Structure exploiting Lanczos method for Hadamard product of low-rank matrices

Milan Reljin
milan.reljin@epfl.ch

Abstract

Hadamard product, comes up frequently in scientific computing and data analysis. Unfortunately, due to rank multiplication and increase, algorithms that rely on low-rank approximation can take significantly more time when run on Hadamard product of two, or more, matrices. We design efficient algorithm for low-rank approximation of said product. Our algorithm is based on Lanczos method, and we make use of the structure of Hadamard product to create fast matrix-vector multiplication, which is crucial for the efficiency of the algorithm.

I. Introduction

In this report we provide a fast algorithm for low-rank approximation of Hadamard product. Given two matrices $\mathbf{A} \in \mathbb{R}^{n\times m}$, $\mathbf{B} \in \mathbb{R}^{n\times m}$ Hadamard product $\mathbf{A}*\mathbf{B}$, or element-wise product, is defined as a matrix $(a_{ij}b_{ij})_{ij} \in \mathbb{R}^{n\times m}$.

Hadamard product comes up frequently in algorithms throughtout scientific computing and data analysis. This fact has many reasons, however the most relevant one for this report is that Hadamard product represents product of two smooth functions. This can be deduced by the following. Let $f, g\colon [0,1]^2 \to \mathbb{R}$ be smooth functions, i.e. functions that are differentiable everywhere (and hence continuous). Let further $0 = x_1 < x_2 < ... < x_h = 1$, $0 = y_1 < y_2 < ... < y_q = 1$, be the discretization of their domain. Defining matrices $\mathbf{A} \coloneqq (f(x_i, y_j))_{ij} \in \mathbb{R}^{h\times q}, \mathbf{B} \coloneqq (g(x_i, y_j))_{ij} \in \mathbb{R}^{h\times q}$, we can see that $\mathbf{A}*\mathbf{B} = (f(x_i, y_j) \times g(x_i, y_j))_{ij}$ corresponds to discretization of $f \times g$.

It is known that matrices $\mathbf{A}, \mathbf{B}$ admit good low-rank approximation. Furthermore their Hadamard product can also be well approximated by low-rank matrices. Unfortunately, this does not directly translate algebraically, due to rank multiplication (and increase), and consequently can result in a computational bottleneck in algorithms which rely on low-rank matrix representation. However, by exploiting the structure of given matrices we'll design efficient algorithm for Hadamard product representation. Our algorithm makes use of Lanczos method for approximation of singular values, and we define fast matrix-vector multiplication method by exploiting low-rank structure of $\mathbf{A}$ and $\mathbf{B}$.

All of our work is based on [1]. Code which reproduces our results is publicly available in the following repository: *github.com/reljadev/lanczos-project*.

Preliminaries: Before describing our algorithm, we first provide definitions and known properties which we'll make use of. Given that non-standard matrix products are used less often and some of the readers may be unfamiliar with them, we here give the definitions of all the products we use. We also define singular value decomposition, as it represents significant part in the design of the algorithm.

Hadamard product. Let $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\mathbf{B} \in \mathbb{R}^{n \times m}$,

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}}_{\mathbf{A}} * \underbrace{\begin{bmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{bmatrix}}_{\mathbf{B}} = \underbrace{\begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1m}b_{1m} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2m}b_{2m} \\ \vdots & & \ddots & \vdots \\ a_{n1}b_{n1} & a_{n2}b_{n2} & \dots & a_{nm}b_{nm} \end{bmatrix}}_{\mathbf{A} * \mathbf{B}} \in \mathbb{R}^{n \times m}$$

Kronecker product. Let $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\mathbf{B} \in \mathbb{R}^{k \times r}$,

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}}_{\mathbf{A}} \otimes \underbrace{\begin{bmatrix} b_{11} & b_{12} & \dots & b_{1r} \\ b_{21} & b_{22} & \dots & b_{2r} \\ \vdots & & \ddots & \vdots \\ b_{k1} & b_{k2} & \dots & b_{kr} \end{bmatrix}}_{\mathbf{B}} = \underbrace{\begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1m}B \\ a_{21}B & a_{22}B & \dots & a_{2m}B \\ \vdots & & \ddots & \vdots \\ a_{n1}B & a_{n2}B & \dots & a_{nm}B \end{bmatrix}}_{\mathbf{A} \otimes \mathbf{B}} \in \mathbb{R}^{nk \times mr}$$

