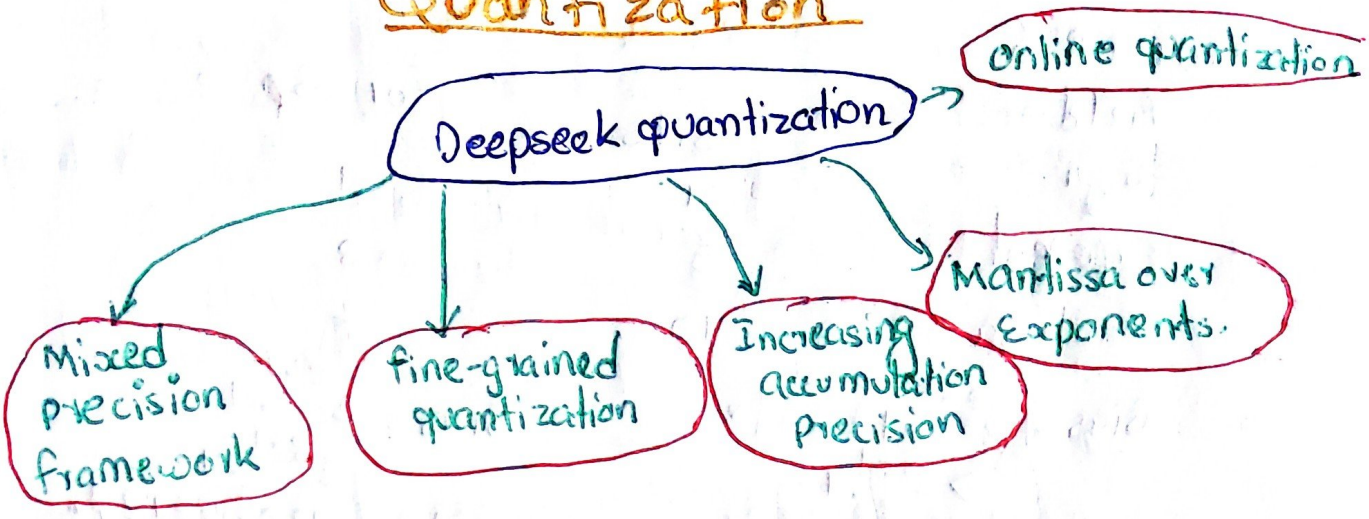
Quantization



How a number is stored in FP32??

↳ According to IEEE-754 standart.

field	Bits
Sign	1
Exponent(e)	8
Mantissa	23.

for FP32 bias = 127.

suppose $x = 3.14159274101257 32$.

Binary(approx) = 11.00100100001111 1111

Normalized = $1.10010010000111111 \times 2^1$

$= (-1)^{sign} (1 + \text{mantissa}) \times 2^{e-127}$ (formulae)

~~accompanying~~

stored 32 bit

0 | 1 0 0 0 0 0 0 0 | 10010010000111111

1 bit | 8 bit | 23 bits

To find the exponent bit we compare the formulae with the Normalized binary of the number.

we get, $e - 127 = 1$

$e = 128$.

↙ we add bias to minimize the negative to occur.

For FP16

field	Bits
Sign	1
Exponent	5
Mantissa	10

$Bias = 2^{e-1} - 1$

$\Rightarrow 2^{5-1} - 1$

$= 15.$

for Bf16

	1
	8
	7

$Bias = 2^{e-1} - 1$

$= 2^{8-1} - 1$

$= 128 - 1 = 127.$

Mixed Precision Framework

↳ Suppose $y = Wx$.

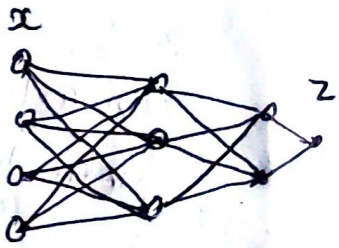
computed initially in FP32 for numerical stability then converted back to BF16.

↳ converted to FP8 from BF16.

↳ maintained in higher precision (BF16 or FP32), then converted to FP8 on the fly. (Fprop in paper figure).

$y_{FP32} = W_{FP8} \times x_{FP8}$

↳ stored in BF16 to optimize memory.



In the middle of a neural network,

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} * W^T$$

↳ it comes as $z = Wx$

$\frac{\partial z}{\partial x} = W$

we know, $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \times \frac{\partial z}{\partial x} = \frac{\partial L}{\partial z} \times$

To maintain dimension in back propagati

computed initially as FP32, then converted to BF16.

stored as BF16 but converted to FP8 for computation.

converted on the fly to FP8 for this calculation.

