

# 6장 MLP(다층 퍼셉트론)

- 퍼셉트론의 작동원리를 이해한다.
- MLP의 작동 원리를 이해한다.
- 경사하강법을 이해한다.
- 역전파 알고리즘을 이해한다.
- 넘파이

MLP는  
신경망에서 아주 중요한  
위치를 차지하고 있습니다.  
여러 가지 개념들을  
확실하게 이해하고  
넘어가세요!



- 퍼셉트론(perceptron)은 1957년에 로젠블라트(Frank Rosenblatt)가 고안한 인공 신경망이다.

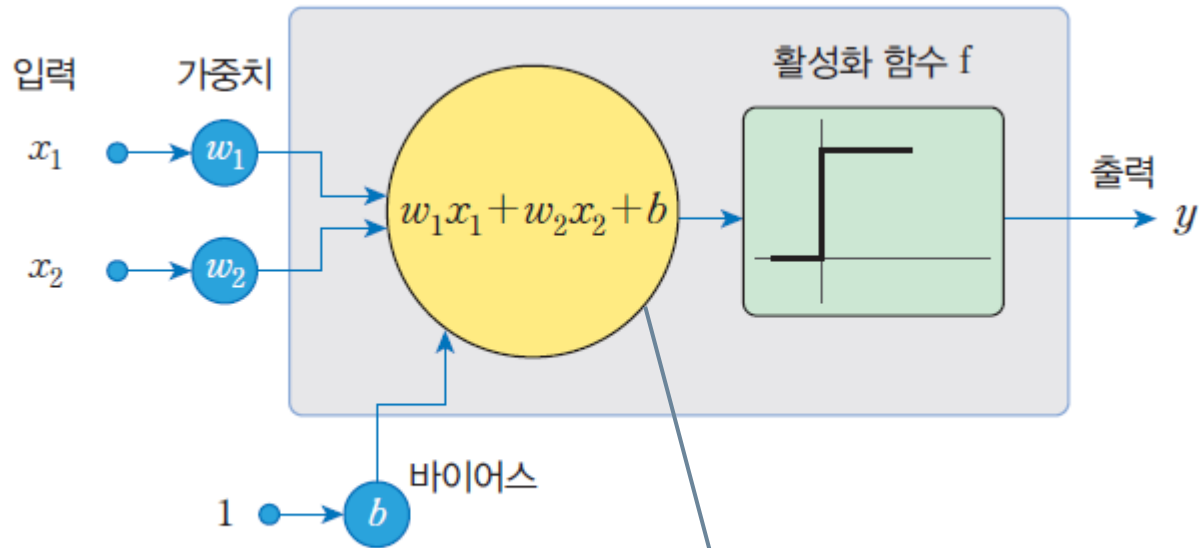


그림 5-5 퍼셉트론에서의 뉴런

$$b = -\theta$$



- 뉴런에서는 입력 신호의 가중치 합이 어떤 임계값을 넘는 경우에만 뉴런이 활성화되어서 1을 출력한다. 그렇지 않으면 0을 출력한다.

$$y = \begin{cases} 1 & \text{if } (w_1x_1 + w_2x_2 + b \geq 0) \\ 0 & \text{otherwise} \end{cases}$$



# 퍼셉트론은 논리 연산을 할 수 있을까?

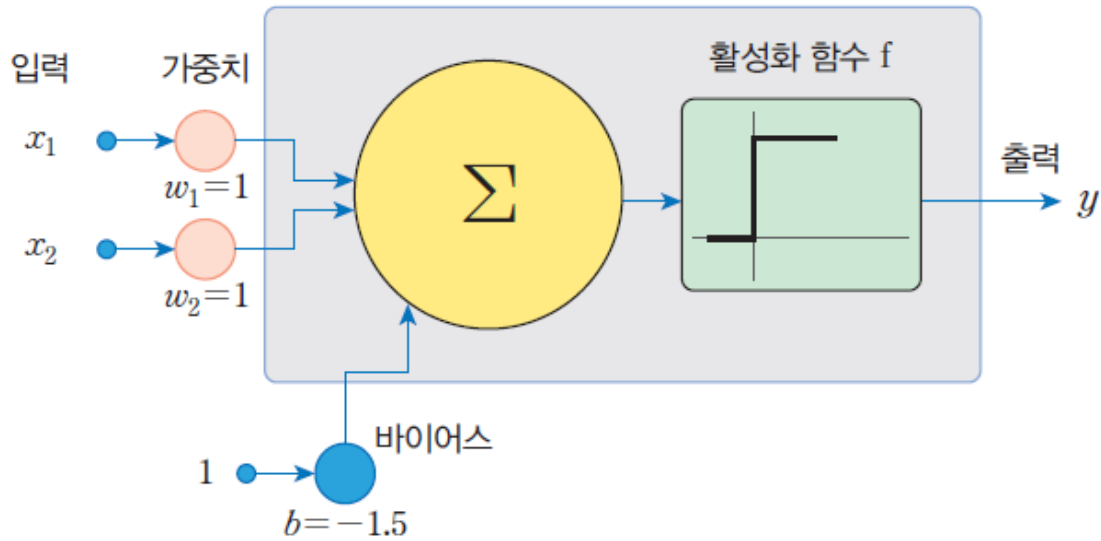


그림 5-6 논리 연산을 하는 퍼셉트론

$x_1$	$x_2$	$y$
0	0	0
1	0	0
0	1	0
1	1	1



# 퍼셉트론은 논리 연산을 학습할 수 있을까?

표 5-2 퍼셉트론 출력 계산

$x_1$	$x_2$	$w_1x_1 + w_2x_2$	$b$	$y$
0	0	$1*0+1*0=0$	-1.5	0
1	0	$1*1+1*0=1$	-1.5	0
0	1	$1*0+1*1=1$	-1.5	0
1	1	$1*1+1*1=2$	-1.5	1

- 계단 함수

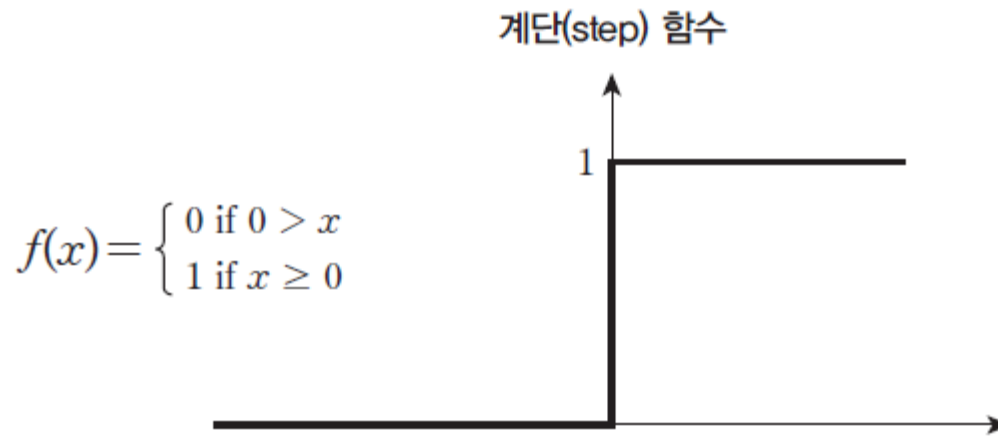


그림 5-7 퍼셉트론에서의 활성화 함수



# 퍼셉트론 구현 #1 (순수 파이썬 사용\_1번째 소스)

```
epsilon = 0.0000001
```

```
def perceptron(x1, x2):
```

```
    w1, w2, b = 1.0, 1.0, -1.5
```

```
    sum = x1*w1+x2*w2+b
```

```
    if sum > epsilon :
```

```
        return 1
```

```
    else :
```

```
        return 0
```

```
print(perceptron(0, 0))
```

```
print(perceptron(1, 0))
```

```
print(perceptron(0, 1))
```

```
print(perceptron(1, 1))
```

# 부동소수점 오차를 방지하기 위하여

```
0  
0  
0  
1
```



# 퍼셉트론 구현 #2(넘파이 사용\_2번째 소스)

```
import numpy as np
epsilon = 0.0000001

def perceptron(x1, x2):
    X = np.array([x1, x2])
    W = np.array([1.0, 1.0])
    B = -1.5
    sum = np.dot(W, X)+B
    if sum > epsilon :
        return 1
    else :
        return 0

print(perceptron(0, 0))
print(perceptron(1, 0))
print(perceptron(0, 1))
print(perceptron(1, 1))
```

```
0
0
0
1
```



# 퍼셉트론의 한계점

- XOR 연산

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

...  
00 -> 1  
01 -> 1  
10 -> 0  
11 -> 0

원하는 출력  
L L 출 L  
L L L L .



# 다층 퍼셉트론으로 XOR 문제를 해결

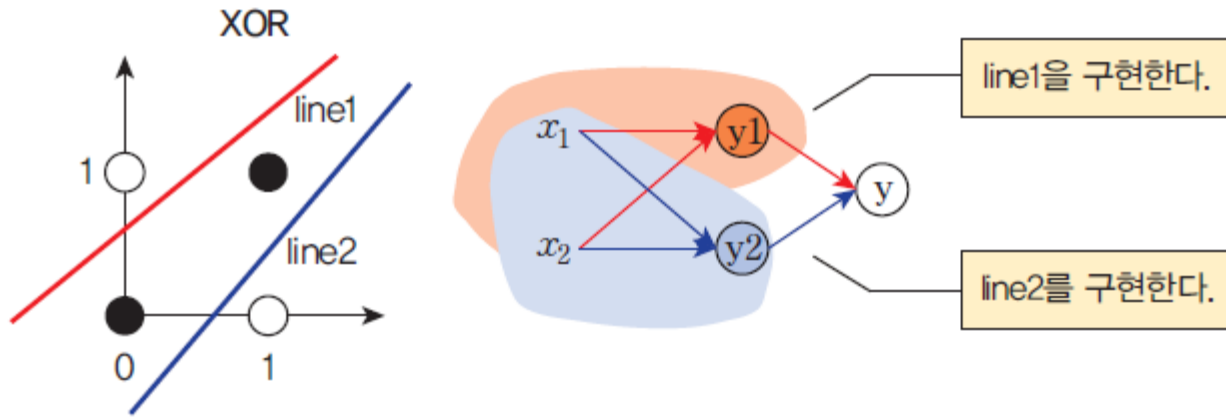
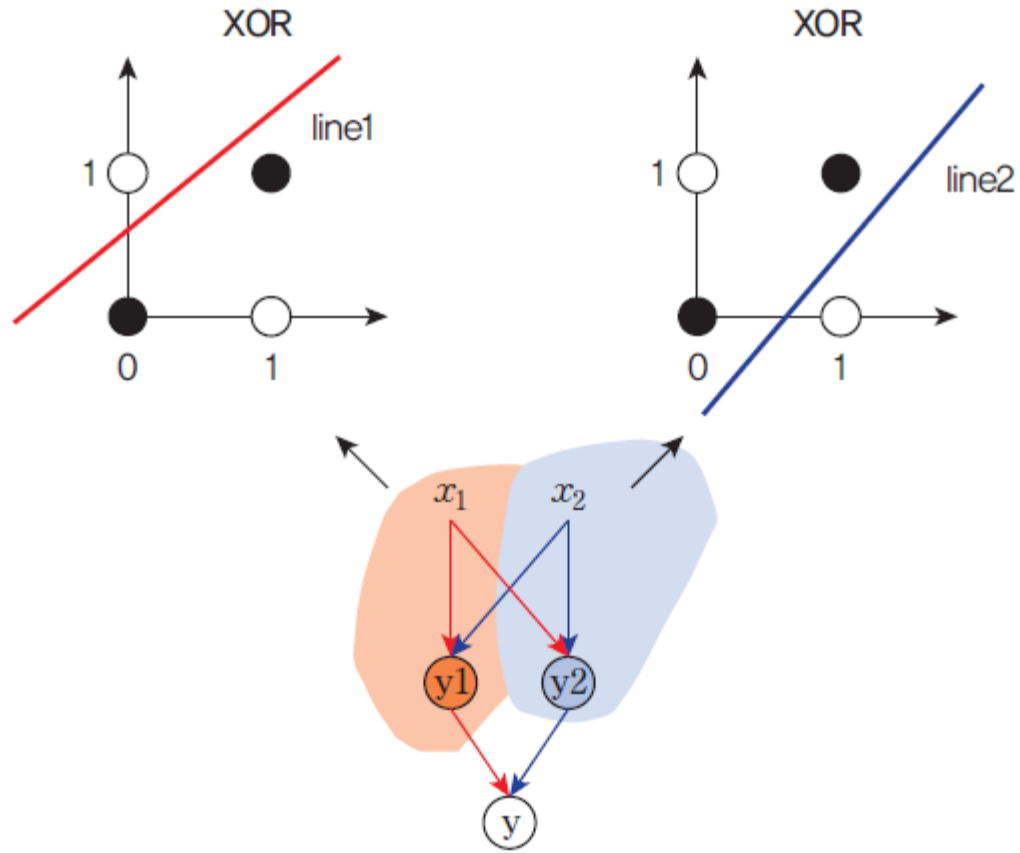


그림 5-13 다층을 사용하는 퍼셉트론



# 다층 퍼셉트론으로 XOR 문제를 해결





# 다층 퍼셉트론으로 XOR 문제를 해결

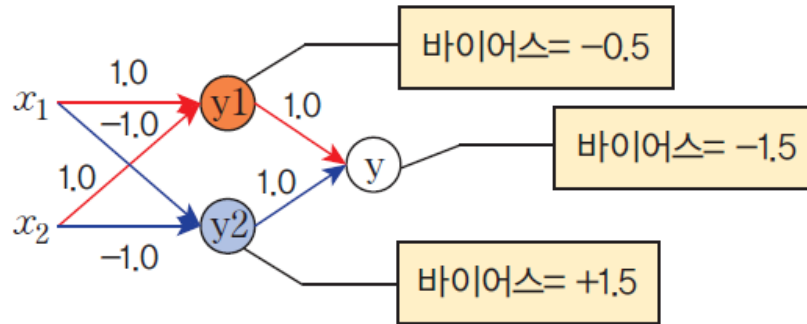


그림 5-14 다층 퍼셉트론에서 XOR 문제 해결

$x_1$	$x_2$	$y_1$	$y_2$	$y$	XOR 출력
0	0	0	1	0	0
1	0	1	1	1	1
0	1	1	1	1	1
1	1	1	0	0	0

- 다층 퍼셉트론(multilayer perceptron: MLP): 입력층과 출력층 사이에 은닉층(hidden layer)을 가지고 있는 퍼셉트론

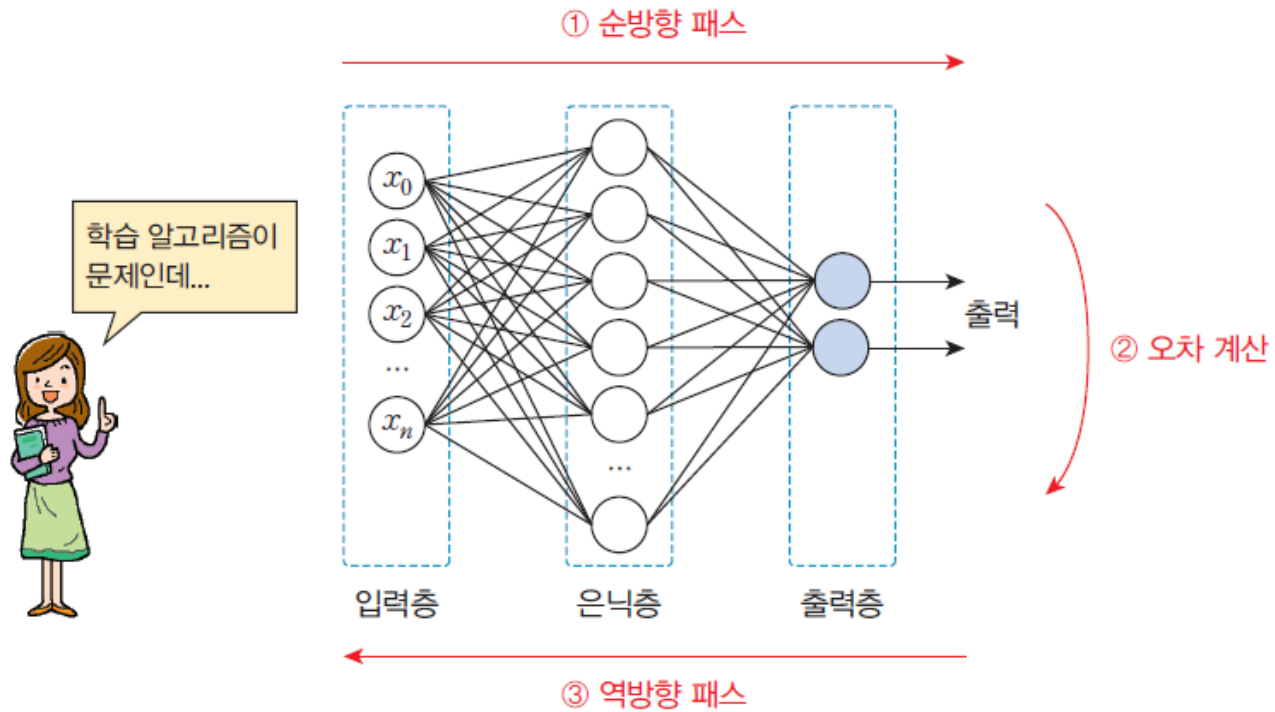


그림 6-1 MLP의 구조

- 활성화 함수(activation function)은 입력의 총합을 받아서 출력값을 계산하는 함수이다.
- MLP에서는 다양한 활성화 함수를 사용한다.

