M-FAC: Efficient Matrix-Free Approximations of Second-Order Information

Anonymous Author(s)
Affiliation
Address
email

Abstract

Efficiently approximating local curvature information of the loss function is a key tool for optimization and compression of deep neural networks. Yet, most existing methods to approximate second-order information have high computational or storage costs, which limits their practicality. In this work, we investigate matrix-free, linear-time approaches for estimating Inverse-Hessian Vector Products (IHVPs) for the case when the Hessian can be approximated as a sum of rank-one matrices, as in the classic approximation of the Hessian by the empirical Fisher matrix. We propose two new algorithms: the first is tailored towards network compression and can compute the IHVP for dimension d, if the Hessian is given as a sum of m rank-one matrices, using $O(dm^2)$ precomputation, $O(dm)$ cost for computing the IHVP, and query cost $O(m)$ for any single element of the inverse Hessian. The second algorithm targets an optimization setting, where we wish to compute the product between the inverse Hessian, estimated over a sliding window of optimization steps, and a given gradient direction, as required for preconditioned SGD. We give an algorithm with cost $O(dm + m^2)$ for computing the IHVP and $O(dm + m^3)$ for adding or removing any gradient from the sliding window. These two algorithms yield state-of-the-art results for network pruning and optimization with lower computational overhead relative to existing second-order methods.

1 Introduction

Given the recent success and increasing impact of deep learning, there has been significant work on improving the fundamental technical tools underpinning its progress. One such tool is the ability to estimate the local geometry of the loss function for deep models, which often comes in the form of estimates for the second-order (Hessian) information. Such information is critical in several settings, such as neural network optimization and pruning.

Directly using Hessian information in the context of deep learning is infeasible: for example, just storing the Hessian matrix for the standard ResNet50 model [12] would occupy 2.5 Petabytes. These constraints have inspired significant work on efficient numerical approximations of the Hessian for deep neural networks, such as the line of work on the K-FAC approximation [16, 28, 5, 41], or efficient block-wise approximations [39, 33]. One of the classic approaches, which we focus on in this paper, is the *empirical Fisher* [11, 3, 4] approximation to the Hessian, written as:

$$\mathbf{H} \simeq \widehat{\mathbf{F}} = \frac{1}{N} \sum_{i=1}^{N} \nabla \ell_i \cdot \nabla \ell_i^\top, \tag{1}$$

where N is the number of samples, $\nabla \ell_i$ denotes the gradient w.r.t. the ith sample at the given point, and $\cdot$ is the outer product of individual gradients, which we view as column vectors.

The empirical Fisher approximation is fairly standard, e.g. [11, 3, 4], and has been recognized to be useful in a variety of practical settings where exact estimation of the Hessian or of the true Fisher information matrix is not feasible. At the same time, there is still active research in the community on the conditions for its applicability [26, 19, 33, 36]. A useful property of this approximation is that it also allows to estimate the *inverse* of the Hessian, which is essential in many applications.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

Specifically, the fact that the empirical Fisher can be written as a sum of rank-one matrices allows the use of the Woodbury-Sherman-Morrison inversion formula [38] to exactly compute its inverse, by recursively integrating terms corresponding to each gradient into the inverse. (Please see Equation 2 for an exact derivation.) This approach was independently proposed by [11, 3], for pruning and optimization, respectively, where it was validated on small networks, with hundreds of weights.

The idea was adapted to deep neural networks (DNNs) by [33], by approximating the inverse in *small diagonal blocks*. The authors show improved approximation quality for the Hessian inverse, and that this leads to state-of-the-art pruning results in terms of accuracy. Yet, this approach is limited by the block-wise approximation: for block size B and dimension d, it requires $\Theta(Bdm)$ time to recursively build the block-wise Fisher approximation using m gradients, and $\Theta(Bd)$ time and memory for computing the Inverse-Hessian-Vector-Products necessary for estimating pruning statistics. Clearly, the ideal $B = d$ case is still intractable at scale, and generally it is still unknown whether linear-time algorithms are possible for computing IHVPs in this context.

Contribution. We resolve this question by introducing two efficient algorithms for computing Inverse-Hessian-Vector-Products (IHVPs) under the empirical Fisher approximation, with computational and storage costs that are *linear* in the dimension d of the model, without the need for block-wise approximations. Concretely, we provide exact *matrix-free* algorithms to compute products of the form $\widehat{\mathbf{F}}^{-1}\mathbf{v}$, where $\widehat{\mathbf{F}}^{-1}$ is the inverse empirical Fisher, and $\mathbf{v}$ is an arbitrary vector. We show that these algorithms can be implemented efficiently at scale, and that they can match or improve the state-of-the-art results for both neural network pruning and optimization.

The Static Algorithm. Our first algorithm assumes a *static* scenario, which is standard in neural network pruning: we are given a fully-trained model $\theta^\star$, for which we wish to estimate IHVPs and diagonal elements of the inverse Hessian, in order to determine the "optimal" pruning update using e.g. the Optimal Brain Surgeon (OBS) framework [22, 11]. For this, we first compute m gradients at $\theta^\star$, which we will use to estimate IHVPs via the empirical Fisher and compute pruning statistics.

The main idea is that, since we only wish to compute products between the inverse and an arbitrary vector (IHVPs), we can rewrite the Woodbury recursion such that we work *exclusively with individual vectors and scalars*, and never with full $d \times d$ matrices. Given model dimension d and m gradients, each defining a rank-one component of the empirical Fisher, the algorithm uses $O(dm^2)$ pre-computation time, and will have $O(dm)$ cost for *exactly* computing the IHVP. Further, we can specialize the algorithm to directly query elements of the Hessian inverse, at a cost of $O(m)$ time per element. This provides efficient, linear-in-d implementations for the OBS pruning framework.

The Dynamic Algorithm. Next, we adapt this approach to SGD optimization, to precondition stochastic gradients by our estimate of the inverse Hessian. In this case, we use the classic idea of bootstrapping the approximation by leveraging previous gradients: the preconditioner at time t is built from gradients obtained during a "sliding window" over the last $\Delta \geq 1$ optimization steps. This requires a *dynamic* representation, allowing addition and removal of gradients *without full recomputation* of second-order statistics.

