Efficient fine-tuning for Large Models: LoRA & QLoRA

Jiaqi Song   Xianwei Zou   Hanshi Sun   Steven Kolawole

In class Presentation on March 13th, 2024

Content

- Motivation & Related works (PEFT)
- LoRA Details
- QLoRA Details
  - Quantization & 4-bit Normal Float Quantization
  - Double Quantization
  - Paged Optimizers
- Result and discussion
- LoRA Code Walkthrough

Motivation: Fine-tuning is expensive

- **Full-parameter Fine-Tuning**
  - Update all model parameters → Require large GPU memory
  - e.g. 16-bit Fine-tuning cost per parameter
    - Weight: 16 bits (2 bytes)
    - Weight Gradient: 16 bits (2 bytes)
    - Optimizer States: 65 bits (8 bytes)
    - 96 bits (12 bytes) per parameter
    - 65B model -> 780 GB of GPU memory -> 17x data center GPUs (34x consumer GPUs)

Tim Dettmers | QLoRA: Efficient Finetuning of Quantized Large Language Models

Motivation: Fine-tuning is expensive

- **Full-parameter Fine-Tuning**
  - Update all model parameters → Require large GPU memory
- **Parameter Efficient Fine-tuning (PEFT)**
  - Only update a small subset of parameters, but not degrade the quality of the model
  - 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
    - 17.6 bits per parameters
    - 65B model -> 143 GB of GPU memory -> 4x data center GPUs (8x consumer GPUS)

Tim Dettmers | QLoRA: Efficient Finetuning of Quantized Large Language Models

Motivation: Fine-tuning is expensive

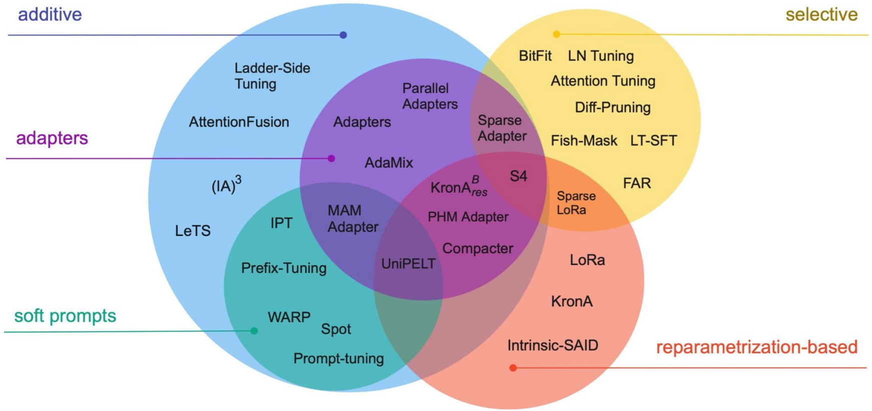
- **Full-parameter Fine-Tuning**
  - Update all model parameters → Require large GPU memory
- **Parameter Efficient Fine-tuning (PEFT)**
  - Only update a small subset of parameters, but not degrade the quality of the model
  - e.g. Fine-tuning cost per parameter with **Q**LoRA
    - Weight: 4 bits
    - Weight Gradient: ~0.4 bit
    - Optimizer State: ~0.8 bit
    - Adapter Weights: ~0.4 bit
    - 5.2 bits per parameters
    - 65B model -> ~~780GB~~ 42 GB of GPU memory -> ~~17x~~ 1x data center GPUs

!!!

Tim Dettmers | QLoRA: Efficient Finetuning of Quantized Large Language Models

Related works: PEFT

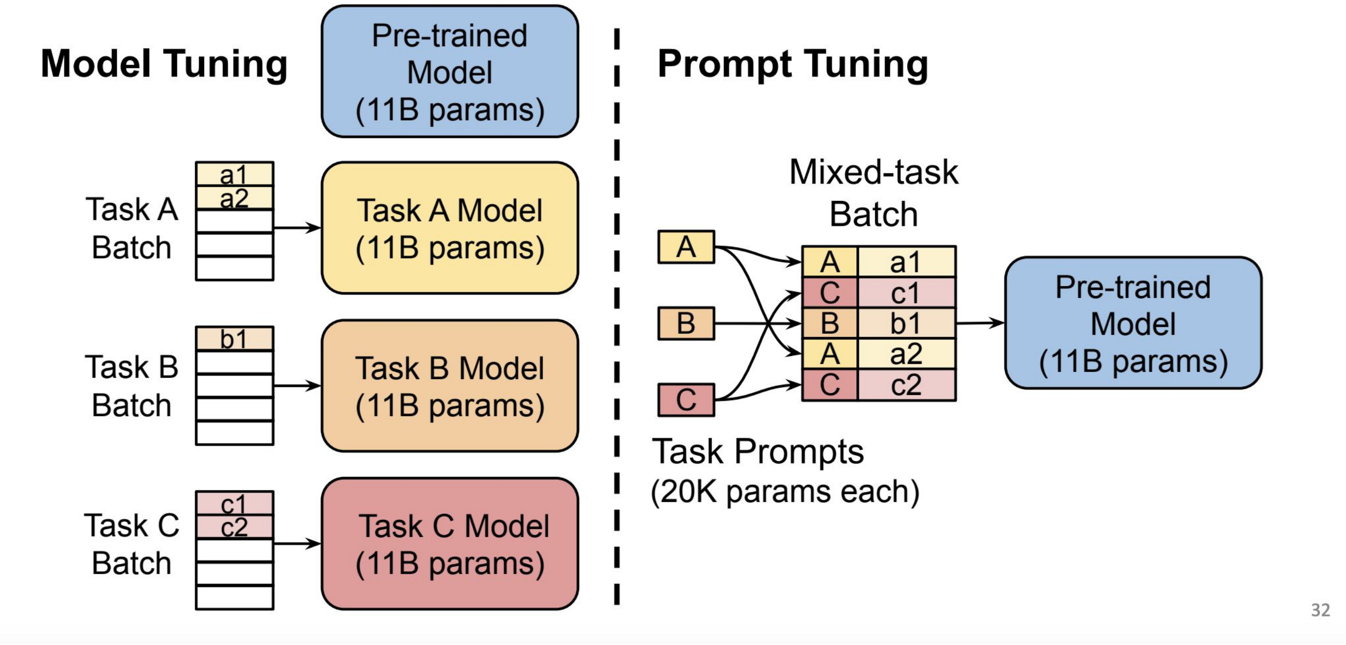
- PEFT Trade-offs
  - Memory Efficiency, Parameter Efficiency, Model Performance, Training Speed, Inference Costs
- PEFT Methods
  - **Selective Methods**:
    - select subsets of parameters to fine-tune
  - **Reparameterization Method:**
    - low-rank representation of model weights
    - e.g. LoRA
  - **Additive Methods**:
    - add trainable layers or parameters to model
    - e.g. Adapters, Soft prompts (Prompt Tuning)



