JAX: un Numpy augmenté

petite introduction

J.E Campagne

12 septembre 2021

1 Introduction

`JAX` est une librairie [1] `Python` développée par Google et utilisée par l'équipe DeepMind dans le contexte de développements d'outils de Machine Learning (ML). Plus précisément, c'est une librairie de calcul numérique [2] que l'on peut considérer comme un `Numpy` augmenté de la **différentiation automatique** [3] à la manière de `Pytorch` ou `TensorFlow`, de la possibilité de faire une **compilation à la volée** dite "Just-in-Time" (`JIT`) pour utiliser l'accélérateur `XLA` de `TensorFlow` pour l'Algèbre Linéaire qui tourne sur les devices CPU et GPU [4], ainsi que de la **vectorisation** [5] des fonctions. La différentiation automatique est très active dans le domaine ML [6] puisque c'est avec cet outil que l'on peut entrainer des réseaux de neurones profonds, mais aussi en statistique à travers les méthodes entre autres de Monte Carlo Hamiltonien dont je parlerai dans une future note (nb. c'est dans ce cadre qu'a été développée la librairie [7] `jax-cosmo`). Cependant, même dans un contexte où la différentiation automatique n'est pas utile, utiliser `JAX` en lieu et place de `Numpy` peut être utile pour bénéficier d'un surplus de performances dans certaines situations.

Dans cette note, à travers quelques exemples, je donne un petit aperçu de comment utiliser `JAX` et pour la majeur partie des exemples un simple CPU suffit. Un notebook sert de how-to/support pour cette note. Le repository est à l'adresse https://github.com/jecampagne/jax-demo. Vous pouvez cloner ce repository voir en faire un fork. Pour tourner avec un GPU, on peut le faire soit au CCIn2p3 (possible en interactif [8] et en batch) soit

1. https://github.com/google/jax
2. Sur CPU la librairie LAPACK/BLAS est utilisée, sur GPU ce sont les versions CUDA comme https://docs.nvidia.com/cuda/cusolver/index.htm.
3. A différentier du *calcul symbolique* ou *formel* comme `Mathematica` peut le faire, et de la *différentiation numérique* ou *méthode d'éléments finis*. Ces deux méthodes ont des problèmes, transposition en code numérique pour la première, accumulation des erreurs pour la seconde aux ordres supérieurs que tend à résoudre la différentiation automatique qui est un sujet de mathématique en soit avec son pionnier Robert Edwin Wengert qui écrit *A simple automatic derivative evaluation program* en 1964. Il s'agit du mode "forward" dont la première implémentation est rapidement faite par la RAND corporation. Le "reverse mode" date de 1980 établit par B. Speelpenning.
4. voire TPU.
5. Je ne parle pas dans cette note de parallélisation.
6. J'en veux pour preuve les workshops dédiés à cette thématique, ex. http://www.autodiff.org/.
7. https://github.com/DifferentiableUniverseInitiative/jax_cosmo
8. Une connexion via un notebook est en cours de développement au moment où j'écris ces lignes.

sur la plateforme collaborative de Google [9].

2 Premiers pas

Dans un premier temps, si ce n'est pas déjà fait pour vous par le système, il faut s'installer une version [10] de la librairie via

```
pip install --upgrade jax jaxlib
```

Dans ce contexte seule la version CPU s'exécutera par la suite. Pour obtenir la version GPU, il est préférable de se renseigner sur la version de CUDA en vigueur, puis faire par exemple au CCIn2p3:

```
pip install --upgrade jax jaxlib==0.1.71+cuda102 -f
    https://storage.googleapis.com/jax-releases/jax_releases.html
```

A présent, importons quelques librairies

```
import jax
import jax.numpy as jnp
import numpy as np
import matplotlib.pyplot as plt
```

Notez qu'outre la partie `JAX`, j'ai importé également les 2 librairies traditionnelles d'un utilisateur lambda. L'extension `Numpy` de `JAX` n'est pas un simple "enrobage" mais les mêmes APIs permettent d'utiliser aussi facilement la version `JAX` pour la quasi-totalité des fonctions. Créons deux vecteurs par exemple

```
x = np.linspace(0,1,10)
y = jnp.linspace(0,2,10)
```

