

# 第九周问答作业

朱士杭 231300027

2024 年 5 月 16 日

## 1 问题一：广播如何逐元素加法

根据广播的原则，从后往前逐一匹配，对于数组 A 与数组 B 最后一个元素为 5，得以匹配；然后再匹配倒数第 2 个元素，发现 A 是 1，B 是 3，因此 A 的 1 应当广播成 3；然后再比较倒数第三个元素这个时候数组 B 没有更高的 axis 因此会扩展一个 axis 并且与 A 进行相应的匹配成 3，因此最后的数组形状为 (3,3,5)，将扩展之后的数组 A 与数组 B 再逐一进行加法操作。

## 2 问题二：广播数组乘法

在两个 ndarray 数组运算过程中，先会进行从右往左的匹配操作，当两个维度长度相等或者其中一个为 1 时，两者 compatible，当其中一个矩阵维度不足时，用长度为 1 的新维度进行补充。而广播则是将长度为 1 的维度内容复制匹配另一矩阵对应长度对于维度为 (2,1) 数组 A 与维度为 (1,3) 数组 B，从右往左匹配的时候发现都可以匹配的上，接下来进行广播，将 A 的 1 广播为 3，B 的 1 广播为 2 都形成 (2,3) 的数组，然后逐元素进行 multiply 操作，最终结果数组形状为 (2,3) 假设数组 A 为  $[[a_{11}], [a_{21}]]$ ，数组 B 形状为  $[[b_{11}, b_{12}, b_{13}]]$ ，广播之后分别为

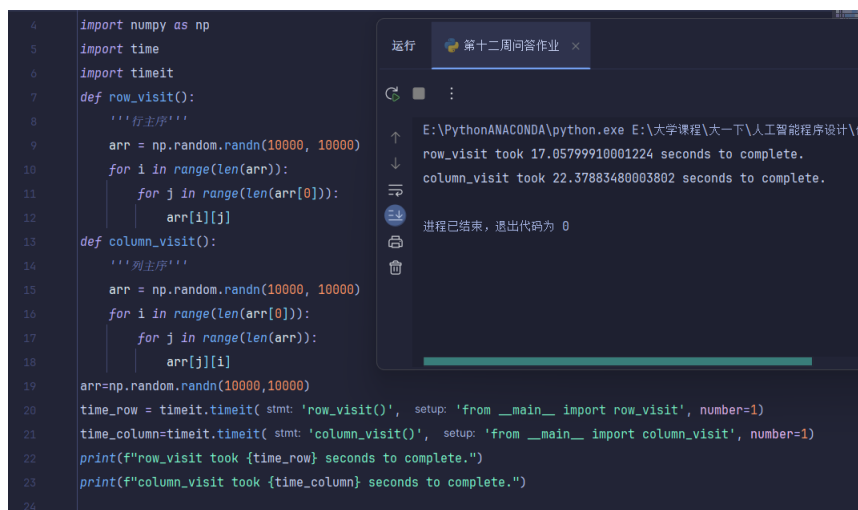
$$[[a_{11}, a_{11}, a_{11}], [a_{21}, a_{21}, a_{21}]]$$

$$[[b_{11}, b_{12}, b_{13}], [b_{11}, b_{12}, b_{13}]]$$

则最终结果为

$$\begin{aligned} & [[a_{11} * b_{11}, a_{11} * b_{12}, a_{11} * b_{13}], \\ & [a_{21} * b_{11}, a_{21} * b_{12}, a_{21} * b_{13}]] \end{aligned} \quad (1)$$

### 3 问题三：行主序与列主序



```
4 import numpy as np
5 import time
6 import timeit
7 def row_visit():
8     '''行主序'''
9     arr = np.random.randn(10000, 10000)
10    for i in range(len(arr)):
11        for j in range(len(arr[0])):
12            arr[i][j]
13 def column_visit():
14     '''列主序'''
15     arr = np.random.randn(10000, 10000)
16     for i in range(len(arr[0])):
17         for j in range(len(arr)):
18             arr[j][i]
19 arr=np.random.randn(10000,10000)
20 time_row = timeit.timeit( stmt: 'row_visit()', setup: 'from __main__ import row_visit', number=1)
21 time_column=timeit.timeit( stmt: 'column_visit()', setup: 'from __main__ import column_visit', number=1)
22 print(f"row_visit took {time_row} seconds to complete.")
23 print(f"column_visit took {time_column} seconds to complete.")
24
```

