LLM Sys

Parameter Efficient Fine-Tuning for LLM

Lei Li

Language Technologies Institute | Carnegie Mellon University School of Computer Science

Ack: Kath Choi, Jiaqi Song, Xianwei Zou, Hanshi Sun, Steven Kolawole

Recap

- Direct quantization for low-bit numbers
  - absmax: linearly scale according to max abs value
  - zero-point: finding zero-point and scale
- Layer-wise quantization approaches
  - AdaQuant / KD: ZeroQuant / LLM.int8()
- GPTQ
  - layer-wise quantization + compensation for errors + precompute
  - accurately compress some of the largest publicly-available models down to 3 and 4 bits, and bring end-to-end speedups

Outline

- Overview of Parameter Efficient Fine-Tuning
- LoRA: Low-rank Adaptation (or Counter-interference adapter, CIAT)
- QLoRA: Quantization + Low-rank training
- Code Walkthrough

LLM Fine-tuning is Expensive

- Full-parameter Fine-Tuning
  - Update all model parameters → Require large GPU memory
- e.g. Harf-precision Fine-tuning cost per parameter (Adam)
  - Weight: 16 bits (2 bytes) * N
  - Weight Gradient: 16 bits (2 bytes) * N
  - Optimizer States: 2 * 16 bits (4 bytes) * N
  - Activations: ~ 1-2x of parameters
  - LLaMA-8B ➔ ~ 80GB working memory

Parameter Efficient Fine-tuning (PEFT)

- Only update a small subset (or low-rank) of parameters
- e.g. Fine-tuning cost per parameter with LoRA
  - Weight: 16 bits
  - Weight Gradient: ~0.4 bits
  - Optimizer State: ~0.8 bits
  - Adapter Weights: ~0.4 bits
  - Activations: ~ 1-2x of parameters
  - LLaMA-8B ➔ ~ 33GB working memory, fits a single A100.

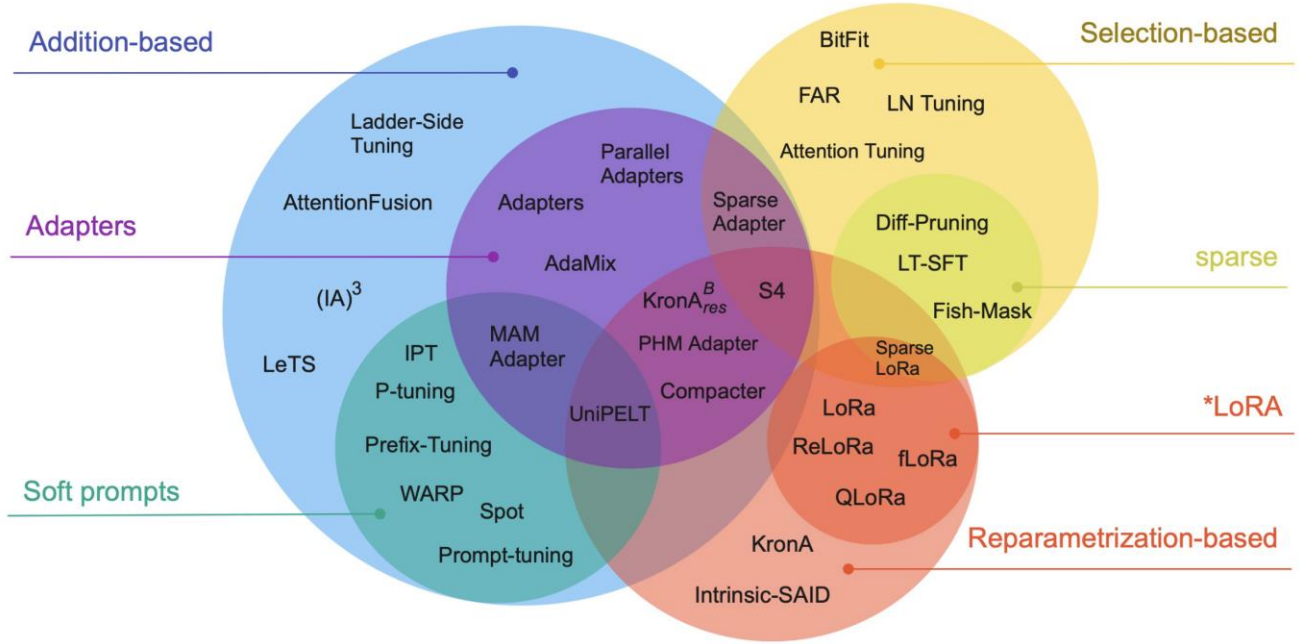
PEFT with Quantization

- Parameter efficient fine-tuning together with quantization
- e.g. Fine-tuning cost per parameter with QLoRA
  - Weight: 4 bits
  - Weight Gradient: ~0.4 bit
  - Optimizer State: ~0.8 bit
  - Adapter Weights: ~0.4 bit
  - Activations: ~ 1-2x of parameters
  - LLaMA-8B ➔ ~ ~~33GB~~ 9.2GB working memory

Dettmers et al. QLoRA: Efficient Finetuning of Quantized Large Language Models. 2022.

PEFT Approaches

- Selective Methods:
  - fune-tune selected subsets
- Reparameterization Method:
  - low-rank representation weights
- Additive Methods:
  - add trainable layers or parameters
  - e.g. Adapters, Soft prompts (Prompt Tuning)



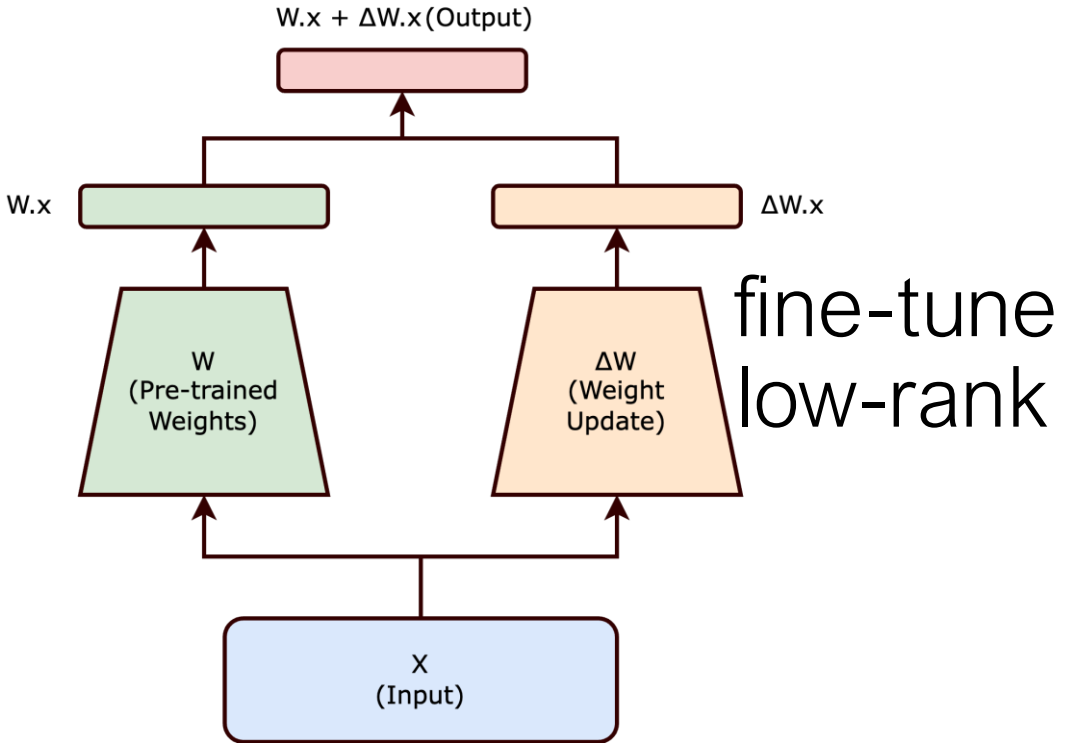
Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning. Lialin et al. 2024.

