인공지능

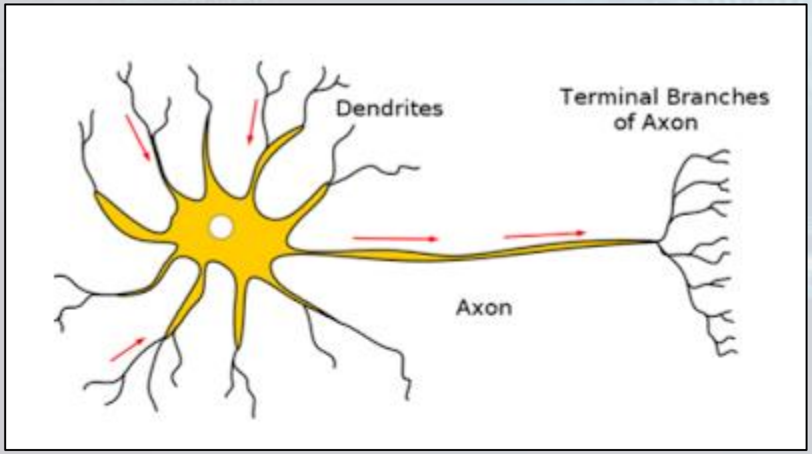
딥러닝 기초_퍼셉트론_활성화함수

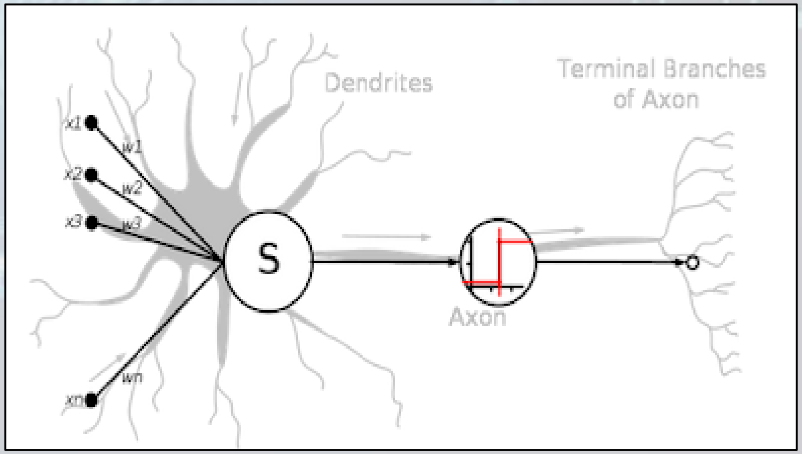
강성관 (silicon1@hanmail.net)

딥러닝 기초

퍼셉트론 (Perceptron)

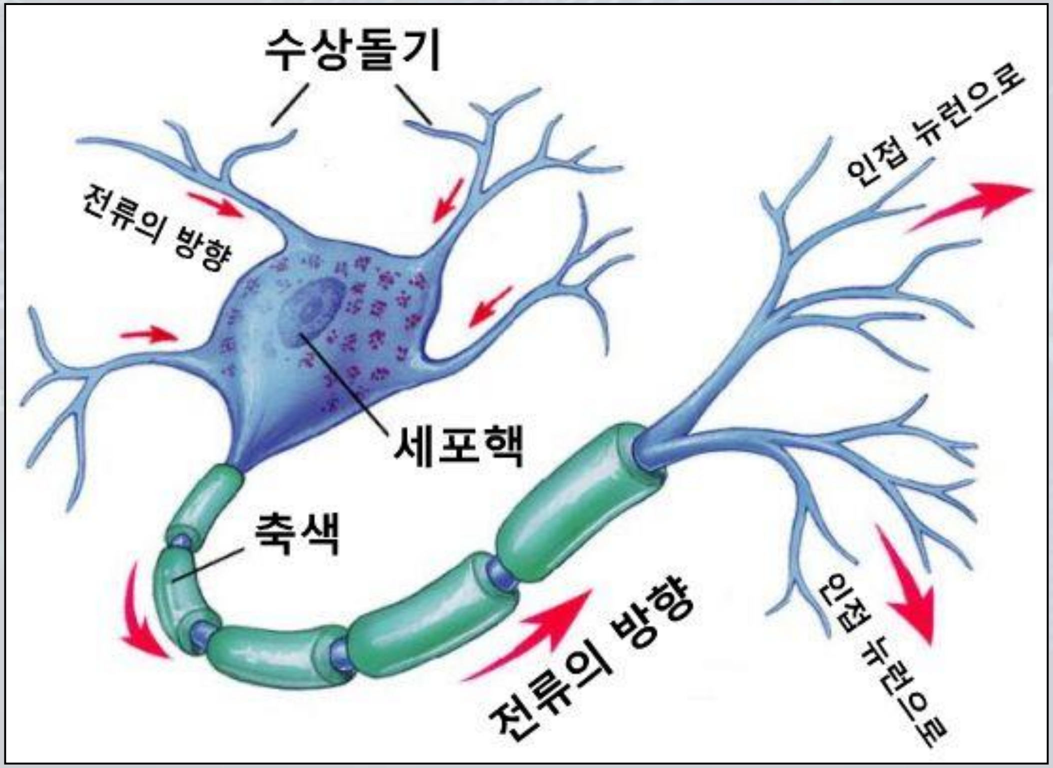
- 1957년 프랑크 로젠블라트에 의해 고안
- 대뇌피질의 신경세포는 **수상돌기(dendrite)**를 통해 다른 신경세포로부터 입력 신호를 받음 → **신경세포체(cell body)**에서 정보 연산을 처리 → **축삭(axon)**을 통해 처리된 정보를 다른 신경세포로 전달
- 이러한 신경세포는 **시냅스**라는 가중 연결자(weighted connector)로 여러 층의 신경망을 구성
- 세포체에서의 연산 : 입력 신호들의 합을 구함 → 그것이 일정한 **임계치가 되면 축삭이 활성화되어 스파크**를 일으킴 → 다른 신경세포로 전기 신호를 전달
- 이러한 특성을 모방하여 추상화한 것이 **인공 신경망(ANN : Artificial Neural Network)**





퍼셉트론 (Perceptron)

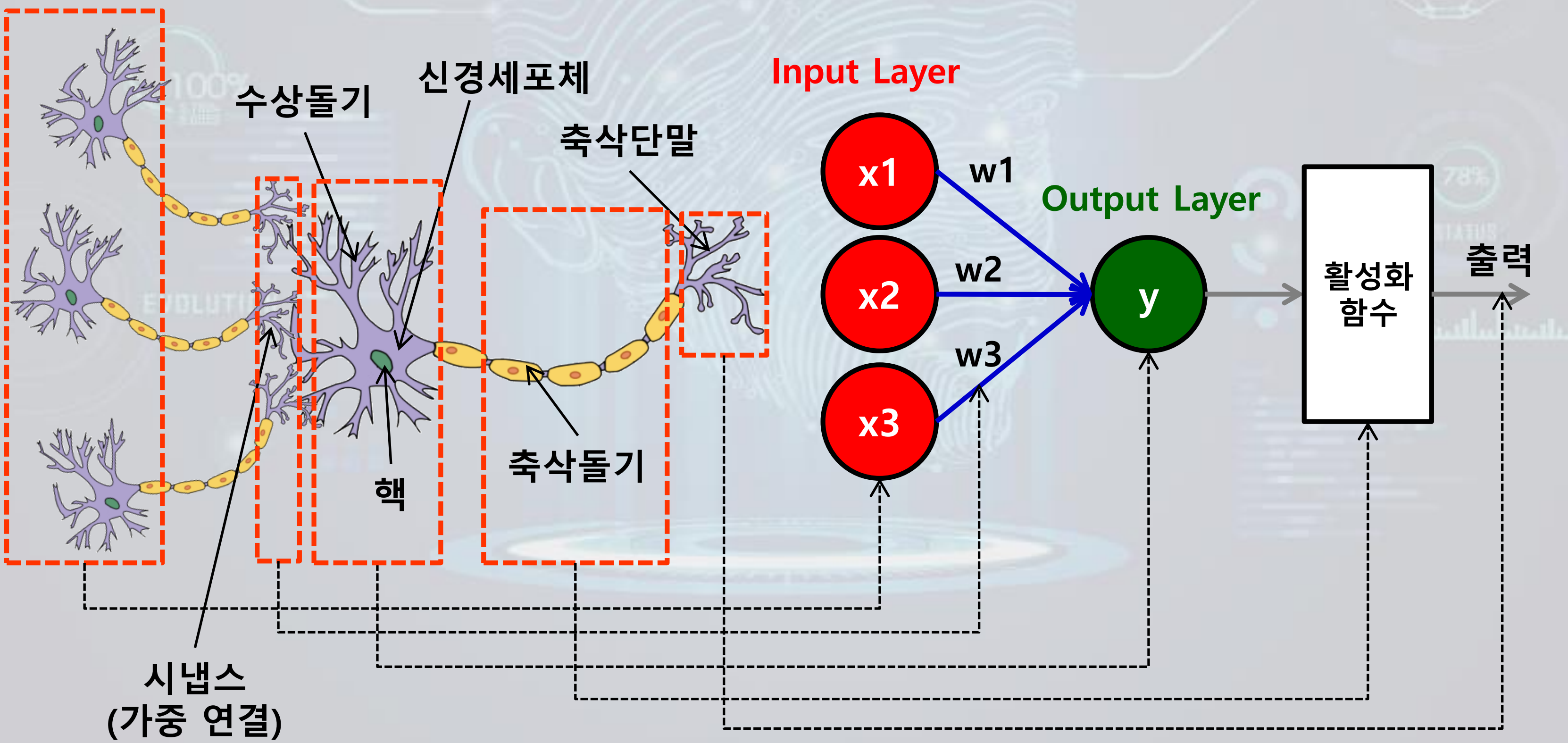
- 뉴런의 구조 : 성인의 뇌는 **850억 개의 뉴런**과 **10^{14}~10^{15}개의 시냅스**로 이루어짐



http://m.blog.daum.net/convoy1/18353942

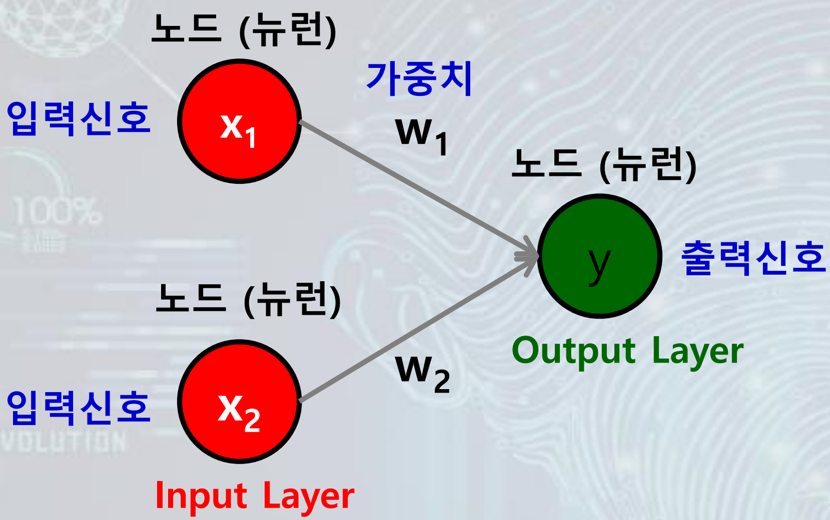
퍼셉트론 (Perceptron)

- **추상화** : 현실 세계를 가상 세계로 모델링하는 작업
- **신경망의 추상화** : 인간의 **뉴런**을 하나를 **노드** (인공 뉴런)으로 가상화하고 각 **노드의 특성** (가중치)를 **다르게 설정**하여 동일한 입력에 대해 다양한 반응을 발생하도록 하게 함.



퍼셉트론 (Perceptron)

- 퍼셉트론은 다수의 신호를 입력받아 하나의 신호를 출력



$$y = w_1 x_1 + w_2 x_2$$

$$y = \sum_{i=0}^{n} w_i x_i$$

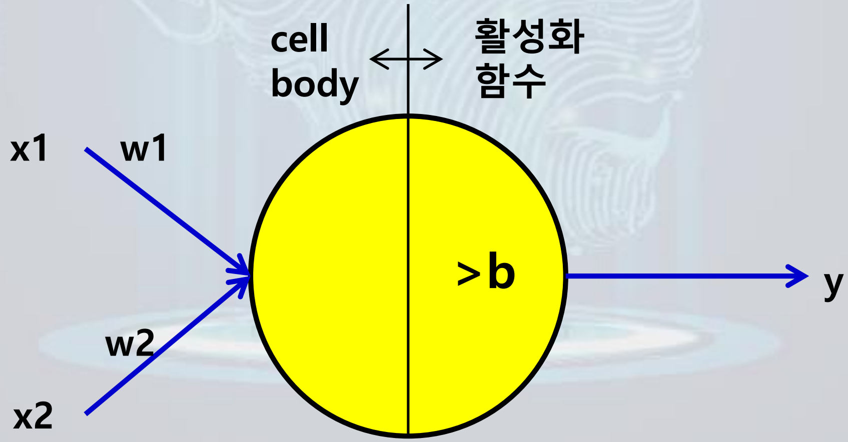
- 입력 신호가 뉴런에 보내질 때 각각의 고유한 **가중치 w_1, w_2가 곱해지고 각 입력은 더해짐**

퍼셉트론 (Perceptron)

- 뉴런에서 보내온 신호의 총합이 정해진 임계값(b)을 넘어설 때만 1을 출력 (뉴런 활성화)

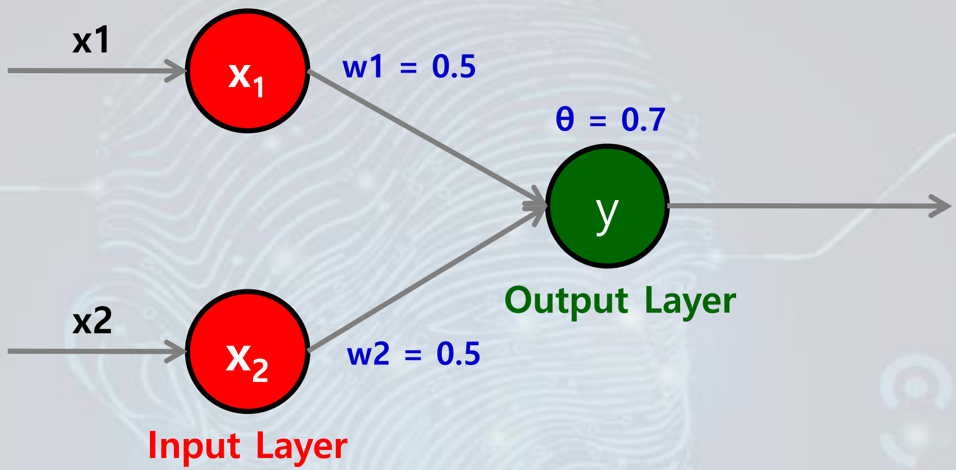
$$y = \begin{cases} 0 & (w_1x_1 + w_2x_2 \le b) \\ 1 & (w_1x_1 + w_2x_2 > b) \end{cases}$$

- **가중치가 클수록 해당 신호가 결과에 영향을 크게 준다는 것을 의미**



퍼셉트론 (Perceptron)

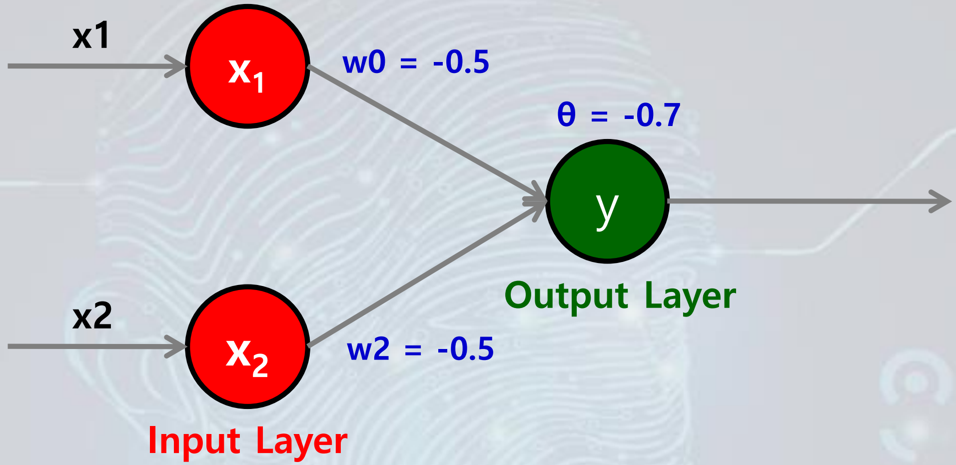
- AND 게이트



x1	x2	AND	Input	θ	y
0	0	0	(0 x 0.5) + (0 x 0.5) = 0	0.7	**0**
0	1	0	(0 x 0.5) + (1 x 0.5) = 0.5		**0**
1	0	0	(1 x 0.5) + (0 x 0.5) = 0.5		**0**
1	1	1	(1 x 0.5) + (1 x 0.5) = 1.0		**1**

