Efficient Riemannian Meta-Optimization by Implicit Differentiation

Xiaomeng Fan,[1] Yuwei Wu,[1]* Zhi Gao,[1] Yunde Jia,[1] Mehrtash Harandi[2]

[1] Beijing Laboratory of Intelligent Information Technology
School of Computer Science, Beijing Institute of Technology, Beijing, China
[2] Department of Electrical and Computer Systems Eng., Monash University, and Data61, Australia
{fanxiaomeng,gaozhi_2017,wuyuwei,jiayunde}@bit.edu.cn, mehrtash.harandi@monash.edu

Abstract

To solve optimization problems with nonlinear constrains, the recently developed Riemannian meta-optimization methods show promise, which train neural networks as an optimizer to perform optimization on Riemannian manifolds. A key challenge is the heavy computational and memory burdens, because computing the meta-gradient with respect to the optimizer involves a series of time-consuming derivatives, and stores large computation graphs in memory. In this paper, we propose an efficient Riemannian meta-optimization method that decouples the complex computation scheme from the meta-gradient. We derive Riemannian implicit differentiation to compute the meta-gradient by establishing a link between Riemannian optimization and the implicit function theorem. As a result, the updating our optimizer is only related to the final two iterations, which in turn speeds up our method and reduces the memory footprint significantly. We theoretically study the computational load and memory footprint of our method for long optimization trajectories, and conduct an empirical study to demonstrate the benefits of the proposed method. Evaluations of three optimization problems on different Riemannian manifolds show that our method achieves state-of-the-art performance in terms of the convergence speed and the quality of optima.

Introduction

In science and engineering, many tasks are modeled as optimization problems with nonlinear constraints (Absil, Mahony, and Sepulchre 2009), including principal component analysis (Liu et al. 2017) and matrix completion with orthogonality constraints (Dai, Milenkovic, and Kerman 2011), and similarity learning with positive definite constraints (Harandi, Salzmann, and Hartley 2017). The primary way to address optimization problems with nonlinear constraints is to formulate them on Riemannian manifolds, and utilize Riemannian optimization algorithms to maintain faithful to the geometry of constraints. Bonnabel (Bonnabel 2013) proposed Riemannian stochastic gradient descent algorithm, and Kasai et al. (Kasai, Jawanpuria, and Mishra 2019) proposed Riemannian adaptive optimization algorithms.

*Corresponding author:Yuwei Wu

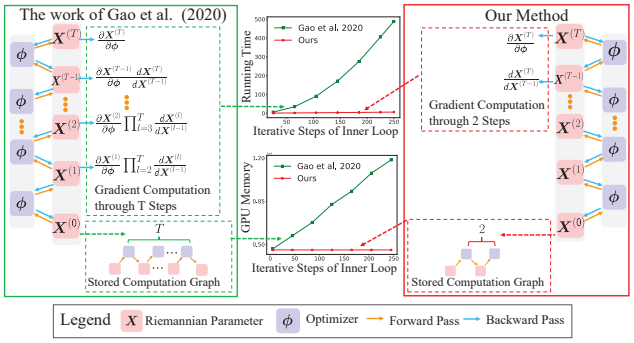

Figure 1: We measure the time and memory consumption of the Riemannian meta-optimization method (Gao et al. 2020) and our method. The method (Gao et al. 2020) has heavy computational and memory burdens with the increase of the number of steps in the inner-loop. In contrast, our method reduces computational cost and memory footprint significantly. This is because the method (Gao et al. 2020) differentiates through the whole inner-loop optimization to compute the meta-gradient, which involves a series of time-consumption derivatives and stores a large computation graph, while our meta-gradient computation is only related to the final two iterations.

Recently, Riemannian meta-optimization methods (Gao et al. 2020; Fan et al. 2021) have shown promise in solving Riemannian optimization problems. In contrast to previous hand-designed Riemannian optimizers, Riemannian meta-optimization methods utilize meta-learning to automatically learn an optimizer in a data-driven fashion. Concretely, the conventional Riemannian gradient descent is formulated as

$$\boldsymbol{X}^{(t+1)} = \Gamma_{\boldsymbol{X}^{(t)}}\left(-\xi \cdot \pi_{\boldsymbol{X}^{(t)}}(\nabla \boldsymbol{X}^{(t)})\right) .$$

Here, $\boldsymbol{X}^{(t)}$ represents the Riemannian parameter of interest evaluated at time t, $\nabla \boldsymbol{X}^{(t)}$ is the gradient of the loss with respect to $\boldsymbol{X}^{(t)}$, and ξ is the stepsize. $\Gamma_{\boldsymbol{X}^{(t)}}(\cdot)$ and $\pi_{\boldsymbol{X}^{(t)}}(\cdot)$ are the Riemannian operations that are used to obtain the update Riemannian parameter and search direction, respectively. Riemannian meta-optimization methods aim to improve the speed and quality of the solution by rewriting the

optimization as

$$\boldsymbol{X}^{(t+1)} = \Gamma_{\boldsymbol{X}^{(t)}}\left(g_{\boldsymbol{\phi}}(\nabla \boldsymbol{X}^{(t)}, \boldsymbol{S}^{(t-1)})\right),$$

where $g_\phi(\cdot)$ is a mapping that corrects the gradient based on the distribution of data, and $\boldsymbol{S}^{(t-1)}$ is the optimization state at time $t-1$. The mapping $g_\phi(\cdot)$ is a neural network that needs to be learned, parameterized by ϕ. The Riemannian meta-optimization methods not only can obtain a promising Riemannian optimizer via exploring the underlying data distribution but also reduce human involvements in designing the optimizer.

Despite the success, training such Riemannian optimizers brings heavy computational and memory burdens. Existing Riemannian meta-optimization methods are cast as bi-level optimization procedures, and use inner-loop and outer-loop optimization stages to train the optimizer (Metz et al. 2019; Chen et al. 2020). The inner-loop is similar to a traditional Riemannian optimization process, where the learnable optimizer updates Riemannian parameters for the target task iteratively. In the outer-loop, the optimizer is trained by minimizing the loss of updated Riemannian parameters, where computing the meta-gradient with respect to the optimizer needs to differentiate through *the whole* computation graph of the inner-loop. This causes a lot of memory to store the computational graph of the entire inner-loop , and involves time-consuming Hessian matrices and derivatives of Riemannian operations (*e.g.*, retraction operation) in the entire inner-loop, as shown in Figure 1.

