PMaF: Deep Declarative Layers for Principal Matrix Features

Zhiwei Xu [1] **Hao Wang**[†] [1] **Yanbin Liu** [1] **Stephen Gould** [1]

Abstract

We explore two differentiable deep declarative layers, namely least squares on sphere (LESS) and implicit eigen decomposition (IED), for learning principal matrix features (PMaF). It can be used to represent data features with a low-dimensional vector containing dominant information from a high-dimensional matrix. We first solve the problems with iterative optimization in the forward pass and then backpropagate the solution for implicit gradients under a bi-level optimization framework. Particularly, adaptive descent steps with the backtracking line search method and descent decay in the tangent space are studied to improve the forward pass efficiency of LESS. Meanwhile, exploited data structures are used to greatly reduce the computational complexity in the backward pass of LESS and IED. Empirically, we demonstrate the superiority of our layers over the off-the-shelf baselines by comparing the solution optimality and computational requirements.

1. Introduction

Principal matrix feature (PMaF)[‡] in this work refers to a single vector summarising a data matrix. It can be used in deep feature representation or learning that is typical in various areas, such as image analysis (Xu et al., 2014; Melas-Kyriazi et al., 2022), natural language processing (Young et al., 2017), weather prediction (Malakar et al., 2021), and so on. It adapts learned features for downstream tasks and studies matrix structures for fine-grained features with such as a high sparsity or a low dimension (Ranzato et al., 2007; Liu & Yan, 2011; Robles-Kelly, 2016; Liu et al., 2017).

In this work, we mainly focus on two optimization problems

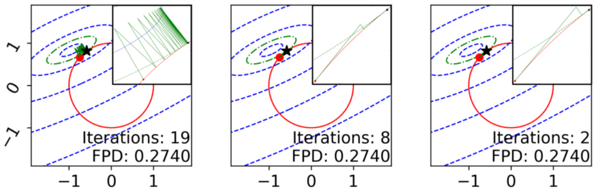


*Figure 1. With BLS and TWD, LESS converges faster and better. The moving path starts from the initial value (**red**-dot) to the optimal (**black**-star). More results are in the Appendix.*

for PMaF and study two deep layers, namely least squares on sphere (LESS) and implicit eigen decomposition (IED), that are superior to off-the-shelf SciPy (nondifferentiable) (Virtanen et al., 2020) and PyTorch (Paszke et al., 2019) baselines in the effectiveness and/or efficiency of optimization and differentiability for end-to-end learning.

In LESS, the proposed adaptive gradient descent steps on the tangent plane greatly reduce the number of iterations, see Fig. 1; in IED, the alternatives, power iteration (PI) (von Mises & Pollaczek-Geiringer, 1929) and simultaneous iteration (SI) (McCormick & Noe, 1977), achieve better solutions for non-negative symmetric and nonsymmetric matrices than the baseline. Meanwhile, we use implicit differentiation methods, mainly deep declarative network (DDN) (Gould et al., 2021) in this work, with exploited matrix structures (Gould et al., 2022) to dramatically reduce the computational requirements. Comprehensive experiments are provided in the Appendix.

2. Deep Declarative Layers

2.1. Least Squares on Sphere

Given $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^{m}$, the least squares problem with solution constrained on a unit sphere is defined as

$$\text{minimize}_{\mathbf{u}\in\mathbb{R}^n}\, \mathtt{f}(\mathbf{A},\mathbf{b},\mathbf{u}) \triangleq \frac{1}{2}\left\|\mathbf{A}\mathbf{u}-\mathbf{b}\right\|^2 \ , \qquad \text{subject to } \left\|\mathbf{u}\right\|^2 = 1 \ , \tag{1}$$

where $\|\cdot\|$ is the ℓ_2-norm. For the notation simplicity, we denote $\mathtt{f}(\mathbf{A},\mathbf{b},\mathbf{u})$ by using $\mathtt{f}(\mathbf{u})$. Since $\mathbf{u}$ is constrained on the sphere, the closed-form of the optimal $\mathbf{y} = \mathbf{A}^{-1}\mathbf{b}$ no longer holds. The projected gradient descent (PGD) method is used to decrease the energy in the gradient descent direction while guaranteeing the solution feasibility.

[†]Parts of this work were completed when Hao Wang pursued a master's degree at the Australian National University. [‡]PMaF for the IED problem is equivalent to principal component analysis (Mackiewicz & Ratajczak, 1993) for a covariance matrix. [1]School of Computing, CECC, ANU, Canberra, Australia. Correspondence to: Zhiwei Xu <zhiwei.xu@anu.edu.au>.

Published at the Differentiable Almost Everything Workshop of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. July 2023. Copyright 2023 by the author(s).

We first describe the vanilla PGD, PGD with direction weight decay, and PGD projected onto the Riemannian manifold (directly on the constraint sphere). For monotonic energy convergence, either the backtracking line search method or a simple yet effective step decay in the tangent space is used for fast optimal solution search.

1) ***Projected Gradient Descent (PGD).*** We refer to the Appendix for the details of the widely used PGD method (Akilov & Kantorovich, 1982; Lemarechal, 2012).

2) ***Direction Weight (DW).*** Since the descent step η in PGD needs to decrease for a fine search when $\mathbf{u}_t$ approaches the optimal solution, reducing η with weight w_t adaptive to the descent direction is desirable. Given $\mathtt{f}(\mathbf{u})$ in Eq. (1), the center $\mathbf{u}_0 = (\mathbf{A}^\top \mathbf{A})^{-1}\mathbf{A}^\top \mathbf{b}$ indicates if the least squares problem is an inner (non-convex) or outer (convex) equality-constrained problem and also the direction from the constraint center to the solution.

With the optimal unconstrained solution, $\mathbf{u}_0$ is parallel to the descent direction $\mathbf{d}_t$. Hence, we define a direction weight for each descent step as

$$w_t = 1 - \mathtt{S_c}(\mathbf{d}_t, \mathbf{u}_0) \ , \tag{2}$$

where

$$\mathbf{d}_t = \begin{cases} -\nabla \mathtt{f}(\mathbf{u}_t) & \text{if } \|\mathbf{u}_0\| \geq 1 \\ \nabla \mathtt{f}(\mathbf{u}_t) & \text{otherwise} \end{cases} \tag{3}$$

and $\mathtt{S_c}(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\top \mathbf{b} / (\|\mathbf{a}\| \, \|\mathbf{b}\|)$ is the cosine similarity to measure the direction homotropy. When $\mathbf{u}_t$ is optimal, $w_t = 0$ such that $w_t \eta = 0$ terminates the update of $\mathbf{u}_t$.

3) ***PGD on Riemannian Manifold (RM).*** As the solution is constrained on a sphere, using Riemannian geodesic distance is promising to learn better feature distribution than using the Euclidean distance (Barachant et al., 2010; Wang et al., 2017). Therefore, we define the solution projection onto the Riemannian manifold (Boumal, 2020) as

$$\mathtt{Proj_{RM}}\left(-\nabla \mathtt{f}(\mathbf{u}_t)\right) = \left(\mathbf{I}_n - \mathbf{u}_t \mathbf{u}_t^\top\right)\left(-\nabla \mathtt{f}(\mathbf{u}_t)\right) \ , \tag{4}$$

where $\mathbf{I}_n$ is an $n \times n$ identity matrix. The solution update at $(t+1)$ follows

$$\mathbf{u}_{t+1} = \mathtt{Proj_{Sph}}\left(\mathbf{u}_t + \eta \mathtt{Proj_{RM}}\left(-\nabla \mathtt{f}(\mathbf{u}_t)\right)\right) \ , \tag{5}$$

where $\eta = 1$ or can be derived from $\nabla \mathtt{f}(\mathbf{u}_{t+1}) = 0$ as

$$\eta = \frac{(\mathbf{A}\mathbf{u}_t - \mathbf{b})^\top \mathbf{A}\left(\mathbf{I}_n - \mathbf{u}_t \mathbf{u}_t^\top\right)\mathbf{A}^\top (\mathbf{A}\mathbf{u}_t - \mathbf{b})}{\left\|\mathbf{A}(\mathbf{I}_n - \mathbf{u}_t\mathbf{u}_t^\top)\mathbf{A}^\top(\mathbf{A}\mathbf{u}_t - \mathbf{b})\right\|^2} \ . \tag{6}$$

See the Appendix for the Riemannian manifold projection.

4) ***Backtracking Line Search (BLS).*** While the direction weight method causes many iterations if the solution is far from optimal, a guarantee of energy reduction in every iteration is crucial, requiring a suitable descent step. For this to hold, we refer to the backtracking line search method (Boyd & Vandenberghe, 2004) and apply the first-order Lagrangian form of Eq. (1)

$$\mathtt{f}(\mathbf{u}_t + \eta \Delta \mathbf{u}_t) \leq \mathtt{f}(\mathbf{u}_t) + \alpha \eta \nabla^\top \mathtt{f}(\mathbf{u}_t) \Delta \mathbf{u}_t \ , \tag{7}$$

where constant $\alpha \in (0, 0.5)$ is for the maximum energy decrease at the $(t+1)^{\text{th}}$ iteration for the descent monotonicity. Otherwise, the descent step would be decayed by $\eta \leftarrow \beta\eta$ with $\beta \in (0,1)$ to avoid surpassing the optimal solution.

5) ***Tangent Weight Decay (TWD).*** Alternatively, since a large descent step tackles the monotonic energy decrease in the update around the optimal solution, it could be unable to converge on the manifold. Hence, the descent step is decayed in the tangent space on the same side of the descent direction with decay rate β, that is $\eta \leftarrow \beta\eta$, when

$$\mathtt{S_c}\left(\mathtt{Proj_{RM}}\left(-\nabla \mathtt{f}(\mathbf{u}_t)\right), \mathtt{Proj_{RM}}\left(-\nabla \mathtt{f}(\mathbf{u}_{t+1})\right)\right) < 0 \ , \tag{8}$$

with

$$\mathbf{u}_{t+1} = \mathbf{u}_t + \eta \mathtt{Proj_{RM}}\left(-\nabla \mathtt{f}(\mathbf{u}_t)\right) \ . \tag{9}$$

In short, Eq. (8) indicates that the solution update at the $(t+1)^{\text{th}}$ iteration causes a "reverse" descent direction, usually in $(90, 180]$ degrees, around the optimal solution, and thus, a smaller step is preferred.

