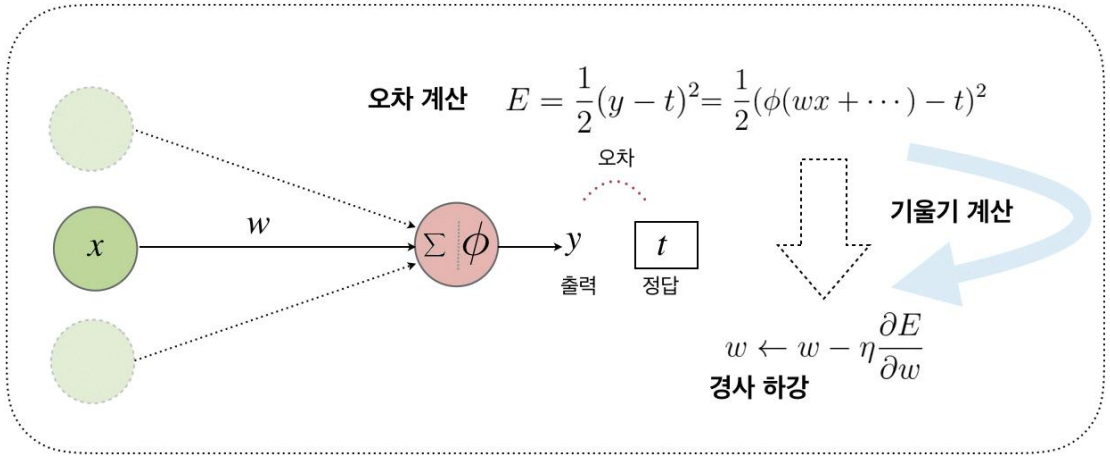
인공 신경망 기초

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

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

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

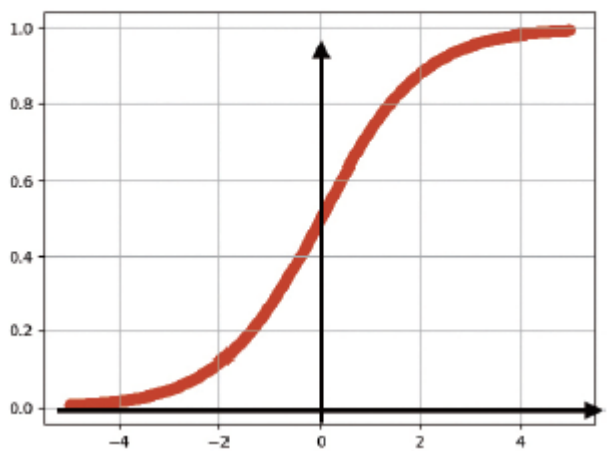
$$= (\phi(wx+\cdots)-t)\cdot \phi'(wx+\cdots)\cdot \frac{\partial}{\partial w}(wx+\cdots)$$

