Vectorisation des calculs

Laurent Signac – CC-BY-SA – 11-02-22 0908 b2a3f278683661271ad7

Nous ne faisons ici qu'éffleurer le sujet. En particulier les questions du matériel et de la parallélisation effective des calculs n'est pas abordée.

Plutôt que d'appliquer une fonction à des scalaires, plusieurs fois au sein d'une boucle (par exemple, mettre tous les éléments d'une liste au carré en parcourant la liste), la vectorisation consiste à appliquer une seule fois la fonction, à un vecteur plutôt qu'à un scalaire (ici le vecteur serait formé des valeurs qu'on veut mettre au carré).

Les implémentations vectorielles sont plus rapides pour de multiples raisons :

- les boucles sous-jacentes ont été particulièrement bien programmées, et sont souvent compilées, alors même qu'on utilise un langage interprété
- les opérations à faire sur la collection d'éléments peuvent souvent être parallélisées, et particulier en utilisant des processeurs graphiques, particulièrement en calculs matriciels et vectoriels.
- …

La plupart du temps, ce qu'on recherche avec la vectorisation, c'est un gain de performance. Si le langage ou la bibliothèque qu'on utilise est adapté, la vectorisation peut aussi simplifier l'écriture du code. On est alors gagnant sur tous les tableaux.

1 `numpy`

Le module `numpy`, qui ne fait pas partie de la bibliothèque standard, est dédié au calcul matriciel. Son utilisation est *similaire* à ce qu'on fait avec Matlab, sans être identique. Ce module est *très largement* utilisé par la communauté scientifique, et il sert de base à de nombreux autres modules très largement utilisés.

Les tableaux numpy sont typés : contrairement aux listes Python, tous les éléments doivent être du même type. Les contraintes sur les données numériques sont celles qu'on rencontre couramment en langage C : entiers, signés ou non, sur combien de chiffre binaires etc.

2 Exemples

Voici un premier exemple jouet : **Parmi les 10000 premiers carrés, lesquels ont un chiffre des dizaines qui vaut 6 ?**

2.1 Avec Python

Voici le code python standard, non vectorisé (50 ms):

```
res = []
for i in range(100000):
    i2 = i ** 2
    di = (i2 // 10) % 10
    if di == 6:
        res.append(i)
print(res)
```

Et maintenant la version vectorisée (10 ms):

```
import numpy as np

res = np.array(range(100000), dtype=np.uint32)
res2 = (res ** 2) // 10 % 10
res3 = res[res2 == 6]
print(res3)
```

La version vectorisée tire profit du fait que `numpy` peut faire des opérations vectorielles, comme `(res ** 2) // 10 % 10` avec `res` qui est un vecteur contenant des nombres entiers (non signés, sur 32 bits).

2.2 Avec Matlab

Avec Matlab, c'est presque la même chose. Voici la version scalaire exécutée en 30 ms :

```
m = [1:100000];
res = false(1, length(m));
tic
for i=1:length(m)
    res(i) = ( mod(fix(m(i)^2/10), 10 ) == 6 );
end
toc
```

Et voici la version vectorisée, exécutée en 10 ms:

```
m = [1:100000];
tic
res = ( mod(fix(m.^2/10), 10 ) == 6 );
toc
```

3 Utilisation de numpy

3.1 Construction des matrices / vecteurs

`numpy` différencie les matrices lignes, des matrices colonnes, des vecteurs (contrairement à Matlab) : un vecteur s'affiche avec 1 seule paire de crochets, une matrice en a deux. Bien comprendre la distinction *vecteur*, *matrice* vous évitera des ennuis, à plus forte raison si vous êtes habitués à Matlab.

