Redes Neuronales (2025)

Ayudas e indicaciones para el Trabajo Práctico N°2 (Trabajo en progreso)[*]

Índice

[*]Reportar errores a: tristan.osan@unc.edu.ar

1. Resolución de ejercicios con NumPy

La librería `NumPy` es la base fundamental para el trabajo numérico en Python. Permite manipular arreglos multidimensionales (`arrays`), realizar operaciones vectorizadas y ejecutar cálculos matemáticos de forma eficiente.

En esta sección veremos ejemplos prácticos que ilustran su uso.

1.1. Crear un vector y calcular estadísticas básicas

El siguiente código crea un vector de enteros del 1 al 10 y calcula su media, desviación estándar y suma total.

```
import numpy as np

# Crear un vector de 1 a 10
v = np.arange(1, 11)

# Calcular estadísticas
media = np.mean(v)
desviacion = np.std(v)
suma_total = np.sum(v)

print("Vector:", v)
print("Media:", media)
print("Desviación estándar:", desviacion)
print("Suma total:", suma_total)
```

Resultado esperado:

```
Vector: [ 1  2  3  4  5  6  7  8  9 10]
Media: 5.5
Desviación estándar: 2.8722813232690143
Suma total: 55
```

1.2. Multiplicación de matrices

NumPy facilita el trabajo con matrices y su multiplicación.

```
# Crear dos matrices 2x2
A = np.array([[1, 2],
              [3, 4]])

B = np.array([[5, 6],
              [7, 8]])

# Multiplicación matricial
C = np.dot(A, B)

print("A * B =\n", C)
```

Resultado esperado:

```
A * B =
 [[19 22]
  [43 50]]
```

1.3. Uso de funciones matemáticas en arrays

```
# Crear un array de ángulos en radianes
angulos = np.array([0, np.pi/4, np.pi/2, np.pi])

# Calcular seno y coseno de cada elemento
senos = np.sin(angulos)
cosenos = np.cos(angulos)

print("Ángulos:", angulos)
print("Senos:", senos)
print("Cosenos:", cosenos)
```

Resultado esperado:

```
Ángulos: [0.         0.78539816 1.57079633 3.14159265]
Senos: [0.         0.70710678 1.         0.        ]
Cosenos: [ 1.          0.70710678  0.         -1.        ]
```

1.4. Indexación y segmentación

```
# Vector del 1 al 10
v = np.arange(1, 11)

# Elementos del 3 al 7
sub_v = v[2:7]

# Elementos pares
pares = v[v % 2 == 0]

print("Subvector (3 a 7):", sub_v)
print("Elementos pares:", pares)
```

Resultado esperado:

```
Subvector (3 a 7): [3 4 5 6 7]
Elementos pares: [ 2  4  6  8 10]
```

1.5. Ejercicio 1: Importando Librerías

Para poder usar las funciones de bibliotecas externas, primero debemos importarlas. Usamos la declaración `import`. A menudo, se les asigna un **alias** (un nombre más corto) para que sea más fácil y rápido llamarlas en el futuro.

Solución y Explicación

```
# Importamos la librería NumPy y le asignamos el alias 'np'
import numpy as np

# Importamos el módulo de álgebra lineal (linalg) de la librería SciPy
from scipy import linalg
```

```
# Importamos el módulo pyplot de Matplotlib y le asignamos el alias '
   plt'
import matplotlib.pyplot as plt
```

- `import numpy as np`: Es la convención estándar en la comunidad de Python para importar NumPy. Ahora podemos acceder a todas las funciones de NumPy usando el prefijo `np.`.
- `from scipy import linalg`: De la biblioteca SciPy, que es muy grande, solo importamos el módulo `linalg`, que contiene funciones específicas de álgebra lineal.
- `import matplotlib.pyplot as plt`: Importamos el módulo `pyplot` de Matplotlib, que es una colección de funciones para crear gráficos y visualizaciones, y le asignamos el alias `plt`.

1.6. Ejercicio 2: Creando Arrays

La forma más común de crear un array en NumPy es a partir de una lista de Python, usando la función `np.array()`. NumPy inferirá automáticamente el tipo de dato (entero, flotante, etc.), aunque también podemos especificarlo.

1.6.1. 1) Crear un array de 1 dimensión

Pasamos una lista simple a la función `np.array()`. El resultado es un objeto array de NumPy con una sola dimensión (un vector).

```
# Creamos una lista de Python
lista_a = [1, 2, 3]

# Creamos un array de NumPy a partir de la lista
a = np.array(lista_a)

# Podemos verificar su contenido y tipo de dato
print(a)
# Salida: [1 2 3]

print(a.dtype)
# Salida: int64 (indica que es un entero de 64 bits)
```

1.6.2. 2) Crear un array de 2 dimensiones (Matriz)

Para crear un array de 2 dimensiones (una matriz), usamos una lista que contiene otras listas, donde cada lista interna representa una **fila** de la matriz.

```
# Creamos una lista de listas
lista_b = [[1.5, 2, 3], [4, 5, 6]]

# Creamos el array bidimensional
b = np.array(lista_b)
```

```
print(b)
# Salida:
# [[1.5 2.  3. ]
#  [4.  5.  6. ]]

print(b.ndim)
# Salida: 2 (indica que tiene 2 dimensiones)
```

Nota: Como uno de los números es `1.5` (un flotante), NumPy convierte todos los elementos del array a tipo flotante (`float64`) para mantener la consistencia.

