QLoRA: Efficient Finetuning of Quantized LLMs

Presenter: Ruize Gao

11/6/2024

Overview

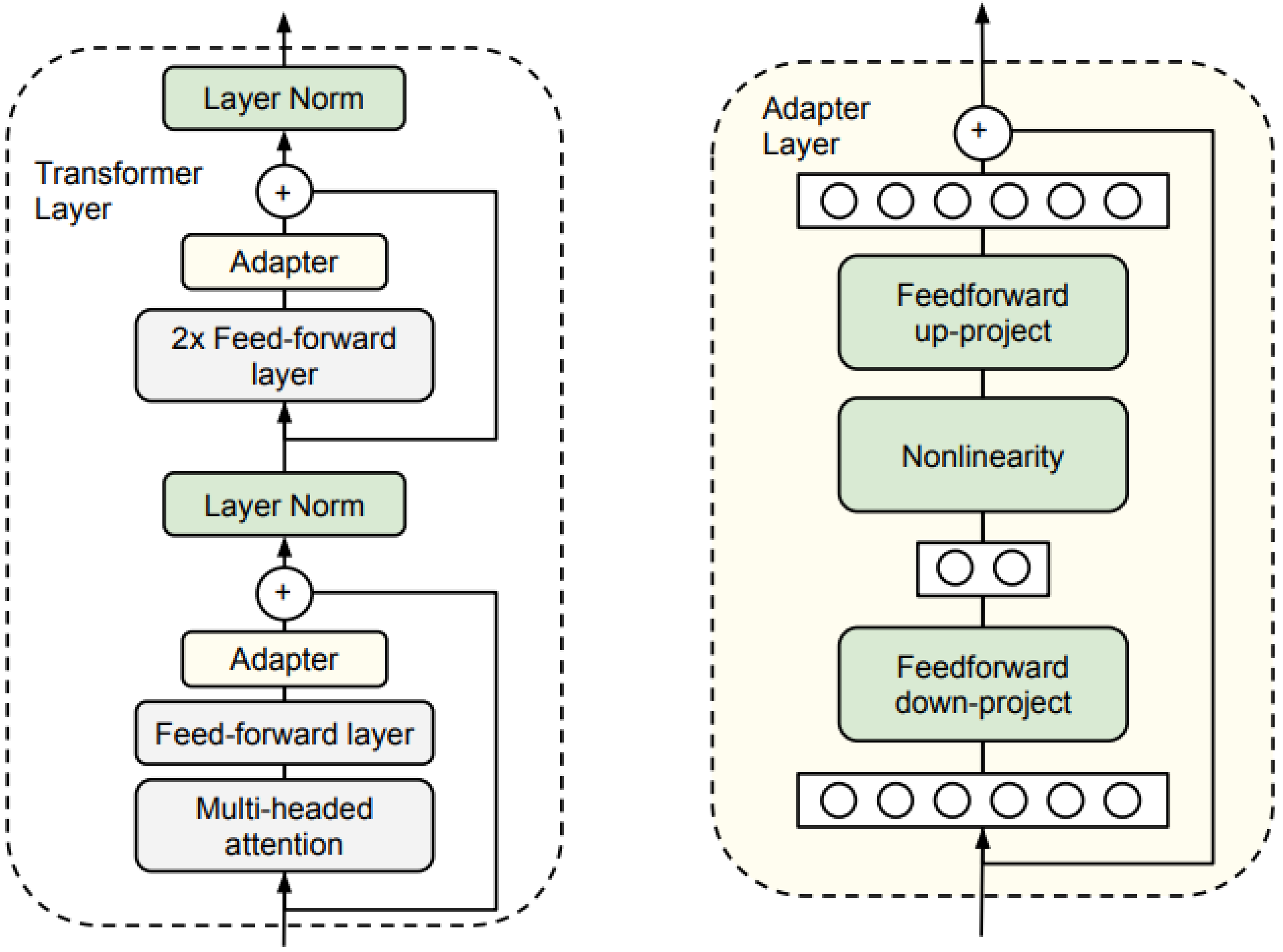
- **Background**: Parameter-Efficient Fine-Tuning (PEFT)
- **Motivation**: Why PEFT, LoRA, and QLoRA?
- **Contribution**: In what ways does QLoRA beat other PEFTs?
- **Methodology**: How does QLoRA beat other PEFTs?
- **Experiments and Results**

Background: PEFT

- Motivation:
  - We need fine-tuning for downstream tasks
  - But fine-tuning the whole model is unrealistic
- Prior Work:
  - Adapter Layer (Houlsby et al., 2019)
  - Prefix Tuning (Li and Liang, 2021)
  - Prompt Tuning (Lester et al., 2021)
  - BitFit (Ben-Zaken et al., 2021)

