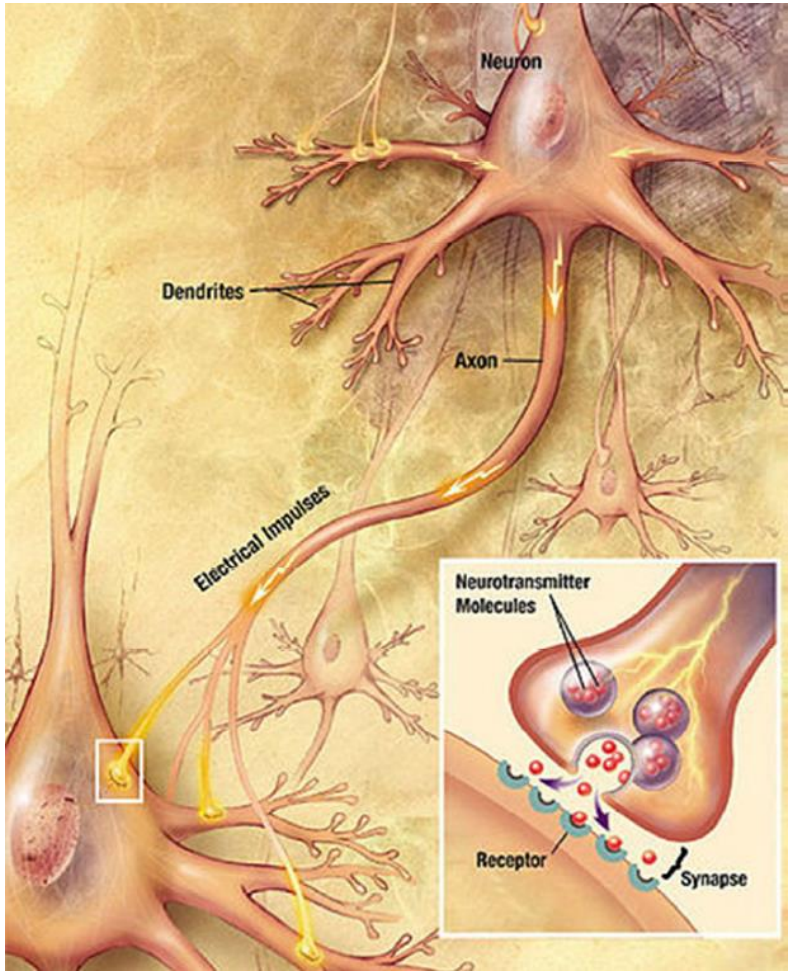
05. 신경망의 기초

- 신경망의 역사 및 종류
- Perceptron의 구조와 학습
- 다층 퍼셉트론(MLP)과 신경망
- 신경망과 학습규칙

(1) 신경망의 역사와 종류

신경망(Neural Network)

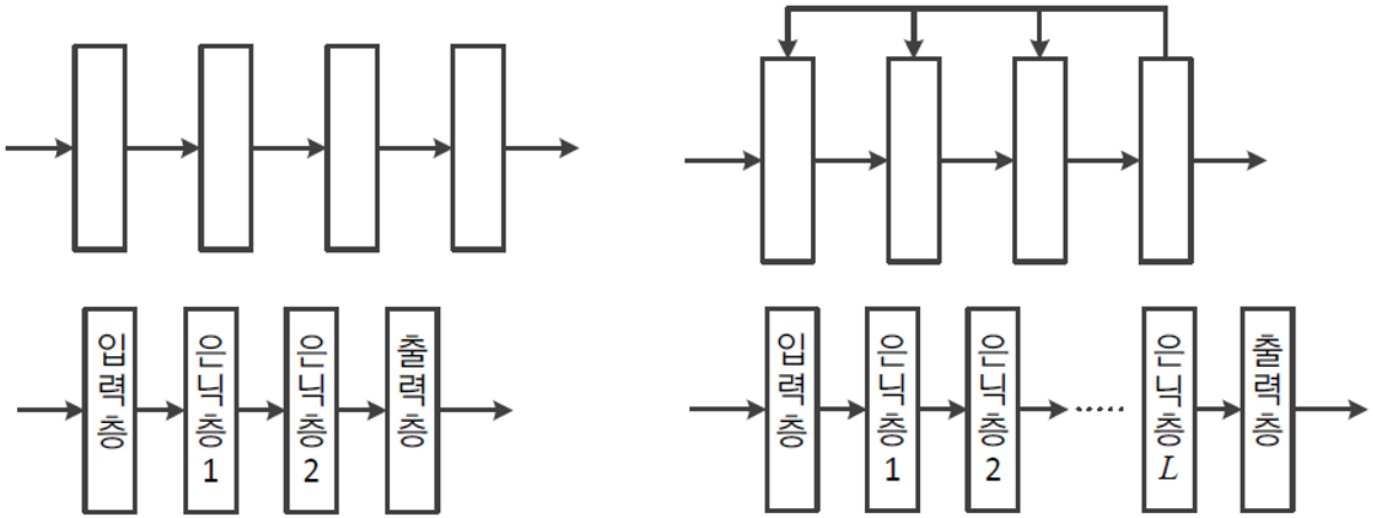
- 뉴런(neuron)
  - 두뇌의 가장 작은 정보처리 단위
    - 세포체(cell body), 수상돌기(dendrite), 축삭돌기(axon)로 구성
    - 사람의 경우 보통 10^{11}개 정도가 있으며, 약 1,000개정도의 다른 뉴런과 연결되어 있으므로 약 10^{14}정도의 연결을 가지고 있음



신경망의 간략한 역사

- 1943년 매컬럭과 피츠의 최초의 신경망
- 1949년 헤브는 최초로 학습 알고리즘 제안
- 1958년 로젠블렛은 퍼셉트론 제안
- 위드로와 호프의 아달린(Adaline)
- 1960년대의 과대 평가
- 1969년 민스키와 페퍼트의 저서 『Perceptrons』는 페셉트론의 한계를 수학적으로 입증
  - 페셉트론은 선형분류기로 XOR 문제조차 해결 못함
- 신경망 연구 퇴조
- 1986년 루멜하트의 저서 『Parallel Distributed Processing』은 다층 퍼셉트론 제안
- 신경망 연구 부활
- 1990년대 SVM에 밀리는 형국
- 2000년대 딥러닝이 실현되어 신경망이 기계 학습의 주류 기술로 자리매김

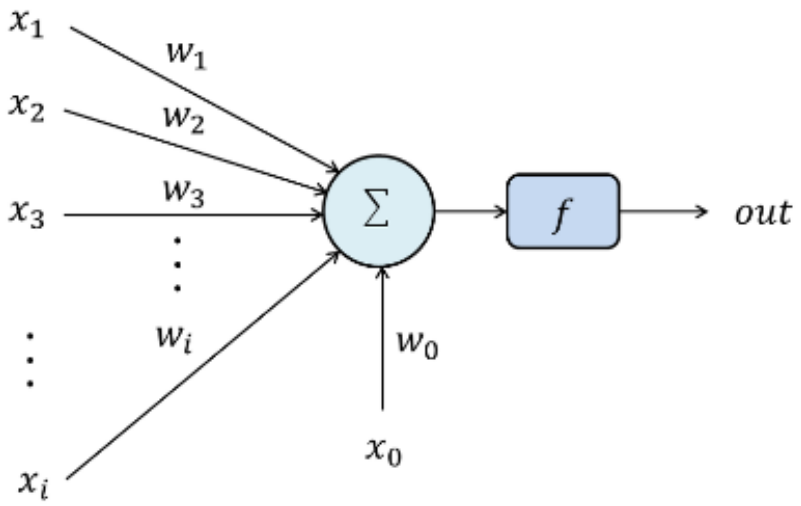
- **신경망에는 아주 다양한 모델이 존재함**
  - **전방 신경망과 순환 신경망**
  - **얕은 신경망과 깊은 신경망**
  - **결정론 신경망과 스토캐스틱 신경망**
    - 결정론신경망 : 입력이 같으면 항상 같은 결과
    - 스토캐스틱신경망 : 매번 다른 결과(난수 사용)



(2) Perceptron의 구조와 학습

Perceptron

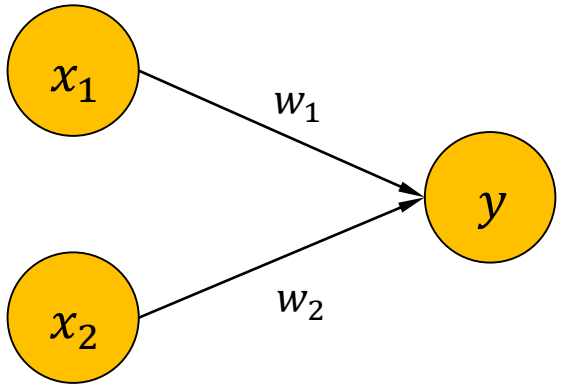
- **퍼셉트론(Perceptron)**
  - 1957년 Frank Rosenblatt가 고안한 알고리즘
  - **단층 신경망 알고리즘**
    - **다수의 신호를 입력으로 받아 하나의 신호를 출력**
    - 신경망(딥러닝)의 기원



Output의 종류

1 : 흐른다
0 : 안 흐른다

퍼셉트론(Perceptron)의 동작원리



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

입력신호 : x_1, x_2　　가중치 : w_1, w_2
출력신호 : y　　임계값 : θ

- 그림의 원 : 뉴런(neuron)/노드(node)
- 가중치는 전류의 저항과 비슷한 의미(가중치가 크면 더 강한신호를 흘려 보냄)
- 뉴런에서 보내온 신호의 총합이 θ(임계값)보다 큰 경우 ⇒ 1을 출력(활성화)

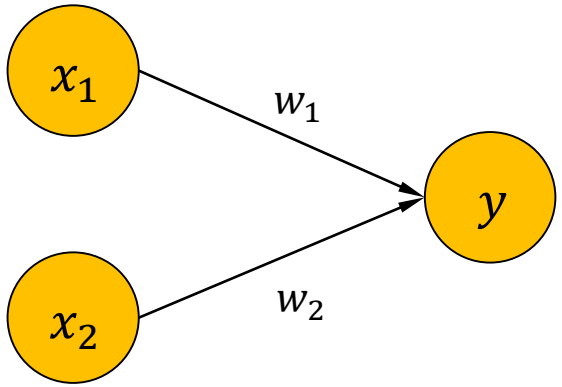
간단한 퍼셉트론의 활용 예제

- 논리회로 AND NAND OR XOR
- 입력은 2개, 출력은 1개인 게이트
- (예) AND 게이트

X_1	X_2	Y
0	0	0
1	0	0
0	1	0
1	1	1

- 퍼셉트론을 이용하여 AND게이트를 표현하기 위해서는 다음 값을 설정해야 함 => 어떻게???
  - **가중치** : w_1 w_2 **임계값** : θ
- 만족하는 가중치와 임계값의 조합은 무수히 많음
  - (0.5, 0.5, 0.7), (0.5, 0.5, 0.8), (1.0, 1.0, 1.0) ...

