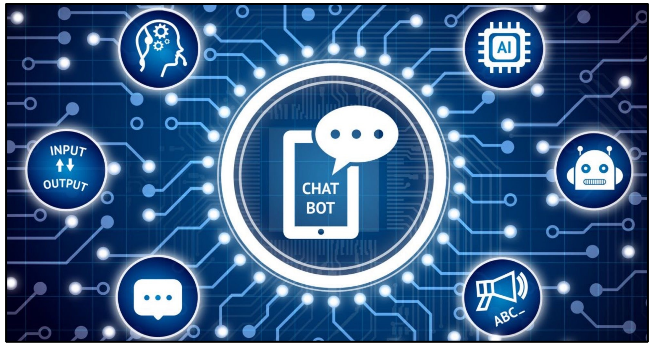
Memory-Efficient LLM Training

Jiawei Zhao

Department of Computing + Mathematical Sciences

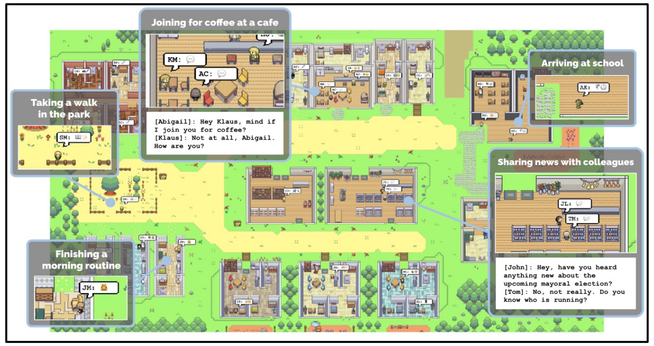
California Institute of Technology

Neural Networks - Foundation Models

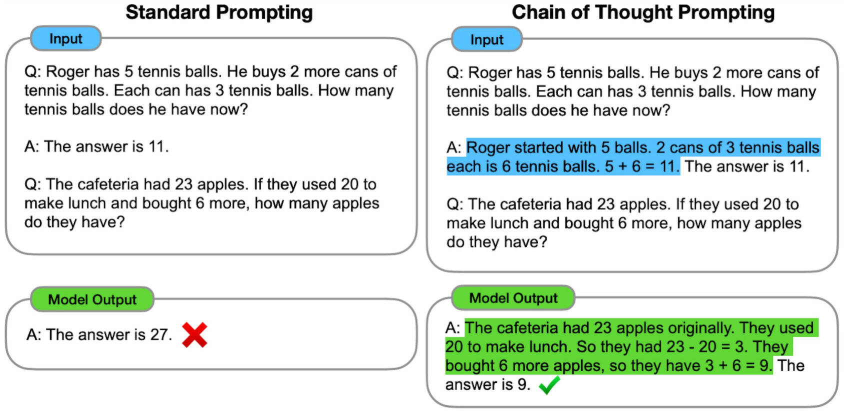


Conversational AI

Content Generation



AI Agents

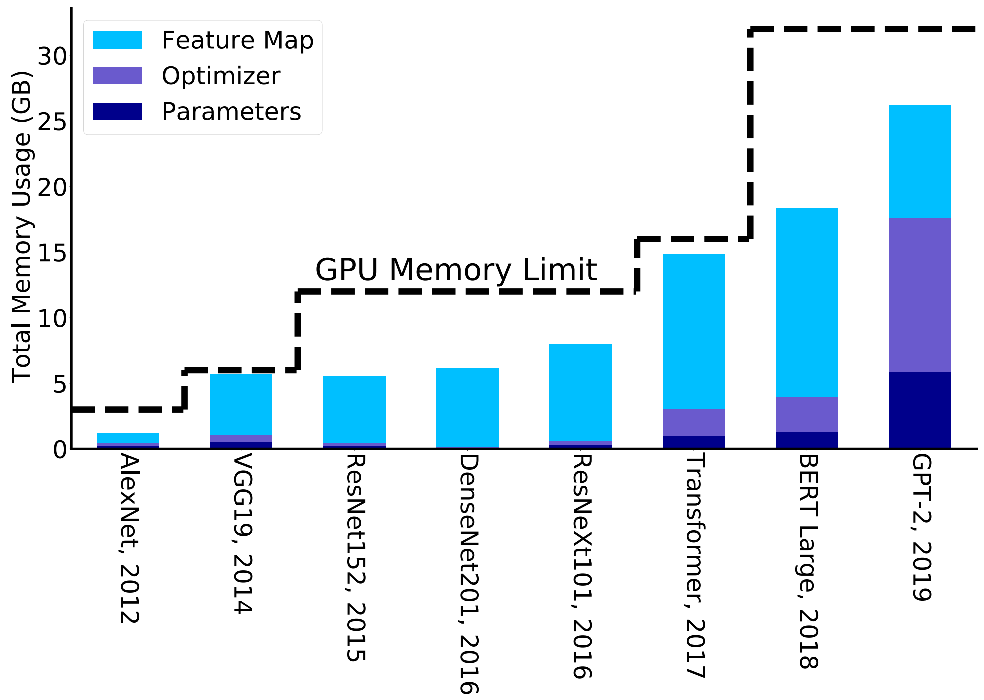


Reasoning

Planning

Training of Large Language Models

GPT-3[1]
- 4 Months
- 1000 GPUs, > $10M
- Massive carbon footprint
- 175B parameters, 45TB data



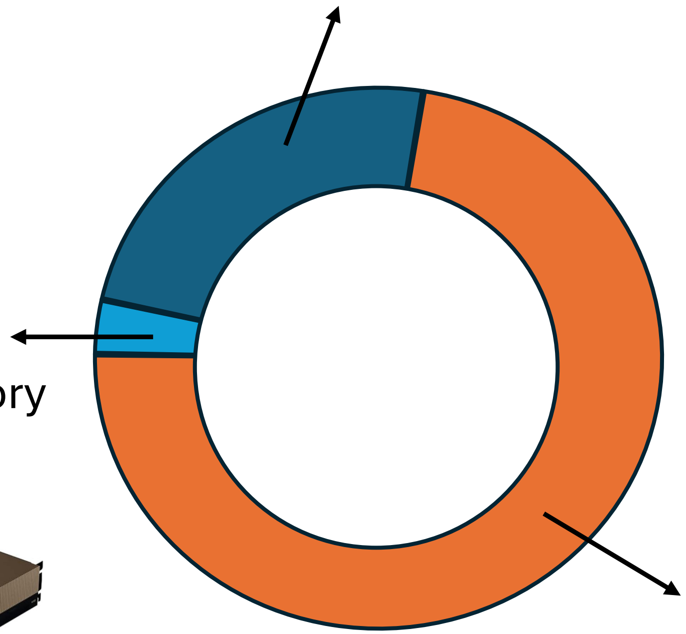
Impact on Memory: Parameter Size > Activation
(optimizer states and weights require more memory than before)

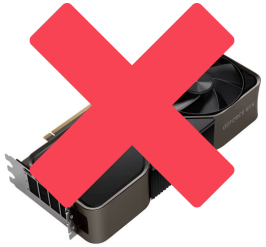
Figure is from Gholami, Amir, et al. "Ai and memory wall." *IEEE Micro* (2024).

Minimum Memory Requirement

Pre-Training LLaMA 7B model (BF16) from scratch with a single batch size requires:

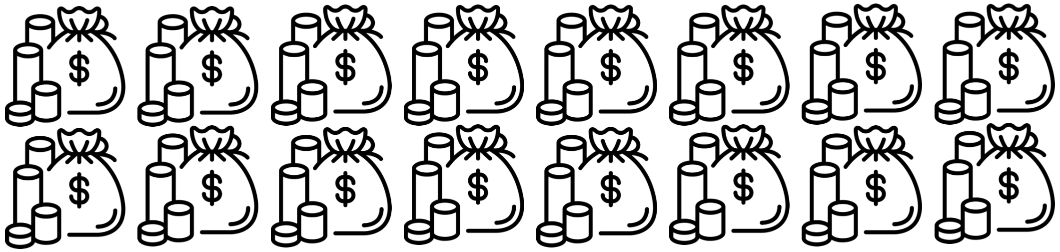
- Trainable Parameters: 14GB
- Adam Optimizer States: 28GB (storing first and second estimates)
- Gradients: 14GB
- Activations: 2GB
- In total: 58GB





RTX 4090: 24GB

H100: 80GB

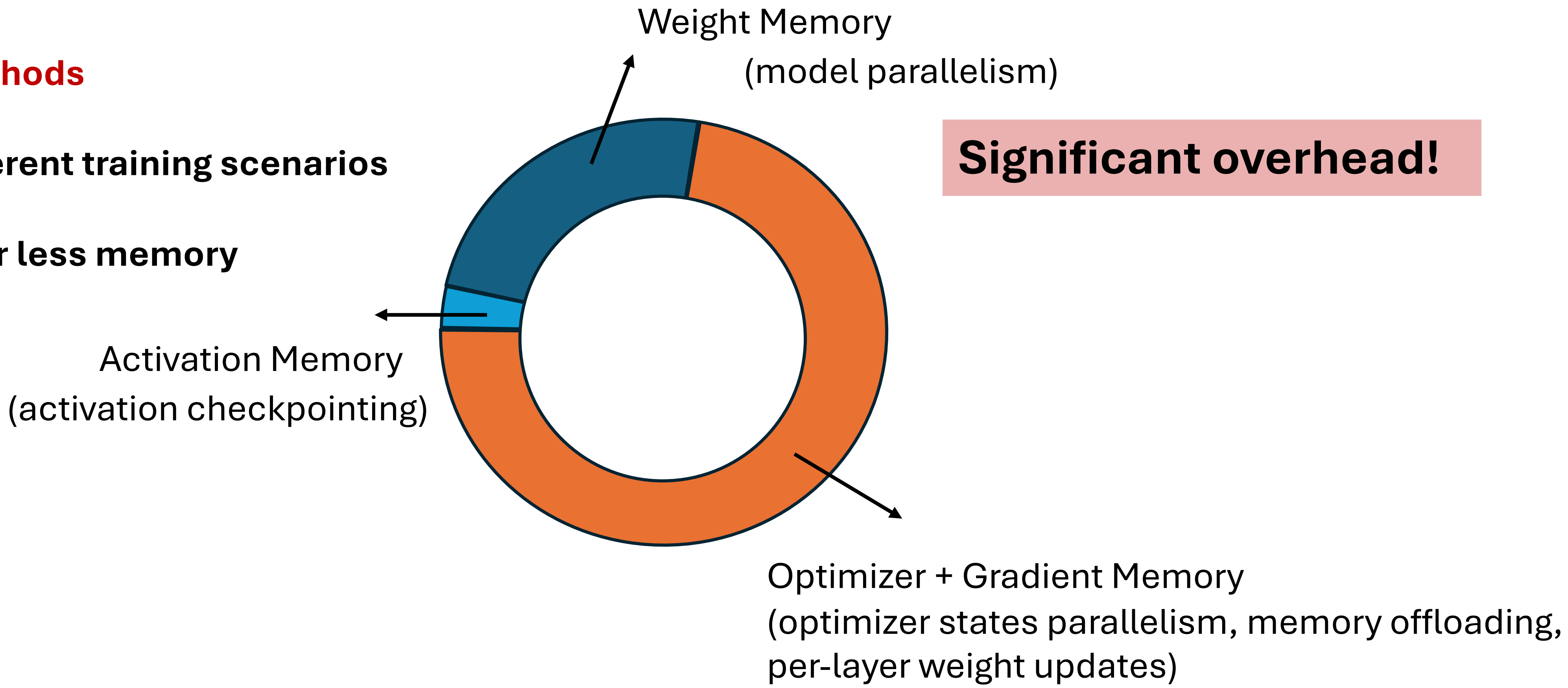


Reducing Memory – System Level

Lossless methods

Work for different training scenarios

Trade time for less memory



Significant overhead!

