딥러닝 스터디

밑바닥 부터 시작하는 딥러닝

김제우

딥러닝스터디

목차

자기소개

- 각자 자기소개
- 파이썬은 얼마나 공부했는지, 딥러닝은 얼마나 공부했는지, 목표
- 딥러닝 공부에 투자할 예정인 시간, 기간
- 취미, MBTI, 등..

파이썬

사용 에디터 : VScode
파이썬 버전 : python 3.6^
스터디 전제 : 조건, 반복, 자료형 등의 파이썬 기본 문법은 안다는 가정!
(잘 모르시면 단톡에 질문 주시거나 따로 질문 주세요~)

numpy

```
>>> x = np.array([1.0,2.0,3.0])
>>> y = np.array([2.0,4.0,6.0])
>>> x + y
array( [3., 6., 9.])
>>> x - y
array( [3., 6., 9.])
>>> x * y
array( [3., 6., 9.])
>>> x / y
array( [3., 6., 9.])
```

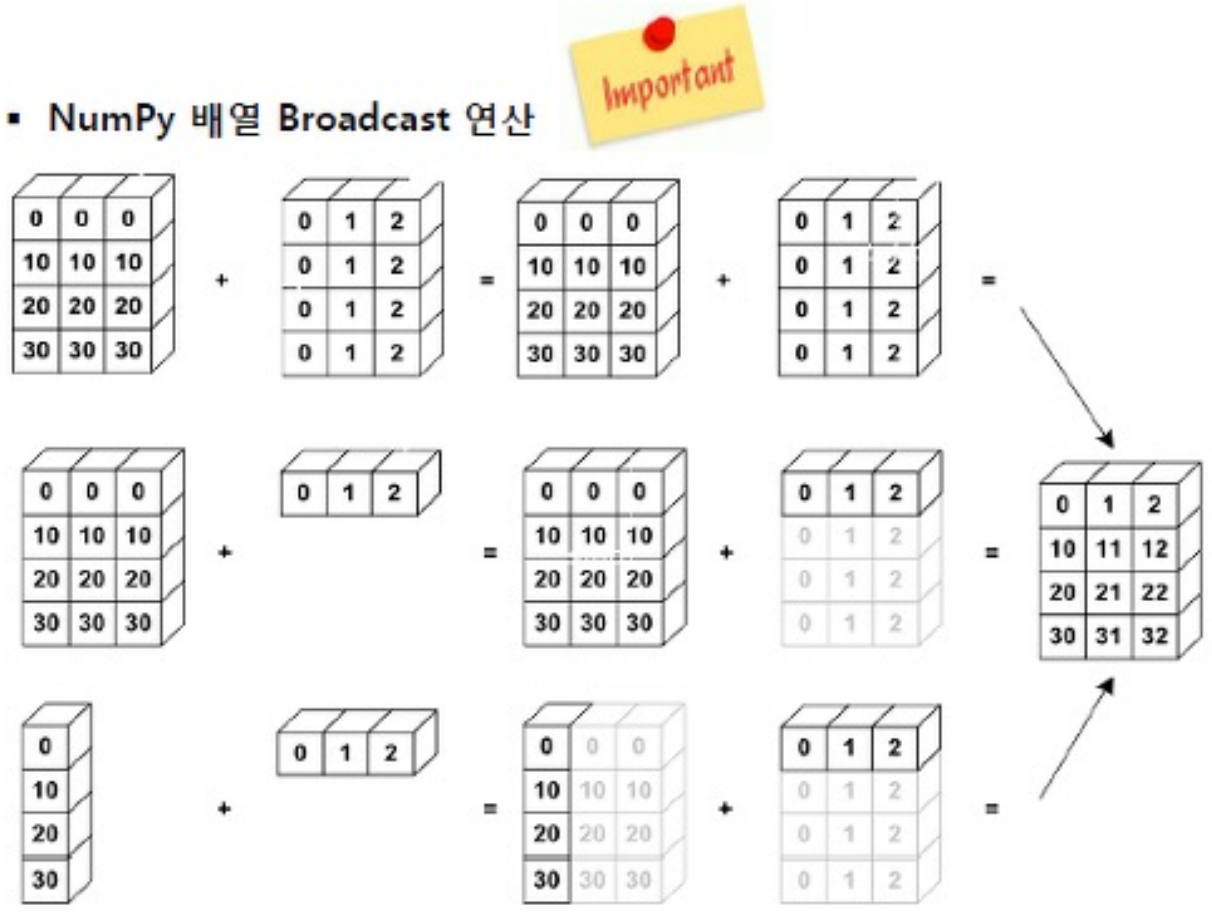
numpy

```
>>> x = np.array([1.0,2.0,3.0])
>>> x / 2.0
array( [0.5, 1., 1.5])

>>> A = np.array([[1,2],[3,4]])
>>> A.shape
(2,2)
>>> B = np.array([[3, 0],[0, 6]])
>>> A + B
array([[4,   2],
          [3, 10]])
```

Numpy 브로드 캐스트

```
>>> A = np.array([[1,2],[3,4]])
>>> B = np.array([10,20])
>>> A * B
array([[10,  40],
       [30,  80]])
```

Numpy 브로드 캐스트



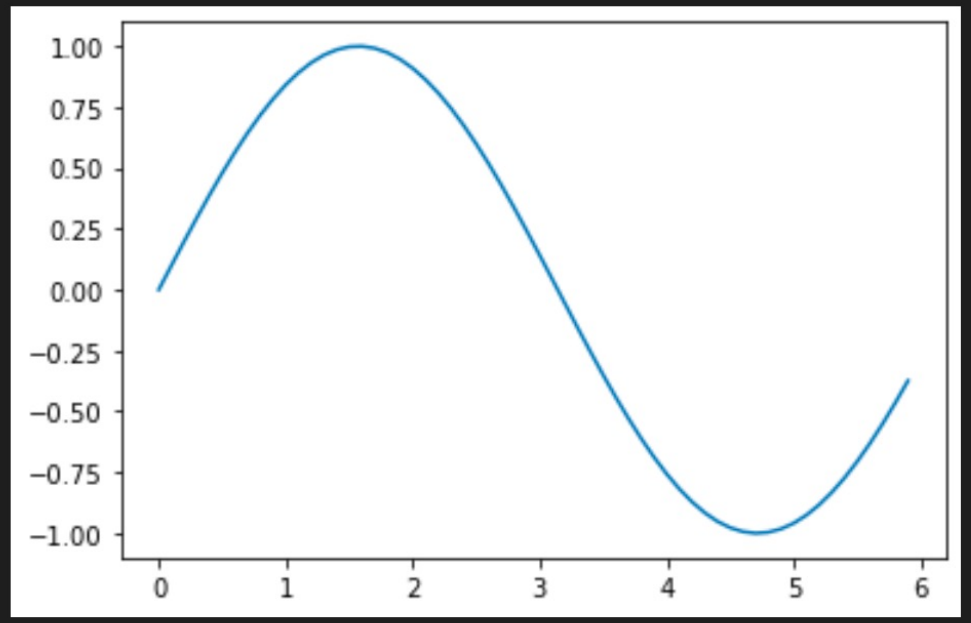
Numpy 원소접근

```
>>> A = np.array([[1,2],[3,4]])
>>> A[0]
array([1,   2])
>>> A[0][1]
2
>>> A > 2
[[False False]
 [ True  True]]
>>> A[1][A[1]>3]
4
```

matplotlib

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 6, 0.1)
y = np.sin(x)

plt.plot(x,y)
plt.show()
```

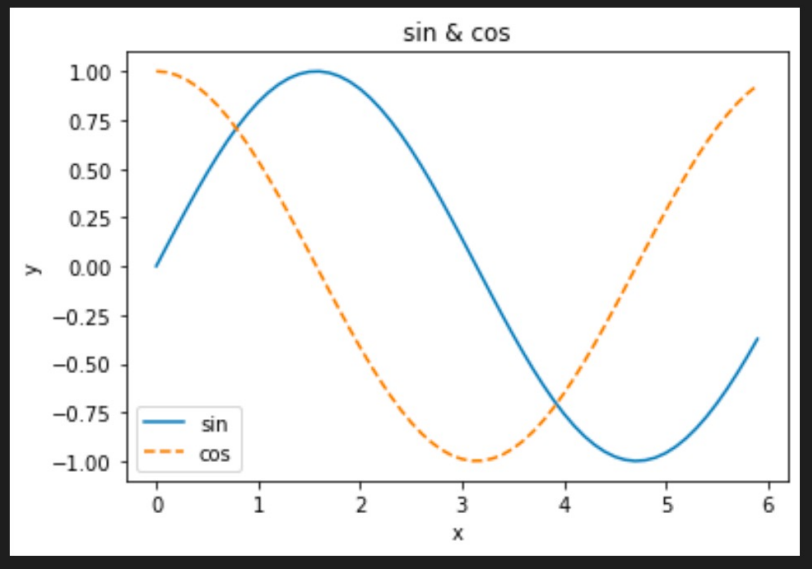
[2] ✓ 0.1s



matplotlib
pyplot기능들

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 6, 0.1)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, label='sin')
plt.plot(x, y2, linestyle='--', label='cos') # cos 함수는 점선으로 그림
plt.xlabel('x') # x축 이름
plt.ylabel('y') # y축 이름
plt.title('sin & cos') # 제목
plt.legend()
plt.show()
```

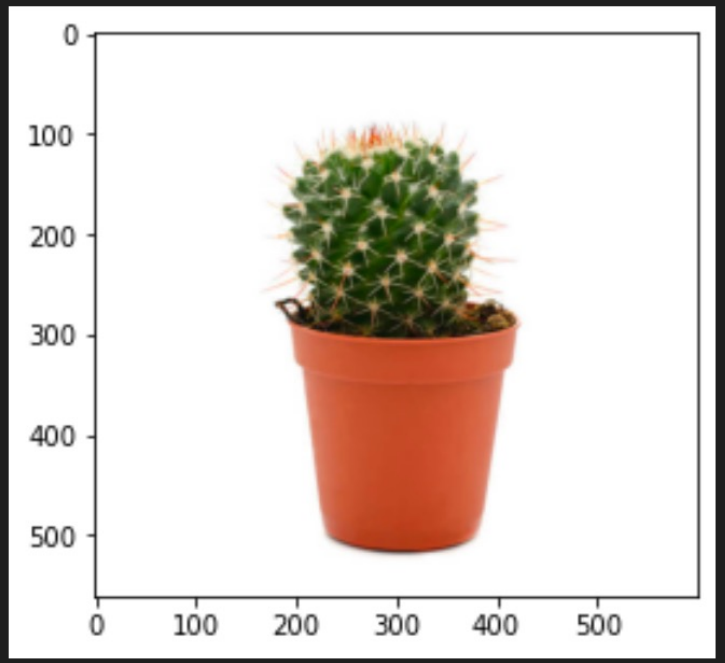
[3] ✓ 0.1s



matplotlib
pyplot기능들

```
import matplotlib.pyplot as plt
from matplotlib.image import imread

img = imread('cactus.jpeg')

plt.imshow(img)
plt.show()
```

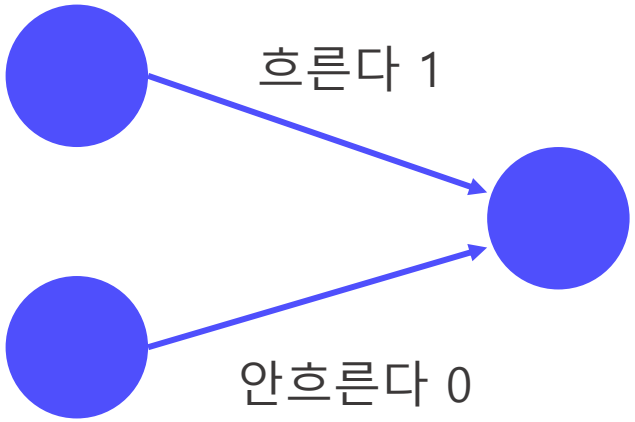
[4] ✓ 0.1s



퍼셉트론

퍼셉트론(perceptron)이란?

