



pytorch eigsh的本征向量梯度传递计算怎么实现的

你问的是 在 PyTorch 中如何实现 eigsh（对称稀疏矩阵特征分解）的本征向量梯度传递 对吧？

这个问题其实很 tricky，因为：

### 1. PyTorch 原生支持：

PyTorch 目前（截至 2.5）有 `torch.linalg.eigh`（对称/厄米矩阵的完整特征分解），是支持自动微分的。但是 `eigsh`（只取部分本征对，比如 SciPy 的 `scipy.sparse.linalg.eigsh` 基于 ARPACK）在 PyTorch 里没有现成的 autograd 实现。

### 2. 梯度传播的问题：

- 对本征值：比较好，公式类似于 Rayleigh quotient，可以直接传播。
- 对本征向量：更复杂，因为特征向量只有 up to a phase/sign 的自由度，而且在存在简并本征值时，导数不唯一。
- PyTorch 的 `torch.linalg.eigh` 已经内置了这些规则，所以如果能容忍 full decomposition（矩阵比较小），直接用它是最稳的。

## 如果你确实要实现 类似 eigsh 的梯度

有两种方法：

### 方法 1：基于 `torch.autograd.Function` 自定义 backward

你可以参考 `torch.linalg.eigh` 的梯度公式。

对于对称矩阵  $A$ ，本征分解：

$$Av_i = \lambda_i v_i$$

设  $V$  是本征向量矩阵， $\Lambda$  是本征值对角矩阵。

若输入的梯度是 wrt  $V, \Lambda$ ，则输入梯度 wrt  $A$  是：

$$\nabla_A = V \left( \text{Sym}(\nabla_\Lambda) + \mathcal{F}(\nabla_V, V, \Lambda) \right) V^\top$$

其中  $\mathcal{F}$  是一个用来处理非对角元的分式：

$$\mathcal{F}_{ij} = \frac{v_i^\top \nabla_V v_j}{\lambda_j - \lambda_i}, \quad i \neq j$$

（如果  $\lambda_i = \lambda_j$  要小心，通常要正交化。）

这个公式就是 PyTorch 内部对 `eigh` 的 backward 实现。

你可以写一个 `torch.autograd.Function`，forward 调用 SciPy 的 `eigsh`（稀疏求解），backward 时手动用上面公式传播梯度。

## 方法 2：通过 迭代方法 + 隐函数定理 (IFT)

另一种思路是：把 `eigsh` 看作是求解优化问题：

$$\max_{\|v\|=1} v^\top A v$$

本征向量是最优解。对这种约束优化问题，可以用 **隐函数定理** 推导梯度。

PyTorch 里可以用 `functorch / torch.autograd.functional` 实现，把 power iteration / LOBPCG 作为 forward 算法，然后在 backward 里自动求解线性系统来获得梯度。

事实上 PyTorch 已经有 `torch.lobpcg`，也支持 autograd（虽然稳定性比 `eigh` 差）。

## 推荐方案

- **矩阵不大**：直接用 `torch.linalg.eigh`，梯度最稳。
- **只要部分本征值/向量**：用 `torch.lobpcg`（支持 autograd），比自己实现容易。
- **真要自定义 eigsh**：写一个自定义 `autograd.Function`，forward 用 SciPy `eigsh`，backward 按上面公式实现。

要不要我帮你写一个 **PyTorch 自定义 autograd.Function 的 eigsh**（forward 用 SciPy，backward 实现梯度公式），你就可以直接在训练中调用？



### 方法1数学原理

好，我们把 **方法 1（自定义 backward 的数学原理）** 展开讲一下。核心问题就是：

给定对称矩阵  $A \in \mathbb{R}^{n \times n}$ ，它的本征分解

$$A v_i = \lambda_i v_i, \quad v_i^\top v_j = \delta_{ij},$$

如果在计算图中我们关心的是  $\lambda_i, v_i$ ，那么如何把  $\frac{\partial L}{\partial \lambda_i}, \frac{\partial L}{\partial v_i}$  反传回去，得到  $\frac{\partial L}{\partial A}$ ？

## 1. 本征分解的微分

我们知道

$$A = V \Lambda V^\top,$$

其中  $V = [v_1, \dots, v_n]$ ， $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ 。

