Numpy

Introducción

Numpy:

Es una librería enfocada al cálculo númerico y manejo de Arrays.

- Es muy veloz, hasta 50 veces más rápido que usar una lista de Python o C.
- Optimiza el almacenamiento en memoria.
- Maneja distintos tipos de datos. Es una librería muy poderosa, se pueden crear redes neuronales desde cero.

Para importar la librería se debe realizar el siguiente codigo:

```
import numpy as np
```

Numpy

Array: El array es el principal objeto de la librería. Representa datos de manera estructurada y se puede acceder a ellos a traves del indexado, a un dato específico o un grupo de muchos datos específicos.

Ejemplo:

Importaremos la libreria y crearemos dos arrays uno en una dimension y otra de 3:

```
import numpy as np

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

# output ---> array([1, 2, 3, 4, 5, 6, 7, 8, 9])

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

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

Indexado:

El indexado nos permite acceder a los elementos de los array y matrices Los elementos se emepiezan a contar desde 0.

```
lista[0]
# output ---> 1

lista[0] + lista[5]
# output ---> 7
```

Slicing:

El slicing nos permite extraer varios datos, tiene un comienzo y un final. En este ejemplo se está extrayendo datos desde la posición 1 hasta la 5. [1:6].

```
lista[0:3]
# output ---> array([1, 2, 3])

matriz[1:,0:2]
# output ---> array([[4, 5],
#                    [7, 8]])
```

Tipos de datos:

Los arrays de NumPy solo pueden contener un tipo de dato, ya que esto es lo que le confiere las ventajas de la optimización de memoria.

Podemos conocer el tipo de datos del array consultando la propiedad .dtype.

```
arr = np.array([1,2,3,4])
arr.dtype
# output ---> dtype('int64')
```

como las redes neuronales trabajan mejor con arreglos tipo flotante (numeros decimales) entonces cambiaremos el formato del arreglo anterior con el siguiente codigo:

```
arr = np.array([1,2,3,4], dtype='float64')
arr.dtype
# output ---> dtype('float64')
```

otra manera de convertirlo es de la siguiente manera (directamente desde la libreria):

```
arr = arr.astype(np.float64)
arr.dtype
# output ---> dtype('float64')
```

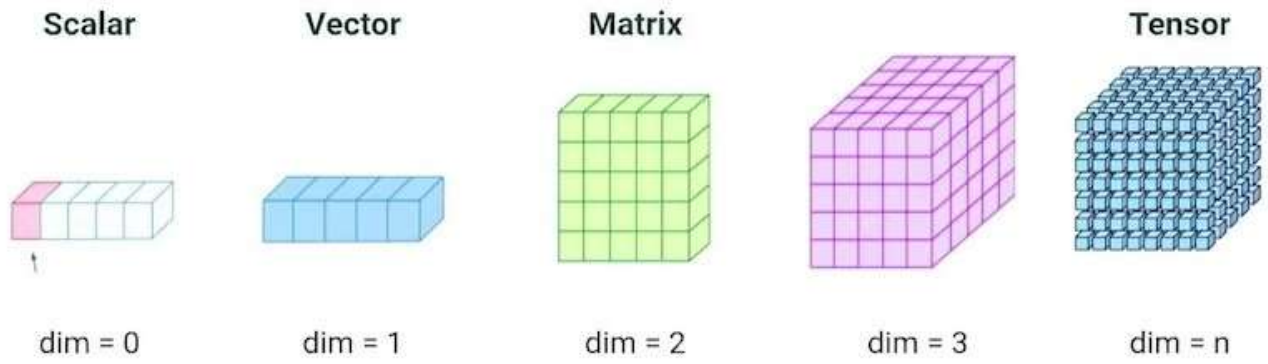
También se puede cambiar a tipo booleano recordando que los números diferentes de 0 se convierten en True.