Source: Lialin et al. 2023, "Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning"

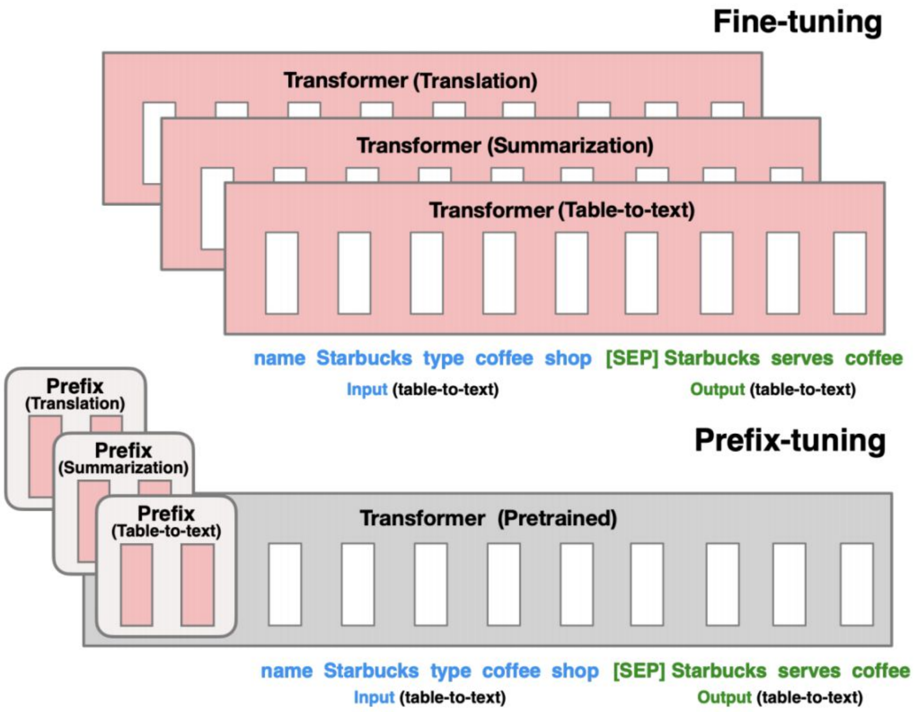
Related works

Prompt tuning (Lester et al. 2021)



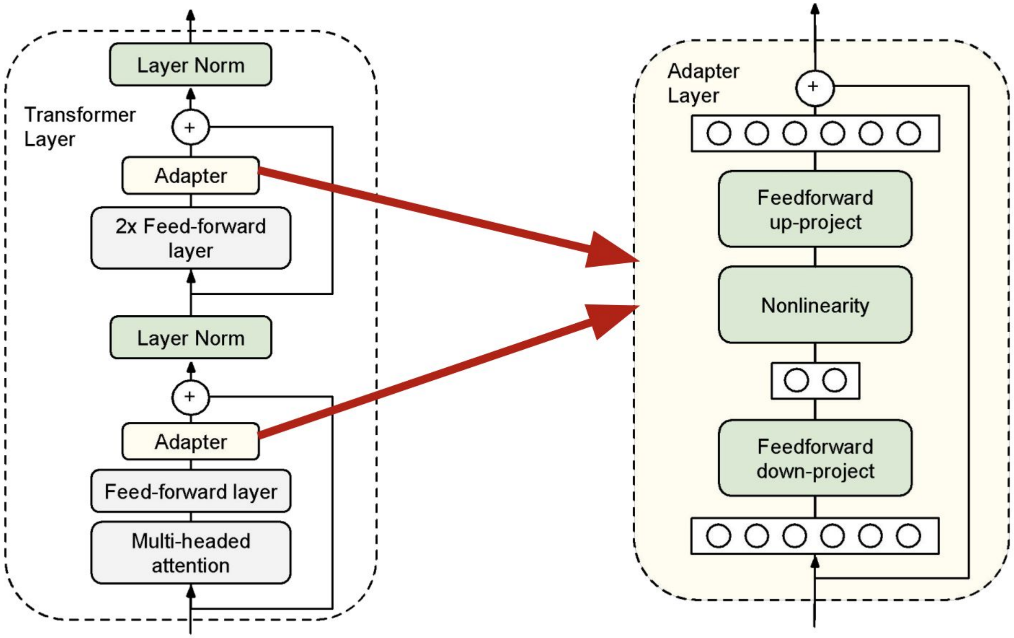
Related works

Prefix tuning (Li and Liang 2021)



Related works

Adapter (Houlsby et al. 2019)



LoRA

It is too expensive to fine-tune all parameters in a large model.