Our main technical contribution is showing that this can be achieved, with approximately $O(dm)$ time and space complexity. The key idea is that, for any set $(\nabla \ell_j)_{j=1}^m$ of m gradients, and any vector $\mathbf{v}$, it is possible to represent the corresponding IHVP estimate $\widehat{\mathbf{F}}_m^{-1}\mathbf{v}$ as a linear combination of terms corresponding to individual gradients $\nabla \ell_j$ and $\mathbf{v}$, of the form $\widehat{\mathbf{F}}_m^{-1}\mathbf{v} = \lambda^{-1}\mathbf{v} - \sum_{j=1}^m c_j^m \nabla \ell_j$, where $\lambda > 0$ is a dampening constant.

Crucially, we ensure that the coefficients c_j^m can be computed just via dot products $\nabla \ell_j^\top \nabla \ell_i$, where $i \leq j$ in the ordering, and $\nabla \ell_j^\top \mathbf{v}$. Then, to replace a given gradient $\nabla \ell_i$ from this representation, we just have to compute m scalar products with the new gradient vector (and update some intermediate information). Hence, the entire update operation has computational cost $O(dm + m^3)$ for adding or removing any gradient from the sliding window, and $O(dm + m^2)$ for computing the IHVP.

Implementation and Experiments. We provide an efficient vectorized implementation for the above algorithms, called *M-FAC*, for *Matrix-Free Approximate Curvature*, as well as pruning and an optimization code in Pytorch [31]. Our implementation introduces several additional optimizations, in particular GPU acceleration, and minimizes the cost of memory transfer between the GPU and main memory via a gradient paging mechanism. For pruning operations, our implementation provides order-of-magnitude improvements over the block-wise approximation of [33] for classic pruning benchmarks such as ResNet50 and MobileNet on the ImageNet dataset. In turn, this allows us to obtain more accurate pruned models by exploring higher numbers of gradients and larger block sizes.

What is more, our preconditioned SGD can be competitive with state-of-the-art optimizers for deep learning in terms of accuracy, and has computational overhead of 5%–55% relative to vanilla SGD.

2 Preliminaries and Related Work

General Definitions. We now briefly introduce the setting and notation; we refer the reader to standard texts [26, 24, 4] for a complete introduction. We start from the standard setting in which we are given a dataset $\mathcal{D} = (\mathbf{x}_i, \mathbf{y}_i)_{i=1}^N$, and wish to identify a d-dimensional model $\theta \in \mathbb{R}^d$ to minimize an empirical loss $L : \mathbb{R}^d \to \mathbb{R}$, defined as $L(\theta) = \frac{1}{N}\sum_{i=1}^N \ell\big(\mathbf{y}_i, f(\mathbf{x}_i; \theta)\big)$. For a twice-differentiable loss L, the Hessian is the matrix of second-order derivatives of L w.r.t. θ, i.e. $\mathbf{H} = \nabla^2_\theta L$. In the *probabilistic* view, each input example $\mathbf{x}$ has some probability of being assigned a given label $\mathbf{y}$. Given input examples $\mathbf{x}$ drawn from a distribution $Q_\mathbf{x}$ and corresponding outputs drawn from a conditional distribution $Q_{\mathbf{y}|\mathbf{x}}$, the goal is to minimize the distance between the target joint distribution $Q_{\mathbf{x},\mathbf{y}} = Q_\mathbf{x}\, Q_{\mathbf{y}|\mathbf{x}}$, and a learned joint distribution $P_{\mathbf{x},\mathbf{y}}(\theta)$, where θ is the model.

The Fisher Matrix. Assuming the probabilistic view, it can be shown that the Fisher information matrix $\mathbf{F}$ of the model's joint distribution satisfies $\mathbf{F} = \mathrm{E}_{P_{\mathbf{x},\mathbf{y}}}\left[-\nabla^2_\theta \log p_\theta(\mathbf{x}, \mathbf{y})\right]$. If the model's output conditional distribution matches the conditional distribution of the data, then the Fisher and Hessian matrices are in fact equivalent [24]. Roughly, this means that, if the model has high accuracy, we can approximate the Hessian of θ with the Fisher matrix.

It is sometimes useful to consider an approximation to the Fisher, where the distribution $P_{\mathbf{x},\mathbf{y}}$ is replaced with the empirical training distribution $\widehat{Q}_{\mathbf{x},\mathbf{y}}$, leading to the *empirical Fisher matrix* $\widehat{\mathbf{F}}$:

$$\widehat{\mathbf{F}} = \mathrm{E}_{\widehat{Q}_\mathbf{x}}\left[\mathrm{E}_{\widehat{Q}_{\mathbf{y}|\mathbf{x}}}\left[\nabla \log p_\theta(\mathbf{y}|\mathbf{x}) \nabla \log p_\theta(\mathbf{y}|\mathbf{x})^\top\right]\right] = \frac{1}{N}\sum_{i=1}^N \underbrace{\nabla \ell\left(\mathbf{y}_i, f(\mathbf{x}_i; \theta)\right)}_{\nabla \ell_i} \nabla \ell\left(\mathbf{y}_i, f(\mathbf{x}_i; \theta)\right)^\top.$$

The Woodbury-Sherman-Morrison Trick. The fact that this approximation is a sum of rank-one matrices has the benefit that it allows efficient exact computation of the Fisher inverse. Specifically, one can apply the classic Woodbury-Sherman-Morrison formula to compute the inverse recursively as

$$\widehat{\mathbf{F}}^{-1} = \widehat{F}_N^{-1} = \widehat{F}_{N-1}^{-1} - \frac{\widehat{F}_{N-1}^{-1} \nabla \ell_N (\nabla \ell_N^\top \widehat{F}_{N-1}^{-1})}{N + \nabla \ell_N^\top \widehat{F}_{N-1}^{-1} \nabla \ell_N}, \tag{2}$$

where the recursion is over samples, and the base step is $\widehat{F}_0^{-1} = \lambda^{-1} I_d$, with λ being a small positive constant. This approach was independently introduced by Hassibi and Stork [11] for pruning, and Amari [3] for natural gradient. Singh and Alistarh [33] showed that it can be scaled to DNNs by block-wise approximation, and that it provides better approximations of the loss than K-FAC-based [37], or diagonal [35] approximations [6], leading to more accurate pruning. The main shortcoming of directly applying (2) is the prohibitive computational cost of $\Omega(d^2 m)$, even when reduced to $\Omega(dBm)$ by B-block-wise approximation. Our method proposes a *matrix-free* approach for exactly calculating IHVPs and querying individual elements of the inverse Hessian, of cost $O(dm)$ after $O(dm^2)$ precomputation. Our method is numerically equivalent to the direct computation above.

