Reduce the Number of Trainable Parameters

Hongrong Cheng

April 2, 2024

Outline

- A Single Linear Layer Yields Task-Adapted Low-Rank Matrices (LREC-COLING 2024)
- VERA: Vector-based Random Matrix Adaption (ICLR 2024)
- LORS: Low-rank Residual Structure for Parameter-Efficient Network Stacking (CVPR 2024)

A Single Linear Layer Yields Task-Adapted Low-Rank Matrices

Hwichan Kim[†*], **Shota Sasaki**[†], **Sho Hoshino**[†], **Ukyo Honda**[†]
†Tokyo Metropolitan University, ‡CyberAgent
†‡ Tokyo, Japan
kim-hwichan@ed.tmu.ac.jp, {sasaki_shota, hoshino_sho, honda_ukyo}@cyberagent.co.jp

Abstract

Low-Rank Adaptation (LoRA) is a widely used Parameter-Efficient Fine-Tuning (PEFT) method that updates an initial weight matrix W_0 with a delta matrix ΔW consisted by two low-rank matrices A and B. A previous study suggested that there is correlation between W_0 and ΔW. In this study, we aim to delve deeper into relationships between W_0 and low-rank matrices A and B to further comprehend the behavior of LoRA. In particular, we analyze a conversion matrix that transform W_0 into low-rank matrices, which encapsulates information about the relationships. Our analysis reveals that the conversion matrices are similar across each layer. Inspired by these findings, we hypothesize that a single linear layer, which takes each layer's W_0 as input, can yield task-adapted low-rank matrices. To confirm this hypothesis, we devise a method named Conditionally Parameterized LoRA (CondLoRA) that updates initial weight matrices with low-rank matrices derived from a single linear layer. Our empirical results show that CondLoRA maintains a performance on par with LoRA, despite the fact that the trainable parameters of CondLoRA are fewer than those of LoRA. Therefore, we conclude that "a single linear layer yields task-adapted low-rank matrices."

Keywords: Pretrained Language Model, Parameter-Efficient Fine-tuning, Low-Rank Adaptation

1. Introduction

In natural language processing (NLP) area, it is common practice to fine-tune pre-trained language models (PLMs) (Devlin et al., 2019; Lewis et al., 2020; Brown et al., 2020) using task-specific data. As the scale of these PLMs has grown considerably, the computational resources required for fine-tuning all parameters have escalated, presenting a substantial computational challenge. In recent years, parameter-efficient fine-Tuning (PEFT) methods, which use a limited number of additional parameters, have been proposed to address this issue. PEFT methods include prompt-tuning (Lester et al., 2021), prefix-tuning (Li and Liang, 2021), and low-rank adaptation (LoRA) (Hu et al., 2022), etc. These methods reduce computational costs to fine-tune PLMs while achieving comparable performance to fine-tuning all of the parameters.

Among the PEFT methods, LoRA has been prominent in NLP area because it shows stable and good performance across various NLP tasks and PLMs (Pu et al., 2023). LoRA fixes an initial weight matrix W_0 and updates W_0 with a delta matrix ΔW consisting of trainable low-rank matrices A and B, significantly reducing the number of trainable parameters compared to fine-tuning all parameters. Subsequent studies (Zhang et al., 2023; Valipour et al., 2023) have analyzed several aspects of LoRA to achieve potentially more effective and efficient PLM fine-tuning. Hu et al. (2022) performed an analysis of the relationship between W_0 and trained ΔW ($= BA$), and they revealed that there is a correlation between W_0 and ΔW. This finding implies the existence of certain relationships between the initial weight matrix W_0 and the low-rank matrices A and B.

In this study, we conduct an in-depth analysis of the relationships between initial weight matrices W_0 and low-rank matrices A and B to gain a deeper understanding of LoRA behavior. Specifically, we analyze a conversion matrix that transforms W_0 into A or B under the assumption that it roughly represents their relationships. Our analysis shows that similarities between each layer's conversion matrix are very high. This empirical observation implies a commonality in the relationships between the initial weight matrices and low-rank matrices regardless of layers. Inspired by the results, we hypothesize that a single linear layer, which takes each layer's W_0 as input, can produce task-adapted low-rank matrices of each layers.

To confirm our hypothesis, we design a method named Conditionally Parameterized LoRA (CondLoRA) that fine-tune PLMs with low-rank matrices derived from a single linear layer (Figure 2). Our experiments demonstrate that CondLoRA achieves competitive performance compared to LoRA in GLUE tasks. Notably, CondLoRA can reduce the number of trainable parameters compared to LoRA, because its parameters are constant regardless of target layers. The success of CondLoRA suggests potential avenues for further minimization of trainable parameters in LoRA variants. Our contributions in this study are twofold:

1. We reveal that conversion matrices that trans-

[*] This work was carried out during his CyberAgent internship period.

● Motivation

A previous study suggested that there is a correlation between W_0 and ΔW. In this study, the authors aim to delve deeper into relationships between W_0 and low-rank matrices A and B to further comprehend the behavior of LoRA.

● Contributions

(1) The authors reveal that conversion matrices that transform initial weight matrices into trained low rank matrices are similar across each layer.

(2) The authors demonstrate that CondLoRA achieves performance comparable to the already parameter-efficient LoRA with fewer parameters.

● Method - ① Discover the relationship between the conversion matrices

$$W_0^{m,l} + \Delta W^{m,l} = W_0^{m,l} + B^{m,l} A^{m,l} \quad ,$$

where $W_0^{m,l} \in \mathbb{R}^{d_1 \times d_2}$, $\Delta W^{m,l} \in \mathbb{R}^{d_1 \times d_2}$, $A^{m,l} \in \mathbb{R}^{r \times d_2}$, $B^{m,l} \in \mathbb{R}^{d_1 \times r}$ with r<< d_1, d_2 , $m \in \{m_1, \ldots, m_k\}$ and $l \in \{1,2, \ldots, N\}$ are target module (e.g., query, value, etc.) and layer, respectively, N is the total number of layers.

