SMMF: Square-Matricized Momentum Factorization for Memory-Efficient Optimization

Kwangryeol Park[1], **Seulki Lee**[2]

[1]Artificial Intelligence Graduate School UNIST, South Korea
[2]Department of Computer Science and Engineering, UNIST, South Korea

Abstract

We propose SMMF (Square-Matricized Momentum Factorization), a memory-efficient optimizer that reduces the memory requirement of the widely used adaptive learning rate optimizers, such as Adam, by up to 96%. SMMF enables flexible and efficient factorization of an arbitrary rank (shape) of the first and second momentum tensors during optimization, based on the proposed square-matricization and one-time single matrix factorization. From this, it becomes effectively applicable to any rank (shape) of momentum tensors, i.e., bias, matrix, and any rank-d tensors, prevalent in various deep model architectures, such as CNNs (high rank) and Transformers (low rank), in contrast to existing memory-efficient optimizers that applies only to a particular (rank-2) momentum tensor, e.g., linear layers. We conduct a regret bound analysis of SMMF, which shows that it converges similarly to non-memory-efficient adaptive learning rate optimizers, such as AdamNC, providing a theoretical basis for its competitive optimization capability. In our experiment, SMMF takes up to 96% less memory compared to state-of-the-art memory-efficient optimizers, e.g., Adafactor, CAME, and SM3, while achieving comparable model performance on various CNN and Transformer tasks.

Code — https://github.com/eai-lab/SMMF
Extended version — https://arxiv.org/abs/2412.08894

1 Introduction

To identify the optimal weight parameters of deep neural networks, various optimization methods (Abdulkadirov, Lyakhov, and Nagornov 2023; Martens 2016; Amari 2010; Liu and Nocedal 1989) have been studied. One of the most popular approaches is SGD (Stochastic Gradient Descent) (Ruder 2016) which takes the weight update direction towards the current gradient with a learning rate uniformly applied to all weight parameters. To further improve SGD's optimization performance, many adaptive learning rate optimizers, such as Adam (Kingma and Ba 2014) and RMSProp (Hinton, Srivastava, and Swersky 2012), have been proposed to leverage 1) history of the gradients to compute the momentum direction (Ruder 2016) and 2) the squared gradients to compute the adaptive learning rate for each weight parameter. Despite their lack of theoretical convergence guarantee in non-convex settings of many deep learning tasks, those adaptive learning rate optimizers have been empirically found to outperform SGD in practice.

However, since the momentum value of each weight parameter, which linearly increases over the size of a deep learning model, should be maintained in memory during the whole training process, the adaptive learning rate optimizers can easily limit the size of models that can be trained on memory-constrained platforms, e.g., embedded systems. Even when training small models like Transformer-base (Vaswani et al. 2017), 1.4 GiB of memory is required. This means it would be unusable in environments with extremely limited memory devices, such as Raspberry Pi (1 GiB). To tackle the memory challenge of the adaptive learning rate optimization, several memory-efficient optimizers have been proposed. Adafactor (Shazeer and Stern 2018) and CAME (Luo et al. 2023) factorize the 2^{nd} momentum in the form of a matrix into a set of vectors to decrease the memory space required to store momentums, achieving comparable performance to Adam. SM3 (Anil et al. 2019) reduces memory usage by approximating the similar elements of the 2^{nd} momentum into a smaller set of variables. Although they effectively reduce the memory space of adaptive learning rate optimizers by projecting a gradient tensor onto several rank-one vectors, 1) they apply only to a specific rank (shape) and pattern of momentum tensors, 2) their memory space is still huge (1.1 GiB) making them unsuitable for memory constrained devices, and 3) their optimization performance has not been theoretically analyzed and compared to that of Adam family (Kingma and Ba 2014).

In this paper, we propose SMMF (Square-Matricized Momentum Factorization), a memory-efficient optimizer amicable to an arbitrary rank (shape) and pattern of both the 1^{st} and 2^{nd} momentum tensors, i.e., a vector, matrix, and rank-d tensor, which reduces the amount of memory required in model optimization by up to 96% compared to existing memory-efficient optimizers, e.g., Adafactor, CAME, and SM3. Unlike such existing memory-efficient optimizers, either confined to a particular 1) momentum rank (shape) (i.e., a rank-2 matrix) and/or 2) momentum pattern (i.e., a set of similar elements in a matrix) (Anil et al. 2019), the proposed SMMF performs competitive optimization without being restricted by the rank (shape) and pattern of momentums al-

lowing the models to be trained on extremely memory constrained embedded systems from ∼0.001 to ∼1 GiB.

Given a rank-d momentum tensor as $\mathbb{R}^{n_1 \times \ldots \times n_d}$, SMMF first finds $\hat{n}, \hat{m} = \arg\min_{n,m} |n - m|$ such that $nm = \prod_{r=1}^{d} n_r$. Next, it converts the momentum $\mathbb{R}^{n_1 \times \ldots \times n_d}$ into a matrix closest to square matrix $\mathbb{R}^{\hat{n} \times \hat{m}}$ with $\hat{n}$ and $\hat{m}$, which we call square-matricization. Then, the matrix $\mathbb{R}^{\hat{n} \times \hat{m}}$ is factorized into two vectors, $\mathbb{R}^{\hat{n} \times 1}$ and $\mathbb{R}^{1 \times \hat{m}}$ at one go, by using NNMF (Non-Negative Matrix Factorization) (Finesso and Spreij 2006). Since SMMF only stores the resulting two vectors $\mathbb{R}^{\hat{n} \times 1}$ and $\mathbb{R}^{1 \times \hat{m}}$ in memory, factorized from both the 1^{st} and 2^{nd} momentum $\mathbb{R}^{\hat{n} \times \hat{m}}$ that has been squared-matricized from the original rank-d momentum $\mathbb{R}^{n_1 \times \ldots \times n_d}$, it can decrease more memory when given high-rank momentums, e.g., the rank-4 weight tensors in CNNs. It is different from existing memory-efficient optimizers, e.g., Adafactor and CAME, that store $\prod_{r=1}^{d-2} n_r$ pairs of vectors factorized from a rank-d momentum $\mathbb{R}^{n_1 \times \ldots \times n_d}$ in memory.

We analyze the regret bound of the proposed SMMF, proving that its optimization performance in a convex setup is similar to one of the Adam-based optimizers, i.e., AdamNC (Reddi, Kale, and Kumar 2019) that applies the beta schedule to Adam. To the best of our knowledge, SMMF is the first factorization-based memory-efficient optimizer that conducts a regret bound analysis; none of the existing memory-efficient optimizers, e.g., Adafactor and CAME, provides such a theoretical study. The experiments on various CNN and Transformer models (Section 5) show the competitive results substantiating our analysis.

2 Related Work

Adafactor (Shazeer and Stern 2018) factorizes the 2^{nd} momentum matrix via Non-Negative Matrix Factorization (NNMF) (Finesso and Spreij 2006) that decomposes a non-negative matrix into two vectors by differentiating the I-divergence (Lee and Seung 1999). Theoretically, it reduces the memory complexity of the 2^{nd} momentum in the form of a non-negative matrix, i.e., $\boldsymbol{V} \in \mathbb{R}^{n \times m}$, from $\mathcal{O}(nm)$ to $\mathcal{O}(n + m)$ with the two factorized vectors. Empirically, it shows comparable optimization to Adam on Transformers.

CAME (Luo et al. 2023), a variant of Adafactor, is proposed as a memory-efficient optimizer for large batch optimization. To alleviate the unstable behavior of Adafactor, it introduces the factorized confidence term that guides the optimization direction, empirically achieving faster convergence on language models (Raffel et al. 2020; Radford et al. 2019) at the cost of using more memory than Adafactor. Since CAME also requires a momentum to be a non-negative matrix to be factorized with NNMF, it slices a high-rank weight tensor, appearing in CNN models such as MobileNet (Dong et al. 2020), into multiple matrices and factorize them separately. Hence, given a rank-d 2^{nd} momentum $\boldsymbol{V} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$, the memory complexity of CAME becomes $\mathcal{O}((n_{d-1} + n_d) \prod_{r=1}^{d-2} n_r)$, which is similar to Adafactor.

