6장 *MLP*(다층 퍼셉트론)

** BACK-PROPAGATION*

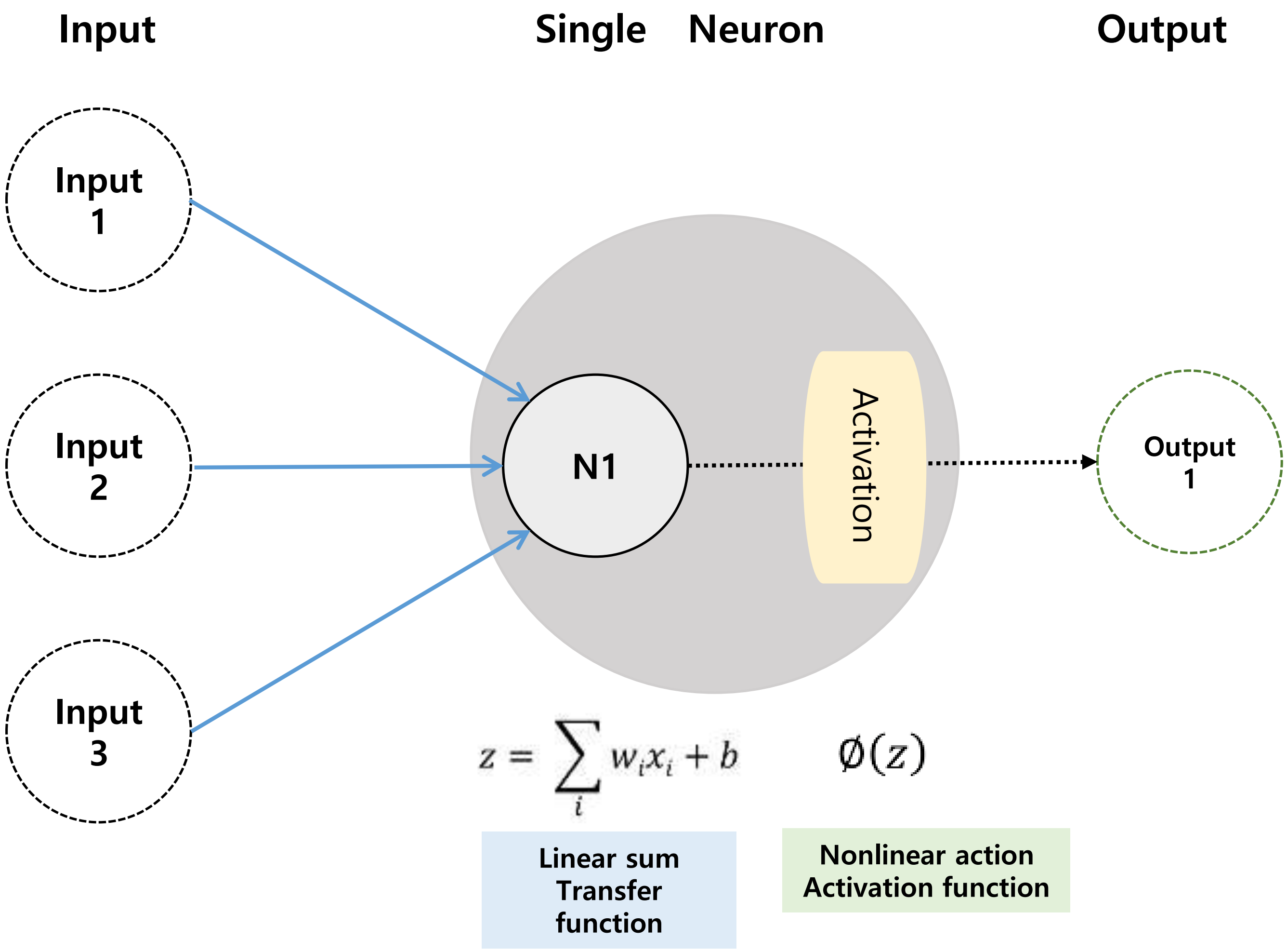
$$\frac{\partial E}{\partial w_{ij}} = \delta_j out_i \qquad \text{여기서} \quad \delta_j = \begin{cases} (out_j - t_j) f'(\mathrm{net_j}) & j\text{가 출력층 유닛이면} \\ \left(\sum_k \mathrm{w_{jk}} \delta_k\right) \mathrm{f}'(\mathrm{net_j}) & j\text{가 은닉층 유닛이면} \end{cases}$$

Multi Layer Perceptron

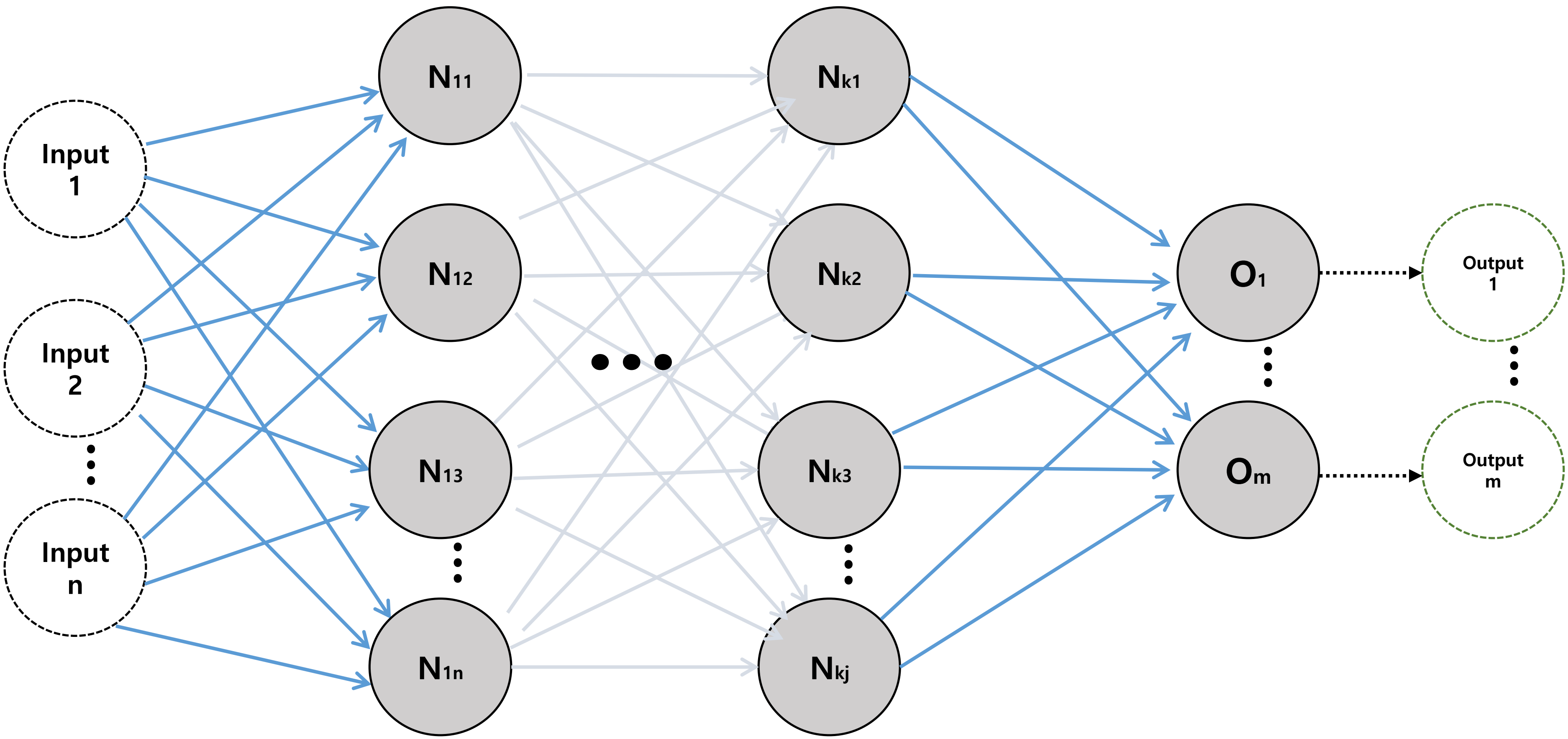
학습 목표

- MLP의 작동 원리를 이해한다.
- **경사하강법**을 이해한다.
- **역전파 알고리즘**을 이해한다.
- 넘파이만으로 MLP를 구현해본다..

AN (Artificial Neuron)



ANN (Artificial Neural Network) → MLP



n-Input

k-Hidden layers → deep layer

m-Output layer

MLP

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

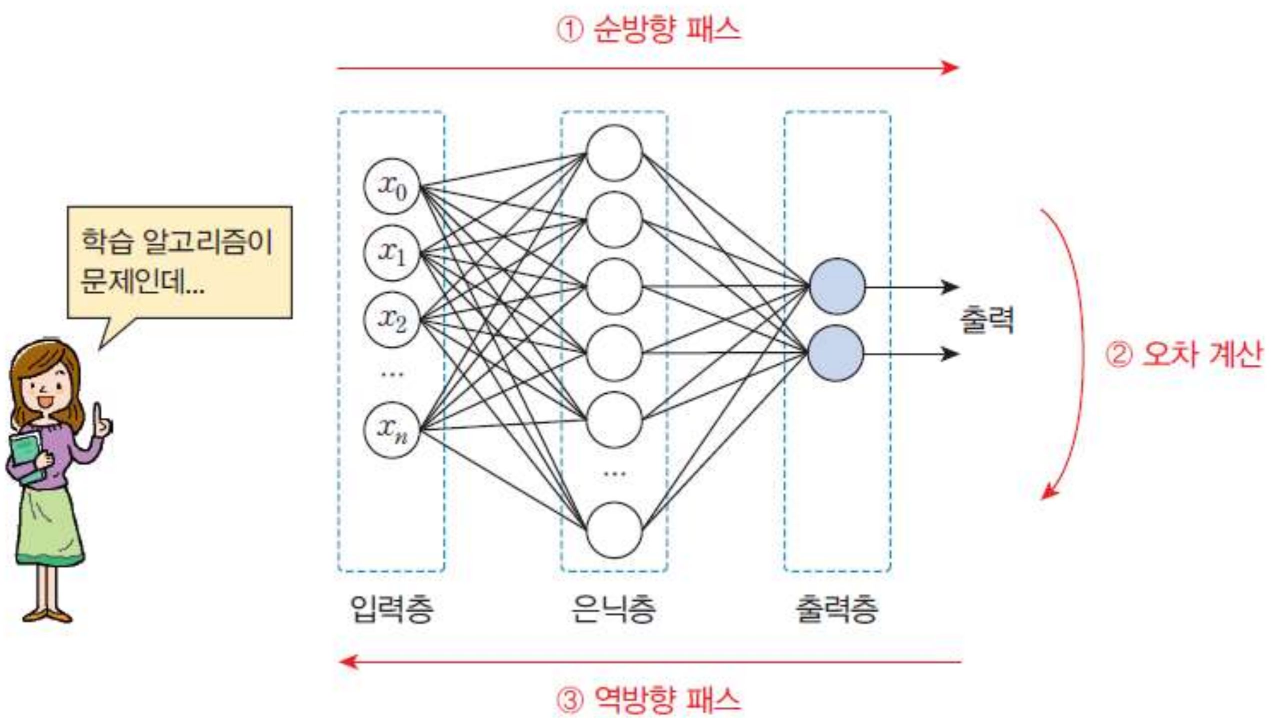


그림 6-1 MLP의 구조

활성화 함수

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

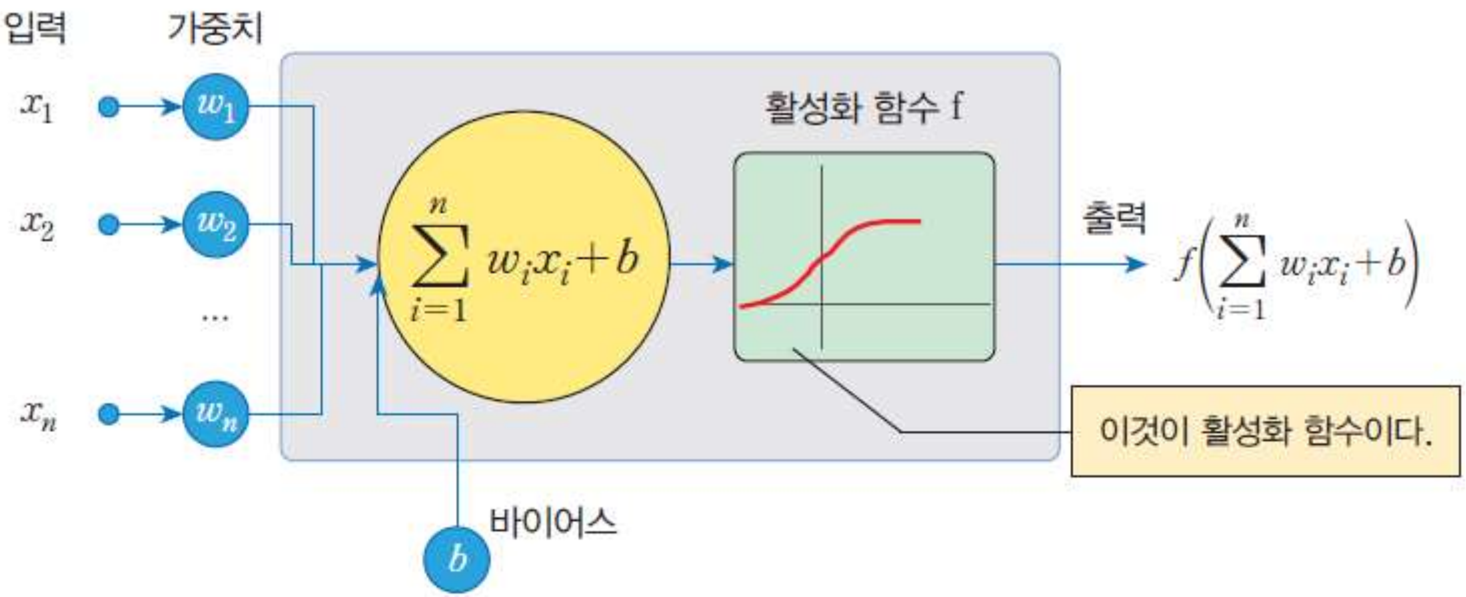


그림 6-2 활성화 함수

일반적으로 많이 사용되는 활성화 함수

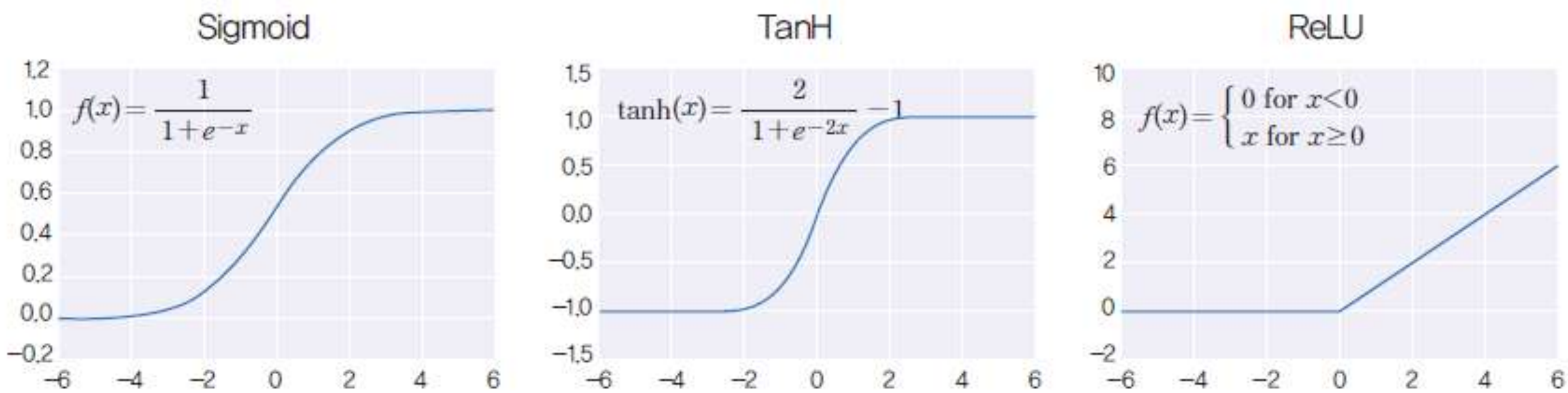


그림 6-3 많이 사용되는 활성화 함수

https://www.v7labs.com/blog/neural-networks-activation-functions

선형 레이어는 많아도 쓸모가 없다.