对其微分：

$$dA = dV \Lambda V^\top + V d\Lambda V^\top + V \Lambda dV^\top.$$

## 2. 梯度传递的目标

假设 loss 是  $L(\Lambda, V)$ ，我们想要

$$\nabla_A L = \frac{\partial L}{\partial A}.$$

利用链式法则，核心就是把  $\nabla_\Lambda L, \nabla_V L$  转换成  $\nabla_A L$ 。

## 3. 本征值部分

对本征值：

Rayleigh 商公式：

$$\lambda_i = v_i^\top A v_i.$$

对  $A$  的微分：

$$d\lambda_i = v_i^\top (dA) v_i.$$

所以梯度是

$$\frac{\partial \lambda_i}{\partial A} = v_i v_i^\top.$$

因此本征值的梯度贡献为：

$$\nabla_A^{(\lambda)} = \sum_i \frac{\partial L}{\partial \lambda_i} v_i v_i^\top.$$

## 4. 本征向量部分

更复杂。我们从特征方程：

$$A v_i = \lambda_i v_i.$$

微分：

$$(dA) v_i + A dv_i = d\lambda_i v_i + \lambda_i dv_i.$$

左乘  $v_j^\top$  ( $j \neq i$ )：

$$v_j^\top (dA) v_i + v_j^\top A dv_i = d\lambda_i v_j^\top v_i + \lambda_i v_j^\top dv_i.$$

利用正交性  $v_j^\top v_i = 0$ ，以及  $A v_j = \lambda_j v_j$ ：

$$v_j^\top (dA) v_i + \lambda_j v_j^\top dv_i = \lambda_i v_j^\top dv_i.$$

化简：

$$v_j^\top dv_i = \frac{v_j^\top (dA) v_i}{\lambda_i - \lambda_j}, \quad j \neq i.$$

这说明了  $dv_i$  对  $dA$  的敏感性。

## 5. 组合公式

设 loss 对向量的梯度是  $\nabla_V L$ ，它的第  $i$  列是  $\frac{\partial L}{\partial v_i}$ 。

那么对  $A$  的贡献可以写为：

$$\nabla_A^{(v)} = \sum_{i \neq j} \frac{1}{\lambda_i - \lambda_j} \left( v_j v_i^\top \left( v_j^\top \frac{\partial L}{\partial v_i} \right) + \text{transpose} \right).$$

更紧凑写法是：

$$\nabla_A^{(v)} = V \left( F \odot (V^\top \nabla_V L) \right) V^\top,$$

其中

- $F_{ij} = \frac{1}{\lambda_i - \lambda_j}$ ,  $i \neq j$ ;  $F_{ii} = 0$ ,
- $\odot$  表示逐元素乘。


## 6. 最终梯度公式

因此：

$$\nabla_A L = V \left( \text{diag}(\nabla_A L) + F \odot (V^\top \nabla_V L) \right) V^\top.$$

这就是 PyTorch 在 `torch.linalg.eigh` backward 里实现的公式。

- 第一项：本征值的梯度。
- 第二项：本征向量的梯度修正项（包含特征值差分分母）。

 注意事项：

1. 如果  $\lambda_i = \lambda_j$ （简并），分母会发散 → 梯度不唯一（特征子空间内任意正交基都行）。PyTorch 在这种情况下会给 **某个稳定但任意的梯度**。
2. 本征向量本身有符号不确定性 ( $v_i \mapsto -v_i$ )，不过这在反传时无碍，因为 loss 对应的梯度也会同步改变符号。

要不要我把这个推导写成一个 **小例子（Python 代码验证公式正确性）**，比如构造  $L = \|v_1\|^2$  然后数值对比自动微分和公式？



in more details about the derivation of above steps

Great — let's go through the **derivation in full detail**.

We want the backward pass for eigenvalues and eigenvectors of a **symmetric (or Hermitian)** matrix  $A \in \mathbb{R}^{n \times n}$ .

## 1. Eigenvalue problem

We assume an eigendecomposition

$$Av_i = \lambda_i v_i, \quad v_i^\top v_j = \delta_{ij}.$$

Stacking eigenvectors:

$$AV = V\Lambda, \quad V^\top V = I,$$

where  $V = [v_1, \dots, v_n]$ , and  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ .

## 2. Differentials of eigen-decomposition

Differentiating

$$AV = V\Lambda$$

gives

$$(dA)V + A dV = dV \Lambda + V d\Lambda.$$

Multiply on the left by  $V^\top$ :

$$V^\top (dA)V + V^\top A dV = V^\top dV \Lambda + d\Lambda.$$

Since  $V^\top A = \Lambda V^\top$ ,

$$V^\top (dA)V + \Lambda (V^\top dV) = (V^\top dV) \Lambda + d\Lambda.$$

Let

$$M := V^\top (dA)V, \quad S := V^\top dV.$$

So we get:

$$M + \Lambda S = S \Lambda + d\Lambda.$$

## 3. Symmetry properties

- From  $V^\top V = I$ , differentiating:

$$(dV)^\top V + V^\top dV = 0 \quad \Rightarrow \quad S^\top + S = 0.$$

So  $S$  is **skew-symmetric**.

- From the equation:

$$M + \Lambda S = S\Lambda + d\Lambda.$$

Compare diagonal and off-diagonal parts separately.

---

## 4. Diagonal part (eigenvalues)

Take the diagonal of the equation:

$$M_{ii} + \lambda_i S_{ii} = S_{ii} \lambda_i + d\lambda_i.$$

But since  $S$  is skew-symmetric,  $S_{ii} = 0$ .

So

$$d\lambda_i = M_{ii}.$$

Recall  $M = V^\top (dA)V$ . Thus:

$$d\lambda_i = v_i^\top (dA)v_i.$$

This proves

$$\frac{\partial \lambda_i}{\partial A} = v_i v_i^\top.$$


---

## 5. Off-diagonal part (eigenvectors)

For  $i \neq j$ , look at entry  $(i, j)$ :

$$M_{ij} + \lambda_i S_{ij} = S_{ij} \lambda_j + d\Lambda_{ij}.$$

But  $d\Lambda$  is diagonal, so  $(i, j)$  entry is 0.

Thus:

$$M_{ij} + \lambda_i S_{ij} = \lambda_j S_{ij}.$$

So

$$M_{ij} = (\lambda_j - \lambda_i) S_{ij}.$$

Therefore

$$S_{ij} = \frac{M_{ij}}{\lambda_j - \lambda_i}, \quad i \neq j.$$


---

## 6. Reconstructing $dV$

We had

$$dV = VS,$$

so the differential of eigenvectors depends linearly on  $M = V^\top (dA)V$ .

## 7. Backpropagation

Suppose loss is  $L(\Lambda, V)$ .

- Gradient wrt eigenvalues:  
We already know

$$dL = \sum_i \frac{\partial L}{\partial \lambda_i} d\lambda_i = \sum_i (\nabla_\Lambda L)_i v_i^\top (dA) v_i.$$

Therefore

$$\nabla_A^{(\lambda)} = V \operatorname{diag}(\nabla_\Lambda L) V^\top.$$

- Gradient wrt eigenvectors:  
Now consider contribution from  $dV$ .  
Write

$$dL = \operatorname{tr}((\nabla_V L)^\top dV).$$

But  $dV = VS$ . So

$$dL = \operatorname{tr}((\nabla_V L)^\top VS) = \operatorname{tr}((V^\top \nabla_V L)^\top S).$$

Let

$$K := V^\top \nabla_V L.$$

Then

$$dL = \operatorname{tr}(K^\top S).$$

But recall  $S_{ij} = \frac{M_{ij}}{\lambda_j - \lambda_i}$ ,  $i \neq j$ , where  $M = V^\top (dA)V$ .