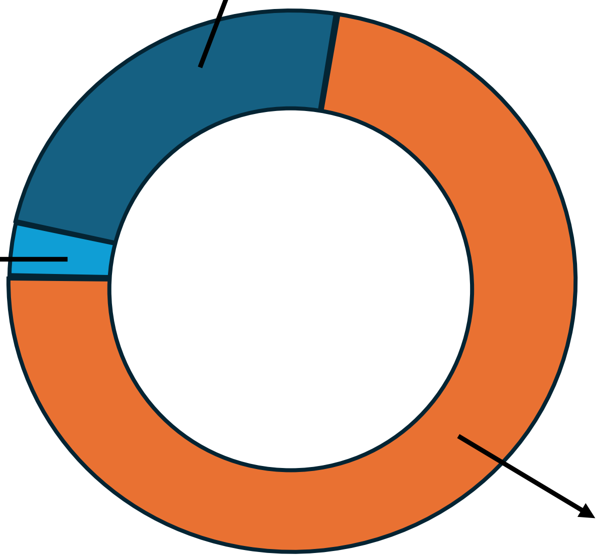
Reducing Memory – System Level

Lossless methods

Work for different training scenarios

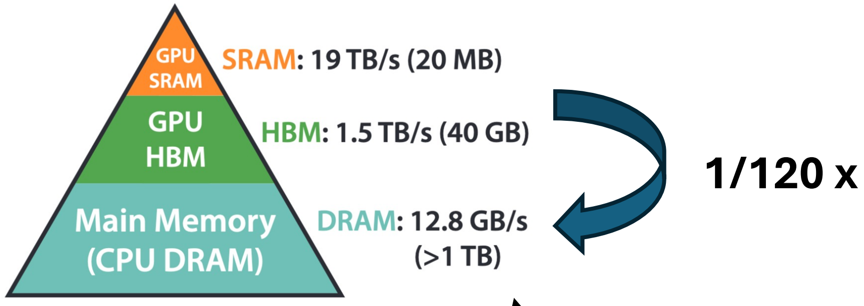
Trade time for less memory

Weight Memory
(model parallelism)



Activation Memory
(activation checkpointing)

Significant overhead!



Memory Hierarchy with Bandwidth & Memory Size

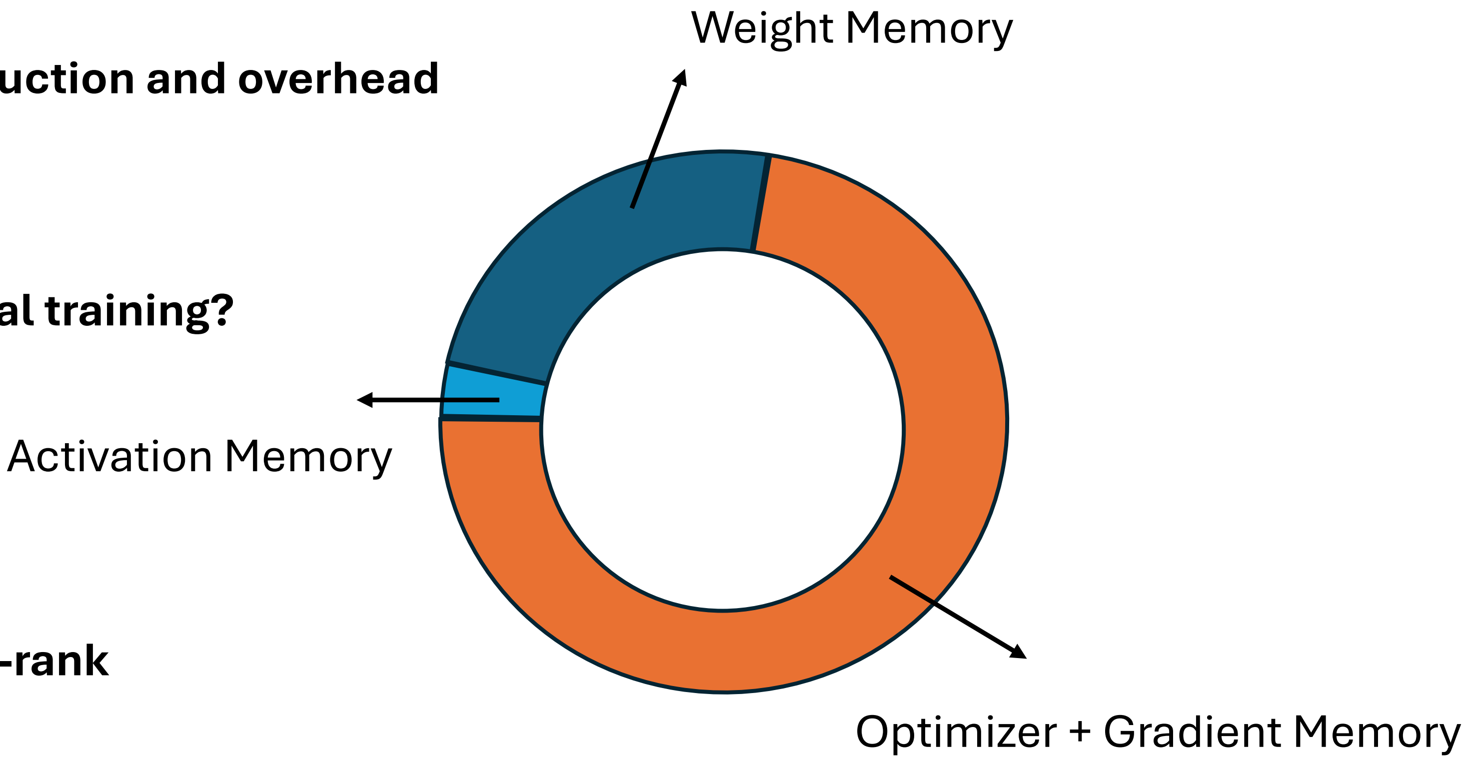
Optimizer + Gradient Memory
(optimizer states parallelism, **memory offloading,** per-layer weight updates)

Reducing Memory – Algorithm Level

Lossy methods

Better trade-off:
between memory reduction and overhead

Task-dependent:
Fine-tuning?
Pre-training / continual training?



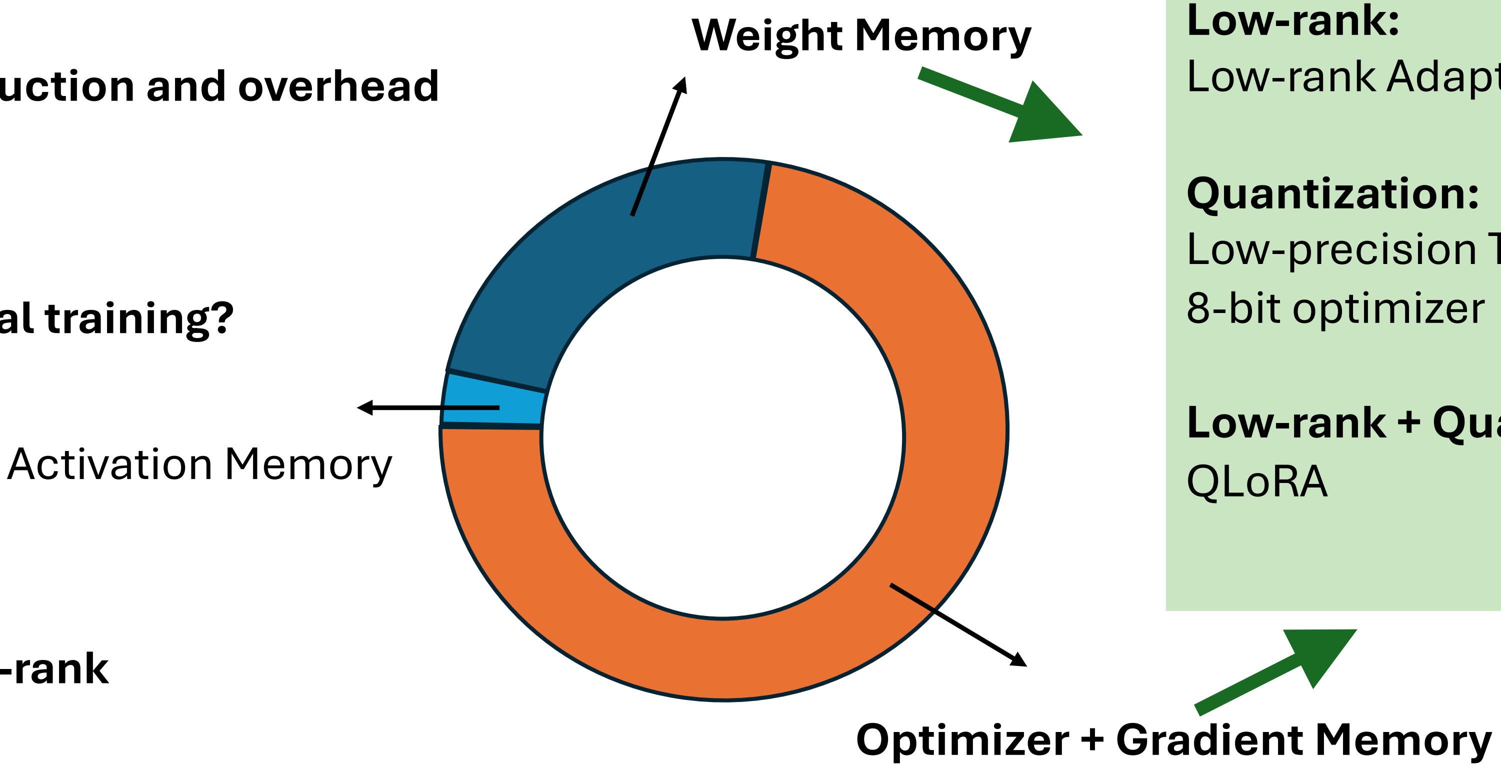
Focus on:
Quantization and low-rank

Reducing Memory – Algorithm Level – Fine-tuning

Lossy methods

Better trade-off:
between memory reduction and overhead

Task-dependent:
Fine-tuning?
Pre-training / continual training?



Focus on:
Quantization and low-rank

Fine-tuning:

Low-rank:
Low-rank Adaptation (LoRA)

Quantization:
Low-precision Training
8-bit optimizer

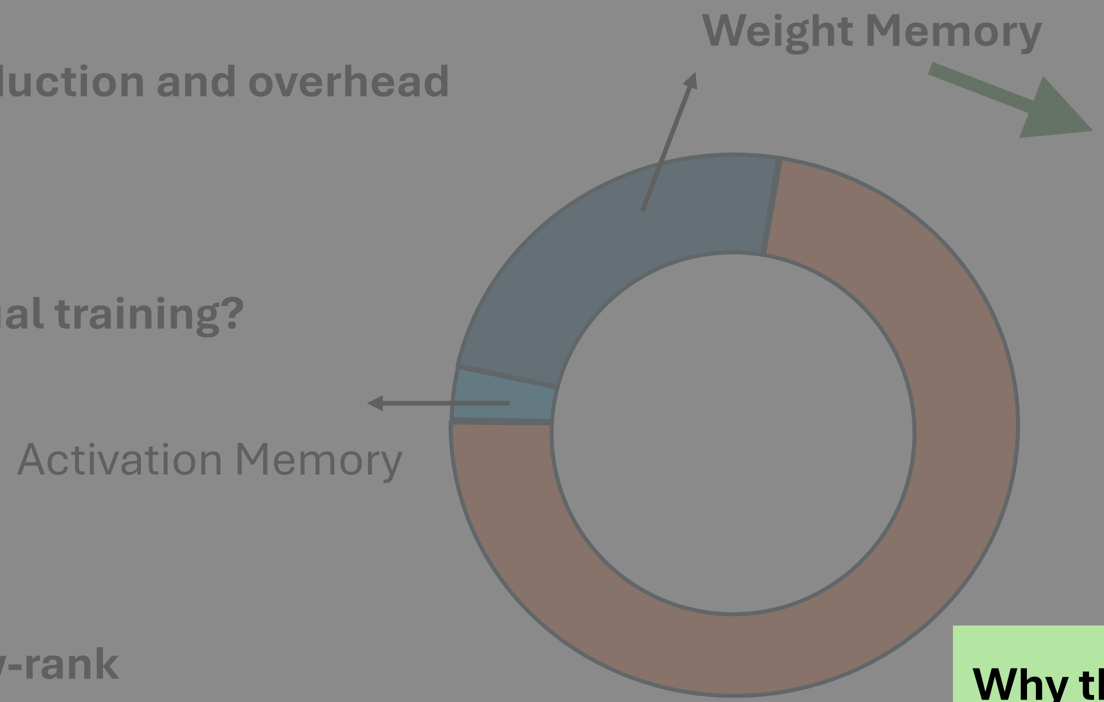
Low-rank + Quantization:
QLoRA

Reducing Memory – Algorithm Level – Fine-tuning

Lossy methods

Better trade-off:
between memory reduction and overhead

Task-dependent:
Fine-tuning?
Pre-training / continual training?



Focus on:
Quantization and low-rank

Fine-tuning:

Low-rank:
Low-rank Adaptation (LoRA)

Quantization:
Low-precision Training
8-bit optimizer

Low-rank + Quantization:
QLoRA

Why they work well in fine-tuning?

“Simple” optimization landscape:
Less sensitive to approximation error

Even lead to better generalization:
Avoid catastrophic forgetting

Reducing Memory – Algorithm Level – Pre-training

How about pre-training?