Khatri-Rao product. Let $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\mathbf{B} \in \mathbb{R}^{k \times m}$,

$$\underbrace{\begin{bmatrix} | & | & & | \\ a_1 & a_2 & \dots & a_m \\ | & | & & | \end{bmatrix}}_{\mathbf{A}} \odot \underbrace{\begin{bmatrix} | & | & & | \\ b_1 & b_2 & \dots & b_m \\ | & | & & | \end{bmatrix}}_{\mathbf{B}} = \underbrace{\begin{bmatrix} | & | & & | \\ a_1 \otimes b_1 & a_2 \otimes b_2 & \dots & a_m \otimes b_m \\ | & | & & | \end{bmatrix}}_{\mathbf{A} \odot \mathbf{B}} \in \mathbb{R}^{nk \times m}$$

Transpose Khatri-Rao product. Let $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\mathbf{B} \in \mathbb{R}^{n \times r}$,

$$\underbrace{\begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_n^T & — \end{bmatrix}}_{\mathbf{A}} \odot^T \underbrace{\begin{bmatrix} — & b_1^T & — \\ — & b_2^T & — \\ & \vdots & \\ — & b_n^T & — \end{bmatrix}}_{\mathbf{B}} = \underbrace{\begin{bmatrix} — & a_1^T \otimes b_1^T & — \\ — & a_2^T \otimes b_2^T & — \\ & \vdots & \\ — & a_n^T \otimes b_n^T & — \end{bmatrix}}_{\mathbf{A} \odot^T \mathbf{B}} \in \mathbb{R}^{n \times mr}$$

Properties. We provide several of the known properties we'll make use of. We give no proof of these. In the following, $vec\colon \mathbb{R}^{n \times m} \to \mathbb{R}^{nm}$ is the standard matrix vectorization. Furthermore, $diag\colon \mathbb{R}^{n \times n} \to \mathbb{R}^{n}$ denotes extraction of diagonal elements from square matrix, and we use the same notation for creating a diagonal matrix $\mathbf{D}$ from vector $\mathbf{v}$, i.e. $\mathbf{D} = diag(\mathbf{v})$, where $diag\colon \mathbb{R}^{n} \to \mathbb{R}^{n \times n}$, and $\mathbf{D}_{ii} \coloneqq \mathbf{v}_i$.

$$(\mathbf{A} \odot \mathbf{B})\, \mathbf{v} = vec\left(\mathbf{B} diag\left(\mathbf{v}\right) \mathbf{A}^{\mathbf{T}}\right) \tag{1}$$

$$(\mathbf{A} \otimes \mathbf{B})\, \mathbf{v} = vec\left(\mathbf{B}\mathbf{V}\mathbf{A}^{\mathbf{T}}\right),\ \mathbf{v} = vec\left(\mathbf{V}\right) \tag{2}$$

$$\left(\mathbf{A} \odot^T \mathbf{B}\right) \mathbf{v} = diag\left(\mathbf{B}\mathbf{V}\mathbf{A}^{\mathbf{T}}\right),\ \mathbf{v} = vec\left(\mathbf{V}\right) \tag{3}$$

Singular value decomposition of a matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ is a factorization into three matrices, such that $\mathbf{A} \coloneqq \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathbf{T}}$, where $\mathbf{U} \in \mathbb{R}^{n \times k}, \mathbf{\Sigma} \in \mathbb{R}^{k \times k}, \mathbf{V} \in \mathbb{R}^{m \times k}$, and k is rank of matrix $\mathbf{A}$. In addition, matrices $\mathbf{U}$ and $\mathbf{V}$ are such that $\mathbf{U}^{\mathbf{T}}\mathbf{U} = \mathbf{V}^{\mathbf{T}}\mathbf{V} = \mathbf{I}_{\mathbf{k}}$, and $\mathbf{\Sigma}$ is a diagonal matrix. Columns of $\mathbf{U}$ and $\mathbf{V}$ are called **singular vectors** and diagonal entries of $\mathbf{\Sigma}$, i.e. $\sigma_{\mathbf{i}} \coloneqq \mathbf{\Sigma}_{\mathbf{ii}}$ are **singular values**. It is known that every real-valued rectangular matrix admits such a factorization.

II. ALGORITHM

A. Representation of Hadamard product

We will make use of the following lemma in the design of our algorithm. We give no proof of this lemma, but curious reader could deduce it by using known properties of Khatri-Rao, Kronecker and Hadamard product.