Diagonal Approximations. A common approximation, both for optimization, e.g. [17, 15] but also for pruning [35], is to assume that the Fisher matrix is *diagonal*. In our notation, this method has setup cost $O(dm)$, and diagonal query cost $O(d)$. However, as evident from the experimental results of [33] this provides lower approximation quality relative to both block-wise methods or K-FAC [37].

K-FAC Approximations. This approach observes that the entries of the true Fisher corresponding to blocks "between" two layers, which can be written as the expectation of a Kronecker product between two matrices, can be approximated as the Kronecker product of the expectations of those two matrices (reversing the order between the product and the expectation). This approximation has been leveraged for both pruning and for optimization, and it allows efficient computation of the inverse [5, 30, 41, 37, 21]; however, it is known that this approximation does not always hold [28]. Another relative drawback is that the Kronecker factorization only occurs naturally for fully-connected layers; there is work on extensions to other layer types [9, 27], via additional approximations. We compare against K-FAC-based optimizers and pruners, and find that our method yields better results.

Additional Approaches. Our approach is similar in spirit to matrix-free methods [29, 23, 25], but the algorithms we present are new. *Hessian-free* optimization [25] also forgoes the explicit computation of Hessians in favor of computing an IHVP with a vector $\mathbf{v}$. However, this estimation is performed by iteratively approximating the solution to the linear system $\mathbf{H}\mathbf{x} = \mathbf{v}$ for some given $\mathbf{x}$

without ever explicitly forming $\mathbf{H}$. One disadvantage of this method in practice is that it requires tuning and several very costly iterations to converge (for a single $\mathbf{v}$), as the underlying Hessian can be ill-conditioned. The L-OBS pruning method [6] approximates second-order information by defining independent layer-wise objectives, which allows the direct approximation of the layer-wise Hessians via a carefully-crafted block structure. In contrast, our approach allows for fully-global Hessian estimation, and yields better pruning results at scale.

Full-matrix adaptive regularization has similar goals, but in the context of adaptive optimizers [7]. Agarwal et al. [2] proposed GGT, which allows the efficient computation of the *inverse square root* of the low-rank matrix resulting from the sum of gradient outer products over a sliding window. At its core, this procedure requires an eigen-decomposition (the authors take an SVD) of an $m \times m$ matrix at every time step, which is reasonably efficient for small values of m.

Our dynamic algorithm solves a similar, but slightly simpler problem, as we only want to invert the matrix (rather than also computing its square root). However, we do not perform any eigen-decompositions; instead, we carefully maintain intermediate information that allows an efficient explicit computation of the scalar coefficients in Equation 6. At the end of Section 3.2, we discuss how this approach is considerably more efficient in practice and allows executing at larger window sizes (with small overhead), which leads to improved model accuracy as we show in our experiments. Our dynamic algorithm has per-step cost $O(dm + m^3)$, relative to $O(dm^2 + m^3)$ for GGT [2]; this can result in lower overhead versus GGT, even when the SVD cost is negligible (e.g. if m is small).

Yao et al. [39] recently provided an alternative method for approximating the diagonal of the inverse Hessian, using Hutchinson's randomized algorithm for estimating the diagonal. To mitigate the variance, the authors introduce non-trivial smoothing heuristics. Their approach has theoretical per-iteration cost of at least $2\times$ versus SGD, as it requires a second backward pass over the network; this cost is usually higher in practice. Our algorithm often matches their accuracy, and (unlike AdaHessian) its cost depends exclusively on the model size irrespective of the underlying structure; thus, as we show in the experimental section, our average *practical overhead* is less than 50% for dense models, and less than 10% for sparse ones.

Approximation Quality. Künstner et al. [19] performed an in-depth analysis of the empirical Fisher, making the point that, in theory, the approximation could become meaningless if all sample gradients are zero. Similarly to [33], we did not find that this occurs in practice for deep neural networks, as sample gradients are never zero. The latter reference provides detailed comparisons of diagonal, K-FAC, and other approximations, and finds that the empirical Fisher can provide competitive approximation quality for DNNs. Further, they demonstrate that better loss approximation implies better accuracy for neural network pruning. We therefore do not repeat their loss analysis, and mainly compare methods in terms of application performance.

3 Algorithm Description

3.1 The Static Algorithm

As a warm-up, we first describe the IHVP algorithm given a static set of gradients. While our notation is customized for the empirical Fisher, all our techniques are applicable to any matrix that can be written as the sum of rank-one components. Specifically, we are given m vectors, which we assume to be the gradient vectors $(\nabla \ell_i)_{i=1}^m$, and must compute quantities related to the inverse of the matrix resulting from the sum (more precisely, the average) of their outer products, which we denote by $\widehat{F}_m^{-1}$. The main idea is to rewrite the recursive description of $\widehat{F}_m^{-1}$ such that, after some precomputation, the matrix-vector-product $\widehat{F}_m^{-1}\mathbf{x}$ can be computed efficiently for any vector $\mathbf{x}$. Using the Sherman-Morrison formula applied to the partial empirical Fisher matrix $\widehat{F}_i$ corresponding to the first i gradients (and scaled by $1/m$), we obtain the following recursion:

$$\widehat{F}_i^{-1}\mathbf{x} = \widehat{F}_{i-1}^{-1}\mathbf{x} - \frac{\widehat{F}_{i-1}^{-1}\nabla \ell_i (\widehat{F}_{i-1}^{-1}\nabla \ell_i)^\top}{m + \nabla \ell_i^\top \widehat{F}_{i-1}^{-1}\nabla \ell_i}\mathbf{x}, \text{ with } \widehat{F}_0^{-1}\mathbf{x} = \lambda^{-1} I_d \mathbf{x} = \lambda^{-1}\mathbf{x} \text{ and } \lambda > 0. \quad (3)$$

For simplicity, let us set $\mathbf{v_i} = \widehat{F}_{i-1}^{-1}\nabla \ell_i$ and unroll the Sherman-Morrison recursion, which gives:

$$\widehat{F}_i^{-1}\mathbf{x} = \widehat{F}_{i-1}^{-1}\mathbf{x} - \mathbf{v_i}\frac{\mathbf{v_i}^\top \mathbf{x}}{m + \nabla \ell_i^\top \mathbf{v_i}} = \lambda^{-1}\mathbf{x} - \sum_{j=1}^{i} \mathbf{v_j}\frac{\mathbf{v_j}^\top \mathbf{x}}{m + \nabla \ell_j^\top \mathbf{v_j}}. \quad (4)$$

