Pass-Efficient Randomized SVD with Boosted Accuracy

Xu Feng[1], Wenjian Yu[1]✉*, and Yuyang Xie[1]

Dept. Computer Science Tech., BNRist, Tsinghua University, Beijing, China
fx17@mails.tsinghua.edu.cn, yu-wj@tsinghua.edu.cn,
xyy18@mails.tsinghua.edu.cn

Abstract. Singular value decomposition (SVD) is a widely used tool in data analysis and numerical linear algebra. Computing truncated SVD of a very large matrix encounters difficulty due to excessive time and memory cost. In this work, we aim to tackle this difficulty and enable accurate SVD computation for the large data which cannot be loaded into memory. We first propose a randomized SVD algorithm with fewer passes over the matrix. It reduces the passes in the basic randomized SVD by half, almost not sacrificing accuracy. Then, a shifted power iteration technique is proposed to improve the accuracy of result, where a dynamic scheme of updating the shift value in each power iteration is included. Finally, collaborating the proposed techniques with several accelerating skills, we develop a Pass-efficient randomized SVD (PerSVD) algorithm for efficient and accurate treatment of large data stored on hard disk. Experiments on synthetic and real-world data validate that the proposed techniques largely improve the accuracy of randomized SVD with same number of passes over the matrix. With 3 or 4 passes over the data, PerSVD is able to reduce the error of SVD result by three or four orders of magnitude compared with the basic randomized SVD and single-pass SVD algorithms, with similar or less runtime and less memory usage.

Keywords: Singular Value Decomposition · Shifted Power Iteration · Random Embedding · Pass-Efficient Algorithm.

1 Introduction

Truncated singular value decomposition (SVD) has broad applications in data analysis and machine learning, such as dimension reduction, matrix completion, and information retrieval. However, for the large and high-dimensional input data from social network analysis, natural language processing and recommender system, etc, computing truncated SVD often consumes tremendous computational resource.

A conventional method of computing truncated SVD, i.e. the first k largest singular values and corresponding singular vectors, is using `svds` in Matlab [3]. In `svds`, Lanczos bidiagonal process is used to compute the truncated SVD [3].

* Corresponding author. This work is supported by NSFC under grant No. 61872206.

Although there are variant algorithms of `svds`, like `lansvd` in PROPACK [11], `svds` is still most robust and runs fastest in most scenarios. However, `svds` needs several times of k times passes over the matrix to produce result, which is not efficient to deal with large data matrices which cannot be stored in RAM. To tackle the difficulty of handling large matrix, approximate algorithms for truncated SVD have been proposed, which consume less time, less memory and fewer passes over input matrix while sacrificing little accuracy [9, 15, 2, 20, 13, 6, 21, 22]. Among them, a class of randomized methods gains a lot of attention which is based on random embedding through multiplying a random matrix [14]. The randomized method obtains a near-optimal low-rank decomposition of the matrix, and has the performance advantages over classical methods, in terms of computational time, pass efficiency and parallelizability. The comprehensive presentation of relevant techniques and theories can be found in [9, 14].

When data is too large to be stored in RAM, traditional truncated SVD algorithms are not efficient, if not impossible, to deal with data stored in slow memory (hard disk). The single-pass SVD algorithms [19, 8, 4, 21, 18] can tackle this difficulty. And, they are also suitable to handle streaming data. Among these algorithms, Tropp's single-pass SVD algorithm in [19] is the state-of-the-art. Although Tropp's single-pass SVD algorithm performs well on matrices with singular values decaying very fast, it results in large error while handling matrices with slow decay of singular values. Therefore, how to accurately compute the truncated SVD of a large matrix stored on hard disk on a computer with limited memory is a problem.

In this paper, we aim to tackle the difficulty of handling large matrix stored on hard disk or slow memory. We propose a pass-efficient randomized SVD (PerSVD) algorithm which accurately computes SVD of large matrix stored on hard disk with less memory and affordable time. Major contributions and results are summarized as follows.

- We propose a technique to reduce the number of passes over the matrix in the basic randomized SVD algorithm. It takes advantage of the row-major format of the matrix and reads it row by row to build $\mathbf{A\Phi}$ and $\mathbf{A}^{\mathrm{T}}\mathbf{A\Phi}$ with just one pass over matrix. With this algorithm, the passes over the matrix in the basic randomized SVD algorithm is reduced by half, with negligible loss of accuracy.
- Inspired by the shift technique in the power method [7], we propose to use the shift skill in the power iteration called shifted power iteration to improve the accuracy of results. A dynamic scheme of updating the shift value in each power iteration is proposed to optimize the performance of the shifted power iteration. This facilitates a pass-efficient randomized SVD algorithm, i.e. PerSVD, which accurately computes truncated SVD of large matrix on a limited-memory computer.
- Experiments on synthetic and real large data demonstrate that the proposed techniques are all beneficial to improve the accuracy of result with same number of passes over the matrix. With same 4 passes the over matrix, the result computed with PerSVD is up to **20,318X** more accurate than that

obtained with the basic randomized SVD. And, the proposed PerSVD with 3 passes over the matrix consumes just 16%-26% memory of Tropp's algorithm [19] while producing more accurate results (with up to **7,497X** reduction of error), with less runtime. For the FERET data stored as a 150GB file, PerSVD just costs 12 minutes and 1.9 GB memory to compute the truncated SVD ($k = 100$) with 3 passes over the data.

2 Preliminaries

Below we follow the Matlab conventions to specify indices of matrix and functions.

2.1 Basics of Truncated SVD

The economic SVD of a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ $(m \geq n)$ can be stated as:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}, \tag{1}$$

where $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_n]$ and $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n]$ are orthogonal or orthonormal matrices, representing the left and right singular vectors of $\mathbf{A}$, respectively. The $n \times n$ diagonal matrix $\mathbf{\Sigma}$ contains the singular values $(\sigma_1, \sigma_2, \cdots, \sigma_n)$ of $\mathbf{A}$ in descending order. The truncated SVD $\mathbf{A}_k$ can be derived, which is an approximation of $\mathbf{A}$:

$$\mathbf{A} \approx \mathbf{A}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^{\mathrm{T}}, \quad k < \min(m, n), \tag{2}$$

where $\mathbf{U}_k$ and $\mathbf{V}_k$ are matrices with the first k columns of $\mathbf{U}$ and $\mathbf{V}$ respectively, and the diagonal matrix $\mathbf{\Sigma}_k$ is the $k \times k$ upper-left submatrix of $\mathbf{\Sigma}$. Notice that, $\mathbf{A}_k$ is the best rank-k approximation of $\mathbf{A}$ in both spectral and Frobenius norm [5].

2.2 Randomized SVD Algorithm with Power Iteration

The basic randomized SVD algorithm [9] can be described as Algorithm 1, where the power iteration in Step 3 through 6 is for improving the accuracy of result. In Alg. 1, $\mathbf{\Omega}$ is a Gaussian random matrix. Although other kinds of random matrix can replace $\mathbf{\Omega}$ to reduce the computational cost of $\mathbf{A\Omega}$, they bring some sacrifice on accuracy. The orthonormalization operation "orth()" can be implemented with a call to a packaged QR factorization.

If there is no power iteration, the $m \times l$ orthonormal matrix $\mathbf{Q} = \mathrm{orth}(\mathbf{A\Omega})$ approximates the basis of dominant subspace of $range(\mathbf{A})$, i.e., $span\{\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_l\}$. So, $\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^{\mathrm{T}}\mathbf{A}$, i.e. $\mathbf{A} \approx \mathbf{QB}$ according to Step 8. By performing the economic SVD, i.e. (1), on the "short-and-fat" $l \times n$ matrix $\mathbf{B}$, one finally obtains the approximate truncated SVD of $\mathbf{A}$. Using the power iteration in Steps 3-6, one obtains $\mathbf{Q} = \mathrm{orth}((\mathbf{A}\mathbf{A}^{\mathrm{T}})^p \mathbf{A\Omega})$. This makes $\mathbf{Q}$ better approximate the basis of dominant subspace of $range((\mathbf{A}\mathbf{A}^{\mathrm{T}})^p \mathbf{A})$, same as that of $range(\mathbf{A})$, because

$(\mathbf{A}\mathbf{A}^{\mathrm{T}})^p\mathbf{A}$'s singular values decay more quickly than those of $\mathbf{A}$ [9]. Therefore, the resulted singular values and vectors have better accuracy, and the larger p makes more accurate results and more computational cost as well.

