北京大学
PEKING UNIVERSITY

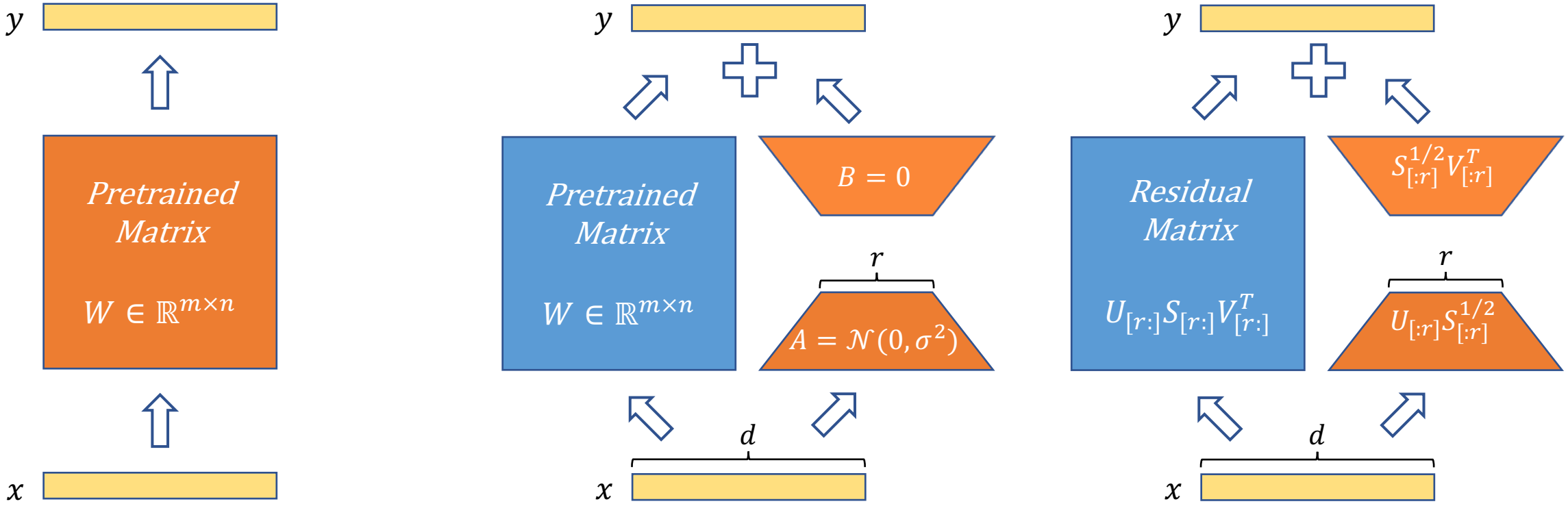
PiSSA: Principal Singular Values and Singular Vectors Adaptation of Large Language Models

Fanxu Meng
Peking University

Parameter-efficient Fine-tuning (PEFT)

北京大学 PEKING UNIVERSITY

- Full Parameters Finetuning finetunes the entire model W.
- LoRA finetunes the low-rank approximations to model changes: $\Delta W = AB$.
- PiSSA finetunes the low-rank approximations of the model $W = AB + W^{res}$.

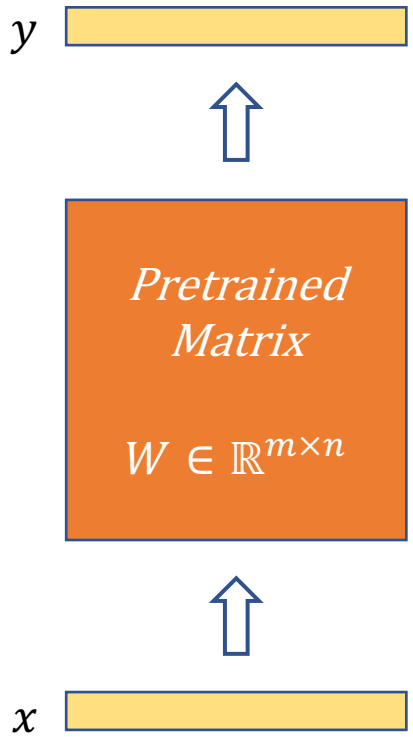


a) Full parameters Fine-tuning

b) LoRA

c) PiSSA

Full parameters Fine-tuning

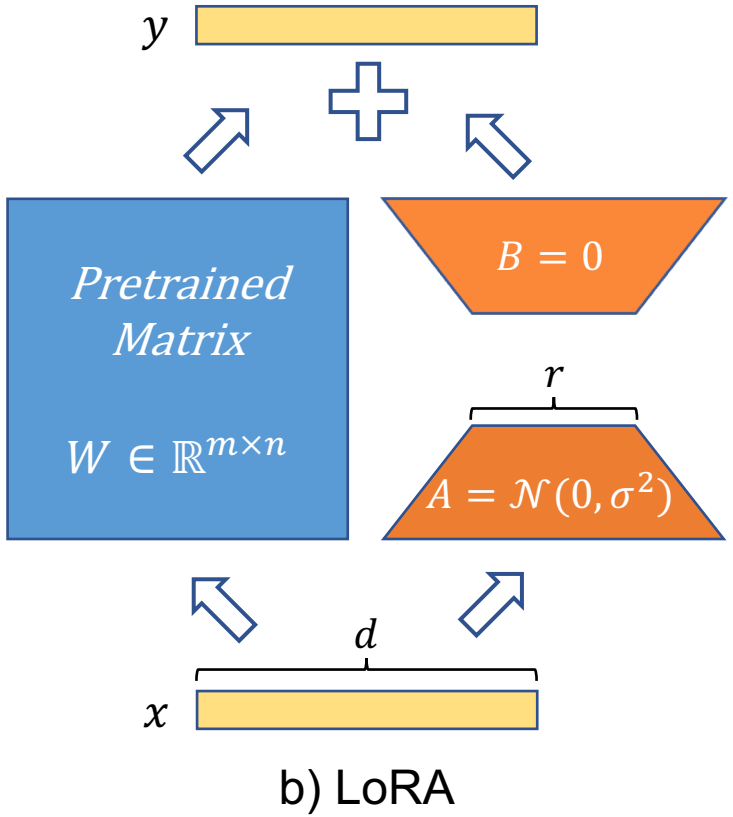


a) Full parameters Fine-tuning

- **Advantage**: straightforward and easy to use.
- **Disadvantage**: requires substantial computational resources.
  - Fine-tuning a LLaMA 65B in 16-bit requires over 780 GB of GPU memory [1].
  - The VRAM consumption for training GPT-3 175B reaches 1.2TB [2].

[1] QLoRA: Efficient Finetuning of Quantized LLMs
[2] LoRA: Low-Rank Adaptation of Large Language Models

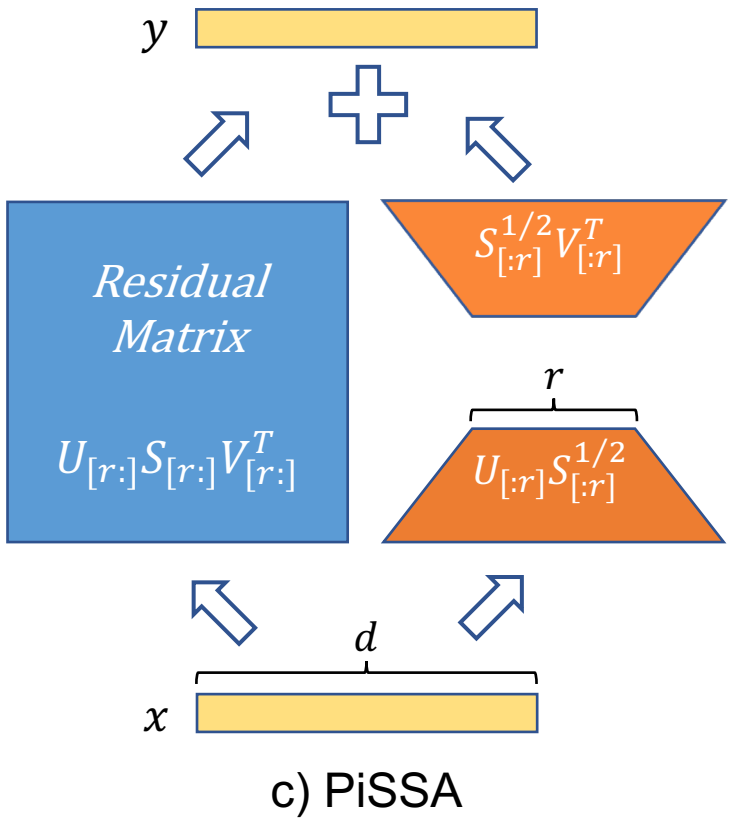
Low-Rank Adaptation (LoRA)



b) LoRA

- The **modifications** to parameters during fine-tuning exhibit **low-rank** properties: $\Delta W = AB$, where $A \in \mathbb{R}^{m\times r}$ and $B \in \mathbb{R}^{r\times n}$, $r \ll \min(m, n)$.
- A random Gaussian initialization is used for A and zero for B, making $\Delta W = AB = 0$.
- **Advantage**: reduce the number of trainable parameters.
- **Disadvantage**: finetune the noise while freezing the W, slow convergence and unsatisfactory performance.