퍼셉트론 (Perceptron)

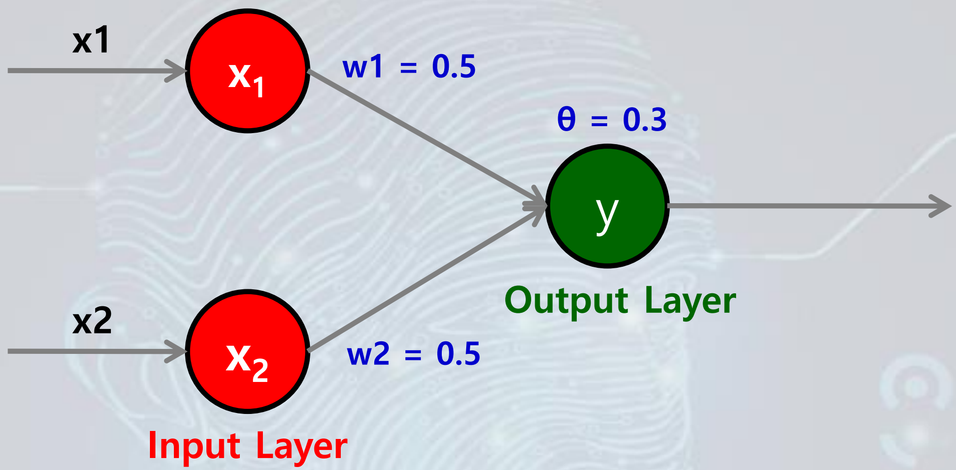
- NAND 게이트



x1	x2	NAND	Input	θ	y
0	0	1	(0 x -0.5) + (0 x -0.5) = 0	-0.7	**1**
0	1	1	(0 x -0.5) + (1 x -0.5) = -0.5		**1**
1	0	1	(1 x -0.5) + (0 x -0.5) = -0.5		**1**
1	1	0	(1 x -0.5) + (1 x -0.5) = -1.0		**0**

퍼셉트론 (Perceptron)

- OR 게이트



x1	x2	OR	Input	θ	y
0	0	0	(0 x 0.5) + (0 x 0.5) = 0	0.3	**0**
0	1	1	(0 x 0.5) + (1 x 0.5) = 0.5		**1**
1	0	1	(1 x 0.5) + (0 x 0.5) = 0.5		**1**
1	1	1	(1 x 0.5) + (1 x 0.5) = 1.0		**1**

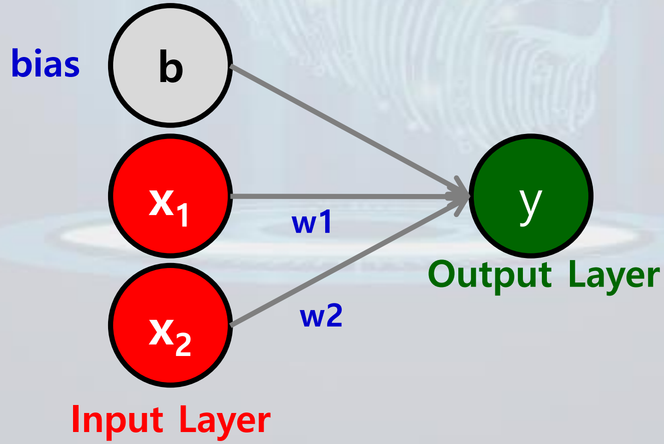
편향

- 이전 식에서 b을 좌변으로 이항시키면 (**-b를 b로 표현하기도 함**)

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

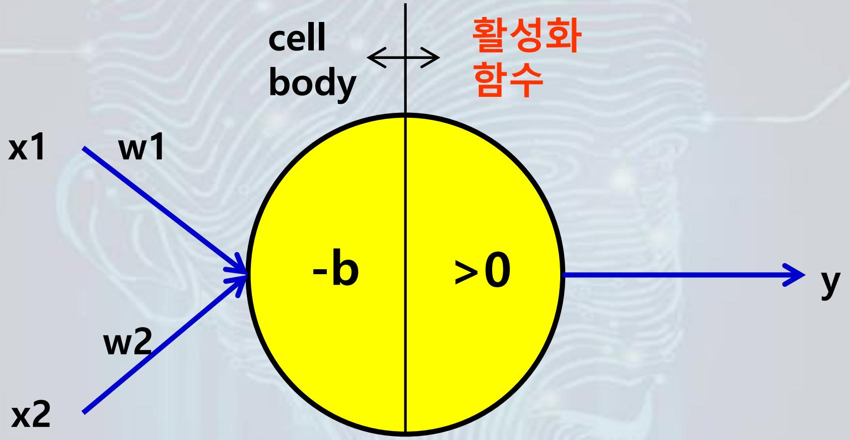
(b : 편향 (bias), w_1, w_2 : 가중치 (weight))

- **가중치 (weight)** : 각 입력 신호가 결과에 주는 **영향력** (중요도)을 조절하는 매개변수
- **편향 (bias)** : 뉴런이 얼마나 쉽게 **활성화** (결과로 1을 출력)하느냐를 조절하는 매개변수



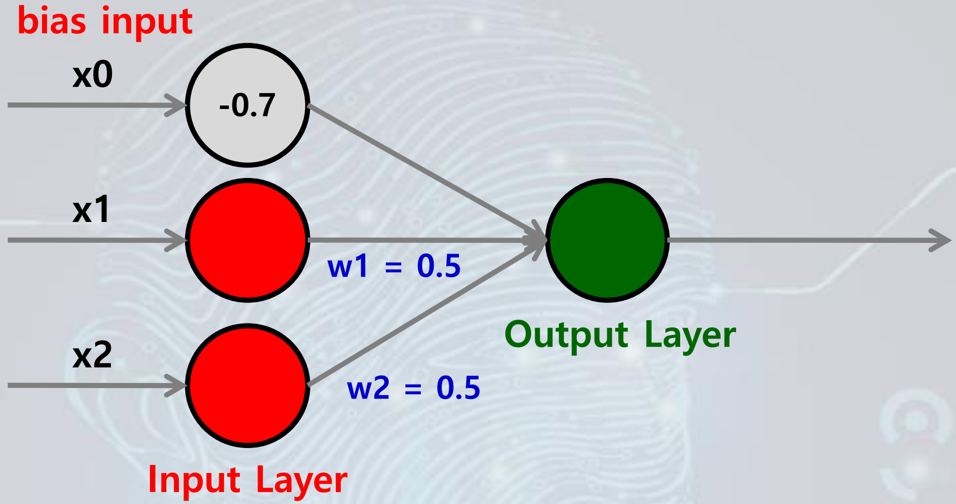
편향

- 편향이 추가된 퍼셉트론 모델



퍼셉트론 (Perceptron)

- AND 게이트



x1	x2	AND	Input	Output
0	0	0	-0.7 + (0 x 0.5) + (0 x 0.5) = -0.7	class 0
0	1	0	-0.7 + (0 x 0.5) + (1 x 0.5) = -0.2	class 0
1	0	0	-0.7 + (1 x 0.5) + (0 x 0.5) = -0.2	class 0
1	1	1	-0.7 + (1 x 0.5) + (1 x 0.5) = 0.3	class 1

실습 (Numpy 라이브러리 사용)

- AND 게이트

```python
import numpy as np

def AND(x1, x2):
    x = np.array([x1, x2])        #입력
    w = np.array([0.5, 0.5])      #가중치
    b = -0.7                      #편향

    tmp = np.sum(w*x) + b

    if tmp <= 0:
        return 0
    else:
        return 1

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

```
0
0
0
1
```

실습 (Numpy 라이브러리 사용)

- AND 게이트

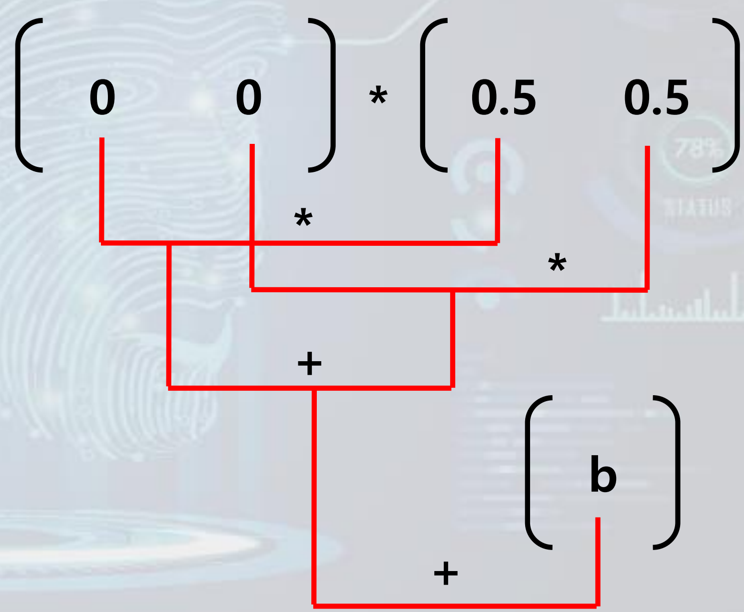
x = np.array([0, 0])

$$x \begin{pmatrix} 0 & 0 \end{pmatrix}$$

w = np.array([0.5, 0.5])

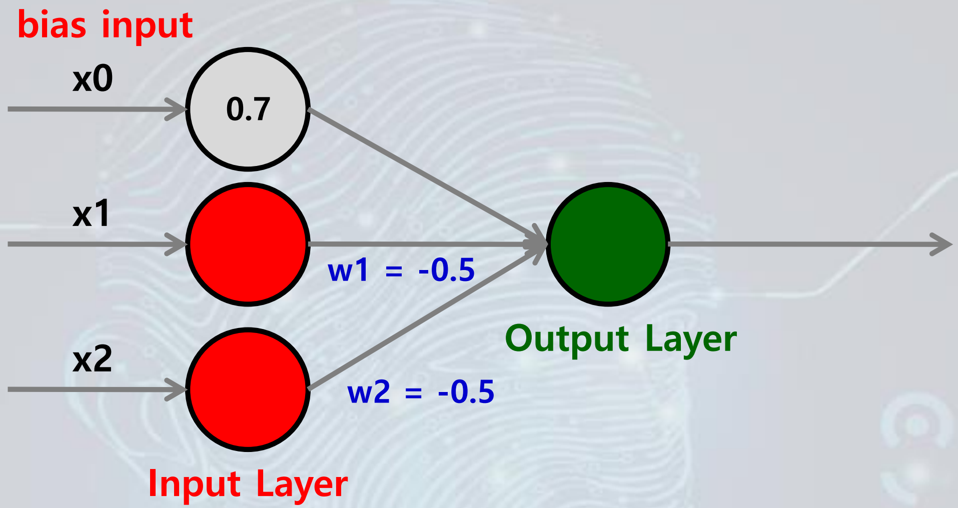
$$w \begin{pmatrix} 0.5 & 0.5 \end{pmatrix}$$

tmp = np.sum(w*x) + b



퍼셉트론 (Perceptron)

- NAND 게이트



x1	x2	NAND	Input	Output
0	0	1	0.7 + (0 x -0.5) + (0 x -0.5) = 0.7	class 1
0	1	1	0.7 + (0 x -0.5) + (1 x -0.5) = 0.2	class 1
1	0	1	0.7 + (1 x -0.5) + (0 x -0.5) = 0.2	class 1
1	1	0	0.7 + (1 x -0.5) + (1 x -0.5) = -0.3	class 0

실습 (Numpy 라이브러리 사용)

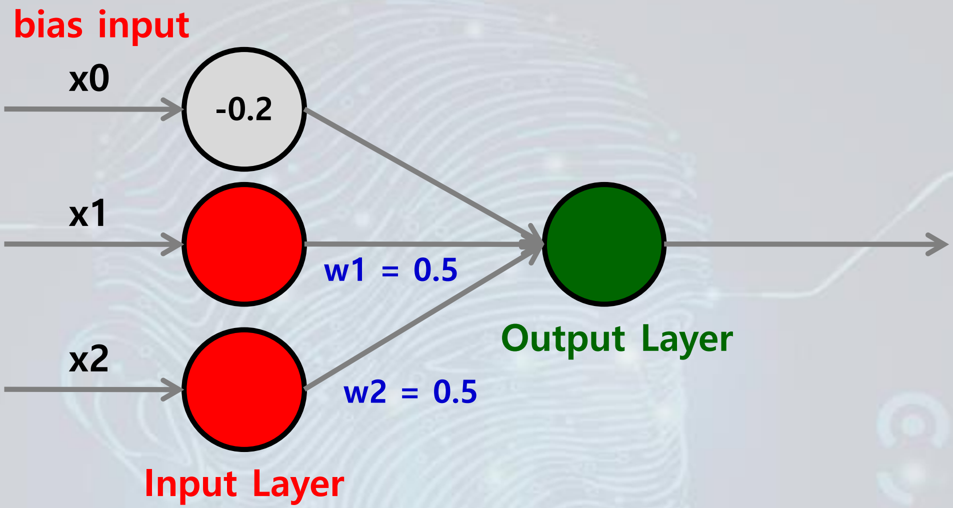
- NAND 게이트

```python
import numpy as np

def NAND(x1, x2):
    x = np.array([x1, x2])        #입력
    w = np.array([-0.5, -0.5])   #가중치
    b = 0.7                       #편향

    tmp = np.sum(w*x) + b

    if tmp <= 0:
        return 0
    else:
        return 1

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

```
1
1
1
0
```

퍼셉트론 (Perceptron)

- OR 게이트



x1	x2	OR	Input	Output
0	0	0	-0.2 + (0 x 0.5) + (0 x 0.5) = -0.2	class 0
0	1	1	-0.2 + (0 x 0.5) + (1 x 0.5) = 0.3	class 1
1	0	1	-0.2 + (1 x 0.5) + (0 x 0.5) = 0.3	class 1
1	1	1	-0.2 + (1 x 0.5) + (1 x 0.5) = 0.8	class 1

실습 (Numpy 라이브러리 사용)

- OR 게이트

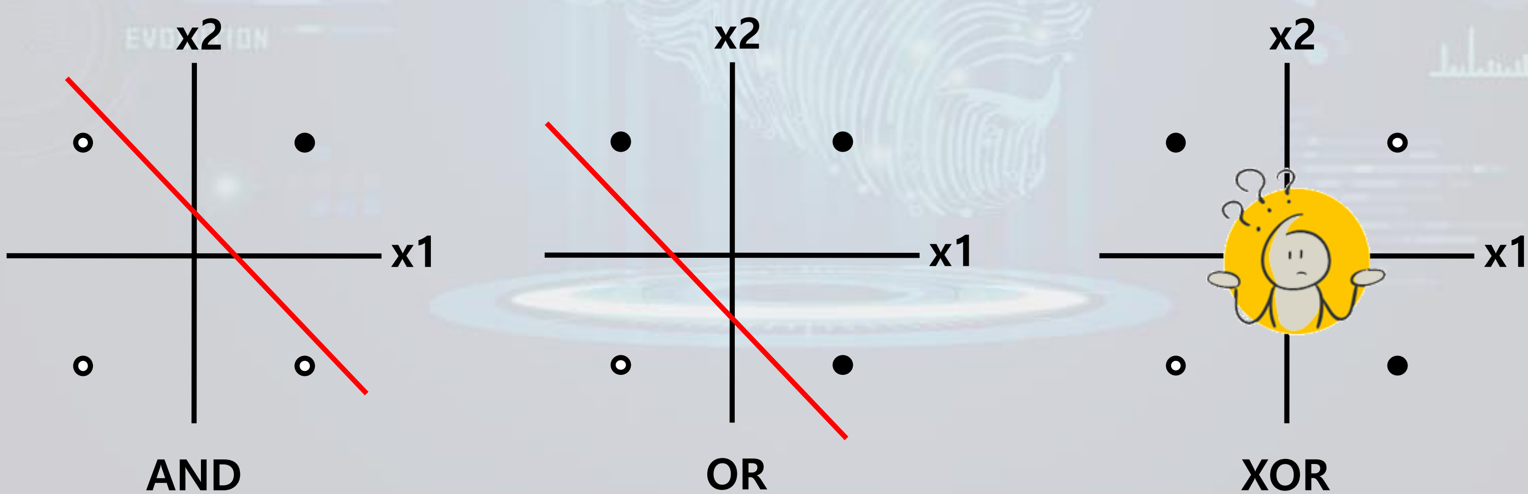
```
import numpy as np

def OR(x1, x2):
    x = np.array([x1, x2])         #입력
    w = np.array([0.5, 0.5])       #가중치
    b = -0.2                       #편향

    tmp = np.sum(w*x) + b

    if tmp <= 0:
        return 0
    else:
        return 1

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