So

$$dL = \sum_{i \neq j} K_{ij} \frac{M_{ij}}{\lambda_j - \lambda_i}.$$

Rewriting:

$$dL = \sum_{i \neq j} \frac{K_{ij}}{\lambda_j - \lambda_i} (v_i^\top (dA) v_j).$$

This matches

$$dL = \text{tr} \left( (F \odot K)^\top M \right),$$

with

$$F_{ij} = \begin{cases} \frac{1}{\lambda_j - \lambda_i}, & i \neq j, \\ 0, & i = j. \end{cases}$$

Recall  $M = V^\top (dA) V$ . So

$$dL = \text{tr} \left( V (F \odot K) V^\top (dA) \right).$$

Thus the gradient contribution is

$$\nabla_A^{(v)} = V (F \odot K) V^\top.$$

## 8. Final formula

Combine both contributions:

$$\nabla_A L = V \left( \text{diag}(\nabla_\Lambda L) + F \odot (V^\top \nabla_V L) \right) V^\top.$$

- The **first term** comes from eigenvalues.
- The **second term** comes from eigenvectors, with the resolvent factor  $\frac{1}{\lambda_j - \lambda_i}$ .

✓ This is the exact formula used in PyTorch's `torch.linalg.eigh` autograd implementation.

⚠ If eigenvalues are degenerate ( $\lambda_i = \lambda_j$ ), the denominator blows up  $\rightarrow$  gradient is **undefined** (non-unique inside the eigensubspace). PyTorch returns some consistent but arbitrary gradient in that case.

Would you like me to **write the step 7 (backprop for eigenvectors)** in index notation (tensor form, no trace shorthand), so you see explicitly how each term appears?



write the step 7 (backprop for eigenvectors) in index notation

好的 — 我把 **第 7 步（本征向量部分反向传播）** 以下标/求和（**index notation**）的形式完整、逐步写出，尽量把每一步的代数变换都展开，便于直接在实现里照搬。

我会用爱因斯坦求和约定（重复下标自动求和），并在需要时展开成显式求和以免歧义。记号说明先列在前：

- $A_{\alpha\beta}$ ：输入的对称矩阵（ $\alpha, \beta = 1 \dots n$  表示矩阵行列索引）。
- $V_{\alpha i}$ ：第  $i$  个特征向量的第  $\alpha$  个分量（列主表示），即  $V = [v_1, \dots, v_n]$ 。
- $\lambda_i$ ：第  $i$  个特征值。
- $L$ ：标量损失，依赖于  $\Lambda$ （特征值）和  $V$ （特征向量）。
- $\frac{\partial L}{\partial V_{\alpha i}}$ ：loss 对  $V_{\alpha i}$  的梯度（记作  $(\nabla_V L)_{\alpha i}$ ）。



- 我们定义  $K_{ij} := V_{\gamma i} (\partial L / \partial V_{\gamma j})$  (即矩阵形式为  $K = V^\top (\nabla_V L)$ ) 。

## 1) 从 $V$ 的微分出发

Loss 对  $V$  的微分的贡献是

$$dL^{(V)} = \frac{\partial L}{\partial V_{\alpha i}} dV_{\alpha i}.$$

(这里对  $\alpha, i$  求和)

我们有关系  $dV = VS$ , 其中  $S$  是  $n \times n$  的矩阵且满足  $S^\top = -S$  (由  $V^\top V = I$  推出)。因此

$$dV_{\alpha i} = V_{\alpha j} S_{ji}.$$

代入得

$$dL^{(V)} = \frac{\partial L}{\partial V_{\alpha i}} V_{\alpha j} S_{ji}.$$

把  $V_{\alpha j} (\partial L / \partial V_{\alpha i})$  合并成  $K_{ji}$  (注意下标顺序) :

$$dL^{(V)} = K_{ji} S_{ji}.$$

(此处  $K_{ji} = V_{\alpha j} (\partial L / \partial V_{\alpha i})$ , 对  $i, j$  求和)

由于  $S_{ii} = 0$  (对角为零), 上式等价于对  $i \neq j$  的求和:

$$dL^{(V)} = \sum_{i \neq j} K_{ji} S_{ji}.$$

## 2) 用 $S$ 与 $M$ 的关系替换

之前我们推导出 (从微分方程)

$$S_{ij} = \frac{M_{ij}}{\lambda_j - \lambda_i}, \quad i \neq j,$$

其中  $M := V^\top (dA)V$ , 所以

$$M_{ij} = V_{\alpha i} dA_{\alpha\beta} V_{\beta j}.$$

因此

$$S_{ji} = \frac{M_{ji}}{\lambda_i - \lambda_j} = \frac{V_{\alpha j} dA_{\alpha\beta} V_{\beta i}}{\lambda_i - \lambda_j}, \quad (i \neq j).$$

