Reducing Memory Footprint in Deep Network Training by Gradient Space Reutilization

Yiming Dong[1] and Zhouchen Lin[1,2]⋆

[1] State Key Lab of General Artificial Intelligence, School of Intelligence Science and Technology, Peking University

[2] Pazhou Laboratory (Huangpu), Guangzhou, Guangdong, China

yimingdong_ml@outlook.com, zlin@pku.edu.cn

Abstract. As deep learning continues to spearhead transformative breakthroughs across various domains, the computational and memory demands for training state-of-the-art models have surged exponentially. This escalation not only challenges the scalability of deep learning systems but also significantly increases the financial cost associated with training. Memory-intensive operations, particularly during the optimization phase of training large models, can drastically inflate budgets, making cutting-edge research and applications less accessible. In response to this challenge, we introduce a novel technique termed *gradient space reutilization*, aiming at reducing memory usage in deep network training by repurposing the memory allocated for gradients once it is no longer needed in the later computing process. This approach is implemented across modified versions of popular optimizers, with their names AdamW-R, Adan-R, and Lion-R, respectively, demonstrating appreciable memory savings without compromising the performance as they are equivalent to the original algorithms. Extensive experiments demonstrate that our simple engineering trick can achieve up to 25.60% memory savings at the best, providing a practical solution for efficient resource management in deep learning training environments.

Keywords: Deep learning optimization · Efficient deep learning · Memory footprint.

1 Introduction

Deep learning [24] has revolutionized the landscape of artificial intelligence, yielding groundbreaking advancements across a myriad of applications, including natural language processing [1,5,32], computer vision [12,16,17], and autonomous systems [4]. Leveraging multi-layer neural networks, deep learning models have demonstrated an unparalleled ability to learn from vast amount of data, resulting in performances that often surpass human expertise. As these models continue to push the boundaries of what is computationally possible, they have become integral components in both academic research and industry solutions, driving innovation and progress [9,39].

⋆ Corresponding author.

However, this rapid expansion in capabilities has not come without challenges. One of the most pressing issues is the substantial memory requirement for training these sophisticated models, particularly in the current era of large language models (LLMs) [20], which is continuously inducing significant financial cost. While training a medium-sized models like BERT-Large [11] costs tens of thousands of dollars, a single training run for GPT-3 could reach the cost of $12M [27]. This limitation not only impacts the scalability of deep learning models but also restricts the democratization of these technologies, as only those with access to high-end computational resources can effectively engage in state-of-the-art model development [40].

Indeed, the memory demands for training large-scale neural networks can be staggering. For example, a LLaMA-7B model [38], trained with the AdamW optimizer [26], requires holding not just gradients, but also the first and second moment estimates for each parameter. In the naive setting, this leads to a memory requirement of over 112GB solely for these optimization-related variables, let alone the storage for input data and activations. Currently, such extensive memory requirements are beyond the capacity of most of the parallel computing devices like GPUs, making large model training inaccessible to the broader AI communities.

As such, recognizing the need for *efficient memory usage* in deep learning is paramount for sustaining the growth and accessibility of this transformative technology. Our work addresses this problem by identifying and capitalizing on opportunities to reduce the memory footprint during the optimization phase. Our main findings are that, for many mainstream optimizers, *the memory space allocated for gradients can be repurposed once it is no longer needed in subsequent computations*. We exemplify this idea on the classic AdamW [26], as well as the newer Adan [42] and Lion [7] optimizers, deriving AdamW-R, Adan-R, and Lion-R, respectively, to demonstrate how this memory reutilization strategy can be effectively applied. It is important to note that this strategy is feasible for a broader range of optimizers, including AdaGrad [13] and Adam [22], but we omit these in the paper as the implementations of their memory-reduced variants are relatively straightforward.

To validate our gradient reutilization approach, we conduct extensive experiments across a variety of vision models, including Vision Transformer (ViT) [12] and ConvNeXt [25], as well as leading large language models such as LLaMA-2 [38], BLOOM [23], Qwen [3], Gemma [37], ChatGLM [43], Phi [15], Falcon [2], and Vicuna [8]. The empirical results show that AdamW-R, Adan-R, and Lion-R exhibit the memory reduction up to 20.75%, 14.99%, and 25.60% at the most, respectively, which are in align with our theoretical predictions. Our contributions can be summarized as follows:

1. We propose to reuse the space of the gradients to reduce the memory footprint in the deep learning optimization phase. Based on this idea, we derive the memory reduced variants named AdamW-R, Adan-R, and Lion-R, respectively.
2. We theoretically demonstrate that our gradient space reutilization method can achieve appreciable memory savings, with AdamW-R, Adan-R, and Lion-R reducing memory usage by 20%, 14.3%, and 25%, respectively.

3. We validate our gradient reutilization method through rigorous empirical evaluations across a diverse set of vision and language models. The experimental results corroborate our theoretical findings.