2.2. Implicit Eigen Decomposition

Generally, the eigen decomposition problem can be formulated by solving the optimization $\mathtt{f} : \mathbf{A} \in \mathbb{R}^{m \times m} \rightarrow \{\boldsymbol{\lambda}, \mathbf{u}\}$ with $\boldsymbol{\lambda}$ as a vector of n largest eigenvalues and $\mathbf{u} \in \mathbb{R}^{m \times n}$ as the corresponding eigenvectors,

$$\begin{aligned} &\text{minimize}_{\mathbf{u} \in \mathbb{R}^{m \times n}} \, \mathtt{f}(\mathbf{A}, \mathbf{u}) \triangleq -\mathtt{tr}\left(\mathbf{u}^\top \mathbf{A} \mathbf{u}\right) \ , \\ &\text{subject to} \quad \mathtt{h}(\mathbf{u}) \triangleq \mathbf{u}^\top \mathbf{u} = \mathbf{I}_n \ , \end{aligned} \tag{10}$$

where $\mathtt{tr}()$ is the trace function over all n eigenvalues. The optimal solution in Eq. (11) satisfies Eq. (12) as

$$\mathbf{y} = \mathtt{argmin}_{\mathbf{u} \in \mathbb{R}^{m \times n}} \mathtt{f}(\mathbf{A}, \mathbf{u}) \ , \tag{11}$$

$$\mathbf{A}\mathbf{y}_i = \lambda_i \mathbf{y}_i, \quad \forall i \in \mathcal{N} = \{1, ..., n\} \ . \tag{12}$$

The principal matrix component refers to the eigenvector associated with the largest eigenvalue, for which we use the power iteration algorithm and the simultaneous iteration algorithm. For a complete analysis, solution update formulas are provided below.

Solver 1. ***Power Iteration (PI).*** Given a randomly initialized eigenvector $\mathbf{u}_0$, the solution update at time $(t+1)$ follows

$$\mathbf{u}_{t+1} = \mathbf{A}\mathbf{u}_t / \|\mathbf{A}\mathbf{u}_t\| \ , \tag{13}$$

and terminates (upon the convergence or the maximum iteration) at $t = K$ for the principal eigenvector and eigenvalue,

$$\mathbf{y} = \mathbf{u}_K \quad \text{and} \quad \lambda = \mathbf{y}^\top \mathbf{A} \mathbf{y} \ . \tag{14}$$

Solver 2. ***Simultaneous Iteration (SI).*** QR decomposition is required for iterative updates of the input and the solution. The initial input $\mathbf{x}_0 = \mathbf{A}$ and the updates at time t follow

$$\{\mathbf{Q}_t, \mathbf{R}_t\} = \texttt{QR}(\mathbf{x}_t) \quad \text{and} \quad \mathbf{x}_{t+1} = \mathbf{x}_t\mathbf{Q}_t \ . \tag{15}$$

The principal eigenvector is the component of $\mathbf{Q}_K$ corresponding to the largest eigenvalue $\lambda = \texttt{max}(\mathbf{R}_K)$.

Solution consistency for effective backpropagation. The eigen decomposition problem defined in Eq. (10) has two optimal solutions with reverse directions due to the quadratics. This could happen either in the intermediately updated solution $\mathbf{u}_t$ (either Eq. (13) or Eq. (15)) or in the optimal solution $\mathbf{y}$ in each learning epoch for the same data sample. To alleviate the ineffectiveness of these updates, one can apply a reference direction $\mathbf{r}$ such that the optimal solution $\mathbf{y}$ updates in the same direction of $\mathbf{r}$ as

$$\textit{Historical}: \quad \mathbf{u}_t \leftarrow \texttt{V}\left(\mathbf{u}_t, \mathbf{u}_{t-1}\right)\mathbf{u}_t \ , \tag{16}$$
$$\textit{Hard-coded}: \quad \mathbf{y} \leftarrow \texttt{V}\left(\mathbf{y}, \mathbf{r}\right)\mathbf{y} \ , \tag{17}$$

where $\texttt{V}(\mathbf{a}, \mathbf{b}) = \texttt{Sign}\left(\mathbf{a}^\top\mathbf{b}\right)$ if $(\mathbf{a} \not\perp \mathbf{b})$ and otherwise 1 and $\texttt{Sign}()$ calculates the sign value of a scalar. If $\mathbf{u}_t$ is for eigenvectors of multiple eigenvalues, only the diagonals of $\mathbf{u}_t^\top\mathbf{u}_{t-1}$ are used for the sign values.

3. Implicit Differentiation

Given the optimal solutions, their gradients over the input entries enable end-to-end learning. Without unrolling the iterations of the solvers, we use a single-step method with implicit differentiation (Gould et al., 2021), fixed-point theorem (for IED), and exploited structures (Gould et al., 2022).

3.1. Deep Declarative Networks based Gradients

With the iterative optimization as the forward pass, the backward pass of $\mathbf{y}$ to $\mathbf{A}$ in Eq. (1) and Eq. (10) is required. Backtracking the forward pass, however, is inefficient and sometimes infeasible due to the discrete solution. Hence, we use the deep declarative network method (Gould et al., 2021) to efficiently calculate $\nabla_X L$ with $\nabla_X \mathbf{y}$, where L is the loss from the upper problem in the bi-level optimization and X and Y indicate the input variable (can be multiple) and the optimization solution respectively. It follows

$$\mathcal{K} = \nabla_Y L\left(\mathcal{H}^{-1}\mathcal{A}^\top\left(\mathcal{A}\mathcal{H}^{-1}\mathcal{A}^\top\right)^{-1}\mathcal{A} - \mathbf{I}_n\right)\mathcal{H}^{-1} \ , \tag{18}$$
$$\nabla_X L = \mathcal{K}\mathcal{B} \ . \tag{19}$$

1) ***LESS.*** The components of $\mathcal{K}$ in Eq. (18) are

$$\mathcal{A} = 2\mathbf{y}^\top \in \mathbb{R}^{1\times n} \ , \tag{20}$$
$$\mathcal{B} = \nabla^2_{XY}\texttt{f}(\mathbf{A}, \mathbf{b}, \mathbf{y}) \in \mathbb{R}^{n\times(m\times n)} \ , \tag{21}$$
$$\mathcal{H} = \mathbf{A}^\top\mathbf{A} - 2\beta\mathbf{I}_n \in \mathbb{R}^{n\times n} \ , \tag{22}$$
$$\beta = \frac{1}{2}\mathbf{y}^\top\mathbf{A}^\top\left(\mathbf{A}\mathbf{y} - \mathbf{b}\right) \in \mathbb{R} \ . \tag{23}$$

Proposition 3.1. *(Exploited Hessian structure for LESS) Rather than applying Jacobian and Hessian operations, an accumulation by parts approach with the exploited structure of $\mathcal{B}$ greatly improves the implementation efficiency. Recall that $\mathcal{B} \in \mathbb{R}^{n\times(m\times n)}$, $\mathbf{A} \in \mathbb{R}^{m\times n}$, and $\mathbf{y} \in \mathbb{R}^n$. Set indices $i, j \in \mathcal{N} = \{1, ..., n\}$ for the dimensions related to n such that $\mathcal{B} = [\mathcal{B}_{ij}]$, $\mathbf{A} = [\mathbf{A}_i]$, and $\mathbf{y} = [\mathbf{y}_j]$. Then,*

$$\boxed{\begin{aligned} &\mathcal{B}_{ij} = \mathbf{A}_i\mathbf{y}_j^\top, \quad \forall i, j \in \mathcal{N} \ , && (24)\\ &\mathcal{B}_{ii} \leftarrow \mathcal{B}_{ii} + (\mathbf{A}\mathbf{y} - \mathbf{b}), \quad \forall i \in \mathcal{N} \ . && (25)\end{aligned}}$$

Proof Sketch. See the Appendix for details. □

2) ***IED.*** Similarly, for IED, one has

$$\mathcal{A} = 2\mathbf{y}^\top \in \mathbb{R}^{1\times m} \ , \tag{26}$$
$$\mathcal{B} = \nabla^2_{XY}\texttt{f}(\mathbf{A}, \mathbf{y}) \in \mathbb{R}^{m\times(m\times m)} \ , \tag{27}$$
$$\mathcal{H} = -\left(\mathbf{A} + \mathbf{A}^\top\right) - 2\beta\mathbf{I}_m \in \mathbb{R}^{m\times m} \ , \tag{28}$$
$$\beta = -\frac{1}{2}\mathbf{y}^\top\left(\mathbf{A}\mathbf{y} + \mathbf{A}^\top\mathbf{y}\right) \in \mathbb{R} \ . \tag{29}$$

Proposition 3.2. *If the input $\mathbf{A}$ of the objective function Eq. (10) is symmetric, the Lagrange multiplier equals the negative largest eigenvalue, that is, $\beta = -\lambda$.*

Proof. Since $\nabla_Y\texttt{f}(\mathbf{A}, \mathbf{y}) = \beta\nabla_Y\texttt{h}(\mathbf{A}, \mathbf{y})$ in Eq. (17) of (Gould et al., 2021) for the minima, $\nabla_Y\texttt{f}(\mathbf{A}, \mathbf{y}) = -(\mathbf{A}\mathbf{y} + \mathbf{A}^\top\mathbf{y})^\top$ and $\nabla_Y\texttt{h}(\mathbf{A}, \mathbf{y}) = 2\mathbf{y}^\top$ for the problem defined in Eq. (10), and $\mathbf{A}\mathbf{y} = \lambda\mathbf{y}$, one has $-(\lambda\mathbf{y} + \mathbf{A}^\top\mathbf{y}) = 2\beta\mathbf{y}$. If $\mathbf{A}^\top = \mathbf{A}$, then $\mathbf{A}^\top\mathbf{y} = \mathbf{A}\mathbf{y} = \lambda\mathbf{y}$, and thus, $\beta = -\lambda$. □

Proposition 3.3. *(Exploited Hessian structure for IED) Recall that $\mathcal{B} \in \mathbb{R}^{m\times(m\times m)}$ and $\mathbf{y} \in \mathbb{R}^m$. Set indices $i, j, k \in \mathcal{M} = \{1, ..., m\}$ for the dimensions related to m such that $\mathcal{B} = [\mathcal{B}_{ijk}]$ and $\mathbf{y} = [\mathbf{y}_i]$. Then,*

$$\boxed{\begin{aligned} &\mathcal{B}_{ijk} = 0, \quad \forall i, j, k \in \mathcal{M} \ , && (30)\\ &\mathcal{B}_{iji} \leftarrow \mathcal{B}_{iji} - \mathbf{y}_j, \quad \forall i, j \in \mathcal{M} \ , && (31)\\ &\mathcal{B}_{iij} \leftarrow \mathcal{B}_{iij} - \mathbf{y}_j, \quad \forall i, j \in \mathcal{M} \ . && (32)\end{aligned}}$$

Proof Sketch. This can be easily proved by finding the structure of $\mathcal{B}$ from an example. □

Nevertheless, $\mathcal{B}$ requires $(4m^3/1024^2)$MB memory in the single-precision floating-point format, for instance, 4,096MB for $m = 1024$. With the exploited Hessian structure of $\mathcal{B}$ and the vector-Jacobian product for $\nabla_X L$, however, $\mathcal{B}$ can be avoided by using merely $\mathbf{y}$.

