FAST DIFFERENTIABLE MATRIX SQUARE ROOT –APPENDIX–

Anonymous authors
Paper under double-blind review

A APPENDIX

A.1 SUMMARY OF OUR ALGORITHMS

Algorithm. 1 and Algorithm. 2 summarize the forward pass (FP) and the backward pass (BP) of our proposed methods, respectively. The hyper-parameter K in Algorithm. 1 means the degrees of power series, and T in Algorithm. 2 denotes the iteration times.

Algorithm 1: FP of our MTP and MPA.

Input: $\boldsymbol{A}$ and K
Output: $\boldsymbol{A}^{\frac{1}{2}}$
if *MTP* **then**
  `// FP method is MTP`
  $\boldsymbol{A}^{\frac{1}{2}} \leftarrow \boldsymbol{I} - \sum_{k=1}^{K} \left| \binom{\frac{1}{2}}{k} \right| (\boldsymbol{I} - \frac{\boldsymbol{A}}{||\boldsymbol{A}||_{\mathrm{F}}})^k$;
else
  `// FP method is MPA`
  $M \leftarrow \frac{K-1}{2}$, $N \leftarrow \frac{K-1}{2}$;
  $\boldsymbol{P}_M \leftarrow \boldsymbol{I} - \sum_{m=1}^{M} p_m (\boldsymbol{I} - \frac{\boldsymbol{A}}{||\boldsymbol{A}||_{\mathrm{F}}})^m$;
  $\boldsymbol{Q}_N \leftarrow \boldsymbol{I} - \sum_{n=1}^{N} q_n (\boldsymbol{I} - \frac{\boldsymbol{A}}{||\boldsymbol{A}||_{\mathrm{F}}})^n$;
  $\boldsymbol{A}^{\frac{1}{2}} \leftarrow \boldsymbol{Q}_N^{-1} \boldsymbol{P}_M$;
end
Post-compensate $\boldsymbol{A}^{\frac{1}{2}} \leftarrow \sqrt{||\boldsymbol{A}||_{\mathrm{F}}} \cdot \boldsymbol{A}^{\frac{1}{2}}$

Algorithm 2: BP of our Lyapunov solver.

Input: $\frac{\partial l}{\partial \boldsymbol{A}^{\frac{1}{2}}}$, $\boldsymbol{A}^{\frac{1}{2}}$, and T
Output: $\frac{\partial l}{\partial \boldsymbol{A}}$
$\boldsymbol{B}_0 \leftarrow \boldsymbol{A}^{\frac{1}{2}}$, $\boldsymbol{C}_0 \leftarrow \frac{\partial l}{\partial \boldsymbol{A}^{\frac{1}{2}}}$, $i \leftarrow 0$;
Normalize $\boldsymbol{B}_0 \leftarrow \frac{\boldsymbol{B}_0}{||\boldsymbol{B}_0||_{\mathrm{F}}}$, $\boldsymbol{C}_0 \leftarrow \frac{\boldsymbol{C}_0}{||\boldsymbol{B}_0||_{\mathrm{F}}}$;
while $i < T$ **do**
  `// Coupled iteration`
  $\boldsymbol{B}_{k+1} \leftarrow \frac{1}{2} \boldsymbol{B}_k (3\boldsymbol{I} - \boldsymbol{B}_k^2)$;
  $\boldsymbol{C}_{k+1} \leftarrow \frac{1}{2} \Big(-\boldsymbol{B}_k^2 \boldsymbol{C}_k + \boldsymbol{B}_k \boldsymbol{C}_k \boldsymbol{B}_k + \boldsymbol{C}_k (3\boldsymbol{I} - \boldsymbol{B}_k^2) \Big)$;
  $i \leftarrow i + 1$;
end
$\frac{\partial l}{\partial \boldsymbol{A}} \leftarrow \frac{1}{2} \boldsymbol{C}_k$;

