

신경망 (Neural Network)

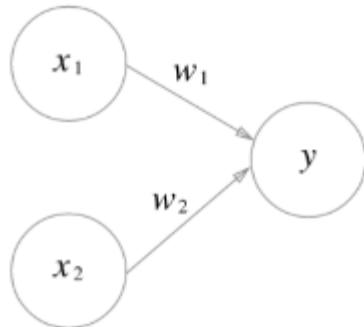
퍼셉트론

퍼셉트론이란

- 퍼셉트론의 시작
 - 프랑크 로젠클라트가 1957년 고안한 알고리즘
 - 신경망(딥러닝)의 기원이 되는 알고리즘
- 퍼셉트론이란?
 - 퍼셉트론은 다수의 신호를 입력으로 받아 하나의 신호를 출력
 - 신호란 전류나 강물처럼 흐름이 있는 것으로 이해하면 좋음
 - 전류가 전선을 타고 흐르는 전자를 내보내듯 퍼셉트론 신호도 흐름을 만들고 정보를 앞으로 전달
 - 다만 실제 전류와 달리 퍼셉트론 신호는 '흐른다/안 흐른다(1이나 0)'의 두 가지 값

퍼셉트론이란

- 퍼셉트론이란?
 - 입력으로 2개의 신호를 받은 퍼셉트론의 예



- x_1 과 x_2 는 입력 신호, y 는 출력 신호, w_1 과 w_2 는 가중치
- 그림의 원을 뉴런 혹은 노드라고 부름
- 입력 신호가 뉴런에 보내질 때는 각각 고유한 가중치가 곱해짐
- 뉴런에서 보내온 신호의 총합이 정해진 한계를 넘어설 때만 1을 출력
- 이를 '뉴런이 활성화한다'라 표현하기도 함
- 이 때 한계를 임계값이라 하며 θ 기호로 나타냄

퍼셉트론이란

- 퍼셉트론이란?

$$y = \begin{cases} 0 & (w_1x_1 + w_2x_2 \leq \theta) \\ 1 & (w_1x_1 + w_2x_2 > \theta) \end{cases}$$

- 퍼셉트론의 복수의 입력 신호 각각에 고유한 가중치를 부여
- 가중치는 각 신호가 결과에 주는 영향력을 조절하는 요소로 작용
- 즉, 가중치가 클수록 해당 신호가 그만큼 더 중요함을 뜻함

- 참고

- 가중치는 전류에서 말하는 저항에 해당
- 저항은 전류의 흐름을 억제하는 매개변수로, 저항이 낮을수록 큰 전류가 흐름
- 한편 퍼셉트론의 가중치는 그 값이 클수록 강한 신호를 흘려보냄
- 저항과 가중치는 서로 작용하는 방향은 반대지만, 신호가 얼마나 잘(혹은 어렵게) 흐르는가를 통제한다는 점에서 저항과 가중치는 같은 기능을 함

단순한 논리 회로

- AND 게이트
 - AND 게이트는 입력이 둘이고 출력은 하나
 - AND 게이트의 진리표

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

- 입력이 모두 1일 때만 1을 출력하고 그 외에는 0을 출력
- AND 게이트를 퍼셉트론으로 표현하려면 위의 진리표대로 작동하는 w_1, w_2, θ 의 값을 정해야함
- 이를 만족하는 (w_1, w_2, θ) 의 조합은 무수히 많음
- 가령 $(0.5, 0.5, 0.7), (0.5, 0.5, 0.8), (1.0, 1.0, 1.0)$ 모두가 AND 게이트 조건을 만족

단순한 논리 회로

- NAND 게이트
 - NAND는 Not AND를 의미하며, 그 동작은 AND 게이트의 출력을 뒤집은 것이 됨
 - NAND 게이트의 진리표

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

- x_1 과 x_2 가 모두 1일 때만 0을 출력하고, 그 외에는 1을 출력
- NAND 게이트를 만족하는 (w_1, w_2, θ) 의 조합의 예 $(-0.5, -0.5, -0.7)$
- AND 게이트를 구현하는 매개변수의 부호를 모두 반전하기만 하면 NAND 게이트가 됨

단순한 논리 회로

- OR 게이트
 - OR 게이트는 입력 신호 중 하나 이상이 1이면 출력이 1이 되는 논리 회로
 - OR 게이트의 진리표

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

- OR 게이트를 만족하는 (w_1, w_2, θ)의 조합 의 예 (0.5, 0.5, 0.2)

- 퍼셉트론 요약
 - 퍼셉트론으로 AND, NAND, OR 논리 회로를 표현할 수 있음
 - 퍼셉트론의 구조는 AND, NAND, OR 게이트 모두에서 똑같음
 - 세가지 게이트에서 다른 것은 매개변수(가중치와 임계값)의 값 뿐임

퍼셉트론 구현하기

- 간단한 구현부터
 - x_1 과 x_2 를 인수로 받는 AND 함수

```
def AND(x1, x2):  
    w1, w2, theta = 0.5, 0.5, 0.7  
    tmp = x1*w1 + x2*w2  
    if tmp <= theta:  
        return 0  
    elif tmp > theta:  
        return 1
```

AND(0,1) #0을 출력

AND(1,0) #0을 출력

AND(0,1) #0을 출력

AND(1,1) #1을 출력

퍼셉트론 구현하기

- 가중치와 편향 도입

$$y = \begin{cases} 0 & (w_1x_1 + w_2x_2 \leq \theta) \\ 1 & (w_1x_1 + w_2x_2 > \theta) \end{cases}$$

- 위의 식에서 θ 를 $-b$ 로 치환하면 퍼셉트론 동작이 아래와 같이 됨

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$

- 여기서 b 를 편향(bias)이라 하며 w_1 과 w_2 는 그대로 가중치
- 즉, 퍼셉트론은 입력 신호에 가중치를 곱한 값과 편향을 합하여 그 값이 0을 넘으면 1을 출력하고 그렇지 않으면 0을 출력

퍼셉트론 구현하기

- 가중치와 편향 구현

```
import numpy as np

def AND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.7
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1
```

- w_1 과 w_2 는 각 입력 신호가 결과에 주는 영향력(중요도)을 조절하는 매개변수고, 편향을 뉴런이 얼마나 쉽게 활성화(결과로 1 출력) 하느냐를 조정하는 매개변수
- 예를 들어 b 가 -0.1이면 각 입력 신호에 가중치를 곱한 값들의 합이 0.1을 초과할 때만 뉴런이 활성화. 반면 b 가 -20이면 각 입력 신호에 가중치를 곱한 값들의 합이 20.0을 넘지 않으면 뉴런은 활성화하지 않음
- 즉, 편향의 값은 뉴런이 얼마나 쉽게 활성화되는지를 결정

퍼셉트론 구현하기

- 가중치와 편향 구현

```
def NAND(x1, x2):  
    x = np.array([x1, x2])  
    w = np.array([-0.5, -0.5])  
    b = 0.7  
    tmp = np.sum(w*x) + b  
    if tmp <= 0:  
        return 0  
    else:  
        return 1
```

```
def OR(x1, x2):  
    x = np.array([x1, x2])  
    w = np.array([0.5, 0.5])  
    b = -0.2  
    tmp = np.sum(w*x) + b  
    if tmp <= 0:  
        return 0  
    else:  
        return 1
```

- AND, NAND, OR 는 모두 같은 구조의 퍼셉트론이고, 차이는 가중치 매개변수의 값 뿐임. 실제 구현을 통해 확인해 볼 수 있었음!

퍼셉트론의 한계

- XOR 게이트 도전
 - 배타적 논리합이라는 회로
 - x_1 과 x_2 중 한쪽이 1일 때만 1을 출력
 - XOR 게이트의 진리표

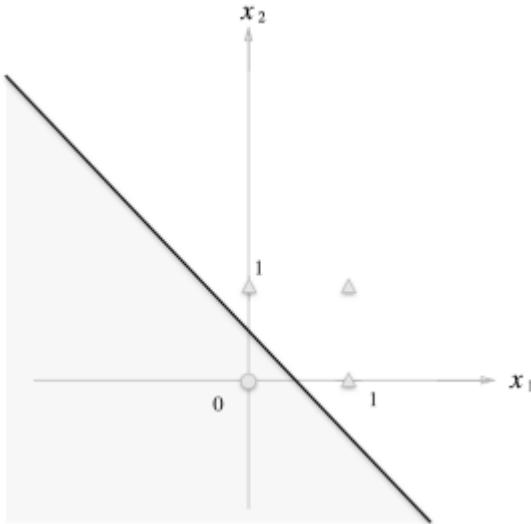
x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0

- 그러나 XOR 게이트를 퍼셉트론으로 구현할 수 있는 매개 변수를 찾을 수 없음
- 예를 들어 OR 게이트는 $(b, w_1, w_2) = (-0.5, 1.0, 1.0)$ 일 때 진리표를 만족
- 이 때의 퍼셉트론 식

$$y = \begin{cases} 0 & (-0.5 + x_1 + x_2 \leq 0) \\ 1 & (-0.5 + x_1 + x_2 > 0) \end{cases}$$

퍼셉트론의 한계

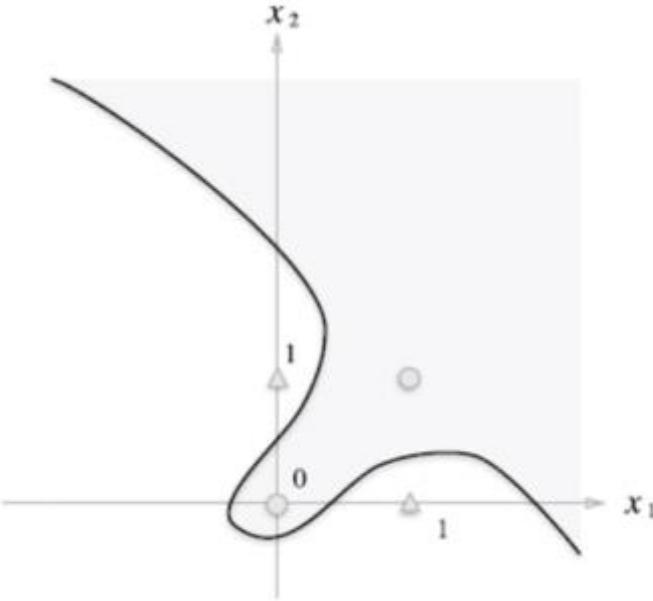
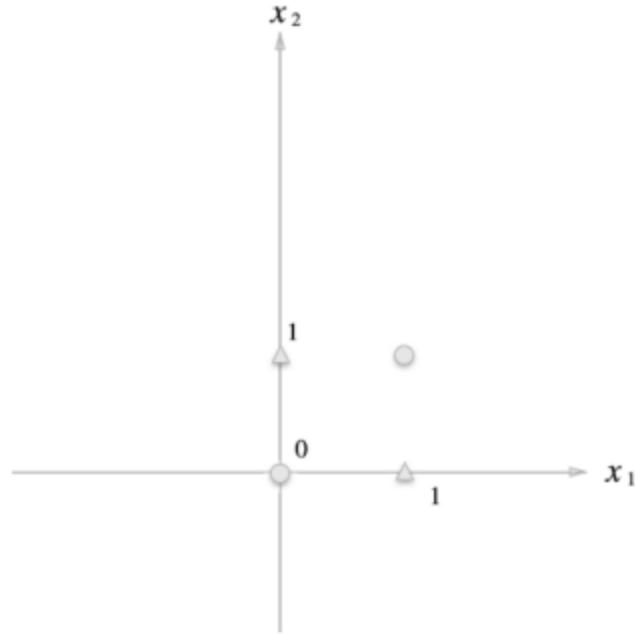
- XOR 게이트 도전
 - 이 때의 퍼셉트론은 직선으로 나눈 두 영역을 만들며, 직선으로 나눈 한쪽 영역은 1을 출력하고 다른 한쪽은 0을 출력



- OR 게이트는 $(x_1, x_2) = (0, 0)$ 일 때 0을 출력하고 $(0, 1), (1, 0), (1, 1)$ 일 때는 1을 출력
- 그림에서는 0을 원, 1을 삼각형으로 표시
- OR 게이트를 만들려면 원과 삼각형을 직선으로 나눠야 함
- 위 그림의 직선을 네 점을 제대로 나누고 있음

퍼셉트론의 한계

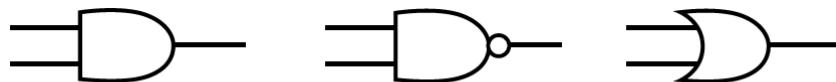
- 선형과 비선형



- XOR 게이트의 경우 원과 삼각형을 나누는 직선을 만드는 것은 불가능
- 그러나 '직선'이라는 제약을 없앴다면 가능
- 즉, 퍼셉트론은 직선 하나로 나눈 영역만 표현할 수 있다는 한계가 있음
- 오른쪽 그림과 같은 곡선 영역을 비선형 영역이라고 부름 (앞의 그림은 직선 영역)

다층 퍼셉트론 등장

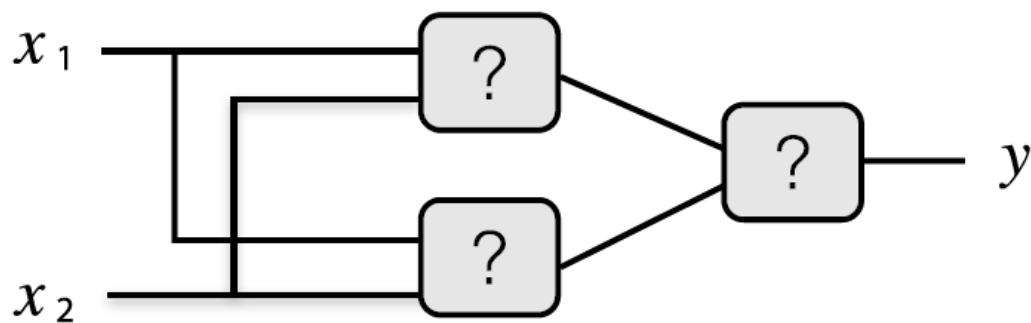
- 다층 퍼셉트론
 - 퍼셉트론은 층을 쌓아 다층 퍼셉트론을 만들 수 있는 장점이 있음
- 기존 게이트 조합하기
 - XOR 게이트를 만드는 방법은 다양
 - AND, NAND, OR 게이트를 조합해 구현한 XOR 게이트



AND

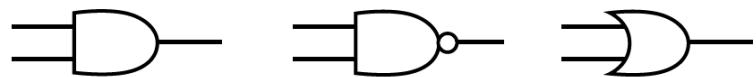
NAND

OR



다층 퍼셉트론 등장

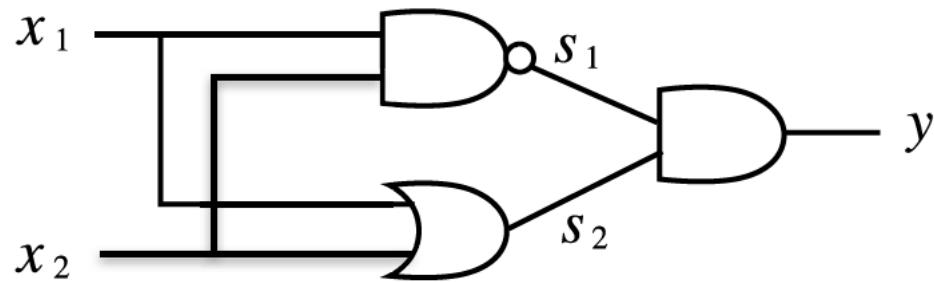
- 다층 퍼셉트론
 - 퍼셉트론은 층을 쌓아 다층 퍼셉트론을 만들 수 있는 장점이 있음
- 기존 게이트 조합하기
 - XOR 게이트를 만드는 방법은 다양
 - AND, NAND, OR 게이트를 조합해 구현한 XOR 게이트



AND

NAND

OR



- x_1 과 x_2 는 NAND와 OR 게이트의 입력이 되고, NAND와 OR의 출력이 AND 게이트의 입력으로 이어짐

다층 퍼셉트론 등장

- 기존 게이트 조합하기
 - XOR 게이트의 진리표

x_1	x_2	s_1	s_2	y
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	1	0

- NAND의 출력을 s_1 , OR의 출력을 s_2 로 해서 진리표를 만들면 위의 그림과 같음

다층 퍼셉트론 등장

- XOR 게이트 구현하기

```
def XOR(x1, x2):  
    s1 = NAND(x1, x2)  
    s2 = OR(x1, x2)  
    y = AND(s1, s2)  
    return y
```

```
XOR(0, 0) # 0을 출력
```

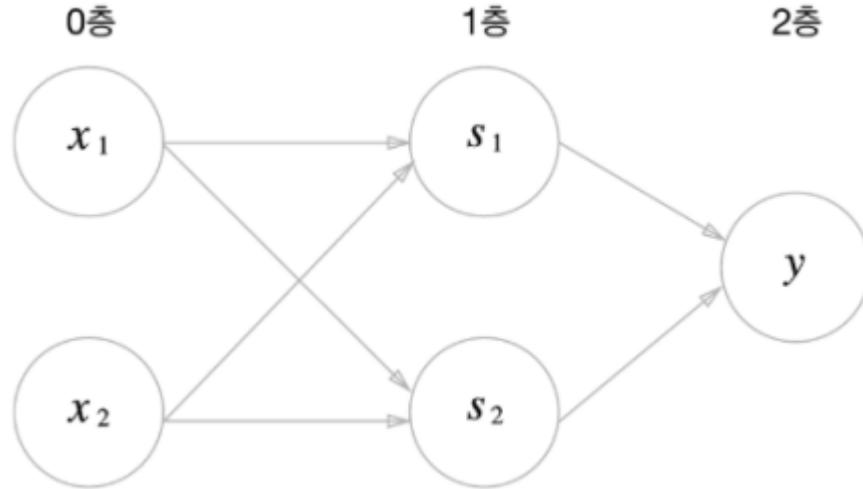
```
XOR(1, 0) # 1을 출력
```

```
XOR(0, 1) # 1을 출력
```

```
XOR(1, 1) # 0을 출력
```

다층 퍼셉트론 등장

- XOR 게이트 구현하기
 - XOR의 퍼셉트론



- 앞서 살펴본 AND, OR 퍼셉트론과 형태가 다름
- AND, OR는 단층 퍼셉트론인 데 반해, XOR는 2층 퍼셉트론임
- 이처럼 층이 여러 개인 퍼셉트론을 다층 퍼셉트론이라 함
- 0층의 두 뉴런이 입력 신호를 받아 1층의 뉴런으로 신호를 보내고,
- 1층의 뉴런이 2층의 뉴런으로 신호를 보내고, 2층의 뉴런은 y 를 출력
- 단층 퍼셉트론으로는 표현하지 못한 것을 층을 하나 늘려 구현할 수 있게 됨

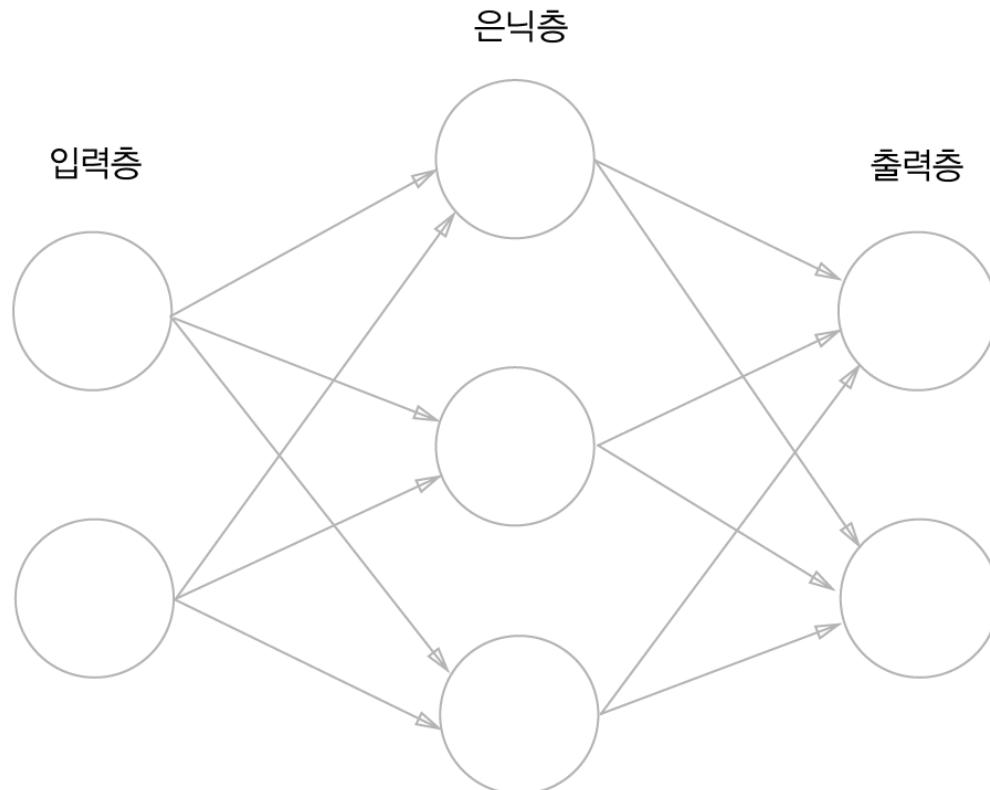
퍼셉트론 정리

- 퍼셉트론에 대해 알게 된 내용
 - 퍼셉트론은 입출력을 갖춘 알고리즘. 입력을 주면 정해진 규칙에 따른 값을 출력
 - 퍼셉트론에서는 '가중치'와 '편향'을 매개변수로 설정
 - 퍼셉트론으로 AND, OR 게이트 등의 논리 회로를 표현할 수 있음
 - XOR 게이트는 단층 퍼셉트론으로는 표현할 수 없음
 - 2층 퍼셉트론을 이용하면 XOR 게이트를 표현할 수 있음
 - 단층 퍼셉트론은 직선형 영역만 표현할 수 있고, 다층 퍼셉트론은 비선형 영역도 표현할 수 있음

신경망

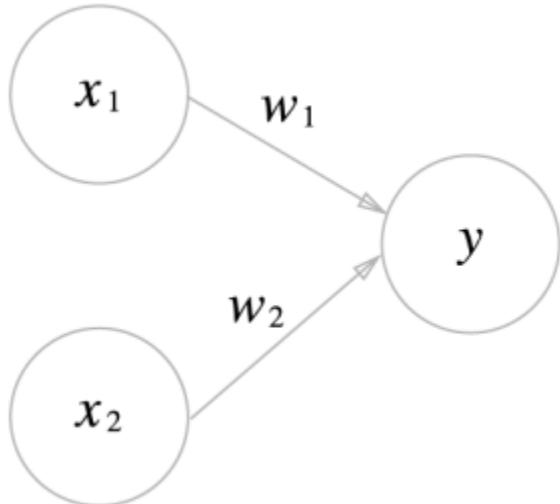
퍼셉트론에서 신경망으로

- 퍼셉트론과 신경망
 - 퍼셉트론은 복잡한 함수도 표현할 수 있음
 - 그러나 가중치를 설정하는 작업은 수동으로 해야 함
 - 신경망은 가중치 값을 데이터로부터 자동으로 학습
- 신경망의 예



퍼셉트론에서 신경망으로

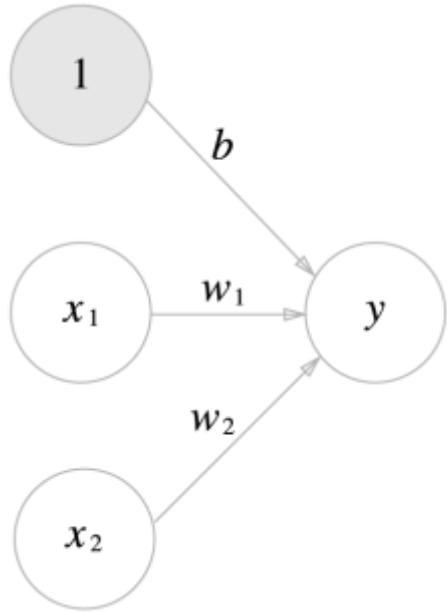
- 퍼셉트론 복습
 - x_1 과 x_2 라는 두 신호를 입력받아 y 를 출력하는 퍼셉트론



- 퍼셉트론 수식
$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$
- b 는 **편향**을 나타내는 매개변수로 뉴런이 얼마나 쉽게 활성화되느냐를 제어
- w_1 과 w_2 는 각 신호의 **가중치**를 나타내는 매개변수로, 각 신호의 영향력을 제어

퍼셉트론에서 신경망으로

- 퍼셉트론 복습
 - 편향을 명시한 퍼셉트론



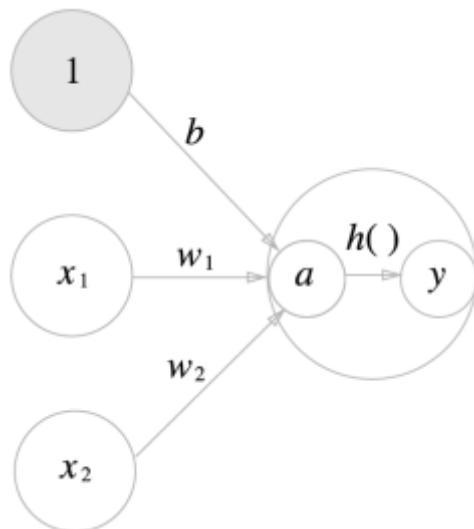
$$y = h(b + w_1x_1 + w_2x_2)$$

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

- 이 퍼셉트론의 동작은 $x_1, x_2, 1$ 이라는 3개의 신호가 뉴런에 입력되어, 각 신호에 가중치를 곱한 후, 다른 뉴런에 전달
- 다른 뉴런에서는 이 신호들의 값을 더하여, 그 합이 0을 넘으면 1을 출력하고 그렇지 않으면 0을 출력

퍼셉트론에서 신경망으로

- 활성화 함수의 등장
 - $h(x)$ 와 같이 입력 신호의 총합을 출력 신호를 변환하는 함수를 일반적으로 **활성화 함수**라고 함
$$a = b + w_1x_1 + w_2x_2$$
$$y = h(a)$$
 - 활성화 함수의 처리 과정



- 가중치 신호를 조합한 결과가 a 라는 노드가 되고, 활성화 함수 $h()$ 를 통하여 y 라는 노드로 변환되는 과정

활성화 함수

- 활성화 함수
 - 앞서 본 것처럼 어떤 임계값을 경계로 출력이 바뀌는 함수를 **계단 함수**라고 함
 - 계단 함수는 활성화 함수 중 하나라고 말할 수 있음

Note . 일반적으로 **단층 퍼셉트론**은 단층 네트워크에서 계단 함수를 사용한 모델을 가리키고 **다층 퍼셉트론**은 신경망(여러층으로 구성되고 시그모이드 함수 등의 매끈한 활성화 함수를 사용하는 네트워크)을 가리킴

- 시그모이드 함수

$$h(x) = \frac{1}{1 + \exp(-x)}$$

- $\exp(-x)$ 는 e^{-x} 을 뜻하며, e는 자연상수로 2.7182...의 값을 갖는 실수
- 신경망에서는 활성화 함수로 시그모이드 함수를 이용하여 신호를 변환하고, 그 변환된 신호를 다음 뉴런에 전달함
- 퍼셉트론과의 차이는 활성화 함수 뿐임. 뉴런이 다층으로 이어진 구조와 신호 전달 방법은 같음

활성화 함수

- 계단 함수 구현하기

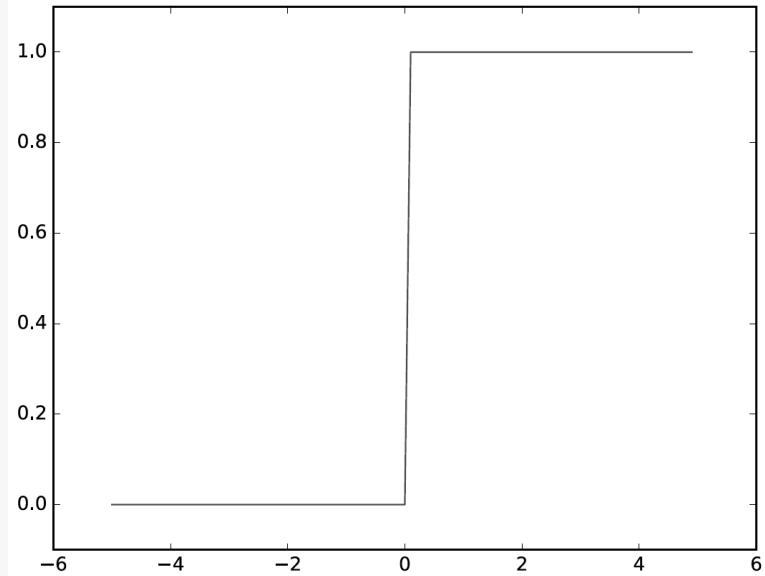
```
def step_function(x):
    return np.array(x > 0, dtype=np.int)
```

- 계단 함수의 그래프

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

def step_function(x):
    return np.array(x > 0, dtype=np.int)

X = np.arange(-5.0, 5.0, 0.1)
Y = step_function(X)
plt.plot(X, Y)
plt.ylim(-0.1, 1.1) # y축의 범위 지정
plt.show()
```



활성화 함수

- 시그모이드 함수 구현하기

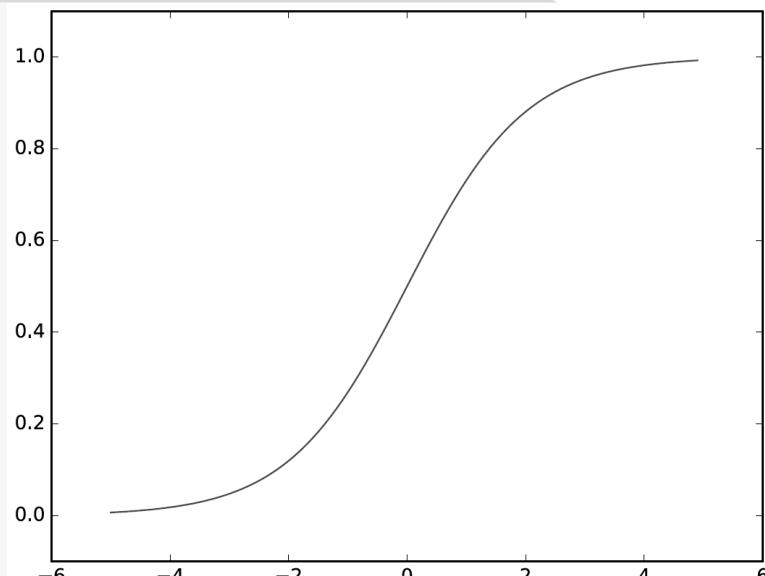
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

- 시그모이드 함수의 그래프

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

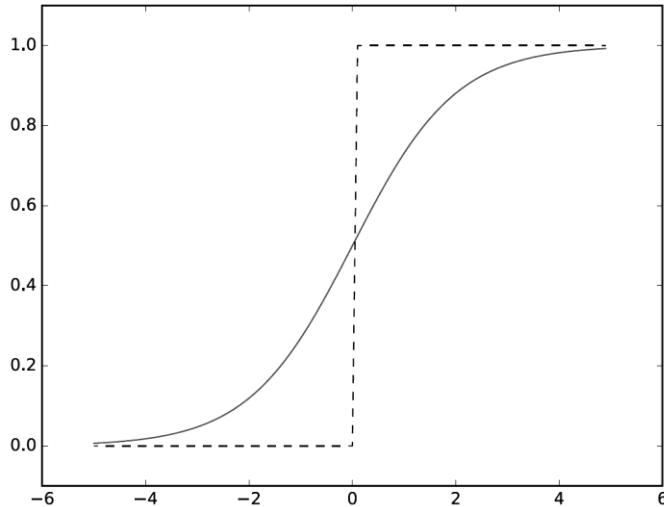
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

X = np.arange(-5.0, 5.0, 0.1)
Y = sigmoid(X)
plt.plot(X, Y)
plt.ylim(-0.1, 1.1)
plt.show()
```



활성화 함수

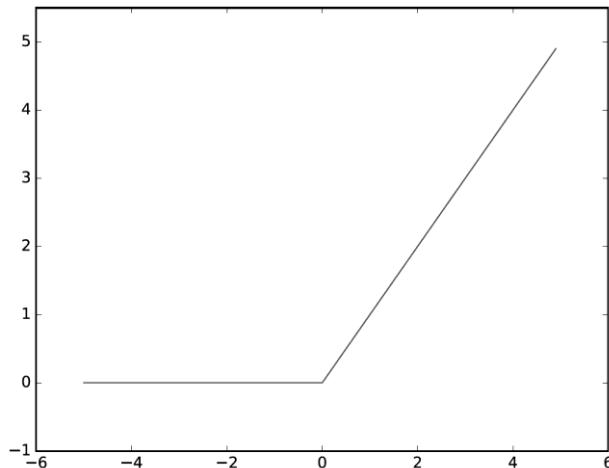
- 시그모이드 함수와 계단 함수 비교



- 매끄러움의 차이
- 시그모이드 함수는 부드러운 곡선이며 입력에 따라 출력이 연속적으로 변화
- 계단 함수는 0을 경계로 출력이 갑자기 바뀜
- 시그모이드 함수의 이 매끈함이 신경망 학습에서 중요한 역할을 함
- 계단 함수가 0과 1 중 하나의 값만 돌려주는 반면 시그모이드 함수는 실수를 돌려준다는 점도 다름. 즉 퍼셉트론에서는 뉴런 사이에 0과 1이 흘렀다면, 신경망에서는 연속적인 실수가 흐름
- 공통점은 입력이 작을 때의 출력은 0에 가깝고, 입력이 커지면 1에 가까워지는 구조
- 즉 입력이 중요하면 큰 값을 출력하고 입력이 중요하지 않으면 작은 값을 출력
- 그리고 입력이 아무리 작거나 커도 출력을 0에서 1사이라는 공통점

활성화 함수

- 비선형 함수
 - 계단 함수와 시그모이드 함수의 중요한 공통점은 **비선형 함수**라는 점
 - 신경망에서는 활성화 함수로 비선형 함수를 사용해야 함
 - 선형 함수를 이용하면 신경망의 층을 깊게 하는 의미가 없어짐
- ReLU 함수
 - ReLU는 입력이 0을 넘으면 그 입력을 그대로 출력하고, 0 이하이면 0을 출력
 - ReLU 함수의 그래프와 수식

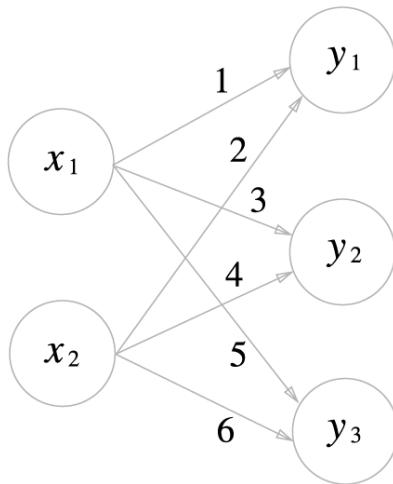


$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

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

다차원 배열

- 신경망에서의 행렬 곱
 - 행렬의 곱으로 신경망의 계산을 수행



$$\begin{array}{c} \left(\begin{array}{ccc} 1 & 3 & 5 \\ 2 & 4 & 6 \end{array} \right) \\ X \quad W = Y \\ \boxed{2 \quad 2 \times 3 \quad 3} \\ \text{일치} \end{array}$$

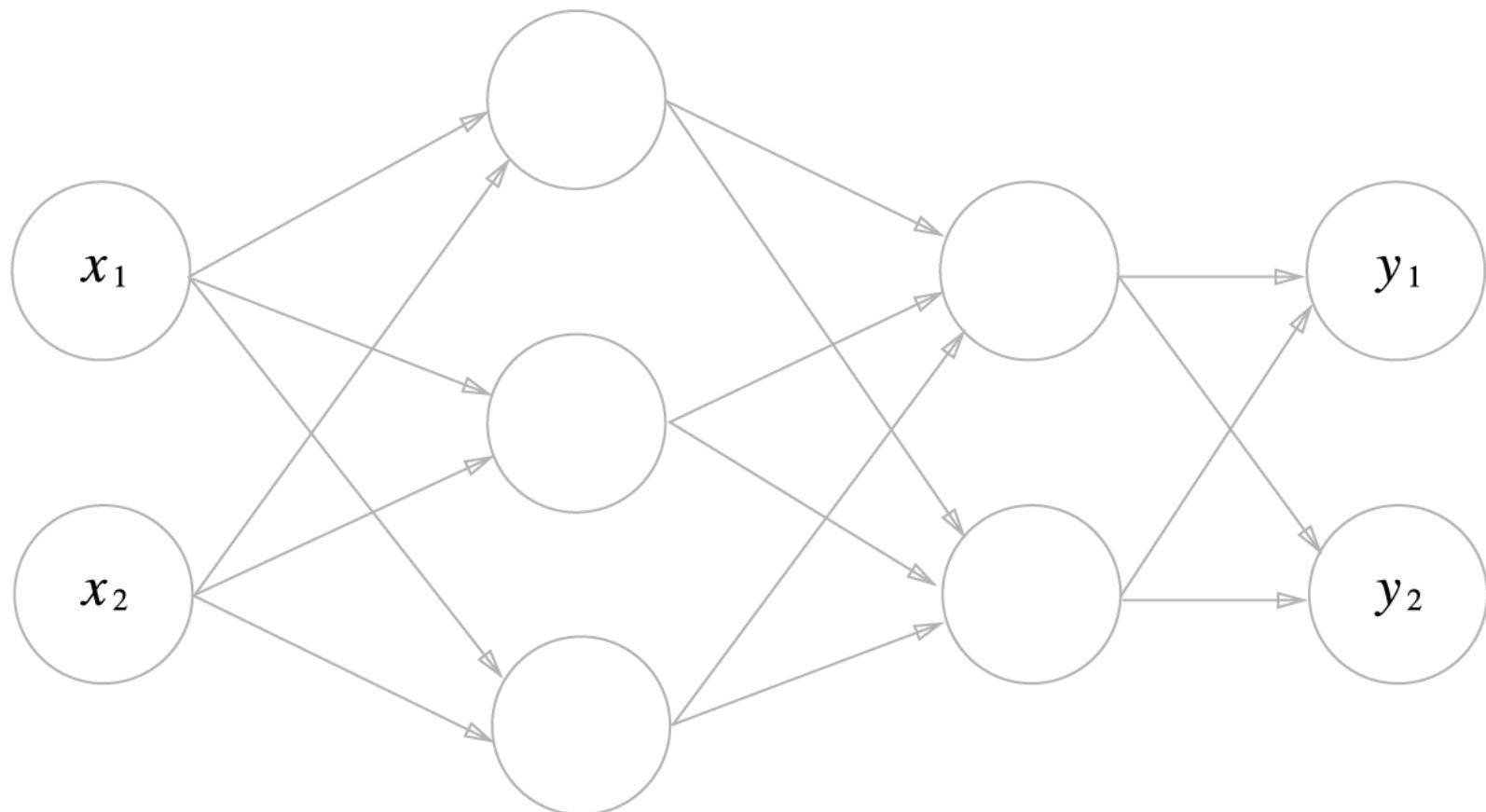
- X, W, Y의 형상에 주의

```
X = np.array([1, 2])
W = np.array([[1, 3, 5], [2, 4, 6]])
Y = np.dot(X, W)
```

- 다차원 배열의 스칼라 곱을 해주는 np.dot을 사용하면 단번에 Y를 계산

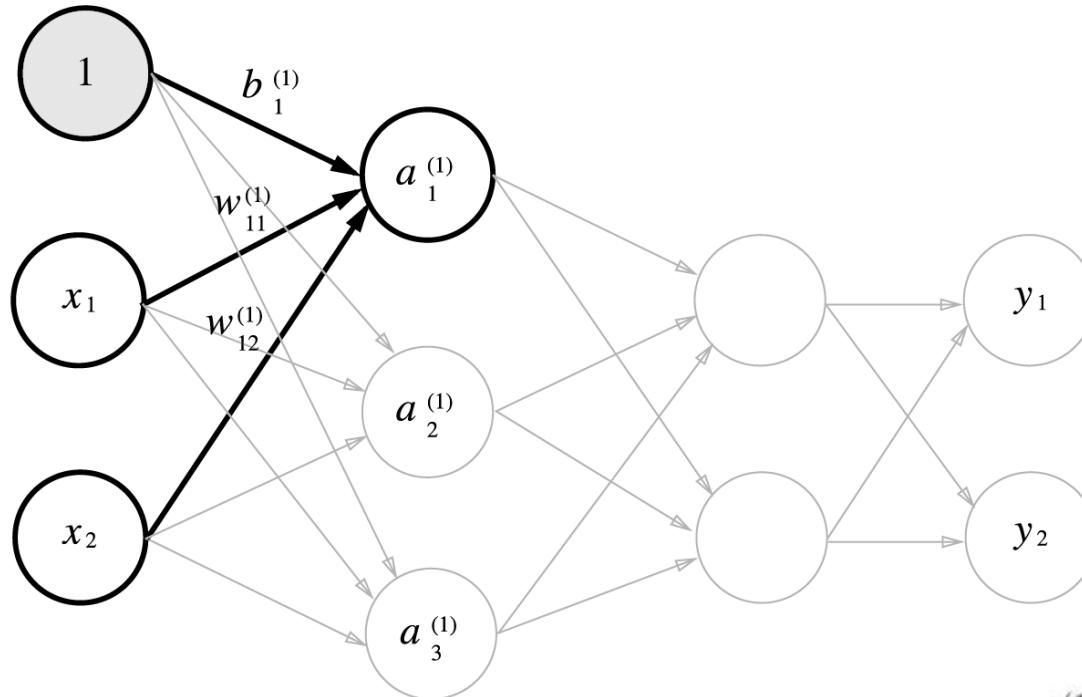
3층 신경망 구현하기

- 3층 신경망
 - 입력층은 2개, 첫 번째 은닉층은 3개, 두 번째 은닉층은 2개, 출력층은 2개의 뉴런으로 구성



3층 신경망 구현하기

- 각 층의 신호 전달 구현하기



$$A^{(1)} = (a_1^{(1)}, a_2^{(1)}, a_3^{(1)})$$

$$X = (x_1, x_1, x_2)$$

$$a_1^{(1)} = w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + b_1^{(1)}$$

$$B^{(1)} = (b_1^{(1)}, b_2^{(1)}, b_3^{(1)})$$

- 편향을 뜻하는 뉴런 추가

$$\mathbf{A}^{(1)} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{B}^{(1)}$$

