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

ICLR 2022

HUMANE Lab 최종현

2025.03.06 랩 세미나

Background

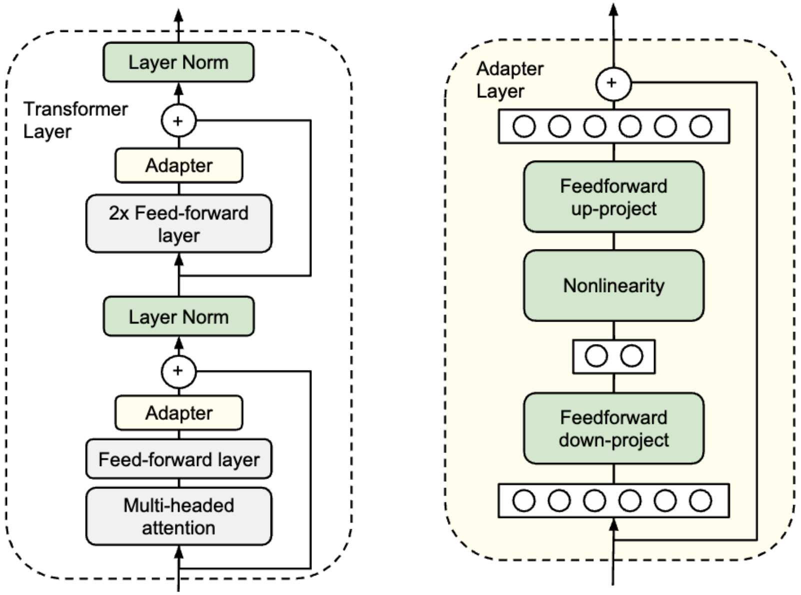
- Language models are getting larger and larger
- Needs fine-tuning to adapt to different domain / purpose
- Fine-tuning LLMs are expensive in terms of memory and compute (e.g., GPT-3)
- Pre-trained models have a low intrinsic dimensionality – most of the parameters aren't needed for effective adaptation (Li et al., 2018)

Previous Methods

- Full Fine-Tuning (FT)
  - Updates all parameters of the pre-trained model
  - Best possible performance for a given task
  - No inference latency
  - Requires lots of VRAM and computation power
  - Storage overhead
  - Not scalable for multiple tasks – training and storing different fine-tuned models is infe asible

Previous Methods

- Adapter Layers
  - Small trainable modules inserted into each Transformer layer
  - Original model weights are frozen – only adapter parameters are updated
  - Bottleneck structure
    - Down-projection: reduces dimensionality
    - Non-linearity: ReLU, GeLU
    - Up-projection: restores original size
  - Storage cost per task
  - Inference latency – each adapter introduces extra computations

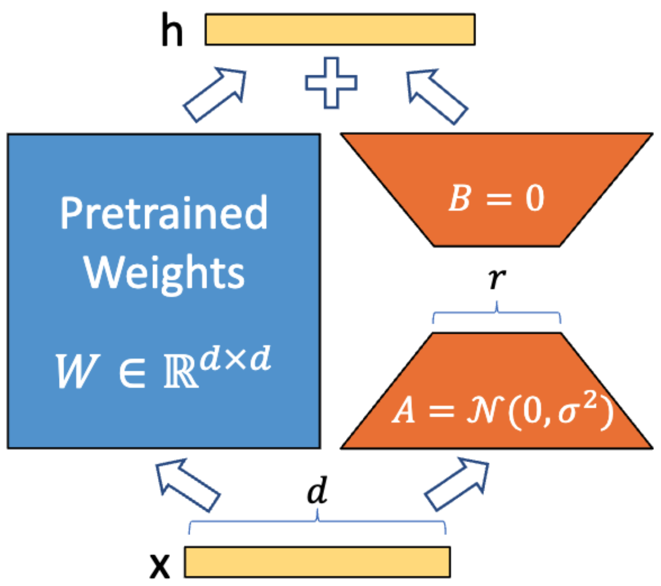


Previous Methods

- BitFit (Bias-Only Fine-Tuning)
  - Only update bias terms in the Transformer layers are updated
  - Lightweight alternative to full fine-tuning
  - Weaker performance
- Prefix-tuning
  - Learn a set of trainable ”prefix” tokens
  - Keeps the original model unchanged
  - Hard to optimize

What is LoRA?

- Low-Rank Adaptation
- Freezes the pre-trained model
- Injects trainable low-rank matrices (A and B) into Tr ansformer layers
- Without LoRA: $h = W_0 x$
- With LoRA: $h = W_0 x + \Delta W x = W_0 x + BAx$



How LoRA works

- Pre-trained model's weight: $W_0 \in \mathbb{R}^{d \times k}$
- Accumulated gradient update (Core idea)
  - $\Delta W = BA$
  - $B \in \mathbb{R}^{d \times r}$ (initialized with 0)
  - $A \in \mathbb{R}^{r \times k}$ (initialized with random Gaussian)
  - $r \ll \min(d, k) - r$ is decided with experiments
- $h = W_0 x + \Delta W x = W_0 x + BAx$
- Only update ΔW, thus only needing to learn $d \times r + r \times k$ instead of $d \times k$

How LoRA works

- Scaling factor - $\frac{\alpha}{r}$
  - r – same as previous r (used in matrices A and B)
  - α – same as r above (but can be tuned separately from r)
- $h = W_0 x + \frac{\alpha}{r} \Delta W x = W_0 x + \frac{\alpha}{r} BAx$

How LoRA works

- $h = W_0 x + \frac{\alpha}{r} \Delta W x = W_0 x + \frac{\alpha}{r} BAx$
- $h = W_q x + \frac{\alpha}{r} \Delta W x = W_q x + \frac{\alpha}{r} BAx$
- $h = W_k x + \frac{\alpha}{r} \Delta W x = W_k x + \frac{\alpha}{r} BAx$
- $h = W_v x + \frac{\alpha}{r} \Delta W x = W_v x + \frac{\alpha}{r} BAx$
- $h = W_O x + \frac{\alpha}{r} \Delta W x = W_O x + \frac{\alpha}{r} BAx$

Choosing r

- 'r' is selective with experiments
- Putting all parameters in ΔW_q, ΔW_k results in lower performance
- Adapting both yields the best result
- Preferable to adapt more weight matrices than a single type of weights with larger rank

	# 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**

Training

- Standard Objective

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

  - Model updates all its parameters
  - Parameters need to be stored and maintained for each task
  - If the pre-trained model is large (e.g., GPT-3), it's computationally expensive
- LoRA Objective

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

  - Pre-trained weights are kept frozen
  - Only low-rank matrices – A and B (collectively denoted as Θ)
  - Same loss, fewer parameters
  - During backpropagation, gradients are computed with respect to A and B (since W_0 is f rozen)

Result

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	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.89}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{72.0}_{\pm.2}$	$2.47_{\pm.02}$

Result

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 ($\text{Adapter}^{\text{H}}$)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 ($\text{Adapter}^{\text{H}}$)	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

Advantages

- Compute and memory efficient
- Reduce VRAM usage by up to 2/3 (on GPT-3, 1.2TB to 350GB)
- Checkpoint size reduced from 350GB to 35MB
- Can train with significantly fewer GPUs and avoid I/O bottlenecks
- Switch between tasks while deployed at a much lower cost by only swappin g the LoRA weights

Summary

- LoRA is an efficient way to train a pre-trained model
- Freezes pre-trained model and injects trainable matrices
- Can reduce memory footprint and computation
- Achieves similar or higher performance compared to fine-tuning

Q&A