Contents

- CHAPTER 5 오차역전파법
  - 5.1 계산 그래프
    - 5.1.1 계산 그래프로 풀다
    - 5.1.2 국소적 계산
    - 5.1.3 왜 계산 그래프로 푸는가?
  - 5.2 연쇄법칙
    - 5.2.1 계산 그래프의 역전파
    - 5.2.2 연쇄법칙이란?
    - 5.2.3 연쇄법칙과 계산 그래프
  - 5.3 역전파
    - 5.3.1 덧셈 노드의 역전파
    - 5.3.2 곱셈 노드의 역전파
    - 5.3.3 사과 쇼핑의 예
  - 5.4 단순한 계층 구현하기
    - 5.4.1 곱셈 계층
    - 5.4.2 덧셈 계층
  - 5.5 활성화 함수 계층 구현하기
    - 5.5.1 ReLU 계층
    - 5.5.2 Sigmoid 계층
  - 5.6 Affine/Softmax 계층 구현하기
    - 5.6.1 Affine 계층
    - 5.6.2 배치용 Affine 계층
    - 5.6.3 Softmax-with-Loss 계층
  - 5.7 오차역전파법 구현하기
    - 5.7.1 신경망 학습의 전체 그림
    - 5.7.2 오차역전파법을 적용한 신경망 구현하기
    - 5.7.3 오차역전파법으로 구한 기울기 검증하기
    - 5.7.4 오차역전파법을 사용한 학습 구현하기
  - 5.8 정리

CHAPTER 5 오차역전파법

4장에서 경사하강법을 이용한 신경망 학습을 공부했는데,
이 때, 수치 미분을 이용해 매개변수의 기울기를 구했습니다.
수치 미분은 간단하고 구현하기 쉽지만 시간이 오래 걸린다는 게 단점입니다.

5장에서는, 가중치 매개변수의 기울기를 효율적으로 계산하는 오차역전파법(backpropagation)을 공부하겠습니다.
수식을 이용해서 오차역전파법을 이해할 수 있지만, 우리는 계산 그래프를 이용해서 '시각적 ' 으로 이해하려 합니다.

5.4. 단순한 계층 구현하기

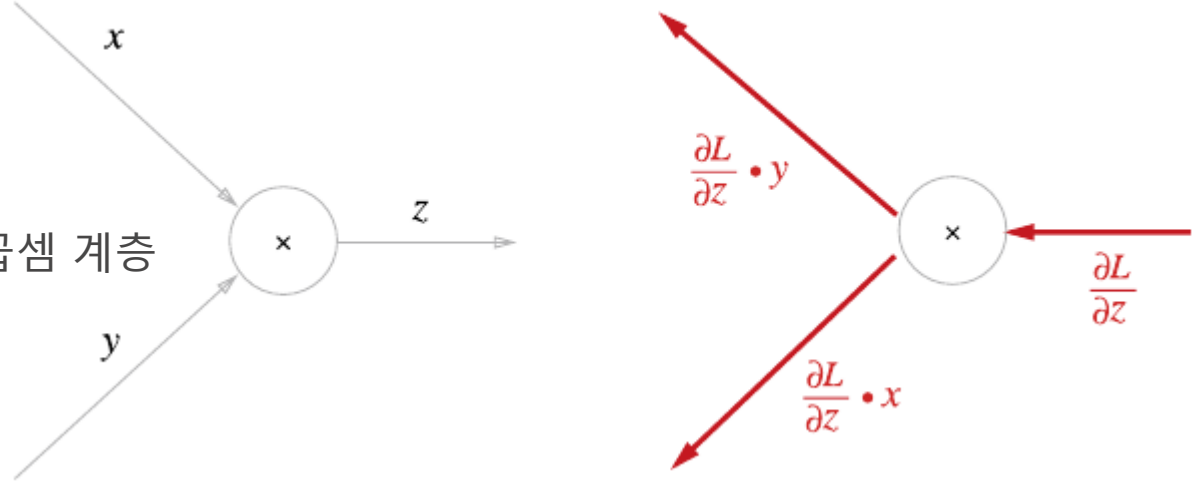
복습

5.4.1 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
```

그림 5-12 곱셈 노드의 역전파 : 왼쪽이 순전파, 오른쪽이 역전파다.

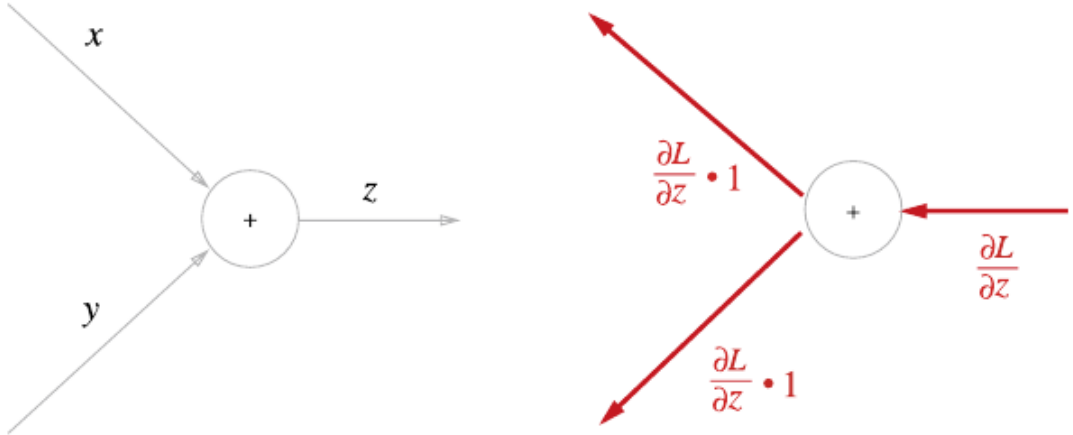
5.4.1 곱셈 계층



5.4.2 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
```

그림 5-9 덧셈 노드의 역전파 : 왼쪽이 순전파, 오른쪽이 역전파다. 덧셈 노드의 역전파는 입력 값을 그대로 흘려보낸다.



5.5 활성화 함수 계층 구현하기

계산 그래프를 신경망에 적용하고자 한다.

먼저 활성화 함수 계층에 대한 계산 그래프를 만들어 보자.

먼저 Relu와 Sigmoid 활성화 함수를 계산 그래프로 구현한다. 그 다음, Affine 계층과 Softmax 계층을 구현한다.