1.6.3. 3) Crear un array de 3 dimensiones (Tensor)

La estructura es una lista que contiene "matrices". El atributo `.shape` nos da una tupla con las dimensiones del array. En este caso, `(2, 2, 3)` nos dice que tenemos **2** matrices, cada una con **2** filas y **3** columnas.

```
# Creamos una lista de listas de listas
lista_c = [[[1.5, 2, 3], [4, 5, 6]], [[3, 2, 1], [4, 5, 6]]]

# Creamos el array tridimensional
c = np.array(lista_c)

print(c)
# Salida:
# [[[1.5 2.  3. ]
#   [4.  5.  6. ]]
#
#  [[3.  2.  1. ]
#   [4.  5.  6. ]]]

print(c.shape)
# Salida: (2, 2, 3)
```

1.7. Ejercicio 3: Inicializando Arrays

NumPy ofrece funciones optimizadas para crear arrays con valores iniciales específicos, como ceros, unos o secuencias numéricas, sin necesidad de crear primero una lista de Python.

1.7.1. 1) Crear un array de ceros con `np.zeros`

Esta función crea un array con las dimensiones especificadas, rellenando todas sus entradas con el valor `0.0`. Por defecto, el tipo de dato es flotante (‘float‘).

```
import numpy as np

# Crea un array de 3 filas y 4 columnas lleno de ceros
zeros_array = np.zeros((3, 4))

print(zeros_array)
# Salida:
```

```
# [[0. 0. 0. 0.]
#  [0. 0. 0. 0.]
#  [0. 0. 0. 0.]]
```

1.7.2. 2) Crear un array de unos con `np.ones`

Similar a 'zeros', pero rellena el array con el valor `1`. Podemos especificar el tipo de dato ('dtype') para que sean enteros en lugar de flotantes.

```
# Crea un array 3D de 2x3x4 lleno de unos de tipo entero
ones_array = np.ones((2, 3, 4), dtype=np.int64)

print(ones_array)
# Salida:
# [[[1 1 1 1]
#   [1 1 1 1]
#   [1 1 1 1]]
#
#  [[1 1 1 1]
#   [1 1 1 1]
#   [1 1 1 1]]]
```

1.7.3. 3) Crear una secuencia con `np.arange`

Crea valores equiespaciados dentro de un intervalo dado. Es similar a la función 'range' de Python, pero devuelve un array de NumPy. La sintaxis es `arange(inicio, fin, paso)`. El intervalo no incluye el valor 'fin'.

```
# Crea un array con valores de 10 a 24, incrementando de 5 en 5
d = np.arange(10, 25, 5)

print(d)
# Salida: [10 15 20]
```

1.7.4. 4) Crear una secuencia con `np.linspace`

Crea un número específico de valores equiespaciados entre un `inicio` y un `fin`. A diferencia de 'arange', 'linspace' sí incluye el valor 'fin'.

```
# Crea 9 valores equiespaciados entre 0 y 9 (ambos inclusive)
linspace_array = np.linspace(0, 9, 9)

print(linspace_array)
# Salida: [0.   1.125  2.25  3.375  4.5  5.625  6.75  7.875  9. ]
```

1.7.5. 5) Crear un array con un valor constante con `np.full`

Crea un array de una forma determinada y lo rellena completamente con un valor que nosotros especifiquemos.

```
# Crea un array de 2x2 donde cada elemento es el entero 7
e = np.full((2, 2), 7)

print(e)
# Salida:
# [[7 7]
#  [7 7]]
```

1.7.6. 6) Crear una matriz identidad con `np.eye`

Crea una matriz cuadrada (mismo número de filas que de columnas) con unos en la diagonal principal y ceros en el resto.

```
# Crea una matriz identidad de 2x2
f = np.eye(2)

print(f)
# Salida:
# [[1. 0.]
#  [0. 1.]]
```

1.7.7. 7) Crear un array con valores aleatorios con `np.random.random`

Crea un array con la forma dada y lo rellena con valores aleatorios extraídos de una distribución uniforme en el intervalo $[0{,}0, 1{,}0)$.

```
# Crea un array de 2x2 con valores aleatorios
random_array = np.random.random((2, 2))

print(random_array)
# Salida (los valores serán diferentes cada vez):
# [[0.18758871 0.35434277]
#  [0.69429532 0.52229517]]
```

1.7.8. 8) Crear un array vacío con `np.empty`

Crea un array de la forma especificada sin inicializar sus entradas a ningún valor en particular. Los valores que contiene son "basura", es decir, lo que sea que estuviera en esa posición de la memoria en ese momento. Es ligeramente más rápido que ‘zeros‘ u ‘ones‘, pero debe usarse con cuidado, asegurándose de llenar cada elemento después.

```
# Crea un array de 3x2 sin inicializar
empty_array = np.empty((3, 2))

print(empty_array)
# Salida (los valores serán impredecibles y pueden variar):
# [[2.12199579e-314 2.12199579e-314]
#  [2.12199579e-314 0.00000000e+000]
#  [0.00000000e+000 0.00000000e+000]]
```

1.8. Ejercicio 4: Inspeccionando Arrays

Una vez que tenemos un array, NumPy nos da varios atributos y funciones para obtener información sobre su estructura y sus datos sin tener que imprimir el array completo.

Listing 1: Definición de arrays para el ejercicio

```
import numpy as np

# Arrays del Ejercicio 2 y 3
a = np.array([1, 2, 3])
b = np.array([[1.5, 2, 3], [4, 5, 6]])
e = np.full((2, 2), 7)
```

1.8.1. 1) Dimensiones de un array con `.shape`

El atributo `.shape` devuelve una tupla que contiene el tamaño del array para cada una de sus dimensiones.

```
# Obtenemos la tupla con las dimensiones de 'a'
dimensiones_a = a.shape

print(dimensiones_a)
# Salida: (3,)
```

Explicación: La salida `(3,)` indica que `a` es un array de una sola dimensión que tiene 3 elementos.