2 Background and Related Work

The pursuit of even more powerful neural network models has led to an exponential increase in their complexity and the resulting demand on memory resources during training. The memory footprint of a deep learning model can be broadly categorized into four main areas: *model parameters*, *gradients*, *activations*, and *optimizer states*. The *model parameters*, during the optimization phase, are typically updated iteratively based on the *gradients* computed by backpropagation. *Activations*, the outputs of each layer given an input, are stored temporarily for usage in the backward pass. *Optimizer states*, particularly in advanced optimizers, include moment estimates and other auxiliary variables that assist in the effective optimization for each parameter.

Beyond the storage of these fundamental components, additional memory is consumed by intermediate variables, including temporary tensors that arise during the forward, backward and optimization computations, and the checkpoints needed for non-sequential models that feature complex connectivity patterns [6]. These parts further complicate the memory usage patterns and further compound to this challenge.

Methods to mitigate the memory demands in neural network training have been diversely explored. *Gradient checkpointing* (also known as *rematerialization*) reduces the memory footprint of activations by selectively storing only a strategic subset and recalculating the rest on-demand during the backward pass [6], which is particularly helpful when the intermediate activations dominate the whole memory footprint. This idea has been extended to checkpoint on Recurrent Neural Networks [14], DenseNets [31], and Transformers [19]. While it excels in scenarios where intermediate activations are the primary memory consumers, such as training smaller models with large batch sizes, its benefits are heavily mitigated in large models where the model parameters overshadows the memory occupied by activations.

Mixed-precision training leverages lower-precision floating point number formats to reduce the memory requirements of both parameters and activations. The seminal work succeeds in halving the memory requirement by downgrading into FP16 format with minimal impact on model accuracy [28], and it could be even lower, like 8-bits [41]. Up to now, Google's BF16 format is the most commonly used one in training LLMs as it takes up 16 bits and maintains a broader dynamic range [21].

As for the implementation, the *Zero Redundancy Optimizer* [33] (ZeRO) emerges as a crucial component in the DeepSpeed library [34] to combat the soaring memory demands. By partitioning the model states across the training devices, ZeRO simultaneously exploits data parallel and model parallel techniques, making it a mark of a significant advancement in the scalability of large model training.

However, despite these advancements, there has been a notable gap in the direct optimization of memory pertaining to optimizer states. Our work presents a novel approach that targets this very aspect, reducing the memory consumed by optimizer states without compromising the training dynamics or model performance.

3 Gradient Space Reutilization

In brief, this paper proposes the idea of gradient space reutilization and then applies it to AdamW [26], Adan [42], and Lion [7], to derive their memory-efficient variants named AdamW-R, Adan-R, and Lion-R, respectively, although this idea can be extended to more optimizers (like AdaGrad [13] and Adam [22]).

3.1 Core Idea

The trick of gradient space reutilization is based on a fundamental observation that the memory for storing the oldest gradient need not always be used for itself. It can be reused for temporary variables whenever the oldest gradient is no longer needed, and once the temporary variables are no longer needed, it can be switched back for storing the latest gradient to prepare for the next iteration. Generally, at the time step t, if we denote the model parameter as $\boldsymbol{\theta}_t \in \mathbb{R}^n$, the historical gradients as $\boldsymbol{G}_t = (\boldsymbol{g}_{t-1}, \cdots, \boldsymbol{g}_{t-T}) \in \mathbb{R}^{n \times T}$, and the momentum variables as $\boldsymbol{M}_t \in \mathbb{R}^{n \times N}$, where n is the model size, T is the gradient context length, and N is the number of momentum variables, then most of the first-order optimization algorithms can be represented as:

$$\boldsymbol{\theta}_{t+1} = \mathcal{F}\left(\boldsymbol{\theta}_t, \boldsymbol{G}_t, \boldsymbol{M}_t\right). \tag{1}$$

Our proposal can be concisely formulated as:

1. Reformulate the algorithm appropriately, if necessary and applicable. For example, steps involving the oldest gradient are put before the computation of the intermediate variables, so that gradient reutilization strategy is possible (see the example of Lion [7]);
2. Reuse the memory space of $\boldsymbol{g}_{t-T}$ in the optimization algorithm once it is no longer needed;
3. Move $(\boldsymbol{g}_{t-1}, \cdots, \boldsymbol{g}_{t-T}) \leftarrow (\boldsymbol{g}_t, \cdots, \boldsymbol{g}_{t-T+1})$.

For practical acceleration, the last step can be achieved by reassigning the pointers, rather than relying on the memory copy function which is more time-consuming.

We exemplify this technique on the popular AdamW [26], and the latest Adan [42] and Lion [7]. Although it is only a simple engineering trick, our extensive experiments testify to its effectiveness. Note that in deep learning, there are many simple yet effective tricks, such as batch normalization [18] and dropout [35], which eventually become indispensable.

