions

[0.97, 0.9 , 0.83, 0.77, 0.7 , 0.63, 0.57] [0.56, 0.62, 0.68, 0.73, 0.79, 0.85, 0.91, 0.97] **(Probability)**

[-1.85, -1.29, -0.97, -0.73, -0.53, -0.34, -0.17] [0.15, 0.3, 0.45, 0.62, 0.81, 1.04, 1.34, 1.85] **(Z-score)**

[-1.85, -1.29, -0.97, -0.73, -0.53, -0.34, -0.17, 0 , 0.15, 0.3, 0.45, 0.62, 0.81, 1.04, 1.34, 1.85] **(Concatenation)**

[-1. , -0.7 , -0.53, -0.39, -0.28, -0.18, -0.09, 0. , 0.08, 0.16, 0.25, 0.34, 0.44, 0.56, 0.72, 1.] **(Normalisation)**

Steps for generating the NF4 data type values:

1. Generate 8 evenly spaced values from 0.56 to 0.97 (Set I).
2. Generate 7 evenly spaced values from 0.57 to 0.97 (Set II).
3. Calculate the z-score values for the probabilities generated in Step 1 and Step 2. For Set II, calculate the negative inverse of the z-scores.
4. Concatenate Set I, a zero value, and Set II together.
5. Normalize the values by dividing them by the absolute maximum value.

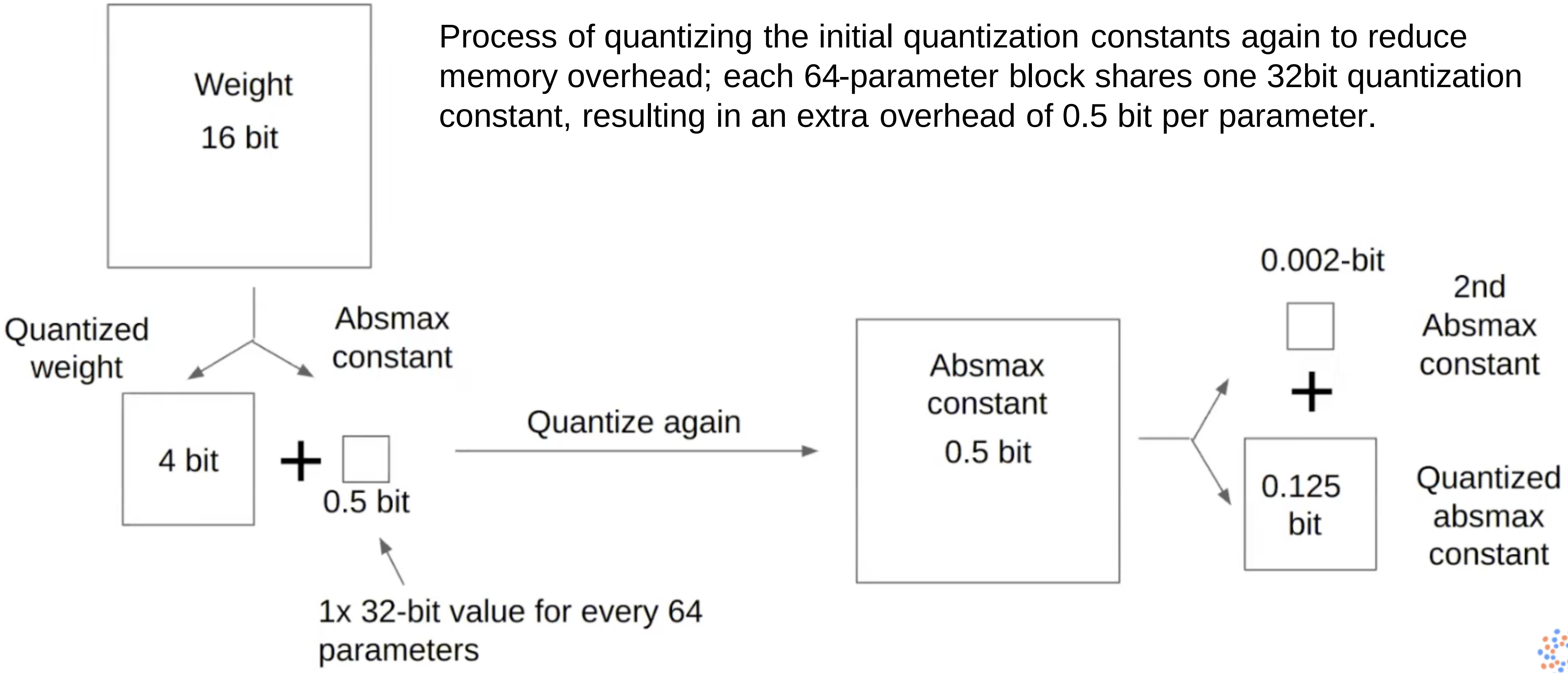
Image
Norah

- Based on quantile quantization techniques, making data conform to a N(0,1) distribution.

- Adjusts weights to match the data type range by using scaling factors and maintaining zero points during quantization.

QLoRA: Double Quantization

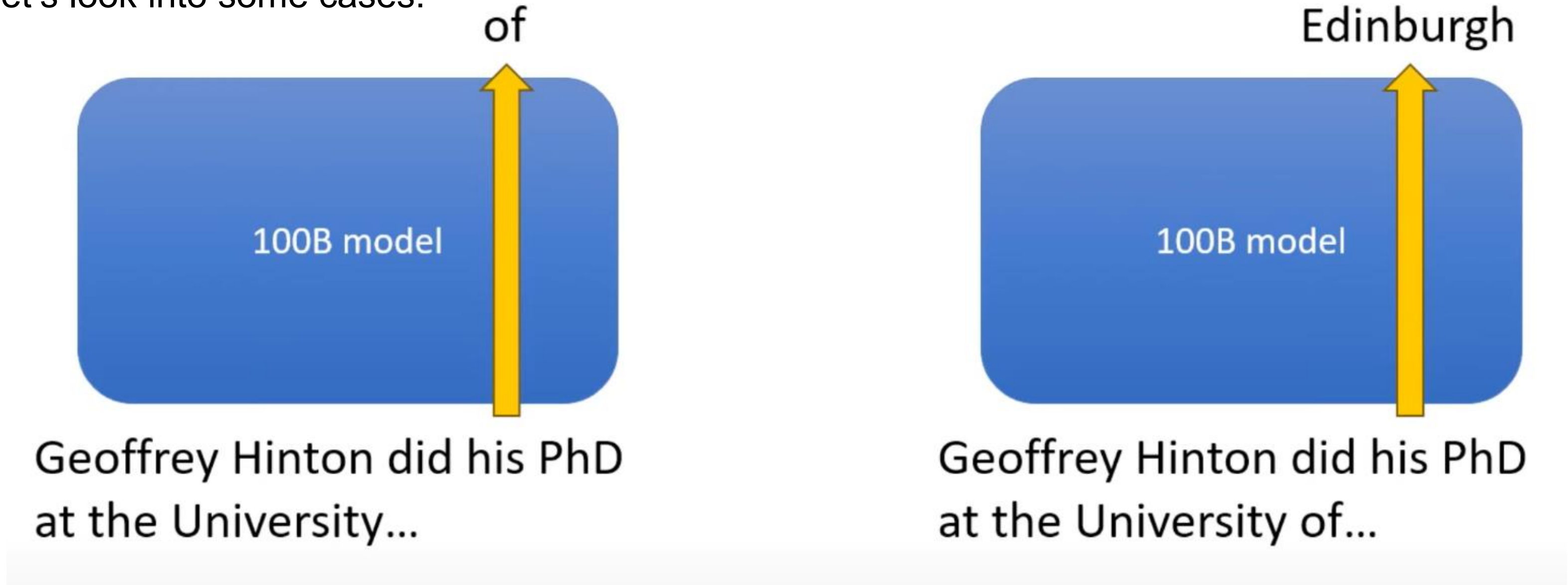
Process of quantizing the initial quantization constants again to reduce memory overhead; each 64-parameter block shares one 32bit quantization constant, resulting in an extra overhead of 0.5 bit per parameter.



Efficiency Beyond Transformers - Speculative Sampling

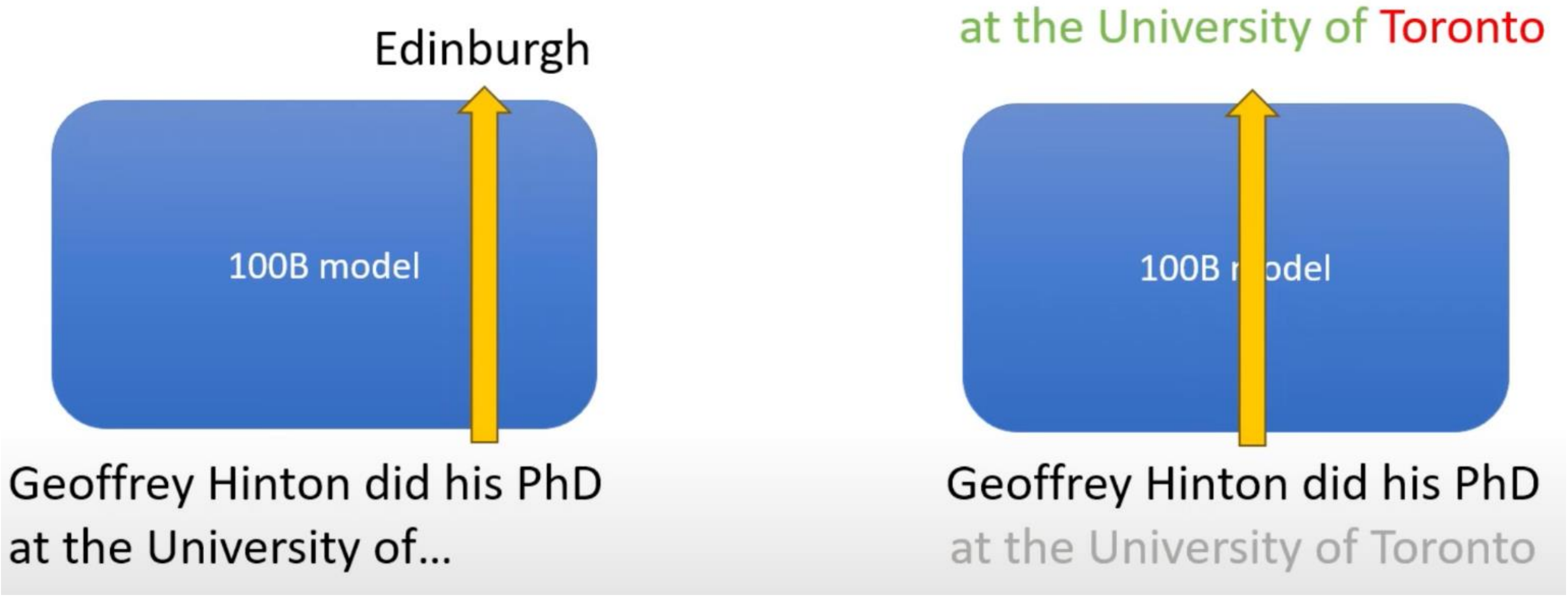
Speculative Sampling - Single Token Prediction

