LoRA: Low-Rank Adaptation of Large Language Models

Edward Hu* **Yelong Shen*** **Phillip Wallis** **Zeyuan Allen-Zhu**
Yuanzhi Li **Shean Wang** **Lu Wang** **Weizhu Chen**
Microsoft Corporation
{edwardhu, yeshe, phwallis, zeyuana,
yuanzhil, swang, luw, wzchen}@microsoft.com
yuanzhil@andrew.cmu.edu
(Version 2)

Abstract

- As we pre-train larger models, full fine-tuning becomes less feasible
- Low-Rank Adaptation (LoRA)
  - greatly reducing the number of trainable parameters for downstream tasks
- LoRA performs on-par or better than fine-tuning in model quality

Introduction

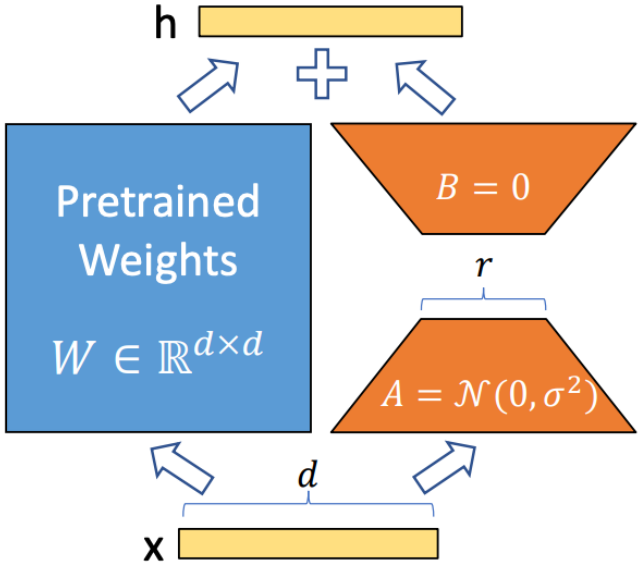
- One large-scale, pre-trained LM -> multiple downstream applications
  - Fine-tuning: updates all the parameters of the pre-trained model
  - GPT-2, RoBERTa large -> mere inconvenience…
  - GPT-3 with 175 billion parameters -> critical deployment challenge!

Introduction

- Instead of fine-tuning,
  - Adapting only some parameters
  - Learning external modules

  -> Greatly boosting the operational efficiency!
- However,
  - Inference latency
  - often fail to match the fine-tuning baselines

Introduction



- Inspiration (prior research)
  - over-parameterized models reside on a low intrinsic dimension
- LoRA
  - hypothesize that the change in weights also has a low "intrinsic dimension"
  - optimizing rank decomposition matrices of the dense layers' change

Problem Statement

- Full fine-tuning

$$\max_{\Phi} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log(P_{\Phi}(y_t|x, y_{<t}))$$

  - Initialized to pre-trained weights Φ_0 and updated to $\Phi_0 + \Delta\Phi$
  - $|\Phi_0| = |\Delta\Phi|$
  - If the pre-trained model is large, storing and deploying many independent instances of fine-tuned models can be challenging

Problem Statement

- More parameter-efficient approach
  - $\Delta\Phi = \Delta\Phi(\Theta)$ is encoded by a much smaller-sized set of parameters Θ
  - The task of finding $\Delta\Phi$ becomes optimizing over Θ

$$\max_{\Theta} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(p_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t})\right)$$

  - Use a low-rank representation to encode $\Delta\Phi$ that is both compute- and memory-efficient

Existing solutions?