편향(bias)을 도입한 퍼셉트론



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

입력신호 : x_1, x_2 가중치 : w_1, w_2
출력신호 : y 임계값 : θ

- $b = -\theta$ 로 변경하면,

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

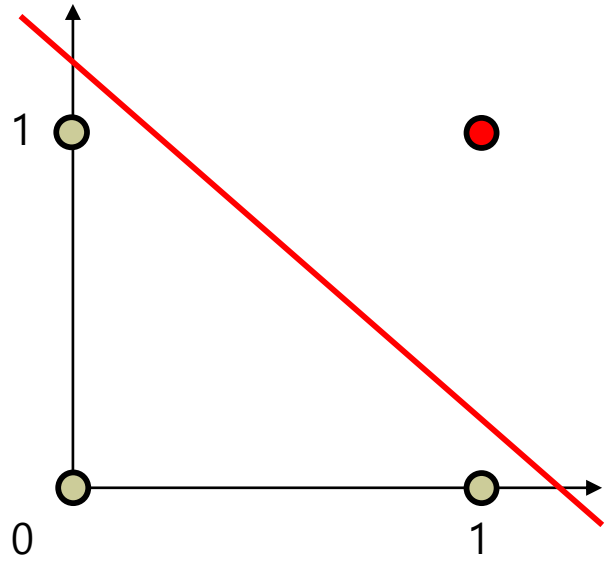
입력신호 : x_1, x_2 가중치 : w_1, w_2
출력신호 : y 편향 : b

Perceptron 구현

- **AND (0.5, 0.5, -0.7)**

```
import numpy as np

def AND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.7
    y = np.sum(w*x) + b
    if y > 0:
        return 1
    else:
        return 0

def main():
    print("(0, 0) => ", AND(0, 0))
    print("(0, 1) => ", AND(0, 1))
    print("(1, 0) => ", AND(1, 0))
    print("(1, 1) => ", AND(1, 1))

main()
```

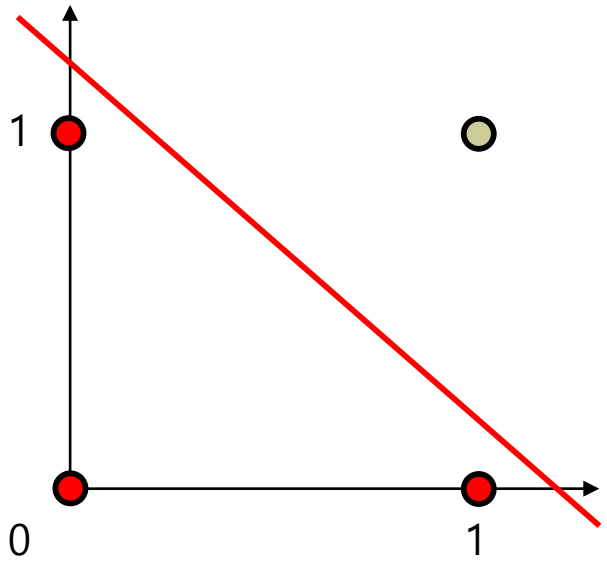
```
(0, 0) =>  0
(0, 1) =>  0
(1, 0) =>  0
(1, 1) =>  1
```



NAND (-0.5, -0.5, 0.7)

```
import numpy as np

def NAND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([-0.5, -0.5])
    b = 0.7
    y = np.sum(w*x) + b
    if y > 0:
        return 1
    else:
        return 0

def main():
    print("(0, 0) => ", NAND(0, 0))
    print("(0, 1) => ", NAND(0, 1))
    print("(1, 0) => ", NAND(1, 0))
    print("(1, 1) => ", NAND(1, 1))

main()
```

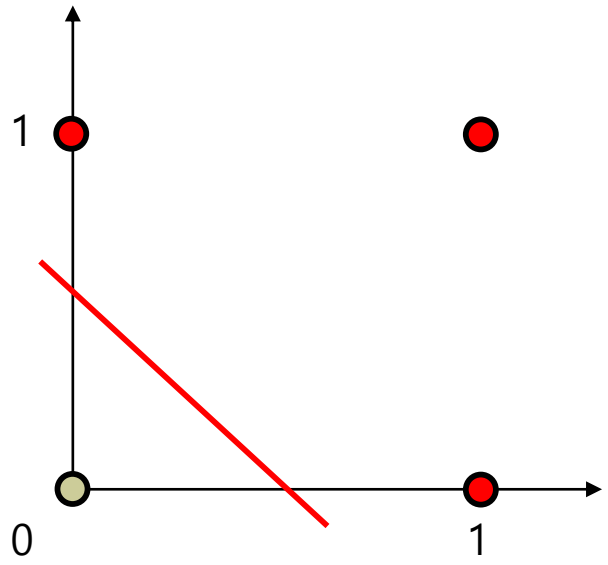
```
(0, 0) =>  1
(0, 1) =>  1
(1, 0) =>  1
(1, 1) =>  0
```



OR (0.5, 0.5, -0.4)

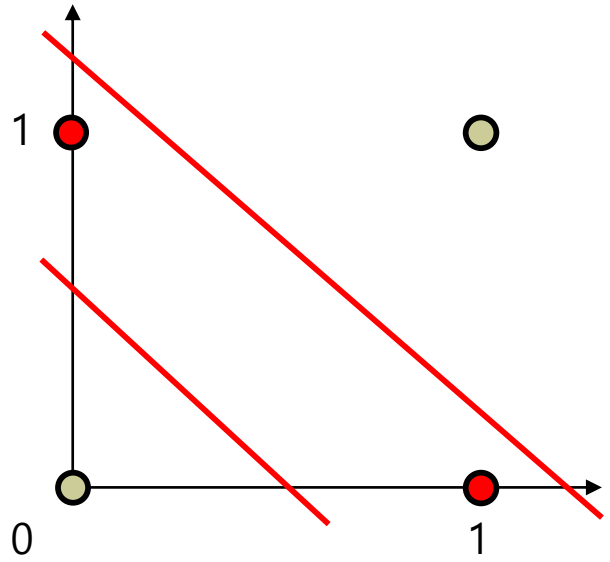
```
import numpy as np

def OR(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.4
    y = np.sum(w*x) + b
    if y > 0:
        return 1
    else:
        return 0

def main():
    print("(0, 0) => ", OR(0, 0))
    print("(0, 1) => ", OR(0, 1))
    print("(1, 0) => ", OR(1, 0))
    print("(1, 1) => ", OR(1, 1))

main()
```

```
(0, 0) =>  0
(0, 1) =>  1
(1, 0) =>  1
(1, 1) =>  1
```



- **XOR**
  - 직선 1개로 나누는 것은??
  - 여러가지 경우를 적용해 봐도...
  - => 불가능

- **왜?**
  - 퍼셉트론은 θ를 기준으로 2개의 부류로만 나누는 단층 신경망

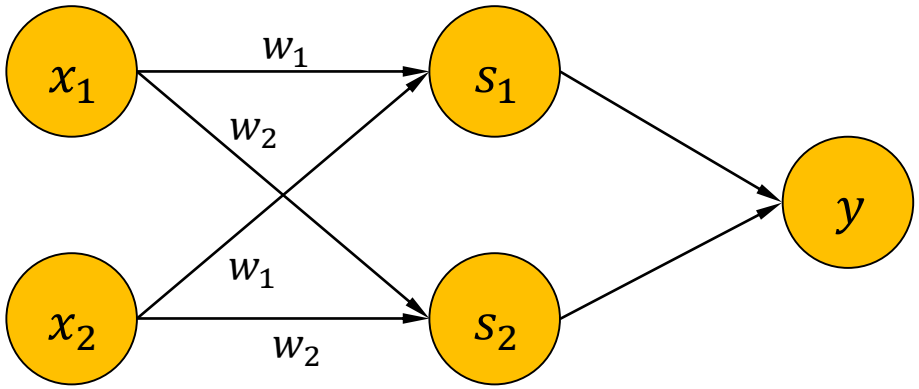


- **그러면 어떻게??**

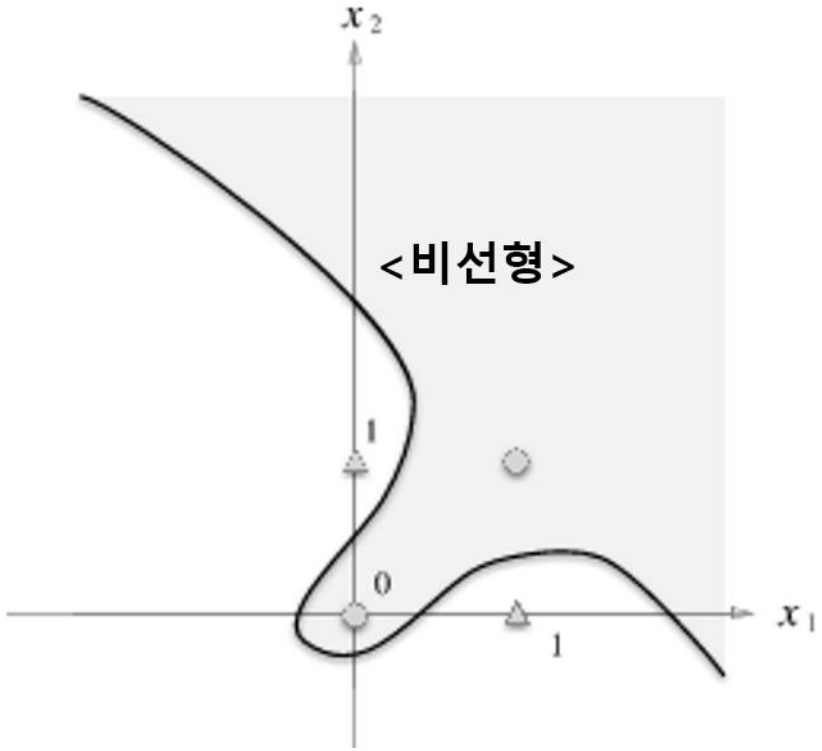
Perceptron의 한계 극복

- XOR 문제 => 다층 퍼셉트론으로 해결
  - AND, NAND, OR 게이트를 조합해서 해결 가능

X_1	X_2	(NAND) S_1	(OR) S_2	(AND) Y
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	1	0



- 다층 퍼셉트론(Multi-layer Perceptron)
  - 비선형 특성을 갖음



XOR의 구현

```python
import numpy as np

def AND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.7
    y = np.sum(w*x) + b
    if y > 0:
        return 1
    else:
        return 0

def NAND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([-0.5, -0.5])
    b = 0.7
    y = np.sum(w*x) + b
    if y > 0:
        return 1
    else:
        return 0
```