A part le "j", rien ne différencie les deux lignes qui définissent les deux vecteurs x et y. Ce qui change donc c'est leurs représentations internes:

```
x : array([0., 0.11111111, 0.22222222, 0.33333333, 0.44444444, 0.55555556,
    0.66666667, 0.77777778, 0.88888889, 1.])
y : DeviceArray([0., 0.22222222, 0.44444445, 0.6666667, 0.8888889,
    1.1111112, 1.3333334, 1.5555556, 1.7777778, 2.], dtype=float32)
```

9. https://colab.research.google.com/. On peut disposer d'un GPU K80 gratuitement pour 24h et exécuter un notebook d'un repository github. Voir pour cela https://stackoverflow.com/questions/62596466/how-can-i-run-notebooks-of-a-github-project-in-google-colab (réponse 2)
10. En l'occurrence, à la date de cette note c'est la '0.2.20'.

La structure des données de `JAX` sont des `DeviceArrays` qui remplacent les tableaux de `Numpy`. Notez que le type par défaut est float32, si besoin pour passer en 64-bits il faut procéder à l'édition d'une variable d'environnement avant l'importation des librairies [11]

```
import os
os.environ['JAX_ENABLE_X64']='True'
```

Les `DeviceArrays` permettent à `JAX` de procéder aux suivis des dépendances des variables et l'optimisation du code pour les différents types de hardwares.

Une chose qui peut dérouter le premier contact avec les `DeviceArrays` de `JAX` quand on est habitué à la librairie `Numpy`, est la chose suivante que l'on peut constater avec les vecteurs x et y définis ci-dessus. S'il est simple de changer le signe de $x[1]$ par contre si l'on fait

```
y[1] *= -1
```

alors on tombe sur une erreur

```
TypeError: '<class 'jaxlib.xla_extension.DeviceArray'>' object does not
    support item assignment. JAX arrays are immutable. Instead of ``x[idx]
    = y``, use ``x = x.at[idx].set(y)`` or another .at[] method:
    https://jax.readthedocs.io/en/latest/jax.ops.html
```

La solution immédiate est de bien entendu d'aller lire la page web suggérée. Ainsi, on trouve la solution suivante

```
y = y.at[1].mul(-1)
```

Mais si la syntaxe est rébarbative, c'est pour la simple est bonne raison que les tableaux `DeviceArrays` sont des `immutables` (ie. ne peuvent être modifiés une fois créés), ce qui suggère qu'il faut sans doute penser autrement la manipulation des tableaux. Il en est de même avec les fonctions, lesquelles pour `JAX` doivent être des fonctions sans effets de bords (ce qui entre parenthèses n'est pas plus mal pour éviter des chausse-trapes pythonesques).

Une fois passer cette difficulté, on peut employer les `DeviceArrays` comme on le fait avec les tableaux `Numpy` par exemple comme arguments de fonctions `matplotlib`:

```
plt.scatter(x,y)
```

11. `dtype=jnp.float32` permet toujours de spécifier une précision réduite par la suite.

3 Variations autour d'une fonction simple

Prenons la fonction suivante

$$f(x) = e^{-x/2}\sin(x) \tag{1}$$

Avec un logiciel symbolique, il est simple d'obtenir les expressions des dérivées successives $f'(x)$ et $f''(x)$, que je ne donne pas mais qui sont notées $fp(x)$ et $fpp(x)$ dans le listing ci-dessous. Avec `JAX` certes les expressions analytiques ne sont pas accessibles, mais il est très simple par contre d'obtenir les graphes de ces fonctions (Fig. 1):

```
from jax import grad, vmap

def f(x):
    return jnp.exp(-x*0.5)*jnp.sin(x)

jfp  = vmap(lambda t:grad(f)(t))
jfpp = vmap(lambda t:grad(grad(f))(t))

fig,_ = plt.subplots(figsize=(8,8))
x=jnp.arange(0,10,0.1)
plt.plot(x,f(x), label="$f(x)$")
plt.plot(x,jfp(x), lw=3,label="$f^{\ \prime}(x)$")
plt.plot(x,fp(x),ls="—",c='k', label="$f^{\ \prime}(x)$ (exact)")
plt.plot(x,jfpp(x), lw=3,label="$f^{\ \prime\prime}(x)$")
plt.plot(x,fpp(x),ls="—", c='cyan',label="$f^{\ \prime\prime}(x)$
    (exact)")
plt.grid()
plt.legend();
```

Notez que j'utilise deux fonctions de la librairie `JAX`: la fonction `grad` qui calcule le gradient de la fonction par rapport à la variable x que l'on peut itérer pour obtenir les dérivées successives, et la fonction `vmap` qui permet d'appliquer la fonction sur des valeurs rangées dans un tableau (batching). Il y a une autre fonction similaire `jax.numpy.vectorize` qui a la même API que la version `Numpy`, mais elle est codée plutôt dans l'esprit de `vmap`. Les règles de vectorisation ne sont pas les mêmes et donc il faut déjà savoir ce que l'on veut

faire en pratique. Il serait trop long d'établir un "how-to" ici [12], et donc je vous renvoie à la doc [13], et exemples de codes que l'on trouve sur le Web.