Conversion matrices $W_{0\to A}^{m,l}$ and $W_{0\to B}^{m,l}$ satisfy $(W_0^{m,l})^T W_{0\to A}^{m,l} = (A^{m,l})^T$ and $W_0^{m,l} W_{0\to B}^{m,l} = B^{m,l}$. Therefore, the conversion matrices are:

$$W_{0\to A}^{m,l} = ((W_0^{m,l})^T)^{-1} (A^{m,l})^T \in \mathbb{R}^{d_1 \times r}$$

$$W_{0\to B}^{m,l} = (W_0^{m,l})^{-1} B^{m,l} \in \mathbb{R}^{d_2 \times r}$$

Experimental Settings:

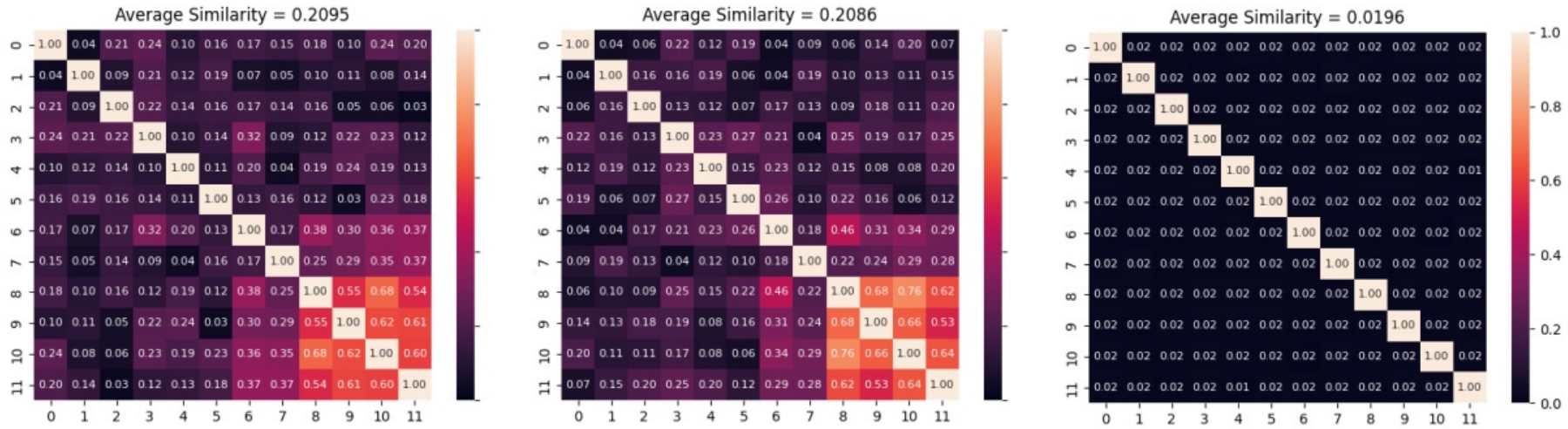
The authors used RoBERTa base as a base model and PEFT library and a single NVIDIA A100 40GB for LoRA tuning on GLUE dataset. Based on evaluation scores in development data, the authors searched learning rates through Optuna (Akiba et al., 2019) and selected the best checkpoint.

Similarity:

$$A = U\Sigma V^T$$

$$\phi(X, Y, i, j) = \frac{\|U_X^{i\top} U_Y^j\|_F^2}{\min(i, j)} \in [0, 1],$$

where X and Y are matrices, U_X is a left or right unitary matrix and U_X^i is top-i singular vectors of U_X.



(a) $W_{0\to A}^{\text{value},l}$ (b) $W_{0\to B}^{\text{value},l}$ (c) Random matrix

Figure 1: Normalized subspace similarities between each layer's conversion matrices and random matrices. Average similarity refers to the average of elements excluding the diagonal elements.

Method - ② Conditionally Parameterized LoRA (CondLoRA) fine-tunes PLMs with low-rank matrices derived from a single linear layer.

$$A_{cond}^{m,l} = Linear(\left(W_0^{m,l}\right)^T; \theta_A^m)^T \in \mathbb{R}^{r \times d_2} \quad ,$$

$$B_{cond}^{m,l} = Linear(W_0^{m,l}; \theta_B^m) \in \mathbb{R}^{d_1 \times r} \quad ,$$

$$\Delta W_{cond}^{m,l} = B_{cond}^{m,l} A_{cond}^{m,l} \quad ,$$

where $\theta_A^m \epsilon \mathbb{R}^{d_1 \times r}$ and $\theta_B^m \epsilon \mathbb{R}^{d_2 \times r}$ are trainable parameters. CondLoRA trains θ_A^m and θ_B^m using downstream task data.

One of the advantages of CondLoRA is its ability to decrease the numbers of trainable parameters. LoRA requires $(d_1 \times r + d_2 \times r) \times k \times N$ trainable parameters. However, CondLoRA requires $(d_1 \times r + d_2 \times r) \times k$ trainable parameters regardless of N, because it use a linear layer per target modules and low-rank matrices. To substantiate our hypothesis,

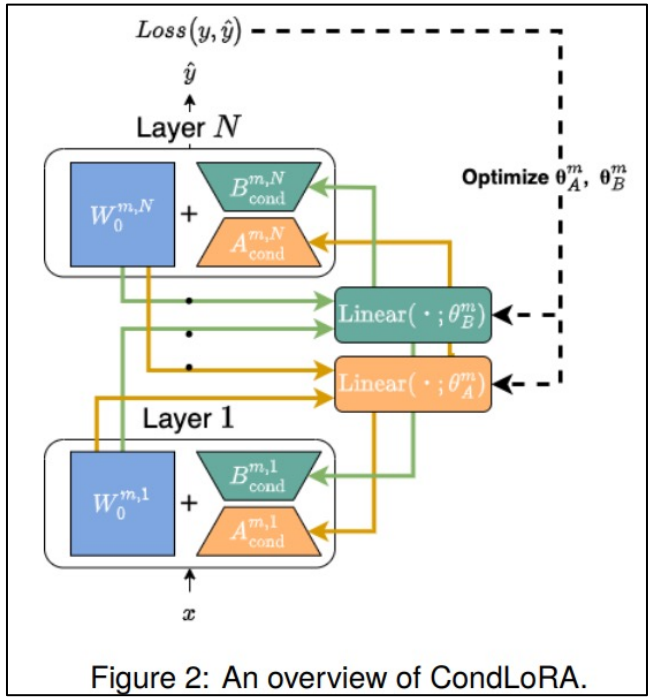


Figure 2: An overview of CondLoRA.

RoBERTa

	Trainable parameters	Speed (examples/s)
LoRA	294,912	39.652
CondLoRA	24,576	40.303

Table 4: Trainable parameters and speed during training.

GPU memory for training ↓

disk space for storing model ↓

computational complexity↑

The size of the original model and the inference time remain unchanged.

	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
LoRA	86.6	93.7	86.2	61.2	92.0	90.5	74.3	89.3	83.38
CondLoRA	86.5	93.8	86.6	61.1	91.8	90.1	74.2	90.3	83.42
Δ↑	-0.1	0.1	0.4	-0.1	-0.2	-0.4	-0.1	1.0	0.04

Table 2: Evaluation on GLUE tasks. These scores are the average of three models.

		l-th layer											
X	Y	1	2	3	4	5	6	7	8	9	10	11	12
$A^{\text{value},l}$	$A^{\text{value},l}_{\text{cond}}$	0.05	0.05	0.06	0.04	0.04	0.08	0.03	0.04	0.03	0.06	0.05	0.07
$B^{\text{value},l}$	$B^{\text{value},l}_{\text{cond}}$	0.04	0.03	0.03	0.05	0.05	0.04	0.04	0.03	0.02	0.07	0.08	0.10
$\Delta W^{\text{value},l}$	$\Delta W^{\text{value},l}_{\text{cond}}$	0.04	0.03	0.03	0.05	0.05	0.05	0.04	0.03	0.02	0.07	0.09	0.12

Table 3: Normalized subspace similarity between matrices from LoRA and CondLoRA.