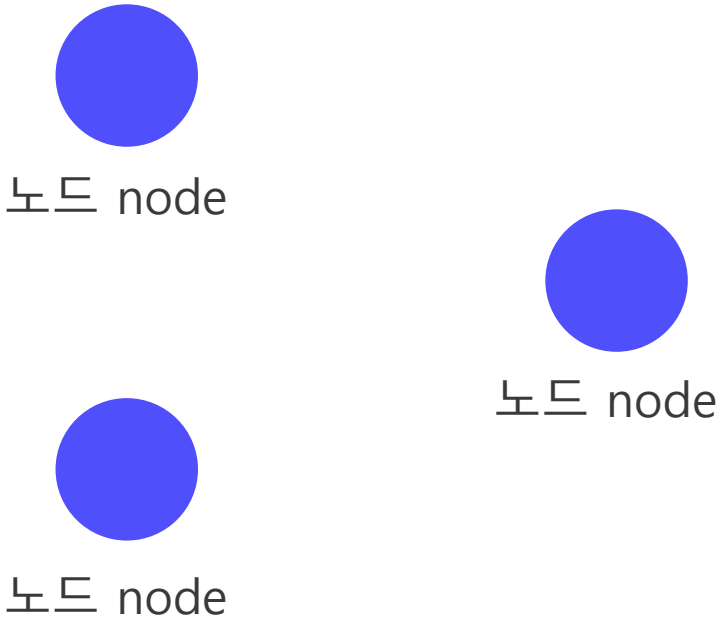
- 프랑크 로젠블라트가 1957년에 고안한 알고리즘!
- 신경망의 기원이 되는 알고리즘
- 다수의 신호를 입력으로 받아 하나의 신호를 출력함.
- 퍼셉트론 신호는 흐른다 / 안흐른다 두가지 상태만 존재 .
- 흐른다 = 1 안흐른다 = 0



입력으로 2개의 신호를 받은 퍼셉트론

퍼셉트론

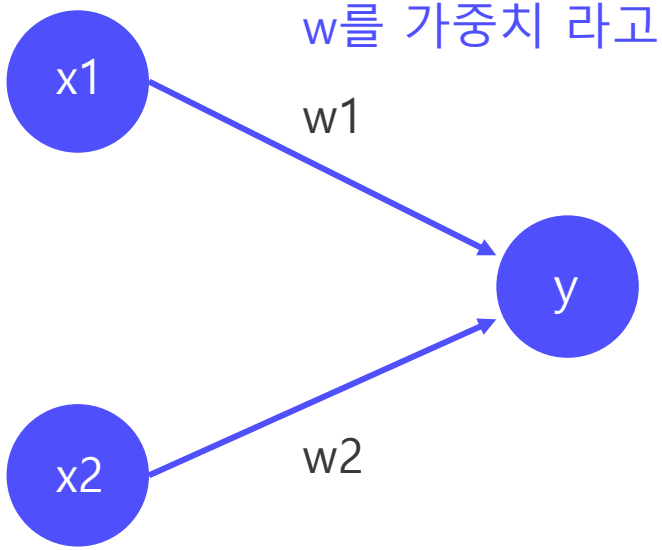
퍼셉트론(perceptron)이란?



노드 node 혹은 뉴런

퍼셉트론

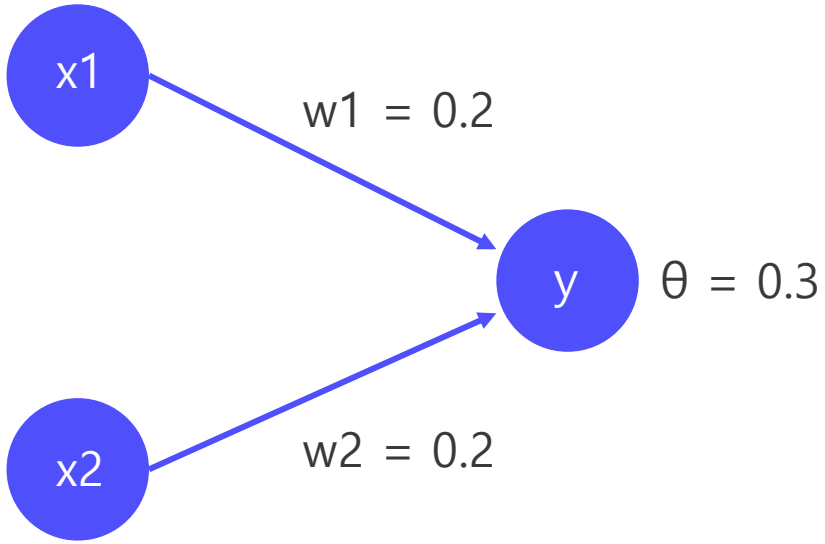
퍼셉트론(perceptron)이란?



$$y \begin{cases} 0 (w1x1 + w2x2) <= \theta \\ 1 (w1x1 + w2x2) > \theta \end{cases}$$

퍼셉트론

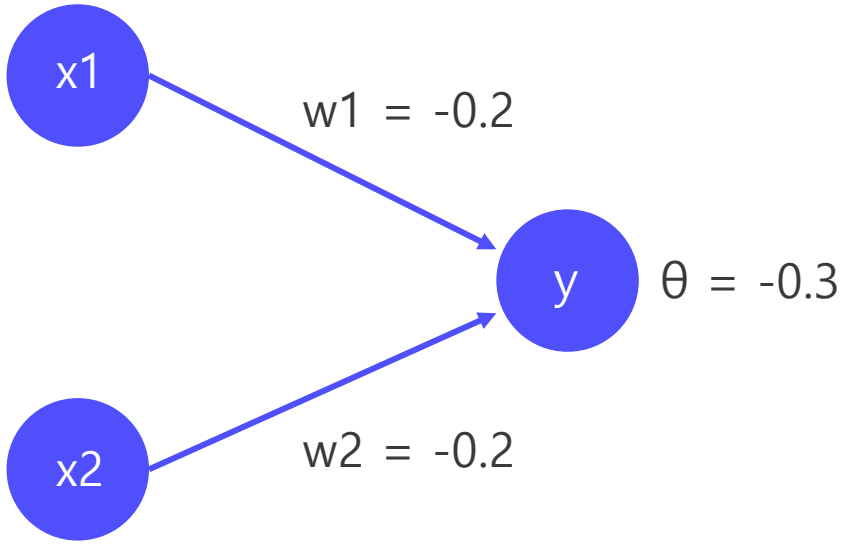
단순한 논리 회로



AND 게이트

x1	x2	y
0	0	0
1	0	0
0	1	0
1	1	1

단순한 논리 회로

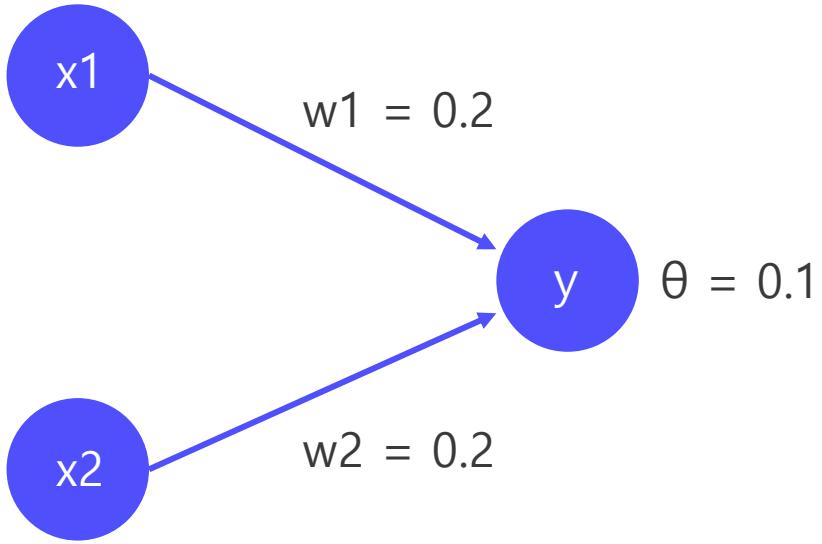


NAND 게이트

x1	x2	y
0	0	1
1	0	1
0	1	1
1	1	0

퍼셉트론

단순한 논리 회로



OR 게이트

x1	x2	y
0	0	0
1	0	1
0	1	1
1	1	1

퍼셉트론

퍼셉트론
AND게이트 구현

```
def AND(x1, x2):
    w1, w2, theta = 0.2, 0.2, 0.3
    tmp = x1*w1 + x2*w2
    if tmp <= theta:
        return 0
    elif tmp > theta:
        return 1
```

[34] ✓ 0.1s

```
print(AND(0,0))
print(AND(1,0))
print(AND(0,1))
print(AND(1,1))
```

[35] ✓ 0.2s

```
0
0
0
1
```

퍼셉트론

퍼셉트론
NAND게이트 구현

```
def NAND(x1, x2):
    w1, w2, theta = -0.2, -0.2, -0.3
    tmp = x1*w1 + x2*w2
    if tmp <= theta:
        return 0
    elif tmp > theta:
        return 1
```

[36] ✓ 0.1s

```
print(NAND(0,0))
print(NAND(1,0))
print(NAND(0,1))
print(NAND(1,1))
```

[37] ✓ 0.2s

```
1
1
1
0
```

퍼셉트론

퍼셉트론
OR게이트 구현

```
def OR(x1, x2):
    w1, w2, theta = 0.2, 0.2, 0.1
    tmp = x1*w1 + x2*w2
    if tmp <= theta:
        return 0
    elif tmp > theta:
        return 1
```

[38] ✓ 0.2s

```
print(OR(0,0))
print(OR(1,0))
print(OR(0,1))
print(OR(1,1))
```

[39] ✓ 0.1s

```
0
1
1
1
```

퍼셉트론

bias

$$y \begin{cases} 0 (w1x1 + w2x2) <= \theta \\ 1 (w1x1 + w2x2) > \theta \end{cases} \Rightarrow y \begin{cases} 0 (w1x1 + w2x2) - \theta <= 0 \\ 1 (w1x1 + w2x2) - \theta > 0 \end{cases}$$

bias

$$y \begin{cases} 0 (w1x1 + w2x2) <= \theta \\ 1 (w1x1 + w2x2) > \theta \end{cases}$$

$$-\theta = b$$

$$y \begin{cases} 0 (w1x1 + w2x2 + b) <= 0 \\ -\theta = b \\ 1 (w1x1 + w2x2 + b) > 0 \end{cases}$$

b를 bias (편향) 이라고 부름

퍼셉트론

bias를 포함한
AND 게이트 구현

θ값 대신 bias를 설정!

```
def AND(x1, x2):
    x = np.array([x1,x2])
    w = np.array([0.2,0.2])
    b = -0.3
    tmp = np.sum(w*x) + b  # np.sum 으로 행렬 덧셈 연산
    if tmp <= 0:
        return 0
    elif tmp > 0:
        return 1
```

[47] ✓ 0.2s

```
print(AND(0,0))
print(AND(1,0))
print(AND(0,1))
print(AND(1,1))
```

[48] ✓ 0.2s

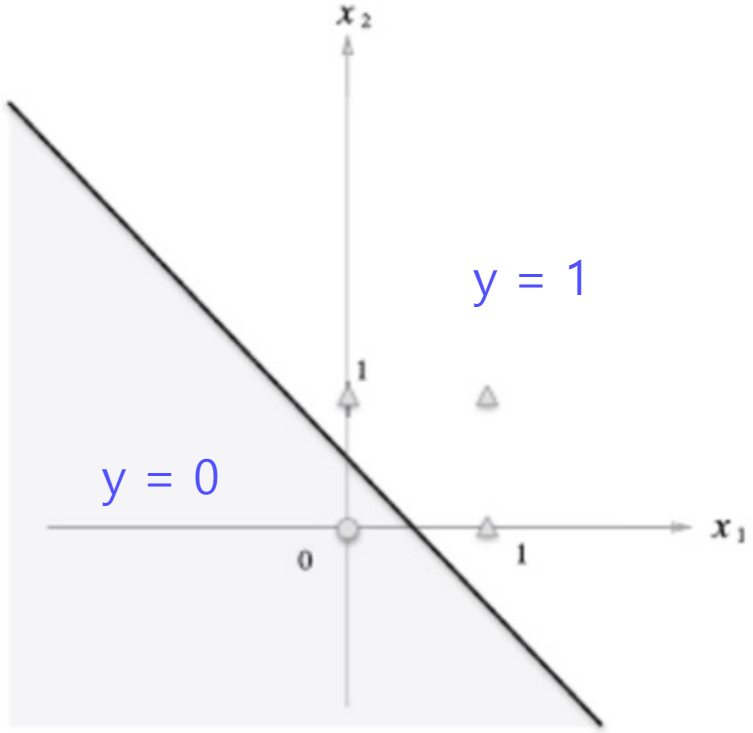
```
0
0
0
1
```

퍼셉트론

bias

- **bias는 뉴런이 얼마나 쉽게 활성화되는지를 결정**합니다.
- 이 책에서는 문맥에 따라 bias를 가중치라고 부를때도 있습니다.

