NUMPY 실습

고민성

왜 Colab, Jupyter를 사용할까?

INTERPRETER 에서도 PYTHON 이 사용 가능하다.

But, 간단한 작업은 쉽게 할 수 있어도, (기본 연산 3+4 등..)
복잡한 작업 시 훨씬 어려움을 겪게 된다.

Numpy

Numpy는 numeric python의 약자로 여러 계산을 쉽게
파이썬에서 구현할 수 있도록 도와주는 필수적인 라이브러리!

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

Numpy를 통해서scalar, vector, matrix 등 표현할 수 있다.

a 스칼라 -> 0차원

b 벡터 -> 1차원

c 행렬 -> 2차원

다차원을 표현할 수 있다.

Numpy

다차원

```
d = np.array([[[1., 2., 3.], [4., 5., 6.]], [[7., 8., 9.], [10., 11., 12.]]])
e = np.array([[[[1., 2., 3.], [1., 2., 3.]], [[4., 5., 6.], [4., 5., 6.]]],
              [[[7., 8., 9.], [7., 8., 9.]], [[10., 11., 12.], [10., 11., 12.]]]])
```

d는 3차원, e는 4차원이고 n 차원까지 표현이 가능하다.

다양한 방식으로 차원을 만들 수 있다.

```
np.random.random((3,3,3))
```

```
array([[[0.1404335 , 0.06057981, 0.85965312],
        [0.78576   , 0.93080583, 0.3212696 ],
        [0.07016747, 0.36161946, 0.59804039]],

       [[0.24798136, 0.01915649, 0.11334369],
        [0.0092824 , 0.67376735, 0.71938968],
        [0.94864568, 0.28541913, 0.44283651]],

       [[0.96285286, 0.8902585 , 0.78113118],
        [0.04837432, 0.25845821, 0.1186602 ],
        [0.64385187, 0.48306001, 0.11118305]]])
```

```
np.ones((2,2,2))
```

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

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

```
np.zeros((2, 5))
```

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

```
np.full((2,5), 3)
```

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

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

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

Numpy

Indexing & slicing

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

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

a[0] = 0, 1, 2, 3, 4 a[-1, -2] =8 a[-1, 0:2] = 5,6 a[-2,-2:] = 3, 4

```
a[-2, 0:5:2] = a[-2][0:5:2] = 0, 2, 4
```

인덱싱과 슬라이싱을 통해 원하는 부분만을 가져올 수 있다.

```
a=np.arange(10).reshape((2,5))
idx = a%2== 1
b=a[idx]; b
```

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

Index boolean을 통해서 조건을 통해서

원하는 부문을 가져올 수도 있다.

Numpy

Math operation

```
# Basic operations
a = np.arange(6).reshape((3, 2))
b = np.ones((3, 2))
print_obj(a, "a")
print_obj(b, "b")

# +, -, *, /
print_obj(a+b, "a+b")
print_obj(a-b, "a-b")
print_obj(a*b, "a*b")
print_obj(a/b, "a/b")
```

```
a:
[[0 1]
 [2 3]
 [4 5]]

b:
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

```
a+b:
[[1. 2.]
 [3. 4.]
 [5. 6.]]

a*b:
[[0. 1.]
 [2. 3.]
 [4. 5.]]

a-b:
[[-1.  0.]
 [ 1.  2.]
 [ 3.  4.]]

a/b:
[[0. 1.]
 [2. 3.]
 [4. 5.]]
```

배열끼리의 연산과 배열 내부에서의 연산 가능.

```
a.sum()
15
```

```
a.sum(axis=1)
array([1, 5, 9])
```

```
a.sum(axis=0)
array([6, 9])
```

axis 에 따라서 연산 방식을 조정할 수 있다. 2차원에서 axis=1 일 경우 행 끼리 연산, 0일 경우 열 끼리 연산 없을 경우 전체 연산

이는 sum 뿐만 아니라, mean, max, min 등 다 적용 가능.

차원이 늘어날 경우 큰 차원부터 axis=0 3차원일 경우 차원이 가장 작은 axis=2 or axis=-1 axis -연산은 리스트와 똑같이 적용!

3차원의 경우 차원 끼리 연산 axis=0, 열끼리 연산 axis=1, 행끼리 연산 axis=2

Numpy

Dot

1차원

```
# Vector dot product
a = np.arange(3).astype('float')
b = np.ones(3)
print_obj(a, "a")
print_obj(b, "b")

print_obj(np.dot(a, b), "a dot b")
```

```
a:
[0. 1. 2.]

b:
[1. 1. 1.]

a dot b:
3.0
```

```
np.dot(2,3)
```

```
6
```

두 입력이 스칼라일 경우 스칼라 반환

두 입력이 벡터일 경우 벡터의 내적을 반환.

2차원

```
a = np.arange(4).reshape(2,2).astype('float')
b = np.ones(4).reshape(2,2)
print_obj(a, "a")
print_obj(b, "b")

print_obj(np.dot(a, b), "a dot b")
```

```
a:
[[0. 1.]
 [2. 3.]]

b:
[[1. 1.]
 [1. 1.]]