Complex optimization landscape

Approximation may lead to strong degradation

Pre-training:

Low-rank:
Low-rank Adaptation (LoRA) ✗

Quantization:
Low-precision Training
8-bit optimizer ✓

Low-rank + Quantization:
QLoRA ✗

Fine-tuning:

Low-rank:
Low-rank Adaptation (LoRA)

Quantization:
Low-precision Training
8-bit optimizer

Low-rank + Quantization:
QLoRA

System-level Memory Reduction:
Activation checkpointing
Parallelism
Memory offloading
Per-layer weight updates ✓

Reducing Memory – Algorithm Level – Pre-training

How about pre-training?

Complex optimization landscape

Approximation may lead to strong degradation

Pre-training:

Low-rank:
Low-rank Adaptation (LoRA)

Quantization:
Low-precision Training
8-bit optimizer

Low-rank + Quantization:
QLoRA

Fine-tuning:

Low-rank:
Low-rank Adaptation (LoRA)

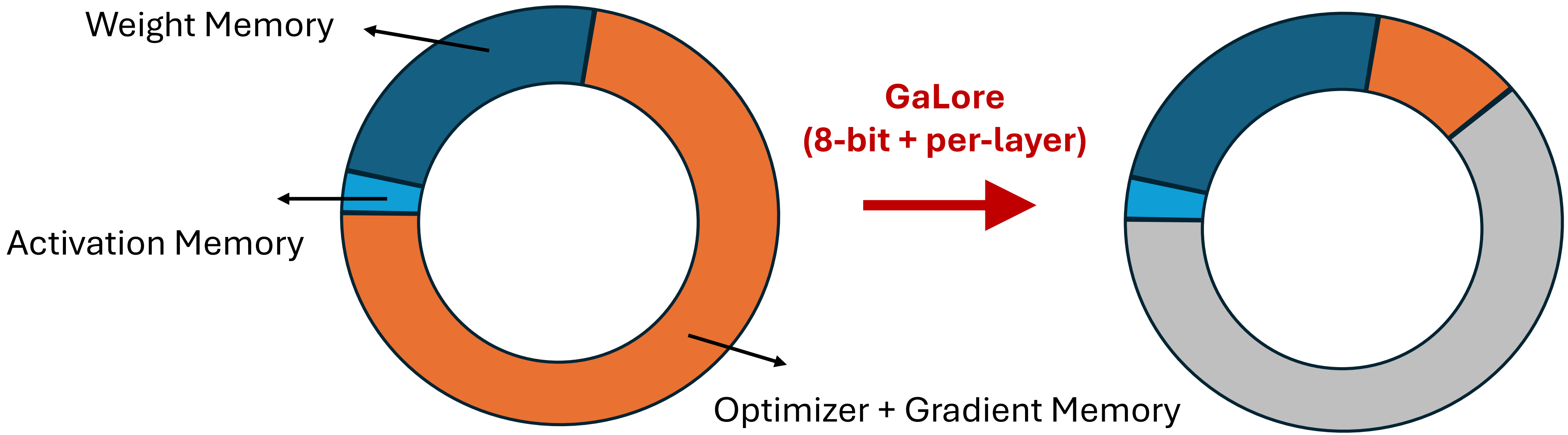
Quantization:
Low-precision Training
8-bit optimizer

Low-rank + Quantization:
QLoRA

Why low-rank adaptation does not work for pre-training?

Any alternative low-rank solutions? **GaLore !**

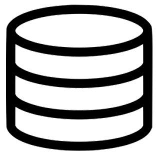
GaLore: Gradient Low-Rank Projection



58 GB **< 24 GB**

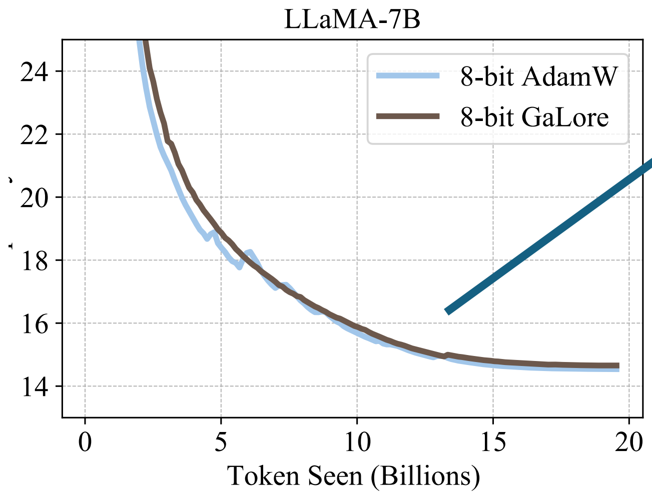
Reduce:
optimizer states and weight gradients

Achieve:
82.5% reduction

 ⊕  ⊕ 

C4 Dataset **LLaMA-7B** **single RTX 4090**

Pre-training - for the first time!



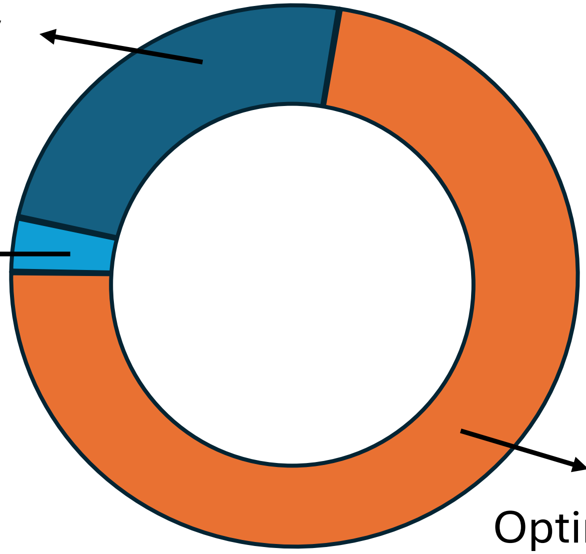
19.7B training tokens

rank= 1024 / 4096

Perplexity:
8-bit GaLore: 14.65
8-bit Adam: 14.61

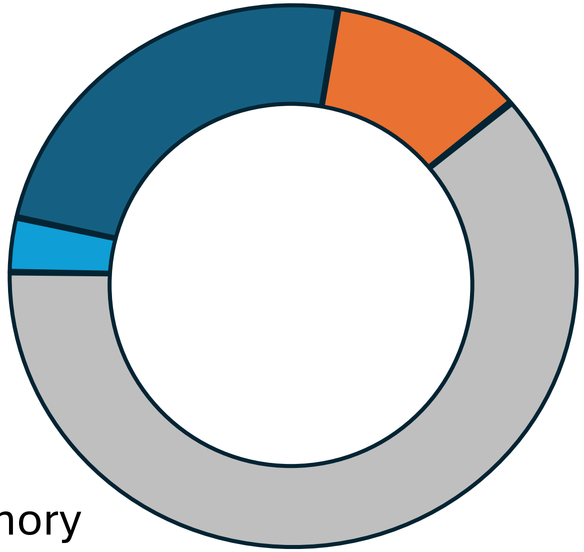
GaLore: Gradient Low-Rank Projection

ICML
International Conference
On Machine Learning



58 GB

GaLore
(8-bit + per-layer)



< 24 GB

Reduce:
optimizer states and weight gradients

Achieve:
82.5% reduction

C4 Dataset +  **LLaMA-7B** + 

single RTX 4090

Pre-training - for the first time!

>1k

10k 1k

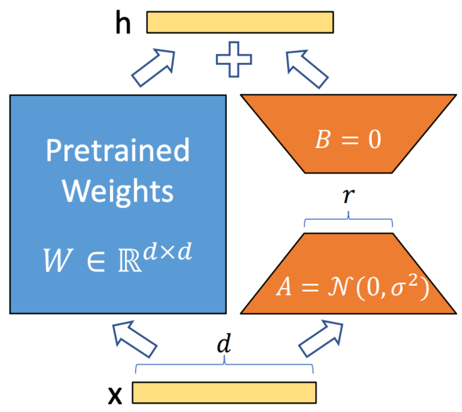
PyTorch

transformers + GaLore

PyTorch Lightning

Colossal-AI

Background: Low-Rank Adaptation (LoRA)



LoRA

Fine-tuning:

$\boldsymbol{W = W^* + AB}$

$\boldsymbol{W^*} \in \mathbb{R}^{\boldsymbol{d \times d}}$ **: fixed pre-trained weights**

$\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{d \times r}}$ $\boldsymbol{B} \in \mathbb{R}^{\boldsymbol{r \times d}}$**: learnable low-rank adaptors**

Pre-training:

$\boldsymbol{W = W_0 + AB}$

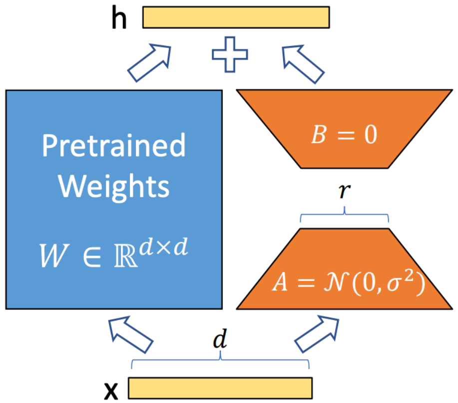
$\boldsymbol{W_0} \in \mathbb{R}^{\boldsymbol{d \times d}}$ **: fixed initial weights**

$\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{d \times r}}$ $\boldsymbol{B} \in \mathbb{R}^{\boldsymbol{r \times d}}$**: learnable low-rank adaptors**

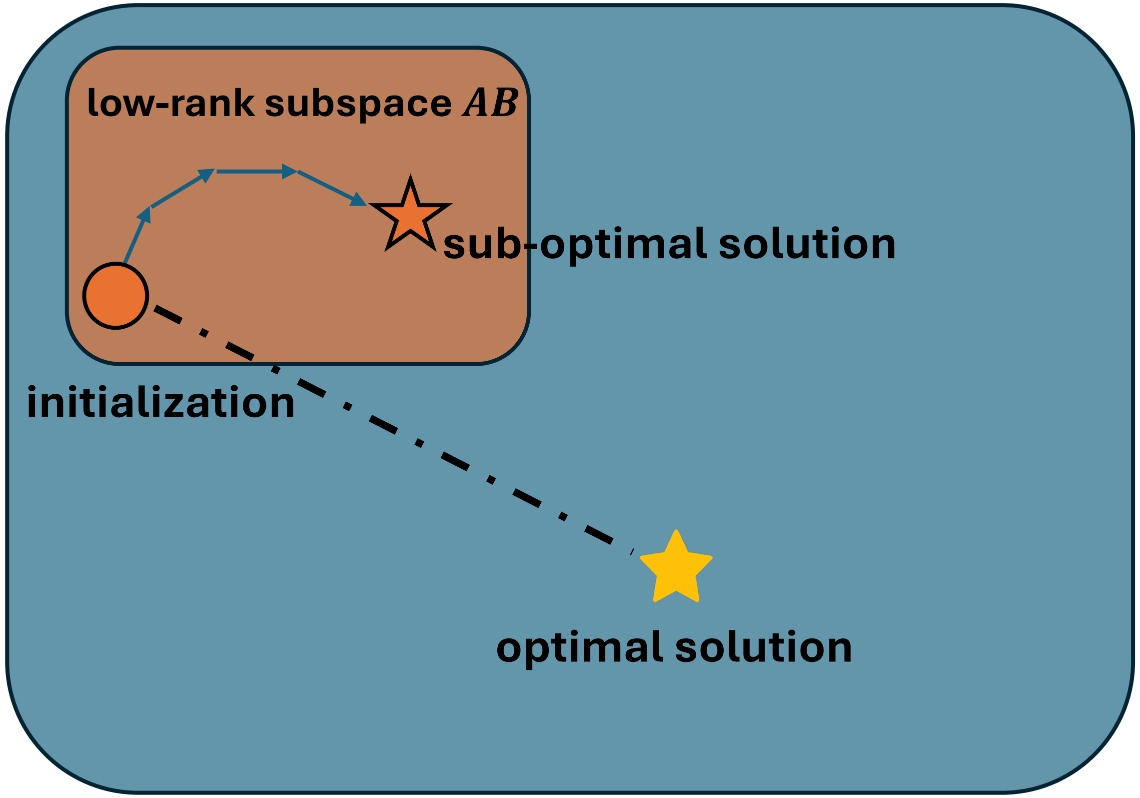
(much larger rank $\boldsymbol{r}$)

Large degradation!