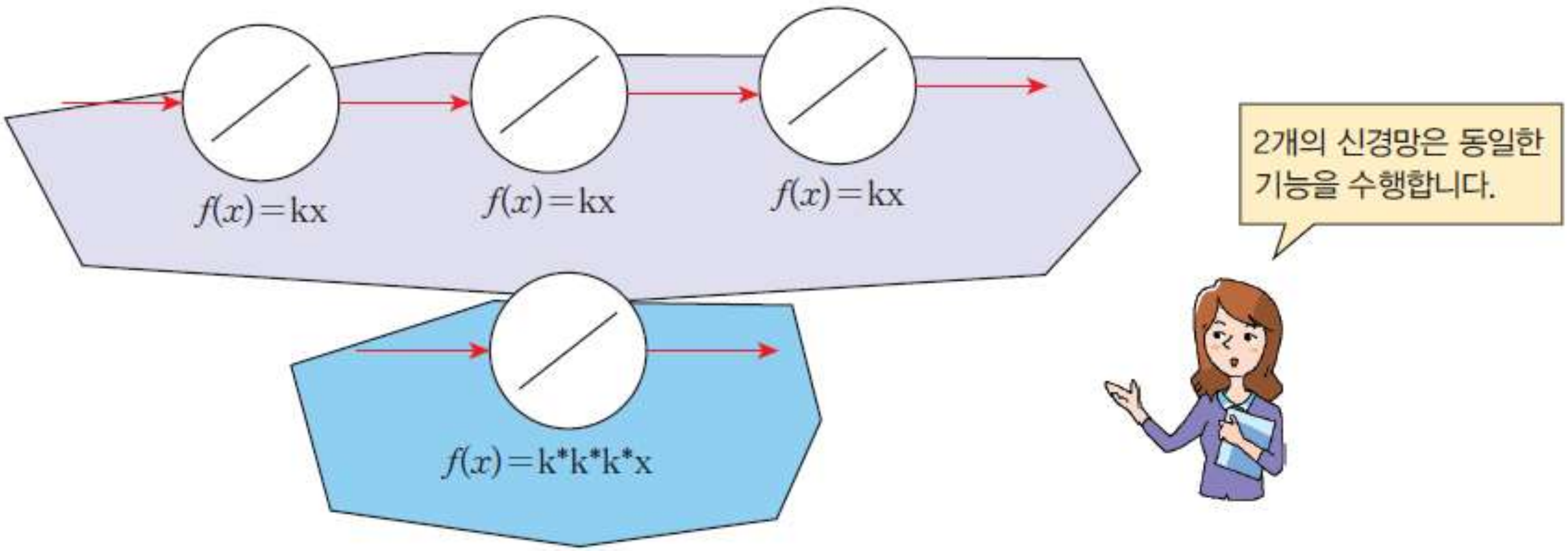
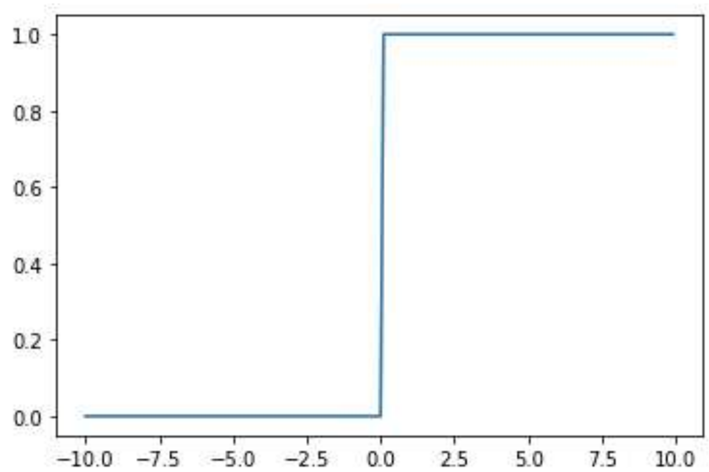


그림 6-4 선형 레이어는 아무리 많아도 하나의 레이어로 대치될 수 있다.

계단 함수 (step function)

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

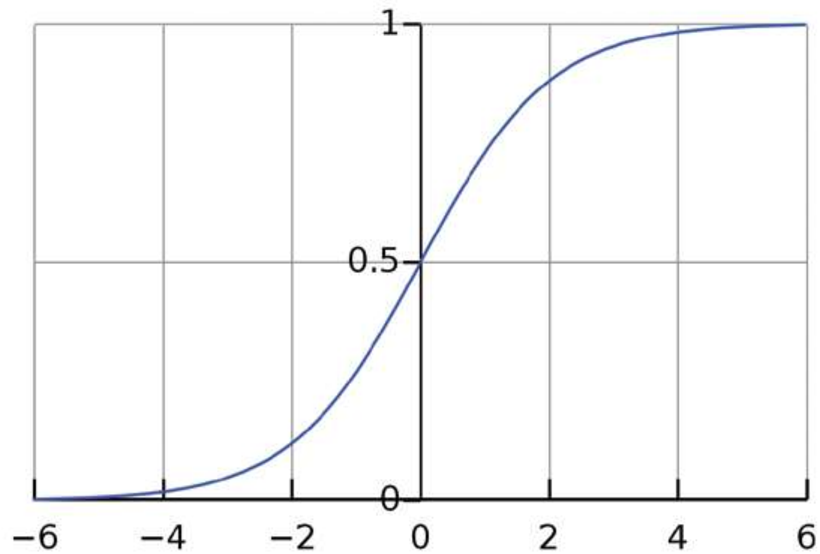
$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$



시그모이드 함수 (Sigmoid function)

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

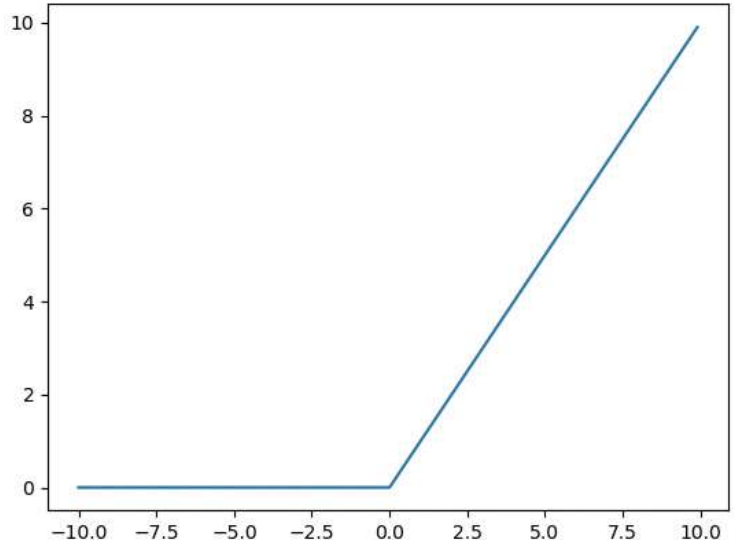
$$f(x) = \frac{1}{1+e^{-x}}$$



ReLU 함수 (Rectifed Linear Unit function)

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

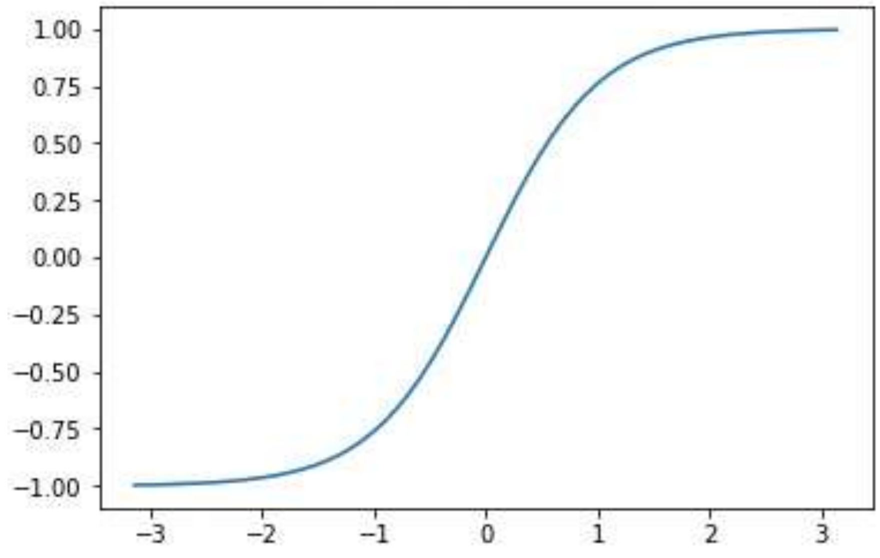
$$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$



tanh 함수

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

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



Lab: 파이썬으로 활성화 함수 구현하기

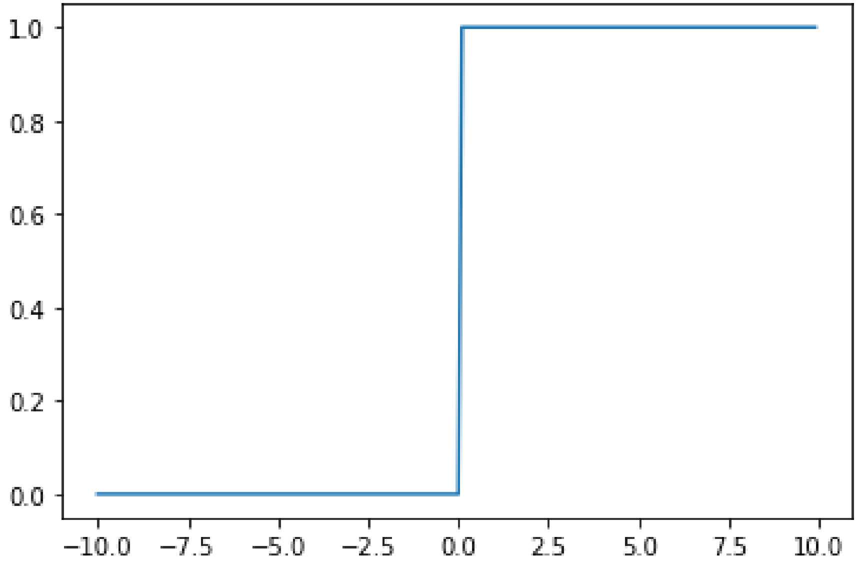
```
def step(x):
  if x > 0.000001:  return 1    # 부동 소수점 오차 방지
  else                return 0
```

넘파이 배열을 받기 위하여 변경한다.

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

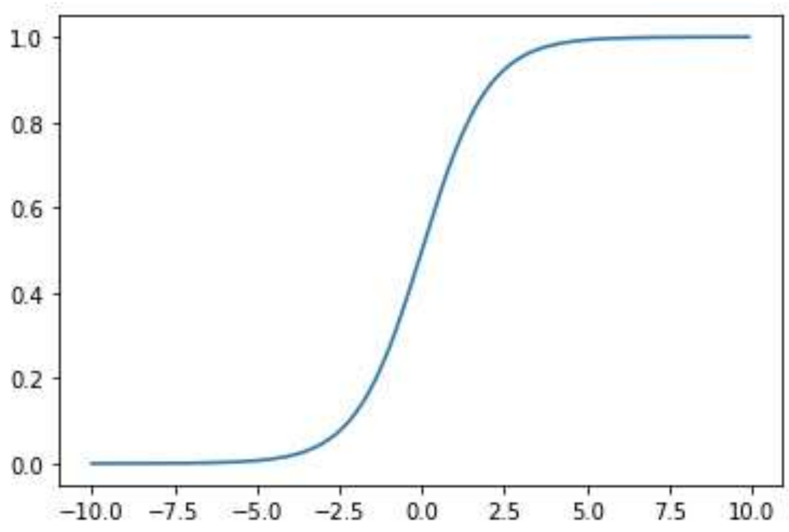
그래프 그리기

```
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()
```



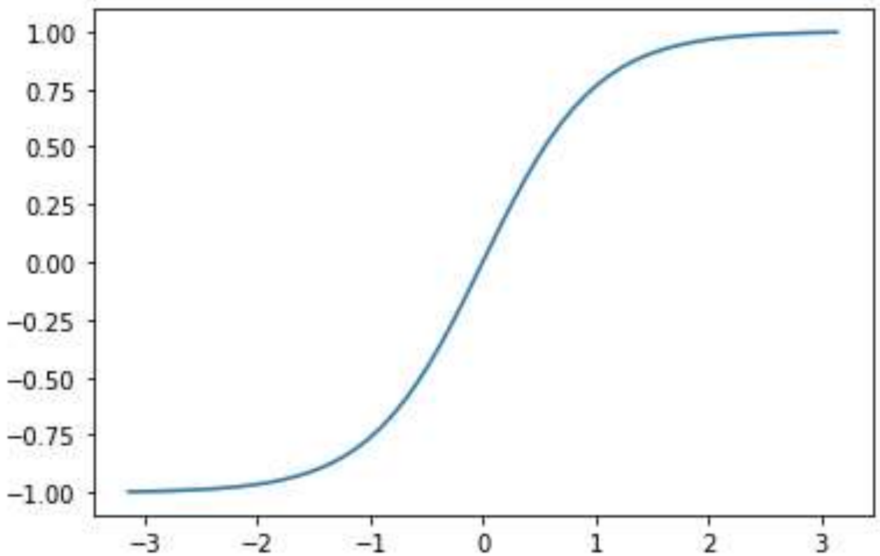
시그모이드 함수

```
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()
```