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

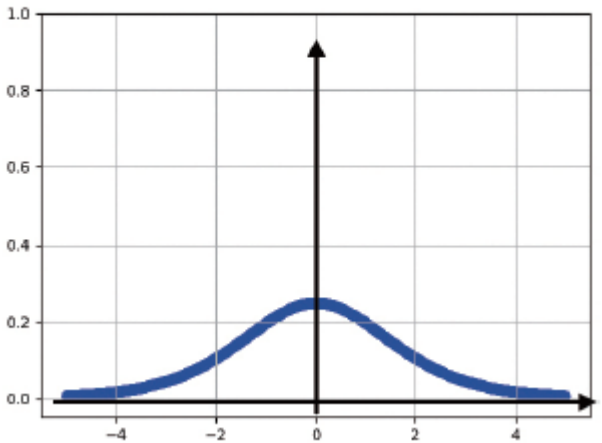
계단 함수는 미분이 불가능

- 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
```

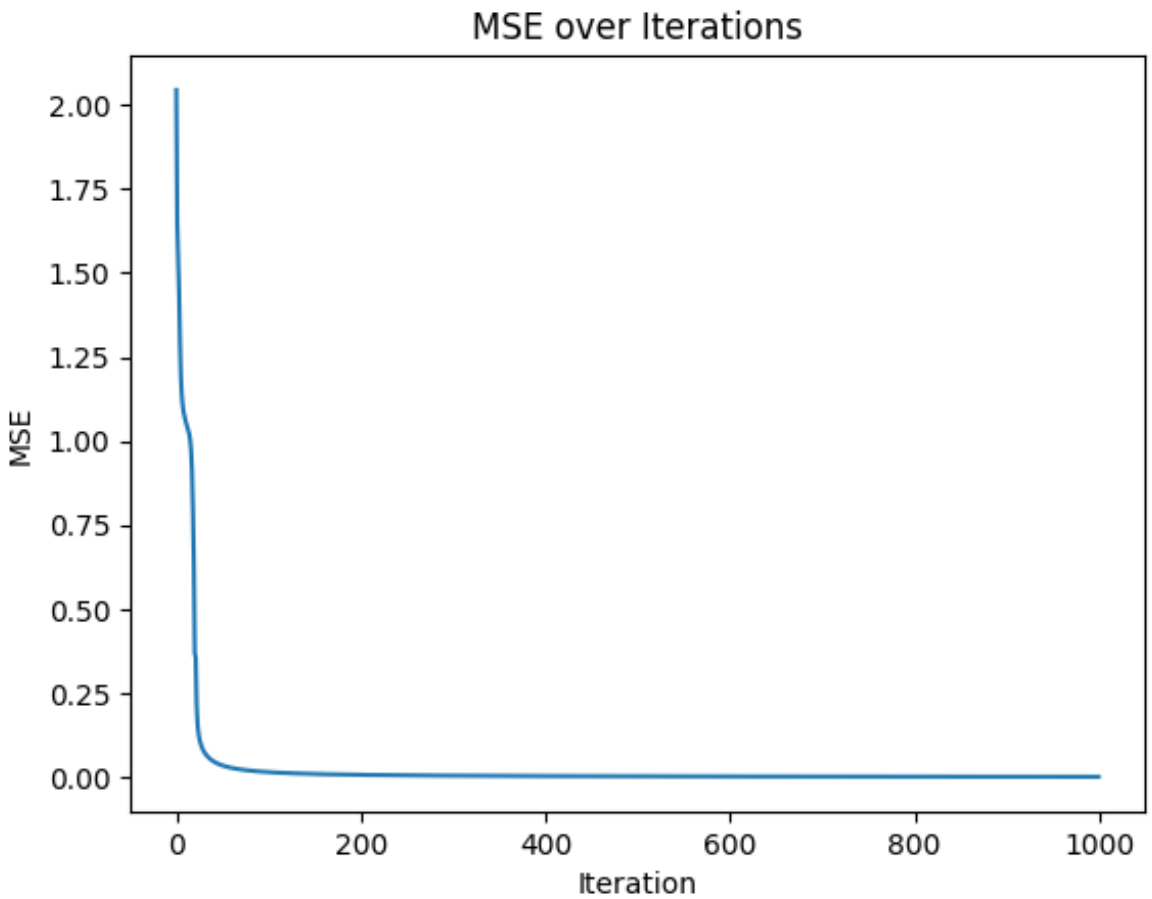
$$w \leftarrow w - \eta(y - t)y(1 - y)x$$

- **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()
```



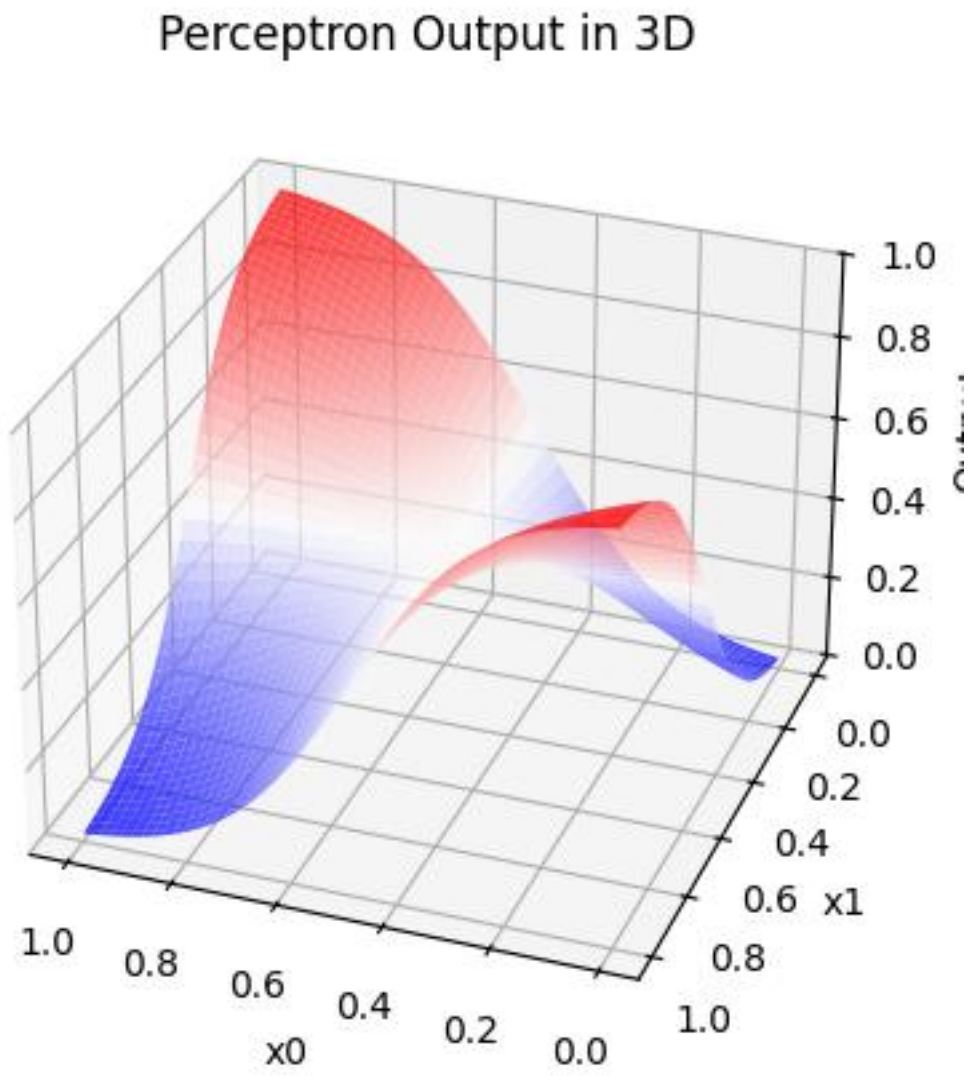
- **학습결과**

```
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

