

CSC 561: Neural Networks and Deep Learning

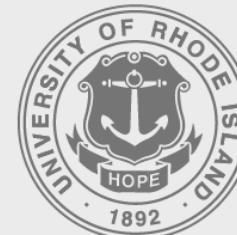
Preliminaries

Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2024

THINK BIG WE DOSM



Linear Algebra

Subset of most relevant concepts to Deep Learning

Scalars and vectors

- Scalars

- ✓ integers, real numbers, rational numbers, etc.
- ✓ usually denoted by lowercase letters

- Vectors

- ✓ 1-D array of elements (scalars)

$$\mathbf{x} \in \mathbb{R}^n$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Matrices and tensors

- Matrices
 - ✓ 2-D array of elements

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \quad A \in \mathbb{R}^{m \times n}$$

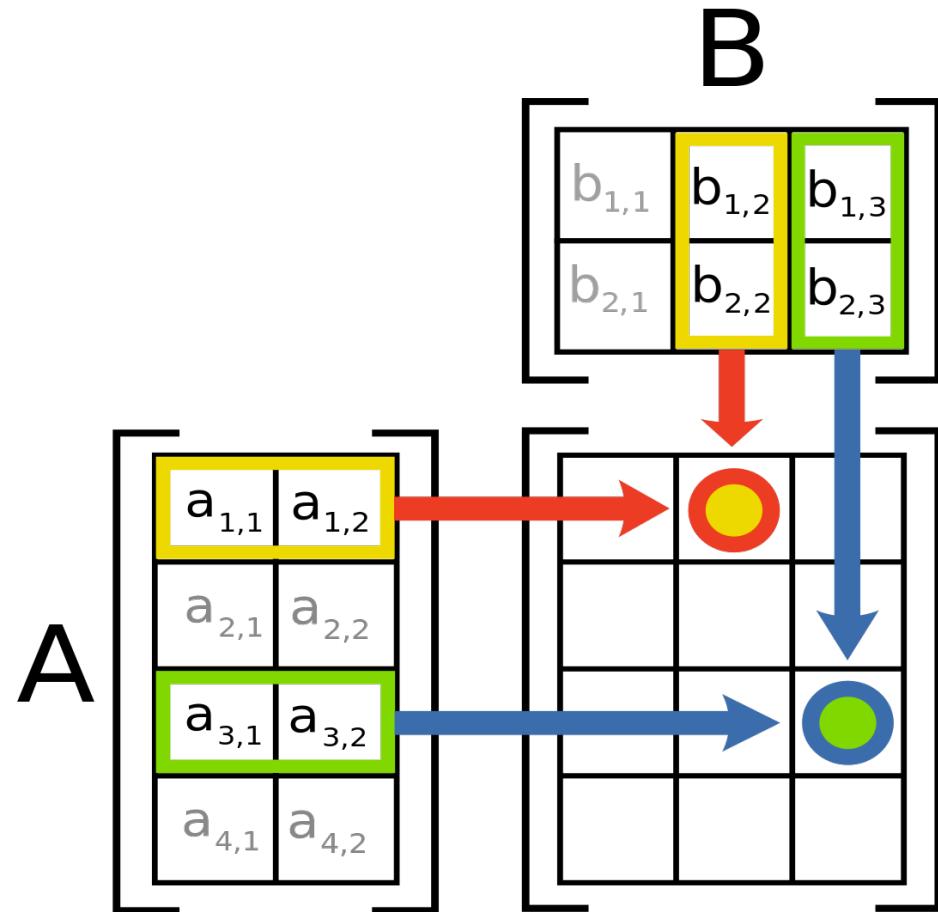
- Tensors
 - ✓ homogeneous arrays that may have **zero (scalar)** or **more dimensions**

