Quantized training

(or low-bit training)

Thien Tran – 2024/10/06

gau.nernst

gau-nernst

Recap – Post-training quantization (PTQ)

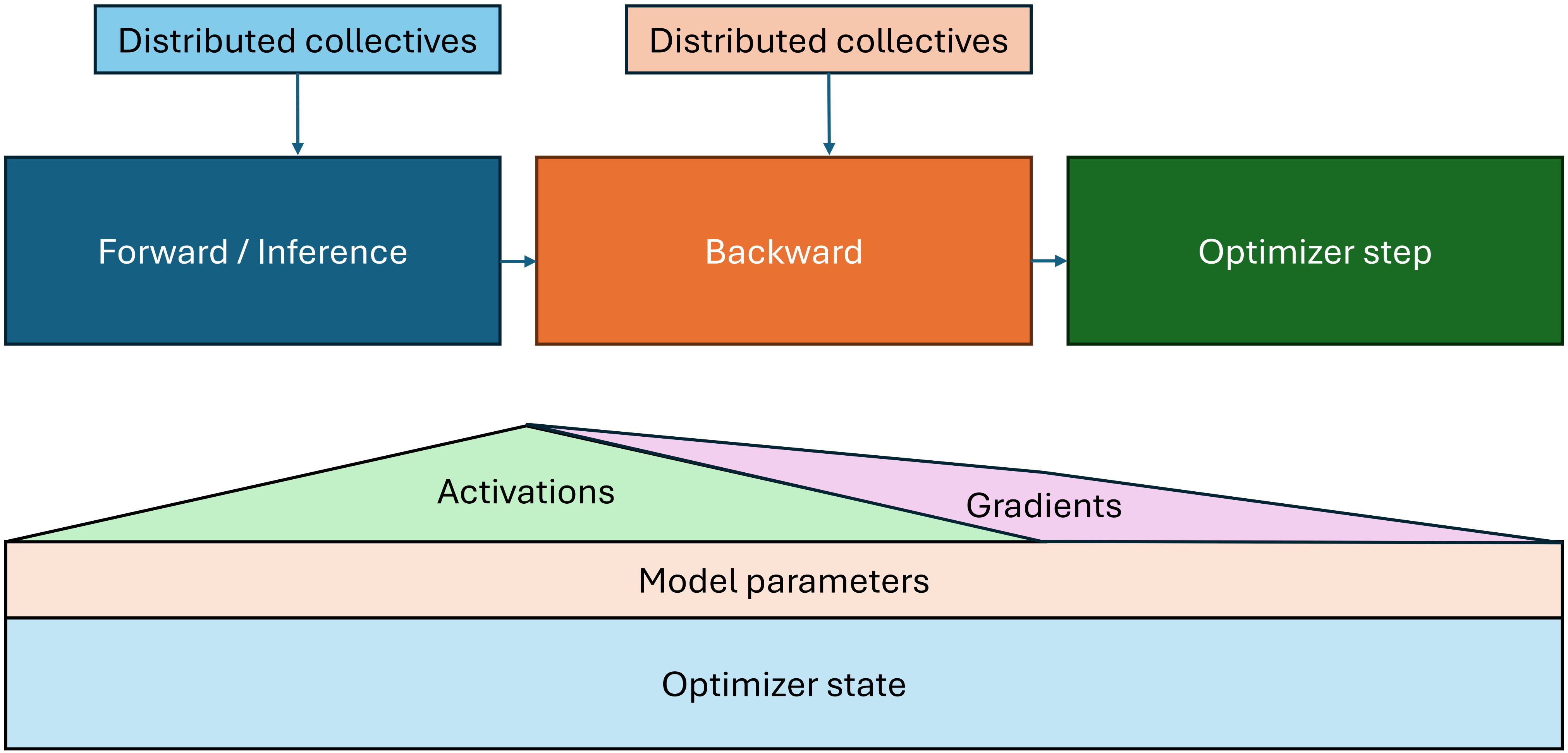
Forward / Inference

- Weight-only
- Dynamic act – weight quant
- Static act – weight quant

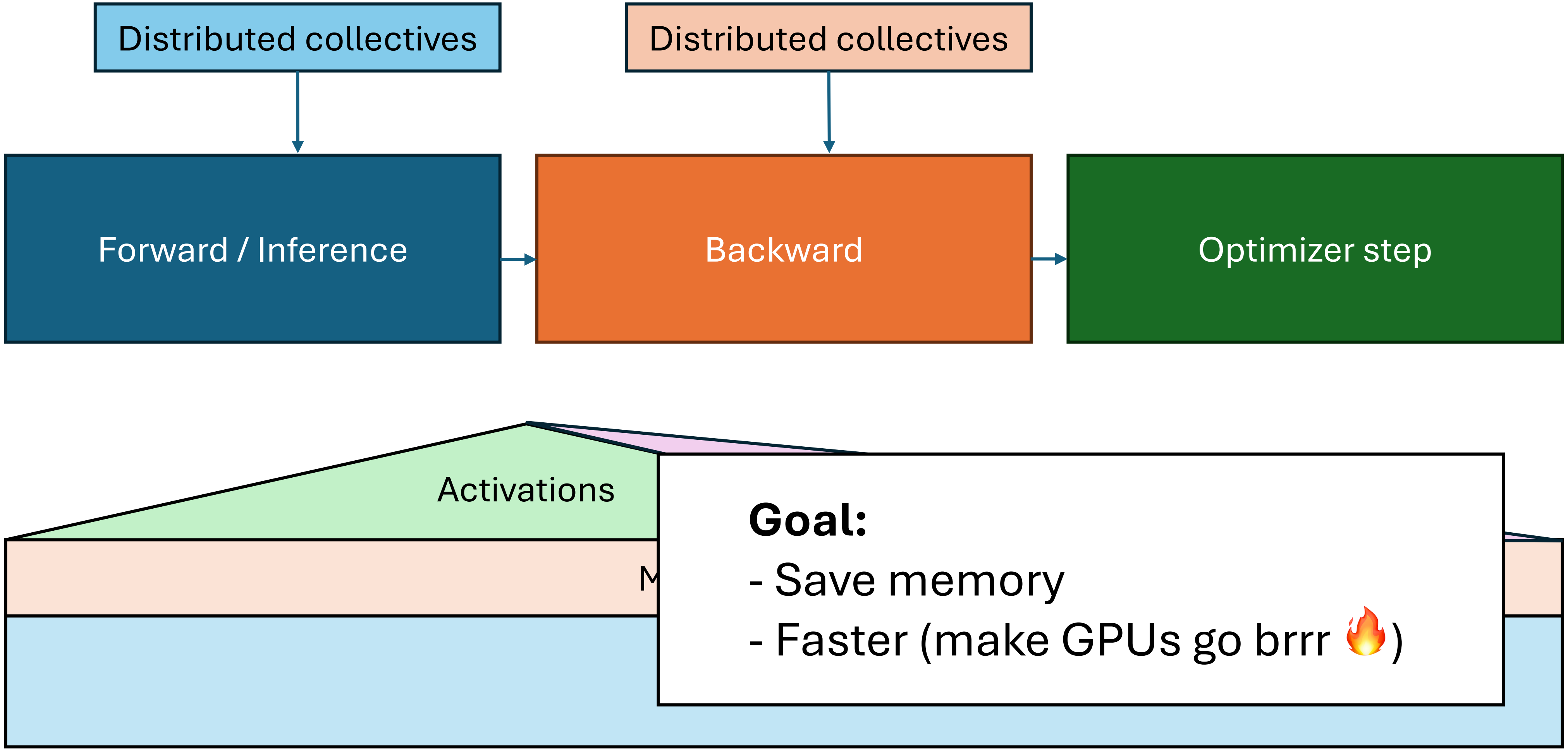
Training

- Weight-only
- Dynamic act – weight quant
- Static act – weight quant

Training (distributed)



Training (distributed)



Goal:
- Save memory
- Faster (make GPUs go brrr 🔥)

Low-bit optimizers

Low-bit optimizers

8-BIT OPTIMIZERS VIA BLOCK-WISE QUANTIZATION

Tim Dettmers[*‡] Mike Lewis[*] Sam Shleifer[*] Luke Zettlemoyer[*‡]
Facebook AI Research[*], {mikelewis,sshleifer}@fb.com
University of Washington[‡], {dettmers,lsz}@cs.washington.edu