```
0
1
1
1
```

퍼셉트론 (Perceptron)의 한계

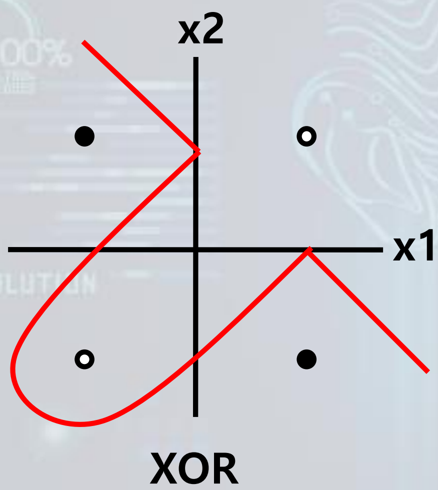
- 한 개의 직선으로만 분류를 하므로 XOR의 경우는 분류가 불가능하다는 점 → 선형 영역에서만 사용 가능

x1	x2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

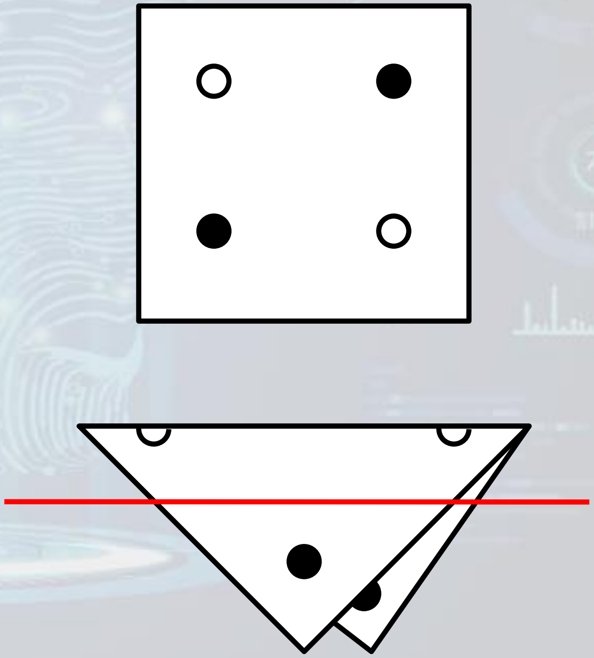


퍼셉트론 (Perceptron)의 한계

- XOR의 경우 비선형 영역을 포함하고 있으므로 퍼셉트론으로는 표현할 수 없음 → **다층 퍼셉트론 (Multi-layer perceptron)**

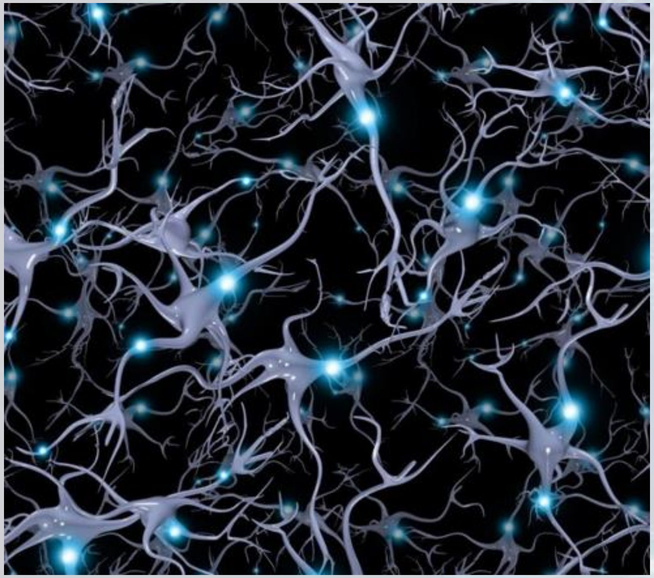


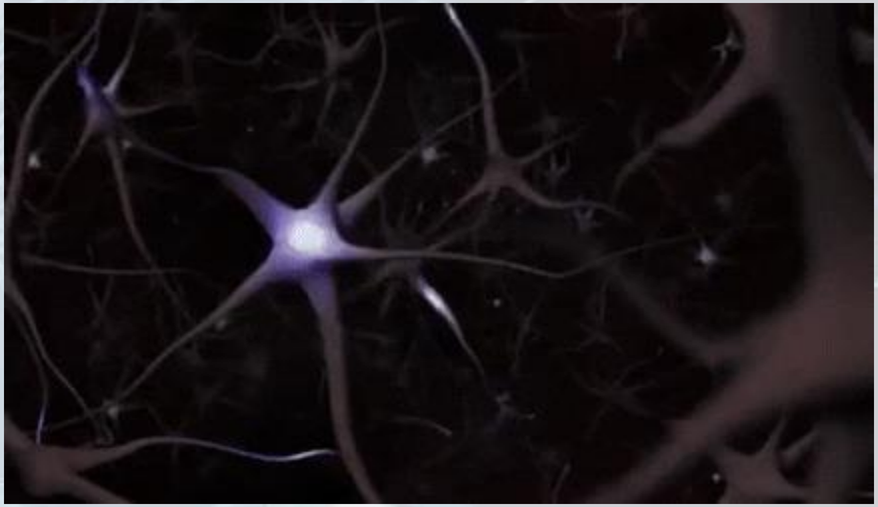
한 선으로 분류하려면 ?



다층 퍼셉트론 (MLP : Multi Layer Perceptron)

- 신경망은 뇌의 신경세포 (뉴런)들이 시냅스로 연결되어 전기 신호를 통해 정보를 주고 받는 모습에서 착안한 것으로 **다층 퍼셉트론** (Multi-layer Perceptron)으로 부름



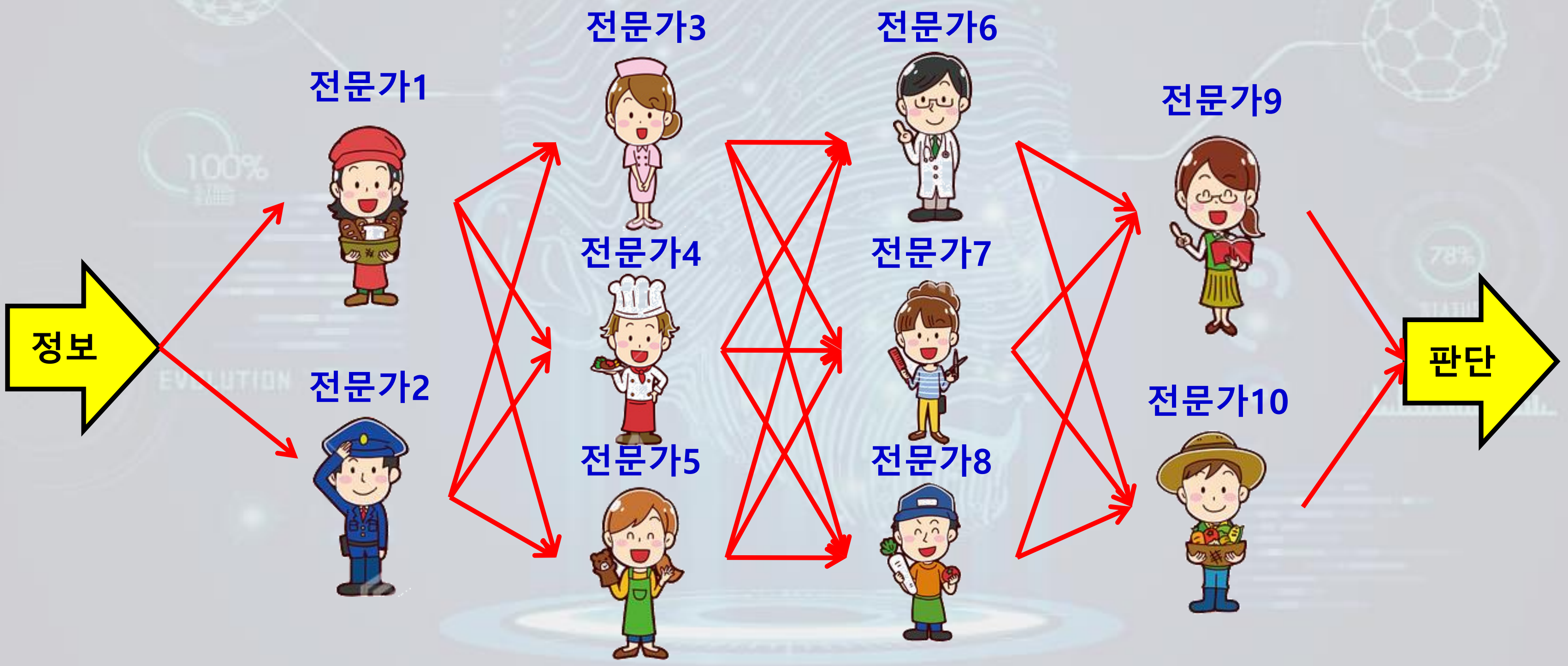


다층 퍼셉트론 (MLP : Multi Layer Perceptron)

- 사람은 대상이 무엇인지 판단하는 정보들의 경계를 느슨하게 가지고 있음 → 정확하지 않음 → 추상적
- 각 신경 세포 (뉴런)은 정보에 대한 각기 다른 기준을 가지고 있음
- 다층 신경망을 거치면서 정보를 판단하게 됨

다층 퍼셉트론 (MLP : Multi Layer Perceptron)

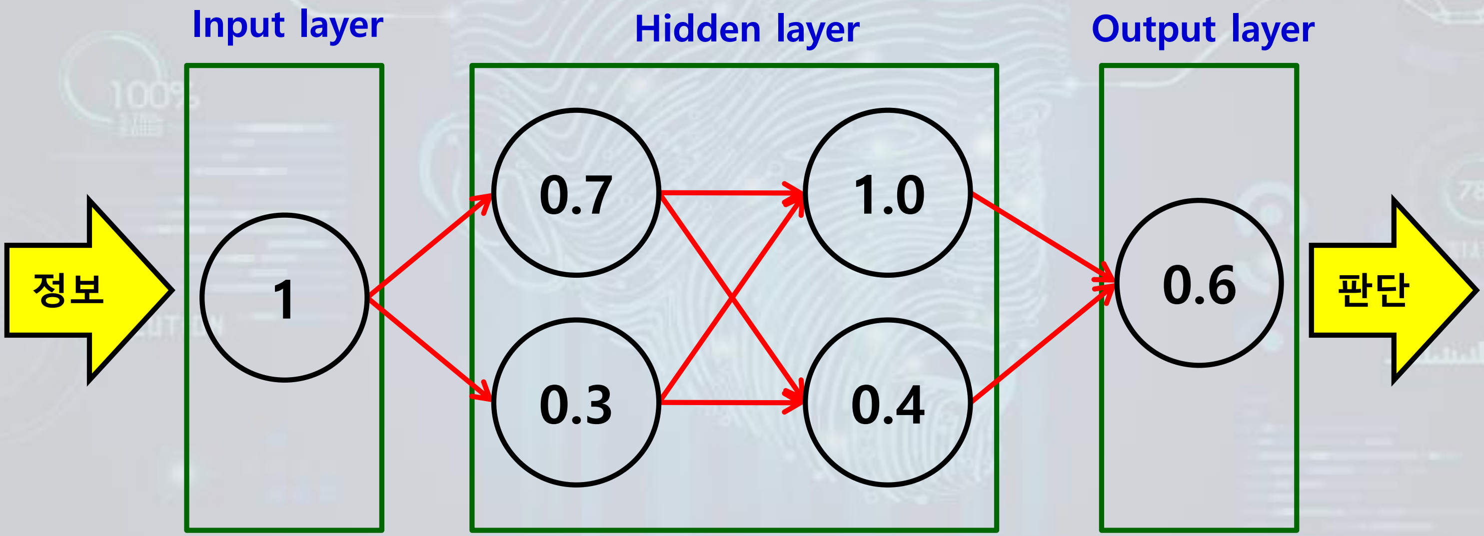
- 다른 관점을 가진 전문가를 활용한 판단



전문가 네트워크

다층 퍼셉트론 (MLP : Multi Layer Perceptron)

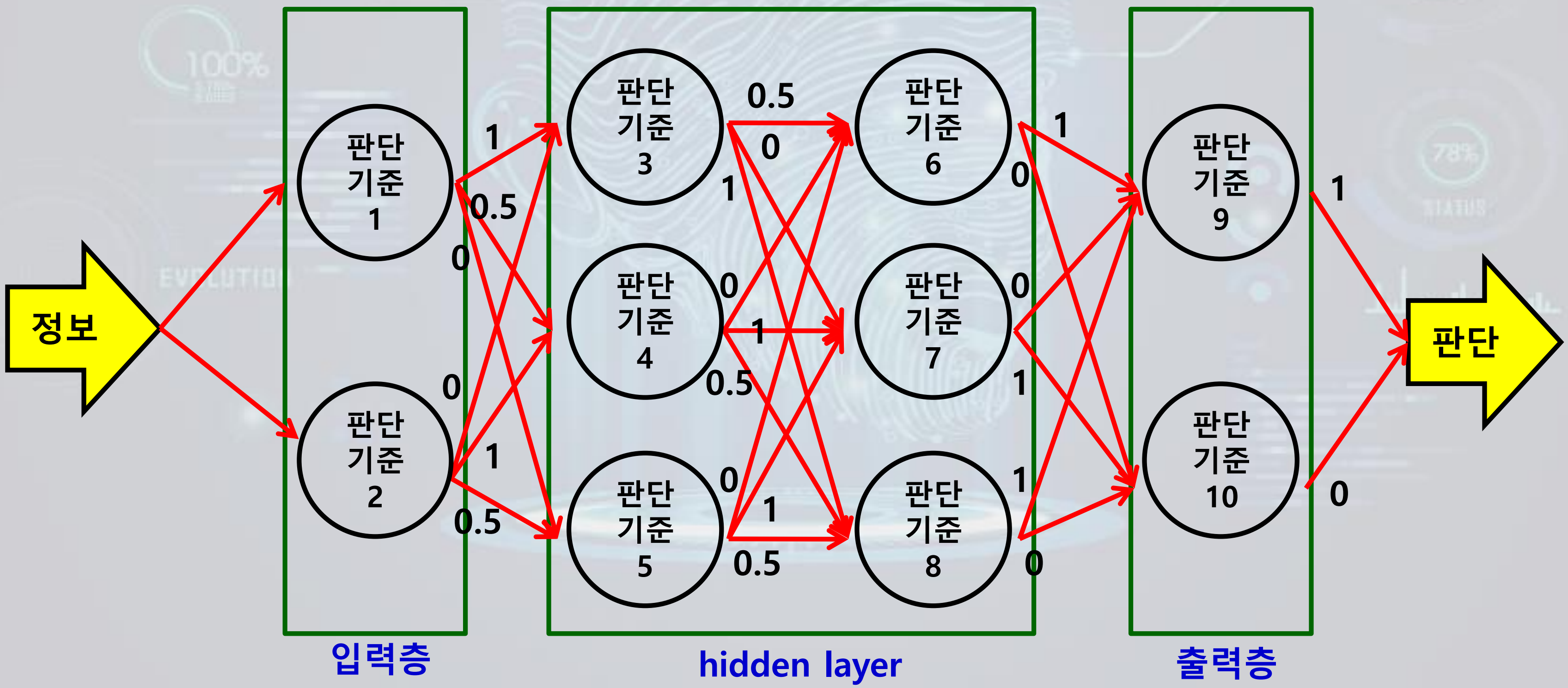
- 하나의 입력에 대해 뉴런들은 다른 판단을 하고 다음 층으로 전달



https://www.youtube.com/watch?v=XdP7kkEDkfM

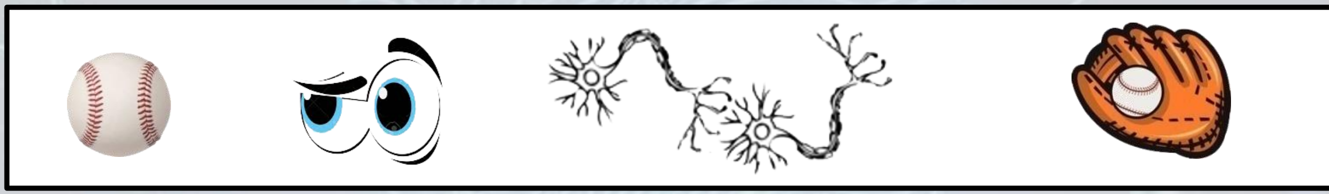
다층 퍼셉트론 (MLP : Multi Layer Perceptron)

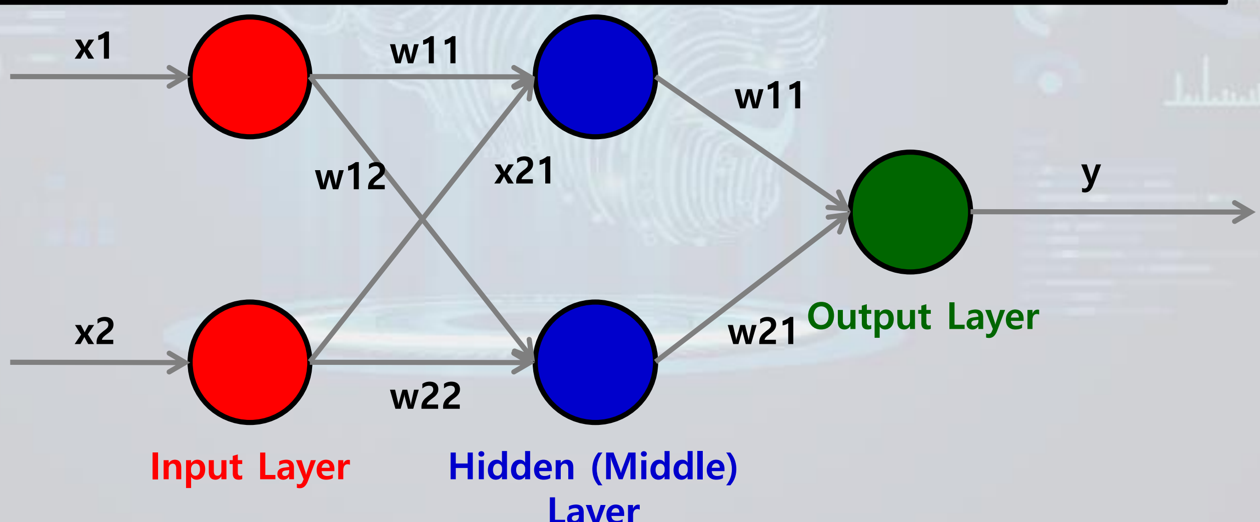
- 신경망을 기초로 모델링 한 것
- hidden layer가 많을수록 (deep) 성능 향상 → Deep Learning
- 오차 증가 가능성 높아짐 → 오차감소 알고리즘 필요



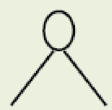
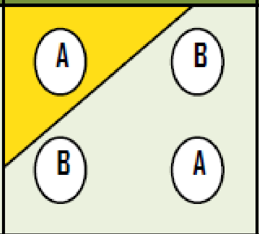
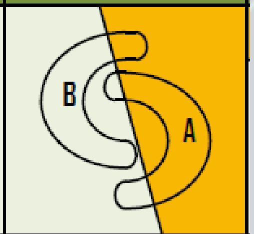
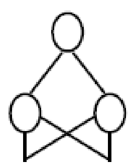
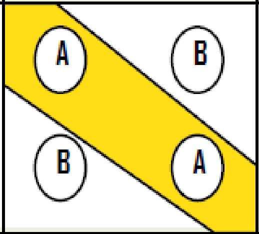
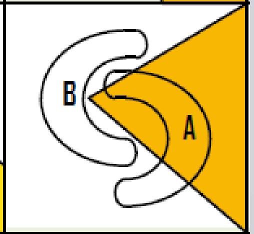
