SIAM J. MATRIX ANAL. APPL.
Vol. 42, No. 1, pp. 165–184

FAST APPROXIMATION OF THE GAUSS–NEWTON HESSIAN MATRIX FOR THE MULTILAYER PERCEPTRON[*]

CHAO CHEN[†], SEVERIN REIZ[‡], CHENHAN D. YU[§], HANS-JOACHIM BUNGARTZ[‡], AND GEORGE BIROS[†]

Abstract. We introduce a fast algorithm for entrywise evaluation of the Gauss–Newton Hessian (GNH) matrix for the fully connected feed-forward neural network. The algorithm has a precomputation step and a sampling step. While it generally requires $\mathcal{O}(Nn)$ work to compute an entry (and the entire column) in the GNH matrix for a neural network with N parameters and n data points, our fast sampling algorithm reduces the cost to $\mathcal{O}(n+d/\epsilon^2)$ work, where d is the output dimension of the network and ϵ is a prescribed accuracy (independent of N). One application of our algorithm is constructing the hierarchical-matrix ($\mathcal{H}$-matrix) approximation of the GNH matrix for solving linear systems and eigenvalue problems. It generally requires $\mathcal{O}(N^2)$ memory and $\mathcal{O}(N^3)$ work to store and factorize the GNH matrix, respectively. The $\mathcal{H}$-matrix approximation requires only $\mathcal{O}(Nr_o)$ memory footprint and $\mathcal{O}(Nr_o^2)$ work to be factorized, where $r_o \ll N$ is the maximum rank of off-diagonal blocks in the GNH matrix. We demonstrate the performance of our fast algorithm and the $\mathcal{H}$-matrix approximation on classification and autoencoder neural networks.

Key words. Gauss–Newton Hessian, fast Monte Carlo sampling, hierarchical matrix, second-order optimization, multilayer perceptron

AMS subject classifications. 65F08, 62D05

DOI. 10.1137/19M129961X

1. Introduction.

Consider a multilayer perceptron (MLP) with L fully connected layers and n data pairs $\{(x_i^0, y_i)\}_{i=1}^n$, where y_i is the label of x_i^0. Given input data point $x_i^0 \in \mathbb{R}^{d_0}$, the output of the MLP is computed via the forward pass,

$$x_i^\ell = s(W_\ell\, x_i^{\ell-1}), \quad \ell = 1, \ldots, L, \tag{1.1}$$

where $x_i^\ell \in \mathbb{R}^{d_\ell}$, $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$, and s is a nonlinear activation function applied to every entry of the input vector. Without loss of generality, (1.1) does not have bias parameters. Otherwise, bias can be included in the weight matrix W_ℓ, and correspondingly vector x_i^ℓ is appended with an additional homogeneous coordinate of value one. For ease of presentation, we assume constant layer size, i.e., $d_\ell \equiv d$, for $\ell = 0, 1, 2, \ldots, L$, so the total number of parameters is $N = d^2L$. Define the weight vector consisting of all weight parameters concatenated together as

$$w = [\texttt{vec}(W_1), \texttt{vec}(W_2), \ldots, \texttt{vec}(W_L)],$$

where $w \in \mathbb{R}^N$ and `vec` is the operator vectorizing matrices.

[*]Received by the editors November 14, 2019; accepted for publication (in revised form) by L. Grigori November 13, 2020; published electronically February 2, 2021.
https://doi.org/10.1137/19M129961X

Funding: The work of the authors was supported by NIH award 5R01NS042645-14; National Science Foundation (NSF) grants CCF-1817048 and CCF-1725743; U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program grant DE-SC0019393; and U.S. Air Force Office of Scientific Research grant FA9550-17-1-0190. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NIH, NSF, DOE, or AFOSR. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF.

[†]University of Texas at Austin, Austin, TX 78705 USA (chenchao.nk@gmail.com, biros@ices.utexas.edu).
[‡]Technical University of Munich, 85748 Munich, Germany (s.reiz@tum.de, bungartz@in.tum.de).
[§]Nvidia Corp., Santa Clara, CA 95051 USA (b94201001@gmail.com).

Given a loss function $f(x_i^L, y_i)$, which measures the misfit between the network output and the true label, we define

$$F(w) = \frac{1}{n}\sum_{i=1}^{n} f\left(x_i^L, y_i\right)$$

as the loss of the MLP with respect to the weight vector w. Note x_i^L is a function of the weights w.

Definition 1.1 ((generalized) Gauss–Newton Hessian). *Let $Q_i = \frac{1}{n}\,\partial_{xx}^2 f(x_i^L, y_i)$ be the Hessian of the loss function $f(x_i^L, y_i)$ for $i = 1, 2, \ldots, n$, and define $Q \in \mathbb{R}^{dn\times dn}$ as a block diagonal matrix with Q_i being the* i*th diagonal block. Let $J_i = \partial_w x_i^L \in \mathbb{R}^{d\times N}$ be the Jacobian of x_i^L with respect to the weights w for $i = 1, 2, \ldots, n$, and define $J \in \mathbb{R}^{dn\times N}$ as the vertical concatenation of all J_i. The (generalized) Gauss–Newton Hessian (GNH) matrix $H \in \mathbb{R}^{N\times N}$ associated with the loss F with respect to the weights w is defined as*

$$H = J^T Q J = \sum_{i=1}^{n} J_i^T Q_i J_i. \tag{1.2}$$

The GNH matrix is closely related to the Hessian matrix in that it is the Hessian matrix of a particular approximation of $F(w)$ constructed by replacing x_i^L with its first-order approximation (on weights w) [30]. Importantly, the GNH matrix is always (symmetric) positive semidefinite when the loss function $f(x_i^L, y_i)$ is convex in x_i^L (Q_i is positive semidefinite), a useful property in many applications. In addition, for several standard choices of the loss function, the GNH matrix is mathematically equivalent to the *Fisher matrix* as used in the natural gradient method.

This paper is concerned with fast entrywise evaluation of the GNH matrix. Such an algorithmic primitive can be used in constructing approximations of the GNH matrix for solving linear systems and eigenvalue problems, which are useful for training and analyzing neural networks [6, 30, 5, 34], for selecting training data to minimize the inference variance [9], for estimating learning rates [25], for network pruning [19], for robust training [41], for probabilistic inference [20], for designing fast solvers [7, 38, 15], and so on.

1.1. Previous work.

We classify related work into two groups. One group avoids entrywise evaluation of the GNH matrix and relies on the matrix-vector multiplication (matvec) with the Hessian or the GNH that is matrix-free [28, 32, 30]. For example, the matrix-free matvec can be used to construct low-rank approximations of the GNH matrix through the randomized singular value decomposition (RSVD) [18], but the numerical rank may not be small [44, 11]. Other examples are the following: [10] introduces a low-rank approximation using the Lanczos algorithm to tackle saddle points, [36] maintains a low-rank approximation of the inverse of the Hessian based on rank-one updates at each optimization step, [15] uses a quasi-Newton-like construction of the low-rank approximation, [43, 40] study the convergence of stochastic Newton methods combined with a randomized low-rank approximation, and [41] uses a matrix-free method with only the layers near the output layer.

The other group of methods are based on evaluating or approximating entries on or close to the diagonal of the GNH matrix [24]. For example, [48] introduces a recursive fast algorithm to construct block-diagonal approximations. As another example, [31, 30] introduce the Kronecker-factored approximate curvature (K-FAC), which is based on an entrywise approximation of the Fisher matrix (mathematically equivalent to the GNH for some popular loss functions). The Fisher matrix is given by

${}^1/_n \sum_{i=1}^n \mathbb{E}_y[g_i(y)g_i(y)^T]$, where g_i is the gradient evaluated for the ith training point x_i^0 and y is sampled from the network's predictive distribution $\propto \exp(-f(x_i^L, y))$. In practice, an extra step of block-diagonal or block-tridiagonal approximation is used for fast inversion purpose. The method has been tested within optimization frameworks on modern supercomputers and has been shown to perform well [35]. However, the sampling in the K-FAC algorithm converges slowly, and block-diagonal approximations do not account for off-diagonal information.

1.2. Contributions. In this paper, we introduce a fast algorithm for entrywise evaluation of the GNH matrix H, i.e., computing

$$H_{km} = e_k^T \, H \, e_m,$$

where e_k and e_m are two canonical bases for $k, m = 1, 2, \ldots, N$. With the fast evaluation, we propose the hierarchical-matrix ($\mathcal{H}$-matrix) approximation [4, 17] of the GNH matrix for the MLP network, which has applications in autoencoders and long-short memory networks and is often used to study the potential of second-order training methods. Notice if the matrix-free matvec is used to evaluate H_{km}, the computational cost would be $\mathcal{O}(Nn)$.

Our fast algorithm includes a precomputation step and a sampling step, which reduces the cost to $\mathcal{O}(n + d)$ work (independent of N), where d is the output dimension of the network. To illustrate the idea, suppose the network employs the mean squared loss, i.e., $f(x_i^L, y_i) = \frac{1}{2}\|x_i^L - y_i\|^2$, and therefore the GNH matrix is $H = \frac{1}{n} J J^T$, where $J \in \mathbb{R}^{dn \times N}$ is the Jacobian of the network output with respect to the weights. Then $H_{km} = \frac{1}{n}(Je_k)^T \, (Je_m)$, and only columns in the Jacobian are required to be computed. Our precomputation algorithm exploits the structure of a feed-forward neural network, where the gradient is back propagated layer by layer, so the intermediate results effectively form a compressed format of the Jacobian with $\mathcal{O}(Nn)$ memory. As a result, every column can be retrieved in only $\mathcal{O}(nd)$ time (note every column has $\mathcal{O}(nd)$ entries).

To accelerate the computation of H_{km}, we introduce a fast Monte Carlo sampling algorithm. Let $v_k(i)$ denote the subvector in the Jacobian's kth column corresponding to the ith data point, and therefore $H_{km} = \frac{1}{n}\sum_{i=1}^n v_k(i)^T v_m(i)$. In the sampling, we draw c (independent of n) independent samples $t_1, t_1, \ldots, t_c$ from $\{1, 2, \ldots, n\}$ with a carefully designed probability distribution P_{km} and compute an estimator

$$\tilde{H}_{km} = \frac{1}{nc} \sum_{j=1}^{c} \frac{v_k(t_j)^T v_m(t_j)}{P_{km}(t_j)}.$$

We prove $|H_{km} - \tilde{H}_{km}| = \mathcal{O}(1/\sqrt{c})$ with high probability. Note it requires only $\mathcal{O}(n{+}dc)$ work to compute $\tilde{H}_{km}$ as an approximation, where d is the output dimension of the network.

With the fast evaluation algorithm, we are able to take advantage of the existing `GOFMM` method [45, 46, 47] to construct the $\mathcal{H}$-matrix approximation of the GNH matrix through evaluating $\mathcal{O}(N)$ entries in the matrix. The $\mathcal{H}$-matrix approximation is a multilevel scheme that stores diagonal blocks and employs low-rank approximations for off-diagonal blocks in the input matrix. So previous work on the (global) low-rank approximation and the block-diagonal approximation can be viewed as the two extremes in the spectrum of our $\mathcal{H}$-matrix approximation, which effectively works for a broader range of problems. $\mathcal{H}$-matrices are algebraic generalizations of the well-known fast n-body calculation algorithms [3, 16] in computational physics, and they have been applied to kernel methods in machine learning [26, 27]. An $\mathcal{H}$-matrix can

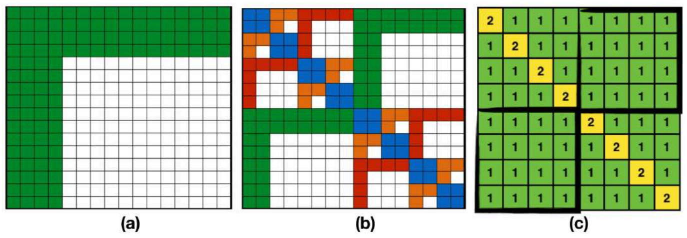

FIG. 1. (a) *A low-rank matrix $H = UV^T$, where U and V are tall-and-skinny matrices;* (b) *a three-level $\mathcal{H}$-matrix, where blue represents dense diagonal blocks and green, red, and orange represent off-diagonal low-rank blocks at levels* 1, 2, *and* 3, *respectively;* (c) *ranks of diagonal and off-diagonal blocks in an $\mathcal{H}$-matrix, where every block has size* 2*-by-*2.

be formulated as

$$H = D + S + UV^T, \tag{1.3}$$

where U and V are tall-and-skinny matrices, S is a block-sparse matrix, and D is a block-diagonal matrix with the blocks being either smaller $\mathcal{H}$-matrices at the next level or dense blocks at the last level. Figure 1 shows the structure of a low-rank matrix and the hierarchically low-rank structure of $\mathcal{H}$-matrices.

Given an $\mathcal{H}$-matrix approximation, the memory footprint is $\mathcal{O}(Nr_o)$,[1] where N is the matrix size or the number of weights in a network and r_o is the maximum off-diagonal rank. Compared to the $\mathcal{O}(N^2)$ storage for the entire matrix, an $\mathcal{H}$-matrix approximation leads to significant memory savings. Once constructed, an $\mathcal{H}$-matrix can be factorized with only $\mathcal{O}(Nr_o^2)$ work, and there exists an entire class of well-established numerical techniques [33, 39, 21, 12, 1, 8, 37]. The factorization can be applied to a vector with $\mathcal{O}(Nr_o)$ work and be used as either a fast direct solver or a preconditioner depending on the approximation accuracy.

To summarize, our work makes the following two major contributions:

- a fast algorithm that requires $\mathcal{O}(Nn)$ storage and requires $\mathcal{O}(n + d/\epsilon^2)$ work to evaluate an arbitrary entry in the GNH matrix, where N and d are the number of parameters and the output dimension of the MLP, respectively; n is the data size; and ϵ is a prescribed accuracy;
- a framework to construct the $\mathcal{H}$-matrix approximation of the GNH matrix, an analysis of the approximation accuracy and the cost, as well as comparison with the RSVD and the K-FAC methods.

Outline. In section 2 we review some background material. In section 3 we present our fast algorithm for evaluating entries in the GNH matrix. In section 4 we show how to construct the $\mathcal{H}$-matrix approximation of the GNH matrix. In section 5 we show numerical results, and in section 6 we conclude with further extensions. Throughout this paper, we use $\|\cdot\|$ to denote the vector/matrix 2-norm and $\|\cdot\|_F$ to denote the matrix Frobenius norm.

2. Background.

In this section, we review the importance of the GNH matrix and the associated computational challenge. The GNH matrix is useful in training and analyzing neural networks, selecting training data, estimating learning rate, and so on. Here we focus on its use in second-order optimization to show the challenge that is common in other applications.

[1]Generally speaking, there may be a $\log(N)$ or $\log^2(N)$ prefactor, as for other complexity results related to $\mathcal{H}$-matrix approximations. But here we focus on the case without such prefactors.

2.1. Neural network training. In an MLP, the weight vector w is obtained via solving the following constrained optimization problem (regularization on w could be added):

$$\min F(w) = \min \frac{1}{n} \sum_{i=1}^{n} f\left(x_i^L, y_i\right) \quad \text{subject to equation (1.1).} \tag{2.1}$$

Recall that f is the loss function and x_i^L is the network output corresponding to input x_i^0, which has label y_i.

To solve for w in problem (2.1), a second-order optimization method solves a sequence of local quadratic approximations of $F(w)$, which requires solving the following linear systems repeatedly:

$$Hp = -g, \tag{2.2}$$

where H is the *curvature matrix* (the Hessian of $F(w)$ in the standard Newton method), $g = \partial_w F$ is the gradient, and p is the update direction. Generally speaking, second-order optimization methods are highly concurrent and could require a much fewer number of iterations to converge than first-order methods, which imply potentially significant speedup on modern distributed computing platforms.

In the Gauss–Newton method, a popular second-order solver, the GNH matrix is employed (with a small regularization) as the curvature matrix in (2.2), which can be solved using the CG method. Since the GNH is mathematically equivalent to the Fisher matrix for several standard choices of the loss, the solution of (2.2) becomes the *natural gradient*, an efficient steepest descent direction in the space of probability distribution with an appropriately defined distance measure [29].

2.2. Back propagation and matrix-free matvec. Table 1 shows the algorithm known as back propagation for evaluating the gradient $g = \partial_w F$ and the matrix-free matvec with the GNH matrix, both of which have complexity $\mathcal{O}(Nn)$. Both algorithms can be derived by introducing Lagrange multipliers z_i^ℓ and $\hat{z}_i^\ell$ for the corresponding weights W_ℓ at every layer [13, 14]. Note that a direct matvec with the full GNH matrix would require $\mathcal{O}(N^2)$ work, not to mention the amount of work to compute the entire matrix.

Based on the two basic ingredients, iterative solvers such as Krylov methods can be used to solve (2.2) as in Hessian-free methods [28, 32]. However, the iteration

TABLE 1
Gradient evaluation and matrix-free matvec with the GNH matrix. Step (a) *of gradient evaluation is the forward pass in* (1.1)*, and steps* (b) *and* (c) *are the well-known back propagation. In the matvec, step* (a) *is known as the linearized forward* ($\hat{x}_i^0 = 0$)*, which computes $J\hat{w}$. Notations: $M_i^\ell = \operatorname{diag}(\dot{s}(W_\ell x_i^{\ell-1}))$, where $\dot{s}$ stands for the derivative; g^ℓ is the gradient for W_ℓ; $\hat{w} = [\mathtt{vec}(\hat{W}_1), \ldots, \mathtt{vec}(\hat{W}_L)]$ is the input of the matvec; and $(H\hat{w})^\ell$ refers to the ℓ_{th} block of the output.*

Evaluate gradient g	Matvec with GNH: $H\hat{w} = J^TQJ\hat{w}$ (Definition 1.1)
(a) $x_i^\ell = s\left(W_\ell x_i^{\ell-1}\right)$ $\forall i, \ell$	(a) $\hat{x}_i^\ell = M_i^\ell \left(W_\ell \hat{x}_i^{\ell-1} + \hat{W}_\ell x_i^{\ell-1}\right)$ $\forall i, \ell$
(b) $z_i^L = \partial_x f(x_i^L, y_i)$, $\forall i$	(b) $\hat{z}_i^L = Q_i \hat{x}_i^L$ $\forall i$
(c) $z_i^{\ell-1} = W_\ell^T M_i^\ell z_i^\ell$ $\forall i, \ell$	(c) $\hat{z}_i^{\ell-1} = W_\ell^T M_i^\ell \hat{z}_i^\ell$ $\forall i, \ell$
(d) $g^\ell = \sum_{i=1}^n \left(M_i^\ell z_i^\ell\right) (x_i^{\ell-1})^T$ $\forall \ell$	(d) $(H\hat{w})^\ell = \sum_{i=1}^n \left(M_i^\ell \hat{z}_i^\ell\right) (x_i^{\ell-1})^T$ $\forall \ell$

count for convergence can grow rapidly in the presence of ill-conditioning, in which case fast solvers or preconditioners for (2.2) are necessary [2, 23, 30].

