1 Introduction

Compilation à la volée (JIT) de fonctions avec le package Numba pour Python.

Python est un langage de programmation interprété. On qualifie habituellement son exécution de lente par rapport aux programmes compilés (écrits en C par exemple). A cette critique, on peut facilement objecter qu'il est certes interprété, mais que les instructions sont en grande partie composées d'appels à des primitives de calculs qui, elles, sont compilées. C'est le cas par exemple des fonctions de la librairie Numpy pour la manipulation des vecteurs et des matrices.

Mais que faire quand nous souhaitons accélérer l'exécution d'un programme composé essentiellement d'instructions natives Python ? Dans ce tutoriel, nous nous intéressons à la compilation à la volée (JIT, ''just-in-time compilation'') des fonctions écrites en Python pour les rendre plus efficaces. Avec un certain amusement, je constate que je m'étais déjà posé ce genre de questions pour R en son temps (''La compilation sous R'', juillet 2012). Maintenant, depuis la version 3.4 de R en particulier, le JIT est directement activé pour certaines parties du code R.

Nous étudions le package Numba pour Python. Il permet de rendre plus performantes des portions de nos programmes (des fonctions essentiellement) en introduisant simplement des ''directives de compilation'', sans autres modifications du code. Nous verrons que l'outil est diablement intéressant. Il l'est d'autant plus que nous pouvons profiter de la parallélisation des calculs dans certaines circonstances. Pour information, je travaille sous Windows 10 Education 64 bits avec la version Python 3.7.3 de la distribution Anaconda.

2 Tri par insertion

2.1 Implémentation usuelle

Pour évaluer l'apport de la compilation, nous implémentons – de manière très basique – l'algorithme de tri par insertion d'un vecteur de valeurs.

```
#algorithme de tri par insertion
#https://fr.wikipedia.org/wiki/Tri_par_insertion
def tri_insertion(tab):
    #faire une copie du vecteur
    vec = numpy.copy(tab)
    #parcourir le vecteur
    for i in range(1,len(vec)):
        #mémorisation
        x = vec[i]
        #décalage
        j = i
        while (j > 0) and (vec[j-1] > x):
```

```
            vec[j] = vec[j-1]
            j = j - 1
        #insertion
        vec[j] = x
    #retourner le vecteur trié
    return vec
```

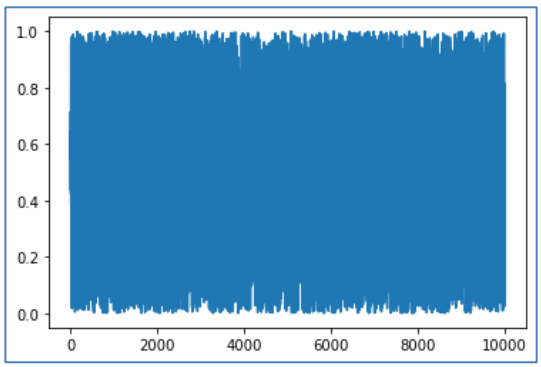
La fonction prend en entrée un vecteur (tab, que l'on considère être un vecteur numpy), effectue une copie interne (vec), trie ce dernier puis le renvoie. Le vecteur initial n'est donc pas modifié.

Pour tester notre fonction, nous générons un vecteur de (`n = 10.000`) valeurs aléatoires, en fixant la valeur initial (seed) du générateur pour que l'expérimentation soit reproductible.

```
#nombre de valeurs à trier
n = 10000
#générer un vecteur de valeurs aléatoires
import numpy
numpy.random.seed(0)
alea = numpy.random.random(size=n)

#librairie graphique
import matplotlib.pyplot as plt
plt.plot(alea)
```

Le graphique sert à vérifier le caractère aléatoire des valeurs. C'est le cas.



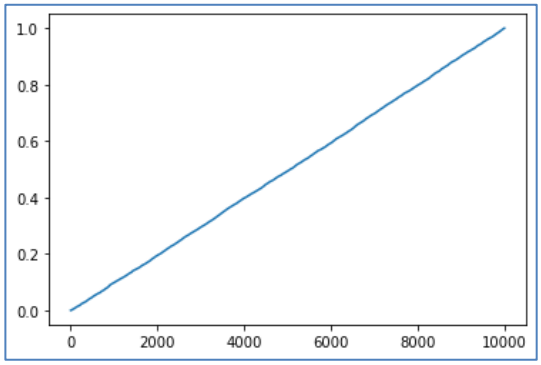
Nous appelons alors notre fonction et mesurons la durée d'exécution en utilisant la fonction perf_counter() de la librairie time.

```
#appel de la fonction de tri
import time
start = time.perf_counter()
alea_tri = tri_insertion(alea)
end = time.perf_counter()
print("Duree = ", end-start," secondes")
```

```
Durée = 11.137963868999577 secondes
```