- During fine-tuning we initialized with pre-trained params $\Phi_0$and $\Phi_0 + \Delta\Phi$ updated to by following the objective: $\max_{\Phi} \sum \sum \log(p_{\Phi}(y_t|x, y_{<t}))$
- We can hypothesize that the update matrices in LM adaptation have a low "intrinsic rank", leading to Low-Rank Adaptation (LoRA)
- For each downstream task, we do not need to store/deploy a different set of $\Delta\Phi$ where $|\Phi_0| = |\Delta\Phi|$

Can we find a param-efficient approach by low intrinsic rank?

$$\boxed{\Phi'} = \boxed{\Phi_0} + \boxed{\Delta\Phi}$$

LoRA in Training and Inference

Previous study shows that

- Pre-trained LLMs have a "low intrinsic dimension"
- LLMs can still learn efficiently despite a low-dim reparametrization

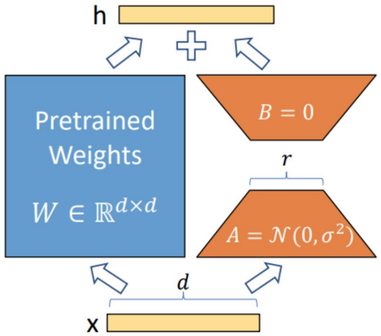


Figure 1: Our reparametrization. We only train A and B.

During training: for pre-trained weight $W_0 \in \mathbb{R}^{d \times k}$, W_0 is fixed

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

$$B \in \mathbb{R}^{d \times r},\ A \in \mathbb{R}^{r \times k},\ r \ll \min(d, k)$$

During inference:

$$W = W_0 + BA$$

Backward Computation of LoRA

What is backward computation?

W_0 is the pre-trained weight matrix that is fixed during training. The gradients of the loss $\mathcal{L}$ with respect to A and B can be derived using the chain rule as follows:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial A} &= \frac{\partial \mathcal{L}}{\partial h} \cdot \frac{\partial h}{\partial A} \\
&= \frac{\partial \mathcal{L}}{\partial h} \cdot \frac{\partial (W_0 x + BAx)}{\partial A} \\
&= \frac{\partial \mathcal{L}}{\partial h} \cdot Bx^T
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial B} &= \frac{\partial \mathcal{L}}{\partial h} \cdot \frac{\partial h}{\partial B} \\
&= \frac{\partial \mathcal{L}}{\partial h} \cdot \frac{\partial (W_0 x + BAx)}{\partial B} \\
&= \frac{\partial \mathcal{L}}{\partial h} \cdot Ax^T
\end{aligned}
$$

These partial derivatives are computed during the backward pass of backpropagation to update the parameters A and B accordingly.

Benefits of LoRA

Applying LoRA to Transformer

- In principle, LoRA can be applied to any weight matrices in DL
- LoRA's original paper only study changing the attention weights

$$W_q, W_k, W_v \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$$

Benefits

- No need to track optimizer state for frozen params, smaller checkpoint size (GPT-3: 1.2TB [before] $\longrightarrow$ 350GB [training] $\longrightarrow$ 35MB [inference])
- No additional inference latency
- speed up during training

Insights of LoRA

How to apply LoRA to Transformer?

- Which weight matrices in Transformer should we apply LoRA to?
- What is the optimal rank for LoRA? - We argue that increasing rank does not cover more meaningful subspaces, which suggests that a low-rank adaptation matrix is sufficient.

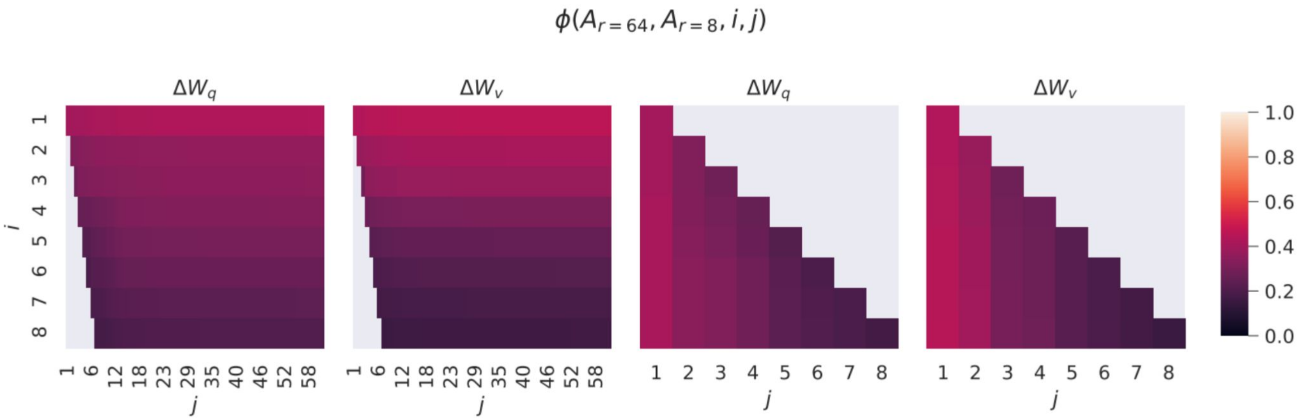
	# of Trainable Parameters = 18M					
Weight Type Rank r	W_q 8	W_k 8	W_v 8	W_o 8	W_q, W_k 4	W_q, W_v 4
WikiSQL ($\pm0.3\%$)	70.4	70.0	73.0	73.2	71.4	**73.7**
MultiNLI ($\pm0.1\%$)	91.0	90.8	91.0	**91.3**	**91.3**	**91.3**

