Research Article

Scalable Thermodynamic Second-order Optimization

Kaelan Donatella[1], Samuel Duffield[1], Denis Melanson[1], Maxwell Aifer[1], Phoebe Klett[1], Rajath Salegame[1], Zach Belateche[1], Gavin Crooks[1], Antonio J. Martinez[1], Patrick J. Coles[1]

1. Normal Computing

Many hardware proposals have aimed to accelerate inference in AI workloads. Less attention has been paid to hardware acceleration of training, despite the enormous societal impact of rapid training of AI models. Physics-based computers, such as thermodynamic computers, offer an efficient means to solve key primitives in AI training algorithms. Optimizers that normally would be computationally out-of-reach (e.g., due to expensive matrix inversions) on digital hardware could be unlocked with physics-based hardware. In this work, we propose a scalable algorithm for employing thermodynamic computers to accelerate a popular second-order optimizer called Kronecker-factored approximate curvature (K-FAC). Our asymptotic complexity analysis predicts increasing advantage with our algorithm as n, the number of neurons per layer, increases. Numerical experiments show that even under significant quantization noise, the benefits of second-order optimization can be preserved. Finally, we predict substantial speedups for large-scale vision and graph problems based on realistic hardware characteristics.

Corresponding author: Kaelan Donatella, kaelan@normalcomputing.ai

1. Introduction

Due to their fast convergence properties[1][2], second-order training methods hold great promise to train neural networks efficiently. Such methods form a local quadratic approximation to the landscape and update the parameters by optimizing this approximation within some region[3]. Thus, these methods should make more progress per iteration than vanilla gradient descent because of their more detailed modeling of the landscape[1]. Beyond neural network training, second-order methods are also highly popular in reinforcement learning[4][5].

Despite these advantages, first-order methods like stochastic gradient descent (SGD) or Adam[6] are typically used in practical settings. This is largely because of the large computational overhead of inverting the relevant curvature matrix (e.g., the Fisher matrix) in second-order methods. Methods that employ block-diagonal approximations to the curvature matrix, such as Kronecker-Factored Approximate Curvature (K-FAC), have enabled large-scale applications of second-order methods, including to modern neural network architectures with billions of parameters. Nevertheless, their computational costs remain substantially higher than those of first-order optimizers, limiting their practical competitiveness.

Our perspective is that optimizer preference is dictated by the underlying computing hardware that is running that optimizer. While SGD may be preferable on digital hardware, second-order methods could be superior if the computational substrate could accelerate the key bottleneck of these methods. In this work, we investigate the potential for thermodynamic computers to address bottlenecks in second-order optimizers. These computers are physics-based devices that utilize a physical system's tendency to relax to thermodynamic equilibrium as an algorithmic primitive[7][8].

Recent work showed that thermodynamic computers could unlock a linear speedup (linear in the matrix dimension) when running linear algebraic primitives[9][10] like solving linear systems, inverting matrices, and exponentiating matrices. Because Natural Gradient Descent (NGD) involves solving a linear system (associated with computing the natural gradient from the standard gradient), running NGD on a thermodynamic computer is predicted to have a linear speedup in the number of neural network parameters[11]. However, Thermodynamic NGD was restricted to fine-tuning applications because it is impractical to build a billion-dimensional device that would be required to fully train all parameters of a large-scale neural network. Thus, it remains an open question for how to make Thermodynamic NGD scalable and practical for applications beyond fine tuning.

Thermodynamic NGD is nevertheless appealing because its asymptotic complexity is similar to that of a first-order method, and hence if it could be made practical, one could run second-order methods at the computational cost of a first-order method. In this work, we address the practicality of Thermodynamic NGD by considering the K-FAC algorithm. We show how K-FAC, which employs a block-diagonal approximation to the Fisher matrix, can be turned into a thermodynamic algorithm.

We show that the asymptotic scaling of our Thermodynamic K-FAC algorithm leads to a linear advantage (i.e., the advantage grows linearly with the width n of the neural network) over standard K-

FAC for both the runtime cost and memory cost. Crucially, the block-diagonal nature of K-FAC allows for practical implementation on thermodynamic hardware, as the matrices involved have dimension on the order of a thousand rather than a billion. Hence this provides a scalable means to do (approximate) natural gradient descent with thermodynamic hardware.

To account for the finite precision of thermodynamic hardware, we ran numerical experiments investigating the impact of quantization noise on the performance. We found that even under significant quantization noise, the benefits of second-order optimization (e.g., relative to the Adam optimizer) can be preserved. This suggests some robustness to imprecision for our Thermodynamic K-FAC algorithm.

Finally, our simulations of large-scale vision and language problems show evidence that Thermodynamic K-FAC outperforms both standard K-FAC as well as Adam. Moreover, by altering the hyperparameters of the model (e.g., increasing the width of the network), speedups even larger than those we show in our numerics can be unlocked.

2. Related work

There is a large body of theoretical research on NGD[1][3][12] arguing that it requires fewer iterations than SGD to converge to the same value of the loss in specific settings. K-FAC[13] aims to reduce this complexity and invokes a block-wise approximation of the curvature matrix, which may not always hold. While first introduced for multi-layer perceptrons, K-FAC has been applied to more complex architectures, such as recurrent neural networks[14] and transformers[15], where additional approximations have to be made and where the associated computational overhead can vary. While the K-FAC approximation is uncontrolled, there is a large body of empirical evidence showing its faster convergence per step with respect to Adam and its variants[13][14][15][16][17]. It has also been shown that K-FAC extends the critical batch size for a variety of tasks[18], decreasing the diminishing returns usually seen by scaling up the batch size in neural network training.

However, because of the per-step overhead of K-FAC, it remains roughly on-par with first-order methods[15] in terms of per-wall clock time performance. In this work we focus on reducing the runtime per step of the K-FAC optimizer, which directly makes it more competitive.

At the core of our approach are thermodynamic algorithms[9] for solving linear systems and inverting matrices. We remark that alternative analog methods for solving these kinds of problems can be found in Refs.[19][20]. In addition, alternative approaches to thermodynamic computing have been proposed[21][22][23][24], applications beyond linear algebra have explored such as Bayesian inference[25] and quadratic programming[26], and closely related to thermodynamic computing is probabilistic computing[27][28] and reversible computing[29].

While several approaches have been proposed to accelerate training of AI models using novel hardware, these efforts typically aim at reducing the constant coefficients appearing in the time cost of computation. For example, analog computing devices have been proposed to achieve reduced time and energy costs of training relative to available digital technology[30][31][32][33]. These devices are generally limited to training a neural network that has a specific architecture (corresponding to the structure of the analog device).

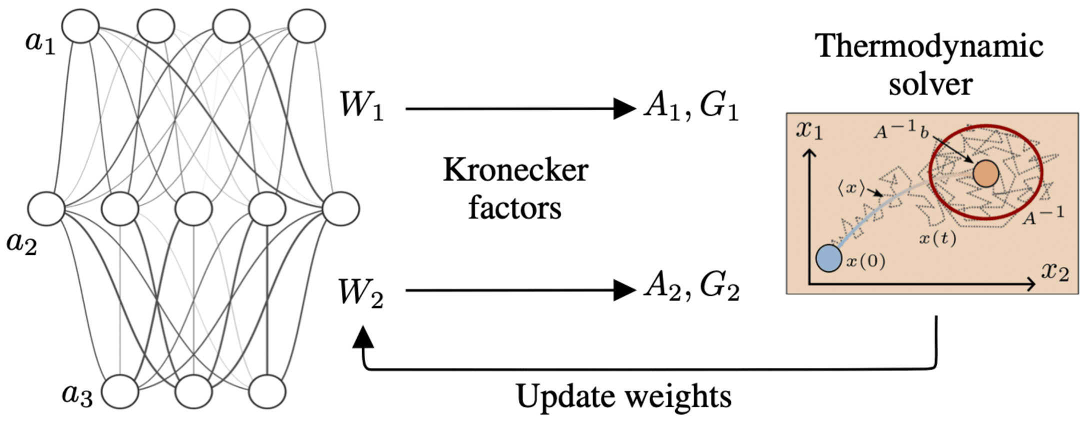

Figure 1. Overview of the thermodynamic algorithm for K-FAC. On the left is shown a two-layer neural network with weight matrices W_1 and W_2 and activations a_1, a_2, a_3 that are stored on a digital device. From these quantities Kronecker factors A_ℓ and B_ℓ are computed and sent to the thermodynamic solver, which inverts them or solves a linear system where they enter as the positive semi-definite matrix. Then, the result is sent back to the digital device and the weights are updated. Note that this algorithm is easily parallelized, e.g., many thermodynamic solvers can be used to compute the K-FAC update rule (Eq. (14)) for one or more layers each.