5.5.1 ReLU 계층

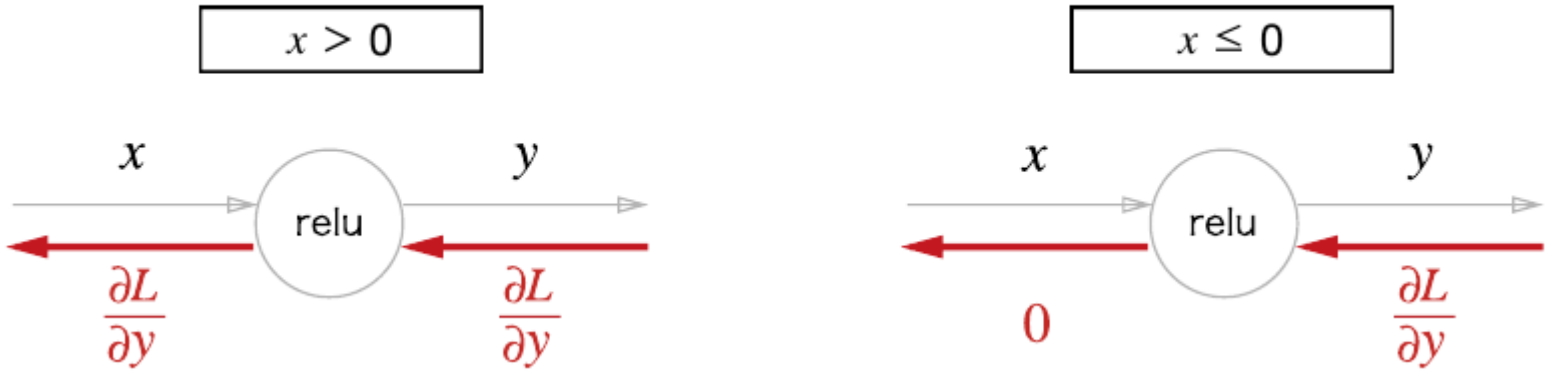
활성화 함수로 사용되는 ReLU 함수는 다음과 같이 정의한다.

$$y = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases} \quad \text{[식 5.7]}$$

Relu 함수의 미분은 다음과 같이 정의 한다.

$$\frac{\partial y}{\partial x} = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases} \quad \text{[식 5.8]}$$

그림 5-18 ReLU 계층의 계산 그래프



순전파 때 계층에 입력된 값의 부호에 따라, 역전파 값이 달라진다.

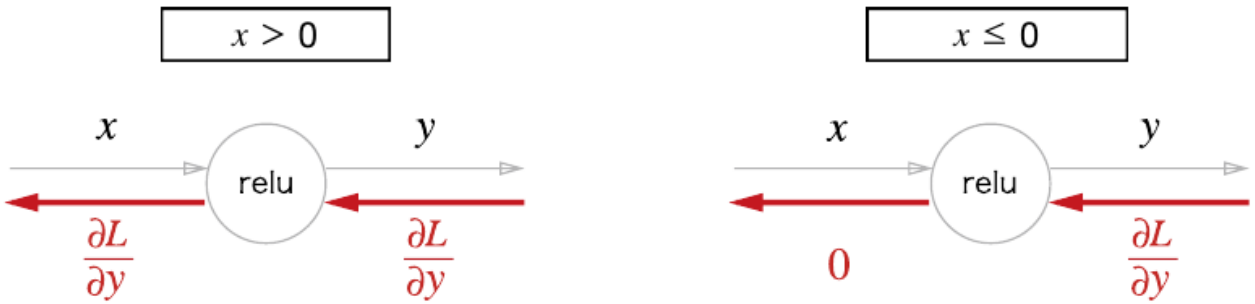
5.5.1 ReLU 계층 파이썬 구현

왜 노드가 아니라, 계층이라 할까?

```
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
```

common/layers.py

그림 5-18 ReLU 계층의 계산 그래프



Relu 클래스는 mask라는 인스턴스 변수를 가진다. mask는 True/False로 구성된 넘파이 배열로, 순전파의 입력인 x의 원소 값이 0 이하인 인덱스는 True, 그 외(0보다 큰 원소)는False로 유지한다

순전파시, (입력 값까지는 필요없고) 입력 값의 부호를 저장하고 있다가, 역전파시 음수는 0을 할당한다.

5.5.1 ReLU 계층 파이썬 구현 (각자 연습)

```
>>> x = np.array( [[1.0, -0.5], [-2.0, 3.0]] )
>>> print(x)
[[ 1.  -0.5]
 [-2.   3. ]]
>>> mask = (x <= 0)
>>> print(mask)
[[False  True]
 [ True False]]
```

5.5.2 Sigmoid 계층

$$y = \frac{1}{1 + \exp(-x)} \qquad [식\ 5.9]$$

계산 그래프:

그림 5-19 Sigmoid 계층의 계산 그래프(순전파)

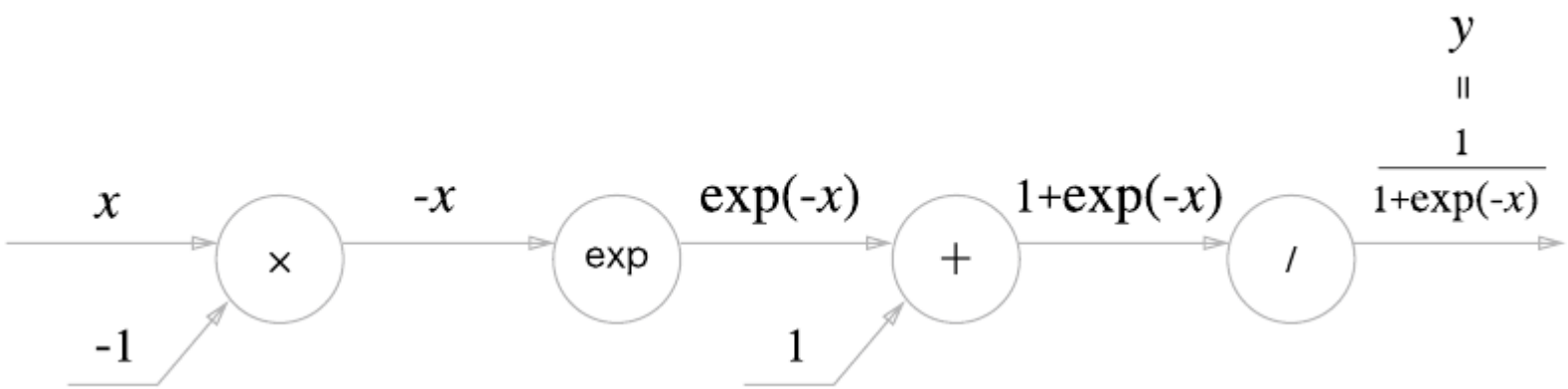
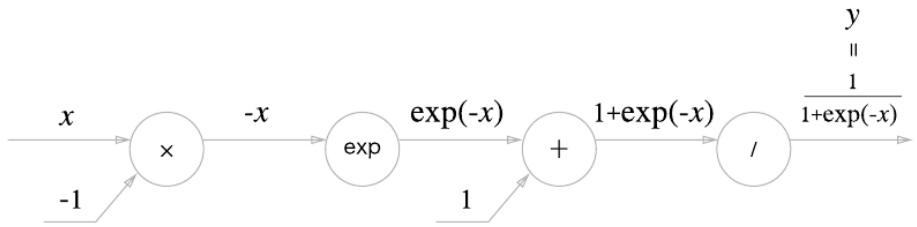