	Weight Type	$r=1$	$r=2$	$r=4$	$r=8$	$r=64$
WikiSQL($\pm0.3\%$)	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 ($\pm0.1\%$)	W_q, W_v	91.3	91.4	91.3	**91.7**	91.4

Insights of LoRA

Subspace similarity between different rank:

- Comparison r=8, r=64 after SVD decomposition, top singular vector space similarity
- The top singular vector overlap significantly between r=8 and r=64, while others do not



Insights of LoRA

Subspace similarity between different rank:

- indicates that the top singular-vector directions of are the most useful, while other directions potentially contain mostly random noises accumulated during training

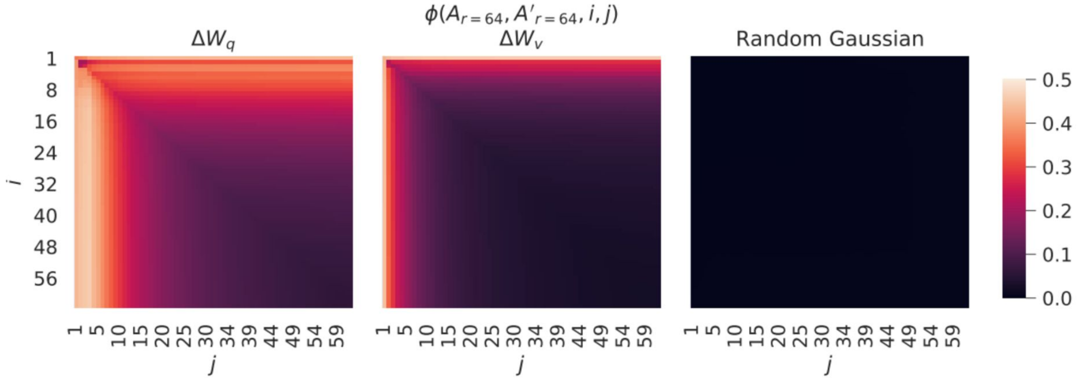


Figure 4: **Left and Middle:** Normalized subspace similarity between the column vectors of $A_{r=64}$ from two random seeds, for both ΔW_q and ΔW_v in the 48-th layer. **Right:** the same heat-map between the column vectors of two random Gaussian matrices. See Sec. G.1 for other layers.

Insights of LoRA

Subspace similarity between different rank:

- ΔW amplifies directions that are important but not emphasized in W
- ΔW with larger r tends to pick directions already emphasized in W

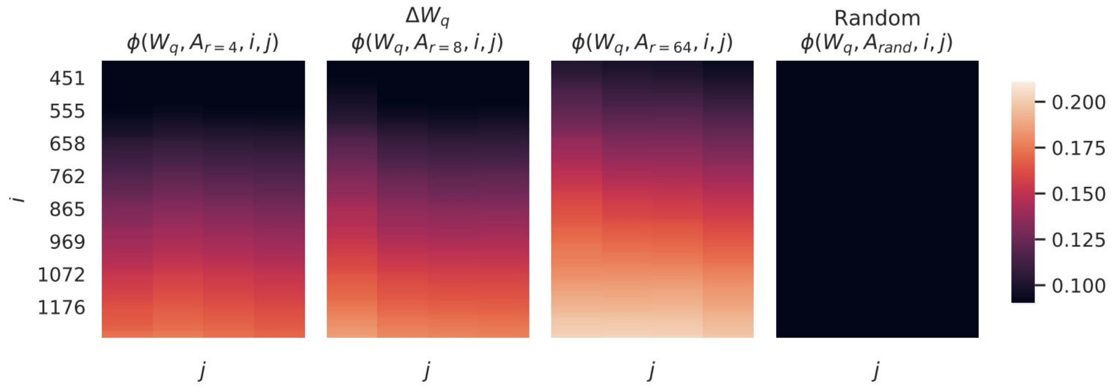


Figure 8: Normalized subspace similarity between the singular directions of W_q and those of ΔW_q with varying r and a random baseline. ΔW_q amplifies directions that are important but not emphasized in W. ΔW with a larger r tends to pick up more directions that are already emphasized in W.

Experiments of LoRA

- Settings
  - Two 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

Experiments of LoRA

Experiments:

- Baseline: Fine-Tune, Bias only, Prefix-embedding tuning, Prefix-layer tuning, adapter tuning
- Faster training, better performance

Method	# of Trainable Parameters	WikiSQL Accuracy (%)	MNLI-m Accuracy (%)	SAMSum R1/R2/RL
GPT-3 175B (Fine-Tune)	175,255.8M	73.0	89.5	52.0/28.0/44.5
GPT-3 175B (Bias Only)	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 175B (PrefixEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 175B (PrefixLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 175B (LoRA)	4.7M	73.4	91.3	52.1/28.3/44.3
GPT-3 175B (LoRA)	37.7M	**73.8**	**91.7**	**53.2/29.2/45.0**

Table 1: Logical form validation accuracy on WikiSQL, validation accuracy on MultiNLI-matched and Rouge-1/2/L on SAMSum achieved by different GPT-3 adaptation methods. LoRA performs better than prior approaches, including conventional fine-tuning. The result on WikiSQL has a fluctuation of $\pm 0.3\%$ and MNLI-m $\pm 0.1\%$.