VeRA: Vector-based Random Matrix Adaptation

Dawid J. Kopiczko[*†]
QUVA Lab
University of Amsterdam

Tijmen Blankevoort
Qualcomm AI Research[‡]

Yuki M. Asano
QUVA Lab
University of Amsterdam

Abstract

Low-rank adapation (LoRA) is a popular method that reduces the number of trainable parameters when finetuning large language models, but still faces acute storage challenges when scaling to even larger models or deploying numerous per-user or per-task adapted models. In this work, we present **Ve**ctor-based **R**andom Matrix **A**daptation (VeRA), which significantly reduces the number of trainable parameters compared to LoRA, yet maintains the same performance. It achieves this by using a single pair of low-rank matrices shared across all layers and learning small scaling vectors instead. We demonstrate its effectiveness on the GLUE and E2E benchmarks, image classification tasks, and show its application in instruction-tuning of 7B and 13B language models.

1 Introduction

In the era of increasingly large and complex language models, the challenge of efficient adaptation for specific tasks has become more important than ever. While these models provide powerful capabilities, their extensive memory requirements pose a significant bottleneck, particularly when adapting them for personalized use. Consider, for example, a cloud-based operating system assistant that continuously learns from and adapts to individual user behaviors and feedback. The need to store multiple checkpoints of finetuned models for each user rapidly escalates the required storage, even more so when multiple tasks come into play.

The situation is further exacerbated when we look at the state-of-the-art models like GPT-4 (OpenAI, 2023). Finetuning techniques like LoRA (Hu et al., 2022), while effective, still introduce considerable memory overhead. As an illustrative example, applying LoRA with a rank of 16 to the query and value layers of GPT-3 (Brown et al., 2020) would demand at least 288MB of memory, if stored in singe-precision – at a million finetuned weights, e.g., one per user, that would amount to 275TB.

Given the recent proliferation of language models and their deployment in personalized assistants, edge devices, and similar applications, efficient adaptation methods are paramount. We believe there is untapped potential for even more efficient approaches. Previous work (Aghajanyan et al., 2021) pointed out the low intrinsic dimensionality of pretrained models' features. These studies reported numbers much lower than the trainable parameters used in LoRA, suggesting there is room for improvement.

In parallel to this, recent research has shown the surprising effectiveness of models utilizing random weights and projections (Peng et al., 2021; Ramanujan et al., 2020; Lu et al., 2022; Schrimpf et al., 2021; Frankle et al., 2021). Such models serve as the basis of our proposed solution, **Ve**ctor-based **R**andom Matrix **A**daptation (VeRA), which minimizes the number of trainable parameters introduced during finetuning by reparametrizing the weights matrices. Specifically, we employ "scaling vectors" to adapt a pair of frozen random matrices shared between layers. With this approach, many more versions of the model can reside in the limited memory of a single GPU.

[*]dj.kopiczko@gmail.com; [‡]Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.
[†]Datasets were solely downloaded and evaluated by the University of Amsterdam.

● Motivation

For example, a cloud-based operating system assistant that continuously learns from and adapts to individual user behaviors and feedback. The need to store multiple checkpoints of finetuned models for each user rapidly escalates the required storage, even more so when multiple tasks come into play.

As an illustrative example, applying LoRA with a rank of 16 to the query and value layers of GPT-3 (Brown et al., 2020) would demand at least 288MB of memory, if stored in singe-precision – at a million finetuned weights, e.g., one per user, that would amount to 275TB.

● Contributions

Significantly reduces the number of trainable parameters compared to LoRA, while yielding comparable results.

Random Models and Projections

- Frankle & Carbin (2019) identified that randomly-initialized neural networks contain subnetworks that are capable of reaching high performance when trained.
- Aghajanyan et al. (2021) showed that training only a small number of parameters, randomly projected back into the full space, could achieve 90% of the full-parameter model performance.
- …

- **Method**

$$D = \begin{pmatrix} k_1 & 0 & \cdots & 0 \\ 0 & k_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & k_m \end{pmatrix}$$

$$h = W_0 x + \Delta W x = W_0 x + \underline{\Lambda_b} B \underline{\Lambda_d} A x \ ,$$

where a pretrained weight matrix $W_0 \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{m \times r}$, $A \in \mathbb{R}^{r \times n}$, r <<min(m,n), $\Lambda_b \in \mathbb{R}^{m \times m}$ and $\Lambda_d \in \mathbb{R}^{r \times r}$ are diagonal matrices, the scaling vectors b and d are trainable.

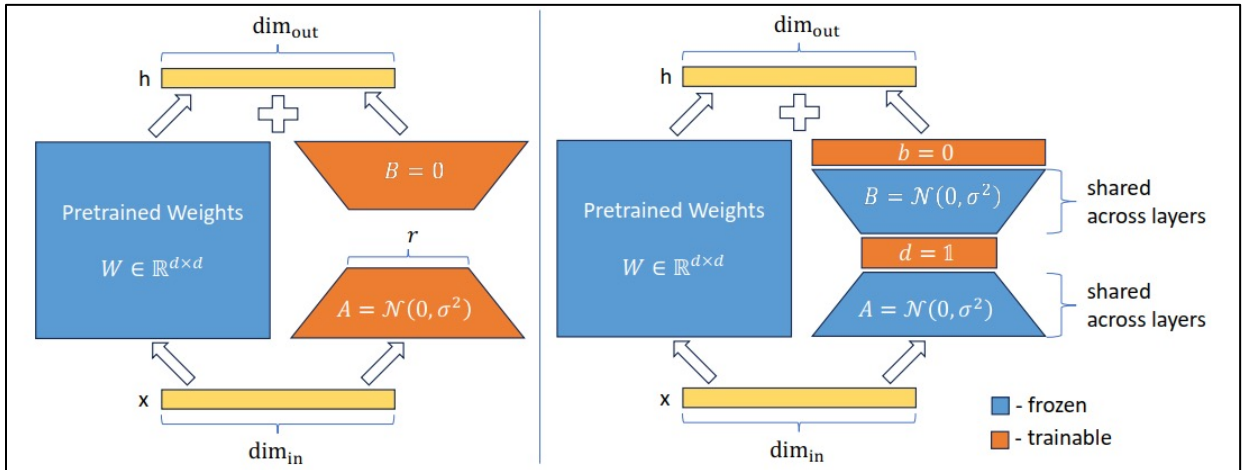


Figure 1: Schematic comparison of LoRA (left) and VeRA (right). LoRA updates the weights matrix W by training the low-rank matrices A and B, with intermediate rank r. In VeRA these matrices are frozen, shared across all layers, and adapted with trainable vectors d and b, substantially reducing the number of trainable parameters. In both cases, low-rank matrices and vectors can be merged into original weights matrix W, introducing no additional latency.