Basics

Matrix Multiplication

$$C = AB$$

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$



number of columns in A must be equal to the number of rows in B

Dot product

$$[x_1 \quad x_2 \quad \dots \quad x_n] \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$$\mathbf{x}^T \mathbf{y} = \sum_{i=1}^n x_i y_i$$

$$\mathbf{x}^T \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \cos \theta$$

$$\mathbf{x}^T \mathbf{y} \in \mathbb{R}$$

Matrix transpose

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix} \Rightarrow A^\top = \begin{bmatrix} A_{1,1} & A_{2,1} & A_{3,1} \\ A_{1,2} & A_{2,2} & A_{3,2} \end{bmatrix}$$

$$(A^T)_{i,j} = A_{j,i} \qquad (AB)^T = B^T A^T$$

Identity matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\forall x \in \mathbb{R}^n, I_n x = x$$

$$A^{-1} A = I_n$$

Norms

- Functions that measure the **magnitude** (“length”) of a vector
 - ✓ **strictly positive**, except for the zero vector
 - ✓ think about the distance between zero and the point represented by the vector

$$f(\mathbf{x}) = 0 \Rightarrow \mathbf{x} = \mathbf{0}$$

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}) \text{ (triangle inequality)}$$

$$\forall \alpha \in \mathbb{R}, f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x})$$

Norms

ℓ_1 -norm:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$$

ℓ_2 -norm:

$$\|\mathbf{x}\|_2 = \left(\sum_{i=1}^n x_i^2 \right)^{\frac{1}{2}}$$

max norm:

$$\|\mathbf{x}\|_\infty = \max_i |\mathbf{x}_i|$$

Special matrices and vectors

- Unit vector

$$\|\mathbf{x}\|_2 = 1$$

- Symmetric matrix

$$A = A^T$$

- Orthogonal matrix

$$A^T A = A A^T = I$$

$$A^{-1} = A^T$$

Programming with Tensors

Non-negotiable (Python)

- › Basic data types
 - ✓ booleans, integers, floating point values, strings
- › Control flow
- › Built-in data structures
 - ✓ lists, tuples, dictionaries, sets
 - ✓ iterators
- › Functions
 - ✓ functions can be assigned, passed, returned, and stored
 - ✓ lambda functions (inline and anonymous)
- › Classes

Numpy

- Library for scientific computing
 - ✓ provides a **high-performance** multidimensional array object and routines for fast operations on these arrays
- The **ndarray** object encapsulates n-dimensional arrays of homogeneous data types
 - ✓ many operations performed in **compiled code** for higher performance
 - ✓ have a fixed size at creation
 - changing the size of an ndarray will create a new array and delete the original

```
n = 100000000
array1 = np.random.rand(n)
array2 = np.random.rand(n)
```

```
def dot_python(x, y):
    sum = 0
    for i in range(len(x)):
        sum += x[i] * y[i]
    return sum
```

```
def dot_numpy(x, y):
    return np.dot(x, y)
```

```
time_taken = timeit.timeit(lambda: dot_numpy(array1, array2), number=1)
print(f'Numpy Time: {time_taken} seconds')
```

```
Numpy Time: 0.05994947799996453 seconds
```

```
array1 = array1.tolist()
array2 = array2.tolist()
time_taken = timeit.timeit(lambda: dot_python(array1, array2), number=1)
print(f'Python Time: {time_taken} seconds')
```

```
Python Time: 8.325287125999978 seconds
```

The following figures are from:

“NumPy Illustrated: The Visual Guide to NumPy” by Lev Maximov

<https://betterprogramming.pub/numpy-illustrated-the-visual-guide-to-numpy-3b1d4976de1d>

Elegant code

```
In [3]: a = [1, 2, 3]  
[q*2 for q in a]
```

```
Out[3]: [2, 4, 6]
```

```
In [4]: a = np.array([1, 2, 3])  
a * 2
```

```
Out[4]: array([2, 4, 6])
```

```
In [1]: a = [1, 2, 3]  
b = [4, 5, 6]  
[q+r for q, r in zip(a, b)]
```

```
Out[1]: [5, 7, 9]
```

```
In [2]: a = np.array([1, 2, 3])  
b = np.array([4, 5, 6])  
a + b
```

```
Out[2]: array([5, 7, 9])
```