https://arxiv.org/abs/2110.02861

Memory Efficient Optimizers with 4-bit States

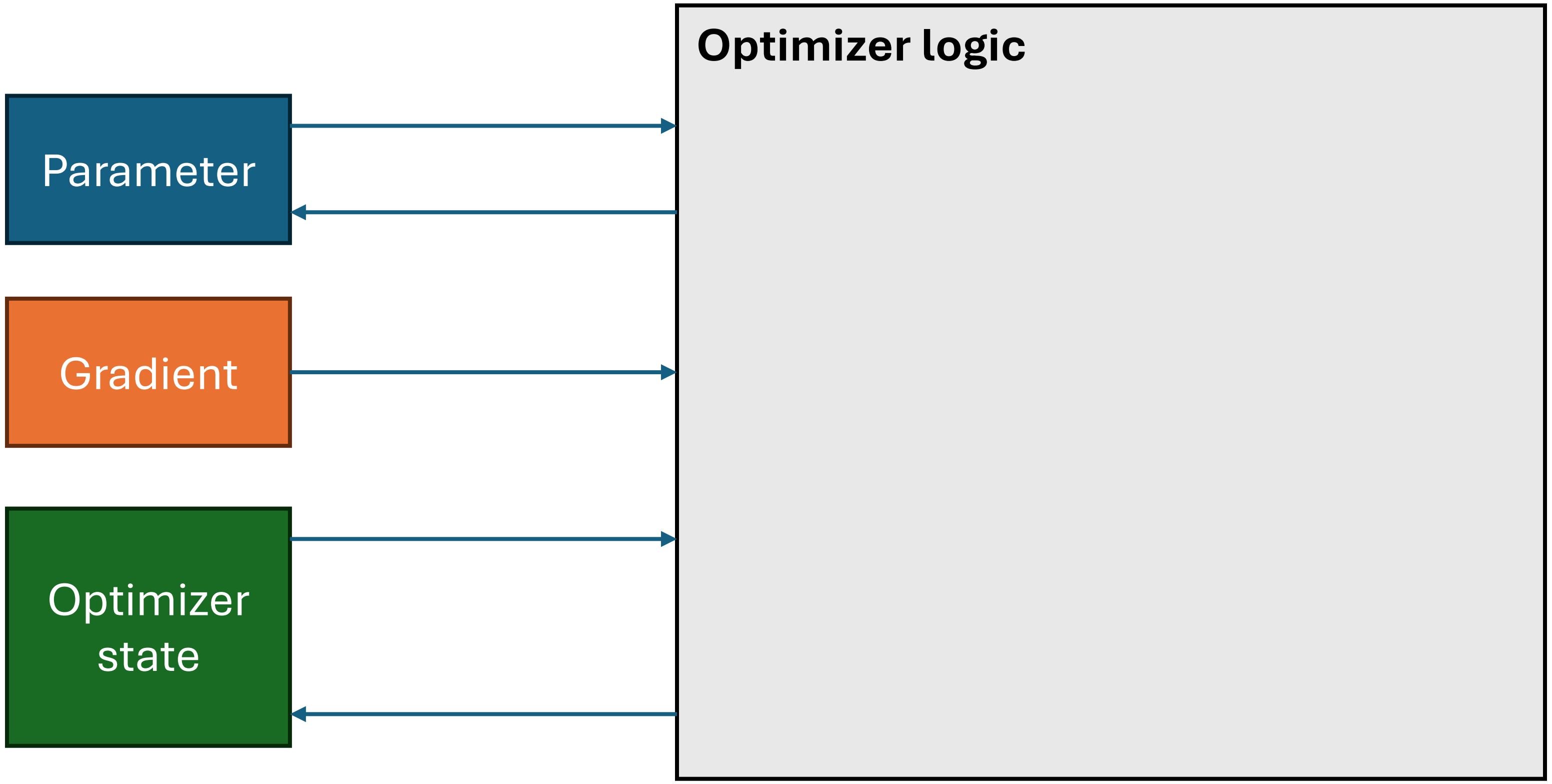
Bingrui Li[1], Jianfei Chen[1†], Jun Zhu[1]
[1]Dept. of Comp. Sci. and Tech., Institute for AI, BNRist Center, THBI Lab,
Tsinghua-Bosch Joint ML Center, Tsinghua University
lbr22@mails.tsinghua.edu.cn; {jianfeic, dcszj}@tsinghua.edu.cn

