FLORA: Low-Rank Adapters Are Secretly Gradient Compressors

Yongchang Hao [*1] **Yanshuai Cao** [2] **Lili Mou** [1 3]

Abstract

Despite large neural networks demonstrating remarkable abilities to complete different tasks, they require excessive memory usage to store the optimization states for training. To alleviate this, the low-rank adaptation (LoRA) is proposed to reduce the optimization states by training fewer parameters. However, LoRA restricts overall weight update matrices to be low-rank, limiting the model performance. In this work, we investigate the dynamics of LoRA and identify that it can be approximated by a random projection. Based on this observation, we propose FLORA, which is able to achieve high-rank updates by resampling the projection matrices while enjoying the sublinear space complexity of optimization states. We conduct experiments across different tasks and model architectures to verify the effectiveness of our approach.

1. Introduction

Gradient-based optimization powers the learning part of deep neural networks. In the simplest form, stochastic gradient descent (SGD) updates the model parameters using noisy estimation of the negative gradient. More advanced methods track various gradient statistics to stabilize and accelerate training (Duchi et al., 2011; Hinton et al., 2012). For example, the momentum technique tracks an exponential moving average of gradients for variance reduction (Cutkosky & Orabona, 2019) and damping (Goh, 2017). On the other hand, gradient accumulation computes the average of gradients in the last few batches to simulate a larger effective batch for variance reduction (Wang et al., 2013). Both cases require an additional memory buffer equal to the model size to store the information.

However, such a linear space complexity of optimization states becomes problematic in modern deep learning. For example, GPT-3 (Brown et al., 2020) and Stable Diffusion (Rombach et al., 2022) are trained with Adam (Kingma & Ba, 2015) where momentum is applied. For each scalar in the parameter set, Adam maintains two additional variables (i.e., first- and second-moment estimates), tripling the memory usage. The largest GPT-3, for example, has 175 billion parameters taking 700GB of memory. Adam requires an additional 1.4TB memory for optimization states. This excessive amount of memory usage poses a scaling challenge.

One line of research saves memory by training a subset of parameters (Houlsby et al., 2019; Zaken et al., 2022), so the optimizer only stores information about a small set of trainable parameters. One notable example is the low-rank adaptation (LoRA, Hu et al., 2022). LoRA updates parameter matrices by low-rank patches, which contain much fewer trainable parameters. In this way, the momentum and gradient accumulation also have much smaller sizes. However, LoRA restricts the weight update to be in the low-rank form, limiting the optimization space of the model parameters.

Another line of work designs new optimizers that use less memory (Dettmers et al., 2021; Feinberg et al., 2023). For instance, Adafactor (Shazeer & Stern, 2018) leverages the closed-form solution of generalized Kullback–Leibler divergence (Finesso & Spreij, 2006) to reconstruct the second-moment estimate in Adam. To optimize a matrix in $\mathbb{R}^{n\times m}$, Adafactor reduces the memory from $O(nm)$ to $O(n+m)$, making the space complexity of second-moment estimation sublinear in model size. However, Adafactor drops the momentum technique to achieve the sublinearity, sacrificing the variance reduction and damping effect of momentum (Rae et al., 2021). Moreover, it does not reduce the memory for gradient accumulation.

In this work, we propose FLORA (**f**rom **L**oRA t**o** high-**r**ank upd**a**tes), which is a novel optimization technique that uses sublinear memory for gradient accumulation and momentum calculation. Our intuition arises from investigating LoRA and observing that a LoRA update is dominated by

[*]Project done during Mitacs internship at Borealis AI. [1]Department of Computing Science & Alberta Machine Intelligence Institute (Amii), University of Alberta [2]Borealis AI [3]Canada CIFAR AI Chair. Correspondence to: Yongchang Hao <yongcha1@ualberta.ca>, Yanshuai Cao <yanshuai.cao@borealisai.com>, Lili Mou <doublepower.mou@gmail.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

a random projection, which compresses the gradient into a lower-dimensional space (Dasgupta, 2000; Bingham & Mannila, 2001). Thus, we propose FLORA that applies such a compression technique directly to the update of the original weight matrix. Our FLORA resamples the random projection and is able to mitigate the low-rank limitation of LoRA. Further, our approach only stores the compressed gradient accumulation and momentum, thus saving the memory usage of optimization states to the sublinear level. We conduct experiments across different tasks and model architectures to verify the effectiveness of our approach. When combined with Adafactor as a base optimizer, our approach yields similar performance to uncompressed, full-matrix update, while largely outperforming other compression techniques such as LoRA. Interestingly, the space complexity of FLORA is in the same order as LoRA but has a smaller constant in practice, leading to less memory usage than LoRA.

2. Approach

In this section, we first present our observation of the dynamics of LoRA updates (§2.1). Then, we show that LoRA can be approximated by random projection (§2.2), which serves as gradient compression (§2.3) and can be used for sublinear-space gradient accumulation and momentum calculation (§2.4).

2.1. Dynamics of low-rank adaptation (LoRA)

For update a pre-trained weight matrix $W \in \mathbb{R}^{n \times m}$, LoRA parameterizes $B \in \mathbb{R}^{n \times r}$ and $A \in \mathbb{R}^{r \times m}$ with $r \ll \min\{n, m\}$. After applying LoRA, the forward pass becomes

$$y = (W + BA)x = Wx + BAx, \tag{1}$$

where $x \in \mathbb{R}^m$ is the input for current layer and $y \in \mathbb{R}^n$ is the pre-activation value of the next layer. At the beginning of LoRA updates, BA should not change the original weight W. The common practice is to initialize the matrix B with an all-zero matrix and A with a normal distribution.

During back-propagation, the matrix W has gradient

$$\nabla_W L = \frac{\partial L}{\partial y} x^\top, \tag{2}$$

where $\frac{\partial L}{\partial y} \in \mathbb{R}^n$ is the partial derivative w.r.t. y. LoRA only calculates the gradient w.r.t. the matrices A and B, given by

$$\frac{\partial L}{\partial A} = B^\top \frac{\partial L}{\partial y} x^\top = B^\top (\nabla_W L) \tag{3}$$

$$\text{and} \quad \frac{\partial L}{\partial B} = \frac{\partial L}{\partial y} x^\top A^\top = (\nabla_W L) A^\top. \tag{4}$$

Our insight is that in Equations (3) and (4), LoRA essentially down-projects the original gradient to a lower dimension. In fact, we found that LoRA recovers the well-known random projection method (Dasgupta, 2000; Bingham & Mannila, 2001). We formally state this in the following theorem.

Theorem 2.1. Let LoRA update matrices A and B with SGD for every step t by

$$A_{t+1} \quad A_t \quad B_t^\top (\nabla_W L_t) \tag{5}$$

$$B_{t+1} \quad B_t \quad (\nabla_W L_t) A_t^\top; \tag{6}$$

where $$ is the learning rate. We assume $\sum_{t=0}^{T} \nabla_W L_t \_F \quad L$ for every T during training, which implies that the model stays within a finite Euclidean ball. In this case, the dynamics of A_t and B_t are given by

$$A_T = A_0 + A_0 f_A(T); \quad B_T = f_B(T) A_0^\top; \tag{7}$$

where the forms of $f_A(t) \in \mathbb{R}^{m \times m}$ and $f_B(t) \in \mathbb{R}^{n \times m}$ are expressed in the proof. In particular, $\|f_A(t)\|_2 \le \frac{L^2 \, 1 \, (\,^2 L^2)^t}{1 \,^2 L^2}$ for every t.

Proof. See Appendix A. □

Theorem 2.1 describes the SGD dynamics of LoRA updates. Without loss of generality, we denote the total change of A and B after T step as ΔA and ΔB, respectively. Then the fine-tuned forward function will be

$$W + (B_0 + \Delta B)(A_0 + \Delta A) \tag{8}$$

$$= W + B_0 A_0 + B_0 \Delta A + \Delta B A_0 + \Delta B \Delta A \tag{9}$$

$$= W + \Delta B A_0 + \Delta B \Delta A; \tag{10}$$

where $B_0 = 0$ is due to the initialization of the B matrix. The final expression dissects the LoRA weight into two parts. We observe that it is the first part that dominates the total weight change, as stated below.

