SIAM J. SCI. COMPUT.
Vol. 44, No. 2, pp. A843–A869

AUTOMATIC DIFFERENTIATION FOR RIEMANNIAN OPTIMIZATION ON LOW-RANK MATRIX AND TENSOR-TRAIN MANIFOLDS[*]

ALEXANDER NOVIKOV[†], MAXIM RAKHUBA[‡], AND IVAN OSELEDETS[§]

Abstract. In scientific computing and machine learning applications, matrices and more general multidimensional arrays (tensors) can often be approximated with the help of low-rank decompositions. Since matrices and tensors of fixed rank form smooth Riemannian manifolds, one of the popular tools for finding low-rank approximations is to use Riemannian optimization. Nevertheless, efficient implementation of Riemannian gradients and Hessians, required in Riemannian optimization algorithms, can be a nontrivial task in practice. Moreover, in some cases, analytic formulas are not even available. In this paper, we build upon automatic differentiation and propose a method that, given an implementation of the function to be minimized, efficiently computes Riemannian gradients and matrix-by-vector products between an approximate Riemannian Hessian and a given vector.

Key words. automatic differentiation, Riemannian optimization, low-rank approximation, tensor-train decomposition

AMS subject classifications. 65K05, 15A69, 53B21

DOI. 10.1137/20M1356774

1. Introduction.

Automatic differentiation (AD) is a powerful tool for numerically calculating derivatives of functions specified as computer programs. It significantly simplifies the programming of derivatives of complicated functions without loss of efficiency, providing better stability properties compared with classical numerical differentiation using finite differences. AD is commonly used in applied mathematics, and in particular, it is at the core of deep learning success, allowing researchers to combine ever more complex neural networks from known modules and train them without worrying about efficient gradient computation.

In this paper, we are concerned with applying AD to the minimization problem

$$\min_{\mathbf{X}\in\mathcal{M}} f(\mathbf{X}),$$

where $f\colon \mathbb{R}^{n_1\times\cdots\times n_d} \to \mathbb{R}$ is a smooth function and $\mathcal{M}$ is a subset of $\mathbb{R}^{n_1\times\cdots\times n_d}$ of fixed-rank matrices ($d = 2$) or fixed-rank tensor-trains (TT) ($d > 2$) [1]. It is known that in both cases, $\mathcal{M}$ forms a Riemannian manifold. One can, therefore, apply Riemannian optimization algorithms [2] that are currently actively used for the development of state-of-the-art algorithms in numerical mathematics, partial differential equations,

[*] Submitted to the journal's Methods and Algorithms for Scientific Computing section August 3, 2020; accepted for publication (in revised form) October 13, 2021; published electronically April 21, 2022.
https://doi.org/10.1137/20M1356774
Funding: This work was supported by the Ministry of Science and Higher Education of the Russian Federation (grant 075-15-2020-801).

[†] Marchuk Institute of Numerical Mathematics of the Russian Academy of Sciences, 119333 Moscow, Russia, and HSE University, Pokrovsky Boulevard 11, Moscow, 109028 Russian Federation (Anovikov@google.com).

[‡] HSE University, Pokrovsky Boulevard 11, Moscow, 109028 Russian Federation (mrakhuba@hse.ru).

[§] Skolkovo Institute of Science and Technology, Skolkovo Innovation Center, 121205 Moscow, Russia, and Marchuk Institute of Numerical Mathematics of the Russian Academy of Sciences, 119333 Moscow, Russia (i.oseledets@skoltech.ru).

and machine learning. A realization of such algorithms requires specific knowledge of computational aspects of low-rank objects and is especially complicated for tensor decompositions, where a number of tricks have to be done to reduce rank dependence and ensure the stability of an algorithm. The AD technique proposed in this work allows for a significant simplification of this process.

We are concerned with computing Riemannian gradients and matrix-vector products with approximate Riemannian Hessians, which are the building blocks for Riemannian optimization algorithms. Note that in the case of low-rank matrix or tensor-train manifolds, the matrix-vector product with the Hessian can be numerically unstable. It happens due to the presence of terms with inverted singular values [3]. We, therefore, consider multiplication by the *approximate* Hessian with an omitted curvature term [4, 5, 6] (see the details in section 3).

In the proposed method, calculating the Riemannian gradient or matrix-vector product with the approximate Riemannian Hessian of a function has the same asymptotic complexity as evaluation of the function itself at a single point.[1] Moreover, thanks to the implementation in TensorFlow (a Python library with AD support), the algorithms can be run both on CPUs and GPUs.

We numerically evaluate the performance of the proposed algorithms on several functions arising from solving systems of linear equations, the eigenvalue problem, the tensor completion problem and in the training of a machine learning model.

Our main contributions are as follows:

- We develop AD algorithms for computing the Riemannian gradient and a matrix-vector product with the approximate Riemannian Hessian of a function for low-rank matrices and TT-tensors. Under mild assumptions, the asymptotic complexity of the proposed method equals the complexity of evaluating the function at one point.
- We implement the proposed algorithms in TensorFlow and make them available in `T3F`[2]—an open-source Python library for working with TT-decomposition.

Related work. There is a large body of work on creating libraries for working with tensors and tensor decompositions, which often include AD abilities (see, e.g., [7, 8, 9, 10, 11], `tntorch`[3]), but most of these libraries do not target the *Riemannian AD*, which is the focus of this paper. Typically, researchers compute the Riemannian gradients manually, but the Riemannian AD libraries [12, 13, 14] are gaining traction, empowering the Riemannian optimization community. However, existing Riemannian AD libraries lack low-rank tensor support. For low-rank matrices, `PyManOpt` [12] supports Riemannian gradients, but no library supports multiplying the Riemannian Hessian by a given vector, which is required for second-order methods.

Note that in [12], an algorithm to compute the Riemannian gradient for low-rank matrices has already been proposed and implemented. Nevertheless, in this work, we present an alternative way of doing it, avoiding inversions of singular values, which can be close to machine epsilon if the rank is overestimated.

A method for automatic second-order Riemannian differentiation for the manifold of low-rank tensors was proposed in [15]. The authors focus on the curvature term of

[1] This holds under the assumption that the function evaluation is at least as expensive as the cost of the orthogonalization operation, which is a necessary step in any Riemannian gradient computation. This assumption holds true for most practical functions (see Propositions 5.2 and 6.2 for more details).

[2] https://github.com/Bihaqo/t3f.

[3] https://tntorch.readthedocs.io.

the Riemannian Hessian (which we omit as explained in section 3) and assume that the other terms can be computed efficiently by a two-step procedure: first computing the Euclidean gradient or Hessian-by-vector product and then projecting it onto the tangent space. This is indeed efficient for some functions, but can be significantly slower than the approach proposed in this paper for some other functions. Thus, the two papers complement each other: one can use [15] for computing the curvature term and the algorithms proposed in this paper for the other terms.

2. Automatic differentiation.

In this section, we give a brief introduction to the AD concept. A reader familiar with this topic can skip this section.

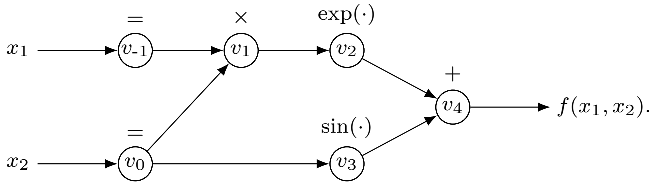
AD is a technique for computing the value of the gradient of a smooth function $f\colon \mathbb{R}^N \to \mathbb{R}$ specified by a computer program. In particular, it is assumed that f can be represented as a sequence of elementary operations, for example, additions, multiplications, trigonometric functions, logarithms, etc. Evaluation of f can also involve other operations such as matrix decompositions, for which differentiation formulas are available. Under this assumption, AD allows for computing derivatives with working precision and with the number of operations, which is only a small constant factor times larger than the number of operations to execute the evaluation of f (i.e., with the same asymptotic complexity).

Let us illustrate the AD concept in a simple example. Letting $f\colon \mathbb{R}^2 \to \mathbb{R}$,

$$f(x_1, x_2) = e^{x_1 x_2} + \sin x_2,$$

and then it can be written as a sequence of elementary operations and depicted as the following computational graph:

$$\begin{aligned}
v_{-1} &= x_1\\
v_0 &= x_2\\
v_1 &= v_{-1}\, v_0\\
v_2 &= e^{v_1}\\
v_3 &= \sin v_0\\
v_4 &= v_2 + v_3\\
f(x_1, x_2) &= v_4
\end{aligned}$$



AD uses the chain rule[4] to find both components of ∇f in one pass through the computational graph in reverse order. Let $\overline{v}_i \triangleq \frac{\partial f}{\partial v_i}$ for $i = -1, \ldots, 4$. We have

$$\begin{aligned}
\overline{v}_4 &= \frac{\partial f}{\partial v_4} = 1,\\
\overline{v}_3 &= \frac{\partial f}{\partial v_4}\,\frac{\partial v_4}{\partial v_3} \equiv \overline{v}_4,\\
\overline{v}_2 &= \frac{\partial f}{\partial v_4}\,\frac{\partial v_4}{\partial v_2} \equiv \overline{v}_4,\\
\overline{v}_1 &= \frac{\partial f}{\partial v_2}\,\frac{\partial v_2}{\partial v_1} \equiv \overline{v}_2 e^{v_1},
\end{aligned}$$

[4] In this paper we focus on *reverse-mode* AD, which is also sometimes called backpropagation. The alternative—*forward-mode* AD—is typically used for functions $f : \mathbb{R}^M \to \mathbb{R}^N$, where $M < N$ because of the smaller asymptotic complexity in this case.

$$\overline{v}_0 = \frac{\partial f}{\partial v_3}\,\frac{\partial v_3}{\partial v_0} + \frac{\partial f}{\partial v_1}\,\frac{\partial v_1}{\partial v_0} \equiv \overline{v}_3 \cos v_0 + \overline{v}_1 v_{-1},$$
$$\overline{v}_{-1} = \frac{\partial f}{\partial v_1}\,\frac{\partial v_1}{\partial v_{-1}} \equiv \overline{v}_1 v_0,$$

where $\overline{v}_{-1} = \frac{\partial f}{\partial v_{-1}} \equiv \frac{\partial f}{\partial x_1}$ and $\overline{v}_0 = \frac{\partial f}{\partial v_0} \equiv \frac{\partial f}{\partial x_2}$.

Thus, AD allows us to calculate all components of ∇f in one pass with $\mathcal{O}(F)$ complexity, where F is the number of FLOP to calculate f at a given $(x_1, \ldots, x_N)$. In general, the computational graph for computing the gradient of a function has as many nodes as the original graph for evaluating the function value, and each node is, at most, a small constant times more expensive than the corresponding node from the original graph.

Let us compare AD with numerical differentiation using finite differences, where components of a gradient of a function $f : \mathbb{R}^N \to \mathbb{R}$ are approximated, e.g., using forward differences

$$\frac{\partial f}{\partial x_i}(x_1, \ldots, x_N) \approx \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_N) - f(x_1, \ldots, x_N)}{h}, \tag{2.1}$$

where h is chosen so that the approximation error is small enough. First, numerical differentiation is computationally more expensive than AD. Indeed, (2.1) requires $N+1$ function evaluations to approximate ∇f and, hence, the complexity is $\mathcal{O}(NF)$. Moreover, due to the error amplification of derivative approximation, (2.1) cannot achieve accuracy better than the square root of machine precision [16]. At the same time, AD is more robust and can achieve machine precision accuracy [17].

Another alternative to AD and numerical differentiation is symbolic differentiation. In it, one assembles the final formula for each component of the gradient using a sequence of rules as product rule, chain rule, etc. Since this constraint of expressing the entire result as a single formula does not allow introducing intermediate variables, in the worst case the final formula may contain exponentially many duplicated fragments. By contrast to the symbolic differentiation, in AD one uses intermediate variables to define those duplicated fragments, allowing one to never evaluate any quantity more than once and providing efficiency guarantees.

For a more in-depth review of AD see, e.g., [18].

3. Riemannian optimization.

Let us briefly introduce the Riemannian optimization concept. Let $\mathcal{M} \subset \mathbb{R}^{n_1 \times \cdots \times n_d}$ be a smooth embedded submanifold. In this paper, we are concerned with the manifold of fixed-rank matrices ($d = 2$) and the manifold of tensors of fixed TT-rank ($d > 2$). The definitions will be given in sections 4.1 and 5.1, respectively. In this section, we only provide an introductory overview without implementation details.

Our goal is to solve a minimization problem with a smooth function $f\colon \mathbb{R}^{n_1 \times \cdots \times n_d} \to \mathbb{R}$:

$$\min_{\mathbf{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}} f(\mathbf{X}).$$

Assume that the solution to this problem can be approximated by a certain point $\mathbf{X}_* \in \mathcal{M}$. Then, we can reformulate the problem as

$$\min_{\mathbf{X} \in \mathcal{M}} f(\mathbf{X}), \tag{3.1}$$

i.e., the search space $\mathbb{R}^{n_1 \times \cdots \times n_d}$ is restricted to a Riemannian manifold $\mathcal{M}$. Riemannian optimization algorithms usually involve computation of Riemannian gradients

$\operatorname{grad} f(\mathbf{X})$, which for embedded submanifolds of $\mathbb{R}^{n_1\times\cdots\times n_d}$ and functions f defined on the ambient space may be written as a projection of the Euclidean gradient $\nabla f(\mathbf{X})$ to the tangent plane $T_{\mathbf{X}}\mathcal{M}$ of $\mathcal{M}$ at the point $\mathbf{X}$:

$$\operatorname{grad} f(\mathbf{X}) = \mathrm{P}_{T_{\mathbf{X}}\mathcal{M}} \nabla f(\mathbf{X}), \tag{3.2}$$

where $\mathrm{P}_{T_{\mathbf{X}}\mathcal{M}} \colon \mathbb{R}^{n_1\times\cdots\times n_d} \to T_{\mathbf{X}}\mathcal{M}$ denotes an operator of orthogonal projection to the tangent plane $T_{\mathbf{X}}\mathcal{M}$ and depends on $\mathbf{X}$ nonlinearly. Given the Riemannian gradient notion, we may solve (3.1) using the Riemannian gradient descent

$$\mathbf{X}_{k+1} = R_{\mathbf{X}_k}(\tau_k \operatorname{grad} f(\mathbf{X}_k))),$$

where $R_{\mathbf{X}_k} \colon T_{\mathbf{X}}\mathcal{M} \to \mathbb{R}^{n_1\times\cdots\times n_d}$ returns a tangent vector back to the manifold (see [19] for different retraction operations) and the parameter τ_k is chosen to ensure decay of the functional. More advanced optimization algorithms, e.g., a Riemannian version of the conjugate gradient method, are also available [2].