A la différence d'une méthode de différences finies qui calcule une approximation des dérivées comme par exemple:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{2}$$

avec h suffisamment petit, `JAX` analyse le schéma de dépendance de f vis-à-vis de x directement à partir du code. On obtient alors une estimation aussi précise que le calcul analytique sans pour autant mener à bien le calcul. Bien entendu, on utilise la différentiation automatique dans des situations où le calcul ne peut pas être mené.

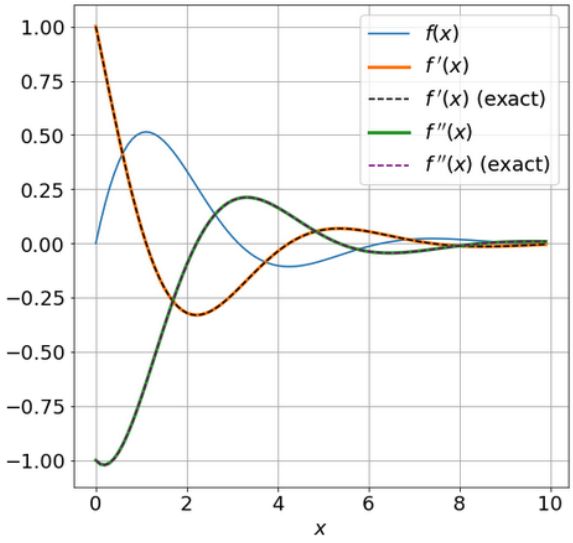

FIGURE 1 – Exemple de l'utilisation de la différentiation automatique pour calculer les dérivées successives d'une fonction de la variable x. On également présentés les graphes des fonctions dérivées dont les expressions analytiques sont connues.

12. Juste comme avant goût: `vmap` map une fonction le long *d'un seul axe spécifique*, tandis que `vectorize` permet d'appliquer le mapping sur plusieurs axes à la manière de `Numpy`. Pour des tableaux 1D mieux vaut utiliser `vmap` qui travaille sur 1 seul axe.

13. https://jax.readthedocs.io/en/latest/index.html

Imaginons à présent que l'on paramètrise la fonction $f(x)$ selon

$$f(p, x) = e^{-p_0 x} \sin(p_1 x) \tag{3}$$

et que l'on cherche à trouver les valeurs de p_0 et p_1 en disposant de mesures bruitées $\{x_i, y_i\}_{i<N}$. En utilisant simplement pour le moment `Numpy` on peut générer les échantillons, en définissant $g(x) = f(p_{true}, x)$ on a le code suivant:

```
ptrue = np.array([0.5,1])
g = lambda x: f(ptrue, x)


xin = np.arange(0,10,1.)
yerr = 0.05
yin = g(xin)+np.random.normal(scale=yerr,size=xin.shape)
```

Afin d'obtenir l'estimation des paramètres p_{est}, on se définit alors un *modèle* qui correspond à la définition de $f(p, x)$ et nous donne pour chaque couple de paramètres les valeurs des $\hat{y}_i = f(p, x_i)$. Ensuite, il nous faut un *fonction de coût* (risque empirique, *loss*) à minimiser. Là encore pas de surprise, nous allons utiliser la MSE (mean square error, ou méthode des moindres carrés):

```
def model(p, xi):
    return f(p,xi)
def mse(y_hat, y):
    return jnp.power(y_hat - y, 2).mean()
def loss_fun(p, xi, yi):
    yhat = model(p, xi)
    return mse(yhat, yi)
```

Au passage la fonction `mse` peut être codée selon

```
def mse(y_hat, y):
        return jnp.mean((y_hat - y)**2)
```