1.8.2. 2) Largo de un array con `len()`

Para arrays de una dimensión, la función `len()` de Python devuelve el número de elementos. Para arrays multidimensionales, devuelve el tamaño de la primera dimensión (el número de filas).

```
# Obtenemos el largo de 'a'
largo_a = len(a)

print(largo_a)
# Salida: 3
```

1.8.3. 3) Número de dimensiones con `.ndim`

El atributo `.ndim` devuelve un número entero que representa cuántos ejes o dimensiones tiene el array.

```
# Obtenemos el numero de dimensiones de 'b'
num_dims_b = b.ndim

print(num_dims_b)
# Salida: 2
```

Explicación: El resultado es `2` porque `b` es una matriz con filas y columnas.

1.8.4. 4) Número total de elementos con `.size`

El atributo `.size` devuelve el número total de elementos que contiene el array. Es el producto de los números en la tupla de `.shape`.

```
# Obtenemos el numero total de entradas en 'e'
total_elementos_e = e.size

print(total_elementos_e)
# Salida: 4
```

Explicación: El array `e` tiene dimensiones (2,2), por lo que su tamaño total es $2 \times 2 = 4$.

1.8.5. 5) Tipo de dato con `.dtype`

El atributo `.dtype` devuelve un objeto que describe el tipo de dato de los elementos del array.

```
# Obtenemos el tipo de dato del array 'b'
tipo_b = b.dtype

print(tipo_b)
# Salida: float64
```

Explicación: NumPy eligió `float64` (un número de punto flotante de 64 bits) para `b` porque uno de sus elementos iniciales era `1.5`.

1.8.6. 6) Nombre del tipo de dato con `.dtype.name`

Podemos obtener el nombre del tipo de dato como una cadena de texto accediendo al atributo `.name` del objeto `dtype`.

```
# Obtenemos el nombre del tipo de dato de 'b'
nombre_tipo_b = b.dtype.name

print(nombre_tipo_b)
# Salida: float64
```

1.8.7. 7) Convertir el tipo de dato con `.astype()`

El método `.astype()` crea una **nueva copia** del array, convirtiendo cada elemento al tipo de dato especificado. No modifica el array original.

```
# Creamos un nuevo array 'b_int' convirtiendo 'b' a enteros
b_int = b.astype(np.int64)

print(b)
# Salida (el original no cambia):
# [[1.5 2.  3. ]
#  [4.  5.  6. ]]

print(b_int)
# Salida (el nuevo array con enteros):
# [[1 2 3]
#  [4 5 6]]
```

Explicación: Al convertir de flotante a entero, NumPy trunca la parte decimal. Por eso, `1.5` se convierte en `1`.

1.9. Ejercicio 5: Tipos de Datos

NumPy soporta una gran variedad de tipos de datos numéricos que se pueden especificar al crear un array. Esto es crucial para la eficiencia, especialmente cuando se trabaja con grandes volúmenes de datos. La forma más directa de especificar el tipo es usando el argumento `dtype`.

1.9.1. Crear un array de tipo `np.int64`

Este tipo de dato representa números enteros de 64 bits, lo que permite almacenar un rango de valores muy amplio. Es el tipo de entero por defecto en la mayoría de los sistemas de 64 bits.

```
import numpy as np

# Creamos el array especificando el tipo de dato
int_array = np.array([1, 5, -10], dtype=np.int64)

print(int_array)
# Salida: [  1   5 -10]

print(int_array.dtype)
# Salida: int64
```

1.9.2. Crear un array de tipo `np.float32`

Este tipo representa números de punto flotante de 32 bits (precisión simple). Ocupan menos memoria que los `float64` (precisión doble) por defecto, lo cual puede ser útil para arrays muy grandes.

```
# Creamos el array especificando el tipo de dato
float_array = np.array([1.0, 3.14, 2.71], dtype=np.float32)

print(float_array)
# Salida: [1.    3.14  2.71]

print(float_array.dtype)
# Salida: float32
```

1.9.3. Crear un array de tipo `np.complex128`

Este tipo permite almacenar números complejos, que tienen una parte real y una parte imaginaria. El número `128` indica que se usan 128 bits para almacenar cada número (64 para la parte real y 64 para la imaginaria). La parte imaginaria se denota con una `j`.

```
# Creamos el array con numeros complejos
complex_array = np.array([1+2j, 3-4j, 5.5j], dtype=np.complex128)
```

```
print(complex_array)
# Salida: [1.+2.j 3.-4.j 0.+5.5j]

print(complex_array.dtype)
# Salida: complex128
```

1.9.4. Crear un array de tipo `np.bool`

Este tipo de dato almacena valores booleanos: `True` o `False`. En NumPy, estos valores se representan internamente como 1 y 0, respectivamente.

```
# Creamos el array de booleanos
bool_array = np.array([True, False, False, True], dtype=np.bool_)

print(bool_array)
# Salida: [ True False False  True]

print(bool_array.dtype)
# Salida: bool
```

1.9.5. Crear un array de tipo `np.object`

Este es un tipo de dato flexible que permite que el array almacene punteros a cualquier objeto de Python. Esto le quita las ventajas de rendimiento a NumPy, pero es útil para contener datos heterogéneos.

```
# Creamos un array con diferentes tipos de objetos de Python
object_array = np.array([1, "Python", True, [2, 3]], dtype=np.object_)

print(object_array)
# Salida: [1 'Python' True list([2, 3])]

print(object_array.dtype)
# Salida: object
```

1.9.6. Crear un array de tipo `np.string_`

Este tipo almacena cadenas de bytes de longitud fija (como las cadenas de texto en C). Al crear el array, NumPy asignará una longitud fija basada en la cadena más larga de la lista. Se denota con una 'S' seguida del número de bytes.

```
# Creamos un array de strings (bytes)
string_array = np.array(['Hola', 'Mundo'], dtype=np.string_)

print(string_array)
# Salida: [b'Hola' b'Mundo']

print(string_array.dtype)
# Salida: S5 (la 'b' indica bytes, 'S5' significa cadena de 5 bytes)
```