Lemma 2.1: Let $\mathbf{A} \in \mathbb{R}^{n\times m}$ and $\mathbf{B} \in \mathbb{R}^{n\times m}$ be matrices with the low-rank structure

$$\mathbf{A} = \mathbf{U_A}\mathbf{\Sigma_A}\mathbf{V_A^T}, \quad \mathbf{B} = \mathbf{U_B}\mathbf{\Sigma_B}\mathbf{V_B^T}$$

where $\mathbf{\Sigma_A} \in \mathbb{R}^{k_A\times k_A}, \mathbf{\Sigma_B} \in \mathbb{R}^{k_B\times k_B}$ and $k_A, k_B \ll min\{n, m\}$, then Hadamard product admits the following representation

$$\mathbf{A} * \mathbf{B} = \left(\mathbf{U_A} \odot^T \mathbf{U_B}\right)\left(\mathbf{\Sigma_A} \otimes \mathbf{\Sigma_B}\right)\left(\mathbf{V_A^T} \odot \mathbf{V_B^T}\right) \tag{4}$$

B. Fast matrix-vector multiplication

As matrix dimensions m and n can be huge, the resulting matrix $\mathbf{A} * \mathbf{B}$ might not fit into memory. In addition computing the Hadamard product directly would require $O(nm)$ time, which would result in major computational bottleneck. This is why we design faster matrix-vector multiplication wihtout forming the product $\mathbf{A} * \mathbf{B}$ explicitly.

Let $\mathbf{A} \in \mathbb{R}^{n\times m}, \mathbf{B} \in \mathbb{R}^{n\times m}$ and $\mathbf{v} \in \mathbb{R}^m$, then using the hadamard product representation, as well as matrix product properties we can calculate the following

$$\begin{aligned}
(\mathbf{A} * \mathbf{B})\mathbf{v} &\overset{(4)}{=} (\mathbf{U_A} \odot^T \mathbf{U_B})(\mathbf{\Sigma_A} \otimes \mathbf{\Sigma_B})(\mathbf{V_A^T} \odot \mathbf{V_B^T})\mathbf{v} \\
&\overset{(1)}{=} (\mathbf{U_A} \odot^T \mathbf{U_B})(\mathbf{\Sigma_A} \otimes \mathbf{\Sigma_B})vec(\mathbf{V_B^T} diag(\mathbf{v})\mathbf{V_A}) \\
&\overset{(2)}{=} (\mathbf{U_A} \odot^T \mathbf{U_B})vec(\mathbf{\Sigma_B}\mathbf{V_B^T} diag(\mathbf{v})\mathbf{V_A}\mathbf{\Sigma_A}) \\
&\overset{(3)}{=} diag(\mathbf{U_B}\mathbf{\Sigma_B}\mathbf{V_B^T} diag(\mathbf{v})\mathbf{V_A}\mathbf{\Sigma_A}\mathbf{U_A^T})
\end{aligned}$$

We end the calculation there, even though last step can be further simplified, $\mathbf{B} = \mathbf{U_B}\mathbf{\Sigma_B}\mathbf{V_B^T}$ and $\mathbf{A} = \mathbf{V_A}\mathbf{\Sigma_A}\mathbf{U_A^T}$, because we refrain ourselves from working directly with matrices $\mathbf{A}$ and $\mathbf{B}$, so as to avoid high computational and memory use, as previously mentioned.

The fast matrix-vector multiplication algorithm is implemented in 3 steps.
Step 1. set $w = \mathbf{V_B^T}(diag(\mathbf{v})\mathbf{V_A})$
Step 2. set $w = (\mathbf{\Sigma_B}\mathbf{w})\mathbf{\Sigma_A}$
Step 3. set $w = diag((\mathbf{U_B}\mathbf{w})\mathbf{U_A^T})$

In the following analyzation of the steps complexity, we won't make distinction between k_A and k_B and we'll denote both with k, so as to save on notation.

First step takes $O(mk^2)$, because we don't form diagonal matrix. Instead we simply multiple elements of matrix $\mathbf{V_A}$ with entries of vector $\mathbf{v}$, which takes $O(mk)$ time.

Second step is implemented as a straight-forward matrix-vector and matrix-matrix multiplication and requires $O(k^3)$ time.

Final step crucially doesn't form the whole $(\mathbf{U_B w})\mathbf{U_A^T}$ matrix, because it would result in quadratic time complexity with respect to n. Instead, we calculate only the diagonal entries, in time $O(nk)$, thus the whole step requires $O(nk^2)$.

At the end vector w is equal to the $(\mathbf{A} * \mathbf{B})\mathbf{v}$. Total complexity of fast-matrix vector multiplication is $O(mk^2 + nk^2 + k^3)$, which depends on n and m only linearly.