A strength of our approach is its flexibility with respect to model architecture. Although this same strength appeared in the Thermodynamic NGD algorithm of Ref.[11], that algorithm would either require (1) a large-scale hardware (with a number of physical components scaling linearly with the number of model parameters) for which important scalability challenges have yet to be solved, or (2) restricting the training tasks only to fine-tuning. In this sense, Ref.[11] did not fully solve the issue of training large-scale foundational models. Thus, a key insight of the current paper is to make the training of large-scale AI models practical and scalable for thermodynamic hardware. Moreover, our complexity analysis (Table 1) suggests that the per-iteration complexity of K-FAC can be made similar to that of a first-order training method.

Optimizer	Runtime	Memory
SGD/Adam	$\mathcal{O}(bn^2)$	$\mathcal{O}(n^2)$
K-FAC	$\mathcal{O}(bn^2 + n^3)$	$\mathcal{O}(bn + n^2)$
Thermodynamic K-FAC	$\mathcal{O}(bn^2 + n^2\kappa^2)$	$\mathcal{O}(bn)$
Thermodynamic K-FAC (w/ EMA)	$\mathcal{O}(bn^2 + n^2\kappa^2)$	$\mathcal{O}(bn + n^2)$

Table 1. Runtime and memory complexity (per layer) of optimizers considered in this paper. Here we consider an MLP, where n is the number of neurons per layer, hence there are n^2 parameters per layer, and b is the batch size. We also assume that the Kronecker factors all have condition numbers at most κ. Full complexities that take into account output and weight sharing dimensions for the expand and reduce techniques[15] can be found in Table 2 in Appendix B.

3. Natural gradient descent and K-FAC

3.1. Natural gradient descent

Let us consider a supervised learning setting, where the goal is to minimize an objective function defined as:

$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} L(y, f_\theta(x)), \tag{1}$$

where $L(y, f_\theta(x)) \in \mathbb{R}$ is a loss function and $f_\theta(x)$ is the forward function that is parametrized by $\theta \in \mathbb{R}^N$. These functions depend on input data and labels $(x, y) \in \mathcal{D}$, with $\mathcal{D}$ a given training dataset.

The update rule for Natural Gradient Descent (NGD)[34] is given by

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot (C^{(t)})^{-1} \nabla_{\theta^{(t)}} \mathcal{L}(y, f_{\theta^{(t)}}(x)) \tag{2}$$

where η is the learning rate, $C^{(t)}$ is a pre-conditioning matrix (which can depend on $\theta^{(t)}, x, y$), and $\nabla_{\theta^{(t)}} \mathcal{L}(y, f_{\theta^{(t)}}(x))$ is the gradient of the objective function. For NGD, the preconditioning matrix is of size $N \times N$, with N the total number of parameters of the model. Therefore, the NGD update quickly becomes impractical for modern neural networks for billions of parameters, as even storing the full pre-conditioning matrix is too expensive.

3.2. Kronecker-factored approximate curvature (K-FAC)

One possible solution to this impractical scaling was proposed in Ref[13], known as Kronecker-factored approximate curvature (K-FAC). This method consists of simplifying the computation of the inverse of the preconditioning matrix C by exploiting its structure.

Let us consider a deep neural network with a layered structure, such that the forward function can be written as:

$$f_\theta = f_{\theta_L} \circ \cdots \circ f_{\theta_2} \circ f_{\theta_1}, \tag{3}$$

with $\theta = \mathrm{concat}(\theta_1, \theta_2, \ldots, \theta_L)$ and L the number of layers; $\mathrm{concat}(\cdot, \ldots, \cdot)$ concatenates vector inputs to a larger vector. For a multilayer perceptron (MLP), we have:

$$f_{\theta_\ell}(a_{\ell-1}) = \phi(W_\ell a_{\ell-1} + v_\ell) = \phi(\overline{W}_\ell \bar{a}_{\ell-1}) \tag{4}$$

where ϕ is an activation function, $a_{\ell-1} \in \mathbb{R}^{n_{\ell,\mathrm{in}}}$, $W_\ell \in \mathbb{R}^{n_{\ell,\mathrm{out}} \times n_{\ell,\mathrm{in}}}$, and $v_\ell \in \mathbb{R}^{n_{\ell,\mathrm{out}}}$. The bias vector v_ℓ can be included as a column in the weight matrix (we denote this expanded weight matrix $\overline{W}_\ell$, and a constant value of unity may be appended to a_ℓ (which we denote $\bar{a}_\ell$), yielding the right-hand side of Eq. (4). We also define the pre-activation $s_\ell := \overline{W}_\ell \bar{a}_{\ell-1}$. The parameter vector can be written as

$$\theta_\ell = \mathrm{vec}(\overline{W}_\ell), \tag{5}$$

where $\mathrm{vec}(\cdot)$ vectorizes a matrix by concatenating its column vectors. The total number of parameters in the ℓth layer (i.e., the length of θ_ℓ) is $P_\ell = n_{\ell,\mathrm{out}} n_{\ell,\mathrm{in}} + n_{\ell,\mathrm{out}}$.

We consider the empirical Fisher as the pre-conditioning matrix and follow Ref. [13] (though everything can be extended to the generalised Gauss-Newton matrix with a block-diagonal approximation). In what follows all expectation values are taken over mini-batches of data, with a batchsize *b*. The empirical Fisher is given by:

$$F := \mathbb{E}[D\theta D\theta^\top] \tag{6}$$

where

$$D\theta = [\mathrm{vec}(D\theta_1)^\top, \mathrm{vec}(D\theta_2)^\top, \ldots, \mathrm{vec}(D\theta_L)^\top]^\top, \quad D\theta_\ell := \frac{d\mathcal{L}(y, f_\theta(x))}{d\theta_\ell}. \tag{7}$$

We can therefore rewrite F as:

$$\begin{aligned} F &= \mathbb{E}[[\mathrm{vec}(D\theta_1)^\top, \mathrm{vec}(D\theta_2)^\top, \ldots, \mathrm{vec}(D\theta_L)^\top]^\top [\mathrm{vec}(D\theta_1)^\top, \mathrm{vec}(D\theta_2)^\top, \ldots, \mathrm{vec}(D\theta_L)]] \\ &= \begin{pmatrix} \mathbb{E}[\mathrm{vec}(D\theta_1)\mathrm{vec}(D\theta_1)^\top] & \mathbb{E}[\mathrm{vec}(D\theta_1)\mathrm{vec}(D\theta_2)^\top] & \ldots & \mathbb{E}[\mathrm{vec}(D\theta_1)\mathrm{vec}(D\theta_L)^\top] \\ \mathbb{E}[\mathrm{vec}(D\theta_2)\mathrm{vec}(D\theta_1)^\top] & \mathbb{E}[\mathrm{vec}(D\theta_2)\mathrm{vec}(D\theta_2)^\top] & \ldots & \mathbb{E}[\mathrm{vec}(D\theta_2)\mathrm{vec}(D\theta_L)^\top] \\ \vdots & \vdots & \ddots & \vdots \\ \mathbb{E}[\mathrm{vec}(D\theta_L)\mathrm{vec}(D\theta_1)^\top] & \mathbb{E}[\mathrm{vec}(D\theta_L)\mathrm{vec}(D\theta_2)^\top] & \ldots & \mathbb{E}[\mathrm{vec}(D\theta_L)\mathrm{vec}(D\theta_L)^\top] \end{pmatrix} \end{aligned} \tag{8}$$

Thus, we see that F is a block matrix with the (ℓ, ℓ')th block being $F_{\ell,\ell'} = \mathbb{E}[\mathrm{vec}(D\theta_\ell)\mathrm{vec}(D\theta_{\ell'}^\top)]$. For an MLP, we have

$$D\theta_\ell = g_\ell \bar{a}_{\ell-1}^\top, \text{ where } g_\ell := Ds_\ell. \tag{9}$$

Using the identity $\mathrm{vec}(BC^\top) = C \otimes B$, we therefore have the blocks $F_{\ell,\ell'}$ given by

$$F_{\ell,\ell'} = \mathbb{E}[(\bar{a}_{\ell-1} \otimes g_\ell)(\bar{a}_{\ell'-1} \otimes g_{\ell'}^\top)] = \mathbb{E}[(\bar{a}_{\ell-1} \otimes g_\ell)(\bar{a}_{\ell'-1}^\top \otimes g_{\ell'}^\top)] = \mathbb{E}[(\bar{a}_{\ell-1}\bar{a}_{\ell'-1}^\top \otimes g_\ell g_{\ell'}^\top)]. \tag{10}$$

Crucially, the K-FAC approximation consists in first replacing the average of the Kronecker product by a Kronecker product of averages, as:

$$F_{\ell,\ell'} \approx \mathbb{E}[\bar{a}_{\ell-1}\bar{a}_{\ell'-1}^\top] \otimes \mathbb{E}[g_\ell g_{\ell'}^\top], \tag{11}$$

which is an empirically motivated approximation (rather than a theoretically motivated one) [13]. We define the following matrices, called the Kronecker factors, as

$$A_\ell = \mathbb{E}[\bar{a}_\ell \bar{a}_\ell^\top] \quad \text{and} \quad G_\ell = \mathbb{E}[g_\ell g_\ell^\top], \tag{12}$$

and we make the further approximation that the Fisher is block-diagonal. Here, the dimensions of A_ℓ and G_ℓ are, respectively, $(n_{\ell,\mathrm{in}} + 1) \times (n_{\ell,\mathrm{in}} + 1)$ and $(n_{\ell,\mathrm{out}}) \times (n_{\ell,\mathrm{out}})$. Note that both A_ℓ and G_ℓ are symmetric positive semi-definite (SPSD) matrices.