SM3 (Anil et al. 2019), unlike Adafactor and its variants such as CAME, applies the min-max scheme to approximate the similar elements of the 2^{nd} momentum to a smaller set of variables. It shows competitive optimization performance to Adam and Adafactor on Transformers that exhibit a grid pattern of similar elements in their weight matrices. Given a rank-d momentum tensor $\mathbb{R}^{n_1 \times \ldots \times n_d}$, the memory complexity of SM3 becomes $\mathcal{O}(\sum_{r=1}^{d} n_r)$ if similar elements appear on each axis of the weight tensor, which can be found in some Transformer weight matrices (Anil et al. 2019).

Although those existing memory-efficient optimizers effectively reduce the memory requirement and perform competitive optimization primarily on the Transformer architectures (Vaswani et al. 2017) by projecting the gradient onto rank-1 vectors, each optimizer has limitations. First, since Adafactor and CAME rely on matrix factorization (Finesso and Spreij 2006), a momentum tensor should be first sliced into multiple matrices before being factorized, degrading the memory reduction effect given a high-rank momentum tensor. Next, SM3 needs sets of similar elements in a momentum tensor to perform effective optimization, neither easy nor guaranteed to find in the huge weight parameter space of many deep neural networks. However, unlike Adafactor and CAME, the proposed SMMF applies one-time single matrix factorization to any rank (shape) of momentum tensors based on the proposed square-matricization without the memory increase caused by tensor-to-matrices slice. Also, since the proposed SMMF utilizes NNMF, it does not require strong patterns on the weight parameter space, readily applying to an arbitrary pattern of weight tensors, in contrast to SM3 that assumes the existence of specific element patterns on each axis in a weight tensor.

Algorithm 1: Overall SMMF applied to each layer. The elements of $\boldsymbol{r}$, $\boldsymbol{c}$, $\boldsymbol{M}$, $\boldsymbol{V}$, and $\boldsymbol{S}$ are initially set to zeros.

Input: Step t, total step T, model $f(\cdot)$ with rank-d weight tensor $\boldsymbol{W}_t \in \mathbb{R}^{n_1 \times \ldots \times n_d}$, learning-rate η_t, regularization constant ϵ, 1^{st} and 2^{nd} momentum hyper-parameters $\beta_{1,t}$ and $\beta_{2,t}$.
for $t = 1$ **to** T **do**
  $\boldsymbol{G}_t = \nabla f(\boldsymbol{W}_{t-1})$
  $\bar{\boldsymbol{G}}_t$ = Square-Matricization($\boldsymbol{G}_t$, $n_1 \ldots n_d$) [Algo 2]
  $\hat{\boldsymbol{M}}_{t-1}$ = Decompression($\boldsymbol{r}_{\boldsymbol{M}_{t-1}}, \boldsymbol{c}_{\boldsymbol{M}_{t-1}}, \boldsymbol{S}_{\boldsymbol{M}_{t-1}}$) [Algo 3]
  $\hat{\boldsymbol{V}}_{t-1}$ = Decompression($\boldsymbol{r}_{\boldsymbol{V}_{t-1}}, \boldsymbol{c}_{\boldsymbol{V}_{t-1}}, \boldsymbol{1}$) [Algo 3]
  $\boldsymbol{M}_t = \beta_{1,t} \hat{\boldsymbol{M}}_{t-1} + (1 - \beta_{1,t}) \bar{\boldsymbol{G}}_t$
  $\boldsymbol{V}_t = \beta_{2,t} \hat{\boldsymbol{V}}_{t-1} + (1 - \beta_{2,t}) \bar{\boldsymbol{G}}_t^2$
  ($\boldsymbol{r}_{\boldsymbol{M}_t}, \boldsymbol{c}_{\boldsymbol{M}_t}, \boldsymbol{S}_{\boldsymbol{M}_t}$) = Compression($\boldsymbol{M}_t$) [Algo 4]
  ($\boldsymbol{r}_{\boldsymbol{V}_t}, \boldsymbol{c}_{\boldsymbol{V}_t}, .$) = Compression($\boldsymbol{V}_t$) [Algo 4]
  $\boldsymbol{U}$ = Reshape($\boldsymbol{M}_t / \sqrt{\boldsymbol{V}_t + \epsilon}$, $n_1 \ldots n_d$)
  $\boldsymbol{W}_t = \boldsymbol{W}_{t-1} - \eta_t \boldsymbol{U}$
end for

3 SMMF (Square-Matricized Momentum Factorization)

Algorithm 1 shows the overall procedure of the proposed SMMF (Square-Matricized Momentum Factorization), applied to the weight tensor (momentum) at each layer of the model. In short, given the 1^{st} and 2^{nd} momentum tensors as $\boldsymbol{M}, \boldsymbol{V} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$, SMMF reduces the memory complexity required for optimization into $\mathcal{O}_M(\hat{n} + \hat{m})$ and $\mathcal{O}_V(\hat{n} + \hat{m})$ for $\boldsymbol{M}$ and $\boldsymbol{V}$, respectively, with $\hat{n}, \hat{m} =$

Algorithm 2: Square-Matricization. It needs to be calculated only once before starting model training (optimization).

Input: Rank-d tensor $\boldsymbol{G} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ and the length of each axis $n_1 \ldots n_d$
Output: Reshaped matrix $\bar{\boldsymbol{G}} \in \mathbb{R}^{\hat{n} \times \hat{m}}$
$N = \prod_{i=1}^{d} n_i$; $s = \lfloor \sqrt{N} \rfloor$
for $i = s$ **to** 1 **do**
  if (N mod i) == 0 **then**
    $\bar{\boldsymbol{G}}$ = Reshape($\boldsymbol{G}$, $\hat{n}$, $\hat{m}$) where $\hat{n} = N/i$ and $\hat{m} = i$; break
  end if
end for

Algorithm 3: Decompression.

Input: Factorized vectors $\boldsymbol{r} \in \mathbb{R}^{\hat{n} \times 1}$ and $\boldsymbol{c} \in \mathbb{R}^{1 \times \hat{m}}$, and a binary sign matrix $\boldsymbol{S} \in \{0,1\}^{\hat{n} \times \hat{m}}$. [$M_{i,j}$ is i^{th} row and j^{th} column element in a matrix $\boldsymbol{M} \in \mathbb{R}^{\hat{n} \times \hat{m}}$]
Output: Decompressed matrix $\boldsymbol{M} \in \mathbb{R}^{\hat{n} \times \hat{m}}$

$$\boldsymbol{M} = \boldsymbol{r} \otimes \boldsymbol{c} \quad [\otimes \text{ is the outer product operator}]$$
$$M_{i,j} = \begin{cases} M_{i,j}, & \text{if } S_{i,j} = 1 \\ -M_{i,j}, & \text{otherwise} \end{cases}$$

$\arg\min_{n,m}|n-m|$ such that $nm = \prod_{r=1}^{d} n_r$ where $n_r, n, m, \hat{n}$, and $\hat{m}$ are in $\mathbb{N}$. It first transforms $\boldsymbol{M}, \boldsymbol{V} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ into a matrix closest to the square (square-matricization), i.e., $\boldsymbol{M}, \boldsymbol{V} \in \mathbb{R}^{\hat{n} \times \hat{m}}$ where $\hat{n} \simeq \hat{m}$, and then applies NNMF (Algorithm 5) to $\boldsymbol{M}, \boldsymbol{V} \in \mathbb{R}^{\hat{n} \times \hat{m}}$ as one-time single matrix factorization (compression). Since the 1^{st} momentum $\boldsymbol{M}$ can be negative, unlike the 2^{nd} momentum $\boldsymbol{V}$ that is non-negative, we apply NNMF to the absolute values of $\boldsymbol{M}$ and store the sign of each element of $\boldsymbol{M}$ as a separate set of binary values (1-bit). Although it incurs extra memory overhead $\mathcal{O}_M(\hat{n}\hat{m})$ on top of $\mathcal{O}_M(\hat{n}+\hat{m})$ and $\mathcal{O}_V(\hat{n}+\hat{m})$, its memory footprint is 32 times smaller than storing the original $\boldsymbol{M}$ with the 32-bit floating-point format. The following subsections describe each step of SMMF.