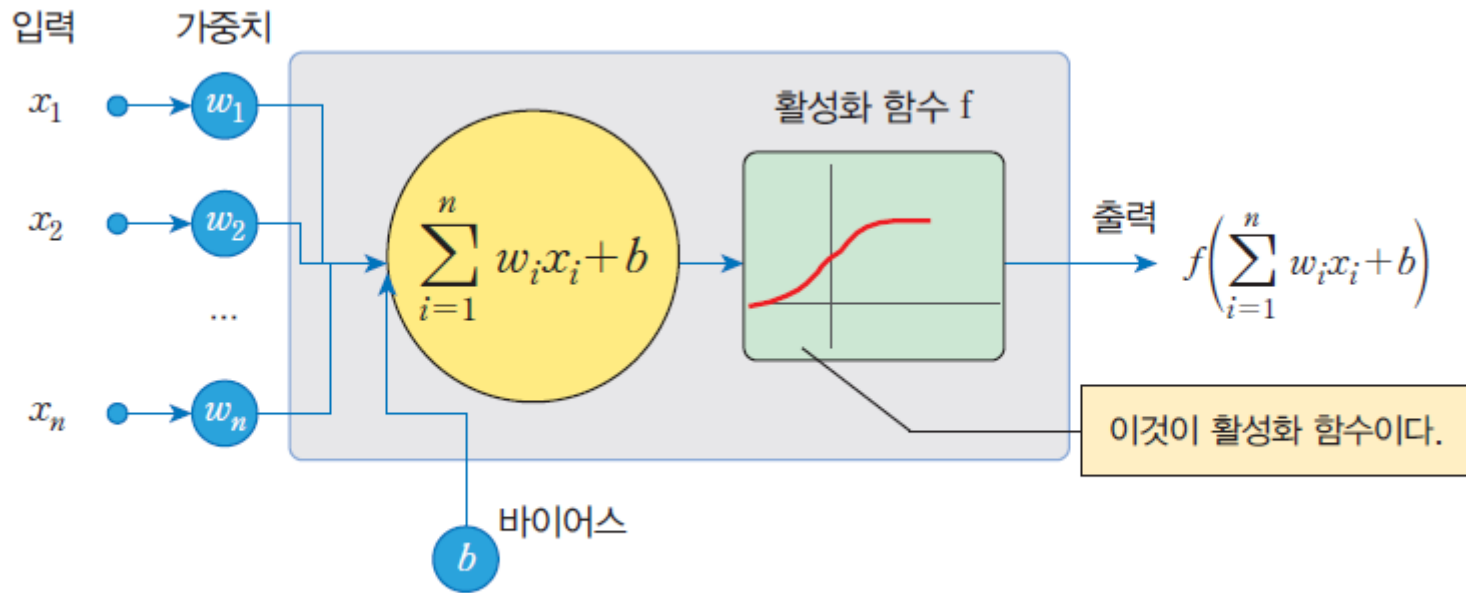
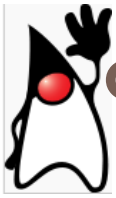


그림 6-2 활성화 함수



# 이반적으로 많이 사용되는 활성화 함수

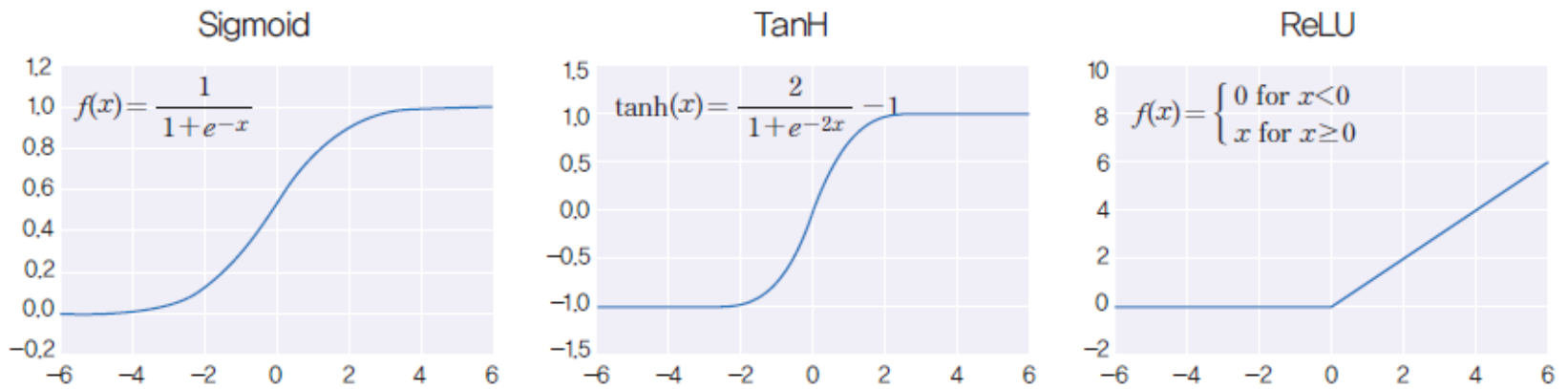
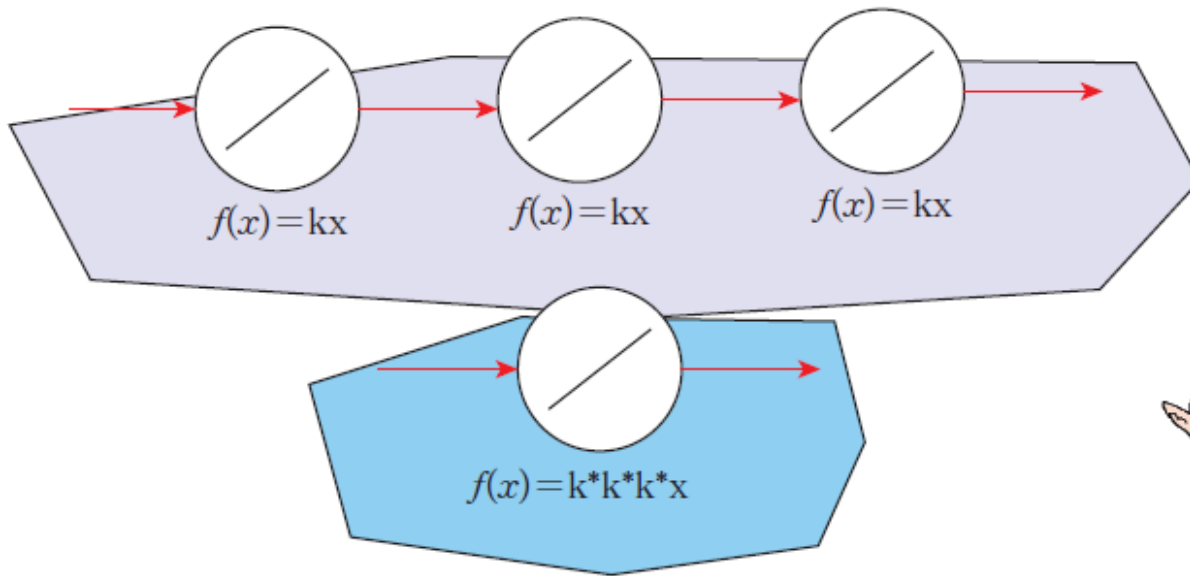


그림 6-3 많이 사용되는 활성화 함수



# 선형 레이어는 많아도 쓸모가 없다.



2개의 신경망은 동일한 기능을 수행합니다.



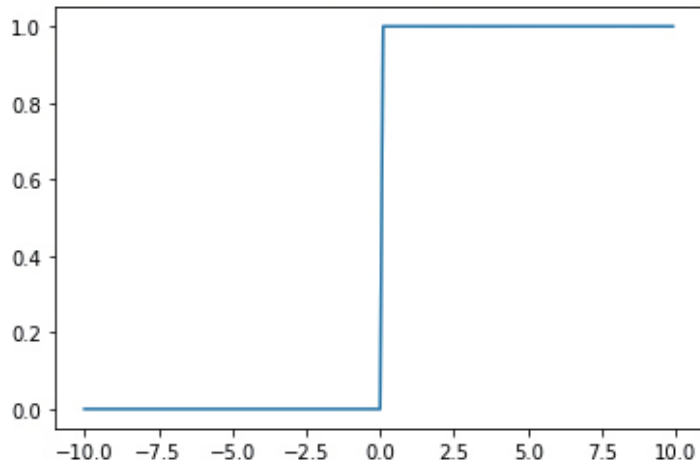
그림 6-4 선형 레이어는 아무리 많아도 하나의 레이어로 대체될 수 있다.



# 계단 함수

- 계단 함수는 입력 신호의 총합이 0을 넘으면 1을 출력하고, 그렇지 않으면 0을 출력하는 함수이다.

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

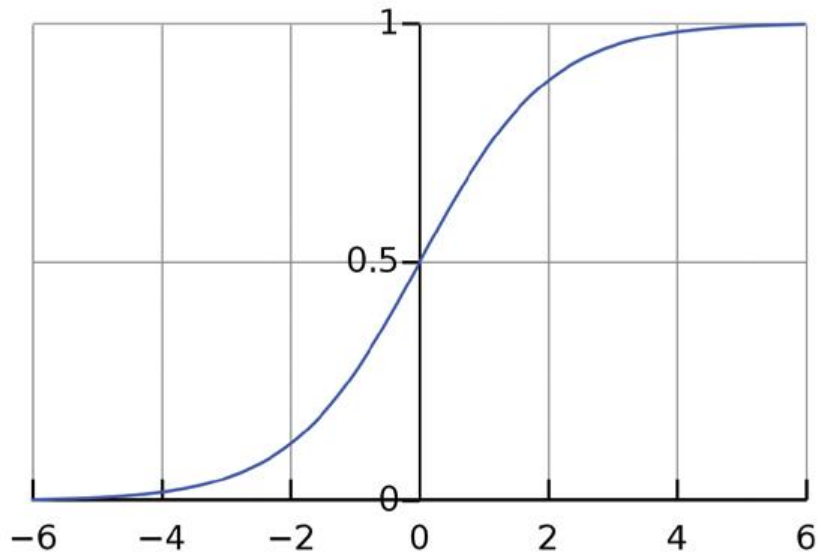




# 시그모이드 함수

- 1980년대부터 사용돼온 전통적인 활성화 함수이다. 시그모이드는 다음과 같이 S자와 같은 형태를 가진다.

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

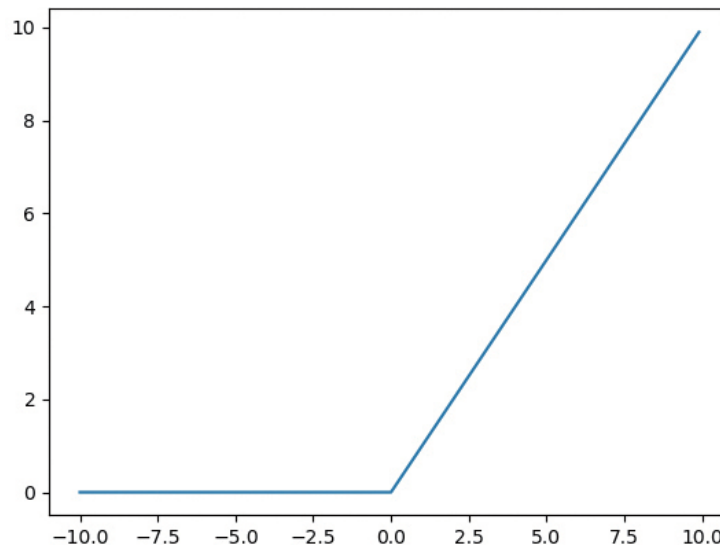




# ReLU 함수 (Rectified Linear Unit function)

- ReLU 함수는 최근에 가장 인기 있는 활성화 함수이다. ReLU 함수는 입력이 0을 넘으면 그대로 출력하고, 입력이 0보다 적으면 출력은 0이 된다.

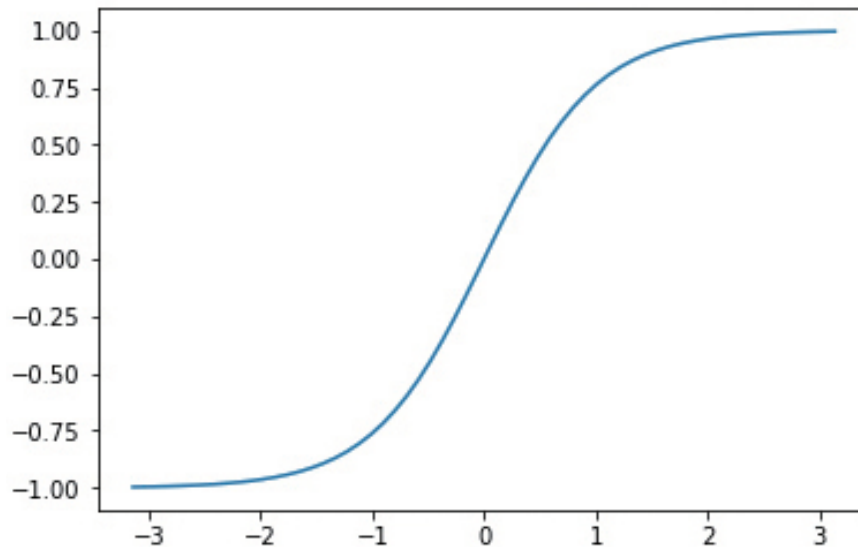
$$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$



# tanh 함수

- $\tanh()$  함수는 넘파이에서 제공하고 있다. 따라서 별도의 함수 작성은 필요하지 않다.  $\tanh()$  함수는 시그모이드 함수와 아주 비슷하지만 출력값이 -1에서 1까지이다.

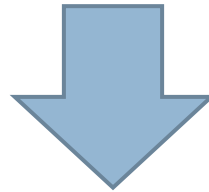
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1$$





# Lab: 파이썬으로 활성화 함수 구현하기 (act\_fun.py)

```
def step(x):  
    if x > 0.000001: return 1 # 부동 소수점 오차 방지  
    else             return 0
```



넘파이 배열을 받기 위하여 변경한다.

