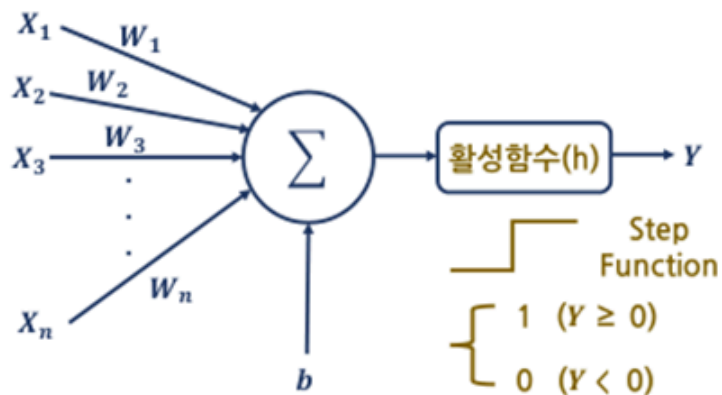
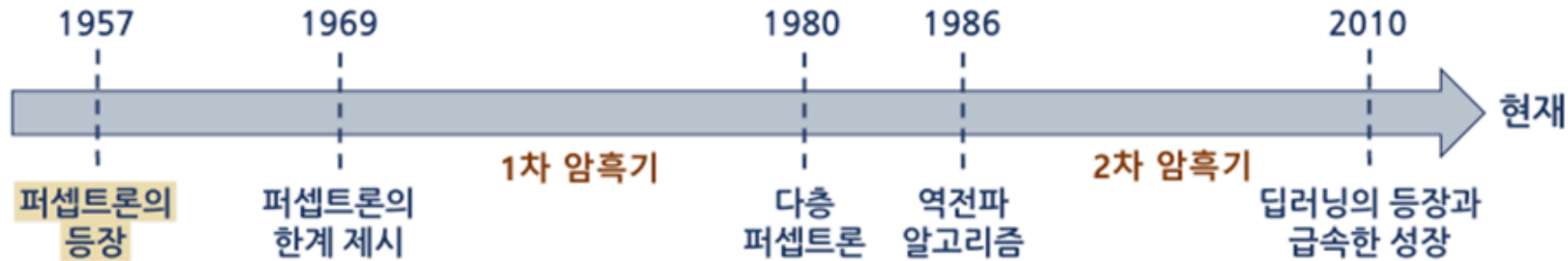


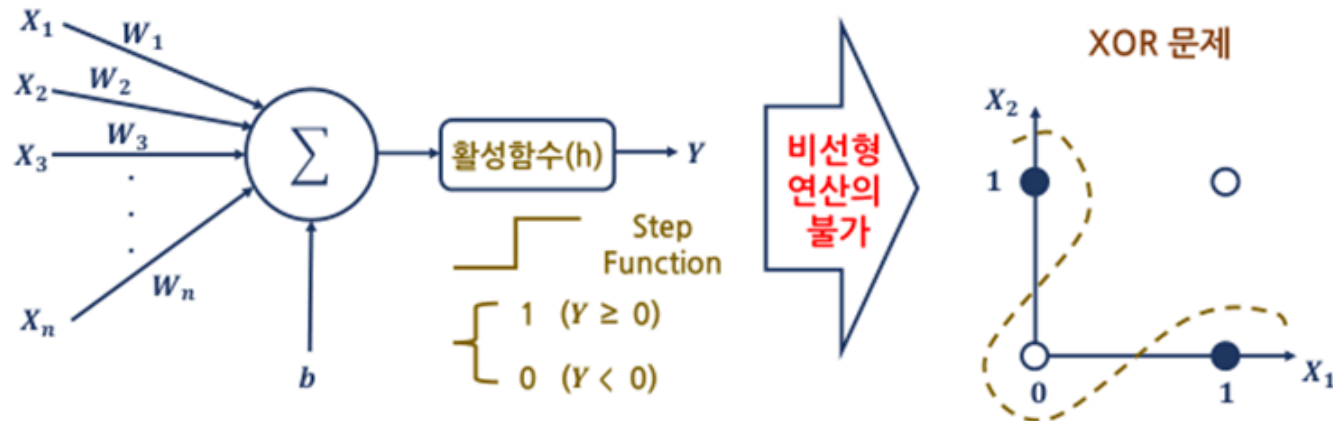
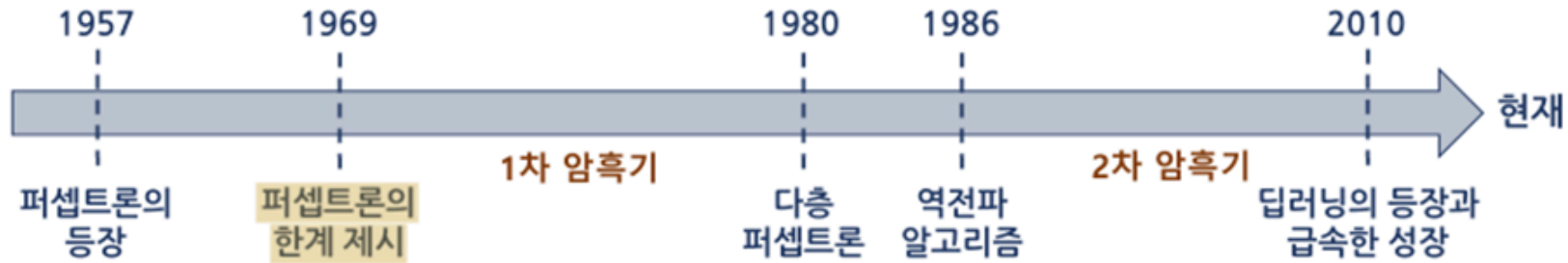
# 16강. 활성화함수의 필요성과 오류역전파

- 활성화함수의 필요성
- 시그모이드 함수
- 오류역전파 알고리즘

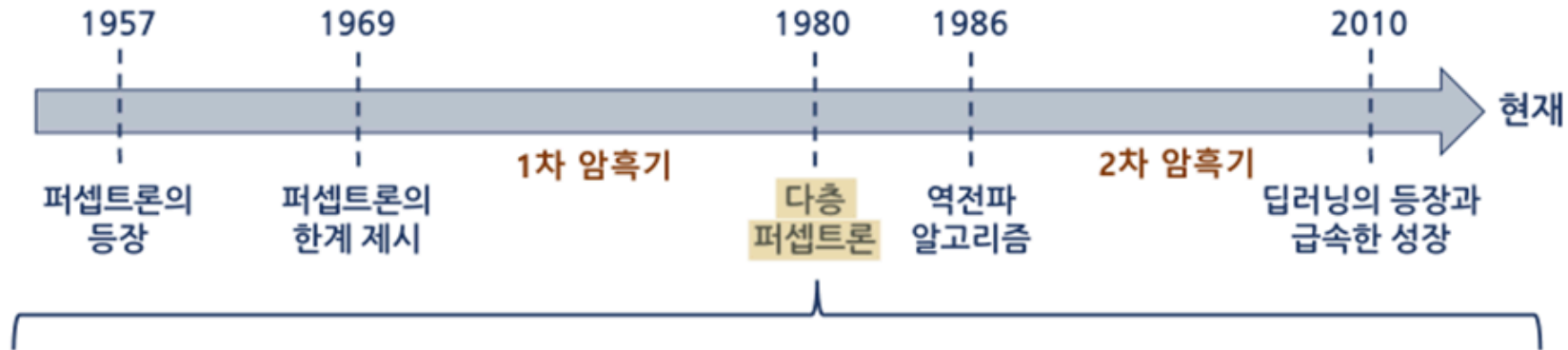
## ■ 단층 퍼셉트론의 등장과 한계



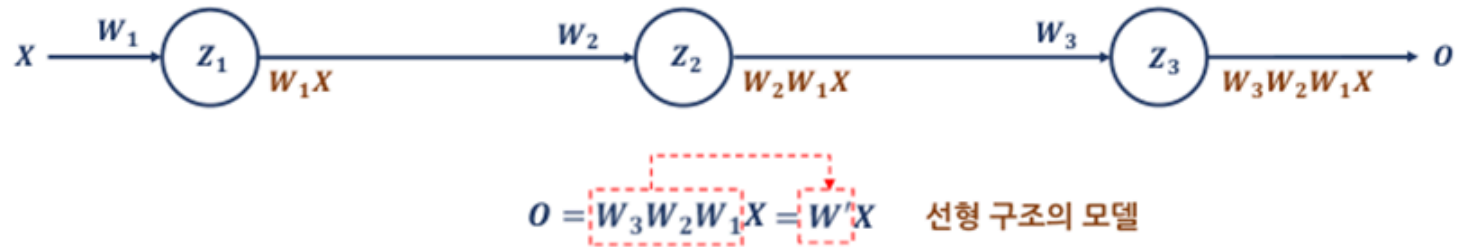
## ■ 단층 퍼셉트론의 등장과 한계



## ■ 활성화함수의 필요성

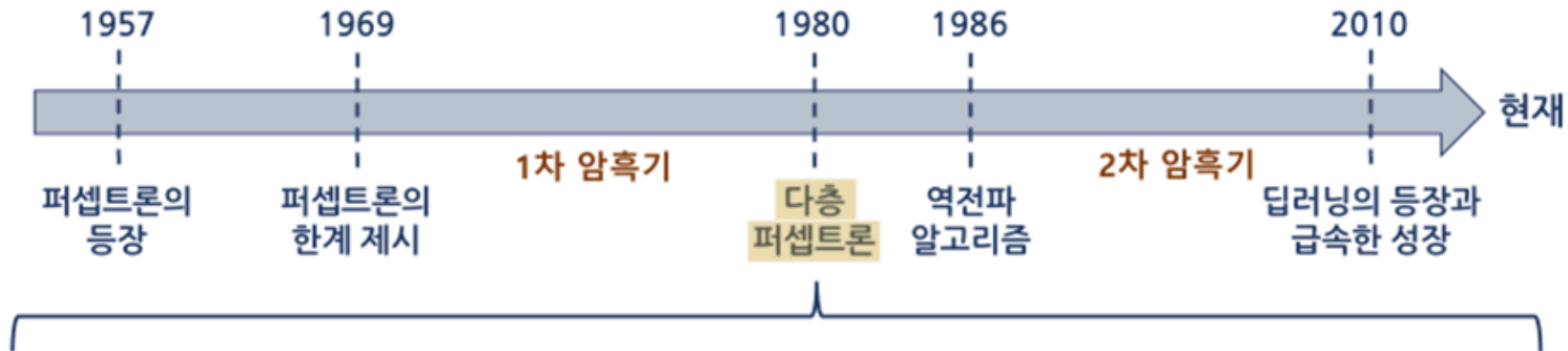


활성함수가 없는 다층 퍼셉트론

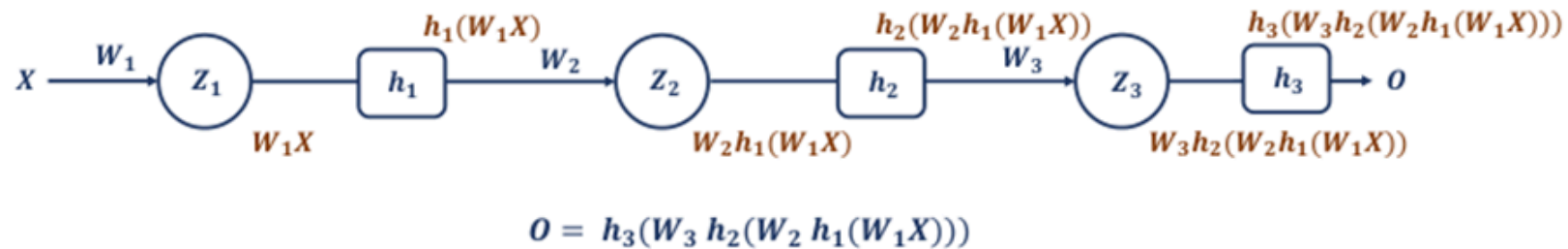


**활성함수가 없다면 다층신경망은 비선형 문제의 해결이 불가!**

## ■ 활성화함수의 필요성



### 활성화함수를 적용한 다층 퍼셉트론



활성함수로 모델의 복잡도를 높여 비선형 문제를 해결

### ✓ 활성화 함수(activation function)가 필요한 이유

- 신경망 모델의 각 layer에서는 input 값과  $W, b$ 를 곱, 합연산을 통해  $a=WX+b$ 를 계산하고 마지막에 활성화 함수를 거쳐  $h(a)$ 를 출력함
- 이렇게 각 layer마다 sigmoid, softmax, relu 등.. 여러 활성화 함수를 이용하는데 그 이유가 뭘까?

