A general-purpose toolbox for efficient Kronecker-based learning

Michiel Stock[1], Tapio Pahikkala[2], Antti Airola[2], and Bernard De Baets[1]

[1]KERMIT, Department of Data Analysis and Mathematical Modelling, Ghent University, Belgium
[2]Department of Future Technologies

ABSTRACT

Pairwise learning is a machine learning paradigm where the goal is to predict properties of pairs of objects. Applications include recommender systems, molecular network inference, and ecological interaction prediction. Kronecker-based learning systems provide a simple yet elegant method to learn from such pairs. Using tricks from linear algebra, these models can be trained, tuned, and validated on large datasets. Our Julia package `Kronecker.jl` aggregates these shortcuts and efficient algorithms using a lazily-evaluated Kronecker product '$\otimes$', such that it is easy to experiment with learning algorithms using the Kronecker product.

Keywords

Pairwise learning, Kronecker product, Linear algebra

1. Background

The Kronecker product, denoted by $\otimes$, between an $(n \times m)$ matrix $A = [A_{ij}]$ and an $(p \times q)$ matrix $B = [B_{kl}]$ is computed as

$$A \otimes B = \begin{bmatrix} A_{1,1}B & \cdots & A_{1,m}B \\ \vdots & \ddots & \vdots \\ A_{n,1}B & \cdots & A_{n,m}B \end{bmatrix}. \tag{1}$$

Simply put, the Kronecker product creates a new $(np \times mq)$ matrix containing all element-wise products between the respective elements of the two matrices.
Though conceptually simple, the Kronecker product gives rise to some elegant mathematics which allows performing many important computations, such as the eigenvalue decomposition, determinant or trace, in an efficient way [5, 7]. The Kronecker product has numerous applications in applied mathematics, for example in defining the matrix normal distribution, modeling complex networks [4] and pairwise learning [6]. The reason that one can use the Kronecker product in large numerical problems is that (1) often does not have to be computed explicitly, but it can be circumvented using various computational shortcuts.

2. Basic use

Our package aims to be a toolkit to effortless build Kronecker-based applications, where the focus is on the mathematics, and computational efficiency is taken care of under the hood. Essentially, it provides a lazily-evaluated Kronecker product of a `Kronecker` type.

```
(n, m), (p, q) = (20, 20), (30, 30);
# A and B do not have to be square
A = rand(n, m); B = randn(p, q);
K = kronecker(A, B)  # lazy Kronecker product
```

Alternatively, one can make use of Unicode, i.e. `K = A ⊗ B`. The elementary functions of `LinearAlgebra` are overloaded to work with the respective subtypes of `GeneralizedKroneckerProduct` and provide the most efficient implementation.

```
tr(K)  # computed as tr(A) * tr(B)
det(K)  # computed as det(A)^n * det(B)^q
eigen(K)  # kronecker(eigen(A), eigen(B))
inv(K)  # yields a Kronecker instance
v = randn(600);
K * v  # computed using the vec trick
```

For example, the last line is evaluated using the so-called "vec trick" [7] with a time complexity of $\mathcal{O}(nm + pq)$ instead of $\mathcal{O}(nmpq)$ naively. Similarly, efficiently solving large shifted Kronecker systems can be done directly as `eigen(A ⊗ B +λI) \ v`, exploiting the fast eigenvalue decomposition for Kronecker products.
Our package fully supports higher-order Kronecker products, e.g. `A ⊗ B ⊗ C`. The structure `KroneckerPower` (for example constructed as `kronecker(A, 4)` for $A \otimes A \otimes A \otimes A$) and its methods provide efficient storage and manipulation of repeated Kronecker multiplications of the same matrix. We also provide the functionality to generate Kronecker graphs.
We provide support for dealing with submatrices of a Kronecker product through the sampled vec trick [1].

```
# subsample a 200 x 100 submatrix of K
i, j = rand(1:n, 200), rand(1:m, 200);
k, l = rand(1:p, 100), rand(1:q, 100);
Ksubset = K[i,j,k,l];
u = randn(100);
Ksubset * u  # computed using sampled vec trick
```

3. Prospects

`Kronecker.jl` is a package in development. The developers are continuously adding new features. To fully make use of the power of Julia, we will explore three directions. Firstly, we will integrate libraries for automatic differentiation, such as `Zygote.jl` [3]. This will allow for developing pairwise learning methods with complex loss and regularization functions. Secondly, we want to leverage the GPU support to make these methods scalable to large datasets using `CuArrays.jl` [2]. Finally, we want to explore how symmetries and anti-symmetries can be incorporated, for example, when switching the order of two matrices would not change the result or only influences the sign of the result.

4. References

[1] Antti Airola and Tapio Pahikkala. Fast Kronecker product kernel methods via generalized vec trick. *IEEE Transactions on Neural Networks and Learning Systems*, 29(8):3374–3387, 2018.

[2] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):827–841, 2019.

[3] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B Shah, and Will Tebbutt. A differentiable programming system to bridge machine learning and scientific computing. 2019.

[4] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research*, 11:985–1042, 2008.

[5] Kathrin Schäcke. On the Kronecker Product. Technical report, 2013.

[6] Michiel Stock, Tapio Pahikkala, Antti Airola, Bernard De Baets, and Willem Waegeman. A comparative study of pairwise learning methods based on kernel ridge regression. *Neural Computation*, 30(8):2245–2283, 2018.

[7] Charles F. Van Loan. The ubiquitous Kronecker product. *Journal of Computational and Applied Mathematics*, 123(1-2):85–100, 2000.