To address this issue, we propose an efficient Riemannian meta-optimization method that decouples the meta-gradient computation from the inner-loop optimization. Specifically, we derive Riemannian implicit differentiation for Riemannian meta-optimization by connecting Riemannian optimization with implicit function theorem (Lorraine, Vicol, and Duvenaud 2020; Rajeswaran et al. 2019). Through the Riemannian implicit differentiation, computing the meta-gradient with respect to the optimizer in our method is independent of the entirety of the inner-loop optimization, and is only related to the final two iterations of it, reducing the memory and computational complexity significantly. We demonstrate theoretically and empirically that our method only needs a small constant memory and computational cost, regardless of the length of the optimization trajectory in the inner-loop. This is because our method does not need to store and differentiate through the whole inner-loop optimization. Moreover, evaluations of three tasks on different Riemannian manifolds show that our method can learn a competitive (and even better) Riemannian optimizer with faster convergence speed and lower loss values than existing Riemannian optimization methods. The code is available at https://github.com/XiaomengFanmcislab/I-RMM.

In summary, our contributions are two-fold.

(1) We propose an efficient Riemannian meta-optimization method that avoids to store and differentiate through the entirety of the inner-loop optimization. Compared with existing Riemannian meta-optimization methods, our method not only requires far less computational resources but also learns a better optimizer.

(2) We derive Riemannian implicit differentiation for Riemannian meta-optimization, through which the meta-gradient computation is only related to the final two iterations of the inner-loop optimization, instead of the whole procedure.

Related Work

Riemannian optimization

Luenberger (Luenberger 1972) proposed the first Riemannian gradient descent (RGD) approach that formulates the constrained optimization problems on Riemannian manifolds and utilizes Riemannian operations to preserve constraints on the manifold. After that, many efforts have been made to develop more powerful Riemannian optimization method. Bonnabel (Bonnabel 2013) extended RGD to the stochastic setting, Zhang and Sra (Zhang and Sra 2018) proposed accelerated optimization methods on Riemannian manifolds, Liu *et al.* (Liu et al. 2017) developed Riemannian stochastic variance reduced gradient (RSVRG) algorithm, and Kasai *et al.* (Kasai, Sato, and Mishra 2018) introduced adaptive optimization approaches in Riemannian manifolds.

Recently, Riemannian meta-optimization methods provide a promising way to address Riemannian optimization problems. Gao *et al.* (Gao et al. 2020) introduced a matrix LSTM as the optimizer that takes the gradient as input and generates stepsize and search directions for symmetric positive definite manifolds. However, training the optimizer brings the exploding gradient problem. To solve this issue, they proposed a gradient-free optimizer on tangent spaces for Riemannian optimization (Fan et al. 2021), which removes gradients computation with respect to Riemannian parameters in the inner-loop optimization. Despite the success, existing Riemannian meta-optimization methods bring heavy computational and memory burdens, since they store and differentiate through their whole inner-loop optimization. In contrast, our Riemannian meta-optimization method uses Riemannian implicit differentiation and only differentiates through the final two iterations of the inner-loop optimization, reducing the memory and computational cost significantly.

Implicit Differentiation

Implicit differentiation has been well applied to bi-level optimization problems, such as hyperparameter optimization (Lorraine, Vicol, and Duvenaud 2020; Navon et al. 2020; Bertrand et al. 2020; Liao et al. 2018; Gudovskiy et al. 2021) and meta-learning (Rajeswaran et al. 2019). By utilizing the implicit function theorem (Larsen et al. 1996; Bengio 2000), implicit differentiation is usually used to efficiently compute the gradient with respect to outer-loop parameters, which avoids differentiating through the inner-loop optimization. While implicit differentiation is successful in many applications, it has not yet been applied to meta-optimization (learning to optimize) (Andrychowicz et al. 2016) to the best of our knowledge. A possible reason is that, their scheme that uses the final step to compute outer-loop gradients is not suitable for the meta-optimization formulation, which makes

computing meta-gradients infeasible. In our method, we derive novel implicit differentiation for meta-optimization by using the final two iterations rather than only the last one iteration. Besides, we extend the implicit differentiation to Riemannian manifolds, through which we can efficiently solve challenging Riemannian optimization problems.

Analysis of Riemannian Meta-Optimization

Riemannian Meta-Optimization

A smooth Riemannian manifold $\mathcal{M}$ is a topological space that is locally Euclidean space (Absil, Mahony, and Sepulchre 2009). For a point $\boldsymbol{X} \in \mathcal{M}$, its tangent space is denoted by $T_{\boldsymbol{X}}\mathcal{M}$, being the set of all tangent vectors to $\mathcal{M}$ at $\boldsymbol{X}$. Due to the nonlinear nature of Riemannian manifolds, gradient-based Riemannian optimization uses retraction and orthogonal projection to preserve the manifold constraints. The retraction operation $\Gamma_{\boldsymbol{X}}(\boldsymbol{P}) : T_{\boldsymbol{X}}\mathcal{M} \rightarrow \mathcal{M}, \boldsymbol{P} \in T_{\boldsymbol{X}}\mathcal{M}$ is a smooth mapping from the tangent space $T_{\boldsymbol{X}}\mathcal{M}$ onto the manifold $\mathcal{M}$ with a local rigidity condition (Absil, Mahony, and Sepulchre 2009). The orthogonal projection $\pi_{\boldsymbol{X}}(\nabla \boldsymbol{X})$ transforms an arbitrary Euclidean gradient $\nabla \boldsymbol{X}$ into the tangent space $T_{\boldsymbol{X}}\mathcal{M}$.

Riemannian meta-optimization methods utilize neural networks to parameterize a Riemannian optimizer $g_{\boldsymbol{\phi}}$, where $\boldsymbol{\phi}$ is the parameter of neural networks. The optimizer automatically produces the stepsize $\xi^{(t)}$ and the search direction $\boldsymbol{\eta}^{(t)}$ for optimization,

$$\begin{cases} \xi^{(t)}, \boldsymbol{\eta}^{(t)} = -g_{\boldsymbol{\phi}}(\nabla \boldsymbol{X}^{(t)}, \boldsymbol{S}^{(t-1)}) \\ \boldsymbol{Y}^{(t)} = -\xi^{(t)} \boldsymbol{\eta}^{(t)} \\ \boldsymbol{X}^{(t+1)} = \Gamma_{\boldsymbol{X}^{(t)}} \left(\boldsymbol{Y}^{(t)} \right) \end{cases}, \tag{1}$$