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

- 행렬의 곱을 이용한 식

3층 신경망 구현하기

- 각 층의 신호 전달 구현하기

```
X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])

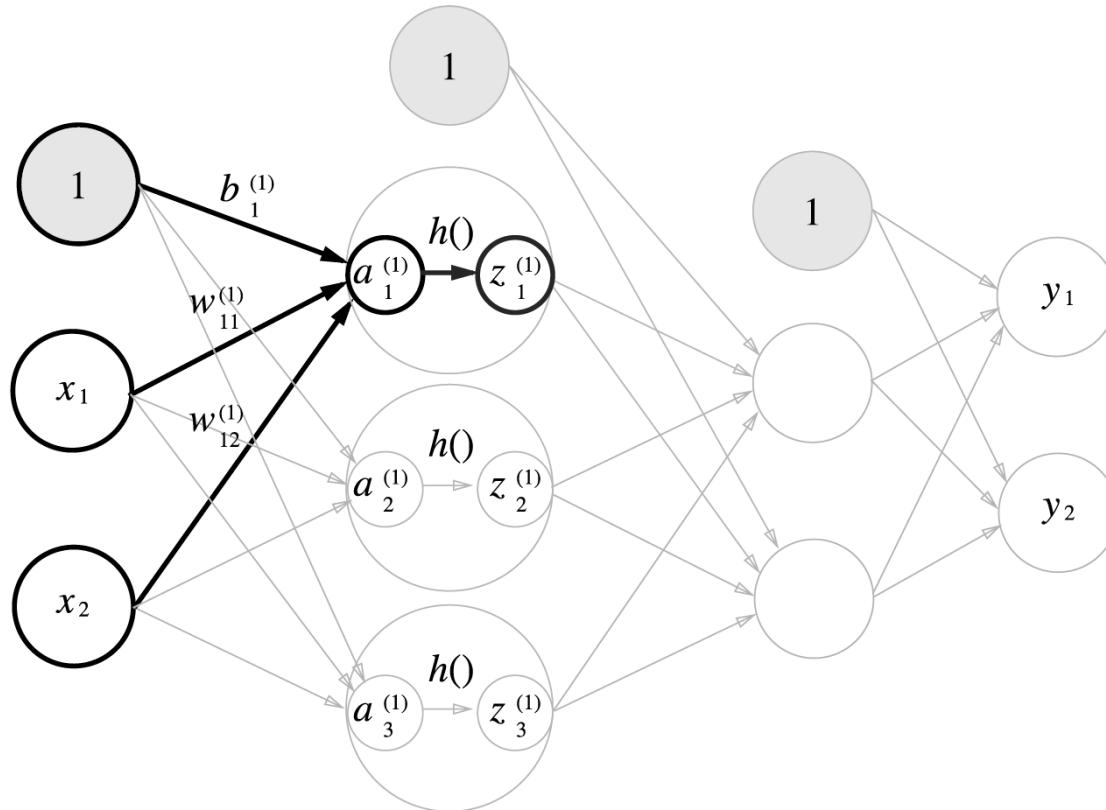
A1 = np.dot(X, W1) + B1
```

```
print(X.shape)
print(W1.shape)
print(B1.shape)
print(A1.shape)
```

```
(2,)
(2, 3)
(3,)
(3,)
```

3층 신경망 구현하기

- 각 층의 신호 전달 구현하기
 - 입력층에서 1층으로의 신호 전달

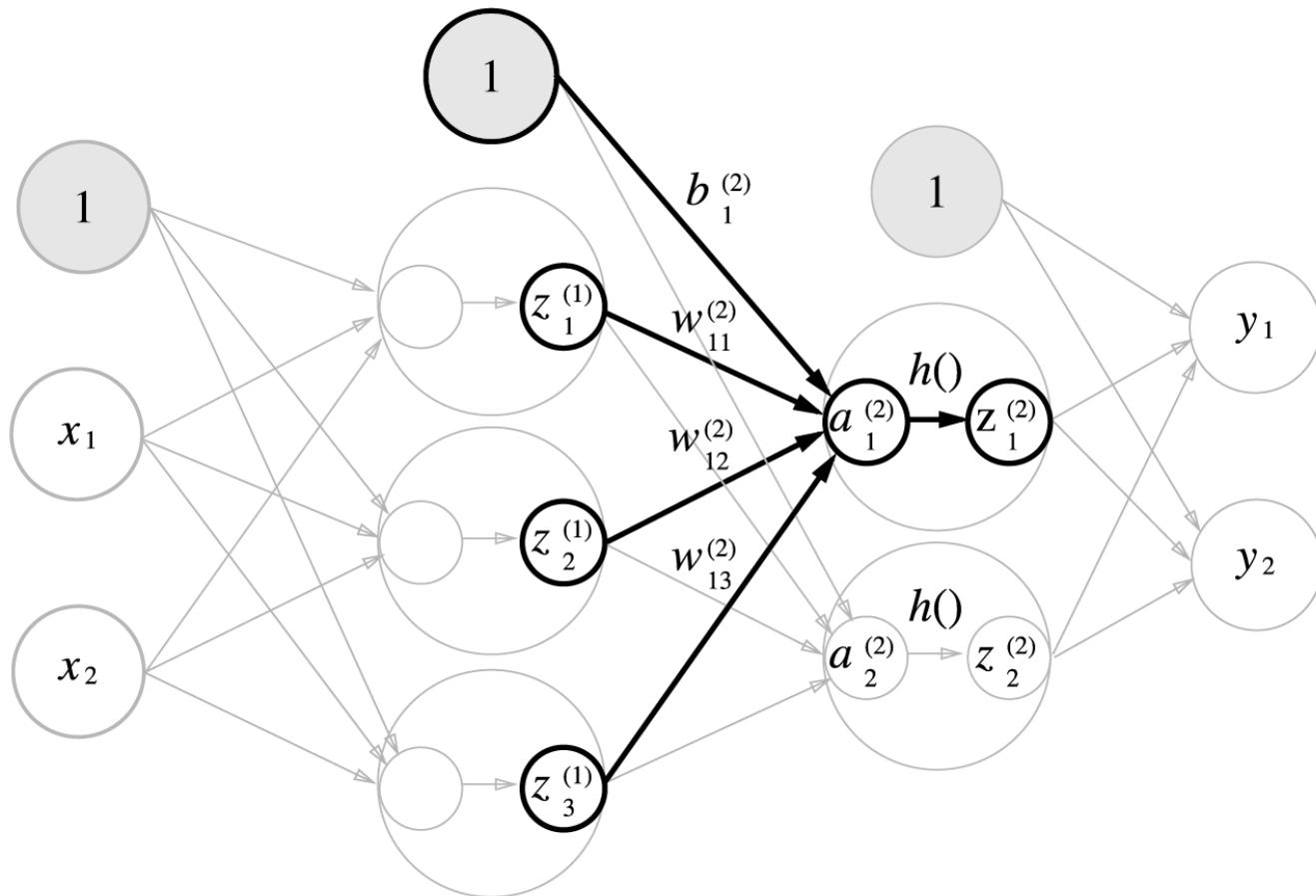


- 은닉층에서 가중치의 합을 a 로 표기, 활성화 함수 $h()$ 로 변환된 신호를 z 로 표기

```
Z1 = sigmoid(A1)
```

3층 신경망 구현하기

- 각 층의 신호 전달 구현하기
 - 1층에서 2층으로의 신호 전달



3층 신경망 구현하기

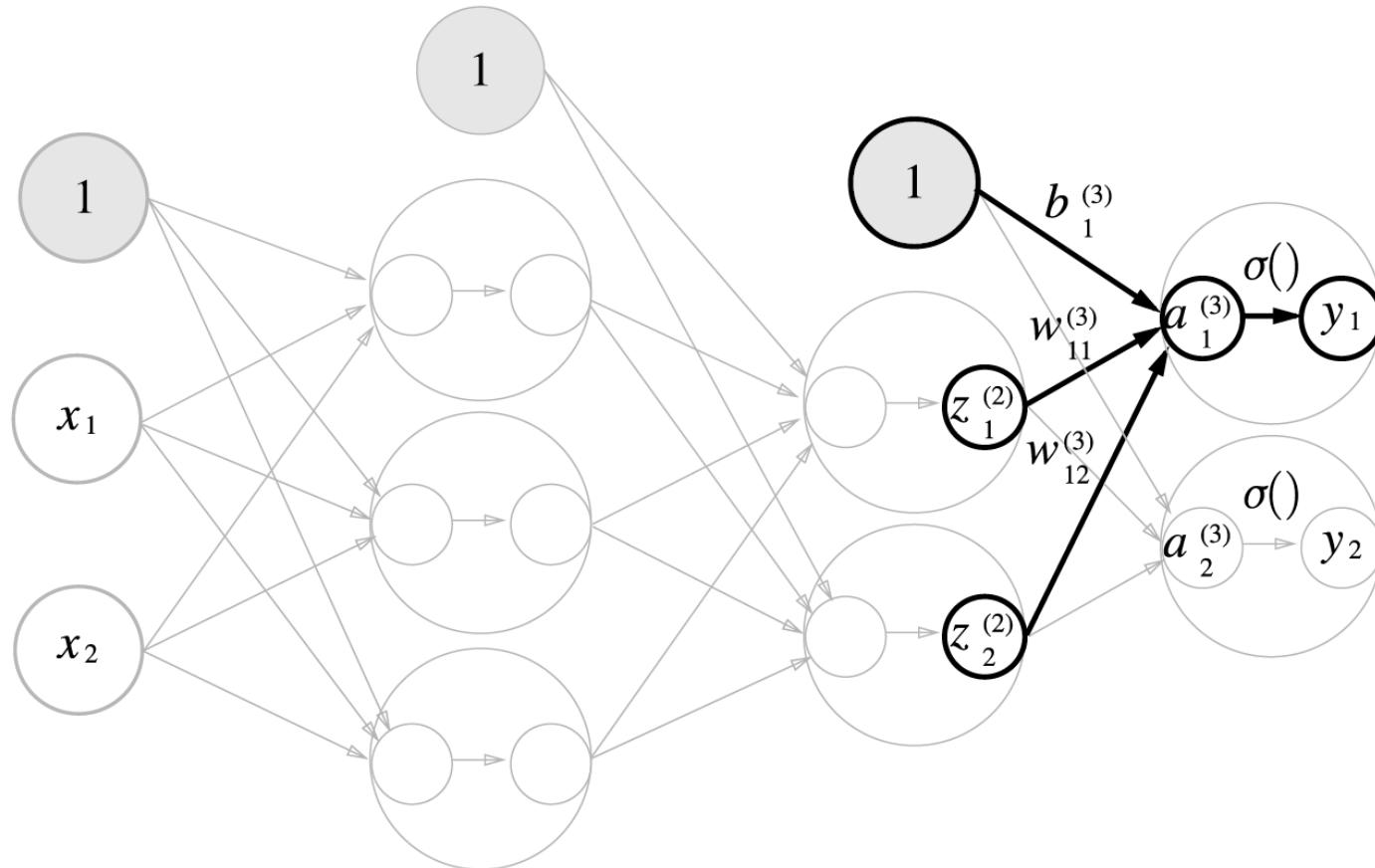
- 각 층의 신호 전달 구현하기
 - 1층의 출력 Z1이 2층의 입력이 됨

```
W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
B2 = np.array([0.1, 0.2])

A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
```

3층 신경망 구현하기

- 각 층의 신호 전달 구현하기
 - 2층에서 출력층으로 신호 전달



3층 신경망 구현하기

- 각 층의 신호 전달 구현하기
 - 2층에서 출력층으로 신호 전달

```
def identity_function(x):
    return x

W3 = np.array([[0.1, 0.3], [0.2, 0.4]])
B3 = np.array([0.1, 0.2])

A3 = np.dot(Z2, W3) + B3
Y = identity_function(A3)
```

- 항등 함수인 `identity_function()`을 정의하고, 이를 출력층의 활성화 함수로 이용
- 항등 함수는 입력을 그대로 출력하는 함수
- 이전 층에서의 흐름과 통일하기 위해 만듬.
- 출력층의 활성화 함수는 풀고자 하는 문제의 성질에 맞게 정하면 됨
- 예를 들어 회귀에서는 항등 함수를, 2 클래스 분류에는 시그모이드 함수를, 다중 클래스 분류에는 소프트맥스 함수를 사용하는 것이 일반적

3층 신경망 구현하기

- 구현 정리

```
def init_network():
    network = []
    network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
    network['B1'] = np.array([0.1, 0.2, 0.3])
    network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
    network['B2'] = np.array([0.1, 0.2])
    network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])
    network['B3'] = np.array([0.1, 0.2])

    return network

def forward(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['B1'], network['B2'], network['B3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = identity_function(a3)

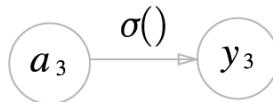
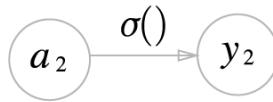
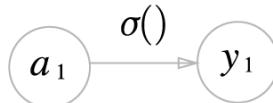
    return y

network = init_network()
x = np.array([1.0, 0.5])
y = forward(network, x)
```

출력층 설계하기

- 항등 함수와 소프트맥스 함수 구현하기

- 항등 함수(identity function)



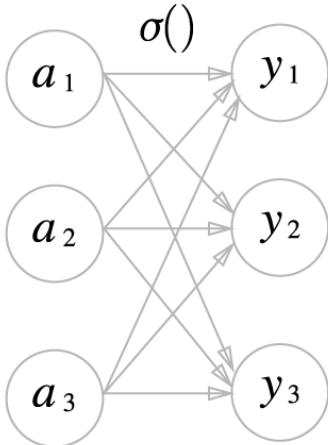
- 분류에서 사용하는 소프트맥스 함수 (softmax function)

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i)}$$

- $\exp(x)$: e^x 을 뜻하는 지수함수.(e는 자연상수)
 - n : 출력층의 뉴런 수
 - y_k : 그중 k번째 출력임을 나타냄
 - 소프트맥스 함수의 분자 = 입력 신호 a_k 의 지수 함수
 - 소프트맥스 함수의 분모 = 모든 입력 신호의 지수 함수의 합

출력층 설계하기

- 항등 함수와 소프트맥스 함수 구현하기
 - 소프트맥스 함수



- 소프트맥스의 출력은 모든 입력 신호로부터 화살표를 받음. 출력층의 각 뉴런이 모든 입력 신호에서 영향을 받기 때문

```
def softmax(a):  
    exp_a = np.exp(a)  
    sum_exp_a = np.sum(exp_a)  
    y = exp_a / sum_exp_a  
  
    return y
```

출력층 설계하기

- 소프트맥스 함수 구현시 주의점
 - 오버플로 문제를 개선한 수식

$$\begin{aligned}y_k &= \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i)} = \frac{C \exp(a_k)}{C \sum_{i=1}^n \exp(a_i)} \\&= \frac{\exp(a_k + \log C)}{\sum_{i=1}^n \exp(a_i + \log C)} \\&= \frac{\exp(a_k + C')}{\sum_{i=1}^n \exp(a_i + C')}\end{aligned}$$

- 소프트맥스의 지수 함수를 계산할 때 어떤 정수를 더해도 결과는 바뀌지 않음
- 오버플로를 막을 목적으로는 입력 신호 중 최댓값을 이용하는 것이 일반적

출력층 설계하기

- 소프트맥스 함수 구현시 주의점
 - 오버플로 문제를 개선한 소프트맥스 함수

```
def softmax(a):
    c = np.max(a)
    exp_a = np.exp(a - c)
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y
```

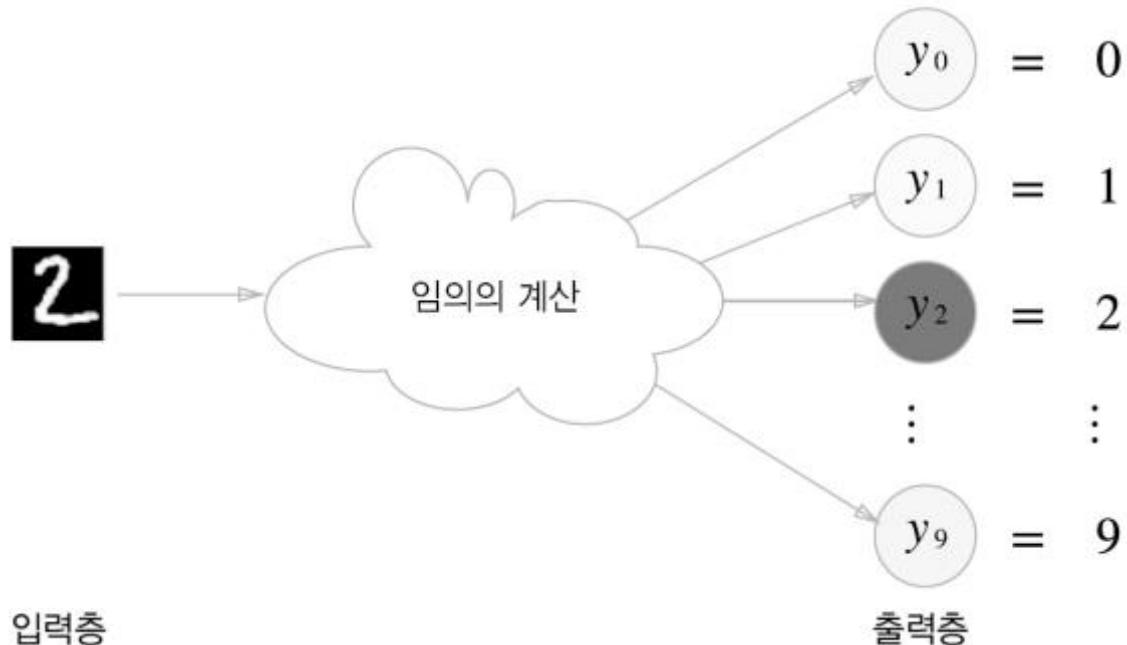
- 입력 신호 중 최댓값(이 예에서는 c)을 빼주면 해결됨

출력층 설계하기

- 소프트맥스 함수의 특징
 - 소프트맥스 함수의 출력은 0에서 1.0 사이의 실수
 - 또 소프트맥스 함수 출력의 총합은 1 (중요한 성질)
 - 이 성질로 소프트맥스 함수의 출력을 '확률'로 해석할 수 있음
 - 주의점은 소프트맥스 함수를 적용해도 각 원소의 대소 관계는 변하지 않음
 - 이는 지수 함수 $y=\exp(x)$ 가 단조 증가 함수이기 때문
 - 따라서 a 의 원소들 사이의 대소 관계가 y 의 원소들 사이의 대소 관계로 그대로 이어짐. 예를 들어 a 에서 가장 큰 원소는 2번째 원소이고, y 에서 가장 큰 원소도 2번째 원소임
 - 신경망을 이용한 분류에서는 일반적으로 가장 큰 출력을 내는 뉴런에 해당하는 클래스만 인식
 - 소프트맥스 함수를 적용해도 출력이 가장 큰 뉴런의 위치는 달라지지 않으므로 신경망으로 분류할 때 출력층의 소프트맥스 함수를 생략 가능
 - Note. 추론 단계에서는 출력층의 소프트맥스 함수를 생략하는 것이 일반적이지만 신경망을 학습시킬 때는 출력층에서 소프트맥스 함수를 사용

출력층 설계하기

- 출력층의 뉴런 수 정하기
 - 출력층의 뉴런 수는 풀려는 문제에 맞게 적절히 정해야 함
 - 분류에서는 분류하고 싶은 클래스 수로 설정하는 것이 일반적
 - 예를 들어 입력 이미지를 숫자 0부터 9 중 하나로 분류하는 문제라면 출력층의 뉴런을 10개로 설정



손글씨 숫자 인식

- 손글씨 숫자 분류
 - 이미 학습된 매개변수를 사용하여 학습 과정은 생략하고, 추론 과정만 구현
 - 이 추론 과정을 신경망의 **순전파**(forward propagation)라고도 함
- MNIST 데이터셋
 - 0부터 9까지의 숫자 이미지로 구성
 - 훈련 이미지가 60,000장, 시험 이미지가 10,000장
 - 일반적으로 이 훈련 이미지들을 사용하여 모델을 학습하고, 학습한 모델로 시험 이미지들을 얼마나 정확하게 분류하는지 평가



- MNIST 이미지 데이터는 28*28 크기의 회색조 이미지(1채널)이며, 각 픽셀을 0에서 255까지의 값을 취함
- 각 이미지에는 '7', '2'와 같이 그 이미지가 의미하는 숫자가 레이블로 붙어 있음

손글씨 숫자 인식

- MNIST 데이터셋

```
import numpy as np
from dataset.mnist import load_mnist
from PIL import Image

def img_show(img):
    pil_img = Image.fromarray(np.uint8(img))
    pil_img.show()

(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize=False)

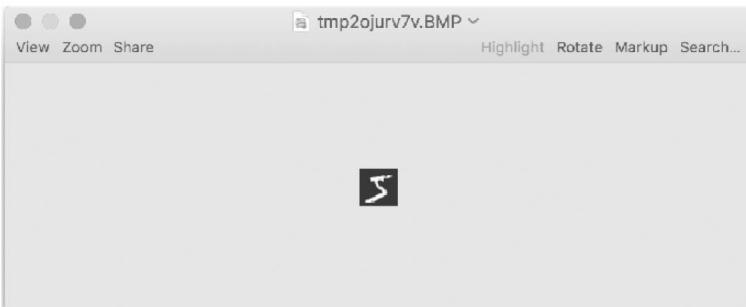
img = x_train[0]
label = t_train[0]
print(label) # 5

print(img.shape) # (784,)
img = img.reshape(28, 28) # 형상을 원래 이미지의 크기로 변형
print(img.shape) # (28, 28)

img_show(img)
```

손글씨 숫자 인식

- MNIST 데이터셋
 - `load_mnist` 함수는 읽은 MNIST 데이터를 "(훈련 이미지, 훈련 레이블), (시험 이미지, 시험 레이블)" 형식으로 반환
 - 인수로는 `normalize`, `flatten`, `one_hot_label` 세 가지를 설정할 수 있음
 - `normalize`는 입력 이미지의 픽셀 값을 0.0~1.0 사이의 값으로 정규화 할지 정함
 - `flatten`은 입력 이미지를 평탄하게, 즉 1차원 배열로 만들지 정함
 - `one_hot_label`은 레이블을 원-핫 인코딩(one-hot encoding) 형태로 저장할지 정함
 - 원-핫 인코딩이란, 예를 들어 [0, 0, 1, 0, 0, 0, 0, 0, 0] 처럼 정답을 뜻하는 원소만 1이고 나머지는 모두 0인 배열



- `flatten = True`로 설정해 읽어 들인 이미지는 1차원 넘파이 배열이므로 이미지를 표시할 때는 원래 형상인 28*28 크기로 다시 변형해야 함(`reshape` 함수)

손글씨 숫자 인식

- 신경망의 추론 처리

```
def get_data():
    (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, flatten=True, one_hot_label=False)
    return x_test, t_test

def init_network():
    with open("sample_weight.pkl", 'rb') as f:
        network = pickle.load(f)
    return network

def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)

    return y
```

손글씨 숫자 인식

- 신경망의 추론 처리
 - 뉴런을 784개, 출력층 뉴런을 10개로 구성
 - 입력층 뉴런이 784개인 이유는 이미지 크기가 $28 \times 28 = 784$ 이기 때문
 - 출력층 뉴런이 10개인 이유는 이 문제가 0에서 9까지의 숫자를 구분하는 문제
 - 은닉층은 총 두개로, 첫 번째 은닉층에는 50개의 뉴런을, 두 번째 은닉층에는 100개의 뉴런을 배치. 50과 100은 임의로 정한 값
 - `init_network()`에서는 pickle 파일인 `sample_weight.pkl`에 저장된 '학습된 가중치 매개변수'를 읽음. 이 파일에는 가중치와 편향 매개변수가 딕셔너리 변수로 저장
 - 추론 수행. 정확도 평가

```
x, t = get_data()
network = init_network()
accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i])
    p= np.argmax(y) # 확률이 가장 높은 원소의 인덱스를 얻는다.
    if p == t[i]:
        accuracy_cnt += 1

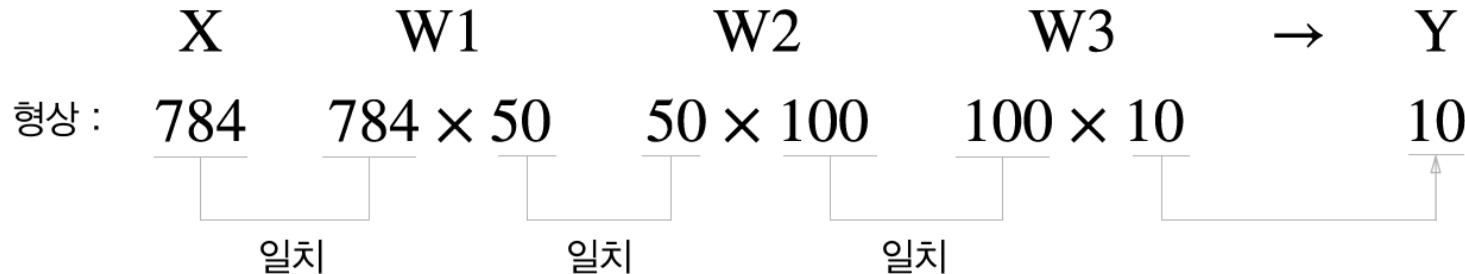
print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

손글씨 숫자 인식

- 신경망의 추론 처리
 - predict() 함수는 각 레이블의 확률을 넘파이 배열로 반환
 - np.argmax() 함수로 이 배열에서 값이 가장 큰(확률이 가장 높은) 원소의 인덱스를 구함. 이것이 바로 예측 결과임
 - 마지막으로 신경망이 예측한 답변과 정답 레이블을 비교하여 맞힌 숫자를 세고 이를 전체 이미지 숫자로 나눠 정확도를 구함
 - 또한 load_mnist 함수의 normalize를 True로 설정해서 0~255 범위인 각 픽셀의 값을 0.0~1.0 범위로 변환(단순히 픽셀의 값을 255로 나눔)
 - 이처럼 데이터를 특정 범위로 변환하는 처리를 **정규화**(normalization)라고 하고, 신경망의 입력 데이터에 특정 변환을 가하는 것을 **전처리**(pre-processing)라 함
 - 여기에서는 입력 이미지 데이터에 대한 전처리 작업으로 정규화를 수행

손글씨 숫자 인식

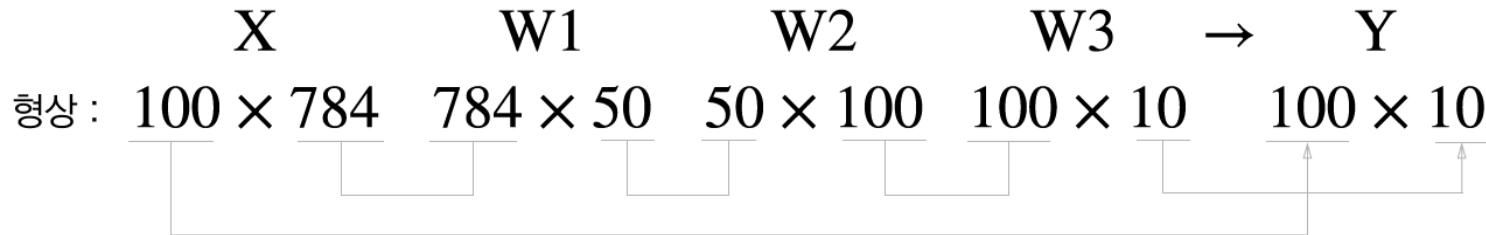
- 배치 처리
 - 신경망 각 층의 배열 형상의 추이



- 전체적으로 보면 원소 784개로 구성된 1차원 배열(원래는 28×28 인 2차원 배열)이 입력되어 마지막에는 원소가 10개인 1차원 배열이 출력되는 흐름
 - 이는 이미지 데이터를 1장만 입력했을 때의 처리 흐름임

손글씨 숫자 인식

- 배치 처리
 - 배치 처리를 위한 배열들의 형상 추이



- 입력 데이터의 형상은 $100*784$, 출력 데이터의 형상은 $100*10$
- 이는 100장 분량 입력 데이터의 결과가 한 번에 출력됨을 나타냄
- 이처럼 하나로 묶음 입력 데이터를 **배치**(batch)라 함
- Note. 배치 처리의 이점
 - 1) 수치 계산 라이브러리 대부분이 큰 배열을 효율적으로 처리할 수 있도록 고도로 최적화 되어 있음
 - 2) 큰 배열을 한꺼번에 계산하는 것이 버스에 주는 부하를 줄여 분할된 작은 배열을 여러 번 계산하는 것보다 빠름

손글씨 숫자 인식

- 배치 처리

```
x, t = get_data()
network = init_network()

batch_size = 100 # 배치 크기
accuracy_cnt = 0

for i in range(0, len(x), batch_size):
    x_batch = x[i:i+batch_size]
    y_batch = predict(network, x_batch)
    p = np.argmax(y_batch, axis=1)
    accuracy_cnt += np.sum(p == t[i:i+batch_size])

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

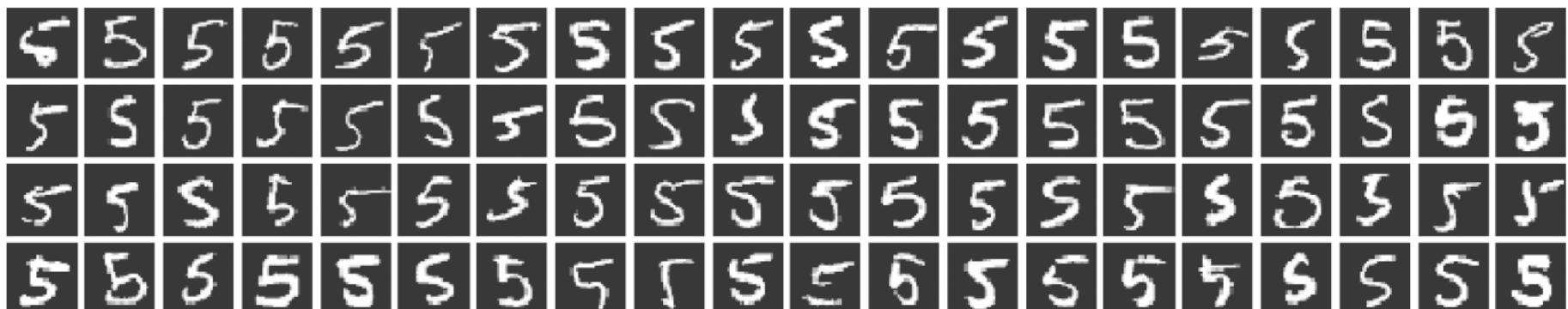
정리

- 신경망
 - 신경망에서는 활성화 함수로 시그모이드 함수와 ReLU 함수 같은 매끄럽게 변화하는 함수를 이용
 - 넘파이의 다차원 배열을 잘 사용하면 신경망을 효율적으로 구현할 수 있음
 - 기계학습 문제는 크게 회귀와 분류로 나눌 수 있음
 - 출력층의 활성화 함수로는 회귀에서는 주로 항등 함수를, 분류에서는 주로 소프트맥스 함수를 이용
 - 분류에서는 출력층의 뉴런 수를 분류하려는 클래스 수와 같게 설정
 - 입력 데이터를 묶은 것을 배치라 하며, 추론 처리를 이 배치 단위로 진행하면 결과를 훨씬 빠르게 얻을 수 있음

신경망 학습

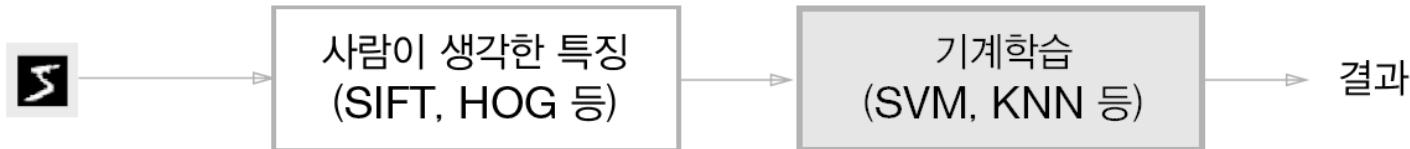
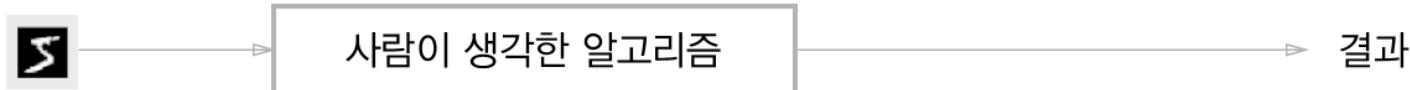
데이터 주도 학습

- 학습이란?
 - 훈련 데이터로부터 가중치 매개변수의 최적값을 자동으로 획득하는 것을 뜻함
 - 신경망이 학습이 가능하도록 해주는 지표인 손실 함수
 - 이 손실 함수의 결괏값을 가장 작게 만드는 가중치 매개변수를 찾는 것이 학습의 목표
- 데이터 주도 학습
 - 기계학습은 데이터가 생명
 - 그래서 기계학습의 중심에는 데이터가 존재함
 - 데이터가 이끄는 접근 방식 덕에 사람 중심 접근에서 벗어날 수 있음



[그림] 손글씨 숫자 '5'의 예 : 사람마다 자신만의 필체가 있다

데이터 주도 학습



[그림] 규칙을 '사람'이 만드는 방식에서 '기계'가 데이터로부터 배우는 방식으로의 패러다임 전환 : 회색 블록은 사람이 개입하지 않음을 뜻함

- 그림과 같이 신경망은 이미지를 '있는 그대로' 학습
- 두 번째 접근 방식(특징과 기계학습 방식)에서는 특징을 사람이 설계했지만, 신경망은 이미지에 포함된 중요한 특징까지도 '기계'가 스스로 학습

훈련 데이터와 시험 데이터

- 기계학습 문제는 데이터를 **훈련 데이터**(training data)와 **시험 데이터**(test data)로 나눠 학습과 실험을 수행하는 것이 일반적
- 우선 훈련 데이터만 사용하여 학습하면서 최적의 매개변수를 찾음
- 그런 다음 시험 데이터를 사용하여 앞서 훈련한 모델의 실력을 평가
- 왜 훈련 데이터와 시험 데이터를 나눠야 할까?
- 그것은 우리가 원하는 것은 범용적으로 사용할수 있는 모델이기 때문
- 이 **범용 능력** 을 제대로 평가하기 위해 훈련 데이터와 시험 데이터를 분리
- 그래서 데이터셋 하나로만 매개변수의 학습과 평가를 수행하면 올바른 평가가 될 수 없음
- 수중의 데이터셋은 제대로 맞히더라도 다른 데이터셋에는 엉망인 일도 벌어짐
- 참고로 한 데이터셋에만 지나치게 최적화된 상태를 **오버피팅**(overfitting) 이라고 함
- 오버피팅 피하기는 기계학습의 중요한 과제

손실 함수

- 신경망 학습에서는 현재의 상태를 '하나의 지표'로 표현
- 그리고 그 지표를 가장 좋게 만들어주는 가중치 매개변수의 값을 탐색
- '행복 지표'를 가진 사람이 그 지표를 근거로 '최적의 인생'을 탐색하듯, 신경망도 '하나의 지표'를 기준으로 최적의 매개변수 값을 탐색
- 신경망 학습에서 사용하는 지표는 **손실 함수**(loss function) 라고 함.
- 이 손실 함수는 임의의 함수를 사용할 수도 있지만 일반적으로는 평균 제곱 오차와 교차 엔트로피 오차를 사용

오차제곱합

- 오차제곱합(sum of squares of error, SSE)

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2$$

- 여기서 y_k 는 신경망의 출력(신경망이 추정한 값), t_k 는 정답 레이블, k 는 데이터의 차원 수를 나타냄
- 이를테면 “손글씨 숫자 인식” 예에서 y_k 와 t_k 는 다음과 같은 원소 10 개짜리 데이터

```
>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

- 이 배열들의 원소는 첫 번째 인덱스부터 순서대로 숫자 '0', '1', '2', ...일 때의 값. 여기에서 신경망의 출력 y 는 소프트맥스 함수의 출력
- 소프트맥스 함수의 출력을 확률로 해석할 수 있음
- 정답 레이블인 t 는 정답을 가리키는 위치의 원소는 1로 그 외에는 0으로 표기
- 이처럼 한 원소만 1로 하고 그 외는 0으로 나타내는 표기법을 원-핫 인코딩

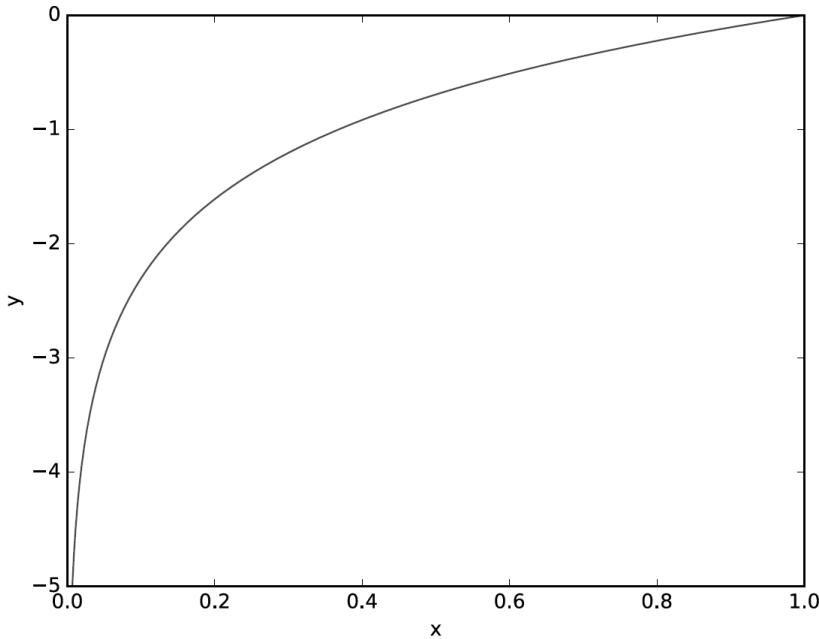
교차 엔트로피 오차

- 또 다른 손실 함수로서 교차 엔트로피 오차(cross entropy error , CEE) 도 자주 이용. 다음은 교차 엔트로피 오차의 수식

$$E = -\sum_k t_k \log y_k$$

- 여기에서 log는 밑이 e인 자연로그(\log_e)
- y_k 는 신경망의 출력, t_k 는 정답 레이블
- t_k 는 정답에 해당하는 인덱스의 원소만 1이고 나머지는 0(원-핫 인코딩)
- 그래서 위 식은 실질적으로 정답일 때의 추정(t_k 가 1일 때의 y_k 의 자연로그를 계산)
- 예를 들어 정답레이블은 '2'가 정답이고 이때의 신경망 출력이 0.6이라면 교차 엔트로피 오차는 $-\log 0.6 = 0.51$
- 또한 같은 조건에서 신경망 출력이 0.1이라면 $-\log 0.1 = 2.30$
- 즉, 교차 엔트로피 오차는 정답일 때의 출력이 전체 값을 정하게 됨

교차 엔트로피 오차



[그림] 자연로그 $y = \ln x$ 의 그래프

- 이 그림에서 보듯이 x 가 1 일 때 y 는 0 이 되고 x 가 0 에 가까워질수록 y 의 값은 점점 작아짐
- 앞의 교차 엔트로피 수식도 마찬가지로 정답에 해당하는 출력이 커질수록 0 에 다가가다가, 그 출력이 1 일 때 0이 됨
- 반대로 정답일 때의 출력이 작아질수록 오차는 커짐

미니배치 학습

- 기계학습 문제는 훈련 데이터에 대한 손실 함수의 값을 구하고, 그 값을 최대한 줄여주는 매개변수를 찾아냄
- 이렇게 하려면 모든 훈련 데이터를 대상으로 손실 함수 값을 구해야 함
- 즉, 훈련 데이터가 100 개 있으면 그로부터 계산한 100 개의 손실 함수 값들의 합을 지표로 삼는 것임

$$E = -\frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk}$$

- 이때 데이터가 N 개라면 t_{nk} 는 n 번째 데이터의 k 번째 값을 의미 (y_{nk} 는 신경망의 출력, t_{nk} 는 정답 레이블)
- 데이터 하나에 대한 손실 함수인 앞의 식을 단순히 N 개의 데이터로 확장하고 마지막에 N으로 나누어 정규화
- N으로 나눔으로써 '평균 손실 함수'를 구하는 것임
- 이렇게 평균을 사용하면 훈련 데이터의 개수와 관계없이 언제든 통일된 지표를 얻을 수 있음

미니배치 학습

- 그런데 MNIST 데이터셋은 훈련 데이터가 60,000개
- 모든 데이터를 대상으로 손실 함수의 합을 구하려면 시간이 걸림
- 더 나아가 빅데이터의 수준이 되면 수백만에서 수천만이 넘는 거대한 값
- 이 많은 데이터를 대상으로 일일이 손실 함수를 계산하는 것은 현실적이지 않음
- 이런 경우 데이터 일부를 추려 전체의 '근사치'로 이용
- 신경망 학습에서도 훈련 데이터로부터 일분만 골라 학습을 수행
- 이 일부를 **미니배치**(mini-batch)라고 함
- 가령 60,000장의 훈련 데이터 중에서 100장을 무작위로 뽑아 그 100장을 사용하여 학습
- 이러한 학습 방법을 **미니배치 학습**이라고 함

(배치용) 교차 엔트로피 오차 구현하기

```
def cross_entropy_error(y,t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    batch_size = y.shape[0]
    return -np.sum(t*np.log(y+1e-7)) /batch_size
```