One can also utilize second-order methods, which involve computation of the Riemannian Hessian operator. For the Riemannian Hessian $\operatorname{Hess} f(\mathbf{X}) \colon T_{\mathbf{X}}\mathcal{M} \to T_{\mathbf{X}}\mathcal{M}$ we can use[5] the formula [3, 4]

$$\operatorname{Hess} f(\mathbf{X}) = \mathrm{P}_{T_{\mathbf{X}}\mathcal{M}} \nabla^2 f(\mathbf{X}) + \mathrm{P}_{T_{\mathbf{X}}\mathcal{M}} \dot{\mathrm{P}}_{T_{\mathbf{X}}\mathcal{M}}(\nabla f(\mathbf{X})), \tag{3.3}$$

where $\nabla^2 f(\mathbf{X})$ is the Euclidean Hessian and $\dot{\mathrm{P}}_{T_{\mathbf{X}}\mathcal{M}}$ denotes the Fréchet derivative of $\mathrm{P}_{T_{\mathbf{X}}\mathcal{M}}$. The second term in (3.3) arises due to the nonlinearity of the manifold. For the manifold of low-rank matrices, it contains the inverse of a matrix of singular values [3]. If singular values are small, this can lead to numerical instabilities. To avoid this problem, the second term in (3.3) can be omitted [4, 20]. In this case, the optimization procedure can be interpreted as a constrained Gauss–Newton method. We, therefore, consider only linearized Hessians and are interested in an efficient matrix-vector product by the first term of (3.3):

$$\mathrm{H}_{\mathbf{X}}[\mathbf{Z}] \equiv \mathrm{P}_{T_{\mathbf{X}}\mathcal{M}} \nabla^2 f(\mathbf{X})\, \mathbf{Z}, \qquad \mathbf{X} \in \mathcal{M}, \quad \mathbf{Z} \in T_{\mathbf{X}}\mathcal{M}. \tag{3.4}$$

Note that first computing $\nabla f(\mathbf{X})$ as in (3.2) and then applying $\mathrm{P}_{T_{\mathbf{X}}\mathcal{M}}$ can be inefficient. Therefore, $\mathrm{P}_{T_{\mathbf{X}}\mathcal{M}} \nabla f(\mathbf{X})$ should be calculated at once. For example, for the manifold of low-rank matrices, the Riemannian gradient $\mathrm{P}_{T_{\mathbf{X}}\mathcal{M}} \nabla f(\mathbf{X})$ can always be represented as a low-rank matrix (see section 4.1 for details), while the Euclidean gradient $\nabla f(\mathbf{X})$ can have an arbitrary large rank. Thus, using the Euclidean gradient in the intermediate calculations can lead to an inefficient algorithm. Similarly, first computing $\nabla^2 f(\mathbf{X})\, \mathbf{Z}$ as in (3.4) and then applying $\mathrm{P}_{T_{\mathbf{X}}\mathcal{M}}$ can be significantly less efficient than calculating $\mathrm{P}_{T_{\mathbf{X}}\mathcal{M}} \nabla^2 f(\mathbf{X})\, \mathbf{Z}$ at once. The goal of this paper is, thus, to develop an efficient tool to calculate (3.2) and (3.4)—the building block operations of Riemannian optimization. The key assumption we make is that we can efficiently evaluate f at any point $\mathbf{X} + T_{\mathbf{X}}\mathcal{M}$, $\mathbf{X} \in \mathcal{M}$. Then, under mild conditions (see Propositions 5.2 and 6.2), the overall complexity of the presented algorithm is only constant times larger than the complexity of the function evaluation.

Let us introduce the scalar product and the associated norm

$$\langle \mathbf{X}, \mathbf{Y} \rangle = \sum_{i_1,\dots,i_d=1}^{n_1,\dots,n_d} \mathbf{X}_{i_1,\dots,i_d} \mathbf{Y}_{i_1,\dots,i_d}, \qquad \|\mathbf{X}\| = \langle \mathbf{X}, \mathbf{X} \rangle^{1/2}.$$

[5] Note that both grad and Hess operations depend on the particular choice of a manifold. Nevertheless, we do not use the subscript $\mathcal{M}$ as it will be clear from context and to not overcomplicate the notation.

Using this notation, possible choices of $f(\mathbf{X})$ are, for example, the following:

- $f(\mathbf{X}) = \|\mathrm{A}\mathbf{X} - \mathbf{F}\|^2$ or $f(\mathbf{X}) = <\mathrm{A}\mathbf{X}, \mathbf{X}> - 2 < \mathbf{F}, \mathbf{X} >$ for given $\mathrm{A}\colon \mathbb{R}^{n_1 \times \cdots \times n_d} \to \mathbb{R}^{n_1 \times \cdots \times n_d}$ and $\mathbf{F} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ that arise when solving linear systems.
- $f(\mathbf{X}) = < \mathrm{A}[\mathbf{X}], \mathbf{X} > / < \mathbf{X}, \mathbf{X} >$ with possibly nonlinear $\mathrm{A}\colon \mathbb{R}^{n_1 \times \cdots \times n_d} \to \mathbb{R}^{n_1 \times \cdots \times n_d}$, which arises when solving (nonlinear) eigenvalue problems.
- $f(\mathbf{X}) = \|\mathrm{P}_\Omega(\mathbf{X} - \mathbf{A})\|^2$, where P_Ω denotes projection on the index set Ω such that
$$\mathrm{P}_\Omega \mathbf{X} = \begin{cases} X_{i_1 \ldots i_d}, & (i_1, \ldots, i_d) \in \Omega, \\ 0 & \text{otherwise.} \end{cases}$$
This type of problem is referred to as a matrix or tensor completion problem.
- $f(\mathbf{X})$ is a neural network loss function, which arises when using TT-decomposition to parametrize a recurrent neural network and applying Riemannian optimization for training (for more details see section 7.1).

In section 5.3, we will also discuss how our approach can be used for operations that are not directly related to a minimization of a function, e.g., how to efficiently compute the preconditioned residual $\mathrm{P}_{T_{\mathbf{X}}\mathcal{M}} \mathrm{B}^{-1}(\mathrm{A}\mathbf{X} - \mathbf{F})$ for noncommuting A and B.

4. Automatic differentiation for the Riemannian gradient: Fixed-rank matrices.

In this section, we propose an approach to automatically compute Riemannian gradients for the manifold of fixed-rank matrices.

4.1. The manifold of fixed-rank matrices.

Let us briefly recall the concepts related to the manifold of fixed-rank matrices. The set of matrices of fixed rank r: $\mathcal{M}_r = \{\mathbf{X} \in \mathbb{R}^{m \times n} : \operatorname{rank}(\mathbf{X}) = r\}$ forms a smooth submanifold of $\mathbb{R}^{m \times n}$ [21, Example 8.14]. Using SVD, any point $\mathbf{X} \in \mathcal{M}_r$ of the manifold can be represented as $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\intercal$, where $\mathbf{U} \in \mathbb{R}^{m \times r}$ and $\mathbf{V} \in \mathbb{R}^{n \times r}$ are matrices with orthonormal columns — singular vectors ($\mathbf{U}^\intercal\mathbf{U} = \mathbf{I}_r$, $\mathbf{V}^\intercal\mathbf{V} = \mathbf{I}_r$) and $\mathbf{S} \in \mathbb{R}^{r \times r}$ is the diagonal matrix of singular values. The tangent space $T_{\mathbf{X}}\mathcal{M}_r$ of the manifold $\mathcal{M}_r$ at a point $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\intercal \in \mathcal{M}_r$ can be written as

$$T_{\mathbf{X}}\mathcal{M}_r = \{\dot{\mathbf{U}}\mathbf{V}^\intercal + \mathbf{U}\dot{\mathbf{V}}^\intercal \mid \dot{\mathbf{U}} \in \mathbb{R}^{m \times r}, \dot{\mathbf{V}} \in \mathbb{R}^{n \times r} : \mathbf{V}^\intercal\dot{\mathbf{V}} = \mathbf{O}_{r \times r}\}, \tag{4.1}$$

where $\mathbf{O}_{r \times r}$ denotes a zero matrix of size $r \times r$. In what follows, we refer to the matrices $\dot{\mathbf{U}}$ and $\dot{\mathbf{U}}$ that define an element of the tangent space as *delta-matrices*. The orthogonal projection of $\mathbf{Z} \in \mathbb{R}^{m \times n}$ to the tangent space $T_{\mathbf{X}}\mathcal{M}_r$ can, thus, be obtained as follows:

$$\mathrm{P}_{T_{\mathbf{X}}\mathcal{M}_r}\mathbf{Z} = \mathbf{Z}\mathbf{V}\mathbf{V}^\intercal + \mathbf{U}\mathbf{U}^\intercal\mathbf{Z}(\mathbf{I} - \mathbf{V}\mathbf{V}^\intercal). \tag{4.2}$$

We refer the reader to, e.g., [22, section 2.1] for a more detailed discussion of the manifold of low-rank matrices, including the derivation of (4.2).

Finally, to simplify the notation, we denote the projection operator as

$$\mathrm{P}_{\mathbf{X}} \triangleq \mathrm{P}_{T_{\mathbf{X}}\mathcal{M}_r}.$$

We also introduce $\mathcal{T}_{\mathbf{X}}$ that maps parametrization matrices to an element of the tangent plane at the point $\mathbf{X}$

$$\mathcal{T}_{\mathbf{X}} : \mathbb{R}^{m \times r} \times \mathbb{R}^{n \times r} \to T_{\mathbf{X}}\mathcal{M}_r, \tag{4.3}$$

namely,

$$\mathbf{T} = \mathcal{T}_{\mathbf{X}}(\dot{\mathbf{U}}, \dot{\mathbf{V}}) = \dot{\mathbf{U}}\mathbf{V}^\intercal + \mathbf{U}\dot{\mathbf{V}}^\intercal. \tag{4.4}$$

This mapping will be used later in section 4.2 to simplify the presentation of the algorithm.

4.2. Automatic differentiation approach. In this section, we propose an efficient way of computing the Riemannian gradient

$$\operatorname{grad} f(\mathbf{X}) = \mathrm{P}_{\mathbf{X}} \nabla f \in \mathcal{T}_{\mathbf{X}}\mathcal{M}_r \subset \mathbb{R}^{m\times n}.$$

The Riemannian gradient $\operatorname{grad} f(\mathbf{X})$ is an $m \times n$ matrix, but as noted in the previous section it can be defined via the delta matrices $\dot{\mathbf{U}}$ and $\dot{\mathbf{V}}$ using just $(m+n)r$ parameters ($(m+n)r - r^2$ if gauge conditions $\mathbf{V}^\intercal \dot{\mathbf{V}} = \mathbf{O}_{r\times r}$ are taken into account). Thus, if we can avoid using full $m \times n$ matrices in intermediate calculations, we can potentially compute the Riemannian gradient with a better asymptotic complexity than $\mathcal{O}(mn)$.

A naive approach of computing the Riemannian gradient is to first compute $\partial f/\partial \mathbf{X}$ with AD and then project the result to the tangent plane by using formula (4.2):

$$\mathrm{P}_{\mathbf{X}}\nabla f = \nabla f \mathbf{V}\mathbf{V}^\intercal + \mathbf{U}\mathbf{U}^\intercal \nabla f\,(\mathbf{I} - \mathbf{V}\mathbf{V}^\intercal). \tag{4.5}$$

The problem with this approach is that it requires finding the full matrix of the Euclidean gradient $\partial f/\partial \mathbf{X}$ of the size $m \times n$, which we want to avoid. Alternatively, we may find the Riemannian gradient without explicitly forming $\partial f/\partial \mathbf{X}$. In particular, we notice that the Riemannian gradient (4.5) involves computing the following multiplication of matrices:

$$(\nabla f \mathbf{V}) \in \mathbb{R}^{m\times r}, \quad (\mathbf{U}^\intercal \nabla f) \in \mathbb{R}^{r\times n}. \tag{4.6}$$

We may find these two quantities by using the classical AD as follows:

$$\nabla f \mathbf{V} = \nabla_{\mathbf{E}} f(\mathbf{E}\mathbf{V}^\intercal)|_{\mathbf{E}=\mathbf{U}\mathbf{S}}, \quad \mathbf{U}^\intercal \nabla f = \nabla_{\mathbf{F}} f(\mathbf{U}\mathbf{F})|_{\mathbf{F}=\mathbf{S}\mathbf{V}^\intercal}.$$

So, one can use classic AD on the function f *twice* (each time with the complexity equal to evaluating the function f at a single point due to AD properties) to compute all the pieces that depend on f.

However, in the rest of this section we propose an alternative way of computing quantities (4.6) by using classic AD a *single time* on a specially introduced auxiliary function. This alternative approach is introduced because it naturally generalizes into an efficient algorithm for the tensor case (see section 5.2).