Il nous reste alors à procéder à l'estimation d'un pas de descente de gradient pour lequel on utilise la différentiation automatique:

```
def gradient_descent_step(p, xi, yi, lr=0.1):
    return p - lr * jax.grad(loss_fun)(p, xi, yi)
```

Notons que par défaut `jax.grad` opère sur le premier argument de la fonction à différentier mais cela est bien entendu ajustable:

```
def func(x, y):
  return 2 * x * y

grad(func)(3., 4.)  # 8.
grad(func, argnums=0)(3., 4.)  # 8.
grad(func, argnums=1)(3., 4.)  # 6
grad(func, argnums=(0, 1))(3., 4.)  # (8., 6.)
```

qui plus est la fonction peut être complexe (ie. holomorphe). Bien, finalement le code qui permet d'obtenir une estimation des paramètres s'écrit comme suit:

```
p_cur = jnp.array([0.1, 0.5])

for t in range(1000):

    if (t % 100) == 0:
        print(t, p_cur)

    new_p = gradient_descent_step(p_cur, xin, yin)

    rel_err = jnp.sqrt(jnp.mean((p_cur - new_p)**2))
    if rel_err < 1e-6:
        print(f"Converged after {t} epochs: p = {new_p}")
        break
```

Typiquement on obtient le message suivant:

```
Converged after 682 epochs: p = [0.47647393 0.9822352 ]
```

ce qui est correct avec l'usage du gradient simple. On peut utiliser la méthode de Newton qui fait appel au hessien de la fonction de coût (ie. matrice des dérivées secondes) selon l'opération

$$p_{n+1} = p_n - \lambda(\nabla_p^2 L(p_n))^{-1}\nabla_p L(p_n) \tag{4}$$

se code ainsi

```
gLoss = lambda p,xi,yi: jacfwd(loss_fun)(p,xi,yi)
hLoss = lambda p,xi,yi: jacfwd(gLoss)(p,xi,yi)

def oneStepNewton(p,xi,yi,lr=0.1):
    return p - lr*jnp.linalg.inv(hLoss(p,xi,yi)) @ gLoss(p,xi,yi)
```

Notons l'usage de la fonction `jacfwd`, elle calcule le jacobien de la fonction entre "()" par rapport au premier argument, mais ce dernier peut-être un vecteur tandis que `grad` devrait être vectorisée pour cela. A coté de `jacfwd` (forward), il y a la fonction `jacrev` (backward) et toutes les deux calculent le jacobien, mais la première est plutôt dédiée pour matrice à grand nombre de lignes tandis que la seconde est taillée pour les matrices à grande nombre de colonnes. Ainsi, pour plus d'efficacité c'est pour cette raison que la fonction `jax.hessian(f)` est codée selon `jacfwd(jacrev(f))`, ce qui serait une autre façon de coder `hLoss`.

Maintenant, en utilisant `oneStepNewton` en lieu et place de `gradient_descent_step`, alors on obtient une convergence plus rapide:

```
Converged after 232 epochs: p = [0.4764733 0.9822363]
```

Notez au passage l'usage de la librairie d'algèbre linéaire de `JAX` et l'usage de l'opérateur `Python` "@" pour la multiplication matricielle.

4 Un autre type d'optimisation

Le type de problème d'optimisation de la section précédente se présente dans beaucoup de domaines via les méthodes lagrangiennes. Par exemple, trouver les dimensions d'un pavé 3D dont la surface totale des cotés vaut par exemple 24, et pour lequel on veut maximiser le volume. Cela se résout en remplaçant la fonction de coût par le lagrangien suivant

$$\begin{aligned} \mathcal{L}(x,y,z,\alpha) &= V(x,y,z) - \alpha(S(x,y,z) - 24) \\ V(x,y,z) = xyz \quad &\text{et} \quad S(x,y,z) = 2(xy + xz + yz) \end{aligned} \tag{5}$$

ce qui se traduit par le code

```
# Volume de la boite
def vol(x):
    return x[0]*x[1]*x[2]
# Surface de la boite
def surf(x):
    return 2.*(x[0]*x[1]+x[0]*x[2]+x[1]*x[2])
```

```
# Constrainte sur la surface
def g(x): return surf(x) - 24

#Lagrangien : p[0:3] = (x1,x2,x3), p[3] = multiplicateur de lagrange
def Lag(p):
    return vol(p[0:3]) - p[3]*g(p[0:3])
```