Le système annonce une durée d'exécution de 11.13 secondes. Nous vérifions que le vecteur a bien été trié correctement.

```
#vérification
plt.plot(alea_tri)
```



Une mesure de la durée d'exécution peut être perturbée par toute une série d'évènements en cours sur l'ordinateur. Il est plus sage de répéter plusieurs fois l'expérimentation pour disposer d'une évaluation plus fiable. Nous utilisons la fonction magique %timeit que nous appelons directement de la console IPython (je travaille dans Spyder). Soit :

```
%timeit tri_insertion(alea)
```

```
11.2 s ± 173 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

La fonction a été lancée 7 fois, le temps moyen d'exécution est de 11.2 secondes. La durée constatée initialement est confirmée. Il faut dire que j'avais veillé au grain en fermant les applications qui auraient pu interférer avec mes calculs sur mon système.

2.2 Compilation avec Numba

Pour compiler la fonction avec Numba, nous la réécrivons exactement tel quel mais en rajoutant une directive de compilation *@jit* devant son en-tête (l'option nopython = True permet d'obtenir l'encodage le plus efficace dixit la documentation), en veillant bien sûr à importer la librairie (la fonction concernée de la librairie) au préalable.

```
#JIT pour fonction
from numba import jit

@jit(nopython=True)
def tri_insertion_jit(tab):
    #faire une copie
    vec = numpy.copy(tab)
    #parcourir le vecteur
    for i in range(1,len(vec)):
        #mémorisation
```

```
        x = vec[i]
        #décalage
        j = i
        while (j > 0) and (vec[j-1] > x):
            vec[j] = vec[j-1]
            j = j - 1
        #insertion
        vec[j] = x
    #retourner le vecteur trié
    return vec

#mesurer de nouveau
start = time.perf_counter()
alea_tri_jit = tri_insertion_jit(alea)
end = time.perf_counter()
print("Duree = ", end-start," secondes")
```

```
Durée = 0.24044019899999824 secondes
```

La durée d'exécution est sans commune mesure ! Le gain est particulièrement spectaculaire pour une seule directive rajoutée dans notre code. On aurait tort de s'en priver.

3 Régression linéaire par descente du gradient

Un tri par insertion est certes intéressant mais n'est pas vraiment représentatif des calculs effectués en machine learning. Dans cette section, nous nous essayons à la programmation de l'estimation des coefficients d'une régression simple via une descente du gradient. On en trouve de nombreuses descriptions et implémentations ici et là sur le net. Dans notre cas, nous donnons la part belle au calcul matriciel dans le but de mettre en évidence la capacité à paralléliser de Numba.

3.1 Régression avec descente du gradient

Le modèle à estimer s'écrit :

$$y = 2 * x + 5$$

En notation matricielle, nous avons :

$$y = X * coef$$

Ou X est une matrice (n_obs, 2) avec dans la première colonnes les valeurs de x, dans la seconde une constante 1. Coef est un vecteur avec les coefficients à estimer (2, 5).

Pour effectuer l'apprentissage, nous avons généré un vecteur **x** de 2.000.000 d'observations. Nous en avons déduit **y** avec y = 2 * x + 5 + ε, où ε est un bruit blanc gaussien $\mathcal{N}(0, 0.05)$.

Nous fixons comme valeur de départ de 'coef' : $coef_0$ = (0.5, 0.5)

Les paramètres de l'algorithme de descente du gradient sont : taux d'apprentissage α = 0.01, nombre d'itérations (epochs) = 1000.

Voici la fonction correspondant à cette description :

```
#regression simple avec descente du gradient
def gradient_regression():
    #paramètres alpha et epochs pour algo du gradient
    alpha = 0.01
    iterations = 1000
    #génération des données
    n_obs = 2000000
    numpy.random.seed(0)
    #matrice X
    X = numpy.full(shape=(n_obs,2),fill_value = 1.0)
    X[:,0] = numpy.random.random(size=n_obs)
    #vecteur y = 2 * x + 5 + bruit aléatoire
    y = 2.0 * X[:,0] + 5.0 + numpy.random.normal(loc=0,scale=0.05,size=n_obs)
    #vecteur des coefs à estimer - valeurs de départ
    coef = numpy.array([0.5,0.5])
    #vecteur des scr (somme des carrés des résidus)
    scr = numpy.zeros(iterations)
    #commencer les itérations
    for i in range(iterations):
        #prédictions sur la base des coef. estimés
        yPred = numpy.dot(X,coef)
        #erreur de prédiction
        err = y-yPred
        #somme des carrés des résidus
        scr[i] = numpy.sum(numpy.square(err))
        #vecteur gradient
        gradient = 2.0 / n_obs * numpy.dot(numpy.transpose(X),err)
        #calcul de la nouvelle version des coefficients estimés
        coef = coef + alpha * numpy.transpose(gradient)
        #fin des itérations
    return scr,coef
```

Elle renvoie la somme des carrés des résidus (SCR) au fil des itérations et le vecteur des coefficients estimés à l'étape finale.

3.2 Version non-compilée

Dans un premier temps, nous lançons la version non-compilée de la fonction. Pour obtenir une mesure stabilisée de la durée d'exécution, nous utilisons directement dans la console **%timeit**

