Report

S. Abramov, D. Koposov, D. Lopatkin,
A. Nagapetyan, V. Prutyanov

December 2018

Abstract

In this work we considered the problem of constructing a near-optimal rank k approximation of a matrix $M \in [0,1]^{m\times n}$ under the streaming data model where the columns of M are revealed sequentially. To solve this problem we implemented Streaming Low-rank Approximation (SLA) method. SLA has optimal memory consumption and outperforms such methods as scipy.svds, Robust PCA [2] and Sketchy CGM [3]. One of the main features of SLA is an ability to work with a streaming data.

1 Introduction

Low-rank approximation is an important problem in a wide range of applications: image compression, noise reduction, latent semantic indexing, etc. We have considered a problem of low-rank approximation of large matrices $M \in [0,1]^{n\times m}$ in a memory and computationaly efficient manner.
For this purpose we implemented Streaming Low-rank Approximation (SLA) algorithm considered in [1]. The main features of this algorithm are online processing of the columns of observation matrix M and random sparsification of entries of M. These features allow SLA to process large matrices M with low consumption of memory.
Every entry of matrix M is revealed with probability δ (also called sampling rate). That allows not only to reduce a memory footprint, but also to improve performance of the algorithm, since our implementation of SLA is suitable for using sparse matrices in an efficient way. In the first step SLA stores l randomly chosen columns of M and uses spectral decomposition to estimate parts of k top right singular vectors of M. After that the algorithm just process remaining columns one by one in a sequential manner. That allows to complete the estimation of singular vectors obtained in the initial step. Finally, SLA constructs approximation $M^{(k)}$ via linear projection of estimated top k singular vectors onto the subspace generated by these vectors.
The analysis of the performance of SLA is presented in [1] (Theorem 7, Theorem 8). In summary, with probability $1 - k\delta$ the output $\hat{M}^{(k)}$ of SLA satisfies:

$$\frac{\|M - \hat{M}^{(k)}\|_F^2}{mn} = O\Big(k^2\Big(\frac{s_{k+1}^2(M)}{mn} + \frac{\log(m)}{\sqrt{\delta m}}\Big)\Big), \tag{1}$$

where s_{k+1} is the $(k+1)$-th singular value of M.
Memory consumption of SLA is $O(k(m+n))$ and this value is optimal since every approximation algorithm needs to store at least its output. Running time of SLA is $O(\delta kmn + \frac{k^2}{\delta} + k^2 m)$, but in most cases the biggest term is the first one – $O(\delta kmn)$.

2 The main part

2.1 Algorithm

We used exactly the same algorithm, which is decribed in the main source[1] (also see Algorithm 1, 2).

It operates simultaneously at most with one dense column of the input matrix M. Moreover, it uses random sparsification, which allows to utilize sparse structures to save memory and speed up algorithm for large matrices. Of course, such techniques affect quality of approximation, however we can easily adjust input parameters to obtain the optimum rate of speed and quality.

Let's consider the case: $m \geq n \gg l > k$. Maximum memory usage can be achieved at the beginning, when two matrices of size $m \times l$ need to be stored in memory, but for small enough δ they become sparse with $O(\delta ml)$ elements. If $\delta l \leq k$, then the upper bound is mk. Matrix W also is of size $m \times k$. Matrix Q is much smaller. It means that in the first part of the algorithm we need memory for $O(mk)$ elements. In the second part (cycle and below) the biggest are matrices $\hat{U}, \hat{V}, \hat{I}$. They have sizes mk, nk, mk, respectively. The maximum memory needed is $O(k(n+m))$.

Complexity of the algorithm also depend linearly on the sampling rate (parameter δ) – $O(\delta kmn)$, if sparse structures are utilized (Fig. 3).

Unfortunately, the structure of the algorithm is not very convenient for an efficient utilization of the sparse matrices. For example, every vector A_t is sparse, it is multiplied by sparse matrix W and the result will be also a sparse vector of matrix $\hat{V}$. After that we produce sparse matrix as a product of two sparse vectors. The first problem here is that we need to transform each vector A_t into one of sparse formats, which requires $O(m)$ operations $\gg O(\delta m)$. Nevertheless, we can still benifit from this transformation, because $O(mn + \delta kmn) \leq O(kmn)$, where first term in the left part is responsible for transorfmations to sparse

Algorithm 1: Streaming Low-rank Approximation (SLA)