Quantities (4.6) can be computed at once by differentiating (using AD) the following auxiliary function defined using mapping (4.4):

$$g \overset{\text{def}}{=} f \circ \mathcal{T}_{\mathbf{X}}.$$

We have

$$g(\mathbf{A}, \mathbf{B}) = f(\mathcal{T}_{\mathbf{X}}(\mathbf{A}, \mathbf{B})) = f(\mathbf{A}\mathbf{V}^\intercal + \mathbf{U}\mathbf{B}^\intercal). \tag{4.7}$$

Indeed, $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\intercal$ can be represented as

$$\mathcal{T}_{\mathbf{X}}(\mathbf{U}\mathbf{S}, \mathbf{O}_{n\times r}) = (\mathbf{U}\mathbf{S}) \cdot \mathbf{V}^\intercal + \mathbf{U} \cdot \mathbf{O}_{n\times r}^\intercal = \mathbf{X},$$

and, hence, the partial derivatives of $\mathbf{T} = \mathcal{T}_{\mathbf{X}}(\dot{\mathbf{U}}, \dot{\mathbf{V}})$ at $(\mathbf{A}, \mathbf{B}) = (\mathbf{U}\mathbf{S}, \mathbf{O}_{n\times r})$ are

$$\frac{\partial T_{ij}}{\partial A_{pq}} = \frac{\partial(\mathbf{A}\mathbf{V}^\intercal + \mathbf{U}\mathbf{B}^\intercal)_{ij}}{\partial A_{pq}} = \delta_{ip} V_{jq},$$

Algorithm 4.1 Computing the Riemannian gradient for low-rank matrices via AD.

```
Require: $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\intercal \in \mathbb{R}^{m\times n}$, $p(\mathbf{L}, \mathbf{R})$—implementation of evaluating $f$ at $\mathbf{L}\mathbf{R}^\intercal$ for
    any $\mathbf{L} \in \mathbb{R}^{m\times 2r}$ and $\mathbf{R} \in \mathbb{R}^{n\times 2r}$.
Ensure: $\dot{\mathbf{U}}, \dot{\mathbf{V}}$ such that $\mathrm{P}_{\mathbf{X}}\nabla f = \dot{\mathbf{U}}\mathbf{V}^\intercal + \mathbf{U}\dot{\mathbf{V}}^\intercal$

function g($\mathbf{A}, \mathbf{B}$)
    return $p([\mathbf{U}\,\mathbf{A}],\ [\mathbf{B}\,\mathbf{V}])$
Using AD, compute $\dot{\mathbf{U}} := \frac{\partial g}{\partial \mathbf{A}}\big|_{\substack{\mathbf{A}=\mathbf{US},\\ \mathbf{B}=\mathbf{O}_{n\times r}}}$
Using AD, compute $\dot{\mathbf{V}} := \frac{\partial g}{\partial \mathbf{B}}\big|_{\substack{\mathbf{A}=\mathbf{US},\\ \mathbf{B}=\mathbf{O}_{n\times r}}}$
$\dot{\mathbf{V}}^\intercal := \dot{\mathbf{V}}^\intercal - (\dot{\mathbf{V}}^\intercal\mathbf{V})\mathbf{V}^\intercal$
```

$$\frac{\partial T_{ij}}{\partial B_{pq}} = \frac{\partial(\mathbf{A}\mathbf{V}^\intercal + \mathbf{U}\mathbf{B}^\intercal)_{ij}}{\partial B_{pq}} = \delta_{jp}U_{iq},$$

where δ_{ip} is the Kronecker delta. Applying the chain rule to (4.7), we get

$$\frac{\partial g}{\partial A_{pq}}\bigg|_{\substack{\mathbf{A}=\mathbf{US},\\ \mathbf{B}=\mathbf{O}_{n\times r}}} = \sum_{i,j} \frac{\partial f}{\partial T_{ij}}\bigg|_{\mathbf{T}=\mathbf{X}} \frac{\partial T_{ij}}{\partial A_{pq}}\bigg|_{\substack{\mathbf{A}=\mathbf{US},\\ \mathbf{B}=\mathbf{O}_{n\times r}}} = \sum_{i,j} \frac{\partial f}{\partial X_{ij}} \delta_{ip} V_{jq} = (\nabla f \mathbf{V})_{pq}\,,$$

$$\frac{\partial g}{\partial B_{pq}}\bigg|_{\substack{\mathbf{A}=\mathbf{US},\\ \mathbf{B}=\mathbf{O}_{n\times r}}} = \sum_{i,j} \frac{\partial f}{\partial T_{ij}}\bigg|_{\mathbf{T}=\mathbf{X}} \frac{\partial T_{ij}}{\partial B_{pq}}\bigg|_{\substack{\mathbf{A}=\mathbf{US},\\ \mathbf{B}=\mathbf{O}_{n\times r}}} = \sum_{i,j} \frac{\partial f}{\partial X_{ij}} \delta_{jp} U_{iq} = (\mathbf{U}^\intercal \nabla f)_{qp}\,.$$

Thus, a low-rank representation of the Riemannian gradient can be written as

$$\mathrm{P}_{\mathbf{X}}\nabla f = \begin{bmatrix}\mathbf{U} & \dot{\mathbf{U}}\end{bmatrix} \begin{bmatrix}\dot{\mathbf{V}} & \mathbf{V}\end{bmatrix}^\intercal$$

with

$$\begin{aligned} \dot{\mathbf{U}} &= \frac{\partial g}{\partial \mathbf{A}}\bigg|_{\substack{\mathbf{A}=\mathbf{US},\\ \mathbf{B}=\mathbf{O}_{n\times r}}}\,, \\ \dot{\mathbf{V}}^\intercal &= \frac{\partial g}{\partial \mathbf{B}^\intercal}\bigg|_{\substack{\mathbf{A}=\mathbf{US},\\ \mathbf{B}=\mathbf{O}_{n\times r}}} \left(\mathbf{I} - \mathbf{V}\mathbf{V}^\intercal\right). \end{aligned} \tag{4.8}$$

The algorithm to compute the Riemannian gradient is summarized in Algorithm 4.1.

4.3. Complexity of the approach. Let us estimate the complexity of computing $\dot{\mathbf{U}}$ and $\dot{\mathbf{V}}$ by the proposed approach, i.e., by defining the auxiliary function g and differentiating it with respect to $\mathbf{A}$ and $\mathbf{B}$.

PROPOSITION 4.1. *Let $f : \mathbb{R}^{m\times n} \to \mathbb{R}$ be a smooth function defined by a program p, which takes as input SVD decomposition of a matrix $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\intercal \in \mathbb{R}^{m\times n}$ and outputs the value $f(\mathbf{X})$ in $F = F(m, n, r)$ FLOP, which is polynomial with respect to the rank of the matrix $\boldsymbol{X}$ (i.e., the program p belongs to the P complexity class). Then, the complexity of using Algorithm* 4.1 *for computing delta terms $\dot{\mathbf{U}}$ and $\dot{\mathbf{V}}$ which define the Riemannian gradient $\mathrm{P}_{\mathbf{X}}\nabla f = \dot{\mathbf{U}}\mathbf{V}^\intercal + \mathbf{U}\dot{\mathbf{V}}^\intercal$ is $\mathcal{O}(F + nr^2)$.*

As an example, computing

$$f(\mathbf{X}) = \langle \mathbf{X}, \mathbf{X} \rangle\,, \quad \mathbf{X} = \mathbf{U}\mathbf{V}^\intercal, \quad \mathbf{U} \in \mathbb{R}^{m\times r}, \mathbf{V} \in \mathbb{R}^{n\times r},$$

leads to $F = \mathcal{O}\left((n+m)r^2\right)$ FLOP, since

$$\langle \mathbf{X}, \mathbf{X} \rangle = \operatorname{trace}(\mathbf{U}\mathbf{V}^\intercal\mathbf{V}\mathbf{U}^\intercal) = \operatorname{trace}\left((\mathbf{U}^\intercal\mathbf{U})(\mathbf{V}^\intercal\mathbf{V})\right).$$

Proof of Proposition 4.1. The auxiliary function $g(\mathbf{A}, \mathbf{B})$ can be constructed by feeding to p the factors of the matrix

$$\mathbf{A}\mathbf{V}^\intercal + \mathbf{U}\mathbf{B}^\intercal = \begin{bmatrix}\mathbf{A} & \mathbf{U}\end{bmatrix}\begin{bmatrix}\mathbf{V} & \mathbf{B}\end{bmatrix}^\intercal,$$

which is represented with the rank $2r$—twice larger than the rank r of the original matrix. As a result, the asymptotic complexity (as a function of n and r) of evaluating the function on such a matrix is still $\mathcal{O}(F)$. Thanks to the properties of AD, computing the derivatives of g with respect to the factors $\mathbf{A}$ and $\mathbf{B}$ has the same complexity $\mathcal{O}(F)$. Finally, computing the factor $\dot{\mathbf{V}}$ using (4.8) can be done in $\mathcal{O}(r^2 n)$, yielding the total complexity $\mathcal{O}(F + nr^2)$. ◻

For most functions used in practice, the asymptotic complexity F of executing the function at one point exceeds $\mathcal{O}(nr^2)$ and the total complexity of the proposed algorithm (as a function of n and r) equals to $\mathcal{O}(F + nr^2) = \mathcal{O}(F)$.

4.4. More general view of the proposed algorithm.

In this section, we look at the proposed algorithm from a more general perspective, trying to avoid specifics of the fixed-rank manifold. The main idea of the proposed algorithm is to introduce the auxiliary function (5.13) and express the desired Riemannian gradient grad $f(\mathbf{X})$ in terms of its derivatives (4.7). Note that we could have used an alternative auxiliary function[6]

$$h(\mathbf{C}, \mathbf{D}) = f(\mathbf{X} + \mathbf{C}\mathbf{V}^\intercal + \mathbf{U}\mathbf{D}^\intercal) = f((\mathbf{U}\mathbf{S} + \mathbf{C})\mathbf{V}^\intercal + \mathbf{U}\mathbf{D}^\intercal) = g(\mathbf{C} + \mathbf{U}\mathbf{S}, \mathbf{D}).$$

If one combines both arguments of the mapping $\mathcal{T}_\mathbf{X}(\mathbf{A}, \mathbf{B})$ (see (4.3)) into a single $(n+m)r$ dimensional vector $\mathbf{v}$, you can define an equivalent mapping $\widehat{\mathcal{T}}_\mathbf{X} \colon \mathbb{R}^{(n+m)r} \to T_\mathbf{X}\mathcal{M}_r$ and an alternative representation of the auxiliary function

$$\widehat{h}(\mathbf{v}) = f(\mathbf{X} + \widehat{\mathcal{T}}_\mathbf{X}(\mathbf{v})). \tag{4.9}$$

Thus, the proposed approach is equivalent to defining a mapping $\widehat{\mathcal{T}}_\mathbf{X}$ from the parametrization of the tangent space onto the tangent space itself, defining an auxiliary function $\widehat{h}(\mathbf{v})$ (4.9), computing its gradient using classical AD, and finally doing certain postprocessing of this gradient: reshaping, enforcing the gauge conditions as in (4.8). It is not surprising that we obtain a Riemannian gradient using these formulas, as informally the gradient of the auxiliary function (4.9) is the fastest ascent direction of $\widehat{h}(\mathbf{v})$ at $\mathbf{v} = 0$ and, hence, of f at $\mathbf{X}$ in the direction of all possible vectors from $T_\mathbf{X}\mathcal{M}_r$.

5. Automatic differentiation for the Riemannian gradient: Fixed-rank tensors.

In this section, we extend the results of section 4.2 to fixed-rank TT tensors, which is a generalization of fixed-rank matrices to multidimensional arrays.

[6] One can use the same derivation as in section 4.2 to prove that differentiating the alternative auxiliary function $\widehat{h}$ yields the same results.

5.1. The manifold of TT-tensors of fixed rank. A tensor $\mathbf{A} \in \mathbb{R}^{n_1 \times \dots \times n_d}$ is said to be represented in the TT-format [23] if each of its elements $A_{i_1 \dots i_d}$ is a product of d matrices:

$$A_{i_1 \dots i_d} = \mathbf{G}_1[i_1] \dots \mathbf{G}_d[i_d], \tag{5.1}$$

where for fixed $i_k = 1, \dots, n_k$, $\mathbf{G}_k[i_k]$ is an $r_{k-1} \times r_k$ matrix for any value of $k = 1, \dots, d$. We require $r_0 = r_d = 1$ such that $\mathbf{G}_1[i_1]$ is $1 \times r_1$ row vector and $\mathbf{G}_d[i_d]$ is $r_{d-1} \times 1$ column vector. The three-dimensional arrays $\mathbf{G}_k$ of sizes $r_{k-1} \times n_k \times r_k$, $k = 1, \dots, d$ are called *TT-cores* and the vector

$$\mathbf{r}_{\mathrm{TT}}(\mathbf{A}) = (r_1, \dots, r_{d-1})$$

is called the *TT-rank* of $\mathbf{A}$. For a more detailed discussion on the properties of the TT-format see [23].

Like in the matrix case (section 4), the set of tensors

$$\mathcal{M}_{\mathbf{r}} = \{\mathbf{A} \in \mathbb{R}^{n_1 \times \dots \times n_d} \mid \mathbf{r}_{\mathrm{TT}}(\mathbf{A}) = \mathbf{r}\}$$

forms a smooth manifold. To parametrize its tangent spaces, we need the notion of orthogonalization of the TT-cores. A TT-representation (5.1) is called μ-orthogonal, $\mu = 2, \dots, d-1$, if

$$\sum_{i_k=1}^{n_k} \mathbf{G}_k[i_k]^{\intercal} \mathbf{G}_k[i_k] = \mathbf{I}_{r_k} \tag{5.2}$$

for $k = 1, \dots, \mu - 1$, and

$$\sum_{i_k=1}^{n_k} \mathbf{G}_k[i_k] \mathbf{G}_k[i_k]^{\intercal} = \mathbf{I}_{r_{k-1}} \tag{5.3}$$

for $k = \mu + 1, \dots, d$. If $\mu = 1$ or $\mu = d$, we only require (5.3) or (5.2), respectively. The cores satisfying (5.2) and (5.3) are called, respectively, left- and right-orthogonal cores. TT-decomposition of a tensor is not unique, and for any $\mu = 1, \dots, d$ there exists a μ-orthogonal representation of a given tensor [24, section 4.2.1]. Moreover, for any $1 \le \mu_1 \le \mu_2 \le d$, the μ_1-orthogonal and μ_2-orthogonal decompositions can be constructed to share the left-orthogonal TT-cores $\mathbf{G}_1, \dots, \mathbf{G}_{\mu_1 - 1}$ satisfying (5.2) and the right-orthogonal TT-cores $\mathbf{G}_{\mu_2+1}, \dots, \mathbf{G}_d$ satisfying (5.3).