Algorithm 1 AdamW-R

Given: momentum factors $\beta_1 = 0.9, \beta_2 = 0.999$, numerical stability number $\epsilon = 10^{-8}$, weight decay factor $\lambda > 0$, objective function f

Initialize: time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_1 \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_0 \leftarrow \mathbf{0}$, second moment vector $\boldsymbol{v}_0 \leftarrow \mathbf{0}$

while stopping criterion is not met **do**

  $t \leftarrow t + 1$

  $\boldsymbol{g}_t \leftarrow \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t)$

  $\boldsymbol{m}_t \leftarrow \beta_1 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_1) \cdot \boldsymbol{g}_t$

  $\boldsymbol{v}_t \leftarrow \beta_2 \cdot \boldsymbol{v}_{t-1} + (1 - \beta_2) \cdot \boldsymbol{g}_t^2$

  $\eta_t \leftarrow \text{Scheduler}(t)$ ▷ Stepwise learning rate handled by the scheduler

  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_t - \lambda \eta_t \cdot \boldsymbol{\theta}_t$ ▷ Decoupled weight decay

  $\eta_t \leftarrow \dfrac{\eta_t}{1 - \beta_1^t}$ ▷ Compound the bias correction factor of $\hat{\boldsymbol{m}}_t$ in AdamW

  AdamW: $\hat{\boldsymbol{m}}_t \leftarrow \dfrac{\boldsymbol{m}_t}{1 - \beta_1^t}$

  $\boldsymbol{g}_t \leftarrow \sqrt{\dfrac{\boldsymbol{v}_t}{1 - \beta_2^t}} + \epsilon$ ▷ Reuse $\boldsymbol{g}_t$ to store the denominator

  AdamW: $\hat{\boldsymbol{v}}_t \leftarrow \dfrac{\boldsymbol{v}_t}{1 - \beta_2^t}$

  $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \cdot \dfrac{\boldsymbol{m}_t}{\boldsymbol{g}_t}$

  AdamW: $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \cdot \dfrac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon}$

end while

return $\boldsymbol{\theta}_{t+1}$

3.2 AdamW-R

The Adam algorithm computes adaptive learning rates for each parameter by utilizing the estimates of first and second moments of the gradients [22]. Building upon it, AdamW introduces a modification to the weight decay regularization strategy, decoupling it from the gradient updates to improve generalization in various tasks [26]. Here, the insight of AdamW-R is to reuse the memory space of gradient to store the rectified square root of the second moment, as shown in Algorithm 1.

Through the redesign of the computations in AdamW, AdamW-R iteratively refines the model parameter while judiciously managing the memory footprint. By reusing the memory allocated for the gradient vector $\boldsymbol{g}_t$ (highlighted in red), AdamW-R reduces the demand for additional storage typically required for the second moment's square root computation. This innovative reuse of memory within the update rule not only conserves resources but also maintains the fidelity of the original AdamW's optimization trajectory. If we set the baseline algorithm to be the PyTorch implementation of AdamW [30], which stores the variables $(\boldsymbol{\theta}, \boldsymbol{g}, \boldsymbol{m}, \boldsymbol{v}, \hat{\boldsymbol{v}})$, and exclude the impact of scalar values, AdamW-R would theoretically exhibit 20% memory usage reduction, thanks to the reutilization of the space for $\boldsymbol{g}$ to store the variable $\hat{\boldsymbol{v}}$.

3.3 Adan-R

Adan introduces a new Nesterov momentum estimation method that circumvents the computational burden associated with traditional Nesterov acceleration [29], and integrates this approach into an adaptive gradient framework to expedite convergence [42]. In the case of Adan-R, the memory reutilization strategy is akin to the technique used in AdamW-R (See Algorithm 2); however, Adan-R distinctively reutilizes the previous step gradient vector $\boldsymbol{g}_{t-1}$ repeatedly within its iterative process.

Algorithm 2 Adan-R

1: **Given:** momentum factors $\beta_1 = 0.02, \beta_2 = 0.08, \beta_3 = 0.01$, numerical stability number $\epsilon = 10^{-8}$, weight decay factor $\lambda > 0$, objective function f

2: **Initialize:** time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_1 \in \mathbb{R}^n$, momentums $\boldsymbol{m}_0 \leftarrow \mathbf{0}$, $\boldsymbol{v}_0 \leftarrow \mathbf{0}$, $\boldsymbol{n}_0 \leftarrow \mathbf{0}$, previous step gradient $\boldsymbol{g}_0 \leftarrow \mathbf{0}$

3: **while** stopping criterion is not met **do**

4: $t \leftarrow t + 1$

5: $\boldsymbol{g}_t \leftarrow \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t)$

6: $\boldsymbol{m}_t \leftarrow (1 - \beta_1) \cdot \boldsymbol{m}_{t-1} + \beta_1 \cdot \boldsymbol{g}_t$

7: $\boldsymbol{g}_{t-1} \leftarrow \boldsymbol{g}_t - \boldsymbol{g}_{t-1}$ ▷ Reuse $\boldsymbol{g}_{t-1}$ to store the gradient difference

8: $\boldsymbol{v}_t \leftarrow (1 - \beta_2) \cdot \boldsymbol{v}_{t-1} + \beta_2 \cdot \boldsymbol{g}_{t-1}$

Adan: $\boldsymbol{v}_t \leftarrow (1 - \beta_2) \cdot \boldsymbol{v}_{t-1} + \beta_2 \cdot (\boldsymbol{g}_t - \boldsymbol{g}_{t-1})$

▷ No reassignment of the oldest gradient (lines 7, 9, and 12)

