



텐서의 전치와 차원변경

최석재 lingua@naver.com

전치 *transpose*

- 딥러닝에서는 사용하는 Layer의 종류에 따라 요구하는 차원이 다르다
- 입력층 ~ 은닉층 ~ 출력층에 이르는 각 층을 통과하게 하기 위하여
- 텐서를 전치하는 경우가 자주 발생한다
- 텐서의 전치는 텐서플로 Tensor의 메소드로도 가능하나,
- Numpy 배열이 데이터 사이언스에서 더 친숙하게 여겨져 더 많이 사용된다
- 여기서는 Numpy 배열로 데이터(텐서)를 전치하는 방법을 알아본다
- Numpy 패키지 임포트는 아래와 같이 한다
- `import numpy as np`

reshape

- 데이터의 값은 유지하면서, 형상을 바꿔야 할 때가 있다
- `np.reshape()`는 형상을 변경할 때 자주 사용된다

- `n1 = np.array(range(24))`
- `print(n1)`

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

1차원 배열을 3행 8열로 reshape

- `n2 = n1.reshape(3, 8)`
- `print(n2)`

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]]
```

reshape 자동계산

- reshape()를 사용할 때는 형상이 맞아야 한다
- 아래는 형상이 맞지 않아 에러가 발생한다
- `n1.reshape(3, 9)` *# Error*
- 이러한 경우, numpy로 하여금 자동으로 계산하게 할 수 있다
- 전체 축에서 한 부분은 -1로 설정하면 자동 계산된다
- `n1.reshape(3, -1)`

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7],  
       [ 8,  9, 10, 11, 12, 13, 14, 15],  
       [16, 17, 18, 19, 20, 21, 22, 23]])
```

reshape 자동계산

- 이러한 방식은 보다 복잡한 방식에서 자주 사용된다
- 아래는 3차원 배열로 만드는 것이며,
- 1D는 2, 2D는 자동 계산(3), 3D는 4로 만든다
- `n3 = n1.reshape(2, -1, 4)`
- `print(n3)`
- `print(n3.shape)`

```
[[[ 0  1  2  3]
   [ 4  5  6  7]
   [ 8  9 10 11]]]
```

```
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
(2, 3, 4)
```

(2, 3, 4) shape가 어떻게 되는지 확인

reshape 로 벡터 만들기

- `reshape(-1)` 은 2D 이상의 데이터를 1D 벡터로 만든다
- `print(n3.reshape(-1))`

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

전치 행렬

- `x1 = np.arange(28).reshape(7, 4)` # 0~27의 숫자를 생성하고, (7행, 4열)의 행렬로 변환
- `x1_T = x1.T` # 만들어진 행렬을 전치하여 행과 열을 맞바꾼다
- `print(x1, '\n')`
- `print(x1_T, '\n')`

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]
 [24 25 26 27]]
```

```
[[ 0  4  8 12 16 20 24]
 [ 1  5  9 13 17 21 25]
 [ 2  6 10 14 18 22 26]
 [ 3  7 11 15 19 23 27]]
```

transpose

- 전치를 3차원 이상의 행렬에 대하여 확장한 것이 np.transpose()

- print(n3)

```
[[[ 0  1  2  3]
   [ 4  5  6  7]
   [ 8  9 10 11]]]
```

- print(n3.shape)

```
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]]
(2, 3, 4)
```

축을 (1, 2, 0) 순서로 교체

즉, 2→3, 3→4, 4→2 로 바뀐다

- n4 = np.transpose(n3, (1, 2, 0))

```
[[[ 0 12]
   [ 1 13]
   [ 2 14]
   [ 3 15]]]
```

- print(n4)

```
[[ 4 16]
 [ 5 17]
 [ 6 18]
 [ 7 19]]
```

- print(n4.shape)