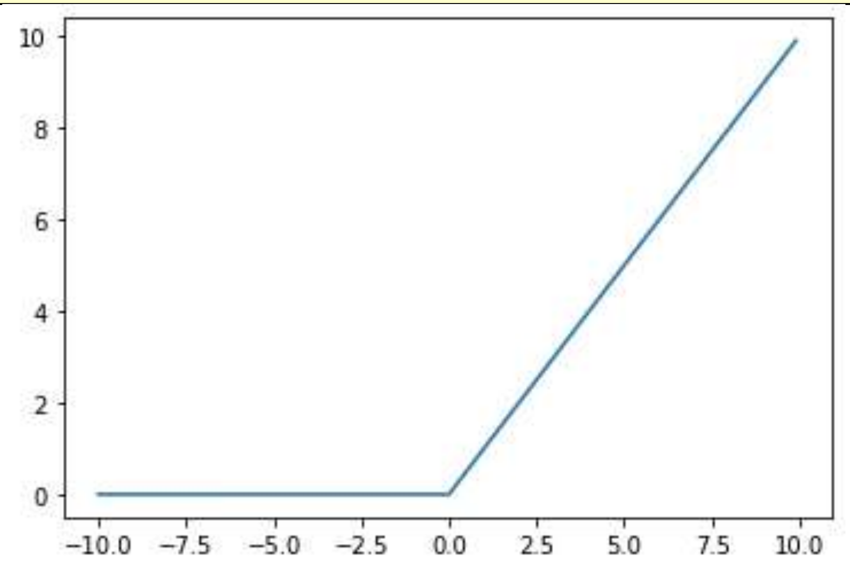
tanh 함수

```
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.

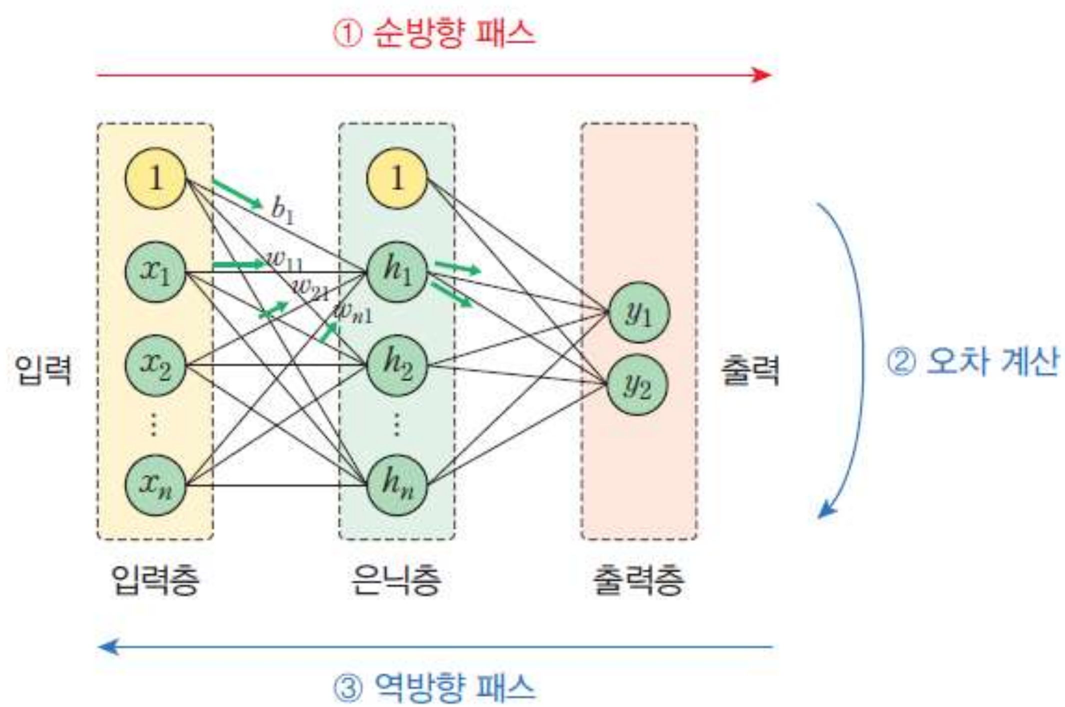
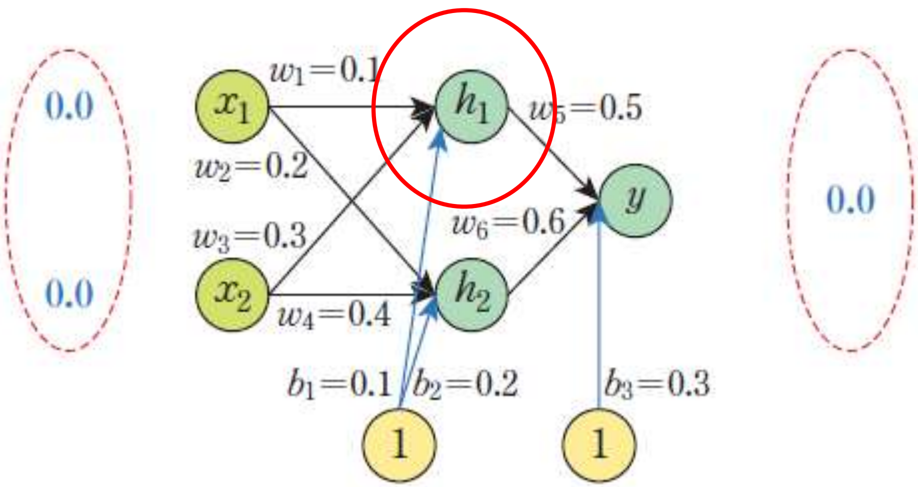


그림 6-5 순방향 패스

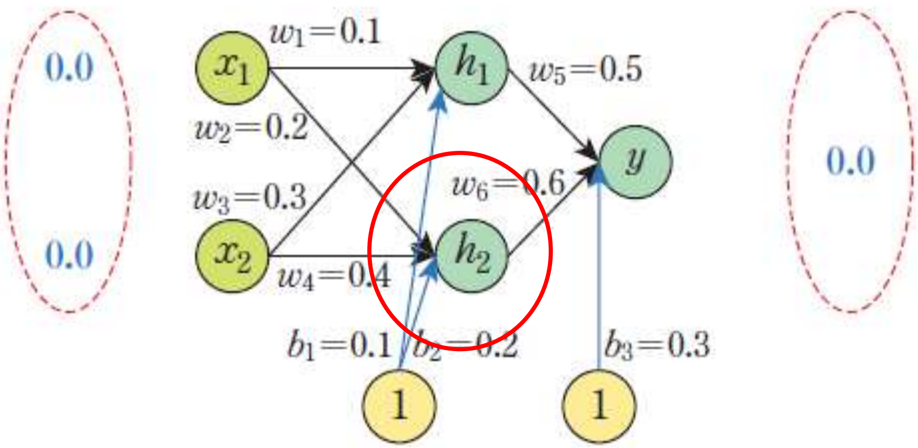
손으로 계산해보자.



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

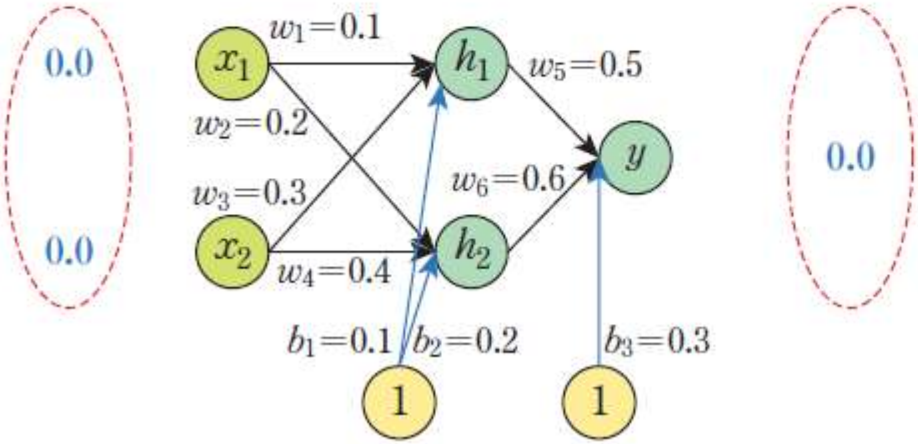
$$a_1 = \frac{1}{1+e^{-z_1}} = \frac{1}{1+e^{-0.1}} = 0.524979$$

손으로 계산해보자.



$$a_2 = 0.549834$$

손으로 계산해보자.

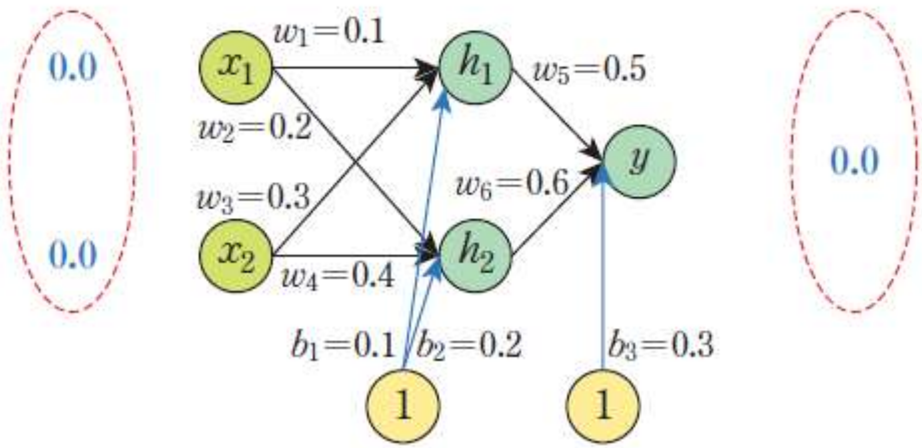


$$z_y = w_5 * a_1 + w_6 * a_2 + b_3$$
$$= 0.5 * 0.524979 + 0.6 * 0.549834 + 0.3 = 0.892389$$

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

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

행렬로 표시해보자.

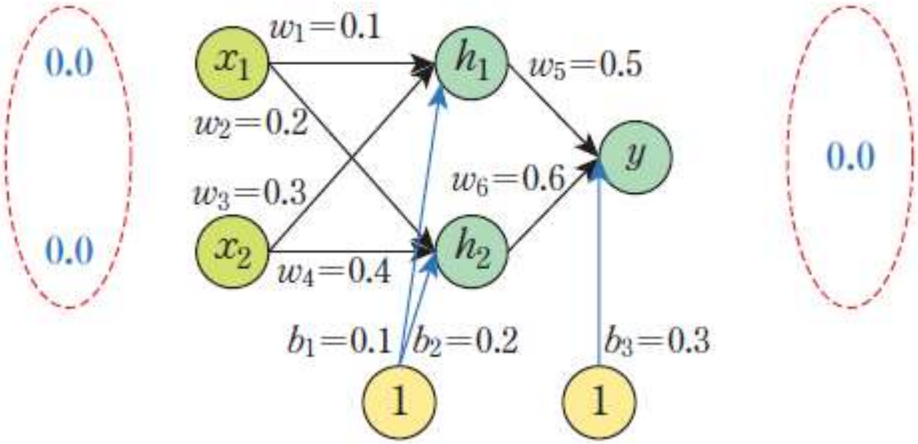


$$z_1 = w_1 * x_1 + w_3 * x_2 + b_1$$
$$z_2 = w_2 * x_1 + w_4 * x_2 + b_2$$

$$Z_1 = XW_1 + B_1$$

행렬로 표시할 수 있다.

행렬로 표시해보자. → *hidden layer*

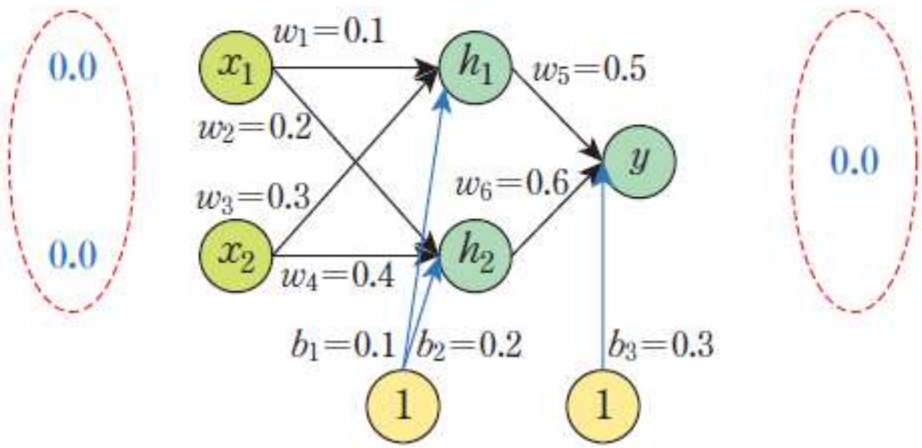


$$X=[x_1 \quad x_2],\ B_1=[b_1 \quad b_2],\ Z_1=[z_1 \quad z_2]$$

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

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

행렬로 표시해보자. → *output layer*



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

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

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

Lab: MLP 순방향 패스

mlp1.py

```
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([[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                          # 입력을 layer0에 대입한다.
        Z1 = np.dot(layer0, W1)+B1          # 행렬의 곱을 계산한다.
        layer1 = actf(Z1)                   # 활성화 함수를 적용한다.
        Z2 = np.dot(layer1, W2)+B2          # 행렬의 곱을 계산한다.
        layer2 = actf(Z2)                   # 활성화 함수를 적용한다.
        return layer0, layer1, layer2
```

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

학습이 없으므로 난수만 출력된다.

학습 → 손실 함수 계산

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

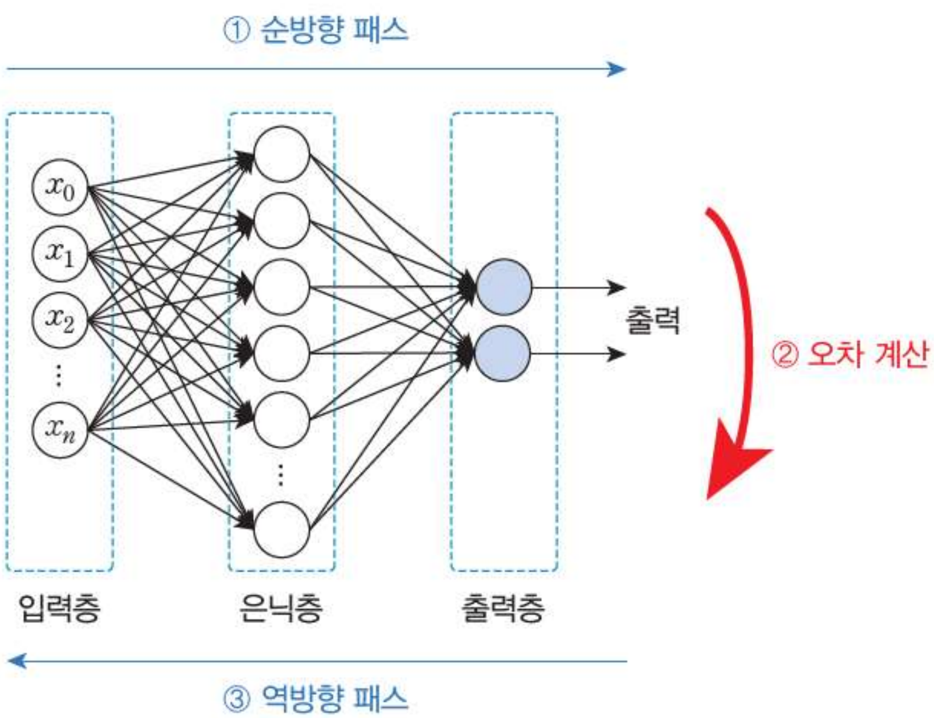
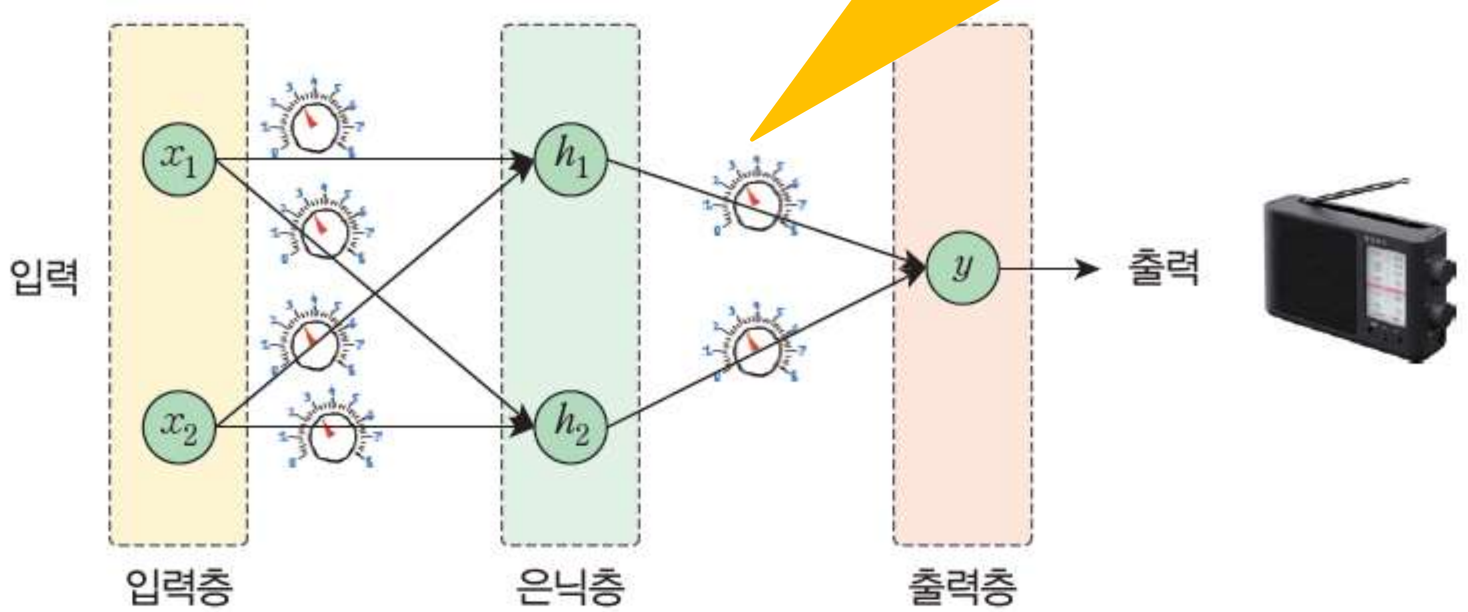


그림 6-6 오차 계산

가중치——다이얼 튜닝(미세 조정)

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



손실 함수

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

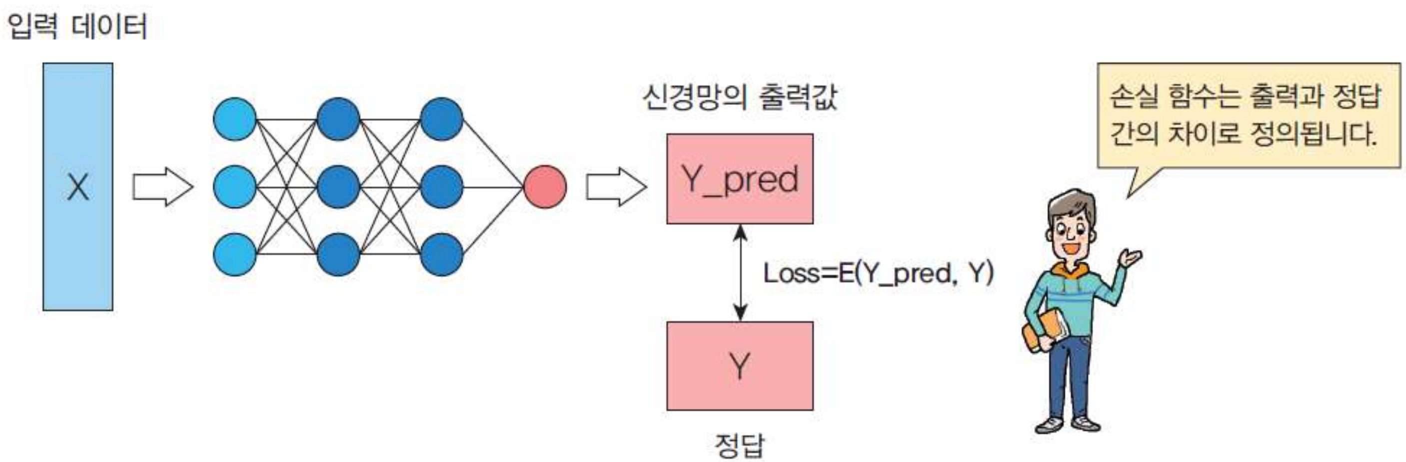


그림 6-7 손실함수의 정의

평균 제곱 오차(MSE)

- 예측값과 정답 간의 평균 제곱 오차

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

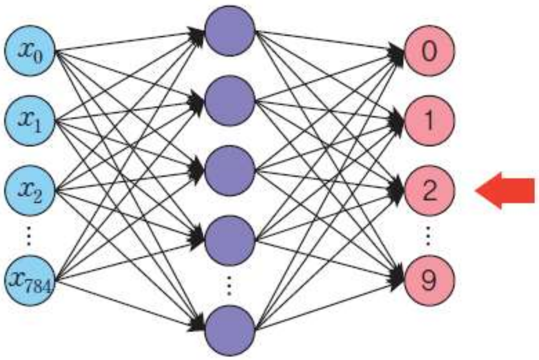


그림 6-8 MNIST 숫자 이미지를 분류하는 신경망

loss_func.py