3. Fast computation of entries in GNH.

This section presents a precomputation algorithm and a fast Monte Carlo algorithm for fast computation of *arbitrary* entries in the GNH matrix of an MLP network.

A naive method. Consider a GNH matrix $H \in \mathbb{R}^{N\times N}$, where an entry H_{km} can be written as

$$H_{km} = e_k^T\, H\, e_m, \tag{3.1}$$

where e_k and e_m are the kth and the mth columns, respectively, of the N-dimensional identity matrix. We can take advantage of the matrix-free matvec with the GNH matrix in Table 1 to compute $H_{km} = e_k^T(He_m)$, which costs the same as one pass of forward propagation plus one pass of backward propagation, i.e., $\mathcal{O}(Nn) = \mathcal{O}(d^2Ln)$ work.

In the following, we introduce a precomputation algorithm that reduces the cost of evaluating an entry in the GHN to $\mathcal{O}(dn)$ work with $\mathcal{O}(Nn)$ memory and a fast Monte Carlo algorithm that further reduces the cost to $\mathcal{O}(n + d/\epsilon^2)$ work, where ϵ is a prescribed accuracy that does not depend on n or N.

3.1. Precomputation algorithm.

The motivation of our precomputation algorithm is to exploit the sparsity of e_k and e_m plus the symmetry of H in (3.1). Recall the definition of H in (1.2), and let $Q_i = R_i^T R_i$ be a symmetric factorization, which can be computed via, e.g., the eigendecomposition or the LDLT factorization with pivoting. We have

$$\begin{aligned} H_{km} &= e_k^T\big(\textstyle\sum_{i=1}^n J_i^T R_i^T R_i J_i\big)e_m = \sum_{i=1}^n (R_iJ_ie_k)^T(R_iJ_ie_m) \\ &:= \textstyle\sum_{i=1}^n v_k(i)^T v_m(i), \end{aligned} \tag{3.2}$$

where $v_k(i)$ and $v_m(i)$ are two d-dimensional vectors,

$$v_k(i) = R_iJ_ie_k, \quad v_m(i) = R_iJ_ie_m, \tag{3.3}$$

for $k, m = 1, 2, \dots, N$ and $i = 1, 2, \dots, n$. We state the following theorem and present the precomputation algorithm in the proof.

THEOREM 3.1. *For an MLP network that has L fully connected layers with constant layer size d (d-by-d weight matrices), every entry H_{km} in the GNH matrix can be computed in $\mathcal{O}(dn)$ time with a precomputation that requires $\mathcal{O}(nN)$ storage and $\mathcal{O}(dnN)$ work.*

Proof. We first compute $x_i^{\ell-1}$ and $M_i^\ell = \operatorname{diag}(\dot{s}(W_\ell x_i^{\ell-1}))$ via the forward pass, i.e., step (a) of gradient evaluation in Table 1, and then we precompute and store

$$C_i^\ell = R_i M_i^L W_L M_i^{L-1} W_{L-1} \cdots M_i^\ell, \quad i = 1, 2, \dots, n;\ \ell = 1, 2, \dots, L. \tag{3.4}$$

Since every C_i^ℓ is a $d\times d$ matrix, the total storage cost is $\mathcal{O}(d^2\, nL) = \mathcal{O}(nN)$, where $N = d^2L$ is the total number of weights. In addition, notice the relation that $C_i^{\ell-1} = C_i^\ell\left(W_\ell M_i^{\ell-1}\right)$, so they can be computed from $\ell = L$ to $\ell = 1$ iteratively, which requires $\mathcal{O}(d^3Ln) = \mathcal{O}(dnN)$ work in total. Note that the forward pass costs $\mathcal{O}(nN)$ work and that computing the symmetric factorizations for Q_i cost $\mathcal{O}(d^3n)$, which is negligible compared to other parts of the computation.

To complete the proof, we show how to compute $v_k(i)$ as defined in (3.3) with $\mathcal{O}(d)$ work. Below we use the same notations as in Table 1, and e_k is the input vector of the matvec corresponding to $\hat{w} = [\texttt{vec}(\hat{W}_1), \dots, \texttt{vec}(\hat{W}_L)]$ in Table 1. Recall step

(a) of the matrix-free matvec (linearized forward) with the GNH in Table 1, and we evaluate $J_i e_k$ as follows:

1. Let $\tau = \lceil k/d^2 \rceil$, $\mu = k \bmod d$, and $\nu = \lceil (k \bmod d^2)/d \rceil$. Since e_k has only one nonzero entry, $\hat{x}_i^\ell = 0$ for $\ell = 1, 2, \ldots, \tau - 1$ because $\hat{W}_\ell$ are all zeros except for $\ell = \tau$. The matrix $\hat{W}_\tau$ has only one nonzero at position (μ, ν) (column-major ordering) as the following:
$$\begin{array}{c} \\ \mu \end{array} \begin{array}{c} \nu \\ \begin{pmatrix} & \vdots & \\ \cdots & 1 & \cdots \\ & \vdots & \end{pmatrix} \end{array} = \hat{W}_\tau.$$
2. Following step (a) of the matvec in Table 1, we have $\hat{x}_i^\tau = M_i^\tau \hat{W}_\tau x_i^{\tau-1}$ at layer τ. Denote $a_i^\tau = \hat{W}_\tau x_i^{\tau-1}$, and we have
$$\begin{aligned} \hat{x}_i^\tau &= M_i^\tau a_i^\tau \\ \hat{x}_i^{\tau+1} &= M_i^{\tau+1} W_{\tau+1} \hat{x}_i^\tau && \text{(since } \hat{W}_{\tau+1} = 0\text{)} \\ &= M_i^{\tau+1} W_{\tau+1} M_i^\tau a_i^\tau \\ &\cdots \\ \hat{x}_i^L &= M_i^L W_L M_i^{L-1} W_{L-1} \cdots M_i^\tau a_i^\tau. \end{aligned}$$
3. Notice that the only nonzero entry in a_i^τ is the μth element, which equals the νth element in $x_i^{\tau-1}$. Therefore,
$$v_k(i) = R_i J_i e_k = R_i \hat{x}_i^L = C_i^\tau a_i^\tau, \tag{3.5}$$
where $C_i^\tau a_i^\tau$ should be interpreted as a scaling of the μth column in C_i^τ by the νth element in $x_i^{\tau-1}$, which costs $\mathcal{O}(d)$ work. ◻

3.2. Fast Monte Carlo algorithm. Recall (3.2), which sums over a large number of data points, and the idea is to sample a subset with a judiciously chosen probability distribution and scale the (partial) sum appropriately to approximate H_{km}. It is important to note that the computation of the probabilities is fast based on the previous precomputation. The fast sampling algorithm is given in Algorithm 3.1.

Algorithm 3.1 Fast Monte Carlo algorithm.

```
1: Input: $\|v_k(i)\|$ and $\|v_m(i)\|$ for $i = 1, 2, \ldots, n$.
2: Compute sampling probabilities for $t = 1, 2, \ldots, n$:
       (3.6)   $P_{km}(t) = \frac{\|v_k(t)\| \, \|v_m(t)\|}{\sum_{j=1}^n \|v_k(j)\| \, \|v_m(j)\|}.$
3: Draw $c$ independent random samples $t_j$ from $\{1, 2, \ldots, n\}$ with replacement.
4: Output:
       (3.7)   $\tilde{H}_{km} = \frac{1}{c} \sum_{j=1}^{c} \frac{v_k(t_j)^T v_m(t_j)}{P_{km}(t_j)}.$
```

Define $v_k = [v_k(1), \ldots, v_k(n)]$ and $v_m = [v_m(1), \ldots, v_m(n)]$ as two vectors in $\mathbb{R}^{dn}$, and (3.2) can be written as the inner product of the two vectors:

$$H_{km} = v_k^T v_m.$$

The following theorem shows that our sampling algorithm returns a good estimator of H_{km}, where the error is measured using $\|v_k\|\|v_m\|$, an upper bound on $|H_{km}|$.

THEOREM 3.2 (sampling error). *Consider an MLP network that has L fully connected layers with constant layer size d (d-by-d weight matrices). For every entry H_{km} in the GNH matrix, Algorithm 3.1 returns an estimator $\tilde{H}_{km}$*

- *that is an unbiased estimator of H_{km}, i.e., $\mathbb{E}[\tilde{H}_{km}] = H_{km}$;*
- *whose variance or mean squared error (MSE) satisfies*

$$\mathrm{Var}[\tilde{H}_{km}] = \mathbb{E}\left[|H_{km} - \tilde{H}_{km}|^2\right] \leq \frac{1}{c}\|v_k\|^2\|v_m\|^2, \tag{3.8}$$

  where c is the number of random samples;
- *whose absolute error satisfies*

$$|H_{km} - \tilde{H}_{km}| \leq \frac{\eta}{\sqrt{c}}\ \|v_k\|\|v_m\|, \tag{3.9}$$

  with probability at least $1-\delta$, where $\delta \in (0,1)$, $\eta = 1 + \sqrt{8\log(1/\delta)}$, and c is the number of random samples.

Proof. Our proof consists of the following three parts.

Unbiased estimator. Define a random variable

$$X_t = \frac{v_k(t)^T v_m(t)}{P(t)},$$

where t is a random sample from $\{1, 2, \ldots, n\}$ with probability distribution $P(t)$ as defined in (3.6). Observe that $\tilde{H}_{km}$ is the mean of c independent identically distributed variables $(X_{t_1}, X_{t_2}, \ldots, X_{t_c})$, and thus

$$\mathbb{E}[\tilde{H}_{km}] = \mathbb{E}[X_t] = \sum_{t=1}^{n} \frac{v_k(t)^T v_m(t)}{P(t)} P(t) = H_{km}.$$

Variance/MSE error. The variance or MSE error of the estimator is the following:

$$\begin{aligned}
&\mathbb{E}\left[|H_{km} - \tilde{H}_{km}|^2\right] = \mathrm{Var}[\tilde{H}_{km}] = \frac{1}{c}\mathrm{Var}[X_t] \\
&= \frac{1}{c}\left(\mathbb{E}[X_t^2] - \mathbb{E}^2[X_t]\right) \\
&= \frac{1}{c}\sum_{t=1}^{n}\left(\frac{v_k(t)^T v_m(t)}{P(t)}\right)^2 P(t) - \frac{H_{km}^2}{c} \\
&\leq \sum_{t=1}^{n} \frac{\left(v_k(t)^T v_m(t)\right)^2}{c\,P(t)} && \text{(drop the last term)} \\
&\leq \sum_{t=1}^{n} \frac{\|v_k(t)\|^2\|v_m(t)\|^2}{c\,P(t)} && \text{(Cauchy–Schwarz)}
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{c}\left(\sum_{t=1}^{n} \|v_k(t)\|\|v_m(t)\|\right)^2 && \text{(equation (3.6))}\\
&\le \frac{1}{c}\left(\sum_{t=1}^{n} \|v_k(t)\|^2\right)\left(\sum_{t=1}^{n} \|v_m(t)\|^2\right) && \text{(Cauchy–Schwarz)}\\
&= \frac{1}{c}\|v_k\|^2\|v_m\|^2.
\end{aligned}
$$

Notice that with Jensen's inequality, we also obtain a bound of the absolute error in expectation:

$$
\mathbb{E}\left[|H_{km} - \tilde{H}_{km}|\right] \le \frac{1}{\sqrt{c}}\|v_k\|\|v_m\|. \tag{3.10}
$$

Concentration result. We will use McDiarmid's (or the Hoeffding-Azuma or bounded differences) inequality to obtain (3.9). See the conditions for the inequality in Appendix A. Define function

$$
F(t_1, t_2, \dots, t_c) = |H_{km} - \tilde{H}_{km}|,
$$

where $t_1, t_2, \dots, t_c$ are random samples, and we show that changing one sample t_i at a time does not affect F too much. Consider changing a sample t_i to t_i' while keeping others the same. The new estimator $\hat{H}_{km}$ differs from $\tilde{H}_{km}$ by only one term. Thus,

$$
\begin{aligned}
|\tilde{H}_{km} - \hat{H}_{km}| &= \left|\frac{v_k(t_i)^T v_m(t_i)}{c\ P(t_i)} - \frac{v_k(t_i')^T v_m(t_i')}{c\ P(t_i')}\right|\\
&\le \left|\frac{v_k(t_i)^T v_m(t_i)}{c\ P(t_i)}\right| + \left|\frac{v_k(t_i')^T v_m(t_i')}{c\ P(t_i')}\right|\\
&\le \frac{\|v_k(t_i)\|\|v_m(t_i)\|}{c\ P(t_i)} + \frac{\|v_k(t_i')\|\|v_m(t_i')\|}{c\ P(t_i')}\\
&= \frac{2}{c}\sum_{j=1}^{n} \|v_k(j)\|\,\|v_m(j)\|\\
&\le \frac{2}{c}\|v_k\|\|v_m\|,
\end{aligned}
$$

where we have used Cauchy–Schwarz inequality twice. Then define $\Delta = \frac{2}{c}\|v_k\|\|v_m\|$; using the triangle inequality, we see

$$
|F(\dots, t_i, \dots) - F(\dots, t_i', \dots)| \le \Delta.
$$

Finally, let $\gamma = \sqrt{2c\log(1/\delta)}\,\Delta$, and we use McDiarmid's inequality to obtain (3.9) as follows:

$$
\begin{aligned}
&\Pr\left[|H_{km} - \tilde{H}_{km}| \ge \frac{\eta}{\sqrt{c}}\ \|v_k\|\|v_m\|\right]\\
&= \Pr\left[|H_{km} - \tilde{H}_{km}| \ge \frac{1}{\sqrt{c}}\|v_k\|\|v_m\| + \gamma\right]\\
&\le \Pr\left[F - \mathbb{E}[F] \ge \gamma\right] && \text{(equation (3.10))}\\
&\le \exp\left(-\frac{\gamma^2}{2c\Delta^2}\right) = \delta && \text{(McDiarmid's inequality).} \qquad \square
\end{aligned}
$$

Remark 3.3. The error ϵ in the approximation of H_{km} depends on only the number of random samples c (but not n) and can be made arbitrarily small as needed. In particular, if $c \geq 1/\epsilon^2$, we have

$$\mathrm{Var}[\tilde{H}_{km}] = \mathbb{E}\left[|H_{km} - \tilde{H}_{km}|^2\right] \leq \epsilon\ \|v_k\|^2\|v_m\|^2,$$

and if $c \geq \eta^2/\epsilon^2$, then with probability at least $1-\delta$, where $\delta \in (0,1)$,

$$|H_{km} - \tilde{H}_{km}| \leq \epsilon\ \|v_k\|\|v_m\|.$$

Furthermore, the error of the entire matrix in the Frobenius norm is

$$\begin{aligned}\|H - \tilde{H}\|_F &\leq \epsilon\sqrt{\sum_k\sum_m \|v_k\|^2\|v_m\|^2} = \epsilon\sum_k \|v_k\|^2 \\ &\overset{(3.2)}{=} \epsilon\sum_k H_{kk} = \epsilon\,\mathrm{trace}(H) \leq \epsilon\ \sqrt{N}\ \|H\|_F.\end{aligned}$$

Remark 3.4. The estimator $\tilde{H}_{km}$ is exact using at most *one* sample when $k = m$. The (trivial) case $H_{kk} = 0$ is implied by the situation that $v_{kk}(i) = 0$ for all i; otherwise, we have $H_{kk} = \|v_k\|^2$, and the sampling probability becomes

$$P_{kk}(t) = \frac{\|v_k(t)\|^2}{\sum_{j=1}^n \|v_k(j)\|^2} = \frac{\|v_k(t)\|^2}{\|v_k\|^2}.$$

Therefore, $\tilde{H}_{kk} = \|v_k(t)\|^2/P_{kk}(t) = H_{kk}$ with any random sample t.

THEOREM 3.5 (computational cost of sampling). *Given the precomputation in Theorem* 3.1*, it requires $\mathcal{O}(nN)$ work to compute $\|v_k(i)\|$ for all i and k as the input of Algorithm* 3.1*, and it requires $\mathcal{O}(n + d/\epsilon^2)$ work to compute every estimator, where ϵ is a prescribed accuracy that does* not *depend on n.*

Proof. Recall from (3.5) that $\|v_k(i)\|$ is proportional to the norm of a column in C_i^ℓ. Since every C_i^ℓ is a d-by-d matrix, computing all the norms requires $\mathcal{O}(d^2nL) = \mathcal{O}(nN)$ work. Once all $\|v_k(i)\|$ have been computed, the sampling probabilities in (3.6) and the estimator in (3.7) require $\mathcal{O}(n)$ and $\mathcal{O}(d/\epsilon^2)$ work, respectively. ◻

4. $\mathcal{H}$-matrix approximation.

This section introduces the $\mathcal{H}$-matrix approximation of the GNH matrix for the MLP. While the low-rank and the block-diagonal approximations focus on the global and the local structure of the problem, respectively, the $\mathcal{H}$-matrix approximation handles both, as they may be equally important.

4.1. Overall algorithm.

Here we take advantage of the `GOFMM` method [45, 46, 47], which evaluates $\mathcal{O}(N)$ entries in a symmetric positive definite (SPD) matrix $H \in \mathbb{R}^{N\times N}$ to construct the $\mathcal{H}$-matrix approximation $H_{\texttt{GOFMM}}$ such that

$$\|H - H_{\texttt{GOFMM}}\|_F \leq \epsilon\,\|H\|_F,$$

where ϵ is a prescribed tolerance.

Since `GOFMM` requires only entrywise evaluation of the input matrix, we apply it with our fast evaluation algorithm to the regularized GNH matrix (note the GNH matrix is symmetric positive semidefinite, so we always add a small regularization of λ times the identity matrix, where λ^2 is the unit roundoff). The overall algorithm

Algorithm 4.1 Compute $\mathcal{H}$-matrix approximation of GNH with GOFMM.

```
Require: training data $\{x_i^0\}_{i=1}^n$, weights in the neural network $w \in \mathbb{R}^N$
Ensure: approximation of the GNH and its factorization
1: Compute $M_i^\ell$ with forward propagation. (step (a) of gradient evaluation in Table 1)
2: Compute $C_i^\ell$ in (3.4). (Theorem 3.1: $\mathcal{O}(Nnd)$ work and $\mathcal{O}(Nn)$ storage)
3: Compute $\|v_k(i)\|$ in (3.5). (Theorem 3.5: $\mathcal{O}(Nn)$ work and $\mathcal{O}(Nn)$ storage)
4: Apply GOFMM and evaluate entries in the GNH matrix through Algorithm 3.1.
   (Theorem 3.5: $\mathcal{O}(n + d/\epsilon^2)$ work/entry)
```

that computes the $\mathcal{H}$-matrix approximation (and approximate factorization) of the GNH matrix using the GOFMM method is shown in Algorithm 4.1.