```
arr = np.array([0, 1, 2, 3, 4])
arr = arr.astype(np.bool_)
arr
# output ---> array([False,  True,  True,  True,  True])
```

el resto de formatos los podemos consultar dando click AQUÍ.

Dimensiones:

- scalar: dim = 0 Un solo dato o valor
- vector: dim = 1 Listas de Python
- matriz: dim = 2 Hoja de cálculo
- tensor: dim > 3 Series de tiempo o Imágenes



Declarando un escalar:

```
scalar = np.array(42)
print(scalar)
scalar.ndim
# output ---> 42
#        ---> 0
```

Declarando un Vector:

```
vector = np.array([1, 2, 3])
print(vector)
vector.ndim
# output ---> [1 2 3]
#        ---> 1
```

Declarando una matriz:

```
matriz = np.array([[1, 2, 3], [4, 5, 6]])
print(matriz)
matriz.ndim
# output ---> [[1 2 3]
#              [4 5 6]]
#        ---> 2
```

Declarando un Tensor:

```
tensor = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]],[[13, 13, 15],
[16, 17, 18], [19, 20, 21], [22, 23, 24]]])
print(tensor)
tensor.ndim
# output ---> [[[ 1  2  3]
#               [ 4  5  6]
#               [ 7  8  9]
#               [10 11 12]]

#              [[13 13 15]
#               [16 17 18]
#               [19 20 21]
#               [22 23 24]]]
#       ---> 3
```

Agregar o elimiar dimensiones:

Se puede definir el número de dimensiones desde la declaración del array

```
vector = np.array([1, 2, 3], ndmin = 10)
print(vector)
vector.ndim
# output ---> [[[[[[[[[[1 2 3]]]]]]]]]]
#        ---> 10
```

Se pueden expandir dimensiones a los array ya existentes. Axis = 0 hace refencia a las filas, mientras que axis = 1 a las columnas.

```
expand = np.expand_dims(np.array([1, 2, 3]), axis = 0)
print(expand)
expand.ndim
# output ---> [[1 2 3]]
#        ---> 2
```

Para remover o comprimirdimensiones que no estan siendo usadas

```
print(vector, vector.ndim)
vector_2 = np.squeeze(vector)
print(vector_2, vector_2.ndim)
# output ---> [[[[[[[[[[1 2 3]]]]]]]]]] 10
#        ---> [1 2 3] 1
```

Creando Arrays:

Este métododе NumPy nos permite generar arrays sin definir previamente una lista.

```
np.arange(0,10)
# output ---> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Un tercer argumento permite definir un tamaño de paso.

```
np.arange(0,20,2)
# output ---> array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

np.zeros() nos permite definir estructuras o esquemas.

```
np.zeros(3)
# output ---> array([0., 0., 0.])
np.zeros((10,5))
# output ---> array([[0., 0., 0., 0., 0.],
#                    [0., 0., 0., 0., 0.],
#                    [0., 0., 0., 0., 0.],
#                    [0., 0., 0., 0., 0.],
#                    [0., 0., 0., 0., 0.],
#                    [0., 0., 0., 0., 0.],
#                    [0., 0., 0., 0., 0.],
#                    [0., 0., 0., 0., 0.],
#                    [0., 0., 0., 0., 0.],
#                    [0., 0., 0., 0., 0.]])
```

De iugal forma tenemos np.ones()

```
np.ones(3)
# output ---> array([1., 1., 1.])
```

np.linspace() permite generar una arrary definiendo un incio, un final y cuantas divisiones tendrá.

```
np.linspace(0, 10 , 10)
# output ---> array([ 0.,1.11111111,2.22222222,  3.33333333,  4.44444444,
#                     5.55555556,  6.66666667,  7.77777778,  8.88888889, 10])
```

También podemos crear una matriz con una diagonal de 1 y el resto de 9.

```
np.eye(4)
# output ---> array([[1., 0., 0., 0.],
#                    [0., 1., 0., 0.],
#                    [0., 0., 1., 0.],
#                    [0., 0., 0., 1.]])
```

Otra metódo importante es generar números aleatorios.

