NumPy As Strided Primer

John Chang

2020
December

Chapter 1

NumPy As Strided

1.1 Motivation

The reason why learning `as_strided` is important is because of speed.

Take for example, if you want to do a 2×2 custom convolution on a 10×10 area, that is, you want to get all possible 2×2 windows to do a function on, it's efficient to do with `as_strided`.

The reason why it's efficient is because `as_strided` gets the values as a reference / view, so performing a **read-only** operation on them is very quick.

1.2 Introduction

As expected of NumPy, when we do operations with their arrays, operations are light-speed fast, this is achieved by careful and memory safe operations. The assumption that the arrays are continuous is essential for fast memory addressing and retrieval.

Take for example, if we want to place 3 numbers next to each other in memory.

$$(10, 20, 30)$$

If they are represented by 1 Byte, in an unsigned fashion, we can achieve a range of $[0, 2^8 - 1] = [0, 255]$. Assume that in this context, they only need to be Byte aligned.

This makes it really easy for NumPy to dispatch a data retrieval. If they are placed together, we can see that the data is simply the following in base 16.

$$(0A_{16}, 14_{16}, 1E_{16})$$

NumPy only then needs to know the following:

- Where does this address start?
- How large is each object?
- How many objects are there?

We're not too concerned about where addresses start since they are usually handled in the back-end, however, the last 2 questions is something we can redefine.

1.3 1D Striding

Redefining strides is done using `np.lib.stride_tricks.as_stride`

There are 2 main arguments we're concerned about.

- Stride
- Shape

Data is **always** stored as a 1-Dimensional array in memory, we'll not go into details on memory retrieval.

1.3.1 Stride By 1

Take for example, the following memory chunk, $n \in [0, 255]$

```
VALUE 000 010 020 030 040 050 100 110 120 130 140 150
----- -----------------------------------------------
BYTE  005 006 007 008 009 010 011 012 013 014 015 016
```

If we would want **all values**, we'd stride by a single byte, that is, we'd retrieve `005, 006, 007, ...`. It's analogous to the step size, take a single byte-sized step every time.

This can be done in the following code. Note that this is not complete as we're missing the `shape` argument but it will still run.

```
arr = np.append(np.arange(0, 60, 10, dtype=np.uint8),
                np.arange(100, 160, 10, dtype=np.uint8))
np.lib.stride_tricks.as_strided(arr, strides=(1,))
```

A few things to note:

1. `np.uint8` makes each value 1 byte large, what happens if you don't use this?
2. `strides` must be a `Iterable`, that's why it's `(1,)` and not `(1)`

What do you expect as the output? Logically it's going to be the same array, and it is.

```
>> array([  0,  10,  20,  30,  40,  50, 100, 110, 120, 130, 140, 150],
       dtype=uint8)
```

1.3.2 Striding By N

It should be intuitive what happens when you stride by N but let's go through to sanity check.

```
VALUE 000 010 020 030 040 050 100 110 120 130 140 150
----- -----------------------------------------------
BYTE  005 006 007 008 009 010 011 012 013 014 015 016
       +       +       +       +       +       +
      ptr
```

It's important to note that when you create an array, there's a pointer `ptr` that points to where the array starts. So when you do strides, it's relative to that point.

```
arr = np.append(np.arange(0, 60, 10, dtype=np.uint8),
                np.arange(100, 160, 10, dtype=np.uint8))

np.lib.stride_tricks.as_strided(arr, strides=(2,))
```

```
>> array([  0,  20,  40, 100, 120, 140,  28,   0,   0,   0,   0,   0],
        dtype=uint8)
```

Interestingly enough, you may not even get the same result as I did, what happened here was an **Out of Bounds** memory access.

Remember that you have an argument called `shape`, that shape is implicitly passed and the algorithm assumes that you require the **same shape**. That's why it went beyond and out of bounds.

As warned in the NumPy Documentation on as_strided, accessing Out of Bounds is fine, but writing on it may corrupt memory, for this practise `writeable` by default, is `False`, therefore you shouldn't run into any corruption problems but it's good to note.

```
VALUE 000 010 020 030 040 050 100 110 120 130 140 150 ??? ??? ???
----- ----------------------------------------------------------- ...
BYTE  005 006 007 008 009 010 011 012 013 014 015 016 017 018 019
       +       +       +       +       +       +       +       +
```

1.3.3 Shape

Shape is the bounds of where your strides should end at.

