Praxisgrundlagen der Informatik

Performance: Vectorization with NumPy

Prof. Dr. Eduard Kromer
University of Applied Sciences Landshut

Vectorization

Vectorization

- vectorization or array programming refers to solutions which allow the application of *operations to an entire set of values at once*
- vectorization allows us to operate on aggregates of data without having to resort to explicit loops of individual scalar operations
- knowing how to do several operations at once will speed up your program

Example: Matrix Multiplication

Multiply the matrices:

$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ und $B = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ What is $A \cdot B =?$

```
# two 3x3 matrices
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
B = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
```

$$A \cdot B = \begin{bmatrix} 4 & 2 & 4 \\ 10 & 5 & 10 \\ 16 & 8 & 16 \end{bmatrix}$$

Example: Matrix Multiplication with Python

```python
def mat_mul(A, B):
    num_rows = len(A)
    num_cols = len(B[0])
    result =[[0 for _ in range(num_cols)] for _ in range(num_rows)]

    for i in range(num_rows):
        for j in range(num_cols):
            for k in range(len(B)):
                result[i][j] += A[i][k] * B[k][j]
    return result
```

```python
mat_mul(A, B)
```

```
[[4, 2, 4], [10, 5, 10], [16, 8, 16]]
```

Example: Matrix Multiplication with NumPy

```
import numpy as np

np.dot(A, B)
```

```
array([[ 4,  2,  4],
       [10,  5, 10],
       [16,  8, 16]])
```

```
A = np.matrix(np.arange(10000).reshape((100,100))).tolist()
B = np.matrix(np.arange(10000).reshape((100,100))).tolist()
```

```
%%timeit
mat_mul(A, B)
```

```
57.5 ms ± 459 µs per loop (mean ± std. dev. of 7 runs, 10
loops each)
```

```
%%timeit
np.dot(A,B)
```

```
1.42 ms ± 1.88 µs per loop (mean ± std. dev. of 7 runs,
1,000 loops each)
```

Code Vectorization vs. Problem Vectorization

Code Vectorization:

- we don’t need to rethink our problem since it is inherently vectorizable
- we can rewrite our existing code with NumPy (or CuPy) without needing to resort to completely different algorithms

Problem Vectorization:

- we need to rethink our problem in order to make it vectorizable
- this is a much harder problem than code vectorization because we might need completely different algorithms to solve our problem

Introduction to NumPy

Introduction to NumPy

- the first step, to be able to work with data and analyze it, is to transform it into an array of numbers
- efficient storage and manipulation of numerical arrays is of utmost importance for data science, scientific computing and artificial intelligence
- one specialized, very important library that enables those types of operations in Python is NumPy (short for Numerical Python)
- NumPy forms the core of nearly the entire ecosystem of data science tools in Python
- NumPy arrays are similar to Python's `list` type – but much more storage efficient and with more efficient data operations

NumPy-enabled Ecosystem

Quantum Computing	Statistical Computing	Signal Processing	Image Processing	Graphs and Networks	Astronomy	Cognitive Psychology
QuTiP	Pandas	SciPy	Scikit-image	NetworkX	AstroPy	PsychoPy
PyQuil	statsmodels	PyWavelets	OpenCV	graph-tool	SunPy	
Qiskit	Xarray	python-control	Mahotas	igraph	SpacePy	
PennyLane	Seaborn			PyGSP		

Bioinformatics	Bayesian Inference	Mathematical Analysis	Chemistry	Geoscience	Geographic Processing	Architecture & Engineering
BioPython	PyStan	SciPy	Cantera	Pangeo	Shapely	COMPAS
Scikit-Bio	PyMC3	SymPy	MDAnalysis	Simpeg	GeoPandas	City Energy Analyst
PyEnsembl	ArviZ	cvxpy	RDKit	ObsPy	Folium	Sverchok
ETE	emcee	FEniCS	PyBaMM	Fatiando a Terra		