Homogeneous and fixed-length

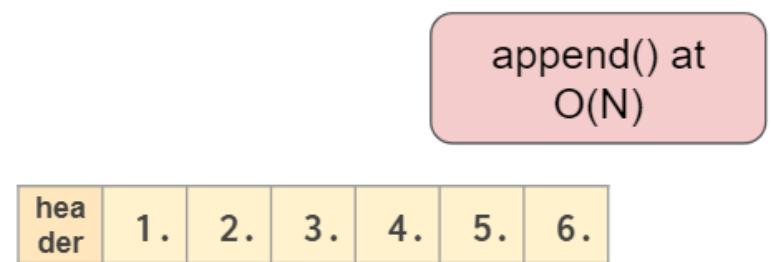
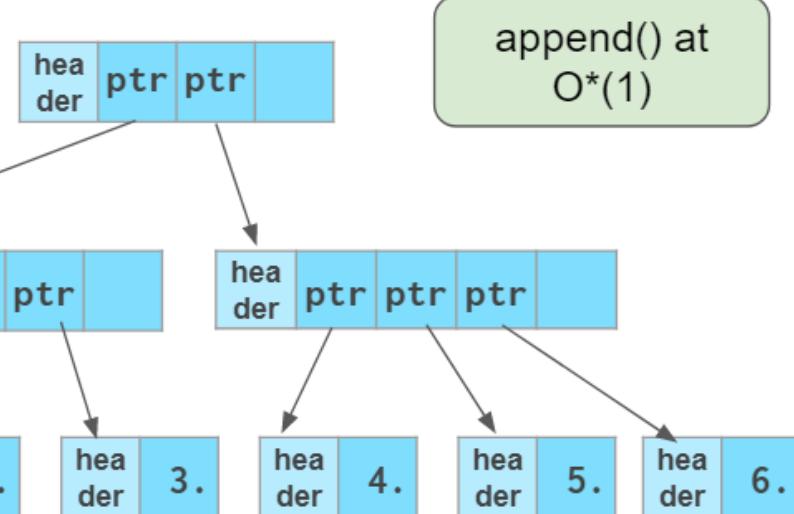
python list

vs

numpy array

1.	2.	3.
4.	5.	6.

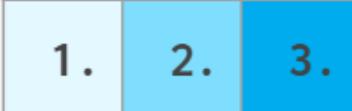
1.	2.	3.
4.	5.	6.



Creating numpy arrays

`a = np.array([1., 2., 3.])`



`a`

`.dtype == np.float64`
`.shape == (3,)`

`np.zeros(3)`





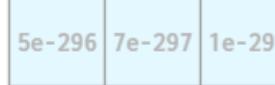
`np.ones(3)`





`np.empty(3)`





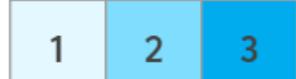
`np.full(3, 7.)`





`np.array([1, 2, 3])`





`np.zeros_like(a)`





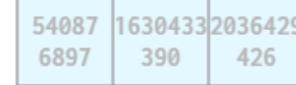
`np.ones_like(a)`





`np.empty_like(a)`





`np.full_like(a, 7)`





Creating numpy arrays

```
a = np.array([[1, 2, 3],  
             [4, 5, 6]])
```



1	2	3
4	5	6

.dtype == np.int32
.shape == (2, 3)

len(a) == a.shape[0]

```
np.zeros((3, 2))
```



0.	0.
0.	0.
0.	0.

```
np.full((3, 2), 7)
```



7	7
7	7
7	7

np.eye(3, 3)

1.	0.	0.
0.	1.	0.
0.	0.	1.