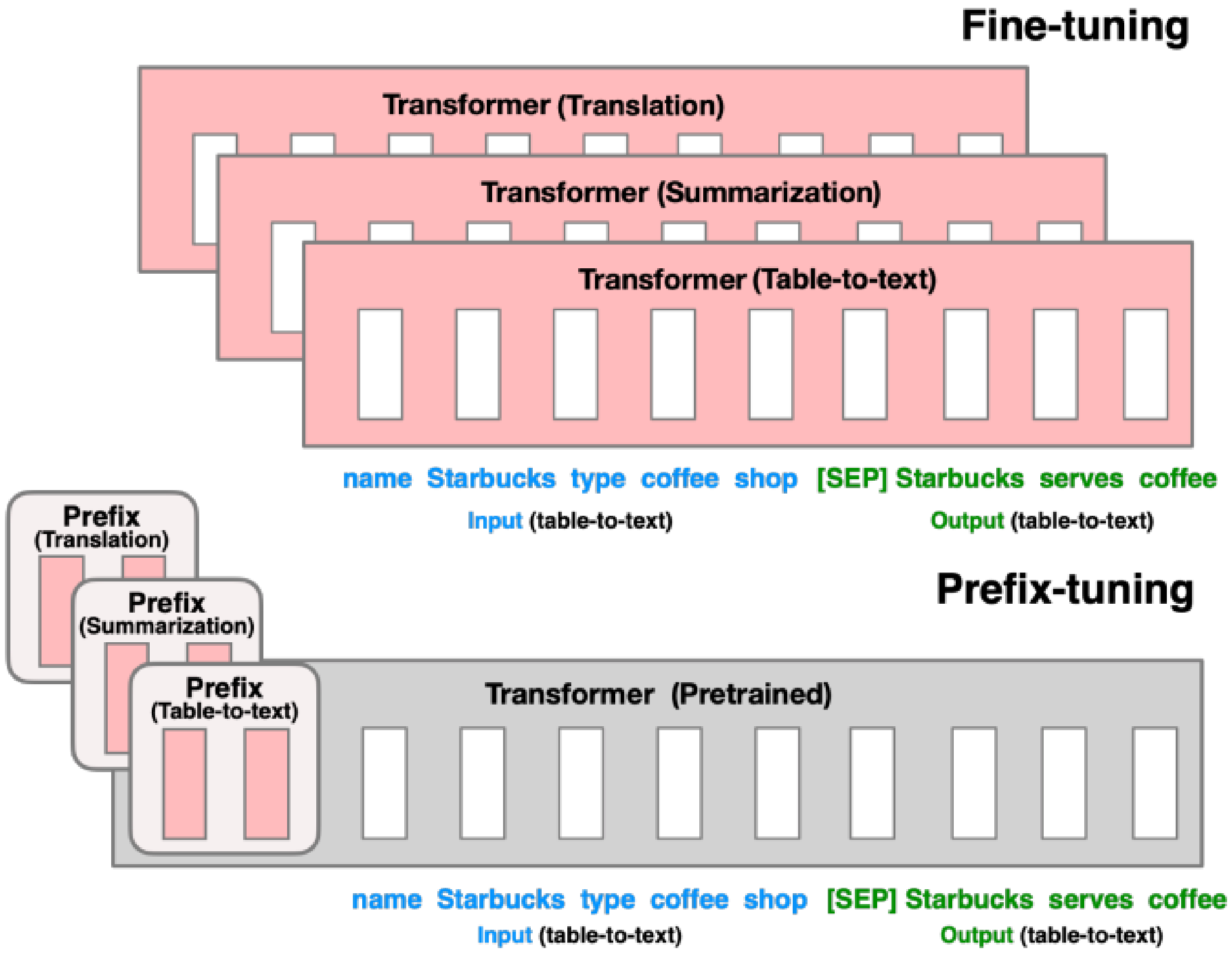
Background: Adaptor Layer

- Extra Layer per block
- Bottleneck architecture
- Inference latency



Background: Prefix-Tuning

- Prefix parameters per layer
- "Steering" the model
- Unstable training
- Wasted Seq Length



Background: Other Directions

- Prompt Tuning
    - Pre-append soft prompt params to input
- BitFit
    - Tune only the bias
- …
- LoRA

Background: Low Rank Adaptation (LoRA)

- Motivation:
  - Limitations of prior research
  - Intrinsic dimensionality (Aghajanyan et al., 2020)
- Contribution:
  - LoRA converges to full fine-tuning
  - Can choose any subset of weights for tuning
  - No inference latency