where $\nabla \boldsymbol{X}^{(t)}$ is the gradient of the loss with respect to the Riemannian parameter, $\boldsymbol{Y}^{(t)}$ is the update vector on the tangent space, and $\boldsymbol{S}^{(t-1)}$ is the optimization state at time $t-1$. Existing Riemannian meta-optimization methods are modeled as a bi-level optimization procedure to train the optimizer, where inner-loop and outer-loop optimization stages are used. In the inner-loop, the Riemannian parameter is updated via Eq. (1). Suppose there are T steps in the inner-loop, in one iteration of the outer-loop, the optimizer is trained by minimizing the meta-objective

$$\min_{\boldsymbol{\phi}} \mathcal{J}(\boldsymbol{\phi}) \triangleq \mathcal{L}_V(\boldsymbol{X}^{(T)}), \tag{2}$$

where $\mathcal{L}_V(\boldsymbol{X}^{(T)})$ is the loss function of the updated Riemannian parameter $\boldsymbol{X}^{(T)}$ on validation data. In this case, the parameter $\boldsymbol{\phi}$ of the Riemannian optimizer is updated by

$$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \frac{d\mathcal{L}_V}{d\boldsymbol{\phi}}. \tag{3}$$

Heavy computation and memory burden

In the outer-loop, the meta-gradient $\frac{d\mathcal{L}_V}{d\boldsymbol{\phi}}$ is calculated to update the parameter $\boldsymbol{\phi}$ of the optimizer. We use d to denote the total derivative and ∂ denote partial derivative. The meta-gradient is given by

$$\frac{d\mathcal{L}_V}{d\boldsymbol{\phi}} = \frac{d\mathcal{L}_V}{d\boldsymbol{X}^{(T)}} \frac{d\boldsymbol{X}^{(T)}}{d\boldsymbol{\phi}}, \tag{4}$$

where $\frac{d\mathcal{L}_V}{d\boldsymbol{X}^{(T)}}$ is computed using differentiation easily. The derivative $\frac{d\boldsymbol{X}^{(T)}}{d\boldsymbol{\phi}}$ needs to differentiate through the whole inner-loop optimization, that is,

$$\frac{d\boldsymbol{X}^{(T)}}{d\boldsymbol{\phi}} = \sum_{k=1}^{T} \left(\frac{\partial \boldsymbol{X}^{(k)}}{\partial \boldsymbol{Y}^{(k-1)}} \cdot \frac{\partial \boldsymbol{Y}^{(k-1)}}{\partial \boldsymbol{\phi}} \prod_{l=k+1}^{T} \left(\frac{\partial \boldsymbol{X}^{(l)}}{\partial \boldsymbol{X}^{(l-1)}} + \frac{\partial \boldsymbol{X}^{(l)}}{\partial \boldsymbol{Y}^{(l-1)}} \cdot \frac{\partial \boldsymbol{Y}^{(l-1)}}{\partial \nabla \boldsymbol{X}^{(l-1)}} \cdot \nabla^2 \boldsymbol{X}^{(l-1)} \right) \right). \tag{5}$$

The computational graph is shown in Figure 1.

Apparently, Eq. (5) includes products of Hessian matrices $\nabla^2 \boldsymbol{X}^{(l-1)}$ and partial derivatives $\frac{\partial \boldsymbol{X}^{(l)}}{\partial \boldsymbol{X}^{(l-1)}}$ and $\frac{\partial \boldsymbol{X}^{(l)}}{\partial \boldsymbol{Y}^{(l-1)}}$ of the retraction operation, over all inner-loop optimization steps. Differentiating through the retraction operation and calculating Hessian matrix impose heavy computational loads. Furthermore, Eq. (5) depends on the whole inner-loop optimization path explicitly, which is completely stored in memory. The time and space complexity of Eq. (5) is proportional to the inner-loop optimization length (which will be proved in the next section). Therefore, choosing the inner-loop optimization length faces a well-known dilemma(Metz et al. 2019; Wu et al. 2018; Chen et al. 2020): a short optimization length results in instability and poor-quality optimizers, while a long optimization length inevitably causes intractable computational and memory burdens.

Our Method

In this paper, we derive Riemannian implicit differentiation for computing the meta-gradient, which does not store and differentiate through the whole inner-loop optimization, reducing much memory and time consumption.

Riemannian Implicit Differentiation

The target of Riemannian implicit differentiation is to compute the meta-gradient $\frac{d\mathcal{L}_V}{d\boldsymbol{\phi}}$ implicitly, which is independent of the whole procedure of the inner-loop. Suppose that the exact solution $\boldsymbol{X}^*$ is obtained after a optimization steps in the inner-loop, *i.e.*, $\frac{d\mathcal{L}_T}{d\boldsymbol{X}^*} = 0$, and $\mathcal{L}_T$ is the loss function on the training data. We have

$$\frac{d}{d\boldsymbol{\phi}} \left(\frac{d\mathcal{L}_T}{d\boldsymbol{X}^*} \right) = \frac{d^2\mathcal{L}_T}{d\boldsymbol{X}^* d\boldsymbol{X}^{*\top}} \frac{d\boldsymbol{X}^*}{d\boldsymbol{\phi}} = 0. \tag{6}$$

We assume that the loss function $\mathcal{L}_T(\cdot)$ is a strictly convex function. Then the Hessian matrix of the loss function $\mathcal{L}_T(\cdot)$ is a symmetric positive definite matrix, $\frac{d^2\mathcal{L}_T}{d\boldsymbol{X}^* d\boldsymbol{X}^{*\top}} \neq 0$, and thus the derivative Jacobian $\frac{d\boldsymbol{X}^*}{d\boldsymbol{\phi}} = 0$. Recall that in Riemannian meta-optimization, the meta-gradient with respect to parameter $\boldsymbol{\phi}$ of our optimizer is given by

$$\frac{d\mathcal{L}_V}{d\boldsymbol{\phi}} = \frac{d\mathcal{L}_V}{d\boldsymbol{X}^*} \frac{d\boldsymbol{X}^*}{d\boldsymbol{\phi}}. \tag{7}$$

Substituting $\frac{d\boldsymbol{X}^*}{d\boldsymbol{\phi}} = 0$ into Eq. (7) makes computing the meta-gradient $\frac{d\mathcal{L}_V}{d\boldsymbol{\phi}}$ infeasible.