퍼셉트론의 한계



OR 게이트의 시각화

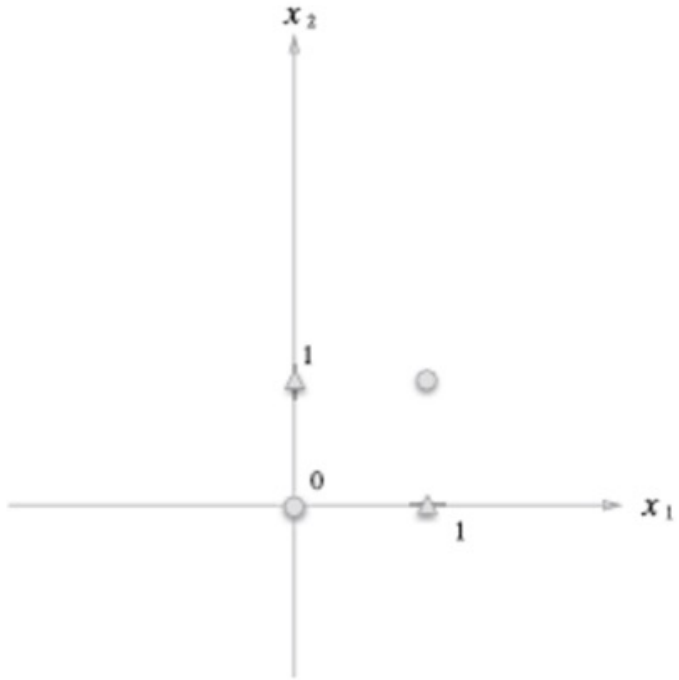
w1 = 1 , w2 =1, b = -0.5

$$y = \begin{cases} 0 & (x1 + x2 - 0.5) <= 0 \\ 1 & (x1 + x2 - 0.5) > 0 \end{cases}$$

x1 = 0	x2 =0	y = 0
x1 = 1	x2 =0	y = 1
x1 = 0	x2 =1	y = 1
x1 = 1	x2 =1	y = 1

퍼셉트론

퍼셉트론의 한계



XOR 게이트의 시각화

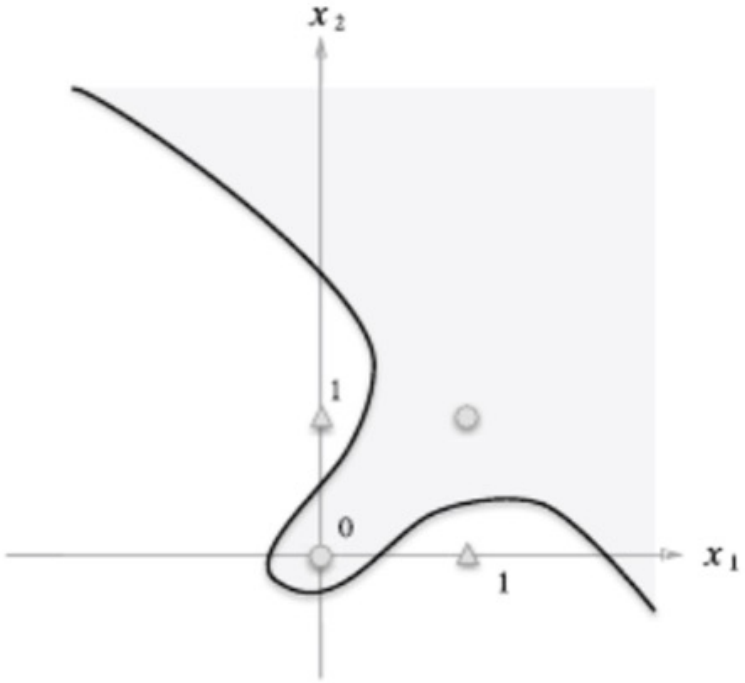
w1 = ? , w2 =?, b = ?

$$y \begin{cases} 0 & (w1x1 + w2x2 + b) <= 0 \\ 1 & (w1x1 + w2x2 + b) > 0 \end{cases}$$

x1 = 0	x2 =0	y = 0
x1 = 1	x2 =0	y = 1
x1 = 0	x2 =1	y = 1
x1 = 1	x2 =1	y = 0

퍼셉트론

퍼셉트론의 한계



XOR 게이트의 시각화

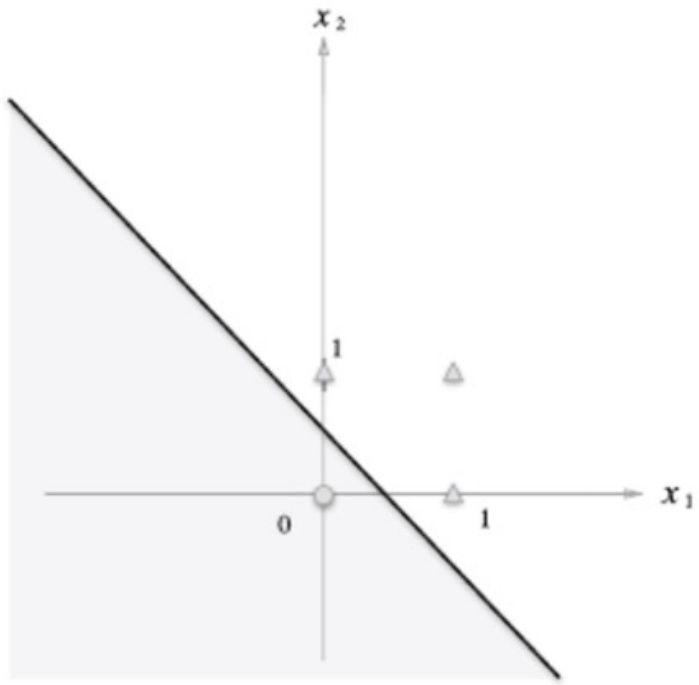
w1 = ? , w2 =?, b = ?

$$y \begin{cases} 0 & (w1x1 + w2x2 + b) <= 0 \\ 1 & (w1x1 + w2x2 + b) > 0 \end{cases}$$

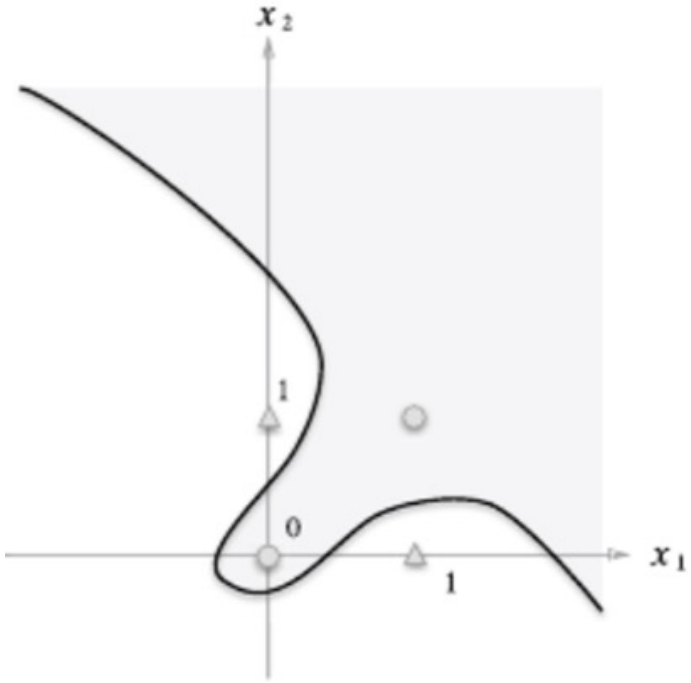
x1 = 0	x2 =0	y = 0
x1 = 1	x2 =0	y = 1
x1 = 0	x2 =1	y = 1
x1 = 1	x2 =1	y = 0

퍼셉트론

퍼셉트론의 한계



선형 = 직선 그래프

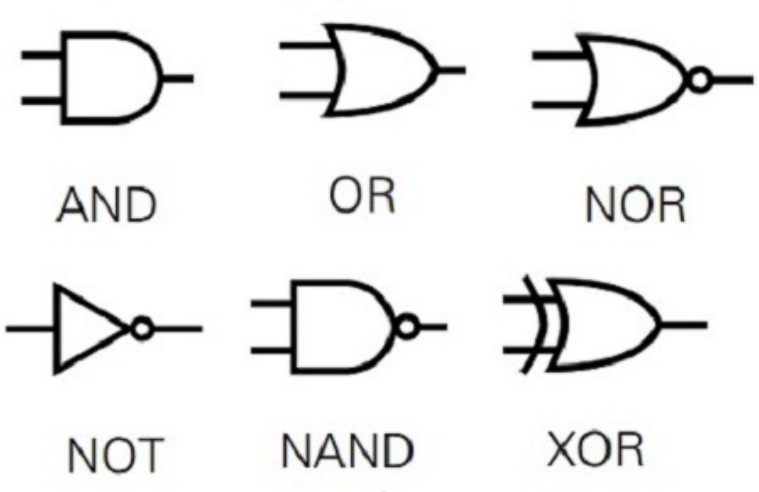


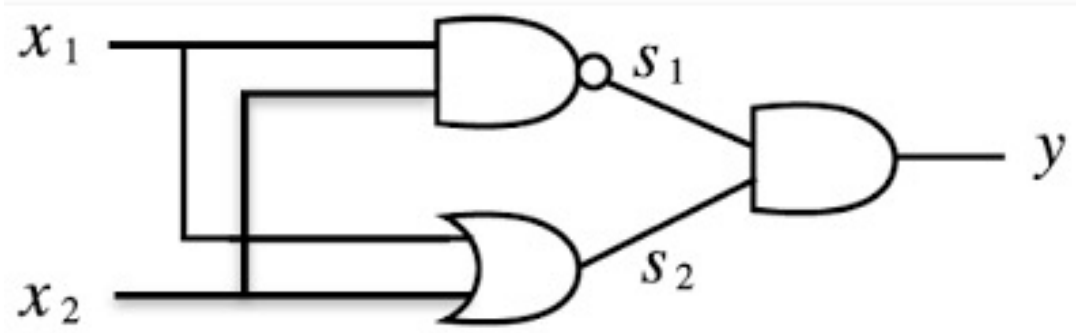
비선형 = 곡선(직선x) 그래프

퍼셉트론의 한계와 다층 퍼셉트론

- 퍼셉트론으로 XOR 게이트는 표현할 수 없다.
- 하지만 **여러개의 퍼셉트론을 이용하면 XOR게이트도 표현 가능!**
- 이걸 **다층 퍼셉트론**이라고 부름

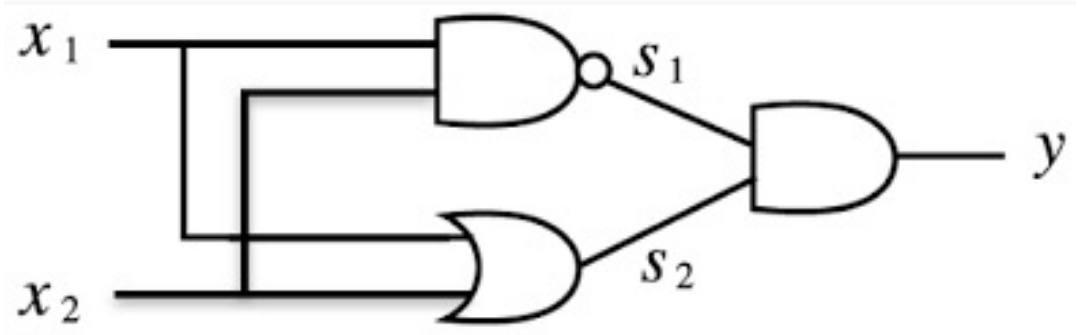
다층 퍼셉트론으로 XOR 게이트 만들기





퍼셉트론

다층 퍼셉트론으로 XOR 게이트 만들기



x1	x2	s1	s2	y
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	1	0

퍼셉트론

XOR 게이트 구현

```
def XOR(x1, x2):
    s1 = NAND(x1, x2)
    s2 = OR(x1, x2)
    y = AND(s1, s2)
    return y
```