3.2 PARAMETER COUNT

Table 1: Theoretical memory required to store trained VeRA and LoRA weights for RoBERTa$_{base}$, RoBERTa$_{large}$ and GPT-3 models. We assume that LoRA and VeRA methods are applied on query and key layers of each transformer block.

	Rank	LoRA # Trainable Parameters	LoRA Required Bytes	VeRA # Trainable Parameters	VeRA Required Bytes
BASE	1	36.8K	144KB	0.5	18.4K	72KB
	16	589.8K	2MB	0.032	18.8K	74KB
	256	9437.1K	36MB	0.0026	24.5K	96KB
LARGE	1	98.3K	384KB		49.2K	192KB
	16	1572.8K	6MB		49.5K	195KB
	256	25165.8K	96MB		61.4K	240KB
GPT-3	1	4.7M	18MB		2.4M	9.1MB
	16	75.5M	288MB		2.8M	10.5MB
	256	1207.9M	4.6GB		8.7M	33MB

$d_{model} = 768$

$L_{model} = 12$

$|\Theta| = 2 \times 12 \times 768 \times 1 \times 2 = 36864$

$36864 \times 4 \div 1024 = 144KB$

$d_{model} = 768$

$L_{model} = 12$

$|\Theta| = 12 \times (768 + 1) \times 2 = 18456$

$18456 \times 4 \div 1024 = 72.09KB$

b d

layers. The number of trainable parameters in VeRA is then governed by $|\Theta| = L_{\text{tuned}} \times (d_{\text{model}} + r)$, contrasting with LoRA's $|\Theta| = 2 \times L_{\text{tuned}} \times d_{\text{model}} \times r$. Specifically, for the lowest rank (i.e., $r = 1$), VeRA requires approximately half the trainable parameters of LoRA. Moreover, as the

3.3 INITIALIZATION STRATEGIES

- **Shared Matrices**: In our method, we employ Kaiming initialization (He et al., 2015) for the frozen low-rank matrices A and B. By scaling the values based on matrix dimensions, it ensures that a matrix product of A and B maintains a consistent variance for all ranks, eliminating the need to finetune the learning rate for each rank.
- **Scaling Vectors**: The scaling vector b is initialized to zeros, which aligns with the initialization of matrix B in LoRA and ensures that the weight matrix is unaffected during the first forward pass. The scaling vector d is initialized with a single non-zero value across all its elements, thereby introducing a new hyperparameter that may be tuned for better performance.

RoBERTa

Table 2: Results for different adaptation methods on the GLUE benchmark. We report Matthew's correlation for CoLA, Pearson correlation for STS-B, and accuracy for the remaining tasks. In all cases, higher values indicate better performance. Results of all methods except VeRA are sourced from prior work (Hu et al., 2022; Zhang et al., 2023a). VeRA performs on par with LoRA with an order of magnitude fewer parameters.

	Method	# Trainable Parameters	SST-2	MRPC	CoLA	QNLI	RTE	STS-B	Avg.
BASE	FT	125M	94.8	90.2	63.6	92.8	78.7	91.2	85.2
	BitFit	0.1M	93.7	**92.7**	62.0	91.8	81.5	90.8	85.4
	Adpt^{D}	0.3M	$94.2_{\pm0.1}$	$88.5_{\pm1.1}$	$60.8_{\pm0.4}$	$93.1_{\pm0.1}$	$71.5_{\pm2.7}$	$89.7_{\pm0.3}$	83.0
	Adpt^{D}	0.9M	$94.7_{\pm0.3}$	$88.4_{\pm0.1}$	$62.6_{\pm0.9}$	$93.0_{\pm0.2}$	$75.9_{\pm2.2}$	$90.3_{\pm0.1}$	84.2
	LoRA	0.3M	$\mathbf{95.1}_{\pm0.2}$	$89.7_{\pm0.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm0.3}$	$\mathbf{86.6}_{\pm0.7}$	$\mathbf{91.5}_{\pm0.2}$	**86.6**
	VeRA	**0.043M**	$94.6_{\pm0.1}$	$89.5_{\pm0.5}$	$\mathbf{65.6}_{\pm0.8}$	$91.8_{\pm0.2}$	$78.7_{\pm0.7}$	$90.7_{\pm0.2}$	85.2
LARGE	Adpt^{P}	3M	$96.1_{\pm0.3}$	$90.2_{\pm0.7}$	$\mathbf{68.3}_{\pm1.0}$	$\mathbf{94.8}_{\pm0.2}$	$83.8_{\pm2.9}$	$92.1_{\pm0.7}$	87.6
	Adpt^{P}	0.8M	$\mathbf{96.6}_{\pm0.2}$	$89.7_{\pm1.2}$	$67.8_{\pm2.5}$	$\mathbf{94.8}_{\pm0.3}$	$80.1_{\pm2.9}$	$91.9_{\pm0.4}$	86.8
	Adpt^{H}	6M	$96.2_{\pm0.3}$	$88.7_{\pm2.9}$	$66.5_{\pm4.4}$	$94.7_{\pm0.2}$	$83.4_{\pm1.1}$	$91.0_{\pm1.7}$	86.8
	Adpt^{H}	0.8M	$96.3_{\pm0.5}$	$87.7_{\pm1.7}$	$66.3_{\pm2.0}$	$94.7_{\pm0.2}$	$72.9_{\pm2.9}$	$91.5_{\pm0.5}$	84.9
	LoRA-FA	3.7M	96.0	90.0	68.0	94.4	86.1	92.0	87.7
	LoRA	0.8M	$96.2_{\pm0.5}$	$90.2_{\pm1.0}$	$68.2_{\pm1.9}$	$\mathbf{94.8}_{\pm0.3}$	$85.2_{\pm1.1}$	$\mathbf{92.3}_{\pm0.5}$	**87.8**
	VeRA	**0.061M**	$96.1_{\pm0.1}$	$\mathbf{90.9}_{\pm0.7}$	$68.0_{\pm0.8}$	$94.4_{\pm0.2}$	$\mathbf{85.9}_{\pm0.7}$	$91.7_{\pm0.8}$	**87.8**

GPT2

Table 3: Results for different adaptation methods on the E2E benchmark and GPT2 Medium and Large models. Results with (^{1,2,3}) are taken from prior work: ^{1}(Hu et al., 2022), ^{2}(Valipour et al., 2022), ^{3}(Zi et al., 2023). VeRA outperforms LoRA with 3 and 4 times less trainable parameters, for GPT2 Medium and Large respectively.