To solve this issue, we utilize $\boldsymbol{X}^{*'}$ that is the Riemannian parameter after $a-1$ steps in the inner-loop to compute the meta-gradient,

$$\frac{d\mathcal{L}_V}{d\boldsymbol{\phi}} = \frac{d\mathcal{L}_V}{d\boldsymbol{X}^{*'}}\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}}. \tag{8}$$

The derivative Jacobian $\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}}$ is computed by the Riemannian implicit differentiation in Theorem 1.

Theorem 1. *If $\boldsymbol{X}^*$ is the exact solution in the inner-loop, the derivative Jacobian $\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}}$ can be computed implicitly by*

$$\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}} = -\Big(\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{X}^{*'}} + \frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\frac{\partial \boldsymbol{Y}^{*'}}{\partial \nabla \boldsymbol{X}^{*'}}\nabla^2 \boldsymbol{X}^{*'}\Big)^{-1}\cdot\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\cdot\frac{\partial \boldsymbol{Y}^{*'}}{\partial \boldsymbol{\phi}}. \tag{9}$$

Proof. From the chain rule, the derivative $\frac{d\boldsymbol{X}^*}{d\boldsymbol{\phi}}$ is given by

$$\frac{d\boldsymbol{X}^*}{d\boldsymbol{\phi}} = \frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{X}^{*'}}\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}} + \frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\frac{d\boldsymbol{Y}^{*'}}{\partial \boldsymbol{\phi}}, \tag{10}$$

where

$$\frac{d\boldsymbol{Y}^{*'}}{d\boldsymbol{\phi}} = \frac{\partial \boldsymbol{Y}^{*'}}{\partial \boldsymbol{\phi}} + \frac{\partial \boldsymbol{Y}^{*'}}{\partial \nabla \boldsymbol{X}^{*'}}\nabla^2 \boldsymbol{X}^{*'}\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}}. \tag{11}$$

We substitute Eq.(11) into Eq.(10), and have

$$\frac{d\boldsymbol{X}^*}{d\boldsymbol{\phi}} = \left[\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{X}^{*'}} + \frac{\partial \boldsymbol{Y}^{*'}}{\partial \nabla \boldsymbol{X}^{*'}}\nabla^2 \boldsymbol{X}^{*'}\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}}\right]\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}} + \frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\frac{\partial \boldsymbol{Y}^{*'}}{\partial \boldsymbol{\phi}}. \tag{12}$$

Due to the derivative $\frac{d\boldsymbol{X}^*}{d\boldsymbol{\phi}}$ equals to 0, we have

$$\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}} = -\Big(\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{X}^{*'}} + \frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\frac{\partial \boldsymbol{Y}^{*'}}{\partial \nabla \boldsymbol{X}^{*'}}\nabla^2 \boldsymbol{X}^{*'}\Big)^{-1}\cdot\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\cdot\frac{\partial \boldsymbol{Y}^{*'}}{\partial \boldsymbol{\phi}}.$$

□

The derived implicit derivative Jacobian $\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}}$ only depends on the final two iterations of the inner-loop optimization, rather than the whole inner-loop procedure. The term $\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{X}^{*'}}$ and $\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}$ are derivatives of the retraction operation. It is notable that the implicit derivative Jacobian $\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}}$ in Theorem 1 needs a matrix inversion (over the combination of Jacobian and Hessian matrix $\Big(\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{X}^{*'}} + \frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\frac{\partial \boldsymbol{Y}^{*'}}{\partial \nabla \boldsymbol{X}^{*'}}\nabla^2 \boldsymbol{X}^{*'}\Big)^{-1}$). This is non-trivial to compute, matrix inversion can become intractable for big matrices. In order to address this problem, we utilize Neumann series (Liao et al. 2018; Shaban et al. 2019) to rewrite the inverse as

$$\begin{aligned}&\Big(\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{X}^{*'}} + \frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\frac{\partial \boldsymbol{Y}^{*'}}{\partial \nabla \boldsymbol{X}^{*'}}\nabla^2 \boldsymbol{X}^{*'}\Big)^{-1}\\ =&\sum_{k=0}^{\infty}\left(\boldsymbol{I} - \Big(\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{X}^{*'}} + \frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\frac{\partial \boldsymbol{Y}^{*'}}{\partial \nabla \boldsymbol{X}^{*'}}\nabla^2 \boldsymbol{X}^{*'}\Big)\right)^k.\end{aligned} \tag{13}$$

We can approximate the inverse by the first K terms in this infinite sum. Then, we utilize Jacobian-vector product and Hessian-vector product (Baydin et al. 2018; Christianson 1992; Griewank and Walther 2008) to compute Eq. (13) instead of computing Jacobian and Hessian matrix directly. Specifically, we initialize two intermediate variable $\boldsymbol{v}^0$ and $\boldsymbol{p}^0$ as the identity matrix. For every iterations $(i = 0, \cdots, K-1)$, we update the intermediate vectors as follows:

$$\begin{cases}\boldsymbol{v}^{(i+1)} = \boldsymbol{v}^{(i)} - \boldsymbol{v}^{(i)}\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{X}^{*'}} - \boldsymbol{v}^{(i)}\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\frac{\partial \boldsymbol{Y}^{*'}}{\partial \nabla \boldsymbol{X}^{*'}}\nabla^2 \boldsymbol{X}^{*'}\\ \boldsymbol{p}^{(i+1)} = \boldsymbol{p}^{(i)} + \boldsymbol{v}^{(i+1)}\end{cases}. \tag{14}$$

After the K iterations, the derivative Jacobian is given by

$$\frac{d\boldsymbol{X}^{*'}}{d\boldsymbol{\phi}} = -\boldsymbol{p}^{(K)}\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\cdot\frac{\partial \boldsymbol{Y}^{*'}}{\partial \boldsymbol{\phi}}. \tag{15}$$

Because the intermediate variables $\boldsymbol{v}$ and $\boldsymbol{p}$ keep being the vector in the aforementioned iterations, Eq. (14) and Eq. (15) only needs to compute Jacobian-vector product and Hessian-vector product that is much easier than the original formula in Eq. (9). Substituting Eq. (15) into Eq. (8), the meta-gradient is

$$\frac{d\mathcal{L}_V}{d\boldsymbol{\phi}} = -\frac{d\mathcal{L}_V}{d\boldsymbol{X}^{*'}}\boldsymbol{p}^{(K)}\frac{\partial \boldsymbol{X}^*}{\partial \boldsymbol{Y}^{*'}}\cdot\frac{\partial \boldsymbol{Y}^{*'}}{\partial \boldsymbol{\phi}}. \tag{16}$$

Implemention

Optimizer Architecture Due to the matrix structure of Riemannian parameters, parameterizing the Riemannian optimzer by conventional neural network may inevitably destroy the matrix structure. In this paper, the generalized matrix LSTM (gmLSTM) (Fan et al. 2021) is used to parameterize the Riemannian optimizer. We leverage two gmLSTM models to compute search direaction $\boldsymbol{\eta}$ and stepsize ξ, respectively.