```
def cross_entropy_error(y,t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t]+1e-7)) /batch_size
```

왜 손실 함수를 설정하는가?

- 숫자 인식의 경우도 궁극적인 목적은 높은 '정확도'를 끌어내는 매개변수를 찾는 것임
- 그런데 '정확도'라는 지표를 놔두고 '손실 함수의 값'이라는 우회적인 방법을 택하는 이유는 무엇일까?
- 이유는 신경망 학습에서의 '미분'의 역할에서 찾을 수 있음
- 신경망 학습에서 최적의 매개변수(가중치와 편향)를 탐색할 때 손실 함수의 값을 가능한 한 작게 하는 매개 변수 값을 찾음
- 이때 매개변수의 미분(정확히는 기울기)을 계산하고, 그 미분 값을 단서로 매개변수의 값을 서서히 갱신하는 과정을 반복
- 가중치 매개변수의 손실 함수의 미분이란 '가중치 매개변수의 값을 아주 조금 변화 시켰을 때, 손실 함수가 어떻게 변하나'라는 의미
- 만약 이 미분 값이 음수면 그 가중치 매개변수를 양의 방향으로 변화시켜 손실 함수의 값을 줄일 수 있음
- 반대로, 미분 값이 양수면 가중치 매개변수를 음의 방향으로 변화시켜 손실 함수의 값을 줄일 수 있음
- 그러나 미분 값이 0이면 가중치 매개변수를 어느 쪽으로 움직여도 손실 함수의 값은 줄어들지 않음

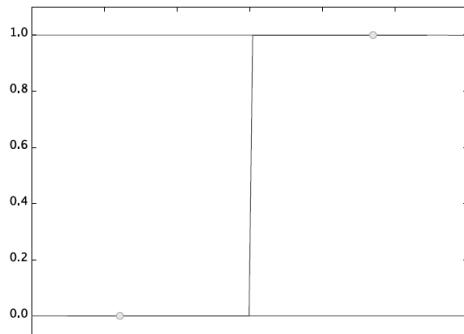
왜 손실 함수를 설정하는가?

- 정확도를 지표로 삼아서는 안 되는 이유는 미분 값이 대부분의 장소에서 0이 되어 매개변수를 갱신할 수 없기 때문
- 정확도를 지표로 삼으면 매개변수의 미분이 대부분의 장소에서 0이 되는 이유?
- 한 신경망이 100장의 훈련 데이터 중 32장을 올바로 인식한다고 가정- 정확도 32%
- 만약 정확도가 지표였다면 가중치 매개변수의 값을 조금 바꾼다고 해도 정확도는 그대로 32%임
- 즉, 매개변수를 약간만 조정해서는 정확도가 개선되지 않고 일정하게 유지
- 혹 정확도가 개선된다 하더라도 그 값을 32.0123%와 같은 연속적인 변화보다는 33%나 34% 처럼 불연속적인 값
- 한편 손실 함수의 경우는 매개변수의 값이 조금 변하면 그에 반응하는 손실 함수의 값도 연속적으로 변화
- 정확도는 매개변수의 미소한 변화에는 거의 반응을 보이지 않고, 반응이 있더라도 그 값이 불연속적으로 갑자기 변화

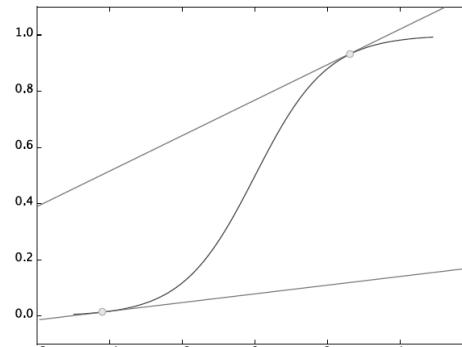
왜 손실 함수를 설정하는가?

- 이는 '계단 함수'를 활성화 함수로 사용하지 않는 이유와도 들어맞음
- 만약 활성화 함수로 계단 함수를 사용하면 지금까지 설명한 것과 같은 이유로 신경망 학습이 잘 이뤄지지 않음
- 계단 함수의 미분은 [그림]과 같이 대부분의 장소(0이외의 곳)에서 0
- 그 결과, 계단 함수를 이용하면 손실 함수를 지표로 삼는 게 아무 의미가 없게 됨.
- 매개변수의 작은 변화가 주는 파장을 계단 함수가 말살하여 손실 함수의 값에는 아무 런 변화가 나타나지 않기 때문

계단 함수



시그모이드 함수



[그림] 계단 함수와 시그모이드 함수 : 계단 함수는 대부분의 장소에서 기울기가 0
이지만, 시그모이드 함수의 기울기(접선)는 0이 아니다

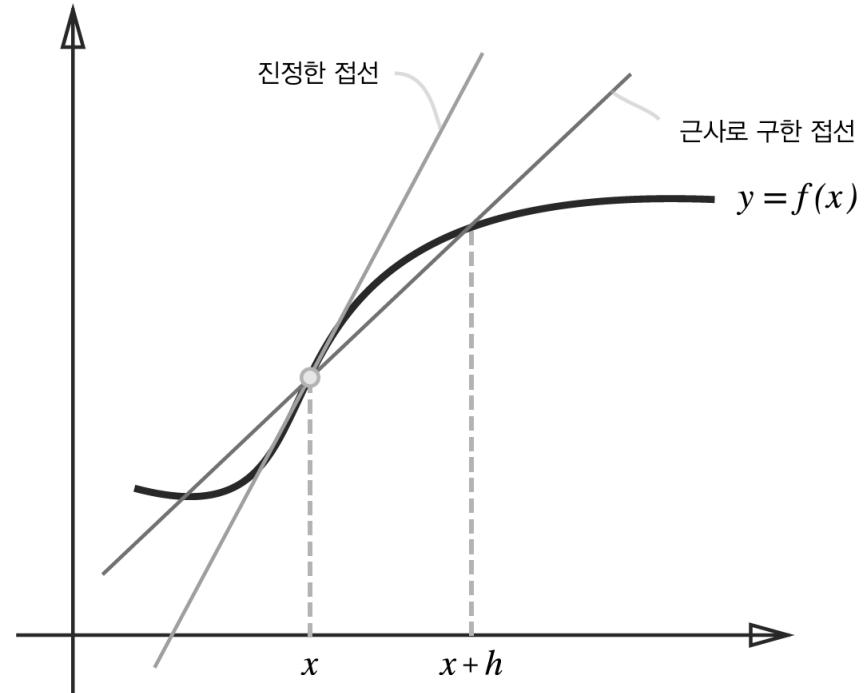
- 계단 함수는 한순간만 변화를 일으키지만, 시그모이드 함수의 미분(접선)은 [그림]과 같이 출력(세로축의 값)이 연속적으로 변하고 곡선의 기울기도 연속적으로 변함
- 즉, 시그모이드 함수의 미분은 어느 장소라도 0이 되지는 않음
- 이는 신경망 학습에서 중요한 성질로, 기울기가 0이 되지 않는 덕분에 신경망에 올바르게 학습할 수 있음

미분

- 미분은 한순간의 변화량

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

- 좌변은 $f(x)$ 의 x 에 대한 미분 (x 에 대한 $f(x)$ 의 변화량)을 나타내는 기호
- 결국, x 의 '작은 변화'가 함수 $f(x)$ 를 얼마나 변화시키느냐를 의미
- 마라톤의 예로 '달린 거리'가 '시간'에 대해 얼마나 변화했는가를 계산할 때 시간의 작은 변화, 즉 시간을 뜻하는 h 를 한없이 0에 가깝게 할 때 순간의 변화량(어느 순간의 속도)를 얻게 됨
- 진정한 미분(진정한 접선)과 수치미분(근사로 구한 접선)의 값은 다름
- 이 차이는 h 를 무한히 0으로 좁히는 것이 불가능해 생기는 한계
- 이 오차를 줄이기 위해 $(x + h)$ 와 $(x - h)$ 일 때의 함수 f 의 차분을 계산하는 방법을 쓰기도 함.
- 이 차분은 x 를 중심으로 그 전후의 차분을 계산 한다는 의미에서 중심 차분 혹은 중앙 차분이라함 (한편, $(x + h)$ 와 x 의 차분은 전방 차분이라 함)



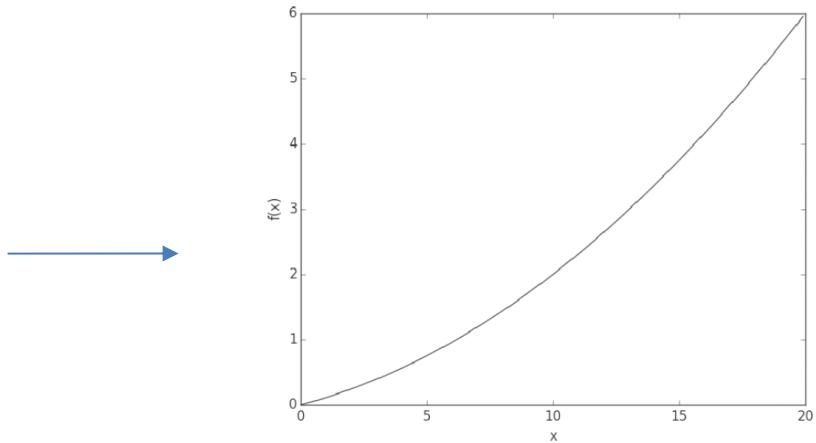
수치 미분의 예

$$y = 0.01x^2 + 0.1x$$

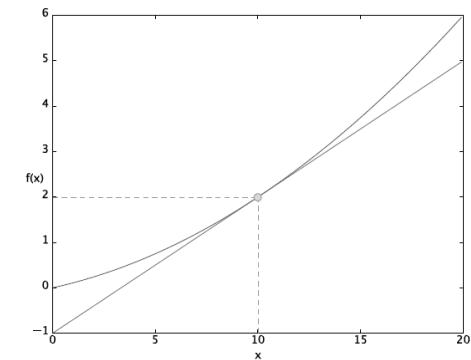
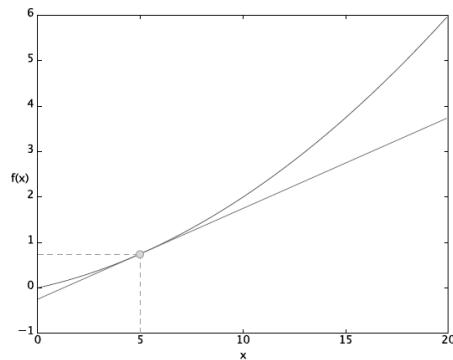


```
def function_1(x):  
    return 0.01*x**2 + 0.1*x
```

```
import numpy as np  
import matplotlib.pyplot as plt  
  
x = np.arange(0.0, 20.0, 0.1)  
y = function_1(x)  
plt.xlabel("x")  
plt.ylabel("f(x)")  
plt.plot(x, y)  
plt.show
```



```
>>> numerical_diff(function_1, 5)  
0.199999999990898  
>>> numerical_diff(function_1, 10)  
0.2999999999986347
```



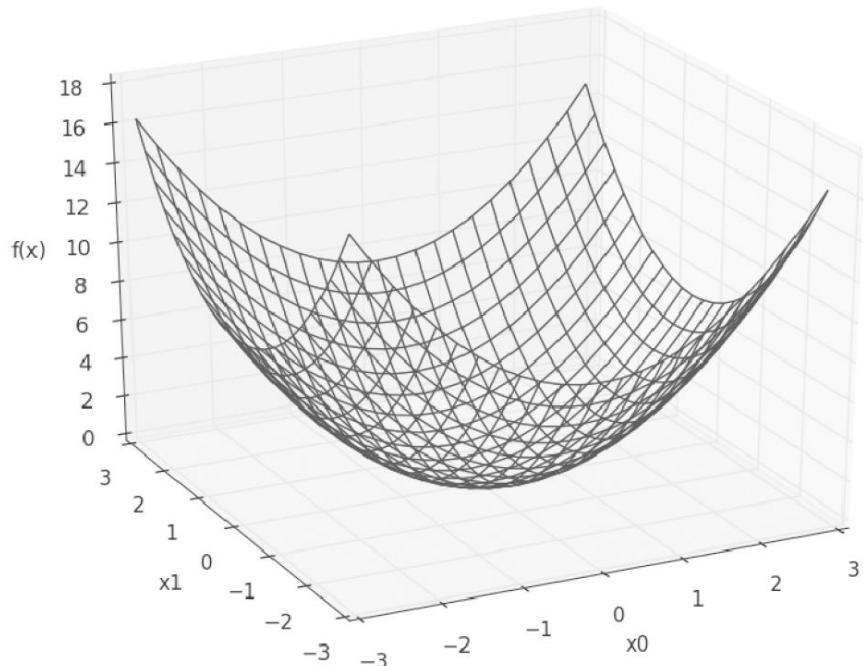
[그림] $x = 5$, $x = 10$ 에서의 접선 : 직선의 기울기는 수치 미분에서 구한 값을 사용

편미분

- 인수들의 제곱합을 구하는 식

$$f(x_0, x_1) = x_0^2 + x_1^2$$

- 위의 식을 미분하기 위해서는 변수가 2개 이므로 '어느 변수에 대한 미분이냐'를 구분해야 함. 이와 같이 변수가 여럿인 함수에 대한 미분을 **편미분**이라고 함



[그림] $f(x_0, x_1)$ 의 그래프

문제1 :

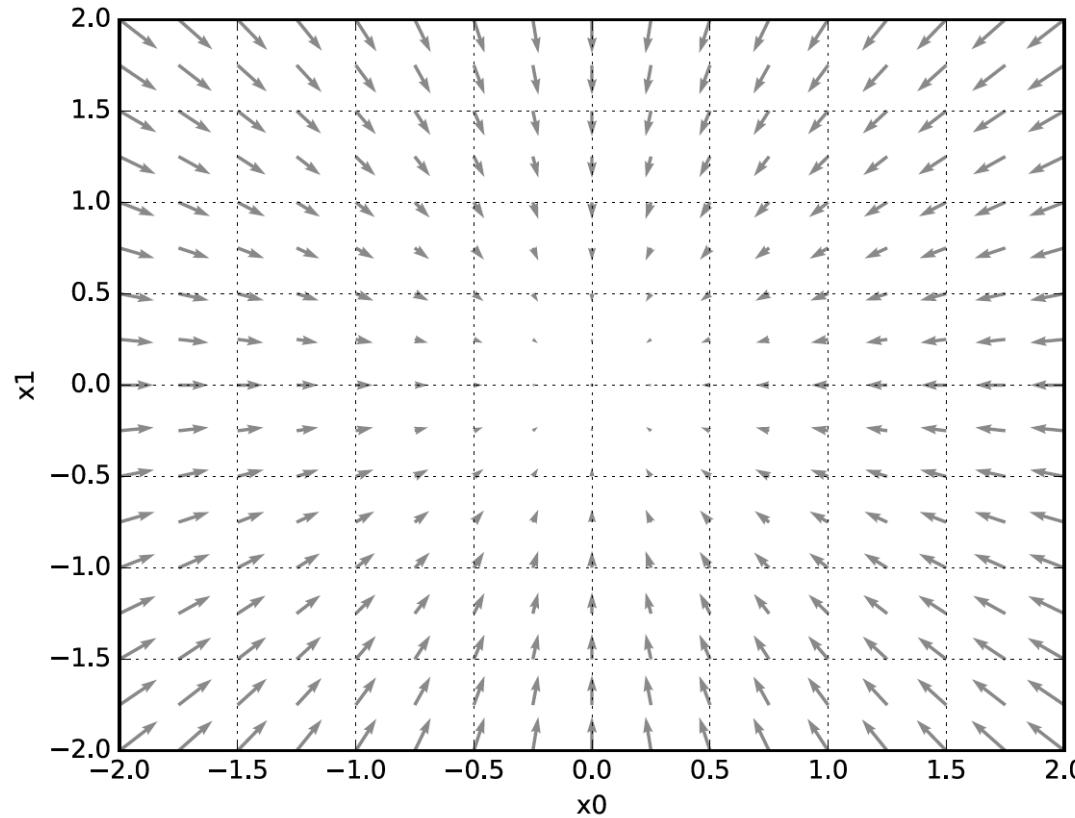
$x_0 = 3, x_1 = 4$ 일 때, x_0 에 대한 편미분을 구하라

문제2 :

$x_0 = 3, x_1 = 4$ 일 때, x_1 에 대한 편미분을 구하라

기울기

- x_0 와 x_1 의 편미분을 동시에 계산하고 싶은 경우
- $(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1})$ 처럼 모든 변수의 편미분을 벡터로 정리 -> **기울기**(gradient)
- **기울기가 가리키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방향**



경사법(경사하강법)

- 기계학습에서와 마찬가지로 신경망에서도 최적은 매개변수를 학습시에 찾아야함
- 최적이란 손실 함수가 최솟값이 될 때의 매개변수(가중치와 편향)
- 그러나 손실 함수는 복잡한 형태로 매개변수 공간이 광대하여 어디가 최솟값이 되는 곳인지를 짐작하기 힘듦
- 이런 상황에서 기울기를 잘 이용해 함수의 최솟값(또는 가능한 한 작은 값)을 찾으려는 것이 **경사법**
- 여기에서 주의할 점은 각 지점에서 함수의 값을 낮추는 방안을 제시하는 지표가 기울기라는 것
- 그러나 기울기가 가리키는 곳에 정말 함수의 최솟값이 있는지, 즉 그쪽이 나아갈 방향인지는 보장할 수 없음
- 실제로 복잡한 함수에서는 기울기가 가리키는 방향에 최솟값이 없는 경우가 대부분
- 기울어진 방향이 꼭 최솟값을 가리키는 것은 아니나, 그 방향으로 가야 함수의 값을 줄일 수 있음
- 따라서 최솟값이 되는 장소를 찾는 문제에서는 기울기 정보를 단서로 나아갈 방향을 정해야 함

경사법(경사하강법)

- 경사법은 현 위치에서 기울어진 방향으로 일정 거리만큼 이동함
- 그런 다음 이동한 곳에서도 마찬가지로 기울기를 구하고, 또 그 기울어진 방향으로 나아가기를 반복
- 이렇게 해서 함수의 값을 점차 줄이는 것이 경사법
- 경사법은 기계학습, 신경망 학습에 많이 사용됨

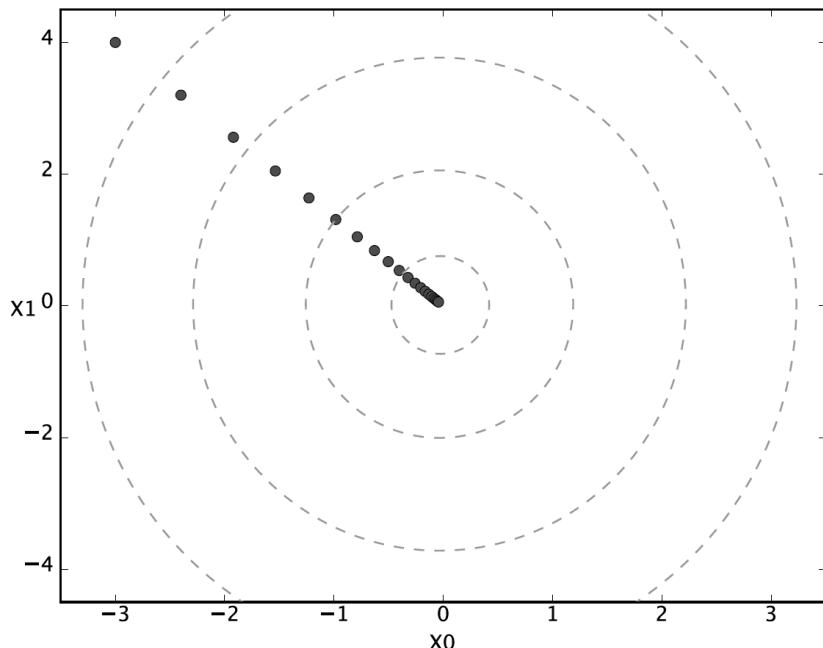
$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}$$

$$x_1 = x_1 - \eta \frac{\partial f}{\partial x_1}$$

- 식의 η 기호(eta, 에타)는 갱신하는 양을 나타냄
- 이를 신경망 학습에서는 **학습률**(learning rate)이라고 함
- 한번의 학습으로 얼마만큼 학습해야 할지, 즉 매개변수 값을 얼마나 갱신하느냐를 정하는 것임
- 이러한 갱신 단계를 여러 번 반복하면서 서서히 함수의 값을 줄임
- 변수가 2개보다 많은 경우도 같은 식(각 변수의 편미분 값)으로 갱신
- 또한 학습률 값은 0.01이나 0.0001 등 미리 특정 값으로 정해 두어야 함
- 이 값이 너무 크거나 작으면 '좋은 장소'를 찾아갈 수 없음

경사법(경사하강법)

- 문제 : 경사법으로 $f(x_0, x_1) = x_0^2 + x_1^2$ 의 최솟값을 구하라



- 학습률이 너무 큰 경우와 너무 작은 경우 실험해보기
- 학습률 같은 매개변수를 하이퍼파라미터(hyper parameter)라고 함
- 가중치와 편향 같은 신경망의 매개변수와는 성질이 다른 매개변수임
- 신경망의 가중치 매개변수는 훈련 데이터와 학습 알고리즘에 의해서 '자동'으로 획득되는 매개변수인 반면, 학습률 같은 하이퍼파라미터는 사람이 직접 설정해야 하는 매개변수임. 이 하이퍼파라미터는 시험을 통해 가장 잘 학습하는 값을 찾게 됨

신경망에서의 기울기

- 신경망 학습에서도 기울기를 구해야 함
- 여기서 말하는 기울기는 가중치 매개변수에 대한 손실 함수의 기울기
- 예를 들어 2×3 , 가중치가 W , 손실 함수가 L 인 신경망을 가정
- 이 경우 경사는 $\frac{\partial L}{\partial W}$ 로 나타낼 수 있음

$$W = \begin{pmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{pmatrix}$$

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{31}} \\ \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{32}} \end{pmatrix}$$

- $\frac{\partial L}{\partial W}$ 의 각 원소는 각각의 원소에 관한 편미분임. 따라서 형상도 W 와 같이 2×3

학습 알고리즘 구현하기

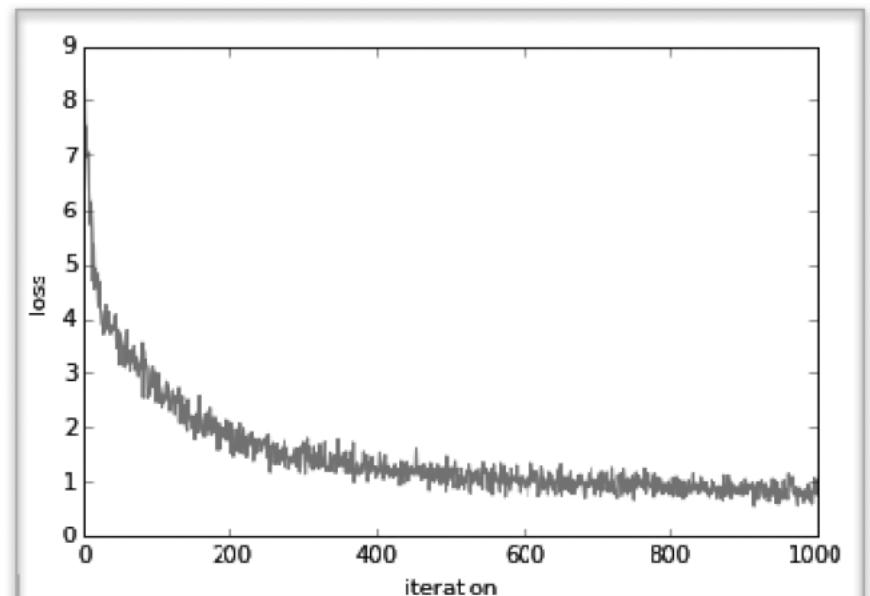
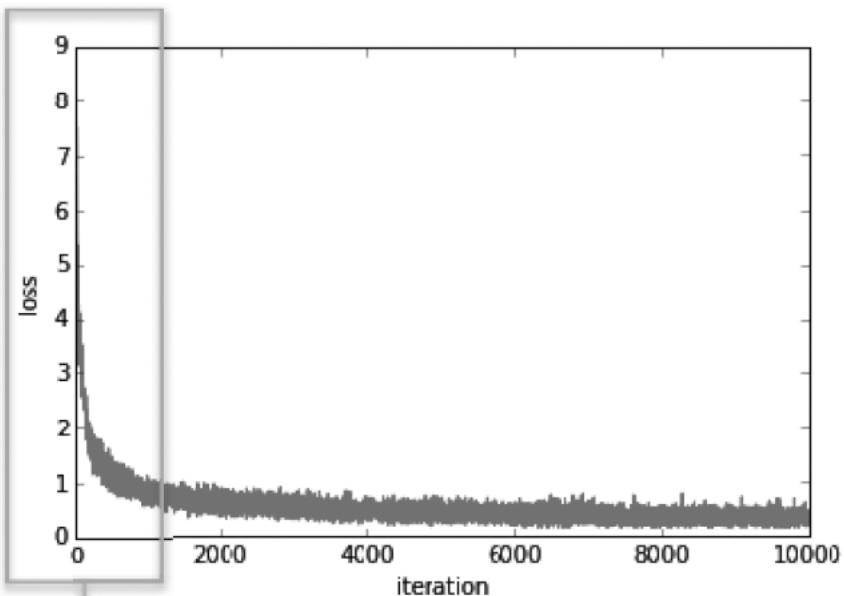
- 전제
 - 신경망에는 적응 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적용하도록 조정하는 과정을 '학습'이라 한다.
 - 신경망 학습은 다음과 같이 4 단계로 수행한다.
- 1 단계 - 미니배치
 - 훈련 데이터 중 일부를 무작위로 가져온다.
 - 이렇게 선별한 데이터를 미니배치라 하며, 그 미니배치의 손실함수 값을 줄이는 것이 목표.
- 2 단계 - 기울기 산출
 - 미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구한다. 기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시한다.
- 3 단계 - 매개변수 갱신
 - 가중치 매개변수를 기울기 방향으로 아주 조금 갱신한다.
- 4 단계 - 반복
 - 1 ~ 3 단계를 반복한다.

학습 알고리즘 구현하기

- 확률적 경사하강법
 - 데이터를 미지배치로 무작위로 선정하기 때문에 **확률적 경사하강법** (stochastic gradient descent, SGD) 이라고 부름
 - '확률적으로 무작위로 골라낸 데이터'에 대해 수행하는 경사 하강법이라는 의미

미니 배치 학습 구현하기

- 미니 배치 학습이란 훈련 데이터 중 일부를 무작위로 꺼내고 그 미니배치에 대해서 경사 하강법으로 매개변수를 갱신
- 미니배치 크기를 100으로 하여 확률적 경사 하강법을 수행해 매개변수를 갱신
- 갱신할 때마다 훈련 데이터에 대한 손실 함수를 계산하고 그 값을 배열에 추가



확대

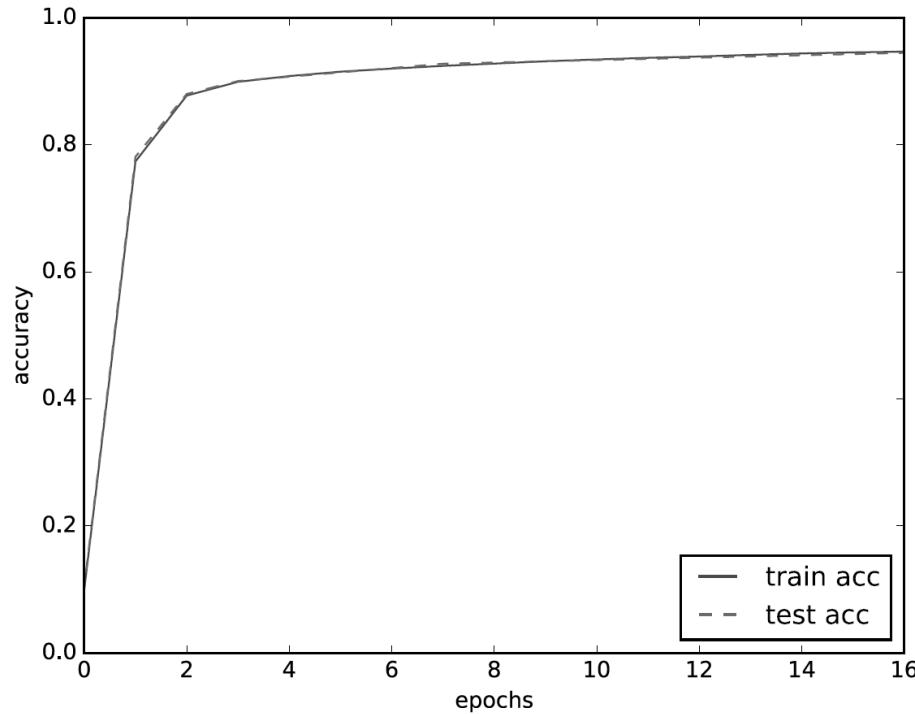
[그림] 손실 함수 값의 추이

시험 데이터로 평가하기

- 앞의 그림의 결과에서 손실 함수의 값이 서서히 내려가는 것을 확인
- 이 때의 손실 함수란 정확히는 '훈련 데이터의 미니배치에 대한 손실 함수'의 값
- 훈련 데이터의 손실 함수 값이 작아지는 것은 신경망이 잘 학습되고 있다는 방증
이지만, 이 결과만으로는 다른 데이터셋에도 비슷한 실력을 발휘할지 불투명
- 신경망 학습에서는 훈련 데이터 외의 데이터를 올바르게 인식하는지 확인
- 즉 '오버피팅'을 일으키지 않는지 확인해야 함
- 오버피팅 되었다는 것은 예를 들어 훈련 데이터에 포함된 이미지만 제대로 구분하고, 그렇지 않은 이미지는 식별할 수 없다는 뜻
- 신경망 학습의 원래 목표는 범용적인 능력을 익히는 것
- 범용 능력을 평가하려면 훈련 데이터에 포함되지 않는 데이터를 사용해 평가

- **에폭은 하나의 단위**
- 1 에폭은 학습에서 훈련데이터를 모두 소진했을 때의 횟수에 해당
- 예를 들어 데이터 10,000개를 100개의 미니배치로 학습할 경우, 확률적 경사 하강법을 100회 반복하면 모든 훈련 데이터를 '소진'. 이 경우 100회가 1 에폭

시험 데이터로 평가하기



[그림] 훈련 데이터와 시험 데이터에 대한 정확도 추이

- 훈련 데이터에 대한 정확도를 실선으로, 시험 데이터에 대한 정확도를 점선으로 표시. 에폭이 진행될 수록(학습이 진행될수록) 훈련 데이터와 시험 데이터를 사용하고 평가한 정확도가 모두 좋아짐
- 두 정확도에는 차이가 없음. 즉 오버피팅이 일어나지 않았음

정리

- 신경망
 - 기계학습에서 사용하는 데이터셋은 훈련 데이터와 시험 데이터로 나눠 사용
 - 훈련 데이터로 학습한 모델의 범용 능력을 시험 데이터로 평가
 - 신경망 학습은 손실 함수를 지표로, 손실 함수의 값이 작아지는 방향으로 가중치 매개변수를 갱신
 - 가중치 매개변수를 갱신할 때는 가중치 매개변수의 기울기를 이용하고, 기울어진 방향으로 가중치의 값을 갱신하는 작업을 반복
 - 아주 작은 값을 주었을 때의 차분으로 미분하는 것을 수치 미분이라고 함
 - 수치 미분을 이용해 가중치 매개변수의 기울기를 구할 수 있음
 - 수치 미분을 이용한 계산에는 시간이 걸리지만, 그 구현은 간단
 - 그에 비해 오차역전파법은 구현은 복잡하지만 기울기를 고속으로 구할 수 있음

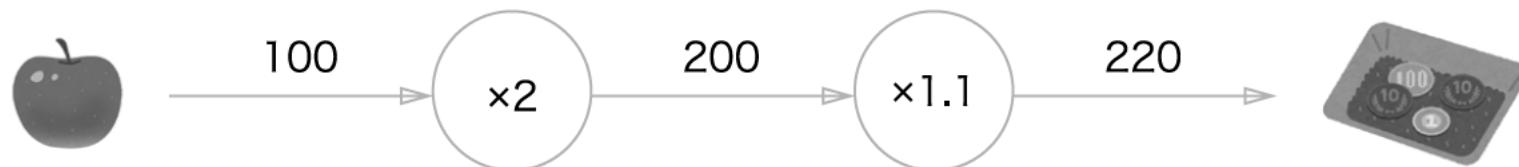
오차역전파법

계산 그래프로 풀다

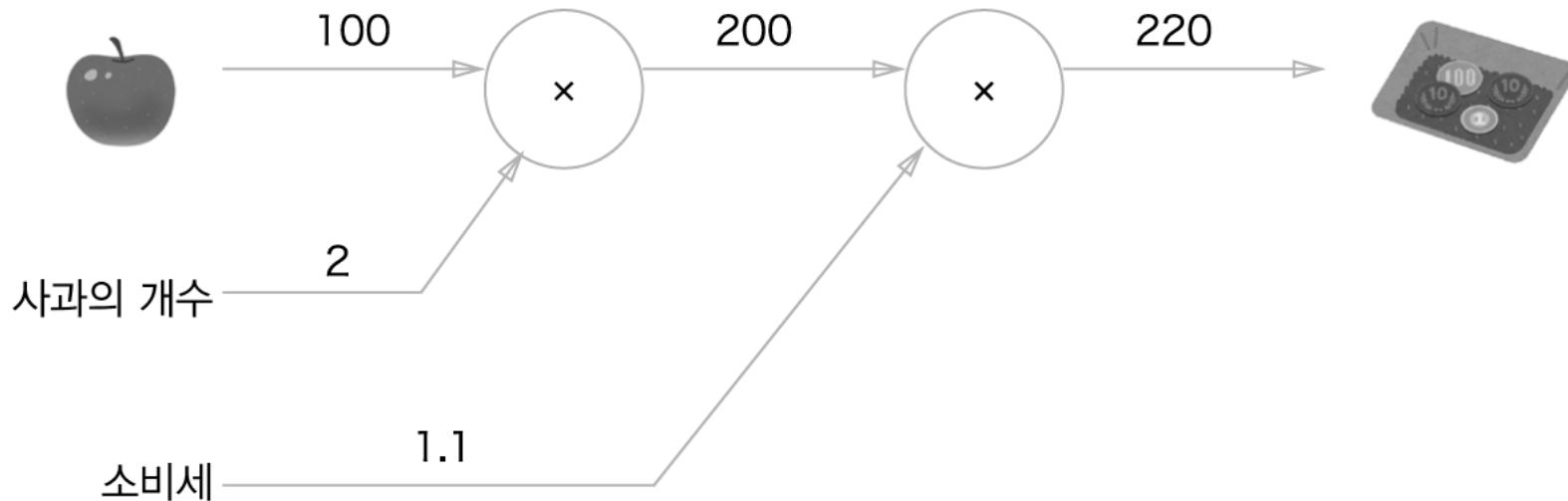
- 계산 그래프
 - 계산 그래프(computational graph)는 계산 과정을 그래프로 나타낸 것
 - 복수의 노드(node)와 에지(edge)로 표현됨
- 문제 1
 - 현빈 군은 슈퍼에서 1개에 100원인 사과를 2개 샀습니다. 이때 지불 금액을 구하세요. 단, 소비세가 10% 부과 됩니다.

계산 그래프로 풀다

- 계산 그래프로 풀어본 문제 1의 답



- 계산 그래프로 풀어본 문제 1의 답 : “사과의 개수”와 “소비세”를 변수로 취급



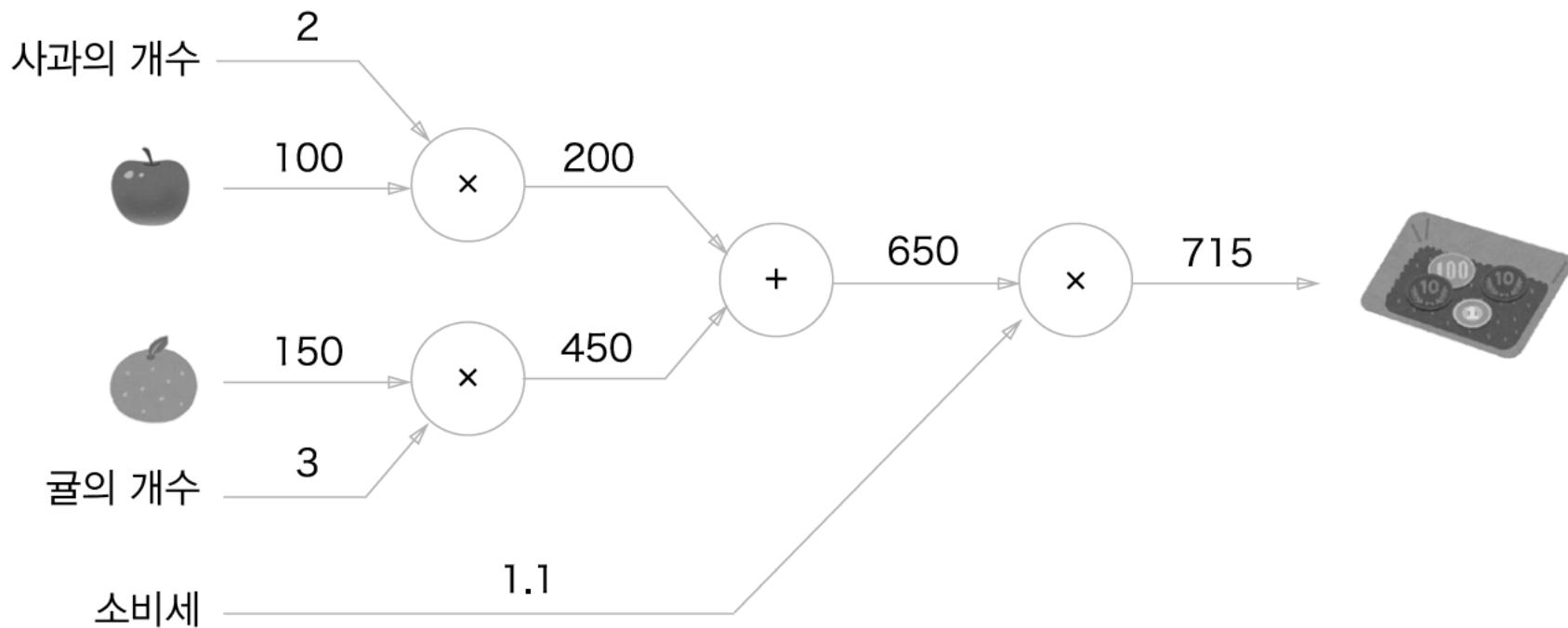
계산 그래프로 풀다

- 문제 2
 - 현빈 군은 슈퍼에서 사과를 2개, 귤을 3개 샀습니다. 사과는 1개에 100원, 귤은 1개 150원 입니다. 소비세가 10%일 때 지불 금액을 구하세요.

계산 그래프로 풀다

- 문제 2

- 현빈 군은 슈퍼에서 사과를 2개, 귤을 3개 샀습니다. 사과는 1개에 100원, 귤은 1개 150원입니다. 소비세가 10%일 때 지불 금액을 구하세요.

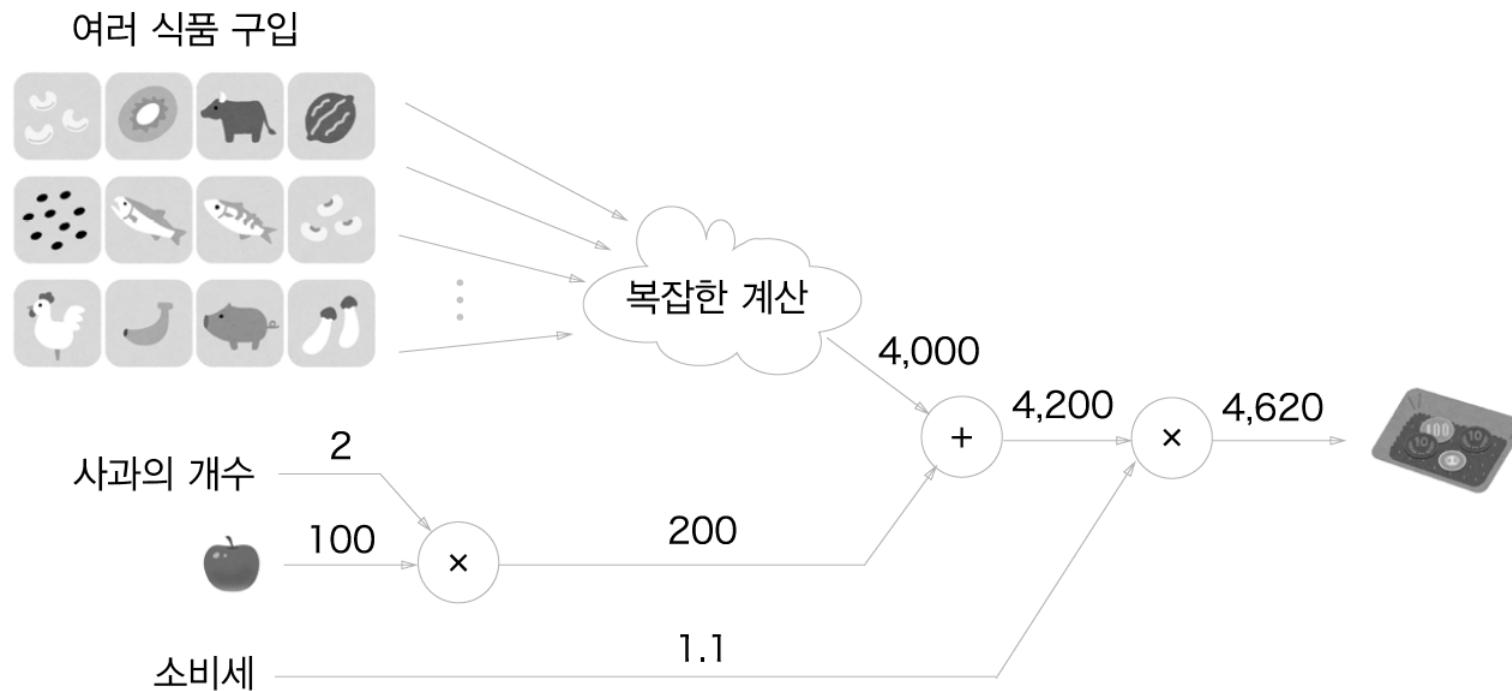


계산 그래프로 풀다

- 계산 그래프를 이용한 문제풀이
 1. 계산 그래프를 구성한다.
 2. 그래프에서 계산을 왼쪽에서 오른쪽으로 진행한다.
- 순전파와 역전파
 - '계산을 왼쪽에서 오른쪽으로 진행'하는 단계를 **순전파**(forward propagation)
 - 반대 방향의 전파를 **역전파**(backward propagation)라고 함
 - 역전파는 미분을 계산할 때 중요한 역할

국소적 계산

- 계산 그래프의 특징
 - '국소적 계산'을 전파함으로써 최종 결과를 얻게 됨
 - 국소적이란 '자신과 직접 관계된 작은 범위'라는 뜻
 - 국소적 계산은 결국 전체에서 어떤 일이 벌어지든 상관없이 자신과 관계된 정보만으로 결과를 출력할 수 있다는 것
- 사과 2개를 포함해 여러 식품을 구입하는 예

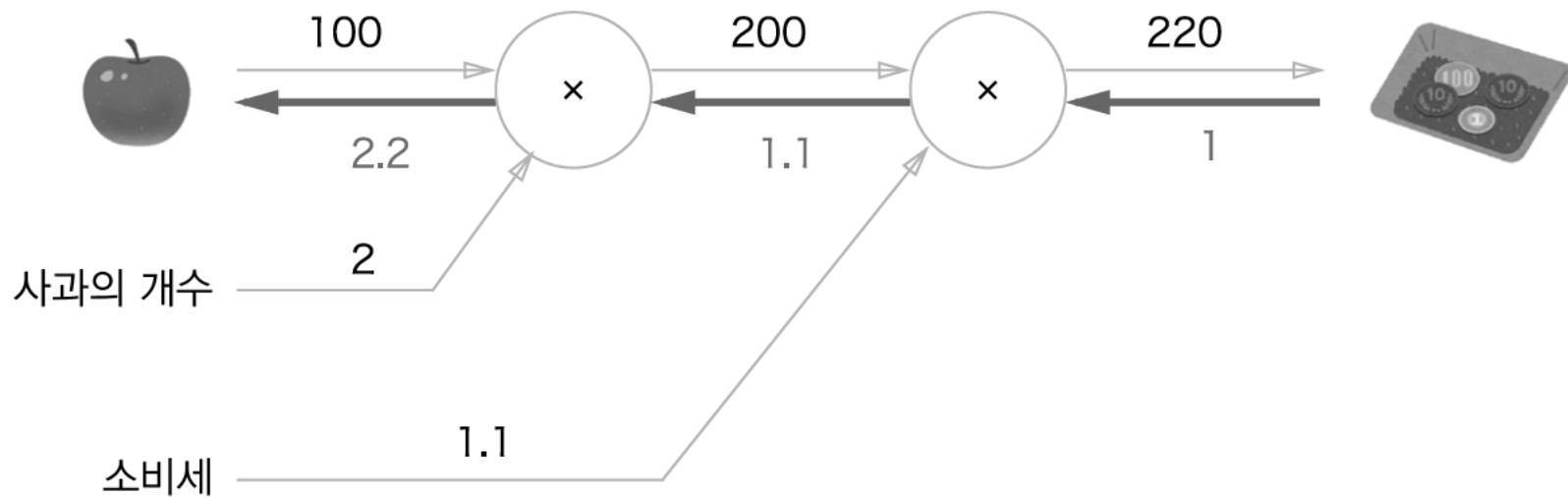


왜 계산 그래프로 푸는가?

- 계산 그래프의 이점
 - 전체가 아무리 복잡해도 각 노드에서는 단순한 계산에 집중하여 문제를 단순화
 - 중간 계산 결과를 모두 보관할 수 있음
 - 역전파를 통해 '미분'을 효율적으로 계산할 수 있음
- 역전파의 이해
 - 문제 1은 사과를 2개 사서 소비세를 포함한 최종 금액을 구하는 것
 - 이 때 사과 가격이 오르면 최종 금액에 어떤 영향을 끼치는지를 알고 싶다고 가정
 - 이는 '사과 가격에 대한 지불 금액의 미분'을 구하는 문제
 - 기호로 사과 값을 x , 지불 금액은 L 이라 했을 때 $\frac{\partial L}{\partial x}$ 를 구하는 것임
 - 이 미분 값은 사과 값이 '아주 조금' 올랐을 때 지불 금액이 얼마자 증가하느냐를 표시한 것

왜 계산 그래프로 푸는가?

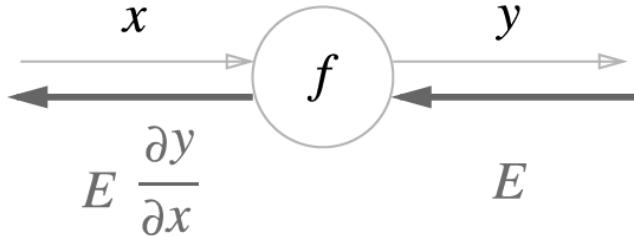
- 역전파에 의한 미분 값의 전달
 - '사과 가격에 대한 지불 금액의 미분'



- 사과가 1원 오르면 최종 금액은 2.2원 오른다는 뜻
- (정확히는 사과 값이 아주 조금 오르면 최종 금액은 그 아주 작은 값의 2.2배만큼 오른다는 뜻)

연쇄법칙

- 계산 그래프의 역전파
 - $y=f(x)$ 라는 계산의 역전파



- 역전파의 계산 절차는 신호 E 에 노드의 국소적 미분($\frac{\partial y}{\partial x}$)을 곱한 후 다음 노드로 전달

연쇄법칙

- 연쇄법칙이란?

- 합성 함수란 여러 함수로 구성된 함수
- 예를 들어 $z = (x + y)^2$ 이라는 식은 아래처럼 두 개의 식으로 구성

$$z = t^2$$

$$t = x + y$$

- 연쇄법칙의 원리 : 합성 함수의 미분에 대한 성질

합성 함수의 미분은 합성 함수를 구성하는 각 함수의 미분의 곱으로 나타낼 수 있다

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \frac{\partial t}{\partial x}$$

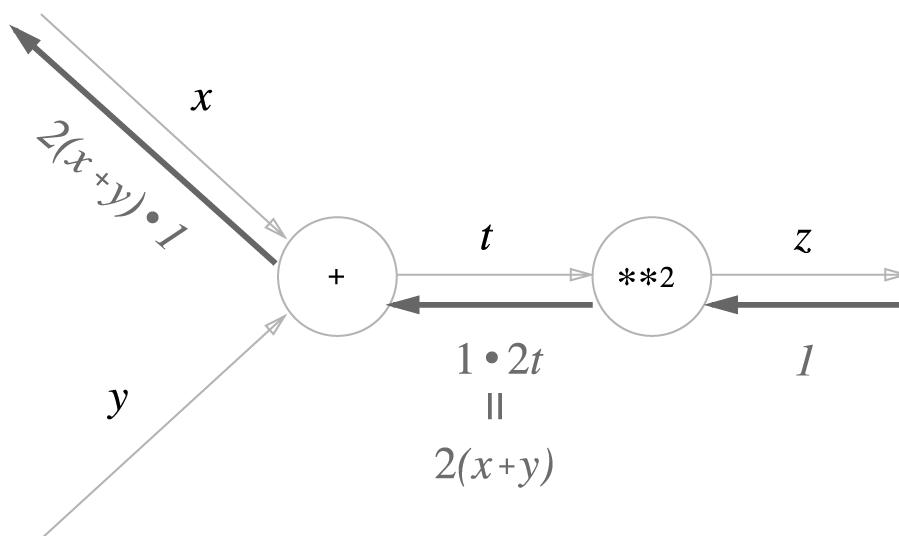
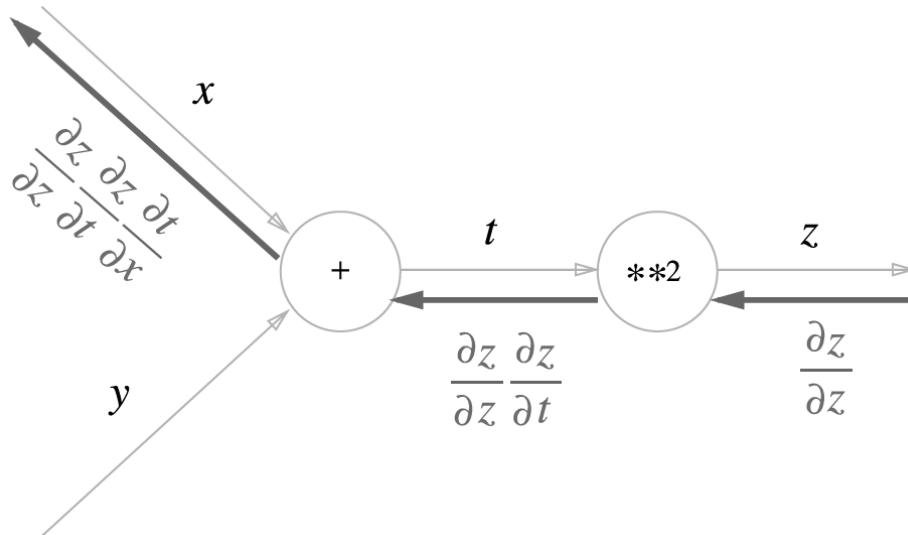
$$\frac{\partial z}{\partial t} = 2t$$

$$\frac{\partial t}{\partial x} = 1$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} = 2t \cdot 1 = 2(x + y)$$

연쇄법칙

- 연쇄법칙과 계산 그래프



역전파

- 덧셈 노드의 역전파

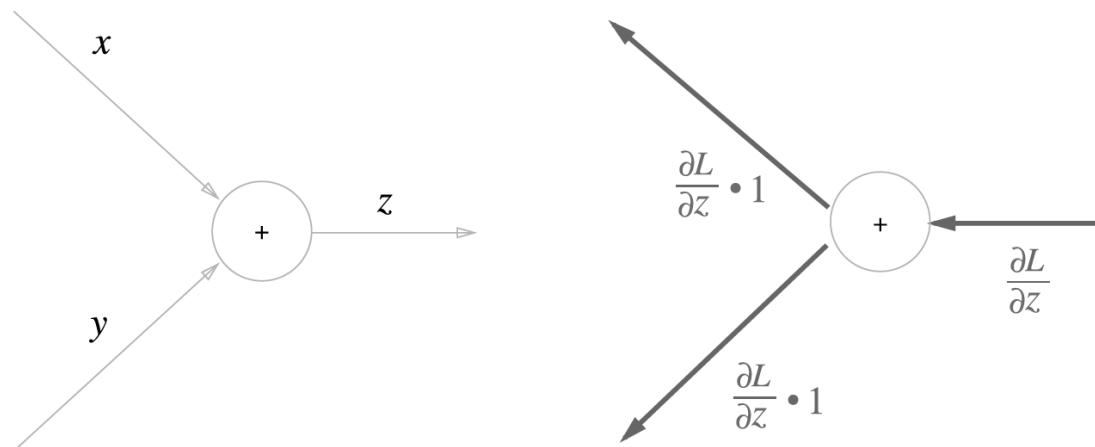
- $z = x+y$ 의 미분

$$\frac{\partial z}{\partial x} = 1$$

$$\frac{\partial z}{\partial y} = 1$$

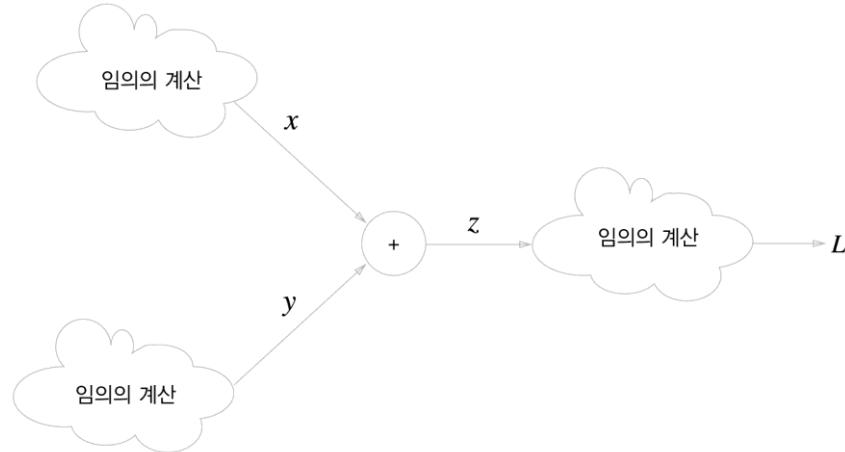
- 덧셈 노드의 역전파

: 1을 곱하기만 할뿐이므로 입력된 값을 그대로 다음노드로 내보냄

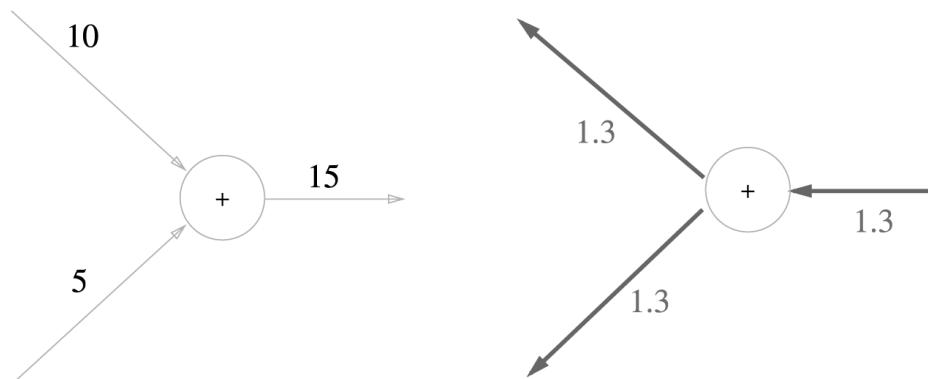


역전파

- 덧셈 노드의 역전파
 - 최종 출력으로 가는 계산의 중간에 덧셈 노드가 존재



- 덧셈 노드 역전파의 구체적인 예



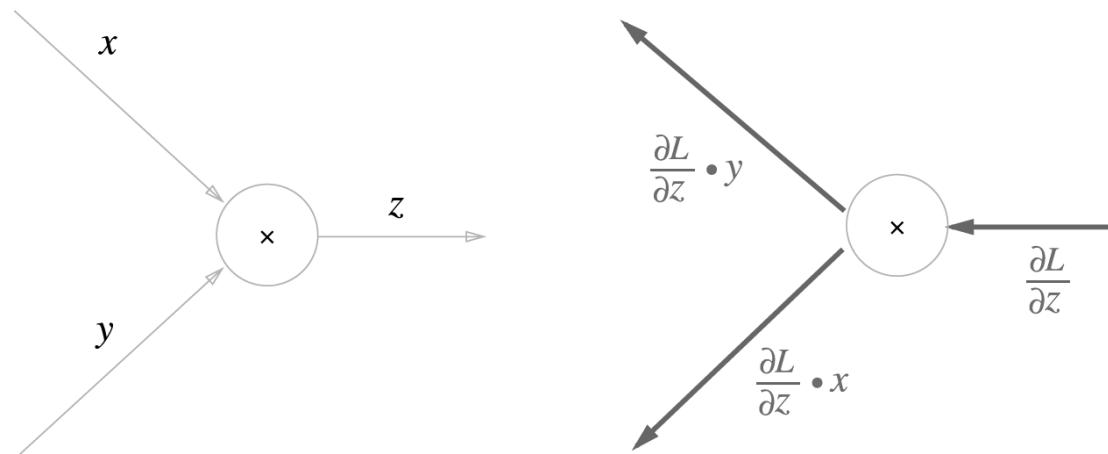
역전파

- 곱셈 노드의 역전파
 - $z = xy$ 의 미분

$$\frac{\partial z}{\partial x} = y$$

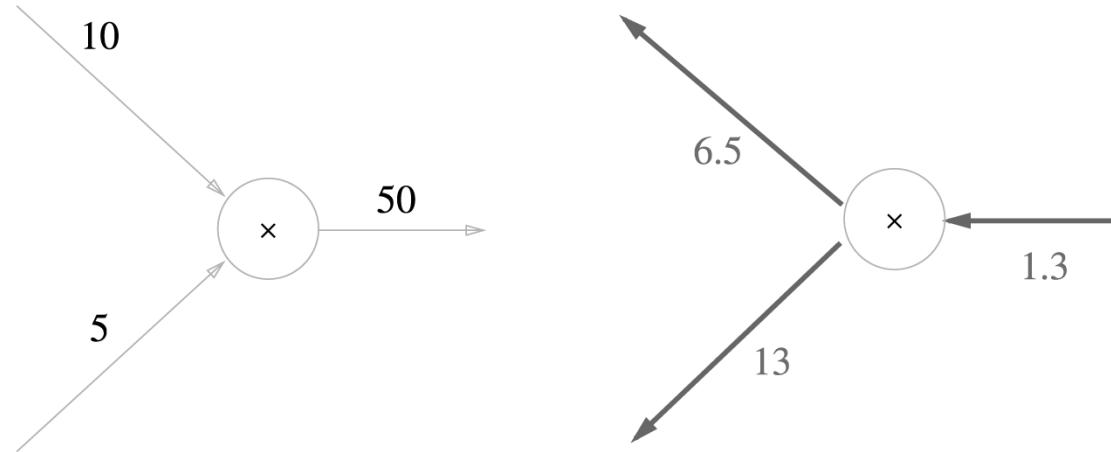
$$\frac{\partial z}{\partial y} = x$$

- 곱셈 노드 역전파:
: 상류의 값에 순전파 때의 입력 신호들을 '서로 바꾼 값'을 곱해서 하류로 보냄



역전파

- 곱셈 노드의 역전파
 - 곱셈 노드 역전파의 구체적인 예

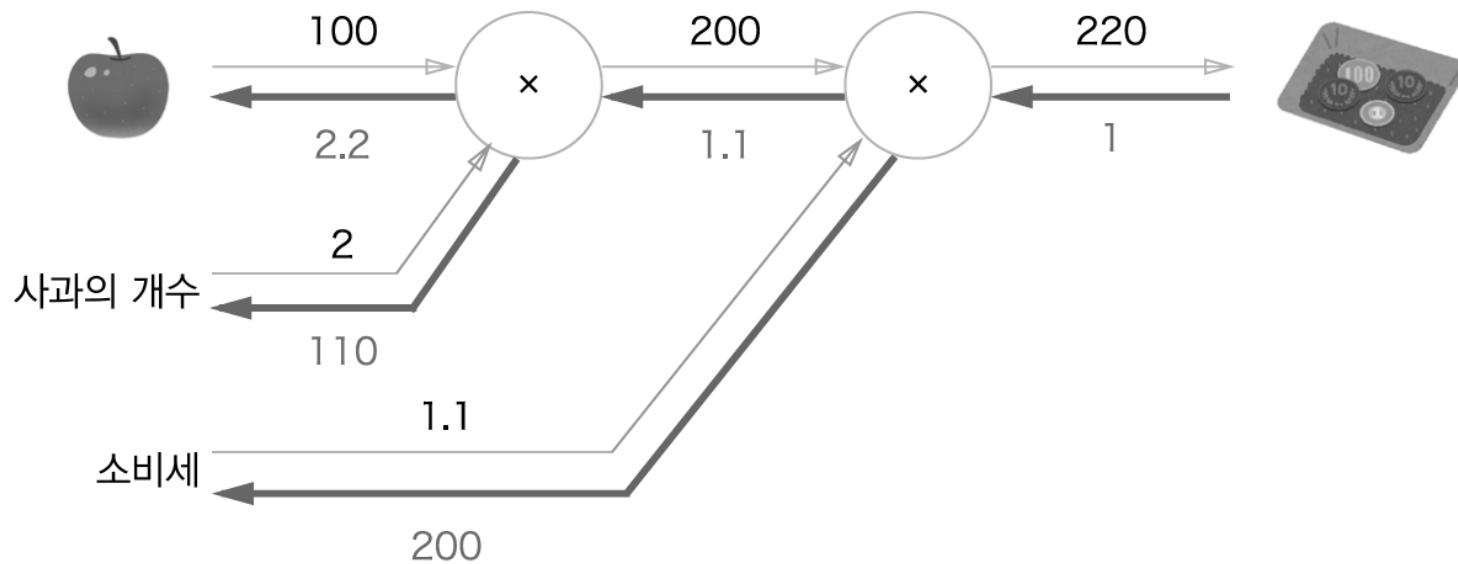


- 역전파 때 상류에서 1.3이 흘러온다고 가정
- 곱셈의 역전파에서는 입력 신호를 바꾼 값을 곱하여 하나는 $1.3 \times 5 = 6.5$, 다른 하나는 $1.3 \times 10 = 13$ 이 됨
- 덧셈의 역전파에서는 상류의 값을 그대로 흘러보내서 순방향 입력 시동의 값은 필요하지 않았으나 곱셈의 역전파는 순방향 입력 신호의 값이 필요. 따라서 곱셈 노드를 구현할 때는 순전파의 입력 신호를 변수에 저장

역전파

- 사과 쇼핑의 예

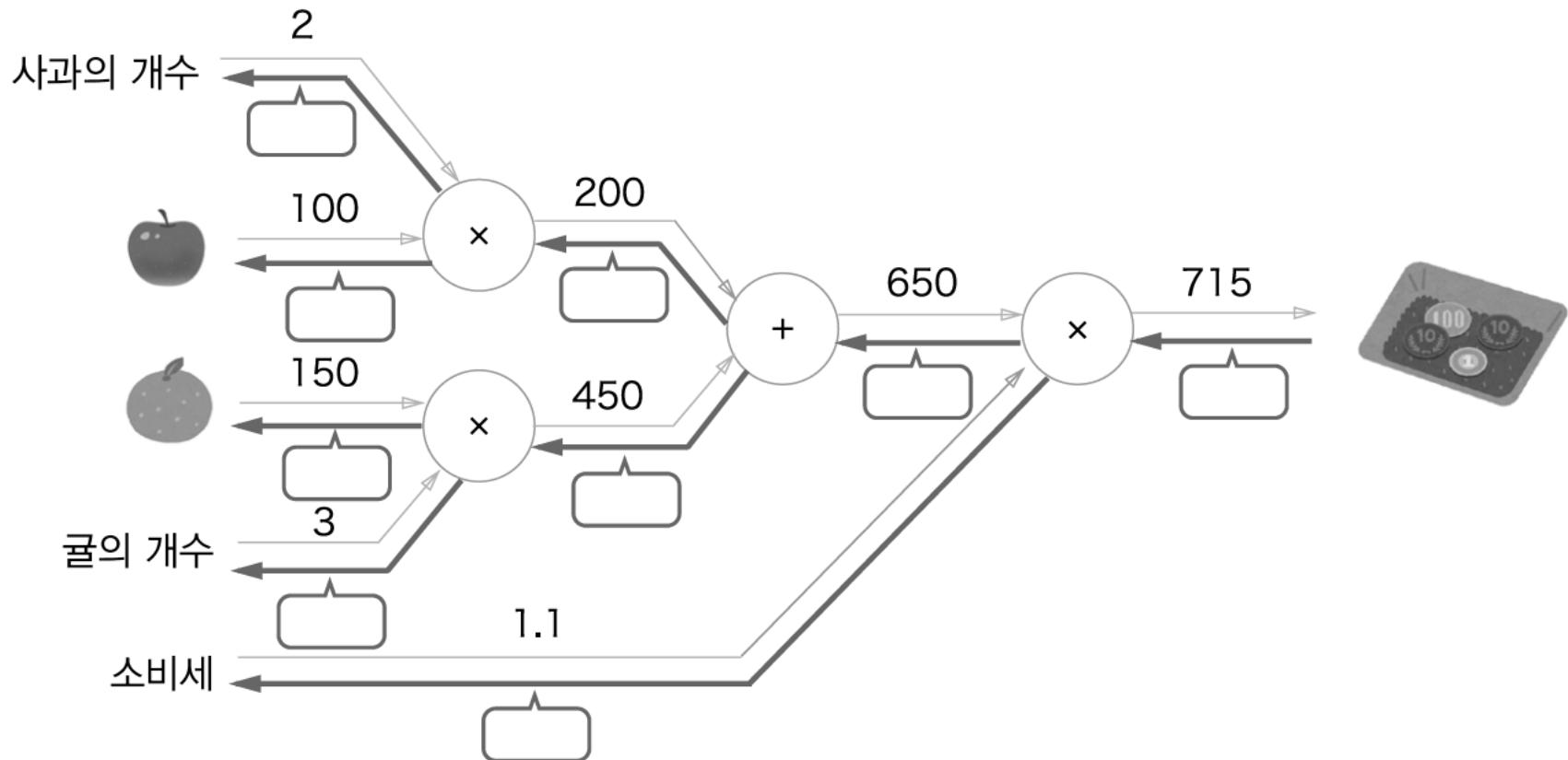
- 사과의 가격, 사과의 개수, 소비세라는 세 변수 각각이 최종 금액에 어떤 영향을 주는지 풀어보기



- 사과 가격의 미분은 2.2, 사과 개수의 미분은 110, 소비세의 미분은 200
 - 이는 소비세와 사과 가격이 같은 양만큼 오르면 최종 금액에는 소비세가 200의 크기로, 사과 가격이 2.2 크기로 영향을 준다고 해석할 수 있음
(단위 다름 주의 : 소비세 1은 100%, 사과 가격 1은 1원)

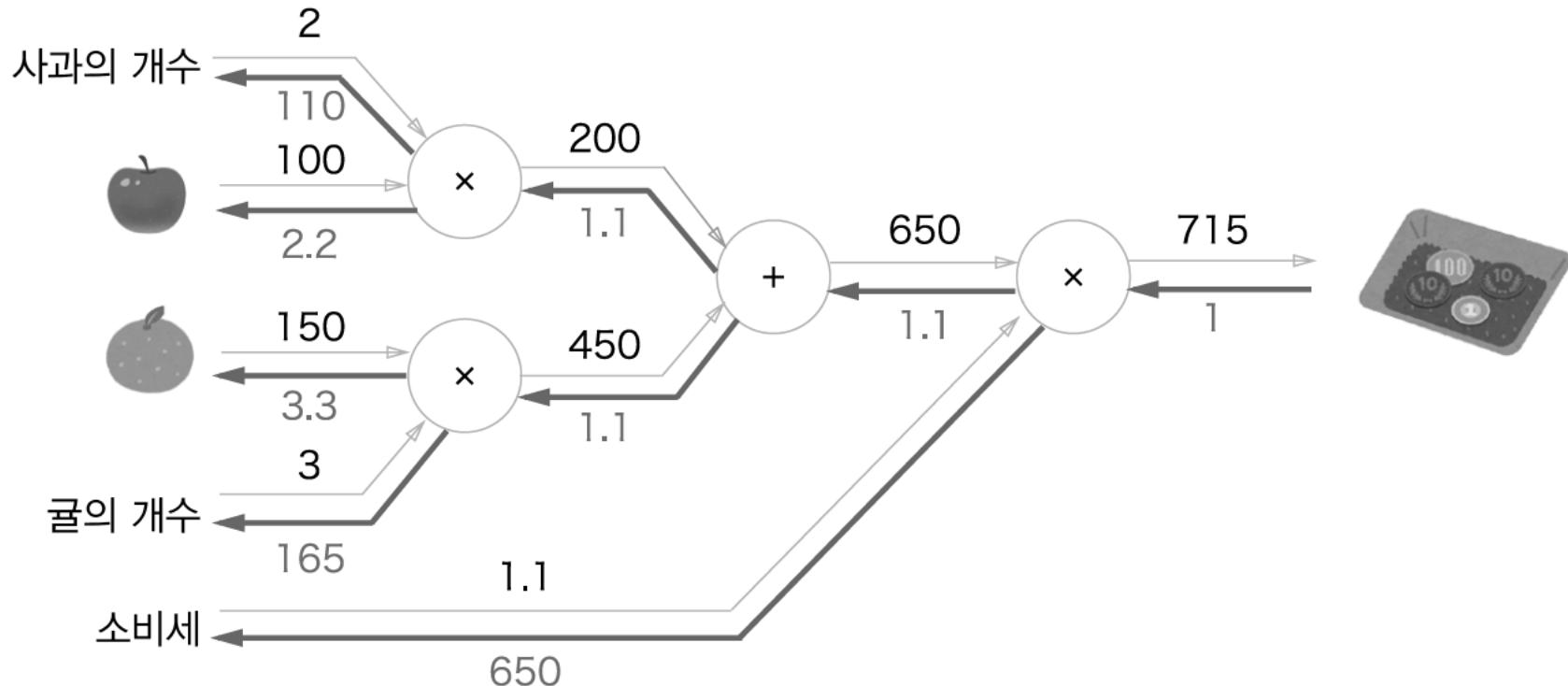
역전파

- 사과와 귤 쇼핑의 역전파 예



역전파

- 사과와 귤 쇼핑의 역전파 예



단순한 계층 구현하기

- 곱셈 계층

