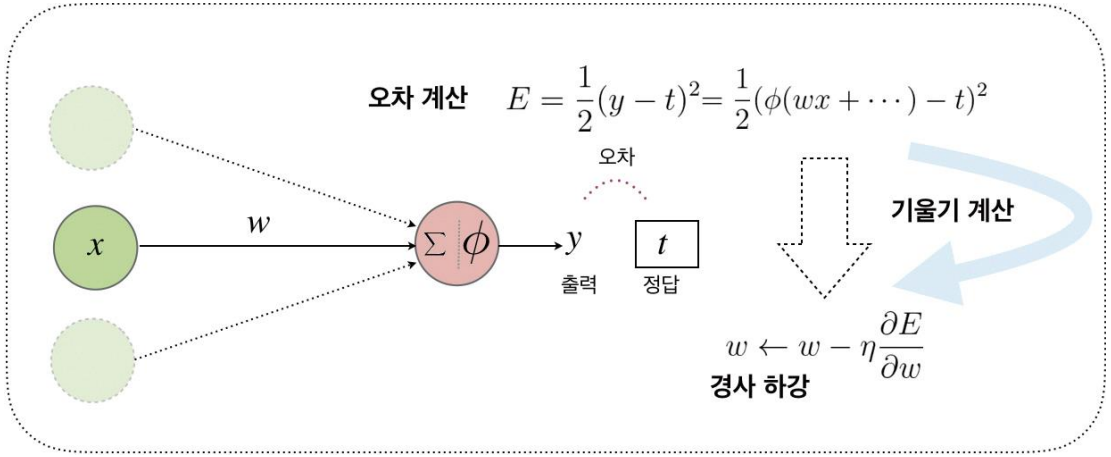
인공 신경망 기초

Part2 오류역전파로 다층 퍼셉트론 학습

경사 하강법에 의한 연결강도 개선

- 인공 신경망에서 연결강도를 조정해 오차를 줄여 보자
- 오차 곡면의 기울기를 연결강도에 대해 구한 뒤에 이 기울기를 따라 내려가는 **경사 하강법**gradient descent을 사용하여 구현할 수 있음
- x가 연결강도 w로 출력 노드에 연결되어 있다고 하자.
- 출력 노드는 x를 비롯한 여러 노드에서 신호를 수신하여 합산하는 부분과 이 값을 활성화함수에 넣어 최종 출력을 결정하는 부분으로 나뉨
- 출력과 정답(목표값)의 차이를 제곱하여 오차를 구하는 일반적인 방법을 사용 가능
- 출력 노드의 출력값 y가 $\phi(wx+\cdots)$이므로, 오차 E는 $1/2\,(\phi(wx+\cdots)-t)^2$



- 오차를 연결강도 w에 대해 편미분

$$\frac{\partial E}{\partial w} = \frac{1}{2}\frac{\partial}{\partial w}(\phi(wx+\cdots)-t)^2$$

$$= (\phi(wx+\cdots)-t)\cdot\frac{\partial}{\partial w}\phi(wx+\cdots)$$

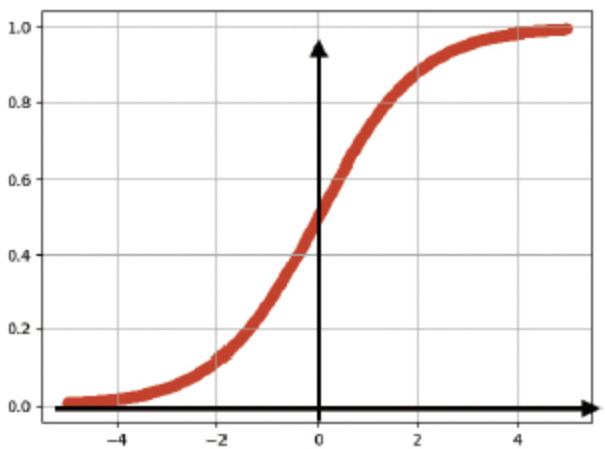
$$= (\phi(wx+\cdots)-t)\cdot\phi'(wx+\cdots)\cdot\frac{\partial}{\partial w}(wx+\cdots)$$

$$= (y-t)\cdot\boxed{\phi'(wx+\cdots)}\cdot x$$

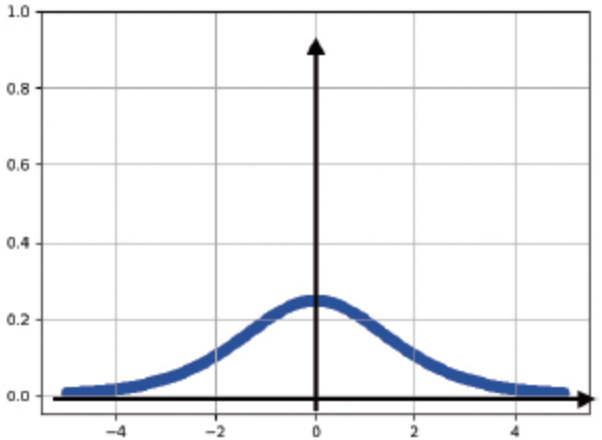
계단 함수는 미분이 불가능

- 기울기를 이용하여 연결강도를 수정
- 활성화 함수 $\phi(\,)$의 미분 $\phi'(\,)$ 만 안다면 입력 x, 출력 y, 목표값 t를 이용하여 간단히 계산할 수 있음
- 계단함수는 미분이 불가능한 지점이 있고, 미분이 되는 곳에서도 미분치가 언제나 0
  - 이 방법을 위해서는 미분이 가능한 새로운 활성화 함수가 필요
  - 신경망에서는 다음과 같은 **시그모이드**sigmoid 혹은 **로지스틱**logistic 함수를 활성화 함수로 도입

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



$$\phi'(x) = \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right)$$



- 이 함수는 미분이 가능하다는 장점과 함께, 이 함수의 미분은 다음과 같이 원래의 함수를 이용하여 표현할 수 있는 장점이 있음

$$\phi'(x) = \phi(x)(1-\phi(x))$$

- 조금 전 구했던 오차의 기울기에서 활성화 함수의 미분으로 표시된 부분을 출력값 y로 표현할 수 있음

$$\begin{aligned}\frac{\partial E}{\partial w} &= (y-t)\cdot\phi'(wx+\cdots)\cdot x \\ &= (y-t)\cdot\phi(wx+\cdots)\cdot(1-\phi(wx+\cdots))\cdot x \\ &= (y-t)\cdot y\cdot(1-y)\cdot x\end{aligned}$$

- 출력값과 목표값만 알면 출력의 오차를 줄이는 연결강도 변경을 아래의 수식을 사용하여 수행할 수 있음

$$w \leftarrow w - \eta \cdot \partial E / \partial w$$

$$w \leftarrow w - \eta(y - t)y'x$$

$$w \leftarrow w - \eta(y - t)y(1 - y)x$$

- **다항 퍼셉트론에 적용**

경사하강법을 적용한 다항퍼셉트론

```
[1]  import numpy as np
```

```
[2]  W_poly = np.array([0.0, 0.0, 0.0, 0.0])      # 파라미터
     learning_rate = 10.1                          # 하이퍼파라미터
```

```
[3]  def poly_transform(x0, x1) :
         return np.array([x0, x1, x0*x1, 1])

     def aggregate(x0, x1):
         X_poly = poly_transform(x0, x1)
         return W_poly.dot(X_poly)

     def phi(v):
         return 1/(1+np.exp(-v))
```

```
[4]  def 다항퍼셉트론(x0, x1):
         return phi(aggregate(x0, x1))
```

- **경사 하강 학습**

```
[6]  ### 학습이 이루어지게 만들자
     def 경사하강학습(x0, x1, target):

         global W_poly
         X_poly = poly_transform(x0, x1)

         출력 = 다항퍼셉트론(x0, x1)
         출력미분 = 출력 * (1-출력)
         E = 출력 - target
         delta = E * 출력미분
         gradient = delta * X_poly
         W_poly -= learning_rate * gradient

         return E**2
```

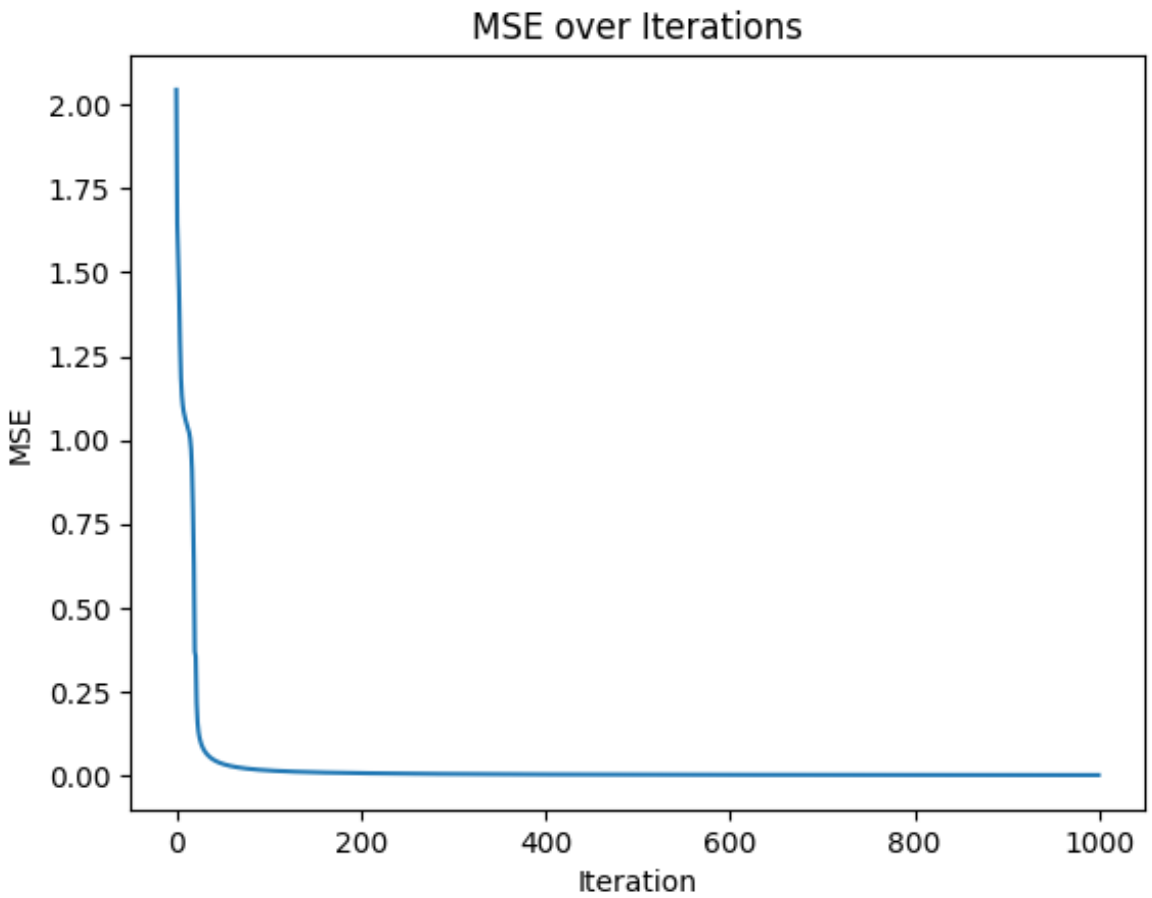
$$w \leftarrow w - \eta(y - t)y(1 - y)x$$

- **경사 하강 학습 실시**

```
### XOR 학습을 실시하자

MSE_list = []
for i in range(1000):
    MSE = 0.0
    MSE += 경사하강학습(0, 0, 0)
    MSE += 경사하강학습(0, 1, 1)
    MSE += 경사하강학습(1, 0, 1)
    MSE += 경사하강학습(1, 1, 0)
    MSE_list.append(MSE)
    if MSE < 0.001:
        print(f'{i}회 반복 후 종료')
        break
```

- 경사 하강 학습 실시

```
# prompt: draw MSE_list with pyplot

import matplotlib.pyplot as plt

plt.plot(MSE_list)
plt.xlabel('Iteration')
plt.ylabel('MSE')
plt.title('MSE over Iterations')
plt.show()
```