- Adapter layers introduce inference latency

Batch Size Sequence Length $\|\Theta\|$	32 512 0.5M	16 256 11M	1 128 11M
Fine-Tune/LoRA	1449.4±0.8	338.0±0.6	19.8±2.7
Adapter^{L}	1482.0±1.0 (+2.2%)	354.8±0.5 (+5.0%)	23.9±2.1 (+20.7%)
Adapter^{H}	1492.2±1.0 (+3.0%)	366.3±0.5 (+8.4%)	25.8±2.2 (+30.3%)

Table 1: Infernece latency of a single forward pass in GPT-2 medium measured in milliseconds, averaged over 100 trials. We use an NVIDIA Quadro RTX8000. "$|\Theta|$" denotes the number of trainable parameters in adapter layers. Adapter^{L} and Adapter^{H} are two variants of adapter tuning, which we describe in Section 5.1. The inference latency introduced by adapter layers can be significant in an online, short-sequence-length scenario. See the full study in Appendix B.

Existing solutions?

- Directly Optimizing the Prompt is Hard
  - Prefix tuning is difficult to optimize
    and performance changes non-monotonically in trainable parameters
  - Reserving a part of the sequence length for adaptation reduces
    the sequence length available to process a downstream task,
    which makes tuning the prompt less performant

Method

- Low-Rank Parameterized Update Matrices
  - pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$ is updated to $W_0 + \Delta W = W_0 + BA$, where $B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}$ and $r \ll \min(d, k)$
    - W_0 is frozen, A and B contain trainable parameters
  - Forward pass:

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

    - a random Gaussian initialization for A, and zero for B, so $\Delta W = BA$ is zero
    - scale $\Delta W x$ by α / r, where α is a constant in r

Method

- Low-Rank Parameterized Update Matrices
  - A Generalization of Full Fine-tuning
    - More general form of fine-tuning: the training of a subset of the pre-trained parameters
    - Setting the LoRA rank r to the rank of the pre-trained weight matrices -> full fine-tuning
  - No Additional Inference Latency
    - compute and store $W = W_0 + BA$ and perform inference as usual
    - recover W_0 by subtracting BA and then adding a different $B'A'$

Method

- Applying LoRA to Transformer
  - Transformer Architecture
    - Four weight matrices in the self-attention module ($W_q, W_k, W_v, W_o \in \mathbb{R}^{d_{model} \times d_{model}}$)
    - Two in the MLP module
  - In this study, **only adapting the attention weights** and freeze MLP modules
    - for simplicity and parameter-efficiency
    - leave future work

Method

- Applying LoRA to Transformer
  - Practical Benefits: Reduction in memory and storage usage
    - reduce that VRAM usage by up to 2/3
    - the checkpoint size is reduced by roughly 10,000×
    - 25% speedup during training
  - This allows us to train with significantly fewer GPUs and avoid I/O bottlenecks
  - We can switch between tasks while deployed at a much lower cost by swapping the LoRA weights

Empirical Experiments

- RoBERTa base/large, DeBERTa XXL

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	$87.5_{\pm.3}$	$\mathbf{95.1}_{\pm.2}$	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm.3}$	$90.8_{\pm.1}$	$\mathbf{86.6}_{\pm.7}$	$\mathbf{91.5}_{\pm.2}$	**87.2**
$\text{RoB}_{\text{large}}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
$\text{RoB}_{\text{large}}$ (LoRA)	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.9}_{\pm1.2}$	$\mathbf{68.2}_{\pm1.9}$	$\mathbf{94.9}_{\pm.3}$	$91.6_{\pm.1}$	$\mathbf{87.4}_{\pm2.5}$	$\mathbf{92.6}_{\pm.2}$	**89.0**
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	3.0M	$90.2_{\pm.3}$	$96.1_{\pm.3}$	$90.2_{\pm.7}$	$\mathbf{68.3}_{\pm1.0}$	$\mathbf{94.8}_{\pm.2}$	$\mathbf{91.9}_{\pm.1}$	$83.8_{\pm2.9}$	$92.1_{\pm.7}$	88.4
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	0.8M	$\mathbf{90.5}_{\pm.3}$	$\mathbf{96.6}_{\pm.2}$	$89.7_{\pm1.2}$	$67.8_{\pm2.5}$	$\mathbf{94.8}_{\pm.3}$	$91.7_{\pm.2}$	$80.1_{\pm2.9}$	$91.9_{\pm.4}$	87.9
$\text{RoB}_{\text{large}}$ (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
$\text{RoB}_{\text{large}}$ (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
$\text{RoB}_{\text{large}}$ (LoRA)†	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.2}_{\pm1.0}$	$68.2_{\pm1.9}$	$\mathbf{94.8}_{\pm.3}$	$91.6_{\pm.2}$	$\mathbf{85.2}_{\pm1.1}$	$\mathbf{92.3}_{\pm.5}$	**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	$\mathbf{91.9}_{\pm.2}$	$96.9_{\pm.2}$	$\mathbf{92.6}_{\pm.6}$	$\mathbf{72.4}_{\pm1.1}$	$\mathbf{96.0}_{\pm.1}$	$\mathbf{92.9}_{\pm.1}$	$\mathbf{94.9}_{\pm.4}$	$\mathbf{93.0}_{\pm.2}$	**91.3**