```
[[ 8 20]
 [ 9 21]
 [10 22]
 [11 23]]]
(3, 4, 2)
```

※ np.transpose()는 2차원 행렬에도 적용할 수 있다

reshape와 transpose 비교

- np.reshape와 np.transpose는 비슷해보이고,
• 실제 결과도 같게 나오는 경우가 있지만 그 진행방식은 전혀 다르다

- n2

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7],  
       [ 8,  9, 10, 11, 12, 13, 14, 15],  
       [16, 17, 18, 19, 20, 21, 22, 23]])
```

- n2.reshape(4, 6)

```
array([[ 0,  1,  2,  3,  4,  5],  
       [ 6,  7,  8,  9, 10, 11],  
       [12, 13, 14, 15, 16, 17],  
       [18, 19, 20, 21, 22, 23]])
```

- n2.transpose(1, 0)

```
array([[ 0,  8, 16],  
       [ 1,  9, 17],  
       [ 2, 10, 18],  
       [ 3, 11, 19],  
       [ 4, 12, 20],  
       [ 5, 13, 21],  
       [ 6, 14, 22],  
       [ 7, 15, 23]])
```

np.reshape는 각 축의 길이를 어떻게 할 것인가이고,
np.transpose는 축의 순서를 어떻게 할 것인가이다

expand_dims 1D → 2D

- `x1 = np.array([0, 1, 2])`
 - `print(x1)`
 - `print("shape:", x1.shape)`
 - `print('-----')`
 - `x1_1 = np.expand_dims(x1, axis=0)`
 - `print(x1_1)`
 - `print("shape:", x1_1.shape)`
 - `print('-----')`
 - `x1_2 = np.expand_dims(x1, axis=1)`
 - `print(x1_2)`
 - `print("shape:", x1_2.shape)`
 - `print('-----')`
- `expand_dims()` 함수로 차원을 늘릴 수 있다

```
[0 1 2]  
shape: (3,)
```

```
-----  
[[0 1 2]]  
shape: (1, 3)
```

```
-----  
[[0]  
 [1]  
 [2]]  
shape: (3, 1)  
-----
```

expand_dims 2D → 3D

- `x1_3 = np.array([[0, 1, 2], [3, 4, 5]])`
- `print(x1_3)`
- `print("shape:", x1_3.shape)`
- `print('-----')`

- `x1_4 = np.expand_dims(x1_3, axis=0)`
- `print(x1_4)`
- `print("shape:", x1_4.shape)`
- `print('-----')`

```
[[0 1 2]
 [3 4 5]]
shape: (2, 3)
-----
[[[0 1 2]
   [3 4 5]]]
shape: (1, 2, 3)
-----
```

newaxis 1D → 2D

- 축을 끼워 넣어 차원을 늘릴 수 있다

- `x2 = np.array([0, 1, 2])`
- `print(x2)`
- `print("shape:", x2.shape)`
- `print('-----')`

- `x2_1 = x2[np.newaxis]`
- `print(x2_1)`
- `print("shape:", x2_1.shape)`
- `print('-----')`

```
[0 1 2]  
shape: (3,)
```

```
-----  
[[0 1 2]]  
shape: (1, 3)  
-----
```

축이 하나 더 생겨 2차원 데이터가 되었다
1D 벡터 → 2D 행렬

- `x2_2 = x2[np.newaxis, :]`
- `print(x2_2)`
- `print("shape:", x2_2.shape)`
- `print('-----')`

결과는 앞과 동일하지만 이와 같이 사용하는 것이 더 명확하다
 행 쪽에 축을 하나 더 넣었다 (1, 3)

- `x2_3 = x2[:, np.newaxis]`
- `print(x2_3)`
- `print("shape:", x2_3.shape)`

열 쪽에 축을 하나 더 넣었다 (3, 1)

```
[[0 1 2]]
shape: (1, 3)
-----
[[0]
 [1]
 [2]]
shape: (3, 1)
```

newaxis 2D → 3D

- `x3 = np.array([[0, 1, 2], [3, 4, 5]])` # 2D 데이터
- `print(x3)`
- `print("shape:", x3.shape)`
- `print('-----')`

