Machine learning for vision and multimedia
(01URPOV)

Preliminaries – Vectorization in Python and Numpy

Lia Morra

2024 – 2025

Politecnico di Torino

Why vectorization

$$\xrightarrow[\text{“vectorization”}]{\text{1 (car) or 0 (non car)?}}$$

$$X = \underbrace{\begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \ldots & x^{(m)} \\ | & | & & | \end{bmatrix}}_{\text{m training examples}} \left.\vphantom{\begin{bmatrix} | \\ x \\ | \end{bmatrix}}\right\} \text{N, or } n_x \text{ features}$$

$$Y = \underbrace{\begin{bmatrix} y^{(1)} & y^{(2)} & \ldots & y^{(m)} \end{bmatrix}}_{m}$$

Why vectorization

- Machine/deep learning libraries operate on **tensors**, a generalization of arrays and matrices to N-dimension
- *"In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a tensor" (I. Goodfellow, Deep learning, p.33)*
- **Vectorization** refers to a style of programming in which traditional **for** loops and **if** statements are replaced by vector operations
- Vectorization supports the efficient **Single Instruction Multiple Data (SIMD)** computational model (GPU)

Vectorization: NumPy

- NumPy is the core library for scientific computing in Python
- It provides a high-performance multidimensional array object, and functions/instructions for efficient manipulation
- Many ML libraries (SciPy, scikit-learn) are built on numpy arrays, which are also useful for loading and preprocessing data
- Tensorflow/Pytorch have their own Tensor implementation but many concepts are similar (e.g., broadcasting)