The orthonormalization is practically used in the power iteration steps to alleviate the round-off error in the floating-point computation. It can be performed after every other matrix-matrix multiplication to save computational cost with little sacrifice on accuracy [20, 6]. This turns Steps 2-7 of Alg. 1 to

2: $\mathbf{Q} \leftarrow \mathrm{orth}(\mathbf{A}\mathbf{\Omega})$
3: **for** $j \leftarrow 1, 2, \cdots p$ **do**
4: $\quad\mathbf{Q} \leftarrow \mathrm{orth}(\mathbf{A}\mathbf{A}^{\mathrm{T}}\mathbf{Q})$
5: **end for**

Notice that the original Step 7 in Alg. 1 is dropped. We use the floating-point operation (*flop*) to specify the time cost of algorithms. Suppose the *flop* count of multiplication of $\mathbf{M} \in \mathbb{R}^{m\times l}$ and $\mathbf{N} \in \mathbb{R}^{l\times l}$ is $C_{mul}ml^2$. Here, C_{mul} reflects one addition and one multiplication. Thus, the *flop* count of Alg. 1 is

$$\mathrm{FC}_1 = (2p+2)C_{mul}mnl + (p+1)C_{qr}ml^2 + C_{svd}nl^2 + C_{mul}ml^2, \qquad (3)$$

where $(2p+2)C_{mul}mnl$ reflects $2p+2$ times matrix-matrix multiplication on $\mathbf{A}$, $(p+1)C_{qr}ml^2$ reflects $p+1$ times QR factorization on $m \times l$ matrix and $C_{svd}nl^2 + C_{mul}ml^2$ reflects SVD and matrix-matrix multiplication in Step 9 and 10.

2.3 Tropp's Single-Pass SVD Algorithm

On a machine with limited memory, single-pass SVD algorithms can be used to handle very large data stored on hard disk [19, 8, 4, 21, 18]. Although the single-pass algorithm in [18] achieves lower computational complexity, it is more

Algorithm 1 Basic randomized SVD with power iteration

Input: $\mathbf{A} \in \mathbb{R}^{m\times n}$, rank parameter k, oversampling parameter l ($l \geq k$), power parameter p
Output: $\mathbf{U} \in \mathbb{R}^{m\times k}$, $\mathbf{S} \in \mathbb{R}^{k\times k}$, $\mathbf{V} \in \mathbb{R}^{n\times k}$
1: $\mathbf{\Omega} \leftarrow \mathrm{randn}(n, l)$
2: $\mathbf{Q} \leftarrow \mathbf{A}\mathbf{\Omega}$
3: **for** $j \leftarrow 1, 2, \cdots, p$ **do**
4: $\quad\mathbf{G} \leftarrow \mathbf{A}^{\mathrm{T}}\mathbf{Q}$
5: $\quad\mathbf{Q} \leftarrow \mathbf{A}\mathbf{G}$
6: **end for**
7: $\mathbf{Q} \leftarrow \mathrm{orth}(\mathbf{Q})$
8: $\mathbf{B} \leftarrow \mathbf{Q}^{\mathrm{T}}\mathbf{A}$
9: $[\mathbf{U}, \mathbf{S}, \mathbf{V}] \leftarrow \mathrm{svd}(\mathbf{B}, 'econ')$
10: $\mathbf{U} \leftarrow \mathbf{Q}\mathbf{U}$
11: $\mathbf{U} \leftarrow \mathbf{U}(:, 1{:}k), \mathbf{S} \leftarrow \mathbf{S}(1{:}k, 1{:}k), \mathbf{V} \leftarrow \mathbf{V}(:, 1{:}k)$

suitable for the matrix with fast decay of singular values and $m \ll n$. Tropp's single-pass SVD algorithm [19] is the state-of-the-art single-pass SVD algorithm, with lower approximation error compared with it's predecessors given the same sketch sizes. Tropp's single-pass SVD algorithm (Alg. 2) first constructs several sketches of the input matrix $\mathbf{A}$ in Step 4-7. Then, QR factorization is performed in Step 8 and 9 to compute the orthonormal basis of row and column space of $\mathbf{A}$, respectively. Then, matrix $\mathbf{C}$ is computed to approximate the core of $\mathbf{A}$ in Step 10. Notice that '/' and '\' represent the left and right division in Matlab, respectively. This is followed by SVD in Step 11 for computing the singular values and vectors of $\mathbf{C}$. Finally, the singular vectors are computed by projecting the orthonormal matrices to the row and column spaces of original matrix in Step 12.

Algorithm 2 Tropp's single-pass SVD algorithm

Input: $\mathbf{A} \in \mathbb{R}^{m\times n}$, rank parameter k
Output: $\mathbf{U} \in \mathbb{R}^{m\times k}$, $\mathbf{S} \in \mathbb{R}^{k\times k}$, $\mathbf{V} \in \mathbb{R}^{n\times k}$
1: $r \leftarrow 4k+1$, $s \leftarrow 2r+1$
2: $\boldsymbol{\Upsilon} \leftarrow \text{randn}(r,m)$, $\boldsymbol{\Omega} \leftarrow \text{randn}(r,n)$, $\boldsymbol{\Phi} \leftarrow \text{randn}(s,m)$, $\boldsymbol{\Psi} \leftarrow \text{randn}(s,n)$
3: $\mathbf{X} \leftarrow \text{zeros}(r,n)$, $\mathbf{Y} \leftarrow \text{zeros}(m,r)$, $\mathbf{Z} \leftarrow \text{zeros}(s,s)$
4: **for** $j \leftarrow 1, 2, \cdots, m$ **do**
5: $\quad \mathbf{a}_i$ is the i-th row of $\mathbf{A}$
6: $\quad \mathbf{X} \leftarrow \mathbf{X} + \boldsymbol{\Upsilon}(i,:)\mathbf{a}_i$, $\mathbf{Y}(i,:) \leftarrow \mathbf{a}_i\boldsymbol{\Omega}$, $\mathbf{Z} \leftarrow \mathbf{Z} + \boldsymbol{\Phi}(i,:)\mathbf{a}_i\boldsymbol{\Psi}^{\mathrm{T}}$
7: **end for**
8: $[\mathbf{Q}, \sim] \leftarrow \text{qr}(\mathbf{Y}, 0)$
9: $[\mathbf{P}, \sim] \leftarrow \text{qr}(\mathbf{X}^{\mathrm{T}}, 0)$
10: $\mathbf{C} \leftarrow ((\boldsymbol{\Phi}\mathbf{Q})/\mathbf{Z})\backslash(\boldsymbol{\Psi}\mathbf{P}^{\mathrm{T}})$
11: $[\mathbf{U}, \mathbf{S}, \mathbf{V}] \leftarrow \text{svd}(\mathbf{C}, 'econ')$
12: $\mathbf{U} \leftarrow \mathbf{Q}\mathbf{U}(:, 1\!:\!k)$, $\mathbf{S} \leftarrow \mathbf{S}(1\!:\!k, 1\!:\!k)$, $\mathbf{V} \leftarrow \mathbf{P}\mathbf{V}(:, 1\!:\!k)$

The peak memory usage of Alg. 2 is $(m+n)(2r+s)\times 8 \approx 16(m+n)k\times 8$ bytes, which is caused by all matrices computed in Alg. 2. And, the *flop* count of Alg. 2 is

$$\begin{aligned}\text{FC}_2 &= C_{mul}m(2nr+ns+s^2) + C_{qr}(m+n)r^2 + C_{mul}(m+n)rs + 2C_{solve}s^2k \\ &\quad + C_{svd}r^3 + C_{mul}(m+n)rk \\ &\approx C_{mul}m(2nr+ns+s^2) + C_{qr}(m+n)r^2 + C_{mul}(m+n)r(s+k) \\ &\approx 16C_{mul}mnk + 100C_{mul}mk^2 + 36C_{mul}nk^2 + 16C_{qr}(m+n)k^2,\end{aligned} \tag{4}$$

where $C_{mul}m(2nr+ns+s^2)$ reflects the matrix-matrix multiplication in Step 4-7, $C_{qr}(m+n)r^2$ reflects the QR factorization in Step 8 and 9, $C_{mul}(m+n)rs + 2C_{solve}s^2k$ reflects the matrix-matrix multiplication and the solve operation in Step 10 and $C_{svd}r^3 + C_{mul}(m+n)rk$ reflects the SVD and matrix-matrix multiplication in Step 11 and 12. When $k \ll \min(m,n)$, we can drop the $2C_{solve}s^2k$ and $C_{svd}r^3$ in FC_2.

