분석 멘토링 C조

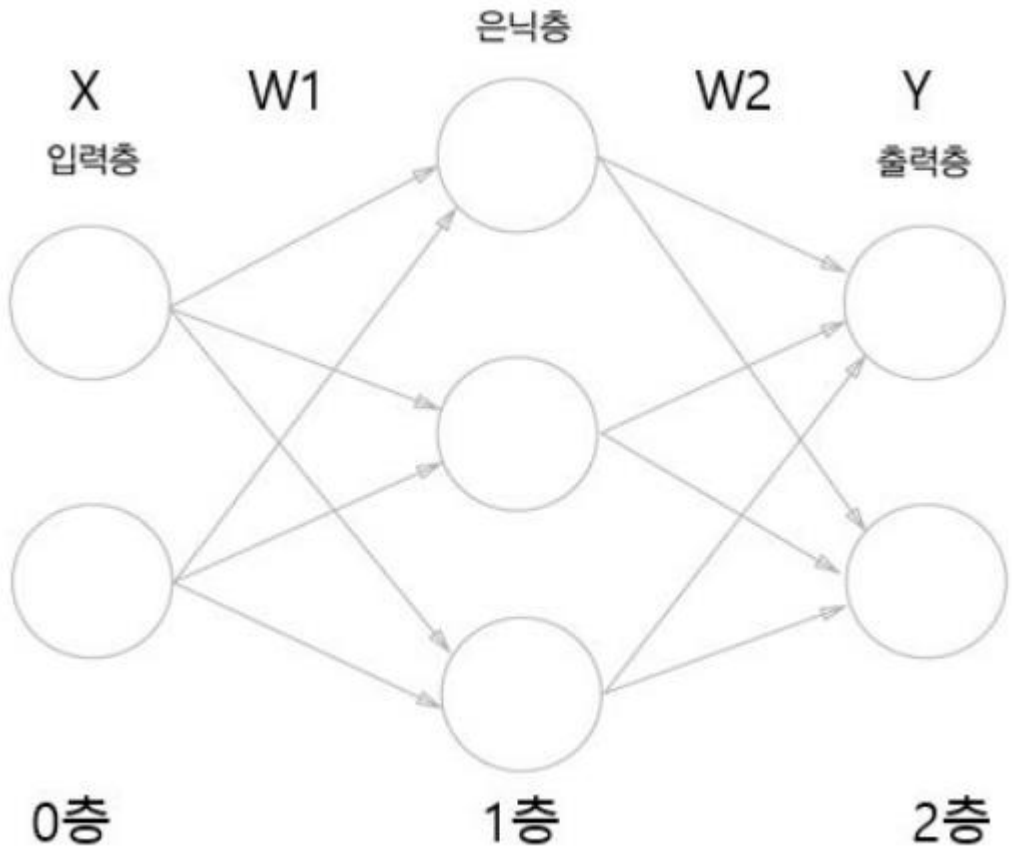
2주차

발표자: 황의린

INDEX

0. 신경망

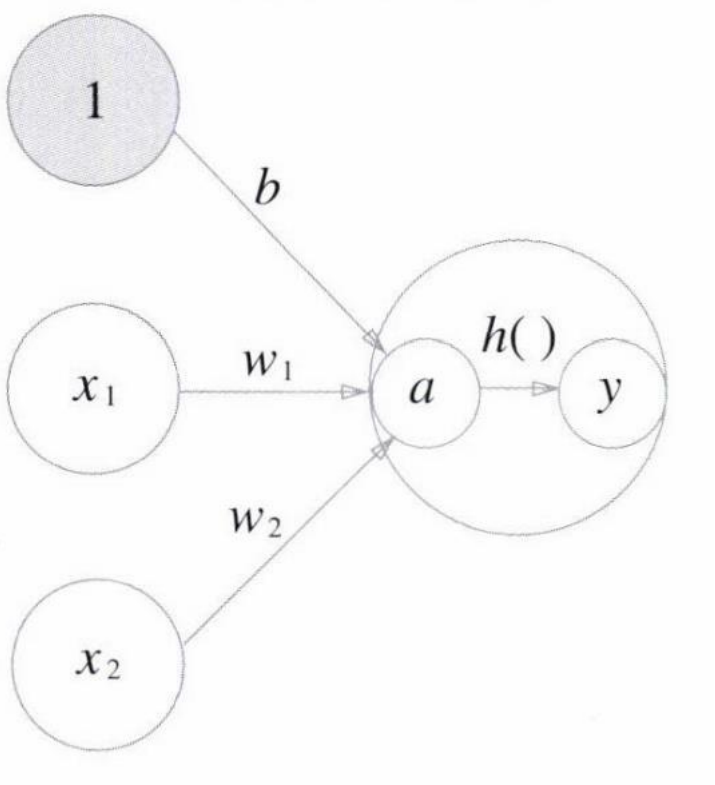
0. 신경망이란



- 단층 퍼셉트론 (AND,OR 게이트)
- 다층 퍼셉트론 (XOR 게이트)

1. 활성화 함수

1. 활성화 함수

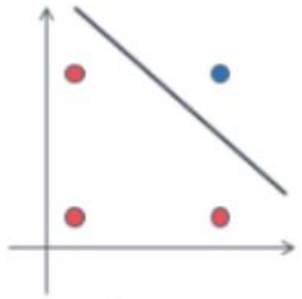


활성화 함수 h(x)

