P1416R0: SG19 Linear Algebra for Data Science and Machine Learning

Date:	2019-01-21(Pre-KONA mailing): 10 AM ET
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience	SG19, SG14, WG21

Authors :	Johann Mabille, Matthieu Brucher,
Contributors	SG19
Emails:	johan.mabille@gmail.com, matthieu.brucher@gmail.com
Reply to:	johan.mabille@gmail.com

Introduction

This paper describes the Linear Algebra tools required for Data Science and Machine Learning. It is based on the experience from other languages, mainly Python / Numpy. Data Science and Machine Learning are all about processing N-dimensional data. This process involves slicing, reshaping, filtering, training a model and predict with it.

This paper also gives feedback to the emerging SG14 paper on Linear Algebra design [Pxxxx] from a Machine Learning and Data Science Perspective. In that way, it is meant to be developed along with the SG14 paper. It is separate from the SG14 paper because the ML aspect is an add-on built on top of the mathematical foundation of the SG14 paper.

Motivation

For a very long time, the C++ Standard programming language lacked a high-level toolset for scientific computing, making it hard to efficiently write code for the full workflow of Data Science and Machine Learning.

This paper will start the description of a list of features inspired by Numpy, which found a wide adoption across all spectrums of Data Science.

Required Features

Multi-dimensional Arrays

The basic data structure used in Data Science is an N-dimensional array (typically 3-D or 4-D). This structure should be able to represent tensors of any rank, including vectors (1D, transpose is itself) and matrices (2D). 0-D arrays should not be handled as a special case, they should provide the same API and semantic as others, and should be convertible from / constructible from a scalar.

We refer to the number of axes and the sizes of these axes as the shape of the array. We can refer to the size of an axis as the dimension of this axis.

Componentwise operations

The most useful operations to preprocess data are componentwise operations. Therefore the N-dimensional array should provide overloaded operators for componentwise addition, subtraction, multiplication, and division.

Using operator* for componentwise multiplication (also known as the Hadamard product) instead of matrix multiplications might be counterintuitive regarding the trend in third-party libraries implementing Linear Algebra (Blitz, Eigen). However higher rank tensor dot may require additional arguments, making it impossible to implement with operator*. Therefore a free function for matrix multiplication is more consistent with this higher tensor dot.

This behavior is similar to what is done in Numpy.

Broadcasting

The term broadcasting describes the rules to apply for operations involving N-dimensional arrays with different shapes. Let's illustrate them with the addition of two arrays, A + B. We can distinguish 3 cases:

- A and B have the same number of dimension, matched up dimensions of the two arrays differ and one of them has size 1: it is broadcast to match the size of the other. This can be illustrated with the following example:
  (x, 1, z) # A
  (x, y, z) # B

  (x, y, z) # R
- A and B have different numbers of dimensions, but matched up dimensions are the same: the array with lesser dimensions is broadcast across the leading dimension of the other, as illustrated below:
  (x, y, z) # A

(y, z) # B

(x, y, z) # R
This case can be combined with the previous one.

- A and B matched up dimensions differ and none of them has size 1: A and B have incompatible shape and the operation should throw an error:
  (x, y, z) # A
  (x, d, z) # B

  ERROR

This behavior is similar to what is done in Numpy.

Access operator

The broadcasting rules have consequences on the access operator: this latter should be able to accept more or less integral arguments than the number of dimensions of the array:

- if fewer arguments are provided, they should be prepended with 0 until the number of arguments matches the number of dimensions of the array
- if more arguments are provided, the most left arguments are removed until the number of arguments matches the number of dimensions of the array.

These rules give the guarantee that for any arrays A and B with compatible shapes (according to the broadcasting rules), the following is always true:

(A + B)(i0, …, iN) = A(i0, …. iN) + B(i0, …… iN)

This behavior is different from the one of Numpy, where passing more arguments to a ndarray than its number of dimension throws an error, while passing fewer arguments returns a view.

Views

A view can be a subset of an N-dimensional array, but also a broadcasting view, expanding the number of dimension of its underlying array. A view should not copy the data, so a change in the view should be reflected in the array.

This behavior is similar to the one of Numpy.

Nice to have

Slice notation

A common operation for data selection is slicing; Numpy provides a concise syntax that we could draw on. If A(i) returns the i-th element of the 1-D array A, A(i:e:s) returns a view that takes 1 element out of s from i to e.

Access operator

Access operator of matrices and N-D arrays in existing third-party libraries are usually implemented with operator(). operator[] could be changed to accept more than one index so that it can be used for accessing data in an N-dimensional array. This would make it consistent with 1-dimensional containers of the standard library.

Dot operator

Matrix multiplications and dot operators should not be implemented with operator*, but with free functions. this can be improved by introducing a new operator dedicated to these operations (Python introduces “@”).

SVD and other operations (solve, inverse…)

These operations can be implemented on top of other operations. SG14 paper tackle different layers, all more complex operations can be considered as layer 1+, whereas this paper tackles explicitly layer 0.