IHVP Computation. Assuming that we have already computed all the vectors $\mathbf{v_j}$, the above expression can be calculated in $O(dm)$ time without any intermediate $d \times d$ matrices: first compute the scalar fractions in (4), and then evaluate the resulting linear combination of terms in $\mathbf{x}$ and $\mathbf{v_j}$. Since $\mathbf{v_i} = \widehat{F}_{i-1}^{-1}\nabla\ell_i$, it can be computed in exactly the same way given all $\mathbf{v_j}$ for $j < i$. Thus, all m vectors $\mathbf{v_i}$ of dimension d can be precomputed in increasing order of i using $O(dm^2)$ total time. As an additional optimization (to cut required memory in half), we also precompute all $q_j = m + \nabla\ell_j^\top \mathbf{v_j}$.

Querying the Inverse. To extract individual elements $[\widehat{F}_m^{-1}]_{ij}$ of the inverse Hessian approximation, we can write $\mathbf{e_i}^\top \widehat{F}_m^{-1}\mathbf{e_j}$ in the form of (4) where $\mathbf{e_i}$ and $\mathbf{e_j}$ are indicator vectors

$$\mathbf{e_i}^\top \widehat{F}_m^{-1}\mathbf{e_j}^\top = \mathbf{e_i}^\top \lambda^{-1}\mathbf{e_j} - \sum_{k=1}^{m} \mathbf{e_i}^\top \mathbf{v_k} \frac{\mathbf{v_k}^\top \mathbf{e_j}}{q_k}. \tag{5}$$

As $\mathbf{e_i}^\top \mathbf{v_k}$ and $\mathbf{v_k}^\top \mathbf{e_j}$ can both be realized as constant time indexing operations, the above turns into a sum over m scalars. Hence, our method admits $O(m)$ access to any element of $\widehat{F}_m^{-1}$ using the same precomputed $\mathbf{v_k}$ and q_k as for the efficient calculation of $\widehat{F}_m^{-1}\mathbf{x}$.

Additional Optimizations. The algorithm admits a fast vectorized implementation, and several optimizations, which we describe in the Appendix. For example, we perform several memory-saving optimizations, as well as explicit page swapping between CPU and GPU memory to mitigate memory transfer costs. Furthermore, the static algorithm can be applied independently to each block of a block-wise approximation: for block size B, the runtime is reduced to d/B. Then, as long as $m < B$, our method is B/m times faster than the direct implementation of the Woodbury inverse.

3.2 The Dynamic Algorithm

We now describe the *dynamic* algorithm, which assumes that gradients arrive in an online fashion and must be integrated into the Fisher estimate. We first present the algorithm itself, i.e. the setup/update/IHVP computations, and perform a complexity analysis. Next, we show how the algorithm can be derived and finally we conclude with notes on an efficient practical implementation.

High-Level View. The main idea of the dynamic algorithm is to write the IHVP as

$$\widehat{F}_m^{-1}\mathbf{x} = \lambda^{-1}\mathbf{x} - \sum_{j=1}^{m} c_j^m \nabla\ell_j, \tag{6}$$

where the scalar coefficients c_j^m can be computed efficiently from just the scalar products $\nabla\ell_i^\top\nabla\ell_j$ and $\nabla\ell_i^\top\mathbf{x}$. Then, any gradient $\nabla\ell_i$ can be replaced by updating just $O(m)$ of the stored scalar product values. In the following, we use $\mathbf{G} = [\nabla\ell_1^\top; \nabla\ell_2^\top; \ldots; \nabla\ell_m^\top]$ to denote the row-wise $m \times d$ matrix of the m gradients for which we wish to compute the inverse empirical Fisher. Further, for each i from 1 to m, let $\mathbf{b^i}$ be a vector with m components $(b_1^i, b_2^i, \ldots, b_m^i)$, such that

$$\widehat{F}_{i-1}^{-1}\nabla\ell_i = \lambda^{-1}\nabla\ell_i - \sum_{j=1}^{i-1} a_j^{i-1}\nabla\ell_j = \sum_{j=1}^{m} b_j^i \nabla\ell_j, \tag{7}$$

i.e. containing the scalar coefficients of $\widehat{F}_{i-1}^{-1}\nabla\ell_i$. This means, $b_j^i = -a_j^{i-1}$ when $j < i$, $b_i^i = \lambda^{-1}$ and $b_j^i = 0$ otherwise.

Initial Precomputation & Update. The dynamic algorithm maintains three $m \times m$ matrices: $\mathbf{G}\mathbf{G}^\top$, $\mathbf{D}$ and $\mathbf{B}$. The first is the symmetric gradient scalar product matrix $[\mathbf{G}\mathbf{G}^\top]_{ij} = \nabla\ell_i^\top\nabla\ell_j$. The second is an upper triangular matrix and stores the precomputed values $[\mathbf{D}]_{ij} = \nabla\ell_i\widehat{F}_{i-1}^{-1}\nabla\ell_j$ for $i \leq j$. The third is the row-wise matrix of the coefficient vectors $[\mathbf{B}]_{ij} = b_j^i$, which makes it lower triangular with a diagonal of λ^{-1}. We now discuss how to compute those matrices.

The initial setup of the dynamic algorithm begins by evaluating $\mathbf{G}\mathbf{G}^\top$ in a straightforward fashion. Next, $\mathbf{D}$ can be computed for $i \leq j$ according to the following recursion:

$$[\mathbf{D}]_{ij} = (\ell_i^\top \widehat{F}_{i-1}^{-1}\nabla\ell_j) \text{ and } (\nabla\ell_j^\top \widehat{F}_0^{-1}\nabla\ell_k) = \lambda^{-1}[\mathbf{G}\mathbf{G}^\top]_{jk} \tag{8}$$

$$(\nabla\ell_j^\top \widehat{F}_i^{-1}\nabla\ell_k) = (\nabla\ell_j^\top \widehat{F}_{i-1}^{-1}\nabla\ell_k) - \frac{(\nabla\ell_j^\top \widehat{F}_{i-1}^{-1}\nabla\ell_i)(\nabla\ell_i^\top \widehat{F}_{i-1}^{-1}\nabla\ell_k)}{m + (\nabla\ell_i^\top \widehat{F}_{i-1}^{-1}\nabla\ell_i)}. \tag{9}$$