Let’s look into some cases:


of
100B model
Geoffrey Hinton did his PhD at the University…
Edinburgh
100B model
Geoffrey Hinton did his PhD at the University of…

- 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 - Utilize Transformer Structure

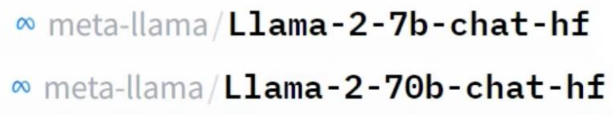

Edinburgh
100B model
Geoffrey Hinton did his PhD at the University of…
at the University of Toronto
100B model
Geoffrey Hinton did his PhD at the University of Toronto

- 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 Sampling - Algorithm

M_p = draft model

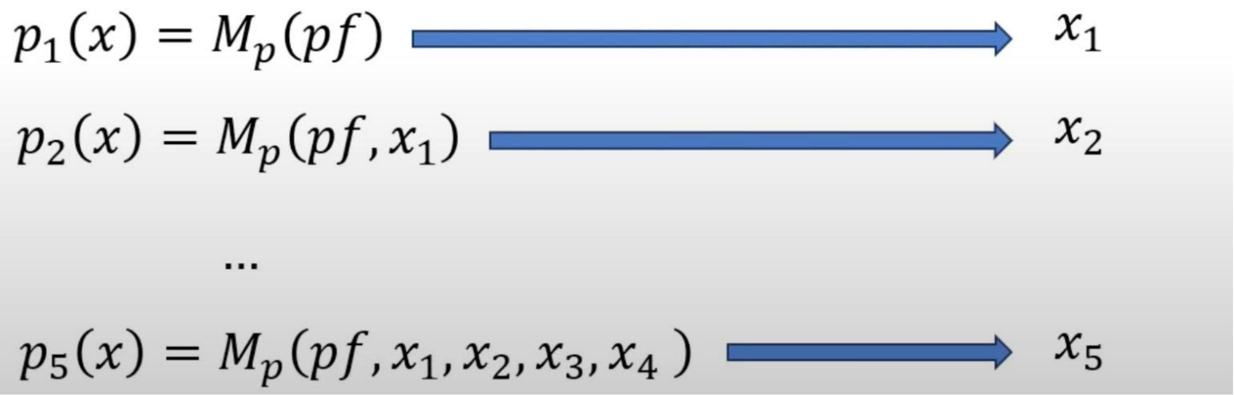
M_q = target model



∞ meta-llama/**Llama-2-7b-chat-hf**

∞ meta-llama/**Llama-2-70b-chat-hf**

pf = prefix, $K = 5$ tokens



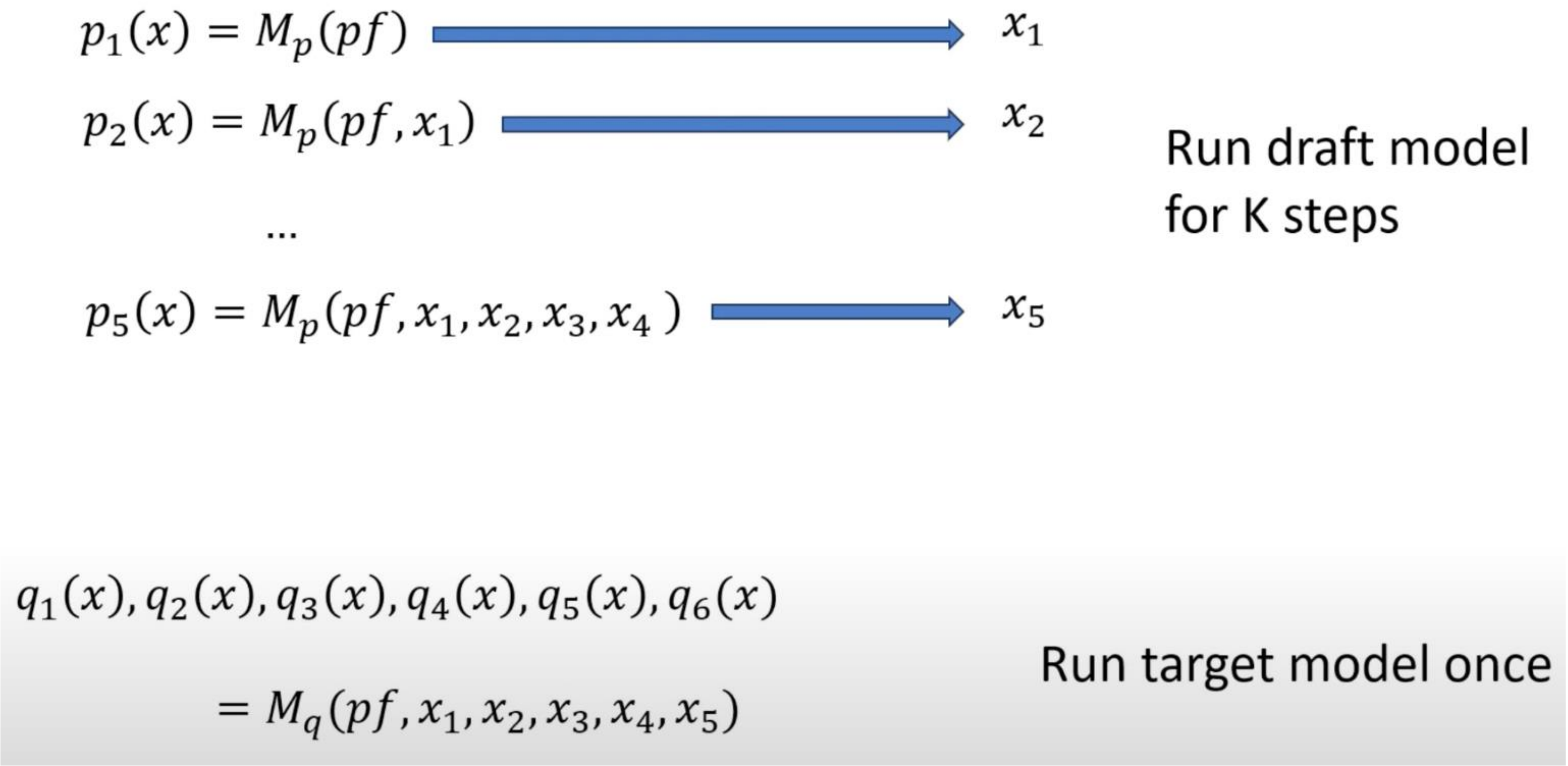
$$p_1(x) = M_p(pf) \longrightarrow x_1$$

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

...

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

Speculative Sampling - Algorithm



$$p_1(x) = M_p(pf) \xrightarrow{\qquad\qquad} x_1$$

$$p_2(x) = M_p(pf, x_1) \xrightarrow{\qquad\qquad} x_2$$

...