그림 5-19 Sigmoid 계층의 계산 그래프(순전파)



5.5.2 Sigmoid 계층

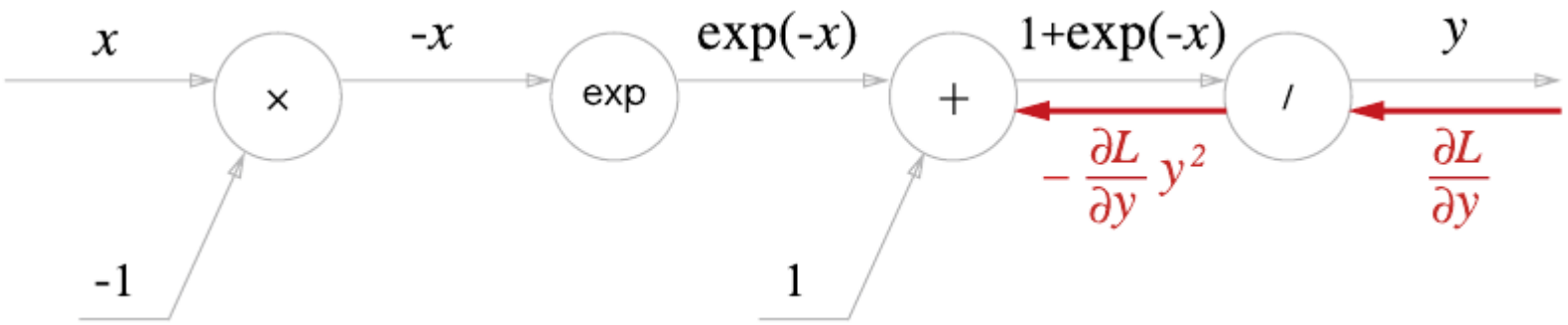
역전파의 흐름을 오른쪽에서 왼쪽으로 한 단계씩 짚어보자.

1단계: / 노드 > $y = \frac{1}{x}$를 미분하면 다음과 같다.

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

$$= -y^2 \qquad \text{[식 5.10]}$$

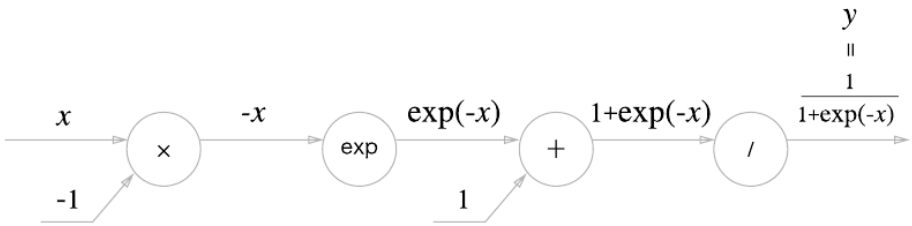
역전파 시, 상류(오른쪽)에서 받은 값에 _____ 을 곱해서 하류(왼쪽) 로 전달



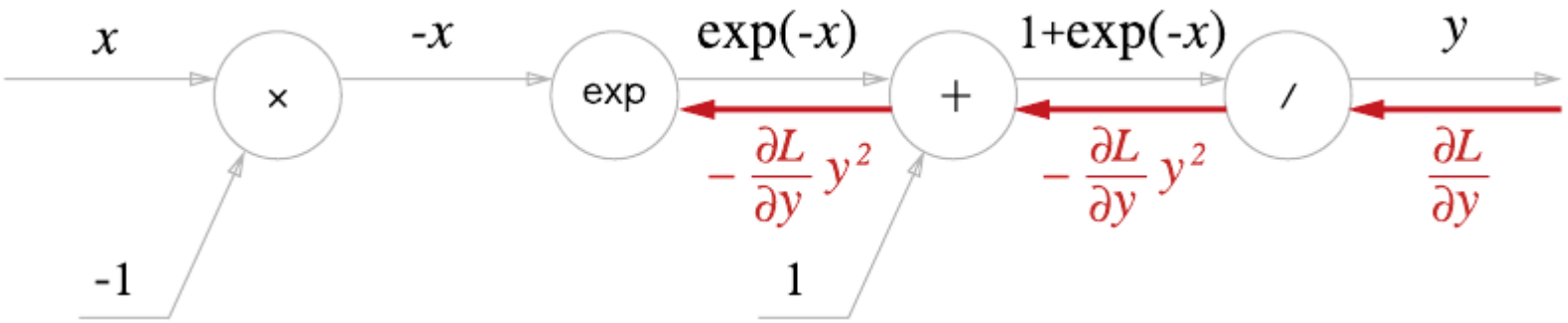
5.5.2 Sigmoid 계층

역전파의 흐름을 오른쪽에서 왼쪽으로 한 단계씩 짚어보자.