- 기존의 퍼셉트론은 AND와 OR문제는 해결할 수 있었지만 선형 분류기라는 한계에 의해 XOR과 같은 non-linear한 문제는 해결할 수 없었음



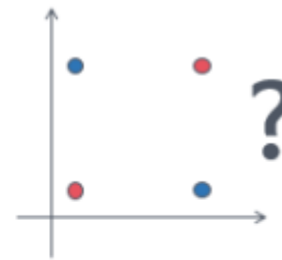
AND

AND		
Input_A	Input_B	Output
0	0	0
0	1	0
1	0	0
1	1	1



OR

OR		
Input_A	Input_B	Output
0	0	0
0	1	1
1	0	1
1	1	1



XOR

XOR		
Input_A	Input_B	Output
0	0	0
0	1	1
1	0	1
1	1	0

- 이를 해결하기 위해 나온 개념이 hidden layer이지만 이 hidden layer도 무작정 쌓기만 한다고 해서 퍼셉트론을 선형분류기에서 비선형분류기로 바꿀 수 있는 것은 아니었음

- 왜냐하면 선형 시스템이 아무리 깊어지더라도  $f(ax+by)=af(x)+bf(y)$ 의 성질 때문에 결국 하나의 layer로 깊은 layer를 구현할 수 있기 때문임
- 즉, linear한 연산을 갖는 layer를 수십개 쌓아도 결국 이는 하나의 linear 연산으로 나타낼 수 있음

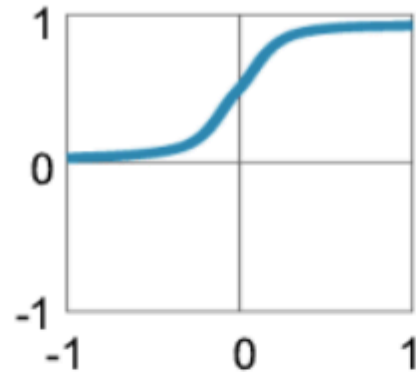
## ✓ 활성화함수

- 활성화 함수를 사용하면 입력값에 대한 출력값이 linear하게 나오지 않으므로 선형분류기를 비선형 시스템으로 만들 수 있음
  - 따라서 MLP(Multiple layer perceptron)는 단지 linear layer를 여러개 쌓는 개념이 아닌 활성화 함수를 이용한 non-linear 시스템을 여러 layer로 쌓는 개념임
  - 이렇게 활성화 함수를 이용하여 비선형 시스템인 MLP를 이용하여 XOR는 해결될 수 있지만, MLP의 파라미터 개수가 점점 많아지면서 각각의 weight와 bias를 학습시키는 것이 매우 어려워 다시 한 번 침체기를 겪게되었음 -> 이를 해결한 알고리즘이 바로 역전파(Back Propagation)임
- 

## ✓ 활성화 함수의 종류 (이진 계단 함수, 선형 활성화 함수, 비선형 활성화 함수)

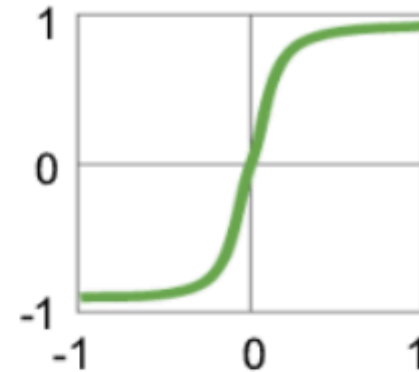
## Traditional Non-Linear Activation Functions

### Sigmoid



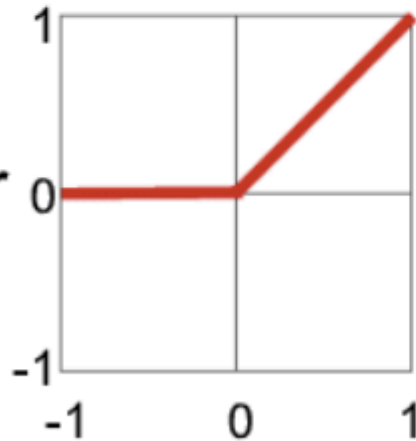
$$y = 1 / (1 + e^{-x})$$

### Hyperbolic Tangent



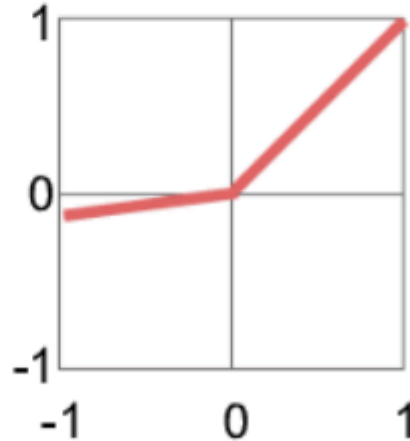
$$y = (e^x - e^{-x}) / (e^x + e^{-x})$$

### Rectified Linear Unit (ReLU)



$$y = \max(0, x)$$

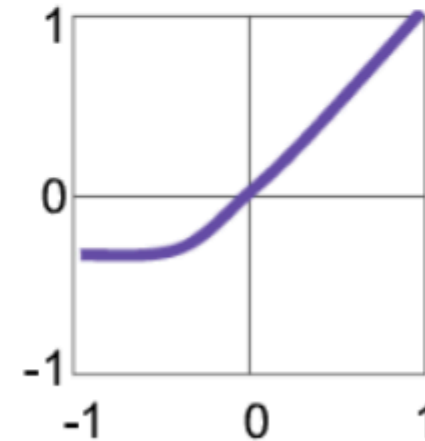
### Leaky ReLU



$$y = \max(\alpha x, x)$$

$\alpha$  = small const. (e.g. 0.1)

### Exponential LU



$$y = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$


## Modern Non-Linear Activation Functions

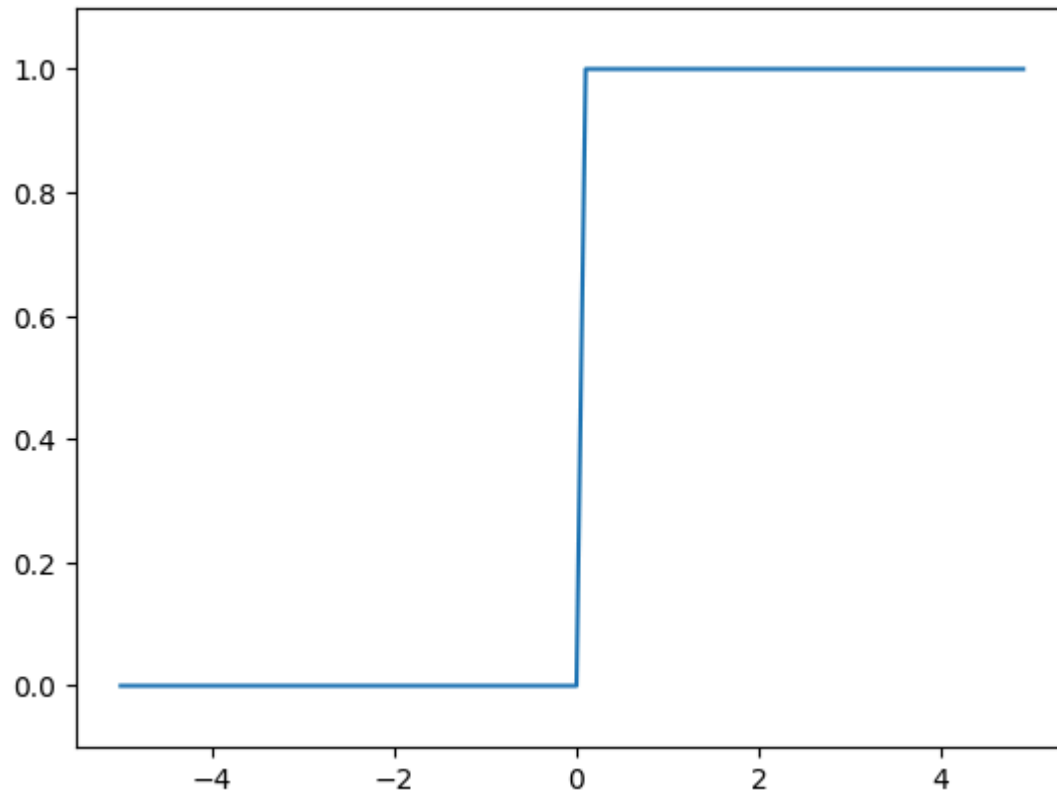
## ✓ 계단 함수(step function)

- 임계값을 경계로 출력이 바뀌는 활성화 함수
  - 입력이 0을 넘으면 1을 출력하고, 그 외에는 0을 출력하는 함수
  - 퍼셉트론에서는 활성화 함수로 "계단 함수"를 이용
  - 활성화 함수로 쓸 수 있는 여러 후보 중에서 퍼셉트론은 계단 함수를 채용
  - 활성화 함수를 계단 함수에서 다른 함수로 변경하는 것이 신경망의 key point!
- 

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def step_function(x):
5     return np.array(x>0, dtype=np)
6
7 x = np.arange(-5.0, 5.0, 0.1)
8 y = step_function(x)
9 plt.plot(x, y)
10 plt.ylim(-0.1, 1.1)
11 plt.show()
```



 <ipython-input-1-b3beb4ce1daa>:5: DeprecationWarning: in the future the `.dtype` attribute of a given datatype object must be a valid dtype  
return np.array(x>0, dtype=np)



## ✓ 시그모이드 함수(Sigmoid Function)

- S자 모양을 가진 비선형 함수로, 인공 신경망과 통계에서 주로 확률을 계산하거나 출력을 정규화하는 데 사용
- 다양한 시그모이드 함수들이 있으며, 대표적으로 다음과 같은 함수들이 있음

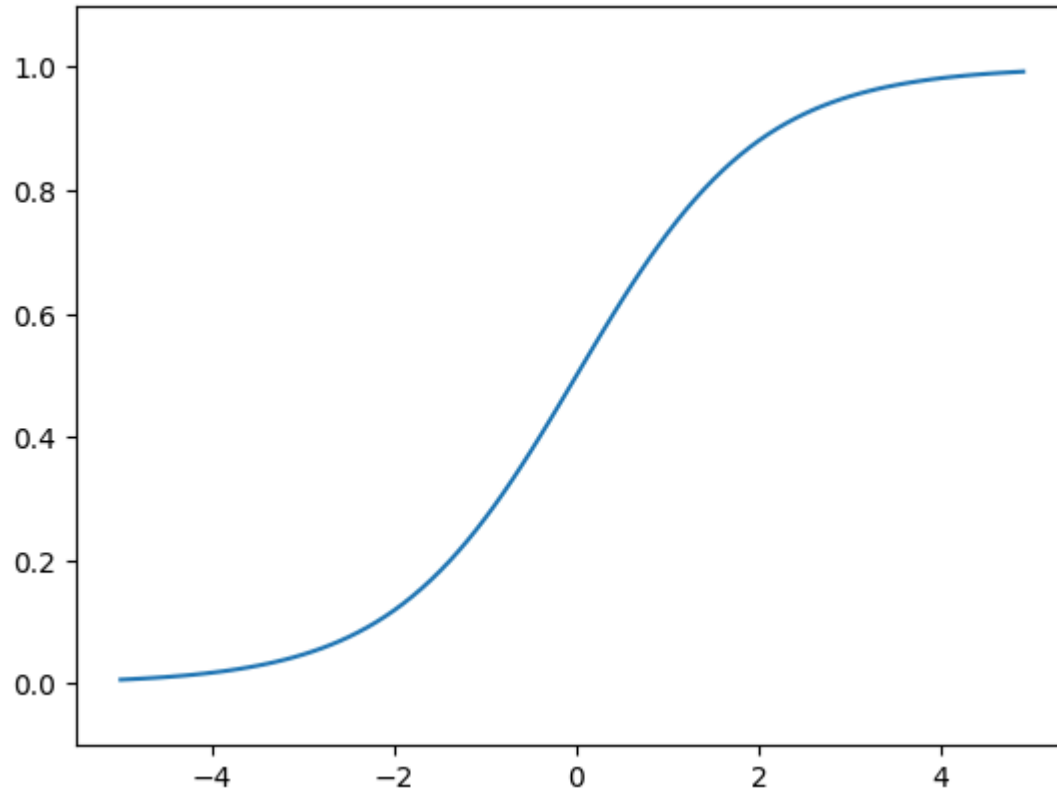
◦ 로지스틱 함수(Logistic Function)

- 하이퍼볼릭 탄젠트 함수 (Hyperbolic Tangent, Tanh)
- 아르크탄젠트 함수 (Arctan)
- 소프트사인 함수 (Softsign)
- Gaussian 시그모이드 함수
- Swish 함수

## ✓ 로지스틱 함수 (Logistic Function)

- 입력값을 0과 1 사이로 변환해, 확률을 계산하는 데 유용
- 기계 학습과 로지스틱 회귀에서 흔히 사용
- 용도 - 이진 분류 문제에서 확률 계산, 인공 신경망의 출력층 활성화 함수로 사용

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def sigmoid(x):
5     return 1 / (1 + np.exp(-x))
6
7 x = np.arange(-5.0, 5.0, 0.1)
8 y = sigmoid(x)
9 plt.plot(x, y)
10 plt.ylim(-0.1, 1.1)
11 plt.show()
```