https://arxiv.org/abs/2309.01507

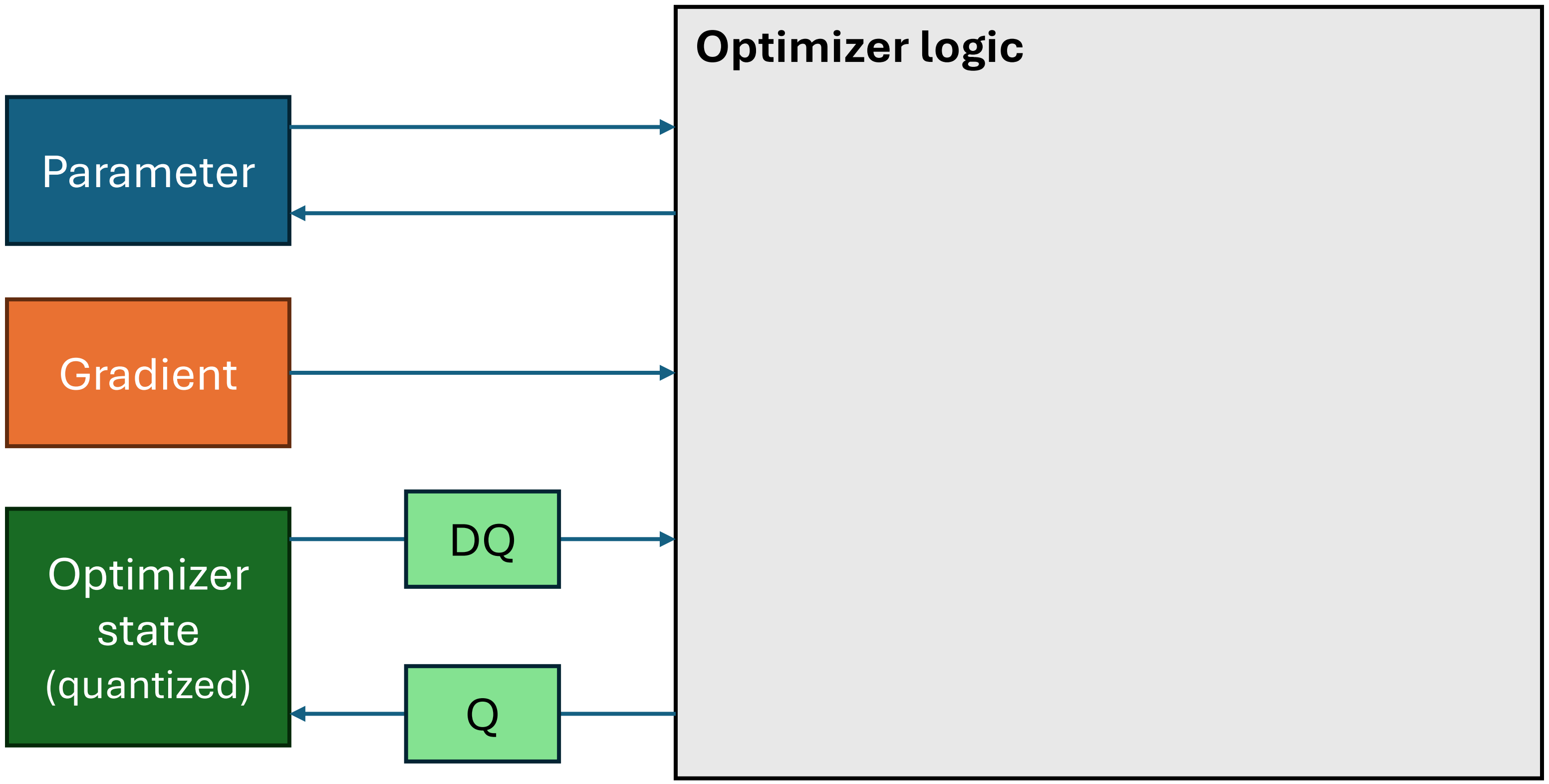
E.g. 8B model

FP32 Adam	8x2x4 = 64 GB
BF16 Adam	8x2x2 = 32 GB
8-bit Adam	8x2x1 = 16 GB
4-bit Adam	8x2x0.5 = 8 GB