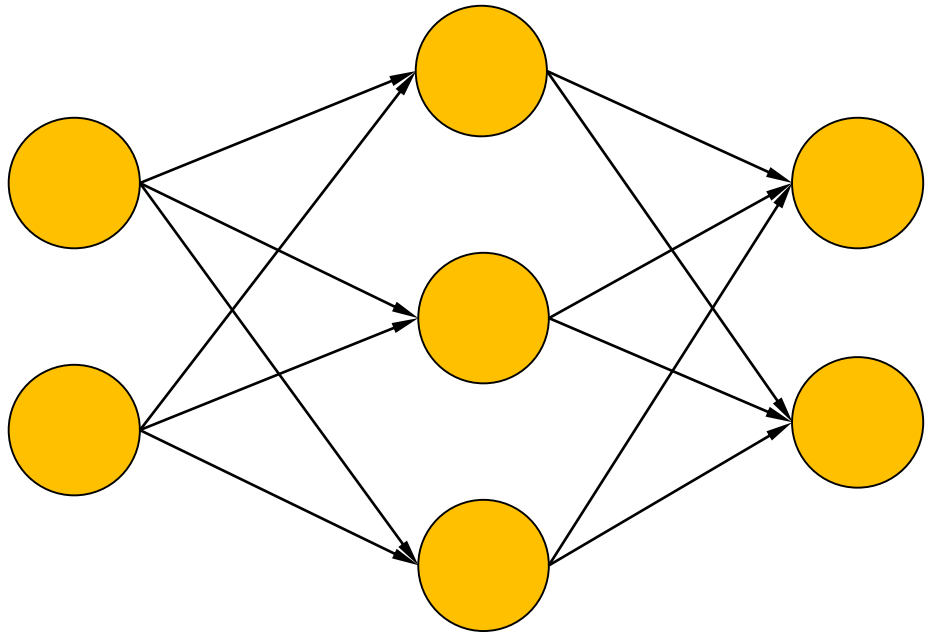
```python
def OR(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.4
    y = np.sum(w*x) + b
    if y > 0:
        return 1
    else:
        return 0

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

def main():
    print("(0, 0) => ", XOR(0, 0))
    print("(0, 1) => ", XOR(0, 1))
    print("(1, 0) => ", XOR(1, 0))
    print("(1, 1) => ", XOR(1, 1))

main()
```

```
(0, 0) =>  0
(0, 1) =>  1
(1, 0) =>  1
(1, 1) =>  0
```

(3) 다층 퍼셉트론과 신경망

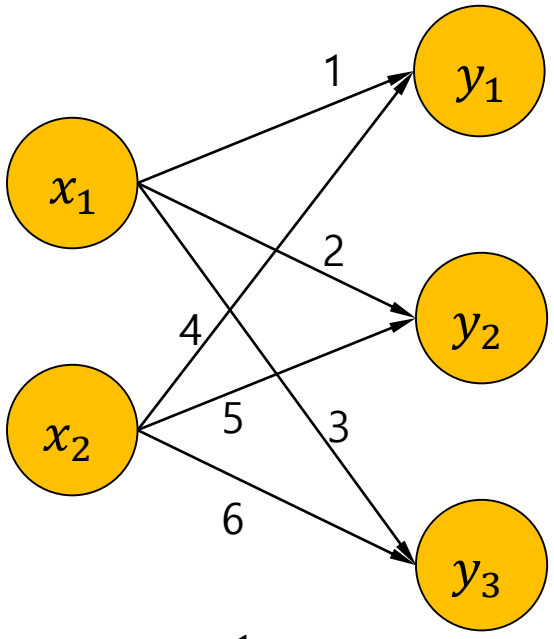
신경망(Neural-Net)

- 신경망의 구성
  - 입력층 – 은닉층 – 출력층
  - 가중치를 갖는 층은 입력층과 은닉층뿐이므로 2층 신경망이라고도 함



간단한 신경망 구현하기

- 행렬의 곱셈으로 신경망 계산을 수행



$x_1 = 1$
$x_2 = 2$

$w_1 = [1, 2, 3]$
$w_2 = [4, 5, 6]$

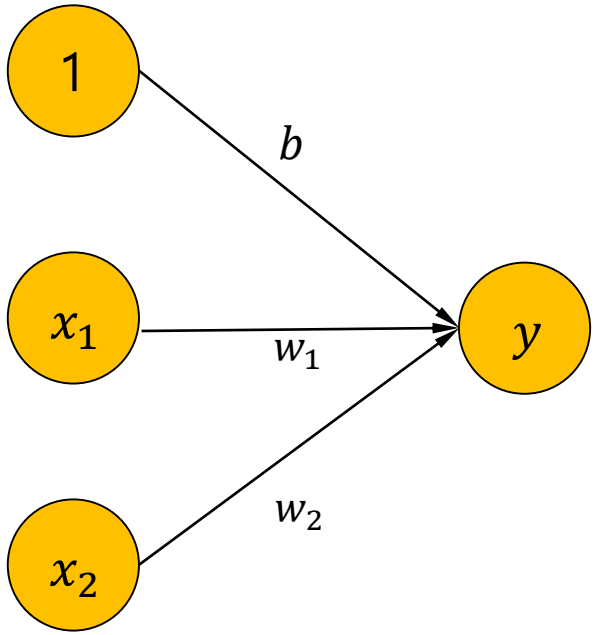
$$[1 \quad 2] \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = [9 \quad 12 \quad 15]$$

(1,2) X (2,3) = (1,3)

```
import numpy as np

X = np.array([1,2])
W = np.array([[1,2,3],
              [4,5,6]])
Y = np.dot(X, W)
# shape : tuple에 요소가 1개이면 콤마를 뒤에 붙임
print('shape of X =', X.shape)
print(X)
print('shape of W =', W.shape)
print(W)
print('shape of Y =', Y.shape)
print(Y)
```

```
shape of X = (2,)
[1 2]
shape of W = (2, 3)
[[1 2 3]
 [4 5 6]]
shape of Y = (3,)
[ 9 12 15]
```

퍼셉트론에 편향(bias) 표현하기



$$y = \boldsymbol{h}(w_1x_1 + w_2x_2 + b)$$

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

활성함수(activation function)

입력신호 : x_1, x_2　　가중치 : w_1, w_2
출력신호 : y　　편향 : b

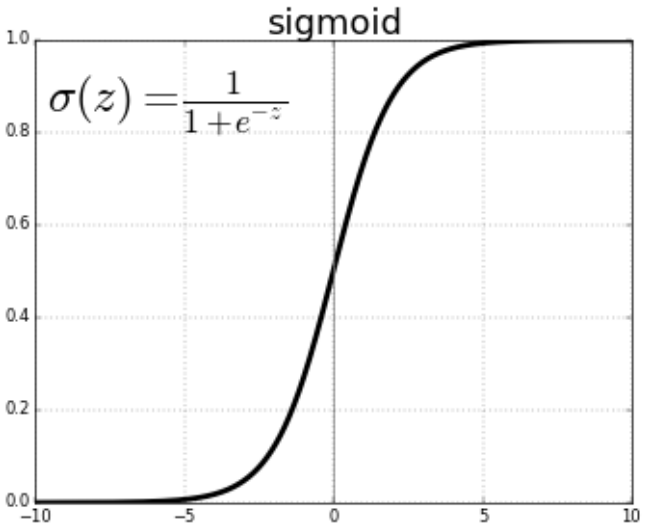
활성함수의 종류

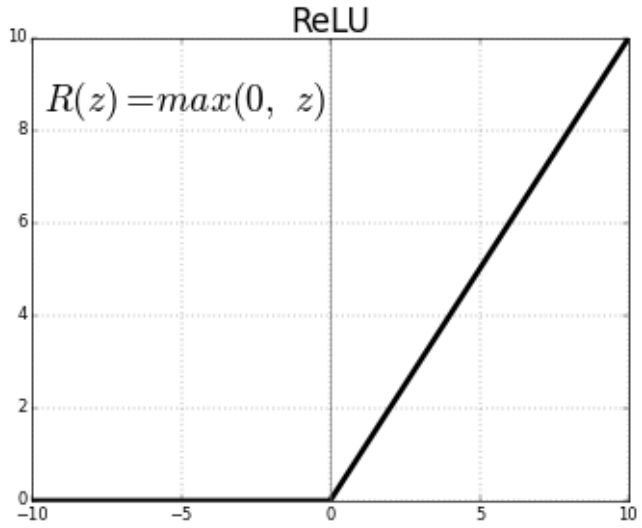
- **Sigmoid function**

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

- **ReLU function**

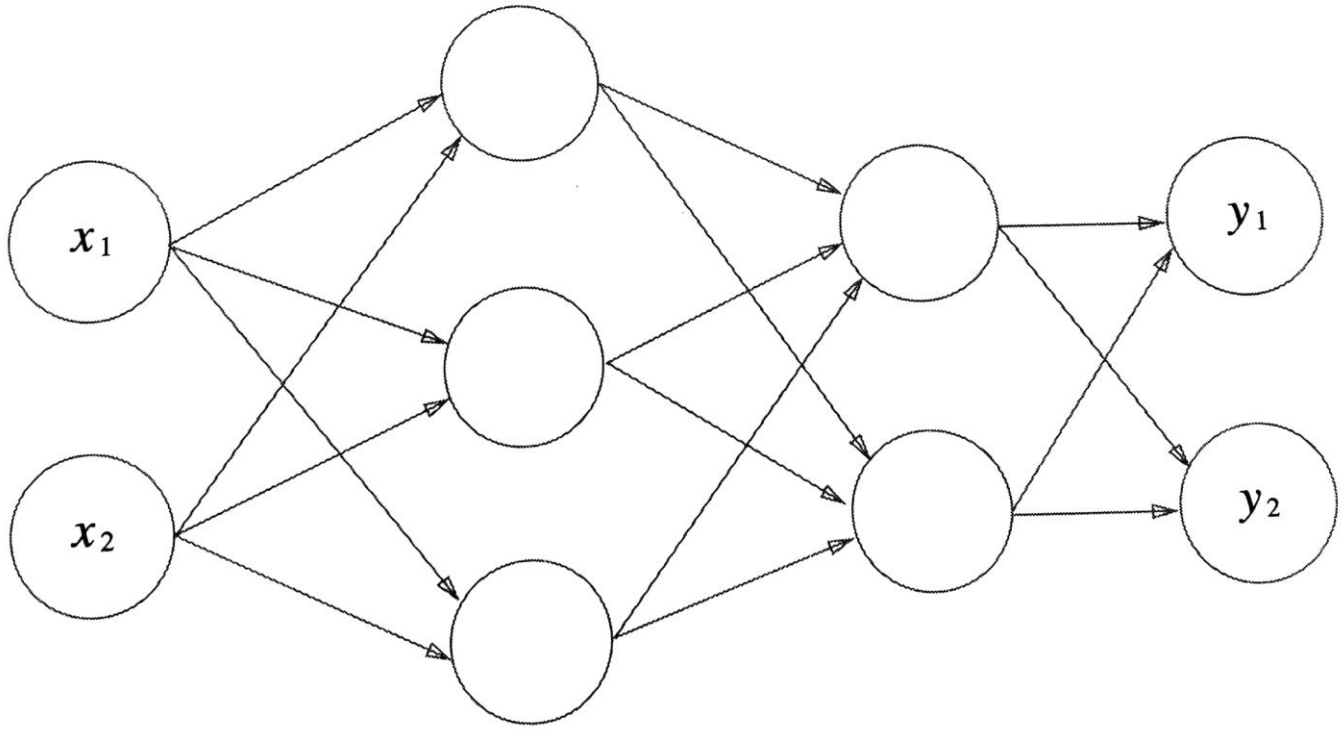
$$h(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$$



