A Unified Framework for Matrix Backpropagation

Gatien Darley and Stéphane Bonnet

***Abstract*—Computing matrix gradient has become a key aspect in modern signal processing/machine learning, with the recent use of matrix neural networks requiring matrix backpropagation. In this field, two main methods exist to calculate the gradient of matrix functions for symmetric positive definite (SPD) matrices, namely, the Daleckiĭ–Kreĭn/Bhatia formula and the Ionescu method. However, there appear to be a few errors. This brief aims to demonstrate each of these formulas in a self-contained and unified framework, to prove theoretically their equivalence, and to clarify inaccurate results of the literature. A numerical comparison of both methods is also provided in terms of computational speed and numerical stability to show the superiority of the Daleckiĭ–Kreĭn/Bhatia approach. We also extend the matrix gradient to the general case of diagonalizable matrices. Convincing results with the two backpropagation methods are shown on the EEG-based BCI competition dataset with the implementation of an SPDNet, yielding around 80% accuracy for one subject. Daleckiĭ–Kreĭn/Bhatia formula achieves an 8% time gain during training and handles degenerate cases.**

***Index Terms*—Backpropagation, matrix derivative, neural network learning, symmetric positive definite (SPD) matrix.**

I. Introduction

A. Contextual Motivation

Matrix analysis has proven itself to be an efficient way for processing data. Indeed, using matrix format avoids vectorizing them [1] and helps efficiently summarize data. This has led to an interest in matrix neural networks, which therefore require matrix backpropagation, which is more efficient than using the α-derivative on vectors [2]. Moreover, symmetric positive definite (SPD) matrices are ubiquitous, and working with them has also become popular in various fields of applications, especially with the growing use of Riemannian geometry in algorithms. For instance, in electroencephalography (EEG) analysis [3], [4], [5], [6], electromyography (EMG) analysis [7], image [8], [9], [10] or video processing [11], [12]. Indeed, covariance matrices—belonging to the Riemannian manifold of SPD matrices—are more and more used as data descriptors in these fields because they efficiently capture relationships between features that might be correlated. Naturally, geometric neural networks handling such structured data have emerged and are undergoing expansion. Particularly, the SPDNet [11] now achieves state-of-the-art performances by combining Riemannian geometry and deep learning. In this context, the need for efficient gradient calculation is crucial.

Ionescu et al. [8] is set to be the precursor in this field of study by developing gradient calculations for matrix backpropagation. Yet the formula proposed in his papers [8], [13] presents some mistakes that we are going to discuss. Later, Brooks et al. [14] brought a result from matrix operator theory—that is, Daleckiĭ–Kreĭn/Bhatia formula dating back from 1956—to calculate the gradient of SPD matrix functions. It is now the main way to implement backpropagation for SPD layers, as it appears in [11] or in [15]. However, the link between the two approaches has not been studied. Engin et al. [16] tried to prove it but relied on the inaccurate formulas of C. Ionescu, yielding an incorrect proof.

Received 31 January 2025; revised 7 July 2025; accepted 2 September 2025. *(Corresponding author: Stéphane Bonnet.)*

The authors are with the CEA, Leti, Université Grenoble Alpes, 38000 Grenoble, France (e-mail: gatien.darley@cea.fr; stephane.bonnet@cea.fr).

Digital Object Identifier 10.1109/TNNLS.2025.3607405

In this field, results are rather scattered in the literature, and our self-contained paper aims at filling this gap by providing several proofs and by numerically comparing both approaches. We also believe and argue that Daleckiĭ–Kreĭn/Bhatia formula is a key tool in manipulating matrix functions. Currently, commonly used libraries in deep learning, such as PyTorch, do not support automatic differentiation on the SPD manifold. Some specific libraries yet exist for calculations on manifolds; however, they are not tailored for deep learning and backpropagation. One can think of Pymanopt (https://github.com/pymanopt/pymanopt), Geomstats (https://geomstats.github.io/), or Geoopt (https://github.com/geoopt/geoopt), but none of them contains the layers' mathematical description required to implement a neural network. The same applies to McTorch (https://github.com/mctorch/mctorch) or TheanoGeometry (https://github.com/SomeoneSerge/theanogeometry), which may also lack engineering maintenance. So, shedding light on Ionescu and Daleckiĭ–Kreĭn/Bhatia formulas might also help to produce a functional tool for SPD deep learning since it allows implementing gradients of neural networks SPD layers. Outside the application of neural networks, these formulas can also be useful to derive gradients for machine learning algorithms, as in [17].

The main contributions of this brief are as follows. 1) to propose a self-contained paper calculating the gradients of matrix functions in a unified framework; 2) to prove the equivalence between the two main approaches to calculate SPD matrix gradients; and 3) to numerically compare them. The brief is organized as follows: in Section II we present and provide a proof of a formula for calculating the gradient of matrix functions in the general framework of diagonalizable matrices. In Section III, we address the particular case of SPD matrices. Finally, in Section IV we show that it is recommended to use Daleckiĭ–Kreĭn/Bhatia formula especially for numerical computation.

B. Notations

Matrices are boldface capital letters, vectors are lowercase boldface letters, and scalars are lowercase italic letters. $\mathbf{A}^T$ denote the transpose of $\mathbf{A}$. The (i, j)th element of $\mathbf{A}$ is denoted A_{ij}. The vector $\mathbf{1}_N$ denotes the vector populated with ones with dimension N. $\mathbf{I}_N$ is the identity matrix of size $N \times N$. The trace of a square matrix $\mathbf{A}$ is $\mathrm{tr}(\mathbf{A}) = \sum_n A_{nn}$. The symmetry operator is defined as $\mathrm{sym}(\mathbf{A}) = (\mathbf{A} + \mathbf{A}^T)/2$ while the skew operator is $\mathrm{skew}(\mathbf{A}) = (\mathbf{A} - \mathbf{A}^T)/2$. $\mathcal{S}(N) = \{\mathbf{S} \in M(N), \mathbf{S}^T = \mathbf{S}\}$ is the space of $N \times N$ symmetric matrices. $\mathcal{P}(N) = \{\mathbf{P} \in \mathcal{S}(N), \mathbf{u}^T\mathbf{P}\mathbf{u} > 0, \forall \mathbf{u} \neq \mathbf{0}_N \in \mathbb{R}^N\}$ the space of SPD matrices of size $N \times N$.

The Hadamard (or elementwise) product between two matrices of the same dimensions $\mathbf{A}$ and $\mathbf{B}$ is a matrix of the same dimensions, denoted as $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$, with matrix entries $C_{nm} = A_{nm}B_{nm}$. Obviously, we have $\mathbf{A} \odot \mathbf{B} = \mathbf{B} \odot \mathbf{A}$ and $(\mathbf{A} \odot \mathbf{B})^T = \mathbf{A}^T \odot \mathbf{B}^T$.

The Frobenius inner product $\langle \mathbf{A}, \mathbf{B} \rangle_F$ between two $N \times M$ real matrices $\mathbf{A}$ and $\mathbf{B}$ is a scalar denoted

$$\langle \mathbf{A}, \mathbf{B} \rangle_F = \mathbf{A} : \mathbf{B} = \mathrm{tr}\left(\mathbf{A}^T\mathbf{B}\right) = \sum_{nm} A_{nm}B_{nm} \tag{1}$$

where the double-contraction notation is intended to simplify notations.