그림 5-19 Sigmoid 계층의 계산 그래프(순전파)



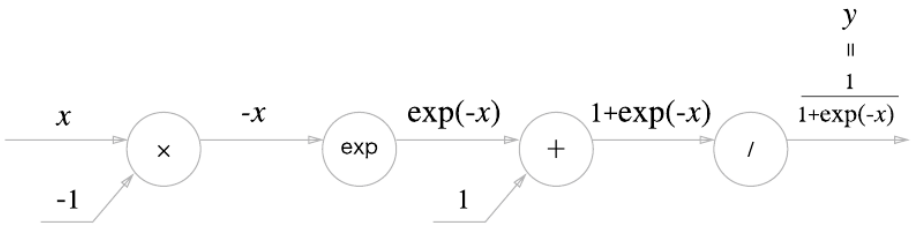
2단계: + 노드 > 상류의 값을 그대로 하류로 전달



5.5.2 Sigmoid 계층

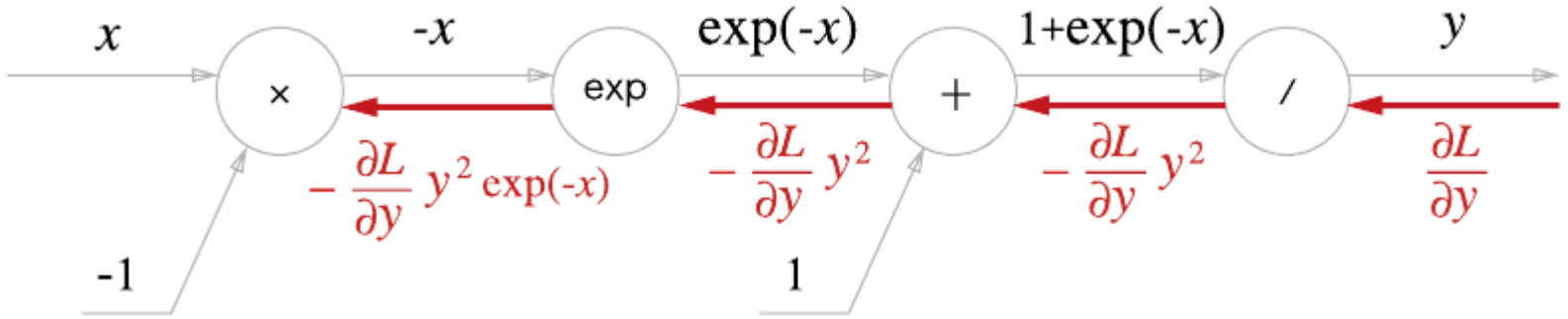
역전파의 흐름을 오른쪽에서 왼쪽으로 한 단계씩 짚어보자.

그림 5-19 Sigmoid 계층의 계산 그래프(순전파)



3단계: exp 노드 > $y = \exp(x)$ 연산을 수행하며, 미분은 다음과 같이 동일하다.

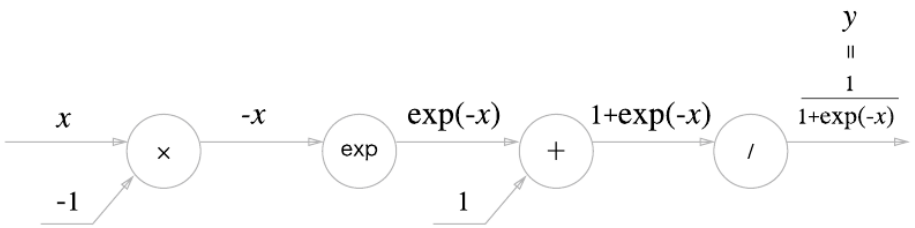
$$\frac{\partial y}{\partial x} = \exp(x) \qquad [식\ 5.11]$$



5.5.2 Sigmoid 계층

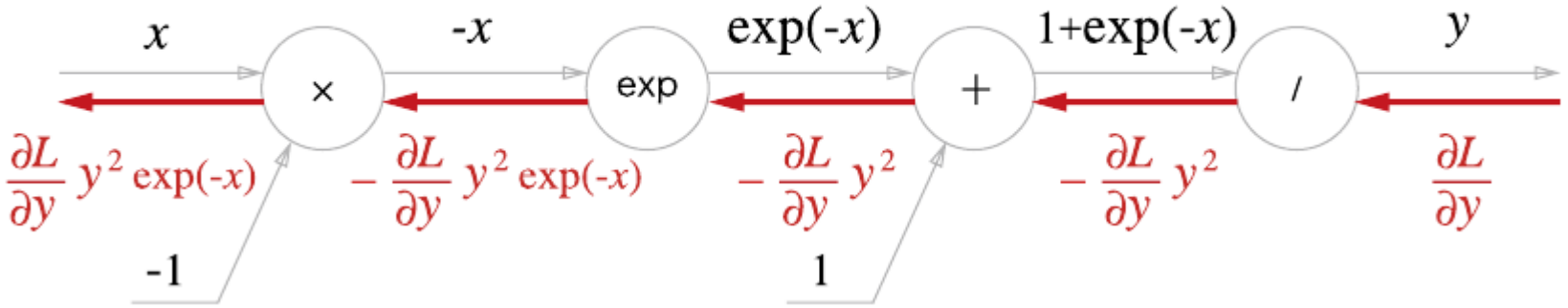
역전파의 흐름을 오른쪽에서 왼쪽으로 한 단계씩 짚어보자.

그림 5-19 Sigmoid 계층의 계산 그래프(순전파)



4단계: 곱하기 x 노드 > 순전파 때 값을 바꾸어 곱하므로, -1을 곱한다

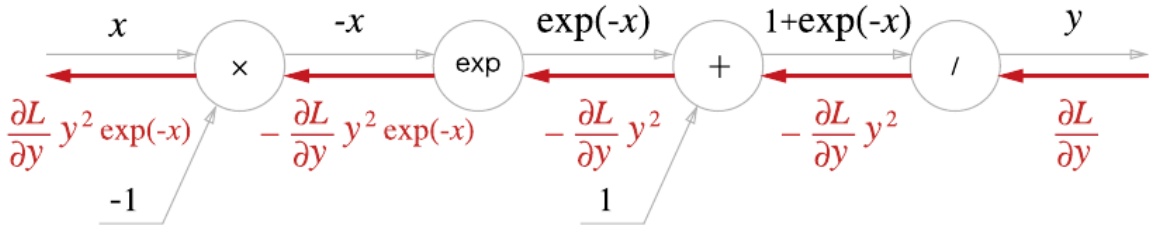
그림 5-20 Sigmoid 계층의 계산 그래프



Sigmoid 계층 그래프 완성!