The error analysis of Algorithm 4.1 is the following. Let $\tilde{H}_\lambda = \tilde{H} + \lambda I$ be computed by Algorithm 3.1, $\lambda > 0$ be a regularization, and $\tilde{H}_{\texttt{GOFMM}}$ be the approximation of $\tilde{H}_\lambda$ computed by GOFMM. Then the error between the output $\tilde{H}_{\texttt{GOFMM}}$ from Algorithm 4.1 and the (regularized) GNH matrix $H_\lambda = H + \lambda I$ is the following (using the triangular equality):

$$\|H_\lambda - \tilde{H}_{\texttt{GOFMM}}\|_F = \|H_\lambda - \tilde{H}_\lambda + \tilde{H}_\lambda - \tilde{H}_{\texttt{GOFMM}}\|_F \leq \|H - \tilde{H}\|_F + \|\tilde{H}_\lambda - \tilde{H}_{\texttt{GOFMM}}\|_F,$$

where the first term is the sampling error from Algorithm 3.1 and the second term is the GOFMM approximation error. For simplicity, we drop the regularization parameter for the rest of this paper.

4.2. GOFMM overview.

Given an SPD matrix H, the GOFMM takes two steps to construct the $\mathcal{H}$-matrix approximation as follows. First of all, a permutation matrix P is computed to reorder the original matrix, which often corresponds to a hierarchical domain decomposition for applications in two- or three-dimensional physical spaces. The recursive domain partitioning is often associated with a tree data structure $\mathcal{T}$. Unlike methods targeting applications in physical spaces, the GOFMM does not require the use of geometric information (thus its name "geometry-oblivious fast multipole method"), which does not exist for neural networks. Instead of relying on geometric information, the GOFMM exploits the algebraic distance measure that is implicitly defined by the input matrix H. As a matter of fact, any SPD matrix $H \in \mathbb{R}^{N\times N}$ is the *Gram matrix* of N *unknown Gram vectors* $\{\phi_i\}_{i=1}^N$ [22]. Therefore, the distance between two row/column indices i and j can be defined as

$$d_{ij} = \sin^2\left(\angle(\phi_i, \phi_j)\right) = 1 - H_{ij}^2/(H_{ii}H_{jj}) \tag{4.1}$$

or

$$d_{ij} = \|\phi_i - \phi_j\| = \sqrt{H_{ii} - 2H_{ij} + H_{jj}}.$$

We refer interested readers to [45] for the discussion and comparison of different distance metrics. With either definition, the GOFMM is able to construct the permutation P and a balanced binary tree $\mathcal{T}$.

The second step is to approximate the reordered matrix P^THP by

$$H_{\texttt{GOFMM}} = \begin{bmatrix} H_{\texttt{ll}} & 0 \\ 0 & H_{\texttt{rr}} \end{bmatrix} + \begin{bmatrix} 0 & S_{\texttt{lr}} \\ S_{\texttt{rl}} & 0 \end{bmatrix} + \begin{bmatrix} 0 & U_{\texttt{lr}}V_{\texttt{lr}}^T \\ U_{\texttt{rl}}V_{\texttt{rl}}^T & 0 \end{bmatrix},$$

where $H_{\texttt{ll}}$ and $H_{\texttt{rr}}$ are two diagonal blocks that have the same structure as $H_{\texttt{GOFMM}}$ unless their sizes are small enough to be treated as dense blocks, which occurs at the leaf level of the tree $\mathcal{T}$; $S_{\texttt{lr}}$ and $S_{\texttt{rl}}$ are block-sparse matrices; and $U_{\texttt{lr}}V_{\texttt{lr}}^T$ and $U_{\texttt{rl}}V_{\texttt{rl}}^T$ are low-rank approximations of the remaining off-diagonal blocks in H. These bases are computed recursively with a postorder traversal of $\mathcal{T}$ using the interpolative decomposition [18] and a nearest neighbor–based fast sampling scheme. There is a trade-off here: While the so-called weak-admissibility criterion sets $S_{\texttt{lr}}$ and $S_{\texttt{rl}}$ to zero and obtains relatively large ranks, the so-called strong-admissibility criterion selects $S_{\texttt{lr}}$ and $S_{\texttt{rl}}$ to be certain subblocks in H corresponding to a few nearest neighbors/indices of every leaf node in $\mathcal{T}$ and achieves smaller (usually constant) ranks.

Here we focus on the *hierarchical semiseparable (HSS)* format among other types of hierarchical matrices. Technically speaking, the HSS format means $S_{\texttt{lr}}$ and $S_{\texttt{rl}}$ are both zero and the bases $U_{\texttt{lr}}/V_{\texttt{lr}}$ and $U_{\texttt{rl}}/V_{\texttt{rl}}$ of a node in $\mathcal{T}$ are recursively defined through the bases of the node's children, i.e., the so-called *nested bases*.

We refer interested readers to [45, 46, 47] for details about the `GOFMM` method.

4.3. Summary and contrast with related work. We summarize the storage and computational complexity of our $\mathcal{H}$-matrix approximation method (HM) and describe its relation with three existing methods, namely, the Hessian-free method (HF) [28, 32], the RSVD [18], and the K-FAC [30, 31]. As before, we assume the MLP network has L layers of constant layer sizes d, so the number of weights is $N = d^2L$. Let n be the number of data points.

HM. The algorithm is given in Algorithm 4.1, where the first three steps require $\mathcal{O}(Nnd)$ work and $\mathcal{O}(Nn)$ storage. Suppose the rank is r_o in the $\mathcal{H}$-matrix approximation. The `GOFMM` needs to call Algorithm 3.1 $\mathcal{O}(Nr_o)$ times, which results in $\mathcal{O}((n + d/\epsilon^2)Nr_o)$ work. Here, ϵ is chosen to be around the same accuracy as the $\mathcal{H}$-matrix approximation with rank r_o. In addition, standard results in the HSS literature [33, 39, 21] state that the factorization requires $\mathcal{O}(Nr_o^2)$ work and $\mathcal{O}(Nr_o)$ storage, which can be applied to solving a linear system with $\mathcal{O}(Nr_o)$ work.

MF. Unlike the other three methods, the MF does not approximate the GNH. It takes advantage of the (exact) matrix-free matvec and utilizes the CG method for solving linear systems. It is based on the two primitives in Table 1, where every iteration costs $\mathcal{O}(Nn)$ work and storage. The number of CG iteration is generally upper bounded by $\mathcal{O}(\sqrt{\kappa})$, where κ is the condition number of the (regularized) GNH matrix.

RSVD. Recall the GNH matrix $H = J^TQJ$. Without loss of generality, assume Q is an identity for ease of description. The algorithm is to compute an approximate SVD of J with the following steps, which naturally leads to an approximate eigenvalue decomposition of H. First, we apply the back propagation in Table 1 with a random Gaussian matrix as input. Second, the QR decomposition of the result is used to estimate the row space of J. Third, the linearized forward is applied to project J onto the approximate row space, and, finally, the SVD is computed on the projection. Overall, the storage is $\mathcal{O}(Nr)$, and the work required is $\mathcal{O}(Nnr + Nr^2 + dnr^2)$, where r is the numerical rank from the QR decomposition. Compared with the HM approximating off-diagonal blocks, the RSVD approximates the entire matrix.

K-FAC. It computes an approximation of the Fisher matrix F (mathematically equivalent to the GNH for some popular loss functions). Let a column vector $g = [\texttt{vec}(g^1), \ldots, \texttt{vec}(g^L)] \in \mathbb{R}^N$ be the gradient and $F = \mathbb{E}[g\, g^T]$ be a L-by-L block matrix with block size d^2-by-d^2. Note the expectation here is taken with respect to both the empirical input data distribution $\hat{Q}_{x^0}$ and the network's predictive

TABLE 2

Asymptotic complexities of the MF, the RSVD, the K-FAC, and the HM with respect to the number of weights N and the data size n ("lower-order" terms not involving Nn are dropped). We assume $r, k, r_o, d < n$, where r, k, and r_o are the parameters in the RSVD, the K-FAC, and the HM, respectively, and d is the (constant/average) layer size. In addition, κ stands for the condition number of the GNH matrix.

	MF	**RSVD**	**K-FAC**	**HM**
construction	-	$\mathcal{O}(Nnr)$	$\mathcal{O}(Nnk)$	$\mathcal{O}(Nn(r_o+d))$
storage	$\mathcal{O}(Nn)$	$\mathcal{O}(Nr)$	$\mathcal{O}(N)$	$\mathcal{O}(Nn)$
solve	$\mathcal{O}(Nn\sqrt{\kappa})$	$\mathcal{O}(Nr)$	$\mathcal{O}(Nd)$	$\mathcal{O}(Nr_o)$

distribution $P_{y|x^\ell}$. In particular, the (ℓ_1,ℓ_2)th block $(\ell_1,\ell_2 = 1,2,\ldots,L)$ is given by

$$F_{\text{block}}(\ell_1,\ell_2) = \mathbb{E}[\text{vec}(g^{\ell_1})\text{vec}(g^{\ell_2})^T]$$

$$= \mathbb{E}[M^{\ell_1} z^{\ell_1} (x^{\ell_1-1})^T \left(M^{\ell_2} z^{\ell_2} (x^{\ell_2-1})^T\right)^T] \tag{4.2}$$

$$= \mathbb{E}[(M^{\ell_1} z^{\ell_1} \otimes x^{\ell_1-1}) \left(M^{\ell_2} z^{\ell_2} \otimes x^{\ell_2-1}\right)^T] \tag{4.3}$$

$$= \mathbb{E}[(M^{\ell_1} z^{\ell_1} \otimes x^{\ell_1-1}) \left((M^{\ell_2} z^{\ell_2})^T \otimes (x^{\ell_2-1})^T\right)] \tag{4.4}$$

$$= \mathbb{E}[M^{\ell_1} z^{\ell_1} (M^{\ell_2} z^{\ell_2})^T \otimes x^{\ell_1-1} (x^{\ell_2-1})^T] \tag{4.5}$$

$$\approx \mathbb{E}[M^{\ell_1} z^{\ell_1} (M^{\ell_2} z^{\ell_2})^T] \otimes \mathbb{E}[x^{\ell_1-1} (x^{\ell_2-1})^T], \tag{4.6}$$

where (4.2) uses the definition of the network gradient in Table 1, (4.3) rewrites the equation using Kronecker products, (4.4) and (4.5) use the properties of the Kronecker product, and (4.6) assumes the statistical independence between the two terms (see section 6.3.1 in [30]). In (4.6), the former expectation is taken with respect to both $\hat{Q}_{x^0}$ and $P_{y|x^\ell}$, and the latter is taken with respect to $\hat{Q}_{x^0}$. To compute the first expectation, k samples are drawn from the distribution $P_{y|x} \propto \exp(-f(x_i^L, y))$, where x_i^L is the network's output corresponding to input x_i^0. In practice, an additional block-diagonal or block-tridiagonal approximation of the inverse is employed for fast solution of linear systems. The main cost of the algorithm is constructing, updating, and inverting $\mathcal{O}(L)$ matrices of size d-by-d, which requires $\mathcal{O}(d^2L) = \mathcal{O}(N)$ storage and $\mathcal{O}((nk+d)N)$ work. Overall, the approximation error of K-FAC has three components: the error of making the assumption (4.6), the sampling error from approximating the expectations in (4.6) and the error of block-diagonal or block-tridiagonal approximation of the inverse.

We summarize the asymptotic complexities of the four methods discussed above in Table 2.

5. Experimental results.

In this section, we show (1) the cost and the accuracy of our $\mathcal{H}$-matrix approximations, (2) the memory savings from using the precomputation algorithm ($\mathcal{O}(N^2) \to \mathcal{O}(Nn)$), and (3) the efficiency of the fast sampling algorithm. In Algorithm 4.1, the first two steps (precomputation) are implemented in MATLAB for the convenience of extracting intermediate values of neural networks, and the last two steps (sampling) are implemented in C++ (`GOFMM` is written in C++).

Networks and datasets. We focus on classification networks and autoencoder networks with the MNIST and CIFAR-10 datasets. In the following, we denote networks' layer sizes as $d_1 \to d_2 \to \cdots \to d_L$ from the input layer to the output layer. Every network has been trained using the stochastic gradient descent for a few steps, so the weights are not random:

1. "classifier": classification networks with the ReLU activation and the cross-entropy loss:
   (a) N=15,910; MNIST dataset; layer sizes: 784→20→10;
   (b) N=61,670; CIFAR-10 dataset; layer sizes: 3072→20→10;
   (c) N=219,818; MNIST dataset; layer sizes: 784→256→64→32→10;
   (d) N=1,643,498; CIFAR-10 dataset; layer sizes: 3072→512→128→32→10.
2. "AE": autoencoder networks with the softplus activation (sigmoid activation at the last layer) and the mean-squared loss:
   (a) N=16,474; MNIST dataset; layer sizes: 784→10→784;
   (b) N=64,522; CIFAR-10 dataset; layer sizes: 3072→10→3072;
   (c) N=125,972; CIFAR-10 dataset; layer sizes: 3072→20→3072.

GOFMM parameters. We employ the default "angle" distance metric in (4.1) and focus on three parameters in the GOFMM that control the accuracy of the $\mathcal{H}$-matrix approximation: (1) the leaf node size m of the hierarchical partitioning $\mathcal{T}$ (equivalent to setting the number of tree levels), (2) the maximum rank r_o of off-diagonal blocks, and (3) the accuracy τ of low-rank approximations. In particular, we ran GOFMM with two different accuracies: "low" ($m = 128$, $r_o = 128$, $\tau = 5E$-2) and "high" ($m = 1024$, $r_o = 1024$, $\tau = 1E$-5).

GOFMM results. We report the following results for our approach:

- t_{build}: time of constructing the $\mathcal{H}$-matrix approximation of the GNH matrix (not including precomputation time);
- t_{matv}: time of applying the $\mathcal{H}$-matrix approximation to 128 random vectors;
- %K: compression rate of the $\mathcal{H}$-matrix approximation, i.e., ratio between the $\mathcal{H}$-matrix storage and the GNH matrix storage;
- ϵ_F: relative error of the $\mathcal{H}$-matrix approximation measured in Frobenius norm, estimated by $\|Hx - H_{\texttt{GOFMM}}\, x\|_F / \|Hx\|_F$, where $x \in \mathbb{R}^{N\times 128}$ is a Gaussian random matrix.

5.1. Cost and accuracy of $\mathcal{H}$-matrix approximation.

Table 3 shows results of our $\mathcal{H}$-matrix approximations for networks that have relatively small numbers of parameters. The GNH matrices are computed and fully stored in memory.

As Table 3 shows, the approximation can achieve four digits' accuracy except for one network (two digits) when the accuracy of low-rank approximations is 1E-5. Since we have enforced the maximum rank r_o, the runtime of constructing $\mathcal{H}$-matrix approximations (t_{build}) increases proportionally to the number of network parameters, and the compression rate scales inverse proportionally to the number of parameters. The reported construction time t_{build} includes the cost of creating an implicit tree data structure $\mathcal{T}$ in GOFMM, which is less than 20% of t_{build}. In addition, applying the $\mathcal{H}$-matrix approximations to 128 random vectors took less than one second for the five networks. These $\mathcal{H}$-matrix approximations can be factorized in linear time for solving linear systems and eigenvalue problems.

Comparison with RSVD and K-FAC. We implemented the RSVD using Keras and TensorFlow for fast back propogation, and we implemented the K-FAC in MATLAB for the convenience of extracting intermediate values. Table 4 shows the accuracies of our method (HM), the RSVD and the K-FAC under about the same compression rate for the low- and high-accuracy settings, respectively. For the RSVD, the storage is rN entries, where r is the numerical rank of the (symmetric) GNH matrix, so the compression rate is r/N. For the K-FAC, we use the relatively more accurate block tridiagonal version. The compression rate of the K-FAC is defined as k/N, and the reason is that the construction of the RSVD and the K-FAC requires the same

TABLE 3

Timing (t_{build} and t_{matv}), compression rate (%K), and accuracy (ϵ_F) of $\mathcal{H}$-matrix approximations corresponding to low- and high-accuracy settings, respectively. t_{build} is the time of applying the `GOFMM` *on the GNH matrices corresponding to* 1,000 *data points. Experiments performed on one node from the Texas Advanced Computing Center "Stampede 2" system, which has two sockets with* 48 *cores of Intel Xeon Platinum* 8160/*"Skylake" and* 192 *GB of RAM.*

#	network	N	`accuracy`	t_{build}	t_{matv}	%K	ϵ_F
1	classifier (a)	16k	`low`	0.24	0.03	1.80%	1.5E-1
2			`high`	5.74	0.11	13.59%	4.4E-4
3	classifier (b)	61k	`low`	0.42	0.08	0.57%	4.7E-1
4			`high`	13.28	0.33	4.78%	4.0E-2
5	AE (a)	16k	`low`	0.27	0.03	1.25%	1.2E-1
6			`high`	5.67	0.10	11.38%	6.5E-4
7	AE (b)	64k	`low`	0.43	0.08	0.53%	5.5E-3
8			`high`	13.26	0.38	4.62%	6.6E-4
9	AE (c)	126k	`low`	0.87	0.17	0.28%	4.1E-3
10			`high`	24.10	0.94	2.32%	5.2E-4

TABLE 4

Comparison of accuracies (ϵ_F) among the $\mathcal{H}$-matrix approximation (HM), the RSVD and the K-FAC with about the same compression rate (%K) for the low- and high-accuracy settings, respectively. For the RSVD and the K-FAC, the compression rate means r/N and k/N, respectively, where r is the rank and k is the number of random samples. For all cases, the GNH matrices correspond to 10,000 *data points ($n = 10,000$ in* (1.2)*).*

		HM-low	HM-high	RSVD-low	RSVD-high	K-FAC-low	K-FAC-high
AE(a)	%K	1.23%	11.77%	1.40%	12.14%	1.40%	12.14%
	ϵ_F	1.7E-1	4.7E-4	4.3E-1	5.1E-3	1.2E-1	7.3E-2
AE(b)	%K	0.53%	4.62%	0.62%	4.65%	0.62%	4.65%
	ϵ_F	5.7E-3	6.4E-4	8.4E-1	2.3E-1	1.7E-1	3.8E-2
AE(c)	%K	0.28%	2.31%	0.32%	2.38%	0.32%	2.38%
	ϵ_F	4.2E-3	4.9E-4	9.1E-1	2.1E-1	1.6E-1	4.1E-2

number of back propagation if $k = r$ (recall Table 2). So we choose k and r to be the same value such that the corresponding compression rate of the RSVD and the K-FAC are slightly higher than the HM.