```
class MulLayer:  
    def __init__(self):  
        self.x = None  
        self.y = None  
  
    def forward(self, x, y):  
        self.x = x  
        self.y = y  
        out = x * y  
  
        return out  
  
    def backward(self, dout):  
        dx = dout * self.y # x와 y를 바꾼다.  
        dy = dout * self.x  
  
        return dx, dy
```

단순한 계층 구현하기

- 곱셈 계층

```
apple = 100
apple_num = 2
tax = 1.1

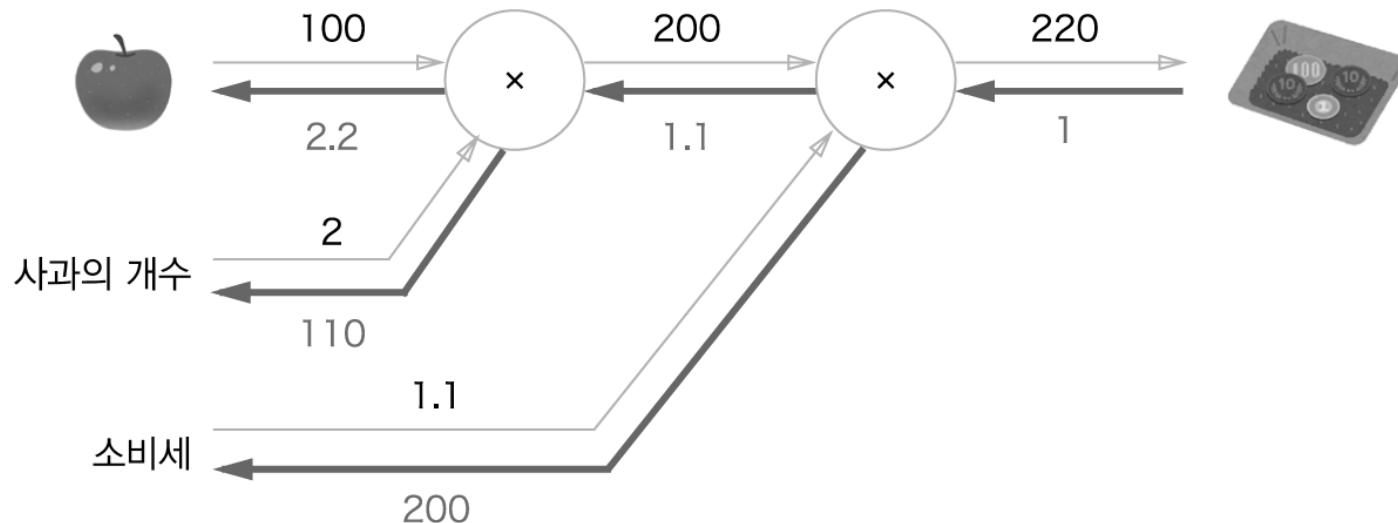
mul_apple_layer = MulLayer()
mul_tax_layer = MulLayer()

# forward
apple_price = mul_apple_layer.forward(apple, apple_num)
price = mul_tax_layer.forward(apple_price, tax)

# backward
dprice = 1
dapple_price, dtax = mul_tax_layer.backward(dprice)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)
```

단순한 계층 구현하기

- 곱셈 계층



```
print("price:", int(price))
print("dApple:", dapple)
print("dApple_num:", int(dapple_num))
print("dTax:", dtax)
```

price: 220
dApple: 2.2
dApple_num: 110
dTax: 200

단순한 계층 구현하기

- 덧셈 계층

```
class AddLayer:  
    def __init__(self):  
        pass  
  
    def forward(self, x, y):  
        out = x + y  
  
        return out  
  
    def backward(self, dout):  
        dx = dout * 1  
        dy = dout * 1  
  
        return dx, dy
```

단순한 계층 구현하기

- 덧셈 계층

```
# layer
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()

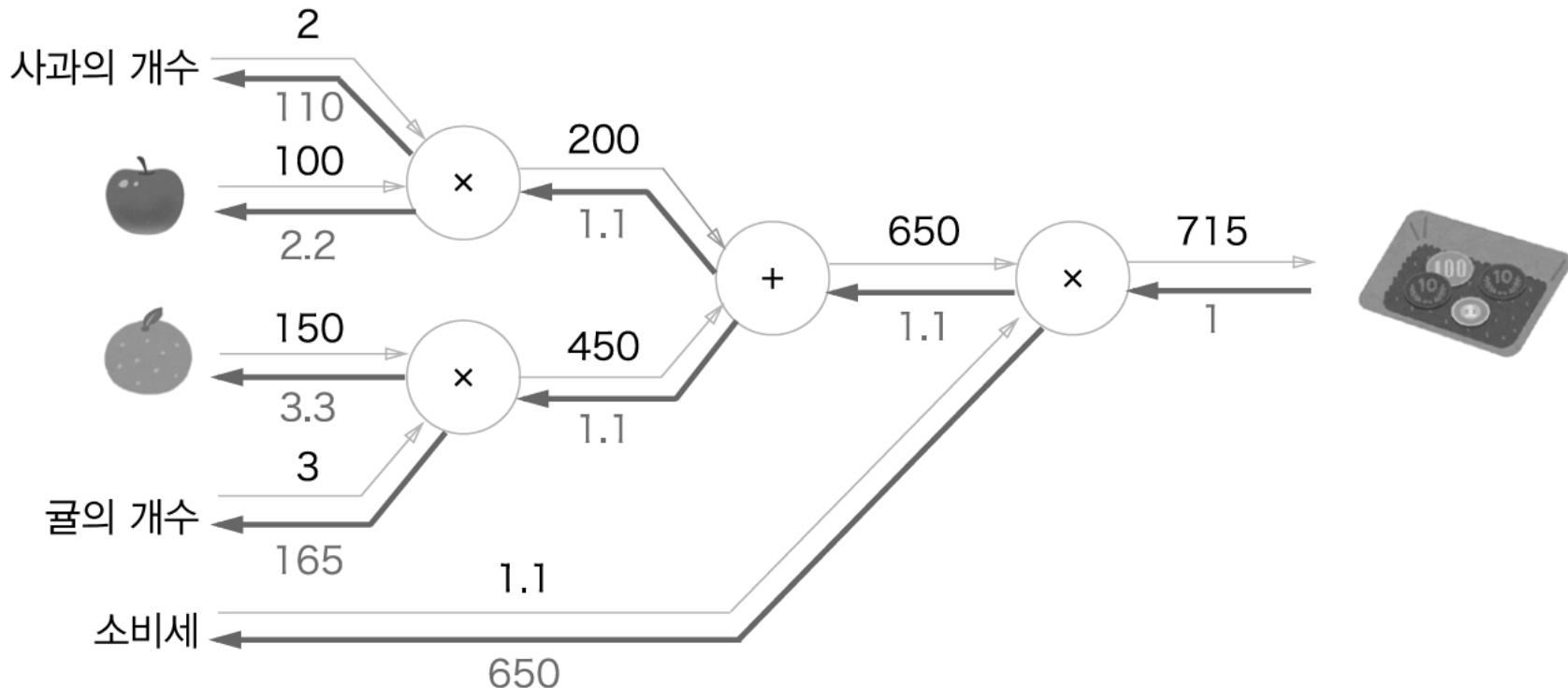
# forward
apple_price = mul_apple_layer.forward(apple, apple_num) # (1)
orange_price = mul_orange_layer.forward(orange, orange_num) # (2)
all_price = add_apple_orange_layer.forward(apple_price, orange_price) # (3)
price = mul_tax_layer.forward(all_price, tax) # (4)

# backward
dprice = 1
dall_price, dtax = mul_tax_layer.backward(dprice) # (4)
dapple_price, dorange_price = add_apple_orange_layer.backward(dall_price) # (3)
dorange, dorange_num = mul_orange_layer.backward(dorange_price) # (2)
dapple, dapple_num = mul_apple_layer.backward(dapple_price) # (1)
```

```
apple = 100
apple_num = 2
orange = 150
orange_num = 3
tax = 1.1
```

단순한 계층 구현하기

- 덧셈 계층



단순한 계층 구현하기

- 덧셈 계층

```
print("price:", int(price))
print("dApple:", dapple)
print("dApple_num:", int(dapple_num))
print("dOrange:", dorange)
print("dOrange_num:", int(dorange_num))
print("dTax:", dtax)
```

price: 715
dApple: 2.2
dApple_num: 110
dOrange: 3.3000000000000003
dOrange_num: 165
dTax: 650

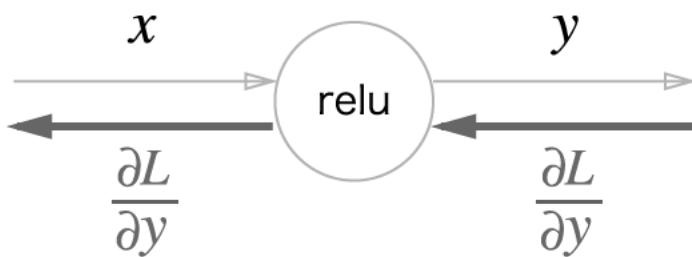
활성화 함수 계층 구현하기

- ReLU 계층
 - ReLU 수식

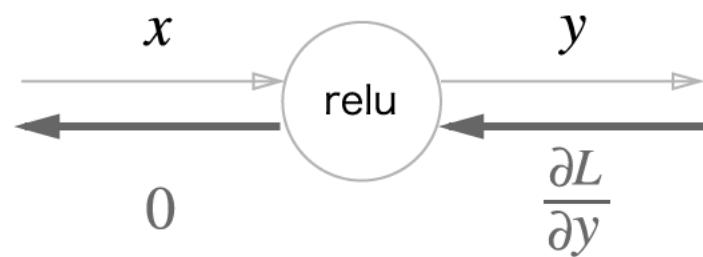
$$y = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

- x 에 대한 y 의 미분
- ReLU 계층의 계산 그래프

$$x > 0$$



$$x \leq 0$$



활성화 함수 계층 구현하기

- ReLU 계층

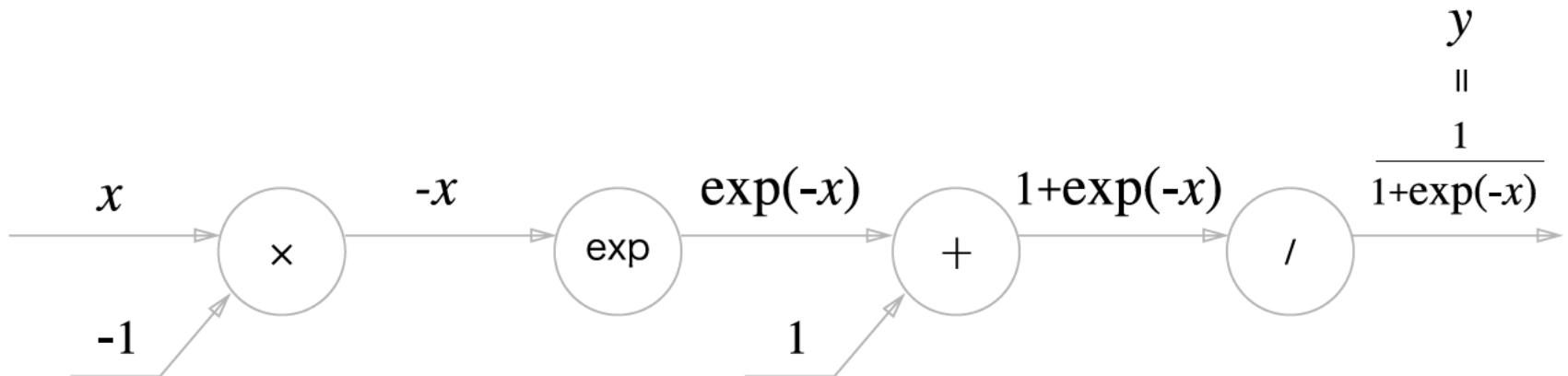
```
class Relu:  
    def __init__(self):  
        self.mask = None  
  
    def forward(self, x):  
        self.mask = (x <= 0)  
        out = x.copy()  
        out[self.mask] = 0  
  
        return out  
  
    def backward(self, dout):  
        dout[self.mask] = 0  
        dx = dout  
  
        return dx
```

활성화 함수 계층 구현하기

- Sigmoid 계층
 - Sigmoid 수식

$$y = \frac{1}{1 + \exp(-x)}$$

- Sigmoid 계층의 계산 그래프(순전파)



활성화 함수 계층 구현하기

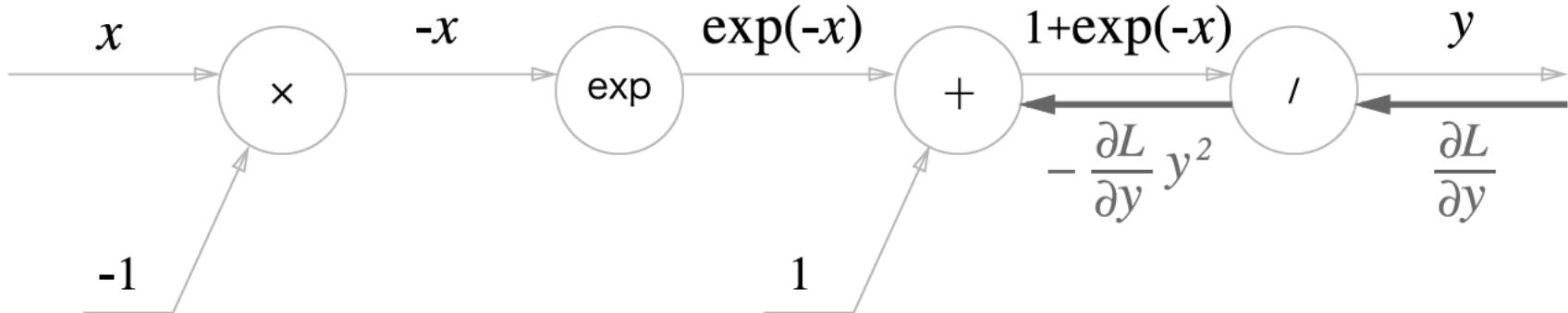
- Sigmoid 계층

- '/' 노드, 즉 $y = 1/x$ 을 미분한 식

$$\frac{\partial y}{\partial x} = -\frac{1}{x^2}$$

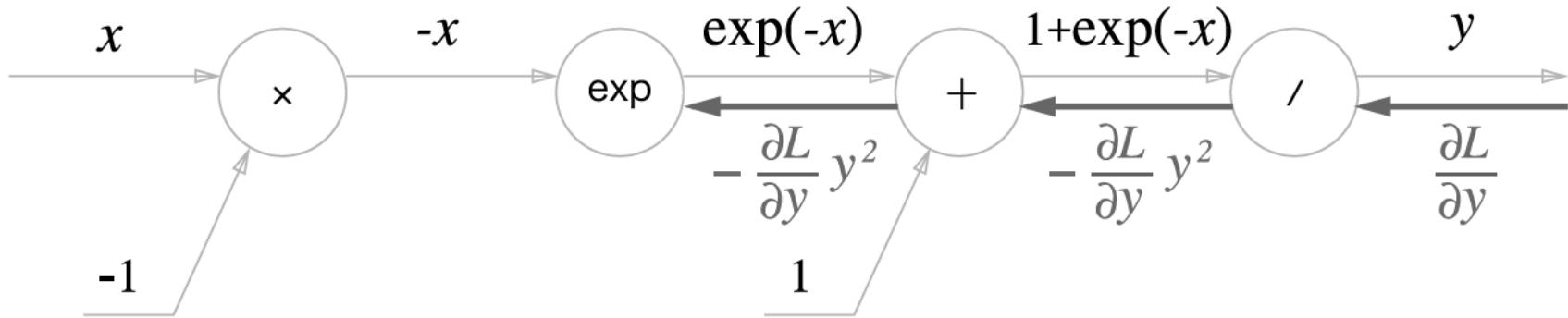
$$= -y^2$$

- Sigmoid 계층의 계산 그래프(역전파 흐름 1단계)



활성화 함수 계층 구현하기

- Sigmoid 계층
 - '+' 노드는 상류의 값을 여과없이 하류로 보냄
 - Sigmoid 계층의 계산 그래프(역전파 흐름 2단계)



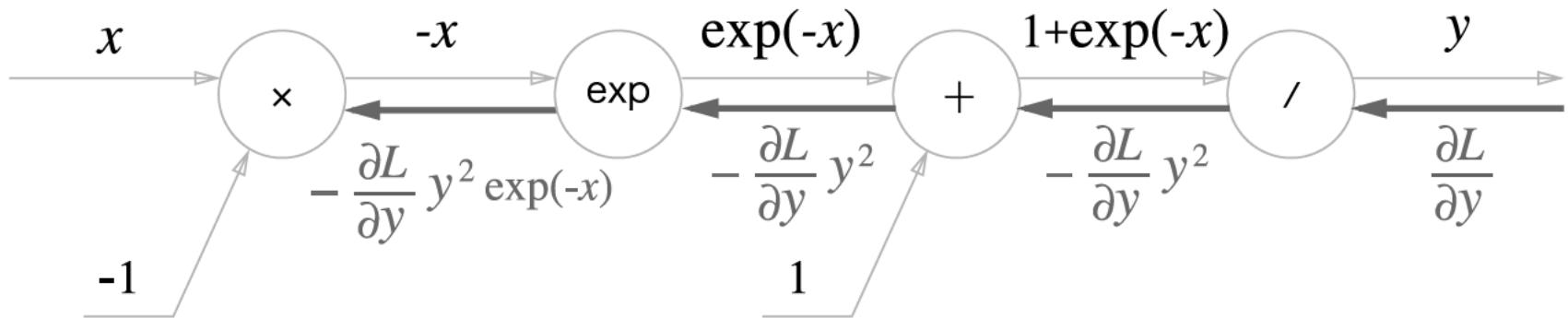
활성화 함수 계층 구현하기

- Sigmoid 계층

- 'exp' 노드, $y = \exp(x)$ 을 미분한 식

$$\frac{\partial y}{\partial x} = \exp(x)$$

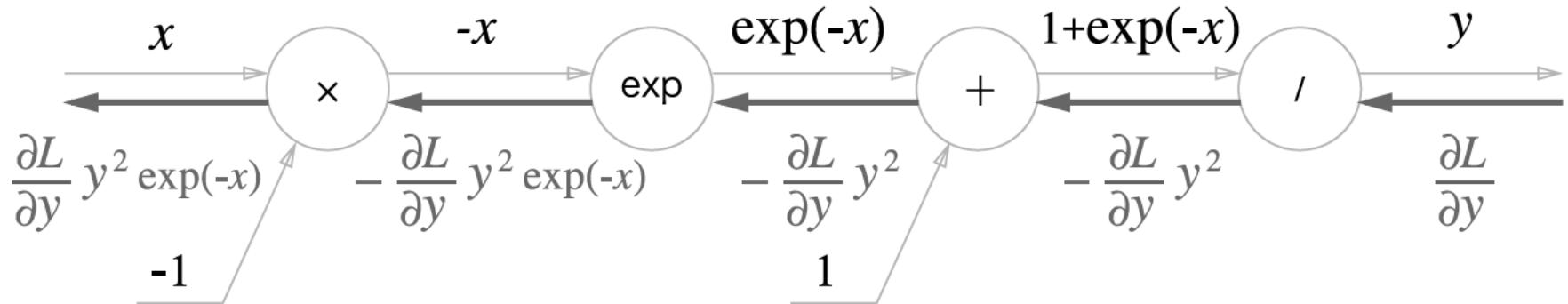
- Sigmoid 계층의 계산 그래프(역전파 흐름 3단계)



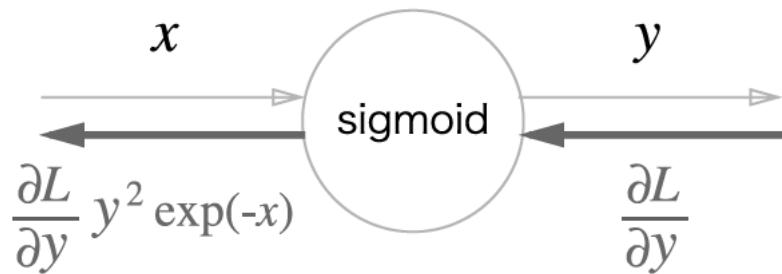
활성화 함수 계층 구현하기

- Sigmoid 계층

- 'X' 노드는 순전파 때의 값을 '서로 바꿔' 곱함
- Sigmoid 계층의 계산 그래프(역전파 흐름 4단계)



- Sigmoid 계층의 계산 그래프(간소화 버전)



- 최종 출력을 순전파의 입력 x 와 출력 y 만으로 계산할 수 있음

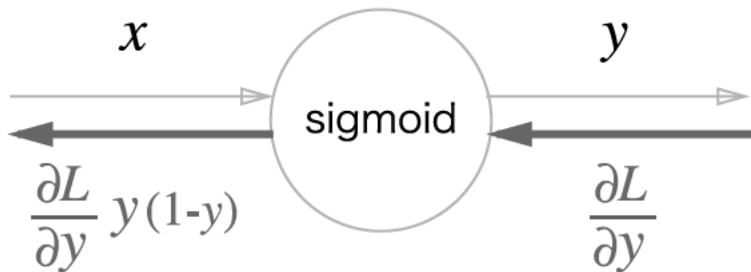
활성화 함수 계층 구현하기

- Sigmoid 계층

- 역전파의 최종 출력을 다음처럼 정리할 수 있음

$$\begin{aligned}\frac{\partial L}{\partial y} y^2 \exp(-x) &= \frac{\partial L}{\partial y} \frac{1}{(1 + \exp(-x))^2} \exp(-x) \\ &= \frac{\partial L}{\partial y} \frac{1}{1 + \exp(-x)} \frac{\exp(-x)}{1 + \exp(-x)} \\ &= \frac{\partial L}{\partial y} y(1-y)\end{aligned}$$

- 즉, Sigmoid 계층의 역전파는 순전파의 출력(y)만으로 계산할 수 있음



활성화 함수 계층 구현하기

- Sigmoid 계층