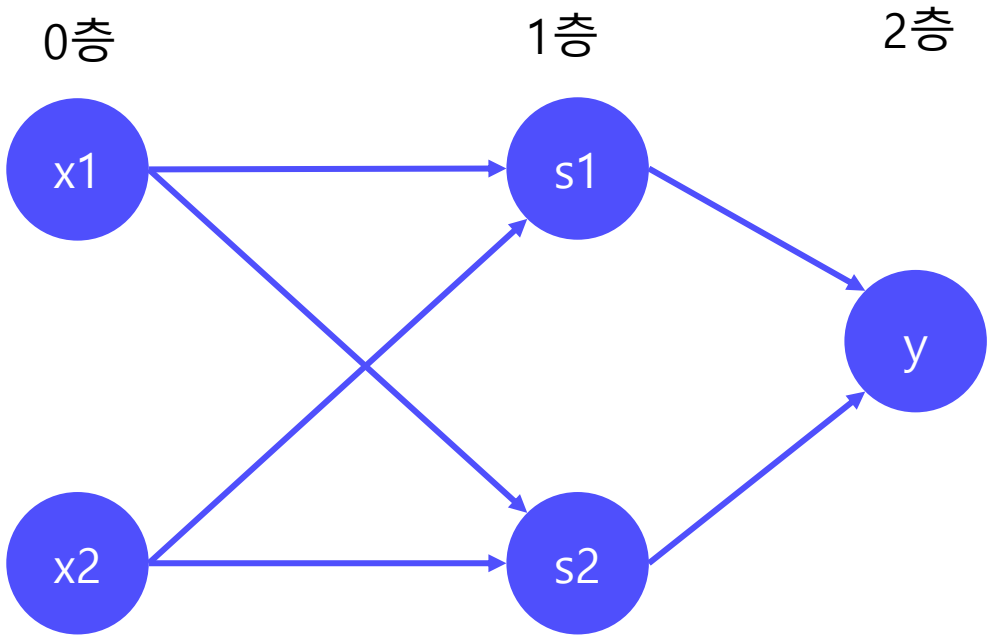
[52] ✓ 0.2s

```
print(XOR(0,0))
print(XOR(1,0))
print(XOR(0,1))
print(XOR(1,1))
```

[53] ✓ 0.2s

```
...  0
     1
     1
     0
```

XOR 게이트 구현



결론

- 퍼셉트론을 쌓으면 모든 논리 연산자를 표현할 수 있다.
- 모든 논리 연산자를 표현할 수 있다는 말은 모든 이진 연산이 가능하다.
- 컴퓨터를 만들수도 있다!
- 퍼셉트론은 입출력을 가진 알고리즘이다. 입력을 주면 정해진 규칙에 따른 값을 출력한다.
- 퍼셉트론에서는 '가중치'와 '편향'을 매개변수로 설정한다.
- 단층 퍼셉트론으로 AND, OR, NAND 를 표현할 수 있다.
- 단층 퍼셉트론으로 비선형을 표현할수는 없지만 다층 퍼셉트론으로는 가능하다.

신경망

- 앞에서 우리는 아래와 같이 w1, w2, b 와 같은 가중치 들을 직접 설정해줬다. -> 지금까지 하던 프로그래밍

- **퍼셉트론의 가중치들을 데이터로부터 자동으로 학습하는 능력**이 신경망의 중요한 성질이다.

- 위의 성질은 4장 신경망 학습에서 배운다.

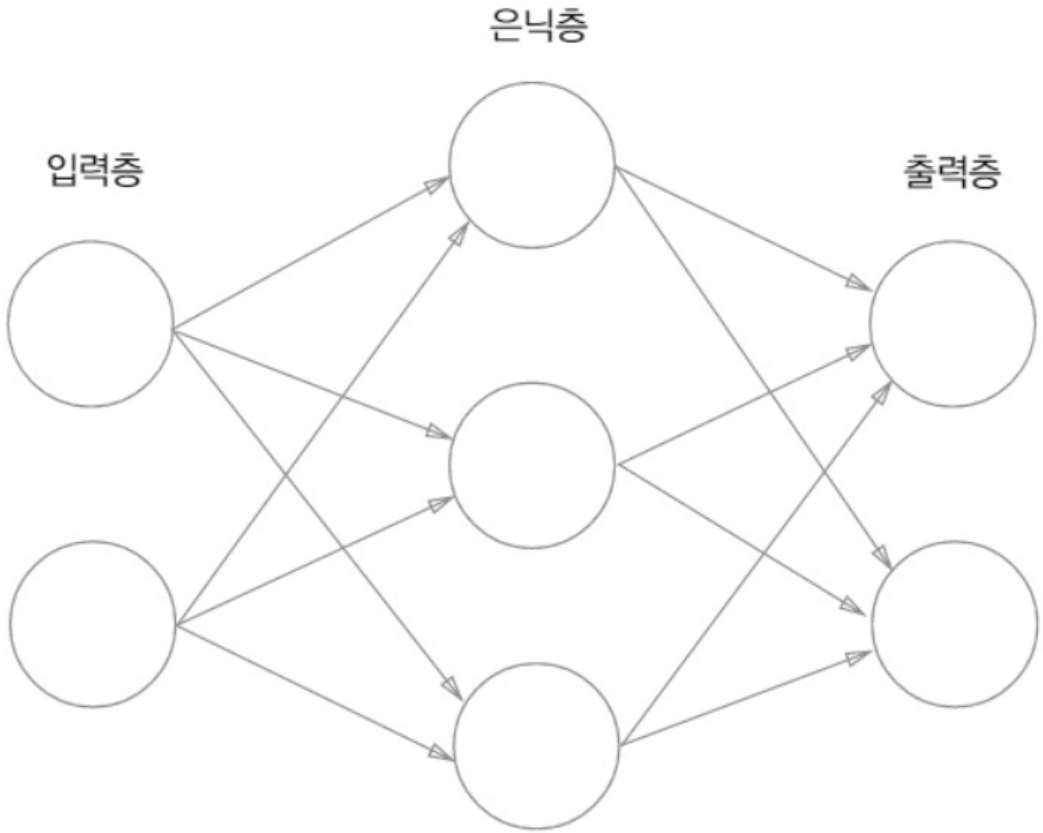
- 이번 장에서는 신경망의 입력 데이터가 무엇인지 식별하는 처리를 배운다.

$$w1 = 1,\ w2 = 1,\ b = -0.5$$

$$y \begin{cases} 0 & (x1 + x2 - 0.5) <= 0 \\ 1 & (x1 + x2 - 0.5) > 0 \end{cases}$$

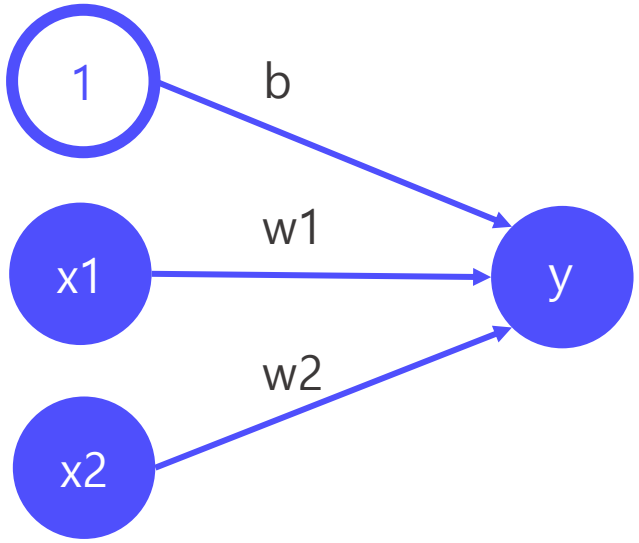
신경망

- 신경망의 예



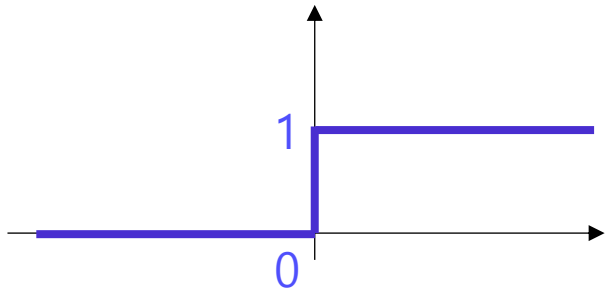
신경망

- 편향을 명시한 퍼셉트론과 활성화 함수(activation function)



$y = h(w1x1 + w2x2 + b)$

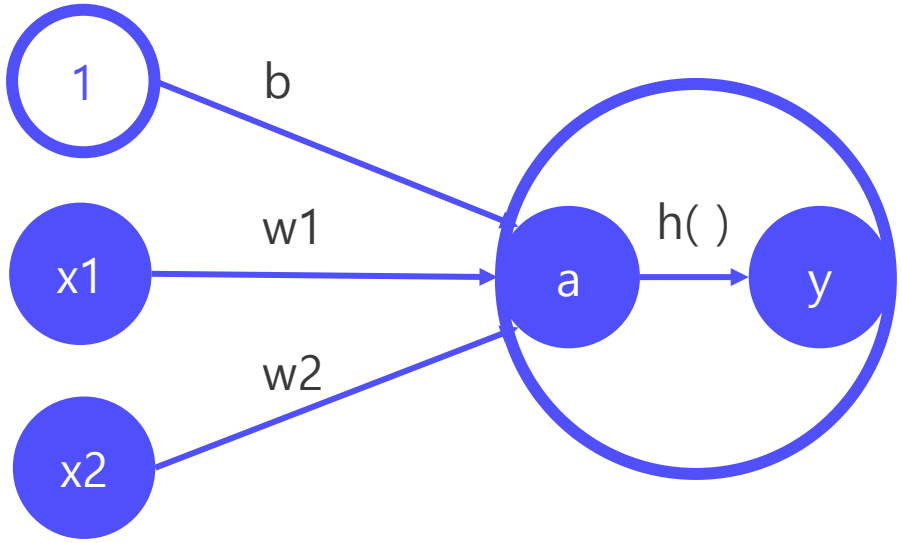
$$h(x) \begin{cases} 0 & (x <= 0) \\ 1 & (x > 0) \end{cases}$$



활성화 함수 : node 의 출력을 조절하는 함수

신경망

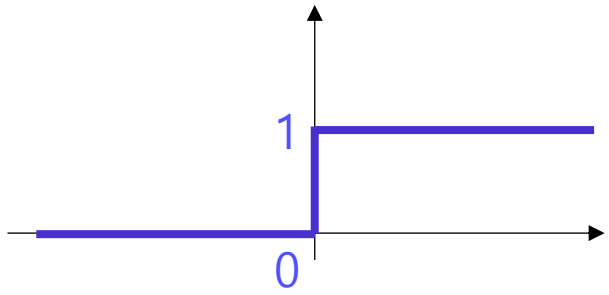
- 편향을 명시한 퍼셉트론과 활성화 함수(activation function)



$$a = w1x1 + w2x2 + b$$

$$y = h(a)$$

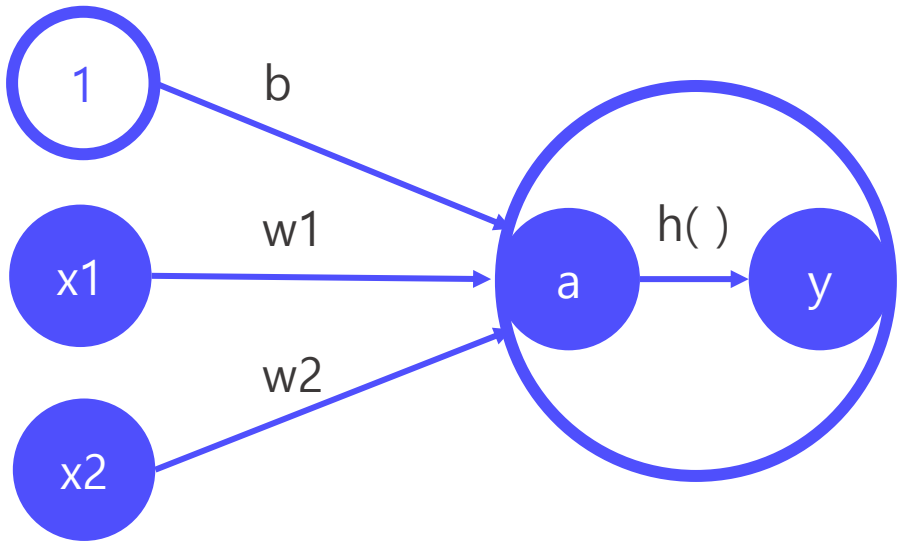
$$h(x) \begin{cases} 0 & (x <= 0) \\ 1 & (x > 0) \end{cases}$$



활성화 함수가 퍼셉트론에서 신경망으로 가기 위한 핵심!