Principal Singular Values and Singular Vectors Adaptation (PiSSA)



c) PiSSA

- The **matrix** W in pre-trained models exhibits **low rank** characteristics: $W = AB + W^{res}$, where

$$A = U_{[:r]}diag\left(S_{[:r]}^{\frac{1}{2}}\right) \in \mathbb{R}^{m\times r},$$

$$B = diag\left(S_{[:r]}^{\frac{1}{2}}\right)V_{[:r]}^T \in \mathbb{R}^{r\times n},$$

$$W^{res} = U_{[r:]}diag\left(S_{[r:]}^{\frac{1}{2}}\right)V_{[r:]}^T = W - AB \in \mathbb{R}^{m\times n}.$$

- **Advantage**: reduce the number of trainable parameters, directly finetune the essential parts of the model while freezing the residual parts, fast convergence and better performance
- **Cost**: taking several seconds for leveraging fast SVD before finetuning.

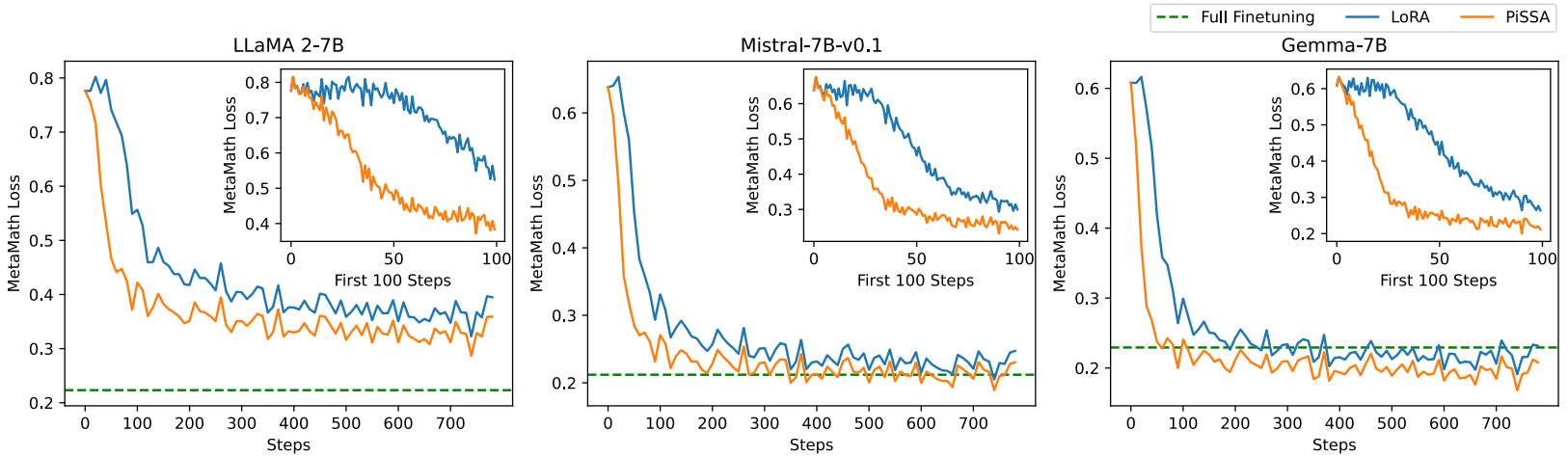
Outperforms LoRA on Five Benchmarks

北京大学 PEKING UNIVERSITY

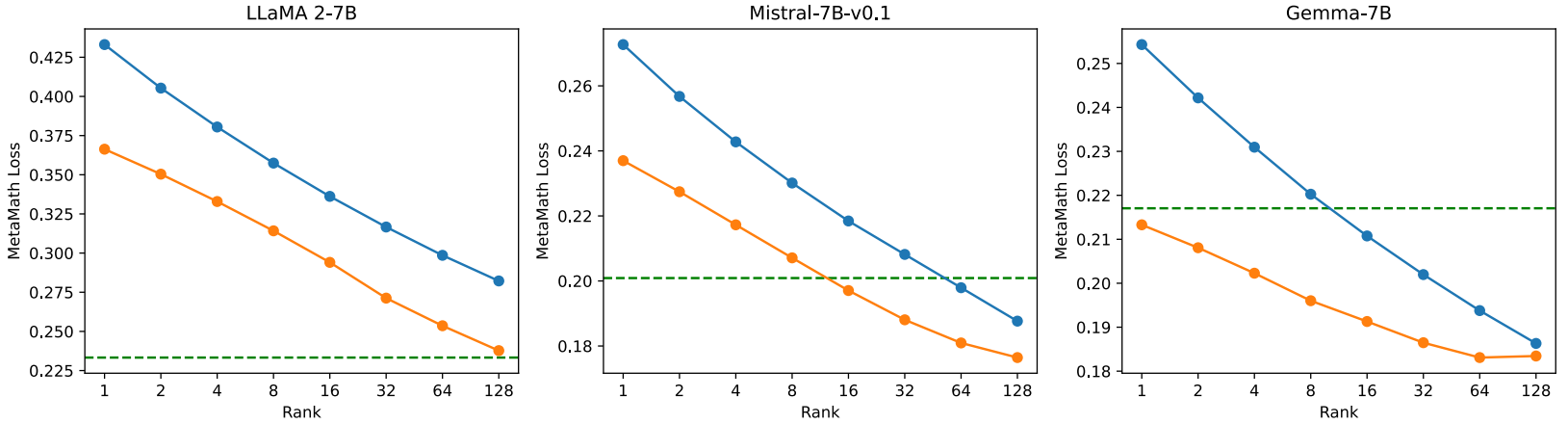
Performance of different models on various tasks

Model	Strategy	Trainable Parameters	GSM8K	MATH	HumanEval	MBPP	MT-Bench
LLaMA 2-7B	Full Fine-tuning	6738M	49.05	7.22	21.34	35.59	**4.91**
	LoRA	320M	42.3	5.5	18.29	35.34	4.58
	PiSSA	320M	**53.07**	**7.44**	**21.95**	**37.09**	4.87
Mistral-7B	Full Fine-tuning	7242M	67.02	18.6	45.12	51.38	4.95
	LoRA	168M	67.7	19.68	43.9	58.39	4.9
	PiSSA	168M	**72.86**	**21.54**	**46.95**	**62.66**	**5.34**
Gemma-7B	Full Fine-tuning	8538M	71.34	22.74	46.95	55.64	5.4
	LoRA	200M	74.9	31.28	53.66	65.41	4.98
	PiSSA	200M	**77.94**	**31.94**	**54.27**	**66.17**	**5.64**

Fast Convergence and Low Final Loss

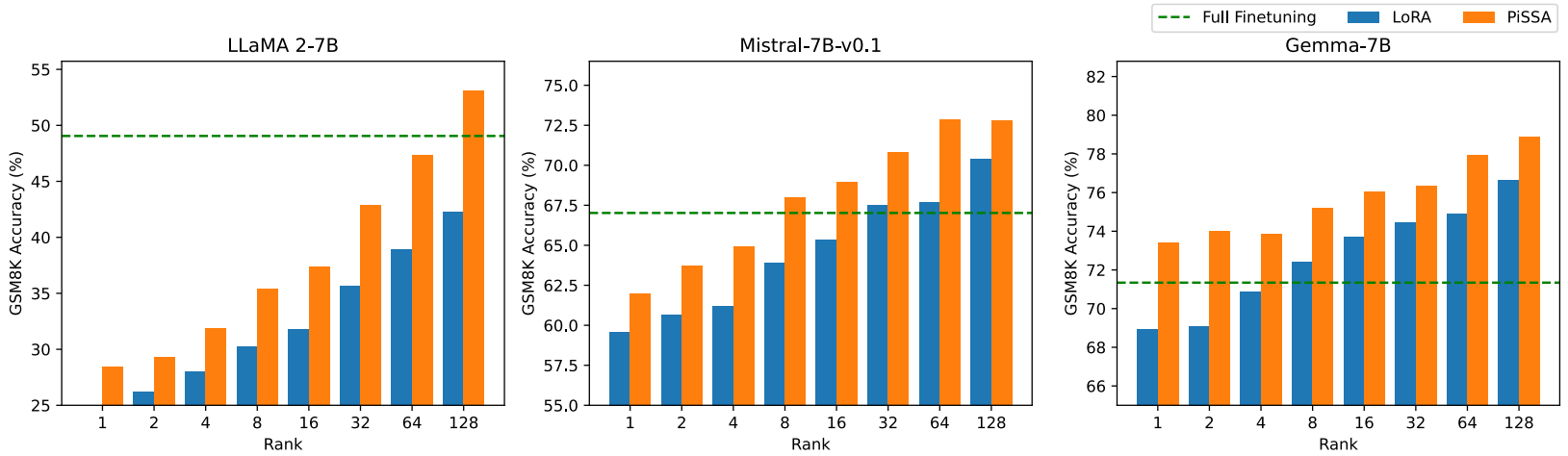


Variation of loss with respect to rank 1 throughout the training phase.

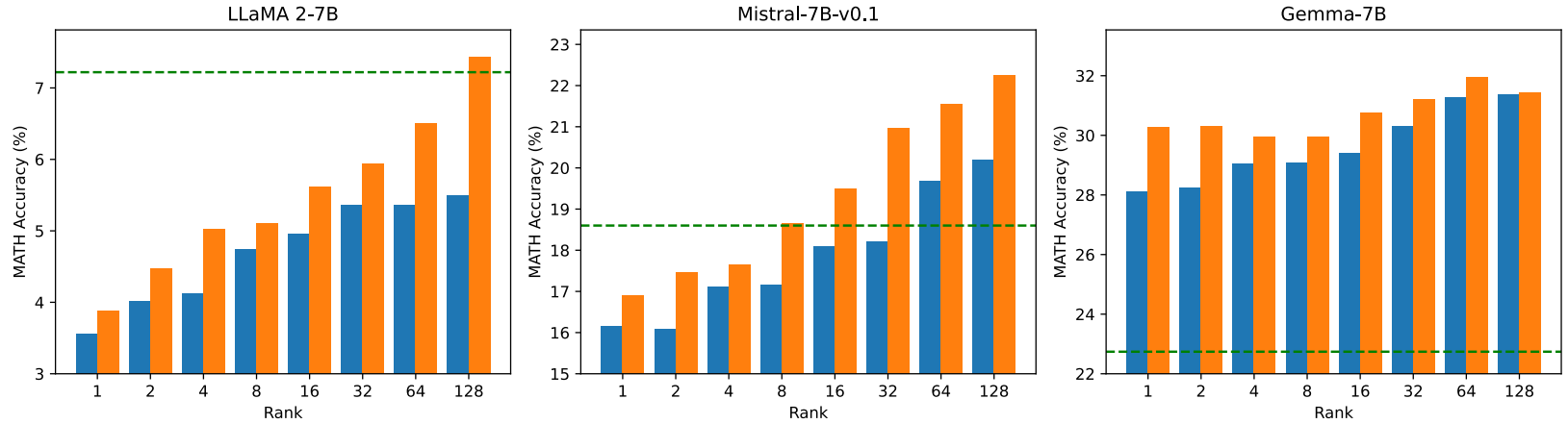


Final training loss across different ranks.