For a given tensor $\mathbf{X}$, one can define a set of left-orthogonal TT-cores $\mathbf{U}_1, \dots, \mathbf{U}_{d-1}$, right-orthogonal TT-cores $\mathbf{V}_2, \dots, \mathbf{V}_d$, and unrestricted TT-cores $\mathbf{S}_1, \dots, \mathbf{S}_d$ such that for any $\mu = 1, \dots, d$, there exists the following μ-orthogonal decomposition of the tensor

$$X_{i_1 \dots i_d} = \mathbf{U}_1[i_1] \dots \mathbf{U}_{\mu-1}[i_{\mu-1}] \mathbf{S}_\mu[i_\mu] \mathbf{V}_{\mu+1}[i_{\mu+1}] \dots \mathbf{V}_d[i_d]. \tag{5.4}$$

Using the left-orthogonal TT-cores $\mathbf{U}_1, \dots, \mathbf{U}_{d-1}$ and the right-orthogonal TT-cores $\mathbf{V}_2, \dots, \mathbf{V}_d$ of tensor $\mathbf{X} \in \mathcal{M}_{\mathbf{r}}$, one may parametrize the tangent space $T_{\mathbf{X}}\mathcal{M}_r$ as follows:

$$\begin{aligned} T_{\mathbf{X}}\mathcal{M}_r = \Big\{ & \mathbf{T} \in \mathbb{R}^{n_1 \times \dots \times n_d} : T_{i_1 \dots i_d} = \dot{\mathbf{S}}_1[i_1]\, \mathbf{V}_2[i_2] \, \dots \mathbf{V}_d[i_d] \\ & + \mathbf{U}_1[i_1] \dot{\mathbf{S}}_2[i_2]\, \mathbf{V}_3[i_3] \dots \mathbf{V}_d[i_d] + \dots + \mathbf{U}_1[i_1] \, \dots \mathbf{U}_{d-1}[i_{d-1}] \, \dot{\mathbf{S}}_d[i_d], \\ & \dot{\mathbf{S}}_k \in \mathbb{R}^{r_{k-1} \times n_k \times r_k},\ k = 1, \dots, d,\ r_0 = r_d = 1 \Big\}. \end{aligned} \tag{5.5}$$

In what follows, we refer to the tensors $\dot{\mathbf{S}}_1, \dots, \dot{\mathbf{S}}_d$ that define an element of the tangent space as *delta-terms*.

Additional gauge conditions are usually introduced[7] to uniquely parametrize elements of the tangent space:

$$\sum_{i_k=1}^{n_k} \mathbf{U}_k[i_k]^\intercal \dot{\mathbf{S}}_k[i_k] = 0, \quad k = 1, \dots, d-1. \tag{5.6}$$

In what follows, we always assume that the deltas $\dot{\mathbf{S}}_1, \dot{\mathbf{S}}_2, \dots, \dot{\mathbf{S}}_d$ that define an element of the tangent space obey the gauge conditions (5.6).

Note that in (5.5), the expression for an element of the tangent space is formally represented as a sum of d TT-tensors, each of TT-rank $\mathbf{r}$, and, hence, can be represented as a single TT-tensor with the TT-rank $\mathbf{r} + \cdots + \mathbf{r} = d\mathbf{r}$ [23, section 4.1]. Nevertheless, thanks to the common cores, it can be represented with the TT-rank equal to $2\mathbf{r}$. Indeed, by directly multiplying the block matrices, one can verify that

(5.7)
$$T_{i_1 \dots i_d} = \begin{bmatrix} \dot{\mathbf{S}}_1[i_1] & \mathbf{U}_1[i_1] \end{bmatrix} \begin{bmatrix} \mathbf{V}_2[i_2] & \\ \dot{\mathbf{S}}_2[i_2] & \mathbf{U}_2[i_2] \end{bmatrix} \dots \begin{bmatrix} \mathbf{V}_{d-1}[i_{d-1}] & \\ \dot{\mathbf{S}}_{d-1}[i_{d-1}] & \mathbf{U}_{d-1}[i_{d-1}] \end{bmatrix} \begin{bmatrix} \mathbf{V}_d[i_d] \\ \dot{\mathbf{S}}_d[i_d] \end{bmatrix}.$$

For convenience, we also introduce a function that maps the delta terms $\dot{\mathbf{S}}_k$ to an element of the tangent space

$$\mathcal{T}_{\mathbf{X}} : \mathbb{R}^{1 \times n_1 \times r_1} \times \mathbb{R}^{r_1 \times n_2 \times r_2} \times \cdots \times \mathbb{R}^{r_{d-2} \times n_{d-1} \times r_{d-1}} \times \mathbb{R}^{r_{d-1} \times n_d \times 1} \to T_{\mathbf{X}} \mathcal{M}_r,$$

namely,

$$\mathbf{T} = \mathcal{T}_{\mathbf{X}}(\dot{\mathbf{S}}_1, \dots, \dot{\mathbf{S}}_d), \tag{5.8}$$

as is defined in (5.7). The following proposition gives the explicit representation of a general tensor projected onto the tangent plane of $\mathcal{M}_{\mathbf{r}}$.

Proposition 5.1 ([24, equation (4.17)]). *The orthogonal projection* $\mathrm{P}_{\mathbf{X}} \boldsymbol{Z}$ *of a given tensor* $\boldsymbol{Z} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ *onto the tangent space* $T_{\mathbf{X}} \mathcal{M}_{\mathbf{r}}$ *is defined as an element of the tangent space* (5.5) *with* $\dot{\mathbf{S}}_k$*:*

(5.9)
$$\underbrace{\dot{\mathbf{S}}_k[j_k]}_{r_{k-1} \times r_k} = \sum_{i_1, \dots, i_d} \Bigg(\underbrace{\mathbf{U}_1[i_1] \dots \mathbf{U}_{k-1}[i_{k-1}] \left(\mathbf{I}_{r_{k-1}} \delta_{j_k i_k} - \mathbf{U}_k[j_k] \mathbf{U}_k[i_k]^\intercal \right)}_{1 \times r_{k-1}} \Bigg)^\intercal Z_{i_1 \dots i_d},$$
$$\Bigg(\underbrace{\mathbf{V}_{k+1}[i_{k+1}] \dots \mathbf{V}_d[i_d]}_{r_k \times 1} \Bigg)^\intercal, \quad k = 1, \dots, d-1,$$

and $\dot{\mathbf{S}}_d$ *as*

$$\mathbf{S}[i_d] = \sum_{i_1, \dots, i_{d-1}} \mathbf{U}_1[i_1] \dots \mathbf{U}_{d-1}[i_{d-1}] Z_{i_1 \dots i_d}.$$

For a more detailed discussion of the manifold of fixed TT-rank tensors (including derivations of the above equations) see, e.g., section 4.3–4.4 of [24].

[7] These gauge conditions generalize the orthogonality constraint $\mathbf{V}^\intercal \dot{\mathbf{V}} = \mathbf{O}_{r \times r}$ in the definition of the matrix tangent space (4.1) to the tensor case.

Algorithm 5.1 Converting delta notation to TT-cores (implementation of (5.8)).

```
Require: TT-tensor $\mathbf{X}$ defined by the TT-cores $\mathbf{G}_k$, tensors $\dot{\mathbf{S}}_k$ that define the tangent space element $\mathbf{T} \in T_{\mathbf{X}}\mathcal{M}_{\mathbf{r}}$ (see (5.7))
Ensure: $\widehat{\mathbf{G}}_k$, $k = 1, \dots, d$ — TT-cores of $\mathbf{T} = \mathcal{T}_{\mathbf{X}}(\dot{\mathbf{S}}_1, \dots, \dot{\mathbf{S}}_d)$
Compute, respectively, left- and right-orthogonal $\{\mathbf{U}_k\}_{k=1}^{d-1}$ and $\{\mathbf{V}_k\}_{k=2}^{d}$, and tensors $\{\mathbf{S}_k\}_{k=1}^{d}$ as in (5.4)
for $i_1 = 1$ to $n_1$ do
    $\widehat{\mathbf{G}}_1[i_1] = \begin{bmatrix} \dot{\mathbf{S}}_1[i_1] & \mathbf{U}_1[i_1] \end{bmatrix}$
for $k = 2$ to $d - 1$ do
    for $i_k = 1$ to $n_k$ do
        $\widehat{\mathbf{G}}_k[i_k] = \begin{bmatrix} \mathbf{V}_k[i_k] & \\ \dot{\mathbf{S}}_k[i_k] & \mathbf{U}_k[i_k] \end{bmatrix}$
for $i_d = 1$ to $n_d$ do
    $\widehat{\mathbf{G}}_d[i_d] = \begin{bmatrix} \mathbf{V}_d[i_d] \\ \dot{\mathbf{S}}_d[i_d] \end{bmatrix}$
```

5.2. Automatic differentiation. Let us find the Riemannian gradient of a function $f : \mathbb{R}^{n_1 \times \cdots \times n_d} \to \mathbb{R}$ at a point $\mathbf{X}$. Similarly to the matrix case, we consider an auxiliary function using (5.8):

$$g \overset{\text{def}}{=} f \circ \mathcal{T}_{\mathbf{X}}.$$

Note that the intuitive explanation of the proposed method provided in section 4.4 still applies in this case.

In particular, we have

$$\mathbf{T} = \mathcal{T}_{\mathbf{X}}(\mathbf{S}_1, \mathbf{O}_2, \dots, \mathbf{O}_d) = \mathbf{X},$$

where $\mathbf{O}_k$, $k = 2, \dots, d$ are zero tensors of appropriate sizes and $\mathbf{S}_1$ is defined in (5.4) for $\mu = 1$. As a result,

$$g(\mathbf{S}_1, \mathbf{O}_2, \dots, \mathbf{O}_d) = f(\mathbf{X}).$$

Consider the derivative of $g(\mathbf{R}_1, \dots, \mathbf{R}_d)$ with respect to $\mathbf{R}_k$ at a point $\mathcal{R}_0 = (\mathbf{S}_1, \mathbf{O}_2, \dots, \mathbf{O}_d)$:

$$\begin{aligned} \frac{\partial g}{\partial \mathbf{R}_k[i_k]}(\mathcal{R}_0) &= \sum_{i_1, \dots, i_{k-1}, i_{k+1}, \dots, i_d} \frac{\partial f}{\partial T_{i_1 \dots i_d}}(\mathbf{X}) \frac{\partial T_{i_1 \dots i_d}}{\partial \mathbf{R}_k[i_k]}(\mathcal{R}_0) \\ &= \sum_{i_1, \dots, i_{k-1}, i_{k+1}, \dots, i_d} (\mathbf{U}_1[i_1] \dots \mathbf{U}_{k-1}[i_{k-1}])^\intercal \frac{\partial f}{\partial \mathcal{X}_{i_1 \dots i_d}} (\mathbf{V}_{k+1}[i_{k+1}] \dots \mathbf{V}_d[i_d])^\intercal . \end{aligned} \tag{5.10}$$

By comparing expressions (5.9) and (5.10), it is easy to see that the $\dot{\mathbf{S}}_k$ that defines the Riemannian gradient $\mathrm{P}_{\mathbf{X}} \nabla f$ can be computed as

$$\begin{aligned} \dot{\mathbf{S}}_k[i_k] &= \frac{\partial g}{\partial \mathbf{R}_k[i_k]}(\mathcal{R}_0) - \mathbf{U}_k[i_k] \sum_{j_k} \mathbf{U}_k^\intercal[j_k] \frac{\partial g}{\partial \mathbf{R}_k[j_k]}(\mathcal{R}_0), \quad k = 1, \dots, d-1, \\ \dot{\mathbf{S}}_d[i_d] &= \frac{\partial g}{\partial \mathbf{R}_d[i_d]}(\mathcal{R}_0). \end{aligned} \tag{5.11}$$

Algorithm 5.2 Computing the Riemannian gradient for low-rank tensors via AD.

```
Require: $\{\mathbf{G}_k\}_{k=1}^d$—TT-cores of $\mathbf{X}$, $p(\widehat{\mathbf{G}}_1,\ldots,\widehat{\mathbf{G}}_d)$—Python implementation of
    $f(\widehat{\mathbf{X}})$ for a point $\widehat{\mathbf{X}}$ given by TT-cores $\widehat{\mathbf{G}}_1,\ldots,\widehat{\mathbf{G}}_d$.
Ensure: The TT-cores $\{\mathbf{J}_k\}_{k=1}^d$ of the Riemannian gradient grad $f(\mathbf{X})$
 1: For $\mathbf{X}$, compute, respectively, left- and right-orthogonal $\{\mathbf{U}_k\}_{k=1}^{d-1}$, $\{\mathbf{V}_k\}_{k=2}^d$ and
    $\{\mathbf{S}_k\}_{k=1}^d$ as in (5.4).
 2: function g$(\mathbf{R}_1,\ldots,\mathbf{R}_d)$
 3:     Run Algorithm 5.1 passing as input $\{\mathbf{G}_k\}_{k=1}^d$, $\{\mathbf{R}_k\}_{k=1}^d$ and write the output
        into $\{\widehat{\mathbf{G}}_k\}_{k=1}^d$
 4:     return $p(\widehat{\mathbf{G}}_1,\ldots,\widehat{\mathbf{G}}_d)$
 5: Using AD, compute $\dot{\mathbf{S}}_k := \frac{\partial g}{\partial \mathbf{R}_k}\Big|_{(\mathbf{R}_1,\mathbf{R}_2,\ldots,\mathbf{R}_d)=(\mathbf{S}_1,\mathbf{O}_2,\ldots,\mathbf{O}_d)}$ for $k=1,\ldots,d$
 6: for $k \leftarrow 1$ to $d-1$ do
 7:     $\mathbf{D}_k := \mathtt{reshape}(\dot{\mathbf{S}}_k,\ (r_{k-1}n_k,\ r_k))$
 8:     $\mathbf{U}_k^{\mathrm{L}} := \mathtt{reshape}(\mathbf{U}_k,\ (r_{k-1}n_k,\ r_k))$
 9:     $\mathbf{D}_k := \mathbf{D}_k + \mathbf{U}_k^{\mathrm{L}}\left(\left(\mathbf{U}_k^{\mathrm{L}}\right)^{\mathsf{T}}\mathbf{D}_k\right)$   ▷ See (5.11). Parentheses indicate the order of
        operations
10:     $\dot{\mathbf{S}}_k := \mathtt{reshape}(\mathbf{D}_k, (r_{k-1}, n_k, r_k))$
11: Run Algorithm 5.1 passing as input $\{\mathbf{G}_k\}_{k=1}^d$ and $\{\dot{\mathbf{S}}_k\}_{k=1}^d$ and write the output
    TT-cores into $\{\mathbf{J}_k\}_{k=1}^d$
```

The algorithm for computing the Riemannian gradient in the TT case is listed in Algorithm 5.2. Hereinafter we use a `reshape` [25] function that changes the shape of an array, preserving the values and the order of elements, where by the order of elements of $\mathbf{X} \in \mathbb{R}^{n_1\times\cdots\times n_d}$ we imply the following ordering:

$$(i_1,\ldots,i_d) \mapsto 1 + \sum_{\alpha=1}^{d}(i_\alpha - 1)\prod_{\beta=\alpha+1}^{d} n_\beta.$$

Let us estimate the complexity of Algorithm 5.2

PROPOSITION 5.2. *Let $f : \mathbb{R}^{n_1\times\ldots\times n_d} \to \mathbb{R}$ be a smooth function defined by a program p, which takes as input TT-cores of the tensor $\boldsymbol{X}$ and outputs the value $f(\boldsymbol{X})$ in F FLOP, which is polynomial with respect to the TT-ranks of the tensor $\boldsymbol{X}$ (i.e., the program p belongs to the P complexity class). Then, the complexity of using Algorithm* 5.2 *for computing the TT-cores of the Riemannian gradient* $\mathrm{P}_{\boldsymbol{X}}\nabla f$ *is $\mathcal{O}(F + dnr^3)$, where $n = \max_{k=1,\ldots,d} n_k$, $r = \max_{k=1,\ldots,d-1} r_k$.*

Proof. Let us estimate the complexity of each step of Algorithm 5.2.