Why does LoRA not work well for pre-training?



$$W = W_0 + AB$$



Complex optimization landscape
(initialization is far from the optimal solution)

Limited expressivity
(optimal solution is outside the low-rank subspace)

Don’t restrict weights in low-rank subspace!
What we need:
A strategy that allows full-parameter training while being memory-efficient

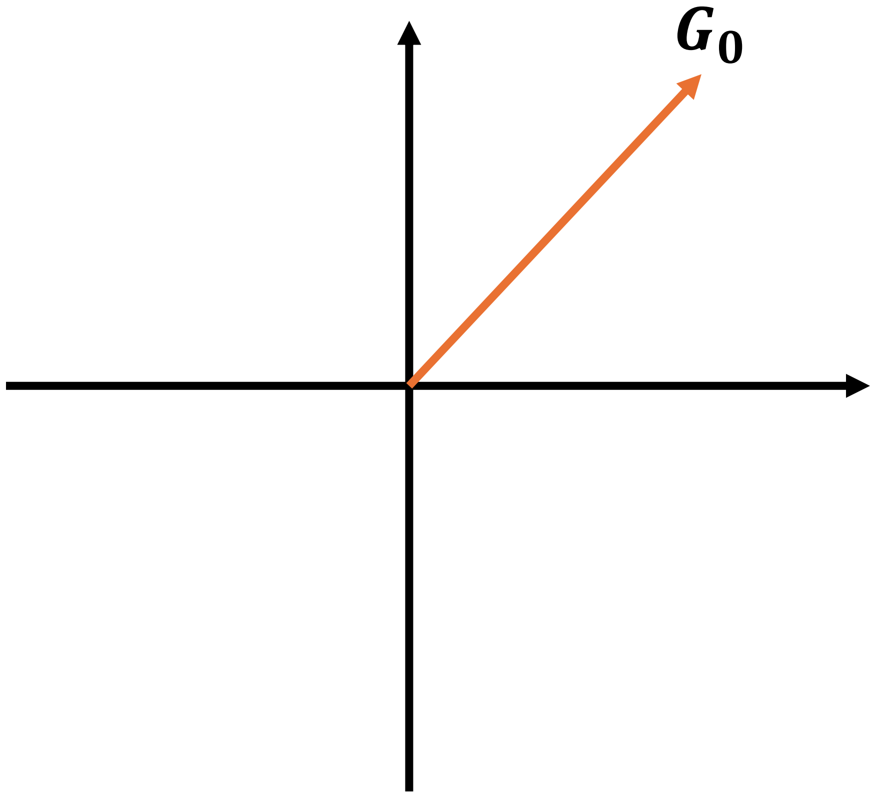
Shift focus: low-rank weights -> low-rank gradients

Weight gradients are low-rank

$G_t \in \mathbb{R}^{d \times d}$ is the gradient of W at step t

$$rank(G_t) \ll d$$

Widely used in practice:
low-rank gradient compression
(PowerSGD)

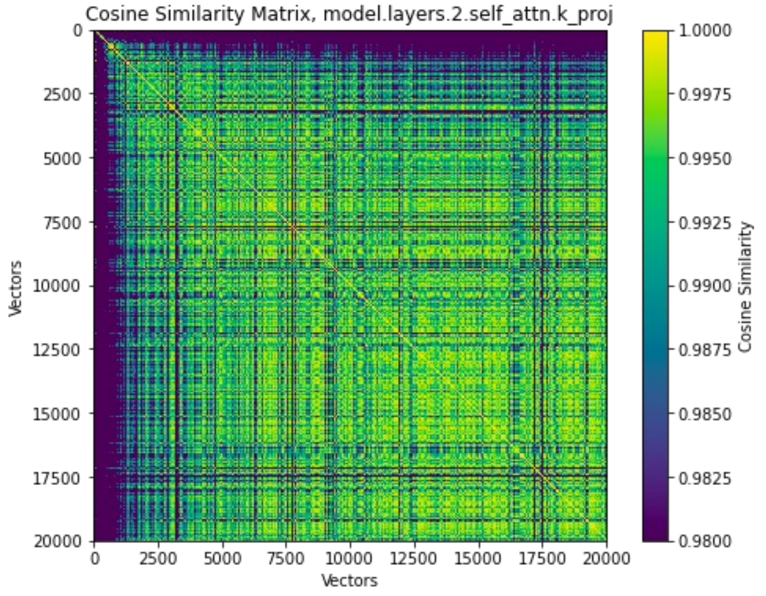


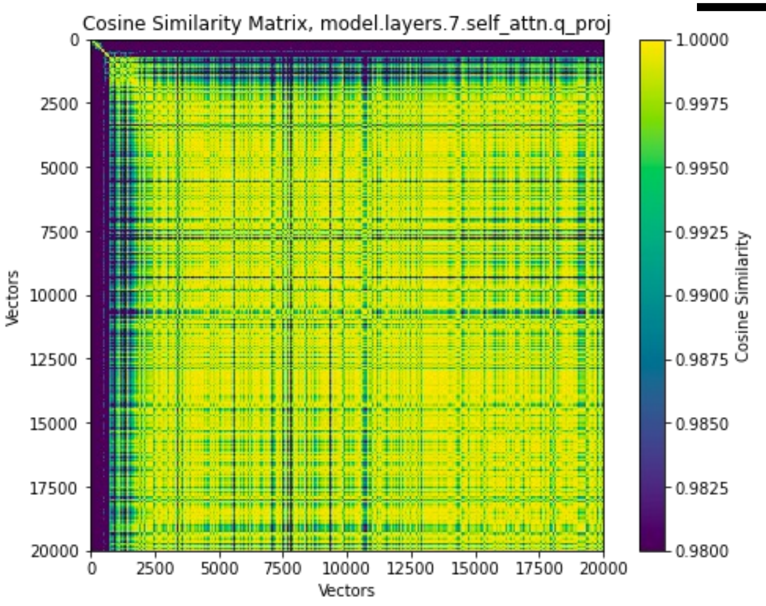
visualize G_t in 2D

Consecutive gradients are similar

$G_0, G_1, G_2, \ldots$ are not only low-rank, but similar to each other:

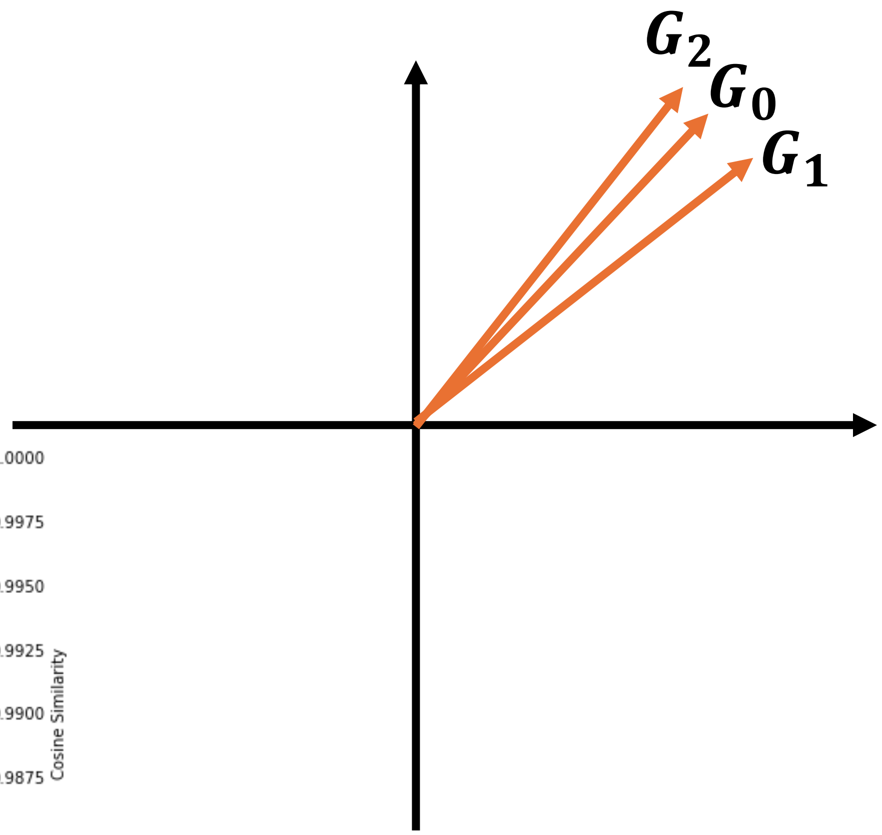
$$G_0 \approx G_1 \approx G_2 \approx \cdots$$





Similarity between gradients G_i and G_j

(for any iterations i and j)

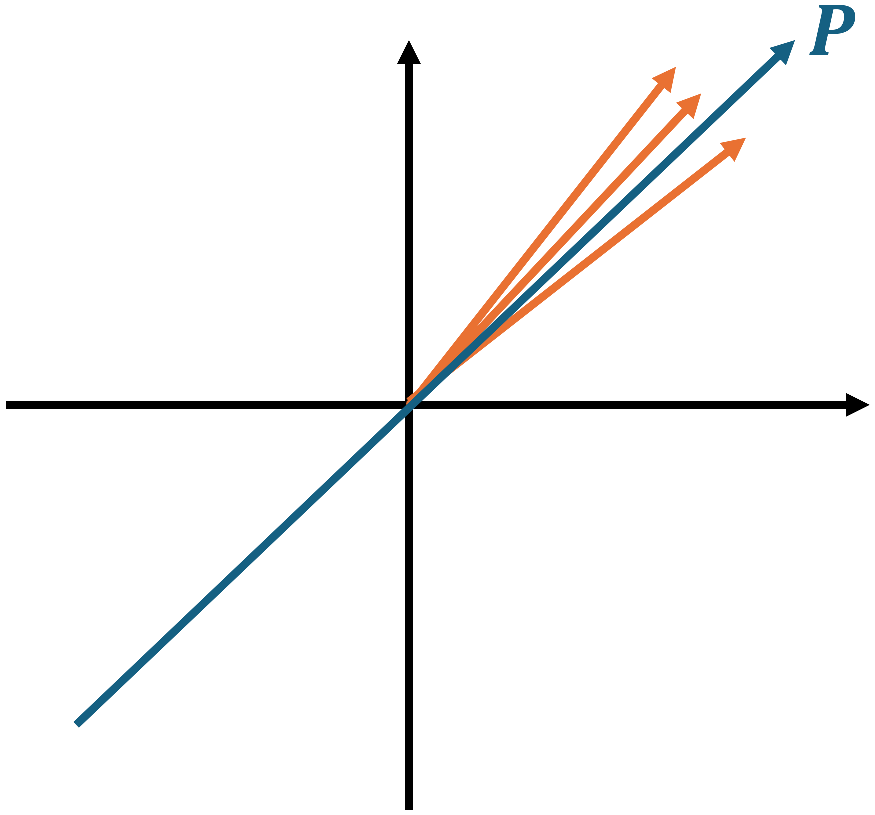


visualize G_t in 2D

Consecutive gradients can be represented in the same subspace P

We can find a subspace P, such that:

$$PP^T(G_0) \approx G_0$$
$$PP^T(G_1) \approx G_1$$
$$PP^T(G_2) \approx G_2$$
$$\dots$$



visualize G_t in 2D

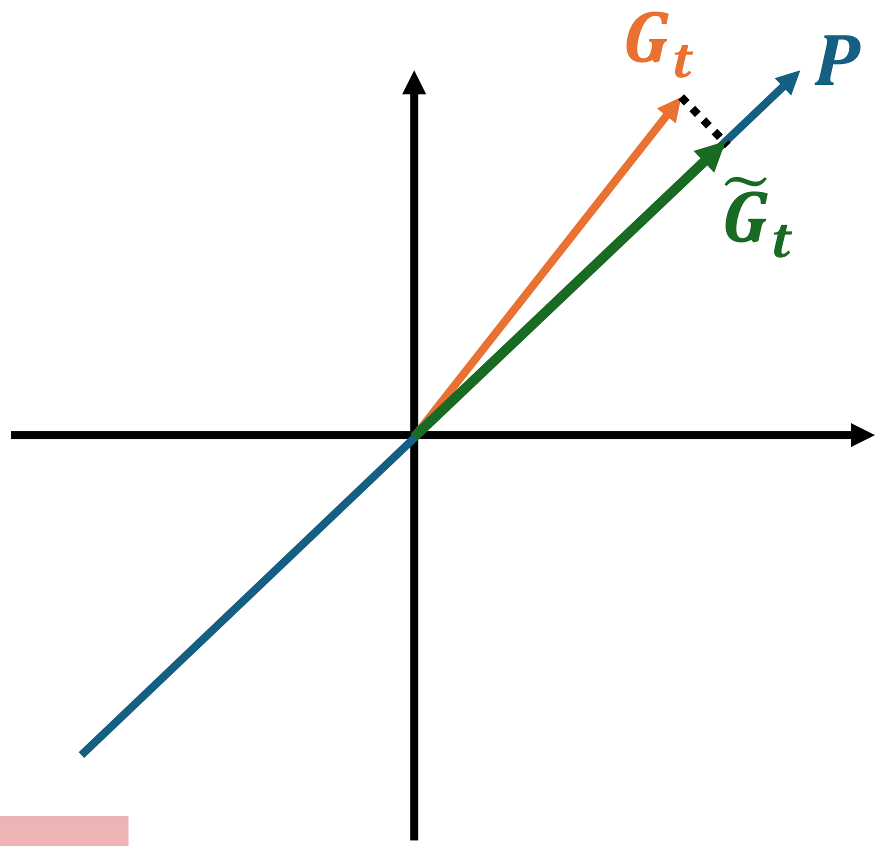
Consecutive gradients can be represented in the same subspace P