	Method	# Trainable Parameters	BLEU	NIST	METEOR	ROUGE-L	CIDEr
MEDIUM	FT^{1}	354.92M	68.2	8.62	46.2	71.0	2.47
	$\text{Adpt}^{\text{L}1}$	0.37M	66.3	8.41	45.0	69.8	2.40
	$\text{Adpt}^{\text{L}1}$	11.09M	68.9	8.71	46.1	71.3	2.47
	$\text{Adpt}^{\text{H}1}$	11.09M	67.3	8.50	46.0	70.7	2.44
	DyLoRA^{2}	0.39M	69.2	8.75	46.3	70.8	2.46
	AdaLoRA^{3}	0.38M	68.2	8.58	44.1	70.7	2.35
	LoRA	0.35M	68.9	8.69	46.4	71.3	**2.51**
	VeRA	**0.098M**	**70.1**	**8.81**	**46.6**	**71.5**	2.50
LARGE	FT^{1}	774.03M	68.5	8.78	46.0	69.9	2.45
	$\text{Adpt}^{\text{L}1}$	0.88M	69.1	8.68	46.3	71.4	2.49
	$\text{Adpt}^{\text{L}1}$	23.00M	68.9	8.70	46.1	71.3	2.45
	LoRA	0.77M	70.1	8.80	46.7	**71.9**	2.52
	VeRA	**0.17M**	**70.3**	**8.85**	**46.9**	71.6	**2.54**

Table 4: Average scores on MT-Bench assigned by GPT-4 to the answers generated by models fine-tuned with VeRA and LoRA methods, and the base Llama 13B model. VeRA closely matches performance of LoRA on the instruction-following task, with 100x reduction in trainable parameters.

rank = 64

Model	Method	# Parameters	Score ↑
Llama 13B	-	-	2.61
LLAMA 7B	LoRA	159.9M	5.03
	VeRA	1.6M	4.77
LLAMA 13B	LoRA	250.3M	5.31
	VeRA	2.4M	5.22
LLAMA2 7B	LoRA	159.9M	5.19
	VeRA	1.6M	5.08
LLAMA2 13B	LoRA	250.3M	5.77
	VeRA	2.4M	5.93

Table 5: Vision models finetuned with VeRA and LoRA on different image classification datasets. VeRA approaches performance of LoRA for the smaller model, and outperforms it in the case of the large model, with over 10x fewer trainable parameters.

	Method	# Trainable Parameters	CIFAR100	Food101	Flowers102	RESISC45
ViT-B	Head	-	77.7	86.1	98.4	67.2
	Full	85.8M	**86.5**	**90.8**	98.9	**78.9**
	LoRA	294.9K	85.9	89.9	98.8	77.7
	VeRA	**24.6K**	84.8	89.0	**99.0**	77.0
ViT-L	Head	-	79.4	76.5	98.9	67.8
	Full	303.3M	86.8	78.7	98.8	**79.0**
	LoRA	786.4K	87.0	**79.5**	99.1	78.3
	VeRA	**61.4K**	**87.5**	79.2	**99.2**	78.6

To evaluate the training time and GPU memory benefits of our method, we conducted a comparison between LoRA and VeRA while fine-tuning LLaMA 7B with the same rank (64) on instruction tuning dataset, introduced earlier in this work. The results are summarized in Table 12:

Table 12: Impact on GPU memory usage and training time.

Method	Training Time	GPU Memory
LoRA	**568** min	23.42GB
VeRA	578 min	**21.69GB**

While VeRA includes more operations than LoRA because of the additional vector multiplies in the forward pass, we find that it only results in a modest 1.8% increase in training time. For the GPU memory, we observe a 7.4% reduction in memory usage with VeRA, as it does not require storing optimizer states and gradients for shared random matrices.

GPU memory for training ↓

disk space for storing model ↓

computational complexity↑

The size of the original model and the inference time remain unchanged.

LORS: Low-rank Residual Structure for Parameter-Efficient Network Stacking

Jialin Li, Qiang Nie, Weifu Fu, Yuhuan Lin, Guangpin Tao, Yong Liu, Chengjie Wang
Youtu Lab, Tencent
{jarenli, ryanwfu, gleelin, guangpintao, choasliu, jasoncjwang}@tencent.com, qnie.cuhk@gmail.com

Abstract

Deep learning models, particularly those based on transformers, often employ numerous stacked structures, which possess identical architectures and perform similar functions. While effective, this stacking paradigm leads to a substantial increase in the number of parameters, posing challenges for practical applications. In today's landscape of increasingly large models, stacking depth can even reach dozens, further exacerbating this issue. To mitigate this problem, we introduce ***LORS*** *(****LO****w-rank* ***R****esidual* ***S****tructure). LORS allows stacked modules to share the majority of parameters, requiring a much smaller number of unique ones per module to match or even surpass the performance of using entirely distinct ones, thereby significantly reducing parameter usage. We validate our method by applying it to the stacked decoders of a query-based object detector, and conduct extensive experiments on the widely used MS COCO dataset. Experimental results demonstrate the effectiveness of our method, as even with a 70% reduction in the parameters of the decoder, our method still enables the model to achieve comparable or even better performance than its original.*

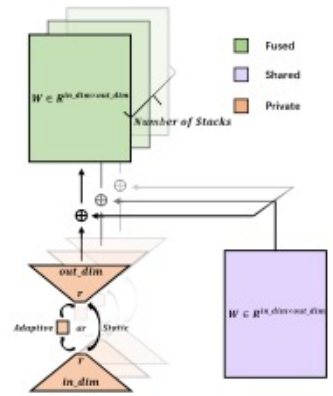


Figure 1. The LORS calculation process, which could be adaptive or static, depending on whether an adaptively generated kernel is used in the matrix manipulation for private parameters.

1. Introduction

In the current era of prosperity for large models, a common issue is the significant increase in the number of parameters, which presents challenges for training, inference, and deployment. Various methods have been proposed to reduce the number of parameters in models, such as knowledge distillation [15, 20], which compresses large models into smaller ones while trying to preserve their performance but may still lead to a decrease in model capacity; pruning [16, 61], which removes redundant parameters from the model but can affect the model's stability; quantization [8], which reduces the numerical precision of model parameters to lower storage and computation but may cause model accuracy loss; and parameter sharing [26], which reduces the number of parameters by sharing them across different layers but may limit the model's expressiveness.

Different from the aforementioned methods, we have observed an important fact contributing to the large number of parameters: the widespread use of stacking in neural networks. Stacking refers to those modules that have identical architectures and perform the same or similar functions, but possess different parameters due to random initialization as well as training updates. Examples of stacking can be found in numerous prominent neural networks, like the classic ResNet model [18] and Transformers [50]. Particularly, Transformers heavily rely on stacked structures and typically employ completely identical multi-layer stacks in both encoders and decoders. It now serves as an indispensable component of many excellent models in fields such as computer vision and natural language processing.