Observation 2.2. When the learning rate is small, we have an approximation that

$$W + (B_0 + \Delta B)(A_0 + \Delta A) \quad W + \Delta B A_0. \tag{11}$$

This can be seen by expanding ΔB and A_0 given by Theorem 2.1. Specifically, we have that

$$W + \Delta B A_0 + \Delta B \Delta A \tag{12}$$

$$= W + f_B(t) A_0^\top A_0 + \,^2 f_B(t) A_0^\top A_0 f_A(t) \tag{13}$$

Our insight is that the third term has a smaller magnitude when the learning rate is not large. This is because $\|f_A(t)\|_2 \quad \|f_A(t)\|_F \quad \frac{L^2 \, 1 \, (\,^2 L^2)^t}{1 \,^2 L^2}$ given by Theorem 2.1. If $\quad 1/L$, we have $\lim_{t \to 1} \quad \|f_A(t)\| \quad 1$, which indicates that the third term is significantly smaller than the second term, making it negligible in the final update.

2.2. Random projection of gradients

By Observation 2.2, the change of the matrix B dominates the final weight. A straightforward simplification is to freeze the matrix A but to tune the matrix B only (denoted by $B^{\sim}$). In this case, we have

$$W + (B_0 + \Delta B)(A_0 + \Delta A) \tag{14}$$
$$W + \Delta B A_0 \tag{15}$$
$$=: W + \tilde{f}_B(T) A_0^\top A_0. \tag{16}$$

In Equation (15), B_0 is dropped because B is initialized as an all-zero matrix. Equation (16) defines $\tilde{f}_B(T)$, which will have the update form

$$\tilde{f}_B(t+1) := \tilde{f}_B(t) - \eta \nabla_W L_t \tag{17}$$

following the derivations in Theorem 2.1. Therefore, $\tilde{f}_B(t) = \sum_i -\eta \nabla_W L_i$. Putting it to Equation (16), we have

$$W + \tilde{f}_B(T) A_0^\top A_0 \tag{18}$$
$$= W + \left(\sum_{t=0}^{T} -\eta \nabla_W L_t\right) A_0^\top A_0 \tag{19}$$
$$= W + \sum_{t=0}^{T} \left[(-\eta \nabla_W L_t) A_0^\top A_0\right]. \tag{20}$$

In other words, our derivation reveals that, with some approximations, LoRA updates can be viewed as performing random projection to the gradient. In particular, it compresses a gradient by a random down-projection $A_0^\top$, and then decompresses it by an up-projection A_0.

2.3. Our interpretation of LoRA

To this end, we provide a novel interpretation of a LoRA update by framing it as the compression and decompression of gradients.

Compression. LoRA first compresses the gradient by a random down-projection, which can be justified by the following result based on the Johnson–Lindenstrauss lemma (Dasgupta & Gupta, 2003; Matoušek, 2008).

Lemma 2.3 (Indyk & Motwani). Let $\varepsilon \in (0, 1/2]$ and $\delta \in (0, 1)$. Let $A \in \mathbb{R}^{r \times m}$ be a random matrix where each element is independently sampled from a standard Gaussian distribution. There exists a constant c such that when $r = c\,\varepsilon^{-2} \log(1/\delta)$, we have

$$(1-\varepsilon)\|x\| \le (1/\sqrt{r})\|Ax\| \le (1+\varepsilon)\|x\| \tag{21}$$

with probability at least $1-\delta$ for every $x \in \mathbb{R}^m$.

Essentially, this lemma suggests that, with a high probability, the projection by a random Gaussian matrix largely preserves the scaled norm in the original space.

Figure 1: The results of LoRA and its simplifications. We apply the LoRA patch to the first layer of the network with a shape of 768×768 and set $r = 8$. The legend "LoRA" is the original LoRA method, while "LoRA (B)" is the simplification where only the matrix B is updated. "RP" (random projection) and "RRP" (resampled RP) follow the same update rule (20), but "RRP" uses different projection matrices at different steps. In addition, we show the results of SGD on the full model for comparison. All experiments use the same $\eta = 0.01$.

In the case of LoRA, such a random projection is applied to each row of the gradient matrices, whose dimension is thus reduced from $\mathbb{R}^{n \times m}$ to $\mathbb{R}^{r \times m}$. The lemma asserts that the norm structure of the rows is approximately preserved.

Decompression. After down-projection by $A_0^\top$, LoRA decomposes the gradient by an up-projection A_0. We show that this will recover the original gradient in expectation:

$$\mathop{\mathbb{E}}_{A_0}\left[W + (-\eta \nabla_W L_t) A_0^\top A_0\right] \tag{22}$$
$$= W + (-\eta \nabla_W L_t) \mathop{\mathbb{E}}_{A_0}[A_0^\top A_0] \tag{23}$$

where $(1/r)\,\mathbb{E}_{A_0}[A_0^\top A_0]$ is an identity matrix. Moreover, the larger the rank r, the closer the expectation is to the identity. We quantify the error in our following theorem.

Theorem 2.4. Let A be a matrix of shape $\mathbb{R}^{r \times m}$ where each element is independently sampled from a standard Gaussian distribution. Let $\varepsilon, \delta \in (0, 1]$. There exists a constant c such that when $r = c \log(2m/\delta)\,\varepsilon^{-2}$, we have for all i, j that

$$[A^\top A - I]_{i,j} \tag{24}$$

with confidence at least $1-\delta$.

Proof. See Appendix B. □

Our Theorem 2.4 implies that A only needs to scale logarithmically to preserve the element-wise reconstruction error, which is efficient in both computation and memory. Further, the logarithmic asymptotic rate makes it an ideal candidate to be applied to the training of modern neural models where m is large.

We empirically verify our interpretation of LoRA by a pilot study on the Fashion-MNIST dataset (Xiao et al., 2017) with a simple feed-forward network. We experiment with a variant of LoRA where only B is tuned; we call the variant LoRA(B). As shown in Figure 1, the performance of LoRA(B) is close to the original LoRA, which is consistent with Observation 2.2 and suggests the overall update of LoRA is dominated by the compression-and-decompression step. Further, the curve is identical to random projection (RP), well aligned with our derivation in Section 2.2.

2.4. Our method: FLORA

Based on the analyses, we propose our method, FLORA (from LoRA to high-rank updates), to enable overall high-rank updates. One of the main insights of FLORA is that it constantly resamples the projection matrix in Equation (20). Therefore, our total weight change is no longer constrained to be low-rank. Moreover, we can apply the random-projection compression to the optimization states for memory saving. We demonstrate two common scenarios where FLORA can be applied: (1) an arithmetic mean (AM) over a period of history, for which a concrete example is gradient accumulation; (2) an exponential moving average (EMA), for which an example could be momentum calculation. We show that compression in FLORA has the same asymptotic rate as LoRA but with a lower constant.

Resampling random projection matrices. With the approximation in Observation 2.2, LoRA can be viewed as having a fixed random projection matrix A_0. This restricts the overall change of W to be low-rank. However, our analysis in Section 2.3 holds for any random matrix at every time step.

Therefore, we propose in FLORA to resample a new random matrix to avoid the total change restricted in a low-rank subspace. In our pilot study, resampling the random matrix (RRP) largely recovers the performance of full-matrix SGD, significantly surpassing both the original LoRA and its approximated version in Equation (20). The empirical evidence highlights the effectiveness of avoiding the low-rank constraint in our FLORA.

It should be emphasized that we cannot resample the down-projection matrix A in LoRA. This is because A and B are coupled during the updates, and if the down-projection matrix A is resampled, the already updated matrix B will not fit. On the other hand, our FLORA directly updates the weight matrix W during training, making it flexible to choose a random down-projection at every step.

Algorithm 1 Gradient accumulation with FLORA.

Require: rank $r \in \mathbb{Z}_+$, accumulating steps $\in \mathbb{Z}_+$
Require: gradient function $\nabla_W f_i(\cdot)$
Require: weight matrices $\mathcal{W} = \{W^{(l)} : \dim(W^{(l)}) = 2\}$

```
▷ Initialization of the accumulator state
for W ∈ 𝒲 do
    C_W ← 0^{n × r}                                   ▷ O(nr)
    s_W ← an independent random seed
end for
▷ Accumulating the compressed gradients
for i ∈ [ ] do
    for W ∈ 𝒲 do
        G_W ← ∇_W f_i(W)                              ▷ G_W ∈ R^{n × m}
        A_W ← N_{s_W}(0; 1/r)                         ▷ A_W ∈ R^{r × m}
        C_W ← C_W + G_W A_W^⊤                         ▷ Compresssion
    end for
end for
▷ Reconstruction
for W ∈ 𝒲 do
    A_W ← N_{s_W}(0; 1/r)                             ▷ A_W ∈ R^{r × m}
    G̃_W ← (1/n) C_W A_W                               ▷ Decompression
end for
return {G̃_W : W ∈ 𝒲}                                  ▷ Overwrite {G_W}
```

Sublinear memory gradient accumulation. One application of our FLORA is to compress the optimization states to save memory during training. We first show this with an example of gradient accumulation, which is widely used in practice to simulate a larger batch size (Smith et al., 2018). Specifically, it calculates the arithmetic mean (AM) of gradients for $$ steps and updates the model by the AM. In this way, the effective batch size is $$ times larger than the original batch size. However, it requires a memory buffer, whose size is equal to the model itself, to store the accumulated gradients.