Background : LoRA

- Intuition:
  - Fine-tuning works b/c low "intrinsic dimentionality" (Aghajanyan et al., 2020)
  - Weight updates can be decomposed into two matrices with lower ranks

$$W = W_0 + \Delta W$$

$$\Delta W \approx B \times A$$

$$\Delta W \in \mathbb{R}^{d \times r}$$

$$B \in \mathbb{R}^{d \times r}$$

$$A \in \mathbb{R}^{r \times d}$$

Background : LoRA

- Example 1 (d=3, r=2):

$$\Delta W = \begin{bmatrix} 3 & 1 & 2 \\ 4 & 2 & 1 \\ 5 & 3 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} \quad A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix}$$

$$B \times A = \begin{bmatrix} 1 & 0 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 2 \\ 4 & 2 & 1 \\ 5 & 3 & 3 \end{bmatrix}$$

Background : LoRA

- Example 1 (d=3, r=1):

$$\Delta W = \begin{bmatrix} 8 & 16 & 4 \\ 4 & 8 & 1 \\ 12 & 24 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} \quad A = \begin{bmatrix} 4 & 8 & 2 \end{bmatrix}$$

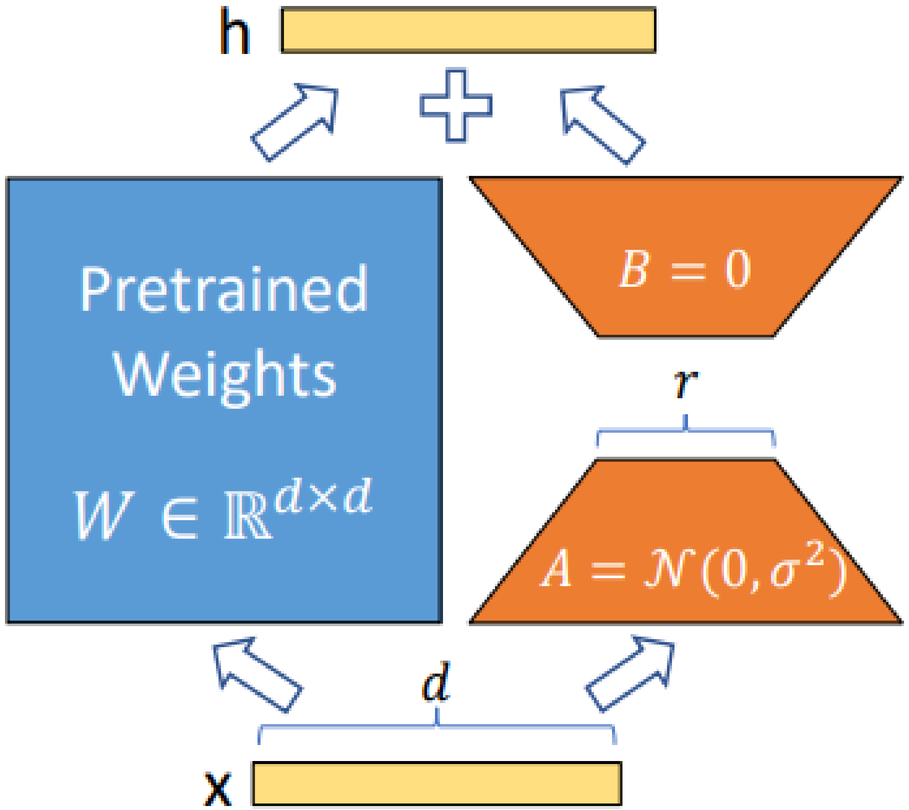
$$B \times A = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} \begin{bmatrix} 4 & 8 & 2 \end{bmatrix} = \begin{bmatrix} 8 & 16 & 4 \\ 4 & 8 & 1 \\ 12 & 24 & 6 \end{bmatrix}$$

Background : LoRA

- Implementation:
  - B initialized to zero
  - A initialized to Gaussian Noise
  - r is a hyperparameter
  - Update B & A rather than ΔW