1.9.7. Crear un array de tipo `np.unicode_`

Este tipo almacena cadenas de texto Unicode de longitud fija, que es el estándar moderno para texto y soporta caracteres de múltiples idiomas. Se denota con una 'U' seguida del número de caracteres.

```
# Creamos un array de strings Unicode
unicode_array = np.array(['NumPy', 'Tutorial', 'Ciencia'], dtype=np.
    unicode_)

print(unicode_array)
# Salida: ['NumPy' 'Tutorial' 'Ciencia']

print(unicode_array.dtype)
# Salida: U8 (la cadena más larga, 'Tutorial', tiene 8 caracteres)
```

1.10. Ejercicio 6: Operando sobre Arrays

NumPy permite realizar operaciones matemáticas sobre arrays completos de forma rápida y eficiente. Las operaciones aritméticas ('+', '-', '*', '/') se realizan, por defecto, **punto a punto** (element-wise). Para operaciones más complejas como el producto matricial, existen funciones específicas.

Listing 2: Definición de arrays para el ejercicio

```
import numpy as np

# Arrays de ejercicios anteriores
a = np.array([1, 2, 3])
b = np.array([[1.5, 2, 3], [4, 5, 6]])
e = np.full((2, 2), 7)
f = np.eye(2)
```

1.10.1. Resta de arrays con el operador -

```
# Restamos a de b. NumPy aplica 'a' a cada fila de 'b'.
g = b - a

print(g)
# Salida:
# [[0.5 0.  0. ]
#  [3.  3.  3. ]]
```

1.10.2. Explicación del fenómeno de Broadcasting

Aunque `a` (shape: `(3,)`) y `b` (shape: `(2, 3)`) tienen dimensiones distintas, NumPy puede realizar la operación gracias a un mecanismo llamado **Broadcasting**. Las reglas de Broadcasting permiten operar con arrays de diferentes formas si son compatibles. En este caso:

1. NumPy compara las dimensiones de los arrays de derecha a izquierda. La última dimensión de ambos es 3, por lo que son compatibles.

2. Al avanzar a la siguiente dimensión, el array `b` tiene una dimensión de tamaño 2, mientras que `a` no tiene más dimensiones.

3. Broadcasting .estira.º "duplica"virtualmente el array `a` a lo largo de la dimensión que le falta para que su forma coincida con la de `b`. Es como si la operación fuera: `[[1.5, 2, 3], [4, 5, 6]] - [[1, 2, 3], [1, 2, 3]]`.

Este mecanismo evita crear copias innecesarias de los datos en memoria, haciendo las operaciones muy eficientes.

1.10.3. Comparación con `np.subtract`

La función universal `np.subtract()` es el equivalente explícito del operador `-`. El resultado es idéntico.

```
g_func = np.subtract(b, a)
print(g_func)
# Salida (idéntica a la anterior):
# [[0.5 0.  0. ]
#  [3.  3.  3. ]]
```

1.10.4. Suma de arrays con el operador `+`

La suma funciona de la misma manera, aplicando Broadcasting. (Nota: Se asumió que la pregunta quería calcular a+b).

```
suma_op = b + a
print(suma_op)
# Salida:
# [[2.5 4.  6. ]
#  [5.  7.  9. ]]
```

1.10.5. Comparación con `np.add`

La función `np.add()` es el equivalente explícito del operador `+`.

```
suma_func = np.add(b, a)
print(suma_func)
# Salida (idéntica a la anterior):
# [[2.5 4.  6. ]
#  [5.  7.  9. ]]
```

1.10.6. División punto a punto con `/`

```
division_op = b / a
print(division_op)
# Salida:
# [[1.5  1.   1.  ]
#  [4.   2.5  2.  ]]
```

1.10.7. Comparación con `np.divide`

La función `np.divide()` es el equivalente explícito del operador `/`.

```
division_func = np.divide(b, a)
print(division_func)
# Salida (idéntica a la anterior):
# [[1.5  1.   1.  ]
#  [4.   2.5  2.  ]]
```

1.10.8. Multiplicación punto a punto con `*`

```
mult_op = b * a
print(mult_op)
# Salida:
# [[ 1.5  4.   9. ]
#  [ 4.  10.  18. ]]
```

1.10.9. Comparación con `np.multiply`

La función `np.multiply()` es el equivalente explícito del operador `*`.

```
mult_func = np.multiply(b, a)
print(mult_func)
# Salida (idéntica a la anterior):
# [[ 1.5  4.   9. ]
#  [ 4.  10.  18. ]]
```

1.10.10. Exponencial punto a punto con `np.exp`

Calcula e^x para cada elemento x en el array.

```
exp_b = np.exp(b)
print(exp_b)
# Salida:
# [[  4.48168907   7.3890561   20.08553692]
#  [ 54.59815003 148.4131591  403.42879349]]
```

1.10.11. Raíz cuadrada punto a punto con `np.sqrt`

```
sqrt_b = np.sqrt(b)
print(sqrt_b)
# Salida:
# [[1.22474487 1.41421356 1.73205081]
#  [2.         2.23606798 2.44948974]]
```

1.10.12. Seno punto a punto con `np.sin`

```
sin_a = np.sin(a)
print(sin_a)
# Salida: [0.84147098 0.90929743 0.14112001]
```

1.10.13. Coseno punto a punto con `np.cos`

(Nota: Se asumió que la pregunta quería calcular el coseno, no el seno de b).

```
cos_b = np.cos(b)
print(cos_b)
# Salida:
# [[ 0.0707372  -0.41614684 -0.9899925 ]
#  [-0.65364362 -0.83907153 -0.9665434 ]]
```

1.10.14. Logaritmo natural punto a punto con `np.log`

```
log_a = np.log(a)
print(log_a)
# Salida: [0.         0.69314718 1.09861229]
```

1.10.15. Producto punto con el método `.dot`

Para arrays de 2 dimensiones, `.dot()` realiza la multiplicación de matrices.

```
dot_method = e.dot(f)
print(dot_method)
# Salida:
# [[7. 0.]
#  [7. 0.]]
```

1.10.16. Comparación con `np.dot`

La función `np.dot(a, b)` es equivalente al método `a.dot(b)`.

```
dot_func = np.dot(e, f)
print(dot_func)
# Salida (idéntica a la anterior):
# [[7. 0.]
#  [7. 0.]]
```

1.10.17. Opciones del producto punto en NumPy

El comportamiento de `np.dot(a, b)` depende de las dimensiones de `a` y `b`:

- **Si ambos son 1D (vectores):** Devuelve un número escalar, que es el producto interno (o producto punto) de los vectores. Ej: `np.dot([1,2], [3,4])` es $1 * 3 + 2 * 4 = 11$.