3.1 Square-Matricization

In Algorithm 1, SMMF first obtains a rank-d gradient $\boldsymbol{G}_t \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ for the weight and bias tensor at each layer of the model and converts it into a matrix closest to a square matrix $\bar{\boldsymbol{G}}_t \in \mathbb{R}^{\hat{n} \times \hat{m}}$ where $\hat{n} \simeq \hat{m}$, for factorization, naturally leading to the square-matricization of the 1^{st} and 2^{nd} momentum, $\boldsymbol{M}$ and $\boldsymbol{V}$. To this end, we propose a square-matricization method that reshapes $\boldsymbol{G}_t \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ into a matrix closest to a square matrix $\bar{\boldsymbol{G}}_t \in \mathbb{R}^{\hat{n} \times \hat{m}}$ such that $nm = \prod_{r=1}^{d} n_r$ and $(\hat{n}, \hat{m}) = \arg\min_{n,m}(n+m) = \arg\min_{n,m}|n-m|$, where $\hat{n}, \hat{m}, n, m \in \mathbb{N}$. Following theorems show that the square-matricization of $\bar{\boldsymbol{G}}_t \in \mathbb{R}^{\hat{n} \times \hat{m}}$, i.e., having $\hat{n} \simeq \hat{m}$, also minimizes $\hat{n} + \hat{m}$.

Theorem 3.1. *Given $n_r \in \mathbb{N}$, $r \in [1, d]$, and a constant $N = \prod_{r=1}^{d} n_r$, then $\prod_{r=1}^{d-2} n_r(n_{d-1} + n_d)$ decreases if both n_{d-1} and n_d increase (Proof provided in Appendix C).*

Corollary 3.1.1. *Given $N = \prod_{r=1}^{d} n_r$, there exist $N = \hat{n}\hat{m}$ such that $\hat{n} + \hat{m} = \min \prod_{r=1}^{d-2} n_r(n_{d-1} + n_d)$, $(\hat{n}, \hat{m}) \in \mathbb{N}$.*

Theorem 3.2. *Given $n_r, n, m \in \mathbb{N}$, $r \in [1, d]$, $N = \prod_{r=1}^{d} n_r = nm$, and $n \leq m$, then $\hat{n}, \hat{m} = \arg\min_{n,m}(n+m) = \arg\min_{n,m}|n-m|$ (Proof provided in Appendix D).*

From Corollary 3.1.1, square-matricizing $\boldsymbol{G}_t \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ into $\bar{\boldsymbol{G}}_t \in \mathbb{R}^{\hat{n} \times \hat{m}}$ reduces the memory complexity since $\prod_{r=1}^{d-2} n_r(n_{d-1} + n_d) \leq \prod_{r=1}^{d} n_r$. Also, based on Theorem 3.2, minimizing $|n-m|$ is equivalent to minimizing $n+m$. From this, we derive the square-matricization algorithm (Algorithm 2) that finds $\hat{n}$ and $\hat{m}$, which minimizes $\hat{n} + \hat{m}$ by solving $(\hat{n}, \hat{m}) = \arg\min_{n,m}|n-m|$.

By reshaping a rank-d gradient into a matrix closest to a square matrix through square-matricization, it becomes able to perform one-time single matrix factorization, which minimizes the memory complexity of $\mathcal{O}_{M,V}(\hat{n}+\hat{m})$ for the 1^{st} and 2^{nd} momentum $\boldsymbol{M}$ and $\boldsymbol{V}$. Thus, the memory usage of SMMF becomes smaller than those of existing memory-efficient optimizers (Shazeer and Stern 2018; Luo et al. 2023) that should slice a high-rank tensor into a bunch of matrices for multiple factorizations, i.e., $\mathcal{O}(\prod_{r=1}^{d-2} n_r(n_{d-1} + n_d))$ given $\boldsymbol{V} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$. That is, the memory complexity of CNNs having high-rank gradient tensors grows over the rank of gradients (Shazeer and Stern 2018; Luo et al. 2023), whereas that of SMMF does not.

3.2 Decompression and Compression

Decompression → Compression. After square-matricizing the gradient, SMMF decompresses the 1^{st} and 2^{nd} momentum from two vectors factorized at the previous step $t-1$ to update the momentums, as in Algorithm 1. Then, it compresses the momentums obtained at step t into vectors and updates the weight $\boldsymbol{W}$ using the decompressed momentums. We call this process the $decompression \rightarrow compression$ scheme, in which the gradient $\bar{\boldsymbol{G}}_t$ at the current step t is reflected to the 1^{st} and 2^{nd} momentum before it is factorized, enabling the precise update of the weight.

The significance of information in the current gradient, e.g., tensor patterns, has been emphasized in some previous works (Anil et al. 2019). Since reshaping and factorization of the gradient may ruin some potentially important patterns of the momentums contained in the previous gradient $\boldsymbol{G}_{\tau<t}$, reflecting the current gradient $\boldsymbol{G}_t$ (and its pattern) is crucial in model performance. Thus, by performing decompression with $\boldsymbol{G}_t$ prior to updating and compressing (factorizing) $\boldsymbol{M}_t$ and $\boldsymbol{V}_t$, it is expected to improve the optimization performance. On the contrary, existing optimizers, such as Adafactor, first compress $\boldsymbol{G}_t$ and then updates momentums through decompression, which we call the $compression \rightarrow decompression$ scheme. In this scheme, some useful information of $\boldsymbol{G}_t$ would be lost by compression (factorization), implying that an optimizer can hardly utilize the intact state of $\boldsymbol{G}_t$, likely to degrade the model performance.

Decompression. First, in the decompression phase (Algorithm 3), the 1^{st} and 2^{nd} momentum, $\{|\hat{\boldsymbol{M}}_{t-1}|, \hat{\boldsymbol{V}}_{t-1}\} \in \mathbb{R}^{\hat{n} \times \hat{m}}$, are defactorized from the two vectors for each, i.e., $\{\boldsymbol{r}_{\boldsymbol{M}_{t-1}}, \boldsymbol{c}_{\boldsymbol{M}_{t-1}}\}$ for $|\hat{\boldsymbol{M}}_{t-1}|$ and $\{\boldsymbol{r}_{\boldsymbol{V}_{t-1}}, \boldsymbol{c}_{\boldsymbol{V}_{t-1}}\}$ for $\hat{\boldsymbol{V}}_{t-1}$, which have been factorized from the square-matricized momentums at the previous step $t-1$, by per-

Algorithm 4: Compression. $\mathbf{1}_d$ is a vector that all elements are one and its length is d.

Input: Matrix $\boldsymbol{M} \in \mathbb{R}^{\hat{n} \times \hat{m}}$ to be factorized
Output: Factorized vectors $\boldsymbol{r} \in \mathbb{R}^{\hat{n} \times 1}$ and $\boldsymbol{c} \in \mathbb{R}^{1 \times \hat{m}}$, and binary sign matrix $\boldsymbol{S} \in \{0,1\}^{\hat{n} \times \hat{m}}$

$$S_{i,j} = \begin{cases} 1, & if M_{i,j} \geq 0 \\ 0, & otherwise \end{cases}, \quad (\boldsymbol{r}, \boldsymbol{c}) = (|\boldsymbol{M}|\mathbf{1}_{\hat{m}}, \mathbf{1}_{\hat{n}}^{\top}|\boldsymbol{M}|)$$

$$\boldsymbol{r} = \begin{cases} \boldsymbol{r}/(\mathbf{1}_{\hat{n}}^{\top}\boldsymbol{r}), & \text{if } \hat{n} \leq \hat{m} \\ \boldsymbol{r}, & \text{otherwise} \end{cases}, \quad \boldsymbol{c} = \begin{cases} \boldsymbol{c}/(\boldsymbol{c}\mathbf{1}_{\hat{m}}), & \text{if } \hat{n} > \hat{m} \\ \boldsymbol{c}, & \text{otherwise} \end{cases}$$

forming outer product between them. To apply NNMF to the 1^{st} momentum $\hat{\boldsymbol{M}}_t$, its sign values are stored as a binary matrix $\boldsymbol{S}_{\boldsymbol{M}_{t-1}} \in \{0,1\}^{\hat{n} \times \hat{m}}$ in the compression phase and restored back to the defactorized 1^{st} momentum $\hat{\boldsymbol{M}}_t$ in an element-wise manner. Then, $\boldsymbol{M}_t$ and $\boldsymbol{V}_t$ are updated by using two coefficients $\beta_{1,t}$ and $\beta_{2,t}$.