As Table 4 shows, the $\mathcal{H}$-matrix approximation achieved higher accuracy than the RSVD and the K-FAC for most cases, especially for the high-accuracy setting. For the RSVD, suppose the eigenvalues of the GHN matrix are $\{\sigma_i\}_i$ and the error of the rank-k approximation measured in the Frobenius norm is proportional to $(\sum_{i>k} \sigma_i^2)^{1/2}$. For autoencoders (b) and (c), the spectrums of the GNH matrices decay slowly, so the RSVD is not efficient. For the K-FAC, the approximation that the expectation of a Kronecker product equals the Kronecker product of expectations (4.6) is, in general, not exact, impeding the overall accuracy of the method.

5.2. Memory savings. Table 5 shows the memory footprint between our precomputation (3.4) and the full GNH matrix, i.e., $\mathcal{O}(N^2)$. Recall from Theorem 3.1 that the storage of our precomputation is $\mathcal{O}(nN)$, where n is the number of data points.

Table 5
Comparison of memory footprint (in single precision) between our precomputation (3.4) *and the full GNH matrix. For each network, we show the compression rate and accuracy of $\mathcal{H}$-matrix approximations for two levels of accuracies. For both cases, the GNH matrices correspond to $n =$ 10,000 data points.*

	N	M_{ours}	M_{GNH}	`accuracy`	%K	ϵ_F
classifier (c)	219k	191 MB	193 GB	`low`	0.165%	3.1E-1
				`high`	1.268%	4.2E-2
classifier (d)	1.6m	423 MB	10.8 TB	`low`	0.012%	1.2E-1
				`high`	0.177%	2.2E-2

Table 6
Accuracy of our fast Monte Carlo (FMC) sampling scheme. The error $\epsilon_K = \|H - \tilde{H}\|_F / \|H\|_F$, where H and $\tilde{H}$ are the exact GNH matrix and its approximation computed using K random samples in Algorithm 3.1*, respectively. The exact GNH matrices correspond to the AE* (a) *network with the entire MNIST dataset and the class* (a) *network with the entire CIFAR-*10 *training dataset, respectively. The reference uniform sampling scheme uses a uniform sampling probability instead of* (3.6) *in Algorithm* 3.1.

	n	`scheme`	ϵ_{10}	ϵ_{100}	$\epsilon_{1,000}$	$\epsilon_{10,000}$
MNIST	60,000	uniform	3.6E-1	1.1E-1	3.6E-1	1.1E-2
		FMC	9.7E-3	3.1E-3	9.6E-4	3.1E-4
CIFAR-10	50,000	uniform	9.7E-1	3.1E-1	9.8E-2	3.6E-2
		FMC	6.1E-1	1.9E-1	6.1E-2	1.9E-2

As Table 5 shows, our precomputation leads to huge memory reduction compared with storing the full GNH matrix. This allows using the `GOFMM` method for networks that have a large number of parameters. For example, the storage of the GNH matrix for the classifier (d) network requires more than 10 TB! But we were able to run `GOFMM` with the compressed storage (at the price of spending $\mathcal{O}(dn)$ work for the evaluation of every entry). The precomputation of (3.4) took merely about 2 seconds and 7 seconds, respectively.

5.3. Fast Monte Carlo sampling. We show the accuracy of our fast Monte Carlo sampling scheme. The relative error measured in the Frobenius norm is between the exact GNH matrix H and the approximation $\tilde{H}$ computed using Algorithm 3.1 with a prescribed number of random samples. For reference, we also run the same sampling scheme but with a uniform probability distribution.

As Table 6 shows, when the number of random samples increases by 100, the accuracy improves by 10, which confirms the standard convergence rate of Monte Carlo in Theorem 3.2. Importantly, the error bound and the convergence rate do *not* depend on the problem size n. Moreover, our sampling scheme outperforms the uniform sampling by at most two orders of magnitude for the MNIST dataset. In other words, the uniform sampling requires 100 times more random samples to achieve about the same accuracy as our sampling scheme.

$\mathcal{H}$-matrix approximation with sampling. Table 7 shows the error of Algorithm 4.1 for a sequence of increasingly large number of random samples. Recall that Algorithm 4.1 computes the $\mathcal{H}$-matrix approximation $\tilde{H}_{\texttt{GOFMM}}$ for the (inexact) GNH matrix, namely, $\tilde{H}$, from Algorithm 3.1. The error between the $\mathcal{H}$-matrix approximation $\tilde{H}_{\texttt{GOFMM}}$ and the exact GNH matrix, namely, H, is bounded as below (using the trian-

TABLE 7

$\mathcal{H}$-matrix approximation with sampling. The exact GNH matrices correspond to the AE (a) *network with the entire MNIST dataset and the class* (a) *network with the entire CIFAR-*10 *training dataset, respectively. The compression rate and the accuracy are shown for low- and high-accuracy settings, respectively.*

(a) MNIST dataset ($n = 60,000$ images)

	10 samples		100 samples		1,000 samples		10,000 samples	
`accuracy`	%K	ϵ_F	%K	ϵ_F	%K	ϵ_F	%K	ϵ_F
`low`	1.74%	2.8E-1	1.69%	1.3E-1	1.68%	1.4E-1	1.68%	6.1E-2
`high`	17.1%	2.2E-2	16.9%	9.5E-3	16.6%	2.7E-3	16.2%	9.6E-4

(b) CIFAR-10 training dataset ($n = 50,000$ images)

	10 samples		100 samples		1,000 samples		10,000 samples	
`accuracy`	%K	ϵ_F	%K	ϵ_F	%K	ϵ_F	%K	ϵ_F
`low`	0.61%	9.7E-1	0.61%	5.3E-1	0.61%	4.1E-1	0.61%	4.3E-1
`high`	4.83%	9.7E-1	4.83%	3.6E-1	4.83%	1.9E-1	4.83%	7.3E-2

gular equality)

$$\|H - \tilde{H}_{\texttt{GOFMM}}\|_F = \|H - \tilde{H} + \tilde{H} - \tilde{H}_{\texttt{GOFMM}}\|_F \leq \|H - \tilde{H}\|_F + \|\tilde{H} - \tilde{H}_{\texttt{GOFMM}}\|_F,$$

where the first term is the sampling error and the second term is the $\mathcal{H}$-matrix approximation error. As Table 6 shows, the former converges to zero and is independent of the data size. Table 7 shows that the latter also converges as the sampling becomes increasingly accurate, which justifies the overall approach.

6. Conclusions.

We have presented a fast method to evaluate entries in the GNH matrix of the MLP network, and our method consists of two parts: a precomputation algorithm and a fast Monte Carlo algorithm. While the precomputation allows evaluating entries in the GNH matrix exactly with reduced storage, the random sampling is based on the precomputation and further accelerates the evaluation. Let N be the number of weights, n be the data size, and d be the constant layer size. Our scheme requires $\mathcal{O}(n + d/\epsilon^2)$ work for any entry in the GNH matrix H, where ϵ is the accuracy, whereas the worst-case complexity to evaluate an entry exactly is $\mathcal{O}(Nn)$ through the matrix-free matvec. For example, the evaluation of $H_{N,N}$ would require $\mathcal{O}(Nn)$ work, while given our precomputation, it requires only $\mathcal{O}(n)$ work to compute a diagonal entry exactly (Remark 3.4). One application of this fast diagonal evaluation would be computing all the diagonals of H to precondition/accelerate the training of neural networks [42]. In this paper, we focused on applying the `GOFMM` to construct the $\mathcal{H}$-matrix approximation for the GNH matrix. As a result, we obtain an $\mathcal{H}$-matrix and its factorization for solving linear systems and eigenvalue problems with the GNH.

Two important directions for future research are (1) extending our method to other types of networks, such as convolutional networks, where weight matrices are highly structured (preliminary experiments on the VGG network show similar results as those in Table 3), and (2) incorporating our method in the context of a learning task, which would also require several algorithmic choices related to optimization, such as initialization, damping, and adding momentum.

Appendix A. McDiarmid's inequality.

THEOREM A.1. *Let $X_1, X_2, \dots, X_n$ be independent random variables taking values in the set $\mathcal{X}$. If a mapping $F : \mathcal{X}^n \to \mathbb{R}$ satisfies*

$$\sup_{x_1,\dots,x_i,x_i',\dots,x_n} |F(x_1,\dots,x_i,\dots,x_n) - F(x_1,\dots,x_i',\dots,x_n)| \le \Delta_i \quad \forall i,$$

where $x_1,\dots,x_i,x_i',\dots,x_n \in \mathcal{X}$, then for all $\epsilon > 0$,

$$Pr(f - \mathbb{E}[f] \ge \epsilon) \le exp\left(\frac{-2\epsilon^2}{\sum_{i=1}^n \Delta_i^2}\right).$$

REFERENCES

[1] A. Aminfar, S. Ambikasaran, and E. Darve, *A fast block low-rank dense solver with applications to finite-element matrices*, J. Comput. Phys., 304 (2016), pp. 170–188.

[2] O. Axelsson, *Iterative Solution Methods*, Cambridge University Press, Cambridge, 1994.

[3] J. Barnes and P. Hut, *A hierarchical O(NlogN) force-calculation algorithm*, Nature, 324 (1986), pp. 446–449.

[4] M. Bebendorf, *Hierarchical Matrices*, Springer-Verlag, Berlin, 2008.

[5] L. Bottou, F. E. Curtis, and J. Nocedal, *Optimization methods for large-scale machine learning*, SIAM Rev., 60 (2018), pp. 223–311.

[6] R. H. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal, *On the use of stochastic Hessian information in optimization methods for machine learning*, SIAM J. Optim., 21 (2011), pp. 977–995.

[7] Y. Carmon and J. C. Duchi, *Analysis of Krylov subspace solutions of regularized nonconvex quadratic problems*, in Advances in Neural Information Processing Systems, 2018, pp. 10728–10738, https://arxiv.org/pdf/1806.09222.pdf.

[8] C. Chen, H. Pouransari, S. Rajamanickam, E. G. Boman, and E. Darve, *A distributed-memory hierarchical solver for general sparse linear systems*, Parallel Comput., 74 (2018), pp. 49–64.

[9] D. A. Cohn, *Neural Network Exploration using Optimal Experiment Design*, in Advances in Neural Information Processing Systems, 1994, pp. 679–686.

[10] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*, in Proceedings of the 27th International Conference on Neural Information Processing Systems, Vol. 2, NIPS'14, MIT Press, Cambridge, MA, 2014, pp. 2933–2941, http://dl.acm.org/citation.cfm?id=2969033.2969154.

[11] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio, *Sharp Minima Can Generalize for Deep Nets*, preprint, arXiv:1703.04933, 2017.

[12] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov, *An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling*, SIAM J. Sci. Comput., 38 (2016), pp. S358–S384, https://doi.org/10.1137/15M1010117.

[13] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press, New York, 1981.

[14] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*, Vol. 1, MIT Press, Cambridge, MA, 2016.

[15] R. M. Gower, N. L. Roux, and F. Bach, *Tracking the Gradients Using the Hessian: A New Look at Variance Reducing Stochastic Methods*, preprint, arXiv:1710.07462, 2017.

[16] L. Greengard, *Fast algorithms for classical physics*, Science, 265 (1994), pp. 909–914.

[17] W. Hackbusch, *Hierarchical Matrices: Algorithms and Analysis*, Springer Series in Computational Mathematics 49, Springer-Verlag, Berlin, 2015.

[18] N. Halko, P.-G. Martinsson, and J. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Rev., 53 (2011), pp. 217–288.

[19] B. Hassibi and D. G. Stork, *Second order derivatives for network pruning: Optimal brain surgeon*, in Advances in Neural Information Processing Systems, 1993, pp. 164–171.

[20] G. Hennequin, L. Aitchison, and M. Lengyel, *Fast sampling-based inference in balanced neuronal networks*, in Advances in Neural Information Processing Systems, 2014, pp. 2240–2248.

[21] K. L. Ho and L. Ying, *Hierarchical Interpolative Factorization for Elliptic Operators: Integral Equations*, preprint, arXiv:1307.2666, 2013.

[22] T. Hofmann, B. Schölkopf, and A. J. Smola, *Kernel methods in machine learning*, Ann. Statist., (2008), pp. 1171–1220.

[23] D. A. Knoll and D. E. Keyes, *Jacobian-free Newton-Krylov methods: A survey of approaches and applications*, J. Comput. Phys., 193 (2004), pp. 357–397.

[24] J. Lafond, N. Vasilache, and L. Bottou, *Diagonal Rescaling for Neural Networks*, preprint, arXiv:1705.09319, 2017.

[25] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, *Efficient backprop*, in Neural Networks: Tricks of the Trade, Springer-Verlag, Berlin, 1998, pp. 9–50.

[26] D. Lee and A. G. Gray, *Fast high-dimensional kernel summations using the Monte Carlo multipole method*, in Proceedings of the 22nd Annual Conference on Neural Information Processing Systems (NIPS), 2008, pp. 929–936.

[27] W. B. March, B. Xiao, C. D. Yu, and G. Biros, *Askit: An efficient, parallel library for high-dimensional kernel summations*, SIAM J. Sci. Comput., 38 (2016), pp. S720–S749, https://doi.org/10.1137/15M1026468.

[28] J. Martens, *Deep learning via Hessian-free optimization*, in International Conference on Machine Learning, Vol. 27, 2010, pp. 735–742.

[29] J. Martens, *New Insights and Perspectives on the Natural Gradient Method*, preprint, arXiv:1412.1193, 2014.

[30] J. Martens, *Second-Order Optimization for Neural Networks*, Ph.D. thesis, University of Toronto, Toronto, ON, Canada, 2016.

[31] J. Martens and R. Grosse, *Optimizing neural betworks with Kronecker-factored approximate curvature*, in International Conference on Machine Learning, 2015, pp. 2408–2417.

[32] J. Martens and I. Sutskever, *Learning recurrent neural networks with Hessian-free optimization*, in Proceedings of the 28th International Conference on Machine Learning (ICML-11), Citeseer, 2011, pp. 1033–1040.

[33] P.-G. Martinsson and V. Rokhlin, *A fast direct solver for boundary integral equations in two dimensions*, J. Comput. Phys., 205 (2005), pp. 1–23.

[34] T. O'Leary-Roseberry, N. Alger, and O. Ghattas, *Inexact Newton Methods for Stochastic Non-Convex Optimization with Applications to Neural Network Training*, preprint, arXiv:1905.06738, 2019.

[35] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, R. Yokota, and S. Matsuoka, *Second-Order Optimization Method for Large Mini-Batch: Training Resnet-*50 *on Imagenet in* 35 *Epochs*, preprint, arXiv:1811.12019, 2018.

[36] N. L. Roux, P.-A. Manzagol, and Y. Bengio, *Topmoumoute online natural gradient algorithm*, in Advances in Neural Information Processing Systems, 2008, pp. 849–856.

[37] T. Takahashi, C. Chen, and E. Darve, *Parallelization of the Inverse Fast Multipole Method with an Application to Boundary Element Method*, preprint, arXiv:1905.10602, 2019.

[38] N. Tripuraneni, M. Stern, C. Jin, J. Regier, and M. I. Jordan, *Stochastic cubic regularization for fast nonconvex optimization*, in Advances in Neural Information Processing Systems, 2018, pp. 2904–2913, http://papers.nips.cc/paper/7554-stochastic-cubic-regularization-for-fast-nonconvex-optimization.pdf.

[39] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, *Fast algorithms for hierarchically semiseparable matrices*, Numer. Linear Algebra Appl., 17 (2010), pp. 953–976.

[40] P. Xu, J. Yang, F. Roosta-Khorasani, C. Ré, and M. W. Mahoney, *Sub-sampled Newton methods with non-uniform sampling*, in Advances in Neural Information Processing Systems, 2016, pp. 3000–3008.

[41] Z. Yao, A. Gholami, Q. Lei, K. Keutzer, and M. W. Mahoney, *Hessian-Based Analysis of Large Batch Training and Robustness to Adversaries*, preprint, arXiv:1802.08241, 2018.

[42] Z. Yao, A. Gholami, S. Shen, K. Keutzer, and M. W. Mahoney, *Adahessian: An Adaptive Second Order Optimizer for Machine Learning*, preprint, arXiv:2006.00719, 2020.

[43] H. Ye, L. Luo, and Z. Zhang, *Approximate Newton methods and their local convergence*, in Proceedings of the 34th International Conference on Machine Learning, Proceedings of Machine Learning Research, International Convention Centre, Sydney, Australia, 2017, pp. 3931–3939, http://proceedings.mlr.press/v70/ye17a.html.

[44] Y. You, Z. Zhang, C. Hsieh, J. Demmel, and K. Keutzer, *Imagenet Training in Minutes*, CoRR, abs/1709.05011, 2017.

[45] C. D. Yu, J. Levitt, S. Reiz, and G. Biros, *Geometry-oblivious FMM for compressing dense SPD matrices*, in Proceedings of SC17, AMC SCxy Conference series, IEEE, Denver, CO, 2017, https://doi.org/10.1145/3126908.3126921.

[46] C. D. Yu, S. Reiz, and G. Biros, *Distributed-memory hierarchical compression of dense SPD matrices*, in Proceedings of the International Conference for High Performance Computing,

Networking, Storage, and Analysis, SC '18, IEEE Press, Piscataway, NJ, 2018, pp. 15:1–15:15, https://dl.acm.org/citation.cfm?id=3291676.

[47] C. D. Yu, S. Riesz, and J. Levitt, `GOFMM` *Home Page*, https://github.com/ChenhanYu/hmlp, 2018.

[48] H. Zhang, C. Xiong, J. Bradbury, and R. Socher, *Block-Diagonal Hessian-Free Optimization for Training Neural Networks*, preprint, arXiv:1712.07296, 2017.

The original version of this article follows.