Although stacking is powerful for enhancing model capacity, as demonstrated by large language models, it also

● Motivation

Although stacking is powerful for enhancing model capacity, as demonstrated by large language models, it also naturally leads to a sharp increase in the number of parameters. How can we enjoy the benefits of stacking while reducing the required number of parameters?

● Contributions

Propose to decompose the parameters of stacked modules into two parts: shared ones representing the commonality and private ones capturing the specific characteristics. Shared parameters are available for all modules and trained jointly by them, while the private parameters are separately owned by each module.

To achieve this goal, the authors introduce the concept of **Lowrank Residual Structure (LORS)**, inspired by the approach of LoRA.

● Method

Take the query-based object detector, **AdaMixter**, as an example. AdaMixter contains a backbone (ResNet50) and a decoder that includes ACM (Adaptive Channel Mixing) utilizes a weight adapted by the object query ***q*** to transform feature x in the channel dimension, enhancing channel semantics. ASM (Adaptive Spatial Mixing) process, which aims to enable the adaptability of the object query ***q*** to spatial structures of sampled features.

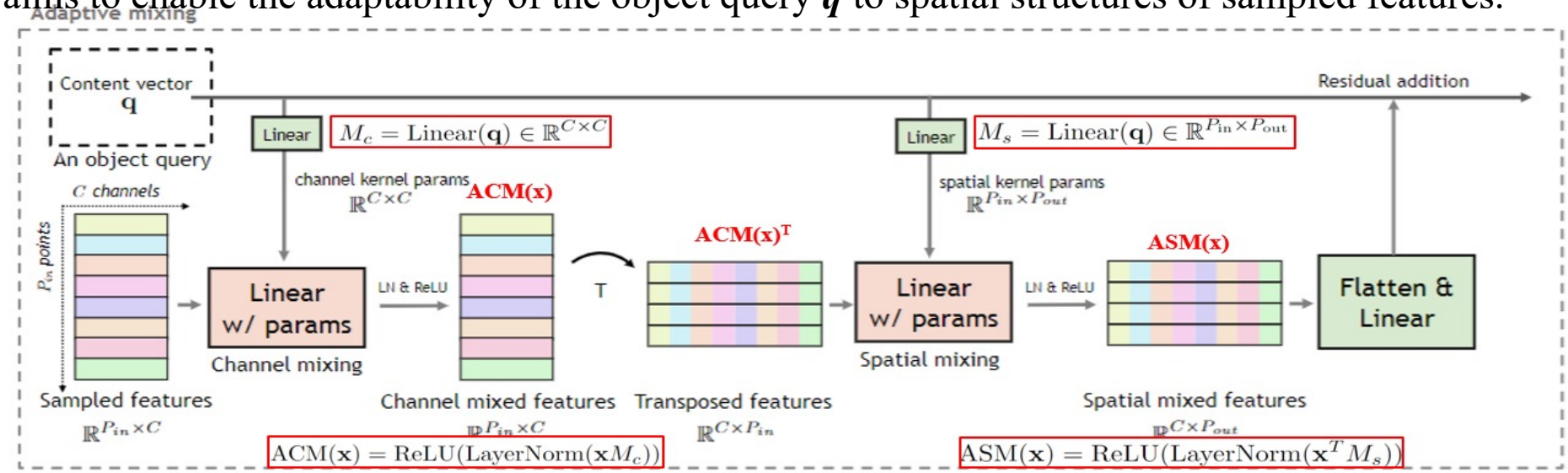


Figure 3. **Adaptive mixing procedure** between an object query and sampled features. The object query first generates adaptive mixing weights and then apply these weights to mix sampled features in the channel and spatial dimension. Note that for clarity, we demonstrate adaptive mixing for one sampling group.

- **Adaptive Low Rank Residual Structure (LORS^{A}).**

$$\hat{W}_i = \hat{W}^{\text{shared}} + \hat{W}_i^{\text{private}} ,$$

where $\widehat{W}_i \in \mathbb{R}^{d\times h}$ is an adaptive generated parameter in i-th stacked layer, the cross-layer shared parameter $\widehat{W}^{shared} \in \mathbb{R}^{d\times h}$ and the layer-specific parameter $\widehat{W}_i^{private} \in \mathbb{R}^{d\times h}$ for each layer are both calculated based on $\boldsymbol{q}$:

$$\hat{W}^{\text{shared}} = \text{Linear}(\mathbf{q}) \in \mathbb{R}^{d\times h} ,$$

$$\hat{W}_i^{\text{private}} = \sum_{k=1}^{K} \hat{B}_{ik}\hat{E}_{ik}\hat{A}_{ik} ,$$

$$\hat{E}_{ik} = \text{Linear}(\mathbf{q}) \in \mathbb{R}^{r\times r} ,$$

where $\hat{B}_{ik} \in \mathbb{R}^{d\times r}$ and $\hat{A}_{ik} \in \mathbb{R}^{r\times h}$, $r << \min(d, h)$, K represents the number of parameter groups used to compute $\widehat{W}_i^{private}$.

- **Static Low Rank Residual Structure (LORS^T).**

$$W_i = W^{\text{shared}} + W_i^{\text{private}},$$

where $W_i \in \mathbb{R}^{d\times h}$ is a parameter matrix belonging to the *i*-th layer, $W^{shared} \in \mathbb{R}^{d\times h}$ represents the shared parameters across all stacked layers, $W_i^{private} \in \mathbb{R}^{d\times h}$ denotes the layer-specific parameters for the i-th layer, which is calculated as follows:

$$W_i^{\text{private}} = \sum_{k=1}^{K} B_{ik} A_{ik},$$

where $B_{ik} \in \mathbb{R}^{d\times r}$ and $A_{ik} \in \mathbb{R}^{r\times h}$, $r \ll \min(d, h)$, and K represents the number of parameter groups used to compute $W_i^{private}$.

