Implementation of the Halko et al's Randomized SVD framework

Yassine Abderrahmani

Laboratoire d'Informatique et Systèmes (LIS)

March 06, 2020

Abstract

Random methods for the development of computational algorithms in linear algebra for the approximation of smaller rank matrices have undergone many developments in the past years. Among them is the calculation of Singular Value Decomposition (SVD). Thus the study below essentially presents an implementation with some numerical results of the framework proposed by Halko et al in the document : Finding Structure with Randomness, Probabilistic Algorithms for Constructing Approximate Matrix Decompositions (published in 2011). We will see that these randomized algorithms for low rank approximation surpass in many cases the classical techniques on: time and memory used while being extremely precise.

Keywords : Randomized SVD, Image processing, Optimisation, Eigenfaces

1. Introduction

Matrix calculations play an important role in the field of applied mathematics. And there are nowadays many algorithms to carry out these tasks. However, it is very common to have to play with matrices containing hundreds of thousands of rows and columns, as is the case for example with recommendation systems for targeted advertising or image processing. For these particular cases, applying classical algorithms would imply a much too important consumption both in terms of memory and time. Although there have been many improvements in terms of pure computer hardware, the data collected through internet users has also become very large. It is therefore necessary to put in place algorithms to optimise our calculations as much as possible. And for this, the approximation of results is a very widespread technique when it is useless to obtain the exact result to reach our goals.

In their paper, Halko et al. present several algorithms according to the type of matrix, whether dense or sparse, whether it is taken into account by the RAM or by the hard disk memory. It is also taken into account the possibility of improving our precision if necessary while optimizing the spatial and temporal complexity of the algorithms and remaining stable. This research paper is based on principles of linear algebra combined with notions of probability in order to include a so-called randomization phenomenon in their calculations. Several application cases have been included to support their research. Here, the goal of this project is to implement their different algorithms to compute a random svd, and to apply them on one of their use cases to reproduce the numerical results while discussing them.

The present project has the following structure : first, the theoretical principles that constitute the svd will be discussed in section 2.1 and then the framework proposed by Halko et al and the reasons for them will be introduced in section 2.2. We will then proceed, in section 2.3, to its implementation and application on one of the use cases described in their paper. Namely the one on the FERET image database of human faces. These implementations will be done under the Python language for reasons that we will discuss in its corresponding section. And finally, we will discuss the results obtained.

1.1. Laboratory description

I led this project in the Computer Science and Systems Laboratory (LIS) located at Ecole Centrale de Marseille under the direction of the teacher-researcher M.Emiya Valentin. This laboratory is divided into several branches according to the fields of activities. The section in which I have been is called the QARMA team, it is a research team within the Data Science pole of the LIS.

This team develops a general expertise in machine learning and modern artificial intelligence, including theory, algorithms and applications. It is composed of about 15-20 members with complementary skills covering all fields of machine learning. This took place in about 3 weeks, and was mainly a question of making a study and putting into practice a research document dating from 2011.

It was proposed to me because of its great usefulness in the field of data science as well as for my interest in it. So I spent three weeks spread over three months in the laboratory. Where I first had to look into the document theory of the study document. Then I had, during 2 sessions per week, some time with my tutor to be guided and to have answers to my questions.

2. The Randomized SVD

This section will, in a global way, present the random SVD and then our implementation followed by some discussion of our results.

2.1. SVD overview

SVD (singular values decomposition) is a mathematical process which is a kind of generalization of the eigenvalue decomposition. It is indeed applicable to rectangular and not only square systems of equations.

Let be an arbitrary matrix $A \in \mathbb{R}^{n\times m}$. If it is of rank r, then it will be possible to write it in the form :

$$A = B \cdot C \tag{1}$$

With B in $\mathbb{R}^{n\times r}$ and C in $\mathbb{R}^{r\times m}$. Reducing at the same time the storage complexity from $O(m \times n)$ to $O(nr + rm)$ with $r \ll \max(n, m)$ which therefore represents a fairly significant memory gain.

Here we are interested in low rank matrix approximations since r is not small enough in most cases to allow sufficient compression. The objective is therefore to approach our reference matrix in the form $A = B \cdot C$ while taking into account the storage space and the precision thereof. To do this, many tools exist among which we find the SVD.

