Component-Wise Natural Gradient Descent - An Efficient Neural Network Optimization

1st Tran Van Sang  
The University of Tokyo  
Japan  
0000-0002-4211-049X  
irvan@yamagula.ics.i.u-tokyo.ac.jp

2nd Mhd Irvan  
The University of Tokyo  
Japan

3rd Rie Shigetomi Yamaguchi  
The University of Tokyo  
Japan

4th Toshiyuki Nakata  
The University of Tokyo  
Japan

Abstract—Natural Gradient Descent (NGD) is a second-order neural network training that precondition the gradient descent with the inverse of the Fisher Information Matrix (FIM). Although NGD provides an efficient preconditioner, it is not practicable due to the expensive computation required when inverting the FIM. This paper proposes a new NGD variant algorithm named Component-Wise Natural Gradient Descent (CW-NGD). CW-NGD is composed of 2 steps. Similar to several existing works, the first step is to consider the FIM matrix as a block-diagonal matrix whose diagonal blocks correspond to the FIM of each layer's weights. In the second step, unique to CW-NGD, we analyze the layer's structure and further decompose the layer's FIM into smaller segments whose derivatives are approximately independent. As a result, individual layers' FIMs are approximated in a block-diagonal form that trivially supports the inversion. The segment decomposition strategy is varied by layer structure. Specifically, we analyze the dense and convolutional layers and design their decomposition strategies appropriately. In an experiment of training a network containing these 2 types of layers, we empirically prove that CW-NGD requires fewer iterations to converge compared to the state-of-the-art first-order and second-order methods.

Index Terms—neural network, network optimization, hessian matrix, fisher information matrix, block-diagonal matrix, quadratic curvature, convolutional layer, natural gradient descent

I. INTRODUCTION

Recently, Machine Learning has been rapidly growing and poses an inevitable role in modern society. Among various machine learning architectures, Neural Network is the most powerful and versatile architecture that commits to the success of machine learning nowadays. Especially since 2013, when ResNet [1], the first successful deep neural network, was introduced, Neural Network has been intensively investigated with regard to the number of layers, architectures, network and complexity, becoming one of the best-performing architectures in Machine Learning. Neural Network has been used in a wide range of applications, including image recognition, text recognition, speech recognition, natural language processing, machine translation, computer vision, and many others. Along with the involvement of a new network model, a crucial component in neural networks that attracts much researcher's attention is the network training (optimization) algorithm, the research field investigating algorithms that finds the optimal weights of the network.

Gradient Descent (GD) is a popular network optimization family that optimizes the weight by following the steepest descent direction of the loss function. First-order GD methods precondition the direction with a scalar learning rate, usually defined via an adaptive strategy [2]–[7], while second-order GD methods precondition the direction with the inverse of the Hessian matrix. Natural Gradient Descent (NGD) [8] is a second-order GD variant that uses the Fisher Information Matrix (FIM) [9] in the place of the Hessian. A great number of works [10]–[17] have shown that NGD and its variants produce faster convergence than first-order GD. Besides, NGD requires fewer hyperparameters, thus simplifying the tuning process. However, the application of NGD is limited due to the burdensome calculation necessary for inverting the FIM. Reducing the amount of calculation in NGD and making it applicable in practice is an open active area of research. Tackling the challenge, we investigate a novel method named Component-Wise Natural Gradient Descent (CW-NGD), an improved variant of NGD that produces high accuracy results and requires a reasonable running time.

CW-NGD is composed of 2 steps. The first step, similar to several existing works [13]–[18], is to consider the FIM matrix as a block-diagonal matrix whose diagonal blocks correspond to the FIM of each layer's weights. In the second step, which is unique to CW-NGD, for dense and convolutional layers, we analyze the layer's structure and further decompose the layer's FIM into smaller segments whose derivatives are approximately independent. As a result, individual layers' FIMs are approximated in a block-diagonal form that trivially supports the inversion.

In an experiment with a network containing convolutional and dense layers on the MNIST dataset, CW-NGD converges within fewer iterations than the best-known first-order and second-order optimization algorithms, Adam [2] and KFAC [14]–[17], respectively.

In the next section, Section II, we discuss several existing works in the literature before describing our method in Section III. The experiment detail is provided in Section IV. Next, we discuss the results of our method in
Section V. Finally, we summarize and discuss the future of our method in Section VI.

II. RELATED WORKS

First-order optimization involves the use of a learning rate adaptive strategy: Adam, AdaMax, RMSProp, AdaDelta, AdaGrad, SGD with Nesterov’s accelerated gradient, Nadam, etc. Some of them, for instance, Adam, AdaGrad [2], use the moving average of squared gradients as a heuristic of the FIM, the matrix contains the quadratic curvature of the loss function. Their prominent usage in practice and well-performed results highly promote the usefulness of the use of quadratic curve information in network optimization.

The initial idea of optimizing networks using quadratic curvature and dividing the network parameters into derivative-independent components was first proposed by Kurita et al. in 1993 [19]. The motivation of the authors was to apply the iterative weighted least squares algorithm, which is usually used in Generalized Linear Model training, to network optimization. However, this approach supports only dense networks with a single hidden layer, and the experiment was on a simple XOR problem that has not been commonly used recently. Nonetheless, it inspired the recently introduced unit-wise FIM approximation [20].