Algorithm 1 Parameter Warmup stage

Require: Initial Riemannian parameter $\boldsymbol{X}$, the empty parameter pool Φ, the maximum size of parameter pool L, and threshold of the gradient norm ϵ.

Randomly select a hand-designed Riemannian optimization method from RSGD, RSGDM, RSRG, and RASA.
while the size of current parameter pool Φ is not reach the maximum size L **do**
  Compute the loss $L_T(\boldsymbol{X})$ on training data.
  Compute the gradient $\nabla \boldsymbol{X}$.
  if $\|\nabla \boldsymbol{X}\| \leq \epsilon$ **then**
    Push the parameter $\boldsymbol{X}$ into the parameter pool Φ.
  end if
  Update the parameter $\boldsymbol{X}$ by using the selected hand-designed optimizer.
end while
Return parameter pool Φ.

Parameter Warmup Actually, it is difficult to optimize the Riemannian parameter from scratch to an exact solution utilizing an untrained Riemannian optimizer. In order to handle this issue, we introduce a warmup scheme that stores some good solutions as initial Riemannian parameters in advance. Specifically, before training the Riemannian optimizer, we utilize a hand-designed Riemannian optimizer such as RSGD to obtain solutions whose gradient norms are smaller than a small threshold, and put them into a parameter pool. In the training stage, we randomly sample initial Riemannian parameters from the parameter pool to learn our optimizer. The process of the proposed warmup scheme is summarized in Algorithm 1.

Algorithm 2 Training our optimizer

Require: Initial optimization state $\boldsymbol{S}^{(0)} = \mathbf{0}$, initial parameters $\boldsymbol{\phi}$ of our optimizer, maximum iteration T of the inner-loop, maximum iteration Υ of the outer-loop, and hyperparameter $\mathcal{B}$ to update the parameter pool.

$\tau = 0$.
while $\tau \leq \Upsilon$ **do**
  if τ mod $\mathcal{B}$ **then**
    Construct the parameter pool $\boldsymbol{\Phi}$ by using the warmup scheme in algorithm 1.
  end if
  Randomly select $\boldsymbol{X}^{(0)}$ from the parameter pool $\boldsymbol{\Phi}$, and set $t = 0$.
  while $t \leq T$ **do**
    Compute the loss on training dataset $\mathcal{L}_T(\boldsymbol{X}^{(t)})$.
    Compute the gradient $\nabla \boldsymbol{X}^{(t)} = \frac{d\mathcal{L}_T}{d\boldsymbol{X}^{(t)}}$.
    Update $\boldsymbol{X}^{(t)}$ by our optimizer $g_{\boldsymbol{\phi}}$ via Eq. (1).
    $t \leftarrow t + 1$.
  end while
  Compute the loss on validation dataset $\mathcal{L}_V(\boldsymbol{X}^{(T-)})$.
  Compute the implicit gradient by Eq. (16).
  Update parameter of our optimizer $\boldsymbol{\phi}$ by Eq. (3).
  $\tau \leftarrow \tau + 1$.
end while
Return the parameter $\boldsymbol{\phi}$ of our optimizer.

Training In each step of the outer-loop, we randomly select a Riemannian parameter from the parameter pool and denote it as $\boldsymbol{X}^{(0)}$. In practice, the number of iterations in the inner-loop to obtain the exact solution is unknown, and it may be different for different initialization and tasks. For simplicity, we set a fixed number T in the inner-loop, and use our Riemannian optimizer to update $\boldsymbol{X}^{(0)}$ for T iterations to obtain an approximate solution $\boldsymbol{X}^{(T)}$. Then, the optimizer is updated by using $\boldsymbol{X}^{(T)}$ and $\boldsymbol{X}^{(T-1)}$ via Eq. (16). To avoid overfitting, after $\mathcal{B}$ steps in the outer-loop, we update the parameter pool by utilizing another hand-designed Riemannian optimizer (*e.g.* RSGDM) to put new initial parameters into it. The training process of the optimizer is summarized in Algorithm 2.

Complexity

Compuatational Complexity

Some works (Griewank 1993; Griewank and Walther 2008) show that the time of computing gradients or Jacobian-vector products of a differentiable function in time is no more than a factor of 5 of the time it takes to compute that function itself, and the time of computing Hessian-vector products is also no more than 5 times of time to compute the gradient. In the computational complexity, we only keep the highest term for simplicity.

By utilizing the above two principles, for a parameter with the size of $p \times d$, the computational complexity of the implicit meta-gradient on the Grassmann manifold is no more than $O((10K+5)p^3+(125K+105)p^2d+(10K+5)pd^2)$, while the computational complexity of the work (Gao et al. 2020) is $O(5T^2p^3 + (\frac{125}{2}T^2 + \frac{75}{2}T + 5)p^2d + 5T^2pd^2)$.

As to a parameter with the size of $p \times d$ on the Stiefel manifold, the computational complexity of the implicit meta-gradient is no more than $O((90K + 90)p^2d + (35K + 25)pd^2 + (30K + 15)d^3)$, while the computational complexity of the work (Gao et al. 2020) is $O((45T^2 + 45T)p^2d + (\frac{35}{2}T^2 + \frac{15}{2}T)pd^2 + 15T^2d^3)$.

For a parameter with the size of $d \times d$ on the SPD manifold, the computational complexity of our implicit meta-gradient is no more than $O((585K + 300)d^3)$, while the computational complexity of the work (Gao et al. 2020) is $O((\frac{585}{2}T^2 - \frac{95}{2}T + 55)d^3)$.

Apparently, the computational complexity of our method is independent of the maximum iteration T of the inner-loop, while that of Riemannian meta-optimization approach quadratically related with T. Thus, with the increase of iteration steps in the inner-loop, training time of existing Riemannian meta-optimization methods increases siginificantly, while the time consumption of our method to calculate the meta-gradient is constant and very small. Though our method is linearly related to iteration of approximate Neumann series K, K is far less than T. Thus, our approach reduced computational time siginificantly compared to existing Riemannian meta-optimization approach.