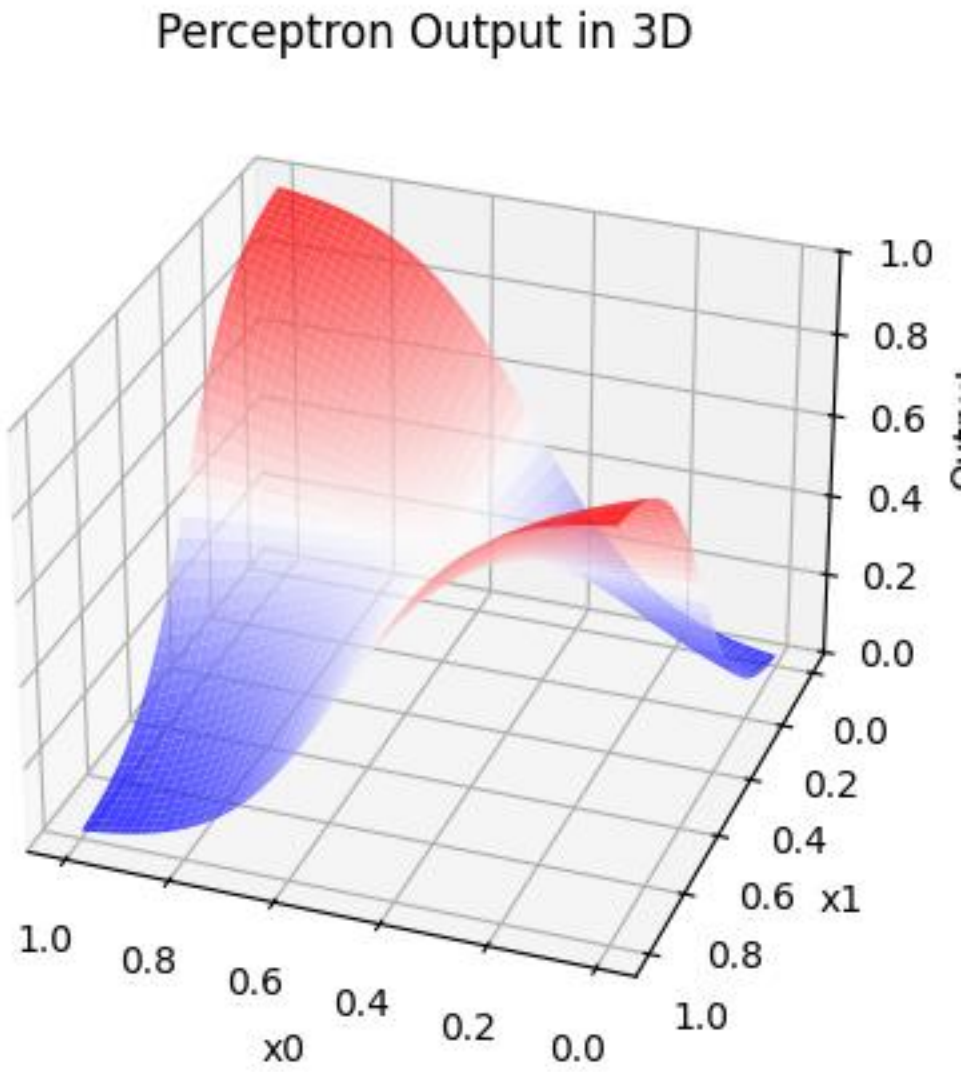
- **학습결과**

```
for x in [ [0, 0], [0, 1], [1, 0], [1, 1]]:
    print(x[0], x[1], 다항퍼셉트론(x[0], x[1]))

W_poly
```

```
0 0 0.021782815806432067
0 1 0.9823602267351613
1 0 0.982295273483868
1 1 0.014320802960500499
array([  7.82067063,   7.8244122 , -16.07209022,  -3.80461032])
```

- 0,1 입력이 아닌 연속적인 입력이 가능한 퍼셉트론

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Create a meshgrid for visualization
x0_vals = np.linspace(0, 1, 150)
x1_vals = np.linspace(0, 1, 150)
x0_grid, x1_grid = np.meshgrid(x0_vals, x1_vals)

# Calculate the output of the perceptron for each point in the grid
output_grid = np.zeros_like(x0_grid, dtype=float)
for i in range(x0_grid.shape[0]):
    for j in range(x0_grid.shape[1]):
        output_grid[i, j] = 다항퍼셉트론(x0_grid[i, j], x1_grid[i, j])

# Create the 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot the surface
ax.plot_surface(x0_grid, x1_grid, output_grid, cmap='bwr', alpha=0.75)