NGD [8], proposed in 1998, later became the most prominent variant of the second-order method in Neural Network optimization. NGD uses the FIM as the preconditioner of the gradient descent update in the place of the Hessian matrix. While the Hessian preconditioner optimizes the Euclidean distance, the FIM preconditioner in NGD is created to adapt to the information geometry. It aims to minimize the distance in the probabilistic space, whose distance is defined by the Kullback-Leibler divergence. Although the initial motivation of NGD is not to approximate the Hessian, it was later proven that the FIM coincides with a generalized Gauss-Newton approximation of the Hessian under overwhelmingly practical cases [22]. This coincidence explicitly implies the classification of NGD as a second-order optimization method.

NGD can produce a more efficient learning step compared to almost any first-order optimization method. However, the FIM inverting operation in NGD requires a significant computation time, which restricts the application of NGD to simple neural network models. To remedy this problem, several existing works [14]–[18], [20], [23], [24] have proposed various approximation schemes of the FIM for a cheap inversion. For instance, in 2015, inspired by [19] and theoretically evaluated by Amari et al. [24], Ollivier et al. [20] extended NGD with the unit-wise NGD method. The authors introduce a theoretical framework to build invariant algorithms, in which they treat NGD as an invariant method that optimizes the distance in Riemannian geometry. Based on the suggested framework, the authors grouped the update by the output nodes and calculated individual group’s FIM for each update, thus, ignoring the interaction among the node groups. However, the approach does not apply to the convolutional layer and is limited to simple neural network models due to the inefficient gradient calculation.

The more modern and practicable extension of NGD is KFAC, initially introduced in 2015 by James et al. [14]–[17]. This is considered the state-of-the-art second-order method in neural network optimization. KFAC works by approximating the off-diagonal blocks of the FIM as zero, then factorizing the diagonal blocks as a Kronecker product of two smaller matrices. The second step reduces the computational cost of inverting the FIM. In contrast, it greatly reduces the accuracy of the approximation.

Besides NGD, which uses the FIM as the preconditioner, another research direction of second-order optimization is to approximate the Hessian preconditioner. The most well-known methods are Broyden Fletcher Goldfarb Shanno (BFGS) and L-BFGS methods [25]. These methods work on any differentiable function and, in reality, are used to optimize the neural network loss function. Instead of calculating the Hessian in every step, the former works by updating the Hessian matrix calculated in previous iterations. Although it is more computationally efficient than directly deriving the second-order derivative of the loss function, it requires storing the update vector in each step, hence is very memory-consuming. The latter was introduced to remedy that problem by only storing the update vector of k last iterations. Nevertheless, these methods are not used in practice for large-scale problems (large networks or large datasets), because they do not perform well in mini-batch settings and produce low-accuracy results [26].

In the case of dense layers, CW-NGD is identical to unit-wise NGD introduced in [13], [20], [24]. We explicitly declare our additional contributions as follows. Firstly, CW-NGD analyzes the layer’s structure to provide an efficient FIM-approximation scheme, while unit-wise NGD works by grouping the network weights into groups by output nodes. They accidentally turn out to be the same approximation scheme in the case of dense layers. Secondly, CW-NGD is applicable to convolutional layers while unit-wise is not. Thirdly, studies on unit-wise NGD [13], [20], [24] focus on theoretical analysis of the method and lack practical application analysis. In contrast, CW-NGD is verified with the MNIST dataset [27], which demonstrates the more potential practicability of CW-NGD over unit-wise NGD.

III. METHODOLOGY

In this section, we first introduce several basic notations and the neural network definition in the first subsection. In the next subsection, we briefly introduce the NGD method, FIM definition, and layer-wise NGD. After that, we describe our proposed method Component-Wise Natural Gradient Descent (CW-NGD) in the following two subsections for the cases of dense and convolutional layers,
respectively. Finally, we provide an efficient implementation for CW-NGD, evaluate the computational performance gained by CW-NGD, and discuss the potential parallelization capability of the method.

To be consistent with the notation of derivatives with respect to vector and matrix, we consider a vector as a row of elements. In other words, a vector from $\mathbb{R}^n$ space is equivalent to a $1 \times n$ matrix in $\mathbb{R}^{1 \times n}$.

### A. Neural Network Definition And Notations

We consider a neural network consisting of $L$ dense layers and a predictive exponential family distribution $r(y|z)$. Note: we refer to a distribution by its density function for convenience. $r(y|z)$ family covers the most popular use cases, for instance, $r(y|z) = \mathcal{N}(y; z, \sigma^2)$ for least-squares regression, $r(y|z) = \sum_{c=1}^{C} y_c \exp (-z_c^2)$ for C-class classification with $y$ is one-hot vector and $z_c$ is the $c$-th element of $z$. The network transforms an input $x = a_0 \in \mathbb{R}^{n_0}$ to output $z = a_L \in \mathbb{R}^{n_L}$ through $L$ layers, and obtain the output’s probability by $r(y|z)$. For $l = 1, 2, \ldots, L$, $s_l = a_{l-1} W_l$, $a_l = \phi_l(s_l)$ where $\phi_l(s_l)$ is a (non-linear) activation function, $s_l \in \mathbb{R}^{n_l}$ is the pre-activation, $a_l \in \mathbb{R}^{n_l}$ is the (post-)activation, $W_l \in \mathbb{R}^{n_l \times n_{l+1}}$ is the weight matrix (Note: the bias term’s effect can be achieved by prepending a homogeneous value 1 to the activation vectors $a_l$).