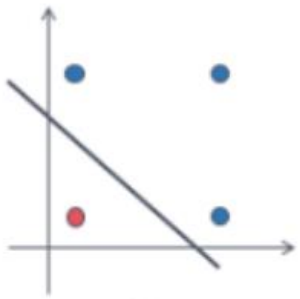
:입력 신호를 출력 신호로 변환해 주는 함수

- 시그모이드 함수
- 계단 함수
- ReLU

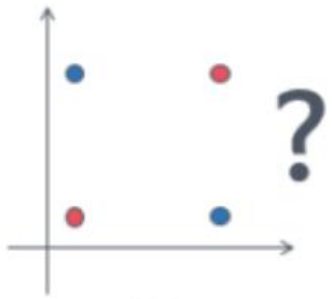
1. 활성화 함수 – 활성화 함수를 사용하는 이유



AND		
Input_A	Input_B	Output
0	0	0
0	1	0
1	0	0
1	1	1



OR		
Input_A	Input_B	Output
0	0	0
0	1	1
1	0	1
1	1	1



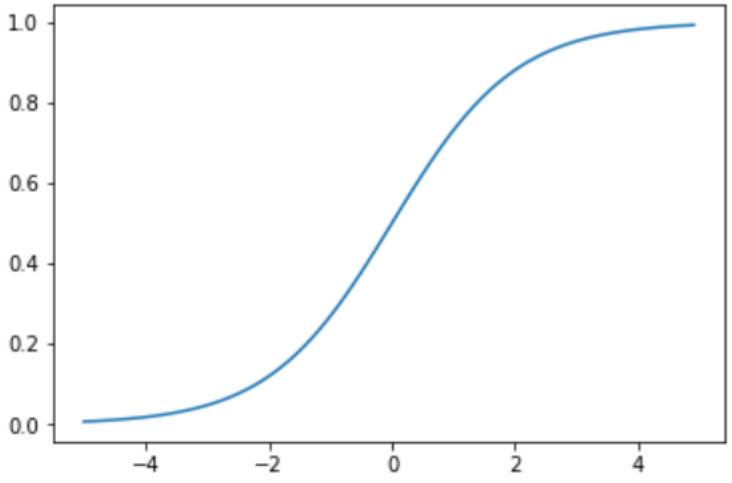
XOR		
Input_A	Input_B	Output
0	0	0
0	1	1
1	0	1
1	1	0

활성화 함수: 데이터를 비선형으로 바꿔주는 역할

1. 활성화 함수

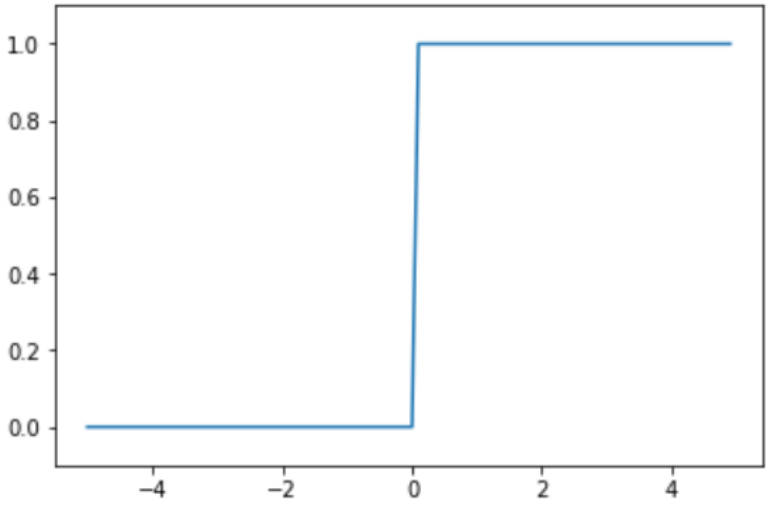
시그모이드 함수

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$



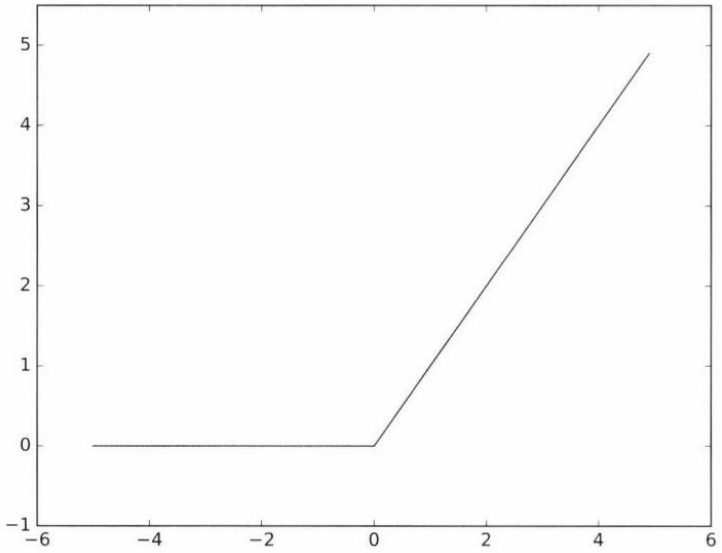
계단 함수

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$



ReLU

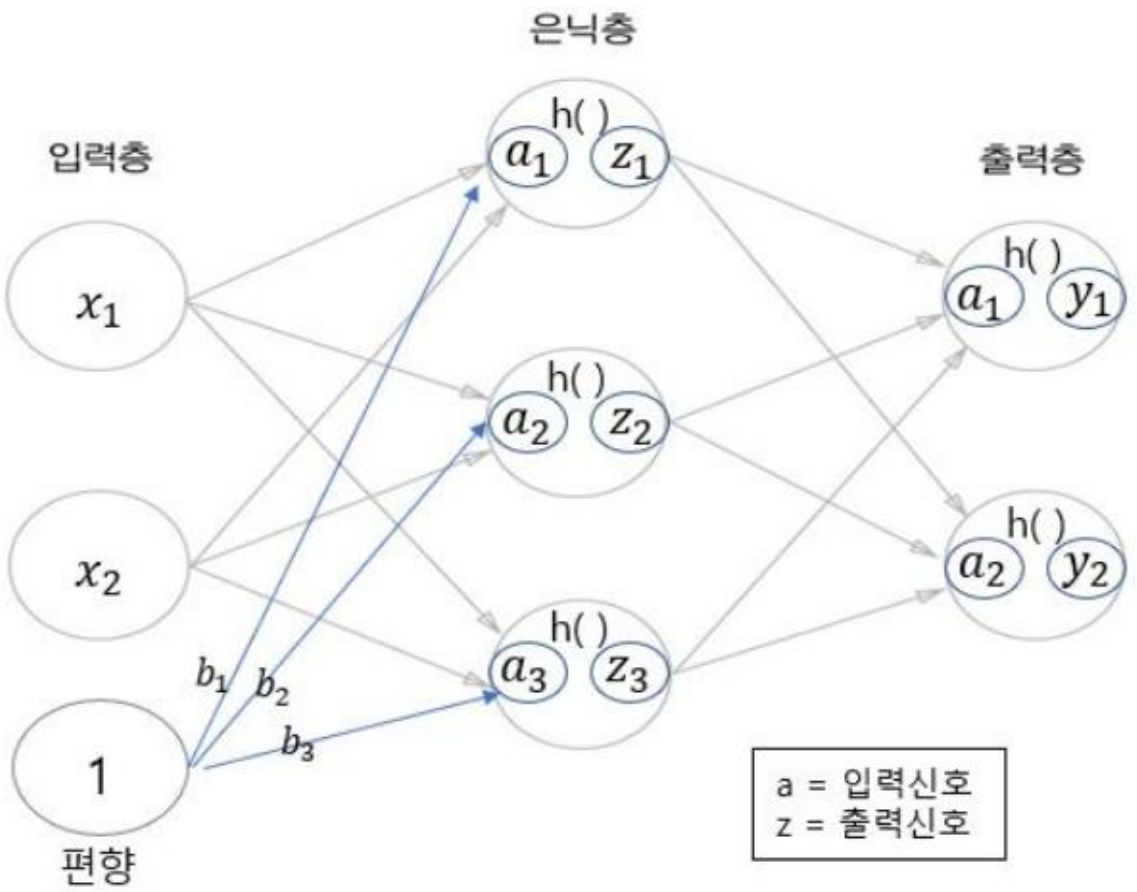
$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$



1. 활성화 함수

● 활성화 함수란

- 입력신호를 출력신호로 변환해주는 함수.