```python
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

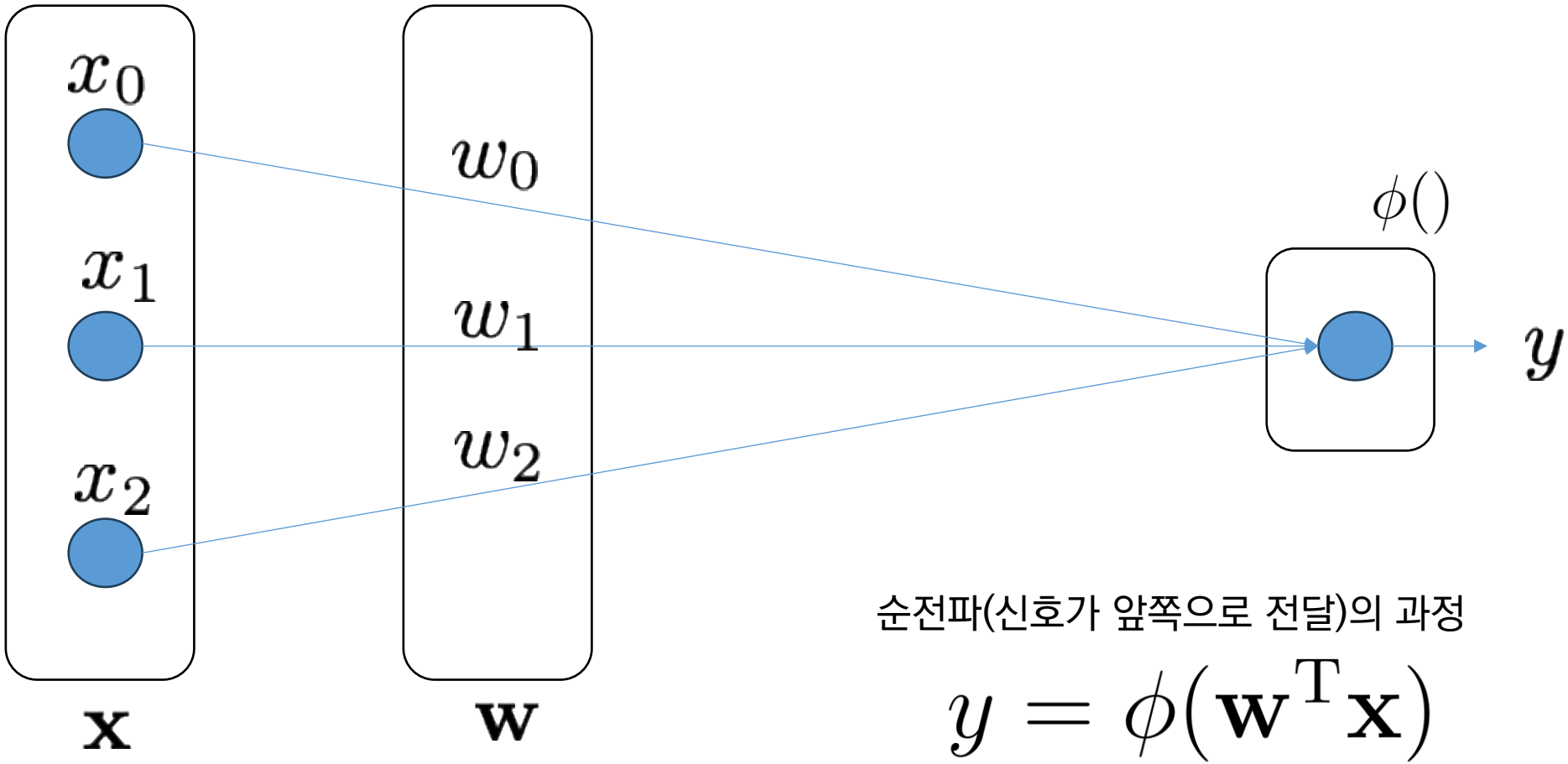
$$w \leftarrow w - \eta \cdot \partial E / \partial w$$