The Frobenius norm of $\mathbf{A}$ is written as $\|\mathbf{A}\|_F = (\langle \mathbf{A}, \mathbf{A} \rangle_F)^{1/2}$. Some useful Frobenius inner product properties are given below for consistency. The cyclic property of the trace $\mathrm{tr}(\mathbf{A}^T\mathbf{B}) = \mathrm{tr}(\mathbf{B}\mathbf{A}^T)$ and

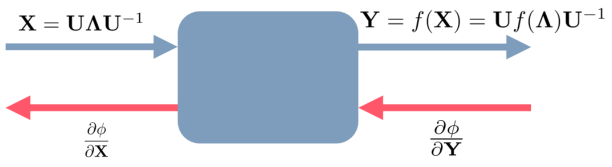
$\mathbf{X} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1}$
$\mathbf{Y} = f(\mathbf{X}) = \mathbf{U}f(\mathbf{\Lambda})\mathbf{U}^{-1}$
$\frac{\partial \phi}{\partial \mathbf{X}}$
$\frac{\partial \phi}{\partial \mathbf{Y}}$

Fig. 1. Layer representation, which typically illustrates matrix backpropagation.

the fact that $\mathrm{tr}(\mathbf{A}^T) = \mathrm{tr}(\mathbf{A})$ allow the Frobenius inner product to be rearranged in a number of ways

$$\mathbf{A} : \mathbf{B} = \mathbf{B} : \mathbf{A} = \mathbf{A}^T : \mathbf{B}^T \quad (2)$$
$$\mathbf{A} : \mathbf{DE} = \mathbf{D}^T\mathbf{A} : \mathbf{E} = \mathbf{AE}^T : \mathbf{D}. \quad (3)$$

A key property is that the Frobenius product commutes with the Hadamard product

$$\mathbf{A} : (\mathbf{B} \odot \mathbf{C}) = (\mathbf{A} \odot \mathbf{B}) : \mathbf{C} = (\mathbf{A} \odot \mathbf{C}) : \mathbf{B}. \quad (4)$$

Other final properties that can be easily demonstrated are

$$\mathbf{A} : \mathrm{sym}(\mathbf{B}) = \mathrm{sym}(\mathbf{A}) : \mathbf{B} \quad (5)$$
$$\mathrm{sym}(\mathrm{skew}(\mathbf{A}) \odot \mathbf{B}) = \mathrm{skew}(\mathbf{A}) \odot \mathrm{skew}(\mathbf{B}). \quad (6)$$

By using general properties of the Frobenius product (and in particular the interplay between the Frobenius and the Hadamard products) and standard rules of differentials, it is possible to derive simple rules to compute almost any matrix gradients. Finally, "Inf" terms denote infinity by the special value Inf due to the (1/0) quotient.

II. Gradient of Matrix Functions

In this section, we will present a concise formula allowing us to calculate the gradient of matrix functions efficiently. The main application is for backpropagation in matrix-based neural networks [8], [11]. Such a formula has been popularized by Bhatia [18, eq.(V.13)] in case of Hermitian matrices, and relies on the work of Ju L. Daleckiĭ and S. G. Kreĭn. In this section, we will extend it to a more general case that is for any diagonalizable matrices and prove it. Let ϕ be a scalar-valued function for which we aim to calculate the sensitivity with respect to the input matrix $\mathbf{X}$ knowing the sensitivity of ϕ with respect to the output $\mathbf{Y} = f(\mathbf{X})$. Fig. 1 shows the layer representation that we can typically find when building neural networks for a loss function ϕ. The blue arrows are the forward model from the known input $\mathbf{X}$ to the output $\mathbf{Y}$. The red arrows are sensitivities. The sensitivity of ϕ with respect to $\mathbf{Y}$, $(\partial\phi/\partial\mathbf{Y})$, is assumed to be known (for example computed by the next layer) and the one with respect to $\mathbf{X}$, $(\partial\phi/\partial\mathbf{X})$, is to be found (for example to be propagated to the previous layer).

A. Concise Formula for the Gradient of Diagonalizable Matrices

Let $\mathbf{X}$ be a diagonalizable matrix with the eigenvalue decomposition (EVD) $\mathbf{X} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1}$ where $\mathbf{\Lambda} = \mathrm{Diag}(\boldsymbol{\lambda})$ is a diagonal matrix containing the eigenvalues $\{\lambda_i\}_{1\le i\le N}$ and $\mathbf{U}$ is the matrix of eigenvectors.

Let us consider $\mathbf{Y} = f(\mathbf{X}) = \mathbf{U}f(\mathbf{\Lambda})\mathbf{U}^{-1}$ with f in the class $\mathcal{C}^1$. A key result is that the differential $d\mathbf{Y}$ is given by

$$\mathrm{d}\mathbf{Y} = \mathbf{U}\left[f^{[1]}(\mathbf{X}) \odot \mathbf{U}^{-1}(\mathrm{d}\mathbf{X})\mathbf{U}\right]\mathbf{U}^{-1}. \quad (7)$$

The elements of the symmetric Loewner matrix $f^{[1]}(\mathbf{X})$ can be written as the first-order divided differences

$$\left[f^{[1]}(\mathbf{X})\right]_{ij} = \begin{cases} \dfrac{f(\lambda_i) - f(\lambda_j)}{\lambda_i - \lambda_j}, & i \neq j \\ f'(\lambda_i), & \text{otherwise.} \end{cases} \quad (8)$$

Very few authors mentioned this formula in this general case; to our knowledge, it only appears in [19] yet without proof. In order to be self-contained, a new proof is given below in Section II-B. Another equivalent writing is the directional derivative of f at $\mathbf{X}$ in the direction of matrix $\mathbf{H}$

$$Df(\mathbf{X})[\mathbf{H}] = \mathbf{U}\left[f^{[1]}(\mathbf{X}) \odot \left(\mathbf{U}^{-1}\mathbf{H}\mathbf{U}\right)\right]\mathbf{U}^{-1}. \quad (9)$$

1) From Differential to Sensitivities: Using (7), we can easily derive the expression of the gradient of ϕ with respect to the matrix $\mathbf{X}$ starting from the definition of the matrix gradient and using the properties of the Frobenius and Hadamard products given in Section I-B

$$\begin{aligned} d\phi &= \frac{\partial\phi}{\partial\mathbf{Y}} : d\mathbf{Y} \\ &= \frac{\partial\phi}{\partial\mathbf{Y}} : \mathbf{U}\left[f^{[1]}(\mathbf{X}) \odot \mathbf{U}^{-1}d\mathbf{X}\mathbf{U}\right]\mathbf{U}^{-1} \\ &= \mathbf{U}^{-T}\left[f^{[1]}(\mathbf{X}) \odot \mathbf{U}^T\frac{\partial\phi}{\partial\mathbf{Y}}\mathbf{U}^{-T}\right]\mathbf{U}^T : d\mathbf{X}. \end{aligned}$$

So the matrix gradient is

$$\frac{\partial\phi}{\partial\mathbf{X}} = \mathbf{U}^{-T}\left[f^{[1]}(\mathbf{X}) \odot \mathbf{U}^T\frac{\partial\phi}{\partial\mathbf{Y}}\mathbf{U}^{-T}\right]\mathbf{U}^T. \quad (10)$$

Knowing the sensitivity of the objective function with respect to the output $(\partial\phi/\partial\mathbf{Y})$, we are able to backpropagate the gradient and compute the sensitivity of the objective function with respect to the output $(\partial\phi/\partial\mathbf{X})$. To our knowledge, this result is new. This is also illustrated in Fig. 1.