Outline

- Overview of Parameter Efficient Fine-Tuning
- LoRA: Low-rank Adaptation (Counter-interference adapter, CIAT)
- QLoRA: Quantization + Low-rank training
- Code Walkthrough

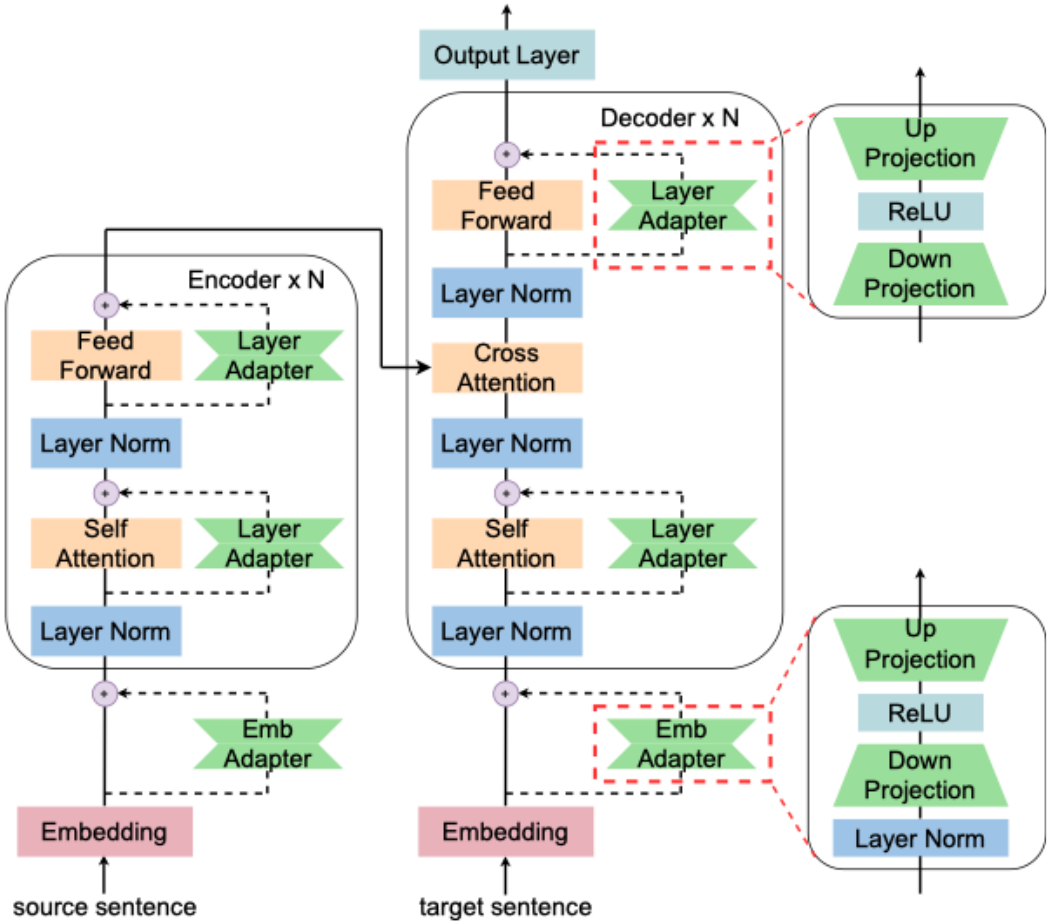
Additive Fine-tuning

- Freeze the pre-trained parameters W_0
- Train a low-rank update to the parameters δ_W
  - $W' = W_0 + \delta_w$



Low-rank Adapation (LoRA) for Fine-tuning

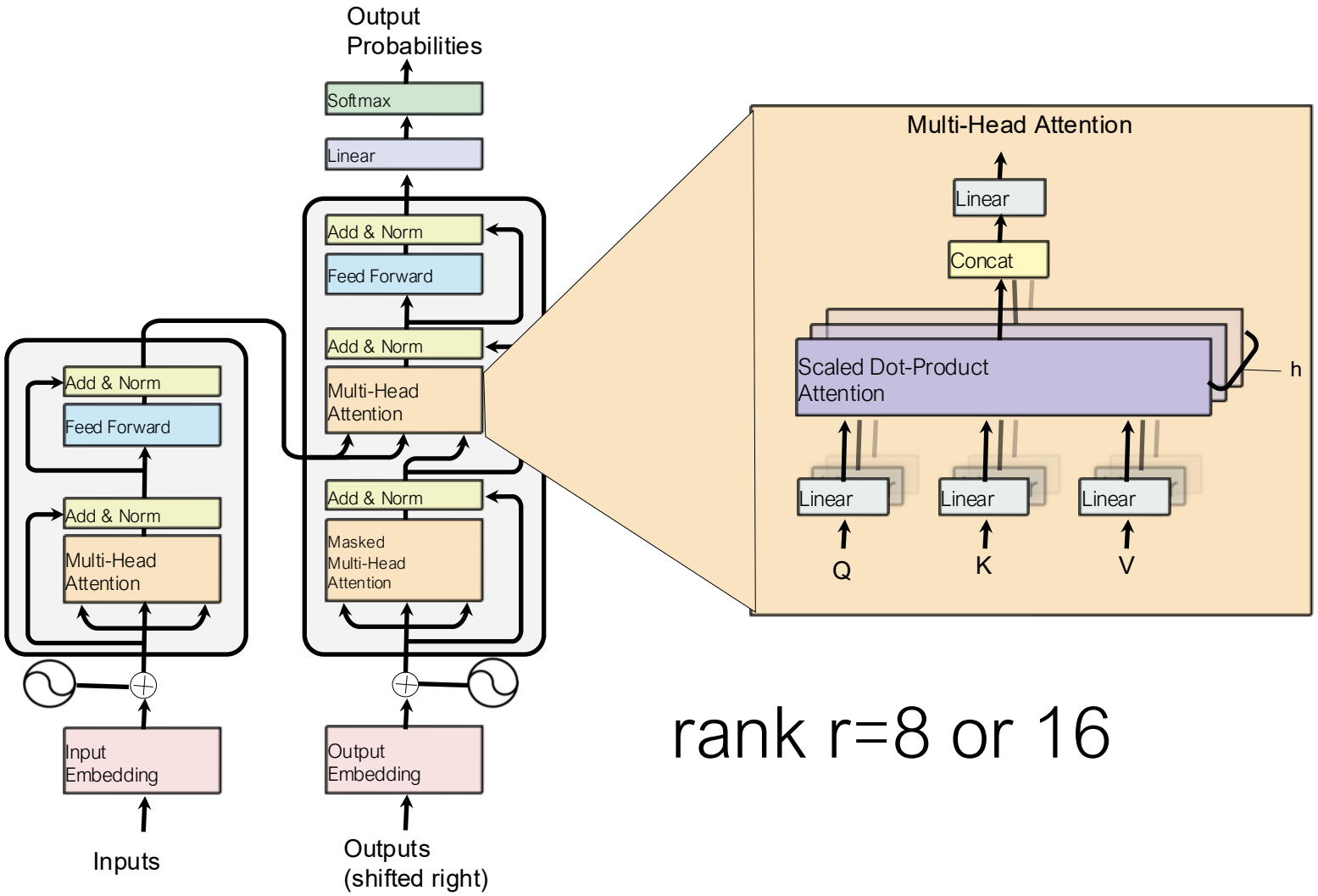
first invented in **Counter-Interference Adapter (CIAT) for Multilingual Machine Translation** [Zhu et al, EMNLP 2021], later re-invented by **Low-Rank Adaptation of Large Language Models** [Hu et al, ICLR 2022]



$$W' = W + A_{d\times r} \cdot B_{r\times d}$$

$$\text{low rank } r \ll d$$

Applying LoRA/CIAT to LLM



rank r=8 or 16

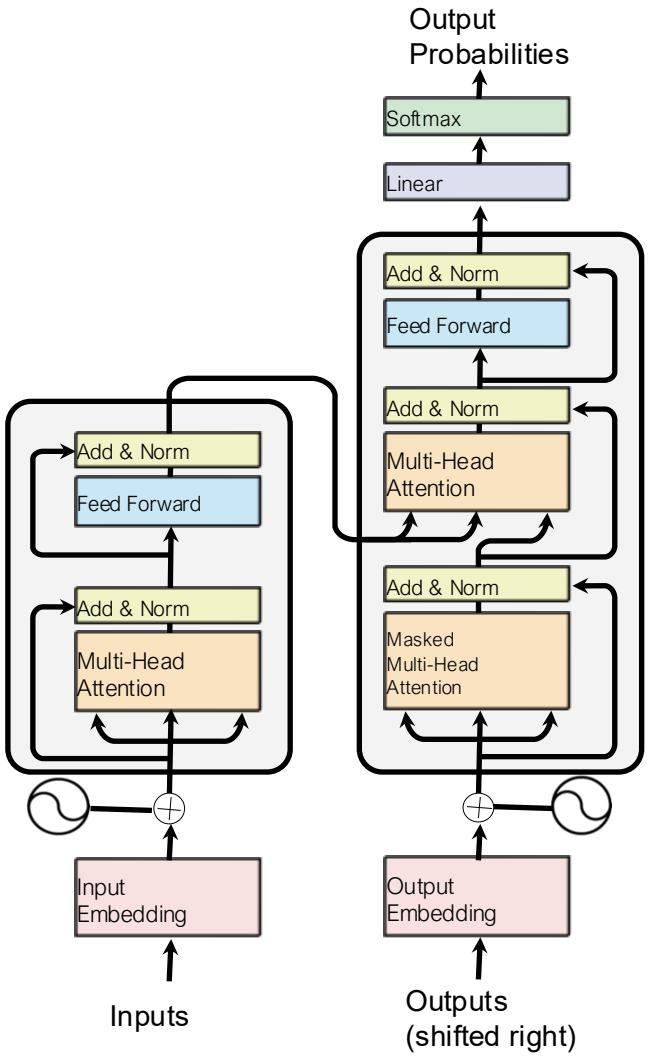
Apply LoRA to weights in attention layer

$$W_Q, W_K, W_V, W$$

but not to FFN's linear weights (storing knowledge)

In CIAT method, it is also applied to embeddings and FFN layers, and improves.

