ISAAC Newton: Input-based Approximate Curvature for Newton's Method

Anonymous Author(s)
Affiliation
Address
email

Abstract

We present ISAAC (Input-baSed ApproximAte Curvature), a novel method that conditions the gradient using selected second-order information and has an asymptotically vanishing computational overhead, assuming a batch size smaller than the number of neurons. We show that it is possible to compute a good conditioner based on only the input to a respective layer without a substantial computational overhead. The proposed method allows effective training even in small-batch stochastic regimes, which makes it competitive to first-order as well as quasi-Newton methods.

1 Introduction

While second-order optimization methods are traditionally much less explored than first-order methods in large-scale machine learning (ML) applications due to their memory requirements and prohibitive computational cost per iteration, they have recently become more popular in ML mainly due to their fast convergence properties when compared to first-order methods [1]. The expensive computation of an inverse Hessian (also known as pre-conditioning matrix) in the Newton step has also been tackled via estimating the curvature from the change in gradients. Loosely speaking, these algorithms are known as *quasi-Newton methods* and a comprehensive treatment can be found in the textbook [2]. In addition, various new approximations to the pre-conditioning matrix have been proposed in the recent literature [3]–[6]. From a theoretical perspective, second-order optimization methods are not nearly as well understood as first-order methods. It is an active research direction to fill this gap [7], [8].

Motivated by the task of training neural networks, and the observation that invoking local curvature information associated with neural network objective functions can achieve much faster progress per iteration than standard first-order methods [9]–[11], several methods have been proposed. One of these methods, that received significant attention, is known as *Kronecker-factored Approximate Curvature (K-FAC)* [12], whose main ingredient is a sophisticated approximation to the generalized Gauss-Newton matrix and the Fisher information matrix quantifying the curvature of the underlying neural network objective function, which then can be inverted efficiently.

Inspired by the K-FAC approximation and the Tikhonov regularization of the Newton method, we introduce a novel two parameter regularized Kronecker-factorized Newton update step. The proposed scheme disentangles the classical Tikhonov regularization and allows us to condition the gradient using selected second-order information and has an asymptotically vanishing computational overhead. While this property makes the presented method highly attractive from the computational complexity perspective, we show that its achieved empirical performance on complicated high-dimensional Machine Learning problems remains comparable to existing state-of-the-art methods.

The contributions of this paper can be summarized as follows: (i) we propose a novel two parameter regularized K-FAC approximated Gauss-Newton update step; (ii) we show that asymptotically—as

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

both regularization parameters vanish—our method recovers the classical K-FAC scheme and in the opposite setting—as both regularization parameters grow—our method asymptotically reduces to classical gradient descent; (iii) we prove that for an arbitrary pair of regularization parameters, the proposed update direction is always a direction of decreasing loss; (iv) in the limit, as one regularization parameter grows, we obtain an efficient and effective conditioning of the gradient with an asymptotically vanishing overhead; (v) we empirically analyze the presented method and find that our efficient conditioning method maintains the performance of its more expensive counterpart; (vi) we demonstrate the effectiveness of the presented method in the setting of small-batch stochastic regimes and observe that it is competitive to first-order as well as quasi-Newton methods.

2 Preliminaries

In this section, we review aspects of second-order optimization, with a focus on generalized Gauss-Newton methods. In combination with Kronecker factorization, this leads us to a new regularized update scheme. We consider the training of an L-layer neural network $f(x;\theta)$ defined recursively as

$$z_i \leftarrow a_{i-1}W^{(i)} \quad \text{(pre-activations)}, \qquad a_i \leftarrow \phi(z_i) \quad \text{(activations)}, \tag{1}$$

where $a_0 = x$ is the vector of inputs and $a_L = f(x;\theta)$ is the vector of outputs. Unless noted otherwise, we assume these vectors to be row vectors (i.e., in $\mathbb{R}^{1\times n}$) as this allows for a direct extension to the (batch) vectorized case (i.e., in $\mathbb{R}^{b\times n}$) introduced later. For any layer i, let $W^{(i)} \in \mathbb{R}^{d_{i-1}\times d_i}$ be a weight matrix and let ϕ be an element-wise nonlinear function. We consider a convex loss function $\mathcal{L}(y, y')$ that measures the discrepancy between y and y'. The training optimization problem is then

$$\arg\min_{\theta} \mathbb{E}_{x,y}\left[\mathcal{L}(f(x;\theta), y)\right], \tag{2}$$

where $\theta = \left[\theta^{(1)}, \ldots, \theta^{(L)}\right]$ with $\theta^{(i)} = \mathrm{vec}(W^{(i)})$.

The classical Newton method for solving (2) is expressed as the update rule

$$\theta' = \theta - \eta\, \mathbf{H}_\theta^{-1} \nabla_\theta \mathcal{L}(f(x;\theta), y), \tag{3}$$

where $\eta > 0$ denotes the learning rate and $\mathbf{H}_\theta$ is the Hessian corresponding to the objective function in (2). The stability and efficiency of an estimation problem solved via the Newton method can be improved by adding a Tikhonov regularization term [13] leading to a regularized Newton method

$$\theta' = \theta - \eta\, (\mathbf{H}_\theta + \lambda \mathbf{I})^{-1} \nabla_\theta \mathcal{L}(f(x;\theta), y), \tag{4}$$

where $\lambda > 0$ is the so-called Tikhonov regularization parameter. It is well-known [14], [15], that under the assumption of approximating the model f with its first-order Taylor expansion, the Hessian corresponds with the so-called generalized Gauss-Newton (GGN) matrix $\mathbf{G}_\theta$, and hence (4) can be expressed as

$$\theta' = \theta - \eta\, (\mathbf{G}_\theta + \lambda \mathbf{I})^{-1} \nabla_\theta \mathcal{L}(f(x;\theta), y). \tag{5}$$

A major practical limitation of (5) is the computation of the inverse term. A method that alleviates this difficulty is known as Kronecker-Factored Approximate Curvature (K-FAC) [12] which approximates the block-diagonal (i.e., layer-wise) empirical Hessian or GGN matrix. Inspired by K-FAC, there have been other works discussing approximations of $\mathbf{G}_\theta$ and its inverse [15]. In the following, we discuss a popular approach that allows for (moderately) efficient computation.

The generalized Gauss-Newton matrix $\mathbf{G}_\theta$ is defined as

$$\mathbf{G}_\theta = \mathbb{E}\left[(\mathbf{J}_\theta f(x;\theta))^\top \nabla_f^2 \mathcal{L}(f(x;\theta), y)\, \mathbf{J}_\theta f(x;\theta)\right], \tag{6}$$

where $\mathbf{J}$ and $\mathbf{H}$ denote the Jacobian and Hessian matrices, respectively. Correspondingly, the diagonal block of $\mathbf{G}_\theta$ corresponding to the weights of the ith layer $W^{(i)}$ is

$$\mathbf{G}_{W^{(i)}} = \mathbb{E}\left[(\mathbf{J}_{W^{(i)}} f(x;\theta))^\top \nabla_f^2 \mathcal{L}(f(x;\theta), y)\, \mathbf{J}_{W^{(i)}} f(x;\theta)\right].$$

According to the backpropagation rule $\mathbf{J}_{\theta^{(i)}} f(x;\theta) = \mathbf{J}_{z_i} f(x;\theta)\, a_{i-1}$, $a^\top b = a \otimes b$, and the mixed-product property, we can rewrite $\mathbf{G}_{W^{(i)}}$ as

$$\mathbf{G}_{W^{(i)}} = \mathbb{E}\Big[\left((\mathbf{J}_{z_i} f(x;\theta)\, a_{i-1})^\top (\nabla_f^2 \mathcal{L}(f(x;\theta), y))^{1/2}\right)\left((\nabla_f^2 \mathcal{L}(f(x;\theta), y))^{1/2}\, \mathbf{J}_{z_i} f(x;\theta)\, a_{i-1}\right)\Big] \tag{7}$$

$$= \mathbb{E}\left[(\bar{g}^\top a_{i-1})^\top (\bar{g}^\top a_{i-1})\right] = \mathbb{E}\left[(\bar{g} \otimes a_{i-1})^\top (\bar{g} \otimes a_{i-1})\right] = \mathbb{E}\left[(\bar{g}^\top \bar{g}) \otimes (a_{i-1}^\top \otimes a_{i-1})\right], \tag{8}$$

where

$$\bar{g} = (\mathbf{J}_{z_i} f(x;\theta))^\top\, (\nabla_f^2 \mathcal{L}(f(x;\theta), y))^{1/2}. \tag{9}$$