- 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

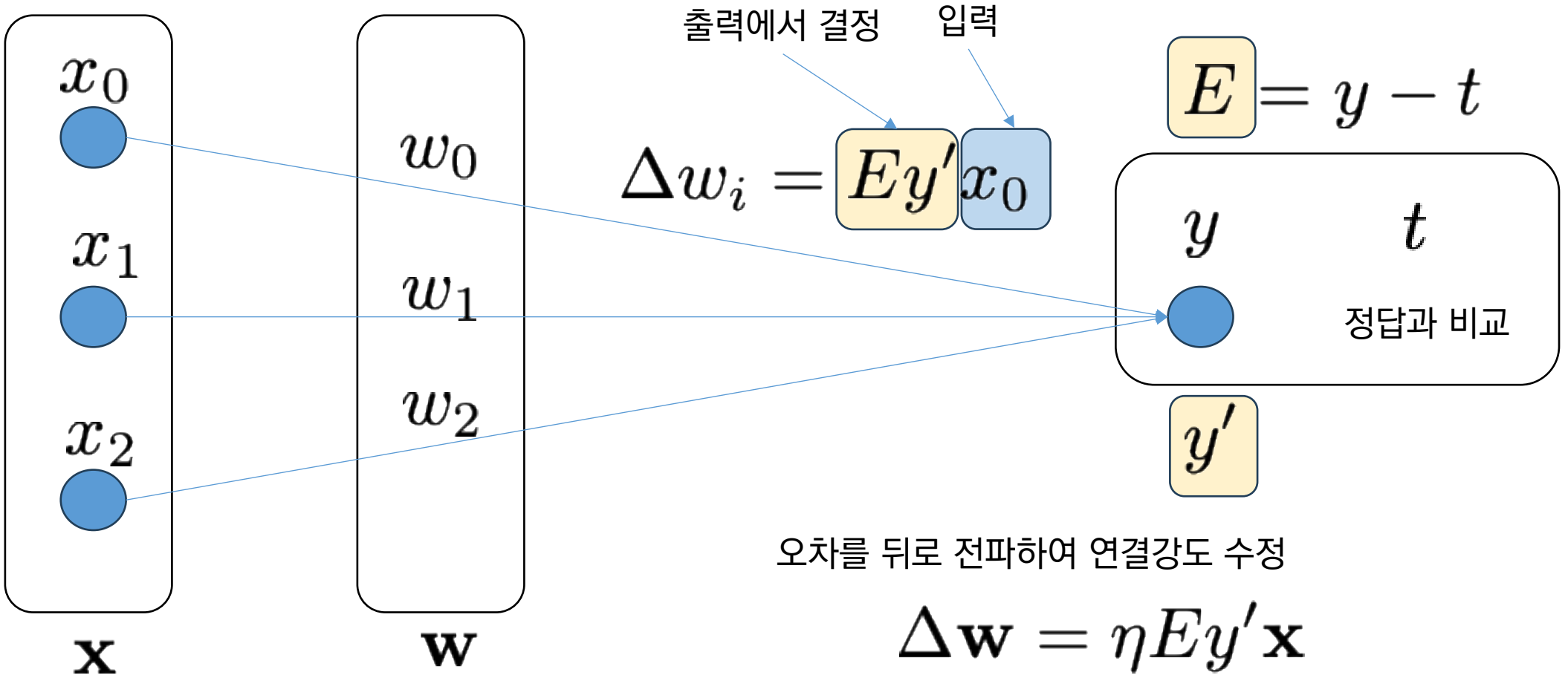
- **퍼셉트론의 동작을 벡터와 행렬의 곱으로 보자**



순전파(신호가 앞쪽으로 전달)의 과정

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

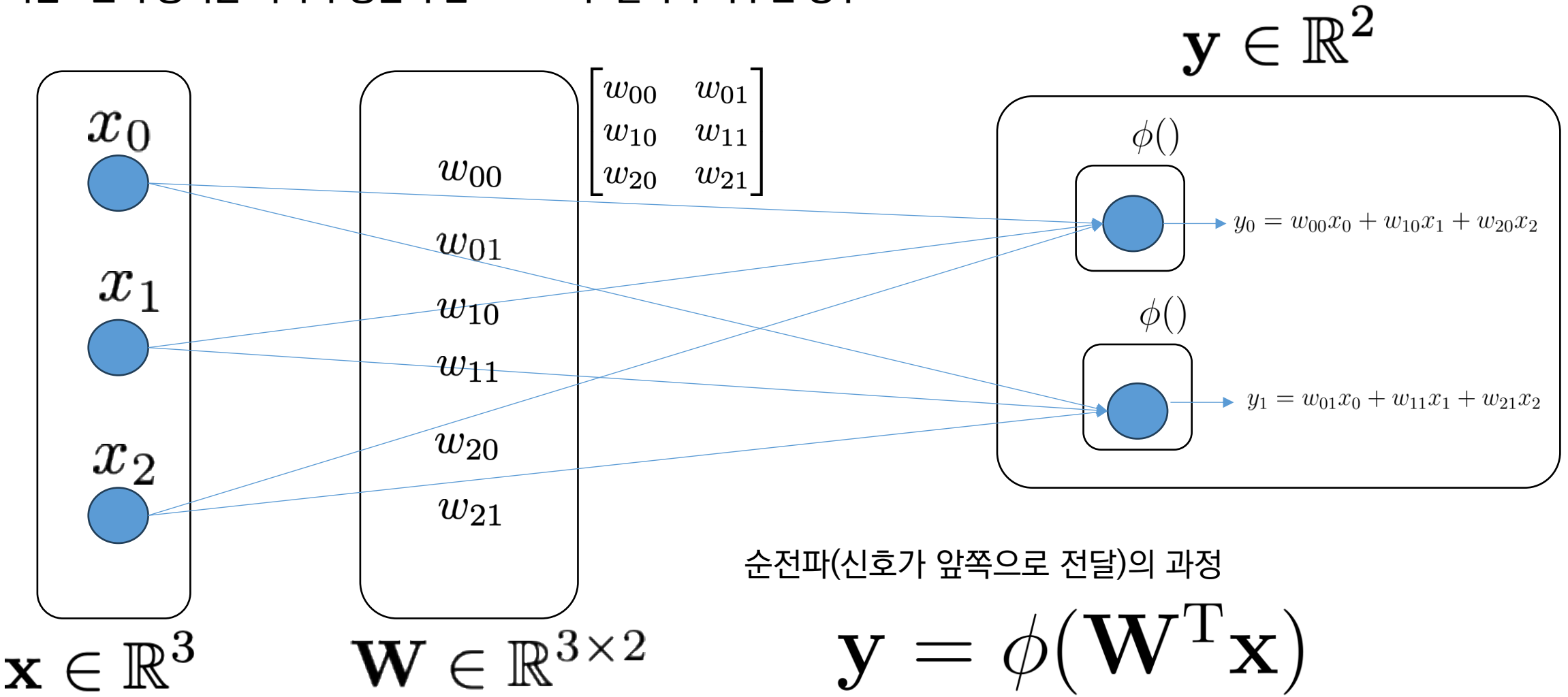
- **퍼셉트론의 동작을 벡터와 행렬의 곱으로 보자 : 오차 수정**



오차를 뒤로 전파하여 연결강도 수정

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

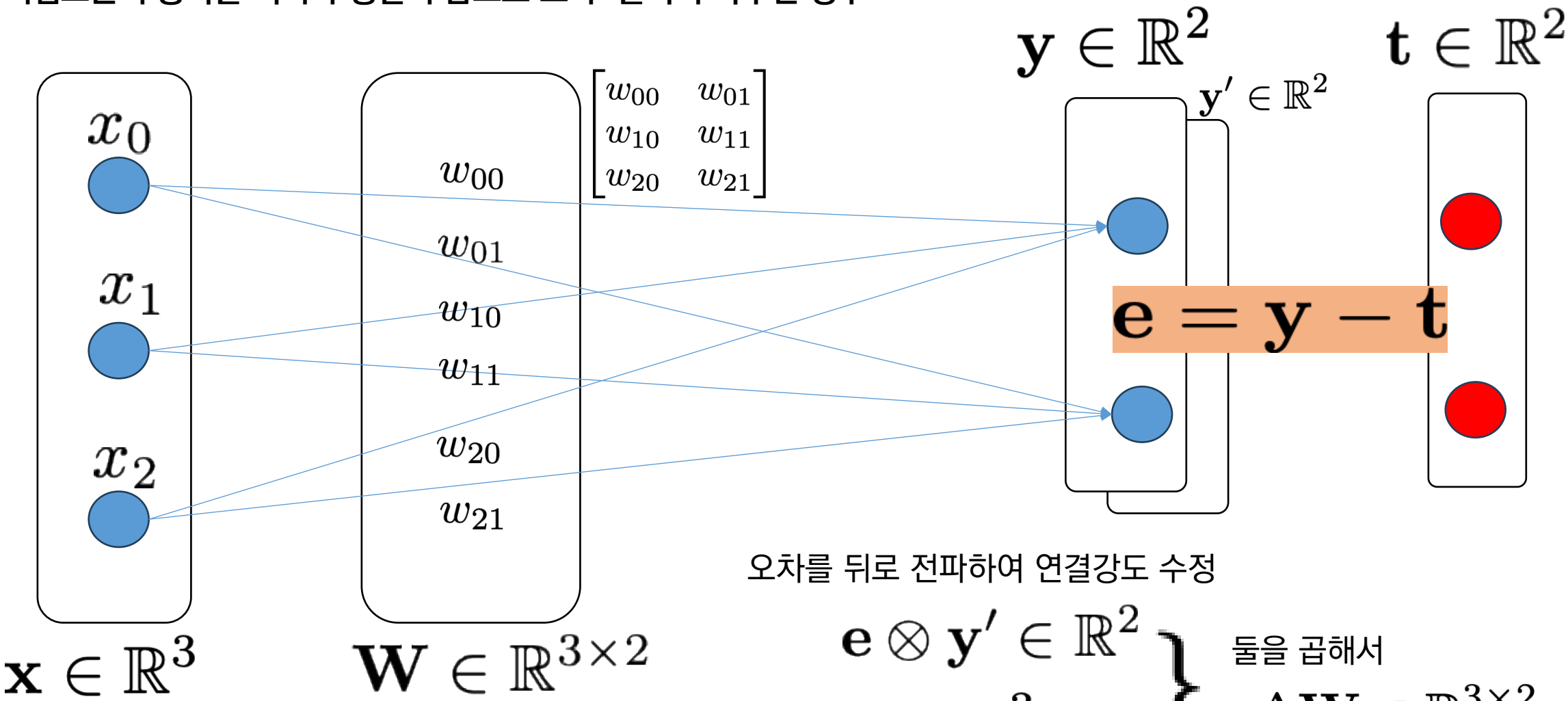
- 퍼셉트론의 동작을 벡터와 행렬의 곱으로 보자: 출력이 복수인 경우



순전파(신호가 앞쪽으로 전달)의 과정

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