은닉층의 입력값

$a = X \cdot W + b$

$a \leq 0;\ z = h(a) = 0$

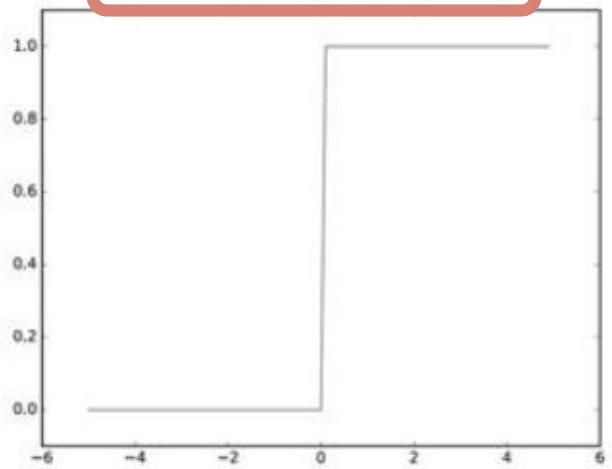
$a > 0;\ z = h(a) = 1$

입력에 대해 출력이 불연속인 결과

다층 퍼셉트론

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$

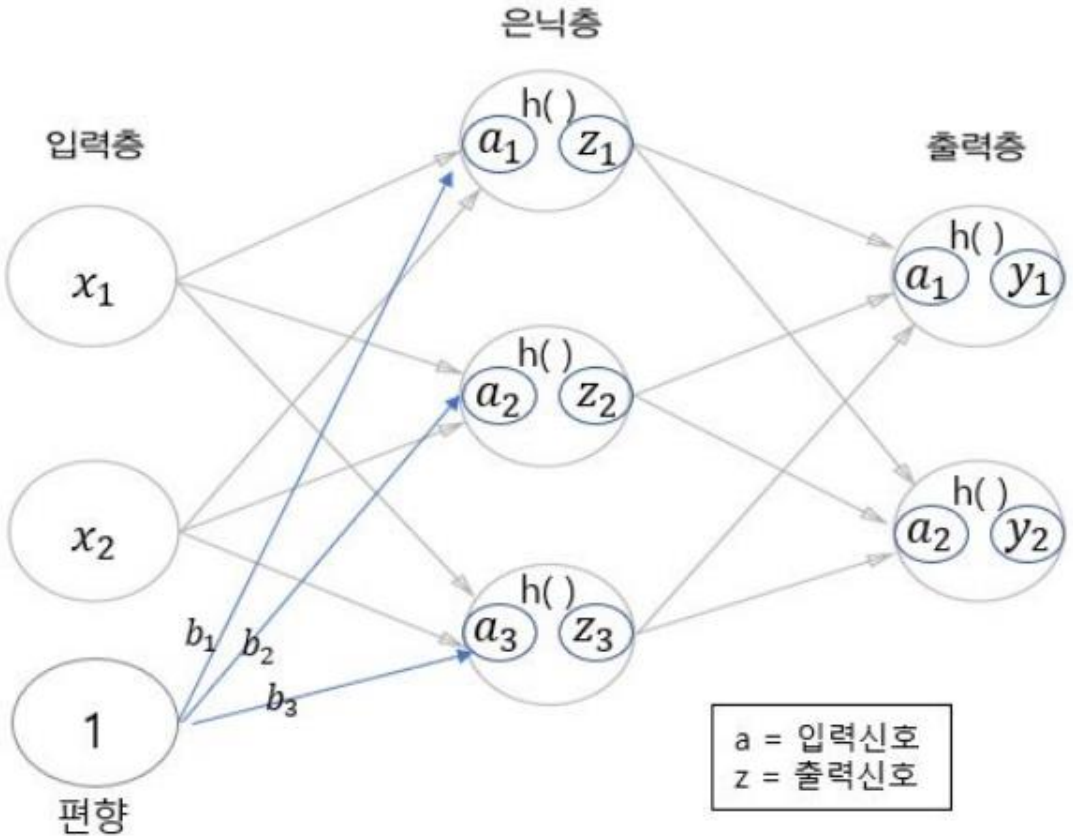
Step 함수



1. 활성화 함수

● 활성화 함수란

- 입력신호를 출력신호로 변환해주는 함수

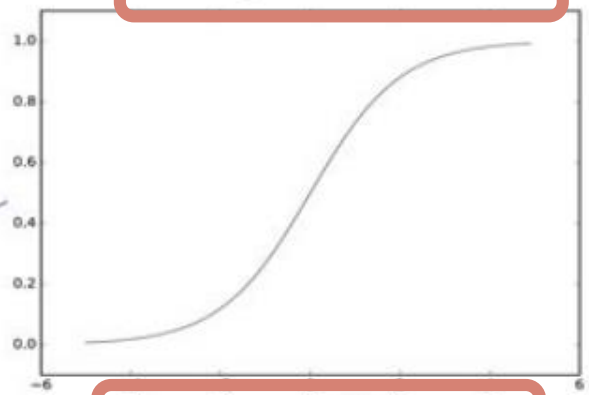


은닉층의 입력값

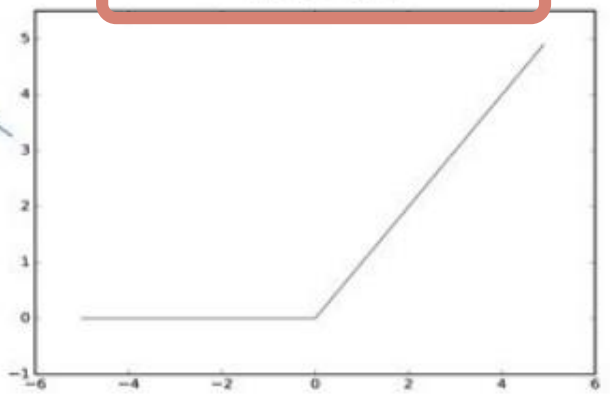
$a = X \cdot W + b$

입력에 대해 출력이 연속인 결과

Sigmoid 함수



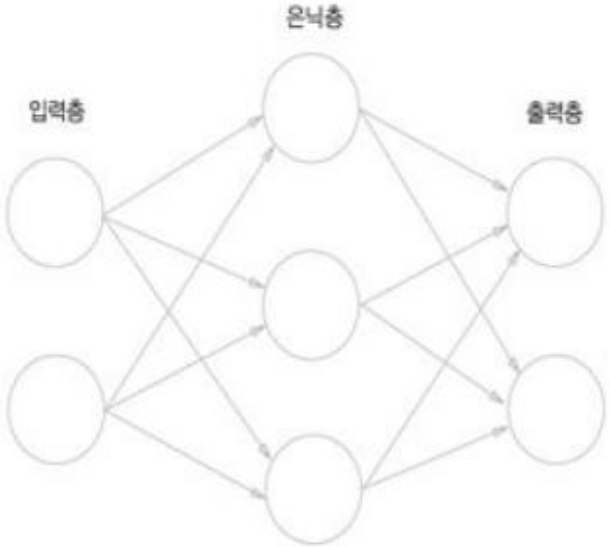
ReLU 함수



1. 활성화 함수

활성화 함수로 비선형 함수만 사용해야 함

- 은닉층이 없는 네트워크로 표현이 됨. 즉 신경망의 층을 깊게하는 의미가 없어진다

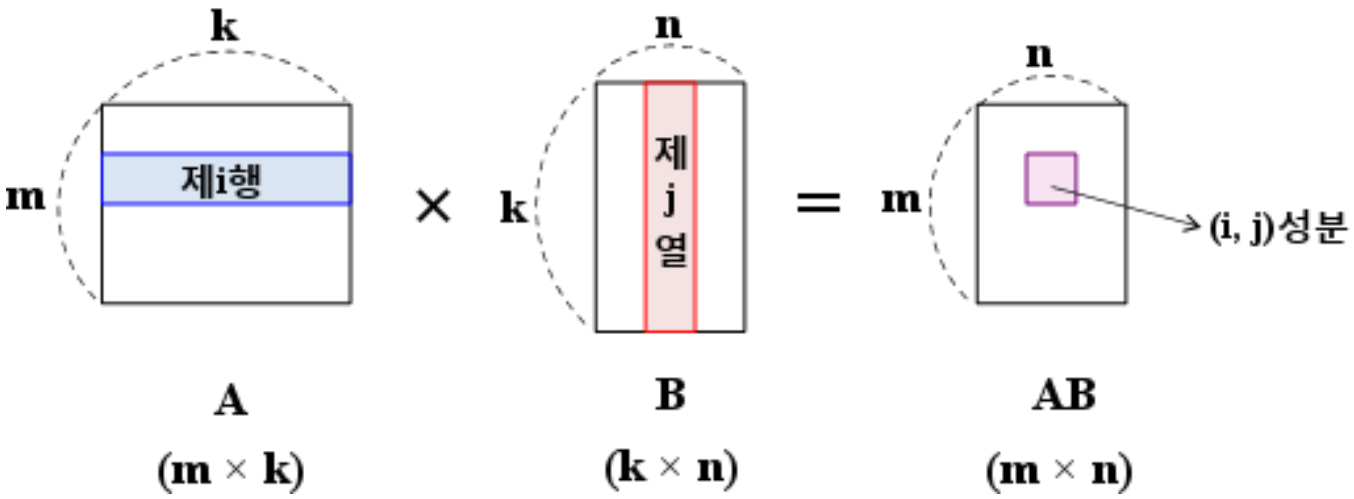


  - 2층 신경망에 $h(x) = cx$라는 선형 활성화 함수를 가정해보면(편향은 0을 가정한다),
    1층의 입력값 : $x \cdot w_1$ 1층의 출력값 : $c(x \cdot w_1)$
    2층의 입력값 : $c(x \cdot w_1) \cdot w_2$ 2층의 출력값 : $c^2 \cdot (x \cdot w_1)w_2$

  - 이 결과는, 그저 입력 x에 가중치 $w_1 \cdot w_2$를 한 뒤, $h(x) = c^2x$ 라는 활성화 함수를 사용한 것과 같다. (1층 신경망으로 표현 가능해 진다)
    즉, 활성화함수로 선형함수를 쓰면, 은닉층을 썼지만 은닉층이 없는 네트워크와 동일함

2. 다차원 배열의 계산

2. 다차원 배열의 계산



$$AB=\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}=\begin{pmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{pmatrix}$$

2. 다차원 배열의 계산

```
>>> import numpy as np
>>>
>>> A = np.array([[1, 2], [3, 4], [5, 6]])
>>> B = np.array([[1, 2, 3], [4, 5, 6]])
>>> C = np.array([1, 2])
>>>
>>> np.dot(A, B)
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51]])
>>> np.dot(B, A)
array([[22, 28],
       [49, 64]])
```