Proposition 3.4. *(Exploited gradient structure for IED) $\mathcal{B}$ can be split into $\mathbf{y}$ dependencies for $\nabla_X L \in \mathbb{R}^{m\times m}$ as*

$$\boxed{\nabla_X L = -\mathcal{K}\mathbf{y}^\top - \mathbf{y}\mathcal{K}^\top \ .} \tag{33}$$

Proof. See the Appendix for details. □

3.2. Implicit Function Theorem based Gradients

The implicit function theorem based gradients only apply to eigen decomposition. By using PI for the largest eigenvalue (to be positive), the implicit function is

$$\mathtt{f}(\mathbf{A}, \mathbf{u}_t, \mathbf{u}_{t+1}) = \mathbf{u}_{t+1} - \mathbf{A}\mathbf{u}_t / \|\mathbf{A}\mathbf{u}_t\| \tag{34}$$

with the input matrix $\mathbf{A} \in \mathbb{R}^{m\times m}$ and the solution $\mathbf{u} \in \mathbb{R}^{m\times n}$. Upon the convergence, $\mathbf{y} = \mathbf{u}_{t+1} = \mathbf{u}_t$, and thus,

$$\mathtt{f}(\mathbf{A}, \mathbf{y}) = \mathbf{y} - \mathbf{A}\mathbf{y} / \|\mathbf{A}\mathbf{y}\| \ . \tag{35}$$

Applying the implicit function theorem to $\mathtt{f}(\mathbf{A}, \mathbf{y})$ achieves

$$\nabla_X \mathtt{f}(\mathbf{A}, \mathbf{y}) + \nabla_Y \mathtt{f}(\mathbf{A}, \mathbf{y}) \nabla_X \mathbf{y} = 0 \ , \tag{36}$$

$$\nabla_X \mathbf{y} = -\left(\nabla_Y \mathtt{f}(\mathbf{A}, \mathbf{y})\right)^{-1} \nabla_X \mathtt{f}(\mathbf{A}, \mathbf{y}) \ . \tag{37}$$

For the notation consistency with Sec. 3.1, we denote $\mathcal{H} = \nabla_Y \mathtt{f}(\mathbf{A}, \mathbf{y}) \in \mathbb{R}^{m\times m}$, $\mathcal{B} = \nabla_X \mathtt{f}(\mathbf{A}, \mathbf{y}) \in \mathbb{R}^{m\times(m\times m)}$, and $\nabla_X \mathbf{y} = -\mathcal{H}^{-1}\mathcal{B} \in \mathbb{R}^{m\times(m\times m)}$. Note that $\nabla_X \mathbf{y}$ is in the same form of Eq. (19) as an unconstrained problem. Then, $\nabla_Y L \in \mathbb{R}^m$ follows

$$\nabla_X L = \nabla_Y L \nabla_X \mathbf{y} = -\nabla_Y L \mathcal{H}^{-1} \mathcal{B} \ , \tag{38}$$

$$\boxed{\begin{aligned} \mathcal{H} &= \mathbf{I}_m - \left(\mathbf{I}_m - \mathbf{y}\mathbf{y}^\top\right) \mathbf{A}/\lambda \ , & (39) \\ \mathcal{B} &= -\left(\mathbf{I}_m - \mathbf{y}\mathbf{y}^\top\right) \mathbf{y}^\top/\lambda \ . & (40) \end{aligned}}$$

Different from using multiple eigenvalues for gradients (Magnus & Neudecker, 2007), only the largest one is used. Again, it is strongly encouraged to use the vector-Jacobian product to calculate $\nabla_X L$ over $\mathcal{B}$ for memory reduction.

4. Experiments

Experiments are on a 3090 GPU and PyTorch 1.12.0. The code is at https://github.com/anucvml/ddn.git. See the Appendix for **evaluation standards** including fixed-point distance (FPD) in Eq. (41), eigen distance in Eq. (43), and mean relative error (MRE) in Eq. (44). We highlight ours .

4.1. Evaluation on LESS. We show the ***effectiveness*** in Fig. 1 and Table 1 and ***efficiency*** in Table 2. Fig. 1 shows a case where ours can achieve fewer iterative steps when optimizing Eq. (1). In Table 1, ours with BLSOne and TWD achieve comparable low MRE with the SciPy solver (Virtanen et al., 2020) (supports only CPU and not differentiable) and outperform the vanilla PGD and PGD+RM. With exploited structures including Prop. 3.1, ours is faster and more memory efficient than without those structures ("AutoDiff"). More results are in the Appendix.

4.2. Evaluation on IED. We show the ***effectiveness*** in Fig. 2 and ***efficiency*** in Table 3. In Fig. 2, PI and SI achieve lower eigen distance and FPD than PyTorch `eigh()` ("AutoDiff") (Paszke et al., 2019). With exploited structures in Table 3, ours achieve higher speed or less memory without ignoring their effectiveness. More results are in the Appendix.

Table 1. With 1,000 random data from $\mathcal{N}(0, 1)$. "In" and "Out" refer to the number of failed cases inner and outer of the sphere respectively. "Imp." is the number of cases with FPD $\leq$ the SciPy baseline. See the Appendix for more results.

Method	Size 2×2				Size 64×32			
	In↓	Out↓	Imp.↑	MRE↓	In↓	Out↓	Imp.↑	MRE↓
PGD	172	353	389	10.84	548	440	52	0.15
+RM	**0**	118	844	4.30	30	**0**	980	**0.00**
+RM+BLSOne	**0**	14	**938**	0.14	**6**	**0**	**994**	**0.00**
+RM+TWD	**0**	**0**	806	**0.00**	29	**0**	954	**0.00**

Table 2. Backward speedup with exploited structures. Numbers are averaged over 100 samples. "PGD+RM+TWD" is used for LESS. "Speedup" is ("AutoDiff"-"LESS")/"LESS".

	Size 256 ×8×64		Size 256×16×128	
	Time (s)	Memory (MB)	Time (s)	Memory (MB)
AutoDiff	4.44	65.22	8.64	519.32
LESS	0.01	49.20	0.02	325.70
Speedup	×444↑	×0.33↑	×432↑	×0.59↑

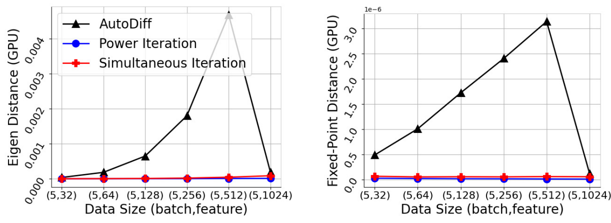


Figure 2. Precision evaluation with symmetric $\mathbf{A}$ in float32. See the Appendix for nonsymmetric $\mathbf{A}$ and simulation on ResNet50.

Table 3. Efficiency evaluation with symmetric $\mathbf{A}$ and solution size 5×512. "J": autodiff Jacobian without exploited structures; "E": our differentiation with exploited structures. See the Appendix for results with nonsymmetric $\mathbf{A}$ and more sizes.

Method	GPU Time (s)		GPU Memory (MB)	
	Forward	Backward	Forward	Backward
AutoDiff	0.0758	0.0005	20.56	24.99
PI-DDN-J	0.0069	0.0436	5.55	2570.06
PI-IFT-J	0.0070	0.6942	5.55	5125.00
PI-DDN-E	0.0069	0.0114	5.55	16.01
PI-IFT-E	0.0069	0.0043	5.55	31.01
SI-DDN-J	1.0623	0.0419	32.02	2570.06
SI-IFT-J	1.0624	0.7120	32.02	5125.00
SI-DDN-E	1.0622	0.0113	32.02	16.01
SI-IFT-E	1.0625	0.0042	32.02	31.01

5. Conclusion

We explore two deep layers, LESS and IED, for PMaF that can be used in end-to-end learning. Either optimization improvements or alternatives enhance the optimization effectiveness and/or efficiency. We also enable the differentiation feasibility and largely reduce computational requirements using DDN and IFT based gradients with exploited structures. Extensive experiments show the benefits of our layers over the baseline from SciPy optimizer or PyTorch function.

References

Akilov, G. P. and Kantorovich, L. V. Functional analysis (2nd edition). In *Pergamon Press*, 1982.

Barachant, A., Bonnet, S., Congedo, M., and Jutten, C. Riemannian geometry applied to BCI classification. In *nternational Conference on Latent Variable Analysisand Signal Separation*, 2010.

Boumal, N. An introduction to optimization on smooth manifolds. http://www.nicolasboumal.net/book, Aug 2020.

Boyd, S. and Vandenberghe, L. *Convex optimization*. Cambridge University Press, 2004.

Gould, S., Hartley, R., and Campbell, D. Deep declarative networks. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

Gould, S., Campbell, D., Ben-Shabat, I., Koneputugodage, C. H., and Xu, Z. Exploiting problem structure in deep declarative networks: Two case studies. In *AAAI Workshop on Optimal Transport and Structured Data Modeling*, 2022.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. In *Springer Science and Business Media LLC*, volume 585, pp. 357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

Lemarechal, C. Cauchy and the gradient method. In *Doc Math Extra*, 2012.

Liu, G. and Yan, S. Latent low-rank representation for subspace segmentationand feature extraction. In *IEEE International Conference on Computer Vision*, 2011.

Liu, J., Wu, Z., Xiao, Z., and Yang, J. Classification of hyperspectral images using kernel fully constrained least squares. In *International Journal of Geo-Information*, 2017.

Mackiewicz, A. and Ratajczak, W. Principal components analysis (PCA). In *Computers and Geosciences*, 1993.

Magnus, J. R. and Neudecker, H. Matrix differential calculus with applications in statistics and econometrics (3rd edition). In *John Wiley and Sons Ltd*, 2007.

Malakar, S., Goswami, S., Ganguli, B., Chakrabarti, A., Roy, S. S., Boopathi, K., and Rangaraj, A. G. A novel feature representation for prediction of global horizontal irradiance using a bidirectional model. In *Machine Learning and Knowledge Extraction*, 2021.