Input : M, k, δ and l
$A_{(B_1)}$, $A_{(B_1)} \leftarrow$ get a column of M and randomly sample its entries at rate δ
Apply PCA for the first l columns: $Q \leftarrow SPCA(A_{(B_1)}, k)$
Nullify rows of $A_{(B_2)}$ with more thant 2 non-zero elements
Nullify columns of $A_{(B_2)}$ with more thant $10\delta m$ non-zero elements
$W \leftarrow A_{(B_2)}Q$
$\hat{V}^{(B_1)} \leftarrow (A_{(B_1)})^T W$
$\hat{I} \leftarrow A_{(B_1)}\hat{V}^{(B_1)}$
Delete $A_{(B_1)}$, $A_{(B_2)}$, Q from memory
for $t = l + 1$ *to* n **do**
  A_t get a new column of M and randomly sample its entries at rate δ
  $\hat{V}^t \leftarrow (A_t)^T W$
  $\hat{I} \leftarrow \hat{I} + A_t\hat{V}$
end
Find such R that $\hat{V}R$ is an orthonormal matrix
$\hat{U} \leftarrow \frac{1}{\delta}\hat{I}RR^T$
Output: $M_k = |\hat{U}\hat{V}^T|_0^1$ - an approximation of the matrix M of rank k

Algorithm 2: Spectral PCA (SPCA)

Input : $A \in [0,1]^{m \times l}$, k
$\Omega \leftarrow l \times k$ Gaussian random matrix
$\hat{A} \leftarrow A$ and set the entries of $\hat{A}$ with more than 10 non-zero elements to 0
$F \leftarrow \hat{A}^T\hat{A} - \text{diag}(\hat{A}^T\hat{A})$
$F \leftarrow F^{\lceil 5 \log(l) \rceil}\Omega$
$Q, R \leftarrow$ find QR-decomposition of F
Output: Q

format, and the second term is the complexity of the following sparse operations. The second problem is about matrix $\hat{V}$. Its sparsity always changes, therefore it becomes impossible to use formats efficient in terms of matrix operations, like CSR/CSC. When we get new column, we have to reconstruct all the matrix from scratch due to the features of the compressed format.

Input parameters of the algorithm

Authors of the article proposed a range of suitable parameters δ:

$$\frac{\log(m)^4}{m} < \delta < m^{-8/9}$$

For some specified $\delta \rightarrow l = \frac{1}{\delta \log m}$. If the parameters are chosen like this, then the relative accuracy of approximation is small (see Theorem 7[1]). The problem is that the aformentioned inequality holds only for small values of m (number of rows in the matrix M). It makes the proposed algorithm inapplicable in real life conditions. However, our experiments showed that the algorithm is able to successfully solve different problems with different values of δ and l. We choose input parameters empirically according to the following rules:

1. Choose any desired rank k
2. Choose $l > k$ (need to construct matrix Q)
3. l need to be as small as possible (less memory consumption)
4. Try to choose $\delta \in [0,1]$ such that $\delta l < k$ (less memory, larger sparsity)
5. If the result is not good enough – increase δ (more information preserved, higher quality)

2.2 Evaluation

2.2.1 Memory analysis

Original article proves that memory complexity of SLA algorithm is $O(k(n+m))$. We have investigated memory consumption of our implementation. All significant memory allocations for matrices were tracked and peak RAM consumption was computed. There are peak memory consumption graphs for backgound extraction (discussed below in details).

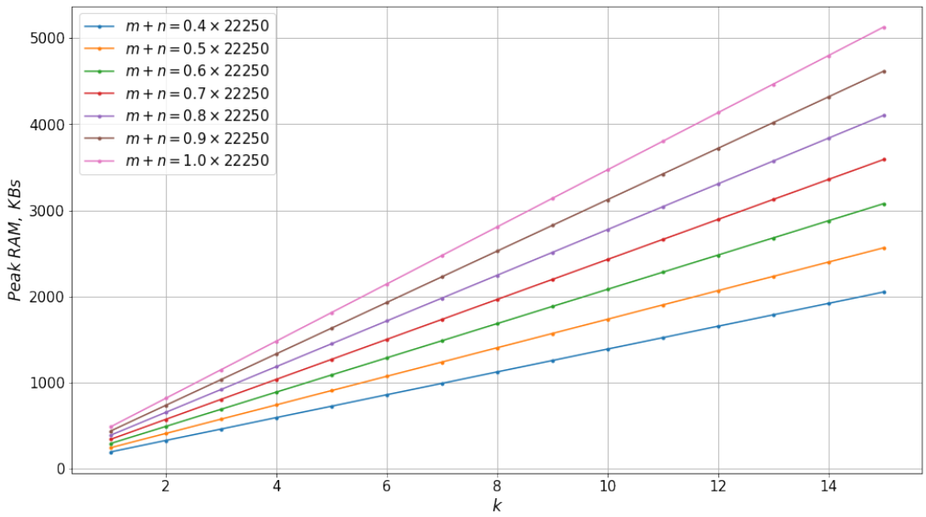

Figure 1: How peak RAM depends on m, n, k

One can see, that peak RAM consumption by matrices memory allocations is linearly dependent on $(m + n)$ and k. Thus, theoretical estimate from original article has been confirmed.