Given $\mathbf{D}$, we can then conclude the precomputation by calculating $\mathbf{B}$ for $i \geq j$ recursively as:

$$[\mathbf{B}]_{ii} = \lambda^{-1} \;\text{ and }\; [\mathbf{B}]_{ij} = -\sum_{k=j}^{i-1} \frac{[\mathbf{D}]_{ik}}{m + [\mathbf{D}]_{ii}} [\mathbf{B}]_{kj}. \tag{10}$$

After the initial setup, gradient $\nabla \ell_k$ can be replaced with the gradient $\nabla \ell'_k$ by first updating the row k of $\mathbf{G}$ and then replacing row and column k in $\mathbf{G}\mathbf{G}^\top$ with $\mathbf{G}\nabla \ell'_k$. Afterwards, the recomputation of columns $j \geq k$ of $\mathbf{D}$ and rows $i \geq k$ of $\mathbf{B}$ completes the update.

Multiplication. Once $\mathbf{G}\mathbf{G}^\top$, $\mathbf{D}$ and $\mathbf{B}$ have been precomputed, one can perform efficient IHVPs of the form $\widehat{F}_m^{-1}\mathbf{x}$ with arbitrary vectors $\mathbf{x}$. This is done by first evaluating $\mathbf{p} = \mathbf{G}\mathbf{x}$ and then computing all m values q_i by the following recursion:

$$q_i = \frac{(\ell_i^\top \widehat{F}_{i-1}^{-1}\mathbf{x})}{m + [\mathbf{D}]_{ii}} \text{ and } (\nabla \ell_j^\top \widehat{F}_0^{-1}\mathbf{x}) = \lambda^{-1} p_j. \tag{11}$$

$$(\nabla \ell_j^\top \widehat{F}_i^{-1}\mathbf{x}) = (\nabla \ell_j^\top \widehat{F}_{i-1}^{-1}\mathbf{x}) - \frac{[\mathbf{D}]_{ij}}{m + [\mathbf{D}]_{ii}} (\nabla \ell_i^\top \widehat{F}_{i-1}^{-1}\mathbf{x}). \tag{12}$$

Eventually, the final result of the IHVP is obtained by:

$$\widehat{F}_m^{-1}\mathbf{x} = \lambda^{-1}\mathbf{x} - \sum_{j=1}^{m} (\sum_{k=j}^{m} q_k [\mathbf{B}]_{kj}) \nabla \ell_j. \tag{13}$$

Complexity Analysis. The dynamic algorithm stores m gradients of dimension d as well as three $m \times m$ matrices (and a few m vectors) and thus has an overall memory complexity of $O(dm + m^2)$.

Next, we analyze the time complexity of all important operations. Initially, $\mathbf{G}\mathbf{G}^\top$ must be computed once, which takes $O(dm^2)$ time. The recursion of $\mathbf{D}$ has three indices with $1 \leq i, j, k \leq m$ and each step takes constant time. Thus, it can be computed in $O(m^3)$ time with dynamic programming. Further, since values for index i depend only on values for index $i - 1$, the dynamic programming can be implemented in $O(m^2)$ space. $\mathbf{B}$ has two indices $1 \leq i, j \leq m$ and every recursion takes $O(m)$ time, meaning that it can also be fully computed in $O(m^3)$ through dynamic programming. Hence, the overall initial setup cost is $O(dm^2 + m^3)$.

To replace one gradient with $\nabla \ell'$, we have to compute $\mathbf{G}\nabla \ell'$ as well as (partially) recalculate $\mathbf{D}$ and $\mathbf{B}$, which takes at worst $O(dm + m^3)$ time. An IHVP requires two matrix-vector products involving $\mathbf{G}$ and a recursion with two indices and therefore has a complexity of $O(dm + m^2)$.

Algorithmic Derivation. The dynamic algorithm can be derived directly from Theorem 1, which we state here in a simplified form (using the definitions of $\mathbf{B}$ and $\mathbf{D}$), and prove in the Appendix.

Theorem 1. *Let $\widehat{F}_i^{-1} = 1/m \cdot \sum_{j=1}^{i} \nabla \ell_j \nabla \ell_j^\top$, then $\widehat{F}_i^{-1}\mathbf{x}$ can be calculated for arbitrary $\mathbf{x}$ as:*

$$\widehat{F}_i^{-1}\mathbf{x} = \lambda^{-1}\mathbf{x} - \sum_{j=1}^{i} c_j^i \nabla \ell_j \;\text{ and }\; c_j^i = \sum_{k=j}^{i} \frac{(\nabla \ell_k^\top \widehat{F}_{k-1}^{-1}\mathbf{x})}{m + [\mathbf{D}]_{kk}} [\mathbf{B}]_{kj}. \tag{14}$$

Equation (14) with $i = m$ is exactly equal to the IHVP computation (13) as the fraction in the innermost sum corresponds to q_k. Similarly, index shifting $i = i - 1$ and setting $x = \nabla \ell_i$, thus turning c_j^i into $-[\mathbf{B}]_{ij}$ as well as the fraction's numerator to $[\mathbf{D}]_{ik}$, recovers the precomputation formula of $\mathbf{B}$ given by (10) for $i > j$. The formulas for $\mathbf{D}$ and q_i follow directly from an expansion of the Woodbury formula followed by an appropriate recursive evaluation that avoids any matrix/vector operations except in the base case (we indicate the corresponding recursive calls by brackets).

Efficient Implementation. Directly implementing the discussed recursive formulas in a modern machine learning framework would likely result in a very slow code. Fortunately, it is possible to implement all computations required for the dynamic algorithm very efficiently on a GPU. We describe how to do this and provide a full implementation in the Supplementary Material. Specifically, we provide complete sample code that is able to perform the calculation of $\mathbf{D}$ and $\mathbf{B}$, i.e. the $O(m^3)$ component of the overall update cost, in < 10 milliseconds (on the NVIDIA RTX 2080 Ti) for values of m as high as 1024 (and the code can still be further optimized). For reference, this is $> 10\times$ faster than the highly optimized $m \times m$ SVD computation done at every step by GGT [2]. We emphasize that this $O(m^3)$ computation being very fast in practice is crucial to reach low overheads, especially when dealing with a sparse model where the $O(dm)$ matrix-vector products are no longer the bottleneck.