Better Evaluation Performance

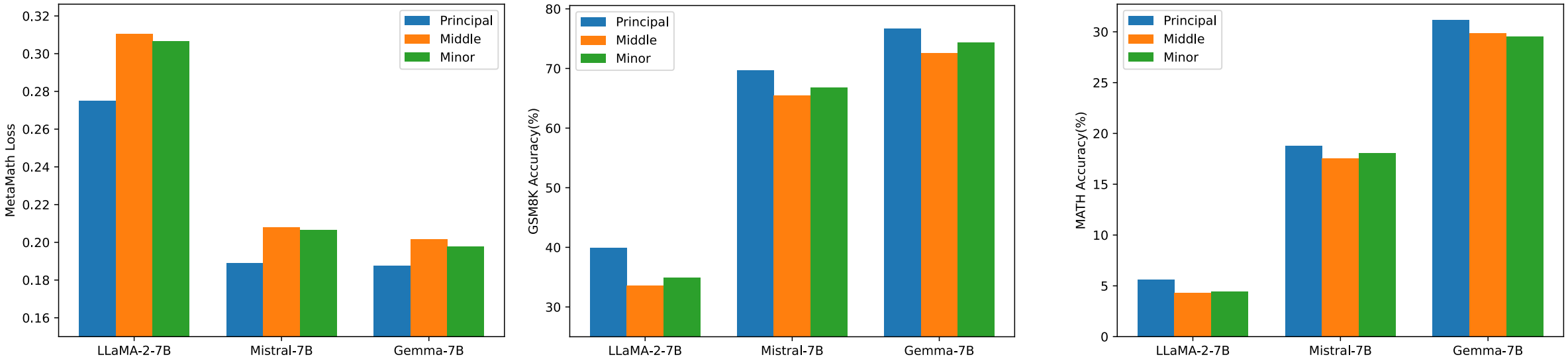


Rank-wise performance evaluated using pass@1 on the GSM8K dataset.



Rank-wise performance evaluated using pass@1 on the MATH dataset.

Principal singular values are important



Initializing with principal, medium, and minor singular values and vectors, the training loss on the MetaMathQA and the accuracy on the GSM8K and MATH validation sets are reported, respectively, for three models.

Takes Only a Few Seconds Using Fast SVD

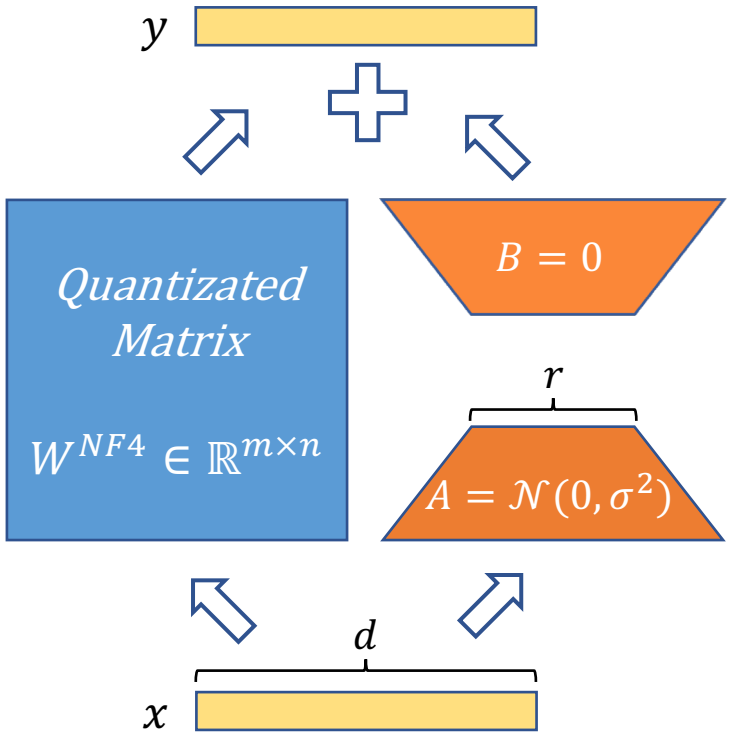
北京大学 PEKING UNIVERSITY

Metric	Niter	1	2	4	8	16	32	64	128
Initialization Time	1	5.05	8.75	5.07	8.42	5.55	8.47	6.80	11.89
	2	4.38	4.71	4.79	4.84	5.06	5.79	7.70	16.75
	4	5.16	4.73	5.09	5.16	5.60	7.01	7.90	11.41
	8	4.72	5.11	5.14	5.40	5.94	7.80	10.09	14.81
	16	6.24	6.57	6.80	7.04	7.66	9.99	14.59	22.67
	∞	434.92	434.15	434.30	435.42	435.25	437.22	434.48	435.84
Initialization Error	1	1.3E-3	1.3E-3	1.5E-3	1.9E-3	1.9E-3	1.9E-3	2.0E-3	1.9E-3
	2	5.8E-4	1.2E-3	1.4E-3	1.4E-3	1.4E-3	1.5E-3	1.4E-3	1.3E-3
	4	6.0E-4	8.7E-4	6.7E-4	1.1E-3	1.0E-3	1.0E-3	1.0E-3	9.7E-4
	8	1.2E-4	2.3E-4	5.2E-4	7.2E-4	5.7E-4	8.2E-4	8.2E-4	7.7E-4
	16	7.9E-5	2.2E-4	1.2E-4	6.5E-4	4.2E-4	6.5E-4	6.0E-4	4.7E-4
	∞	--	--	--	--	--	--	--	--
Training Loss	1	0.3629	0.3420	0.3237	0.3044	0.2855	0.2657	0.2468	0.2301
	2	0.3467	0.3337	0.3172	0.2984	0.2795	0.2610	0.2435	0.2282
	4	0.3445	0.3294	0.3134	0.2958	0.2761	0.2581	0.2414	0.2271
	8	0.3425	0.3279	0.3122	0.2950	**0.2753**	0.2571	0.2406	0.2267
	16	0.3413	0.3275	**0.3116**	0.2946	0.2762	0.2565	0.2405	0.2266
	∞	**0.3412**	**0.3269**	**0.3116**	**0.2945**	0.2762	**0.2564**	**0.2403**	**0.2264**

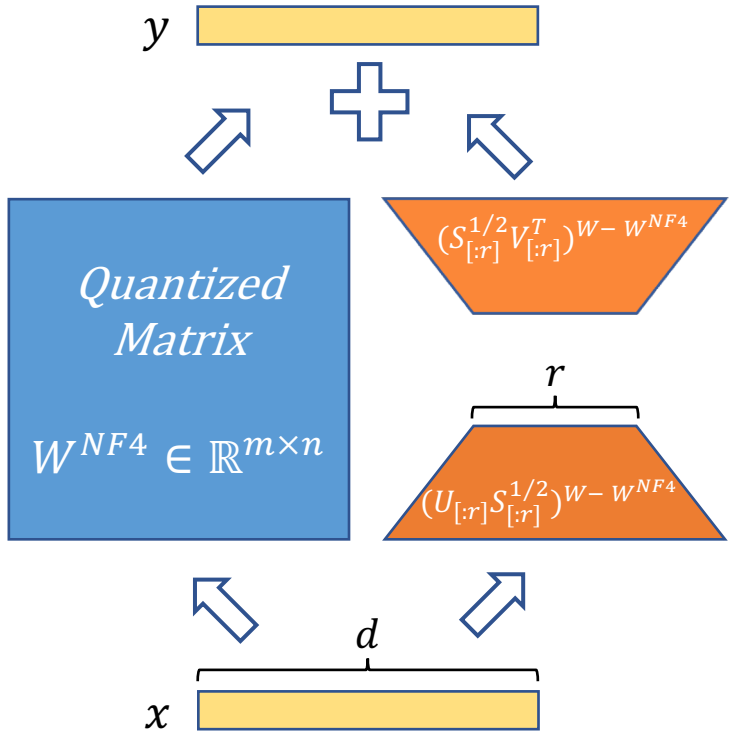
Memory Requirement of Parameter-Efficient Finetuning

- For inference only
  - In float32, every parameter of the model is stored in 32 bits or 4 bytes. Hence 4 bytes / parameter * 7 billion parameters = 28 billion bytes = 28 GB of GPU memory required.
  - In half precision, each parameter would be stored in 16 bits, or 2 bytes. Hence you would need 14 GB for inference.
  - There are now also 8 bit and 4 bit algorithms, so with 4 bits (or half a byte) per parameter you would need 3.5 GB of memory for inference.
- For training, it depends on the optimizer you use.
  - In case you use regular AdamW, then you need 8 bytes per parameter (as it not only stores the parameters, but also their gradients and second order gradients). Hence, for a 7B model you would need 8 bytes per parameter * 7 billion parameters = 56 GB of GPU memory.
  - If you use AdaFactor, then you need 4 bytes per parameter, or 28 GB of GPU memory.
  - With the optimizers of bitsandbytes (like 8 bit AdamW), you would need 2 bytes per parameter, or 14 GB of GPU memory.
- This guide: Efficient Training on a Single GPU which goes over all of this in much more detail.