Inference for LoRA-trained models



- Inference: use $W' = W + A \cdot B$ for each weight matrix and store A,B as well (small additional cost, r=8 or 16)

- We can switch between LoRA weight for each task/domain
  - $W'' = W' - A \cdot B + A'' \cdot B''$

LoRA weights for current domain

LoRA weights for new domain

Counter-Interference Adapter (CIAT) for Multilingual Machine Translation [Zhu et al, EMNLP 2021]
Low-Rank Adaptation of Large Language Models [Hu et al, ICLR 2022]

Backward Computation of LoRA/CIAT

$$W' = W_0 + A \cdot B$$

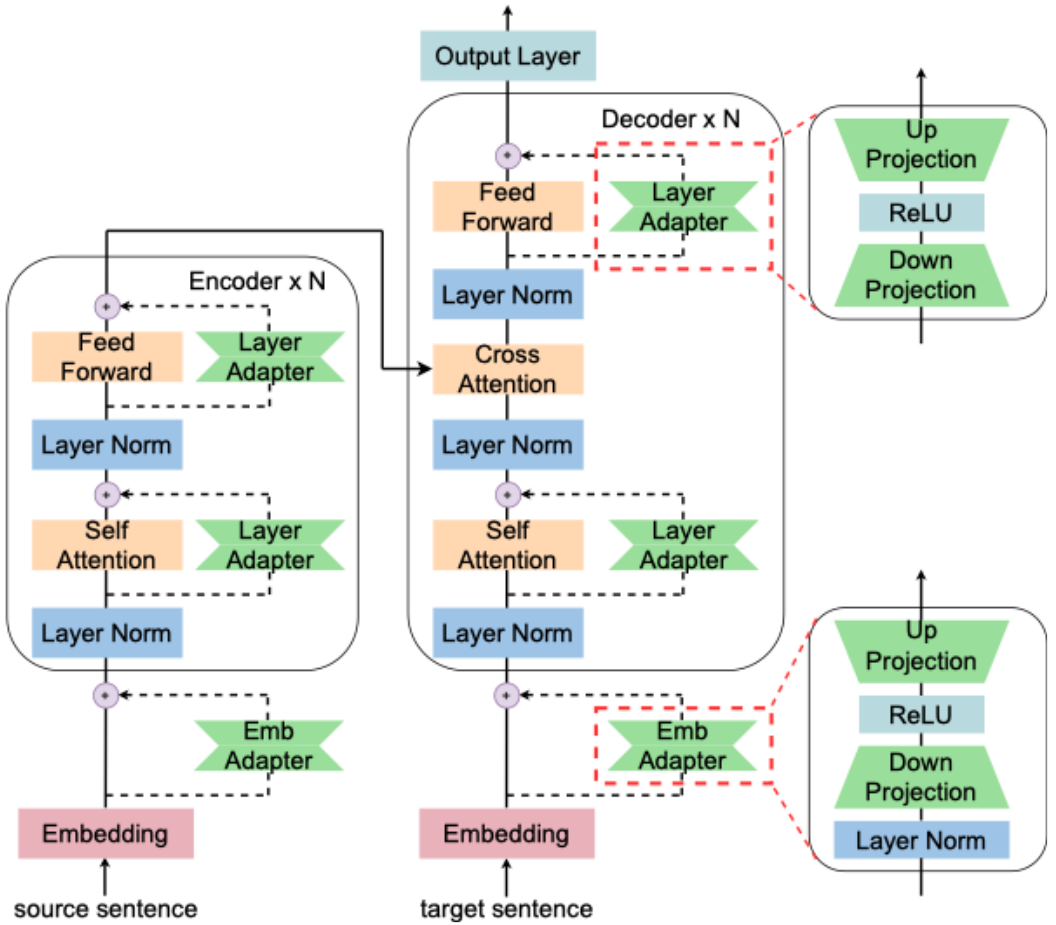
- The original pre-trained weight matrix is fixed. We only need to compute the gradients with respect to A and B.

$$\frac{\partial L}{\partial A} = g_{out} \cdot \frac{\partial (W_0 x + A \cdot Bx)}{\partial A} = g_{out} \cdot (Bx)^T$$

$$\frac{\partial L}{\partial B} = g_{out} \cdot \frac{\partial (W_0 x + A \cdot Bx)}{\partial B} = (g_{out}^T \cdot A) x^T$$

Training LoRA/CIAT

- No need to store original parameter states
- Only need to store:
  - original parameters
  - adapter weights: 2 x d x r
  - adapter gradients: 2 x d x r
  - adapter states: first and second moments, 2 x d x r
  - activations



Counter-Interference Adapter (CIAT) for Multilingual Machine Translation [Zhu et al, EMNLP 2021]
Low-Rank Adaptation of Large Language Models [Hu et al, ICLR 2022]

Memory Consumption of LoRA Training

- LoRA reduce the number of trainable parameters thus reducing the GPU memory requirement

- LLaMA 8B:
  8B parameters -> 16GB (BF16)
  optimizer states -> 48GB (Adam)

- With LoRA/CIAT:
  4M parameters -> 8MB
  optimizer states -> 24MB (Adam)

Benefits of LoRA for Large Models

- Reduced Memory Footprint
  - no need to store state for frozen params
  - LLaMA 8B: 8B → 4M
- Faster Training (fine-tuning)
  - training time
- → Enable fine-tuning of large models, and on smaller devices.
- Inference is as normal LLM.

Experiments of LoRA

- Tasks
  - Natural Language Understanding (NLU): RoBERTa, DeBERTa
    - Subtasks: MNLI, SST-2, MRPC, CoLA, QNLI, QQP, RTE, STS-B
  - Natural Language Generation (NLG): GPT-2, GPT-3
    - Metrics: BLEU, NIST, MET, ROUGE-L, CIDEr
- Six Baselines
  - Fine-Tuning, Bias-only or BitFit, Prefix-embedding tuning (PreEmbed), Prefix-layer tuning (PreLayer), Adapter tuning, LoRA

LoRA is more effective than other fine-tuning methods

LoRA enhances model adaptation with fewer parameters, ensuring both efficiency and improved performance

NLU Tasks

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB_{base} (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB_{base} (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB_{base} ($Adpt^D$)*	0.3M	$87.1_{\pm.0}$	$94.2_{\pm.1}$	$88.5_{\pm1.1}$	$60.8_{\pm.4}$	$93.1_{\pm.1}$	$90.2_{\pm.0}$	$71.5_{\pm2.7}$	$89.7_{\pm.3}$	84.4
RoB_{base} ($Adpt^D$)*	0.9M	$87.3_{\pm.1}$	$94.7_{\pm.3}$	$88.4_{\pm.1}$	$62.6_{\pm.9}$	$93.0_{\pm.2}$	$90.6_{\pm.0}$	$75.9_{\pm2.2}$	$90.3_{\pm.1}$	85.4
RoB_{base} (LoRA)	0.3M	$87.5_{\pm.3}$	**$95.1_{\pm.2}$**	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	**$93.3_{\pm.3}$**	$90.8_{\pm.1}$	**$86.6_{\pm.7}$**	**$91.5_{\pm.2}$**	**87.2**
RoB_{large} (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
RoB_{large} (LoRA)	0.8M	**$90.6_{\pm.2}$**	$96.2_{\pm.5}$	**$90.9_{\pm1.2}$**	**$68.2_{\pm1.9}$**	**$94.9_{\pm.3}$**	$91.6_{\pm.1}$	**$87.4_{\pm2.5}$**	**$92.6_{\pm.2}$**	**89.0**
RoB_{large} ($Adpt^P$)†	3.0M	$90.2_{\pm.3}$	$96.1_{\pm.3}$	$90.2_{\pm.7}$	**$68.3_{\pm1.0}$**	**$94.8_{\pm.2}$**	**$91.9_{\pm.1}$**	$83.8_{\pm2.9}$	$92.1_{\pm.7}$	88.4
RoB_{large} ($Adpt^P$)†	0.8M	**$90.5_{\pm.3}$**	**$96.6_{\pm.2}$**	$89.7_{\pm1.2}$	$67.8_{\pm2.5}$	**$94.8_{\pm.3}$**	$91.7_{\pm.2}$	$80.1_{\pm2.9}$	$91.9_{\pm.4}$	87.9
RoB_{large} ($Adpt^H$)†	6.0M	$89.9_{\pm.5}$	$96.2_{\pm.3}$	$88.7_{\pm2.9}$	$66.5_{\pm4.4}$	$94.7_{\pm.2}$	$92.1_{\pm.1}$	$83.4_{\pm1.1}$	$91.0_{\pm1.7}$	87.8
RoB_{large} ($Adpt^H$)†	0.8M	$90.3_{\pm.3}$	$96.3_{\pm.5}$	$87.7_{\pm1.7}$	$66.3_{\pm2.0}$	$94.7_{\pm.2}$	$91.5_{\pm.1}$	$72.9_{\pm2.9}$	$91.5_{\pm.5}$	86.4
RoB_{large} (LoRA)†	0.8M	**$90.6_{\pm.2}$**	$96.2_{\pm.5}$	**$90.2_{\pm1.0}$**	$68.2_{\pm1.9}$	**$94.8_{\pm.3}$**	$91.6_{\pm.2}$	**$85.2_{\pm1.1}$**	**$92.3_{\pm.5}$**	**88.6**
DeB_{XXL} (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeB_{XXL} (LoRA)	4.7M	**$91.9_{\pm.2}$**	$96.9_{\pm.2}$	**$92.6_{\pm.6}$**	**$72.4_{\pm1.1}$**	**$96.0_{\pm.1}$**	**$92.9_{\pm.1}$**	**$94.9_{\pm.4}$**	**$93.0_{\pm.2}$**	**91.3**

