① Mixed-Precision Training

Problem:

normally, model weights and gradients use FP32 (32-bit floating point), this takes a lot of memory (4 bytes per value) can cause CUDA out-of-memory errors

Solution:

△1 FP16

Use FP16 (16-bit) for smaller memory usage and faster training

But it has less precision and smaller numeric range, causing:

- Gradient underflow (small gradient become 0)
- Inaccurate weight update
  - because for $w = w - \alpha \cdot g$
    if $\alpha \cdot g <$ minimum representable value
    $\alpha \cdot g \approx 0$ => update value disappear => can't learn

↓

△2 Mixed Precision (FP16 + FP32)

Use FP16 for forward & backward passes

Keep a master copy in FP32 for updates

Apply loss scaling (multiply the loss by a large number before backpop) to prevent underflow

Steps:

keep FP32 "master weight"

↓

Forward propagation in FP16

↓

Scale loss

↓

Backward propagation in FP16

↓

copy grads to FP32 and divide by scale

↓

Update master weights in FP32

↓

Copy back to FP16 version

new format: bfloat16

same range as FP32, fewer bits for precision;

no loss scaling needed

② Multi-GPU Training

why use it?

Large model and large batches don't fit on the GPU; Distribute work to train faster and fit larger models.

① DDP (Distributed Data parallel)

Each GPU:

Has a copy of the model

↓

Processes its own batch

↓

Compute grads then all GPUs synchronize grads (all-reduce)

But the problem:

every GPU need to store FP16 model, gradients, master weights, optimizer states (like momentum, variance) → High memory usage

② ZeRO (Zero Redundancy Optimizer

Core idea: shard (split) optimizer data across GPUs to save memory and reduce redundancy

stages: ZeRO-1 => ZeRO-2 => ZeRO-3

③ FSDP (Fully Sharded Data Parallel)

trait:

1. model parameters, gradients, optimizer states are splited across GPUs
2. During forward/backward passes, GPUs gather parameters when needed and scatter them afterward

   => Great for very large models that can't fit anywhere fully

In this case, GPU memory usage mainly comes from: Gradients (FP16), master weights (FP32), Adam momentum (FP32), Adam variance (FP32), Activations

(depends on batch size)

(3) Parameter-Efficient Fine-tuning (LoRA)

Problem:

full-tuning updates all parameters of a model, so it's too expensive to store and train for each task

Solution: Parameter-Efficient Fine-tuning

Train only a small subset of new parameters (keep pretrained weights frozen)
it can make model much cheaper and faster, still similar performance to full-tuning

LoRA (Low-Rank Adaptation)

instead of directly changing large weight matrices W:

$$W' = W + \Delta W$$

Represent ΔW as a low-rank product

$$\Delta W = BA$$

$A \in R^{r \times d_{in}}$ $B \in R^{d_{out} \times r}$ r(rank) is small (eg 4 or 8)

So we only train A and B, not the whole W.

→ Huge memory savings, no extra latency inference

In practice:

Work great on GPT, BERT, and T5 models and often used in attention layer

Advantage

similar or better accuracy than full fine-tuning with <1% of parameters.