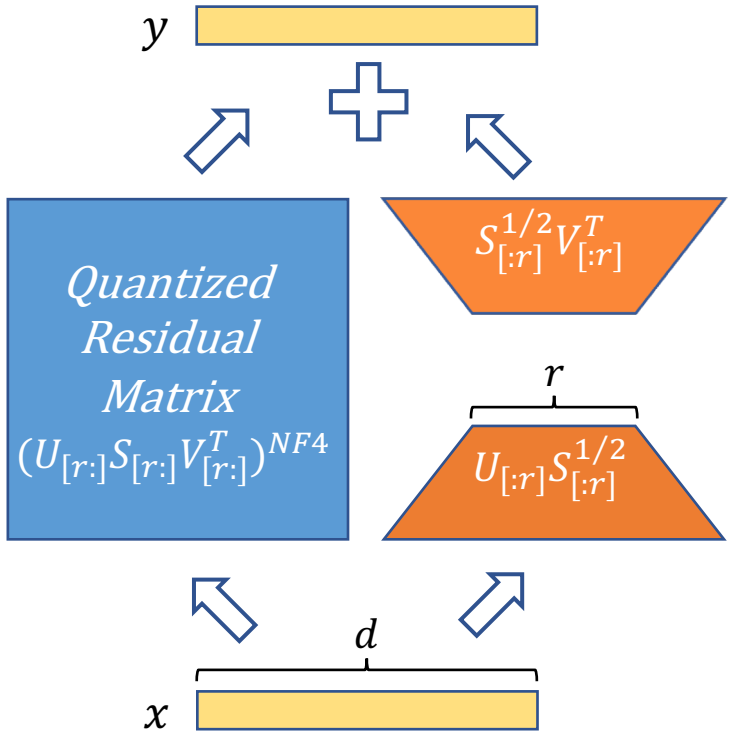
LoRA + Quantization



a) QLoRA (NeurIPS 2023)

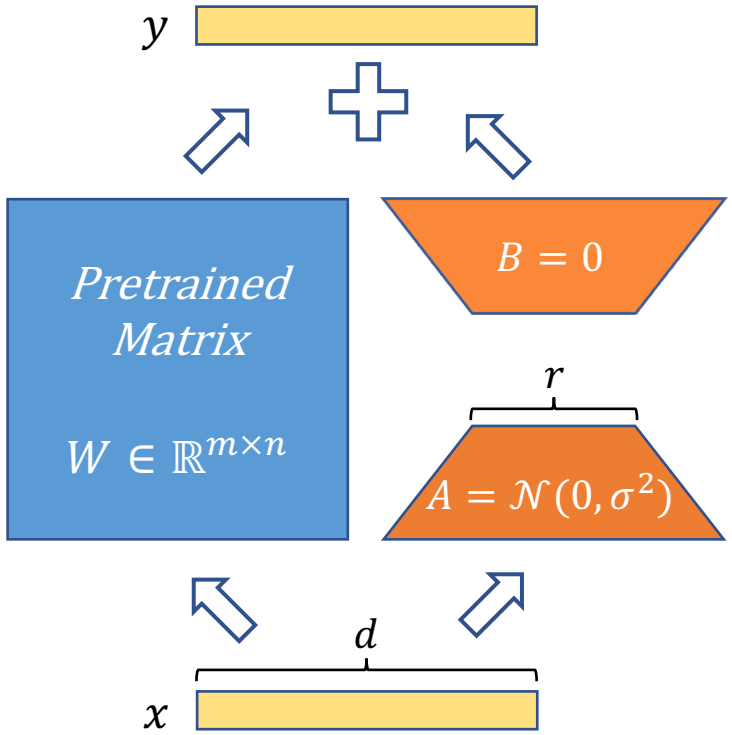


b) LoftQ (ICLR 2024 oral)

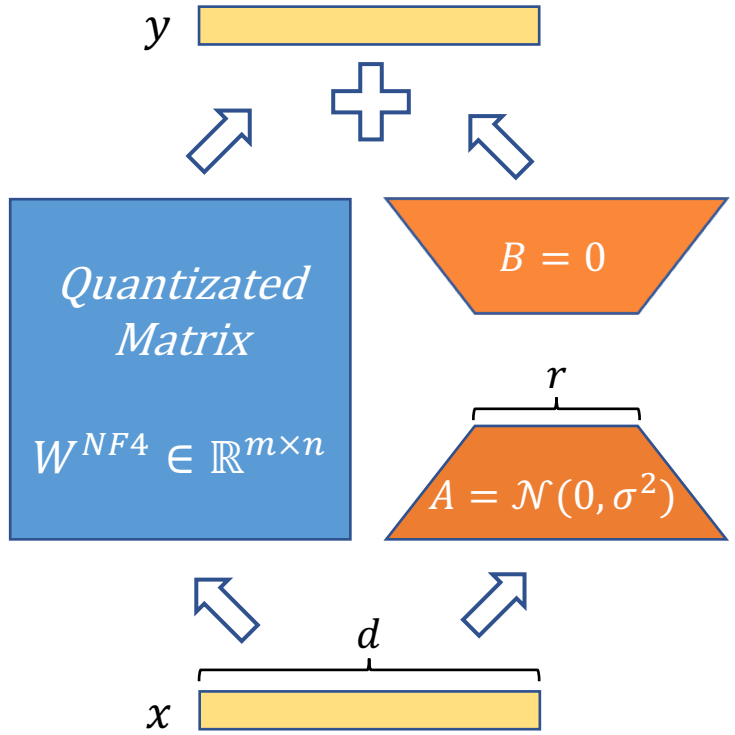


c) PiSSA

QLoRA



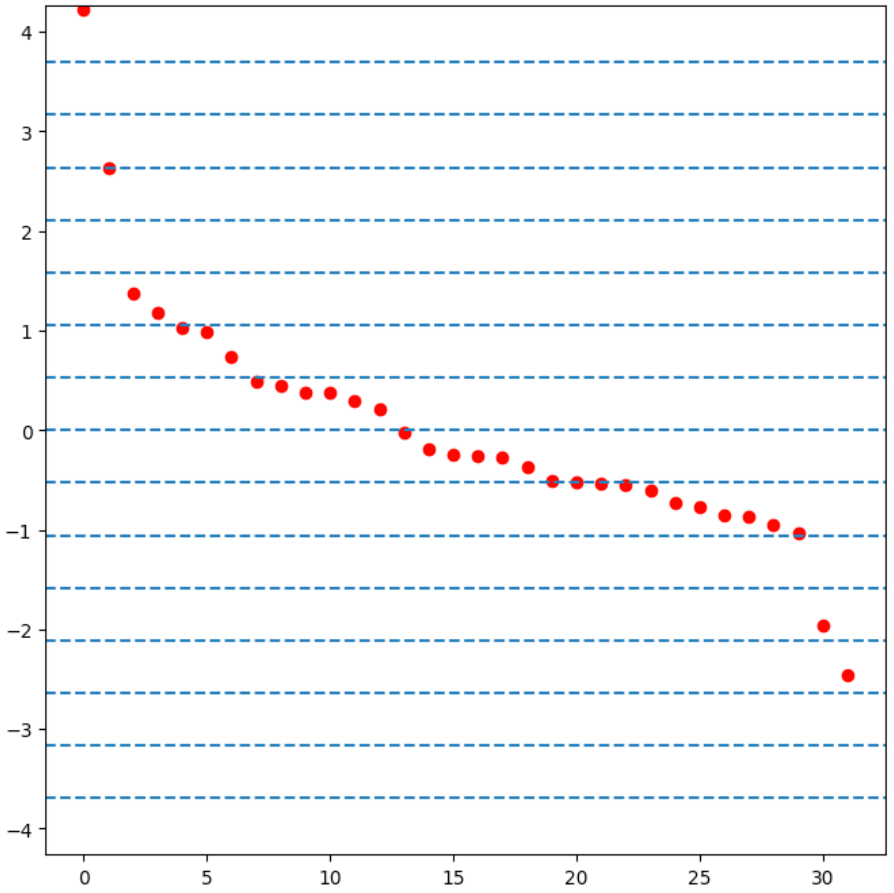
a) LoRA



b) QLoRA

$$Y^{BF16} = X^{BF16} doubleDequant\left(c_1^{FP32}, c_2^{FP8}, W^{NF4}\right) + X^{BF16} L_1^{BF16} L_2^{BF16}$$

Vanilla Quantization



Quantizing a 32-bit Floating Point (FP32) tensor into a Int8 tensor with range [−127, 127]:

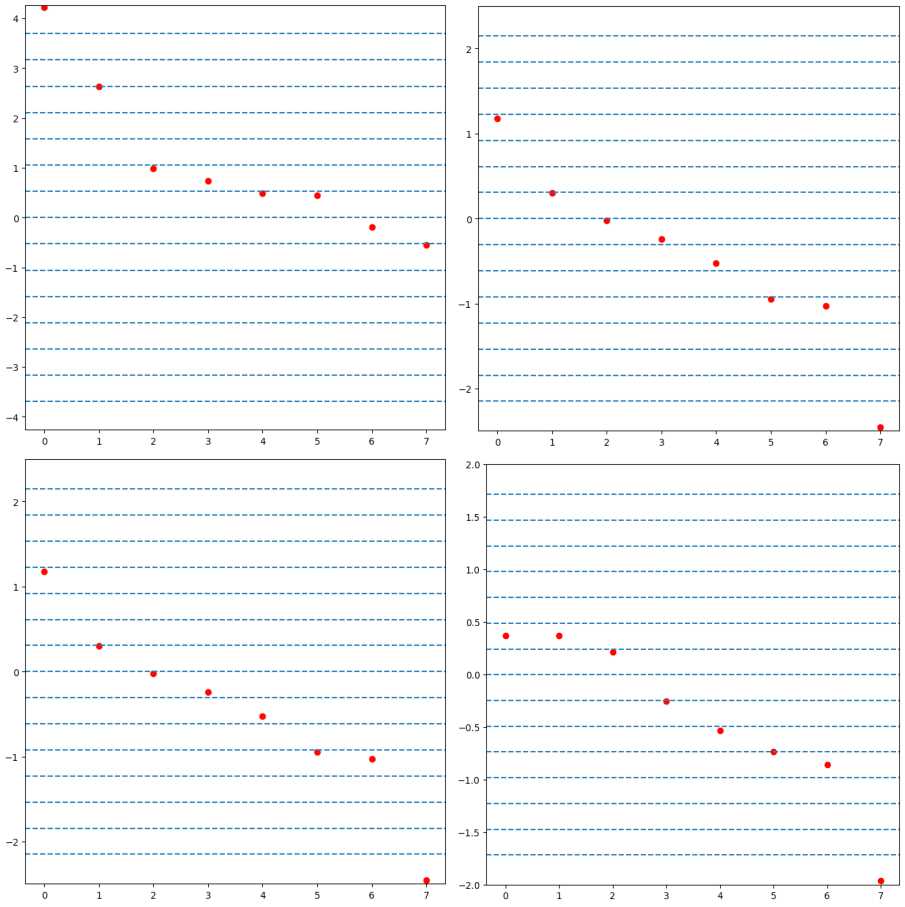
$$W^{int8} = round\left(\frac{127}{absmax(W^{FP32})} W^{FP32}\right)$$

$$= round(c^{FP32} \cdot W^{FP32})$$

where c^{FP32} is the *quantization constant*.
Dequantization is the inverse:

$$dequant(c^{FP32}, W^{int8}) = \frac{W^{int8}}{c^{FP32}} = W^{FP32}$$

Block-wise Double Quantization

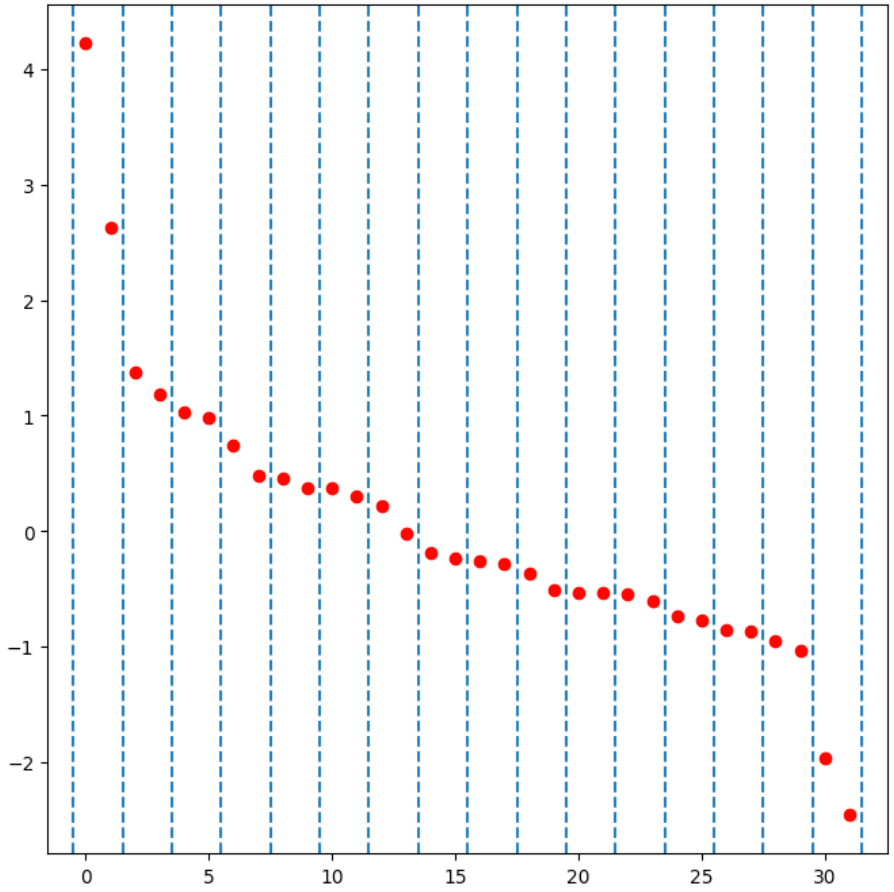


Using 32-bit constants and a blocksize of 64 for W, quantization constants add 32/64 = 0.5 bits per parameter on average.

$$doubleDequant\left(c_1^{FP32}, c_2^{FP8}, W^{NF4}\right)$$
$$= dequant\left(dequant\left(c_1^{FP32}, c_2^{FP8}\right), W^{NF4}\right)$$
$$= W^{BF16}$$

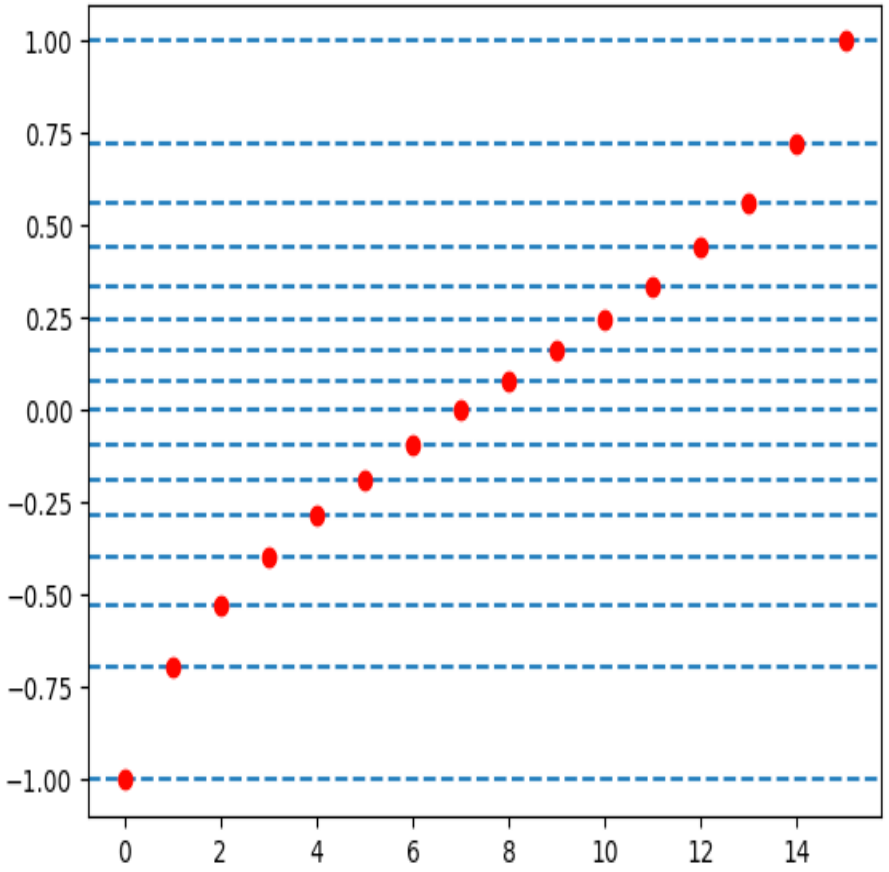
Using 8-bit Floats with a blocksize of 256 for the second quantization, reduces the memory footprint per parameter to 8/64 + 32/(64 · 256) = 0.127 bits, a reduction of 0.373 bits per parameter.

Quantile Quantization



- Information-theoretically optimal data type that ensures each quantization bin has an equal number of values assigned from the input tensor.
- The main limitation of quantile quantization is that the process of quantile estimation is expensive.
- The data type has large quantization errors for outliers, which are often the most important values.

Normal Float Quantile Function



Pretrained neural network weights usually have a zero-centered normal distribution with standard deviation σ.

Estimate the 2^k values q_i of the data type as follows:

$$q_i = \frac{1}{2}\left(Q_X\left(\frac{i}{2^k+1}\right) + Q_X\left(\frac{i+1}{2^k+1}\right)\right)$$

where $Q_X(\cdot)$ is the quantile function of the standard normal distribution $N(0,1)$.