On définit également le gradient et le hessien du lagrangien

```
gLag = jacfwd(Lag)
hLag = hessian(Lag)
```

comme précédemment, puis l'étape de la méthode de Newton

```
@jit
def solveLagrangian(p, lr=0.1):
    return p - lr*jnp.linalg.inv(hLag(p)) @ gLag(p)
```

Enfin, résoudre l'équation $\nabla\mathcal{L} = 0$ à la recherche du point critique se fait via

```
p_cur = jnp.array([1.5,0.5,1.0,0.1])

for t in range(200):

    if (t % 10) == 0:
        print(t, p_cur)

    new_p = solveLagrangian(p_cur)

    rel_err = jnp.max(jnp.abs(p_cur - new_p))
    if rel_err < 1e-6:
        print(f"Converged after {t} epochs")
        break

    p_cur = new_p
```

Ainsi, on obtient une estimation des paramètres dont on connaît par ailleurs la solution exacte:

```
Converged after 112 epochs
p_fin:  [1.9999964  1.9999899  1.9999952  0.49999797] : True x=y=z=2, lambda=0.5
v_fin:  7.999926 : True vol  = 2^3
s_fin:  23.999853 : True surf = 24
```

Même si on peut se rendre compte que l'environnement du point scelle solution n'est pas très bien "marqué", on peut tuner le learning rate pour mener à bien la minimisation, si ce n'est pas le cas il faut un peut tâtonner. Cependant, même dans cet exercice où l'optimisation est simple, on peut se demander s'il n'y a pas une sensibilité de la solution trouvée par rapport aux valeurs initiales que l'on donne (ie. la première valeur des paramètres `p_cur`). Bien entendu, on peut le faire à la main via une fonction qui prend en argument les valeurs initiales des paramètres que l'on tire au hasard. Mais, on peut le faire faire (plus efficacement [14]) par `JAX` aussi!

La première chose à faire est de vectoriser la fonction `solveLagrangian` pour quelle accepte plusieurs vecteurs `p_cur` à l'initialisation, et quelle puisse les mettre à jour en même temps:

```
vfuncsLAG = vmap(solveLagrangian)
```

Ensuite, il faut pouvoir tirer aléatoirement les dites valeurs initiales. Pour se faire `JAX` a une routine *ad hoc* `jax.random.PRNGKey(seed)` pour garantir la génération de nombres aléatoires *reproductible, vectorisable* et *parallélisable*. Ainsi, on utilise le code suivant pour générer 100 vecteurs `p_cur` (de dimension 4), qui je pense est assez clair:

```
key = jax.random.PRNGKey(0)
p_cur = jax.random.uniform(key, shape=(100,4), dtype='float32', minval=0.,
    maxval=1.0)
```

On procède à un certain nombre d'itérations

```
for t in range(3000):
    p_cur = vfuncsLAG(p_cur)
```

pour obtenir à l'étape finale 100 jeux de paramètres. Pour déterminer le meilleur, on utilise la fonction `vol` pour déterminer le jeu qui réalise le volume maximal. Pour se faire on utilise la fonction `nanargmax` de `jax.numpy` qui fonctionne comme son pendant de `Numpy`:

```
maxfunc  = jax.vmap(vol)
maximums = maxfunc(p_cur)
arglist = jnp.nanargmax(maximums)
argmax = p_cur[arglist]
maxvol = maximums[arglist]
print(f"The max. volume is {maxvol:.6f}, the lengthes are
    ({argmax[0]:.6f},{argmax[1]:.6f},{argmax[2]:.6f})")
```

14. nb. dans le notebook, il est judicieux de laisser l'activation de `JIT` pour ne pas attendre inutilement.

On obtient alors si on a 'bien' choisi le learning rate et suffisamment d'itérations

```
The max. volume is 8.000013, the lengthes are (2.000001,2.000001,2.000001)
```

Encore une fois si le problème est mal conditionné, ce n'est pas `JIT` qui sauvera sa résolution. Bien entendu des choses plus complexes sont possibles, en particulier le sampling de densités de probabilités par méthodes Monte Carlo de type MCMC Hamiltonien que j'aborderai dans une autre note à venir.