Compression. Next, in the compression phase (Algorithm 4), the sign values of $\boldsymbol{M}_t$ is stored as a binary sign matrix $\boldsymbol{S}_{\boldsymbol{M}_t}$ for the next step of optimization, and both $|\boldsymbol{M}_t|$ and $\boldsymbol{V}_t$ are factorized into two vectors for each, i.e., $\{\boldsymbol{r}_{M_t}, \boldsymbol{c}_{M_t}\}$ and $\{\boldsymbol{r}_{V_t}, \boldsymbol{c}_{V_t}\}$, respectively. To reduce the computation required for compression, it determines whether to normalize $\boldsymbol{r}$ or $\boldsymbol{c}$ based on the shape of the matrix.

Weight Update. Lastly, the weight update term $\boldsymbol{U} \in \mathbb{R}^{\hat{n} \times \hat{m}}$ is computed as $\boldsymbol{M}_t/\sqrt{\boldsymbol{V}_t + \epsilon}$ and reshaped back into the original dimension of the gradient $\boldsymbol{G}_t \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ to update the weight $\boldsymbol{W}_t$.

3.3 Time (Computation) Complexity of SMMF

The time complexity of SMMF consists of two parts, i.e., square-matricization and decompression/compression. First, computing $\hat{n}$ and $\hat{m}$ for square-matricization (Algorithm 2) is $\mathcal{O}(\sqrt{N})$, where N is the number of elements in the momentum tensor. However, this computational overhead is negligible since $\hat{n}$ and $\hat{m}$ are calculated only once before starting model training (optimization). Next, the time complexity of decompression (Algorithm 3) and compression (Algorithm 4) are both $\mathcal{O}(N)$, which is asymptotically equivalent to existing memory-efficient optimizers, i.e., Adafactor and CAME. While taking a similar computational complexity to existing memory-efficient optimizers (Shazeer and Stern 2018; Luo et al. 2023), SMMF is able to save up to 96% of memory, as shown in Section 5.

4 Regret Bound Analysis

We analyze the convergence of SMMF by deriving the upper bound of the regret that indicates an optimizer's convergence (Kingma and Ba 2014). The regret $R(T)$ is defined as the sum of differences between two convex functions $f_t(\boldsymbol{w}_t)$ and $f_t(\boldsymbol{w}^*)$ for all $t \in [1, T]$, where $\boldsymbol{w}^*$ is an optimal point.

$$R(T) = \sum_{t=1}^{T} (f_t(\boldsymbol{w}_t) - f_t(\boldsymbol{w}^*)) \tag{1}$$

Since SMMF factorizes (compresses) the momentum tensors, unlike Adam, we introduce some compression error terms in our analysis as follows. First, $\hat{\boldsymbol{m}}_t$ and $\hat{\boldsymbol{v}}_t$ are the decompressed and vectorized vectors of the 1^{st} and 2^{nd} momentum $\boldsymbol{M}_t$ and $\boldsymbol{V}_t$, containing $\boldsymbol{e}_{m,t}$ and $\boldsymbol{e}_{v,t}$, which denote compression errors of $\boldsymbol{M}_t$ and $\boldsymbol{V}_t$, respectively, i.e., $\boldsymbol{e}_{m,t} = \hat{\boldsymbol{m}}_t - \boldsymbol{m}_t$ and $\boldsymbol{e}_{v,t} = \hat{\boldsymbol{v}}_t - \boldsymbol{v}_t$. Similarly, we also define $\tilde{\boldsymbol{e}}_{m,t}$, $\tilde{\boldsymbol{e}}_{v,t}$, $\tilde{\boldsymbol{g}}_{m,t}$, $\tilde{\boldsymbol{g}}_{v,t}$, and $\hat{\boldsymbol{g}}_{m,t}$. The detailed definitions are given in Lemmas E.3 and E.4 in Appendix E.

Theorem 4.1. *Let $\boldsymbol{w}_t$ and $\boldsymbol{v}_t$ be the vectorized $\boldsymbol{W}_t$ and $\boldsymbol{V}_t$, respectively, in Algorithm 1, and $\eta_t = \eta/\sqrt{t}$, $\beta_1 = \beta_{1,1}$, $\beta_2 = \beta_{2,1}$, $\beta_{1,t} \leq \beta_1$ for all $t \in [1, T]$, and $\zeta_1 > 0$. We assume that $||\boldsymbol{w}_t - \boldsymbol{w}^*||_2 \leq D$, $||\boldsymbol{w}_k - \boldsymbol{w}_l||_\infty \leq D_\infty$, and all $\boldsymbol{w}$ is in $\mathcal{F}$ where D is the diameter of the feasible space $\mathcal{F}$. Furthermore, let $\beta_{1,t}$ and $\beta_{2,t}$ follow the following conditions, where the conditions (a) and (b) are from (Reddi, Kale, and Kumar 2019).*

$$(a)\ \eta_{t-1}\sqrt{\boldsymbol{v}_{t,i}} \geq \eta_t \sqrt{\boldsymbol{v}_{t-1,i}}$$

$$(b)\ \frac{1}{\eta_t}\sqrt{\sum_{j=1}^{t}\prod_{k=1}^{t-j}\beta_{2,t-k+1}(1-\beta_{2,j}\tilde{\boldsymbol{g}}_{v,j,i})} \geq \frac{1}{\zeta_2}\sqrt{\sum_{j=1}^{t}\tilde{\boldsymbol{g}}_{v,j,i}}$$
$$for\ some\ \zeta_2 > 0\ and\ all\ t \in [:T]$$

Given the above conditions, the upper bound of the regret $R(T)$ on a convex function becomes:

$$R(T) \leq \sum_{i=1}^{d}\frac{D^2\sqrt{\boldsymbol{v}_{T,i}}}{2\eta_T(1-\beta_1)} + \sum_{t=1}^{T}\sum_{i=1}^{d}\frac{\beta_{1,t}D_\infty^2\sqrt{\boldsymbol{v}_{t,i}}}{2\eta_t(1-\beta_{1,t})} + \frac{\zeta_1\zeta_2(1+\beta_1)\sqrt{d}}{(1-\beta_1)^3}\sqrt{\sum_{i=1}^{d}||\boldsymbol{g}_{1:T,i}||_2}\,. \tag{2}$$

The full proof of Theorem 4.1 is provided in Appendix E. It shows that SMMF has a similar regret bound ratio to AdamNC (Reddi, Kale, and Kumar 2019) (a variant of Adam), i.e., $\mathcal{O}(\sqrt{T})$ where T is the total optimization steps, under the conditions (a) and (b). It allows SMMF to perform consistent optimization comparable to Adam-based and existing memory-efficient optimizers, as shown in the experiment (Section 5). The conditions (a) and (b) can be satisfied by properly scheduling $\beta_{2,t}$ (Reddi, Kale, and Kumar 2019).

5 Experiment

We implement the proposed SMMF using PyTorch (Paszke et al. 2017), which is available both on an GitHub and in Appendix M. For evaluation, we compare the memory usage and optimization performance of SMMF against four (memory-efficient) optimizers, i.e., Adam, Adafactor, SM3, and CAME, with two types of deep learning tasks: 1) CNN models for image tasks and 2) Transformer models for NLP tasks. The detailed experimental setups and training configurations are provided in Appendix L.