QLoRA

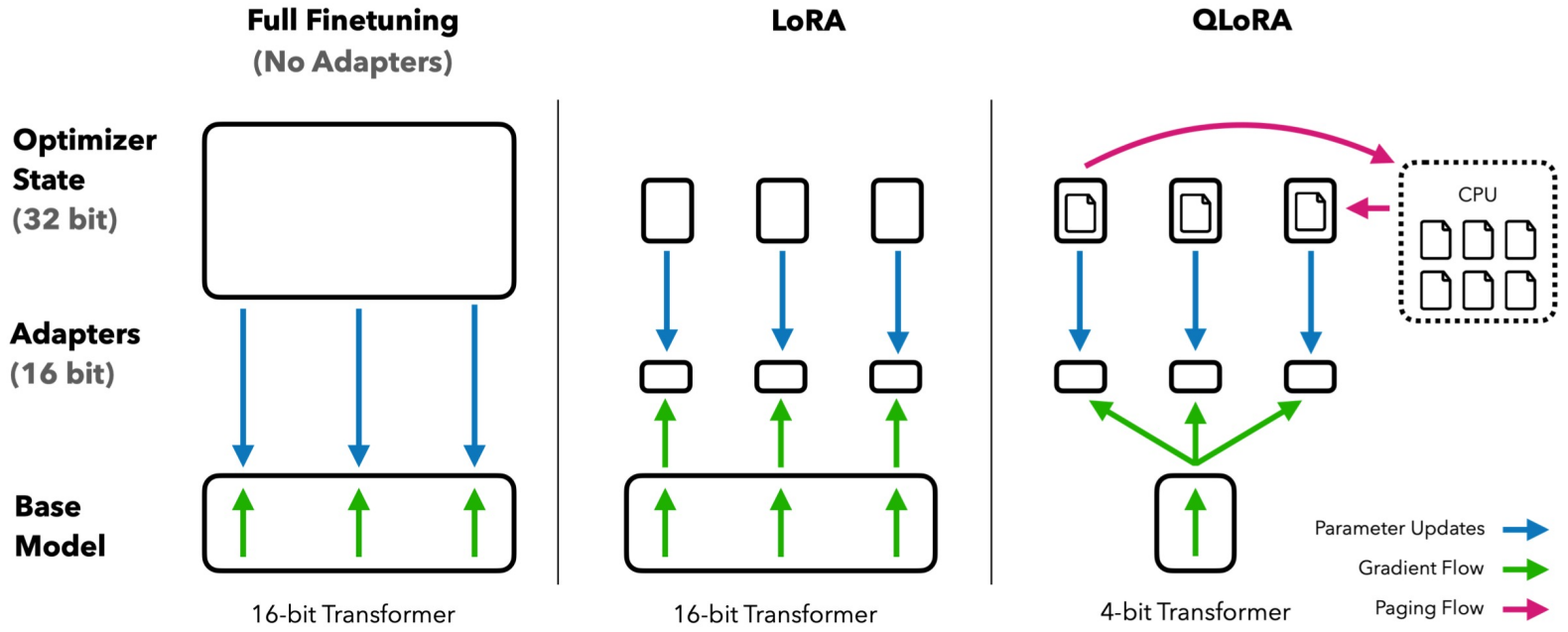


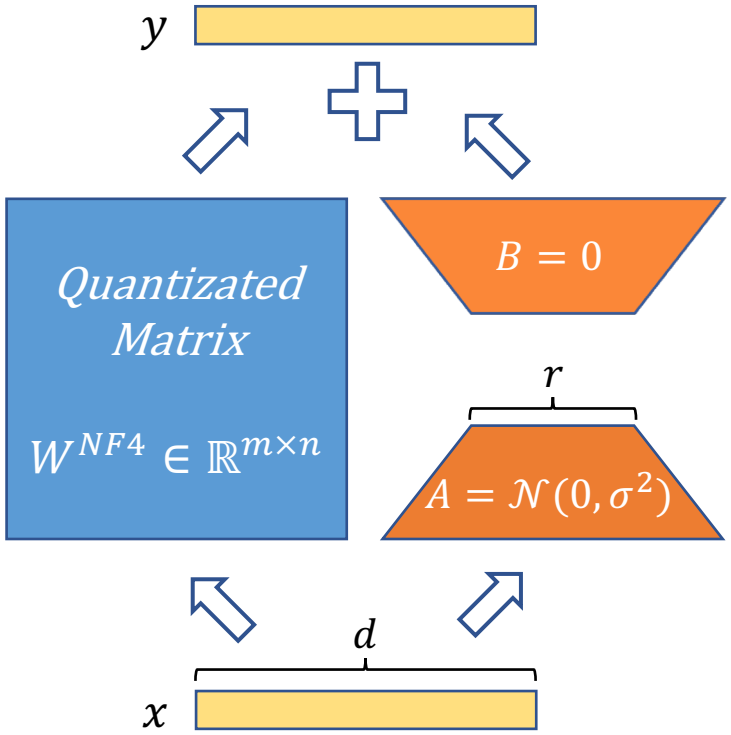
Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

$$Y^{BF16} = X^{BF16} doubleDequant\left(c_1^{FP32}, c_2^{FP8}, W^{NF4}\right) + X^{BF16} L_1^{BF16} L_2^{BF16},$$

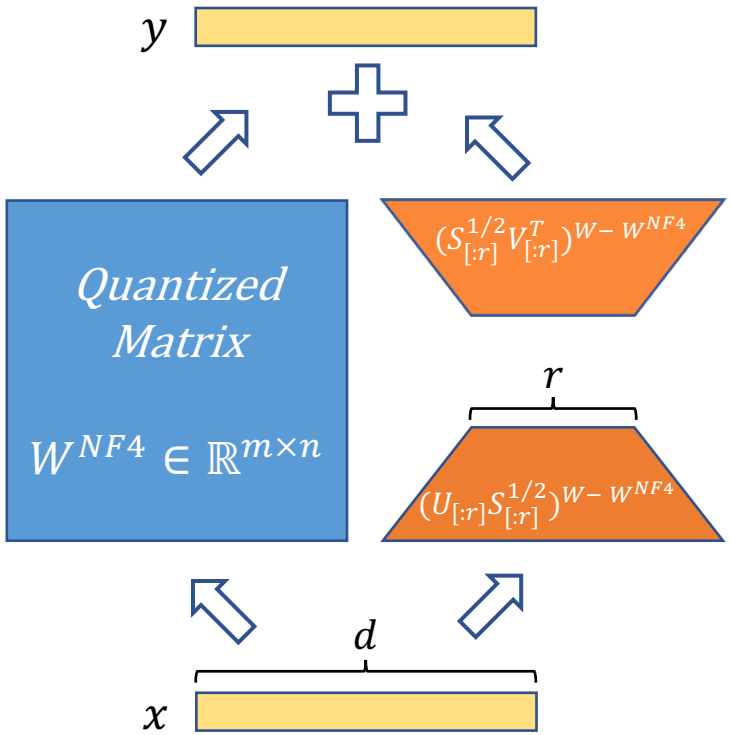
where $doubleDequant(\cdot)$ is defined as:

$$doubleDequant\left(c_1^{FP32}, c_2^{FP8}, W^{NF4}\right) = dequant\left(dequant\left(c_1^{FP32}, c_2^{FP8}\right), W^{NF4}\right) = W^{BF16}$$

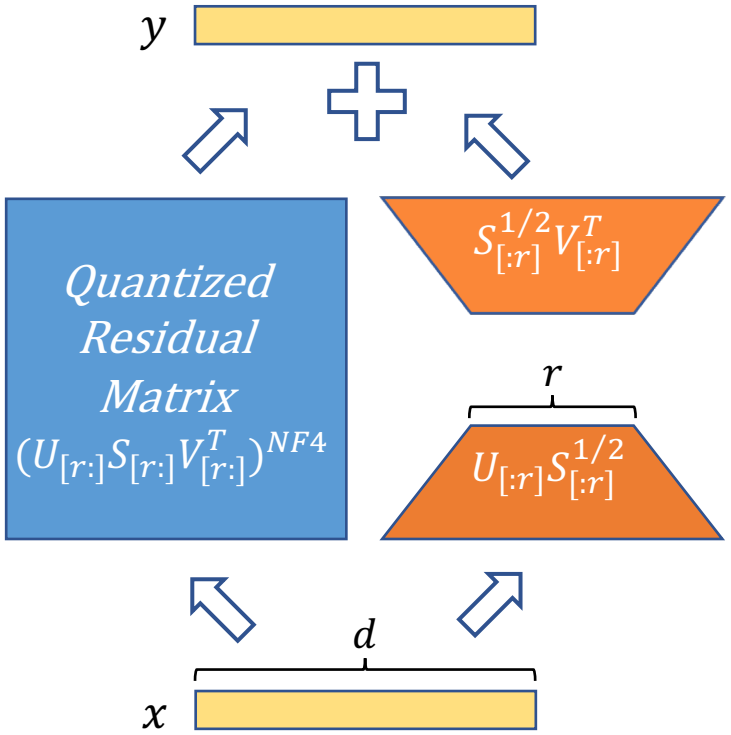
LoRA + Quantization



a) QLoRA (NeurIPS 2023)

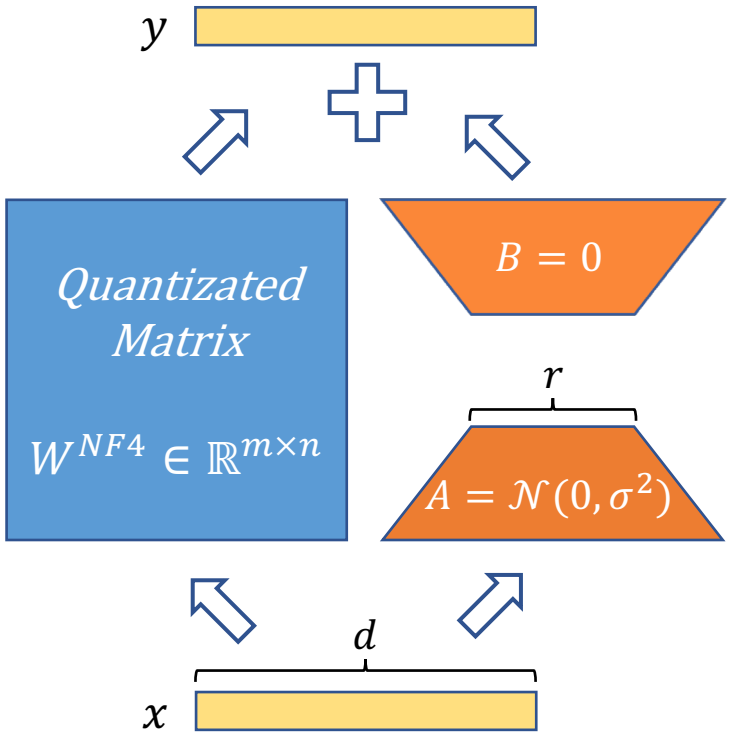


b) LoftQ (ICLR 2024 oral)