It should be pointed out that the Tropp's algorithm does not perform well on matrices with slow decay of singular values, exhibiting large error on computed singular values/vectors. However, the matrices with slow decay of singular values are common in real applications.

3 Pass-Efficient SVD with Shifted Power Iteration

In this section, we develop a pass-efficient randomized SVD algorithm named PerSVD. Firstly, we develop a pass-efficient scheme to reduce the passes over $\mathbf{A}$ within basic randomized SVD algorithm. Secondly, inspired by the shift technique in the power method [7], we propose a technique of shifted power iteration to improve the accuracy of result. Finally, combining with the proposed shift updating scheme in each power iteration, we describe the pass-efficient PerSVD algorithm which is able to accurately compute SVD of large matrix on hard disk with less memory and affordable time.

3.1 Randomized SVD with Fewer Passes

Suppose $\mathbf{a}_i$ is the i-th ($i \leq m$) row of matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, and $\mathbf{\Phi} \in \mathbb{R}^{n\times l}$. To compute $\mathbf{Y} = \mathbf{A\Phi}$ and $\mathbf{W} = \mathbf{A}^{\mathrm{T}}\mathbf{A\Phi}$ with $\mathbf{a}_i$, we have

$$\mathbf{Y}(i,:) = \mathbf{a}_i\mathbf{\Phi}, \mathbf{W} = \mathbf{A}^{\mathrm{T}}\mathbf{Y} = \sum_{i=1}^{m} \mathbf{a}_i^{\mathrm{T}}\mathbf{Y}(i,:), \tag{5}$$

which reflects the data stored in the row-major format can be read once to compute $\mathbf{Y} = \mathbf{A\Phi}$ and $\mathbf{W} = \mathbf{A}^{\mathrm{T}}\mathbf{A\Phi}$. This idea is similar to that employed in [21, 22]. Below, we derive that it can be repeatedly used to compute the $(\mathbf{A}^{\mathrm{T}}\mathbf{A})^p\mathbf{\Omega}$ in the basic randomized SVD algorithm with power iteration.

With $\mathbf{Y}$ and $\mathbf{W}$, we can develop the formulation of $\mathbf{Q}$ and $\mathbf{B}$ in Step 7 and 8 of Alg. 1. Suppose $\mathbf{\Phi} = (\mathbf{A}^{\mathrm{T}}\mathbf{A})^p\mathbf{\Omega}$, $\mathbf{Y} = \mathbf{A\Phi}$, $\mathbf{W} = \mathbf{A}^{\mathrm{T}}\mathbf{A\Phi}$ and $[\mathbf{Q}, \tilde{\mathbf{S}}, \tilde{\mathbf{V}}] \leftarrow \mathrm{svd}(\mathbf{Y}, \text{'econ'})$. Combining the fact $\mathbf{W} = \mathbf{A}^{\mathrm{T}}\mathbf{Y}$, we can derive

$$\mathbf{W} = \mathbf{A}^{\mathrm{T}}\mathbf{Y} = \mathbf{A}^{\mathrm{T}}\mathbf{Q}\tilde{\mathbf{S}}\tilde{\mathbf{V}}^{\mathrm{T}} \Rightarrow \mathbf{W}\tilde{\mathbf{V}}\tilde{\mathbf{S}}^{-1} = \mathbf{A}^{\mathrm{T}}\mathbf{Q} \tag{6}$$

which implies $\mathbf{Q}(\mathbf{W}\tilde{\mathbf{V}}\tilde{\mathbf{S}}^{-1})^{\mathrm{T}} = \mathbf{Q}\mathbf{Q}^{\mathrm{T}}\mathbf{A} \approx \mathbf{A}$. Because $\mathbf{Q}$ is the orthonormalization of $\mathbf{A}(\mathbf{A}^{\mathrm{T}}\mathbf{A})^p\mathbf{\Omega}$, this approximation performs with same accuracy as $\mathbf{A} \approx \mathbf{QB}$ in Alg. 1. With (5) and (6) combined, the randomized SVD with fewer passes is proposed and described as Algorithm 3. Now, the randomized SVD with p power iteration just needs $p+1$ passes over $\mathbf{A}$, which reduces half of the passes in Alg. 1.

The above deduction reveals Alg. 3 is mathematically equivalent to the basic randomized SVD (the modified Alg. 1) in exact arithmetic. In the practice considering numerical error, the computational results of the both algorithms are very close which means Alg. 3 largely reduces the passes with just negligible loss of accuracy. Besides, it is easy to see that we can read multiple rows once to compute $\mathbf{Y}$ and $\mathbf{W}$ by (5).

Algorithm 3 Randomized SVD with fewer passes the over matrix

Input: $\mathbf{A} \in \mathbb{R}^{m\times n}$, k, l $(l \geq k)$, p
Output: $\mathbf{U} \in \mathbb{R}^{m\times k}$, $\mathbf{S} \in \mathbb{R}^{k\times k}$, $\mathbf{V} \in \mathbb{R}^{n\times k}$

$\mathbf{\Omega} \leftarrow \text{randn}(n, l)$
$\mathbf{Q} \leftarrow \text{orth}(\mathbf{\Omega})$
for $j = 1, 2, \cdots, p+1$ **do**
  $\mathbf{W} \leftarrow \text{zeros}(n, l)$
  for $i = 1, 2, \cdots, m$ **do**
    $\mathbf{a}_i$ is the i-th row of matrix $\mathbf{A}$
    $\mathbf{Y}(i,:) \leftarrow \mathbf{a}_i\mathbf{Q}$, $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{a}_i^{\mathrm{T}}\mathbf{Y}(i,:)$
  end for
  if $j == p+1$ **break**
  $\mathbf{Q} \leftarrow \text{orth}(\mathbf{W})$
end for
$[\mathbf{Q}, \tilde{\mathbf{S}}, \tilde{\mathbf{V}}] \leftarrow \text{svd}(\mathbf{Y}, 'econ')$
$\mathbf{B} \leftarrow \tilde{\mathbf{S}}^{-1}\tilde{\mathbf{V}}^{\mathrm{T}}\mathbf{W}^{\mathrm{T}}$
$[\mathbf{U}, \mathbf{S}, \mathbf{V}] \leftarrow \text{svd}(\mathbf{B}, 'econ')$
$\mathbf{U} \leftarrow \mathbf{Q}\mathbf{U}(:, 1:k), \mathbf{S} \leftarrow \mathbf{S}(1:k, 1:k), \mathbf{V} \leftarrow \mathbf{V}(:, 1:k)$

3.2 The Idea of Shifted Power Iteration

In Alg. 3, the computation $\mathbf{Q} \leftarrow \mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q}$ in power iteration is the same as that in the power method for computing the largest eigenvalue and corresponding eigenvector of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. For the power method, the shift skill can be used to accelerate the convergence of iteration by reducing the ratio between the second largest eigenvalue and the largest one [7]. This inspires our idea of shifted power iteration. To derive our method, we first give two Lemmas [7].

Lemma 1. *For symmetric matrix $\mathbf{A}$, its singular values are the absolute values of its eigenvalues. And, for any eigenvalue λ of $\mathbf{A}$, the left singular vector corresponding to its singular value $|\lambda|$ is the normalized eigenvector for λ.*

Lemma 2. *Suppose matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, and a shift $\alpha \in \mathbb{R}$. For any eigenvalue λ of $\mathbf{A}$, $\lambda - \alpha$ is an eigenvalue of $\mathbf{A} - \alpha\mathbf{I}$, where $\mathbf{I}$ is the identity matrix. And, the eigenspace of $\mathbf{A}$ for λ is the same as the eigenspace of $\mathbf{A} - \alpha\mathbf{I}$ for $\lambda - \alpha$.*

Because $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ is a symmetric positive semi-definite matrix, its singular value is its eigenvalue according to Lemma 1. We use $\sigma_i(\cdot)$ to denote the i-th largest singular value. Along with Lemma 2, we see that $\sigma_i(\mathbf{A}^{\mathrm{T}}\mathbf{A}) - \alpha$ is the eigenvalue of $\mathbf{A}^{\mathrm{T}}\mathbf{A} - \alpha\mathbf{I}$. Then, $|\sigma_i(\mathbf{A}^{\mathrm{T}}\mathbf{A}) - \alpha|$ is the singular value of $\mathbf{A}^{\mathrm{T}}\mathbf{A} - \alpha\mathbf{I}$ according to Lemma 1. This can be illustrated by Fig. 1. Notice that $|\sigma_i(\mathbf{A}^{\mathrm{T}}\mathbf{A}) - \alpha|$ is not necessarily the i-th largest singular value.