CNN-based Models and Tasks. We apply the five optimizers, including SMMF, to two representative image tasks, i.e., image classification and object detection, and evaluate them by 1) training ResNet-50 (He et al. 2016) and MobileNetV2 (Dong et al. 2020) on CIFAR100 (Krizhevsky, Hinton et al. 2009) and ImageNet (Russakovsky et al. 2015), and 2) training YOLOv5s and YOLOv5m (Ultralytics 2021) on COCO (Lin et al. 2015).

Transformer-based Models and Tasks. For NLP tasks, we train several Transformer-based models from small scale to large scale with three training methods, i.e., full-training,

CNN Models and Tasks (Optimizer and End-to-End Memory [MiB]), Model Performance					
Dataset	(1) CIFAR100 (**Image Classification**)				
Model	**Adam**	**Adafactor**	**SM3**	**CAME**	**SMMF**
MobileNet (V2)	(18, 36) 73.6	(26, 43) 69.4	(9, 27) 70.0	(43, 60) 66.2	(**0.7**, **19**) 74.1
ResNet (50)	(184, 366) 72.4	(215, 397) 72.9	(93, 227) 73.8	(340, 526) 67.2	(**3.5**, **185**) 74.5
Dataset	(2) ImageNet (**Image Classification**)				
MobileNet (V2)	(27, 54) 68.6	(30, 58) 69.3	(14, 41) 67.6	(47, 75) 69.5	(**0.8**, **28**) 69.5
ResNet (50)	(195, 394) 73.7	(220, 419) 69.5	(99, 298) 75.8	(346, 546) 72.3	(**3.7**, **197**) 73.7
Dataset	(3) COCO (**Object Detection**)				
YOLO (v5s)	(57, 121) 52.7	(61, 92) 53.6	(28, 92) 50.3	(94, 159) 53.3	(**1.4**, **65**) 54.1
YOLO (v5m)	(168, 340) 58.0	(174, 258) 58.8	(84, 260) 57.0	(267, 438) 59.3	(**3.4**, **176**) 59.6

Table 1: **(First and second tables) Image classification**: the optimizer memory usage (MiB) including the binary sign matrix $\boldsymbol{S}_M$, end-to-end training (one-batch) memory usage (MiB) at 100 iterations, and top-1 validation accuracy of MobileNetV2 and ResNet-50 on CIFAR100 and ImageNet. **(Third table) Object detection**: the optimizer memory usage (MiB) including $\boldsymbol{S}_M$, end-to-end training (one-batch) memory usage (MiB) at 100 iterations, and validation mAP50 of YOLO (v5s and v5m) on COCO.

pre-training, and fine-tuning. We 1) full-train Transformer-base and big models (Vaswani et al. 2017) on WMT32k (Bojar et al. 2014), 2) pre-train BERT (Devlin et al. 2018), GPT-2 (Radford et al. 2019), and T5 (Raffel et al. 2020) on BookCorpus (Zhu et al. 2015) & Wikipedia, and 3) fine-tune BERT, GPT-2, T5-small, and LLaMA-7b (Touvron et al. 2023a) on QNLI, MNLI, QQP, STSB, and MRPC (Wang et al. 2018) datasets. To fine-tune LLaMA-7b, we use LoRA (Hu et al. 2021). Additionally, we train various Transformer models on other NLP tasks such as question-answering (Rajpurkar et al. 2016), translation (Bojar et al. 2016), and summarization (Narayan, Cohen, and Lapata 2018). Due to the page limit, we provide the detailed experiment results in Appendix K.

Optimization Time Measurement. We measure the optimization time of five optimizers with the CNN and Transformer models, i.e., 1) MobileNetV2 and ResNet-50 on ImageNet, 2) Transformer-base and big on WMT32K.

5.1 CNN-based Models and Tasks

Image Classification. The first and second tables of Table 1 summarize the optimizer memory usage, end-to-end training (one-batch) memory usage, and top-1 classification accuracy of MobileNetV2 and ResNet-50 on CIFAR100 and ImageNet, respectively, showing that SMMF achieves comparable accuracy among the five optimizers with the lowest memory footprint. For instance, for ResNet-50 on ImageNet, SMMF substantially reduces the optimizer memory usage from 220 to 3.7 MiB (59x smaller) compared to Adafactor while achieving a higher accuracy (73.7%). Its memory usage is also smaller than the other two memory-efficient optimizers, i.e., SM3 (99 vs. 3.7 MiB) and CAME (346 vs. 3.7 MiB). On the other hand, both Adafactor and CAME take more memory than not only SMMF but also Adam due to the overhead of slicing the momentum tensor into multiple matrices for factorization based on the last two rank of the tensor corresponding to the size of a CNN kernel (i.e., $C_k^i \times C_k^o \times H_k \times W_k$). Since H_k and W_k are usually small, e.g., $H_k = W_k = 1$, 3 or 5, whereas C_k^i and C_k^o are large, e.g., 512, the memory reduction effect of both Adafactor and CAME becomes marginal in CNNs. Figure 1 (left) plots the top-1 validation accuracy of MobileNetV2 on ImageNet over training steps.

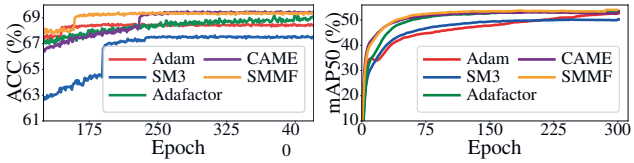

Figure 1: **(Left)** The validation top-1 accuracy of MobileNetV2 on ImageNet. **(Right)** The validation mAP50 of YOLOv5s on COCO of the five optimizers.

Object Detection. The third table of Table 1 summarizes the optimizer memory usage, end-to-end training (one-batch) memory usage, and the model performance metric (i.e., mAP50) of YOLOv5s and YOLOv5m on the COCO object detection task. Figure 1 (right) shows the mAP50 of YOLOv5s on COCO over training steps. Similar to image classification tasks, SMMF achieves comparable mAP50, e.g., 59.6 in YOLOv5m and 54.1 in YOLOv5s, over all the four optimizers with the lowest memory up to 78x reduction, e.g., 267 vs. 3.4 MiB. This result implies that YOLOv5s (65 MiB) can be trained on an off-the-shelf memory-constrained device, e.g., Coral Dev Micro (64 MiB) (Google 2022), along with other memory-efficient training methods such as gradient checkpointing (Chen et al. 2016).

These experiment results show that SMMF performs consistent and reliable optimization for both image classification and object detection tasks with different CNN models, taking the smallest memory compared to existing memory-efficient optimizers, i.e., Adafactor, SM3, and CAME.

5.2 Transformer-based Models and Tasks

Full-training. As shown in Table 2, SMMF achieves comparable perplexity with up to 70x smaller optimizer memory when full-training (i.e., training models from scratch) both the Transformer-base and big models. Since SMMF square-matricizes both the 1^{st} and 2^{nd} momentums and factorizes them, its memory usage is at least half lower than the other memory-efficient optimizers, i.e., Adafactor, SM3, and CAME. Given that most Transformer architectures consist of two-dimensional matrices, e.g., attention and linear layers, the memory reduction effect of SM3 that is good at compressing a high-rank tensor, becomes insignificant, making its memory usage similar to Adafactor and CAME. On the other hand, SMMF can effectively reduce memory required to factorize a two-dimensional matrix with square-matricization, e.g., saving 69% of memory