9: $\boldsymbol{g}_{t-1} \leftarrow \boldsymbol{g}_t + (1 - \beta_2) \cdot \boldsymbol{g}_{t-1}$

▷ Reuse $\boldsymbol{g}_{t-1}$ to store the gradient combination

10: $\boldsymbol{n}_t \leftarrow (1 - \beta_3) \cdot \boldsymbol{n}_{t-1} + \beta_3 \cdot \boldsymbol{g}_{t-1}^2$

Adan: $\boldsymbol{n}_t \leftarrow (1 - \beta_3) \cdot \boldsymbol{n}_{t-1} + \beta_3 \cdot \left[\boldsymbol{g}_t + (1 - \beta_2) \cdot (\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^2\right]$

11: $\eta_t \leftarrow \text{Scheduler}(t)$ ▷ Stepwise learning rate handled by the scheduler

12: $\boldsymbol{g}_{t-1} \leftarrow \sqrt{\boldsymbol{n}_t} + \epsilon$ ▷ Reuse $\boldsymbol{g}_{t-1}$ to store the denominator

13: $\boldsymbol{\theta}_{t+1} \leftarrow (1 - \lambda\eta_t) \cdot \boldsymbol{\theta}_t - \eta_t \cdot \dfrac{\boldsymbol{m}_t}{\boldsymbol{g}_{t-1}}$

Adan: $\boldsymbol{u}_t \leftarrow \boldsymbol{m}_t + (1 - \beta_2) \cdot \boldsymbol{v}_t$

14: $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_{t+1} - (1 - \beta_2)\eta_t \cdot \dfrac{\boldsymbol{v}_t}{\boldsymbol{g}_{t-1}}$

▷ Amortize the parameter update regarding to $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$ with two steps

Adan: $\boldsymbol{\theta}_{t+1} \leftarrow (1 - \lambda\eta_t) \cdot \boldsymbol{\theta}_t - \eta_t \cdot \dfrac{\boldsymbol{u}_t}{\sqrt{\boldsymbol{n}_t} + \epsilon}$

15: $\boldsymbol{g}_{t-1} \leftarrow \boldsymbol{g}_t$ ▷ Move by reassigning the pointers

16: **end while**

17: **return** $\boldsymbol{\theta}_{t+1}$

Through this innovative approach, Adan-R significantly reduces the algorithm's memory requirements without altering the fundamental computational logic of the original Adan algorithm. This is achieved by repurposing the space allocated for $\boldsymbol{g}_{t-1}$, which is reused three times within a single update iteration– the gradient difference, the gradient combination, and the square root of the $\boldsymbol{n}_t$ vector. Consequently, if we consider the baseline PyTorch implementation of Adan, which stores the variables $\left(\boldsymbol{g}_t, \boldsymbol{g}_{t-1}, \boldsymbol{m}, \boldsymbol{v}, \boldsymbol{n}, \boldsymbol{\theta}, \sqrt{\boldsymbol{n}}\right)$, Adan-R eliminates the need for additional memory that would typically be allocated for $\sqrt{\boldsymbol{n}}$, yielding $1/7 = 14.3\%$ memory saving in theory.

3.4 Lion-R

The Lion optimizer is discovered by an evolutionary strategy based program search, seminally showing the feasibility and generalizability of incorporating sign function into optimization algorithms [7]. It is relatively memory-efficient among the mainstream optimizers because it only keeps track of the first order momentum without maintaining a separate adaptive learning rate for each parameter. Stepping further, Lion-R reveals the potentials of being more memory-efficient with a non-trivial reformulation on the original algorithm.

Algorithm 3 Lion-R

1: **Given:** momentum factors $\beta_1 = 0.9, \beta_2 = 0.99$, weight decay factor $\lambda > 0$, objective function f
2: **Initialize:** time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_1 \in \mathbb{R}^n$, momentum $\boldsymbol{m}_0 \leftarrow \mathbf{0}$
3: **while** stopping criterion is not met **do**
4: $t \leftarrow t + 1$
5: $\boldsymbol{g}_t \leftarrow \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t)$
6: $\boldsymbol{m}_t \leftarrow \beta_2 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_2) \cdot \boldsymbol{g}_t$
Lion: $\boldsymbol{c}_t \leftarrow \beta_1 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_1) \cdot \boldsymbol{g}_t$
7: $\boldsymbol{g}_t \leftarrow \frac{\beta_1}{\beta_2} \cdot \boldsymbol{m}_t + \left(1 - \frac{\beta_1}{\beta_2}\right) \cdot \boldsymbol{g}_t$
▷ Reformulate Lion optimizer and reuse $\boldsymbol{g}_t$ to store the original $\boldsymbol{c}_t$
Lion: $\boldsymbol{m}_t \leftarrow \beta_2 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_2) \cdot \boldsymbol{g}_t$
8: $\eta_t \leftarrow \text{Scheduler}(t)$ ▷ Stepwise learning rate handled by the scheduler
9: $\boldsymbol{\theta}_{t+1} \leftarrow (1 - \lambda\eta_t) \cdot \boldsymbol{\theta}_t - \eta_t \cdot \text{sign}(\boldsymbol{g}_t)$
Lion: $\boldsymbol{\theta}_{t+1} \leftarrow (1 - \lambda\eta_t) \cdot \boldsymbol{\theta}_t - \eta_t \cdot \text{sign}(\boldsymbol{c}_t)$
10: **end while**
11: **return** $\boldsymbol{\theta}_{t+1}$