Step 1 consists of orthogonalizing the cores of the tensor $\mathbf{X}$ and can be done in $\mathcal{O}(dnr^3)$ FLOP [23, end of section 3].

Steps 3 *and* 11 are running Algorithm 5.1, which consists of copying and rearranging some of the arrays which already exist in the memory. Therefore, it has linear complexity with respect to the sizes of the arrays, i.e., at most $\mathcal{O}(dnr^2)$.

Step 4 computes the output of the program p on TT-cores $\widehat{\mathbf{G}}_1,\ldots,\widehat{\mathbf{G}}_d$. Under the assumptions of the statement, the complexity F of the function evaluation is polynomial with respect to the TT-rank r. Let q be the degree of this polynomial. Since the TT-cores $\widehat{\mathbf{G}}_1,\ldots,\widehat{\mathbf{G}}_d$ define a TT-tensor with TT-ranks $2\mathbf{r}$—twice larger when compared to the original TT-rank $\mathbf{r}$—the program p will be executed on these

TT-cores with the complexity $\mathcal{O}(2^q F) = \mathcal{O}(F)$. Thus, the complexity of evaluating the function g at a given point is at most $\mathcal{O}(F)$.

Step 5 uses classic AD to compute the gradient of the function g with respect to its arguments. Since the asymptotic complexity of the classical AD equals the asymptotic complexity of computing the function at one point [26], this substep can also be done in $\mathcal{O}(F)$ FLOP.

Steps 7, 8, *and* 10 consist of repeating the reshape operation $d-1$ times. The reshape operation can be done with constant complexity and in the worst case (when doing this operation in-place is not available) has the complexity equal to the size of arrays, i.e., $\mathcal{O}(nr^2)$ per iteration.

Step 9 consists of evaluating the following expression $d-1$ times: $\mathbf{D}_k := \mathbf{D}_k + \mathbf{U}_k^L((\mathbf{U}_k^L)^\intercal\, \mathbf{D}_k)$. The multiplication $(\mathbf{U}_k^L)^\intercal\, \mathbf{D}_k$ of a $r_k \times r_{k-1}n_k$ matrix times a $r_{k-1}n_k \times r_k$ matrix results into a $r_k \times r_k$ matrix and costs $\mathcal{O}(r_{k-1}r_k^2 n_k)$. The remaining operations are of the same or smaller asymptotic complexity. Thus, updating all $\mathbf{D}_k$ can be done in $\mathcal{O}(dnr^3)$ FLOP.

Summing the complexity across all steps yields the total complexity $\mathcal{O}(F + dnr^3)$. ◻

For most functions used in practice, the asymptotic complexity F of executing the function at one point exceeds $\mathcal{O}(dnr^3)$ and the total complexity (as a function of n, d, and r) of the proposed algorithm equals to $\mathcal{O}(F + dnr^3) = \mathcal{O}(F)$. For example, the functions listed at the end of section 3 (except for the recurrent neural network example) and their combinations such as

$$f(\mathbf{X}) = \|\mathrm{P}_\Omega(\mathbf{X} - \mathbf{A})\|^2 + \lambda\|\mathbf{X}\|^2$$

are at least as expensive to evaluate as $\mathcal{O}(dnr^3)$.

5.3. Stop-gradient and a wider class of functionals.

Suppose that we want to calculate projection to a tangent plane that cannot be easily associated with a Riemannian gradient of a functional. As an example, in [4], to solve a linear system $\mathrm{A}\mathbf{X} = \mathbf{F}$, a preconditioned version of the Riemannian gradient descent was considered:

$$\mathbf{X}_{k+1} = \mathbf{X}_k - \tau_k \mathrm{P}_{\mathbf{X}_k}\mathrm{B}\left(\mathrm{A}\mathbf{X}_k - \mathbf{F}\right), \tag{5.12}$$

where B is a preconditioner and $\tau_k \in \mathbb{R}$ is an iteration parameter. If B is an identity operator and A is symmetric positive-definite, then the iteration (5.12) is a Riemannian gradient descent associated with the function

$$f_\mathrm{A}(\mathbf{X}) = \frac{1}{2}\left\langle \mathrm{A}\mathbf{X}, \mathbf{X}\right\rangle - \left\langle \mathbf{F}, \mathbf{X}\right\rangle. \tag{5.13}$$

The problem is that to obtain $\mathrm{P}_{\mathbf{X}_k}\mathrm{B}(\mathrm{A}\mathbf{X}_k - \mathbf{F})$, we cannot simply calculate the Riemannian gradient of (5.13) with BA instead of A, and B$\mathbf{F}$ instead of $\mathbf{F}$, since BA is, in general, not symmetric even if both A and B are. A similar problem arises for preconditioned eigensolvers. To overcome it, we will use the notion of the *stop-gradient operator*, which is available in most AD frameworks.

The stop-gradient operator $c(\mathbf{X})$ is formally defined by the following two properties: $c(\mathbf{X}) = \mathbf{X}$ and $\nabla c(\mathbf{X}) = \mathbf{O}$—zero tensor of the same size as $\mathbf{X}$. It allows avoiding differentiating some parts of an expression when applying AD. For example, for $x \in \mathbb{R}$ the derivative of $g(x) \equiv f(xc(x))$ is $g'(x) = f'(x^2)$ instead of $f'(x^2)2x$.

We, thus, can (in the code) replace the function f_A with $h_{\mathrm{A},\mathrm{B}}$:

$$h_{\mathrm{A},\mathrm{B}}(\mathbf{X}) = \left\langle \mathrm{BA}\, c(\mathbf{X}), \mathbf{X}\right\rangle - \left\langle \mathrm{B}\mathbf{F}, \mathbf{X}\right\rangle.$$

As a result, we obtain

$$\mathrm{P}_{\mathbf{X}}\nabla h_{\mathrm{A},\mathrm{B}}(\mathbf{X}) = \mathrm{P}_{\mathbf{X}}\mathrm{B}\left(\mathrm{A}\mathbf{X} - \mathbf{F}\right), \tag{5.14}$$

so we can simply apply the proposed AD approach to $h_{\mathrm{A},\mathrm{B}}(\mathbf{X})$. Note that

$$\langle \mathrm{BA}\, c(\mathbf{X}), \mathbf{X}\rangle = \langle \mathrm{A}c(\mathbf{X}), \mathrm{B}^{\intercal}\mathbf{X}\rangle$$

and it can be implemented in $\mathcal{O}(dnR_AR_Br^3 + dn^2(R_A+R_B)R_AR_Br^2)$ FLOP. Hence, using the proposed AD, we can calculate the Riemannian gradient (5.14) with the same asymptotic complexity. If B is a sum of ρ_{B} rank-1 terms, for example, for a preconditioner based on exponential sums [27, 28], then the complexity can be additionally reduced to $\mathcal{O}(dnR_A\rho_{\mathrm{B}}r^3 + dn^2\rho_{\mathrm{B}}R_A^2r^2)$.

6. Approximate Hessian-by-vector product.

In this section, we show how to compute the product between the approximate Riemannian Hessian and a vector from the tangent space (3.4).

In the classical AD, there are two main ways of implementing Hessian-by-vector products given first-order AD implementation. The first approach consists of computing the gradient $\nabla f(\mathbf{x})$, then defining an auxiliary function $w\colon \mathbb{R}^n \to \mathbb{R}$, $w(\mathbf{x}) = \langle \nabla f(\mathbf{x}),\ \mathbf{z}\rangle$, and finally using first-order AD on the auxiliary function $\nabla^2 f(\mathbf{x})\ \mathbf{z} = \nabla w(\mathbf{x})$. The second approach consists of defining an auxiliary function $h\colon \mathbb{R} \to \mathbb{R}^n$, $h(t) = \nabla_{\mathbf{x}} f(\mathbf{x}+t\mathbf{z})$ by using first-order AD at the point $\mathbf{x}+t\mathbf{z}$, and then using forward mode AD,[8] on the auxiliary function h at the point $t=0$ to get the Hessian-by-vector product $\nabla^2 f(\mathbf{x})\ \mathbf{z} = h'(t)|_{t=0}$ (see, e.g., [29] for more details).

Both of these classical approaches can be generalized to the Riemannian case. Here we focus on the first approach, as the second approach requires forward mode AD which is not natively supported by the AD library we use for numerical experiments (TensorFlow). In the generalization of the first approach we additionally use the fact that when computing the auxiliary scalar-product function w, we are working with two vectors from the same tangent space. This allows us to compute their inner product more efficiently than in the general case.

Recall the definition of the approximate Riemannian Hessian-by-vector product

$$\mathrm{H}_{\mathbf{X}}[\mathbf{Z}] = \mathrm{P}_{\mathbf{X}}\nabla^2 f(\mathbf{X})\,\mathbf{Z}, \quad \mathbf{Z} \in T_{\mathbf{X}}\mathcal{M}. \tag{6.1}$$

Note that the exact (nonapproximate) Riemannian Hessian (3.3) also includes the term for the derivative of the projection operator $\mathrm{P}_{\mathbf{X}}$ with respect to the tensor $\mathbf{X}$, which we ignore in (6.1).

Let us transform (6.1) using the fact that $\mathbf{Z} \in T_{\mathbf{X}}\mathcal{M}$, which implies $\mathbf{Z} = \mathrm{P}_{c(\mathbf{X})}\mathbf{Z}$. Note that we use the stop-gradient operator c defined in section 5.3 to make sure that we are computing the approximate Riemannian Hessian, i.e., that we are not differentiating the projection operator $\mathrm{P}_{c(\mathbf{X})}$. In this case,

$$\mathrm{H}_{\mathbf{X}}[\mathbf{Z}] = \mathrm{P}_{\mathbf{X}}\nabla^2 f(\mathbf{X})\ \mathbf{Z} = \mathrm{P}_{\mathbf{X}}\frac{\partial}{\partial\mathbf{X}}\left\langle \nabla f, \mathrm{P}_{c(\mathbf{X})}\mathbf{Z}\right\rangle.$$

Using the symmetry of the orthogonal projection $\mathrm{P}_{c(\mathbf{X})}$, we may write

$$\mathrm{H}_{\mathbf{X}}[\mathbf{Z}] = \mathrm{P}_{\mathbf{X}}\frac{\partial}{\partial\mathbf{X}}\left\langle \nabla f, \mathrm{P}_{c(\mathbf{X})}\mathbf{Z}\right\rangle = \mathrm{P}_{\mathbf{X}}\frac{\partial}{\partial\mathbf{X}}\left\langle \mathrm{P}_{c(\mathbf{X})}\nabla f, \mathbf{Z}\right\rangle.$$

[8] Using reverse mode AD would not be efficient in this case as the function h has nonscalar output.

Assume that we have access to the Riemannian gradient with the stop-gradient operator applied to the projection $\mathrm{P}_{c(\mathbf{X})}\nabla f$ (see below on how to obtain it). Then, we can compute

$$w(\mathbf{X}) = \left\langle \mathrm{P}_{c(\mathbf{X})}\nabla f, \mathbf{Z}\right\rangle \tag{6.2}$$

and use the first-order Riemannian AD to find $\mathrm{H}_{\mathbf{X}}[\mathbf{Z}] = \mathrm{P}_{\mathbf{X}}\nabla w(\mathbf{X})$—the desired approximate Riemannian Hessian-by-vector product.

Note that (6.2) is a scalar product of two vectors belonging to the same tangent plane. Let us consider this operation in more detail. Suppose we are given two tensors $\mathbf{Y}, \mathbf{Z} \in T_{\mathbf{X}}\mathcal{M}$. If $\mathcal{M}$ is a manifold of fixed-rank matrices, we can parametrize $\mathbf{Y}$ and $\mathbf{Z}$ with the matrices $\dot{\mathbf{U}}_{\mathbf{Y}}, \dot{\mathbf{V}}_{\mathbf{Y}}$ and $\dot{\mathbf{U}}_{\mathbf{Z}}, \dot{\mathbf{V}}_{\mathbf{Z}}$ (see (4.1)). Hence,

$$\langle \mathbf{Y}, \mathbf{Z}\rangle = \left\langle \dot{\mathbf{U}}_{\mathbf{Y}}\mathbf{V}^\intercal + \mathbf{U}\dot{\mathbf{V}}_{\mathbf{Y}}^\intercal, \dot{\mathbf{U}}_{\mathbf{Z}}\mathbf{V}^\intercal + \mathbf{U}\dot{\mathbf{V}}_{\mathbf{Z}}^\intercal\right\rangle = \left\langle \dot{\mathbf{U}}_{\mathbf{Z}}, \dot{\mathbf{U}}_{\mathbf{Y}}\right\rangle + \left\langle \dot{\mathbf{V}}_{\mathbf{Z}}, \dot{\mathbf{V}}_{\mathbf{Y}}\right\rangle . \tag{6.3}$$

Similarly, if $\mathcal{M}$ is the manifold of fixed-rank TT tensors and $\mathbf{Y}, \mathbf{Z} \in T_{\mathbf{X}}\mathcal{M}$ are parametrized as in (5.5) by $\{\dot{\mathbf{S}}_k^{\mathbf{Y}}\}_{k=1}^d$ and $\{\dot{\mathbf{S}}_k^{\mathbf{Z}}\}_{k=1}^d$, respectively, then

$$\langle \mathbf{Y}, \mathbf{Z}\rangle = \sum_{k=1}^{d} \left\langle \dot{\mathbf{S}}_k^{\mathbf{Y}}, \dot{\mathbf{S}}_k^{\mathbf{Z}}\right\rangle . \tag{6.4}$$

Note that (6.3) and (6.4) for matrices and tensors from the same tangent plane lead to faster computation of scalar products than for two general tensors of the same rank (see [24, section 4.4.4]).

