CS 598: Communication Cost Analysis of Algorithms

Lecture 21: Approximate low-rank dense matrix and tensor factorizations

Edgar Solomonik

University of Illinois at Urbana-Champaign

November 2, 2016

Low rank factorizations

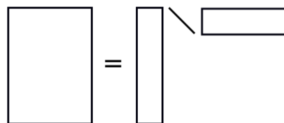
In the next lectures, we will explore ways to express data and operators in a reduced form

- a basic application is image/video compression
  - data is a dense matrix/tensor
- an advanced application is recommender systems
  - given a sparse subset of a matrix/tensor, predict unseen data
- a suitable low rank factorization can identify desired features
- for matrices, especially sparse ones, a low rank representation can be hard to compute, but is efficient to apply
- for tensors, we will often aim to find a reduced-*order* representation, e.g. by writing a 3rd order tensor as a product of matrices

Rank-k Singular Value Decomposition (SVD)

For any matrix $A \in \mathbb{R}^{m \times n}$ of rank k there exists a factorization

$$A = UDV^T$$



- $U \in \mathbb{R}^{m \times k}$ is a matrix of orthonormal left singular vectors
- $D \in \mathbb{R}^{k \times k}$ is a nonnegative diagonal matrix of singular values in decreasing order
- $V \in \mathbb{R}^{n \times k}$ is a matrix of orthonormal right singular vectors
- if A is symmetric $U = V$ and D are the eigenvalues

Low rank matrix approximation

Given $A \in \mathbb{R}^{m \times n}$ of rank at least k what is its best rank k approximation?

$$B = \underset{B \in \mathbb{R}^{m \times n}}{\operatorname{argmin}}(||A - B||_F)$$

- Frobenius norm reminder: $||X||_F^2 = \sum_{ij} X(i,j)^2$
- Eckart-Young theorem: let $A = UDV^T$ then

$$B = U(:, 1:k)D(1:k, 1:k)V(1:k, :)$$

this is the **truncated SVD** of A

Computing the SVD

We briefly discussed eigenvalues computation for symmetric matrices in a previous lecture

- reduce matrix to tridiagonal form then use MRRR or similar algorithm
- use QR for elimination, perform two-sided updates
- updates affect each-other, so parallelization more complex than QR
- achieving good cache complexity and synchronization cost requires reducing to a banded matrix
- band-to-band reduction done again by elimination via QR factorization, however, updates create fill, which must be chased down the band
- computing k singular vectors can be done by reordering accordingly and reapplying all orthogonal transformations to identity matrix

Cost of computing the SVD

A recent algorithm for tridiagonalization of a square matrix obtains cost

$$O\left(T_{\text{MM}}(n,P) + \frac{n^2\log(P)}{\sqrt{cP}}\cdot\nu + \sqrt{cP}\log^2(P)\cdot\alpha\right)$$

where $T_{\text{MM}}(n,P)$ is the cost of multiplying $n\times n$ matrices

- as before $M = O(cn/P)$ for $c\in[1,P^{1/3}]$
- logarithmic factors of overhead with respect to LU and QR
- requires $\Theta(\log(P))$ band-to-band reductions
- each reduction requires $O(n^2k)$ computation for computing k singular vectors, since fill structure isn't preserved in back-transformations
- algorithm is then work and interprocessor communication-efficient for computing up to $n/\log(P)$ singular vectors

Rank-revealing QR

If A is of rank k and its first k columns are linearly independent

$$A = Q \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

where R_{11} is $k \times k$ and $Q = YTY^T$ with $n \times k$ matrix Y

- to obtain this factorization for arbitrary A we need column ordering Π

$$A = QR\Pi$$

- column pivoting (proposed by Golub) is an effective method for this
  - pivot so that the leading column has largest 2-norm
  - method can break in the presence of roundoff error (see Kahan matrix), but is very robust in practice

Low rank factorization with pivoted QR

QR with column pivoting can be used to either

- determine the (numerical) rank of A
- compute a low-rank approximation with a bounded error
- Q: how many operations are executed if we stop at column k?
- A: the amount of computation is $O(mnk)$
- compare to $O(mn^2)$ for a full QR or SVD

Cache-efficient pivoted QR

Lets consider the memory-bandwidth cost of QR with column pivoting

- a challenge arises in the need for computing column norms
- updating the trailing matrix for every column would require

$$O(n^3 \cdot \nu)$$

  memory-cache traffic assuming $H < n^2$
- however, we can circumvent computing the full update, by exploiting its orthogonality
- Q: if Q is orthogonal, what do we know about $||w||_2 = ||Qv||_2$?
- A: $||w||_2 = ||v||_2$, the update does not change the norm of the column
- since the next trailing matrix excludes the top row, it each column norm can be updated with $O(1)$ operations
- this property permits an efficient "BLAS-3" implementation