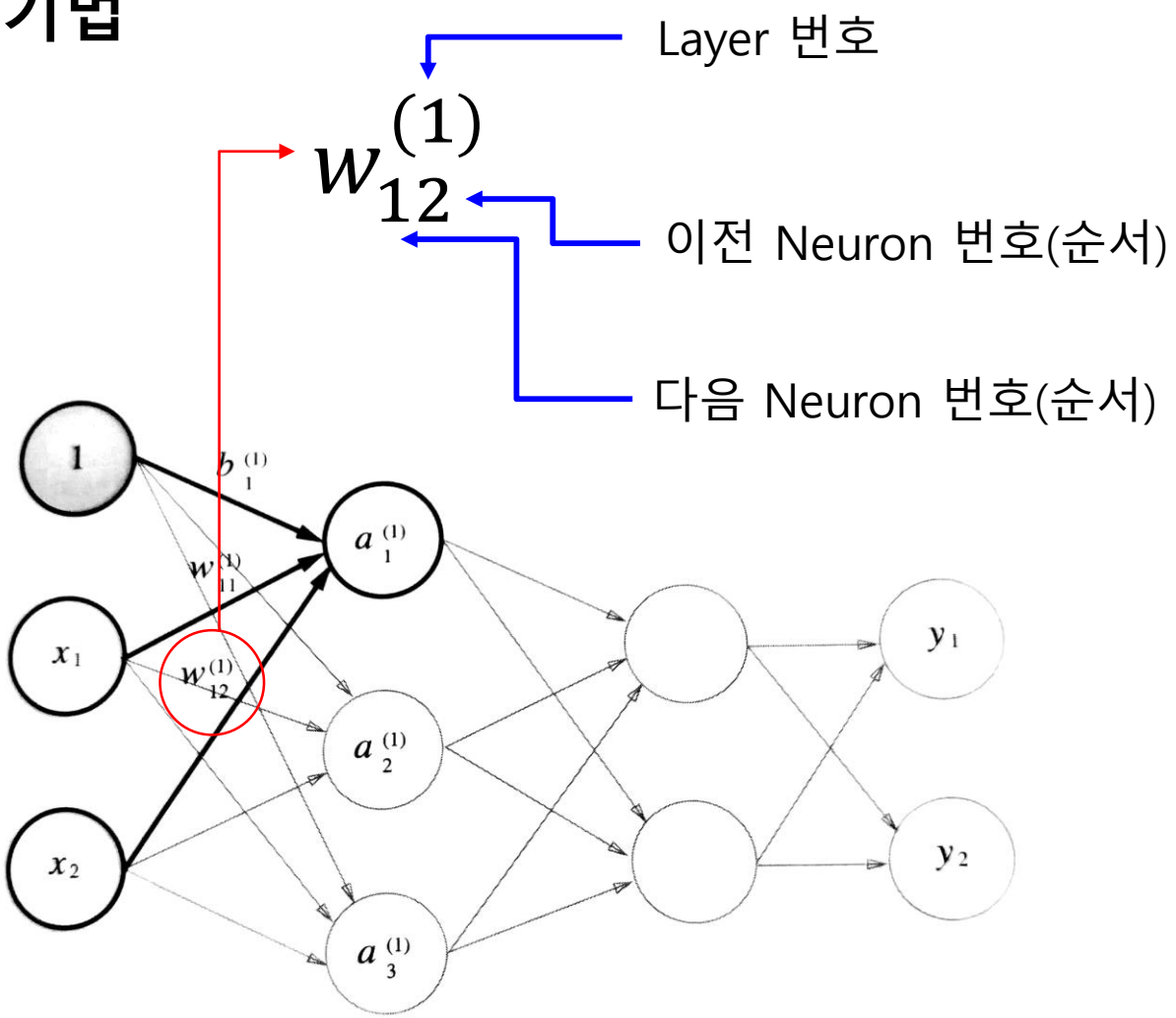


다층 퍼셉트론의 구현

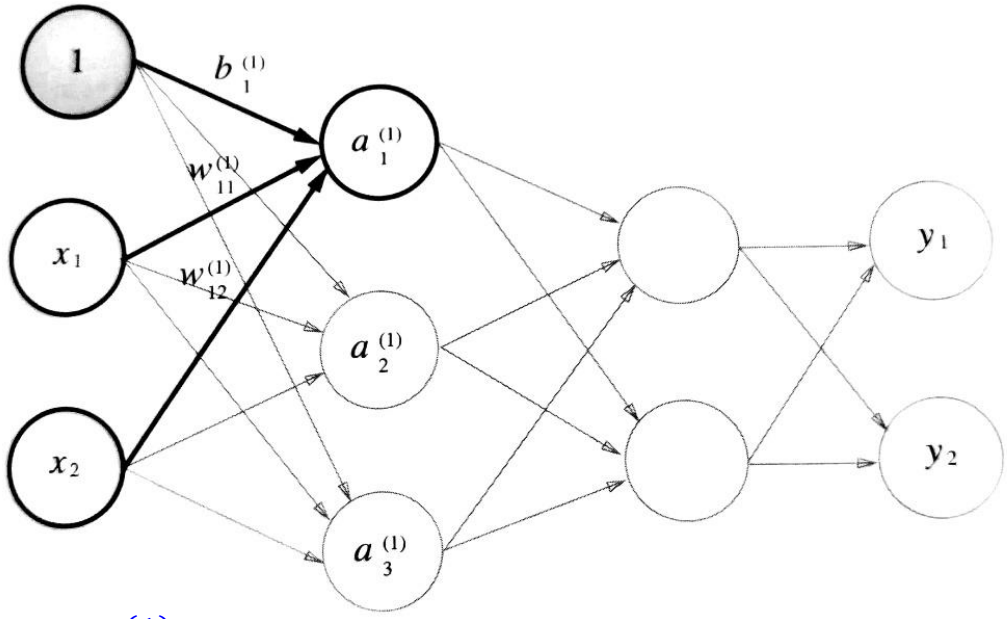
- 간단한 3층 신경망을 구현해 보자!
  - "밑바닥부터 시작하는 딥러닝" 참조



표기법


Layer 번호
$w_{12}^{(1)}$
이전 Neuron 번호(순서)
다음 Neuron 번호(순서)
1
$b_1^{(1)}$
$a_1^{(1)}$
$w_{11}^{(1)}$
x_1
$w_{12}^{(1)}$
$a_2^{(1)}$
x_2
$a_3^{(1)}$
y_1
y_2

신호의 계산



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

$$a_2^{(1)} = w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + b_2^{(1)}$$

$$a_3^{(1)} = w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + b_3^{(1)}$$

- 따라서,

$$A^{(1)} = X\,W^{(1)} + B^{(1)}$$

$$W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{bmatrix}$$

$$X = [x_1 \;\; x_2]$$

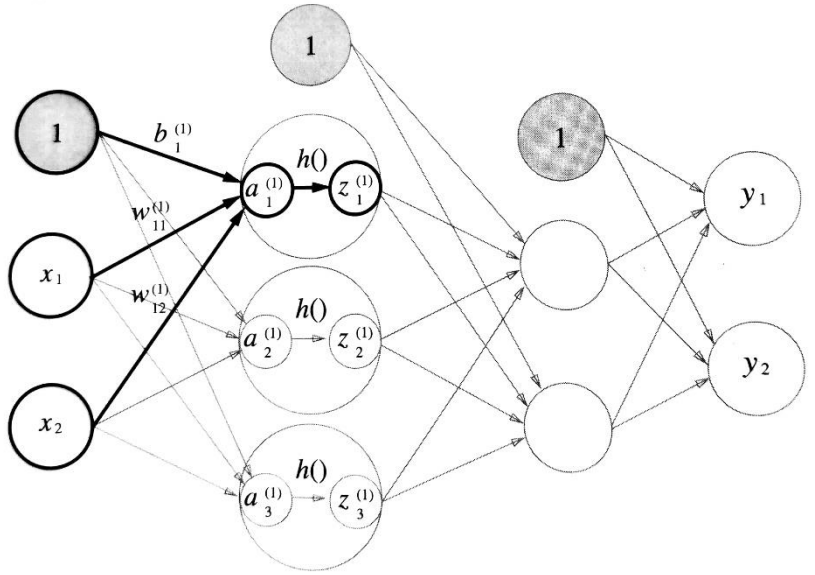
$$B^{(1)} = [b_1^{(1)} \;\; b_2^{(1)} \;\; b_3^{(1)}]$$

$$A^{(1)} = [a_1^{(1)} \;\; a_2^{(1)} \;\; a_3^{(1)}]$$

step1: MLP구현(1번 레이어)

- 가중치 합
  - $A^{(1)} = X\,W^{(1)} + B^{(1)}$
- 활성함수 적용
  - sigmoid function

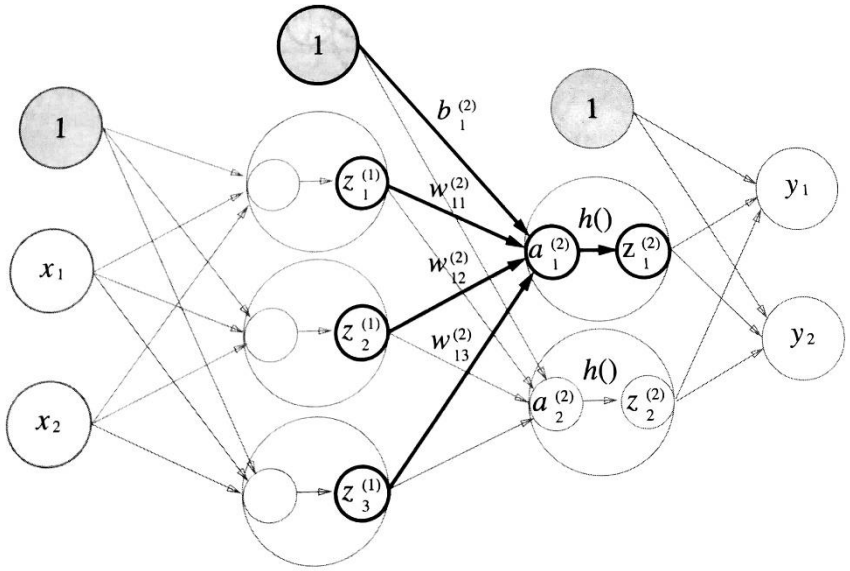
```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# input nodes
X = np.array([1.0, 0.5])

# compute layer1 nodes
W1 = np.array([[0.1, 0.3, 0.5],
               [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])
A1 = np.dot(X, W1) + B1
Z1 = sigmoid(A1)

print(A1)
print(Z1)
```




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

step2: MLP구현(2번 레이어)

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# input nodes
X = np.array([1.0, 0.5])

# compute layer1 nodes
W1 = np.array([[0.1, 0.3, 0.5],
               [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])
A1 = np.dot(X, W1) + B1
Z1 = sigmoid(A1)