- `x3_1 = x3[np.newaxis]`
- `print(x3_1)`
- `print("shape:", x3_1.shape)`
- `print('-----')`

- `x3_2 = x3[np.newaxis, :, :]`
- `print(x3_2)`
- `print("shape:", x3_2.shape)`
- `print('-----')`

위와 같지만 이와 같이 사용하는 것이 더 명확하다
첫 번째 축에 차원을 추가로 끼워넣었다

```
[[0 1 2]
 [3 4 5]]
shape: (2, 3)
-----
```

```
[[[0 1 2]
   [3 4 5]]]
shape: (1, 2, 3)
-----
```

```
[[[0 1 2]
   [3 4 5]]]
shape: (1, 2, 3)
-----
```

- `x3_3 = x3[:, :, np.newaxis]`
- `print(x3_3)`
- `print("shape:", x3_3.shape)`
- `print('-----')`

세 번째 축에 차원을 추가로 끼워넣었다

- `x3_4 = x3[:, np.newaxis, :]`
- `print(x3_4)`
- `print("shape:", x3_4.shape)`

두 번째 축에 차원을 추가로 끼워넣었다

```
[[[0]
  [1]
  [2]]

 [[3]
  [4]
  [5]]]
shape: (2, 3, 1)
-----
[[[0 1 2]]

 [[3 4 5]]]
shape: (2, 1, 3)
```

vstack

- 열의 사이즈가 같으면 두 행렬을 쌓을 수 있다

(2행, 3열)과 (3행 3열)의 두 행렬

- `n5 = np.arange(1, 7).reshape(2, 3)`
- `n6 = np.arange(7, 16).reshape(3, 3)`
- `print(n5)`

```
[[1 2 3]
 [4 5 6]]
```
- `print(n6)`

```
[[ 7  8  9]
 [10 11 12]
 [13 14 15]]
```


vstack

열의 사이즈가 같으면 vstack으로 쌓을 수 있다

- `n7 = np.vstack([n5, n6])`
- `print(n7)`

```
[[ 1  2  3] } n5  
[ 4  5  6] }  
[ 7  8  9] }  
[10 11 12] } n6  
[13 14 15]] }
```

행렬과 벡터의 연결도 가능하다

- `n8 = np.arange(1, 4)`
- `np.vstack([n7, n8])`

```
array([[ 1,  2,  3],  
       [ 4,  5,  6],  
       [ 7,  8,  9],  
       [10, 11, 12],  
       [13, 14, 15],  
       [ 1,  2,  3]])
```

} n7
} n8

hstack

- 같은 방식으로 행의 사이즈가 같으면 두 행렬을 쌓을 수 있다

(2행, 3열)과 (2행 4열)의 두 행렬

- `n9 = np.array(range(1, 7)).reshape(2, 3)`

`np.arange(1, 7).reshape(2, 3)`과 동일

- `n10 = np.array(range(7, 15)).reshape(2, 4)`

- `print(n9)`
[[1 2 3]
 [4 5 6]]
- `print(n10)`
[[7 8 9 10]
 [11 12 13 14]]

hstack

- `n11 = np.hstack([n9, n10])`

- `print(n11)`

- `print('-----')`

행렬과 벡터의 연결도 가능하다

- `n12 = np.array(range(1, 15)).reshape(2, -1)` # 14개의 원소를 만든 뒤, 2행을 갖추도록 함

- `print(n12)`

- `print('-----')`

- `np.hstack([n11, n12])`

```
[[ 1  2  3  7  8  9 10]
 [ 4  5  6 11 12 13 14]]
```

```
-----
[[ 1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14]]
```

```
array([[ 1,  2,  3,  7,  8,  9, 10,  1,  2,  3,  4,  5,  6,  7],
       [ 4,  5,  6, 11, 12, 13, 14,  8,  9, 10, 11, 12, 13, 14]])
```