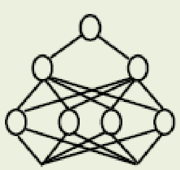
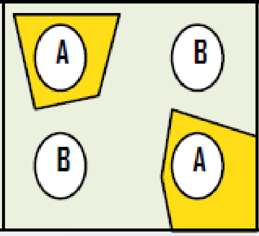
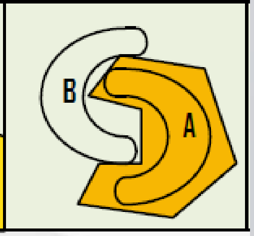
다층 퍼셉트론 (MLP : Multi Layer Perceptron)

- 특징
  - 비선형 데이터를 분리할 수 있다.
  - 학습 시간이 **오래 걸리고** 파라미터 수가 많으므로 **과대적합**을 일으키기 쉬우며 가중치가 초기값에 민감하여 **지역 최적점에 빠지기 쉽다**





다층 퍼셉트론 (MLP : Multi Layer Perceptron)

구조	결정 영역 형태	Exclusive-OR 문제	얽힌 결정 영역을 갖는 클래스들
단층 	초평면에 의해 나뉘어지는 반평면 (Hyper plane)		
두 개층 	볼록한 모양 또는 닫힌 영역		
세 개층 	임의의 형태 (노드들의 개수에 따라 복잡도가 결정됨)		

다층 퍼셉트론 (MLP : Multi Layer Perceptron)

- 추상적인 판단이 필요한 분야
  - (1) 음성인식, 영상인식, 자연어 처리 (챗봇), 가상실험 등 (일정하게 정해진 값이 존재하지 않는 데이터 → 같은 사람의 목소리라도 환경에 따라 달라짐)
  - (2) 긴 학습 시간이 필요한 데이터
  - (3) 학습 예제에 에러가 존재하는 경우
  - (4) 여러 속성에 의해 표현되는 데이터 등

다층 퍼셉트론 (MLP : Multi Layer Perceptron)

- XOR를 AND, NAND, OR 게이트로 표현하기

x1	x2	NAND
0	0	1
0	1	1
1	0	1
1	1	0

x1	x2	OR
0	0	0
0	1	1
1	0	1
1	1	1

S1	S2	y
NAND	OR	AND
1	0	0
1	1	1
1	1	1
0	1	0

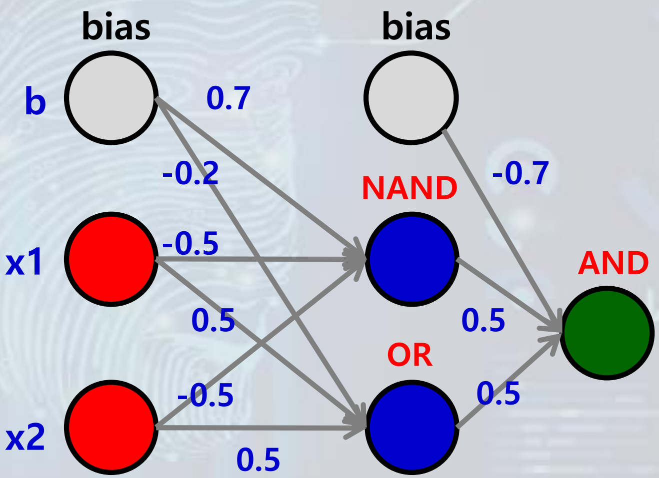
=

XOR
0
1
1
0

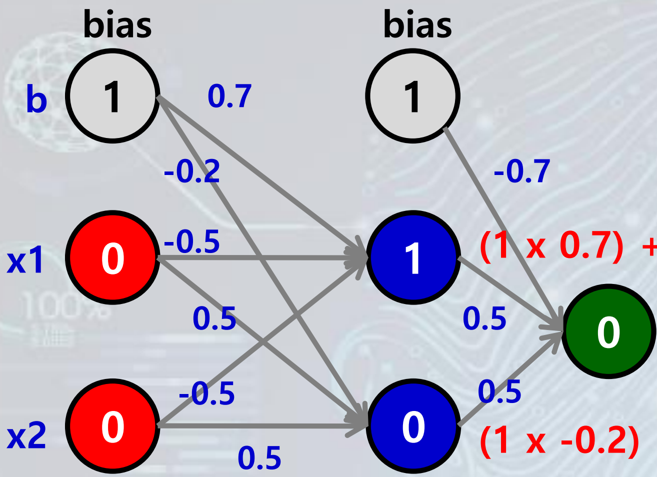
다층 퍼셉트론 (MLP : Multi Layer Perceptron)

- XOR 게이트

x1	x2	XOR
0	0	0
0	1	1
1	0	1
1	1	0



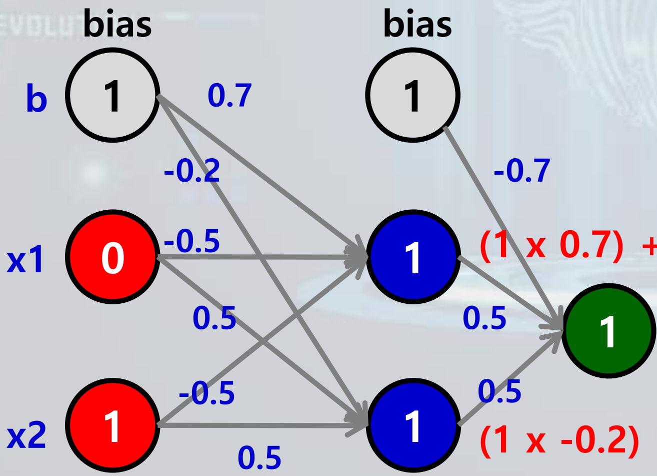
다층 퍼셉트론 (MLP : Multi Layer Perceptron)



(1 x 0.7) + (0 x -0.5) + (0 x -0.5) = 0.7

(1 x -0.7) + (1 x 0.5) + (0 x 0.5) = -0.2

(1 x -0.2) + (0 x 0.5) + (0 x 0.5) = -0.2

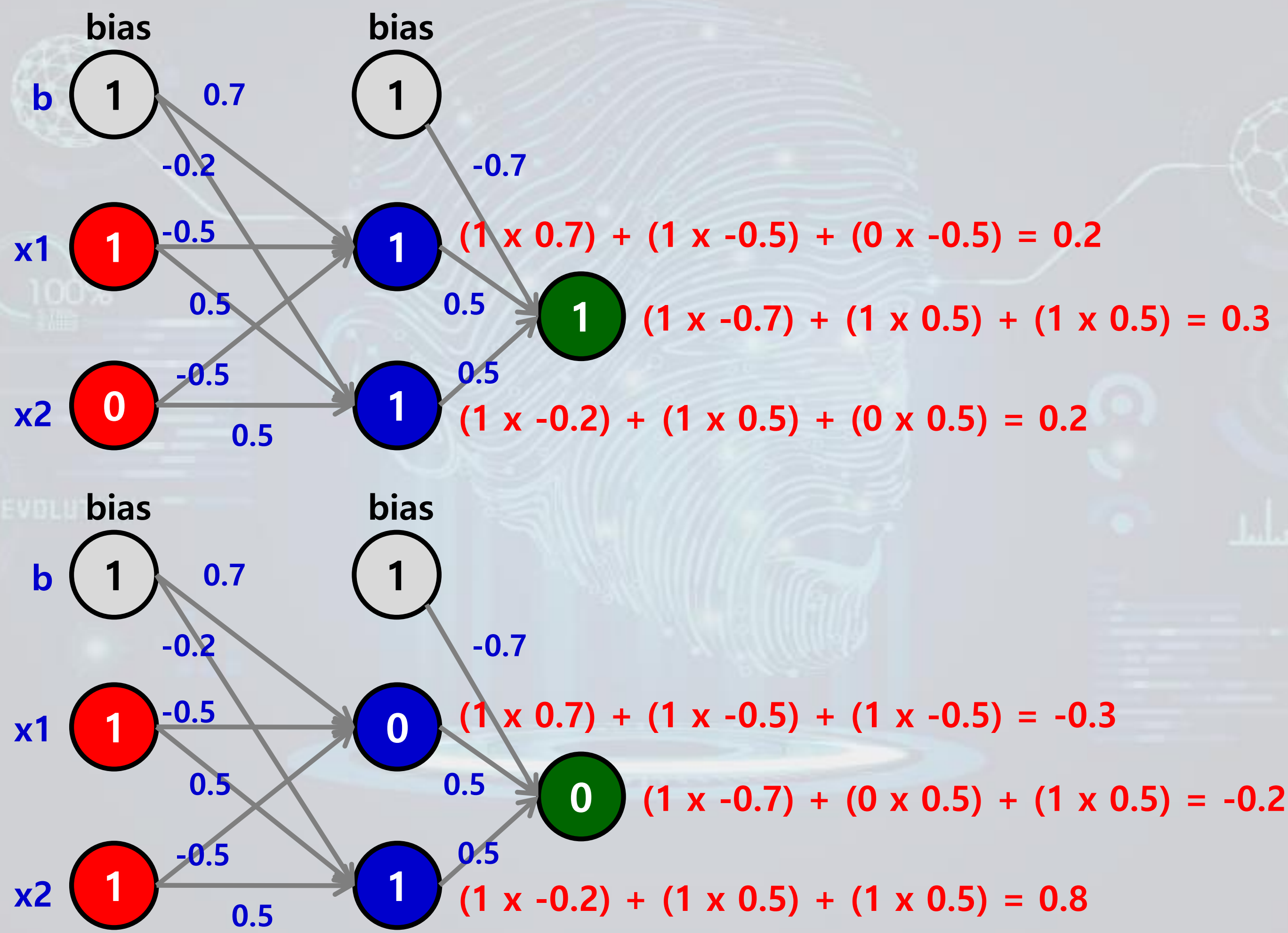


(1 x 0.7) + (0 x -0.5) + (1 x -0.5) = 0.2

(1 x -0.7) + (1 x 0.5) + (1 x 0.5) = 0.3

(1 x -0.2) + (0 x 0.5) + (1 x 0.5) = 0.3

다층 퍼셉트론 (MLP : Multi Layer Perceptron)



실습 (Numpy 사용)

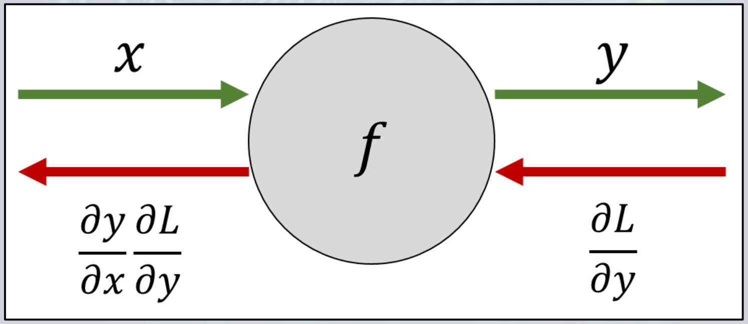
- XOR 게이트