- 퍼셉트론의 동작을 벡터와 행렬의 곱으로 보자: 출력이 복수인 경우



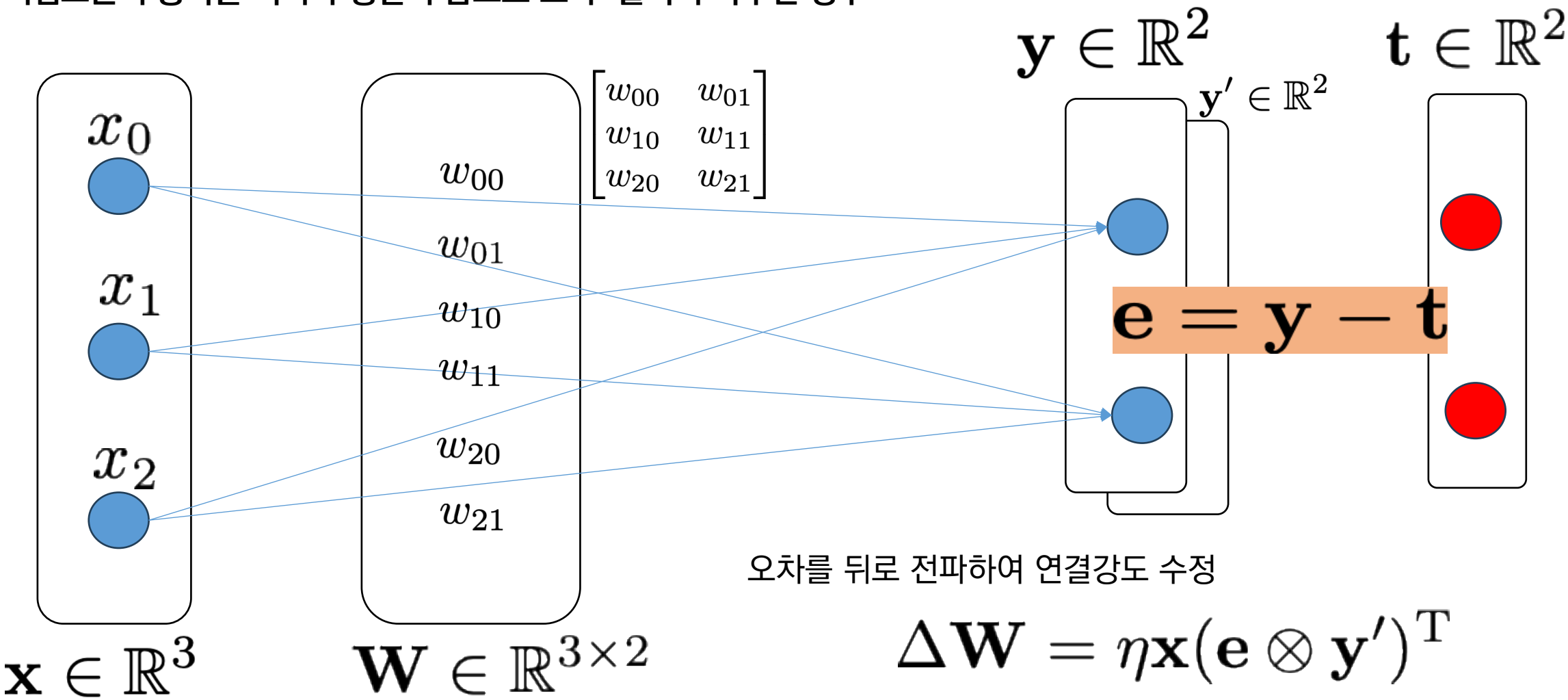
오차를 뒤로 전파하여 연결강도 수정

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

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

} 둘을 곱해서 $\Delta\mathbf{W} \in \mathbb{R}^{3\times 2}$

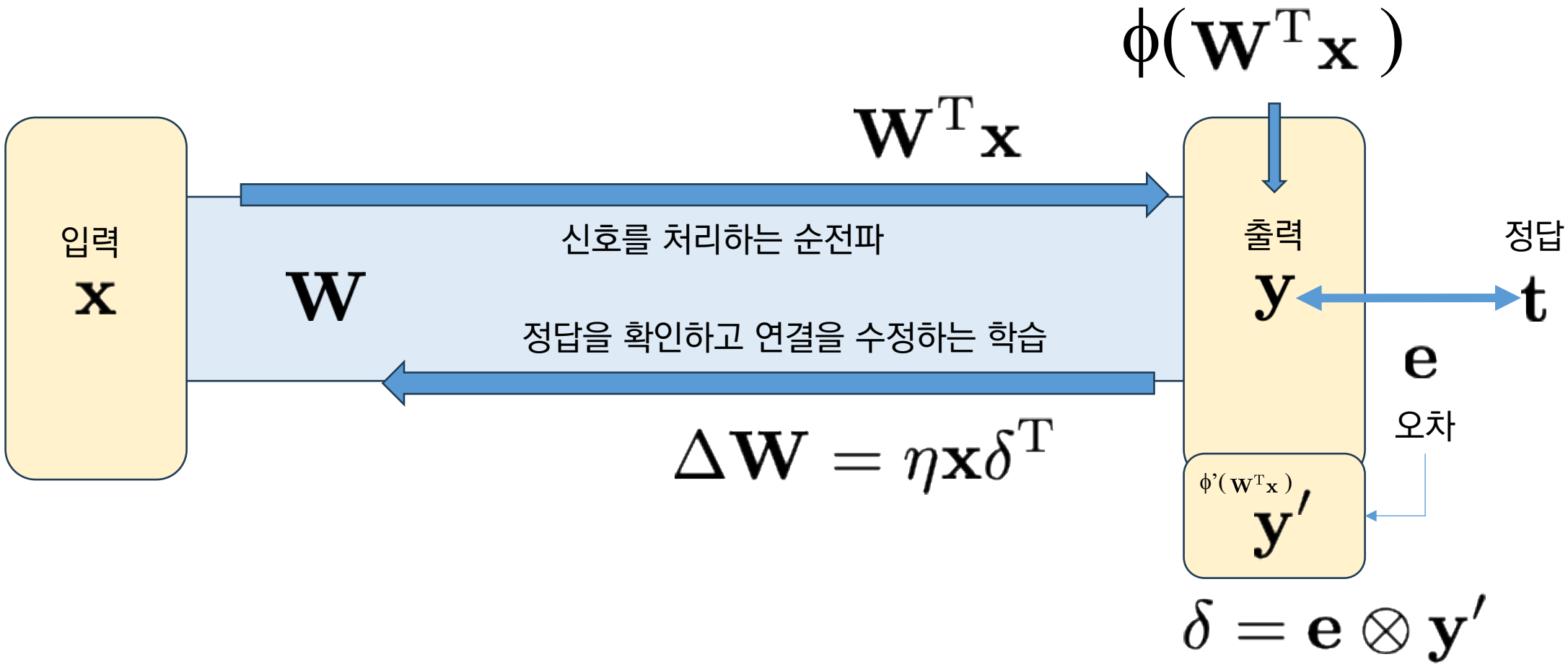
- 퍼셉트론의 동작을 벡터와 행렬의 곱으로 보자: 출력이 복수인 경우



오차를 뒤로 전파하여 연결강도 수정

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

- 벡터 / 행렬 계산으로 표현


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

