# SVD Factorization for Tall-and-Fat Matrices on Map/Reduce Architectures

Burak Bayramlı

October 15, 2013

**Abstract**

We demonstrate an implementation for an approximate rank-k SVD factorization, combining well-known randomized projection techniques with previously implemented map/reduce solutions in order to compute steps of the random projection based SVD procedure, such QR and SVD. We structure the problem in a way that it reduces to Cholesky and SVD factorizations on $k \times k$ matrices computed on a single machine, greatly easing the computability of the problem.

## 1 Introduction

[**?**] presents many excellent techniques for utilizing map/reduce architectures to compute QR and SVD for the so-called tall-and-skinny matrices. QR factorization is turned into an $A^TA$ computation problem to be computed in parallel using map/reduce, and its key element the Cholesky decomposition, can be performed on a single machine. Let's use $C = A^TA$ and, since

$$C = A^TA = (QR)^T(QR) = R^TQ^TQR = R^TR$$

and because Cholesky factorization of an $n \times n$ symmetric positive definite matrix is

$$C = LL^T$$

where $L$ is an $n \times n$ lower triangular matrix, and $R$ is upper triangular, we can conclude if we factorize $C$ into $L$ and $L^T$, this implies $C = LL^T = RR^T$, we have a method of calculating $R$ of QR using Cholesky factorization

on $A^T A$. The key observation here is $A^T A$ computation results an $n \times n$ matrix and if $A$ is "skinny" then $n$ is relatively small (in the thousands), then Cholesky decomposition can be executed on a small $n \times n$ matrix on a single computer utilizing an already available function in a scientific computing library. $Q$ is computed simply as $Q = AR^{-1}$. This again is relatively cheap because $R$ is $n \times n$, the inverse is computed locallly, matrix multiplication with $A$ can be performed through map/reduce.

SVD is an additional step. SVD decomposition is

$$A = U\Sigma V^T$$

If we expand it with $A = QR$

$$QR = U\Sigma V^T$$

$$R = Q^T U\Sigma V^T$$

Let's call $\tilde{U} = Q^T U$

$$R = \tilde{U}\Sigma V^T$$

This means if we run a local SVD on $R$ (we just calculated above with Cholesky) which is an $n \times n$ matrix, we will have calculated $\tilde{U}$, the real $\Sigma$, and real $V^T$.

Now we have a map/reduce way of calculating QR and SVD on $m \times n$ matrices where $n$ is small.

### 1.1 Approximate rank-k SVD

Switching gears, we look at another method for calculating SVD. The motivation is while computing SVD, if $n$ is large, creating a "fat" matrix which might have columns in the billions would require reducing the dimensionality of the problem. According to [**?**], one way to achieve is through random projection. First we draw an $n \times k$ Gaussian random matrix $\Omega$. Then we calculate

$$Y = A\Omega$$

We perform QR decomposition on $Y$

$$Y = QR$$

Then form $k \times n$ matrix

$$B = Q^T A$$

Then we can calculate SVD on this small matrix

$$B = \hat{U}\Sigma V^T$$

Then form the matrix

$$U = Q\hat{U}$$

The main idea is based on

$$A = QQ^T A$$

if replace Q which comes from random projection Y,

$$A \approx \tilde{Q}\tilde{Q}^T A$$

Q and R of the projection are close to that of A. In the multiplication above R is called B where $B = \tilde{Q}^T A$, and,

$$A \approx \tilde{Q}B$$

then, as in [**?**], we can take SVD of B and apply the same transition rules to obtain an approximate U of A.

This approximation works because of the fact that projecting points to a random subspace preserves distances between points, or in detail, projecting the n-point subset onto a random subspace of $O(\log n/\epsilon^2)$ dimensions only changes the interpoint distances by $(1 \pm \epsilon)$ with positive probability [**?**]. It is also said that Y is a good representation of the span of A.

## 1.2 Combining Both Methods

Our idea was using approximate k-rank SVD calculation steps where $k << n$, and using map/reduce based QR and SVD methods to implement those steps. By utilizing random projection, we would be able to work in a smaller dimension which would translate to local Cholesky, and SVD calls on $k \times k$ matrices that can be performed in a speedy manner. Below we outline each map/reduce job.

First random projection job (whose reduce is a no-op). Each value of A will arrive to the algorithm as a key and value pair. Key is line number or

```
random_projection_map(key, value)
1   input A
2   returns Y
3   Tokenize value and pick out id value pairs
4   result = zeros(1,k)
5   for each j^th token ∈ value
6       Initialize seed with j
7       j = generate k random numbers
8       result = result + r · token[j]
9   emit key, result
```

```
A^T A cholesky_job_map(key k, value a)
1   for i, row in enumerate a^T a
2       emit i, row
```

other identifier per row of A. Value is a collection of id value pairs where id is column id this time, and value is the value for that column. Sparsity is handled through this format, if an id for a column does not appear in a row of A, it is assumed to be zero. The resulting Y matrix has dimensions $m \times k$.

