Programowanie w Pythonie
Biblioteka numpy

Łukasz Mioduszewski, UKSW 2022

Czym jest Numpy?

- Numpy, Scipy i Matplotlib dają Pythonowi funkcjonalność programu MATLAB
- Cechy Numpy:
  - Wielowymiarowe tablice (macierze) konkretnego typu
  - Szybkie obliczenia numeryczne (zwłaszcza macierzowe)
  - Różne funkcje matematyczne

Czemu warto używać Numpy

- Pętle w Pythonie są powolne
- Mnożenie macierzy 1000 x 1000
  - Potrójna pętla w Pythonie zajmuje > 10 minut
  - To samo mnożenie w Numpy to ~0.03 sekund

NumPy – najważniejsze ficzery

1. Tablice
2. Transpozycja macierzy i inne zmiany "kształtu"
3. Operacje matematyczne
4. Indeksowanie i cięcie po indeksach [:]

Tablice

Coś jak listy liczb, tylko szybsze i mniej elastyczne

1. **Wektory**
2. **Macierze**
3. Obrazy
4. Tensory
5. Sieci neuronowe (convnet)

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Tablice

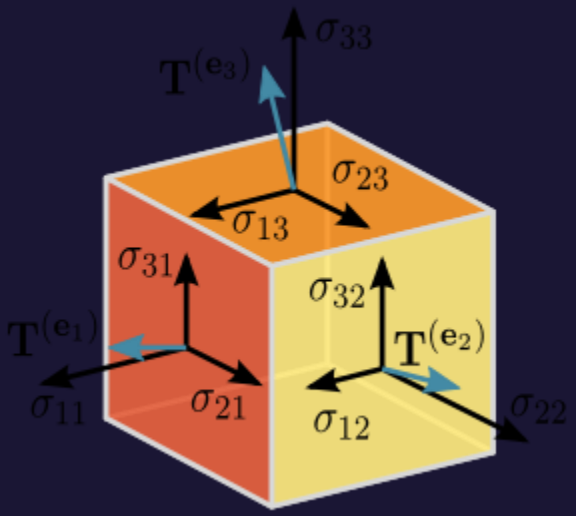
Coś jak listy liczb, tylko szybsze i mniej elastyczne

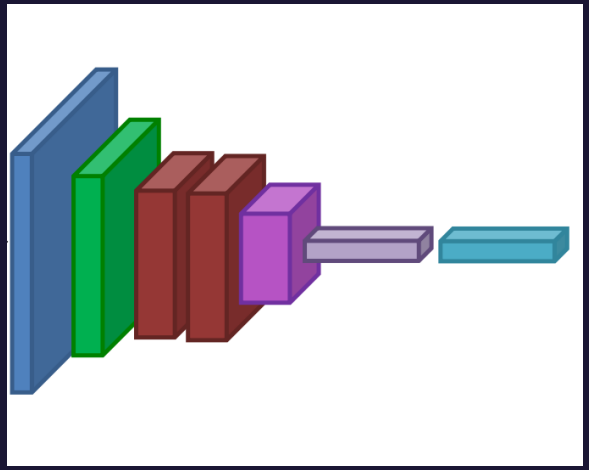
1. Wektory
2. Macierze
3. **Obrazy**
4. Tensory
5. Sieci neuronowe (convnet)

Tablice

Coś jak listy liczb, tylko szybsze i mniej elastyczne

1. Wektory
2. Macierze
3. Obrazy
4. **Tensory**
5. Sieci neuronowe (convnet)





Tablice

Coś jak listy liczb, tylko szybsze i mniej elas

1. Wektory
2. Macierze
3. Obrazy
4. Tensory
5. **Sieci neuronowe (convnet)**

Właściwości tablic

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]],dtype=np.float32)

print a.ndim, a.shape, a.dtype
```

1. Dowolna liczba wymiarów: 0 (skalar), 1 (wektor), 2 (macierz)…
2. Mają typ: np.uint8, np.int64, np.float32, np.float64
3. Określony rozmiar, każdy element istnieje i jest tego samego typu

Tworzenie tablic

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

Tworzenie tablic

- **np.ones, np.zeros**
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.ones((3,5),dtype=np.float32)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]], dtype=float32)
```

```
>>> np.zeros((6,2),dtype=np.int8)
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8)
```

Tworzenie tablic

- np.ones, np.zeros
- **np.arange**
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.arange(1334,1338)
array([1334, 1335, 1336, 1337])
```

Tworzenie tablic

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A = np.ones((2,3))
>>> B = np.zeros((4,3))
>>> np.concatenate([A,B])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
```

Tworzenie tablic

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A = np.ones((4,1))
>>> B = np.zeros((4,2))
>>> np.concatenate([A,B], axis=1)
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