$$p_5(x) = M_p(pf, x_1, x_2, x_3, x_4\,) \xrightarrow{\qquad\qquad} x_5$$

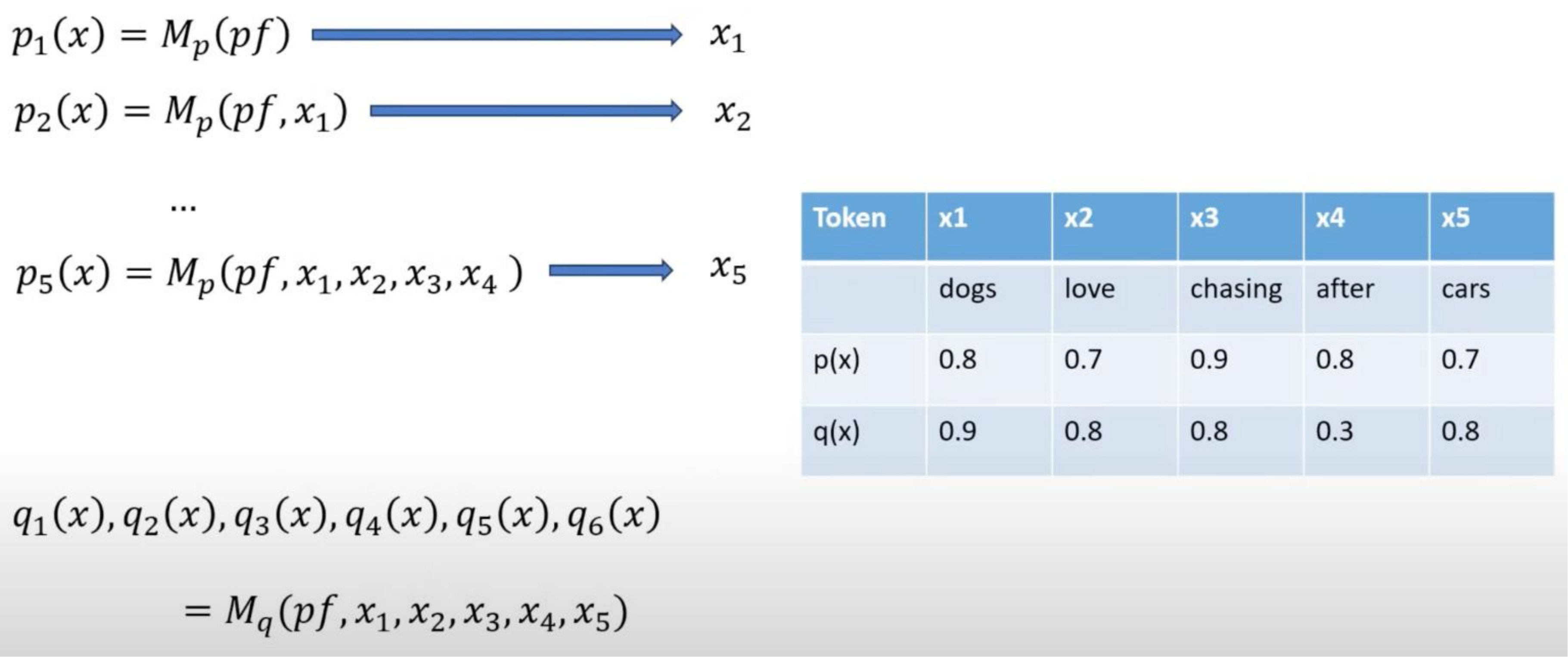
Run draft model for K steps

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

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

Run target model once

Speculative Sampling - Algorithm



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

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

$$\dots$$

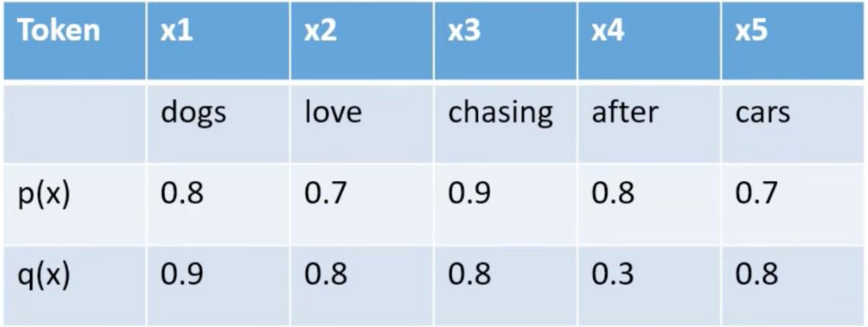
$$p_5(x) = M_p(pf, x_1, x_2, x_3, x_4) \longrightarrow x_5$$

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

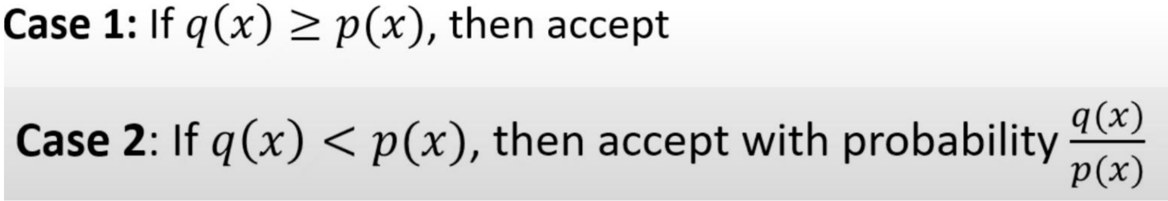
$$q_1(x), q_2(x), q_3(x), q_4(x), q_5(x), q_6(x)$$

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

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

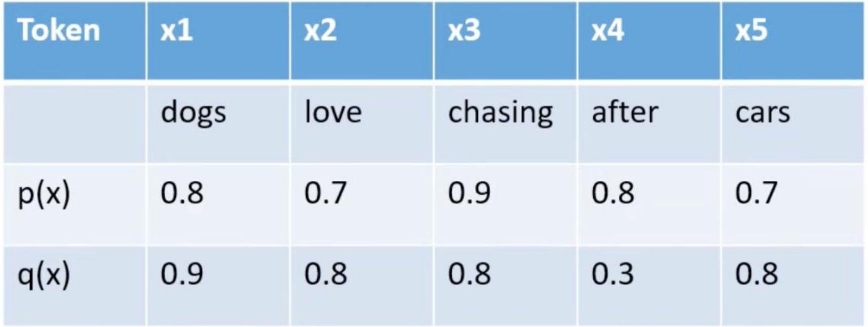


Case 1: If $q(x) \geq p(x)$, then accept

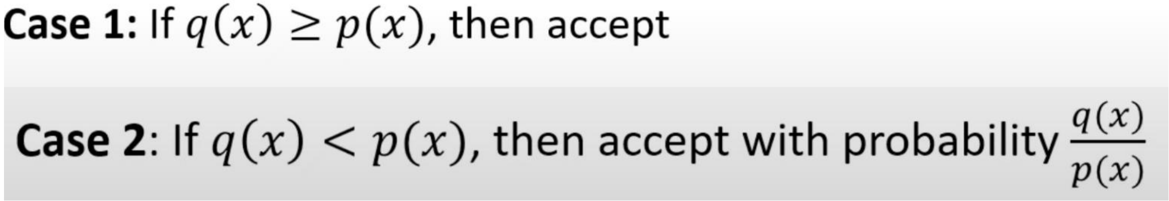
Case 2: If $q(x) < p(x)$, then accept with probability $\frac{q(x)}{p(x)}$

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

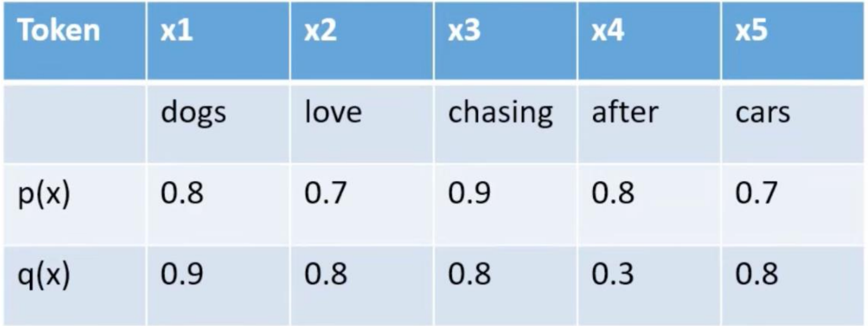


Case 1: If $q(x) \geq p(x)$, then accept

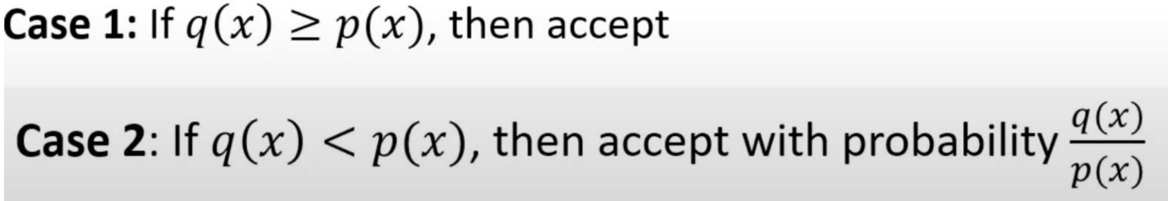
Case 2: If $q(x) < p(x)$, then accept with probability $\frac{q(x)}{p(x)}$

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

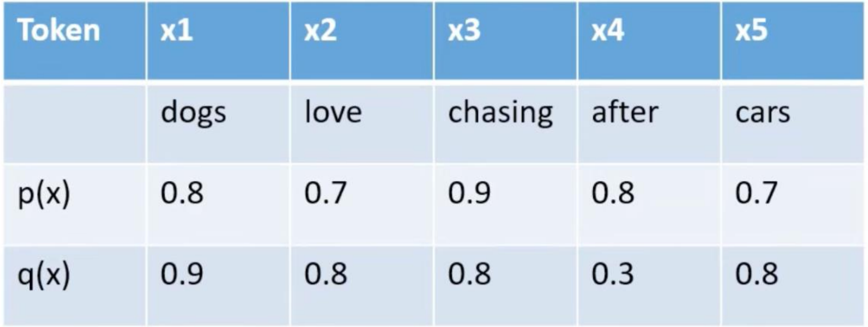


Case 1: If $q(x) \geq p(x)$, then accept

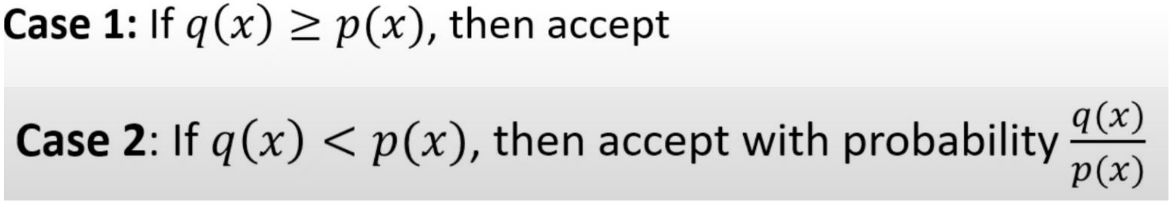
Case 2: If $q(x) < p(x)$, then accept with probability $\frac{q(x)}{p(x)}$