2.2.2 Complexity analysis

In Fig. 2 and Fig. 3 we present complexity analysis of our algorithm.

Execution time depends on δ and matrix size. For small matrices **svds** from scipy.sparse.linalg works faster, but after some threshold SLA works better. This threshold depends on matrix size, k, δ.
Also implementation of SLA for sparse matrices works faster for large δ. For dense matrix implementation there is no correlation between execution time and δ.

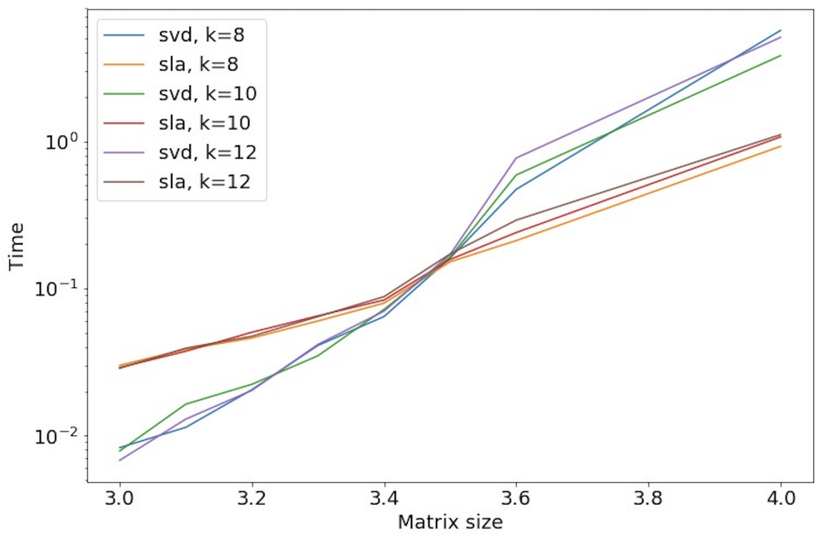

Figure 2: Dependency of SLA/SVD execution time on matrix size

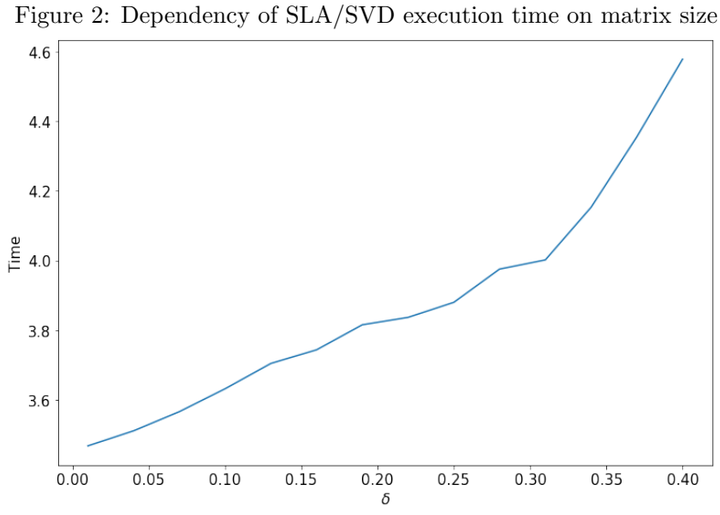

Figure 3: Dependency of SLA execution time on δ

2.3 Applications

2.3.1 Foreground/background separation in videos

Low-rank approximation methods as a whole, and SLA in particular can be used for background extraction from video. The idea is following: we have a sequence of video frames:

$$F_i,\ i = 1, ..., n$$

We stack vectorized frames and put it into the observation matrix

$$Y = [\text{vec}(F_1), ..., \text{vec}(F_n)]$$

The background component is of low-rank, since the background is static within a short period (ideally it is rank one as the image would be the same). The problem fits into the following signal model

$$Y = X + E$$

where Y is the observation, X is a low-rank matrix (background) and E is a sparse matrix (foreground).

We have tested our SLA implementation with video of 3050 frames of 160x120 pixels. It has taken about 3 sec on a laptop to process this video. In comparison, Sketchy CGM implementation spends approximately 30 min to do this work. The video can be downloaded from here.