a dot b:
[[1. 1.]
 [5. 5.]]
```

행렬의 경우 행렬의 곱 연산
(첫 번째 행렬의 마지막 축과 두 번째 행렬의 마지막에서 두번째 축이 같아야 한다.)

한 쪽이 스칼라일 경우는 단순 곱셈 연산.

행렬의 곱에서는 dot 보다는 @ matual 을 권장함.

Numpy

Matual (@ operater)

행렬의 곱연산을 위해 정의된 함수이다.

Dot과는 유사하나, 3차원 이상의 행렬곱부터 계산 방식이 달라진다.

그리고 dot은 행렬과 스칼라의 곱셈에서 오류를 일으킴.

Mutual 은 3차원 이상의 행렬의 경우 예를 들어 2*3*5인 경우, 3*5의 행렬을 2개 갖고 있는 것으로 본다.

따라서 마지막 두 개의 축이 서로 행렬의 곱이 가능한 경우만 계산이 가능하다.

그리고 배열의 개수 또한 동일하게 가져야 계산이 가능한데, 예를 들어 a= 2*3*5 일 경우,

B1 = 3*5*3 은 오류가 발생하지만, B2= 2*5*4는 계산이 가능함.

Numpy

Dot & mutual

정리하자면 dot은 첫 번째 행렬의 마지막 축과, 두 번째 형렬의 마지막에서 두 번째 축으로 연산을 함

3차원 , 3차원의 경우 -> 4차원

Mutual은 양 쪽 행렬 다 끝에서 두개의 축이 행렬 연산이 가능하다면 두 개의 축을 연산 한 뒤, 앞에 나머지 축의 곱은 개수로 판단

3차원, 3차원의 경우 -> 3차원

Mutual의 연산 값은 3차원으로 고정

Numpy

Dot & mutual

Function	Condition (3-D)	Formula (3-D)

np.dot	A.shape # (a1, a2, a3)
	B.shape # (b1, b2, b3)
	--> a3==b2
	
	>>> C = np.dot(A,B)
	>>> C.shape
	(a1, a2, b1, b3)
		C[i,j,k,m] =
		np.sum(A[i,j,:] * B[k,:,m])

np.matmul	A.shape # (a1, a2, a3)
	B.shape # (b1, b2, b3)
	--> (a1==b1) and (a3==b2)
	
	>>> C = np.matmul(A,B)
	>>> C.shape
	(a1, a2, b3)
		C[i,j,k] =
		np.sum(A[i,j,:] * B[i,:,k])

https://blog.naver.com/cjh226/221356884894

Numpy

Shape manipulation

```
a = np.arange(24).reshape((2, 3, 4))
```

```
c = a.reshape((6, -1))    # -1은 숫자 개수에 맞춰서 알맞은 수 대입
print_obj(c, "c")
```

```
# Quiz: What would d=a.reshape((6, 4, -1)) look like?   -> (6,4)
# Quiz: What would d=a.reshape((6, 4, 1, -1)) look like?   -> (6,4)
```

Reshape 을 통해서 모양을 변환시킬 수 있다.

Reshape에 -1을 넣어주면 원소의 개수에 맞춰서 적합한 수 자동으로 대입.

차원 모양으로 1을 대입해도 차원의 영향 x
-> 1 과 [1] 은 같은 차원.

Numpy

Stack, concat

```
a:
[[1. 1.]
 [1. 1.]
 [1. 1.]]

b:
[[0. 0.]
 [0. 0.]
 [0. 0.]]
```

```
print_obj(np.vstack([a, b]), "a,b vstack")
print_obj(np.hstack([a, b]), "a,b hstack")
```

```
a,b vstack:
[[1. 1.]
 [1. 1.]
 [1. 1.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]

a,b hstack:
[[1. 1. 0. 0.]
 [1. 1. 0. 0.]
 [1. 1. 0. 0.]]
```

V 스택은 행을 쌓아주는 함수

H 스택은 열을 쌓아주는 함수

Stack 은 쌓아주는 역할이기에 두 개의 행렬의 차원이 완전히 같아야 한다.

```
print_obj(np.concatenate([a, b], axis=0), "a,b concat axis=0")
print_obj(np.concatenate([a, b], axis=1), "a,b concat axis=1")
```

```
a,b concat axis=0:
[[1. 1.]
 [1. 1.]
 [1. 1.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]

a,b concat axis=1:
[[1. 1. 0. 0.]
 [1. 1. 0. 0.]
 [1. 1. 0. 0.]]
```

Axis 의 값에 따라 붙여주는 방향이 다름

Stack 은 붙여주는 역할이기에 두 개의 행렬의 한 개의 기준만 같으면 가능하다.

Numpy

Shape manipulation

```
# Matrix transpose
a = np.arange(6).reshape((3, 2))
print_obj(a, "a")

print_obj(a.T, "a.T")
```

```
a:
[[0 1]
 [2 3]
 [4 5]]

a.T:
[[0 2 4]
 [1 3 5]]
```

a.T를 이용하여 차원의 스와핑

차원의 스와핑을 통해 연산이 가능해짐.

Broadcasting

```
# Vector and scalar
a = np.arange(3)   #벡터와 스칼라는 연산이 안되지만 numpy가 그걸 가능케 함

b = 2.    # 스칼라 b를 브로드캐스팅하여 b-> [2,2,2]로 변환해줌
print_obj(a, "a")

print_obj(a+b, "a+b")
print_obj(a-b, "a-b")
```

```
a:
[0 1 2]

a+b:
[2. 3. 4.]

a-b:
[-2. -1.  0.]

a*b:
[0. 2. 4.]

a/b:
[0.  0.5 1. ]
```

원래 차원이 다르면 연산이 불가능하나,

Numpy에서는 자동적으로 broadcasting을 통해서

가능케 해준다.

B = 2 -> [2,2,2]로 변환해줌

NUMPY 실습 끝 !