4 Experimental Validation

4.1 Application 1: Pruning DNNs using the Static Algorithm

Background. Given a trained model $\theta^\star$, the goal of pruning is to find the weight θ_i, or the set of weights, whose setting to 0 would lead to a minimal increase in the training loss. Under a local quadratic approximation of the loss function, the OBD framework [22] shows that the "optimal" weight to be removed is the one with the lowest value of the saliency metric $\rho(\theta_i) = \frac{\theta_i^2}{2\,[\widehat{\mathbf{F}}^{-1}]_{ii}}$ and proposed to estimate $[\widehat{\mathbf{F}}^{-1}]_{ii}$ by diagonal approximation. The OBS framework [11] observed that the *remaining* weights θ should also be updated via the optimal perturbation $\delta\theta = -\theta_i \widehat{\mathbf{F}}^{-1}\mathbf{e}_i / [\widehat{\mathbf{F}}^{-1}]_{ii}$, where $\mathbf{e}_i$ is the ith basis vector. Our algorithm supports these operations in linear-in-d time.

Wang et al. [37] raised the valid point that applying the OBS update above to multiple weights being removed at once may be incorrect, since it ignores possible correlations between those weights. We considered this point in detail, comparing results between OBD pruning [22], the OBS update [11], and an augmented version of OBS which disentangles the correlations by solving a corresponding linear system [33]. The results, presented in the Appendix, suggest that the OBD update is quite beneficial even in its approximate form, and that the effect of correlation is small for unstructured pruning. We use the augmented version of OBS, solving for correlations when feasible.

Experimental Setup. We prune CNNs (ResNet-50 [13] and MobileNet-V1 [14]) on the ImageNet dataset [32]. The models are standard in the pruning literature [8], and therefore several strong baselines exist. Timing experiments are run on a machine with NVIDIA RTX 2080 Ti GPUs, a 48-core Intel CPU, and 512 GB of RAM. Following [33], we used batched gradients (of size 16) as single samples inside the Fisher approximation. This does not alter results, but reduces variance.

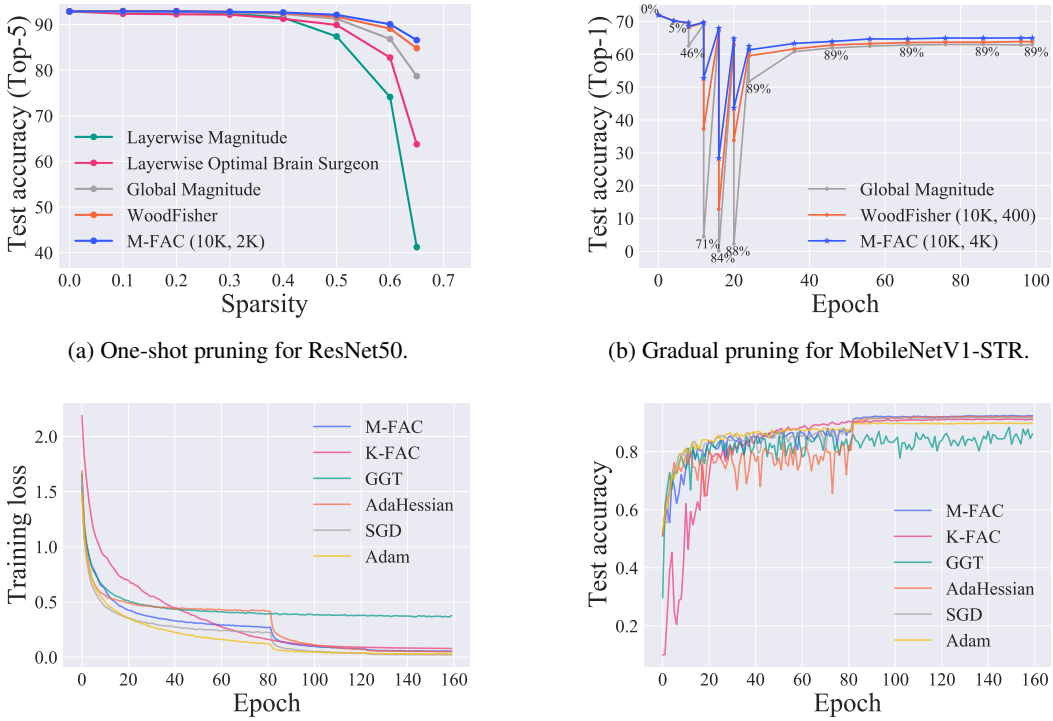
We compare against Global Magnitude Pruning (GMP) [10], Layer-wise Optimal Brain Surgeon (L-OBS) [6], Soft Threshold Reparametrization (STR) [20], and WoodFisher [33]. The latter two methods are state-of-the-art for gradual pruning. WoodFisher is a special case of our method, using a block approximation. It is numerically equivalent at the same parameter settings, but has significantly higher computational and storage cost. Relative to it, we therefore focus on executing at higher parameter values. We compare against their public implementation. The Appendix contains full hyper-parameters, additional experiments and ablations with respect to block size and number of gradients, and a comparison with K-FAC pruning in a simpler setting [41].

Gradual pruning is performed in incremental steps, according to a polynomial schedule [42], followed by fine-tuning, where the learning rate is gradually decreased. We follow the hyper-parameters of [33]: pruning is performed in steps at 4 epochs apart for the first 24 epochs, followed by fine-tuning. Following [33], we used batched gradients (of size 16) as single samples inside the Fisher approximation. This does not change the nature of the results, but reduces variance.

Pruning Results. Figure 1a shows the Top-5 accuracy of one-shot ResNet50 pruned models for L-OBS, GMP, WoodFisher, and M-FAC, where our method is executed with a parameter setting that is infeasible for WoodFisher due to the required computational and storage costs. (We use Top-5 accuracy for this experiment following [6].) We note the improved accuracy of global methods relative to L-OBS and layer-wise magnitude pruning, and the fact that the Fisher approximation yields consistently better results relative to the global magnitude. This suggests that a better approximation of the Fisher inverse also yields better pruning results, and is in line with the loss approximation study of [33]. Generally, we found that estimating using more gradients always improves results, until saturation. However, for some models, smaller block sizes sometimes yield better results for a fixed number of gradients, presumably due to gradient scaling effects.