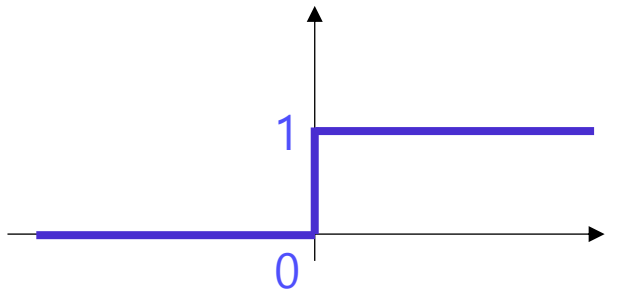
활성화 함수

계단 함수



y = h(a)

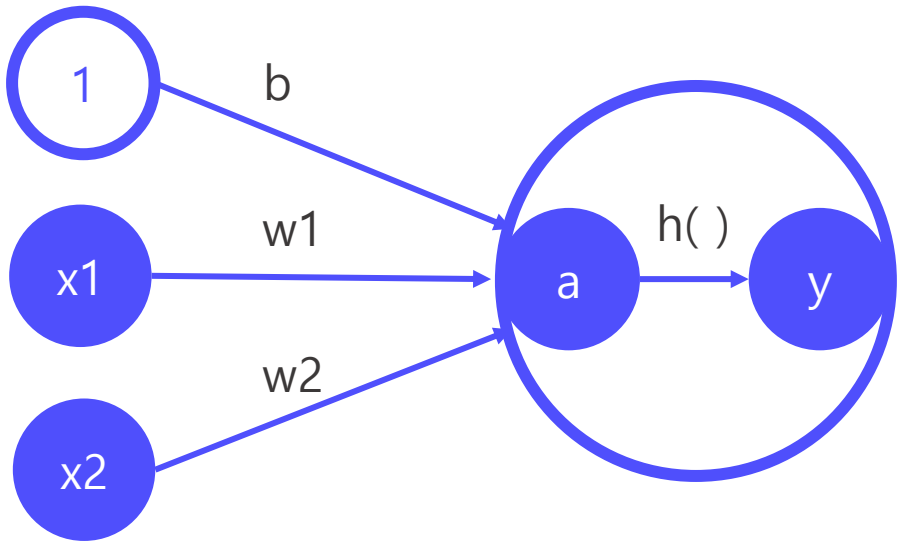
h(x) { 0 (x<= 0)
1 (x > 0)



계단 함수 : 0보다 클때 1을 출력하고 그외에는 0을 출력하는 함수

활성화 함수

시그모이드 함수 : 신경망에서 자주 이용하는 활성함수



$$\exp(-x) = e^{-x}$$

$$h(x) = \frac{1}{1 + \exp(-x)}$$

활성화 함수

계단함수 구현

```
[2]
import numpy as np
✓ 0.4s
```

```
[3]
def step_function(x):
    if x > 0 :
        return 1
    else :
        return 0
# x를 숫자 하나만 넣을때는 되는데 numpy 배열을 입력으로 받고 싶다.
✓ 0.2s
```

```
[4]
def step_function(x):
    y = x > 0
    return y.astype(np.int)
✓ 0.1s
```

활성화 함수

계단함수 구현

```
    x = np.array([-1.0,1.0,2.0])
    y = x > 0
    print(y)
[13]  ✓  0.3s
...  [False  True  True]

    y.astype(np.int) # bool -> int
[14]  ✓  0.2s
...  array([0, 1, 1])
```

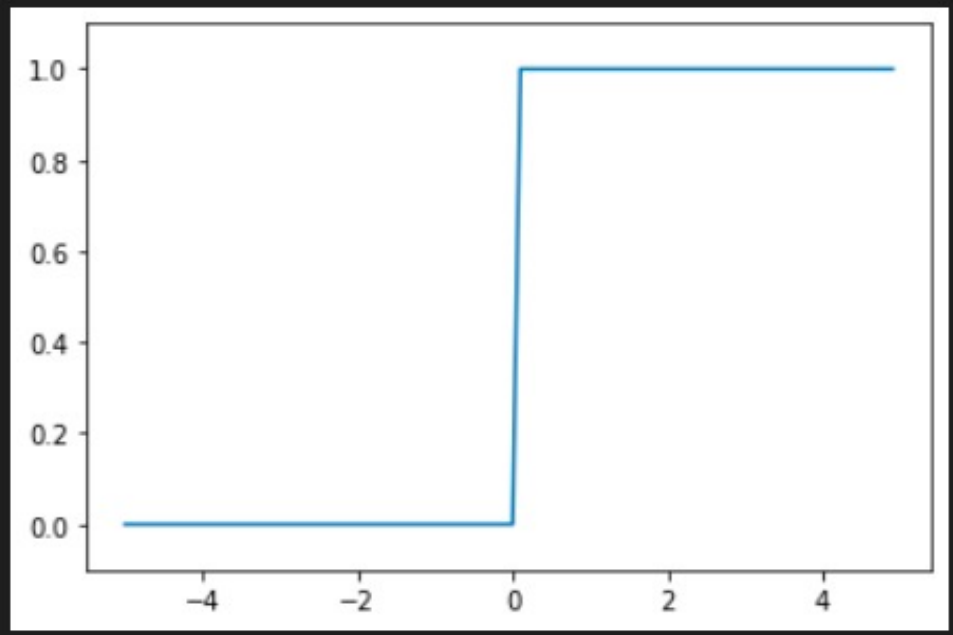
활성화 함수

계단함수 구현

```
x = np.arange(-5.0,5.0,0.1)
y = step_function(x)
plt.plot(x,y)
plt.ylim(-0.1,1.1)#y축의 범위 지정
plt.show()
```

[15] ✓ 0.1s

...



활성화 함수

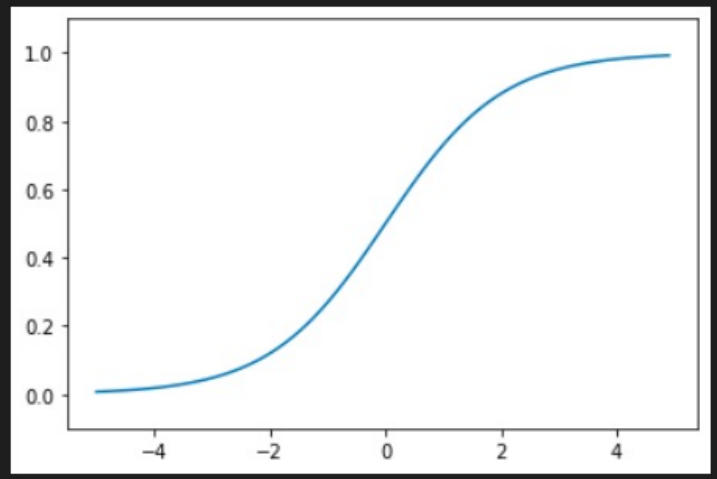
시그모이드함수 구현

```
def sigmoid(x):
    return 1/(1+np.exp(-x)) #브로드 캐스트
```

✓ 0.2s

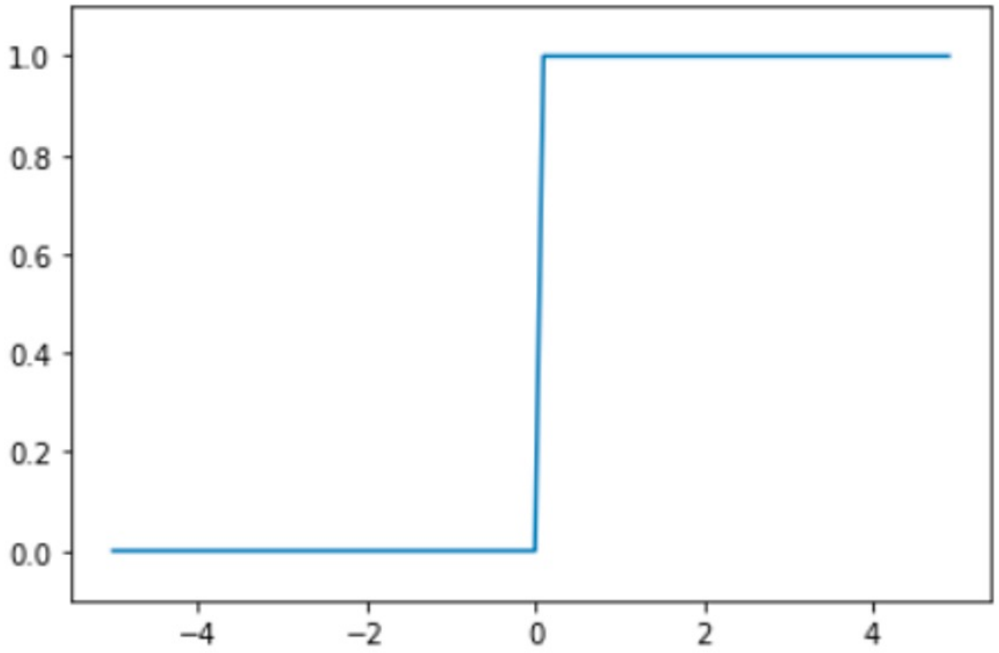
```
x = np.arange(-5.0,5.0,0.1)
y = sigmoid(x)
plt.plot(x,y)
plt.ylim(-0.1,1.1)
plt.show()
```

[20] ✓ 0.1s

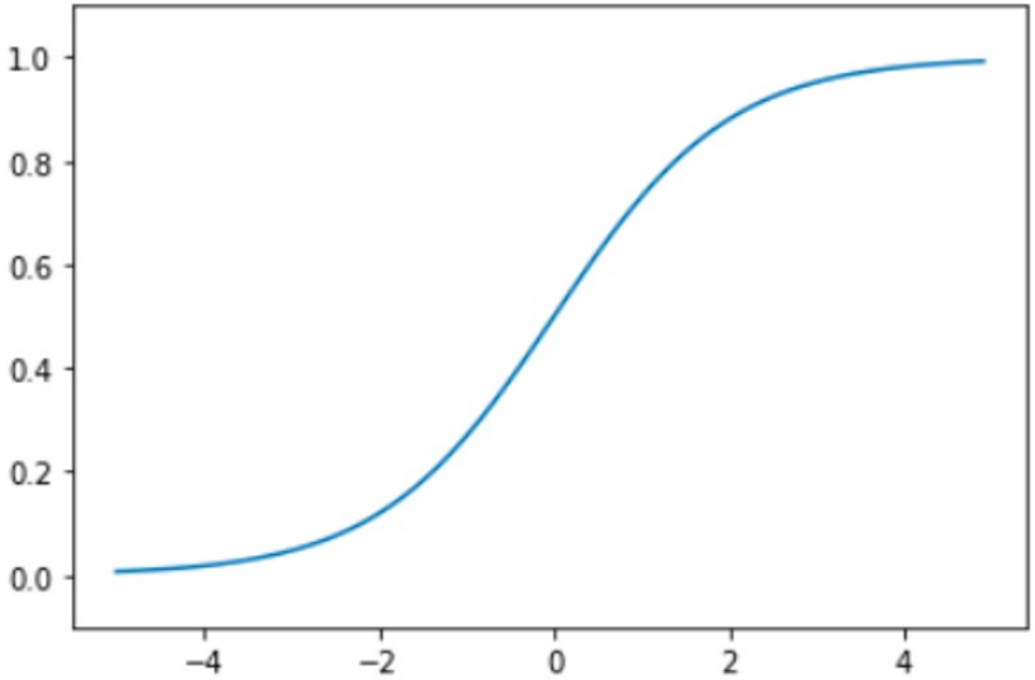


활성화 함수

계단 vs 시그모이드



계단 함수



시그모이드 함수

활성화 함수

계단 vs 시그모이드

	계단함수	시그모이드 함수
차이점	0 또는 1	0에서 1사이 실수
공통점	입력이 작을때는 0 입력이 커지면 1에 가까워짐	
	비선형	

비선형 함수

- 왜 비선형 함수를 써야할까?

  선형 함수를 이용하면 신경망의 층을 깊게 하는 의미가 없어지기 때문이다!

  예시 :

  h(x) = cx 일때 층을 깊게 쌓으면 c*(c*(c*x) = (c^3)*x -> 처음부터 y = a*x의 형태로 굳이 퍼셉트론을 사용하지 않아도 된다.

