Error Feedback Can Accurately Compress Preconditioners

Ionut-Vlad Modoranu [1]   **Aleksei Kalinov** [1]   **Eldar Kurtic** [1]   **Elias Frantar** [1]   **Dan Alistarh** [1]

Abstract

Leveraging second-order information about the loss at the scale of deep networks is one of the main lines of approach for improving the performance of current optimizers for deep learning. Yet, existing approaches for accurate full-matrix preconditioning, such as Full-Matrix Adagrad (GGT) or Matrix-Free Approximate Curvature (M-FAC) suffer from massive storage costs when applied even to small-scale models, as they must store a sliding window of gradients, whose memory requirements are multiplicative in the model dimension. In this paper, we address this issue via a novel and efficient error-feedback technique that can be applied to compress preconditioners by up to two orders of magnitude in practice, without loss of convergence. Specifically, our approach compresses the gradient information via sparsification or low-rank compression *before* it is fed into the preconditioner, feeding the compression error back into future iterations. Experiments on deep neural networks show that this approach can compress full-matrix preconditioners to up to 99% sparsity without accuracy loss, effectively removing the memory overhead of full-matrix preconditioners such as GGT and M-FAC. Our code is available on our GitHub repository https://github.com/IST-DASLab/EFCP/.

1. Introduction

The remarkable success of stochastic gradient descent (SGD)-based optimizers in deep learning motivated a long line of research for accelerated preconditioned variants that can still scale to the massive parameter and dataset sizes common in the area. Among these, optimizers based on adaptive regularization, such as Adagrad (Duchi et al., 2010) and Adam (Kingma & Ba, 2015) are extremely well-established, with myriads of extensions, e.g. (Agarwal et al.,

2017; 2019; Yao et al., 2021; Gupta et al., 2018).

Yet, most work in this area restricts preconditioning of the descent direction to *a diagonal matrix*. By contrast, a promising—but considerably less developed—direction employs what we will call *full-matrix preconditioning*, i.e. pre-multiplying the descent direction with a full matrix, whose structure is justified either via adaptive regularization (Agarwal et al., 2019), or via approximations of natural gradient such as the Fisher approximation (Amari, 1998; Frantar et al., 2021).

Yet, existing implementations of full-matrix preconditioning are currently impractical given their massive memory costs. Interestingly, although they have different theoretical justifications, both full-matrix adaptive preconditioning methods such as GGT (Agarwal et al., 2019) and full-matrix natural-gradient methods such as M-FAC (Frantar et al., 2021) run into the same key practical limitation: each maintains a "sliding window" of several past gradients, employed in estimating the preconditioner at each step. Even for small-scale models, maintaining the gradient history results in unmanageable total cost of $\Theta(md)$ memory, where d is the model dimension, usually in the millions, and m is the size of the gradient history, usually of size 100-1000. For example, a standard recipe for fine-tuning the medium-scale BERT-base model using the M-FAC preconditioner with standard settings requires more than 450GB of GPU RAM, unavailable on any single GPU. Thus, while showing promising results in terms of accuracy, full-matrix preconditioning is currently impractical at scale due to memory constraints.

Contribution. In this paper, we present a new algorithmic approach for reducing the massive memory cost of full-matrix preconditioning at the scale of DNNs, called Error Feedback for Compressed Preconditioning (EFCP), which applies to both adaptive (Adagrad/GGT (Agarwal et al., 2019)) and natural-gradient (M-FAC (Frantar et al., 2021)) full-matrix preconditioners. In a nutshell, EFCP works by compressing the gradient window either via high sparsity or extremely low rank. In practice, we are able to compress the gradient history by one to two orders of magnitude *without impacting accuracy*, effectively removing the memory limitations of full-matrix preconditioning.

Our results are enabled by two technical innovations. On

---

the conceptual side, the surprising observation behind our work is that the sizable error in the preconditioner induced by gradient sparsification or low-rank projection can in fact be handled via an error feedback mechanism applied to the gradients *before they are stored* into the gradient window. For both variants, we provide efficient compression-aware data structures which can leverage compression for memory savings, and can apply to both M-FAC and GGT. We also provide partial theoretical justification for the good convergence properties of our algorithm.

On the practical side, our main contribution is an efficient GPU implementation for EFCP in the case of high sparsity, which realizes our theoretical gains in practice. Specifically, this is enabled by a novel GPU implementation of a *dynamic sparse ring-buffer* data structure, in which each element is a sparse gradient tensor, and which enables efficient sparse multiplication against new gradients, as needed for preconditioner computations, as well as fast gradient insertions and removals, as required by the sliding window semantics.

We validate our implementation experimentally on standard vision (ResNet/ImageNet) and language modeling (BERT/GLUE) tasks. Results show that EFCP enables us to significantly reduce the memory overhead of full-matrix preconditioning, to the point where it is in the same range as the memory required by standard optimizers such as SGD with Momentum or Adam.

We show this extensively by testing the EFCP implementation in the context of the M-FAC optimizer, compressing gradients to $\sim 99\%$ sparsity (a variant we call Sparse MFAC), as well as via smaller-scale experiments for low-rank compression and the GGT full-matrix Adagrad optimizer. For example, for the standard ResNet-18 model (He et al., 2016) *trained from scratch* on ImageNet (Russakovsky et al., 2015), the Sparse MFAC variant with full-matrix preconditioning uses a maximum of 22.5 GB of GPU RAM, compared to 21.3 GB used by SGD with momentum with the same parameters. At the same time, Sparse MFAC outperforms the well-tuned SGD baseline by approximately 1% Top-1 test accuracy, recovering the accuracy of Dense M-FAC in the same setup (which uses more than 56GB GPU RAM in this setting). We provide additional validation for Sparse M-FAC in the context of training ViT (Dosovitskiy et al., 2021) and BERT (Devlin et al., 2019) models as well, and perform ablations w.r.t. sparsity and model size.

In summary, for the first time, EFCP enables researchers to experiment with full-matrix preconditioner methods within the memory capacity of a single GPU in reasonable time, with negligible accuracy loss relative to full preconditioning.

2. Background and Related Work

2.1. General Setting

We consider a standard classification setting, in which we are given a training dataset $\mathcal{D} = \{(x_i, y_i)_{i=1}^N\}$ containing N i.i.d. samples x_i and their associated labels y_i. We suppose input pairs (x, y) are drawn from a true, unknown distribution $p_{true} \triangleq p_t$ having the density $p_t(x, y) = p_t(x)p_t(y|x)$ and our model $\theta \in \mathbb{R}^d$ parameterizes a probabilistic distribution $p_\theta(y|x) \triangleq p(y|f(x, \theta))$ with density $p_\theta(x, y) = p_t(x)p_\theta(y|x)$. Our aim is to minimize the "difference" between the true conditional distribution $p_t(y|x)$ and the modelled conditional distribution $p_\theta(y|x)$, measured in some appropriate metric. If we pick this to be the KL divergence between the two distributions, we obtain the following loss function $KL(p_t(x, y)||p_\theta(x, y)) = \int p_t(x, y) \log \frac{p_t(x,y)}{p_\theta(x,y)} dxdy = \mathbb{E}_{p_t(x)}[KL(p_t(y|x)||p_\theta(y|x))]$.

In practice, we only have a finite number of samples $x \sim p_t(x)$ and no information about the density function. In this case, we are using the empirical training distribution $\hat{p}_t$ instead of p_t, which is given by the samples in $\mathcal{D}$ and thus we can define the loss as $L(\theta) \triangleq KL(p_t(x, y)||p_\theta(x, y)) = \mathbb{E}_{\hat{p}_t(x)}[KL(p_t(y|x)||p_\theta(y|x))] \approx -\frac{1}{N} \sum_{i=1}^N \log p_\theta(y_i|x_i)$, which is the standard objective function minimized for the maximum likelihood estimation. The loss $L : \mathbb{R}^d \to \mathbb{R}$ is assumed to be continuous and twice differentiable.

In practice, the loss $L(\theta)$ defined above is often minimized using variants of Stochastic Gradient Descent (SGD) $\theta_{t+1} \leftarrow \theta_t - \eta_t g_t$, where $g_t \triangleq \nabla_\theta L(\theta_t)$ and $\eta_t \in \mathbb{R}$ is the learning rate. However, theoretical results suggest that convergence and end-results can be improved by *preconditioning* this gradient with a matrix C which incorporates information about the geometry of the loss landscape, leading to the parameter update rule $\theta_{t+1} \leftarrow \theta_t - \eta_t C_t^{-1} g_t$.

Natural Gradient. Instead of using the Euclidean distance to adapt to the information geometry, Natural Gradient (Amari, 1998) relies on KL-divergence to measure the discrepancy between the true distribution and the modelled distribution. In this context, the preconditioner C takes the form of the Fisher Information Matrix (FIM), defined as $F(\theta) \triangleq \frac{1}{N} \sum_{i=1}^N \mathbb{E}_{p_\theta(y|x_i)}[\nabla_\theta \log p_\theta(y|x_i) \nabla_\theta \log p_\theta(y|x_i)^T]$. One can show that the FIM is the expected Hessian with respect to the log-likelihood. Since the exact FIM is difficult to compute due to the expectation term in its definition, different efficient approximations of it exist in the literature. For neural networks, K-FAC (Grosse & Martens, 2016; Martens et al., 2018) provides a diagonal and tri-diagonal approximation of FIM by sampling a limited number of y values from $p_\theta(y|x_i)$ in order to compute the expectation.

Another popular approximation to the FIM is the Empirical Fisher, defined as $\hat{F}(\theta) \triangleq \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log p_\theta(y_i|x_i)\nabla_\theta \log p_\theta(y_i|x_i)^T$. Instead of sample y's from model's predictive distribution, it uses the associated label y_i. It is important to note that the empirical Fisher is a good approximation of the FIM if $p_t(y|x) \approx p_\theta(y|x)$, e.g. when the model has good accuracy. The M-FAC method (Frantar et al., 2021) (Section 2.2) provides an efficient implementation for this approach.

Adaptive Regularization. A different justification for obtaining good preconditioners is via *adaptive regularization*, arising from online convex optimization. Broadly, the idea is to adapt the learning rate corresponding to each parameter in order to minimize regret, defined as the cumulative difference between the optimizer's performance and the best decision in hindsight, by taking into account the history of gradients observed up to a given point. By and large, the community has focused on diagonal preconditioners, which allow computationally-efficient and extremely popular algorithms, such as Adam (Kingma & Ba, 2015) and AdaGrad (Duchi et al., 2010). The GGT result (Agarwal et al., 2019) (Section 2.2) showed that results can be improved by taking "second-order" weight correlations into account, leading to a full-matrix, $d \times d$ preconditioner, and provided an efficient approximation of this preconditioner.

2.2. Efficient Full-Matrix Preconditioners

Overview. We now provide a detailed description of the M-FAC full-matrix preconditioner (Frantar et al., 2021), which will serve as the main illustration of our preconditioner compression method in the paper body. A fully-worked-out example for the GGT preconditioner (Agarwal et al., 2019) is also provided in the Appendix. A common point of both methods is that they store a history of the past m gradients in the matrix

$$G = [g_{t-(m-1)}^T, g_{t-(m-2)}^T, ..., g_{t-1}^T, g_t^T] \in \mathbb{R}^{m \times d}, \quad (1)$$

which can be represented as a ring buffer, updated at each step by replacing the oldest gradient $g_{t-(m-1)}$ with the most recently-generated one g_t. M-FAC and GGT then use the ring buffer G differently to approximate the preconditioner. To preserve performance, the buffer G must be stored in the GPU memory. For best accuracy, both methods require storing a large window of m gradients, where m is usually between 100 and 1000, each of which are of size d (the model dimension). Thus, the memory cost of each method becomes intractable in large-scale settings. Next, we briefly describe the algorithmic implementation of M-FAC.

The M-FAC Optimizer. M-FAC provides an efficient estimator for Inverse-Hessian-Vector-Products (IHVPs) for

the special case where the Hessian is approximated via the Empirical Fisher approximation $\hat{F} = \frac{1}{m}\sum_{i=1}^{m} g_i g_i^T$, where the gradients $g_t \triangleq \nabla_\theta L(\theta_t)$ are stored in the matrix G, defined above. The first idea is to use the classic Woodbury-Sherman-Morrison (WSM) formula to integrate new gradient information as rank-1 updates into the current inverse. This leads to the following recursion to compute the IVHPs with an arbitrary vector $x \in \mathbb{R}^d$ as

$$\hat{F}_{t+1}^{-1}x = \left(\hat{F}_t + \frac{1}{m}g_t g_t^T\right)^{-1}x$$

$$\stackrel{WSM}{=} \hat{F}_t^{-1}x - \frac{\hat{F}_t^{-1}g_t(\hat{F}_t^{-1}g_t)^T}{m + g_t^T \hat{F}_t^{-1}g_t}x$$

$$= \frac{1}{\lambda}x - \sum_{k=1}^{t} \frac{\hat{F}_k^{-1}g_k(\hat{F}_k^{-1}g_k^T)}{m + g_k^T \hat{F}_k^{-1}g_k}x,$$

where $F_0 = \lambda I_d$ and $\lambda > 0$ is a damping parameter.