Quite briefly, the decomposition into singular values (SVD) of a matrix $A \in \mathbb{R}^{n\times m}$ is defined as :

$$A = U.\Sigma.V^T \tag{2}$$

Where :

- $U \in \mathbb{R}^{n\times n}$: Unit matrix of the left singular vectors.
- $\Sigma \in \mathbb{R}^{n\times m}$: Diagonal matrix containing all the singular values σ_i ordered decreasingly and all positive.
- $V^T \in \mathbb{R}^{m\times m}$: Unit matrix of the right singular vectors.

Computing a complete SVD of a matrix requires a complexity of $O\left(\min\left(mn^2, m^2n\right)\right)$ which is very resource intensive especially for the application case that we will establish in Section 2.3. Similarly, calculating a truncated SVD giving a sufficiently fine approximation also requires almost similar complexity.

Hence the need here to use another means, namely the use of **randomized algorithms.**

2.2. Randomized SVD and Framework

Let, then, talk about Halko et al's framework to compute a randomized SVD. They propose an algorithm divided into two stages, each of them having some different way to be computed regarding our matrix behaviour as we'll see it below. As said before, we want to compute an approximate matrix of A of rank k ¡¡ n, thus the following procedure have been proposed in the Halko et al's paper.

Algorithm 1: Prototype for Randomized SVD

Given an $m \times n$ matrix A, a target number k of singular vectors, and an exponent q (say, q = 1 or q = 2), this procedure computes an approximate rank-2k factorization $U.\Sigma.V^T$, where U and V are orthonormal, and Σ is nonnegative and diagonal.

Stage A :

1 Generate an $n \times 2k$ Gaussian test matrix Ω.
2 Form $Y = (AA^*)^q A\Omega$ by multiplying alternately with A and A^*.
3 Construct a matrix Q whose columns form an orthonormal basis for the range of Y.

Stage B :

4 Form $B = Q^*A$.
5 Compute an SVD of the small matrix: $B = \tilde{U}\Sigma V^*$.
6 Set $U = Q\tilde{U}$.

Briefly, what we get here is :

Stage 1 : Randomness

We want here to make a projection of our initial matrix into one which captures the action of A and thus would give us an approximation of the range of A, and $\boldsymbol{A \approx QQ^*A}$.

We compute a matrix **Q with 2 x k** orthonormal columns based on a Gaussian density function. Q has $2 \times k$ (could also be l = k + 5 as explained in their document, taking a few more sample helps to get better accuracy) to achieve near optimal approximation errors. And this doesn't change the scaling of the asymptotic computational cost of taking it instead of l = k + 5 as demonstrated by Halko's document.

Stage 2 : deterministic

Apply a classical SVD on the smaller matrix $B = Q^*A$ we get on from stage 1.

The reason why this algorithm being less greedy and well accurate is mainly because of the application, during stage B, of the svd on a smaller matrix, obtained during stage A, containing the action of the initial one. But this framework has also some drawbacks inside of the stage A which are easily countred by another way to compute Y. It is called the **Randomized Subspace Iteration** and it is an alternative to what they call the **Randomized Power Iteration** which correspond to Stage A.

In fact, computation of Y is vulnerable to round-off errors, to counter that we just need to compute an orthonormalization between each application of A and $A*$ as shown in the following algorithm. It's explained on algorithm 4.4 from their paper and is shown in our Section 2.2.1 below, we'll be using it for our implementation too.

The principal reason why these approaches are so efficient is that they are based on the fact that we create l random vectors $\left\{w^{(i)} : i = 1, \ldots, k\right\}$. And because of the randomness behind it, applying A on them $\left\{y^{(i)} = Aw^{(i)}\right\}$ allow us to get a linearly independent samples of vectors which **spans the range of A**. In fact, the results will nearly span most of its range with a high probability (and more if we take a little bit more samples, here we took $l = 2 \times k$ as suggested by their paper). Besides that, the probability of missing elements is almost negligible.

After that, computing $B = Q^*A$ is much easier and faster because the number of columns of Q $2 \times k$ is less than both dimensions of A. Calculating a direct SVD on B then would be less consuming.

2.2.1. Randomized Subspace Iteration

This section has a huge impact on some particular case as the one we will see in Section 3. Indeed, when the input matrix A has slowly decaying Singular Values it appears that implementing a random svd on it will result on having higher error. To fix this problem, Halko et al have proposed the Power iteration algorithm (Algorithm 4.4) corresponding to Stage A on the precedent algorithm.

This would reduce the relative weight of smaller singular values and without changing them. Indeed, knowing that $A = U\Sigma V^*$, It can be demonstrated by simply calculating the SVD of $(AA^*)^q A$ which led to : $\boldsymbol{U\Sigma^{2q+1}V^*}$. And because the SVD is unique the singular values remains the same, only the weights would be changed.