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

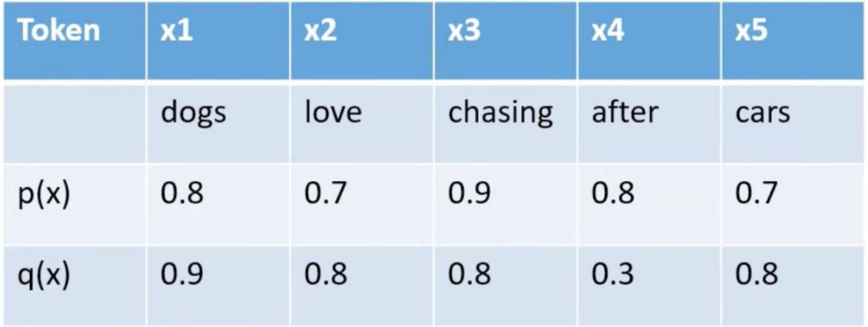


Case 1: If $q(x) \geq p(x)$, then accept

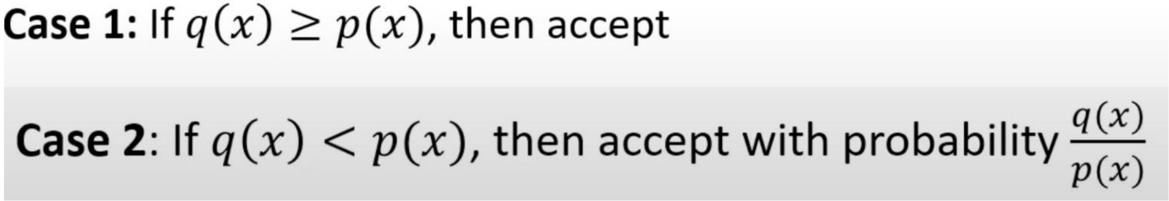
Case 2: If $q(x) < p(x)$, then accept with probability $\frac{q(x)}{p(x)}$

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

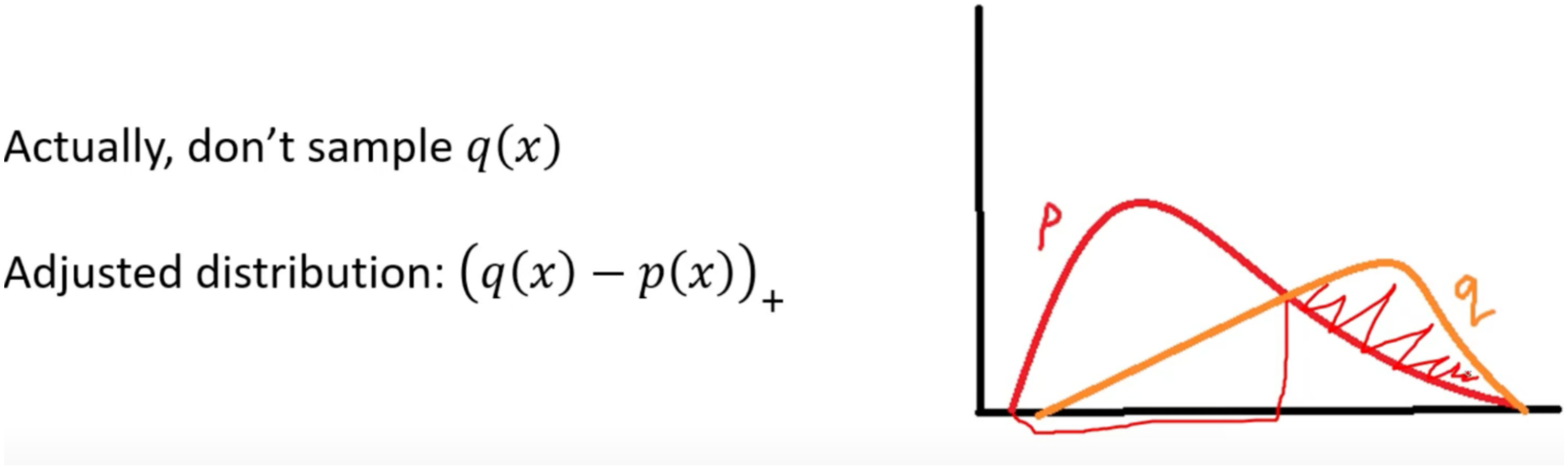


Case 1: If $q(x) \geq p(x)$, then accept

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)!

Speculative Sampling - Rejection Sampling



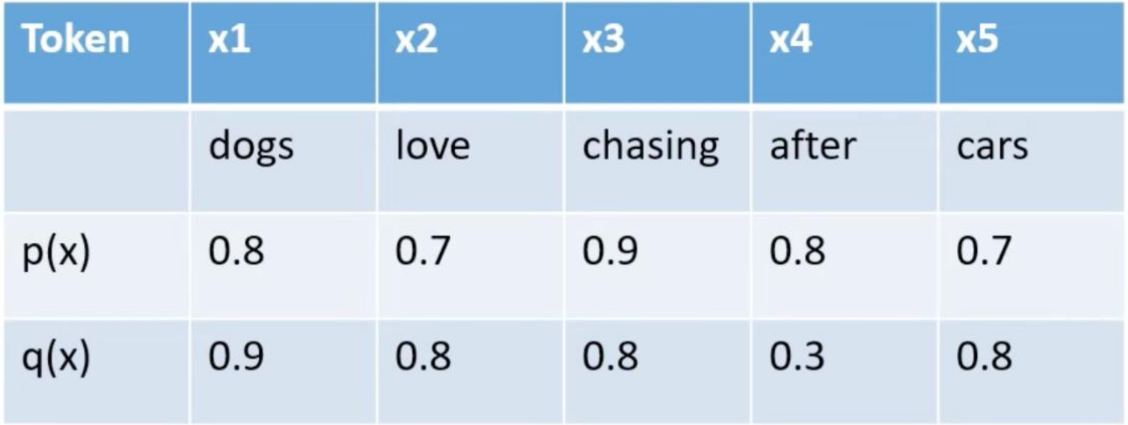
Actually, don’t sample $q(x)$

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

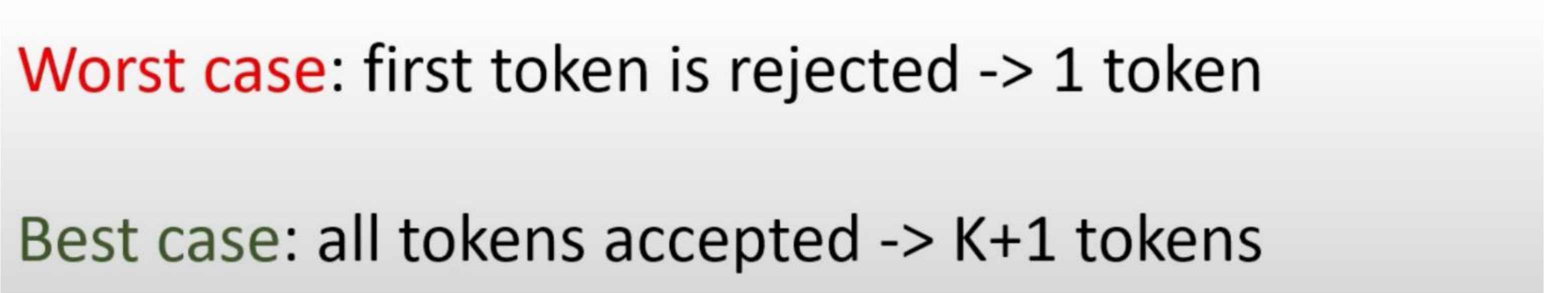
We sample the 4th token by (q(4) - p(4))+!

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

Speculative Sampling - #tokens generated in one pass



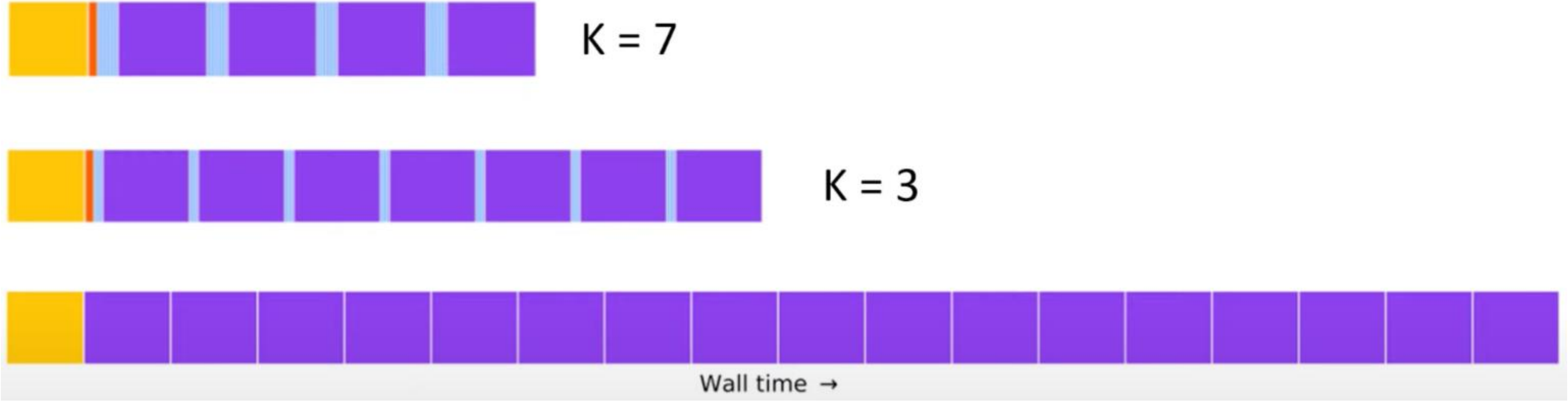
Token	x1	x2	x3	x4	x5
	dogs	love	chasing	after	cars
p(x)	0.8	0.7	0.9	0.8	0.7
q(x)	0.9	0.8	0.8	0.3	0.8