Standard optimizers

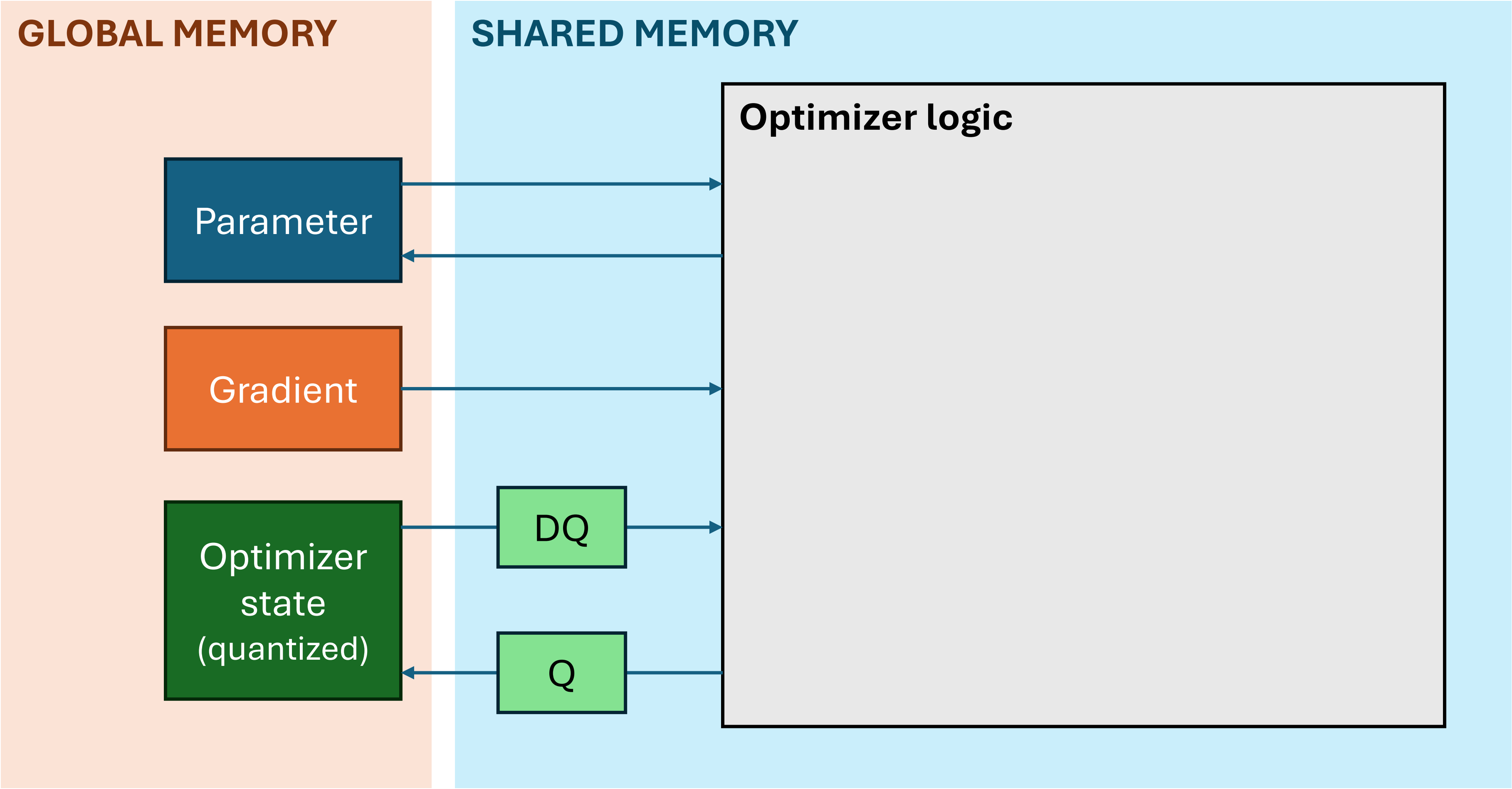


Low-bit optimizers



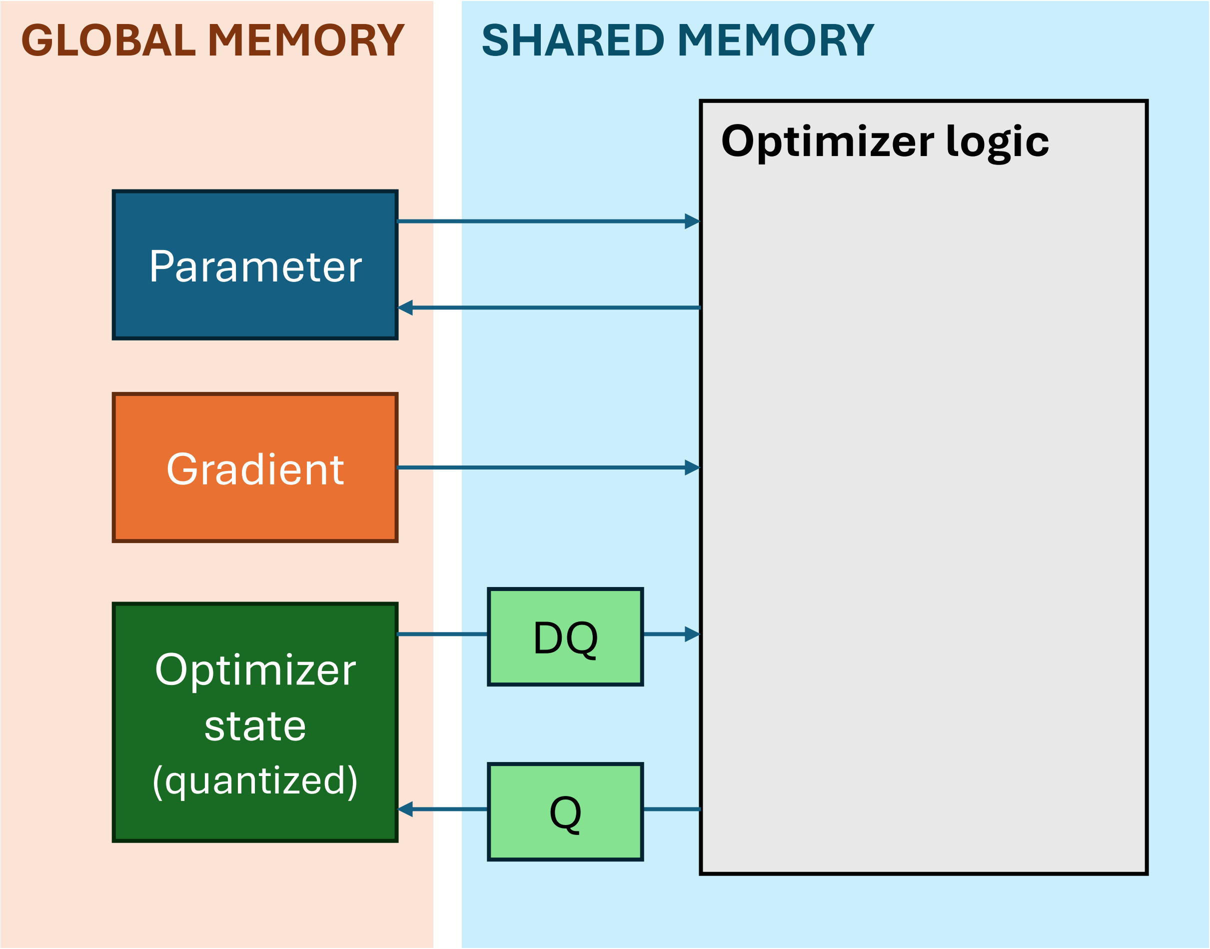
DQ: dequantize
Q: quantize

Low-bit optimizers



DQ: dequantize
Q: quantize

Low-bit optimizers



DQ: dequantize
Q: quantize

- Must not materialized dequantized tensors in global memory -> fuse DQ and Q

- Fused quantization -> must use small quantization group-size / block-size

Implementation
https://github.com/pytorch/ao/blob/main/torchao/prototype/low_bit_optim/adam.py

https://github.com/pytorch/ao/blob/main/torchao/prototype/low_bit_optim/subclass_8bit.py
- aten.lerp.Scalar
- aten.copy_.default

Mini Demo

Low-bit weight-only training