```
class Sigmoid:  
    def __init__(self):  
        self.out = None  
  
    def forward(self, x):  
        out = sigmoid(x)  
        self.out = out  
        return out  
  
    def backward(self, dout):  
        dx = dout * (1.0 - self.out) * self.out  
  
        return dx
```

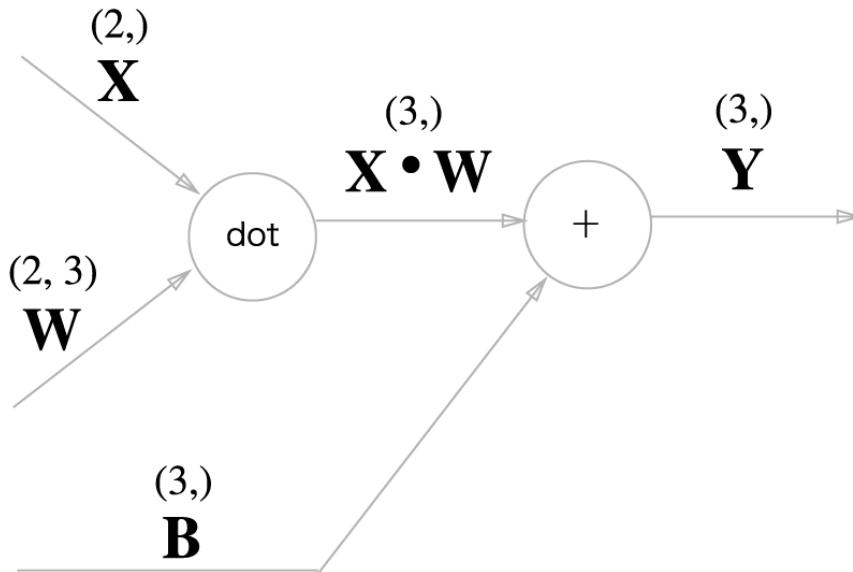
Affine/Softmax 계층 구현하기

- Affine 계층
 - 행렬의 곱에서는 대응하는 차원의 원소 수를 일치시킴

$$\begin{matrix} \mathbf{X} & \bullet & \mathbf{W} & = & \mathbf{O} \\ (2,) & & (2, 3) & & (3,) \\ \text{일치} & & \text{일치} & & \text{일치} \end{matrix}$$

- Note. 신경망의 순전파 때 수행하는 행렬의 곱은 기하학에서는 **어파인 변환** (Affine transformation)이라고 함

- Affine 계층의 계산 그래프 (X, W, B 가 행렬임에 주의)



Affine/Softmax 계층 구현하기

- Affine 계층
 - Affine 계층의 역전파

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \cdot \mathbf{W}^T$$

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^T \cdot \frac{\partial L}{\partial \mathbf{Y}}$$

- \mathbf{W}^T 의 T는 전치행렬. 전치행렬은 W의 (i, j) 위치의 원소를 (j,i) 위치로 바꾼것

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{pmatrix}$$

$$\mathbf{W}^T = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{pmatrix}$$

Affine/Softmax 계층 구현하기

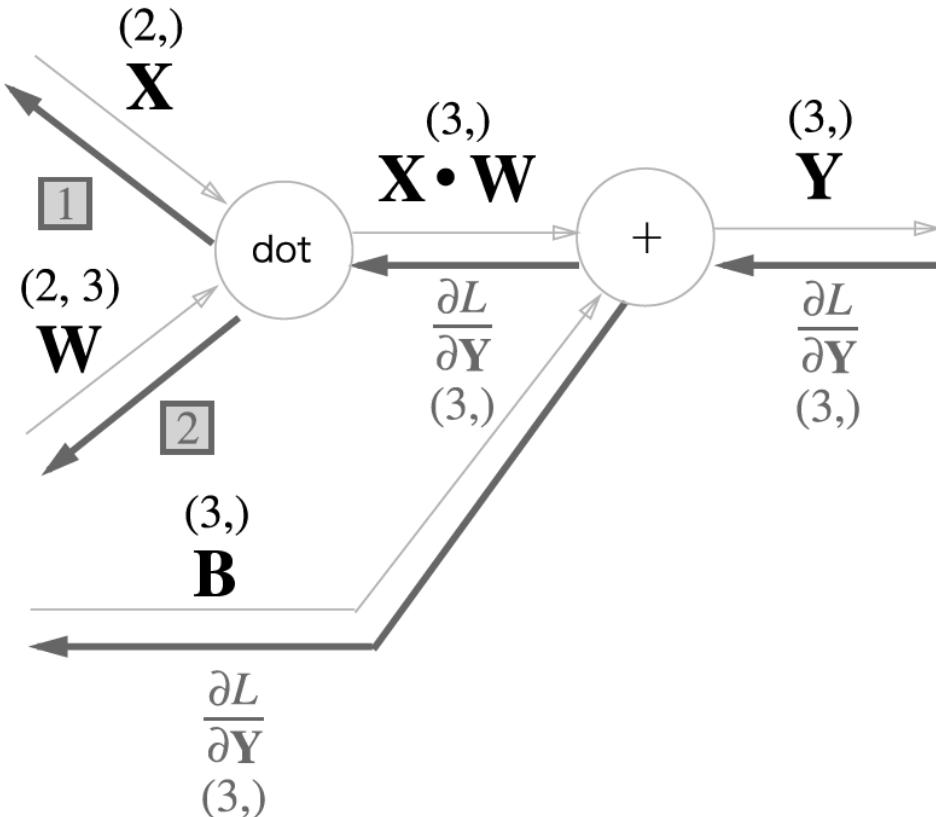
- Affine 계층
 - Affine 계층의 역전파

[1] $\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \mathbf{W}^T$

(2,) (3,) (3, 2)

[2] $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^T \frac{\partial L}{\partial \mathbf{Y}}$

(2, 3) (2, 1) (1, 3)



- \mathbf{X} 와 $\frac{\partial L}{\partial \mathbf{X}}$ 은 같은 형상, \mathbf{W} 와 $\frac{\partial L}{\partial \mathbf{W}}$ 도 같은 형상

$$\mathbf{X} = (x_0, x_1, \dots, x_n)$$

$$\frac{\partial L}{\partial \mathbf{X}} = \left(\frac{\partial L}{\partial x_0}, \frac{\partial L}{\partial x_1}, \dots, \frac{\partial L}{\partial x_n} \right)$$

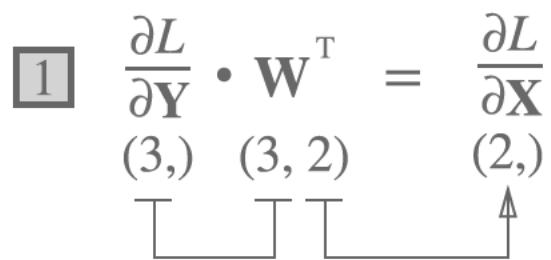
Affine/Softmax 계층 구현하기

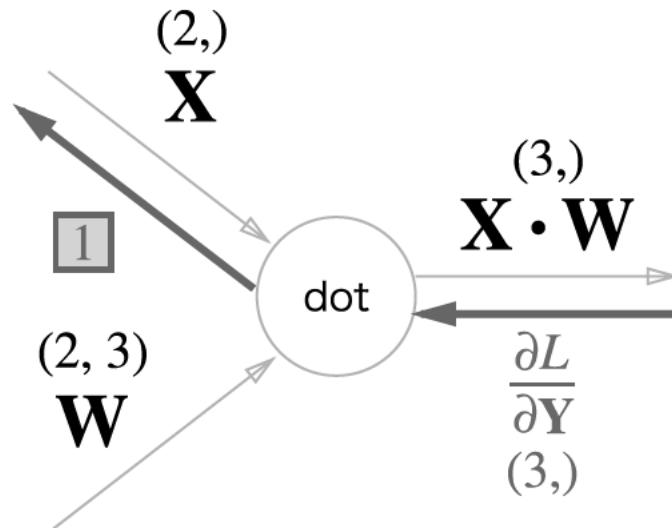
- Affine 계층

- 행렬 곱('dot')의 역전파는 행렬의 대응하는 차원의 원소 수가 일치하도록 곱을 조립하여 구할 수 있음

$$\boxed{1} \quad \frac{\partial L}{\partial \mathbf{Y}} \cdot \mathbf{W}^T = \frac{\partial L}{\partial \mathbf{X}}$$

$(3,)$ $(3, 2)$ $(2,)$





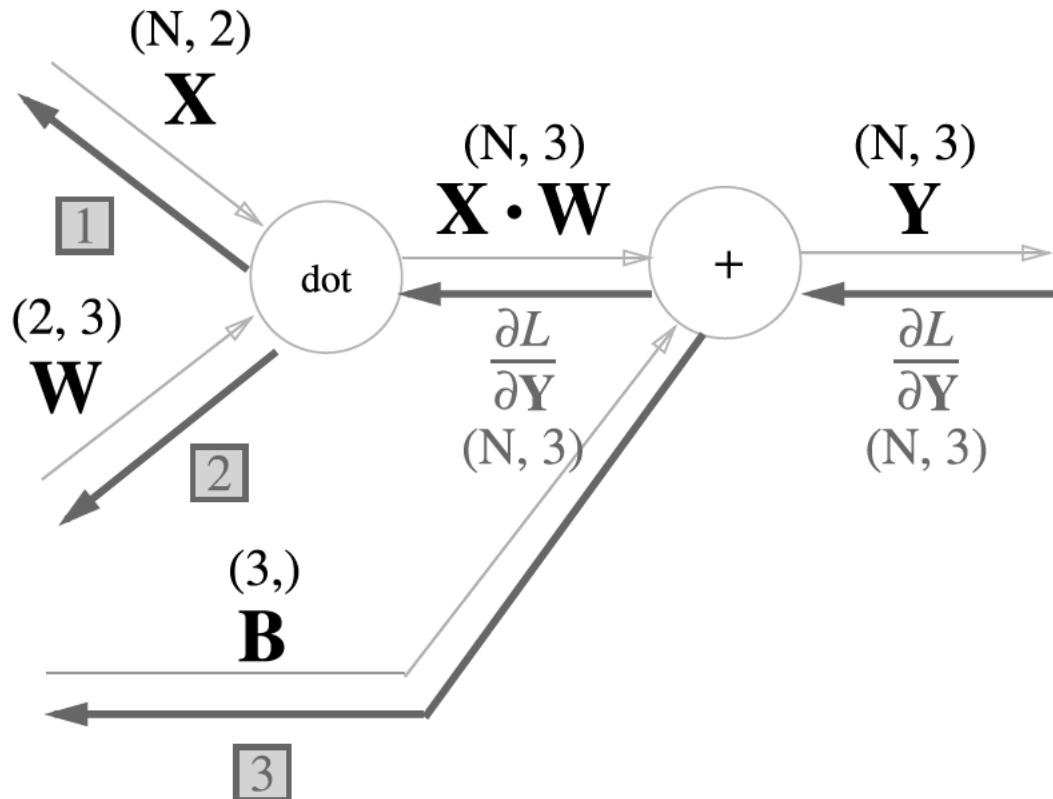
Affine/Softmax 계층 구현하기

- 배치용 Affine 계층
 - 배치용 Affine 계층의 계산 그래프

[1] $\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \cdot \mathbf{W}^T$
 $(N, 2) \quad (N, 3) \quad (3, 2)$

[2] $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^T \cdot \frac{\partial L}{\partial \mathbf{Y}}$
 $(2, 3) \quad (2, N) \quad (N, 3)$

[3] $\frac{\partial L}{\partial \mathbf{B}} = \frac{\partial L}{\partial \mathbf{Y}}$ 의 첫 번째 축(0축, 열방향)의 합
 $(3) \quad (N, 3)$



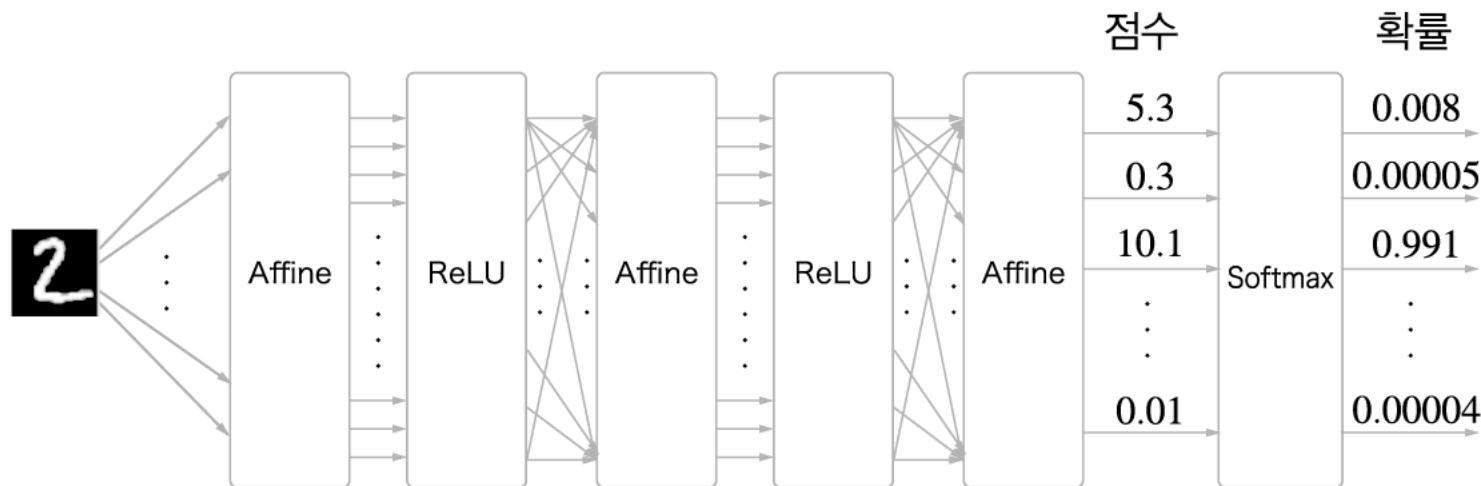
Affine/Softmax 계층 구현하기

- 배치용 Affine 계층

```
class Affine:  
    def __init__(self, W, b):  
        self.W = W  
        self.b = b  
        self.x = None  
        self.dW = None  
        self.db = None  
  
    def forward(self, x):  
        self.x = x  
        out = np.dot(self.x, self.W) + self.b  
  
        return out  
  
    def backward(self, dout):  
        dx = np.dot(dout, self.W.T)  
        self.dW = np.dot(self.x.T, dout)  
        self.db = np.sum(dout, axis=0)  
        return dx
```

Affine/Softmax 계층 구현하기

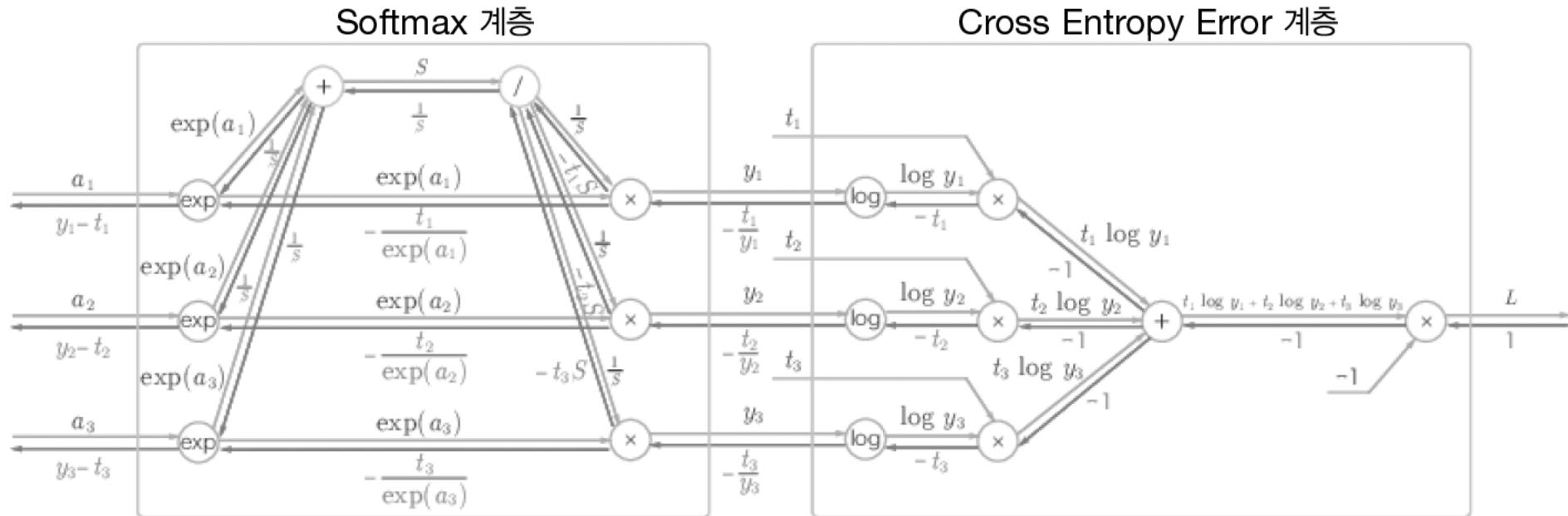
- Softmax-with-Loss 계층
 - 손글씨 숫자 인식에서의 Softmax 계층의 출력



- 입력 이미지가 Affine 계층과 ReLU 계층을 통과하며 변환되고, 마지막 Softmax 계층에 의해서 10개의 입력이 정규화 됨
- 숫자 '0'의 점수는 5.3이며, 이것이 Softmax 계층에 의해서 0.008(0.8%)로 변환
- 숫자 '2'의 점수는 10.1에서 0.991(99.1%)로 변환

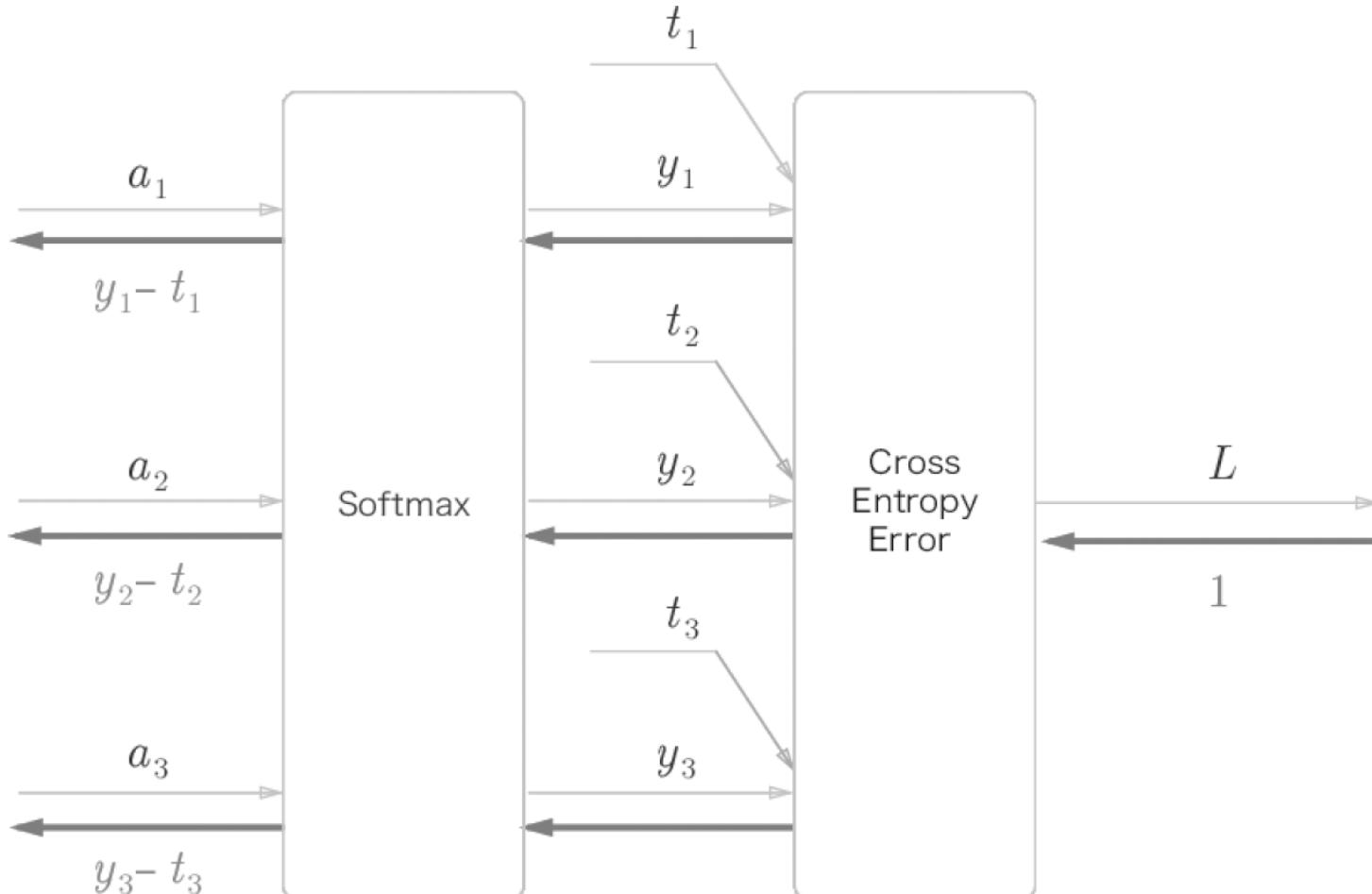
Affine/Softmax 계층 구현하기

- Softmax-with-Loss 계층
 - Softmax-with-Loss 계층의 계산 그래프



Affine/Softmax 계층 구현하기

- Softmax-with-Loss 계층
 - ‘간소화한’ Softmax-with-Loss 계층의 계산 그래프



Affine/Softmax 계층 구현하기

- Softmax-with-Loss 계층

```
class SoftmaxWithLoss:  
    def __init__(self):  
        self.loss = None # 손실함수  
        self.y = None    # softmax의 출력  
        self.t = None    # 정답 레이블(원-핫 인코딩 형태)  
  
    def forward(self, x, t):  
        self.t = t  
        self.y = softmax(x)  
        self.loss = cross_entropy_error(self.y, self.t)  
  
        return self.loss  
  
    def backward(self, dout=1):  
        batch_size = self.t.shape[0]  
        if self.t.size == self.y.size: # 정답 레이블이 원-핫 인코딩 형태일 때  
            dx = (self.y - self.t) / batch_size  
        else:  
            dx = self.y.copy()  
            dx[np.arange(batch_size), self.t] -= 1  
            dx = dx / batch_size  
  
        return dx
```

오차역전파법 구현하기

- 전제
 - 신경망에는 적응 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 '학습'이라 한다.
 - 신경망 학습은 다음과 같이 4단계로 수행한다.
- 1단계 - 미니배치
 - 훈련 데이터 중 일부를 무작위로 가져온다. 이렇게 선별한 데이터를 미니배치라 하며, 그 미니배치의 손실함수 값을 줄이는 것이 목표
- 2단계 - 기울기 산출
 - 미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구한다. 그 기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시
- 3단계 - 매개변수 갱신
 - 가중치 매개변수를 기울기 방향으로 아주 조금 갱신
- 4단계 - 반복
 - 1~3단계를 반복

오차역전파법 구현하기

표 5-1 TwoLayerNet 클래스의 인스턴스 변수

인스턴스 변수	설명
params	딕셔너리 변수로, 신경망의 매개변수를 보관 params['W1']은 1번째 층의 가중치, params['b1']은 1번째 층의 편향 params['W2']는 2번째 층의 가중치, params['b2']는 2번째 층의 편향
layers	순서가 있는 딕셔너리 변수로, 신경망의 계층을 보관 layers['Affine1'], layers['Relu1'], layers['Affine2']와 같이 각 계층을 순서대로 유지
lastLayer	신경망의 마지막 계층 이 예에서는 SoftmaxWithLoss 계층

표 5-2 TwoLayerNet 클래스의 메서드

메서드	설명
__init__(self, input_size, hidden_size, output_size, weight_init_std)	초기화를 수행한다. 인수는 앞에서부터 입력층 뉴런 수, 은닉층 뉴런 수, 출력층 뉴런 수, 가중치 초기화 시 정규분포의 스케일
predict(self, x)	예측(추론)을 수행한다. 인수 x는 이미지 데이터
loss(self, x, t)	손실 함수의 값을 구한다. 인수 x는 이미지 데이터, t는 정답 레이블
accuracy(self, x, t)	정확도를 구한다.
numerical_gradient(self, x, t)	가중치 매개변수의 기울기를 수치 미분 방식으로 구한다(앞 장과 같음).
gradient(self, x, t)	가중치 매개변수의 기울기를 오차역전파법으로 구한다.

오차역전파법으로 구한 기울기 검증하기

- 기울기를 구하는 방법 두가지
 - 수치 미분
 - : 구현하기 쉬우나 느림
 - 해석적으로 수식을 풀어 구현하는 방법(오차역전파법)
 - : 매개변수가 많아도 효율적으로 계산
 - 오차역전파법은 구현하기 복잡해서 오류가 생길 수 있음
 - 그런 이유로 수치 미분의 결과와 오차역전파법의 결과를 비교하여 오차역전파법으로 제대로 구현했는지 검증
 - 이처럼 두 방식으로 구한 기울기가 일치함을 확인하는 작업을 **기울기 확인**(gradient check)이라고 함

오차역전파법을 사용한 학습 구현하기

```
import sys, os
sys.path.append(os.pardir)

import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 데이터 읽기
(x_train, t_train), (x_test, t_test) =
    load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)
```

오차역전파법을 사용한 학습 구현하기

```
for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 기울기 계산
    #grad = network.numerical_gradient(x_batch, t_batch) # 수치 미분 방식
    grad = network.gradient(x_batch, t_batch) # 오차역전파법 방식(훨씬 빠르다)

    # 갱신
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

if i % iter_per_epoch == 0:
    train_acc = network.accuracy(x_train, t_train)
    test_acc = network.accuracy(x_test, t_test)
    train_acc_list.append(train_acc)
    test_acc_list.append(test_acc)
    print(train_acc, test_acc)
```

정리

- 오차역전파법
 - 계산 그래프를 이용하면 계산 과정을 시각적으로 파악할 수 있음
 - 계산 그래프의 노드는 국소적 계산으로 구성됨. 국소적 계산을 조합해 전체 계산을 구성
 - 계산 그래프의 순전파는 통상의 계산을 수행. 한편, 계산 그래프의 역전파로는 각 노드의 미분을 구할 수 있음
 - 신경망의 구성 요소를 계층으로 구현하여 기울기를 효율적으로 계산할 수 있음 (오차역전파법)
 - 수치 미분과 오차역전파법의 결과를 비교하면 오차역전파법의 구현에 잘못이 없는지 확인할 수 있음(기울기 확인)

학습 관련 기술들

매개변수 갱신

- 최적화
 - 신경망 학습의 목적은 손실 함수의 값을 가능한 한 낮추는 매개변수를 찾는 것
 - 이는 매개변수의 최적값을 찾는 문제이며 이러한 문제를 푸는 것을 **최적화**(optimization)라고 함
 - 그러나 신경망 최적화는 굉장히 어려운 문제임
 - 매개변수 공간은 매우 넓고 복잡해서 최적이 솔루션을 쉽게 못 찾음
 - 수식을 풀어 순식간에 최솟값을 구하는 방법은 없으며, 심층 신경망에서는 매개변수의 수가 엄청나게 많아져서 더욱 힘들어짐
 - 앞에서 최적의 매개변수 값을 찾는 단서로 매개변수의 기울기(미분)를 이용했음
 - 매개변수의 기울기를 구해, 기울어진 방향으로 매개변수 값을 갱신하는 일을 몇 번이고 반복해서 점점 최적의 값에 다가갔음
 - 이것이 확률적 경사 하강법(SGD)이란 단순한 방법인데, 매개변수 공간을 무작정 찾는 것보다 '똑똑한' 방법임
 - SGD는 단순하지만 이보다 더 똑똑한 최적화 기법이 있음

매개변수 갱신

- 확률적 경사하강법(SGD)

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

```
class SGD:  
    def __init__(self, lr=0.01):  
        self.lr = lr  
    def update(self, params, grads):  
        for key in params.keys():  
            params[key] -= self.lr * grads[key]
```

```
network = TwoLayerNet(...)  
optimizer = SGD()  
  
for i in range(10000):  
    ...  
    x_batch, t_batch = get_mini_batch(...)  
    grads = network.gradient(x_batch, t_batch)  
    params = network.params  
    optimizer.update(params, grads)  
    ...
```

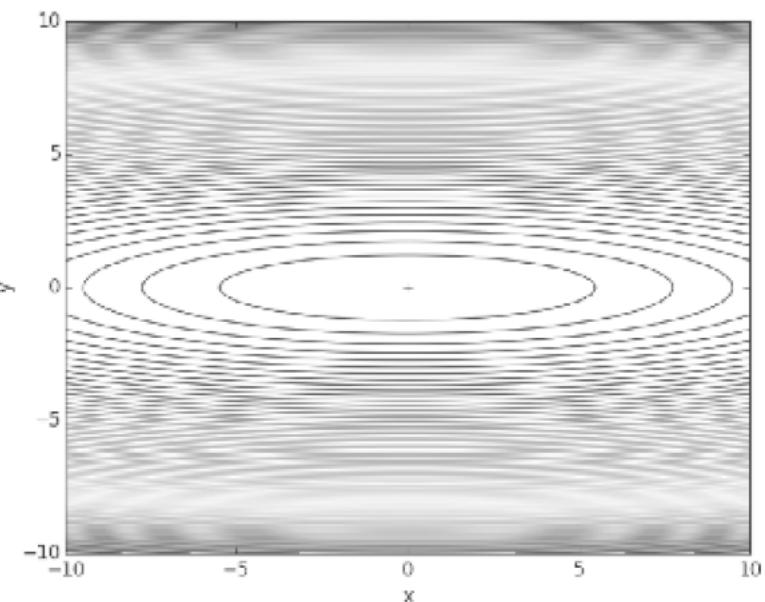
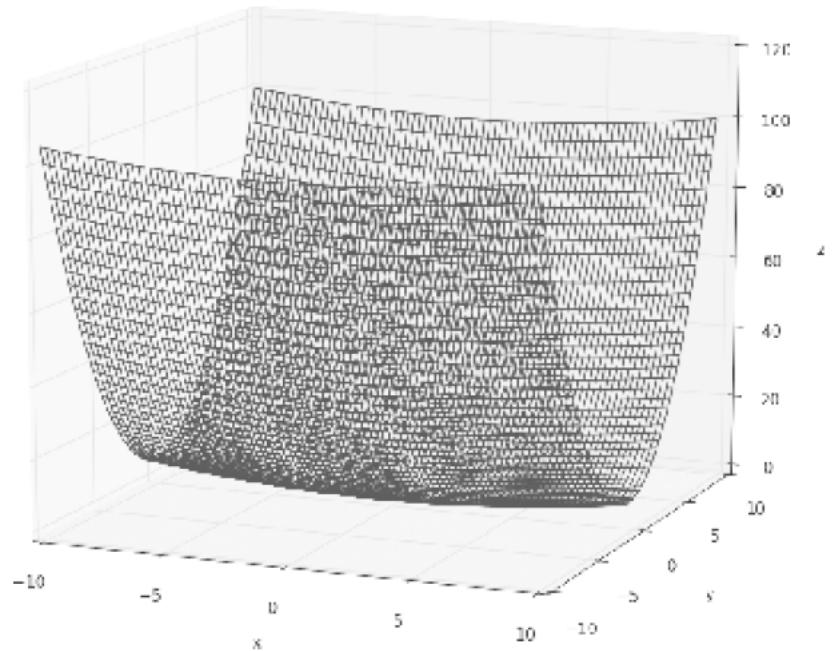
최적화를 담당하는 클래스를 분리해 구현하면 기능을 모듈화하기 좋음

매개변수 갱신

- SGD의 단점
 - SGD는 단순하고 구현도 쉽지만, 문제에 따라서는 비효율적일 때가 있음

$$f(x,y) = \frac{1}{20}x^2 + y^2$$

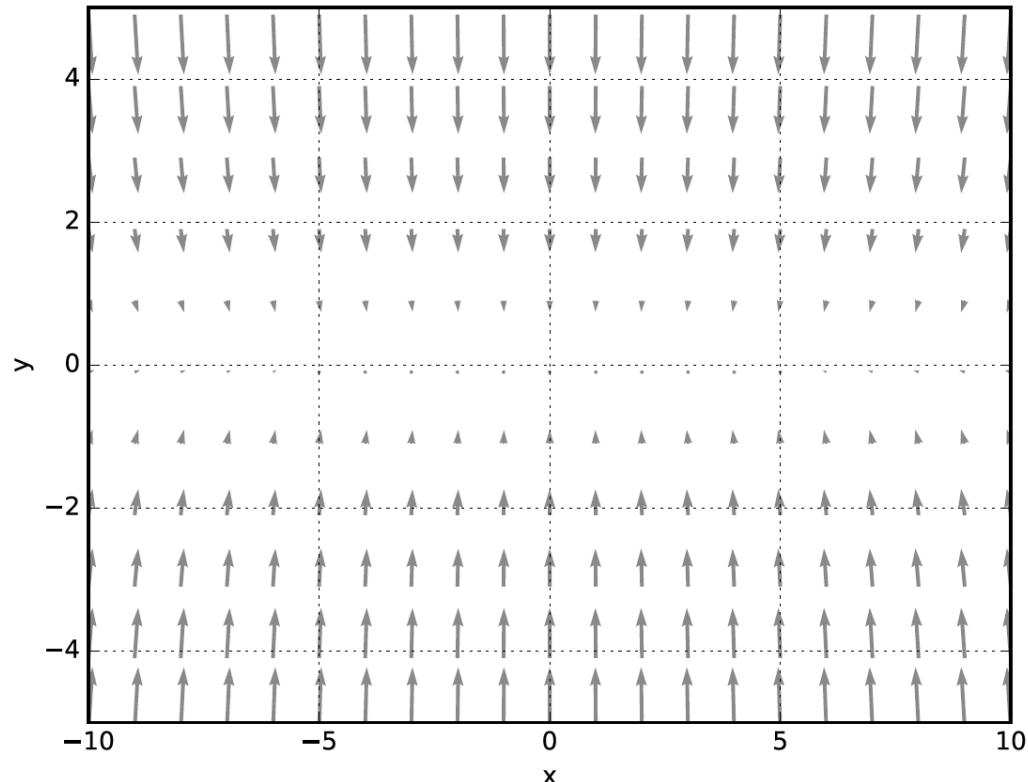
- 이 함수는 아래 그림의 왼쪽과 같이 '밥그릇'을 x 축 방향으로 늘인 듯한 모습이고, 실제로 그 등고선은 오른쪽과 같이 x 축 방향으로 늘인 타원으로 되어 있음



매개변수 갱신

- SGD의 단점

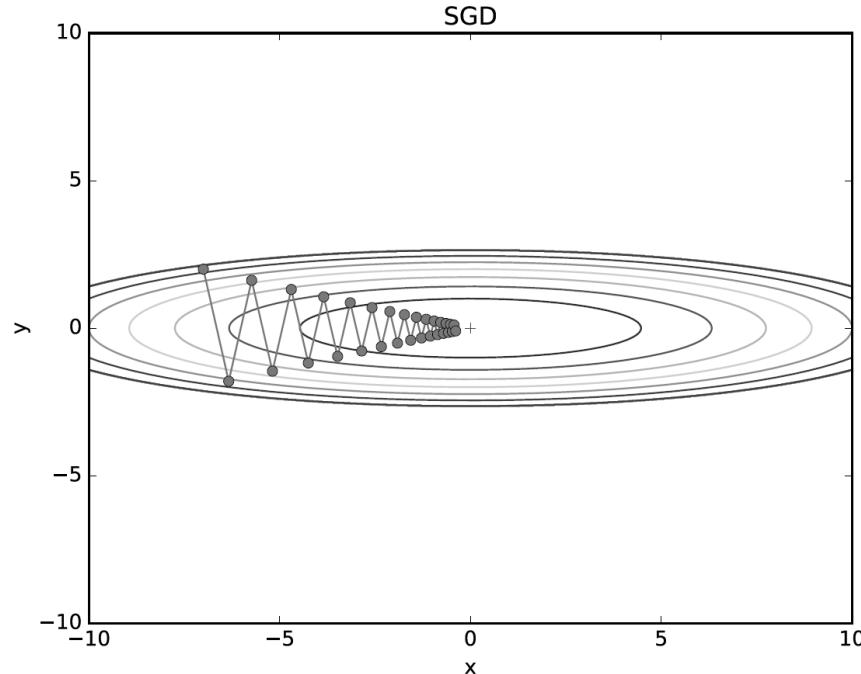
- $f(x,y) = \frac{1}{20}x^2 + y^2$ 식의 기울기



- 그림의 기울기는 y 축 방향은 크고 x 축 방향은 작다는 것이 특징
- y 축 방향은 가파른데 x 축 방향은 완만
- 또한 최솟값이 되는 장소는 $(x, y) = (0, 0)$ 이지만, 그림이 보여주는 기울기 대부분은 $(0, 0)$ 을 가리키지 않음

매개변수 갱신

- SGD의 단점
 - 초깃값 $(x, y) = (-7.0, 2.0)$ 에서 SGD에 의한 최적화 갱신 경로



- SGD가 심하게 굽이진 움직임, 비효율적인 움직임
- SGD의 단점은 비등방성(anisotropy) 함수(방향에 따라 성질, 즉 여기에서는 기울기가 달라지는 함수)에서는 탐색 경로가 비효율적임
- 또한 SGD가 지그재그로 탐색하는 근본 원인은 기울어진 방향이 본래의 최솟값과 다른 방향을 가리켜서임

매개변수 갱신

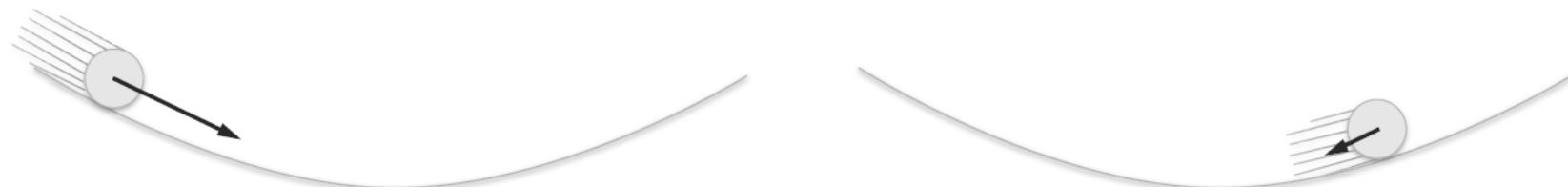
- 모멘텀

- 모멘텀(Momentum)은 '운동량'을 뜻하는 단어로 물리와 관계가 있음
- 모멘텀 기법 수식

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

- \mathbf{W} 는 갱신할 가중치 매개변수, $\frac{\partial L}{\partial \mathbf{W}}$ 는 \mathbf{W} 에 대한 손실 함수의 기울기, η 는 학습률
- \mathbf{v} 는 속도(velocity)
- 기울기 방향으로 힘을 받아 물체가 가속된다는 물리 법칙



- $\alpha \mathbf{v}$ 항은 물체가 아무런 힘을 받지 않을 때 서서히 하강시키는 역할 (물리에서 지면 마찰이나 공기 저항에 해당, α 는 보통 0.9)

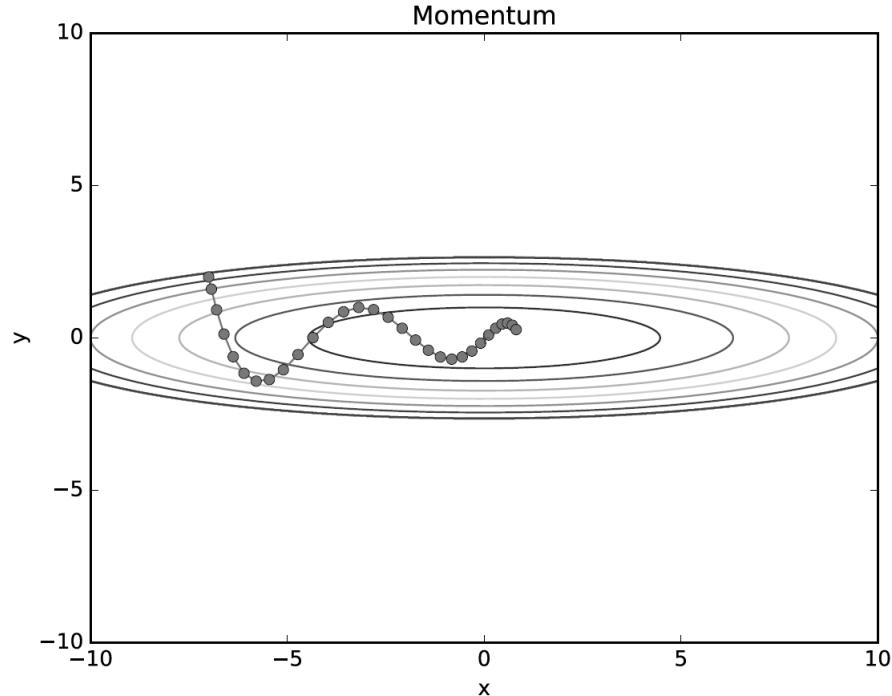
매개변수 갱신

- 모멘텀