Dgrad in paper

$$\frac{\partial L}{\partial x_{FP32}} = \frac{\partial L}{\partial z_{FP8}} * W^T_{FP8}$$

↳ stored as BF16.

weights are at stored at higher precision so that we can confirm, there's update in it. it's learning!!

w grad in paper

$$\frac{dL}{dW} = X^T \frac{\partial L}{\partial Z}$$

- X^T: stored in FP8. ↳ converted from BF16 to FP8
- $\frac{dL}{dW}$: computed & stored in FP32 to ensure numerical stability for weight updates.

Why to do in such approaches?

FP8 Precision

↳ significantly reduces computational & memory cost.
↳ Provides up to 2x speed improvement over BF16 operations.
↳ Ideal for computationally intensive GEMM operations.

Higher Precision Retained (FP32/BF16):

1) Embedding modules, output heads, gating modules, normalization ~~operator~~ attention operators are sensitive. Hence, these are retained in higher precision.
2) Weights & optimizer states also stored in FP32 ensure training stability.

This balance of high-precision storage with low-precision calculation allows the best trade-off between computational efficiency, memory reduction & numerical stability.

b) Fine grained quantization

↳ This method was introduced to protect the precision of weight & dimension when they are quantized to lower precision.

* If any outlier occur it make the number with different value in quantization & dequantization.

Suppose: $x = [2, 3, 4, 500] \times \frac{127}{500} = [0.508, 0.762, 1.016, 127.0]$

Now, quantize to FP8: ↳ assume E4M3 quantization.

$= [0.5, 0.75, 1.0, 127]$

Dequantize back

$$\text{Recovered Tensor} = \frac{[0.5, 0.75, 1.0, 127] \times 500}{127}$$

$\approx [1.97, 2.95, 3.94, 500]$

This occur due to outlier.

So, what to do??

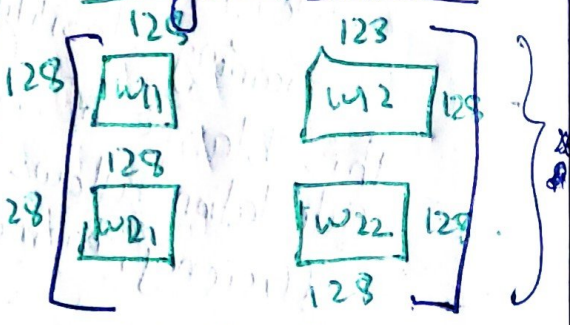
we have billion parameters in weight, in h_dim also, we have large no. of value, so, we will group them in group & the do quantization, so, that outlier in one group didn't affect the precision of another group.

group 1 | group 2

$x = [1.2, 2.1, 2.001, 3 \;|\; 76, 0.1, 1.2, 3]$

scaled by factor 1 | scaled by factor 2.

weight 256 dim



w_{11} → scaling factor 1
w_{12} → scaling factor 2
w_{21} → scaling factor 3
w_{22} → scaling factor 4.

$y = x \times w$

Tensor core. (General matrix mul occur in FP8 inside NVIDIA Tensor core.)

In cuda core.
FP8 * upscaling factor = FP_{32}

~~low precision~~

Increasing accumulation Precision

↳ Using higher-precision arithmetic only for the accumulation (summing) step during neural-network computations. Even though the inputs, weights, & outputs are stored in lower precision.

We know, in almost every neural network layer:

$$y_i = \sum_{j=1}^{d} x_j \cdot w_{ji}$$

This has two different operation.

i) Multiply → $x_j * w_{ji}$
ii) Accumulate (sum) → $\sum_j (\cdot)$.

Deepseek-V3 keeps (1) in low precision & (2) in higher precision.

why low-precision accumulation is dangerous??

↳ suppose Deepseek uses FP8/BF16 for speed.

problem:

↳ FP8/BF16 has few mantissa bits.

↳ when summing thousands of terms, small values get.

- rounded away
- lost entirely.

This causes:

- large numerical error.
- unstable training.
- exploding / vanishing gradients.

WGMMA

↳ wrap group matrix Multiply Accumulate.

↳ It's a low-level GPU instruction family introduced with NVIDIA Hoppe (H100) & it's central to how models like V3 run FP8 efficiently.

key: multiplication in low precision in tensor core & accumulation is done in FP32 in cuda core.

Mantissa over Exponents

In FP8 there are two format → $E4M3$ → 3 mantissa bit higher presion
→ $E5M2$. → lower presicision.
↓ 5 exponent bit high range

Deepseek prefer
Always use
E_4M_3 format.

but prior works prefers:

* Forward pass : E_4M_3
* Backward pass : E_5M_2.

Online quantization

↳ Scaling factor is decided based on the numbers that actually come in.

main problem of offline quantization (Delayed quantization)
↳ we run a callibration dataset and decide the scaling factor at once & after that we do quantization based on that.

* If a rare token appears, overflow or loss.

What deepseek does?
↳ it calculate the scaling factor for the fly comming Tensors.

* In inference, each weight tensor has one (or a small set of) scale value(s), computed ONCE, & reused for all tokens.

so, in inference, we use offline quantization.