= np.eye(3)

```
np.ones((3, 2))
```

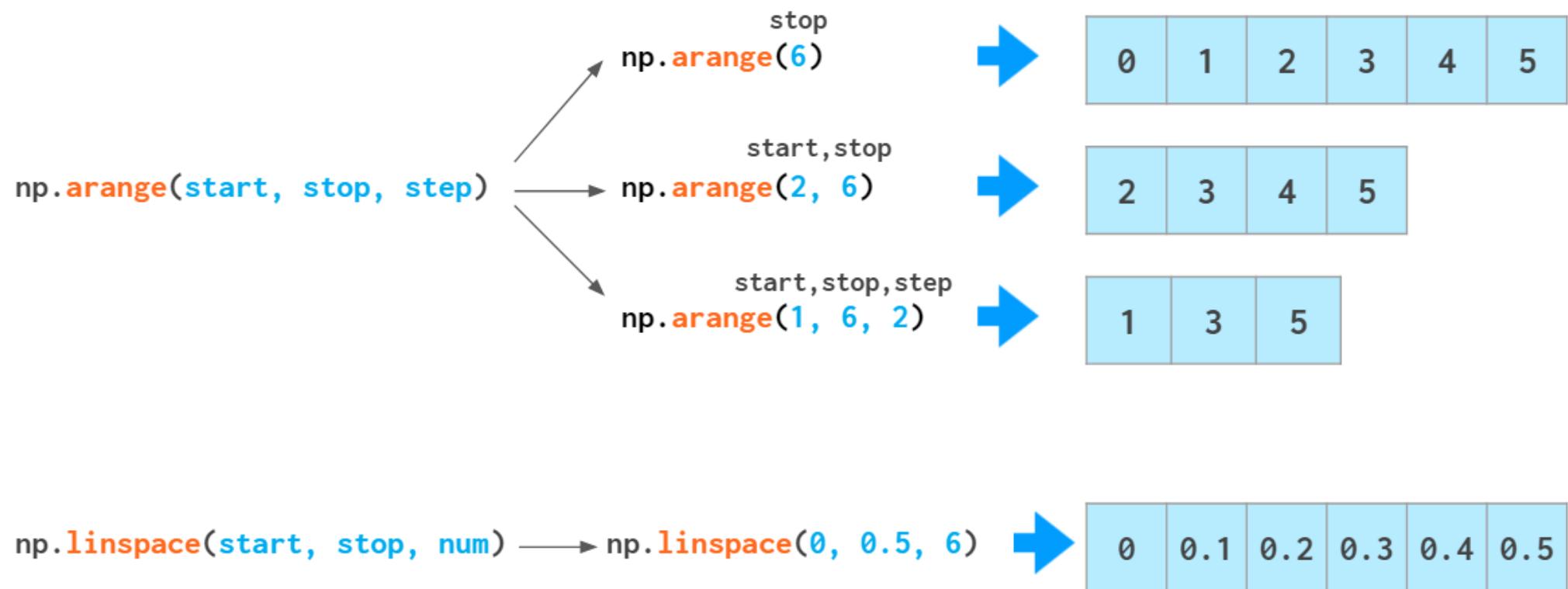


1.	1.
1.	1.
1.	1.

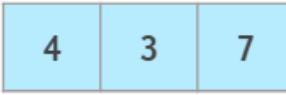
```
np.empty((3, 2))
```



Creating numpy arrays

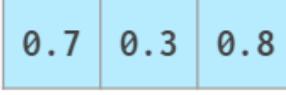


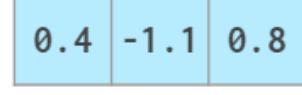
Creating numpy arrays

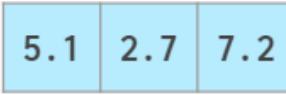
`np.random.randint(0, 10, 3)` → 
uniform, $x \in [0, 10]$

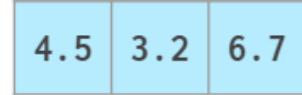
Careful!

`np.random.randint(0, 10)` is $[0, 10]$, but
`random.randint(0, 10)` is $[0, 10]$

`np.random.rand(3)` → 
uniform, $x \in [0, 1)$

`np.random.randn(3)` → 
normal, $\mu=0$, $\sigma=1$

`np.random.uniform(1, 10, 3)` → 
uniform, $x \in [1, 10)$

`np.random.normal(5, 2, 3)` → 
normal, $\mu=5$, $\sigma=2$

Indexing

```
a = np.arange(1, 6)
```

1	2	3	4	5
0	1	2	3	4

a[1]

2

a[2:4]

3	4
---	---

a[-2:]

4	5
---	---

a[::-2]

1	3	5
---	---	---

a[[1,3,4]]

2	4	5
---	---	---

"fancy indexing"

a[2:4] = 0



1	2	0	0	5
---	---	---	---	---

view

a	hea	1	2	3	4	5
	der					

a[2:4]

hea	ptr
-----	-----



Careful when making copies

python list

```
a = [1, 2, 3]
b = a          # no copy
c = a[:]       # copy
d = a.copy()   # copy
```

vs

numpy array

```
a = np.array([1, 2, 3])
b = a          # no copy
c = a[:]       # no copy!!!
d = a.copy()   # copy
```

