Photo by Adi Goldstein (https://unsplash.com/@adigold1) on Unsplash (https://unsplash.com/)

Working with arrays: broadcasting

6 minute read

In this lesson we will talk about one of the most handy, if not useful, features of Julia: **array broadcasting**. Furthermore we will deal with some **important differences** between **Julia** and other programming languages (like **Python** or **Matlab**) which make extensive use of operations on arrays and with arrays.

> If you are new to programming or you come from a compiled language (like C++), some of the considerations made in this lesson may sound redundant and useless, and they may even be, but they are intended to point out to users coming from Python or Matlab which are the main differences between those languages and Julia and indicate how to write fast, efficient and simple code in this language.

Working with functions and arrays

Introduction

Many languages (such as Python with **numpy** and **Matlab**) make extensive use of optimised C or Fortran routines under the hood to perform fast mathematical operations, as such the user is encouraged to write **vectorised code**, so that these routines can perform faster for loops, as a big part of the CPU time is spent on calling the underlying compiled routine, and not computing the actual result. More or less what happens is that the user writes vectorised code which communicates to C code that can run fast for loops and the result is then returned to the user in the form of an array (or matrix).

In **Julia**, since for loops are already as fast as they can be (close to the speed of C) there is **no need to write vectorised code**, as the interpreter will directly compile your code in optimised machine code which will run as fast as possible on your machine. In Julia **nothing happens under the hood** (beside the compilation of the functions) and almost all the functions in Julia are written in Julia, just like all the functions in C are written in C.

Operations with arrays

Julia by default deals with operations on arrays and matrices as one would do in mathematics.

Let's start with an example: from a mathematical point of view, we don't know how to compute the `sin` of an array, as the sine function is defined only on single (dimensionless) values. At the same time the `exp` can work both on single values and matrices (as the exponential of a matrix has a well-defined geometrical meaning). For the same reason, you cannot multiply two arrays together, unless their size is matching correctly (i.e. one array is a row array and the other one is a column array) and in this case the multiplication of two arrays becomes the well-defined geometrical product of two arrays (which can be a scalar or a matrix, depending on the order of the multiplication):

```
a = [1,2,3] # is a column vector
b = [4,5,6] # is a column vector

>>> a*b
ERROR: MethodError: no method matching *(::Array{Int64,1}, ::Array{Int64,1})

c = [4 5 6] # is a row vector

>>> a*c
    3×3 Array{Int64,2}:
      4   5   6
      8  10  12
     12  15  18

>>> c*a
    1-element Array{Int64,1}:
     32

d = reshape([1,2,3,4,5,6,7,8,9],3,3)
>>> d*a
    3-element Array{Int64,1}:
     30
     36
     42
```

This makes perfectly sense from a mathematical point of view and operators behave how we would mathematically expect. Nonetheless, in programming it is often useful to write operations which work on an element by element basis, and for this reason **broadcasting** comes to our help.

Broadcasting

In Julia, with **broadcasting** we indicate the action of mapping a function or an operation (which are the same in Julia) over an array or a matrix element by element.

The broadcasting notation for operators consists of adding a dot `.` before the operator (for example `.*`)

Considering the example we get:

```
>>> a .* c
    3×3 Array{Int64,2}:
      4   5   6
      8  10  12
     12  15  18

>>> c .* a
    3×3 Array{Int64,2}:
      4   5   6
      8  10  12
     12  15  18

>>> a .* d
    3×3 Array{Int64,2}:
     1   4   7
     4  10  16
     9  18  27
```

Notice that when we broadcast the multiplication with a matrix and an array, the array gets multiplied "in the same direction" as it is written, in the sense that if a vector is a column it gets applied column by column etc.

We can use the **broadcasting notation** also to **map a function over an n-dimensional array**. There is no speed gain in doing so, as it will be exactly equivalent to writing a for loop, but its conciseness may be useful sometimes. So the core idea in Julia is to **write functions that take single values** and use broadcasting when needed, **unless the functions must explicitly work on arrays** (for example to compute the mean of a series of values, perform matrix operations, vector multiplications, etc).

To broadcast a function over an array it is sufficient to put a dot before the brackets `.()`

```
a = [1,2,3]
>>> sin.(a)
    3-element Array{Float64,1}:
     0.8414709848078965
     0.9092974268256817
     0.1411200080598672
```

Tip: do not try to write vectorised code in Julia (like you would do in Python and Matlab) if you don't need to, even if you are used to coding this way, as the code will become less readable and more prone to errors. Use instead broadcasting and for loops, when needed, to map a function over several values.

It is also possible to map a function over several values using the `map` function, but there are no real advantages in doing so and the broadcasting syntax is often more flexible.

As we will see in the future, in Julia it is really easy to write parallel code using **multi-processing** and **multi-threading**, and it is particularly simple to write such parallel structures when we encounter a `for` loop or a `map` function (which will become a parallel map through `pmap`), thus it is convenient to get used to writing your code in such paradigm.

If you are new to programming or you come from a compiled language (like C++), you should find it easy to think in terms of for loops and you should think of broadcasting as a fast and concise way to compute the value of a function over several input values.

Conclusions

We have learned what is broadcasting and how it can be used to perform element by element operations between vectors and how to map a function over an array using the concise broadcasting syntax.

We have also pointed out how Julia does not gain in performance by using "vectorised" notations (while other languages do) because the core Julia operations are implemented directly in the Julia language and thus there is no need to call compiled routines coded in other languages under the hood to offload all the heavy work.

As a disclaimer, this guide in not meant to incite programming language elitism and criticise languages relying on other compiled routines to be efficient (I'm a Python user myself and in the end Julia is basically a really sophisticated wrapper around the LLVM, which is coded in C++). It is meant to point out the differences with other interpreted languages (like Python and Matlab) and show the capabilities of a language which is both compiled just in time and has an interpreter responsible for type inference. Thus **Julia solves the two languages problem** by being a programming language both easy to write and highly efficient (with speed comparable to C).

If you liked this lesson and you would like to receive further updates on what is being published on this website, I encourage you to subscribe to the **newsletter** (https://techytok.com/newsletter/)! If you have any **question** or **suggestion**, please post them in the **discussion below**!

Thank you for reading this lesson and see you soon on TechyTok!

Tags: Julia

Categories: Lessons Tutorial

Updated: November 25, 2019

LEAVE A COMMENT

Luuk • 8 months ago

Great article, thanks for the effort, looking forward to the next articles!
Using map or broadcasting a function, is that ‘under the hood’ equal and thus equal performance?

△ | ▽ • Reply • Share ›

aureamerio Mod > Luuk • 8 months ago

In Julia, using broadcasting is as efficient as using a for loop, while map may be a little bit slower depending on the situation.
If we compare Julia to Python's numpy (for example) since there is no data moving between libraries in Julia (i.e transferring the arrays to the optimized C routine which does the vectorised operations) it tends to be faster while using small arrays and it is as fast as Python with bigger arrays (in this case since most of the time is spent actually doing the computations and not transferring data, we are comparing compiled code, so the performance is expected to be more or less the same).

△ | ▽ • Reply • Share ›