The parameter network $\theta \in \mathbb{R}^{1 \times (n_0 n_1 + n_1 n_2 + \cdots + n_L n_L)}$ with $\theta_l = \text{vec}(W_l)$ is a vectorization of $W_l^T$ composed by chaining rows of matrix $W_l$ into a single row-vector. We define $D_\theta = \nabla_{\theta} (-\log r_\theta(y|x))$, function $f(x; \theta) = z$, and conditional distribution $r_\theta(y|x) = r(y|f(x; \theta))$ of the joint distribution $r_\theta(x,y) = r(x)r_\theta(y|z)$. Given a dataset of $P$ pairs of samples and associated labels $(x_p, y_p)$, let $D_\theta^{(p)} = \nabla_{\theta} (-\log r_\theta(y_p|x_p))$.

To easily identify elements, we use the notation $\theta_{(i,j)}$ (or $D_{\theta_{(i,j)}}$) to refer to the element of the flattened vector $\theta$ (or $D_\theta$) corresponding to the cell of the $l$-th component matrix of $W_l$ (or $D_{W_l}$) at $i$-th row and $j$-th column. Similarly, for matrix $M$ (e.g., the FIM $F_\theta$ introduced in the next subsection) whose both dimensions are equal to the vectorized vector’s dimension, we use the notation $M_{(l_1, i_1, j_1), (l_2, i_2, j_2)}$ (e.g., $\frac{\partial^2 \ell}{\partial \theta_{(l_1, i_1, j_1)} \partial \theta_{(l_2, i_2, j_2)}}$) to refer to the cell of $M$ at $(l_1, i_1, j_1)$-th row and $(l_2, i_2, j_2)$-th column.

### B. Fisher Information Matrix And Natural Gradient Descent

This subsection begins with the definitions of NGD and FIM. Then it describes the method of approximating FIM via a layer-wise block-diagonal matrix.

Gradient Descent is a family of network optimization methods that optimizes the weight by following the steepest descent, equivalently, the negative gradient, direction of the loss function. First-order optimizations precondition the direction with a scalar learning rate. Second-order optimizations precondition the direction with the inverse of the Hessian matrix. NGD [8] is a second-order optimization variant that uses the FIM [9] $F_\theta$’s inverse as the preconditioner.

\[
F_\theta = \sum_{p=1}^{P} \left( D_\theta^{(p)} \right)^T D_\theta^{(p)} 
\]

\[
F_\theta^{(\text{stat})} = \mathbb{E}_{x,y \sim r(x,y)} \left[ D_\theta^{(p)} D_\theta^{(p)^T} \right]
\]

\[
F_\theta^{(\text{EF})} = \sum_{p=1}^{P} \mathbb{E}_{x,y \sim r(y|x_p)} \left[ D_\theta^{(p)} D_\theta^{(p)^T} \right]
\]

There is usually confusion among the 3 definitions of the FIM [22]. The definition given by Eq. 3 is called Empirical FIM (EF) by statisticians [28], [29] but is called FIM in Machine Learning literature [30], [31]. On the other hand, Eq. 2 is what statisticians call the FIM, while Eq. 1 is called EF in Machine Learning. We explicitly use the definition in Eq. 1 as FIM in this work for simplicity.

NGD provides an efficient preconditioner for the gradient descent in practice [10]–[17]. However, its usage is limited due to the expensive calculation required to invert the FIM. The common workaround is to approximate the FIM as a block-diagonal matrix whose diagonal blocks are the FIM of each layer $F_{\theta_l}$, such as in KFAC [14]–[17].

\[
F_{\theta_l} = \text{diag}(F_{\theta_1}, \ldots, F_{\theta_L})
\]

\[
F_{\theta_l}^{-1} = \text{diag}(F_{\theta_1}^{-1}, \ldots, F_{\theta_L}^{-1})
\]

with $diag(A_1, A_2, \ldots, A_n)$ is a block diagonal matrix whose diagonal blocks are $A_1, A_2, \ldots, A_n$ in the given order. By this approximation, instead of inverting one large $F_{\theta_l}$, we can invert individual layer FIMs $F_{\theta_{l,i}}$ ($l = 1, 2, \ldots, L$) and aggregate them into a block-diagonal matrix. However, in practice, $F_{\theta_{l,i}}$ is usually large and still expensive to invert. KFAC [14]–[17] remedies this by decomposing the layer FIM as a Kronecker product of smaller matrices, but it produces a low-accuracy approximation.

In the next subsections, we propose our method named Component-Wise Natural Gradient Descent (CW-NGD), a novel precise and practicable approximation of the layer FIMs which also supports the inversion trivially. Specifically, subsections III-C and III-D describe the method that applies to dense layers and convolutional layers, respectively.