McCormick, S. F. and Noe, T. D. Simultaneous iteration for the matrix eigenvalue problem. In *Linear Algebra and its Applications*, 1977.

Melas-Kyriazi, L., Rupprecht, C., Laina, I., and Vedaldi, A. Deep spectral methods: A surprisingly strong baseline for unsupervised semantic segmentation and localization. In *IEEE International Conference on Computer Vision*, 2022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019. URL https://pytorch.org/docs/stable/generated/torch.linalg.eigh.html.

Ranzato, M., Boureau, Y. L., and LeCun, Y. Sparse feature learning for deep belief networks. In *Conference on Neural Information Processing Systems*, 2007.

Robles-Kelly, A. Least-squares regression with unitary constraints for network behaviour classifications. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition and Structural and Syntactic Pattern Recognition*, 2016.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. In *Nature Methods*, 2020. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html.

von Mises, R. Mathematical theory of probability and statistics. In *New York: Academic Press*, 1964.

von Mises, R. and Pollaczek-Geiringer, H. Praktische verfahren der gleichungsauflosung. In *Zeitschrift fur Angewandte Mathematik und Mechanik*, 1929.

Wang, Y., Li, W., Dai, D., and Gool, L. V. Deep domain adaptation by geodesic distance minimization. In *IEEE International Conference on Computer Vision*, 2017.

Xu, Y., Mo, T., Feng, Q., Zhong, P., Lai, M., and Chang, E. I.-C. Deep learning of feature representation with multiple instancelearning for medical image analysis. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2014.

Young, T., Hazarika, D., Poria, S., and Cambria, E. Recent trends in deep learning based natural language processing. In *arXiv:1708.02709*, 2017.

Appendix

A. Evaluation Standards

1) Fixed Point Distance (FPD). This measures the distance of the estimated solution to the minimum of the objective function, where the minimum objective value is unnecessary to be zero given problem constraints.

$$\textit{LESS}: \quad \mathtt{F}(\mathbf{y}) = \|\mathbf{A}\mathbf{y} - \mathbf{b}\| \ , \tag{41}$$

$$\textit{IED}: \quad \mathtt{F}(\mathbf{y}) = \left\| \mathbf{y} - \frac{\mathbf{A}\mathbf{y}}{\|\mathbf{A}\mathbf{y}\|} \right\| \ . \tag{42}$$

2) Eigen Distance. This is intuitive from Eq. (12) to evaluate the eigen solution,

$$D(\mathbf{y}) = \|\mathbf{A}\mathbf{y} - \lambda\mathbf{y}\| \ . \tag{43}$$

3) Mean Relative Error (MRE). Given a reference (generally ground truth) $\mathbf{y}^r$ and k^{th} sample, the mean relative error is

$$\mathtt{MRE}(\mathbf{y}, \mathbf{y}^r) = \frac{1}{K} \sum_{k=1}^{K} \frac{\mathtt{F}(\mathbf{y}_k) - \mathtt{F}(\mathbf{y}_k^r)}{\mathtt{F}(\mathbf{y}_k^r)} \times 100\% \ , \tag{44}$$

which can be negative when the estimation $\mathbf{y}$ outperforms the reference $\mathbf{y}^r$.

B. Projected Gradient Descent and Riemannian Manifold Projection

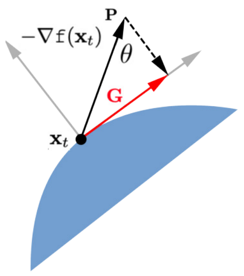


Figure 3. Descent direction projected onto a Riemannian manifold for the sphere constraints. $-\nabla\mathtt{f}(\mathbf{x}_t)$ is the PGD direction, $\mathbf{G} = \textit{Proj}_{\textit{RM}}(-\nabla\mathtt{f}(\mathbf{x}_t))$ is the descent direction on the Riemannian manifold, and $\mathbf{x}_t$ is on the Riemannian manifold.

B.1. Projected Gradient Descent

For ease of optimization, the solution is initialized with the least squares solution without constraints as

$$\mathbf{u}_0 = \left(\mathbf{A}^\top\mathbf{A}\right)^{-1}\mathbf{A}^\top\mathbf{b} \ , \tag{45}$$

$$\mathbf{u}_0 \leftarrow \mathtt{Proj}_{\mathtt{Sph}}(\mathbf{u}_0) \ . \tag{46}$$

Then, the solution is iteratively updated in the gradient descent direction $\Delta\mathbf{u}_t$ at t^{th} iteration by following

$$\mathbf{u}_{t+1} = \mathbf{u}_t + \eta\Delta\mathbf{u}_t \ , \tag{47}$$

$$\mathbf{u}_{t+1} \leftarrow \mathtt{Proj}_{\mathtt{Sph}}(\mathbf{u}_{t+1}) \tag{48}$$

with

$$\Delta\mathbf{u}_t = -\nabla\mathtt{f}(\mathbf{u}_t) \ , \quad \eta = \frac{\left\|\mathbf{A}^\top(\mathbf{A}\mathbf{u}_t - \mathbf{b})\right\|^2}{\left\|\mathbf{A}\mathbf{A}^\top(\mathbf{A}\mathbf{u}_t - \mathbf{b})\right\|^2} \ , \tag{49}$$

where η is derived by setting $\nabla\mathtt{f}(\eta) \equiv \nabla\mathtt{f}(\mathbf{u}_{t+1}) = 0$ and $\mathtt{Proj}_{\mathtt{Sph}}(\mathbf{u}) = \mathbf{u}/\|\mathbf{u}\|$ is the projection onto the unit sphere. The solution update terminates when the objective value decrease is no greater than a predefined tolerance, $1.0e^{-7}$ in our case, or when it reaches the maximum number of iterations, 100 in our case.

B.2. Riemannian Manifold Projection

Since the solution is constrained on the sphere, a simple projection onto the sphere via gradient descent is inefficient considering the inactive orthogonal descent component. In Fig. 3, considering only the descent component on the tangent plane benefits the solution optimality and optimization process, comparing "PGD" and "PGD+RM" in Table 4.

$$\mathbf{P} = \frac{-\mathbf{x}_t}{\|\mathbf{x}_t\|} \|\nabla \mathtt{f}(\mathbf{x}_t)\| \cos\theta = \frac{-\mathbf{x}_t}{\|\mathbf{x}_t\|} \|\nabla \mathtt{f}(\mathbf{x}_t)\| \frac{\nabla \mathtt{f}(\mathbf{x}_t) \frac{-\mathbf{x}_t}{\|\mathbf{x}_t\|}}{\|\nabla \mathtt{f}(\mathbf{x}_t)\| \left\| \frac{\mathbf{x}_t}{\|\mathbf{x}_t\|} \right\|} = \frac{\mathbf{x}_t \mathbf{x}_t^\top}{\|\mathbf{x}_t\|^2} \nabla \mathtt{f}(\mathbf{x}_t) \,, \tag{50}$$

then

$$\mathbf{G} = -\nabla \mathtt{f}(\mathbf{x}_t) + \mathbf{P} = -\nabla \mathtt{f}(\mathbf{x}_t) + \frac{\mathbf{x}_t \mathbf{x}_t^\top}{\|\mathbf{x}_t\|^2} \nabla \mathtt{f}(\mathbf{x}_t) \,. \tag{51}$$

Hence,

$$\mathtt{Proj}_{\mathtt{RM}}(-\nabla \mathtt{f}(\mathbf{x}_t)) = \left(\mathbf{I}_n - \frac{\mathbf{x}_t \mathbf{x}_t^\top}{\|\mathbf{x}_t\|^2} \right) (-\nabla \mathtt{f}(\mathbf{x}_t)) \,, \tag{52}$$

where $\mathbf{x}_t \in \mathbb{R}^n$, $\nabla \mathtt{f}(\mathbf{x}_t) \in \mathbb{R}^n$, $\mathbf{I}_n$ is an $n \times n$ identity matrix.

C. Gradient Transformation with Flipped Eigenvalues

For different orderings of multiple eigenvalues, for instance, descending-order (concerning the order of eigenvalues) versus ascending-order eigenvalues, the gradients of their associated eigenvectors need to be flipped according to the flipping of the eigenvalues. Recall the eigen decomposition problem defined in Eq. (10). Reserve all of the eigenvalues of $\mathbf{A}$, that is $n = m$, a flipping function is defined as $\mathtt{g}(\mathbf{x}) : \mathbb{R}^{m \times m} \to \mathbb{R}^{m \times m}$, that is

$$\mathtt{g}(\mathbf{x}) = \begin{bmatrix} 0 & \cdots & 1 \\ \vdots & \iddots & \vdots \\ 1 & \cdots & 0 \end{bmatrix} \mathbf{x} \begin{bmatrix} 0 & \cdots & 1 \\ \vdots & \iddots & \vdots \\ 1 & \cdots & 0 \end{bmatrix} = \mathbf{I}_m^* \mathbf{x} \mathbf{I}_m^* \,, \tag{53}$$

where $(\mathbf{I}_n^* \mathbf{x})$ flips $\mathbf{x}$ in the vertical direction and $(\mathbf{x}\mathbf{I}_n^*)$ in the horizontal direction. For the notation simplicity, we denote $\mathtt{g}(\mathbf{x}) = \mathbf{x}^{\text{flip}}$. Then,

$$\mathbf{A}\mathbf{u} = \boldsymbol{\Lambda}\mathbf{u} = \mathbf{I}_m^{\text{flip}} \boldsymbol{\Lambda}^{\text{flip}} \mathbf{I}_m^{\text{flip}} \mathbf{u} \,, \tag{54}$$

$$\mathbf{I}_m^{\text{flip}} \mathbf{A}\mathbf{u} = \boldsymbol{\Lambda}^{\text{flip}} \mathbf{I}_m^{\text{flip}} \mathbf{u} \,, \tag{55}$$

$$\mathbf{I}_m^{\text{flip}} \mathbf{A} \left(\mathbf{I}_m^{\text{flip}} \mathbf{u}^{\text{flip}} \mathbf{I}_m^{\text{flip}} \right) = \boldsymbol{\Lambda}^{\text{flip}} \mathbf{u}^{\text{flip}} \mathbf{I}_m^{\text{flip}} \,, \tag{56}$$

$$\mathbf{A}^{\text{flip}} \mathbf{u}^{\text{flip}} = \boldsymbol{\Lambda}^{\text{flip}} \mathbf{u}^{\text{flip}} \,, \tag{57}$$

where the input $\mathbf{A} \in \mathbb{R}^{m \times m}$, eigenvectors $\mathbf{u} \in \mathbb{R}^{m \times m}$, and eigenvalue matrix $\boldsymbol{\Lambda} \in \mathbb{R}^{m \times m}$. It indicates that to obtain the reverse-order eigenvalues, $\mathbf{A}$ needs to be flipped, which also leads to the flipping of $\mathbf{u}$. Therefore, to compare the descending-order eigen decomposition with an ascending-order one, both $\mathbf{A}$ and $\mathbf{u}$ need to be flipped.