Boolean indexing

a	1	2	3	4	5	6	7	6	5	4	3	2	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---

a > 5	False	False	False	False	False	True	True	True	False	False	False	False	False
-------	-------	-------	-------	-------	-------	------	------	------	-------	-------	-------	-------	-------

`np.any(a > 5)`

True

`a[a > 5]`

6	7	6
---	---	---

`np.all(a > 5)`

False

`a[a > 5] = 0`

a	1	2	3	4	5	0	0	0	5	4	3	2	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---

& and
| or
^ xor
~ not

`a[(a >= 3) & (a <= 5)] = 0`

a	1	2	0	0	0	6	7	6	0	0	0	2	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Vector operations

$$\begin{bmatrix} 1 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 10 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \end{bmatrix} - \begin{bmatrix} 4 & 8 \end{bmatrix} = \begin{bmatrix} -3 & -6 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 8 \end{bmatrix} * \begin{bmatrix} 2 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 40 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 8 \end{bmatrix} / \begin{bmatrix} 2 & 5 \end{bmatrix} = \begin{bmatrix} 2.0 & 1.6 \end{bmatrix} \text{ np.float64}$$

$$\begin{bmatrix} 4 & 8 \end{bmatrix} // \begin{bmatrix} 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 1 \end{bmatrix} \text{ np.int32}$$

$$\begin{bmatrix} 3 & 4 \end{bmatrix} ** \begin{bmatrix} 2 & 3 \end{bmatrix} = \begin{bmatrix} 9 & 64 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \end{bmatrix} + \begin{bmatrix} 3 \end{bmatrix} = \begin{bmatrix} 4 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \end{bmatrix} - \begin{bmatrix} 3 \end{bmatrix} = \begin{bmatrix} -2 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \end{bmatrix} * \begin{bmatrix} 3 \end{bmatrix} = \begin{bmatrix} 3 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \end{bmatrix} / \begin{bmatrix} 3 \end{bmatrix} = \begin{bmatrix} 0.33 & 0.67 \end{bmatrix} \text{ np.float64}$$

$$\begin{bmatrix} 1 & 2 \end{bmatrix} // \begin{bmatrix} 2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \end{bmatrix} \text{ np.int32}$$

$$\begin{bmatrix} 3 & 4 \end{bmatrix} ** \begin{bmatrix} 2 \end{bmatrix} = \begin{bmatrix} 9 & 16 \end{bmatrix}$$

Math functions

$$a^2 = \begin{bmatrix} 2 & 3 \end{bmatrix} \star\star 2 = \begin{bmatrix} 4 & 9 \end{bmatrix}$$

$$\sqrt{a} = \text{np.sqrt}(\begin{bmatrix} 4 & 9 \end{bmatrix}) = \begin{bmatrix} 2. & 3. \end{bmatrix}$$

$$e^a = \text{np.exp}(\begin{bmatrix} 1 & 2 \end{bmatrix}) = \begin{bmatrix} 2.72 & 7.39 \end{bmatrix}$$

$$\ln a = \text{np.log}(\text{np.e} \mid \text{np.e}^{**2}) = \begin{bmatrix} 1. & 2. \end{bmatrix}$$

$$\vec{a} \cdot \vec{b} = \text{np.dot}(\begin{bmatrix} 1 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 4 \end{bmatrix}) = \begin{bmatrix} 1 & 2 \end{bmatrix} @ \begin{bmatrix} 3 & 4 \end{bmatrix} = \begin{bmatrix} 11 \end{bmatrix}$$

$$\vec{a} \times \vec{b} = \text{np.cross}(\begin{bmatrix} 2 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 3 & 0 \end{bmatrix}) = \begin{bmatrix} 0 & 0 & 6 \end{bmatrix}$$

$$\text{np.floor}(\begin{bmatrix} 1.1 & 1.5 & 1.9 & 2.5 \end{bmatrix}) = \begin{bmatrix} 1. & 1. & 1. & 2. \end{bmatrix}$$