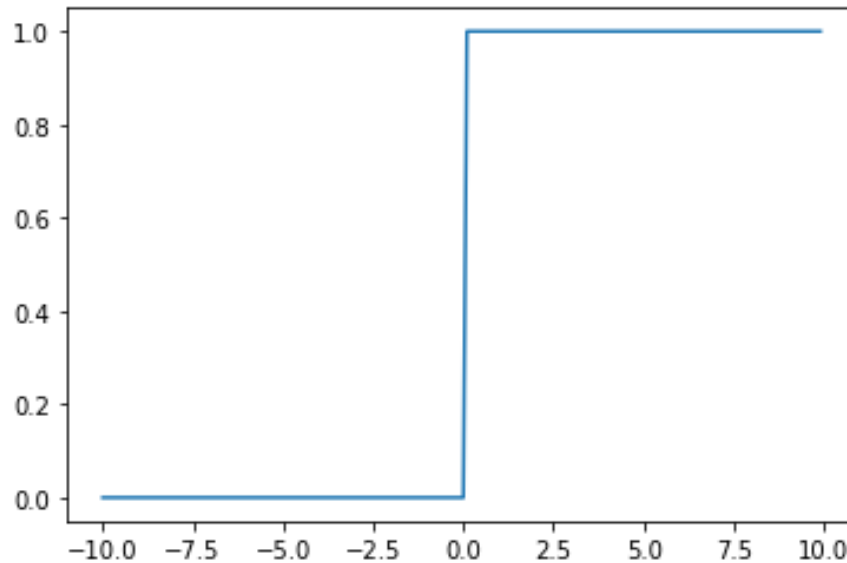
```
def step(x):  
    result = x > 0.000001 # True 또는 False  
    return result.astype(np.int) # 정수로 반환
```



# 그래프 그리기

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

```
x = np.arange(-10.0, 10.0, 0.1)
y = step(x)
plt.plot(x, y)
plt.show()
```



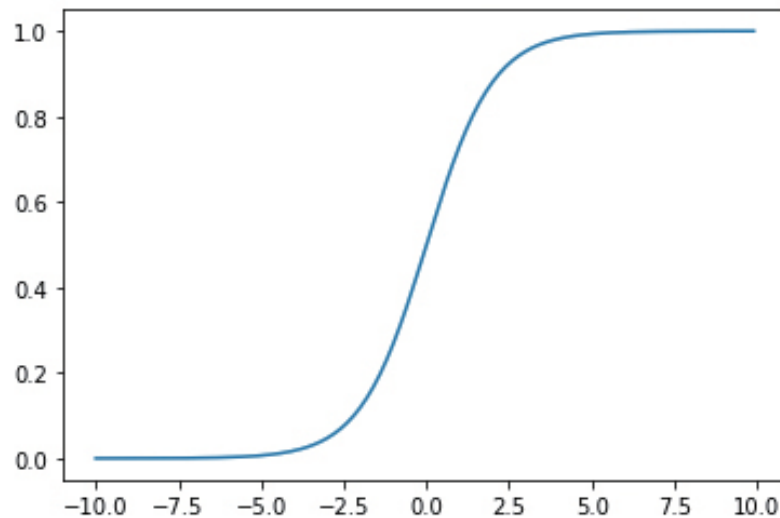


# 시그모이드 함수

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

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

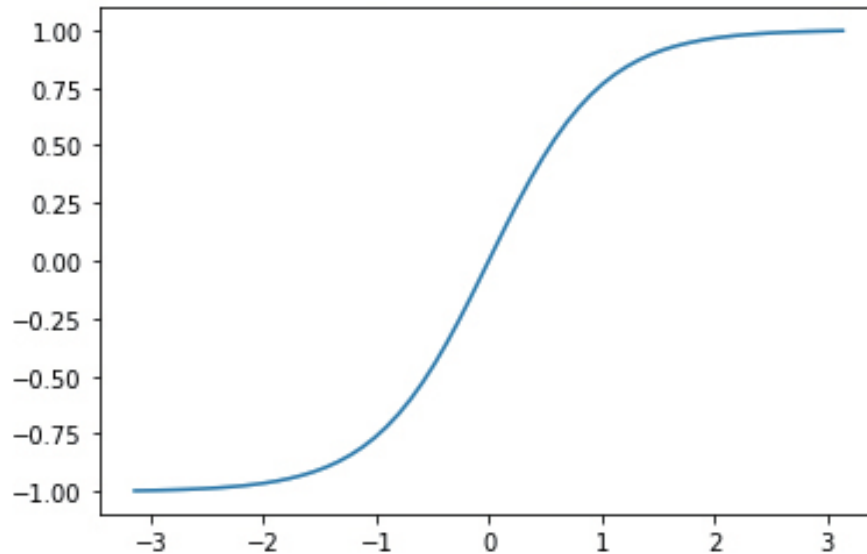
x = np.arange(-10.0, 10.0, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.show()
```





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

```
x = np.linspace(-np.pi, np.pi, 60)
y = np.tanh(x)
plt.plot(x, y)
plt.show()
```

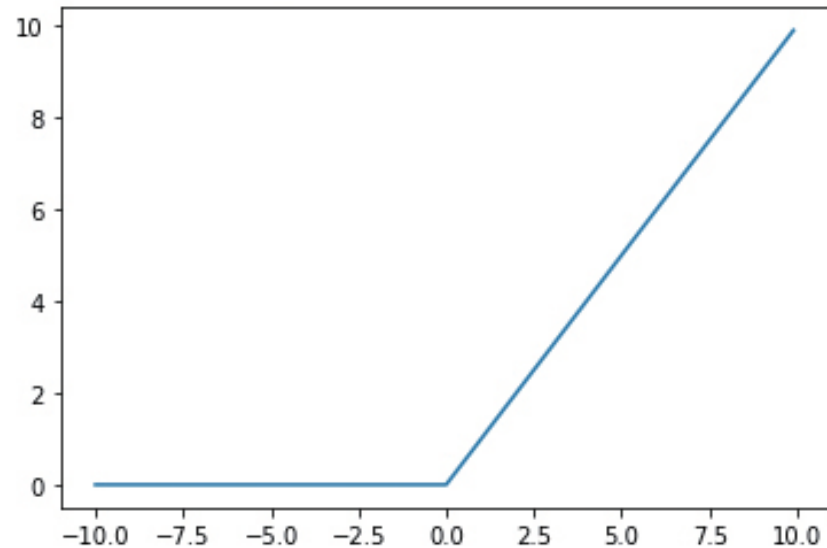


# ReLU 함수

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

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

```
x = np.arange(-10.0, 10.0, 0.1)
y = relu(x)
plt.plot(x, y)
plt.show()
```





# MLP의 순방향 패스

- 순방향 패스란 입력 신호가 입력층 유닛에 가해지고 이들 입력 신호가 은닉층을 통하여 출력층으로 전파되는 과정을 의미한다.

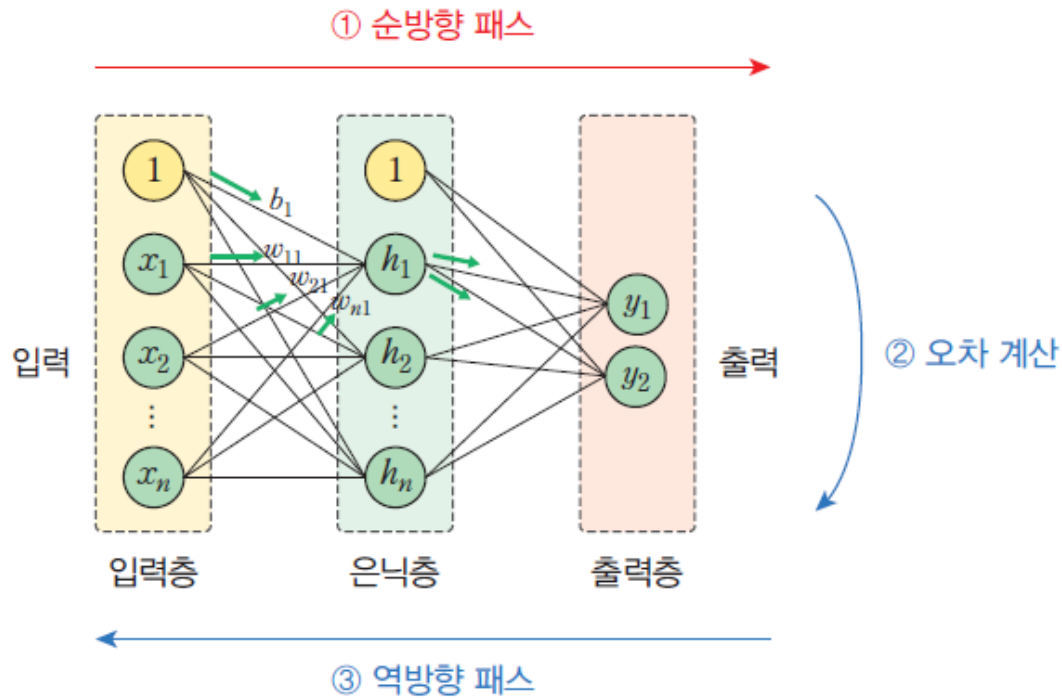
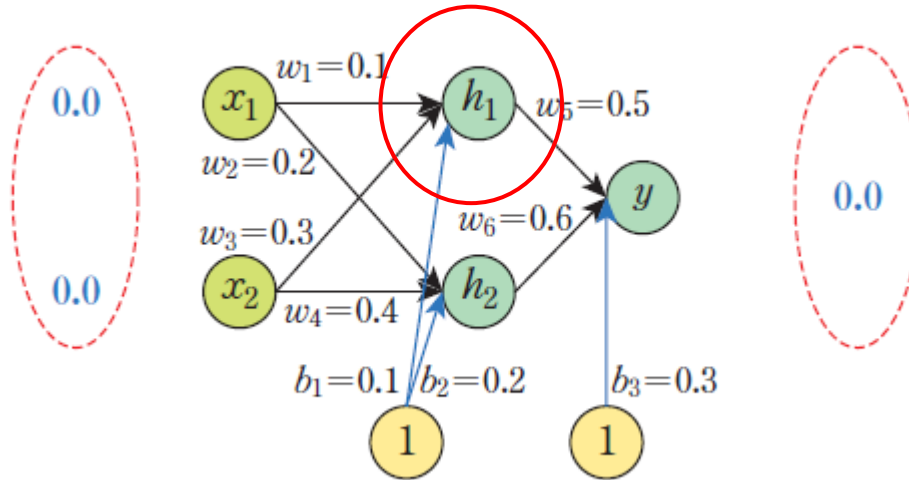


그림 6-5 순방향 패스



# 손으로 계산해보자.

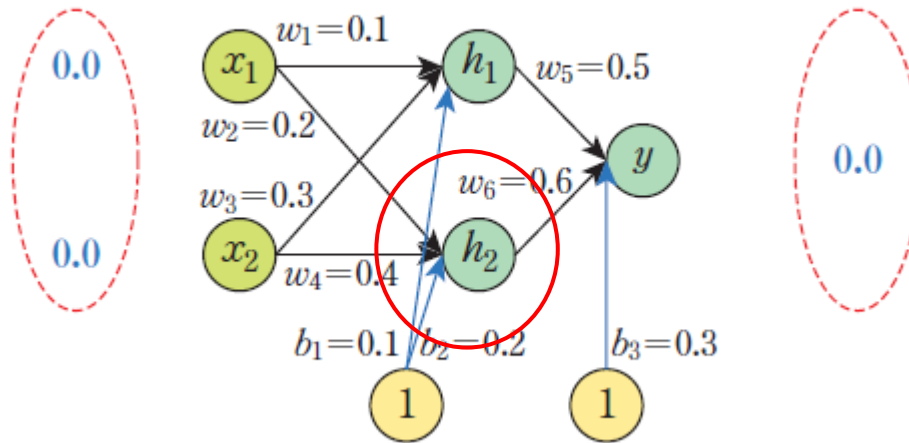


$$z_1 = w_1 * x_1 + w_3 * x_2 + b_1 = 0.1 * 0.0 + 0.3 * 0.0 + 0.1 = 0.1$$

$$a_1 = \frac{1}{1 + e^{-z_1}} = \frac{1}{1 + e^{-0.1}} = 0.524979$$



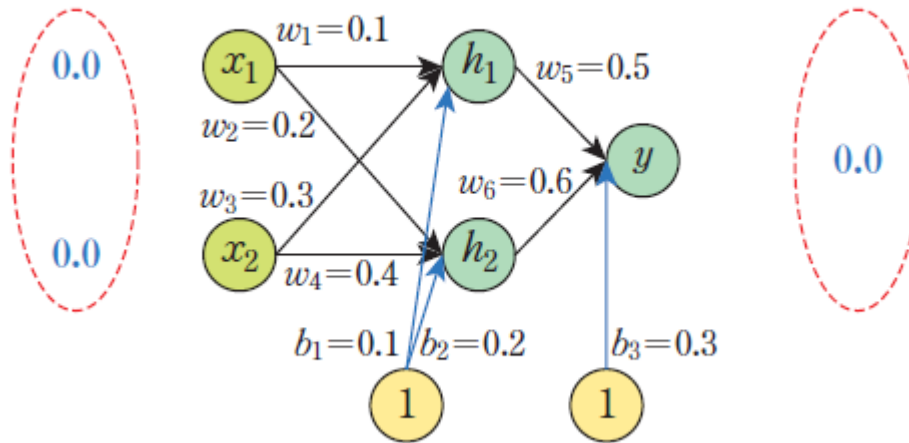
# 손으로 계산해보자.



$$a_2 = 0.549834$$



# 손으로 계산해보자.



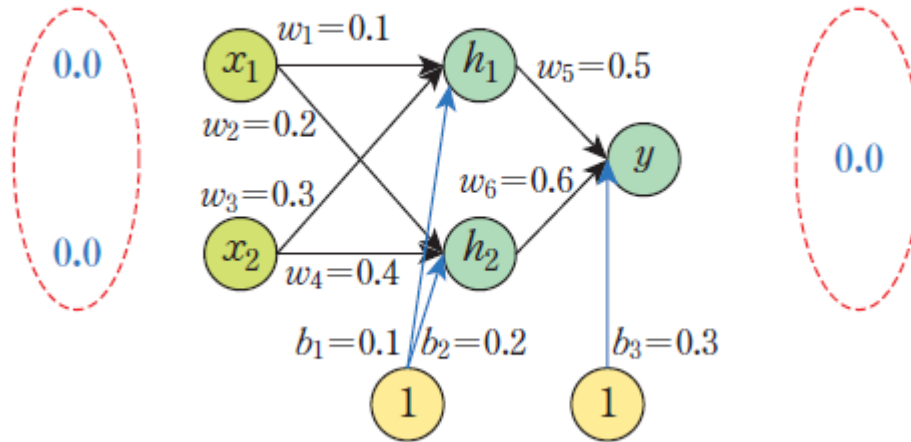
$$z_y = w_5 * a_1 + w_6 * a_2 + b_3$$
$$= 0.5 * 0.524979 + 0.6 * 0.549834 + 0.3 = 0.892389$$

$$a_y = \frac{1}{1 + e^{-z_y}} = \frac{1}{1 + e^{-0.892389}} = 0.709383$$

정답은 0이지만 신경망의 출력은 0.71 정도이다.  
오차가 상당함을 알 수 있다.



# 행렬로 표시해보자.



$$z_1 = w_1 * x_1 + w_3 * x_2 + b_1$$

$$z_2 = w_2 * x_1 + w_4 * x_2 + b_2$$

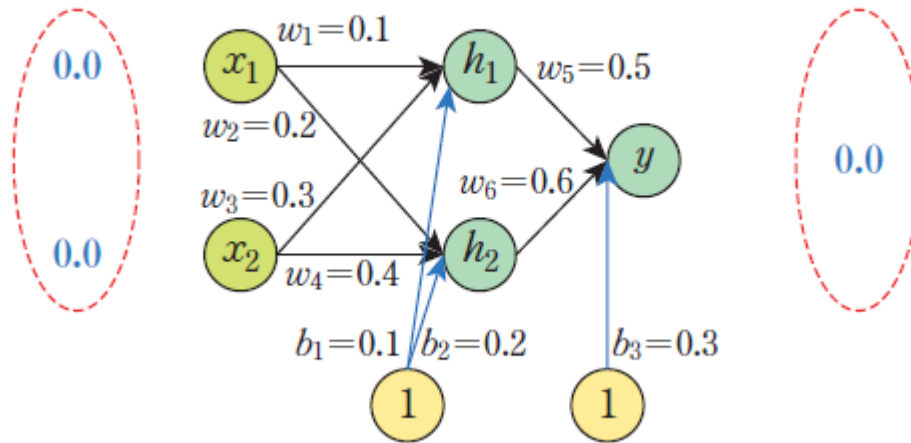


$$Z_1 = XW_1 + B_1$$

행렬로 표시할 수 있다.



# 행렬로 표시해보자



$$X = [x_1 \ x_2], \quad B_1 = [b_1 \ b_2], \quad Z_1 = [z_1 \ z_2]$$

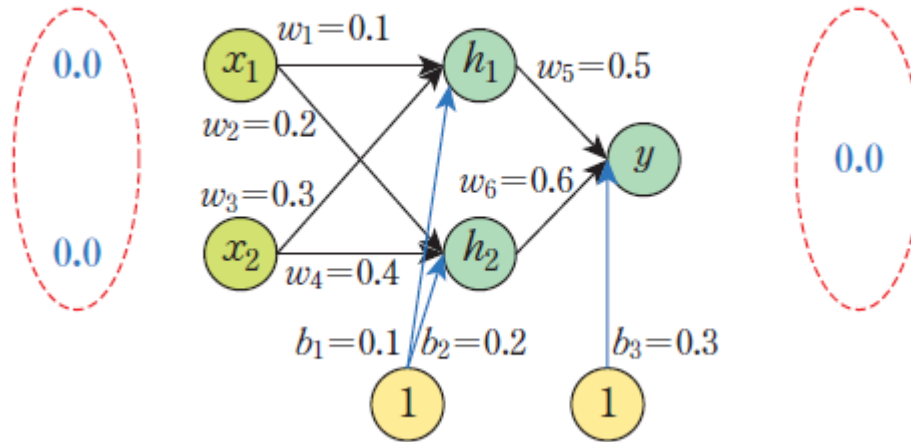
$$W_1 = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix}$$



$$Z_1 = [z_1 \ z_2] = XW_1 + B_1 = [x_1 \ x_2] \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} + [b_1 \ b_2]$$



# 행렬로 표시해보자



$$W_2 = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix}$$

$$Z_2 = A_1 W_2 + B_2 = [a_1 \ a_2] \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} + [b_3]$$

$$A_2 = [y] = f(Z_2)$$



# Lab: MLP 순방향 패스 (MLP1)

