Amortized Eigendecomposition for Neural Networks

Tianbo Li, Zekun Shi, Jiaxi Zhao, Min Lin

SEA AI Lab
National University of Singapore

An Optimization Problem Involving Eigendecomposition

$$\min_{\boldsymbol{\theta}} f\big(h_{\boldsymbol{\theta}}(\boldsymbol{X}), \boldsymbol{V}, \boldsymbol{\Lambda}\big)$$

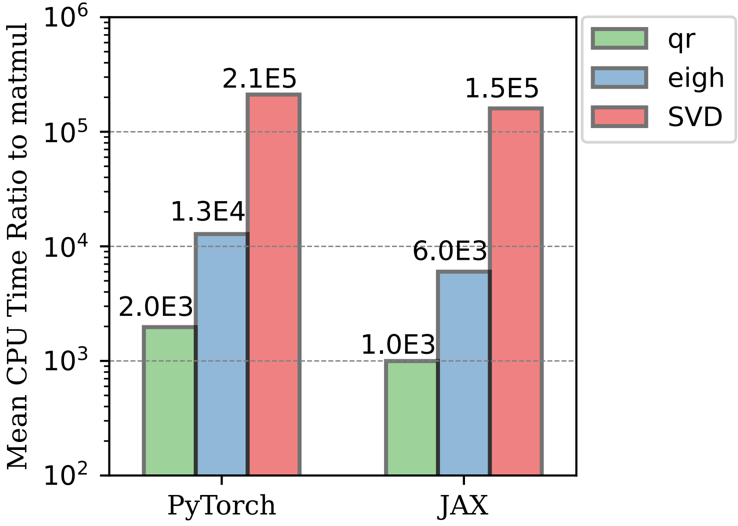
$$\text{s.t. } \boldsymbol{V}^{\top}\boldsymbol{\Lambda}\boldsymbol{V} = \boldsymbol{A}$$ where A is constructed from h(x)

Nuclear Norm Regularization

$$\min_{\boldsymbol{\theta}} \underbrace{f\big(h_{\boldsymbol{\theta}}(\boldsymbol{X})\big)}_{\text{Model loss}} + \underbrace{\eta\|\boldsymbol{\theta}\|_*}_{\text{Nuclear norm of parameters}}$$

Eigendecomposition is Slow

In both PyTorch and JAX, performing Singular Value Decomposition and eigendecomposition can be 10x to 100x slower than matrix multiplication for a 10000 x 10000 matrix.



The idea

Eigendecomposition itself is an optimization process;

however, it does not need to fully converge at each step;

it only needs to reach the desired outcome in the end.

Amortized Eigendecomposition

Algorithm 1 The conventional eigendecomposition in a neural network outlined in Eq. (2)

Input: Dataset $\boldsymbol{X}$, encoder $h_{\boldsymbol{\theta}}$, task $f_{\boldsymbol{\omega}}$;
Initialize model parameter $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$;
while not converged **do**
  compute $\boldsymbol{A}$ from $h_{\boldsymbol{\theta}}(\boldsymbol{X})$;
  $\boldsymbol{V}, \boldsymbol{\Lambda} = \mathsf{eigh}(\boldsymbol{A})$;
  compute $f_{\boldsymbol{\omega}}(h_{\boldsymbol{\theta}}(\boldsymbol{X}), \boldsymbol{V}, \boldsymbol{\Lambda})$;
  update $\boldsymbol{\theta}$, $\boldsymbol{\omega}$ by gradient descent;
end while

Algorithm 2 The amortized eigendecomposition technique outlined in Eq. (12)

Input: Dataset $\boldsymbol{X}$, encoder $h_{\boldsymbol{\theta}}$, task $f_{\boldsymbol{\omega}}$, and η;
Initialize model parameter $\boldsymbol{\theta}$, $\boldsymbol{\omega}$ and $\boldsymbol{W}$;
while not converged **do**
  compute $\boldsymbol{A}$ from $h_{\boldsymbol{\theta}}(\boldsymbol{X})$;
  $\boldsymbol{U} = \mathsf{QR}(\boldsymbol{W}), \boldsymbol{\Sigma} = \mathsf{diag}(\boldsymbol{U}^\top \boldsymbol{A} \boldsymbol{U})$;
  compute $f_{\boldsymbol{\omega}}(h_{\boldsymbol{\theta}}(\boldsymbol{X}), \boldsymbol{U}, \boldsymbol{\Sigma}) + \eta g(\boldsymbol{A}, \boldsymbol{U})$;
  update $\boldsymbol{\theta}$, $\boldsymbol{\omega}$, $\boldsymbol{W}$ by gradient descent;
end while

Table 1: Evaluation of execution times per iteration on three tasks.

Task	Dimension	Backbone time (s/iter) t_0	Backbone+ eigh/svd time (s/iter) t_1	Backbone+ our method time (s/iter) t_2	Speed-up $\frac{t_1-t_0}{t_2-t_0}$
Nuclear norm regularization	128×128	5.275E-2	8.323E-2	6.025E-2	4.06×
	256×256	5.600E-2	1.209E-1	6.080E-2	13.5×
	512×512	7.186E-2	2.616E-1	7.366E-2	105.4×
Latent-space PCA	256×2	4.178E-3	1.446E-2	1.117E-2	1.47×
	512×2	6.792E-3	2.918E-2	2.224E-2	1.45×
	1028×2	1.434E-2	7.018E-2	5.467E-2	1.39×
Low-rank GCN	2708×16	1.021E-3	1.769E-2	1.732E-3	23.4×
	3312×16	1.367E-3	2.825E-2	2.498E-3	23.7×
	19717×16	1.931E-2	4.941E+0	2.731E-2	615.2×