## ✓ 시그모이드의 함수의 미분

- 역전파를 보내기 위해서는 해당 함수의 편미분값을 보내야함
- 시그모이드 함수를 수식적으로 미분해보면 다시 시그모이드의 함수로 구할 수 있음

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

$$\frac{df}{dt} = -\frac{1}{t^2}, \quad (t = 1 + e^{-x})$$

$$\Rightarrow \frac{1}{(1 + e^{-x})^2} \cdot e^{-x}$$

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

$$\Rightarrow \frac{1}{1 + e^{-x}} \cdot \frac{1}{(1 + e^{-x})^2}$$

$$\Rightarrow \frac{1}{1 + e^{-x}} \cdot \left(1 - \frac{1}{1 + e^{-x}}\right)$$

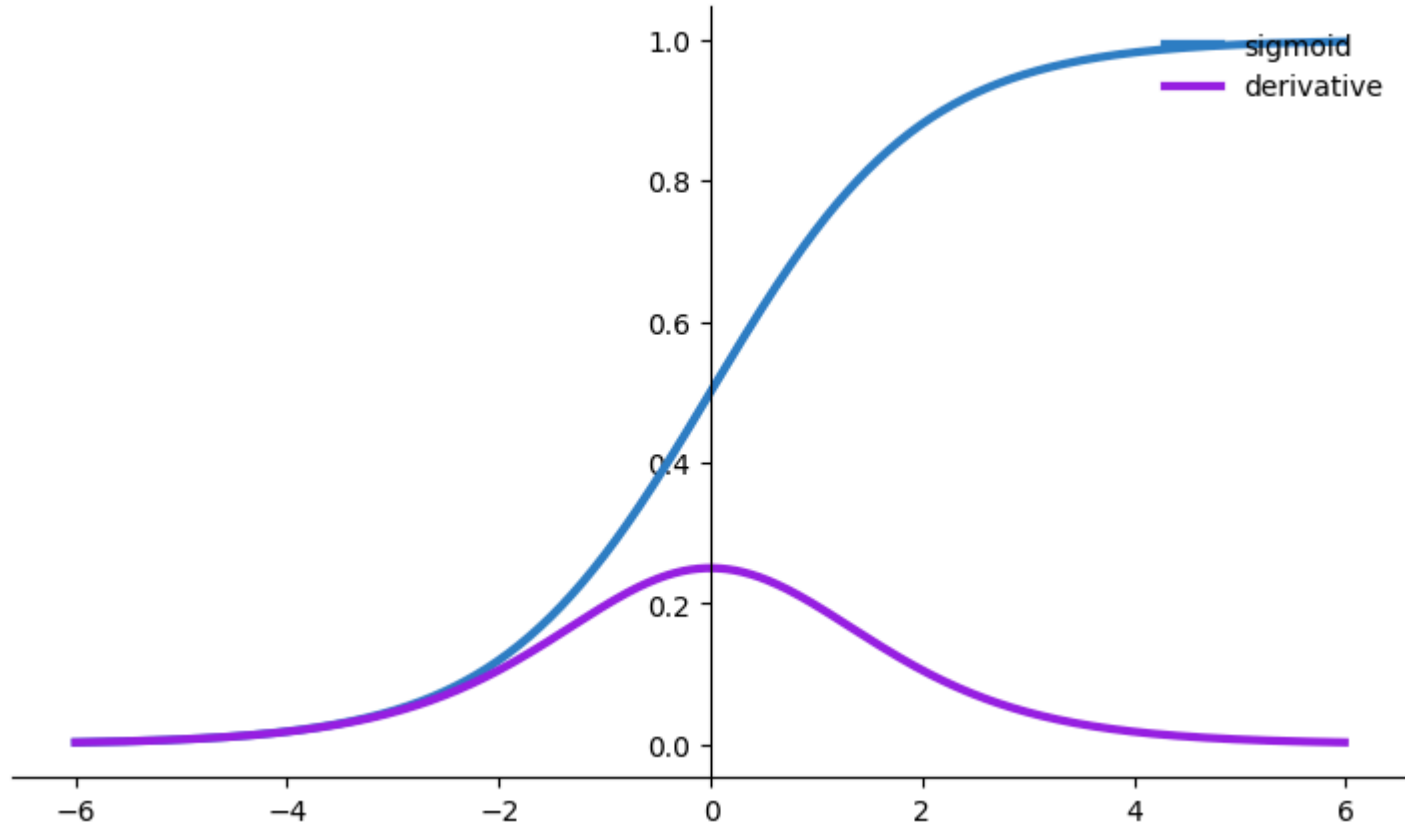
$$\Rightarrow f(x) \cdot (1 - f(x))$$

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 def sigmoid(x):
5     s=1/(1+np.exp(-x))
6     ds=s*(1-s)
7     return s,ds
8 x=np.arange(-6,6,0.01)
9 sigmoid(x)
10 # Setup centered axes
11 fig, ax = plt.subplots(figsize=(9, 5))
12 ax.spines['left'].set_position('center')
13 ax.spines['right'].set_color('none')
14 ax.spines['top'].set_color('none')
15 ax.xaxis.set_ticks_position('bottom')

```

```
16 ax.yaxis.set_ticks_position('left')
17 # Create and show plot
18 ax.plot(x, sigmoid(x)[0], color="#307EC7", linewidth=3, label="sigmoid")
19 ax.plot(x, sigmoid(x)[1], color="#9621E2", linewidth=3, label="derivative")
20 ax.legend(loc="upper right", frameon=False)
21 fig.show()
```



✓ 왜 현대의 모델들은 Relu(렐루)함수를 많이 쓰는 걸까?

- ReLU는 양수면 입력값을 출력하고, 아니면 0을 출력
- 양수의 입력값에 대해서는 기울기가 1이고, 음수 입력 값에 대해서는 기울기가 0임

$$f(x) = \max(0, x)$$

---

## ✓ ReLU가 Sigmoid보다 선호되는 이유

### • 1. 계산 효율성

- 활성화 함수의 계산 효율성은 입력이 주어졌을 때 출력을 얼마나 빨리 계산할 수 있는가임
- 대규모 데이터셋이나 복잡한 모델을 다룰 때는 당연히 더 간단한 함수가 계산 효율성이 높을 것임
- 그런 측면에서 sigmoid는 지수 함수가 포함되기 때문에 나눗셈 같은 복잡한 수학적 연산이 필요하지만, ReLU는 간단하게 계산할 수 있기 때문에 계산 효율성을 높일 수 있고 이는 훈련 프로세스를 더 빠르게 만들고 최적의 값으로 수렴하는 속도도 높여줄 수 있음

### • 2. Gradient(기울기) 소실 방지

- sigmoid 함수는 그래디언트 소실 문제가 있음
- 입력값이 매우 크거나 작다면 기울기도 0에 가까워짐
- 기울기가 0에 가까워지면 가중치가 매우 느리게 업데이트되므로 학습 과정이 늦어지고, 신경망이 수렴하기 어렵게 됨
- local minimum에 갇히게 될 수도 있음
- ReLU는 양수 입력 값에 대해 일정한 기울기를 갖고 있으므로 이 문제를 방지할 수 있다는 것임!!!!

### • 3. Saturation problem(포화 문제) 방지

- 심층 네트워크에 더 적합

- ReLU는 시그모이드 함수에서 발생할 수 있는 포화 문제를 방지할 수 있음
- 여기서 포화는 활성화 함수의 출력이 1 또는 0에 매우 가까워져서 가중치와 편향을 업데이트하기 어려워져버려 신경망의 학습 능력이 제한되어 버리는 경우임

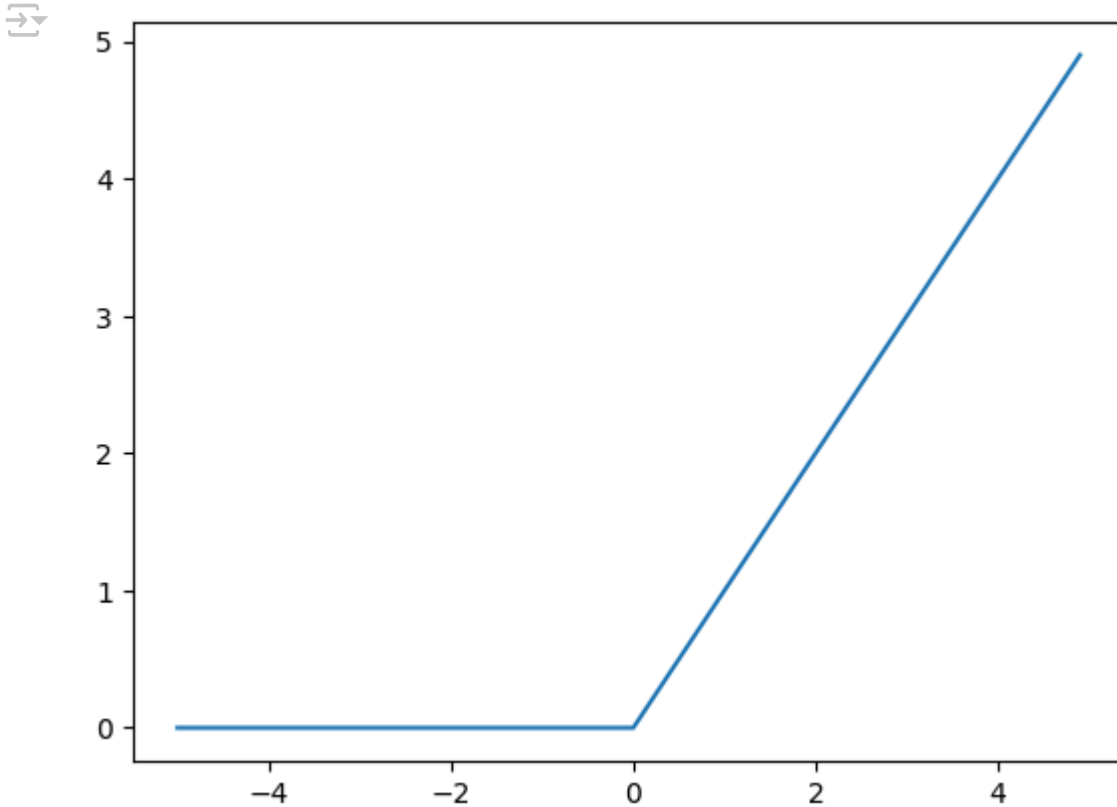
- 4. ReLU는 sparse activation(희소 활성화)를 생성함
- 희소 활성화라는 의미는 많은 뉴런이 0을 출력하게 해서 활성화를 적게 한다는 뜻임 - 한 번에 몇 개의 뉴런만 활성화됨
- ReLU함수의 그래프를 생각해보면, 음수는 모두 0을 출력하고 양수만 변경하지 않고 출력하게 되는데 이를 희소 활성화라고 함.
- 이렇게 하면 계산 비용이 줄고, 과적합을 줄이는 데도 도움이 됨(희소 활성화는 보통 자연어처리, 이미지인식 시에 일반화 성능을 높이는데 도움이 됨)
- 단점

- ReLU함수에도 단점이 있는데, DNN에서 일부 뉴런이 비활성화되어서 출력이 0이 되는 'dying ReLU'문제를 겪을 수 있음
- 이 문제를 완화하려면 Leaky ReLU 함수를 사용해서, ReLU함수의 음수 부분에 작은 기울기를 추가해서 음수 값에 대한 기울기가 0인 문제를 방지할 수 있음!!!

```

1 # -*- coding: utf-8 -*-
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 def relu(x):
6     return np.maximum(0,x)
7
8 x = np.arange(-5.0,5.0,0.1)
9 y = relu(x)
10 plt.plot(x,y)
11 plt.show()

```



## ✓ 하이퍼볼릭 탄젠트 함수(Hyperbolic Tangent, Tanh)

- 출력 범위가 -1에서 1 사이로, 로지스틱 함수와 비슷하지만 중심이 0에 있어 데이터가 양수와 음수로 구분될 때 더 유리함
- 용도 - 신경망에서 은닉층 활성화 함수로 자주 사용되며, 데이터가 -1에서 1 사이로 정규화될 때 효과적임

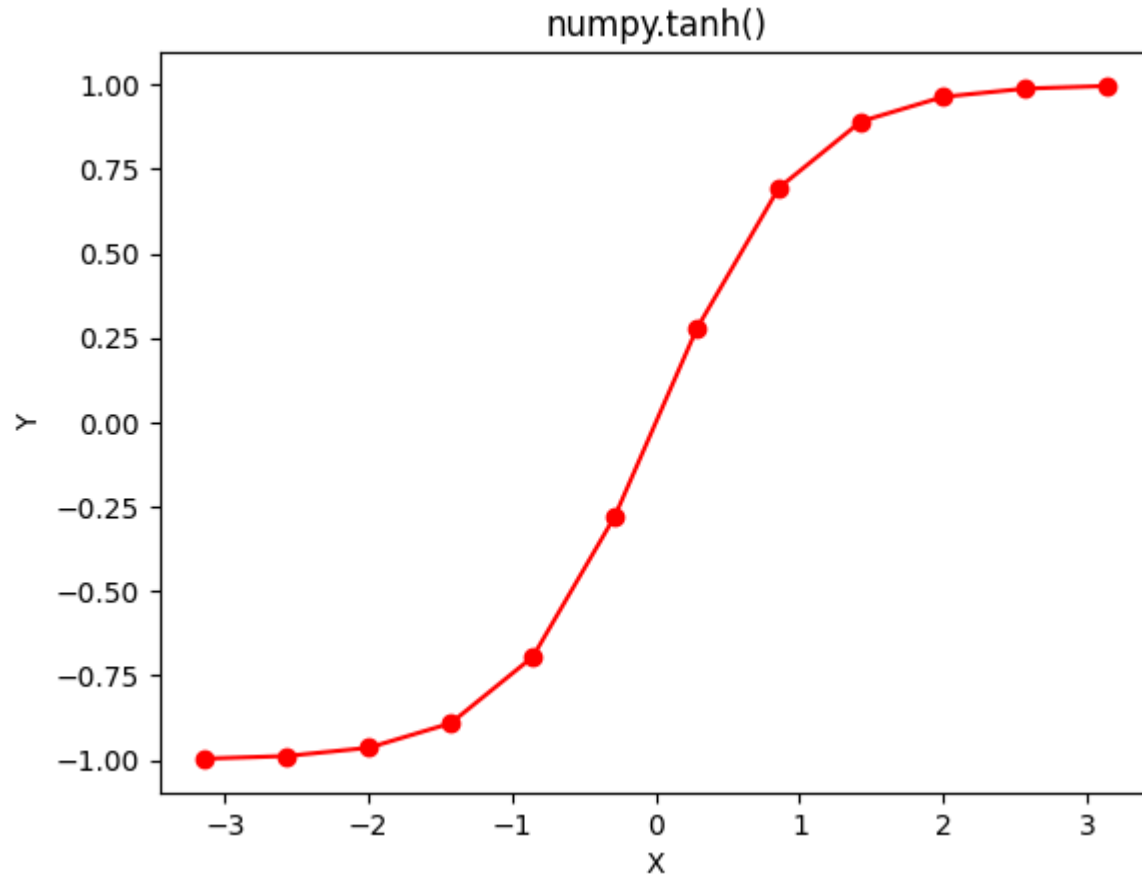
```
1 # Python program showing Graphical  
2 # representation of tanh() function  
3 import numpy as np
```