By halving the shape and rounding it, we can limit where the stride should end. This requires knowledge of the shape, so we can just grab the `shape` property.

```
np.lib.stride_tricks.as_strided(arr, strides=(2,), shape=(a.shape[0]//2,))
```

Note the extra comma to cast it as a `Iterable`.

```
>> array([  0,  20,  40, 100, 120, 140, 160], dtype=uint8)
```

1.4 2D Striding

This may be more confusing as we're going to wrap a 1D memory as a 2D object

Take for example the following

```
VALUE 000 001 002 010 011 012 020 021 022 030 031 032
----- -----------------------------------------------
BYTE  001 002 003 004 005 006 007 008 009 010 011 012
```

The end result we want is to wrap it such that we have

$$\begin{bmatrix} 0 & 1 & 2 \\ 10 & 11 & 12 \\ 20 & 21 & 22 \\ 30 & 31 & 32 \end{bmatrix}$$

In Python, it'll be shown as

```
[[0, 1, 2], [10, 11, 12], [20, 21, 22], [30, 31, 32]]
```

Let's go by logic on how you would construct this matrix when given the 1D representation

1.4.1 Understanding Multidimensional Striding

Since you know that each row has 3 values, we take the first 3 values

```
VALUE 000 001 002
----- -----------
BYTE  001 002 003
```

Then file them under R_0.

```
VALUE 010 011 012
----- -----------
BYTE  004 005 006
```

Take the next, file them under R_1.

```
VALUE 000 001 002 | 010 011 012 | 020 021 022 | 030 031 032
----- ----------- | ----------- | ----------- | -----------
BYTE  001 002 003 | 004 005 006 | 007 008 009 | 010 011 012
          R_0           R_1           R_2           R_3
```

You get the following separation.

Note the properties that you've taken.

- For Each Row
  - The step size is 1, that is we take every value
  - The shape is 3, that is we have 3 values each row
- For Each Column
  - The step size is 3, that is, we move 3 steps in memory for each value
  - The shape is 4, that is we have 4 values each column
- Stride: $(C = 3, R = 1)$
- Shape: $(C = 4, R = 3)$

It might be confusing why we move 3 steps in memory for columns. Let's look at the following illustration, this is **important** to understand.

```
VALUE 000 001 002 010 011 012 020 021 022 030 031 032
----- -----------------------------------------------
BYTE  001 002 003 004 005 006 007 008 009 010 011 012
DIM 0  0   1   2   0   1   2   0   1   2   0   1   2
DIM 1  0           1           2           3
```

Figure 1.1: Dimensional Mapping Diagram

$$
\begin{bmatrix}
001 & \xrightarrow{1B} & 002 & \xrightarrow{1B} & 003 \\
3B \downarrow & & & & \\
004 & \xrightarrow{1B} & 005 & \xrightarrow{1B} & 006 \\
3B \downarrow & & & & \\
007 & \xrightarrow{1B} & 008 & \xrightarrow{1B} & 009 \\
3B \downarrow & & & & \\
010 & \xrightarrow{1B} & 011 & \xrightarrow{1B} & 012
\end{bmatrix}
$$

Figure 1.2: Memory addresses of the selected elements

Here, the horizontal elements represent Dim 0, while the vertical elements represent Dim 1.

```
arr = np.asarray([0, 1, 2,
                  10, 11, 12,
                  20, 21, 22,
                  30, 31, 32], dtype=np.uint8)

np.lib.stride_tricks.as_strided(arr, strides=(3,1), shape=(4,3))

>> array([[ 0,  1,  2],
          [10, 11, 12],
          [20, 21, 22],
          [30, 31, 32]], dtype=uint8)
```

1.4.2 Skipping Values

This is not too interesting as it seems to be logical on how it works, but what happens if you stride a bit further?

```
VALUE 000 001 002 010 011 012 020 021 022 030 031 032 ??? ??? ???
----- -----------------------------------------------------------
BYTE  001 002 003 004 005 006 007 008 009 010 011 012 013 014 015
DIM 0  0   1   2       0   1   2       0   1   2       0   1   2
DIM 1  0               1               2               2
```

$$
\begin{bmatrix}
001 & \xrightarrow{1B} & 002 & \xrightarrow{1B} & 003 \\
4B \downarrow & & & & \\
005 & \xrightarrow{1B} & 006 & \xrightarrow{1B} & 007 \\
4B \downarrow & & & & \\
009 & \xrightarrow{1B} & 010 & \xrightarrow{1B} & 011 \\
4B \downarrow & & & & \\
013 & \xrightarrow{1B} & 014 & \xrightarrow{1B} & 015
\end{bmatrix}
$$

Figure 1.3: Memory addresses of the selected elements

Similarly, horizontal is Dim 0, vertical is Dim 1.