# compute Layer2 nodes
W2 = np.array([[0.1, 0.4],
               [0.2, 0.5],
               [0.3, 0.6]])
B2 = np.array([0.1, 0.2])
A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)

print(A2)
print(Z2)
```



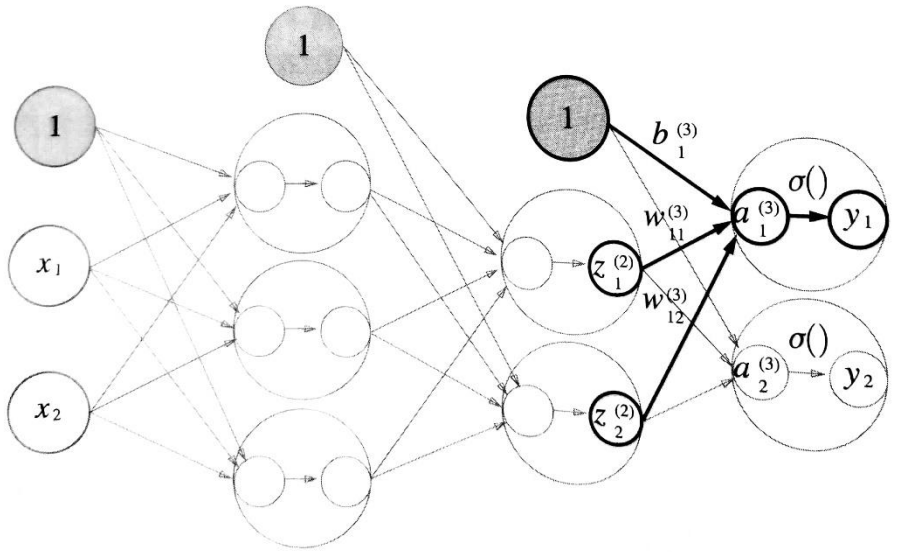

```
[ 0.51615984  1.21402696]
[ 0.62624937  0.7710107 ]
```

step3: MLP구현(출력 레이어)

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# input nodes
X = np.array([1.0, 0.5])

# compute layer1 nodes
W1 = np.array([[0.1, 0.3, 0.5],
               [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])
A1 = np.dot(X, W1) + B1
Z1 = sigmoid(A1)

# compute Layer2 nodes
W2 = np.array([[0.1, 0.4],
               [0.2, 0.5],
               [0.3, 0.6]])
B2 = np.array([0.1, 0.2])
A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
```

```python
# compute output layer nodes
W3 = np.array([[0.1, 0.3],
               [0.2, 0.4]])
B3 = np.array([0.1, 0.2])
A3 = np.dot(Z2, W3) + B3

print(A3)
```

```
[ 0.31682708  0.69627909]
```



- 참고 : softmax function
  - 출력층의 값을 어떻게 출력할지를 결정
  - 출력값들을 확률로 변환
  - 주로 분류(classification) 문제에 사용됨

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

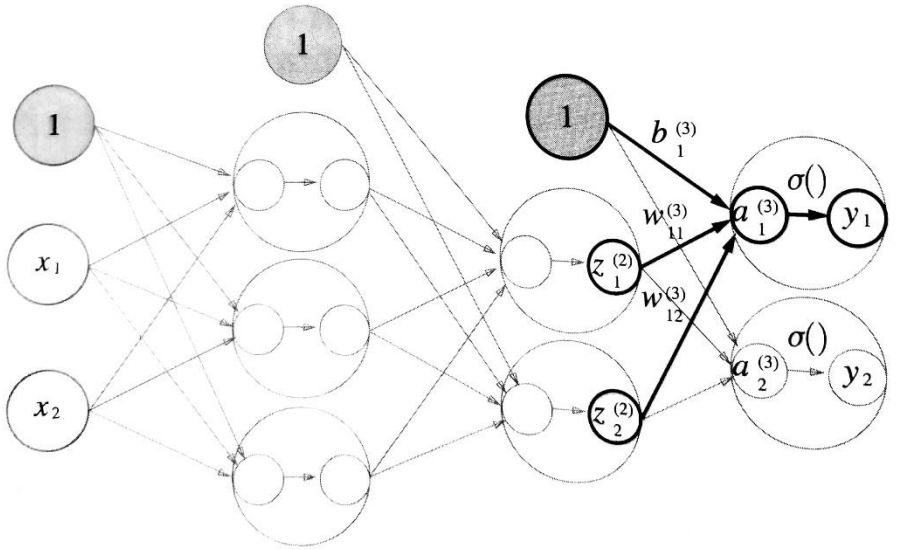
- 지수함수의 결과는 매우 큰 값을 가지는 경우가 많음
- 지수함수는 양쪽에 어떤 값을 더하거나 빼도 그 결과는 같음
- 따라서 a리스트의 최대값을 빼서 작은 값을 가지도록 하면 오버플로우를 방지할 수 있음

```
def softmax(a):
    m = np.max(a)
    exp_a = np.exp(a - m) # overflow 방지
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a
    return y
```

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# input nodes
X = np.array([1.0, 0.5])

# compute layer1 nodes
W1 = np.array([[0.1, 0.3, 0.5],
               [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])
A1 = np.dot(X, W1) + B1
Z1 = sigmoid(A1)

# compute Layer2 nodes
W2 = np.array([[0.1, 0.4],
               [0.2, 0.5],
               [0.3, 0.6]])
B2 = np.array([0.1, 0.2])
A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
```

```python
# compute output layer nodes
W3 = np.array([[0.1, 0.3],
               [0.2, 0.4]])
B3 = np.array([0.1, 0.2])
A3 = np.dot(Z2, W3) + B3