Can we train quantized weights without high precision copy?

Training with low-bit recap

FP16/BF16 mixed-precision training

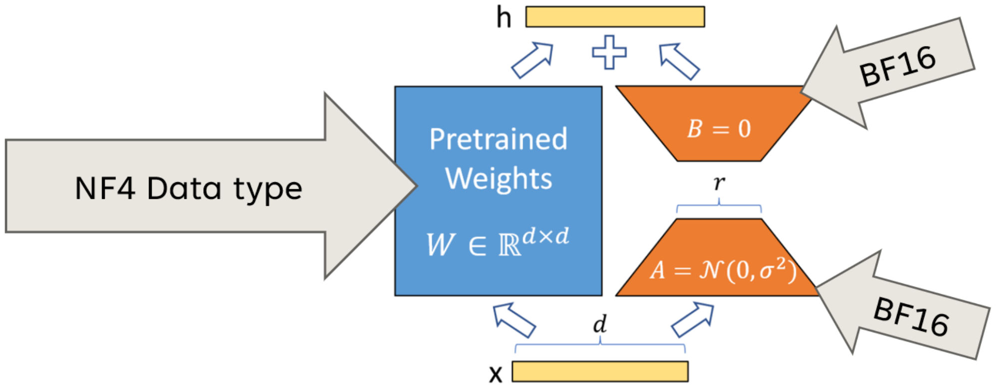
Dynamically cast weight to FP16/BF16 to utilize Tensor Cores.
<u>Keep FP32 weights for weight update.</u>

Full FP16/BF16 training

Only keep FP16/BF16 weights
-> Might have issues (next slide)

QLoRA

Quantize base weight to NF4, while LoRA weights are in high precision (FP32 or BF16)
Quantized base weights are not trained.



QLoRA: https://arxiv.org/abs/2305.14314

Weight-update underflow

$$p_{t+1} = p_t - \overbrace{lr \cdot Optim(p_t, g_t, m_t, t)}^{\text{Weight update might underflow}}$$