# Add labels and title
ax.set_xlabel('x0')
ax.set_ylabel('x1')
ax.set_zlabel('Output')
ax.set_title('Perceptron Output in 3D')
ax.view_init(elev=30, azim=110)
# Show the plot
plt.show()
```



- 출력값과 목표값만 알면 출력의 오차를 줄이는 연결강도 변경을 아래의 수식을 사용하여 수행할 수 있음

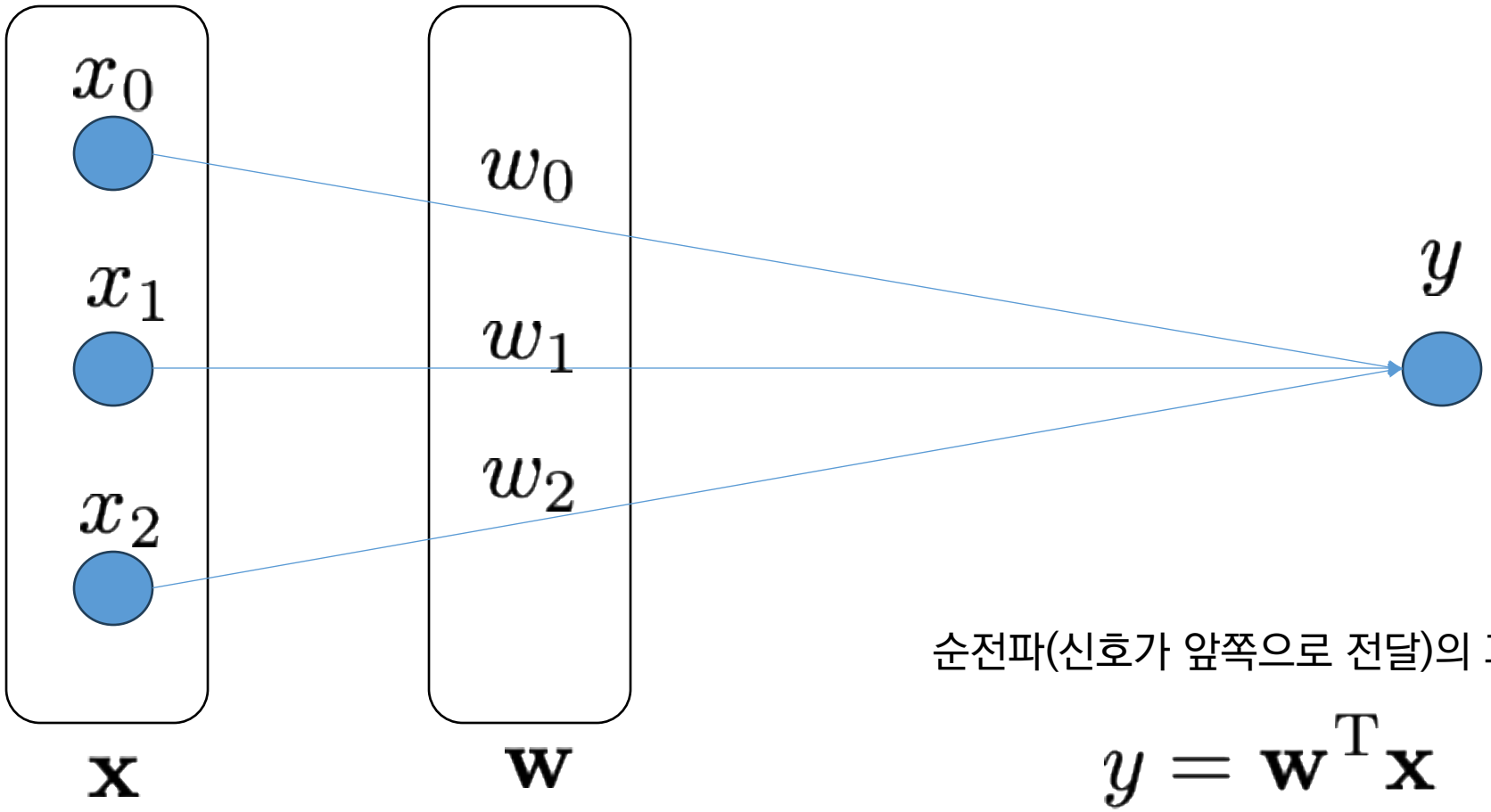
$$w \leftarrow w - \eta \cdot \partial E / \partial w$$

- 여기서 출력값 y와 목표값 t만 알면 연결강도 수정에 필요한 $\delta = (y - t)\phi'(wx + \cdots)$ 가 $\delta = (y - t) \cdot y \cdot (1 - y)$를 통해 쉽게 계산되며 δ를 이용하여 연결강도를 더 좋은 상태로 바꿀 수 있음

$$w \leftarrow w - \eta \cdot \delta \cdot x$$

- 신경망의 각 연결강도에 적용하여 오차를 줄일 수 있다면, 복잡한 신경망도 학습이 가능하지 않을까???
  - 이것이 퍼셉트론의 한계를 극복하는 돌파구가 될 수 있다
    - 오류 역전파 알고리즘의 등장

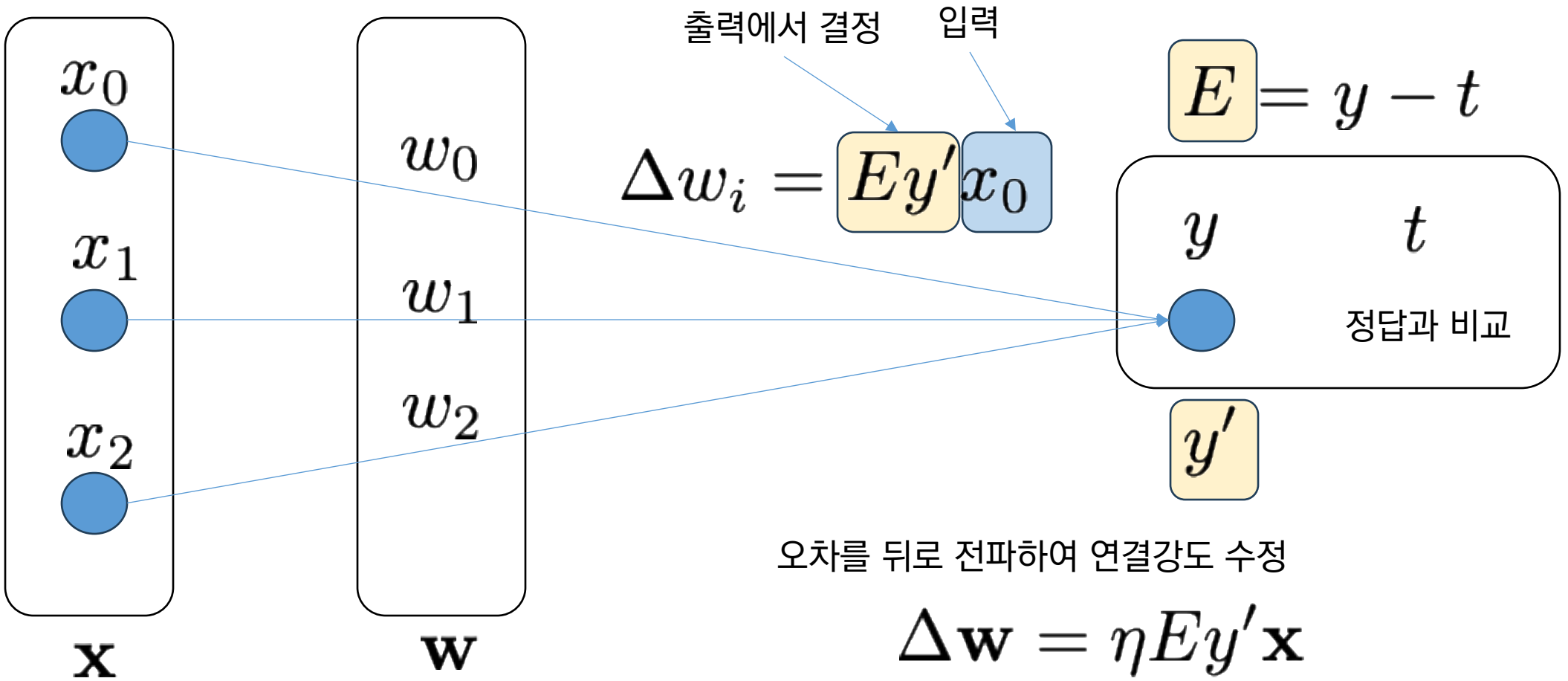
- **퍼셉트론의 동작을 벡터와 행렬의 곱으로 보자**



순전파(신호가 앞쪽으로 전달)의 과정

$$y = \mathbf{w}^{\mathrm{T}}\mathbf{x}$$

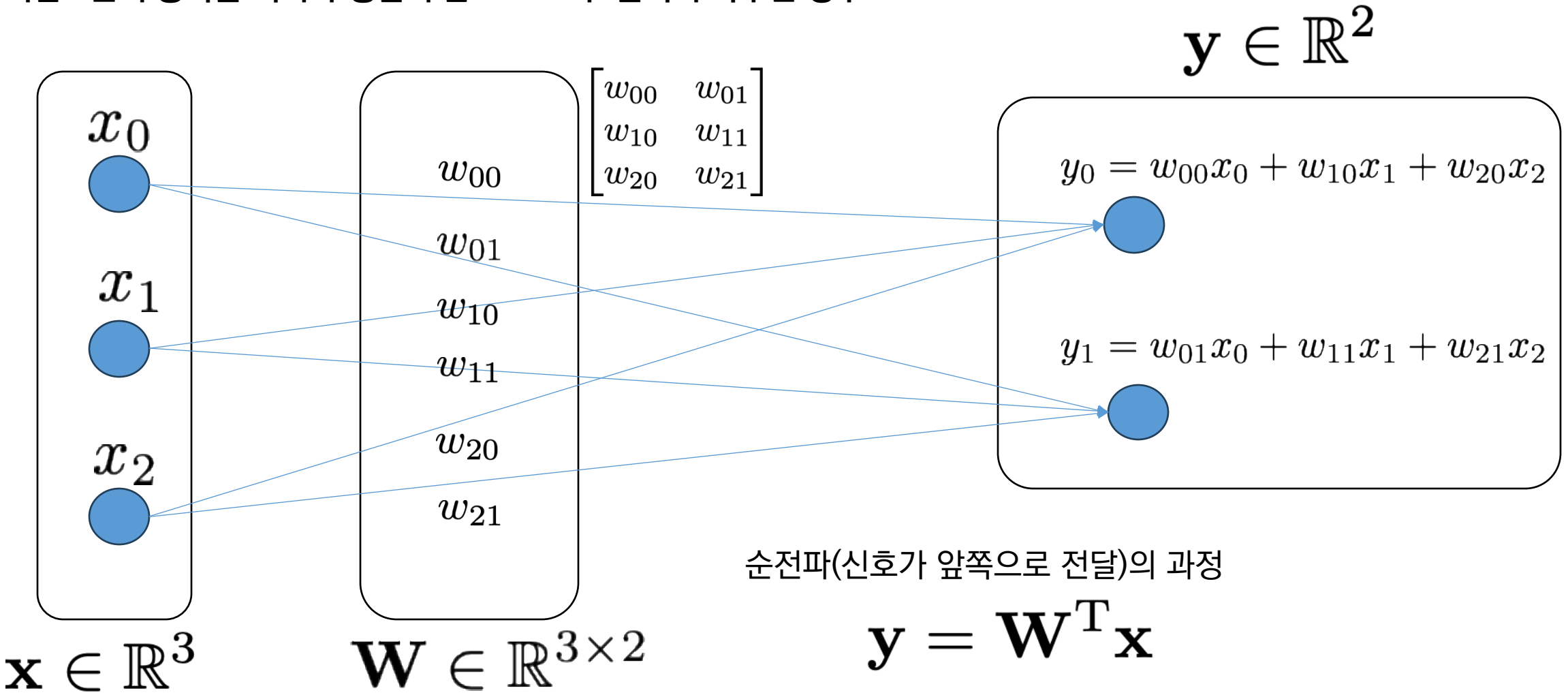
- **퍼셉트론의 동작을 벡터와 행렬의 곱으로 보자 : 오차 수정**



오차를 뒤로 전파하여 연결강도 수정

$$\Delta \mathbf{w} = \eta E y' \mathbf{x}$$

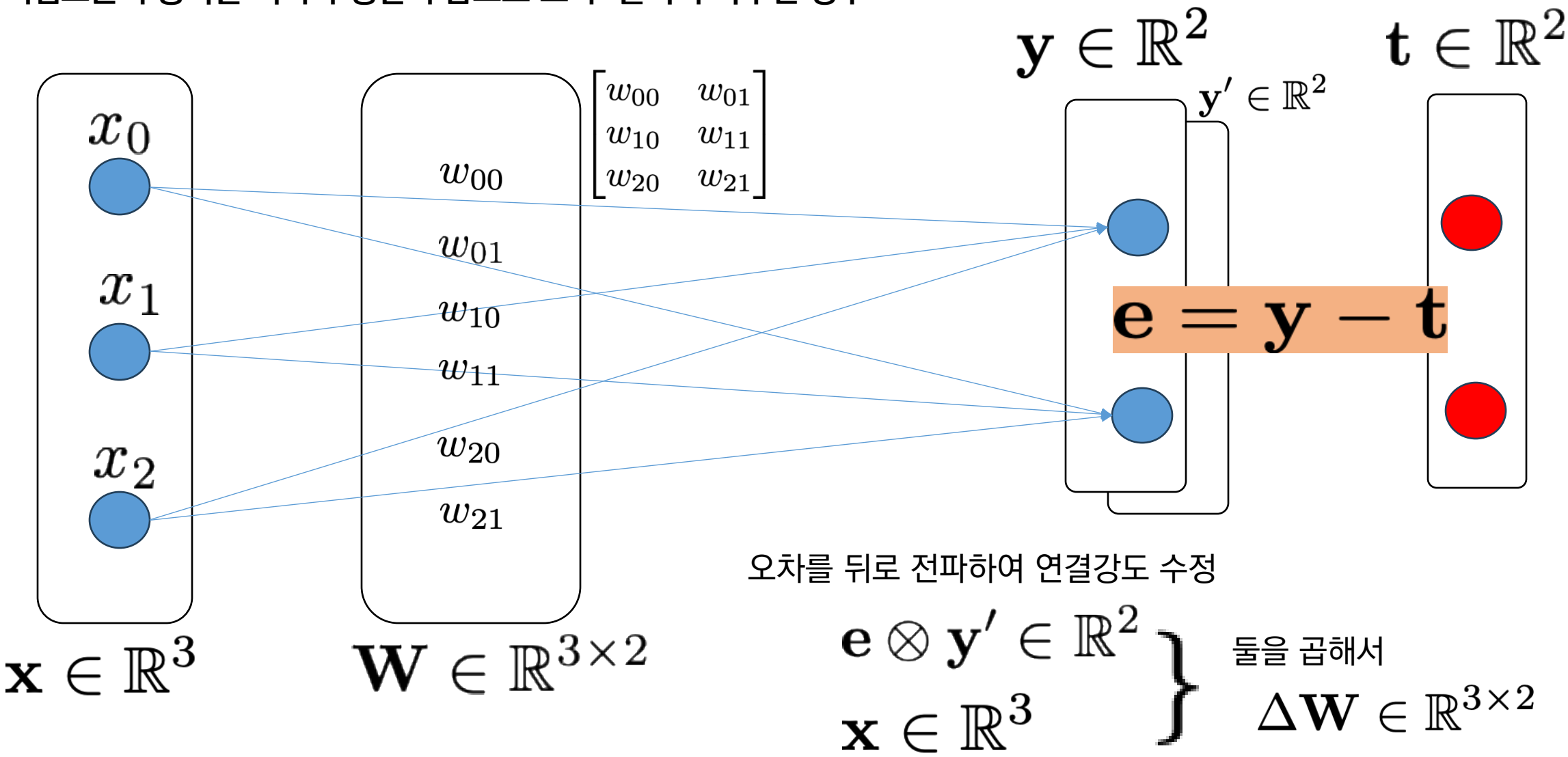
- 퍼셉트론의 동작을 벡터와 행렬의 곱으로 보자: 출력이 복수인 경우



순전파(신호가 앞쪽으로 전달)의 과정

$$\mathbf{y} = \mathbf{W}^{\mathrm{T}}\mathbf{x}$$

- 퍼셉트론의 동작을 벡터와 행렬의 곱으로 보자: 출력이 복수인 경우



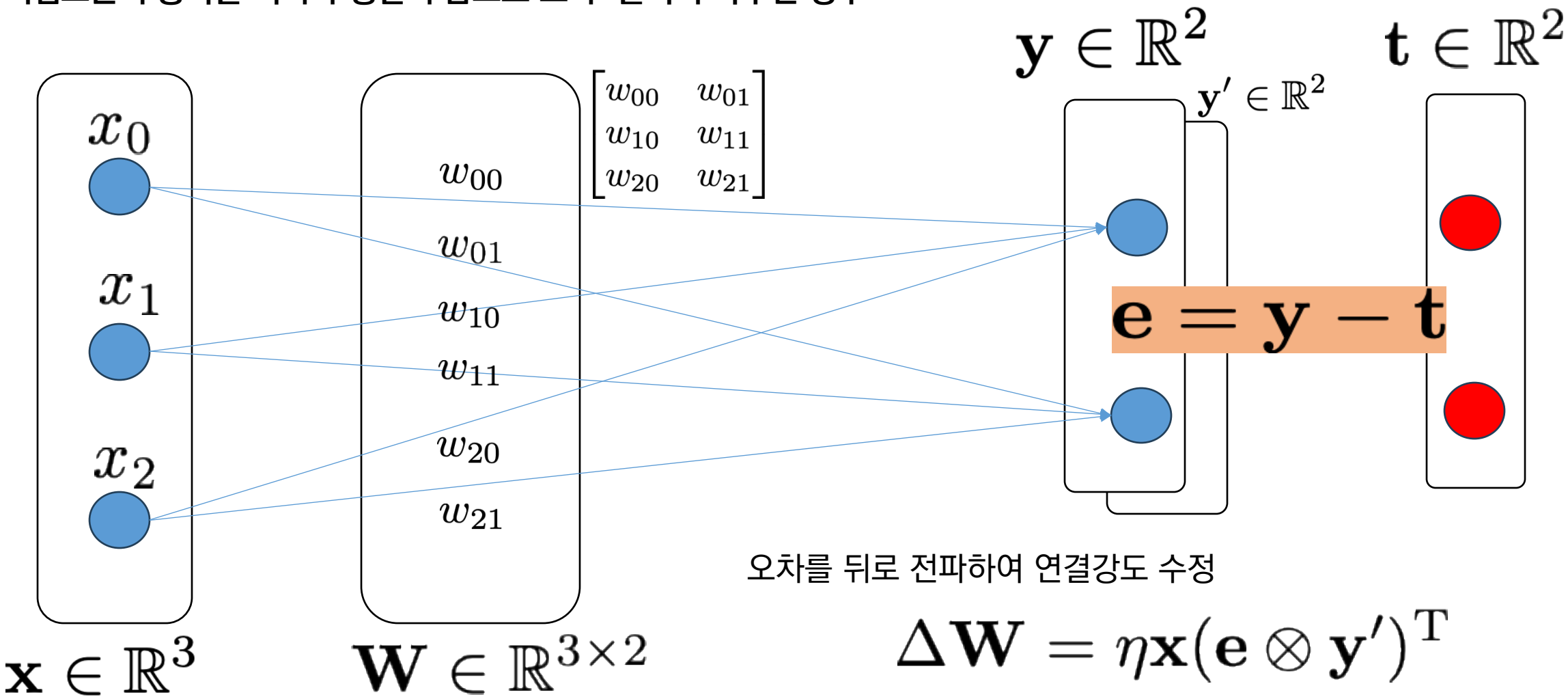
오차를 뒤로 전파하여 연결강도 수정

$\mathbf{e} \otimes \mathbf{y}' \in \mathbb{R}^2$

$\mathbf{x} \in \mathbb{R}^3$

} 둘을 곱해서 $\Delta\mathbf{W} \in \mathbb{R}^{3\times2}$

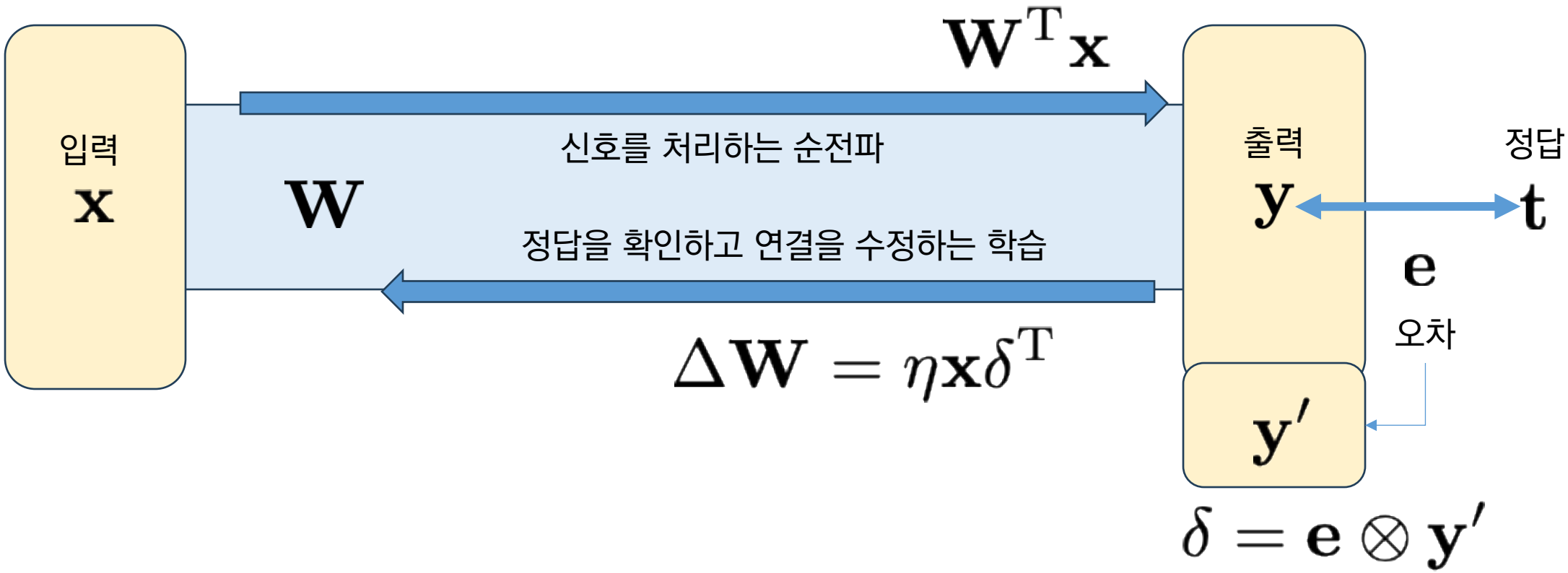
- 퍼셉트론의 동작을 벡터와 행렬의 곱으로 보자: 출력이 복수인 경우



오차를 뒤로 전파하여 연결강도 수정

$$\Delta \mathbf{W} = \eta \mathbf{x} (\mathbf{e} \otimes \mathbf{y}')^{\mathrm{T}}$$

- **벡터 / 행렬 계산으로 표현**


입력
$\mathbf{x}$
$\mathbf{W}^{\mathrm{T}}\mathbf{x}$
신호를 처리하는 순전파
$\mathbf{W}$
정답을 확인하고 연결을 수정하는 학습
$\Delta\mathbf{W} = \eta\mathbf{x}\delta^{\mathrm{T}}$
출력
$\mathbf{y}$
정답
$\mathbf{t}$
$\mathbf{e}$
오차
$\mathbf{y}'$
$\delta = \mathbf{e} \otimes \mathbf{y}'$

- 벡터 / 행렬 계산으로 표현

다항퍼셉트론은 벡터