运行 第十二周问答作业 ×

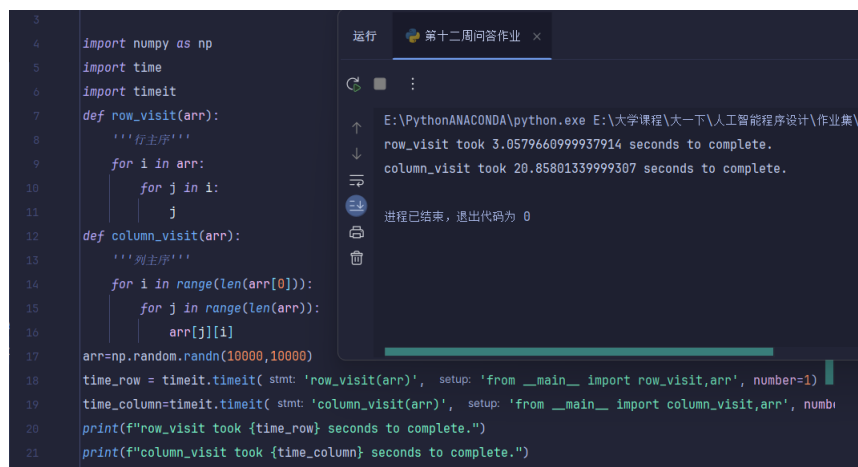
E:\Python\ANACONDA\python.exe E:\大学课程\大一下\人工智能程序设计\作业集\

row\_visit took 17.05799910001224 seconds to complete.

column\_visit took 22.37883480003802 seconds to complete.

进程已结束，退出代码为 0

图 1: 行主序与列主序访问 ndarray 数组性能差异（优化前）



```
3
4 import numpy as np
5 import time
6 import timeit
7 def row_visit(arr):
8     '''行主序'''
9     for i in arr:
10         for j in i:
11             j
12 def column_visit(arr):
13     '''列主序'''
14     for i in range(len(arr[0])):
15         for j in range(len(arr)):
16             arr[j][i]
17 arr=np.random.randn(10000,10000)
18 time_row = timeit.timeit( stmt: 'row_visit(arr)', setup: 'from __main__ import row_visit,arr', number=1)
19 time_column=timeit.timeit( stmt: 'column_visit(arr)', setup: 'from __main__ import column_visit,arr', numb
20 print(f"row_visit took {time_row} seconds to complete.")
21 print(f"column_visit took {time_column} seconds to complete.")

```

运行 第十二周问答作业 ×

E:\Python\ANACONDA\python.exe E:\大学课程\大一下\人工智能程序设计\作业集\

row\_visit took 3.0579660999937914 seconds to complete.

column\_visit took 20.85801339999307 seconds to complete.

进程已结束，退出代码为 0

图 2: 行主序与列主序访问 ndarray 数组性能差异（优化后）

结果很明显，按行访问的时间效率远远高于按列访问的时间效率

### 3.1 为什么会有性能差异

在内部存储当中，numpy 实际上是按照一行进行存储的，实际上就是一个一维数组，而计算机每一次从内存读取数据的时候都是先读一段到缓存当中（cpu 当中的寄存器）

由于时间重复性与空间重复性（即被访问的元素接下被再次访问到的概率更高，以及被访问的元素下面的元素被访问的概率也更高）因此同一行的元素被读进寄存器当中，如果按行进行访问的话每一次访问直接往下偏移即可，而如果按列进行访问的话，每一次又要从内存当中把下一行的数组读进寄存器当中，会大大降低时间效率

## 4 聚合函数

### 4.1 聚合函数内部实现机制

NumPy 的聚合函数有如 `np.mean`、`np.sum`、`np.max` 等，是 NumPy 中非常强大和常用的功能。这些函数可以沿着数组的指定轴进行操作，并且能够有效地处理多维数据。