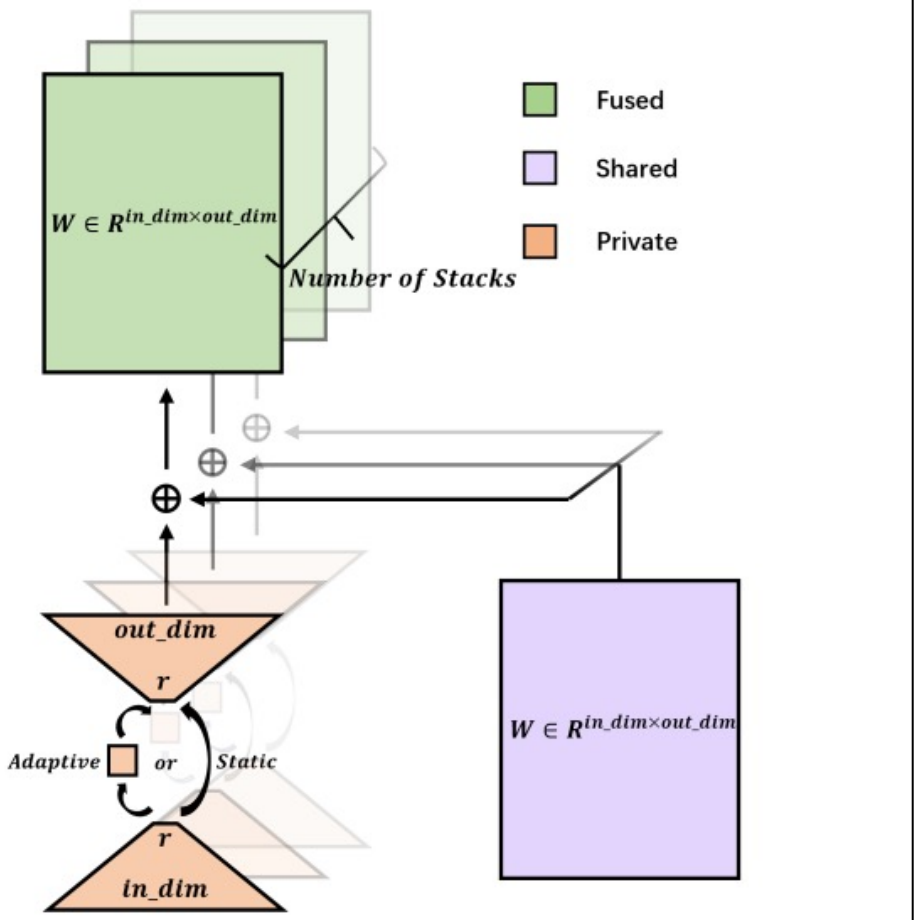


Figure 1. The LORS calculation process, which could be adaptive or static, depending on whether an adaptively generated kernel is used in the matrix manipulation for private parameters.

While LoRA is originally designed for fine-tuning, the authors train the LoRA-like operation on parameters from scratch.

Initialization Strategies We tried various initialization methods for each component in LORS and determined the overall initialization method as follows:

- **LORS$^{\mathbf{T}}$**: For static LORS, we employ Kaiming initialization [17] for W^{shared} and each B, and zero initialization for each A.
- **LORS$^{\mathbf{A}}$**: For adaptive LORS, we apply Kaiming initialization [17] to the linear transformation weights forming each $\hat{W}^{\text{shared}}$, as well as each $\hat{B}$ and $\hat{A}$. Additionally, we use zero initialization for the linear transformation weights forming each $\hat{E}$.

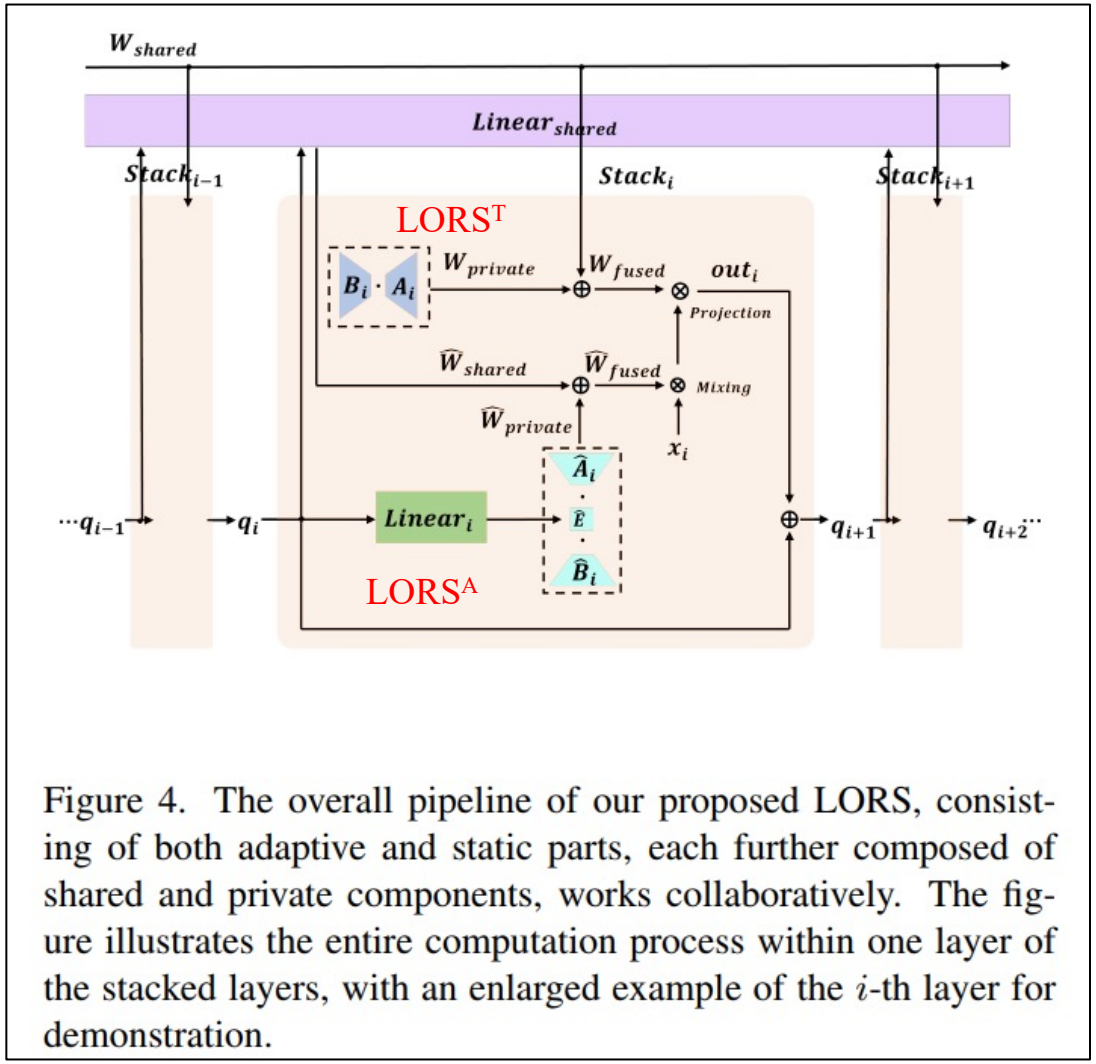


Figure 4. The overall pipeline of our proposed LORS, consisting of both adaptive and static parts, each further composed of shared and private components, works collaboratively. The figure illustrates the entire computation process within one layer of the stacked layers, with an enlarged example of the i-th layer for demonstration.