Tworzenie tablic

- np.ones, np.zeros
- np.arange
- np.concatenate
- **np.astype**
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A
array([[ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5]], dtype=float32)
>>> print(A.astype(np.uint16))
[[4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]]
```

Tworzenie tablic

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- **np.zeros_like, np.ones_like**
- np.random.random

```
>>> a = np.ones((2,2,3))
>>> b = np.zeros_like(a)
>>> print(b.shape)
```

Tworzenie tablic

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- **np.random.random**

```
>>> np.random.random((10,3))
array([[ 0.61481644,  0.55453657,  0.04320502],
       [ 0.08973085,  0.25959573,  0.27566721],
       [ 0.84375899,  0.2949532 ,  0.29712833],
       [ 0.44564992,  0.37728361,  0.29471536],
       [ 0.71256698,  0.53193976,  0.63061914],
       [ 0.03738061,  0.96497761,  0.01481647],
       [ 0.09924332,  0.73128868,  0.22521644],
       [ 0.94249399,  0.72355378,  0.94034095],
       [ 0.35742243,  0.91085299,  0.15669063],
       [ 0.54259617,  0.85891392,  0.77224443]])
```

Na co uważać przy tablicach

- Wszystkie elementy muszą być podane (nigdzie nie może być None)
- Muszą być wszystkie jednego typu
- Nie można łączyć tablic różnych rozmiarów i kształtów

```
>>> np.ones([7,8]) + np.ones([9,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together
 with shapes (7,8) (9,3)
```

"Kształt" tablic

```
a = np.array([1,2,3,4,5,6])
a = a.reshape(3,2)
a = a.reshape(2,-1)
a = a.ravel() # spłaszczenie do 1d
```

1. Całkowita liczba elementów pozostaje ta sama
2. Jeżeli któryś z rozmiarów wynosi -1, wyliczy się na podstawie pozostałych
3. Najpierw wiersze, potem kolumny

Co funkcje zwracają

- Funkcje Numpy zwracają widoki (**views**) albo kopie (**copies**).
- Widoki działają jak typy mutowalne w Pythonie – ich zmiana zmienia oryginał
- Dokumentacja mówi która funkcja co zwraca
- Można używać np.copy oraz np.view do tworzenia kopii i widoków

Transpozycja

```
a = np.arange(10).reshape(5,2)

a = a.T

a = a.transpose((1,0))
```

np.transpose zamienia osie

a.T transponuje pierwsze dwie osie

Serializacja

```
np.savez('data.npz', a=a)

data = np.load('data.npz')

a = data['a']
```

1. Pliki npz mogą przechowywać wiele tablic naraz
2. np.savez_compressed podobnie

Obrazki

Obrazki to tablice 3d: szerokość, wysokość i kanały

Przykładowe formaty:

- height x width x RGB ([x][y][3])
- height x width (każdy kolor oddzielnie)

Uwaga:

- Kanały mogą być w kolejności BGR (biblioteka OpenCV tak robi)
- Czasem może być [width x height] zamiast [height x width]

Zapisywanie i wczytywanie obrazków

SciPy: `skimage.io.imread,skimage.io.imsave`

height x width x RGB

PIL / Pillow: `PIL.Image.open, Image.save`

width x height x RGB

OpenCV: `cv2.imread, cv2.imwrite`

height x width x BGR

Przerwa

Wiemy jak tworzyć tablice, zmieniać ich kształt i permutować osie

Operacje matematyczne

- **Operacje arytmetyczne są zawsze element po elemencie**
- Operacje logiczne zwracają tablicę booli
- Operacja w miejscu modyfikuje tablicę

```
>>> a
array([1, 2, 3])
>>> b
array([ 4,  4, 10])
>>> a * b
array([ 4,  8, 30])
```

Operacje matematyczne i logiczne

- Operacje arytmetyczne są zawsze element po elemencie
- **Operacje logiczne zwracają tablicę booli**
- Operacja w miejscu modyfikuje tablicę

```
>>> a
array([[ 0.93445601,  0.42984044,  0.12228461],
       [ 0.06239738,  0.76019703,  0.11123116],
       [ 0.14617578,  0.90159137,  0.89746818]])
>>> a > 0.5
array([[ True, False, False],
       [False,  True, False],
       [False,  True,  True]], dtype=bool)
```

Operacje matematyczne

- Operacje arytmetyczne są zawsze element po elemencie
- Operacje logiczne zwracają tablicę booli
- **Operacja w miejscu modyfikuje tablicę**