print(softmax(A3))
```

```
[ 0.31682708  0.69627909]
```



(4) 신경망과 학습규칙

신경망과 학습

- **신경망의 특징**
  - **데이터를 보고 학습할 수 있다!**
    - 가중치와 편향을 **입력된 데이터들을 이용하여 자동으로 결정**한다!

- **학습의 필요성**
  - **퍼셉트론의 경우(논리회로)**
    - 가중치와 편향을 수작업으로 설정
    - 매개변수는 3개
  - **신경망의 경우(딥러닝)**
    - 매개변수가 수천 수만 수억 개 이상이 되는 경우가 많음
    - 따라서 각종 매개변수의 값을 자동으로 결정하는 것이 필요

- **그러면 어떻게 매개변수의 값을 결정할까?**

- **델타규칙(delta rule)**
  - 단층 신경망을 학습시키는 방법 중의 하나
  - 경사하강법(gradient descent)을 이용하여 손실함수(loss function)의 최소값을 찾아내는 방법
    - 최적의 가중치를 찾는 방법

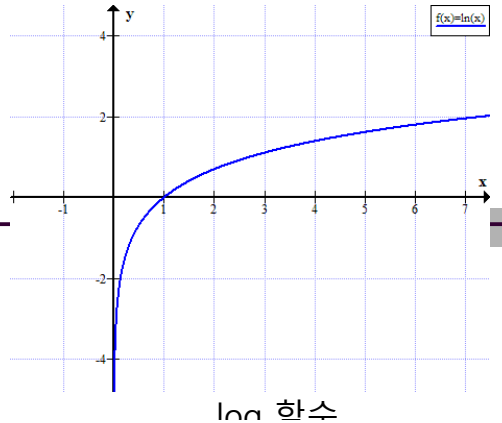
- **손실함수**
  - 학습이 얼마나 잘 되어있는지를 나타내는 지표
  - MSE(Mean squared error) 또는 Cross entropy를 주로 사용

- **경사하강법**
  - 손실함수의 값을 최소화하는 가중치를 찾는 원리

손실함수

- **MSE(Mean Squared Error)**

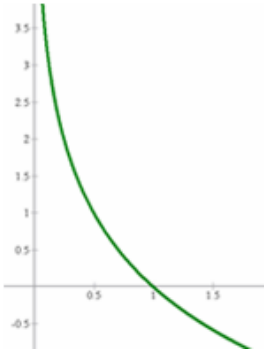
$$E = \frac{1}{n}\sum_{k}(y_k - t_k)^2$$



log 함수

- **Cross Entropy**

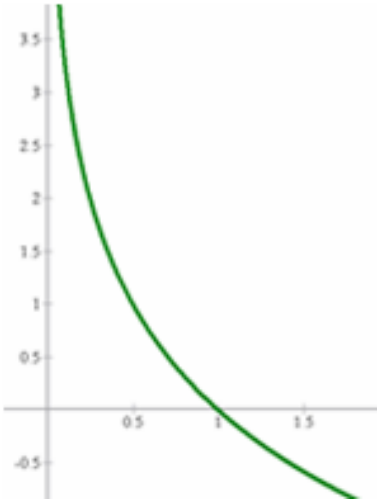
$$E(T,Y) = -\sum_{k} t_k \log y_k = \sum_{k} t_k \cdot -\log y_k$$



  - 두 확률분포(t_k와 y_k)의 차이
    - Entropy : 정보량을 나타내는 단위(음의 log)
      - 정보이론 : 잘 일어나지 않는 사건은 자주 발생하는 사건보다 정보량이 많다
      - (섀넌 엔트로피) 어떤 수치정보를 표현할 때, 필요한 bit의 수를 계산하기 위해 사용(압축 등에 사용)
    - 참고: log함수=>log를 이용하면 곱셈을 덧셈으로 해결
      - $\log xy = \log x + \log y$

왜 Cross-Entropy를 손실함수로 사용할까?

$$E(T,Y) = -\sum_k t_k \log y_k = \sum_k t_k \cdot -\log y_k$$

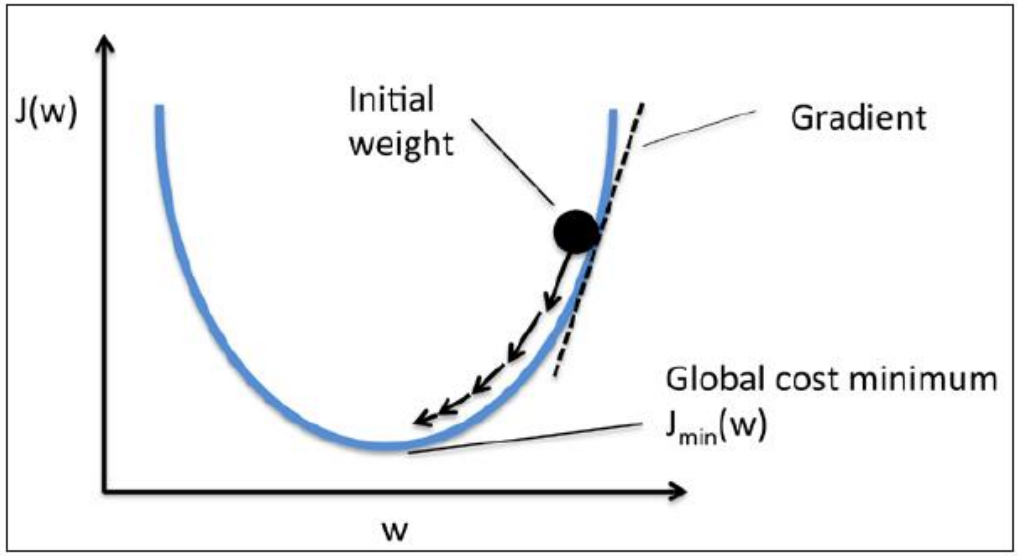


- 예측된 값 t_k에 대해,
  - 옳은 결과 : cost를 0에 가깝게
    즉, 정확도가 높으면(1에 가까우면) 0에 가깝게
  - 틀린 결과 : cost를 무한대에 가깝게
    즉, 정확도가 낮으면(0에 가까우면) 무한대에 가깝게

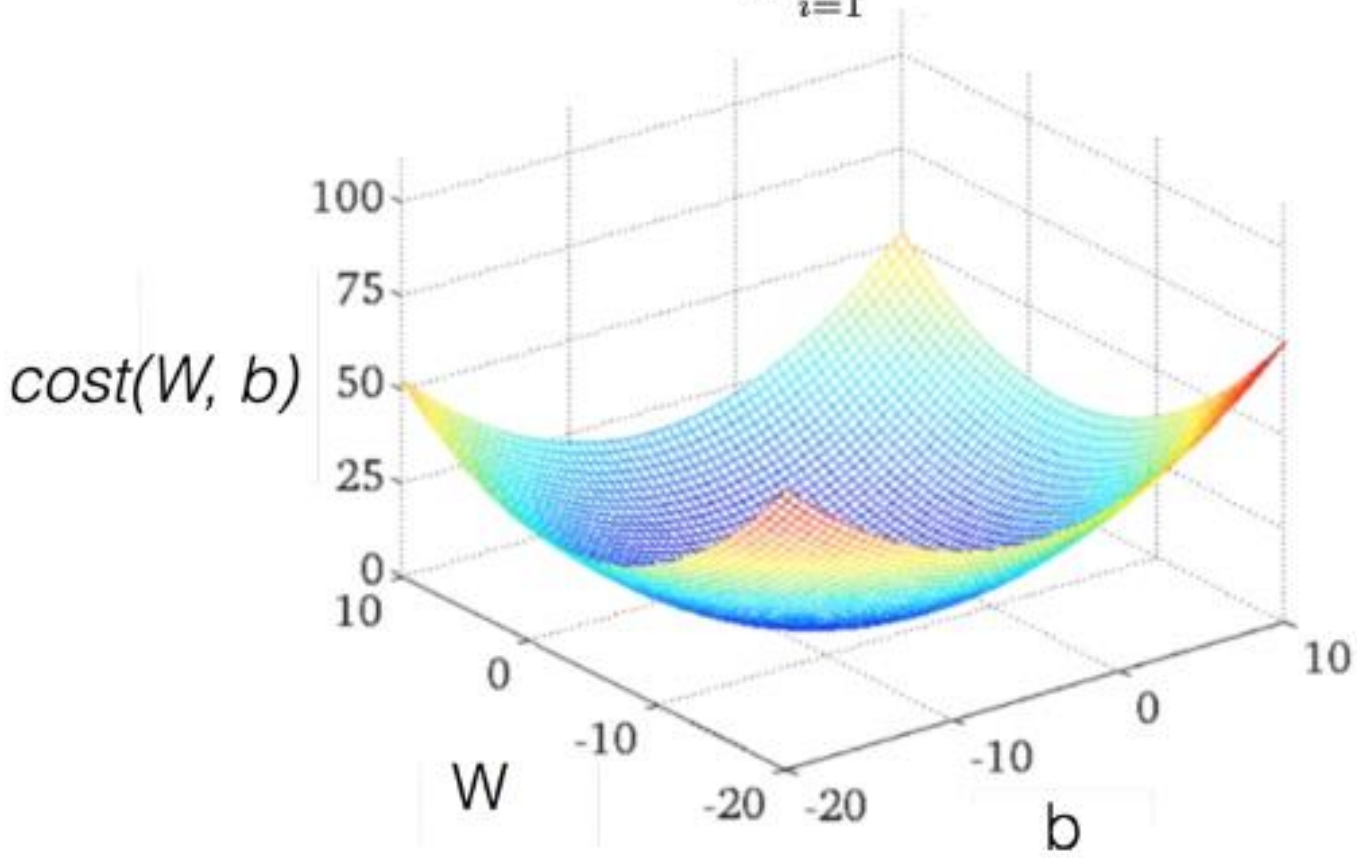
경사하강법

- **Gradient Descent Algorithm**
  - 임의의 곳에서 시작하여 경사도(gradient)에 따라 W를 변경시켜가면서 cost함수의 값이 최소화되는 W를 구하는 알고리즘