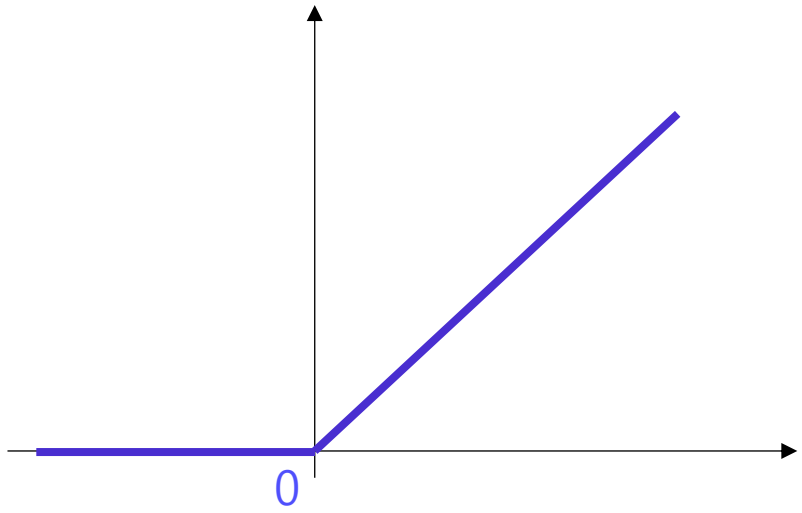
  다시 말해, 은닉층이 없는 네트워크로 구현이 가능하다.

- 층을 쌓는 혜택을 얻고 싶다면 활성화 함수로는 반드시 비선형 함수를 사용해야한다.

활성화 함수

ReLU 함수

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



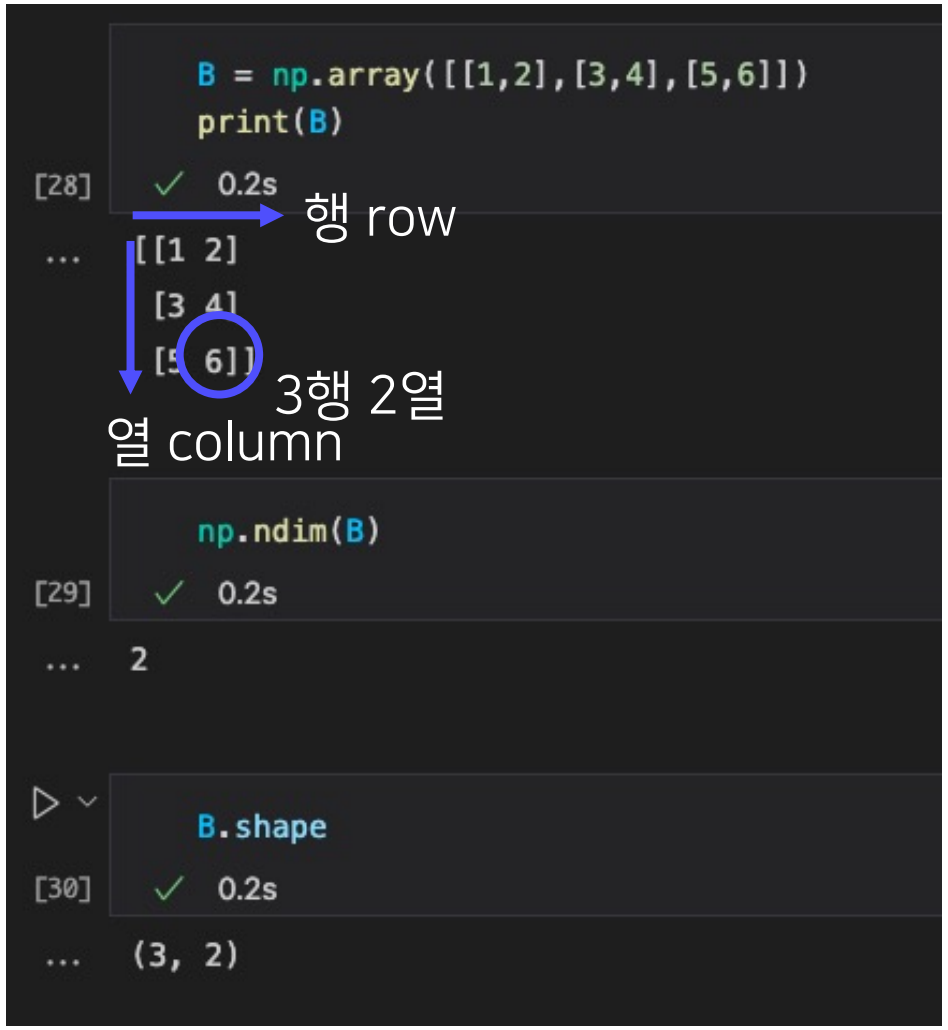
활성화 함수

ReLU 함수 구현

```
def relu(x):
    return np.maximum(0,x)
# x -> [1,2,-1,-2]
# 0 -> [0,0, 0, 0]
```

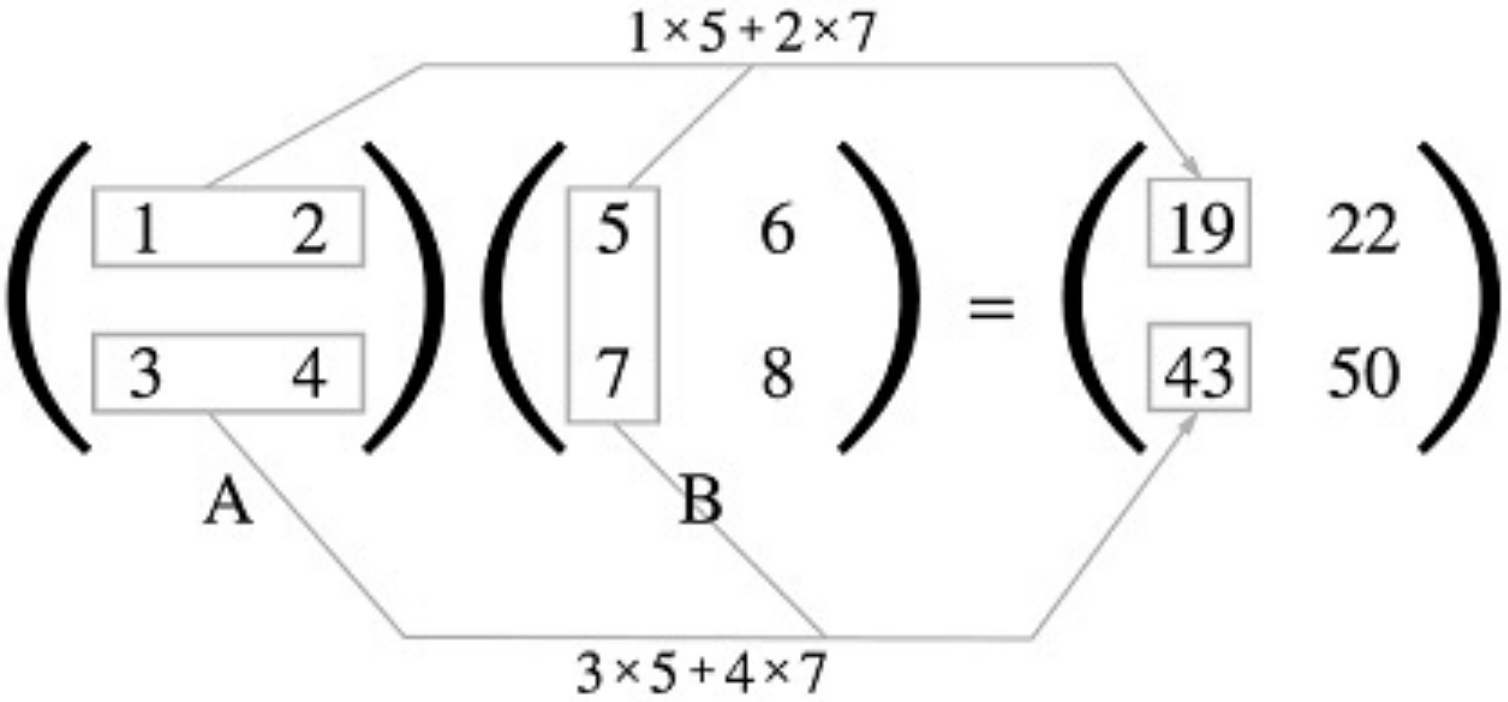
다차원 배열의 연산

다차원배열



다차원 배열의 연산

행렬의 곱



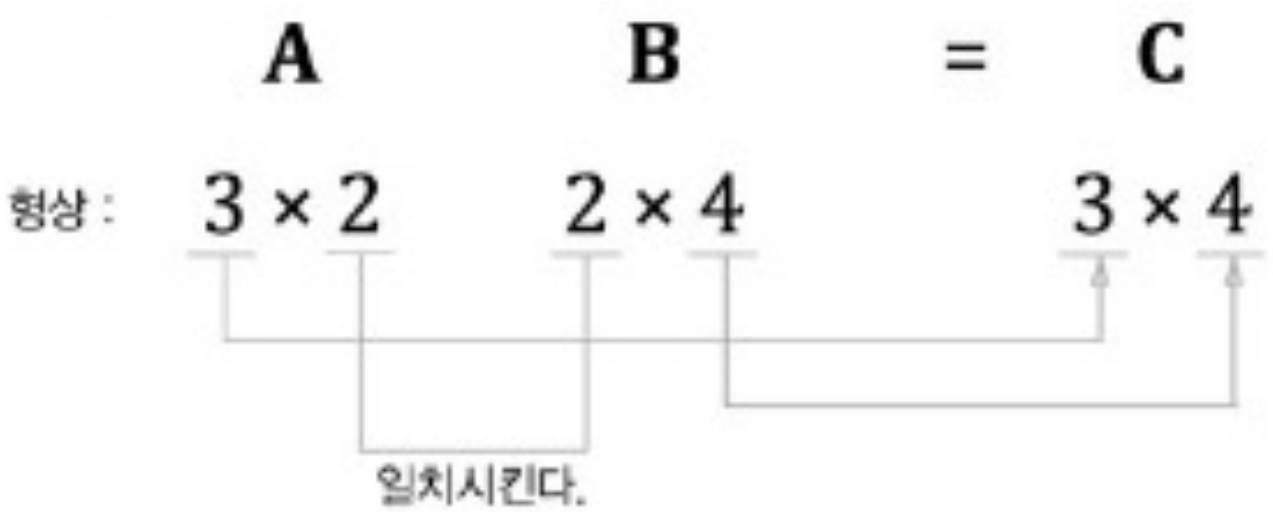
다차원 배열의 연산

행렬의 곱

```
[31]  A = np.array([[1,2],[3,4]])
      A.shape
      ✓ 0.1s
...   (2, 2)

      B = np.array([[5,6],[7,8]])
      B.shape
[32]  ✓ 0.2s
...   (2, 2)

      np.dot(A,B) #행렬의 곱 연산
[33]  ✓ 0.2s
...   array([[19, 22],
             [43, 50]])
```

다차원 배열의 연산

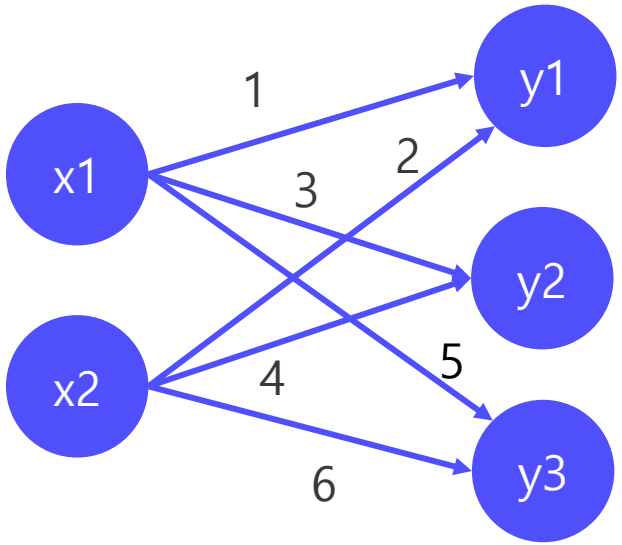
행렬의 곱

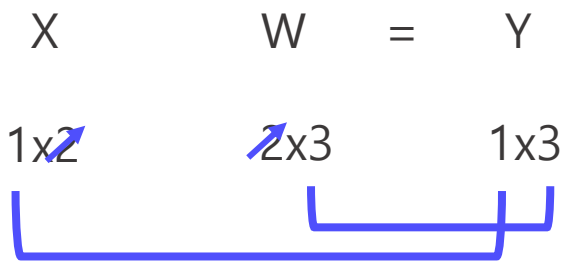


차원 수 맞추기 매우 중요!