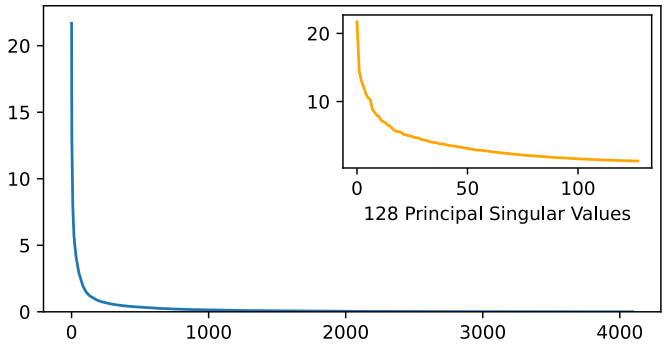


c) PiSSA

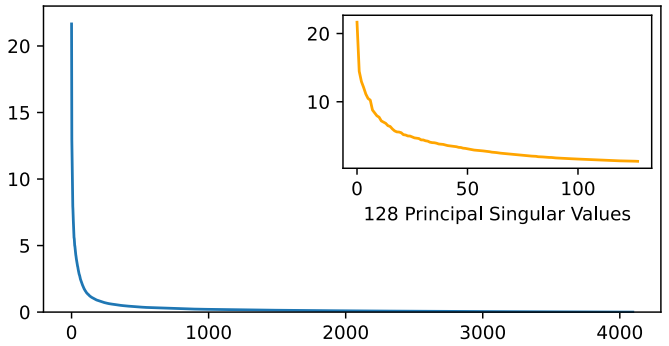
Quantization Error of QLoRA



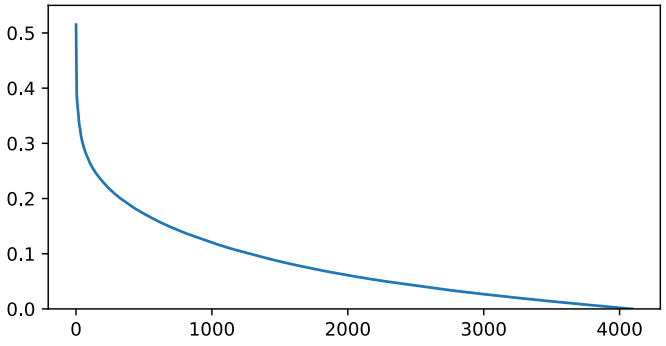
$$QLoRA^{err} = W - nf4(W)$$



a) Original Matrix W

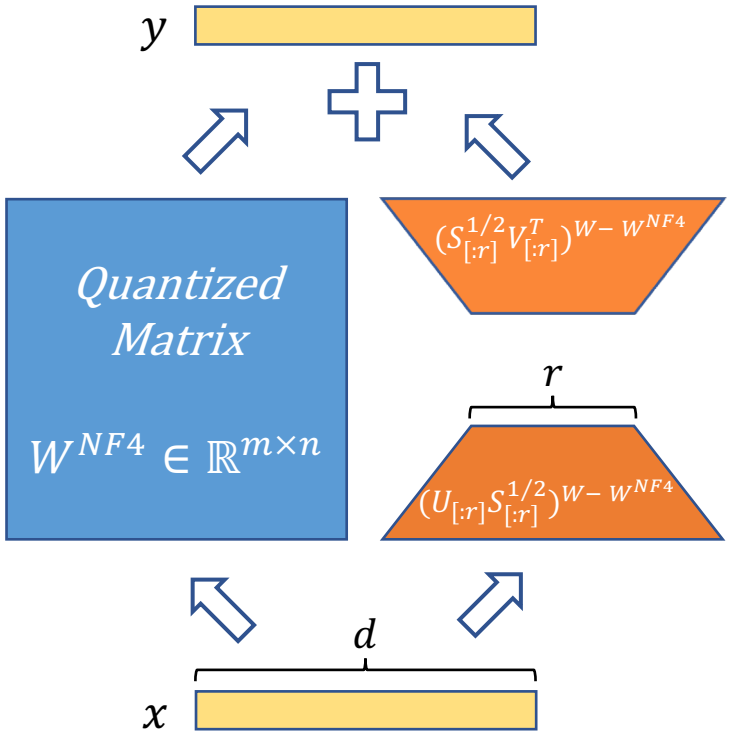


b) Quantized Matrix W^{NF4}

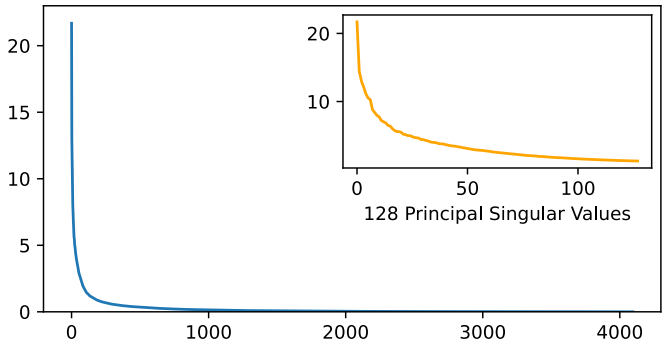


d) $QLoRA^{err}$

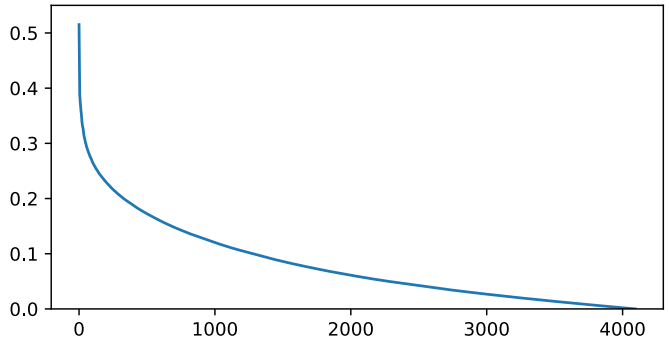
Quantization Error of LoftQ



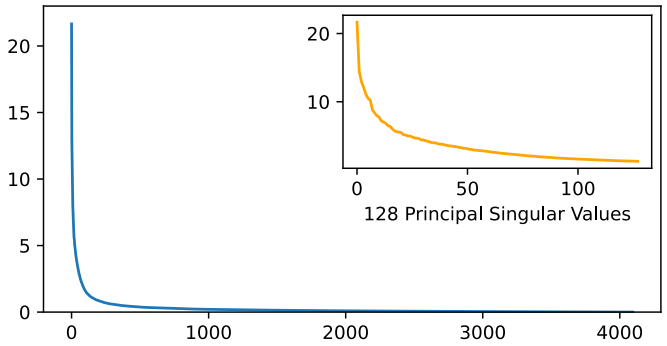
$$U^{err} S^{err} (U^{err})^T = QLoRA^{err}$$
$$LoftQ^{err} = U_{[r:]}^{err} S_{[r:]}^{err} (V_{[r:]}^{err})^T$$