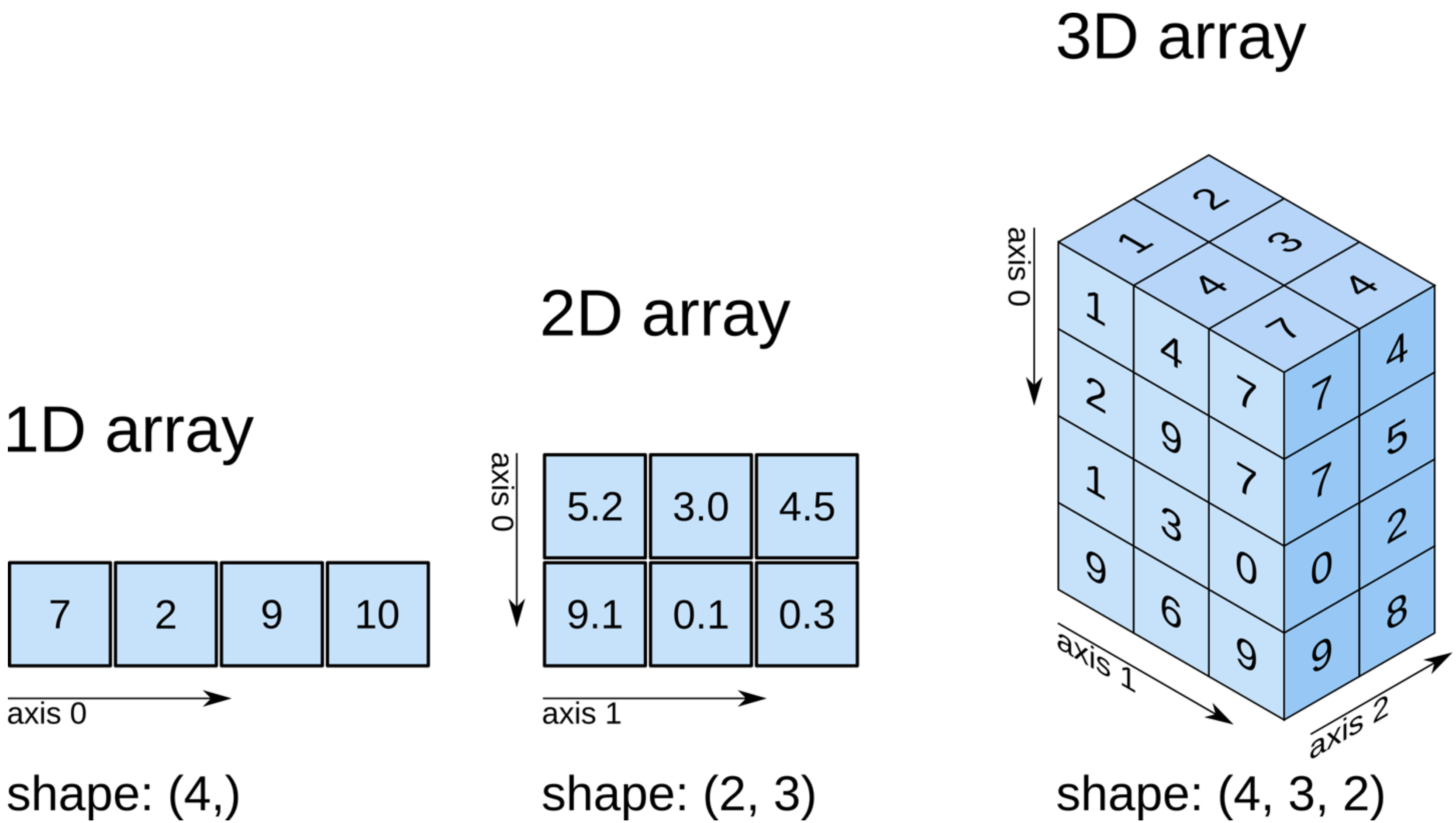
NumPy vs. Matlab

- NumPy is quite similar, in many ways, to Matlab
- Like Matlab, it is heavily optimized for vectorized operations
  - It is considerable easier to switch from Matlab to Python/numpy than from C/C++/Java or other procedural languages
  - There are however many differences in notation
- A quick reference of the main differences is available here:
  - https://numpy.org/doc/stable/user/numpy-for-matlab-users.html

NumPy arrays

- An array is a grid of values, all of the same type, indexed by nonnegative integers
- The number of dimensions is the **rank** of the array
- The **shape** of an array is a tuple of integers giving the size of the array along each dimension

NumPy arrays (II)



https://www.oreilly.com/library/view/elegant-scipy/9781491922927/ch01.html

Creating arrays

- Arrays can be created using a variety of functions:
  - each initializes the values differently (zeros, ones, eyes, random)
- All take as input the shape
  - (1, 2, 2) creates a tensor of rank 3 with dimensions equal to 1, 2 and 2 along axis 0, 1 and 2, respectively
  - (2, 2) creates a tensor of rank 2 with dimensions equal to 2 and 2 along axis 0 and 1, respectively
  - (2, 256, 256, 3) creates a tensor of rank 4 with dimensions equal to 2, 256, 256 and 3, respectively, or… a batch of 2 images, of size 256 by 256, with 3 channels (RGB)

