Numerical_Tricks_for_Python

January 22, 2022

1 Efficient Python for Numerical and Scientific Calculation

1.1 Outline

- Numpy
- Numba
- Multiprocessing
- torch as numpy with gpu
- lru_cache

[1]:
```
import numpy as np # numerical python
import numba # JIT complier for python
from multiprocessing import Pool # multiprocessing
import torch # ML library(it can be used for gpu calculation)
import time
import matplotlib.pyplot as plt
```

1.2 Numpy

1.2.1 Demo 1: numpy is faster than for-loop in python

[11]:
```
a = np.random.rand(10000)
b = np.random.rand(10000)
```

[12]:
```
%%time
a = a + b
```

```
CPU times: user 29 µs, sys: 5 µs, total: 34 µs
Wall time: 38.6 µs
```

[4]:
```
a = np.random.rand(10000)
b = np.random.rand(10000)
```

[5]:
```
%%time
for i in range(len(a)):
    a[i] = a[i] + b[i]
```

```
CPU times: user 2.71 ms, sys: 0 ns, total: 2.71 ms
Wall time: 2.72 ms
```

1.2.2 Why is python slow?

- **interprete instead of compile**: how the code is executed is not optimized
- **dynamic** language: the type of variables are not fixed (cost for analyzing type)

1.3 Quote From Numpy User Guide

1.3.1 What is NumPy?

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just "behind the scenes" in **optimized, pre-compiled C code**. Vectorized code has many advantages, among which are:

- vectorized code is more concise and easier to read
- fewer lines of code generally means fewer bugs
- the code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs)

vectorization results in more "Pythonic" code. Without vectorization, our code would be littered with inefficient and difficult to read for loops.

1.4 More on numpy - broadcasting

[14]:
```
A = np.array([[1,2,3],[4,5,6]])
B = np.array([[1],[2]])
C = np.array([[1,2,3]])
print("A:\n", A)
print("A shape:", A.shape)
print("B:\n", B)
print("B shape:", B.shape)
```

```
print("C:\n", C)
print("C shape:", C.shape)
```

```
A:
 [[1 2 3]
 [4 5 6]]
A shape: (2, 3)
B:
 [[1]
 [2]]
B shape: (2, 1)
C:
 [[1 2 3]]
C shape: (1, 3)
```