Table 2: $RoBERTa_{base}$, $RoBERTa_{large}$, and $DeBERTa_{XXL}$ with different adaptation methods on the GLUE benchmark. We report the overall (matched and mismatched) accuracy for MNLI, Matthew's correlation for CoLA, Pearson correlation for STS-B, and accuracy for other tasks. Higher is better for all metrics. * indicates numbers published in prior works. † indicates runs configured in a setup similar to Houlsby et al. (2019) for a fair comparison.

NLG Tasks

Model & Method	# Trainable Parameters	E2E NLG Challenge BLEU	NIST	MET	ROUGE-L	CIDEr
GPT-2 M (FT)*	354.92M	68.2	8.62	46.2	71.0	2.47
GPT-2 M ($Adapter^L$)*	0.37M	66.3	8.41	45.0	69.8	2.40
GPT-2 M ($Adapter^L$)*	11.09M	68.9	8.71	46.1	71.3	2.47
GPT-2 M ($Adapter^H$)	11.09M	$67.3_{\pm.6}$	$8.50_{\pm.07}$	$46.0_{\pm.2}$	$70.7_{\pm.2}$	$2.44_{\pm.01}$
GPT-2 M (FT^{Top2})*	25.19M	68.1	8.59	46.0	70.8	2.41
GPT-2 M (PreLayer)*	0.35M	69.7	8.81	46.1	71.4	2.49
GPT-2 M (LoRA)	0.35M	**$70.4_{\pm.1}$**	**$8.85_{\pm.02}$**	**$46.8_{\pm.2}$**	**$71.8_{\pm.1}$**	**$2.53_{\pm.02}$**
GPT-2 L (FT)*	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L ($Adapter^L$)	0.88M	$69.1_{\pm.1}$	$8.68_{\pm.03}$	$46.3_{\pm.0}$	$71.4_{\pm.2}$	**$2.49_{\pm.0}$**
GPT-2 L ($Adapter^L$)	23.00M	$68.9_{\pm.3}$	$8.70_{\pm.04}$	$46.1_{\pm.1}$	$71.3_{\pm.2}$	$2.45_{\pm.02}$
GPT-2 L (PreLayer)*	0.77M	70.3	8.85	46.2	71.7	2.47
GPT-2 L (LoRA)	0.77M	**$70.4_{\pm.1}$**	**$8.89_{\pm.02}$**	**$46.8_{\pm.2}$**	**$72.0_{\pm.2}$**	$2.47_{\pm.02}$

Table 3: GPT-2 medium (M) and large (L) with different adaptation methods on the E2E NLG Challenge. For all metrics, higher is better. LoRA outperforms several baselines with comparable or fewer trainable parameters. Confidence intervals are shown for experiments we ran. * indicates numbers published in prior works.

How to choose a proper rank in LoRA?

- Increasing rank does not cover more meaningful subspaces
  ➔ a low-rank adaptation matrix (r=8)

	Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL(±0.3%)	W_q, W_v	73.4	73.3	**73.7**	**73.8**	73.5
	W_q	68.8	69.6	**70.5**	**70.4**	70.0
MultiNLI (±0.1%)	W_q, W_v	91.3	91.4	91.3	**91.7**	91.4

Experiments of LoRA

NLG Stress Test: Scale up to GPT-3 with 175B parameter

Not all methods benefit monotonically from an increase in trainable parameters.

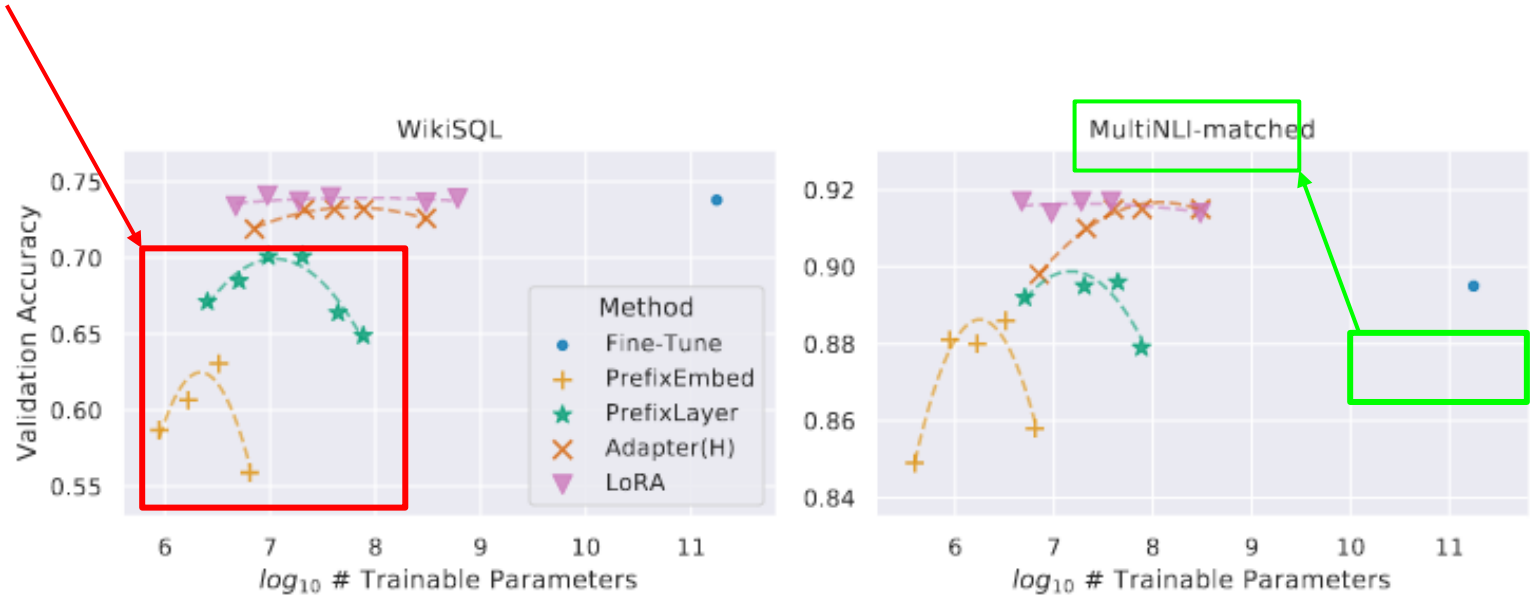
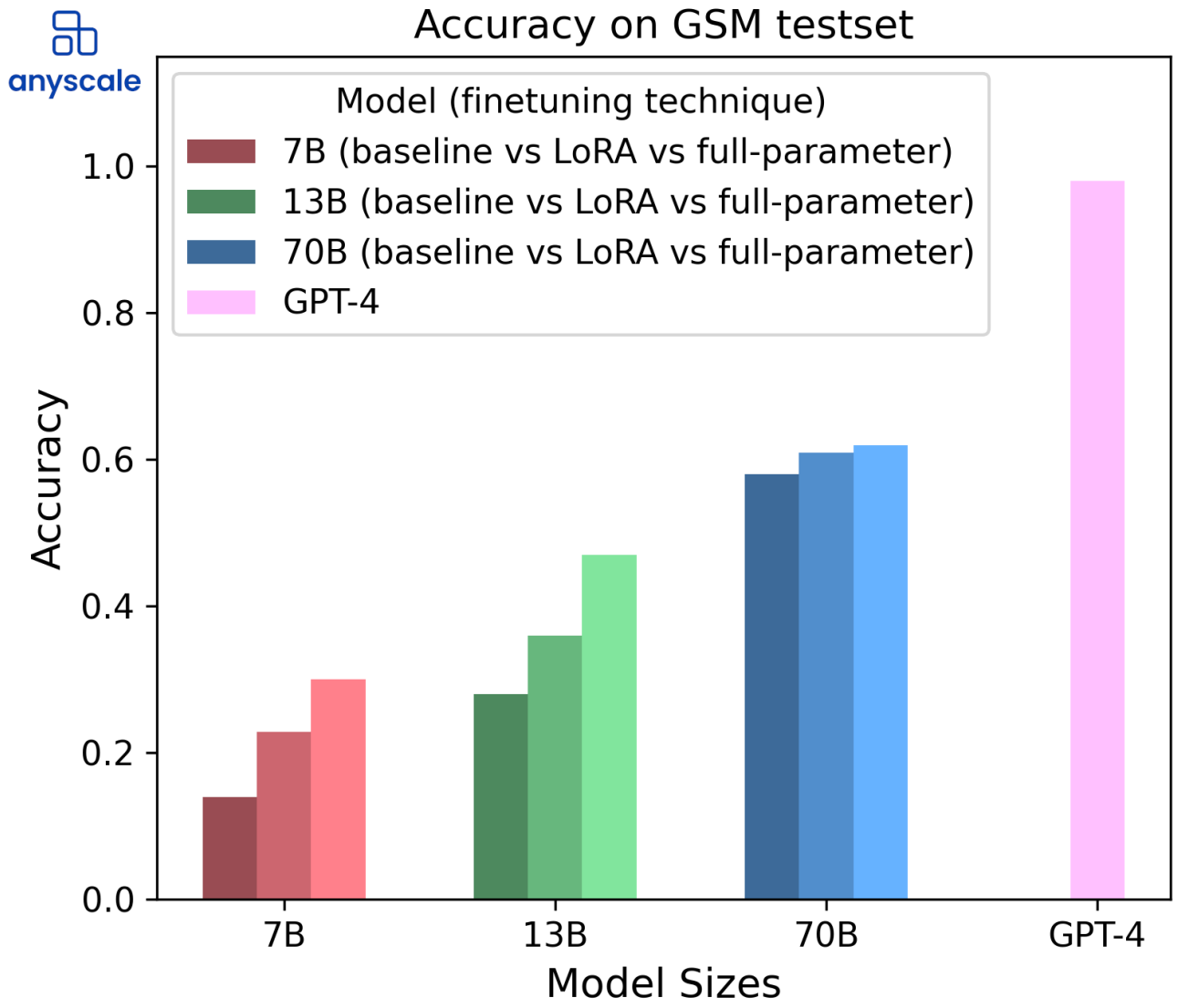