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

3*2 2*3

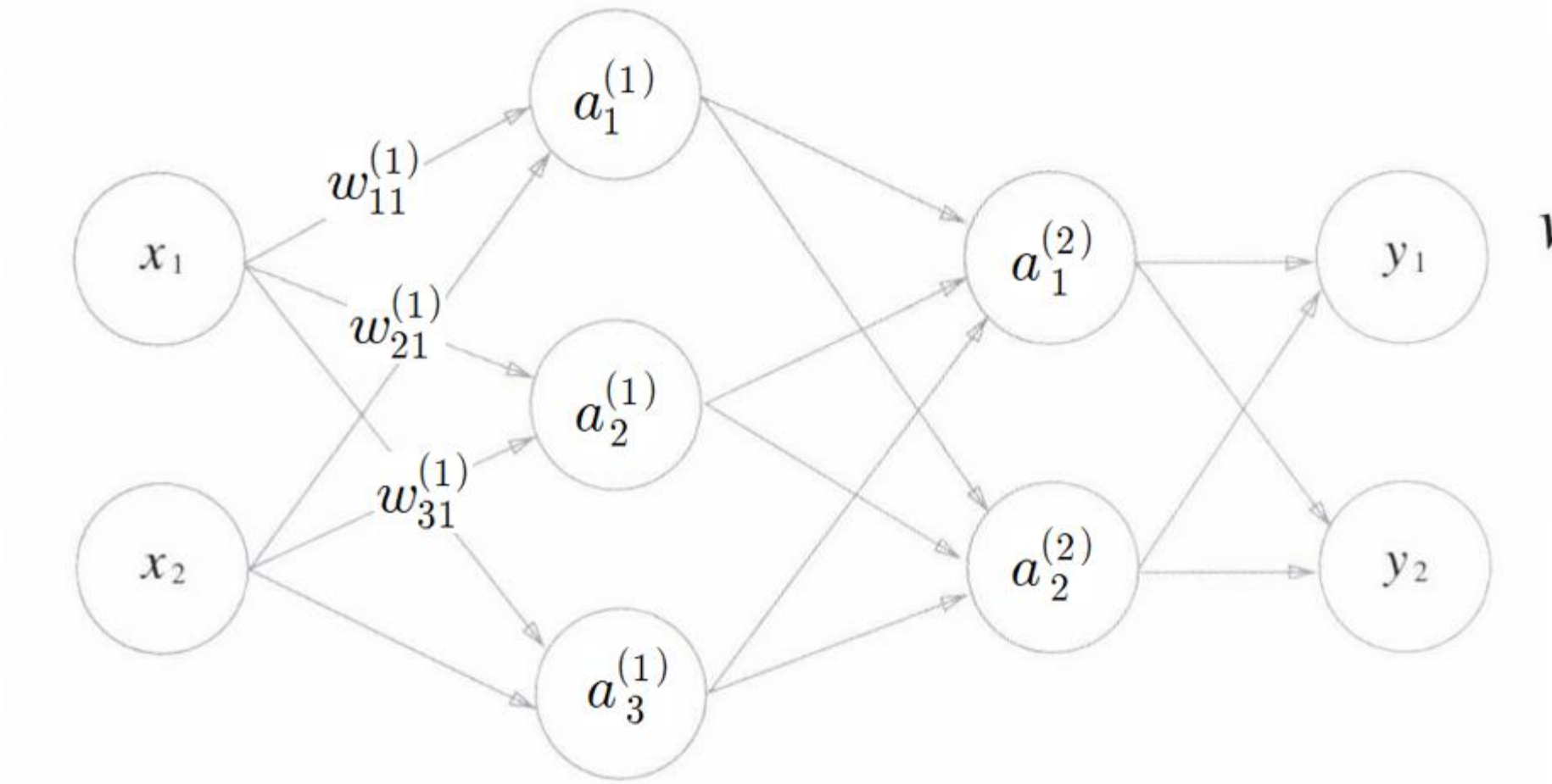
c.f.) numpy에서 행렬의 계산

np.dot(A,B) : 행렬의 곱셈

A*B : 행렬의 원소별 곱셈

3. 3층 신경망 구현하기

3. 3층 신경망 구현하기



$W_{1\,2}^{(1)}$

1층의 가중치