Method	# of Trainable Parameters	E2E BLEU	NIST	MET	ROUGE-L	CIDEr
GPT-2 M (Fine-Tune)	354.92M	68.2	8.62	46.2	71.0	2.47
GPT-2 M (Adapter)	11.48M	68.9	8.71	46.1	71.3	2.47
GPT-2 M (FT-Top2)	25.19M	68.1	8.59	46.0	70.8	2.41
GPT-2 M (Prefix)	0.35M	69.7	8.81	46.1	71.4	2.49
GPT-2 M (LoRA)	0.35M	**70.4**	**8.85**	**46.8**	**71.8**	**2.53**
GPT-2 L (Fine-Tune)	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L (Prefix)	0.77M	70.3	8.85	46.2	71.7	**2.47**
GPT-2 L (LoRA)	0.77M	**70.4**	**8.89**	**46.8**	**72.0**	**2.47**

Table 2: 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.

Experiments of LoRA

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.

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.

Model&Method	# Trainable Parameters	WikiSQL Acc. (%)	MNLI-m Acc. (%)	SAMSum R1/R2/RL
GPT-3 (FT)	175,255.8M	**73.8**	89.5	52.0/28.0/44.5
GPT-3 (BitFit)	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 (PreEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 (PreLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 (Adapter[H])	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 (Adapter[H])	40.1M	73.2	**91.5**	53.2/29.0/45.1
GPT-3 (LoRA)	4.7M	73.4	**91.7**	**53.8/29.8/45.9**
GPT-3 (LoRA)	37.7M	**74.0**	91.6	53.4/29.2/45.1

Table 4: Performance of different adaptation methods on GPT-3 175B. We report the logical form validation accuracy on WikiSQL, validation accuracy on MultiNLI-matched, and Rouge-1/2/L on SAMSum. LoRA performs better than prior approaches, including full fine-tuning. The results on WikiSQL have a fluctuation around $\pm0.5\%$, MNLI-m around $\pm0.1\%$, and SAMSum around $\pm0.2/\pm0.2/\pm0.1$ for the three metrics.

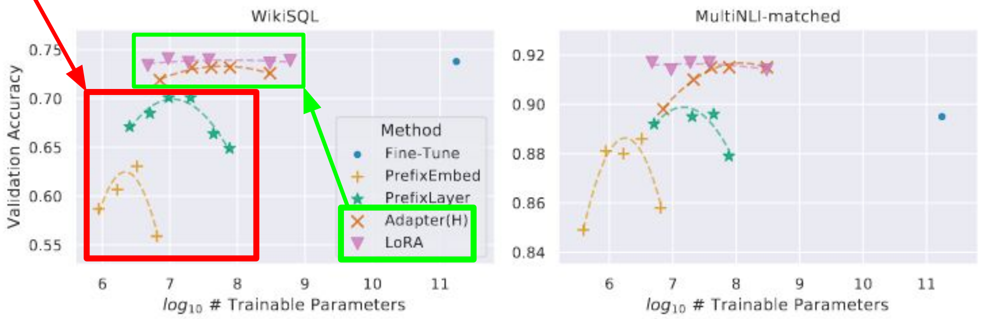


Figure 2: GPT-3 175B validation accuracy vs. number of trainable parameters of several adaptation methods on WikiSQL and MNLI-matched. LoRA exhibits better scalability and task performance. See Section F.2 for more details on the plotted data points.

Experiments of LoRA

How well will LoRA perform in low(limited) training data?

Method	MNLI(m)-100	MNLI(m)-1k	MNLI(m)-10k	MNLI(m)-392K
GPT-3 (Fine-Tune)	60.2	**85.8**	88.9	89.5
GPT-3 (PrefixEmbed)	37.6	75.2	79.5	88.6
GPT-3 (PrefixLayer)	48.3	82.5	85.9	89.6
GPT-3 (LoRA)	**63.8**	85.6	**89.2**	**91.7**

Table 16: Validation accuracy of different methods on subsets of MNLI using GPT-3 175B. MNLI-n describes a subset with n training examples. We evaluate with the full validation set. LoRA performs exhibits favorable sample-efficiency compared to other methods, including fine-tuning.

- **PrefixEmbed** barely performed better than random chance (37.6% vs. 33.3% accuracy).
- **PrefixLayer** did perform better than **PrefixEmbed** but was still significantly worse compared to **fine-tuning** and **LoRA**.
- As the number of training examples increased, the performance gap between **prefix-based** approaches and methods like **fine-tuning** or **LoRA** decreased, suggesting that **prefix-based** approaches might not be well-suited for very low-data scenarios in models like GPT-3.

Conclusion of LoRA

Conclusion:

- **Efficiency:** LoRA enables cost-effective adaptation of large models by modifying fewer parameters.
- **Performance:** Matches or exceeds fine-tuning across various tasks with fewer trainable parameters.
- **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.
- **Zero Latency and Full Capacity:** LoRA maintains model quality without adding inference latency or reducing input sequence length.
- **Broad Applicability:** LoRA's principles are adaptable to various neural network architectures beyond language models.

QLoRA

QLoRA = Quantized pre-train LLM + LoRA

- Major innovations:
  - 4-bit Normal Float
  - Double Quantization
  - Page Optimizer
- 4-bit storage data type
- Bfloat16 computational data type

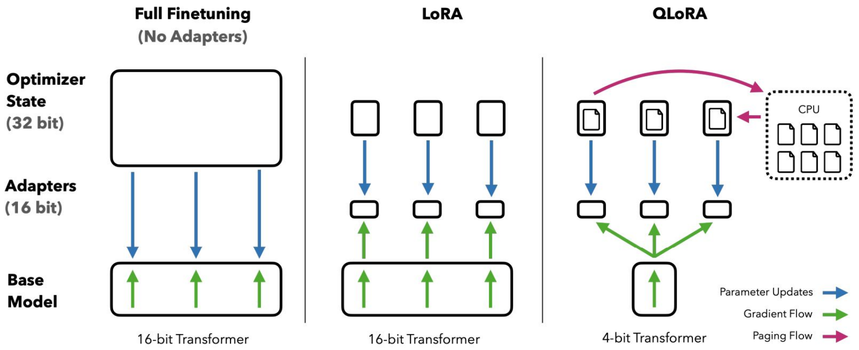


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

➔ 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 fully fine-tuned baseline.