For Alg. 1, the decay trend of the first l singular values of handled matrix affects the accuracy of result. If $\sigma_i(\mathbf{A}^{\mathrm{T}}\mathbf{A}) - \alpha > 0$, $(i \leq l)$, and they are the l largest singular values of $\mathbf{A}^{\mathrm{T}}\mathbf{A} - \alpha\mathbf{I}$, these shifted singular values obviously

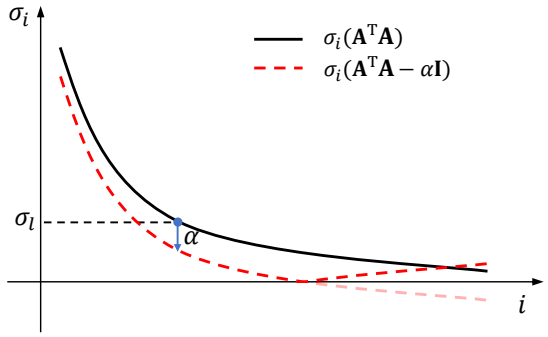


Fig. 1: The illustration of the singular value curves of $\mathbf{A}^\mathrm{T}\mathbf{A}$ and $\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I}$.

exhibit faster decay (see Fig. 1). The following Theorem states when these conditions are satisfied.

Theorem 1. *Suppose positive number* $\alpha \leq \sigma_l(\mathbf{A}^\mathrm{T}\mathbf{A})/2$ *and* $i \leq l$*. Then,* $\sigma_i(\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I}) = \sigma_i(\mathbf{A}^\mathrm{T}\mathbf{A})-\alpha$*, where* $\sigma_i(\cdot)$ *denotes the* i*-th largest singular value. And, the left singular vector corresponding to the* i*-th singular value of* $\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I}$ *is the same as the left singular vector corresponding to the* i*-th singular value of* $\mathbf{A}^\mathrm{T}\mathbf{A}$*.*

The complete proof is in Appendix A.1. The first statement of Theorem 1 is straightforward from Fig. 1. The second statement can be derived from the statements on relationships between eigenvectors and singular vectors in Lemma 1 and 2.

According to Theorem 1, if we choose a shift $\alpha \leq \sigma_l(\mathbf{A}^\mathrm{T}\mathbf{A})/2$ we can change the computation $\mathbf{Q} = \mathbf{A}^\mathrm{T}\mathbf{A}\mathbf{Q}$ to $\mathbf{Q} = (\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I})\mathbf{Q}$ in the power iteration, with the approximated dominant subspace unchanged. We called this *shifted power iteration*. For each step of shifted power iteration, this makes $\mathbf{AQ}$ approximate the basis of dominant subspace of $range(\mathbf{A})$ to a larger extent than executing an original power iteration step, because the singular values of $\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I}$ decay faster than those of $\mathbf{A}^\mathrm{T}\mathbf{A}$. Therefore, the shifted power iteration would improve the accuracy of the randomized SVD algorithm with same power parameter p. The remaining problem is how to set the shift α.

Consider the change of ratio of singular values from $\frac{\sigma_i(\mathbf{A}^\mathrm{T}\mathbf{A})}{\sigma_1(\mathbf{A}^\mathrm{T}\mathbf{A})}$ to $\frac{\sigma_i(\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I})}{\sigma_1(\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I})}$, for $i \leq l$. It is easy to see

$$\frac{\sigma_i(\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I})}{\sigma_1(\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I})} = \frac{\sigma_i(\mathbf{A}^\mathrm{T}\mathbf{A})-\alpha}{\sigma_1(\mathbf{A}^\mathrm{T}\mathbf{A})-\alpha} < \frac{\sigma_i(\mathbf{A}^\mathrm{T}\mathbf{A})}{\sigma_1(\mathbf{A}^\mathrm{T}\mathbf{A})}, \tag{7}$$

if the assumption of α in Theorem 1 holds. And, the larger value of α, the smaller the ratio $\frac{\sigma_i(\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I})}{\sigma_1(\mathbf{A}^\mathrm{T}\mathbf{A}-\alpha\mathbf{I})}$, meaning faster decay of singular values. Therefore, to maximize the effect of shifted power iteration on improving the accuracy, we should choose the shift α as large as possible while satisfying $\alpha \leq \sigma_l(\mathbf{A}^\mathrm{T}\mathbf{A})/2$. Notice that calculating $\sigma_l(\mathbf{A}^\mathrm{T}\mathbf{A})$ is very difficult. Our idea is using the singular

value of $\mathbf{A}^{\mathrm{T}}\mathbf{AQ}$ at the first step of the power iteration to approximate $\sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{A})$ and set the shift α.

Lemma 3. *[10] Let $\mathbf{A}, \mathbf{C} \in \mathbb{R}^{m\times n}$ be given. The following inequality holds for the decreasingly ordered singular values of $\mathbf{A}$, $\mathbf{C}$ and $\mathbf{AC}^{\mathrm{T}}$ ($1 \le i, j \le \min(m, n)$ and $i + j - 1 \le \min(m, n)$)*

$$\sigma_{i+j-1}(\mathbf{AC}^{\mathrm{T}}) \le \sigma_i(\mathbf{A})\sigma_j(\mathbf{C}) \ , \tag{8}$$

and

$$\sigma_{i+j-1}(\mathbf{A} + \mathbf{C}) \le \sigma_i(\mathbf{A}) + \sigma_j(\mathbf{C}) \ . \tag{9}$$

Based on Lemma 3, i.e. (3.3.17) and (3.3.18) in [10], we prove Theorem 2

Theorem 2. *Suppose $\mathbf{Q} \in \mathbb{R}^{m\times l}(l \le m)$ is an orthonormal matrix and $\mathbf{A} \in \mathbb{R}^{m\times m}$. Then, for any $i \le l$*

$$\sigma_i(\mathbf{AQ}) \le \sigma_i(\mathbf{A}). \tag{10}$$

Proof. We append zero columns to $\mathbf{Q}$ to get an $m \times m$ matrix $\mathbf{C}^{\mathrm{T}} = [\mathbf{Q}, \mathbf{0}] \in \mathbb{R}^{m\times m}$. Since $\mathbf{Q}$ is an orthonormal matrix, $\sigma_1(\mathbf{C}) = 1$. According to (8) in Lemma 3,

$$\sigma_i(\mathbf{AC}^{\mathrm{T}}) \le \sigma_i(\mathbf{A})\sigma_1(\mathbf{C}) = \sigma_i(\mathbf{A}). \tag{11}$$

Because $\mathbf{AC}^{\mathrm{T}} = [\mathbf{AQ}, \mathbf{0}]$, for any $i \le l$, $\sigma_i(\mathbf{AQ}) = \sigma_i(\mathbf{AC}^{\mathrm{T}})$. Then, combining (11) we can prove (10).

Suppose $\mathbf{Q} \in \mathbb{R}^{m\times l}$ is the orthonormal matrix in power iteration of Alg. 3. According to Theorem 2,

$$\sigma_i(\mathbf{A}^{\mathrm{T}}\mathbf{AQ}) \le \sigma_i(\mathbf{A}^{\mathrm{T}}\mathbf{A}),\ i \le l, \tag{12}$$

which means we can set $\alpha = \sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{AQ})/2$ to guarantee the requirement of α in Theorem 1 for performing the shifted power iteration. In order to do the orthonormalization for alleviating round-off error and calculate $\sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{AQ})$, we implement "orth(·)" with the economic SVD. This has similar computational cost as using QR factorization, and the resulting matrix of left singular vectors includes the orthonormal basis of same subspace.