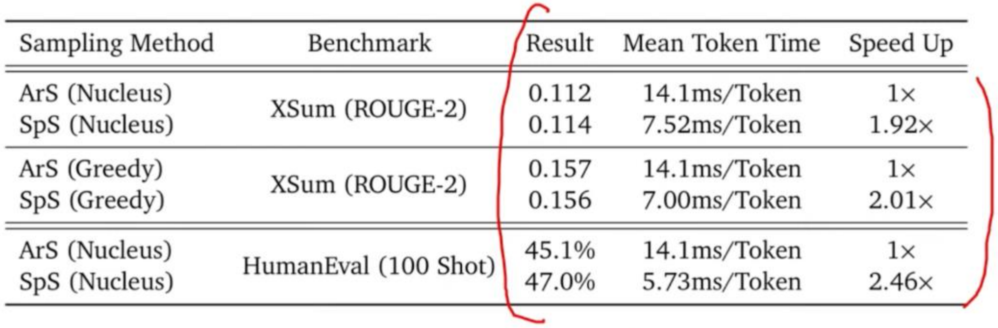
Worst case: first token is rejected -> 1 token

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

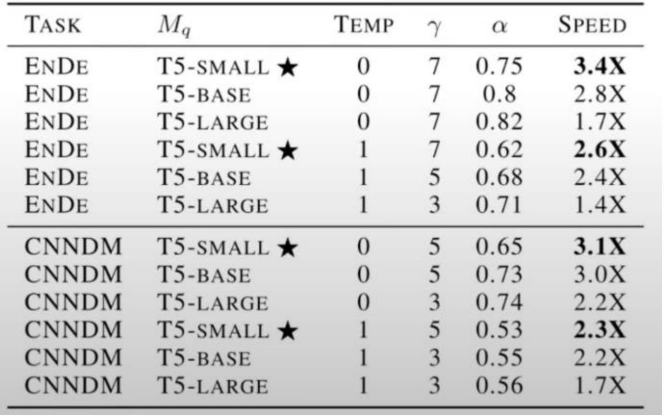
Recommends K = 3-4

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

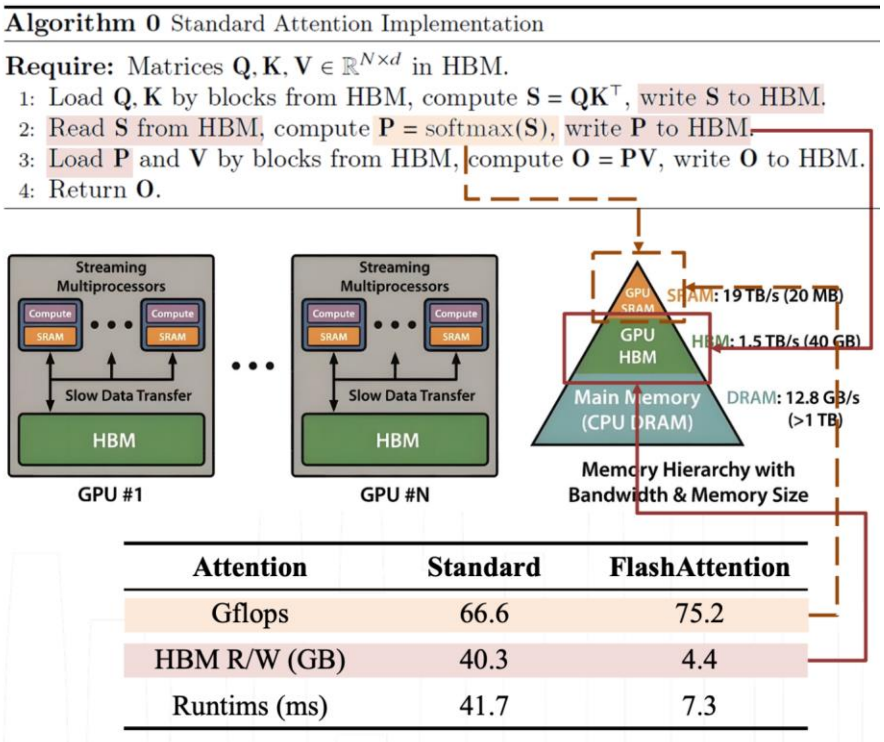
System Support for Efficiency - Flash-Attention

Flash-Attention Overview

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)



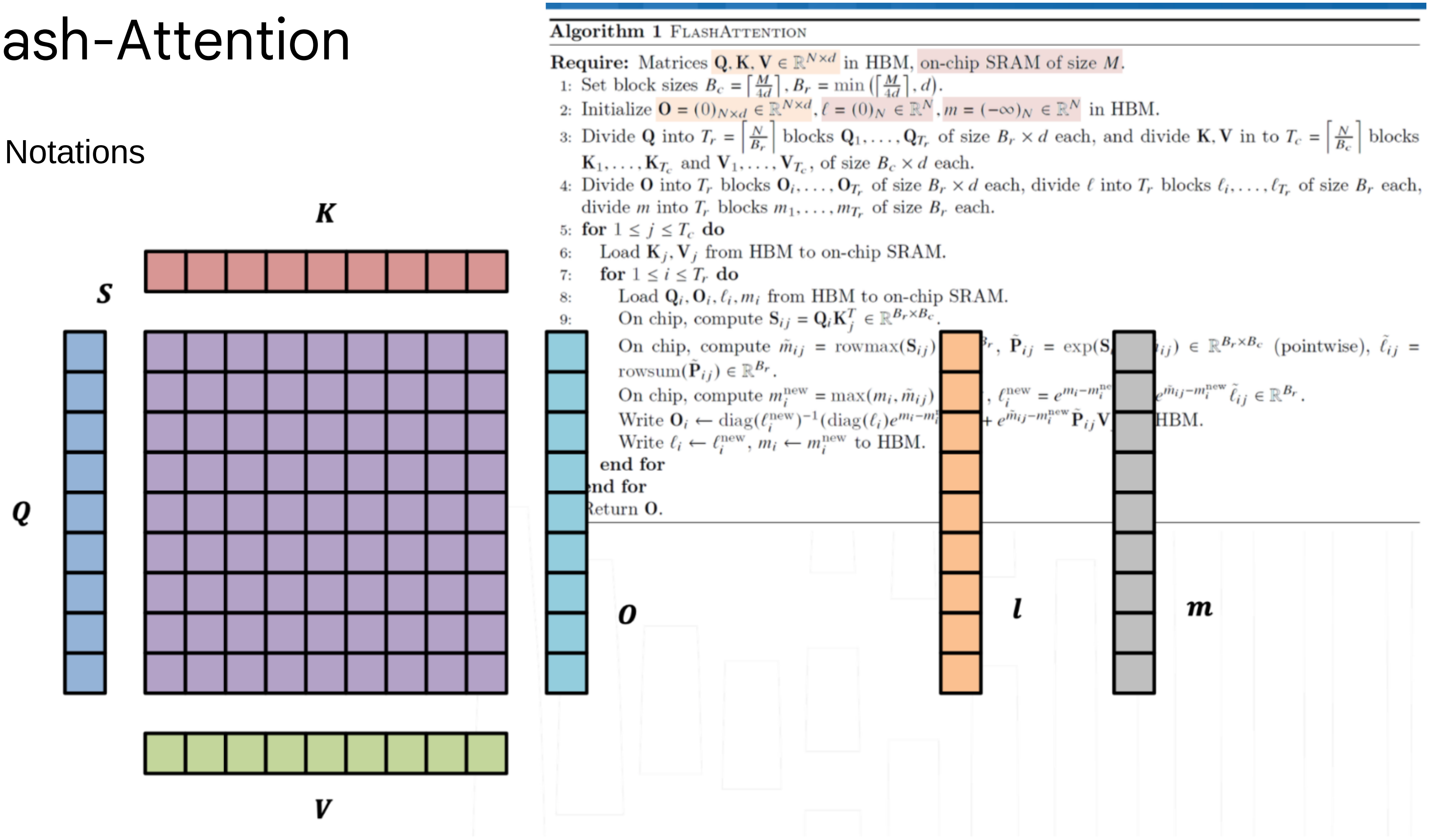
Algorithm 0 Standard Attention Implementation

```
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.
Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
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

Flash-Attention

Notations

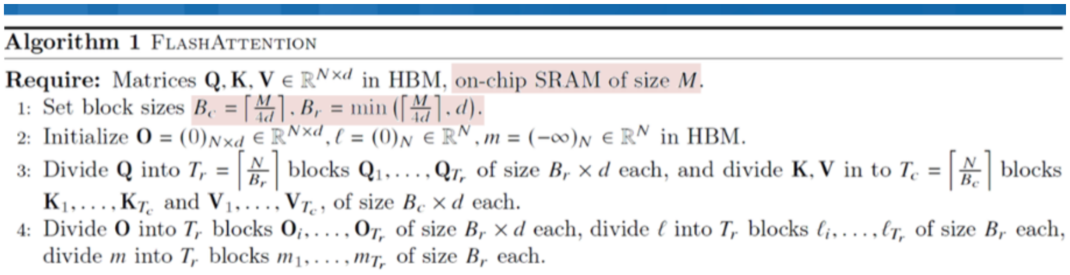


Algorithm 1 FlashAttention