```
class Momentum:  
    def __init__(self, lr=0.01, momentum=0.9):  
        self.lr = lr  
        self.momentum = momentum  
        self.v = None  
  
    def update(self, params, grads):  
        if self.v is None:  
            self.v = {}  
        for key, val in params.items():  
            self.v[key] = np.zeros_like(val)  
  
        for key in params.keys():  
            self.v[key] = self.momentum * self.v[key] - self.lr * grads[key]  
            params[key] += self.v[key]
```

매개변수 갱신

- 모멘텀
 - 모멘텀에 의한 최적화 갱신 경로



- 모멘텀 갱신 경로는 공이 그릇 바닥을 구르듯 움직임. SGD 비교하면 '지그재그 정도' 가 덜함. 이는 x축의 힘은 아주 작지만 방향은 변하지 않아서 한 방향으로 일정하게 가속하기 때문임 거꾸로 y축은 힘은 크지만 위아래로 번갈아 받아서 상충하여 y축 방향의 속도는 안정적이지 않음. 전체적으로 SGD보다 x축 방향으로 바르게 다가가 지그재그 움직임이 줄어듬

매개변수 갱신

- AdaGrad
 - 신경망 학습에서는 학습률(수식에서는 η) 값이 중요
 - 이 값이 너무 작으면 학습 시간이 너무 길어지고, 반대로 너무 크면 발산하여 학습이 제대로 이뤄지지 않음
 - 이 학습률을 정하는 효과적 기술로 학습률 감소(learning rate decay)가 있음
 - 이는 학습을 진행하면서 학습률을 점차 줄여가는 방법
 - 처음에는 크게 학습하다가 조금씩 작게 학습
 - AdaGrad는 개별 매개변수에 적응적으로(adaptive) 학습률을 조정하면서 학습을 진행
- $$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

- \mathbf{W} 는 갱신할 가중치 매개변수, $\frac{\partial L}{\partial \mathbf{W}}$ 은 \mathbf{W} 에 대한 손실 함수의 기울기, η 는 학습률
- \mathbf{h} 는 기존 기울기 값을 제곱하여 계속 더해줌. \odot 기호는 행렬의 원소별 곱셈
- 매개변수를 갱신할 때 $\frac{1}{\sqrt{\mathbf{h}}}$ 을 곱해 학습률을 조정.
- 즉 매개변수의 원소 중에서 많이 움직인(크게 갱신된) 원소는 학습률이 낮아진다는 의미(학습률 감소가 매개변수의 원소마다 다르게 적용)

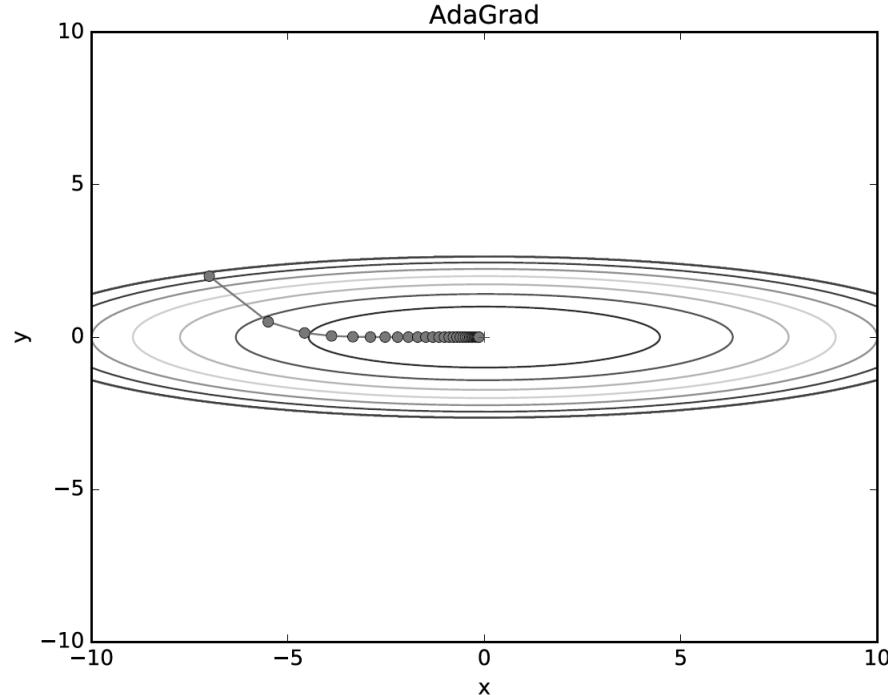
매개변수 갱신

- AdaGrad

```
class AdaGrad:  
    def __init__(self, lr=0.01):  
        self.lr = lr  
        self.h = None  
  
    def update(self, params, grads):  
        if self.h is None:  
            self.h = {}  
            for key, val in params.items():  
                self.h[key] = np.zeros_like(val)  
  
        for key in params.keys():  
            self.h[key] += grads[key] * grads[key]  
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

매개변수 갱신

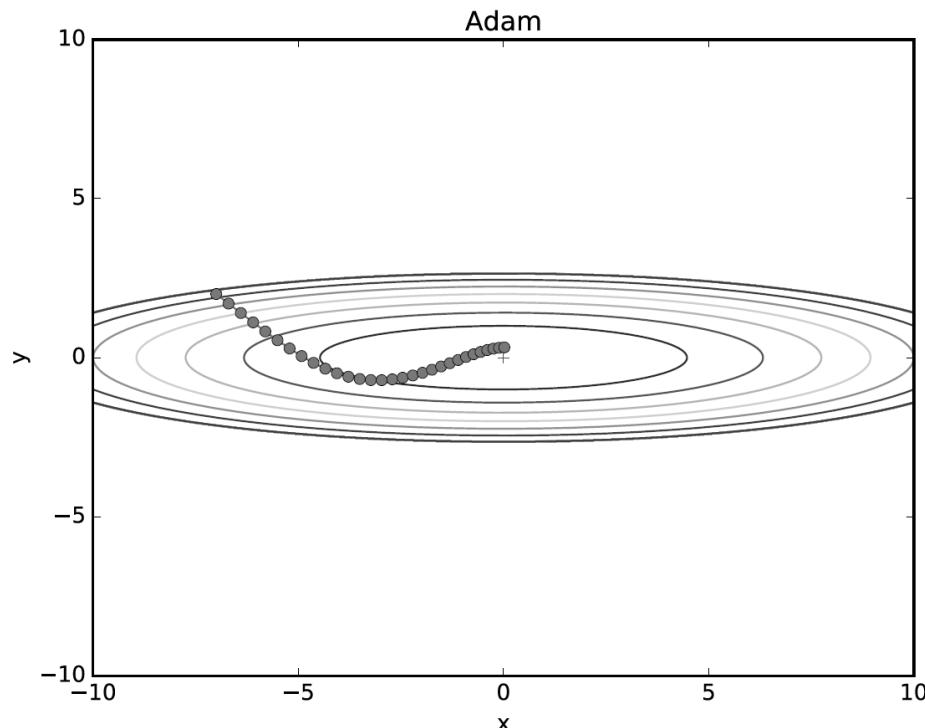
- AdaGrad
 - AdaGrad에 의한 최적화 갱신 경로



- 최솟값을 향해 효율적으로 움직임
- Y축 방향을 기울기가 커서 처음에는 크게 움직이지만, 그 큰 움직임에 비례해 갱신 정도도 큰 폭으로 작아지도록 조정. 그래서 y축 방향으로 갱신 강도가 빠르게 약해지고, 지그재그 움직임이 줄어듬

매개변수 갱신

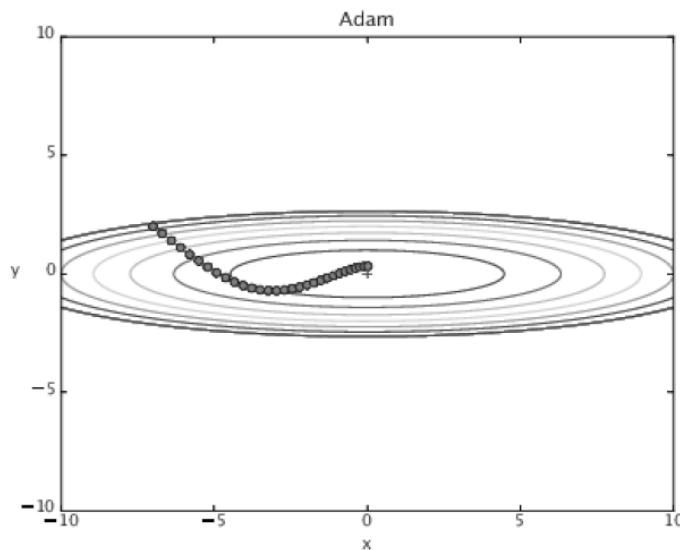
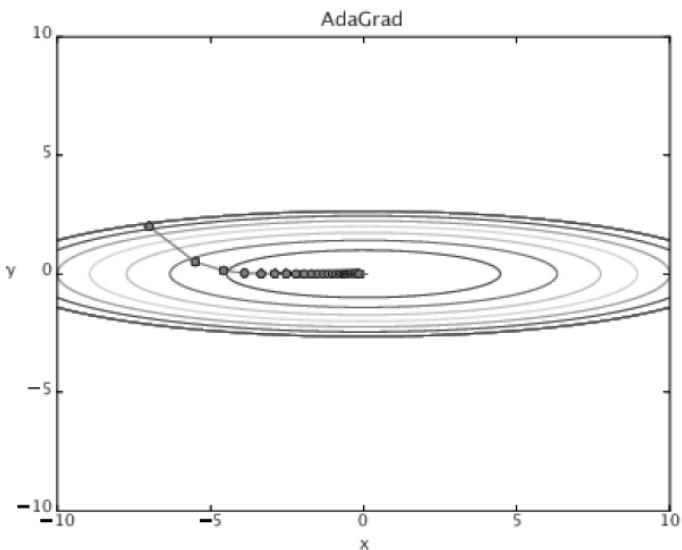
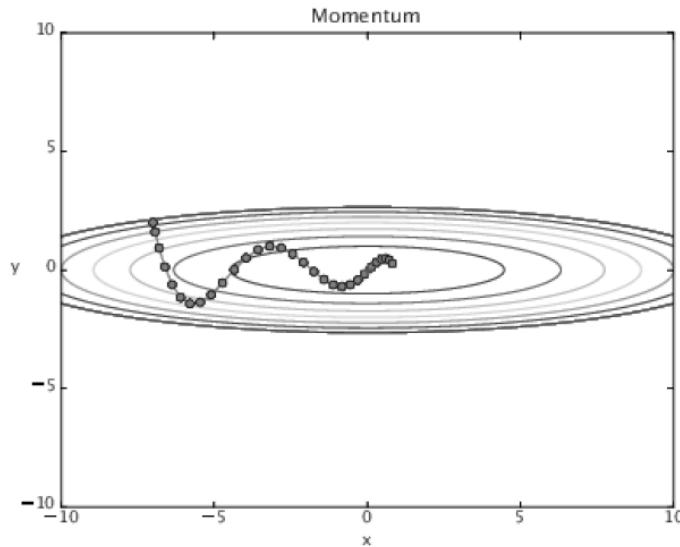
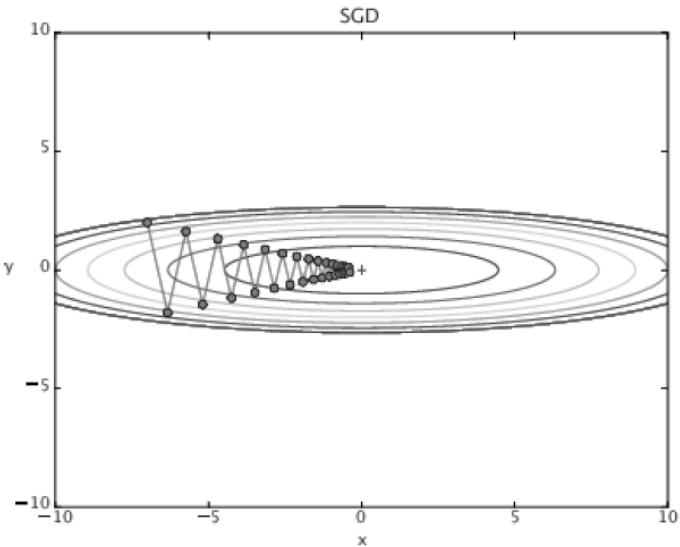
- Adam
 - Adam은 2015년에 제안된 방법
 - 이론은 다소 복잡하지만 직관적으로는 모멘텀과 AdaGrad를 융합한 듯한 방법
 - 하이퍼파라미터의 '편향 보정'이 진행



- 모멘텀과 같이 그릇 바닥을 구르듯 움직이나 공의 좌우 흔들림은 적음. 이는 학습의 갱신 강도를 적응적으로 조정해서 얻는 혜택임

매개변수 갱신

- 어느 갱신 방법을 이용할 것인가?

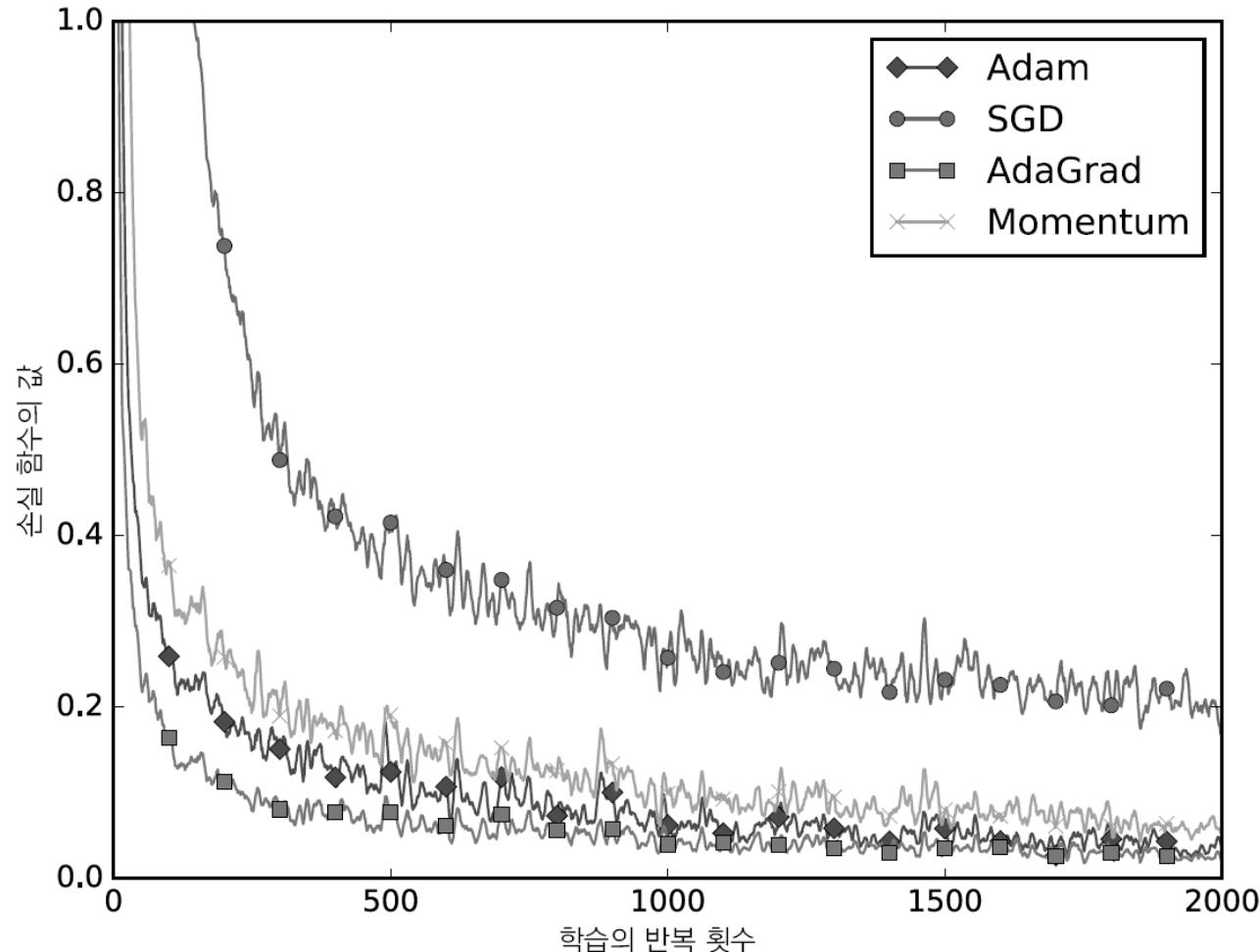


매개변수 갱신

- 어느 갱신 방법을 이용할 것인가?
 - 그림에서는 AdaGrad가 가장 나은 것 같은데, 사실 그 결과는 풀어야 할 문제가 무엇이냐에 따라 달라질 수 있음
 - 또한 학습률 등의 하이퍼파라미터를 어떻게 설정하느냐에 따라서도 결과가 바뀜
 - SGD, 모멘텀, AdaGrad, Adam의 네 후보 중 어느 것이 더 뛰어난 기법이라는 것은 없음. 각자의 장단이 있어 잘 푸는 문제와 서툰문제가 있음

매개변수 갱신

- MNIST 데이터셋으로 본 갱신 방법 비교
 - MNIST 데이터셋에 대한 학습 진도 비교



매개변수 갱신

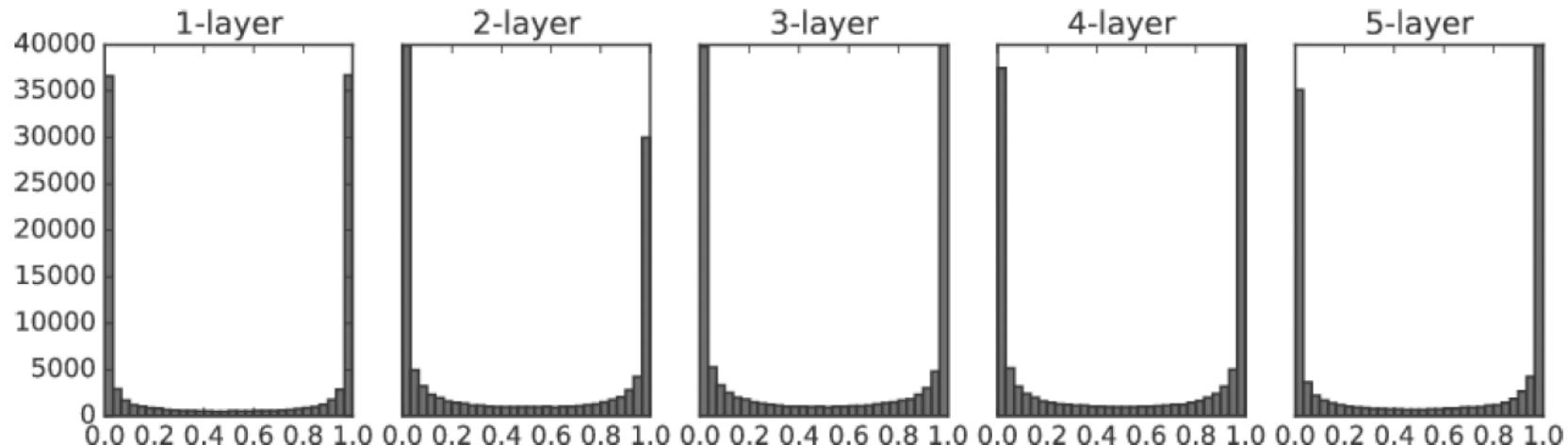
- MNIST 데이터셋으로 본 갱신 방법 비교
 - 이 실험은 각 층이 100개의 뉴런으로 구성된 5층 신경망에서 ReLU 활성화 함수를 사용해 측정
 - SGD의 학습 진도가 가장 느리고 나머지 세 기법의 진도는 비슷. 그 중 AdaGrad 가 조금 빠름
 - 여기서 주의할 점은 하이퍼파라미터인 학습률과 신경망의 구조(층 깊이 등)에 따라 결과가 달라진다는 것
 - 다만 일반적으로 SGD보다 다른 세 기법이 빠르게 학습하고, 때로는 최종 정확도도 높에 나타남

가중치의 초기값

- 초기값을 0으로 하면?
 - 가중치 감소
가중치의 매개변수의 값이 작아지도록 학습하는 방법으로 오버피팅을 억제해 범용 성능을 높이는 테크닉
 - 앞에서 사용한 가중치의 초기값
 $0.01 * np.random.randn(input_size, hidden_size)$ 처럼 정규분포에서 생성되는 값을 0.01배 한 작은 값(표준편차가 0.01인 정규분포)를 사용
 - 가중치의 초기값을 모두 0으로 설정하면?
학습이 올바로 이뤄지지 않음
 - 초기값을 모두 0으로 해서는 안되는 이유(가중치를 균일한 값으로 설정하면 안되는 이유)
오차역전파법에서 모든 가중치의 값이 똑같이 갱신됨
 - 예를 들어 2층 신경망에서 첫 번째와 두 번째 층의 가중치가 0이라고 가정하면 순전파 때는 입력층의 가중치가 0이기 때문에 두 번째 층의 뉴런에 모두 같은 값이 전달됨
 - 두 번째 층의 모든 뉴런에 같은 값이 입력된다는 것은 역전파 때 두 번째 층의 가중치가 모두 똑같이 갱신된다는 의미(예: 곱셈노드의 역전파)
 - 그래서 가중치들은 같은 초기값에서 시작하고 갱신을 거쳐도 여전히 같은 값을 유지하게 됨. 이는 가중치를 여러 개 갖는 의미를 사라지게 함
 - 이런 상황을 막기 위해서는 초기값을 무작위로 설정해야 함

가중치의 초기값

- 은닉층의 활성화값 분포
 - 은닉층의 활성화값(활성화 함수의 출력 데이터)의 분포를 관찰하면 중요한 정보를 얻을 수 있음
 - 활성화 함수로 시그모이드 함수를 사용하는 5층 신경망에 무작위로 생성한 입력 데이터를 흘리며 각 층의 활성화값 분포를 하스토그램으로 그려보기

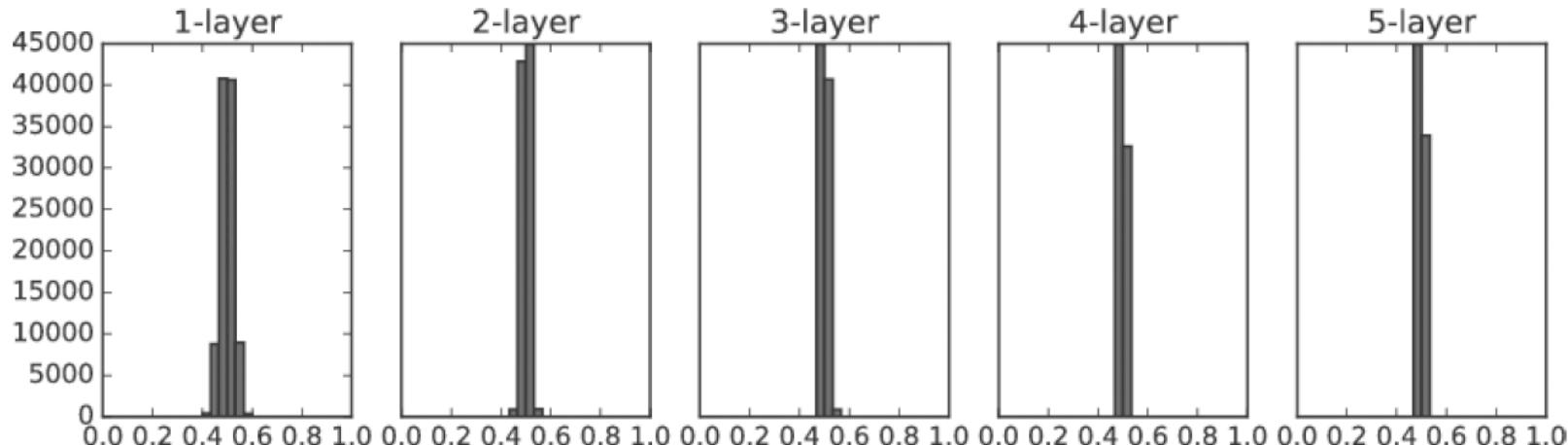


- 각 층의 활성화값들이 0과 1에 치우쳐 분포
- 시그모이드 함수의 출력이 0 또는 1에 가까워지면 그 미분은 0에 다가감
- 그래서 데이터가 0과 1에 치우쳐 분포하게 되면 역전파의 기울기 값이 점점 작아지다가 사라짐. 이것이 **기울기 소실**(gradient vanishing)이라 알려진 문제임
- 층을 깊게 하는 딥러닝에서는 기울기 소실은 더 심각한 문제가 될 수 있음

가중치의 초기값

- 은닉층의 활성화값 분포

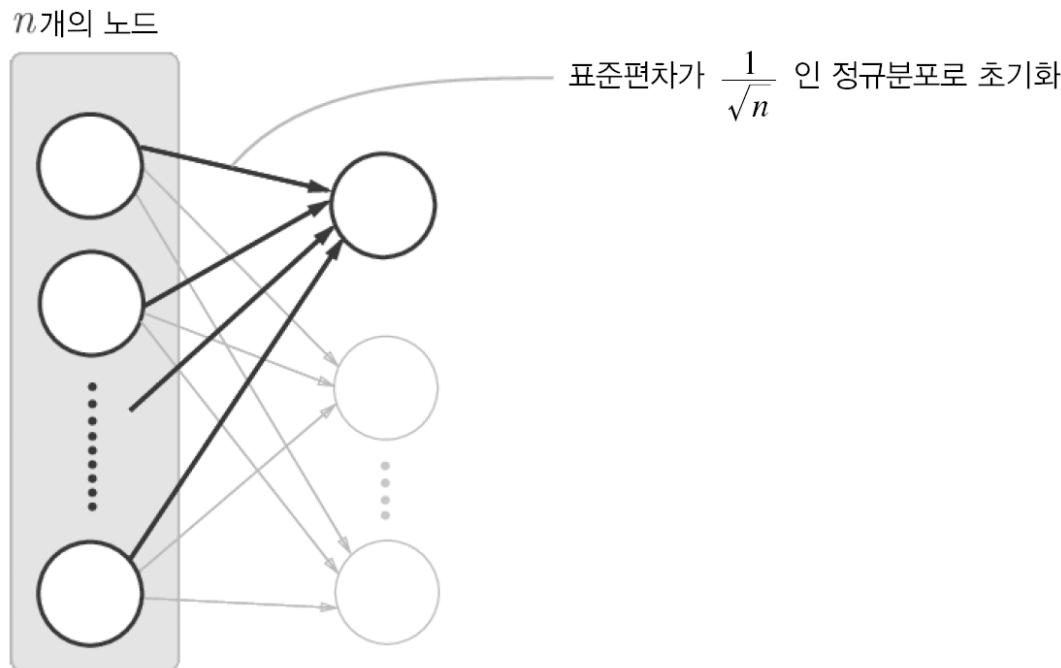
- 가중치의 표준편차를 0.01로 바꿔 같은 실험 반복



- 이번에는 0.5 부근에 집중
 - 앞의 예처럼 0과 1로 치우치진 않았으니 기울기 소실 문제는 일어나지 않지만, 활성화 값들이 치우쳤다는 것은 표현력 관점에서는 큰 문제가 될 수 있음
 - 즉, 다수의 뉴런이 거의 같은 값을 출력하고 있으니 뉴런을 여러 개 둔 의미가 없어진다는 의미
 - 예를 들어 뉴런 100개가 거의 같은 값을 출력한다면 뉴런 1개짜리와 다른게 없음
 - 그래서 활성화 값들이 치우치면 **표현력을 제한한다는** 관점에서 문제가 됨

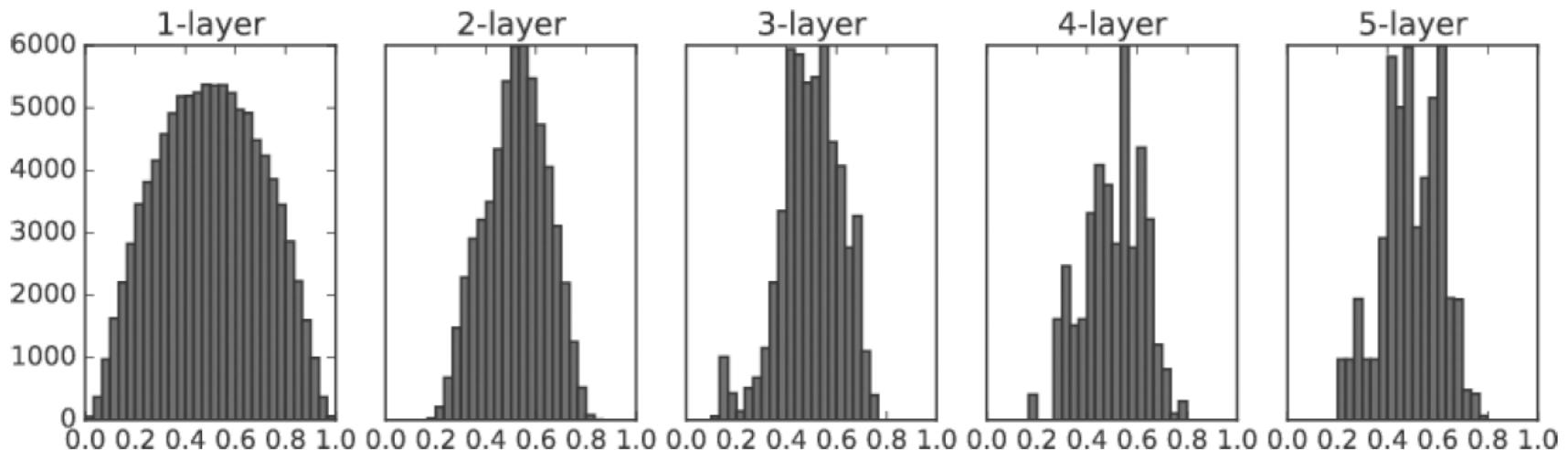
가중치의 초기값

- 은닉층의 활성화값 분포
 - 사비에르 글로로트와 요슈아 벤지오의 논문에서 권하는 **Xavier 초기값**
 - Xavier 초기값은 일반적인 딥러닝 프레임워크들의 표준적으로 이용
 - 이 논문은 각 층의 활성화값들을 광범위하게 분포시킬 목적으로 가중치의 적절한 분포를 찾고자 했음
 - 그리고 앞 계층의 노드가 n 개라면 표준편차가 $\frac{1}{\sqrt{n}}$ 인 분포를 사용하면 된다는 결론을 이끔



가중치의 초기값

- 은닉층의 활성화값 분포
 - Xavier 초기값을 사용하면 앞 층의 노드가 많을수록 대상 노드의 초기값으로 설정하는 가중치가 좁게 퍼짐
 - 가중치의 초기값으로 'Xavier 초기값'을 이용할 때의 각 층의 활성화값 분포



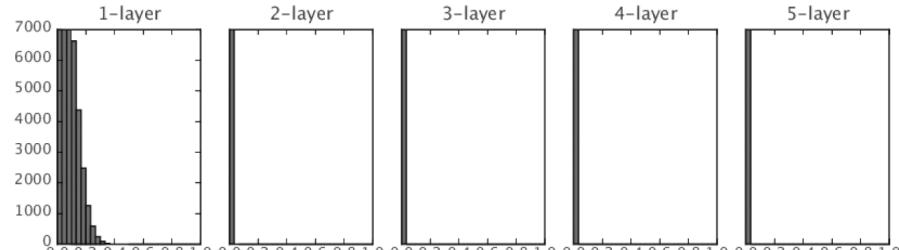
- Xavier 초기값을 사용한 결과를 보면 층이 깊어지면서 형태가 다소 일그러지지만, 앞에서 본 방식보다는 확실히 넓게 분포됨을 알 수 있음
- 각 층에 흐르는 데이터가 적당히 퍼져 있으므로, 시그모이드 함수의 표현력도 제한 받지 않고 학습이 효율적으로 이뤄질 것임

가중치의 초기값

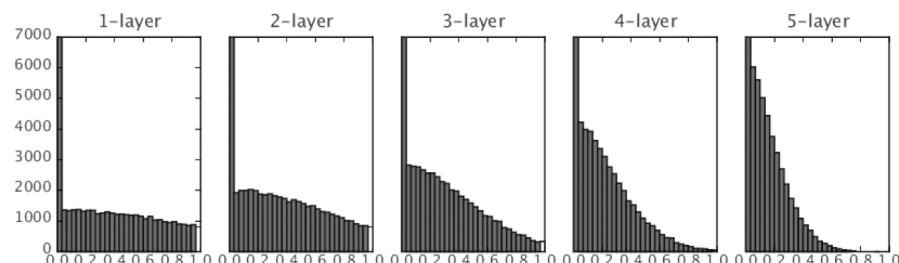
- ReLU를 사용할 때의 가중치 초기값
 - Xavier 초기값은 활성화 함수가 선형인 것을 전제로 이끈 결과
 - Sigmoid 함수와 tanh 함수는 좌우 대칭이라 중앙 부근이 선형인 함수로 볼 수 있어서 Xavier 초기값이 적당
 - 반면 ReLU를 이용할 때는 ReLU에 특화된 초기값을 이용하라고 권장
 - 이 특화된 초기값을 찾아낸 카이밍 히의 이름을 따 **He 초기값** 이라 함
 - He 초기값은 앞 계층의 노드가 n개일 때, 표준 편차가 $\frac{2}{\sqrt{n}}$ 인 정규분포를 사용
 - ReLU는 음의 영역이 0이라서 더 넓게 분포시키기 위해 2배의 계수가 필요하다고 해석할 수 있음

가중치의 초기값

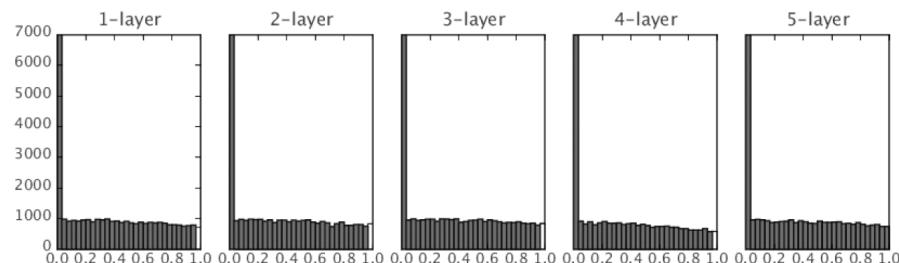
- ReLU를 사용할 때의 가중치 초기값
 - 활성화 함수로 ReLU를 사용한 경우의 가중치 초기값에 따른 활성화 분포 변화



표준편차가 0.01인 정규분포를 가중치 초기값으로 사용한 경우



Xavier 초기값을 사용한 경우



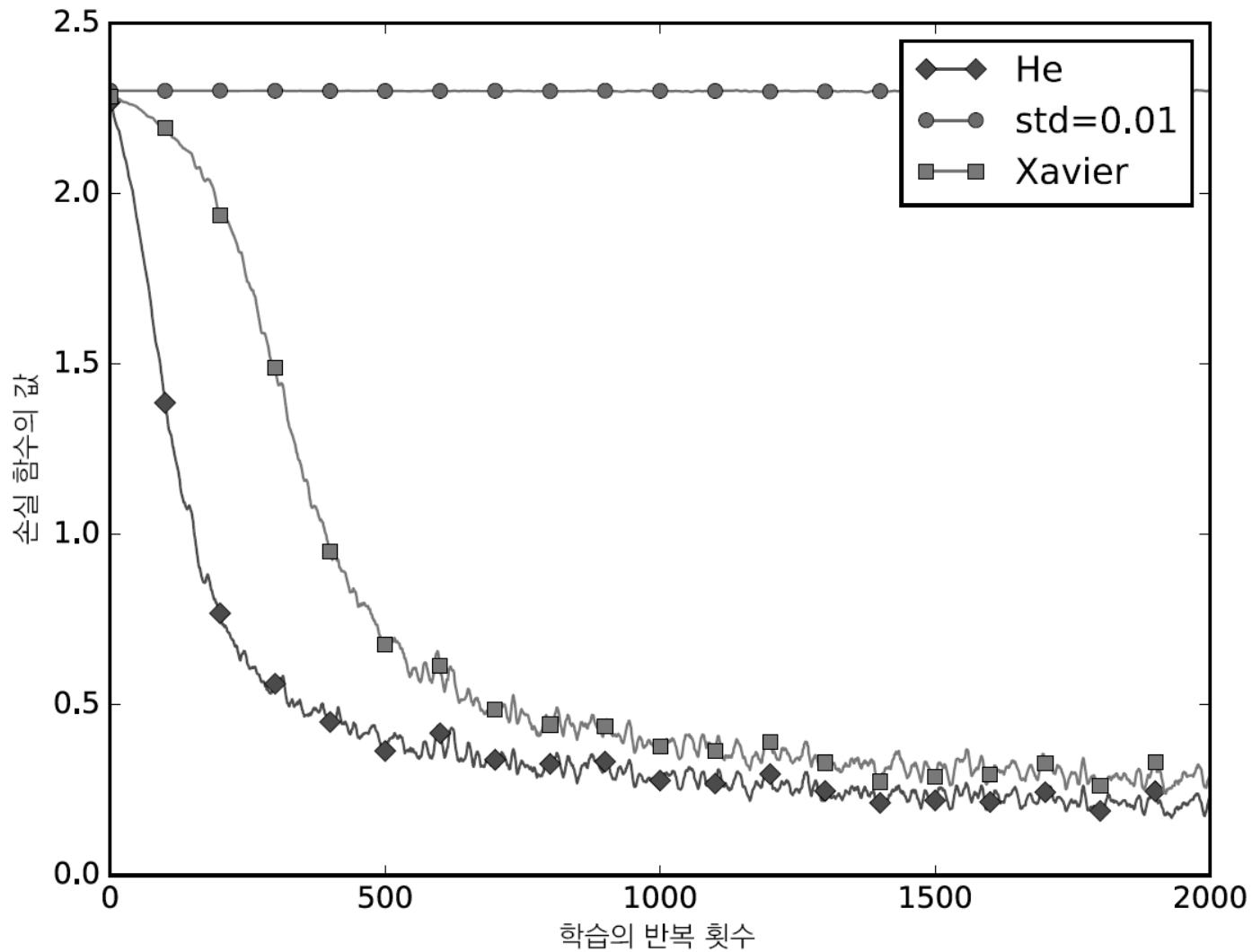
He 초기값을 사용한 경우

가중치의 초기값

- ReLU를 사용할 때의 가중치 초기값
 - std=0.01 일 때의 각 층의 활성화 값들은 아주 작은 값들임
 - 신경망에 아주 작은 데이터가 흐른다는 것은 역전파 때 가중치의 기울기 역시 작아진다는 의미
 - 이는 중대한 문제이며, 실제로도 학습이 거의 이뤄지지 않음
 - Xavier 초기값 결과를 보면 이쪽은 층이 깊어지면서 치우침이 조금씩 커짐
 - 실제로 층이 깊어지면 활성화값들의 치우침도 커지고, 학습할 때 '기울기 소실' 문제를 일으킴
 - He 초기값은 모든 층에서 균일하게 분포
 - 층이 깊어져도 분포가 균일하게 유지되기에 역전파 때도 적절한 값이 나올 것으로 기대
 - (모범 사례) 활성화 함수로 ReLU를 사용할 때는 He 초기값을, sigmoid나 tanh 등의 S자 모양 곡선일 때는 Xavier 초기값을 사용