Empirical Experiments

- GPT-2 medium/large

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	$\mathbf{70.4}_{\pm .1}$	$\mathbf{8.85}_{\pm .02}$	$\mathbf{46.8}_{\pm .2}$	$\mathbf{71.8}_{\pm .1}$	$\mathbf{2.53}_{\pm .02}$
GPT-2 L (FT)*	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L (Adapter^{L})	0.88M	$69.1_{\pm .1}$	$8.68_{\pm .03}$	$46.3_{\pm .0}$	$71.4_{\pm .2}$	$\mathbf{2.49}_{\pm .0}$
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	$\mathbf{70.4}_{\pm .1}$	$\mathbf{8.89}_{\pm .02}$	$\mathbf{46.8}_{\pm .2}$	$\mathbf{72.0}_{\pm .2}$	$2.47_{\pm .02}$

Empirical Experiments

- GPT-3 175B

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

Understanding the Low-Rank Updates

- Which weight matrices in Transformer should we apply LoRA to?

	# 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	W_q, W_k, W_v, W_o 2
WikiSQL ($\pm 0.5\%$)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI ($\pm 0.1\%$)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

  - Adapting both W_q and W_v gives the best performance overall

Understanding the Low-Rank Updates

- What is the optimal rank r for LoRA?

	Weight Type	$r=1$	$r=2$	$r=4$	$r=8$	$r=64$
WikiSQL($\pm 0.5\%$)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI ($\pm 0.1\%$)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4

  - LoRA already performs competitively with a very small r
  - This suggests the update matrix ΔW could have a very small “intrinsic rank”
    - do not expect a small r to work for every task or dataset