Note how $013, 014, 015$ are all undefined in this context!

```
np.lib.stride_tricks.as_strided(arr, strides=(4,1), shape=(4,3))

>> array([[ 0,  1,  2],
          [11, 12, 20],
          [22, 30, 31],
          [28,  2,  0]], dtype=uint8)
```

As expected, we get random values at the end, however, we managed to **skip** values during striding.

1.4.3 Overlapping Values

What happens if you do too short of a stride?

```
VALUE 000 001 002 010 011 012 020 021 022 030 031 032
----- -----------------------------------------------
BYTE  001 002 003 004 005 006 007 008 009 010 011 012
DIM 0  0   1   2       0   1   2
DIM 0          0   1   2       0   1   2
DIM 1  0       1       2       3
```

$$\begin{bmatrix} 001 & \xrightarrow{1B} & 002 & \xrightarrow{1B} & 003 \\ 2B\downarrow & & & & \\ 003 & \xrightarrow{1B} & 004 & \xrightarrow{1B} & 005 \\ 2B\downarrow & & & & \\ 005 & \xrightarrow{1B} & 006 & \xrightarrow{1B} & 007 \\ 2B\downarrow & & & & \\ 007 & \xrightarrow{1B} & 008 & \xrightarrow{1B} & 009 \end{bmatrix}$$

Figure 1.4: Memory addresses of the selected elements

Similarly, horizontal is Dim 0, vertical is Dim 1.

Note how we can repeat the addresses, this is not dangerous as read-only data.

```
np.lib.stride_tricks.as_strided(arr, strides=(2,1), shape=(4,3))

>> array([[ 0,  1,  2],
          [ 2, 10, 11],
          [11, 12, 20],
          [20, 21, 22]], dtype=uint8)
```

Interestingly enough, we got overlapping values, we can take it 1 step further by doing a `(1,1)` stride.

```
VALUE 000 001 002 010 011 012 020 021 022 030 031 032
----- -----------------------------------------------
BYTE  001 002 003 004 005 006 007 008 009 010 011 012
DIM 0  0   1   2   0   1   2
DIM 0      0   1   2
DIM 0          0   1   2
DIM 1  0   1   2   3
```

$$\begin{bmatrix} 001 & \xrightarrow{1B} & 002 & \xrightarrow{1B} & 003 \\ 1B\downarrow & & & & \\ 002 & \xrightarrow{1B} & 003 & \xrightarrow{1B} & 004 \\ 1B\downarrow & & & & \\ 003 & \xrightarrow{1B} & 004 & \xrightarrow{1B} & 005 \\ 1B\downarrow & & & & \\ 004 & \xrightarrow{1B} & 005 & \xrightarrow{1B} & 006 \end{bmatrix}$$

Figure 1.5: Memory addresses of the selected elements

Horizontal is Dim 0, Vertical is Dim 1.

```
np.lib.stride_tricks.as_strided(arr, strides=(1,1), shape=(4,3))

>> array([[ 0,  1,  2],
          [ 1,  2, 10],
          [ 2, 10, 11],
          [10, 11, 12]], dtype=uint8)
```

This is extremely useful if you want to create a sliding window.

Before we discuss about 3D Striding, let's take a look at the underlying representation of some arrays.

1.5 Creation of Arrays

When we create arrays, they are always 1-D under the fancy NumPy Interface.

To illustrate, when you do `np.asarray([[3,4],[7,8]], dtype=np.uint8)` you get the following in the memory.

```
VALUE  3   4   7   8
----- ---------------
BYTE   1   2   3   4
```

Note that the Byte Address doesn't necessarily have to start from 1, it's for simplicity

What NumPy does under the hood is that it creates specific strides and shapes such that it is interpreted as `[[3,4],[7,8]]`

```
arr = np.asarray([[3,4],[7,8]], dtype=np.uint8)
arr.strides
>> (2, 1)
arr.shape
>> (2, 2)
```

This is so that you don't have to deal with wrapping the array, and this depicts why NumPy is so fast in transformation such as `reshape()`

