Tensor Algebra
University Federal do Ceará
PPGETI - 2020.2
Saulo Mendes de Melo

Homework 2

Khatri-Rao Product

1. Generate $\mathbf{X} = \mathbf{A} \diamond \mathbf{B} \in R^{I \otimes R}$, for randomly chosen $\mathbf{A} \in C^{I \otimes R}$ and $\mathbf{B} \in R^{I \otimes R}$. Compute the left pseudo-inverse of $\mathbf{X}$ and obtain a graph that shows the run time vs. number of rows (I) for the following methods

   Note: Consider the range of values $I \in \{2, 4, 8, 16, 32, 64, 128, 256\}$ and plot the curves for $R = 2$ and $R = 4$.

   (a) Method 1: Matlab/Octave* function $pinv(\mathbf{X}) = pinv(\mathbf{A} \diamond \mathbf{B})$

   Solution: *This work uses instead the Python function *numpy.linalg.pinv()* from the Numpy package. The Khatri-Rao product function in this homework is custom made, and is listed in the Appendix 1. The code below shows the basic loop used for the experiments in **Problem 1**

```
runtimes = []
for e in range(100):

    sublist_runtimes = []
    list_runtimes = []

    for R in [2, 4]:
        for I in [2, 4, 8, 16, 32, 64, 128, 256]:

            A = randn(I, R)
            B = randn(I, R)

            A = A + randn(I, R)*1j # A  complex

            #### TIME MEASUREMENT ######
            start_time = time.time()
            X_pinv = pinv(khatri_rao(A, B))
            end_time = time.time()
            ###########################

            sublist_runtimes.append(end_time-start_time)

        list_runtimes.append(sublist_runtimes)
        sublist_runtimes = []
```

```python

    runtimes.append(list_runtimes)
    list_runtimes = []

runtimes = np.array(runtimes)
Method1_runtimes = np.mean(runtimes, axis=0)
```

In this loop, we run the same experiment 100 times and then average all the results, as to capture the possible variations in runtime due to external factors.

(b) Method 2: $\mathbf{X}^\dagger = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T = [(\mathbf{A}\diamond\mathbf{B})^T(\mathbf{A}\diamond\mathbf{B})]^{-1}(\mathbf{A}\diamond\mathbf{B})^T$

Solution: The method below implements the Method 2 taking $\mathbf{X}$ as input.

```python
def Method2_pinv(X):
    return np.matmul(inv(np.matmul(X.T, X)), X.T)
```

For this experiment the same basic loop from the previous item was used. In the line where the `Method2_pinv` function was called, we use `khatri_rao` with parameters $\mathbf{A}$ and $\mathbf{B}$ as input, as to account for the time elapsed to calculate $\mathbf{X}$.

```
X_pinv = Method2_pinv(khatri_rao(A,B))
```

(c) Method 3: $\mathbf{X}^\dagger = [(\mathbf{A}\diamond\mathbf{B})^T(\mathbf{A}\diamond\mathbf{B})]^{-1}(\mathbf{A}\diamond\mathbf{B})^T = [(\mathbf{A}^T\mathbf{A})\odot(\mathbf{B}^T\mathbf{B})]^{-1}(\mathbf{A}\diamond\mathbf{B})^T$

Solution: The method below implements the Method 3 taking $\mathbf{A}$ and $\mathbf{B}$ as input. This same function was used in the same experiment loop listed in the first item.

```python
def Method3_pinv(A, B):
    return np.matmul(inv(np.multiply(np.matmul(A.T, A), \\
                 np.matmul(B.T, B))), khatri_rao(A, B).T)
```

Results

The Methods listed describe algebraic formulas for the calculation of the Pseudo-inverse making use of different properties, which leads to very different complexities in their algorithms, and such, different average runtimes.

From the figure 1 it is clear that Method 1 is the worst case. For R=2 it starts comparable to other methods, but progressively gets slower as the number of rows in the matrices grows, showing a tendency characteristic of higher complexity algorithms. This is due to the fact that it most likely applies an algorithm for pseudo-inverse calculation that is generalized for any matrix, without making use inherent properties of the pseudo-inverse of a Khatri-Rao Product.