### C. Approximate FIM Inverse for Dense Layer

For the sake of transparent interpretation, we consider a simple dense network illustrated in Figure 1a. The original NGD algorithm requires inverting the FIM depicted in Figure 1b. Figure 1c represents the Layer-Wise NGD algorithm family’s mechanism, where they approximate the FIM as a block-diagonal matrix and inverts each diagonal block. In CW-NGD, we divide the weight vector $\theta_l$ into groups in which elements of a group share the same output node. To invert one $\theta_l$, we divide $\theta_l$ into $n_l$ groups at the $i$-th layer: $n_l = \phi_{(l,i)}$ (or $D_{\theta_{(l,i)}}$) ($i = 1, 2, \ldots, n_l$). Thanks to the definition of the vectorization function vector of the weight matrix, elements in the same group are
Fig. 1: The approximation of a dense layer’s FIM and its inverse
consecutive in the vectorized presentation of the weight. This results in \( F_{\theta,l} \) being a \( n_l \) by \( n_l \) block matrix in which the \( i \)-th row and \( j \)-th block is \( F_{\theta,l}(i, \cdot, i)(\cdot, j) \). By definition of \( F_{\theta} \) given in Eq. 1, we have:

\[
F_{\theta,l}(i, \cdot, i)(\cdot, j) = \sum_{p=1}^{P} \left( D_{\theta}^{(p)}(i, \cdot, i)^T D_{\theta}^{(p)}(\cdot, j) \right)
\]

There is a well-known assumption that holds very well in practice [32] called gradient independence assumption in the mean field theory. It is inspired by [33], first introduced in [34], and later applied in numerous studies in Deep Neural Networks [33], [32], [35]-[39]. Under this assumption, the network weight for backpropagation is assumed to be different from those used in forwarding propagation and identically independently sampled from the same distributions. That means their covariance is zero. For \( i \neq j \), we have:

\[
0 = \text{cov}(D_{\theta,l}(i, \cdot, i), D_{\theta,l}(\cdot, j))
\]

\[
0 = \mathbb{E} \left[ D_{\theta}^{(p)}(i, \cdot, i)^T \right] \mathbb{E} \left[ D_{\theta}^{(p)}(\cdot, j) \right] - \mathbb{E} \left[ D_{\theta}^{(p)}(i, \cdot, i)^T D_{\theta}^{(p)}(\cdot, j) \right]
\]

The last equality is based on the zero expectation property of the score function \( \mathbb{E} \left[ \left( D_{\theta,l}(i, \cdot, i) \right) \right] = 0 \) that satisfies under overwhelmingly the majority of the cases within the application of neural networks [41] and can be trivially proven using the Leibniz Integral Rule [42]. Evaluating the Eq. 7 with samples from dataset, we have \( \sum_{p=1}^{P} \left( D_{\theta}^{(p)}(i, \cdot, i)^T D_{\theta}^{(p)}(\cdot, j) \right) = 0 \) for \( i \neq j \). Correspondingly, off-diagonal blocks of layer FIM (Eq. 4) are zero. As a result, the dense layer’s FIM becomes a block-diagonal matrix whose inverse can be calculated by inverting its diagonal block sub-matrices within a reasonable time. Figure 1d illustrates the mechanism of CW-NGD.

\[
F_{\theta,l} = \text{diag}(F_{\theta,1}(1,1), \ldots, F_{\theta,l}(n_l, n_l))
\]

\[
F_{\theta,l}^{-1} = \text{diag}(F_{\theta,1}^{-1}(1,1), \ldots, F_{\theta,l}^{-1}(n_l, n_l))
\]

In summary, in the CW-NGD method, we apply the derivative independence assumption to pairs of edges associated with different output nodes (Figure 1c). This division is based on the observation that elements across different groups do not directly contribute to the loss function at the \( l \)-th layer stage of the calculation in \( f(x, \theta) \). To put it in another way, 2 elements belonging to 2 different groups contribute to different output nodes whose values are passed through the activation function before contributing to the same output node in the next layer. By the chaining rule which is used in backpropagation, the derivative with respect to the next layer’s output node must be multiplied with the derivative with respect to different output nodes in the current layer before being used to calculate the derivative with respect to these 2 elements. These analyses are further supported by our empirical result, which will be shown in Section V.

D. Approximating the FIM for Convolutional Layers

This subsection focuses on approximating the FIM for convolutional layers. Figure 2 exposes the structure of a sample convolutional with filter size \( 8 \times 8 \), input layer having 128 channels, output layer having 256 channels. The array of blocks in the middle of Figure 2 represents the weight of the layer consisting of 256 blocks with each block having \( 8 \times 8 \times 128 = 8192 \) scalar elements. Because convolutional layers have a different structure from dense layers, we need to adapt the notation accordingly. Most of the formulae are the same as of the dense layer case, though.