Transformer Models and Tasks (Optimizer and End-to-End Memory [GiB]), Model Performance					
Dataset	WMT32k (**Full-Trining**)				
Model	**Adam**	**Adafactor**	**SM3**	**CAME**	**SMMF**
Transformer (base)	(0.7, 1.4) 6.6	(0.4, 1.1) 6.6	(0.4, 1.1) 7.8	(0.4, 1.1) 6.6	(**.01**, **0.8**) 6.7
Transformer (big)	(2.1, 4.2) 6.9	(1.1, 3.2) 6.6	(1.1, 3.2) 7.9	(1.1, 3.2) 7.5	(**.04**, **2.1**) 6.8
Dataset	Book Corpus & Wikipedia (**Pre-Training**)				
BERT	(2.5, 6.3) 16.1	(1.3, 5.0) 30.6	(1.3, 5.0) 27.5	(1.3, 5.0) 20.1	(**.04**, **3.8**) 20.4
GPT-2	(2.6, 6.7) 19.2	(1.3, 5.3) NaN	(1.3, 5.3) 19.4	(1.3, 5.3) 19.1	(**.04**, **4.0**) 19.2
T5	(1.7, 4.2) 2.6	(0.8, 3.4) 2.6	(0.8, 3.4) 2.8	(0.9, 3.4) 2.6	(**.03**, **2.5**) 2.6

Table 2: **Full-training**: the optimizer memory usage (GiB), including the $\boldsymbol{S}_M$, end-to-end training (one-batch) memory usage (GiB) at 100 iterations, and test perplexity of the Transformer-base and big models on WMT32k. **Pre-training**: the optimizer memory usage (GiB), including the $\boldsymbol{S}_M$, end-to-end training (one-batch) memory usage (GiB) at 100 iterations, and test perplexity of BERT, GPT-2, and T5. We use Adam without the bias correction term to obtain lower perplexity.

for the embedding weight matrix of BERT, as the weight matrix in $\mathbb{R}^{30522\times768}$ becomes $\mathbb{R}^{5087\times4608}$. Although the square-matricization may spoil the gradient pattern of the Transformer's weight parameters (Anil et al. 2019), SMMF performs comparable optimization (e.g., 6.7 perplexity for WMT32k) using much less memory by fully utilizing the intact latest gradient before it is compressed. It is possible by taking the proposed $decompression \rightarrow compression$ scheme that first reflects the intact gradient pattern, if any, to the momentum and then performs factorization, unlike existing optimizers, e.g., Adafactor, SM3, and CAME, which apply the $compression \rightarrow decompression$ scheme. Figure 2 (left) shows the test perplexity of the Transformer-base model full-trained on WMT32k from scratch.

Pre-training. Table 2 shows the optimizer memory usage including $\boldsymbol{S}_M$, end-to-end training (one-batch) memory usage, and the perplexity of BERT, GPT-2, and T5 pre-trained for the BookCorpus & Wikipedia dataset. For GPT-2 pre-trained with Adafactor, we failed to obtain its perplexity since it diverged (NaN) even with multiple trials of various settings, e.g., different machines, hyper parameters, seeds, etc. On the other hand, SMMF performs competitive optimization for all pre-training of BERT, GPT-2, and T5 using up to 60x lower optimizer memory. Figure 2 (right) shows that SMMF (yellow) curtails the optimizer memory usage in the pre-training of BERT about 1.26 GiB (from 1.3 to 0.04 GiB) while exhibiting the similar test perplexity trajectory to CAME (purple) taking much more memory to maintain the similar optimization performance. Overall, Adafactor, CAME, and SMMF show similar perplexity trajectories, confirming that SMMF retains the optimization performance in pre-training of Transformers with the lowest memory, enabled by 1) square-matricization of any rank (shape)

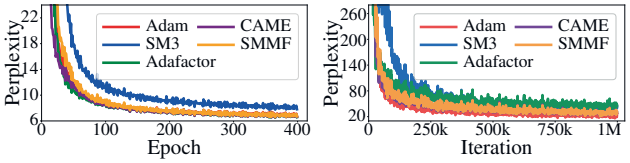

Figure 2: The test perplexity of the Transformer-base model on WMT32k during full-training steps **(Left)** and BERT on BookCorpus & Wikipedia during pre-training steps **(Right)**

of momentums (e.g., a vector, matrix, tensor, etc.) and 2) the $decompression \rightarrow compression$ scheme that uses the uncompressed gradients at the current update step. They jointly allow it to pre-train Transformers aptly during a massive number of pre-training steps by persistently conducting effective and efficient optimization over a long time horizon.

That being said, SMMF also exhibits some occasional loss spike (Takase et al. 2024) at the early steps of optimization (training), which stabilizes as the training proceeds. It is a well-known phenomenon that commonly occurs in the pre-training of many large language models (Raffel et al. 2020; Radford et al. 2019) optimized with existing optimizers, such as Adam, Adafactor, SM3, and CAME. We discuss it in more detail in Section 6.

Fine-tuning. Table 3 summarizes the optimization memory usage including $\boldsymbol{S}_M$, end-to-end training(one-batch) memory usage, and the model performance of GPT-2, T5-small, and LLaMA-7b, which are fine-tuned for the QNLI, MNLI, QQP, STSB, and MRPC datasets from pre-trained models. As shown in the table, SMMF achieves comparable model performance in the five datasets compared to the other four optimizers with the lowest memory usage. For instance, SMMF provides similar accuracy (90.6%) for GPT-2 on QQP compared to CAME, using much smaller optimizer and end-to-end training memory, respectively (16 vs. 489 MiB and 0.96 vs. 1.43 GiB). It demonstrates that SMMF is also apt at fine-tuning Transformer models, which entails delicate and intricate updates of weight parameters, otherwise likely to degrade the learning performance for downstream tasks. In practice, it suggests that some Transformer models, such as T5, would be fine-tuned on a low-end device, e.g., Raspberry Pi Zero (0.5 GiB), with similar model performance to CAME (i.e., 83.0% vs. 82.8%), as SMMF curtails the end-to-end training memory requirement of fine-tuning down to 0.47 GiB. On the other way around, SMMF can scale up training of Transformers by enabling memory-efficient optimization of enormous Transformer models that require a gigantic amount of memory, e.g., hundreds of GiB.

Similar to the CNN-based models and tasks, the experimental results of Transformers demonstrate that SMMF steadily performs competitive optimization for a wide range of Transformer models, tasks, and training methods (i.e., full-, pre-, and fine-tuning) with the smallest memory usage.

5.3 Optimization Time Measurement

Table 4 shows the optimization time measured for a single training step of the five optimizers when training MobileNetV2 and ResNet-50 on ImageNet, and Transformer-

Transformer Models and Tasks (Fine-Tuning) (Optimizer Memory [MiB] and End-to-End Memory [GiB]), Model Performance						
Optimizer	**Model**	**QNLI (ACC)**	**MNLI (ACC)**	**QQP (ACC)**	**STSB (Pearson)**	**MRPC (ACC)**
Adam	GPT-2	(957, 1.89) 84.5	(952, 1.89) 72.4	(973, 1.89) 86.4	(962, 1.89) 83.2	(861, 1.89) 81.4
Adafactor		(478, 1.43) 74.7	(478, 1.43) 71.7	(488, 1.43) 80.1	(481, 1.43) 84.4	(481, 1.43) 82.6
SM3		(478, 1.43) 88.0	(478, 1.43) 81.1	(487, 1.43) 88.8	(481, 1.43) 84.1	(481, 1.43) 83.3
CAME		(468, 1.43) 88.6	(479, 1.43) 81.9	(489, 1.43) 90.6	(481, 1.43) 86.4	(478, 1.43) 83.3
SMMF		(**16**, **0.96**) 88.9	(**16**, **0.96**) 82.2	(**16**, **0.96**) 90.6	(**16**, **0.96**) 83.8	(**16**, **0.96**) 81.6
Adam	T5 (small)	(464, 0.92) 88.4	(464, 0.92) 77.5	(434, 0.92) 86.8	(456, 0.92) 84.7	(464, 0.92) 77.5
Adafactor		(233, 0.70) 90.1	(233, 0.70) 80.3	(233, 0.70) 88.7	(233, 0.70) 87.9	(233, 0.70) 80.3
SM3		(233, 0.70) 88.8	(233, 0.70) 79.4	(233, 0.70) 88.1	(233, 0.70) 79.4	(233, 0.70) 79.4
CAME		(233, 0.70) 90.7	(234, 0.70) 83.0	(233, 0.70) 90.4	(233, 0.70) 87.5	(234, 0.70) 83.0
SMMF		(**8**, **0.47**) 90.6	(**8**, **0.47**) 82.8	(**8**, **0.47**) 90.2	(**8**, **0.47**) 84.7	(**8**, **0.47**) 82.8
Adam	LLaMA-7b	(153, 24.9) 93.0	(153, 24.9) 87.5	(153, 24.9) 84.4	(153, 24.9) 96.6	(153, 24.9) 90.6
Adafactor		(86, 24.9) 93.8	(86, 24.9) 84.4	(86, 24.9) 93.0	(86, 24.9) 96.3	(86, 24.9) 85.9
SM3		(86, 24.9) 65.6	(86, 24.9) 64.8	(86, 24.9) 71.9	(86, 24.9) 34.9	(86, 24.9) 70.3
CAME		(86, 24.9) 69.5	(86, 24.9) 43.0	(86, 24.9) 75.8	(86, 24.9) 34.8	(86, 24.9) 70.3
SMMF		(**3.9**, **24.8**) 91.4	(**3.9**, **24.8**) 87.5	(**3.9**, **24.8**) 90.6	(**3.9**, **24.8**) 96.5	(**3.9**, **24.8**) 89.8