n11

n12

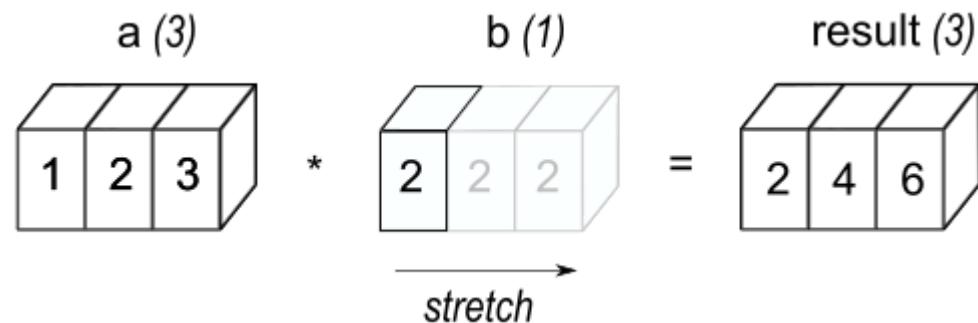
Broadcasting

- 넘파이는 브로드캐스팅 기능을 제공하여 형상이 맞지 않아도 연산이 가능한(되게 하는) 경우가 있다
- 같은 내용을 전파한다는 뜻으로 broadcasting이라 한다
- 브로드캐스팅의 결과 저차원 배열이 고차원 배열로 확장된다

- 대표적인 경우는 배열과 스칼라 값을 연산할 때이다
- 간단히는, 스칼라 값이 늘어나는 것으로 생각할 수 있다
(실제로는 원본 스칼라 값이 반복적으로 사용됨)

- `n13 = np.array([1, 2, 3, 4, 5])`

- `print(n13 * 2)` `[2 4 6 8 10]`
- `print(n13 + 2)` `[3 4 5 6 7]`



Broadcasting

- 브로드캐스팅은 스칼라와의 연산 뿐 아니라, 배열 또는 행렬 사이에서도 가능하다
- 다만, 브로드캐스팅하는 차원의 길이가 같거나 비교 대상 차원의 둘 중 하나의 길이가 1이어야 한다
- n15는 (3,) 이므로 누락된 차원을 1로 간주하여 (1, 3)으로 확장한 뒤([[0 1 2]]) 브로드캐스팅을 수행한다

- n14 = np.arange(1, 10).reshape(3, 3)
- print(n14)

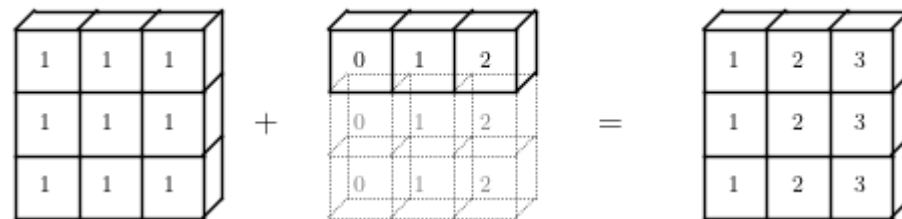
- print('-----')
• n15 = np.arange(3)
• print(n15)
- ```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

[0 1 2] 내부적으로 [[0 1 2]]로 만들
```

- print('-----')
  - print(n14 + n15)
- ```
[[ 1  3  5]
 [ 4  6  8]
 [ 7  9 11]]
```

1행은 계산이 되었는데,
2행과 3행을 계산할 부분이 없어 브로드캐스팅

np.ones((3, 3)) + np.arange(3)



Broadcasting

- 다음은 (3, 1) 행렬과 (1, 3) 행렬 사이의 브로드캐스팅이다
- 이 경우에는 각 비교 대상 차원에서 한 쪽의 길이가 1이므로 브로드캐스팅이 수행될 수 있다