代入  $dL^{(V)}$  :

$$dL^{(V)} = \sum_{i \neq j} K_{ji} \frac{V_{\alpha j} dA_{\alpha\beta} V_{\beta i}}{\lambda_i - \lambda_j}.$$

把  $dA_{\alpha\beta}$  提到外面（因为我们要把  $dL$  表达为  $dA_{\alpha\beta}$  的线性形式）：

$$dL^{(V)} = dA_{\alpha\beta} \sum_{i \neq j} \frac{K_{ji} V_{\alpha j} V_{\beta i}}{\lambda_i - \lambda_j}.$$

### 3) 交换求和下标以得到更对称的形式

将求和中  $i$  与  $j$  互换（把右边的表达式写成以  $K_{ij}$  出现的形式更直观）。把  $i \leftrightarrow j$  替换后有

$$\sum_{i \neq j} \frac{K_{ji} V_{\alpha j} V_{\beta i}}{\lambda_i - \lambda_j} = \sum_{i \neq j} \frac{K_{ij} V_{\alpha i} V_{\beta j}}{\lambda_j - \lambda_i}.$$

因此

$$dL^{(V)} = dA_{\alpha\beta} \sum_{i \neq j} \frac{K_{ij} V_{\alpha i} V_{\beta j}}{\lambda_j - \lambda_i}.$$

### 4) 将 $V$ 部分与 $\Lambda$ 部分合并得到 $\partial L / \partial A$

我们还记得来自特征值的梯度贡献（在第 4 步）：

$$dL^{(\Lambda)} = \sum_i \frac{\partial L}{\partial \lambda_i} d\lambda_i = \sum_i \frac{\partial L}{\partial \lambda_i} V_{\alpha i} dA_{\alpha\beta} V_{\beta i},$$

即

$$dL^{(\Lambda)} = dA_{\alpha\beta} \sum_i (\partial L / \partial \lambda_i) V_{\alpha i} V_{\beta i}.$$

把两部分加起来，总的微分是

$$dL = dA_{\alpha\beta} \left( \sum_i (\partial L / \partial \lambda_i) V_{\alpha i} V_{\beta i} + \sum_{i \neq j} \frac{K_{ij} V_{\alpha i} V_{\beta j}}{\lambda_j - \lambda_i} \right).$$

由  $dL = (\partial L / \partial A_{\alpha\beta}) dA_{\alpha\beta}$ （对  $\alpha, \beta$  求和），我们得到 **最终的索引形式梯度**：

$$\boxed{\frac{\partial L}{\partial A_{\alpha\beta}} = \sum_i \frac{\partial L}{\partial \lambda_i} V_{\alpha i} V_{\beta i} + \sum_{i \neq j} \frac{K_{ij} V_{\alpha i} V_{\beta j}}{\lambda_j - \lambda_i}}$$

其中  $K_{ij} = V_{\gamma i} \frac{\partial L}{\partial V_{\gamma j}}$ （对  $\gamma$  求和）。

### 5) 转换成更紧凑的矩阵形式（以便在代码中实现）

上面的索引式等价于矩阵表达：

$$\nabla_A L = V \operatorname{diag}(\nabla_\Lambda L) V^\top + V (F \odot (V^\top \nabla_V L)) V^\top,$$

其中

$$F_{ij} = \begin{cases} \frac{1}{\lambda_j - \lambda_i}, & i \neq j \\ 0, & i = j \end{cases}, \quad \text{and} \quad (V^\top \nabla_V L)_{ij} = K_{ij}.$$

## 6) 关于退化 ( $\lambda_i = \lambda_j$ ) 和数值稳定性的提醒

- 当  $\lambda_j - \lambda_i \rightarrow 0$  时，第二项分母发散  $\rightarrow$  梯度理论上不唯一（对简并子空间本征向量的选择会导致不同的基表示）。通常实现会需要：
  - 在分母上加一个很小的常数  $\varepsilon$ （数值正则化）用于稳定，或
  - 在检测到近似重根时采用子空间敏感的处理（例如把梯度限制到不改变子空间的方向上）。
- PyTorch 在 `eigh` 的实现里对数值和退化情况作了工程处理以避免 NaN，但数学上梯度在退化点确实是不唯一的。