```
4 import matplotlib.pyplot as plt
5
6 in_array = np.linspace(-np.pi, np.pi, 12)
7 out_array = np.tanh(in_array)
8
9 print("in_array : ", in_array)
10 print("out_array : ", out_array)
11
12 # red for numpy.tanh()
13 plt.plot(in_array, out_array, color = 'red', marker = "o")
14 plt.title("numpy.tanh()")
15 plt.xlabel("X")
16 plt.ylabel("Y")
17 plt.show()
```

```
in_array : [-3.14159265 -2.57039399 -1.99919533 -1.42799666 -0.856798 -0.28559933  
0.28559933 0.856798 1.42799666 1.99919533 2.57039399 3.14159265]
```

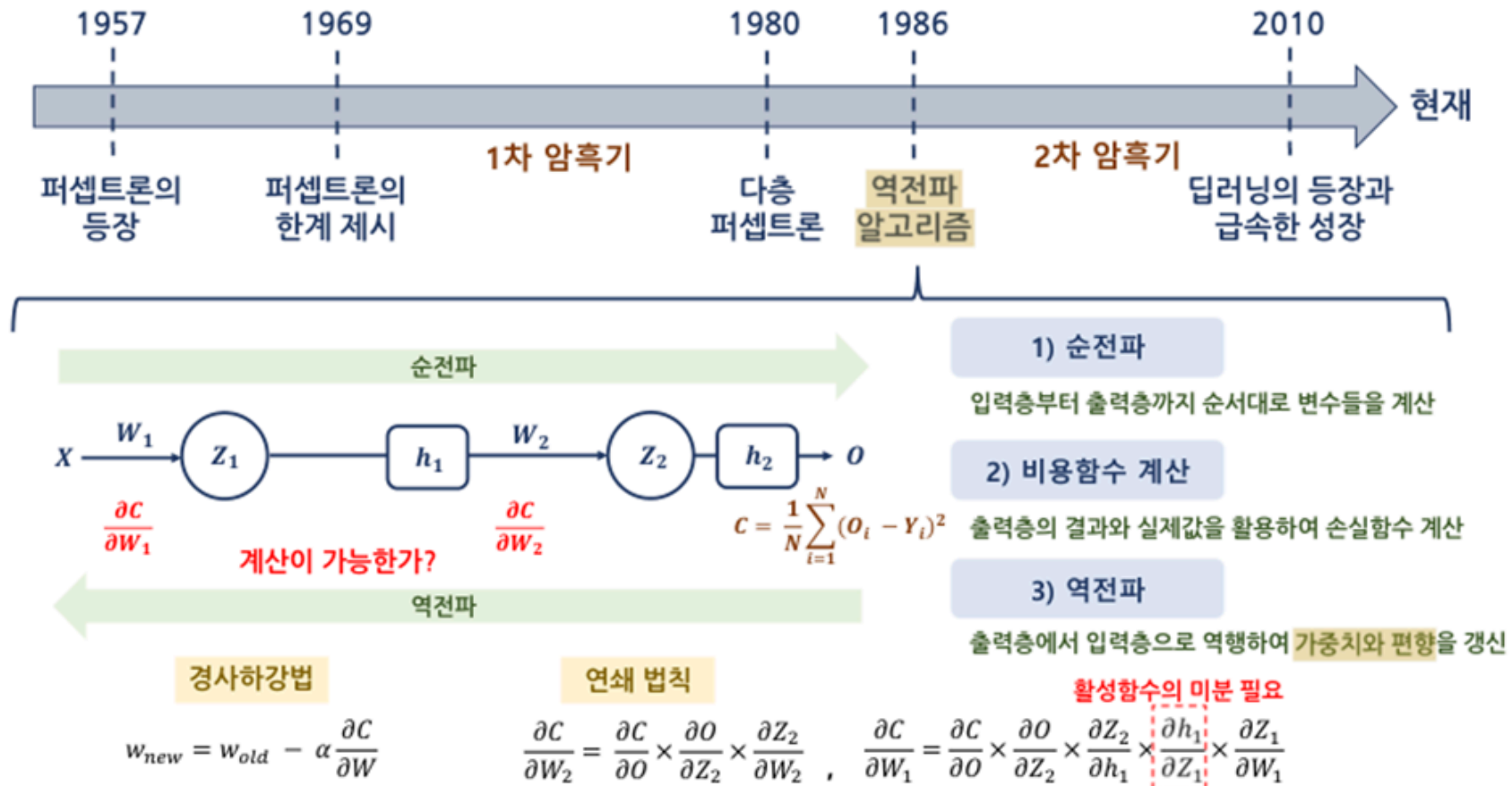
```
out_array : [-0.99627208 -0.98836197 -0.96397069 -0.89125532 -0.69460424 -0.27807943  
0.27807943 0.69460424 0.89125532 0.96397069 0.98836197 0.99627208]
```



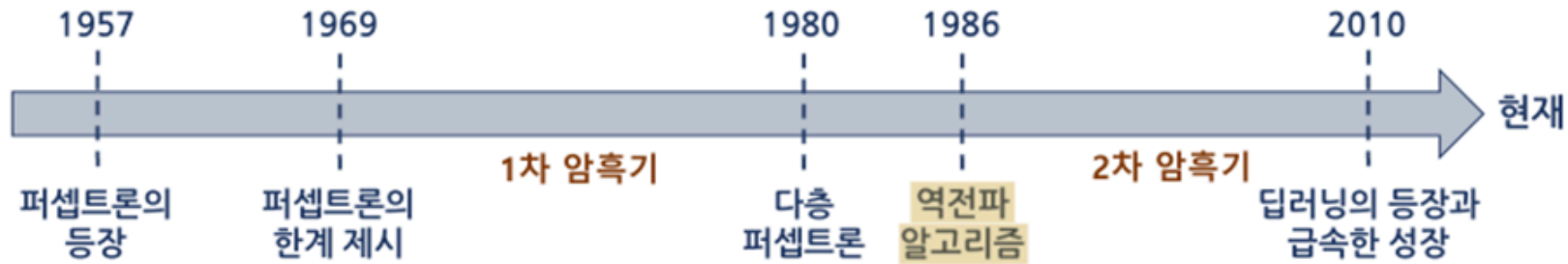
## ▽ 역전파

- 인공 신경망의 최적화에 필수적인 머신 러닝 기법
- 역방향으로 오차를 전파시키면서 각 층의 가중치를 업데이트하고 최적의 학습 결과를 찾아가는 방법
- 이는 최신 인공지능(AI)을 구동하는 딥 러닝 모델이 '학습'하는 방식인 네트워크 가중치를 업데이트하는 데 경사 하강법 알고리즘을 사용할 수 있도록 지원

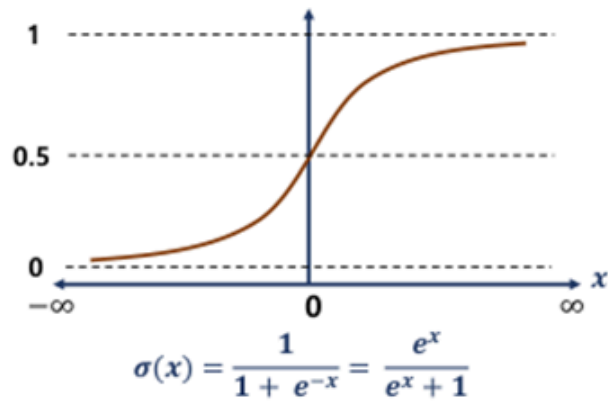
## ■ 역전파 알고리즘



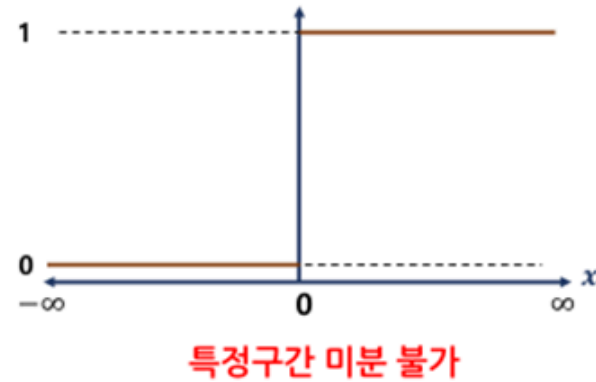
## 역전파 알고리즘



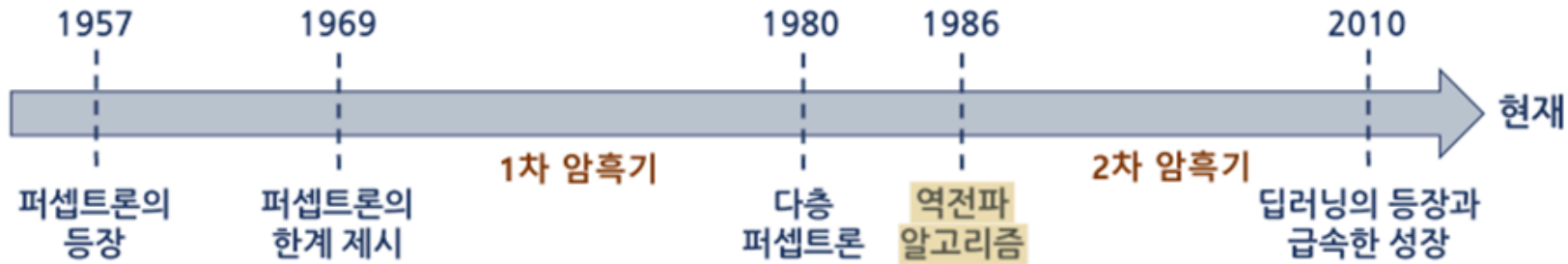
시그모이드



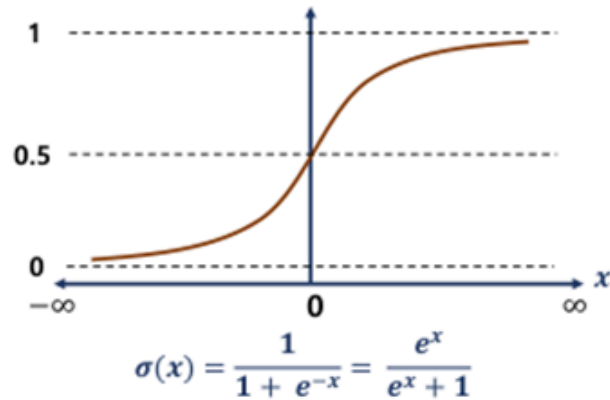
계단 함수



## 역전파 알고리즘



### 시그모이드

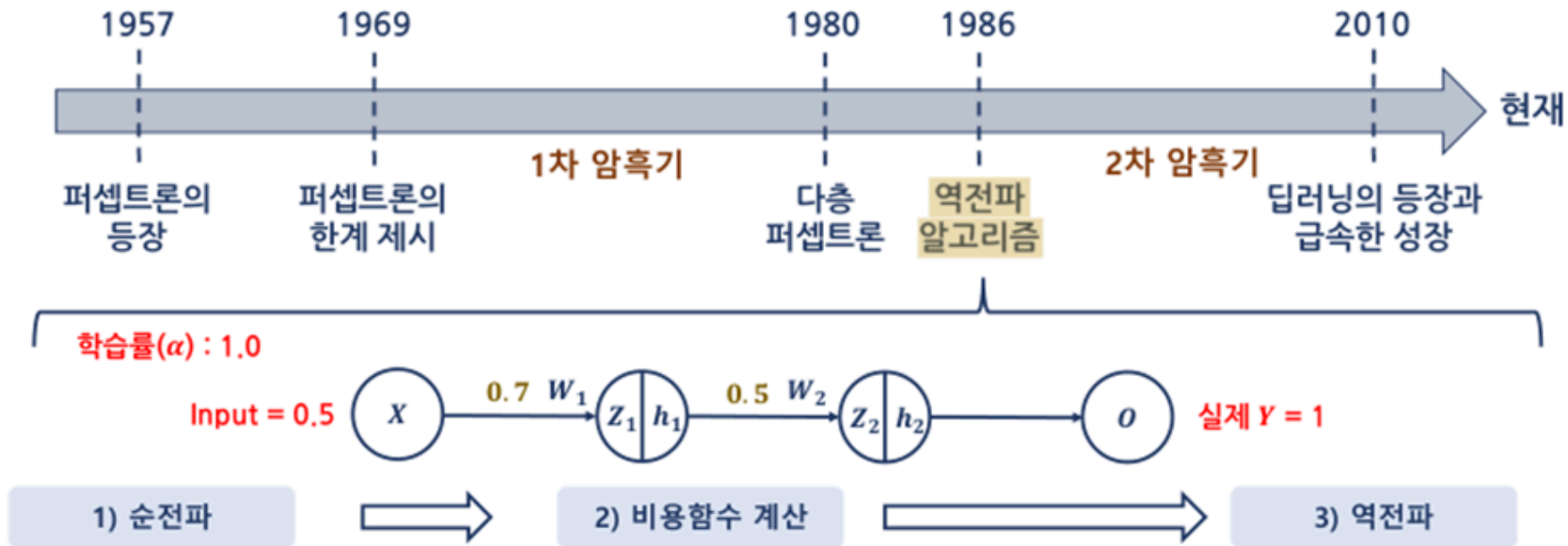


- 0 ~ 1 사이의 신호를 전달
- 전 구간에서의 미분이 가능
- 미분 계산이 단순 (컴퓨팅 연산 효율)

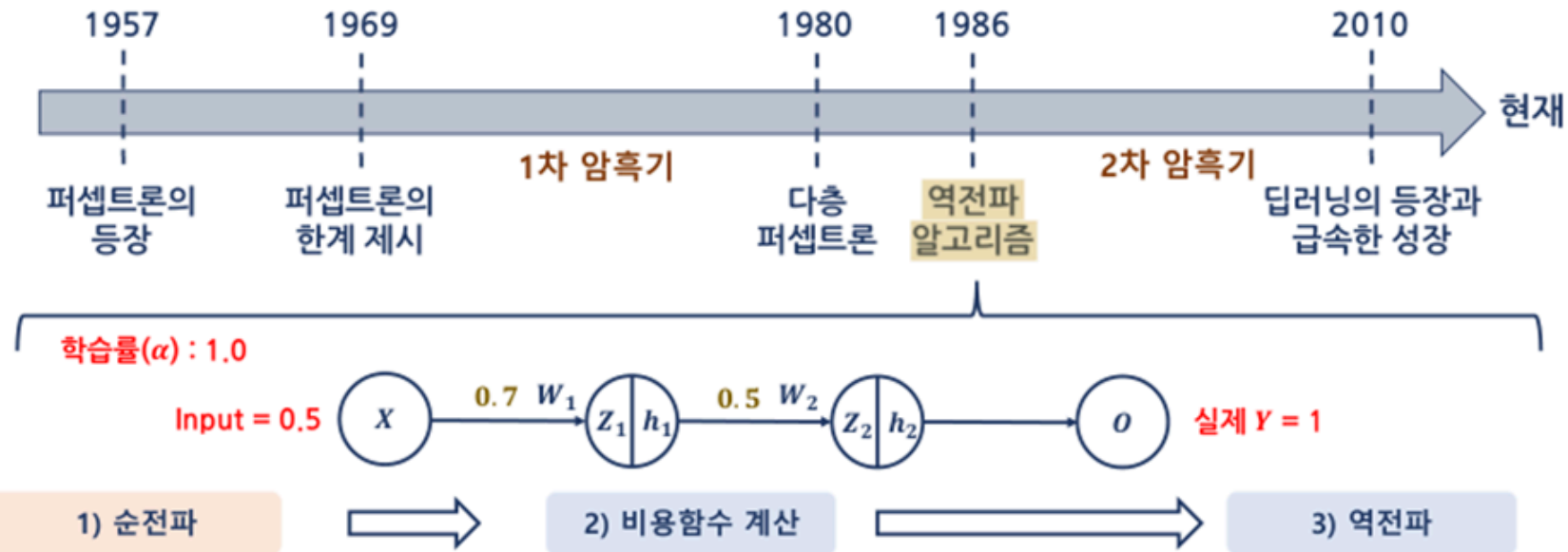
$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

시그모이드 함수를 활성화함수로 사용

## 역전파 알고리즘

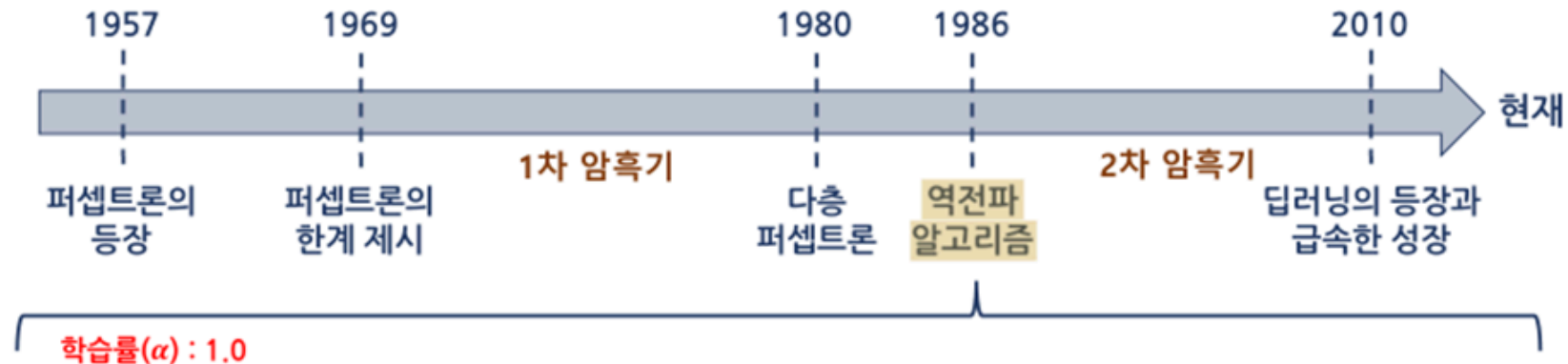


## 역전파 알고리즘



- $Z_1 = W_1 X = 0.7 \times 0.5 = 0.35$
- $h_1 = \sigma(Z_1) = \sigma(0.35) = 0.587$
- $Z_2 = W_2 h_1 = 0.5 \times 0.587 = 0.294$
- $O = h_2 = \sigma(Z_2) = \sigma(0.294) = 0.573$

## 역전파 알고리즘



1) 순전파

2) 비용함수 계산

3) 역전파

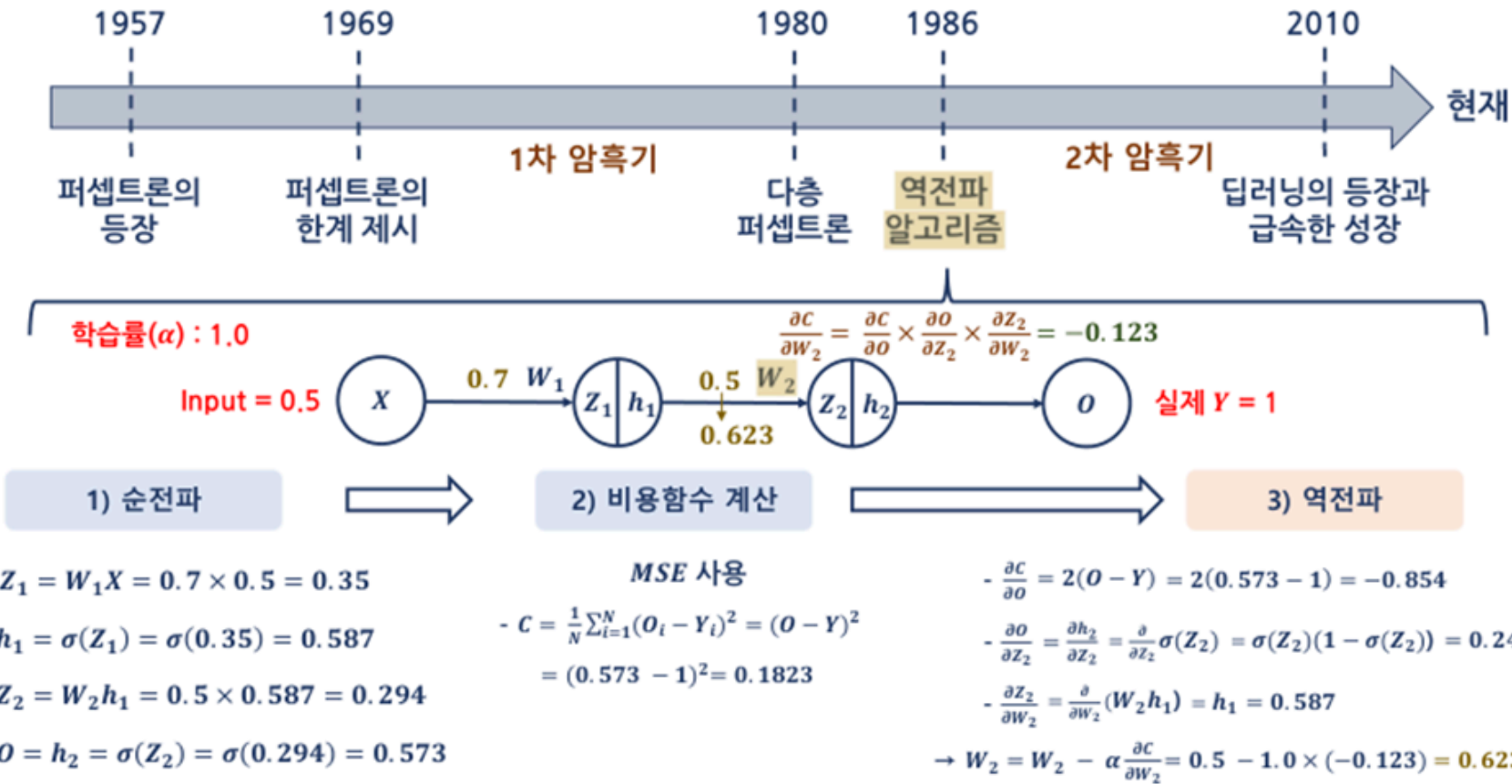
- $Z_1 = W_1 X = 0.7 \times 0.5 = 0.35$
- $h_1 = \sigma(Z_1) = \sigma(0.35) = 0.587$
- $Z_2 = W_2 h_1 = 0.5 \times 0.587 = 0.294$
- $O = h_2 = \sigma(Z_2) = \sigma(0.294) = 0.573$

MSE 사용

$$\begin{aligned}
 - C &= \frac{1}{N} \sum_{i=1}^N (O_i - Y_i)^2 = (O - Y)^2 \\
 &= (0.573 - 1)^2 = 0.1823
 \end{aligned}$$



## 역전파 알고리즘



$$- Z_1 = W_1 X = 0.7 \times 0.5 = 0.35$$

$$- h_1 = \sigma(Z_1) = \sigma(0.35) = 0.587$$

$$- Z_2 = W_2 h_1 = 0.5 \times 0.587 = 0.294$$

$$- O = h_2 = \sigma(Z_2) = \sigma(0.294) = 0.573$$

MSE 사용

$$- C = \frac{1}{N} \sum_{i=1}^N (O_i - Y_i)^2 = (O - Y)^2$$

$$= (0.573 - 1)^2 = 0.1823$$

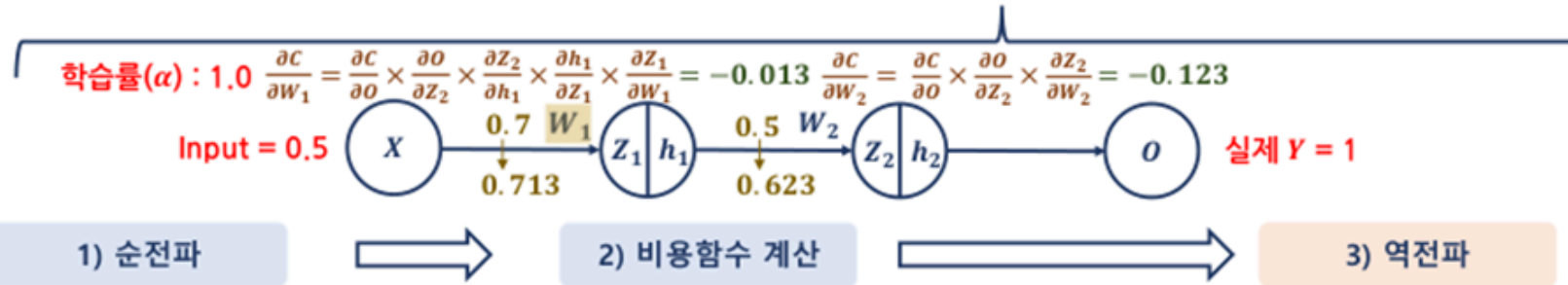
$$- \frac{\partial C}{\partial O} = 2(O - Y) = 2(0.573 - 1) = -0.854$$

$$- \frac{\partial O}{\partial Z_2} = \frac{\partial h_2}{\partial Z_2} = \frac{\partial}{\partial Z_2} \sigma(Z_2) = \sigma(Z_2)(1 - \sigma(Z_2)) = 0.245$$