By exploiting the following relations:

$$(B \otimes C)^{-1} = B^{-1} \otimes C^{-1} \quad \text{and} \quad (B \otimes C)\mathrm{vec}(X) = \mathrm{vec}(CXB^\top), \tag{13}$$

we can then derive the per-layer parameter update from Eq. (2). This results in the K-FAC update rule:

$$\theta_\ell^{(t+1)} = \theta_\ell^{(t)} - \alpha u_\ell^{(t)}, \tag{14}$$

where $u_\ell^{(t)} = \mathrm{vec}(U_\ell^{(t)})$. The matrix $U_\ell^{(t)}$ is given by

$$U_\ell^{(t)} = \left(G_\ell^{(t)}\right)^{-1} (D\Theta_\ell) \left(A_{\ell-1}^{(t)}\right)^{-1} \tag{15}$$

where $D\Theta_\ell$ is the matrix satisfying $\text{vec}(D\Theta_\ell) = D\theta_\ell$.

Because the Kronecker factors are estimated with minibatches, it is common to compute exponential moving averages (EMA) on them in order to aggregate batch information and reduce minibatch noise. These moving averages, denoted $\tilde{A}_\ell^{(t+1)}$ and $\tilde{G}_\ell^{(t+1)}$, are defined as:

$$\tilde{A}_\ell^{(t+1)} = \beta_A \tilde{A}_\ell^{(t)} + (1-\beta_A) A_\ell^{(t+1)} \tag{16}$$

$$\tilde{G}_\ell^{(t+1)} = \beta_G \tilde{G}_\ell^{(t)} + (1-\beta_G) G_\ell^{(t+1)}. \tag{17}$$

As will be explained further, using an EMA requires to explicitly construct the Kronecker factors, and hence modifies the requirements of the thermodynamic hardware used to accelerate K-FAC.

3.3. K-FAC for general weight-sharing neural networks

While the derivation of the K-FAC update shown in Eq. (14) is done for MLPs, a similar treatment may be done to any weight-sharing neural network[15], which includes convolutional neural networks, graph neural networks[35], and transformers[36]. This modifies the computation of the Kronecker factors, as the activations a_ℓ now are expanded with a weight-sharing dimension of size R (for sequence to sequence models, the sequence length). The gradients g_ℓ are also redefined as Jacobians $g_{\ell,r,r'} = \frac{d\mathcal{L}(y, f_\theta(x))_r}{ds_{l,r'}}$. For minibatches of size b, the activations are therefore of size $n_{\text{in}} \times b \times R$ (denoted as $a_{\ell,k,r}$) and the gradients are of size $n_{\text{out}} \times b \times R \times R$(denoted as $g_{\ell,k,j,r}$)[15]. The Kronecker factors may therefore be computed as:

$$A_\ell = \frac{1}{bR} \sum_{k,r}^{b,R} a_{\ell,k,r} a_{\ell,k,r}^\top \tag{18}$$

$$G_\ell = \sum_{k,j,r}^{b,R,R} g_{\ell,k,j,r} g_{\ell,k,j,r}^\top \tag{19}$$

when the loss has bR terms (in the case of language generation or translation, for example, where tokens have to be compared along the whole sequence when computing the loss). This is the expand setting, and this form of K-FAC is known as K-FAC-expand[15]. For a layer with $n_{\text{in}} = n_{\text{out}} = n$, K-FAC-expand has a per-layer time complexity $\mathcal{O}(bRn^2 + n^3)$ (where the first term comes from calculating the A_ℓ and B_ℓ factors and the second term from the matrix inverse) and memory $\mathcal{O}(bRn + n^2)$.

When the loss has b terms (in the case of classification), the summation over the weight-sharing dimension may in fact be performed before computing the model output $f_\theta(x)$. This results in gradients being defined as $g_{\ell,r} = \frac{d\mathcal{L}(y,f_\theta(x))}{ds_{l,r}}$ (thus having one less dimension of size R). This is known as the reduce setting, in which the Kronecker factors then become:

$$A_\ell = \frac{1}{bR^2}\sum_k^b \left(\sum_r^R a_{\ell,k,r}\right)\left(\sum_{r'}^R a_{\ell,k,r'}^\top\right) \tag{20}$$

$$G_\ell = \sum_k^b \left(\sum_r^R g_{\ell,k,r}\right)\left(\sum_{r'}^R g_{\ell,k,r'}^\top\right) \tag{21}$$

This is known as K-FAC-reduce, and has better computational complexity (due to less sums being performed over the sequence length dimension) while yielding similar results[15] hence is preferred in the reduce setting. K-FAC-reduce has a per-layer time complexity $\mathcal{O}(bn(n+R)+n^3)$ (dominated by computing the G_ℓ factors) and memory $\mathcal{O}(bn+n^2)$.

4. Accelerating K-FAC with thermodynamic hardware

4.1. Potential bottleneck due to matrix inversion

Mathematically, K-FAC amounts to estimating the natural gradient for each layer, which involves computing the matrix $U_\ell^{(t)}$ in Eq. (15), repeated here for convenience:

$$U_\ell^{(t)} = \left(G_\ell^{(t)}\right)^{-1}(D\Theta_\ell)\left(A_{\ell-1}^{(t)}\right)^{-1}$$

Under certain conditions, the computation of these $U_\ell^{(t)}$ matrices could potentially be a computational bottleneck (for standard digital hardware), due to the matrix inversions required. Specifically, once the Kronecker factors $G_\ell^{(t)}$ and $A_{\ell-1}^{(t)}$ have been constructed, the computation of $U_\ell^{(t)}$ can be performed with either of the following two methods:

- **Method 1 (Inversion method)** Invert $G_\ell^{(t)}$ and $A_{\ell-1}^{(t)}, \forall \ell$, then multiply the inverses with $(D\Theta_\ell)$.
- **Method 2 (Linear systems method)** For each column j of the matrix $D\Theta_l$, solve the linear systems $G_\ell^{(t)}x = (D\Theta_l)_j$ to obtain the matrix $Q_\ell^{(t)} = \left(G_\ell^{(t)}\right)^{-1}(D\Theta_\ell)$. Then solve the linear systems $(Q_\ell^{(t)})_k = xA_{\ell-1}^{(t)}$ for each column k of $Q_\ell^{(t)}$.

These two methods both have cubic complexities, as for the inversion a Cholesky decomposition or singular value decomposition would be used and in the second case $O(n)$ linear systems are solved with a complexity $O(n^2\kappa)$ if the conjugate gradient method is used, with κ the condition number of the matrices involved. For our experiments presented below (in Section 5), Method 1 was used and only the inversion was profiled and replaced with another solver due to easier implementation.

However, we expect Method 2 to be more efficient when using the thermodynamic algorithm presented below as it does not require one to construct the inverse explicitly in memory (and has less operations overall).

4.2. Thermodynamic linear algebra framework

Both of the methods given above for computing $U_\ell^{(t)}$ can be accelerated with thermodynamic hardware[9]. Here we briefly review the thermodynamic linear algebra framework.

Suppose that one is given a positive semi-definite matrix M, and the goal is to either to compute the inverse, M^{-1}, or to solve a linear system $Mx = b$ for some vector b. In either case, one can upload the matrix M to the coupling matrix for a system of coupled harmonic oscillators. This system of coupled harmonic oscillators can, e.g., take the form of RC circuits coupled through capacitive or resistive coupling as discussed in Refs.[8][9][37] (in which case M would be mapped to either the capacitive couplings or the resistive couplings). For more details, see Appendix E. Moreover, we assume that each oscillator in this system has a stochastic noise source (with inverse temperature β), and hence the overall system can be viewed as a thermodynamic computer that is called the stochastic processing unit (SPU).

To solve the relevant linear algebra problem, one allows the SPU to evolve over time according to its natural physical dynamics, which are described by an Ornstein–Uhlenbeck (OU) process. Namely, the dynamics are given by the following stochastic differential equation (SDE):

$$dx = -(Mx - b)dt + \mathcal{N}[0, 2\beta^{-1}dt], \tag{22}$$

where the vector b is only relevant to the linear systems case and it can be set to zero if only matrix inversion is desired. (Note that, physically, the b vector corresponds to a local force on each oscillator and hence could corresponds to a locally applied DC voltage in a circuit implementation of an SPU.)