e.g. using INT8
1 + 0.1 = 1

p: parameter
g: gradient
m: optimizer state
t: step
lr: learning rate

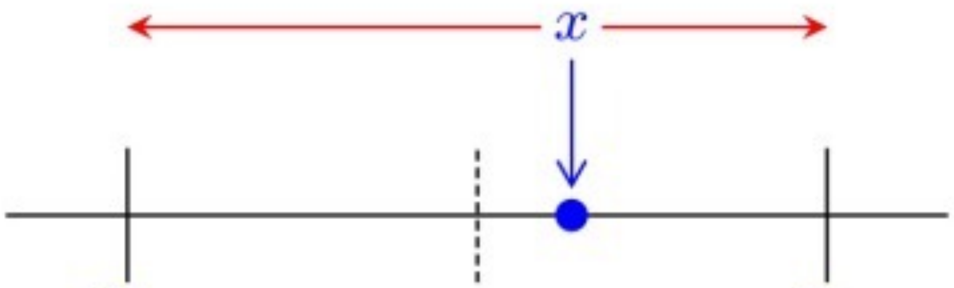
Stochastic rounding

Definition. x is rounded up with the probability of $p = x - \lfloor x \rfloor$

$$\Rightarrow \mathbb{E}[SR(x)] = x$$

1. Avoid weight-update underflow
2. Many small additions follow the correct trajectory

More likely to round to the nearer number



$\lfloor x \rfloor$ $\qquad$ $\lfloor x \rfloor + 1$

Mini demo:
- SR for INT8
- SR for BF16

Implementation & Results

https://github.com/pytorch/ao/pull/644
- LLM fine-tuning
- LLM pre-training

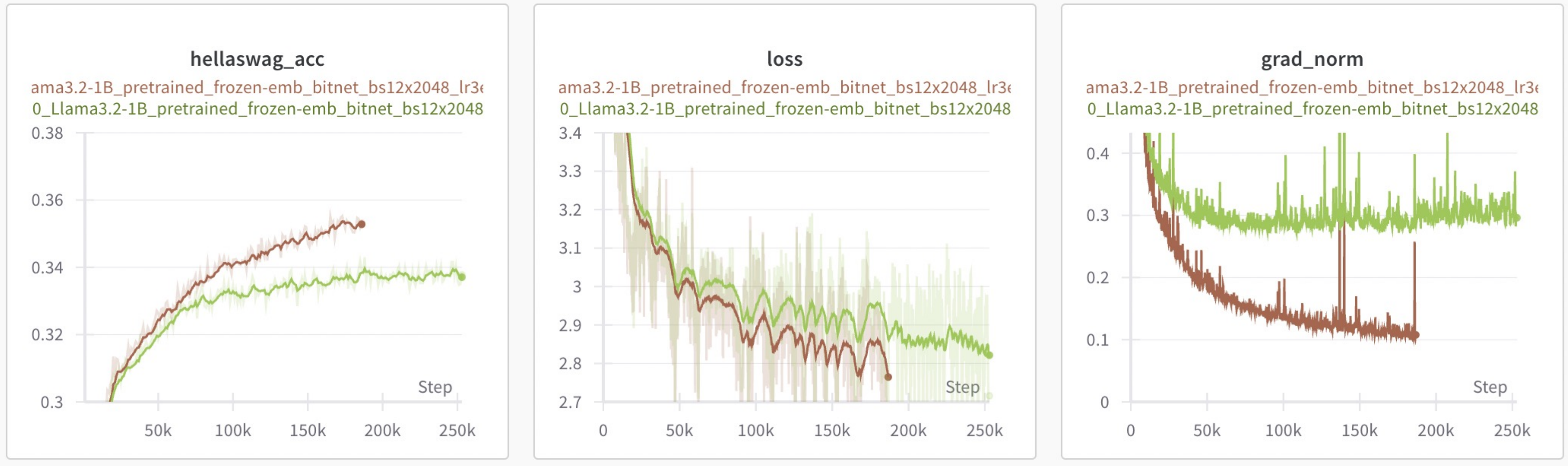
https://github.com/pytorch/ao/blob/main/torchao/prototype/quantized_training/int8.py
- Custom Autograd function
- SR logic

Some thoughts
- Still not very attractive: memory reduction is not yet ideal (might be due to row-wise scaling) + slower training due to quantization overhead + accuracy is slightly lower
- For fine-tuning, QLoRA is simpler

BF16 training w/ stochastic rounding

https://arxiv.org/abs/2010.06192

Implementations
https://github.com/karpathy/llm.c/blob/7ecd8906afe6ed7a2b2cdb731c042f26d525b820/llmc/adamw.cuh#L19-L46
https://github.com/gau-nernst/quantized-training/blob/c42a7842ff6a9fe97bea54d00489e597600ae683/other_optim/bf16_sr.py#L108-L122



Low-bit mixed-precision training

Faster training w/ low-bit matmul

	4090	A100	H100	GB200
BF16 TFLOPS	165.2	312	989.4	5,000
INT8 TOPS	660.6 (x4)	624 (x2)	1,978.9 (x2)	10,000 (x2)
FP8 TFLOPS	330.3 (x2)	-	1,978.9 (x2)	10,000 (x2)
INT4 TOPS	1,321.2 (x8)	1,248 (x4)	-	-
FP4 FLOPS	-	-	-	20,000 (x4)

https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-V2.02.pdf
https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf
https://resources.nvidia.com/en-us-tensor-core
https://resources.nvidia.com/en-us-blackwell-architecture/blackwell-architecture-technical-brief

