파이썬의 브로드캐스팅

Calories from Carbs, Proteins, Fats in 100g of different foods:

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

행렬의 네 열 안의 수의 합을 구하고 행렬 전체를 나눠서 네가지 음식 안의 탄단지가 주는 칼로리의 백분율을 구한다

- for 문을 쓰지 않고 한다

```
# 행렬 생성
import numpy as np
A= np.array([[56.0,0.0,4.4,68.0],
             [1.2,104.0,52.0,8.0],
             [1.8,135.0,99.0,0.9]])
print(A)
```

```
[[ 56.    0.    4.4  68. ]
 [  1.2 104.   52.    8. ]
 [  1.8 135.   99.    0.9]]
```

```
# 각 열마다 총합 구하기
cal=A.sum(axis=0) # 세로로 더해라 가로축은 axis 0 이다
cal2 = A.sum(axis=1) # 가로로 더해라
print(cal)
print(cal2)
```

```
[ 59.  239.  155.4  76.9]
[128.4 165.2 236.7]
```

```
# 백분율 행렬 만들기
print(cal.reshape(1,4)) #원래 1차원 행렬을 1by4 2차원 행렬로 만들어준다 근데 여기서 굳이 할 필요는 없음
# reshape 은 상수 시간이 소요된다.
percentage = 100* A/cal#.reshape(1,4)
print(percentage)
```

```
[[ 59.  239.  155.4  76.9]]
[[94.91525424  0.          2.83140283 88.42652796]
 [ 2.03389831 43.51464435 33.46203346 10.40312094]
 [ 3.05084746 56.48535565 63.70656371  1.17035111]]
```

```
B=np.array([1,2,3,4])
print(B+100) # 100을 자동으로 4*1 벡터로 만들어주어 각각의 원소에 대해 연산 가능하게한다.
```

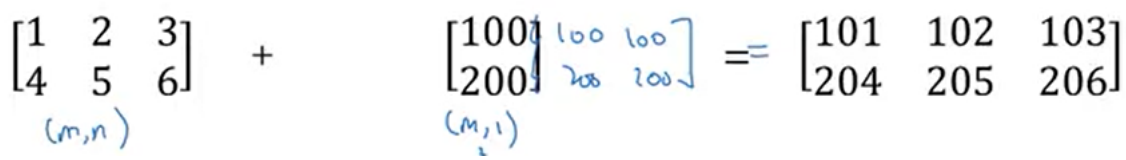
```
[101 102 103 104]
```

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\\ 100 & 200 & 300\end{bmatrix} = \begin{bmatrix}101 & 202 & 303\\ 104 & 205 & 306\end{bmatrix}$$

(m,n) (2,3) (1,n) ⇝ (m,n) (2,3)

```
A=np.array([[1,2,3],
            [4,5,6]])
B=np.array([100,200,300])
print(A+B)
```

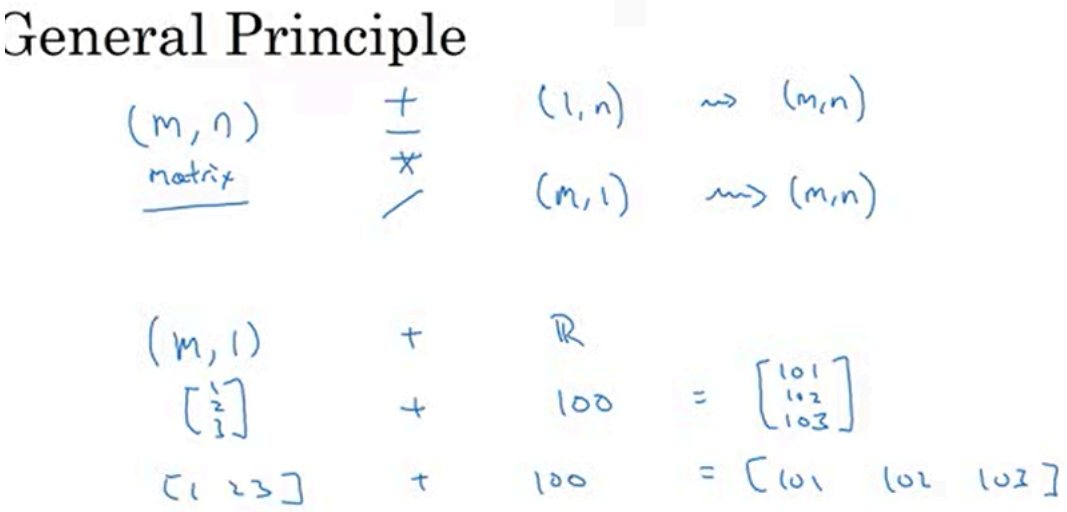
```
[[101 202 303]
 [104 205 306]]
```



```
1 B=np.array([[100],[200]])
2 print(A+B)
```

```
[[101 102 103]
 [204 205 206]]
```