앞 층의 2번째 뉴런

다음 층의 1번째 뉴런

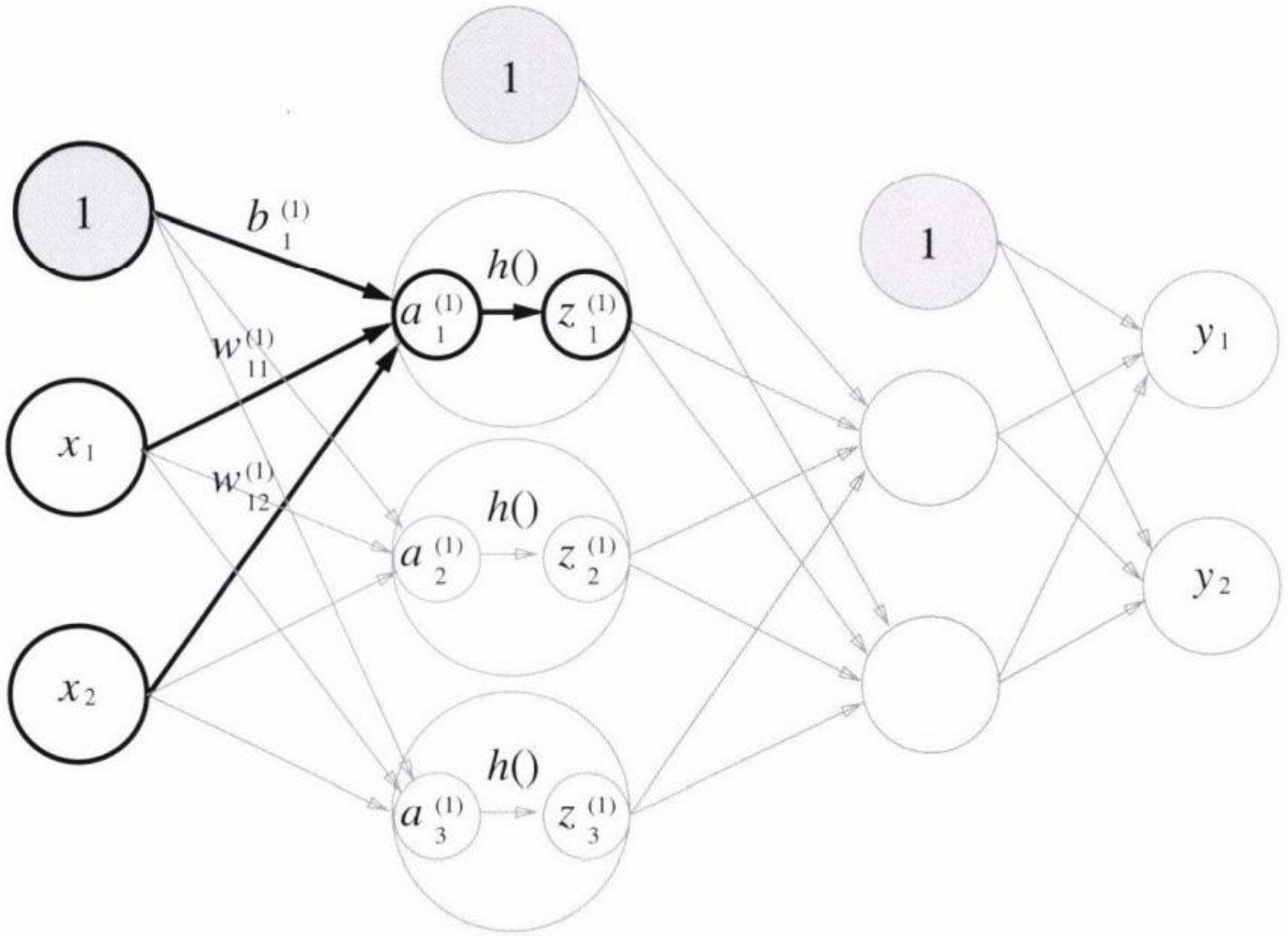
$a_1^{(2)}$

2층의 뉴런

(편향 제외)
1번째 노드

3. 3층 신경망 구현하기

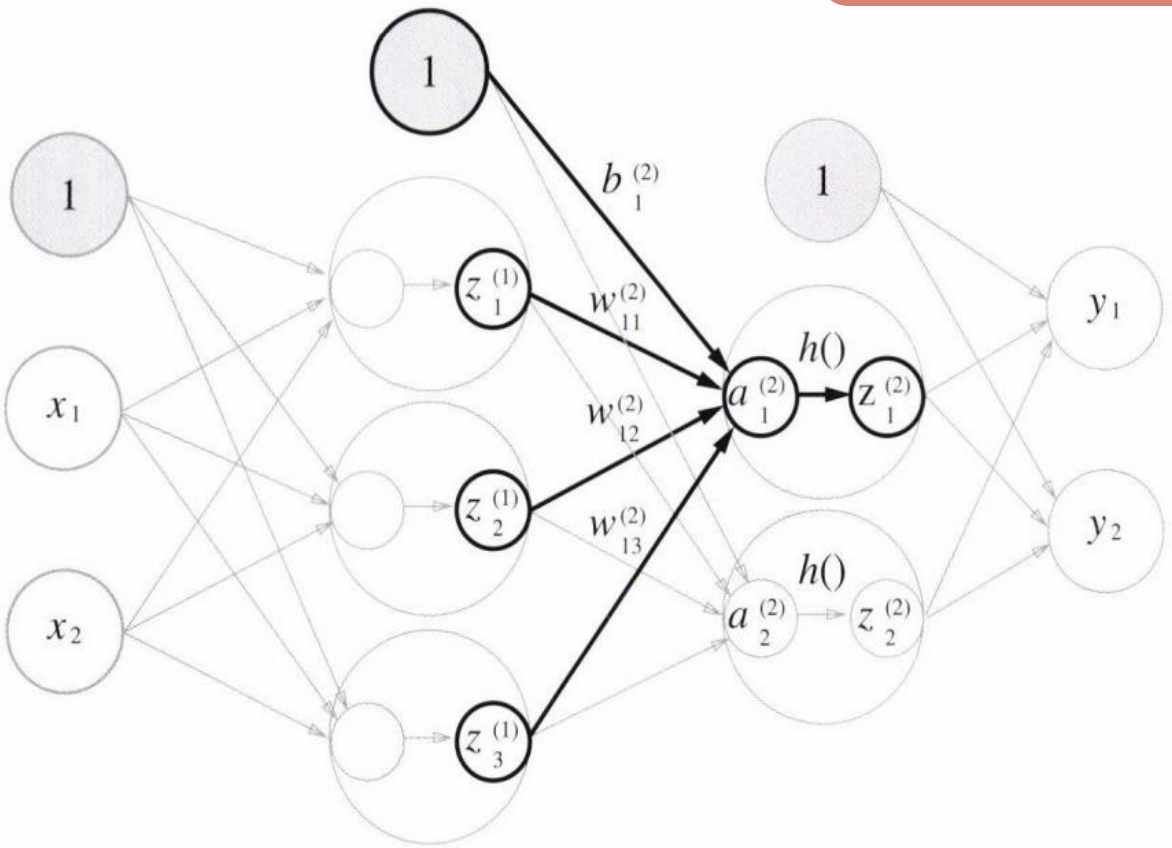
$$a_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$