But as said above, stage A may sometimes have some flaws being indeed vulnerable to round-off errors. Thus the following algorithm was more recommended by Halko et al.

Algorithm 2: Randomized Subspace Iteration

Given an $m \times n$ matrix A and integers l and q, this algorithm computes an $m \times l$ orthonormal matrix Q whose range approximates the range of A.

1 Draw an $n \times l$ standard Gaussian matrix Ω.
2 Form $Y_0 = A\Omega$ and compute its QR factorization $Y_0 = Q_0R_0$.
3 **for** $j \leftarrow 1$ **to** q **do**
4 Form $\tilde{Y}_j = A^*Q_{j-1}$ and compute its QR factorization $\tilde{Y}_j = \tilde{Q}_j\tilde{R}_j$.
5 Form $Y_j = A\tilde{Q}_j$ and compute its QR factorization $Y_j = Q_jR_j$.
6 **end**
7 $Q = Q_q$.

The algorithm presented above called Randomized Subspace Iteration orthonormalize the columns of the sample matrix between each application of A and A^*. And it is algebraically equivalent to the previous stage A being thus executed in exact arithmetic.

2.2.2. Computation Errors

In their document, Halko et al have also demonstrated an important result allowing us to know **the approximate error** of our algorithm at all times. And allow us to compare it at the same time with **the theoretical minimum error:**

$$e_l = \left\| A - Q^l Q^{*l} A \right\| \tag{3}$$

Then, they claims that the theoretical minimum of any low-rank matrix approximation of A is equal to the **(j+1)th singular value:**

$$\min_{rank(x) \leq j} \|A - x\| = \sigma_{j+1} \tag{4}$$

X being equal to : $\boldsymbol{QQ^*A}$

In the prototype algorithm they used $l = 2 \times k$ (for reasons we already give above) but in fact we could use only $l = k + p$ with $p = 5$. Indeed, Halko et al gave us on their Theorem 1.1 the following inequation :

$$E \|A - QQ^*A\| \leq \left[1 + \frac{4\sqrt{k+p}}{p-1} \cdot \sqrt{\min(m,n)} \right] \sigma_{k+1} \tag{5}$$

where E denotes expectation with respect to the random test matrix and σ_{k+1} is the (k + 1)th singular value of A.

The theorem asserts that, on average, the algorithm produces a basis whose error lies within a small polynomial factor of the theoretical minimum.

Then the probability that the error satisfies :

$$\|A - QQ^*A\| \leq \left[1 + 9\sqrt{k+p} \cdot \sqrt{\min(m,n)}\right] \sigma_{k+1} \tag{6}$$

is at least equal to $1 - 3.p^{-p}$ under very mild assumptions on p. Justifying thus why one can use a smaller number of oversampling $p = 5$

2.3. Implementation, results and discussion

To illustrate our implementation of this framework, we used the **FERET Databank of faces images.** In fact, Halko et al have presented different kind of implementation to illustrate their work on section 7, and the Feret database belong to them.

We'll be using **Python** as programing langage due to its high usage on the field of data science. Plus, it has many packages such as Numpy and Scipy which have roots on C and Fortran, both being languages said to be close to the machine and therefore very efficient. Thus calculation must be reasonably fast enough.

We will use in our experiment a sample of **2722 images** of human faces, each of dimensions 384x256 which thus represent 98 304 pixels. The loading of such a quantity will therefore lead to the creation of **a dense matrix A of size 98 304 x 2722.** In fact there are many more images inside of it (something like 14 000), but here there is no need to take them all in order to show how well these random algorithm are performing.

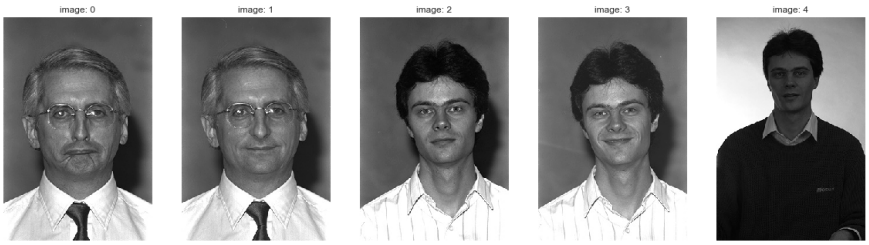

Figure 1: some samples from the FERET database

Before dealing with it, we'll first center each of its columns and scale it to unit norm. This will allow a better comparison of these images, it should be noted that this can be very resource consuming due to the huge dimension of the matrix. After that one can simply save the matrix as npy type for a potential future reuse.