So far, we can obtain the value of α at the first step of power iteration, and then we perform $\mathbf{Q} = (\mathbf{A}^{\mathrm{T}}\mathbf{A} - \alpha\mathbf{I})\mathbf{Q}$ in the following iteration steps. Combining the shifted power iteration with fixed shift value, we derive a randomized SVD algorithm with shifted power iteration as Algorithm 4.

3.3 Update Shift in Each Power Iteration

In order to improve the accuracy of Alg. 4, we try to make the shift α as large as possible in each power iteration. Therefore, we further propose a dynamic scheme to set α, which updates α with larger values and thus increases the decay of singular values.

Algorithm 4 Randomized SVD with shifted power iteration

Input: $\mathbf{A} \in \mathbb{R}^{m\times n}$, k, l $(l \geq k)$, p
Output: $\mathbf{U} \in \mathbb{R}^{m\times k}$, $\mathbf{S} \in \mathbb{R}^{k\times k}$, $\mathbf{V} \in \mathbb{R}^{n\times k}$

$\mathbf{\Omega} \leftarrow \text{randn}(n, l)$
$\mathbf{Q} \leftarrow \text{orth}(\mathbf{\Omega})$
$\alpha \leftarrow 0$
for $j = 1, 2, \cdots, p+1$ **do**
  $\mathbf{W} \leftarrow \text{zeros}(n, l)$
  for $i = 1, 2, \cdots, m$ **do**
    $\mathbf{a}_i$ is the i-th row of matrix $\mathbf{A}$
    $\mathbf{Y}(i,:) \leftarrow \mathbf{a}_i\mathbf{Q}$, $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{a}_i^{\mathrm{T}}\mathbf{Y}(i,:)$
  end for
  if $j == p+1$ **break**
  $[\mathbf{Q}, \hat{\mathbf{S}}, \sim] \leftarrow \text{svd}(\mathbf{W} - \alpha\mathbf{Q}, 'econ')$
  if $\alpha == 0$ **then** $\alpha \leftarrow (\hat{\mathbf{S}}(l,l) + \alpha)/2$
end for
$[\mathbf{Q}, \tilde{\mathbf{S}}, \tilde{\mathbf{V}}] \leftarrow \text{svd}(\mathbf{Y}, 'econ')$
$\mathbf{B} \leftarrow \tilde{\mathbf{S}}^{-1}\tilde{\mathbf{V}}^{\mathrm{T}}\mathbf{W}^{\mathrm{T}}$
$[\mathbf{U}, \mathbf{S}, \mathbf{V}] \leftarrow \text{svd}(\mathbf{B}, 'econ')$
$\mathbf{U} \leftarrow \mathbf{Q}\mathbf{U}(:, 1:k), \mathbf{S} \leftarrow \mathbf{S}(1:k, 1:k), \mathbf{V} \leftarrow \mathbf{V}(:, 1:k)$

In the iteration steps of shifted power iteration, it is convenient to calculate the singular values of $(\mathbf{A}^{\mathrm{T}}\mathbf{A}-\alpha\mathbf{I})\mathbf{Q}$. The following Theorems state how to use it to approximate $\sigma_i(\mathbf{A}^{\mathrm{T}}\mathbf{A})$. So, in each iteration step we obtain a valid value of shift and update α with it if we have a larger α.

Theorem 3. *Suppose* $\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{Q} \in \mathbb{R}^{n\times l}(l < n)$ *is an orthonormal matrix and* $0 < \alpha < \sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{A})/2$. *Then,*

$$\sigma_i((\mathbf{A}^{\mathrm{T}}\mathbf{A}-\alpha\mathbf{I})\mathbf{Q}) + \alpha \leq \sigma_i(\mathbf{A}^{\mathrm{T}}\mathbf{A}),\ i \leq l. \tag{13}$$

Proof. For any $i \leq l$,

$$\sigma_i((\mathbf{A}^{\mathrm{T}}\mathbf{A}-\alpha\mathbf{I})\mathbf{Q}) + \alpha \leq \sigma_i(\mathbf{A}^{\mathrm{T}}\mathbf{A}-\alpha\mathbf{I}) + \alpha = \sigma_i(\mathbf{A}^{\mathrm{T}}\mathbf{A}), \tag{14}$$

due to Theorem 2 and Theorem 1.

Theorem 4. *Suppose* $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{Q} \in \mathbb{R}^{n\times l}(l < n)$ *is an orthonormal matrix,* $\alpha^{(0)} = 0$ *and* $\alpha^{(u)} = (\sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q} - \alpha^{(u-1)}\mathbf{Q}) + \alpha^{(u-1)})/2$ *for any* $u > 0$. *Then,* $\alpha^{(0)}, \alpha^{(1)}, \alpha^{(2)}, \cdots$ *are in ascending order.*

Proof. We prove this Theorem using induction.

When $u = 1$, $\alpha^{(1)} = \sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q}) \geq \alpha^{(0)}$.

When $u > 1$, suppose $\alpha^{(u-1)} \geq \alpha^{(u-2)}$. Then, according to (9) in Lemma 3

$$\begin{aligned}\sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q} - \alpha^{(u-2)}\mathbf{Q}) =& \sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q} - \alpha^{(u-1)}\mathbf{Q} + (\alpha^{(u-1)} - \alpha^{(u-2)})\mathbf{Q}) \\ \leq& \sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q} - \alpha^{(u-1)}\mathbf{Q}) + \alpha^{(u-1)} - \alpha^{(u-2)}.\end{aligned} \tag{15}$$

Therefore,

$$\alpha^{(u-1)} = \frac{\sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q} - \alpha^{(u-2)}\mathbf{Q}) + \alpha^{(u-2)}}{2} \leq \frac{\sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q} - \alpha^{(u-1)}\mathbf{Q}) + \alpha^{(u-1)}}{2} = \alpha^{(u)}. \tag{16}$$

According to these equations, this Theorem is proven.

Remark 2. According to the proof of Theorem 4, it can be simply proven that $\alpha^{(0)}, \alpha^{(1)}, \alpha^{(2)}, \cdots$ are in ascending order when $\alpha^{(0)} \leq \alpha^{(1)}$, where $\alpha^{(0)} \geq 0$ and $\alpha^{(u)} = (\sigma_l(\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q} - \alpha^{(u-1)}\mathbf{Q}) + \alpha^{(u-1)})/2$ for any $u > 0$.

So, we can increase α in each shifted power iteration with the following steps.

11: **while** α dose not converge **do**
12: $[\sim, \hat{\mathbf{S}}, \sim] \leftarrow \mathrm{svd}(\mathbf{W} - \alpha\mathbf{Q},\ 'econ')$
13: **if** $\alpha > \hat{\mathbf{S}}(l,l)$ **then break**
14: $\alpha \leftarrow (\hat{\mathbf{S}}(l,l) + \alpha)/2$
15: **end while**

These steps are appended in the front of Step 11 in Alg. 4. Among them, performing SVD consumes much time and can be optimized with the following trick. Suppose $\mathbf{C} = \mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q} - \alpha\mathbf{Q}$. Then, the singular values of $\mathbf{C}$ are the square root of the eigenvalues of $\mathbf{C}^{\mathrm{T}}\mathbf{C}$. We can derive

$$\mathbf{C}^{\mathrm{T}}\mathbf{C} = \mathbf{Q}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q} - 2\alpha\mathbf{Q}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{Q} + \alpha^2\mathbf{I} = \mathbf{W}^{\mathrm{T}}\mathbf{W} - 2\alpha\mathbf{Y}^{\mathrm{T}}\mathbf{Y} + \alpha^2\mathbf{I}. \tag{17}$$

Therefore, we can firstly compute two matrices $\mathbf{D}_1 = \mathbf{W}^{\mathrm{T}}\mathbf{W}$ and $\mathbf{D}_2 = \mathbf{Y}^{\mathrm{T}}\mathbf{Y}$ to replace SVD with $[\sim, \hat{\mathbf{S}}^2] \leftarrow \mathrm{eig}(\mathbf{D}_1 - 2\alpha\mathbf{D}_2 + \alpha^2\mathbf{I})$, which just applies eigenvalue decomposition on $l \times l$ matrix to update α.