在内部实现上，这些 NumPy 聚合函数主要使用 C 语言编写，使得它能够高效地处理数组运算。当在多维数组上应用聚合函数时，NumPy 会根据 `axis` 参数的值来确定如何遍历数组。`axis` 参数指定了操作的轴，即沿着哪个维度进行聚合。NumPy 内部存储当然是以一维数组去存储，并通过 `strides` 来确定访问某个元素的时候需要跳过多少个元素才能访问的到

### 4.2 如何指定 `axis` 参数

对于形状为 (3,4,5) 的三维数组，

#### 4.2.1 如果按照 `axis=0` 进行聚合

确定 `strides` 为 (20,5,1)，然后根据 0,1,2 对 3 个 (4,5) 的块进行遍历累加求平均值（这里默认是跳过多少个元素而不是跳过多少个字节）最后变成一个 (4,5) 的二维数组

```
22 arr=[[[[1,2],[3,4],[5,6],[7,8],[9,10]],
23        [[1,2],[3,4],[5,6],[7,8],[9,10]],
24        [[1,2],[3,4],[5,6],[7,8],[9,10]],
25        [[1,2],[3,4],[5,6],[7,8],[9,10]]],
26      [[[1,2],[3,4],[5,6],[7,8],[9,10]],
27        [[1,2],[3,4],[5,6],[7,8],[9,10]],
28        [[1,2],[3,4],[5,6],[7,8],[9,10]],
29        [[1,2],[3,4],[5,6],[7,8],[9,10]]],
30      [[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]],
31        [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]],
32        [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]],
33        [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]]]]
34 #arr为(3,4,5,2)的数组
35 arr=np.mean(arr,axis=0)#arr变为(4,5,2)的数组
36 print(arr)
```

```
[[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]
 [ 9. 10.]]
 [[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]
 [ 9. 10.]]
 [[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]
 [ 9. 10.]]]
```

图 3: 使用 mean 函数将一个 (3,4,5,2) 的数组按照 axis=0 进行聚合

#### 4.2.2 如果按照 axis=1 进行聚合

如果要按照 axis=1 进行聚合，对 4 个 (3,5) 的块进行遍历累加

#### 4.2.3 如果按照 axis=0 进行聚合

如果要按照 axis=2 进行聚合，对 5 个 (3,4) 的块进行遍历累加

### 4.3 背后的数据迭代逻辑

在内部，NumPy 使用了一种 strider 步进器来遍历数组。步进器是一种包含数组元数据和如何遍历数组的信息的数据结构，当一个聚合函数被调用时，NumPy 会创建一个步进器来遍历数组，对于每个轴，步进器会计算该轴的步长 strides 跳过多少个元素到达下一个沿着该轴的元素，在多维数组中，步进器会根据 axis 参数来确定如何遍历数组

## 5 聚合函数的具体应用

```
40 arr=np.random.uniform( low= 0, high= 1, size= (100,))
41 print("数组的平均值为:",arr.mean())
42 print("数组的总和为:",arr.sum())
43 print("数组的最大值为:",arr.max())
44 print("数组的最小值为:",arr.min())
```

```
E:\Python\ANACONDA\python.exe E:\大学课程\大一下\人工智能程序设计\作业集\
数组的平均值为: 0.5142493576615944
数组的总和为: 51.42493576615944
数组的最大值为: 0.9973630430763769
数组的最小值为: 0.01793285634765851
```

图 4: 计算并打印 ndarray 数组平均值、总和、最大值和最小值

## 6 条件函数筛选 ndarray 数组

```
44 arr=np.random.randint(1,100,size=(10,10))
45 print("所有大于50元素的标准差为:",arr[arr>50].std())
46
```

E:\Python\ANACONDA\python.exe E:\大学课程\大一下\人工智能程序设计\作业\第十  
所有大于50元素的标准差为: 16.31126248794425  
进程已结束，退出代码为 0