$$\text{np.ceil}(\begin{bmatrix} 1.1 & 1.5 & 1.9 & 2.5 \end{bmatrix}) = \begin{bmatrix} 2. & 2. & 2. & 3. \end{bmatrix}$$

$$\text{np.round}(\begin{bmatrix} 1.1 & 1.5 & 1.9 & 2.5 \end{bmatrix}) = \begin{bmatrix} 1. & 2. & 2. & 2. \end{bmatrix}$$

$$\text{np.max}(\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}) = \begin{bmatrix} 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}.max() = \begin{bmatrix} 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}.argmax() = \begin{bmatrix} 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}.min() = \begin{bmatrix} 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}.argmin() = \begin{bmatrix} 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}.sum() = \begin{bmatrix} 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}.mean() = \begin{bmatrix} 2 \end{bmatrix}$$

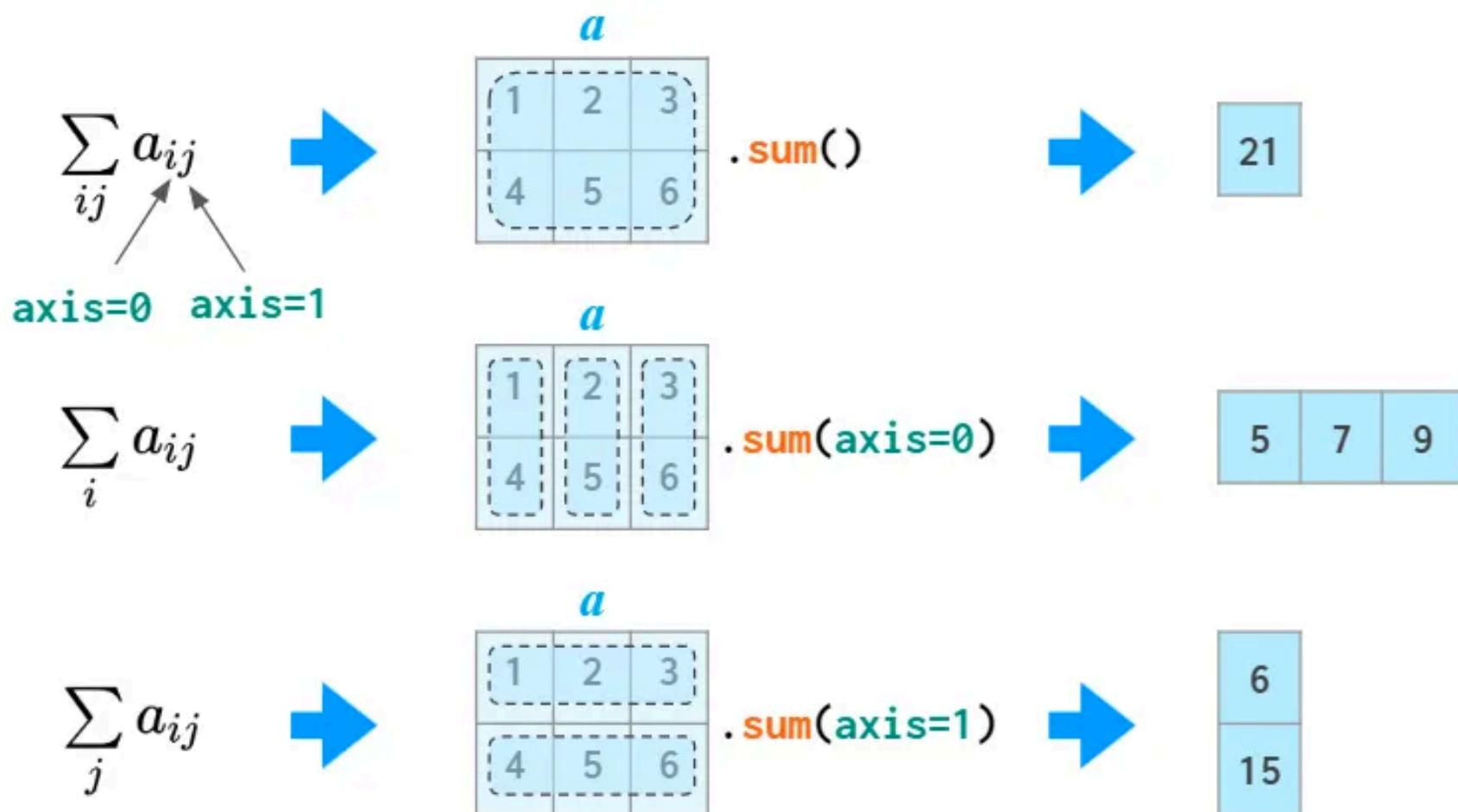
$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}.var() = \begin{bmatrix} 0.67 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}.std() = \begin{bmatrix} 0.82 \end{bmatrix}$$