5 JIT: Just-In-Time compilation

Tout au long des exemples précédents, il y avait des lignes "@jit", il s'agit d'un décorateur Python qui appelle l'optimisation du code par son traitement par `XLA`. `JIT` permet de produire un code "optimisé" pour le hardware utilisé, essentiellement CPU et GPU/Nvidia (ie. le TPU est très spécifique). Une première idée de ce que peut apporter la compilation peut être touché du doigt en *commentant* les lignes "@jit" dans le notebook et en le ré-exécutant. Le gain de l'usage de la compilation est suffisamment significatif pour qu'il soit "visible" même sur un CPU. Ce n'est pas toujours le cas. Voyons par exemple un code purement en `Numpy` qui calcule par approximation de Simpson des intégrales d'un lot de 200 fonctions de type

$$f_n(x) = x^{n/10} e^{-x} \tag{6}$$

sur 100 intervalles:

```
def simps(f, a, b, N):
    #N doit etre paire
    dx = (b - a) / N
    x = np.linspace(a, b, N + 1)
    y = f(x)
    w = np.ones_like(y)
    w[2:-1:2] = 2.
    w[1::2]   = 4.
    S = dx / 3 * np.einsum("i...,i...",w,y)
    return S

def funcN(x):
    return np.stack([x**(i/10) * np.exp(-x) for i in range(200)],axis=1)
```

```
a = np.arange(0,10,0.1)
b=a+0.05
```

Le timing du code dans le jupyter notebook se fait via

```
%timeit simps(funcN,a,b, 512)
```

et donne typiquement

```
1 loop, best of 5: 1.05 s per loop
```

Maintenant envisageons un code en `JAX/JIT`:

```
from functools import partial
from jax.config import config
config.update("jax_enable_x64", True)

@partial(jit, static_argnums=(0,3))
def jax_simps(f, a,b, N):
    dx = (b - a) / N
    x = jnp.linspace(a, b, N + 1)
    y = f(x)
    w = jnp.ones_like(y)
    w = w.at[2:-1:2].set(2.)
    w = w.at[1::2].set(4.)
    S = dx / 3. * jnp.einsum('i...,i...',w,y)
    return S

@jit
def jax_funcN(x):
    return jnp.stack([x**(i/10) * jnp.exp(-x) for i in range(200)],axis=1)

ja = jnp.arange(0,10,0.1)
jb = ja+0.05
```

La première chose spécifique à la comparaison ici entre `Numpy` et `JAX` est le fait que l'on exige de travailler en double précision qui est le fonctionnement par défaut de `Numpy`. Ensuite, on remarque les décorateurs `Python` "`@partial`" et "`@jit`": ce sont juste des raccourcis. En effet, je pouvais très bien écrire

```
jit_funcN = jax.jit(jax_funcN)
```

qui permet d'utiliser la fonction compilée `jit_funcN(x)`, mais le décorateur permet d'utiliser directement `jax_funcN` compilée sans faire appel à une autre définition. Pour ce qui concerne le code utilisant "`@partial`", il se déploie comme cela

```
jit_simps = partial(jax.jit(jax_simps(f, a,b, N), static_argnums=(0,3))
```

En fait, `partial` permet de définir de nouvelles fonctions, à partir d'une fonction à plusieurs arguments dont certains sont fixés au moment de l'exécution du `partial`: ici la fonction c'est `jax.jit` et l'argument fixé est `static_argnums=(0,3)`. Maintenant, que fait cet argument (ie. `static_argnums`) de `jax.jit`. Il fige comme le terme "static" l'indique les arguments en position 0 et 3, à savoir les paramètres "f" et "N". En gros, le compilateur doit connaître un certain nombre de caractéristiques sur les arguments: concernant le paramètre "f" ce n'est pas un tableau standard puisque qu'il contiendra par la suite des fonctions "callable" ce que ne peut anticiper le compilateur, donc il faut le rendre statique. On aurait généré le même problème avec une fonction triviale qui a deux arguments "(f,x)" et retourne "f(x)". Concernant "N", c'est le `jnp.linspace(a,b,N+1)` qui pose problème et l'erreur aurait été la même que si on essaye d'opérer un `jit` sur une fonction de "N" qui renvoit `jnp.linspace(0,1,N+1)`, `XLA` doit pouvoir inférer la taille de tableau généré. Passant au benchmarking,

```
#warm up
jax_simps(jax_funcN,ja,jb, 512).block_until_ready()
#Go
%timeit jax_simps(jax_funcN,ja,jb, 512).block_until_ready()
```