Understanding the Low-Rank Updates

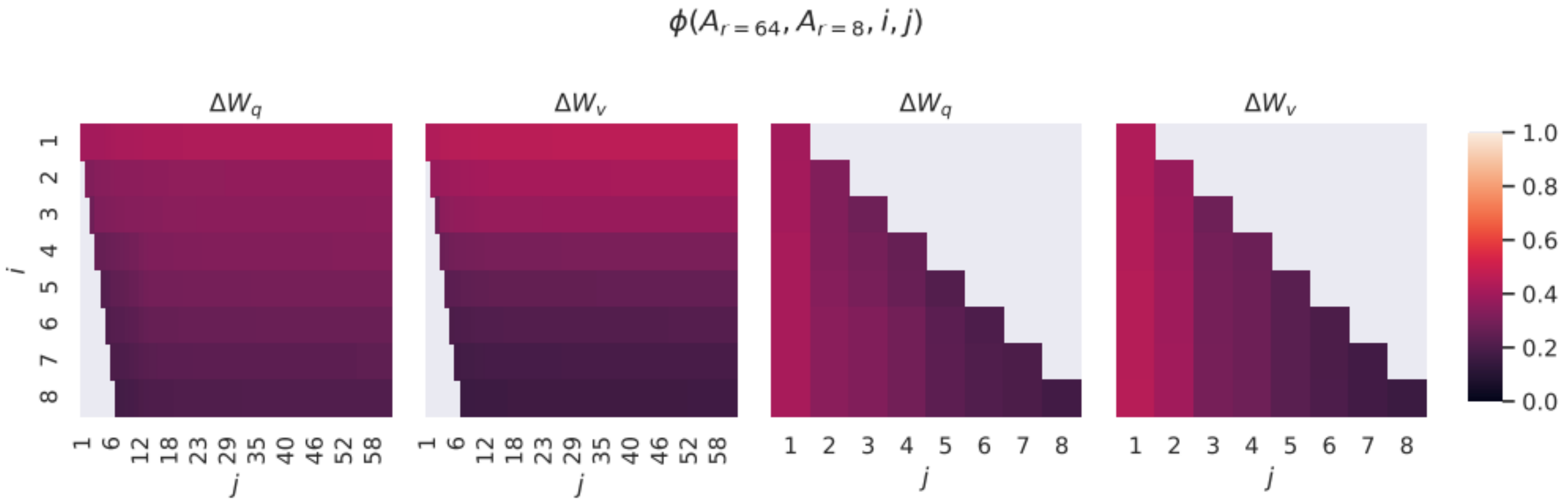
- What is the optimal rank r for LoRA?
  - Subspace similarity between different r
    - $A_{r=8}, A_{r=64}$: learned adaptation matrices with rank $r = 8, 64$
    - $U_{A_{r=8}}, U_{A_{r=64}}$: the right-singular unitary matrices by singular value decomposition

$$\phi(A_{r=8}, A_{r=64}, i, j) = \frac{||U_{A_{r=8}}^{i\top} U_{A_{r=64}}^{j}||_F^2}{\min(i, j)} \in [0, 1]$$

    - Grassmann distance:
      How much of the subspace spanned by the top i singular vectors in $U_{A_{r=8}}$
      is contained in the subspace by the top j singular vectors of $U_{A_{r=64}}$

Understanding the Low-Rank Updates

- What is the optimal rank r for LoRA?



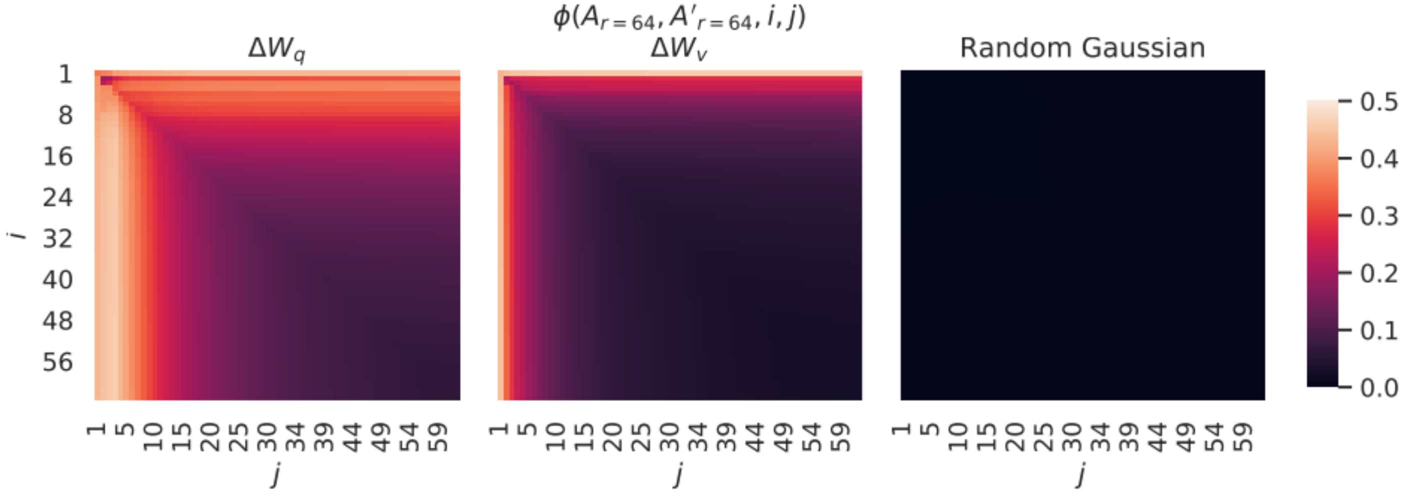
  - Directions corresponding to the top singular vector overlap between $U_{A_r=8}$ and $U_{A_r=64}$, while others do not