```
>>> 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 차이나는 경우

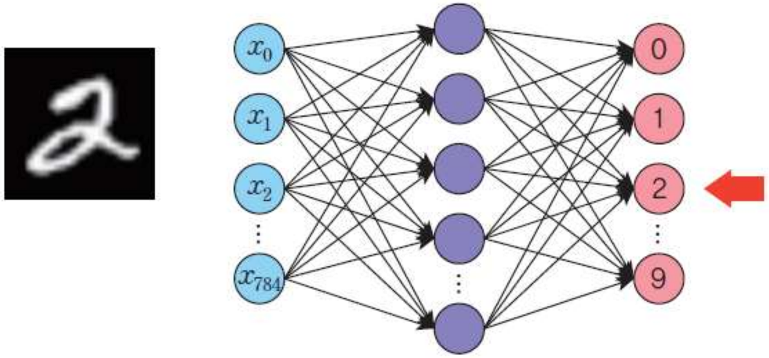


그림 6-8 MNIST 숫자 이미지를 분류하는 신경망

```
>>> 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 decent)

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

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

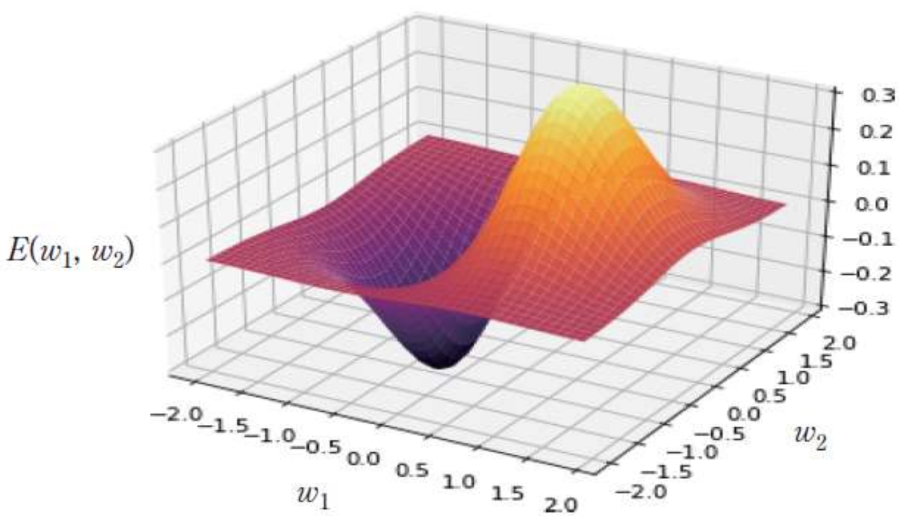


그림 6-9 경사 하강법

경사하강법

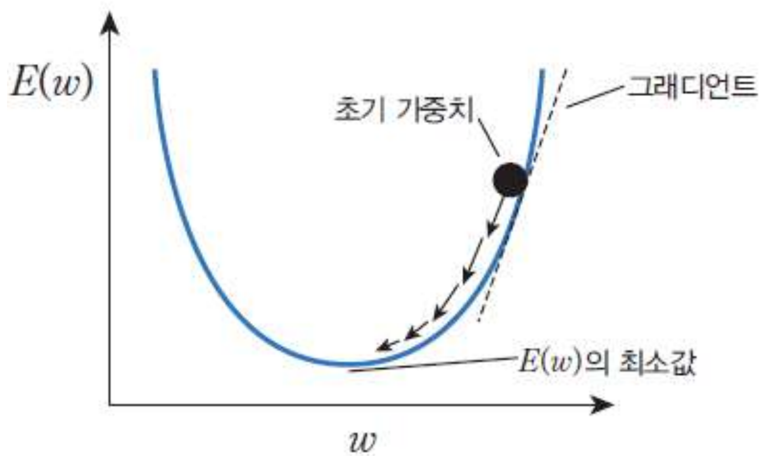


그림 6-11 경사 하강법

손실함수를 가중치로 미분한 값이 양수이면	➡	가중치를 감소시킨다.
손실함수를 가중치로 미분한 값이 음수이면	➡	가중치를 증가시킨다.

Lab: 경사하강법의 실습

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

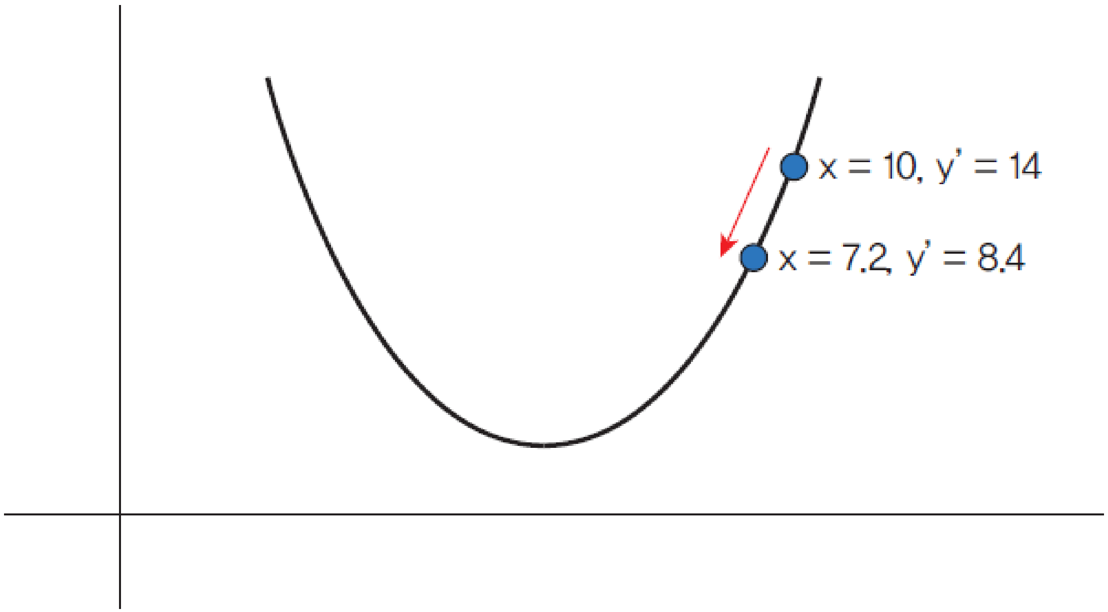


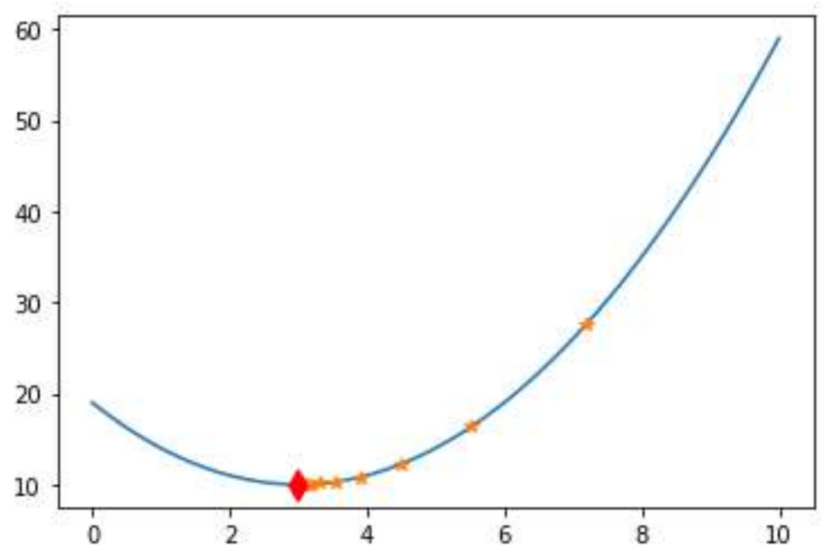
그림 6-12 그래디언트의 계산

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)
```

실행 결과

```
손실함수값( 7.1999999999999 )= 27.639999999999993
손실함수값( 5.52 )= 16.350399999999997
손실함수값( 4.512 )= 12.286143999999998
손실함수값( 3.9071999999999996 )= 10.82301184
손실함수값( 3.54432 )= 10.2962842624
...
손실함수값( 3.0000000000000004 )= 10.0
최소값 =  3.0000000000000004
```

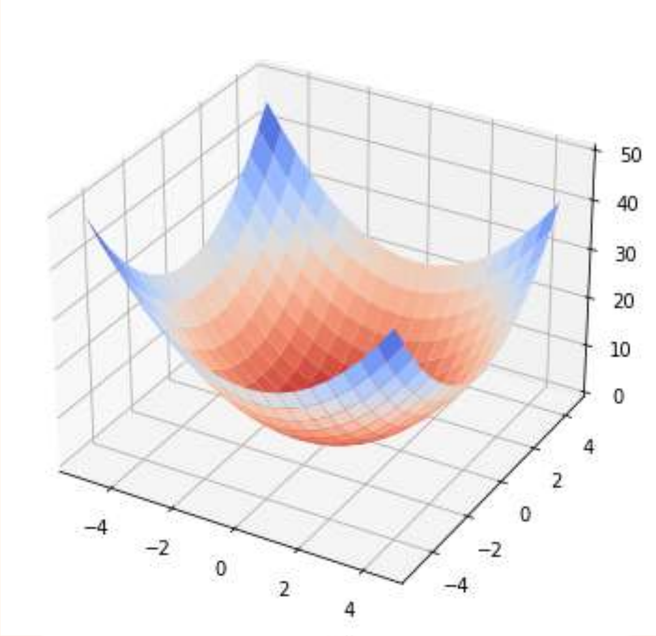


grad_desc1_A.py

Lab: 2차원 그래디언트 시각화

grad_desc2.py

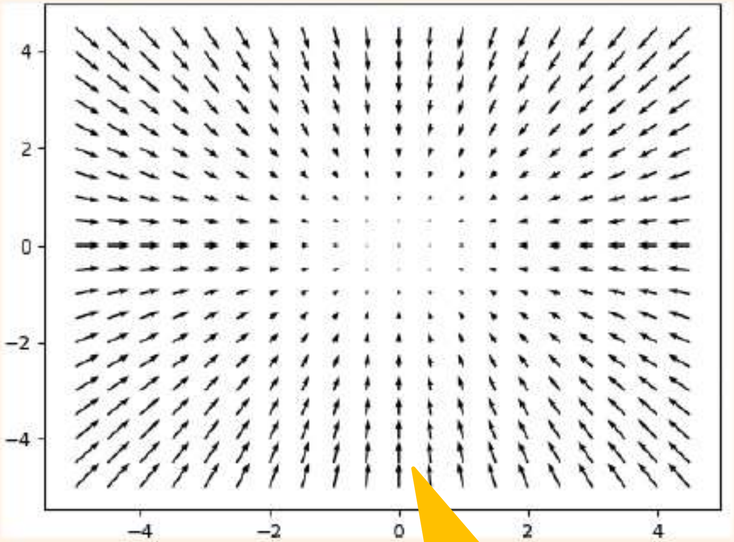
```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

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

```python
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

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

그래디언트의 음수



화살표가 최소값을 가리키고 있음을 알 수 있다.

역전파 학습 알고리즘 (back-propagation)

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

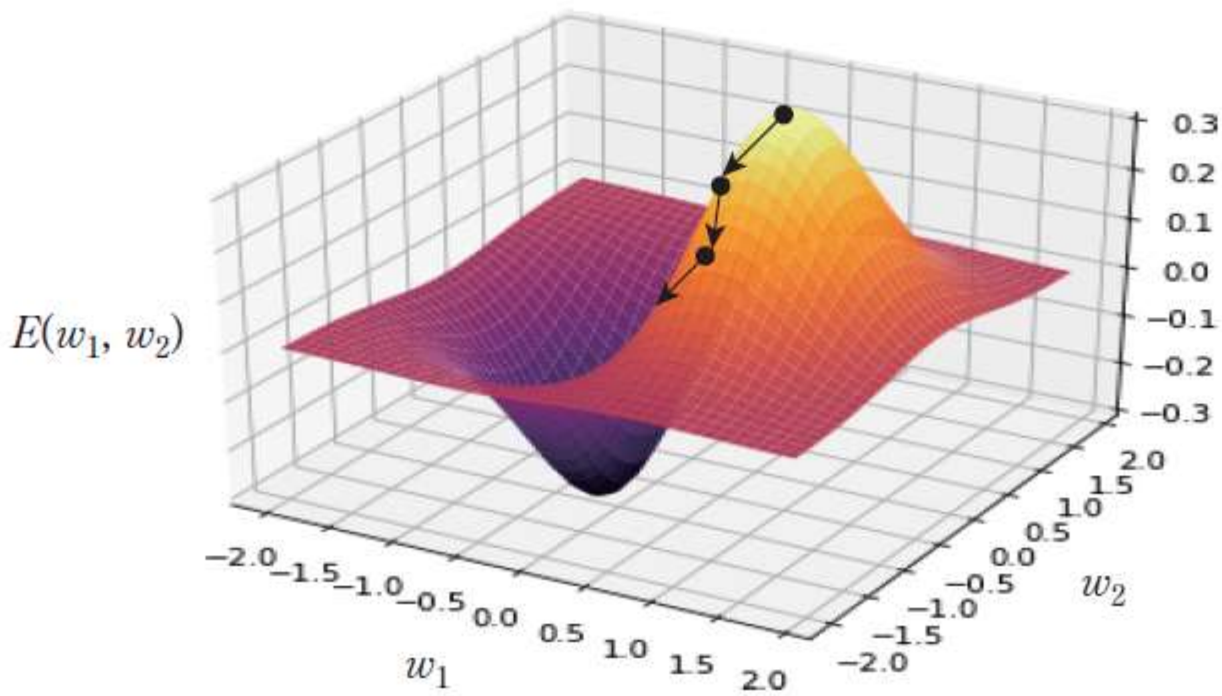
역전파 학습 알고리즘 (back-propagation)

- 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 E의 그래디언트 $\partial E / \partial w$을 계산한다.
④ $w(t+1) = w(t) - \eta \frac{\partial E}{\partial w}$

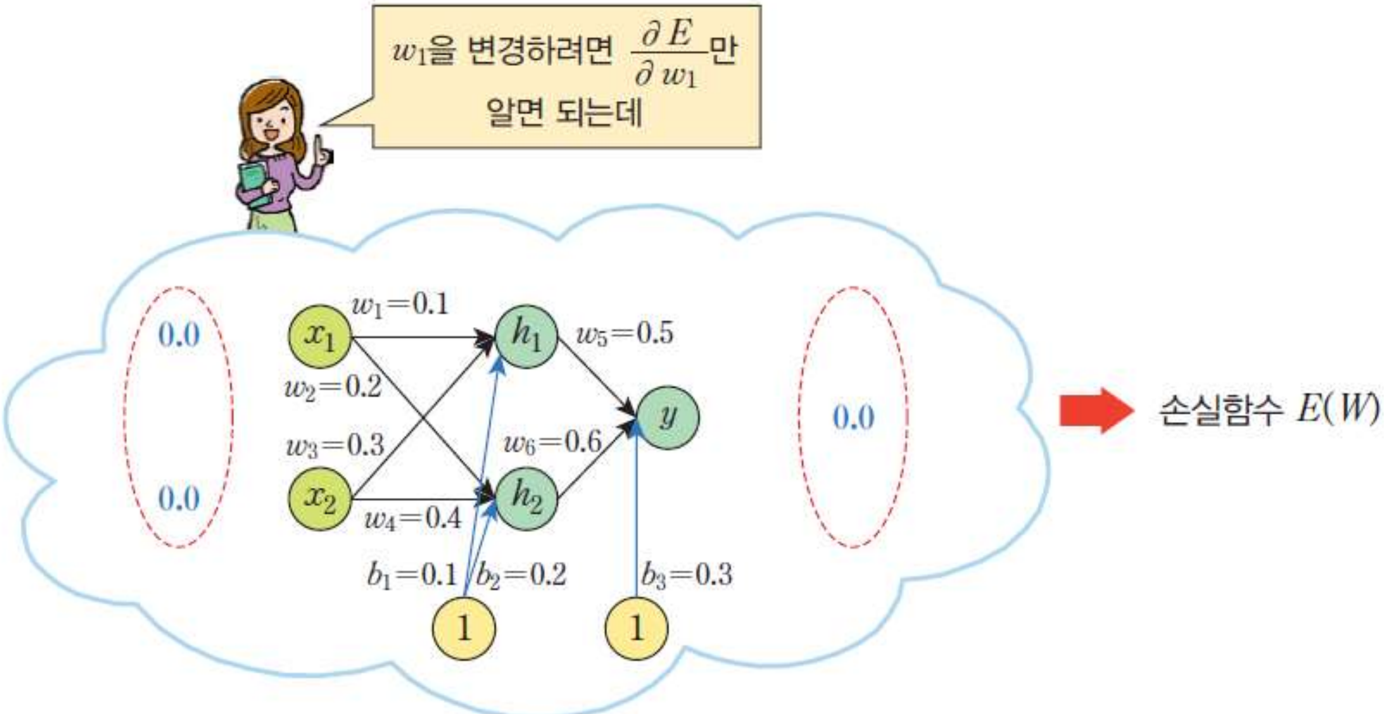
역전파 학습 알고리즘



가중치 *w1, w2*를 조금씩 변화시키며 손실 *(E(w1,w2))*이 최소로 되는 *(w1,w2)*를 찾는 과정

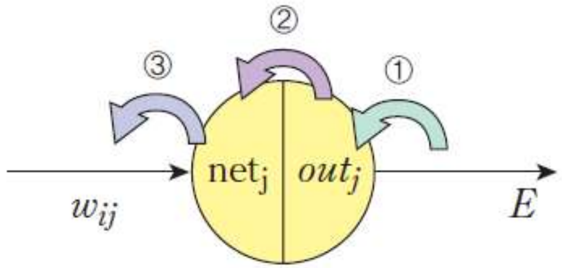
역전파 알고리즘의 유도 (생략 가능)

- 미분의 체인룰을 이용하여 유도가 가능하다.