Python lists vs. NumPy arrays

- Python lists are very flexible and allow to store different types of objects
  - `[5.5, 3, True, "Python"]` → `[float, int, bool, str]`
- this flexibility comes at a cost since each element of the list carries both data and type information
- fixed-type NumPy arrays lack this flexibility, but are much more efficient in terms of storage and data manipulation

```
import numpy as np


numbers = np.array([1,2,3,4,5])
numbers.dtype
```

```
dtype('int64')
```

NumPy arrays

If types don't match in a numpy array, NumPy tries to upcast, if possible:

```
numbers = np.array([1,2,3,4,5.5])
numbers.dtype
```

```
dtype('float64')
```

Use `dtype` to set the data type explicitly:

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

```
dtype('float32')
```

- Read about array types and conversions between types in NumPy here.

Multidimensional Arrays

- NumPy arrays can be multidimensional (unlike Python lists)
- How to create arrays in NumPy:

```
np.zeros(5, dtype=int)
```

```
array([0, 0, 0, 0, 0])
```

```
np.ones((2,2,2), dtype=float)
```

```
array([[[1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.]]])
```

Multidimensional Arrays

How to create arrays in NumPy:

```
np.full((2,3), 8)
```

```
array([[8, 8, 8],
       [8, 8, 8]])
```

```
np.arange(0,8,2)
```

```
array([0, 2, 4, 6])
```

```
np.linspace(0,10,5)
```

```
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

Distributions: Uniform Distribution

For all available distributions, see the Random Generator documentation.

Uniform Distribution:

```
# floats from the uniform distribution
np.random.default_rng().uniform(-1,1,10)
```

```
array([ 0.12296559,  0.04440568,  0.80050969, -0.52581581, -0.38279829,
       -0.54750314, -0.91251035,  0.73433773, -0.94700489, -0.85324058])
```

```
# integers from the "discrete uniform" distribution
np.random.default_rng().integers(0,5,dtype=np.int64, size=5)
```

```
array([4, 2, 2, 1, 1])
```

Distributions: Normal Distribution

(Standard) Normal Distribution:

```
# standard normal distribution
np.random.default_rng().standard_normal(size=5)
```

```
array([-0.63230721,  1.66399012,  0.91965747,  1.34380089, -0.63777862])
```

```
# normal distribution with mu=5 and sigma=2
np.random.default_rng().normal(5, 2, size=(2,3))
```

```
array([[5.87324574, 8.92706654, 3.02382205],
       [4.07345125, 3.05197676, 4.95142636]])
```

Array Attributes

Each NumPy array has useful attributes we can access:

```
x = np.arange(6, dtype=np.int16).reshape(2,3)
x.shape
```

```
(2, 3)
```

```
x.size # the total size of the array
```

```
6
```

```
x.dtype # the data type of the array
```

```
dtype('int16')
```

```
x.itemsize # size in bytes of each element
```

```
2
```

```
x.nbytes # total size in bytes of the array
```

```
12
```

```
x.itemsize * x.size == x.nbytes
```

```
True
```

Array Indexing

- NumPy array indexing, for *one-dimensional* arrays, works very similarly to Python lists
  - counting from zero, you can access the values by specifying the desired index in squared brackets
  - `numbers[0]` or `numbers[5]` for the NumPy array `numbers`
- for multidimensional arrays, you access the elements using comma-separated tuples of indices
  - `numbers[0,1]` or `numbers[3,2]` for a two-dimensional NumPy array
- you can modify values in your array this way as well
  - `numbers[0,1] = 12` set the value of the NumPy array `numbers` at index position `(0,1)` to `12`

Array Indexing

```
numbers = np.random.default_rng().integers(0,5,dtype=np.int64, size=(2,3))
numbers
```