In FLORA, we propose to compress the gradient accumulation with the down-projection. Within an accumulation cycle, we only maintain the accumulated gradient in the randomly down-projected space. During decompression, we can reuse the memory for gradients to reduce additional overheads. The resampling of the projection matrix occurs when an accumulation cycle is finished. The overall algorithm is summarized in Algorithm 1.

Sublinear memory momentum. The momentum technique is widely used in modern deep learning to reduce the variance of the gradient and accelerate training (Nesterov, 1998; Goh, 2017; Jelassi & Li, 2022). However, it requires maintaining an additional momentum scalar for each parameter, which is expensive when the model is large.

Similar to the compressed gradient accumulation, we can

Algorithm 2 Momentum with FLORA.

Require: decay rates β 1
Require: rank $r \in \mathbb{Z}_+$, interval $\in \mathbb{Z}_+$
Require: gradient function $\nabla_W f(\cdot)$
Require: weight $W = \{W^{(l)} : \dim(W^{(l)}) = 2\}$

B Initialize the optimizer state
$t \leftarrow 0$
for $W \in W$ do
  $M_{t;W} \leftarrow 0^{n \times r}$
  $s_{t;W} \leftarrow$ an independent random seed
end for
B Training procedure
while training not converged do
  for $W \in W$ do
    $G_{t;W} \leftarrow \nabla_W f_t(W)$ B $G_{t;W} \in \mathbb{R}^{n \times m}$
    $A_{t;W} \leftarrow N(0; 1/r)$ B $A_{t;W} \in \mathbb{R}^{r \times m}$
    if $t \equiv 0 \pmod{\ }$ then
      $s_{t+1;W} \leftarrow$ an independent random seed
      $A'_W \leftarrow N(0; 1/r)$ B $A'_W \in \mathbb{R}^{r \times m}$
      $M' \leftarrow M_{t;W} A_{t;W} A'^{\top}_W$
    else
      $s_{t+1;W} \leftarrow s_{t;W}$
      $A'_W \leftarrow A_{t;W}$
      $M' \leftarrow M_{t;W}$
    end if
    $M_{t+1;W} \leftarrow M' + (1 - \beta) G_{t;W} A'^{\top}_W$
  end for
  yield $\{M_{t+1;W} A'^{\top}_W : W \in W\}$ B Decompression
  $t \leftarrow t + 1$
end while

also compress the momentum with FLORA. For each time step t, we down-project the new gradient G_t by a random projection matrix $A_t^\top$. However, the difficulty for accumulating momentum emerges when we use a different A_t from A_{t-1}, as we cannot reconstruct the original momentum from $G_{t-1}A_{t-1}^\top + G_t A_t^\top$. This difficulty applies to all EMA updates where the number of accumulation steps is not finite. In this case, resampling new matrices will result in a loss of historical accumulation.

We propose two remedies to address this issue. First, we keep the same projection matrix for a long time to reduce the distortion. Second, we propose to transfer the compressed momentum from the old projection to a new one by $M' = M_{t-1}A_{t-1}A_t^\top$. This is justified by $A_{t-1}A_{t-1}^\top$ and $A_t^\top A_t$ are approximately the identity matrix based on Theorem 2.4.

The final algorithm is shown in Algorithm 2. Overall, for each weight matrix $W \in \mathbb{R}^{n \times m}$, we preserve the momentum term M_t with sublinear memory. Compared with the original momentum, we reduce the memory from $O(nm)$ to $O(nr)$. It should be pointed out that although we track momentum specifically in this case, our algorithm can be easily extended to other EMA-based statistics.

Memory analysis. It should be pointed out that neither LoRA nor our FLORA saves the memory for back-propagation. This is because $\frac{\partial \mathcal{L}}{\partial W}$ is needed for the update of A and B in LoRA (Dettmers et al., 2023), while in our approach we also compress and decompress the gradient.

That being said, saving the memory of optimization states alone could be critical to training large models (Dettmers et al., 2021). Our FLORA compresses both the AM and EMA of gradients to the sublinear level, which shares the same asymptotic rate as LoRA. In implementation, we may store the random seed that generates the projection matrix—which is highly efficient, as each element can be sampled independently with simple operations—instead of maintaining the same project matrix over batches. This allows the program to further save memory in practice with buffer reuse. On the contrary, LoRA needs to maintain two weight matrices A and B, as well as their AM or EMA matrices. Empirically shown in Section 3, FLORA consumes less memory than LoRA while facilitating high-rank updates and outperforming LoRA to a large extent.

3. Experiments

In this section, we empirically verify the effectiveness of our approach across different model architectures and datasets.

3.1. Experiment setup

Models. Given the exceptional ability of language models, we consider Transformer-based models for our experiments. Specifically, we select two representative models, including the T5 (Raffel et al., 2020) and GPT-2 (Radford et al., 2019) series. For the T5 series, we use T5-small to represent small models and T5-3B to represent large models. T5-small has around 60M parameters, with a hidden dimension set to 512, while T5-3B has around 3B parameters with a hidden dimension of 1024. For the GPT-2 series, we use the base version to represent small models and GPT-2-XL to represent large models. The base version has around 110M parameters, with a hidden dimension set to 768, while the large version has around 1.5B parameters with a hidden dimension of 1600.

Datasets. To facilitate evaluation, we use two conditional language modeling tasks, including summarization and translation.

For the summarization task, we train[1] T5 on the XSum dataset (Narayan et al., 2018). Each sample is a news article with a summary. The task is to generate a summary of the article. For each input, we prepend the prefix "summarize:" to the source sentence in the encoder (Raffel et al., 2020). The source and target sentences are truncated to 512 and

[1] In our experiments, we fine-tune a pre-trained model in the gradient accumulation experiment, while training from scratch in the momentum experiment (Section 3.2).

Table 1: The results of different methods to compress gradient accumulation. The size indicates the total number of the original model parameters. The numbers in the brackets denote the rank r of the random projection matrix.

(a) The results of T5 variants on XSum.

Size	Accumulation	Mem	Δ_M	$R_1/R_2/R_L$
60M	None	0.75	-	33.4/11.4/26.4
	Naive	0.87	0.12	34.0/11.5/26.7
	LoRA(8)	0.82	0.07	30.4/8.60/23.6
	LoRA(32)	0.86	0.11	30.7/8.90/23.9
	LoRA(128)	0.94	0.19	31.0/9.10/24.1
	LoRA(256)	1.07	0.32	31.4/9.34/24.5
	FLORA(8)	0.75	0.00	31.5/9.67/24.6
	FLORA(32)	0.75	0.00	32.2/10.3/25.2
	FLORA(128)	0.77	0.02	33.2/10.9/26.0
	FLORA(256)	0.79	0.04	33.6/11.3/26.5
3B	None	16.7	-	42.5/19.1/34.6
	Naive	26.6	9.9	44.4/20.9/36.3
	LoRA(16)	27.8	11.1	42.2/18.4/34.0
	LoRA(64)	29.5	12.8	42.3/18.6/34.1
	LoRA(256)	33.4	16.7	42.6/18.9/34.4
	LoRA(512)	OOM	-	-
	FLORA (16)	17.0	0.3	43.5/20.0/35.5
	FLORA (64)	18.2	1.5	43.9/20.3/35.8
	FLORA (256)	19.5	2.8	44.3/20.7/36.2
	FLORA (512)	22.1	5.4	44.5/20.9/36.4

(b) The results of GPT-2 variants on IWSLT17 De-En.

Size	Accumulation	Mem	Δ_M	BLEU
110M	None	2.77	-	17.9
	Naive	3.24	0.47	24.9
	LoRA(8)	3.25	0.48	9.94
	LoRA(32)	3.29	0.52	11.2
	LoRA(128)	3.38	0.60	12.2
	LoRA(256)	3.52	0.75	13.4
	FLORA (8)	2.93	0.15	16.3
	FLORA (32)	2.94	0.16	22.0
	FLORA (128)	2.98	0.20	24.0
	FLORA (256)	3.03	0.26	25.4
1.5B	None	20.8	-	28.2
	Naive	26.5	5.78	33.2
	LoRA(16)	26.8	6.02	17.4
	LoRA(64)	27.4	6.68	19.5
	LoRA(256)	28.9	8.15	20.7
	LoRA(512)	OOM	-	-
	FLORA (16)	21.1	0.34	29.7
	FLORA (64)	21.3	0.52	31.6
	FLORA (256)	21.9	1.17	33.2
	FLORA (512)	22.8	2.04	33.6

128 tokens, respectively.