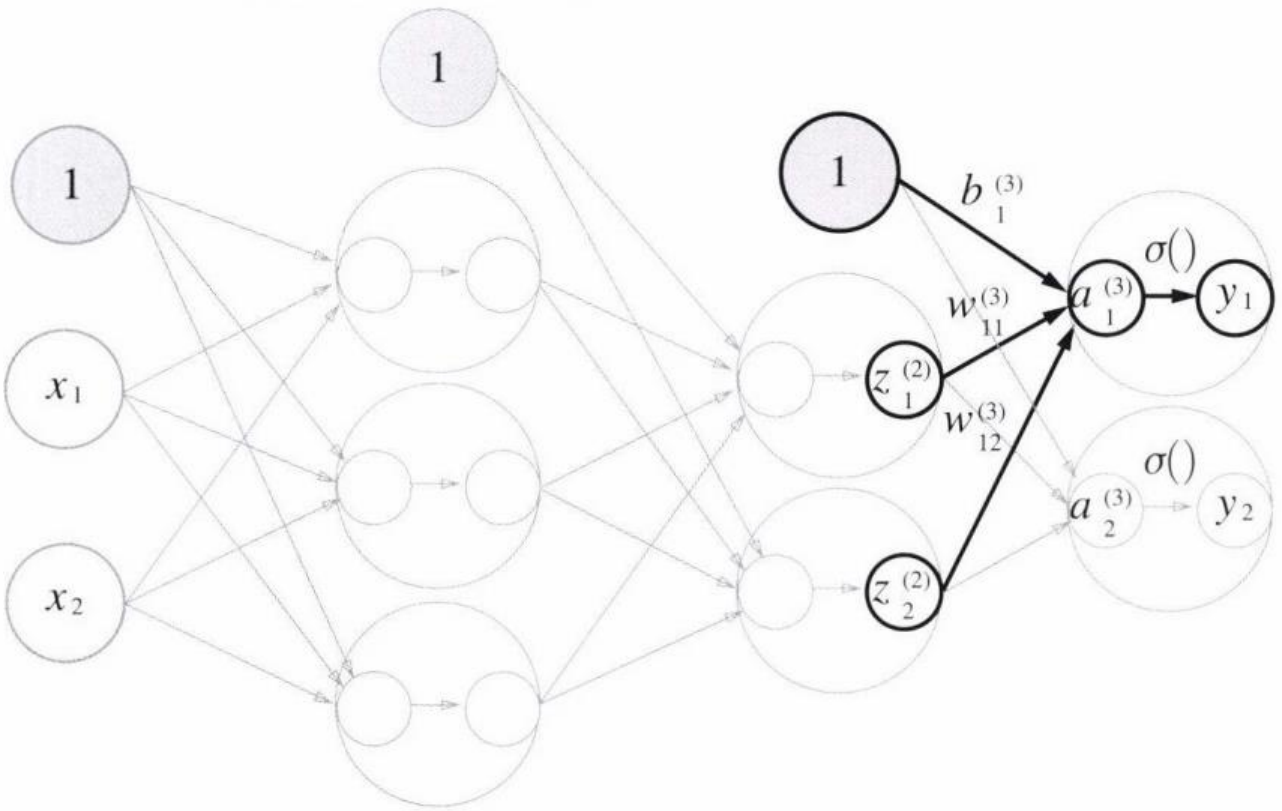
0층 -> 1층

3. 3층 신경망 구현하기

Forward propagation

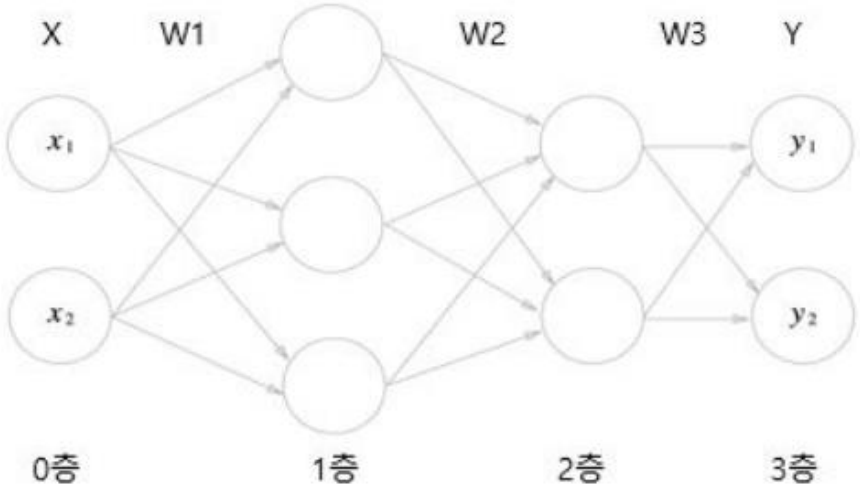


1층 -> 2층



2층 -> 3층

3. 3층 신경망 구현하기



[파라미터]

$$X = [1 \;\; 0.5]$$

$$W1 = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \quad b1 = [0.1 \;\; 0.2 \;\; 0.3]$$

$$W2 = \begin{bmatrix} 0.1 & 0.4 \\ 0.2 & 0.5 \\ 0.3 & 0.6 \end{bmatrix} \quad b2 = [0.1 \;\; 0.2]$$

$$W3 = \begin{bmatrix} 0.1 & 0.3 \\ 0.2 & 0.4 \end{bmatrix} \quad b3 = [0.1 \;\; 0.2]$$

```
X=np.array([1,0.5])
W1=np.array([[0.1,0.3,0.5],[0.2,0.4,0.6]])
B1=np.array([0.1,0.2,0.3])
W2=np.array([[0.1,0.4],[0.2,0.5],[0.3,0.6]])
B2=np.array([0.1,0.2])
W3=np.array([[0.1,0.3],[0.2,0.4]])
B3=np.array([0.1,0.2])

A1=np.dot(X,W1)+B1
Z1=sigmoid(A1)
A2=np.dot(Z1,W2)+B2
Z2=sigmoid(A2)
A3=np.dot(Z2,W3)+B3
Y=identity_function(A3)
```

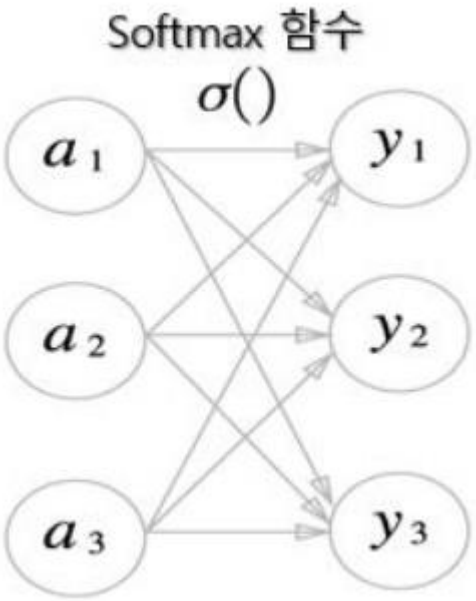
4. 출력층 설계하기

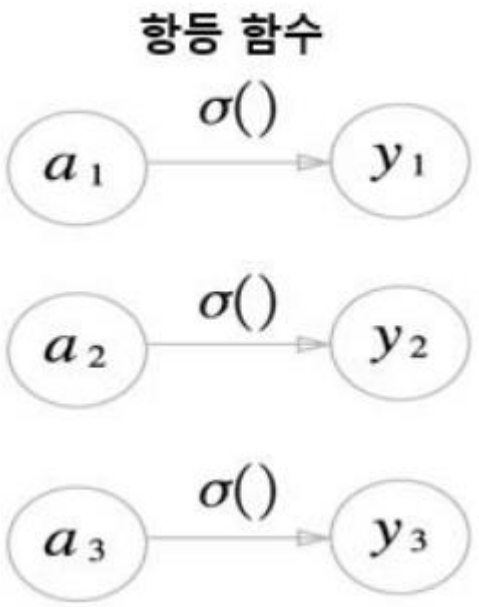
4. 출력층 설계하기

기계학습

- 분류 (스팸메일 분류) -> 소프트맥스 함수

- 회귀 (집값 예측) -> 항등 함수





4. 출력층 설계하기

소프트맥스 함수

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

단점: 오버플로 문제 발생

-> 입력 신호 중 최댓값을 빼준다.

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} = \frac{C\exp(a_k)}{C\sum_{i=1}^{n} \exp(a_i)}$$

$$= \frac{\exp(a_k + \log C)}{\sum_{i=1}^{n} \exp(a_i + \log C)}$$

$$= \frac{\exp(a_k + C')}{\sum_{i=1}^{n} \exp(a_i + C')}$$

C′=-max(ai)

```python
def softmax(a):
    exp_a=np.exp(a)
    sum_exp_a=np.sum(exp_a)
    y=exp_a/sum_exp_a

    return y
a=np.array([1010,1000,990])
print(softmax(a)) #nan
```

→

```python
def softmax(a):
    c=np.max(a) #오버플로 대처
    exp_a=np.exp(a)
    sum_exp_a=np.sum(exp_a)
    y=exp_a/sum_exp_a

    return y

a=np.array([1010,1000,990])
print(softmax(a)) #[9.99954600e-01 4.53978686e-05 2.06106005e-09]
```