The key technical idea behind M-FAC is the authors' non-trivial observation that IHVPs can be expressed as a linear combination between the past gradients in the sliding window, where the coefficients are efficiently computed based only on scalar products $g_i^T g_j$ and $g_i^T x$, leading to the following expression for the IHVPs as

$$\hat{F}_m^{-1}x = \frac{1}{\lambda}x - \sum_{k=1}^{m} c_k g_k. \quad (2)$$

This allows M-FAC to efficiently add or remove gradients from the sliding window, done by computing the corresponding coefficients c_k, and then making the resulting update into the formula for $\hat{F}_m^{-1}x$.

A key aspect regarding M-FAC is the interaction with regularization. Recent work (Zhang et al., 2019) explains that weight decay and L_2-regularization coincide for SGD, but are different for preconditioned methods. Simply inputting $g_t = \nabla_\theta L(\theta) + \gamma\theta$ into the M-FAC preconditioner invalidates the empirical Fisher definition due to the additional term $\gamma\theta$. Based on this, and differring from M-FAC (Frantar et al., 2021), we will decouple the regularization such that the empirical Fisher definition holds and use the update $\theta_{t+1} = (1 - \gamma\eta_t)\theta_t - \eta_t\hat{F}_t^{-1}\nabla_\theta L(\theta_t)$, where $\gamma > 0$ is the weight decay parameter.

Data Structures and Operations. Next, we briefly summarize the main operations in the original M-FAC optimizer (denoted next by Dense M-FAC). We detail the challenges that appear when the internal buffer G is sparse, as well as our GPU-efficient solution, in the further sections. First, notice that the main operations in M-FAC are the Scalar Products (SP) and the Linear Combination of Gradients (LCG), which are needed to express the IHVPs as a linear combination.

Scalar Products (SP). This first operation corresponds to the pairwise scalar products of rows in $G \in \mathbb{R}^{m \times d}$ stored in an intermediary matrix $S = GG^T \in \mathbb{R}^{m \times m}$, where $S_{ij} = g_i^T g_j$. When the buffer G is updated (e.g. the oldest gradient is replaced by a new gradient $g \in \mathbb{R}^d$ at row i), the ith row and i^{th} column of matrix S are updated as $S_{i,:} = S_{:,i} = G \cdot g \in \mathbb{R}^m$. This operation can be easily performed on GPU using frameworks such as Pytorch.

Linear Combination of Gradients (LCG). The second main operation of M-FAC is the linear combination of gradients, having the coefficients vector $c \in \mathbb{R}^m$. For a *dense* buffer G, the LCG is easily computed by multiplying each component c_i with the ith row of G and then computing the sum of all rows in G. This operation can also be easily performed in PyTorch on GPU.

2.3. Related Methods

In the Appendix, we provide a full discussion of GGT (Agarwal et al., 2019), as well as its compressed version, which can be seen as a full-matrix version of AdaGrad optimizer. More generally, obtaining efficient approximations of second-order (curvature) information in deep learning is an active area. The *diagonal* approximation of the Hessian (Krishnan et al., 2018; Kingma & Ba, 2015) is very popular; however, it is known to provide lower quality relative to both block-wise methods or K-FAC (Wang et al., 2019b; Singh & Alistarh, 2020; Frantar et al., 2021).

K-FAC provides a block diagonal approximation of the FIM, and allows efficient computation of the inverse (Ba et al., 2017; Osawa et al., 2019; Zeng & Urtasun, 2019; Wang et al., 2019b; Laurent et al., 2018); however, it is known that its prerequisites do not always hold (Martens & Grosse, 2015). *Hessian-free* optimization (Martens, 2010) forgoes the explicit computation of Hessians in favor of computing an IHVP, but may require several "inner" iterations to converge at each step. AdaHessian (Yao et al., 2021) provided an alternative approximation of the inverse Hessian diagonal, using Hutchinson's randomized algorithm for estimating the diagonal, which requires smoothing heuristics, and has at least 2x theoretical iteration cost vs SGD, which in practice is usually around 3x. Shampoo (Gupta et al., 2018) provides an efficient heuristic approximation of GGT, reducing memory cost by maintaining preconditioner estimates per model tensor, using an efficient iterative method to compute preconditioner inverses.

The Error Feedback Mechanism. The error feedback mechanism has been studied extensively in the context of gradient compression for standard SGD, in the absence of complex preconditioning. The first analyses of sparsified SGD with error feedback were provided by (Stich et al., 2018; Alistarh et al., 2018) under additional assumptions,

while (Karimireddy et al., 2019; Stich & Karimireddy, 2020; Nadiradze et al., 2021) provided additions to the analysis. More recently, a very complex argument by Li et al. (2022) analyzed convergence of diagonal Adagrad with error feedback. We show that their analysis extends to a diagonal version of our algorithm.

3. Method

Algorithm 1 EFCP: Error Feedback for Accurate Compressed Full-Matrix Preconditioning

---

1: Parameters: T = steps count; m = gradient count;
2: $\xi_0 = 0_d \in \mathbb{R}^d \triangleright$ initialize error feedback
3: **for** each step $t \in \{1, 2, ...T\}$ **do**
4: $\quad g_t \leftarrow \nabla_\theta L(\theta_t) \triangleright$ get gradient
5: $\quad a_t \leftarrow \xi_{t-1} + g_t \triangleright$ error feedback
6: $\quad c_t \leftarrow \text{COMPRESS}(a_t) \triangleright$ compress the accumulator
7: $\quad \xi_t \leftarrow a_t - c_t \triangleright$ update the error
8: $\quad u_t \leftarrow \mathcal{A}(c_t) \triangleright$ update G and precondition using $\mathcal{A}$
9: $\quad \theta_t \leftarrow \theta_{t-1} - \eta_t u_t \triangleright$ update model parameters
10: **end for**

Overview. Our general approach for compressing the gradient history required by full-matrix preconditioners is described in Algorithm 1. Conceptually, we assume a standard gradient-based optimization setup with an oracle g_t at each step t, augmented by a preconditioning algorithm $\mathcal{A}$, based on a gradient sliding window G, whose large memory cost we want to reduce. We propose to do this by feeding *compressed* gradients into the preconditioner estimate maintained by algorithm $\mathcal{A}$. Yet, doing so directly by projecting the gradients onto a sparse or low-rank support diverges.

Instead, we apply *error feedback* (Seide et al., 2014; Alistarh et al., 2018; Karimireddy et al., 2019) to gradients *before they are fed into the preconditioner data structure*, using an error accumulator vector $\xi_t \in \mathbb{R}^d$, initially set to zero. At each step, we obtain the gradient g_t of the model parameters w.r.t. the loss, after which we feed back the error ξ_{t-1} into the current gradient g_t, obtaining the *accumulator* a_t. Next, the accumulator a_t is compressed, using e.g. sparsity or low-rank, to obtain a compressed representation c_t. The compression error ξ_t is updated to the difference between the accumulator a_t and its compressed version c_t. Finally, the compressed gradient c_t is fed directly to the preconditioning algorithm $\mathcal{A}$.

Notice that, other than the three lines corresponding to error feedback (blue in the algorithm), the optimization loop is the same as the uncompressed one. However, we emphasize that the internal buffer G of the preconditioned optimizer will store a sequence of m representations $[c_{t-(m-1)}, c_{t-(m-2)}, ..., c_{t-1}, c_t]$, each of which is highly-compressed, leading to memory savings.

While this blueprint appears simple, two key questions remain: (1) *how can we leverage the compressed gradient window for actual memory and computational speedups on complex real-world preconditioners?* and (2) *why does this approach preserve convergence?* We address them below.

3.1. Compressing Preconditioners via Sparsity

We now expand on accumulator compression via sparsity, specifically by using Top-k sparsification and error feedback. Specifically, the preconditioner data structure will only store $k \ll d$ entries from each accumulated gradient, where k is around 1% of the full dimension d. As we will show experimentally, this will essentially remove the memory overhead of full-matrix preconditioning.

We detail our approach for the Sparse M-FAC algorithm; the Sparse GGT variant is slightly simpler and is described in Appendix H due to space limitations. For Sparse M-FAC, we describe the changes required by sparsity for the buffer G and the Scalar Product (SP) and Linear Combination of Gradients (LCG) operations. A key concern will be re-interpreting the mathematical formulation in order to be able to leverage fast CUDA kernels, which is notoriously challenging in the context of sparsity.

Sparsity format for G. To store the gradient history G, we need a sparse matrix-like data structure for storing gradients that fulfills the following two properties at the same time: i) it allows easy replacement of any row (corresponding to adding and removing gradients) and ii) it allows computing the scalar product (SP) and linear combination of gradients (LCG) operations efficiently. To the best of our knowledge, no such data structure is known in the literature.

Specifically, for Sparse M-FAC, we want to input sparse accumulators c_t of size $k \ll d$ into the buffer G. Instead of storing $G \in \mathbb{R}^{m \times d}$ densely, we will store $G = (\mathcal{I}, \mathcal{V})$, where $\mathcal{I} \in \mathbb{N}^{m \times k}$ will store the indices returned by Top-k and $\mathcal{V} \in \mathbb{R}^{m \times k}$ will store the corresponding values. When the jth row of G is updated with the new indices-values pair (i_j, v_j), we replace the jth row in $\mathcal{I}$ and $\mathcal{V}$ with i_j and v_j, respectively. Breaking G into two matrices of indices and values means that G loses the matrix structure in the dense sense and the SP and LCG operations must be redefined because we cannot use the basic PyTorch operations for matrix multiplications anymore. Below we describe our solution for these two operations that involve CUDA specific kernels to efficiently perform the computations on a GPU for large models, but first we would like to provide details about the sparsity pattern which we can leverage to obtain more efficient kernels for our particular problem.

Sparsity Pattern. One observation is that, due to the algorithm structure, each row in $G = (\mathcal{I}, \mathcal{V})$ has the same

sparsity. To leverage this, at each step we compress the accumulator a_t using Top-k applied in blocks of (large) fixed size B. The block separation distributes compressed values more uniformly, speeding up the LCG operation. We express k as a percentage of total number of weights d and then convert it to a particular integer value that represents the density of values used in the Top-k call. For example, in most of our experiments we will have $k = 1\%$, meaning that for each block of size B we will have exactly $\lceil B/100 \rceil$ non-zero values. This way we know that for any block of size B we have the same number of weights in a specific interval. We provide more details about choosing B in the section where we describe the LCG operation.

Format for $\mathcal{V}$. Starting with the CUDA capability 8.0, the Nvidia GPUs have bfloat16 support, which is a half precision data type optimized for Deep Learning. We found that our optimizer still converges when we use bfloat16 instead of float32. This is particularly useful when automatic mixed precision (AMP) is activated because it speeds up the overall training and has a lower memory footprint (both for the model's state and for the optimizer's state).

Sparse Scalar Products. This operation is commonly known as sparse-matrix vector multiplication (SPMV) and several implementations already exist for a few sparsity representations (e.g. CSR, COO), but unfortunately they cannot be directly used for our particular choice for the matrix $G = (\mathcal{I}, \mathcal{V})$, each of size $m \times k$. We implement an SPMV kernel that leverages the properties of our buffer. We use m thread blocks, each containing 1024 threads. One thread block processes one row and has a shared memory array of 1024 floats to store the dot product computed on a subset of the row. We perform memory coalesced reads for the indices and values from the global memory. Lastly we perform parallel reduction over the shared memory buffer to compute the sum (the final value for the dot product) in logarithmic complexity. We show our implementation of the SP operation in the Appendix G.

Sparse LCG. To make computation of linear combination of gradients efficient, we heavily rely on the limited block sizes B with the fixed number of non-zero values in each block. When Top-k compression is applied blockwise, each selected index indicates a relative position inside the block and not an absolute position in the large weight matrix. The limited range of possible index values, combined with the freedom to choose the block size, allows us to reuse these indices directly as positions in the shared memory. We accumulate the linear combination of gradients across a slice of size $m \times B$ from $G = (\mathcal{I}, \mathcal{V})$ in shared memory and transfer the aggregated result in the output global memory at the correct offsets. Similar to the SP case, we also perform memory coalesced reads for the indices and values from the

global memory only once for each component. We show our implementation of the LCG operation in the Appendix G.