One might think that the first-order Riemannian AD described in section 4.2 and 5.2 yields $\mathrm{P}_{\mathbf{X}}\nabla f$ instead of $\mathrm{P}_{c(\mathbf{X})}\nabla f$ and thus cannot be utilized here. However, since first-order Riemannian AD works by differentiating at $\mathbf{X}$ the auxiliary function g defined on a linear space $T_{\mathbf{X}}\mathcal{M}_r$, a Riemannian gradient obtained this way lacks any information about the nonlinearity of the manifold. So, the method for computing the Riemannian gradient (sections 4.2 and 5.2) actually yields $\mathrm{P}_{c(\mathbf{X})}\nabla f$. This nuance is irrelevant when computing the first-order Riemannian gradient because the two quantities coincide in value, but it becomes important when differentiating through this operation. Thus, we can reuse the proposed first-order Riemannian gradient to compute the product between the approximate Riemannian Hessian and a given vector with the method described above.

The algorithms to compute the multiplication of the approximate Riemannian Hessian by a vector are summarized in Algorithm 6.1 for the matrix case and in Algorithm 6.2 for the tensor case. Note that they require only a few additional operations compared to the algorithm for computing the Riemannian gradient.

Let us estimate the complexity of the proposed algorithm.

PROPOSITION 6.1. *Let $f : \mathbb{R}^{m\times n} \to \mathbb{R}$ be a smooth function defined by a program p, which takes as input SVD decomposition of a matrix $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\intercal \in \mathbb{R}^{m\times n}$ and outputs the value $f(\mathbf{X})$ in $F = F(m, n, r)$ FLOP, which is polynomial with respect to the rank of the matrix $\boldsymbol{X}$ (i.e., the program p belongs to the P complexity class). Then, the complexity of using Algorithm* 6.1 *for computing delta terms $\dot{\mathbf{U}}$ and $\dot{\mathbf{V}}$, which define the product of the approximate Riemannian Hessian by a given vector (for the manifold of fixed-rank matrices) $\mathrm{H}_{\mathbf{X}}[\mathbf{Z}] = \mathrm{P}_{\mathbf{X}}\nabla^2 f(\mathbf{X})\,\mathbf{Z} = \dot{\mathbf{U}}\mathbf{V}^\intercal + \mathbf{U}\dot{\mathbf{V}}^\intercal$ is $\mathcal{O}(F + nr^2)$.*

Proof. The algorithm for computing the approximate Riemannian Hessian-by-vector product in the matrix case (Algorithm 6.1) is similar to the algorithm for

Algorithm 6.1 Computing the approximate Riemannian Hessian-by-vector product for low-rank matrices via AD.

```
Require: $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\intercal \in \mathbb{R}^{m\times n}$, matrices $\dot{\mathbf{U}}^{\mathbf{Z}}, \dot{\mathbf{V}}^{\mathbf{Z}}$ that define $\mathbf{Z} = \dot{\mathbf{U}}^{\mathbf{Z}}\mathbf{V}^\intercal + \mathbf{U}(\dot{\mathbf{V}}^{\mathbf{Z}})^\intercal \in T_{\mathbf{X}}\mathcal{M}_r$ (see (4.1)), $p(\mathbf{L}, \mathbf{R})$ – implementation of evaluating $f$ at $\mathbf{L}\mathbf{R}^\intercal$ for any $\mathbf{L} \in \mathbb{R}^{m\times 2r}$ and $\mathbf{R} \in \mathbb{R}^{n\times 2r}$.
Ensure: $\dot{\mathbf{U}}, \dot{\mathbf{V}}$ such that $\mathrm{H}_{\mathbf{X}}[\mathbf{Z}] = \mathrm{P}_{\mathbf{X}}\nabla^2 f(\mathbf{X})\,\mathbf{Z} = \dot{\mathbf{U}}\mathbf{V}^\intercal + \mathbf{U}\dot{\mathbf{V}}^\intercal$
 1: function g($\mathbf{A}, \mathbf{B}$)
 2:     return $p([\mathbf{U}\,\mathbf{A}],\ [\mathbf{B}\,\mathbf{V}])$
 3: function w($\widehat{\mathbf{A}}, \widehat{\mathbf{B}}$)
 4:     $\dot{\mathbf{U}} := \frac{\partial g}{\partial \mathbf{A}}\Big|_{(\mathbf{A},\mathbf{B})=(\widehat{\mathbf{A}},\widehat{\mathbf{B}})}$ using AD
 5:     $\dot{\mathbf{V}} := \frac{\partial g}{\partial \mathbf{B}}\Big|_{(\mathbf{A},\mathbf{B})=(\widehat{\mathbf{A}},\widehat{\mathbf{B}})}$ using AD
 6:     $\dot{\mathbf{V}}^\intercal := \dot{\mathbf{V}}^\intercal - (\dot{\mathbf{V}}^\intercal\mathbf{V})\mathbf{V}^\intercal$
 7:     return $\left\langle \dot{\mathbf{U}}^{\mathbf{Z}}, \dot{\mathbf{U}} \right\rangle + \left\langle \dot{\mathbf{V}}^{\mathbf{Z}}, \dot{\mathbf{V}} \right\rangle$
 8: $\dot{\mathbf{U}} := \frac{\partial w}{\partial \mathbf{A}}\big|_{(\mathbf{A},\mathbf{B})=(\mathbf{U}\mathbf{S},\mathbf{O})}$ using AD
 9: $\dot{\mathbf{V}} := \frac{\partial w}{\partial \mathbf{B}}\big|_{(\mathbf{A},\mathbf{B})=(\mathbf{U}\mathbf{S},\mathbf{O})}$ using AD
10: $\dot{\mathbf{V}}^\intercal := \dot{\mathbf{V}}^\intercal - (\dot{\mathbf{V}}^\intercal\mathbf{V})\mathbf{V}^\intercal$
```

computing the Riemannian gradient (Algorithm 4.1): subfunctions g are identical in both algorithms, and steps 4–6 and 8–10 in Algorithm 6.1 are identical to steps 3–5 Algorithm 4.1 (so it at most doubles the work and does not affect the asymptotic complexity). The only new operation is computing the dot product between the tangent space elements (step 7), which takes $\mathcal{O}(nr^2)$ arithmetic operations. Thus, computing the approximate Riemannian Hessian-by-vector product asymptotic complexity is still $\mathcal{O}(F + nr^2)$. □

As is noted at the end of section 4.3, for most practical functions $f(\mathbf{X})$ the complexity F of evaluating the function at a single point dominates the added complexity $\mathcal{O}(nr^2)$ of the proposed algorithm, making the total complexity of the algorithm coincide with the complexity of evaluating the function: $\mathcal{O}(F + nr^2) = \mathcal{O}(F)$.

PROPOSITION 6.2. *Let $f : \mathbb{R}^{n_1\times\cdots\times n_d} \to \mathbb{R}$ be a smooth function defined by a program p, which takes as input TT-cores of the tensor $\boldsymbol{X}$ and outputs the value $f(\boldsymbol{X})$ in F FLOP, which is polynomial with respect to the TT-ranks of the tensor $\boldsymbol{X}$ (i.e., the program p belongs to the P complexity class). Then, the complexity of Algorithm* 6.2 *for computing the product of the approximate Riemannian Hessian by a given vector (for the manifold of tensors of fixed TT-rank)* $\mathrm{P}_{\boldsymbol{X}}\nabla^2 f(\boldsymbol{X})\ \boldsymbol{Z}$ *is $\mathcal{O}(F + dnr^3)$, where $n = \max_{k=1,\dots,d} n_k$, $r = \max_{k=1,\dots,d-1} r_k$.*

Proof. Similarly to the first-order case, let us estimate the complexity of each step of Algorithm 6.2.

Steps 1, 2, 3, 4 define the function $g(\mathbf{R}_1, \dots, \mathbf{R}_d)$ that can be evaluated at a given point in $\mathcal{O}(F + dnr^3)$ FLOP (equivalently to the first-order case; see the proof of statement 5.2 for details).

Steps 6–11 use classic AD to compute the gradient of the function g with respect to its arguments and then project the resulting gradients onto the gauge conditions. These steps are equivalent to steps 5–10 of Algorithm 5.2 and can be done in $\mathcal{O}(F + dnr^3)$ (again, see proof of statement 5.2 for details).

Algorithm 6.2 Computing the approximate Riemannian Hessian-by-vector product for low-rank tensors via AD.

```
Require: ${\mathbf{G}_k}_{k=1}^d$—TT-cores of $\mathbf{X}$, the delta terms $\dot{\mathbf{S}}_1^{\mathbf{Z}}, \dots, \dot{\mathbf{S}}_d^{\mathbf{Z}}$ that define the projection (onto the tangent space) of the tensor $\mathbf{Z}$ which has to be multiplied by the approximate Riemannian Hessian, $p(\widehat{\mathbf{G}}_1, \dots, \widehat{\mathbf{G}}_d)$—Python implementation of $f(\widehat{\mathbf{X}})$ for a point $\widehat{\mathbf{X}}$ given by TT-cores $\widehat{\mathbf{G}}_1, \dots, \widehat{\mathbf{G}}_d$.
Ensure: The TT-cores $\{\mathbf{H}_k\}_{k=1}^d$ of the approximate Riemannian Hessian-by-vector product (6.1)

1: For $\mathbf{X}$, compute left- and right-orthogonal TT-cores $\{\mathbf{U}_k\}_{k=1}^{d-1}$, $\{\mathbf{V}_k\}_{k=2}^d$, respectively, and $\{\mathbf{S}_k\}_{k=1}^d$ as in (5.4).
2: function g$(\mathbf{R}_1, \dots, \mathbf{R}_d)$
3:     Run Algorithm 5.1 passing as input $\{\mathbf{G}_k\}_{k=1}^d$ and $\{\mathbf{R}_k\}_{k=1}^d$ and write the output TT-cores into $\{\widehat{\mathbf{G}}_k\}_{k=1}^d$
4:     return $p(\widehat{\mathbf{G}}_1, \dots, \widehat{\mathbf{G}}_d)$
5: function w$(\widehat{\mathbf{R}}_1, \dots, \widehat{\mathbf{R}}_d)$
6:     Using AD compute $\dot{\mathbf{S}}_k := \frac{\partial g}{\partial \mathbf{R}_k}\Big|_{(\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_d) = (\widehat{\mathbf{R}}_1, \dots, \widehat{\mathbf{R}}_d)}$ for $k = 1, \dots, d$
7:     for $k \leftarrow 1$ to $d-1$ do
8:         $\mathbf{D}_k := \texttt{reshape}(\dot{\mathbf{S}}_k, (r_{k-1}n_k, r_k))$
9:         $\mathbf{U}_k^L := \texttt{reshape}(\mathbf{U}_k, (r_{k-1}n_k, r_k))$
10:        $\mathbf{D}_k := \mathbf{D}_k + \mathbf{U}_k^L \left( \left(\mathbf{U}_k^L\right)^\intercal \mathbf{D}_k \right)$        ▷ See (5.11)
11:        $\dot{\mathbf{S}}_k := \texttt{reshape}(\mathbf{D}_k, (r_{k-1}, n_k, r_k))$
12:    return $\sum_{k=1}^d \left\langle \dot{\mathbf{S}}_k, \dot{\mathbf{S}}_k^{\mathbf{Z}} \right\rangle$
13: Using AD compute $\dot{\mathbf{S}}_k := \frac{\partial w}{\partial \mathbf{R}_k}\Big|_{(\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_d) = (\mathbf{S}_1, \mathbf{O}_2, \dots, \mathbf{O}_d)}$ for $k = 1, \dots, d$
14: for $k \leftarrow 1$ to $d-1$ do
15:     $\mathbf{D}_k := \texttt{reshape}(\dot{\mathbf{S}}_k, (r_{k-1}n_k, r_k))$
16:     $\mathbf{U}_k^L := \texttt{reshape}(\mathbf{U}_k, (r_{k-1}n_k, r_k))$
17:     $\mathbf{D}_k := \mathbf{D}_k + \mathbf{U}_k^L \left( \left(\mathbf{U}_k^L\right)^\intercal \mathbf{D}_k \right)$        ▷ See (5.11)
18:     $\dot{\mathbf{S}}_k := \texttt{reshape}(\mathbf{D}_k, (r_{k-1}, n_k, r_k))$
19: Run Algorithm 5.1 passing as input $\{\mathbf{G}_k\}_{k=1}^d$ and $\{\dot{\mathbf{S}}_k\}_{k=1}^d$ and write the output TT-cores into $\{\mathbf{H}_k\}_{k=1}^d$
```

Step 12 computes the dot product between two elements of the same tangent space, which (as noted above) can be computed with the complexity that equals to the number of elements in the delta-terms, i.e., $\mathcal{O}(ndr^2)$. So the total complexity of evaluating the function $w(\widehat{\mathbf{R}}_1, \dots, \widehat{\mathbf{R}}_d)$ at a point is $\mathcal{O}(F + ndr^3)$ FLOP.

Step 13 uses classic AD to compute the gradient of $w(\widehat{\mathbf{R}}_1, \dots, \widehat{\mathbf{R}}_d)$ with respect to its arguments, which can be done in $\mathcal{O}(F + dnr^3)$ FLOP.

Steps 14–19 are equivalent to steps 6–11 of Algorithm 5.2 and (as discussed in the proof of Statement 5.2) take at most $\mathcal{O}(dnr^3)$ FLOP.

Combining the complexity from all the steps yields $\mathcal{O}(F + dnr^3)$. □

Similarly to the matrix case, for most practical functions $f(\mathbf{X})$, the total complexity of the algorithm is $\mathcal{O}(F + dnr^3) = \mathcal{O}(F)$.

Table 1

Ranks of tensors, matrices, and vectors involved in different tiers of experiments. See section 7.1 *for more details.*

Function	Small		Medium		Large	
	Tensor $\mathbf{X}$	Operator A	Tensor $\mathbf{X}$	Operator A	Tensor $\mathbf{X}$	Operator A
$\langle \mathrm{A}\mathbf{X}, \mathbf{X} \rangle$	10	10	20	20	20	20
$\langle \mathrm{A}^\intercal \mathrm{A}\mathbf{X}, \mathbf{X} \rangle$	10	5	20	10	20	20
RayleighQuotient	10	10	20	20	20	20
completion	5	-	10	-	20	-
ExpMachines	5	-	10	-	20	-

7. Numerical experiments.

In this section, we compare three ways of computing Riemannian gradients and approximate Riemannian Hessian-by-vector products: "naive"—by deriving the expression for the TT-format of the Euclidean gradient and then projecting the Euclidean gradient onto the tangent space[9]; "improved"—similar to the "naive" approach, but with additional tricks to speed up the computations using optimized primitives[10] implemented in [30]; "AD"—by using the proposed AD method. All methods give the same answer (as verified by tests for all the functions described below), so we only consider speed and memory usage when comparing the methods.