B. Proof of (7)

Lemma 1: **Right and left multiplication by a diagonal matrix.** Let $\mathbf{A}$ be a $N \times N$ square matrix and $\mathbf{\Lambda} = \mathrm{Diag}(\boldsymbol{\lambda})$ a $N \times N$ diagonal matrix. Then

$$[\mathbf{A}, \mathbf{\Lambda}] \doteq \mathbf{A\Lambda} - \mathbf{\Lambda A} = 2\mathrm{skew}\left(\mathbf{1}_N\boldsymbol{\lambda}^T\right) \odot \mathbf{A}.$$

Proof: The Proof of Lemma 1 is immediate using the definition of the skew operator

$$\begin{aligned} \mathbf{A\Lambda} - \mathbf{\Lambda A} &= \mathbf{1}_N\boldsymbol{\lambda}^T \odot \mathbf{A} - \boldsymbol{\lambda}\mathbf{1}_N^T \odot \mathbf{A} \\ &= 2\mathrm{skew}\left(\mathbf{1}_N\boldsymbol{\lambda}^T\right) \odot \mathbf{A}. \end{aligned}$$

1) Step 1—Calculation of the Differentials $d\mathbf{U}, d\mathbf{\Lambda}$: Let us consider the EVD $\mathbf{X} = \mathbf{U\Lambda U}^{-1}$ with $\mathbf{\Lambda} = \mathrm{Diag}(\boldsymbol{\lambda})$. After some manipulations, one obtains the differential $d\mathbf{X}$ $d\mathbf{X} = (d\mathbf{U})\mathbf{\Lambda U}^{-1} + \mathbf{U}(d\mathbf{\Lambda})\mathbf{U}^{-1} - \mathbf{U\Lambda U}^{-1}(d\mathbf{U})\mathbf{U}^{-1}$ and with use of Lemma 1

$$\begin{aligned} \mathbf{B} \doteq \mathbf{U}^{-1}(d\mathbf{X})\mathbf{U} &= (d\mathbf{C})\mathbf{\Lambda} + d\mathbf{\Lambda} - \mathbf{\Lambda}(d\mathbf{C}) \\ &= 2\mathrm{skew}\left(\mathbf{1}_N\boldsymbol{\lambda}^T\right) \odot d\mathbf{C} + d\mathbf{\Lambda} \end{aligned} \quad (11)$$

where $d\mathbf{C} = \mathbf{U}^{-1}(d\mathbf{U})$ and $d\mathbf{U}^{-1} = -\mathbf{U}^{-1}(d\mathbf{U})\mathbf{U}^{-1}$.

The matrix $2\mathrm{skew}(\mathbf{1}_N\boldsymbol{\lambda}^T)$ is antisymmetric with (i,j)th element equal to $\lambda_j - \lambda_i$.

Diagonal elements of matrix $\mathbf{B}$ coincide with the diagonal elements of $d\mathbf{\Lambda}$ which we write

$$d\mathbf{\Lambda} = \mathbf{I}_N \odot \mathbf{B}. \quad (12)$$

From here, it is convenient to set the diagonal elements of $d\mathbf{C}$ to zero [20] and introduce the antisymmetric matrix $\mathbf{K}$ with elements

$$K_{ij} = \begin{cases} \dfrac{1}{\lambda_i - \lambda_j}, & i \neq j \\ 0, & \text{otherwise.} \end{cases} \quad (13)$$

It follows $d\mathbf{C} = \mathbf{U}^{-1}d\mathbf{U} = \mathbf{K}^T \odot \mathbf{B}$ and thus:

$$d\mathbf{U} = \mathbf{U}\left[\mathbf{K}^T \odot \mathbf{B}\right]. \quad (14)$$

This result was first shown by Giles in [20].

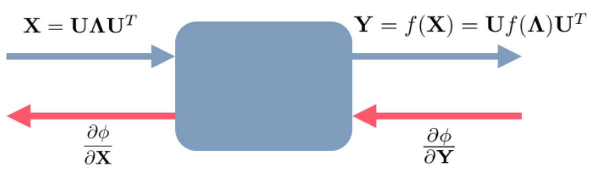
$\mathbf{X} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T$
$\mathbf{Y} = f(\mathbf{X}) = \mathbf{U}f(\boldsymbol{\Lambda})\mathbf{U}^T$
$\frac{\partial\phi}{\partial\mathbf{X}}$
$\frac{\partial\phi}{\partial\mathbf{Y}}$

Fig. 2. Eigen-layer representation for SPD matrices.

2) Step 2—Calculation of the Differential $d\mathbf{Y}$: Let consider now $\mathbf{Y} = f(\mathbf{X}) = \mathbf{U}f(\boldsymbol{\Lambda})\mathbf{U}^{-1}$. The same arguments as in previous paragraph and (12), (14) give

$$\begin{aligned}\mathbf{U}^{-1}(d\mathbf{Y})\mathbf{U} &= 2\mathrm{skew}\left(\mathbf{1}_N f(\lambda)^T\right)\odot d\mathbf{C} + f'(\boldsymbol{\Lambda})\odot d\boldsymbol{\Lambda}\\ &= 2\mathrm{skew}\left(\mathbf{1}_N f(\lambda)^T\right)\odot\left[\mathbf{K}^T\odot\mathbf{B}\right]\\ &\quad + f'(\boldsymbol{\Lambda})\odot\mathbf{B}\\ &= f^{[1]}(\mathbf{X})\odot\mathbf{B}\end{aligned}$$

with the symmetric Loewner matrix $f^{[1]}(\mathbf{X})$

$$f^{[1]}(\mathbf{X}) = 2\mathrm{skew}\left(\mathbf{1}_N f(\lambda)^T\right)\odot\mathbf{K}^T + f'(\boldsymbol{\Lambda}).$$

The elements of the Loewner matrix can be simply written as divided differences

$$\left[f^{[1]}(\mathbf{X})\right]_{ij} = \begin{cases}\dfrac{f(\lambda_i) - f(\lambda_j)}{\lambda_i - \lambda_j}, & i \neq j\\ f'(\lambda_i), & \text{otherwise.}\end{cases}$$

Putting things together yields the key differential

$$d\mathbf{Y} = \mathbf{U}\left[f^{[1]}(\mathbf{X})\odot\mathbf{U}^{-1}(d\mathbf{X})\mathbf{U}\right]\mathbf{U}^{-1}. \tag{15}$$

III. SPD Matrix Gradient

We will now address a particular case of interest when $\mathbf{X}$ is an SPD matrix, as this case is more frequently encountered in the literature. General formulas for the gradient have been developed by Bhatia [18], Daleckiǐ and Kreǐn [21], and Ionescu [8]. This section will first present and correct these two different ways to calculate SPD matrix gradients alongside important differentials, and second, prove their equivalence. The EVD now writes the $N \times N$ matrix $\mathbf{X} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T$ with $\mathbf{U}^T\mathbf{U} = \mathbf{I}_N$ and $\boldsymbol{\Lambda} = \mathrm{Diag}(\lambda)$ with positive diagonal elements. Let us consider $\mathbf{Y} = f(\mathbf{X}) = \mathbf{U}f(\boldsymbol{\Lambda})\mathbf{U}^T$ with f in the class $\mathcal{C}^1$.

Fig. 2 is now a layer representation in the SPD case for a matrix function f. For instance, in the network developed by Huang and Gool [11], the *LogEig* layer simply implements the matrix logarithm $\mathbf{Y} = \log(\mathbf{X})$, i.e., $f(x) = \log(x)$, while the *ReEig* layer implements $\mathbf{Y} = \mathbf{U}\max(\boldsymbol{\Lambda}, \epsilon\mathbf{I}_N)\mathbf{U}^T$, i.e., $f(x) = \max(x, \epsilon)$.

A. Daleckiǐ–Kreǐn/Bhatia Formula

The previous developments to prove (7) for any diagonalizable matrices still hold since they include the case of $\mathbf{X}$ being SPD (see Fig. 2). In this case, $\mathbf{X} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T$ with $\mathbf{U}^{-1} = \mathbf{U}^T$.

The differential of $\mathbf{X}$ writes in the SPD case $d\mathbf{X} = (d\mathbf{U})\boldsymbol{\Lambda}\mathbf{U}^T + \mathbf{U}(d\boldsymbol{\Lambda})\mathbf{U}^T + \mathbf{U}\boldsymbol{\Lambda}(d\mathbf{U})^T$.