Thread Blocks for LCG. Choosing the right number of thread blocks is critical for the performance of the Sparse LCG, since it needs to balance memory transfer and compute load. To address this, we provide an automatic procedure for choosing this parameter, based on the GPU capabilities. Specifically, given a number of threads and maximum available GPU shared memory size, we determine the total number of thread blocks to be used to allow for full utilization, avoiding waiting in the queue at each SM.

Efficiency Improvements. The described technical data structure optimizations are critical for good performance. Relative to a naive Pytorch implementation of the same approach, which stores the gradients as sparse but performs the operations in dense format, the above approach is 1.5-2x faster and uses 20-30% less memory.

Memory Savings. Compressing the buffer using sparsity yields $\tilde{O}(mk)$ space complexity, where k is the target density of the gradients after Top-k compression, ignoring logarithmic factors due to indexing. In practice, we obtain stable convergence for $k = d/100$, which translates into practical space savings of approximately 20x (relative to GGT) and 30x (relative to M-FAC). We provide exact numbers in the experimental section.

3.2. Low-Rank Compression of Preconditioners

We also implement the COMPRESS step in Algorithm 1 via a low-rank approximation of the gradient, computed using an efficient variant of power iteration (Vogels et al., 2019). Similar to Top-k compression, we reformulate algebraic operations of M-FAC to leverage low-rank matrices for memory and computational savings. We provide implementation details in Appendix I and additionally explain the relation between matrix decomposition rank ρ and compression rate k of Top-k in Appendix J.

3.3. Theoretical Justification

Next, we turn our attention to theoretical justifications for our method. These are detailed in Appendix K, and summarized here. First, in the case where the preconditioning is only performed over the *diagonal*, corresponding for instance to the AdaGrad optimizer (Duchi et al., 2010) but with error feedback, we can show that our approach recovers a convergence rate of $O(1/\sqrt{T} + \sigma^2/\sqrt{T} + d/T)$ relative to the Adagrad baseline where gradients are uncompressed, by showing that it satisfies the preconditions of the analysis of Li et al. (2022) for diagonal adaptive methods with gradient compression.

In the much more challenging full-matrix case, we provide an argument showing that our approach can indeed come close to the strong $O(\sqrt{d/T})$ average regret bound of full-matrix Adagrad (Duchi et al., 2010), over T rounds and for dimension d, under some additional "smoothness" assumptions on the rate of change of the Hessian approximation.

In brief, our result says that, if 1) the Hessian approximation H_t is stable from one iteration t to another w.r.t. the error feedback, in the sense that $H_t H_{t-1}^{-1} \xi_t \simeq \xi_t$ for all $t \geq 1$, where ξ_t denotes the error buffer; and 2) if the misalignment between stochastic gradients and the preconditioning direction is bounded over time, then Sparse M-FAC obtains $O(\sqrt{d/T} \cdot \sqrt{d/k})$ average regret over T rounds.

Specifically, the additional regret due to sparse preconditioning is proportional to $\sqrt{d/k}$, the square root of the fraction of the gradient ℓ_2 norm preserved by the Top-k truncation operation. While $\Theta(\sqrt{d/k})$ is a worst-case bound, in practice it is usually lower due to the fact that gradient values tend to be normally-distributed. Overall, this argument provides partial justification for our method, although further work is needed to fully understand the remarkable practical convergence of Sparse M-FAC, which we illustrate next.

4. Experiments

We now validate our results experimentally. In the main text, we focus on comparing Sparse M-FAC with other optimizers, as it proved to be the most practical variant of EFCP. We show results for low-rank M-FAC and Sparse GGT in the Appendix.

In the following, we validate Sparse M-FAC (S-MFAC) experimentally as an optimizer for training in the context of standard vision and language modelling tasks. Specifically, we integrate our S-MFAC optimizer in ASDL (Osawa et al., 2023), a benchmarking library for second-order optimizers, but also examine larger-scale tasks such as ImageNet training of ResNet-18 and compare S-MFAC with AdamW and Dense M-FAC (D-MFAC) on BERT models on GLUE tasks. We report training loss, test accuracy and total running times and top memory usage for the entire training process.

Notations. We use the notation E for number of epochs, η for learning rate, γ for weight decay, B for batch size, m for the number of gradients (sliding window size), $k = 1\%$ for the gradient density. We use the prefixes **D** for **D**ense and **S** for **S**parse in the context of adaptive optimizers M-FAC. For the gradient window size m, we use the standard settings for D-MFAC (Frantar et al., 2021), which is $m = 1000$ unless otherwise specified.

The ASDL Benchmark. In the first experiment, we integrate S-MFAC into the ASDL benchmarking framework for

second-order optimizers (Osawa et al., 2023), for the task of fine-tuning a ViT-T model pre-trained on ImageNet. We report the test accuracy for the run that has largest validation accuracy. Table 1 shows the test accuracy of our S-MFAC optimizer compared to the other optimizers reported in the paper. (We had to remove P-SGD from the comparison due to errors raised by the implementation during experiments.) We observe that S-MFAC is competitive with other preconditioned gradient methods, outperforming SGD, AdamW, K-FAC and SENG, and matching the highest accuracy obtained on this task (98%) with affordable time and memory overheads relative to SGD.

Table 1. Test accuracy of S-MFAC for ResNet-18 and ViT-Tiny on CIFAR-10. The accuracies for the other optimizers are obtained from ASDL (Osawa et al., 2023). We also report relative time and memory overheads w.r.t. SGD.

Optimizer	ViT-T	Time	Memory
SGD	97.8	$1\times$	$1\times$
AdamW	97.9	$1\times$	$1.01\times$
K-FAC(1mc)	97.4	$1.12\times$	$1.13\times$
SENG	97.7	$35\times$	$7.31\times$
Shampoo	98.0	$1.6\times$	$1.07\times$
S-MFAC	98.0	$1.63\times$	$1.15\times$

ImageNet/ResNet-18. Next, we move to a more complex vision task, by training ResNet-18 on ImageNet (Deng et al., 2009) *from scratch* using a highly-tuned version of SGD (with momentum) from the FFCV (Leclerc et al., 2022) library, as well as S/D-MFAC. All hyper-parameters are provided in the Appendix E. Figure 1 shows the validation accuracy curves for this experiment, while Figure 2 shows the best achieved training loss and validation accuracy for each method, including a detailed sparsity sweep below 1% density for S-MFAC. The loss curves show similar trends, and are provided in the Appendix.

We begin by noting that SGD obtains training loss with value of 2.237 and accuracy of 69.12%. D-MFAC reaches the loss value of 2.164 and significantly higher Top-1 validation accuracy, 70.03%. Despite slightly higher training loss than D-MFAC (2.185), S-MFAC has comparable 70.01% validation accuracy, which is strong for this model and training recipe. Thus, even for this challenging task, compressing the preconditioner essentially recovers the accuracy of the dense M-FAC optimizer. When we examine the behavior of S-MFAC as we decrease gradient density, we observe that, remarkably, S-MFAC with $k = 0.17\%$ (only $\approx 20k$ entries!) still achieves comparable accuracy to SGD, and that accuracy decreases gradually. However, the resulting preconditioned gradients are not necessarily very sparse.

We also run Shampoo in the same settings and we found

Figure 1. Validation (Top-1) accuracy for ResNet18 training from scratch on ImageNet-1K. M-FAC and 99%-Sparse M-FAC reach approximately 1% higher validation accuracy relative to SGD using the well-tuned FFCV recipe (Leclerc et al., 2022).
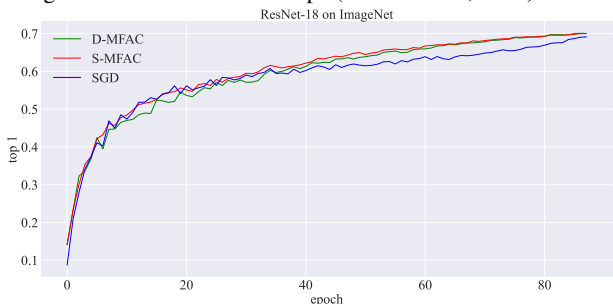
that the memory usage is lower, but comes at a significantly higher running time (around $3\times$). We provide details on Shampoo comparison in Appendix C.

To examine memory, we ran the experiments on a single A100 GPU and recorded the max memory usage throughout the entire process via the NVIDIA-SMI tool. We find that SGD uses 12.8 GB and S-MFAC 14.3 GB, while D-MFAC in the same setting needs 59 GB.

Figure 2. Training loss and validation (Top-1) accuracy for ResNet18 training from scratch on ImageNet-1K under a detailed sparsity sweep, between 1% and 0.04% density. Dense M-FAC outperforms SGD in this setting by about 1%, which is recovered by S-MFAC at 1% density. Even with 0.17% gradient density, M-FAC slightly outperforms SGD in terms of training loss and test accuracy.

Density k	Train Loss	Test Accuracy
D-MFAC (100%)	2.164	70.03%
117k (1.00%)	**2.185**	**70.01%**
100k (0.85%)	2.183	70.18%
75k (0.64%)	2.187	69.99%
50k (0.42%)	2.190	69.68%
25k (0.21%)	2.206	69.43%
20k (0.17%)	**2.217**	**69.26%**
15k (0.12%)	2.230	69.05%
10k (0.08%)	2.246	68.84%
5k (0.04%)	2.300	68.27%
SGD	2.237	69.12%

BERT/GLUE-MNLI. Finally, we test our S-MFAC implementation for BERT-TINY/MINI/BASE models on the MNLI task from the GLUE benchmark (Wang et al., 2019a), comparing against the Adam optimizer with optimized default settings from Hugging Face (Devlin et al., 2019), which reproduce or improve upon the original paper (Devlin et al., 2019). Dense M-FAC with a window size of $m = 1024$ can only be executed on the TINY and MINI model variants, requiring 18 GB and 45 GB memory, respectively, and runs

out of memory (OOM) for the BASE model (the required memory usage for the ring buffer G alone is 417GB).

The results in Table 2 show that our S-MFAC implementation not only recovers the performance of D-MFAC on BERT-TINY/MINI, but also allows us to run on BERT-BASE, with superior validation accuracy relative to Adam using standard hyper-parameters (e.g., learning rate $\eta = 2 \cdot 10^{-5}$). For the TINY model S-MFAC achieves almost 2% higher accuracy compared to D-MFAC and close to 5% higher validation accuracy compared to Adam. The former improvement can be justified by the regularization effect of sparsity relative to D-MFAC in a fine-tuning task, whereas the latter is due to leveraging second-order information.

At the same time we observe that, accuracy differences become smaller as model size increases. This behavior is also reported in the original D-MFAC (Frantar et al., 2021) optimizer for the same GLUE/MNLI task. While this is another indication that the sparse implementation preserves the behavior of the dense counterpart, it suggests that the benefits of second-order information diminish in the case of over-parametrized fine-tuning tasks.

Table 2. Evaluation accuracy for BERT-**T**iny/**M**ini/**B**ase on GLUE-MNLI. OOM stands for Out of Memory error.

	T	M	B
Adam	66.86	74.77	84.6
S-MFAC	**71.06**	**76.40**	**84.86**
D-MFAC	69.26	**76.39**	OOM

Running times and memory usage. Finally, we examine running times and memory usage for the algorithm. The results across our experiments are shown in Table 3. We note that, in the current implementation, both D-MFAC and our optimizer have a constant additional cost relative to the highly-optimized Adam or SGD implementations in Pytorch, due to the additional data structures employed. This cost can be reduced via optimizations.

Specifically, our current implementation was primarily optimized for residual networks. This is reflected in the runtime and memory numbers: on ResNet-18 (11.7M params), D-MFAC requires $4.6\times$ more memory compared to SGD with momentum and 25% larger running time. Our S-MFAC implementation diminishes these costs bringing memory usage to 25% of D-MFAC and making it comparable with SGD both in the running time and memory usage.

For BERT-TINY (4M params), D-MFAC uses $14\times$ more memory and BERT-MINI (11M params) uses $15\times$ more memory compared to S-MFAC. Further, we find that EFCP is extremely effective for large models like BERT-BASE (110M params). In this case, running D-MFAC would raise

an out of memory (OOM) error (just the dense ring buffer used to store the gradient window requires 417 GB of memory under standard parameters).

Although our primary concern in experiments is memory cost, we observe that, in terms of running time, our algorithm has essentially no overhead on ResNet-18. On BERT models, there is a notable runtime overhead, which is due in part to the fact that our CUDA kernels are not specifically adapted to BERT models. We expect this overhead to be reduced via tuning and further optimizations, which we plan to investigate in future work. However, we note that these single-GPU runtimes are very manageable already.