Remark 1 (Monte-Carlo Low-Rank Approximation for $\bar{g}^\top \bar{g}$)**.** *As $\bar{g}$ is a matrix of shape $m \times d_i$ where m is the dimension of the output of f, $\bar{g}$ is generally expensive to compute. Therefore, [12] use a low-rank Monte-Carlo approximation to estimate $\mathbf{H}_f \mathcal{L}(f(x;\theta), y)$ and thereby $\bar{g}^\top \bar{g}$. For this, we need to use the distribution underlying the probabilistic model of our loss $\mathcal{L}$ (e.g., Gaussian for MSE loss, or a categorical distribution for cross entropy). Specifically, by sampling from this distribution $p_f(x)$ defined by the network output $f(x;\theta)$, we can get an estimator of $\mathbf{H}_f \mathcal{L}(f(x;\theta), y)$ via the identity*

$$\mathbf{H}_f \mathcal{L}(f(x;\theta), y) = \mathbb{E}_{\hat{y} \sim p_f(x)} \left[\nabla_f \mathcal{L}(f(x;\theta), \hat{y})^\top \nabla_f \mathcal{L}(f(x;\theta), \hat{y}) \right]. \tag{10}$$

An extensive reference for this (as well as alternatives) can be found in Appendix A.2 of Dangel et al. [15]. The respective rank-1 approximation (denoted by $\triangleq$) of $\mathbf{H}_f \mathcal{L}(f(x;\theta))$ is

$$\mathbf{H}_f \mathcal{L}(f(x;\theta), y) \triangleq \nabla_f \mathcal{L}(f(x;\theta), \hat{y})^\top \nabla_f \mathcal{L}(f(x;\theta), \hat{y}),$$

where $\hat{y} \sim p_f(x)$. Respectively, we can estimate $\bar{g}^\top \bar{g}$ using this rank-1 approximation with

$$\bar{g} \triangleq \left(\mathbf{J}_{z_i} f(x;\theta)\right)^\top \nabla_f \mathcal{L}(f(x;\theta), \hat{y}) = \nabla_{z_i} \mathcal{L}(f(x;\theta), \hat{y}). \tag{11}$$

In analogy to $\bar{g}$, we introduce the gradient of training objective with respect to pre-activations z_i as

$$\mathrm{g}_i = \left(\mathbf{J}_{z_i} f(x;\theta)\right)^\top \nabla_f \mathcal{L}(f(x;\theta), y) = \nabla_{z_i} \mathcal{L}(f(x;\theta), y). \tag{12}$$

In other words, for a given layer, let $\mathrm{g} \in \mathbb{R}^{1 \times d_i}$ denote the gradient of the loss between an output and the ground truth and let $\bar{g} \in \mathbb{R}^{m \times d_i}$ denote the derivative of the network f times the square root of the Hessian of the loss function (which may be approximated according to Remark 1), each of them with respect to the output z_i of the given layer i. Note that $\bar{g}$ is not equal to g and that they require one backpropagation pass each (or potentially many for the case of $\bar{g}$). This makes computing $\bar{g}$ costly.

Applying the K-FAC [12] approximation to (8) the expectation of Kronecker products can be approximated as the Kronecker product of expectations as

$$\mathbf{G} = \mathbb{E}((\bar{g}^\top \bar{g}) \otimes (\mathrm{a}^\top \mathrm{a})) \approx \mathbb{E}(\bar{g}^\top \bar{g}) \otimes \mathbb{E}(\mathrm{a}^\top \mathrm{a}), \tag{13}$$

where, for clarity, we drop the index of a_{i-1} in (8) and denote it with a; similarly we denote $\mathbf{G}_{W^{(i)}}$ as $\mathbf{G}$. While the expectation of Kronecker products is generally not equal to the Kronecker product of expectations, this K-FAC approximation (13) has been shown to be fairly accurate in practice and to preserve the "coarse structure" of the GGN matrix [12]. The K-FAC decomposition in (13) is convenient as the Kronecker product has the favorable property that for two matrices A, B the identity $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ which significantly simplifies the computation of an inverse.

In practice, $\mathbb{E}(\bar{g}^\top \bar{g})$ and $\mathbb{E}(\mathrm{a}^\top \mathrm{a})$ can be computed by averaging over a batch of size b as

$$\mathbb{E}(\bar{g}^\top \bar{g}) \simeq \bar{\mathbf{g}}^\top \bar{\mathbf{g}} / b, \qquad \mathbb{E}(\mathrm{a}^\top \mathrm{a}) \simeq \mathbf{a}^\top \mathbf{a} / b, \tag{14}$$

where we denote batches of g, $\bar{g}$ and a, as $\mathbf{g} \in \mathbb{R}^{b \times d_i}$, $\bar{\mathbf{g}} \in \mathbb{R}^{rb \times d_i}$ and $\mathbf{a} \in \mathbb{R}^{b \times d_{i-1}}$, where our layer has d_{i-1} inputs, d_i outputs, b is the batch size, and r is either the number of outputs m or the rank of an approximation according to Remark 1. Correspondingly, the K-FAC approximation of the GGN matrix and its inverse are concisely expressed as

$$\mathbf{G} \approx (\bar{\mathbf{g}}^\top \bar{\mathbf{g}}) \otimes (\mathbf{a}^\top \mathbf{a}) / b^2 \qquad \mathbf{G}^{-1} \approx \left(\bar{\mathbf{g}}^\top \bar{\mathbf{g}}\right)^{-1} \otimes \left(\mathbf{a}^\top \mathbf{a}\right)^{-1} \cdot b^2. \tag{15}$$

Equipped with the standard terminology and setting, we now introduce the novel, regularized update step. First, inspired by the K-FAC approximation (13), the Tikhonov regularized Gauss-Newton method (5) can be approximated by

$$\theta^{(i)\prime} = \theta^{(i)} - \eta (\bar{\mathbf{g}}^\top \bar{\mathbf{g}} / b + \lambda \mathbf{I})^{-1} \otimes (\mathbf{a}^\top \mathbf{a} / b + \lambda \mathbf{I})^{-1} \nabla_{\theta^{(i)}} \mathcal{L}(f(x;\theta)), \tag{16}$$

with regularization parameter $\lambda > 0$. A key observation, which is motivated by the structure of the above update, is to disentangle the two occurrences of λ into two independent regularization parameters $\lambda_{\mathbf{g}}, \lambda_{\mathbf{a}} > 0$. By defining the Kronecker-factorized Gauss-Newton update step as

$$\boldsymbol{\zeta} = \lambda_{\mathbf{g}} \lambda_{\mathbf{a}} (\bar{\mathbf{g}}^\top \bar{\mathbf{g}} / b + \lambda_{\mathbf{g}} \mathbf{I})^{-1} \otimes (\mathbf{a}^\top \mathbf{a} / b + \lambda_{\mathbf{a}} \mathbf{I})^{-1} \nabla_{\theta^{(i)}} \mathcal{L}(f(x;\theta)), \tag{17}$$

we obtain the concise update equation

$$\theta^{(i)\prime} = \theta^{(i)} - \eta^* \boldsymbol{\zeta}. \tag{18}$$

This update (18) is equivalent to update (16) when in the case of $\eta^* = \frac{\eta}{\lambda_{\mathbf{g}}\lambda_{\mathbf{a}}}$ and $\lambda = \lambda_{\mathbf{g}} = \lambda_{\mathbf{a}}$. This equivalence does not restrict $\eta^*, \lambda_{\mathbf{g}}, \lambda_{\mathbf{a}}$ in any way, and changing $\lambda_{\mathbf{g}}$ or $\lambda_{\mathbf{a}}$ does not mean that we change our learning rate or step size η^*. Parameterizing $\boldsymbol{\zeta}$ in (17) with the multiplicative terms $\lambda_{\mathbf{g}}\lambda_{\mathbf{a}}$ makes the formulation more convenient for analysis.

In this paper, we investigate the theoretical and empirical properties of the iterative update rule (18) and in particular show how the regularization parameters $\lambda_{\mathbf{g}}, \lambda_{\mathbf{a}}$ affect the Kronecker-factorized Gauss-Newton update step $\boldsymbol{\zeta}$. When analyzing the Kronecker-factorized Gauss-Newton update step $\boldsymbol{\zeta}$, a particularly useful tool is the vector product identity,

$$\left(\left(\bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}\right)^{-1} \otimes \left(\mathbf{a}^\top \mathbf{a}\right)^{-1}\right) \operatorname{vec}(\mathbf{g}^\top \mathbf{a}) = \operatorname{vec}\left(\left(\bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}\right)^{-1} \mathbf{g}^\top \mathbf{a} \left(\mathbf{a}^\top \mathbf{a}\right)^{-1}\right), \tag{19}$$

where the gradient with respect to the weight matrix is $\mathbf{g}^\top \mathbf{a}$.