Because $\mathbf{U}^T\mathbf{U} = \mathbf{I}_N$, then $(d\mathbf{U})^T\mathbf{U} + \mathbf{U}^T(d\mathbf{U}) = \mathbf{0}$. Introducing the matrix $d\mathbf{C} = \mathbf{U}^T(d\mathbf{U})$, we immediately see that $d\mathbf{C}$ is antisymmetric, i.e., $d\mathbf{C}^T = -d\mathbf{C}$. It holds

$$\begin{aligned}\mathbf{U}^T(d\mathbf{X})\mathbf{U} &= (d\mathbf{C})\boldsymbol{\Lambda} + d\boldsymbol{\Lambda} + \boldsymbol{\Lambda}(d\mathbf{C})^T\\ &= (d\mathbf{C})\boldsymbol{\Lambda} + d\boldsymbol{\Lambda} - \boldsymbol{\Lambda}(d\mathbf{C}).\end{aligned}$$

The expression is identical to (11) so we can apply the same reasoning as in Section II-B to obtain the new differentials

$$\begin{aligned}d\boldsymbol{\Lambda} &= \mathbf{I}_N\odot\mathbf{U}^T(d\mathbf{X})\mathbf{U}\\ d\mathbf{U} &= \mathbf{U}\left[\mathbf{K}^T\odot\mathrm{sym}\left(\mathbf{U}^T(d\mathbf{X})\mathbf{U}\right)\right].\end{aligned}$$

The differential formulation of Daleckiǐ–Kreǐn/Bhatia formula that we can find in [18] is given by

$$d\mathbf{Y} = \mathbf{U}\left[f^{[1]}(\mathbf{X})\odot\mathbf{U}^T(d\mathbf{X})\mathbf{U}\right]\mathbf{U}^T \tag{16}$$

which yields similar to (10) the following expression of the gradient known as Daleckiǐ–Kreǐn/Bhatia formula:

$$\frac{\partial\phi}{\partial\mathbf{X}} = \mathbf{U}\left[f^{[1]}(\mathbf{X})\odot\mathbf{U}^T\frac{\partial\phi}{\partial\mathbf{Y}}\mathbf{U}\right]\mathbf{U}^T. \tag{17}$$

Another equivalent writing is the directional derivative of f at $\mathbf{X}$ in the direction of matrix $\mathbf{H}$

$$Df(\mathbf{X})[\mathbf{H}] = \mathbf{U}\left[f^{[1]}(\mathbf{X})\odot\left(\mathbf{U}^T\mathbf{H}\mathbf{U}\right)\right]\mathbf{U}^T. \tag{18}$$

This is [18, eq. (V.13)].

B. Ionescu Approach

Ionescu et al. [8] calculated gradients for matrix backpropagation to map the partial derivatives at two consecutive layers, generalizing the chain rule to matrices. In 2016, an extended version of this brief was published with some modifications [13]. However, the formula (14) given in [8] contains a misplaced sym operator, and in the formulation (35) in [13] this operator does not appear. Yet, the calculated gradient has to be symmetric, so these formulas contain mistakes. Moreover, some authors like Huang and Gool [11] with (15) or like Engin et al. [16] with *Proposition 1* rely on the inaccurate expression mentioned in [8] (14). Furthermore, Ionescu does not give the final expression of the gradient, although the intermediate partial derivatives are given (in Section 4.2 in [8] and through *Proposition 4* in [13]). We give here a correct alternative to Daleckiǐ–Kreǐn/Bhatia formula, based on the ideas of Ionescu [8], [13], using the matrix $\mathbf{C} = \mathbf{U}^T(\partial\phi/\partial\mathbf{Y})\mathbf{U}$ and the matrix $\mathbf{K}$ defined in (13)

$$\frac{\partial\phi}{\partial\mathbf{X}} = \mathbf{U}\left[2\mathrm{sym}\left(\mathbf{K}^T\odot\mathbf{C}f(\boldsymbol{\Lambda})\right) + \mathbf{I}_N\odot f'(\boldsymbol{\Lambda})\mathbf{C}\right]\mathbf{U}^T. \tag{19}$$

We also propose a new equivalent formulation with the skew operator using property (6)

$$\frac{\partial\phi}{\partial\mathbf{X}} = \mathbf{U}\left[\mathbf{K}^T\odot 2\mathrm{skew}\left(\mathbf{C}f(\boldsymbol{\Lambda})\right) + \mathbf{I}_N\odot f'(\boldsymbol{\Lambda})\mathbf{C}\right]\mathbf{U}^T. \tag{20}$$

The proofs of (19) and (20) are given in Section III-B1.

1) Proof of (19): This proof aims at giving an alternative to Daleckiǐ–Kreǐn/Bhatia formula (for the EVD of $\mathbf{X}$ SPD) and is inspired by the work of Ionescu, who specifically calculated matrix gradients for neural networks backpropagation before Daleckiǐ–Kreǐn/Bhatia formula became widespread.

Some authors tried to demonstrate the equivalence between Daleckiǐ–Kreǐn/Bhatia formula and Ionescu formulation, such as [16, Appendix II]. However, the starting point of the proof relies on the inaccurate formulation of the matrix gradient proposed in [8], which is not symmetric. Their proof, therefore, has flaws.

As it has been done in step 1 and mentioned in step 2, we can easily show that (12) and (14) still hold for $\mathbf{X}$ SPD

$$\begin{aligned}d\boldsymbol{\Lambda} &= \mathbf{I}_N\odot\mathbf{B}\\ d\mathbf{U} &= \mathbf{U}\left[\mathbf{K}^T\odot\mathrm{sym}(\mathbf{B})\right]\end{aligned}$$

with $\mathbf{B} \doteq \mathbf{U}^T(d\mathbf{X})\mathbf{U}$ symmetric.

Knowing that $(\partial\phi/\partial\mathbf{Y})$ is also symmetric, it yields

$$\begin{aligned}d\phi &= \frac{\partial\phi}{\partial\mathbf{Y}} : d\mathbf{Y}\\ &= \frac{\partial\phi}{\partial\mathbf{Y}} : (d\mathbf{U})f(\boldsymbol{\Lambda})\mathbf{U}^T + \mathbf{U}f'(\boldsymbol{\Lambda})(\mathrm{d}\boldsymbol{\Lambda})\mathbf{U}^T + \mathbf{U}f(\boldsymbol{\Lambda})(d\mathbf{U})^T\\ &= 2\frac{\partial\phi}{\partial\mathbf{Y}}\mathbf{U}f(\boldsymbol{\Lambda}) : d\mathbf{U} + f'(\boldsymbol{\Lambda})\left[\mathbf{U}^T\frac{\partial\phi}{\partial\mathbf{Y}}\mathbf{U}\right] : d\boldsymbol{\Lambda}\\ &= 2\mathbf{K}^T\odot\mathbf{C}f(\boldsymbol{\Lambda}) : \mathrm{sym}(\mathbf{B}) + f'(\boldsymbol{\Lambda})\mathbf{C} : d\boldsymbol{\Lambda}\\ &= \left[2\mathrm{sym}\left[\mathbf{K}^T\odot\mathbf{C}f(\boldsymbol{\Lambda})\right] + \mathbf{I}_N\odot f'(\boldsymbol{\Lambda})\mathbf{C}\right] : \mathbf{B}\\ &= \mathbf{U}\left[2\mathrm{sym}\left[\mathbf{K}^T\odot\mathbf{C}f(\boldsymbol{\Lambda})\right] + \mathbf{I}_N\odot f'(\boldsymbol{\Lambda})\mathbf{C}\right]\mathbf{U}^T : d\mathbf{X}.\end{aligned}$$

C. Discussion

In the literature, authors often refer to the expression of SPD matrix gradient (17) as Daleckiĭ–Kreĭn/Bhatia formula, especially since Brooks et al. [14] spread it with that denomination in. We ourselves adhered to that denomination. The original formulation proposed by Daleckiĭ and Kreĭn [21] known as Daleckiĭ–Kreĭn theorem proved in Section IV-A is more theoretical (cf. (2) in [21]) and in practice Rajendra Bhatia's formulation [18, eq. (V.13)] is more used to derive the gradient.