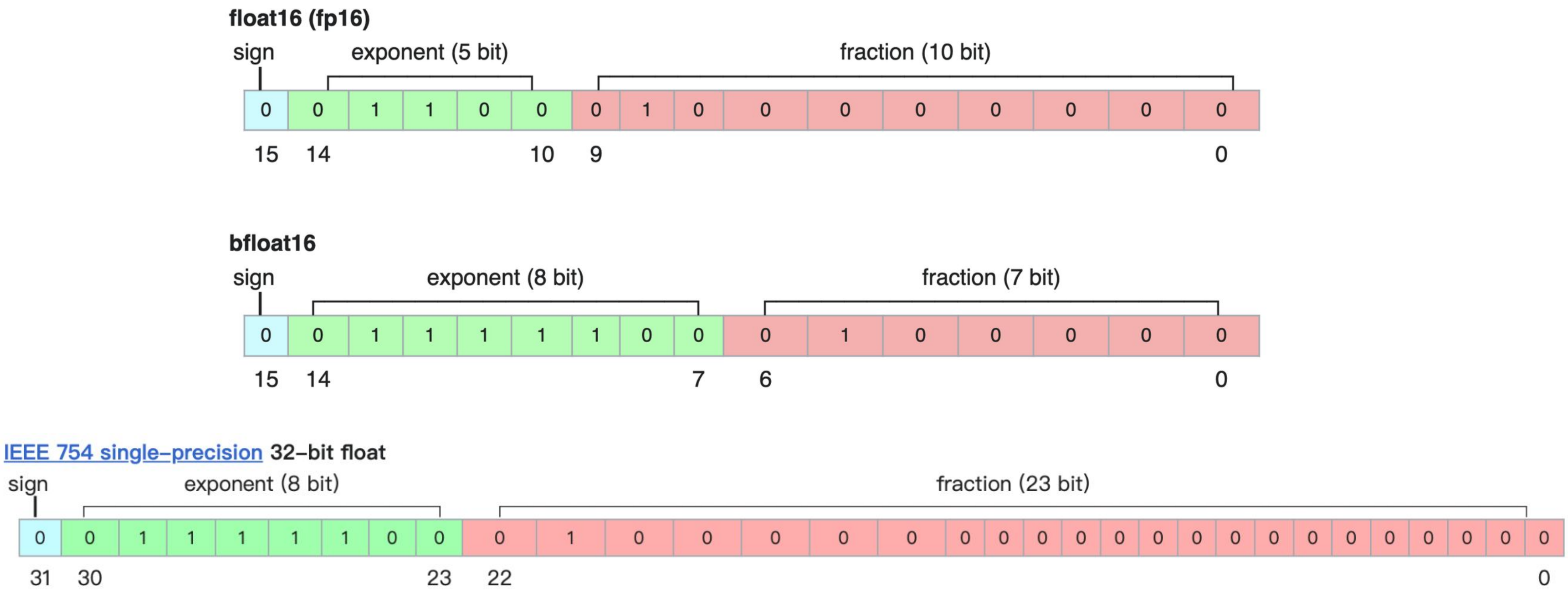
Model Quantization

Keep the model the same but reduce the number of bits.

- **Post-Training Quantization (PTQ)**: converting the weights of an already trained model to a lower precision without any retraining. PTQ might degrade the model's performance.
- **Quantization-Aware Training (QAT)**: integrates the weight conversion process during the training stage. Results in superior model performance. (QLoRA)

Model Quantization

Floating-point numbers:



Model Quantization

Example: use Absolute Maximum (absmax) to quantize a 32-bit Floating Point (FP32) tensor into a Int8 tensor with range [-127, 127]:

$$\mathbf{X}_{\text{quant}} = \text{round}\left(\frac{127}{\max|\mathbf{X}|} \cdot \mathbf{X}\right)$$

$$\mathbf{X}_{\text{dequant}} = \frac{\max|\mathbf{X}|}{127} \cdot \mathbf{X}_{\text{quant}}$$

E.g. Given FP32 [1.2, -3.1, 0.8, 2.4, 5.4]

Scale Factor = 127 / 5.4 = 23.5 (**quantization constant**)

New Int8 [28, -73, 19, 56, 127]

4-bit Normal Float Quantization

- **Motivation:** Weights in pretrain LLM usually has a zero-centered normal distribution
- **Advantage of NF-4:** it is an information theoretically optimal quantization data type for normally distributed data that yields better empirical results than 4-bit Integers and 4-bit Floats

Computation process of NF-4 (k = 4):

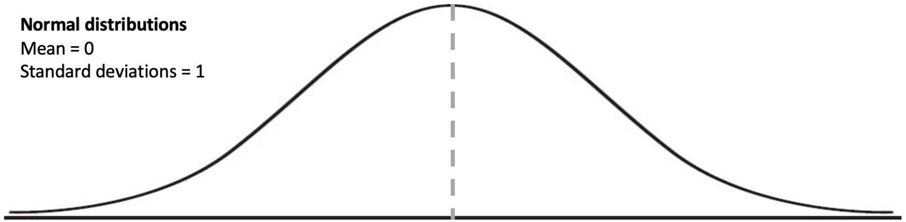
- (1) Estimate the 16 + 1 quantiles of a theoretical N(0, 1) distribution
- (2) Normalized its value into [-1, 1] range
- (3) Quantize the input weight tensor into [-1, 1] range