Combining the techniques proposed in last subsections with the dynamic scheme to update α in each power iteration, we derive the PerSVD algorithm which is described as Algorithm 5. In Step 14 and 18, it checks if $\frac{\sigma_l((\mathbf{A}^{\mathrm{T}}\mathbf{A} - \alpha\mathbf{I})\mathbf{Q}) + \alpha}{2}$ is larger than α. For same setting of power parameter p, Alg. 5 has comparable computational cost as Alg. 3, but achieves results with better accuracy due to the usage of shifted power iteration. For matrix $\mathbf{Q}$ computed with Alg. 5, we have derived a bound of $||\mathbf{Q}\mathbf{Q}^{\mathrm{T}}\mathbf{A} - \mathbf{A}||$ which reflects how close the computed truncated SVD is to optimal. And, we prove that the bound is smaller than that derived in [17] for $\mathbf{Q}$ computed with the basic randomized SVD algorithm. The complete proof is given in Appendix A.2. To accelerate Alg. 5, we can use eigenvalue decomposition to compute the economic SVD or the orthonormal basis of a "tall-and-skinny" matrix in Step 2, 20 and 22. This is accomplished by using the eigSVD algorithm from [6], described in Appendix A.3.

3.4 Analysis of Computational Cost

Firstly, we analyze the peak memory usage of Alg. 5. Because the SVD in Step 17 costs extra $2nl \times 8$ bytes memory except for the space of $\mathbf{W}$ and $\mathbf{Q}$, the memory usage in Step 17 is $(m + 4n)l \times 8$ bytes, which reflects the space of one $m \times l$ and

Algorithm 5 Pass-efficient randomized SVD with shifted power iteration (PerSVD)

Input: $\mathbf{A} \in \mathbb{R}^{m\times n}$, k, l $(l \geq k)$, p
Output: $\mathbf{U} \in \mathbb{R}^{m\times k}$, $\mathbf{S} \in \mathbb{R}^{k\times k}$, $\mathbf{V} \in \mathbb{R}^{n\times k}$

1: $\mathbf{\Omega} \leftarrow \text{randn}(n, l)$
2: $\mathbf{Q} \leftarrow \text{orth}(\mathbf{\Omega})$
3: $\alpha \leftarrow 0$
4: **for** $j = 1, 2, \cdots, p+1$ **do**
5: $\quad \mathbf{W} \leftarrow \text{zeros}(n, l)$
6: $\quad$ **for** $i = 1, 2, \cdots, m$ **do**
7: $\quad\quad \mathbf{a}_i$ is the i-th row of matrix $\mathbf{A}$
8: $\quad\quad \mathbf{Y}(i,:) \leftarrow \mathbf{a}_i\mathbf{Q}$, $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{a}_i^{\mathrm{T}}\mathbf{Y}(i,:)$
9: $\quad$ **end for**
10: $\quad$ **if** $j == p+1$ **break**
11: $\quad \mathbf{D}_1 \leftarrow \mathbf{W}^{\mathrm{T}}\mathbf{W}$, $\mathbf{D}_2 \leftarrow \mathbf{Y}^{\mathrm{T}}\mathbf{Y}$
12: $\quad$ **while** α dose not converge **do**
13: $\quad\quad [\sim, \hat{\mathbf{S}}^2] \leftarrow \text{eig}(\mathbf{D}_1 - 2\alpha\mathbf{D}_2 + \alpha^2\mathbf{I})$
14: $\quad\quad$ **if** $\alpha > \hat{\mathbf{S}}(l,l)$ **then break**
15: $\quad\quad \alpha \leftarrow (\hat{\mathbf{S}}(l,l) + \alpha)/2$
16: $\quad$ **end while**
17: $\quad [\mathbf{Q}, \hat{\mathbf{S}}, \sim] \leftarrow \text{svd}(\mathbf{W} - \alpha\mathbf{Q}, 'econ')$
18: $\quad$ **if** $\alpha < \hat{\mathbf{S}}(l,l)$ **then** $\alpha \leftarrow (\hat{\mathbf{S}}(l,l) + \alpha)/2$
19: **end for**
20: $[\mathbf{Q}, \tilde{\mathbf{S}}, \tilde{\mathbf{V}}] \leftarrow \text{svd}(\mathbf{Y}, 'econ')$
21: $\mathbf{B} \leftarrow \tilde{\mathbf{S}}^{-1}\tilde{\mathbf{V}}^{\mathrm{T}}\mathbf{W}^{\mathrm{T}}$
22: $[\mathbf{U}, \mathbf{S}, \mathbf{V}] \leftarrow \text{svd}(\mathbf{B}, 'econ')$
23: $\mathbf{U} \leftarrow \mathbf{Q}\mathbf{U}(:, 1:k), \mathbf{S} \leftarrow \mathbf{S}(1:k, 1:k), \mathbf{V} \leftarrow \mathbf{V}(:, 1:k)$

two $n \times l$ matrices and the space caused by SVD operation. Because the SVD in Step 20 is replaced by eigSVD, the memory usage at Step 20 is $(2m+n)l\times 8$ bytes. Therefore, the peak memory usage of Alg. 5 is $\max((m+4n)l, (2m+n)l)\times 8$ bytes. Usually we set $l = 1.5k$, so the memory usage of Alg. 2 ($16(m+n)k \times 8$ bytes) is several times larger than that of Alg. 5 ($\max(1.5(m+4n)k, 1.5(2m+n)k) \times 8$ bytes).

Secondly, we analyze the *flop* count of Alg. 5. Because eigSVD is used in Step 2, 20 and 22, and the *flop* count of eigSVD on a $m \times l$ matrix is $2C_{mul}ml^2 + C_{eig}l^3$, the *flop* count of those computations is $C_{mul}(4n+2m)l^2 + 3C_{eig}l^3$. Because the computations in Step 12-16 are all about $l \times l$ matrices and $l \ll \min(m, n)$,

we drop the *flop* count in Step 12-16. Therefore, the *flop* count of Alg. 5 is

$$\begin{aligned}\mathrm{FC}_5 =& (2p+2)C_{mul}mnl + pC_{mul}(m+n)l^2 + pC_{svd}nl^2 + C_{mul}nl^2 + C_{mul}mlk \\ &+ C_{mul}(4n+2m)l^2 + 3C_{eig}l^3 \\ \approx& (2p+2)C_{mul}mnl + pC_{mul}(m+n)l^2 + pC_{svd}nl^2 + C_{mul}(mlk + 2ml^2 + 5nl^2)\end{aligned} \tag{18}$$

where $(2p+2)C_{mul}mn$ reflects $2p+2$ times matrix-matrix multiplication in Step 6-9, $pC_{mul}(m+n)l^2$ reflects the matrix-matrix multiplications in Step 11, $pC_{svd}nl^2$ reflects the SVD in Step 17, and $C_{mul}nl^2 + C_{mul}mlk$ reflects the matrix-matrix multiplications in Step 21 and 23.

Because $\min(m,n) \gg l$ and FC_1 and FC_5 both contain $(2p+2)C_{mul}mnl$ which reflects the main computation, the *flop* count of Alg. 5 is similar with *flop* count of Alg. 1 in (3) with the same p. Because the *flop* count of main computation in Alg. 5 is $(2p+2)C_{mul}mnl$ and that of Alg. 2 is $16C_{mul}mnk$, according to the fact $l = 1.5k$, $(2p+2)C_{mul}mnl = (3p+3)C_{mul}mnk$ is less than $16C_{mul}mnk$ when $p \leq 4$. If $p = 2$ or 3, we see that the total runtime of the two algorithms may be comparable, considering Alg. 5 reads data multiple times from the hard disk and Alg. 2 just reads data once.

4 Experimental Results

In this section, numerical experiments are carried out to validate the proposed techniques. We first compare Alg. 1, Alg. 3, Alg. 4 and Alg. 5 (PerSVD), to validate whether the proposed schemes in Section 3 can remarkably reduce the passes with the same accuracy of results. Then, we compare PerSVD and Tropp's algorithm to show the advantage of PerSVD on accurately computing SVD of large matrix stored on hard disk[1].