5.5.2 Sigmoid 계층

그림 5-20 Sigmoid 계층의 계산 그래프

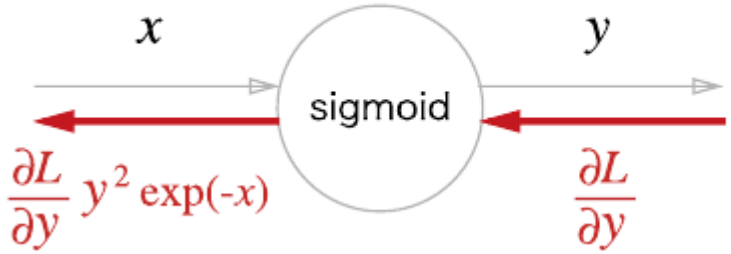


Sigmoid 계층의 계산 그래프:

중간 과정을 생략하고 [그림 5-21]과 같이 입력과 출력만 고려하여 하나의 노드로 구현 가능하다

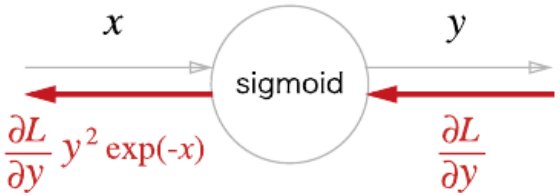
결론

그림 5-21 Sigmoid 계층의 계산 그래프(간소화 버전)



5.5.2 Sigmoid 계층

그림 5-21 Sigmoid 계층의 계산 그래프(간소화 버전)



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

[식 5.12]

(입력 x는 배제하고) 출력 y 값만 가지고 계산 가능하다. > 구현시 출력값 저장 필요

5.5.2 Sigmoid 계층

그림 5-21 Sigmoid 계층의 계산 그래프(간소화 버전)

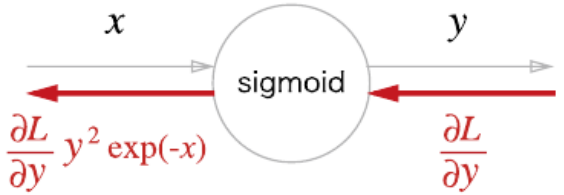
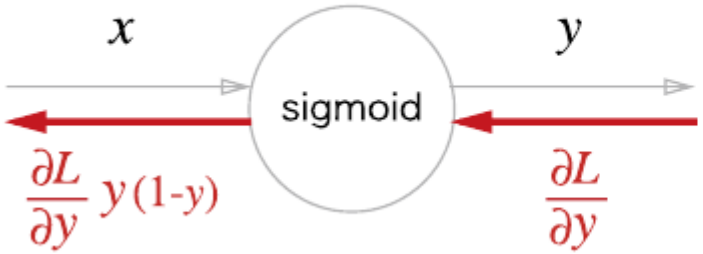
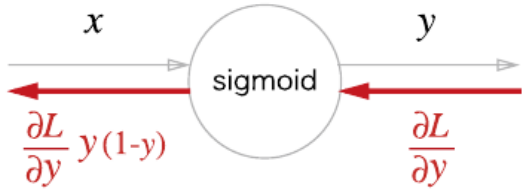


그림 5-22 Sigmoid 계층의 계산 그래프 : 순전파의 출력 y만으로 역전파를 계산할 수 있다.



5.5.2 Sigmoid 계층

그림 5-22 Sigmoid 계층의 계산 그래프 : 순전파의 출력 y만으로 역전파를 계산할 수 있다.




```
class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = 1 / (1 + np.exp(-x))
        self.out = out                  # 출력 y 저장

        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out

        return dx
```

다음 동영상에

5.6 Affine/Softmax 계층 구현하기

5.6.1 Affine 계층

그림 3-18 입력층에서 1층으로의 신호 전달

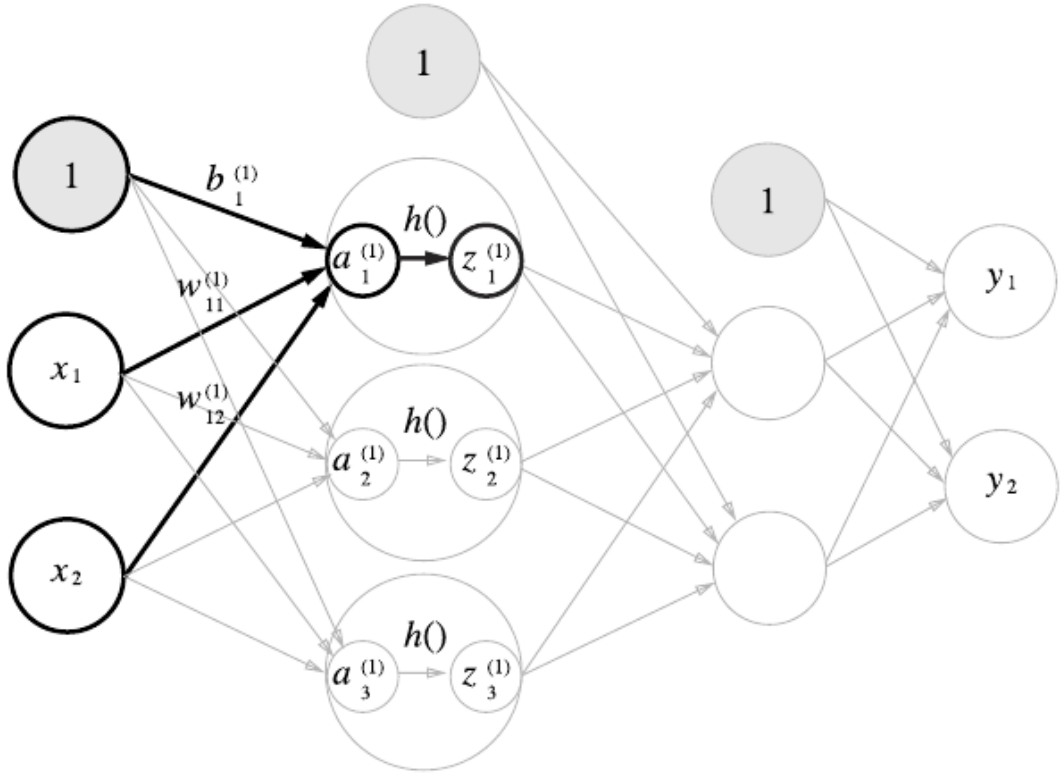
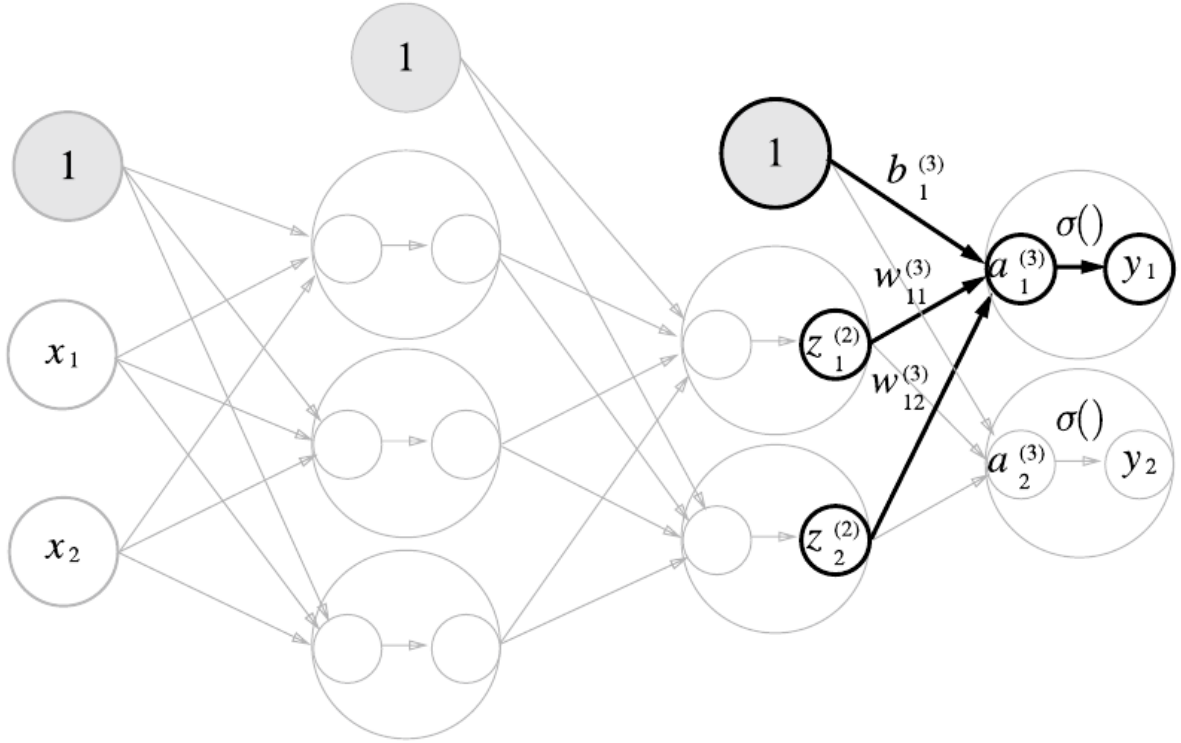