To decrease memory usage and running time, one can try reducing the number of gradients m. However, it is clear from prior work (Agarwal et al., 2019; Frantar et al., 2021) that large values of m are needed to provide better accuracy results, and specifically to improve over standard baseline. This is theoretically motivated in part by the fact that the rank of matrix G is at most m, which can be extremely low compared to the rank of Hessian matrix for large models and the approximation would be very poor for low values of m. Thus, large models require large number of gradients m, which renders our technique effective. To our knowledge, our approach allows full-matrix adaptive optimizers to be run on a consumer GPU for the first time. Appendix F contains a detailed theoretical analysis for memory savings.

Table 3. Running times and memory usages for ImageNet/ResNet-18 and GLUE/BERT reported on an NVIDIA A100 GPU with 82GB RAM. OOM stands for Out Of Memory.

Model	Optimizer	Memory	Training Time
ResNet-18	SGD	12.8 GB	4h 10m
	S-MFAC	14.3 GB	4h 21m
	D-MFAC	58.5 GB	5h 33m
BERT-Tiny	Adam	0.8 GB	11m
	S-MFAC	1.3 GB	1h 5m
	D-MFAC	17.9 GB	1h 20m
BERT-Mini	Adam	1.3 GB	21m
	S-MFAC	2.9 GB	1h 5m
	D-MFAC	44.9 GB	1h 50m
BERT-Base	Adam	7.1 GB	1h 45m
	S-MFAC	14.8 GB	3h 20m
	D-MFAC	OOM	-

Error Feedback Evolution During Training. In line 5 of Algorithm 1 the error feedback mechanism ensures that the top largest values by magnitude (the outliers) are always transferred at each step from the error buffer ξ into the optimizer state and we are interested in whether the EF stays unbounded relative to the gradient values applied. To showcase this practically, we measured the size of the error feedback relative to the total gradient norm applied up to a

certain step, formally $||\xi_t||_2/(||g_1||_2 + \ldots + ||g_t||_2)$. Intuitively, this should tell us whether the gradient information is stuck in the error feedback (equivalent to an increase or large constant value of this metric) or whether the error feedback acts like a temporary buffer before the information is transferred to the optimizer's state (in which case it should become smaller over time). We plot this metric during the entire training process for two experiments, BERT-Base/GLUE-MNLI and ResNet-20/CIFAR-10, where we use $k = 1\%$ gradient density for S-MFAC. We found that in both experiments this metric decreases exponentially, which indicates the accumulated gradient information is indeed transferred from the error feedback ξ to the optimizer state. We show the experiment result in Appendix A.

Generalization. Since the preconditioner is highly sparse, we are interested in how our technique affects generalization. To address this, we inspected the eigenvalues of the loss landscape by conducting an experiment on ResNet-20 / CIFAR-10 to compute the maximum eigenvalue of the Hessian matrix of the loss function. We used the Power Iteration method for SGD, D-MFAC and S-MFAC with gradient sparsities 90%, 95%, 99%, 99.5% and 99.9%. We use the entire train and test datasets and report the maximum eigenvalue after the initialization and then at the end of each epoch. We found that D-MFAC and S-MFAC yield much flatter solutions that SGD and S-MFAC yields only slightly larger maximum eigenvalues compared to D-MFAC. This is an indication that compressing the buffer yields the same optimizer properties. We provide the results in Appendix B.

Tuning Recipe. Our method has similar behavior as the original D-MFAC. The most important hyper-parameters are learning rate η, dampening λ, window size m and batch size. Briefly, the batch size controls the noise in the gradient and D/S-MFAC have stable convergence for both large and small batch sizes; regarding dampening, we found some general rules of thumb for how to tune the dampening in relationship with the learning rate, based on the idea that changing dampening makes S/D-MFAC to behave like as SGD (for large values) or like Newton's method (for small values); regarding buffer size m, as explained in the original D-MFAc paper, it is recommended to use $m = 1024$ for best results; regarding gradient density, we observed that there is a natural density threshold above which using higher density yields same results. In Appendix D we provide more details on how to tune these parameters to obtain the best out of our method.

5. Conclusions and future work

We have provided a versatile new tool for addressing the memory costs of adaptive full-matrix preconditioning, in the form of a series of methods which compress the gradi-

ent history, by leveraging an instance of the error feedback mechanism. We complement this mechanism via efficient sparse/low-rank algorithms for maintaining existing preconditioners, and have shown experimentally that our approach can essentially provide lossless compression when applied via sparsity, with remarkably large compression rates. We provide CUDA implementations for a sparse data structure that 1) allows easy replacement of any row (adding/replacing new compressed representations of error feedback accumulators and 2) allows computing the SP and LCG operations efficiently which, to the best of our knowledge, is the first such data structure known in the literature.

In future work, we plan to further investigate theoretical justification for our method. Obtaining general bounds appears extremely challenging, as even in the case of standard SGD understanding error feedback required significant technical effort, e.g. (Alistarh et al., 2018; Karimireddy et al., 2019). Another direction of future work is exploring larger-scale validation of our method for additional architectures and tasks, and for additional preconditioning mechanisms.

Acknowledgements

Impact Statement

This paper presents work whose goal is to improve certain aspects of existing algorithms in the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here because our work can be used by anyone without our control.

References

Agarwal, N., Bullins, B., and Hazan, E. Second-order stochastic optimization for machine learning in linear time. *Journal of Machine Learning Research*, 18(116): 1–40, 2017.

Agarwal, N., Bullins, B., Chen, X., Hazan, E., Singh, K., Zhang, C., and Zhang, Y. Efficient full-matrix adaptive regularization. In *International Conference on Machine Learning*, pp. 102–110. PMLR, 2019.

Alistarh, D., Hoefler, T., Johansson, M., Konstantinov, N., Khirirat, S., and Renggli, C. The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*, pp. 5973–5983, 2018.

Amari, S.-i. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998.

Ba, J., Grosse, R., and Martens, J. Distributed second-order optimization using Kronecker-factored approximations. *International Conference on Learning Representations*, 2017.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. IEEE, 2009.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, 2019.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proceedings of the Ninth International Conference on Learning Representations*, 2021.

Duchi, J. C., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2010.

Fey, M. et al. Pytorch-sparse library. https://github.com/rusty1s/pytorch_sparse, 2018.

Frantar, E., Kurtic, E., and Alistarh, D. M-FAC: Efficient matrix-free approximations of second-order information. *Advances in Neural Information Processing Systems*, 35, 2021.

Grosse, R. and Martens, J. A Kronecker-factored approximate fisher matrix for convolution layers. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48, pp. 573–582, 2016.

Gupta, V., Koren, T., and Singer, Y. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pp. 1842–1850. PMLR, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Karimireddy, S. P., Rebjock, Q., Stich, S. U., and Jaggi, M. Error feedback fixes SignSGD and other gradient compression schemes. In *Proceedings of the Thirty-sixth International Conference on Machine Learning*, pp. 3252–3261, 2019.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations*, 2015.

Krishnan, S., Xiao, Y., and Saurous, R. A. Neumann optimizer: A practical optimization algorithm for deep neural networks. In *International Conference on Learning Representations*, 2018.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

Laurent, C., George, T., Bouthillier, X., Ballas, N., and Vincent, P. An evaluation of Fisher approximations beyond Kronecker factorization. In *International Conference on Learning Representations*, 2018.

Leclerc, G., Ilyas, A., Engstrom, L., Park, S. M., Salman, H., and Madry, A. FFCV: Accelerating training by removing data bottlenecks. https://github.com/libffcv/ffcv/, 2022.

Li, X., Karimi, B., and Li, P. On distributed adaptive optimization with gradient compression. In *International Conference on Learning Representations*, 2022.

Martens, J. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning*, pp. 735–742, 2010.

Martens, J. and Grosse, R. Optimizing neural networks with Kronecker-factored approximate curvature. In *Proceedings of the 32th International Conference on Machine Learning*, 2015.

Martens, J., Ba, J., and Johnson, M. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*, 2018.

Nadiradze, G., Markov, I., Chatterjee, B., Kungurtsev, V., and Alistarh, D. Elastic consistency: A general consistency model for distributed stochastic gradient descent. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(10):9037–9045, May 2021.

Osawa, K., Tsuji, Y., Ueno, Y., Naruse, A., Yokota, R., and Matsuoka, S. Large-scale distributed second-order optimization using Kronecker-factored approximate curvature for deep convolutional neural networks. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2019.

Osawa, K., Ishikawa, S., Yokota, R., Li, S., and Hoefler, T. ASDL: A unified interface for gradient preconditioning in pytorch, 2023.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3): 211–252, 2015.

Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

Singh, S. P. and Alistarh, D. WoodFisher: Efficient second-order approximation for neural network compression. In *Advances in Neural Information Processing Systems*, volume 33, pp. 18098–18109, 2020.

Stich, S. U. and Karimireddy, S. P. The error-feedback framework: Better rates for sgd with delayed gradients and compressed communication. *Journal of machine learning research*, 2020.

Stich, S. U., Cordonnier, J.-B., and Jaggi, M. Sparsified sgd with memory. *Advances in Neural Information Processing Systems*, 31, 2018.

Vogels, T., Karimireddy, S. P., and Jaggi, M. PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the Seventh International Conference on Learning Representations*, 2019a.

Wang, C., Grosse, R., Fidler, S., and Zhang, G. EigenDamage: Structured pruning in the Kronecker-factored eigenbasis. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97, pp. 6566–6575, 2019b.

Yao, Z., Gholami, A., Shen, S., Keutzer, K., and Mahoney, M. W. ADAHESSIAN: An adaptive second order optimizer for machine learning. *AAAI*, 2021.

Zeng, W. and Urtasun, R. MLPrune: Multi-layer pruning for automated neural network compression, 2019. URL https://openreview.net/forum?id=r1g5b2RcKm.

Zhang, G., Wang, C., Xu, B., and Grosse, R. Three mechanisms of weight decay regularization. In *International Conference on Learning Representations*, 2019.

A. Error Feedback Evolution During Training

In this section we show the experimental results for the metric $||\xi_t||_2/(||g_1||_2 + \ldots + ||g_t||_2)$

In Figure 3 for BERT, the stochastic gradient is clipped to norm 1 and the cumulative sum of the norm of the gradients up to step t will always be t. In this case, the error feedback norm is much smaller than $t = ||g_1|| + \ldots + ||g_t||$, leading to an exponential decrease in the metric.

In Figure 4 for ResNet-20, the stochastic gradient is not clipped and its norm decreases naturally during the training. In this case, the error feedback norm decreases, also leading to an exponential decrease in the metric.

These two examples clearly indicate that the error feedback does not accumulate significant errors over time, but it acts as a temporary buffer of past gradient information before the top-k operator removes the outliers and are sent to the preconditioner.

Figure 3. Norms of the error feedback $||\xi_t||_2$, norm of the gradient $||g_t||_2$ and the EF metric value for BERT-Base.