For the translation task, we follow the setting of Lin et al. (2020) and train GPT-2 on the IWSLT-2017 German-English dataset (Cettolo et al., 2017). Each sample is a German sentence with its English translation. The task is to generate the English translation of the German sentence. For each input, we use the template "translate German to English: [source] . English: [target] " for training (Raffel et al., 2020).

Evaluation metrics. For the summarization task, we use the widely used ROUGE scores (Lin, 2004), including ROUGE-1, ROUGE-2, and ROUGE-L ($R_1/R_2/R_L$) to evaluate the quality of the generated summary. For the translation task, we use the most commonly used SacreBLEU score (Post, 2018) to evaluate the translation quality. For both ROUGE and SacreBLEU scores, the higher the score, the better the quality of the generated text.

To get more insights into the training process, we monitor the peak memory usage with the built-in JAX profiling tool (Bradbury et al., 2018). We also show the excessive memory Δ_M compared with the method where accumulation or momentum is disabled. The memory is reported in GiB (1024^3 bytes).

Competing methods. In our experiment, we take Adafactor as the base optimizer, which is the default optimizer for many Transformer models including T5 (Raffel et al., 2020), and is reported to be empirically better than Adam (Rae et al., 2021). We use the official Adafactor implementation in Optax (DeepMind et al., 2020).

We compare the following methods: (1) None: a baseline that does not use gradient accumulation or momentum; (2) Naive: a naive implementation of gradient accumulation or momentum, which stores the full information along training; (3) LoRA: the original LoRA method where only the LoRA patches are trained; (4) FLORA: our approach that compresses the gradients and decompresses them when updating the original weights. For LoRA and FLORA, we apply the projections to attention and feed-forward layers only, while following the naive procedure for other layers (i.e., token embeddings and vector weights).

For small models (T5-small and GPT-2 base), we test the rank r from 8 to 256, ranging from the very low dimension to half of the hidden dimension, for a thorough examination of different methods. For large models (T5-3B and GPT-2-XL), we test r from 16 to 512 to approximately maintain the same percentage of memory saving as small models. We do not apply learning rate schedules (Loshchilov & Hutter, 2017) or weight decay (Loshchilov & Hutter, 2019) in any experiments to rule out the influence of these techniques.

3.2. Main results

Gradient accumulation. In this setting, we fine-tune pre-trained models with 16 gradient accumulation steps. To achieve a minimal memory footprint and fit large models,

Table 2: The results of compressing momentum. The size indicates the total number of the original model parameters. The numbers in the brackets denote the rank r of the random projection matrix.

Setting	Momentum	Mem	Results
T5 60M XSum	None	1.65	29.4/9.11/23.3
	Naive	1.89	29.9/9.40/23.8
	LoRA(8)	1.88	18.0/3.33/14.9
	LoRA(32)	1.91	20.4/4.20/16.7/
	LoRA(128)	2.05	21.5/4.82/17.4
	LoRA(256)	2.13	22.2/5.04/17.9
	FLORA (8)	1.71	25.5/6.56/20.4
	FLORA (32)	1.72	26.9/7.32/21.5
	FLORA (128)	1.75	29.1/8.76/23.2
	FLORA (256)	1.79	30.2/9.51/24.0
GPT-2 110M IWSLT17	None	8.95	19.4
	Naive	9.45	19.9
	LoRA(8)	9.42	4.98
	LoRA(32)	9.46	6.76
	LoRA(128)	9.55	8.72
	LoRA(256)	9.76	9.83
	FLORA (8)	9.09	9.14
	FLORA (32)	9.10	14.9
	FLORA (128)	9.14	18.6
	FLORA (256)	9.20	19.9

the physical batch size is set to 1. We sweep the learning rate from 10^{-5} to 10^{-1} with the naive accumulation method on the validation loss. The best learning rate is applied to other methods, excluding LoRA, which is tuned individually as it is reported to have different optimal learning rates (Hu et al., 2022). For each run, the model is fine-tuned for 1 epoch to prevent over-fitting following the common practice (Wu et al., 2021). The results are reported on the test set based on the checkpoint with the lowest validation loss.

We present the results in Table 1. As shown, the naive gradient accumulation improves the ROUGE scores over the method without accumulation, but it leads to a large memory usage, which is similar to the model size, to store the accumulation. For LoRA, we empirically observe that it generally does not reduce memory usage in this case, as the state of Adafactor is already sublinear. In fact, it increases memory because it stores another four low-rank matrices for each weight matrix and adds an additional Jacobian path for the automatic differentiation.

On the contrary, our FLORA reduces the memory footprint on all benchmarks compared with the naive accumulation. In addition, when r is reasonably large, our method is able to recover the performance (in ROUGE or BLEU scores) of full-matrix accumulation and surpass the baseline that accumulation is not enabled. Notably, for the large models (T5-3B and GPT-2-XL), the memory overhead of FLORA ($r = 256$) is only 30% of the naive accumulation, while the performance is on par.

Table 3: The effect of κ in momentum.

Setting	κ	Mem	$R_1/R_2/R_L$
T5 60M XSum	1	1.79	0.00/0.00/0.00
	10	1.79	27.5/7.68/31.8
	100	1.79	29.3/8.89/23.2
	1000	1.79	30.4/9.70/24.2
	10000	1.79	29.5/9.11/23.5

Momentum. Given that the momentum technique is ineffective in fine-tuning (Li et al., 2020), we train all models from scratch in this setting. The physical batch size is set to 4 as a result of balancing the generalization and variance reduction (Masters & Luschi, 2018). We disable the gradient accumulation technique to rule out its impact. Due to the expense of training from scratch, we only test the small variants of each series. Similar to the settings in gradient accumulation, we sweep the learning rate for the naive momentum method from 10^{-5} to 10^{-1} on the validation loss. The best learning rates are applied to all methods excluding LoRA, which again has its own optimal learning rate. The hyper-parameter κ (resampling interval) is set to 1000 for all runs of FLORA. The effect of different values of κ is shown in Section 3.3.

As shown in Table 2, the naive momentum technique achieves better performance than at a cost of more memory usage. Similar to the results in gradient accumulation, LoRA does not save memory given the optimization state is already sublinear. It also has a significantly lower performance when trained from scratch, as the overall matrix update can only be low-rank.

Our FLORA utilizes less memory than the naive momentum. In addition, our method recovers (or even surpasses) the performance of naive momentum when r is increased. This significantly distinguishes our methods from LoRA as it achieves memory-efficient training even when the initialization is random.

3.3. In-depth analyses

The effect of κ in momentum. In our momentum implementation, we have a hyper-parameter κ controlling the resampling frequency of the random-projection matrix. In this part, we analyze the effect of κ with T5-small on the summarization task as our testbed, due to the limit of time and resources. We vary κ by keeping other hyper-parameters the same as in Section 3.2.

The results are shown in Table 3. It is seen that, when κ is below 1000, the ROUGE scores increase with κ. After a certain threshold, however, we see that the performance starts to decrease. This aligns with our interpretation that the information is better preserved within the interval, but

Table 4: The results of linear-memory optimizers.

Setting	Accumulation	Mem	R1/R2/RL
T5 60M XSum	None	0.99	33.0/11.1/26.1
	Naive	1.12	34.0/11.5/26.7
	LoRA(8)	0.82	28.7/7.51/22.0
	LoRA(32)	0.86	29.0/7.71/22.3
	LoRA(128)	1.00	29.7/8.02/22.9
	LoRA(256)	1.20	30.0/8.28/23.2
	FLORA (8)	1.00	31.6/9.72/24.7
	FLORA (32)	1.00	32.3/10.3/25.3
	FLORA (128)	1.00	33.2/10.9/26.0
	FLORA (256)	1.04	33.5/11.1/26.3

each interval is bottlenecked by the rank. Given the results, we choose = 1000 in Section 3.2 to balance the preserved information and the overall rank of momentum.

Optimizer with linear memory. In our main experiments, we observed a counter-intuitive phenomenon that LoRA empirically increases memory usage. This is likely because the optimization states in Adafactor are already sublinear, rendering the ineffectiveness of LoRA to save memory in this case. To further verify our method in linear-memory optimizers, we test the performance with a variant of Adafactor where the second-moment estimates are not factorized, essentially making it a linear-memory optimizer. All the other hyper-parameters remain the same as Section 3.2.