Understanding the Low-Rank Updates

- What is the optimal rank r for LoRA?
  - the top singular-vector directions of $U_{A_{r=8}}$ and $U_{A_{r=64}}$ are the most useful
    - ΔW_v and ΔW_q of $A_{r=8}$ and ΔW_v and ΔW_q of $A_{r=64}$ share a subspace of dimension 1 with normalized similarity > 0.5
      -> reason why $r = 1$ performs quite well
  - the adaptation matrix can indeed have a very low rank

Understanding the Low-Rank Updates

- What is the optimal rank r for LoRA?
  - Subspace similarity between different random seeds



  - ΔW_q appears to have a higher "intrinsic rank" than ΔW_v

Understanding the Low-Rank Updates

- How does the adaptation matrix ΔW compare to W?
  - project W onto the r-dimensional subspace of ΔW by computing $U^T W V^T$
  - compare the Frobenius norm

	$r=4$			$r=64$		
	ΔW_q	W_q	Random	ΔW_q	W_q	Random
$\|\|U^\top W_q V^\top\|\|_F =$	0.32	21.67	0.02	1.90	37.71	0.33
$\|\|W_q\|\|_F = 61.95$	$\|\|\Delta W_q\|\|_F = 6.91$			$\|\|\Delta W_q\|\|_F = 3.57$		

Understanding the Low-Rank Updates

- How does the adaptation matrix ΔW compare to W?

	$r=4$			$r=64$		
	ΔW_q	W_q	Random	ΔW_q	W_q	Random
$\|\|U^\top W_q V^\top\|\|_F =$	0.32	21.67	0.02	1.90	37.71	0.33
$\|\|W_q\|\|_F = 61.95$	$\|\|\Delta W_q\|\|_F = 6.91$			$\|\|\Delta W_q\|\|_F = 3.57$		

  - ΔW has a stronger correlation with W compared to a random matrix
    -> ΔW amplifies some features that are already in W
  - ΔW only amplifies directions that are not emphasized in W
  - the amplification factor is rather huge: $21.5 \approx 6.91/0.32$ for $r = 4$ (evidence for low-rank)

Understanding the Low-Rank Updates

- How does the adaptation matrix ΔW compare to W?

	$r=4$			$r=64$		
	ΔW_q	W_q	Random	ΔW_q	W_q	Random
$\lvert\lvert U^\top W_q V^\top \rvert\rvert_F =$	0.32	21.67	0.02	1.90	37.71	0.33
$\lvert\lvert W_q \rvert\rvert_F = 61.95$	$\lvert\lvert \Delta W_q \rvert\rvert_F = 6.91$			$\lvert\lvert \Delta W_q \rvert\rvert_F = 3.57$		

  - This suggests that the low-rank adaptation matrix potentially amplifies the important features for specific downstream tasks that were learned

Conclusion

- LoRA, an efficient adaptation strategy
  - eliminate inference latency
  - not reduces input sequence length while retaining high model quality
  - quick task-switching
  - generally applicable

Further Research

- QLoRA: Efficient Finetuning of Quantized LLMs
  - an efficient fine-tuning approach that reduces memory usage