```
[15] nX = 4
     nY = 1

     X_poly =
     W_poly =

     def PolyP
         globa
         X_pol
         return
```

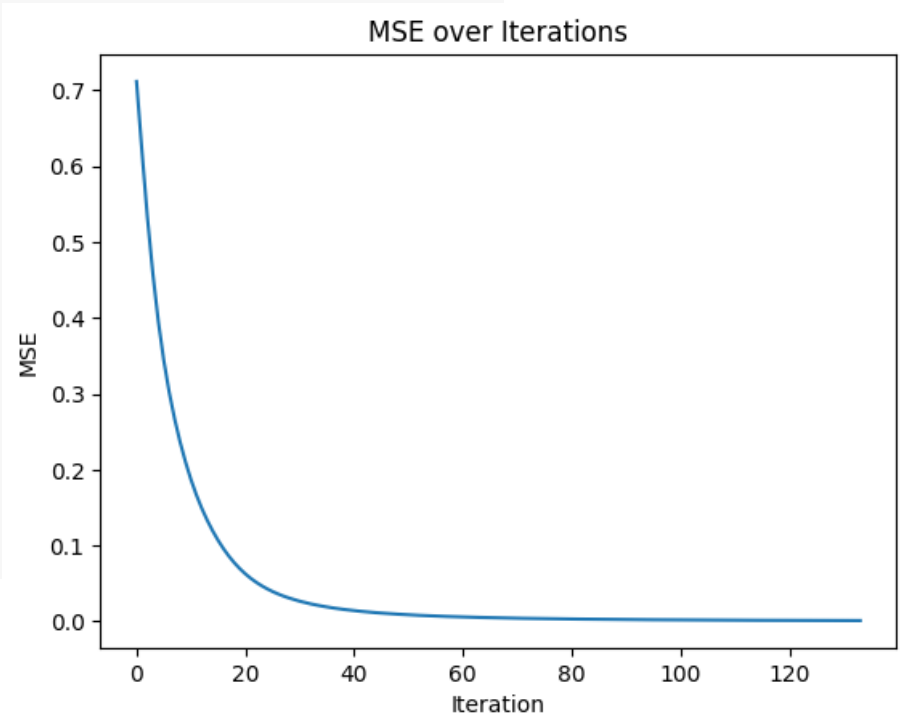
```
[25] W_poly = np.array([0.10, 0.20, 0.30, 0.40])      # 파라미터
     learning_rate = 1.5

     MSE_list = []
     for i in range(1000):
         MSE = 0.0
         MSE += Train(0, 0, 1)
         MSE += Train(0, 1, 0.5)
         MSE += Train(1, 0, 0.2)
         MSE += Train(1, 1, 1)
         MSE_list.append(MSE)
         if MSE < 0.001:
             print(f'{i}회 반복 후 종료')
             break
```

```
자
rget):


on(x0, x1)
1- y )


riv

y.dot(delta.T)
delta, E)
ng_rate * gradient
```

```
return E**2
```

- 벡터 / 행렬 계산으로 표현: 학습의 수행

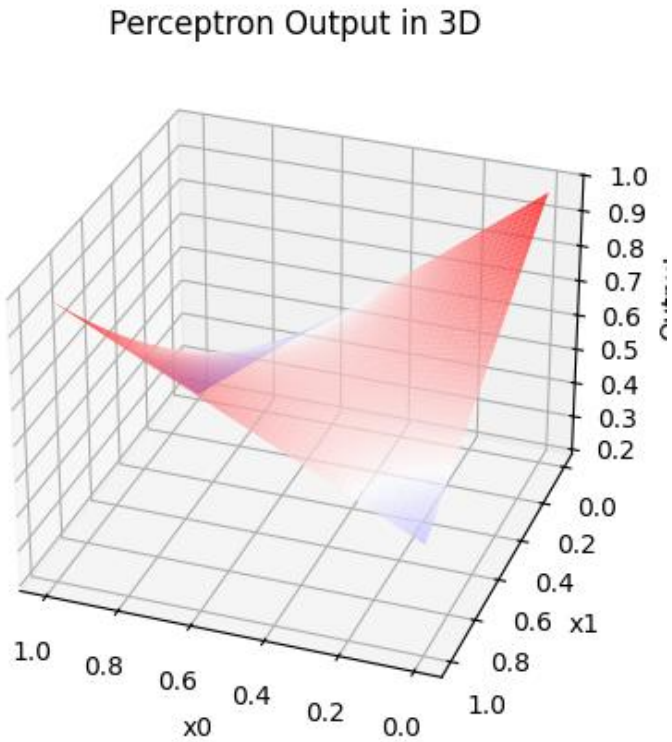
```
[25] W_poly = np.array([0.10, 0.20, 0.30, 0.40])      # 파라미터
     learning_rate = 1.5

     MSE_list = []
     for i in range(1000):
         MSE = 0.0
         MSE += Train(0, 0, 1)
         MSE += Train(0, 1, 0.5)
         MSE += Train(1, 0, 0.2)
         MSE += Train(1, 1, 1)
         MSE_list.append(MSE)
         if MSE < 0.001:
             print(f'{i}회 반복 후 종료')
             break
```



- **벡터 / 행렬 계산으로 표현: 학습의 수행**

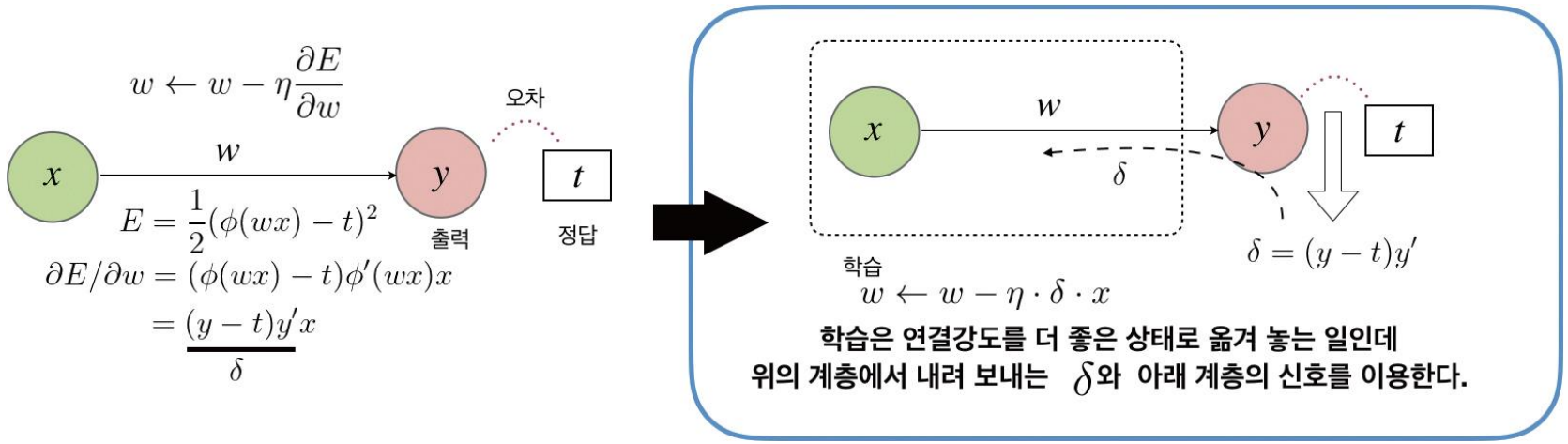
```
for x in [ [0, 0], [0, 1], [1, 0], [1, 1]]:
    print(x[0], x[1], PolyPerceptron(x[0], x[1]))