```
import numpy as np

# 시그모이드 함수
def actf(x):
    return 1/(1+np.exp(-x))

# 시그모이드 함수의 미분치
def actf_deriv(x):
    return x*(1-x)

# 입력유닛의 개수, 은닉유닛의 개수, 출력유닛의 개수
inputs, hidden, outputs = 2, 2, 1
learning_rate=0.2

# 훈련 샘플과 정답
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
T = np.array([[0], [1], [1], [0]])
```



# Lab: MLP 순방향 패스

```
W1 = np.array([[0.10, 0.20],
               [0.30, 0.40]])
W2 = np.array([[0.50], [0.60]])
B1 = np.array([0.1, 0.2])
B2 = np.array([0.3])

# 순방향 전파 계산
def predict(x):
    layer0 = x
    Z1 = np.dot(layer0, W1)+B1
    layer1 = actf(Z1)
    Z2 = np.dot(layer1, W2)+B2
    layer2 = actf(Z2)
    return layer0, layer1, layer2
```

```
# 입력을 layer0에 대입한다.
# 행렬의 곱을 계산한다.
# 활성화 함수를 적용한다.
# 행렬의 곱을 계산한다.
# 활성화 함수를 적용한다.
```



# Lab: MLP 순방향 패스

```
def test():  
    for x, y in zip(X, T):  
        x = np.reshape(x, (1, -1))      # x를 2차원 행렬로 만든다. 입력은 2차원이어야  
        한다.  
        layer0, layer1, layer2 = predict(x)  
        print(x, y, layer2)  
test()
```

```
[[0 0]] [1] [[0.70938314]]  
[[0 1]] [0] [[0.72844306]]  
[[1 0]] [0] [[0.71791234]]  
[[1 1]] [1] [[0.73598705]]
```

학습이 없으므로 난  
수만 출력된다.

# 손실 함수 계산

- 신경망에서 학습을 시킬 때는 실제 출력과 원하는 출력 사이의 오차를 이용한다. 오차를 계산하는 함수를 손실함수(loss function)라고 한다.

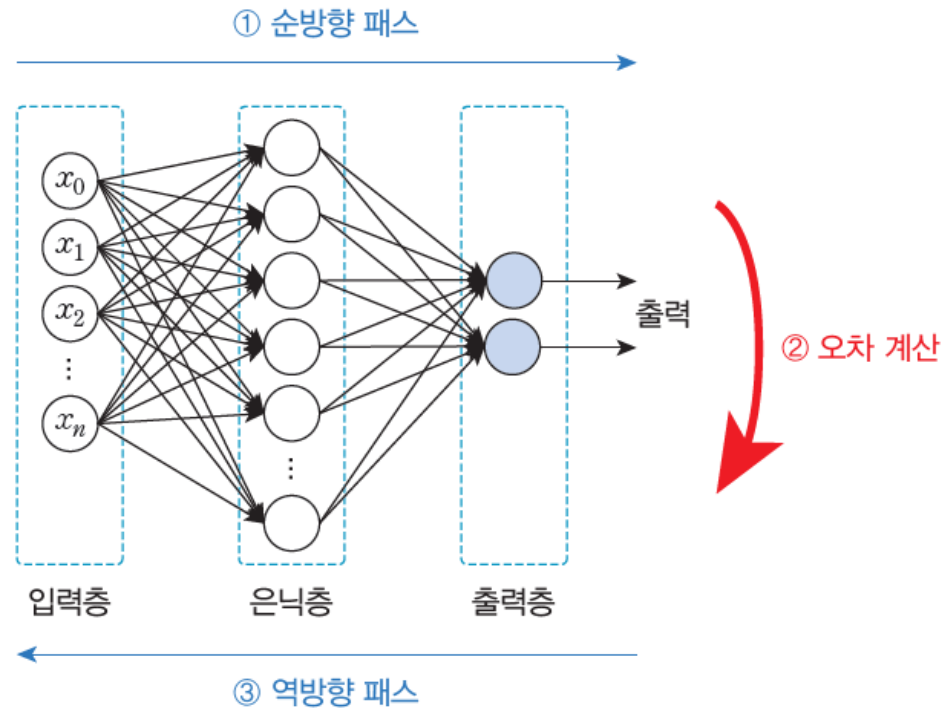
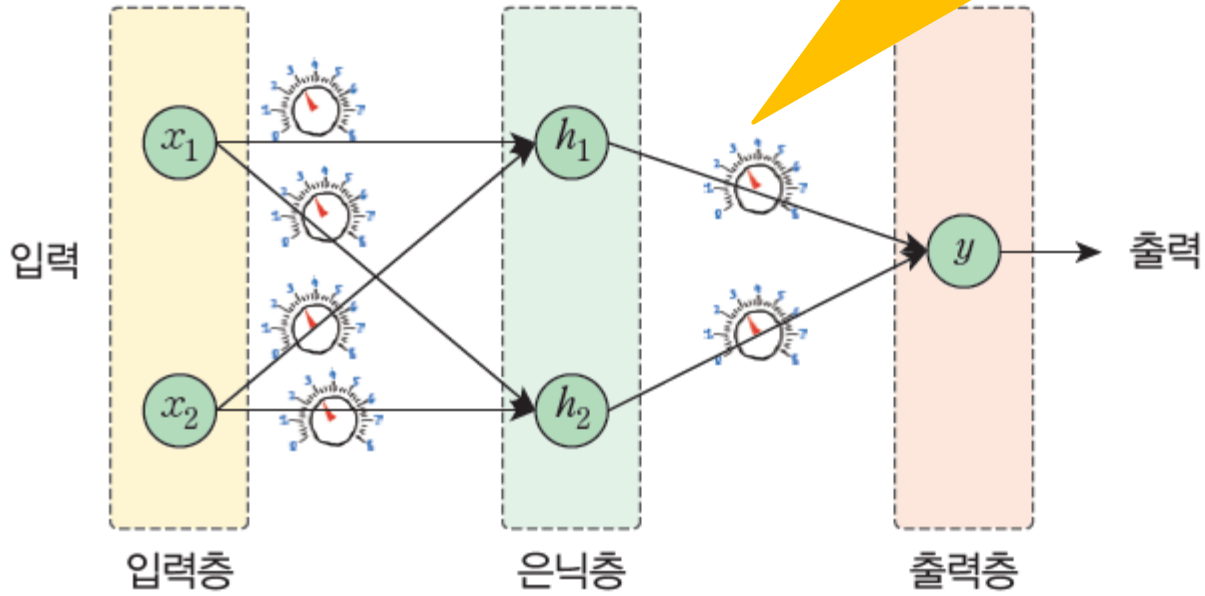


그림 6-6 오차 계산



# 가중치 == 다이얼

가중치를 조절한다는 것은 스피커에서 나는 소리를 들으면서 튜너 다이얼을 돌리는 것과 같다.



- 신경망에서도 학습의 성과를 나타내는 지표가 있어야 한다. 이것을 손실함수(loss function)이라고 한다

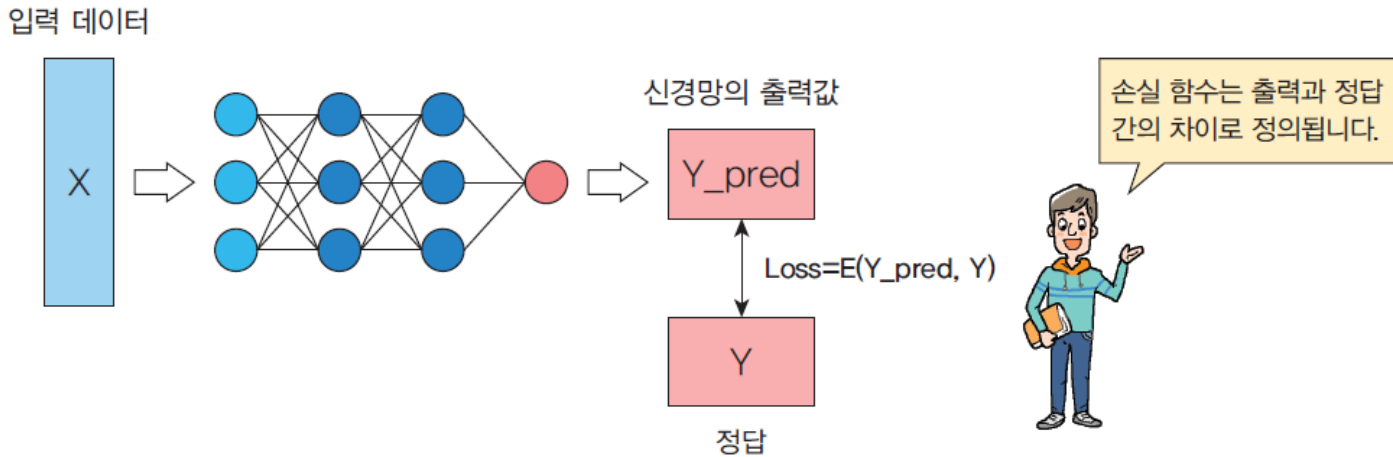


그림 6-7 손실함수의 정의

# 평균 제곱 오차 (MSE)

- 예측값과 정답 간의 평균 제곱 오차

$$E(w) = \frac{1}{2} \sum_i (y_i - t_i)^2$$

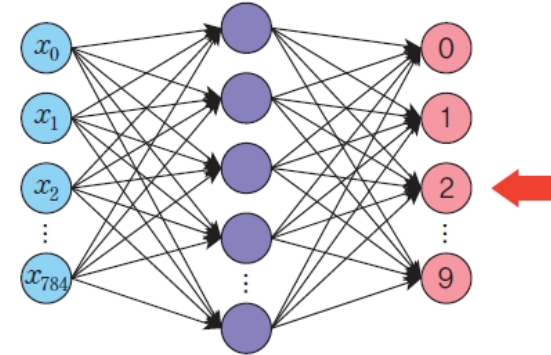


그림 6-8 MNIST 숫자 이미지를 분류하는 신경망

```
>>> y = np.array([ 0.0, 0.0, 0.8, 0.1, 0.0, 0.0, 0.0, 0.1, 0.0, 0.0 ])
>>> target = np.array([ 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ])

>>> def MSE(target, y):
    return 0.5 * np.sum((y-target)**2)

>>> MSE(target, y)
0.029999999999999992
```



# 예측값과 정답이 많이 차이 나는 경우

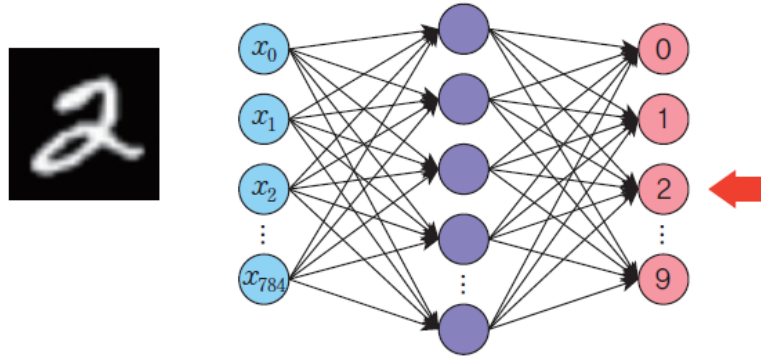


그림 6-8 MNIST 숫자 이미지를 분류하는 신경망

```
>>> y = np.array([ 0.9, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ] )
>>> target = np.array([ 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ] )