Figure 1b examines the effect of the improvement in one-shot pruning accuracy at each step of pruning, on the final result, when pruning MobileNetV1-STR gradually to 89% sparsity. WoodFisher uses block size $10K$, and $m = 400$ gradients, while M-FAC uses the same block size but $m = 4000$ gradients. Executing WoodFisher with the same parameters would be extremely slow. Note the gap in one-shot accuracy following each pruning step: this translates in the final accuracy gap of more than 1% even after fine-tuning.

Table 1 presents gradual pruning results for MobileNetV1-STR/ImageNet at 89% sparsity and ResNet50/ImageNet at 95% sparsity, relative to other state-of-the-art methods. M-FAC outperforms previous methods in the case of MobileNetV1-STR by more than 1.5% Top-1 accuracy. Further, despite using $10\times$ more gradients to estimate the empirical Fisher, the per-step pruning cost of



(a) One-shot pruning for ResNet50.

(b) Gradual pruning for MobileNetV1-STR.

(c) Training loss on ResNet20/CIFAR-10.

(d) Top-1 test accuracy on ResNet20/CIFAR-10.

Figure 1: **(a)** Accuracy vs. sparsity for one-shot pruning on ResNet50/ImageNet. **(b)** Accuracy during gradual pruning for the MobileNetV1-STR/ImageNet experiment. **(c)**, **(d)** Comparison for different optimizers on the ResNet20/CIFAR-10 example.

M-FAC is $> 6\times$ lower than WoodFisher's as can be seen in Table 2 (for the same parameters settings M-FAC is $> 100\times$ faster). The ResNet50 results show similar gains, but by smaller margins.

Table 1: Gradual pruning results (Top-1 accuracy) for **MobileNetV1-STR/ImageNet** at 89% sparsity and **ResNet50/ImageNet** at 95% sparsity.

Method	MBv1 Top-1 (%)	RN50 Top-1 (%)
Dense baseline	72.00	77.01
STR [20]	62.10	70.40
Global Magnitude	62.94	71.72
WoodFisher [33]	63.87	72.12
M-FAC	**65.01**	**72.32**

Table 2: Minutes per pruning step compared with WoodFisher (WF). OOM denotes out of memory on a system with 512GB RAM.

Model	B	m	WF	**M-FAC**
MBv1-STR	10k	400	60m	**0.5m**
MBv1-STR	10k	4k	OOM	**8.7m**
MBv1-STR	all	1k	OOM	**1.2m**
ResNet50	2k	400	35m	**2.5m**
ResNet50	10k	1k	OOM	**14.0m**
ResNet50	all	512	OOM	**4.7m**

4.2 Application 2: Optimization using the Dynamic Algorithm

Matrix-Free Preconditioned SGD. The dynamic algorithm can provide an efficient implementation for the following variant of SGD. Let t be the current iteration index, and $\Delta < t$ be the length of the sliding window during which the estimation of the inverse Hessian is performed. Consider the iteration $\theta_{t+1} = \theta_t - \eta_t \widehat{F}^{-1}_{[t-\Delta,t]} \nabla \ell_t$, where η_t is the learning rate. Let $\nabla \ell_{t-\tau+1}, \ldots, \nabla \ell_t$ be the gradients obtained at steps $(t-\tau+1), \ldots, t$, respectively. Then, we define the preconditioning matrix as $\widehat{F}_{[t-\Delta,t]} = 1/\Delta \cdot \sum_{\tau=0}^{\Delta-1} \nabla \ell_{t-\tau} \cdot \nabla \ell_{t-\tau}^\top$, that is, an empirical Fisher matrix generated with respect to the sliding window of Δ gradients. At each step, our dynamic algorithm first removes $\nabla \ell_{t-\Delta+1}$ and adds $\nabla \ell_t$ to the inverse estimate, and then computes $\widehat{F}^{-1}_{[t-\Delta,t]} \nabla \ell_t$.

Setup and Objectives. We now experiment with the above optimizer, implemented using the dynamic algorithm, without momentum. We compare against standard optimizers such as SGD with momentum and Adam [15], but also against approximate second-order methods such as K-FAC [27], GGT [2], and AdaHessian [39]. For the former two, we use the implementations contributed to

Method	ResNet20		ResNet32	
	Acc.	Overhead*	Acc.	Overhead*
SGD	91.78	1.00×	92.80	1.00×
Adam	89.67	1.05×	90.56	1.10×
KFAC	91.65	1.21× (2.35×)	90.09	1.50× (3.05×)
GGT	88.38	1.05× (8.35×)	89.14	1.05× (8.88×)
AdaHessian	92.17	3.50×	**92.81**	3.30×
M-FAC	**92.34**	1.55×	92.65	1.50×

Model	SGD	Adam	M-FAC (Overh.)
WRN 22-2	**69.93**	66.90	69.76 (1.40×)
WRN 40-2	71.75	70.14	**72.42** (1.35×)
WRN 22-4	73.13	72.52	**74.06** (1.43×)
MBv1 Dense	68.06	67.92	**68.96** (1.25×)
MBv1 Sparse	64.11	–	**64.78** (1.03×)
RN50 Sparse	74.78	–	**75.10** (1.05×)

Table 4: **Left**: Comparison of second-order optimizers with individual tuning and weight decay. **Right**: Additional experiments with no tuning and weight decay. Sparse results are from finetuning after the last step of gradual pruning to $\approx 90\%$ sparsity. * For KFAC/GGT we indicate in parentheses the overhead in a comparable setting to M-FAC (i.e. same batch size/number of gradients).

TensorFlow [1] by the authors, while for the latter we use the authors' PyTorch implementation. For fairness, we always follow the settings recommended by the authors (even though this makes overheads harder to compare); for the best accuracy on the models we considered, we grid-searched learning rates and weight-decay for each method in turn.

We examine the *loss and test accuracy* for each method, on the standard ResNet20 and ResNet32 models [13] on CIFAR-10 [18] (see also Figures 1d and 1c). Since implementations have slightly different framework requirements, our main performance metric is *overhead over SGD*, measured in each algorithm's environment, on an NVIDIA RTX 3090 GPU for PyTorch and Titan RTX for TensorFlow. For all second-order methods, we compute the preconditioner at each step—reducing the frequency reduces overheads proportionally for each method, but introduces an additional hyper-parameter. We provide code, as well as additional information, in the Supplementary.