```
np.random.rand()
# output --->0.37185218178880153
```

También se pueden generar vectores.

```
np.random.rand(4)
# output ---> array([0.77923054, 0.90495575, 0.12949965, 0.55974303])
```

Y a su vez generar matrices.

```
np.random.rand(4,4)
# output ---> array([[0.26920153, 0.24873544, 0.02278515, 0.08250538],
            #        [0.16755087, 0.59570639, 0.83604996, 0.57717126],
            #        [0.00161574, 0.27857138, 0.33982786, 0.19693596],
            #        [0.69474123, 0.01208492, 0.38613157, 0.609117  ]])
```

NumPy nos permite tambien generar números enteros. En este caso números enteros entre el 1 y 14

```
np.random.randint(1,15)
# output ---> 4
```

También podemos llevarlos a una estructura definida.

```
np.random.randint(1,15, (3,3))
# output ---> array([[ 8,  2,  6],
            #        [ 7,  1,  8],
            #        [11, 14,  4]])
```

Shape y Reshape

shape me indica la forma que tiene un arreglo, es decir, me indica con que estructura de datos estoy trabajando. Reshape transforma el arreglo mientras se mantengan los elementos.

```
arr = np.random.randint(1,10,(3,2))
arr.shape
# output ---> (3, 2)
```

Reshape

con el siguiente arreglo

```
array([[4, 9],
       [9, 2],
       [3, 4]])

# Aplicamos el Reshape

arr.reshape(1,6)

# output ---> array([[4, 9, 9, 2, 3, 4]])
```

otra forma de lograr el reshape

```
np.reshape(arr,(1,6))
# output ---> array([[4, 9, 9, 2, 3, 4]])
```

Se puede hacer un reshape como lo haría el lenguaje C.

```
np.reshape(arr,(2,3), 'C')
# output ---> array([[5, 6, 4],
#                    [6, 2, 3]])
```

También se puede hacer reshape a como lo haría Fortran.

```
np.reshape(arr,(2,3), 'F')
# output ---> array([[5, 2, 6],
#                    [4, 6, 3]])
```

Además existe la opción de hacer reshape según como esté optimizado nuestro computador. En este caso es como en C.

```
np.reshape(arr,(2,3), 'A')
# output ---> array([[5, 6, 4],
#                    [6, 2, 3]])
```

Funciones principales de NumPy

Trabajaremos con el siguiente vector y la siguiente matriz:

```
arr ---> array([ 4, 19, 16, 12,  5,  6, 19,  6,  1,  8])

matriz ---> array([[ 4, 19, 16, 12,  5],
                   [ 6, 19,  6,  1,  8]])
```

Función max()

```
arr.max()
# output ---> 19
```

si la quiero mostrar en la matriz por eje

```
matriz.max(0)
# output ---> array([ 6, 19, 16, 12,  8])
```

para mostrar en que indice se encuentra el valor maximo (me muestra el indice menor)

```
arr.argmax()
# output ---> 1
```

Función min()

nos entrega el menos valor, contiene las mismas funciones de max pero nos devuelve el menor valor

Función ptp()

me trae la diferencia numerica entre el valor maximo y el valor minimo

```
arr.ptp()
# output ---> 18
```

Función percentile()

Me especifica directamente el percentil que le indico

```
np.percentile(arr, 50)
# output ---> 7.0
```

Función sort()

Me ordena de mayor a menor la información del arreglo o matriz

```
arr.sort()
arr
# output ---> array([ 1,  4,  5,  6,  6,  8, 12, 16, 19, 19])
```

Función median()

Me entrega la mediana de un arreglo

```
np.median(arr)
# output ---> 7.0
```

Función std()

Me calcula la desviación estandar del arreglo

```
np.std(arr)
# output ---> 6.151422599691879
```

Función var()

Me calcula la varianza del arreglo

```
np.var(arr)
# output ---> 37.84

# desviación estandar ** 2 me da como resultado la varianza

np.std(arr) ** 2
# output ---> 37.84
```

Función mean()

Me calcula la media del arreglo