```
import numpy as np

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

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

```
0
1
1
0
```

오차역전파

- **역전파 (Back Propagation)** : 에러를 출력층에서 입력층쪽으로 전파시키면서 최적의 학습 결과를 찾아가는 것 → **학습**
- 순전파 (Foward Propagation) : 입력 데이터를 입력층에서부터 출력층까지 전파시키면서 출력 값을 찾아가는 과정 → **추론**



https://ratsgo.github.io/deep%20learning/2017/05/14/backprop/

오차역전파

- 역전파 (Back Propagation)

말 전달하기 게임 중이예요 ^^ 그런데 ㅠㅠ

틀린 이유를 분석하려고 하는 어떤 것이 나을까요 ?

진행방향

진행반대방향

오차역전파

- 역전파 (Back Propagation)

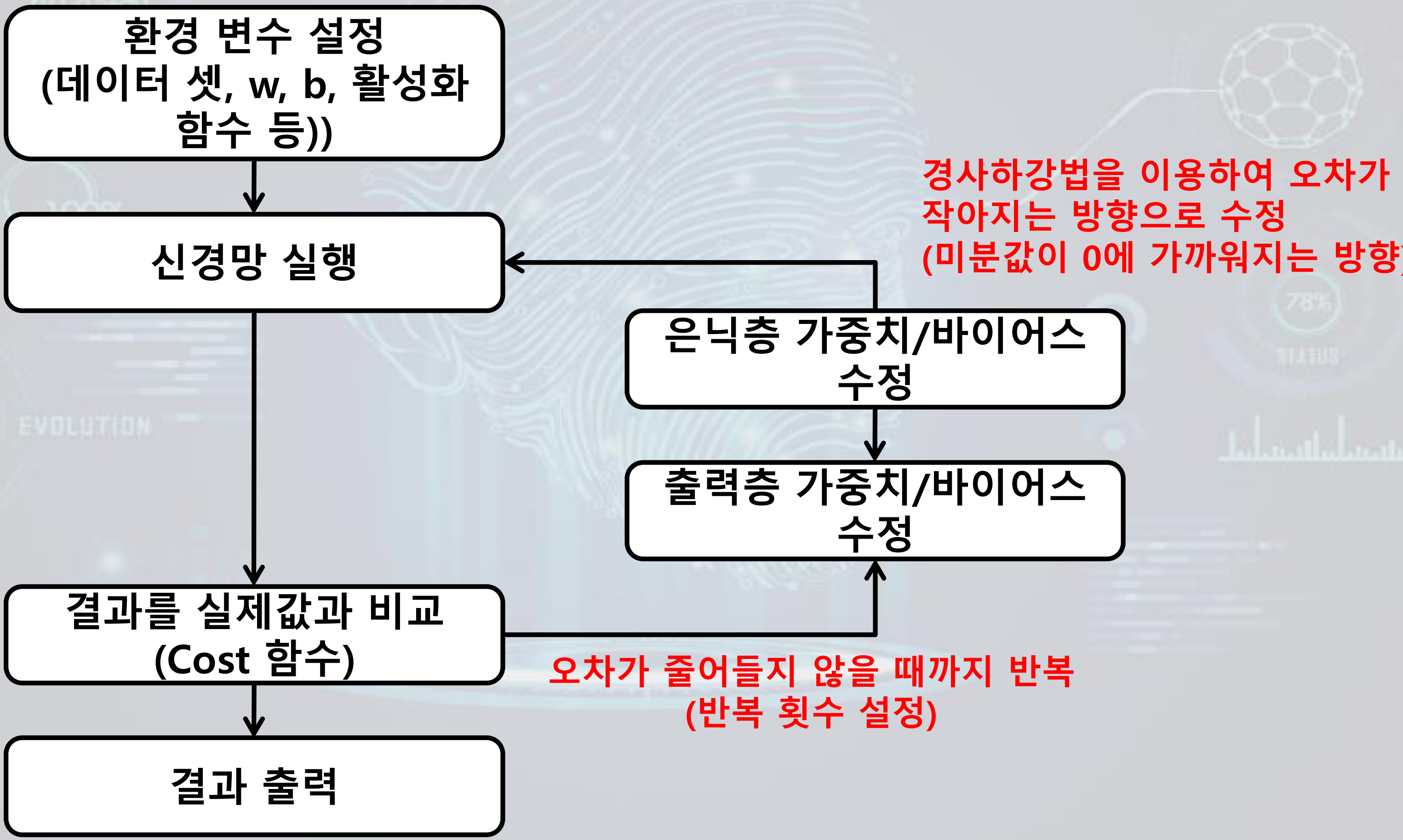
유통 시스템의 문제점을 찾으려면 어떤 것이 나을까요 ?

진행방향

진행반대방향

오차역전파

- 역전파 (Back Propagation) 과정



오차역전파

- 가중치 수정

$$\underbrace{W(t+1)}_{\text{새 가중치}} = \underbrace{Wt}_{\text{이전 가중치}} - \underbrace{\frac{\partial E}{\partial W}}_{\text{이전 가중치에 대한 오차의 기울기 (미분값)}}$$

- **편미분 (∂)** : 여러 개의 변수 중 하나에 대해서만 미분하고 다른 변수 값은 상수로 취급

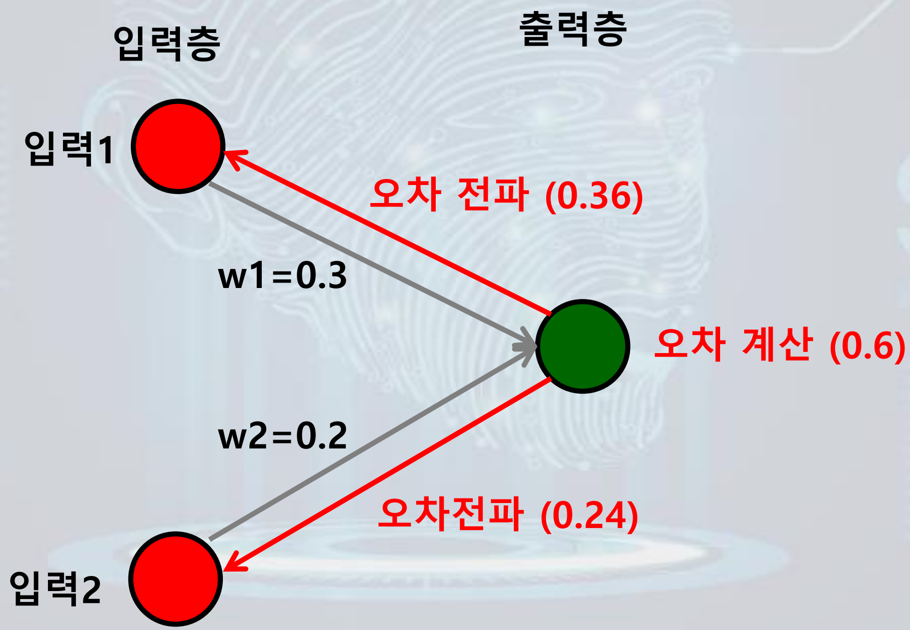
y = 2a + 3b

$$\frac{\partial a}{\partial y} = 2$$

$$\frac{\partial b}{\partial y} = 3$$

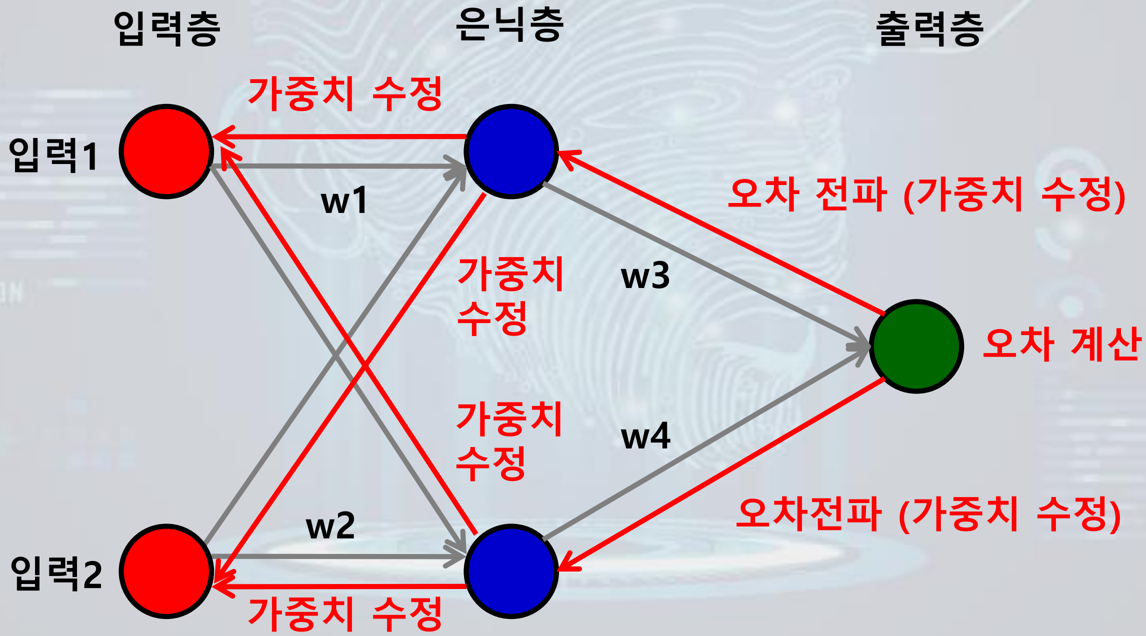
퍼센트론의 오차역전파

- 출력층에서 오류가 0.6이 발생하였다면 입력1 노드로 0.36을 입력2 노드로 0.24의 에러를 전파시켜서 각각이 가중치를 갱신 → **오차 역전파**



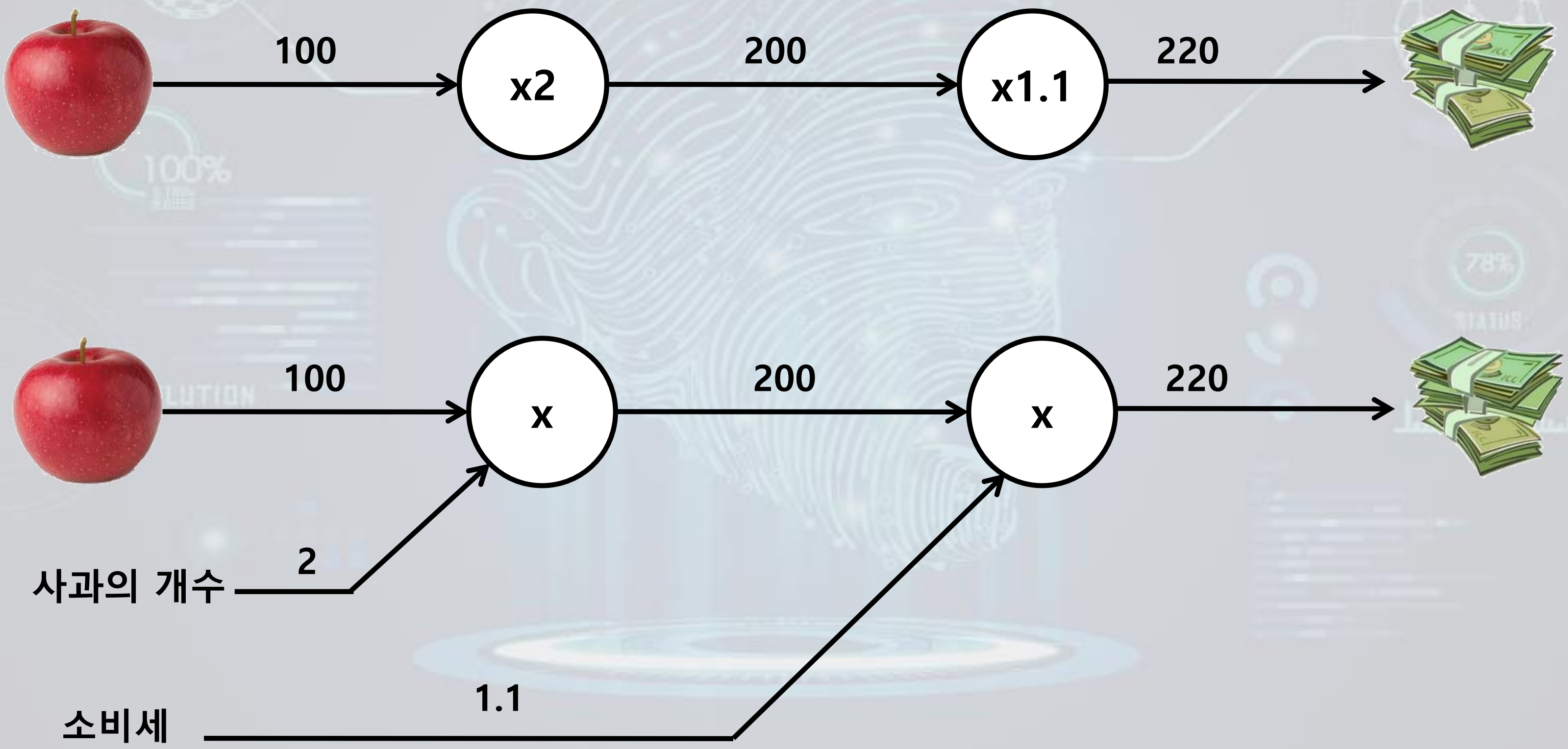
MLP의 오차역전파

- MLP에서도 출력층의 오차를 은닉층으로 전파시켜 가중치나 바이어스를 갱신하고 다시 입력층으로 전파하여 가중치나 바이어스를 갱신



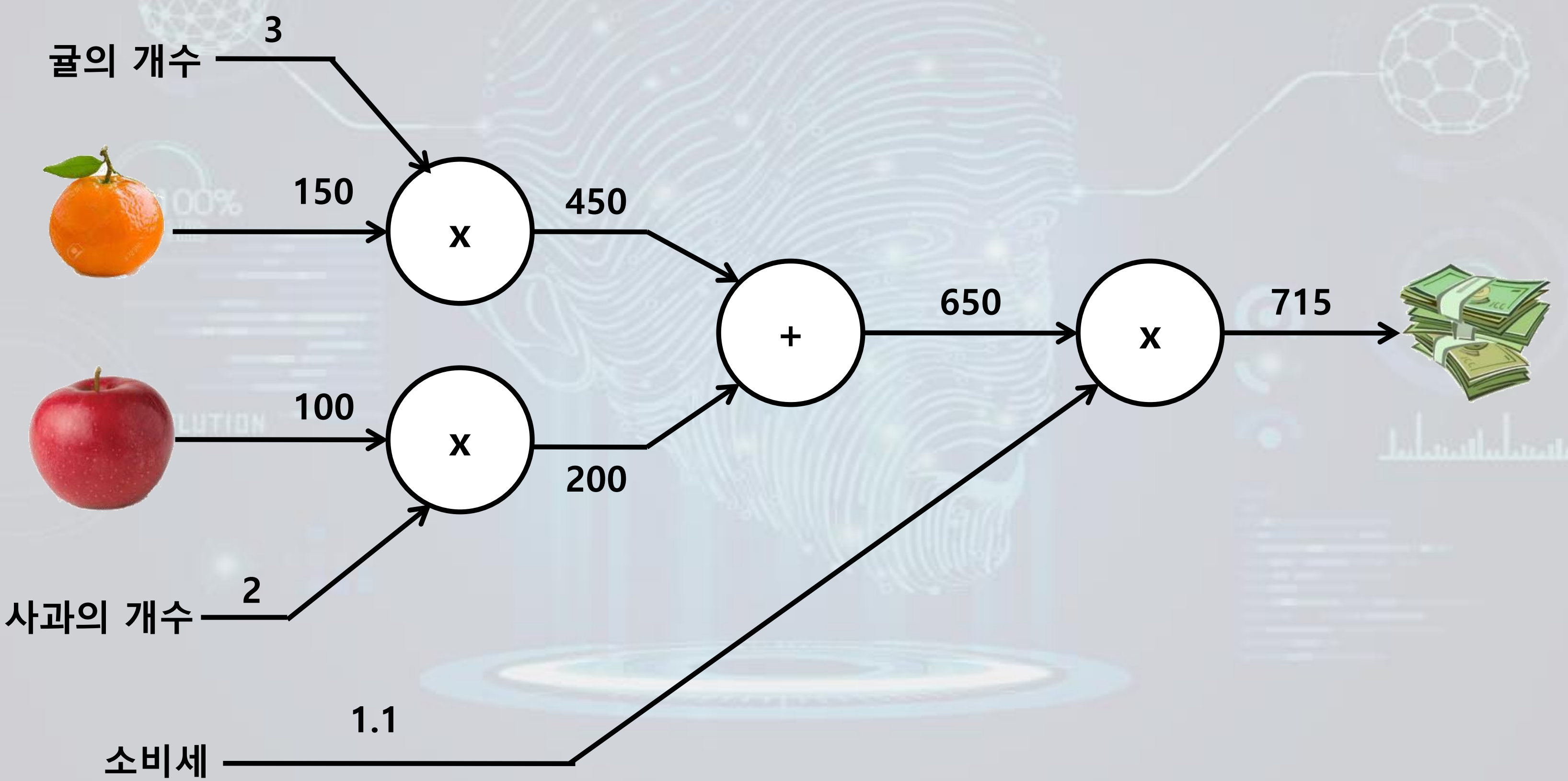
계산 그래프

- 수퍼에서 1개에 100원인 사과를 2개 샀다면 지불 금액은 ? (단, 소비세가 10% 부과됨)