Starting from the EVD of $\mathbf{X}$ we derived both Daleckiĭ–Kreĭn/Bhatia formula (17) and the expression of the gradient as Ionescu (19) or (20), and proved that these formulations are equivalent. Mathematically we see that the main difference between the two formulas lies in the fact that the $\mathbf{C} = \mathbf{U}^T(\partial\phi/\partial\mathbf{Y})\mathbf{U}$ matrix has been isolated in Daleckiĭ–Kreĭn/Bhatia formula (17) and not in (19). So overall, Daleckiĭ–Kreĭn/Bhatia formula contains fewer operations and is more concise. We will see in Section IV-C that this has a positive impact on computational time. Moreover, we will also prefer Daleckiĭ–Kreĭn/Bhatia formula for its numerical stability compared with (19), as evoked in [15]. The key to seeing it lies in the fact that the latter requires the computation of the matrix $\mathbf{K}$ (13) and, in particular of the term $(1/\lambda_i - \lambda_j)$ which tends toward infinity when the eigenvalues become close to each other. It triggers numerical instability. On the other side Daleckiĭ–Kreĭn/Bhatia formula implements Loewner matrix (8) with the off-diagonal term $(f(\lambda_i) - f(\lambda_j)/\lambda_i - \lambda_j)$ which approaches the derivative of f at λ_i when two eigenvalues are close. So in this case, it is possible to numerically correct the *Inf* values—caused by close eigenvalues—by forcing them to be equal to $f'(\lambda_i)$.

We are convinced that clarifying literature in this field of study will serve as a basis to implement useful tools. Indeed, even if manifold deep learning became a major technique in various fields, there are currently no tools in PyTorch to implement SPD neural networks efficiently. Daleckiĭ–Kreĭn/Bhatia formula lays the foundations for building SPD layers requiring matrix backpropagation.

IV. Results

In this section, we describe how the theory developed previously can be applied on practical examples. We want first to illustrate how to apply Daleckiĭ–Kreĭn/Bhatia formula to derive useful results of the literature and then to numerically compare the two methods described on real-world applications involving SPD matrices.

A. Daleckiĭ–Kreĭn Theorem

We propose hereafter, via Proposition 1, a way of connecting with the original calculations proposed by Daleckiĭ and Kreĭn [21, Th. 1].

Proposition 1: Let us introduce the parameter t such that the elements of the matrix $\mathbf{X}(t)$ are differentiable functions of that parameter. We denote by $\dot{\mathbf{X}}(t)$ the derivative of $\mathbf{X}$ with respect to t.

Start with (7) to write

$$d\mathbf{Y} = \mathbf{U}\left[f^{[1]}(\mathbf{X}) \odot \mathbf{U}^T (\mathrm{d}\mathbf{X})\,\mathbf{U}\right]\mathbf{U}^T = \mathbf{U}\mathbf{G}\mathbf{U}^T dt$$

where $\mathbf{G} = f^{[1]}(\mathbf{X}) \odot \mathbf{U}^T\dot{\mathbf{X}}(t)\mathbf{U} = \sum_{i,j}\mathbf{G}_{ij}\mathbf{e}_i\mathbf{e}_j^T$.

The gradient is thus

$$\begin{aligned}\dot{\mathbf{Y}}(t) &= \frac{\partial\mathbf{Y}}{\partial t} = \mathbf{U}\mathbf{G}\mathbf{U}^T\\ &= \sum_{i,j} G_{ij}\mathbf{U}\left(\mathbf{e}_i\mathbf{e}_j^T\right)\mathbf{U}^T = \sum_{i,j} G_{ij}\mathbf{u}_i\mathbf{u}_j^T\\ &= \sum_{i,j} L_{ij}\mathbf{u}_i\left(\mathbf{e}_i^T\mathbf{U}^T\dot{\mathbf{X}}(t)\,\mathbf{U}\mathbf{e}_j\right)\mathbf{u}_j^T\\ &= \sum_{i,j} L_{ij}\mathbf{u}_i\left[\mathbf{u}_i^T\dot{\mathbf{X}}(t)\,\mathbf{u}_j\right]\mathbf{u}_j^T\\ &= \sum_{i,j} L_{ij}\mathbf{E}_i\dot{\mathbf{X}}(t)\,\mathbf{E}_j \end{aligned}\tag{21}$$

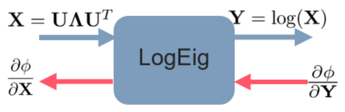
$\mathbf{X} = \mathbf{U}\Lambda\mathbf{U}^T$
$\mathbf{Y} = \log(\mathbf{X})$
LogEig
$\frac{\partial\phi}{\partial\mathbf{X}}$
$\frac{\partial\phi}{\partial\mathbf{Y}}$

Fig. 3. LogEig layer representation.

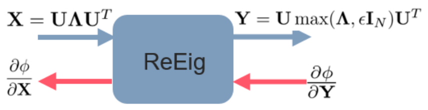
$\mathbf{X} = \mathbf{U}\Lambda\mathbf{U}^T$
$\mathbf{Y} = \mathbf{U}\max(\Lambda, \epsilon\mathbf{I}_N)\mathbf{U}^T$
ReEig
$\frac{\partial\phi}{\partial\mathbf{X}}$
$\frac{\partial\phi}{\partial\mathbf{Y}}$

Fig. 4. ReEig layer representation.

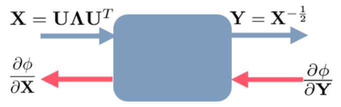
$\mathbf{X} = \mathbf{U}\Lambda\mathbf{U}^T$
$\mathbf{Y} = \mathbf{X}^{-\frac{1}{2}}$
$\frac{\partial\phi}{\partial\mathbf{X}}$
$\frac{\partial\phi}{\partial\mathbf{Y}}$

Fig. 5. Inverse square root layer representation.

where $\mathbf{E}_i = \mathbf{u}_i\mathbf{u}_i^T$ is the projection matrix onto the ith eigenvector of $\mathbf{X}$ and $L_{i,j}$ denotes the coefficients of the Loewner matrix $\mathbf{L} = f^{[1]}(\mathbf{X})$ given in (8). Equation (21) is known as Daleckiĭ–Kreĭn theorem [21, eq. (2)].

B. Applications

First, we present various applications of Daleckiĭ–Kreĭn/Bhatia formula, allowing to efficiently prove useful results.

1) We define some layers of the literature. Figs. 3–5 describe their representation as we introduced previously. Daleckiĭ–Kreĭn/Bhatia differential formulation makes it straightforward to calculate the differential $d\mathbf{Y}$ for each of them.

LogEig layer is described with Fig. 3.

It is easy to calculate the Loewner matrix using (8) on the corresponding analytic function $x \in \mathbb{R} \mapsto \log(x)$

$$\left[\log^{[1]}(\mathbf{X})\right]_{ij} = \begin{cases}\dfrac{\log(\lambda_i) - \log(\lambda_j)}{\lambda_i - \lambda_j}, & i \neq j\\[2mm] \dfrac{1}{\lambda_i}, & \text{otherwise.}\end{cases}$$

It yields the differential

$$\mathrm{d}\mathbf{Y} = \mathbf{U}\left[\log^{[1]}(\mathbf{X}) \odot \mathbf{U}^T(d\mathbf{X})\,\mathbf{U}\right]\mathbf{U}^T. \tag{22}$$

Similarly, for *ReEig* layer Fig. 4 one gets

$$\left[\max^{[1]}(\mathbf{X})\right]_{ij} = \begin{cases}\dfrac{\max(\epsilon, \lambda_i) - \max(\epsilon, \lambda_j)}{\lambda_i - \lambda_j}, & i \neq j\\[2mm] \begin{cases}1, & \lambda_i > \epsilon\\ 0, & \lambda_i \leq \epsilon,\end{cases} & \text{otherwise.}\end{cases}$$

It yields the differential

$$\mathrm{d}\mathbf{Y} = \mathbf{U}\left[\max^{[1]}(\mathbf{X}) \odot \mathbf{U}^T(d\mathbf{X})\,\mathbf{U}\right]\mathbf{U}^T.$$

LogEig and *ReEig* layers are both mainly used for SPD neural networks, see [11].