```
>>> a
array([[ 4, 15],
       [20, 75]])
>>> b
array([[ 2,  5],
       [ 5, 15]])
>>> a /= b
>>> a
array([[2, 3],
       [4, 5]])
```

Upkonwersja

W przypadku dwóch różnych typów wynik ma typ bardziej precyzyjny

uint64 + uint16 => uint64

float32 / int32 => float32

Uwaga: upkonwersja nie zapobiega błędom overflow/underflow

Przykład: Obrazki zwykle zapisujemy jako uint8 (256 wartości dla R,G,B). Przekonwertuj na float32 albo float64 przed skomplikowaną matematyką

Uniwersalne funkcje

Działają na wszystkim element po elemencie

Przykłady:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

```
>>> a
array([[ 1,  4],
       [ 9, 16],
       [25, 36]])
>>> np.sqrt(a)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
```

Indeksowanie

```
x[0,0]    # top-left element
x[0,-1]   # first row, last column
x[0,:]    # first row (many entries)
x[:,0]    # first column (many entries)
```

Uwagi:

- Zawsze indeksujemy od zera
- Indeksy wielowymiarowe oddzielamy przecinkiem (i możemy przekazywać jako tuplę)

Indeksowanie i cięcie

```
I[1:-1,1:-1]          # select all but one-pixel border

I = I[:,:,::-1]       # swap channel order

I[I<10] = 0           # set dark pixels to black

I[[1,3], :]           # select 2nd and 4th row
```

1. Cięcia są **widokami**. Zapis do cięcia nadpisze oryginalną tablicę
2. Można także indeksować listą albo tablicą booli (wtedy elementy True będą w cięciu, a elementów False nie będzie)
3. Można indeksować listą albo tablicą liczb, wtedy zostaną wzięte wiersze/kolumny… (zależy który wymiar indeksujemy) o danym numerze

Cięcia

Składnia: start:stop:krok

```
a = list(range(10))

a[:3] # indices 0, 1, 2

a[-3:] # indices 7, 8, 9

a[3:8:2] # indices 3, 5, 7

a[4:1:-1] # indices 4, 3, 2 (this one is tricky)
```

Osie

```
a.sum() # sum all entries

a.sum(axis=0) # sum over rows

a.sum(axis=1) # sum over columns

a.sum(axis=1, keepdims=True) # na następnym slajdzie
```

1. Parametr axis mówi na której osi Numpy ma pracować
2. Zwykle w sumowaniu oś znika, keepdims nie zmienia wymiarów tablicy

Osie

osie.py

```
 1  import numpy as np
 2  a = np.random.rand(2,3,4)
 3  a.shape  # => (2, 3, 4)
 4  # Note: axis=0 refers to the first dimension of size 2
 5  #       axis=1 refers to the second dimension of size 3
 6  #       axis=2 refers to the third dimension of size 4
 7
 8  a.sum(axis=0).shape # => (3, 4)
 9  # Simple sum over the first dimension, we "lose" that
10  # dimension because we did an aggregation (sum) over it
11
12  a.sum(axis=0, keepdims=True).shape # => (1, 3, 4)
13  # Same sum over the first dimension, but instead of
14  # "loosing" that dimension, it becomes 1.
```

Rozszerzanie