[119]:
```
A+B
```

[119]:
```
array([[2, 3, 4],
       [6, 7, 8]])
```

[120]:
```
A+C
```

[120]:
```
array([[2, 4, 6],
       [5, 7, 9]])
```

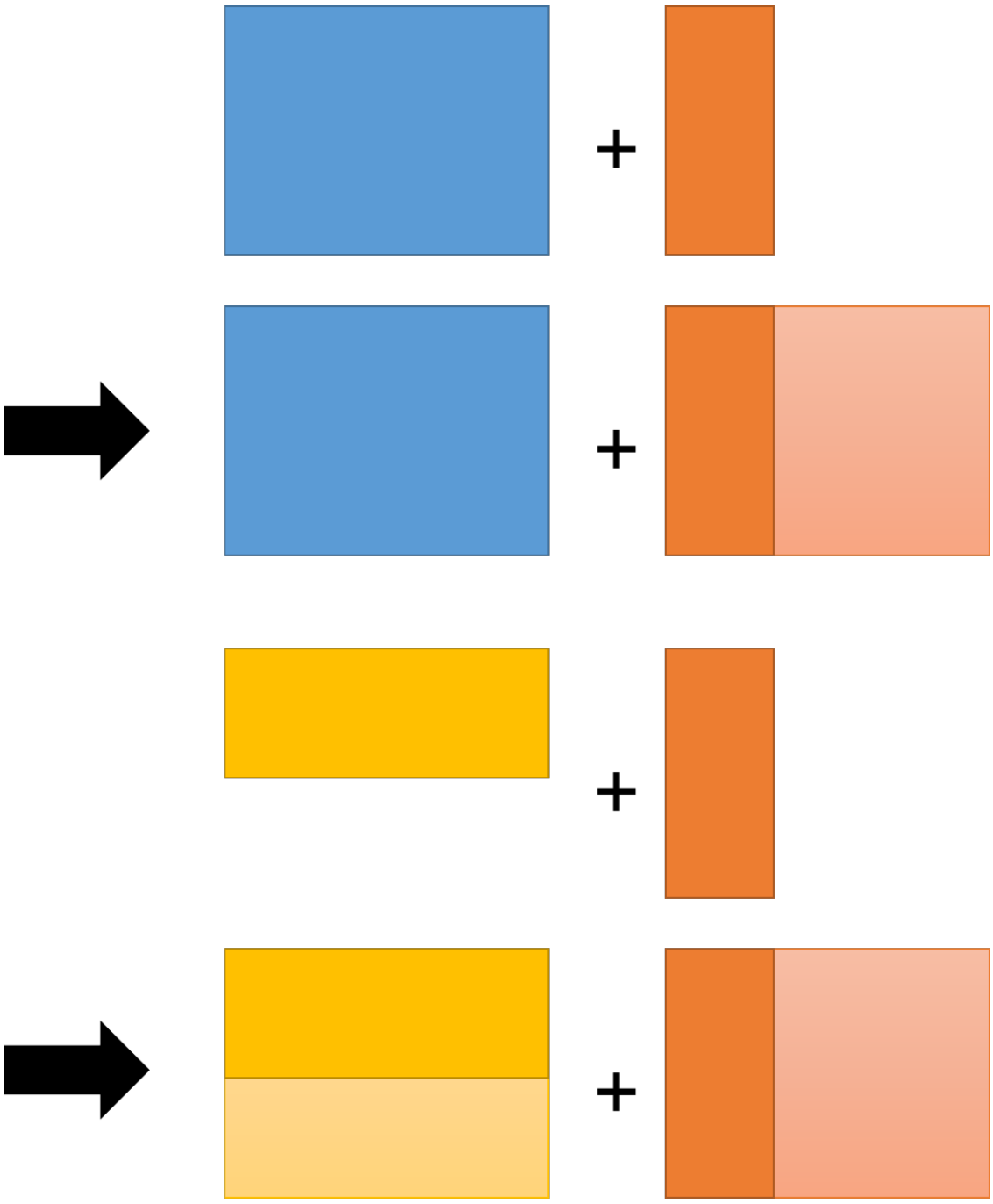
[121]:
```
B+C
```

[121]:
```
array([[2, 3, 4],
       [3, 4, 5]])
```

1.4.1 The idea of broadcasting:

A (2,3) + B(2,1) It would automately repeat B 3 times along axis 1, so A (2,3) + B_repeat (2,3)

B (2,1) + C(1,3) => B_repeat (2,3) + C_repeat (2,3)



1.4.2 Be careful about the dimension of array, especially 1D array

[128]:
```
G = np.array([[1,2,3]])
print("G shape:", G.shape)
print("transpose G shape:", G.T.shape)
G*G.T # T for transpose
```

```
G shape: (1, 3)
transpose G shape: (3, 1)
```

[128]:
```
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])
```

[129]:
```
G = np.array([1,2,3])
print("G shape:", G.shape)
print("transpose G shape:", G.T.shape)
G*G.T
```

```
G shape: (3,)
transpose G shape: (3,)
```

[129]:
```
array([1, 4, 9])
```

1.4.3 Note for broadcasting: The shapes need to be aligned by the "last" dimension

[17]:
```
D = np.array([1,2,3]) #shape:3
print("A shape:", A.shape)
print("D shape:", D.shape)
A+D
```

```
A shape: (2, 3)
D shape: (3,)
```