>>> def MSE(target, y):
    return 0.5 * np.sum((y-target)**2)
>>> MSE(target, y)
0.81
```

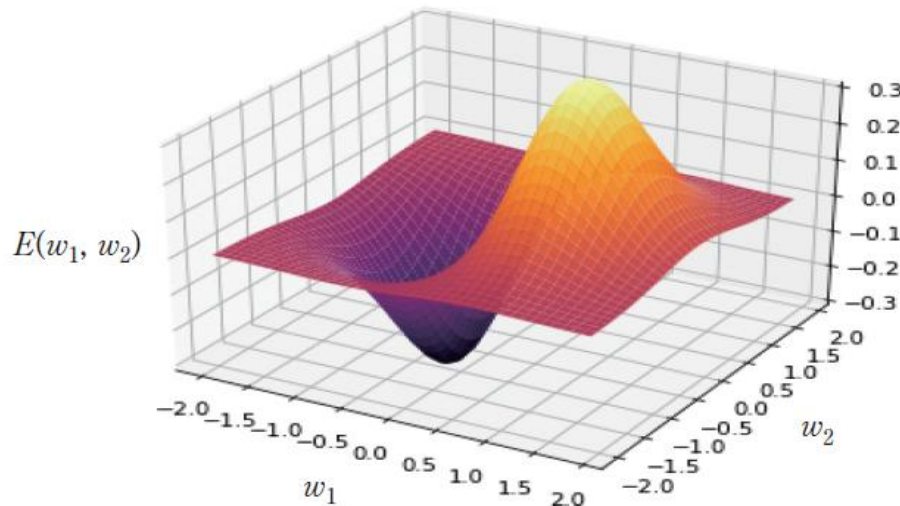


# 경사하강법(Gradient Descent)

- 역전파 알고리즘은 신경망 학습 문제를 최적화 문제(optimization)로 접근한다. 우리는 손실함수 값을 최소로 하는 가중치를 찾으면 된다.

$$W^* = \underset{W}{\operatorname{argmin}} E(W)$$

Augustin-Louis Cauchy(수학자, 1847년)



옆의 평면에서 가장 낮은 값을 찾으면 됩니다.

그림 6-9 경사 하강법

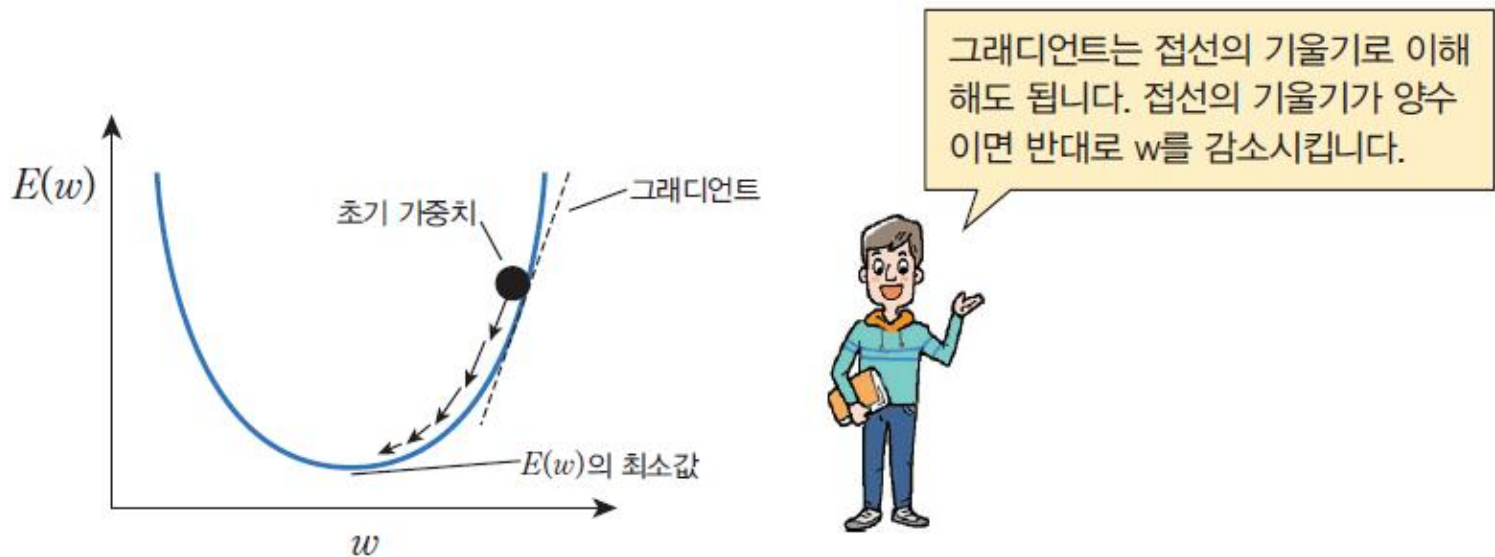
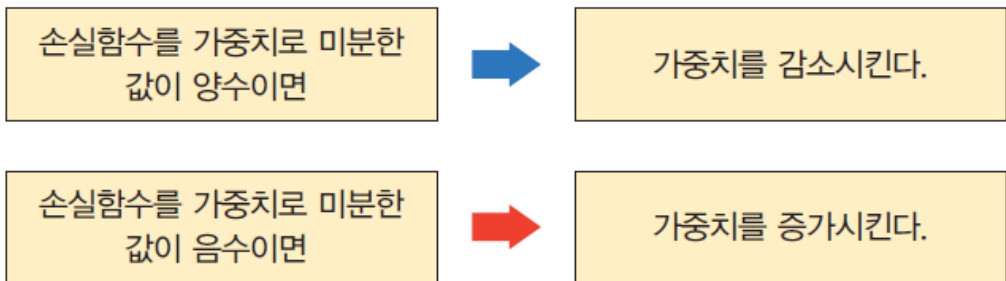


그림 6-11 경사 하강법





# 선형 회귀에서 경사하강법

$$Loss(W, b) = \frac{1}{n} \sum_{i=1}^n ((Wx_i + b) - y_i)^2$$

$$\frac{\partial Loss(W, b)}{\partial W} = \frac{1}{n} \sum_{i=1}^n 2((Wx_i + b) - y_i)(x_i) = \frac{2}{n} \sum_{i=1}^n x_i((Wx_i + b) - y_i)$$

$$\frac{\partial Loss(W, b)}{\partial b} = \frac{2}{n} \sum_{i=1}^n ((Wx_i + b) - y_i)$$

$$W = W - 0.01 * \frac{\partial Loss}{\partial W}$$

$$b = b - 0.01 * \frac{\partial Loss}{\partial b}$$



# 경사하강법의 종류

- 확률적 경사 하강법(Stochastic Gradient Descent, SGD)

랜덤성, 확률이 포함된 개념

전체 데이터 셋이 아닌 random하게 선택된 샘플링을 통해 업데이트  
(하나의 데이터 포인트)

매번 다른 샘플을 사용하기 때문에 기울기 추정 시 노이즈가 포함될 수 있음

속도가 빠르다



# 경사하강법의 종류

- 미니 배치 경사 하강법(Mini-batch Gradient Descent)

미니 배치 사이즈의 데이터를 활용하여 업데이트 하는 방식  
BGD와 SGD의 중간 형태

일정한 크기의 미니 배치 사이즈를 사용하기 때문에

- 속도가 빠르다
- 안정적임



# 경사하강법의 종류

- 배치 경사 하강법 (Batch Gradient Descent)

한 번에 모든 데이터를 사용하여 업데이트 함

안정적이지만 속도가 느리고 메모리가 많이 사용됨

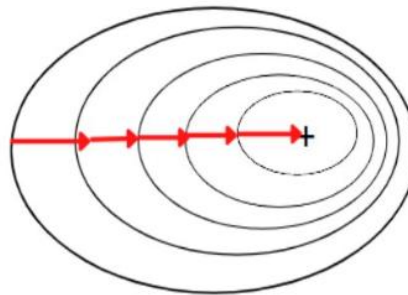


# 경사하강법의 종류

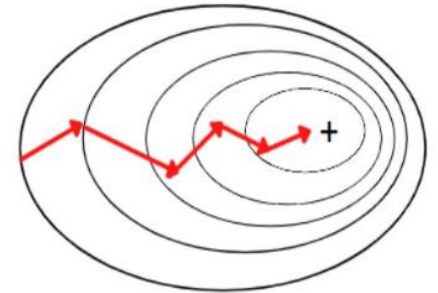


- Batch gradient descent
- Mini-batch gradient Descent
- Stochastic gradient descent

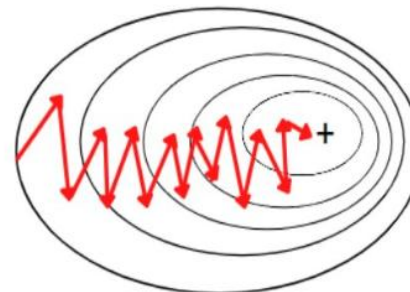
**Batch Gradient Descent**



**Mini-Batch Gradient Descent**



**Stochastic Gradient Descent**

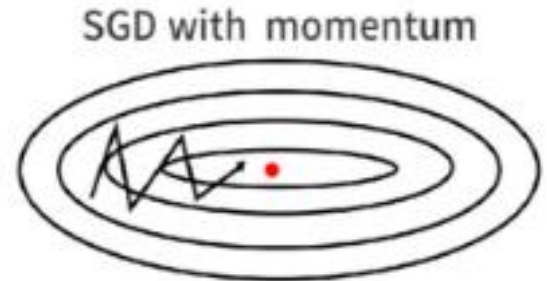
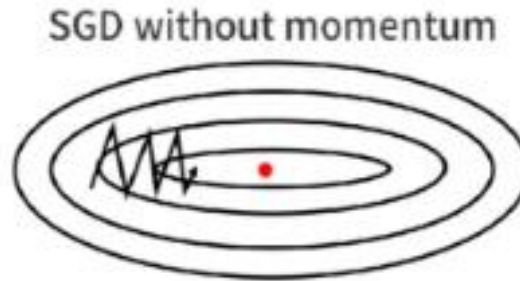
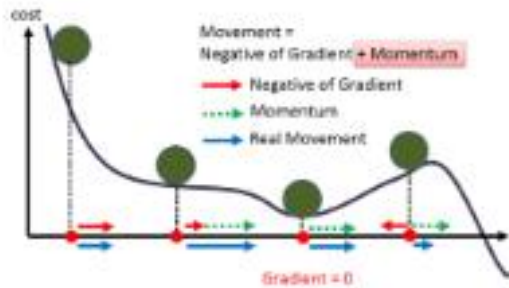


# 경사하강법의 종류

- Momentum

‘관성’을 뜻함

이전 단계의 기울기를 고려하여 현재 단계의 기울기에 관성을 부여  
수렴 속도를 높이고 지역 최저점에 빠지지 않도록 함.





# 경사하강법의 종류

- Adagrad

Adaptation 즉 조정을 이용하는 방식

학습에 따라 많이 변화하고 자주 갱신되는 파라미터의 학습률은 감소시키고, 덜 자주 갱신되는 파라미터의 학습률은 증가시키는 기법

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

for numerical stability

Sum of gradient squares

지금까지 해당 파라미터가 얼마나 변화했는지를 각각 추적하면서 (변화의 제곱합 : sum of gradient squares) 학습률을 조정

Adagrad의 경우에, 학습이 반복 누적되고 파라미터 변화의 제곱합인 G가 무한대로 커지게 되면 학습이 매우 더뎠게 된다.



# 경사하강법의 종류

- Adadelta

Adagrad 의 단점을 개선한 알고리즘

학습률 감소 문제를 해결하기 위해 제한된 **moving average 윈도우 사이즈(window)**을 사용하여 매개변수의 학습률을 조정

과거의 모든 기울기 제곱을 누적하지 않고 **학습률이 아닌 이동평균된** 제곱 그래디언트의 제곱근(EMA of gradient squares)의 비율을 사용하여 학습률을 동적으로 조절한다. (learning rate이 없음)

EMA of gradient squares

$$G_t = \gamma G_{t-1} + (1 - \gamma)g_t^2$$

$$W_{t+1} = W_t - \frac{\sqrt{H_{t-1} + \epsilon}}{\sqrt{G_t} + \epsilon} g_t$$

EMA of difference squares

$$H_t = \gamma H_{t-1} + (1 - \gamma)(\Delta W_t)^2$$

There is **no learning rate** in Adadelta.



# 경사하강법의 종류

- RMSprop (Root Mean Square Propagation)

Geoff Hinton이 강의에서 제안한 기법.

지수 이동 평균을 사용하여 최근 기울기들을 반영하는 알고리즘

\*Adagrad의 안정적인 학습을 가능하게 함. [학습률 감소 문제를 해결]

\*Adadelta와 다르게 learning rate을 나타내는 [stepsize가 존재]

EMA of gradient squares

$$G_t = \gamma G_{t-1} + (1 - \gamma)g_t^2$$
$$W_{t+1} = W_t - \frac{\eta}{\sqrt{G_t} + \epsilon} g_t$$

← stepsize

제곱 그래디언트의 제곱근(EMA of gradient squares)



# 경사하강법의 종류

- Adam (Adaptive Moment Estimation)

가장 무난하고 많이 사용하는 성능 좋은 optimizer 중 하나

Momentum 과 RMSprop을 결합한 최적화 기법

- 1) Gradient square의 크기에 따라서 adaptive하게 학습률을 변경
- 2) 이전 Gradient 정보에 해당하는 momentum

Momentum

EMA of gradient squares

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$W_{t+1} = W_t - \frac{\eta}{\sqrt{v_t} + \epsilon} \frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t} m_t$$



# Lab: 경사하강법의 실습

- 손실 함수  $y = (x - 3)^2 + 10$
- 그래디언트:  $y' = 2x - 6$

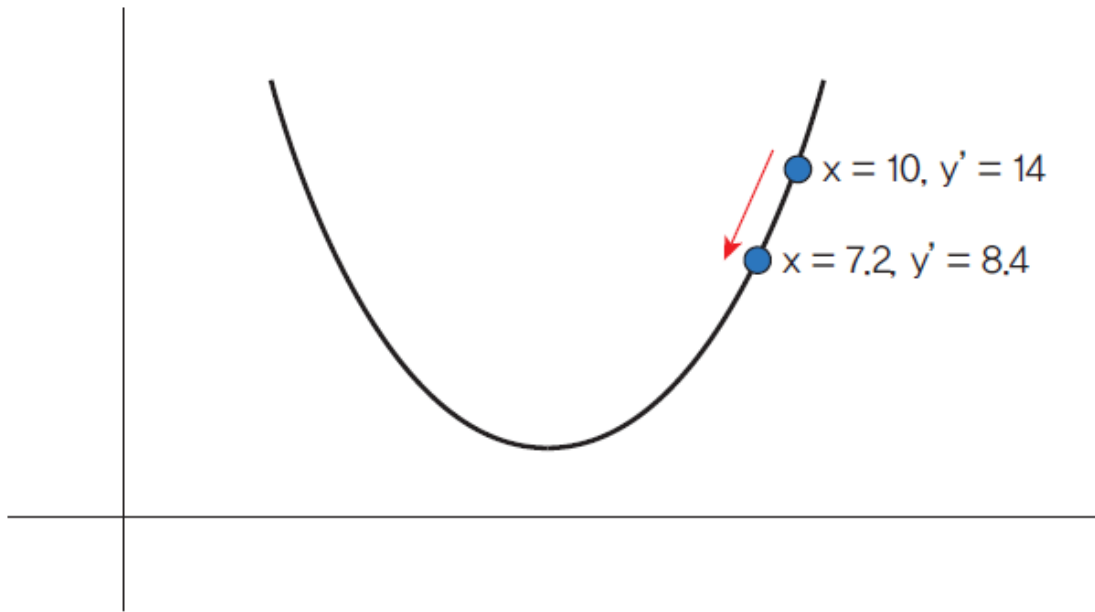


그림 6-12 그래디언트의 계산



# 경사 하강법 프로그래밍

```
x = 10
learning_rate = 0.2
precision = 0.00001
max_iterations = 100

# 손실함수를 람다식으로 정의한다.
loss_func = lambda x: (x-3)**2 + 10

# 그래디언트를 람다식으로 정의한다. 손실함수의 1차 미분값이다.
gradient = lambda x: 2*x-6

# 그래디언트 강하법
for i in range(max_iterations):
    x = x - learning_rate * gradient(x)
    print("손실함수값(", x, ")=", loss_func(x))

print("최소값 = ", x)
```



손실함수값( 7.199999999999999 )= 27.639999999999993

손실함수값( 5.52 )= 16.350399999999997

손실함수값( 4.512 )= 12.286143999999998

손실함수값( 3.907199999999996 )= 10.82301184

손실함수값( 3.54432 )= 10.2962842624

...

손실함수값( 3.0000000000000004 )= 10.0

최소값 = 3.0000000000000004



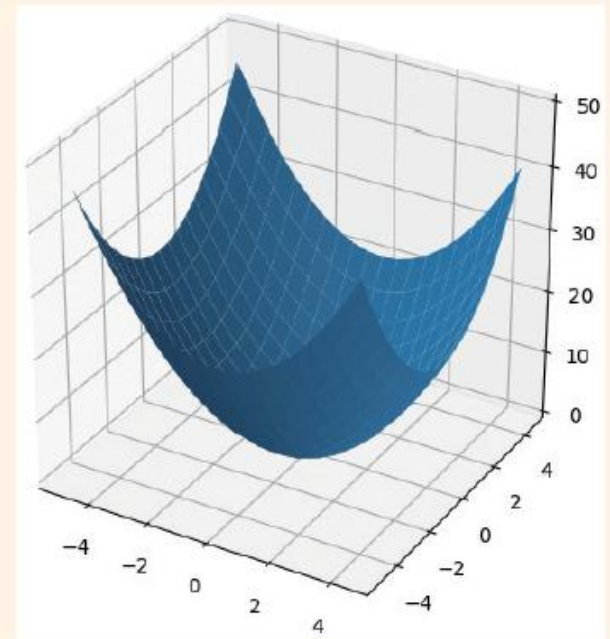
# Lab: 2차원 그래디언트 시각화

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
```

```
x = np.arange(-5, 5, 0.5)
y = np.arange(-5, 5, 0.5)
X, Y = np.meshgrid(x, y) # 참고 박스
Z = X**2 + Y**2          # 넘파이 연산
```

```
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='3d')
```

```
# 3차원 그래프를 그린다.
ax.plot_surface(X, Y, Z)
plt.show()
```





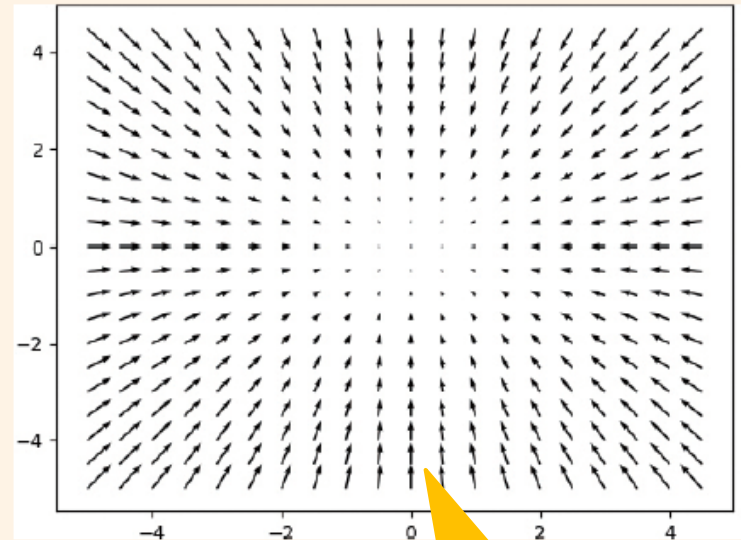
# Lab: 2차원 그래디언트 시각화

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

```
x = np.arange(-5,5,0.5)  
y = np.arange(-5,5,0.5)  
X, Y = np.meshgrid(x,y)  
U = -2*X  
V = -2*Y
```

그래디언트의 음수

```
plt.figure()  
Q = plt.quiver(X, Y, U, V, units='width')  
plt.show()
```



화살표가 최소값을  
가리키고 있음을 알  
수 있다.