We consider several matrices stored in the row-major format on hard disk as the test data, which are listed in Table 1. Firstly, two 40,000 × 40,000 matrices (denoted by Dense1 and Dense2) are synthetically generated. Dense1 is randomly generated with the i-th singular value following $\sigma_i = 1/i$. Then, Dense2 is randomly generated with the i-th singular value following $\sigma_i = 1/\sqrt{i}$, which reflects the singular values of Dense2 decay slower than those of Dense1. Then we construct two matrices from real-world data. We use the training set of MNIST [12] which has 60k images of size 28×28, and we reshape each image into a vector in size 784 to obtain the first matrix in size $60,000 \times 784$ for experiment. The second matrix is obtained from images of faces from FERET database [16]. As in [8], we add two duplicates for each image into the data. For each duplicate, the value of a random choice of 10% of the pixels is set to random numbers uniformly chosen from 0,1,$\cdots$,255. This forms a 102,042 × 393,216 matrix, whose rows consist of images. We normalize the matrix by subtracting from each row its mean, and then dividing it by its Euclidean norm.

[1] The code is avaliable at https://github.com/XuFengthucs/PerSVD

Table 1: Test matrices.

Matrix	# of rows	# of columns	Space usage on hard disk
Dense1	40,000	40,000	6.0 GB
Dense2	40,000	40,000	6.0 GB
MNIST	60,000	782	180 MB
FERET	102,042	393,216	150 GB

All experiments are carried out on a Ubuntu server with two 8-core Intel Xeon CPU (at 2.10 GHz) and 32 GB RAM. The proposed techniques, basic randomize SVD algorithm and Tropp's algorithm are all implemented in C with MKL [1] and OpenMP directives for multi-thread parallel computing. `svds` in Matlab 2020b is used for computing the accurate results and for error metrics. We set $l = 1.5k$, and each time we read k rows of matrix to avoid extra memory cost for all algorithms. All the programs are evaluated with wall-clock runtime and peak memory usage. To simulate the machine with limited computational resources, we just use 8 threads on one CPU to test the algorithms. Because the matrix FERET is too large to be loaded in Matlab, the experiments on FERET are without accurate results to compute the error metrics.

4.1 Error Metrics

Theoretical research has revealed that the randomized SVD with power iteration produces the rank-k approximation close to optimal. Under spectral (or Frebenius) norm the computational result ($\hat{\mathbf{U}}$, $\hat{\mathbf{\Sigma}}$ and $\hat{\mathbf{V}}$) has the following multiplicative guarantee:

$$\|\mathbf{A} - \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\hat{\mathbf{V}}^{\mathrm{T}}\| \leq (1+\epsilon)\|\mathbf{A} - \mathbf{A}_k\|, \tag{19}$$

with high probability. Based on (19), we use

$$\epsilon_{\mathrm{F}} = (\|\mathbf{A} - \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\hat{\mathbf{V}}^{\mathrm{T}}\|_F - \|\mathbf{A} - \mathbf{A}_k\|_F)/\|\mathbf{A} - \mathbf{A}_k\|_F, \quad \text{and} \tag{20}$$

$$\epsilon_{\mathrm{s}} = (\|\mathbf{A} - \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\hat{\mathbf{V}}^{\mathrm{T}}\|_2 - \|\mathbf{A} - \mathbf{A}_k\|_2)/\|\mathbf{A} - \mathbf{A}_k\|_2, \tag{21}$$

as first two error metrics to evaluate the accuracy of randomized SVD algorithms for Frobenius norm and spectral norm.

Another guarantee proposed in [15], which is stronger and more meaningful in practice, is:

$$\forall i \leq k, \quad |\mathbf{u}_i^{\mathrm{T}}\mathbf{A}\mathbf{A}^{\mathrm{T}}\mathbf{u}_i - \hat{\mathbf{u}}_i^{\mathrm{T}}\mathbf{A}\mathbf{A}^{\mathrm{T}}\hat{\mathbf{u}}_i| \leq \epsilon\sigma_{k+1}(\mathbf{A})^2, \tag{22}$$

where $\mathbf{u}_i$ is the i-th left singular vector of $\mathbf{A}$, and $\hat{\mathbf{u}}_i$ is the computed i-th left singular vector. This is called *per vector error* bound for singular vectors. In [15], it is demonstrated that the $(1+\epsilon)$ error bound in (19) may not guarantee any accuracy in the computed singular vectors. In contrary, the per vector guarantee

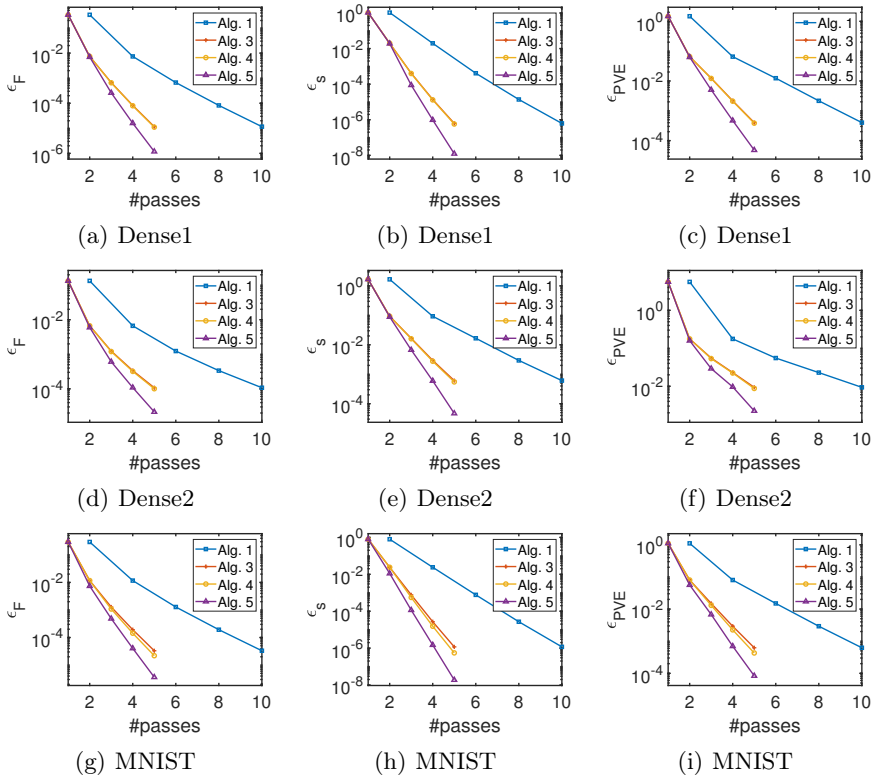
(22) requires each computed singular vector to capture nearly as much variance as the corresponding accurate singular vector. Based on the per vector error bound (22), we use

$$\epsilon_{\text{PVE}} = \max_{i \le k} \frac{|\mathbf{u}_i^{\mathrm{T}} \mathbf{A}\mathbf{A}^{\mathrm{T}} \mathbf{u}_i - \hat{\mathbf{u}}_i^{\mathrm{T}} \mathbf{A}\mathbf{A}^{\mathrm{T}} \hat{\mathbf{u}}_i|}{\sigma_{k+1}(\mathbf{A})^2} \tag{23}$$

to evaluate the accuracy of randomized SVD algorithms. Notice that the metric (20), (21) and (23) were also used in [2] with name "Fnorm", "spectral" and "rayleigh(last)".

4.2 Comparison with Basic Randomized SVD Algorithm

In order to validate proposed techniques, we set different power parameter p and perform the basic randomized SVD (Alg. 1), Alg. 3 with pass-efficient scheme, Alg. 4 with fixed shift value and the proposed PerSVD with shifted power iteration (Alg. 5) on test matrices. The orthonormalization is used in the power iteration of Alg. 1. With the accurate results obtained from `svds`, the corresponding



(a) Dense1 (b) Dense1 (c) Dense1

(d) Dense2 (e) Dense2 (f) Dense2

(g) MNIST (h) MNIST (i) MNIST

Fig. 2: Error curves of the randomized SVD algorithms with different number of passes ($k = 100$).

error metrics (20), (21) and (23) are calculated to evaluate the accuracy. For the largest matrix FERET in Table 1, the accurate SVD cannot be computed with `svds` due to out-of-memory error. So, the results for it are not available. The curves of error metrics vs. the number of passes over the matrix $\mathbf{A}$ are drawn in Fig. 2. We set $k = 100$ in this experiment.