2.3.1. Accuracy of the Framework

One way to verify how our results are is to compute the different approximation error $e_l = \left\|A - Q^l Q^{*l} A\right\|$ as said above in section 2.2.2. While increasing each time q as advised in the document because our matrix containing all our images has slowly decaying values. Which is equivalent in the Halko et al document to the figure 7.6.

Firstly let show how singular values' behaviour is when increasing the number of subspace iterations q. The matrix has the shape : $(AA^*)^q A$

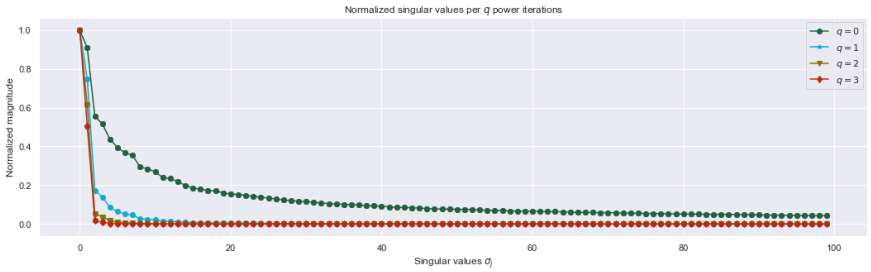

Figure 2: Influence of randomized subspace iteration upon decreasing the singular values of a Matrix

We can clearly see an improvement on how rapidly are decaying the singular values of our matrix.

Then, we'll try here to reproduce the Figure 7.6 of Halko et al paper. Thus for each sampling parameter l will be computed :

- $e_l = \left\| A - Q^l Q^{*l} A \right\|$: The actual error using the Frobenius norm.
- σ_{k+1} : The minimum rank k approximation error

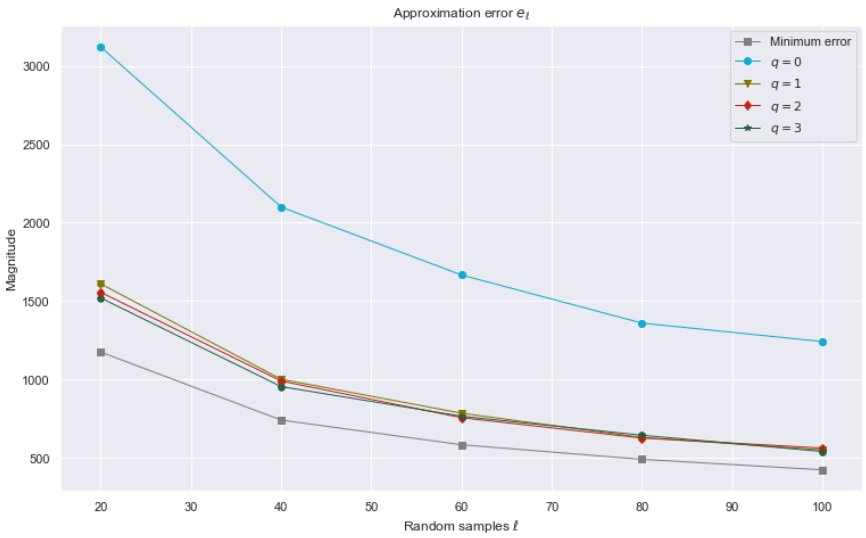

Figure 3: Illustration of the power iteration algorithm against the actual error

This experience shows us here that actually increasing the number of iterations of our subspace allows, while having a matrix with singular values which decrease faster, to have much more precise results. Here we notice that for q = 3 the error is almost the same as the theoretical minimum error of the full svd.

2.3.2. Time and Memory consumption

Let then talk about the most important thing after accuracy, which are both Time and Memory consumption of this framework when it's implemented.

Let thus show some figures on the precedent human faces Matrix by still using the same different samples l = 20, 40, 60, 80, 100 and the complete svd as comparaison.