```
array([[0, 0, 1],
       [3, 3, 3]])
```

```
numbers[0], numbers[1]
```

```
(array([0, 0, 1]), array([3, 3, 3]))
```

```
numbers[0,1]
```

```
0
```

Array Slicing: One-Dimensional Arrays

- you can access subarrays in your array using the *slice* notation `:`
- slicing in NumPy works like slicing in Python's list
    - `numbers[start:stop:step]`

```
x = np.arange(10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
x[:3], x[7:]
```

```
(array([0, 1, 2]), array([7, 8, 9]))
```

```
x[2:8:2], x[1::2]
```

```
(array([2, 4, 6]), array([1, 3, 5, 7, 9]))
```

Array Slicing: Multidimensional Arrays

```
x = np.arange(12).reshape(3,4)
x
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
x[:,2] # access only one specific column
```

```
array([ 2,  6, 10])
```

```
x[:2, ::2] # slicing works the same way, but now individually for each index
```

```
array([[0, 2],
       [4, 6]])
```

```
x[::-1, ::1]
```

```
array([[ 8,  9, 10, 11],
       [ 4,  5,  6,  7],
       [ 0,  1,  2,  3]])
```

Subarrays are no-copy views

- array slices return views of the original array data
  - this is different to Python lists

If you modify the subarray obtained by slicing, you will modify the original array.

```
x
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
x[:2,:2] = np.full((2,2), 8)
x
```

```
array([[ 8,  8,  2,  3],
       [ 8,  8,  6,  7],
       [ 8,  9, 10, 11]])
```

Array Concatenation

- you can combine multiple existing arrays into one by using
  - `np.vstack` – vertical stack
  - `np.hstack` – horizontal stack
  - `np.dstack` – stack along the third axis
  - `np.concatenate` – general method for stacking

Array Concatenation

```
x = np.array([[1,2,3], [4,5,6]])
y = np.array([[7,8,9], [5,5,5]])
```

```
stacked_x = np.concatenate([x,y], axis=0)
stacked_x
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9],
       [5, 5, 5]])
```

```
stacked_y = np.concatenate([x,y], axis=1)
stacked_y
```

```
array([[1, 2, 3, 7, 8, 9],
       [4, 5, 6, 5, 5, 5]])
```

Array Splitting

- array splitting works in NumPy with the following commands:
  - `np.split`
  - `np.hsplit`
  - `np.vsplit`

Array Splitting

```
stacked_y
```

```
array([[1, 2, 3, 7, 8, 9],
       [4, 5, 6, 5, 5, 5]])
```

```
np.split(stacked_y, 2, axis=1)
```

```
[array([[1, 2, 3],
        [4, 5, 6]]),
 array([[7, 8, 9],
        [5, 5, 5]])]
```

```
np.split(stacked_y, [1], axis=0)
```

```
[array([[1, 2, 3, 7, 8, 9]]), array([[4, 5, 6, 5, 5, 5]])]
```

Universal Functions

NumPy documentation on universal functions:

- a universal function is a function that operates on NumPy arrays in an element-by-element fashion

```
x = np.arange(3)
print(f"x = {x}")
print(f"x+2 = {x+2}")
print(f"x-2 = {x-2}")
print(f"x*2 = {x*2}")
print(f"x/2 = {x/2}")
print(f"x//2 = {x//2}")
```

```
x = [0 1 2]
x+2 = [2 3 4]
x-2 = [-2 -1  0]
x*2 = [0 2 4]
x/2 = [0.  0.5 1. ]
x//2 = [0 0 1]
```

Universal Functions: Operators

The operators are wrappers for specific universal functions:

Operator	ufunc	Description
+	np.add	addition
-	np.subtract	subtraction
-	np.negative	unary negation
*	np.multiply	multiplication
/	np.divide	division
//	np.floor_divide	floor division
**	np.power	exponentiation
%	np.mod	modulus / remainder

Overview of all available mathematical functions in NumPy.

Specifying the Output for ufuncs

- for large calculations it is often very useful to be able to specify the array where the result of the calculation will be stored
- you can do this using the `out` argument of the ufunc