We can find a subspace P, such that:

$$PP^T(G_0) \approx G_0$$
$$PP^T(G_1) \approx G_1$$
$$PP^T(G_2) \approx G_2$$
$$\dots$$

Project gradient at iteration t:

$$R_t = P^T(G_t), \widetilde{G}_t = PP^T(G_t)$$



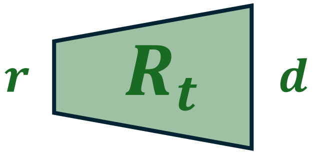
visualize G_t in 2D

2D case:

P^T: 2D $\rightarrow$ 1D $\qquad$ R_t is a scalar

dim = d:

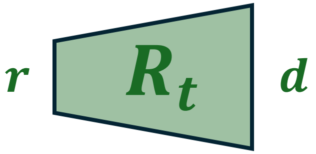
P^T: d $\rightarrow$ r, where r $\ll$ d $\qquad$ $R_t \in \mathbb{R}^{r \times d}$



Why projected gradient is related to memory reduction?

Project gradient at iteration t:

$$R_t = P(G_t), \tilde{G}_t = P^T P(G_t)$$

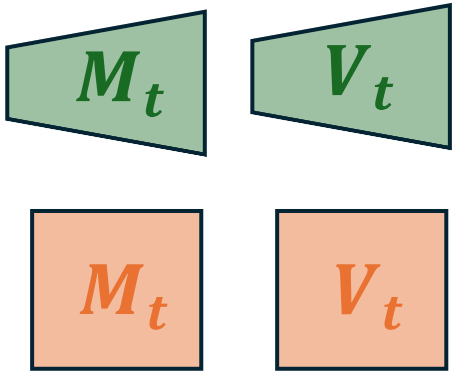


Low-rank gradient R_t provides major information of gradient G_t

For optimizers that require to store optimizer states, such as Adam:

Adam(R_t): low-rank R_t -> low-rank optimizer states M_t, V_t

Adam(G_t): full-rank G_t -> full-rank optimizer states M_t, V_t



Reduce optimizer memory

Preserve true gradient statistics in optimizers

GaLore: Gradient Low-Rank Projection

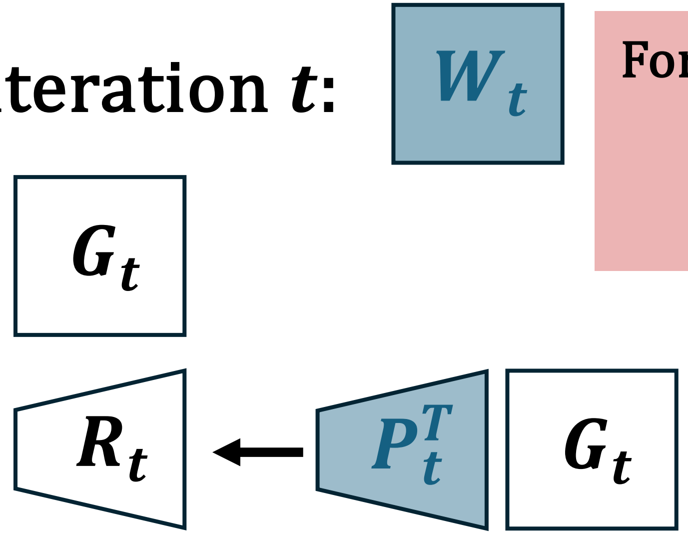
Intermediate
Stored

For any weight matrix W_t at iteration t:

For every T iterations:
Compute and store P_t = SVD(G_t)
P_t is the projection matrix.

Given its gradient matrix G_t

$R_t \leftarrow project(G_t)$



GaLore: Gradient Low-Rank Projection

For any weight matrice W_t at iteration t:

For every T iterations:

Compute and store $P_t = \mathrm{SVD}(G_t)$

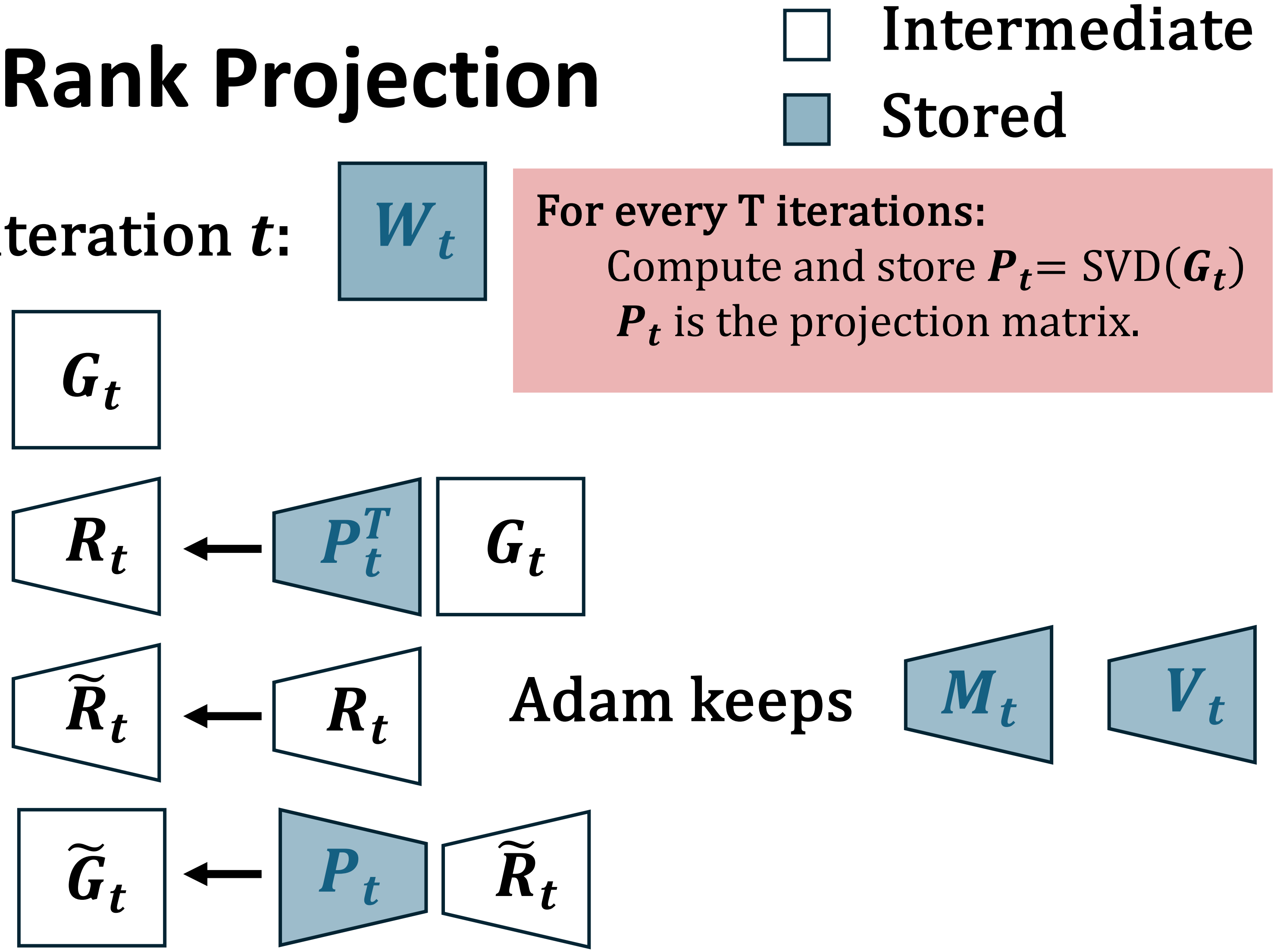
P_t is the projection matrix.

Given its gradient matrix G_t

$R_t \leftarrow project(G_t)$

$\widetilde{R}_t \leftarrow update(R_t)$

$\widetilde{G}_t \leftarrow project\_back(\widetilde{R}_t)$



GaLore: Gradient Low-Rank Projection

Intermediate
Stored

For any weight matrice W_t at iteration t:

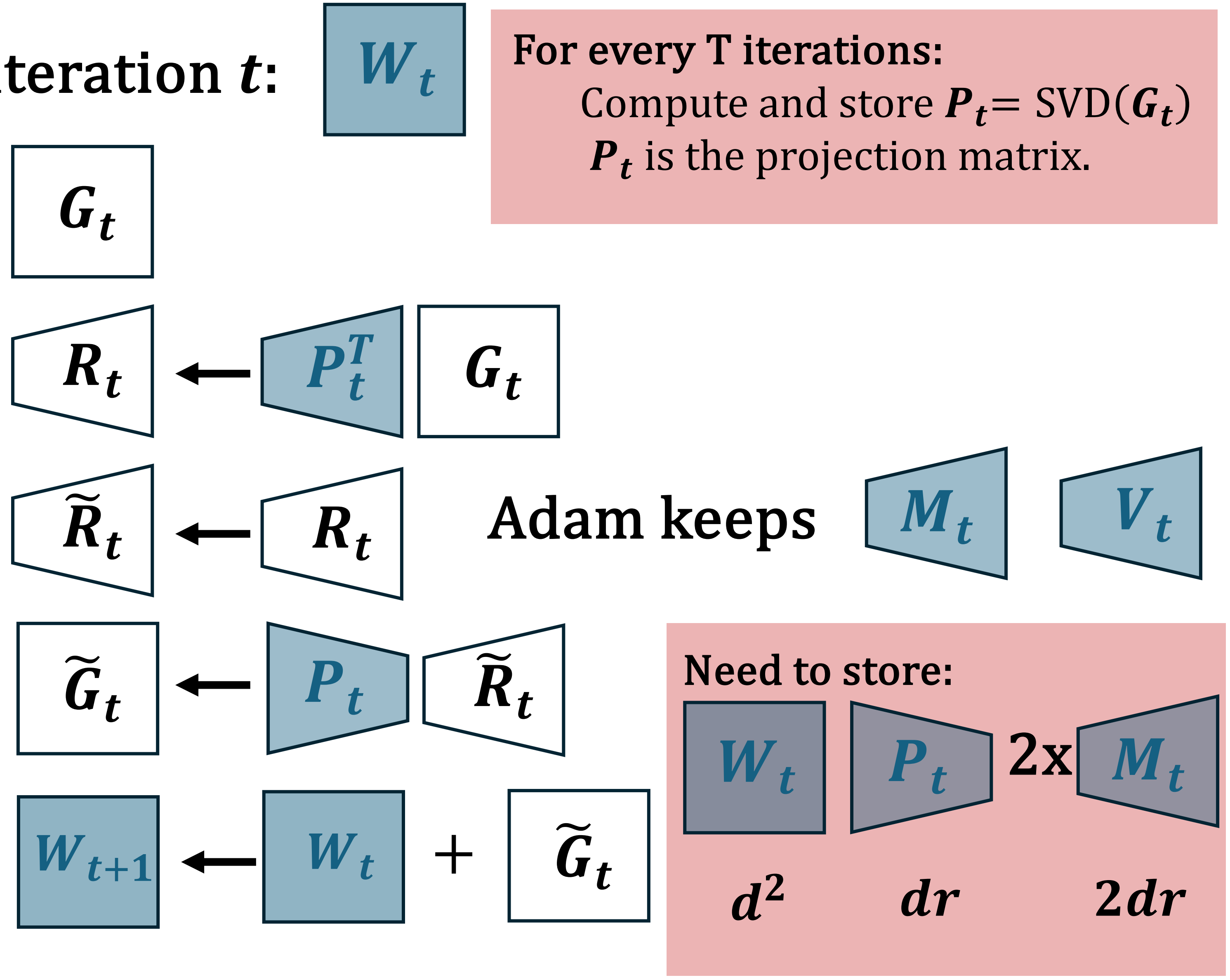
Given its gradient matrix G_t

$R_t \leftarrow project(G_t)$

$\tilde{R}_t \leftarrow update(R_t)$

$\tilde{G}_t \leftarrow project\_back(\tilde{R}_t)$

$W_{t+1} \leftarrow W_t + \tilde{G}_t$



GaLore: Gradient Low-Rank Projection

For any weight matrice W_t at iteration t:

For every T iterations:

Compute and store $P_t = \text{SVD}(G_t)$

P_t is the projection matrix.