We obtain the following results :

			time	memory
func_name	samples	q_power		
R-SVD	20	0	5.66 s	15.42 Mb
		1	2.44 s	15.42 Mb
		2	3.12 s	15.42 Mb
		3	4.37 s	15.42 Mb
	60	0	3.47 s	46.25 Mb
		1	4.43 s	46.25 Mb
		2	6.71 s	46.25 Mb
		3	9.68 s	46.25 Mb
	100	0	5.62 s	77.08 Mb
		1	9.05 s	77.08 Mb
		2	15.49 s	77.08 Mb
		3	17.98 s	77.08 Mb
	200	0	11.48 s	154.15 Mb
		1	16.95 s	154.15 Mb
		2	29.67 s	154.15 Mb
		3	41.80 s	154.15 Mb
SVD	full	None	107.04 s	2098.05 Mb

Figure 4: Time and Memory consumption : Random SVD vs Classical SVD

As shown in the figure above, we have a **very good improvement here.** Indeed, for the Time part even for $l = 200$ and $q = 3$ we compute it in almost 1/3 time of the classical svd. And more important here, the Memory part is completely negligible next to the classical svd. Having indeed **saved about 13 times the storage initially required.**

But it must be mentioned that such result are completely relative to the computer used during it. In fact we used here a PC with a processor intel core i5 with maximum speed of 3.5GHz. Same calculus have also been made on the super calculator named Hydra hosted at Ecole Centrale de Marseille and results were 5 times faster (without any specific optimisation of the algorithm to use the complete potential of it).

The most important fact here is that on a typical everyday computer, such an algorithm can actually be very good and perform better than using a better computer to compute a complete SVD.

2.3.3. Reconstruction of a Noisy Picture

Finally, among all the stuff we can accomplish by doing a SVD, denoising of data is one of them. Thus we can compute a simple reconstruction on a blured image of a human face, which was also taken from the Feret Dataset but that has not been part of the training of our Matrix containing all of our features.

Let recall first that : $A = U\Sigma V^*$.

The U matrix contains all the features that make up a human face. These are contained in each of its columns which therefore represent a precise image. In the figure below some samples of its columns :

Figure 5: Some samples from U columns

On the other hand, $\boldsymbol{\Sigma V^*}$ represent our **weights**. That is to say the proportion according to which each of the columns of U must be chosen to reconstruct correctly our image.

Thus here, we just need to multiplicate the matrix involved by our flooted image with our Weight Matrix containing all the informations needed here : $A = \tilde{U}\Sigma V^*$. Moreover, there is no need to take all of our singular values as the largest one's are already accurate enough to get good results.

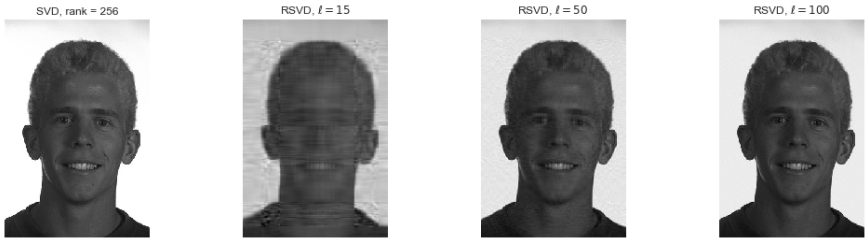

Figure 6: Image reconstruction with random SVD

We can observes that the more higher the number of sampling we chose the more precise is the reconstruction of our noisy image.

One can also, when dealing with images, create a classification model. The first strongest singular values are those which, generally, contains all the information that are the same among all people (eyes, nose, mouth. . .). And the first principal component that captures the most variance in this data are the information that most of us have in common. The information that is different from person to person are present in the lower singular values. And will be thus playing an important role to the classification.

3. Conclusion

To conclude, we have during this project studied a popular framework to approximate low rank matrices via a random SVD. Then we were able to verify some of the different theoretical and numerical results provided in the document by Halko et al.

The use of random in linear algebra calculations is a remarkably powerful and effective tool. Beyond the theoretical aspect, this approach was able to show its strengths as we have seen through our previous results. Thus, if initially a raw computation required a certain time and a memory much too consequent. The application of such algorithms could approach the expected result with very high precision while using extremely weak resources. Thus allowing us to run this kind of algorithms on more modest computers.

The use of randomness is however only possible for tools as long as the approximation of our results is approved by the user. Finally, there would also be other ways to speed up our implementations even more by running them on distributed computing platforms, like Spark for example. Which also gives the possibility of being used with Python, being the language of our study.

Acknowledgements

I would like to thank Mr.Emiya Valentin for the time he took to give me more information to understand this study. As well as for the choice of the subject in question which allowed me to learn more about an area of interest to me.

References

[1] N.Halko, P.G. Martinsson,J.A. Tropp, Finding structure with randomness: Probalistic algorithms for constructing approximate matrixdecompositions, Siam Journals (2011) p.217–288.

[2] National Institute of Standards and Technology, The FERET database : https://www.nist.gov/itl/products-and-services/color-feret-database