Python加速技巧

一、概述

1. 加速的数据处理概述
    i. 为什么需要加速
    ii. 有哪些加速方法
2. Python 自身的局限性
    i. 原生加速技巧（参考流畅的Python）
    ii. 多核加速技巧（参考多进程计算）
    iii. 尽量使用 numpy
    iv. numba 加速技巧
    v. 多集群加速技巧

二、原生加速技巧

技巧一：

del 和垃圾回收

在 Python 中，当一个对象失去了最后一个引用时，会当作垃圾回收掉。但要注意，`del` 并不是删除掉对象，而是删除掉「对象」和「变量」的联系。只有当一个对象没有任何「引用」时，它才会被回收掉。

```
m1 = model
m2 = model
del m1
# model 对象仍然在，因为 m2 和 model 还有联系。model 对象仍然占据了内存。
```

技巧二：

for 和列表推导

在 Python 中，涉及 list , dict , tuple 等操作，尽量少使用 for 语句，能使用列表推导，尽量使用列表推导。使用列表推导，不仅减少了代码的书写量，它的执行效率也明显快于 for 语句。


```
lst = []
for i in range(10000000):
    lst.append(i)

[i for i in range(10000000)]


lst = []
for i in range(10000000):
    if i % 5 == 0 :
```

```
        lst.append(i/2)
    else:
        lst.append(i*2)

[i/2 if i % 5 == 0 else i*2 for i in range(10000000)]
```

技巧三：

dict 和 list操作时间复杂度

dict 和 list 有些情况下虽然可以混用，但由于它们的本身数据结构的特点，会导致它们的在做相同操作下的时间复杂度会大相径庭。

list 操作的时间复杂度

操作	操作说明	时间复杂度
index(value)	查找list某个元素的索引	O(1)
a = index(value)	索引赋值	O(1)
append(value)	队尾添加	O(1)
pop()	队尾删除	O(1)
pop(index)	根据索引删除某个元素	O(n)
insert(index, value)	根据索引插入某个元素	O(n)
iterration	列表迭代	O(n)
search(in)	列表搜索（其实就是in关键字）	O(n)
slice [x:y]	切片, 获取x, y为O(1), 获取x,y 中间的值为O(k)	O(k)
del slice [x:y]	删除切片，删除切片后数据需要重新移动/合并	O(n)
reverse	列表反转	O(n)
sort	排序	O(nlogn)

dict 操作的时间复杂度

操作	操作说明	时间复杂度
copy	复制	O(n)
get(value)	获取	O(1)
set(value)	修改	O(1)
delete(value)	删除	O(1)
search(in)	字典搜索	O(1)
iterration	字典迭代	O(n)

技巧四：

全局变量与局部变量

Pyhton 的全局变量相比于局部变量调用增加耗时，在不大幅度增加代码复杂度的情况下，尽量少使用全局变量。

```
1 import time
2
```

```
num = 100000000
def count1():
    for i in range(num):
        i
    return

def count2():
    num = 100000000
    for i in range(num):
        i
    return

s1 = time.time()
count1()
s2 = time.time()
print(s2-s1)

s1 = time.time()
count2()
```

```
s2 = time.time()
print(s2-s1)
```

技巧四：

import 和 from import

python 在进行属性访问时，会调用对象的 `__getattribute__` 和 `__getattr__` 方法，从而导致额外的开销。所以在不影响代码阅读的情况下，可以尽量使用 `from import` ，代替 `import` 。

```
import math
import time

def func():
    lst = []
    for i in range(1, 10000000):
        lst.append(math.sqrt(i))
```

```
    return lst

start = time.time()
lst = func()
end = time.time()
print(end-start)

from math import sqrt
import time

def func():
    lst = []
    for i in range(1, 10000000):
        lst.append(sqrt(i))
    return lst

start = time.time()
lst = func()
end = time.time()
```

```
27 print(end-start)
```

三、多核加速技巧

由于 Python GIL 锁的存在，导致无论是 Pyhton2 还是 Python3 的解释器在执行时只能调用一个 CPU 核心。对于 CPU 计算密集型的任务是非常吃亏的。解决这个问题的一个简单的想法是使用多进程，每个进程都是单独的运行环境，每个进程都是单独的 GIL 锁，所以可以使用 CPU 的多核计算。

注意，在 I/O 密集型的场景，Python 的这个短板相比于它语法的简易性可忽略不计。

请大家思考两个问题：

问题一：I/O 密集型最典型场景是什么？ 问题二：Python GIL 锁带来的短板会影响

AI 的计算吗?