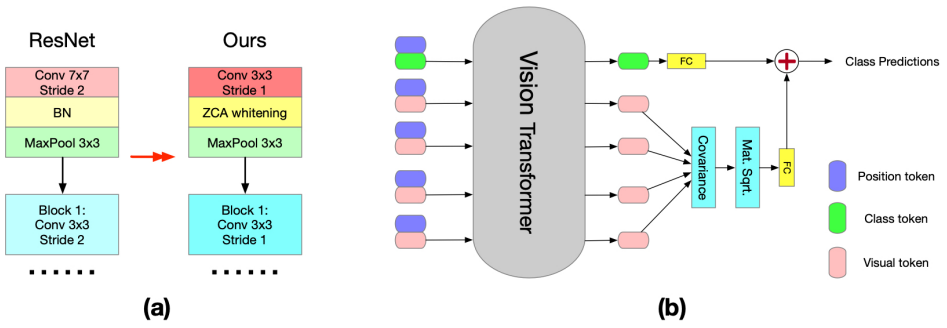

Figure 1: **(a)** The architecture changes of ResNet models in the experiment of ZCA whitening. Following Wang et al. (2021), the decorrelated batch normalization layer is inserted after the first convolutional layer. The kernel sizes, the stride of the first convolution layer, and the stride of the first ResNet block are changed correspondingly. **(b)** The scheme of So-ViT (Xie et al., 2021). The covariance square root of the visual tokens are computed to assist the classification. In the original vision transformer (Dosovitskiy et al., 2020), only the class token is utilized for class predictions.

A.2 ZCA WHITENING

Consider the reshaped feature map $\boldsymbol{X}\in\mathbb{R}^{C\times BHW}$. The ZCA whitening procedure first computes its sample covariance as:

$$\boldsymbol{A}=(\boldsymbol{X}-\mu(\boldsymbol{X}))(\boldsymbol{X}-\mu(\boldsymbol{X}))^T+\epsilon\boldsymbol{I} \tag{1}$$

where $\boldsymbol{A}\in\mathbb{R}^{C\times C}$, $\mu(\boldsymbol{X})$ is the mean of $\boldsymbol{X}$, and ϵ is a small constant to make the covariance strictly positive definite. Afterwards, the inverse square root is calculated to whiten the feature map:

$$\boldsymbol{X}_{whitend}=\boldsymbol{A}^{-\frac{1}{2}}\boldsymbol{X} \tag{2}$$

By doing so, the eigenvalues of $\boldsymbol{X}$ are all ones, *i.e.,* the feature is uncorrelated. During the training process, the training statistics are stored for the inference phase. Compared with the ordinary batch normalization that only standardizes the data, the ZCA whitening further eliminates the correlation of the data. The detailed architecture changes of ResNet are displayed in Fig. 1 (a). For the NS iteration and our methods, we first calculate the matrix square root $\boldsymbol{A}^{\frac{1}{2}}$ and compute $\boldsymbol{X}_{whitend}$ by solving the linear system $\boldsymbol{A}^{\frac{1}{2}}\boldsymbol{X}_{whitend}=\boldsymbol{X}$.

A.3 SECOND-ORDER VISION TRANSFORMER

For the task of image recognition, ordinary vision transformer (Dosovitskiy et al., 2020) attaches an empty class token to the sequence of visual tokens and only uses the class token for prediction, which may not exploit the rich semantics embedded in the visual tokens. Instead, So-ViT (Xie et al., 2021) proposes to leverage the high-level visual tokens to assist the task of classification:

$$y=\mathrm{FC}(c)+\mathrm{FC}\Big((\boldsymbol{X}\boldsymbol{X}^T)^{\frac{1}{2}}\Big) \tag{3}$$

where c is the output class token, $\boldsymbol{X}$ denotes the visual token, and y is the combined class predictions. We show the model overview in Fig. 1 (b). Equipped with the covariance pooling layer, the second-order vision transformer removes the need for pre-training on the ultra-large-scale datasets and achieves state-of-the-art performance even when trained from scratch. To reduce the computational budget, the So-ViT further proposes to use Power Iteration (PI) to approximate the dominant eigenvector and eigenvalue of the covariance $\boldsymbol{X}\boldsymbol{X}^T$.