SIAM J. Matrix Anal. Appl. © 2021 Society for Industrial and Applied Mathematics
Vol. 42, No. 1, pp. 165–184

FAST APPROXIMATION OF THE GAUSS–NEWTON HESSIAN MATRIX FOR THE MULTILAYER PERCEPTRON[*]

CHAO CHEN[†], SEVERIN REIZ[‡], CHENHAN D. YU[§], HANS-JOACHIM BUNGARTZ[‡], AND GEORGE BIROS[†]

Abstract. We introduce a fast algorithm for entrywise evaluation of the Gauss–Newton Hessian (GNH) matrix for the fully connected feed-forward neural network. The algorithm has a precomputation step and a sampling step. While it generally requires $\mathcal{O}(Nn)$ work to compute an entry (and the entire column) in the GNH matrix for a neural network with N parameters and n data points, our fast sampling algorithm reduces the cost to $\mathcal{O}(n+d/\epsilon^2)$ work, where d is the output dimension of the network and ϵ is a prescribed accuracy (independent of N). One application of our algorithm is constructing the hierarchical-matrix ($\mathcal{H}$-matrix) approximation of the GNH matrix for solving linear systems and eigenvalue problems. It generally requires $\mathcal{O}(N^2)$ memory and $\mathcal{O}(N^3)$ work to store and factorize the GNH matrix, respectively. The $\mathcal{H}$-matrix approximation requires only $\mathcal{O}(Nr_o)$ memory footprint and $\mathcal{O}(Nr_o^2)$ work to be factorized, where $r_o \ll N$ is the maximum rank of off-diagonal blocks in the GNH matrix. We demonstrate the performance of our fast algorithm and the $\mathcal{H}$-matrix approximation on classification and autoencoder neural networks.

Key words. Gauss–Newton Hessian, fast Monte Carlo sampling, hierarchical matrix, second-order optimization, multilayer perceptron

AMS subject classifications. 65F08, 62D05

DOI. 10.1137/19M129961X

1. Introduction.

Consider a multilayer perceptron (MLP) with L fully connected layers and n data pairs $\{(x_i^0, y_i)\}_{i=1}^n$, where y_i is the label of x_i^0. Given input data point $x_i^0 \in \mathbb{R}^{d_0}$, the output of the MLP is computed via the forward pass,

$$x_i^\ell = s(W_\ell\, x_i^{\ell-1}), \quad \ell = 1, \dots, L, \tag{1.1}$$

where $x_i^\ell \in \mathbb{R}^{d_\ell}$, $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$, and s is a nonlinear activation function applied to every entry of the input vector. Without loss of generality, (1.1) does not have bias parameters. Otherwise, bias can be included in the weight matrix W_ℓ, and correspondingly vector x_i^ℓ is appended with an additional homogeneous coordinate of value one. For ease of presentation, we assume constant layer size, i.e., $d_\ell \equiv d$, for $\ell = 0, 1, 2, \dots, L$, so the total number of parameters is $N = d^2 L$. Define the weight vector consisting of all weight parameters concatenated together as

$$w = [\texttt{vec}(W_1), \texttt{vec}(W_2), \dots, \texttt{vec}(W_L)],$$

where $w \in \mathbb{R}^N$ and `vec` is the operator vectorizing matrices.

[*]Received by the editors November 14, 2019; accepted for publication (in revised form) by L. Grigori November 13, 2020; published electronically February 2, 2021.
https://doi.org/10.1137/19M129961X

Funding: The work of the authors was supported by NIH award 5R01NS042645-14; National Science Foundation (NSF) grants CCF-1817048 and CCF-1725743; U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program grant DE-SC0019393; and U.S. Air Force Office of Scientific Research grant FA9550-17-1-0190. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NIH, NSF, DOE, or AFOSR. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF.

[†]University of Texas at Austin, Austin, TX 78705 USA (chenchao.nk@gmail.com, biros@ices.utexas.edu).
[‡]Technical University of Munich, 85748 Munich, Germany (s.reiz@tum.de, bungartz@in.tum.de).
[§]Nvidia Corp., Santa Clara, CA 95051 USA (b94201001@gmail.com).

Given a loss function $f(x_i^L, y_i)$, which measures the misfit between the network output and the true label, we define

$$F(w) = \frac{1}{n}\sum_{i=1}^{n} f\left(x_i^L, y_i\right)$$

as the loss of the MLP with respect to the weight vector w. Note x_i^L is a function of the weights w.

Definition 1.1 ((generalized) Gauss–Newton Hessian). *Let $Q_i = \frac{1}{n}\,\partial^2_{xx} f(x_i^L, y_i)$ be the Hessian of the loss function $f(x_i^L, y_i)$ for $i = 1, 2, \ldots, n$, and define $Q \in \mathbb{R}^{dn\times dn}$ as a block diagonal matrix with Q_i being the* i*th diagonal block. Let $J_i = \partial_w x_i^L \in \mathbb{R}^{d\times N}$ be the Jacobian of x_i^L with respect to the weights w for $i = 1, 2, \ldots, n$, and define $J \in \mathbb{R}^{dn\times N}$ as the vertical concatenation of all J_i. The (generalized) Gauss–Newton Hessian (GNH) matrix $H \in \mathbb{R}^{N\times N}$ associated with the loss F with respect to the weights w is defined as*

$$H = J^T Q J = \sum_{i=1}^{n} J_i^T Q_i J_i. \tag{1.2}$$

The GNH matrix is closely related to the Hessian matrix in that it is the Hessian matrix of a particular approximation of $F(w)$ constructed by replacing x_i^L with its first-order approximation (on weights w) [30]. Importantly, the GNH matrix is always (symmetric) positive semidefinite when the loss function $f(x_i^L, y_i)$ is convex in x_i^L (Q_i is positive semidefinite), a useful property in many applications. In addition, for several standard choices of the loss function, the GNH matrix is mathematically equivalent to the *Fisher matrix* as used in the natural gradient method.

This paper is concerned with fast entrywise evaluation of the GNH matrix. Such an algorithmic primitive can be used in constructing approximations of the GNH matrix for solving linear systems and eigenvalue problems, which are useful for training and analyzing neural networks [6, 30, 5, 34], for selecting training data to minimize the inference variance [9], for estimating learning rates [25], for network pruning [19], for robust training [41], for probabilistic inference [20], for designing fast solvers [7, 38, 15], and so on.

1.1. Previous work. We classify related work into two groups. One group avoids entrywise evaluation of the GNH matrix and relies on the matrix-vector multiplication (matvec) with the Hessian or the GNH that is matrix-free [28, 32, 30]. For example, the matrix-free matvec can be used to construct low-rank approximations of the GNH matrix through the randomized singular value decomposition (RSVD) [18], but the numerical rank may not be small [44, 11]. Other examples are the following: [10] introduces a low-rank approximation using the Lanczos algorithm to tackle saddle points, [36] maintains a low-rank approximation of the inverse of the Hessian based on rank-one updates at each optimization step, [15] uses a quasi-Newton-like construction of the low-rank approximation, [43, 40] study the convergence of stochastic Newton methods combined with a randomized low-rank approximation, and [41] uses a matrix-free method with only the layers near the output layer.

The other group of methods are based on evaluating or approximating entries on or close to the diagonal of the GNH matrix [24]. For example, [48] introduces a recursive fast algorithm to construct block-diagonal approximations. As another example, [31, 30] introduce the Kronecker-factored approximate curvature (K-FAC), which is based on an entrywise approximation of the Fisher matrix (mathematically equivalent to the GNH for some popular loss functions). The Fisher matrix is given by

$1/n \sum_{i=1}^{n} \mathbb{E}_y[g_i(y)g_i(y)^T]$, where g_i is the gradient evaluated for the ith training point x_i^0 and y is sampled from the network's predictive distribution $\propto \exp(-f(x_i^L, y))$. In practice, an extra step of block-diagonal or block-tridiagonal approximation is used for fast inversion purpose. The method has been tested within optimization frameworks on modern supercomputers and has been shown to perform well [35]. However, the sampling in the K-FAC algorithm converges slowly, and block-diagonal approximations do not account for off-diagonal information.

1.2. Contributions. In this paper, we introduce a fast algorithm for entrywise evaluation of the GNH matrix H, i.e., computing

$$H_{km} = e_k^T \, H \, e_m,$$

where e_k and e_m are two canonical bases for $k, m = 1, 2, \ldots, N$. With the fast evaluation, we propose the hierarchical-matrix ($\mathcal{H}$-matrix) approximation [4, 17] of the GNH matrix for the MLP network, which has applications in autoencoders and long-short memory networks and is often used to study the potential of second-order training methods. Notice if the matrix-free matvec is used to evaluate H_{km}, the computational cost would be $\mathcal{O}(Nn)$.

Our fast algorithm includes a precomputation step and a sampling step, which reduces the cost to $\mathcal{O}(n + d)$ work (independent of N), where d is the output dimension of the network. To illustrate the idea, suppose the network employs the mean squared loss, i.e., $f(x_i^L, y_i) = \frac{1}{2}\|x_i^L - y_i\|^2$, and therefore the GNH matrix is $H = \frac{1}{n} J J^T$, where $J \in \mathbb{R}^{dn \times N}$ is the Jacobian of the network output with respect to the weights. Then $H_{km} = \frac{1}{n}(Je_k)^T (Je_m)$, and only columns in the Jacobian are required to be computed. Our precomputation algorithm exploits the structure of a feed-forward neural network, where the gradient is back propagated layer by layer, so the intermediate results effectively form a compressed format of the Jacobian with $\mathcal{O}(Nn)$ memory. As a result, every column can be retrieved in only $\mathcal{O}(nd)$ time (note every column has $\mathcal{O}(nd)$ entries).

To accelerate the computation of H_{km}, we introduce a fast Monte Carlo sampling algorithm. Let $v_k(i)$ denote the subvector in the Jacobian's kth column corresponding to the ith data point, and therefore $H_{km} = \frac{1}{n}\sum_{i=1}^{n} v_k(i)^T v_m(i)$. In the sampling, we draw c (independent of n) independent samples $t_1, t_1, \ldots, t_c$ from $\{1, 2, \ldots, n\}$ with a carefully designed probability distribution P_{km} and compute an estimator

$$\tilde{H}_{km} = \frac{1}{nc} \sum_{j=1}^{c} \frac{v_k(t_j)^T v_m(t_j)}{P_{km}(t_j)}.$$

We prove $|H_{km} - \tilde{H}_{km}| = \mathcal{O}(1/\sqrt{c})$ with high probability. Note it requires only $\mathcal{O}(n{+}dc)$ work to compute $\tilde{H}_{km}$ as an approximation, where d is the output dimension of the network.

With the fast evaluation algorithm, we are able to take advantage of the existing `GOFMM` method [45, 46, 47] to construct the $\mathcal{H}$-matrix approximation of the GNH matrix through evaluating $\mathcal{O}(N)$ entries in the matrix. The $\mathcal{H}$-matrix approximation is a multilevel scheme that stores diagonal blocks and employs low-rank approximations for off-diagonal blocks in the input matrix. So previous work on the (global) low-rank approximation and the block-diagonal approximation can be viewed as the two extremes in the spectrum of our $\mathcal{H}$-matrix approximation, which effectively works for a broader range of problems. $\mathcal{H}$-matrices are algebraic generalizations of the well-known fast n-body calculation algorithms [3, 16] in computational physics, and they have been applied to kernel methods in machine learning [26, 27]. An $\mathcal{H}$-matrix can

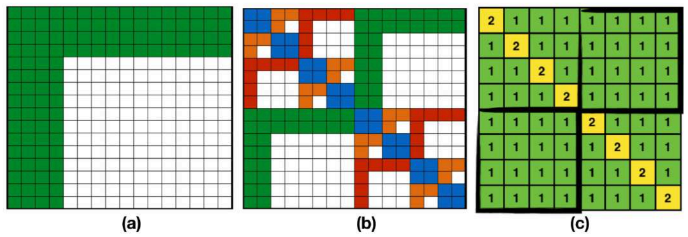

FIG. 1. (a) *A low-rank matrix $H = UV^T$, where U and V are tall-and-skinny matrices;* (b) *a three-level $\mathcal{H}$-matrix, where blue represents dense diagonal blocks and green, red, and orange represent off-diagonal low-rank blocks at levels* 1, 2, *and* 3, *respectively;* (c) *ranks of diagonal and off-diagonal blocks in an $\mathcal{H}$-matrix, where every block has size* 2-*by*-2.

be formulated as

$$H = D + S + UV^T, \tag{1.3}$$

where U and V are tall-and-skinny matrices, S is a block-sparse matrix, and D is a block-diagonal matrix with the blocks being either smaller $\mathcal{H}$-matrices at the next level or dense blocks at the last level. Figure 1 shows the structure of a low-rank matrix and the hierarchically low-rank structure of $\mathcal{H}$-matrices.

Given an $\mathcal{H}$-matrix approximation, the memory footprint is $\mathcal{O}(Nr_o)$,[1] where N is the matrix size or the number of weights in a network and r_o is the maximum off-diagonal rank. Compared to the $\mathcal{O}(N^2)$ storage for the entire matrix, an $\mathcal{H}$-matrix approximation leads to significant memory savings. Once constructed, an $\mathcal{H}$-matrix can be factorized with only $\mathcal{O}(Nr_o^2)$ work, and there exists an entire class of well-established numerical techniques [33, 39, 21, 12, 1, 8, 37]. The factorization can be applied to a vector with $\mathcal{O}(Nr_o)$ work and be used as either a fast direct solver or a preconditioner depending on the approximation accuracy.

To summarize, our work makes the following two major contributions:

- a fast algorithm that requires $\mathcal{O}(Nn)$ storage and requires $\mathcal{O}(n + d/\epsilon^2)$ work to evaluate an arbitrary entry in the GNH matrix, where N and d are the number of parameters and the output dimension of the MLP, respectively; n is the data size; and ϵ is a prescribed accuracy;
- a framework to construct the $\mathcal{H}$-matrix approximation of the GNH matrix, an analysis of the approximation accuracy and the cost, as well as comparison with the RSVD and the K-FAC methods.

Outline. In section 2 we review some background material. In section 3 we present our fast algorithm for evaluating entries in the GNH matrix. In section 4 we show how to construct the $\mathcal{H}$-matrix approximation of the GNH matrix. In section 5 we show numerical results, and in section 6 we conclude with further extensions. Throughout this paper, we use $\|\cdot\|$ to denote the vector/matrix 2-norm and $\|\cdot\|_F$ to denote the matrix Frobenius norm.

2. Background.

In this section, we review the importance of the GNH matrix and the associated computational challenge. The GNH matrix is useful in training and analyzing neural networks, selecting training data, estimating learning rate, and so on. Here we focus on its use in second-order optimization to show the challenge that is common in other applications.

[1]Generally speaking, there may be a $\log(N)$ or $\log^2(N)$ prefactor, as for other complexity results related to $\mathcal{H}$-matrix approximations. But here we focus on the case without such prefactors.

2.1. Neural network training. In an MLP, the weight vector w is obtained via solving the following constrained optimization problem (regularization on w could be added):

$$\min F(w) = \min \frac{1}{n}\sum_{i=1}^{n} f\left(x_i^L, y_i\right) \quad \text{subject to equation (1.1).} \tag{2.1}$$

Recall that f is the loss function and x_i^L is the network output corresponding to input x_i^0, which has label y_i.

To solve for w in problem (2.1), a second-order optimization method solves a sequence of local quadratic approximations of $F(w)$, which requires solving the following linear systems repeatedly:

$$Hp = -g, \tag{2.2}$$

where H is the *curvature matrix* (the Hessian of $F(w)$ in the standard Newton method), $g = \partial_w F$ is the gradient, and p is the update direction. Generally speaking, second-order optimization methods are highly concurrent and could require a much fewer number of iterations to converge than first-order methods, which imply potentially significant speedup on modern distributed computing platforms.

In the Gauss–Newton method, a popular second-order solver, the GNH matrix is employed (with a small regularization) as the curvature matrix in (2.2), which can be solved using the CG method. Since the GNH is mathematically equivalent to the Fisher matrix for several standard choices of the loss, the solution of (2.2) becomes the *natural gradient*, an efficient steepest descent direction in the space of probability distribution with an appropriately defined distance measure [29].

2.2. Back propagation and matrix-free matvec. Table 1 shows the algorithm known as back propagation for evaluating the gradient $g = \partial_w F$ and the matrix-free matvec with the GNH matrix, both of which have complexity $\mathcal{O}(Nn)$. Both algorithms can be derived by introducing Lagrange multipliers z_i^ℓ and $\hat{z}_i^\ell$ for the corresponding weights W_ℓ at every layer [13, 14]. Note that a direct matvec with the full GNH matrix would require $\mathcal{O}(N^2)$ work, not to mention the amount of work to compute the entire matrix.

Based on the two basic ingredients, iterative solvers such as Krylov methods can be used to solve (2.2) as in Hessian-free methods [28, 32]. However, the iteration

TABLE 1

Gradient evaluation and matrix-free matvec with the GNH matrix. Step (a) *of gradient evaluation is the forward pass in* (1.1)*, and steps* (b) *and* (c) *are the well-known back propagation. In the matvec, step* (a) *is known as the linearized forward* ($\hat{x}_i^0 = 0$)*, which computes $J\hat{w}$. Notations: $M_i^\ell = \operatorname{diag}(\dot{s}(W_\ell x_i^{\ell-1}))$, where $\dot{s}$ stands for the derivative; g^ℓ is the gradient for W_ℓ; $\hat{w} = [\texttt{vec}(\hat{W}_1), \dots, \texttt{vec}(\hat{W}_L)]$ is the input of the matvec; and $(H\hat{w})^\ell$ refers to the ℓ_{th} block of the output.*

Evaluate gradient g			Matvec with GNH: $H\hat{w} = J^TQJ\hat{w}$ (Definition 1.1)		
(a) x_i^ℓ	$= s\left(W_\ell x_i^{\ell-1}\right)$	$\forall i, \ell$	(a) $\hat{x}_i^\ell$	$= M_i^\ell\left(W_\ell \hat{x}_i^{\ell-1} + \hat{W}_\ell x_i^{\ell-1}\right)$	$\forall i, \ell$
(b) z_i^L	$= \partial_x f(x_i^L, y_i)$,	$\forall i$	(b) $\hat{z}_i^L$	$= Q_i \hat{x}_i^L$	$\forall i$
(c) $z_i^{\ell-1}$	$= W_\ell^T M_i^\ell z_i^\ell$	$\forall i, \ell$	(c) $\hat{z}_i^{\ell-1}$	$= W_\ell^T M_i^\ell \hat{z}_i^\ell$	$\forall i, \ell$
(d) g^ℓ	$= \sum_{i=1}^n \left(M_i^\ell z_i^\ell\right)(x_i^{\ell-1})^T$	$\forall \ell$	(d) $(H\hat{w})^\ell$	$= \sum_{i=1}^n \left(M_i^\ell \hat{z}_i^\ell\right)(x_i^{\ell-1})^T$	$\forall \ell$

count for convergence can grow rapidly in the presence of ill-conditioning, in which case fast solvers or preconditioners for (2.2) are necessary [2, 23, 30].