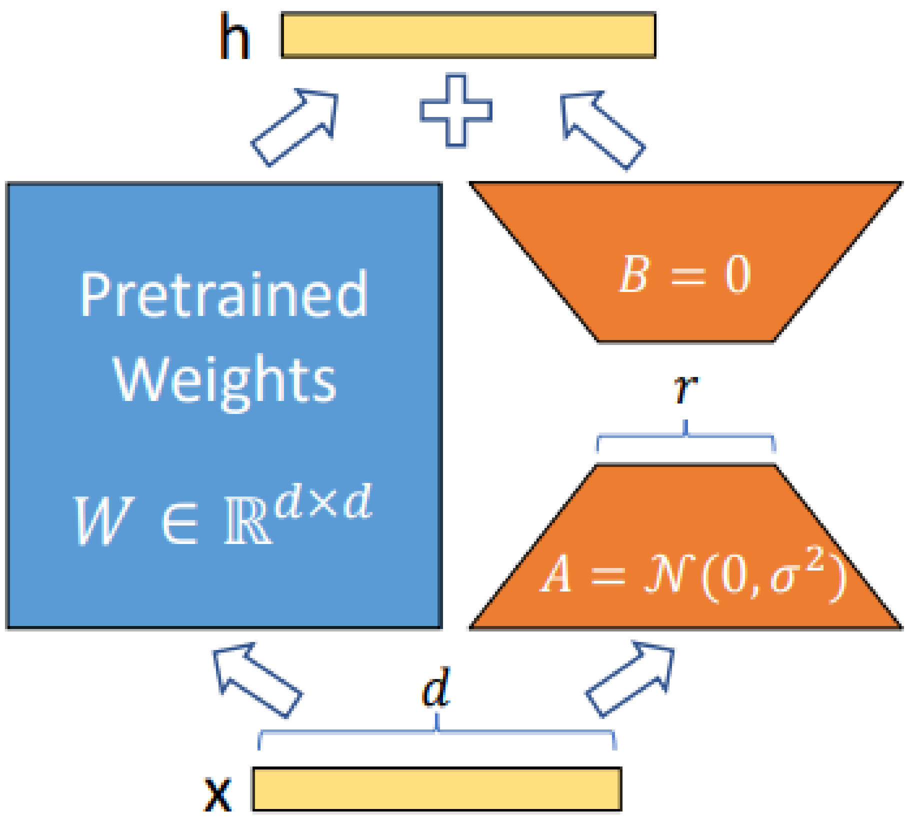
$$\Delta W \in \mathbb{R}^{d \times r} \qquad B \in \mathbb{R}^{d \times r} \qquad A \in \mathbb{R}^{r \times d}$$

$$h = W_0 x + \Delta W x = W_0 x + BAx$$



Background : LoRA

- Results
  - Update only the attention weights
  - Reduce VRAM by 2/3
  - Reduce checkpoint memory by 10,000x (r=4)
  - 175B trainable params down to 4.7M



Overview

- **Background**: Parameter-Efficient Fine-Tuning (PEFT)
- **Motivation**: Why QLoRA?
- **Contribution**: In what ways does QLoRA beat other PEFTs?
- **Methodology**: How does QLoRA beat other PEFTs?
- **Experiments and Results**

Motivation for QLoRA

- LoRA is still not enough

	Weight/Param	Weight Gradient/Param	Optimizer State/Param	Adapter Weights/Param	Total/Param	70B-Param Model
Full FT	16 bits	16 bits	64 bits	N/A	96 bits	840 GB
LoRA	16 bits	0.4 bit	0.8 bit	0.4 bits	17.6 bits	154 GB

Contribution of QLoRA

- Reduce weight per param
- Fit the training process within 2x consumer GPUs

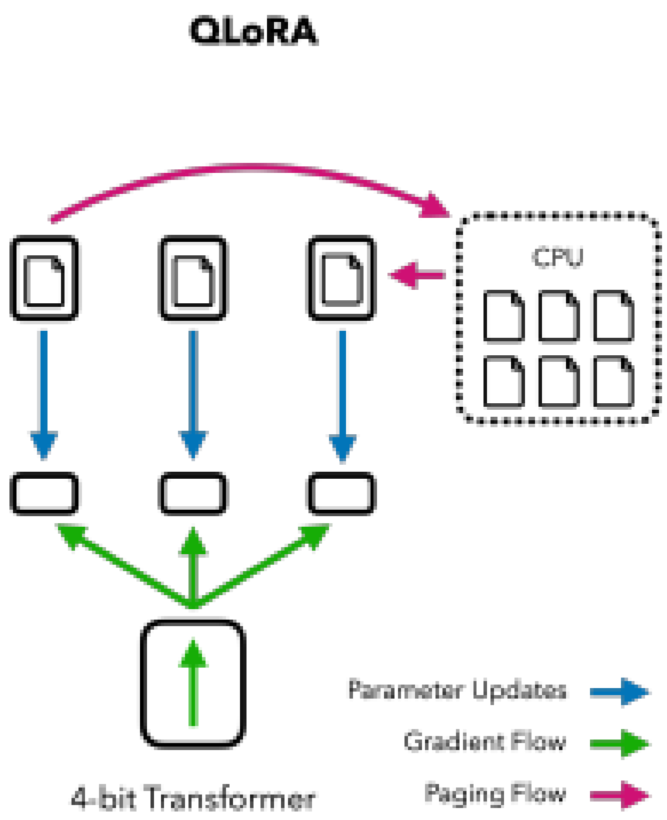
	Weight/Param	Weight Gradient/Param	Optimizer State/Param	Adapter Weights/Param	Total/Param	70B-Param Model
Full FT	16 bits	16 bits	64 bits	N/A	96 bits	840 GB
LoRA	16 bits	0.4 bit	0.8 bit	0.4 bits	17.6 bits	154 GB
QLoRA	4 bits	0.4 bit	0.8 bit	0.4 bit	5.6 bits	46 GB

Challenges

- Under-utilization of quantization bins due to outliers
- Large quantization constants still cost some memory
- Sudden memory spikes can cause CUDA out of memory