[17]:
```
array([[2, 4, 6],
       [5, 7, 9]])
```

[18]:
```
D = np.array([1,2]) #shape:2
print("A shape:", A.shape)
print("D shape:", D.shape)
A+D
```

```
A shape: (2, 3)
D shape: (2,)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_1890458/1986523861.py in <module>
      2 print("A shape:", A.shape)
      3 print("D shape:", D.shape)
----> 4 A+D

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

[19]:
```
D = np.array([[1,2,3]]) #shape:1x3
print("A shape:", A.shape)
print("D shape:", D.shape)
```

```
A+D
```

```
A shape: (2, 3)
D shape: (1, 3)
```

[19]:
```
array([[2, 4, 6],
       [5, 7, 9]])
```

[20]:
```
D = np.array([[1],[3]]) #shape:2x1
print("A shape:", A.shape)
print("D shape:", D.shape)
A+D
```

```
A shape: (2, 3)
D shape: (2, 1)
```

[20]:
```
array([[2, 3, 4],
       [7, 8, 9]])
```

1.5 Numba

1.5.1 What is Numba?

Quote: Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code. ### What is JIT? **From wiki:** In computing, just-in-time (JIT) compilation (also dynamic translation or run-time compilations) is a way of executing computer code that involves compilation during execution of a program (at run time) rather than before execution. This may consist of source code translation but is more commonly bytecode translation to machine code, which is then executed directly. A system implementing a JIT compiler typically continuously analyses the code being executed and identifies parts of the code where the speedup gained from compilation or recompilation would outweigh the overhead of compiling that code.

- Time for compilation while executing
- Optimize compilation during running

1.5.2 Comparation between numpy, numba.jit, loop

[134]:
```
a = np.random.randn(10000)
b = np.random.randn(10000)
```

Loop

[135]:
```
def product_sum_loop(a,b):
    out = 0
    for i in range(len(a)):
        out += a[i]*b[i]
    return out
def f_loop(x):
    return product_sum_loop(a,b)
```

[136]:
```
%%time
c = [f_loop(i) for i in range(1000)]
```

```
CPU times: user 1.69 s, sys: 0 ns, total: 1.69 s
Wall time: 1.69 s
```

Numba.jit

[137]:
```
@numba.jit
def product_sum(a,b):
    out = 0
    for i in range(len(a)):
        out += a[i]*b[i]
    return out
def f_jit(x):
    return product_sum(a,b)
```

[138]:
```
%%time
c = [f_jit(i) for i in range(1000)]
```

```
CPU times: user 103 ms, sys: 34.9 ms, total: 138 ms
Wall time: 205 ms
```

[139]:
```
%%time
c = [f_jit(i) for i in range(1000)]
```