출력층 유닛의 경우

$$\frac{\partial E}{\partial w_{ij}} = \underbrace{\frac{\partial E}{\partial out_j}}_{①}\underbrace{\frac{\partial out_j}{\partial \mathrm{net_j}}}_{②}\underbrace{\frac{\partial \mathrm{net_j}}{\partial w_{ij}}}_{③}$$



출력층 유닛 j

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

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

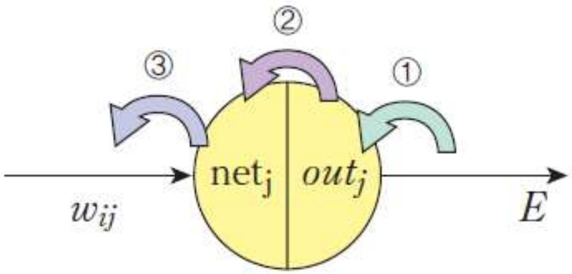
③ $$\frac{\partial \mathrm{net_j}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=0}^{n} w_{kj}out_k\right) = \frac{\partial}{\partial w_{ij}} w_{ij}\, out_i = out_i$$

가중치의 변화에 따른 net$_j$의 변화율이라고
할 수 있다.

$$\therefore \frac{\partial E}{\partial w_{ij}} = ①\times②\times③ = (out_j - target_j)\times f'(\mathrm{net_j})\times \mathrm{out_i}$$

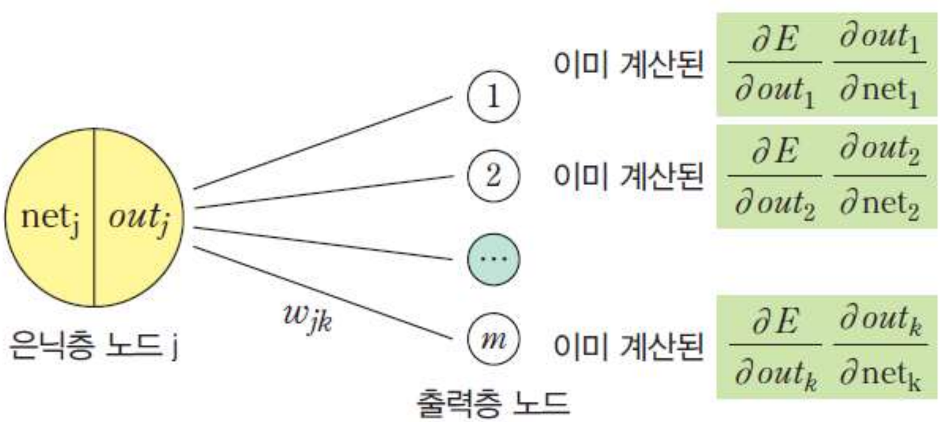
은닉층 유닛의 경우

$$\frac{\partial E}{\partial w_{ij}} = \underbrace{\frac{\partial E}{\partial out_j}}_{\textcircled{1}}\underbrace{\frac{\partial out_j}{\partial \mathrm{net}_j}}_{\textcircled{2}}\underbrace{\frac{\partial \mathrm{net}_j}{\partial w_{ij}}}_{\textcircled{3}}$$



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

이것도 체인룰이다!



델타

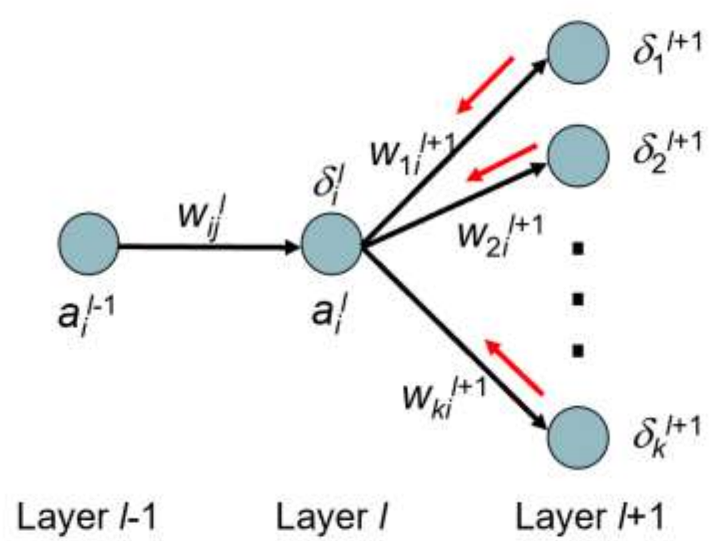
- $\frac{\partial E}{\partial out_k} \frac{\partial out_k}{\partial \mathrm{net}_k}$ 은 여러 문헌에서 델타라는 이름으로 불리는 값이다.

- 가중치가 보는 유닛 k에서의 "오차"라고 생각해도 된다.
- 이 델타가 신경망을 통하여 역전파된다.

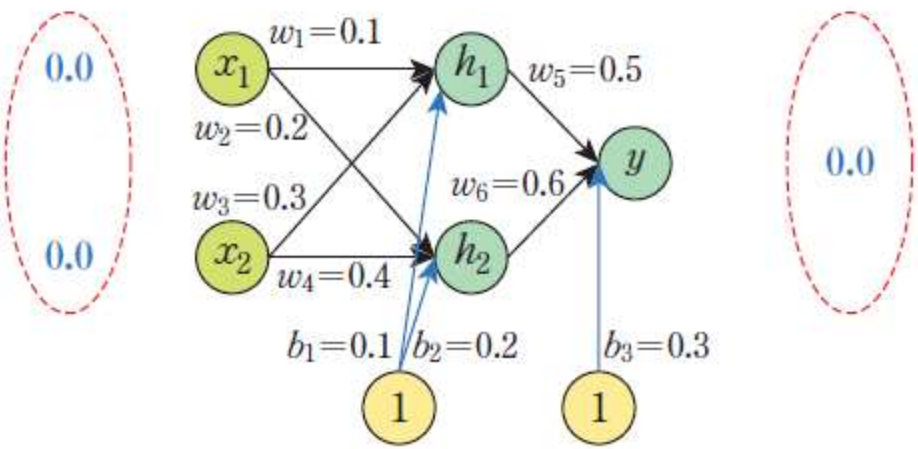
역전파 알고리즘 정리

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

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



역전파 알고리즘을 손으로 계산해보자.



- 순방향 패스

$$\text{net}_y = w_5 * out_{h1} + w_6 * out_{h2} + b_3$$
$$= 0.5 * 0.524979 + 0.6 * 0.549834 + 0.3 = 0.89239$$
$$out_y = \frac{1}{1+e^{-\text{net}_y}} = \frac{1}{1+e^{-0.89239}} = 0.709383$$

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

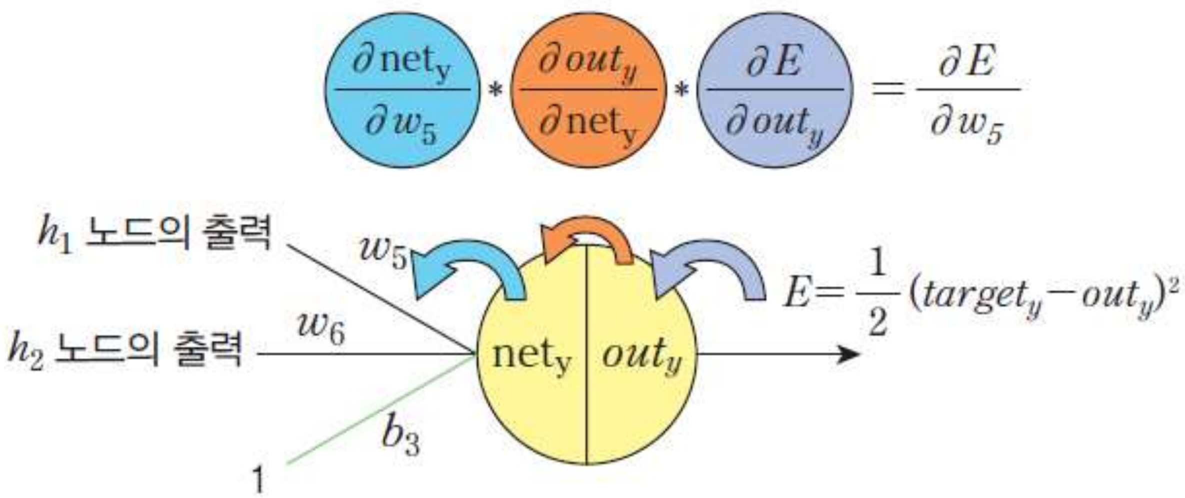
앞절 참조!

역전파 알고리즘을 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 $$\frac{\partial out_y}{\partial \mathrm{net}_y} = f'(out_y) = out_y{}^*(1 - out_y) = 0.709383^*(1 - 0.709383) = 0.206158$$

- ③ $$net_y = w_5 * out_{h1} + w_6 * out_{h2} + b_3 * 1$$
$$\frac{\partial net_y}{\partial w_5} = 1 * out_{h1} + 0 + 0 = 0.524979$$

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial out_y} \frac{\partial out_y}{\partial \mathrm{net}_y} \frac{\partial \mathrm{net}_y}{\partial w_5}$$
$$= 0.709383^*0.206158^*0.524979 = 0.076775$$

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

은닉층->출력층의 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 원하는 출력값은 0이기 때문이다.

바이어스는 기존 값 보다 낮아지게 된다. 따라서 다음 번에는 유닛의 출력을 더 낮게 만들 것 이다. 현재 우리가 원하는 출력값은 0이기 때문이다.

입력층->은닉층의 가중치와 바이어스

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

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

입력값이 0이어서 가중치는 변경되지 않았다(이점은 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 원하는 출력값은 0이기 때문이다.

손실함수 평가 (훈련 결과 → 오차 최소화!)

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

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 학습시킬수 있다.

- 역전파할 때는 가중치 행렬를 전치시켜서 사용한다.
- 바이어스는 입력을 1.0으로 고정하고, 이 입력에 붙은 가중치로 생각한다.

넘파이를 이용한 *MLP* 구현

mlp2.py

```
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([[0], [1], [1], [0]])
```

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                          # 입력을 layer0에 대입한다.
        Z1 = np.dot(layer0, W1)+B1          # 행렬의 곱을 계산한다.
        layer1 = actf(Z1)                   # 활성화 함수를 적용한다.
        Z2 = np.dot(layer1, W2)+B2          # 행렬의 곱을 계산한다.
        layer2 = actf(Z2)                   # 활성화 함수를 적용한다.
        return layer0, layer1, layer2
```

오차 역전파 구현

```
# 역방향 전파 계산
def fit():
    global W1, W2, B1, B2                      # 우리는 외부에 정의된 변수를 변경해야 한다.
    for i in range(90000):                     # 9만번 반복한다.
        for x, y in zip(X, T):                 # 학습 샘플을 하나씩 꺼낸다.
            x = np.reshape(x, (1, -1))         # 2차원 행렬로 만든다. ①
            y = np.reshape(y, (1, -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)        #
```

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

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()
```

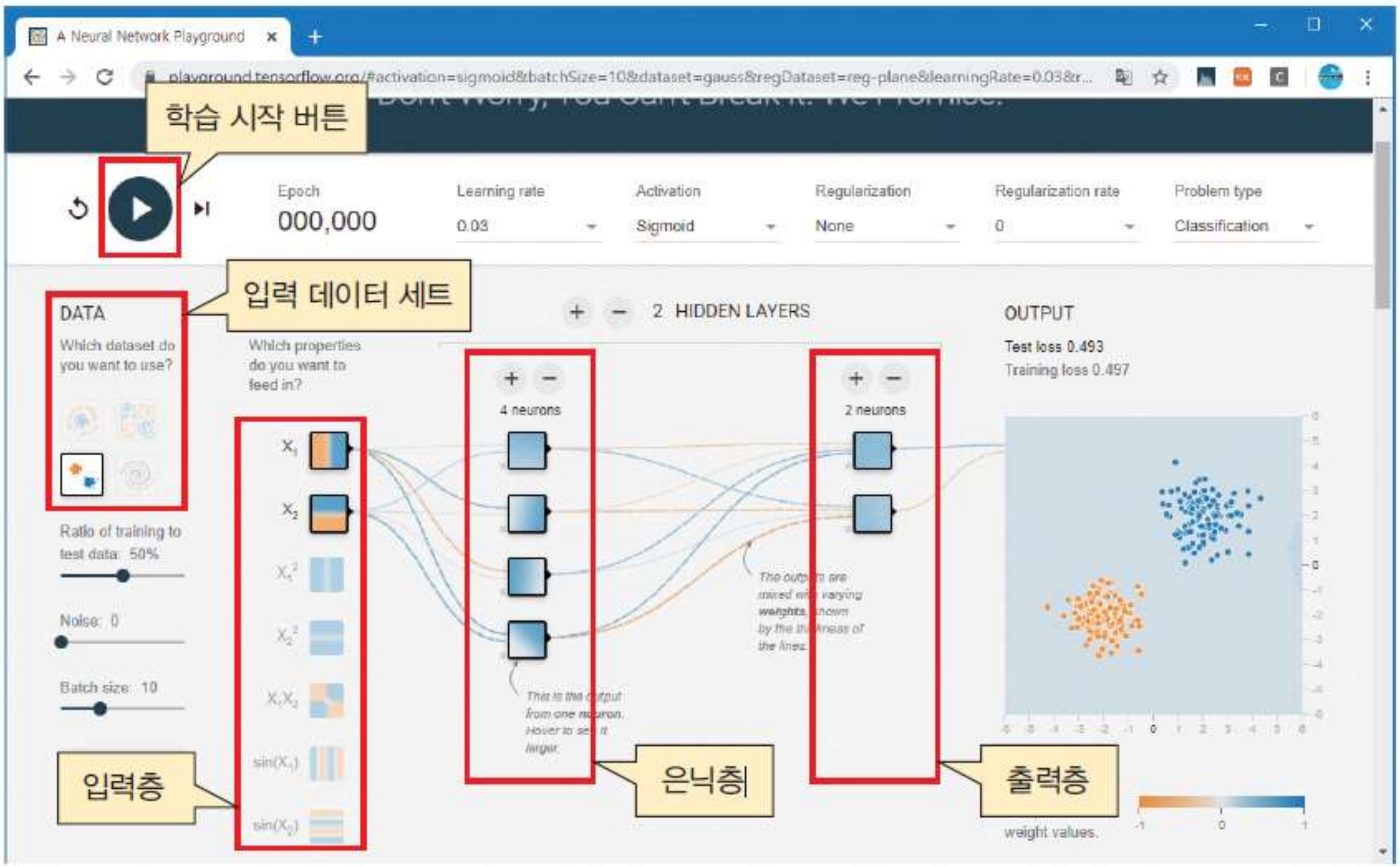
```
[[0 0]] [0] [[0.99196032]]
[[0 1]] [1] [[0.00835708]]
[[1 0]] [1] [[0.00836107]]
[[1 1]] [0] [[0.98974873]]
```

```
[[0 0]] [0] [[0.00814407]]
[[0 1]] [1] [[0.99154105]]
[[1 0]] [1] [[0.99152258]]
[[1 1]] [0] [[0.01038517]]
[[-6.7181009  -5.22651178]
 [-6.75291089 -5.23369975]]
[[-11.19451524]
 [ 11.04437812]]
[2.8151334  7.80251731] [-5.28026114]
```

구글의 플레이그라운드

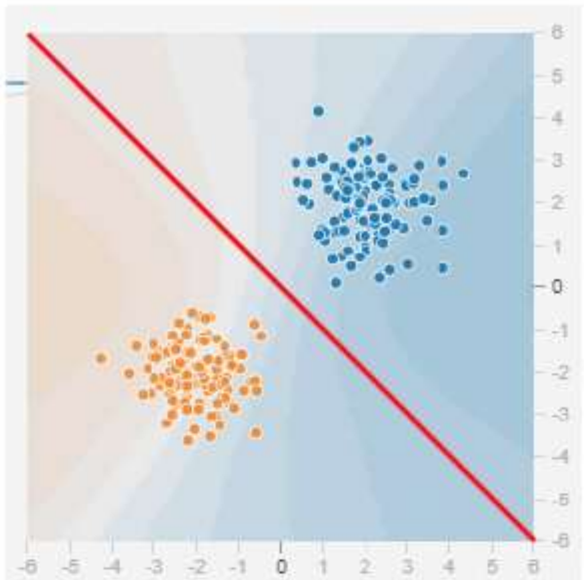
- 사이트(https://playground.tensorflow.org)
- 텐서 플로우 플레이그라운드는 자바 스크립트로 작성된 웹 애플리케이션으로 웹 브라우저에서 실행
- 이 사이트에서는 사용자가 딥러닝 모델을 구성하고 여러 가지 매개변수를 조정하면서 실험할 수 있는 기능을 제공한다.