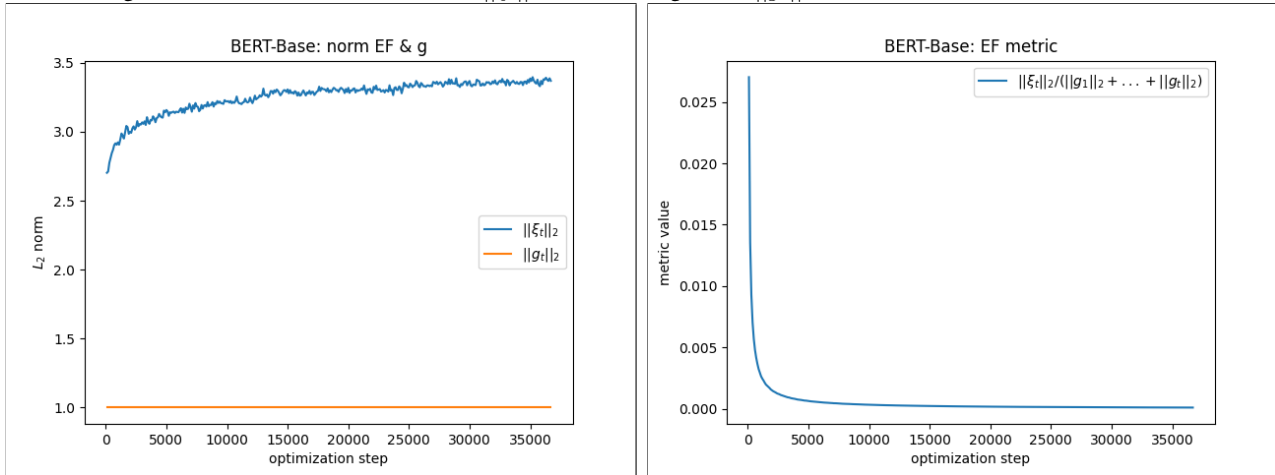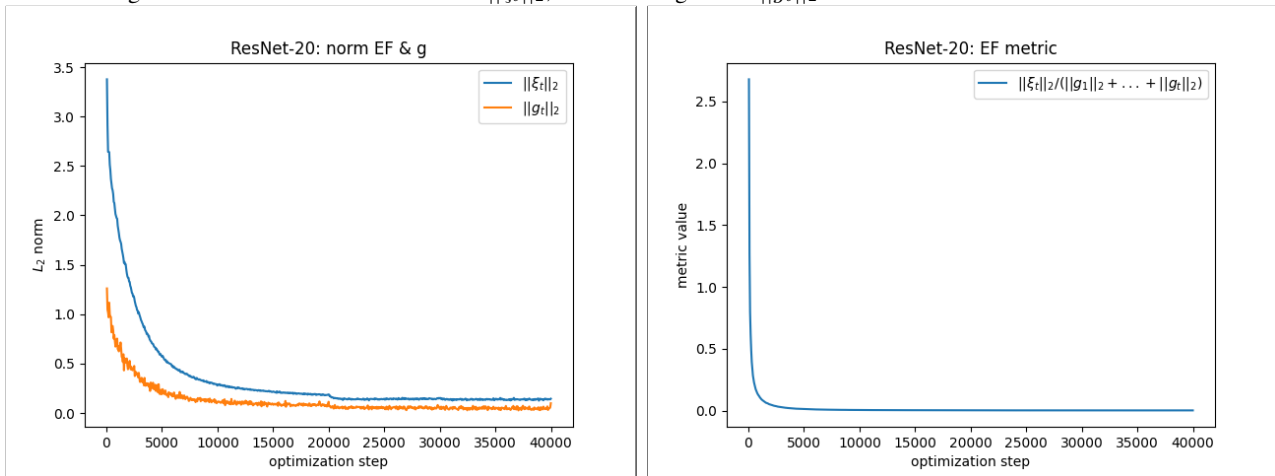*Figure 4.* Norms of the error feedback $||\xi_t||_2$, norm of the gradient $||g_t||_2$ and the EF metric value for ResNet-20.

B. Generalization.

In Table 4 we show the results for our experiment on maximum eigenvalue of the hessian of the loss function. We observe that all M-FAC versions (dense and sparse) lead to a flat minima where the maximum eigenvalue is close to zero for both training and test datasets, while SGD leads to a sharp minimum with a maximum eigenvalue much larger than 0 for both

training and test datasets. This behavior is similar to the one illustrated in Figure 1 of Keskar et al., available here.

Table 4. Maximum eigenvalues computed on train / test datasets.

epoch	SGD	D-MFAC	k = 0.1%	k = 0.5%	k = 1%	k = 5%	k = 10%
init			1231.34 / 942.1				
10	29.04 / 30.36	1.0 / 1.36	11.55 / 14.81	2.68 / 3.8	1.84 / 2.25	1.44 / 1.72	2.16 / 2.56
20	17.37 / 29.46	0.22 / 0.23	1.55 / 1.83	0.38 / 0.43	0.27 / 0.34	0.26 / 0.33	0.35 / 0.42
40	14.61 / 20.84	0.09 / 0.11	0.25 / 0.39	0.11 / 0.14	0.09 / 0.1	0.08 / 0.11	0.11 / 0.14
60	10.38 / 21.34	0.05 / 0.09	0.13 / 0.23	0.05 / 0.11	0.05 / 0.1	0.05 / 0.08	0.06 / 0.11
80	10.19 / 24.51	0.05 / 0.1	0.11 / 0.25	0.07 / 0.12	0.06 / 0.1	0.05 / 0.1	0.06 / 0.11
100	9.12 / 24.51	0.04 / 0.1	0.1 / 0.23	0.07 / 0.1	0.06 / 0.11	0.05 / 0.11	0.05 / 0.12

C. ImageNet / ResNet-18 / Shampoo

The ASDL experiments presented in Table 1 show that Shampoo is a strong competitor to our work with a lower memory usage. To address this, we integrated the PyTorch implementation of Shampoo optimizer from Google Research (Gupta et al., 2018) into our ImageNet training pipeline. This reveals a few trade-offs when running Shampoo at scale. Specifically, its two key parameters are:

- **preconditioning_compute_steps**: an integer that controls when the preconditioner is updated. For S-MFAC, this is always 1, since we re-compute at each step; however, Shampoo becomes very slow in this setting, so we had to increase this to 10 to run at reasonable speed.

- **block_size**: an integer that controls the matrix approximation. We used the default 128, which means that we are performing a block diagonal approximation for the preconditioner.

The efficiency-accuracy trade-off for Shampoo is the following: a large block size makes Shampoo be closer to the more accurate full-matrix version, and should be paired with a large update interval. The configuration for Shampoo that makes it equivalent to D/S-MFAC would be to update the preconditioner at each step with maximum block size.

However, we were unable to run this "full" version of Shampoo at scale, due to speed/memory issues. For instance, the maximum block size we could run in an RTX 3090 GPU for ResNet18 was 8192 (3x slower than the default 128, and about 10x slower than S-MFAC in the same setting), and we had to reduce **preconditioning_compute_steps** to 10 to get reasonable training times on ImageNet.

We used Shampoo for training in exactly the same setting as in our paper, following the learning rates described in section 6.1 of the original paper [1]. To get reasonable running times, we re-computed the preconditioner once every 10 steps, and used the default suggested block size 128. We present our Shampoo results in Table 5.

Table 5. Results for Shampoo / ImageNet / ResNet-18.

lr	top1	time	memory usage
1	68.82%	11h 38m	12.66GB
2.5	69.62%	11h 19m	12.66GB
5	70.24%	11h 21m	12.66GB

In comparison, we report the results for our optimizer in the following table:

We ran Shampoo in exactly the same settings as SGD, D-MFAC and our S-MFAC: linearly decaying learning rate, no image scaling, weight decay $1e-5$ for 88 epochs and batch size 1024 (1251 steps per epoch). We mention that some Shampoo runs crashed towards the end of training due to numerical instability, but we were still able to extract good-quality checkpoints. In Figure 5 we show the plots for top-1 accuracy and training loss for Shampoo.

Table 6. Results for S-MFAC / ImageNet / ResNet-18.

top1	time	memory usage
S-MFAC (k=1%)	14.3 GB	4h 21m

We would like to mention that we also tried to train BERT-Base on GLUE/MNLI, but unfortunately the run was too slow. The total number of training steps is 36k and it took 2h 48m for 1200 steps. Since the model is large (110M params), we set the block size to 16384. A small block size (e.g. the default 128) would have resulted in a lot of smaller blocks that resulted in similar running times. Our S-MFAC approch takes 3h 20m for the entire run.

Figure 5. Comparison between S-MFAC and Shampoo on ResNet-18 / ImageNet
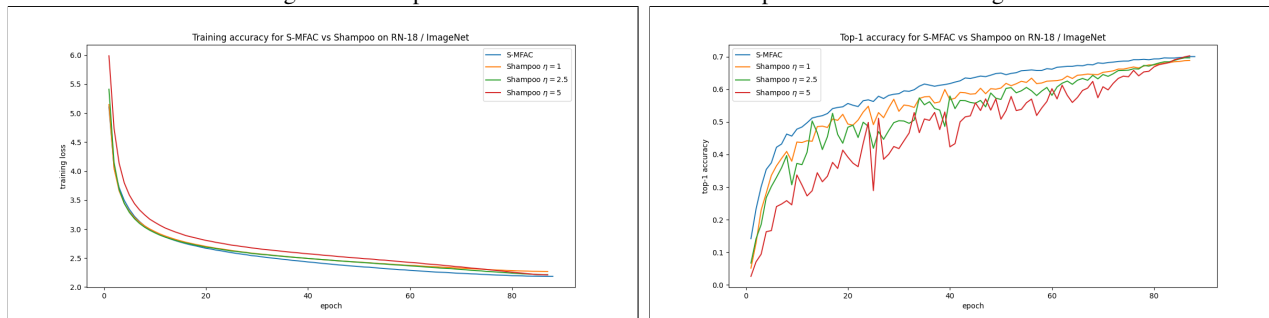
D. Tuning Recipe

In this section we explain the influence of the main hyper-parameters.

Batch size.   The first parameter of interest is the batch size, which controls the noise in the gradient. We performed an ablation study on both S/D-MFAC for ResNet-20/CIFAR-10 (we use this small datasets/model for time reasons) without weight decay and with parameters that are reported by the original D-MFAC.

We tried batch sizes 32, 64, 128, 256 and 512. Both S/D-MFAC have stable convergence at both small and large batch sizes, although using a small batch size will result in a large total number of optimization steps and this might increase the running time (as it happens for any optimizer).

Using a large batch size for D-MFAC can lead to an Out of Memory Error (OOM) because both model and optimizer state have high memory footprint. In contrast, S-MFAC will keep a low memory footprint and one can increase the batch size accordingly such that the GPU memory usage is maximized.

Dampening λ.   Since M-FAC gives a Hessian approximation based on empirical Fisher, it is implicitly a trust region method. The dampening parameter λ is inversely proportional to the unknown trust region radius R and it controls the behavior of the optimizer. When λ is small, M-FAC is similar to Newton's method, while large λ reduces to SGD with a scaled learning rate η_t/λ. We found two rules of thumb that proved to be useful in tuning the learning rate η and dampening λ:

1. if the model diverges, then the dampening is too small and it should be increased. This means S/D-MFAC is too close to Newton's method and by increasing it we make it more similar to SGD. The dampening influences the norm of the linear combination parameters in Equation 2. If the norm becomes too large, this is equivalent to a behavior similar to Newton's method (too small dampening).

2. if the model underfits, then the dampening is too large and it should be decreased. This means S/D-MFAC is too close to SGD and by decreasing it we make it more similar to Newton's method.

14

History size m. As explained in the original D-MFAC paper, it is recommended to use a large history size for the following reasons:

From a theoretical perspective, the rank of the Empirical Fisher Matrix (EFM) is at most m. This is already a limitation when approximating the real Hessian matrix (which is likely to have a rank $r >> m$) and using a value of m as large as possible is recommended; from a practical perspective, the recommended value of m is up to 1024. On the one hand, this is recommended since the maximum number of threads in a block (across which parallelization is performed) on recent NVIDIA GPUs Nvidia is 1024; on the other, we do not observe additional accuracy improvements for larger history sizes.

Sparsity. We are using the same experiment on ResNet-20/CIFAR-10 to provide explanations about how we should set the sparsity. In our experiments so far we used the following density percentages k: 0.1%, 0.5%, 1%, 5% and 10%. We have observed that there is always a natural density percentage threshold k^* above which using higher density (equivalent to lower sparsity) yields the same results. For example, in our experiments so far we observed that $k = 1\%$ obtains best results for $m = 1024$ gradients, while for larger values of k we observe a slight decline in test accuracy.

As a summary, we would suggest using $k = 1\%, m = 1024$ for the optimizer parameters, as "universal" parameters, which worked well across all our experiments. The user can choose the batch size to maximize memory/compute usage for the given GPU/TPU.

E. Hyper-parameters

We restate the notations that we introduced in the main body of our work. We use notation μ for momentum, η for the initial learning rate, γ for weight decay, λ for M-FAC dampening, k for gradient density, ρ for rank, E for epochs, B for batch size.

E.1. ASDL

We perform hyper-parameter tuning for CIFAR-10 and ViT-Tiny, as described in the appendix section of the ASDL (Osawa et al., 2023) paper. Concretely, the general grids we use are $\{32, 128, 512, 2048\}$ for the batch size, $\{3e-1, 1e-1, 3e-2, 1e-2, 3e-3, 1e-3\}$ for the learning rate, $E = 20$ epochs, we clip the preconditioned gradient norm to 10 and perform grid search for the dampening parameter λ over the grid $\{1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5, 1e-5, 5e-6, 1e-6\}$ for S-MFAC. In particular, ViT-Tiny uses $\gamma = 1e-4$.

E.2. FFCV

For the ImageNet dataset and ResNet-18 model, we use $E = 88$ epochs, batch size $B = 1024$, linear decay schedule for the learning rate η. In particular, we set the weight decay $\gamma = 1e-5$ for convolution and fully connected parameters and $\gamma = 0$ for batch-normalization parameters, thus fixing the issue in the main FFCV repository. We use linear learning rate decay for all three optimizers and disable image rescaling. We present the final hyper-parameters in Table 9.

