Numpy : Lecture 4

Broadcasting

- **Broadcasting**. The term **broadcasting** describes how **numpy** treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is "**broadcast**" across the larger array so that they have compatible shapes.
- Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

Alternative to Broadcasting

- For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

- import numpy **as** np
-
- *# We will add the vector v to each row of the matrix x,*
- *# storing the result in the matrix y*
- x **=** np**.**array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
- v **=** np**.**array([1, 0, 1])
- y **=** np**.**empty_like(x) *# 4,3 Create an empty matrix with the same shape as x*
-

Alternative to Broadcasting

- *# Add the vector v to each row of the matrix x with an explicit loop*
- **for** i **in** range(4):
-     y[i, :] **=** x[i, :] **+** v
-
- *# Now y is the following*
- *# [[2 2 4]*
- *# [5 5 7]*
- *# [8 8 10]*
- *# [11 11 13]]*
- **print**(y)

Alternative to Broadcasting

- This works; however when the matrix x is very large, computing an explicit loop in Python could be slow.
- Note that adding the vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically.
- Then performing elementwise summation of x and vv. We could implement this approach like this:

Using Tile

- import numpy **as** np
- *# We will add the vector v to each row of the matrix x,*
- *# storing the result in the matrix y*
- x **=** np**.**array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
- v **=** np**.**array([1, 0, 1])
- vv **=** np**.**tile(v, (4, 1)) *# Stack 4 copies of v on top of each other*
- **print**(vv) *# Prints "[[1 0 1]*
- *# [1 0 1]*
- *# [1 0 1]*
- *# [1 0 1]]"*
- y **=** x **+** vv *# Add x and vv elementwise*
- **print**(y) *# Prints "[[2 2 4*
- *# [5 5 7]*
- *# [8 8 10]*
- *# [11 11 13]]"*