Suppose that between the \( (l-1) \)-th layer with \( c_{l-1} \) channels and the \( l \)-th layer with \( c_l \) channels, there is a 2D convolutional layer with a \( k_l \times k_l \)-size kernel. The convolutional layer has the weight tensor in a shape \( k_l \times k_l \times c_{l-1} \times c_l \) and the bias in a shape \( c_l \). Diverging from the case of dense layer, for the convolutional layer, we use the index notation \( \theta_{r,h,i,o} \) (or \( D_{\theta,r,h,i,o} \)) to refer to the element of the flattened vector \( \theta \) (or \( D_{\theta} \)) associated with the element of the weight of the \( l \)-th (convolutional) layer at \( (r,h,i,o) \) index with respect to kernels’ indices and \( i,o \) being input layer and output filter’s indices, respectively. For the bias element, we use the notations \( \theta_{l,o} \) and \( D_{\theta,l,o} \) instead.

The associated layer FIM for this convolutional layer turns in a shape \( (c_{l-1} + k_l^2 c_{l-1} - 1) \times (c_{l-1} + k_l^2 c_{l-1} - 1) \). We index this FIM using the notations \( F_{\theta_l, (l,1), (l,1), \ldots, (l,1), (l,1), (l,2), (l,2), \ldots, (l,2), (l,2)} \), \( F_{\theta_l, (l,1), (l,1), \ldots, (l,1), (l,2), (l,2)} \), \( F_{\theta_l, (l,1), (l,2), (l,2)} \), and \( F_{\theta_l, (l,1), (l,2)} \) for respective pairs of element types described using the index notation for the weight tensor \( \theta \).

The convolutional layer structure does not affect the generality of the derivative independence assumption and its derivation to the block-diagonal matrix approximation of the FIM described in Subsection III-C. All we need to do is to provide an appropriate grouping scheme for an efficient approximation. For the case of the dense layer, we divide the weight into groups by output node. In contrast, for the convolutional layer, we divide the weight and bias into groups by their output layers. In other words, \( D_{\theta,l} \) is divided into \( c_l \) groups. For \( o = 1, 2, \ldots, c_l \), the \( o \)-th group contains 1 element of \( D_{\theta,l}(o) \) and \( k_l^2 c_{l-1} \) elements from \( D_{\theta,l}(i, \cdot, o) \). In the flattened representation of the weight tensor, we place the elements in each group consecutively, beginning with the bias element. Applying the derivative independence assumption, the FIM becomes a block-diagonal matrix consisting of \( c_l \) blocks, each of which is a \( (1 + k_l^2 c_{l-1}) \times (1 + k_l^2 c_{l-1}) \) matrix. In this form, the FIM can be inverted in a reasonable time. Figure 3 shows how the FIM and its inverse are approximated for the case of the sample convolutional layer.

The division strategy for the convolutional layer is designed in the same way as the dense layer case. In which elements belonging to different groups do not directly
Each block: \(8 \times 8 \times 128 = 8192\) elements

Fig. 2: Sample CNN

Fig. 3: CW-NGD’s FIM’s approximation for CNN layers contribute to the forward propagation of the network at the same stage of the current layer. To put it another way, their values contribute to different output nodes whose values are passed through the activation function before contributing to the same output node in the next layer. Hence, in backpropagation, the derivative with respect to the next layer’s output node must be multiplied with the derivative of the activation function with respect to each output node in the current layer before being used to calculate the network loss’s derivative with respect to these 2 elements.

The mechanism of CW-NGD for convolutional layers clearly clarifies our contribution compared to the unit-wise NGD method [20], [24]. Unit-wise NGD coincides with CW-NGD in the case of dense layers. However, for convolutional layers, the layer has \(W_iH_iC_l\) output nodes and many of them share the same weights, making unit-wise NGD impossible to group the weights by output nodes for update.

On the other hand, regarding the derivative independence assumption, it is theoretically possible to apply it to all pairs of different weight elements and yield FIM in form of a scalar diagonal matrix. [13] calls this method as entry-wise NGD, evaluates it on a crafted dataset and shows that it converges faster than the pure first-order methods (stochastic gradient descent), but slower than layer-wise NGD and unit-wise NGD. In our proposed method CW-NGD, we apply the derivative independence assumption to selected pairs of elements based on the grouping strategy which is designed based on the layer structure. Additionally, we empirically show that CW-NGD produces an efficient training convergence in a more practical dataset in Section V.

E. Efficient CW-NGD Implementation and Discussion

In this subsection, we provide an efficient implementation for CW-NGD. Next, we evaluate the computational complexity gained by the approximation in CW-NGD. Finally, we discuss the parallelization capabilities of CW-NGD with SIMD and MIMD.

Pseudocode of CW-NGD is described in Algorithm 1. CW-NGD takes two variables, inputs \(X\) and labels \(Y\), and updates the network weight with our proposed method. This procedure should be repeated multiple times, where the number of times is determined by the user. We drop the bias element for simplicity. In Algorithm 1, \(P, \alpha, \gamma, I\) are batch size, scalar learning rate, scalar damping factor, and unit matrix of appropriate dimension, respectively. The notation \([i=1,...,L]\) denotes an array of \(L\) element numbered by \(l\) for example \(S\) (or \(A\)), and \(S[i]\) (or \(A[i]\)) refers to the \(l\)-th element of the array \(S\) (or \(A\)). \(\text{FORWARDPROP}(X)\) is the forward propagation of the network which returns \(L\) elements arrays of pre-activation and (post-)activation evaluation of the network. The network might contain layers of other different types (MaxPooling, Flatten, etc.), but here we focus on and denote the 2 trainable types of dense \((\mathbb{R}^{P \times n_i})\) and convolutional \((\mathbb{R}^{P \times h_i \times w_i \times c_i})\) layers.
Algorithm 1 Component-Wise Natural Gradient Descent