Communication cost of pivoted QR

In distributed-memory, column pivoting poses further challenges

- need at least one message to retrieve each column, which leads to $\Omega(k)$ synchronizations
- to find a solution, we can take motivation from tournament pivoting
  - selects a set of rows at a time for LU factorization
- Q: would selecting a set of columns with high norm be a good heuristic?
- A: no, they may be linearly dependent and annihilated by the same orthogonal transformation
- what we need is a set of columns that are linearly independent

Communication-avoiding rank-revealing QR

[Demmel, Grigori, Gu, Xiang 2015] propose

- tournament of RRQR (rank-revealing QR) factorizations
  - For $A \in \mathbb{R}^{m\times b}$ with $m \gg b$ can compute $A = QR\Pi$, by first computing $A = Q_1 R_1$, then $R_1 = Q_2 R\Pi$ and $Q = Q_1 Q_2$
- consider $m \times n$ matrix on a $p_r \times p_c$ processor grid
- recursively find b linearly independent columns from first $n/2$ and last $n/2$ columns of the matrix, X and Y in parallel, then compute

$$Z = \begin{bmatrix} X & Y \end{bmatrix} \Pi^T = QR$$

  and pick b columns $Z[:, 1 : b]$
- base case ($n = b$, $p_c = 1$) return columns (slight modification of above reference)

Cost analysis of CARRQR

Lets analyze the cost of this algorithm in BSP, for each of k/b steps

- focus on the RRQR tournament, the rest is the same as 2D QR
- tournament tree is of height $O(\log(n/b))$
- so long as $b < n/p_c$, each step is dominated by $m \times b$ QR
- done by processor column of height p_r with row recursion

$$T_{\mathrm{TSQR}}(m, b, p_r) = O((mb^2/p_r + b^3 \log(p_r)) \cdot \gamma + b^2 \log(p_r) \cdot \beta + \log(p_r) \cdot \alpha)$$

- therefore, the total additional cost is $\log_2(n/b) T_{\mathrm{TSQR}}(n, b, p_r)$
- Q: assume $m = n = k$ and $p_r = p_c = \sqrt{p}$, what b do we need so as to have bandwidth cost $O(n^2/\sqrt{P} \cdot \beta)$?
- A: we need $b = O(n/(\sqrt{P} \log^2(P)))$

Short pause

Tensors

A tensor is a multidimensional matrix T, with elements $T(i, j, k, \ldots)$

- the **order** of the tensor is the number of **modes** (indices $i, j, k, \ldots$)
- the **dimensions** are the ranges of the modes
- Kolda and Bader 2009 provide a good review of tensor factorizations
  - we will be using somewhat different notation
  - they denote tensors $\mathcal{T}$, matrices and vectors as $\mathbf{M}$, $\mathbf{v}$
- it is common to fold tensors into higher order tensors and unfold into lower order
- we will denote folding via, e.g. $T(i \circ k, j, l) = W(i, j, k, l)$, where if the range of i is n

$$T(i + kn, j, l) = W(i, j, k, l)$$

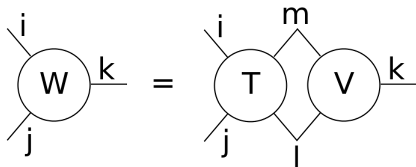
- we will also omit often quantifiers $\forall$ and write $\sum_i \sum_j$ as $\sum_{ij}$

Tensor contractions

A tensor contraction is a generalization of matrix and vector products, e.g.

$$W(i,j,k) = \sum_{lm} T(i,l,j,m) \cdot V(m,k,l)$$

its often convenient to express contractions and factorizations by diagrams



Any tensor contraction can be reduced to matrix multiplication $C = AB$, for instance for the contraction above,

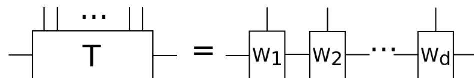
$$A(i \circ j, l \circ m) = T(i,l,j,m), \quad B(l \circ m, k) = V(m,k,l)$$

and $C(i \circ j, k) = W(i,j,k)$

Tensor train

A *low-order* tensor factorization expresses a tensor as contractions of smaller order ones

- an easy to compute and practical factorization is the tensor train
- given order $d+2$ tensor T, factorize into d tensors



- Q: what order is each factor W_i?
- A: 3
- other variants of the factorization for an order d tensor define W_1 and W_d to be order 2, but the intermediate nodes need to be at least order 3
- the quality of the factorization depends on the dimension of the internal (contracted/auxiliary) modes

Computing a tensor train factorization

The tensor train decomposition can be computed using low-rank matrix factorizations