Broadcasting

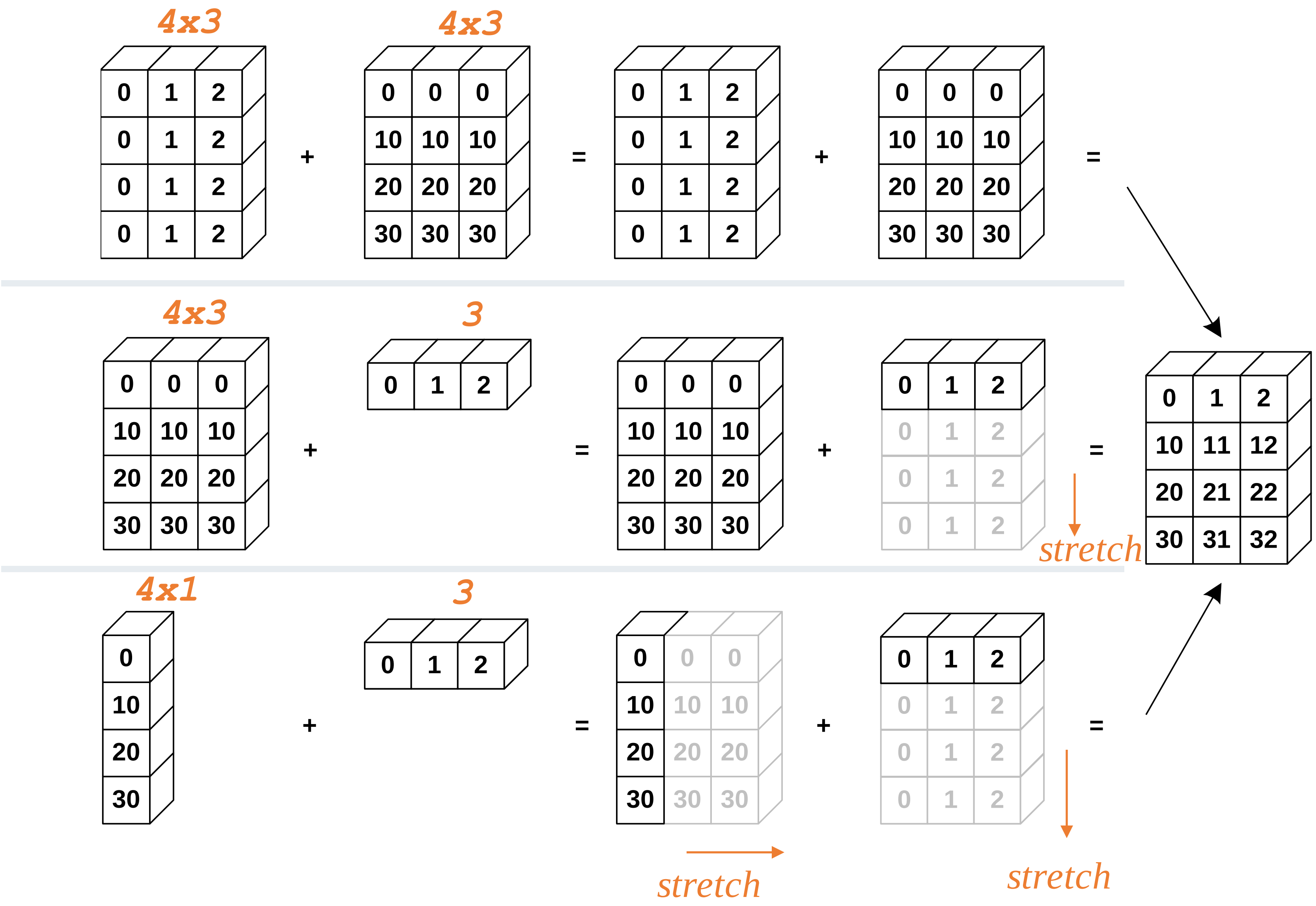
- Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. Consider this version, using broadcasting:
- import numpy **as** np
-
- *# We will add the vector v to each row of the matrix x,*
- *# storing the result in the matrix y*
- x **=** np**.**array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
- v = np**.**array([1, 0, 1])
- y **=** x **+** v *# Add v to each row of x using broadcasting*
- **print**(y) *# Prints "[[2 2 4]*
-           *#   [5 5 7]*
-           *#   [8 8 10]*
-           *#   [11 11 13]]"*

Broadcasting

- The line y = x + v works even though x has shape (4, 3) and v has shape (3,) due to broadcasting; this line works as if v actually had shape (4, 3), where each row was a copy of v, and the sum was performed elementwise.

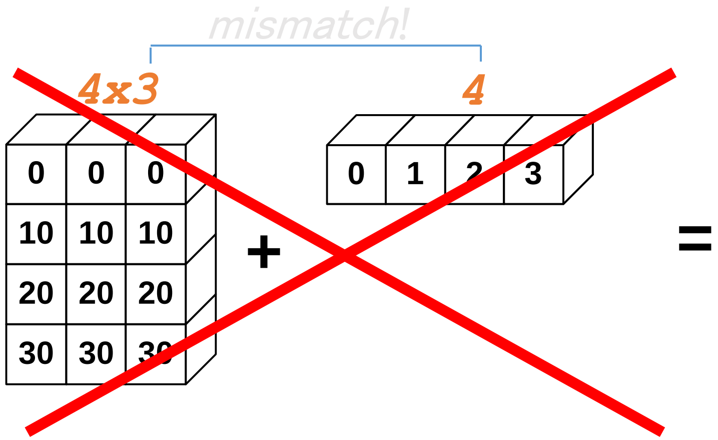
- Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