For the gradient of loss L over the flipped eigenvectors $\mathbf{u}^{\text{flip}}$, that is $\nabla_{\mathbf{u}^{\text{flip}}} L$, since $\mathbf{u}^{\text{flip}} = \mathbf{I}_n^{\text{flip}} \mathbf{u} \mathbf{I}_n^{\text{flip}}$,

$$\nabla_{\mathbf{u}^{\text{flip}}} L = \mathbf{I}_n^{\text{flip}} \nabla_{\mathbf{u}^{\text{flip}}} L \mathbf{I}_n^{\text{flip}} = (\nabla_{\mathbf{u}} L)^{\text{flip}} \,. \tag{58}$$

In sum, for the reverse-order eigenvalues, gradients of their associated eigenvectors need to be flipped in both the horizontal and vertical directions, that is to apply Eq. (53), when the eigenvalue order (or equivalently the eigenvector order) is flipped.

D. Proof and Demo Code for Proposition 3.1

We explore the matrix structure by taking an example with $m = 2$ and $n = 3$, that is $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{y}$ as

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \,, \quad \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \,, \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} \,. \tag{59}$$

The Jacobian vector and Hessian matrices of the objective function in Eq. (1) over the solution $\mathbf{y}$ and the Hessian matrix over entries $\mathbf{A}$ and $\mathbf{b}$ are

$$\begin{aligned}
\nabla_Y \mathtt{f}(\mathbf{A},\mathbf{b},\mathbf{y}) &= \mathbf{A}^\top(\mathbf{A}\mathbf{y}-\mathbf{b}) \\
&= \begin{bmatrix} (a_{00}^2+a_{10}^2)y_0 + (a_{00}a_{01}+a_{10}a_{11})y_1 + (a_{00}a_{02}+a_{10}a_{12})y_2 \\ (a_{01}a_{00}+a_{11}a_{10})y_0 + (a_{01}^2+a_{11}^2)y_1 + (a_{01}a_{02}+a_{11}a_{12})y_2 \\ (a_{02}a_{00}+a_{12}a_{10})y_0 + (a_{02}a_{01}+a_{12}a_{11}y_1 + (a_{02}^2+a_{12}^2)y_2) \end{bmatrix} - \begin{bmatrix} a_{00}b_0+a_{10}b_1 \\ a_{01}b_0+a_{11}b_1 \\ a_{02}b_0+a_{12}b_1 \end{bmatrix}, \qquad (60)
\end{aligned}$$

$$\nabla^2_{bY}\mathtt{f}(\mathbf{A},\mathbf{b},\mathbf{y}) = -\begin{bmatrix} a_{00} & a_{10} \\ a_{01} & a_{11} \\ a_{02} & a_{12} \end{bmatrix} = -\mathbf{A}^\top, \qquad (61)$$

$$\begin{aligned}
\nabla^2_{AY}\mathtt{f}(\mathbf{A},\mathbf{b},\mathbf{y}) &= \begin{bmatrix} \begin{bmatrix} 2a_{00}y_0+a_{01}y_1+a_{02}y_2 & a_{00}y_1 & a_{00}y_2 \\ 2a_{10}y_0+a_{11}y_1+a_{12}y_2 & a_{10}y_1 & a_{10}y_2 \end{bmatrix} \\ \begin{bmatrix} a_{01}y_0 & a_{00}y_0+2a_{01}y_1+a_{02}x_2 & a_{01}y_2 \\ a_{11}y_0 & a_{10}y_0+2a_{11}y_1+a_{12}x_2 & a_{11}y_2 \end{bmatrix} \\ \begin{bmatrix} a_{02}y_0 & a_{02}y_1 & a_{00}y_0+a_{01}y_1+2a_{02}y_2 \\ a_{12}y_0 & a_{12}y_1 & a_{10}y_0+a_{11}y_1+2a_{12}y_2 \end{bmatrix} \end{bmatrix} - \begin{bmatrix} \begin{bmatrix} b_0 & 0 & 0 \\ b_1 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & b_0 & 0 \\ 0 & b_1 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & b_0 \\ 0 & 0 & b_1 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} a_{00}y_0 & a_{00}y_1 & a_{00}y_2 \\ a_{10}y_0 & a_{10}y_1 & a_{10}y_2 \end{bmatrix} \\ \begin{bmatrix} a_{01}y_0 & a_{01}y_1 & a_{01}y_2 \\ a_{11}y_0 & a_{11}y_1 & a_{11}y_2 \end{bmatrix} \\ \begin{bmatrix} a_{02}y_0 & a_{02}y_1 & a_{02}y_2 \\ a_{12}y_0 & a_{12}y_1 & a_{12}y_2 \end{bmatrix} \end{bmatrix} + \begin{bmatrix} \begin{bmatrix} a_{00}y_0+a_{01}y_1+a_{02}y_2 & 0 & 0 \\ a_{10}y_0+a_{11}y_1+a_{12}y_2 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & a_{00}y_0+a_{01}y_1+a_{02}y_2 & 0 \\ 0 & a_{10}y_0+a_{11}y_1+a_{12}y_2 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & a_{00}y_0+a_{01}y_1+a_{02}y_2 \\ 0 & 0 & a_{10}y_0+a_{11}y_1+a_{12}y_2 \end{bmatrix} \end{bmatrix} - \begin{bmatrix} \begin{bmatrix} b_0 & 0 & 0 \\ b_1 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & b_0 & 0 \\ 0 & b_1 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & b_0 \\ 0 & 0 & b_1 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} a_{00} \\ a_{10} \end{bmatrix} \mathbf{y}^\top \\ \begin{bmatrix} a_{01} \\ a_{11} \end{bmatrix} \mathbf{y}^\top \\ \begin{bmatrix} a_{02} \\ a_{12} \end{bmatrix} \mathbf{y}^\top \end{bmatrix} + \begin{bmatrix} \mathbf{Ay} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{Ay} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Ay} \end{bmatrix} - \begin{bmatrix} \mathbf{b} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{b} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{b} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{A}_0\mathbf{y}^\top \\ \mathbf{A}_1\mathbf{y}^\top \\ \mathbf{A}_2\mathbf{y}^\top \end{bmatrix} + \begin{bmatrix} \mathbf{Ay}-\mathbf{b} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{Ay}-\mathbf{b} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Ay}-\mathbf{b} \end{bmatrix}, \qquad (62)
\end{aligned}$$

where the left term can be written as $\mathbf{A}_i$ for all $i \in \mathcal{N} = \{1, ..., n\}$ and the right term assigns $(\mathbf{Ay}-\mathbf{b})$ to the diagonal of $\nabla^2_{AY}\mathtt{f}(\mathbf{A},\mathbf{b},\mathbf{y})$ which is $\mathcal{B}$ in Eq. (21). One can represent it as

$$\mathcal{B}_{ij} = \mathbf{A}_i\mathbf{y}_j^\top, \quad \forall i,j \in \mathcal{N}, \qquad (63)$$

$$\mathcal{B}_{ii} \leftarrow \mathcal{B}_{ii} + (\mathbf{Ay}-\mathbf{b}), \quad \forall i \in \mathcal{N}. \qquad (64)$$

One can also obtain the same structure from different examples. It is clear that forming $\mathcal{B}$ using $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{y}$ in this way avoids calculating the zero vectors and matrices, and thus, becoming more efficient than using Jacobian to calculate every element of such a spare matrix. To this end, we arrive at the result in Prop. 3.1. □

Meanwhile, we show an implementation example comparing the automatic Jacobian (without exploited structures) and ours with the exploited structure in Listing 1.

Listing 1. *An implementation example of exploited Hessian structure for LESS.*

```python
import torch
from torch.autograd import grad
from torch.autograd.functional import jacobian as J

def obj_fnc(A, u, b):
    Au_b = torch.einsum('bmn,bn->bm', A, u) - b
    loss = 0.5 * torch.einsum('bm,bm->b', Au_b, Au_b)

    return loss.sum()

# ==== Autogradient
def dfdy_auto_fnc(A, u, b):
    return grad(obj_fnc(A, u, b), (u), create_graph=True)[0]

def dffdAy_auto_fnc(A, u, b):
    m, n = A.shape[1:3]
    D = []

    with torch.enable_grad():
        for A_p, u_p, b_p in zip(A, u, b):
            A_p = A_p.view(1, m, n)
            u_p = u_p.view(1, n)
            b_p = b_p.view(1, m)
            D.append(J(lambda A: dfdy_auto_fnc(A, u_p, b_p), (A_p)))

        D = torch.cat(D, dim=0)

    return D

# ==== Structure exploited
def dfdy_fnc(A, u, b):
    Au_b = torch.einsum('bmn,bn->bm', A, u) - b

    return torch.einsum('bmn,bm->bn', A, Au_b)

def dffdAy_fnc(A, u, b):
    batch, m, n = A.shape
    D = A.new_zeros(batch, n, m, n)

    with torch.no_grad():
        for i in range(n):
            D_1 = torch.einsum('bm,bn->bmn', A[:, :, i], u)
            D_2 = torch.einsum('bmn,bn->bm', A, u) - b
            D[:, i] = D_1
            D[:, i, :, i] += D_2

    return D
```

E. Proof for Proposition 3.4

For the principal matrix features, the implicit differentiation is on the eigenvector associated with the largest eigenvalue as

$$\nabla_X L(\mathbf{y}) = \nabla_Y L(\mathbf{y}) \nabla_X \mathbf{y} = \nabla_Y L(\mathbf{y}) \left(\mathcal{H}^{-1} \mathcal{A}^T \left(\mathcal{A} \mathcal{H}^{-1} \mathcal{A}^\top \right)^{-1} \mathcal{A} \mathcal{H}^{-1} - \mathcal{H}^{-1} \right) \mathcal{B} = \mathcal{K} \mathcal{B} \in \mathbb{R}^{m \times m} \,, \tag{65}$$

where $\mathcal{K} \in \mathbb{R}^m$ is a vector given the upper-level loss $L \in \mathbb{R}$ from the bi-level problem. Then,

$$\nabla_X L(\mathbf{y}) = -\mathcal{K} \mathbf{y}^\top - \mathbf{y} \mathcal{K}^\top \,. \tag{66}$$