다차원 배열의 연산

신경망에서의 행렬 곱






```
[34]  X = np.array([1,2])
      X.shape
      ✓ 0.2s
...   (2,)

      W = np.array([[1,3,5],[2,4,6]])
      print(W)
[35]  ✓ 0.2s
...   [[1 3 5]
       [2 4 6]]

      W.shape
[37]  ✓ 0.2s
...   (2, 3)

      Y = np.dot(X,W)
      print(Y)
[39]  ✓ 0.1s
...   [ 5 11 17]
```

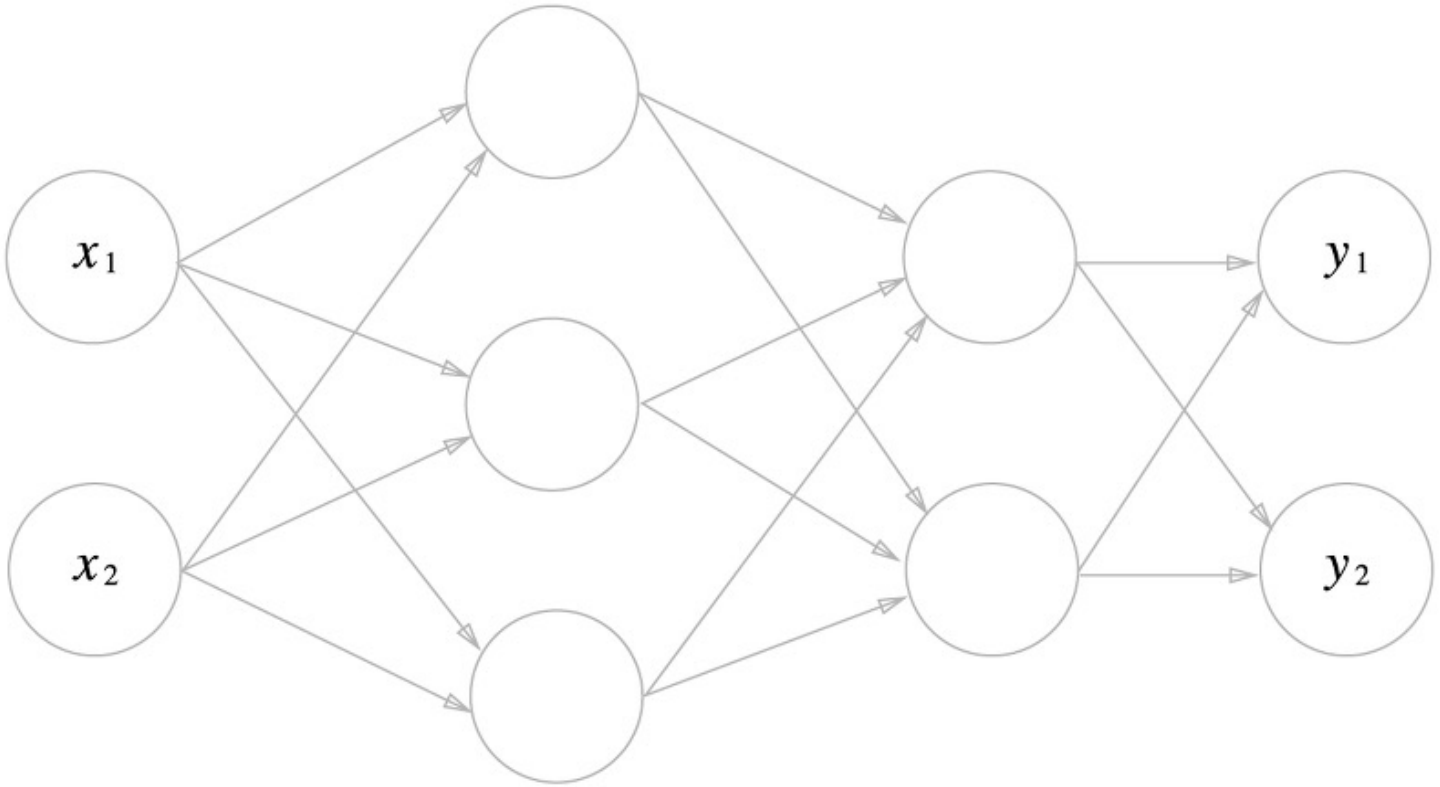
3층 신경망 구현하기

입력층 2개

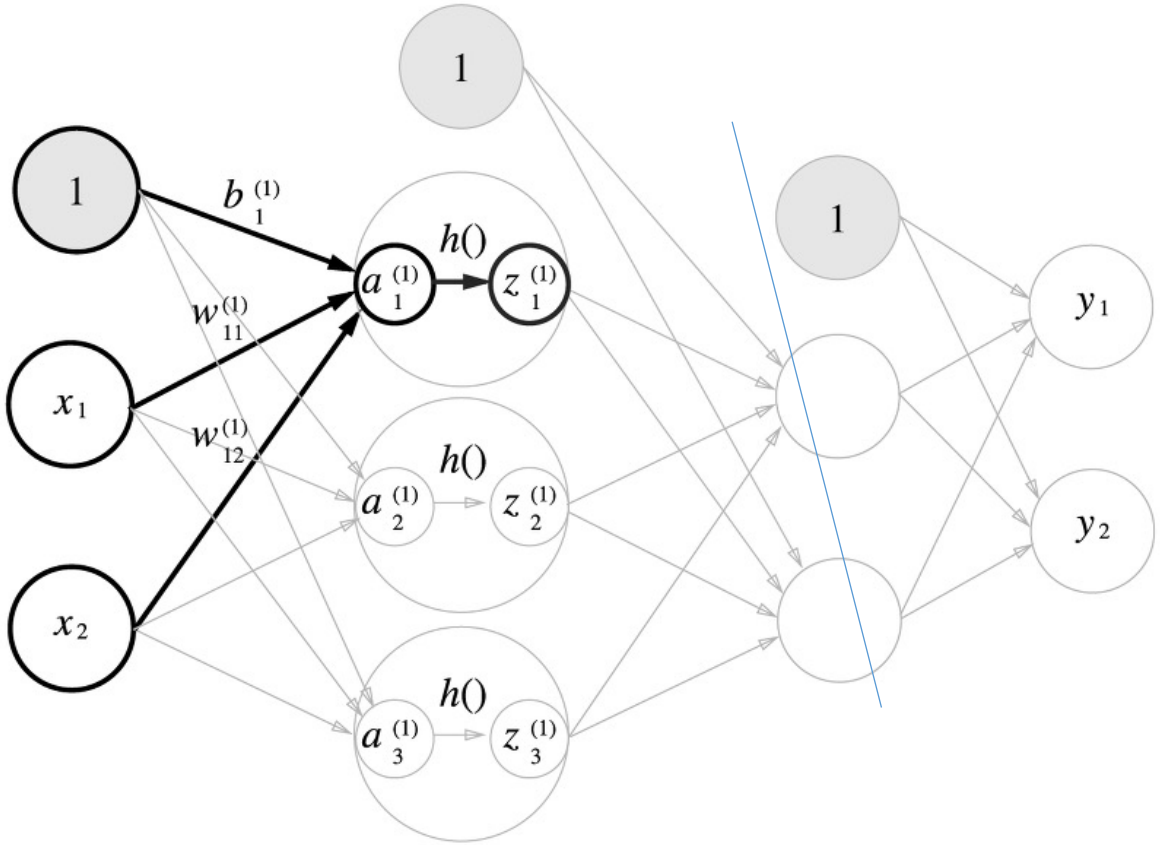
첫번째 은닉층 3개

두번째 은닉층 2개

출력층 2개



3층 신경망 구현하기




```
X = np.array([1.0,0.5])
W1 = np.array([[0.1,0.3,0.5],[0.2,0.4,0.6]])
B1 = np.array([0.1,0.2,0.3])
```

[41] ✓ 0.6s

```
print(W1.shape)
print(X.shape)
print(B1.shape)

A1 = np.dot(X, W1) + B1
```

[42] ✓ 0.3s

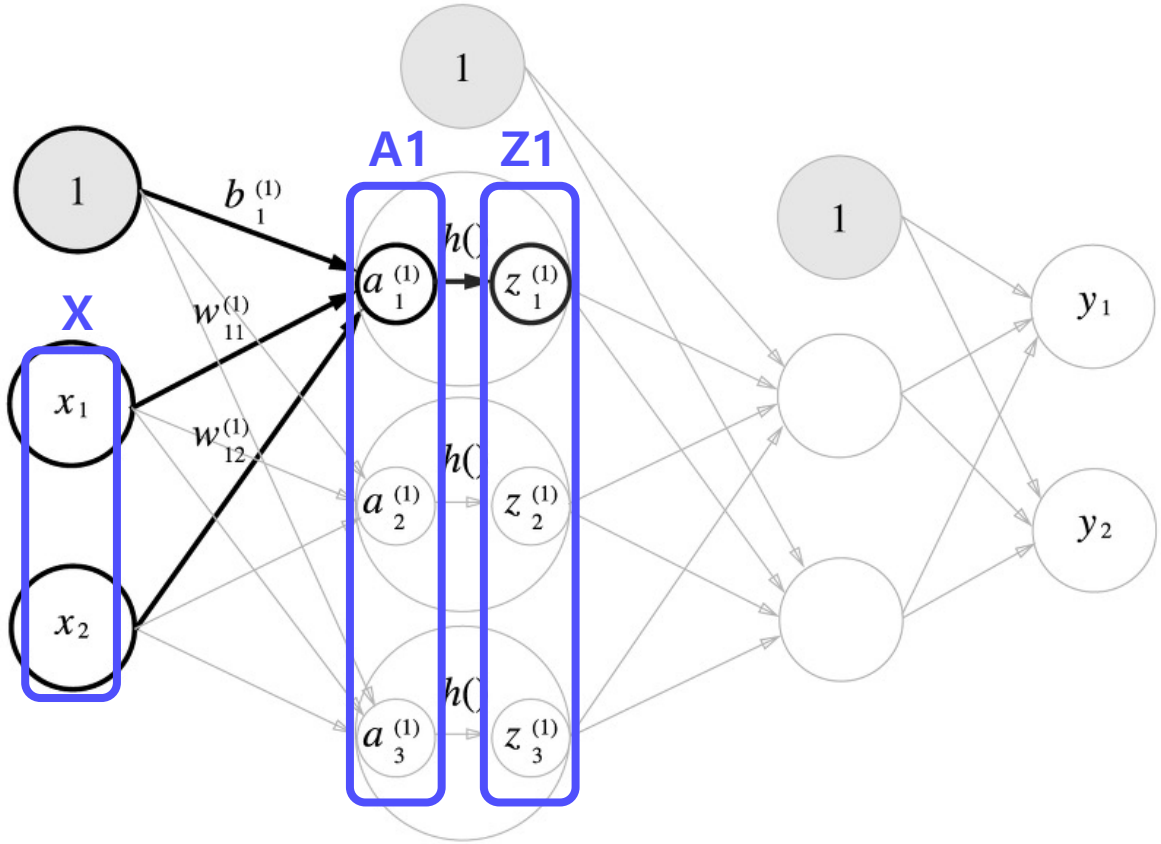
```
(2, 3)
(2,)
(3,)
```

```
Z1 = sigmoid(A1)
print(A1)
print(Z1)
```

[44] ✓ 0.1s

```
[0.3 0.7 1.1]
[0.57444252 0.66818777 0.75026011]
```

3층 신경망 구현하기




```
X = np.array([1.0,0.5])
W1 = np.array([[0.1,0.3,0.5],[0.2,0.4,0.6]])
B1 = np.array([0.1,0.2,0.3])
```

[41] ✓ 0.6s

```
print(W1.shape)
print(X.shape)
print(B1.shape)

A1 = np.dot(X, W1) + B1
```