Indexing and slicing

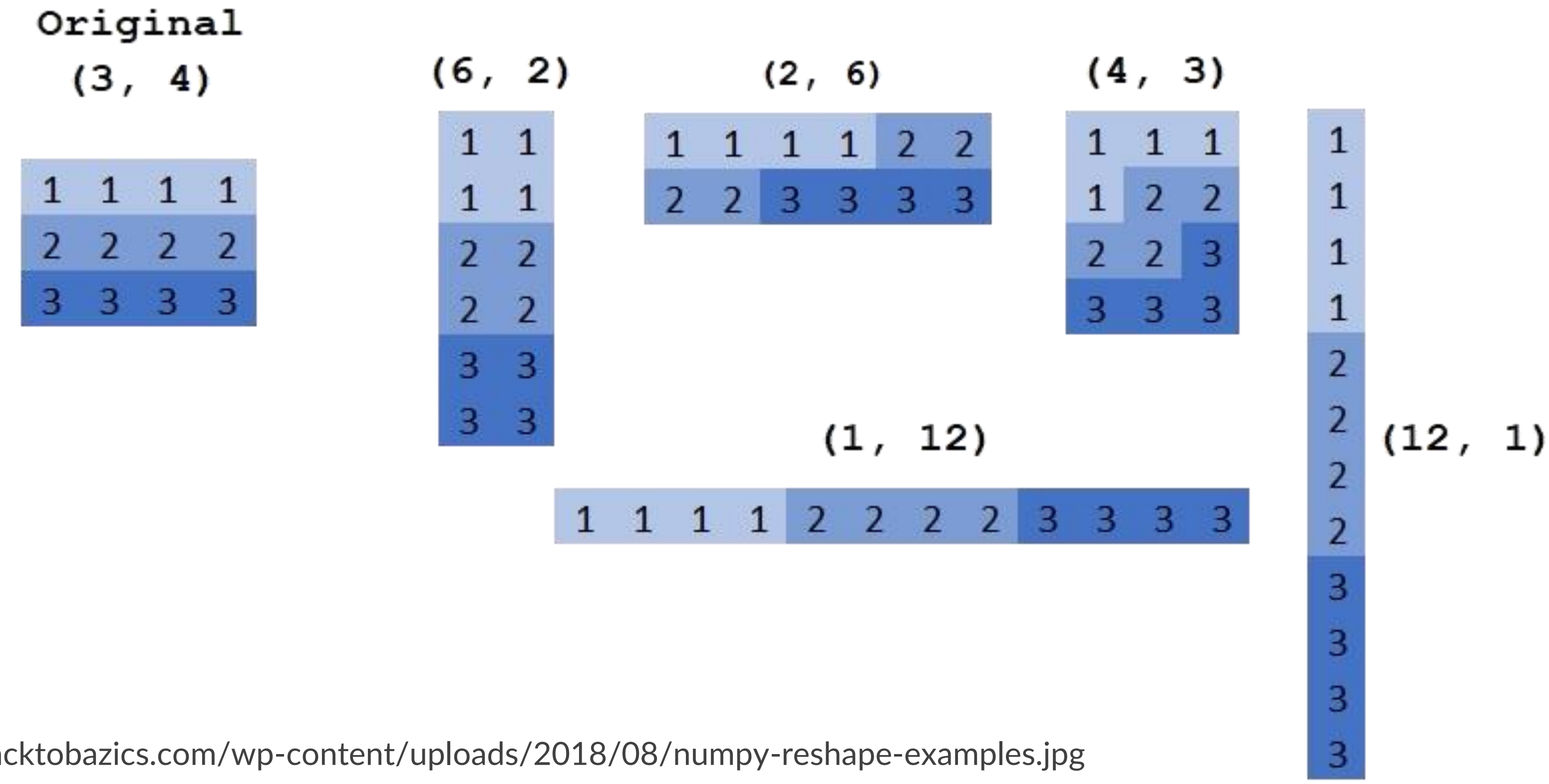
- **Integer** indexing: tensor values can be accesses by their position (from 0 to n – 1)
  - X[2]: retrieves the 3rd value
  - X[-2]: negatives indices count from the end of the array
  - X[i,j]: retrieves the value at position (i,j) from a 2-dimensional array
- **Slicing** provides a concise syntax to access subarrays
  - X[a:b] retrieves from index a (included) to b (excluded)
  - if a is empty X[:b], the slice starts from 0
  - if b is empty X[a:], the slice stops at the end of the array
  - In the case of N-dimensional arrays, each dimension can be sliced independently

Indexing and slicing (II)

- It is possible to mix integer indexing and slicing, but this will affect the rank
- Example: let a be an array of size (3,4). The following instructions will both copy the second row of array a into a new array
  - r1 = a[1, :]
  - r2 = a[1:2, :]
- However:
  - r1 has rank 1, i.e. its shape is equal to (4)
  - r2 has rank 2, i.e. its shape is equal to (1,4)

Reshaping

- An array can be **reshaped** by rearranging its content, as long as the total number of elements does not change



http://backtobazics.com/wp-content/uploads/2018/08/numpy-reshape-examples.jpg

Array math

- Basic mathematical functions operate **element-wise** on arrays, and are available both as **overloaded operators** (+, -, *, /) and as functions (add, subtract, multiply, divide)
- Most mathematical functions operate element-wise on arrays
  - E.g. np.sqrt(x) returns a vector, of the same shape of x, where each element *i* is the square root of x_i
- ND-array multiplications can be computed using the functions **dot** or **matmul**
  - For 1-D arrays, they return the inner product
  - For 2-D arrays, they return the matrix product
  - For N-D arrays, they have different behaviors

Vectorization – Linear regression

$$Z \quad = Xw \quad + \quad b$$

$$\begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}^T \begin{bmatrix} x_{11} & \cdots & x_{m1} \\ \vdots & & \vdots \\ x_{1n} & \cdots & x_{mn} \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

```
z =np.dot(w, x) + b
```

Element-wise addition

Given M examples and N features:

- x is a matrix m×n
- w and b are vectors of size n
- z is a vector of size m

2D-array multiplication

$$c_{11} = a_{11}\ b_{11} + a_{12}\ b_{21} + a_{13}\ b_{31} + a_{14}\ b_{41}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{2e} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