Si le device est le CPU qui a servi au test avec le code `Numpy`ci-dessus, alors le timing est très similaire

```
1 loop, best of 5: 846 ms per loop
```

donc là pour le coup on a un gain relativement faible de 20%, on pourrait même avoir une perte de temps si on demande moins de calcul d'intégrations dans cet exercice. Le gain peut être dû à la fusion d'opérations lors de l'analyse du code. Par contre, si on dispose d'un GPU (ex. K80) alors c'est différent

```
100 loops, best of 5: 7.02 ms per loop
```

or le code n'a pas changé pour être adapté à la gestion du GPU, pour l'utilisateur c'est un réel gain. Notons que si on avait travaillé en simple précision le temps aurait été divisé

par 2. Pour finir on m'a signalé également que l'on peut accélérer le code CPU (presque un facteur 2) qu'il soit en `Numpy` ou bien en `JAX` en utilisant le code suivant pour définir les fonctions `funcN`:

```
def funcN(x):
  x = x[:, None, :]
  i = np.arange(200)[:, None]
  return x**(i/10) * np.exp(-x)
```

comme quoi il y a toujours moyen d'optimiser son code...

La phase de "warmup" utilisée avant de compter le temps d'exécution est là pour ne pas prendre en compte le temps de la compilation. Dans des applications plus complexes ce temps de compilation, où `XLA` fait son boulot comme il peut pour optimiser le code, peut prendre "un certain temps" (long). Concernant le `block_until_ready`, cela évite d'avoir de trop bonnes surprises, car `JAX` fonctionne en mode asynchrone et on pourrait ne comptabiliser qu'une partie de son temps d'exécution: il redonnerait la main à `Python` avec un `DeviceArray` qui ne serait rempli que dans le "futur".

Notez que `JIT` lance une nouvelle compilation de `jax_simps` si "N" change, si les tableaux `(a,b)` changent de taille et si les fonctions changent, et qu'il ne trouve pas une version déjà compilée satisfaisant au nouveau cas de figure dans son cache. Ce n'est donc pas une compilation comme on le penserait en C/C++. Mais cet état de fait pourrait changer dans des versions futures. Un "hack" pour détecter si la fonction est recompilée est de faire imprimer une string (ex. `print`("compilation en cours")), car cet affichage est un effet de bord qui n'est pas tracé par `XLA`, et un appel de fonction déjà compilée n'exécutera pas cet ordre. Un exemple est donné dans le notebook. Il est aussi montré que si une fonction (`titi(x)`) en appelle une en interne (`toto(x)`), la compilation de la première déclenche celle de la seconde. C'est quelque chose qui également pourrait évoluer dans le futur: ex. imaginons que `titi(x)` fasse des calculs sans rapport avec `toto(x)`, pour l'heure la modification de ces calculs déclenche la cascade de compilations. Donc, les codes les plus performants seront ceux qui n'entrainent qu'un minimum de compilations.

Concernant le contrôle du flux, `JIT` peut demander un peu de travail aussi. Par exemple, pour coder la fonction

$$f(x) = \lambda \begin{cases} x & \text{si } x > 0 \\ \alpha(e^x - 1) & \text{si } x \leq 0 \end{cases} \tag{7}$$

avec $\alpha = 1.67$ et $\lambda = 1.05$ (valeurs recommandées pour cette fonction d'activation), on serait tenté de procéder comme suit (horreur...):

```
@jit
def f(x,alpha=1.67, lambda_=1.05):
        if x>0:
                return lambda_*x
        else:
                return lambda_*alpha*(jnp.exp(x) -1)
```

A l'appel de `f(1.)` il y a une erreur pour la cause suivante:

```
ConcretizationTypeError: Abstract tracer value encountered where concrete
value is expected: Traced<ShapedArray(bool[], weak_type=True)>with
<DynamicJaxprTrace(level=0/1)>
The problem arose with the `bool` function.
While tracing the function f at /tmp/ipykernel_4111/3594871729.py:1
for jit, this concrete value was not available in Python
 because it depends on the value of the argument 'x'.
```