[42] ✓ 0.3s

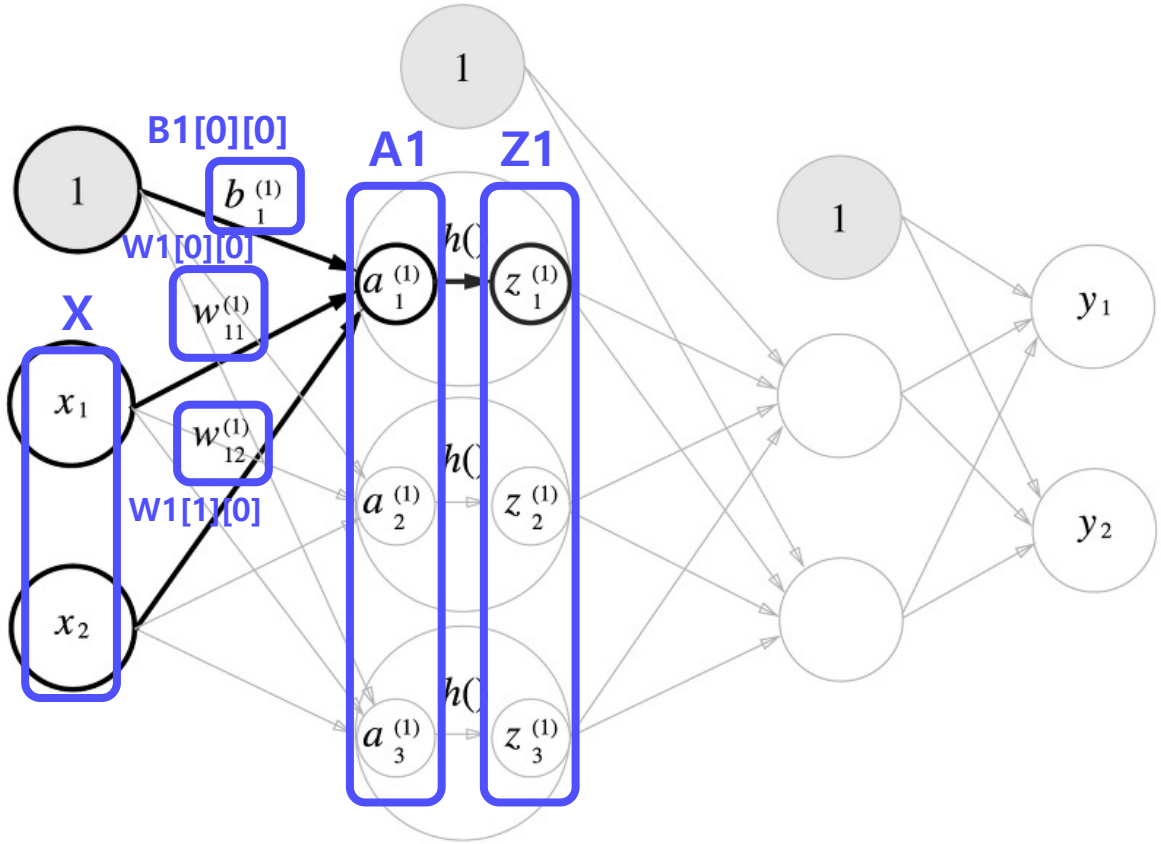
```
(2, 3)
(2,)
(3,)
```

```
Z1 = sigmoid(A1)
print(A1)
print(Z1)
```

[44] ✓ 0.1s

```
[0.3 0.7 1.1]
[0.57444252 0.66818777 0.75026011]
```

3층 신경망 구현하기




```
X = np.array([1.0,0.5])
W1 = np.array([[0.1,0.3,0.5],[0.2,0.4,0.6]])
B1 = np.array([0.1,0.2,0.3])
```

[41] ✓ 0.6s

```
print(W1.shape)
print(X.shape)
print(B1.shape)

A1 = np.dot(X, W1) + B1
```

[42] ✓ 0.3s

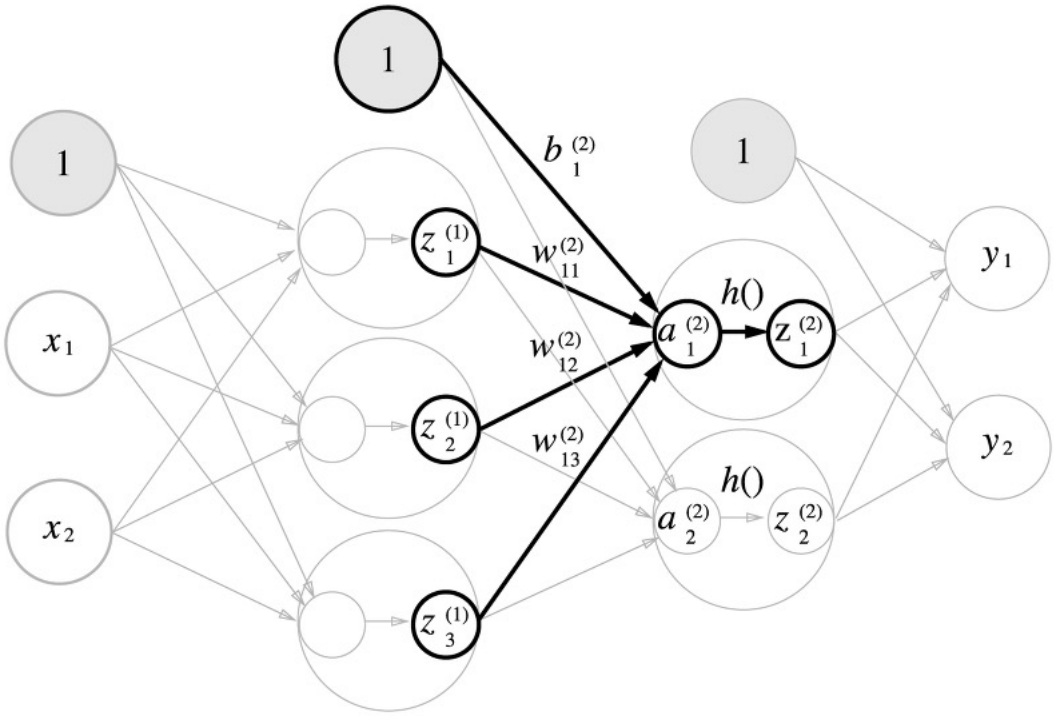
```
(2, 3)
(2,)
(3,)
```

```
Z1 = sigmoid(A1)
print(A1)
print(Z1)
```

[44] ✓ 0.1s

```
[0.3 0.7 1.1]
[0.57444252 0.66818777 0.75026011]
```

3층 신경망 구현하기



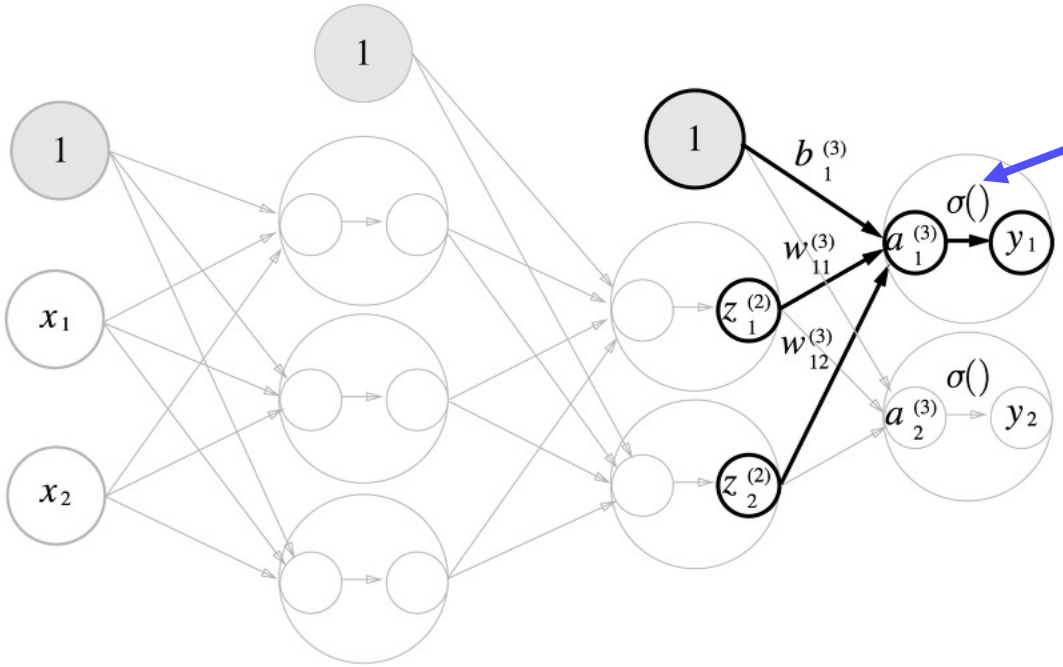

```
W2 = np.array([[0.1,0.4],[0.2,0.5],[0.3,0.6]])
B2 = np.array([0.1,0.2])

print(Z1.shape)
print(W2.shape)
print(B2.shape)

A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
```

[45] ✓ 0.2s

```
(3,)
(3, 2)
(2,)
```

3층 신경망 구현하기

은닉층의 활성함수와 출력층의 활성함수가 다름!




```python
def identity_function(x):
    return x

W3 = np.array([[0.1,0.3],[0.2, 0.4]])
B3 = np.array([0.1,0.2])

A3 = np.dot(Z2, W3) + B3
Y = identity_function(A3)
```

✓ 0.5s

3층 신경망 구현하기

구현정리

```python
def init_network():
    network = {}
    network['W1'] = np.array([[0.1,0.3,0.5],[0.2,0.4,0.6]])
    network['b1'] = np.array([0.1,0.2,0.3])
    network['W2'] = np.array([[0.1,0.4],[0.2,0.5],[0.3,0.6]])
    network['b2'] = np.array([0.1,0.2])
    network['W3'] = np.array([[0.1,0.3],[0.2, 0.4]])
    network['b3'] = np.array([0.1,0.2])
    return network

def forward(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 = identity_function(a3)

    return y

network = init_network()
x = np.array([1.0,0.5])
y = forward(network, x)
print(y)
```

[50] ✓ 0.3s

```
[0.31682708 0.69627909]
```

출력층 설계하기

기계 학습 문제는 분류 와 회귀로 나뉩니다.

분류는 데이터가 어느 클래스에 속하는지를 예측하는것

회귀는 입력 데이터에서 (연속적인) 수치를 예측하는것

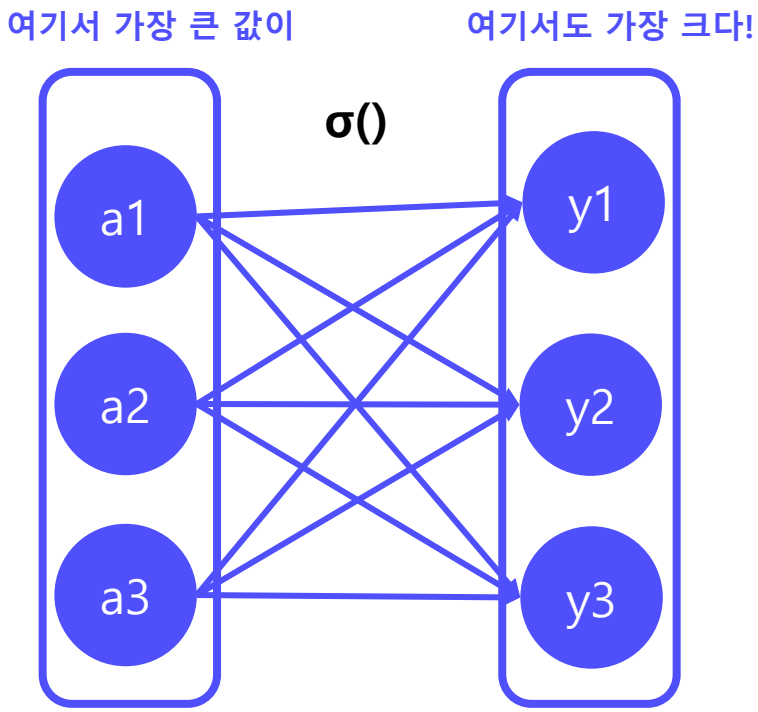
분류인지 회귀인지에 따라 출력층의 활성함수가 달라진다!

출력층 설계하기

항등 함수 : 입력을 그대로 출력함. (회귀)

소프트 맥스 함수 : 입력을 0~1사이의 값들로 만든다.(분류)

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



출력층 설계하기

소프트맥스 구현하기

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

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

출력층 설계하기

소프트맥스 구현 시 주의점

- 지수함수의 내의 a값이 조금만 커져도 컴퓨터가 연산할 수 있는 범위를 넘어서서 계산 자체를 못하는 경우가 생긴다!
- 올바르게 연산이 되지 않을 수 있다.
- 해결방안

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

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

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