02_numpy_closest

May 31, 2017

1 Interet du calcul vectoriel

Le but de cet exemple d'illustrer l'efficacité des tableaux numpy par rapport aux listes python.

A partir d'un nuage de points générés aléatoirement dans [0, 1[, on cherche à identifier celui qui se trouve le plus proche d'un point M0(x0, y0). On crée une fonction closest où les arguments sont un tuple (position de M0) et une liste (ou tableau) des coordonnées des points.

```
In [1]: def closest(pos0, points):
            x0, y0 = pos0
            dbest, ibest = None, None # avantage de python initialiser
                                      # avec une valeur indéfinie
            for i, (x, y) in enumerate(points):
                # carre de la distance au point M0
                d = (x - x0) ** 2 + (y - y0) ** 2
                if dbest is None or d < dbest:
                    dbest, ibest = d, i
            return ibest
```

1.1 Utilisation des listes python

Comment générer une liste de coordonnées (x,y) en python, puis appeler la fonction closest

```
In [2]: import random
        N=100000
        random.seed(123)
        positions = [(random.random(), random.random()) for _ in range(N)]
        # utilisation d'une variable muette

        pos0=(0.5, 0.5)
        %time i=closest(pos0, positions)
        %timeit i=closest(pos0, positions)
        # donne le meilleur temps = seul qui est reproductible
```

```
CPU times: user 51.6 ms, sys: 1.49 ms, total: 53.1 ms
Wall time: 52.6 ms
10 loops, best of 3: 37.9 ms per loop
```

Utilisation du compilateur jit du module numba

```
In [3]: import random
        from numba import jit

        @jit
        def closest(pos0, points):
               x0, y0 = pos0
               dbest, ibest = None, None
               for i, (x, y) in enumerate(points):
                   # carre de la distance au point M0
                   d = (x - x0) ** 2 + (y - y0) ** 2
                   if dbest is None or d < dbest:
                       dbest, ibest = d, i
               return ibest

        N=100000
        random.seed(123)
        positions = [(random.random(), random.random()) for _ in range(N)]
        #print(positions)
        pos0=(0.5, 0.5)
        %time i=closest(pos0, positions)
        %timeit i=closest(pos0, positions)
```

```
CPU times: user 249 ms, sys: 17.7 ms, total: 267 ms
Wall time: 291 ms
100 loops, best of 3: 3.89 ms per loop
```

Le premier temps explose car il intègre la phase de compilation Le second utilise la version compilée de la fonction, d'où le gain en performance

1.2 Utilisation d'un tableau numpy

Comment générer un tableau numpy 2D , puis appeler la fonction closest

```
In [4]: import numpy as np

        def numpy_closest(pos0, points):
            x0, y0 = pos0
            x,  y  = points[:,0], points[:,1]
            d = (x - x0) ** 2 + (y - y0) ** 2
            return d.argmin()

        N=100000
        np.random.seed(123)
        positions = np.random.rand(N, 2)
        #print(positions)
        #print(type(positions))
```

```
        pos0=(0.5, 0.5)
        %timeit i=closest(pos0, positions)
        %timeit i=numpy_closest(pos0, positions)
```

```
1 loop, best of 3: 168 ms per loop
1000 loops, best of 3: 470 µs per loop
```

Le gain de performance est du à la suppression de la boucle explicite
Utilisation du compilateur jit du module numba

```
In [5]: import random
        import numpy as np
        from numba import jit

        @jit
        def numpy_closest(pos0, points):
            x0, y0 = pos0
            x,  y  = points[:,0], points[:,1]
            d = (x - x0) ** 2 + (y - y0) ** 2
            return d.argmin()

        N=100000
        np.random.seed(123)
        positions = np.random.rand(N, 2)
        #print(positions)
        #print(type(positions))
        pos0=(0.5, 0.5)
        %timeit i=closest(pos0, positions)
        %timeit i=numpy_closest(pos0, positions)
```

```
10 loops, best of 3: 170 ms per loop
The slowest run took 809.09 times longer than the fastest. This could mean that an
1000 loops, best of 3: 225 µs per loop
```

1.3 Utilisation d'un arbre

A titre informel, la libraire scipy contient une recherche de plus proches voisins optimisée à base d'arbre binaire

```
In [6]: import scipy.spatial as spatial
        import random

        N=100000
        np.random.seed(123)
        positions = np.random.rand(N, 2)

        kdtree = spatial.KDTree(positions)
```

```
        pos0=(0.5, 0.5)
        %timeit d, i = kdtree.query(pos0, 1)
        d, i = kdtree.query(pos0, 1)
        print(i)
        print(positions[i, :])
```

```
1000 loops, best of 3: 325 µs per loop
61502
[ 0.49981353  0.49615511]
```