- **벡터 / 행렬 계산으로 표현**

다항퍼셉트론은 벡터행렬 연산으로 표현

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

```
nX = 4
nY = 1
learning_rate = 0.1
X_poly = np.zeros((nX, ))   # nX 차원 벡터
W_poly = np.zeros((nX, nY)) # nX x nY 행렬
y = np.zeros((nY, ))        # nY 차원 벡터

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

def PolyPerceptron(x0, x1):
    global X_poly, W_poly
    X_poly = np.array([x0, x1, x0*x1, 1])    # 벡터 입력 (4차원)
    return phi(W_poly.T.dot(X_poly))         # phi( W.T x  )
```

```
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 [0.5]
0 1 [0.5]
1 0 [0.5]
1 1 [0.5]
array([[0.],
       [0.],
       [0.],
       [0.]])
```

- **학습 모델**

```
### 학습이 이루어지게 만들자
def Train(x0, x1, 정답):

    global W_poly

    출력 = PolyPerceptron(x0, x1)
    출력미분 = 출력 * ( 1- 출력 )

    오차 = 출력 - 정답
    델타 = 오차 * 출력미분
    # x delta^T : 입력벡터와 오차 델타의 외적(outer product) nX x 1 x 1 x nY => nX x nY
    # nX x nY 개 가중치에 대한 최적화 방향 계산
    오차제곱기울기 = np.outer(X_poly, 델타)

    W_poly -= learning_rate * 오차제곱기울기

    return 오차**2
```