- **Analysis on Parameter Reduction**

Let $W \in \mathbb{R}^{d \times h}$ be a weight parameter that exists in every layer of stacked structures, N is the number of stacked layers. If static, it originally has $d \times h$ parameters, while using LORS$^{\mathrm{T}}$ requires $\frac{1}{N} \times d \times h + K \times (d \times r + r \times h)$ parameters on average per layer; if it is adaptive, generating it by q

ters on average per layer; if it is adaptive, generating it by q
with linear transformation requires $d_q \times d \times h$ parameters, where d_q is the dimension of q, and using LORS$^{\mathrm{A}}$ requires $\frac{1}{N} \times d_q \times d \times h + K \times (d_q \times r^2 + d \times r + r \times h)$ parameters on average per layer. To more intuitively display

		r=4	r=8	r=16	r=32
After	W/ . LORS^{T}	1.53M	1.66M	1.93M	2.46M
Before	W/O. LORS^{T}	8.39M	8.39M	8.39M	8.39M
	Percentage	18.3%	19.8%	23.0%	29.3%
After	W/ . LORS^{A}	0.36M	0.39M	0.49M	0.89M
Before	W/O. LORS^{A}	2.10M	2.10M	2.10M	2.10M
	Percentage	17.2%	18.6%	23.3%	42.4%

Table 1. Analysis of the parameter reduction effect of LORS^{T} and LORS^{A} with varying rank r values. Our default selections are colored gray. Here the average amount of shared parameters over each layer has already been taken into account.

Method	Queries	Epochs	Decoder Params(M)	Params(M)	Decoder GFLOPs	GFLOPs	Training Hours	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
AdaMixer	100	12	110	135	**12**	**104**	**8.5h**	**42.7**	**61.5**	45.9	24.7	45.4	**59.2**
AdaMixer + LORS	100	12	**35**	**60**	18	110	9.5h	42.6	61.4	**46.0**	**25.0**	**45.6**	58.5
AdaMixer	300	12	113	139	**39**	**132**	**10h**	**44.1**	**63.4**	47.4	27.0	46.9	**59.5**
AdaMixer + LORS	300	12	**35**	**60**	56	149	12h	**44.1**	63.0	**47.7**	**27.8**	**47.0**	**59.5**

Table 2. **1× training scheme** performance on COCO 2017 val set with ResNet-50 as backbone. AdaMixer with LORS can achieve competitive results while employing a notably reduced number of parameters. FPS is obtained with a single Nvidia V100 GPU.

Method	backbone	Queries	Epochs	Params(M)	GFLOPs	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
AdaMixer	R-50	100	36	135	**104**	43.2	61.8	46.7	25.0	46.1	58.8
AdaMixer + LORS	R-50	100	36	**60**	110	**43.7**	**62.3**	**47.3**	**25.5**	**46.4**	**60.0**
AdaMixer	R-50	300	36	139	**132**	47.0	66.0	51.1	30.1	**50.2**	61.8
AdaMixer + LORS	R-50	300	36	**60**	149	**47.6**	**66.6**	**52.0**	**31.1**	**50.2**	**62.5**
AdaMixer	R-101	300	36	158	**208**	48.0	67.0	52.4	30.0	51.2	63.7
AdaMixer + LORS	R-101	300	36	**79**	225	**48.2**	**67.5**	**52.6**	**31.7**	**51.3**	**63.8**
AdaMixer	Swin-S	300	36	164	**234**	51.3	71.2	55.7	34.2	54.6	67.3
AdaMixer + LORS	Swin-S	300	36	**85**	250	**51.8**	**71.6**	**56.4**	**35.4**	**55.0**	**68.4**

Table 3. **3× training scheme** performance on COCO 2017 val set, considering different combinations of backbone and query numbers. Longer training time allows LORS to be fully trained and perform better. The comprehensive improvement in performance and the significant reduction in parameters demonstrate the effectiveness of LORS. The best results in each category are highlighted in bold.

GPU memory for training ↓

disk space for storing model ↓

computational complexity↑

The size of the original model and the inference time remain unchanged.

Parameter Groups	Rank r	Decoder Params(M)	AP
1 1 1 1 1 1	8	32	41.8
1 1 1 1 1 1	16	34	42.1
2 2 2 2 2 2	16	35	42.4
1 1 2 2 3 3	16	35	42.6
1 1 1 1 1 1	32	38	41.2

Table 6. Validation accuracy with different groups of parameters and rank r in LORS^{A}. Default choice for our model is colored gray. Each row's first six numbers indicate the number of parameter groups sequentially used in each of the six decoder layers.

Parameter Groups	Rank r	Decoder Params(M)	AP
1 1 1 1 1 1	2	34	42.3
1 1 1 1 1 1	4	34	41.9
1 1 1 1 1 1	8	35	42.6
1 1 1 1 1 1	16	37	42.1
1 1 1 1 1 1	32	40	41.7

Table 7. Validation accuracy with different rank r in LORS^{T}. Default choice for our model is colored gray. Each row's first six numbers indicate the number of parameter groups sequentially used in each of the six decoder layers.

Decoder Number	Decoder Params(M)	AP
3	28	38.9
6	35	42.6
9	42	42.0
12	50	41.0

Table 8. Model performance using different numbers of stacked decoders when employing LORS. The default choice for our model is colored gray. This experiment with varying numbers of decoders demonstrates that six layers still yield the best performance for the model, which is not changed by LORS.

LORS^{A}	LORS^{T}	Decoder Params(M)	AP
		110	42.5†
✓		79	42.6
	✓	66	42.6
✓	✓	35	42.6

Table 4. Effect of LORS^{A} and LORS^{T}. "†" denotes this AP result was reproduced by ourselves. Both LORS^{T} and LORS^{A} can reduce parameters without compromising performance.

Conclusion

- **A Single Linear Layer Yields Task-Adapted Low-Rank Matrices (LREC-COLING 2024)**

$$A_{cond}^{m,l} = Linear((W_0^{m,l})^T; \theta_A^m)^T \in \mathbb{R}^{r \times d_2}$$
$$B_{cond}^{m,l} = Linear(W_0^{m,l}; \theta_B^m) \in \mathbb{R}^{d_1 \times r}$$
$$\Delta W_{cond}^{m,l} = B_{cond}^{m,l} A_{cond}^{m,l}$$

- **VERA: Vector-based Random Matrix Adaption (ICLR 2024)**

$$h = W_0 x + \Delta W x = W_0 x + \underline{\Lambda_b} B \underline{\Lambda_d} A x$$

- **LORS: Low-rank Residual Structure for Parameter-Efficient Network Stacking (CVPR 2024)**

$$\hat{W}_i = \hat{W}^{\text{shared}} + \boxed{\hat{W}_i^{\text{private}}} \rightarrow \hat{W}_i^{\text{private}} = \sum_{k=1}^{K} \hat{B}_{ik} \hat{E}_{ik} \hat{A}_{ik}$$

$$W_i = W^{\text{shared}} + \boxed{W_i^{\text{private}}} \rightarrow W_i^{\text{private}} = \sum_{k=1}^{K} B_{ik} A_{ik}$$

GPU memory for training ↓

disk space for storing model ↓

computational complexity↑

The size of the original model and the inference time remain unchanged.

Thanks