```
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size $M$.
Set block sizes $B_c = \left\lceil \frac{M}{4d} \right\rceil, B_r = \min\left(\left\lceil \frac{M}{4d} \right\rceil, d\right)$.
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.
Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ 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.
Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\ell$ 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.
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 $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij})$ $^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}$ $_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} =$
        $\mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
    On chip, compute $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij})$ , $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{ne}}}$ $e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
    Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1}(\mathrm{diag}(\ell_i) e^{m_i - m_i}$ $+ e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}$ HBM.
    Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}, m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
  end for
nd for
Return $\mathbf{O}$.
```

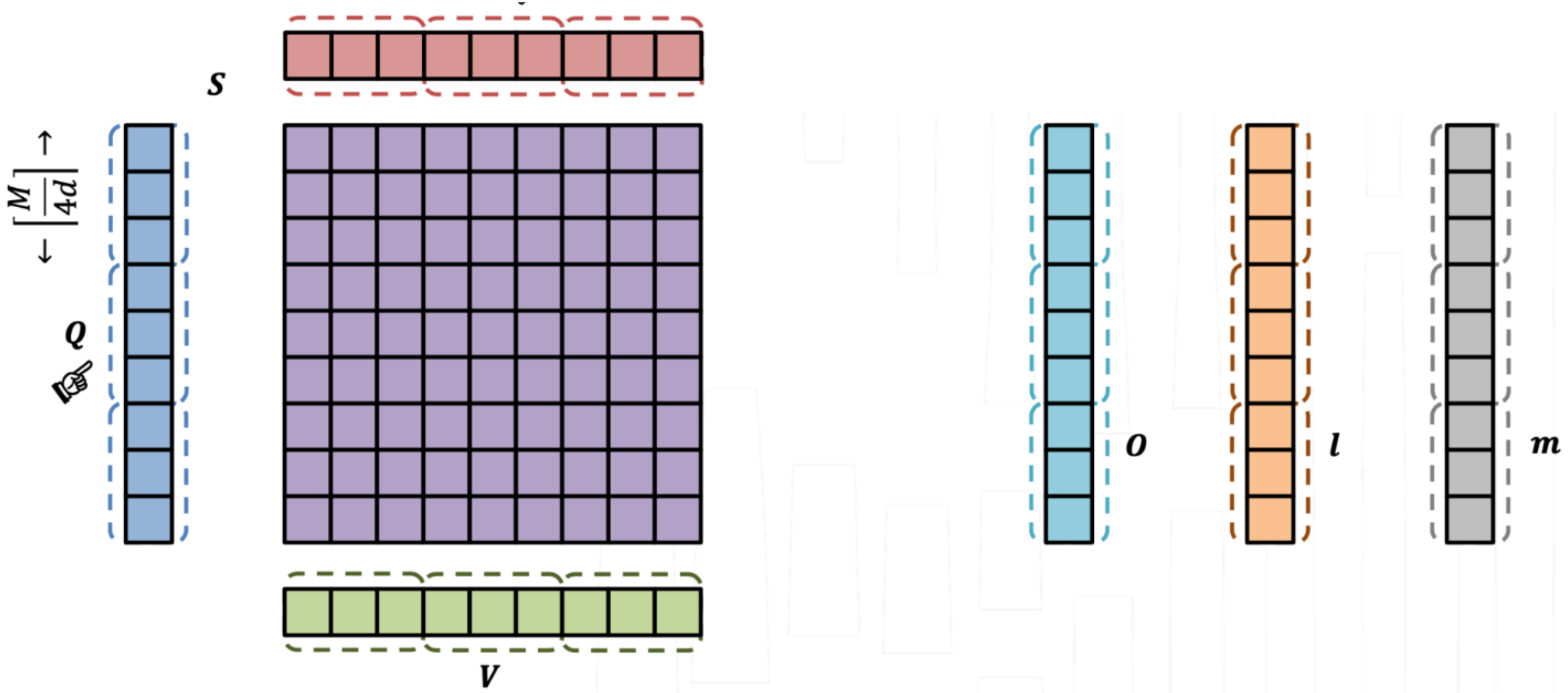
Flash-Attention

Split Blocks

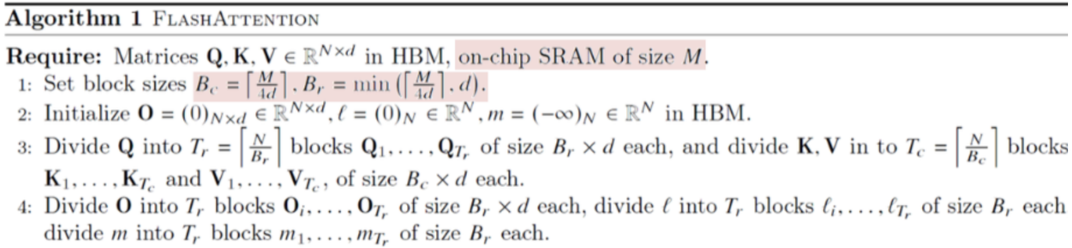


Algorithm 1 FlashAttention

```
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 $\ell$ 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.
```

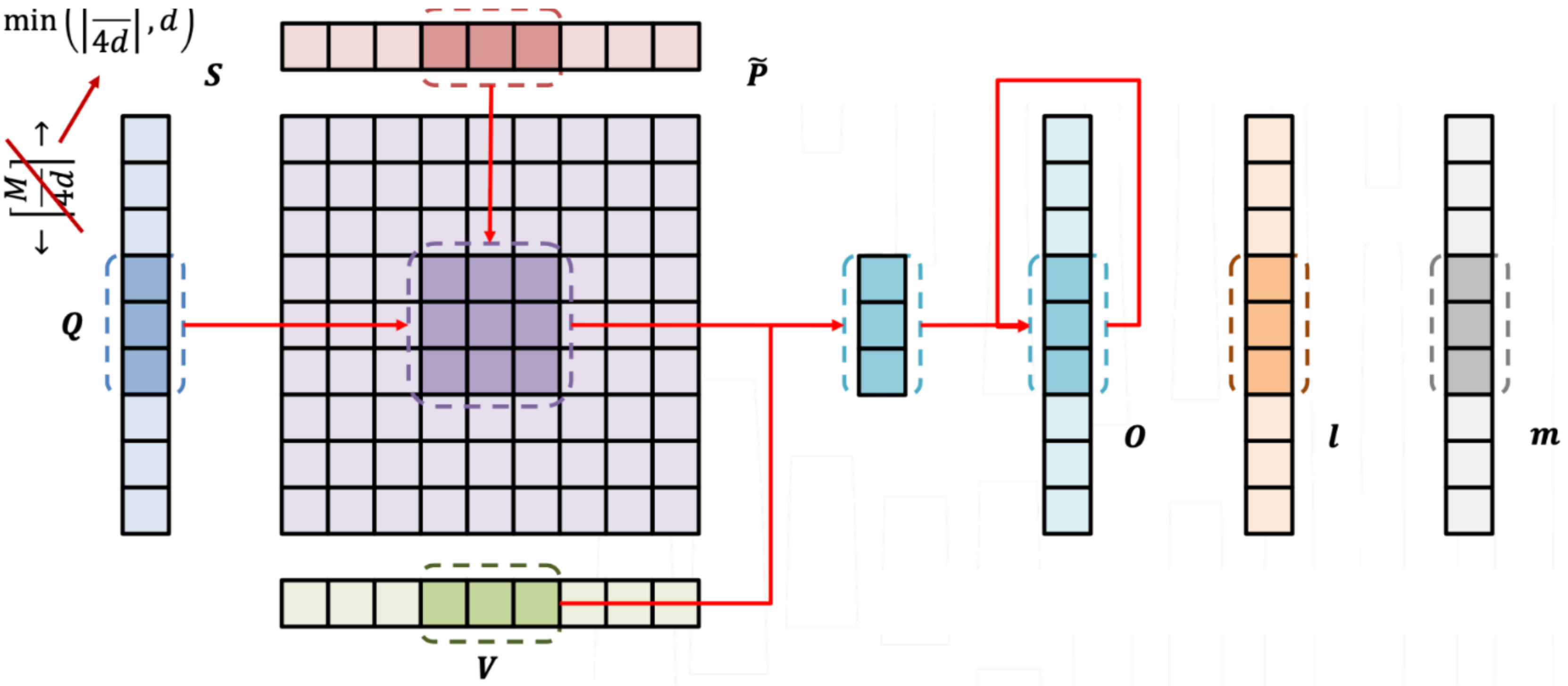


Flash-Attention



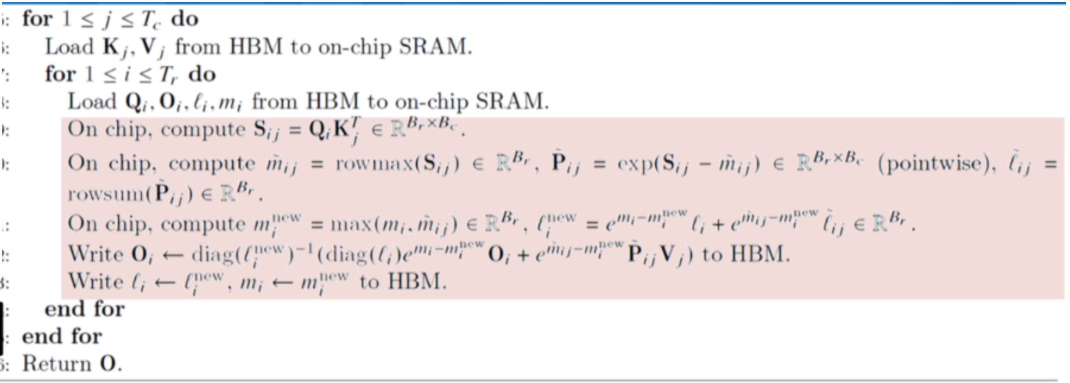
```
Algorithm 1 FlashAttention
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size $M$.
Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
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.
Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\ell$ 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