$$- \frac{\partial Z_2}{\partial W_2} = \frac{\partial}{\partial W_2} (W_2 h_1) = h_1 = 0.587$$

$$\rightarrow W_2 = W_2 - \alpha \frac{\partial C}{\partial W_2} = 0.5 - 1.0 \times (-0.123) = 0.623$$

## 역전파 알고리즘



- $Z_1 = W_1 X = 0.7 \times 0.5 = 0.35$
- $h_1 = \sigma(Z_1) = \sigma(0.35) = 0.587$
- $Z_2 = W_2 h_1 = 0.5 \times 0.587 = 0.294$
- $O = h_2 = \sigma(Z_2) = \sigma(0.294) = 0.573$

MSE 사용

$$C = \frac{1}{N} \sum_{i=1}^N (O_i - Y_i)^2 = (O - Y)^2$$

$$= (0.573 - 1)^2 = 0.1823$$

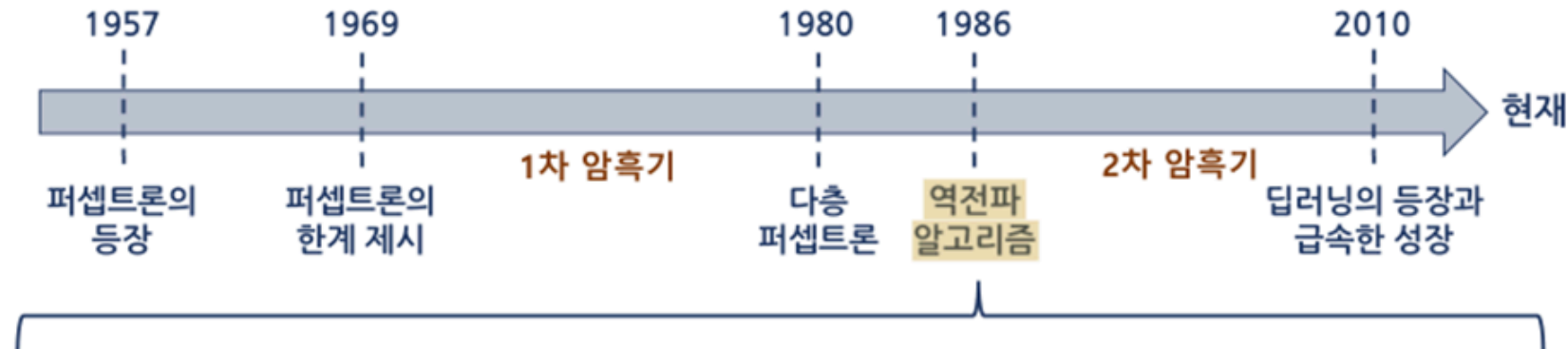
$$-\frac{\partial Z_2}{\partial h_1} = \frac{\partial}{\partial h_1} (W_2 h_1) = W_2 = 0.5$$

$$-\frac{\partial h_1}{\partial Z_1} = \frac{\partial}{\partial Z_1} \sigma(Z_1) = \sigma(Z_1)(1 - \sigma(Z_1)) = 0.242$$

$$-\frac{\partial Z_1}{\partial W_1} = \frac{\partial}{\partial W_1} (W_1 X) = X = 0.5$$

$$\rightarrow W_2 = W_2 - \alpha \frac{\partial C}{\partial W_2} = 0.7 - 1.0 \times (-0.013) = 0.713$$

## 역전파 알고리즘



1) 순전파

2) 비용함수 계산

3) 역전파

$$- Z_1 = W_1 X = 0.713 \times 0.5 = 0.357$$

$$- h_1 = \sigma(Z_1) = \sigma(0.357) = 0.588$$

$$- Z_2 = W_2 h_1 = 0.623 \times 0.588 = 0.366$$

$$- O = h_2 = \sigma(Z_2) = \sigma(0.366) = 0.59$$

- 이전 비용함수 : 0.1823

$$- C = \frac{1}{N} \sum_{i=1}^N (O_i - Y_i)^2 = (O - Y)^2 = (0.59 - 1)^2 = 0.1681$$

비용함수가 감소

비용함수의 최솟값 찾을때 까지 역전파 알고리즘 반복

### 순전파와 역전파 예제

- 순전파(Feedforward)를 통해 최종 출력값과 실제값의 오차를 확인하고 역전파(Backpropagation)를 통해 최종 출력값과 실제값의 오차가 최소화 되도록 가중치(Weight)와 바이어스(Bias)를 업데이트하며 진행

- 예를 들어, 농구에서 자유투를 던지는 과정은 순전파(Feedforward)라고 할 수 있고, 던진 공이 어느 지점에 도착했는지를 확인하고 던질 위치를 수정하는 과정을 역전파(Backpropagation)이라고 할 수 있습니다.

- 예측값 y1 은 0.609 이지만, 실제값은 0.3 이고, y2 는 0.633 이지만, 실제값은 0.9 인 것을 확인
- Error 측정 방식은 MSE 방식을 사용했고, 전체 Error(E\_tot)는 0.083385(=(0.3-0.609)^2+(0.9-0.633)^2)/2=(0.095481+0.071289)/2)인 것을 알 수 있음

$$E_{tot} = \frac{1}{2} \sum (target - output)^2$$

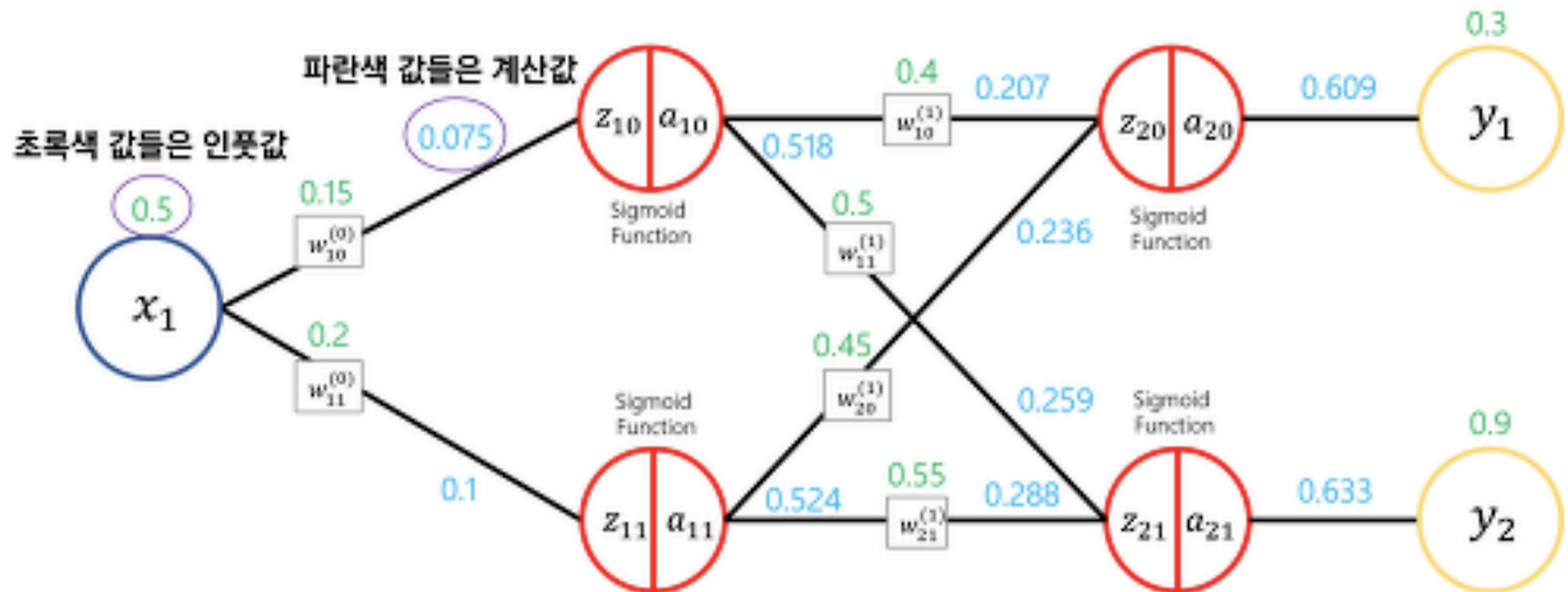
## ✓ 가중치 업데이트 공식

- gradient는 오차를 해당 가중치로 미분한 값이며, 해당 가중치가 오차에 미치는 영향을 의미
- 따라서, 오차에 큰 영향을 주는 가중치일수록, gradient가 커지게 됨

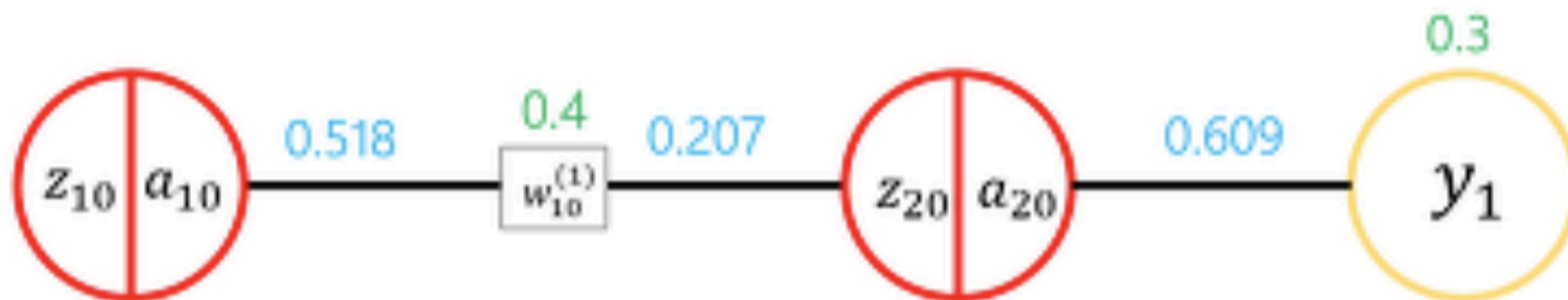
$$w^+ = w - \eta * \frac{\partial E}{\partial w}$$

learning rate :  
한번에 얼마나 학습할지

gradient :  
어떤 방향으로 학습할지



- $w_{10}(1)$ 이 전체  $\text{Error}(E_{\text{tot}})$ 에 미치는 영향부터 계산



- $w_{10}(1)$  이  $E_{tot}$  에 미치는 영향

$$\frac{\partial E_{tot}}{\partial w_{10}^{(1)}} = \frac{\partial E_{tot}}{\partial a_{20}} \frac{\partial a_{20}}{\partial z_{20}} \frac{\partial z_{20}}{\partial w_{10}^{(1)}}$$

- $a_{20}$  이  $E_{tot}$  에 미치는 영향

$$E_{tot} = \frac{1}{2} \left( (target_{y1} - a_{20})^2 + (target_{y2} - a_{21})^2 \right)$$

$$\frac{\partial E_{tot}}{\partial a_{20}} = (target_{y1} - a_{20}) * -1 + 0$$

- $(0.3 - 0.609) * -1$

---