W_poly
```

```
0 0 0.9731936917032972
0 1 0.5005084094802026
1 0 0.20204740432990598
1 1 0.9857243710355242
array([-0.77114629, -0.47268528,  1.25636225,  0.97319369])
```

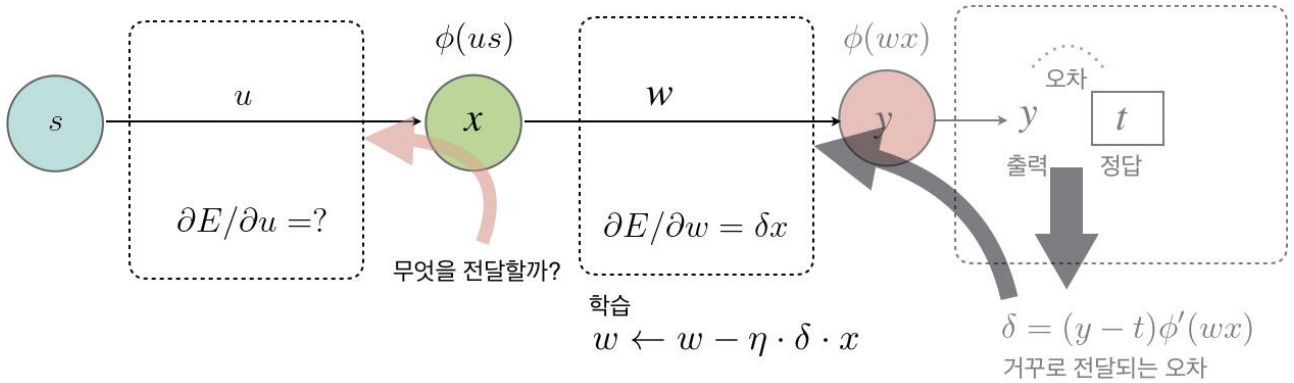


다층 퍼셉트론의 학습 – 오차를 아래로 전파하자

- 복잡한 문제를 해결하기 위해서는 일반적으로 신경망 연결을 여러 층으로 만들어야 하며 아래 그림과 같은 신경망이 연결된 구조에서 신호가 x를 거쳐 y로 전달되는데 이 방향을 **순전파**forward propagation라고 함
- 앞 절에서 오차를 줄이는 방향으로 연결강도를 변경하는 방법을 아래와 같이 출력 부분에서 확인할 수 있는 목표 t와출력 y의 차이, 그리고 출력의 미분 y'가 연결망을 거꾸로 타고 전달되어 가중치를 조정하는 것으로 봄



- s를 거쳐 x를 통과한 다음 출력노드 y로 연결되는 다층 신경망을 고려해보자.
- 연결강도 w를 갱신하는 방법을 그 이전 단계에 있는 연결강도 u 에도 전파할 수 있으며 이것이 오차 역전파error backpropagation의 기본 개념
  - 신경망 연구의 초창기에는 이 방법이 알려지지 않았기 때문에 민스키 등의 비판을 이겨낼 수 없었음



- u를 갱신하는 학습을 위해 우리는 오차 곡면의 기울기를 u기준으로 계산

$$\frac{\partial E}{\partial u} = \frac{\partial E}{\partial x} \cdot \frac{\partial x}{\partial u}$$

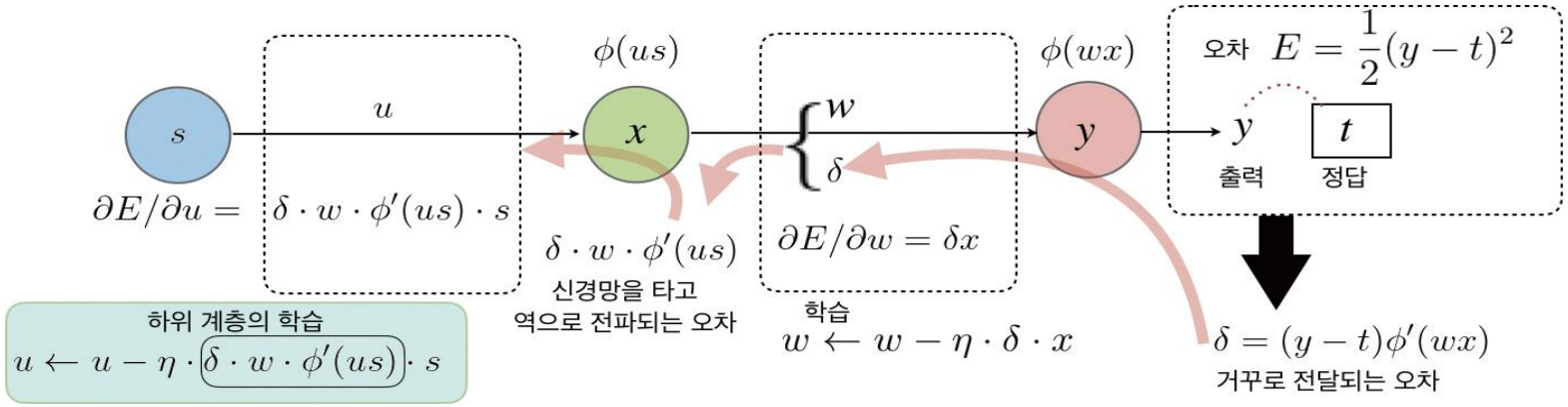
- $\partial E/\partial u$는 제곱 오차 $1/2\,(\phi(wx)-t)^2$를 w가 아니라 x로 미분한 것이므로 앞에서 구한 $\partial E/\partial w$식에서 w와 x의 역할을 바꿈

$$\partial E/\partial w = (y-t)y'x = \delta x$$
$$\partial E/\partial x = (y-t)y'w = \delta w$$

- 출력에 더 멀어진 단계의 가중치 u를 갱신하기 위한 오차의 미분을 계산하면 x값은 노드 s신호에 연결강도 u가 곱해진 뒤 활성화 함수를 통과한 결과인 $\phi(us)$이므로

$$\frac{\partial E}{\partial u} = \frac{\partial E}{\partial x}\frac{\partial x}{\partial u} = \delta \cdot w \cdot \frac{\partial \phi(us)}{\partial u} = \delta \cdot w \cdot \phi'(us) \cdot s$$

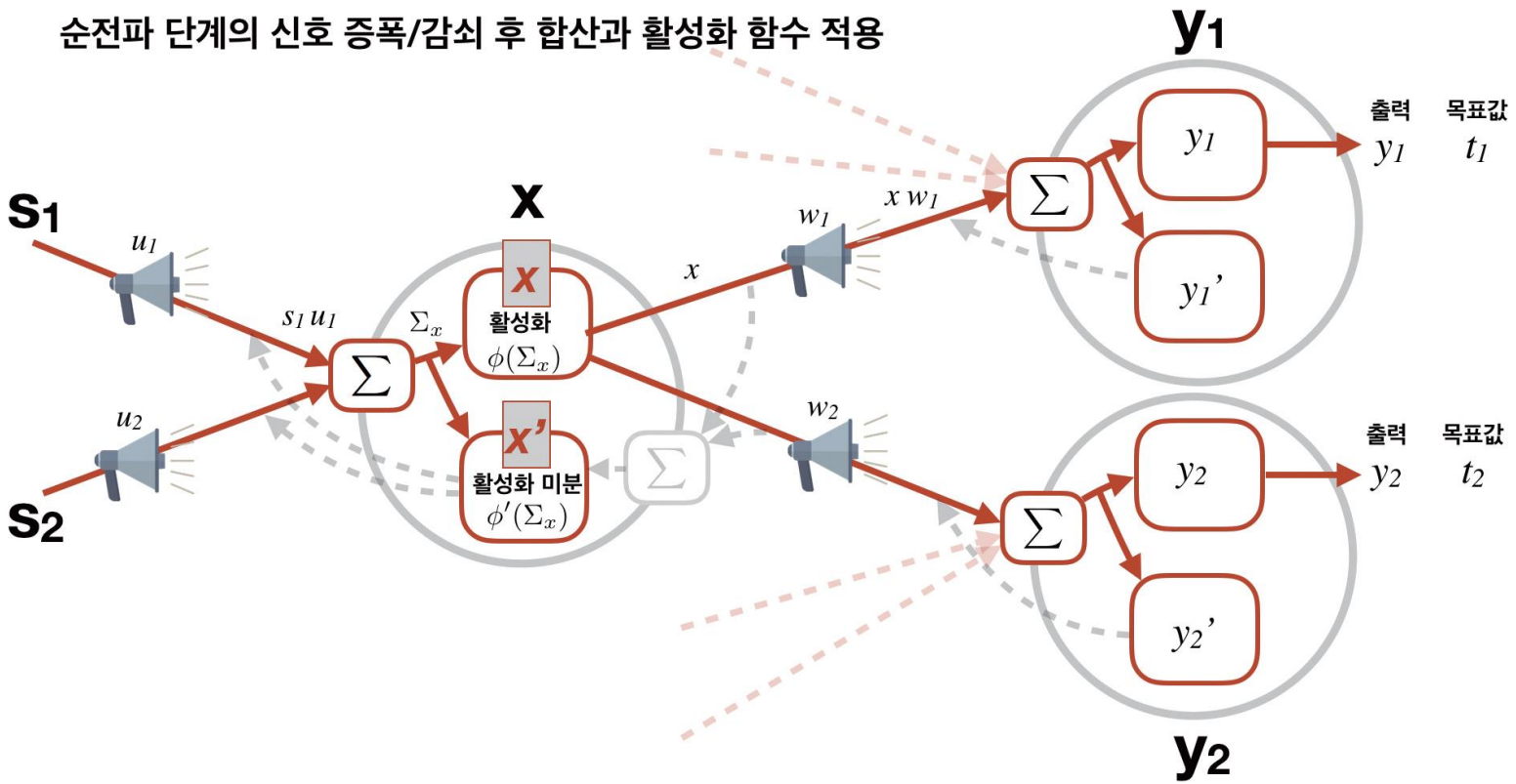
- 순전파forward propagation 단계에서는 입력의 신호가 연결강도로 증폭되고 활성화 함수를 통과하여 출력 노드로 전달되는 과정
- 학습 과정은 연결강도를 수정하는 것이기 때문에, 출력단의 오차가 역방향으로 활성화 함수의 미분 함수를 거친 뒤에 연결강도에 의해 증폭되어 입력단까지 전달되는 과정으로, 이러한 신경망 학습 방법을 오차 역전파error backpropagation 알고리즘



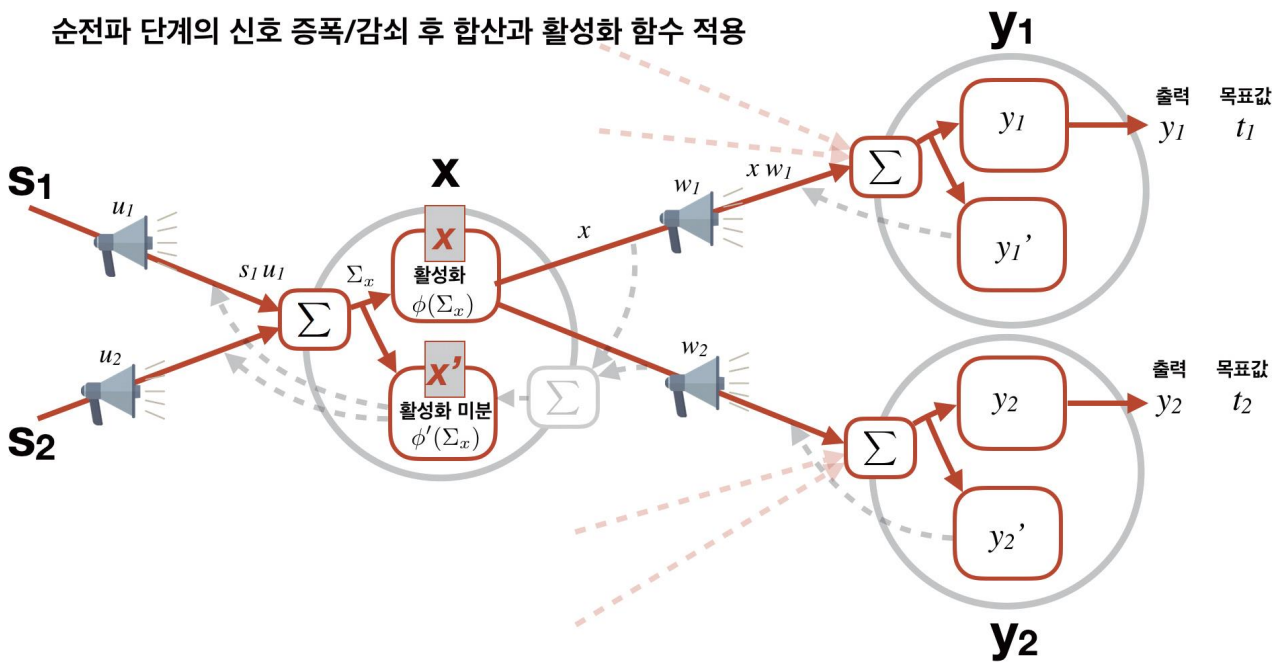
- 역전파 알고리즘은 1980년대 후반 발표
  - 데이비드 럼멜하트David Rumelhart, 제프리 힌튼Geoffrey Hinton, 로널드 윌리엄스Ronald Williams가 발표
- 이와 같은 아이디어는 폴 워보스Paul Werbos의 1974년도 박사학위 논문에서 제안됨
  - 순서가 있는 미분ordered derivative으로 제안된 것

7.8 역전파 알고리즘에 대한 직관적 이해

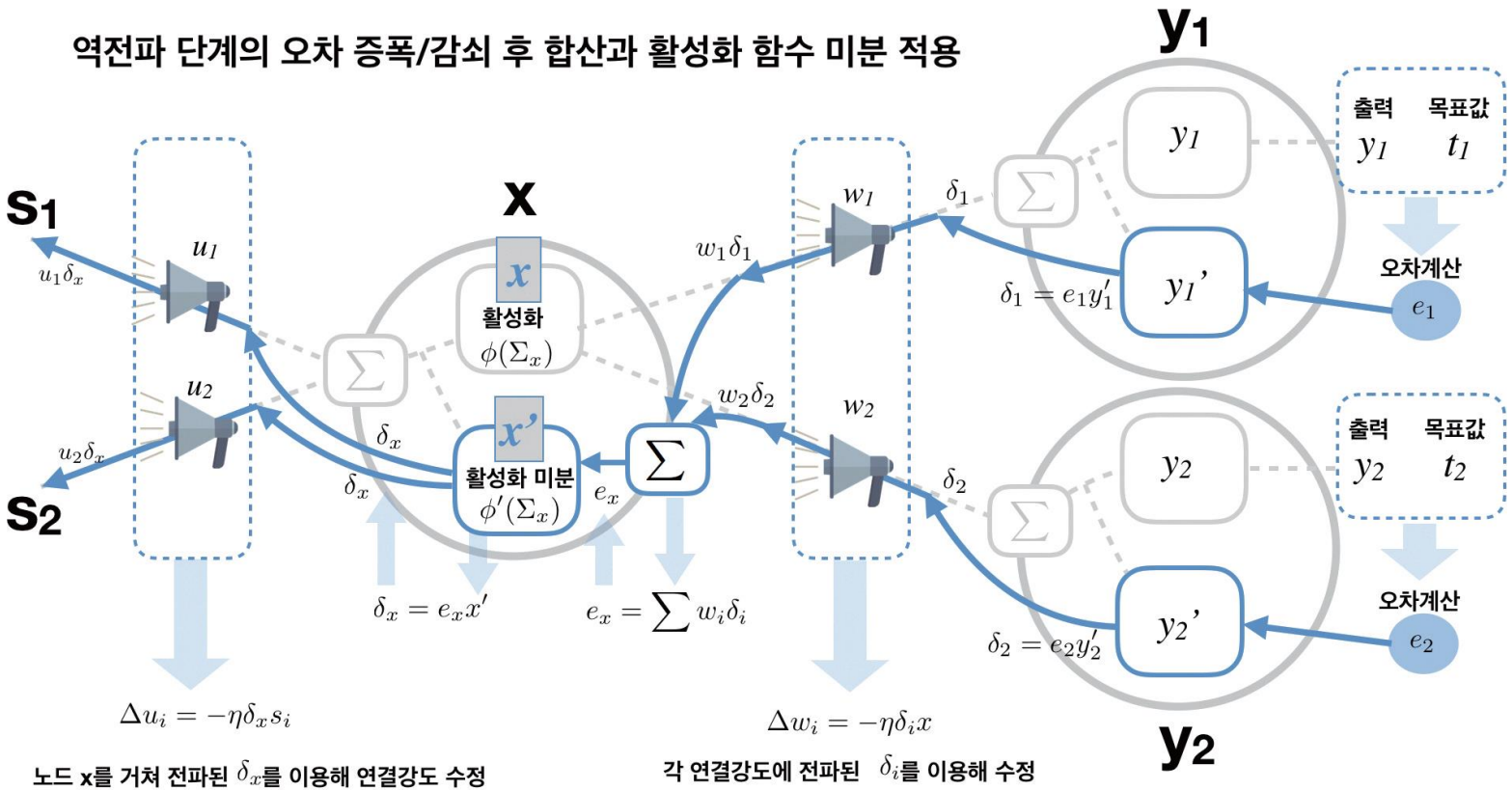
- 역전파 알고리즘을 이용한 연결강도 갱신 학습법을 이해하기 위해 아래와 같이 순전파 경로와 역전파 경로를 모두 표시한 연결망을 살펴보자