Allowing the SPU to evolve over time according to Eq. (22) corresponds to allowing the system to relax to thermodynamic equilibrium. Once it reaches equilibrium, x is Boltzmann distributed, where the Boltzmann distribution is a Gaussian (since the potential is a quadratic form). Specifically, x is distributed according to:

$$x \sim \mathcal{N}[M^{-1}b, \beta^{-1}M^{-1}]. \tag{23}$$

One can see that the first moment of this distribution is the solution to the linear system $Mx = b$, and hence the linear system is solved by sampling x and estimating the mean value. Moreover, second moment of this distribution is proportional to M^{-1}, and hence the inverse is computed by estimating the covariances of the samples of x.

4.3. Thermodynamic K-FAC algorithm

Thermodynamic linear algebra routines can be directly applied to the K-FAC updates in Eq. (15). In particular, a thermodynamic solver can handle an n-dimensional linear system in $\mathcal{O}(n\kappa^2)$ time[1], where κ is the matrix condition number. By solving one system per column, we can effectively invert an $n \times n$ matrix in $\mathcal{O}(n^2\kappa^2)$ time, or $\mathcal{O}(n\kappa^2)$ time if the systems are solved in parallel. In the K-FAC setting, this applies directly to the matrix inversions appearing in Method 1, or the linear systems appearing in Method 2 (see Sec. 4.1 for discussion of these two methods). We also note that the matrix inverses may be directly obtained by estimating the covariance of the stationary distribution given in Eq. (23). However, Method 2 could be more efficient as it completely avoids matrix-matrix multiplications to compute Eq. (14). Thus the thermodynamic K-FAC algorithm using Method 2 has runtime complexity $\mathcal{O}(n^2\kappa^2)$ as it involves solving $2n$ linear systems.

Furthermore, one may embed the necessary Kronecker factors (e.g., $G_\ell^{(t)}, A_{\ell-1}^{(t)}$) onto thermodynamic hardware for direct, on-device computation of solutions to the corresponding linear systems to reduce the digital memory footprint. This can be done in two ways:

- Compute and store the Kronecker factors digitally, then transfer them to thermodynamic hardware to perform the linear solves. This is straightforward to parallelize across different thermodynamic solvers, each handling one or more layers.
- Port activations and gradients directly into hardware, implementing the sums over indices (as in Refs. [11] and shown in Appendix E) on rectangular resistor arrays. This method will be most efficient for small batch sizes and sequence lengths, as the number of physical components scales linearly with these two quantities. For large batches or sequence lengths, it may be more practical to pre-compute the factors rather than porting high-dimensional data to the thermodynamic solver.

Finally, note that even if the linear solves or inversions are performed directly on thermodynamic hardware, one still needs explicit Kronecker factors when using an exponential moving average on G_ℓ and $A_{\ell-1}$, since one needs to average previously calculated estimates with current values, and this cannot be done by updating activations or gradients directly. Thus, digital computation of these factors remains necessary in this case, leading to a worse runtime and memory complexity when using moving averages.

4.4. Computational complexity

The layerwise block diagonal K-FAC approximation allows the optimizer to scale to very deep neural networks thus avoiding a key limitation of natural gradient descent[1][11]. However, the scaling for wide networks with many neurons per layer remains expensive due to the matrix inversions required in (15). Here, the matrices A and G(dropping the layer ℓ subscript and iteration t superscript for brevity) have dimension $n \times n$ where n is the number of neurons or width of the layer. Thus the matrix inversion represents the key computational bottleneck for reasonable wide layers, with complexity scaling as $\mathcal{O}(n^3)$ in general implementations.

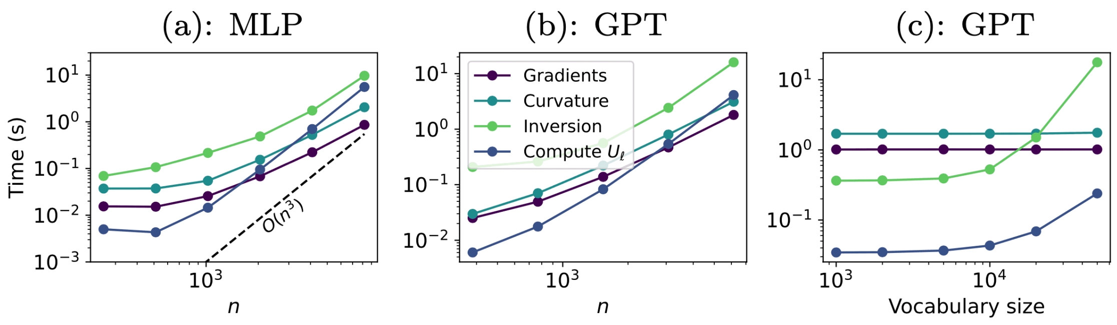

Figure 2. Profiling of the K-FAC update for different architectures. Panel (a): K-FAC update time contributions for an MLP with a fixed depth of 50, with varying number of neurons n on each layer. Panel (b): K-FAC update time contributions for a GPT architecture (based on Ref. [38]). with varying embedding dimension, which is the number of neurons n in the linear layers. Panel (c): GPT architecture with varying vocabulary size with a fixed embedding dimension. For all plots the reported times are averaged over 10 repetitions and measured on an Nvidia A100 GPU.

In Fig. 2, the total K-FAC update times is shown for an MLP (panel (a)) and a transformer (GPT)[38], for varying widths, in Figs. 2(a)-(b) and vocabulary sizes, in Fig. 2(c). For all panels the update time is broken into four components: calculating gradients (through auto-differentiation), constructing the curvature matrices, inverting matrices and computing the final update U_ℓ (Eq.(14)). Throughout this work we consider that the gradient calculation must be performed on a digital device, as it uses auto-differentiation. We observe that as expected in panels (a) and (b), the inversion time and the computation of U_ℓ (which contains matrix multiplication) approach cubic scaling as n increases (and is not exactly cubic for low dimensions thanks to parallelization). We also observe that inversion is the main bottleneck across all values of n and for large vocabulary sizes. In the case of large vocabulary sizes, the inversion completely dominates over other contributions, as vocabulary size directly impacts the Kronecker factors of the embedding layers. These measurements are all performed on an Nvidia A100 GPU.

As described in Section 4.2, thermodynamic hardware[8] utilizes the physical equilibration of an analog system to efficiently solve linear systems and matrix inversions faster than digital counterparts[9]. In both the inversion and linear systems methods described in Section 4.1, the thermodynamic solver described in[9] costs $\mathcal{O}(n^2\kappa^2)$ for a single layer and matrix condition number κ. We also note that in practice, the time constant of the hardware's dynamics enters as a constant factor in the runtime, which can be engineered to be extremely small (on the order of a microsecond[37][19]).

Table 1 compares the computational single-iteration, single-layer complexities of the introduced thermodynamic K-FAC optimizers (using the linear systems solves to avoid matrix multiplications when computing U_ℓ) with SGD and digital K-FAC[14]. SGD and variants such as Adam use a diagonal approximation to the Fisher information and are therefore very cheap to run per iteration, however for practical neural networks, the diagonal approximation to the Fisher is a poor one and leads to many more iterations to reach convergence[13][39][15]. In the simpler case where we do not use the moving average Kronecker factors in Equations (16-17), the rectangular components of the matrices in (12) can be sent directly to the thermodynamic hardware at a memory cost of $\mathcal{O}(n)$ avoiding the $\mathcal{O}(n^2)$ memory cost of constructing the full Kronecker factors. However, to apply the moving averages (which smooth out the noise from mini-batching), the Kronecker factors need to be constructed. More extensive details on computational complexities for the weight-sharing K-FAC techniques for more general models[15] can be found in Table 2 in Appendix B.

4.5. Sources of error

The Thermodynamic K-FAC algorithm is intended for thermodynamic hardware, an inherently noisy analog platform. The dominant sources of error in this setting include device mismatch, input quantization, and output quantization. Device mismatch arises from fabrication inconsistencies and is thus outside the scope of this paper. Instead, we focus on how quantization errors affect the stability and performance of the optimizer. Quantization noise is general (readout noise with analog-to-digital converters is essentially output quantization) and common to both analog and digital accelerators, making our analysis broadly relevant. In subsequent sections, we strategies to reduce these errors through better quantization schemes. Note that error mitigation methods may also be employed to reduce such errors[40], and that the thermodynamic nature of our algorithm makes it inherently robust to thermal noise.

4.5.1. *Input quantization*

Given a positive semi-definite input matrix M, the thermodynamic hardware stores an approximate quantized version, $\tilde{M}$, up to a certain level of bit-precision in integer format. This quantization can make $\tilde{M}$ no longer positive semi-definite, thus creating instabilities in the thermodynamic system. This would have catastrophic effect on the solver and the algorithm would simply fail. The traditional method to overcome this issue is to ensure that the hardware has more bits of precision for each element of the matrix, such as 32 or 64 bits[39], in order to not have the quantization error make the matrix non-definite. This approach works well, but is quite costly in terms of resources. An alternative method to overcome the problem is to modify how we quantize and store the input matrix using a conservative quantizer, that is, a quantizer that conserves the definiteness of the input matrix. In our numerical experiments (presented below), we use a diagonal-dominant quantization method[41]:

- Round each off-diagonal matrix element to the nearest value available in the hardware.
- Calculate the sum of off-diagonal rounding errors in each row of the matrix.
- Add the sum of rounding errors to the diagonal.
- Round all the shifted diagonal elements up.