Estimate the 16 values qi through:

$$q_i = \frac{1}{2}\left(Q_X\left(\frac{i}{2^k+1}\right) + Q_X\left(\frac{i+1}{2^k+1}\right)\right)$$

4-bit Normal Float Quantization

Exact values of the NF4 data type:



[0.97, 0.9 , 0.83, 0.77, 0.7 , 0.63, 0.57] [0.56, 0.62, 0.68, 0.73, 0.79, 0.85, 0.91, 0.97] **(Probability)**

[-1.85, -1.29, -0.97, -0.73, -0.53, -0.34, -0.17] [0.15, 0.3, 0.45, 0.62, 0.81, 1.04, 1.34, 1.85] **(Z-score)**

[-1.85, -1.29, -0.97, -0.73, -0.53, -0.34, -0.17, **0** , 0.15, 0.3, 0.45, 0.62, 0.81, 1.04, 1.34, 1.85] **(Concatenation)**

[-1. , -0.7 , -0.53, -0.39, -0.28, -0.18, -0.09, 0. , 0.08, 0.16, 0.25, 0.34, 0.44, 0.56, 0.72, 1.] **(Normalisation)**

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.

4-bit Normal Float Quantization

Problems with the original quantization method:

Outliers in input tensor will lead to inefficient use of quantization bins.

Solution (Block-wise Quantization):

We can chunk input tensor into n contiguous block of size B. with their own **quantization constant** c.

If need more quantization constant, use double quantization, which can help reduce the memory footprint of quantization constants

Double Quantization

To perform dequantization technique, we need to store the quantization constants.

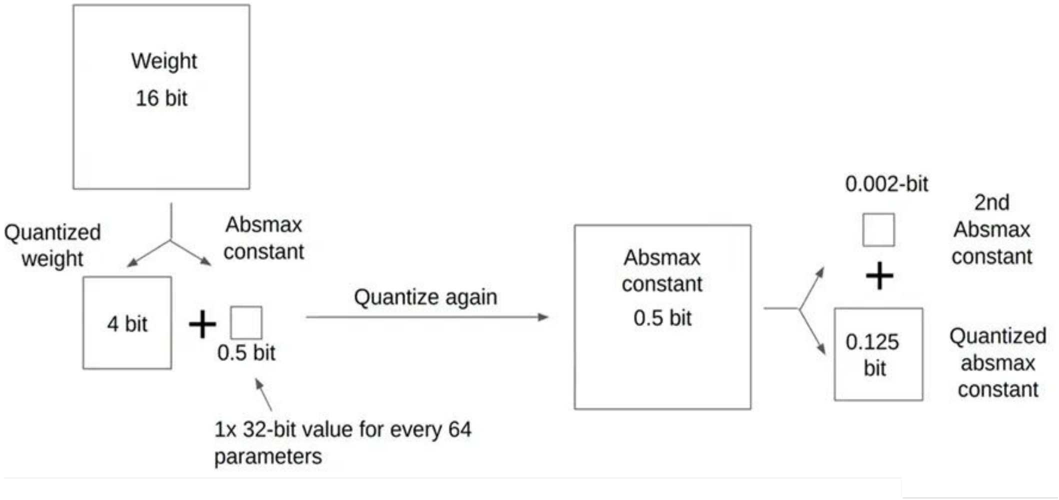


Image source: *Democratizing Foundation Models via k-bit Quantization by Tim Dettmers*

If we employed blockwise quantization, then we will have *n* quantization constants in their original data type.

Double Quantization

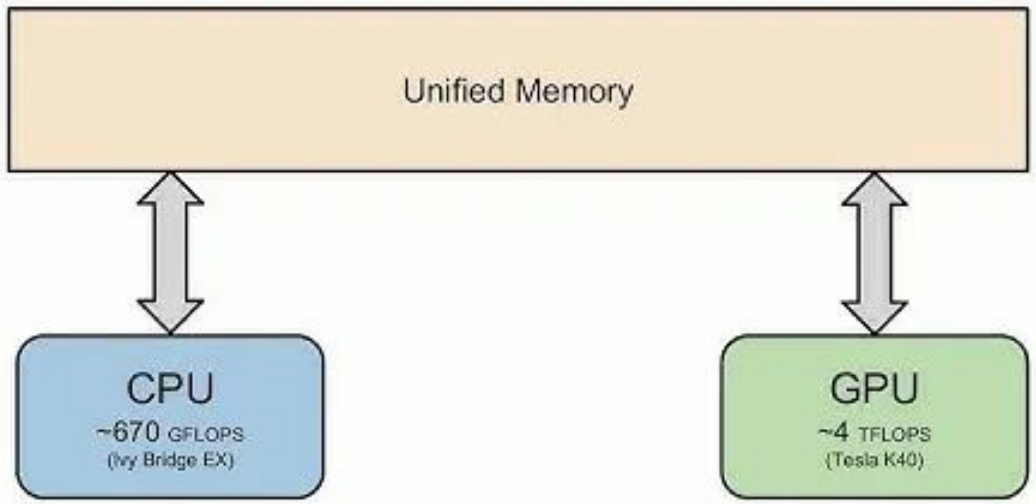
Motivation: While a small blocksize is required for precise 4-bit quantization, it also has a considerable overhead.

- 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.

Paged Optimizers

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



Paged optimizers are used to manage memory usage during training.

Paged Optimizers

Paged Optimizers use the NVIDIA unified memory feature which does page-to-page transfers between the CPU and GPU for error-free GPU processing when the GPU occasionally runs out-of-memory.