We find the matrix structure by taking the example of $m = 2$, where $\mathcal{K} = [k_0, k_1]^\top$ and $\mathbf{y} = [y_0, y_1]^\top$. Then,

$$\mathcal{B} = \mathcal{B}_0 + \mathcal{B}_1 = \begin{bmatrix} -y_0 & -y_1 & 0 & 0 \\ 0 & 0 & -y_0 & -y_1 \end{bmatrix} + \begin{bmatrix} -y_0 & 0 & -y_1 & 0 \\ 0 & -y_0 & 0 & -y_1 \end{bmatrix} \tag{67}$$

and

$$\mathcal{K}\mathcal{B} = \mathcal{K}\mathcal{B}_0 + \mathcal{K}\mathcal{B}_1 = \begin{bmatrix} -k_0 y_0 & -k_0 y_1 \\ -k_1 y_0 & -k_1 y_1 \end{bmatrix} + \begin{bmatrix} -k_0 y_0 & -k_1 y_0 \\ -k_0 y_1 & -k_1 y_1 \end{bmatrix} = -\begin{bmatrix} k_0 \\ k_1 \end{bmatrix} \begin{bmatrix} y_0 & y_1 \end{bmatrix} - \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \begin{bmatrix} k_0 & k_1 \end{bmatrix} = -\mathcal{K}\mathbf{y}^\top - \mathbf{y}\mathcal{K}^\top , \tag{68}$$

where $\mathcal{K}\mathcal{B}_0$ is the outer-product of $\mathcal{K}$ and $(-\mathbf{y})$, both $\mathcal{K}\mathcal{B}_1$ and $\mathcal{K}\mathcal{B}_0$ will be reshaped to $m \times m$ matrices. One can also apply the same rule to different examples, which will have the same structure in Eq. (66). Clearly, using $\mathbf{y} \in \mathbb{R}^m$ for $\nabla_X L(\mathbf{y})$ is much more efficient and requires much less memory than using $\mathcal{B} \in \mathbb{R}^{m\times(m\times m)}$. To this end, we prove Prop. 3.4. $\square$

F. More Results of LESS Optimization

In addition to Fig. 1, Fig. 4 contains the full list of the compared methods. The inputs for Fig. 1 are

$$\mathbf{A} = \begin{bmatrix} 0.569525 & -1.254572 \\ 0.414020 & 0.124439 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} -1.583332 \\ -0.286124 \end{bmatrix} . \tag{69}$$

In this case, the vanilla PGD requires more iterations than our "PGD+RM+BLSOne" and "PGD+RM+TWD". Statistically, from Table 4, when comparing "PGD" with "PGD+RM", all the metrics are better than "PGD". With "+BLSOne" and "+TWD", these can be further improved.

Table 4. Effectiveness evaluation on 1,000 random data sampled from $\mathcal{N}(0, 1)$ normal distribution. ***Bold****: the best, underline: the second best. "PGD": projected gradient descent (maximum 100 iterations), "RM": Riemannian manifold, "BLS": backtracking line search (α=0.5 and β=0.8), "TWD": tangent weight decay on Riemannian manifold (β=0.9), "DW": direction weight, "In" and "Out": the number of failed cases inner and outer of the sphere respectively, where the case is regarded as failed when the solution update reaches the maximum 100 iterations, "Imp.": the number of cases with energy no greater than SciPy. Problem sizes, m×n for A, are 2×2 (282 inner and 718 outer), 64×32 (555 inner and 445 outer), and 1024×256 (all inner).*

SciPy	PGD	RM	BLS		TWD	DW	2×2				64×32				1024×256			
			$\eta = 1$	$\eta = \nabla$			In↓	Out↓	Imp.↑	MRE↓	In↓	Out↓	Imp.↑	MRE↓	In↓	Out↓	Imp.↑	MRE↓
✓							-	-	-	-	-	-	-	-	-	-	-	-
	✓						$\underline{172}$	353	389	10.84	548	440	52	0.15	1000	-	0	1.35
	✓					✓	179	438	37	4.89	474	363	63	0.07	1000	-	0	0.47
	✓	✓					**0**	118	844	4.30	30	**0**	$\underline{980}$	**0.00**	$\underline{4}$	-	**1000**	**-0.03**
	✓	✓	✓				**0**	$\underline{14}$	**938**	$\underline{0.14}$	**6**	**0**	**994**	**0.00**	$\underline{4}$	-	**1000**	**-0.03**
	✓	✓	✓			✓	175	421	40	3.51	285	$\underline{142}$	286	$\underline{0.01}$	984	-	$\underline{999}$	**-0.03**
	✓	✓		✓			**0**	108	$\underline{853}$	4.21	$\underline{7}$	**0**	**994**	**0.00**	**2**	-	**1000**	**-0.03**
	✓	✓		✓		✓	204	536	17	1.19	472	355	92	0.04	1000	-	0	$\underline{0.42}$
	✓	✓			✓		**0**	**0**	806	**0.00**	29	**0**	954	**0.00**	$\underline{4}$	-	**1000**	**-0.03**
	✓	✓			✓	✓	204	533	15	3.12	466	352	90	0.04	1000	-	0	$\underline{0.42}$

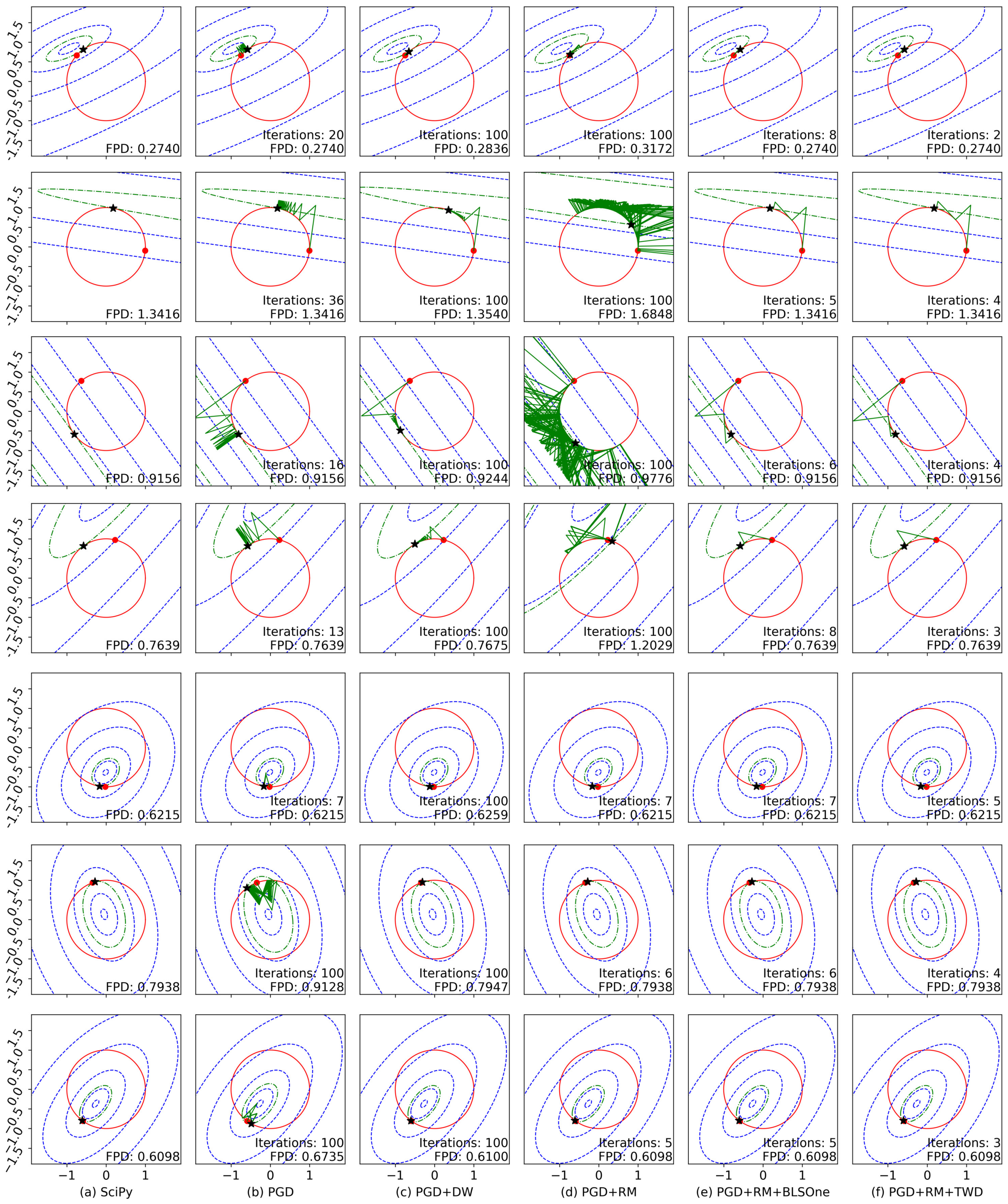
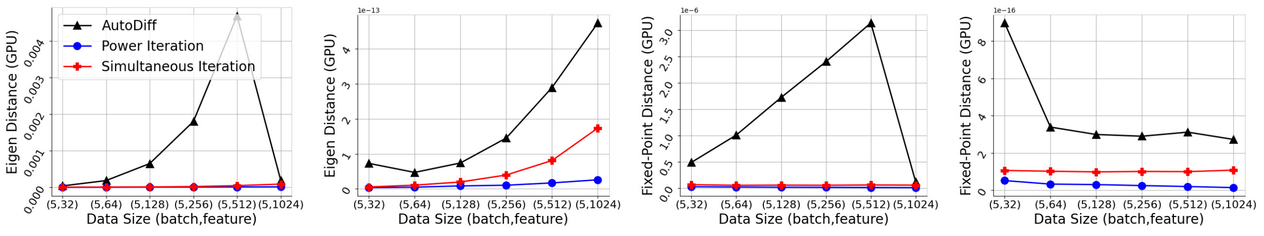


*Figure 4. Illustration of the iterative optimization in LESS. Each row is a sample with 6 methods. "**blue** dash": the least squares function; "**red** solid": the sphere constraints; "**green** solid": solution updates before and after* $\mathtt{Proj}_{RM}()$*; "**green** dot-dash": the least squares function with the final solution; "**red** dot": the initial solution; "**black** star": the final solution, not always the optimal. The least squares function with the exact solution (**green** dot-dash) should be tangent to the sphere (**red** solid). Our "PGD+RM+BLSOne" and "PGD+RM+TWD" require much fewer iterations than the others for the optimal solution with comparable fixed point distance (FPD).*