如果你想，我可以：

- 把上面的公式直接写成 **可运行的 PyTorch 代码片段**（按索引/矩阵实现 `backward`），或
- 给出一个带数值示例（构造  $A$ 、计算前向的部分特征向量/值，然后用上面公式数值对比自动微分），帮助检验实现是否正确。哪一种你更需要？



方法 2：通过 迭代方法 + 隐函数定理 (IFT)? detailed math steps

好 — 我把 **方法 2（用迭代解 / 隐函数定理, Implicit Function Theorem, IFT）** 的数学推导做得尽可能完整且逐步清晰：从把特征向量/值视为方程的根出发，应用 IFT 求导出  $\partial v / \partial A, \partial \lambda / \partial A$  的显式线性系统形式，然后说明如何在反向传播里用该线性系统得到对  $A$  的梯度（这是在实现基于迭代求解器可微化时常用的做法）。我会同时给出“如何在代码里数值上实现”的方案（解线性系统求伴随向量），并说明它如何等价于前面方法 1 的分式公式。

## 设定（单一非退化本征值， $v$ 单位范数）

令  $A \in \mathbb{R}^{n \times n}$  对称， $(\lambda, v)$  满足

$$Av = \lambda v, \quad v^\top v = 1,$$

且假设  $\lambda$  为简单（非重根）。把  $x = (v, \lambda)$  看作  $A$  的函数，通过方程组把它视为隐函数的根。

定义映射  $F: \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^n \times \mathbb{R}$ ：

$$F(v, \lambda; A) = \begin{pmatrix} (A - \lambda I) v \\ v^\top v - 1 \end{pmatrix}.$$

根的条件是  $F(v, \lambda; A) = 0$ .

IFT 告诉我们（当  $\partial_{(v,\lambda)} F$  在该点可逆时）

$$\frac{\partial(v, \lambda)}{\partial A} = -(\partial_{(v,\lambda)} F)^{-1} \partial_A F.$$

接下来要明确两块雅可比矩阵并求解。

## 1) 计算雅可比块矩阵

把  $F$  写成  $\begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$  :

- $F_1(v, \lambda; A) = (A - \lambda I)v \in \mathbb{R}^n$ .
  - $\partial_v F_1 = A - \lambda I$ （对  $v$  的导数，形状  $n \times n$ ）。
  - $\partial_\lambda F_1 = -v$ （形状  $n \times 1$ ）。
  - $\partial_A F_1$  是对  $A$  的线性映射：对任意  $dA$  有  $\partial_A F_1[dA] = dA v$ .
- $F_2(v) = v^\top v - 1 \in \mathbb{R}$ .
  - $\partial_v F_2 = 2v^\top$ （行向量，形状  $1 \times n$ ）。
  - $\partial_\lambda F_2 = 0$ .
  - $\partial_A F_2 = 0$ .

因此块雅可比（关于  $x = (v, \lambda)$ ）为

$$J := \partial_{(v,\lambda)} F = \begin{pmatrix} A - \lambda I & -v \\ 2v^\top & 0 \end{pmatrix} \in \mathbb{R}^{(n+1) \times (n+1)}.$$

而  $\partial_A F$ （把  $dA$  映到  $F$  的变化）是映射

$$\partial_A F[dA] = \begin{pmatrix} dA v \\ 0 \end{pmatrix}.$$

由 IFT :

$$\begin{pmatrix} dv \\ d\lambda \end{pmatrix} = -J^{-1} \begin{pmatrix} dA v \\ 0 \end{pmatrix}.$$

这是“原始微分表达式”。但我们通常是在 **反向传播** 中需要把标量 loss  $L$  关于  $v, \lambda$  的灵敏度（梯度）反传回  $A$ 。用对偶/伴随（adjoint）方式更直接。

## 2) 伴随法（在反向传播中常用）

令我们在计算图中有标量损失  $L(v, \lambda)$ 。写

$$g_v := \frac{\partial L}{\partial v} \in \mathbb{R}^n, \quad g_\lambda := \frac{\partial L}{\partial \lambda} \in \mathbb{R}.$$