Figure 2: GPT-3 175B validation accuracy vs. number of trainable parameters of several adaptation methods on WikiSQL and MNLI-matched. LoRA exhibits better scalability and task performance. See Section F.2 for more details on the plotted data points.

LoRA Performance on Math (GSM8K)



Outline

- Overview of Parameter Efficient Fine-Tuning
- LoRA: Low-rank Adaptation (Counter-interference adapter, CIAT)
- QLoRA: Quantization + Low-rank training
- Code Walkthrough

Quantization-Aware Training

- GPTQ is Post-Training Quantization (PTQ): converting the weights of an already trained model to a lower precision without any retraining.
- Quantization-Aware Training (QAT): integrates the weight conversion process during the training stage. often superior model performance. (QLoRA)

Overview of QLoRA

QLoRA = Low-rank + Quantized training

- Major innovations:
  - 4-bit Normal Float for storage
  - Double Quantization
  - Page Optimizer
- BF16 for computation

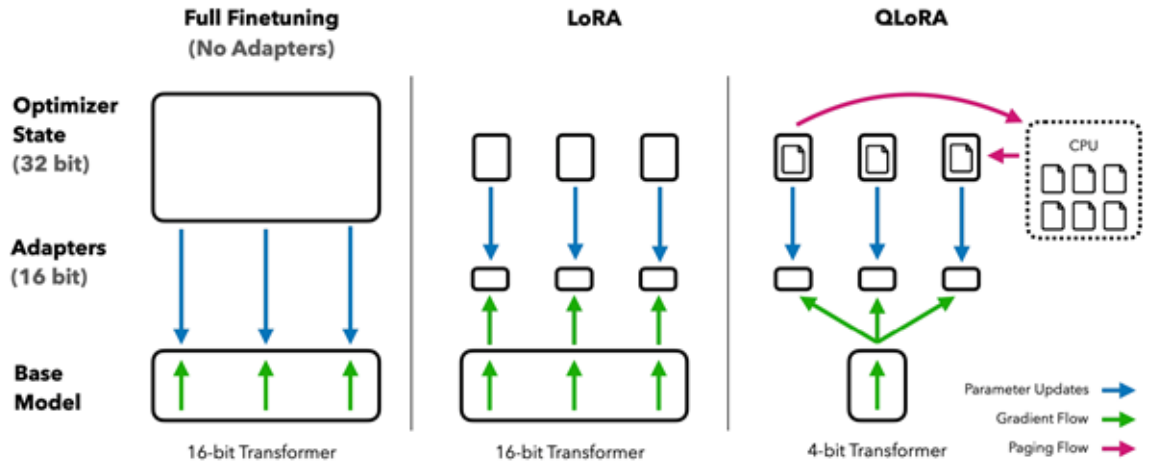


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

➔ Reduces the average memory requirements of fine-tuning a 65B parameter model from 780GB of GPU memory to 48GB on a single GPU.

➔ Preserving performance compared to a 16- bit finetuning

Dettmers et al. QLORA: Efficient Finetuning of Quantized LLMs. NeurIPS 2023.

QLoRA algorithm

QLORA has one storage data type (usually 4-bit NormalFloat) and a computation data type (BF16)

1. Double Quantize the model weights to NF4 format
2. Double De-quantize the weights to BF16
3. Perform forward and backward pass in BF16
4. Compute weight gradients (BF 16) only for the LoRA parameters

Dettmers et al. QLORA: Efficient Finetuning of Quantized LLMs. NeurIPS 2023.

4-bit Number

- A 4-bit number can represent 16 values

- Standard quantization (absmax) divides data into equal-sized bins
  - high quantization error for non-uniform distributed data
  - lots of data with small difference, all get quantized to the same value

Normal Float 4 bit quantization (NF-4)

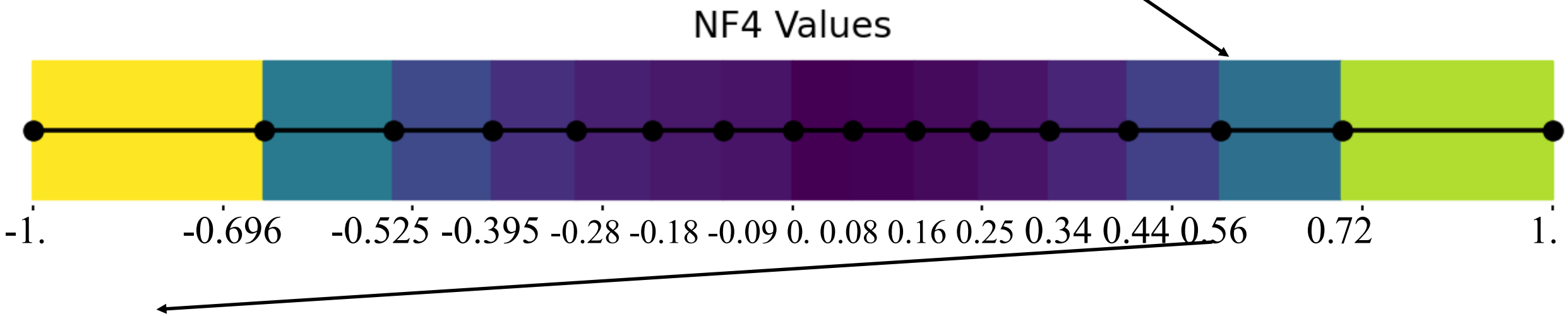
- split weights into blocks of 64 numbers
- Find max abs value ➔ scale
- Normalize the 64 numbers w/ scale ($\hat{x} = \frac{x}{scale}$)
- Find the nearest value from a lookup table

  [-1. -0.6962 -0.5251 -0.3949 -0.2844 -0.1848 -0.0911 0. 0.0796 0.1609 0.2461 0.3379 0.4407 0.5626 0.723 1.]