- **Si ambos son 2D (matrices):** Devuelve una matriz, resultado de la multiplicación de matrices convencional. El número de columnas de la primera matriz debe ser igual al número de filas de la segunda.
- **Si uno es un escalar y el otro un array:** Devuelve un array donde cada elemento ha sido multiplicado por el escalar. Es equivalente al operador $*$.
- **Si uno es N-D y el otro es 1D:** Suma el producto sobre el último eje del array N-D.
- **Si ambos son N-D:** El producto se suma sobre el último eje del primer array y el penúltimo eje del segundo. Esto es una generalización de la multiplicación de matrices. Para estas operaciones, a menudo es más claro usar `np.matmul` o el operador `@`, que se comportan de manera más predecible con arrays de más de 2 dimensiones.

1.11. Ejercicio 7: Comparando Arrays

NumPy permite comparar arrays completos de manera eficiente, lo cual es fundamental para el análisis de datos y la lógica condicional.

Listing 3: Definición de arrays para el ejercicio

```
import numpy as np

# Arrays de ejercicios anteriores
a = np.array([1, 2, 3])
b = np.array([[1.5, 2, 3], [4, 5, 6]])
c = np.array([[[1.5, 2, 3], [4, 5, 6]], [[3, 2, 1], [4, 5, 6]]])
```

1.11.1. Comparación punto a punto con ==

El operador == realiza una comparación punto a punto. Al igual que con las operaciones aritméticas, se aplican las reglas de **Broadcasting**. El resultado es un nuevo array de tipo booleano (`True`/`False`).

```
# Compara cada elemento de 'a' con los de cada fila de 'b'
comp_ab = (a == b)

print(comp_ab)
# Salida:
# [[False  True  True]
#  [False False False]]
```

Explicación: El array `a` se compara con la primera fila de `b`, resultando en `[False, True, True]`, y luego con la segunda fila, resultando en `[False, False, False]`.

1.11.2. Comparación de un array con un escalar con $<$

También podemos comparar cada elemento de un array con un único valor escalar.

```
# Compara si cada elemento de 'b' es menor que 2
comp_b_scalar = (b < 2)

print(comp_b_scalar)
```

```
# Salida:
# [[ True False False]
#  [False False False]]
```

1.11.3. Verificar igualdad total con `np.array_equal`

Esta función comprueba si dos arrays son idénticos en **forma y contenido**. Devuelve un único valor booleano. Es más estricta que el operador ==.

```
# Comprueba si a y b son completamente iguales
son_iguales = np.array_equal(a, b)

print(son_iguales)
# Salida: False
```

Explicación: La función devuelve `False` inmediatamente porque `a.shape` es `(3,)` y `b.shape` es `(2, 3)`, por lo que no tienen la misma forma.

1.12. Ejercicio 8: Copiando Arrays

Es muy importante entender la diferencia entre una **vista** (shallow copy) y una **copia** (deep copy) de un array para evitar modificar datos por accidente.

1.12.1. Crear una vista con `.view()`

Una vista de un array es un nuevo objeto array que mira a los **mismos datos** que el array original. Si modificas la vista, el original también cambia.

```
# Creamos una vista de 'a'
h = a.view()
print(f"Original a: {a}")
print(f"Vista h:    {h}")

# Modificamos un elemento de la vista 'h'
h[0] = 99

print(f"Vista h modificada: {h}")
print(f"Original a DESPUES de modificar h: {a}")
# Salida:
# Original a: [1 2 3]
# Vista h:    [1 2 3]
# Vista h modificada: [99  2  3]
# Original a DESPUES de modificar h: [99  2  3]
```

1.12.2. Crear una copia con `.copy()`

Una copia de un array crea un nuevo objeto array con una **copia completa de los datos originales**. Modificar la copia **no** afecta al original.

```
# Reseteamos 'a' a su valor original
a = np.array([1, 2, 3])
```

```
# Creamos una copia profunda de 'a'
copia_a = a.copy()
print(f"Original a: {a}")
print(f"Copia:      {copia_a}")

# Modificamos la copia
copia_a[0] = 99

print(f"Copia modificada: {copia_a}")
print(f"Original a DESPUES de modificar la copia: {a}")
# Salida:
# Original a: [1 2 3]
# Copia:      [1 2 3]
# Copia modificada: [99  2  3]
# Original a DESPUES de modificar la copia: [1 2 3]
```

1.13. Ejercicio 9: Ordenado

1.13.1. Ordenar un array con `.sort()`

El método `.sort()` ordena un array **en el mismo lugar (in-place)**, es decir, modifica el array original y no devuelve nada.

```
# Creamos un array desordenado para el ejemplo
a_desordenado = np.array([3, 1, 2])
print(f"Array original: {a_desordenado}")

# Ordenamos el array. La operacion modifica a_desordenado directamente
a_desordenado.sort()

print(f"Array ordenado: {a_desordenado}")
# Salida:
# Array original: [3 1 2]
# Array ordenado: [1 2 3]
```