Same type of method can be used for $(\mathbf{A} * \mathbf{B})^T\mathbf{v}$ multiplication, which we'll also need in the main algorithm.

C. Lanczos method

Lancozs method, designed for approximation of eigenvalues and eigenvectors of symmetric matrix, can be used for approximation of a matrix as well, which was first demonstrated by Simon and Zha [2]. Algorithm pseudo-code follows.

Algorithm 1: Lanczos method

input : Tolerance **tol**, Number of iterations **maxIt**, Starting vector $\mathbf{x}$
output: Tridiagonal matrix $\mathbf{H}$
set $\mathbf{u} = \mathbf{x}/\left\|\mathbf{x}\right\|, \mathbf{U} = [\mathbf{u}]$;
for $k \leftarrow 1$ **to** $maxIt$ **do**
  $\mathbf{r} = \mathbf{A}\mathbf{u}$;
  $\alpha_\mathbf{k} = \mathbf{u^T r}$;
  $\mathbf{r} = \mathbf{r} - \alpha_\mathbf{k}\mathbf{u}$;
  Orthogonalize $\mathbf{r}$ versus the columns of U, and reorthogonalize every iteration;
  $\beta_\mathbf{k} = \|\mathbf{r}\|$;
  if $\beta_\mathbf{k} <$ **tol then**
    break;
  end
  $\mathbf{u} = \mathbf{r}/\beta_\mathbf{k}$;
  $\mathbf{U} = [\mathbf{U}\ \ \mathbf{u}]$;
end
$\mathbf{H} = \text{tridiag}((\alpha_\mathbf{1} \dots \alpha_\mathbf{k}), (\beta_\mathbf{1} \dots \beta_\mathbf{k}))$;

At the end of the algorithm we end up with matrix $\mathbf{U}$ with orthonormal columns and a tridiagonal matrix $\mathbf{H}$, such that the following holds

$$\mathbf{AU} = \mathbf{UH} + \beta_\mathbf{k}\mathbf{u_{k+1}}\mathbf{e_k^T}$$

from which follows $\mathbf{A} = \mathbf{UHU^T}$. Because of the heuristic criterion $\beta_k < tol$, approximation satisfies

$$\left\|\mathbf{A} - \mathbf{UHU^T}\right\| < tol \tag{5}$$

D. Computing eigenvalues and eigenvectors using Gramian matrix

Instead of computing singular value decomposition of matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ directly, we can obtain it from eigenvalues/vectors of it's gramian, $\mathbf{AA^T}$. To see this, let's suppose that

$$\mathbf{A} = \mathbf{U\Sigma V^T}, \ \ \mathbf{U}, \mathbf{V} \text{ are orthogonal} \tag{6}$$

Then we can re-write $\mathbf{AA^T}$ as

$$\mathbf{AA^T} = \mathbf{U\Sigma V^T V\Sigma U^T} = \mathbf{U\Sigma^2 U^T}$$

This means that computing eigendecomposition of $\mathbf{AA^T}$, and taking the square-root of it's eigenvalues we obtain left singular vectors and singular values of $\mathbf{A}$. To get right singular values as well, we note that the expression (6) can be rewritten as

$$\mathbf{A^T u_i} = \sigma_\mathbf{i} \mathbf{v_i}, \;\; i = 1, 2 \ldots n$$

or equivalently

$$\mathbf{v_i} = \frac{\mathbf{A^T u_i}}{\sigma_\mathbf{i}}, \;\; i = 1, 2 \ldots n$$

where $\sigma_\mathbf{i}$ are diagonal entries of $\mathbf{\Sigma}$, and $\mathbf{u_i}, \mathbf{v_i}$ are columns of $\mathbf{U}, \mathbf{V}$ respectively.

Similar connection exists for matrix $\mathbf{A^T A}$. We prefer to work with $\mathbf{A^T A} \in \mathbb{R}^{n \times n}$, if $n \ll m$, and with $\mathbf{AA^T} \in \mathbb{R}^{m \times m}$ in case $m \ll n$.

