Aug 18, 2017 · 6 min read

Speeding up your code (2): vectorizing the loops with Numpy

From this series:

1. The example of the mean shift clustering in Poincaré ball space
2. Vectorizing the loops with Numpy (this post)
3. Batches and multiprocess
4. In-time compilation with Numba

In the previous post I described the working environment and the basic code for clusterize points in the Poincaré ball space. Here I will improve that code transforming two loops to matrix operations.

I ended that post with a very promising plot about the speed improvement on a element-wise product of two vectors. So let's detail it.

Vectorized element-wise product

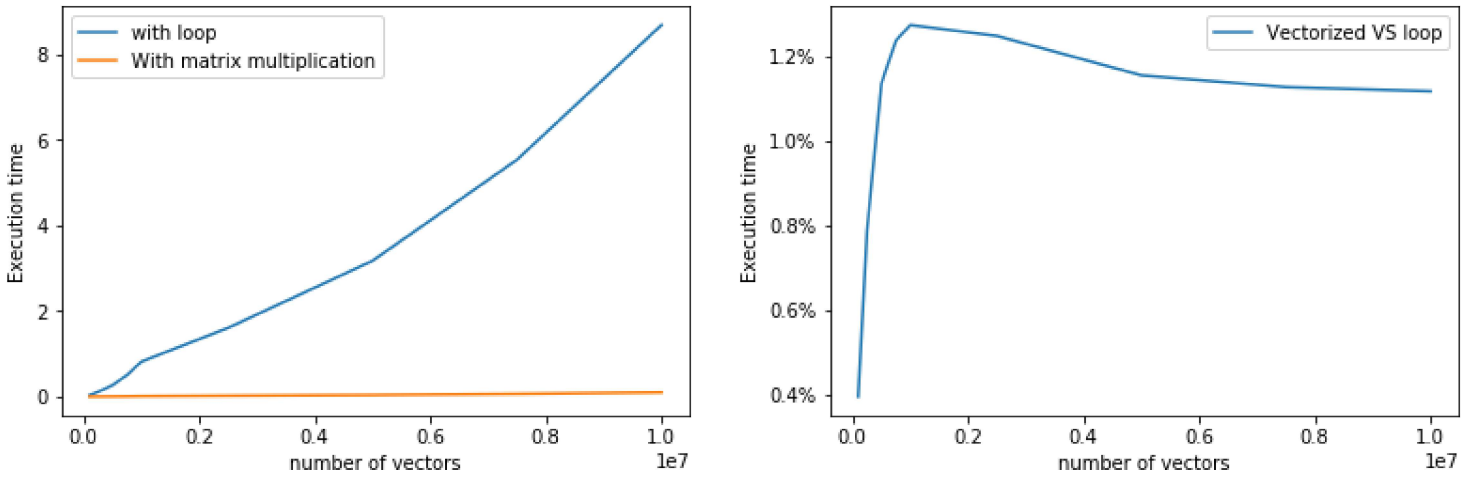
Suppose we have two arrays:

$$v_1 = (a_1, a_2, ..., a_n) \ , \ v_2 = (b_1, b_2, ..., b_n)$$

and we want to obtain as result an array where the elements are the element-wise multiplication of them:

$$res = (a_1 * b_1 \ , \ a_2 * b_2 \ , \ \dots \ , \ a_n * b_n)$$

We can do it in two ways: with a loop over the elements, or with a vectorized operation. Now: what happens in terms of execution time? I did this calculations with arrays of different dimensions, ranging from 100.000 to 10.000.000.



Execution time: relative on the left, absolute on the right.

In the right plot you see the execution times of the two operations: the vectorized version is MUCH faster than the looped one. How much faster? You see this in the left plot: the vectorized version is executed in less than 1.3% of the time!

Actually when we use the broadcasting capabilities of Numpy like we did in the previous post, under the hood all the operations are automatically vectorized. So using broadcasting not only speed up writing code, it's also faster the execution of it! In the vectorized element-wise product of this example, in fact i used the Numpy np.dot function.

Now, how can apply such strategy to get rid of the loops?

Vectorizing the loop in the distance function

Let's begin with the loop in the distance function. From the previous post:

```
def _dist_poinc(a, b):
    num=np.dot(a-b, a-b)
    den1=1-np.dot(a,a)
    den2=1-np.dot(b,b)
    return np.arccosh(1+ 2* (num) / (den1*den2))
def dist_poinc(a, A):
    res=np.empty(A.shape[0])
```

We execute this function for each vector of the collection: that's one of the loops we want to avoid. And we feed the function with all the vectors, one at a time (a) together with the whole collection (A): that's the other loop which we will vectorize.