```
np.mean(arr)
# output ---> 9.6
```

Función concatenate()

me une o concatena la informacion de dos arreglos

trabajaremos con los siguientes arreglos

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

para unir el array **a** con el array **b**

```
np.concatenate((a,b))
# output ---> array([[1, 2],
#                    [3, 4],
#                    [5, 6]])
```

Si queremos concatenarlo de manera que nos quede en una matriz 2x3 debemos trasponer la información del array **b** ya que solo tiene una dimension; quedaria de la siguiente manera:

```
np.concatenate((a,b.T), axis=1)
# output --->array([[1, 2, 5],
#                   [3, 4, 6]])
```

- Existen más funciones estadisticas, para poder consultarlas dar click AQUÍ
- Y más funciones matemáticas dando click AQUÍ

Copy

.copy() nos permite copiar un array de NumPy en otra variable de tal forma que al modificar el nuevo array los cambios no se vean reflejados en array original.

```
arr = np.arange(0, 11)
# output ----> array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

arr[0:6] ----> array([0, 1, 2, 3, 4, 5])
trozo_de_arr = arr[0:6]
trozo_de_arr[:] = 0
# output trozo_de_arr ----> array([0, 0, 0, 0, 0, 0])
```

Se han modificado los datos del array original porque seguía haciendo referencia a esa variable.

```
arr
# output ----> array([ 0,  0,  0,  0,  0,  0,  6,  7,  8,  9, 10])

arr_copy = arr.copy()
arr_copy[:] = 100

arr_copy
# output ----> array([100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100])
arr
# output ----> array([ 0,  0,  0,  0,  0,  0,  6,  7,  8,  9, 10])
```

Condiciones

son un conjunto de parametros que colocamos al hacer un slicing o un idexing dentro de un array con alguna condición, ejemplo quiero extraer del array los numeros pares. nos permite realizar consultas al array mas especificas que sean dificil lograr con un simple slicing.

```
arr = np.linspace(1,10,10, dtype='int8')
# output ----> array([ 1,  2,  3,  4,  5,  6,  7,  8, 9, 10], dtype=int8)
```

si yo creo una vartiable con una condicion, ejemplo los numeros > a 5 del array

```
indices_cond = arr > 5
# output ----> array([False, False, False, False, False,  True,  True,  True,
True, True])
```

Si yo realizo slicing al array inicial "arr" poniendo como index el "indices_cond" tenemos como resultado la condición planteada dentro de la variable "indices_cond":

```
arr[indices_cond]
# output ----> array([ 6,  7,  8,  9, 10], dtype=int8)
```

otro ejemplo con dos condiciones:

```
arr[(arr > 5) & (arr < 9)]
# output ----> array([6, 7, 8], dtype=int8)
```

De igual forma modificar los valores que cumplan la condición.

```
arr[arr > 5] = 99
# output ----> array([ 1,  2,  3,  4,  5, 99, 99, 99, 99, 99], dtype=int8)
```

Operaciones

Cuando realizamos una multiplicacionen donde queremos aplicar el factor a cada elemento en una lista en python, el sistema entiende que lo que queremos es duplicar los datos, ejemplo:

```
lista = [1,2]
lista * 2
# output ---> [1, 2, 1, 2]
```

realicemos las operaciones desde numpy:

```
arr = np.arange(0,10)
arr2 = arr.copy() # realizamos copia para no modificar el array principal

# output ---> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

arr * 2
# output ---> array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Con numpy tambien podemos hacer operaciones entre arrays

```
arr + arr2
# output ---> array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Se pueden hacer operaciones de producto punto entre vectores con las dos siguientes maneras:

```
np.matmul(matriz, matriz2.T)
# output ---> array([[ 30,  80],
#                    [ 80, 255]])
matriz @ matriz2.T
# output ---> array([[ 30,  80],
#                    [ 80, 255]])
```