그림 3-20 2층에서 출력층으로의 신호 전달



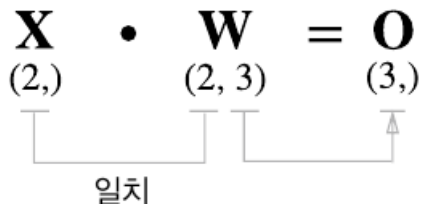
5.6 Affine/Softmax 계층 구현하기

다음 동영상에

5.6.1 Affine 계층

```
>>> X = np.random.rand(2)     # 입력
>>> W = np.random.rand(2,3)   # 가중치
>>> B = np.random.rand(3)     # 편향
>>>
>>> X.shape # (2,)
>>> W.shape # (2, 3)
>>> B.shape # (3,)
>>>
>>> Y = np.dot(X, W) + B
```

그림 5-23 행렬의 곱에서는 대응하는 차원의 원소 수를 일치시킨다.



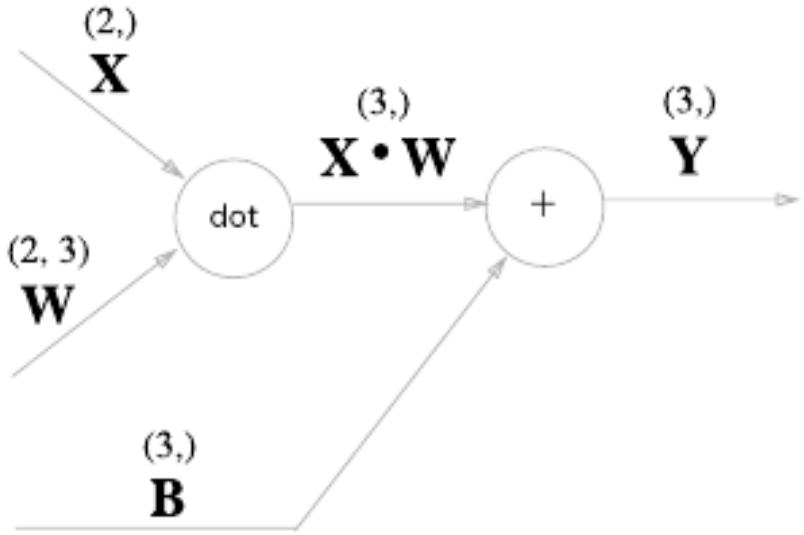
NOTE_ 신경망의 순전파 때 수행하는 행렬의 곱은 기하학에서는 **어파인 변환**affine transformation이라고 합니다. 그래서 이 책에서는 어파인 변환을 수행하는 처리를 'Affine 계층'이라는 이름으로 구현합니다.

5.6.1 Affine 계층: 순전파

```
Y = np.dot(X, W) + B
```

순전파:

그림 5-24 Affine 계층의 계산 그래프 : 변수가 행렬임에 주의. 각 변수의 형상을 변수명 위에 표기했다.



* 위에 행렬의 형상을 표시함

5.6.1 Affine 계층: 역전파

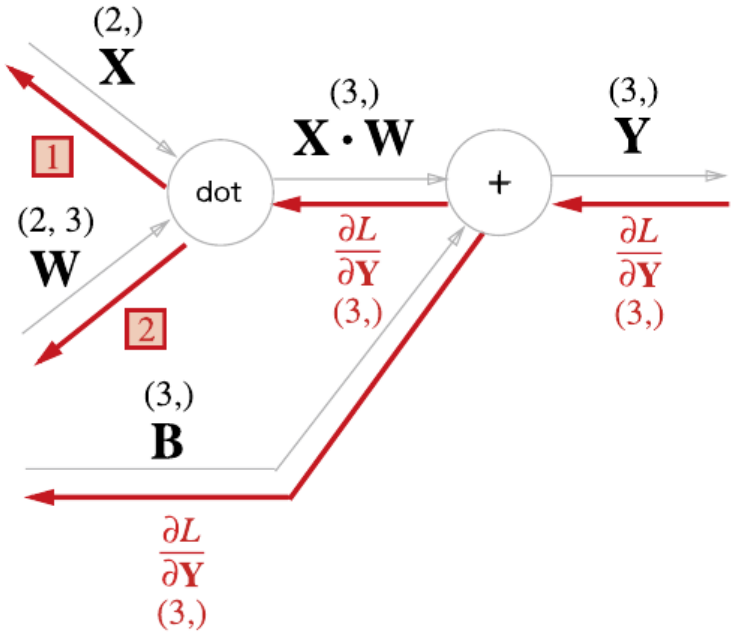
```
Y = np.dot(X, W) + B
```

역전파:

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

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^{\mathrm{T}} \cdot \frac{\partial L}{\partial \mathbf{Y}} \qquad \text{[식 5.13]}$$

그림 5-25 Affine 계층의 역전파 : 변수가 다차원 배열임에 주의. 역전파에서의 변수 형상은 해당 변수명 아래에 표기했다.