```

1 class Function1:
2     def forward(self, x):
3         z = x - 2
4         return z
5
6     def backward(self, dy_dz):
7         self.dy_dx = 1
8         self.dy_dx *= dy_dz # chain rule

```


```
9         return self.dy_dx
10
11 class Function2:
12     def forward(self, z):
13         self.y = 2*(z**2)
14         return self.y
15
16     def backward(self):
17         return self.y * 4
18
19 class FunctionUnion:
20     def __init__(self):
21         print(f'f(x) = 2*(x - 2)**2\n')
22         self.f1 = Function1()
23         self.f2 = Function2()
24
25     def __call__(self, x):
26         self.x = x
27         self.forward(self.x)
28         self.backward()
29         self.print()
30
31     def forward(self, x):
32         z = self.f1.forward(x)
33         self.y = self.f2.forward(z)
34         return self.y
35
36     def backward(self):
37         dy_dz = self.f2.backward()
38         self.dy_dx = self.f1.backward(dy_dz)
39         return self.dy_dx
40
41     def print(self):
42         print(f'---- x = {self.x} ----')
43         print(f'순전파: f({self.x}) = {self.y}')
44         print(f'역전파: f'({self.x}) = {self.dy_dx}\n')
45
```



```

46 f = FunctionUnion()
47
48 print(f'순전파 : {f.forward(4)}')
49 print(f'역전파 : {f.backward()}')
50
51
52 X = [i for i in range(-1, 4)]
53 print(f'x = {X}')
54 for x in range(len(X)):
55     f(x)

```

  $f(x) = 2*(x - 2)**2$

```

순전파 : 8
역전파 : 32
x = [-1, 0, 1, 2, 3]
---- x = 0 ----
순전파: f(0) = 8
역전파:f'(0) = 32