Methodology

- 4-bit NormalFloat Quantization
  - Deal with weight outliers
- Double Quantization
  - Reduces quantization constant memory
- Page Optimizers
  - Handle occasional memory spikes

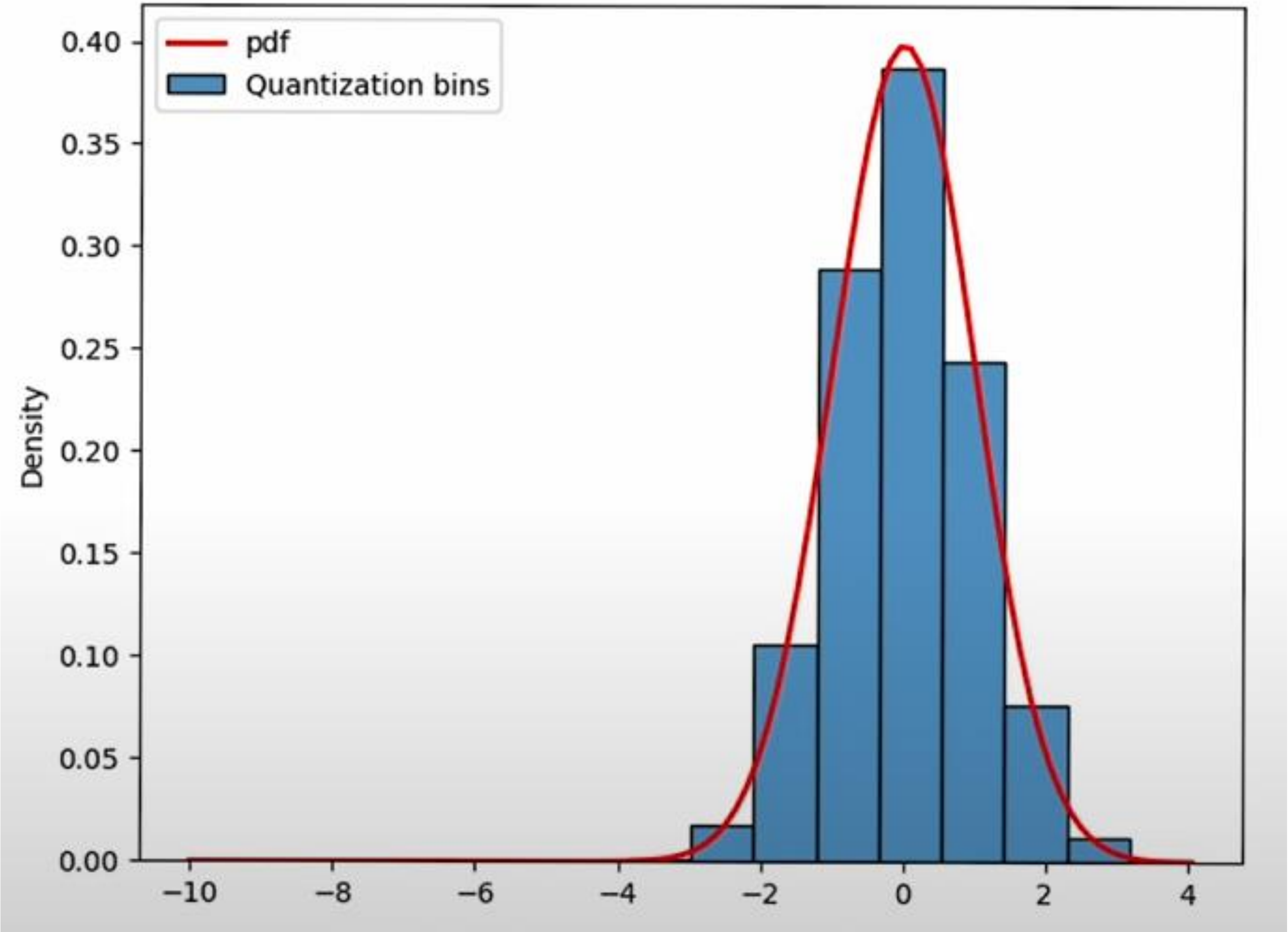


Methodology: 4-bit NormalFloat Quantization

- Conventional Quantization
  - Scale down tensor by absmax (normalization)
  - Find the closest value in the target data type
  - Example with 2-bit data type:
    - Input: [10, -3, 5, 4]; target: [-1.0, -0.3, 0.5, 1.0] with index [0, 1, 2, 3]
    - Normalize: [1.0, -0.3, 0.5, 0.4]
    - Find the closest value: [1.0, -0.3, 0.5, 0.5] with index [3, 1, 2, 2]
    - Dequantization: with index [3, 1, 2, 2], find [1.0, -0.3, 0.5, 0.5], then scale up to [10, -3, 5, 5]