- Output the quantized values (index from lookup table) [-7, 8]

Illustration: NF-4

value = 0.0045
scale = max(abs(x)) = 0.0071
normalized value = 0.0045 / 0.0071 = 0.63

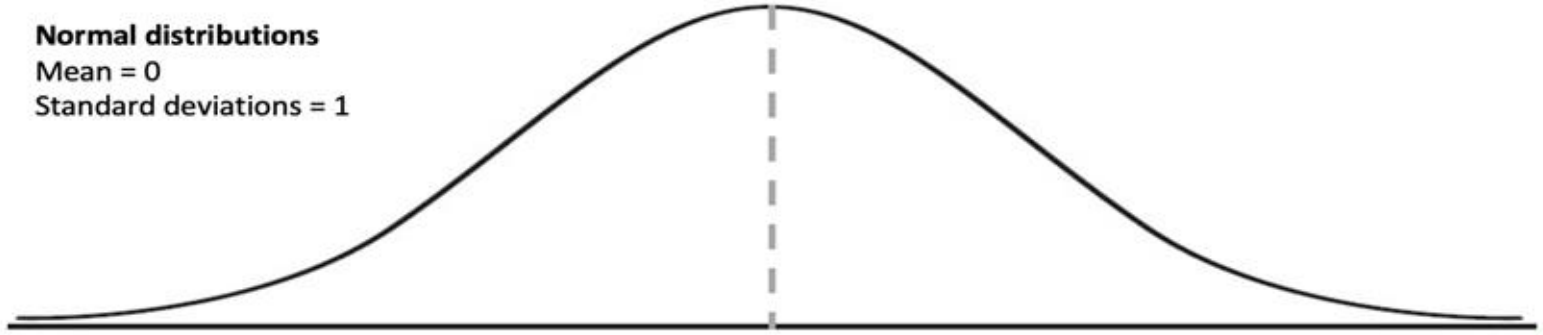


mapped value = 0.56

quantized value = 6

Constructing NF-4 Lookup Table

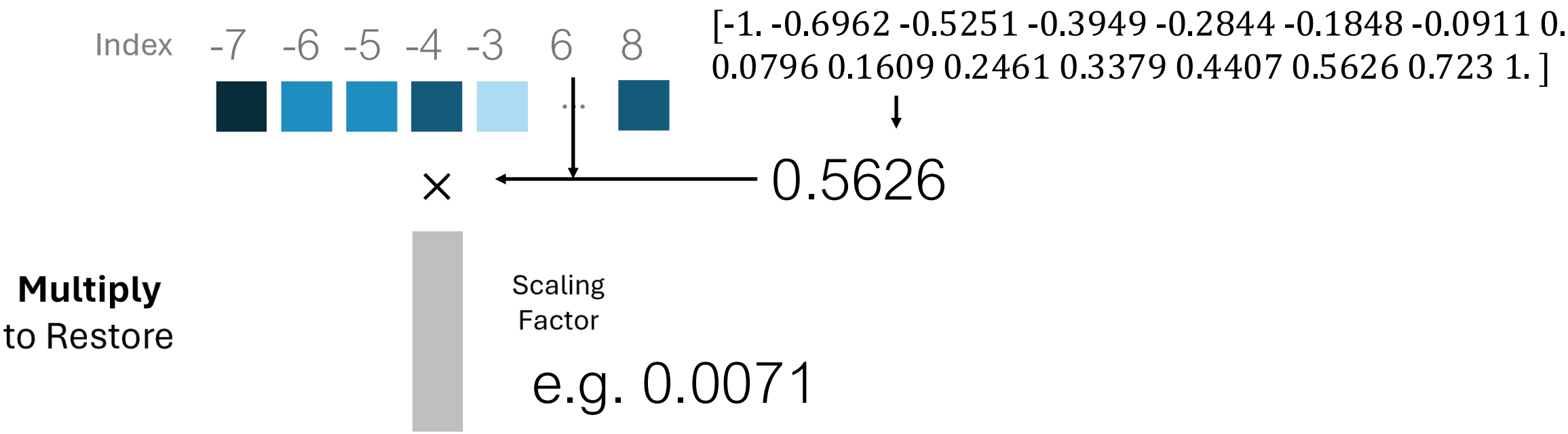
Exact values of the NF4 data type:



Steps for generating the NF4 data type values:

1. Generate 8 evenly spaced values from 0.56 to 0.97 (Set I).
2. Generate 7 evenly spaced values from 0.57 to 0.97 (Set II).
3. Calculate the z-score values for the probabilities generated in Step 1 and Step 2. For Set II, calculate the negative inverse of the z-scores.
4. Concatenate Set I, a zero value, and Set II together.
5. Normalize the values by dividing them by the absolute maximum value.

De-quantization of NF-4



(with slight rounding error)

de-quantized value =0.00399

rounding error = 0.0045-0.00399 = 0.0005

Double Quantization for Scale Factors

Motivation: While a small blocksize is required for precise 4-bit quantization, it also has a considerable overhead.

- E.g. using 32-bit constants and a blocksize of 64, quantization constants add 32/64 = 0.5 bits per parameter on average.

Double Quantization (DQ) quantized the quantization constants for additional memory savings.

Dettmers et al. QLORA: Efficient Finetuning of Quantized LLMs. NeurIPS 2023.

QLoRA Double Quantization

- For every block of 64 values, apply NF-4 quantization
- For every block of 256 scales (from NF-4), apply FP8 quantization, store 2nd scale in FP32

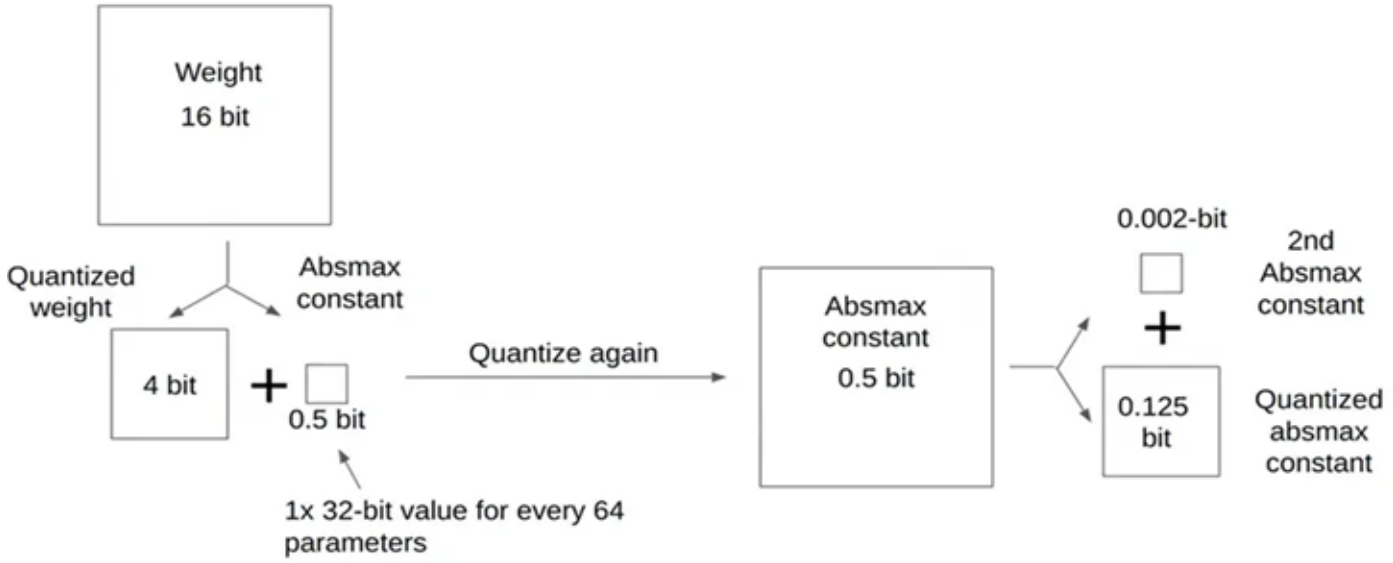


Image source: *Democratizing Foundation Models via k-bit Quantization by Tim Dettmers*

memory needed: 0.5 + 1/64 + 1/(64*256) * 4 = 0.52 bytes per param

QLoRA

- Store pre-trained weights in NF4 with double quantization
  - C1 in FP32 (block size 256), C2 in FP8 (block size 64)
- Apply LoRA, store LoRA weights in BF16
- Store input in BF16, computing (forward/backward/opt) in BF16

$$\mathbf{Y}^{\text{BF16}} = \mathbf{X}^{\text{BF16}}\text{doubleDequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{NF4}}) + \mathbf{X}^{\text{BF16}}\mathbf{L}_1^{\text{BF16}}\mathbf{L}_2^{\text{BF16}},$$