Table 4 shows the results. As seen, LoRA indeed saves more memory than our FLORA when the rank is low ($r < 128$) in linear-memory optimizers. However, our FLORA becomes more memory-efficient for $r = 256$, because we have a lower constant in the complexity. Moreover, our FLORA largely outperforms LoRA in all settings by 2–3 ROUGE points, showing the superiority of our approach.

3.4. Additional experiments

We additionally conducted several preliminary experiments during the author response phase and found the following observations: (1) FLORA performs well on images with ViT (Appendix C.1), (2) FLORA outperforms the concurrent GaLore in both memory reduction and model quality (Appendix C.2), and (3) FLORA can be combined with activation checkpointing (AC) and layer-wise update (LOMO) to further reduce the peak memory (Appendix C.3).

4. Related work and discussion

Parameter-efficient fine-tuning. Many methods have been proposed to improve the parameter efficiency of fine-tuning large models. A straightforward way is to tune a subset of the model, such as the top layers (Li & Liang, 2021) and bias vectors (Zaken et al., 2022). Another way is to add small tunable modules, e.g., the Adapter (Houlsby et al., 2019) and LoRA (Hu et al., 2022), to the pre-trained model. Although reducing the optimization memory, these methods suffer from the problem that the model parameters are restricted. For example, the total weight change of LoRA is constrained to be low-rank. In an attempt to achieve high-rank updates, ReLoRA (Lialin et al., 2023) proposes to periodically reinitialize the LoRA patch. However, it requires full-weight pre-training to work properly, growing the peak memory linearly in model size. We hence do not include ReLoRA as a sublinear-memory baseline. By contrast, our method is able to start directly from scratch and achieve the full training performance while maintaining a sublinear complexity throughout the process.

Matrix compression. Our method is closely connected to matrix compression techniques. For example, principal component analysis (Shlens, 2014) or matrix sketching (Liberty, 2013; Rothchild et al., 2020) use singular value decomposition (SVD) to approximate the large matrix with smaller matrices. However, the SVD procedure is computationally expensive and difficult to parallelize, making it impractical for large-scale training. Another way to compress the matrix is to use random projection (Bingham & Mannila, 2001), which lies the foundation of our method. Our method additionally involves a simple and efficient decompression procedure justified by Theorem 2.4. The simplification saves both computation and memory usage.

Memory-efficient optimizers. Optimization states contribute significantly to memory usage for large-scale training (Dettmers et al., 2021). Memory-efficient optimizers (Shazeer & Stern, 2018; Feinberg et al., 2023) are shown to effectively reduce the memory footprint. Our method is orthogonal to these methods, as it can be applied to enhance existing optimizers by compressing the momentum or gradient accumulation.

Memory-efficient automatic differentiation. It is also possible to reduce the memory footprint of back-propagation with advanced techniques like activation checkpointing (Chen et al., 2016), layer-by-layer updating (Lv et al., 2023), mixed-precision training (Micikevicius et al., 2018), randomized auto differentiation (Oktay et al., 2020), or zeroth-order optimization (Malladi et al., 2023). Technically, FLORA can be combined with these methods to further save memory. We demonstrate the combination of FLORA, activation checkpointing, and layer-by-layer updating in Appendix C.2 as an example. We leave further exploration to future work, given our focus in this paper is to compress optimization states.

5. Conclusion

Summary. In this work, we introduce FLORA, a method based on random projections that achieves sublinear memory usage for gradient accumulation and momentum. In addition, our approach effectively addresses the low-rank limitation of LoRA by resampling projection matrices. Experimental results demonstrate significant memory reduction with maintained model performance, highlighting the potential of random projection in deep learning.

Future work. In this paper, the largest model is 7B. For extremely large models like GPT-3, we estimate that the compressed optimization state of FLORA with $r = 256$ is only 2.08% of its original memory, which would be of great practical significance. We leave the empirical verification to future work. Further, the applicable scope of FLORA is not limited to Transformer-based models. We would like to test it on more architectures.

Impact statement

Our paper presents a memory-efficient method for deep learning to advance the field of machine learning. It is not expected to have direct consequences on the general public. We, however, anticipate it to have a positive impact on the environment by reducing the resources of model training.

Acknowledgments

We thank all reviewers for their insightful comments. The research is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), a Mitacs Accelerate project, the Amii Fellow Program, the Canada CIFAR AI Chair Program, an Alberta Innovates Program, and the Digital Research Alliance of Canada (alliancecan.ca).

References

Bingham, E. and Mannila, H. Random projection in dimensionality reduction: applications to image and text data. In KDD, pp. 245–250, 2001. URL https://dl.acm.org/doi/10.1145/502512.502546.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In NeurIPS, pp. 1877–1901, 2020. URL https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Cettolo, M., Federico, M., Bentivogli, L., Niehues, J., Stüker, S., Sudoh, K., Yoshino, K., and Federmann, C. Overview of the IWSLT 2017 evaluation campaign. In IWSLT, pp. 2–14, 2017. URL https://aclanthology.org/2017.iwslt-1.1.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174, 2016. URL https://arxiv.org/abs/1604.06174.

Cutkosky, A. and Orabona, F. Momentum-based variance reduction in non-convex SGD. In NeurIPS, 2019. URL https://papers.nips.cc/paper_files/paper/2019/hash/b8002139cdde66b87638f7f91d169d96-Abstract.html.

Dasgupta, S. Experiments with random projection. UAI, pp. 143–151, 2000. URL https://dl.acm.org/doi/10.5555/647234.719759.

Dasgupta, S. and Gupta, A. An elementary proof of a theorem of Johnson and Lindenstrauss. Random Structures & Algorithms, 22(1):60–65, 2003. URL https://doi.org/10.1002/rsa.10073.

DeepMind, Babuschkin, I., Baumli, K., Bell, A., Bhupatiraju, S., Bruce, J., Buchlovsky, P., Budden, D., Cai, T., Clark, A., Danihelka, I., Dedieu, A., Fantacci, C., Godwin, J., Jones, C., Hemsley, R., Hennigan, T., Hessel, M., Hou, S., Kapturowski, S., Keck, T., Kemaev, I., King, M., Kunesch, M., Martens, L., Merzic, H., Mikulik, V., Norman, T., Papamakarios, G., Quan, J., Ring, R., Ruiz, F., Sanchez, A., Sartran, L., Schneider, R., Sezener, E., Spencer, S., Srinivasan, S., Stanojević, M., Stokowiec, W., Wang, L., Zhou, G., and Viola, F. The DeepMind JAX Ecosystem, 2020. URL http://github.com/google-deepmind.

Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L. 8-bit optimizers via block-wise quantization. In ICLR, 2021. URL https://openreview.net/forum?id=shpkpVXzo3h.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. QLoRA: efficient finetuning of quantized LLMs. In NeurIPS, 2023. URL https://openreview.net/forum?id=OUIFPHEgJU.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale. In ICML, 2020. URL https://openreview.net/forum?id=YicbFdNTTy.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. JMLR, 12(7), 2011. URL https://jmlr.org/papers/v12/duchi11a.html.

Feinberg, V., Chen, X., Sun, Y. J., Anil, R., and Hazan, E. Sketchy: Memory-efficient adaptive regularization with frequent directions. In NeurIPS, 2023. URL https://openreview.net/forum?id=DeZst6dKyi.

Finesso, L. and Spreij, P. Nonnegative matrix factorization and I-divergence alternating minimization. Linear Algebra and its Applications, 416(2-3):270–287, 2006. URL https://doi.org/10.1016/j.laa.2005.11.012.

Goh, G. Why momentum really works. Distill, 2017. URL http://distill.pub/2017/momentum.

Hinton, G., Srivastava, N., and Swersky, K. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. Coursera, 2012. URL https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., de Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-efficient transfer learning for NLP. In ICML, pp. 2790–2799, 2019. URL https://proceedings.mlr.press/v97/houlsby19a.html.

Hu, E. J., yelong shen, Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-rank adaptation of large language models. In ICLR, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Indyk, P. and Motwani, R. Approximate nearest neighbors: towards removing the curse of dimensionality. In STOC, pp. 604–613, 1998. URL https://dl.acm.org/doi/10.1145/276698.276876.

Jelassi, S. and Li, Y. Towards understanding how momentum improves generalization in deep learning. In ICML, pp. 9965–10040, 2022. URL https://openreview.net/forum?id=lf0W6tcWmh-.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), ICLR, 2015. URL http://arxiv.org/abs/1412.6980.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. Technical report, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Laurent, B. and Massart, P. Adaptive estimation of a quadratic functional by model selection. Annals of statistics, 28(5):1302–1338, 2000. URL https://www.jstor.org/stable/2674095.