1.6 Value Size

Notice how I always declared values as `np.uint8`, this is because it's **exactly 1 Byte**. This simplifies the examples, let's take a look what happens if you have a `dtype` of `np.uint16`.

```
arr = np.asarray([[0, 1], [2, 3]], dtype=np.uint16)

arr.strides
>> (4, 2)

arr.shape
>> (2, 2)
```

Compared to `np.uint8`

```
arr = np.asarray([[0, 1], [2, 3]], dtype=np.uint8)

arr.strides
>> (2, 1)

arr.shape
>> (2, 2)
```

Notice how the stride is doubled, this is because `np.uint16` is double the size of `np.uint8`.

np.uint8 memory storage

```
VALUE  0   1   2   3
----- ---------------
BYTE   1   2   3   4
```

np.uint16 memory storage

```
VALUE  0       1       2       3
----- ------------------------------
BYTE   1   2   3   4   5   6   7   8
```

Since it takes 2 bytes, the stride must be larger. This difference is important when defining our `as_strided` as it doesn't factor in the size of each value.

Stride too small or oddly and you'll get weird values

Too small of a Stride

```
arr = np.asarray([[0, 1], [2, 3]], dtype=np.uint16)
np.lib.stride_tricks.as_strided(arr, strides=(2, 1), shape=(2, 2))

>> array([[  0, 256],
          [  1, 512]], dtype=uint16)
```

```
VALUE  0       1       2       3
----- ------------------------------
BYTE   1   2   3   4   5   6   7   8
DIM 0  0   1   0   1
DIM 1  0       1
```

$$\begin{bmatrix} 1 & \xrightarrow{1B} & 2 \\ 2B\downarrow & & \\ 3 & \xrightarrow{1B} & 4 \end{bmatrix}$$

Too large of a Stride

```
arr = np.asarray([[0, 1], [2, 3]], dtype=np.uint16)
np.lib.stride_tricks.as_strided(arr, strides=(3, 1), shape=(2, 2))

>> array([[  0, 256],
          [512,   2]], dtype=uint16)
```

```
VALUE  0       1       2       3
----- ------------------------------
BYTE   1   2   3   4   5   6   7   8
DIM 0  0   1       0   1
DIM 1  0           1
```

$$\begin{bmatrix} 1 & \xrightarrow{1B} & 2 \\ 3B\downarrow & & \\ 4 & \xrightarrow{1B} & 5 \end{bmatrix}$$

1.6.1 Getting Value Sizes

To get sizes of each `dtype`, call `ar.itemsize` for the size in **Bytes**. To ensure that your strides is correctly factored, we just multiply every value with the item size.

```
arr = np.asarray([[0, 1], [2, 3]], dtype=np.uint16)
np.lib.stride_tricks.as_strided(arr,
                                strides=arr.itemsize * np.asarray([2, 1]),
                                shape=(2, 2))
np.lib.stride_tricks.as_strided(arr,
                                strides=[4, 2], shape=(2, 2))
```

The last 2 function calls are equivalent

1.7 3D Striding

To avoid clutter, we'll omit the Byte row, assume it's the same as the value

```
VALUE 000 001 002 003 004 005 ... 020 021 022 023
```

Let's say we want to wrap this in a 2x3x4 cube matrix.

$$\begin{bmatrix}0 & 1 & 2\\ 3 & 4 & 5\end{bmatrix}\begin{bmatrix}6 & 7 & 8\\ 9 & 10 & 11\end{bmatrix}\begin{bmatrix}12 & 13 & 14\\ 15 & 16 & 17\end{bmatrix}\begin{bmatrix}18 & 19 & 20\\ 21 & 22 & 23\end{bmatrix}$$

$$\begin{bmatrix}0 & \xrightarrow{1B} & 1 & \xrightarrow{1B} & 2\\ 3B\downarrow & & & & \\ 3 & \xrightarrow{1B} & 4 & \xrightarrow{1B} & 5\end{bmatrix} \xrightarrow{6B} \begin{bmatrix}6 & \xrightarrow{1B} & 7 & \xrightarrow{1B} & 8\\ 3B\downarrow & & & & \\ 9 & \xrightarrow{1B} & 10 & \xrightarrow{1B} & 11\end{bmatrix}$$
$$\xrightarrow{6B} \begin{bmatrix}12 & \xrightarrow{1B} & 13 & \xrightarrow{1B} & 14\\ 3B\downarrow & & & & \\ 15 & \xrightarrow{1B} & 16 & \xrightarrow{1B} & 17\end{bmatrix} \xrightarrow{6B} \begin{bmatrix}18 & \xrightarrow{1B} & 19 & \xrightarrow{1B} & 20\\ 3B\downarrow & & & & \\ 21 & \xrightarrow{1B} & 22 & \xrightarrow{1B} & 23\end{bmatrix}$$