E.3. BERT/GLUE

We use the HuggingFace repository and plug in our D/S-MFAC optimizers. We compare against default Adam baselines as follows. For GLUE/MNLI we use $E = 3, \eta = 2e-5, \gamma = 0$. D-MFAC uses the default parameters from the original paper, e.g. $\eta = 1e-4, \lambda = 1e-6, \gamma = 0$ and we adopt the same values for S-MFAC for BERT-TINY/MINI. For BERT-BASE we use $\lambda = 5e-5$ for S-MFAC.

E.4. CIFAR-10/RN-20

For D/S-GGT we use window size $m = 100, k = 1\%$ and the grid in Table 7. We show the corresponding pairs (η, ϵ) for the best runs extracted from the search grids. In these experiments we use $E = 164, B = 128$ and decay the learning rate by a factor of $\gamma_\eta = 0.1$ at 50% and 75% of training. The final hyper-parameters for each method are summarized in table 8.

F. Memory savings.

In this section we detail the theoretical memory savings for the sparse preconditioners by computing the total memory in bytes and the ratio of dense over sparse preconditioners to describe de memory savings.

Table 7. Search grid for learning rate η and dampening ϵ for S/D-GGT.

η, ϵ	1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1
η_D, ϵ_D	loss: (1, 1e-6), accuracy: (1, 1e-5)
η_S, ϵ_S	loss: (1, 1e-6), accuracy: (1, 1e-5)

Table 8. Final hyper-parameters for CIFAR-10 / ResNet-20.

SGD	$\eta = 0.1, \mu = 0.9, \gamma = 5e - 4$
D-GGT	$m = 100, \eta = 0.1, \epsilon = 1e - 5, \gamma = 0$
S-GGT	$m = 100, \eta = 0.1, \epsilon = 1e - 5, \gamma = 0$
D-MFAC	$m = 1024, \eta = 1e - 3, \lambda = 1e - 6, \mu = 0, \gamma = 1e - 4$
S-MFAC	$k = 1\%, m = 1024, \eta = 1e - 3, \lambda = 1e - 4, \mu = 0, \gamma = 1e - 4$
LR-MFAC	$\rho = 4, m = 1024, \eta = 1e - 3, \lambda = 1e - 6, \mu = 0, \gamma = 0$

Table 9. Hyper-parameters for SGD and S/D-MFAC for FFCV/ImageNet.

SGD	$\eta = 0.5, \mu = 0.9$
D-MFAC	$\eta = 1e - 3, \lambda = 1e - 6, \mu = 0$
S-MFAC	$\eta = 1e - 3, \lambda = 1e - 7, \mu = 0, k = 1\%$

F.1. Sparse M-FAC vs Dense M-FAC

For M-FAC we will consider that $m = 1024$ and $k = d/100$.

Dense M-FAC stores a matrix $G \in \mathbb{R}^{m \times d}$ of 32 bit floats, having a total of $4md$ bytes, resulting in a theoretical memory footprint of $4096d$ bytes.

On the other hand, Sparse M-FAC stores multiple tensors:

- $\mathcal{I} \in \mathbb{R}^{m \times k}$ of 32 bit integers for the indices

- $\mathcal{V} \in \mathbb{R}^{m \times k}$ of 32 bit floats or 16 bit bfloat16s for the values

- $\xi \in \mathbb{R}^d$ of 32 bit floats for the error feedback

- $r_m \in \mathbb{R}^m$ of 32 bit floats to store the SP result

- $r_d \in \mathbb{R}^d$ of 32 bit floats to store the LCG result

When $\mathcal{V}$ holds 16 bit bfloat16s, the memory footprint for $\mathcal{I}, \mathcal{V}, \xi, r_m, r_d$ altogether is $4mk + 2mk + 4d + 4m + 4d = 0.06md + 8d + 4m = 61.44d + 8d + 4m = 69.44d + 4m \approx 70d$.

When $\mathcal{V}$ holds 32 bit floats, the memory footprint for $\mathcal{I}, \mathcal{V}, \xi, r_m, r_d$ altogether is $4mk + 4mk + 4d + 4m + 4d = 0.08md + 8d + 4m = 81.92d + 8d + 4m = 89.92d + 4m \approx 90d$.

In the end, the memory savings for M-FAC is $\frac{4096d}{90d} \approx 45.5\times$ for float and $\frac{4096d}{70d} \approx 58.5\times$ for bfloat16.

F.2. Sparse GGT vs Dense GGT

For GGT we will consider that $m = 100$ and $k = d/100$.

Dense GGT stores a matrix $G \in \mathbb{R}^{m \times d}$ of 32 bit floats, having a total of $4md$ bytes, resulting in a theoretical memory footprint of $400d$ bytes.

Sparse GGT stores the following tensors:

- $\mathcal{I} \in \mathbb{R}^{m \times k}$ of 32 bit integers for the indices

- $\mathcal{V} \in \mathbb{R}^{m \times k}$ of 32 bit floats for the values

- $\xi \in \mathbb{R}^d$ of 32 bit floats for the error feedback

- $r_m \in \mathbb{R}^m$ of 32 bit floats to store the SP result

The theoretical memory footprint for $\mathcal{I}, \mathcal{V}, \xi, r_m$ altogether is $4mk + 4mk + 4d + 4m = 8mk + 4d + 4m = 0.08md + 4d + 4m = 8d + 4d + 4m = 12d + 4m \approx 12d$. In the end, the memory savings for GGT is $\frac{400d}{12d} \approx 33.3\times$.

G. CUDA Kernels

In this section we describe our efficient CUDA kernels for large models in algorithmic form. We reiterate that M-FAC has two operations that have to be reimplemented when using a sparse ring buffer $G = (\mathcal{I}, \mathcal{V})$, the scalar products (SP) and linear combination of gradients (LCG). In the next paragraphs we provide a brief algorithmic description for SP and LCG.

G.1. SP Kernel

For this operation we have to compute the scalar product between each row in G and a general gradient g. The sparse matrix G is splitted into two matrices, the indices matrix $\mathcal{I} \in \mathbb{N}^{m \times k}$ and the values matrix $\mathcal{I} \in \mathbb{R}^{m \times k}$. We use 1024 thread blocks, each containing 1024 threads and one thread block will process one row. In the algorithm we will consider that $\mathcal{I}$ and $\mathcal{V}$ are 2-dimensional, but in practice they are 1-dimensional. We use $\omega \in \mathbb{R}^m$ to store result of the SP operation in Algorithm 2

Algorithm 2 SP kernel (CUDA)

1: Input: $m, k, g, \mathcal{I}, \mathcal{V}, \omega$
2: mem is a shared memory buffer with T components
3: $\sigma \leftarrow 0 \triangleright$ stores the partial dot product in current thread
4: $Bid \leftarrow$ block id $\triangleright$ gives the row index in $\mathcal{I}$ and $\mathcal{V}$
5: $Tid \leftarrow$ thread id
6: $T \leftarrow$ number of threads in the block
7: $r \leftarrow Bid \triangleright$ row index
8: **for** $j \in \{T_{id} < k, T\} \triangleright j$ from T_{id} to k with step T using coalesced memory access **do**
9: $\quad i \leftarrow \mathcal{I}_{r,j}$
10: $\quad v \leftarrow \mathcal{V}_{r,j}$
11: $\quad \sigma \leftarrow \sigma + v \cdot g_i \triangleright$ accumulate partial dot product
12: **end for**
13: $mem_{Tid} \leftarrow \sigma$
14: parallel_reduce(mem, T) $\triangleright$ logarithmic sum: result is in mem_0
15: **if** $Tid = 0$ **then**
16: $\quad \omega_r \leftarrow mem_0$
17: **end if**

G.2. LCG Kernel

The LCG operation is more complex than the SP operation and requires providing more details. We will start from the compression operator from line 6 of Algorithm 1, which actually applies Top-k on the accumulator a_t, returning a compressed representation c_t (not to be confused with the vector c that stores the coefficients of the linear combination) - the c_t is placed on a specific row of $G = (\mathcal{I}, \mathcal{V})$ at each iteration t.

The Top-k operator is applied in blocks of size B_d (the subscript d means that this block is part of a d-dimensional vector), which is chosen automatically to be the maximum number of floating point values that can fit in the entire shared memory space for a thread block. For example, the A100 GPU from Nvidia has CUDA capability 8.0 and the maximum amount of shared memory per block is 163KB, meaning that we can store $B_d = 41728$ float32 values in the shared memory. The value k is inputed as a percentage at runtime and represents the density percentage of c_t. We convert this percentage to an

actual density in each block B_d. For example, for $k = 1\%$ we will have around $B_k = B_d/100$ (we will further skip some implementation details).

The key detail of our LCG kernel is we use the shared memory of size B_d to accumulate partial linear combinations over slices of size (m, B_d) of the sparse matrix G and then write the result to the output $\omega \in \mathbb{R}^d$. This is possible because we apply Top-k operator in blocks of size B_d and we know the ends of the interval where we write the output to. Each thread block can process multiple slices of size (m, B_d), depending on how large the model is. The pseudocode of the LCG kernel is provided in Algorithm 3.

Algorithm 3 LCG kernel (CUDA)

 1: Input: $B_d, B_k, d, m, k, c, \mathcal{I}, \mathcal{V}, \omega$
 2: $B \leftarrow$ total number of thread blocks
 3: $Bid \leftarrow$ block id $\triangleright$ gives the row index in $\mathcal{I}$ and $\mathcal{V}$
 4: $Tid \leftarrow$ thread id
 5: $T \leftarrow$ number of threads in the block
 6: mem is a shared memory buffer with T components
 7: $mem \leftarrow 0 \triangleright$ initialize shared memory (the LCG accumulator) with zero
 8: synchronize threads $\triangleright$ wait for all threads to finish initializing
 9: $n_{B_d} \leftarrow \lceil \frac{d}{B_d} \rceil \triangleright$ how many blocks of size B_d we have
10: $workload \leftarrow \lceil \frac{n_{B_d}}{B} \rceil \triangleright$ how many blocks of size B_d each thread block processes
11: **for** $i \in \{0 < workload\} \triangleright$ iterate through all slices (m, B_k) (workload) for the curent thread block **do**
12:      $start_d \leftarrow B_d \cdot B_{id} \cdot workload + B_d \cdot i \triangleright$ where slice starts in g
13:      $end_d \leftarrow min(d, start_d + B_d) \triangleright$ where slice ends in g
14:      $start_k \leftarrow B_k \cdot B_{id} \cdot workload + B_k \cdot i \triangleright$ where slice starts in G
15:      $end_k \leftarrow min(B_k, start_k + B_k) \triangleright$ where slice ends in G
16:      **for** $row \in \{0 < workload\} \triangleright$ iterate all rows in the slice (m, B_k) between $(start_k, end_k)$ **do**
17:          **for** $col \in \{start_k + T_{id}, min(end_k, end_d), T\} \triangleright$ jump T steps (coalesced memory access) **do**
18:              $i \leftarrow \mathcal{I}_{r,col} \triangleright$ read index
19:              $v \leftarrow \mathcal{V}_{r,col} \triangleright$ read value
20:              $mem_{i-start_d} += c_r \cdot v \triangleright$ multiply the coefficient of current row with value
21:          **end for** $\triangleright$ finished processing one row
22:          synchronize threads
23:      **end for** $\triangleright$ finished processing all rows
24:      $\triangleright$ finished processing a slice of size (m, B_k), now write result
25:      **for** $i \in \{T_{id} < min(B_d, d - start_d), T\} \triangleright$ write result (coalesced memory access) **do**
26:          $\omega_{i+start_d} \leftarrow mem_i \triangleright$ write result from mem to output ω
27:          $mem_i \leftarrow 0 \triangleright$ zerorize shared memory to prepare for the next step
28:      **end for**
29:      synchronize threads
30: **end for**

H. Sparse GGT.

In principle, the same sparsity format can be used for GGT as in the case of M-FAC. However, we would like to provide a slightly different approach just to show that an existing sparsity format is not completely efficient with respect to memory since they store redundant information. Our Sparse GGT uses the same sparsity format as S-MFAC and here we want to emphasize some drawbacks of COO format.

At a high level, we wish to leverage sparsity in the logical gradient history matrix $G \in \mathbb{R}^{m \times d}$. For this, we store it in the COOrdinate list sparse format, which requires storing matrix $\mathcal{V} \in \mathbb{R}^{m \times k}$ for values, matrix $\mathcal{R} \in \mathbb{Z}^{m \times k}$ for rows and $\mathcal{C} \in \mathbb{Z}^{m \times k}$ for columns. All these matrices can be stored in vector format, with dimension $m \cdot k$. This will be compatible with standard sparse operations, such as the Sparse Matrix-Matrix (SPMM) method in the PyTorch Sparse library (Fey et al., 2018).