1.13.2. Ordenar un array a lo largo de un eje

Podemos usar la función `np.sort(array, axis=...)` para ordenar a lo largo de un eje específico. Esta función devuelve una **nueva copia** ordenada del array.

```
# Ordenamos el array tridimensional 'c' a lo largo del eje 0
c_ordenado_axis0 = np.sort(c, axis=0)

print("Array 'c' original:\n", c)
print("\nArray 'c' ordenado sobre el eje 0:\n", c_ordenado_axis0)
# Salida:
# Array 'c' original:
#  [[[1.5 2.  3. ]
#    [4.  5.  6. ]]
#
#   [[3.  2.  1. ]
```

```
#     [4.   5.   6. ]]]
#
# Array 'c' ordenado sobre el eje 0:
#  [[[1.5  2.   1. ]
#    [4.   5.   6. ]]
#
#   [[3.   2.   3. ]
#    [4.   5.   6. ]]]
```

Explicación: El `axis=0` en un array de forma (2, 2, 3) es el eje que "atraviesa" las dos matrices. Al ordenar sobre este eje, NumPy compara los elementos que están en la misma posición de fila y columna, pero en matrices diferentes, y los ordena. Por ejemplo:

- El elemento en la posición `[0,0,0]` (que es 1.5) se compara con el elemento en `[1,0,0]` (que es 3). El menor (1.5) va a la primera matriz del resultado y el mayor (3) a la segunda.
- El elemento en `[0,0,2]` (que es 3) se compara con `[1,0,2]` (que es 1). El menor (1) va a la primera matriz del resultado en esa posición y el mayor (3) a la segunda.
- Los elementos `[0,1,1]` (5) y `[1,1,1]` (5) son iguales, por lo que su orden no cambia.

1.14. Ejercicio 10: Indexado y slicing (rebanado) de arrays

El indexado y rebanado son operaciones fundamentales para acceder y manipular subconjuntos de datos dentro de un array de NumPy.

Listing 4: Definición de arrays para el ejercicio

```
import numpy as np

# Arrays de ejercicios anteriores
a = np.array([1, 2, 3])
b = np.array([[1.5, 2., 3.], [4., 5., 6.]])
c = np.array([[[1.5, 2., 3.], [4., 5., 6.]], [[3., 2., 1.], [4., 5.,
   6.]]])
```

1.14.1. Seleccionar un elemento por su índice

En Python y NumPy, el indexado comienza en 0. Por lo tanto, el segundo elemento se encuentra en el índice 1.

```
# Seleccionamos el segundo elemento de 'a' (indice 1)
segundo_elemento = a[1]

print(segundo_elemento)
# Salida: 2
```

1.14.2. Seleccionar un elemento en un array 2D

Se utiliza la sintaxis `[fila, columna]` para acceder a un elemento en un array de 2 dimensiones.

```
# Seleccionamos el elemento en la fila 1, columna 2 de 'b'
elemento_b = b[1, 2]

print(elemento_b)
# Salida: 6.0
```

1.14.3. Seleccionar un rango de elementos (Slicing)

La sintaxis `[n:m]` selecciona los elementos desde el índice `n` hasta el índice `m-1`.

```
# Seleccionamos los elementos en los indices 0 y 1 de 'a'
rango_a = a[0:2]

print(rango_a)
# Salida: [1 2]
```

1.14.4. Combinar slicing e indexado

Podemos usar rebanadas para las filas y un índice entero para las columnas.

```
# Seleccionamos las filas 0 y 1, pero solo de la columna 1
slice_b = b[0:2, 1]

print(slice_b)
# Salida: [2. 5.]
```

1.14.5. Seleccionar todas las columnas

Un dos puntos `:` por sí solo en una dimensión significa "seleccionar todos los elementos de este eje".

```
# Seleccionamos la fila 0 y todas sus columnas
fila_completa = b[0:1, :]

print(fila_completa)
# Salida: [[1.5 2.  3. ]]
```

1.14.6. Uso de la elipsis (...)

La elipsis `...` es un atajo que se expande a tantos dos puntos (`:`) como sean necesarios para completar la selección en las dimensiones restantes. Es especialmente útil en arrays de muchas dimensiones.

```
# Seleccionamos el segundo 'bloque' y todo lo que contiene
seleccion_elipsis = c[1, ...]
seleccion_normal = c[1, :, :]

print("Con elipsis:\n", seleccion_elipsis)
print("\nSon iguales?:", np.array_equal(seleccion_elipsis,
   seleccion_normal))
# Salida:
```

```
# Con elipsis:
#  [[3. 2. 1.]
#   [4. 5. 6.]]
#
# Son iguales?: True
```

Explicación: Sí, para un array de 3 dimensiones, `[1,...]` es exactamente lo mismo que `[1,:,:]`.

1.14.7. Invertir un array con slicing

Usando un tercer parámetro en la sintaxis de rebanado `[inicio:fin:paso]`, podemos definir el "salto.entre elementos. Un paso de `-1` recorre el array en orden inverso.

```
# Invertimos el array 'a'
a_invertido = a[::-1]

print(a_invertido)
# Salida: [3 2 1]
```

1.14.8. Indexado booleano

Podemos pasar un array de booleanos para seleccionar únicamente los elementos donde el valor sea `True`.

```
# 1. Creamos una condicion booleana
condicion = (a < 2)
print(f"La condicion (a < 2) es: {condicion}")

# 2. Usamos la condicion para indexar el array
resultado = a[condicion]
print(f"Elementos que cumplen la condicion: {resultado}")
# Salida:
# La condicion (a < 2) es: [ True False False]
# Elementos que cumplen la condicion: [1]
```

1.14.9. Indexado avanzado (Fancy Indexing)

Podemos pasar listas de índices para seleccionar elementos específicos en cualquier orden. Debemos proporcionar una lista para cada dimensión.

```
# Indices de fila y columna deseados
indices_fila = [1, 0, 1, 0]
indices_col = [0, 1, 2, 0]

# Seleccionamos los elementos (1,0), (0,1), (1,2) y (0,0) de 'b'
elementos_fancy = b[indices_fila, indices_col]

print(elementos_fancy)
# Salida: [4.  2.  6.  1.5]
```

Explicación: NumPy toma el primer índice de fila (1) y el primer índice de columna (0) para seleccionar `b[1,0]`. Luego toma el segundo par (0,1) para seleccionar `b[0,1]`, y así sucesivamente.