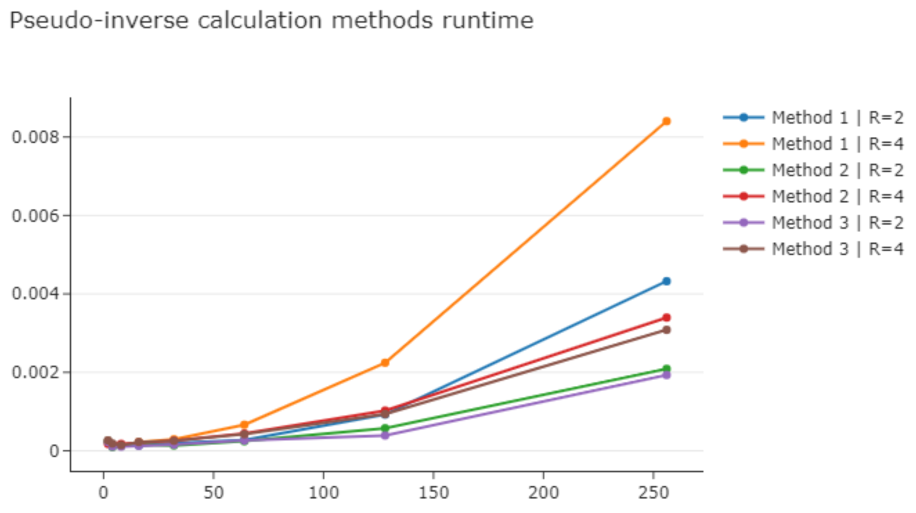

Figure 1: Pseudo-inverse calculation methods runtime

For both Method 2 and Method 3, the results form groups that increase together for each value of R, but in both cases, Method 3 is slightly faster than Method 2. An explanation for this case is that Method 2 involves more multiplications of $\mathbf{A} \diamond \mathbf{B}$ Products, which are become matrices with a very large number of rows, for the higher N values. The advantage of Method 3 is that two of the three Khatri-Rao Product matrices are substituted for elements that are made of common matrix multiplications and a Hadamard product, which do not lead to operations in huge numbers of rows.

2. Generate $\mathbf{X} = \diamond_{n=1}^{N} \mathbf{A}_{(n)} = \mathbf{A}_{(1)} \diamond ... \diamond \mathbf{A}_{(n)}$, where every $\mathbf{A}_{(n)}$ has dimensions 4×2, $n = 1, ..., N$. Evaluate the run time associated with the computation of the Khatri-Rao product as a function of the number N of matrices for the above methods.

   Note: Consider the range of values $N \in \{2, 4, 6, 8, 10\}$.

   Solution: The code loop used to calculate the runtimes for each N value is listed below. It runs the experiment 100 times and then calculates the average runtime for each N value.

```
total_runtimes = []

for e in range(100):
    runtimes = []

    for N in [2, 4, 6, 8, 10]:

        A_start = randn(4, 2)

        ##### TIME MEASUREMENT ###########
        start_time = time.time()
        for n in range(0, N):
            if n == 0:
                A = A_start
            else:
                A = khatri_rao(A, A_start)

        end_time = time.time()
        ###################################

        runtimes.append(end_time-start_time)

    total_runtimes.append(runtimes)

total_runtimes = np.array(total_runtimes)
problem2_runtimes = np.mean(total_runtimes, axis=0)
```

Results

The Khatri-Rao product applies the Kronecker Product between the columns of its inputs. This means that the row dimension of the columns increases with each application of Kronecker, leading to matrices with the same number of columns, but an ever larger number of rows. Figure 2 clearly denotes this tendency ocurring for **Problem 2**. For each n matrix being operated, the number of rows increases rapidly to the point where it takes much longer to calculate the next Khatri-Rao Product.

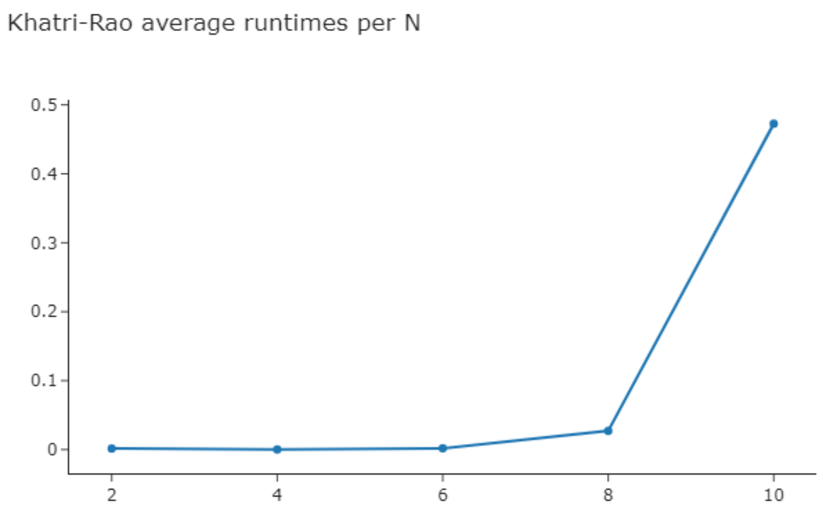

Figure 2: Khatri-Rao average runtimes per N value

Appendix 1: Code

```python
def khatri_rao(A, B):
    # Verify that matrices have the same number of columns
    assert A.shape[1] == B.shape[1]

    # transposing matrices as looping through rows is simpler in Python
    A = A.T
    B = B.T

    result = []

    for Ai in range(0, len(A)):
        for Bi in range(0, len(B)):
            if Ai == Bi:
                result.append(np.kron(A[Ai], B[Bi]))

    return np.array(result).T # transposing back to column format
```

Listing 1: Khatri-Rao Product implementation