<참고>

전치(transpose):

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

$$\mathbf{W}^{\mathrm{T}} = \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{pmatrix}$$

<참고>

임의의 함수 L에 대해 다음 두 형상은 항상 같다.

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

$$\frac{\partial L}{\partial \mathbf{X}} = \left(\frac{\partial L}{\partial x_0}, \frac{\partial L}{\partial x_1}, \cdots, \frac{\partial L}{\partial x_n} \right) \qquad \text{[식 5.14]}$$

그림 5-9 덧셈 노드의 역전파 : 왼쪽이 순전파, 오른쪽이 역전파다. 덧셈 노드의 역전파는 입력 값을 그대로 흘려보낸다.

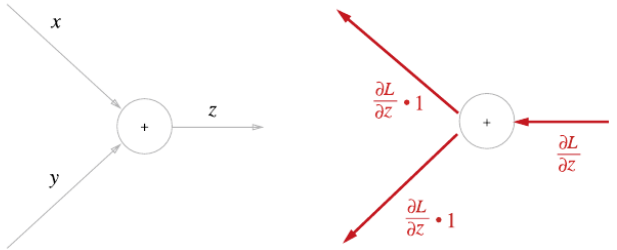
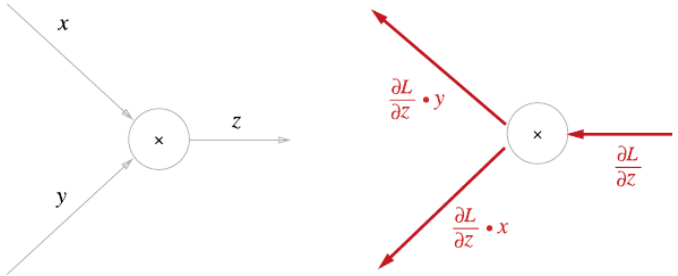


그림 5-12 곱셈 노드의 역전파 : 왼쪽이 순전파, 오른쪽이 역전파다.



5.6.1 Affine 계층: 역전파

그림 5-26 행렬 곱('dot' 노드)의 역전파는 행렬의 대응하는 차원의 원소 수가 일치하도록 곱을 조립하여 구할 수 있다.

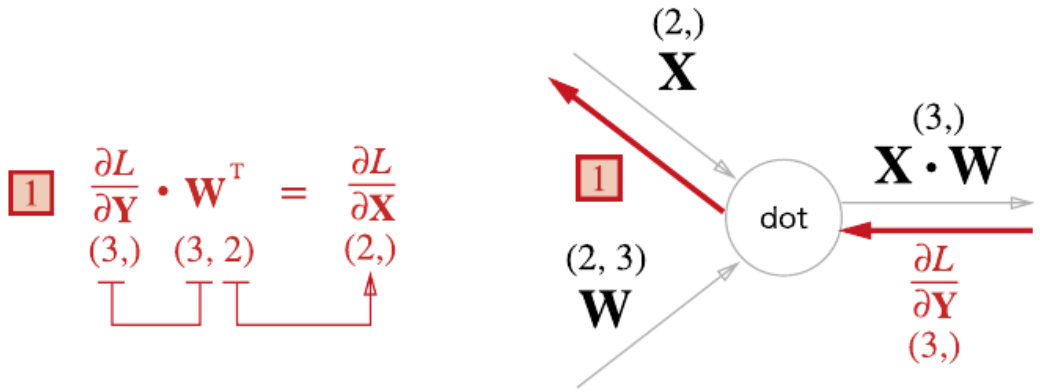
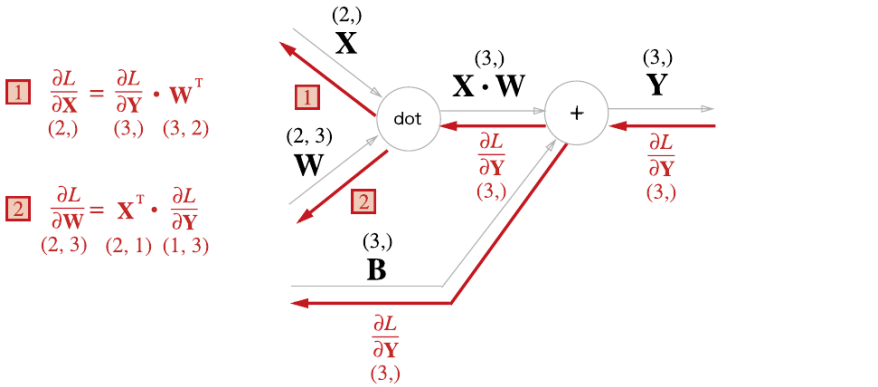
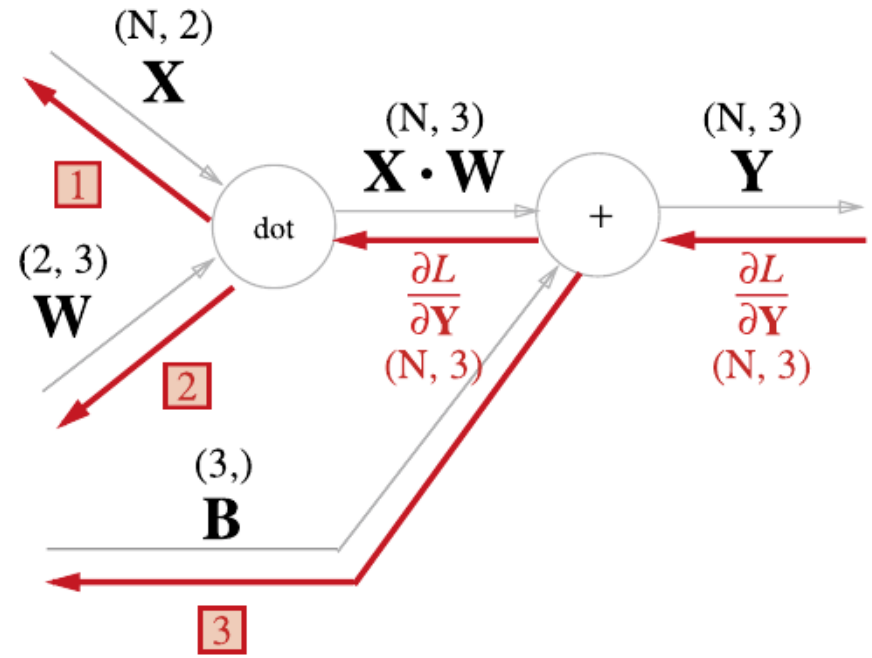


그림 5-25 Affine 계층의 역전파 : 변수가 다차원 배열임에 주의. 역전파에서의 변수 형상은 해당 변수명 아래에 표기했다.



