Clase 015 — NumPy: ufuncs y vectorización

Parte: 0 — Prerrequisitos · Fuente: VanderPlas, cap. 2, § 2.3 Computation on NumPy Arrays: Universal Functions. · Duración estimada: 75 min.

Clase 015 — NumPy: ufuncs y vectorización

> *Parte: 0 — Prerrequisitos · Fuente: VanderPlas, cap. 2, § 2.3 Computation on NumPy Arrays: Universal Functions. · Duración estimada: 75 min.*

Objetivo

Que el alumno abandone los for loops sobre arrays NumPy y use ufuncs (universal functions) para operaciones elementwise — la fuente real del speedup. Ufuncs son C compilado vectorizado; un for Python sobre array es lo peor de ambos mundos.

Resultados de aprendizaje

Al finalizar la clase, el alumno podrá:

1. Identificar una ufunc (np.add, np.multiply, np.sin, np.exp, np.log, comparadores).
2. Reemplazar un for+append por una expresión vectorizada y medir el speedup.
3. Usar el parámetro out= para escribir el resultado in-place (evita allocar memoria extra).
4. Combinar ufuncs con operadores aritméticos (+, -, , /, *).
5. Reconocer las trampas de la vectorización (overflow, NaN propagación, división por cero).

Temas

#	Tema	Por qué importa
1	¿Qué es una ufunc?	Función C vectorizada elementwise.
2	Ufuncs unarias y binarias	np.exp(x) vs np.add(x, y).
3	Operadores → ufuncs	a + b ≡ np.add(a, b).
4	out= para in-place	Memoria O(1) extra.
5	Trampas: overflow, NaN, inf, división por	NumPy avisa pero no para.
6	np.where(cond, a, b)	Ternario vectorizado.

Definiciones y características

Ufunc (universal function)

: Función NumPy implementada en C que opera elementwise y vectorizada (SIMD cuando posible). Características: rápida (10-100× vs Python), broadcasting automático, soporta out= para in-place.

Vectorización

: Operar sobre arrays completos en vez de loops Python: arr 2 en vez de [x2 for x in arr]. La operación corre en C compilado sobre memoria contigua, sin overhead del intérprete por elemento.

In-place (out=)

: Escribir el resultado de una ufunc en un array existente, sin allocar memoria nueva: np.multiply(a, 2, out=a). Útil con arrays grandes donde la copia temporal duplicaría la memoria pico.

Propagación de NaN

: Cualquier operación que tenga NaN como input produce NaN. np.array([1, np.nan, 3]).sum() → nan. Para ignorar usa variantes nan*: nansum, nanmean, nanmedian.

np.where(cond, a, b)

: Ternario vectorizado: para cada elemento, si cond es True usa a, si no usa b. Equivale a [a if c else b for c, a, b in zip(cond, a, b)] pero ~100× más rápido.

Dataset / recursos

Sintético: arrays grandes para benchmark. Sin descarga.

Ejercicios

1. Benchmark. Calcula [xx + 2x + 1 for x in range(1_000_000)] vs arrarr + 2arr + 1. Mide con %timeit.

2. Logaritmo y exponencial. Con np.exp y np.log, verifica que log(exp(x)) ≈ x para 1000 valores. Reporta el error máximo.

3. In-place vs alloc. arr = arr * 2 + 1 vs np.multiply(arr, 2, out=arr); np.add(arr, 1, out=arr). Compara tracemalloc.

4. np.where ternario. Dado un array de notas, crea otro array con 'aprobado' si nota >= 4, 'reprobado' si no.

5. Trampa NaN. Crea np.array([1, 2, np.nan, 4]).sum() y .mean(). Compara con np.nansum y np.nanmean.

Homework verificable

Notebook: (a) reescribe 3 loops como expresiones vectorizadas + tabla con %timeit (3 N distintos); (b) demuestra out= con tracemalloc; (c) usa np.where para clasificar datos; (d) maneja NaN con nansum/nanmean y compara con propagación.

Criterio de aceptación: Speedup >50× en N=1M. out= muestra memoria ≈ 0 extra. NaN-handling correcto.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
for i in range(len(arr)): arr[i] = ... es	Loops Python sobre array NumPy = lo peor d
RuntimeWarning: divide by zero / invalid v	NumPy avisa pero no para: 1/0 → inf, 0/0 →
out= con dtype incompatible	np.add(int_arr, 0.5, out=int_arr) falla —
Resultado de np.where no es lo esperado	Los 3 args se evalúan completos: np.where(
arr.sum() da NaN y no sé por qué	Hay un NaN escondido en el array. Fix: pri

Preguntas frecuentes

¿Cuánto más rápido es vectorizar?

Típicamente 50-100× para arrays de 1M elementos. Para arrays pequeños (<100), la ganancia es menor o nula (overhead constante). Mide con %timeit, no asumas.

¿arr + 1 o np.add(arr, 1)?

Equivalentes. Operadores son sintaxis dulce sobre ufuncs. Usa np.add(...) cuando necesitas out= (in-place) o where= (mask).

¿NumPy aprovecha mi GPU?

No — solo CPU. Para GPU: CuPy (drop-in replacement), PyTorch tensors, JAX. NumPy 2 está mejorando vectorización CPU (SIMD wider, BLAS) pero sigue siendo CPU.

¿Por qué arr 2 es más rápido que arr arr?*

Suelen empatar (** también es ufunc). Para potencias enteras pequeñas (2, 3), NumPy a veces usa atajos. Mide con %timeit en tu caso específico.

¿np.where o boolean mask?

Mask (arr[cond] = valor) si vas a modificar in-place o filtrar (arr[arr>0]). np.where(cond, a, b) si necesitas un array nuevo con dos valores posibles según condición.

Referencias

- VanderPlas, cap. 2 § 2.3 Computation on NumPy Arrays.
- NumPy ufuncs reference

Siguiente clase

Clase 016 — NumPy: agregaciones

Apéndice: notebook (primer bloque)

Primera celda ejecutable del notebook de la clase.

```
import numpy as np
import time, tracemalloc
rng = np.random.default_rng(42)
```

Archivos complementarios

- notebook.ipynb