While this method ensures the definiteness of the quantized matrix, it increases the error significantly in some cases since it adds an error term proportional to the dimension of the matrix. We note that alternative approaches can be used to make the diagonal shift smaller and thus reduce the error, but these are either more computationally demanding[41] or are based on empirically determined constants that are application-specific[42].

4.5.2. *Output quantization*

An additional source of quantization error one can expect from a dedicated linear algebra accelerator is the output quantization. This quantization comes from the conversion of the precision of the accelerator and the precision of the rest of the computation. For example, if the accelerator is analog, then the output quantization comes from the analog to digital conversion.

5. Experiments

5.1. AlgoPerf experiments

In this section we present numerical results of simulating the thermodynamic K-FAC method on AlgoPerf workloads[43]. These workloads provide strong baselines and fixed model architectures, enabling direct comparisons of training algorithms without confounding factors such as architecture changes. We consider two workloads: training a vision transformer (ViT) on ImageNet, and training a graph neural network (GNN) on the ogbg-molpcba data set, a popular property prediction dataset for small molecules. We note that these datasets were also used for benchmarking in Ref.[15], which introduced K-FAC-reduce and K-FAC-expand.

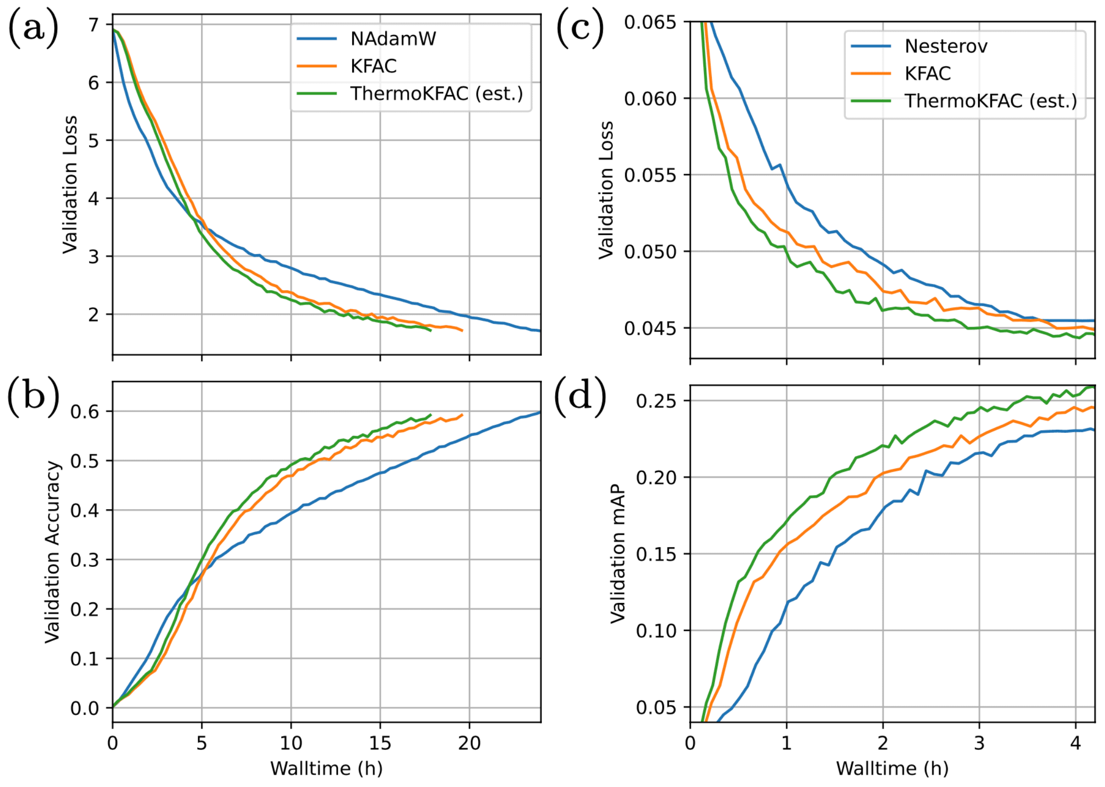

Figure 3. Results on ImageNet and OGBG. Panels (a-b): validation loss and validation accuracy for the NAdamW (the baseline given by AlgoPerf), K-FAC and Thermodynamic K-FAC (estimated) optimizers as a function of the wall-clock time for training a ViT on ImageNet. Panels (c-d): validation loss and validation mean-average precision (mAP) for the Nesterov (baseline), K-FAC and Thermodynamic K-FAC (estimated) optimizers as a function of the wall-clock time for training a GNN on ogbg-molpcba. For the baselines, the hyperparameters are directly taken from the AlgoPerf benchmark and were tuned for the K-FAC optimizers (see Appendix C).

5.1.1. ViT on ImageNet

The first workload that we consider is training a ViT on ImageNet. This task involves a state-of-the-art vision model, on a challenging image classification dataset. The baseline optimizer for this workload is Nesterov-accelerated Adam with weight-decay (NAdamW)[44]. For these experiments as with others in this work, the baselines use the hyperparameters from the AlgoPerf paper, thus setting a robust baseline to beat that was found externally after extensive tuning and benchmarking. In Fig. 3(a-b), the achieved validation loss and the validation accuracy are shown as a function of wall-clock time, that is measured for the baseline and the K-FAC optimizer, and estimated for

Thermodynamic K-FAC. This estimation is done by measuring the fraction of the computation time spent on the inverse, which in this case is 11%, and estimating a speedup on the matrix inversion for the dimensions considered based on numerics for the matrix inversion primitive, with assumptions detailed in the Appendix and in related work such as Refs.[37][9][11].

5.1.2. GNN on ogbg-molpcba

Another workload we consider is training a GNN on ogbg-molpcba, with our results plotted in Fig. 3(c-d). The baseline optimizer for this workload is Nesterov[45]. For the K-FAC optimizer, here the fraction of the computation time spent on the inversions is 27%, meaning a larger speedup can be unlocked, reaching an overall speedup to reach the same validation metrics than the baseline by about 50%.

5.2. Quantization experiments

To assess the effect of the input and output quantization on a real task, we ran an image classification task on a ResNet using K-FAC with input or output integer quantization. The results are plotted in Fig. 4. According to these results, the input quantization does not seem to have a large impact on accuracy. We believe that this is partly due to an effective conditioning of the matrix (similar to damping) brought on by the specific diagonally dominant quantization scheme used. The impact of output quantization on training accuracy is more pronounced as compared to input quantization. This provides guidance for how to build a potential hardware architecture for this application, highlighting the need for an output resolution of at least 8 bits while being much less sensitive to input quantization. The broader message of Fig. 4 is a general robustness to quantization that K-FAC appears to have, as even an 8-bit K-FAC optimizer is competitive against a full precision Adam optimizer. This lends hope that a moderate precision hardware architecture could be employed to accelerate K-FAC, with good performance expected.

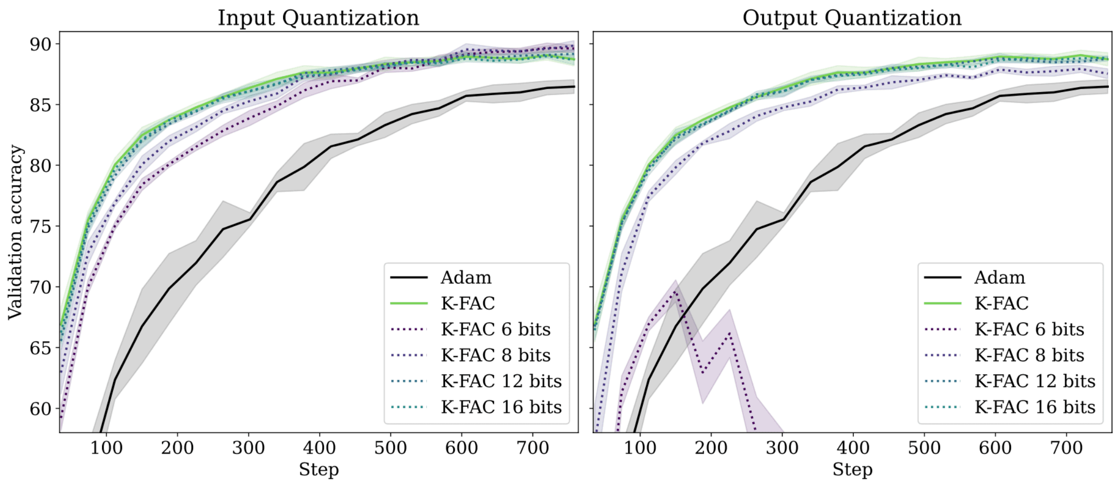

Figure 4. Effect of quantization on K-FAC training accuracy. Validation accuracy from training a ResNet on image classification with either the Adam optimizer or with the K-FAC optimizer for various levels of precision (integer 6, 8, 12 and 16 bits, and floating-point 32 bits at full precision). The left panel shows the effects of input quantization while the right panel shows the effect of output quantization. The lines correspond to the mean values over 5 runs, while the shaded areas represent one standard deviation away from the mean. Brighter colours indicate higher precision.