3 Theoretical Guarantees

In this section, we investigate the theoretical properties of the Kronecker-factorized Gauss-Newton update direction $\boldsymbol{\zeta}$ as defined in (17). We recall that $\boldsymbol{\zeta}$ introduces a Tikonov regularization, as it is commonly done in implementations of second order-based methods. Not surprisingly, we show that by decreasing the regularization parameters $\lambda_{\mathbf{g}}, \lambda_{\mathbf{a}}$ the update rule (18) collapses (in the limit) to the classical Gauss-Newton method, and hence in the regime of small $\lambda_{\mathbf{g}}, \lambda_{\mathbf{a}}$ the variable $\boldsymbol{\zeta}$ describes the Gauss-Newton direction. Moreover, by increasing the regularization strength, we converge (in the limit) to the conventional gradient descent update step.

The key observation is that, as we disentangle the regularization of the two Kronecker factors $\bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}$ and $\mathbf{a}^\top \mathbf{a}$, and consider the setting where only one regularizer is large ($\lambda_{\mathbf{g}} \to \infty$ to be precise), we obtain an update direction that can be computed highly efficiently. We show that this setting describes an approximated Gauss-Newton update scheme, whose superior numerical performance is then empirically demonstrated in Section 4.

Theorem 1 (Properties of $\boldsymbol{\zeta}$)**.** *The K-FAC based update step $\boldsymbol{\zeta}$ as defined in* (17) *can be expressed as*

$$\boldsymbol{\zeta} = \left(\mathbf{I}_m - \frac{1}{b\lambda_{\mathbf{g}}} \bar{\boldsymbol{g}}^\top \left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{g}}} \bar{\boldsymbol{g}}\bar{\boldsymbol{g}}^\top\right)^{-1} \bar{\boldsymbol{g}}\right) \cdot \mathbf{g}^\top \cdot \left(\mathbf{I}_b - \frac{1}{b\lambda_{\mathbf{a}}} \mathbf{a}\mathbf{a}^\top \left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{a}}} \mathbf{a}\mathbf{a}^\top\right)^{-1}\right) \cdot \mathbf{a}\,. \tag{20}$$

Moreover, $\boldsymbol{\zeta}$ admits the following asymptotic properties:

(i) In the limit of $\lambda_{\mathbf{g}}, \lambda_{\mathbf{a}} \to 0$, $\frac{1}{\lambda_{\mathbf{g}}\lambda_{\mathbf{a}}}\boldsymbol{\zeta}$ is the K-FAC approximation of the Gauss-Newton step, i.e., $\lim_{\lambda_{\mathbf{g}}, \lambda_{\mathbf{a}} \to 0} \frac{1}{\lambda_{\mathbf{g}}\lambda_{\mathbf{a}}}\boldsymbol{\zeta} \approx \mathbf{G}^{-1} \nabla_{\theta^{(i)}} \mathcal{L}(f(x;\theta))$*, where $\approx$ denotes the K-FAC approximation* (15)*.*

(ii) In the limit of $\lambda_{\mathbf{g}}, \lambda_{\mathbf{a}} \to \infty$, $\boldsymbol{\zeta}$ is the gradient, i.e., $\lim_{\lambda_{\mathbf{g}}, \lambda_{\mathbf{a}} \to \infty} \boldsymbol{\zeta} = \nabla_{\theta^{(i)}} \mathcal{L}(f(x;\theta))$*.*

The Proof is deferred to the Supplementary Material.

We want to show that $\boldsymbol{\zeta}$ is well-defined and points in the correct direction, not only for $\lambda_{\mathbf{g}}$ and $\lambda_{\mathbf{a}}$ numerically close to zero because we want to explore the full spectrum of settings for $\lambda_{\mathbf{g}}$ and $\lambda_{\mathbf{a}}$. Thus, we prove that $\boldsymbol{\zeta}$ is a direction of increasing loss, independent of the choices of $\lambda_{\mathbf{g}}$ and $\lambda_{\mathbf{a}}$.

Theorem 2 (Correctness of $\boldsymbol{\zeta}$ is independent of $\lambda_{\mathbf{g}}$ and $\lambda_{\mathbf{a}}$)**.** *$\boldsymbol{\zeta}$ is a direction of increasing loss, independent of the choices of $\lambda_{\mathbf{g}}$ and $\lambda_{\mathbf{a}}$.*

Proof. Recall that $(\lambda_{\mathbf{g}}\mathbf{I}_m + \bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}/b)$ and $(\lambda_{\mathbf{a}}\mathbf{I}_n + \mathbf{a}^\top \mathbf{a}/b)$ are positive semi-definite (PSD) matrices by definition. Their inverses $(\lambda_{\mathbf{g}}\mathbf{I}_m + \bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}/b)^{-1}$ and $(\lambda_{\mathbf{a}}\mathbf{I}_n + \mathbf{a}^\top \mathbf{a}/b)^{-1}$ are therefore also PSD. As the Kronecker product of PSD matrices is PSD, the conditioning matrix $((\lambda_{\mathbf{g}}\mathbf{I}_m + \bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}/b)^{-1} \otimes (\lambda_{\mathbf{a}}\mathbf{I}_n + \mathbf{a}^\top \mathbf{a}/b)^{-1} \approx \mathbf{G}^{-1})$ is PSD, and therefore the direction of the update step remains correct. □

From our formulation of $\boldsymbol{\zeta}$, we can find that, in the limit for $\lambda_{\mathbf{g}} \to \infty$, Equation (21) does not depend on $\bar{\boldsymbol{g}}$. This is computationally very beneficial as computing $\bar{\boldsymbol{g}}$ is costly as it requires one or even many additional backpropagation passes. In addition, it allows conditioning the gradient update by multiplying a $b \times b$ matrix between $\mathbf{g}^\top$ and $\mathbf{a}$, which can be done very fast.

Theorem 3 (Efficient Update Direction)**.** *In the limit of $\lambda_{\mathbf{g}} \to \infty$, the update step $\boldsymbol{\zeta}$ converges to* $\lim_{\lambda_{\mathbf{g}}\to\infty} \boldsymbol{\zeta} = \boldsymbol{\zeta}^*$*, where*

$$\boldsymbol{\zeta}^* = \mathbf{g}^\top \cdot \left(\mathbf{I}_b - \frac{1}{b\lambda_{\mathbf{a}}} \mathbf{a}\mathbf{a}^\top \left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{a}}} \mathbf{a}\mathbf{a}^\top \right)^{-1} \right) \cdot \mathbf{a} \,. \tag{21}$$

(i) Here, the update direction $\boldsymbol{\zeta}^*$ is based only on the inputs and does not require computing $\bar{\boldsymbol{g}}$ (which would require a second backpropagation pass), making it efficient.

(ii) The computational cost of computing the update $\boldsymbol{\zeta}^*$ lies in $\mathcal{O}(bn^2 + b^2 n + b^3)$, where n is the number of neurons in each layer. This comprises the conventional cost of computing the gradient $\nabla = \mathbf{g}^\top \mathbf{x}$ lying in $\mathcal{O}(bn^2)$, and the overhead of computing $\boldsymbol{\zeta}^*$ instead of ∇ lying in $\mathcal{O}(b^2 n + b^3)$. The overhead is vanishing, assuming $n \gg b$. For $b > n$ the complexity lies in $\mathcal{O}(bn^2 + n^3)$.

Proof. We first show the property (21). Note that according to (22), $\lambda_{\mathbf{g}} \cdot \left(\lambda_{\mathbf{g}} \mathbf{I}_m + \bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}/b\right)^{-1}$ converges in the limit of $\lambda_{\mathbf{g}} \to \infty$ to $\mathbf{I}_m$, and therefore (21) holds.
(i) The statement follows from the fact that the term $\bar{\boldsymbol{g}}$ does not appear in the equivalent characterization (21) of $\boldsymbol{\zeta}^*$.
(ii) We first note that the matrix $\mathbf{a}\mathbf{a}^\top$ is of dimension $b \times b$, and can be computed in $\mathcal{O}(b^2 n)$ time. Next, the matrix

$$\left(\mathbf{I}_b - \frac{1}{b\lambda_{\mathbf{a}}} \mathbf{a}\mathbf{a}^\top \left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{a}}} \mathbf{a}\mathbf{a}^\top \right)^{-1} \right)$$

is of shape $b \times b$ and can be multiplied with $\mathbf{a}$ in $\mathcal{O}(b^2 n)$ time. □

Notably, (21) can be computed with a vanishing computational overhead and with only minor modifications to the implementation. Specifically, only the $\mathbf{g}^\top \mathbf{a}$ expression has to be replaced by (21) in the backpropagation step. As this can be done independently for each layer, this lends itself also to applying it only to individual layers.

As we see in the experimental section, in many cases in the mini-batch regime (i.e., $b < n$), the optimal (or a good) choice for $\lambda_{\mathbf{g}}$ actually lies in the limit to ∞. This is a surprising result, leading to the efficient and effective $\boldsymbol{\zeta}^* = \boldsymbol{\zeta}_{\lambda_{\mathbf{g}}\to\infty}$ optimizer.