Li, H., Chaudhari, P., Yang, H., Lam, M., Ravichandran, A., Bhotika, R., and Soatto, S. Rethinking the hyperparameters for fine-tuning. In ICLR, 2020. URL https://openreview.net/forum?id=B1g8VkHFPH.

Li, X. L. and Liang, P. Prefix-tuning: Optimizing continuous prompts for generation. In ACL-IJCNLP, volume 1, pp. 4582–4597, 2021. URL https://aclanthology.org/2021.acl-long.353.

Lialin, V., Shivagunde, N., Muckatira, S., and Rumshisky, A. Stack more layers differently: High-rank training through low-rank updates. arXiv preprint arXiv: 2307.05695, 2023. URL https://arxiv.org/abs/2307.05695.

Liberty, E. Simple and deterministic matrix sketching. In KDD, pp. 581–588, 2013. URL https://doi.org/10.1145/2487575.2487623.

Lin, C.-Y. ROUGE: A package for automatic evaluation of summaries. In Text Summarization Branches Out, pp. 74–81, 2004. URL https://aclanthology.org/W04-1013.

Lin, Z., Madotto, A., and Fung, P. Exploring versatile generative language model via parameter-efficient transfer learning. In EMNLP Findings, pp. 441–459, 2020. URL https://aclanthology.org/2020.findings-emnlp.41.

Loshchilov, I. and Hutter, F. SGDR: Stochastic gradient descent with warm restarts. In ICLR, 2017. URL https://openreview.net/forum?id=Skq89Scxx.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In ICLR, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Lv, K., Yang, Y., Liu, T., Gao, Q., Guo, Q., and Qiu, X. Full parameter fine-tuning for large language models with limited resources. arXiv preprint arXiv: 2306.09782, 2023. URL https://arxiv.org/abs/2306.09782.

Malladi, S., Gao, T., Nichani, E., Damian, A., Lee, J. D., Chen, D., and Arora, S. Fine-tuning language models with just forward passes. In NeurIPS, 2023. URL https://openreview.net/forum?id=Vota6rFhBQ.

Masters, D. and Luschi, C. Revisiting small batch training for deep neural networks. arXiv preprint arXiv:1804.07612, 2018. URL https://arxiv.org/abs/1804.07612.

Matousek, J. On variants of the Johnson–Lindenstrauss lemma. Random Structures & Algorithms, 33(2):142–156, 2008. URL https://doi.org/10.1002/rsa.20218.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training. In ICLR, 2018. URL https://openreview.net/forum?id=r1gs9JgRZ.

Narayan, S., Cohen, S. B., and Lapata, M. Don't give me the details, just the summary! Topic-aware convolutional neural networks for extreme summarization. In EMNLP, pp. 1797–1807, 2018. URL https://aclanthology.org/D18-1206.

Nesterov, Y. Introductory Lectures on Convex Optimization: A Basic Course. Springer, 1998. URL https://doi.org/10.1007/978-1-4419-8853-9.

Oktay, D., McGreivy, N., Aduol, J., Beatson, A., and Adams, R. P. Randomized automatic differentiation. In ICLR, 2020. URL https://openreview.net/forum?id=xpx9zj7CUlY.

Post, M. A call for clarity in reporting BLEU scores. In WMT, pp. 186–191, 2018. URL https://aclanthology.org/W18-6319.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. OpenAI blog, 2019. URL https://openai.com/research/better-language-models.

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., et al. Scaling language models: Methods, analysis & insights from training Gopher. arXiv preprint arXiv:2112.11446, 2021. URL https://arxiv.org/abs/2112.11446.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. JMLR, 21(1):5485–5551, 2020. URL http://jmlr.org/papers/v21/20-074.html.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. High-resolution image synthesis with latent diffusion models. In CVPR, pp. 10684–10695, 2022. URL https://openaccess.thecvf.com/content/CVPR2022/html/Rombach_High-Resolution_Image_Synthesis_With_Latent_Diffusion_Models_CVPR_2022_paper.html.

Rothchild, D., Panda, A., Ullah, E., Ivkin, N., Stoica, I., Braverman, V., Gonzalez, J., and Arora, R. FetchSGD: Communication-efficient federated learning with sketching. In ICML, pp. 8253–8265, 2020. URL https://proceedings.mlr.press/v119/rothchild20a.html.

Shazeer, N. and Stern, M. Adafactor: Adaptive learning rates with sublinear memory cost. In ICML, pp. 4596–4604, 2018. URL https://proceedings.mlr.press/v80/shazeer18a.html.

Shlens, J. A tutorial on principal component analysis. arXiv preprint arXiv:1404.1100, 2014. URL https://arxiv.org/abs/1404.1100.

Smith, S. L., Kindermans, P.-J., Ying, C., and Le, Q. V. Don't decay the learning rate, increase the batch size. In ICLR, 2018. URL https://openreview.net/forum?id=B1Yy1BxCZ.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv: 2307.09288, 2023. URL https://arxiv.org/abs/2307.09288.

Wang, C., Chen, X., Smola, A. J., and Xing, E. P. Variance reduction for stochastic gradient optimization. In NIPS, 2013. URL https://papers.nips.cc/paper_files/paper/2013/hash/9766527f2b5d3e95d4a733fcfb77bd7e-Abstract.html.

Wu, J., Ouyang, L., Ziegler, D. M., Stiennon, N., Lowe, R., Leike, J., and Christiano, P. Recursively summarizing books with human feedback. arXiv preprint arXiv:2109.10862, 2021. URL https://arxiv.org/abs/2109.10862.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-MNIST: a novel image dataset for benchmarking machine learning

algorithms. arXiv preprint arXiv:1708.07747, 2017. URL https://arxiv.org/abs/1708.07747.

Zaken, E. B., Goldberg, Y., and Ravfogel, S. BitFit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In ACL, volume 2, pp. 1–9, 2022. URL https://aclanthology.org/2022.acl-short.1.

Zhao, J., Zhang, Z., Chen, B., Wang, Z., Anandkumar, A., and Tian, Y. GaLore: Memory-efficient LLM training by gradient low-rank projection. arXiv preprint arXiv:2403.03507, 2024. URL https://arxiv.org/abs/2403.03507.

A. Proof of Theorem 2.1

Theorem 2.1. Let LoRA update matrices A and B with SGD for every step t by

$$A_{t+1} \quad A_t \quad B_t^{\top}(r \ W L_t) \tag{5}$$

$$B_{t+1} \quad B_t \quad (r \ W L_t)A_t^{\top}; \tag{6}$$

where $$ is the learning rate. We assume $\sum_{t=0}^{T} r \ W L_t \ _F \quad L$ for every T during training, which implies that the model stays within a nite Euclidean ball. In this case, the dynamics of A_t and B_t are given by

$$A_T = A_0 + \ A_0 f_A(T); \quad B_T = \ f_B(T)A_0^{\top}; \tag{7}$$

where the forms of $f_A(t) \ 2 \ R^{m \ m}$ and $f_B(t) \ 2 \ R^{n \ m}$ are expressed in the proof. In particular, $\|f_A(t)\|_2 \ \frac{L^2 \ 1 \ (\ ^2L^2)^t}{1 \ ^2L^2}$ for every t.

Before proving the theorem, we need to obtain the form of $f_A(t)$ and $f_B(t)$. We derive them in the following lemma.

Lemma A.1. When $t = 0$, $f_A(0) = f_B(0) = 0$. For $t \quad 1$, the values of $f_A(t)$ and $f_B(t)$ are iteratively obtained by:

$$f_A(t) = \sum_{i=0}^{t \ 1} f_B^{\top}(i)(r \ W L_i) \tag{25}$$

$$f_B(t) = \sum_{i=0}^{t \ 1} (r \ W L_i)(\ f_A^{\top}(i) + I) \tag{26}$$

Proof. We prove this by induction. For the base case $t = 0$, it is straightforward to show $f_A(0) = f_B(0) = 0$.

Assume $A_t = A_0 + \ A_0 f_A(t)$ and $B_t = \ f_B(t)A_0^{\top}$ holds with such functions f_A and f_B for $1 : : : t$. Then for $t + 1$, we have

$$A_{t+1} = A_t \quad B_t^{\top}(r \ W L_t) \tag{27}$$

$$= A_0 + \ A_0 f_A(t) \quad ^2 A_0 f_B^{\top}(t)(r \ W L_t) \tag{28}$$

$$= A_0 + \ A_0 \ f_A(t) \quad f_B^{\top}(t)(r \ W L_t) \tag{29}$$

$$= A_0 + \ A_0 (f_A(t+1)) \ ; \tag{30}$$

where we have $f_A(t+1) = f_A(t) \quad f_B^{\top}(t)(r \ W L_t) = \sum_{i=0}^{t} f_B^{\top}(i)(r \ W L_i)$ in the last line.