微分链式：

$$dL = g_v^\top dv + g_\lambda d\lambda.$$

把  $dv, d\lambda$  用 IFT 表达式替换：

$$dL = - \begin{pmatrix} g_v^\top & g_\lambda \end{pmatrix} J^{-1} \begin{pmatrix} dA v \\ 0 \end{pmatrix}.$$

将上式改写为“求伴随变量  $y$ ”的形式：定义伴随向量  $y \in \mathbb{R}^{n+1}$  为

$$y := J^{-\top} \begin{pmatrix} g_v \\ g_\lambda \end{pmatrix}, \quad \text{等价于} \quad J^\top y = \begin{pmatrix} g_v \\ g_\lambda \end{pmatrix}.$$

(在数值实现里通常直接解这个线性系统得到  $y$ 。)

令将  $y$  分块为  $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ，其中  $y_1 \in \mathbb{R}^n$ ,  $y_2 \in \mathbb{R}$ 。

由上面的等式可得

$$dL = -y^\top \begin{pmatrix} dA v \\ 0 \end{pmatrix} = -y_1^\top (dA v).$$

把  $dA$  从向量乘法中移出并转为 Frobenius 内积（冒号表示矩阵的 Frobenius 内积  $\langle X, Y \rangle = \text{tr}(X^\top Y)$ ）：

$$dL = -\text{tr}((y_1 v^\top)^\top dA) = -\langle y_1 v^\top, dA \rangle.$$

**注意：**上式是对任意（非对称） $dA$  的表达。但我们原始问题  $A$  是对称矩阵，实际允许的  $dA$  仅为对称扰动。将上述式子用对称化获得针对对称  $A$  的梯度：

$$dL = -\frac{1}{2} \langle y_1 v^\top + v y_1^\top, dA \rangle.$$

因此对称矩阵  $A$  的梯度为

$$\boxed{\frac{\partial L}{\partial A} = -\frac{1}{2} (y_1 v^\top + v y_1^\top)}.$$

这就是基于 IFT / 伴随法给出的通用表达：先解线性系统  $J^\top y = [g_v; g_\lambda]$ ，然后用上式构造  $\partial L / \partial A$ 。

### 3) 展开块线性系统（便于数值实现）

写出  $J^\top y = [g_v; g_\lambda]$  的显式块形式（便于在程序中解）：

由于

$$J = \begin{pmatrix} A - \lambda I & -v \\ 2v^\top & 0 \end{pmatrix}, \quad J^\top = \begin{pmatrix} A - \lambda I & 2v \\ -v^\top & 0 \end{pmatrix},$$

所以  $J^\top y = \begin{pmatrix} g_v \\ g_\lambda \end{pmatrix}$  展开为

$$\begin{cases} (A - \lambda I) y_1 + 2v y_2 = g_v, & (1) \\ -v^\top y_1 = g_\lambda. & (2) \end{cases}$$

这是一组  $n + 1$  线性方程，未知量是  $y_1 \in \mathbb{R}^n, y_2 \in \mathbb{R}$ 。在数值实现上通常直接构造并求解整个  $(n + 1) \times (n + 1)$  稀疏/稠密线性系统，或用分块消元技巧：

- 从 (2) 得到  $v^\top y_1 = -g_\lambda$  (约束) ,
- 将 (1) 投影到  $v^\perp$  子空间上 (因为  $A - \lambda I$  在  $v^\perp$  上可逆, 当  $\lambda$  是简单特征值时), 在  $v^\perp$  上我们可以用  $(A - \lambda I)|_{v^\perp}^{-1}$  求  $y_1$  的分量, 剩下的分量沿  $v$  的方向由 (2) 决定。

但在实现上——最简单且健壮的做法是直接通用线性求解器求解  $J^\top y = [g_v; g_\lambda]$ 。

## 4) 与方法 1 的分式公式等价性 (解析展开)

如果把  $y_1$  在特征向量基上展开  $y_1 = \sum_j \alpha_j v_j$  并解上面的方程, 可以得到

$$\alpha_j = -\frac{K_{ji}}{\lambda_j - \lambda_i} \quad (j \neq i),$$

并且  $\alpha_i$  由约束得到 (相关于  $g_\lambda$ )。把结果带回  $-\frac{1}{2}(y_1 v^\top + v y_1^\top)$  并化简, 会恢复前面方法 1 中的公式

$$\nabla_A L = V \left( \text{diag}(\nabla_\Lambda L) + F \odot (V^\top \nabla_V L) \right) V^\top,$$