일반화하면



파이썬과 넘파이 벡터

파이썬의 유연성은 장단점이 될 수 있다

- 단점: 잘 모르면 가끔 이상하고 찾기 어려운 에러가 발생한다. 에러가 안 뜨고 예상 못 한 값이 나올 수 있음..

```
1 import numpy as np
2
3 a= np.random.randn(5) # 가우시안 분포를 따르는 변숫값 5개를 배열 a에 저장한다.
4 print(a)
```

```
[ 2.25839726  0.29299917 -0.38316699  1.15719586 -1.24652479]
```

```
1 print(a.shape) # rank 1 array , 행벡터 열벡터도 아님 -> 직관적이지 않은 결과를 도출한다
```

```
(5,)
```

```
1 print(a.T)
2 print(a.T.shape)
```

```
[ 2.25839726  0.29299917 -0.38316699  1.15719586 -1.24652479]
(5,)
```

```
1 print(np.dot(a,a.T)) # 행렬이 나와야하는데 상숫값이 나온다
```

```
8.225949978886533
```

rank 1 array 는 아예 사용하지 마라

- 결과가 직관적이지 않음

```
1 a=np.random.randn(5,1)
2 print(a) # 괄호가 두개임
3 print(a.shape)
```

```
[[ 0.91083025]
 [-0.12195084]
 [ 0.84231119]
 [-0.78111727]
 [ 0.13557301]]
(5, 1)
```

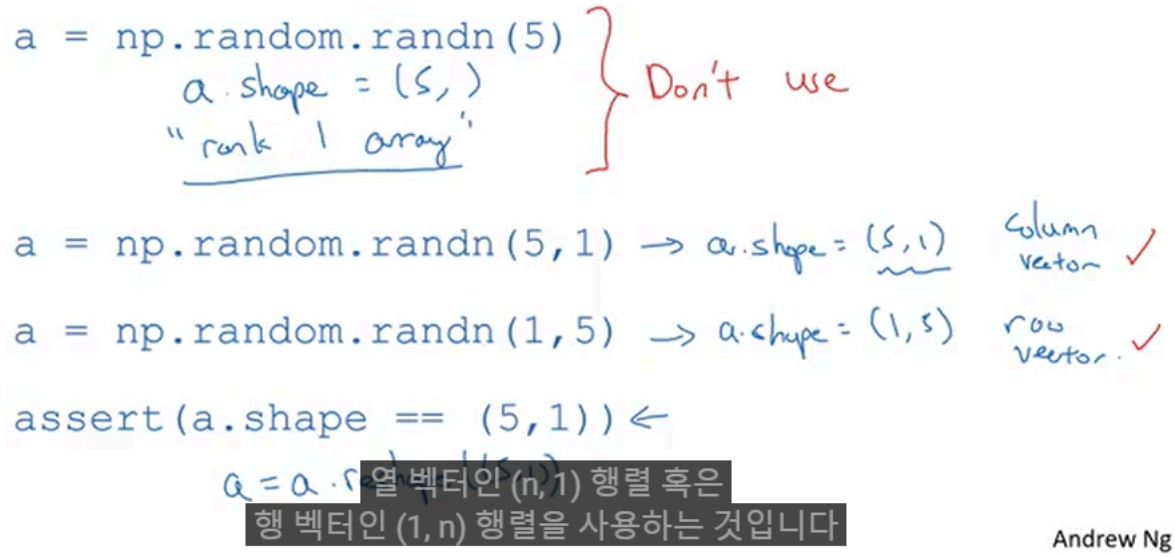
```
1 print(a.T)
```

```
[[ 0.91083025 -0.12195084  0.84231119 -0.78111727  0.13557301]]
```

```
1 print(np.dot(a,a.T)) # 벡터의 외적 => 행렬
2 print(np.dot(a.T,a))
```

```
[[ 0.82961175 -0.11107652  0.76720252 -0.71146524  0.123484  ]
 [-0.11107652  0.01487201 -0.10272056  0.09525791 -0.01653324]
 [ 0.76720252 -0.10272056  0.70948814 -0.65794382  0.11419467]
 [-0.71146524  0.09525791 -0.65794382  0.61014419 -0.10589842]
 [ 0.123484   -0.01653324  0.11419467 -0.10589842  0.01838004]]
[[2.18249614]]
```

```
1 assert(a.shape==(5,1)) # 검증 코드
```