A.4 BASELINES

In the experiment section, we compare our proposed two methods with the following baselines:

- Power Iteration (PI). It is suggested in the original So-ViT to compute only the dominant eigenpair.
- SVD-PI (Wang et al., 2019) that uses PI to compute the gradients of SVD.
- SVD-Taylor (Wang et al., 2021; Song et al., 2021) that applies the Taylor polynomial to approximate the gradients.
- SVD-Padé (Song et al., 2021) that proposes to closely approximate the SVD gradients using Padé approximants. Notice that our MTP/MPA used in the FP is fundamentally different from the Taylor polynomial or Padé approximants used in the BP of SVD-Padé. For our method, we use Matrix Taylor Polynomial (MTP) and Matrix Padé Approximants (MPA) to derive the matrix square root in the FP. For the SVD-Padé, they use scalar Taylor polynomial and scalar Padé approximants to approximate the gradient $\frac{1}{\lambda_i-\lambda_j}$ in the BP. That is to say, their aim is to use the technique to compute the gradient and this will not involve the back-propagation of Taylor polynomial or Padé approximants.
- NS iteration (Schulz, 1933; Higham, 2008) that uses the Newton-Schulz iteration to compute the matrix square root. It has been widely applied in different tasks, including covariance pooling (Li et al., 2018) and ZCA whitening (Huang et al., 2018). We note that although Huang et al. (2019) and Higham (2008) use different forms of NS iteration, the two representations are equivalent to each other. Huang et al. (2019) just replace $\boldsymbol{Y}_k$ with $\boldsymbol{Z}_k\boldsymbol{A}$ and re-formulate the iteration using one variable. The computation complexity is still the same.

As the ordinary differentiable SVD suffers from the gradient explosion issue and easily causes the program to fail, we do not take it into account for comparison.

Unlike previous methods such as SVD and NS iteration, our MPA-Lya/MTP-Lya does not have a consistent FP and BP algorithm. However, we do not think it will bring any caveat to the stability or performance. Our ablation study in the Appendix A.7.2 implies that our BP Lyapunov solver approximates the real gradient very well (*i.e.,* $||\boldsymbol{B}_k-\boldsymbol{I}||_{\mathrm{F}}<3e{-}7$ and $||0.5\boldsymbol{C}_k-\boldsymbol{X}||_{\mathrm{F}}<7e{-}6$). Also, our experiments on ZCA whitening and vision transformer demonstrate superior performances. In light of these experimental results, we argue that as long as the BP algorithm is accurate enough, the inconsistency between the BP and FP is not an issue.

A.5 Implementation Details

All the source codes are implemented in Pytorch. For the SVD methods, the forward eigendecomposition is performed using the official Pytorch function TORCH.SVD, which calls the LAPACK's routine *gesdd* that uses the Divide-and-Conquer algorithm for the fast calculation.

All the numerical tests are conducted on a single workstation equipped with a Tesla K40 GPU and a 6-core Intel(R) Xeon(R) GPU @ 2.20GHz.

A.5.1 Training Settings of ZCA Whitening

Suggested by Wang et al. (2020), we truncate the Taylor polynomial to degree 20 for SVD-Taylor. To make Padé approximant match the same degree with Taylor polynomial, we set the degree of both numerator and denominator to 10 for SVD-Padé. For SVD-PI, the iteration times are also set as 20. For the NS iteration, according to the setting in Li et al. (2018); Huang et al. (2018), we set the iteration times to 5. The other experimental settings follow the implementation in Wang et al. (2021). We use the workstation equipped with a Tesla K40 GPU and a 6-core Intel(R) Xeon(R) GPU @ 2.20GHz for training.

A.5.2 Training Settings of Covariance Pooling

We use 8 Tesla G40 GPUs for distributed training and the NVIDIA Apex mixed-precision trainer is used. Except that the spectral layer uses the single-precision (*i.e.,* float32), other layers use the half-precision (*i.e.,* float16) to accelerate the training. Other implementation details follow the experimental setting of the original So-ViT (Xie et al., 2021).

Following the experiment of covariance pooling for CNNs (Song et al., 2021), the degrees of Taylor polynomial are truncated to 100 for SVD-Taylor, and the degree of both the numerator and denominator of Padé approximants are set to 50 for SVD-Padé. The iteration times of SVD-PI are set to 100. In the experiment of covariance pooling, more terms of the Taylor series are used because the covariance pooling meta-layer requires more accurate gradient estimation (Song et al., 2021).

For the SVD-based methods, usually the double-precision is required to ensure an effective numerical representation of the eigenvalues. Using a lower precision would make the model fail to converge at the beginning of the training (Song et al., 2021). To resolve this issue, we first apply the NS iteration to train the network for 50 epochs, then switch to the corresponding SVD method and continue the training till the end. This hybrid approach can avoid the non-convergence of the SVD methods at the beginning of the training phase.