On peut construire des matrices initialisées à 0 ou à 1 :

```
>>> np.zeros((3, 2))
array([[0., 0.],      # <=== [[ donc c'est une matrice
       [0., 0.],
       [0., 0.]])

>>> np.ones(5)
array([1., 1., 1., 1., 1.])  # <=== [ donc c'est un vecteur
```

Il est aussi possible de construire des matrice ou des vecteurs :

- à partir d'itérables : `np.array([1, 4, 5])`
- aléatoires :
  - `np.random.random((2, 4))`
  - `np.random.randint(low=-10, high=10, size=(3,3))`
  - `np.random.choice((54, 42), size=(3, 3))`
- régulières :
  - `np.linspace(start=1, stop=2, num=5 )`
  - `np.arange(2, 3, 0.1)`

3.2 Changement de forme

On est assez souvent amenés à modifier la forme d'une matrice. Par exemple une matrice de 2 lignes et 3 colonnes contient 6 valeurs, qui pourraient aussi être disposés dans une matrice de 3 lignes et 2 colonnes, ou dans un vecteur de taille 6. Il est utile de connaître les fonction qui réalisent ces transformations :

- `m.reshape((2, 3))` transforme `m` en matrice 2×3
- `m.reshape(6)` transforme `m` en vecteur de taille 6
- `m.ravel()` ou `m.flatten()` *aplatissent* la matrice `m`, c'est à dire la transforment en vecteur.

Attention, `reshape` et `ravel` renvoient de **vues** et non des copies (voir la partie sur les écueils pour la notion de *vue*, ici fondamentale). Par contre, `flatten` renvoie bien une *copie* des données.

On obtient la taille d'une matrice ainsi :

- `m.shape` (c'est un tuple de une valeur pour un vecteur ou de deux valeurs pour une matrice).

3.3 Opérations matricielles et vectorielles

Par défaut, les opérations sont des opérations membre à membre (comme si on utilisait `.*` au lieu de `*` avec Matlab) :

- produit membre à membre : `a * b`
- somme : `a + b`
- carré membre à membre : `a ** 2`
- certaines fonctions habituelles (du module de math) sont en version vectorisée dans numpy : `np.sin(a)`

Les opérations matricielle sont donc accessibles autrement :

- produit matriciel : `np.dot(a, b)` ou `a @ b`
- transposition (*uniquement pour les matrices*) : `a.transpose()` ou `a.T`

3.4 Accès aux valeurs des matrices / vecteur

3.4.1 Par lignes/colonnes

On écrit `m[lignes, colonnes]` où `lignes` et `colonnes` peuvent être des entiers, des itérables d'entiers, des slices… Comme pour les séquences Python, le premier indice est 0, contrairement à Matlab.

Exemple :

```
>>> m = np.random.randint(low=0, high=10, size=(4, 4))

   [[7, 6, 9, 8],
    [3, 7, 1, 5],
    [9, 0, 8, 1],
    [7, 9, 6, 3]]

>>> m[(1, 3), 1:] # lignes 1 et 3, colonnes de 1 jusqu'à la fin

   [[7, 1, 5],
    [9, 6, 3]]
```

3.4.2 Utilisation des masques

Les masques sont une fonctionnalité très puissante, qui permettent d'adresser rapidement et facilement certaines cases (lorsqu'on ne peut pas les décrire simplement en termes de lignes ou colonnes par exemple).

Dans l'exemple qui suit, on crée une matrice `m` de taille (4, 4) contenant des entiers, puis un masque (un masque est une matrice de booléens), de taille (4, 4) aussi.

On peut ensuite appliquer des opérations sur `m` en précisant que ces opérations ne sont à appliquer que pour les cases pour lesquelles la valeur du masque est à `True`. Ces opérations peuvent être un simple accès comme dans l'exemple qui suit, ou tout type de calcul.

```
>>> m = np.random.randint(low=0, high=10, size=(4, 4))

   [[7, 6, 9, 8],
    [3, 7, 1, 5],
    [9, 0, 8, 1],
    [7, 9, 6, 3]]

>>> mask = np.random.choice((True, False), size=(4, 4))

   [[False, False, False,  True],
    [False, False,  True,  True],
    [False, False, False, False],
    [False, False,  True,  True]]

>>> m[mask]

    [8, 1, 5, 6, 3]  # On récupère un vecteur
```