From Fig. 2 we see that, the dynamic scheme for setting the shift consistently (i.e Alg. 5) produces results with remarkably better accuracy than the scheme with and without the fixed shift. On Dense1 with 4 passes, the reduction of ϵ_s of Alg. 5 is 14X and 13X compared with Alg. 3 and Alg. 4, respectively. And, the results of Alg. 4 are more accurate than the results of Alg. 3 with same number of passes. Notice the error metrics of Alg. 4 on Dense1 and Dense2 are less than those of Alg. 3 although the curves are indistinguishable in Fig. 2. For example, on Dense1 with 5 passes, the ϵ_s of Alg. 4 is 5.8×10^{-7} less than 6.2×10^{-7} of Alg. 3, which reflects fixed shift value can improve limited accuracy of results. With the same number of passes over $\mathbf{A}$, PerSVD (Alg. 5) exhibits much larger accuracy than the basic randomized SVD algorithm (Alg. 1) and the advantage increases with the number of passes. Besides, the reduction of result's error of PerSVD increases with the number of passes. With same 4 passes over the matrix Dense1, the result computed with PerSVD is up to **20,318X** more accurate than that obtained with the basic randomized SVD. According to the analysis in Section 3.4, when the number of passes is the same, the runtime excluding the reading time of Alg. 5 is about twice that of Alg. 1. Since the time for reading data is dominant, the total runtime of Alg. 5 is slightly more than that of Alg. 1 with same number of passes.

4.3 Comparison with Single-Pass SVD Algorithm

Now, we compare PerSVD with Tropp's algorithm in terms of computational cost and accuracy of results. The results are listed in Table 2. In this experiment, we set $p = 2$ for PerSVD, which implies 3 passes over $\mathbf{A}$. Table 2 shows that PerSVD not only costs less memory but also produces more accurate results than Tropp's algorithm. The reduction of result's ϵ_s is up to **7,497X** on Dense1 with $k = 50$. The peak memory usage of PerSVD is 16%-26% of that of Tropp's algorithm, which matches the analysis in Section 3.4. Although PerSVD costs about 3X time than Tropp's algorithm on reading data from hard disk, the total runtime of PerSVD is less than Tropp's algorithm, which also matches our analysis. On the largest data FERET, PerSVD just costs 1.9 GB and about 12 minutes to compute truncated SVD with $k = 100$ of all 150 GB data stored on hard disk, which reflects the efficiency of PerSVD.

5 Conclusion

We have developed a pass-efficient randomized SVD algorithm named PerSVD to accurately compute the truncated top-k SVD. PerSVD builds on a technique reducing the passes over the matrix and an innovative shifted power iteration

Table 2: The runtime, memory cost and result errors of the Tropp's algorithm and our PerSVD algorithm ($p = 2$). The unit of runtime is second.

Matrix	k	Tropp's algorithm						PerSVD ($p = 2$)					
		t_r	t	Memory	ϵ_{F}	ϵ_{s}	ϵ_{PVE}	t_r	t	Memory	ϵ_{F}	ϵ_{s}	ϵ_{PVE}
Dense1	50	4.7	27	592 MB	0.38	0.46	0.87	15	26	144 MB	4E-4	6E-5	0.009
	100	4.4	41	1133 MB	0.39	0.51	0.66	14	30	260 MB	4E-4	0.001	0.01
Dense2	50	4.4	27	591 MB	0.49	3.57	9.2	16	27	144 MB	7E-4	0.006	0.04
	100	4.4	50	1133 MB	0.51	3.36	9.2	14	31	260 MB	8E-4	0.02	0.04
MNIST	50	0.29	3.9	498 MB	0.32	0.42	0.67	0.90	1.4	81 MB	4E-4	0.001	0.008
	100	0.31	8.9	966 MB	0.14	0.10	0.24	0.79	1.5	156 MB	4E-4	3E-4	0.006
FERET	50	140	648	3.71 GB	-	-	-	296	597	0.96 GB	-	-	-
	100	178	1366	7.25 GB	-	-	-	293	703	1.90 GB	-	-	-

t_r means the time for reading the data, while t means the total runtime (including t_r).
"-" means the error metrics are not available for FERET matrix.

technique. It aims to handle the real-world data with slowly-decayed singular values and accurately compute the top-k singular triplets with a couple of passes over the data. Experiments on various matrices have verified the effectiveness of PerSVD in terms of runtime, accuracy and memory cost, compared with existing randomized SVD and single-pass SVD algorithms. PerSVD is expected to become a powerful tool for computing SVD of really large data.

References

1. Intel oneAPI Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html (2021)
2. Allen-Zhu, Z., Li, Y.: LazySVD: Even faster SVD decomposition yet without agonizing pain. In: Advances in Neural Information Processing Systems. pp. 974–982 (2016)
3. Baglama, J., Reichel, L.: Augmented implicitly restarted lanczos bidiagonalization methods. SIAM Journal on Scientific Computing **27**(1), 19–42 (2005)
4. Boutsidis, C., Woodruff, D.P., Zhong, P.: Optimal principal component analysis in distributed and streaming models. In: Proc. the 48th annual ACM Symposium on Theory of Computing. pp. 236–249 (2016)
5. Eckart, C., Young, G.: The approximation of one matrix by another of lower rank. Psychometrika **1**(3), 211–218 (1936)
6. Feng, X., Xie, Y., Song, M., Yu, W., Tang, J.: Fast randomized PCA for sparse data. In: Proc. the 10th Asian Conference on Machine Learning (ACML). pp. 710–725 (14–16 Nov 2018)
7. Golub, G.H., Van Loan, C.F.: Matrix Computations. JHU Press (2012)
8. Halko, N., Martinsson, P.G., Shkolnisky, Y., Tygert, M.: An algorithm for the principal component analysis of large data sets. SIAM Journal on Scientific Computing **33**(5), 2580–2594 (2011)

9. Halko, N., Martinsson, P.G., Tropp, J.A.: Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. SIAM Review **53**(2), 217–288 (2011)
10. Horn, R.A., Johnson, C.R.: Topics in Matrix Analysis. Cambridge University Press (1991). https://doi.org/10.1017/CBO9780511840371
11. Larsen, R.M.: Propack-software for large and sparse SVD calculations. Available online. http://sun.stanford.edu/~rmunk/PROPACK (2004)
12. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
13. Li, H., Linderman, G.C., Szlam, A., Stanton, K.P., Kluger, Y., Tygert, M.: Algorithm 971: An implementation of a randomized algorithm for principal component analysis. ACM Transactions on Mathematical Software **43**(3), 1–14 (2017)
14. Martinsson, P.G., Tropp, J.A.: Randomized numerical linear algebra: Foundations and algorithms. Acta Numerica **29**, 403–572 (2020)
15. Musco, C., Musco, C.: Randomized block Krylov methods for stronger and faster approximate singular value decomposition. In: Advances in Neural Information Processing Systems. pp. 1396–1404 (2015)
16. Phillips, P.J., Moon, H., Rizvi, S.A., Rauss, P.J.: The feret evaluation methodology for face-recognition algorithms. IEEE Transactions on pattern analysis and machine intelligence **22**(10), 1090–1104 (2000)
17. Rokhlin, V., Szlam, A., Tygert, M.: A randomized algorithm for principal component analysis. SIAM Journal on Matrix Analysis and Applications **31**(3), 1100–1124 (2010)
18. Shishkin, S.L., Shalaginov, A., Bopardikar, S.D.: Fast approximate truncated SVD. Numerical Linear Algebra with Applications **26**(4), e2246 (2019)
19. Tropp, J.A., Yurtsever, A., Udell, M., Cevher, V.: Streaming low-rank matrix approximation with an application to scientific simulation. SIAM Journal on Scientific Computing **41**(4), A2430–A2463 (2019)
20. Voronin, S., Martinsson, P.G.: RSVDPACK: An implementation of randomized algorithms for computing the singular value, interpolative, and CUR decompositions of matrices on multi-core and GPU architectures. arXiv preprint arXiv:1502.05366 (2015)
21. Yu, W., Gu, Y., Li, J., Liu, S., Li, Y.: Single-pass PCA of large high-dimensional data. In: Proc. International Joint Conference on Artificial Intelligence (IJCAI). pp. 3350–3356 (Aug 2017)
22. Yu, W., Gu, Y., Li, Y.: Efficient randomized algorithms for the fixed-precision low-rank matrix approximation. SIAM Journal on Matrix Analysis and Applications **39**(3), 1339–1359 (2018)