로지스틱 회귀의 손실 비용함수

$$\hat{y} = \sigma(w^T x + b) \qquad \sigma(z) = \frac{1}{1+e^{-z}}$$

$\hat{y}$ = $P(y=1 \mid x)$

$y=1$) $P = \hat{y}$

$y=0$) $P = 1-\hat{y}$

) 1개의 $\hat{y}$로 2개 표현

$\therefore P(y|x) = \hat{y}^{y}(1-\hat{y})^{(1-y)}$ $\quad y=1$

예측을 잘할수록 $\hat{y}$은 1에 가깝다

∴ 1 인것만 잘예측하면됨

log 함수는 강한단조증가이기 때문에 $\log P(y|x)$를 증가시키는 건 $P(y|x)$를 증가시키는 것과 같다

$\log P(y|x) = \log \hat{y}^{y}(1-\hat{y})^{(1-y)} = y\log\hat{y} + (1-y)\log(1-\hat{y}) = -\mathcal{L}(\hat{y}, y)$

작을수록 최대화, $\mathcal{L}$ 함수는 최소화

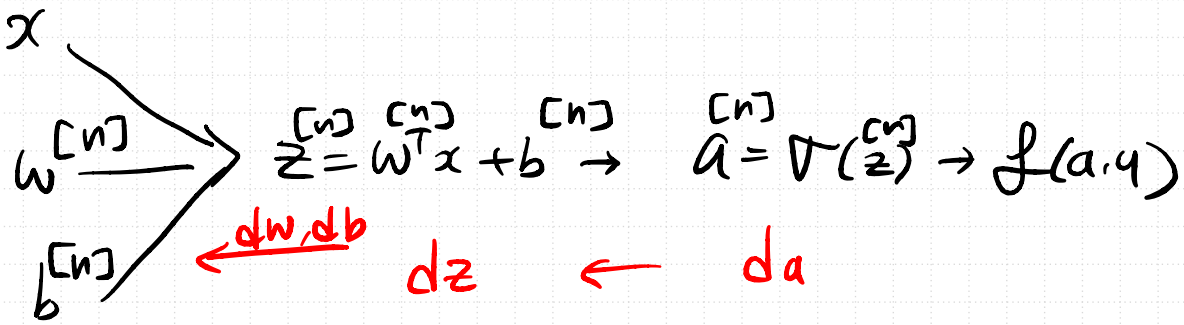
독립동일분포이기 때문에 다 곱하기 가능

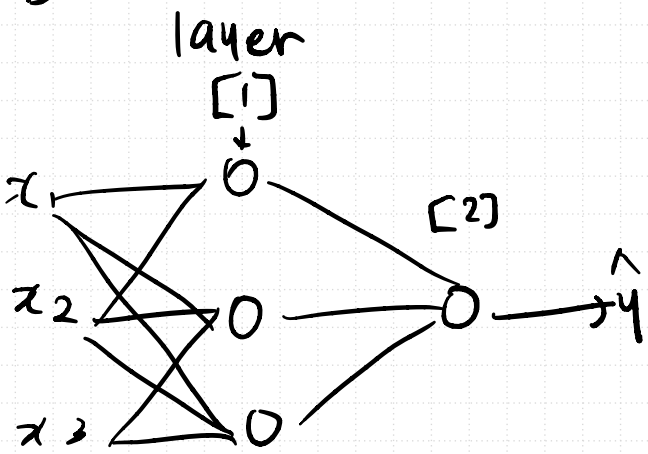
$$\log p(\cdots) = \sum_{i=1}^{m} \log P(y^{(i)} \mid x^{(i)}) = -\sum_{i=1} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$$\text{Cost} = J(w,b) = \left(\frac{1}{m}\right)\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

for scaling

NN 구현





$a^{[0]} = X$

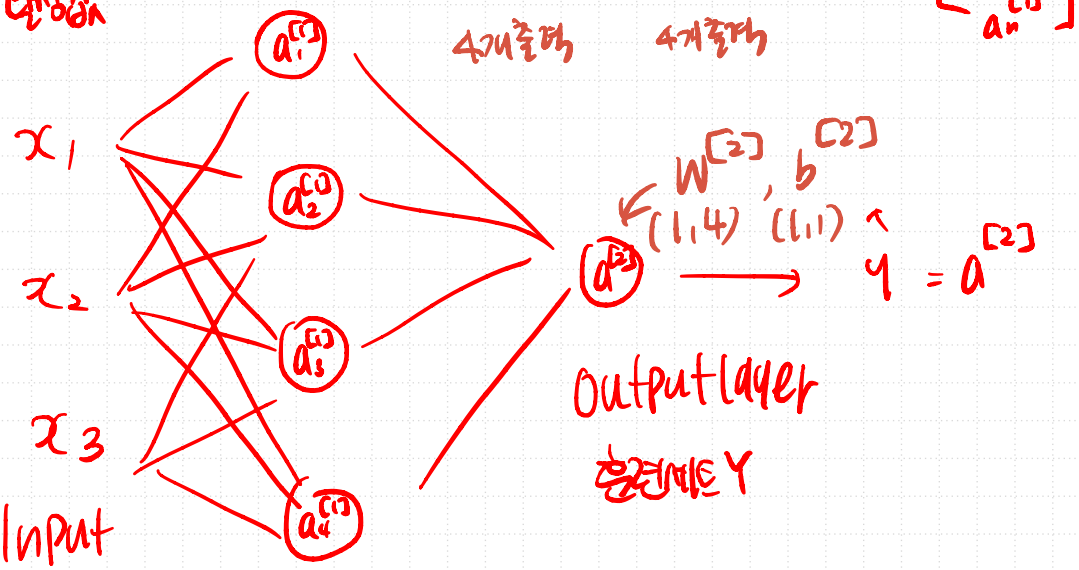
입력층의 activation

$a^{[1]}$ ← $W^{[1]}$, $b^{[1]}$ 곱해져

(4,3) (4,1)