The original Lion algorithm mainly differs from Lion-R at lines 6 and 7, which is shown in Algorithm 3. By altering the sequence of lines 6 and 7 in the original algorithm, we obtain:

$$\begin{cases} \boldsymbol{m}_t \leftarrow \beta_2 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_2) \cdot \boldsymbol{g}_t, \\ \boldsymbol{c}_t \leftarrow \beta_1 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_1) \cdot \boldsymbol{g}_t. \end{cases} \tag{2}$$

By substituting $\boldsymbol{m}^{t-1} = \frac{1}{\beta_2}(\boldsymbol{m}^t - (1-\beta_2)\boldsymbol{g}^t)$ into the expression of $\boldsymbol{c}^t$, the update rule can be reformulated as:

$$\begin{cases} \boldsymbol{m}_t \leftarrow \beta_2 \cdot \boldsymbol{m}_{t-1} + (1-\beta_2) \cdot \boldsymbol{g}_t, \\ \boldsymbol{c}_t \leftarrow \dfrac{\beta_1}{\beta_2} \cdot \boldsymbol{m}_t + \left(1 - \dfrac{\beta_1}{\beta_2}\right) \cdot \boldsymbol{g}_t. \end{cases} \tag{3}$$

As such, the intermediate variable $\boldsymbol{c}_t$ is readily to be applied to our gradient reutilization strategy. As is shown in line 7, Lion-R reutilizes $\boldsymbol{g}_t$ to store this expression thus eliminates the need of storing the extra variable $\boldsymbol{c}_t$. Compared to Lion which requires storing the data $(\boldsymbol{\theta}, \boldsymbol{g}, \boldsymbol{m}, \boldsymbol{c})$, Lion-R would theoretically reduce the memory requirement by around 25%.

4 Experiments

To assess the efficacy of our proposed gradient reutilization methods across various architectures, we conduct a comprehensive suite of experiments on both vision and language models with various model sizes and scales.

For vision models, we perform experiments on the representative ViTs [12] and ConvNeXts [25], choosing to test multiple model sizes within each architecture. For language tasks, our experiments cover most of the popular large models, including LLaMA-2 [38], BLOOM [23], Qwen [3], Gemma [37], ChatGLM [43], Phi [15], Falcon [2], and Vicuna [8], and test on different model sizes when available. Since the baseline optimizers are proven to be valid and effective on all kinds of tasks, the metric of interest in this paper should be peak memory usage statistics, as our memory reduced variants are exactly equivalent to their original ones.

To illustrate the performance equivalence between the original and memory-reduced optimizers, we conduct specific training sessions for both the AdamW and Adan optimizers along with their respective memory-reduced variants, with the ViT-S and ViT-B models on ImageNet [10] dataset. The results in Table 1 confirm that both pairs of original and memory-reduced optimizers yields exactly the same top-1 accuracies.

Optimizer	**ViT-S**			**ViT-B**		
	Top-1 Acc. (%)		**Memory (MB)**	**Top-1 Acc. (%)**		**Memory (MB)**
	150	300		150	300	
AdamW	78.3	79.9	526	79.5	81.8	2007
AdamW-R	78.3	79.9	**417**	79.5	81.8	**1629**
Adan	79.6	80.9	711	81.7	82.6	2806
Adan-R	79.6	80.9	**621**	81.7	82.6	**2407**

Table 1. Comparison of ImageNet Top-1 accuracies and memory usage for original and memory-reduced variants of AdamW and Adan across 150 and 300 epochs on ViT-S and ViT-B models. The accuracies for AdamW and Adan are cited from [42].

As mentioned in Section 1, while small models work well, it is unrealistic to directly feed LLMs into the GPUs. One of the common solutions is to apply the ZeRO strategy [33] for enjoying the benefit of distributed training. Unlike traditional optimizers, ZeRO does not alter the underlying mathematical logic of optimization; rather, it is a technological solution designed to optimize memory usage and scale training across distributed environments efficiently. ZeRO achieves this through three progressive stages, in which the State 3 is the heaviest memory efficient setting that partitions parameters, gradients, and optimizer states across all GPUs. To maximize the utilization of these advantages and to mimic an environment where memory is limited, we adopt this configuration in our LLM experiments.

We report the training settings in detail for reproducibility. For ViTs, we adopt five architectures, ViT-Small, ViT-Base, ViT-large, ViT-Huge, and ViT-Giant, with the input RGB image being 224×224 pixels and split into 32×32 pixel patches. The ConvNeXt has five scales, ConvNeXt-Tiny, ConvNeXt-Small, ConvNeXt-Base, ConvNeXt-Large, and ConvNeXt-XL, whose image inputs are of the same size. We train them with 5 iterations and measure the peak memory usage. For language models, we utilize a truncated version of the Alpaca dataset [36] containing 2,000 examples, configure the context length to 1,024 tokens and limit the training to 3 epochs. We set batch size to 1 throughout the experiments. This abbreviated training section is justified by our observations that the memory footprint remains consistent across all handling processes (forward pass + backward pass + parameter optimization) for a single minibatch.