The `cholesky_job_final_local_reduce` step is a function provided in most map/reduce frameworks, it is a central point that collects the output of all reducers, naturally a single machine which makes it ideal to execute the final Cholesky call on by now a very small ($k \times k$) matrix. The output is R.

There is no reducer in the `Q_job`, it is a very simple procedure, it merely computes multiplication between row of Y and a local matrix R. Matrix R is very small, $k \times k$, hence it can be kept locally in every node. The initialiation is used to store the inverse of R locally, once the mapper is initialized, it will always use the same $R^{-1}$ for every multiplication.

The job above takes an AQ matrix which is assumed to be a join between A and Q, per row, based on key. We split the row and deduce the A part and the Q part, then iterate cells of A one by one, which is assumed to

```
cholesky_job_reduce(key, value)
1   emit k, sum(v_j^k)
```

```
cholesky_job_final_local_reduce(key, value)
1   result = cholesky(A_sum)
2   emit result
```

```
Q_job_map(key, value)
1   During initialization R_inv = R^{-1}, store it once for each mapper
2   for row in Y
3       emit key, row · R_inv
```

```
A^T Q_job_map(key, value)
1   left = row from A
2   right = row from Q
3   for each non-zero j^th cell in left
4       emit j, left[j] · right
```

```
A^T Q_job_reduce(key, value)
1   returs B^T
2   result = zeros(1, k)
3   for row in value
4       result = result + row
5   emit key, result
```

```
   QŨ_job_map(key, value)
1      input Q, R
2      returns U
3      initialization Ũ = svd of R
4      right = row from Q
5      for  row in Q
6          emit  key, row · Ũ
```

```
   map_reduce_svd
1      Y = random_projection_map(A)
2      R_Y = A^T A_cholesky_job(Y)
3      Q_Y = Q_job
4      R_BT = A^T A_cholesky_job(B^T)
5      U = QŨ_job(R_BT, Q)
```

be sparse, and multiply the entire row of Q. Then for each $j^{th}$ non-zero cell of A, we multiply this value with the row from Q and emit the multiplication result with key j.

The $Q^T A$ job's formula can be seen at **??**. For implementation purposes we changed this formula into

$$B^T = A^T Q$$

because as output we needed to have a $n \times k$ matrix instead of a $k \times n$ one, which would allow us to use map/reduce SVD that translates into a local Cholesky and SVD on $k \times k$ matrices. Since we take SVD of $B^T$ instead of B, that changes the output as well,

$$B = U\Sigma V^T$$

becomes

$$B^T = V\Sigma U^T$$

In other words, in order to obtain U of B, we need to take $(U^T_{BT})^T$ from the SVD of $B^T$. This is how $A^T A$ Cholesky Job is called, this time with $B^T$ as its input data.

### 1.3 Discussion

We performed our experiments on the Netflix dataset which has about 100 million from over 480,000 customers on 17770 movies. The implementation was programmed on Sasha distributed framework [**?**], and SVD calculation on the full dataset with $k = 7$ on two notebook computers, utilizing in total 6 cores took 20 minutes. Scipy SVD calculation on the same dataset is much faster, however, we need to stress our algorithms are prepared for cases where N is very large, i.e. in the billions. As such, for example during projection we did not simply create and pre-store a $n \times k$ random matrix and multiply multiple rows of $A$ with this matrix. This would certainly be possible for Netflix data where $n$ is relatively small, but would not work well in cases where $A$ is "fat". All code relevant for this paper can be found here [**?**].

There are only two passes necessary on the full dataset, and three passes on $m$ rows but with reduced $k$ dimensions this time. Perhaps predictably, the procedure spends most of its time at $A^T Q$ Job. This step performs not only a join between $A$ and $Q$, it also emits $k$ cells per non-zero value of $A$'s rows, then creates partial sums these $k$ vectors creating $n \times k$ result. If for simplicity we assume $k$ non-zero cells in each $A$ row, the complexity of this step would be $O(mk)$.

## References


[1] Gleich, Benson, Demmel, *Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures*, `arXiv:1301.1071 [cs.DC]`, 2013

[2] N. Halko, *Randomized methods for computing low-rank approximations of matrices*, University of Colorado, Boulder, 2010

[3] S. Dangupta, A. Gupta *An Elementary Proof of a Theorem of Johnson and Lindenstrauss*, Wiley Periodicals, 2002

[4] M. Kurucz, A. A. Benczúr, K. Csalogány, *Methods for large scale SVD with missing values*, ACM, 2007

[5] B. Bayramli, *Sasha Framework*, `git@github.com:burakbayramli/sasha.git` Github, 2013

[6] B. Bayramli, *Map/Reduce Code for Netflix SVD Analysis*, `http://github.com/burakbayramli/classnotes/tree/master/stat/stat_mr_rnd_svd/sasha`, Github, 2013