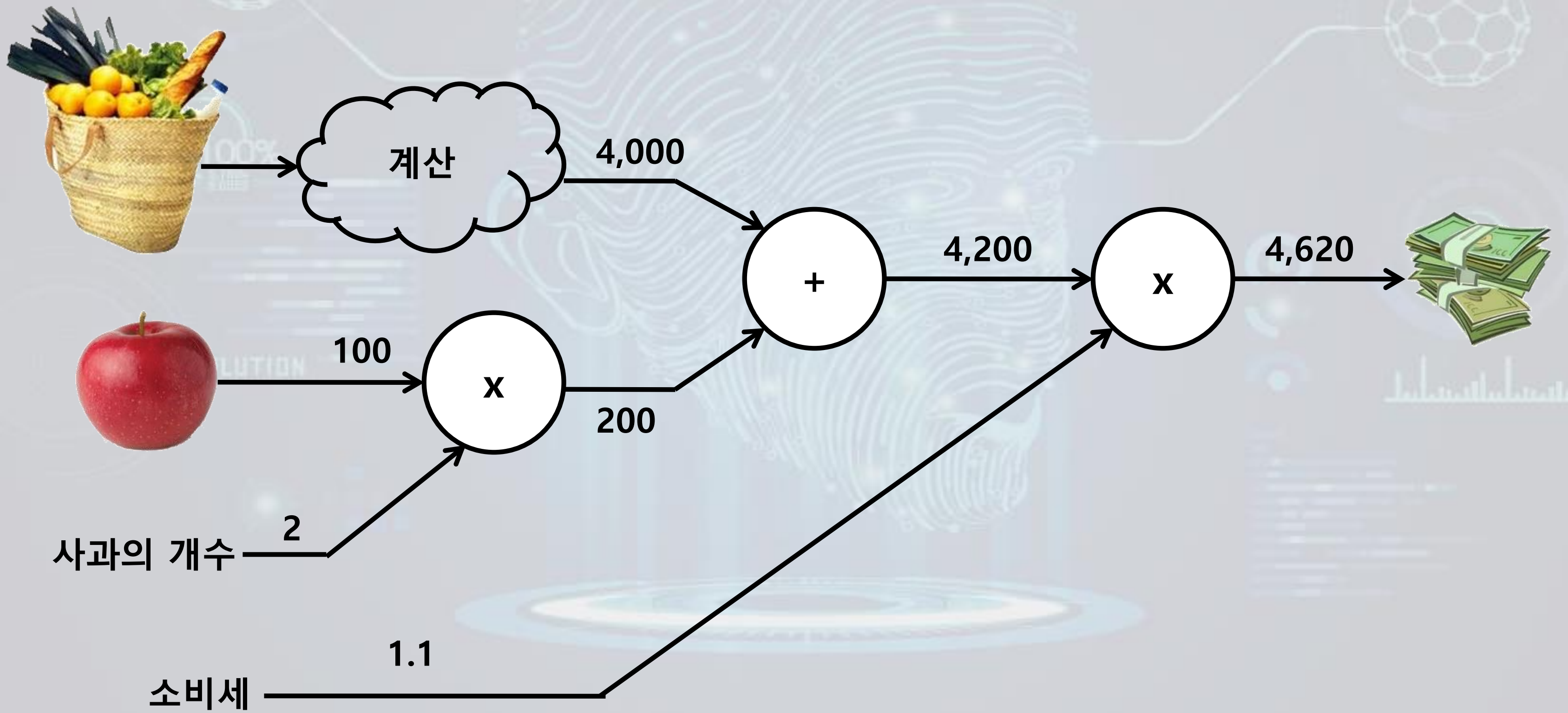
계산 그래프

- 수퍼에서 100원 짜리 사과 2개, 150원 짜리 귤 3개를 샀다면 지불 금액은 ?



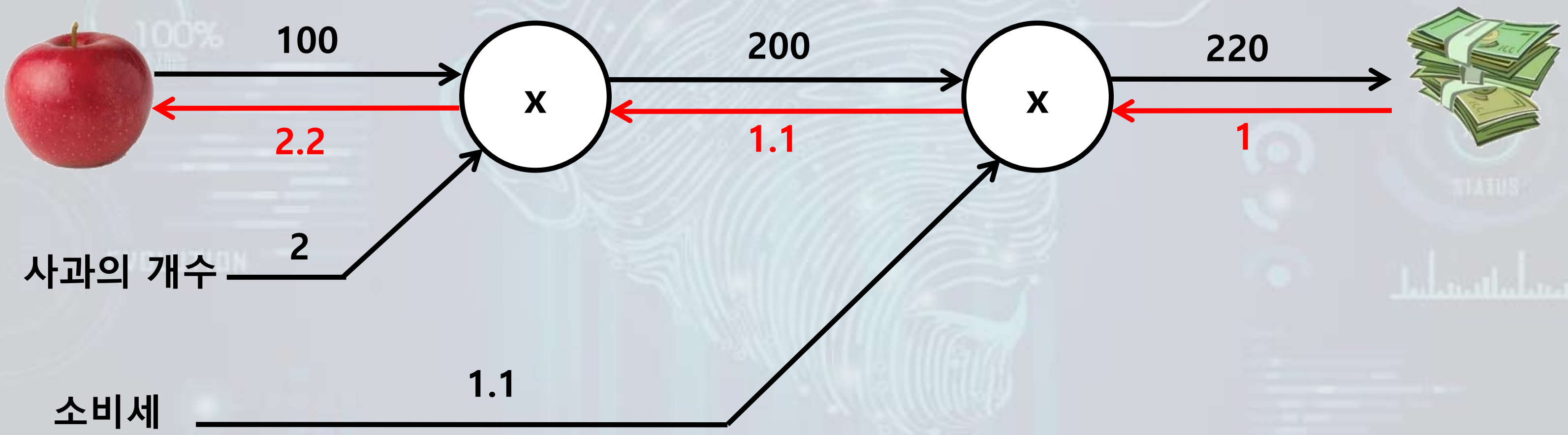
계산 그래프

- 수퍼에서 과일바구니와 사과 2개, 150원 짜리 귤 3개를 샀다면 지불 금액은 ?



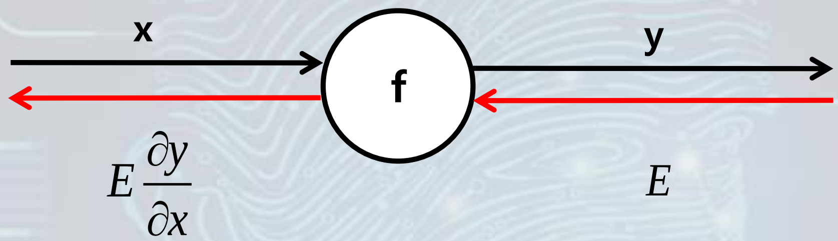
역전파 계산 그래프

- 사과 가격에 대한 지불 금액의 미분값은 2.2 → 사과 1원 오르면 최종 금액은 2.2원 오른다는 의미



연쇄법칙

- 역전파는 국소적 미분을 오른쪽에서 왼쪽으로 전달

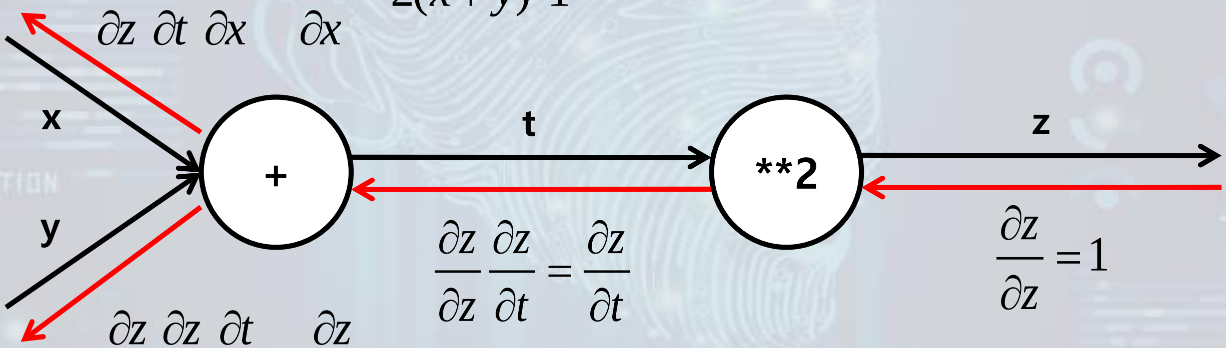


$$\frac{\partial y}{\partial x} = \frac{\text{출력}}{\text{입력}} = \text{입력에대한출력의비}$$

연쇄법칙

$$z = t^2 \qquad t = x + y$$

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



$$\frac{\partial z}{\partial z} = 1$$

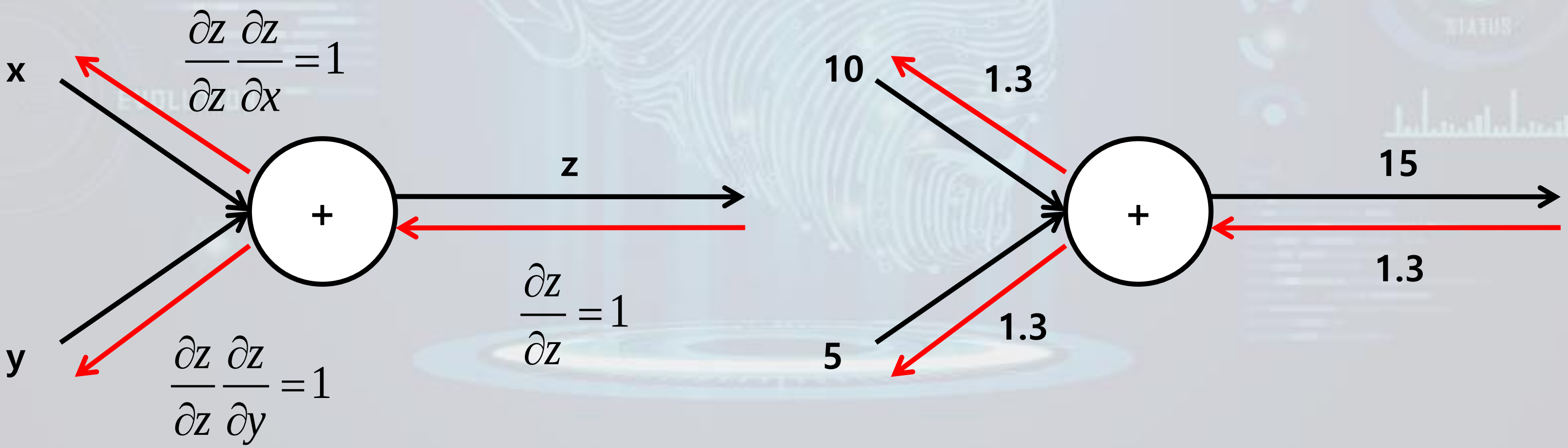
$$\frac{\partial z}{\partial z}\frac{\partial z}{\partial t} = \frac{\partial z}{\partial t}$$
$$= 2t = 2(x+y)$$

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

덧셈 노드의 역전파

- 덧셈 노드의 역전파는 입력 값을 그대로 전파

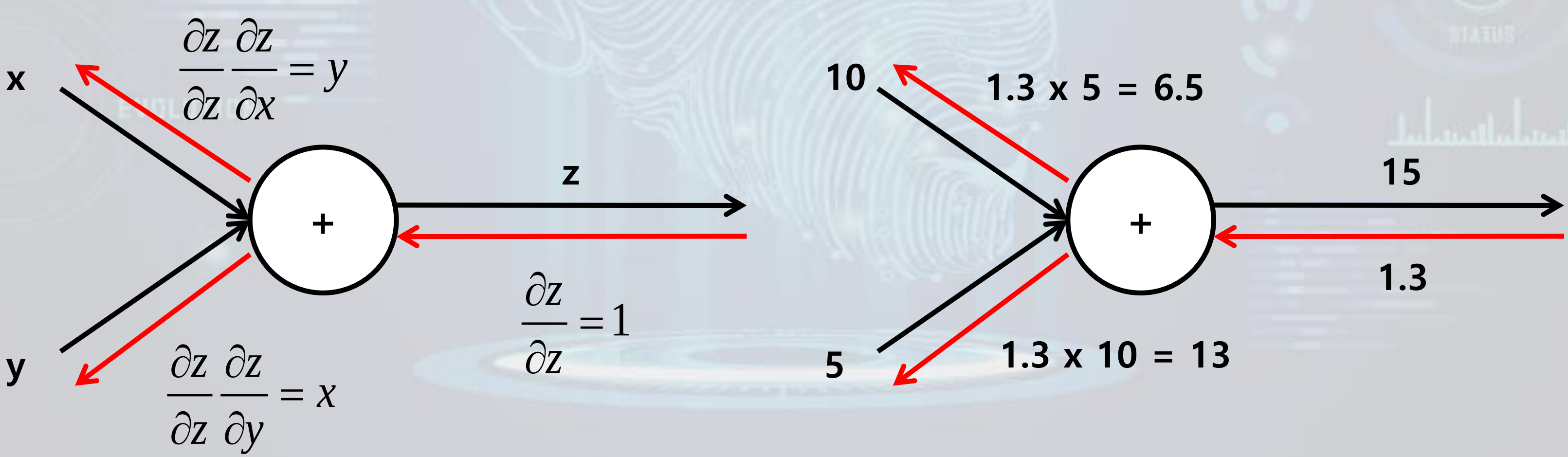
$$z = x + y$$



곱셈 노드의 역전파

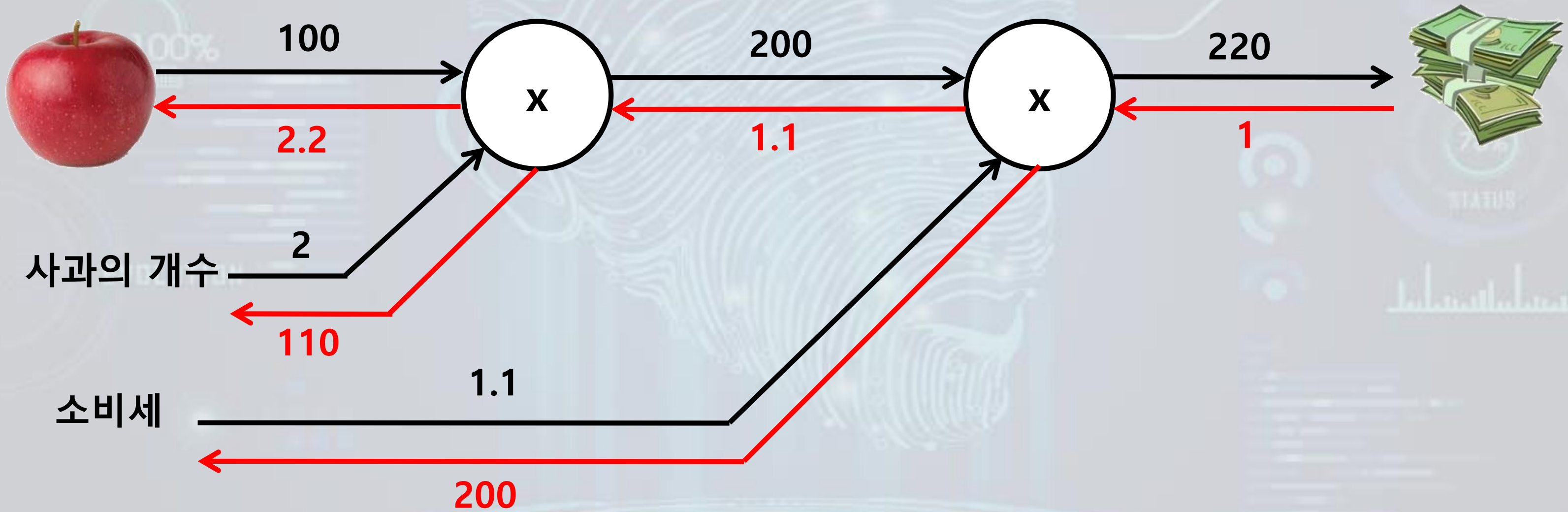
- 곱셈 노드의 역전파는 상류 값에 순전파 때의 입력 신호들을 서로 바꾼 값을 곱해서 하류로 보냄

z = xy



곱셈 노드의 역전파

- 소비세와 사과 가격이 같은 양만큼 오르면 최종 금액에는 소비세가 200의 크기로 사과 가격이 2.2 크기로 영향을 준다는 의미



역전파 실습

- 곱셈계층 클래스

```
class MulLayer:
    def __init__self(self):
        self.x = None
        self.y = None

    def forward(self, x, y):
        self.x = x
        self.y = y
        out = x * y

        return out

    def backward(self, dout):
        dx = dout * self.y
        dy = dout * self.x

        return dx, dy
```