Methodology: 4-bit NormalFloat Quantization

- Limitation of Conventional Quantization
  - 4-bit quantization of tensor with outlier at –10
  - Only half of the bins are used

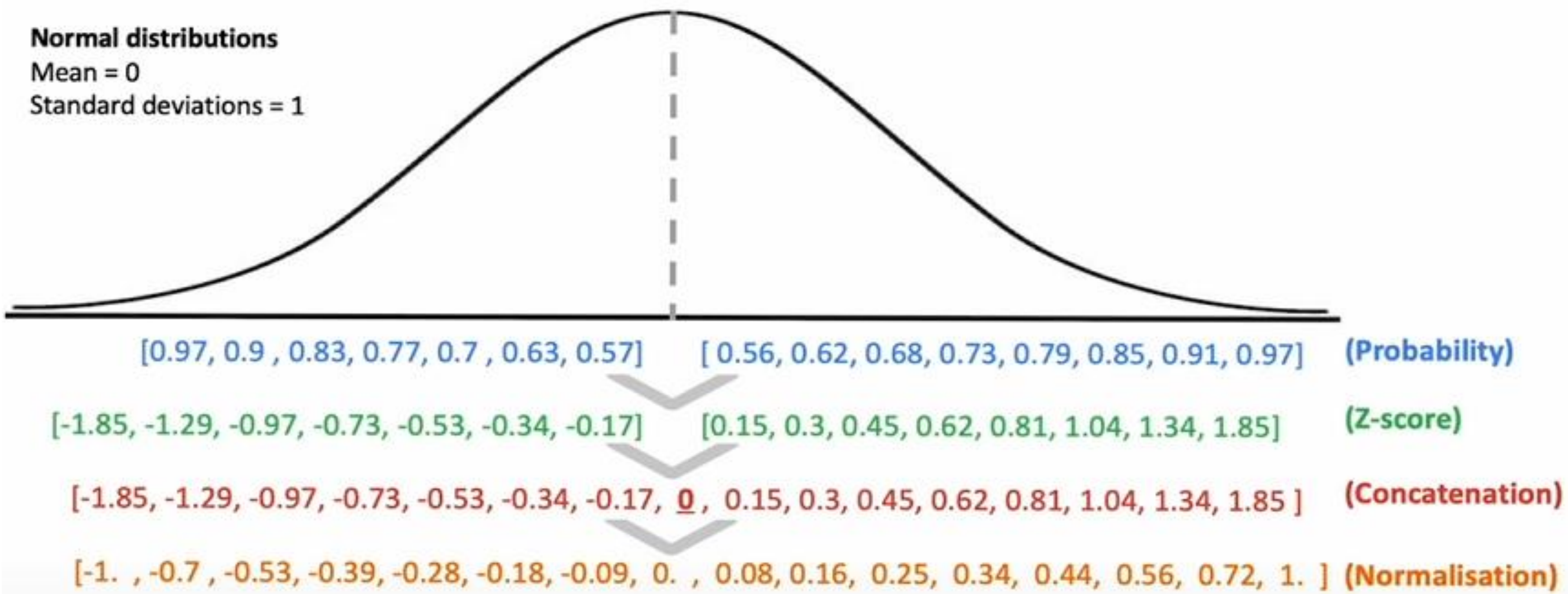


Methodology: 4-bit NormalFloat Quantization

- 4-bit NormalFloat Quantization
  - Flatten each tensor and divide into chunk
  - Find quantization constant for each chunk
  - Quantize each chunk individually

Methodology: 4-bit NormalFloat Quantization (cont.)

- An implementation detail
  - Use asymmetric quantization for zero padding



Methodology: Double Quantization

- Memory cost of quantization constant
  - FloatPoint-32 with blocksize 64
    - 64 weight params get quantized together with a FP32 constant
  - 32 bits / 64 params = 0.5 bit / param
- Can we further reduce this cost?

Methodology: Double Quantization (cont.)