# 역전파 학습 알고리즘

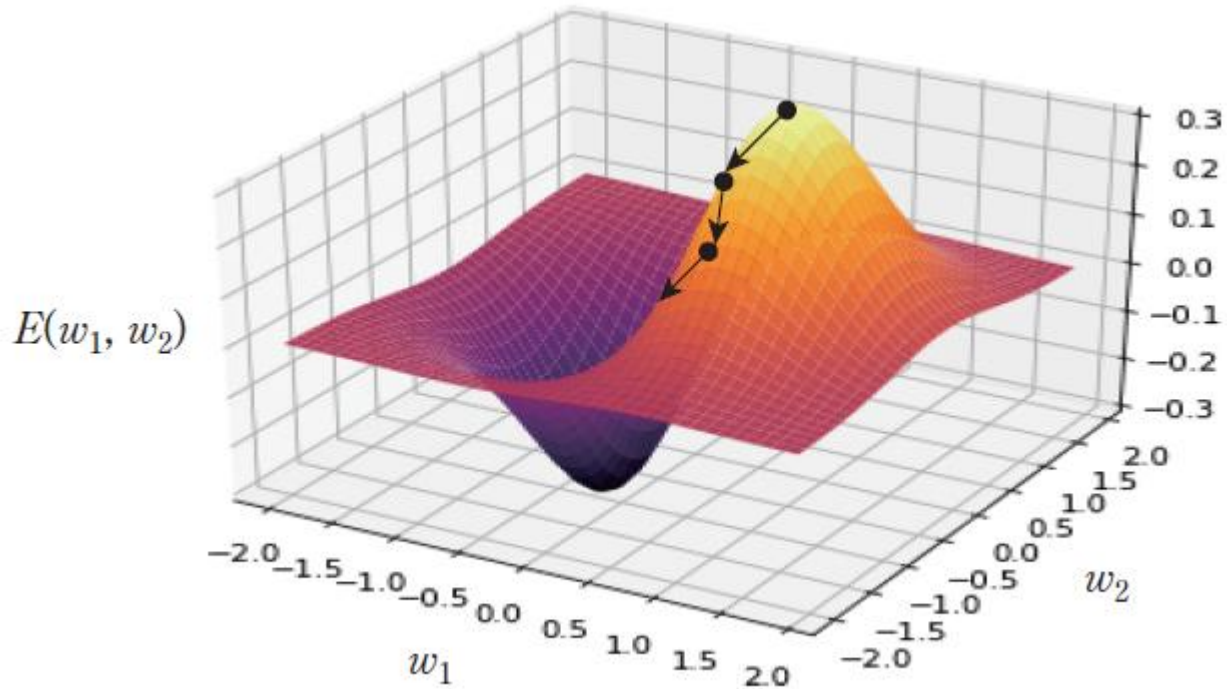
- 역전파 알고리즘은 입력이 주어지면 순방향으로 계산하여 출력을 계산한 후에 실제 출력과 우리가 원하는 출력 간의 오차를 계산한다.
- 이 오차를 역방향으로 전파하면서 오차를 줄이는 방향으로 가중치를 변경한다.

$$w(t+1) = w(t) - \eta \frac{\partial E}{\partial w}$$

- ① 가중치와 바이어스를 0부터 1 사이의 난수로 초기화한다.
- ② 수렴할 때까지 모든 가중치에 대하여 다음을 반복한다.
- ③ 손실함수  $E$ 의 그래디언트  $\partial E / \partial w$ 을 계산한다.
- ④  $w(t+1) = w(t) - \eta \frac{\partial E}{\partial w}$



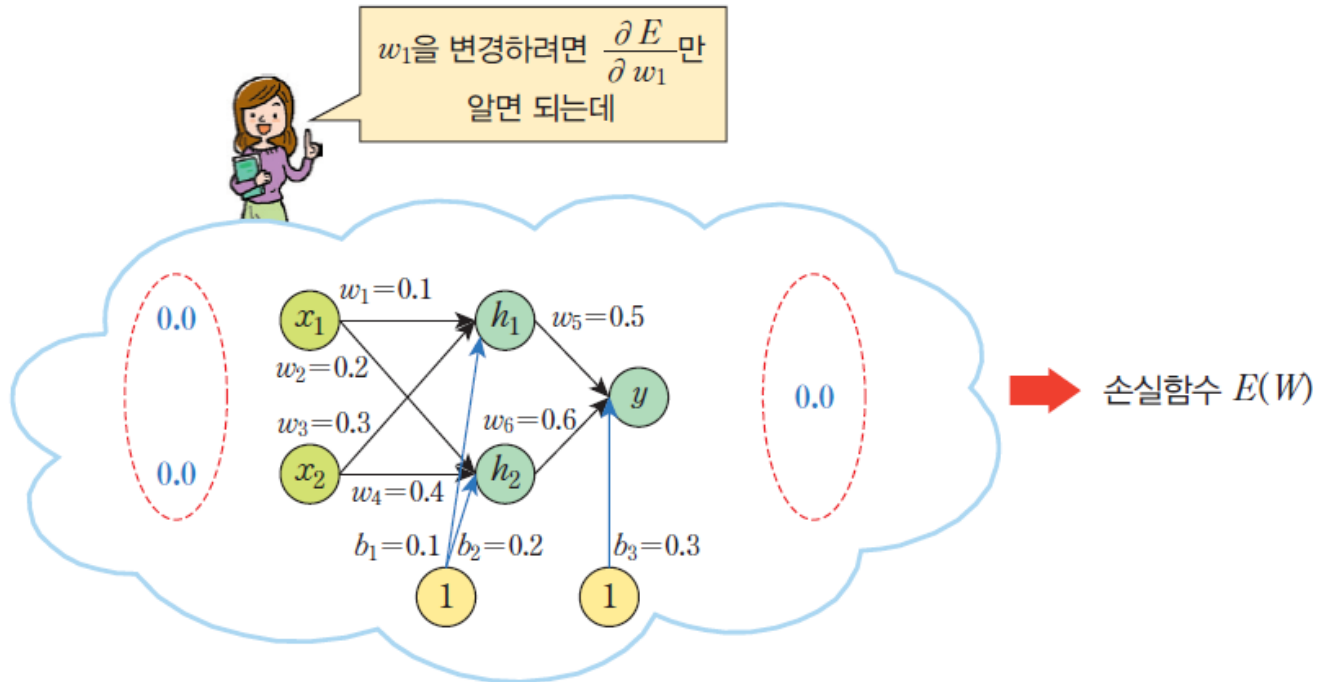
# 역전파 학습 알고리즘





# 역전파 알고리즘의 유도 (생략 가능)

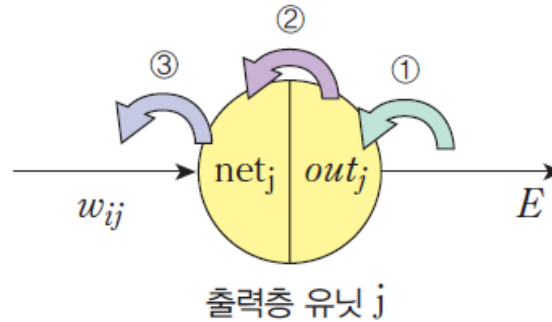
- 미분의 체인룰을 이용하여 유도가 가능하다.



# 출력층 유닛의 경우

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial out_j} \frac{\partial out_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

①                      ②                      ③



①  $\frac{\partial E}{\partial out_j} = \frac{\partial}{\partial out_j} \sum \frac{1}{2} (target_k - out_k)^2 = out_j - target_j$

유닛의 출력값 변환에 따른 오차의 변화율이다.

②  $\frac{\partial out_j}{\partial net_j} = \frac{\partial f(net_j)}{\partial net_j} = f'(net_j)$

입력값의 변화에 따른 유닛 j의 출력 변화율이다.  
활성화 함수의 미분값이다.

③  $\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left( \sum_{k=0}^n w_{kj} out_k \right) = \frac{\partial}{\partial w_{ij}} w_{ij} out_i = out_i$

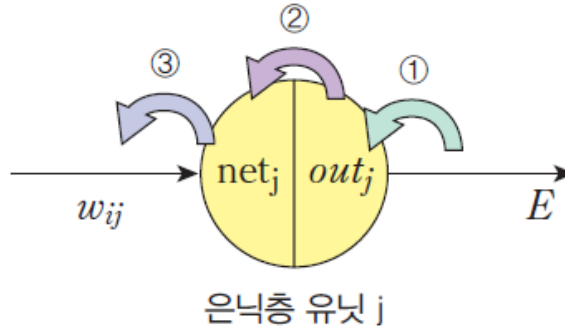
가중치의 변화에 따른 net\_j의 변화율이라고 할 수 있다.

$$\therefore \frac{\partial E}{\partial w_{ij}} = ① \times ② \times ③ = (out_j - target_j) \times f'(net_j) \times out_i$$

# 은닉층 유닛의 경우

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial out_j} \frac{\partial out_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

①                      ②                      ③

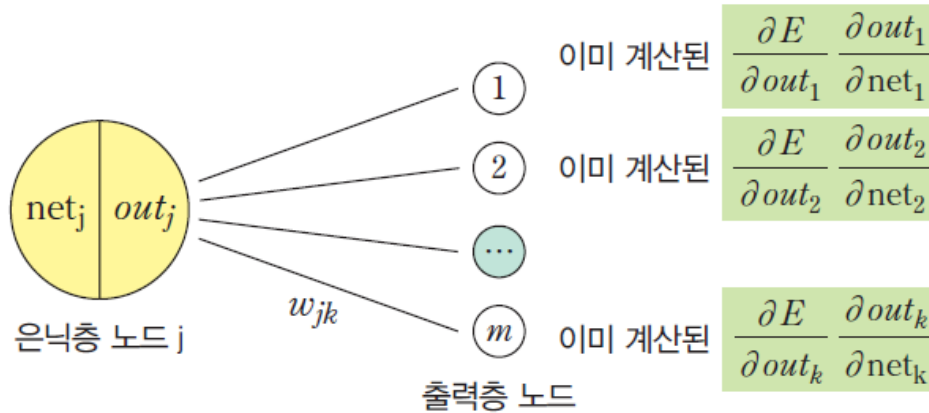


그런데 은닉층 유닛의 오차는 어떻게 계산하지?



$$\frac{\partial E}{\partial out_j} = \sum_{k \in L} \left( \frac{\partial E}{\partial out_k} \frac{\partial out_k}{\partial net_k} \frac{\partial net_k}{\partial out_j} \right) = \sum_{k \in L} \left( \frac{\partial E}{\partial out_k} \frac{\partial out_k}{\partial net_k} w_{jk} \right)$$

이것도 체인룰이다!





- $\frac{\partial E}{\partial out_k} \frac{\partial out_k}{\partial net_k}$  은 여러 문헌에서 델타라는 이름으로 불리는 값이다.
- 가중치가 보는 유닛 k에서의 “오차”라고 생각해도 된다.
- 이 델타가 신경망을 통하여 역전파된다.



# 역전파 알고리즘 정리

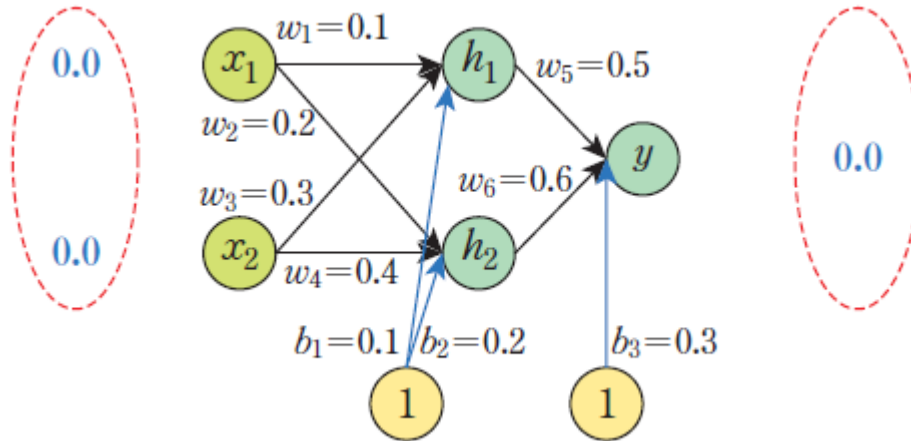
- 결론적으로 그래디언트는 델타에 유닛의 출력값을 곱하면 구할 수 있다.
- 델타는 신경망의 레이어에 따라서 다음과 같이 구분하여서 계산할 수 있다.

$$\frac{\partial E}{\partial w_{ij}} = \delta_j out_i \quad \text{여기서} \quad \delta_j = \begin{cases} (out_j - t_j) f'(net_j) & j \text{가 출력층 유닛이면} \\ \left( \sum_k w_{jk} \delta_k \right) f'(net_j) & j \text{가 은닉층 유닛이면} \end{cases}$$

$out_i$  : 이전 층 유닛  $i$ 의 출력값



# 역전파 알고리즘 손으로 계산해보자.



- 순방향 패스

$$\begin{aligned} \text{net}_y &= w_5 * \text{out}_{h1} + w_6 * \text{out}_{h2} + b_3 \\ &= 0.5 * 0.524979 + 0.6 * 0.549834 + 0.3 = 0.89239 \\ \text{out}_y &= \frac{1}{1 + e^{-\text{net}_y}} = \frac{1}{1 + e^{-0.89239}} = 0.709383 \end{aligned}$$

아저씨 참조!

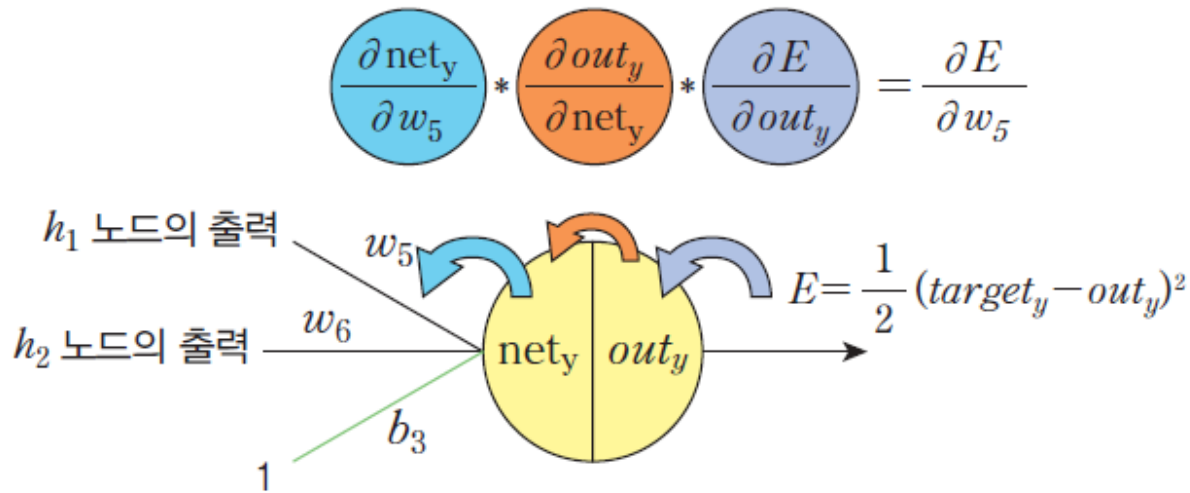


# 역전파 알고리즘 손으로 계산해보자.

- 총오차 계산

$$E = \frac{1}{2}(target_y - out_y)^2 = \frac{1}{2}(0.00 - 0.709383)^2 = 0.251612$$

- $\frac{\partial E}{\partial w_5}$ 만 계산해보자.





# 경사하강법 적용

- ①  $\frac{\partial E}{\partial out_y} = 2 * \frac{1}{2}(target_y - out_y)^{2-1} * (-1) = (out_y - target_y)$   
 $= (0.709383 - 0.00) = 0.709383$
- ②  $\frac{\partial out_y}{\partial net_y} = f'(out_y) = out_y * (1 - out_y) = 0.709383 * (1 - 0.709383) = 0.206158$
- ③  $net_y = w_5 * out_{h1} + w_6 * out_{h2} + b_3 * 1$   
 $\frac{\partial net_y}{\partial w_5} = 1 * out_{h1} + 0 + 0 = 0.524979$



$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial out_y} \frac{\partial out_y}{\partial net_y} \frac{\partial net_y}{\partial w_5}$$
$$= 0.709383 * 0.206158 * 0.524979 = 0.076775$$

$$w_5(t+1) = w_5(t) + \eta * \frac{\partial E}{\partial w_5} = 0.5 - 0.2 * 0.076775 = 0.484645$$



# 입력값 -> 출력값의 가중치와 바이어스

$$w_5(t+1) = w_5(t) + \eta * \frac{\partial E}{\partial w_5} = 0.5 - 0.2 * 0.076775 = 0.484645$$

$$w_6(t+1) = 0.583918$$

$$b_3(t+1) = 0.270750$$

가중치도 낮아지게 된다. 현재 우리가 얻는 출력값은 0이기 때문이다.