구글의 플레이그라운드



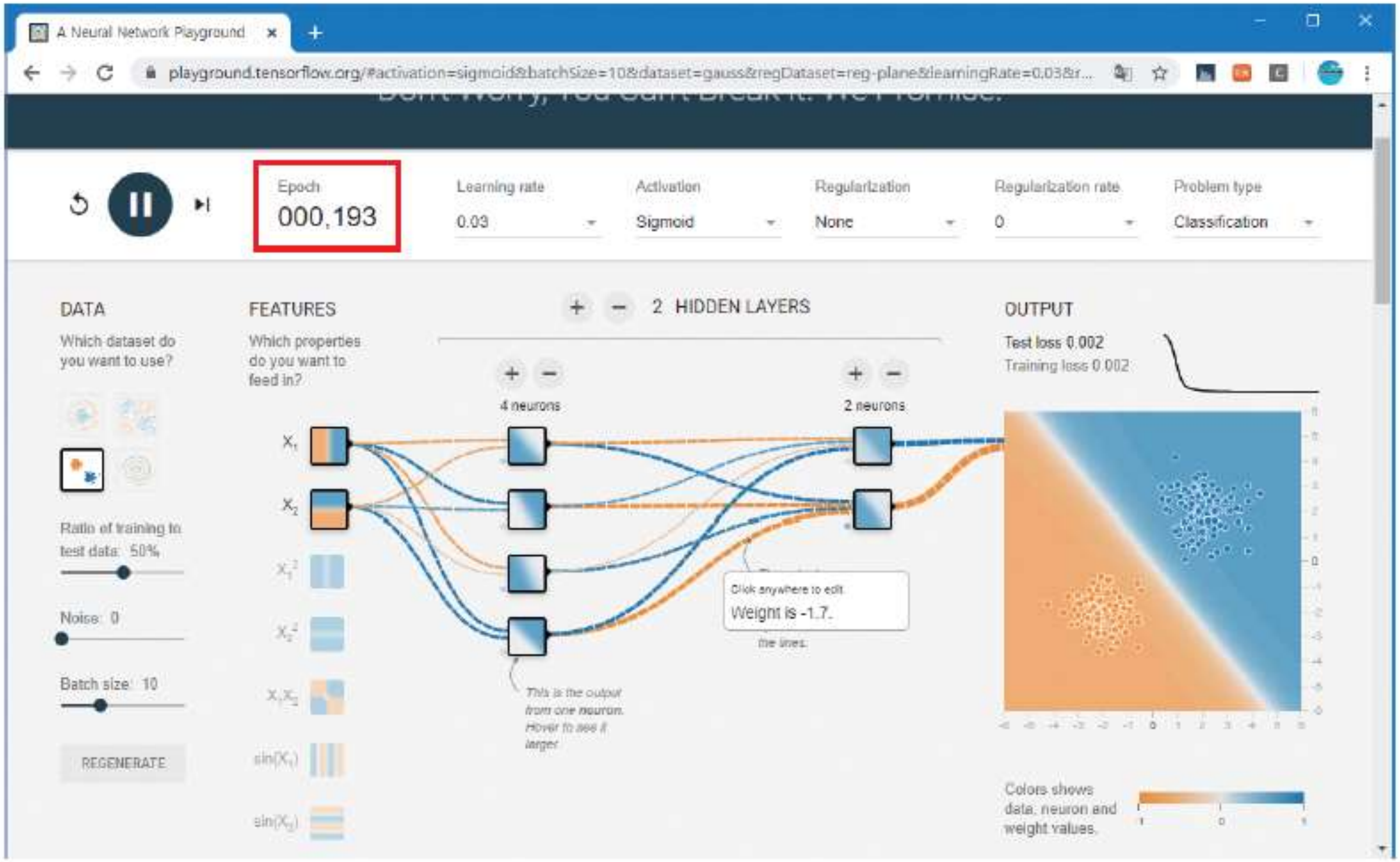
A Neural Network Playground
학습 시작 버튼
Epoch
000,000
Learning rate
0.03
Activation
Sigmoid
Regularization
None
Regularization rate
0
Problem type
Classification
입력 데이터 세트
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
Which properties do you want to feed in?
2 HIDDEN LAYERS
4 neurons
2 neurons
OUTPUT
Test loss 0.493
Training loss 0.497
입력층
은닉층
출력층
weight values.

선형 분리 가능한 입력 데이터

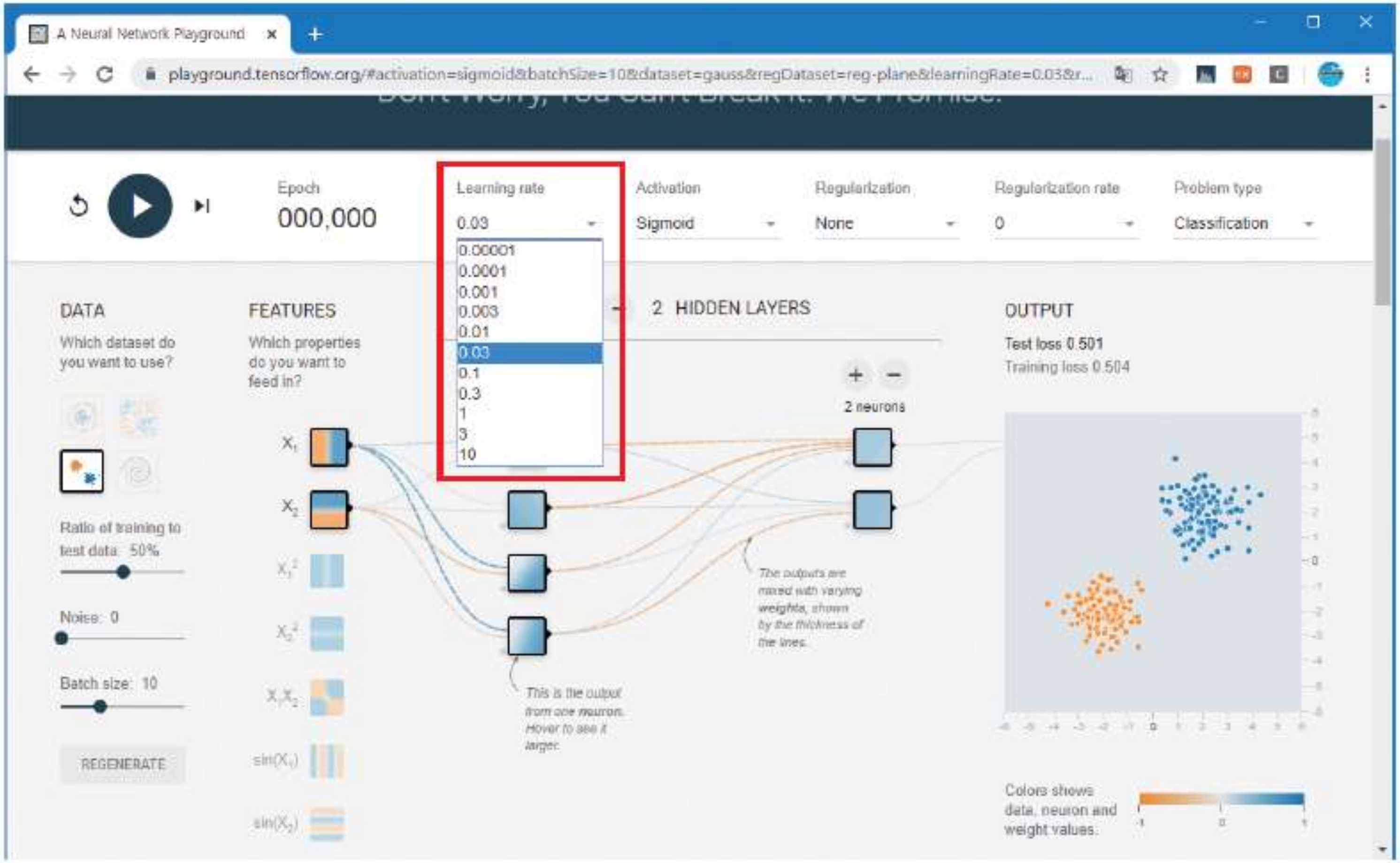


$$w_1x_1 + w_2x_2 + b = 0$$

에포크

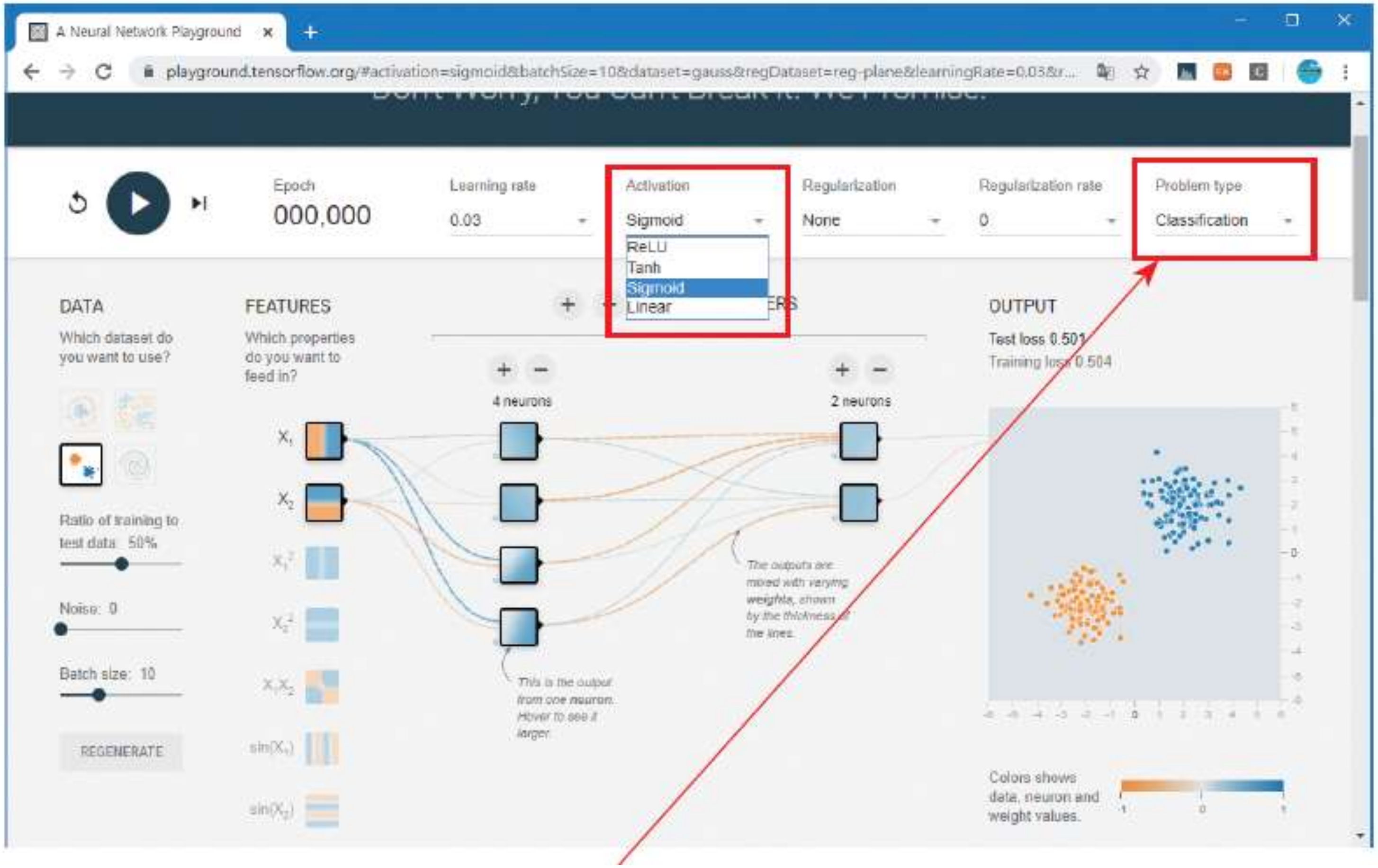

A Neural Network Playground
playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&r...
Epoch
000,193
Learning rate
0.03
Activation
Sigmoid
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
2 HIDDEN LAYERS
4 neurons
2 neurons
Click anywhere to edit.
Weight is -1.7.
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.

학습률



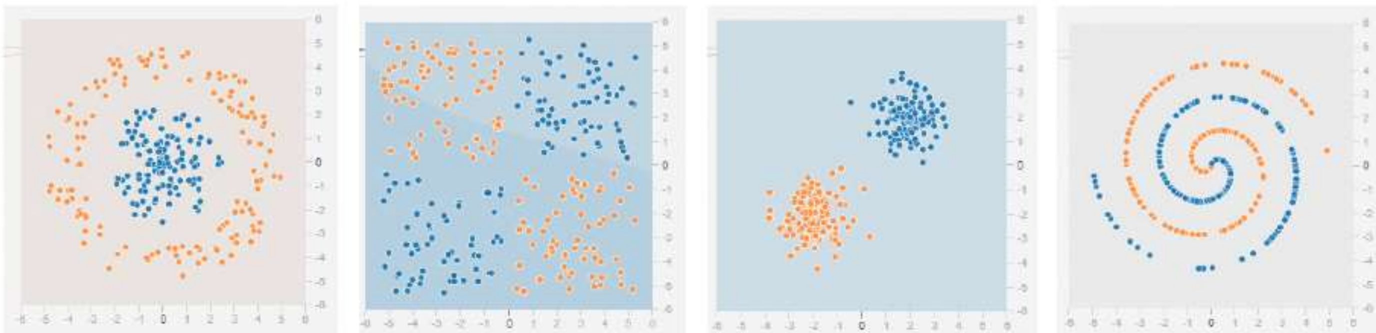
A Neural Network Playground
playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&r...
Epoch
000,000
Learning rate
0.03
0.00001
0.0001
0.001
0.003
0.01
0.03
0.1
0.3
1
3
10
Activation
Sigmoid
Regularization
None
Regularization rate
0
Problem type
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?
2 HIDDEN LAYERS
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 0.501
Training loss 0.504
Colors shows data, neuron and weight values.

