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

[1] presents many excellent techniques for utilizing map/reduce architectures to compute QR and SVD for the so-called tall-and-skinny matrices. The idea is based on the fact that QR factorization can be turned into an $A^T A$ computation problem computer in parallel, en masse using map/reduce, and through this to a Cholesky decomposition performed on a single machine. Since

$$A^T A = (QR)^T (QR) = R^T Q^T QR = R^T R$$

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

$$A = LL^T$$

where L is an $n \times n$ lower triangular matrix, and R is upper triangular, we can conclude if we factorize A into L and L^T, this implies $LL^T = RR^T$, 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), and Cholesky decomposition can be executed on a small $n \times n$ matrix on a single computer. 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 Σ, 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 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 [2], one way to achieve is through random projection. First we draw an $n \times k$ Gaussian random matrix Ω. 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 [1], 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 [3]. It is also said that Y is a good representation of the span of A.

1.2 Combining Both Methods

What if n is also very large? In this case local Cholesky or SVD computations would take a long time as well. Our idea was using approximate k-rank SVD where $k << n$, before map/reduce based QR and SVD methods presented in section 1, in order to reduce dimension. This way, we are again able to work with small matrices, locally, $k \times k$ this time on which Cholesky, SVD can be performed in a speedy manner. Below we outline each map/reduce job.

Algorithm 1: Random Projection Job

```
input  : A
output: Y
function MAP(key, value)
    Tokenize value and pick out id value pairs
    result ← zeros(1,k)
    for each j^th token ∈ value do
        Initialize seed with j
        r ← generate k random numbers
        result ← result + r · token[j]
    end
    emit key, result
function REDUCE(key, value)
    noop
```

Each value of A will arrive to the algorithm as a key and value pair. Key is line number or 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 $m \times k$.

Algorithm 2: $A^T A$ Cholesky Job

```
input  : Y
output: R
function MAP(key k, val a)
    for i, row in enumerate(a^T a) do
        emit i, row
    end
function REDUCE(key, value)
    emit (k,sum(< v_j^k >)
function FINAL LOCAL REDUCE (key, value)
    result ← Cholesky(A_sum)
    emit (result)
```

The 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.

Algorithm 3: Q Job

```
input  : Y,R
output: Q
function INIT()
    R_inv = R^{-1}
function MAP(key, value)
    for row in Y do
        emit (key, row · R_inv)
    end
```

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 $INIT$ function 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.

Algorithm 4: $A^T Q$ Job

```
input  : A,Q
output: B^T
function REDUCE (key, value)
    for row in value do
        if row is from A then
            left = row
        end
        if row is from Q then
            right = row
        end
    end
    for nonzero j^th cell in left do
        emit j, left[j] · right
    end
function REDUCESUM (key, value)
    result ← zeros(1,k)
    for row in value do
        result ← result + row
    end
    emit key, result
```

The job above takes A and Q matrices at the same time. Both of these matrices are based on the same key (line number, or other pre-existing id) and we need to join them first. If a mapper is a pass-through mapper, in other words if it does not exist, it is assumed to simply re-emit the key and value, which will, indirectly force matching rows with the same id from A and Q to be sent to the first reducer. Then for each unique id, the first reducer gets exactly two rows. This is an indirect way of performing a join in map/reduce environment.

Once we have these two rows we need to deduce if the row is a Q row or an A row. We need this information because which row we iterate depends on it. We prefer to iterate cells of A one by one, which is assumed to 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.

This job's formula in 1.1 is described $Q^T A$. 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 $(U_{BT}^T)^T$ from the SVD of B^T. That usage can be seen in the job below:

Algorithm 5: $Q\tilde{U}$ Job

input : Q,R
output: U
function *INIT()*
  $\tilde{U}$ = svd of R
function *MAP(key, value)*
  for *row in* Q **do**
    emit *(key, row* $\cdot \tilde{U}$*)*
  end

The order of execution for everything is as follows:

Algorithm 6: Map/Reduce SVD

Y = Random Projection Job (A)
$R_Y = A^T A$ Cholesky Job(Y)
$Q_Y =$ Q Job
$B^T = A^T Q$ Job
$R_{BT} = A^T A$ Cholesky Job(B^T)
$U = Q\tilde{U}$ Job(R_{BT}, Q)

1.3 Conclusion

We performed few experiments on the Netflix challange dataset which has 17K columns, 700K rows with %1 non-zero values. The implementation was coded on Hadoop through mrjob framework. SVD calculation on the full dataset with $k = 7$ on a single notebook computer running on two cores took 2 hours. Numbers reported by [4] for Lanczos SVD on the same dataset reports runtimes ranging from 2 to 7 hours. The added benefit of our algorithm is that it can scale horizontally, and linearly proportional to the number of nodes in a cluster. All code relevant for this paper can be found at `https://github.com/burakbayramli/classnotes/tree/master/stat/stat_hadoop_rnd_svd`

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