---- x = 1 ----
순전파: f(1) = 2
역전파:f'(1) = 8

---- x = 2 ----
순전파: f(2) = 0
역전파:f'(2) = 0

---- x = 3 ----
순전파: f(3) = 2
역전파:f'(3) = 8

---- x = 4 ----
순전파: f(4) = 8
역전파:f'(4) = 32

```

## ✓ Backpropagation Implementation in Python for XOR Problem

```
1 #Defining Neural Network
2 #Input layer with 2 inputs
3 #Hidden layer with 4 neurons
4 #Output layer with 1 output neuron
5 #Using Sigmoid function as activation function
6
7 import numpy as np
8
9 class NeuralNetwork:
10     def __init__(self, input_size, hidden_size, output_size):
11         self.input_size = input_size
12         self.hidden_size = hidden_size
13         self.output_size = output_size
14
15         self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
16         self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
17
18         self.bias_hidden = np.zeros((1, self.hidden_size))
19         self.bias_output = np.zeros((1, self.output_size))
20
21     def sigmoid(self, x):
22         return 1 / (1 + np.exp(-x))
23
24     def sigmoid_derivative(self, x):
25         return x * (1 - x)
26
27     # Defining Feed Forward Network
28     def feedforward(self, X):
29         self.hidden_activation = np.dot(X, self.weights_input_hidden) + self.bias_hidden
30         self.hidden_output = self.sigmoid(self.hidden_activation)
31
32         self.output_activation = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
33         self.predicted_output = self.sigmoid(self.output_activation)
```

```

34
35         return self.predicted_output
36
37         #Defining Backward Network
38     def backward(self, X, y, learning_rate):
39         output_error = y - self.predicted_output
40         output_delta = output_error * self.sigmoid_derivative(self.predicted_output)
41
42         hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
43         hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)
44
45         self.weights_hidden_output += np.dot(self.hidden_output.T, output_delta) * learning_rate
46         self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
47         self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate
48         self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
49
50         # Training Network
51     def train(self, X, y, epochs, learning_rate):
52         for epoch in range(epochs):
53             output = self.feedforward(X)
54             self.backward(X, y, learning_rate)
55             if epoch % 4000 == 0:
56                 loss = np.mean(np.square(y - output))
57                 print(f"Epoch {epoch}, Loss:{loss}")

```

```

1 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
2 y = np.array([[0], [1], [1], [0]])
3
4 nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)
5 nn.train(X, y, epochs=10000, learning_rate=0.1)
6
7 output = nn.feedforward(X)
8 print("Predictions after training:")
9 print(output)
10 #The final predictions are close to the expected XOR outputs: approximately 0 for [0, 0] and [1, 1]
11 #and approximately 1 for [0, 1] and [1, 0] indicating that the network successfully learned to approximate the XOR function.

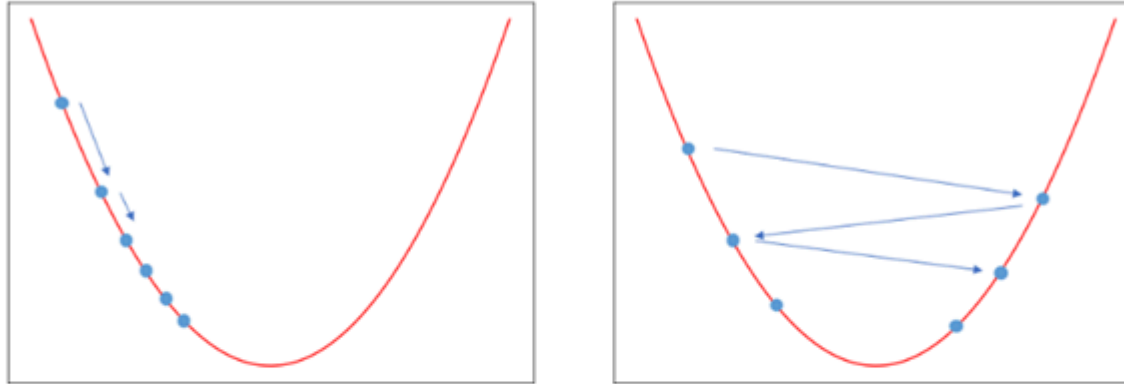
```

```
Epoch 0, Loss:0.3349915309471196  
Epoch 4000, Loss:0.00972100515524983  
Epoch 8000, Loss:0.002648291357954204  
Predictions after training:  
[[0.03831387]  
 [0.94726826]  
 [0.97071689]  
 [0.04935671]]
```

---

## ✓ 경사하강법

- 딥러닝에서 모델을 학습한다는 것은?
  - 실제 값과 예측 값의 오차를 최소화하는 가중치를 찾는 과정
- 비용 함수(Cost function)
  - '오차'를 정의하는 함수
  - 비용 함수가 최솟값을 갖는 방향으로 가중치를 업데이트하는 작업이 필요
- 경사 하강법(Gradient Descent)
  - 최솟값을 찾을 때 사용할 수 있는 최적화 알고리즘
  - 먼저, 최솟값을 찾을 함수를 설정한 후, 임의의 값으로 초기화하고 해당 값의 기울기를 빼면서 최솟값에 가까워질 때까지 반복하는 방법



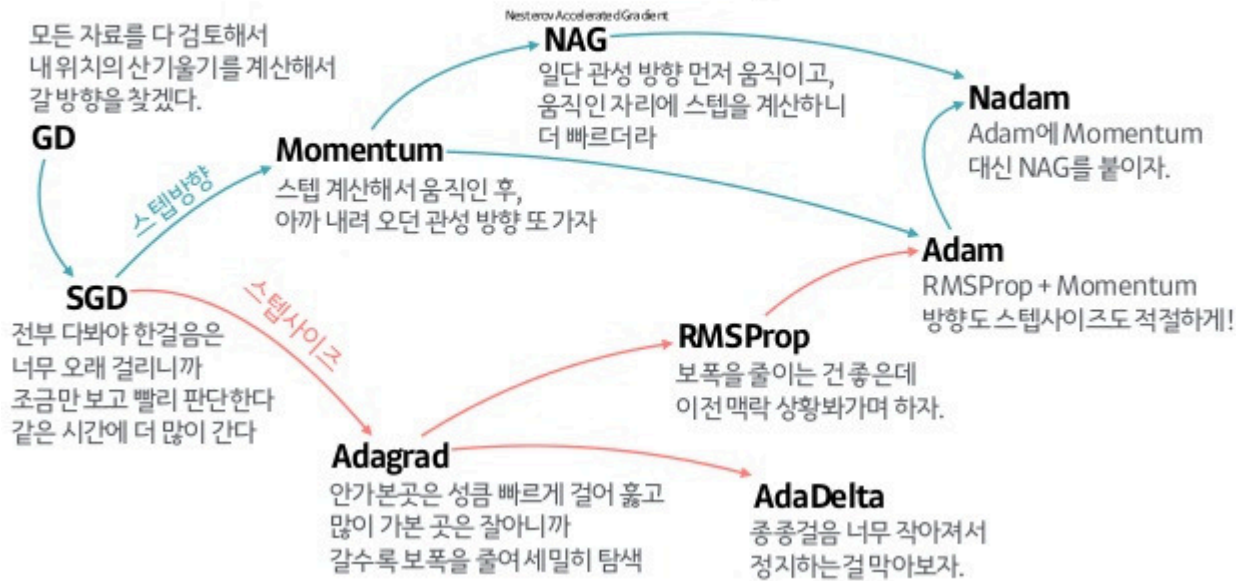
t 시점의 w에 대해 편미분한 값을 빼서 t+1 시점의 w를 구하는 과정을 식으로 나타낸 것

$$w^{t+1} = w^t - \mu \frac{dE(w)}{dw}$$

- E(w)는 오차를 계산하는 비용 함수(목적 함수)
- w는 오차를 구하는 과정에서 사용된 가중치
- 미분 값 앞에 곱해져 있는 상수는 학습률(learning rate)
- 학습률이 작으면 왼쪽 그래프처럼 값이 천천히 변하고, 학습률이 크면 오른쪽 그래프처럼 큰 보폭으로 움직임

## ▽ 경사 하강법 종류

# 산 내려오는 작은 오솔길 찾기(Optimizer)의 발달 계보



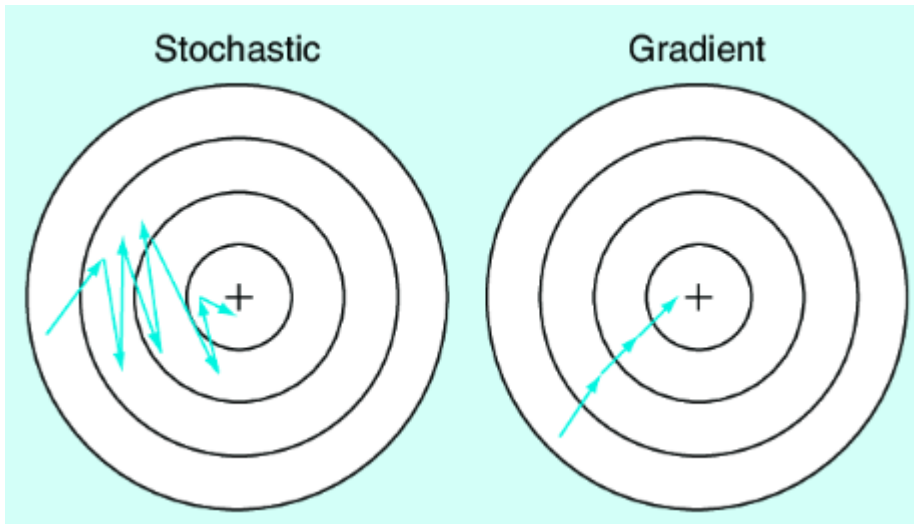
	경사하강법	확률적 경사하강법	미니배치 경사하강법
해를 찾는 과정	Gradient Descent 	Stochastic Gradient Descent 	Mini-Batch Gradient Descent 
1 Iteration에 사용되는 데이터 크기	모든 데이터	임의의 하나의 데이터	설정된 미니배치 사이즈만큼

## ✓ 배치 경사 하강법(Batch Gradient Descent)

- 가장 기본적인 경사 하강법으로 Vanilla Gradient Descent라고 부르기도 함
- 배치 경사 하강법은 데이터셋 전체를 고려하여 손실함수를 계산함
- 배치 경사 하강법은 한 번의 Epoch에 모든 파라미터 업데이트를 단 한 번만 수행함 -> Batch의 개수와 Iteration은 1이고 Batch size는 전체 데이터의 개수임
- 파라미터 업데이트할 때 한 번에 전체 데이터셋을 고려하기 때문에 모델 학습 시 많은 시간과 메모리가 필요하다는 단점이 있음

## ✓ 확률적 경사 하강법(Stochastic Gradient Descent)

- 배치 경사 하강법이 모델 학습 시 많은 시간과 메모리가 필요하다는 단점을 개선하기 위해 제안된 기법
- 확률적 경사 하강법은 Batch size를 1로 설정하여 파라미터를 업데이트하기 때문에 배치 경사 하강법보다 훨씬 빠르고 적은 메모리로 학습이 진행됨
- 확률적 경사 하강법은 파라미터 값의 업데이트 폭이 불안정하기 때문에 배치 경사 하강법보다 정확도가 낮은 경우가 생길 수도 있음
- 단점을 개선하는 모멘텀, AdaGrad, Adam 방법이 있음



## ✓ 미니 배치 경사 하강법(Mini-Batch Gradient Descent)

- Batch size가 1도 전체 데이터 개수도 아닌 경우를 말함
- 미니 배치 경사 하강법은 배치 경사 하강법보다 모델 학습 속도가 빠르고, 확률적 경사 하강법보다 안정적인 장점이 있음
- 딥러닝 분야에서 가장 많이 활용하는 경사 하강법
- Batch size는 어떻게 정하면 좋을까요? 일반적으로 32, 64, 128과 같이 2의 n제곱에 해당하는 값으로 사용하는 게 보편적임

---

```

1 # coding: utf-8
2 import numpy as np
3
4 # 경사법
5 # 현 위치에서 기울어진 방향으로 일정 거리만큼 이동
6 # 그런다음 다시 기울기를 구하고 기울어진 방향으로 이동
7 # 점차 함수의 값을 줄이는 것
8 # 기계 학습을 최적화 하는데 흔히 쓰는 방법.
9
10 # 경사 하강법
11 # init_x : 초기값
12 # lr : learning rate 학습률
13 # step_num 경사법에 따른 반복 횟수
14 # 함수의 기울기에 학습률을 곱한값으로 갱신하는 처리는 step_num번 반복
15 # 학습률과 같은 매개변수를 하이퍼파라미터 라고함
16 # 이는 가중치와 편향같은 신경망의 매개변수와는 성질이 다른 매개변수.
17
18 def gradient_descent(f, init_x, lr=0.01, step_num=100):
19     x = init_x
20     for i in range(step_num):
21         grad = numerucal_gradient(f, x)
22         x -= lr * grad
23     return x

```



```

24
25 def numerucal_gradient(f, x):
26     h = 1e-4 # 0.0001
27     grad = np.zeros_like(x)
28     for idx in range(x.size):
29         tmp_val = x[idx]
30         # f(x+h) 계산
31         x[idx] = tmp_val + h
32         fxh1 = f(x)
33         # f(x-h) 계산
34         x[idx] = tmp_val - h
35         fxh2 = f(x)
36         grad[idx] = (fxh1 - fxh2) / (2 * h)
37         x[idx] = tmp_val
38     return grad

```