Other layers can be used for normalization purposes [14], [22]. For instance, that is the case of the matrix inverse square root layer Fig. 5. Let $f : x \mapsto (1/(x)^{1/2}), x \neq 0$ it holds $f'(x) = (-1/2x(x)^{1/2})$.

The Loewner matrix has elements

$$\left[f^{[1]}(\mathbf{X})\right]_{ij} = \begin{cases}\dfrac{\lambda_i^{-1/2} - \lambda_j^{-1/2}}{\lambda_i - \lambda_j}, & i \neq j\\[2mm] \dfrac{-1}{2\lambda_i\sqrt{\lambda_i}}, & \text{otherwise.}\end{cases}$$

In this particular case, we can simply write

$$\left[f^{[1]}(\mathbf{X})\right]_{ij} = \frac{-1}{\sqrt{\lambda_i}\sqrt{\lambda_j}\left(\sqrt{\lambda_i}+\sqrt{\lambda_j}\right)}$$

which is consistent with the results obtained in [22] through a different method. We can then calculate

$$d\mathbf{Y} = \mathbf{U}\left[f^{[1]}(\mathbf{X})\odot\mathbf{U}^T(d\mathbf{X})\mathbf{U}\right]\mathbf{U}^T.$$

1) Outside the applications of neural networks Daleckiĭ–Kreĭn/Bhatia formula is a convenient tool to compute some SPD matrix gradients. Next, we propose a way to calculate the squared Frobenius norm of the matrix logarithm. Let $\mathbf{X} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T$ be an SPD matrix

$$\begin{aligned}\phi &= \frac{1}{2}\left\|\log(\mathbf{X})\right\|_F^2\\ d\phi &= \log(\mathbf{X}) : d\log(\mathbf{X})\\ &= \left[\mathbf{U}^T\log(\mathbf{X})\mathbf{U}\right]\odot\log^{[1]}(\mathbf{X}) : \mathbf{U}^T(d\mathbf{X})\mathbf{U}\\ &= \log(\boldsymbol{\Lambda})\odot\log^{[1]}(\mathbf{X}) : \mathbf{U}^T(d\mathbf{X})\mathbf{U}\\ &= \log(\boldsymbol{\Lambda})\boldsymbol{\Lambda}^{-1} : \mathbf{U}^T(d\mathbf{X})\mathbf{U}\\ &= \mathbf{U}\left[\log(\boldsymbol{\Lambda})\boldsymbol{\Lambda}^{-1}\right]\mathbf{U}^T : d\mathbf{X}\end{aligned}$$

since $[\log^{[1]}(\mathbf{X})]_{ii} = (1/\lambda_i)$. The matrix gradient is

$$\frac{\partial\phi}{\partial\mathbf{X}} = \log(\mathbf{X})\mathbf{X}^{-1} = \mathbf{X}^{-1}\log(\mathbf{X}).$$

2) One can also use Daleckiĭ–Kreĭn/Bhatia formula to derive metrics or kernels as for example in [17], although the authors did not mention its name. Here is how to find the canonical log-Euclidean kernel with this method. In the following, we denote $\mathbf{P} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T$, $\log\mathbf{P} = \mathbf{U}\boldsymbol{\Gamma}\mathbf{U}^T$ with $\boldsymbol{\Gamma} \doteq \log(\boldsymbol{\Lambda})$, i.e., $\gamma_i = \log\lambda_i$:

$$\begin{aligned}&\log_{\mathbf{P}}(\mathbf{X})\\ &\quad\doteq D\exp(\log(\mathbf{P}))[\log\mathbf{X}-\log\mathbf{P}]\\ &\quad= \mathbf{U}\left[\exp^{[1]}(\Gamma)\odot\mathbf{U}^T[\log\mathbf{X}-\log\mathbf{P}]\mathbf{U}\right]\mathbf{U}^T.\end{aligned}$$

Using $D\log(\mathbf{P})[\mathbf{H}] = \mathbf{U}[\log^{[1]}(\boldsymbol{\Lambda})\odot\mathbf{U}^T\mathbf{H}\mathbf{U}]\mathbf{U}^T$, it follows:

$$\begin{aligned}&D\log(\mathbf{P})\left[\log_{\mathbf{P}}(\mathbf{X})\right]\\ &\quad= \mathbf{U}\left[\log^{[1]}(\Lambda)\odot\exp^{[1]}(\Gamma)\odot\right.\\ &\quad\left.\mathbf{U}^T[\log\mathbf{X}-\log\mathbf{P}]\mathbf{U}\right]\mathbf{U}^T\\ &\quad= \mathbf{U}\left[\mathbf{J}\odot\mathbf{U}^T[\log\mathbf{X}-\log\mathbf{P}]\mathbf{U}\right]\mathbf{U}^T \qquad (23)\\ &\quad= \log\mathbf{X}-\log\mathbf{P} \qquad (24)\end{aligned}$$

where $\mathbf{J} = \log^{[1]}(\boldsymbol{\Lambda})\odot\exp^{[1]}(\boldsymbol{\Gamma})$.
One goes from (23) to (24) knowing that $\mathbf{J}$ is the all-ones matrix and that $\mathbf{U}\mathbf{U}^T = \mathbf{I}_N$. Indeed, diagonal terms of $\mathbf{J}$ are $(1/\lambda_i)\cdot\exp[\gamma_i] = 1$ and its off-diagonal terms are also $(\log\lambda_i - \log\lambda_j/\lambda_i-\lambda_j)\cdot(\exp\gamma_i-\exp\gamma_j/\gamma_i-\gamma_j) = 1$.

C. Numerical Comparison

The goal of this section is to numerically compare the two presented approaches: Ionescu approach (19) and Daleckiĭ–Kreĭn/Bhatia formula (17) used for calculating SPD matrix gradients. In order to alleviate SPD matrices' ill-conditioning, we add a small constant of 10^{-5} to the diagonal elements. This well-known regularization prevents zero eigenvalues [23]. In the following section, we aim to show whether one method is computationally recommended to implement compared with the other.

TABLE I

Gradients Computation Time VIA Formulas (17) and (19) for Different Number of Input Matrices. The Last Column Gives the Difference in Absolute Value Between the Two Computation Times. In Boldface is the Fastest Time

# Samples	Method	Time (s)	Abs. time diff (s)
1,000	(17) (19)	**0.15621** 0.25004	0.09383
10,000	(17) (19)	**1.76713** 2.46879	0.70166

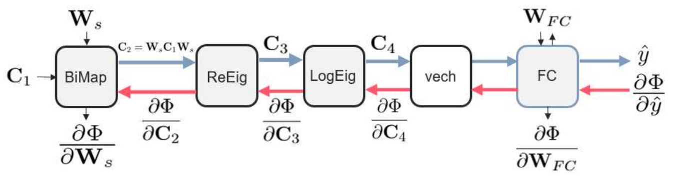
$\mathbf{W}_s$
$\mathbf{C}_1$
BiMap
$\mathbf{C}_2 = \mathbf{W}_s\mathbf{C}_1\mathbf{W}_s$
ReEig
$\mathbf{C}_3$
LogEig
$\mathbf{C}_4$
vech
$\mathbf{W}_{FC}$
FC
$\hat{y}$
$\frac{\partial\Phi}{\partial\hat{y}}$
$\frac{\partial\Phi}{\partial\mathbf{W}_s}$
$\frac{\partial\Phi}{\partial\mathbf{C}_2}$
$\frac{\partial\Phi}{\partial\mathbf{C}_3}$
$\frac{\partial\Phi}{\partial\mathbf{C}_4}$
$\frac{\partial\Phi}{\partial\mathbf{W}_{FC}}$

Fig. 6. SPDNet architecture used for the classification of MI tasks. Forward pass is represented with blue arrows: $\mathbf{C}_1$ is the input covariance matrix, ReEig computes $\mathbf{C}_3 = \mathbf{U}_2\max(\boldsymbol{\Lambda}_2, \epsilon\mathbf{I}_N)\mathbf{U}_2^T$ knowing the EVD of $\mathbf{C}_2 = \mathbf{U}_2\boldsymbol{\Lambda}_2\mathbf{U}_2^T$, LogEig computes $\mathbf{C}_4 = \log(\mathbf{C}_3)$, vech is a vectorizing operation for symmetric matrices, FC is a fully connected layer. Backward propagation of the gradient is in red. Φ is the cost function. The weights $\mathbf{W}$ and $\mathbf{W}_{FC}$ are the parameters learned by the network via BP.