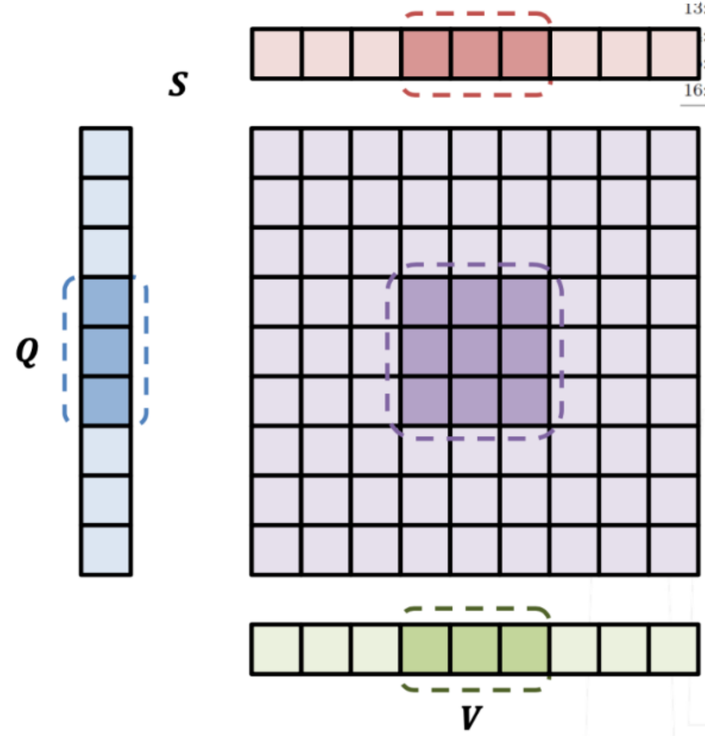

$\min\left(\lceil \overline{4d} \rceil, d\right)$
$\cancel{\lceil \frac{M}{4d} \rceil}$
S
$\tilde{P}$
Q
O
l
m
V

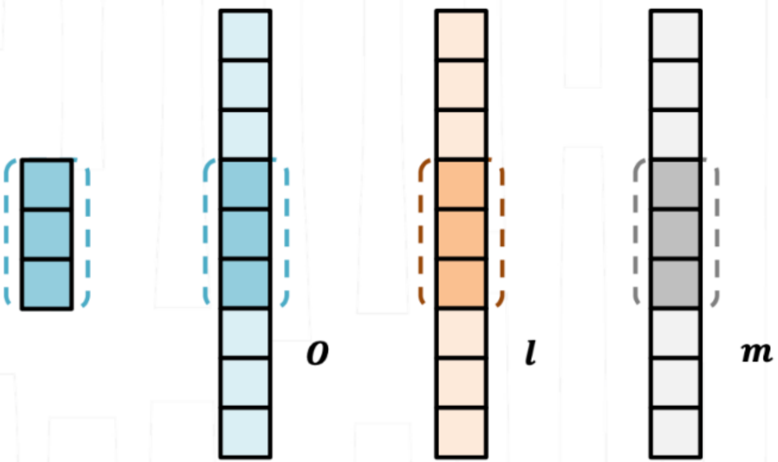
Flash-Attention

Softmax Reduction



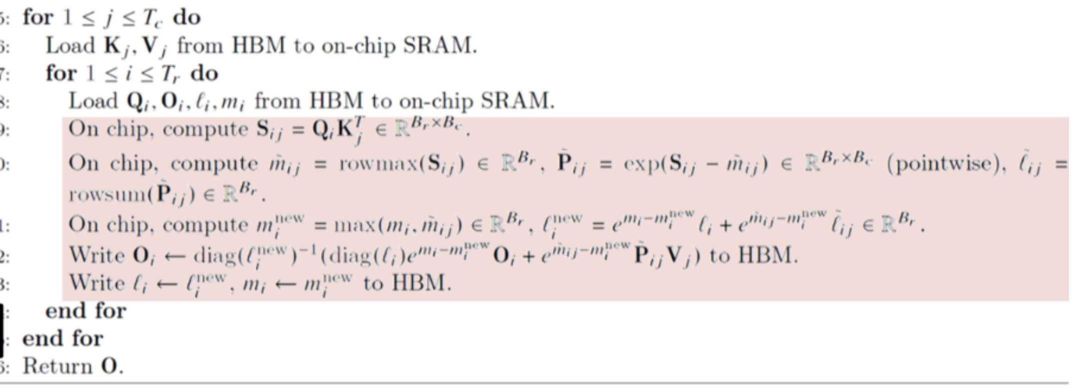
```
for $1 \le j \le T_c$ do
    Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
    for $1 \le i \le T_r$ do
        Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
        On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i\mathbf{K}_j^T \in \mathbb{R}^{B_r \times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



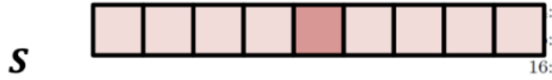


Flash-Attention

Softmax Reduction

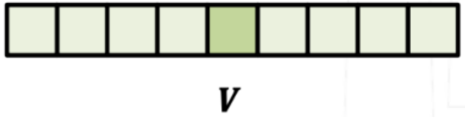


```
for $1 \le j \le T_c$ do
    Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
    for $1 \le i \le T_r$ do
        Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
        On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i\mathbf{K}_j^T \in \mathbb{R}^{B_r \times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



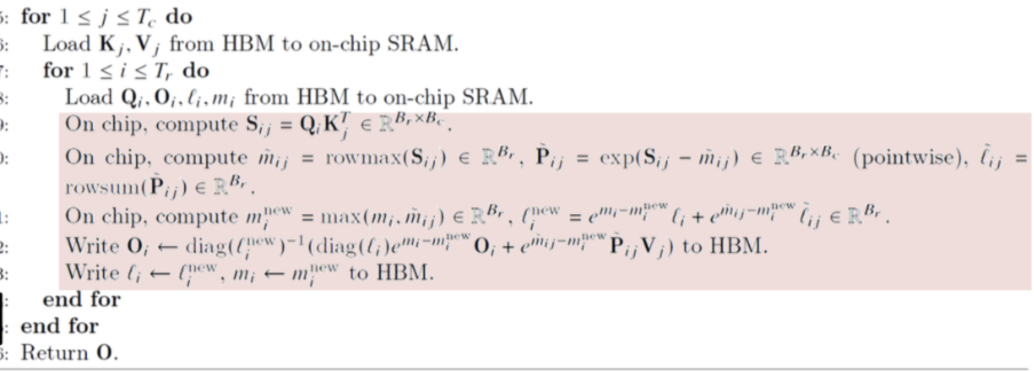
$$O_i = \frac{e^{Q_iK_1^T}\cdot V_1}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} + \cdots + \frac{e^{Q_iK_N^T}\cdot V_N}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}}$$

online softmax $O_1 = \frac{e^{Q_iK_1^T}\cdot V_1}{\sum_{j'}^{1} e^{Q_iK_{j'}^T}}$ $O_j = O_{j-1}\cdot\frac{\sum_{j'}^{j-1} e^{Q_iK_{j'}^T}}{\sum_{j'}^{j} e^{Q_iK_{j'}^T}} + \frac{e^{Q_iK_j^T}\cdot V_j}{\sum_{j'}^{j} e^{Q_iK_{j'}^T}}$

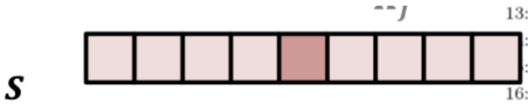


Flash-Attention

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 $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
    On chip, compute $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}}\ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}}\tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
    Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1}(\mathrm{diag}(\ell_i)e^{m_i - m_i^{\mathrm{new}}}\mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}}\tilde{\mathbf{P}}_{ij}\mathbf{V}_j)$ to HBM.
    Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
  end for
end for
Return $\mathbf{O}$.
```

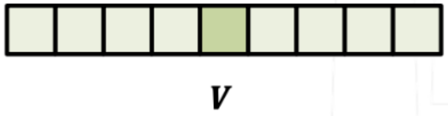


$$O_i = \frac{e^{Q_iK_1^T}\cdot V_1}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} + \cdots + \frac{e^{Q_iK_N^T}\cdot V_N}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} = \frac{e^{Q_iK_1^T}\cdot V_1}{\sum_{j'}^{1} e^{Q_iK_{j'}^T}} \cdot \frac{\sum_{j'}^{1} e^{Q_iK_{j'}^T}}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} + \cdots + \frac{e^{Q_iK_j^T}\cdot V_j}{\sum_{j'}^{j} e^{Q_iK_{j'}^T}} \cdot \frac{\sum_{j'}^{j} e^{Q_iK_{j'}^T}}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} + \cdots + \frac{e^{Q_iK_N^T}\cdot V_N}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} \cdot \frac{\sum_{j'}^{N} e^{Q_iK_{j'}^T}}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}}$$

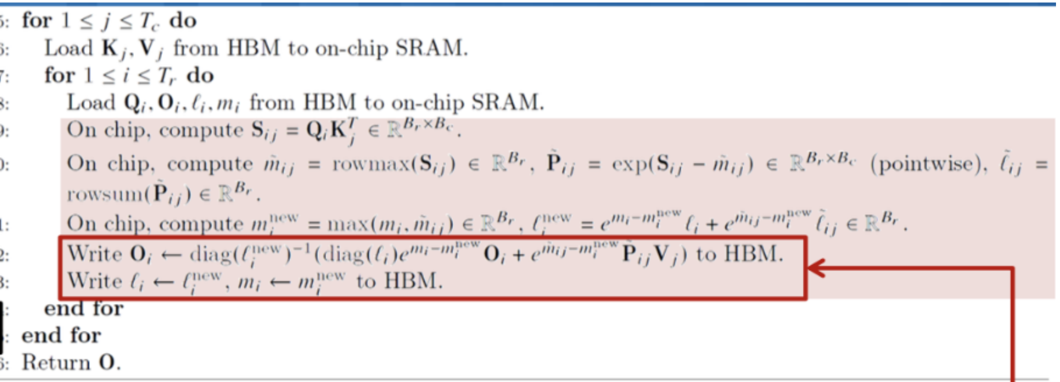
online softmax

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

$$l_{i1} = e^{S_{i1}} \qquad l_{ij} = l_{ij-1} + e^{S_{ij}}$$

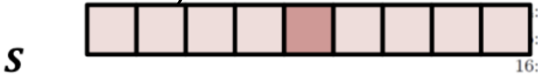


Flash-Attention