```
x = np.arange(4).reshape(2,2)
y = np.empty((2,2))
np.multiply(x, 2, out=y)
x, y
```

```
(array([[0, 1],
        [2, 3]]),
 array([[0., 2.],
        [4., 6.]]))
```

This is more efficient than the operation:

```
y = x * 2
y
```

```
array([[0, 2],
       [4, 6]])
```

since this operation

- creates a temporary array to store the result of `x*2`
- copies this array into the `y` array

Broadcasting

- broadcasting allows operations to be performed on arrays of different sizes
- they follow a strict set of rules

Broadcasting rules:

Rule 1: if two arrays differ in the number of dimensions, the shape of the one with fewer dimensions is padded with ones on its left side

Rule 2: if the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape

Rule 3: if in any dimension the sizes disagree and neither is equal to 1, an error is raised

Broadcasting rules applied

```
x = np.ones((3,2))
y = np.arange(2)
print(x.shape, y.shape)
x + y

# rules 1 and 2 are used
```

```
(3, 2) (2,)

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

```
x = np.ones((3,2))
y = np.arange(3)
x + y

# this is rule 3
ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Comparison Operators

- in NumPy it is also possible to do element-to-element comparisons of two arrays
- the comparison operators are implemented as `ufuncs`

Operator	ufunc
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

Example: Comparison Operators

```
x = np.arange(5)
print(x)
x > 3
```

```
[0 1 2 3 4]
```

```
array([False, False, False, False,  True])
```

```
x == 3
```

```
array([False, False, False,  True, False])
```

```
x <= 3
```

```
array([ True,  True,  True,  True, False])
```

Boolean Masks

- boolean arrays can be very useful to implement filters and conditional counts
- the functions `np.any` and `np.all` can be used to check if any or all the values of an array satisify a certain condition

```
x
```

```
array([0, 1, 2, 3, 4])
```

```
np.any(x > 3)
```

```
True
```

```
np.all(x > 3)
```

```
False
```

Filters and Conditional Counts

```
x = np.random.randint(6, size=(3,2))
x
```

```
array([[3, 5],
       [4, 0],
       [5, 4]])
```

```
np.sum(x > 2)
```

```
5
```

```
np.sum(x> 2, axis=1)
```

```
array([2, 1, 2])
```

```
np.sum((x>2) & (x<4)), x[(x>2) & (x<4)]
```

```
(1, array([3]))
```

Fancy Indexing

- instead of using single scalars as indices we can pass arrays of indices to access and modify complicated subsets of an array's values

```
x = np.random.default_rng().integers(0,30,dtype=np.int32, size=8)
x
```

```
array([19, 13, 19,  0, 14, 22,  2, 14], dtype=int32)
```

```
x[[3,5,4]]
```

```
array([ 0, 22, 14], dtype=int32)
```

```
ind = np.array([[2,3,3],[5,3,6]])
y = x[ind]
y
```

```
array([[19,  0,  0],
       [22,  0,  2]], dtype=int32)
```

```
y[[0,1], [1,2]]
```

```
array([0, 2], dtype=int32)
```

```
y[1, [0,1]]
```

```
array([22,  0], dtype=int32)
```

Sorting

- if we work with NumPy arrays, we should use NumPy's sorting functions
- they are much *more efficient* than Python's `sort` and `sorted` functions that work with lists
- to return a sorted array *without modifying the input*, you can use

```
x = np.array([8,7,6,1,2])
np.sort(x)
```

```
array([1, 2, 6, 7, 8])
```

- if you prefer sorting in-place, then use

```
x.sort()
x
```

```
array([1, 2, 6, 7, 8])
```

- `argsort` returns the *indices* of the sorted elements

```
x = np.array([8,7,6,1,2])
ind = np.argsort(x)
ind
```

```
array([3, 4, 2, 1, 0])
```

- you can also sort along specific rows or columns of a multidimensional array:

```
x = np.random.randint(10, size=(5,2))
np.sort(x, axis=0)
```

```
array([[3, 0],
       [3, 3],
       [4, 4],
       [7, 4],
       [9, 6]])
```

Partitioning

- if we are not interested in sorting the whole array, but want to find the s smallest values in the array, we can use `np.partition`
- `np.partition` uses an array and a number s as input and outputs an array with the smallest s values to the left of the partition and the remaining values to the right

```
x = np.array([7, 1, 7, 7, 1, 5, 7, 2, 3, 2, 6, 2, 3, 0])
```

```
np.partition(x, 4)
```

```
array([0, 1, 2, 1, 2, 5, 2, 3, 3, 6, 7, 7, 7, 7])
```

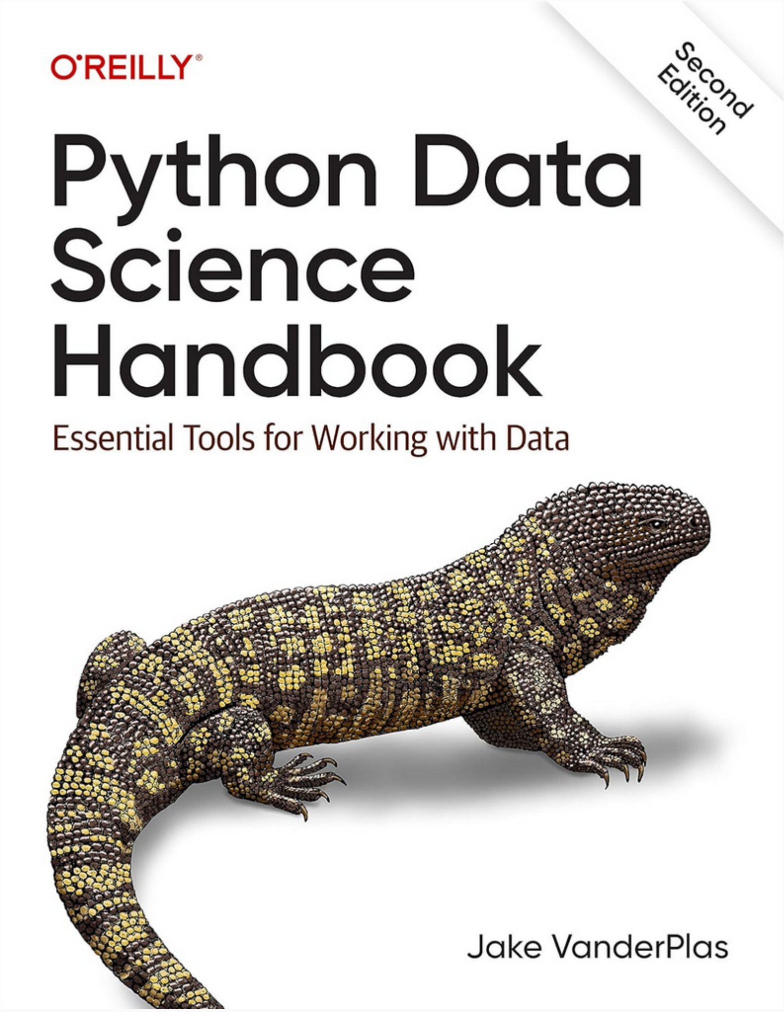
```
np.argpartition(x, 4)
```

```
array([13,  4,  7,  1,  9,  5, 11, 12,  8, 10,  0,  3,  2,  6])
```

Further Reading: Utilizing your GPU with CuPy

- read about CuPy, especially the user guide about
  - Basics of CuPy
  - Memory Management
  - Performance Best Practices
  - Difference between CuPy and NumPy
- try to port a few of the algorithms we implementend with NumPy to CuPy (on a machine with an NVIDIA GPU – K019); Do you see a performance improvement?

Literature


O'REILLY
Second Edition
Python Data Science Handbook
Essential Tools for Working with Data
Jake VanderPlas