E. Main algorithm

To put it all together now, let's denote $\mathbf{A} * \mathbf{B}$ by $\mathbf{C}$, where $\mathbf{A} \in \mathbb{R}^{n \times m}, \mathbf{B} \in \mathbb{R}^{n \times m}$. We first run Lanczos method on $\mathbf{CC^T}$ (or $\mathbf{C^T C}$). Note, we don't form matrix $\mathbf{C}$ explicitly, instead we use fast matrix-vector multiplication from section II-B. After algorithm terminates we obtain matrices $\mathbf{U}$ and $\mathbf{H}$ such that

$$\left\| \mathbf{CC^T} - \mathbf{UHU^T} \right\| < tol \;\; \text{(see (5))}$$

We then compute spectral decomposition of H such that

$$\mathbf{H} = \mathbf{V\Lambda V^T}, \;\; \mathbf{\Lambda} \in \mathbb{R}^{n \times n}, \mathbf{V} \in \mathbb{R}^{n \times n}$$

Setting $\mathbf{Q} = \mathbf{UV}$, we obtain

$$\left\| \mathbf{CC^T} - \mathbf{Q\Lambda Q^T} \right\| < tol$$

Finally, to get the low-rank approximation of $\mathbf{C}$ we use the trick described in the previous section.

III. Results

All our results were obtained using matrices $\mathbf{A}, \mathbf{B}$ which are formed by evaluating functions

$$f(x, y) = \frac{1}{x + y}, \;\; g(x, y) = \frac{1}{\sqrt{x^2 + y^2}}$$

respectively, on the grid $\{0.1, 0.2, \ldots M\}, \{0.1, 0.2, \ldots N\}$.

A. Rank Comparison

In order to compare the ranks we first obtained low-rank approximations of $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ by truncating their singular values lower than $\epsilon = 10^{-4}$. What we've found is that rank of truncated SVD of $\mathbf{C}$ is 7, rank of representation (4) is 10, and rank of our approximation is 8. This confirms that we have indeed found lower-rank representation of Hadamard product then the on in (4).

B. Precision

For both precision and time complexity comparison we've formed matrices $\mathbf{A}, \mathbf{B}$ by setting $M = N = 50, 100, 150, 200, 250, 300$. Tolerance was set to 10^{-8}, and number of maximum iterations was set to 30. Initial vector was chosen at random.

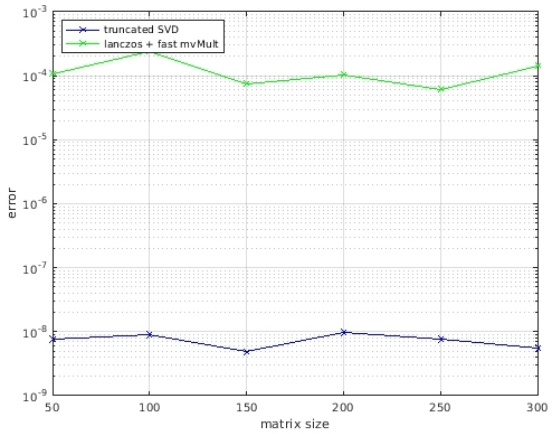

Figure 1: Errors of our algorithm and truncated SVD

As SVD is the best low-rank approximation, error of our method is greater. However, as we can see from the above figure, it's around 10^{-4}, which is still decent approximation, and it doesn't seem to correlate with the matrix size.

C. Time Complexity

The following figure depicts time taken to obtain truncated SVD directly as well as our low-rank approximation of Hadamard product.

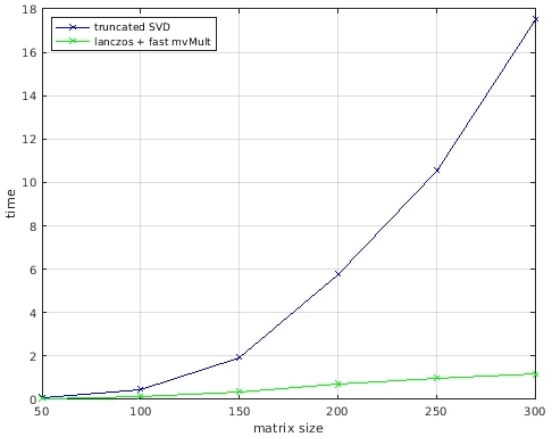

Figure 2: Time complexity of our algorithm and truncated SVD

As we can see, our algorithm takes significantly less time to terminate compared to SVD, which seems to grow exponentially with respect to matrix size. In particular, for the largest matrix we tested at size 300×300, SVD finishes after 18s, whereas our algorithm completes in under to 2s.

REFERENCES

[1] D. Kressner and L. Periša, "Recompression of hadamard products of tensors in tucker format," *SIAM Journal on Scientific Computing*, vol. 39, pp. A1879–A1902, 01 2017.

[2] H. Simon, "Low rank matrix approximation using the lanczos bidiagonalization process with applications," *SIAM Journal on Scientific Computing*, vol. 21, 10 1997.