Similarly, we have

$$B_{t+1} = B_t \quad (r \ W L_t)A_t^{\top} \tag{31}$$

$$= f_B(t)A_0^{\top} \quad (r \ W L_t)(\ f_A^{\top}(t) + I)A_0^{\top} \tag{32}$$

$$= \ f_B(t) \quad (r \ W L_t)(\ f_A^{\top}(t) + I) \ A_0^{\top} \tag{33}$$

$$= f_B(t+1) A_0^{\top}; \tag{34}$$

where we have $f_B(t+1) = f_B(t) \quad (r \ W L_t)(\ f_A^{\top}(t) + I) = \sum_{i=0}^{t} (r \ W L_i)(\ f_A^{\top}(i) + I)$ in the last line. □

Proof of Theorem 2.1. Define $a_t := \frac{L^2 \ 1 \ (\ ^2L^2)^t}{1 \ ^2L^2}$. We prove $\|f_A(t)\| \quad a_t$ by induction. For the base case $t = 0$, it is trivial to see $\|f_A(0)\| = 0 \quad a_0$. We then assume $\|f_A(i)\| \quad a_i$ for $i \quad t \quad 1$. Since a_t is monotonically increasing, we know $\|f_A(i)\| \quad a_{t \ 1}$ for $i \quad t \quad 1$. By using Lemma A.1, we have

$$f_A(t) = \sum_{i=0}^{t \ 1} f_B^{\top}(i)(r \ W L_i) \tag{35}$$

$$= \sum_{i=1}^{t \ 1} \sum_{j=0}^{i \ 1} (\ f_A(j) + I)(r \ W L_j)^{\top}(r \ W L_i): \tag{36}$$

Taking the norm, we have

$$\|f_A(t)\|_F = \Big\| \sum_{i=1}^{t-1} \sum_{j=0}^{i-1} (\eta f_A(j) + I)(r_W L_j)^\top (r_W L_i) \Big\|_F \tag{37}$$

$$\leq \eta^2 \Big\| \sum_{j=0}^{t-2} (f_A(j))(r_W L_j)^\top \sum_{i=j+1}^{t-1} (r_W L_i) \Big\|_F + \Big\| \sum_{i=1}^{t-1} \sum_{j=0}^{i-1} (r_W L_j)^\top (r_W L_i) \Big\|_F \tag{38}$$

$$\leq \eta^2 L^2 \Big\| \sum_{j=0}^{t-2} (f_A(j))(r_W L_j)^\top \Big\|_F + L^2 \qquad \text{(Lemma A.2)}$$

$$\leq \eta^2 L^2 a_{t-1} + L^2 \qquad \text{(Lemma A.3)}$$

$$= \eta^2 L^2 \frac{L^2 - (\eta^2 L^2)^t}{1 - \eta^2 L^2} + L^2 \tag{39}$$

$$= \frac{L^2 - (\eta^2 L^2)^{t+1}}{1 - \eta^2 L^2} \tag{40}$$

$$= a_t. \tag{41}$$

Therefore, we have $\|f_A(t)\| \leq \|f_A(t)\|_F \leq a_t$ for every t. □

Lemma A.2. If $\|\sum_{k=0}^{t-1} (r_W L_j)^\top\|_F \leq L$, we have

$$\Big\| \sum_{i=1}^{t-1} \sum_{j=0}^{i-1} (r_W L_j)^\top (r_W L_i) \Big\|_F \leq L^2 \tag{42}$$

for every t.

Proof. Let $G(k) := r_W L_k$ for simplicity. Taking the square,

$$\Big\| \sum_{n} \sum_{m=1}^{n} G(m)^\top G(n) \Big\|_F^2 \tag{43}$$

$$= \sum_{i,j} \sum_{n} \sum_{m=1}^{n} [G(m)^\top G(n)]_{i,j}^2 \tag{44}$$

$$= \sum_{i,j} \sum_{n} \sum_{m=1}^{n} \Big(\sum_{k} [G(m)]_{k,i} [G(n)]_{k,j} \Big)^2 \tag{45}$$

$$\leq \sum_{m} \sum_{i} \sum_{k} [G(m)]_{k,i}^2 \sum_{m=n} \sum_{j} \sum_{k} [G(n)]_{k,j}^2 \qquad \text{(Cauchy-Schwarz)}$$

$$= \Big(\sum_{m} \sum_{i} \sum_{k} [G(m)]_{k,i}^2 \Big)^2 \tag{46}$$

$$= \Big(\sum_{m} \sum_{i} \sum_{k} [G(m)]_{k,i}^2 \Big)^2 \tag{47}$$

$$= \Big(\sum_{m} \|G(m)\|_F^2 \Big)^2 \tag{48}$$

$$\leq L^4, \tag{49}$$

which completes the proof by taking square roots on both sides. □

Lemma A.3. If $\|f_A(k)\|_F \le a_k$ for all $k < t$, we have

$$\left\| \sum_{k=0}^{t-1} f_A(k)(r \otimes L_k)^\top \right\|_F \le a_{t-1} L \tag{50}$$

for every t.

Proof. Let $A(k) := f_A(k)$ and $G(k) = (r \otimes L_k)$ for simplicity. Taking the square,

$$\left\| \sum_{k=0}^{t-1} A(k)G(k)^\top \right\|_F^2 \tag{51}$$

$$= \sum_{k=0}^{t-1} \sum_i \sum_j [A(k)G(k)^\top]_{i,j}^2 \tag{52}$$

$$= \sum_{k=0}^{t-1} \sum_i \sum_j \sum_l [A(k)]_{i,l} [G(k)]_{j,l}^2 \tag{53}$$

$$= \sum_{k=0}^{t-1} \sum_i \sum_l [A(k)]_{i,l}^2 \sum_j \sum_l [G(k)]_{j,l}^2 \qquad \text{(Cauchy–Schwarz)}$$

$$\le \max_{0 \le k < t} \sum_i \sum_l [A(k)]_{i,l}^2 \sum_{k=0}^{t-1} \sum_j \sum_l [G(k)]_{j,l}^2 \tag{54}$$

$$\le \max_{0 \le k < t} \sum_i \sum_l [A(k)_{i,l}]^2 \sum_j \sum_l \Big(\sum_{k=0}^{t-1} [G(k)]_{j,l} \Big)^2 \tag{55}$$

$$= \max_{0 \le k < t} \|A(k)\|_F^2 \left\| \sum_{k=0}^{t-1} G(k) \right\|_F^2 \tag{56}$$

$$\le a_{t-1}^2 L^2; \tag{57}$$

which completes the proof by taking square roots on both sides. □

B. Proof of Theorem 2.4

Theorem 2.4. Let A be a matrix of shape $\mathbb{R}^{r \times m}$ where each element is independently sampled from a standard Gaussian distribution. Let $\varepsilon, \delta \in (0, 1]$. There exists a constant c such that when $r = c \log(2m/\delta)/\varepsilon^2$, we have for all i, j that

$$[A^\top A - I]_{i,j} \tag{24}$$

with confidence at least $1 - \delta$.

Proof. For each element of $A^\top A$, we have

$$[A^\top A]_{i,j} = \begin{cases} \sum_{k=1}^{r} a_{k,i}^2, & \text{if } i = j; \\ \sum_{k=1}^{r} a_{k,i} a_{k,j}, & \text{otherwise,} \end{cases} \tag{58}$$

where each element $a_{i,k}$ is an independent random variable following $\mathcal{N}(0, 1)$.

For $z_{i,i} := \sum_{k=1}^{r} a_{k,i}^2$, it follows the $\chi^2(r)$ distribution. By the standard Laurent-Massart bounds (Laurent & Massart, 2000), we obtain

$$\varepsilon_{i,i} := \frac{z_{i,i}}{r} - 1 \le 2\sqrt{\frac{\log(2/\delta)}{r}} + 2\frac{\log(2/\delta)}{r} \tag{59}$$

with probability at least $1 - \frac{\delta_0}{1}$.