```
CPU times: user 9.76 ms, sys: 0 ns, total: 9.76 ms
Wall time: 9.54 ms
```

One can see the time difference between the first time and the second time due to compiling

Numpy

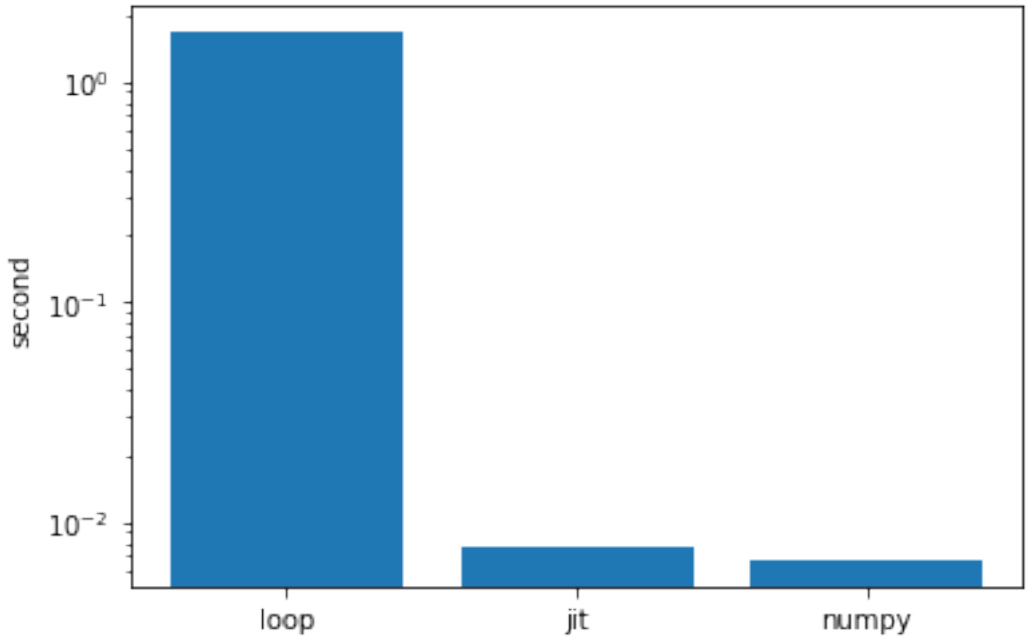
[140]:
```
def f_numpy(x):
    return (a*b).sum()
```

[141]:
```
%%time
c = [f_numpy(i) for i in range(1000)]
```

```
CPU times: user 11.2 ms, sys: 0 ns, total: 11.2 ms
Wall time: 9.68 ms
```

[144]:
```
t0 = time.time()
c = [f_loop(i) for i in range(1000)]
t1 = time.time()
c = [f_jit(i) for i in range(1000)]
t2 = time.time()
c = [f_numpy(i) for i in range(1000)]
```

```
t3 = time.time()
plt.bar(["loop", "jit", "numpy"], [t1-t0, t2-t1, t3-t2])
plt.ylabel("second")
plt.yscale('log')
```



1.6 More on Numba

- **vectorize**: you can vectorize a function so that it becomes the element-wise function for numpy array.
- **optimization**: jit compilation would optimize your code.

[145]:
```
from math import sin
@numba.vectorize()
def nbvc_sin2(x):
    return sin(x)*sin(x)
@numba.vectorize()
def nbvc_sin(x):
    return sin(x)
```

[146]:
```
%%time
for i in range(10000):
    nbvc_sin2(a)
```

```
CPU times: user 819 ms, sys: 0 ns, total: 819 ms
Wall time: 823 ms
```

second time

[147]:
```
%%time
for i in range(10000):
    nbvc_sin2(a)
```

```
CPU times: user 754 ms, sys: 0 ns, total: 754 ms
Wall time: 753 ms
```

[148]:
```
%%time
for i in range(10000):
    np.sin(a)*np.sin(a)
```

```
CPU times: user 1.65 s, sys: 0 ns, total: 1.65 s
Wall time: 1.65 s
```

The numpy calculation above is much slower since sin function is excuted twice. **Due to optimization, numba could be much faster than vectorizing operation of numpy(if you are not careful while coding)**

[149]:
```
%%time
for i in range(10000):
    np.sin(a)**2
```

```
CPU times: user 812 ms, sys: 2.06 ms, total: 814 ms
Wall time: 811 ms
```

1.7 multiprocessing

You can parallelize loop with **multiprocessing.pool** to do calculation with many cores.

[151]:
```
%%time
c = []
for i in range(1,20001):
    a = np.random.randn(i)
    b = np.random.randn(i)
    c.append((a*b).sum())
```

```
CPU times: user 6.4 s, sys: 3.37 ms, total: 6.4 s
Wall time: 6.4 s
```

[152]:
```
%%time
def f(x):
    a = np.random.randn(x)
    b = np.random.randn(x)
    return (a*b).sum()
with Pool(5) as p:
    c = p.map(f, list(range(20000)))
```