Table 3: **Fine-tuning**: the optimizer and end-to-end training (one-batch) memory usage [MiB, GiB] at 100 iterations including $\boldsymbol{S_M}$, and the performance of GPT-2, T5-small, and LLaMA-7b fine-tuned on QNLI, MNLI, QQP, STSB, and MRPC.

Optimization Time (ms) for a Single Training Step (Iteration)					
Model	**Adam**	**Adafactor**	**SM3**	**CAME**	**SMMF**
MobileNetV2	127±16	168±16	140±17	160±17	205± 13
ResNet-50	273±16	316±14	286±14	307±20	349± 15
Transformer-base	129± 7	160± 7	134± 7	156± 7	171± 7
Transformer-big	321±18	372±18	325±18	369±18	389± 18

Table 4: The single optimization time (milliseconds) per step of the four optimizers, and SMMF (8-bit format $\boldsymbol{S_M}$ in Algorithm 3 and 4): 1) MobileNetV2 and ResNet-50 on ImageNet, and 2) Transformer-base and big on WMT32k.

base and big on WMT32k. Except for Adam, the four optimizers take similar optimization times, where SMMF takes a little more time than the other three. That is because it trades off the memory space and optimization time, i.e., the time required for square-matricization and the sign matrix operations for the 1^{st} momentum (Algorithms 3 and 4). However, SMMF offers huge memory reduction compared to the amount of the increased optimization time, e.g., 7.9x memory reduction vs. 1.2x time increase for Adam and SMMF with the 8-bit format $\boldsymbol{S_M}$ in Algorithms 3 and 4 applied to Transformer-big on WMT32k, and 7.8x vs. 1.6x for Adam and SMMF with $\boldsymbol{S_M}$ applied to ResNet-50 on ImageNet.

6 Limitations and Discussions

Overhead of Binary Signs. It is not easy to effectively factorize the 1^{st} momentum $\boldsymbol{M}$ having negative elements. While SMMF circumvents this by storing the binary sign matrix (1-bit) that is much smaller than the original matrix (32-bit), there are some methods that can further reduce the memory required to store the sign matrix. For instance, Binary Matrix Factorization (Kumar et al. 2019) can be employed to factorize the binary matrix into two lower-rank matrices, reducing the memory space for storing the binary matrix with a high restoration rate.

Optimization Time. Section 3.3 finds that the time complexity of SMMF is similar to existing memory-efficient optimizers, e.g., Adafactor, showing that SMMF effectively saves a significant amount of memory (up to 96%) compared to them with slightly increased optimization time.

Loss Spike at Initial Training Steps. We observe some loss spike at the initial training steps, especially in Transformer models, which is a well-known issue commonly observed in other works (Takase et al. 2024) and many optimizers, i.e., Adam (with the bias correction term), Adafactor, SM3, and CAME. With appropriate hyper-parameter tuning, e.g., learning rate and weight-decay, it can be stabilized as the training proceeds, like other optimizers.

Extremely Large Models and Other Tasks. Due to the limited computing resources, we have not been able to experiment SMMF with extremely large models, e.g, GPT-4 (OpenAI et al. 2024), LLaMA-2 70B (Touvron et al. 2023b), and diffusion models (Rombach et al. 2022). From our theoretical study and empirical result, we expect SMMF to perform competitive optimization with them. We hope to have a chance to test SMMF with them.

7 Conclusion

We introduce SMMF, a memory-efficient optimizer that decreases the memory requirement of adaptive learning rate optimization methods, e.g., Adam and Adafactor. Given arbitrary-rank momentum tensors, SMMF reduces the optimization memory usage through the proposed square-matricization and matrix compression of the first and second momentums. The empirical evaluation shows that SMMF reduces up to 96% of optimization memory when compared to existing memory-efficient optimizers, i.e., Adafactor, SM3, and CAME, with competitive optimization performance on various models (i.e., CNNs and Transformers) and tasks (i.e., image classification, object detection, and NLP tasks over full-training, pre-training, and fine-tuning). Our analysis of regret bound proves that SMMF converges in a convex function with a similar bound of AdamNC.

Acknowledgements

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea goverment(MSIT) (No.RS-2024-00508465) and Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2020-II201336, Artificial Intelligence Graduate School Program(UNIST)).

References

Abdulkadirov, R.; Lyakhov, P.; and Nagornov, N. 2023. Survey of Optimization Algorithms in Modern Neural Networks. *Mathematics*, 11(11): 2466.

Amari, S.-i. 2010. Information geometry in optimization, machine learning and statistical inference. *Frontiers of Electrical and Electronic Engineering in China*, 5: 241–260.

Anil, R.; Gupta, V.; Koren, T.; and Singer, Y. 2019. Memory Efficient Adaptive Optimization. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Bojar, O.; Buck, C.; Federmann, C.; Haddow, B.; Koehn, P.; Leveling, J.; Monz, C.; Pecina, P.; Post, M.; Saint-Amand, H.; et al. 2014. Findings of the 2014 workshop on statistical machine translation. In *Proceedings of the ninth workshop on statistical machine translation*, 12–58.

Bojar, O. r.; Chatterjee, R.; Federmann, C.; Graham, Y.; Haddow, B.; Huck, M.; Jimeno Yepes, A.; Koehn, P.; Logacheva, V.; Monz, C.; Negri, M.; Neveol, A.; Neves, M.; Popel, M.; Post, M.; Rubino, R.; Scarton, C.; Specia, L.; Turchi, M.; Verspoor, K.; and Zampieri, M. 2016. Findings of the 2016 Conference on Machine Translation. In *Proceedings of the First Conference on Machine Translation*, 131–198. Berlin, Germany: Association for Computational Linguistics.

Chen, T.; Xu, B.; Zhang, C.; and Guestrin, C. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dong, K.; Zhou, C.; Ruan, Y.; and Li, Y. 2020. MobileNetV2 Model for Image Classification. In *2020 2nd International Conference on Information Technology and Computer Application (ITCA)*, 476–480.

Finesso, L.; and Spreij, P. 2006. Nonnegative matrix factorization and I-divergence alternating minimization. *Linear Algebra and its Applications*, 416(2): 270–287.

Google. 2022. Coral Dev Board Micro. Https://coral.ai/products/dev-board-micro/.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778.

Hinton, G.; Srivastava, N.; and Swersky, K. 2012. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8): 2.

Hu, E. J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; and Chen, W. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krizhevsky, A.; Hinton, G.; et al. 2009. Learning multiple layers of features from tiny images.

Kumar, R.; Panigrahy, R.; Rahimi, A.; and Woodruff, D. 2019. Faster Algorithms for Binary Matrix Factorization. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 3551–3559. PMLR.

Lee, D. D.; and Seung, H. S. 1999. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755): 788–791.

Lin, T.-Y.; Maire, M.; Belongie, S.; Bourdev, L.; Girshick, R.; Hays, J.; Perona, P.; Ramanan, D.; Zitnick, C. L.; and Dollár, P. 2015. Microsoft COCO: Common Objects in Context. arXiv:1405.0312.