Remark 2 (Relation between Update Direction $\boldsymbol{\zeta}$ and $\boldsymbol{\zeta}^*$)**.** *When comparing the update direction $\boldsymbol{\zeta}$ in* (20) *without regularization (i.e., $\lambda_{\mathbf{g}} \to 0, \lambda_{\mathbf{a}} \to 0$) with $\boldsymbol{\zeta}^*$ (i.e., $\lambda_{\mathbf{g}} \to \infty$) as given in* (21)*, it can be directly seen that $\boldsymbol{\zeta}^*$ corresponds to a particular pre-conditioning of $\boldsymbol{\zeta}$, since $\boldsymbol{\zeta}^* = M\boldsymbol{\zeta}$ for $M = \frac{1}{b\lambda_{\mathbf{g}}} \bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}$.*

As the last theoretical property of our proposed update direction $\boldsymbol{\zeta}^*$, we show that in specific networks $\boldsymbol{\zeta}^*$ coincides with the Gauss-Newton update direction.

Theorem 4 ($\boldsymbol{\zeta}^*$ is Exact for the Last Layer)**.** *For the case of linear regression or, more generally, the last layer of networks, with the mean squared error, $\boldsymbol{\zeta}^*$ is the Gauss-Newton update direction.*

Proof. The Hessian matrix of the mean squared error loss is the identity matrix. Correspondingly, the expectation value of $\bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}$ is $\mathbf{I}$. Thus, $\boldsymbol{\zeta}^* = \boldsymbol{\zeta}$. □

Remark 3. *The direction $\boldsymbol{\zeta}^*$ corresponds to the Gauss-Newton update direction with an approximation of $\mathbf{G}$ that can be expressed as $\mathbf{G} \approx \mathbb{E}\left[\mathbf{I} \otimes (\mathrm{a}^\top \mathrm{a})\right]$.*

Remark 4 (Extension to the Natural Gradient)**.** *In some cases, it might be more desirable to use the Fisher-based natural gradient instead of the Gauss-Newton method. The difference to this setting is that in* (5) *the GGN matrix $\mathbf{G}$ is replaced by the empirical Fisher information matrix $\mathbf{F}$.*

We note that our theory also applies to $\mathbf{F}$, and that $\boldsymbol{\zeta}^*$ also efficiently approximates the natural gradient update step $\mathbf{F}^{-1}\nabla$. The i-th diagonal block of $\mathbf{F}$ ($\mathbf{F}_{\theta^{(i)}} = \mathbb{E}\big[(\mathrm{g}_i^\top \mathrm{g}_i) \otimes (a_{i-1}^\top \otimes a_{i-1})\big]$), has the same form as a block of the GGN matrix $\mathbf{G}$ ($\mathbf{G}_{\theta^{(i)}} = \mathbb{E}\big[(\bar{g}_i^\top \bar{g}_i) \otimes (a_{i-1}^\top \otimes a_{i-1})\big]$). Thus, we can replace $\bar{\boldsymbol{g}}$ with $\mathbf{g}$ in our theoretical results to obtain their counterparts for $\mathbf{F}$.

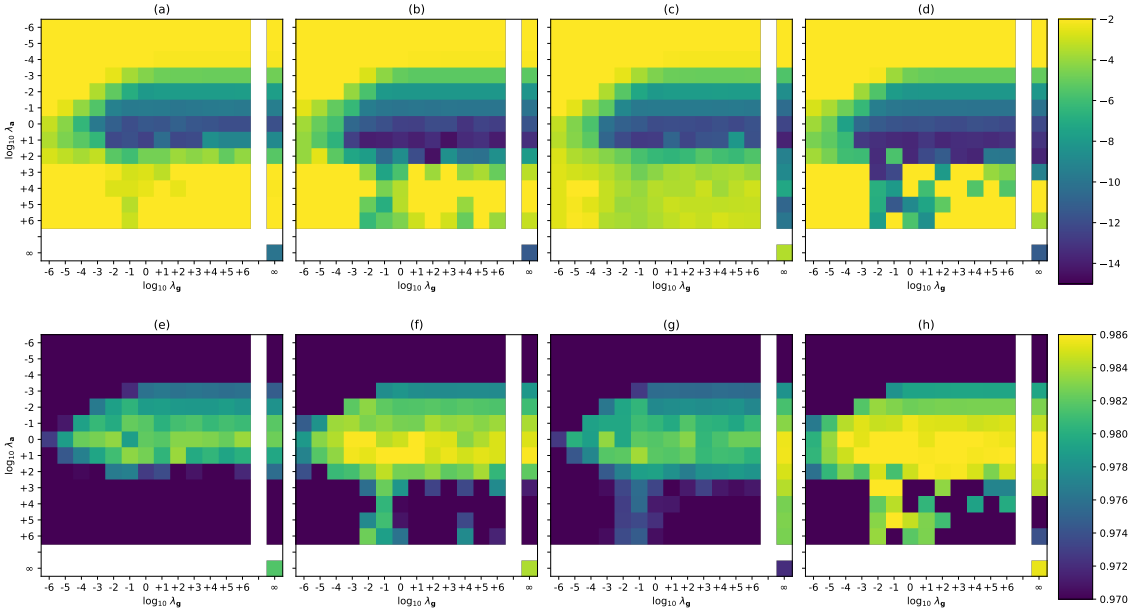

Figure 1: Logarithmic training loss (top) and test accuracy (bottom) on the MNIST classification task. The axes are the regularization parameters $\lambda_{\mathbf{g}}$ and $\lambda_{\mathbf{a}}$ in logarithmic scale with base 10. Training with a 5-layer ReLU activated network with 100 (left, a, e), 400 (center, b, c, f, g), and 1 600 (right, d, h) neurons per layer. The optimizer is SGD except for (c, g) where the optimizer is SGD with momentum. The top-left sector is $\boldsymbol{\zeta}$, the top-right column is $\boldsymbol{\zeta}^*$, and the bottom-right corner is ∇ (gradient descent). For each experiment and each of the three sectors, we use one learning rate, i.e., $\boldsymbol{\zeta}$, $\boldsymbol{\zeta}^*$, ∇ have their own learning rate to make a fair comparison between the methods; within each sector the learning rate is constant. We can observe that in the limit of $\lambda_{\mathbf{g}} \to \infty$ (i.e., in the limit to the right) the performance remains good, showing the utility of $\boldsymbol{\zeta}^*$.

4 Experiments

In the previous section, we discussed the theoretical properties of the proposed update directions $\boldsymbol{\zeta}$ and $\boldsymbol{\zeta}^*$ with the aspect that $\boldsymbol{\zeta}^*$ would actually be "free" to compute in the mini-batch regime. In this section, we provide empirical evidence that $\boldsymbol{\zeta}^*$ is a good update direction, even in deep learning. Specifically, we demonstrate that

(E1) $\boldsymbol{\zeta}^*$ achieves similar performance to K-FAC, while being substantially cheaper to compute.

(E2) The performance of our proposed method can be empirically maintained in the mini-batch regime ($n \gg b$).

(E3) $\boldsymbol{\zeta}^*$ may be used for individual layers, while for other layers only the gradient ∇ is used. This still leads to improved performance.

(E4) $\boldsymbol{\zeta}^*$ also improves the performance for training larger models such as BERT and ResNet.

(E5) The runtime and memory requirements of $\boldsymbol{\zeta}^*$ are comparable to those of gradient descent.

E1: Impact of Regularization Parameters

For (E1), we study the dependence of the model's performance on the regularization parameters $\lambda_{\mathbf{g}}$ and $\lambda_{\mathbf{a}}$. Here, we train a 5-layer deep neural network on the MNIST classification task [16] with a batch size of 60 for a total of 40 epochs or 40 000 steps.

The plots in Figure 1 demonstrate that the advantage of training by conditioning with curvature information can be achieved by considering both layer inputs $\mathbf{a}$ and gradients with respect to random samples $\bar{\boldsymbol{g}}$, but also using only layer inputs $\mathbf{a}$. In the plot, we show the performance of $\boldsymbol{\zeta}$ for different choices of $\lambda_{\mathbf{g}}$ and $\lambda_{\mathbf{a}}$, each in the range from 10^{-6} to 10^{6}. The right column shows $\boldsymbol{\zeta}^*$, i.e., $\lambda_{\mathbf{g}} = \infty$, for different $\lambda_{\mathbf{a}}$. The bottom-right corner is gradient descent, which corresponds to $\lambda_{\mathbf{g}} = \infty$ and $\lambda_{\mathbf{a}} = \infty$.