A.6 Extension to Inverse Square Root

Although we use the LU factorization to derive the inverse square root in the paper for comparison fairness, our proposed method can naturally extend to the inverse square root. As the matrix square root $\boldsymbol{A}^{\frac{1}{2}}$ of our MPA is calculated as $\sqrt{||\boldsymbol{A}||_{\mathrm{F}}}\boldsymbol{Q}_N^{-1}\boldsymbol{P}_M$, the inverse square root can be directly computed as $\frac{1}{\sqrt{||\boldsymbol{A}||_{\mathrm{F}}}}\boldsymbol{P}_M^{-1}\boldsymbol{Q}_N$.

A.7 ABLATION STUDIES

We conduct three ablation studies to illustrate the impact of the degree of power series in the forward pass, the termination criterion during the back-propagation, and the possibility of combining our Lyapunov solver with the SVD and the NS iteration.

A.7.1 DEGREE OF POWER SERIES TO MATCH FOR FORWARD PASS

Table 1 displays the performance of our MPA-Lya for different degrees of power series. As we use more terms of the power series, the approximation error gets smaller and the performance gets steady improvements from the degree $[3, 3]$ to $[5, 5]$. When the degree of our MPA is increased from $[5, 5]$ to $[6, 6]$, there are only marginal improvements. We hence set the forward degrees as $[5, 5]$ for our MPA and as 11 for our MTP as a trade-off between speed and accuracy.

Table 1: Performance of our MPA-Lya versus different degrees of power series to match.

Degrees	Time (ms)	ResNet-18				ResNet-50	
		CIFAR10		CIFAR100		CIFAR100	
		mean±std	min	mean±std	min	mean±std	min
$[3, 3]$	0.80	4.64±0.11	4.54	21.35±0.18	21.20	20.14±0.43	19.56
$[4, 4]$	0.86	4.55±0.08	4.51	21.26±0.22	21.03	19.87±0.29	19.64
$[6, 6]$	0.98	**4.45±0.07**	4.33	**21.09±0.14**	21.04	**19.51±0.24**	19.26
$[5, 5]$	0.93	**4.39±0.09**	**4.25**	**21.11±0.12**	**20.95**	**19.55±0.20**	**19.24**

A.7.2 TERMINATION CRITERION FOR BACKWARD PASS

Table 2 compares the performance of backward algorithms with different termination criteria as well as the exact solution computed by the Bartels-Steward algorithm (BS algorithm) (Bartels & Stewart, 1972). Since the NS iteration has the property of quadratic convergence, the errors $||\boldsymbol{B}_k - \boldsymbol{I}||_{\mathrm{F}}$ and $||0.5\boldsymbol{C}_k - \boldsymbol{X}||_{\mathrm{F}}$ decrease at a larger rate for more iteration times. When we iterate more than 7 times, the error becomes sufficiently neglectable, *i.e.,* the NS iteration almost converges. Moreover, from 8 iterations to 9 iterations, there are no obvious performance improvements. We thus terminate the iterations after iterating 8 times.

The exact gradient calculated by the BS algorithm does not yield the best results. Instead, it only achieves the least fluctuation on ResNet-50 and other results are inferior to our iterative solver. This is because the formulation of our Lyapunov equation is based on the assumption that the accurate matrix square root is computed, but in practice we only compute the approximate one in the forward pass. In this case, calculating *the accurate gradient of the approximate matrix square root* might not necessarily work better than *the approximate gradient of the approximate matrix square root*.

Table 2: Performance of our MPA-Lya versus different iteration times. The residual errors $||\boldsymbol{B}_k - \boldsymbol{I}||$ and $||0.5\boldsymbol{C}_k - \boldsymbol{X}||_{\mathrm{F}}$ are measured based on $10,000$ randomly sampled covariance matrices.