- 노드 x에는 두 개의 신호가 전달되며 s_1과 s_2를 연결강도 u_1과 u_2에 각각 곱한 뒤 모두 합한 것
  - 그림에서 Σ_x로 표시
  - 신경세포는 이렇게 모아진 신호가 임계치를 넘으면 출력을 하는 활성화 함수를 가짐
- 활성화 함수를 $\phi()$라 하면 노드 x는 $\phi(\Sigma_x)$를 계산하여 이 값을 x 로 출력하는 것
- 주의해서 볼 값은 x'
  - 순전파 과정에서는 다음 노드로 전달되지는 않는 값
  - 활성화 함수의 미분 $x' = \phi'(\Sigma_x)$값으로 나중에 사용하기 위해 미리 계산해 두는 값
  - 이러한 일은 그 다음 단계인 y_1과 y_2 노드처럼 신경망 전체에서 동일하게 이루어짐

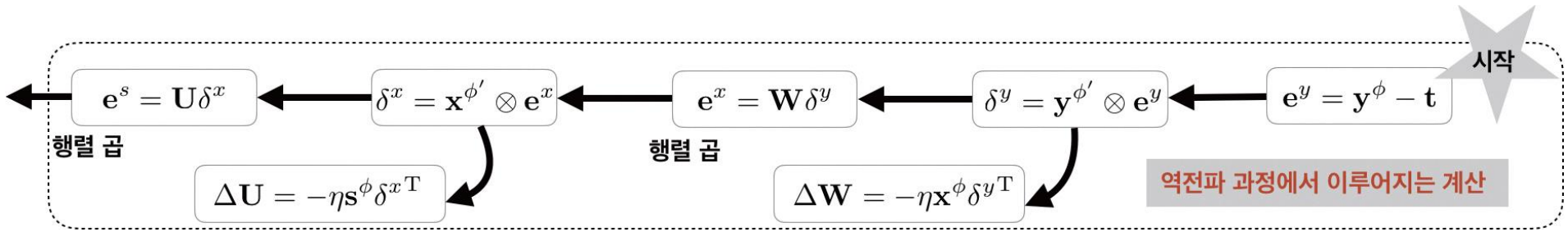


- 역전파 알고리즘은 순전파 과정에서 활성화 함수를 적용할 때마다 활성화 함수의 미분을 구해 두기만 한다면, 오차를 계산하여 순전파와 동일한 신호의 증폭과 합산으로 계산 다만 그 방향이 반대일 뿐



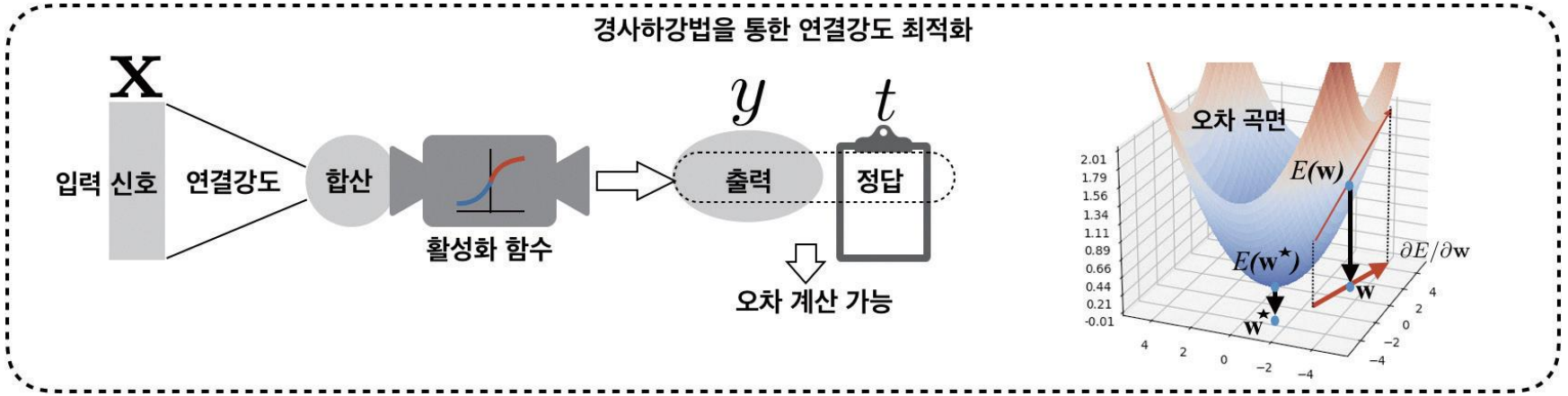
- 역전파 알고리즘을 적용하는 조건
    1. 오차를 계산하는 방법이 존재
    2. 활성화 함수는 미분이 가능한 함수
- 역전파 방법은 계속해서 아래 계층으로 내려갈 수 있음
    - 다층 퍼셉트론의 계층을 여러 층으로 쌓아도 모두 학습이 가능하다는 것을 의미
    - 인공 신경망 연구의 획기적인 발전을 이루는데 큰 기여

- 역전파 과정의 신호 전달을 행렬 연산으로 정리
  - 아래 그림의 오른쪽에서 왼쪽으로 진행하는 계산

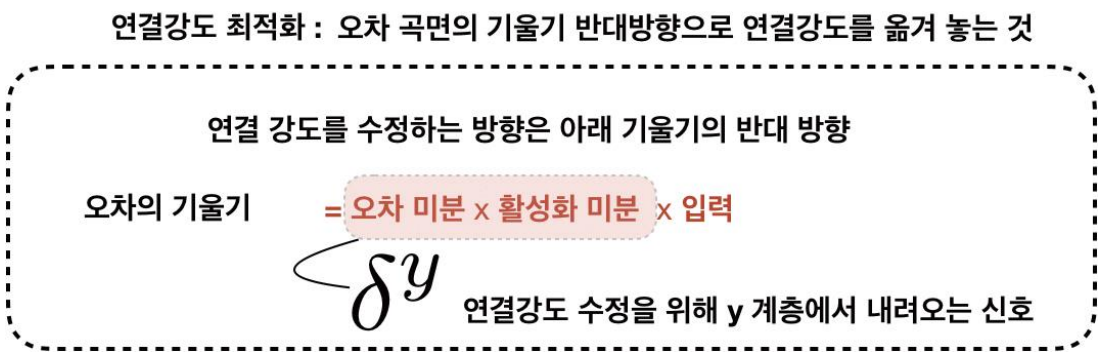


7.10 역전파 알고리즘의 요약

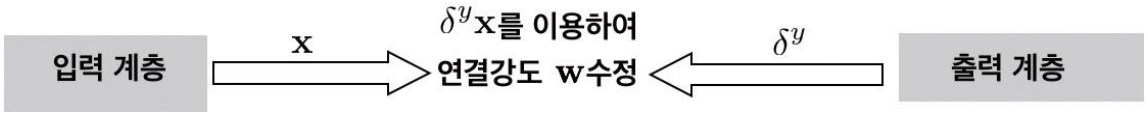
- 다층 퍼셉트론의 학습을 가능하게 한 신경망 분야의 중요한 알고리즘
- 기본은 물론 경사 하강법이며 그림처럼 입력 계층 x 가 연결강도를 통해 출력 계층으로 연결되어 연결강도에 의해 모아진 신호가 활성화 함수를 통과하여 y로 출력



- 연결강도 최적화는 연결강도 공간에서 오차의 기울기를 구해 그 반대 방향으로 파라미터를 움직여가는 것
- 기울기는 아래와 같이 오차 함수의 미분, 활성화 함수의 미분에 입력을 곱한 것으로 표현

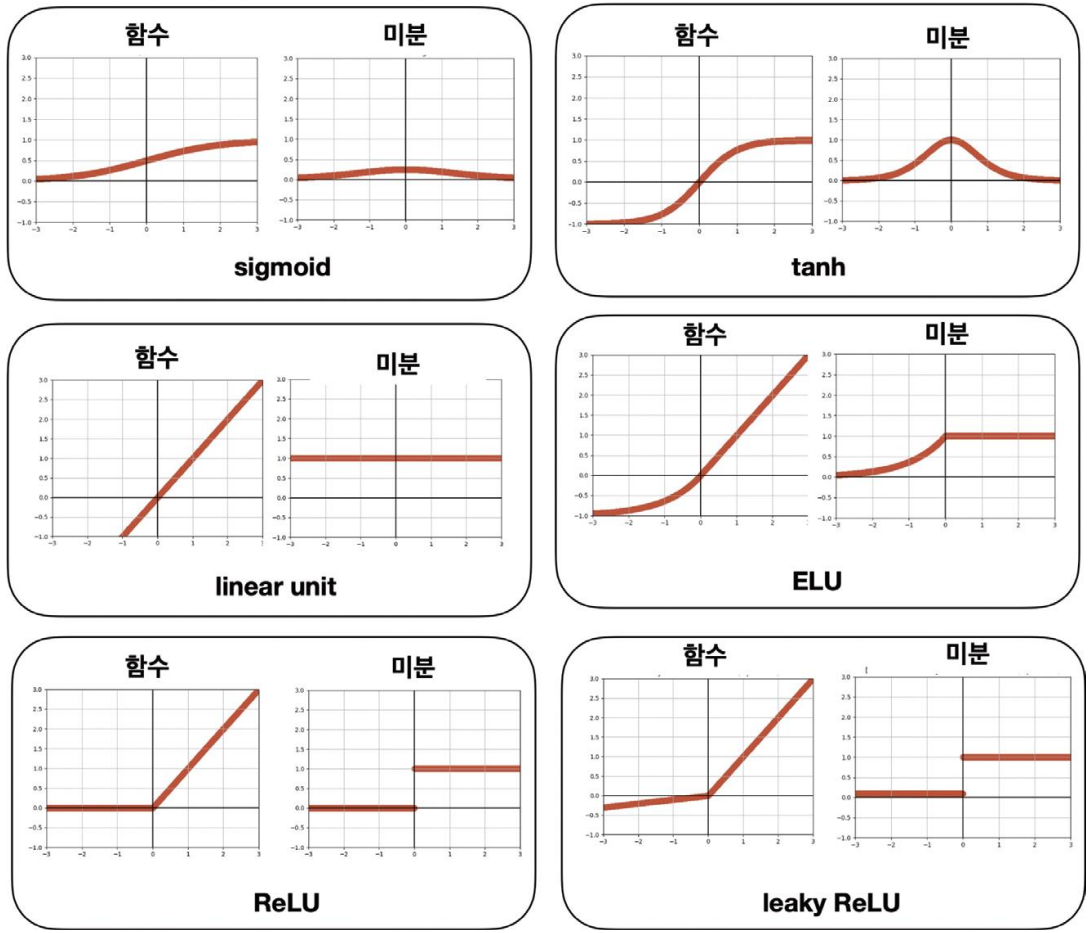


- 오차 기울기를 결정하는 오차 함수의 미분과, 활성화 함수의 미분은 출력 계층에서 알 수 있는 값이고, x 는 입력 계층의 값이므로 두 계층은 다음과 같은 정보를 이용해 협력하여 연결 강도를 수정



- 이러한 동작을 모든 계층에 일반화하면 y와 x 가 협력한 것과 똑같은 방식으로 x 계층과 그 아래에 있는 계층에도 협력하여 연결강도를 수정

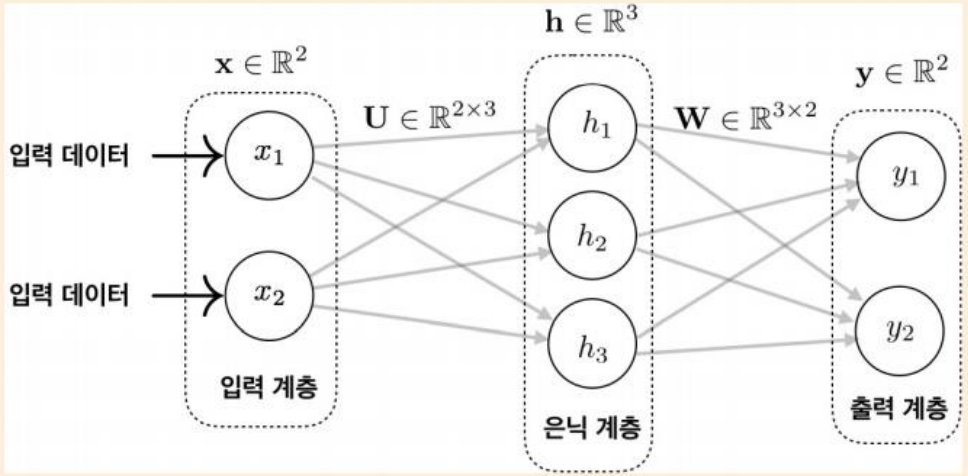
- 역전파 알고리즘은 오차 계산 함수와 활성화 함수만 결정되면 이들의 미분 함수들을 이용하여 계속해서 층별 δ 벡터를 구하는 일
- 시그모이드 함수를 활성화 함수로 사용해 보았는데, 그 외에도 다양한 활성화 함수를 도입할 수 있음



LAB 7-5 XOR 연산이 가능한 다층 퍼셉트론 만들기

실습 목표

역전파 모델로 아래와 같은 다층 퍼셉트론을 훈련시켜 XOR 연산을 수행하게 하라. 이 퍼셉트론의 출력은 y_1 이 켜지면 참, y_2 가 켜지면 거짓을 나타낸다.



힌트 앞에서 살펴본 바와 같이 각 계층으로 유입되는 오차를 미분치에 곱해서 델타 벡터를 만들고, 이를 이용하여 연결강도를 수정한다. 그리고 이 델타 벡터는 가중치를 타고 내려가게 하면 된다.

교재의 해답 코드를 하나하나 단계별로 수행해 봅니다

1. 넘파이와 맷플롯립을 임포트하여 수치 데이터를 다루고 가시화할 수 있게 한다.

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

2. 연결강도 U와 W를 난수를 발생시켜 준비한다. 실습 목표에 나타나 있는 퍼셉트론을 구현하기 위해서는 각각 (2,3)과 (3,2)의 크기를 갖는다. 학습률은 1.0으로 설정해 보겠다.