```
CPU times: user 22.6 ms, sys: 150 ms, total: 172 ms
Wall time: 1.83 s
```

stackoverflow: multiprocessing, pickle

1.7.1 Simple way - use jit with parallelization

[163]:
```
@numba.jit(parallel=True)
def nb_pl_product_sum():
    c = np.zeros(20001)
    for i in numba.prange(1, 20001):
        a = np.random.randn(i)
        b = np.random.randn(i)
        c[i-1] = (a*b).sum()
    return c
```

[164]:
```
%%time
c = nb_pl_product_sum()
```

```
CPU times: user 9.84 s, sys: 10.6 ms, total: 9.86 s
Wall time: 1.51 s
```

second time

[165]:
```
%%time
c = nb_pl_product_sum()
```

```
CPU times: user 9.46 s, sys: 0 ns, total: 9.46 s
Wall time: 1.08 s
```

1.8 Torch - Using the Power of GPU

- Nvidia RTX 3080: 8960 cores
- Intel Core i9-11900K : 8 cores

If your calculation can be parallelized, GPU might be faster for large-scale arithmetic calculation.

1.8.1 Pytorch

"PyTorch is an optimized tensor library for deep learning using GPUs and CPUs." - GPU supported - autograd - DL library

- other choices: jax, cupy

1.8.2 Comparison between numpy and torch

[6]:
```
a = np.random.randn(200000)
b = np.random.randn(200000)
```

[7]:
```
%%time
c = [(a*b).sum() for i in range(1000)]
```

```
CPU times: user 146 ms, sys: 0 ns, total: 146 ms
Wall time: 144 ms
```

[8]:
```
torch.set_default_tensor_type(torch.DoubleTensor)
a = torch.randn(200000,device="cuda")
b = torch.randn(200000,device="cuda")
def f_torch(x):
    return (a*b).sum()
```

[9]:
```
%%time
c = [f_torch(i) for i in range(1000)]
```

```
CPU times: user 51.4 ms, sys: 52.1 ms, total: 104 ms
Wall time: 102 ms
```

1.8.3 Note:

In the case above, the speed-up is not significant since `sum` can not be easily parallelized. Consider the case below, which is just the element-wise multiplication. We can find the speed-up is significant.

[12]:
```
a = np.random.randn(200000)
b = np.random.randn(200000)
def f2_numpy(x):
    c = (a*b)
    return
```

[13]:
```
%%time
c = [f2_numpy(i) for i in range(1000)]
```

```
CPU times: user 69.1 ms, sys: 0 ns, total: 69.1 ms
Wall time: 67.2 ms
```

[10]:
```
torch.set_default_tensor_type(torch.DoubleTensor)
a = torch.randn(200000,device="cuda")
```

```
b = torch.randn(200000,device="cuda")

def f2_torch(x):
    c = (a*b)
    return
```

[11]:
```
%%time
c = [f2_torch(i) for i in range(1000)]
```

```
CPU times: user 7.07 ms, sys: 743 µs, total: 7.81 ms
Wall time: 6.57 ms
```

1.9 More topic - lru_cache

With the decorator `lru_cache`, the return values of certain arguments would be recorded so the calculation won't be executed many times.

[112]:
```
from functools import lru_cache
@lru_cache
def fibonacci_cache(n):
    return fibonacci_cache(n-1) + fibonacci_cache(n-2) if n>=2 else 1
```

[113]:
```
%%time
fibonacci_cache(35)
```

```
CPU times: user 39 µs, sys: 3 µs, total: 42 µs
Wall time: 49.1 µs
```

[113]:
```
14930352
```

[107]:
```
def fibonacci(n):
    return fibonacci(n-1) + fibonacci(n-2) if n>=2 else 1
```

[111]:
```
%%time
fibonacci(35)
```

```
CPU times: user 1.69 s, sys: 2.58 ms, total: 1.7 s
Wall time: 1.7 s
```

[111]:
```
14930352
```