$$\bar{S}^2 = \frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^2, \quad \bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$$

$$a = 2 \pm 0.82$$

The axis



Matrix operations

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 3 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 3 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \color{red}{\ast} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \color{red}{\circ} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \color{red}{/} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 0.5 & 2 \\ 3 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \color{red}{**} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 16 \end{bmatrix}$$

Broadcasting

1	2	3
4	5	6
7	8	9

/

9	9	9
9	9	9
9	9	9

=

.1	.2	.3
.4	.5	.7
.8	.9	1.

normalization

1	2	3
4	5	6
7	8	9

*

-1	0	1
-1	0	1
-1	0	1

=

-1	0	3
-4	0	6
-7	0	9

multiplying several columns at once

1	2	3
4	5	6
7	8	9

/

3	3	3
6	6	6
9	9	9

=

.3	.7	1.
.6	.8	1.
.8	.9	1.

row-wise normalization

1	2	3
1	2	3
1	2	3

*

1	1	1
2	2	2
3	3	3

=

1	2	3
2	4	6
3	6	9

outer product

General broadcasting rules

- › When operating on two arrays, NumPy compares their shapes element-wise
 - ✓ starts with the rightmost dimension and works backwards
 - ✓ two dimensions are compatible when:
 - they are equal, or
 - one of them is 1
 - ✓ otherwise a `ValueError: operands could not be broadcast together` exception is thrown
- › Input arrays do not need to have the same number of dimensions
 - ✓ resulting array will have the same number of dimensions as the input array with the greatest number of dimensions
 - ✓ the size of each dimension is the largest size of the corresponding dimension among the input arrays
 - ✓ missing dimensions are assumed to have size one

A (4d array): 8 x 1 x 6 x 1
B (3d array): 7 x 1 x 5
Result (4d array): 8 x 7 x 6 x 5

A (2d array): 5 x 4
B (1d array): 1
Result (2d array): 5 x 4

A (2d array): 5 x 4
B (1d array): 4
Result (2d array): 5 x 4

A (3d array): 15 x 3 x 5
B (3d array): 15 x 1 x 5
Result (3d array): 15 x 3 x 5

A (3d array): 15 x 3 x 5
B (2d array): 3 x 5
Result (3d array): 15 x 3 x 5

A (3d array): 15 x 3 x 5
B (2d array): 3 x 1
Result (3d array): 15 x 3 x 5

A (1d array): 3
B (1d array): 4 # mismatch

A (2d array): 2 x 1
B (3d array): 8 x 4 x 3 # mismatch

Must know your shapes ...

$$\begin{array}{c|c} 1 \\ 2 \\ 3 \end{array} @ \begin{array}{c|c|c} 1 & 2 & 3 \end{array} = \begin{array}{c|c|c} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{array} \text{ outer product}$$

$$\begin{array}{c|c|c} 1 & 2 & 3 \end{array} @ \begin{array}{c|c} 1 \\ 2 \\ 3 \end{array} = 14 \text{ inner (or dot) product}$$

Concatenation

a

1	2	3	4
5	6	7	8
9	10	11	12

c

1	2
3	4
5	6

`np.hstack((a, c))`

1	2	3	4	1	2
5	6	7	8	3	4
9	10	11	12	5	6



b

1	2	3	4
5	6	7	8



`np.vstack((a, b))`

1	2	3	4
5	6	7	8
9	10	11	12
1	2	3	4
5	6	7	8