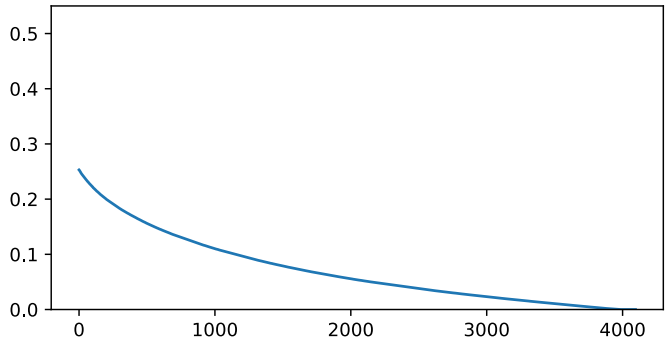
a) Original Matrix W



d) $QLoRA^{err}$

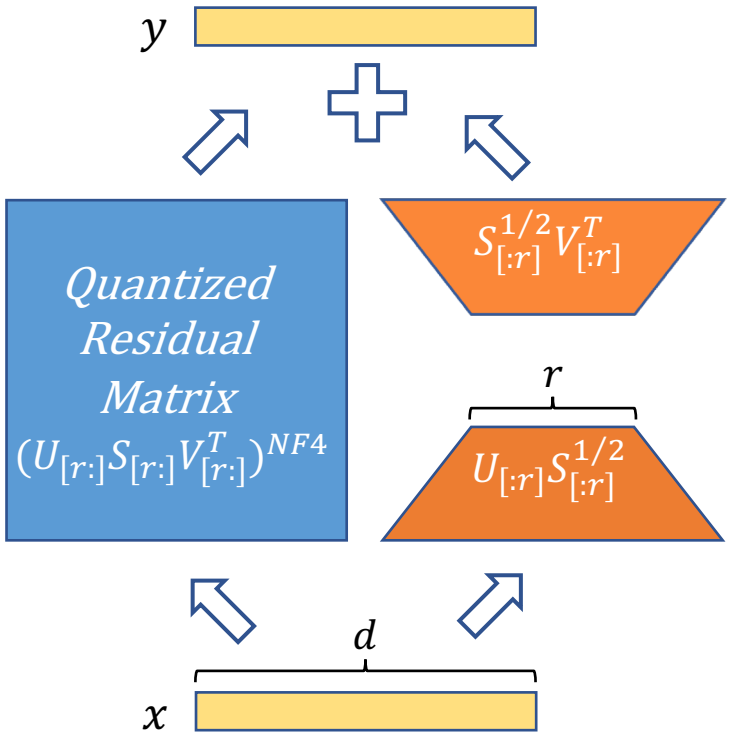


b) Quantized Matrix W^{NF4}

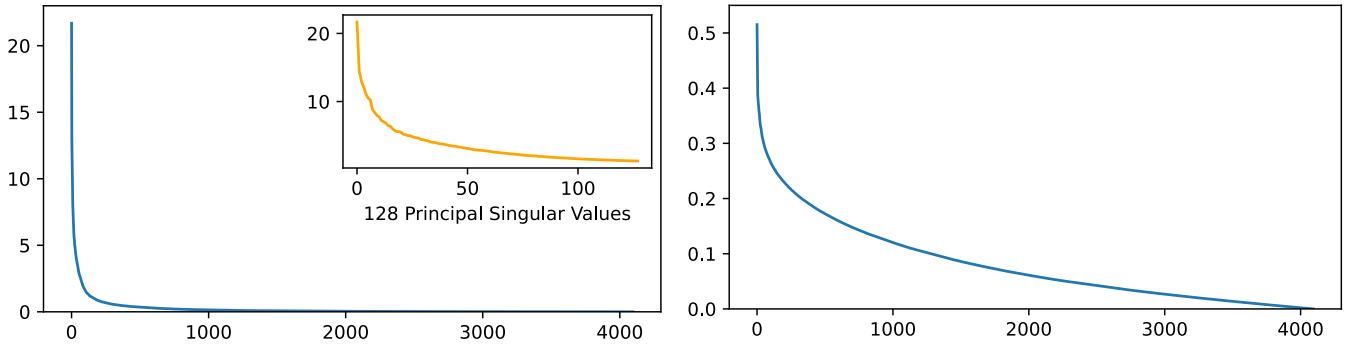


e) $LoftQ^{err}$

Quantization Error of PiSSA

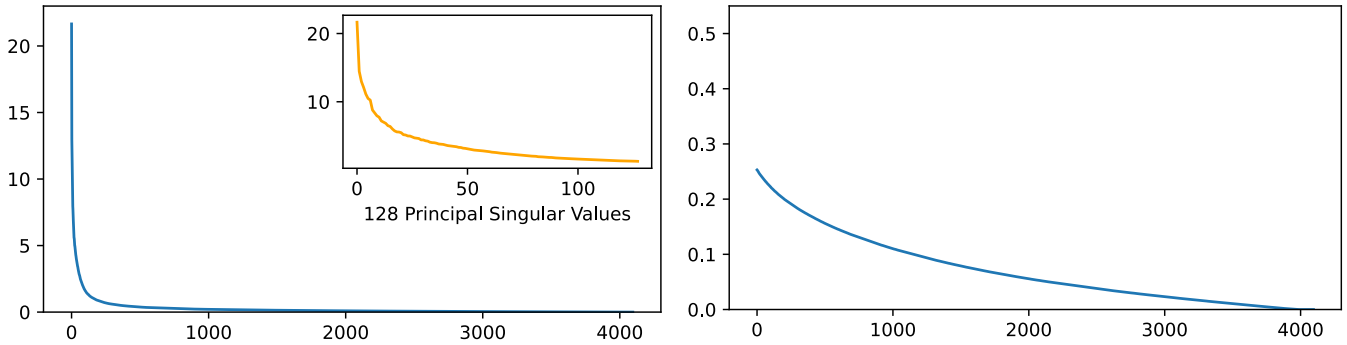


$$W^{res} = W - U_{[:r]}\,S_{[:r]}V_{[:r]}^T$$
$$PiSSA^{err} = W^{res} - nf4(W^{res})$$



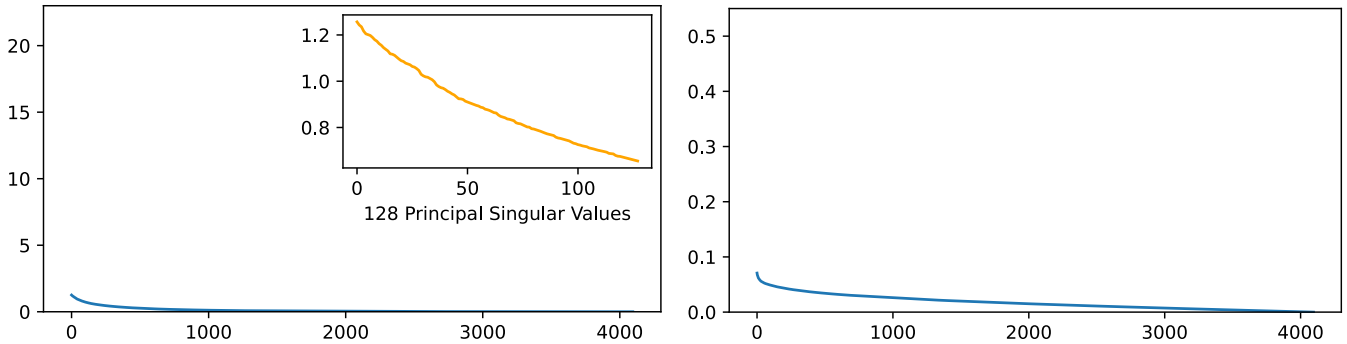
a) Original Matrix W

d) $QLoRA^{err}$



b) Quantized Matrix W^{NF4}

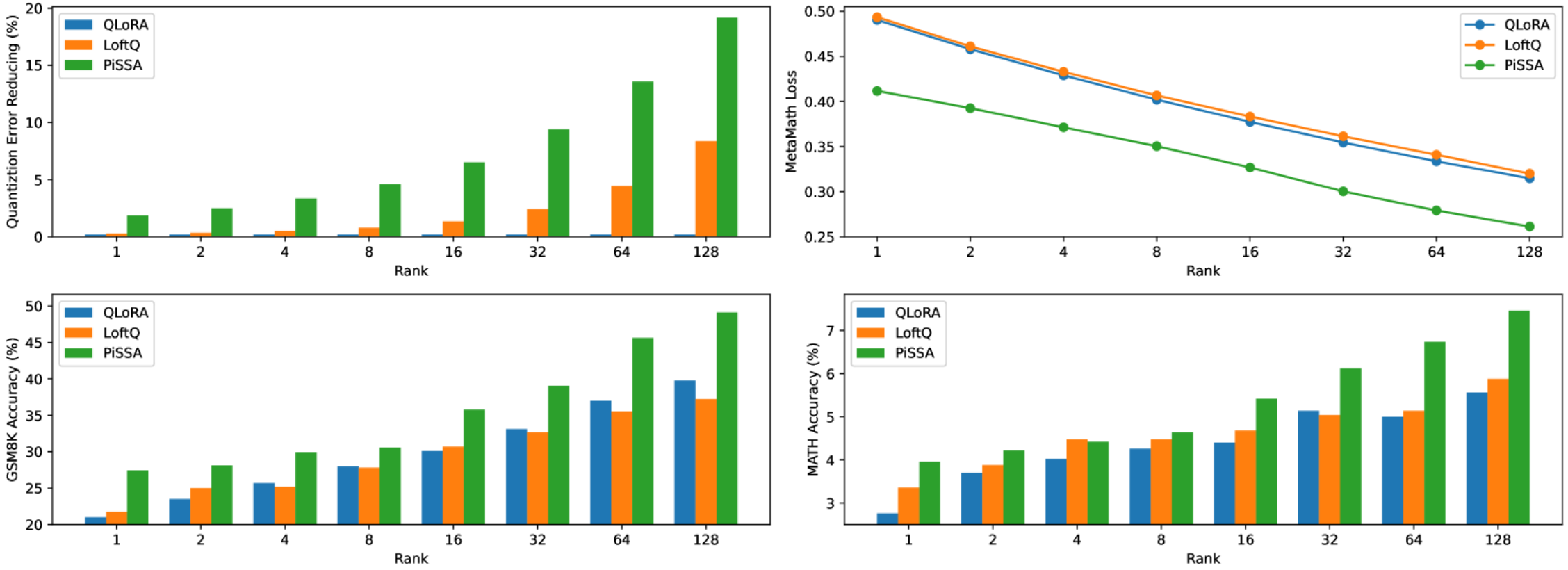
e) $LoftQ^{err}$



c) Residual Matrix W^{res}

f) $PiSSA^{err}$

LoRA + Quantization Fine-tuning

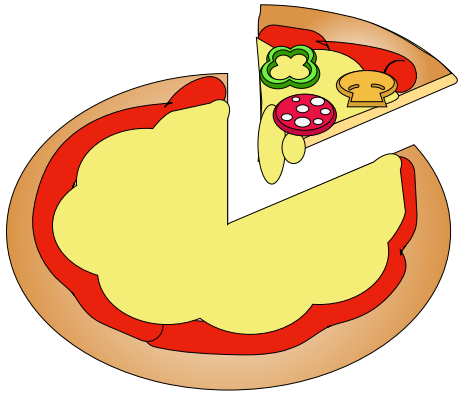


Comparing the quantization error, the fine-tuning loss on the MetaMathQA and the accuracy on the GSM8K and MATH validation sets.

Conclusion

北京大学 PEKING UNIVERSITY

- Proposes a new parameter efficient fine-tuning method **PiSSA**
- Shares the **same architecture** as LoRA, **different initializations** of W^{res} and AB
- A few lines of code change to LoRA, **significant performance improvement**!



- Paper arxiv: https://arxiv.org/pdf/2404.02948.pdf
- Github link: https://github.com/GraphPKU/PiSSA
- SR code:

Thanks for listening!