If you change the perspective, you can see the collection of vectors as a matrix, and the vectors becomes just rows of the matrix. In this vision, the operations between the vectors becomes operations between the rows of the matrix. For example, 1000 vectors with two components will make a (1000,2) matrix:

$$\begin{bmatrix} v_{1a} & v_{1b} \\ v_{2a} & v_{2b} \\ \dots & \dots \\ v_{1000a} & v_{1000b} \end{bmatrix}$$

Example of 1000 vectors with two components

Let's step back to the original formula, and focus at each part of it separately:

$$d_{poinc} = arcosh\left(1 + 2\frac{\|a-b\|^2}{\left(1-\|a\|^2\right)\left(1-\|b\|^2\right)}\right)$$

Distance in Poincaré ball

where the double pipe character (like ||a||) means the euclidean norm.

For better visualize the following operations, I will simplify the notation by omitting the vectors' components. So when I will write *v1*, you have to remember that that vector has his own components. As consequence, the 2-d matrices which contains vectors will be 3-d matrices, with the vectors' components along the third, hidden, dimension.

The numerator

In the numerator of the fraction we have to do a subtraction between vectors, before calculating the squared norm. And we want to vectorize it, i.e. make all the subtractions in a single passage. In other words, we want to obtain this matrix:

$$\begin{bmatrix} v_1 - v_1 & v_1 - v_2 & v_1 - v_3 & \dots & v_1 - v_{1000} \\ v_2 - v_1 & v_2 - v_2 & \dots & \dots & v_2 - v_{1000} \\ \dots & \dots & \dots & \dots & \dots \\ v_{1000} - v_1 & \dots & \dots & \dots & v_{1000} - v_{1000} \end{bmatrix}$$

Matrix of the subtraction between all vector of the collection

This will be a symmetric matrix (element (1,2) is equal to element (2,1), etc) and will have all the diagonal elements equal to zero. This means that we actually double the calculations, but that's the better way of proceed along the next operations.

For obtaining such a matrix it's convenient to leverage on the broadcasting capabilities of Numpy. As example, if we focus for a moment to the first row of it, which is composed by the differences between v1 and all the vectors of the collection *S*, we can obtain it by simply call the subtraction *v1-S*. And given that we have to do the same operation with all the vectors, we have to execute the following operation:

$$\begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{1000} \end{bmatrix} - \begin{bmatrix} v_1 & v_2 & \dots & v_{1000} \\ v_1 & v_2 & \dots & v_{1000} \\ \dots & \dots & \dots & \dots \\ v_1 & v_2 & \dots & v_{1000} \end{bmatrix}$$

The subtraction operation. In the code, the first operand is called "trans", and the second "tiled". Note that broadcasting has been used.

At this point we have to calculate the squared norm of the obtained elements, i.e. we have to square everything in the matrix and then sum up those squares along the vectors' component axis, which is the omitted third dimension in the matrices, as already said.

Speaking in Python/Numpy language, this is the code for obtaining the numerator:

```
def num(points):
    expd=np.expand_dims(points,2) #need another dimension...
    tiled=np.tile(expd, points.shape[0]) #...to tile up the
    trans=np.transpose(points) #Also need to transpose the p
    diff=trans-tiled           #doing the difference, exploi
    num=np.sum(np.square(diff), axis=1) #an then obtain the
```

Code for obtaining the numerator of the formula

The denominator

Now the denominator. The two operands between parenthesis can actually be seen as an operation within the same matrix, because we are treating all the vectors at the same time. So beforehand we calculate the squared norm of all the vectors, then subtract the obtained vector from a vector composed by ones. Using broadcasting:

$$1 - \left[\|v_1\|^2 \quad \|v_2\|^2 \quad \dots \quad \|v_{1000}\|^2\right]$$

Now, to obtain the matrix with all the denominator terms we have to multiply this vector by the transpose of himself:

All this is done by the following code:

```
def den(points):
    sq_norm=1-np.sum(np.square(points),1) #subtracting from
    expd=np.expand_dims(sq_norm,1)   #this operation is need
    den_all=expd * expd.T #multiply the object by his transp
    return den_all
```

The whole distance formula, and the rest of mean shift

Now we have both the numerator and the denominator. All the rest of the operations to get the distances are just element-wise operations

made with broadcasting (Thank you Numpy!):

```
def poinc_dist_vec(points):
    num=num(points)
    den=num(points)
    return np.arccosh(1+2*num/ den)
```

The function we built gives as result a matrix with the distances between points in the Poincaré ball:

$$\begin{bmatrix} d(v_1,v_1) & d(v_1,v_2) & \dots & d(v_1,v_{1000}) \\ d(v_2,v_1) & d(v_2,v_2) & \dots & d(v_2,v_{1000}) \\ \dots & \dots & \dots & \dots \\ d(v_{1000},v_1) & d(v_{1000},v_2) & \dots & d(v_{1000},v_{1000}) \end{bmatrix}$$

The matrix of the distances between points.

Now we have to adapt the main function to use this matrix for running the mean shift algorithm, and at the same time we will get rid of the other loop.

Recalling the mean shift algorithm, we have to calculate the weights using a Gaussian kernel. For doing this we can use the same function we already prepared, because thanks to broadcasting it will work for a matrix too, and we will get as result a matrix with the same shape.