For $z_{i,j} := \sum_{k=1}^{r} a_{k,i} a_{k,j}$ where $i \neq j$, we can rewrite it as $z_{i,j} = \sum_{k=1}^{r} [(\frac{a_{k,i} + a_{k,j}}{2})^2 - (\frac{a_{k,i} - a_{k,j}}{2})^2]$. In addiiton, all $(\frac{a_{k,i} + a_{k,j}}{2})^2$ and $(\frac{a_{k,i} - a_{k,j}}{2})^2$ are i.i.d. $\chi^2(1)$ distributions. Define

$$z^+_{i,j} := \sum_{k=1}^{r} \left(\frac{a_{k,i} + a_{k,j}}{2}\right)^2 \quad \text{and} \quad z^-_{i,j} := \sum_{k=1}^{r} \left(\frac{a_{k,i} - a_{k,j}}{2}\right)^2; \tag{60}$$

it is easy to see that $z^+_{i,j}$ and $z^-_{i,j}$ are i.i.d. $\chi^2(r)$ distributions. In addition $z_{i,j} = z^+_{i,j} - z^-_{i,j}$. Therefore, we have

$$\epsilon_{i,j} := \frac{z_{i,j}}{r} = \frac{z^+_{i,j}}{r} - 1 - \frac{z^-_{i,j}}{r} - 1 \tag{61}$$

$$\leq \frac{z^+_{i,j}}{r} - 1 + \left|\frac{z^-_{i,j}}{r} - 1\right| \tag{62}$$

$$\leq 4\sqrt{\frac{\log(4/\delta_0/2)}{r}} + 4\frac{\log(4/\delta_0/2)}{r} \tag{63}$$

with probability at least $1 - \frac{\delta_0}{2}$.

By using a union bound upon Equations (59) and (63), we can obtain

$$\epsilon_{i,j} \leq 4\sqrt{\frac{2\log(2m/\delta)}{r}} + 4\frac{2\log(2m/\delta)}{r} \tag{64}$$

for all i, j with probability at least $1 - \delta$. Under this condition, we further have

$$\epsilon_{i,j} = 4\sqrt{\frac{2\log(2m/\delta)}{128\log(2m/\delta)^2}} + 4\frac{2\log(2m/\delta)}{128\log(2m/\delta)^2} \qquad (\text{Let } r = 128\log(2m/\delta)^2)$$

$$= \frac{1}{2}(\epsilon + \epsilon^2) \tag{65}$$

$$\leq \epsilon; \qquad (\epsilon^2 \leq \epsilon \text{ as } \epsilon \leq 1)$$

concluding the proof by noticing $\epsilon_{i,j} = [A^\top A - I]_{i,j}$. □

C. Additional results

C.1. Beyond the text modality

Although our main experiments were conducted on datasets with text generation, our method can actually be applied to the matrix multiplication in different architectures. To verify the performance on other modalities beyond text, we evaluate the performance with the Vision Transformer (ViT) (Dosovitskiy et al., 2020) on the CIFAR-100 (Krizhevsky et al., 2009) image classification dataset.

The results are shown in Table 5. Consistent with main experiments, FLORA saves as much as 32.4% of training memory without sacrificing the accuracy. This confirms that FLORA is agnostic to model architectures and domains, demonstrating the generality of our approach.

C.2. Comparison with GaLore

Contemporary to our work, another optimizer, GaLore (Zhao et al., 2024), adapts similar down- and up-projections for memory efficiency. The key difference between FLORA and GaLore is that FLORA randomly generates the projection

Table 5: The image classification results on CIFAR-100 with ViT model.

Model size	Optimizer	Accuracy	Memory
Base	Adam	91.93	4.12 GiB
	FLORA	**92.15**	**3.14 GiB** (-23.8%)
Large	Adam	92.97	8.57 GiB
	FLORA	**92.98**	**5.79 GiB** (-32.4%)

Table 6: The results of language modeling on the C4 dataset with FLORA and GaLore. "PPL" is the token-level perplexity. The lower the PPL, the better. For the large model (7B), we only report the memory usage with a smaller batch size of 16 as the training takes months to complete.

Model size	Optimizer	PPL	Memory
60 M	GaLore	34.64	27.7
	FLORA	**32.52**	**27.5**
350 M	GaLore	27.17	36.50
	FLORA	**23.69**	**36.48**
7 B	GaLore	-	22.9
	FLORA	-	**21.2**

matrices on the fly, whereas GaLore performs SVD operations and stores the matrices on the device. To understand their empirical performance, we evaluate both methods on the language modelling pre-training task using the C4 dataset (Raffel et al., 2020). We apply the same hyper-parameters as suggested in the original GaLore paper (Zhao et al., 2024) for both methods, except that the learning rate is 3 times smaller for FLORA. The model architectures are adapted from Llama-2 (Touvron et al., 2023) but modified to contain target parameter sizes.

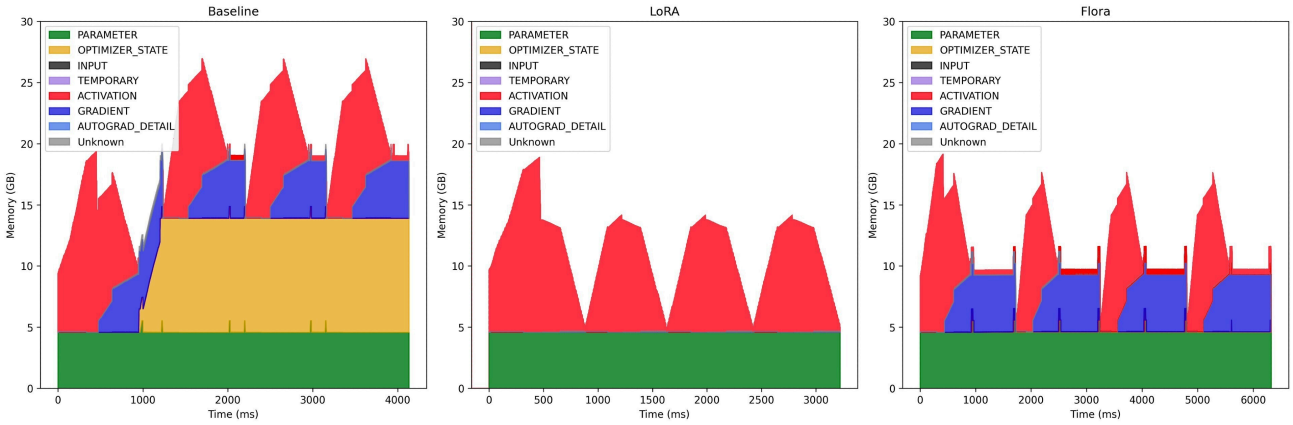
We present the results in Table 6. We notice that GaLore shows a similar level of memory usage as FLORA with a small overhead. This difference in memory is likely because GaLore stores additional projection matrices. Further, our FLORA achieves a better performance in terms of perplexity, suggesting its strong capability in optimization.

As discussed in Section 4, Our FLORA is orthogonal to many memory-efficient automatic differentiation techniques. To show this, we additionally enable the activation checkpointing (Chen et al., 2016) and layer-by-layer updating like LOMO (Lv et al., 2023) for the 7B model. Specifically, activation checkpointing recomputes the activations during back-propagation instead of storing them in the forward pass. LOMO, on the other hand, promptly updates the layer weight upon obtaining its gradient without waiting until all layers are finished. Both techniques are unaffected when using FLORA. As a result, the peak memory allocated is only 21.2GiB for the 7B model, surpassing GaLore's memory saving of the same setting.

C.3. Profiling results

We theoretically analyzed the memory usage of FLORA and LoRA in Section 2.4. In particular, we assert that both FLORA and LoRA reduce memory usage by maintaining smaller optimization states. To empirically verify this, we conduct the memory profiling analysis throughout the training time. Specifically, we perform 4 training steps of a T5 model for all methods, including the vanilla training with Adam, LoRA, and FLORA. The sequence length is padded to 512, and the batch size is set as 4. We categorize the memory usage following PyTorch's convention. Additionally, we evaluate these methods in the setting where more memory-efficient training techniques like activation checkpointing (Chen et al., 2016) and LOMO (Lv et al., 2023) are enabled. We plot the profiling results in Figure 2.

In Figure 2a, we observe that both LoRA and FLORA have negligible memory footprint of the optimization states. Although FLORA uses a larger gradient storage, the peak memory is dominated by activations rather than gradients, resulting in a similar amount of peak memory usage. Further, as observed in Figure 2b, the difference in the gradient storage becomes less significant when activation checkpointing (AC) and LOMO are applied. In this case, both LoRA and FLORA demonstrate a



(a) The vanilla training.

(b) AC and LOMO enabled.

Figure 2: Profiling the memory usage by categories during four iterations of training steps.

similar memory usage pattern. We additionally provide the animation for the procedure for better illustration.[2]

[2]Please refer to our repository at `https://github.com/BorealisAI/flora-opt`.