```
U = np.random.rand(2,3)    # 연결강도 U
W = np.random.rand(3,2)    # 연결강도 W
learning_rate = 1.0        # 학습률 η
```

3. 퍼셉트론이 사용할 활성화 함수로 시그모이드 함수를 구현하고, 이 함수의 미분함수도 구현한다.

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

def derivative_sigmoid(v):
    s = sigmoid(v)
    return s*(1-s)
```

4. 입력은 2차원 벡터인 input 변수, 은닉층 3개 노드들은 입력에서 넘어오는 신호를 합산하는 h_sum, 활성화 함수를 거친 h_out, 활성화 함수의 미분함수를 거친 h_deriv, 그리고 역전파 단계에서 넘겨받을 오차 h_error, 그리고 은닉계층이 아래로 내려보낼 델타 벡터 h_delta를 준비한다. 이것들은 모두 설계에 따라 3차원 벡터이다. 출력 계층은 2차원 벡터이고 동일한 준비를 한다.

```
input = np.zeros(2)

# 순전파시 계산될 값들 h^Σ, h^φ, h^φ'
h_sum, h_out, h_deriv = np.zeros(3), np.zeros(3), np.zeros(3)
# 역전파시 계산될 값들 e^h, δ^h
h_error, h_delta = np.zeros(3), np.zeros(3)

# 순전파시 계산될 값들 y^Σ, y^φ, y^φ'
yy_error, y_delta = np.zeros(2), np.zeros(2)
# 역전파시 계산될 값들 e^y, δ^y
y_error, y_delta = np.zeros(2), np.zeros(2)
```

5. 입력을 은닉층으로 전파하는 순전파 과정을 구현하자.

```
def forward_xh(x):
    global input, h_sum, h_out, h_deriv
    input = x
    h_sum = U.T.dot(input)                  # h^Σ = U^T x
    h_out = sigmoid(h_sum)                  # h^φ = φ(h^Σ)
    h_deriv = derivative_sigmoid(h_sum)     # h^φ' = φ'(h^Σ)
```

6. 은닉층에서 출력층으로 전파하는 순전파 과정도 비슷하게 구현된다. 다만 여기서는 내부에 있는 값을 이용하므로 입력을 따로 받지 않는다.

```
def forward_hy():
    global y_sum, y_out, y_deriv
    y_sum = W.T.dot(h_out)                  # y^Σ = W^T h^φ
    y_out = sigmoid(y_sum)                  # y^φ = φ(y^Σ)
    y_deriv = derivative_sigmoid(y_sum)     # y^φ' = φ'(y^Σ)
```

7. 출력을 목표치와 비교하여 오차를 구하는 것은 다음과 같다.

```
def compute_error(target):
    return y_out - target                   # y^φ − t
```

8. 이제 역전파 단계를 구현하자. 출력 계층의 동작은 출력과 목표값의 차이를 이용한다. 이 오차를 미분치에 곱하여 아래 계층으로 내려 보낼 델타 벡터를 계산한다. 그리고 이 델타 벡터와 이전 단계의 출력을 이용하여 연결강도를 수정한다.

```
def backward_y(error):
    global y_error, y_delta, W
    y_error = error                                          # e^y  error
    y_delta = y_deriv * y_error                              # δ^y = y^φ' ⊗ e^y
    ## 연결강도 W 수정
    dW = - learning_rate * np.outer(h_out, y_delta)          # ΔW = −η h^φ δ^yT
    W = W + dW                                               # W ← W + ΔW
```

9. 은닉 계층에서 이루어지는 역전파도 출력 계층과 크게 다르지 않다. 다만 사용하는 오차가 다를 뿐이다. 이 계층에서 사용하는 오차는 출력 계층에서 뒤로 넘겨준 델타 벡터가 두 계층을 연결하는 연결벡터 행렬에 곱해져 전달된다.

```
def backward_h():
   global h_error, h_delta, U
   h_error = W.dot(y_delta)                                  # e^h = W δ^y
   h_delta = h_deriv * h_error                               # δ^h = h^φ' ⊗ e^h
   ## 연결강도 U 수정
   dU = - learning_rate * np.outer(input, h_delta)           # ΔU = −η x δ^hT
   U = U + dU                                                # U ← U + ΔU
```

10. 퍼셉트론을 훈련 시키는 것은 입력과 목표를 제공해야 한다. 아래 **train**() 함수는 입력 데이터를 이용하여 순전파를 실행하고, 목표값과 비교하여 오차를 계산한다. 그리고 이 오차를 이용하여 역전파를 실시한다. 훈련을 할 때마다 계산된 오차의 제곱을 반환하게 하자.

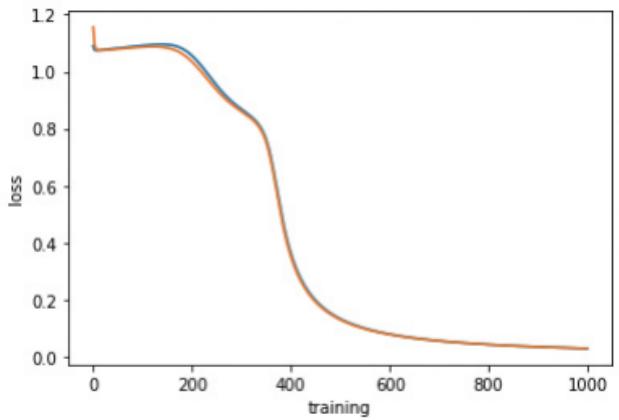
```
def train(x, target):
    forward_xh(x)
    forward_hy()
    e = compute_error(target)
    backward_y(e)
    backward_h()
    return e**2
```

11. 다음으로 **1000**번의 훈련을 실시하도록 하였다. 출력은 참일 경우에는 (**1**,**0**), 거짓일 경우에 (**0**,**1**)이 되도록 하였다. 하나의 노드만을 출력으로 사용하여 **0** 또는 **1**을 출력하는 방식도 있지만, 신경망에서는 서로 다른 범주에 속하는 결과를 구분할 때는 다수의 출력 노드를 사용해 각 범주마다 하나의 출력 노드가 활성화되게 하는 방법을 선호한다. 이 경우는 출력 노드가 두 개이다. 앞서 구현한 **train**() 함수를 이용하여 가능한 입력 **4**개에 대해 목표값을 주고 훈련하였다. 모든 데이터에 대한 훈련이 한 번 이루어질 때마다 제곱 오차를 누적한 결과를 **loss** 리스트에 추가한다.

```
loss = []

for i in range(1000):
    e_accum=0
    true = np.array([1,0])
    false = np.array([0,1])
    e_accum += train(np.array([0,0]), false)     # 훈련 데이터 1
    e_accum += train(np.array([0,1]), true)      # 훈련 데이터 2
    e_accum += train(np.array([1,1]), false)     # 훈련 데이터 4
    e_accum += train(np.array([1,0]), true)      # 훈련 데이터 3
    loss.append(e_accum)
```

12. 훈련 중에 계산된 오차를 화면에 그려보자. 독자 여러분이 실행할 경우 난수값에 의해 아래 그래프는 다소 차이가 날 수 있다.

```
plt.plot(loss)
plt.ylabel('loss')
plt.xlabel('training')
plt.show()
```



파란색과 주황색은 y_1과 y_2 출력 노드의 오차를 의미한다. 둘 모두 훈련이 지속됨에 따라 0 근처로 접근한다.

13. 테스트를 할 수 있는 함수를 만들어 보자. 입력 데이터의 리스트 X를 넘기면 각 인스턴스에 대해 순전파를 실시해 얻은 출력을 y_hat에 모아 리턴한다.

```
def test(X):
    y_hat = []
    for x in X:
        forward_xh(x)
        forward_hy()
        y_hat.append(y_out)
    return y_hat
```

14. 네 가지 가능한 모든 경우를 입력으로 제공해 결과를 확인하자. 아래 출력 결과에서 1에 가깝게 나타난 숫자를 붉게 표시하였다. 결과는 제대로 된 XOR 연산이 수행됨을 보인다.

```
test(np.array([[0, 0], [0, 1], [1, 0], [1, 1]]))
```

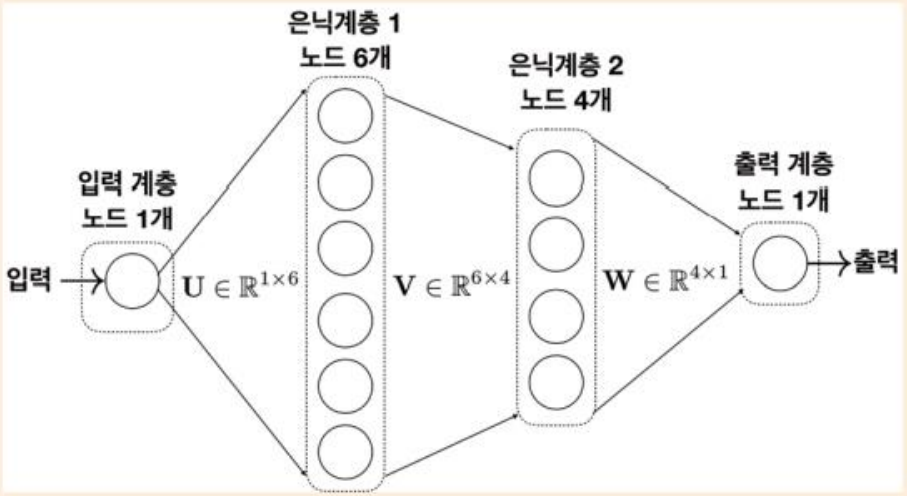
```
[array([0.08325217, 0.91679919]),
 array([0.94941716, 0.05054992]),
 array([0.94932782, 0.05063896]),
 array([0.01942864, 0.9805926 ])]
```

LAB$^{7-6}$ 다층 퍼셉트론으로 비선형 회귀 구현하기

실습 목표

퍼셉트론의 층을 증가시켜 아래와 같은 모델로 비선형 회귀를 구현해 보자. 적용할 데이터는 LAB$^{6-1}$에서 사용한 것과 같이 다음 URL에 있는 데이터이다.

https://github.com/dknife/ML/raw/main/data/nonlinear.csv

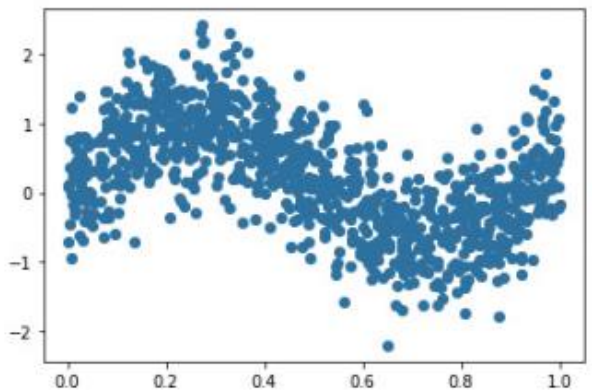


힌트 층을 늘이는 것은 순전파와 역전파 단계가 하나 늘어나는 것뿐이고, 계산은 동일한 방식으로 이루어진다.

교재의 해답 코드를 하나하나 단계별로 수행해 봅니다

1. 데이터를 읽어 들여서 확인해 보자. 이미 6장에서 사용해 본 데이터이다.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
data_loc = 'https://github.com/dknife/ML/raw/main/data/'
df = pd.read_csv(data_loc+'nonlinear.csv')
plt.scatter(df['x'], df['y'])
```



2. 각 계층의 노드 수는 입력, 은닉 계층 1, 은닉 계층 2, 출력 계층에서 각각 1개, 6개, 4개 1개이다. 이를 변수로 지정하고, 필요한 연결강도를 난수로 생성한다. 각 연결강도 행렬의 크기는 연결하는 두 계층의 노드 수로 크기가 결정된다. 층이 하나 더 늘어 연결강도는 U, V, W가 필요하다. 학습률은 0.1로 잡아 보았다.

```
nx, nh1, nh2, ny = 1, 6, 4, 1
U = np.random.randn(nx , nh1)*2
V = np.random.randn(nh1, nh2)*2
W = np.random.randn(nh2, ny)*2
learning_rate = 0.1
```

3. 활성화 함수는 여전히 시그모이드를 사용하자. 시그모이드의 미분 함수는 구현하지 않았는데, $\sigma'(x) = \sigma(x)(1-\sigma(x))$이기 때문에 간단히 시그모이드의 결과만 가지고 미분치를 구할 수 있기 때문이다.

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

4. 앞의 실습에서는 합산 벡터, 오차 벡터 등을 모두 구했는데, 효율적인 코딩을 위해 노드의 출력, 미분, 델타 벡터만 준비하겠다. 각 벡터의 크기는 해당 계층의 노드 개수로 정해진다.

```
input = np.zeros(nx)

h1_out, h1_deriv = np.zeros(nh1), np.zeros(nh1)   # 순전파시 계산 - 은닉계층 1
h1_delta = np.zeros(nh1)                          # 역전파시 계산

h2_out, h2_deriv = np.zeros(nh2), np.zeros(nh2)   # 순전파시 계산 - 은닉계층 2
h2_delta = np.zeros(nh2)                          # 역전파시 계산

y_out, y_deriv = np.zeros(ny), np.zeros(ny)       # 순전파시 계산 - 출력계층
y_delta = np.zeros(ny)                            # 역전파시 계산
```

5. 순전파를 단계별로 구하지 않고 시스템 전체로 구현하자.

```
def forward(x):
    global input, h1_out, h1_deriv, h2_out, h2_deriv, y_out, y_deriv
    input = x
    h1_out = sigmoid ( U.T.dot(input) )          # 은닉계층 1로 전파
    h1_deriv = h1_out * (1 - h1_out)             # 은닉계층 1의 미분

    h2_out = sigmoid ( V.T.dot(h1_out) )         # 은닉계층 2로 전파
    h2_deriv = h2_out * (1 - h2_out)             # 은닉계층 2의 미분

    y_out = sigmoid( W.T.dot(h2_out) )           # 출력계층으로 전파
    y_deriv = y_out * (1 - y_out)                # 출력계층의 미분
```

6. 오차는 출력과 목표값의 차이로 계산한다.