We measure the peak memory usage of AdamW, Adan, and Lion as well as their memory reduced variants within the PyTorch framework, reporting the number of model parameters, the percentage of memory saved and the usage of ZeRO strategy when applicable. The results are summarized in Tables 2 to 4. All of the experiments are carried out on $8 \times$ NVIDIA RTX A6000 GPUs.

The experimental results demonstrate that our proposed memory-reduced optimizers, AdamW-R, Adan-R, and Lion-R, generally achieve close to the theoretical predictions of memory savings, particularly in vision-related tasks. In vision scenarios such as with ViT and ConvNeXt models, the savings are significant, aligning well with our theoretical expectations.

However, the results across LLMs present a more nuanced picture, especially with the integration of ZeRO strategy. In such cases, while we still observe notable memory savings, the overall impact of our optimizers is somewhat tempered. This could be attributed to ZeRO's sophisticated mechanisms, which might dilute the relative contribution of our memory optimization strategies. Moreover, in some settings, the peak memory consumption does not occur during the optimization phase but rather during the forward or backward phases, leading to minimal observed savings from our optimizers. Despite the relatively smaller percentage reductions in memory usage for LLMs, the actual financial savings can still be substantial due to the extremely high costs associated with training such large models. Even a modest percentage reduction in memory usage still translates into significant financial cost reductions.

Model		# Params	AdamW (MB)	AdamW-R (MB)	Savings (%)	ZeRO
ViT	ViT-S	22.9M	526	417	**20.71**	✗
	ViT-B	88.2M	2007	1629	**18.81**	✗
	ViT-L	305.5M	6367	5046	**20.75**	✗
	ViT-H	630.8M	13336	10777	**19.19**	✗
	ViT-G	1.0B	21542	17408	**19.19**	✗
ConvNeXt	ConvNeXt-T	28.6M	684	621	**9.20**	✗
	ConvNeXt-S	50.2M	1177	1009	**14.26**	✗
	ConvNeXt-B	88.6M	1894	1629	**13.95**	✗
	ConvNeXt-L	197.8M	4387	3706	**15.54**	✗
	ConvNeXt-XL	350.2M	7218	6004	**16.82**	✗
BLOOM	BLOOM-560M	559.2M	15531	13822	**11.00**	✗
	BLOOM-560M	559.2M	5339	5011	**6.15**	✓
	BLOOM-3B	3.0B	23477	21964	**6.45**	✓
	BLOOM-7B	7.1B	44826	41296	**7.87**	✓
Phi	Phi-1.5	1.4B	36650	36008	1.75	✗
	Phi-1.5	1.4B	18616	17949	**3.59**	✓
	Phi-2	2.8B	27581	26132	**5.26**	✓
Qwen	Qwen-0.5B	464.0M	12581	11272	**10.40**	✗
	Qwen-0.5B	464.0M	4897	4837	1.23	✓
	Qwen-1.8B	1.8B	46410	38986	**16.00**	✗
	Qwen-1.8B	1.8B	12756	11902	**6.69**	✓
LLaMA-2	LLaMA-2-7B	6.7B	32325	29002	**10.28**	✓
	LLaMA-2-13B	13.0B	49103	45768	**6.79**	✓
Gemma	Gemma-2B	2.5B	19609	18365	**6.35**	✓
	Gemma-7B	8.5B	47029	42841	**8.90**	✓
Vicuna	Vicuna-7B	6.7B	32351	28993	**10.38**	✓
	Vicuna-13B	13.0B	49327	46089	**6.57**	✓
ChatGLM3	ChatGLM3-6B	6.2B	31491	28369	**9.92**	✓
Falcon	Falcon-7B	6.9B	33643	30168	**10.33**	✓

Table 2. Peak memory usage of AdamW and AdamW-R on vision and language models. The ✓ sign means that ZeRO is utilized.

5 Conclusion

In this paper, we propose the idea of gradient space reutilization for deep learning optimizers, and point out that we can reuse the oldest gradient space once it is no longer needed for later computations. We successfully apply this idea and derive three memory efficient optimizers, namely AdamW-R, Adan-R, and Lion-R, though this strategy can be extended. Our theoretical and experimental