1) Computation Time: In order to analyse the computation time of the two methods, we take as an example the *LogEig* layer, see Fig. 3. For that we implement $\mathbf{Y} = \log(\mathbf{X})$. Let us randomly generate samples of two SPD matrices corresponding to the input $\mathbf{X}$ and the sensitivity with respect to $\mathbf{Y}$ (report to Fig. 3), respectively. Given that, we evaluate the computation time when calculating the sensitivity with respect to $\mathbf{X}$ through the two methods. Results are presented in Table I for matrices of size 4×4.

Obviously, we need a rather large number of samples to better observe the computation times; otherwise, with few samples, it is too fast to really observe a difference. The implementation of Daleckiĭ–Kreĭn/Bhatia formula is faster. Indeed Table I shows that it is 0.0938 s faster than (19) with 1000 matrices, that is 37.51% faster. For 10 000 matrices it is 28.42% faster. We made similar observations with bigger matrices, for example, taking 10 matrices of size 1000×1000 computing (19) takes around 107 s and (17) is almost 15 s (i.e., 13.6%) faster. This aspect is to keep of note when implementing algorithms operating on large amounts of data, which can be the case of neural networks, for example.

2) EEG-Based MI Classification by SPDNet: We will compare the two formulas for matrix backpropagation (BP) through the same use case: the classification of MI tasks via a neural network specially developed for SPD matrices called SPDNet [11]. Its architecture is shown on Fig. 6. For this study, the implementation of Daleckiĭ–Kreĭn/Bhatia formula (17) corrects "Inf" values in (8) by replacing the off-diagonal elements by the derivative of f at λ_i. No correction can be brought to Ionescu formula (19) as explained in Section III-C.

The architecture of Fig. 6 is implemented with the two different backpropagation formulas in such a way that the sensitivities of Φ with respect to $\mathbf{C}_3(\partial\Phi/\partial\mathbf{C}_3)$, and with respect to $\mathbf{C}_2(\partial\Phi/\partial\mathbf{C}_2)$ are calculated either with Daleckiĭ–Kreĭn/Bhatia formula (17) or with Ionescu approach (19).

The data we will be using for this study are from the well-known public dataset BCI Competition IV 2A [24] that can be found on MOABB [25]. Throughout two sessions, nine subjects were asked to perform motor imagery tasks, which consist of four classes of movements: left-hand, right-hand, feet, and tongue; following a specific protocol [25]. One refers to a trial as a segment triggered by an external stimulus (the cue) corresponding to one mental motor imagery task. In total, there are 72 4-s length trials per class, yielding

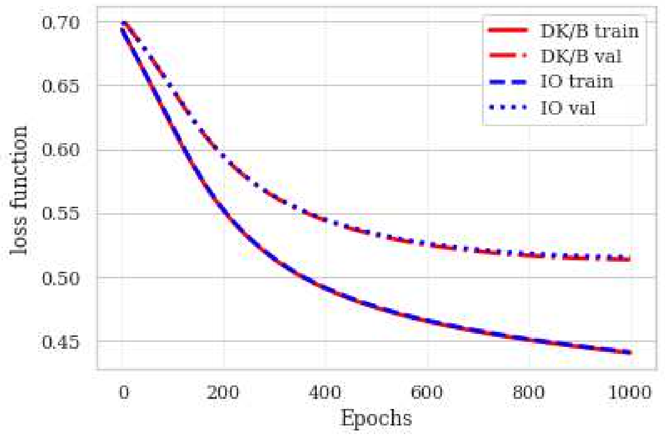

Fig. 7. Loss function curves comparison of the two BP methods for subject S1, session 1, left-hand/right-hand motor imagery. The dataset used is BCI Competition IV 2A. The loss is represented during training and during validation. DK/B and IO, respectively, denote formulas (17) and (19).

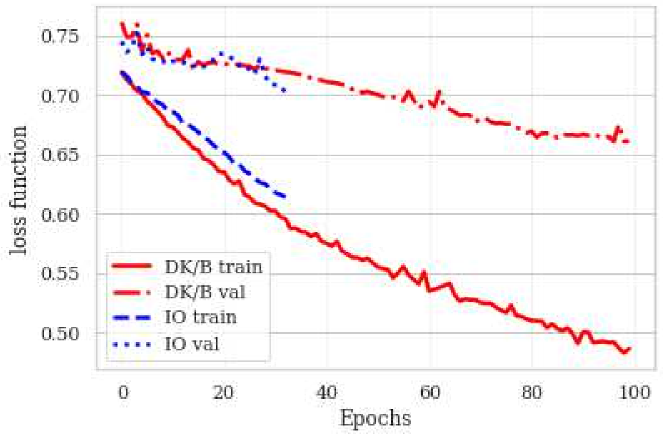

Fig. 8. Training and validation curves for the two BP methods, subject S1, session 1, left-hand/right-hand motor imagery. The trial duration is now 40 ms to point out stability issues with the Ionescu formula. DK/B and IO, respectively, denote formulas (17) and (19).

a total of 288 trials per subject per session. In our study, we will focus on classifying left-hand/right-hand movements of subject S1 for session 1. It represents 144 trials that are converted into spatial covariance matrices (that are SPD) for the input of the network. We train the network on 80% of these data and perform validation on the rest. The loss function is chosen to be a cross-entropy loss.

1) *First Result—Convergence of the Two BP Methods:* We chose this example as a basic use case of matrix backpropagation more than for the data analysis. We see from Fig. 7 that the training takes place in a similar way with both BP methods. Indeed, the learning curves decrease and superpose, proving convergence and equivalence. It achieves almost the same accuracy results: **79.31%**. In this dataset with the presented framework, there is no degenerate case affecting stability, which is why we obtain the same training curves. It shows that both formulas (17) and (19) can be used when implementing the neural network.
2) *Second Result—Faster Computation Time With Daleckiĭ–Kreĭn/Bhatia Formula:* However, we did observe that the training is faster with Daleckiĭ–Kreĭn/Bhatia formula. Indeed, averaging over five trainings, it takes 90.5 s with this formula compared with 98.5 s with the Ionescu approach, that is an **8% time gain**.
3) *Third Result—Stability Issue With Ionescu Formula:* Depending on the input data, stability can be an issue for backpropagation, as shown through the practical example below. Notably, this may occur when the trial length is small compared with the number of channels. Indeed, estimating covariance matrices on a few temporal samples produces non-full-rank matrices. This induces eigenvalues close to each other, but nonzero since they are regularized by adding a constant 10^{-5}. Focusing on small trials is the principle of EEG micro-states analysis, for example [26]. We train again the SPDNet with both formulas (17) and (19) on the same BCI Competition IV 2A dataset, except that the trial duration is now 40 ms. In that case the covariance matrices have indeed close eigenvalues so that at some point during training, here in Fig. 8 after 33 epochs, the matrix $\mathbf{K}$ (13) contains "Inf" values yielding an error in the gradient computation. That is why the training process stops. One also notes that in this particular degenerate case, Daleckiĭ–Kreĭn/Bhatia formula is slightly better than the Ionescu formula since the learning curves are lower in red than in blue.

One can see that the correction brought in the implementation of Daleckiĭ–Kreĭn/Bhatia formula allows the computation of the gradient and the training of the SPDNet with the described data. Therefore, this shows the superiority of Daleckiĭ–Kreĭn/Bhatia formula that can work for more applications thanks to its stability.