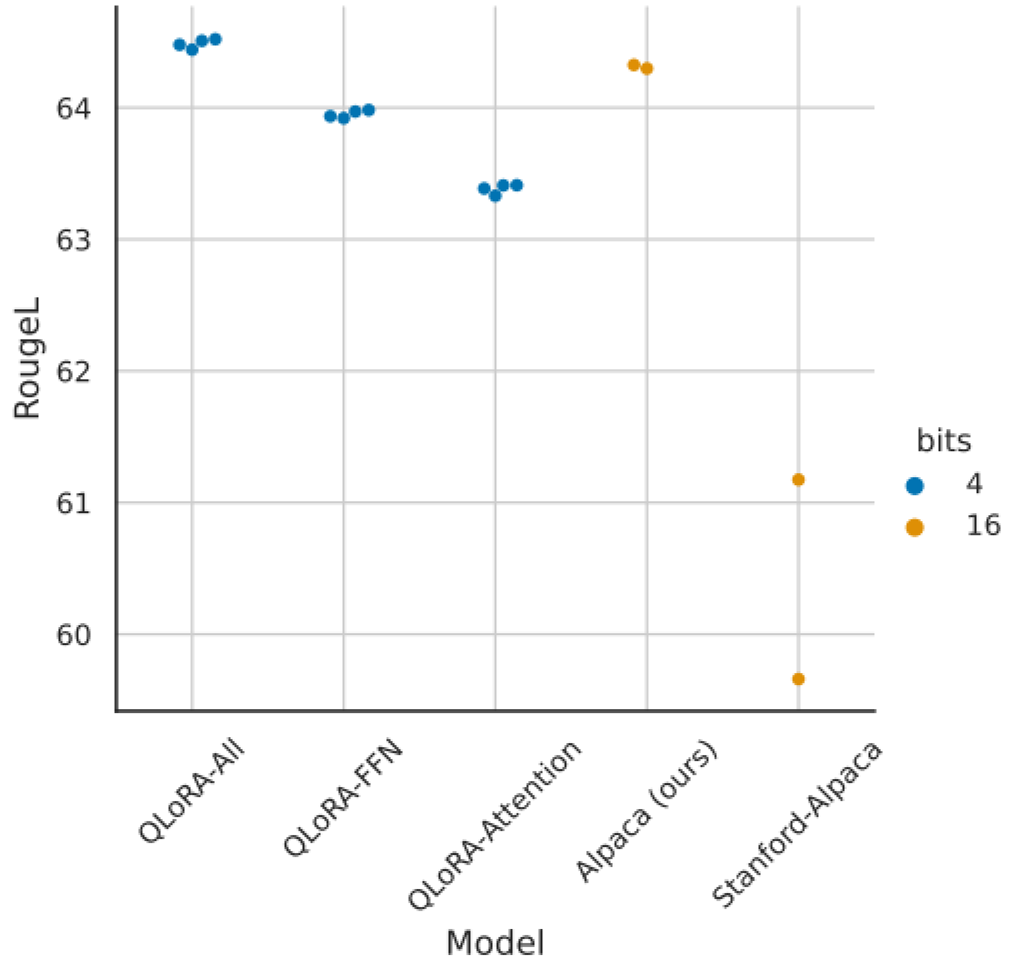
- Double quantization
  - Quantize the quantization constant of weight params
  - FloatPoint-8 with blocksize 256
  - 8 / 64 + 32 / (64 * 256) = 0.127 bit / param
- Cost reduce = 0.5 - 0.127 = 0.373 bit /param

Methodology: Paged Attention

- Black-box NIVIDIA memory feature
- Automatic page-to-page memory transfer between CPU and GPU
- Evict optimizer states memory to CPU when CUDA out of memory