4개출력 4개출력

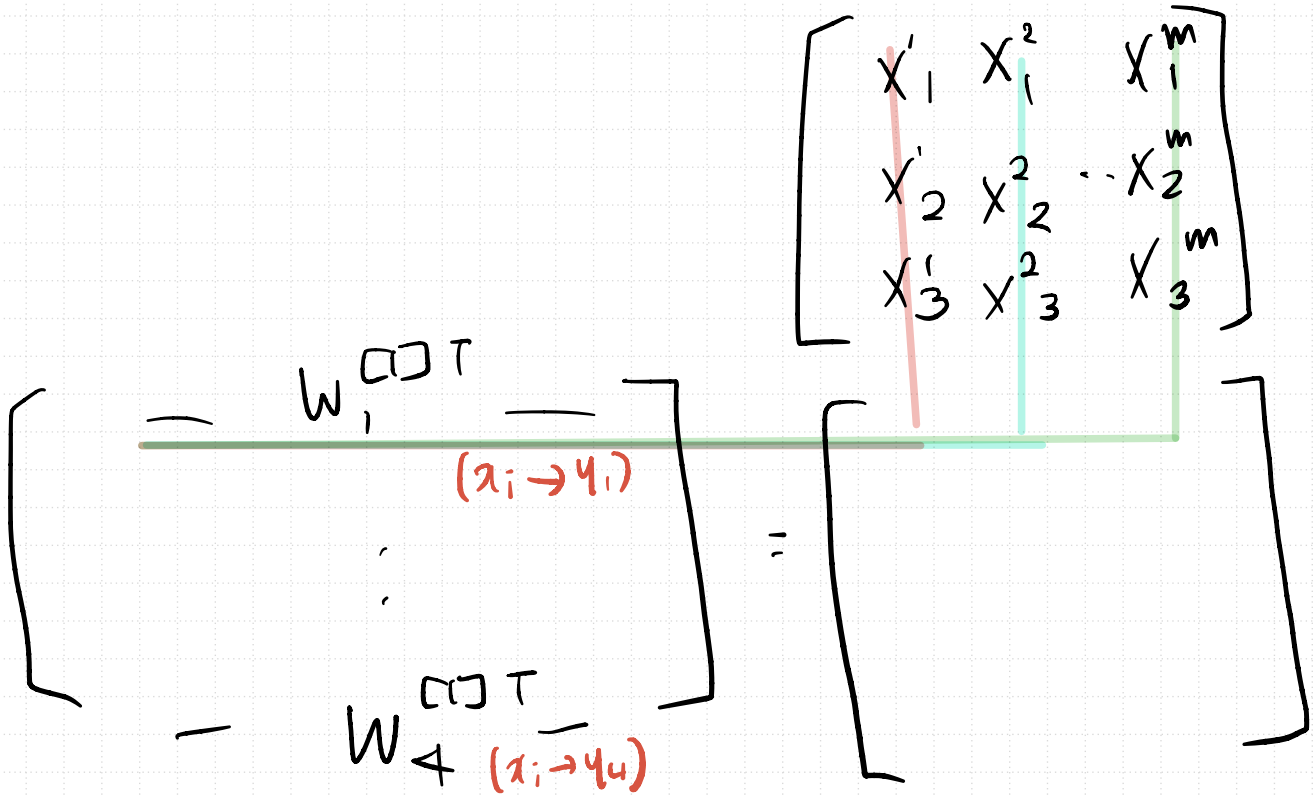
$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_n^{[1]} \end{bmatrix}$

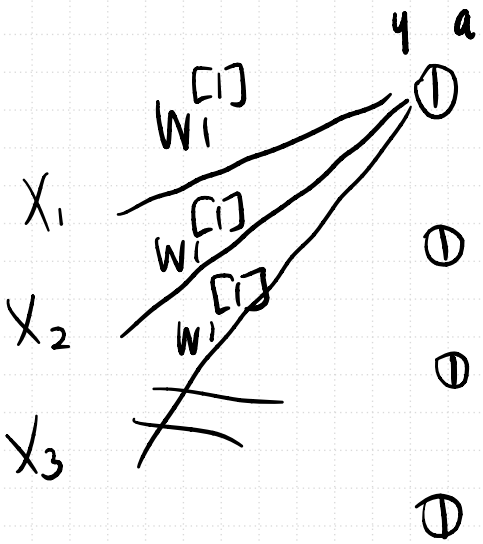


⇒ 2층 NN

(입력 layer는 세지 않는다)

m 샘플 벡터화 표현


X_1^1 X_1^2 X_1^m
X_2^1 X_2^2 X_2^m
X_3^1 X_3^2 X_3^m
$W_1^{[1]T}$
$(x_i \to y_1)$
$W_4^{[1]T}$
$(x_i \to y_4)$
=


y a
$W_1^{[1]}$
$W_2^{[1]}$
$W_3^{[1]}$
X_1
X_2
X_3
1
1
1
1