1.14.10. Indexado avanzado por pasos

Este ejercicio es una reformulación del anterior, ilustrando cómo el indexado avanzado selecciona un elemento por cada fila especificada.

```
# 1. Seleccionamos las filas en el orden deseado
filas_seleccionadas = b[[1, 0, 1, 0]]
# Esto crea una nueva matriz 4x3 con las filas repetidas
# [[4. 5. 6.]
#  [1.5 2. 3.]
#  [4. 5. 6.]
#  [1.5 2. 3.]]

# 2. De este resultado, seleccionamos los elementos usando indices de
   columna
# La logica es la misma que el punto anterior, combinando los indices
columnas_a_seleccionar = [0, 1, 2, 0]
resultado_final = b[[1, 0, 1, 0], columnas_a_seleccionar]

print(resultado_final)
# Salida: [4.  2.  6.  1.5]
```

1.15. Ejercicio 11: Transposición de Arrays

La transposición es una operación fundamental del álgebra lineal que intercambia las filas por las columnas de una matriz.

Listing 5: Definición de arrays para el ejercicio

```
import numpy as np

# Arrays de ejercicios anteriores
a = np.array([1, 2, 3])
b = np.array([[1.5, 2., 3.], [4., 5., 6.]]) # shape (2, 3)

# Array 'g' de Ej. 6 (b - a)
g = np.array([[0.5, 0., 0.], [3., 3., 3.]]) # shape (2, 3)

# Array 'h' de Ej. 8 (vista de 'a')
h = a.view() # h es [1, 2, 3]
```

1.15.1. Calcular la transpuesta con `np.transpose()`

La función `np.transpose()` toma un array y devuelve una nueva vista del array con sus ejes permutados. Para un array 2D, esto significa que las filas se convierten en columnas y viceversa.

```
# 'b' tiene forma (2, 3)
i = np.transpose(b)

print("Array original b (shape 2x3):\n", b)
print("\nArray transpuesto i (shape 3x2):\n", i)
# Salida:
```

```
# Array original b (shape 2x3):
#  [[1.5 2.  3. ]
#   [4.  5.  6. ]]
#
# Array transpuesto i (shape 3x2):
#  [[1.5 4. ]
#   [2.  5. ]
#   [3.  6. ]]
```

1.15.2. Acceder a la transpuesta con el atributo `.T`

El atributo `.T` es un atajo conveniente para calcular la transpuesta de un array.

```
# Transponemos 'i' (que ya era la transpuesta de 'b')
i_transpuesta = i.T

print("Transpuesta de i (.T):\n", i_transpuesta)
print("\nEs i.T igual a b?:", np.array_equal(i_transpuesta, b))
# Salida:
# Transpuesta de i (.T):
#  [[1.5 2.  3. ]
#   [4.  5.  6. ]]
#
# Es i.T igual a b?: True
```

Explicación: Sí, la transpuesta de la transpuesta de una matriz es la matriz original. Por lo tanto, `i.T` es igual a `b`.

1.16. Ejercicio 12: Redimensionado

1.16.1. Aplanar un array con `.ravel()`

El método `.ravel()` .ªplanaün array multidimensional, devolviendo una vista 1D contigua de todos sus elementos.

```
# Aplanamos el array 'b' de 2x3
b_achatado = b.ravel()

print(b_achatado)
# Salida: [1.5 2.  3.  4.  5.  6. ]
```

1.16.2. Redimensionar con `.reshape()`

El método `.reshape()` permite cambiar las dimensiones de un array sin cambiar sus datos, siempre que el número total de elementos sea el mismo.

```
# 'g' tiene 6 elementos (shape 2x3)
# Lo redimensionamos a 3 filas y un numero de columnas inferido
g_redimensionado = g.reshape((3, -1))

print(g_redimensionado)
# Salida:
```

```
# [[0.5 0. ]
#  [0.  3. ]
#  [3.  3. ]]
```

1.16.3. Significado del índice negativo (`-1`) en reshape

En `.reshape()`, un valor de `-1` en una de las dimensiones actúa como un comodín. Le indica a NumPy que **calcule automáticamente el tamaño de esa dimensión** basándose en el número total de elementos del array y el tamaño de las otras dimensiones especificadas. En el ejemplo anterior, como el array `g` tiene 6 elementos y especificamos 3 filas, NumPy calcula que la segunda dimensión debe ser $6/3 = 2$.

1.17. Ejercicio 13: Agregando y Quitando Elementos

1.17.1. Redimensionar y rellenar con `np.resize()`

A diferencia de `.reshape()`, `np.resize()` crea un **nuevo array** con la forma especificada. Si la nueva forma es más grande que la original, los elementos del array original se **repiten** para rellenar el espacio.

```
# 'h' tiene 3 elementos: [1, 2, 3]
# Creamos un nuevo array de 2x6 (12 elementos)
h_redimensionado = np.resize(h, (2, 6))

print(h_redimensionado)
# Salida:
# [[1 2 3 1 2 3]
#  [1 2 3 1 2 3]]
```

1.17.2. Añadir elementos con `np.append()`

La función `np.append()` anexa valores a una copia de un array. Si no se especifica un eje (`axis`), los arrays se aplanan antes de unirse.

```
# 'h' es 1D (3 elem.), 'g' es 2D (6 elem.)
# NumPy los aplanara a ambos y luego los unira
h_y_g = np.append(h, g)

print(h_y_g)
# Salida: [1.  2.  3.  0.5 0.  0.  3.  3.  3. ]
```

Explicación: NumPy resuelve la diferencia de dimensiones aplanando primero `h` (que ya es plano) y `g` a versiones 1D, y luego concatenando el segundo al final del primero.