多进程计算一般会使用 `multiprocessing` 这个包，它的大致流程是，一个主进程运行代码，将代码的内容分发到其他子进程上分别进行异步计算，此时一般主进程会阻塞（也可以不阻塞）。等待其余进程计算完成后，主进程继续后面的代码逻辑。

```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print('Run task %s (%s)...' % (name, os.getpid()))
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print('Task %s runs %0.2f seconds.' % (name, (end - start))
)

```

```
if __name__=='__main__':
    print('Parent process %s.' % os.getpid())
    p = Pool(4)
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print('Waiting for all subprocesses done...')
    p.close()
    p.join()
    print('All subprocesses done.')
```

上面的实现中，各个进程间是没有交互的，一般情况下进程之间的内存是不共享的。如果想让进程的计算结果进行交互，则需要进程间的通信，在 `multiprocessing` 包中，最常用的使用的是 `Queue`，这是一种队列的数据结构。

```
from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码:
```

```
def write(q):
    print('Process to write: %s' % os.getpid())
    for value in ['A', 'B', 'C']:
        print('Put %s to queue...' % value)
        q.put(value)
        time.sleep(random.random())

# 读数据进程执行的代码:
def read(q):
    print('Process to read: %s' % os.getpid())
    while True:
        value = q.get(True)
        print('Get %s from queue.' % value)

if __name__=='__main__':
    # 父进程创建Queue，并传给各个子进程:
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
```

```
    # 启动子进程pw，写入：
    pw.start()
    # 启动子进程pr，读取：
    pr.start()
    # 等待pw结束：
    pw.join()
    # pr进程里是死循环，无法等待其结束，只能强行终止：
    pr.terminate()
```

有时，我们仅需将大数据划分几部分，每部分做相同计算操作，然后合并在一起。经常使用的是 `Pool` 类中的 `imap` 方法。

```
from multiprocessing import Pool
def func(lst):
    return sum(lst)

ret_lst = []
with Pool(4) as pool:
    lst = [[i for i in range(10000000)] for i in range(4)]
```

```
    iter = pool.imap(func, lst)
    for ret in iter:
        ret_lst.append(ret)

```


四、尽量使用 numpy

虽然 Python 自带的数据结构和原生语法已经能处理大量的复杂的逻辑计算。但是在数值计算着一块，要尽量使用 numpy 的语法而不是原生的 Python 实现。

numpy 在程序语言上，是用的 C 和 Fortran 等古老的编译型语言。

numpy 在工程实践上，复用了 blas , lapack , intel MKL , OpenBlas , ATLAS 等优秀的数值计算库，工程上做了大量优化。

numpy 在算法优化上，数值计算的相关算法几乎是优化到了极致。

只要涉及到数值计算的相关内容，能用 numpy 替换 python 原生代码就尽量替换。

五、numba 加速技巧

numba官方文档 numba中文文档

虽然 numpy 相比于原生的 python 代码在数值计算上有了很大的提升。但是一段计算基本是由很多部分的逻辑组成。可能除了 numpy 的计算部分外，还有 python 的一些 for 分支语句。除此外，现在 GPU 计算已经非常成熟，如果能使用 GPU 代替 CPU 进行计算则能进一步的加快计算的效率。那么 numba 的原理到底是怎么样的呢?

讲解这个之前，我们想来了解编程语言的两种分类：

- 编译器语言

- 解释器语言

编译器语言的代表是 C 语言，解释器语言的代表是 Python。高级程序语言都需要转换为机器可以理解的机器码才能交由 CPU 或者 GPU 执行。编译器语言可以理解为由编译器一次性将代码翻译成机器码（翻译过程中会有优化），解释器语言则是在执行过程中通过解释器一行一行翻译，解释器在其中起到了类似于同传「翻译官」的作用。

从上面简单的描述能看到，编译器型的语言无论在优化上还是在效率上相比解释器语言会更快。所以一个优化解释器语言的一种简单想法是尽量减少解释器在代码翻译中的参与度。

`numba` 可以看成一种动态编译的工具，即将 Python 中的代码部分甚至全部直接翻译成机器码，这样就能部分或者全部的绕过 Python 的解释器。所以 `numba` 的加速有两种选项：

- nopython 模式

    - 将选定的 Python 代码 **完全** 翻译成机器码，不让 Python 解释器参与选定代码的执行。
- object mode 模式
    - 将选定的 Pyhton 代码 **部分** 翻译成机器码，Python 解释器仍会部分参与选定代码的执行。（编译哪部分将由 numba 自动选择）

```
@jit(nopython=True)
def func(x):
    pass
```

5.1 编译 Python 代码

```
# 仅在程序第一执行时才会编译，数据类型将通过执行时输入的数据自行推断。
from numba import jit
@jit
def f(x, y):
```

```
    # A somewhat trivial example
    return x + y
```

```
# 人工指定数据的类型
from numba import jit, int32

@jit(int32(int32, int32))
def f(x, y):
    # A somewhat trivial example
    return x + y
```

```
# 编译函数可以嵌套调用
@jit
def square(x):
    return x ** 2

@jit
def hypot(x, y):
```

```
    return math.sqrt(square(x) + square(y))
```