Newton's method or the general K-FAC approximation corresponds to the area with small $\lambda_{\mathbf{g}}$ and $\lambda_{\mathbf{a}}$. The interesting finding here is that the performance does not suffer by increasing $\lambda_{\mathbf{g}}$ toward ∞, i.e., from left to right in the plot.

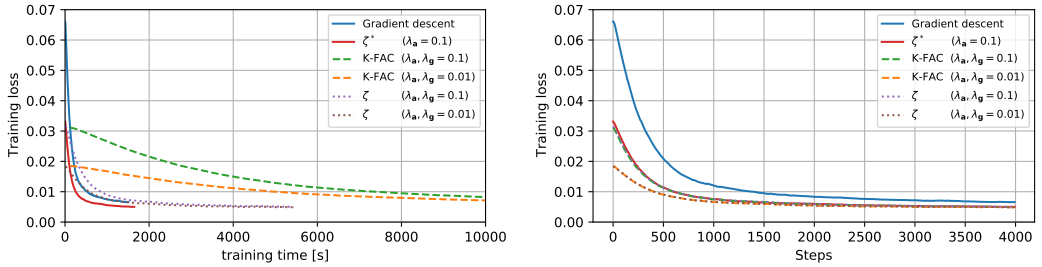

Figure 2: Training loss of the MNIST auto-encoder trained with gradient descent, K-FAC, ζ, and ζ^*. Comparing the performance per real-time (left) and per number of update steps (right). Runtimes are for a CPU core.

In addition, in Figure 3, we consider the case of regression with an auto-encoder trained with the MSE loss on MNIST [16] and Fashion-MNIST [17]. Here, we follow the same principle as above and also find that ζ^* performs well.

In Figure 7, we compare the loss for different methods. Here, we distinguish between loss per time (left) and loss per number of steps (right). We can observe that, for $\lambda = 0.1$, K-FAC, ζ, and ζ^* are almost identical per update step (right), while ζ^* is by a large margin the fastest, followed by ζ, and the conventional K-FAC implementation is the slowest (left). On the other hand, for $\lambda = 0.01$ we can achieve a faster convergence than with $\lambda = 0.1$, but here only the K-FAC and ζ methods are numerically stable, while ζ^* is unstable in this case. This means in the regime of very small λ, ζ^* is not as robust as K-FAC and ζ, however, it achieves good performance with small but moderate λ like $\lambda = 0.1$. For $\lambda < 0.01$, also K-FAC and ζ become numerically unstable in this setting and, in general, we observed that the smallest valid λ for K-FAC is 0.01 or 0.001 depending on model and task. Under consideration of the runtime, ζ^* performs best as it is almost as fast as gradient descent while performing equivalent to K-FAC and ζ. Specifically, a gradient descent step is only about 10% faster than ζ^*.

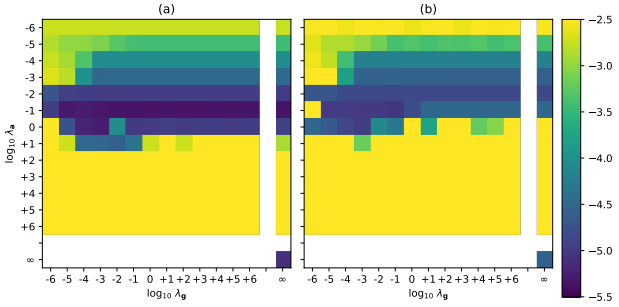

Figure 3: Training an auto-encoder on MNIST (left) and Fashion-MNIST (right). The model is the same as used by Botev *et al.* [18], i.e., it is a ReLU-activated 6-layer fully connected model with dimensions `784-1000-500- 30-500-1000-784`. Displayed is the logarithmic training loss.

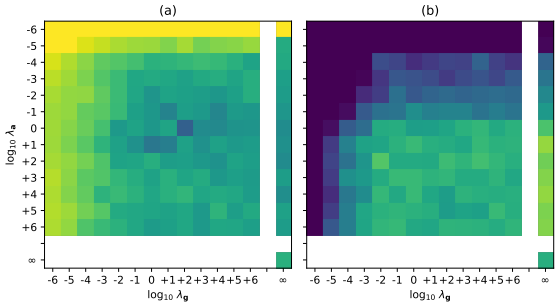

Figure 4: Training a 5-layer ReLU network with 400 neurons per layer on the MNIST classification task (as in Figure 1) but with the Adam optimizer [19].

E2: Minibatch Regime

For (E2), in Figure 1, we can see that training performs well for $n \in \{100, 400, 1\,600\}$ neurons per layer at a batch size of only 60. Also, in all other experiments, we use small batch sizes of between 8 and 100.

E3: ζ^* in Individual Layers

In Figure 5, we train the 5-layer fully connected model with 400 neurons per layer. Here, we consider the setting that we use ζ^* in some of the layers while using the default gradient ∇ in other layers. Specifically, we consider the

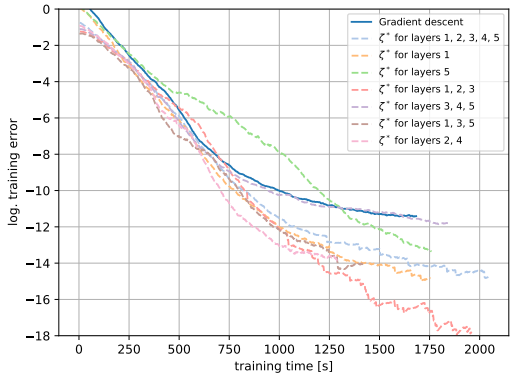

Figure 5: Training on the MNIST classification task using ζ^* only in selected layers. Runtimes are for CPU.

Table 1: BERT results for fine-tuning pre-trained BERT-Base (B-B) and BERT-Mini (B-M) models on the COLA, MRPC, and STSB text classification tasks. Larger values are better for all metrics. MCC is the Matthews correlation. Results averaged over 10 runs.

Method / Setting	CoLA (B-B)	CoLA (B-M)	MRPC (B-B)		STS-B (B-M)	
Metric	MCC	MCC	Acc.	F1	Pearson	Spearman
Gradient baseline	54.20 ± 7.56	21.08 ± 2.88	82.52 ± 1.22	87.88 ± 0.74	76.98 ± 1.10	76.88 ± 0.79
ζ^*	57.62 ± 1.59	24.67 ± 2.62	83.28 ± 0.89	88.28 ± 0.70	81.09 ± 1.58	80.82 ± 1.57

settings, where all, the first, the final, the first three, the final three, the odd numbered, and the even numbered layers are updated by ζ^*. We observe that all settings with ζ^* perform better than plain gradient descent, except for "ζ^* for layers 3,4,5" which performs approximately equivalent to gradient descent.

E4: Large-scale Models

BERT To demonstrate the utility of ζ^* also in large-scale models, we evaluate it for fine-tuning BERT [20] on three natural language tasks. In Table 1, we summarize the results for the BERT fine-tuning task. For the "Corpus of Linguistic Acceptability" (CoLA) [21] data set, we fine-tune both the BERT-Base and the BERT-Mini models and find that we outperform the gradient descent baseline in both cases. For the "Microsoft Research Paraphrase Corpus" (MRPC) [22] data set, we fine-tune the BERT-Base model and find that we outperform the baseline both in terms of accuracy and F1-score. Finally, on the "Semantic Textual Similarity Benchmark" (STS-B) [23] data set, we fine-tune the BERT-Mini model and achieve higher Pearson and Spearman correlations than the baseline. While for training with CoLA and MRPC, we were able to use the Adam optimizer [19] (which is recommended for this task and model) in conjunction with ζ^* in place of the gradient, for STS-B Adam did not work well. Therefore, for STS-B, we evaluated it using the SGD with momentum optimizer. For each method, we performed a grid search over the hyperparameters. We note that we use a batch size of 8 in all BERT experiments.

ResNet In addition, we conduct an experiment where we train the last layer of a ResNet with ζ^*, while the remainder of the model is updated using the gradient ∇. Here, we train a ResNet-18 [24] on CIFAR-10 [25] using SGD with a batch size of 100 in a vanilla setting, i.e., without additional tricks employed in by He *et al.* [24] and others. Specifically, we use (i) a constant learning rate for each training (optimal from $(1, 0.3, 0.1, 0.03, 0.01)$) and (ii) vanilla SGD and not momentum-based SGD. The reason behind this is that we want a vanilla experiment and with aspects such as extensively tuning multiple parameters of learning rate scheduler would make the evaluation less transparent; however, therefore, all accuracies are naturally lower than SOTA. In Figure 6, we plot the test accuracy against time. The results show that the proposed method outperforms vanilla SGD when applied to the last layer of a ResNet-18. To validate that the learning rate is not the cause for the better performance, we also plot the neighboring learning rates and find that even with a too small or too large learning rate ζ^* outperforms gradient descent with the optimal learning rate.