```

1 #경사법으로 f(x0,x1) = x0^2+x1^2의 최소값을 구하라. https://blog.naver.com/ssdyka/221300959357
2 def function_2(x):
3     # return x[0]**2 + x[1]**2
4     return np.sum(x ** 2)
5
6 init_x = np.array([-3.0, 4.0])
7 res = gradient_descent(function_2, init_x=init_x, lr=0.1, step_num=100)
8 print(res)

```

```

⇒ (array([-6.11110793e-10,  8.14814391e-10]), array([[ -3.00000000e+00,  4.00000000e+00],
          [-2.40000000e+00,  3.20000000e+00],
          [-1.92000000e+00,  2.56000000e+00],
          [-1.53600000e+00,  2.04800000e+00],
          [-1.22880000e+00,  1.63840000e+00],
          [-9.83040000e-01,  1.31072000e+00],
          [-7.86432000e-01,  1.04857600e+00],
          [-6.29145600e-01,  8.38860800e-01],
          [-5.03316480e-01,  6.71088640e-01],
          [-4.02653184e-01,  5.36870912e-01],
          [-3.22122547e-01,  4.29496730e-01],
          [-2.57698038e-01,  3.43597384e-01],
          [-2.06158430e-01,  2.74877907e-01],

```

```
[ -1.64926744e-01,  2.19902326e-01 ],  
[ -1.31941395e-01,  1.75921860e-01 ],  
[ -1.05553116e-01,  1.40737488e-01 ],  
[ -8.44424930e-02,  1.12589991e-01 ],  
[ -6.75539944e-02,  9.00719925e-02 ],  
[ -5.40431955e-02,  7.20575940e-02 ],  
[ -4.32345564e-02,  5.76460752e-02 ],  
[ -3.45876451e-02,  4.61168602e-02 ],  
[ -2.76701161e-02,  3.68934881e-02 ],  
[ -2.21360929e-02,  2.95147905e-02 ],  
[ -1.77088743e-02,  2.36118324e-02 ],  
[ -1.41670994e-02,  1.88894659e-02 ],  
[ -1.13336796e-02,  1.51115727e-02 ],  
[ -9.06694365e-03,  1.20892582e-02 ],  
[ -7.25355492e-03,  9.67140656e-03 ],  
[ -5.80284393e-03,  7.73712525e-03 ],  
[ -4.64227515e-03,  6.18970020e-03 ],  
[ -3.71382012e-03,  4.95176016e-03 ],  
[ -2.97105609e-03,  3.96140813e-03 ],  
[ -2.37684488e-03,  3.16912650e-03 ],  
[ -1.90147590e-03,  2.53530120e-03 ],  
[ -1.52118072e-03,  2.02824096e-03 ],  
[ -1.21694458e-03,  1.62259277e-03 ],  
[ -9.73555661e-04,  1.29807421e-03 ],  
[ -7.78844529e-04,  1.03845937e-03 ],  
[ -6.23075623e-04,  8.30767497e-04 ],  
[ -4.98460498e-04,  6.64613998e-04 ],  
[ -3.98768399e-04,  5.31691198e-04 ],  
[ -3.19014719e-04,  4.25352959e-04 ],  
[ -2.55211775e-04,  3.40282367e-04 ],  
[ -2.04169420e-04,  2.72225894e-04 ],  
[ -1.63335536e-04,  2.17780715e-04 ],  
[ -1.30668429e-04,  1.74224572e-04 ],  
[ -1.04534743e-04,  1.39379657e-04 ],  
[ -8.36277945e-05,  1.11503726e-04 ],  
[ -6.69022356e-05,  8.92029808e-05 ],  
[ -5.35217885e-05,  7.13623846e-05 ],  
[ -4.28174308e-05,  5.70899077e-05 ],  
[ -3.42539446e-05,  4.56719262e-05 ],  
[ -2.74031557e-05,  3.65375409e-05 ],  
[ -2.19225246e-05,  2.92300327e-05 ],
```

```

[-1.75380196e-05,  2.33840262e-05],
[-1.40304157e-05,  1.87072210e-05],
[-1.12243326e-05,  1.49657768e-05],
[ 0.07046606e-05,  1.10796214e-05]

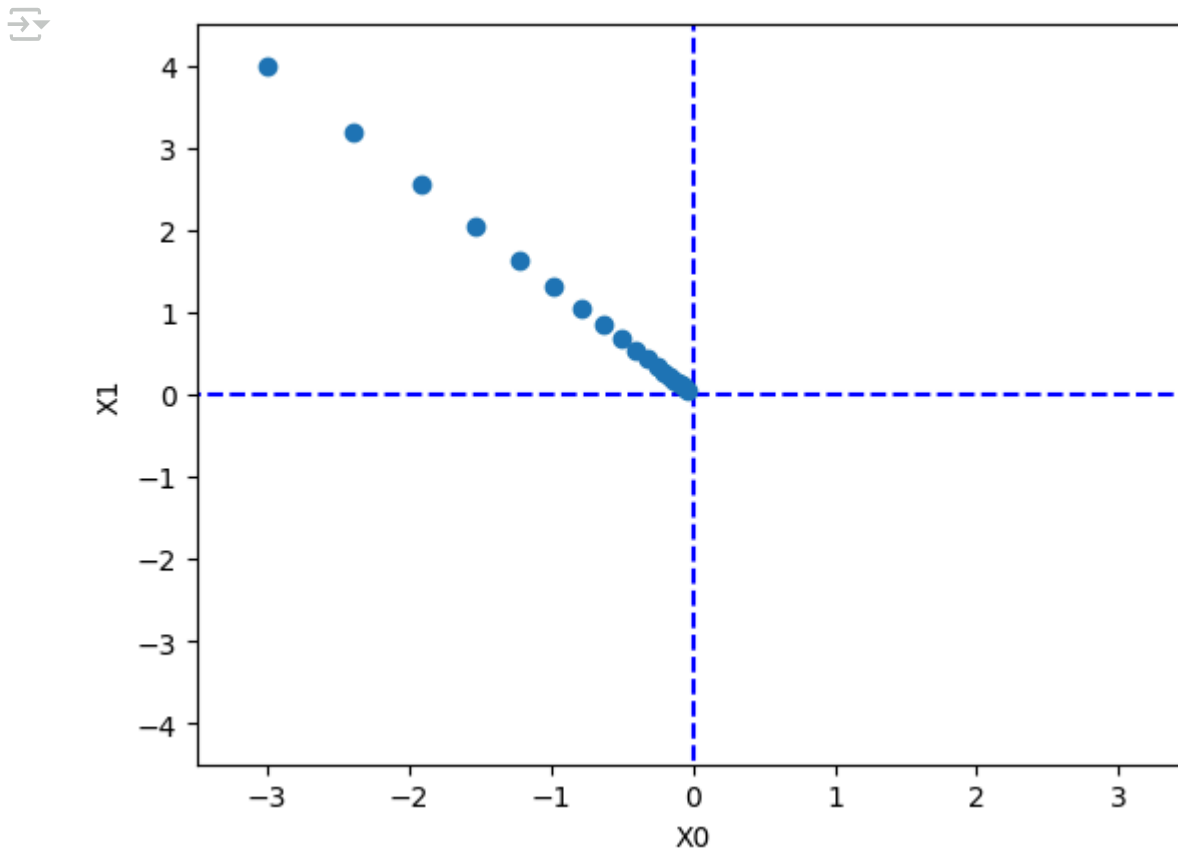
```

```

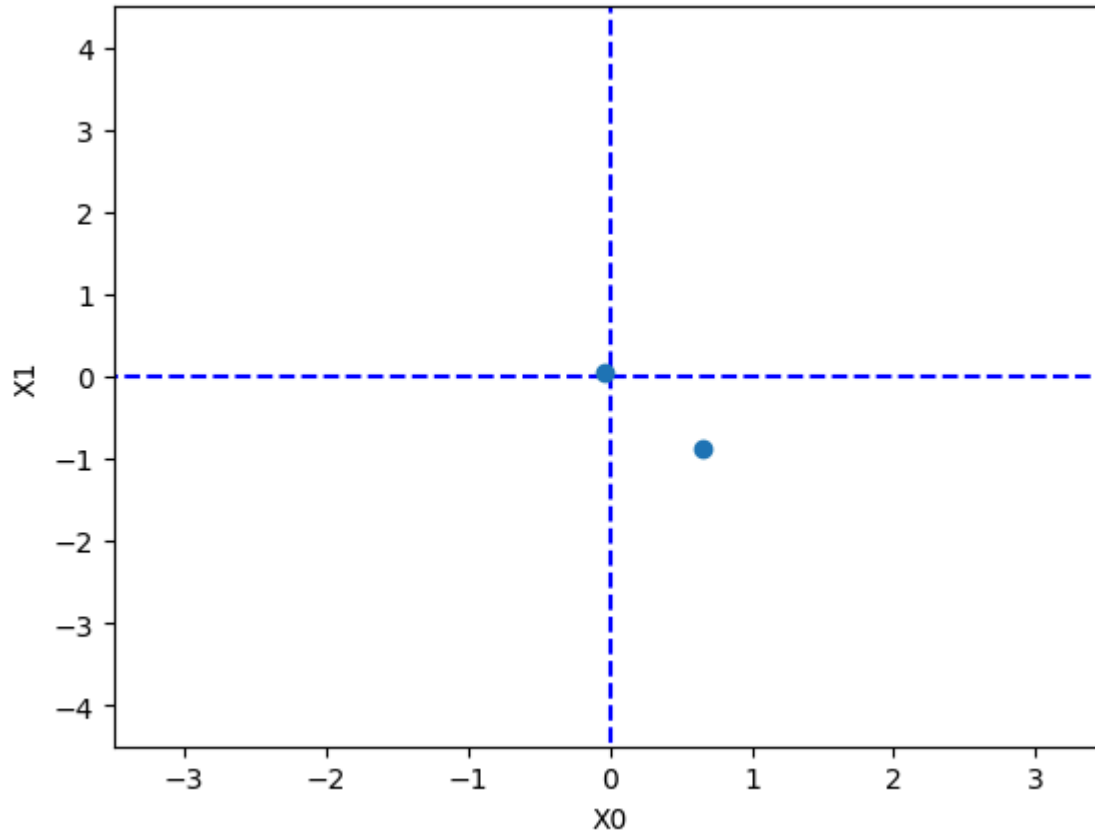
1 #갱신 과정
2 # coding: utf-8
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 def _numerical_gradient_no_batch(f, x):
7     h = 1e-4 # 0.0001
8     grad = np.zeros_like(x)
9     for idx in range(x.size):
10         tmp_val = x[idx]
11         x[idx] = float(tmp_val) + h
12         fxh1 = f(x) # f(x+h)
13         x[idx] = tmp_val - h
14         fxh2 = f(x) # f(x-h)
15         grad[idx] = (fxh1 - fxh2) / (2*h)
16         x[idx] = tmp_val
17     return grad
18
19 def gradient_descent(f, init_x, lr=0.01, step_num=100):
20     x = init_x
21     x_history = []
22     for i in range(step_num):
23         x_history.append( x.copy() )
24         grad = _numerical_gradient_no_batch(f, x)
25         x -= lr * grad
26     return x, np.array(x_history)
27
28 def function_2(x):
29     return x[0]**2 + x[1]**2
30

```

```
31 init_x = np.array([-3.0, 4.0])
32 lr = 0.1
33 step_num = 20
34 x, x_history = gradient_descent(function_2, init_x, lr=lr, step_num=step_num)
35 plt.plot([-5, 5], [0,0], '--b')
36 plt.plot([0,0], [-5, 5], '--b')
37 plt.plot(x_history[:,0], x_history[:,1], 'o')
38 plt.xlim(-3.5, 3.5)
39 plt.ylim(-4.5, 4.5)
40 plt.xlabel("X0")
41 plt.ylabel("X1")
42 plt.show()
```



```
1 ##### 학습률이 너무 큰 예
2 # 학습률이 크면 발산함
3 lr = 10
4 step_num = 20
5 x, x_history = gradient_descent(function_2, init_x, lr=lr, step_num=step_num)
6 plt.plot([-5, 5], [0,0], '--b')
7 plt.plot([0,0], [-5, 5], '--b')
8 plt.plot(x_history[:,0], x_history[:,1], 'o')
9 plt.xlim(-3.5, 3.5)
10 plt.ylim(-4.5, 4.5)
11 plt.xlabel("X0")
12 plt.ylabel("X1")
13 plt.show()
```



```
1 ##### 학습률이 너무 작은 예
2 # 값이 너무 작으면 거의 갱신되지 않은채 끝남
3 lr =0.1
4 step_num = 1
5 x, x_history = gradient_descent(function_2, init_x, lr=lr, step_num=step_num)
6 plt.plot( [-5, 5], [0,0], '--b')
7 plt.plot( [0,0], [-5, 5], '--b')
8 plt.plot(x_history[:,0], x_history[:,1], 'o')
9 plt.xlim(-3.5, 3.5)
10 plt.ylim(-4.5, 4.5)
11 plt.xlabel("X0")
12 plt.ylabel("X1")
13 plt.show()
```