4. 출력층 설계하기

장점: 결과를 확률로써 해석 가능

```
a=np.array([0.3,2.9,4.0])
y=softmax(a)
print(y) #[0.01821127 0.24519181 0.73659691]
np.sum(y) #1
```

5. 손글씨 숫자 인식

5. 손글씨 숫자 인식

MNIST 데이터셋

normalize = True
→ 0~1 사이 값
e.x) mean(x1) = 1000,
mean(x2) = 3
→ normalize
→ mean(x1)=0.5, mean(x2)=0.3

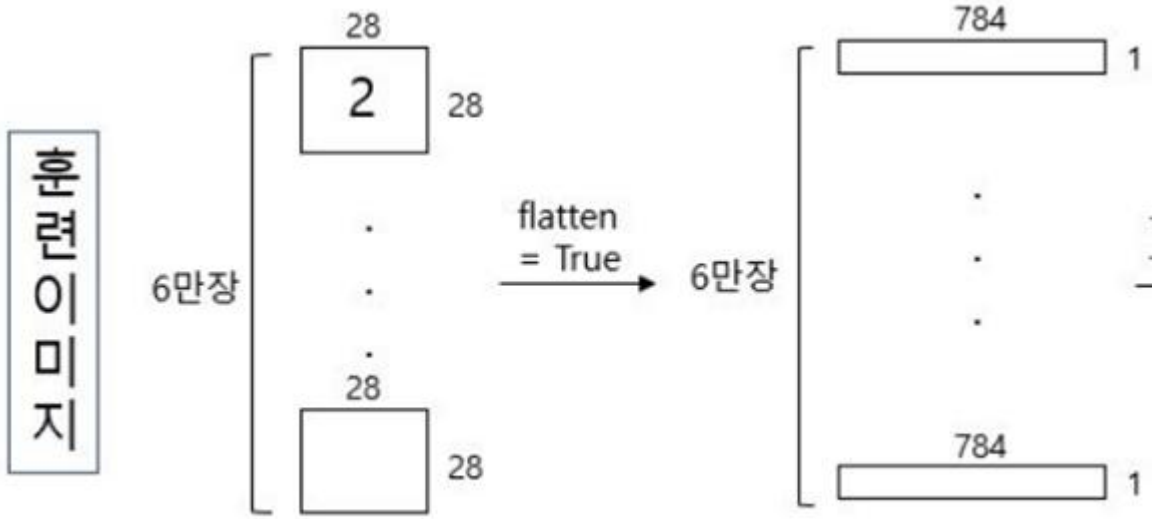
```
import numpy as np
import sys, os
sys.path.append(os.pardir)
from dataset.mnist import load_mnist

(x_train, t_train), (x_test, t_test)
        =load_mnist(normalize=True, flatten=True, one_hot_label=False)
```

flatten = True
1*28*28 → 784,

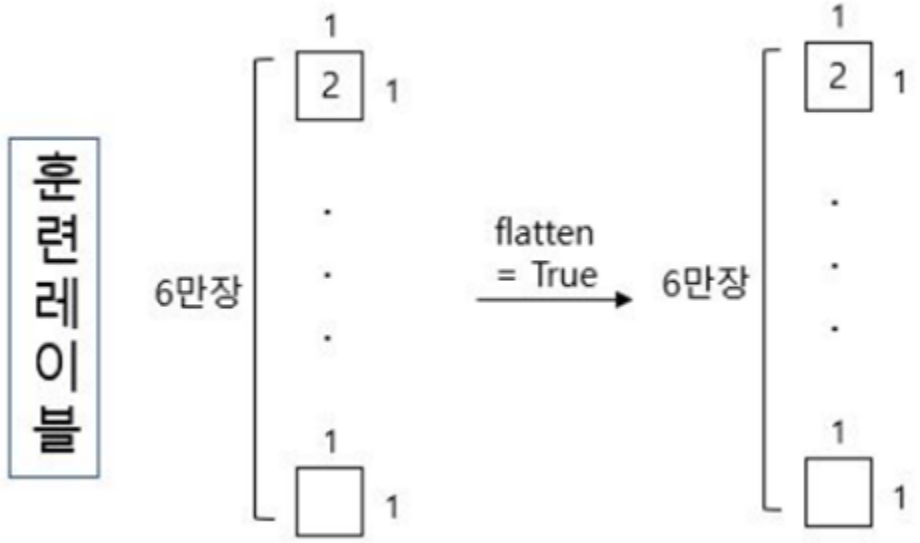
one_hot_label = True
3 → [0 0 0 1 0 0 0 0 0 0]

5. 손글씨 숫자 인식

x_train



t_train




```
print(x_train.shape) #(60000, 784)
print(t_train.shape) #(60000,)
print(x_test.shape)  #(10000, 784)
print(t_test.shape)  #(10000,)
```

5. 손글씨 숫자 인식 – 신경망 구현

```
def get_data():
    (x_train, t_train), (x_test, t_test)
    = load_mnist(normalize=True, flatten=True, one_hot_label=False)
    return x_test, t_test

def init_network():
    with open("sample_weight.pkl", 'rb') as f:
        network = pickle.load(f)
    return network

def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)

    return y
```

5. 손글씨 숫자 인식 – 신경망 실행 & 정확도 측정

```
x, t = get_data()
network = init_network()
accuracy_cnt = 0
for i in range(len(x)):  #len(x)=10000
    y = predict(network, x[i])
    p= np.argmax(y) # 확률이 가장 높은 원소의 인덱스를 얻는다.
    if p == t[i]:
        accuracy_cnt += 1

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

```
Accuracy:0.9352
```

```
print(x_test.shape)  #(10000, 784)
print(t_test.shape)  #(10000,)
```

```
predict(network,x[1])
```

```
array([4.83633112e-03, 1.10458629e-03, 9.44252372e-01, 1.43091455e-02,
       5.69895633e-07, 6.67604618e-03, 2.75333561e-02, 1.27084354e-06,
       1.28642377e-03, 4.78646243e-08], dtype=float32)
```

5. 손글씨 숫자 인식 – 배치 처리

```
x, t = get_data()
network = init_network()

batch_size = 100 # 배치 크기
accuracy_cnt = 0

for i in range(0, len(x), batch_size):
    x_batch = x[i:i+batch_size]
    y_batch = predict(network, x_batch)
    p = np.argmax(y_batch, axis=1)
    accuracy_cnt += np.sum(p == t[i:i+batch_size])

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

```
Accuracy:0.9352
```

배치(batch)
: 입력 데이터를 하나로 묶어서 한 번에 처리 하는 것

x_batch: 100 * 784
y_batch: 100 * 10