Figure 4: Original and it's corresponding background and foreground frames

2.3.2 Shadow removing

Another application of the SLA method is shadow removing. In this case, we are given sequence of photos of an object under different illumination conditions:

$$P_i,\ i = 1, ..., n$$

To obtain the observation matrix Y, we need to stack vectorized photos altogether.

$$Y = [\text{vec}(P_1), ..., \text{vec}(P_n)]$$

The object image is of low-rank, since it is static and remains the same for all pictures. Therefore, this problem can be represented in the next way:

$$Y = X + E$$

where Y is the observation, X is a low-rank matrix (object image) and E is a sparse matrix (shadows).

We have tested our SLA implementation on The Extended Yale Face Database B. This dataset contains images of human subjects under different illumination conditions. It has taken about 100 ms to process the sequence of 58 photos of one person (photos without shadows were excluded) (Fig. 5).

3 Related works

There are some works that consider sparsification approach for effective computations and memory usage. The best additive error bound were obtained in [4]. When $k = O(1)$ additive error bound of SLA is larger only by a logarithmic term of m than the bound of [4]. But memory consumption of [4] is $O(\delta mn)$, where δ is a sampling rate. So, SLA gives more efficient memory usage in comparison with [4].
Also there are algorithms that use streaming PCA approach to process observaton matrix in a sequential manner. One of these methods considered in [5]. Such methods demand two passes of an algorithm on observaton matrix (first pass – to estimate left singular vectors, second pass – to obtain right singular vectors) and this is the main disadvantage of these streaming methods. Article [6] estimates running time of the algorithm from [5]. It's estimation is $O(kmn\epsilon^{-1})$, where ϵ is the relative error of approximation. This running time is much greater than for SLA.

4 Code

Code is available at our GitHub repository.
Robust PCA implementation is available here.
Sketchy CGM implementation is available here.

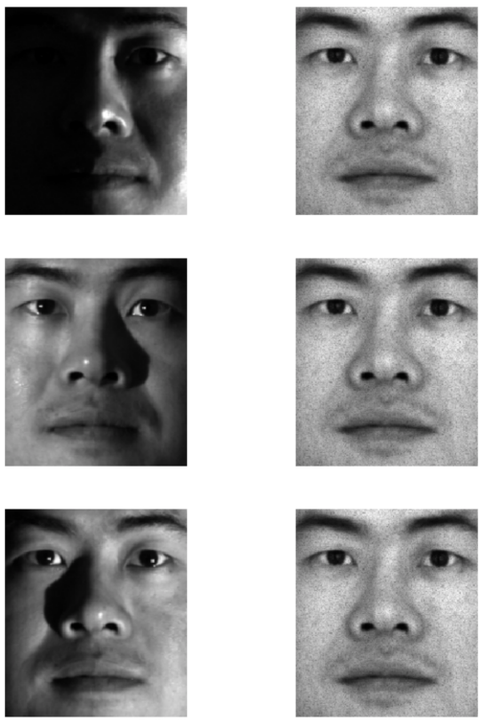

Figure 5: Removing shadow from the face

5 Contribution

- Semen Abramov
  - SLA implementation
  - Robust PCA implementation
  - Applications: faces shadow removing
- Denis Koposov
  - SLA implementation
  - Robust PCA implementation
  - Applications: faces shadow removing
- Daniil Lopatkin
  - SLA implementation
  - Sketchy CGM implementation
  - Applications: video foreground/background separation
- Albert Nagapetyan
  - Performance analysis
- Viktor Prutyanov
  - Memory consumption analysis
  - Sketchy CGM implementation
  - Applications: video foreground/background separation

References

[1] Yun, S. Y. and Proutiere, A. (2015). Fast and memory optimal low-rank matrix approximation. In Advances in Neural Information Processing Systems (pp. 3177-3185).

[2] Candès, E. J., Li, X., Ma, Y. and Wright, J. (2011). Robust principal component analysis. Journal of the ACM (JACM), 58(3), 11.

[3] Yurtsever, A., Udell, M., Tropp, J. A. and Cevher, V. (2017). Sketchy decisions: Convex low-rank matrix optimization with optimal storage. arXiv preprint arXiv:1702.06838.

[4] Dimitris Achlioptas, Frank Mcsherry. Fast computation of low-rank matrix approxi- mations. Journal of the ACM (JACM), 54(2):9, 2007.

[5] Edo Liberty. Simple and deterministic matrix sketching. In Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 581– 588. ACM, 2013.

[6] Mina Ghashami and Jeff M Phillips. Relative errors for deterministic low-rank matrix approximations. In SODA, pages 707–717. SIAM, 2014.