바이어스는 기준 값 보다 낮아지게 된다. 따라서 다음 번에는 유닛의 출력을 더 낮게 만들 것이다. 현재 우리가 얻는 출력값은 0이기 때문이다.



# 이러춤 > 인니춤의 가중치와 바이어스

$$w_1(t+1) = w_1(t) + \eta * \frac{\partial E}{\partial w_1} = 0.10 - 0.2 * 0.0 = 0.10$$

$$w_2(t+1) = 0.2, w_3(t+1) = 0.3, w_4(t+1) = 0.4$$

이러춤이 0이어서 가중치는 변경되지 않았다 (이것은 퍼셉트론과 유사하다. 입력이 0이면 가중치를 아무리 바꿔도 무슨 소용인가?).

$$b_1(t+1) = 0.096352, b_2(t+1) = 0.195656$$

이런 경우에는 바이어스가 큰 역할을 한다 (이래서 바이어스는 반드시 있어야 한다) 바이어스는 기존 값 보다 낮아지게 된다. 따라서 다음 번에는 유닛의 출력을 더 낮게 만들 것이다. 현재 우리가 얻는 출력값은 0이기 때문이다.

$$E = \frac{1}{2}(target - out_y)^2 = \frac{1}{2}(0.00 - 0.709383)^2 = 0.251612$$

경사하강법 1번 적용

$$E = \frac{1}{2}(target - out_y)^2 = \frac{1}{2}(0.00 - 0.699553)^2 = 0.244687$$

경사하강법 10000번 적용

$$E = \frac{1}{2}(target - out_y)^2 = \frac{1}{2}(0.00 - 0.005770)^2 = 0.000016$$

오차가 크게 줄었다.



# 넘파이를 이용하여 MLP 구현

- 넘파이의 기능을 이용하면 모든 것을 행렬과 벡터로 표시할 수 있다.
- 행렬을 이용하면 동시에 여러 개의 예제를 동시에 학습시킬 수 있다.
- 역전파할 때는 가중치 행렬을 전치시켜서 사용한다.
- 바이어스는 입력을 1.0으로 고정하고, 이 입력에 붙은 가중치로 생각한다.



# 넘파이를 이용한 MLP 구현 (MLP2)

```
import numpy as np

# 시그모이드 함수
def actf(x):
    return 1/(1+np.exp(-x))

# 시그모이드 함수의 미분치
def actf_deriv(x):
    return x*(1-x)

# 입력유닛의 개수, 은닉유닛의 개수, 출력유닛의 개수
inputs, hiddens, outputs = 2, 2, 1
learning_rate=0.2

# 훈련 샘플과 정답
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
T = np.array([[1], [0], [0], [1]])
```



# 순방향 전파 구현

```
W1 = np.array([[0.10,0.20], [0.30,0.40]])
```

```
W2 = np.array([[0.50],[0.60]])
```

```
B1 = np.array([0.1, 0.2])
```

```
B2 = np.array([0.3])
```

```
# 순방향 전파 계산
```

```
def predict(x):
```

```
    layer0 = x
```

```
    Z1 = np.dot(layer0, W1)+B1
```

```
    layer1 = actf(Z1)
```

```
    Z2 = np.dot(layer1, W2)+B2
```

```
    layer2 = actf(Z2)
```

```
    return layer0, layer1, layer2
```

```
# 입력을 layer0에 대입한다.
```

```
# 행렬의 곱을 계산한다.
```

```
# 활성화 함수를 적용한다.
```

```
# 행렬의 곱을 계산한다.
```

```
# 활성화 함수를 적용한다.
```



# 오차 역전파 구현

# 역방향 전파 계산

def fit():

    global W1, W2, B1, B2

    for i in range(90000):

        for x, y in zip(X, T):

            x = np.reshape(x, (1, -1))

            y = np.reshape(y, (1, -1))

        # 우리는 외부에 정의된 변수를 변경해야 한다.

        # 9만번 반복한다.

        # 학습 샘플을 하나씩 꺼낸다.

        # 2차원 행렬로 만든다. ①

        # 2차원 행렬로 만든다.

        layer0, layer1, layer2 = predict(x)

        # 순방향 계산

        layer2\_error = layer2 - y

        # 오차 계산

        layer2\_delta = layer2\_error \* actf\_deriv(layer2)

        # 출력층의 델타 계산

        layer1\_error = np.dot(layer2\_delta, W2.T)

        # 은닉층의 오차 계산 ②

        layer1\_delta = layer1\_error \* actf\_deriv(layer1)

        # 은닉층의 델타 계산 ③

        W2 += -learning\_rate \* np.dot(layer1.T, layer2\_delta)

        # ④

        W1 += -learning\_rate \* np.dot(layer0.T, layer1\_delta)

        #

        B2 += -learning\_rate \* np.sum(layer2\_delta, axis=0) # ⑤

        B1 += -learning\_rate \* np.sum(layer1\_delta, axis=0) #



# 오차 역전파 구현

```
def test():  
    for x, y in zip(X, T):  
        x = np.reshape(x, (1, -1))          # 하나의 샘플을 꺼내서 2차원 행렬로 만든다.  
        layer0, layer1, layer2 = predict(x)  
        print(x, y, layer2)                # 출력층의 값을 출력해본다.  
  
fit()  
test()
```

```
[[0 0]] [1] [[0.99196032]]  
[[0 1]] [0] [[0.00835708]]  
[[1 0]] [0] [[0.00836107]]  
[[1 1]] [1] [[0.98974873]]
```



# 구글의 플레이그라운드

- 사이트(<https://playground.tensorflow.org>)
- 텐서 플로우 플레이그라운드는 자바 스크립트로 작성된 웹 애플리케이션으로 웹 브라우저에서 실행
- 이 사이트에서는 사용자가 딥러닝 모델을 구성하고 여러 가지 매개 변수를 조정하면서 실험할 수 있는 기능을 제공한다.
- [구글링에서 텐서플로우 플레이그라운드 검색해보기](#)



# 구글의 플레이그라운드

The image shows a screenshot of the TensorFlow Playground Neural Network Playground interface. The browser address bar shows the URL: `playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&r...`

Annotations in Korean are placed over the interface:

- 학습 시작 버튼** (Start Learning Button): Points to the play button icon.
- 입력 데이터 세트** (Input Dataset): Points to the 'DATA' section on the left.
- 입력층** (Input Layer): Points to the input layer options, including  $X_1$ ,  $X_2$ ,  $X_1^2$ ,  $X_2^2$ ,  $X_1X_2$ ,  $\sin(X_1)$ , and  $\sin(X_2)$ .
- 은닉층** (Hidden Layer): Points to the first hidden layer with 4 neurons.
- 출력층** (Output Layer): Points to the second hidden layer with 2 neurons.

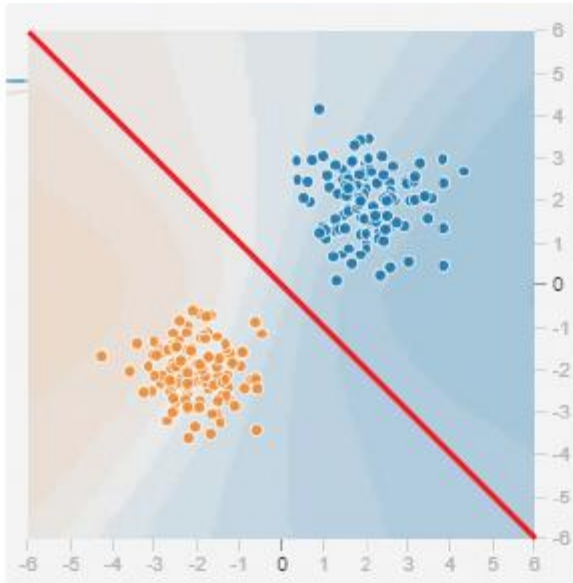
The interface displays the following information:

- Epoch:** 000,000
- Learning rate:** 0.03
- Activation:** Sigmoid
- Regularization:** None
- Regularization rate:** 0
- Problem type:** Classification

The network configuration is **2 HIDDEN LAYERS**. The first hidden layer has 4 neurons, and the second has 2 neurons. The output is a scatter plot showing two clusters of data points (blue and orange) separated by a decision boundary. The test loss is 0.493 and the training loss is 0.497. A color scale for weight values ranges from -1 (blue) to 1 (orange).



# 선형 분리 가능한 입력 데이터



$$w_1x_1 + w_2x_2 + b = 0$$



A Neural Network Playground

playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&r...

Don't worry, you can't break it. We promise.

Epoch **000,193** Learning rate 0.03 Activation Sigmoid Regularization None Regularization rate 0 Problem type Classification

DATA  
Which dataset do you want to use?  
Ratio of training to test data: 50%  
Noise: 0  
Batch size: 10  
REGENERATE

FEATURES  
Which properties do you want to feed in?  
 $X_1$   
 $X_2$   
 $X_1^2$   
 $X_2^2$   
 $X_1 X_2$   
 $\sin(X_1)$   
 $\sin(X_2)$

2 HIDDEN LAYERS  
4 neurons  
2 neurons

Click anywhere to edit. Weight is -1.7. the lines.

This is the output from one neuron. Hover to see it larger.

OUTPUT  
Test loss 0.002  
Training loss 0.002

Colors shows data, neuron and weight values.

A Neural Network Playground

playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&r...

Don't worry, you can't break it. We'll fix it.

Epoch: 000,000

Learning rate: 0.03

Activation: Sigmoid

Regularization: None

Regularization rate: 0

Problem type: Classification

DATA: Which dataset do you want to use?

FEATURES: Which properties do you want to feed in?

2 HIDDEN LAYERS

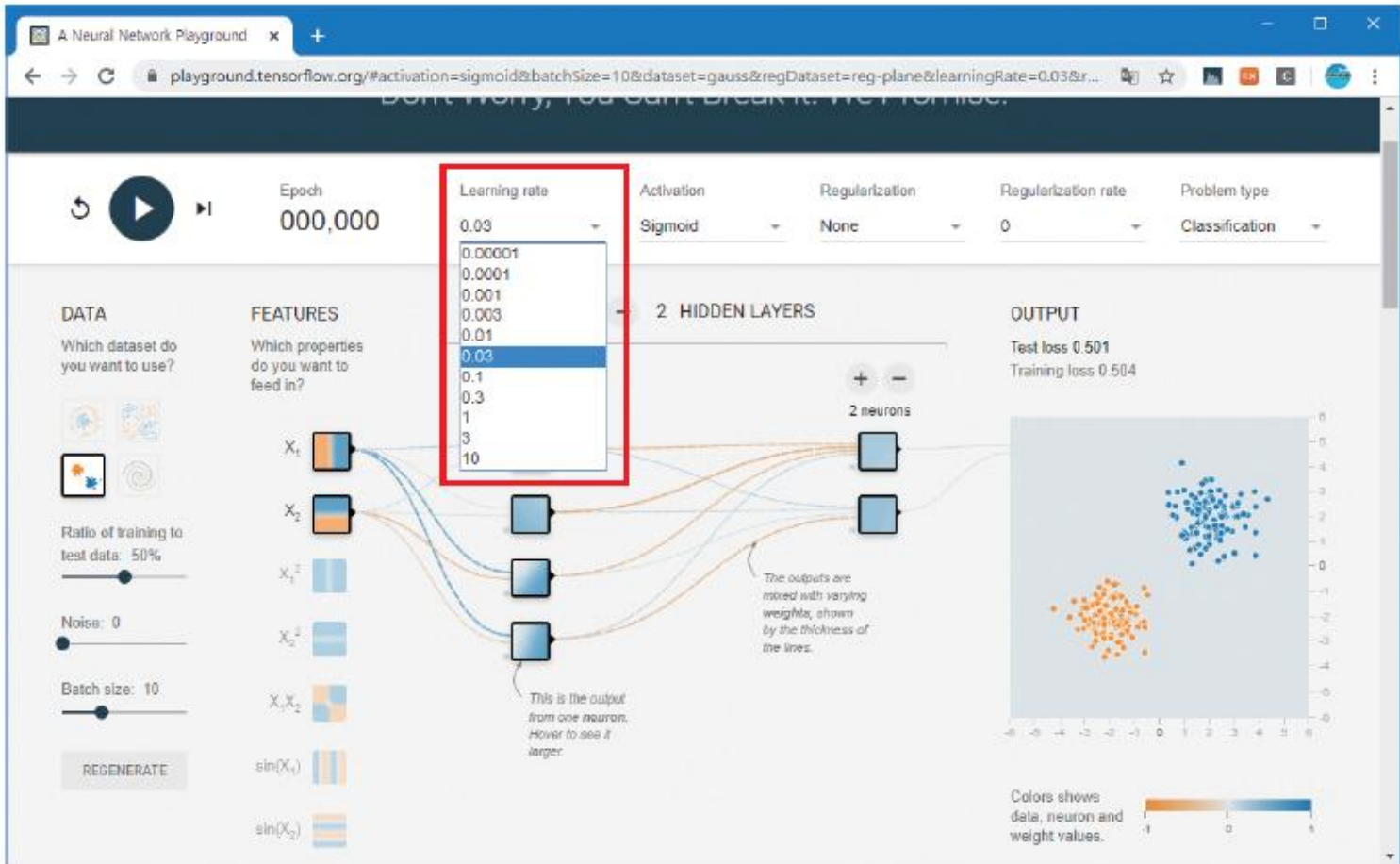
OUTPUT: Test loss 0.501, Training loss 0.504

2 neurons

The outputs are mixed with varying weights, shown by the thickness of the lines.

This is the output from one neuron. Hover to see it larger.

Colors shows data, neuron and weight values.



# 활성화 함수 선택

The screenshot shows the TensorFlow Playground interface with the following settings and components:

- Epoch:** 000,000
- Learning rate:** 0.03
- Activation:** Sigmoid (dropdown menu is open, showing options: ReLU, Tanh, Sigmoid, Linear)
- Regularization:** None
- Regularization rate:** 0
- Problem type:** Classification

**DATA:** Which dataset do you want to use? (Gauss, Reg-plane, etc.)

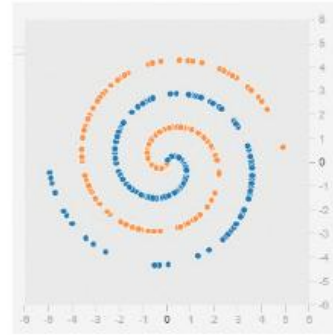
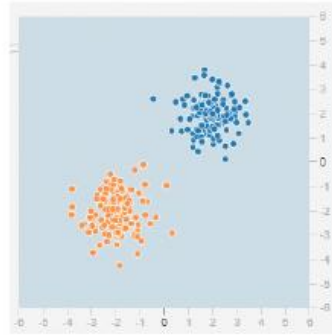
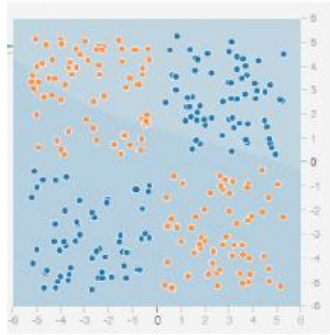
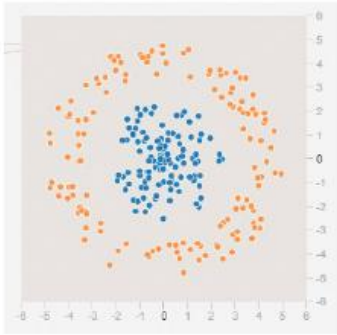
**FEATURES:** Which properties do you want to feed in? (X<sub>1</sub>, X<sub>2</sub>, X<sub>1</sub><sup>2</sup>, X<sub>2</sub><sup>2</sup>, X<sub>1</sub>X<sub>2</sub>, sin(X<sub>1</sub>), sin(X<sub>2</sub>))

**NEURONS:** 4 neurons in the hidden layer, 2 neurons in the output layer.

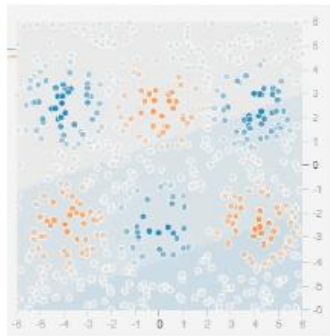
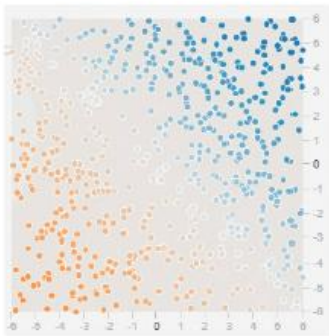
**OUTPUT:** Test loss 0.501, Training loss 0.504. A scatter plot shows two clusters of data points (blue and orange) separated by a decision boundary.