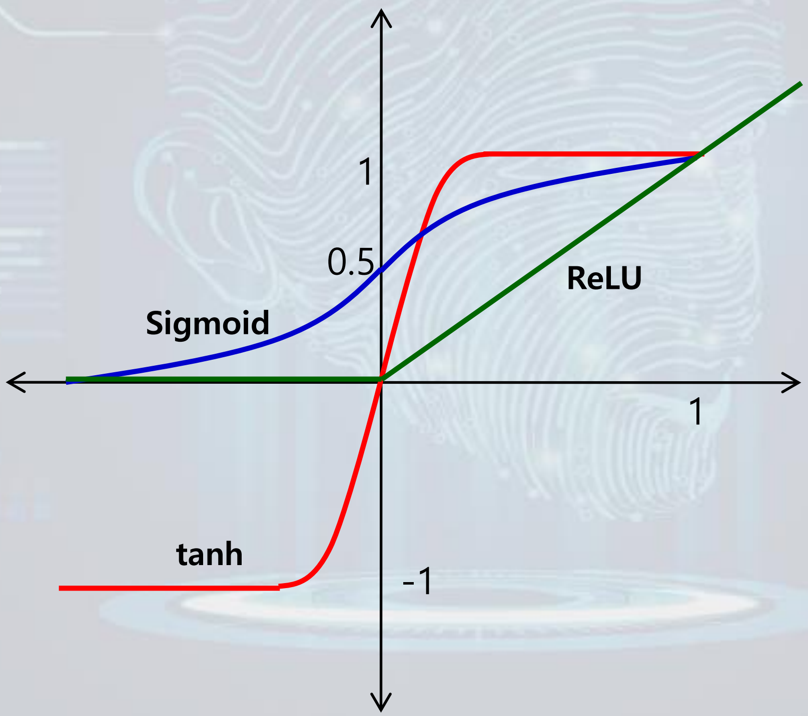
```
220
2.2 110 200
```

역전파 실습

```
apple = 100
apple_num = 2
tax = 1.1

mul_apple_layer = MulLayer()
mul_tax_layer = MulLayer()

apple_price = mul_apple_layer.forward(apple, apple_num)
price = mul_tax_layer.forward(apple_price, tax)

print(price)

dprice = 1
dapple_price, dtax = mul_tax_layer.backward(dprice)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)

print(dapple, dapple_num, dtax)
```

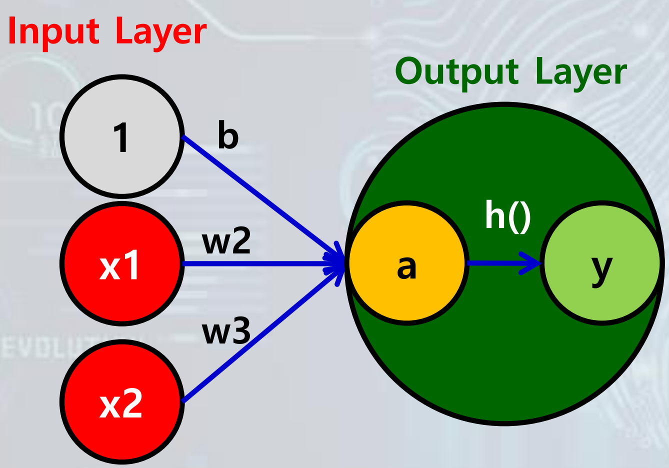
활성화 함수

- **활성화 함수 (Activation Function)** : 뉴런에서 **입력 신호가 일정 크기 이상(임계값)일 때만 신호를 전달**하는 메커니즘을 모방한 함수 → 계단함수, 시그모이드 (sigmoid), tanh, ReLU (Rectified Unit)



활성화 함수 (Activation Function)

- **활성화 과정** : 가중치 신호를 조합한 결과가 a라는 노드가 되고 활성화 함수 h()를 통과하여 y라는 노드로 변환되는 과정



$$y = h(w_1x_1 + w_2x_2 + b)$$

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

- **편향은 뉴런이 활성화되는 기준값**이라고 할 수 있음 → **활성화 함수**
  - 만약 편향이 -0.1이라면 입력신호의 가중치를 곱한 값들의 합이 0.1을 초과할 때만 뉴런이 활성화
  - **편향이 큰 음수라면 그만큼 뉴런이 활성화가 되기 어려운 환경이 됨**

활성화 함수 (Activation Function)

- np.array() : 조건에 따라 False, True를 반환 → int로 변환 필요 (astype())

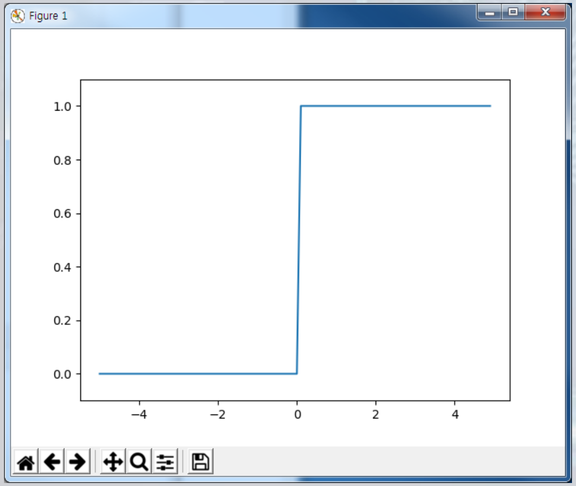
```
import numpy as np

x = np.array([-1.0, 1.0, 2.0])
print(x)
print(x > 0)
print((x > 0).astype(np.int))
```

```
[-1.  1.  2.]
[False  True  True]
[0 1 1]
```

활성화 함수

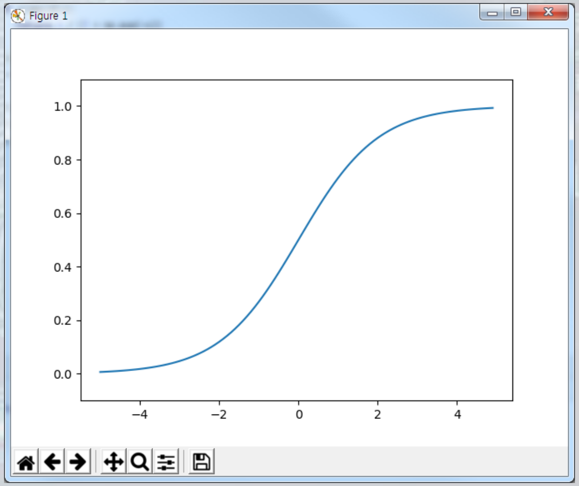
- Sigmoid 함수 문제점
  - Gradient vanishing 문제 발생 : 극단의 미분값(gradient)이 0이 곱해지면 전파되지 않음
  - 활성 함수 결과 값의 중심이 0이 아닌 0.5 → **모두 양수이거나 모두 음수일 가능성**
  - **지수이므로 계산이 복잡**

Step 함수



$$f(x)=\begin{cases}0 & x<0\\ 1 & x\geq 0\end{cases}$$

Sigmoid 함수



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

활성화 함수 (Activation Function)

- 계단 (step) 함수 구현

```python
import numpy as np
import matplotlib.pylab as plt

def step_function(x):
    return np.array(x > 0, dtype=np.int)    #bool형을 int형으로 변환

x = np.arange(-5.0, 5.0, 0.1)
y = step_function(x)

plt.plot(x, y)          #그래프를 그린다
plt.ylim(-0.1, 1.1)    #y축 범위
plt.show() #그래프를 보여준다
```

- np.array() : 넘파이 배열 생성
- np.arange(초기, 끝, 증가) : 넘파이 배열을 생성

활성화 함수 (Activation Function)

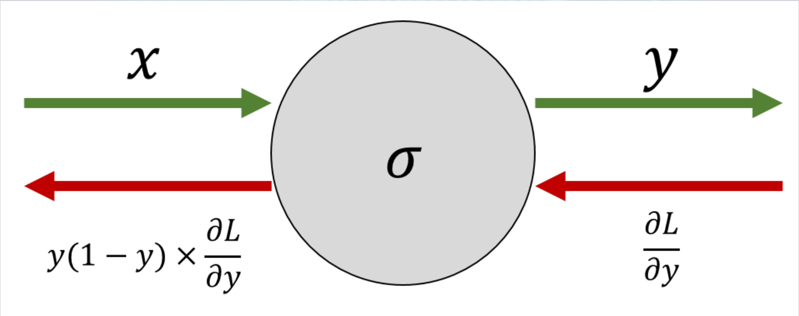
- Sigmoid 함수 구현

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

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

x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)