Array Broadcasting



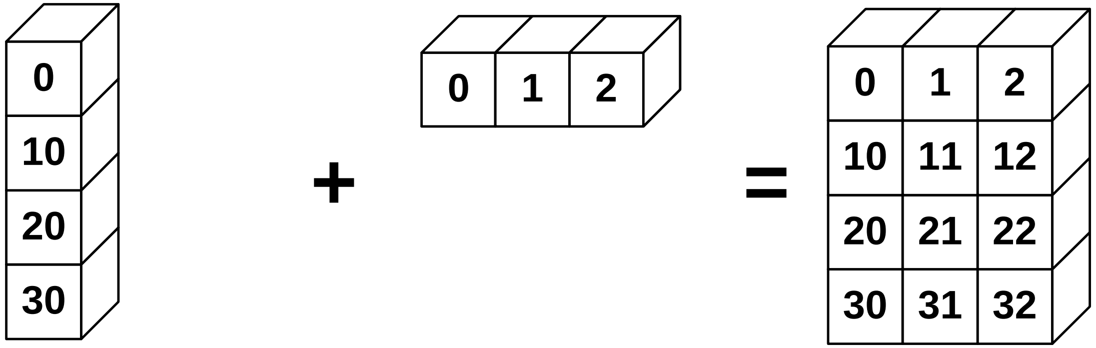
Broadcasting Rules

The *trailing* axes of both arrays must either be 1 or have the same size for broadcasting to occur. Otherwise, a `"ValueError: frames are not aligned"` exception is thrown.



Broadcasting in Action

```
>>> a = array((0,10,20,30))
>>> b = array((0,1,2))
>>> y = a[:, None] + b
```



Broadcasting

- Broadcasting two arrays together follows these rules:
- If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
- The two arrays are said to be *compatible* in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
- The arrays can be broadcast together if they are compatible in all dimensions.
- After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
- In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

- import numpy **as** np
- *# Compute outer product of vectors*
- v **=** np**.**array([1,2,3]) *# v has shape (3,)*
- w **=** np**.**array([4,5])   *# w has shape (2,)*
- *# To compute an outer product, we first reshape v to be a column vector of shape (3, 1);*
- *we can then broadcast it against w to yield an output of shape (3, 2), which is the outer product of v and w:*
- *# [[4 5]   [8 10] [12 15]]*
- **print**(np**.**reshape(v, (3, 1)) ***** w)
- *# Add a vector to each row of a matrix*
- x **=** np**.**array([[1,2,3], [4,5,6]])
- *# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),*
- *# giving the following matrix:*
- *# [[2 4 6]*
- *# [5 7 9]]*
- **print**(x **+** v)
- *# Add a vector to each column of a matrix*
- *# x has shape (2, 3) and w has shape (2,).*
- *# If we transpose x then it has shape (3, 2) and can be broadcast*

- *# against w to yield a result of shape (3, 2); transposing this result*
- *# yields the final result of shape (2, 3) which is the matrix x with*
- *# the vector w added to each column. Gives the following matrix:*
- *# [[5 6 7]*
- *# [9 10 11]]*
- **print**((x**.**T **+** w)**.**T)
- *# Another solution is to reshape w to be a column vector of shape (2, 1);*
- *# we can then broadcast it directly against x to produce the same*
- *# output.*
- **print**(x **+** np**.**reshape(w, (2, 1)))
-
- *# Multiply a matrix by a constant:*
- *# x has shape (2, 3). Numpy treats scalars as arrays of shape ();*
- *# these can be broadcast together to shape (2, 3), producing the*
- *# following array:*
- *# [[2 4 6]*
- *# [8 10 12]]*
- **print**(x ***** 2)

Broadcasting

When there are multiple inputs, then they all must be "broadcastable" to the same shape.

- All arrays are promoted to the same number of dimensions (by pre-prending 1's to the shape)
- All dimensions of length 1 are expanded as determined by other inputs with non-unit lengths in that dimension.

```
x = [1,2,3,4];
y=[[10],[20],[30]]
print N.add(x,y)
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
x = array(x)
y = array(y)
print x+y
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

x has shape (4,) the ufunc sees it as having shape (1,4)

y has shape (3,1)

The ufunc result has shape (3,4)