```
def compute_error(target):
    return y_out - target
```

7. 역전파 과정도 오차를 입력하면 출력 계층까지 한 번에 이루어지게 구현하자.

```
def backward(error):
    global y_delta, W, h2_delta, V, h1_delta, U

    y_delta = y_deriv * error                           # 출력 계층의 델타
    dW = - learning_rate * np.outer(h2_out, y_delta)    # W의 수정

    W = W + dW
    h2_delta = h2_deriv * W.dot(y_delta)                # 은닉 계층 2의 델타
    dV = - learning_rate * np.outer(h1_out, h2_delta)   # V의 수정

    V = V + dV
    h1_delta = h1_deriv * V.dot(h2_delta)               # 은닉 계층 1의 델타
    dU = - learning_rate * np.outer(input, h1_delta)    # U의 수정
```

8. 훈련은 순전파를 실시하고, 오차를 구해 역전파를 실시한다. 매번 제곱 오차도 반환한다.

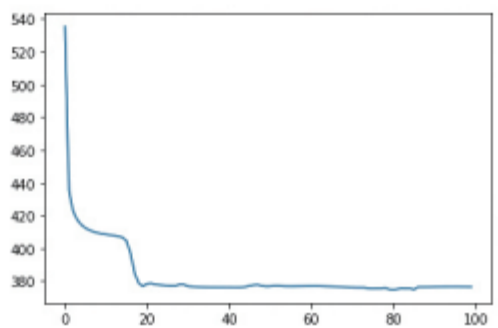
```
def train(x, target):
    forward(x)
    e = compute_error(target)
    backward(e)
    return e**2
```

9. 읽어 들인 데이터에 'x'라는 이름의 열을 입력 데이터로 사용하고, 'y'라는 이름의 열은 **레이블**label로 사용하자. 이 두 데이터를 넘파이 배열로 만들어 차례로 훈련 함수에 넘긴다. 매 훈련 세트에 대한 훈련마다 오차를 누적해 loss 리스트에 기록한다.

```
loss = []
X = df['x'].to_numpy()
y_label = df['y'].to_numpy()
for i in range(100):
    e_accum = 0
    for x, y in zip(X, y_label):
        e_accum += train(x, y)
    loss.append(e_accum)
```

10. 오차를 그려 학습이 되는지 확인해 보자. 학습이 진행될 수록 오차값도 작아진다.

```
err_log = np.array(loss).flatten()
plt.plot(err_log)
plt.show()
```

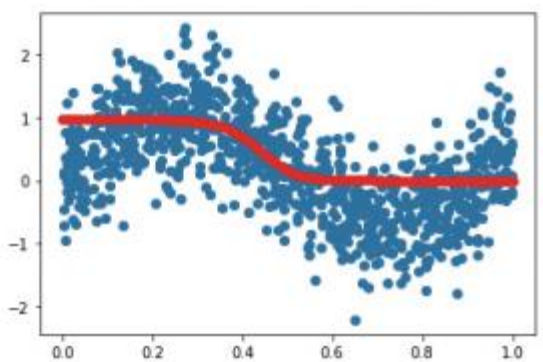


11. 입력 데이터를 받아 예측을 실시하는 **predict()** 함수를 만들어 보자. 여러 개의 입력이 리스트로 들어오면 하나씩 가져와 순전파를 실시하고, 출력을 저장한다.

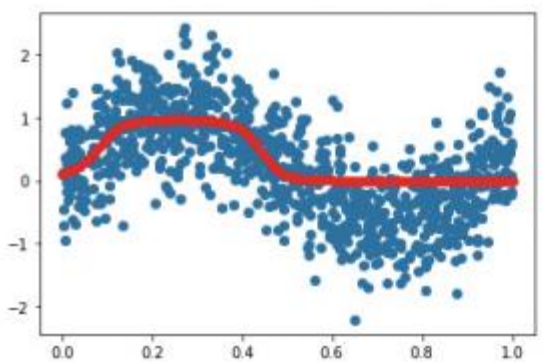
```
def predict(X):
    y_hat = []
    for x in X:
        forward(x)
        y_hat.append(y_out)
    return y_hat
```

12. 이제 **0**에서 **1**까지의 데이터를 생성해 이 모델에 입력으로 제공하여 얻는 예측치를 화면에 그려보자. 원래의 데이터와 예측 값을 비교하는 코드이다.

```
domain = np.linspace(0, 1, 100).reshape(-1,1)      # 입력은 2차원 벡터로 변형
y_hat = predict(domain)
plt.scatter(df['x'], df['y'])
plt.scatter(domain, y_hat, color='r')
```



50번의 역전파



300번의 역전파

민스키의 재등판

잠깐 – 역전파 알고리즘은 신경망의 부활을 가져올까?

1980년대에 신경망 분야는 역전파 알고리즘으로 활기를 되찾게 된다. 그런데 민스키는 역전파 알고리즘에 대해서도 퍼셉트론의 2판을 써서 다음과 같은 비판적인 의견을 내었다.

> *"많은 연결주의자들은 역전파가 단순히 기울기를 계산하는 특별한 방법에 불과하다는 것을 이해하지 못하고, 이것이 **언덕 오르기**hill-climbing 알고리즘의 기본적 한계를 극복하는 새로운 학습 기법이라 여기는 것 같다."* [10]

민스키의 지적대로 역전파가 마법은 아니지만, 신경망으로 해결할 수 있는 문제는 크게 확장되었다. 그런데, 신경망 분야는 금세 다시 침체에 빠지게 된다. 역전파가 무력해지는 경우가 많기 때문이다. 이 문제의 이유와 극복 과정이 앞으로 살펴볼 내용이다.

---

10) Minsky, M. and Papert, S. (1988) Perceptrons: An Introduction to Computational Geometry 2nd Ed., MIT Press, 260–61.

7.11 신경망을 설계하고 훈련을 실시할 수 있는 도구 : 텐서플로우

- 오차 역전파를 통한 신경망 학습의 방법을 살펴보고 직접 구현하면서 여러 층으로 이루어진 신경세포 층들 사이의 신호 전달이 동일한 방식으로 차원만 바꾸어 진행되는 행렬-벡터 연산임을 확인
- 신경망 구현은 텐서플로우TensorFlow라는 도구를 사용할 것이며 텐서플로우는 구글에서 만든 것으로 지금까지 실습을 진행하기 위해 사용한 구글 코래버러토리에 설치가 되어있어 준비없이 바로 사용할 수 있음
- 텐서플로우는 케라스keras라는 심층신경망 구현을 위한 API가 포함되어 있는데, 이를 이용하면 간단히 신경망을 만들고 학습 가능

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# 텐서플로우와 케라스를 사용할 수 있도록 준비
import tensorflow as tf     # 텐서플로우는 주로 tf라는 별명을 사용한다
from tensorflow import keras
```

- 모델은 신경 계층이 차례로 이어져 순차적으로 신호를 전달하는 모델
- 이런 모델은 케라스에서 제공하는 시퀀셜Sequential 클래스로 구현
- LAB^{7-6}의 회귀 모델은 입력 계층을 제외하면 6개, 4개, 1개의 노드를 가진 층layer들이 쌓여 있으며 이 모든 층들은 이전 단계의 층이 가지는 신경세포 노드와 빠짐없이 다 연결되어 있음
  - 이러한 연결을 밀집dense 계층
  - 각 층이 가진 활성화 함수는 시그모이드

```
model = keras.models.Sequential( [
    keras.layers.Dense(6, activation= 'sigmoid'),
    keras.layers.Dense(4, activation= 'sigmoid'),
    keras.layers.Dense(1, activation= 'sigmoid'),
])
```

- 전체 데이터를 다 사용하지 않고 임의로 선택된 데이터 인스턴스만을 가지고 경사 하강법을 적용하는 확률적 경사 하강법stochastic gradient descent:SGD을 사용
- 확률적 경사 하강법은 전체 데이터를 이용하여 학습하는 경사 하강법과는 다르게 임의적으로 추출한 일부 데이터를 사용해서 가중치를 조절 하기에 가중치 조절의 속도가 개선되는 효과
- 모델을 사용하기 위해서는 컴파일compile 과정
  - 컴파일 과정에서는 최적화 방법을 지정하고, 오차 측정을 어떤 기준으로 할 것인지 지정
  - 평균 제곱 오차를 손실함수로 사용한다면 'mse'를 손실 매개변수 loss의 인자로 지정

```
optimizer = keras.optimizers.SGD(learning_rate=5.0)
model.compile(optimizer=optimizer, loss='mse')
```

- 모델을 훈련시킬 데이터는 LAB^{7-6}에서 사용한 것과 동일

```
data_loc = 'https://github.com/dknife/ML/raw/main/data/'
df = pd.read_csv(data_loc+'nonlinear.csv')
X = df['x'].to_numpy()
y_label = df['y'].to_numpy()
```

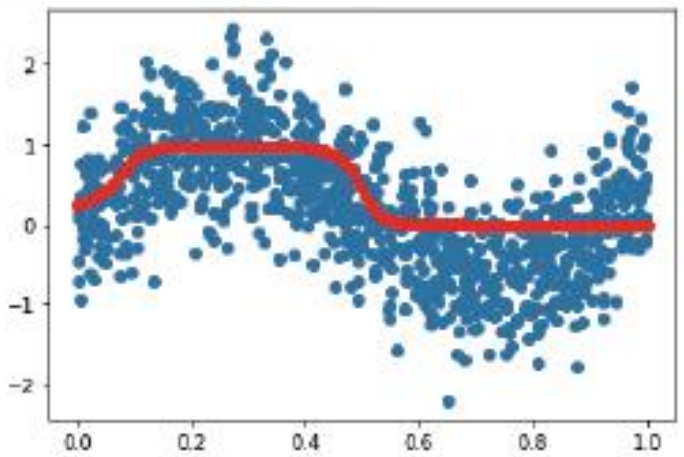
- 모델의 훈련은 입력과 레이블을 fit() 메서드
- 에폭epoch을 지정하는데, 에폭은 하나의 데이터셋에 대해 몇 번 훈련을 반복할 것인지를 의미

```
model.fit(X, y_label, epochs=100)
```

```
Epoch 1/100
32/32 [==============================] - 1s 1ms/step - loss: 0.6938
...
Epoch 100/100
32/32 [==============================] - 0s 1ms/step - loss: 0.3652
```

- LAB^{7-6} 에서 사용한 테스트용 입력을 모델에 넣어서 예측
- 케라스 모델에서 예측을 수행하는 방법은 predict() 메서드에 입력 데이터를 넘김

```
domain = np.linspace(0, 1, 100).reshape(-1,1) # 입력은 2차원 벡터로 변형
y_hat = model.predict(domain)
plt.scatter(df['x'], df['y'])
plt.scatter(domain, y_hat, color='r')
```



핵심 정리

- 맥컬록과 피츠는 1940년대 초 선구적인 연구를 통해 신경세포의 신호전달 모델을 이용하여 계산을 할 수 있음을 보였다.
- 맥컬록과 피츠의 신경망 모델은 학습이 불가능하다는 단점이 존재한다.
- 로젠블랫은 퍼셉트론 모델을 통해 신경세포 모델을 이용하여 스스로 학습하여 문제를 해결할 수 있는 모델을 1950년대에 제안하였다. 이 때의 학습 방법은 헵의 학습 규칙에 근거한 것이었다.
- 헵의 학습 규칙은 "같이 활성화 되는 신경세포는 함께 연결된다"로 요약할 수 있다.
- 로젠블랫의 퍼셉트론 모델은 당시 큰 관심을 모았고 많은 연구자들이 퍼셉트론과 같은 방식의 인공 신경 망 연구에 참여하였다.
- 기호주의 인공지능 연구자들은 퍼셉트론에 대한 과열된 관심이 언론을 편파적으로 활용한 비과학적인 주장 때문이라고 보았다.
- 기호주의 인공지능 분야의 대가 민스키는 패퍼트와 함께 퍼셉트론이라는 제목의 책을 저술하여 퍼셉트론이 가진 한계를 보였다.
- 민스키의 비판 가운데 가장 잘 알려진 것은 퍼셉트론이 XOR와 같은 단순한 문제도 풀 수 없는 한계를 가 진다는 것이었다.

핵심 정리

- XOR 문제는 당시 퍼셉트론이 가진 구조적 단순성으로는 극복할 수 없었다.
- 복잡한 구조의 퍼셉트론으로 민스키가 제기한 문제를 해결할 수 있지만, 당시에는 복잡한 구조의 신경망을 학습시킬 방법이 없었다.
- 1980년대에 럼멜하트, 힌튼, 윌리엄스가 오차 역전파 알고리즘을 개발함으로써 여러 층으로 이루어진 퍼셉트론의 학습이 가능하게 되었다.
- 역전파 알고리즘은 1974년 워보스의 박사학위 논문에서 같은 개념이 제안되었다.
- 역전파 알고리즘은 인공 신경망 분야의 새로운 활기를 가져왔다.
- 역전파 알고리즘은 순전파와 마찬가지로 계층들 사이의 신호 전달을 하는 것이며, 이 과정은 행렬과 벡터 의 연산으로 표현할 수 있다.
- 텐서플로우와 같은 도구를 이용하면 다층 구조의 신경망을 쉽게 정의하고 사용할 수 있다.
- 경사 하강법의 속도를 개선하는 방법으로 학습시 전체 데이터를 사용하지 않고 랜덤하게 추출한 데이터를 사용하는 확률적 경사 하강법이 있다.