- `n16 = np.arange(3).reshape(3, 1)`
- `print(n16)`
- `print(n16.shape)`

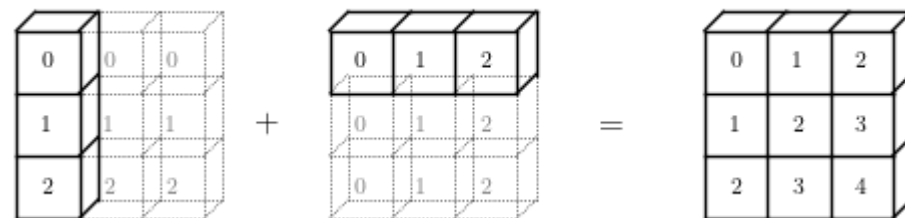
- `print('-----')`
- `n17 = np.arange(3).reshape(1, 3)`
- `print(n17)`
- `print(n17.shape)`

- `print('-----')`
- `print(n16 + n17)`

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

한쪽은 열을, 한쪽은 행을 브로드캐스팅하면
연산이 되므로 브로드캐스팅 수행

`np.arange(3).reshape((3, 1)) + np.arange(3)`



Broadcasting이 되지 않는 경우

- 다음은 브로드캐스팅하는 배열의 차원의 길이가 같지 않아 성립되지 않는 경우이다
- n19를 (1, 4)와 같이 확장한다고 하여도
- 첫 번째 비교 대상이 n18의 3과 n19의 4이므로 브로드캐스팅이 일어날 수 없다

- n18 = np.arange(1, 13).reshape(4, 3)
- print(n18)

- print('-----')
- n19 = np.arange(4)
- print(n19)

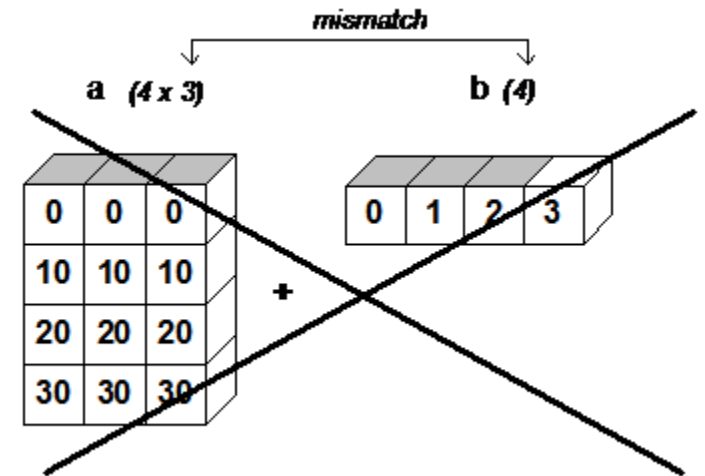
- print('-----')
- print(n18 + n19)

Error

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

```
-----
[0 1 2 3]
-----
```

ValueError



Broadcasting이 되지 않는 경우

- 다음도 n20의 형상이 (3, 2)이므로 비교 대상인 n20의 2와 n21의 3이 길이가 같지 않아 브로드캐스팅이 되지 않는다
- 만약 n20의 형상이 (3, 1)이라면 n20쪽이 확장되어 브로드캐스팅이 될 수 있다

- `n20 = np.arange(6).reshape(3, 2)`
- `print(n20)`
- `print(n20.shape)`

- `print('-----')`
- `n21 = np.arange(3).reshape(1, 3)`
- `print(n21)`
- `print(n21.shape)`