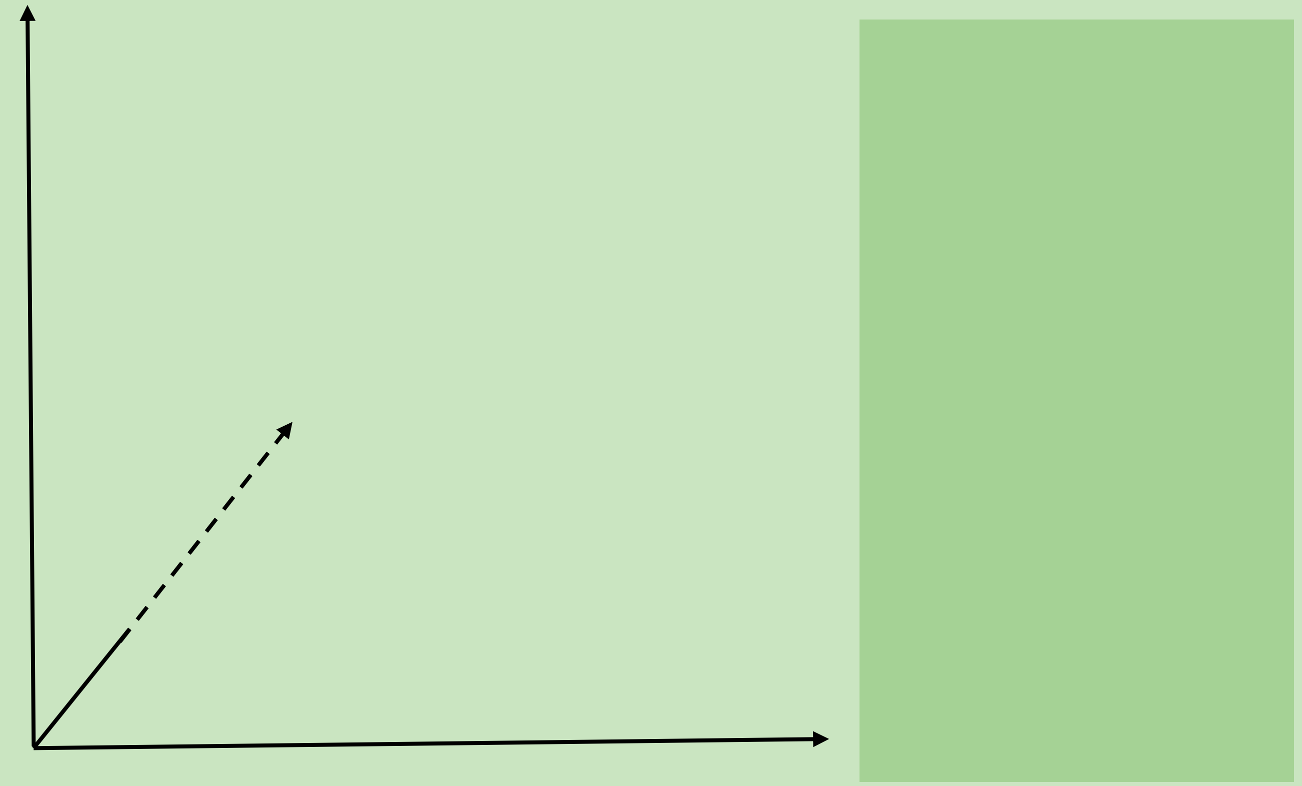
Given its gradient matrix G_t

$R_t \leftarrow project(G_t)$

$\tilde{R}_t \leftarrow update(R_t)$

$\tilde{G}_t \leftarrow project\_back(\tilde{R}_t)$

$W_{t+1} \leftarrow W_t + \tilde{G}_t$



visualize G_t in 3D

GaLore: Gradient Low-Rank Projection

For any weight matrice W_t at iteration t:

Given its gradient matrix G_t

$R_t \leftarrow project(G_t)$

$\tilde{R}_t \leftarrow update(R_t)$

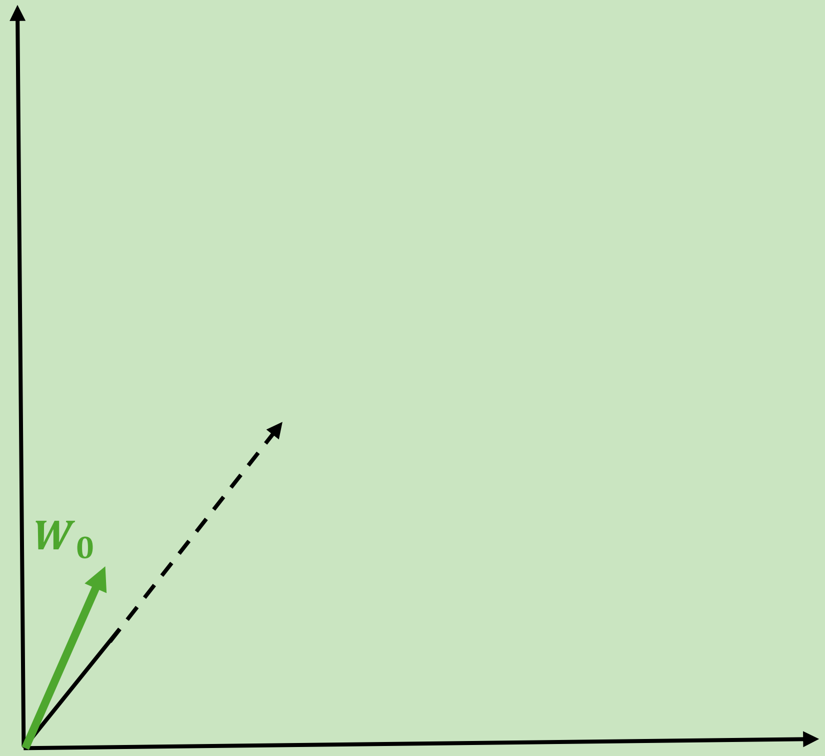
$\tilde{G}_t \leftarrow project\_back(\tilde{R}_t)$

$W_{t+1} \leftarrow W_t + \tilde{G}_t$

For every T iterations:

Compute and store P_t= SVD(G_t)

P_t is the projection matrix.



t=0
Initialize W_0

visualize G_t in 3D

GaLore: Gradient Low-Rank Projection

For any weight matrice W_t at iteration t:

Given its gradient matrix G_t

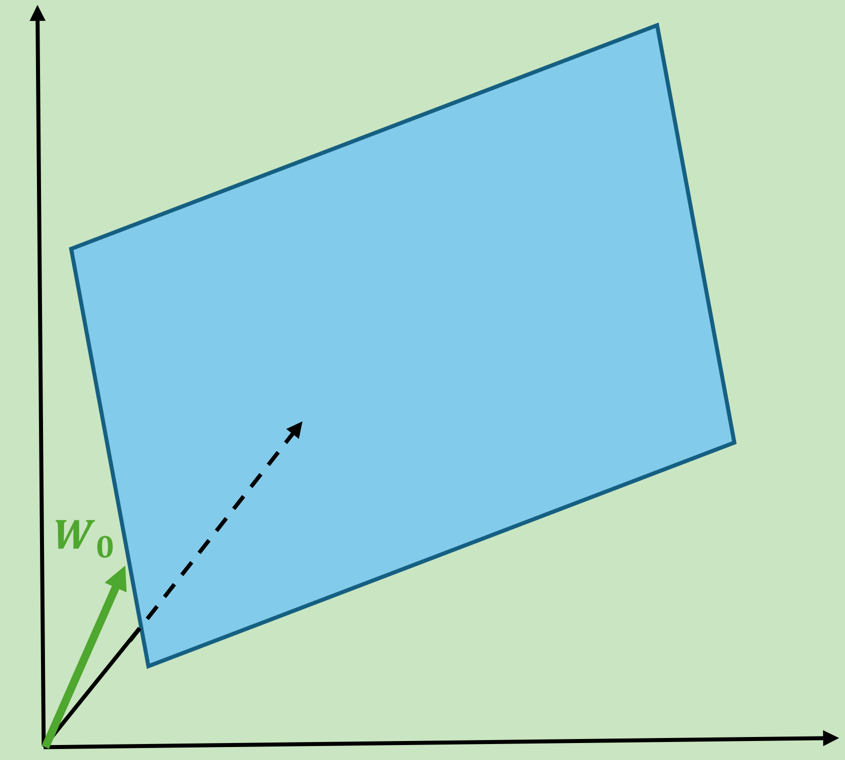
$R_t \leftarrow project(G_t)$

$\widetilde{R}_t \leftarrow update(R_t)$

$\widetilde{G}_t \leftarrow project\_back(\widetilde{R}_t)$

$W_{t+1} \leftarrow W_t + \widetilde{G}_t$

For every T iterations:
Compute and store P_t= SVD(G_t)
P_t is the projection matrix.



t=0
Initialize W_0
Initialize P_0
(first subspace)

visualize G_t in 3D

GaLore: Gradient Low-Rank Projection

For any weight matrice W_t at iteration t:

For every T iterations:
Compute and store $P_t = \mathrm{SVD}(G_t)$
P_t is the projection matrix.

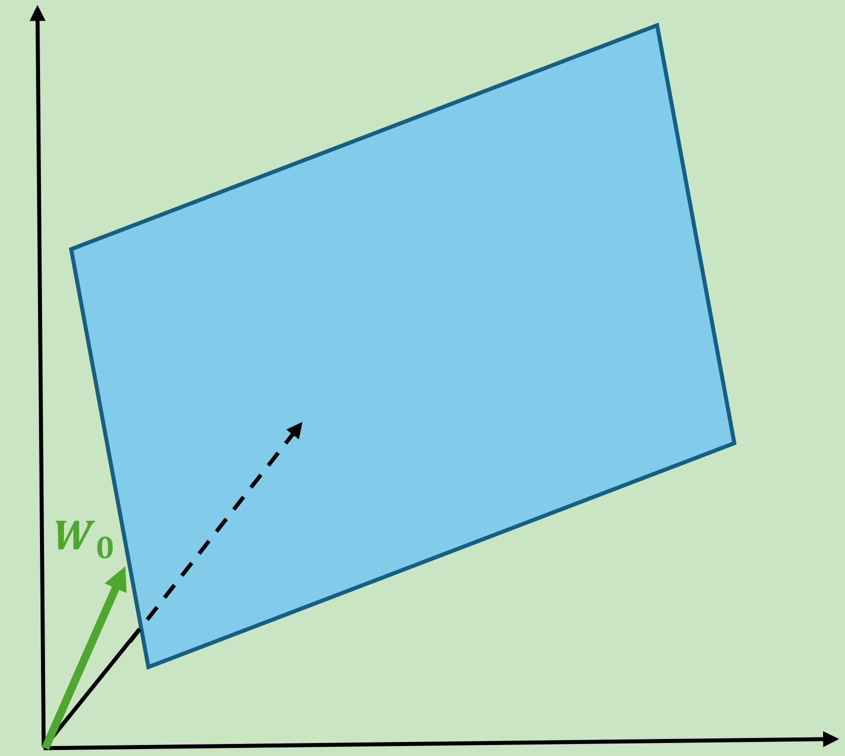
Given its gradient matrix G_t

$$R_t \leftarrow project(G_t)$$

$$\tilde{R}_t \leftarrow update(R_t)$$

$$\tilde{G}_t \leftarrow project\_back(\tilde{R}_t)$$

$$W_{t+1} \leftarrow W_t + \tilde{G}_t$$



$t=0$
Initialize W_0
Initialize P_0
(first subspace)

$t \in (1, \mathrm{T}_1)$
Learning through $\tilde{G}_t$

visualize G_t in 3D

GaLore: Gradient Low-Rank Projection

For any weight matrice W_t at iteration t:

Given its gradient matrix G_t

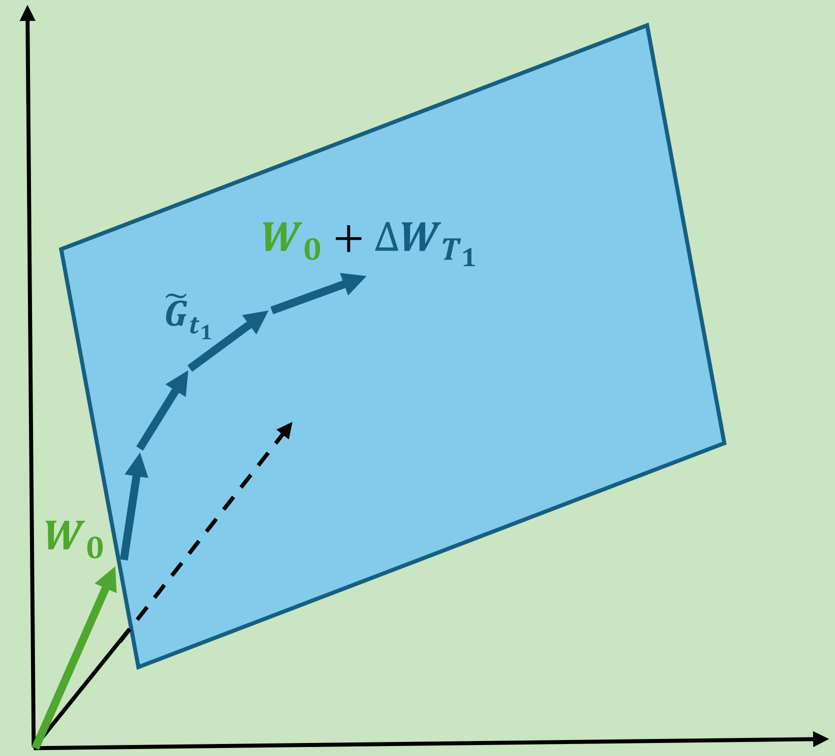
$R_t \leftarrow project(G_t)$

$\tilde{R}_t \leftarrow update(R_t)$

$\tilde{G}_t \leftarrow project\_back(\tilde{R}_t)$

$W_{t+1} \leftarrow W_t + \tilde{G}_t$

For every T iterations:
Compute and store P_t= SVD(G_t)
P_t is the projection matrix.



t=0
Initialize W_0
Initialize P_0
(first subspace)

$t \in (1, T_1)$
Learning through
$\tilde{G}_t$

visualize G_t in 3D

GaLore: Gradient Low-Rank Projection

For any weight matrice W_t at iteration t:

Given its gradient matrix G_t

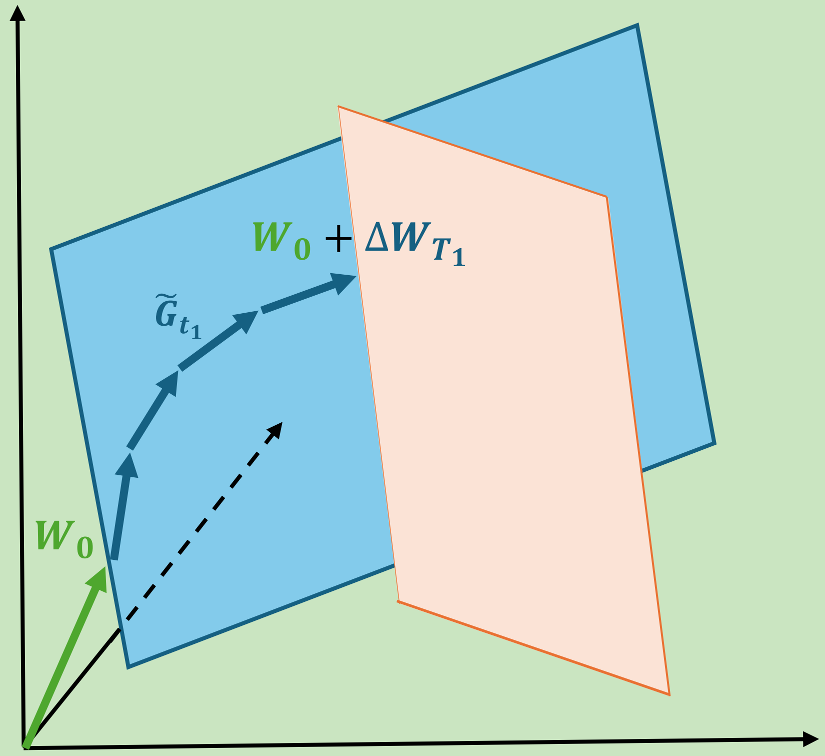
$$R_t \leftarrow project(G_t)$$

$$\tilde{R}_t \leftarrow update(R_t)$$

$$\tilde{G}_t \leftarrow project\_back(\tilde{R}_t)$$

$$W_{t+1} \leftarrow W_t + \tilde{G}_t$$

For every T iterations:
Compute and store P_t= SVD(G_t)
P_t is the projection matrix.



t= T_1 + 1
Compute P_{T_1}
(change subspace)

visualize G_t in 3D

GaLore: Gradient Low-Rank Projection

For any weight matrice W_t at iteration t:

Given its gradient matrix G_t

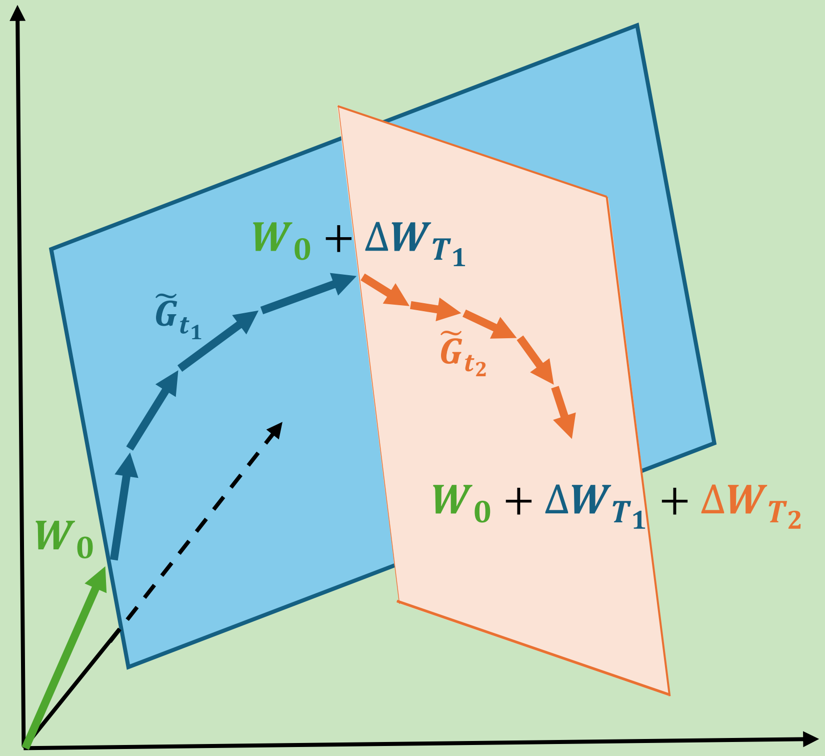
$R_t \leftarrow project(G_t)$

$\tilde{R}_t \leftarrow update(R_t)$

$\tilde{G}_t \leftarrow project\_back(\tilde{R}_t)$

$W_{t+1} \leftarrow W_t + \tilde{G}_t$

For every T iterations:
Compute and store $P_t = \text{SVD}(G_t)$
P_t is the projection matrix.



$t = T_1 + 1$
Compute P_{T_1}
(change subspace)

$t \in (T_1 + 1, T_2)$
$P_t = P_{T_1}$
Learning through
$\tilde{G}_t$

visualize G_t in 3D

GaLore: Gradient Low-Rank Projection

For any weight matrice W_t at iteration t:

Given its gradient matrix G_t

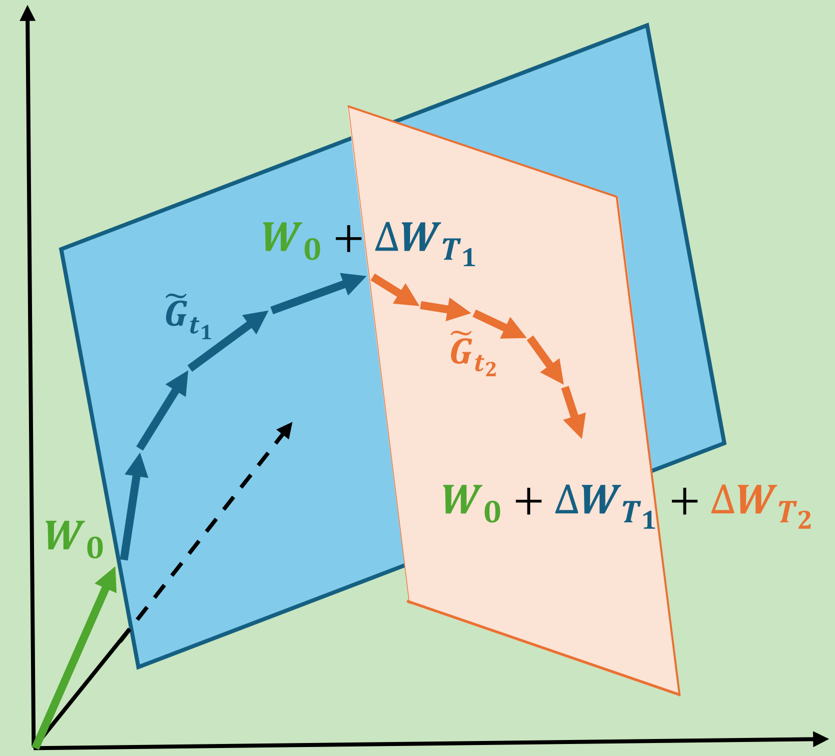
$R_t \leftarrow project(G_t)$

$\tilde{R}_t \leftarrow update(R_t)$

$\tilde{G}_t \leftarrow project\_back(\tilde{R}_t)$

$W_{t+1} \leftarrow W_t + \tilde{G}_t$

For every T iterations:
Compute and store P_t= SVD(G_t)
P_t is the projection matrix.



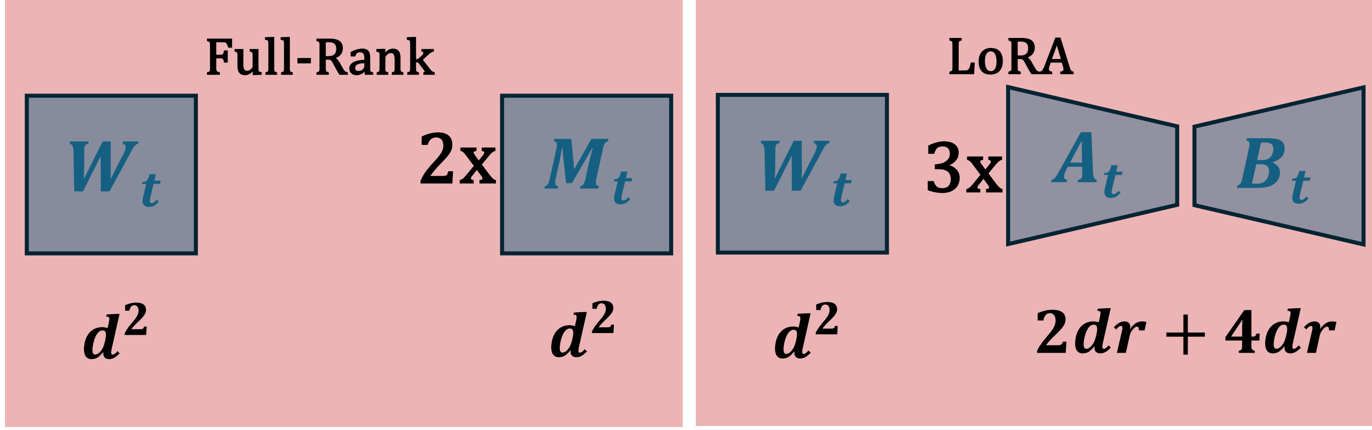
$t = \mathrm{T}_1 + 1$
Compute P_{T_1}
(change subspace)

$t \in (\mathrm{T}_1 + 1, \mathrm{T}_2)$
$P_t = P_{T_1}$
Learning through $\tilde{G}_t$

Continue until convergence

visualize G_t in 3D

GaLore: Memory Usage

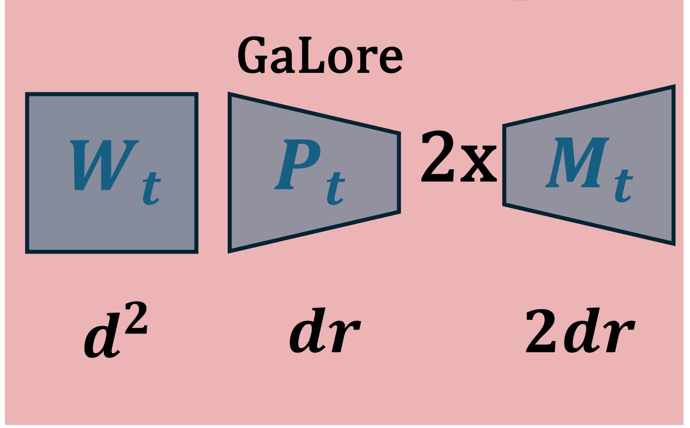


Memory Usage	Weight (W)	Optim States (M_t, V_t)	Projection (P)	Total
Full-rank	d^2	$2d^2$	0	$3d^2$
LoRA	$d^2 + 2dr$	$4dr$	0	$mn + 6dr$
GaLore	d^2	$2dr$	dr	$mn + 3dr$

$\ll dr$, low-bit projection direction does not need to be precise

More memory efficient than LoRA !