Moreover, we remark that error mitigation methods have been developed for thermodynamic computing that effectively boost the input precision by several bits[40]. Similar methods could likely be developed for output precision. These error mitigation methods could be incorporated into our Thermodynamic K-FAC optimizer to further enhance performance.

6. Conclusion

This work introduces a scalable approach to second-order optimization by leveraging thermodynamic hardware to accelerate the K-FAC optimizer. By offloading operations to a physical system that efficiently solves linear algebraic primitives, we achieve a significant reduction in per-iteration asymptotic runtime complexity. This enables K-FAC to approach the efficiency of first-order methods while preserving the convergence benefits of second-order optimization on AlgoPerf benchmarks.

Experimental results suggest that our method is robust to quantization noise, making it a viable candidate for low-precision analog or mixed-signal implementations. Additionally, our complexity analysis highlights a growing advantage over digital K-FAC as the network width increases.

Extensions of this approach include integrating thermodynamic solvers into large-scale deep learning pipelines and investigating alternative hardware accelerators for second-order methods. Additionally, optimizing the interaction between digital and thermodynamic components could further improve the practicality of our method for real-world training workloads. Finally, this approach could be extended to various other local approximations to the objective function, which may be accurate over a larger region. This could potentially further reduce the number of optimization iterations necessary over first and second-order methods and remains an important direction for future work.

Appendix A. Energy-based viewpoint

Consider the following energy-based viewpoint of our work. Training a machine learning model can be formulated as the task of minimizing an energy function; it is therefore a natural application for thermodynamic computing devices, where the damped stochastic dynamics of a system minimizes its physical free energy.

Second-order training methods like NGD employ a local quadratic approximation to the loss landscape and then optimize that approximate landscape[3]. Given a thermodynamic device with quadratic potential energy (e.g., the coupled harmonic oscillator hardware discussed in our work), the local approximation to the objective function can be mapped to the device's potential energy, allowing the optimization problem to be solved via the physical dissipation of energy. This highlights the deep connection between NGD and dissipative harmonic oscillator systems.

Assuming a thermodynamic device with a non-quadratic potential energy, this approach could be extended to various other local approximations to the objective function, which may be accurate over a larger region. This could potentially further reduce the number of optimization iterations necessary over first and second-order methods, and remains an important direction for future work.

Appendix B. Computational Complexities

Table 2 expands on Table 1 for the reduce and expand versions of K-FAC[15]. These versions of K-FAC extend to more general weight-sharing architectures such as CNNs and transformers and when using the generalised Gauss-Newton matrix as the curvature matrix. The complexities are more subtle due to the additional complexity added to extend to general weight-sharing architectures - in particular added dependence on the output and weight-sharing dimensions. However the cubic to quadratic runtime speedup via the thermodynamic hardware in terms of the width of the network (i.e., the number of neurons per layer) remains.

Optimizer	Runtime	Memory
SGD/Adam	$\mathcal{O}(bn^2)$	$\mathcal{O}(n^2)$
K-FAC-reduce	$\mathcal{O}(bCn(C+n+R) + n^3)$	$\mathcal{O}(bn+n^2)$
Thermodynamic K-FAC-reduce	$\mathcal{O}(bCn(C+n+R) + n^2\kappa^2)$	$\mathcal{O}(bn)$
Thermodynamic K-FAC-reduce (w/ EMA)	$\mathcal{O}(bCn(C+n+R) + n^2\kappa^2)$	$\mathcal{O}(bn+n^2)$
K-FAC-expand	$\mathcal{O}(bRCn(C+n) + n^3)$	$\mathcal{O}(bRn+n^2)$
Thermodynamic K-FAC-expand	$\mathcal{O}(bRCn(C+n) + n^2\kappa^2)$	$\mathcal{O}(bRn)$
Thermodynamic K-FAC-expand (w/ EMA)	$\mathcal{O}(bRCn(C+n) + n^2\kappa^2)$	$\mathcal{O}(bRn+n^2)$

Table 2. *Runtime and memory complexity (per layer) of optimizers considered in this paper.* Here n is the number of neurons per layer, hence there are n^2 parameters per layer, and b is the batch size. C is the output dimension, and R is the weight-sharing dimension. We assume that the Kronecker factors have condition number at most κ. Expand and reduce refer to the weight-sharing techniques described in Section 3.3 and Ref. [15].

Appendix C. Code implementation and hyperparameters

Our code implementation is based on asdl[46] with an addition of K-FAC-reduce and K-FAC-expand[15] for linear and convolutional layers. For the experiments reported in Fig 2(a), an MLP with 50 layers was used, with inputs being CIFAR-10 images and a batch size $b = 512$. For the GPT in Fig 2(b), the vocabulary size is fixed to 10^4, with a sequence length $R = 64$, a batch size $b = 64$ and 3 transformer layers. (this was necessary for the model to fit in memory on a single GPU). The GPT results in Fig 2(c) were obtained with $b = 64$, $R = 512$, 12 layers and $n = 768$.

For the K-FAC training experiments, Table 3 shows the relevant hyperparameters for the experiments we performed. All were performed with the conservative quantization method explained in Section 4.5.

Experiment	Inv. I/O resolution	Machine	Optimizer	LR	Damping	EMA decay
CIFAR-10	(varying)	1xA100	SGD	0.1	0.001	0.9999
ImageNet-ViT	12/12	4xV100	NAdamW	0.0012	0.001	0.999
OGBG-GNN	12/12	4xV100	Nesterov	10	0.005	0.999

Table 3. Hyperparameters and configurations for experiments. For the AlgoPerf experiments, we combined K-FAC with other optimizers as in Ref. [15] as it showed superior performance over the configurations we tested. We leave the exploration of why this leads to superior performance to future work.

Appendix D. Inversion time for AlgoPerf workloads in a multi-GPU setting

We profiled the time spent on matrix inversion in multi-GPU environments, where additional synchronization and communication overhead reduces the fraction of total compute time devoted to inversion. To validate this, we measured the same workloads on an NVIDIA 4×V100 system and on an NVIDIA 8×A100 system with NVLink (providing better communication bandwidth). Table 4 shows that higher communication bandwidth shifts the bottleneck back to compute, increasing the fraction of time spent on inversion. We also note that the current asdl K-FAC implementation is not heavily optimized for further offloading of inversions; thus, inversion cost may still not dominate overall runtime. Similarly, the linear-systems approach (Method 2) described in the main text should increase this computational load.

Workload	4xV100	8xA100
ImageNet-ViT	11%	16%
OGBG-GNN	27%	35%

Table 4. Total time spent on inversion for AlgoPerf workloads. Shown are the percentages for 4xV100 and 8xA100 systems. The 8xA100's improved communication bandwidth reduces overhead elsewhere, increasing the share of total runtime spent on inversion.

Appendix E. Potential hardware architecture

The thermodynamic K-FAC algorithm can be implemented via a similar hardware architecture to what is presented in Refs.[9][37][11]. The Kronecker factors can be digitally constructed and sent onto a system whose evolution is described by the differential equation (we take here A_ℓ but it is equally valid for the G_ℓ's):

$$dV = -(A_\ell + \lambda\mathbb{I})V dt - b dt + \mathcal{N}(0, 2\kappa_0 dt) \tag{24}$$

where κ_0 is the noise variance and $V = (V_1, V_2, \ldots, V_N)$ is a vector of voltages.

One may only consider the activations a_ℓ and gradients g_ℓ that enter the construction of A_ℓ and B_ℓ respectively and send them directly onto similar hardware. This alternative implementation is comprised of two arrays of resistors of size $(b, N), (b, N)$ for encoding a_ℓ and $a_\ell^\top$, respectively. These arrays of resistors enable one to implement the following differential equation in hardware:

$$dV = -(a_\ell a_\ell^\top + \lambda\mathbb{I})V dt - b dt + \mathcal{N}(0, 2\kappa_0 dt). \tag{25}$$

This system may be implemented with the circuit diagram shown in Fig. 5, where $N = 3$, $b = 2$. We assume the capacitors all have the same value C, and the resistors with no labels all have the same value R_0. By Kirchhoff's current law, we obtain the equation of motion for the voltage vector $V = (V_1, V_2, V_3)$ as:

$$C\dot{V} = -(\mathcal{G}V + \lambda V - R^{-1}V_{in})$$

with $V_{in} = (V_{in1}, V_{in2}, V_{in3})$, $R = \text{diag}(R_1, R_2, R_3)$, $\lambda = \text{diag}(1/R_{\lambda_1}, 1/R_{\lambda_2}, 1/R_{\lambda_3})$. We have

$$\mathcal{G} = a_\ell a_\ell^\top = \begin{pmatrix} \frac{1}{R^a_{11}} & \frac{1}{R^a_{12}} \\ \frac{1}{R^a_{21}} & \frac{1}{R^a_{22}} \\ \frac{1}{R^a_{31}} & \frac{1}{R^a_{32}} \end{pmatrix} \begin{pmatrix} \frac{1}{R^a_{11}} & \frac{1}{R^a_{21}} & \frac{1}{R^a_{31}} \\ \frac{1}{R^a_{12}} & \frac{1}{R^a_{22}} & \frac{1}{R^a_{32}} \end{pmatrix} \frac{1}{R_0^2}, \tag{26}$$

where we therefore have a set of resistors R^a representing the a tensor and its transpose. At steady state the average voltage vector corresponds to the natural gradient estimate, since for $\dot{V} = 0$, the average voltage vector is $\langle V \rangle = \mathcal{G}^{-1} R^{-1} V_{in}$, which corresponds to the solution of the linear system $Ax = b$ with $A = \mathcal{G}$, $x = V$, $b = R^{-1} V_{in}$.