가중치의 초기값

- MNIST 데이터셋으로 본 가중치 초기값 비교



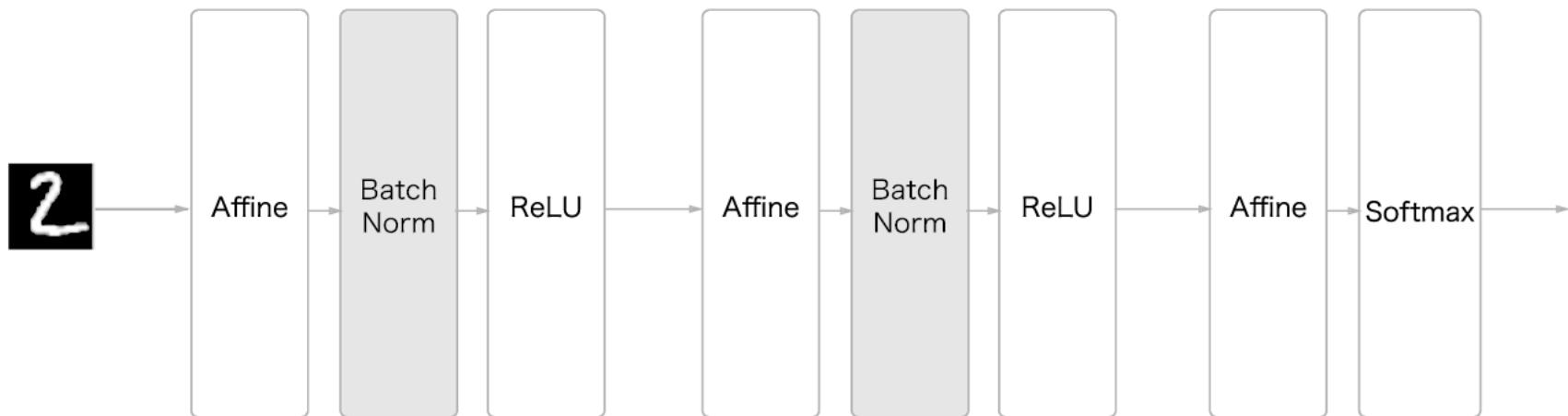
가중치의 초기값

- MNIST 데이터셋으로 본 가중치 초기값 비교
 - 층별 뉴런 수가 100개인 5층 신경망에서 활성화 함수로 ReLU를 사용
 - $\text{std} = 0.01$ 일 때는 학습이 전혀 이뤄지지 않음
 - 순전파 때 너무 작은 값, 즉 0 근처로 밀집한 데이터가 흐르기 때문
 - 그로 인해 역전파 때의 기울기도 작아져 가중치가 거의 갱신되지 않음
 - 반대로 Xavier와 He 초기값의 경우는 학습이 순조롭게 이뤄지고 있음
 - 학습 진도는 He 초기값 쪽이 더 빠름
 - (정리) 가중치의 초기값은 신경망 학습에 아주 중요한 포인트. 가중치의 초기값에 따라 신경망 학습의 성패가 갈림

배치 정규화

- 배치 정규화 알고리즘

- 배치 정규화(Batch Normalization)는 가중치 초기값을 적절히 설정함으로 층의 활성화 분포가 적당히 퍼지면서 학습이 잘 되었던 것 처럼, 각 층의 활성화를 적당히 퍼뜨리도록 강제하자는 아이디어에서 출발
- 배치 정규화가 주목 받는 이유
 1. 학습을 빨리 진행할 수 있음
 2. 초기값에 크게 의존하지 않음
 3. 오버피팅을 억제
- 배치 정규화를 사용한 신경망의 예



배치 정규화

- 배치 정규화 알고리즘

- 배치 정규화는 그 이름과 같이 학습 시 미니배치 단위로 정규화
 - 구체적으로는 데이터 분포가 평균이 0, 분산이 1이 되도록 정규화

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

- 여기에서 미니배치 $B = \{x_1, x_2, x_3, \dots, x_m\}$ 이라는 m 개의 입력 데이터의 집합에 대해 평균 μ_B 와 분산 σ_B^2 을 구함
 - 그리고 평균이 0, 분산이 1이 되게(적절한 분포가 되게) 정규화 함
 - ϵ 기호(eplison, 엡실론)는 작은 값(예컨대 $10e-7$)으로 0으로 나누는 사태를 예방
 - 미니배치 입력 데이터를 평균 0, 분산 1인 데이터를 변환하는 이 처리를 활성화 함수의 앞(혹은 뒤)에 삽입함으로써 데이터 분포가 덜 치우치게 할 수 있음

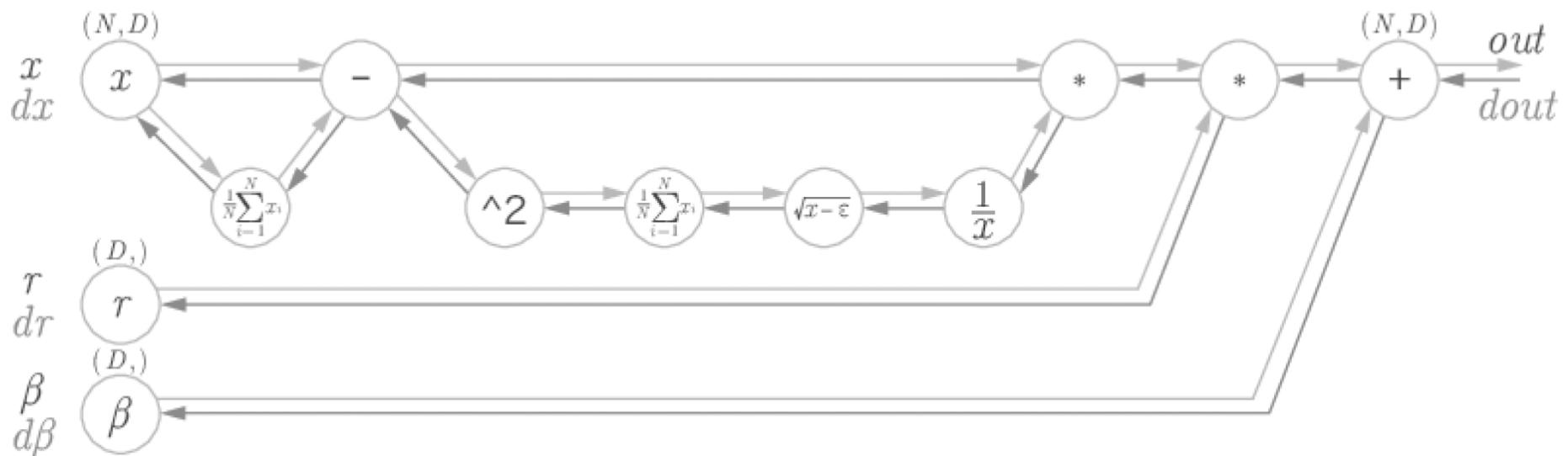
배치 정규화

- 배치 정규화 알고리즘

- 배치 정규화 계층마다 이 정규화된 데이터에 고유한 확대(scale)와 이동(shift) 변환을 수행

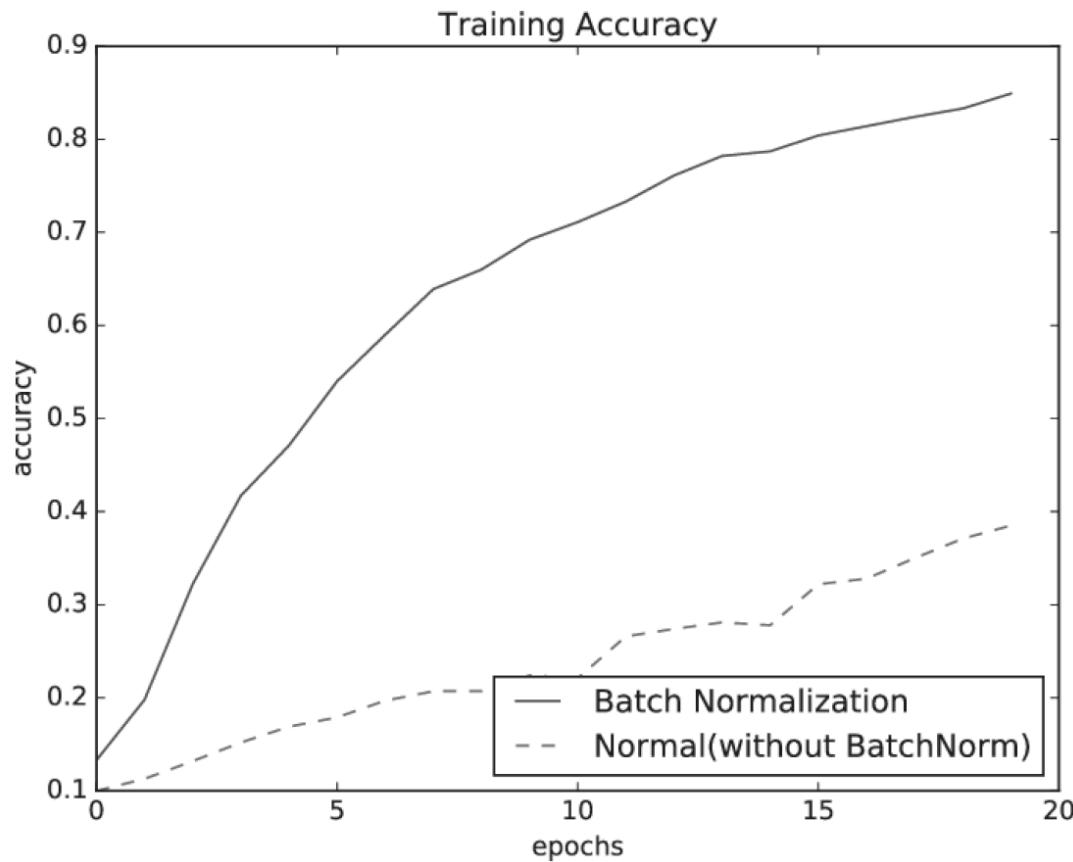
$$y_i \leftarrow \gamma \hat{x}_i + \beta$$

- 이 식에서 γ 가 확대를, β 가 이동을 담당
 - 두 값은 처음에는 $\gamma = 1$, $\beta = 0$ 부터 시작하고 학습하면서 적합한 값으로 조정
 - 배치 정규화의 계산 그래프



배치 정규화

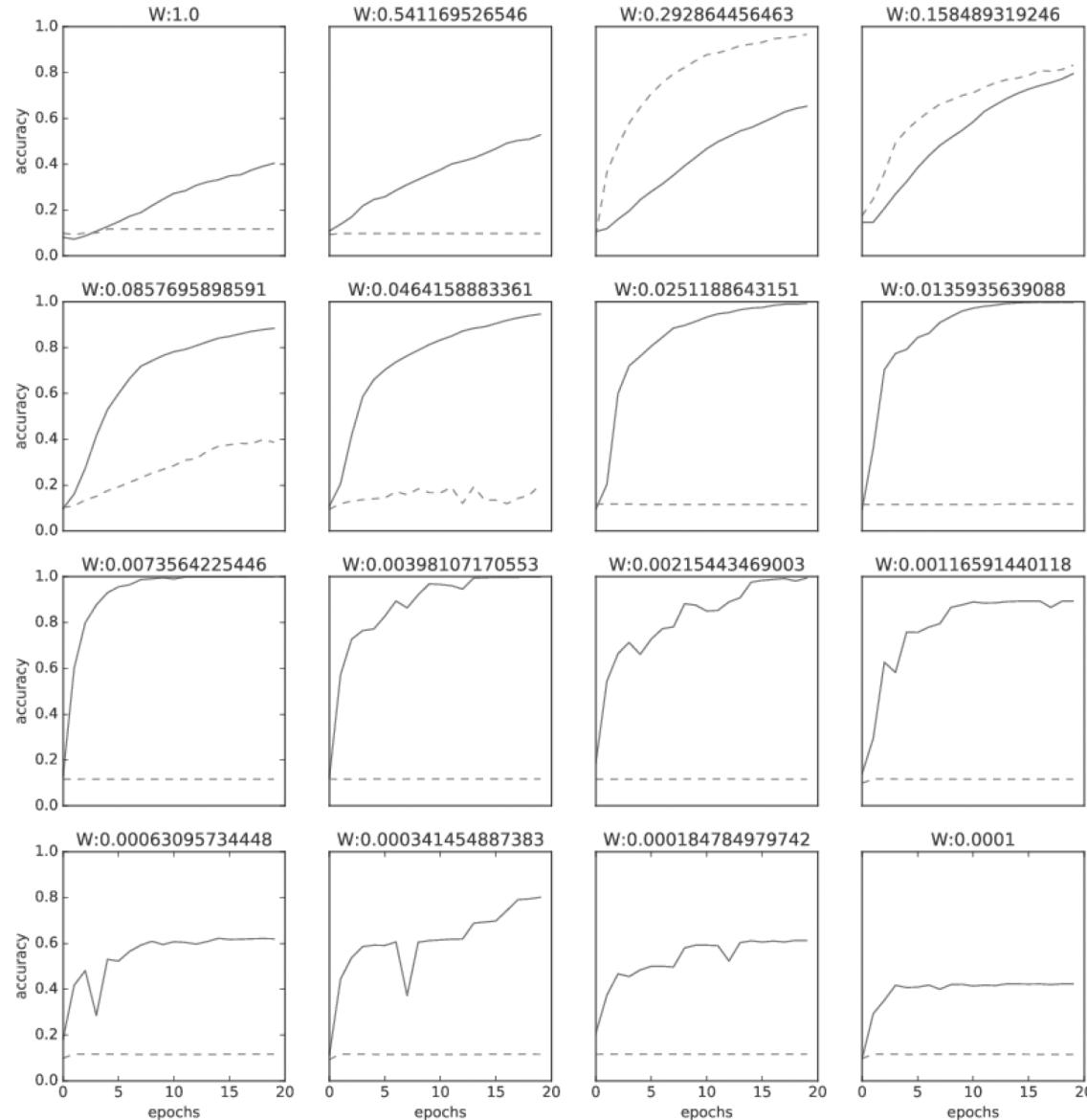
- 배치 정규화의 효과



- 배치 정규화가 학습을 빨리 진전시키고 있음

배치 정규화

- 배치 정규화의 효과



배치 정규화

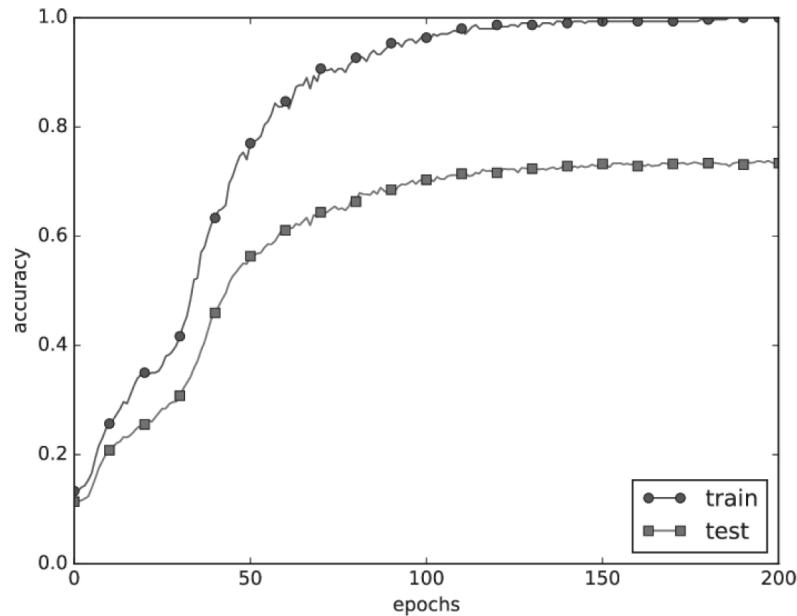
- 배치 정규화의 효과
 - 가중치 초기값의 표준편차를 다양하게 바꿔가며 학습 경과 관찰
 - 거의 모든 경우에서 배치 정규화를 사용할 때(실선)의 학습 진도가 빠름
 - 실제로 배치 정규화를 이용하지 않는 경우엔 초기값이 잘 분포되어 있지 않으면 학습이 전혀 진행되지 않는 결과
 - 이렇게 배치 정규화를 사용하면 학습이 빨라지고, 가중치 초기값에 크게 의존하지 않아도 됨

바른 학습을 위해

- 오버피팅
 - 기계학습에서는 오버피팅이 문제가 되는 일이 많음
 - 오버피팅이란 신경망이 훈련 데이터에만 지나치게 적응되어 그 외의 데이터에는 제대로 대응하지 못하는 상태
 - 기계학습은 범용 성능을 지향하므로 훈련 데이터에 포함되지 않는, 아직보지 못한 데이터가 주어져도 바르게 식별해내는 모델이 바람직
 - 복잡하고 표현력이 높은 모델을 만들 수는 있지만, 그만큼 오버피팅을 억제하는 기술이 중요
 - 오버피팅은 다음 두 경우에 일어남
 1. 매개변수가 많고 표현력이 높은 모델
 2. 훈련 데이터가 적음

바른 학습을 위해

- 오버피팅
 - 오버피팅을 재현하기 위해 MNIST 데이터셋의 훈련 데이터 중 300개만 사용하고, 7층 네트워크를 사용해 네트워크의 복잡성을 높임. 각 층의 뉴런은 100개, 활성화 함수는 ReLU 사용



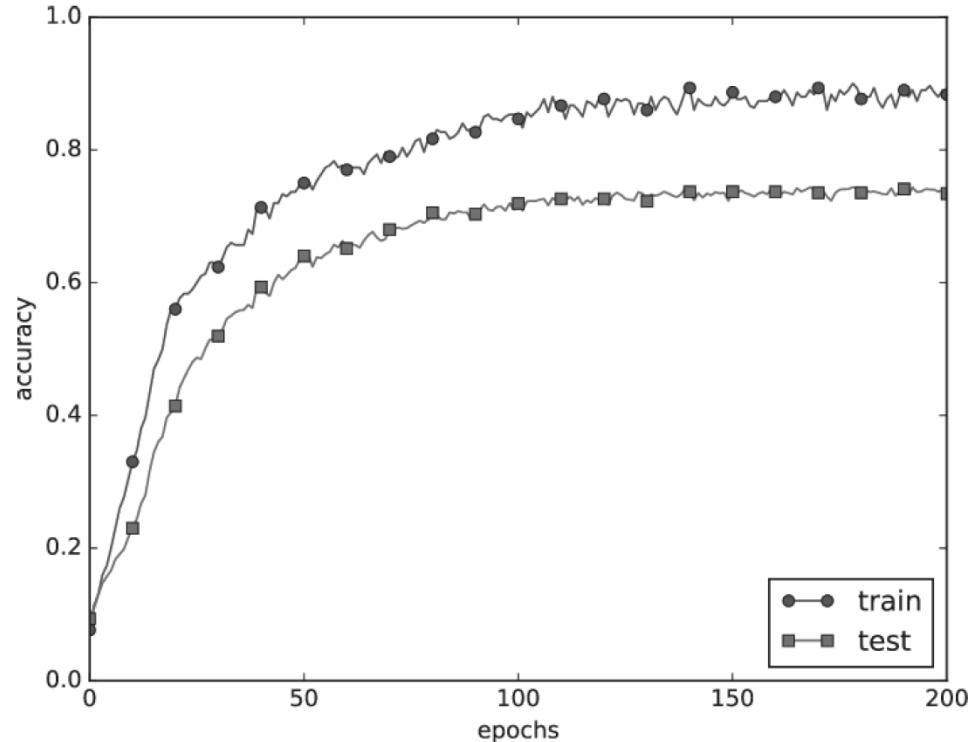
- 훈련 데이터를 사용하여 측정한 정확도는 100 에폭을 지나는 무렵부터 거의 100%이나 시험 데이터에 대해서는 큰 차이
- 이처럼 정확도가 크게 벌어지는 것은 훈련 데이터에만 적응(fitting)해버린 결과
- 범용 데이터(시험 데이터)에는 제대로 대응하지 못함을 확인할 수 있음

바른 학습을 위해

- 가중치 감소
 - 오버피팅 억제용으로 많이 이용해온 방법 중 **가중치 감소**(weight decay)가 있음
 - 이는 학습 과정에서 큰 가중치에 대해서는 그에 상응하는 큰 패널티를 부과하여 오버피팅을 억제하는 방법임
 - 원래 오버피팅은 가중치 매개변수의 값이 커서 발생하는 경우가 많기 때문임
 - 가중치의 제곱 노름(L2 노름)을 손실함수에 더함으로 가중치가 커지는 것을 억제 할 수 있음
 - 가중치를 W 라고 하면 L2 노름에 따른 가중치 감소는 $\frac{1}{2}\lambda W^2$ 이 되고, 이 $\frac{1}{2}\lambda W^2$ 을 손실함수에 더함
 - 여기에서 λ (람다)는 정규화의 세기를 조절하는 하이퍼파라미터
 - λ 를 크게 설정할수록 큰 가중치에 대한 패널티가 커짐
 - 또 $\frac{1}{2}\lambda W^2$ 의 앞쪽 $\frac{1}{2}$ 은 $\frac{1}{2}\lambda W^2$ 의 미분 결과인 λW 를 조정하는 역할의 상수
 - 가중치 감소는 모든 가중치 각각의 손실 함수에 $\frac{1}{2}\lambda W^2$ 을 더함
 - 따라서 가중치의 기울기를 구하는 계산에서는 그동안의 오차역전파법에 따를 결과에 정규화 항을 미분한 λW 를 더함

바른 학습을 위해

- 가중치 감소
 - 가중치 감소를 이용한 훈련 데이터와 시험 데이터에 대한 정확도 추이

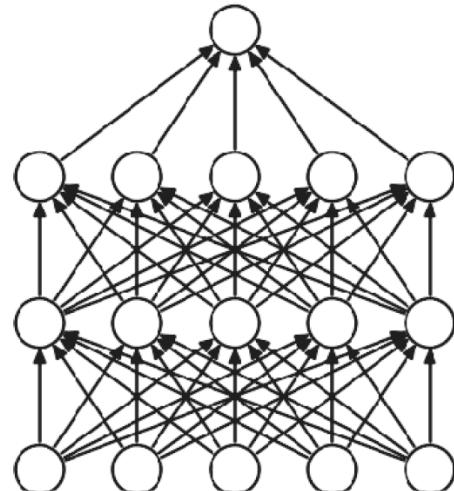


- 훈련 데이터와 시험 데이터의 정확도에는 여전히 차이가 있지만, 가중치 감소를 이용하지 않은 이전의 그림과 비교하면 그 차이가 줄었음. 즉 오버피팅이 억제됐다는 의미. 그리고 앞서와 달리 훈련 데이터에 대한 정확도가 100%에 도달하지 못한 점도 주목

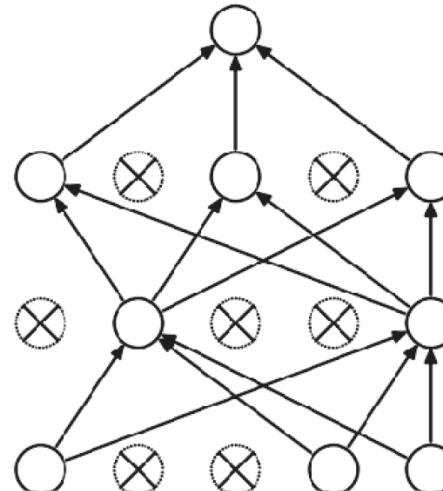
바른 학습을 위해

- 드롭아웃

- 신경망 모델이 복잡해지면 가중치 감소만으로는 대응하기 어려워짐
- 이럴 때 흔히 드롭아웃(Dropout)이라는 기법을 이용
- 드롭아웃은 뉴런을 임의로 삭제하면서 학습하는 방법
- 훈련 때 은닉층의 뉴런을 무작위로 골라 삭제함
- 삭제된 뉴런은 아래 그림과 같이 신호를 전달하지 않게 됨
- 훈련 데이터는 데이터를 훌릴 때마다 삭제한 뉴런을 무작위로 선택하고
- 시험 때는 모든 뉴런에 신호를 전달함
- 단, 시험 때는 각 뉴런의 출력에 훈련 때 삭제 안 한 비율을 곱하여 출력



(a) 일반 신경망



(b) 드롭아웃을 적용한 신경망

바른 학습을 위해

- 드롭아웃
 - 신경망 모델이 복잡해지면 가중치 감소만으로는 대응하기 어려워짐
 - 이럴 때 흔히 드롭아웃(Dropout)이라는 기법을 이용
 - 드롭아웃은 뉴런을 임의로 삭제하면서 학습하는 방법
 - 훈련 때 은닉층의 뉴런을 무작위로 골라 삭제함
 - 삭제된 뉴런은 아래 그림과 같이 신호를 전달하지 않게 됨
 - 훈련 데이터는 데이터를 훌릴 때마다 삭제한 뉴런을 무작위로 선택하고
 - 시험 때는 모든 뉴런에 신호를 전달함
 - 단, 시험 때는 각 뉴런의 출력에 훈련 때 삭제 안 한 비율을 곱하여 출력

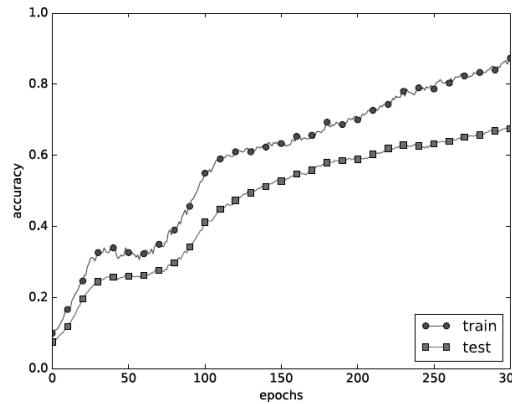
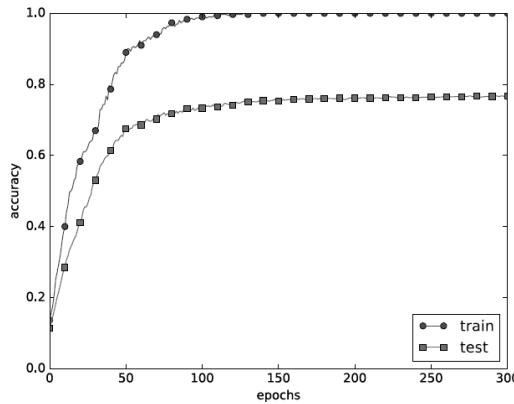
바른 학습을 위해

- 드롭아웃

```
class Dropout:  
    """  
        http://arxiv.org/abs/1207.0580  
    """  
  
    def __init__(self, dropout_ratio=0.5):  
        self.dropout_ratio = dropout_ratio  
        self.mask = None  
  
    def forward(self, x, train_flg=True):  
        if train_flg:  
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio  
            return x * self.mask  
        else:  
            return x * (1.0 - self.dropout_ratio)  
  
    def backward(self, dout):  
        return dout * self.mask
```

바른 학습을 위해

- 드롭아웃
 - 훈련 시에는 순전파 때마다 self.mask에 삭제할 뉴런을 False로 표시
 - self.mask는 x와 형상이 같은 배열을 무작위로 생성하고, 그 값이 dropout_ratio 보다 큰 원소만 True로 설정
 - 역전파 때의 동작은 ReLU와 같음. 즉, 순전파 때 신호를 통과시키는 뉴런은 역전파 때도 신호를 그대로 통과시키고, 순전파 때 통과시키지 않은 뉴런은 역전파 때도 신호를 차단
 - MNIST 데이터셋으로 확인한 드롭아웃의 효과 (오른쪽이 드롭아웃을 적용)



- 그림과 같이 드롭아웃을 적용했을 때 훈련 데이터와 시험 데이터에 대한 정확도 차이가 줄어듬. 훈련 데이터에 대한 정확도도 100%에 도달하지 않았음
- 이처럼 드롭아웃을 이용하면 표현력을 높이면서도 오버피팅을 억제할 수 있음

적절한 하이퍼파라미터 값 찾기

- 검증 데이터
 - 신경망에 등장하는 하이퍼파라미터의 예
뉴런 수, 배치 크기, 매개변수 갱신 시의 학습률, 가중치 감소 등
 - 이러한 하이퍼파라미터 값을 적절히 설정하지 않으면 모델의 성능이 크게 떨어지기도 함. 따라서 하이퍼파라미터의 값을 매우 중요
 - 그러나 그 값을 결정하기까지는 일반적으로 많은 시행착오를 겪음
 - 이렇게 하이퍼파라미터를 다양한 값으로 설정하고 검증할 때 주의할 점은 하이퍼파라미터의 성능을 평가할 때는 시험 데이터를 사용해서는 안된다는 것
 - 그 이유는 시험 데이터를 사용하여 하이퍼파라미터를 조정하면 하이퍼파라미터 값이 시험 데이터에 오버피팅되기 때문
 - 즉, 하이퍼파라미터 값의 '좋음'을 시험 데이터로 확인하게 되므로 하이퍼파라미터의 값이 시험데이터에만 적합하도록 조정되어 버림
 - 그렇게 되면 다른 데이터에는 적응하지 못하니 범용 성능이 떨어지는 모델이 됨
 - 그래서 하이퍼파라미터를 조정할 때는 하이퍼파라미터 전용 확인 데이터가 필요
 - 하이퍼파라미터 조정용 데이터를 일반적으로 검증 데이터(validation data)라고 함

적절한 하이퍼파라미터 값 찾기

- 검증 데이터

- Note.

- 훈련 데이터 : 매개변수 학습

- 검증 데이터 : 하이퍼파라미터 성능 평가

- 시험 데이터 : 신경망의 범용 성능 평가

```
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)
```

```
x_train, t_train = shuffle_dataset(x_train, t_train)
```

```
# 20%를 검증 데이터로 분할
```

```
validation_rate = 0.20
```

```
validation_num = int(x_train.shape[0] * validation_rate)
```

```
x_val = x_train[:validation_num]
```

```
t_val = t_train[:validation_num]
```

```
x_train = x_train[validation_num:]
```

```
t_train = t_train[validation_num:]
```

적절한 하이퍼파라미터 값 찾기

- 하이퍼 파라미터 최적화
 - 하이퍼파라미터를 최적화할 때의 핵심은 하이퍼파라미터의 '최적 값'이 존재하는 범위를 조금씩 줄여가는 것
 - 범위를 조금씩 줄이려면 우선 대략적인 범위를 설정하고 그 범위에서 무작위로 하이퍼파라미터 값을 골라낸(샘플링) 후, 그 값으로 정확도를 평가
 - 정확도를 살피면서 이 작업을 여러 번 반복하면서 하이퍼파라미터의 '최적 값' 범위를 좁혀감
 - 하이퍼파라미터의 범위는 '대략적으로' 지정하는 것이 효과적
 - 실제로도 0.001에서 1,000사이와 같이 '10의 거듭제곱' 단위로 범위를 지정
 - 이를 '로그 스케일로 지정'한다고 함
 - 하이퍼파라미터를 최적화할 때는 딥러닝 학습에는 오랜 시간이 걸린다는 점을 기억해야함.
 - 따라서 나쁠 듯한 값을 일찍 포기하고 학습을 위한 에폭을 작게하여, 1회 평가에 걸리는 시간을 단축하는 것이 효과적

적절한 하이퍼파라미터 값 찾기

- 하이퍼 파라미터 최적화 (정리)
 - 0단계
하이퍼파라미터 값의 범위를 설정
 - 1단계
설정됨 범위에서 하이퍼파라미터의 값을 무작위로 추출
 - 2단계
1단계에서 샘플링한 하이퍼파라미터 값을 사용하여 학습하고, 검증 데이터로 정확도를 평가(단, 에폭은 작게 설정)
 - 3단계
1단계와 2단계를 특정 횟수(100회 등) 반복하며, 그 정확도의 결과를 보고 하이퍼파라미터의 범위를 좁힘

적절한 하이퍼파라미터 값 찾기

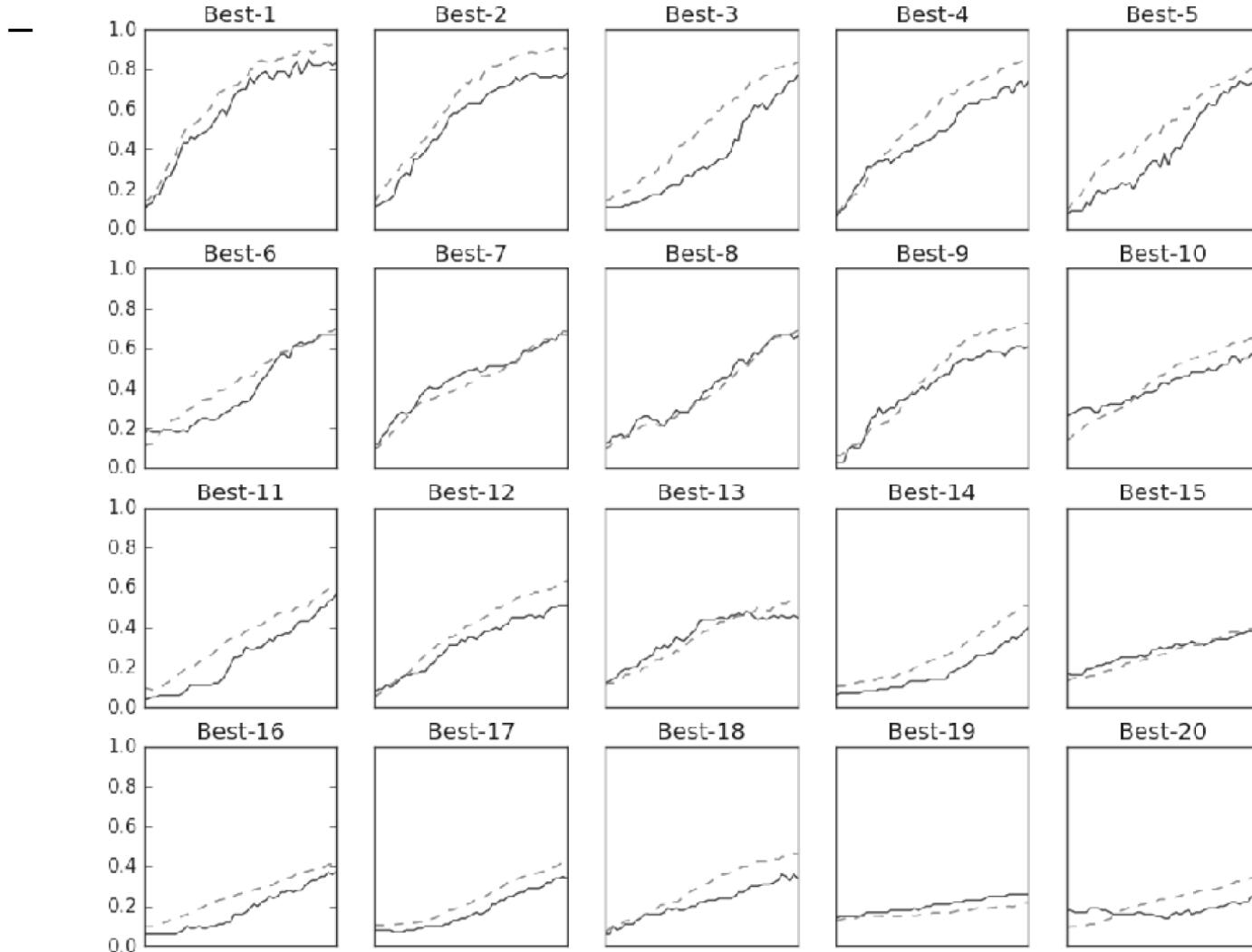
- 하이퍼 파라미터 최적화 구현하기
 - 가중치 감소를 $10^{-8} \sim 10^{-4}$, 학습률을 $10^{-6} \sim 10^{-2}$ 범위부터 시작
 - 이 경우 하이퍼파라미터 무작위 추출코드는 다음과 같음

```
weight_decay = 10 ** np.random.uniform(-8, -4)  
lr = 10 ** np.random.uniform(-6, -2)
```

- 이렇게 무작위로 추출한 값을 사용하여 학습 수행
- 그 후에는 여러 차례 다양한 하이퍼파라미터 값으로 학습을 반복하며 신경망에 좋을 것 같은 값이 어디에 존재하는지 관찰

적절한 하이퍼파라미터 값 찾기

- 하이퍼 파라미터 최적화 구현하기
 - 실선은 검증 데이터에 대한 정확도, 점선은 훈련 데이터에 대한 정확도



적절한 하이퍼파라미터 값 찾기

- 하이퍼 파라미터 최적화 구현하기
 - 'Best-5' 까지의 하이퍼파라미터의 값(학습률과 가중치 계수)

```
Best-1 (val acc:0.83) | lr:0.0092, weight decay:3.86e-07
Best-2 (val acc:0.78) | lr:0.00956, weight decay:6.04e-07
Best-3 (val acc:0.77) | lr:0.00571, weight decay:1.27e-06
Best-4 (val acc:0.74) | lr:0.00626, weight decay:1.43e-05
Best-5 (val acc:0.73) | lr:0.0052, weight decay:8.97e-06
```

- 이 결과를 보면 학습이 잘 진행될 때의 학습률은 0.001~0.01, 가중치 감소 계수는 $10^{-8} \sim 10^{-6}$ 정도라는 것을 알 수 있음
- 이처럼 잘될 것 같은 값의 범위를 관찰하고 범위를 좁혀감
- 그런 다음 축소된 범위로 똑같은 작업을 반복
- 이렇게 적절한 값이 위치한 범위를 좁혀가다가 특정 단계에서 최종 하이퍼파라미터 값을 하나 선택

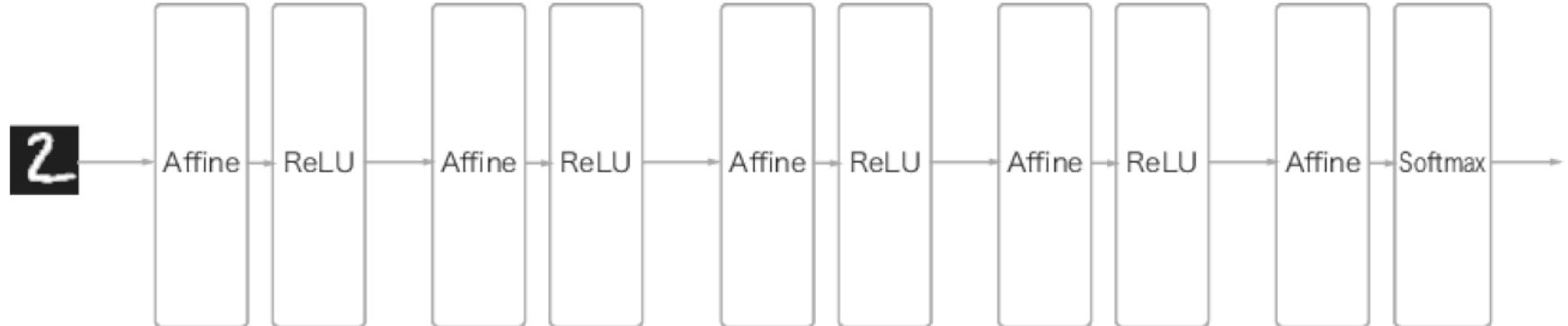
정리

- 학습 관련 기술들
 - 매개변수 갱신 방법에는 확률적 경사 하강법(SGD) 외에도 모멘텀, AdaGrad, Adam 등이 있음
 - 가중치 초기값을 정하는 방법은 올바른 학습을 하는 데 매우 중요
 - 가중치의 초기값으로는 'Xavier 초기값'과 'He 초기값'이 효과적
 - 배치 정규화를 이용하면 학습을 빠르게 진행할 수 있으며, 초기값에 영향을 덜 받게 됨
 - 오버피팅을 억제하는 정규화 기술로는 가중치 감소와 드롭아웃이 있음
 - 하이퍼파라미터 값 탐색은 최적 값이 존재할 범한 범위를 점차 좁히면서 하는 것이 효과적

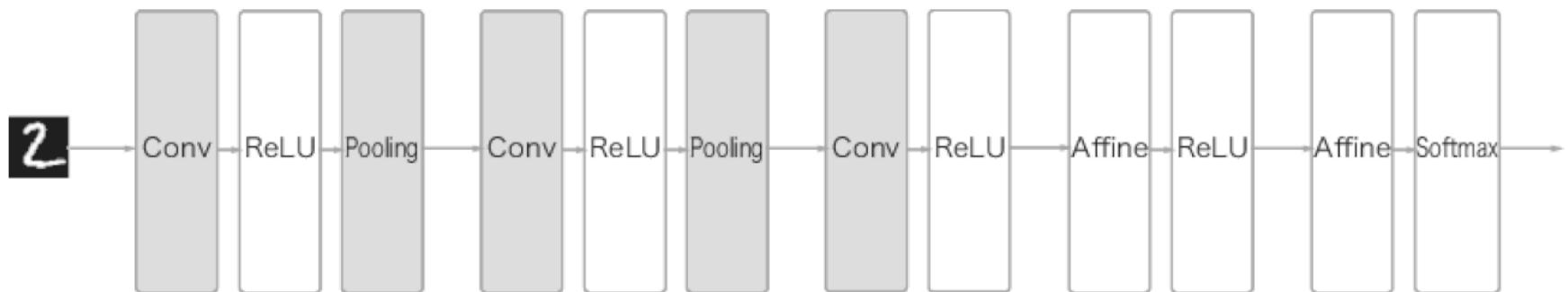
합성곱 신경망(CNN)

전체 구조

- 완전연결 계층(Affine 계층)으로 이루어진 네트워크 예



- CNN으로 이루어진 네트워크 예 : 합성곱 계층과 풀링 계층이 새로 추가



합성곱 계층

- 완전연결 계층의 문제점
 - 완전연결 계층에서는 인접하는 계층의 뉴런이 모두 연결되고 출력의 수는 임의로 정할 수 있었음
 - 그러나 이 계층의 문제는 '데이터의 형상이 무시'된다는 점
 - 입력 데이터가 이미지인 경우 통상 세로, 가로, 채널(색상)로 구성된 3차원 데이터
 - 그러나 완전연결 계층에 입력할 때는 3차원 데이터를 평평한 1차원 데이터로 평탄화해줘야 함
 - MNIST 데이터의 경우 (1채널, 세로 28픽셀, 가로 28픽셀)를 1줄로 세운 784개의 데이터를 Affine 계층에 입력했음
 - 이미지는 3차원 영상이며, 이 형상에는 소중한 공간적 정보가 담겨 있음
 - 예를 들어 공간적으로 가까운 픽셀은 값이 비슷하거나, RGB의 각 채널은 서로 밀접하게 관련되어 있거나, 거리가 먼 픽셀끼리는 별 연관이 없는 등, 3차원 속에서 의미를 갖는 본질적인 패턴이 숨어 있을 것임
 - 그러나 완전연결 계층은 형상을 무시하고 모든 입력 데이터를 동등한 뉴런(같은 차원의 뉴런)으로 취급하여 형상에 담긴 정보를 살릴 수 없음

합성곱 계층

- 완전연결 계층의 문제점
 - 합성곱 계층은 형상을 유지
 - 이미지도 3차원 데이터로 입력받으며, 다음계층에도 3차원 데이터로 전달
 - 그래서 CNN에서는 이미지처럼 형상을 가진 데이터를 제대로 이해할 것임
 - CNN에서는 합성곱 계층의 입출력 데이터를 특징 맵(feature map)이라고 함
 - 합성곱 계층의 입력 데이터를 입력 특징 맵(input feature map), 출력 데이터를 출력 특징 맵(output feature map)

합성곱 계층

- 합성곱 연산

- 합성곱 계층에서는 합성곱 연산을 처리. 이미지 처리에서 말하는 필터 연산임
- 합성곱 연산의 예

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

(*)

2	0	1
0	1	2
1	0	2



15	16
6	15

입력 데이터

필터

- 입력 데이터는 세로, 가로 방향의 형상을 가졌고, 필터 역시 세로, 가로 방향의 차원을 가짐
- 데이터의 필터와 형상을 (높이(height), 너비(width))로 표기하면, 이 예에서는 입력은 (4, 4), 필터는 (3, 3), 출력은 (2, 2). 필터를 커널이라고도 함

합성곱 계층

- 합성곱 연산
 - 합성곱 연산은 필터의 윈도우(window)를 일정 간격으로 이동해가며 입력 데이터에 적용
 - 여기서 말하는 윈도우는 다음 그림의 회색 3×3 부분을 가리킴
 - 입력과 필터에 대응하는 원소끼리 곱한 후 그 총합을 구함
 - 그리고 그 결과를 출력의 해당 장소에 저장
 - 이 과정을 모든 장소에서 수행하면 합성곱 연산의 출력이 완성

합성곱 계층

- 합성곱 연산의 계산 순서

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

*

2	0	1
0	1	2
1	0	2



15	

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

*

2	0	1
0	1	2
1	0	2



15	16

합성곱 계층

- 합성곱 연산의 계산 순서

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

※

2	0	1
0	1	2
1	0	2



15	16
6	

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

※

2	0	1
0	1	2
1	0	2



15	16
6	15

합성곱 계층

- 합성곱 연산의 편향
 - 필터를 적용한 원소에 고정값(편향)을 더함

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

입력 데이터

⊗

2	0	1
0	1	2
1	0	2

필터



15	16
6	15

+

3



18	19
9	18

편향

출력 데이터

합성곱 계층

- 패딩

- 합성곱 연산 수행 전에 입력 데이터 주변을 특정 값(예컨대 0)으로 채우는 것을 패딩(padding)이라고 하며, 합성곱 연산에서 자주 이용
- 합성곱 연산의 패딩 처리

1	2	3	0	
0	1	2	3	
3	0	1	2	
2	3	0	1	

(4, 4)

입력 데이터(패딩 : 1)



2	0	1
0	1	2
1	0	2

(3, 3)

필터



7	12	10	2
4	15	16	10
10	6	15	6
8	10	4	3

(4, 4)

출력 데이터

합성곱 계층

- 패딩

- Note.