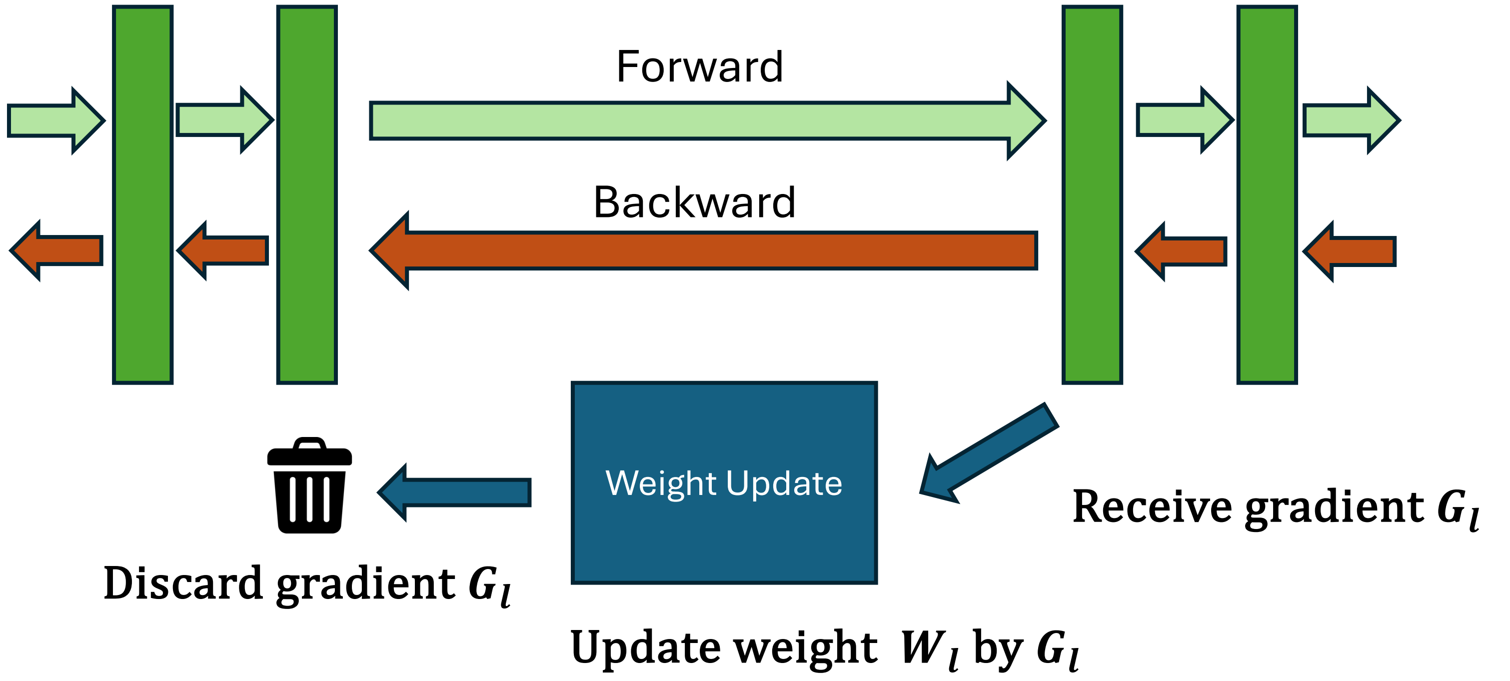
Especially for pre-training, where r is large (1/4 full-rank)



Combing with existing techniques

1. 8-bit Optimizer
   Store optimizer states in 8-bit

2. Per-Layer Weight Update
   Avoid memory for storing full-model weight gradients

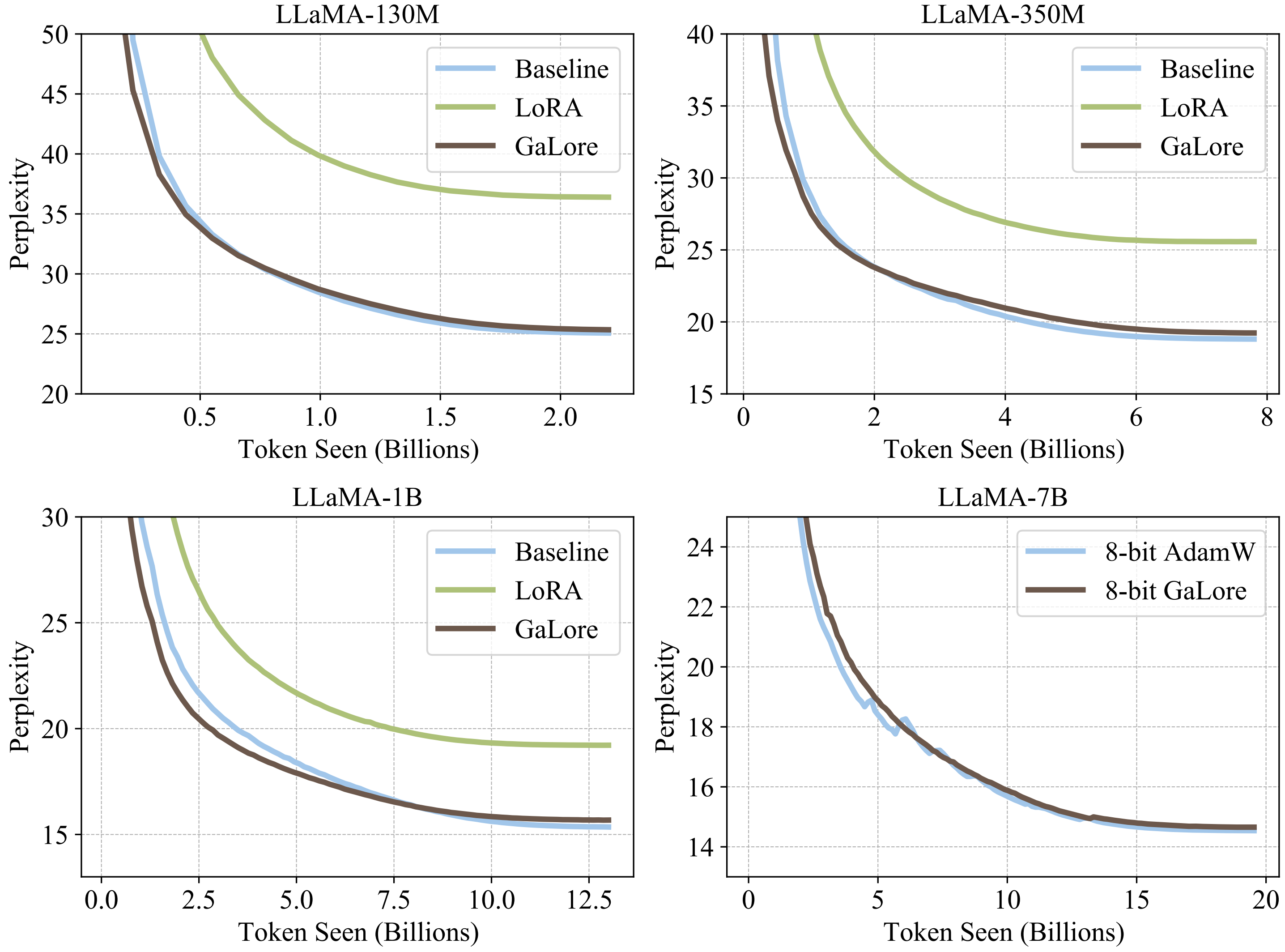


GaLore: Benchmark – Pre-Training on C4

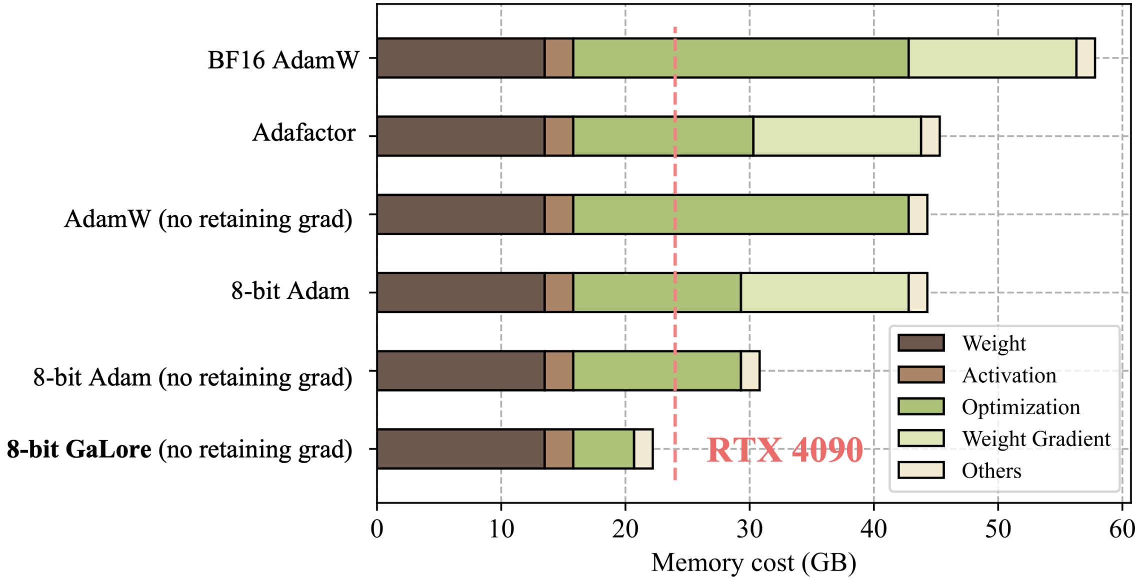
GaLore and LoRA:
rank = 1/4 full-rank

Significantly outperform LoRA!

Match full-rank baseline



GaLore: Benchmark – Efficiency



Memory Comparison

	Rank	Retain grad	Memory	Token/s
8-bit AdamW		Yes	40GB	1434
8-bit GaLore	16	Yes	28GB	1532
8-bit GaLore	128	Yes	29GB	1532
16-bit GaLore	128	Yes	30GB	**1615**
16-bit GaLore	128	No	**18GB**	1587
8-bit GaLore	1024	Yes	36GB	1238

* SVD takes around 10min for 7B model, but runs every T=500-1000 steps.

Throughput Comparison
(Third-Party Evaluation by LLaMA Factory)

$\oplus$

LLaMA-7B **single RTX 4090**

galore-torch

Github: https://github.com/jiaweizzhao/GaLore

galore-torch > 1k

```
from galore_torch import GaLoreAdamW, GaLoreAdamW8bit
# define param groups as galore_params and non_galore_params

param_groups = [{'params': non_galore_params},
                 {'params': galore_params, 'rank': 128, 'update_proj_gap': 200,
                 'scale': 0.25, 'proj_type': 'std'}]

optimizer = GaLoreAdamW(param_groups, lr=0.01)
```

Plug it in!

PyTorch

transformers + GaLore

PyTorch Lightning

Colossal-AI

More integrations are underway!

GaLore: Future Works

- **GaLore-Distributed**:
  - A single RTX 4090 takes **3 months** to fully pre-train a LLaMA 7B model on C4
  - Multiple 4090s are needed
  - Low-bandwidth elastic training
- **Better GaLore**
  - Improve throughput efficiency
  - Further reduce memory overhead

Memory-Efficient Training: Future Direction

- Focus more on challenging tasks:
  - Difficult fine-tuning tasks (e.g., reasoning tasks)
  - Continual training
  - Pre-training
- Consider optimization and memory limitations jointly
  - Studying what are “redundancy” inside training dynamic
- Memory reduction for weight and activation

Thank you!