5.2 便捷生成 numpy 通用函数

numpy 通用函数(ufunc) 可以简单理解为是一种可以支持数组广播的函数。numpy 的数组广播功能是它能做快速数值计算的一个很重要的原因之一(很多博客和教材中把它称为 numpy 的向量化计算)。原生 numpy 的通用函数书写很复杂，numba 简化了这个操作。

```
from numba import vectorize, float32, float64, int32, int64

@vectorize([float64(float64, float64)])
def f(x, y):
    return x + y

@vectorize([int32(int32, int32),
            int64(int64, int64),
```

```
            float32(float32, float32),
            float64(float64, float64)])
def f(x, y):
    return x + y
```

5.3 numba 的自动并行化功能

numpy 通用函数速度快的原因在于它是一种并行化的计算，比如前面的两个 `array` 相加，可以将 `array` 划分成很多个子 `array`，然后并行相加。对于一个复杂的计算逻辑，我们期望有一套自动的进行并行化的方法。

numba 的自动并行化是指，在编译时，将向量维度（张量）不变的算子（计算操作）融合到一起，形成一个或多个可以并行自动运行的内核。

```py
@numba.jit(nopython=True, parallel=True)
def logistic_regression(Y, X, w, iterations):
    for i in range(iterations):
        w -= np.dot(((1.0 / (1.0 + np.exp(-Y * np.dot(X, w))) - 1.0) * Y), X)
    return w
```

我们不会讨论算法的细节，而是关注该程序如何使用自动并行化：

1. 输入 `Y` 是大小为 `N` 的向量，`X` 是 `N x D` 矩阵，`w` 是大小为 `D` 的向量。
2. 函数体是一个迭代循环，它更新变量 `w` 。循环体由一系列向量和矩阵运算组成。
3. 内部 `dot` 操作产生一个大小为 `N` 的向量，然后是标量和大小为 `N` 的向量之间的一系列算术运算，或两个大小为 `N` 的向量。
4. 外部 `dot` 产生一个大小为 `D` 的向量，然后在变量 `w` 上进行就地数组减法。
5. 通过自动并行化，将生成大小为 `N` 的数组的所有操作融合在一起，成为单个并行内核。这包括内部 `dot` 操作和后面的所有逐点数组操作。
6. 外部 `dot` 操作产生不同维度的结果数组，并且不与上述内核融合。

对于 GPU 环境下的 numba 加速这里不再进一步展开。

请大家思考一个问题：

问题三：如此看来 `numba` 加速的原因在于编译。将人写的代码编译成机器码，并自动做了一定的优化。那么除了人写的代码能「编译优化」外？你觉得还有什么也可能进行「编译优化」呢？

六、多集群加速

前面讲解的 Python 加速都是针对于单机多核的加速。在大数据时代，数据的量有时候是非常庞大的，庞大到单机的计算资源已远远无法满足计算的需求。比如当你要处理的数据超过了 1T 怎么办？

> 一台机器搞不定，那就交给多台机器来搞定。

一个计算任务，交由多台机器（集群）共同完成可以简单的分为两个过程。

- 分治过程：将任务拆分开，让大家分别并行执行。
- 聚合过程：将任务结果聚合，分发到下一个计算任务。

计算任务这里是一个比较抽象的概念，一个 Python 的计算脚本可能是由很多计算任务组成。上面的这种编程思想其实就是著名的MapReduce。注意，一个计算任务可能会形成一个很复杂的有向无环图。这类编程可以称为面向 dataflow 编程（包括 tensorflow，pytorch）

目前 Python 的数据处理的多集群计算引擎常用的主要有以下框架，以下框架的学习内容都非常的繁多，这里只能做简单的介绍，有兴趣的读者可以自行搜集相关资料学习。

- Hive
    - 底层由 Java 开发，经典的 MapReduce 框架，使用中心化的节点形成调度的 graph，Hive 实际是一套把 SQL 转换成 MapRecude 编程范式的框架。

在 SQL 中我们可以写 UDF。那么利用 Python-UDF 就能完成分布式的 Python 脚本计算。

- Spark
  - 底层由 Scala 开发，利用 Pyspark 接口可以完成 Python 脚本的分布是计算。
- Dask
  - 底层由 Python 开发，Python 脚本可以无缝替换到 Dask 上进行分布是计算。
- Ray
  - 底层主要由 C++ 开发，actor 模型，非常适用于流图计算，在线学习，强化学习等场景。Ray 假设了函数（被 @ray.remote 修饰的函数）自身需要满足幂等性（即同样的输入多次执行结果一致并且没有副作用），这简化了重试操作和错误恢复（但 actor 可以是有状态的）