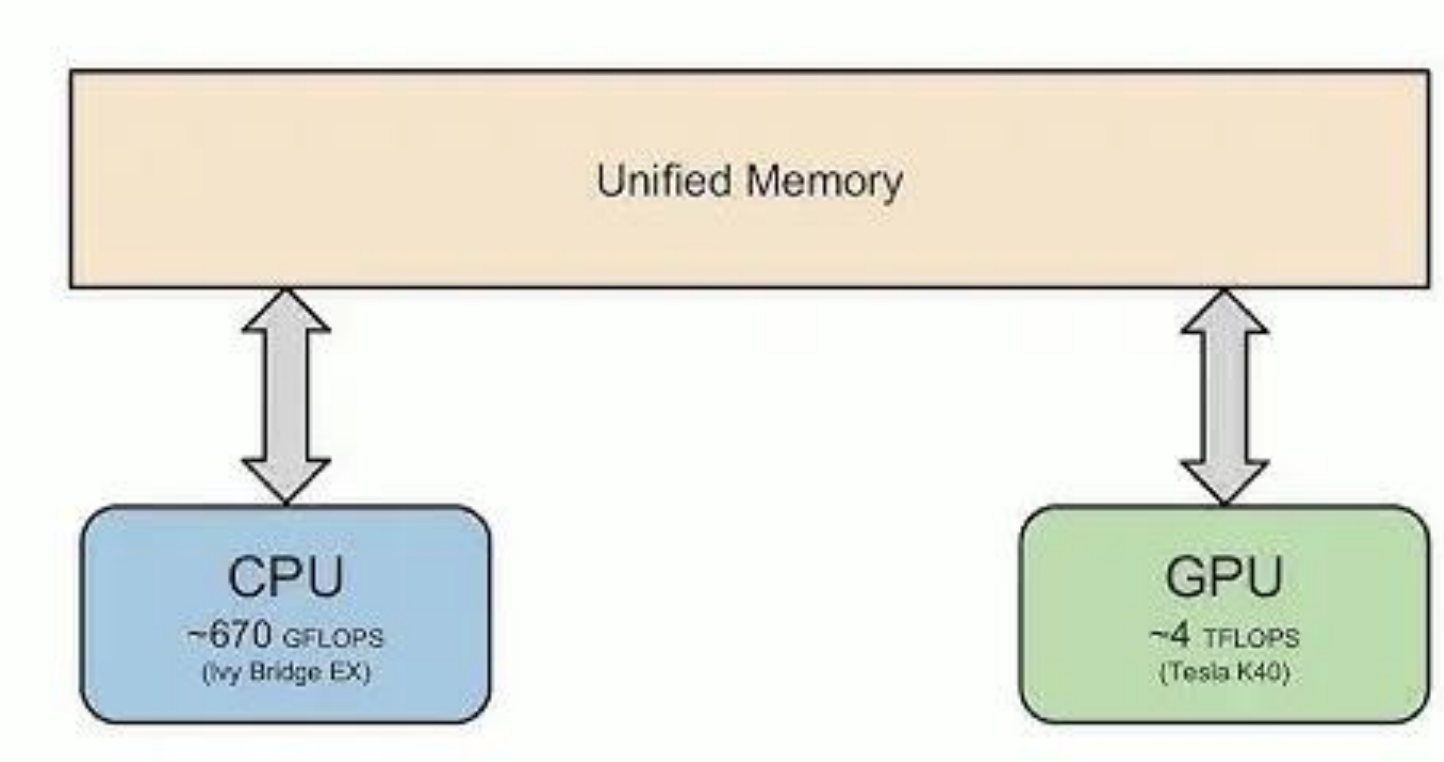
where $\text{doubleDequant}(\cdot)$ is defined as:

$$\text{doubleDequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{k-bit}}) = \text{dequant}(\text{dequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}), \mathbf{W}^{\text{4bit}}) = \mathbf{W}^{\text{BF16}}$$

Dettmers et al. QLORA: Efficient Finetuning of Quantized LLMs. NeurIPS 2023.

Paged Optimizers

Motivation: When training LLMs, GPU's OOM error is a common problem.



Paged optimizers are used to manage memory usage during training.

Dettmers et al. QLORA: Efficient Finetuning of Quantized LLMs. NeurIPS 2023.

Paged Optimizers

- Using the NVIDIA unified memory which does page-to-page transfers between the CPU and GPU for error-free GPU processing when the GPU occasionally runs out-of-memory.
  - like regular memory paging between CPU RAM and the disk.
  - Feature allocates paged memory for the optimizer states which are then automatically evicted to CPU during GPU OOM and back into GPU memory when memory is needed in the optimizer update step

Paged Optimizers

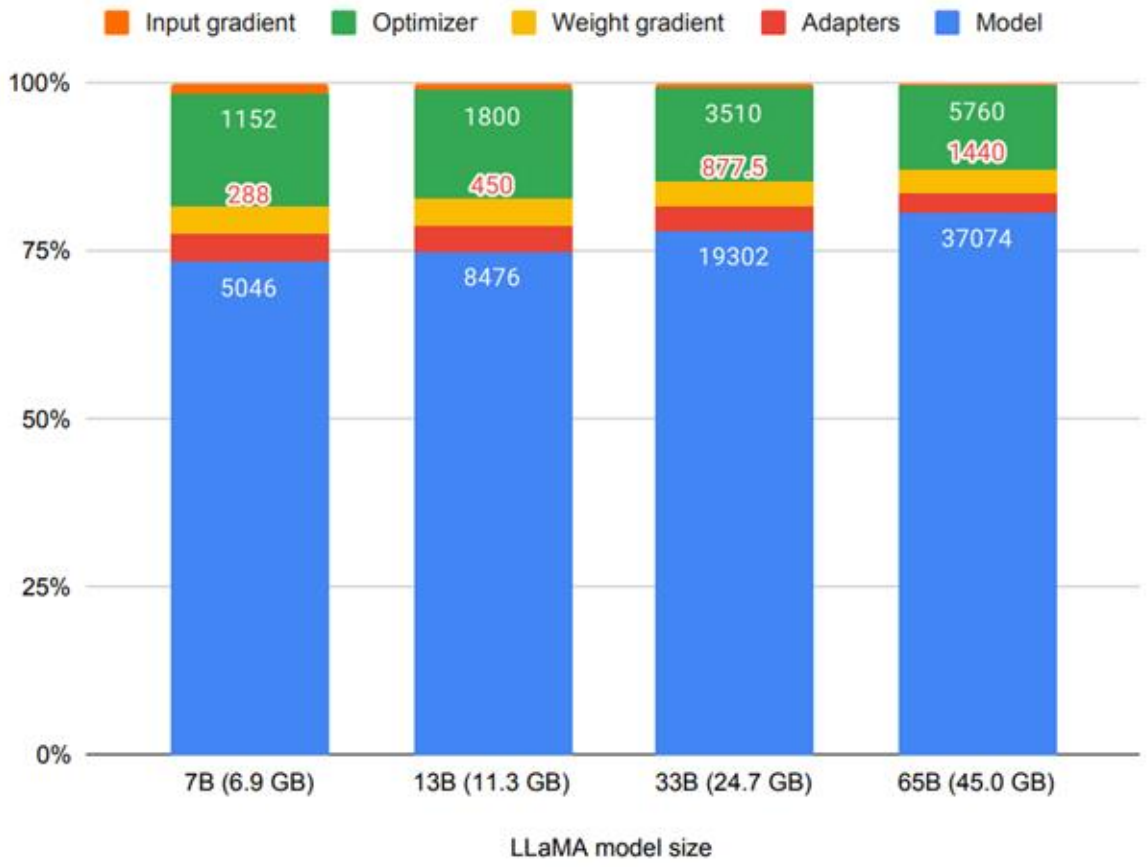
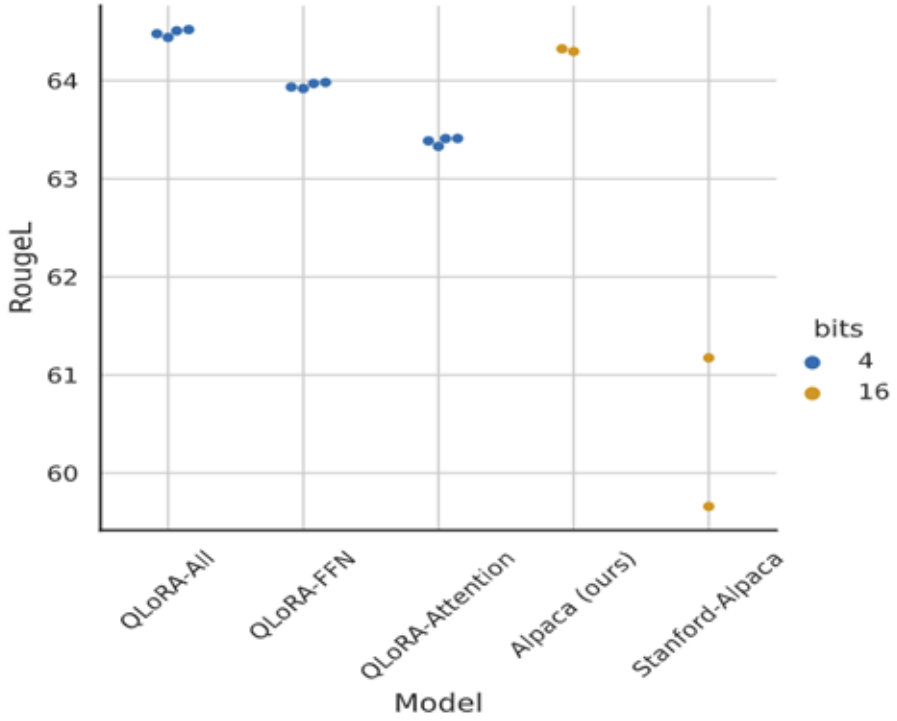


Figure 6: Breakdown of the memory footprint of different LLaMA models. The input gradient size is for batch size 1 and sequence length 512 and is estimated only for adapters and the base model weights (no attention). Numbers on the bars are memory footprint in MB of individual elements of the total footprint. While some models do not quite fit on certain GPUs, paged optimzier provide enough memory to make these models fit.