1: procedure CW-NGD\( (X \in \mathbb{R}^{P \times n_0}, Y \in \mathbb{R}^{P}) \)
2: \( S, A \in \bigcup_{i=1, \ldots, L} \mathbb{R}^{P \times n_i} \)
3: \( \text{ForwardProp}(X) \)
4: \( D_0 \in \mathbb{R}^{P \times n_0} \leftarrow \text{Cost}'(A_{[L]}, Y) \)
5: \( \text{delete } A_{[L]} \)
6: for \( l \) from \( L \) to \( 2 \) do
7: \( D_0 \in \mathbb{R}^{P \times n_l} \leftarrow \text{actf}'(S_{[l]}) \)
8: \( \text{BackProp}_l(D_0) \)
9: if layer\(_l\) is dense then
10: \( D_w \in \mathbb{R}^{P \times n_{l-1}} \leftarrow A_{[l-1]} \times D_s \)
11: \( F \in \mathbb{R}^{n_l \times P} \leftarrow \text{Reshape}(D_w) \)
12: \( D_w \in \mathbb{R}^{n_{l-1} \times n_l} \leftarrow \text{AVG}_{1}(D_W) \)
13: \( D' \in \mathbb{R}^{n_{l-1} \times n_l} \leftarrow \text{Reshape}(D_w) \)
14: \( U' \in \mathbb{R}^{n_{l-1}} \leftarrow D'(F + \gamma I) \)
15: \( W_l \leftarrow W_l - \alpha U \)
16: else if layer\(_l\) is convolutional then
17: \( W_w \in \mathbb{R}^{P \times k_{x} \times k_{y} \times c_{l-1} \times c_l} \leftarrow \text{Conv2DGrad}(A_{[l]}, D_s) \)
18: \( D'_w \in \mathbb{R}^{k_{x} \times k_{y} \times c_{l-1} \times c_l} \leftarrow \text{Reshape}(D_W) \)
19: \( F \in \mathbb{R}^{n_{l-1} \times k_{x} \times k_{y} \times c_{l-1}} \leftarrow \text{AVG}_{1}(D_W) \)
20: \( D_w \in \mathbb{R}^{n_{l-1} \times k_{x} \times k_{y} \times c_{l-1}} \leftarrow \text{Reshape}(D_w) \)
21: \( U' \in \mathbb{R}^{n_{l-1} \times k_{x} \times k_{y} \times c_{l-1}} \leftarrow D'(F + \gamma I) \)
22: \( W_l \leftarrow W_l - \alpha U \)
23: end if
24: delete \( A_{[l-1]}, S_{[l]} \)
25: end for
26: end procedure

only. \( \text{Cost}'(A_{[L]}, Y) \) is the derivative function of the cost function with respect to the last layer. The loop from Line 5 iterates from the last layer to the second layer, and carries out the backpropagation while inverting the FIM to precondition the gradient for updating the weights. In this for-loop, \( \text{actf}'(S_{[l]}) \) returns the Jacobian of the \( l \)-th layer’s activation function, \( \text{BackProp}_l(D_0) \) is the backpropagation of the \( l \)-th layer which returns the derivative with respect to the previous layer (the \((n-1)\)-th layer)’s (post-)activation. For matrix multiplication \((\times)\), transpose\((\top)\), and inversion\((\top^{-1})\) operations, when the operands are tensors of a dimension larger than 2, the operation is performed on the last two dimensions. For example, multiplying a \( P \times N_1 \times N_2 \) tensor with a \( P \times N_2 \times N_3 \) tensor results a \( P \times N_1 \times N_3 \) tensor by aggregating results of \( P \) multiplications of a \( N_1 \times N_2 \) matrix with a \( N_2 \times N_3 \) matrix. \( \text{AVG}_1(A) \) is the average of the \( A \) tensor along the first dimension. \( \text{Reshape}(A) \) is the reshape function that re-arranges elements of the \( A \) tensor into a tensor of a shape specified by the target shape on the left-hand side.

It is worth noting that the backpropagation (Line 7) calculates the derivative of the cost function with respect to each sample (post-)activation. This is different from the typical backpropagation in popular libraries that only supports calculating the sum of these derivatives. Besides, when taking the gradient (Line 12, Line 21), we use the average of the derivative with respect to the weight evaluated at each sample instead of the sum. The aggregation depends on how the loss across batch samples is aggregated. If the loss is averaged across samples, then the gradient should be averaged. Otherwise, when the loss is summed, the gradient should be summed as well. \( \gamma \) is a scalar damping factor that is used (Line 14, Line 23) to prevent the FIM from becoming singular by decreasing its condition number. This technique is the well-known Tikhonov regularization [43] with the Tikhonov matrix being the identity matrix multiplied by \( \gamma \). Tikhonov regularization is proven to be equivalent to inverting a sphere region called the trust region on the update vector [44]. So that, sometimes it is also called the trust region method. Tikhonov regularization is the first-to-use and very simple regularization method used in many quadratic curvature-related problems. In CW-NGD, we use a relatively small damping factor \( \gamma \) and find it highly effective in making the FIMs invertible.