G. Evaluation of IED Precision and Computational Requirements

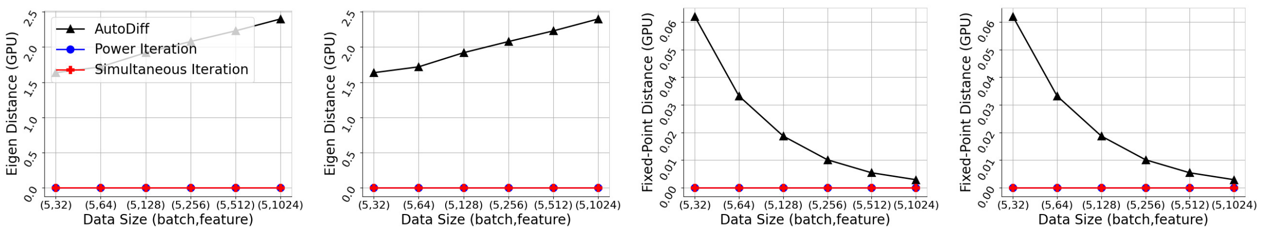
G.1. Symmetric and Nonsymmetric Data Sampling

Sampling Distributions. Our evaluated data $\mathbf{A}$ are sampled from *the standard Gaussian distribution* $\mathcal{N}(0, 1)$, *uniform distribution* in $[0, 1)$, *von Mises distribution* in $\mathcal{V}(0, 1)$ (von Mises, 1964), and *random choices* from [0.0, 10.0), with absolute values, where $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{A}^\top$ is applied for symmetry. We use `numpy.random.randn`(), `numpy.random.uniform`(), `numpy.random.vonmises`$(0, 1)$, and `numpy.random.choice`(10.0), respectively, from the NumPy library (Harris et al., 2020). They achieve similar numerical ranges as that from the Gaussian distribution in Figs. 5-6. Hyperparameters required by the sampling methods are not limited to the settings in this work, such as the mean and variance in the von Mises distribution and the value upper bound in the random choices.



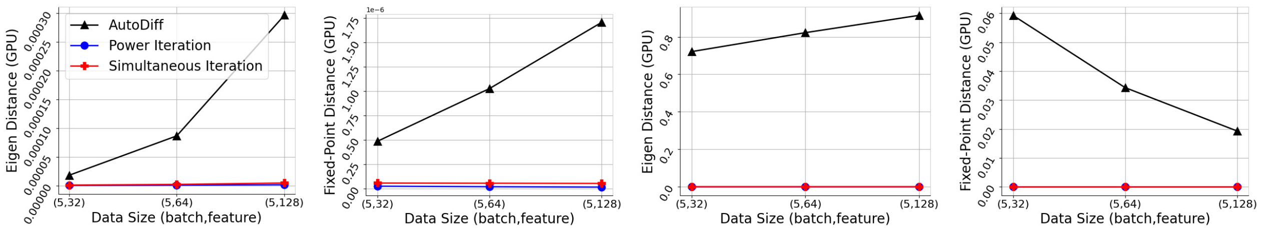
(a) ED, float32 (b) ED, float64 (c) FPD, float32 (d) FPD, float64

Figure 5. Symmetric $\mathbf{A}$ *sampled from Gaussian distribution with activation, precision evaluation with eigen distance and FPD.*



(a) ED, float32 (b) ED, float64 (c) FPD, float32 (d) FPD, float64

Figure 6. Nonsymmetric $\mathbf{A}$ *sampled from Gaussian distribution with activation, precision evaluation with eigen distance and FPD.*



(a) Symmetric $\mathbf{A}$, ED (b) Symmetric $\mathbf{A}$, FPD (c) Nonsymmetric $\mathbf{A}$, ED (d) Nonsymmetric $\mathbf{A}$, FPD

Figure 7. Evaluation on 3 data sizes within the memory capacity as it requires a $(2048 \times 8^2) \times m^2$ *dimension fully-connected layer in ResNet50, where* $m = \{32, 64, 128\}$. *All data is in float32. Both metrics are the less the better.*

The feasibility lies in 1) the simulation of normalized outputs (such as logits or softmax probability) of neural networks with an absolute operation, 2) the requirement of positive largest eigenvalue for IFT in Eq. (34). For nonsymmetric sampling, as we test 10,000 random samples (each with batch size 5) under this setting, the power iteration and simultaneous iteration algorithms can always achieve almost 0 eigen distance and 0 fixed-point distance. In contrast, PyTorch `eigh`()[*], denoted as "AutoDiff", has a much larger eigen distance, particularly in float32.

Simulation on ResNet50. We further generate random samples by using *a modified ResNet50* with pretrained weights, removing the adaptive average pooling layer and adjusting the last fully-connected layer to obtain m^2 out channels (instead

[*]PyTorch `eigh`() was designed for symmetric $\mathbf{A}$ and PyTorch `eig`() for complex and general data for nonsymmetric $\mathbf{A}$, but we are 1) not aim at complex data and 2) to avoid using `eigh`() for symmetric matrix but rather PI or SI by investigating these results.

of 1,000) which will then be reshaped to a $m \times m$ matrix, followed by the absolute operation and symmetry for $\mathcal{A}$ if required. It follows: random data with size $(5 \times 3 \times 256 \times 256)$ that is (batch $\times$ channel $\times$ height $\times$ width) sampled from $\mathcal{N}(0, 1) \rightarrow$ `ResNet()` $\rightarrow$ reshape to $m \times m \rightarrow$ absolute operation $\rightarrow$ $(\mathbf{A} \leftarrow \mathbf{A} + \mathbf{A}^\top)$ if symmetric $\mathbf{A}$ is required. In Figs. 7, "PI" and "SI" still achieve better eigen solutions than "AutoDiff" on both symmetric and nonsymmetric $\mathbf{A}$.

G.2. Additional Experiments on Computational Requirements

We evaluate the solution precision with eigen distance and fixed-point distance on symmetric and nonsymmetric $\mathbf{A}$ in Fig. 5 and Fig. 6 respectively. Additionally, the computational requirements include running time and GPU memory in Tables 5-10 for both symmetric and nonsymmetric $\mathbf{A}$. Although "AutoDiff" achieves faster speed in most cases, its precision is inferior to "PI" and "SI", particularly in float32. For the evaluation completion, we provide all of these results.

Meanwhile, since a solver achieves the same solutions given the same inputs, it consumes the same (at least quite similar) computational resources in the backward pass. We modulate each solver with different backward propagation methods into individual layers and provide all the forward and backward results.

In all these tables, implicit differentiation without exploited structure ("J") and with our exploited structure ("E") clearly distinguishes the benefits of exploiting the Jacobian and Hessian matrices in both the running time and memory requirements. For those with out-of-memory issues or extensive running time, we mark the results with "-" in the tables.

Table 5. Implicit eigen decomposition (IED) layer, CPU time (s) for symmetric $\mathbf{A}$. "AutoDiff": PyTorch `eigh()` function; "DDN": deep declarative network; "IFT": implicit function theorem; "unroll": unrolling the forward iteration for gradients via PyTorch autodiff mechanism; "J": autodiff Jacobian without exploited structure; "E": our implicit differentiation with exploited structure. Our best suggestions are highlighted considering the overall solution precision and computational requirements.

Method	5×32		5×64		5×128		5×256	
	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward
AutoDiff-unroll	0.0004	0.0001	0.0007	0.0002	0.0022	0.0003	0.0075	0.0011
AutoDiff-DDN-E	0.0004	0.0491	0.0008	0.0942	0.0021	0.1909	0.0080	2.0692
PI-unroll	0.0047	0.0121	0.0049	0.0175	0.0051	0.0344	0.0058	0.0984
PI-DDN-J	0.0038	0.0014	0.0039	0.0026	0.0042	0.0096	0.0049	0.0462
PI-IFT-J	0.0037	0.0230	0.0039	0.0452	0.0041	0.1103	0.0052	1.0283
PI-DDN-E	0.0037	0.0005	0.0039	0.0007	0.0041	0.0012	0.0048	0.0025
PI-IFT-E	0.0038	0.0003	0.0040	0.0004	0.0041	0.0007	0.0056	0.0018
SI-unroll	0.0138	0.0133	0.0401	0.0206	0.1198	0.0437	0.3927	0.1322
SI-DDN-J	0.0127	0.0014	0.0387	0.0026	0.1159	0.0096	0.3552	0.0464
SI-IFT-J	0.0127	0.0230	0.0383	0.0452	0.1096	0.1098	0.3791	1.0431
SI-DDN-E	0.0127	0.0005	0.0384	0.0007	0.1131	0.0012	0.3666	0.0028
SI-IFT-E	0.0127	0.0003	0.0393	0.0004	0.1139	0.0007	0.3667	0.0015

Table 6. Implicit eigen decomposition (IED) layer, GPU time (s) for symmetric $\mathbf{A}$. "AutoDiff": PyTorch `eigh()` function; "DDN": deep declarative network; "IFT": implicit function theorem; "unroll": unrolling the forward iteration for gradients via PyTorch autodiff mechanism; "J": autodiff Jacobian without exploited structure; "E": our implicit differentiation with exploited structure. Our best suggestions are highlighted considering the overall solution precision and computational requirements.