  - 경사도는 미분값
  - ∝는 학습률

$$W = W - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(W x^i - y^i \right) x^i$$



$$cost(W,b) = \frac{1}{m}\sum_{i=1}^{m}(H(x^{(i)}) - y^{(i)})^2$$



손실함수와 경사하강법(미분)

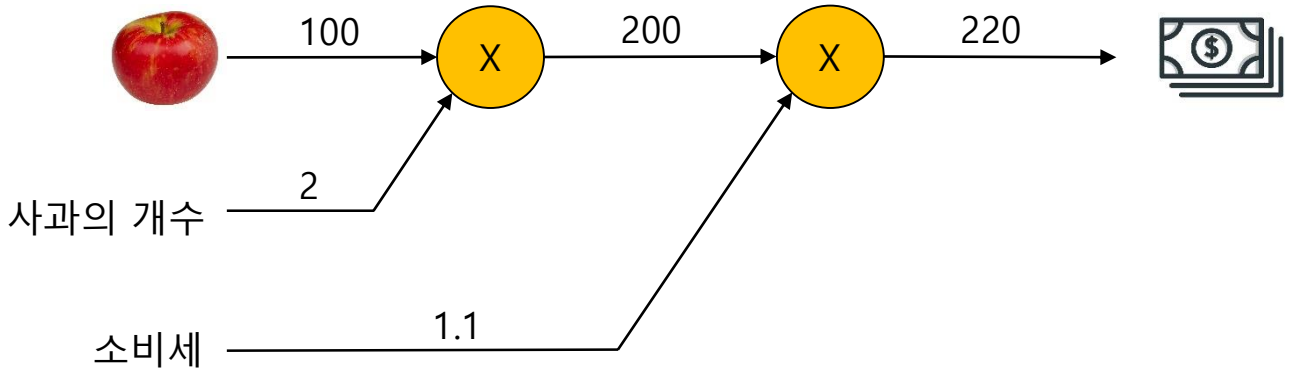
- **왜 손실함수를 사용할까?**
  - **정확성 : 학습이 얼마나 잘 되어있는가 하는 것**
  - **최적의 매개변수 탐색**
    - 매개변수의 미분을 계산하고 매개변수의 값을 서서히 갱신하는 과정을 반복(미분값이 0이면 중단)
  - **정확성을 지표로 사용하는 경우**
    - 정확성(%)은 계단함수와 같이 불연속적인 특성을 갖으므로 대부분의 장소에서 미분값이 0이 됨
  - **손실함수를 지표를 사용하는 경우**
    - MSE나 cross entropy와 같은 지표는 연속적인 특성을 가지고 있으며 기울기도 연속적으로 변하므로, 어느 곳에서도 미분값이 0이 되지는 않음
- **손실함수를 최소화하는 방향으로 서서히 매개변수들을 학습**
  - **손실함수를 최소화하는 방향을 찾기 위해 미분을 사용**

- **수치미분**
  - **경사하강법**
    - 손실함수를 최소화하는 값을 찾기 위해 사용
    - 경사하강법에서는 매번 수치미분(기울기)을 계산
      => 시간이 오래 걸림

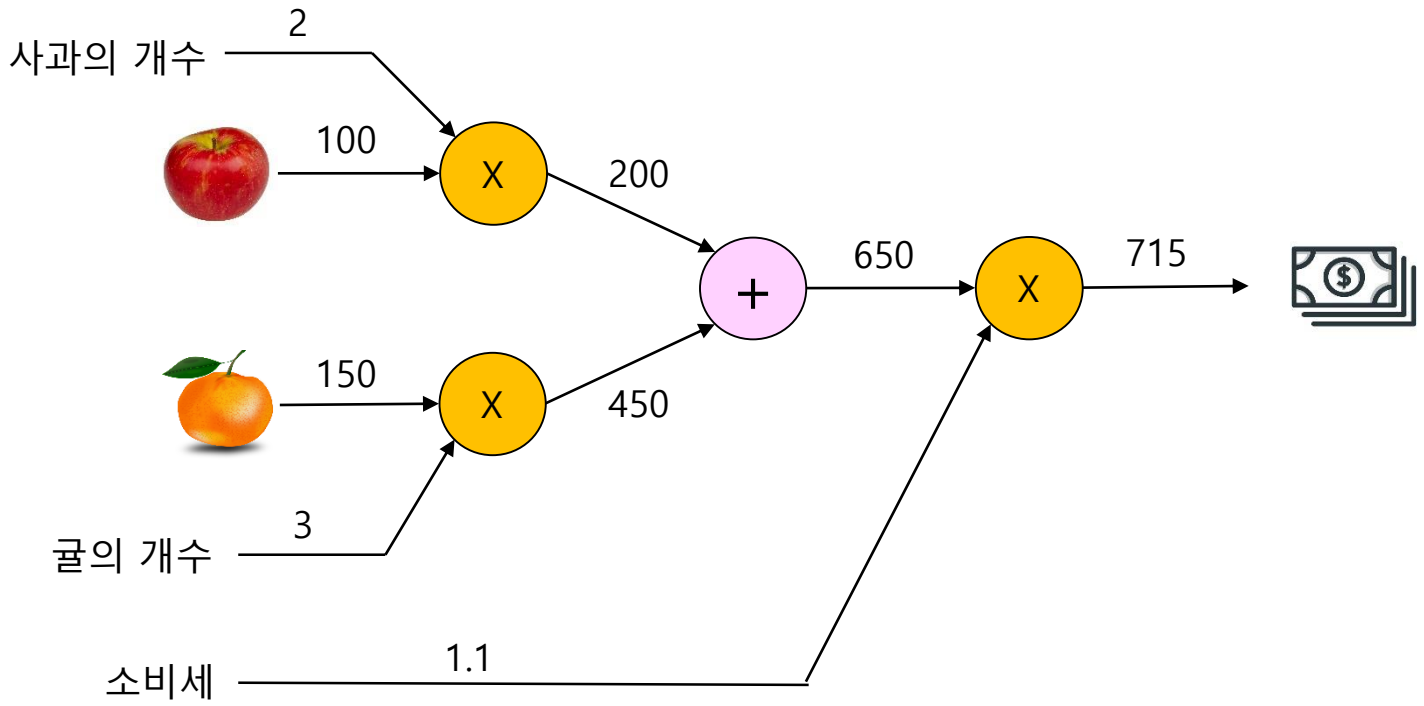
- **수치미분의 단점을 극복하기 위한 방법**
  - **오차역전파 방법**
    - 중간의 미분결과를 공유(활용)할 수 있으므로...
    - 매개변수가 많은 경우에도, 기울기를 효율적으로 계산
  - **순전파와 역전파를 이용하여 기울기를 효율적으로 계산**
  - **기본원리 : Chain rule**
    - '국소미분'을 전달하는 원리

Backpropagation의 이해

- 계산그래프(computational graph)
  - 계산과정을 그래프(graph)로 표현하여 해결하는 방법
  - (예) 1개에 100원인 사과를 2개 샀다. 지불금액은 얼마인가? 단, 소비세가 10% 부과된다.

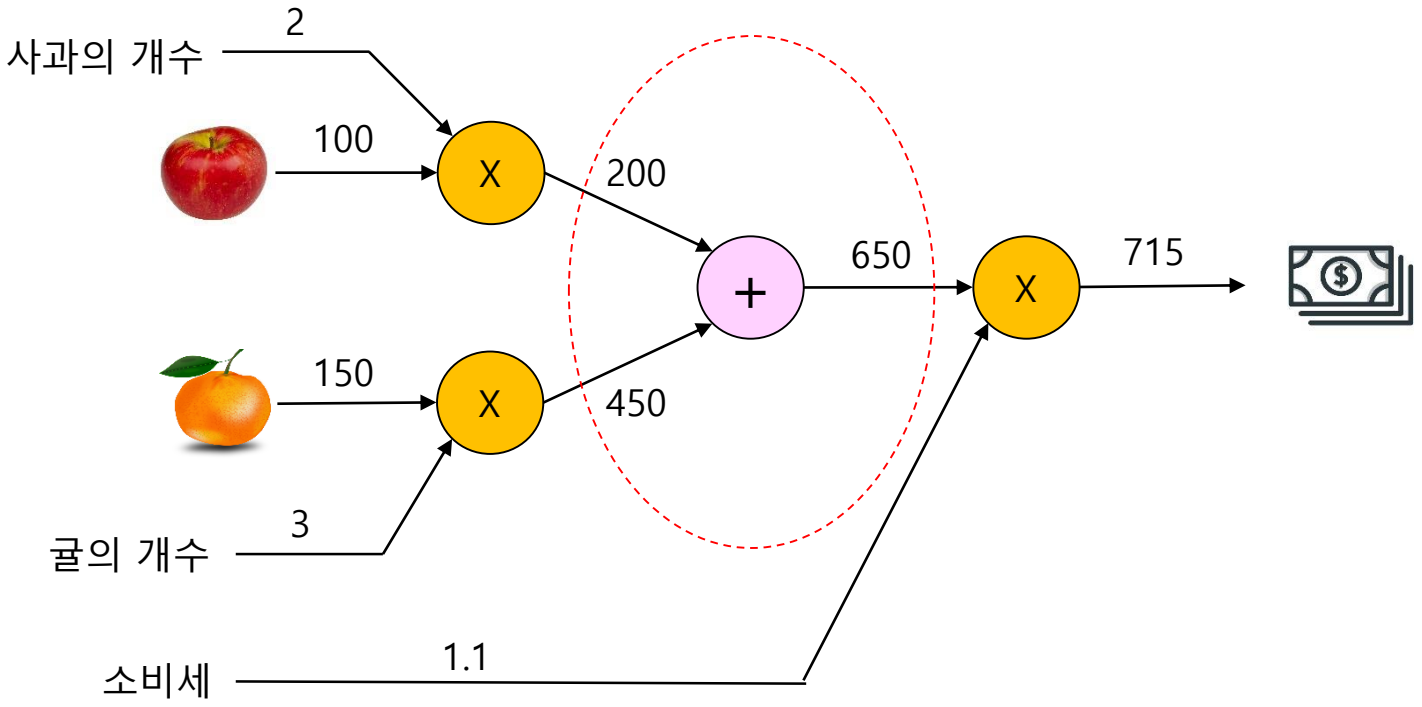


- (예) 사과를 2개, 귤은 3개 샀다. 사과는 1개에 100원, 귤은 1개에 150원이다. 지불금액은 얼마인가? 단, 소비세가 10% 부과된다.

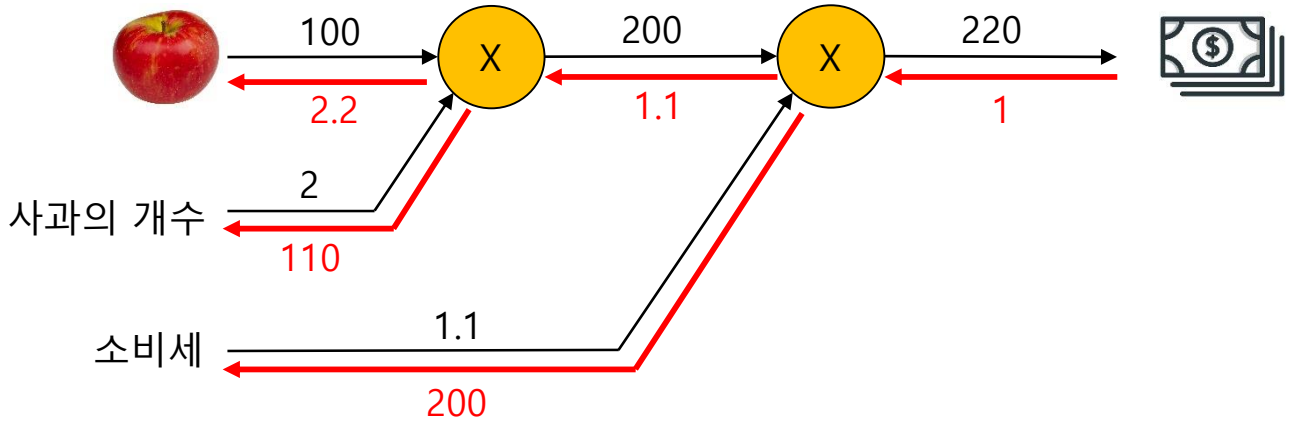


국소계산

- 각 노드의 계산은 자신과 관련된 계산만 수행
- 다른 노드의 복잡한 계산은 신경 쓸 필요가 없음



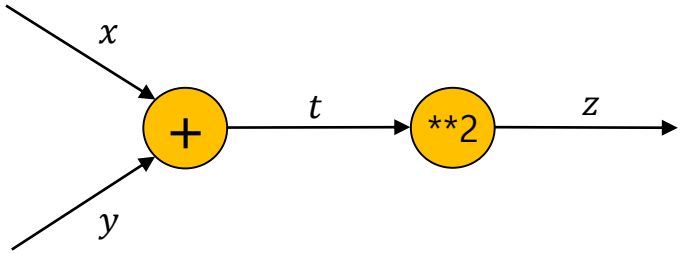
- 사과가격이 오르면 지불금액에는 어떠한 영향이 있을까?
  - 사과가격(x)에 대한 지불금액(L)의 미분($\frac{\partial L}{\partial x}$)을 구하는 문제
  - 역전파(backpropagation)를 통해 미분을 구할 수 있음