The resistor values R^a_{ij} can directly be calculated as $1/a_{ij}$ (or $1/a_{ji}$ for the transpose), and the total number of resistors in the circuit is $2bN$ (12 in the schematic shown). One may operate the thermodynamic linear solver by setting the voltage values V_{in} to the rows of the gradient matrix $(D\Theta_\ell)$ with a digital-to-analog converter, and set the values of the programmable resistors thanks to a digital controller. The time for the system to relax to equlibrium (and therefore for the linear system to be solved) is:

$$\tau = \frac{RC}{\alpha_{\min}}$$

where R is a resistance scale (which means that all resistances R_{ij} are a multiple of this), C is the capacitance (assuming all the capacitances are the same), and $\alpha_{\min}$ is the smallest eigenvalue of the (unitless) $\mathcal{G}$ matrix. After this time, all the modes of the system will have relaxed, which may be too conservative (for example, in the case where there is only one slow mode, and all other modes are fast). With regularization, $\alpha_{\min}$ is lower-bounded by the regularization factor λ (which is between 10^{-3} and 0.5 for all experiments). For timing purposes, RC is kept as the relaxation time, because of the problem-dependence of $\alpha_{\min}$. The estimated speedups on the matrix inverse primitive (see also Refs.[9][37]) are based on the following assumptions:

- 16 bits of precision (less bits of precision will lead to weaker encoding requirements, therefore a larger speedup).
- A digital transfer speed of 50 Gb/s.
- $R = 10^3\,\Omega$, $C = 1\text{nF}$, which means $RC = 1\mu\text{s}$ is the characteristic timescale of the system.

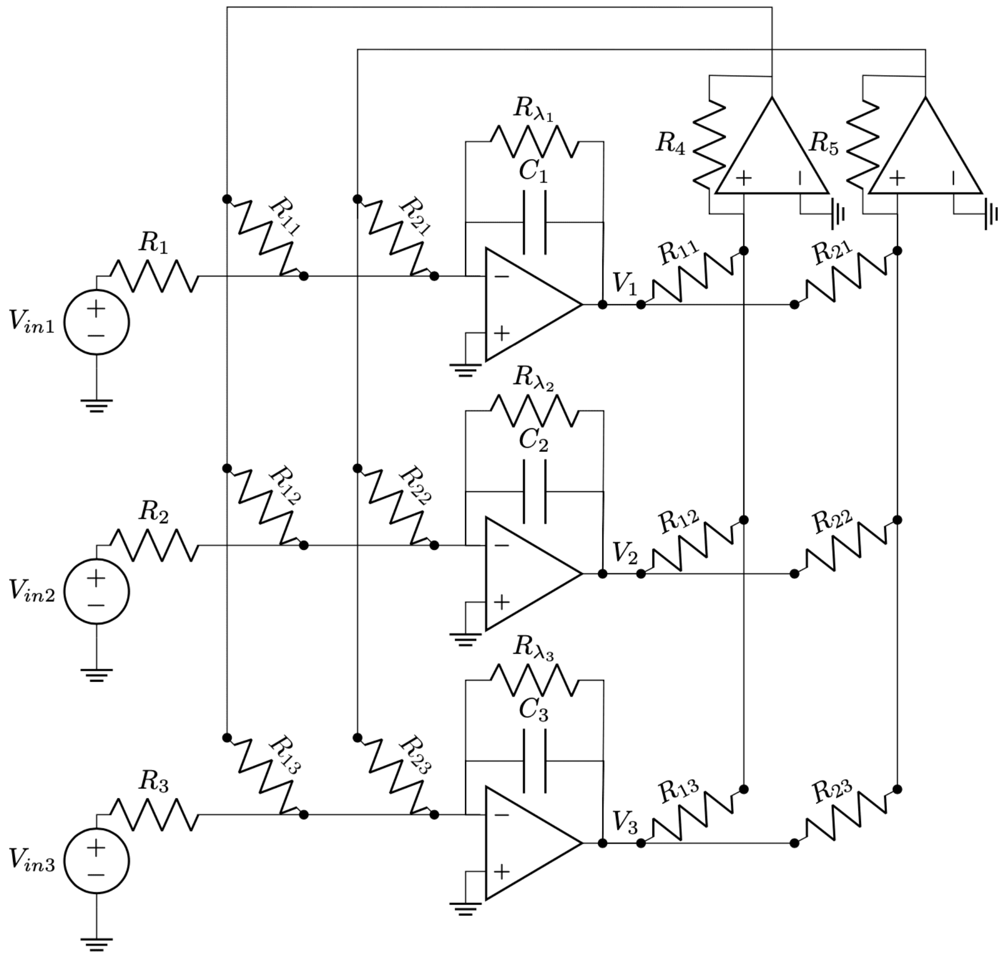

Figure 5. Circuit diagram of a possible implementation of the thermodynamic solver.

Acknowledgements

Thanks to the Advanced Research + Invention Agency's (ARIA) Scaling Compute programme for funding this work.

Footnotes

[2] This time complexity emerges from a worst-case upper bound, and there is some evidence that the quadratic dependence on κ can be improved in the average-case, which will be published in forthcoming work.

References

1. a, b, c, d *Amari S. (1998). "Natural gradient works efficiently in learning". Neural computation.* ***10*** *(2): 251–276. http://cognet.mit.edu/journal/10.1162/089976698300017746.*
2. ^ *Zhang G, Martens J, Grosse RB (2019). "Fast convergence of natural gradient descent for over-parameterized neural networks". Advances in Neural Information Processing Systems.* ***32****.*
3. a, b, c *Martens J (2020). "New insights and perspectives on the natural gradient method". The Journal of Machine Learning Research.* ***21*** *(1): 5776–5851. Available from: https://arxiv.org/abs/1412.1193.*
4. ^ *Kakade SM (2001). "A natural policy gradient". Advances in neural information processing systems.* ***14****.*
5. ^ *Grondman I, Busoniu L, Lopes GA, Babuska R (2012). "A survey of actor-critic reinforcement learning: Standard and natural policy gradients". IEEE Transactions on Systems, Man, and Cybernetics, part C (applications and reviews).* ***42*** *(6): 1291–1307.*
6. ^ *Kingma DP, Ba J (2015). "Adam: A Method for Stochastic Optimization." In: Proceedings of the 3rd International Conference on Learning Representations (ICLR). Available from: http://arxiv.org/abs/1412.6980.*
7. ^ *Conte T, DeBenedictis E, Ganesh N, Hylton T, Strachan JP, Williams RS, Alemi A, Altenberg L, Crooks GE, Crutchfield J, et al. Thermodynamic computing. arXiv preprint arXiv:1911.01968. 2019. Available from: https://arxiv.org/abs/1911.01968.*
8. a, b, c *Coles PJ, Szczepanski C, Melanson D, Donatella K, Martinez AJ, Sbahi F. "Thermodynamic AI and the fluctuation frontier." In: 2023 IEEE International Conference on Rebooting Computing (ICRC). IEEE; 2023. p. 1–10. doi:10.1109/ICRC60800.2023.10386858.*
9. a, b, c, d, e, f, g, h, i *Aifer M, Donatella K, Gordon MH, Duffield S, Ahle T, Simpson D, Crooks G, Coles PJ (2024). "Thermodynamic linear algebra". npj Unconventional Computing.* ***1*** *(1): 13. doi:10.1038/s44335-024-00014-0.*
10. ^ *Duffield S, Aifer M, Crooks G, Ahle T, Coles PJ (2023). "Thermodynamic Matrix Exponentials and Thermodynamic Parallelism". arXiv preprint arXiv:2311.12759. Available from: https://arxiv.org/abs/2311.12759.*
11. a, b, c, d, e, f, g *Donatella K, Duffield S, Aifer M, Melanson D, Crooks G, Coles PJ (2024). "Thermodynamic Natural Gradient Descent". arXiv preprint arXiv:2405.13817. Available from: https://arxiv.org/abs/2405.13817.*

12. ^Bottou L, Curtis FE, Nocedal J (2018). "Optimization methods for large-scale machine learning". SIAM review. 60 (2): 223–311. doi:10.1137/16M1080173.