**Annotations:** A red box highlights the 'Activation' and 'Problem type' dropdowns. A red arrow points from the 'Problem type' dropdown to the scatter plot. A text box notes: "The outputs are mixed with varying weights, shown by the thickness of the lines." Another text box notes: "This is the output from one neuron. Hover to see it larger."

- 분류 문제



- 회귀 문제





# 데이터와 테스트 데이터의 비율

A Neural Network Playground

playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=gauss&regDataset=reg-gauss&learningRate=0.03&r...

Epoch: 000,000 | Learning rate: 0.03 | Activation: Sigmoid | Regularization: None | Regularization rate: 0 | Problem type: Classification

DATA: Which dataset do you want to use?

FEATURES: Which properties do you want to feed in?

2 HIDDEN LAYERS

4 neurons | 2 neurons

OUTPUT: Test loss 0.00 | Training loss 0.50

점의 개수가 적어진다.

Ratio of training to test data: 20%

Noise: 0 | Batch size: 10

REGENERATE

Colors shows data, neuron and weight values.

A Neural Network Playground

playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&n...

Epoch: 000,000 | Learning rate: 0.03 | Activation: Tanh | Regularization: None | Regularization rate: 0 | Problem type: Classification

**DATA**

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 20

Batch size: 10

GENERATE

**FEATURES**

Which properties do you want to feed in?

- $X_1$
- $X_2$
- $X_1^2$
- $X_2^2$
- $X_1X_2$
- $\sin(X_1)$
- $\sin(X_2)$

2 HIDDEN LAYERS

4 neurons | 2 neurons

The outputs are mixed with varying weights, shown by the thickness of the lines.

This is the output from one neuron. Hover to see it larger.

**OUTPUT**

Test loss 0.504  
Training loss 0.514

Colors shows data, neuron and weight values.

Show test data  Discretize output

노이즈가 많아질  
수록  
다.

# 이력 특징 선택

The screenshot shows the TensorFlow Playground interface. At the top, the browser address bar displays the URL: `playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=circle&regDataset=reg-gauss&learningRate=0.03&tr...`. The main control panel includes a play button, a progress indicator for Epochs (000,000), and dropdown menus for Learning rate (0.03), Activation (Sigmoid), Regularization (None), Regularization rate (0), and Problem type (Classification).

The interface is divided into several sections:

- DATA:** A section for selecting datasets and adjusting parameters like the ratio of training to test data (50%), noise (0), and batch size (10). A "REGENERATE" button is located at the bottom of this section.
- FEATURES:** A section where users can select input features. The features listed are  $X_1$ ,  $X_2$ ,  $X_1^2$ ,  $X_2^2$ ,  $X_1X_2$ ,  $\sin(X_1)$ , and  $\sin(X_2)$ . This section is highlighted with a red border in the image.
- 2 HIDDEN LAYERS:** A diagram showing a neural network with 4 neurons in the first hidden layer and 2 neurons in the second hidden layer. Lines of varying thickness represent weights between neurons. A tooltip states: "The outputs are mixed with varying weights, shown by the thickness of the lines." Another tooltip points to a line: "This is the output from one neuron. Hover to see it larger."
- OUTPUT:** A scatter plot showing the training data (orange and blue dots) and the network's output. The test loss is 0.498 and the training loss is 0.502. A color scale at the bottom indicates that colors represent data, neuron, and weight values.

# 인공지능 추가하기

A Neural Network Playground

playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=circle&regDataset=reg-gauss&learningRate=0.03&tr...

Epoch Learning rate Activation Regularization Regularization rate Problem type  
Classification

은닉층의 개수를 증가시킨다. 은닉층의 개수를 감소시킨다.

+ - 6 HIDDEN LAYERS

DATA: Which dataset do you want to use?  
FEATURES: Which properties do you want to feed in?  
OUTPUT: Test loss 0.545, Training loss 0.541

6 neurons 6 neurons 6 neurons 6 neurons 6 neurons 2 neurons

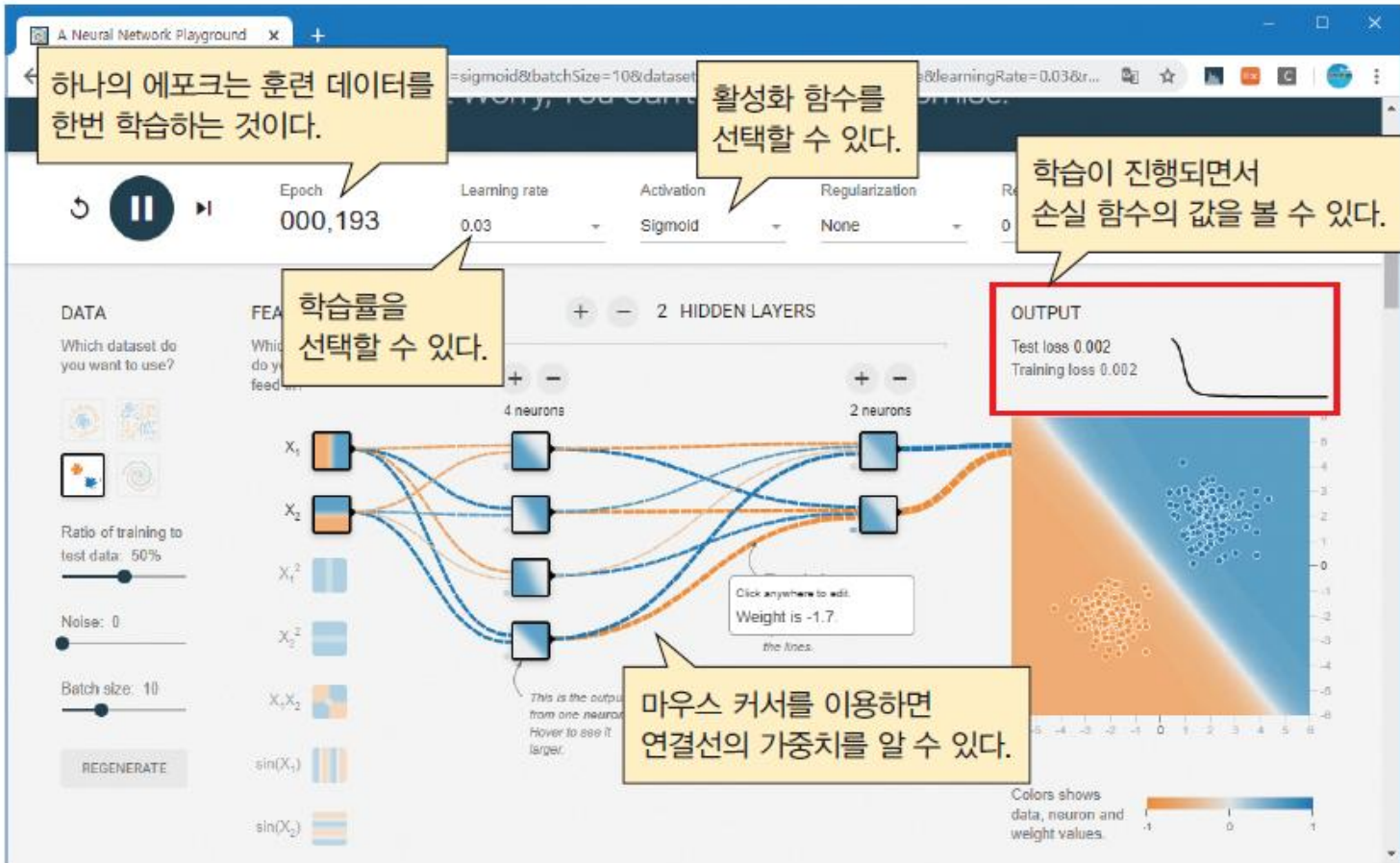
각 레이어의 노드를 증가시키거나 감소시킨다.

test data: 50%  
Noise: 0  
Batch size: 10  
REGENERATE

$X_1^2$   
 $X_2^2$   
 $X_1, X_2$   
 $\sin(X_1)$   
 $\sin(X_2)$

This is the output from one neuron. Hover to see it.  
The outputs are mixed with varying weights, shown by the thickness of

Colors shows data, neuron and weight values.



하나의 에포크는 훈련 데이터를 한번 학습하는 것이다.

활성화 함수를 선택할 수 있다.

학습이 진행되면서 손실 함수의 값을 볼 수 있다.

학습률을 선택할 수 있다.

OUTPUT  
Test loss 0.002  
Training loss 0.002

Click anywhere to edit.  
Weight is -1.7  
the lines.

마우스 커서를 이용하면 연결선의 가중치를 알 수 있다.

Colors shows data, neuron and weight values.

Epoch: 000,193  
Learning rate: 0.03  
Activation: Sigmoid  
Regularization: None

2 HIDDEN LAYERS  
4 neurons  
2 neurons

DATA  
Which dataset do you want to use?  
Ratio of training to test data: 50%  
Noise: 0  
Batch size: 10  
REGENERATE

FEA  
Which do you feed in?  
 $X_1$   
 $X_2$   
 $X_1^2$   
 $X_2^2$   
 $X_1 X_2$   
 $\sin(X_1)$   
 $\sin(X_2)$

Neural Network Playground

playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&r...

Epoch: 010,734 | Learning rate: 0.03 | Activation: Sigmoid | Regularization: None | Regularization rate: 0 | Problem type: Classification

DATA: Which dataset do you want to use?

FEATURES: Which properties do you want to feed in?

OUTPUT: Test loss 0.000 | Training loss 0.000

2 HIDDEN LAYERS

4 neurons | 2 neurons

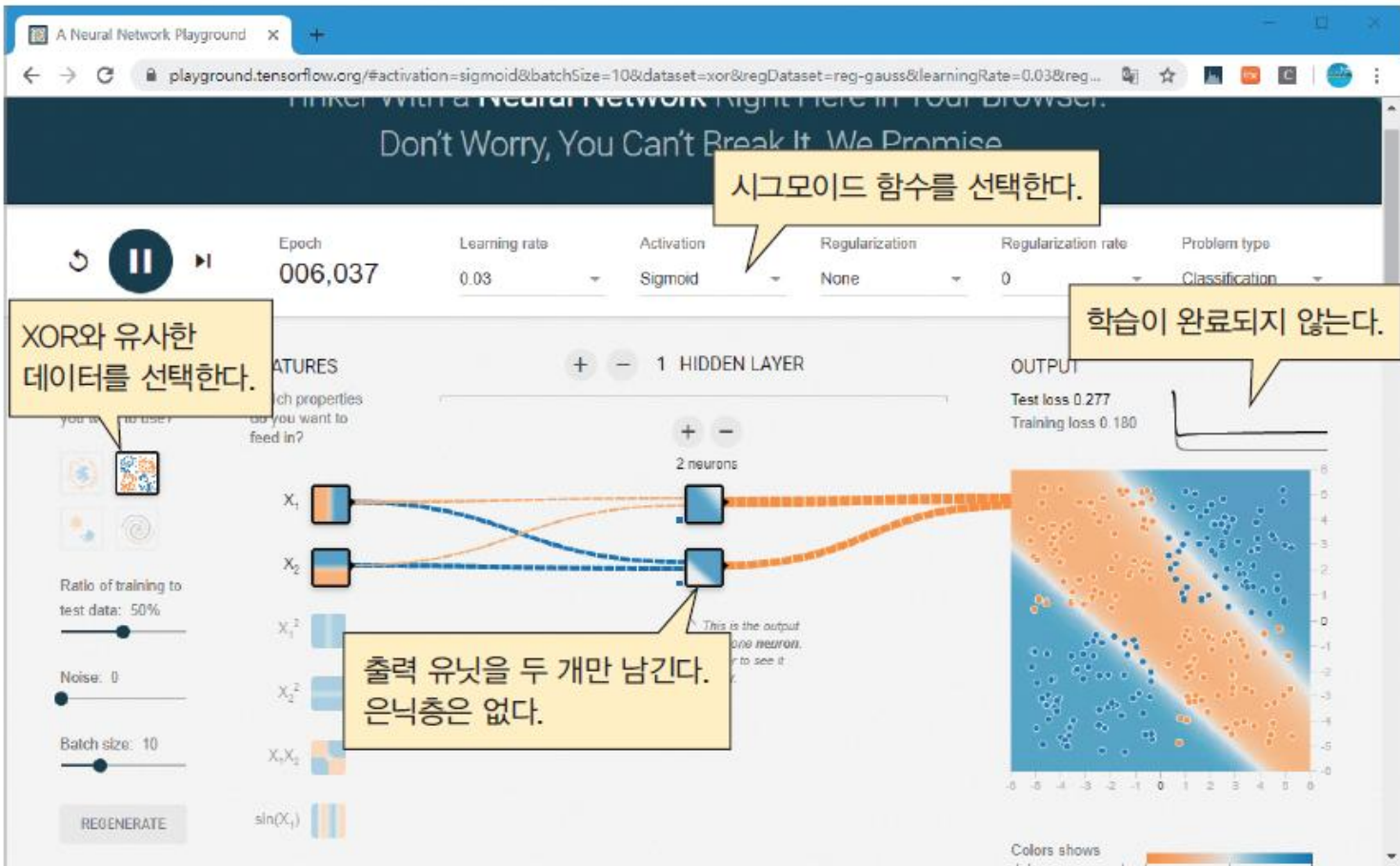
*This is the output from one neuron. Hover to see it larger.*

*The outputs are mixed with varying weights, shown by the thickness of the lines.*

이것이 입력을 2개의 카테고리로 나누는 판단 경계선이다.



# 인공지능이 바로 실생활



A Neural Network Playground

playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=xor&regDataset=reg-gauss&learningRate=0.03&reg...

Don't Worry, You Can't Break It. We Promise.

Epoch: 006,037 | Learning rate: 0.03 | Activation: Sigmoid | Regularization: None | Regularization rate: 0 | Problem type: Classification

1 HIDDEN LAYER (2 neurons)

OUTPUT: Test loss 0.277, Training loss 0.180

Annotations:

- XOR와 유사한 데이터를 선택한다. (Select data similar to XOR.)
- 시그모이드 함수를 선택한다. (Select the sigmoid function.)
- 출력 유닛을 두 개만 남긴다. 은닉층은 없다. (Leave only two output units. There is no hidden layer.)
- 학습이 완료되지 않는다. (Learning is not complete.)



# 인공지능 추가한 시스템

A Neural Network Playground

playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=xor&regDataset=reg-gauss&learningRate=0.03&reg...

Don't Worry, You Can't Break It. We Promise.

Epoch: 000,750 | Learning rate: 0.03 | Activation: Sigmoid | Regularization: None | Regularization rate: 0 | Problem type: Classification

DATA: Which dataset do you want to use? (xor selected)

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES: Which properties do you want to feed in?

- $X_1$
- $X_2$
- $X_1^2$
- $X_2^2$
- $X_1 X_2$
- $\sin(X_1)$

2 HIDDEN LAYERS

- 4 neurons
- 2 neurons

OUTPUT: Test loss 0.026, Training loss 0.019

거의 학습이 완료된다.

은닉층을 추가한다.

This is the output

The outputs are mixed with varying weights, shown by the thickness of the lines.



# Summary

- 입력층과 출력층 사이에 은닉층(hidden layer)을 가지고 있는 신경망을 다층 퍼셉트론(multilayer perceptron: MLP)이라고 부른다.
- MLP를 학습시키기 위하여 역전파 알고리즘(back-propagation)이 재발견되었다. 이 알고리즘이 지금까지도 신경망 학습 알고리즘의 근간이 되고 있다.
- 역전파 알고리즘은 입력이 주어지면 순방향으로 계산하여 출력을 계산한 후에 실제 출력과 우리가 원하는 출력 간의 오차를 계산한다. 이 오차를 역방향으로 전파하면서 오차를 줄이는 방향으로 가중치를 변경한다.



# Q & A