Method	5×32		5×64		5×128		5×256		5×512		5×1024	
	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward
AutoDiff-unroll	0.0004	0.0003	0.0041	0.0003	0.0097	0.0003	0.0261	0.0003	0.0758	0.0005	0.2193	0.0008
AutoDiff-DDN-E	0.0004	0.0784	0.0042	0.1523	0.0102	0.2946	0.0274	0.5861	0.0765	1.1695	0.2258	2.3861
PI-unroll	0.0077	0.0168	0.0077	0.0168	0.0079	0.0171	0.0079	0.0174	0.0082	0.0172	0.0086	0.0174
PI-DDN-J	0.0065	0.0028	0.0066	0.0050	0.0068	0.0096	0.0067	0.0182	0.0069	0.0436	0.0068	0.5281
PI-IFT-J	0.0065	0.0427	0.0065	0.0840	0.0067	0.1685	0.0068	0.3313	0.0070	0.6942	-	-
PI-DDN-E	0.0065	0.0065	0.0065	0.0067	0.0067	0.0033	0.0068	0.0059	0.0069	0.0114	0.0070	0.0414
PI-IFT-E	0.0065	0.0007	0.0066	0.0009	0.0068	0.0013	0.0068	0.0023	0.0069	0.0043	0.0070	0.0152
SI-unroll	0.0591	0.0173	0.1831	0.0169	0.2679	0.0227	0.4631	0.0580	1.0811	0.1443	2.6506	0.4876
SI-DDN-J	0.0590	0.0029	0.1847	0.0051	0.2694	0.0089	0.4642	0.0173	1.0623	0.0419	2.5715	0.1566
SI-IFT-J	0.0588	0.0426	0.1833	0.0835	0.2677	0.1663	0.4642	0.3380	1.0624	0.7120	-	-
SI-DDN-E	0.0584	0.0045	0.1832	0.0021	0.2676	0.0032	0.4646	0.0058	1.0622	0.0113	2.5676	0.0396
SI-IFT-E	0.0583	0.0007	0.1833	0.0009	0.2676	0.0013	0.4641	0.0022	1.0625	0.0042	2.6315	0.0144

Table 7. Implicit eigen decomposition (IED) layer, GPU memory (MB) for symmetric $\mathbf{A}$. "AutoDiff": PyTorch `eigh()` *function; "DDN": deep declarative network; "IFT": implicit function theorem; "unroll": unrolling the forward iteration for gradients via PyTorch autodiff mechanism; "J": autodiff Jacobian without exploited structure; "E": our implicit differentiation with exploited structure. Our best suggestions are highlighted considering the overall solution precision and computational requirements.*

Method	5×32		5×64		5×128		5×256		5×512		5×1024	
	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward
AutoDiff-unroll	0.5903	0.0972	0.8247	0.3896	1.7671	1.5605	5.5298	6.2456	20.5552	24.9907	80.6060	99.9810
AutoDiff-DDN-E	0.5903	0.0884	0.8232	0.3467	1.7646	1.3789	5.5249	5.5063	20.5454	22.0112	80.5864	88.0210
PI-unroll	0.8228	0.0537	0.9810	0.2280	1.4146	0.9282	2.8506	3.7334	7.5977	14.9688	24.5918	59.9395
PI-DDN-J	0.5293	0.6743	0.5898	5.1694	0.8281	40.6440	1.7759	322.5327	5.5464	2570.0620	20.5874	20520.1206
PI-IFT-J	0.5293	1.2705	0.5898	10.0791	0.8281	80.3135	1.7759	641.2510	5.5464	5125.0010	-	-
PI-DDN-E	0.5293	0.0674	0.5898	0.2534	0.8281	1.0044	1.7759	4.0068	5.5464	16.0117	20.5874	64.0215
PI-IFT-E	0.5293	0.1230	0.5898	0.4868	0.8281	1.9409	1.7759	7.7559	5.5464	31.0107	20.5874	124.0205
SI-unroll	4.6211	0.0962	16.5869	0.3882	65.0103	1.5576	256.5151	6.2402	1022.0249	24.9805	4083.0444	99.9609
SI-DDN-J	0.7539	0.6743	1.1182	5.1694	3.1353	40.6440	9.0151	322.5327	32.0249	2570.0620	123.0444	20520.1206
SI-IFT-J	0.7539	1.2705	1.1182	10.0791	3.1353	80.3135	9.0151	641.2510	32.0249	5125.0010	-	-
SI-DDN-E	0.7539	0.0674	1.1182	0.2534	3.1353	1.0044	9.0151	4.0068	32.0249	16.0117	123.0444	64.0215
SI-IFT-E	0.7539	0.1230	1.1182	0.4868	3.1353	1.9409	9.0151	7.7559	32.0249	31.0107	123.0444	124.0205

Table 8. Implicit eigen decomposition (IED) layer, CPU time (s) for nonsymmetric $\mathbf{A}$. "AutoDiff": PyTorch `eigh()` *function; "DDN": deep declarative network; "IFT": implicit function theorem; "unroll": unrolling the forward iteration for gradients via PyTorch autodiff mechanism; "J": autodiff Jacobian without exploited structure; "E": our implicit differentiation with exploited structure. Our best suggestions are highlighted considering the overall solution precision and computational requirements.*

Method	5×32		5×64		5×128		5×256	
	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward
AutoDiff-unroll	0.0004	0.0001	0.0008	0.0002	0.0022	0.0003	0.0075	0.0011
AutoDiff-DDN-E	0.0004	0.0483	0.0008	0.0925	0.0021	0.1902	0.0075	2.0929
PI-unroll	0.0045	0.0113	0.0046	0.0150	0.0048	0.0306	0.0054	0.0923
PI-DDN-J	0.0036	0.0014	0.0037	0.0026	0.0039	0.0096	0.0045	0.0463
PI-IFT-J	0.0037	0.0221	0.0038	0.0432	0.0040	0.1040	0.0050	0.9539
PI-DDN-E	0.0036	0.0004	0.0037	0.0006	0.0039	0.0012	0.0044	0.0024
PI-IFT-E	0.0036	0.0003	0.0038	0.0004	0.0039	0.0006	0.0046	0.0015
SI-unroll	0.0136	0.0131	0.0396	0.0198	0.1159	0.0426	0.3661	0.1298
SI-DDN-J	0.0127	0.0014	0.0387	0.0027	0.1182	0.0097	0.3651	0.0468
SI-IFT-J	0.0125	0.0220	0.0380	0.0428	0.1104	0.1045	0.3977	1.0048
SI-DDN-E	0.0126	0.0004	0.0388	0.0006	0.1162	0.0012	0.3617	0.0028
SI-IFT-E	0.0127	0.0003	0.0385	0.0004	0.1184	0.0007	0.3739	0.0016

Table 9. Implicit eigen decomposition (IED) layer, GPU time (s) for nonsymmetric $\mathbf{A}$. "AutoDiff": PyTorch `eigh()` *function; "DDN": deep declarative network; "IFT": implicit function theorem; "unroll": unrolling the forward iteration for gradients via PyTorch autodiff mechanism; "J": autodiff Jacobian without exploited structure; "E": our implicit differentiation with exploited structure. Our best suggestions are highlighted considering the overall solution precision and computational requirements.*

Method	5×32		5×64		5×128		5×256		5×512		5×1024	
	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward
AutoDiff-unroll	0.0004	0.0003	0.0041	0.0003	0.0097	0.0003	0.0262	0.0003	0.0759	0.0005	0.2188	0.0008
AutoDiff-DDN-E	0.0004	0.0816	0.0042	0.1586	0.0102	0.3062	0.0275	0.5802	0.0766	1.1564	0.2271	2.3584
PI-unroll	0.0072	0.0166	0.0072	0.0165	0.0074	0.0166	0.0074	0.0166	0.0077	0.0168	0.0080	0.0170
PI-DDN-J	0.0060	0.0028	0.0061	0.0050	0.0063	0.0090	0.0063	0.0174	0.0064	0.0431	0.0063	0.5642
PI-IFT-J	0.0061	0.0411	0.0061	0.0811	0.0065	0.1613	0.0064	0.3218	0.0066	0.6743	-	-
PI-DDN-E	0.0061	0.0064	0.0061	0.0068	0.0069	0.0033	0.0063	0.0059	0.0065	0.0113	0.0067	0.0410
PI-IFT-E	0.0060	0.0006	0.0060	0.0008	0.0064	0.0012	0.0064	0.0021	0.0065	0.0041	0.0066	0.0151
SI-unroll	0.0583	0.0166	0.1830	0.0163	0.2674	0.0226	0.4646	0.0582	1.0884	0.1454	2.6464	0.4886
SI-DDN-J	0.0588	0.0029	0.1847	0.0050	0.2692	0.0089	0.4624	0.0173	1.0584	0.0420	2.5674	0.1567
SI-IFT-J	0.0590	0.0403	0.1832	0.0793	0.2671	0.1581	0.4623	0.3211	1.0582	0.6744	-	-
SI-DDN-E	0.0584	0.0045	0.1833	0.0021	0.2671	0.0031	0.4625	0.0057	1.0583	0.0112	2.5622	0.0395
SI-IFT-E	0.0583	0.0007	0.1833	0.0009	0.2671	0.0013	0.4624	0.0022	1.0590	0.0041	2.6394	0.0144

Table 10. Implicit eigen decomposition (IED) layer, GPU memory (MB) for nonsymmetric $\mathbf{A}$*. "AutoDiff": PyTorch* `eigh()` *function; "DDN": deep declarative network; "IFT": implicit function theorem; "unroll": unrolling the forward iteration for gradients via PyTorch autodiff mechanism; "J": autodiff Jacobian without exploited structure; "E": our implicit differentiation with exploited structure. Our best suggestions are highlighted considering the overall solution precision and computational requirements.*

Method	5×32		5×64		5×128		5×256		5×512		5×1024	
	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward
AutoDiff-unroll	0.5747	0.0972	0.8091	0.3896	1.7515	1.5605	5.5142	6.2456	20.5396	24.9907	80.5903	99.9810
AutoDiff-DDN-E	0.5747	0.0884	0.8076	0.3467	1.7490	1.3789	5.5093	5.5063	20.5298	22.0112	80.5708	88.0210
PI-unroll	0.8071	0.0537	0.9653	0.2280	1.3989	0.9282	2.8350	3.7334	7.5820	14.9688	24.5762	59.9395
PI-DDN-J	0.5137	0.6743	0.5742	5.1694	0.8125	40.6440	1.7603	322.5327	5.5308	2570.0620	20.5718	20520.1206
PI-IFT-J	0.5137	1.2705	0.5742	10.0791	0.8125	80.3135	1.7603	641.2510	5.5308	5125.0010	-	-
PI-DDN-E	0.5137	0.0674	0.5742	0.2534	0.8125	1.0044	1.7603	4.0068	5.5308	16.0117	20.5718	64.0215
PI-IFT-E	0.5137	0.1230	0.5742	0.4868	0.8125	1.9409	1.7603	7.7559	5.5308	31.0107	20.5718	124.0205
SI-unroll	4.6055	0.0962	16.5713	0.3882	64.9946	1.5576	256.4995	6.2402	1022.0093	24.9805	4083.0288	99.9609
SI-DDN-J	0.7383	0.6743	1.1025	5.1694	3.1196	40.6440	8.9995	322.5327	32.0093	2570.0620	123.0288	20520.1206
SI-IFT-J	0.7383	1.2705	1.1025	10.0791	3.1196	80.3135	8.9995	641.2510	32.0093	5125.0010	-	-
SI-DDN-E	0.7383	0.0674	1.1025	0.2534	3.1196	1.0044	8.9995	4.0068	32.0093	16.0117	123.0288	64.0215
SI-IFT-E	0.7383	0.1230	1.1025	0.4868	3.1196	1.9409	8.9995	7.7559	32.0093	31.0107	123.0288	124.0205