패딩은 주로 출력 크기를 조정할 목적으로 사용

합성곱 연산을 되풀이 하다보면 어느 시점에 출력 크기가 1이 되어 버리는데, 이는 합성곱 연산을 적용할 수 없다는 의미로 신경망 학습에서 문제가 됨

이러한 사태를 막기 위해 패딩을 사용

패딩의 폭을 1로 설정하니 (4, 4) 입력에 대한 출력이 같은 크기인 (4, 4)로 유지 즉, 입력 데이터의 공간적 크기를 고정한 채로 다음 계층에 전달할 수 있음

합성곱 계층

- **스트라이드**

- 필터를 적용하는 위치의 간격을 스트라이드(stride)라고 함
- 스트라이드가 2인 합성곱 연산

1	2	3	0	1	2	3
0	1	2	3	0	1	2
3	0	1	2	3	0	1
2	3	0	1	2	3	0
1	2	3	0	1	2	3
0	1	2	3	0	1	2
3	0	1	2	3	0	1

⊗

2	0	1
0	1	2
1	0	2



15		

스트라이드 : 2

1	2	3	0	1	2	3
0	1	2	3	0	1	2
3	0	1	2	3	0	1
2	3	0	1	2	3	0
1	2	3	0	1	2	3
0	1	2	3	0	1	2
3	0	1	2	3	0	1

⊗

2	0	1
0	1	2
1	0	2



15	17	

합성곱 계층

-

스트라이드

- 스트라이드를 키우면 출력 크기는 작아짐
- 입력 크기를 (H , W), 필터 크기를 (FG , FW), 출력 크기를 (OH , OW), 패딩을 P , 스트라이드를 S 라 하면, 출력 크기는 다음 식으로 계산

예 1 : [그림 7-6]의 예

$$OH = \frac{H + 2P - FH}{S} + 1$$

입력 : (4, 4), 패딩 : 1, 스트라이드 : 1, 필터 : (3, 3)

$$OH = \frac{4+2 \cdot 1 - 3}{1} + 1 = 4$$

$$OW = \frac{W + 2P - FW}{S} + 1$$

$$OW = \frac{4+2 \cdot 1 - 3}{1} + 1 = 4$$

예 2 : [그림 7-7]의 예

입력 : (7, 7), 패딩 : 0, 스트라이드 : 2, 필터 : (3, 3)

$$OH = \frac{7+2 \cdot 0 - 3}{2} + 1 = 3$$

$$OW = \frac{7+2 \cdot 0 - 3}{2} + 1 = 3$$

예 3

입력 : (28, 31), 패딩 : 2, 스트라이드 : 3, 필터 : (5, 5)

$$OH = \frac{28+2 \cdot 2 - 5}{3} + 1 = 10$$

$$OW = \frac{31+2 \cdot 2 - 5}{3} + 1 = 11$$

합성곱 계층

- 3차원 데이터의 합성곱 연산
 - 3차원 데이터 합성곱 연산의 예

	4	2	1	2
3	0	6	5	4
1	2	3	0	3
0	1	2	3	2

1	2	3	0	3
0	1	2	3	2
3	0	1	2	1
2	3	0	1	

입력 데이터



	4	0	2
0	1	3	0
2	0	1	2
0	1	2	0

2	0	1	2
0	1	2	0
1	0	2	

필터

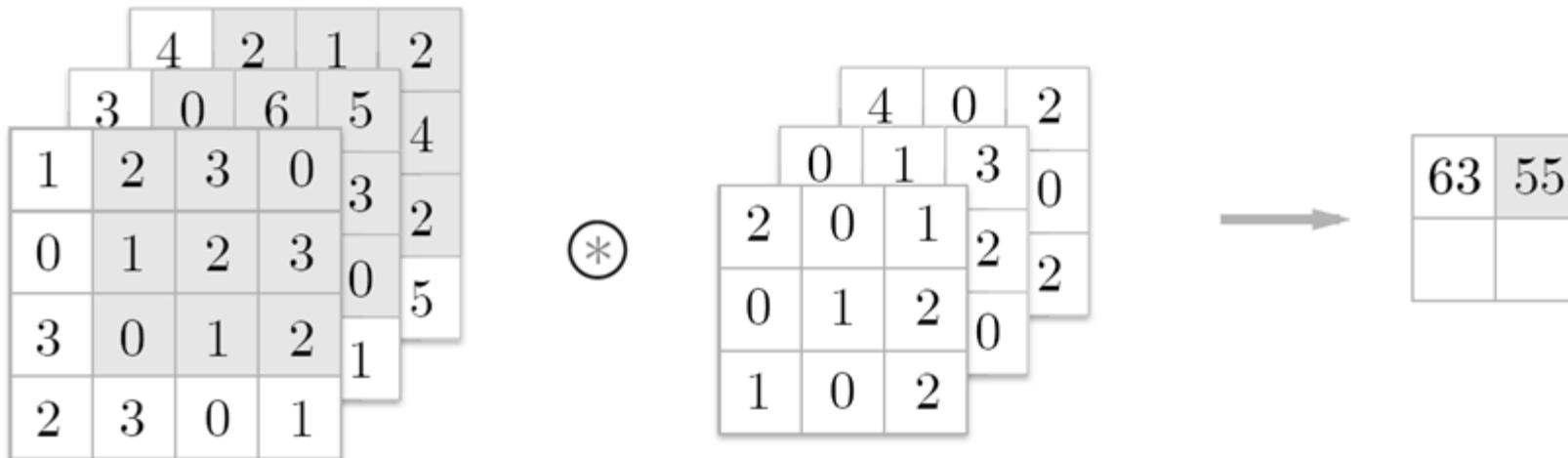
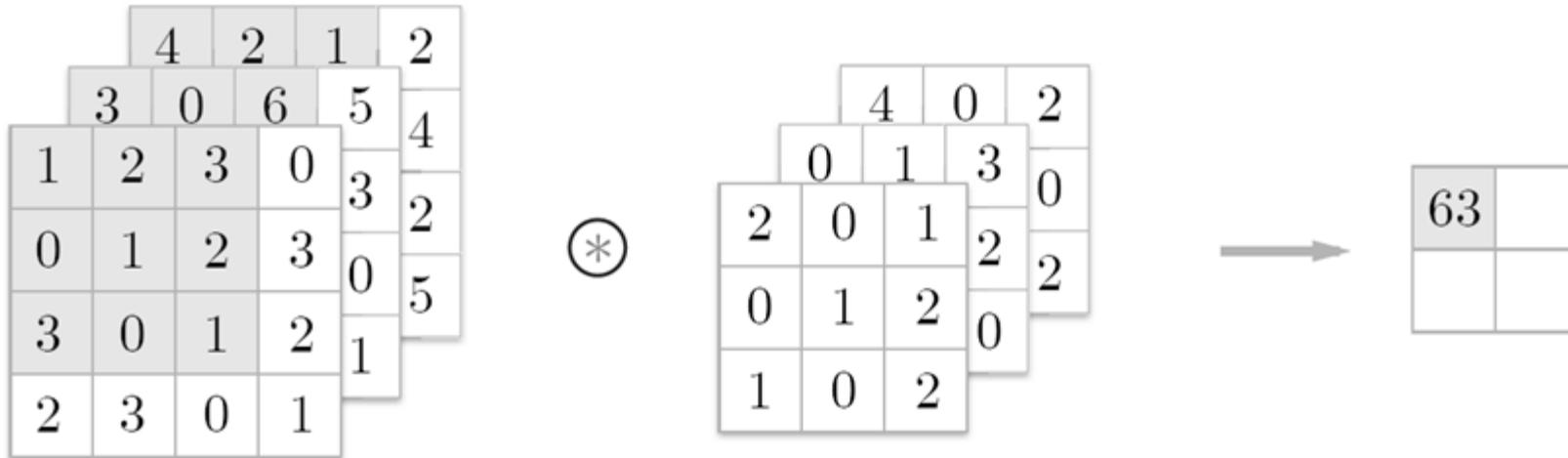


63	55
18	51

출력 데이터

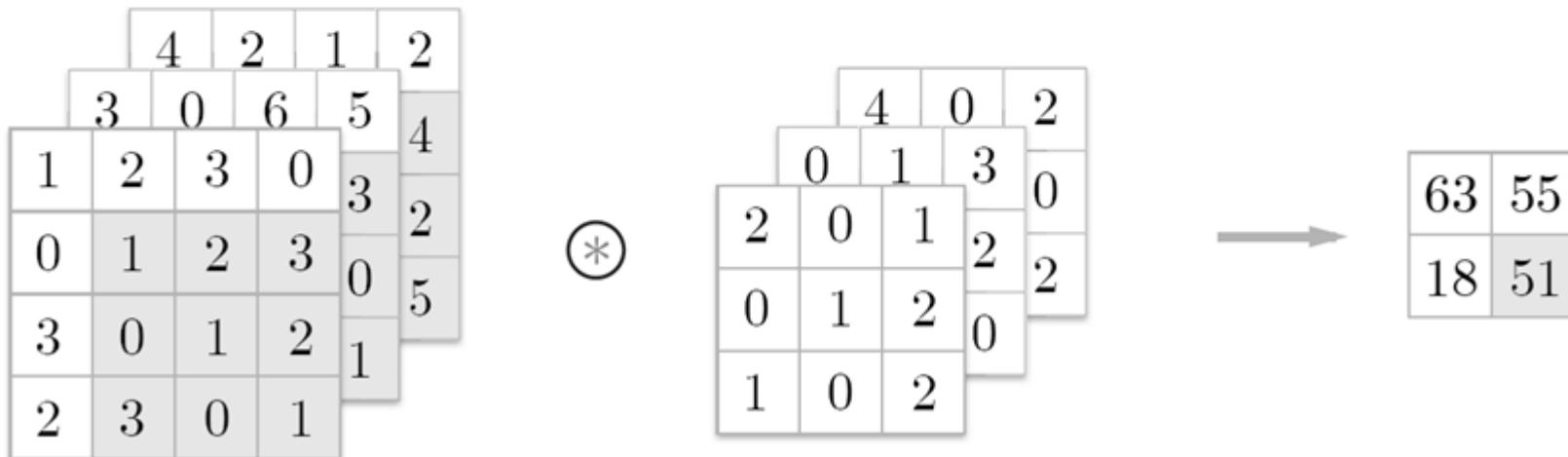
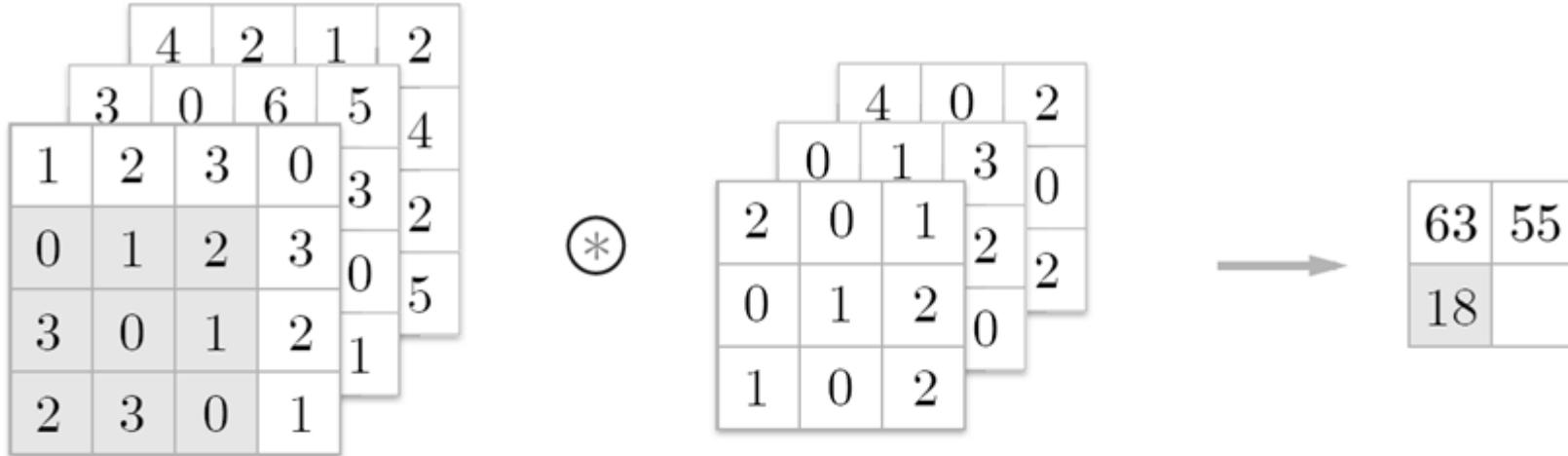
합성곱 계층

- 3차원 데이터의 합성곱 연산의 계산 순서



합성곱 계층

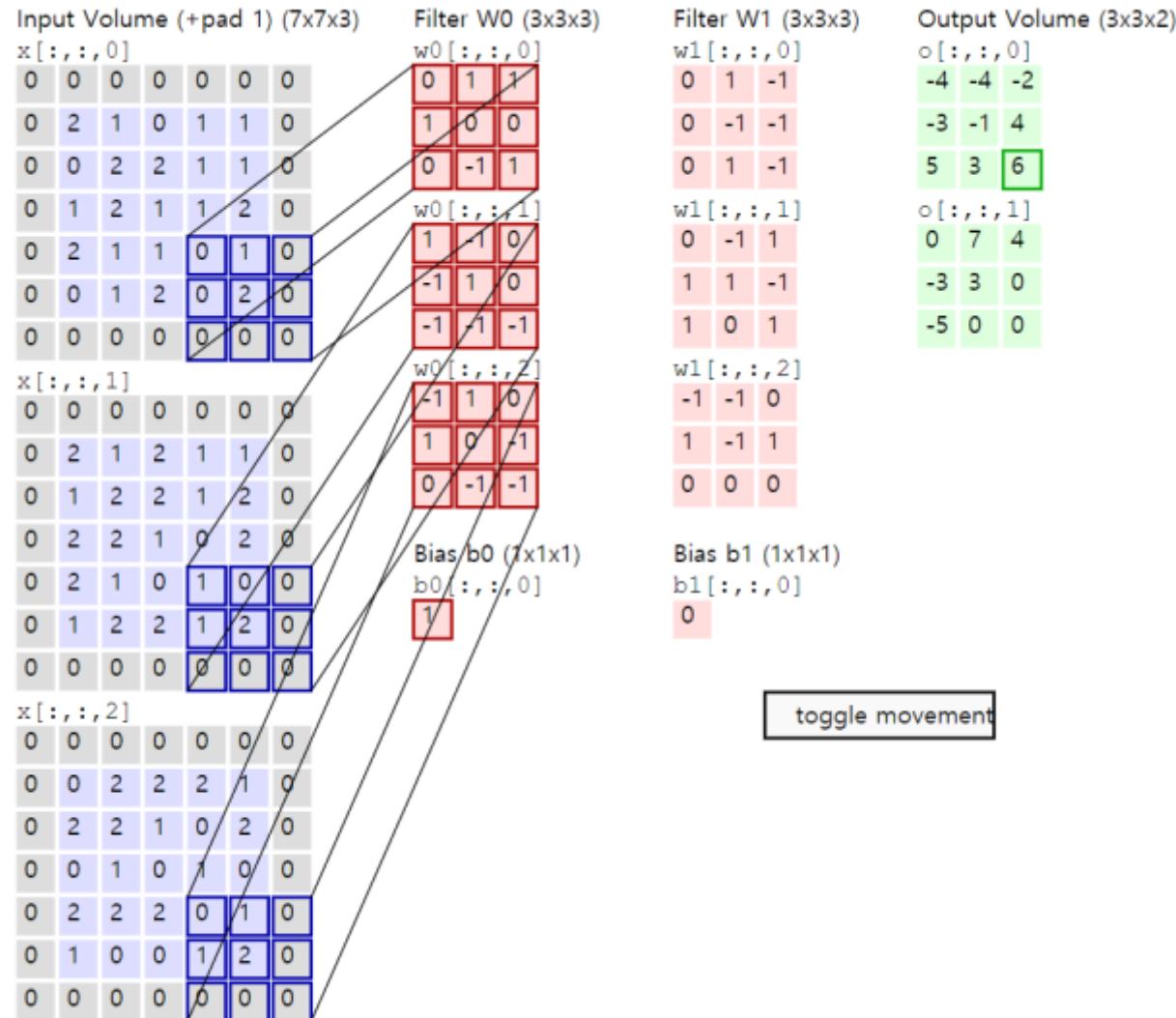
- 3차원 데이터의 합성곱 연산의 계산 순서



합성곱 계층

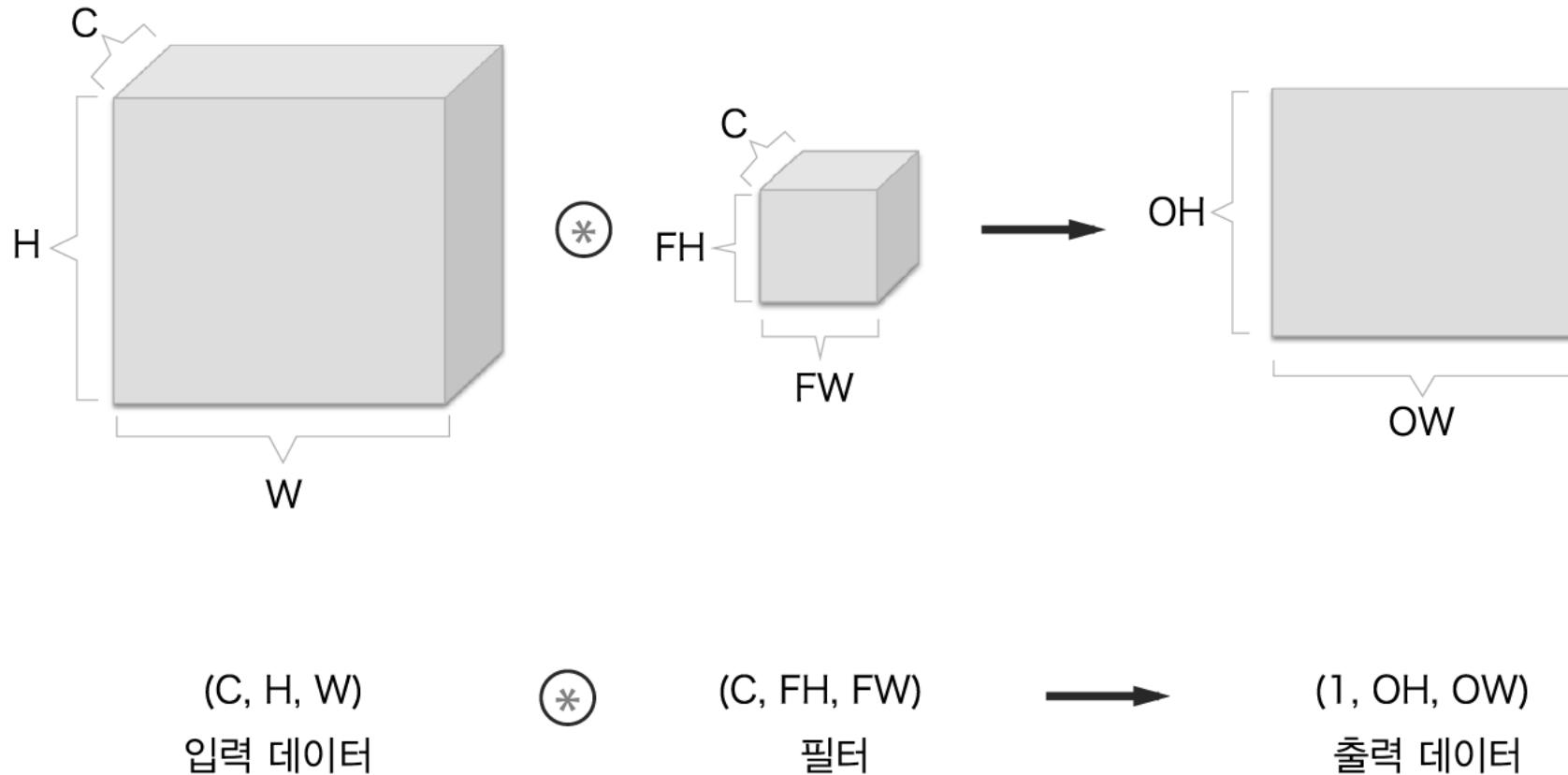
- 3차원 데이터의 합성곱 연산

- [CS231n Convolutional Neural Networks for Visual Recognition](#)



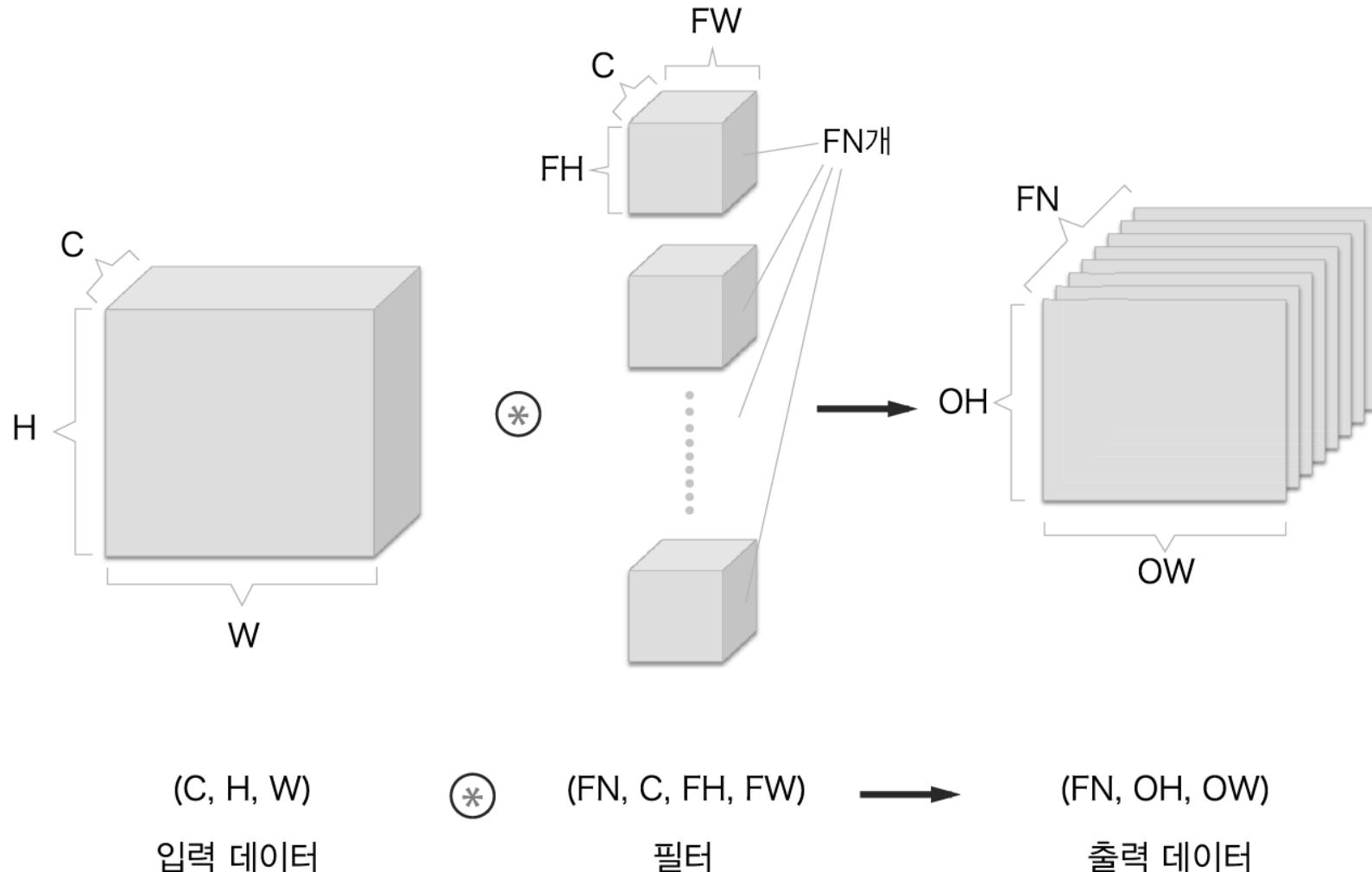
합성곱 계층

- 3차원 데이터의 합성곱 연산
 - 합성곱 연산을 직육면체 블록으로 생각하기



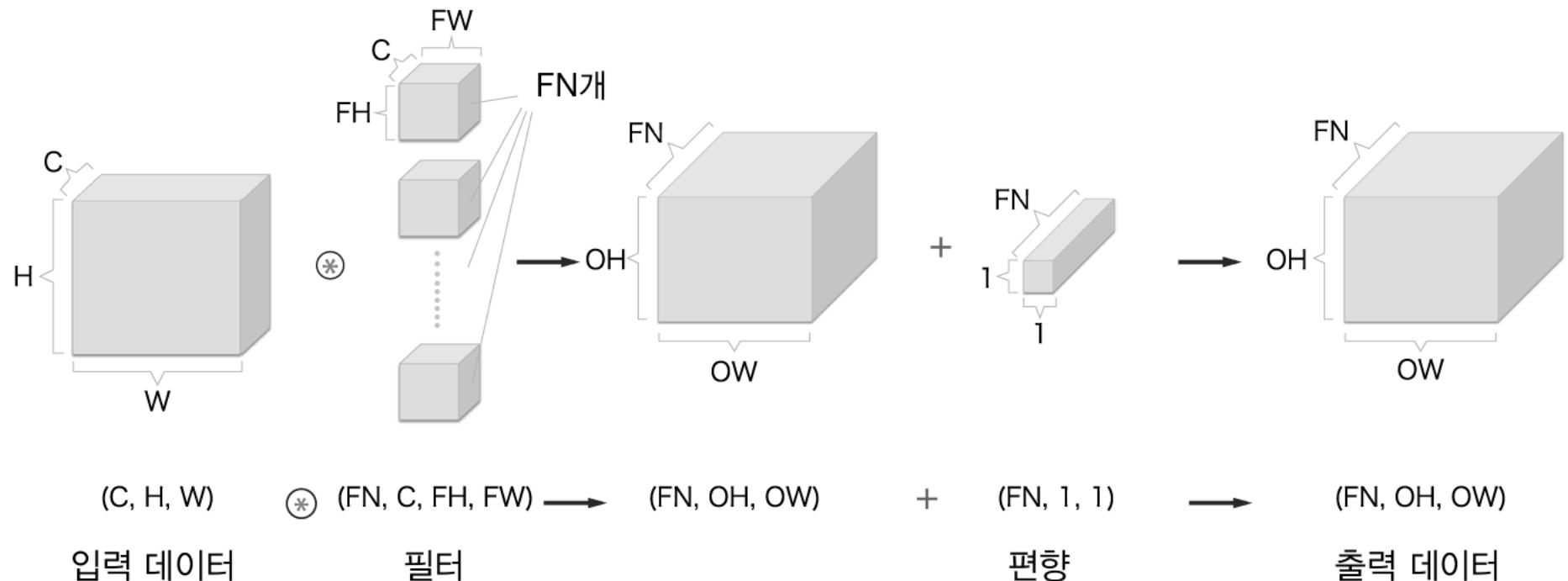
합성곱 계층

- 3차원 데이터의 합성곱 연산
 - 여러 필터를 사용한 합성곱 연산의 예



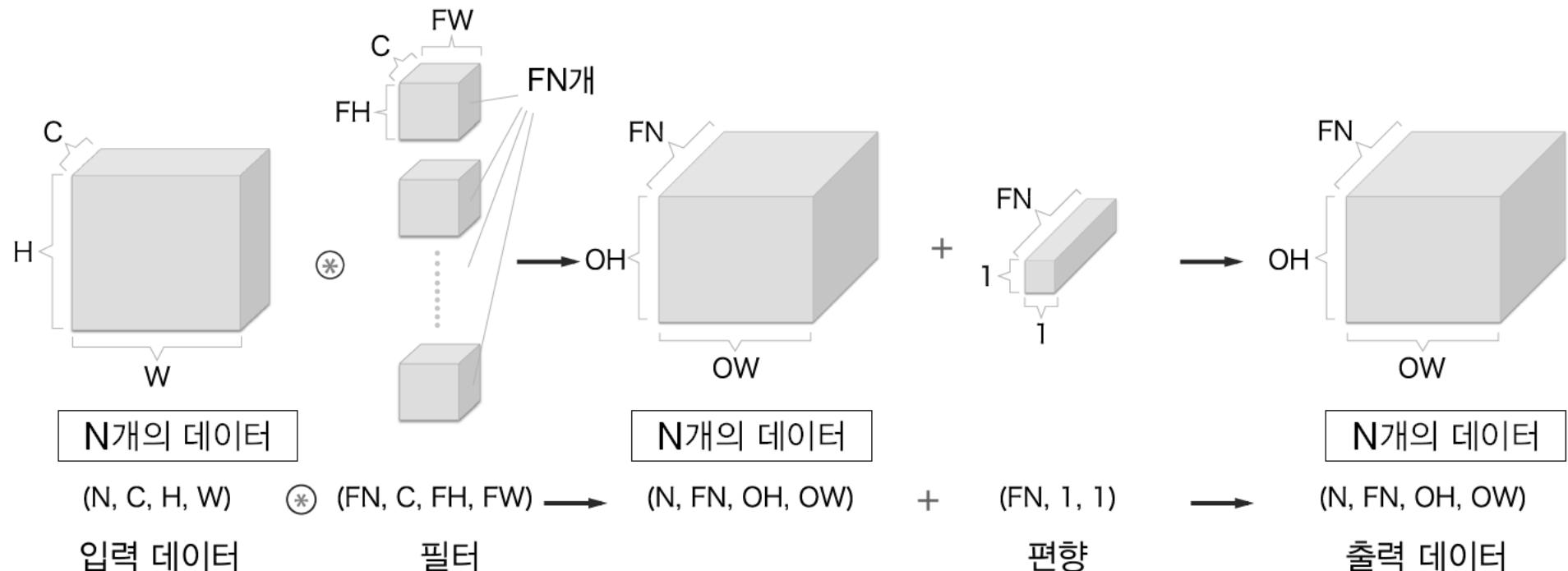
합성곱 계층

- 3차원 데이터의 합성곱 연산
 - 합성곱 연산의 처리 흐름(편향 추가)



합성곱 계층

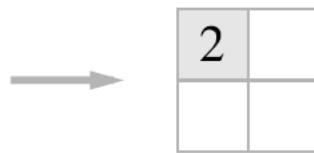
- 배치 처리
 - 합성곱 연산의 처리 흐름(배치)



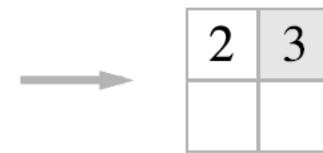
풀링 계층

- 풀링 계층
 - 최대 풀링의 처리 순서

1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



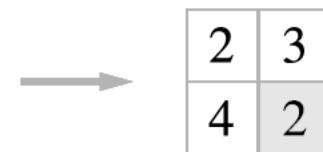
1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



풀링 계층

- 풀링 계층의 특징
 - 학습해야 할 매개변수가 없다
 - 채널 수가 변하지 않는다

	4	2	1	2
3	0	6	5	4
1	2	1	0	3
0	1	2	3	2
3	0	1	2	0
2	4	0	1	5

입력 데이터



4	4
3	6
2	5
2	3
4	2

출력 데이터

- 입력의 변화에 영향을 적게 받는다(강건하다)

1	2	0	7	1	0
0	9	2	3	2	3
3	0	1	2	1	2
2	4	0	1	0	1
6	0	1	2	1	2
2	4	0	1	8	1



9	7
6	8

1	1	2	0	7	1
3	0	9	2	3	2
2	3	0	1	2	1
3	2	4	0	1	0
2	6	0	1	2	1
1	2	4	0	1	8



9	7
6	8

합성곱/풀링 계층 구현하기

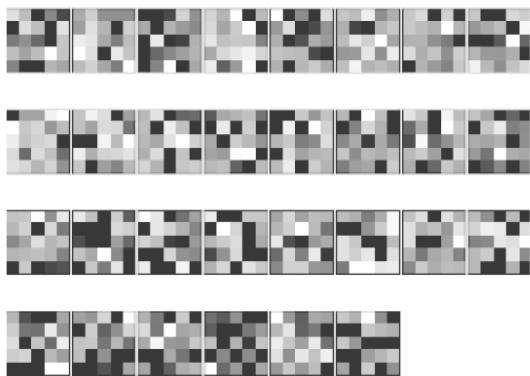
-

CNN 시각화하기

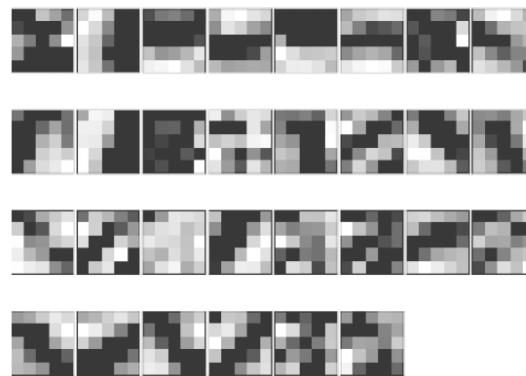
- 1번째 층의 가중치 시각화하기

- MNIST 데이터셋으로 CNN 학습을 했을 경우 1번째 층의 합성곱 계층의 가중치는 그 형상이 (30, 1, 5, 5). 즉, (필터 30개, 채널 1개, 5×5 크기)
- 필터의 크기가 5×5 이고 채널이 1개라는 것은 이 필터를 1채널의 회색조 이미지로 시각화할 수 있다는 의미
- 학습 전과 후의 1번째 층의 합성곱 계층의 가중치

학습 전



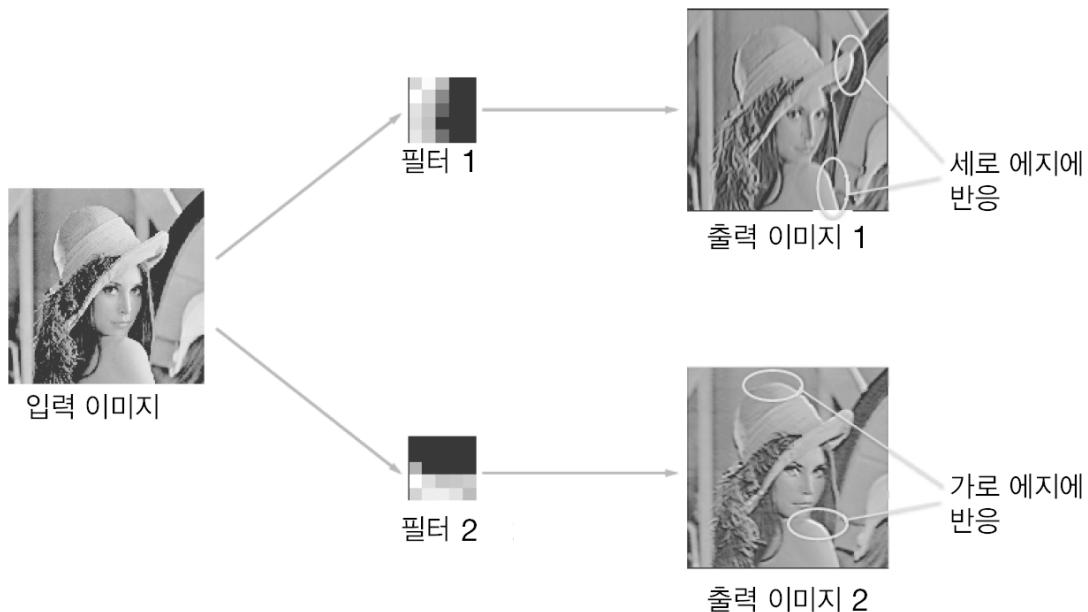
학습 후



- 학습 전 필터는 무작위로 초기화되어 있어 흑백의 정도에 규칙성이 없음
- 한편, 학습을 마친 필터는 규칙성 있는 이미지가 됨
- 흰색에서 검은색으로 점차 변화하는 필터, 덩어리(블롭, blob)가 진 필터 등 규칙을 띠는 필터로 바뀜

CNN 시각화하기

- 1번째 층의 가중치 시각화하기
 - 그런데 이 필터는 '무엇을 보고 있는' 것일까?
 - 그것은 에지(색상이 바뀐 경계선)와 블롭(국소적으로 덩어리진 영역) 등을 보고 있음
 - 가령 왼쪽 절반이 흰색이고 오른쪽 절반이 검은색인 필터는 아래 그림과 같이 세로 방향의 에지에 반응하는 필터임



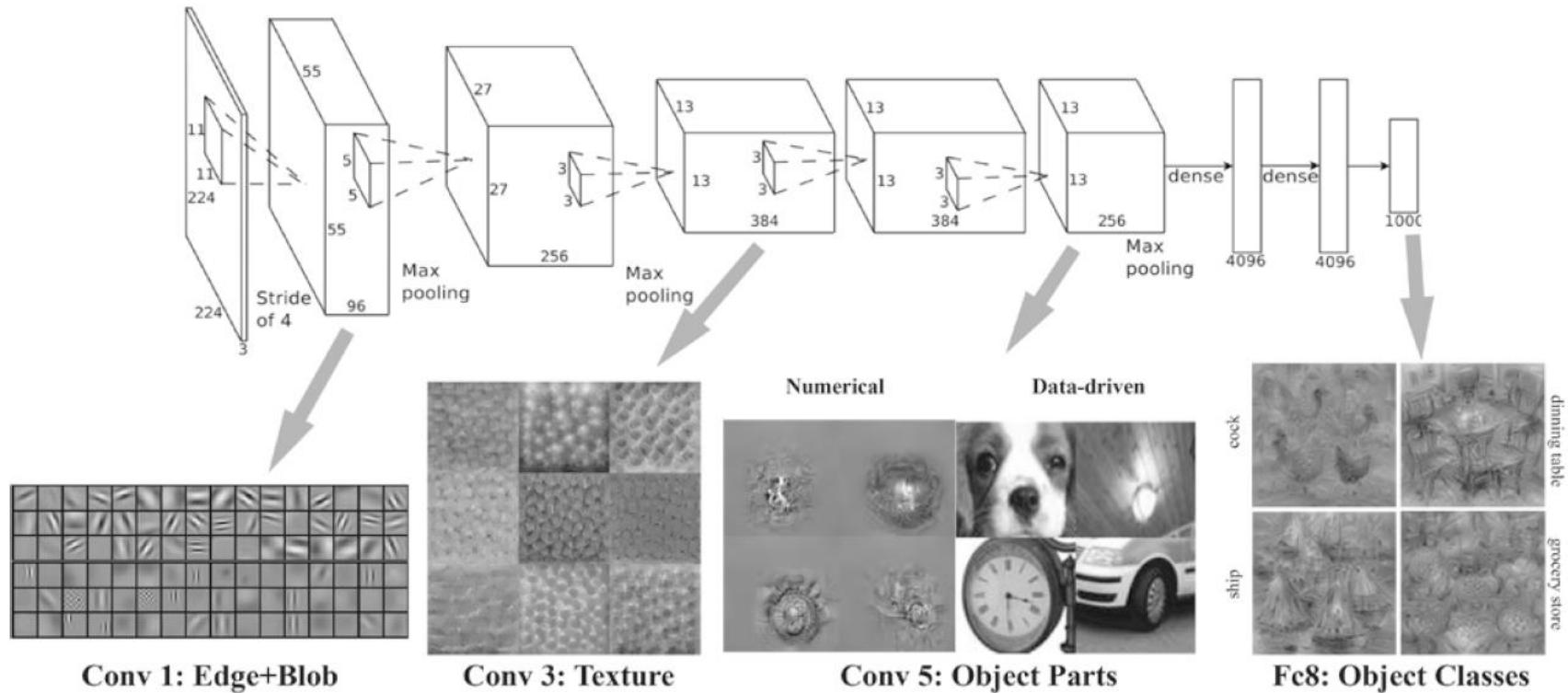
- 위 그림은 학습된 필터 2개를 선택하여 입력 이미지에 합성곱 처리를 한 결과로, '필터 1'은 세로 에지에 반응하며 '필터 2'는 가로 에지에 반응

CNN 시작화하기

- 층 깊이에 따른 추출 정보 변화
 - 앞의 결과는 1번째 층의 합성곱 계층을 대상으로 한 것임
 - 1번째 층의 합성곱 계층에서는 에지나 블록 등의 저수준 정보가 추출된다 치고, 그럼 겹겹이 쌓인 CNN의 각 계층에서는 어떤 정보가 추출될까?
 - 계층이 깊어질수록 추출되는 정보는 더 추상화됨
 - 다음 장의 그림은 일반 사물 인식(자동차나 개 등)을 수행한 8층의 CNN
 - 이 네트워크 구조는 AlexNet이라 하는데, 합성곱 계층과 풀링 계층을 여러 겹 쌓고, 마지막으로 완전연결 계층을 거쳐 결과를 출력하는 구조
 - 그림에서 블록으로 나타낸 것은 중간 데이터이며, 그 중간 데이터에 합성곱 연산을 연속해서 적용

CNN 시작화하기

- 층 깊이에 따른 추출 정보 변화



- 딥러닝의 흥미로운 점은 합성곱 계층을 여러 겹 쌓으면, 층이 깊어지면서 더 복잡하고 추상화된 정보가 추출된다는 것
- 처음 층은 단순한 에지에 반응하고, 이어서 텍스처에 반응하고, 더 복잡한 사물의 일부에 반응하도록 변화. 즉, 층이 깊어지면서 뉴런이 반응하는 대상이 단순한 모양에서 '고급' 정보로 변화. 즉, 사물의 '의미'를 이해하도록 변화