So, as explained, we will prefer Daleckiĭ–Kreĭn/Bhatia formula for its numerical stability and speed.

V. Conclusion

This study aims at proving and comparing two approaches to compute gradients of matrix functions: the Ionescu approach and Daleckiĭ–Kreĭn/Bhatia formula. This self-contained paper identifies flaws in prior works and provides detailed proofs of these two approaches. We first derived an important result by extending Daleckiĭ–Kreĭn/Bhatia formula for the general case of a diagonalizable matrix, yielding a concise formula for the sensitivity. Then we focused on the SPD matrices case and theoretically showed the equivalence between Daleckiĭ–Kreĭn/Bhatia formula and the expression of the sensitivity calculated according to Ionescu's approach. We tested it via a numerical application on an EEG dataset: the independent implementations of the two methods for backpropagation in an SPD matrix neural network called SPDNet. Even though these methods are equivalent, many network implementations are using Daleckiĭ–Kreĭn/Bhatia formula rather than the Ionescu approach, but it was not clear why this choice prevails. So we showed that Daleckiĭ–Kreĭn/Bhatia formula is a preferable choice because of its stability and for computational speed. It then works on more applications, such as EEG micro-states analysis. Daleckiĭ–Kreĭn/Bhatia formula does not only serve for matrix backpropagation but is also an efficient tool to easily calculate matrix gradients as illustrated with some theoretical results. This allows us to affirm that Daleckiĭ–Kreĭn/Bhatia formula is a key tool to calculate gradients of matrix functions in various applications and especially for matrix neural networks. Finally, in order to implement efficient Python libraries for SPD manifold deep learning, Daleckiĭ–Kreĭn/Bhatia formula might be essential for describing the mathematical operations of the layers.

References

[1] J. Taghia, M. Bånkestad, F. Lindsten, and T. Schön. (2020). *Matrix Multilayer Perceptron*. [Online]. Available: https://openreview.net/forum?id=Hye5TaVtDH
[2] J. R. Magnus, "On the concept of matrix derivative," *J. Multivariate Anal.*, vol. 101, no. 9, pp. 2200–2206, Oct. 2010.
[3] A. Barachant, S. Bonnet, M. Congedo, and C. Jutten, "Multiclass brain–computer interface classification by Riemannian geometry," *IEEE Trans. Biomed. Eng.*, vol. 59, no. 4, pp. 920–928, Apr. 2012.

[4] C. Ju and C. Guan, "Tensor-CSPNet: A novel geometric deep learning framework for motor imagery classification," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 12, pp. 10955–10969, Dec. 2023.

[5] I. Carrara, B. Aristimunha, M.-C. Corsi, R. Y. de Camargo, S. Chevallier, and T. Papadopoulo, "Geometric neural network based on phase space for BCI-EEG decoding," *J. Neural Eng.*, vol. 22, no. 1, Feb. 2025, Art. no. 016049, doi: 10.1088/1741-2552/ad88a2.

[6] W.-G. Chang, T. You, S. Seo, S. Kwak, and B. Han, "Domain-specific batch normalization for unsupervised domain adaptation," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*. Red Hook, NY, USA: Curran Associates Inc., Jun. 2019, pp. 7346–7354.

[7] H. T. Gowda and L. M. Miller, "Topology of surface electromyogram signals: Hand gesture decoding on Riemannian manifolds," *J. Neural Eng.*, vol. 21, no. 3, Jun. 2024, Art. no. 036047, doi: 10.1088/1741-2552/ad5107.

[8] C. Ionescu, O. Vantzos, and C. Sminchisescu, "Matrix backpropagation for deep networks with structured layers," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 2965–2973.

[9] E. Sandoval, J. Olmos, and F. Martínez, "A self-supervised deep Riemannian representation to classify parkinsonian fixational patterns," *Artif. Intell. Med.*, vol. 157, Nov. 2024, Art. no. 102987.

[10] O. Tuzel, F. Porikli, and P. Meer, "Region covariance: A fast descriptor for detection and classification," in *Computer Vision—ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds., Berlin, Germany: Springer, 2006, pp. 589–600.

[11] Z. Huang and L. V. Gool, "A Riemannian network for SPD matrix learning," in *Proc. Assoc. Advancement Artif. Intell. (AAAI)*, vol. 31, 2017, pp. 2036–2042.

[12] Z. Chen, T. Xu, X.-J. Wu, R. Wang, Z. Huang, and J. Kittler, "Riemannian local mechanism for SPD neural networks," *Proc. AAAI Conf. Artif. Intell.*, vol. 37, no. 6, pp. 7104–7112, Jun. 2023.

[13] C. Ionescu, O. Vantzos, and C. Sminchisescu, "Training deep networks with structured layers by matrix backpropagation," 2015, *arXiv:1509.07838*.

[14] D. Brooks, O. Schwander, F. Barbaresco, J.-Y. Schneider, and M. Cord, "Riemannian batch normalization for SPD neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 1–11.

[15] Z. Chen, Y. Song, Y. Liu, and N. Sebe, "A lie group approach to Riemannian batch normalization," in *Proc. 12th Int. Conf. Learn. Represent.*, 2024, pp. 1–25. [Online]. Available: https://openreview.net/forum?id=okYdj8Ysru

[16] M. Engin, L. Wang, L. Zhou, and X. Liu, "DeepKSPD: Learning Kernel-matrix-based SPD representation for fine-grained image recognition," in *Computer Vision—ECCV 2018*. Cham, Switzerland: Springer, 2018, pp. 629–645.

[17] G. W. vom Berg et al., "A new canonical log-Euclidean kernel for symmetric positive definite matrices for EEG analysis," *IEEE Trans. Biomed. Eng.*, vol. 72, no. 3, pp. 1000–1007, Oct. 2024.

[18] R. Bhatia, "Matrix analysis," in *Graduate Texts in Mathematics*, vol. 169. New York, NY, USA: Springer, 1996.

[19] V. Noferini, "A formula for the Fréchet derivative of a generalized matrix function," *SIAM J. Matrix Anal. Appl.*, vol. 38, no. 2, pp. 434–457, Jan. 2017, doi: 10.1137/16m1072851.

[20] M. B. Giles, "Collected matrix derivative results for forward and reverse mode algorithmic differentiation," in *Advances in Automatic Differentiation*, C. H. Bischof, H. M. Bücker, P. Hovland, U. Naumann, and J. Utke, Eds., Berlin, Germany: Springer, 2008, pp. 35–44.

[21] J. L. Daleckiĭ and S. G. Kreĭn, "Integration and differentiation of functions of Hermitian operators and applications to the theory of perturbations," in *Thirteen Papers on Functional Analysis and Partial Differential Equations*, vol. 47. Providence, RI, USA: American Mathematical Society Translations: Series 2, 1965, pp. 1–30.

[22] J. Mairal, "Errata on the paper 'end-to-end kernel learning with supervised convolutional kernel networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*. Barcelona, France: Curran Associates, Dec. 2016, pp. 1399–1407. [Online]. Available: https://inria.hal.science/hal-01387399

[23] R. Wang, X.-J. Wu, T. Xu, C. Hu, and J. Kittler, "U-SPDNet: An SPD manifold learning-based neural network for visual classification," *Neural Netw.*, vol. 161, pp. 382–396, Apr. 2023.

[24] M. Tangermann et al., "Review of the BCI competition IV," *Frontiers Neurosci.*, vol. 6, p. 55, Jul. 2012.

[25] *Moabb.datasets.bnci2014001*. Accessed: Jul. 1, 2025. [Online]. Available: https://moabb.neurotechx.com/docs/api.htmlmymargin

[26] S. Pan, T. Shen, Y. Lian, and L. Shi, "A task-related EEG microstate clustering algorithm based on spatial patterns, Riemannian distance, and a deep autoencoder," *Brain Sci.*, vol. 15, no. 1, p. 27, Dec. 2024. [Online]. Available: https://www.mdpi.com/2076-3425/15/1/27