Model		# Params	Adan (MB)	Adan-R (MB)	Savings (%)	ZeRO
ViT	ViT-S	22.9M	711	621	**12.68**	✗
	ViT-B	88.2M	2806	2407	**14.20**	✗
	ViT-L	305.5M	8812	7491	**14.99**	✗
	ViT-H	630.8M	18639	16110	**13.57**	✗
	ViT-G	1.0B	30130	25910	**14.00**	✗
ConvNeXt	ConvNeXt-T	28.6M	927	864	**6.78**	✗
	ConvNeXt-S	50.2M	1634	1466	**10.27**	✗
	ConvNeXt-B	88.6M	2632	2355	**10.52**	✗
	ConvNeXt-L	197.8M	6078	5417	**10.87**	✗
	ConvNeXt-XL	350.2M	10008	8823	**11.84**	✗
BLOOM	BLOOM-560M	559.2M	20005	18296	**8.55**	✗
	BLOOM-560M	559.2M	5859	5544	**5.38**	✓
	BLOOM-3B	3.0B	26472	24965	**5.69**	✓
	BLOOM-7B	7.1B	48355	48184	0.35	✓
Phi	Phi-1.5	1.4B	20098	19370	**3.62**	✓
	Phi-2	2.8B	30301	28907	**4.59**	✓
Qwen	Qwen-0.5B	464.0M	16437	15129	**7.96**	✗
	Qwen-0.5B	464.0M	5509	5491	0.33	✓
	Qwen-1.8B	1.8B	14691	13673	**6.93**	✓
LLaMA-2	LLaMA-2-7B	6.7B	39115	35713	**8.70**	✓
Gemma	Gemma-2B	2.5B	22118	20870	**5.64**	✓
	Gemma-7B	8.5B	49424	48484	1.91	✓
Vicuna	Vicuna-7B	6.7B	32351	28993	**10.38**	✓
ChatGLM3	ChatGLM3-6B	6.2B	37670	34614	**8.11**	✓
Falcon	Falcon-7B	6.9B	40548	37099	**8.51**	✓

Table 3. Peak memory usage of Adan and Adan-R on vision and language models. The ✓sign means that ZeRO is utilized.

analyses demonstrate that these techniques can lead to appreciable memory cost savings without compromising the optimization performance of the models.

The empirical results affirm that our proposed methods could achieve close to theoretical memory reduction across various architectures and scales. Even with the employment of ZeRO strategy, these optimizer variants still possess the memory efficient feature, indicating that our method could be helpful for large scale deep learning practitioners.

Acknowledgment

Zhouchen Lin was supported by National Key R&D Program of China (2022ZD0160300), the NSF China (No. 62276004), and Qualcomm.

Model		# Params	Lion (MB)	Lion-R (MB)	Savings (%)	ZeRO
ViT	ViT-S	22.9M	415	327	**21.21**	✗
	ViT-B	88.2M	1629	1231	**24.45**	✗
	ViT-L	305.5M	5144	3827	**25.60**	✗
	ViT-H	630.8M	10687	8087	**24.33**	✗
	ViT-G	1.0B	17226	13189	**23.43**	✗
ConvNeXt	ConvNeXt-T	28.6M	552	489	**11.41**	✗
	ConvNeXt-S	50.2M	958	791	**17.51**	✗
	ConvNeXt-B	88.6M	1529	1281	**16.19**	✗
	ConvNeXt-L	197.8M	3521	2861	**18.77**	✗
	ConvNeXt-XL	350.2M	5862	4618	**21.22**	✗
BLOOM	BLOOM-560M	559.2M	13294	11996	**9.76**	✗
	BLOOM-560M	559.2M	4513	4508	0.12	✓
	BLOOM-3B	3.0B	21957	20462	**6.81**	✓
	BLOOM-7B	7.1B	41306	37761	**8.58**	✓
Phi	Phi-1.5	1.4B	17950	17273	**3.77**	✓
	Phi-2	2.8B	26159	24809	**5.15**	✓
Qwen	Qwen-0.5B	464.0M	10614	9666	**8.93**	✗
	Qwen-0.5B	464.0M	4897	4855	0.86	✓
	Qwen-1.8B	1.8B	38986	31562	**19.04**	✗
	Qwen-1.8B	1.8B	11913	10945	**8.13**	✓
LLaMA-2	LLaMA-2-7B	6.7B	29007	25618	**11.68**	✓
	LLaMA-2-13B	13.0B	47297	39249	**17.02**	✓
Gemma	Gemma-2B	2.5B	18347	17123	**6.67**	✓
	Gemma-7B	8.5B	48279	39416	**8.08**	✓
Vicuna	Vicuna-7B	6.7B	28978	25596	**11.67**	✓
	Vicuna-13B	13.0B	47596	39514	**16.98**	✓
ChatGLM3	ChatGLM3-6B	6.2B	28302	25180	**11.03**	✓
Falcon	Falcon-7B	6.9B	30187	26719	**11.49**	✓

Table 4. Peak memory usage of Lion and Lion-R on vision and language models. The ✓sign means that ZeRO is utilized.