Next, we discuss the computational complexity gained by the approximation in CW-NGD and the memory complexity of the implementation. The computational complexity of CW-NGD is determined by the most expensive operation of inverting the FIM. In which, for each layer, we need to invert \( n_l \) matrices of dimension \( n_l \times n_l \) (dense) or invert \( c_l \) matrices of dimension \( k_{l}c_{l-1} \times k_{l}c_{l-1} \) (convolutional). Equivalently, the overall computation complexity is \( O(Ln^2_l) + O(Lk^3_l c^3_l d^4) \), reduced \( O(\max(L^2n^2_l, L^2c^2_l)) \) times from \( O(L^3n^3_l) + O(L^3k^3_l c^3_l d^4) \) if we invert the full FIM.

From the memory perspective, we evaluate the individual FIM of each layer and update the layer’s weight during the backpropagation. We reuse many large-size variables such as \( D_w, D_w', F, D_w, D_w', U, U' \), which efficiently reduces memory usage. Additionally, we remove the pre- and post-activations of each layer immediately after use (Line 4, Line 27).

Lastly, we discuss the parallelization capabilities of CW-NGD with SIMD and MIMD. The SIMD technique can be applied on various matrix operations, especially the FIM inversion (Line 14, Line 23). Moreover, the inverting operations of the FIM for individual layers and individual components are independent. So that they can be evaluated in parallel, and the computational complexity can be further scaled down to \( Ln_1 \) parallel processes for dense layer (or \( Lc_l \) processes for convolutional layer) with each process’s computation complexity being \( O(n^2_l) \) \( O(k^3_l c^3_l d^4) \) for convolutional layer).
IV. Experiment

In this section, we describe the details of the conducted experiment including implementation framework, network architecture, dataset, training parameters, and the running environment.

To verify the effectiveness of our proposed method CW-NGD, we train the model described in Figure 4 on the MNIST dataset [27] using 3 different training methods: CW-NGD, KFAC [14], [15], and Adam [2]. The latter 2 methods are the state-of-the-art second-order and first-order optimization methods in the literature, respectively. The trained model starts with a convolutional layer and a max-pooling layer, followed by 2 dense layers. Kernel sizes, max pooling window size, and dense layer output sizes are all illustrated in Figure 4. Additionally, we use He Initialization [45] for all trainable weights.

Our implementation of CW-NGD is based on the latest TensorFlow [46] framework version 2.9.1. For Adam, we use the library call available in the framework. For KFAC, we use the implementation provided by TensorFlow team at [47] that requires Tensorflow version 1.15.

The experiment is conducted on the MNIST dataset [27] including 60,000 training samples and 10,000 testing samples, which are used to determine the training accuracy and validation accuracy described in Section V. We use the 1024 batch size and train the model in 100 epochs. In CW-NGD, we set learning rate $\alpha = 1$ and damping factor $\gamma = 0.0001$ according to Algorithm 1. For Adam, we use the parameter values and decay strategy provided in [47] tuned for the CIFAR10 dataset [48]: momentum $\beta = 0.97$, learning rate: initialized at 2.24 decaying to 0.0001 at rate 0.9994, epsilon: initialized at 0.183 decaying to $10^{-8}$ at rate 0.9998. Similarly, for KFAC, we use the parameter values and decay strategy provided in [47] tuned for the CIFAR10 dataset [48]: momentum $\beta = 0.98$, learning rate: initialized at 0.23 decaying to 0.0001 at rate 0.9989, damping factor: initialized at 0.289 decaying to $10^{-6}$ at rate 0.9997. Due to the resource constraint, we use the hyperparameter values tuned for CIFAR10 to use in this experiment which trains on the MNIST dataset for a heuristic comparison. This heuristic comparison is sufficient because the hyperparameters only affect the early iterations of the training.

Because the available KFAC implementation is implemented in the legacy Tensorflow version which requires a specific environment (CUDA 10.0, GCC 7.5.0), to simplify the experiment process, we run the experiment on 2 machines. One is equipped with a TITAN RTX GPU, and the other is equipped with a GTX 3080 GPU. The former is installed with Tensorflow version 2.9.1 (Python 3.10, CUDA 11.2, Cudnn 8.1, GCC 12.1.0) to evaluate CW-NGD and Adam, while the latter is installed with Tensorflow version 1.15 (Python 3.7, CUDA 10.0, Cudnn 7.4, GCC 7.5.0) to evaluate KFAC. In 2 machines, Tensorflow automatically selects and makes use of GPGPU for the training.

V. Results

| Optimization   | CW-NGD | Adam | KFAC |
|----------------|--------|------|------|
| 1st training iteration training accuracy | 87.97% | 85.85% | 90.32% |
| 2nd training iteration training accuracy | 98.03% | 95.89% | 95.68% |
| Number of iterations needed for convergence | 13 | 49 | 38 |
| validation accuracy | 100% | 100% | 100% |
| training accuracy | 98.96% | 98.68% | 98.32% |

TABLE I: Comparison among 3 optimization methods

In this section, we discuss the experiment results of 3 optimization methods (CW-NGD, Adam, KFAC) based on the graphs shown in Figure 5, with several statistic results in Table I.