Puisqu'un masque est une matrice de booléens, la plupart des opérations de comparaison renvoient un masque. Par exemple, si `m` est une matrice (4, 4), alors `m % 2 == 0` est un masque qui vaut `True` aux emplacements des cases paires de `m`. On utilise ceci dans l'exemple suivant pour ajouter 1 uniquement sur les cases paires. Il est fondamental de bien comprendre cet exemple :

```
>>> m = np.random.randint(low=0, high=10, size=(4, 4))

   [[7, 6, 9, 8],
    [3, 7, 1, 5],
    [9, 0, 8, 1],
    [7, 9, 6, 3]]

>>> m[m % 2 == 0]

   [6, 8, 0, 8, 6]

>>> m[m % 2 == 0] += 1
>>> m

   [[7, 7, 9, 9],
    [3, 7, 1, 5],
    [9, 1, 9, 1],
    [7, 9, 7, 3]]
```

Les masques étant des matrices contenant des booléens, on dispose des opérations booléennes habituelles sur des matrices de booléens (opérations membre à membre) :

- `np.logical_or(a, b)` ou `a | b`
- `np.logical_and(a, b)` ou `a & b`
- `np.logical_xor(a, b)` ou `a ^ b`
- `np.logical_not(a)` ou `~a`

Le masque suivant, appliqué à une matrice d'entiers `m` permet de cibler les cases paires contenant des nombres à au moins 2 chiffres : `(m % 2 == 0) & (m >= 10)`

3.5 Quelques écueils

L'utilisation de `numpy` n'est pas toujours facile. On indique ici quelques erreurs courantes.

3.5.1 Vues

numpy propose la notion de *vue*, qui permet de *regarder* un tableau sous un certain angle (géométrique, type de données), sans toutefois en faire une copie. (c'est donc très rapide, puisqu'on ne fait pas de copie, mais modifier des éléments dans une vue modifie l'objet original).

Faire la distinction entre une **copie** et une **vue** n'est pas évident et peut conduire à des bugs complexes à trouver. Heureusement, on dispose d'un moyen sûr de vérifier ce que font les fonctions (renvoient-elles une copie ou une vue ?), en utilisant l'attribut `base` des objets. Voici un exemple :

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = a.ravel()   # Vue ou copie ?
>>> c = a.flatten() # Vue ou copie ?

>>> b
   [1, 2, 3, 4, 5, 6]
>>> c
   [1, 2, 3, 4, 5, 6]

>>> b[0] = 42
>>> c[5] = 54

>>> a
    [[42,  2,  3], # Ah... ravel remble renvoyer une vue
     [ 4,  5,  6]]

>>> b.base is a    # Effectivement, b est une vue sur a
  True

>>> c.base is a    # Mais c n'est pas une vue sur a
  False

>>> a.base is None # a n'est pas une vue...
  True
```

3.5.2 Tests

Comme nous l'avons vu, les comparaisons `m > 2` ou `m == 2` sont des masques (matrices de booléens). On ne peut donc pas écrire `if m == 3` puisqu'après `if` il faut un booléen (et quel sens donnerait-on à l'instruction ?).

On peut transformer une matrice de booléens en booléen (pour usage avec `if` ou `while` par exemple) avec `np.all` et `np.any`, ce qui donne un sens ou l'autre à la transformation (toutes les valeurs de la matrice sont-elles à `True` ou existe-t-il au moins une valeur de la matrice à `True`) :

```
>>> m = np.array([1, 2, 3, 4, 5])
>>> r = (m >= 3)
[False, False, True, True, True]
>>> np.all(r)
False
>>> np.any(r)
True
>>> np.all(m > 0)
True
```

3.5.3 Priorité des opérateurs

Les opérateurs `&` et `|` ont une précédence élevée, et ils sont prioritaires sur les opérateurs de comparaison. Or on les utilise ici *un peu comme* des **ou** et des **et**, qui eux ont une précédence plus faible que les opérateurs de comparaison. Il est donc facile de se tromper et d'oublier les parenthèses, qui sont nécessaires avec `|` et `&`.

```
>>> m5 = np.random.randint(-10,10, (3,3))
>>> print(m5)

    [[ -5   2 -10]
     [ -7   7   9]
     [ -9  -7   0]]

>>> print((m5 < 0) | (m5 > 5)) # les parenthèses sont requises

    [[ True False  True]
     [ True  True  True]
     [ True  True False]]
```