- The feature works 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

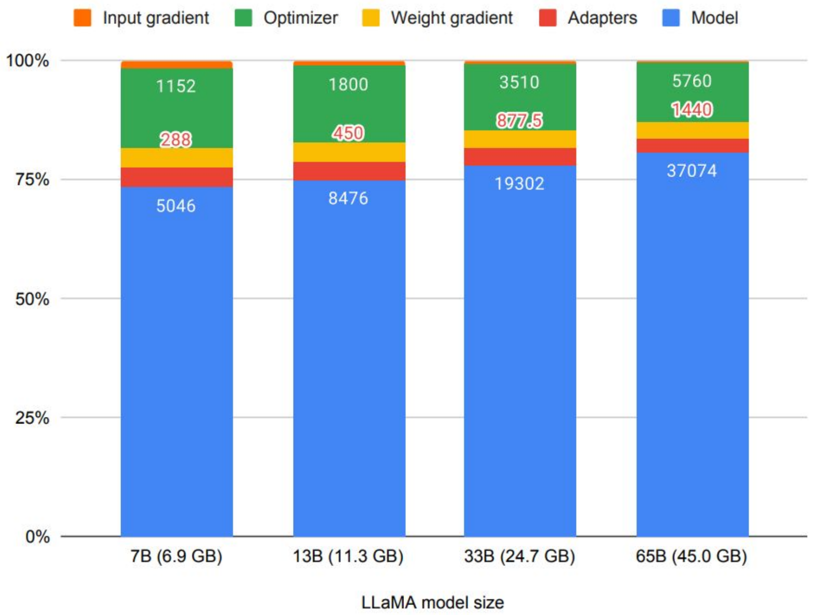


Figure 6: Breakdown of the memory footprint of different LLaMA models. The input gradient size is for batch size 1 and sequence length 512 and is estimated only for adapters and the base model weights (no attention). Numbers on the bars are memory footprint in MB of individual elements of the total footprint.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

QLoRA

Given all above components, **QLoRA** for a single linear layer in the quantized based model with a single LoRA adapter is defined as follows;

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

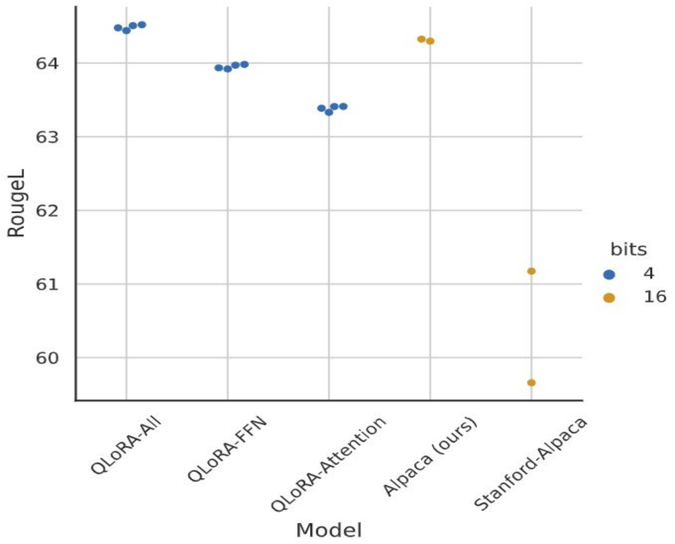
where doubleDequant$(\cdot)$ is defined as:

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

- NF4 -> W;
- FP8 -> c_2;
- blocksize of 64 -> W (for higher quantization precision);
- blocksize of 256 for c_2 (to conserve memory)

Experiments of QLoRA

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



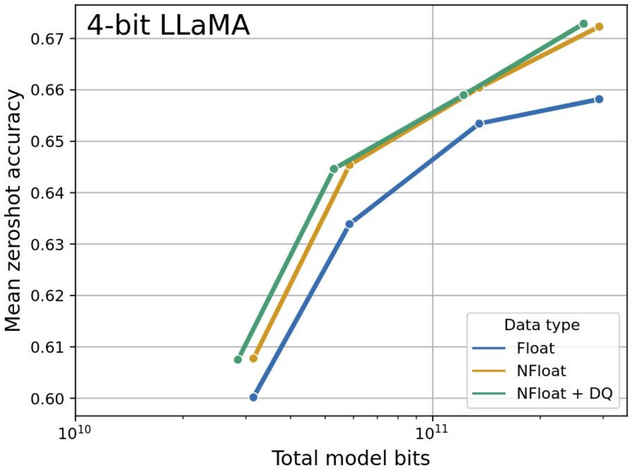


Table 2: Pile Common Crawl mean perplexity for different data types for 125M to 13B OPT, BLOOM, LLaMA, and Pythia models.

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

Experiments of QLoRA

- k-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

Conclusion of QLoRA

Conclusion:

- QLoRA can replicate 16-bit full fine-tuning performance with a 4-bit base model and Low-rank Adapters.
- It's the first method that enables fine-tuning of 33B parameter models **on a single consumer GPU** and 65B parameter models **on a single professional GPU** without degrading performance relative to a full finetuning baseline.
- QLoRA's best 33B model, trained on the Open Assistant dataset, can rival ChatGPT on the Vicuna benchmark, making fine-tuning widespread and accessible, especially for researchers with limited resources.

Conclusion of QLoRA

Limitations:

- Unable to establish that QLoRA matches 16-bit fine-tuning performance at 33B and 65B scales due to immense resource cost.
- Did not evaluate on BigBench, RAFT, and HELM benchmarks, making it unclear if evaluations generalize to these benchmarks.
- The performance likely depends on how similar the fine-tuning data is to the benchmark dataset, highlighting the need for better benchmarks and evaluation metrics that reflect real-world applications.
- Did not evaluate different bit-precisions or other PEFT methods beyond LoRA, which might yield better performance or enable more aggressive quantization.

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:

$$h = W_0x + \Delta W x = W_0 x + BAx$$

Thanks