2x4 **4x3** **2x3**

$$c_{22} = a_{21}\ b_{21} + a_{22}\ b_{22} + a_{23}\ b_{32} + a_{24}\ b_{42}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{2e} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

Reminder – matrix multiplication

- Matrix multiplication is not commutative
- Matrix multiplication is not defined if the inner numbers of the sizes do not match

Array operations

- Numpy provides many useful functions for performing computations over arrays, such as taking the sum, mean, min and max of its value
- Array operations can be performed on the whole array (default) or along specific axis

$$\begin{array}{cc} & \text{axis 1} \rightarrow \\ \text{axis 0} \downarrow & \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 10 \end{bmatrix} \end{array} \qquad \begin{bmatrix} 6 \\ 21 \end{bmatrix}$$

```
np.sum(a, axis=1)
```

$$[6 \quad 8 \quad 13] \qquad 27$$

```
np.sum(a)
```

```
np.sum(a, axis=0)
```

Broadcasting

- Broadcasting allows, under certain conditions, to combine arrays with incompatible shapes during mathematical operations
- The smaller array is "broadcasted" by copying its content across the larger array, thus avoiding:
  - looping in Python (computational inefficiency)
  - creating array copies (memory inefficiency)
- Broadcasting compares the shapes of the arrays to check for their compatibility
- If the compatibility conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown

Broadcasting rules

- Broadcasting compares the shapes of the arrays starting with the trailing dimensions and going forwards
- Arrays do not need to have the same rank (number of dimension)
- Two dimensions are compatible if
  - they have the same value, or
  - one of them is 1
- The output size for each axis is the largest one (i.e., the one that is not 1)

Broadcasting rules (II)

Shapes that broadcast

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

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

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

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

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

Shapes that do not broadcast

```
A      (2d):     4
B      (1d):     3

A      (2d): 5 x 4
B      (1d): 5 x 3

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

A      (3d): 15 x 3 x 5
B      (2d):      3 x 6

A      (3d): 15 x 3 x 5
B      (2d):      4 x 1
```

Broadcasting example

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 10 \end{bmatrix} * [5] = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 10 \end{bmatrix} * \begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \end{bmatrix} = \begin{bmatrix} 5 & 10 & 15 \\ 25 & 30 & 50 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 5 & 0 & 0 \end{bmatrix}$$

Exercise 1

- Given the amount of Carbs, Proteins, Fats in 100g of different foods:

$$\begin{array}{r c} & \begin{array}{cccc} \text{Pear} & \text{Broccoli} & \text{Pasta} & \text{Avocado} \end{array} \\ \begin{array}{r} \text{Carb} \\ \text{Protein} \\ \text{Fat} \end{array} & \begin{bmatrix} 27 & 5.8 & 41.4 & 18.0 \\ 0.7 & 2.5 & 8 & 4.0 \\ 0.2 & 0.3 & 1.2 & 29.5 \end{bmatrix} \end{array}$$

- knowing that carbs and proteins provides 4 calories and fats 9,
- calculate the % of calories from carbs, proteins and fats for each food.

Exercise 2

- Given two 1-D arrays of values **x** and an array of binary labels **y**, calculate the vector **z** so that:

$$z_i = \begin{cases} \|x_i\|^2 & if\ y_i = 1 \\ \|m - x_i\|^2 & if\ y_i = 0 \end{cases}$$

- where m is a constant parameter.

Exercise 3

- Min–max scaling or min–max normalization is the simplest feature rescaling method and consists in rescaling each feature to the range [0, 1]
- The general formula for min–max normalization is as follows: $x' = \frac{x - \min(x)}{\max(x) - \min(x)}$
- Given a matrix **x** of size M by N, where M is the number of samples and N is the number of features, write a vectorized expression to perform min–max scaling

Exercise 4

- Given 1D array, calculate the average of each consecutive triplet

[1 3 5 | 10 15 12 | 23 5 6]

↓

[**3** | **12.3** | **11.3**]

References and tutorials

- https://cs231n.github.io/python-numpy-tutorial/
- https://numpy.org/devdocs/user/index.html
- https://numpy.org/doc/stable/user/numpy-for-matlab-users.html
- https://numpy.org/doc/stable/reference/routines.math.html
- https://www.w3resource.com/python-exercises/numpy/index.php