Methods	Time (ms)	$\|\|\boldsymbol{B}_k - \boldsymbol{I}\|\|_{\mathrm{F}}$	$\|\|0.5\boldsymbol{C}_k - \boldsymbol{X}\|\|_{\mathrm{F}}$	ResNet-18				ResNet-50	
				CIFAR10		CIFAR100		CIFAR100	
				mean±std	min	mean±std	min	mean±std	min
BS algorithm	2.34	–	–	4.57±0.10	4.45	21.20±0.23	21.01	**19.60±0.16**	19.55
#iter 5	1.05	≈0.3541	≈0.2049	4.48±0.13	4.31	21.15±0.24	**20.84**	20.03±0.19	19.78
#iter 6	1.23	≈0.0410	≈0.0231	4.43±0.10	4.28	21.16±0.19	20.93	19.83±0.24	19.57
#iter 7	1.43	$\approx 7e{-}4$	$\approx 3.5e{-}4$	4.45±0.11	4.29	21.18±0.20	20.95	19.69±0.20	19.38
#iter 9	1.73	$\approx 2e{-}7$	$\approx 7e{-}6$	**4.40±0.07**	4.28	**21.08±0.15**	20.89	**19.52±0.22**	19.25
#iter 8	1.59	$\approx 3e{-}7$	$\approx 7e{-}6$	**4.39±0.09**	**4.25**	**21.11±0.12**	20.95	**19.55±0.20**	**19.24**

A.7.3 ITERATIVE LYAPUNOV SOLVER AS A GENERAL BACKWARD ALGORITHM

Notice that our proposed iterative Lyapunov solver is a general backward algorithm for computing the matrix square root. That is to say, it should be also compatible with the SVD and NS iteration as the forward pass. Table 3 compares the performance of different methods that use the Lyapunov solver as the backward algorithm. As can be seen, the SVD-Lya can achieve competitive performances compared with other differentiable SVD methods. However, the combination of Lyapunov

solver with the NS iteration, *i.e.,* the NS-Lya, cannot converge on any datasets. Although the NS iteration is applied in both the FP and BP of the NS-Lya, the implementation and the usage are different. For the FP algorithm, the NS iteration is two coupled iterations that use two variables $\boldsymbol{Y}_k$ and $\boldsymbol{Z}_k$ to compute the matrix square root. For the BP algorithm, the NS iteration is defined to compute the matrix sign and only uses the variable $\boldsymbol{Y}_k$. The term $\boldsymbol{Z}_k$ is not involved in the BP and we have no control over the gradient back-propagating through it. We conjecture this might introduce some instabilities to the training process. Despite the analysis, developing an effective remedy is a direction of our future work.

Table 3: Performance comparison of SVD-Lya and NS-Lya.

Methods	Time (ms)	ResNet-18				ResNet-50	
		CIFAR10		CIFAR100		CIFAR100	
		mean±std	min	mean±std	min	mean±std	min
SVD-Lya	4.47	4.45±0.16	**4.20**	21.24±0.24	21.02	**19.41±0.11**	19.26
SVD-PI	3.49	4.59±0.09	4.44	21.39±0.23	21.04	19.94±0.44	19.28
SVD-Taylor	3.41	4.50±0.08	4.40	21.14±0.20	**20.91**	19.81±0.24	19.26
SVD-Padé	3.39	4.65±0.11	4.50	21.41±0.15	21.26	20.25±0.20	19.98
NS-Lya	2.87	–	–	–	–	–	–
NS Iteration	2.96	4.57±0.15	4.37	21.24±0.20	21.01	**19.39±0.30**	**19.01**
MPA-Lya	2.52	**4.39±0.09**	4.25	**21.11±0.12**	20.95	**19.55±0.20**	19.24
MTP-Lya	**2.36**	4.49±0.13	4.31	21.42±0.21	21.24	20.55±0.37	20.12

A.8 Computation Analysis on Matrix Inverse versus Solving Linear System

Suppose we want to compute $\boldsymbol{C}=\boldsymbol{A}^{-1}\boldsymbol{B}$. There are two options available. One is first computing the inverse of $\boldsymbol{A}$ and then calculating the matrix product $\boldsymbol{A}^{-1}\boldsymbol{B}$. The other option is to solve the linear system $\boldsymbol{AC}=\boldsymbol{B}$. This process involves first performing the GPU-friendly LU decomposition to decompose $\boldsymbol{A}$ and then conducting two substitutions to obtain $\boldsymbol{C}$.