```
%timeit gradient_regression()
```

```
53.2 s ± 122 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Le temps moyen d'exécution est 53.2 secondes. Pour vérification, le vecteur des coefficients estimés est (2.36, 4.80). Le bruit introduit a fait dévier, un peu, la droite de régression.

3.3 Version compilée

Pour la version compilée, nous ajoutons la directive de compilation, soit :

```
@jit(nopython=True)
def gradient_regression():
    #paramètre alpha algo gradient
    alpha = 0.005
    iterations = 1000
    #génération des données
    n_obs = 2000000
    etc...
```

Puis nous faisons appel à **%timeit** :

```
%timeit gradient_regression()
```

```
47.6 s ± 101 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

On est passé à 47.6 secondes. Le gain est décevant, c'est le moins qu'on puisse dire. Mais il se comprend. Une grande partie des calculs consiste à faire appel aux fonctions de numpy avec du calcul matriciel. La compilation JIT n'est pas d'un grand secours dans cette situation.

3.4 Version compilée et parallélisée

Refaire un tour dans la documentation nous est d'un grand secours à ce stade. En effet, j'avais noté que Numba était capable de paralléliser **automatiquement** certains calculs pourvu qu'on lui en donne l'occasion. C'est le cas par exemple des opérations avec les structures numpy, qui sont justement très présentes dans notre fonction d'estimation des coefficients de la régression (http://numba.pydata.org/numba-doc/latest/user/parallel.html ; Section 1.10).

Fort de cette information, nous introduisons l'option (parallel = True) dans la directive de compilation, soit :

```
@jit(nopython=True, parallel=True)
def gradient_regression():
    #paramètre alpha algo gradient
    alpha = 0.005
    iterations = 1000
    #génération des données
    n_obs = 2000000
    etc...
```

```
11.1 s ± 65.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Le gain est bluffant. Nous sommes passés à une durée moyenne d'exécution de 11.1 secondes. Encore une fois, sans avoir à modifier une seule ligne de code de notre fonction.

Je m'arrête ici en ce qui me concerne. Mais la lecture de la documentation est très intéressante. Elle montre les différentes opportunités que nous pouvons saisir pour améliorer les temps d'exécution. On nous dit notamment que travailler avec un profiler est une excellente approche pour identifier les goulots d'étranglement de nos programmes. Je suis évidemment tout à fait d'accord avec ce conseil (''Programmer efficacement sous Python'', avril 2019).

4 Conclusion

A terme, tout comme sous R, je pense que la compilation à la volée fera partie intégrante de Python. Le gain que l'on peut en obtenir est important dans certaines circonstances, sans que l'on ait à fournir un effort de conception supplémentaire dans l'écriture de notre programme. Le développeur est gagnant sur toute la ligne (de code).

Plus impressionnant je trouve est la possibilité de paralléliser automatiquement les traitements, toujours via une simple directive de compilation. Bien sûr, il faut se placer dans un contexte favorable pour en profiter pleinement, notamment en effectuant des choix de structure judicieux. Pour l'algorithme de tri par exemple, l'option de parallélisation n'a pas été opérant (Numba a même envoyé un ''warning'' indiquant qu'il n'y avait rien à paralléliser dans ce cas). Mais bon, je crois que pour un moment encore, l'efficacité des outils ne nous dispense pas de réfléchir en amont et de concevoir au mieux notre code. Il ne faudrait pas l'oublier….

Enfin, je ne l'ai pas testé dans ce tutoriel parce que je ne dispose pas de la configuration adéquate, Numba permet d'exploiter les capacités des GPU (processeurs graphiques, graphics processing unit) sur les machines disposant de cartes graphiques performantes. A évaluer dès que je trouve une machine qui s'y prête.

5 Références

Numba, http://numba.pydata.org/