Memory Cost

Because the implicit meta-gradient only depends on the final two steps of inner-loop optimization, the memory cost of our implicit Riemannian meta-optimization is $O(4dp + H)$, where H is the size of parameters of our Riemannian optimizer. The memory cost of the Riemannian meta-optimization method (Gao et al. 2020) is $O(3Tdp + H)$, since this method stores the whole inner-loop optimization in computing the meta-gradient by Eq.(5).

Experiments

Setting

In this section, we compared our optimizer with hand-designed optimizers: RSGD (Bonnabel 2013), RSVRG (Zhang, Reddi, and Sra 2016), RSRG (Kasai, Sato, and Mishra 2018), RSGDM (Kumar, Mhammedi, and Harandi 2018), and RASA (Kasai, Jawanpuria, and Mishra

Inner Loop Steps	5	25	45	65	85	105	125	145	165	185	205	225	245	250
RMM (Gao et al. 2020)	3.80×10^{-1}	5.74	1.74×10^{1}	3.51×10^{1}	6.00×10^{1}	9.05×10^{1}	1.27×10^{2}	1.71×10^{2}	2.20×10^{2}	2.76×10^{2}	3.38×10^{2}	4.05×10^{2}	4.78×10^{2}	4.88×10^{2}
GF-RMM (Fan et al. 2021)	3.01×10^{-1}	7.75×10^{-1}	1.26	1.74	2.35	2.87	3.41	3.90	4.45	4.95	5.50	6.70	7.31	7.40
Ours	2.90×10^{-1}	7.70×10^{-1}	1.22	1.69	2.14	2.70	3.21	3.51	4.03	4.45	4.99	5.43	5.87	6.18

Table 1: Training time (seconds) comparisons on the PCA task.

Inner Loop Steps	5	25	45	65	85	105	125	145	165	185	205	225	245	250
RMM (Gao et al. 2020)	4.64×10^{3}	5.17×10^{3}	5.71×10^{3}	6.24×10^{3}	6.78×10^{3}	7.71×10^{3}	8.24×10^{3}	8.78×10^{3}	9.32×10^{3}	1.02×10^{4}	1.08×10^{4}	1.13×10^{4}	1.19×10^{4}	1.20×10^{4}
GF-RMM (Fan et al. 2021)	5.28×10^{3}	5.92×10^{3}	6.58×10^{3}	7.23×10^{3}	7.90×10^{3}	8.54×10^{3}	9.20×10^{3}	9.84×10^{3}	1.05×10^{4}	1.12×10^{4}	1.18×10^{4}	1.14×10^{4}	1.19×10^{4}	1.20×10^{4}
Ours	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}	4.55×10^{3}

Table 2: Training memory (MB) comparisons on the PCA task.

2019). These works were achieved the best performance by tuning their hyperparameters. We also compared our optimizer with the Riemannian meta-optimization method (RMM) (Gao et al. 2020) and gradient-free Riemannian meta-optimization method (GF-RMM) (Fan et al. 2021). Experiments were conducted on three tasks: principal component analysis (PCA) on the Grassmann manifold, face Recognition on the Stiefel Manifold, and clustering on the SPD manifold, detailed as follows.

PCA on the Grassmann Manifold. Principal component analysis (PCA) aims to learn an orthogonal matrix $\boldsymbol{X}$ that linearly projects original data to lower-dimensional data while preserving as much of the energy as possible. We used MNIST dataset to evaluate our method on the PCA task. We set the iteration of Neumann series as 10, and set the learning rate of meta-learning as 1×10^{-3}.

Face Recognition on the Stiefel Manifold. We modeled the face recognition using a linear classifier with the orthogonality constraint. We utilized the YaleB dataset (Lee, Ho, and Kriegman 2005) to conduct this experiment. We set the iteration of Neumann series as 5, and set the learning rate of meta-learning as 3×10^{-4}.

Clustering on the SPD Manifold. We also conducted experiments on the clustering task of SPD representations. We utilized the Kylberg texture dataset (Kylberg 2011), and represented each image by a 5×5 covariance descriptor. We set the iteration of Neumann series as 5, and set the learning rate of meta-learning as 1×10^{-4}.

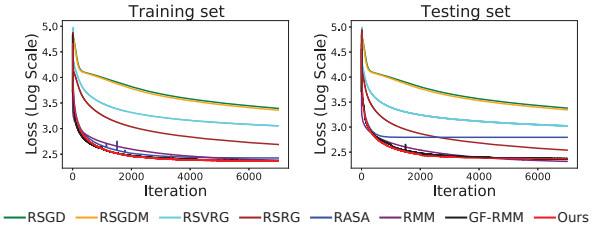

Figure 2: Plots for the PCA task (in the log scale).

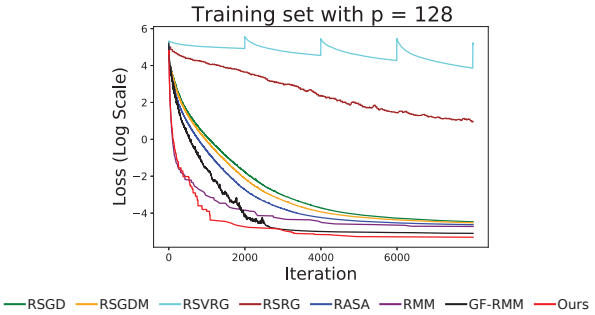

Figure 3: Plots for the face recognition task (in the log scale).

Efficiency Analysis

Training time We compared our training time of each outer-loop step with that of Riemannian meta-optimization methods RMM (Gao et al. 2020) and GF-RMM (Fan et al. 2021) for different number of optimization steps in the inner-loop. Results on the three tasks are shown in Table 1, Table 3, and Table 5. For RMM (Gao et al. 2020), its training time is highly related to the number of optimization steps in the inner-loop, and a large number of optimization steps leads to high time consumption. In contrast, the training time of our method is a small constant value, independent of the number of optimization steps in inner-loop. Thus, our method reduces much time consumption. For example, when $T = 525$, our approach requires 2.72 seconds on the face recognition task, while RMM requires 9.68×10^{2} seconds, being a factor of 350 times of our method. Although GF-RMM avoids the time-consumption retraction operation in optimization, it still requires more time than ours. Our method allows us to set a large number of optimization steps in the inner-loop, leading to an optimizer with better performance and training stability.

Training memory We evaluated the memory consumption of our method, and results are shown in Table 2, Table 4, and Table 5. Similar to the results of training time, the training memory of our method is less than the compared methods RMM and GF-RMM, independent of number of optimization steps of the inner-loop. For example, when $T = 525$, our approach only needs 2.60×10^{3} MB for each outer-loop step on the face recognition task, while the compared methods requires 9.04×10^{3} MB and 9.25×10^{3} MB, being a factor of 3.5 times of our method.