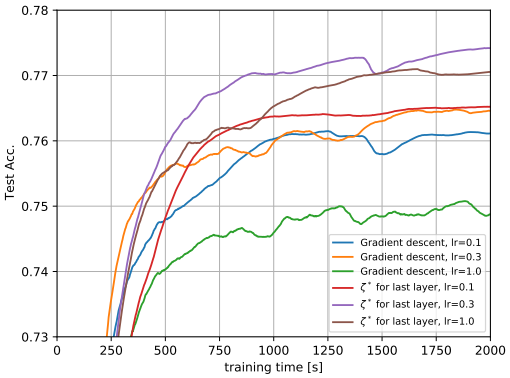

Figure 6: ResNet-18 trained on CIFAR-10. Runtimes are for a GPU. Results are averaged over 5 runs.

E5: Runtime and Memory

Finally, we also evaluate the runtime and memory requirements of each method. The runtime evaluation is displayed in Table 2. We report both CPU and GPU runtime using PyTorch [26] and (for K-FAC) the backpack library [15]. Note that the CPU runtime is more representative of the pure computational cost, as for the first rows of the GPU runtime the overhead of calling the GPU is dominant. When comparing runtimes between the gradient and ζ^* on the GPU, we can observe that we have an overhead of around $2.5\,s$ independent of the model size. The overhead for CPU time is also very small at less than 1% for the largest model, and only $1.3\,s$ for the smallest model. In

contrast, the runtime of ζ^* is around 4 times the runtime of the gradient, and K-FAC has an even substantially larger runtime. Regarding memory, ζ^* (contrasting the other approaches) also requires only a small additional footprint.

Remark 5 (Implementation)**.** *The implementation of ζ^* can be done by replacing the backpropagation step of a respective layer by* (21)*. As all "ingredients" are already available in popular deep learning frameworks, it requires only little modification (contrasting K-FAC and ζ, which require at least one additional backpropagation.)*

Table 2: Runtimes and memory requirements for different models. Runtime is the training time per epoch on MNIST at a batch size of 60, i.e., for 1 000 training steps. The K-FAC implementation is from the `backpack` library [15]. The GPU is an Nvidia A6000.

	Gradient			K-FAC			ζ			ζ^*		
Model	CPU time	GPU time	Memory	CPU time	GPU t.	Memory	CPU time	GPU t.	Memory	CPU t.	GPU t.	Memory
5 layers w/ 100 n.	2.05 s	1.79 s	1.0 MB	62.78 s	17.63 s	11.5 MB	8.65 s	11.76 s	1.6 MB	3.34 s	4.07 s	1.0 MB
5 layers w/ 400 n.	23.74 s	1.84 s	4.8 MB	218.48 s	32.00 s	22.4 MB	38.67 s	12.62 s	7.7 MB	13.62 s	4.19 s	4.9 MB
5 layers w/ 1 600 n.	187.87 s	1.93 s	51.0 MB	6985.48 s	156.48 s	212.2 MB	665.80 s	12.53 s	85.8 MB	291.01 s	4.49 s	51.4 MB
5 layers w/ 6 400 n.	3439.59 s	8.22 s	691.0 MB	—	1320.81 s	3155.3 MB	9673 s	31.87 s	1197.8 MB	3451.61 s	10.24 s	692.5 MB
Auto-Encoder	78.61 s	2.20 s	16.2 MB	1207.58 s	74.09 s	70.7 MB	193.25 s	14.19 s	33.8 MB	87.39 s	4.93 s	16.5 MB

We will publish the source code of our implementation. In the appendix, we give a PyTorch [26] implementation of the proposed method (ζ^*).

5 Related Work

Our methods are related to K-FAC by Martens and Grosse [12]. K-FAC uses the approximation (13) to approximate the blocks of the Hessian of the empirical risk of neural networks. In most implementations of K-FAC, the off-diagonal blocks of the Hessian are also set to zero. One of the main claimed benefits of K-FAC is its speed (compared to stochastic gradient descent) for large-batch size training. That said, recent empirical work has shown that this advantage of K-FAC disappears once the additional computational costs of hyperparameter tuning for large batch training is accounted for. There is a line of work that extends the basic idea of K-FAC to convolutional layers [27]. Botev *et al.* [18] further extend these ideas to present KFLR, a Kronecker factored low-rank approximation, and KFRA, a Kronecker factored recursive approximation of the Gauss-Newton step. Singh and Alistarh [28] propose WoodFisher, a Woodbury matrix inverse-based estimate of the inverse Hessian, and apply it to neural network compression. Yao *et al.* [29] propose AdaHessian, a second-order optimizer that incorporates the curvature of the loss function via an adaptive estimation of the Hessian. Frantar *et al.* [6] propose M-FAC, a matrix-free approximation of the natural gradient through a queue of the (e.g., 1 000) recent gradients. These works fundamentally differ from our approach in that their objective is to approximate the Fisher or Gauss-Newton matrix inverse vector products. In contrast, this work proposes to approximate the Gauss-Newton matrix by only one of its Kronecker factors, which we find to achieve good performance at a substantial computational speedup and reduction of memory footprint. For an overview of this area, we refer to Kunstner *et al.* [30] and Martens [31]. For an overview of the technical aspects of backpropagation of second-order quantities, we refer to Dangel *et al.* [15], [32]

Taking a step back, K-FAC is one of many Newton-type methods for training neural networks. Other prominent examples of such methods include subsampled Newton methods [33], [34] (which approximate the Hessian by subsampling the terms in the empirical risk function and evaluating the Hessian of the subsampled terms) and sketched Newton methods [3]–[5] (which approximate the Hessian by sketching, e.g., by projecting the Hessian to a lower-dimensional space by multiplying it with a random matrix). The main features that distinguish K-FAC from this group of methods are K-FAC's superior empirical performance and K-FAC's lack of theoretical justification.

6 Conclusion

In this work, we presented ISAAC Newton, a novel approximate curvature method based on layer-inputs. We demonstrated it to be a special case of the regularization-generalized Gauss-Newton method and empirically demonstrate its utility. Specifically, our method features an asymptotically vanishing computational overhead in the mini-batch regime, while achieving competitive empirical performance on various benchmark problems.

References

[1] N. Agarwal, B. Bullins, and E. Hazan, "Second-order stochastic optimization for machine learning in linear time," *Journal on Machine Learning Research*, vol. 18, no. 1, pp. 4148–4187, 2017.

[2] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2e. New York, NY, USA: Springer, 2006.

[3] A. Gonen and S. Shalev-Shwartz, "Faster SGD using sketched conditioning," *arXiv preprint, arXiv:1506.02649*, 2015.

[4] M. Pilanci and M. J. Wainwright, "Newton sketch: A near linear-time optimization algorithm with linear-quadratic convergence," *SIAM Journal on Optimization*, vol. 27, 2017.

[5] M. A. Erdogdu and A. Montanari, "Convergence rates of sub-sampled Newton methods," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2015.

[6] E. Frantar, E. Kurtic, and D. Alistarh, "M-FAC: Efficient matrix-free approximations of second-order information," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2021.

[7] N. Doikov and Y. Nesterov, "Convex Optimization based on Global Lower Second-order Models," in *Proc. Neural Information Processing Systems (NeurIPS)*, Curran Associates, Inc., 2020.

[8] Y. Nesterov and B. T. Polyak, "Cubic regularization of Newton method and its global performance," *Mathematical Programming*, vol. 108, 2006.

[9] S. Becker and Y. Lecun, "Improving the convergence of back-propagation learning with second-order methods," 1989.

[10] T. Schaul, S. Zhang, and Y. LeCun, "No more pesky learning rates," in *International Conference on Machine Learning (ICML)*, 2013.

[11] Y. Ollivier, "Riemannian metrics for neural networks i: Feedforward networks," *Information and Inference*, vol. 4, pp. 108–153, Jun. 2015.

[12] J. Martens and R. Grosse, "Optimizing neural networks with Kronecker-factored approximate curvature," in *International Conference on Machine Learning (ICML)*, 2015.

[13] A. N. Tikhonov and V. Y. Arsenin, *Solutions of Ill-posed problems*. W.H. Winston, 1977.

[14] P. Chen, "Hessian matrix vs. Gauss—Newton Hessian matrix," *SIAM Journal on Numerical Analysis*, 2011.

[15] F. Dangel, F. Kunstner, and P. Hennig, "Backpack: Packing more into backprop," in *International Conference on Learning Representations*, 2020.

[16] Y. LeCun, C. Cortes, and C. Burges, "MNIST Handwritten Digit Database," *ATT Labs*, 2010.

[17] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms," *arXiv*, 2017.

[18] A. Botev, H. Ritter, and D. Barber, "Practical Gauss-Newton optimisation for deep learning," in *International Conference on Machine Learning (ICML)*, 2017.

[19] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015.

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2018.