```
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 $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
    On chip, compute $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
    Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1} (\mathrm{diag}(\ell_i) e^{m_i - m_i^{\mathrm{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
    Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
  end for
end for
Return $\mathbf{O}$.
```

Softmax Reduction
(Numerical Stable)



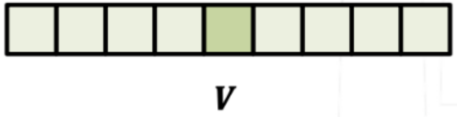
$$O_i = \frac{e^{Q_iK_1^T} \cdot V_1}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} + \cdots + \frac{e^{Q_iK_N^T} \cdot V_N}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} = \frac{e^{Q_iK_1^T} \cdot V_1}{\sum_{j'}^{1} e^{Q_iK_{j'}^T}} \cdot \frac{\sum_{j'}^{1} e^{Q_iK_{j'}^T}}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} + \cdots + \frac{e^{Q_iK_j^T} \cdot V_j}{\sum_{j'}^{j} e^{Q_iK_{j'}^T}} \cdot \frac{\sum_{j'}^{j} e^{Q_iK_{j'}^T}}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} + \cdots + \frac{e^{Q_iK_N^T} \cdot V_N}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}} \cdot \frac{\sum_{j'}^{N} e^{Q_iK_{j'}^T}}{\sum_{j'}^{N} e^{Q_iK_{j'}^T}}$$

online softmax

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

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

$_iK_{j'}^T$



Flash-Attention

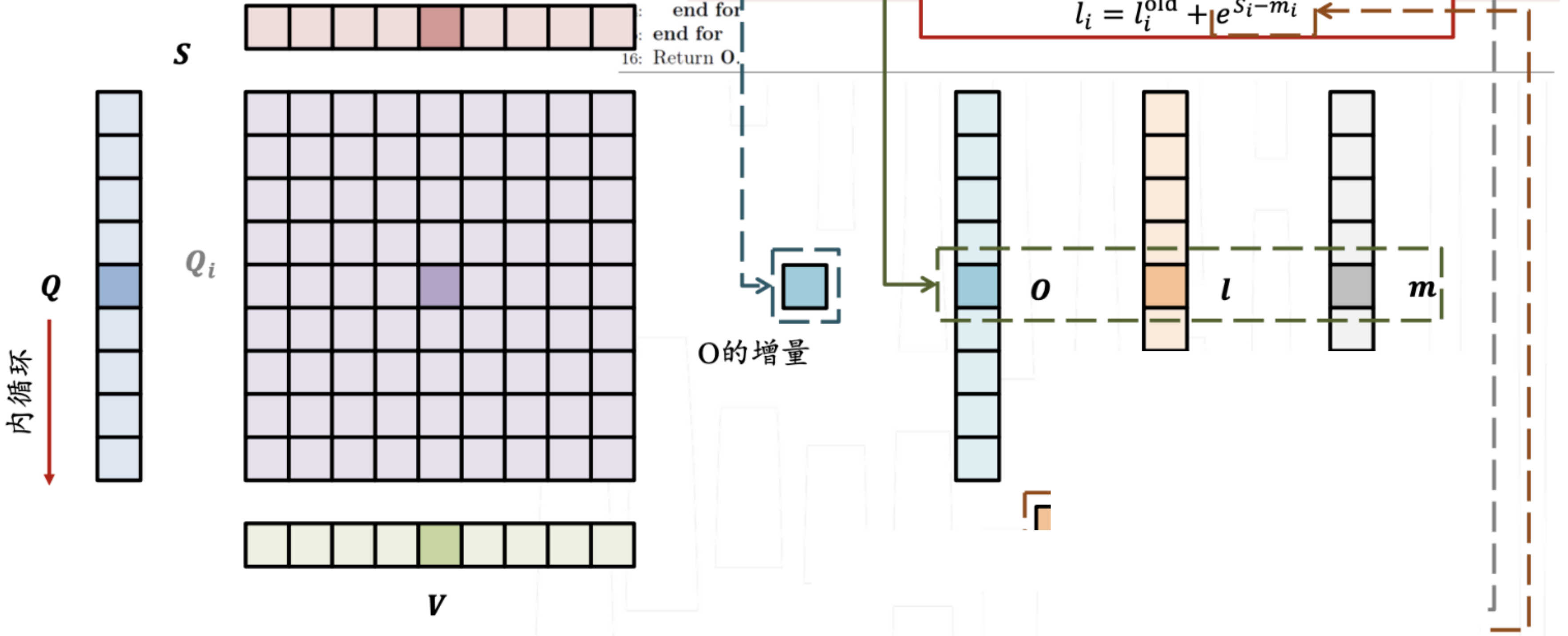
Softmax Reduction

```
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 $\mathbf{S}_{ij} = \mathbf{Q}$
        On chip, compute $\tilde{m}_{ij} =$ … wise), $\tilde{\ell}_{ij} =$
        rowsum$(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
        On chip, compute $m_i^{\text{new}} =$
        Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}($
        Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{n}}$
    end for
end for
Return $\mathbf{O}$.
```

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

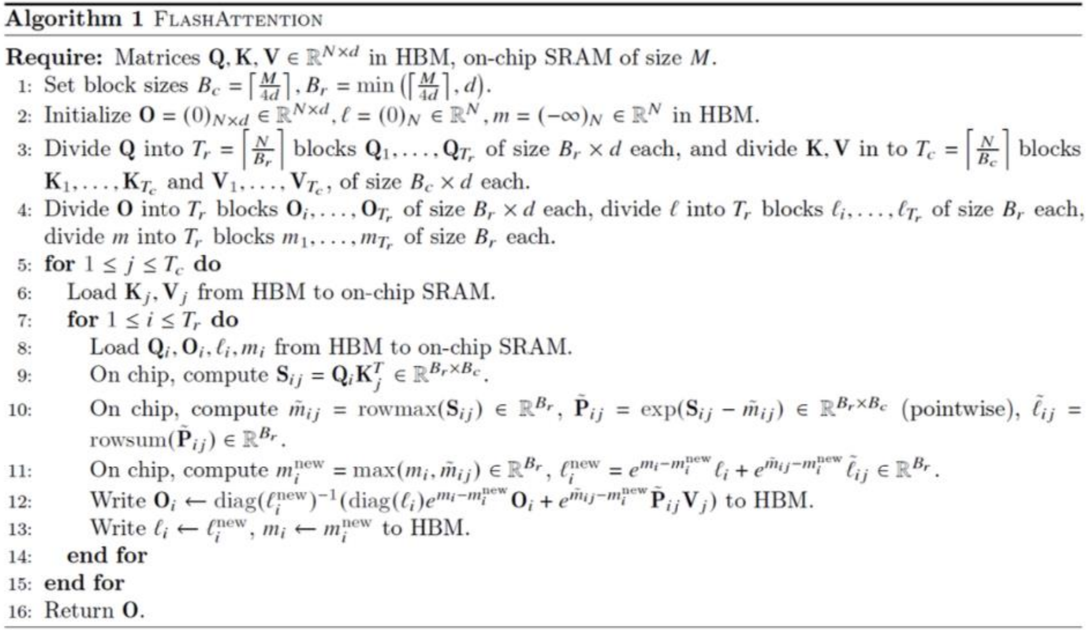
$$m_i = \max(m_i^{\text{old}}, S_i)$$

$$l_i = l_i^{\text{old}} + e^{S_i - m_i}$$



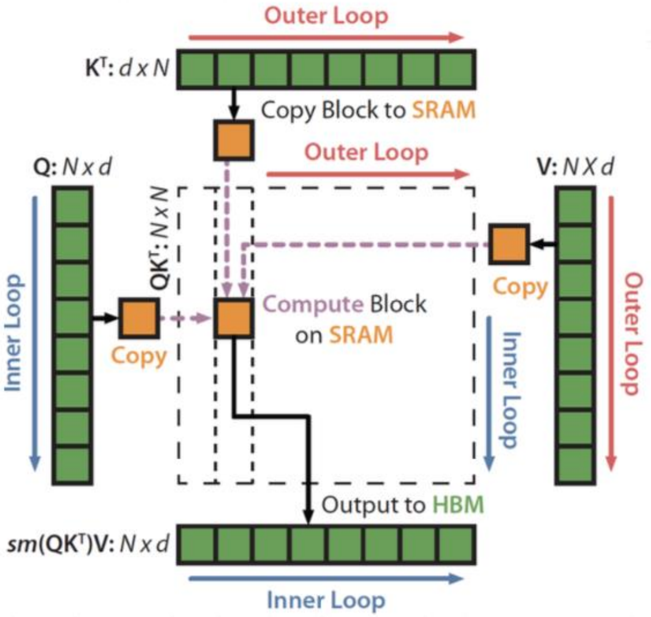
Flash-Attention

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 $M$.
Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
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.
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.
Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\ell$ 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.
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 $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
        On chip, compute $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
        Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1}(\mathrm{diag}(\ell_i) e^{m_i - m_i^{\mathrm{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
        Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
    end for
end for
Return $\mathbf{O}$.
```


Outer Loop
K^T: d x N
Copy Block to SRAM
Outer Loop
Q: N x d
V: N X d
QK^T: N x N
Copy
Compute Block on SRAM
Copy
Inner Loop
Inner Loop
Outer Loop
Output to HBM
sm(QK^T)V: N x d
Inner Loop