We will use the tuple $(\mathcal{R}, \mathcal{C}, \mathcal{V})$ to refer to the sparse buffer G. We distinguish between $\mathcal{V}$, which is the internal state of the

sparse buffer (holding the k most significant values from a_t) and $\mathcal{V}_t$, which is the output of Top-k operator at a single step. *With slight abuse of notation, we call the compressed accumulator* $c_t = (\mathcal{V}_t, \mathcal{I}_t)$ *a sparse gradient, even though it is an accumulation of previous gradients.* The matrix $\mathcal{C}$ is created in such a way that each row contains the index of that row (e.g. [0,...,0; 1,...,1;...; m-1,...,m-1]) and is represented as a single vector where the rows are concatenated (here is where we have redundancy, e.g. duplicated data - the values in this tensor never change).

Let $i \in \{0, ..., m-1\}$ be an index indicating the location where the new gradient will be placed in the buffer $(\mathcal{R}, \mathcal{C}, \mathcal{V})$. At each step, we receive the accumulator a_t, the most significant k values $\mathcal{V}_t$ and their corresponding indices $\mathcal{I}_t$. The expression $a_t[\mathcal{I}_t] \in \mathbb{R}^d$ will represent a "densified" version of a sparse vector, which contains zeros at indices $j \notin \mathcal{I}_t$ (sparse tensor which also contains the zeros - does not provide any memory reduction).

We integrate the values and indices in the internal buffers $\mathcal{V}$ and $\mathcal{I}$ in row i using the indices α and β. Since the sparse $G \in \mathbb{R}^{m \times k}$ and we have to compute the scalar product $\delta = G^T x$, we can easily transpose the sparse G by swapping the rows and columns in the SPMM call. The scalar products matrix $G^T G$ is updated by overwriting the ith row and column by δ. At this point we can perform eigenvalue decomposition of $G^T G$ and finally compute the update direction u_t by preconditioning the sparse accumulator $a_t[\mathcal{I}_t]$. The full procedure is presented in Algorithm 4.

GGT results for CIFAR-10/ResNet-20. Our experimental setup covers ResNet-20 (He et al., 2016) ($d = 0.27$M) for CIFAR-10 (Krizhevsky & Hinton, 2009). We train models from scratch using SGD with momentum, dense and sparse GGT (99% sparsity) and M-FAC and low-rank M-FAC (rank 4). We decouple the weight decay γ from the gradient, and apply it in the optimizer, which allows us to compare the loss values. The results in Table 10 show validation accuracy on the ResNet20/CIFAR-10 experiment. We observe that 1) Dense(D) M-FAC reaches similar accuracy to SGD, whereas Dense(D) GGT achieves lower accuracy in this task; 2) Sparse(S) M-FAC reaches the same accuracy as the dense variant (and SGD); 3) Low-Rank(LR) M-FAC with rank 4 reaches ¡ 0.8% lower accuracy relative to the best results.

Table 10. Results for ResNet-20 on CIFAR10. The first row reports the training loss and second row validation accuracy. The number in parentheses indicate stdev (for training loss, this is within 0.001).

	SGD	D-MFAC	S-MFAC	D-GGT	S-GGT	LR-MFAC
Loss	0.032	0.017	0.017	0.108	0.097	0.021
Acc.	92.16 (.10)	92.12 (.13)	92.25 (.21)	87.77 (.22)	87.73 (.34)	91.43 (.30)

Algorithm 4 Detailed Sparse GGT Implementation

1: Variables: $\mathcal{V} \leftarrow 0 \in \mathbb{R}^{mk}, \mathcal{R} \leftarrow 0 \in \mathbb{Z}^{mk}, i \leftarrow 0, G^T G \leftarrow 0 \in \mathbb{R}^{m \times m}$
2: Operators: $\mathcal{C} \leftarrow vec([r \cdot 1_k]_{r=0}^{m-1}) \in \mathbb{Z}^{mk}$

3: **function** SPARSEGGT-STEP($t, a_t \in \mathbb{R}^d, \mathcal{V}_t \in \mathbb{R}^k, \mathcal{I}_t \in \mathbb{Z}^k$)
4: $\quad \alpha \leftarrow i \cdot k, \ \beta \leftarrow \alpha + k, \ \mathcal{V}_{\alpha:\beta} \leftarrow \mathcal{V}_t, \ \mathcal{R}_{\alpha:\beta} \leftarrow \mathcal{I}_t \vartriangleright$ integrate values and indices
5: $\quad \delta \leftarrow$ SPMM($Sparse(r = \mathcal{C}, c = \mathcal{R}, v = \mathcal{V}), Dense(a_t[\mathcal{I}_t])) \vartriangleright$ compute $G^T a_t[\mathcal{I}_t]$
6: $\quad row_i(G^T G) \leftarrow \delta \vartriangleright$ update dot products in row and col i
7: $\quad col_i(G^T G) \leftarrow \delta \vartriangleright$ update dot products in row and col i
8: $\quad G^T G \leftarrow V \Sigma_m^2 V^T \vartriangleright$ eigenvalue decomposition
9: $\quad i \leftarrow (i+1) \mod m \vartriangleright$ index update
10: $\quad U_m \leftarrow$ SPMM($Sparse(r = \mathcal{R}, c = \mathcal{C}, v = \mathcal{V}), Dense(V \sqrt{\Sigma_m}^\dagger)$)
11: $\quad u_t \leftarrow \frac{1}{\epsilon}(a_t[\mathcal{I}_t]) + U_m \left[(\Sigma_m + \epsilon I_m)^{-1} - \frac{1}{\epsilon} I_m \right] U_m^T (a_t[\mathcal{I}_t]) \vartriangleright$ compute GGT direction
12: **return** u_t

I. Compressing Preconditioners via Low-Rank Approximation and Error Feedback

In this section, we discuss an alternative approach for compressed preconditioners, via low-rank compression of the gradients. Specifically, we will implement the COMPRESS step in Algorithm 1 via a low-rank approximation of the gradient, using an efficient variant of power iteration (Vogels et al., 2019). Then, we consider the M-FAC algorithm, and reformulate its operations to leverage low-rank matrix operations for memory savings. We start by describing how the algorithm works a

single layer, and then present the global algorithm, which computes preconditioning for the whole model.

Low-Rank M-FAC for a Single Layer. Assume that the gradient at a given layer of the model is represented by the s-dimensional tensor $g \in \mathbb{R}^{p_1 \times p_2 \times \ldots \times p_s}$. To compress the tensor, we use a variant of power iteration (Vogels et al., 2019) by firstly unfolding the tensor into a matrix $\tilde{g} \in \mathbb{R}^{p_1 \times p_{2:s}}$, where $p_{2:s} = \prod_{i=2}^{s} p_i$. This matrix is then iteratively multiplied by a rank ρ matrix $Q \in \mathbb{R}^{p_{2:s} \times \rho}$, which is reused from the previous iteration, to obtain the left decomposition matrix $P \in \mathbb{R}^{p_1 \times \rho}$. After orthogonalization of P, the updated right matrix Q is obtained from the relation $\tilde{g} = PQ^T$. Notice that P and Q have $(p_1 + p_{2:s})\rho$ elements which can be significantly smaller than $p_1 \cdot p_{2:s}$ for smaller ranks ρ. This compression procedure is outlined in Algorithm 5.

In order to "introduce" the current gradient g into the M-FAC preconditioner data structure, we firstly need to compute an inner product between previously stored gradients and g (see Equation 2). In matrix representation the product between two gradients corresponds to the Frobenius inner product. Specifically, for a given low-rank representation of gradient matrices $\tilde{g}_i = P_i Q_i^T$ and $\tilde{g}_j = P_j Q_j^T$, the Frobenius product can be written as $\text{Tr}(\tilde{g}_i^T \tilde{g}_j) = \text{Tr}(Q_i P_i^T P_j Q_j^T) = \text{Tr}((P_j^T P_i)^T (Q_j^T Q_i))$. Notice that matrix products $P_j^T P_i$ and $Q_j^T Q_i$ are only of size ρ by ρ, so we compute them explicitly and then perform summation over their element-wise products to get the final result. As full gradients do not appear in this expression, we store just left and right gradients P_i and Q_i and update the M-FAC inner product matrix as needed.

The final preconditioned gradient update is computed as a weighted sum of previous gradients. The weights are computed via dynamic programming on top of the inner product values as in the original M-FAC. The final weighted sum of gradients is obtained by sequentially adding gradients reconstructed from their low-rank representation $g_i = P_j Q_j^T$.

Algorithm 5 Power iteration compression algorithm (Vogels et al., 2019)

1: **PowerCompress**(gradient tensor g, previous Q) ▷ applied for each layer
2: $\tilde{g} \leftarrow g.\text{Reshape}(p_1, p_{2:s})$
3: $P \leftarrow \tilde{g}Q$
4: $P \leftarrow \text{Orthogonalize(P)}$
5: $Q \leftarrow \tilde{g}^T P$
6: **return** (P, Q)

Algorithm 6 Low Rank M-FAC preconditioning

1: INPUT: per-layer compressed $((P^1, Q^1), \ldots, (P^L, Q^L))$ gradients
2: STATE: per-layer ring buffers G^ℓ with compressed gradient history; full inner products $G^T G$
3: **Low Rank M-FAC**$(P^1, Q^1), \ldots, (P^L, Q^L)$
4: $\delta \leftarrow$ zero vector of size m ▷ new full inner product
5: **for** each layer $\ell \in \{1, 2, ..., L\}$ ▷ aggregate inner products across the whole model **do**
6:     Update G^ℓ with current gradient (P^ℓ, Q^ℓ)
7:     **for** each time step $\tau \in \{t - m + 1, ..., t\}$ **do**
8:         $\delta_{t-\tau} \leftarrow$ low-rank inner product $\text{Tr}((g_\tau^\ell)^T (P^\ell (Q^\ell)^T))$
9:     **end for**
10: **end for**
11: Update row and column in $G^T G$ from δ
12: **M-FAC:** compute summation weights $w \in \mathbb{R}^m$ from $G^T G$
13: **for** each layer $\ell \in \{1, 2, ..., L\}$ **do**
14:     $u_t^\ell \leftarrow \epsilon^{-1} P^\ell (Q^\ell)^T + \sum_{\tau=t-m+1}^{t} w_\tau g_\tau$ ▷ aggregate gradients from the ring buffer
15: **end for**
16: **return** vector of per-layer updates u_t

Global Low-Rank M-FAC. To obtain an equivalent algorithm to the original M-FAC, independent per-layer steps are not sufficient as we would lose cross-layer correlation information. Instead, we use linearity of the inner product to compute the final result efficiently. In the original version of M-FAC, a current complete gradient from a L-layered model is represented as a concatenation of flattened per-layer gradients $\widehat{g} = \begin{pmatrix} \widehat{g}^1 & \widehat{g}^2 & \ldots & \widehat{g}^L \end{pmatrix}^T$, where $\hat{g}^i$ is the gradient from

layer i. In this, representation an inner product between two gradients $\widehat{g}_i$ and $\widehat{g}_j$ can be expressed in a block form as $(\widehat{g}_i, \widehat{g}_j) = \sum_{\ell=1}^{L} (\widehat{g}_i^\ell, \widehat{g}_j^\ell)$. This dot product is then used to update the inner product matrix $G^T G$.

In the context of our efficient scheme for single-layer low-rank update, we replace each inner product block by its low-rank counterpart to obtain the equivalent values. Recall that the computation of summation weights in the M-FAC needs access only to these inner products, so this part of the algorithm remains the same. The final gradient update is computed per-layer and we rely on linearity of the sum again to share the summation weights and dampening coefficient ϵ between layers. For more details refer to the pseudocode in Algorithm 6.

Results for Low-Rank M-FAC. We present and discuss the experiments for Low-Rank M-FAC on CIFAR-10/ResNet-20 in Table 10 and hyper-parameters in Appendix E. Next, we present an ablation study of low rank decomposition on both CIFAR-10/ResNet-20 and GLUE/MNLI for BERT-TINY.

To determine how the size of the decomposition rank ρ affects the effectiveness of the low rank M-FAC algorithm, we perform a series of experiments on ResNet-20 and BERT models. For ResNet-20 we use CIFAR-10 dataset and the same parameters as in the main experiments. For BERT family we use TINY, MINI and BASE variants and evaluate them on MNLI task from GLUE benchmark. For both setups we vary the rank from 1 to 8 and report the final test accuracy in the Table 11.

We observe that the decomposition rank significantly affects smaller BERT models but barely influences BERT-BASE and RN-20 model results. While the exact reason for such behaviour is not obvious, we hypothesise that for RN-20 the decomposition is effective even at rank 1 due to gradient tensors already having a small first dimension, resulting in small compression error. We also note that Rank 8 BERT-BASE run diverged, requiring more investigation into the stability of the training.