Solving the linear system is often preferred over the matrix inverse for accuracy, stability, and speed reasons. When the matrix is ill-conditioned, the matrix inverse is very unstable because the eigenvalue λ_i would become $\frac{1}{\lambda_i}$ after the inverse, which might introduce instability when λ_i is very small. For the LU-based linear system, the solution is still stable for ill-conditioned matrices. As for the accuracy aspect, according to Higham (2008), the errors of the two computation methods are bounded by:

$$\begin{aligned} |\boldsymbol{B}-\boldsymbol{AC}_{inv}| &\leq \alpha|\boldsymbol{A}||\boldsymbol{A}^{-1}||\boldsymbol{B}| \\ |\boldsymbol{B}-\boldsymbol{AC}_{LU}| &\leq \alpha|\boldsymbol{L}||\boldsymbol{U}||\boldsymbol{C}_{LU}| \end{aligned} \tag{4}$$

We roughly have the relation $|\boldsymbol{A}| \approx |\boldsymbol{L}||\boldsymbol{U}|$. So the difference is only related to $|\boldsymbol{A}^{-1}||\boldsymbol{B}|$ and $|\boldsymbol{C}_{LU}|$. When $\boldsymbol{A}$ is ill-conditioned, $|\boldsymbol{A}^{-1}|$ could be very large and the error of $|\boldsymbol{B}-\boldsymbol{AC}_{inv}|$ will be significantly larger than the error of linear system. About the speed, the matrix inverse option takes about $\frac{14}{3}n^3$ flops, while the linear system consumes about $\frac{2}{3}n^3+2n^2$ flops. Consider the fact that LU is much more GPU-friendly than the matrix inverse. The actual speed of the linear system could even be faster than the theoretical analysis.

For the above reasons, solving a linear system is a better option than matrix inverse when we need to compute $\boldsymbol{A}^{-1}\boldsymbol{B}$.

A.9 Speed Comparison on Larger Matrices

In the paper, we only compare the speed of each method till the matrix dimension is 128. Here we add the comparison on larger matrices till the dimension 1024. We choose the maximal dimension as 1024 because it should be large enough to cover the applications in deep learning. Table 4 compares the speed of our methods against the NS iteration.

As can be observed, our MPA-Lya is consistently faster than the NS iteration. When the matrix dimension increases, the computation budget of solving the linear system for deriving our MPA increases, but the cost of matrix multiplication also increases. Consider the huge amount of matrix

Table 4: Time consumption (ms) for a single matrix whose dimension is larger than 128. The measurement is based on $10,000$ randomly sampled matrices.

Matrix Dimension	$1\times256\times256$	$1\times512\times512$	$1\times1024\times1024$
MTP-Lya	4.88	5.43	7.32
MPA-Lya	5.56	7.47	11.28
NS iteration	6.28	9.86	12.69

multiplications of NS iteration. The computational cost of the NS iteration grows at a larger speed compared with our MPA-Lya.

A.10 Extra Experiment on Global Covariance Pooling for CNNs

Table 5: Comparison of validation accuracy on ImageNet (Deng et al., 2009) and ResNet-50 (He et al., 2016). The time consumption is measured on a single step (FP+BP). We follow Song et al. (2021) for all the experimental settings.

Methods	Covariance Size ($B\times C\times C$)	Time (ms)	top-1 acc (%)	top-5 acc (%)
MPA-Lya	$256\times256\times256$	110.61	77.13	93.45
NS iteration		164.43	77.19	93.40

To evaluate the performance of our MPA-Lya on batched larger matrices, we conduct another experiment on the task of Global Covariance Pooling (GCP) for CNNs (Li et al., 2017; 2018; Song et al., 2021). In the GCP model, the matrix square root of the covariance of the final convolutional feature is fed into the FC layer for exploiting the second-order statistics. Table 5 presents the speed and performance comparison. As can be seen, the performance of our MPA-Lya is very competitive against the NS iteration, but our MPA-Lya is about 1.5X faster.

A.11 Defect of Padé Approximants

When there is the presence of spurious poles, the Padé approximants do have the well-known defects of instability. The spurious poles mean that when the approximated function has very close poles and zeros, the corresponding Padé approximants will also have close poles and zeros. Consequently, the Padé approximants will become very unstable in the region of defects (*i.e.,* when the input is in the neighborhood of poles and zeros). Generalized to the matrix case, the spurious poles can happen when the determinant of the matrix denominator is zero (*i.e.* $\det\left(\boldsymbol{Q}_N\right)=0$).

However, in our case, the approximated function is $(1-z)^{\frac{1}{2}}$ for $|z|<1$, which only has one zero at $z=1$ and does not have any poles. Therefore, the spurious pole does not exist in our approximation and there are no defects of our Padé approximants.