图 5: 条件函数筛选大于 50 元素，并计算这些元素标准差

## 7 三维矩阵类 ThreeDimMatrix

```
1个用法
46 class ThreeDimMatrix:
47     def __init__(self,depth,rows,columns):
48         '''初始化三维数组'''
49         self.depth=depth
50         self.rows=rows
51         self.columns=columns
52         self.arr=np.random.randint(0,10000,size=(depth,rows,columns))
53
54 4个用法
55 def slice(self,start,end,step=[1,1,1]):
56     '''切片操作 其中start、end、step对应三个切片，step可以缺失默认为1'''
57     step+=[1]*(3-len(step))
58     return self.arr[start[0]:end[0]:step[0],
59                    start[1]:end[1]:step[1],
60                    start[2]:end[2]:step[2]]
61
62 def verify_view(self):
63     '''通过slice方法得到视图，对视图进行修改，查看结果'''
64     ...
65
66 s = self.slice( start: (1, self.rows//3, self.columns//2),
67                end: (self.depth-1, self.rows-1, self.columns))
68
69 print("在verify_view()函数当中随机矩阵切片为:",s)
70 print("接下来将切片中所有元素置0")
71 s[:, :, :] = 0
72 print("将切片元素置为0后原来数组为:",self.arr)
73
74 matrix=ThreeDimMatrix( depth: 6, rows: 6, columns: 7)
```

图 6: 定义一个三维矩阵类 ThreeDimMatrix

首先先定义一个三维矩阵类 ThreeDimMatrix，其中包含初始化、切片操作、视图验证三个成员函数

```

78 matrix=ThreeDimMatrix( depth: 6, rows: 6, columns: 7)
79 print("原始矩阵为: ",matrix.arr)
80 s=matrix.slice( start: [0,2,1], end: [2,4,4], step: [1,2,1])
81 print("经过第一次矩阵切片后为: ",s)
82 s[:,:,:]=0
83 print("对切片进行修改后值为: ",matrix.arr)
84 # matrix.verify_view()
85 s=matrix.slice( start: [0,1,0], end: [3,2,4])
86 print("经过第二次矩阵切片后为: ",s)
87 s[:,:,:]=0
88 print("对切片进行修改后值为: ",matrix.arr)
89 s=matrix.slice( start: [2,0,1], end: [7,4,7])
90 print("经过第三次矩阵切片后为: ",s)
91 s[:,:,:]=0
92 print("对切片进行修改后值为: ",matrix.arr)

```

图 7: 多次切片操作

然后实例化之后再进行一次切片操作，并且将视图值全部置为 0，并且查看最终的修改结果

```

对切片进行修改后值为: [[[8295 103 3270 8688 420 3252 6865]
[ 0 0 0 0 1838 536 7727]
[5880 0 0 0 2713 7762 8720]
[9791 7659 7847 9171 1557 639 6729]
[7514 5139 3261 7082 7875 9268 6628]
[7789 1987 8398 3693 1184 6983 6015]]]

[[ 705 3312 9004 2829 9582 404 8270]
[ 0 0 0 0 449 5573 1964]
[ 616 0 0 0 6678 1915 9299]
[5836 619 6989 6660 9865 51 6773]
[3001 4057 837 3239 6954 9455 7334]
[6391 8167 2459 3673 2928 5040 9508]]

[[[8351 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0]
[ 155 0 0 0 0 0 0]
[6551 0 0 0 0 0 0]
[1979 4752 2166 1480 2363 2042 632]
[3096 1723 9910 4426 1802 5183 9922]]]

[[[4215 0 0 0 0 0 0]
[4661 0 0 0 0 0 0]
[1617 0 0 0 0 0 0]
[7583 0 0 0 0 0 0]
[8741 211 2536 17 5553 962 5157]
[7084 2874 2900 3576 2879 5303 557]]]

[[[5497 0 0 0 0 0 0]
[8879 0 0 0 0 0 0]
[5815 0 0 0 0 0 0]
[4612 0 0 0 0 0 0]
[4025 5546 3625 5689 426 491 74]
[8374 3475 9657 6200 4108 8310 7897]]]

[[[2537 0 0 0 0 0 0]
[6898 0 0 0 0 0 0]
[4363 0 0 0 0 0 0]
[7767 0 0 0 0 0 0]
[4548 7956 8413 3202 9527 3140 7980]
[3259 2770 2062 1742 5252 9430 6446]]]

进程已结束，退出代码为 0

```

图 8: 对视图的修改影响原始矩阵

最后发现对于视图的修改是会影响到原始矩阵的，因此可以得出结论，视图并不会创建新的 ndarray 数组对象，而是在原先的数组当中直接进行修改