Inner Loop Steps	5	45	85	125	165	205	245	285	325	365	405	445	485	525
RMM (Gao et al. 2020)	1.80×10^{-1}	8.24	2.72×10^{1}	5.73×10^{1}	9.86×10^{1}	1.51×10^{2}	2.15×10^{2}	2.88×10^{2}	3.77×10^{2}	4.72×10^{2}	5.84×10^{2}	7.20×10^{2}	8.27×10^{2}	9.68×10^{2}
GF-RMM (Fan et al. 2021)	9.19×10^{-2}	5.71×10^{-1}	1.03	1.49	2.26	2.80	3.36	3.91	4.47	5.04	5.56	6.00	6.53	7.10
Ours	4.00×10^{-2}	2.70×10^{-1}	4.50×10^{-1}	6.60×10^{-1}	8.50×10^{-1}	1.13	1.48	1.55	1.94	1.97	2.05	2.39	2.43	2.72

Table 3: Training time (seconds) comparisons on the face recognition task.

Inner Loop Steps	5	45	85	125	165	205	245	285	325	365	405	445	485	525
RMM (Gao et al. 2020)	2.80×10^{3}	3.28×10^{3}	3.76×10^{3}	4.24×10^{3}	4.72×10^{3}	5.20×10^{3}	5.68×10^{3}	6.16×10^{3}	6.64×10^{3}	7.12×10^{3}	7.60×10^{3}	8.08×10^{3}	8.56×10^{3}	9.04×10^{3}
GF-RMM (Fan et al. 2021)	2.75×10^{3}	3.25×10^{3}	3.75×10^{3}	4.25×10^{3}	4.75×10^{3}	5.25×10^{3}	5.75×10^{3}	6.25×10^{3}	6.75×10^{3}	7.25×10^{3}	7.75×10^{3}	8.25×10^{3}	8.75×10^{3}	9.25×10^{3}
Ours	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}	2.60×10^{3}

Table 4: Training memory (MB) comparisons on the face recognition task.

Inner loop steps	Training time		Training memory	
	RMM (Gao et al. 2020)	Ours	RMM (Gao et al. 2020)	Ours
5	2.33×10^{1}	2.30×10^{1}	6.77×10^{2}	6.67×10^{2}
15	6.82×10^{1}	6.47×10^{1}	7.05×10^{2}	6.67×10^{2}
25	1.19×10^{2}	1.06×10^{2}	7.33×10^{2}	6.67×10^{2}
35	1.69×10^{2}	1.45×10^{2}	7.57×10^{2}	6.67×10^{2}
45	2.26×10^{2}	1.91×10^{2}	7.89×10^{2}	6.67×10^{2}
55	2.63×10^{2}	2.31×10^{2}	8.17×10^{2}	6.67×10^{2}
65	3.20×10^{2}	2.70×10^{2}	8.45×10^{2}	6.67×10^{2}
75	3.53×10^{2}	3.13×10^{2}	8.73×10^{2}	6.67×10^{2}
85	4.15×10^{2}	3.58×10^{2}	9.01×10^{2}	6.67×10^{2}

Table 5: Training memory (MB) and time (seconds) comparisons on the clustering task.

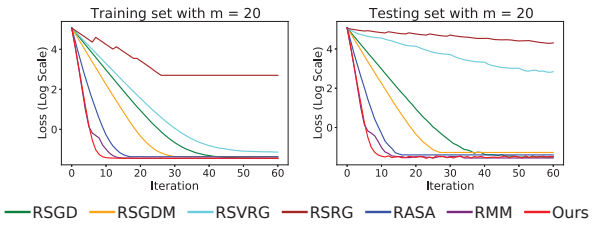

Figure 4: Plots for the clustering task (in the log scale).

Convergence Analysis

We evaluated the convergence performance of our optimizer on the three tasks. On the PCA and clustering tasks, we trained our optimizer on the training set and evaluated it on both the training and test sets. On the face recognition task, we trained and evaluated our optimizer on the training set. Experimental results on the three tasks are shown in Figure 2, Figure 3, and Figure 4. Our learned optimizer achieves better performance than hand-designed optimizers, in respect of the convergence speed and the final loss value on both seen training data and unseen test data. This shows the effectiveness of our learned optimizer that capture underlying data structures to obtain a better optimization trajectory. Compared with the RMM and GF-RMM, our method performs competitively and even surpasses them. For example, in the face recognition task, our method achieves the lowest loss value -5.3 (log scale), while the final loss value of (Gao et al. 2020) is -4.7, higher than ours.

Method	Face Recognition	Clustering
RSGD (Bonnabel 2013)	79.4	85.1
RSGDM (Kumar, Mhammedi, and Harandi 2018)	79.6	85.0
RSVRG (Zhang, Reddi, and Sra 2016)	79.6	80.3
RSRG (Kasai, Sato, and Mishra 2018)	78.0	83.9
RASA (Kasai, Jawanpuria, and Mishra 2019)	80.2	85.2
RMM (Gao et al. 2020)	89.0	85.2
GF-RMM (Fan et al. 2021)	90.2	-
Ours	**95.1**	**86.7**

Table 6: Accuracy (%) of solved classifiers and centers.

Accuracy Analysis

We evaluated the accuracy performance of the solved linear classifier in the face recognition task and the solved centers in the clustering task. Specifically, in the face recognition task, we solved the classifier on the training set using our learned optimizer, and measured the accuracy of the test set. In the clustering task, we regarded the solved centers as category prototypes and then computed distance between test data and prototypes for classification. Results are shown in Table 6. The performance of the solved classifier and prototypes by our optimizer surpasses all compared Riemannian optimizers. The best accuracy of existing methods on the two tasks is 90.2% on the face recognition task and 85.2% on the clustering task, achieved by the RMM and GF-RMM. Compared with them, our method achieves 6.1% and 1.5% improvements. This shows that our method arrives at a better optima and has a good generalization ability.

Conclusion

In this paper, we have presented an efficient Riemannian meta-optimization method by deriving the Riemannian implicit differentiation. The proposed method provides an analytic expression for meta-gradient that only depends on the final two optimization steps in the inner-loop. Our method avoids saving and differentiating through the whole inner-loop procedure, which reduces computation and memory cost significantly. We demonstrate theoretically and empirically that our method only needs a small constant memory and computational cost. Noticeably, compared with existing Riemannian meta-optimization methods on the face recognition task, our method achieves better performance using less than 0.0025 time and 0.28 time GPU memory consumption. Furthermore, experiments on three tasks demonstrate that our method can learn a good optimization trajectory.