Our first results are presented in Table 4 (left), where we examine the best Top-1 test accuracy obtained by each method, as well as its overhead relative to SGD. For K-FAC, the best accuracy is obtained with batch size 1000, while all other methods operate on batch size 128. Similarly, GGT's best results are achieved with window size $m = 100$, while for M-FAC larger window sizes improve results; we use $m = 512$. Due to these discrepancies, we measure overheads with "best accuracy" settings, for each method but also in the same setup as M-FAC; the latter is given in brackets.

In terms of accuracy, M-FAC scores highest in the ResNet20 experiment, and comes third, to tuned SGD and AdaHessian (performing almost the same), on ResNet32. The overhead of M-FAC is around 50%, which is much lower than AdaHessian, on par with K-FAC, but higher than GGT. However, these overheads are comparable only when GGT uses 5x less gradients (counter-intuitively, larger window sizes sometimes make results worse for GGT). On the same settings, M-FAC provides up to 5x lower overhead. We examine this gap in detail in the Appendix, and determine that the difference is due to 1) the better performance of our dynamic algorithm relative to the SVD computation used by GGT; 2) the difference in complexity due to the $\Theta(dm^2)$ term in GGT, versus our $O(dm)$ term. Similarly, K-FAC's overhead for the same batch size as M-FAC is substantially higher.

Table 4 (right) provides a more realistic scenario, where we run SGD, Adam, and M-FAC *without tuning or weight-decay* on Wide Residual Networks (WRN) [40] for CIFAR-100, as well as on MobileNetV1/ImageNet, and examine test accuracy. M-FAC achieves the highest accuracy on most runs, even on ImageNet. However, we emphasize that these results can probably be improved significantly by parameter tuning. Another interesting application of M-FAC is fine-tuning sparse models. We fine-tune 90% sparse MobileNet/ResNet50 models for 30 epochs starting from the last gradual pruning step and find that M-FAC reaches higher final accuracy than SGD in both cases. At the same time, it is only negligibly slower ($\leq 5\%$ overhead) since our algorithm's complexity is linear in the model *density*.

5 Discussion

We presented static and dynamic linear-in-d algorithms for computing IHVPs when the Hessian matrix can be approximated by a sum of rank-1 matrices. We used the classic empirical Fisher approximation, but our results can apply more broadly. The main limitation is the cost of storing the m additional gradients. For the static algorithm, we can leverage system memory via an efficient paging mechanism, described in the Appendix. The dynamic algorithm could parallelize gradient storage, or perform gradient compression [34]. We plan to investigate this in future work, as well as a larger-scale study of optimization performance.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Naman Agarwal, Brian Bullins, Xinyi Chen, Elad Hazan, Karan Singh, Cyril Zhang, and Yi Zhang. Efficient full-matrix adaptive regularization. In *International Conference on Machine Learning*, pages 102–110. PMLR, 2019.

[3] Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998.

[4] Shun-ichi Amari. *Information geometry and its applications*, volume 194. Springer, 2016.

[5] Jimmy Ba, Roger Grosse, and James Martens. Distributed second-order optimization using kronecker-factored approximations. 2016.

[6] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon, 2017.

[7] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2010.

[8] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks, 2019.

[9] Roger Grosse and James Martens. A kronecker-factored approximate fisher matrix for convolution layers, 2016.

[10] Masafumi Hagiwara. A simple and effective method for removal of hidden units and weights. *Neurocomputing*, 6(2):207 – 218, 1994. Backpropagation, Part IV.

[11] Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*, page 164–171, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

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

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[14] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[16] Diederik P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, pages 2575–2583. Curran Associates, Inc., 2015.

[17] Shankar Krishnan, Ying Xiao, and Rif A Saurous. Neumann optimizer: A practical optimization algorithm for deep neural networks. *arXiv preprint arXiv:1712.03298*, 2017.

[18] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[19] Frederik Kunstner, Philipp Hennig, and Lukas Balles. Limitations of the empirical fisher approximation for natural gradient descent. In *Advances in Neural Information Processing Systems*, pages 4156–4167, 2019.

[20] Aditya Kusupati, Vivek Ramanujan, Raghav Somani, Mitchell Wortsman, Prateek Jain, Sham Kakade, and Ali Farhadi. Soft threshold weight reparameterization for learnable sparsity, 2020.

[21] César Laurent, Thomas George, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. An evaluation of fisher approximations beyond kronecker factorization, 2018.

[22] Yann Le Cun, John S. Denker, and Sara A. Solla. *Optimal Brain Damage*, page 598–605. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[23] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

[24] Alexander Ly, Maarten Marsman, Josine Verhagen, Raoul Grasman, and Eric-Jan Wagenmakers. A tutorial on fisher information, 2017.

[25] James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 735–742, Madison, WI, USA, 2010. Omnipress.

[26] James Martens. New insights and perspectives on the natural gradient method. *arXiv preprint arXiv:1412.1193*, 2014.

[27] James Martens, Jimmy Ba, and Matt Johnson. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*, 2018.

[28] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature, 2015.

[29] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[30] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. 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.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[32] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[33] Sidak Pal Singh and Dan Alistarh. Woodfisher: Efficient second-order approximation for neural network compression, 2020.

[34] Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. meProp: Sparsified back propagation for accelerated deep learning with reduced overfitting. In *Proceedings of the Thirty-Fourth International Conference on Machine Learning*, 2017.

[35] Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. Faster gaze prediction with dense networks and fisher pruning, 2018.

[36] Valentin Thomas, Fabian Pedregosa, Bart Merriënboer, Pierre-Antoine Manzagol, Yoshua Bengio, and Nicolas Le Roux. On the interplay between noise and curvature and its effect on optimization and generalization. In *International Conference on Artificial Intelligence and Statistics*, pages 3503–3513. PMLR, 2020.

[37] Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. Eigendamage: Structured pruning in the kronecker-factored eigenbasis, 2019.

[38] Max A Woodbury. *Inverting modified matrices*. Statistical Research Group, 1950.

[39] Zhewei Yao, Amir Gholami, Sheng Shen, Kurt Keutzer, and Michael W Mahoney. Adahessian: An adaptive second order optimizer for machine learning. *arXiv preprint arXiv:2006.00719*, 2020.

[40] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[41] Wenyuan Zeng and Raquel Urtasun. MLPrune: Multi-layer pruning for automated neural network compression, 2019.

[42] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [No] Assets are open-source with standard permissive licenses.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]