활성화 함수 선택

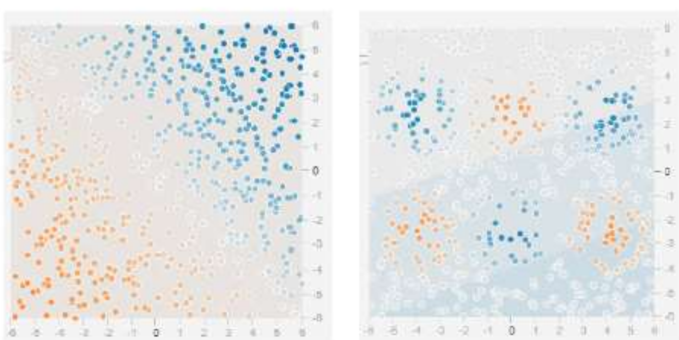

A Neural Network Playground
playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&r...
Epoch
000,000
Learning rate
0.03
Activation
Sigmoid
ReLU
Tanh
Sigmoid
Linear
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which 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?
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.
OUTPUT
Test loss 0.501
Training loss 0.504
Colors shows data, neuron and weight values.

문제 유형

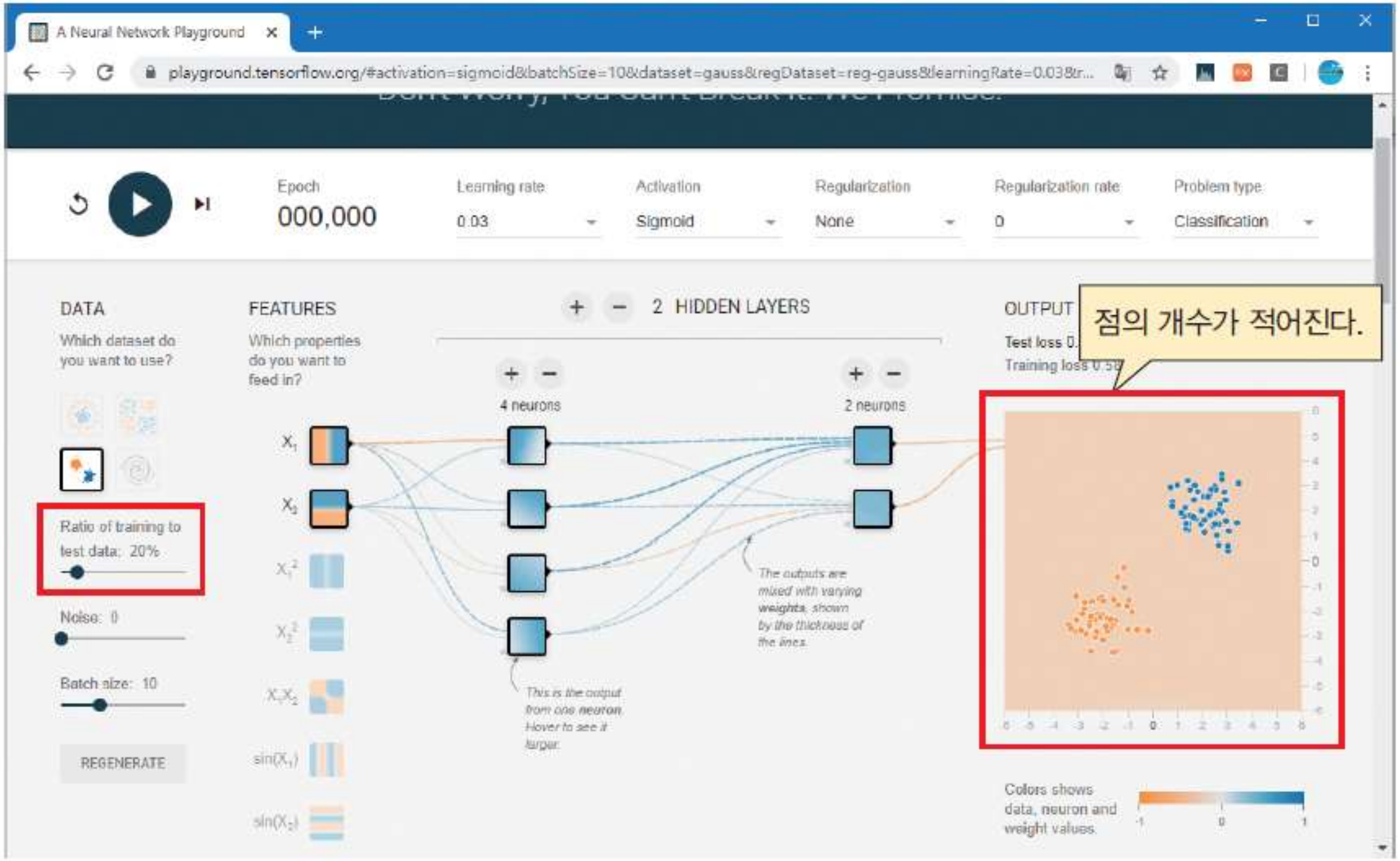
- 분류 문제



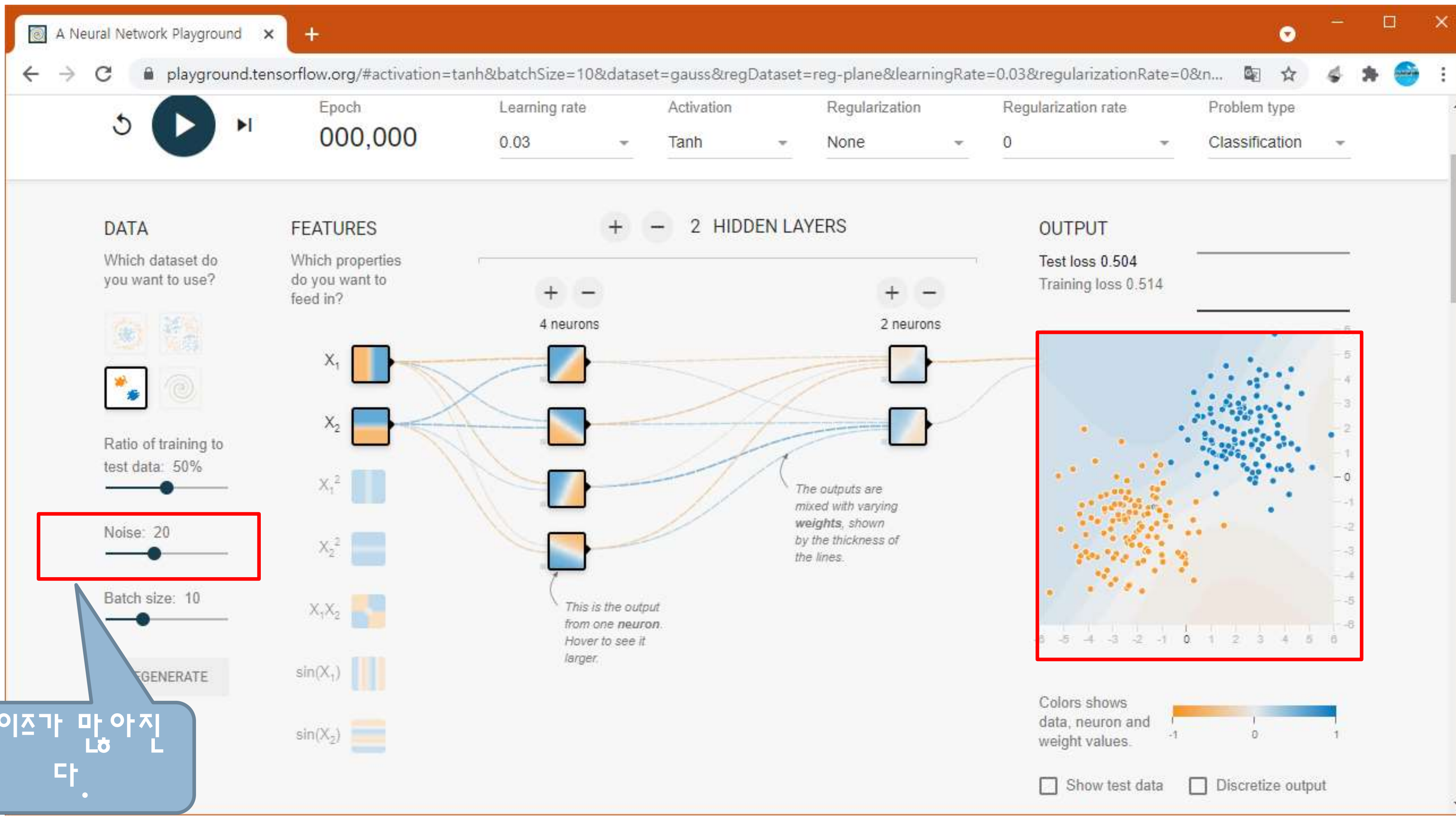
- 회귀 문제



학습 데이터와 테스트 데이터의 비율

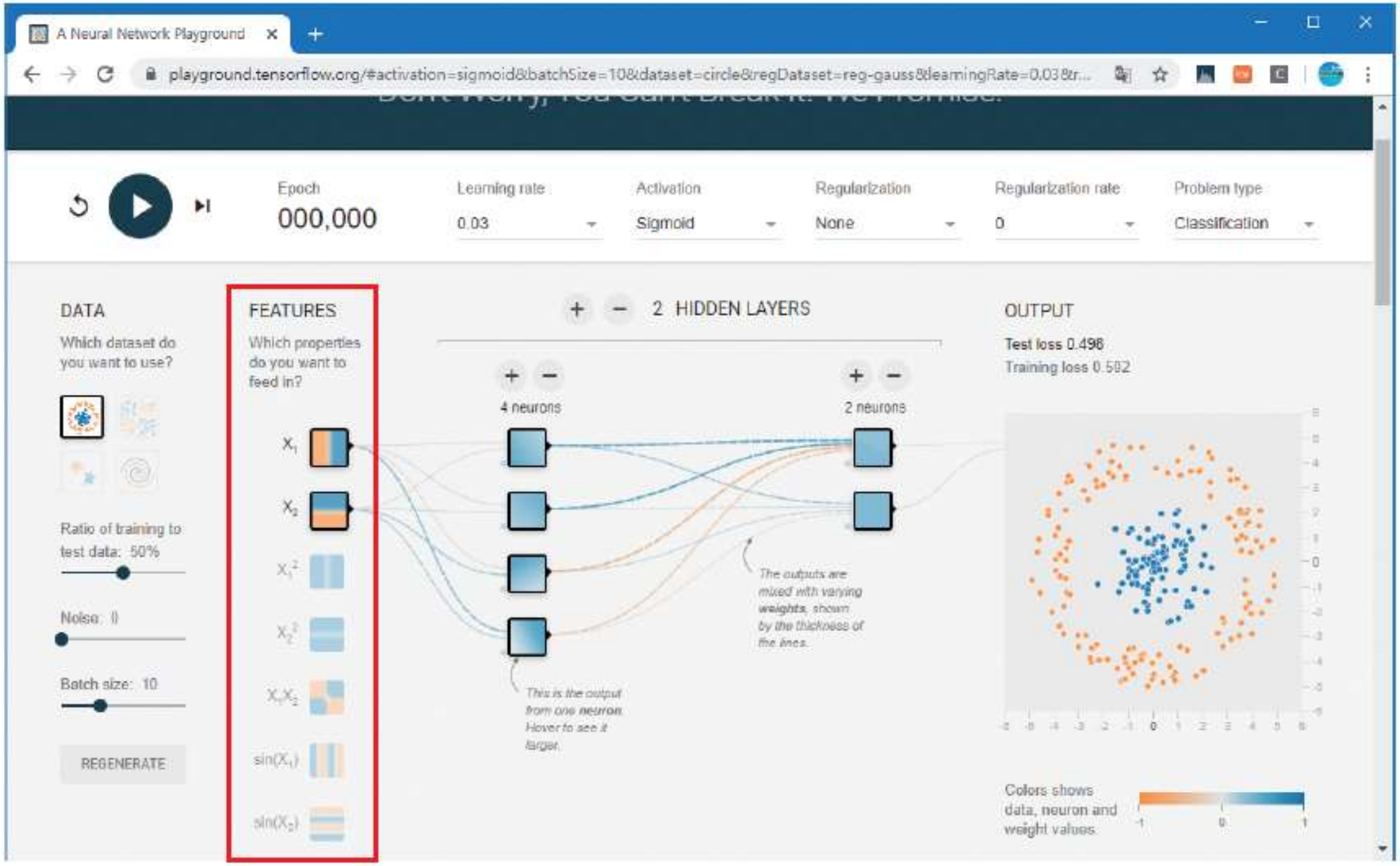


노이즈 비율

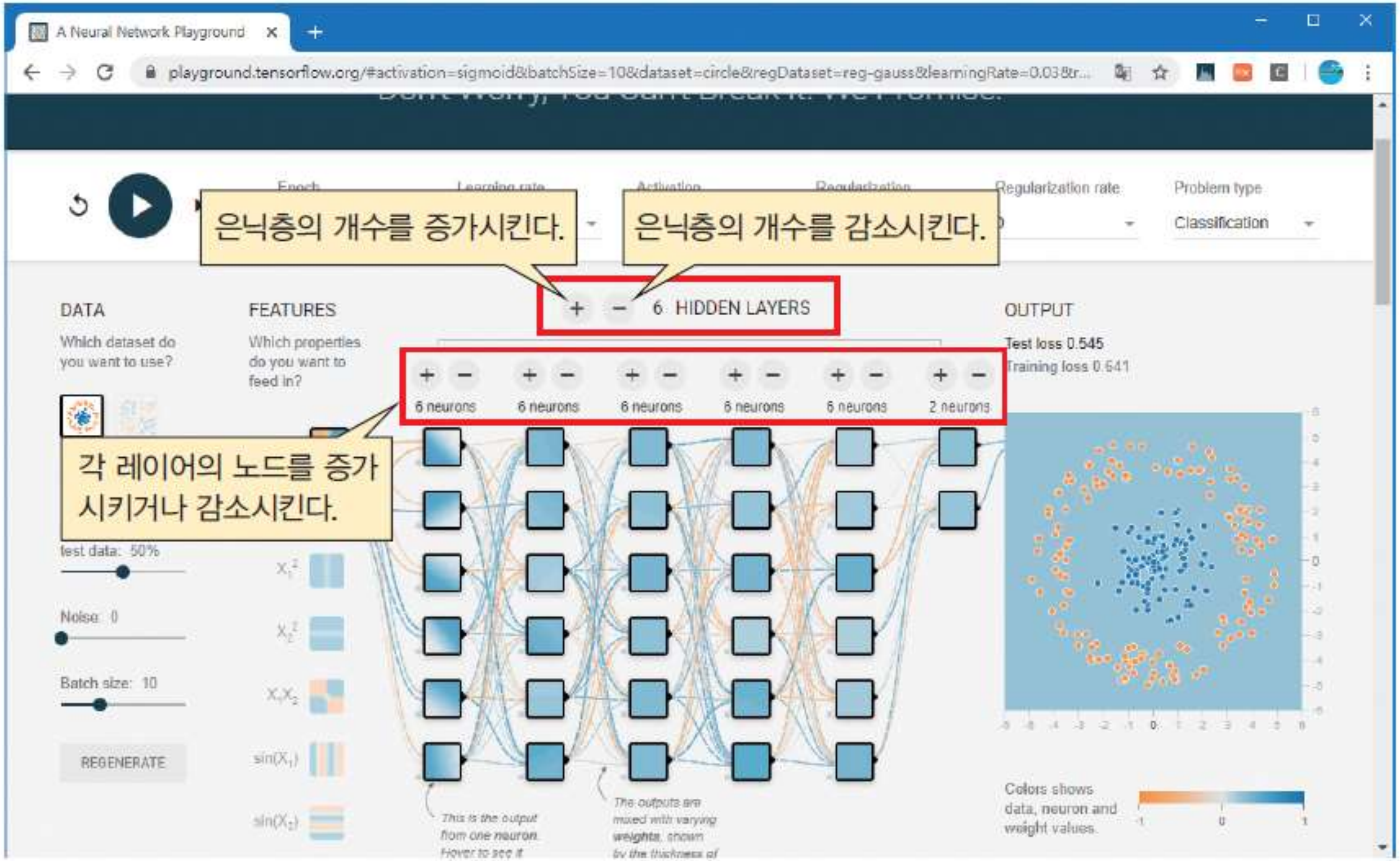


노이즈가 많아진다.