13. a, b, c, d, e, fMartens J, Grosse R. "Optimizing neural networks with kronecker-factored approximate curvature." In: International conference on machine learning. PMLR; 2015. p. 2408-2417. Available from: https://proceedings.mlr.press/v37/martens15.html.

14. a, b, cMartens J, Ba J, Johnson M. Kronecker-factored curvature approximations for recurrent neural networks. In: International Conference on Learning Representations; 2018. Available from: https://openreview.net/pdf?id=HyMTkQZAb.

15. a, b, c, d, e, f, g, h, i, j, k, l, m, n, oEschenhagen R, Immer A, Turner R, Schneider F, Hennig P. Kronecker-factored approximate curvature for modern neural network architectures. In: Advances in Neural Information Processing Systems. 2023;36. Available from: https://proceedings.neurips.cc/paper_files/paper/2023/file/6a6679e3d5b9f7d5f09cdb79a5fc3fd8-Paper-Conference.pdf.

16. ^Ren Y, Goldfarb D (2019). "Efficient subsampled gauss-newton and natural gradient methods for training neural networks". arXiv preprint arXiv:1906.02353. Available from: https://arxiv.org/abs/1906.02353.

17. ^Gargiani M, Zanelli A, Diehl M, Hutter F (2020). "On the promise of the stochastic generalized Gauss-Newton method for training DNNs". arXiv preprint arXiv:2006.02409. Available from: https://arxiv.org/abs/2006.02409.

18. ^Zhang G, Li L, Nado Z, Martens J, Sachdeva S, Dahl G, Shallue C, Grosse RB. "Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model." Advances in neural information processing systems. 32, 2019.

19. a, bSun Z, Pedretti G, Ambrosi E, Bricalli A, Wang W, Ielmini D (2019). "Solving matrix equations in one step with cross-point resistive arrays". Proceedings of the National Academy of Sciences. **116** (10): 4123–4128.

20. ^Sun Z, Pedretti G, Mannocci P, Ambrosi E, Bricalli A, Ielmini D (2020). "Time complexity of in-memory solution of linear systems". IEEE Transactions on Electron Devices. **67** (7): 2945–2951.

21. ^Hylton T (2020). "Thermodynamic neural network". Entropy. 22 (3): 256. doi:10.3390/e22030256. https://www.mdpi.com/1099-4300/22/3/256.

22. ^Ganesh N. A thermodynamic treatment of intelligent systems. In: 2017 IEEE International Conference on Rebooting Computing (ICRC). 2017. p. 1-4. doi:10.1109/ICRC.2017.8123676.

23. ^Lipka-Bartosik P, Perarnau-Llobet M, Brunner N (2024). "Thermodynamic computing via autonomo us quantum thermal machines". Science Advances. **10** (36): eadm8792.

24. ^Whitelam S, Casert C (2024). "Thermodynamic computing out of equilibrium". arXiv preprint arXiv:24 12.17183. Available from: https://arxiv.org/abs/2412.17183.

25. ^Aifer M, Duffield S, Donatella K, Melanson D, Klett P, Belateche Z, Crooks G, Martinez AJ, Coles PJ (202 4). "Thermodynamic Bayesian Inference". arXiv preprint arXiv:2410.01793. Available from: https://arxi v.org/abs/2410.01793.

26. ^Bartosik PL, Donatella K, Aifer M, Melanson D, Perarnau-Llobet M, Brunner N, Coles PJ (2024). "Ther modynamic Algorithms for Quadratic Programming". arXiv preprint arXiv:2411.14224. Available from: https://arxiv.org/abs/2411.14224.

27. ^Aadit NA, Grimaldi A, Carpentieri M, Theogarajan L, Martinis JM, Finocchio G, Camsari KY (2022). "M assively parallel probabilistic computing with sparse Ising machines". Nat. Electron.. **5** (7): 460–468. do i:10.1038/s41928-022-00774-2.

28. ^Kaiser J, Datta S, Behin-Aein B. Life is probabilistic—why should all our computers be deterministic? C omputing with p-bits: Ising solvers and beyond. In: 2022 International Electron Devices Meeting (IED M). IEEE; 2022. p. 21–4.

29. ^Frank MP, Brocato RW, Tierney BD, Missert NA, Hsia AH. "Reversible computing with fast, fully static, f ully adiabatic CMOS." In: 2020 International Conference on Rebooting Computing (ICRC). IEEE; 2020. p. 1-8.

30. ^Kim S, Gokmen T, Lee HM, Haensch WE. "Analog CMOS-based resistive processing unit for deep neura l network training." In: 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MW SCAS). IEEE; 2017. p. 422-425. Available from: https://ieeexplore.ieee.org/abstract/document/8052950.

31. ^Ambrogio S, Narayanan P, Tsai H, Shelby RM, Boybat I, Di Nolfo C, Sidler S, Giordano M, Bodini M, Far inha NCP, et al. Equivalent-accuracy accelerated neural-network training using analogue memory. Nat ure. **558** (7708): 60–67, 2018. https://www.nature.com/articles/s41586-018-0180-5.

32. ^Cristiano G, Giordano M, Ambrogio S, Romero LP, Cheng C, Narayanan P, Tsai H, Shelby RM, Burr GW (2018). "Perspective on training fully connected networks with resistive memories: Device requirements for multiple conductances of varying significance". J. Appl. Phys. **124** (15). doi:10.1063/1.5042462.

33. ^Aguirre F, Sebastian A, Le Gallo M, Song W, Wang T, Yang JJ, Lu W, Chang MF, Ielmini D, Yang Y, et al. Hardware implementation of memristor-based artificial neural networks. Nature Communications. **15** (1): 1974, 2024. https://www.nature.com/articles/s41467-024-45670-9.

34. ^Martens J, et al. Deep learning via hessian-free optimization. In: ICML. 2010; 27:735-742. Available from: https://www.cs.toronto.edu/~asamir/cifar/HFO_James.pdf.

35. ^Izadi MR, Fang Y, Stevenson R, Lin L. Optimization of graph neural networks with natural gradient descent. In: 2020 IEEE international conference on big data (big data). IEEE; 2020. p. 171–179.

36. ^Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017). "Attention is all you need". Advances in neural information processing systems. 30. Available from: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

37. a, b, c, d, eMelanson D, Khater M, Aifer M, Donatella K, Gordon MH, Ahle T, Crooks G, Martinez AJ, Sbahi F, Coles PJ (2023). "Thermodynamic Computing System for AI Applications". arXiv preprint arXiv:2312.04836. Available from: https://arxiv.org/abs/2312.04836.

38. a, bKarpathy A (2022). "NanoGPT". GitHub repository. Available from: https://github.com/karpathy/nanoGPT.

39. a, bAnil R, Gupta V, Koren T, Regan K, Singer Y (2020). "Scalable second order optimization for deep learning". arXiv preprint arXiv:2002.09018.

40. a, bAifer M, Melanson D, Donatella K, Crooks G, Ahle T, Coles PJ (2024). "Error Mitigation for Thermodynamic Computing". arXiv. Available from: https://arxiv.org/abs/2401.16231.

41. a, bFunk C, Noack B, Hanebeck UD (2021). "Conservative Quantization of Covariance Matrices with Applications to Decentralized Information Fusion". Sensors. 21 (9): 3059. doi:10.3390/s21093059. PMID 33924751.

42. ^Higham NJ, Pranesh S (2021). "Exploiting Lower Precision Arithmetic in Solving Symmetric Positive Definite Linear Systems and Least Squares Problems". SIAM Journal on Scientific Computing. 43 (1): A258–A277. doi:10.1137/19M1298263.

43. ^Dahl GE, Schneider F, Nado Z, Agarwal N, Sastry CS, Hennig P, Medapati S, Eschenhagen R, Kasimbeg P, Suo D, et al. Benchmarking neural network training algorithms. arXiv preprint arXiv:2306.07179. 2023.

44. ^Loshchilov I, Hutter F (2017). "Decoupled weight decay regularization". arXiv preprint arXiv:1711.05101. Available from: https://arxiv.org/abs/1711.05101.

45. ^Sutskever I, Martens J, Dahl G, Hinton G (2013). "On the importance of initialization and momentum in deep learning." In: International conference on machine learning. PMLR. pp. 1139–1147.

46. ^Osawa K, Ishikawa S, Yokota R, Li S, Hoefler T (2023). "Asdl: A unified interface for gradient preconditioning in pytorch". arXiv preprint arXiv:2305.04684. arXiv:2305.04684.

Declarations

Funding: Thanks to the Advanced Research + Invention Agency's (ARIA) Scaling Compute programme for funding this work.

Potential competing interests: No potential competing interests to declare.