1.17.3. Insertar elementos con `np.insert()`

Crea un nuevo array con valores insertados en una posición (eje) específica.

```
# Insertamos el numero 5 en la posicion de indice 1 del array 'a'
a_insertado = np.insert(a, 1, 5)
```

```
print(f"Array original: {a}")
print(f"Array con insercion: {a_insertado}")
# Salida:
# Array original: [1 2 3]
# Array con insercion: [1 5 2 3]
```

1.17.4. Eliminar elementos con `np.delete()`

Crea un nuevo array eliminando elementos de una posición (eje) específica.

```
# Eliminamos el elemento en la posicion de indice 1 de 'a'
a_eliminado = np.delete(a, 1)

print(f"Array original: {a}")
print(f"Array con eliminacion: {a_eliminado}")
# Salida:
# Array original: [1 2 3]
# Array con eliminacion: [1 3]
```

1.18. Ejercicio 14

1.18.1. Concatenar con `np.concatenate()`

Esta función une una secuencia de arrays a lo largo de un eje existente. Para arrays 1D, los une uno a continuación del otro.

Listing 6: Definición de arrays para el ejercicio

```
import numpy as np

# Arrays de ejercicios anteriores
a = np.array([1, 2, 3])
d = np.arange(10, 25, 5) # Array [10, 15, 20]
b = np.array([[1.5, 2., 3.], [4., 5., 6.]])
e = np.full((2, 2), 7)
f = np.eye(2)
c = np.array([[[1.5, 2., 3.], [4., 5., 6.]], [[3., 2., 1.], [4., 5.,
   6.]]])
```

```
# Concatenamos los arrays 1D 'a' y 'd'
ad_concat = np.concatenate((a, d))

print(ad_concat)
# Salida: [ 1  2  3 10 15 20]
```

1.18.2. Apilar verticalmente con `np.vstack()`

Apila arrays en secuencia vertical (por filas). Trata a los arrays 1D como si fueran filas.

```
# Apilamos 'a' (1D) encima de 'b' (2D)
# NumPy convierte 'a' a un array 2D de forma (1,3) para que coincida
ab_vstack = np.vstack((a, b))
```

```
print(ab_vstack)
# Salida:
# [[1.  2.  3. ]
#  [1.5 2.  3. ]
#  [4.  5.  6. ]]
```

1.18.3. Apilar por filas

```
np.row_stack() es un alias para np.vstack(). El comportamiento es idé
   ntico.

# Apilamos 'e' y 'f', ambos de forma (2,2)
ef_rowstack = np.row_stack((e, f))

print(ef_rowstack)
# Salida:
# [[7. 0.]  <- fila de e
#  [7. 0.]  <- fila de e
#  [1. 0.]  <- fila de f
#  [0. 1.]] <- fila de f
```

1.18.4. Apilar horizontalmente con `np.hstack()`

Apila arrays en secuencia horizontal (por columnas).

```
# Apilamos 'e' y 'f' uno al lado del otro
ef_hstack = np.hstack((e, f))

print(ef_hstack)
# Salida:
# [[7. 7. 1. 0.]
#  [7. 7. 0. 1.]]
```

1.18.5. Apilar por columnas

Para arrays 1D, esta función los convierte primero en columnas antes de apilarlos horizontalmente.

```
# 'a' y 'd' son 1D. Se convertiran en columnas de (3,1) y luego se
   uniran
ad_colstack = np.column_stack((a, d))

print(ad_colstack)
# Salida:
# [[ 1 10]
#  [ 2 15]
#  [ 3 20]]
```

1.18.6. Separar horizontalmente con `np.hsplit()`

Divide un array en múltiples sub-arrays horizontalmente (a lo largo del eje 1). Para arrays 1D, divide a lo largo del eje 0.

```
# Separamos el array 'a' de 3 elementos en 3 partes iguales
a_split = np.hsplit(a, 3)

print(a_split)
# Salida: [array([1]), array([2]), array([3])]
```

Explicación: El resultado es una lista de Python que contiene los nuevos arrays.

1.18.7. Separar verticalmente con `np.vsplit()`

Divide un array en múltiples sub-arrays verticalmente (a lo largo del eje 0).

```
# 'c' tiene forma (2, 2, 3). Lo separamos en 2 por el primer eje (el
   de tamaño 2)
c_split = np.vsplit(c, 2)

# Imprimimos la forma de cada parte para ver el resultado
print(f"Shape de la primera parte: {c_split[0].shape}")
print(f"Shape de la segunda parte: {c_split[1].shape}")
# Salida:
# Shape de la primera parte: (1, 2, 3)
# Shape de la segunda parte: (1, 2, 3)
```

Explicación: La función dividió el array `c` en dos "bloques" a lo largo de su primera dimensión, resultando en una lista de dos arrays.

1.18.8. Separar a lo largo de un eje específico

La función genérica es `np.split(array, secciones, axis=...)`. Las funciones `hsplit` y `vsplit` son atajos para `axis=1` y `axis=0` respectivamente.

```
# Creamos el array z
z = np.array([[1, 2, 3, 4], [2, 0, 0, 2], [3, 1, 1, 0]])

# Separamos z en 2 partes a lo largo del eje 1 (columnas)
z_split = np.split(z, 2, axis=1)

print("Primera parte:\n", z_split[0])
print("\nSegunda parte:\n", z_split[1])
# Salida:
# Primera parte:
#  [[1 2]
#   [2 0]
#   [3 1]]
#
# Segunda parte:
#  [[3 4]
#   [0 2]
#   [1 0]]
```