We ran the experiments on a machine with 240 Gb of RAM and an NVIDIA V100 GPU which has 16 Gb of video memory available. For each problem, we tried to choose a realistic problem size (specified separately for each particular function below) and ran all the experiments with three different tiers of TT-ranks: Small, Medium, and Large. We choose the Large ranks for each problem to be the largest TT-rank that fits RAM of the machine we ran the experiments on (240 Gb), Medium to be the largest TT-rank that fits the GPU memory (16 Gb), and Small TT-ranks to be twice smaller than the Medium TT-ranks. See Table 1 for the TT-ranks for each function.

7.1. Functions.

Below we talk in detail about the five functions considered in numerical experiments.

Quadratic form. The first function we consider is quadratic form $f(\mathbf{X}) = < \mathrm{A}\mathbf{X}, \mathbf{X} >$ with symmetric A, which is relevant for solving systems of linear equations. Its Euclidean gradient equals $2\mathrm{A}\mathbf{X}$, and the product of its Hessian by a given vector $\mathbf{Z}$ equals $2\mathrm{A}\mathbf{Z}$. The naive method for computing the projection of the matrix-by-vector product (e.g., the product of the approximate Riemannian Hessian by a given vector $\mathrm{P}_{\mathbf{X}} 2\mathrm{A}\mathbf{Z}$) consists of first computing the matrix-by-vector product $\mathrm{A}\mathbf{Z}$ and then projecting the result. The combined complexity of the naive approach is $\mathcal{O}(dnr_x r_z^2 R^2)$, where the TT-rank of the tensor $\mathbf{X}$ is $\mathbf{r}_x = (r_x, r_x, \dots, r_x)$, TT-rank of tensor $\mathbf{Z}$ is $\mathbf{r}_z = (r_z, r_z, \dots, r_z)$, and TT-rank of the operator A is $\mathbf{R} = (R, R, \dots, R)$. An improved version of this operation combines the matrix-by-vector multiplication and the projection onto the tangent space into a single step $\mathrm{P}_{\mathbf{X}}\mathrm{A}\mathbf{Z}$ and exploits the

[9] Note that in this process we never materialize the dense representation of any tensor and always work with TT-representations.

[10] Examples of additional tricks used: implementing projection of matrix-by-vector multiplication $P_{\mathbf{x}}\mathrm{A}\mathbf{b}$ as a single operation, instead of a doing them one-by-one, allows us to speed things up; using the fact that projection is a linear operation and thus $P_{\mathbf{X}} \sum_i \mathbf{A}_i = \sum_i P_{\mathbf{X}} \mathbf{A}_i$.

structure of arising operations to decrease complexity to $\mathcal{O}(dn^2 r_x r_z R^2)$ (for details of implementation of this operation see section 4.1 of [31]).

In the experiments below, we consider a 40-dimensional tensor $\mathbf{X} \in \mathbb{R}^{20\times\dots\times 20}$ and represent the operator A by a TT-matrix of size $20^{40} \times 20^{40}$. We use TT-ranks $r_{\mathrm{A}} = 10$, $r_{\mathbf{X}} = 10$, $r_{\mathbf{Z}} = 20$ for the Small TT-rank experiment and $r_{\mathrm{A}} = 20$, $r_{\mathbf{X}} = 20$, $r_{\mathbf{Z}} = 40$ for the Medium, and Large TT-rank experiments.

Quadratic form with a Gram matrix. The second function is quadratic form $f(\mathbf{X}) =< \mathrm{A}^\intercal \mathrm{A}\mathbf{X}, \mathbf{X} >$ (operator factored into the product of two TT-matrices $\mathrm{A}^\intercal \mathrm{A}$ arised, e.g., in [32]). The Euclidean gradient equals to $2\mathrm{A}^\intercal \mathrm{A}\mathbf{X}$ and the product of its Hessian by a given vector $\mathbf{Z}$ equals to $2\mathrm{A}^\intercal \mathrm{A}\mathbf{Z}$. We use the same trick to optimize the projection of the product of two matrices by a vector as in the quadratic form case. Note that it takes significant effort to derive and implement the improved version here.

In the experiments below, we consider a 10-dimensional tensor $\mathbf{X} \in \mathbb{R}^{20\times\dots\times 20}$ and represent the operator A by a TT-matrix of size $20^{10} \times 20^{10}$. We use TT-ranks $r_{\mathrm{A}} = 10$, $r_{\mathbf{X}} = 5$, $r_{\mathbf{Z}} = 10$ for the Small TT-rank experiment, $r_{\mathrm{A}} = 20$, $r_{\mathbf{X}} = 10$, $r_{\mathbf{Z}} = 20$ for the Medium TT-rank experiments, and $r_{\mathrm{A}} = 20$, $r_{\mathbf{X}} = 20$, $r_{\mathbf{Z}} = 40$ for the Large TT-rank experiments.

Rayleigh quotient. The Rayleigh quotient $f(\mathbf{X}) = < \mathrm{A}[\mathbf{X}], \mathbf{X} > / < \mathbf{X}, \mathbf{X} >$ with symmetric A is relevant for solving eigenvalue problems. The Euclidean gradient is $\frac{2}{\langle \mathbf{X},\mathbf{X}\rangle}(\mathrm{A}[\mathbf{X}] - f(\mathbf{X})\mathbf{X})$, and the product of its Hessian by a given vector $\mathbf{Z}$ is

$$\begin{aligned}\nabla^2 f(\mathbf{X})\ \mathbf{Z} =& \frac{2}{\langle \mathbf{X},\mathbf{X}\rangle}\mathrm{A}\mathbf{Z} - 2\frac{f(\mathbf{X})}{\langle \mathbf{X},\mathbf{X}\rangle}\mathbf{Z} - 4\frac{\langle \mathrm{A}\mathbf{X},\mathbf{Z}\rangle}{\langle \mathbf{X},\mathbf{X}\rangle^2}\mathbf{X}\\ &- 4\frac{\langle \mathbf{X},\mathbf{Z}\rangle}{\langle \mathbf{X},\mathbf{X}\rangle^2}\mathrm{A}\mathbf{X} + 8f(\mathbf{X})\frac{\langle \mathbf{X},\mathbf{Z}\rangle}{\langle \mathbf{X},\mathbf{X}\rangle^2}\mathbf{X}.\end{aligned}$$

The improved version of the Riemannian gradient and approximate-Riemannian-Hessian-by-vector product is computed by representing the projection of a sum of terms as a sum of projections and using the optimized projection of matrix-by-vector multiplication where appropriate, e.g., for the Riemannian gradient, we get

$$\mathrm{P}_{\mathbf{X}}\,\nabla f = \frac{2}{\langle \mathbf{X},\mathbf{X}\rangle}\mathrm{P}_{\mathbf{X}}\,\mathrm{A}\mathbf{X} - \frac{2f(\mathbf{X})}{\langle \mathbf{X},\mathbf{X}\rangle}\mathbf{X},$$

where we use the fact that $\mathrm{P}_{\mathbf{X}}\,\mathbf{X} = \mathbf{X}$.

In the experiments below we consider a 40-dimensional tensor $\mathbf{X} \in \mathbb{R}^{20\times\dots\times 20}$ and represent the operator A by a TT-matrix of size $20^{40} \times 20^{40}$. We use TT-ranks $r_{\mathrm{A}} = 10$, $r_{\mathbf{X}} = 10$, $r_{\mathbf{Z}} = 20$ for the Small TT-rank experiment and $r_{\mathrm{A}} = 20$, $r_{\mathbf{X}} = 20$, $r_{\mathbf{Z}} = 40$ for the Medium, and Large TT-rank experiments.

Completion problem. The following function is used when solving low-rank matrix and tensor completion problems: $f(\mathbf{X}) = \|\mathrm{P}_\Omega(\mathbf{X}-\mathbf{A})\|^2$, where P_Ω denotes projection on the index set Ω such that

$$\mathrm{P}_\Omega \mathbf{X} = \begin{cases} X_{i_1\cdots_d} & (i_1,\dots,i_d) \in \Omega,\\ 0 & \text{otherwise.}\end{cases}$$

Its Euclidean gradient is $2\mathrm{P}_\Omega(\mathbf{X} - \mathbf{A})$, and the product of its Euclidean Hessian by a given vector $\mathbf{Z}$ is $\mathrm{P}_\Omega \mathbf{Z}$. For the improved implementation, we represent the projection of a tensor on the index set as the sum of its nonzero entries $\mathrm{P}_\Omega \mathbf{X} = \sum_{(i_1,\dots,i_d)\in\Omega} X_{i_1\dots i_d} \mathbf{E}^{i_1\dots i_d}$, where by $\mathbf{E}^{i_1\dots i_d}$ we denote the tensor with value 1 in the position $(i_1,\dots,i_d)$ and zero everywhere else. Tensor $\mathbf{E}^{i_1\dots i_d}$ has TT-rank 1. Then, we use the fact that the projection of a sum of terms is the sum of projections and, thus, instead of projecting the tensor $\mathrm{P}_\Omega \mathbf{X}$ which has high TT-rank, we project TT-rank-1 tensors $\mathbf{E}$, which leads to a significant speed-up.

In the experiments below we consider a 10-dimensional tensor $\mathbf{X} \in \mathbb{R}^{20\times\dots\times 20}$ and the index set Ω consisting of $10dnr_{\mathrm{X}}^2$ elements (i.e., the TT-rank of $\mathrm{P}_\Omega \mathbf{X}$ equals to $10dnr_{\mathrm{X}}^2$ and does not fit to memory for the naive implementation even in the Small TT-rank case). The values of the target tensor $\mathbf{A}$ at the randomly chosen $10dnr_{\mathrm{X}}^2$ elements are sampled from the standard normal distribution. For the Small TT-rank experiment, we use $r_{\mathrm{X}} = 5$, $r_{\mathbf{Z}} = 10$, and $10dnr_{\mathrm{X}}^2 = 50{,}000$ observed elements; for the Medium TT-rank experiment, we use $r_{\mathrm{X}} = 10$, $r_{\mathbf{Z}} = 20$, and $10dnr_{\mathrm{X}}^2 = 200{,}000$ observed elements; for the Large TT-rank experiment, we use $r_{\mathrm{X}} = 20$, $r_{\mathbf{Z}} = 40$, and $10dnr_{\mathrm{X}}^2 = 800{,}000$ observed elements;.

Exponential machines. For a machine learning related function, we used the empirical risk of the exponential machines model (see [33] for details and justification):

$$f(\mathbf{X}) = \sum_{i=1}^{N} h(\langle \mathbf{X}, \mathbf{W}^{(i)} \rangle, y^{(i)}),$$

where $h(x, y)$ is the loss function $h(x, y) = \log(1 + \exp(-yx))$,[11] tensors $\mathbf{W}^{(i)}$ have TT-rank 1, and $y^{(i)}$ are binary numbers.

As argued in [33], this model corresponds to a type of recurrent neural network, so we refer to this example as a neural network loss in the rest of the paper.

The gradient of this function is

$$\nabla f = -\sum_{i=1}^{N} \frac{\exp(-y^{(i)}\langle \mathbf{X}, \mathbf{W}^{(i)} \rangle)}{1 + \exp(-y^{(i)}\langle \mathbf{X}, \mathbf{W}^{(i)} \rangle)} \mathbf{W}^{(i)}$$

and the product of the Hessian of this function by a given vector is

$$\nabla^2 f(\mathbf{X})\mathbf{Z} = \sum_{i=1}^{N} \frac{\exp(-y^{(i)}\langle \mathbf{X}, \mathbf{W}^{(i)} \rangle)}{(1 + \exp(-y^{(i)}\langle \mathbf{X}, \mathbf{W}^{(i)} \rangle))^2} \langle \mathbf{Z}, \mathbf{W}^{(i)} \rangle \mathbf{W}^{(i)}.$$

Again, by using linearity of the projection, to implement the improved version we can independently compute the cheap projections $\mathrm{P}_{\mathbf{X}} \mathbf{W}^{(i)}$ and sum them up.

In the experiments below we consider a 10-dimensional tensor $\mathbf{X} \in \mathbb{R}^{500\times\dots\times 500}$ (which corresponds to a machine learning problem with 10 categorical features, each of which can take 500 different values) and number of objects (in the minibatch) $N = 32$. We use TT-ranks $r_{\mathrm{X}} = 5$, $r_{\mathbf{Z}} = 10$ for the Small TT-rank experiment, $r_{\mathrm{X}} = 10$, $r_{\mathbf{Z}} = 20$ for the Medium TT-rank experiments, and $r_{\mathrm{X}} = 20$, $r_{\mathbf{Z}} = 40$ for the Large TT-rank experiments.

[11] This loss adapted from [33] is equivalent to the cross-entropy loss when the label y takes values from $\{-1, 1\}$ instead of the more common $\{0, 1\}$.

7.2. Results. We used the T3F library [30] for implementing all three algorithms for the five functions described above. The T3F library provides the primitives used above such as the optimized projection of a matrix-by-vector product and also supports GPU execution (thanks to the underlying use of TensorFlow library [7]). We implemented the Riemannian AD functionality as a part of the T3F library as well.

Results of the main numerical experiments are presented in Tables 2 and 3, plus additional results on Small and Large ranks are presented in the appendix (Tables 4–8). The proposed AD method outperformed both the naive and the improved implementations for computing the Riemannian gradient on CPU both in terms of runtime and memory usage (Tables 4(a), 2(a), and 5(a)).

Note that sometimes the improved implementation runs slower than the naive implementation. After profiling our implementation of the methods we believe that this happens because the improved implementation operates with tensors of larger dimensionality (e.g., when the naive version operates with a tensor of size $1024 \times 1024 \times 1024$, the improved implementation may operate with the same tensor, but reshaped to $32 \times 32 \times 32 \times 32 \times 32 \times 32$ for better flexibility), which causes an additional overhead when permuting dimensions. This overhead is typically neglectable when executing on GPU, because GPUs have the required flops to compute all the necessary permute operations in parallel.