$$D_0 : (Stride = 1, Shape = 3)$$
$$D_1 : (Stride = 3, Shape = 2)$$
$$D_2 : (Stride = 6, Shape = 4)$$

```
arr = np.arange(0, 24, dtype=np.uint8)
np.lib.stride_tricks.as_strided(arr, strides=(6, 3, 1), shape=(4, 2, 3))

>> array([[[ 0,  1,  2],
           [ 3,  4,  5]],
          [[ 6,  7,  8],
           [ 9, 10, 11]],
          [[12, 13, 14],
           [15, 16, 17]],
          [[18, 19, 20],
           [21, 22, 23]]], dtype=uint8)
```

1.7.1 Overlapping Windows on 1D

Let's say we want overlapping windows

$$\begin{bmatrix}0 & 1 & 2\\ 3 & 4 & 5\end{bmatrix} \begin{bmatrix}3 & 4 & 5\\ 6 & 7 & 8\end{bmatrix} \begin{bmatrix}6 & 7 & 8\\ 9 & 10 & 11\end{bmatrix} \begin{bmatrix}9 & 10 & 11\\ 12 & 13 & 14\end{bmatrix} \begin{bmatrix}12 & 13 & 14\\ 15 & 16 & 17\end{bmatrix} \begin{bmatrix}15 & 16 & 17\\ 18 & 19 & 20\end{bmatrix} \begin{bmatrix}18 & 19 & 20\\ 21 & 22 & 23\end{bmatrix}$$

$$\begin{bmatrix}0 & \xrightarrow{1B} & 1 & \xrightarrow{1B} & 2\\ 3B\downarrow & & & & \\ 3 & \xrightarrow{1B} & 4 & \xrightarrow{1B} & 5\end{bmatrix} \xrightarrow{3B} \begin{bmatrix}3 & \xrightarrow{1B} & 4 & \xrightarrow{1B} & 5\\ 3B\downarrow & & & & \\ 6 & \xrightarrow{1B} & 7 & \xrightarrow{1B} & 8\end{bmatrix}$$
$$\xrightarrow{3B} \begin{bmatrix}6 & \xrightarrow{1B} & 7 & \xrightarrow{1B} & 8\\ 3B\downarrow & & & & \\ 9 & \xrightarrow{1B} & 10 & \xrightarrow{1B} & 11\end{bmatrix} \xrightarrow{3B} \begin{bmatrix}9 & \xrightarrow{1B} & 10 & \xrightarrow{1B} & 11\\ 3B\downarrow & & & & \\ 12 & \xrightarrow{1B} & 13 & \xrightarrow{1B} & 14\end{bmatrix}$$
$$\xrightarrow{3B} \begin{bmatrix}12 & \xrightarrow{1B} & 13 & \xrightarrow{1B} & 14\\ 3B\downarrow & & & & \\ 15 & \xrightarrow{1B} & 16 & \xrightarrow{1B} & 17\end{bmatrix} \xrightarrow{3B} \begin{bmatrix}15 & \xrightarrow{1B} & 16 & \xrightarrow{1B} & 17\\ 3B\downarrow & & & & \\ 18 & \xrightarrow{1B} & 19 & \xrightarrow{1B} & 20\end{bmatrix}$$
$$\xrightarrow{3B} \begin{bmatrix}18 & \xrightarrow{1B} & 19 & \xrightarrow{1B} & 20\\ 3B\downarrow & & & & \\ 21 & \xrightarrow{1B} & 22 & \xrightarrow{1B} & 23\end{bmatrix}$$

$$\begin{aligned} D_0 &: (Stride = 1, Shape = 3)\\ D_1 &: (Stride = 3, Shape = 2)\\ D_2 &: (Stride = 3, Shape = 7) \end{aligned}$$

```
arr = np.arange(0, 24, dtype=np.uint8)
np.lib.stride_tricks.as_strided(arr, strides=(3, 3, 1), shape=(7, 2, 3))

>> array([[[ 0,  1,  2],
           [ 3,  4,  5]],
          [[ 3,  4,  5],
           [ 6,  7,  8]],
          [[ 6,  7,  8],
           [ 9, 10, 11]],
          [[ 9, 10, 11],
           [12, 13, 14]],
          [[12, 13, 14],
           [15, 16, 17]],
          [[15, 16, 17],
           [18, 19, 20]],
          [[18, 19, 20],
           [21, 22, 23]]], dtype=uint8)
```