3. Fast computation of entries in GNH.

This section presents a precomputation algorithm and a fast Monte Carlo algorithm for fast computation of *arbitrary* entries in the GNH matrix of an MLP network.

A naive method. Consider a GNH matrix $H \in \mathbb{R}^{N\times N}$, where an entry H_{km} can be written as

$$H_{km} = e_k^T \, H \, e_m, \tag{3.1}$$

where e_k and e_m are the kth and the mth columns, respectively, of the N-dimensional identity matrix. We can take advantage of the matrix-free matvec with the GNH matrix in Table 1 to compute $H_{km} = e_k^T(He_m)$, which costs the same as one pass of forward propagation plus one pass of backward propagation, i.e., $\mathcal{O}(Nn) = \mathcal{O}(d^2Ln)$ work.

In the following, we introduce a precomputation algorithm that reduces the cost of evaluating an entry in the GHN to $\mathcal{O}(dn)$ work with $\mathcal{O}(Nn)$ memory and a fast Monte Carlo algorithm that further reduces the cost to $\mathcal{O}(n + d/\epsilon^2)$ work, where ϵ is a prescribed accuracy that does not depend on n or N.

3.1. Precomputation algorithm.

The motivation of our precomputation algorithm is to exploit the sparsity of e_k and e_m plus the symmetry of H in (3.1). Recall the definition of H in (1.2), and let $Q_i = R_i^T R_i$ be a symmetric factorization, which can be computed via, e.g., the eigendecomposition or the LDLT factorization with pivoting. We have

$$\begin{aligned} H_{km} &= e_k^T \big(\textstyle\sum_{i=1}^n J_i^T R_i^T R_i J_i \big) e_m = \sum_{i=1}^n (R_i J_i e_k)^T (R_i J_i e_m) \\ &:= \textstyle\sum_{i=1}^n v_k(i)^T v_m(i), \end{aligned} \tag{3.2}$$

where $v_k(i)$ and $v_m(i)$ are two d-dimensional vectors,

$$v_k(i) = R_i J_i e_k, \quad v_m(i) = R_i J_i e_m, \tag{3.3}$$

for $k, m = 1, 2, \ldots, N$ and $i = 1, 2, \ldots, n$. We state the following theorem and present the precomputation algorithm in the proof.

THEOREM 3.1. *For an MLP network that has L fully connected layers with constant layer size d (d-by-d weight matrices), every entry H_{km} in the GNH matrix can be computed in $\mathcal{O}(dn)$ time with a precomputation that requires $\mathcal{O}(nN)$ storage and $\mathcal{O}(dnN)$ work.*

Proof. We first compute $x_i^{\ell-1}$ and $M_i^\ell = \text{diag}(\dot{s}(W_\ell x_i^{\ell-1}))$ via the forward pass, i.e., step (a) of gradient evaluation in Table 1, and then we precompute and store

$$C_i^\ell = R_i M_i^L W_L M_i^{L-1} W_{L-1} \cdots M_i^\ell, \quad i = 1, 2, \ldots, n; \; \ell = 1, 2, \ldots, L. \tag{3.4}$$

Since every C_i^ℓ is a $d \times d$ matrix, the total storage cost is $\mathcal{O}(d^2\, nL) = \mathcal{O}(nN)$, where $N = d^2L$ is the total number of weights. In addition, notice the relation that $C_i^{\ell-1} = C_i^\ell \left(W_\ell M_i^{\ell-1} \right)$, so they can be computed from $\ell = L$ to $\ell = 1$ iteratively, which requires $\mathcal{O}(d^3Ln) = \mathcal{O}(dnN)$ work in total. Note that the forward pass costs $\mathcal{O}(nN)$ work and that computing the symmetric factorizations for Q_i cost $\mathcal{O}(d^3n)$, which is negligible compared to other parts of the computation.

To complete the proof, we show how to compute $v_k(i)$ as defined in (3.3) with $\mathcal{O}(d)$ work. Below we use the same notations as in Table 1, and e_k is the input vector of the matvec corresponding to $\hat{w} = [\texttt{vec}(\hat{W}_1), \ldots, \texttt{vec}(\hat{W}_L)]$ in Table 1. Recall step

(a) of the matrix-free matvec (linearized forward) with the GNH in Table 1, and we evaluate $J_i e_k$ as follows:

1. Let $\tau = \lceil k/d^2 \rceil$, $\mu = k \bmod d$, and $\nu = \lceil (k \bmod d^2)/d \rceil$. Since e_k has only one nonzero entry, $\hat{x}_i^\ell = 0$ for $\ell = 1, 2, \ldots, \tau - 1$ because $\hat{W}_\ell$ are all zeros except for $\ell = \tau$. The matrix $\hat{W}_\tau$ has only one nonzero at position (μ, ν) (column-major ordering) as the following:

$$\begin{array}{cc} & \nu \\ \mu & \begin{pmatrix} & \vdots & \\ \cdots & 1 & \cdots \\ & \vdots & \end{pmatrix} \end{array} = \hat{W}_\tau.$$

2. Following step (a) of the matvec in Table 1, we have $\hat{x}_i^\tau = M_i^\tau \hat{W}_\tau x_i^{\tau-1}$ at layer τ. Denote $a_i^\tau = \hat{W}_\tau x_i^{\tau-1}$, and we have

$$\begin{aligned} \hat{x}_i^\tau &= M_i^\tau a_i^\tau \\ \hat{x}_i^{\tau+1} &= M_i^{\tau+1} W_{\tau+1} \hat{x}_i^\tau && \text{(since } \hat{W}_{\tau+1} = 0\text{)} \\ &= M_i^{\tau+1} W_{\tau+1} M_i^\tau a_i^\tau \\ &\cdots \\ \hat{x}_i^L &= M_i^L W_L M_i^{L-1} W_{L-1} \cdots M_i^\tau a_i^\tau. \end{aligned}$$

3. Notice that the only nonzero entry in a_i^τ is the μth element, which equals the νth element in $x_i^{\tau-1}$. Therefore,

$$v_k(i) = R_i J_i e_k = R_i \hat{x}_i^L = C_i^\tau a_i^\tau, \tag{3.5}$$

where $C_i^\tau a_i^\tau$ should be interpreted as a scaling of the μth column in C_i^τ by the νth element in $x_i^{\tau-1}$, which costs $\mathcal{O}(d)$ work. $\square$

3.2. Fast Monte Carlo algorithm. Recall (3.2), which sums over a large number of data points, and the idea is to sample a subset with a judiciously chosen probability distribution and scale the (partial) sum appropriately to approximate H_{km}. It is important to note that the computation of the probabilities is fast based on the previous precomputation. The fast sampling algorithm is given in Algorithm 3.1.

Algorithm 3.1 Fast Monte Carlo algorithm.

```
1: Input: $\|v_k(i)\|$ and $\|v_m(i)\|$ for $i = 1, 2, \ldots, n$.
2: Compute sampling probabilities for $t = 1, 2, \ldots, n$:
       (3.6)   $P_{km}(t) = \frac{\|v_k(t)\| \, \|v_m(t)\|}{\sum_{j=1}^n \|v_k(j)\| \, \|v_m(j)\|}$.
3: Draw $c$ independent random samples $t_j$ from $\{1, 2, \ldots, n\}$ with replacement.
4: Output:
       (3.7)   $\tilde{H}_{km} = \frac{1}{c} \sum_{j=1}^{c} \frac{v_k(t_j)^T v_m(t_j)}{P_{km}(t_j)}$.
```

Define $v_k = [v_k(1), \dots, v_k(n)]$ and $v_m = [v_m(1), \dots, v_m(n)]$ as two vectors in $\mathbb{R}^{dn}$, and (3.2) can be written as the inner product of the two vectors:

$$H_{km} = v_k^T v_m.$$

The following theorem shows that our sampling algorithm returns a good estimator of H_{km}, where the error is measured using $\|v_k\|\|v_m\|$, an upper bound on $|H_{km}|$.

THEOREM 3.2 (sampling error). *Consider an MLP network that has L fully connected layers with constant layer size d (d-by-d weight matrices). For every entry H_{km} in the GNH matrix, Algorithm* 3.1 *returns an estimator $\tilde{H}_{km}$*

- *that is an unbiased estimator of H_{km}, i.e., $\mathbb{E}[\tilde{H}_{km}] = H_{km}$;*
- *whose variance or mean squared error (MSE) satisfies*

$$\mathrm{Var}[\tilde{H}_{km}] = \mathbb{E}\left[|H_{km} - \tilde{H}_{km}|^2\right] \le \frac{1}{c}\|v_k\|^2\|v_m\|^2, \tag{3.8}$$

  where c is the number of random samples;
- *whose absolute error satisfies*

$$|H_{km} - \tilde{H}_{km}| \le \frac{\eta}{\sqrt{c}}\ \|v_k\|\|v_m\|, \tag{3.9}$$

  with probability at least $1-\delta$, where $\delta \in (0,1)$, $\eta = 1 + \dots$ and c is the number of random samples.

Proof. Our proof consists of the following three parts.

Unbiased estimator. Define a random variable

$$X_t = \frac{v_k(t)^T v_m(t)}{P(t)},$$

where t is a random sample from $\{1, 2, \dots, n\}$ with probability distribution $P(t)$ as defined in (3.6). Observe that $\tilde{H}_{km}$ is the mean of c independent identically distributed variables $(X_{t_1}, X_{t_2}, \dots, X_{t_c})$, and thus

$$\mathbb{E}[\tilde{H}_{km}] = \mathbb{E}[X_t] = \sum_{t=1}^{n} \frac{v_k(t)^T v_m(t)}{P(t)} P(t) = H_{km}.$$

Variance/MSE error. The variance or MSE error of the estimator is the following:

$$\begin{aligned}
&\mathbb{E}\left[|H_{km} - \tilde{H}_{km}|^2\right] = \mathrm{Var}[\tilde{H}_{km}] = \frac{1}{c}\mathrm{Var}[X_t] \\
&= \frac{1}{c}\left(\mathbb{E}[X_t^2] - \mathbb{E}^2[X_t]\right) \\
&= \frac{1}{c}\sum_{t=1}^{n}\left(\frac{v_k(t)^T v_m(t)}{P(t)}\right)^2 P(t) - \frac{H_{km}^2}{c} \\
&\le \sum_{t=1}^{n} \frac{\left(v_k(t)^T v_m(t)\right)^2}{c\,P(t)} && \text{(drop the last term)} \\
&\le \sum_{t=1}^{n} \frac{\|v_k(t)\|^2\|v_m(t)\|^2}{c\,P(t)} && \text{(Cauchy–Schwarz)}
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{c}\left(\sum_{t=1}^{n}\|v_k(t)\|\|v_m(t)\|\right)^2 && \text{(equation (3.6))}\\
&\le \frac{1}{c}\left(\sum_{t=1}^{n}\|v_k(t)\|^2\right)\left(\sum_{t=1}^{n}\|v_m(t)\|^2\right) && \text{(Cauchy–Schwarz)}\\
&= \frac{1}{c}\|v_k\|^2\|v_m\|^2.
\end{aligned}
$$

Notice that with Jensen's inequality, we also obtain a bound of the absolute error in expectation:

$$
\mathbb{E}\left[|H_{km} - \tilde{H}_{km}|\right] \le \frac{1}{\sqrt{c}}\|v_k\|\|v_m\|. \tag{3.10}
$$

Concentration result. We will use McDiarmid's (or the Hoeffding-Azuma or bounded differences) inequality to obtain (3.9). See the conditions for the inequality in Appendix A. Define function

$$
F(t_1, t_2, \ldots, t_c) = |H_{km} - \tilde{H}_{km}|,
$$

where $t_1, t_2, \ldots, t_c$ are random samples, and we show that changing one sample t_i at a time does not affect F too much. Consider changing a sample t_i to t_i' while keeping others the same. The new estimator $\hat{H}_{km}$ differs from $\tilde{H}_{km}$ by only one term. Thus,

$$
\begin{aligned}
|\tilde{H}_{km} - \hat{H}_{km}| &= \left|\frac{v_k(t_i)^T v_m(t_i)}{c\ P(t_i)} - \frac{v_k(t_i')^T v_m(t_i')}{c\ P(t_i')}\right|\\
&\le \left|\frac{v_k(t_i)^T v_m(t_i)}{c\ P(t_i)}\right| + \left|\frac{v_k(t_i')^T v_m(t_i')}{c\ P(t_i')}\right|\\
&\le \frac{\|v_k(t_i)\|\|v_m(t_i)\|}{c\ P(t_i)} + \frac{\|v_k(t_i')\|\|v_m(t_i')\|}{c\ P(t_i')}\\
&= \frac{2}{c}\sum_{j=1}^{n}\|v_k(j)\|\,\|v_m(j)\|\\
&\le \frac{2}{c}\|v_k\|\|v_m\|,
\end{aligned}
$$

where we have used Cauchy–Schwarz inequality twice. Then define $\Delta = \frac{2}{c}\|v_k\|\|v_m\|$; using the triangle inequality, we see

$$
|F(\ldots, t_i, \ldots) - F(\ldots, t_i', \ldots)| \le \Delta.
$$

Finally, let $\gamma = \sqrt{2c\log(1/\delta)}\,\Delta$, and we use McDiarmid's inequality to obtain (3.9) as follows:

$$
\begin{aligned}
&\Pr\left[|H_{km} - \tilde{H}_{km}| \ge \frac{\eta}{\sqrt{c}}\ \|v_k\|\|v_m\|\right]\\
&= \Pr\left[|H_{km} - \tilde{H}_{km}| \ge \frac{1}{\sqrt{c}}\|v_k\|\|v_m\| + \gamma\right]\\
&\le \Pr\left[F - \mathbb{E}[F] \ge \gamma\right] && \text{(equation (3.10))}\\
&\le \exp\left(-\frac{\gamma^2}{2c\Delta^2}\right) = \delta && \text{(McDiarmid's inequality).}
\end{aligned}
$$

□

Remark 3.3. The error ϵ in the approximation of H_{km} depends on only the number of random samples c (but not n) and can be made arbitrarily small as needed. In particular, if $c \geq 1/\epsilon^2$, we have

$$\mathrm{Var}[\tilde{H}_{km}] = \mathbb{E}\left[|H_{km} - \tilde{H}_{km}|^2\right] \leq \epsilon\ \|v_k\|^2\|v_m\|^2,$$

and if $c \geq \eta^2/\epsilon^2$, then with probability at least $1-\delta$, where $\delta \in (0,1)$,

$$|H_{km} - \tilde{H}_{km}| \leq \epsilon\ \|v_k\|\|v_m\|.$$

Furthermore, the error of the entire matrix in the Frobenius norm is

$$\|H - \tilde{H}\|_F \leq \epsilon\sqrt{\sum_k \sum_m \|v_k\|^2\|v_m\|^2} = \epsilon \sum_k \|v_k\|^2$$
$$\overset{(3.2)}{=} \epsilon \sum_k H_{kk} = \epsilon\, \mathrm{trace}(H) \leq \epsilon\ \sqrt{N}\ \|H\|_F.$$

Remark 3.4. The estimator $\tilde{H}_{km}$ is exact using at most *one* sample when $k = m$. The (trivial) case $H_{kk} = 0$ is implied by the situation that $v_{kk}(i) = 0$ for all i; otherwise, we have $H_{kk} = \|v_k\|^2$, and the sampling probability becomes

$$P_{kk}(t) = \frac{\|v_k(t)\|^2}{\sum_{j=1}^n \|v_k(j)\|^2} = \frac{\|v_k(t)\|^2}{\|v_k\|^2}.$$

Therefore, $\tilde{H}_{kk} = \|v_k(t)\|^2/P_{kk}(t) = H_{kk}$ with any random sample t.

THEOREM 3.5 (computational cost of sampling). *Given the precomputation in Theorem* 3.1*, it requires* $\mathcal{O}(nN)$ *work to compute* $\|v_k(i)\|$ *for all* i *and* k *as the input of Algorithm* 3.1*, and it requires* $\mathcal{O}(n + d/\epsilon^2)$ *work to compute every estimator, where* ϵ *is a prescribed accuracy that does* not *depend on* n.

Proof. Recall from (3.5) that $\|v_k(i)\|$ is proportional to the norm of a column in C_i^ℓ. Since every C_i^ℓ is a d-by-d matrix, computing all the norms requires $\mathcal{O}(d^2nL) = \mathcal{O}(nN)$ work. Once all $\|v_k(i)\|$ have been computed, the sampling probabilities in (3.6) and the estimator in (3.7) require $\mathcal{O}(n)$ and $\mathcal{O}(d/\epsilon^2)$ work, respectively. ☐

4. $\mathcal{H}$-matrix approximation.

This section introduces the $\mathcal{H}$-matrix approximation of the GNH matrix for the MLP. While the low-rank and the block-diagonal approximations focus on the global and the local structure of the problem, respectively, the $\mathcal{H}$-matrix approximation handles both, as they may be equally important.

4.1. Overall algorithm.

Here we take advantage of the `GOFMM` method [45, 46, 47], which evaluates $\mathcal{O}(N)$ entries in a symmetric positive definite (SPD) matrix $H \in \mathbb{R}^{N\times N}$ to construct the $\mathcal{H}$-matrix approximation $H_{\texttt{GOFMM}}$ such that

$$\|H - H_{\texttt{GOFMM}}\|_F \leq \epsilon\, \|H\|_F,$$

where ϵ is a prescribed tolerance.

Since `GOFMM` requires only entrywise evaluation of the input matrix, we apply it with our fast evaluation algorithm to the regularized GNH matrix (note the GNH matrix is symmetric positive semidefinite, so we always add a small regularization of λ times the identity matrix, where λ^2 is the unit roundoff). The overall algorithm

Algorithm 4.1 Compute $\mathcal{H}$-matrix approximation of GNH with GOFMM.

```
Require: training data $\{x_i^0\}_{i=1}^n$, weights in the neural network $w \in \mathbb{R}^N$
Ensure: approximation of the GNH and its factorization
1: Compute $M_i^\ell$ with forward propagation. (step (a) of gradient evaluation in Table 1)
2: Compute $C_i^\ell$ in (3.4). (Theorem 3.1: $\mathcal{O}(Nnd)$ work and $\mathcal{O}(Nn)$ storage)
3: Compute $\|v_k(i)\|$ in (3.5). (Theorem 3.5: $\mathcal{O}(Nn)$ work and $\mathcal{O}(Nn)$ storage)
4: Apply GOFMM and evaluate entries in the GNH matrix through Algorithm 3.1. (Theorem 3.5: $\mathcal{O}(n + d/\epsilon^2)$ work/entry)
```

that computes the $\mathcal{H}$-matrix approximation (and approximate factorization) of the GNH matrix using the GOFMM method is shown in Algorithm 4.1.