In some cases, the proposed AD method is outperformed in terms of the runtime by the improved implementation. This happens due to the (constant) overhead that arises when performing AD. For example, when computing the approximate Riemann-

Table 2

Comparison of three methods for Medium TT-rank setting (see Table 1) for computing the Riemannian gradient of various functions in terms of execution time and memory used on CPU and GPU. A dash means that the respective method ran out of memory.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X} \rangle$	2.4	6.4	3.3	2.8	**1**	**0.62**
$\langle \mathrm{A}^\intercal \mathrm{A}\mathbf{X}, \mathbf{X} \rangle$	2	10	-	-	**0.54**	**0.3**
RayleighQuotient	3.1	6.9	3.4	2.8	**1.1**	**0.62**
completion	-	-	3.4	13	**0.98**	**6.2**
ExpMachines	0.18	0.082	0.12	0.042	**0.078**	**0.03**

(a) Comparison of computing Riemannian gradient by three methods on CPU for Medium TT-ranks.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X} \rangle$	0.17	11	**0.057**	**0.56**	0.085	0.62
$\langle \mathrm{A}^\intercal \mathrm{A}\mathbf{X}, \mathbf{X} \rangle$	-	-	-	-	**0.032**	**0.28**
RayleighQuotient	0.22	12	**0.07**	**0.57**	0.1	0.63
completion	-	-	-	-	**0.25**	**5.6**
ExpMachines	0.034	0.11	**0.027**	0.04	**0.027**	**0.017**

(b) Comparison of computing Riemannian gradient by three methods on GPU for Medium TT-ranks.

TABLE 3

Comparison of three methods for Medium TT-rank setting (see Table 1) for computing the approximate Riemannian Hessian-by-vector product of various functions in terms of execution time and memory used on CPU and GPU. A dash means that the respective method ran out of memory.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	6.9	28	3.6	3.2	**2.2**	**1.1**
$\langle \mathrm{A}^\intercal \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	5	36	-	-	**1.1**	**0.49**
RayleighQuotient	18	56	4.9	4	**2.4**	**1.1**
completion	-	-	3.5	22	**2.6**	**12**
ExpMachines	0.21	0.075	**0.12**	**0.053**	0.13	0.079

(a) Comparison of computing the approximate Riemannian Hessian-by-vector product by three methods on CPU for Medium TT-ranks.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	-	-	**0.067**	**0.66**	0.16	1
$\langle \mathrm{A}^\intercal \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	-	-	-	-	**0.06**	**0.54**
RayleighQuotient	-	-	**0.14**	**0.73**	0.19	1.1
completion	-	-	-	-	**0.64**	**11**
ExpMachines	0.036	0.11	**0.028**	**0.04**	0.03	0.043

(b) Comparison of computing the approximate Riemannian Hessian-by-vector product by three methods on GPU for Medium TT-ranks.

ian Hessian-by-vector product of the quadratic form, the improved implementation can directly compute the desired quantity $\mathrm{P}_{\mathbf{X}}\mathrm{A}\mathbf{Z}$, while the AD is forced to first compute the function $< \mathrm{A}\mathbf{X}, \mathbf{X} >$, then perform the classic AD twice, each time doubling the computational graph, making the computational graph four times larger than the original one.

However, we believe that despite some overhead compared to the improved implementation which appears in individual cases, the proposed method is still valuable since it significantly simplifies the implementation of Riemannian optimization algorithms while getting reasonable (and in many cases superior) performance.

8. Conclusion.

In this paper, we propose a way of exactly computing the Riemannian gradient and the approximate Riemannian Hessian-by-vector product of a function for low-rank matrices and tensors in time proportional to the time it takes to compute the value of the function at one point. In experiments, the proposed approach in many cases shows superior performance compared to both considered baselines in terms of memory and time, while being significantly easier to use. The code of the proposed algorithms is published online in the open-source library T3F.

Appendix A. Additional experimental results.

Here we provide additional experimental results. If in the main text, only the Medium TT-rank experiments were provided, here we also provide results on input tensors of Small and Large TT-ranks (see section 7.1 for a detailed explanation of the setup and of the TT-ranks chosen for all experiments).

Table 4

Comparison of three methods for Small TT-rank setting (see Table 1) for computing the Riemannian gradient of various functions in terms of execution time and memory used on CPU and GPU. A dash means that the respective method ran out of memory.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	0.32	0.42	0.43	0.26	**0.2**	**0.1**
$\langle \mathrm{A}^\intercal\mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	0.098	0.15	-	-	**0.085**	**0.033**
RayleighQuotient	0.52	0.48	0.49	0.22	**0.23**	**0.11**
completion	-	-	0.46	0.88	**0.074**	**0.41**
ExpMachines	0.14	0.061	0.034	0.013	**0.026**	**0.0097**

(a) Comparison of computing Riemannian gradient by three methods on CPU for Small TT-ranks.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	**0.029**	0.69	**0.029**	0.14	0.036	**0.1**
$\langle \mathrm{A}^\intercal\mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	**0.0083**	0.23	-	-	0.01	**0.031**
RayleighQuotient	**0.038**	0.76	0.039	0.14	0.042	**0.11**
completion	-	-	0.092	1.2	**0.062**	**0.37**
ExpMachines	0.031	0.1	**0.011**	0.013	**0.011**	**0.0048**

(b) Comparison of computing Riemannian gradient by three methods on GPU for Small TT-ranks.

Table 5

Comparison of three methods for Large TT-rank setting (see Table 1) for computing the Riemannian gradient of various functions in terms of execution time and memory used on CPU and GPU. A dash means that the respective method ran out of memory.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	2.4	6.4	3.3	2.8	**1**	**0.62**
$\langle \mathrm{A}^\intercal\mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	17	162	-	-	**1.9**	**1.1**
RayleighQuotient	3.1	6.9	3.4	2.8	**1.1**	**0.62**
completion	-	-	-	-	**13**	**98**
ExpMachines	0.35	0.12	1	0.15	**0.33**	**0.11**

(a) Comparison of computing Riemannian gradient by three methods on CPU for Large TT-ranks.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	0.17	11	**0.057**	**0.56**	0.085	0.62
$\langle \mathrm{A}^\intercal\mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	-	-	-	-	**0.15**	**0.94**
RayleighQuotient	0.22	12	**0.07**	**0.57**	0.1	0.63
completion	-	-	-	-	-	-
ExpMachines	**0.15**	0.12	**0.15**	0.14	**0.15**	**0.067**

(b) Comparison of computing Riemannian gradient by three methods on GPU for Large TT-ranks.

TABLE 7

Comparison of computing the approximate Riemannian Hessian-by-vector product by three methods on GPU for Small TT-ranks.

(a) Comparison of three methods for Small TT-rank setting (see Table 1) for computing the approximate Riemannian Hessian-by-vector product of various functions in terms of execution time and memory used on CPU and GPU. A dash means that the respective method ran out of memory.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	0.71	1.7	**0.43**	0.33	0.48	**0.17**
$\langle \mathrm{A}^\intercal \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	0.32	0.56	-	-	**0.14**	**0.058**
RayleighQuotient	1.8	4.2	0.8	0.55	**0.52**	**0.17**
completion	-	-	0.47	0.91	**0.19**	**0.9**
ExpMachines	0.22	0.056	**0.047**	0.032	0.057	**0.03**

(b) Comparison of computing the approximate Riemannian Hessian-by-vector product by three methods on CPU for Small TT-ranks.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	0.053	2.7	**0.035**	0.21	0.066	**0.16**
$\langle \mathrm{A}^\intercal \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	**0.018**	0.67	-	-	0.019	**0.058**
RayleighQuotient	0.16	7	0.093	0.58	**0.079**	**0.2**
completion	-	-	**0.1**	1.8	0.15	**0.72**
ExpMachines	0.031	0.1	**0.011**	0.013	0.014	**0.012**

TABLE 8

Comparison of three methods for Large TT-rank setting (see Table 1) for computing the approximate Riemannian Hessian-by-vector product of various functions in terms of execution time and memory used on CPU and GPU. A dash means that the respective method ran out of memory.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	6.9	28	3.6	3.2	**2.2**	**1.1**
$\langle \mathrm{A}^\intercal \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	-	-	-	-	**4.2**	**2**
RayleighQuotient	18	56	4.9	4	**2.4**	**1.1**
completion	-	-	-	-	-	-
ExpMachines	**0.38**	**0.11**	1	0.14	0.52	0.23

(a) Comparison of computing the approximate Riemannian Hessian-by-vector product by three methods on CPU for Large TT-ranks.

Function	Naive		Improved		AD	
	(s)	(Gb)	(s)	(Gb)	(s)	(Gb)
$\langle \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	-	-	**0.067**	**0.66**	0.16	1
$\langle \mathrm{A}^\intercal \mathrm{A}\mathbf{X}, \mathbf{X}\rangle$	-	-	-	-	**0.31**	**2**
RayleighQuotient	-	-	**0.14**	**0.73**	0.19	1.1
completion	-	-	-	-	-	-
ExpMachines	0.16	**0.12**	**0.15**	0.14	**0.15**	0.17

(b) Comparison of computing the approximate Riemannian Hessian-by-vector product by three methods on GPU for Large TT-ranks.

REFERENCES

[1] S. Holtz, T. Rohwedder, and R. Schneider, *On manifolds of tensors of fixed TT–rank*, Numer. Math., 120 (2012), pp. 701–731.

[2] P.-A. Absil, R. Mahony, and R. Sepulchre, *Optimization Algorithms on Matrix Manifolds*, Princeton University Press, Princeton, NJ, 2008.

[3] P-A Absil, R. Mahony, and J. Trumpf, *An extrinsic look at the Riemannian Hessian*, in Proceedings of the International Conference on Geometric Science of Information, Springer, New York, 2013, pp. 361–368.

[4] D. Kressner, M. Steinlechner, and B. Vandereycken, Preconditioned low-rank Riemannian optimization for linear systems with tensor product structure. SIAM J. Sci. Comput., 38 (2016), pp. A2018–A2044.

[5] B. Vandereycken and S. Vandewalle, *A Riemannian optimization approach for computing low-rank solutions of Lyapunov equations*, SIAM J. Matrix Anal. Appl., 31 (2010), pp. 2553–2579.

[6] M. Rakhuba and I. Oseledets, *Jacobi–Davidson method on low-rank matrix manifolds*, SIAM J. Sci. Comput., 40 (2018), pp. A1149–A1170.

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2015; software available from https://tensorflow.org.

[8] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, *Tensorly: Tensor Learning in Python*, preprint, arXiv:1610.09555, 2016.

[9] I. V. Oseledets, S. Dolgov, V. Kazeev, D. Savostyanov, O. Lebedeva, P. Zhlobich, T. Mach, and L. Song, *TT-Toolbox*, 2011; available online from https://github.com/oseledets/TT-Toolbox.

[10] L. Ma, J. Ye, and E. Solomonik, *Autohoot: Automatic high-order optimization for tensors*, in Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques, 2020, pp. 125–137.

[11] D. Suess and M. Holzäpfel, *mpnum: A matrix product representation library for python*, J. Open Source Software, 2 (2017), p. 465.

[12] J. Townsend, N. Koep, and S. Weichwald, *Pymanopt: A python toolbox for optimization on manifolds using automatic differentiation*, The Journal of Machine Learning Research, 17 (2016), pp. 4755–4759.

[13] H. Sommer, C. Pradalier, and P. Furgale, *Automatic differentiation on differentiable manifolds as a tool for robotics*, in Robotics Research, Springer, New York, 2016, pp. 505–520.

[14] L. Koppel and S. L Waslander, *Manifold Geometry with Fast Automatic Derivatives and Coordinate Frame Semantics Checking in C++*, preprint, arXiv:1805.01810, 2018.

[15] M. Psenka and N. Boumal, *Second-Order Optimization for Tensors with Fixed Tensor-Train Rank*, preprint, arXiv:2011.13395, 2020.

[16] W. Gander, M. J Gander, and F. Kwok, *Scientific Computing—An Introduction Using Maple and MATLAB*, Texts in Comput. Sci. Eng. 11, Springer, New York, 2014.

[17] C. C. Margossian, *A review of automatic differentiation and its efficient implementation*, in Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 9 (2019), p. e1305.

[18] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, Philadelphia, 2008.

[19] P. A. Absil and I. V. Oseledets, *Low-rank retractions: A survey and new results*, Comput. Optim. Appl., 62 (2015), pp. 5–29.

[20] M. V. Rakhuba and I. V. Oseledets, *Jacobi–Davidson method on low-rank matrix manifolds*, SIAM J. Sci. Comput., 40 (2018), pp. A1149–A1170.

[21] J. M Lee, *Introduction to Smooth Manifolds*, Grad. Texts in Math. 218, Springer, New York, 2003.

[22] B. Vandereycken, *Low-rank matrix completion by Riemannian optimization*, SIAM J. Optim., 23 (2013), pp. 1214–1236.

[23] I. V. Oseledets, *Tensor-train decomposition*, SIAM J. Sci. Comput., 33 (2011), pp. 2295–2317.

[24] M. Steinlechner, Riemannian Optimization for Solving High-dimensional Problems with Low-rank Tensor Structure, Create Space, Scotts Valley, CA, 2016.

[25] T. E. Oliphant, Guide to NumPy, Trelgol Publishing, USA, 2006.

[26] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, *Automatic differentiation in machine learning: A survey*, J. Mach. Learn. Res., 18 (2017), pp. 5596–5637.

[27] W. Hackbusch and B. N. Khoromskij, *Low-rank Kronecker-product approximation to multi-dimensional nonlocal operators. I. Separable approximation of multi-variate functions*, Computing, 76 (2006), pp. 177–202.

[28] B. N. Khoromskij, *Tensor-structured preconditioners and approximate inverse of elliptic operators in $\mathbb{R}^d$*, Constr. Approx., 30 (2009), pp. 599–620.

[29] B. A. Pearlmutter, *Fast exact multiplication by the Hessian*, Neural Comput., 6 (1994), pp. 147–160.

[30] A. Novikov, P. Izmailov, V. Khrulkov, M. Figurnov, and I. Oseledets, *Tensor train decomposition on tensorflow (t3f)*, J. Mach. Learn. Res., 21 (2020).

[31] M. Rakhuba, A. Novikov, and I. Oseledets, *Low-rank Riemannian eigensolver for high-dimensional Hamiltonians*, J. Comput. Phys., 396 (2019), pp. 718–737.

[32] M. Bachmayr and V. Kazeev, *Stability of low-rank tensor representations and structured multilevel preconditioning for elliptic PDEs*, Found. Comput. Math., 20 (2020), pp. 1–62.

[33] A. Novikov, M. Trofimov, and I. Oseledets, *Exponential machines*, Bull. Pol. Acad. Sci. Tec. Sciences, 6, 2018.