We can also prove this claim for the matrix case. Consider the denominator of our Padé approximants:

$$\boldsymbol{Q}_N=\boldsymbol{I}-\sum_{n=1}^{N}q_n(\boldsymbol{I}-\frac{\boldsymbol{A}}{||\boldsymbol{A}||_{\mathrm{F}}})^n \tag{5}$$

Its determinant is calculated as:

$$\det\left(\boldsymbol{Q}_N\right)=\prod_{i=1}(1-\sum_{n=1}^{N}q_n(1-\frac{\lambda_i}{\sqrt{\sum_i\lambda_i^2}})^n) \tag{6}$$

The coefficients q_n of our $[5,5]$ Padé approximant are pre-computed as $[2.25,-1.75,0.54675,-0.05859375,0.0009765625]$. Let x_i denotes $(1-\frac{\lambda_i}{\sqrt{\sum_i\lambda_i^2}})$. Then x_i is in the range of $[0,1]$, and we have:

$$\begin{gathered}f(x_i)=1-2.25x_i+1.75x_i^2-0.54675x_i^3+0.05859375x_i^4-0.0009765625x_i^5\\ \det\left(\boldsymbol{Q}_N\right)=\prod_{i=1}(f(x_i))\end{gathered} \tag{7}$$

The polynomial $f(x_i)$ does not have any zero in the range of $x \in [0, 1]$. The minimal is 0.0108672 when $x = 1$. This implies that $\det(\boldsymbol{Q}_N) \neq 0$ always holds for any $\boldsymbol{Q}_N$ and our Padé approximants do not have any pole. Accordingly, there will be no spurious poles and defects. Hence, our MPA is deemed stable. Throughout our experiments, we do not encounter any instability issue of our MPA.

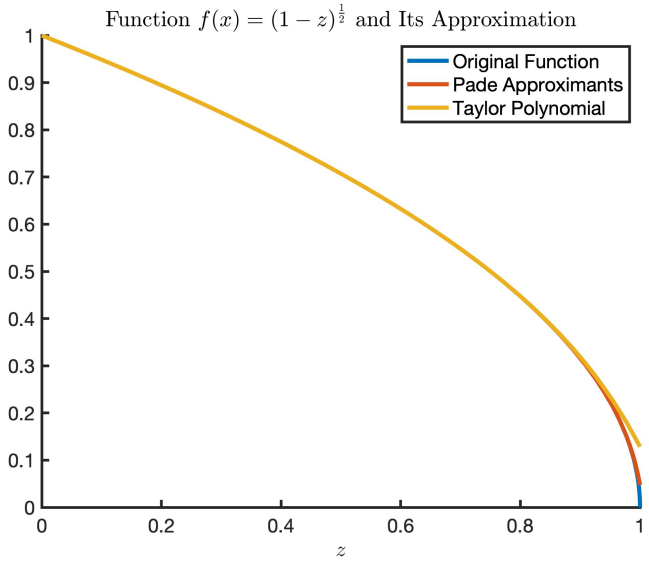

Figure 2: The function $(1-z)^{\frac{1}{2}}$ in the range of $|z| < 1$ and its approximation including Taylor polynomial and Padé approximants. The two approximation techniques do not have any spurious poles or defect regions.

Fig. 2 depicts the function $(1-z)^{\frac{1}{2}}$ and the corresponding approximation of Taylor polynomial and Padé approximants. As can be seen, our approximating techniques do not suffer from any spurious poles or defect regions, and the stability is guaranteed. Moreover, when z is close to 1, the Taylor polynomial gets a relatively large approximation error but the Padé approximants still give a reasonable approximation. This confirms the superiority of our MPA.