Scaled low-bit matmul

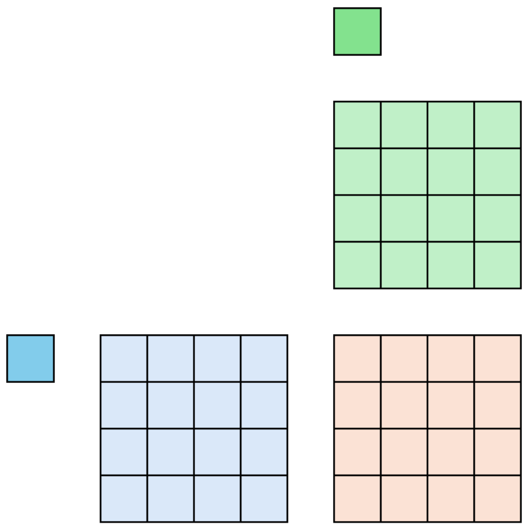
Problem Reduced range -> overflow/underflow

dtype	absmin	absmax
INT8	1	127/128
FP8 E4M3FN	2^(-9)	448

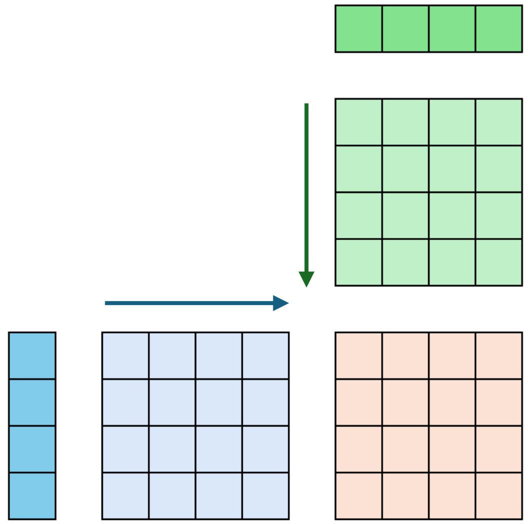
Solution Scaled matmul

(m,k) x (k,n) = (m,n)

Easiest way (w/o modification to matmul inner loop)
- Scale inputs to appropriate range
- Scale back outputs to correct values



Tensor-wise scaling
FP8 training



Row-wise scaling
& Column-wise scaling
INT8 PTQ

Low-bit mixed-precision traning

Weight is in high precision (typically FP32)

For matmul A @ B, dynamically quantize both A and B to low precision, both in forward and backward passes

Matmul dtype	Scaling strategy
FP16	No scaling in forward pass. Loss scaling in backward pass
BF16	No scaling
FP8	Tensor-wise scaling for A and B
INT8	Row-wise scaling for A and column-wise scaling for B

Implementation & Results

Scaled INT8 matmul in Triton
https://github.com/pytorch/ao/blob/9e2a2536d56626e59618a8932e2d1e160f7f76ca/torchao/prototype/quantized_training/int8_mm.py#L54-L125

Custom Autograd function
https://github.com/pytorch/ao/blob/9e2a2536d56626e59618a8932e2d1e160f7f76ca/torchao/prototype/quantized_training/int8_mixed_precision.py#L179-L219

Results
https://github.com/pytorch/ao/pull/748

INT8 weight-only + INT8 matmul?

Row-wise scaling in forward pass
➔ Become **column-wise scaling** in backward pass

- Tensor-wise scaling won't have this issue.
- Also possible to dequant and re-quant in the other axis, but will incur extra overhead.
- QLoRA + FP8/INT8 matmul: need to dequant weight before matmul anyway.

Foward

$$Y = X \cdot W^T$$

(m,n) = (m,k) x (k,n)

Backward

$$G_X = G_Y \cdot W$$

(m,k) = (m,n) x (n,k)

$$G_W = G_Y^T \cdot X$$

(n,k) = (n,m) x (m,k)

Y: output
X: input
W: weight
G_x: gradient wrt x

Derivation: https://web.eecs.umich.edu/~justincj/teaching/eecs442/notes/linear-backprop.html

BitNet 1.58-bit

https://arxiv.org/abs/2402.17764

Weight: tensor-wise abs-mean scaling to ternary (-1, 0, 1)
Activation: per-token (row-wise) abs-max scaling to INT8

Originally trained with Quantization-Aware Training (QAT)

➔ We can use INT8 Tensor Cores! (and 2-bit all-gather for FSDP)
https://github.com/pytorch/ao/pull/930

(we can quantize backward pass too...)

Other interesting research works

Pareto-Optimal Quantized ResNet Is Mostly 4-bit
- INT8 ResNet outperforms BF16 ResNet (at the same params count)
- INT4 ResNet is the best (for a given model size in MB/GB)

Binarized Neural Machine Translation
- Inspired BitNet

Jetfire: Efficient and Accurate Transformer Pretraining with INT8 Data Flow and Per-Block Quantization
- Tile-wise quantization, with quantized matmul outputs
- INT8 LayerNorm and INT8 GELU

Concluding remarks

If you want faster training -> use **INT8/FP8 tensor cores**

- torch.compile() and Triton make things very simple to integrate

Stochastic rounding can help with high-bit -> low-bit casting

- Low-bit all-reduce?

Ideas to explore

- INT4 Tensor Cores 👀 (requires cutlass)
- Output low-bit activations from matmul -> low-bit RMSNorm / GELU / SiLU