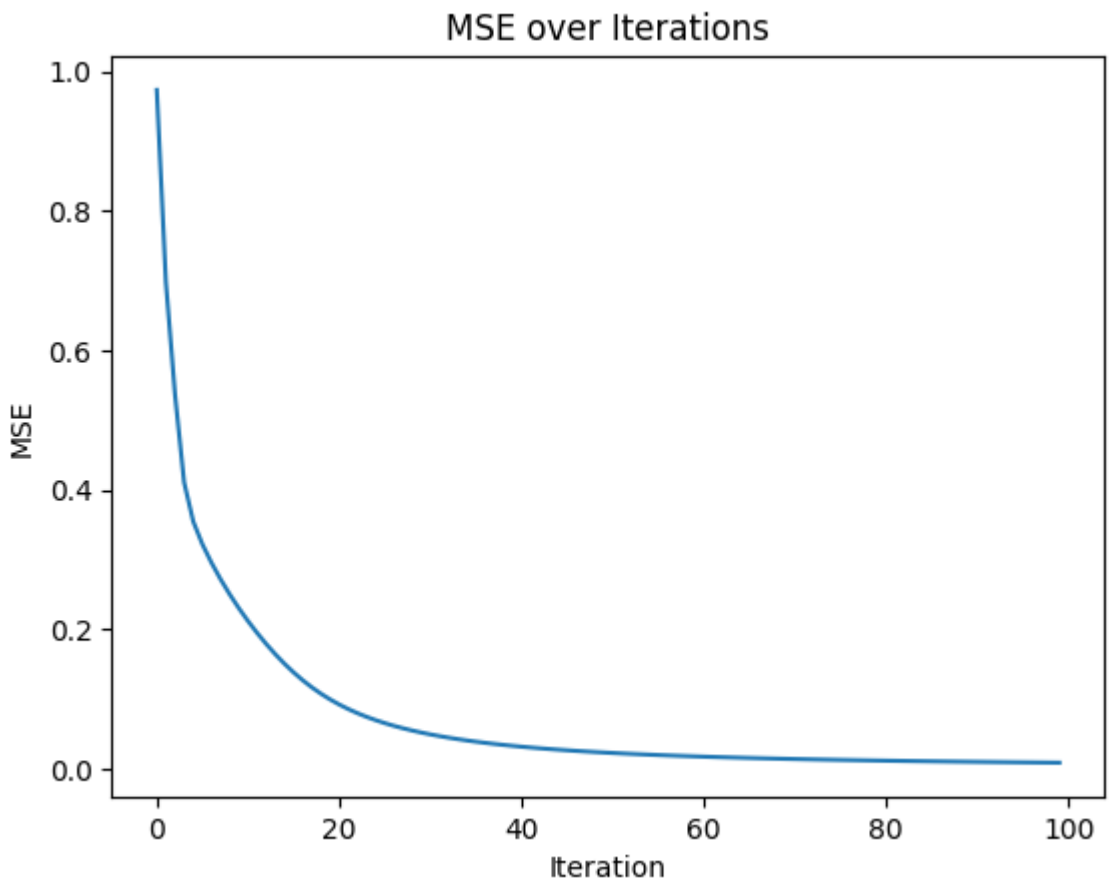
- **학습을 실시하자**

```
[ ]  W_poly = np.array([0.0, 0.0, 0.0, 0.0]).reshape((nX, nY))      # 파라미터
     learning_rate = 5.5
```

```
     MSE_list = []
     for i in range(100):
         MSE = 0.0
         MSE += Train(0, 0, 0.2)
         MSE += Train(0, 1, 1.0)
         MSE += Train(1, 0, 0.5)
         MSE += Train(1, 1, 0.0)

         MSE_list.append(MSE)
         if MSE < 0.0001:
             print(f'{i}회 반복 후 종료')
             break
```

- **오차의 변화**

```
import matplotlib.pyplot as plt
mse = np.array(MSE_list).flatten()
plt.plot(mse)
plt.xlabel('Iteration')
plt.ylabel('MSE')
plt.title('MSE over Iterations')
plt.show()
```



- **테스트 결과**

```
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 [0.22809666]
0 1 [0.93514101]
1 0 [0.48875107]
1 1 [0.05017192]
array([[ 1.17408651],
       [ 3.88757169],
       [-6.78339383],
       [-1.21908983]])
```

- **테스트 결과**

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

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

# Generate surface points
x0_vals = np.linspace(0, 1, 100)
x1_vals = np.linspace(0, 1, 100)
X0, X1 = np.meshgrid(x0_vals, x1_vals)
Z = np.zeros_like(X0)

for i in range(100):
    for j in range(100):
        Z[i, j] = PolyPerceptron(X0[i, j], X1[i, j])

ax.plot_surface(X0, X1, Z, cmap='viridis')

X0 = [0, 0, 1, 1]
X1 = [0, 1, 0, 1]
y  = [0.2, 1.0, 0.5, 0]
ax.scatter(X0, X1, y, color='red', label='Training Data')
ax.scatter(X0, X1, [ PolyPerceptron(X0[i], X1[i])[0] for i in range(len(X0)) ], color='blue', label='Test Data')

ax.set_xlabel('x0')
ax.set_ylabel('x1')
ax.set_zlabel('Output')
ax.set_title('Perceptron Decision Surface')
ax.legend()
plt.show()
```