plt.plot(x, y)
plt.ylim(-0.1, 1.1)
plt.show()
```

Sigmoid 노드의 역전파

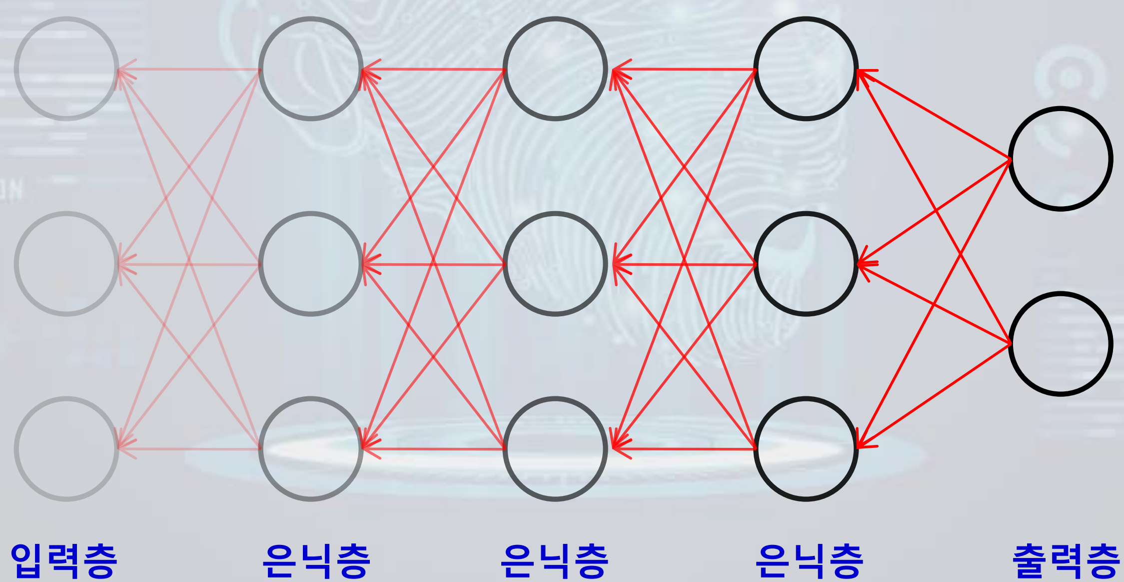
$$y = \frac{1}{1+e^{-x}}$$

$$\frac{\partial y}{\partial x} = y(1-y)$$



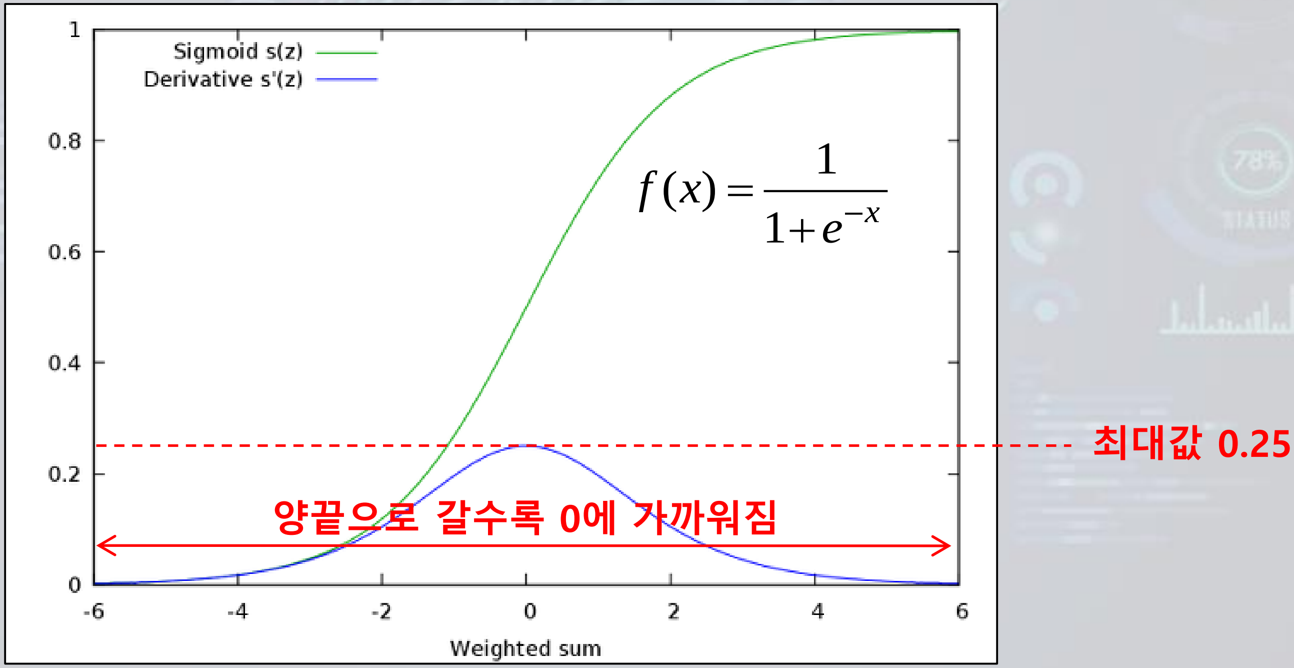
신경망에서 딥러닝으로

- MLP이 오차 역전파를 만나 신경망이 되었고 XOR 문제를 해결하였음.
- 하지만 오차 역전파는 출력층부터 입력층까지 하나씩 앞으로 돌아가면서 각 층의 가중치를 수정하는데 이때 미분 (기울기)을 하기 때문에 중간에 기울기가 0이 되는 경우가 발생 → Gradient Vanishing



신경망에서 딥러닝으로

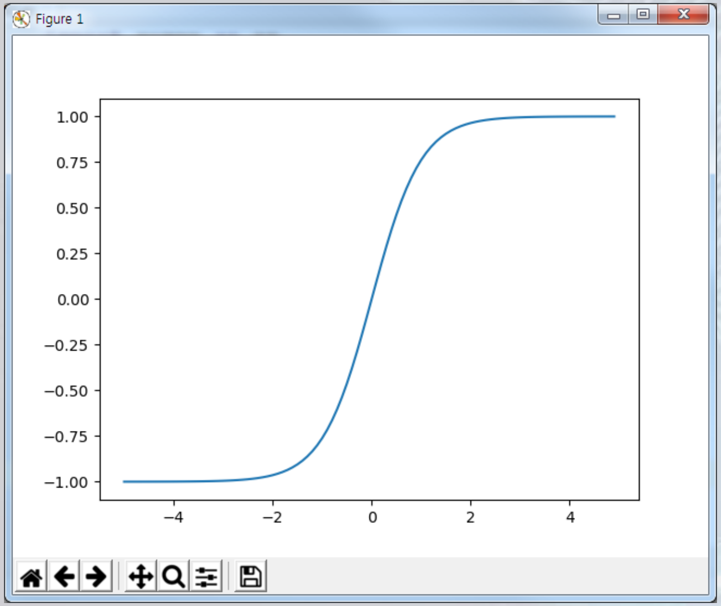
- Back propagation을 학습하는 과정 중에서 Sigmoid의 미분함수가 들어가는데 **Sigmoid의 미분 함수의 최대값이 0.25이고 양쪽으로 갈수록 0에 가까워짐** → 학습이 계속 진행되면서 gradient가 0에 수렴해질 가능성이 생김 → **Gradient Vanishing**

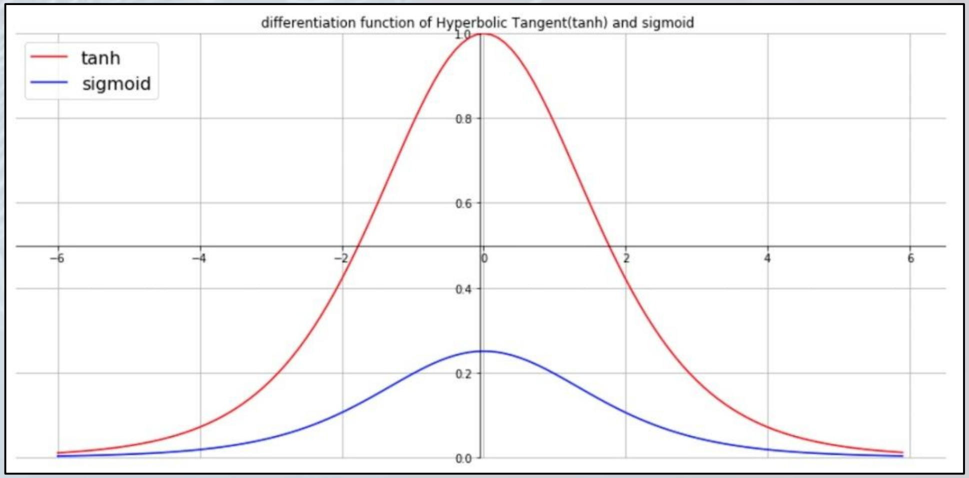


시그모이드 함수를 미분

활성화 함수

- Gradient vanishing 문제가 Sigmoid 함수보다는 **tanh 함수가 덜 발생** → 결과 값이 [-1, 1] 사이로 제한되고 중심값이 0이므로
- 여전히 vanishing gradient 문제 발생





시그모이드와 tanh의 미분 그래프

$$\tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

활성화 함수 (Activation Function)

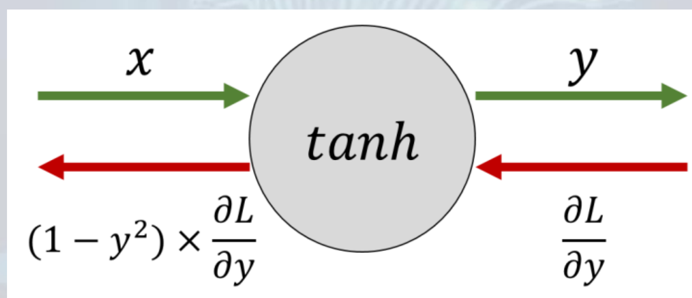
- tanh 함수 구현

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

def tanhFun(x):
    return np.tanh(x)

x = np.arange(-5.0, 5.0, 0.1)
y = tanhFun(x)

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

tanh 노드의 역전파

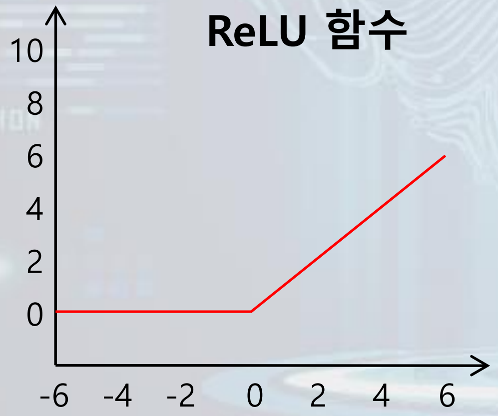
$$y = \tanh(x)$$

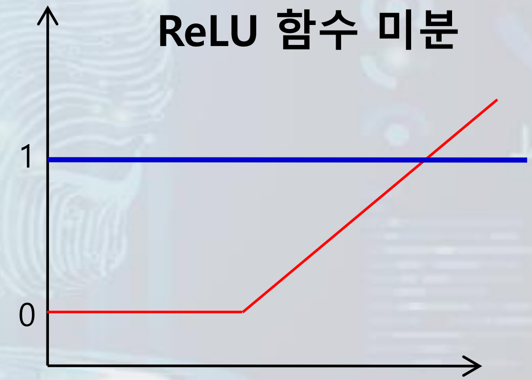
$$\frac{\partial y}{\partial x} = 1 - y^2$$



활성화 함수

- ReLU 함수는 vanishing gradient를 감소시키는 가장 큰 대안
  - **Sparse activation** : 0이하의 입력에 대해 0을 출력함으로써 부분적으로 활성화시킬 수 있음
  - **Efficient gradient propagation** : **gradient의 vanishing이 없음** → 양극단 값이 포화되지 않음 (gradient가 exploding되지 않음)
  - **Efficient computation** : **선형함수이므로 미분 계산이 간단** (6배 정도 빠름)
  - **Scale-invariant** : 스케일 조정에 따라 값이 변하지 않음



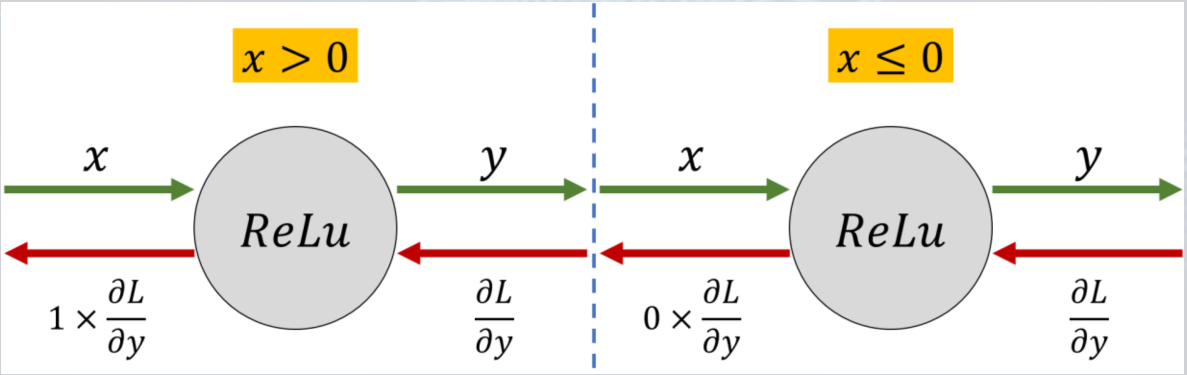


$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

ReLU 노드의 역전파

$$\begin{cases} y = x & (x > 0) \\ y = 0 & (x \le 0) \end{cases}$$

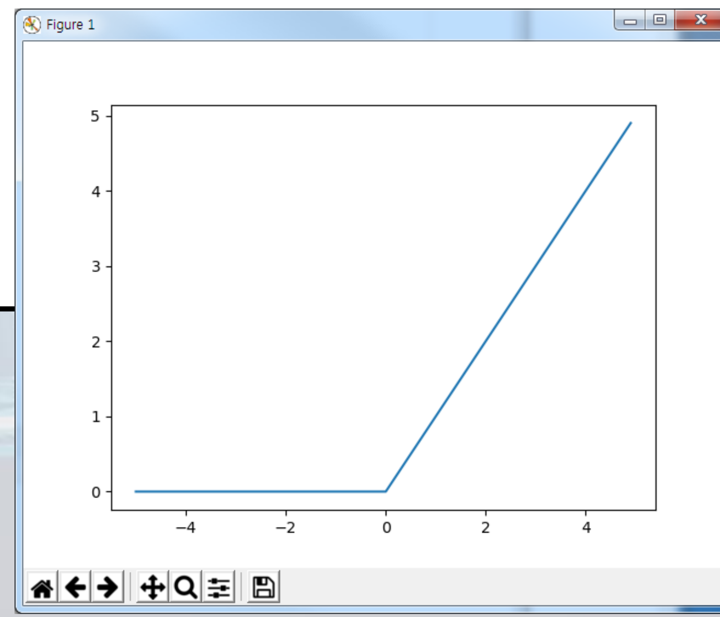
$$\begin{cases} \dfrac{\partial y}{\partial x} = 1 & (x > 0) \\ \dfrac{\partial y}{\partial x} = 0 & (x \le 0) \end{cases}$$



활성화 함수 (Activation Function)

- ReLU 함수 구현

```python
import numpy as np
import matplotlib.pylab as plt

def relu(x):
    return np.maximum(0, x)

x = np.arange(-5.0, 5.0, 0.1)
y = relu(x)

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



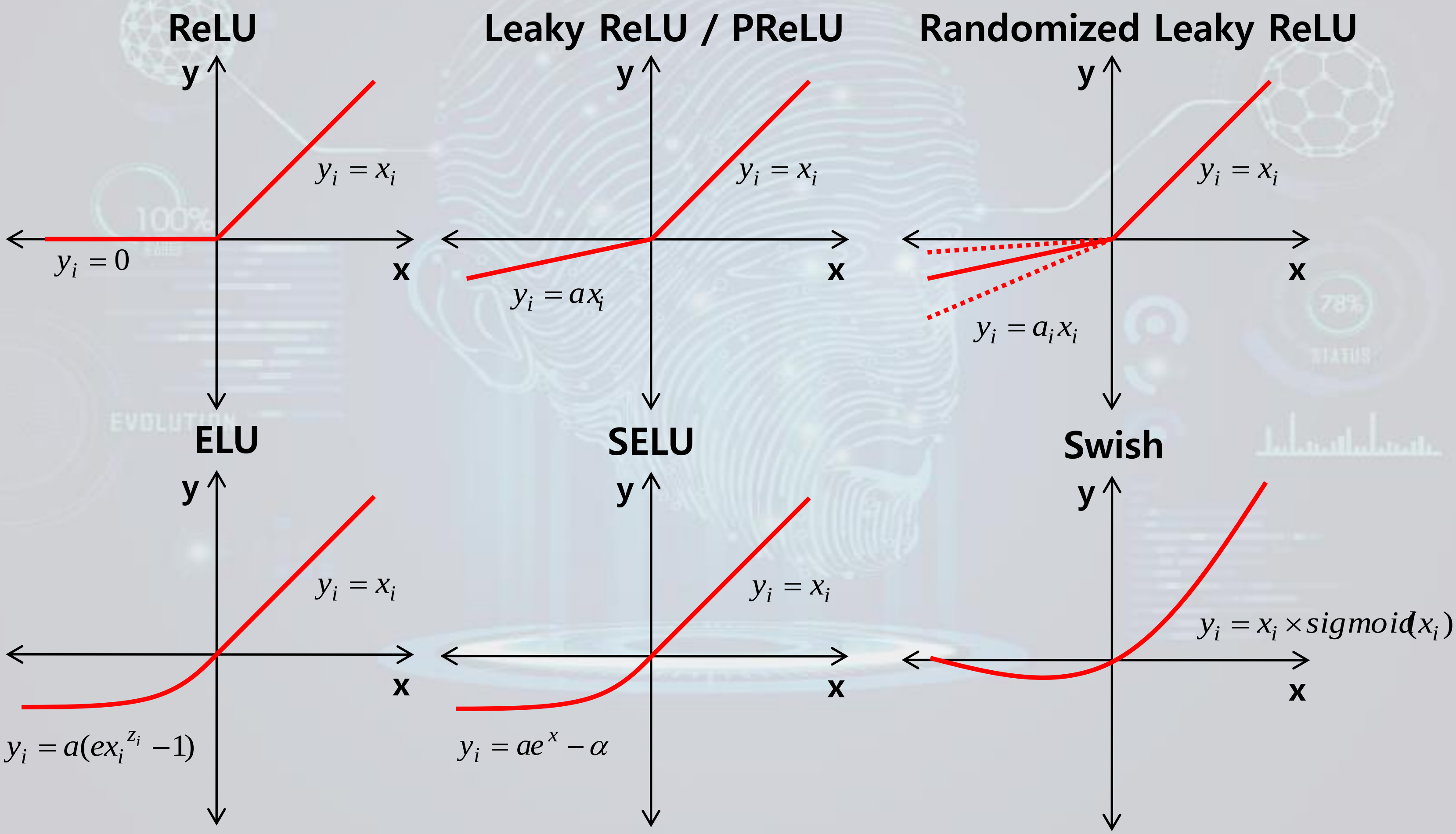
활성화 함수

- 단점 : **음수를 모두 0으로 처리하기 때문에 한번 음수가 나오면 그 노드는 학습되지 않는다는 문제**
  → 좋지 않는 성능 → leaky ReLU나 다른 ReLU 사용
  - Leaky ReLU : 음수의 기울기 값을 0이 아닌 작은 값(0.1, 0.01 등)으로 설정
  - Parametric ReLU (PReLU) : 음수의 기울기 값이 변경
  - ELU (Exponential Linear Unit) : 음수의 기울기 값을 지수 형태로 설정 → PReLU가 비슷한 성능
  - SELU (Scaled ELU) : 2개의 파라미터를 이용하여 기울기를 변경 → 일정한 분산 → PReLU와 비슷
  - Swish : 구글에서 나온 함수 → 성능 우수
  - maxout : 두 개의 w와 b 중에서 큰 값을 선택 → 성능 우수

- 활성화 함수로 ReLU로 사용하더라도 **마지막 Layer는 Sigmoid나 tanh 함수를 사용** → 0~1 사이의 값을 나타내야 정확히 분류 하는데 좋기 때문

활성화 함수

- ReLU의 종류



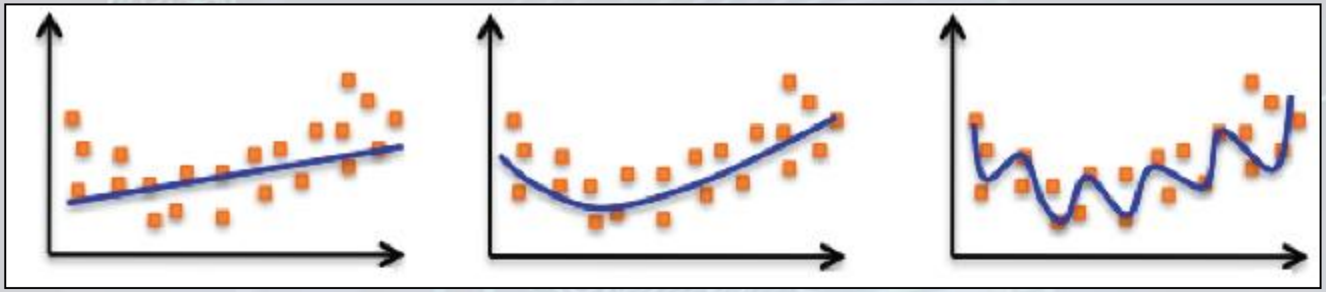
활성화 함수

- 문제 유형에 따라 사용되는 활성화 함수와 오차함수의 종류

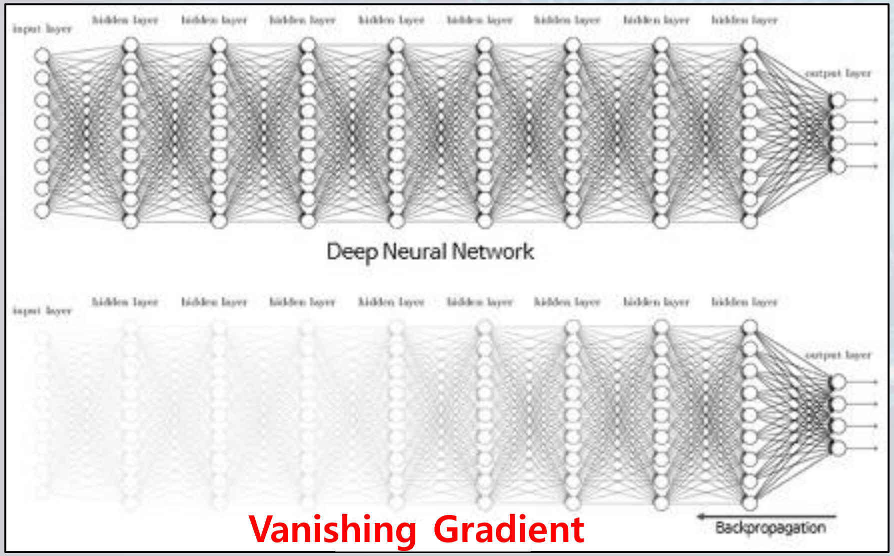
유형	출력층 활성화 함수	오차함수
회귀	항등 함수	제곱오차
이진 분류	로지스틱 함수	교차 엔트로피
다클래스 분류	소프트맥스 함수	교차 엔트로피

다층 퍼셉트론 (MLP : Multi Layer Perceptron)

- MLP의 2가지 문제



Underfitting   Generalization   Overfitting



역전파를 사용할 때 sigmoid 함수 사용

→ 0과 1값이 심하게 변형되어 감소됨

→ 층이 깊어지면 0에 가까운 값이 됨

→ Vanishing (Exploding)