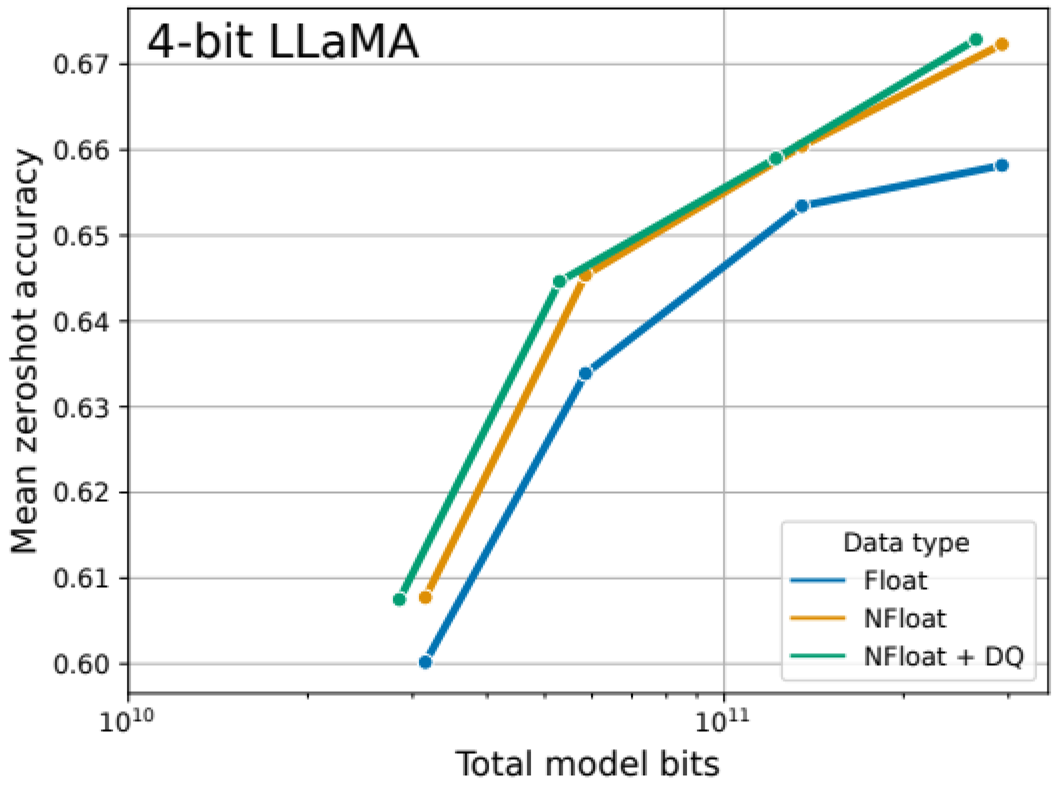
Experiments

- Setup:
  - LLaMA 7B model
  - RougeL: longest matching seq
- Important takeaway:
  - All layers must be tuned to match full fine-tuning performance



Experiments (cont.)

- Ablation Study
  - 4-bit Float vs. 4-bit NormalFloat vs. 4-bit NormalFLoat + Double Quantization
- Important takeaway:
  - 4-bit NormalFloat is both theoretically and emprically effective



Experiments (cont.)

- Multitask Benchmark
  - LLaMA models tuned with different adapters
- Important Takeaway
  - NFloat64 + DQ can replicate 16-bit fine-tuning performance

	Mean 5-shot MMLU Accuracy								
LLaMA Size	7B		13B		33B		65B		Mean
Dataset	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	
BFloat16	38.4	45.6	47.2	50.6	57.7	60.5	61.8	62.5	53.0
Float4	37.2	44.0	47.3	50.0	55.9	58.5	61.3	63.3	52.2
NFloat4 + DQ	39.0	44.5	47.5	50.7	57.3	59.2	61.8	63.9	53.1

Experiments (cont.)

- Setup:
  - MMLU 5-shot
  - Multiple instruction datasets
- Important takeaway:
  - FLAN v2 is the best
  - Dataset matters less as model size increases

Dataset	7B	13B	33B	65B
LLaMA no tuning	35.1	46.9	57.8	63.4
Self-Instruct	36.4	33.3	53.0	56.7
Longform	32.1	43.2	56.6	59.7
Chip2	34.5	41.6	53.6	59.8
HH-RLHF	34.9	44.6	55.8	60.1
Unnatural Instruct	41.9	48.1	57.3	61.3
Guanaco (OASST1)	36.6	46.4	57.0	62.2
Alpaca	38.8	47.8	57.3	62.5
FLAN v2	44.5	51.4	59.2	63.9

Experiments (cont.)

- Chatbot FT
- FLAN v2 is the worst
- Guanaco is the best

Model / Dataset	Params	Model bits	Memory	ChatGPT vs Sys	Sys vs ChatGPT	Mean	95% CI
GPT-4	-	-	-	119.4%	110.1%	**114.5%**	2.6%
Bard	-	-	-	93.2%	96.4%	94.8%	4.1%
Guanaco	65B	4-bit	41 GB	96.7%	101.9%	**99.3%**	4.4%
Alpaca	65B	4-bit	41 GB	63.0%	77.9%	70.7%	4.3%
FLAN v2	65B	4-bit	41 GB	37.0%	59.6%	48.4%	4.6%
Guanaco	33B	4-bit	21 GB	96.5%	99.2%	**97.8%**	4.4%
Open Assistant	33B	16-bit	66 GB	91.2%	98.7%	94.9%	4.5%
Alpaca	33B	4-bit	21 GB	67.2%	79.7%	73.6%	4.2%
FLAN v2	33B	4-bit	21 GB	26.3%	49.7%	38.0%	3.9%
Vicuna	13B	16-bit	26 GB	91.2%	98.7%	**94.9%**	4.5%
Guanaco	13B	4-bit	10 GB	87.3%	93.4%	90.4%	5.2%
Alpaca	13B	4-bit	10 GB	63.8%	76.7%	69.4%	4.2%
HH-RLHF	13B	4-bit	10 GB	55.5%	69.1%	62.5%	4.7%
Unnatural Instr.	13B	4-bit	10 GB	50.6%	69.8%	60.5%	4.2%
Chip2	13B	4-bit	10 GB	49.2%	69.3%	59.5%	4.7%
Longform	13B	4-bit	10 GB	44.9%	62.0%	53.6%	5.2%
Self-Instruct	13B	4-bit	10 GB	38.0%	60.5%	49.1%	4.6%
FLAN v2	13B	4-bit	10 GB	32.4%	61.2%	47.0%	3.6%
Guanaco	7B	4-bit	5 GB	84.1%	89.8%	**87.0%**	5.4%
Alpaca	7B	4-bit	5 GB	57.3%	71.2%	64.4%	5.0%
FLAN v2	7B	4-bit	5 GB	33.3%	56.1%	44.8%	4.0%

Future Directions

- Fast 4-bit inference
- Better chatbot evaluation