Dettmers et al. QLORA: Efficient Finetuning of Quantized LLMs. NeurIPS 2023.

Performance of QLoRA

- Default LoRA Hyperparameters do not match 16-bit performance
- NF4 yield better performance than 4-bit Float (FP4)



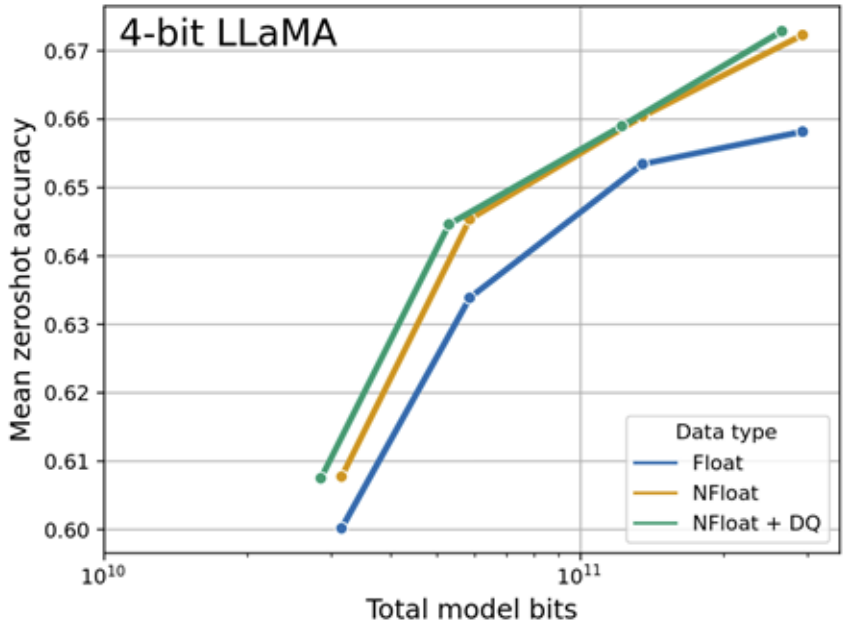


Table 2: Pile Common Crawl mean perplexity for different data types for 125M to 13B OPT, BLOOM, LLaMA, and Pythia models.

Data type	Mean PPL
Int4	34.34
Float4 (E2M1)	31.07
Float4 (E3M0)	29.48
NFloat4 + DQ	**27.41**

Dettmers et al. QLORA: Efficient Finetuning of Quantized LLMs. NeurIPS 2023.

Experiments of QLoRA

- 4-bit QLoRA matches 16-bit full fine-tuning and 16-bit LoRA performance

Table 4: Mean 5-shot MMLU test accuracy for LLaMA 7-65B models finetuned with adapters on Alpaca and FLAN v2 for different data types. Overall, NF4 with double quantization (DQ) matches BFloat16 performance, while FP4 is consistently one percentage point behind both.

	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

Dettmers et al. QLORA: Efficient Finetuning of Quantized LLMs. NeurIPS 2023.

Outline

- Overview of Parameter Efficient Fine-Tuning
- LoRA: Low-rank Adaptation (Counter-interference adapter, CIAT)
- QLoRA: Quantization + Low-rank training
- Code Walkthrough

LoRA Code Walkthrough

```
class LoRALayer():
    def __init__(
        self,
        r: int,
        lora_alpha: int,
        lora_dropout: float,
        merge_weights: bool,
    ):
        self.r = r
        self.lora_alpha = lora_alpha
        # Optional dropout
        if lora_dropout > 0.:
            self.lora_dropout = nn.Dropout(p=lora_dropout)
        else:
            self.lora_dropout = lambda x: x
        # Mark the weight as unmerged
        self.merged = False
        self.merge_weights = merge_weights
```

- Define the LoRA Layer

LoRA Code Walkthrough

```
class Linear(nn.Linear, LoRALayer):
    # LoRA implemented in a dense layer
    def __init__(
        self,
        in_features: int,
        out_features: int,
        r: int = 0,
        lora_alpha: int = 1,
        lora_dropout: float = 0.,
        fan_in_fan_out: bool = False, # Set this to True if the layer to replace stores weight like (fan_in, fan_out)
        merge_weights: bool = True,
        **kwargs
    ):
        nn.Linear.__init__(self, in_features, out_features, **kwargs)
        LoRALayer.__init__(self, r=r, lora_alpha=lora_alpha, lora_dropout=lora_dropout,
                           merge_weights=merge_weights)

        self.fan_in_fan_out = fan_in_fan_out
        # Actual trainable parameters
        if r > 0:
            self.lora_A = nn.Parameter(self.weight.new_zeros((r, in_features)))
            self.lora_B = nn.Parameter(self.weight.new_zeros((out_features, r)))
            self.scaling = self.lora_alpha / self.r
            # Freezing the pre-trained weight matrix
            self.weight.requires_grad = False
        self.reset_parameters()
        if fan_in_fan_out:
            self.weight.data = self.weight.data.transpose(0, 1)
```

- LoRA implement in the linear layer
- Initialize the LoRA A and B layer
- Freeze the pre-trained weight matrix

LoRA Code Walkthrough

```python
def train(self, mode: bool = True):
    def T(w):
        return w.transpose(0, 1) if self.fan_in_fan_out else w
    nn.Linear.train(self, mode)
    if mode:
        if self.merge_weights and self.merged:
            # Make sure that the weights are not merged
            if self.r > 0:
                self.weight.data -= T(self.lora_B @ self.lora_A) * self.scaling
            self.merged = False
    else:
        if self.merge_weights and not self.merged:
            # Merge the weights and mark it
            if self.r > 0:
                self.weight.data += T(self.lora_B @ self.lora_A) * self.scaling
            self.merged = True

def forward(self, x: torch.Tensor):
    def T(w):
        return w.transpose(0, 1) if self.fan_in_fan_out else w
    if self.r > 0 and not self.merged:
        result = F.linear(x, T(self.weight), bias=self.bias)
        result += (self.lora_dropout(x) @ self.lora_A.transpose(0, 1) @ self.lora_B.transpose(0, 1)) * self.scaling
        return result
    else:
        return F.linear(x, T(self.weight), bias=self.bias)
```

- Train module merge the weights of LoRA layer into the pre-train weights
- Given an input x, the forward process compute the sum of the result from two branches:

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

QLoRA code

To train with QLoRA, load the model with the quantization config, then run with LoRA. BitsAndBytes contains custom wrapper for CUDA quantization operations.

```
AutoModelForCausalLM.from_pretrained( "meta-llama/Llama-2-70b",
quantization_config=BitsAndBytesConfig(
        load_in_4bit=args.bits == 4,
        load_in_8bit=args.bits == 8,
        llm_int8_threshold=6.0,
        llm_int8_has_fp16_weight=False,
        bnb_4bit_compute_dtype=compute_dtype,
        bnb_4bit_use_double_quant=args.double_quant,
        bnb_4bit_quant_type=args.quant_type,))
```

Inference Demo:
https://github.com/artidoro/qlora/blob/main/examples/guanaco_7B_demo_colab.ipynb

Summary

- LoRA/CIAT enables cost-effective adaptation of large models by modifying fewer parameters (low-rank).
- Scalability: Effective for giant models like GPT-3, making adaptation more accessible.
- Low-Data Efficacy: Superior in low-data settings, reducing the need for large datasets.

Summary

- QLoRA: double quantizing weights using NF4, computing in BF16
- QLoRA matches original Low-rank Adapter performance.
- 1st method that enables fine-tuning of 33B LLM on a single consumer GPU

Quiz

- on canvas