References

1. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., et al.: GPT-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
2. Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., et al.: The Falcon series of open language models. arXiv preprint arXiv:2311.16867 (2023)
3. Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., et al.: Qwen technical report. arXiv preprint arXiv:2309.16609 (2023)
4. Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., et al.: End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316 (2016)

5. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., et al.: Language models are few-shot learners. Advances in Neural Information Processing Systems **33**, 1877–1901 (2020)
6. Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174 (2016)
7. Chen, X., Liang, C., Huang, D., Real, E., Wang, K., et al.: Symbolic discovery of optimization algorithms. Advances in Neural Information Processing Systems **36** (2024)
8. Chiang, W.L., Li, Z., Lin, Z., Sheng, Y., Wu, Z., Zhang, H., Zheng, L., Zhuang, S., Zhuang, Y., Gonzalez, J.E., Stoica, I., Xing, E.P.: Vicuna: An open-source chatbot impressing GPT-4 with 90%* ChatGPT quality (March 2023), `https://lmsys.org/blog/2023-03-30-vicuna/`
9. Choudhary, K., DeCost, B., Chen, C., Jain, A., Tavazza, F., et al.: Recent advances and applications of deep learning methods in materials science. NPJ Computational Materials **8**(1), 59 (2022)
10. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. pp. 248–255. Ieee (2009)
11. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
12. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., et al.: An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929 (2020)
13. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research **12**(7) (2011)
14. Gruslys, A., Munos, R., Danihelka, I., Lanctot, M., Graves, A.: Memory-efficient backpropagation through time. Advances in Neural Information Processing Systems **29** (2016)
15. Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C.C.T., Del Giorno, A., et al.: Textbooks are all you need. arXiv preprint arXiv:2306.11644 (2023)
16. He, L., Chen, Y., Dong, Y., Wang, Y., Lin, Z., et al.: Efficient equivariant network. Advances in Neural Information Processing Systems **34**, 5290–5302 (2021)
17. He, L., Dong, Y., Wang, Y., Tao, D., Lin, Z.: Gauge equivariant transformer. Advances in Neural Information Processing Systems **34**, 27331–27343 (2021)
18. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: International Conference on Machine Learning. pp. 448–456. PMLR (2015)
19. Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., Stoica, I.: Checkmate: Breaking the memory wall with optimal tensor rematerialization. Proceedings of Machine Learning and Systems **2**, 497–511 (2020)
20. Kaddour, J., Harris, J., Mozes, M., Bradley, H., Raileanu, R., McHardy, R.: Challenges and applications of large language models. arXiv preprint arXiv:2307.10169 (2023)
21. Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., et al.: A study of BFLOAT16 for deep learning training. arXiv preprint arXiv:1905.12322 (2019)
22. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
23. Le Scao, T., Fan, A., Akiki, C., Pavlick, E., Ilić, S., et al.: BLOOM: A 176B-parameter open-access multilingual language model (2022)
24. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)

25. Liu, Z., Mao, H., Wu, C.Y., Feichtenhofer, C., Darrell, T., Xie, S.: A ConvNet for the 2020s. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 11976–11986 (2022)
26. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101 (2017)
27. Mehta, S., Shah, D., Kulkarni, R., Caragea, C.: Semantic tokenizer for enhanced natural language processing. arXiv preprint arXiv:2304.12404 (2023)
28. Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., et al.: Mixed precision training. arXiv preprint arXiv:1710.03740 (2017)
29. Nesterov, Y.: A method of solving a convex programming problem with convergence rate O(1/k2). Doklady Akademii Nauk SSSR **269**(3), 543 (1983)
30. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in PyTorch. In: NIPS 2017 Workshop on Autodiff (2017)
31. Pleiss, G., Chen, D., Huang, G., Li, T., Van Der Maaten, L., Weinberger, K.Q.: Memory-efficient implementation of DenseNets. arXiv preprint arXiv:1707.06990 (2017)
32. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., et al.: Language models are unsupervised multitask learners. OpenAI Blog **1**(8), 9 (2019)
33. Rajbhandari, S., Rasley, J., Ruwase, O., He, Y.: ZeRO: Memory optimizations toward training trillion parameter models. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–16 (2020)
34. Rasley, J., Rajbhandari, S., Ruwase, O., He, Y.: DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 3505–3506 (2020)
35. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. Journal of Machine Learning Research **15**(1), 1929–1958 (2014)
36. Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., Hashimoto, T.B.: Stanford Alpaca: An instruction-following LLaMA model (2023)
37. Team, G., Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., et al.: Gemma: Open models based on Gemini research and technology. arXiv preprint arXiv:2403.08295 (2024)
38. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., et al.: LLaMA 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023)
39. Tropsha, A., Isayev, O., Varnek, A., Schneider, G., Cherkasov, A.: Integrating QSAR modelling and deep learning in drug discovery: the emergence of deep QSAR. Nature Reviews Drug Discovery **23**(2), 141–155 (2024)
40. Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S.L., Xu, Z., Kraska, T.: Superneurons: Dynamic GPU memory management for training deep neural networks. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 41–53 (2018)
41. Wang, N., Choi, J., Brand, D., Chen, C.Y., Gopalakrishnan, K.: Training deep neural networks with 8-bit floating point numbers. Advances in Neural Information Processing Systems **31** (2018)
42. Xie, X., Zhou, P., Li, H., Lin, Z., Yan, S.: Adan: Adaptive nesterov momentum algorithm for faster optimizing deep models. arXiv preprint arXiv:2208.06677 (2022)
43. Zeng, A., Liu, X., Du, Z., Wang, Z., Lai, H., et al.: GLM-130B: An open bilingual pre-trained model. arXiv preprint arXiv:2210.02414 (2022)