NumPy views: saving memory, leaking memory, and subtle bugs

https://pythonspeed.com/articles/numpy-memory-views/

Itamar Turner-Trauring Thu Aug, 12 18:19

If you're using Python's NumPy library, it's usually because you're processing large arrays that use plenty of memory. To reduce your memory usage, chances are you want to minimize unnecessary copying,

NumPy has a built-in feature that does this transparently, in many common cases: memory views. However, this feature can also cause higher memory usage by preventing arrays from being garbage collected. And in some cases it can cause bugs, with data being mutated in unexpected ways.

To avoid these problems, let's learn how views work and the implications for your code.

Preliminary: Python lists

Before looking at NumPy arrays and views, let's consider a somewhat similar data structure: Python lists.

Python lists, like NumPy arrays, are contiguous chunks of memory. When you slice a Python list, you get a completely different list, which means you're allocating a new chunk of memory:

```
>>> from psutil import Process
>>> Process().memory_info().rss
12247040
>>> list1 = [None] * 1_000_000
>>> Process().memory_info().rss
20463616
>>> list2 = list1[:500_000]
>>> Process().memory_info().rss
24580096
```

Slicing the list allocated more memory. And since the second list is an independent copy, if we mutate it this won't affect the first list:

```
>>> list2[0] = 'abc'
>>> print(list2[0])
abc
>>> print(list1[0])
None
```

Note that the data that gets copied into the second list is *pointers* to Python objects, not the contents of the objects themselves. So even though the lists themselves are distinct, the underlying objects are still shared between the two.

NumPy arrays don’t copy when slicing (usually)

NumPy arrays work differently. Because the presumption is that you might be working with very large arrays, many operations don’t copy the array, they just give you a *view* into the same contiguous chunk of memory that the original array points at.

The first consequence is that slicing doesn’t allocate more memory, since it’s just a view into the original array:

```
>>> from psutil import Process
>>> import numpy as np
>>> arr = np.arange(0, 1_000_000)
>>> Process().memory_info().rss
37810176
>>> view = arr[:500_000]
>>> Process().memory_info().rss
37810176
```

The `view` object looks like a 500,000-long array of `int64`, and so if it were a new array it would have allocated about 4MB of memory. But it’s just a view into the same original array, so no additional memory is needed.

> Technically a tiny bit of memory might be allocated for the view object itself, but that’s negligible unless you have lots of view objects. In this case, no new memory showed up in the RSS (resident memory) measure because Python pre-allocates larger chunks of memory, and then fills those chunks with small Python objects.

Leaking memory with views

One consequence of using views is that you might leak memory, rather than saving memory. This is because views keep the original array from being garbage collected—the *whole* array.

Let’s say you’ve decided you only need to use a small chunk of a large array:

```
>>> import numpy as np
>>> from psutil import Process
>>> arr = np.arange(0, 100_000_000)
>>> Process().memory_info().rss
830181376
>>> small_slice = arr[:10]
>>> del arr
>>> Process().memory_info().rss
830111744
```

If this were a Python list, deleting the original object would free the memory. In this case, however, even though we don't have a direct reference to the array, the view still does, which means the memory isn't freed, even though we're only interested in a tiny part of it.

You can actually access the original array via the `view`:

```
>>> small_slice
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> small_slice.base
array([0, 1, 2, ..., 99999997, 99999998, 99999999])
```

As a result, only once we delete all views does the original array's memory get freed:

```
>>> del small_slice
>>> Process().memory_info().rss
29642752
```

Unexpected mutation

Another consequence of views is that modifying a view will change the original array. Recall that for Python lists, modifying a sliced result doesn't modify the original list, because the new object is a copy:

```
>>> l = [1, 2, 3]
>>> l2 = l[:]
>>> l2[0] = 17
>>> l2
[17, 2, 3]
>>> l
[1, 2, 3]
```

With NumPy views, mutating the view *does* mutate the original object, they're both pointing at the same memory:

```
>>> arr = np.array([1, 2, 3])
>>> view = arr[:]
>>> view[0] = 17
>>> view
array([17,  2,  3])
>>> arr
array([17,  2,  3])
```

This result might not be what you wanted!

Unexpected mutation is made more likely by the fact that some NumPy APIs may return either views or copies, depending on circumstances. For example, some slicing results might not be views:

```
>>> arr = np.array([1, 2, 3])
>>> arr2 = arr[:]
>>> arr2.base is arr
True
>>> arr3 = arr[[True, False, True]]
>>> arr3.base is arr
False
```

Mutating `arr2` would mutate `arr` as well, but mutating `arr3` would not mutate `arr`.

Explicit copying with `copy()`

When you don't want to refer to the original memory, explicit copying allows you to create a new array. This can be useful to prevent mutation, and also in the case where you don't want to keep the original array around in memory:

```
>>> arr = np.arange(0, 100_000_000)
>>> Process().memory_info().rss
829464576
>>> small_slice = arr[:10].copy()
>>> del arr
>>> Process().memory_info().rss
29700096
>>> print(small_slice.base)
None
```

In this case deleting `arr` freed up the memory, because `small_slice` is a copy, not a view.

Takeaways: using views efficiently and safely

Given views are returned automatically by various NumPy APIs, you need to think of them as you write your code:

1. Pay attention in the documentation whether an API will return views, copies, or both.
2. If you have a large array you want to clear from memory, ensure that not only are there no direct references to it, but also there are no views referring to it.
3. If you are going to mutate an array, ensure it won't accidentally mutate some other array due to actually being a view.
4. If you don't want a view, use the `copy()` method.