[21] A. Warstadt, A. Singh, and S. R. Bowman, "Neural network acceptability judgments," *Transactions of the Association for Computational Linguistics*, vol. 7, 2019.

[22] W. B. Dolan and C. Brockett, "Automatically constructing a corpus of sentential paraphrases," in *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005.

[23] D. Cer, M. Diab, E. Agirre, I. Lopez-Gazpio, and L. Specia, "SemEval-2017 task 1: Semantic textual similarity multilingual and crosslingual focused evaluation," in *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, Vancouver, Canada: Association for Computational Linguistics, 2017.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[25] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (Canadian Institute for Advanced Research)," 2009.

[26] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2019.

[27] R. Grosse and J. Martens, "A Kronecker-factored approximate Fisher matrix for convolution layers," in *International Conference on Machine Learning (ICML)*, 2016.

[28] S. P. Singh and D. Alistarh, "Woodfisher: Efficient second-order approximation for neural network compression," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2020.

[29] Z. Yao, A. Gholami, S. Shen, M. Mustafa, K. Keutzer, and M. W. Mahoney, "Adahessian: An adaptive second order optimizer for machine learning," in *AAAI Conference on Artificial Intelligence*, 2021.

[30] F. Kunstner, L. Balles, and P. Hennig, "Limitations of the empirical Fisher approximation for natural gradient descent," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2019.

[31] J. Martens, "New insights and perspectives on the natural gradient method," *Journal of Machine Learning Research*, 2020.

[32] F. Dangel, S. Harmeling, and P. Hennig, "Modular block-diagonal curvature approximations for feedforward architectures," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2020.

[33] F. Roosta-Khorasani and M. W. Mahoney, "Sub-Sampled Newton Methods I: Globally Convergent Algorithms," *arXiv: 1601.04737*, 2016.

[34] P. Xu, J. Yang, F. Roosta, C. Ré, and M. W. Mahoney, "Sub-sampled Newton Methods with Non-uniform Sampling," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2016.

Checklist

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

A PyTorch Implementation

We display a PyTorch [26] implementation of ISAAC for a fully-connected layer below. Here, we mark the important part (i.e., the part beyond the boilerplate) with a red rectangle.

```
import torch

class ISAACLinearFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weight, bias, la, inv_type):
        ctx.save_for_backward(input, weight, bias)
        ctx.la = la
        if inv_type == 'cholesky_inverse':
            ctx.inverse = torch.cholesky_inverse
        elif inv_type == 'inverse':
            ctx.inverse = torch.inverse
        else:
            raise NotImplementedError(inv_type)
        return input @ weight.T + (bias if bias is not None else 0)

    @staticmethod
    def backward(ctx, grad_output):
        input, weight, bias = ctx.saved_tensors
        if ctx.needs_input_grad[0]:
            grad_0 = grad_output @ weight
        else:
            grad_0 = None

        if ctx.needs_input_grad[1]:

            aaT = input @ input.T / grad_output.shape[0]
            I_b = torch.eye(aaT.shape[0], device=aaT.device, dtype=aaT.dtype)
            aaT_IaaT_inv = aaT @ ctx.inverse(aaT / ctx.la + I_b)
            grad_1 = grad_output.T @ (
                    I_b - 1. / ctx.la * aaT_IaaT_inv
            ) @ input

        else:
            grad_1 = None

        return (
            grad_0,
            grad_1,
            grad_output.mean(0, keepdim=True) if bias is not None else None,
            None, None, None,
        )

class ISAACLinear(torch.nn.Linear):
    def __init__(self, in_features, out_features,
                 la, inv_type='inverse', **kwargs):
        super(ISAACLinear, self).__init__(
            in_features=in_features, out_features=out_features, **kwargs
        )
        self.la = la
        self.inv_type = inv_type

    def forward(self, input: torch.Tensor) -> torch.Tensor:
        return ISAACLinearFunction.apply(
            input, self.weight,
```

```
            self.bias.unsqueeze(0) if self.bias is not None else None,
            self.la,
            self.inv_type
        )
```

B Implementation Details

Unless noted differently, for all experiments, we tune the learning rate on a grid of $(1, 0.3, 0.1, 0.03, 0.01, 0.003, 0.001)$. We verified this range to cover the full reasonable range of learning rates. Specifically, for every single experiment, we made sure that there is no learning rate outside this range which performs better.

For all language model experiments, we used the respective Huggingface PyTorch implementation.

All other hyperparameter details are given in the main paper.

The code will be made publicly available.

C Additional Proofs

Proof of Theorem 1. We first show, that $\boldsymbol{\zeta}$ as defined in (17) can be expressed as in (20). Indeed by using (19), the Woodbury matrix identity and by regularizing the inverses, we can see that

$$
\begin{aligned}
\boldsymbol{\zeta} &= \lambda_{\mathbf{g}}\lambda_{\mathbf{a}}(\bar{\boldsymbol{g}}^\top\bar{\boldsymbol{g}}/b + \lambda_{\mathbf{g}}\mathbf{I})^{-1} \otimes (\mathbf{a}^\top\mathbf{a}/b + \lambda_{\mathbf{a}}\mathbf{I})^{-1}\mathbf{g}^\top\mathbf{a} \\
&= \lambda_{\mathbf{g}}\lambda_{\mathbf{a}} \cdot \left(\lambda_{\mathbf{g}}\mathbf{I}_m + \bar{\boldsymbol{g}}^\top\bar{\boldsymbol{g}}/b\right)^{-1}\mathbf{g}^\top\mathbf{a}\left(\lambda_{\mathbf{a}}\mathbf{I}_n + \mathbf{a}^\top\mathbf{a}/b\right)^{-1} \\
&= \lambda_{\mathbf{g}}\lambda_{\mathbf{a}} \cdot \left(\frac{1}{\lambda_{\mathbf{g}}}\mathbf{I}_m - \frac{1}{b{\lambda_{\mathbf{g}}}^2}\bar{\boldsymbol{g}}^\top\left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{g}}}\bar{\boldsymbol{g}}\bar{\boldsymbol{g}}^\top\right)^{-1}\bar{\boldsymbol{g}}\right) \\
&\quad \mathbf{g}^\top\mathbf{a}\left(\frac{1}{\lambda_{\mathbf{a}}}\mathbf{I}_n - \frac{1}{b{\lambda_{\mathbf{a}}}^2}\mathbf{a}^\top\left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{a}}}\mathbf{a}\mathbf{a}^\top\right)^{-1}\mathbf{a}\right) \\
&= \left(\mathbf{I}_m - \frac{1}{b\lambda_{\mathbf{g}}}\bar{\boldsymbol{g}}^\top\left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{g}}}\bar{\boldsymbol{g}}\bar{\boldsymbol{g}}^\top\right)^{-1}\bar{\boldsymbol{g}}\right) \cdot \mathbf{g}^\top \\
&\quad \cdot \mathbf{a} \cdot \left(\mathbf{I}_n - \frac{1}{b\lambda_{\mathbf{a}}}\mathbf{a}^\top\left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{a}}}\mathbf{a}\mathbf{a}^\top\right)^{-1}\mathbf{a}\right) \\
&= \left(\mathbf{I}_m - \frac{1}{b\lambda_{\mathbf{g}}}\bar{\boldsymbol{g}}^\top\left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{g}}}\bar{\boldsymbol{g}}\bar{\boldsymbol{g}}^\top\right)^{-1}\bar{\boldsymbol{g}}\right) \cdot \mathbf{g}^\top \\
&\quad \cdot \left(\mathbf{a} - \frac{1}{b\lambda_{\mathbf{a}}}\mathbf{a}\mathbf{a}^\top\left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{a}}}\mathbf{a}\mathbf{a}^\top\right)^{-1}\mathbf{a}\right) \\
&= \left(\mathbf{I}_m - \frac{1}{b\lambda_{\mathbf{g}}}\bar{\boldsymbol{g}}^\top\left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{g}}}\bar{\boldsymbol{g}}\bar{\boldsymbol{g}}^\top\right)^{-1}\bar{\boldsymbol{g}}\right) \cdot \mathbf{g}^\top \\
&\quad \cdot \left(\mathbf{I}_b - \frac{1}{b\lambda_{\mathbf{a}}}\mathbf{a}\mathbf{a}^\top\left(\mathbf{I}_b + \frac{1}{b\lambda_{\mathbf{a}}}\mathbf{a}\mathbf{a}^\top\right)^{-1}\right) \cdot \mathbf{a}
\end{aligned}
$$

To show Assertion (i), we note that according to (17)

$$
\begin{aligned}
&\lim_{\lambda_{\mathbf{g}},\lambda_{\mathbf{a}}\to 0} \frac{1}{\lambda_{\mathbf{g}}\lambda_{\mathbf{a}}}\boldsymbol{\zeta} \\
&= \lim_{\lambda_{\mathbf{g}},\lambda_{\mathbf{a}}\to 0} (\bar{\boldsymbol{g}}^\top\bar{\boldsymbol{g}}/b + \lambda_{\mathbf{g}}\mathbf{I})^{-1} \otimes (\mathbf{a}^\top\mathbf{a}/b + \lambda_{\mathbf{a}}\mathbf{I})^{-1}\mathbf{g}^\top\mathbf{a} \\
&= (\bar{\boldsymbol{g}}^\top\bar{\boldsymbol{g}})^{-1} \otimes (\mathbf{a}^\top\mathbf{a})^{-1}\mathbf{g}^\top\mathbf{a} \\
&\approx \mathbf{G}^{-1}\mathbf{g}^\top\mathbf{a},
\end{aligned}
$$

where the first equality uses the definition of $\boldsymbol{\zeta}$ in (17). The second equality is due to the continuity of the matrix inversion and the last approximate equality follows from the K-FAC approximation (15).