Table 11. Test accuracy on CIFAR-10 dataset for ResNet-20 (RN-20) and on MNLI for BERT-TINY, -MINI and -BASE, trained with Low-Rank M-FAC with different decomposition ranks ρ.

rank ρ	RN-20	TINY	MINI	BASE
1	91.40	57.34	66.88	84.42
2	91.19	63.49	68.76	84.44
4	91.43	66.44	70.93	84.45
8	91.13	67.90	72.22	—

J. Rank and Density Connection for Similar Memory Usage

To apply a low rank matrix decomposition we cast s-dimensional gradient tensor at each layer of the model into a matrix by keeping the first dimension intact and flattening out the others. The choice of rank to achieve equivalent memory savings is dependent on the model structure as each layer dimensions are different.

We would like to provide a theoretical approach to compute the corresponding gradient density $k = \Delta d$ (d is the model size) such that S-MFAC has the same theoretical memory footprint as the LR-MFAC with rank ρ fixed to a specific value, such as one in $1, 2, 4, 8, 16$. Let n be the total number of layers, which are denoted by $L_1, ..., L_n$ and $p_1^{(i)}, ..., p_{n_i}^{(i)}$ be the dimensions of L_i, where n_i is the number of dimensions for L_i. Moreover, the total number of elements in L_i is $\prod_{j=1}^{n_i} p_j^{(i)}$. For example, if $L_1 \in \mathbb{R}^{100 \times 200 \times 300}$, we will have $n_i = 3$ and $(p_1^{(1)}, p_2^{(1)}, p_3^{(1)}) = (100, 200, 300)$ and the total number of elements is $100 \cdot 200 \cdot 300 = 6M$.

Note that the model size $d = \sum_{i=1}^{n} \prod_{j=1}^{n_i} p_j^{(i)}$. As described in Appendix E of the paper, LR-MFAC with rank ρ reduces the size of L_i from $\prod_{j=1}^{n_i} p_j^{(i)}$ to $(p_1^{(i)} + \prod_{j=2}^{n_i} p_j^{(i)}) \cdot \rho$. By extrapolating this to all layers, the total size of a full gradient $g_t \in \mathbb{R}^d$ will be compressed to the size $\rho \cdot \sum_{i=1}^{n} (p_1^{(i)} + \prod_{j=2}^{n_i} p_j^{(i)})$. Suppose we have already set ρ for LR-MFAC and we would like to determine the corresponding fraction Δ of weights such that the gradient density $k = \Delta \cdot d$ for S-MFAC such that LR-MFAC and S-MFAC require the same amount of memory.

$$\Delta \cdot d = \rho \left[\sum_{i=1}^{n} \left(p_1^{(i)} + \prod_{j=2}^{n_i} p_j^{(i)} \right) \right] \implies \Delta = \frac{\rho}{d} \sum_{i=1}^{n} \left(p_1^{(i)} + \prod_{j=2}^{n_i} p_j^{(i)} \right)$$

We apply this analysis to ResNet-20 (272k params) with 10 classes and present the values in Table 12. We can see that if we set $\Delta = 2.98\%$, S-MFAC will have the same memory usage as LR-MFAC with $\rho = 1$. The experiments in the main body of the paper show that we can reach good results using $\Delta = 1\%$, which results in a lower memory footprint than LR-MFAC. Table 8 in Appendix shows an ablation on rank ρ for ResNet-20 and BERT models.

In the Table 12 below we show the optimal $k = \Delta \cdot d$ such that LR-MFAC with rank ρ has the same memory footprint with S-MFAC.

Table 12. The rank ρ for Low-Rank M-FAC and corresponding gradient density Δ for Top-Δ M-FAC such that both methods have similar memory usage.

ρ	1	2	4	8	16
Δ	2.98%	5.97%	11.93%	23.86%	47.72%

K. Theoretical Discussion

In this section, we discuss theoretical justifications for the proposed method.

The Diagonal Case. First, in the *diagonal case*, described in Algorithm 7, we can adapt a recent argument by Li, Karimi, and Li (Li et al., 2022) to show that the method should guarantee standard rates of $O(1/\sqrt{T} + \sigma^2/\sqrt{T} + d/T)$ under standard assumptions on the objective, that is 1) smoothness; 2) unbiased and bounded stochastic gradients; and 3) bounded variance. Specifically, obtaining this result follows by specializing Corollary 1 in Li et al. (2022), specifically adapting their AMSGrad instance to Algorithm 7, and adjusting the constants correspondingly.

Algorithm 7 Diagonal AdaGrad with Error Feedback

$\xi_0 = 0_d \in \mathbb{R}^d \triangleright$ initialize error
$G_0 = 0_d \in \mathbb{R}^{d \times d} \triangleright$ initialize diagonal
for each step $t \in \{1, 2, ...T\}$ **do**
  $g_t \leftarrow \nabla_\theta L(\theta_t)$
  $a_t \leftarrow \xi_{t-1} + g_t \triangleright$ add gradient to previous error
  $c_t \leftarrow \text{TOPK}(a_t) \triangleright$ compress the accumulator
  $\xi_t \leftarrow a_t - c_t \triangleright$ update the error
  $G_t \leftarrow G_{t-1} + diag(c_t^2) \triangleright$ update diagonal
  $\theta_{t+1} \leftarrow \theta_t - \eta_t diag(G_t)^{-1/2} c_t$
end for

Discussion of the Full-Matrix Case. In the highly-complex full-matrix case, further investigation is necessary to provide a full theoretical characterization of convergence. We expect that this will be very challenging since even the analysis of standard SGD with error feedback required significant technical advances (Alistarh et al., 2018; Karimireddy et al., 2019), and generally the convergence of full-matrix preconditioner variants is less well understood.

Specifically, our discussion will consider the convergence of an "intermediate" version of the Sparse GGT algorithm, which we call Scaled Sparse GGT. The only difference from the Sparse GGT algorithm in Section 3.1 is the fact that we perform rescaling of the accumulated dense descent direction a_t before and after taking the Top-k operation. (Intuitively, this is required so that the accumulator a_t is considered in the "correct basis" at the current iteration, as per the standard AdaGrad analysis.) Experimentally, we find that the practical convergence of the scaled and un-scaled variants is nearly identical in the diagonal case.

More precisely, denote by H_t the Sparse M-FAC approximation for the preconditioner at step t. Then, assuming we are interested in an implementation of full-matrix Adagrad, we let the scaled Top-k operator T_{H_t} be defined as

$$T_{H_t}(v) = H_t^{1/2} \text{TopK}\left(H_t^{-1/2} v\right).$$

Assume that we use this scaled Top-k operator instead of the standard one in the GGT implementation. We begin with the following two technical results, whose proofs are immediate.

Lemma K.1. *Given a PSD matrix H, let the operator T_H defined as*

$$T_H(v) = H^{1/2} TopK\left(H^{-1/2}v\right)$$

Then it always holds that

$$\|T_H(v)\|^2_{H^{-1}} \geq \frac{k}{n} \|v\|^2_{H^{-1}}$$

The second technical lemma will allow us to use an appropriate potential function.

Lemma K.2. *Let A be a PSD matrix and consider a rank-2 update $A' = A + uu^\top + vv^\top$. Then it always holds that*

$$\ln \det(A') \geq \ln \det(A) + \ln\left(1 + \frac{1}{2}\|u\|^2_{A^{-1}} + \frac{1}{2}\|v\|^2_{A^{-1}}\right).$$

Next, we can re-write our algorithm's iterations as follows:

$$x_{t+1} = x_t - H_t^{-1} T_{H_t}(m_t + g_t) \simeq x_t - H_t^{-1} \underbrace{T_{H_t}\left(H_t H_{t-1}^{-1} m_t + g_t\right)}_{\widetilde{g}_t}$$

$$m_{t+1} = (m_t + g_t) - T_{H_t}(m_t + g_t) \simeq \left(H_t H_{t-1}^{-1} m_t + g_t\right) - \underbrace{T_{H_t}\left(H_t H_{t-1}^{-1} m_t + g_t\right)}_{\widetilde{g}_t},$$

where in the approximation we only needed to assume that the Hessian estimate is stable from one iteration to another, i.e. that $H_t H_{t-1}^{-1} m_t \simeq m_t$.

In this context, our Hessian approximation is updated via the rule:

$$H_0 = \delta I$$

$$H_t = H_{t-1} + 2\eta T_{H_t}(m_t + g_t) T_{H_t}(m_t + g_t)^\top \simeq H_{t-1} + \eta \widetilde{g}_t \widetilde{g}_t^\top + \eta T_{H_t}(g_t) T_{H_t}(g_t)^\top.$$

Under these approximations, needed to consider the correct "basis" for the updates, we can examine the change in potential function involving the accumulation:

$$\frac{1}{2}\left\|\left(x_{t+1} - H_t^{-1} m_{t+1}\right) - x^*\right\|^2_{H_t} = \frac{1}{2}\left\|\left(\left(x_t - H_t^{-1}\widetilde{g}_t\right) - H_t^{-1}\left(H_t H_{t-1}^{-1} m_t + g_t - \widetilde{g}_t\right)\right) - x^*\right\|^2_{H_t}$$

$$= \frac{1}{2}\left\|\left(x_t - H_{t-1}^{-1} m_t\right) - x^* - H_t^{-1} g_t\right\|^2_{H_t}$$

$$= \frac{1}{2}\left\|\left(x_t - H_{t-1}^{-1} m_t\right) - x^*\right\|^2_{H_t} + \frac{1}{2}\|g_t\|^2_{H_t^{-1}} - \left\langle g_t, \left(x_t - H_{t-1}^{-1} m_t\right) - x^*\right\rangle$$

from where we conclude that

$$\langle g_t, x_t - x^*\rangle = \frac{1}{2}\left\|\left(x_t - H_{t-1}^{-1} m_t\right) - x^*\right\|^2_{H_t} - \frac{1}{2}\left\|\left(x_{t+1} - H_t^{-1} m_{t+1}\right) - x^*\right\|^2_{H_t} + \frac{1}{2}\|g_t\|^2_{H_t^{-1}} + \left\langle g_t, H_{t-1}^{-1} m_t\right\rangle$$

where we may use the upper bound

$$\left\langle g_t, H_{t-1}^{-1} m_t\right\rangle \leq \frac{1}{2}\|g_t\|^2_{H_{t-1}^{-1}} + \frac{1}{2}\|m_t\|^2_{H_{t-1}^{-1}} \leq \frac{d}{k}\left(\frac{1}{2}\|\widetilde{g}_t\|^2_{H_{t-1}^{-1}} + \frac{1}{2}\|T_{H_t}(g_t)\|^2_{H_{t-1}^{-1}}\right)$$

for which the latter inequality follows from Lemma K.1. At this point, the entire analysis would go through just as in the standard full-matrix Adagrad, except for the fact that the term $\left\langle g_t, H_{t-1}^{-1} m_t\right\rangle$ may be very large whenever H_{t-1} fails to align well with any of m_t and g_t.

On the other hand, if we are willing to assume that these quantities are always well-aligned, in the sense that there exists an ϵ such that

$$\|\widetilde{g}_t\|^2_{H^{-1}_{t-1}} + \|T_{H_t}(g_t)\|^2_{H^{-1}_{t-1}} \leq \frac{k\epsilon}{2d}$$

then we can obtain via a standard analysis with iterate bound R_∞ that the regret $\mathcal{R}_T$ is bounded as

$$\mathcal{R}_T \leq \frac{1}{2}R^2_\infty \left(\delta d + \eta \sum_{t=0}^{T-1} \left(\|\widetilde{g}_t\|^2_2 + \|T_{H_t}(g_t)\|^2_2\right)\right) + T\frac{\epsilon}{2} + \frac{d}{k}\frac{1}{2\eta} \ln \det \left(I + \frac{\eta}{\delta} \sum_{t=0}^{T-1} \left(\widetilde{g}_t\widetilde{g}_t^\top + T_{H_t}(g_t)T_{H_t}(g_t)^\top\right)\right)$$

$$\leq \widetilde{O}\left(R_\infty \sqrt{\sum_{t=0}^{T-1} \left(\|\widetilde{g}_t\|^2_2 + \|T_{H_t}(g_t)\|^2_2\right) \cdot d \cdot \frac{d}{k}} + T\epsilon\right)$$

Thus, provided that the truncated gradient norms are $O(1)$ this upper bound is

$$\epsilon + \widetilde{O}\left(\frac{R_\infty \sqrt{T \cdot d \cdot \frac{d}{k}}}{T}\right)$$

which follows the non-truncated version, after increasing the number of iterations by a worst-case factor of $\frac{d}{k}$. In turn, this factor is natural, and also occurs in the worst-case analyses of error feedback without pre-conditioning, e.g. (Alistarh et al., 2018).