Acknowledgments

This work was supported by the Natural Science Foundation of China (NSFC) under Grants No. 62176021 and No. 62172041.

References

Absil, P.-A.; Mahony, R.; and Sepulchre, R. 2009. *Optimization algorithms on matrix manifolds*. Princeton University Press.

Andrychowicz, M.; Denil, M.; Gomez, S.; Hoffman, M. W.; Pfau, D.; Schaul, T.; Shillingford, B.; and De Freitas, N. 2016. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems (NeurIPS)*, 3981–3989.

Baydin, A. G.; Pearlmutter, B. A.; Radul, A. A.; and Siskind, J. M. 2018. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18.

Bengio, Y. 2000. Gradient-based optimization of hyperparameters. *Neural computation*, 12(8): 1889–1900.

Bertrand, Q.; Klopfenstein, Q.; Blondel, M.; Vaiter, S.; Gramfort, A.; and Salmon, J. 2020. Implicit differentiation of Lasso-type models for hyperparameter optimization. In *International Conference on Machine Learning*, 810–821. PMLR.

Bonnabel, S. 2013. Stochastic Gradient Descent on Riemannian Manifolds. *IEEE Transactions on Automatic Control*, 58(9): 2217–2229.

Chen, T.; Zhang, W.; Jingyang, Z.; Chang, S.; Liu, S.; Amini, L.; and Wang, Z. 2020. Training stronger baselines for learning to optimize. *Advances in Neural Information Processing Systems*, 33.

Christianson, B. 1992. Automatic Hessians by reverse accumulation. *IMA Journal of Numerical Analysis*, 12(2): 135–150.

Dai, W.; Milenkovic, O.; and Kerman, E. 2011. Subspace evolution and transfer (SET) for low-rank matrix completion. *IEEE Transactions on Signal Processing*, 59(7): 3120–3132.

Fan, X.; Gao, Z.; Wu, Y.; Jia, Y.; and Harandi, M. 2021. Learning a Gradient-free Riemannian Optimizer on Tangent Spaces. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(8): 7377–7384.

Gao, Z.; Wu, Y.; Jia, Y.; and Harandi, M. 2020. Learning to Optimize on SPD Manifolds. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 7700–7709.

Griewank, A. 1993. Some bounds on the complexity of gradients, Jacobians, and Hessians. In *Complexity in numerical optimization*, 128–162. World Scientific.

Griewank, A.; and Walther, A. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

Gudovskiy, D.; Rigazio, L.; Ishizaka, S.; Kozuka, K.; and Tsukizawa, S. 2021. AutoDO: Robust AutoAugment for Biased Data with Label Noise via Scalable Probabilistic Implicit Differentiation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 16601–16610.

Harandi, M.; Salzmann, M.; and Hartley, R. 2017. Joint dimensionality reduction and metric learning: A geometric take. In *International Conference on Machine Learning*, 1404–1413. PMLR.

Kasai, H.; Jawanpuria, P.; and Mishra, B. 2019. Riemannian adaptive stochastic gradient algorithms on matrix manifolds. In *International Conference on Machine Learning (ICML)*, 3262–3271.

Kasai, H.; Sato, H.; and Mishra, B. 2018. Riemannian Stochastic Recursive Gradient Algorithm. In *International Conference on Machine Learning (ICML)*, 2516–2524.

Kumar, S., Roy; Mhammedi, Z.; and Harandi, M. 2018. Geometry aware constrained optimization techniques for deep learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 4460–4469.

Kylberg, G. 2011. *Kylberg Texture Dataset v. 1.0*.

Larsen, J.; Hansen, L. K.; Svarer, C.; and Ohlsson, M. 1996. Design and regularization of neural networks: the optimal use of a validation set. In *Neural Networks for Signal Processing VI. Proceedings of the 1996 IEEE Signal Processing Society Workshop*, 62–71. IEEE.

Lee, K.-C.; Ho, J.; and Kriegman, D. J. 2005. Acquiring linear subspaces for face recognition under variable lighting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(5): 684–698.

Liao, R.; Xiong, Y.; Fetaya, E.; Zhang, L.; Yoon, K.; Pitkow, X.; Urtasun, R.; and Zemel, R. 2018. Reviving and improving recurrent back-propagation. In *International Conference on Machine Learning*, 3082–3091. PMLR.

Liu, Y.; Shang, F.; Cheng, J.; Cheng, H.; and Jiao, L. 2017. Accelerated First-order Methods for Geodesically Convex Optimization on Riemannian Manifolds. In *Advances in Neural Information Processing Systems (NeurIPS)*, 4868–4877.

Lorraine, J.; Vicol, P.; and Duvenaud, D. 2020. Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics*, 1540–1552. PMLR.

Luenberger, D. G. 1972. The gradient projection method along geodesics. *Management Science*, 18(11): 620–631.

Metz, L.; Maheswaranathan, N.; Nixon, J.; Freeman, D.; and Sohl-Dickstein, J. 2019. Understanding and correcting pathologies in the training of learned optimizers. In *the International Conference on Machine Learning (ICML)*, 4556–4565.

Navon, A.; Achituve, I.; Maron, H.; Chechik, G.; and Fetaya, E. 2020. Auxiliary Learning by Implicit Differentiation. In *International Conference on Learning Representations*.

Rajeswaran, A.; Finn, C.; Kakade, S.; and Levine, S. 2019. Meta-learning with implicit gradients.

Shaban, A.; Cheng, C.-A.; Hatch, N.; and Boots, B. 2019. Truncated back-propagation for bilevel optimization. In *The 22nd International Conference on Artificial Intelligence and Statistics*, 1723–1732. PMLR.

Wu, Y.; Ren, M.; Liao, R.; and Grosse, R. 2018. Understanding short-horizon bias in stochastic meta-optimization. *arXiv preprint arXiv:1803.02021*.

Zhang, H.; Reddi, S. J.; and Sra, S. 2016. Riemannian SVRG: Fast stochastic optimization on Riemannian manifolds. In *Advances in Neural Information Processing Systems (NeurIPS)*, 4592–4600.

Zhang, H.; and Sra, S. 2018. Towards Riemannian accelerated gradient methods. *arXiv*, preprint:1806.02812.