Figure 5a gives an overview of the training accuracy of the three optimization methods. The higher curve of CW-NGD indicates that it performs better than the other two methods by converging within fewer iterations. Specifically, CW-NGD converges at the 13th iteration, while the other two methods converge at the 49th and 38th iterations (Table I). In other words, CW-NGD requires nearly one-third fewer iterations to converge than the other two methods.

Additionally, in Figure 5a, CW-NGD curve smoothly and monotonically rises without fluctuating. This result can be partially explained by the absence of decaying strategy in CW-NGD training. In contrast, the other two methods’ graphs fluctuate due to the involvement of decaying strategies. CW-NGD achieves a higher training accuracy than the other two methods even before the first fluctuation, implying that CW-NGD produces a higher precondition for the gradient update compared to the other two methods. Table I shows that CW-NGD starts with a lower training accuracy than KFAC, better than Adam, then immediately gets a better training accuracy than both from the second iteration.

Regarding the validation accuracy, Figure 5b clearly indicates that CW-NGD outperforms the other two methods. To be precise, CW-NGD’s final validation accuracy is

Fig. 4: Model used to train
CW-NGD outperforms KFAC and Adam, 2 state-of-the-art second-order and first-order optimization methods in the literature, respectively. Specifically, CW-NGD only requires one-third of the iterations required by KFAC or Adam to converge.

In another experiment, we compare the validation accuracy of the 3 methods. The results reveal that CW-NGD clearly has the highest validation accuracy.

From the performance perspective, theoretically, the matrix inversion of the FIM can be divided into independent subproblems corresponding to the individual FIM of a component, each of which can be solved by a single process. This indicates the potential for a high parallelization level of CW-NGD. In the future, we will investigate a distributed solution of CW-NGD. Additionally, we will also extend CW-NGD to support layers other than dense and convolution. Besides, we are going to evaluate CW-NGD on more sophisticated datasets such as CIFAR10 [48], CIFAR100 [48], ImageNet [49], etc.

VI. Conclusion

In this work, we propose a novel method, named Component-Wise Natural Gradient Descent (CW-NGD), for training neural networks. CW-NGD is a second-order optimization that works by approximating the Fisher Information Matrix (FIM) required during the training as a block-diagonal matrix. Each block matrix in the diagonal corresponds to the FIM of a group of network weights in a layer. We introduce grouping strategies for CW-NGD that efficiently works on dense and convolutional layers. Our investigations show that CW-NGD runs in a reasonable time while preserving the accuracy of the model at a high level.

We provide the detail of an efficient implementation of CW-NGD that generates the FIM, inverts it, and updates the weights while doing the backpropagation. The implementation also reduces memory usage by reusing several large-size variables.

In an experiment that compares the number of iterations required for training convergence, we show that CW-NGD outperforms KFAC and Adam, 2 state-of-the-art second-order and first-order optimization methods in the literature, respectively. Specifically, CW-NGD only requires one-third of the iterations required by KFAC or Adam to converge.

In another experiment, we compare the validation accuracy of the 3 methods. The results reveal that CW-NGD clearly has the highest validation accuracy.

From the performance perspective, theoretically, the matrix inversion of the FIM can be divided into independent subproblems corresponding to the individual FIM of a component, each of which can be solved by a single process. This indicates the potential for a high parallelization level of CW-NGD. In the future, we will investigate a distributed solution of CW-NGD. Additionally, we will also extend CW-NGD to support layers other than dense and convolution. Besides, we are going to evaluate CW-NGD on more sophisticated datasets such as CIFAR10 [48], CIFAR100 [48], ImageNet [49], etc.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, “Identity mappings in deep residual networks,” in Computer Vision – ECCV 2016. Springer International Publishing, 2016.
[2] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.
[3] K. S. Geoffrey Hinton, Nitish Srivastava, Neural networks for machine learning, overview of mini-batch gradient descent. [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
[4] M. D. Zeiler, “Adadelta: An adaptive learning rate method,” ArXiv, 2012.
[5] J. Duch, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” J. Mach. Learn. Res., 2011.
[6] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in Proceedings of the 30th International Conference on Machine Learning, 2013.
[7] S. Li, D. Li, and Y. Zhang, “Incorporating nesterov’s momentum into distributed adaptive gradient method for online optimization,” in 2021 China Automation Congress (CAC), 2021.
[8] S.-i. Amari, “Natural Gradient Works Efficiently in Learning,” Neural Computation, 1998.
[9] L. LeCam, Asymptotic methods in statistical decision theory, ser. Springer series in statistics. New York, NY [u.a.]: Springer, 1986.
[10] A. Bernacchia, M. Lengyel, and G. Hennquin, “Exact natural gradient in deep linear networks and its application to the nonlinear case,” in Advances in Neural Information Processing Systems, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/7818e7b301a66568981c6b9339f6e8-Paper.pdf
[11] G. Zhang, J. Martens, and R. B. Grosse, “Fast convergence of natural gradient descent for over-parameterized neural networks,” in Advances in Neural Information Processing Systems, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/1da546f25222c1ee710ce7c7f7ac380c-Paper.pdf