```
a = a + 1 # dodaj 1 do każdego elementu
```

Kiedy operujemy na wielu tablicach, stosujemy reguły rozszerzania

Każdy wymiar musi się zgadzać (od prawej do lewej)

1. Wymiary rozmiaru 1 będą rozszerzone (tak jakby dana wartość się powtarzała).
2. Jeśli wymiar ma inny rozmiar niż 1, musi się zgadzać
3. Jeśli po lewej brakuje wymiaru o rozmiarze 1, zostanie dodany

Rozszerzanie – przykład

Chcemy dodać kolor do obrazka

a.shape wynosi 100, 200, 3

b.shape wynosi 3

a + b rozszerzy b tak że będzie miało efektywny kształt 1 x 1 x 3.

W dodawaniu rozszerzone zostały pierwsze dwa wymiary (zostały dodane po lewej)

Kiedy rozszerzanie nie działa

a.shape wynosi 100, 200, 3

b.shape wynosi 4

wtedy a + b się nie uda, bo ostatni wymiar musi być albo równy (w tym przypadku 3), albo wynosić 1 (wtedy zostanie rozszerzony)

Jak unikać błędów

1. Pamiętaj jakiego typu są tablice
2. Pilnuj czy pracujesz na widoku czy na kopii
3. Używaj matplotlib do robienia wykresów
4. Zapamiętaj funkcje (a i b to wektory o rozmiarze n):
   np.dot (iloczyn skalarny), np.dot(a,b) zwróci skalar
   np.cross(iloczyn wektorowy), np.cross(a,b) zwróci wektor o rozmiarze n
   np.multiply(a,b) to to samo co a*b, zwróci wektor o rozmiarze n

Okna

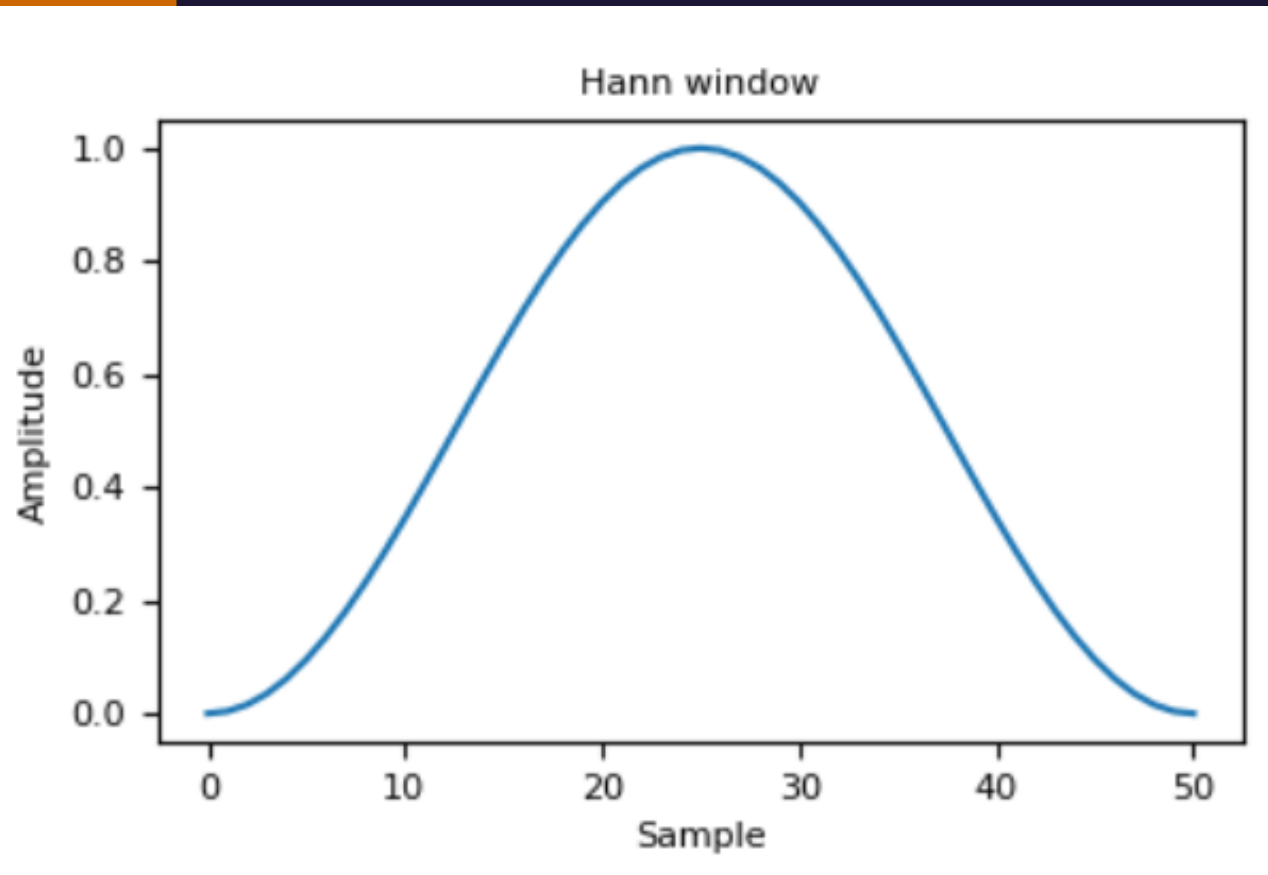
- Używane w analizie sygnałów, poprzez konwolucję zmieniają właściwości obrazu
- np.hanning(M) zwraca okno von Hanna, czyli M-elementową jednowymiarową tablicę, która zawiera kolejne punkty z funkcji Hanninga, znormalizowanej do 1

```
>>> np.hanning(12)
array([0.        , 0.07937323, 0.29229249, 0.57115742, 0.82743037,
       0.97974649, 0.97974649, 0.82743037, 0.57115742, 0.29229249,
       0.07937323, 0.        ])
```

Okna

hanning.py

```
 1  import matplotlib.pyplot as plt
 2  from numpy.fft import fft, fftshift
 3  window = np.hanning(51)
 4  plt.plot(window)
 5  [<matplotlib.lines.Line2D object at 0x
 6  plt.title("Hann window")
 7  Text(0.5, 1.0, 'Hann window')
 8  plt.ylabel("Amplitude")
 9  Text(0, 0.5, 'Amplitude')
10  plt.xlabel("Sample")
11  Text(0.5, 0, 'Sample')
12  plt.show()
```



Okna

- Blackman

Hanning

Bartlett