- consider low rank matrix factorization $A = UV$ where $A \in \mathbb{R}^{m \times n}$ and $U \in \mathbb{R}^{m \times k}$, $V \in \mathbb{R}^{k \times n}$ and $k \leq n \leq m$
- we can compute $A = UV$ via SVD or QR with column pivoting or some other method
- a tensor train factorization of T can be computed via matrix factorization of $T(i_1, i_2 \circ i_3 \circ \cdots \circ i_d) = UV$ followed by a tensor train factorization of (the refolded) V
- other orderings to break apart the indices are also possible
  - for a given accuracy, the resulting internal modes will have different rank depending on the ordering

CP decomposition

The most natural generalization of the matrix SVD is the CP decomposition

- decompose a tensor T of any order d into matrices

$$T(i_1, \ldots, i_d) = \sum_{k=1}^{R} \lambda(k) \cdot U_1(i_1, k) \cdot \ldots U_d(i_d, k)$$

- R is referred to as the **tensor rank**
- the problem of computing the rank is NP hard and the approximation problem is ill-posed

Matrix multiplication algorithms via CP decomposition

A very important application of the CP decomposition is the search for Strassen-like matrix multiplication algorithms

- in Strassen's algorithm and other fast matrix multiplication algorithms, we compute R linear combinations of elements of A, which can be defined by U

$$\forall k \in [1, R] \quad \bar{A}(k) = \sum_{ij} U(i,j,k)A(i,j)$$

- and R linear combinations of elements of B, defined by V

$$\forall k \in [1, R] \quad \bar{B}(k) = \sum_{ij} V(i,j,k)B(i,j)$$

- finally these combinations are accumulated into C, as defined by W

$$C(i,j) = \sum_{k=1}^{R} \quad W(i,j,k)\bar{A}(k) \cdot \bar{B}(k)$$

Matrix multiplication algorithms via CP decomposition

Overall the matrix multiplication problem can be defined as

$$C(i_1,j_1) = \sum_{i_2 j_2 i_3 j_3} T(i_1,j_1,i_2,j_2,i_3,j_3)A(i_2,j_2)B(i_3,j_3)$$

- where $T(i_1,j_1,i_2,j_2,i_3,j_3) = 1$ when $i_1 = i_2$, $j_2 = j_3$ and $j_2 = i_3$
- we compute matrix multiplication via the linear combinations defined by U, V, and W

$$C(i_1,j_1) = \sum_{k=1}^{R} W(i_1,j_1,k) \sum_{i_2 j_2 i_3 j_3} (U(i_2,j_2,k)A(i_2,j_2))(V(i_3,j_3,k)B(i_3,j_3))$$

- rearranging this we obtain

$$C(i_1,j_1) = \sum_{i_2 j_2 i_3 j_3} \left[\sum_{k=1}^{R} W(i_1,j_1,k)U(i_2,j_2,k)V(i_3,j_3,k)\right] A(i_2,j_2)B(i_3,j_3)$$

- so $W(i_1 \circ j_1, k)$, $U(i_2 \circ j_2, k)$, $V(i_3 \circ j_3, k)$ are CP factors of $T(i_1 \circ j_1, i_2 \circ j_2, i_3 \circ j_3)$ with rank R

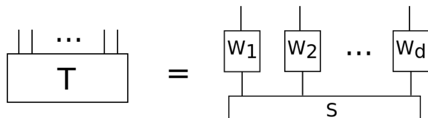
Matrix multiplication algorithms via CP decomposition

The rank of the CP decomposition of the 3rd order unfolding of T gives the number of multiplications that need to be done

- when all dimensions of T are 2, i.e. it represents multiplication of matrices with dimension 2×2, the rank of T is 7 (Strassen's algorithm)
- let the CP decomposition of T with dimensions $m \times n \times m \times k \times k \times n$ have rank R
- we can construct a matrix multiplication algorithm with R recursive calls that reduces the problem by a factor of mnk
- for multiplications of matrices $N \times N$ we would have complexity $O(N^{\omega})$ where $\omega = 3 \log_{mnk}(R)$

Tucker decomposition

The Tucker decomposition provides a more general decomposition for an order d tensor T



- unlike the CP decomposition, we have a *core tensor* S of order d

$$T(i_1, \ldots, i_d) = \sum_{k_1 \cdots k_d} S(k_1, \ldots, k_d) \cdot W_1(i_1, k_1) \cdot \ldots W_d(i_d, k_d)$$

- we achieve an improvement if the dimensions of S are less than T
- the CP decomposition can be seen as having a 'diagonal' tensor S
- Q: is Tucker a low-order factorization?
- A: no, the order of S is the same as T, it should be interpreted as a low-rank decomposition

Alternating least squares (ALS)

ALS is the standard procedure for computing CP and Tucker decompositions

- assumes there is a given rank R
- contract together all components except one into V, yielding $T = U_i \cdot V$
- optimize U_i, for instance solve for U_i in $T = U_i V$ under constraint that columns of U_i are orthogonal
- alternate components (optimize U_j)
- Tucker permits more general optimization, since we can modify S
- ALS gives no general convergence guarantees