입력 특징 선택

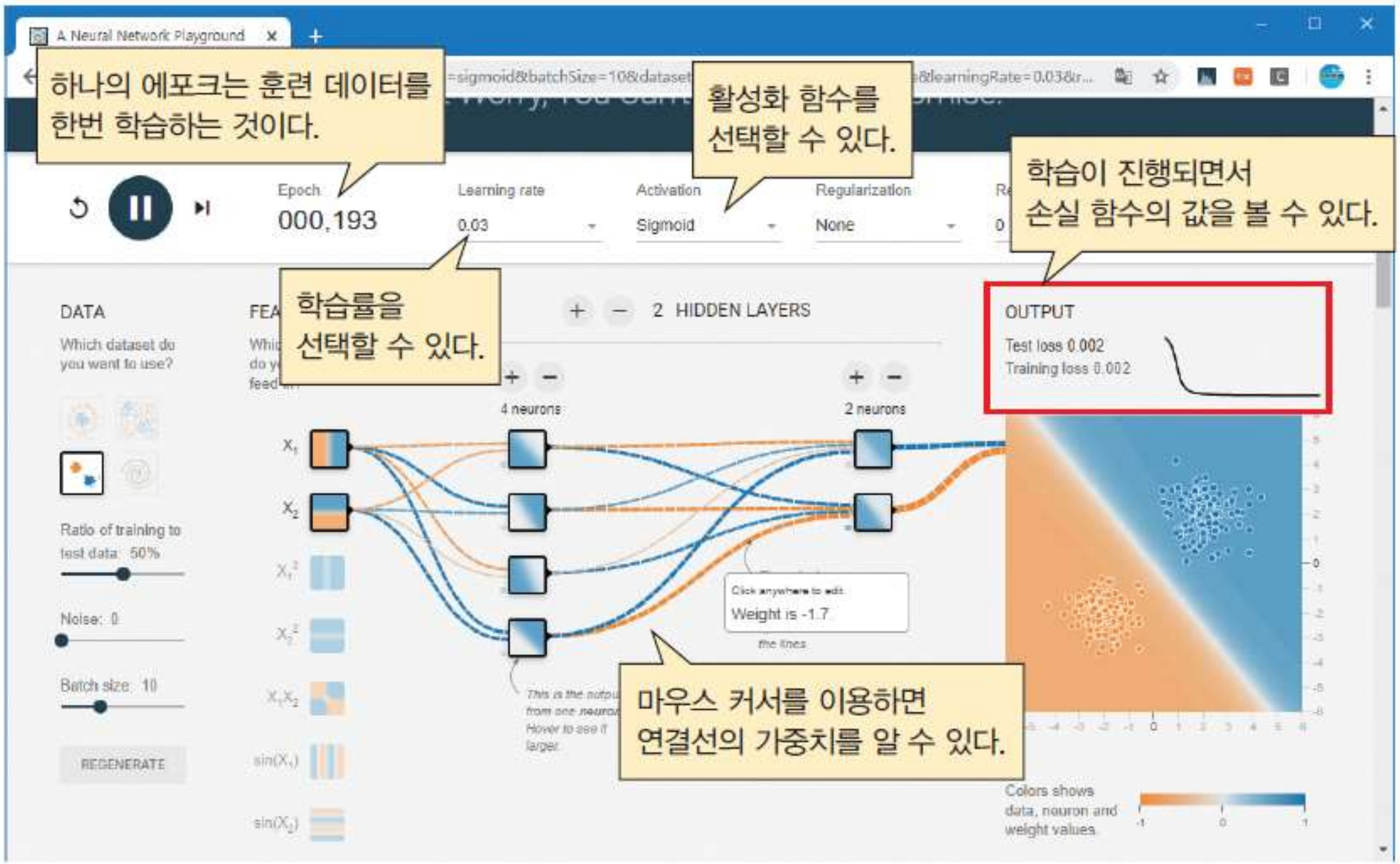

A Neural Network Playground
playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=circle&regDataset=reg-gauss&learningRate=0.03&r...
Epoch
000,000
Learning rate
0.03
Activation
Sigmoid
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
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.
OUTPUT
Test loss 0.498
Training loss 0.502
Colors shows data, neuron and weight values.

은닉층 추가하기



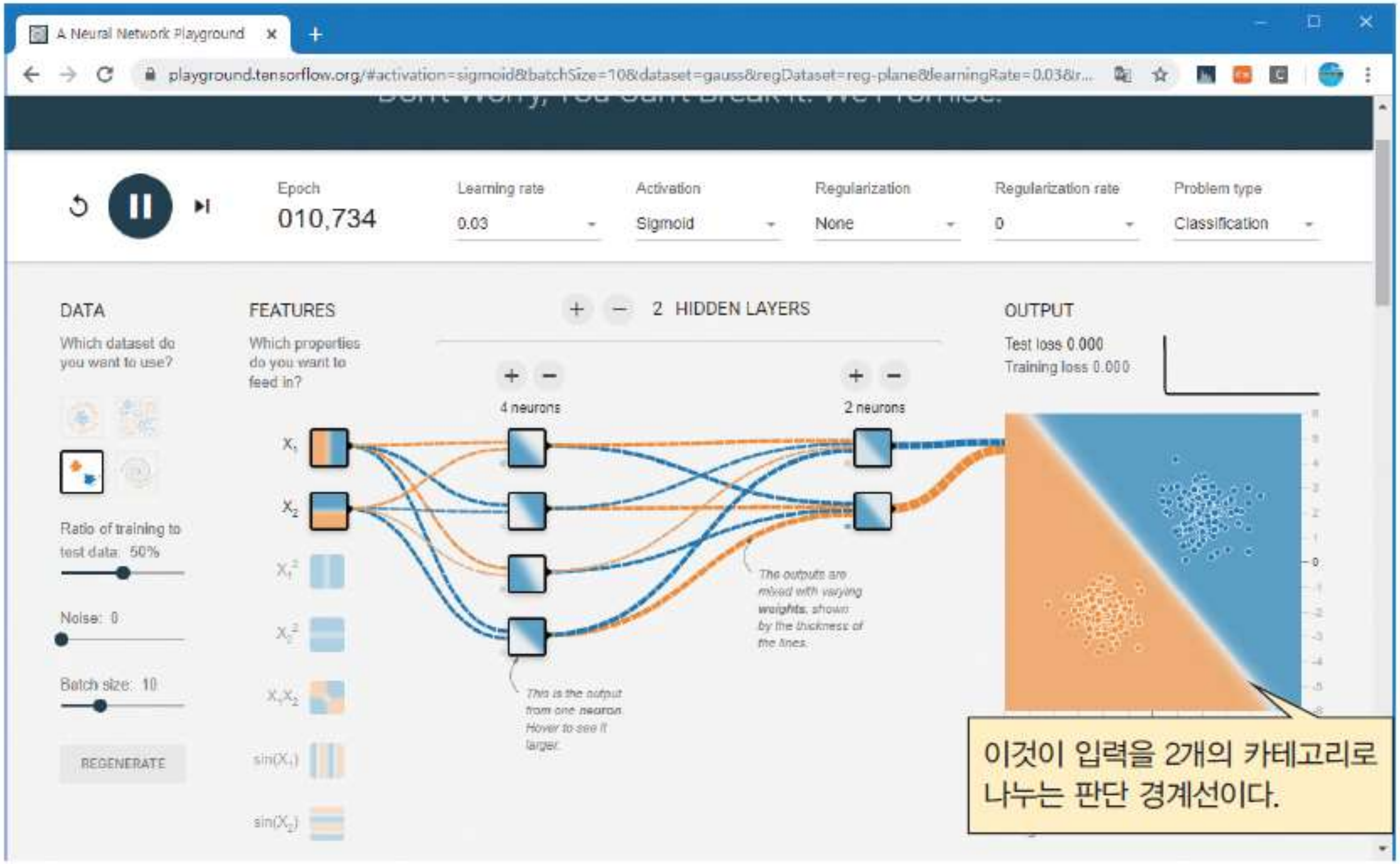
A Neural Network Playground
playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=circle&regDataset=reg-gauss&learningRate=0.03&r...
은닉층의 개수를 증가시킨다.
은닉층의 개수를 감소시킨다.
Regularization rate
Problem type
Classification
DATA
Which dataset do you want to use?
FEATURES
Which properties do you want to feed in?
6 HIDDEN LAYERS
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
Colors shows data, neuron and weight values.

학습 시작

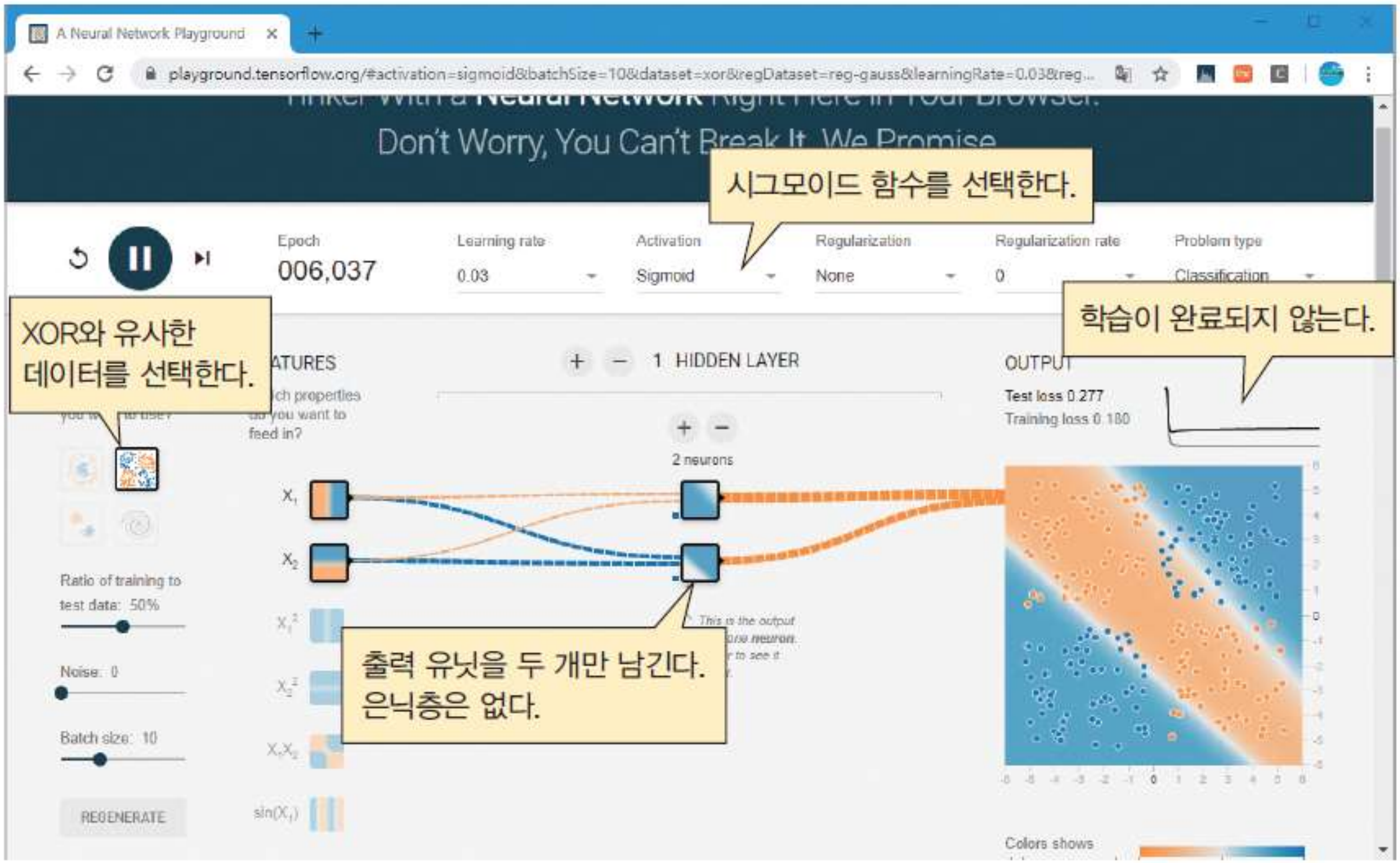


하나의 에포크는 훈련 데이터를 한번 학습하는 것이다.
활성화 함수를 선택할 수 있다.
학습이 진행되면서 손실 함수의 값을 볼 수 있다.
학습률을 선택할 수 있다.
마우스 커서를 이용하면 연결선의 가중치를 알 수 있다.
A Neural Network Playground
Epoch
000,193
Learning rate
0.03
Activation
Sigmoid
Regularization
None
DATA
Which dataset do you want to use?
2 HIDDEN LAYERS
4 neurons
2 neurons
OUTPUT
Test loss 0.002
Training loss 0.002
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
Click anywhere to edit.
Weight is -1.7
Colors shows data, neuron and weight values.

학습 완료

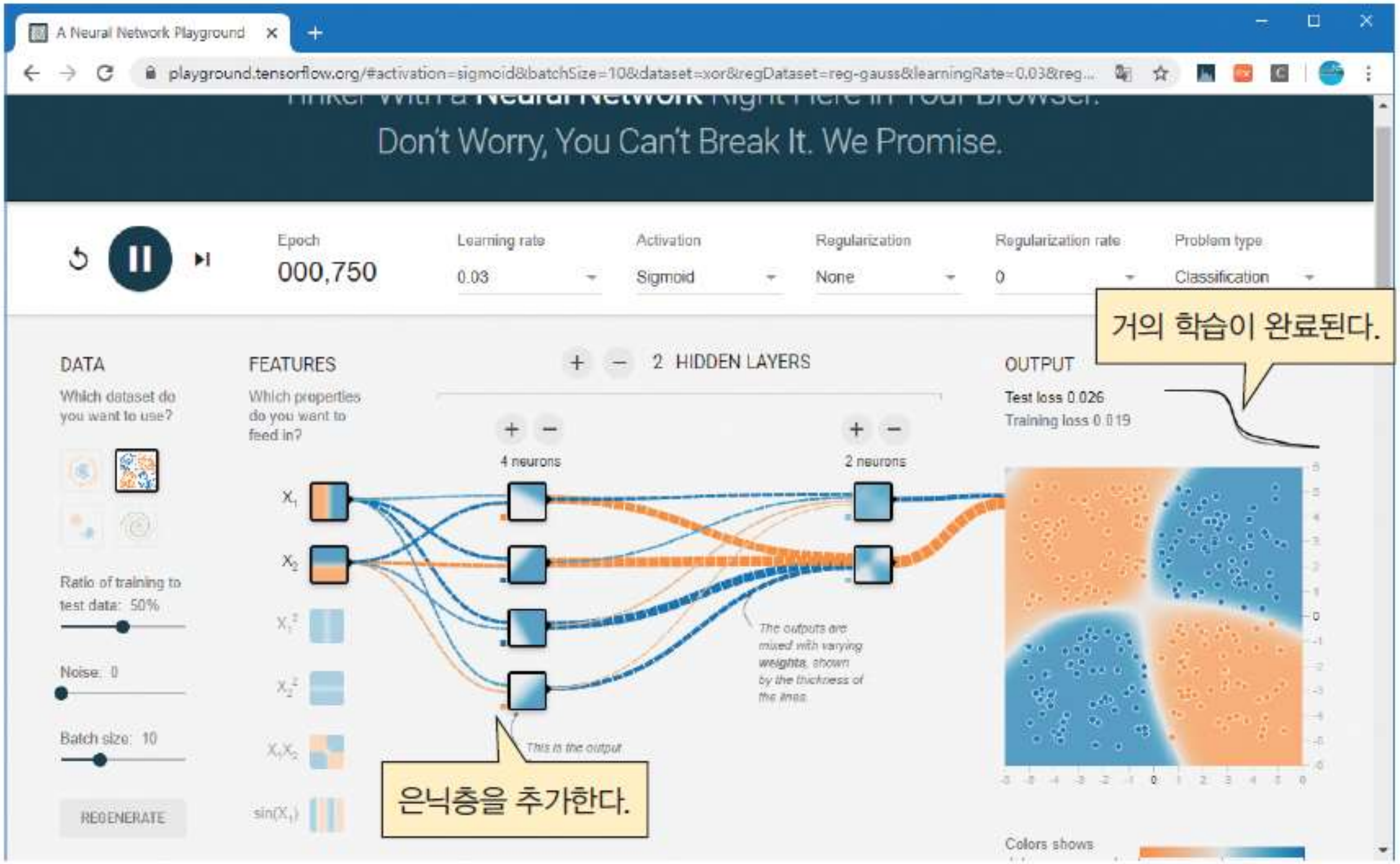

A Neural Network Playground
playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&r...
Epoch
010,734
Learning rate
0.03
Activation
Sigmoid
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
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.
OUTPUT
Test loss 0.000
Training loss 0.000
이것이 입력을 2개의 카테고리로 나누는 판단 경계선이다.

은닉층 없이 분류 실습



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.
시그모이드 함수를 선택한다.
Epoch
006,037
Learning rate
0.03
Activation
Sigmoid
Regularization
None
Regularization rate
0
Problem type
Classification
XOR와 유사한 데이터를 선택한다.
학습이 완료되지 않는다.
1 HIDDEN LAYER
2 neurons
OUTPUT
Test loss 0.277
Training loss 0.180
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
출력 유닛을 두 개만 남긴다. 은닉층은 없다.
Colors shows

은닉층을 추가한 실습


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
거의 학습이 완료된다.
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?
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
은닉층을 추가한다.
OUTPUT
Test loss 0.026
Training loss 0.019
Colors shows

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 **재발견**되었다. 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