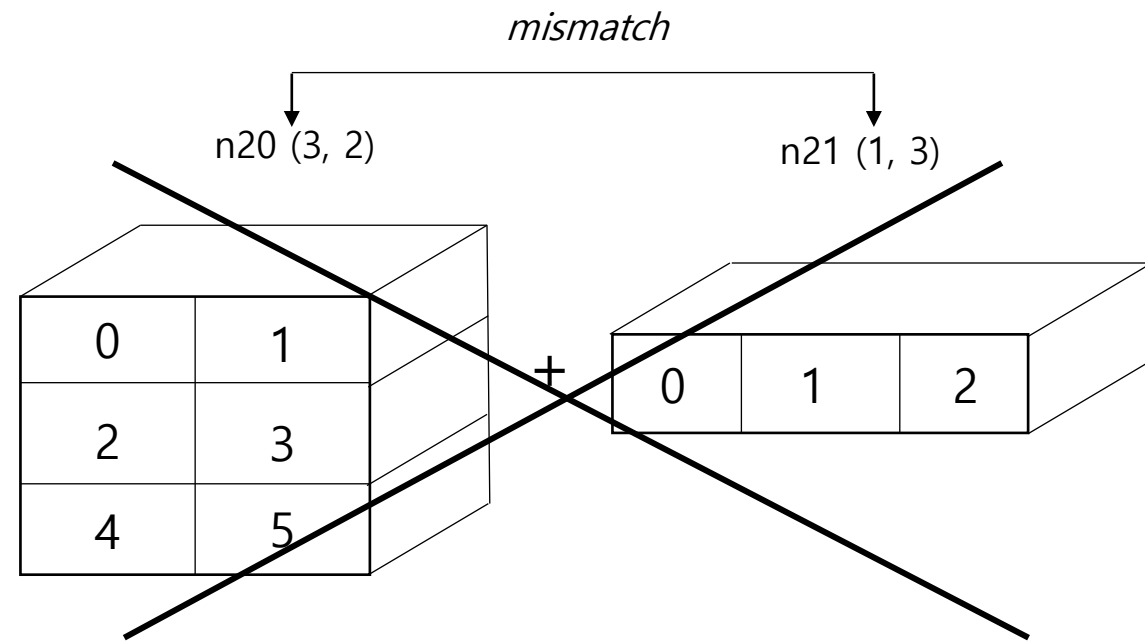
- `print('-----')`
- `print(n20 + n21)`

Error

```
[[0 1]
 [2 3]
 [4 5]]
(3, 2)
```

```
-----
[[0 1 2]]
(1, 3)
```

ValueError



연습문제 1

- 배열을 아래와 같이 변경 전 모양으로 만든 뒤,
- 변경 후로 수정하시오

변경 전:

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [10 11 12 13]]
```

shape: (3, 4)

변경 후:

```
[[ 1  2  3  4  5  6]
 [ 7  8 10 11 12 13]]
```

shape: (2, 6)

연습문제 2

- reshape()를 이용하여 앞에서 만든 2D 배열을 1D 배열로 변경하시오

```
[ 1  2  3  4  5  6  7  8 10 11 12 13]
```

연습문제 3


- np.arange() 함수로 24개의 숫자를 만드시오
- 만들어진 숫자는 (2, 3, 4)의 shape를 갖도록 하시오
- 만들어진 3D 데이터를 (4, 3, 2)의 shape를 갖도록 변형하시오

```
[[[ 0 12]
   [ 4 16]
   [ 8 20]]

 [[ 1 13]
   [ 5 17]
   [ 9 21]]

 [[ 2 14]
   [ 6 18]
   [10 22]]

 [[ 3 15]
   [ 7 19]
   [11 23]]]
(4, 3, 2)
```



```
[[[ 0 1 2 3]
   [ 4 5 6 7]
   [ 8 9 10 11]]

 [[12 13 14 15]
   [16 17 18 19]
   [20 21 22 23]]]
(2, 3, 4)
```

연습문제 4

- ① 행렬간 hstack의 예를 만드시오
- ② 행렬간 vstack의 예를 만드시오