即两种方法在数学上是等价的; IFT/伴随法给出的是一种“解线性系统”的数值路线, 而方法 1 给出了当你显式知道所有特征分解时的封闭式表达 (含  $\frac{1}{\lambda_j - \lambda_i}$  项)。IFT 方法更通用、也更易用在“forward 用迭代法求根 (没有显式给出全部特征)”的场景下: 只要能把 forward 的迭代 (或固定点方程) 写成  $F(x; A) = 0$ , 就能用同样方式求导。

## 5) 数值/实现要点 (实践建议)

- 在自动微分的反传实现里, 常见流程:
  1. 前向: 用迭代方法 (例如 power iteration、Lanczos、LOBPCG 或自定义的迭代求解器) 求解  $v, \lambda$  (并保存迭代中必要的中间量以便加速伴随求解, 或仅保存  $v, \lambda$  并构造  $J$ )。
  2. 反向: 已知  $g_v, g_\lambda$ , 构造  $J^\top$  并解  $J^\top y = [g_v; g_\lambda]$ , 得到  $y_1$ 。
  3.  $\partial L / \partial A = -\frac{1}{2}(y_1 v^\top + v y_1^\top)$ 。(如果在 forward 中保证  $A$  严格对称, 则返回对称梯度。)
- 解线性系统的数值稳定性:  $J$  的条件数依赖于  $\min_{j \neq i} |\lambda_j - \lambda_i|$ 。当特征值接近退化 (小间距) 时,  $J$  将变得病态, 解出的  $y$  会很不稳定, 这也对应梯度本身在数学上趋于不定 (与方法 1 的  $\frac{1}{\lambda_j - \lambda_i}$  发散一致)。  
实用策略:
  - 在求解器中对分母做微小正则化 (如  $\lambda_j - \lambda_i \mapsto \lambda_j - \lambda_i + \varepsilon$ ) , 或
  - 在检测到退化时退回计算整个子空间并对梯度做子空间正交化处理 (更复杂)。
- 计算代价: 构造并直接解  $(n + 1) \times (n + 1)$  线性系统 (稠密情形) 成本约  $O(n^3)$ 。但如果  $A$  非常大且稀疏, 你可以利用迭代线性解算器 (例如 GMRES / MINRES) 求解关于  $y$  的系统, 且每一步只需能做矩

阵-向量乘  $(A - \lambda I) \cdot x$  与  $v$  的内积，这样在大规模稀疏场景更可行。

## 6) 简短总结（流程）

1. 将特征方程写成  $F(v, \lambda; A) = 0$ 。
2. 计算  $J = \partial_{(v, \lambda)} F$  和  $\partial_A F[dA] = (dA) v$ 。
3. 用伴随法：解线性系统  $J^\top y = [g_v; g_\lambda]$ 。
4. 得到  $\partial L / \partial A = -\frac{1}{2}(y_1 v^\top + v y_1^\top)$ 。

这就是基于 IFT 的完整且通用的数学与数值框架——它在实现可微的迭代特征求解器（例如把 power iteration / ARPACK / Lanczos 封装为 differentiable operator）时非常有用，因为你不需要显式给出全部本征向量的分式公式，只需解一个伴随线性系统即可得到反向梯度。

如果你愿意，我可以接下来做两件事（你选其一或两个都要）：

1. 写出**具体的符号推导**：把上面“伴随法”里  $y_1$  在特征基上展开，显示如何还原出  $\frac{K_{ij}}{\lambda_j - \lambda_i}$  的分式表达，从而证明等价性（完整代数推导）。
2. 给出**可运行的 PyTorch 伪代码 / 真代码**（含如何在反向里构造  $J^\top$  并解线性系统得到梯度；并包含对称化步骤与退化时的简单数值正则化），便于你直接在项目里复用。

你想先看哪一个？