    - 사과가 1원 오르면, 지불금액은 2.2원 오른다.



Chain Rule

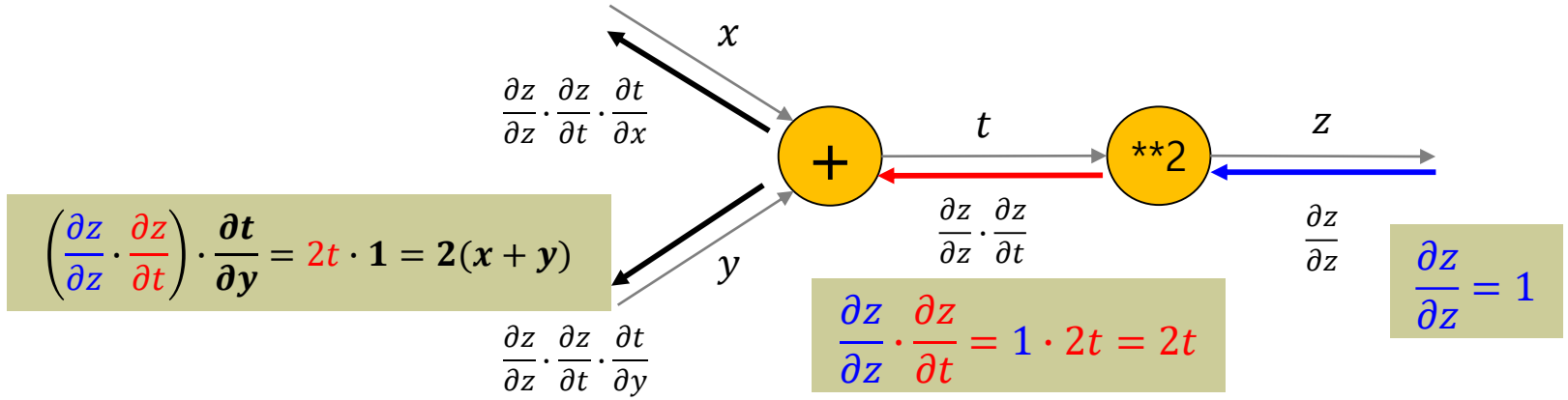
- 합성함수
  - $z = (x + y)^2$ $\qquad$ $z = t^2$ ⇒ $t = x + y$
- 연쇄법칙(chain rule)
  - 합성함수의 미분 => 함수를 구성하는 각 함수 미분의 곱
  - $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \cdot \frac{\partial t}{\partial x}$
- 함수 $z = (x + y)^2$의 미분
  - 합성함수로 표현하면 $z = t^2$ $\qquad$ $t = x + y$
  - 따라서,
    - $\frac{\partial z}{\partial t} = 2t$ $\qquad$ $\frac{\partial t}{\partial x} = 1$
    - $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \cdot \frac{\partial t}{\partial x} = 2t \cdot 1 = 2(x + y)$

- 함수 $z = (x + y)^2$의 계산그래프
  - $z = t^2$ $\quad t = x + y$

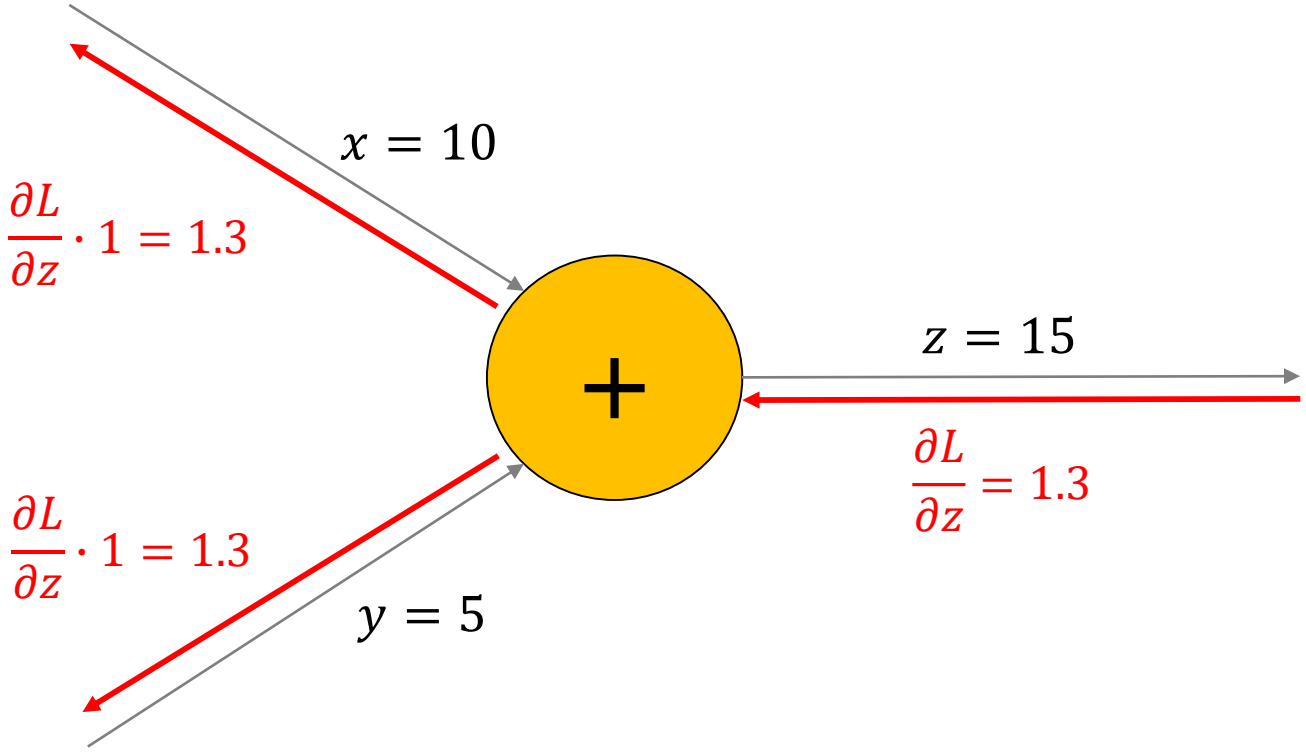


- 역전파를 이용한 미분계산

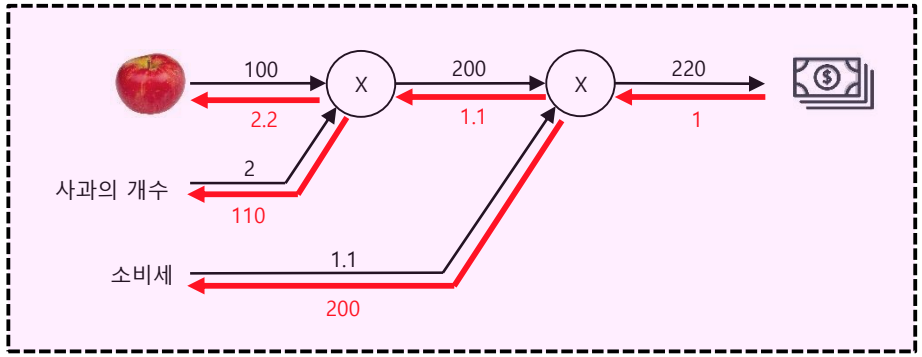
$$\left(\frac{\partial z}{\partial z}\cdot\frac{\partial z}{\partial t}\right)\cdot\frac{\partial t}{\partial x} = 2t\cdot 1 = 2(x+y)$$

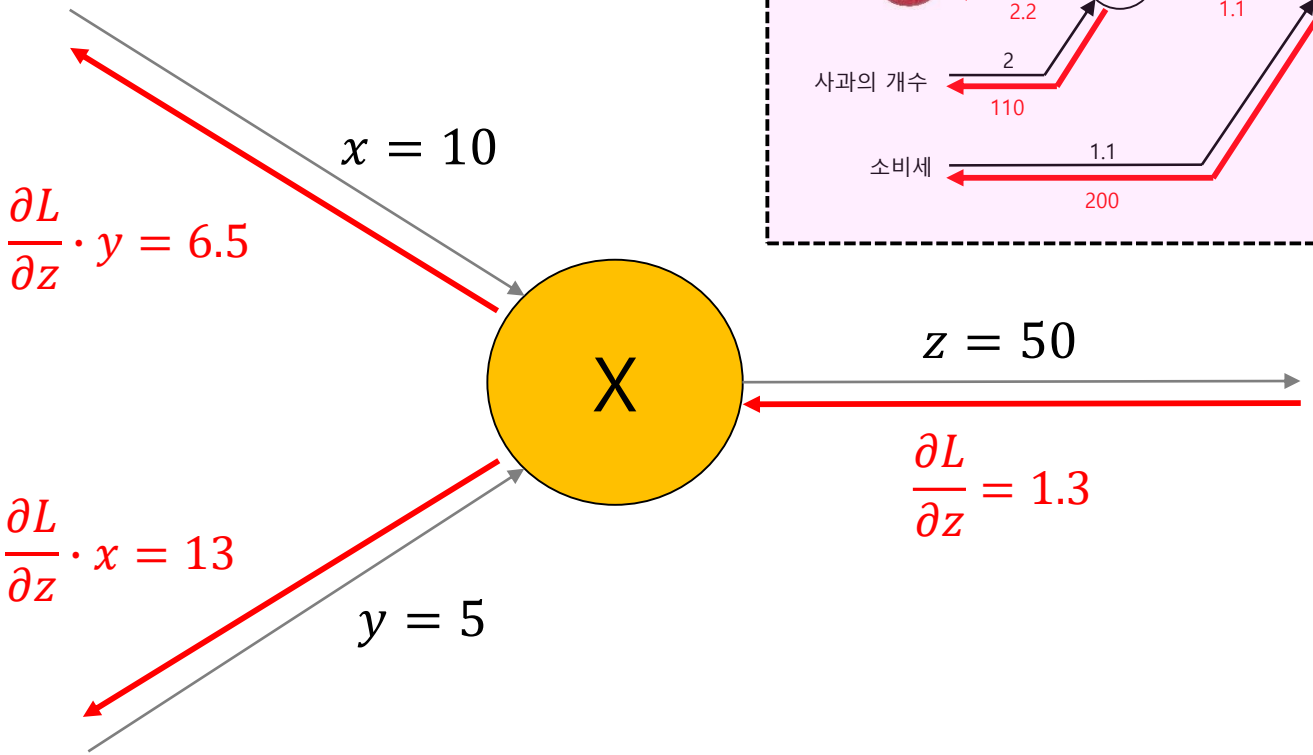


- 덧셈노드의 역전파
  - 이전의 미분값을 그대로 흘려보낸다.
  - 따라서, 순방향 입력신호의 값이 불필요



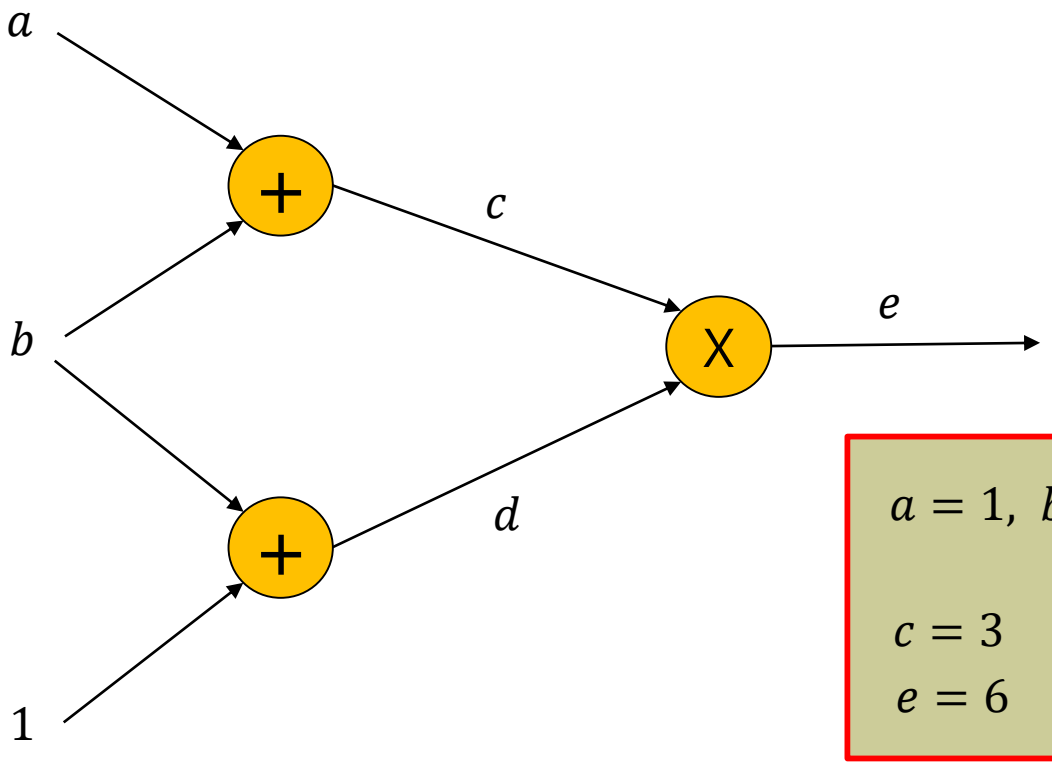
- 곱셈노드의 역전파
  - 입력신호를 바꾼 값을 이전의 미분값에 곱한다.
  - 따라서, 순방향 입력신호의 값이 필요





또 다른 예를 보자!

- 함수 $e = (a + b)(b + 1)$의 계산 그래프
  - $c = a + b$ $\quad d = b + 1$ 이면, $\quad e = c \cdot d$



$a = 1,\ b = 2$ 인 경우

$c = 3 \quad d = 2$

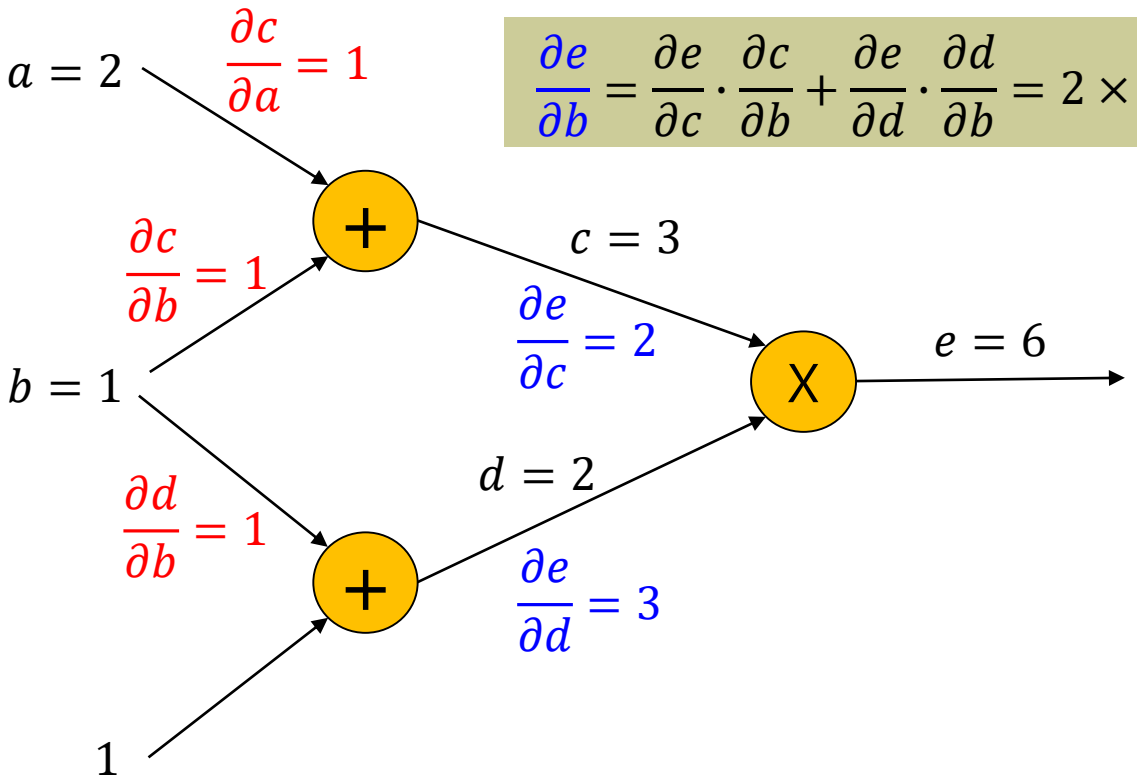
$e = 6$

계산 그래프의 순방향 미분

- 모든 노드에 대한 e의 미분은 별도로 계산해야 함
- 만일, 노드가 100만개라면?

$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial a} = 2 \times 1 = 2$$

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 2 \times 1 + 3 \times 1 = 5$$



계산 그래프의 역방향 미분

- 모든 노드에 대한 e의 미분을 '국소미분'으로 전달
- 따라서, 매우 효율적인 방법임