A.12 Padé Coefficients of Other Degrees and Their Stability

The property of stability also generalizes to diagonal Padé approximants of other degrees as there are no poles in the original function $(1-z)^{\frac{1}{2}}$ for $|z| < 1$. To better illustrate this point, here we attach the Padé coefficients from the degree $[3, 3]$ to the degree $[6, 6]$. The numerator p_m is:

$$\begin{aligned} p_3 &= [1.75, -0.875, 0.109375]. \\ p_4 &= [2.25, -1.6875, 0.46875, -0.03515625]. \\ p_5 &= [2.75, -2.75, 1.203125, -0.21484375, 0.0107421875]. \\ p_6 &= [3.25, -4.0625, 2.4375, -0.7109375, 0.0888671875, -0.003173828125]. \end{aligned} \tag{8}$$

And the corresponding denominator q_n is:

$$\begin{aligned} q_3 &= [1.25, -0.375, 0.015625]. \\ q_4 &= [1.75, -0.9375, 0.15625, -0.00390625]. \\ q_5 &= [2.25, -1.75, 0.54675, -0.05859375, 0.0009765625]. \\ q_6 &= [2.75, -2.8125, 1.3125, -0.2734375, 0.0205078125, -0.000244140625]. \end{aligned} \tag{9}$$

Similar to the deduction in Eqs. (5) to (7), we can get the polynomial for deriving $\det(\boldsymbol{Q}_N)$ as:

$$\begin{aligned} f(x_i)_{q_3} &= 1 - 1.25x_i + 0.375x_i^2 - 0.015625x_i^3 \\ f(x_i)_{q_4} &= 1 - 1.75x_i + 0.9375x_i^2 - 0.15625x_i^3 + 0.00390625x_i^4 \\ f(x_i)_{q_5} &= 1 - 2.25x_i + 1.75x_i^2 - 0.54675x_i^3 + 0.05859375x_i^4 - 0.0009765625x_i^5 \\ f(x_i)_{q_6} &= 1 - 2.75x_i + 2.8125x_i^2 - 1.3125x_i^3 + 0.2734375x_i^4 - 0.0205078125x_i^5 + 0.000244140625x_i^6 \end{aligned} \tag{10}$$

It can be easily verified that all of the polynomials monotonically decrease in the range of $x_i \in [0, 1]$ and have their minimal at $x_i = 1$ as:

$$\begin{aligned} \min_{x_i \in [0,1]} f(x_i)_{q_3} &= 0.109375. \\ \min_{x_i \in [0,1]} f(x_i)_{q_4} &= 0.03515625. \\ \min_{x_i \in [0,1]} f(x_i)_{q_5} &= 0.0108672. \\ \min_{x_i \in [0,1]} f(x_i)_{q_6} &= 0.003173828125. \end{aligned} \tag{11}$$

As indicated above, all the polynomials are positive and we have $\det(\boldsymbol{Q}_N) \neq 0$ consistently holds for the degree $N = 3, 4, 5, 6$.

REFERENCES

Richard H. Bartels and George W Stewart. Solution of the matrix equation ax+ xb= c [f4]. *Communications of the ACM*, 15(9):820–826, 1972.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2020.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

Nicholas J Higham. *Functions of matrices: theory and computation*. SIAM, 2008.

Lei Huang, Dawei Yang, Bo Lang, and Jia Deng. Decorrelated batch normalization. In *CVPR*, 2018.

Lei Huang, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. Iterative normalization: Beyond standardization towards efficient whitening. In *CVPR*, 2019.

Peihua Li, Jiangtao Xie, Qilong Wang, and Wangmeng Zuo. Is second-order information helpful for large-scale visual recognition? In *ICCV*, 2017.

Peihua Li, Jiangtao Xie, Qilong Wang, and Zilin Gao. Towards faster training of global covariance pooling networks by iterative matrix square root normalization. In *CVPR*, 2018.

Günther Schulz. Iterative berechung der reziproken matrix. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 13(1):57–59, 1933.

Yue Song, Nicu Sebe, and Wei Wang. Why approximate matrix square root outperforms accurate svd in global covariance pooling? In *ICCV*, 2021.

Qilong Wang, Jiangtao Xie, Wangmeng Zuo, Lei Zhang, and Peihua Li. Deep cnns meet global covariance pooling: Better representation and generalization. *TPAMI*, 2020.

Wei Wang, Zheng Dang, Yinlin Hu, Pascal Fua, and Mathieu Salzmann. Backpropagation-friendly eigendecomposition. In *NeurIPS*, 2019.

Wei Wang, Zheng Dang, Yinlin Hu, Pascal Fua, and Mathieu Salzmann. Robust differentiable svd. *TPAMI*, 2021.

Jiangtao Xie, Ruiren Zeng, Qilong Wang, Ziqi Zhou, and Peihua Li. So-vit: Mind visual tokens for vision transformer. *arXiv preprint arXiv:2104.10935*, 2021.