Liu, D. C.; and Nocedal, J. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3): 503–528.

Luo, Y.; Ren, X.; Zheng, Z.; Jiang, Z.; Jiang, X.; and You, Y. 2023. CAME: Confidence-guided Adaptive Memory Efficient Optimization. In Rogers, A.; Boyd-Graber, J.; and Okazaki, N., eds., *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 4442–4453. Toronto, Canada: Association for Computational Linguistics.

Martens, J. 2016. *Second-order optimization for neural networks*. University of Toronto (Canada).

Narayan, S.; Cohen, S. B.; and Lapata, M. 2018. Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization. *ArXiv*, abs/1808.08745.

OpenAI; Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; Avila, R.; Babuschkin, I.; Balaji, S.; Balcom, V.; Baltescu, P.; Bao, H.; Bavarian, M.; Belgum, J.; Bello, I.; Berdine, J.; Bernadett-Shapiro, G.; Berner, C.; Bogdonoff, L.; Boiko, O.; Boyd, M.; Brakman, A.-L.; Brockman, G.; Brooks, T.; Brundage, M.; Button, K.; Cai, T.; Campbell, R.; Cann, A.; Carey, B.; Carlson, C.; Carmichael, R.; Chan, B.; Chang, C.; Chantzis, F.; Chen, D.; Chen, S.; Chen, R.; Chen, J.; Chen, M.; Chess, B.; Cho, C.; Chu, C.; Chung, H. W.; Cummings, D.; Currier, J.; Dai, Y.; Decareaux, C.; Degry, T.; Deutsch, N.; Deville, D.; Dhar, A.; Dohan, D.; Dowling, S.; Dunning, S.; Ecoffet, A.; Eleti, A.; Eloundou, T.; Farhi, D.; Fedus, L.; Felix, N.; Fishman, S. P.; Forte, J.; Fulford, I.; Gao, L.; Georges, E.; Gibson, C.; Goel, V.; Gogineni, T.; Goh, G.; Gontijo-Lopes, R.; Gordon, J.; Grafstein, M.; Gray, S.; Greene, R.; Gross, J.; Gu, S. S.; Guo, Y.; Hallacy, C.; Han, J.; Harris,

J.; He, Y.; Heaton, M.; Heidecke, J.; Hesse, C.; Hickey, A.; Hickey, W.; Hoeschele, P.; Houghton, B.; Hsu, K.; Hu, S.; Hu, X.; Huizinga, J.; Jain, S.; Jain, S.; Jang, J.; Jiang, A.; Jiang, R.; Jin, H.; Jin, D.; Jomoto, S.; Jonn, B.; Jun, H.; Kaftan, T.; Łukasz Kaiser; Kamali, A.; Kanitscheider, I.; Keskar, N. S.; Khan, T.; Kilpatrick, L.; Kim, J. W.; Kim, C.; Kim, Y.; Kirchner, J. H.; Kiros, J.; Knight, M.; Kokotajlo, D.; Łukasz Kondraciuk; Kondrich, A.; Konstantinidis, A.; Kosic, K.; Krueger, G.; Kuo, V.; Lampe, M.; Lan, I.; Lee, T.; Leike, J.; Leung, J.; Levy, D.; Li, C. M.; Lim, R.; Lin, M.; Lin, S.; Litwin, M.; Lopez, T.; Lowe, R.; Lue, P.; Makanju, A.; Malfacini, K.; Manning, S.; Markov, T.; Markovski, Y.; Martin, B.; Mayer, K.; Mayne, A.; McGrew, B.; McKinney, S. M.; McLeavey, C.; McMillan, P.; McNeil, J.; Medina, D.; Mehta, A.; Menick, J.; Metz, L.; Mishchenko, A.; Mishkin, P.; Monaco, V.; Morikawa, E.; Mossing, D.; Mu, T.; Murati, M.; Murk, O.; Mély, D.; Nair, A.; Nakano, R.; Nayak, R.; Neelakantan, A.; Ngo, R.; Noh, H.; Ouyang, L.; O'Keefe, C.; Pachocki, J.; Paino, A.; Palermo, J.; Pantuliano, A.; Parascandolo, G.; Parish, J.; Parparita, E.; Passos, A.; Pavlov, M.; Peng, A.; Perelman, A.; de Avila Belbute Peres, F.; Petrov, M.; de Oliveira Pinto, H. P.; Michael; Pokorny; Pokrass, M.; Pong, V. H.; Powell, T.; Power, A.; Power, B.; Proehl, E.; Puri, R.; Radford, A.; Rae, J.; Ramesh, A.; Raymond, C.; Real, F.; Rimbach, K.; Ross, C.; Rotsted, B.; Roussez, H.; Ryder, N.; Saltarelli, M.; Sanders, T.; Santurkar, S.; Sastry, G.; Schmidt, H.; Schnurr, D.; Schulman, J.; Selsam, D.; Sheppard, K.; Sherbakov, T.; Shieh, J.; Shoker, S.; Shyam, P.; Sidor, S.; Sigler, E.; Simens, M.; Sitkin, J.; Slama, K.; Sohl, I.; Sokolowsky, B.; Song, Y.; Staudacher, N.; Such, F. P.; Summers, N.; Sutskever, I.; Tang, J.; Tezak, N.; Thompson, M. B.; Tillet, P.; Tootoonchian, A.; Tseng, E.; Tuggle, P.; Turley, N.; Tworek, J.; Uribe, J. F. C.; Vallone, A.; Vijayvergiya, A.; Voss, C.; Wainwright, C.; Wang, J. J.; Wang, A.; Wang, B.; Ward, J.; Wei, J.; Weinmann, C.; Welihinda, A.; Welinder, P.; Weng, J.; Weng, L.; Wiethoff, M.; Willner, D.; Winter, C.; Wolrich, S.; Wong, H.; Workman, L.; Wu, S.; Wu, J.; Wu, M.; Xiao, K.; Xu, T.; Yoo, S.; Yu, K.; Yuan, Q.; Zaremba, W.; Zellers, R.; Zhang, C.; Zhang, M.; Zhao, S.; Zheng, T.; Zhuang, J.; Zhuk, W.; and Zoph, B. 2024. GPT-4 Technical Report. arXiv:2303.08774.

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A. 2017. Automatic Differentiation in PyTorch. In *NIPS 2017 Workshop on Autodiff*.

Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I.; et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8): 9.

Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; and Liu, P. J. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1): 5485–5551.

Rajpurkar, P.; Zhang, J.; Lopyrev, K.; and Liang, P. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. arXiv:1606.05250.

Reddi, S. J.; Kale, S.; and Kumar, S. 2019. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*.

Rombach, R.; Blattmann, A.; Lorenz, D.; Esser, P.; and Ommer, B. 2022. High-Resolution Image Synthesis with Latent Diffusion Models. arXiv:2112.10752.

Ruder, S. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M.; Berg, A. C.; and Fei-Fei, L. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3): 211–252.

Shazeer, N.; and Stern, M. 2018. Adafactor: Adaptive Learning Rates with Sublinear Memory Cost. In Dy, J.; and Krause, A., eds., *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, 4596–4604. PMLR.

Takase, S.; Kiyono, S.; Kobayashi, S.; and Suzuki, J. 2024. Spike No More: Stabilizing the Pre-training of Large Language Models. arXiv:2312.16903.

Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.-A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; Rodriguez, A.; Joulin, A.; Grave, E.; and Lample, G. 2023a. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971.

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. 2023b. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288.

Ultralytics. 2021. YOLOv5: A state-of-the-art real-time object detection system. https://docs.ultralytics.com.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Wang, A.; Singh, A.; Michael, J.; Hill, F.; Levy, O.; and Bowman, S. R. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Zhu, Y.; Kiros, R.; Zemel, R.; Salakhutdinov, R.; Urtasun, R.; Torralba, A.; and Fidler, S. 2015. Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books. In *The IEEE International Conference on Computer Vision (ICCV)*.