To show Assertion (ii), we consider $\lim_{\lambda_{\mathbf{g}} \to \infty}$ and $\lim_{\lambda_{\mathbf{a}} \to \infty}$ independently, that is

$$\begin{aligned} &\lim_{\lambda_{\mathbf{g}} \to \infty} \lambda_{\mathbf{g}} \cdot \left(\lambda_{\mathbf{g}} \mathbf{I}_m + \bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}/b\right)^{-1} \\ &= \lim_{\lambda_{\mathbf{g}} \to \infty} \left(\mathbf{I}_m + \frac{1}{b\lambda_{\mathbf{g}}} \bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}\right)^{-1} = \mathbf{I}_m, \end{aligned} \tag{22}$$

and

$$\begin{aligned} &\lim_{\lambda_{\mathbf{a}} \to \infty} \lambda_{\mathbf{a}} \cdot \left(\lambda_{\mathbf{a}} \mathbf{I}_n + \mathbf{a}^\top \mathbf{a}/b\right)^{-1} \\ &= \lim_{\lambda_{\mathbf{a}} \to \infty} \left(\mathbf{I}_n + \frac{1}{b\lambda_{\mathbf{a}}} \mathbf{a}^\top \mathbf{a}\right)^{-1} = \mathbf{I}_n. \end{aligned} \tag{23}$$

This then implies

$$\begin{aligned} \lim_{\lambda_{\mathbf{g}}, \lambda_{\mathbf{a}} \to \infty} &\lambda_{\mathbf{g}} \left(\lambda_{\mathbf{g}} \mathbf{I}_m + \bar{\boldsymbol{g}}^\top \bar{\boldsymbol{g}}/b\right)^{-1} \cdot \mathbf{g}^\top \\ &\cdot \mathbf{a} \cdot \lambda_{\mathbf{a}} \left(\lambda_{\mathbf{a}} \mathbf{I}_n + \mathbf{a}^\top \mathbf{a}/b\right)^{-1} \\ &= \mathbf{I}_m \cdot \mathbf{g}^\top \mathbf{a} \cdot \mathbf{I}_n = \mathbf{g}^\top \mathbf{a}, \end{aligned} \tag{24}$$

which concludes the proof. □

D Additional Experiments

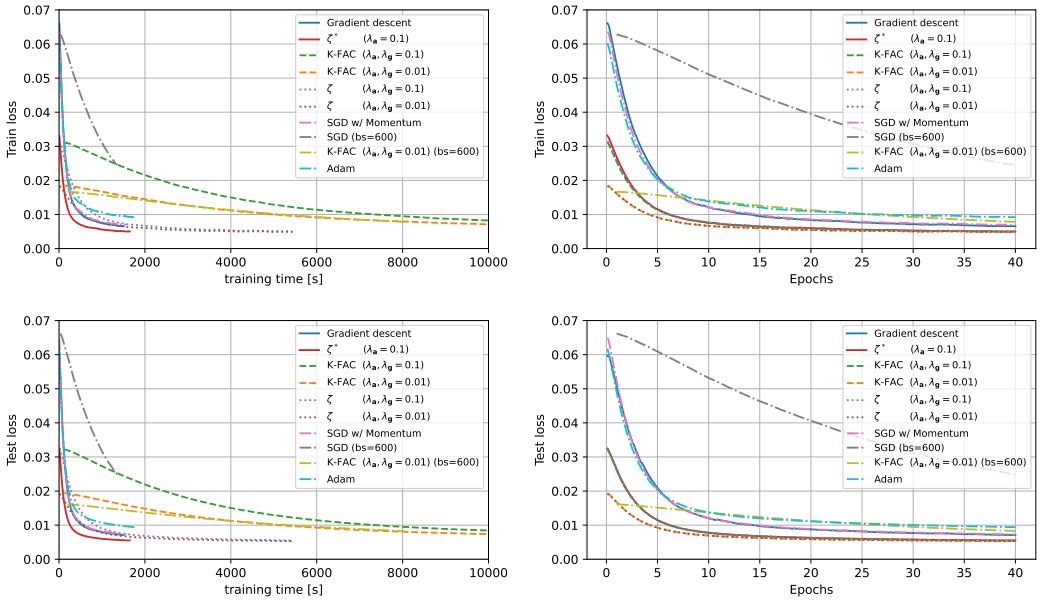

Figure 7: Training loss of the MNIST auto-encoder trained with gradient descent, K-FAC, $\boldsymbol{\zeta}$, $\boldsymbol{\zeta}^*$, as well as SGD w/ momentum, SGD with a $10\times$ larger batch size (600), K-FAC with a $10\times$ larger batch size (600), and Adam. Comparing the performance per real-time (left) and per number of epochs (right). We display both the training loss (top) as well as the test loss (bottom) Runtimes are for a CPU core.

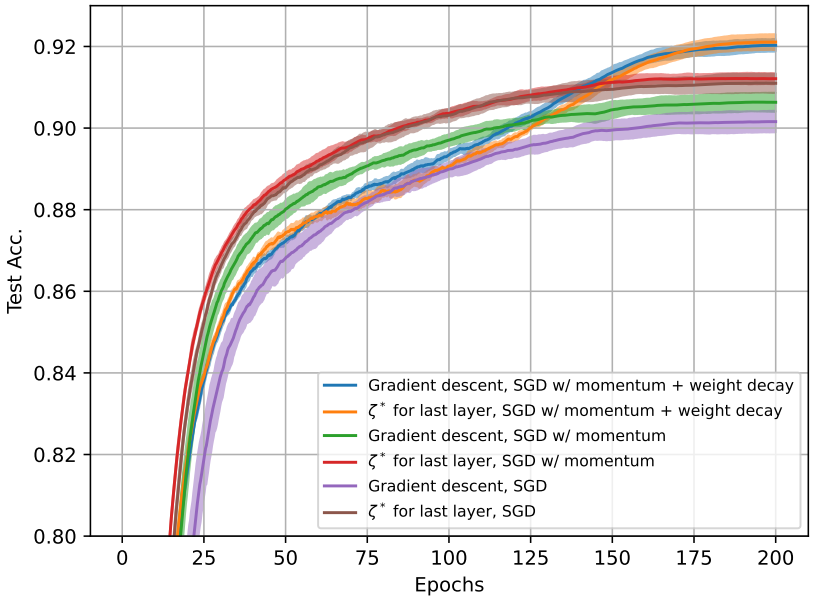

Figure 8: ResNet-18 trained on CIFAR-10 with image augmentation and a cosine learning rate schedule. The first line (blue) uses the hyperparameters of a public implementation. To ablate the optimizer, two additional settings are added, specifically, without weight decay and without momentum. Results are averaged over 5 runs and the standard deviation is indicated with the colored areas.

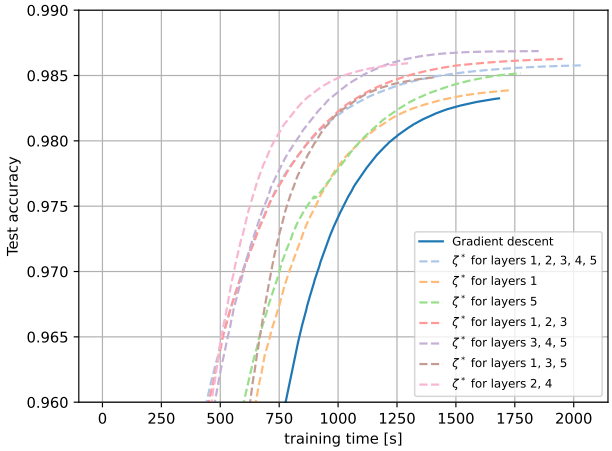

Figure 9: Test accuracy for training on the MNIST classification task using $\boldsymbol{\zeta}^*$ only in selected layers. Runtimes are for CPU.