The error analysis of Algorithm 4.1 is the following. Let $\tilde{H}_\lambda = \tilde{H} + \lambda I$ be computed by Algorithm 3.1, $\lambda > 0$ be a regularization, and $\tilde{H}_{\texttt{GOFMM}}$ be the approximation of $\tilde{H}_\lambda$ computed by GOFMM. Then the error between the output $\tilde{H}_{\texttt{GOFMM}}$ from Algorithm 4.1 and the (regularized) GNH matrix $H_\lambda = H + \lambda I$ is the following (using the triangular equality):

$$\|H_\lambda - \tilde{H}_{\texttt{GOFMM}}\|_F = \|H_\lambda - \tilde{H}_\lambda + \tilde{H}_\lambda - \tilde{H}_{\texttt{GOFMM}}\|_F \leq \|H - \tilde{H}\|_F + \|\tilde{H}_\lambda - \tilde{H}_{\texttt{GOFMM}}\|_F,$$

where the first term is the sampling error from Algorithm 3.1 and the second term is the GOFMM approximation error. For simplicity, we drop the regularization parameter for the rest of this paper.

4.2. GOFMM overview. Given an SPD matrix H, the GOFMM takes two steps to construct the $\mathcal{H}$-matrix approximation as follows. First of all, a permutation matrix P is computed to reorder the original matrix, which often corresponds to a hierarchical domain decomposition for applications in two- or three-dimensional physical spaces. The recursive domain partitioning is often associated with a tree data structure $\mathcal{T}$. Unlike methods targeting applications in physical spaces, the GOFMM does not require the use of geometric information (thus its name "geometry-oblivious fast multipole method"), which does not exist for neural networks. Instead of relying on geometric information, the GOFMM exploits the algebraic distance measure that is implicitly defined by the input matrix H. As a matter of fact, any SPD matrix $H \in \mathbb{R}^{N\times N}$ is the *Gram matrix* of N *unknown Gram vectors* $\{\phi_i\}_{i=1}^N$ [22]. Therefore, the distance between two row/column indices i and j can be defined as

$$d_{ij} = \sin^2(\angle(\phi_i, \phi_j)) = 1 - H_{ij}^2/(H_{ii}H_{jj}) \tag{4.1}$$

or

$$d_{ij} = \|\phi_i - \phi_j\| = \sqrt{H_{ii} - 2H_{ij} + H_{jj}}.$$

We refer interested readers to [45] for the discussion and comparison of different distance metrics. With either definition, the GOFMM is able to construct the permutation P and a balanced binary tree $\mathcal{T}$.

The second step is to approximate the reordered matrix $P^T HP$ by

$$H_{\texttt{GOFMM}} = \begin{bmatrix} H_{\texttt{ll}} & 0 \\ 0 & H_{\texttt{rr}} \end{bmatrix} + \begin{bmatrix} 0 & S_{\texttt{lr}} \\ S_{\texttt{rl}} & 0 \end{bmatrix} + \begin{bmatrix} 0 & U_{\texttt{lr}}V_{\texttt{lr}}^T \\ U_{\texttt{rl}}V_{\texttt{rl}}^T & 0 \end{bmatrix},$$

where $H_{\mathtt{ll}}$ and $H_{\mathtt{rr}}$ are two diagonal blocks that have the same structure as $H_{\mathtt{GOFMM}}$ unless their sizes are small enough to be treated as dense blocks, which occurs at the leaf level of the tree $\mathcal{T}$; $S_{\mathtt{lr}}$ and $S_{\mathtt{rl}}$ are block-sparse matrices; and $U_{\mathtt{lr}}V_{\mathtt{lr}}^T$ and $U_{\mathtt{rl}}V_{\mathtt{rl}}^T$ are low-rank approximations of the remaining off-diagonal blocks in H. These bases are computed recursively with a postorder traversal of $\mathcal{T}$ using the interpolative decomposition [18] and a nearest neighbor–based fast sampling scheme. There is a trade-off here: While the so-called weak-admissibility criterion sets $S_{\mathtt{lr}}$ and $S_{\mathtt{rl}}$ to zero and obtains relatively large ranks, the so-called strong-admissibility criterion selects $S_{\mathtt{lr}}$ and $S_{\mathtt{rl}}$ to be certain subblocks in H corresponding to a few nearest neighbors/indices of every leaf node in $\mathcal{T}$ and achieves smaller (usually constant) ranks.

Here we focus on the *hierarchical semiseparable (HSS)* format among other types of hierarchical matrices. Technically speaking, the HSS format means $S_{\mathtt{lr}}$ and $S_{\mathtt{rl}}$ are both zero and the bases $U_{\mathtt{lr}}/V_{\mathtt{lr}}$ and $U_{\mathtt{rl}}/V_{\mathtt{rl}}$ of a node in $\mathcal{T}$ are recursively defined through the bases of the node's children, i.e., the so-called *nested bases*.

We refer interested readers to [45, 46, 47] for details about the GOFMM method.

4.3. Summary and contrast with related work. We summarize the storage and computational complexity of our $\mathcal{H}$-matrix approximation method (HM) and describe its relation with three existing methods, namely, the Hessian-free method (HF) [28, 32], the RSVD [18], and the K-FAC [30, 31]. As before, we assume the MLP network has L layers of constant layer sizes d, so the number of weights is $N = d^2L$. Let n be the number of data points.

HM. The algorithm is given in Algorithm 4.1, where the first three steps require $\mathcal{O}(Nnd)$ work and $\mathcal{O}(Nn)$ storage. Suppose the rank is r_o in the $\mathcal{H}$-matrix approximation. The GOFMM needs to call Algorithm 3.1 $\mathcal{O}(Nr_o)$ times, which results in $\mathcal{O}((n + d/\epsilon^2)Nr_o)$ work. Here, ϵ is chosen to be around the same accuracy as the $\mathcal{H}$-matrix approximation with rank r_o. In addition, standard results in the HSS literature [33, 39, 21] state that the factorization requires $\mathcal{O}(Nr_o^2)$ work and $\mathcal{O}(Nr_o)$ storage, which can be applied to solving a linear system with $\mathcal{O}(Nr_o)$ work.

MF. Unlike the other three methods, the MF does not approximate the GNH. It takes advantage of the (exact) matrix-free matvec and utilizes the CG method for solving linear systems. It is based on the two primitives in Table 1, where every iteration costs $\mathcal{O}(Nn)$ work and storage. The number of CG iteration is generally upper bounded by $\mathcal{O}(\sqrt{\kappa})$, where κ is the condition number of the (regularized) GNH matrix.

RSVD. Recall the GNH matrix $H = J^TQJ$. Without loss of generality, assume Q is an identity for ease of description. The algorithm is to compute an approximate SVD of J with the following steps, which naturally leads to an approximate eigenvalue decomposition of H. First, we apply the back propagation in Table 1 with a random Gaussian matrix as input. Second, the QR decomposition of the result is used to estimate the row space of J. Third, the linearized forward is applied to project J onto the approximate row space, and, finally, the SVD is computed on the projection. Overall, the storage is $\mathcal{O}(Nr)$, and the work required is $\mathcal{O}(Nnr + Nr^2 + dnr^2)$, where r is the numerical rank from the QR decomposition. Compared with the HM approximating off-diagonal blocks, the RSVD approximates the entire matrix.

K-FAC. It computes an approximation of the Fisher matrix F (mathematically equivalent to the GNH for some popular loss functions). Let a column vector $g = [\mathtt{vec}(g^1), \ldots, \mathtt{vec}(g^L)] \in \mathbb{R}^N$ be the gradient and $F = \mathbb{E}[g\,g^T]$ be a L-by-L block matrix with block size d^2-by-d^2. Note the expectation here is taken with respect to both the empirical input data distribution $\hat{Q}_{x^0}$ and the network's predictive

TABLE 2

Asymptotic complexities of the MF, the RSVD, the K-FAC, and the HM with respect to the number of weights N and the data size n ("lower-order" terms not involving Nn are dropped). We assume $r, k, r_o, d < n$, where r, k, and r_o are the parameters in the RSVD, the K-FAC, and the HM, respectively, and d is the (constant/average) layer size. In addition, κ stands for the condition number of the GNH matrix.

	MF	**RSVD**	**K-FAC**	**HM**
construction	-	$\mathcal{O}(Nnr)$	$\mathcal{O}(Nnk)$	$\mathcal{O}(Nn(r_o+d))$
storage	$\mathcal{O}(Nn)$	$\mathcal{O}(Nr)$	$\mathcal{O}(N)$	$\mathcal{O}(Nn)$
solve	$\mathcal{O}(Nn\sqrt{\kappa})$	$\mathcal{O}(Nr)$	$\mathcal{O}(Nd)$	$\mathcal{O}(Nr_o)$

distribution $P_{y|x^\ell}$. In particular, the (ℓ_1, ℓ_2)th block $(\ell_1, \ell_2 = 1, 2, \ldots, L)$ is given by

$$F_{\text{block}}(\ell_1,\ell_2) = \mathbb{E}[\text{vec}(g^{\ell_1})\text{vec}(g^{\ell_2})^T]$$

$$= \mathbb{E}[M^{\ell_1} z^{\ell_1}(x^{\ell_1-1})^T \left(M^{\ell_2} z^{\ell_2}(x^{\ell_2-1})^T\right)^T] \tag{4.2}$$

$$= \mathbb{E}[(M^{\ell_1} z^{\ell_1} \otimes x^{\ell_1-1}) \left(M^{\ell_2} z^{\ell_2} \otimes x^{\ell_2-1}\right)^T] \tag{4.3}$$

$$= \mathbb{E}[(M^{\ell_1} z^{\ell_1} \otimes x^{\ell_1-1}) \left((M^{\ell_2} z^{\ell_2})^T \otimes (x^{\ell_2-1})^T\right)] \tag{4.4}$$

$$= \mathbb{E}[M^{\ell_1} z^{\ell_1}(M^{\ell_2} z^{\ell_2})^T \otimes x^{\ell_1-1}(x^{\ell_2-1})^T] \tag{4.5}$$

$$\approx \mathbb{E}[M^{\ell_1} z^{\ell_1}(M^{\ell_2} z^{\ell_2})^T] \otimes \mathbb{E}[x^{\ell_1-1}(x^{\ell_2-1})^T], \tag{4.6}$$

where (4.2) uses the definition of the network gradient in Table 1, (4.3) rewrites the equation using Kronecker products, (4.4) and (4.5) use the properties of the Kronecker product, and (4.6) assumes the statistical independence between the two terms (see section 6.3.1 in [30]). In (4.6), the former expectation is taken with respect to both $\hat{Q}_{x^0}$ and $P_{y|x^\ell}$, and the latter is taken with respect to $\hat{Q}_{x^0}$. To compute the first expectation, k samples are drawn from the distribution $P_{y|x} \propto \exp(-f(x_i^L, y))$, where x_i^L is the network's output corresponding to input x_i^0. In practice, an additional block-diagonal or block-tridiagonal approximation of the inverse is employed for fast solution of linear systems. The main cost of the algorithm is constructing, updating, and inverting $\mathcal{O}(L)$ matrices of size d-by-d, which requires $\mathcal{O}(d^2L) = \mathcal{O}(N)$ storage and $\mathcal{O}((nk+d)N)$ work. Overall, the approximation error of K-FAC has three components: the error of making the assumption (4.6), the sampling error from approximating the expectations in (4.6) and the error of block-diagonal or block-tridiagonal approximation of the inverse.

We summarize the asymptotic complexities of the four methods discussed above in Table 2.

5. Experimental results.

In this section, we show (1) the cost and the accuracy of our $\mathcal{H}$-matrix approximations, (2) the memory savings from using the precomputation algorithm ($\mathcal{O}(N^2) \to \mathcal{O}(Nn)$), and (3) the efficiency of the fast sampling algorithm. In Algorithm 4.1, the first two steps (precomputation) are implemented in MATLAB for the convenience of extracting intermediate values of neural networks, and the last two steps (sampling) are implemented in C++ (`GOFMM` is written in C++).

Networks and datasets. We focus on classification networks and autoencoder networks with the MNIST and CIFAR-10 datasets. In the following, we denote networks' layer sizes as $d_1 \to d_2 \to \cdots \to d_L$ from the input layer to the output layer. Every network has been trained using the stochastic gradient descent for a few steps, so the weights are not random:

1. "classifier": classification networks with the ReLU activation and the cross-entropy loss:
   (a) N=15,910; MNIST dataset; layer sizes: 784→20→10;
   (b) N=61,670; CIFAR-10 dataset; layer sizes: 3072→20→10;
   (c) N=219,818; MNIST dataset; layer sizes: 784→256→64→32→10;
   (d) N=1,643,498; CIFAR-10 dataset; layer sizes: 3072→512→128→32→10.
2. "AE": autoencoder networks with the softplus activation (sigmoid activation at the last layer) and the mean-squared loss:
   (a) N=16,474; MNIST dataset; layer sizes: 784→10→784;
   (b) N=64,522; CIFAR-10 dataset; layer sizes: 3072→10→3072;
   (c) N=125,972; CIFAR-10 dataset; layer sizes: 3072→20→3072.

`GOFMM` parameters. We employ the default "angle" distance metric in (4.1) and focus on three parameters in the `GOFMM` that control the accuracy of the $\mathcal{H}$-matrix approximation: (1) the leaf node size m of the hierarchical partitioning $\mathcal{T}$ (equivalent to setting the number of tree levels), (2) the maximum rank r_o of off-diagonal blocks, and (3) the accuracy τ of low-rank approximations. In particular, we ran `GOFMM` with two different accuracies: "`low`" ($m = 128$, $r_o = 128$, $\tau = 5E$-2) and "`high`" ($m = 1024$, $r_o = 1024$, $\tau = 1E$-5).

`GOFMM` results. We report the following results for our approach:

- t_{build}: time of constructing the $\mathcal{H}$-matrix approximation of the GNH matrix (not including precomputation time);
- t_{matv}: time of applying the $\mathcal{H}$-matrix approximation to 128 random vectors;
- %K: compression rate of the $\mathcal{H}$-matrix approximation, i.e., ratio between the $\mathcal{H}$-matrix storage and the GNH matrix storage;
- ϵ_F: relative error of the $\mathcal{H}$-matrix approximation measured in Frobenius norm, estimated by $\|Hx - H_{\texttt{GOFMM}}\, x\|_F / \|Hx\|_F$, where $x \in \mathbb{R}^{N\times 128}$ is a Gaussian random matrix.

5.1. Cost and accuracy of $\mathcal{H}$-matrix approximation. Table 3 shows results of our $\mathcal{H}$-matrix approximations for networks that have relatively small numbers of parameters. The GNH matrices are computed and fully stored in memory.

As Table 3 shows, the approximation can achieve four digits' accuracy except for one network (two digits) when the accuracy of low-rank approximations is 1E-5. Since we have enforced the maximum rank r_o, the runtime of constructing $\mathcal{H}$-matrix approximations (t_{build}) increases proportionally to the number of network parameters, and the compression rate scales inverse proportionally to the number of parameters. The reported construction time t_{build} includes the cost of creating an implicit tree data structure $\mathcal{T}$ in `GOFMM`, which is less than 20% of t_{build}. In addition, applying the $\mathcal{H}$-matrix approximations to 128 random vectors took less than one second for the five networks. These $\mathcal{H}$-matrix approximations can be factorized in linear time for solving linear systems and eigenvalue problems.

Comparison with RSVD and K-FAC. We implemented the RSVD using Keras and TensorFlow for fast back propogation, and we implemented the K-FAC in MATLAB for the convenience of extracting intermediate values. Table 4 shows the accuracies of our method (HM), the RSVD and the K-FAC under about the same compression rate for the low- and high-accuracy settings, respectively. For the RSVD, the storage is rN entries, where r is the numerical rank of the (symmetric) GNH matrix, so the compression rate is r/N. For the K-FAC, we use the relatively more accurate block tridiagonal version. The compression rate of the K-FAC is defined as k/N, and the reason is that the construction of the RSVD and the K-FAC requires the same

TABLE 3

Timing (t_{build} and t_{matv}), compression rate (%K), and accuracy (ϵ_F) of $\mathcal{H}$-matrix approximations corresponding to low- and high-accuracy settings, respectively. t_{build} is the time of applying the `GOFMM` on the GNH matrices corresponding to 1,000 data points. Experiments performed on one node from the Texas Advanced Computing Center "Stampede 2" system, which has two sockets with 48 cores of Intel Xeon Platinum 8160/"Skylake" and 192 GB of RAM.

#	network	N	`accuracy`	t_{build}	t_{matv}	%K	ϵ_F
1	classifier (a)	16k	`low`	0.24	0.03	1.80%	1.5E-1
2			`high`	5.74	0.11	13.59%	4.4E-4
3	classifier (b)	61k	`low`	0.42	0.08	0.57%	4.7E-1
4			`high`	13.28	0.33	4.78%	4.0E-2
5	AE (a)	16k	`low`	0.27	0.03	1.25%	1.2E-1
6			`high`	5.67	0.10	11.38%	6.5E-4
7	AE (b)	64k	`low`	0.43	0.08	0.53%	5.5E-3
8			`high`	13.26	0.38	4.62%	6.6E-4
9	AE (c)	126k	`low`	0.87	0.17	0.28%	4.1E-3
10			`high`	24.10	0.94	2.32%	5.2E-4

TABLE 4

Comparison of accuracies (ϵ_F) among the $\mathcal{H}$-matrix approximation (HM), the RSVD and the K-FAC with about the same compression rate (%K) for the low- and high-accuracy settings, respectively. For the RSVD and the K-FAC, the compression rate means r/N and k/N, respectively, where r is the rank and k is the number of random samples. For all cases, the GNH matrices correspond to 10,000 data points ($n = 10,000$ in (1.2)).

		HM-low	HM-high	RSVD-low	RSVD-high	K-FAC-low	K-FAC-high
AE(a)	%K	1.23%	11.77%	1.40%	12.14%	1.40%	12.14%
	ϵ_F	1.7E-1	4.7E-4	4.3E-1	5.1E-3	1.2E-1	7.3E-2
AE(b)	%K	0.53%	4.62%	0.62%	4.65%	0.62%	4.65%
	ϵ_F	5.7E-3	6.4E-4	8.4E-1	2.3E-1	1.7E-1	3.8E-2
AE(c)	%K	0.28%	2.31%	0.32%	2.38%	0.32%	2.38%
	ϵ_F	4.2E-3	4.9E-4	9.1E-1	2.1E-1	1.6E-1	4.1E-2

number of back propagation if $k = r$ (recall Table 2). So we choose k and r to be the same value such that the corresponding compression rate of the RSVD and the K-FAC are slightly higher than the HM.