La notion de "tracer" est reliée à la façon dont `XLA` manipule ses objets pour suivre la dépendance des fonctions vis-à-vis de leurs arguments (ie. pour faire la différentiation). Ici, il a un hic, comme d'ailleurs l'aurait `Numpy` si on tentait d'exécuter le même code avec un tableau: `XLA` ne sait pas traiter le contrôle de flux "x>0". En gros, il essaye de concevoir un code avec un niveau d'abstraction de l'élément "x" comme si c'était un tableau de dimension fixe d'un type donné (ShapedArray). Imaginons qu'un code fonctionne pour un tableau ShapedArray 1D de dimension 10 et de type float, alors le code sera compilé pour tous ces types de tableaux et gardé en cache. Mais dans notre cas, il y a un conflit, car génériquement comment décider de la valeur a priori True ou False d'un tableau. D'une manière générale, il faut raisonner différemment. Dans ce cas précis, il faut coder la fonction selon:

```
@jit
def f(x,alpha=1.67, lambda_=1.05):
        return lambda_ * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)
```

On peut alors ploter la fonction ainsi que son gradient:

```
x = jnp.arange(-1,1,0.01)
plt.plot(x,vmap(fok)(x))
plt.plot(x,vmap(grad(fok))(x)) # le gradient...
```

(nb. le code en pure `Numpy` serait le même en utilisant `np.where` et `np.vectorize` pour ploter la fonction, mais pas le gradient). Une autre possibilité aurait était de déclarer static l'argument associé à "x" mais alors on serait confronté à un autre problème c'est que `JIT` exige des hashtables pour ce type d'argument. La raison en est que les fonctions compilées sont rangées dans des hashtables. Moyennant quoi on ne peut utiliser des tableaux (même jax-like). Et cette solution serait également bizarre car on ne pourrait pas la différentier... Bref, à retenir la solution du `jnp.where`.

Une autre thématique concerne le traitement des `for-loop`. Voici deux codes fonctionnels qui calculent tous les deux une approximation de la racine carré de x:

```
@jit
def root(x):
    val = x
    for i in range(0,10):
        val = (val+x/val)/2.
    return val
@jit
def jax_root(x):
    def body(i,val):
        return (val+x/val)/2.
    return jax.lax.fori_loop(0,10,body,x)
```

Ils compilent tout à fait, peuvent être appelés avec des tableaux de nombres et calculés les gradients. Concernant le timing, les deux codes sont semblables sur CPU, et avec un avantage sur GPU à la méthode `root`. Par contre le calcul du gradient est en faveur de l'implémentation avec la fonction `jax.lax.fori_loop`. D'une manière générale, `XLA` quand il analyse la fonction `root` déplie la boucle `for-loop` et optimise le code, dans le second cas il détecte l'usage de `jax.lax.fori_loop` et à la possibilité de procéder à d'autres ajustements. En tout état de cause, déplier toutes les "for-loop" que peut contenir une fonction tend à faire grossir le code compilé, donc il y a un tradeoff. Sur GPU, les échanges trop répétés avec le CPU peuvent par ailleurs obérer les performances. Il est donc préférable d'envisager l'usage des boucles `fori_loop` ainsi que `while_loop` de `lax` quand on a beaucoup de boucles dans son code.

6 Take away message

Dans cette note j'ai voulu vous sensibiliser à `JAX` qui fait partie des librairies qui mettent en œuvre la différentiation automatique et qui peut servir à optimiser son code `Python`. La différentiation automatique est un monde en soit avec sa communauté, et donc j'ai laissé de coté pas mal d'aspects [15]. Maintenant, vous pouvez toujours utiliser le code `Numpy` que vous connaissez, et vous exercer à le remplacer petit-à-petit par son équivalent `JAX`, puis utiliser `JIT`. Vous verrez bien si cela vous aide ou pas.

Maintenant, il y a des librairies qui sont tout ou partie réécrites en `JAX`: nous avons vu l'implémention de `jax.numpy`, sachez qu'il y a aussi une partie de `scipy` qui est déjà accessible (https://jax.readthedocs.io/en/latest/jax.scipy.html?highlight=scipy) (voir aussi https://jax.readthedocs.io/en/latest/jax.html), qu'il y a une implémentation de quelques méthodes à noyaux de `sklearn` (https://github.com/ExpectationMax/sklearn-jax-kernels), et que la librairie de calcul statistique `Pyro` est réécrite en `JAX` (`Numpyro` en version beta, http://num.pyro.ai/en/stable/). Nous verrons une application avec cette dernière librairie. Enfin, cette liste n'est pas exhaustive des projets de softwares écrits à base de `JAX`, donc *stay tuned*...

15. Voir la revue de Charles C. Margossian (Jan. 2019) https://arxiv.org/pdf/1811.05031.pdf