Then we have to use this matrix to weight all the vectors of the collection. This is a usual matrix product:

$$\begin{bmatrix} w(v_1,v_1) & w(v_1,v_2) & \dots & w(v_1,v_{1000}) \\ w(v_2,v_1) & w(v_2,v_2) & \dots & w(v_2,v_{1000}) \\ \dots & \dots & \dots & \dots \\ w(v_{1000},v_1) & w(v_{1000},v_2) & \dots & w(v_{1000},v_{1000}) \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{1000} \end{bmatrix} = \begin{bmatrix} v_{1w} \\ v_{2w} \\ \dots \\ v_{1000w} \end{bmatrix}$$

Matrix multiplication to get the weighted vectors. The first matrix is actually a 2-d matrix, but the 1-d arrays include the vectors' components hidden dimension.

And we have also to normalize each vector by the sum of the weights needed to build it. As example, for having the normalized version of *v1w* we have to divide it by the sum *w(v1,v1) + w(v1,v2) + …. + w(v1, v1000)*.

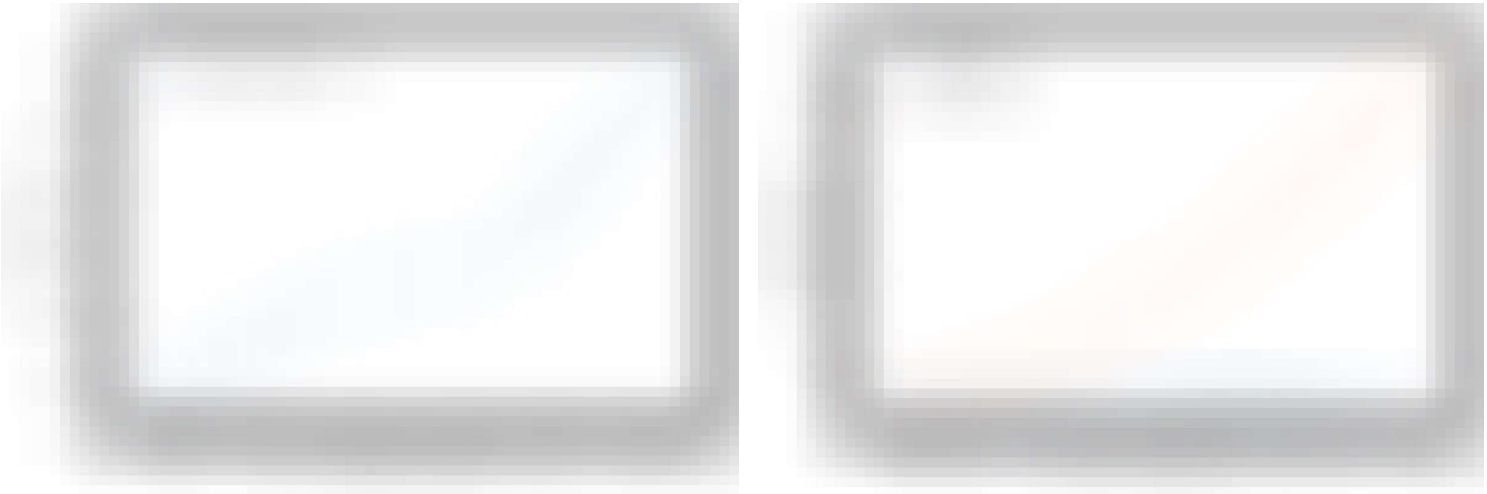
So we have to sum up the weight matrix along the rows, obtaining an array of summed weights that we will use for divide element-wise the weighted vectors.

Here the code needed for all those operations:

```
def gaussian(d, bw):
    return np.exp(-0.5*(d/bw)**2) / (bw*np.sqrt(2*np.pi))
def meanshift_vec(points):
    dists=poinc_dist_vec(points) #the matrix of the distance
    weights = gaussian(dists, sigma) #the matrix of the weig
    expd_w=np.dot(weights, points) #the weighted vectors
    summed_weight=np.sum(weights,0) # the array of the summe
```

And that's all!

Now let's see how much is faster with respect to the looped version. The following plots are made running both the algorithms with collections of two dimensional vectors.



Execution time for the mean shift algorithm: relative on the left, absolute on the right.

As expected, the vectorized version is much faster then the looped one: with 10.000 2-d vectors it took 844 seconds for execute the looped algorithm, while the vectorized version it's executed in just 28 seconds. It's to be said that the execution speed decrease with big dataset. This is

an expected behavior, because for N vectors we deal with N^2 tensor elements.

Now: what if we want to make it faster? And what if the number of vectors are so big that they cannot fit in memory?

In the next post I will show a batch version of the algorithm, i.e. an algorithm which will process the vectors a bunch at time. And thanks to this, we can execute it in a multiprocess way, where each CPU core will take care of each batch.

Stay tuned!