As Table 4 shows, the $\mathcal{H}$-matrix approximation achieved higher accuracy than the RSVD and the K-FAC for most cases, especially for the high-accuracy setting. For the RSVD, suppose the eigenvalues of the GHN matrix are $\{\sigma_i\}_i$ and the error of the rank-k approximation measured in the Frobenius norm is proportional to $(\sum_{i>k}\sigma_i^2)^{1/2}$. For autoencoders (b) and (c), the spectrums of the GNH matrices decay slowly, so the RSVD is not efficient. For the K-FAC, the approximation that the expectation of a Kronecker product equals the Kronecker product of expectations (4.6) is, in general, not exact, impeding the overall accuracy of the method.

5.2. Memory savings. Table 5 shows the memory footprint between our precomputation (3.4) and the full GNH matrix, i.e., $\mathcal{O}(N^2)$. Recall from Theorem 3.1 that the storage of our precomputation is $\mathcal{O}(nN)$, where n is the number of data points.

Table 5

Comparison of memory footprint (in single precision) between our precomputation (3.4) *and the full GNH matrix. For each network, we show the compression rate and accuracy of $\mathcal{H}$-matrix approximations for two levels of accuracies. For both cases, the GNH matrices correspond to $n = 10,000$ data points.*

	N	M_{ours}	M_{GNH}	`accuracy`	%K	ϵ_F
classifier (c)	219k	191 MB	193 GB	`low`	0.165%	3.1E-1
				`high`	1.268%	4.2E-2
classifier (d)	1.6m	423 MB	10.8 TB	`low`	0.012%	1.2E-1
				`high`	0.177%	2.2E-2

Table 6

Accuracy of our fast Monte Carlo (FMC) sampling scheme. The error $\epsilon_K = \|H - \tilde{H}\|_F/\|H\|_F$, where H and $\tilde{H}$ are the exact GNH matrix and its approximation computed using K random samples in Algorithm 3.1*, respectively. The exact GNH matrices correspond to the AE* (a) *network with the entire MNIST dataset and the class* (a) *network with the entire CIFAR-*10 *training dataset, respectively. The reference uniform sampling scheme uses a uniform sampling probability instead of* (3.6) *in Algorithm* 3.1.

	n	`scheme`	ϵ_{10}	ϵ_{100}	$\epsilon_{1,000}$	$\epsilon_{10,000}$
MNIST	60,000	uniform	3.6E-1	1.1E-1	3.6E-1	1.1E-2
		FMC	9.7E-3	3.1E-3	9.6E-4	3.1E-4
CIFAR-10	50,000	uniform	9.7E-1	3.1E-1	9.8E-2	3.6E-2
		FMC	6.1E-1	1.9E-1	6.1E-2	1.9E-2

As Table 5 shows, our precomputation leads to huge memory reduction compared with storing the full GNH matrix. This allows using the `GOFMM` method for networks that have a large number of parameters. For example, the storage of the GNH matrix for the classifier (d) network requires more than 10 TB! But we were able to run `GOFMM` with the compressed storage (at the price of spending $\mathcal{O}(dn)$ work for the evaluation of every entry). The precomputation of (3.4) took merely about 2 seconds and 7 seconds, respectively.

5.3. Fast Monte Carlo sampling. We show the accuracy of our fast Monte Carlo sampling scheme. The relative error measured in the Frobenius norm is between the exact GNH matrix H and the approximation $\tilde{H}$ computed using Algorithm 3.1 with a prescribed number of random samples. For reference, we also run the same sampling scheme but with a uniform probability distribution.

As Table 6 shows, when the number of random samples increases by 100, the accuracy improves by 10, which confirms the standard convergence rate of Monte Carlo in Theorem 3.2. Importantly, the error bound and the convergence rate do *not* depend on the problem size n. Moreover, our sampling scheme outperforms the uniform sampling by at most two orders of magnitude for the MNIST dataset. In other words, the uniform sampling requires 100 times more random samples to achieve about the same accuracy as our sampling scheme.

$\mathcal{H}$-matrix approximation with sampling. Table 7 shows the error of Algorithm 4.1 for a sequence of increasingly large number of random samples. Recall that Algorithm 4.1 computes the $\mathcal{H}$-matrix approximation $\tilde{H}_{\texttt{GOFMM}}$ for the (inexact) GNH matrix, namely, $\tilde{H}$, from Algorithm 3.1. The error between the $\mathcal{H}$-matrix approximation $\tilde{H}_{\texttt{GOFMM}}$ and the exact GNH matrix, namely, H, is bounded as below (using the trian-

TABLE 7
$\mathcal{H}$-matrix approximation with sampling. The exact GNH matrices correspond to the AE (a) *network with the entire MNIST dataset and the class* (a) *network with the entire CIFAR-*10 *training dataset, respectively. The compression rate and the accuracy are shown for low- and high-accuracy settings, respectively.*

(a) MNIST dataset ($n = 60,000$ images)

	10 samples		100 samples		1,000 samples		10,000 samples	
`accuracy`	%K	ϵ_F	%K	ϵ_F	%K	ϵ_F	%K	ϵ_F
`low`	1.74%	2.8E-1	1.69%	1.3E-1	1.68%	1.4E-1	1.68%	6.1E-2
`high`	17.1%	2.2E-2	16.9%	9.5E-3	16.6%	2.7E-3	16.2%	9.6E-4

(b) CIFAR-10 training dataset ($n = 50,000$ images)

	10 samples		100 samples		1,000 samples		10,000 samples	
`accuracy`	%K	ϵ_F	%K	ϵ_F	%K	ϵ_F	%K	ϵ_F
`low`	0.61%	9.7E-1	0.61%	5.3E-1	0.61%	4.1E-1	0.61%	4.3E-1
`high`	4.83%	9.7E-1	4.83%	3.6E-1	4.83%	1.9E-1	4.83%	7.3E-2

gular equality)

$$\|H - \tilde{H}_{\texttt{GOFMM}}\|_F = \|H - \tilde{H} + \tilde{H} - \tilde{H}_{\texttt{GOFMM}}\|_F \le \|H - \tilde{H}\|_F + \|\tilde{H} - \tilde{H}_{\texttt{GOFMM}}\|_F,$$

where the first term is the sampling error and the second term is the $\mathcal{H}$-matrix approximation error. As Table 6 shows, the former converges to zero and is independent of the data size. Table 7 shows that the latter also converges as the sampling becomes increasingly accurate, which justifies the overall approach.

6. Conclusions.

We have presented a fast method to evaluate entries in the GNH matrix of the MLP network, and our method consists of two parts: a precomputation algorithm and a fast Monte Carlo algorithm. While the precomputation allows evaluating entries in the GNH matrix exactly with reduced storage, the random sampling is based on the precomputation and further accelerates the evaluation. Let N be the number of weights, n be the data size, and d be the constant layer size. Our scheme requires $\mathcal{O}(n + d/\epsilon^2)$ work for any entry in the GNH matrix H, where ϵ is the accuracy, whereas the worst-case complexity to evaluate an entry exactly is $\mathcal{O}(Nn)$ through the matrix-free matvec. For example, the evaluation of $H_{N,N}$ would require $\mathcal{O}(Nn)$ work, while given our precomputation, it requires only $\mathcal{O}(n)$ work to compute a diagonal entry exactly (Remark 3.4). One application of this fast diagonal evaluation would be computing all the diagonals of H to precondition/accelerate the training of neural networks [42]. In this paper, we focused on applying the `GOFMM` to construct the $\mathcal{H}$-matrix approximation for the GNH matrix. As a result, we obtain an $\mathcal{H}$-matrix and its factorization for solving linear systems and eigenvalue problems with the GNH.

Two important directions for future research are (1) extending our method to other types of networks, such as convolutional networks, where weight matrices are highly structured (preliminary experiments on the VGG network show similar results as those in Table 3), and (2) incorporating our method in the context of a learning task, which would also require several algorithmic choices related to optimization, such as initialization, damping, and adding momentum.

Appendix A. McDiarmid's inequality.

THEOREM A.1. *Let $X_1, X_2, \dots, X_n$ be independent random variables taking values in the set $\mathcal{X}$. If a mapping $F : \mathcal{X}^n \to \mathbb{R}$ satisfies*

$$\sup_{x_1,\dots,x_i,x_i',\dots,x_n} |F(x_1,\dots,x_i,\dots,x_n) - F(x_1,\dots,x_i',\dots,x_n)| \leq \Delta_i \quad \forall i,$$

where $x_1,\dots,x_i,x_i',\dots,x_n \in \mathcal{X}$, then for all $\epsilon > 0$,

$$Pr(f - \mathbb{E}[f] \geq \epsilon) \leq exp\left(\frac{-2\epsilon^2}{\sum_{i=1}^n \Delta_i^2}\right).$$

REFERENCES

[1] A. AMINFAR, S. AMBIKASARAN, AND E. DARVE, *A fast block low-rank dense solver with applications to finite-element matrices*, J. Comput. Phys., 304 (2016), pp. 170–188.

[2] O. AXELSSON, *Iterative Solution Methods*, Cambridge University Press, Cambridge, 1994.

[3] J. BARNES AND P. HUT, *A hierarchical O(NlogN) force-calculation algorithm*, Nature, 324 (1986), pp. 446–449.

[4] M. BEBENDORF, *Hierarchical Matrices*, Springer-Verlag, Berlin, 2008.

[5] L. BOTTOU, F. E. CURTIS, AND J. NOCEDAL, *Optimization methods for large-scale machine learning*, SIAM Rev., 60 (2018), pp. 223–311.

[6] R. H. BYRD, G. M. CHIN, W. NEVEITT, AND J. NOCEDAL, *On the use of stochastic Hessian information in optimization methods for machine learning*, SIAM J. Optim., 21 (2011), pp. 977–995.

[7] Y. CARMON AND J. C. DUCHI, *Analysis of Krylov subspace solutions of regularized nonconvex quadratic problems*, in Advances in Neural Information Processing Systems, 2018, pp. 10728–10738, https://arxiv.org/pdf/1806.09222.pdf.

[8] C. CHEN, H. POURANSARI, S. RAJAMANICKAM, E. G. BOMAN, AND E. DARVE, *A distributed-memory hierarchical solver for general sparse linear systems*, Parallel Comput., 74 (2018), pp. 49–64.

[9] D. A. COHN, *Neural Network Exploration using Optimal Experiment Design*, in Advances in Neural Information Processing Systems, 1994, pp. 679–686.

[10] Y. N. DAUPHIN, R. PASCANU, C. GULCEHRE, K. CHO, S. GANGULI, AND Y. BENGIO, *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*, in Proceedings of the 27th International Conference on Neural Information Processing Systems, Vol. 2, NIPS'14, MIT Press, Cambridge, MA, 2014, pp. 2933–2941, http://dl.acm.org/citation.cfm?id=2969033.2969154.

[11] L. DINH, R. PASCANU, S. BENGIO, AND Y. BENGIO, *Sharp Minima Can Generalize for Deep Nets*, preprint, arXiv:1703.04933, 2017.

[12] P. GHYSELS, X. S. LI, F.-H. ROUET, S. WILLIAMS, AND A. NAPOV, *An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling*, SIAM J. Sci. Comput., 38 (2016), pp. S358–S384, https://doi.org/10.1137/15M1010117.

[13] P. E. GILL, W. MURRAY, AND M. H. WRIGHT, *Practical Optimization*, Academic Press, New York, 1981.

[14] I. GOODFELLOW, Y. BENGIO, A. COURVILLE, AND Y. BENGIO, *Deep Learning*, Vol. 1, MIT Press, Cambridge, MA, 2016.

[15] R. M. GOWER, N. L. ROUX, AND F. BACH, *Tracking the Gradients Using the Hessian: A New Look at Variance Reducing Stochastic Methods*, preprint, arXiv:1710.07462, 2017.

[16] L. GREENGARD, *Fast algorithms for classical physics*, Science, 265 (1994), pp. 909–914.

[17] W. HACKBUSCH, *Hierarchical Matrices: Algorithms and Analysis*, Springer Series in Computational Mathematics 49, Springer-Verlag, Berlin, 2015.

[18] N. HALKO, P.-G. MARTINSSON, AND J. TROPP, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Rev., 53 (2011), pp. 217–288.

[19] B. HASSIBI AND D. G. STORK, *Second order derivatives for network pruning: Optimal brain surgeon*, in Advances in Neural Information Processing Systems, 1993, pp. 164–171.

[20] G. HENNEQUIN, L. AITCHISON, AND M. LENGYEL, *Fast sampling-based inference in balanced neuronal networks*, in Advances in Neural Information Processing Systems, 2014, pp. 2240–2248.

[21] K. L. Ho and L. Ying, *Hierarchical Interpolative Factorization for Elliptic Operators: Integral Equations*, preprint, arXiv:1307.2666, 2013.

[22] T. Hofmann, B. Schölkopf, and A. J. Smola, *Kernel methods in machine learning*, Ann. Statist., (2008), pp. 1171–1220.

[23] D. A. Knoll and D. E. Keyes, *Jacobian-free Newton-Krylov methods: A survey of approaches and applications*, J. Comput. Phys., 193 (2004), pp. 357–397.

[24] J. Lafond, N. Vasilache, and L. Bottou, *Diagonal Rescaling for Neural Networks*, preprint, arXiv:1705.09319, 2017.

[25] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, *Efficient backprop*, in Neural Networks: Tricks of the Trade, Springer-Verlag, Berlin, 1998, pp. 9–50.

[26] D. Lee and A. G. Gray, *Fast high-dimensional kernel summations using the Monte Carlo multipole method*, in Proceedings of the 22nd Annual Conference on Neural Information Processing Systems (NIPS), 2008, pp. 929–936.

[27] W. B. March, B. Xiao, C. D. Yu, and G. Biros, *Askit: An efficient, parallel library for high-dimensional kernel summations*, SIAM J. Sci. Comput., 38 (2016), pp. S720–S749, https://doi.org/10.1137/15M1026468.

[28] J. Martens, *Deep learning via Hessian-free optimization*, in International Conference on Machine Learning, Vol. 27, 2010, pp. 735–742.

[29] J. Martens, *New Insights and Perspectives on the Natural Gradient Method*, preprint, arXiv:1412.1193, 2014.

[30] J. Martens, *Second-Order Optimization for Neural Networks*, Ph.D. thesis, University of Toronto, Toronto, ON, Canada, 2016.

[31] J. Martens and R. Grosse, *Optimizing neural betworks with Kronecker-factored approximate curvature*, in International Conference on Machine Learning, 2015, pp. 2408–2417.

[32] J. Martens and I. Sutskever, *Learning recurrent neural networks with Hessian-free optimization*, in Proceedings of the 28th International Conference on Machine Learning (ICML-11), Citeseer, 2011, pp. 1033–1040.

[33] P.-G. Martinsson and V. Rokhlin, *A fast direct solver for boundary integral equations in two dimensions*, J. Comput. Phys., 205 (2005), pp. 1–23.

[34] T. O'Leary-Roseberry, N. Alger, and O. Ghattas, *Inexact Newton Methods for Stochastic Non-Convex Optimization with Applications to Neural Network Training*, preprint, arXiv:1905.06738, 2019.

[35] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, R. Yokota, and S. Matsuoka, *Second-Order Optimization Method for Large Mini-Batch: Training Resnet-*50 *on Imagenet in* 35 *Epochs*, preprint, arXiv:1811.12019, 2018.

[36] N. L. Roux, P.-A. Manzagol, and Y. Bengio, *Topmoumoute online natural gradient algorithm*, in Advances in Neural Information Processing Systems, 2008, pp. 849–856.

[37] T. Takahashi, C. Chen, and E. Darve, *Parallelization of the Inverse Fast Multipole Method with an Application to Boundary Element Method*, preprint, arXiv:1905.10602, 2019.

[38] N. Tripuraneni, M. Stern, C. Jin, J. Regier, and M. I. Jordan, *Stochastic cubic regularization for fast nonconvex optimization*, in Advances in Neural Information Processing Systems, 2018, pp. 2904–2913, http://papers.nips.cc/paper/7554-stochastic-cubic-regularization-for-fast-nonconvex-optimization.pdf.

[39] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, *Fast algorithms for hierarchically semiseparable matrices*, Numer. Linear Algebra Appl., 17 (2010), pp. 953–976.

[40] P. Xu, J. Yang, F. Roosta-Khorasani, C. Ré, and M. W. Mahoney, *Sub-sampled Newton methods with non-uniform sampling*, in Advances in Neural Information Processing Systems, 2016, pp. 3000–3008.

[41] Z. Yao, A. Gholami, Q. Lei, K. Keutzer, and M. W. Mahoney, *Hessian-Based Analysis of Large Batch Training and Robustness to Adversaries*, preprint, arXiv:1802.08241, 2018.

[42] Z. Yao, A. Gholami, S. Shen, K. Keutzer, and M. W. Mahoney, *Adahessian: An Adaptive Second Order Optimizer for Machine Learning*, preprint, arXiv:2006.00719, 2020.

[43] H. Ye, L. Luo, and Z. Zhang, *Approximate Newton methods and their local convergence*, in Proceedings of the 34th International Conference on Machine Learning, Proceedings of Machine Learning Research, International Convention Centre, Sydney, Australia, 2017, pp. 3931–3939, http://proceedings.mlr.press/v70/ye17a.html.

[44] Y. You, Z. Zhang, C. Hsieh, J. Demmel, and K. Keutzer, *Imagenet Training in Minutes*, CoRR, abs/1709.05011, 2017.

[45] C. D. Yu, J. Levitt, S. Reiz, and G. Biros, *Geometry-oblivious FMM for compressing dense SPD matrices*, in Proceedings of SC17, AMC SCxy Conference series, IEEE, Denver, CO, 2017, https://doi.org/10.1145/3126908.3126921.

[46] C. D. Yu, S. Reiz, and G. Biros, *Distributed-memory hierarchical compression of dense SPD matrices*, in Proceedings of the International Conference for High Performance Computing,

Networking, Storage, and Analysis, SC '18, IEEE Press, Piscataway, NJ, 2018, pp. 15:1–15:15, https://dl.acm.org/citation.cfm?id=3291676.

[47] C. D. Yu, S. Riesz, and J. Levitt, `GOFMM` *Home Page*, https://github.com/ChenhanYu/hmlp, 2018.

[48] H. Zhang, C. Xiong, J. Bradbury, and R. Socher, *Block-Diagonal Hessian-Free Optimization for Training Neural Networks*, preprint, arXiv:1712.07296, 2017.