1.7.2 Overlapping Windows on 2D

Let's try another to understand better. This is much more complicated, but if we do the diagram, it's slightly clearer.

$$
\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}
\begin{bmatrix} 3 & 4 & 5 \\ 4 & 5 & 6 \\ 5 & 6 & 7 \\ 6 & 7 & 8 \end{bmatrix}
[\ldots]
\begin{bmatrix} 15 & 16 & 17 \\ 16 & 17 & 18 \\ 17 & 18 & 19 \\ 18 & 19 & 20 \end{bmatrix}
\begin{bmatrix} 18 & 19 & 20 \\ 19 & 20 & 21 \\ 20 & 21 & 22 \\ 21 & 22 & 23 \end{bmatrix}
$$

$$
\begin{bmatrix}
0 & \xrightarrow{1B} & 1 & \xrightarrow{1B} & 2 \\
1B\downarrow & & & & \\
1 & \xrightarrow{1B} & 2 & \xrightarrow{1B} & 3 \\
1B\downarrow & & & & \\
2 & \xrightarrow{1B} & 3 & \xrightarrow{1B} & 4 \\
1B\downarrow & & & & \\
3 & \xrightarrow{1B} & 4 & \xrightarrow{1B} & 5
\end{bmatrix}
\xrightarrow{3B}
\begin{bmatrix}
3 & \xrightarrow{1B} & 4 & \xrightarrow{1B} & 5 \\
1B\downarrow & & & & \\
4 & \xrightarrow{1B} & 5 & \xrightarrow{1B} & 6 \\
1B\downarrow & & & & \\
5 & \xrightarrow{1B} & 6 & \xrightarrow{1B} & 7 \\
1B\downarrow & & & & \\
6 & \xrightarrow{1B} & 7 & \xrightarrow{1B} & 8
\end{bmatrix}
\xrightarrow{3B} \ldots
$$

$$
\ldots \xrightarrow{3B}
\begin{bmatrix}
15 & \xrightarrow{1B} & 16 & \xrightarrow{1B} & 17 \\
1B\downarrow & & & & \\
16 & \xrightarrow{1B} & 17 & \xrightarrow{1B} & 18 \\
1B\downarrow & & & & \\
17 & \xrightarrow{1B} & 18 & \xrightarrow{1B} & 19 \\
1B\downarrow & & & & \\
18 & \xrightarrow{1B} & 19 & \xrightarrow{1B} & 20
\end{bmatrix}
\xrightarrow{3B}
\begin{bmatrix}
18 & \xrightarrow{1B} & 19 & \xrightarrow{1B} & 20 \\
1B\downarrow & & & & \\
19 & \xrightarrow{1B} & 20 & \xrightarrow{1B} & 21 \\
1B\downarrow & & & & \\
20 & \xrightarrow{1B} & 21 & \xrightarrow{1B} & 22 \\
1B\downarrow & & & & \\
21 & \xrightarrow{1B} & 22 & \xrightarrow{1B} & 23
\end{bmatrix}
$$

$$
\begin{aligned}
D_0 &: (Stride = 1, Shape = 3) \\
D_1 &: (Stride = 1, Shape = 4) \\
D_2 &: (Stride = 3, Shape = 7)
\end{aligned}
$$

```
arr = np.arange(0, 24, dtype=np.uint8)
np.lib.stride_tricks.as_strided(arr, strides=(3, 1, 1), shape=(7, 4, 3))

>> array([[[ 0,  1,  2],
           [ 1,  2,  3],
           [ 2,  3,  4],
           [ 3,  4,  5]],
          [[ 3,  4,  5],
           [ 4,  5,  6],
           [ 5,  6,  7],
           [ 6,  7,  8]],
           ...
          [[15, 16, 17],
           [16, 17, 18],
           [17, 18, 19],
           [18, 19, 20]],
          [[18, 19, 20],
           [19, 20, 21],
           [20, 21, 22],
           [21, 22, 23]]], dtype=uint8)
```

As long as you use the template, I don't think you should run into any issues, however, you're free to reinterpret it.

1.8 References

(Stack Overflow) How can I simply calculate the rolling/moving variance of a time series in python?

(Stack Overflow) How to understand numpy strides for layman? (1)

(Stack Overflow) How to understand numpy strides for layman? (2)