5.6.2 배치용 Affine 계층

그림 5-27 배치용 Affine 계층의 계산 그래프



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

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

2 $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^{\mathrm{T}} \cdot \frac{\partial L}{\partial \mathbf{Y}}$

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

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

(3) (N, 3)

5.6.2 배치용 Affine 계층: 편향 처리

순전파의 편향 덧셈은 각각의 데이터(1번째 데이터, 2번째 데이터, ...)에 더해진다.

```
>>> X_dot_W = np.array([[0, 0, 0], [10, 10, 10]])
>>> B = np.array([1, 2, 3])
>>>
>>> X_dot_W
array([[ 0,  0,  0],
       [ 10, 10, 10]])
>>> X_dot_W + B
```

```
array([[ 1,  2,  3],
       [11, 12, 13]])
```

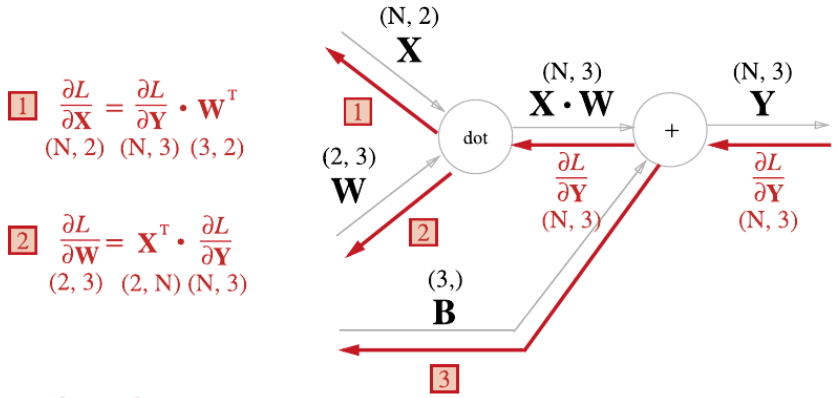
역전파 때는 각 데이터의 역전파 값이 편향의 원소에 모여야 한다.

```
>>> dY = np.array([[1, 2, 3], [4, 5, 6]])
>>> dY
array([[1, 2, 3],
       [4, 5, 6]])
>>>
>>> dB = np.sum(dY, axis=0)
>>> dB
array([5, 7, 9])
```

5.6.2 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(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
```

그림 5-27 배치용 Affine 계층의 계산 그래프



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

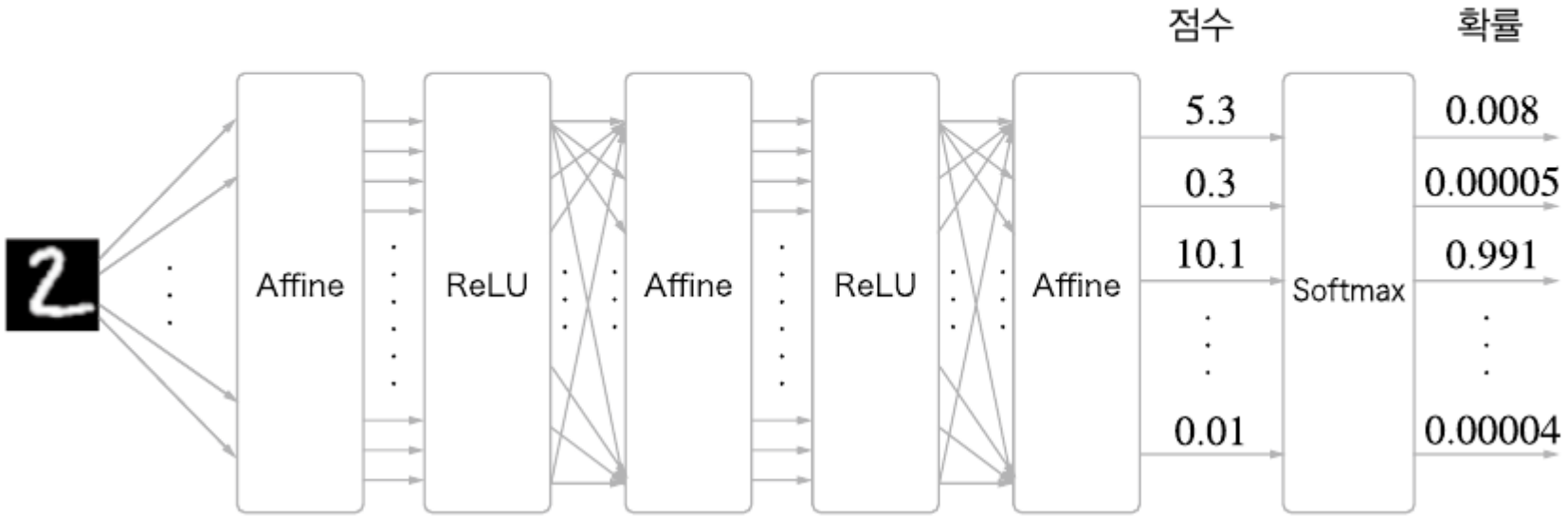
2 $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^{\mathrm{T}} \cdot \frac{\partial L}{\partial \mathbf{Y}}$ (2, 3) (2, N) (N, 3)

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

5.6.3 Softmax-with-Loss 계층

MNIST 데이터를 위한 3층 신경망 예시

그림 5-28 입력 이미지가 Affine 계층과 ReLU 계층을 통과하며 변환되고, 마지막 Softmax 계층에 의해서 10개의 입력이 정규화된다. 이 그림에서는 숫자 '0'의 점수는 5.3이며, 이것이 Softmax 계층에 의해서 0.008(0.8%)로 변환된다. 또, '2'의 점수는 10.1에서 0.991(99.1%)로 변환된다.



소프트맥스 함수는 입력 값을 정규화하여 출력

NOTE_ 신경망에서 수행하는 작업은 **학습**과 **추론** 두 가지입니다. 추론할 때는 일반적으로 Softmax 계층을 사용하지 않습니다. 예컨대 [그림 5-28]의 신경망은 추론할 때는 마지막 Affine 계층의 출력을 인식 결과로 이용합니다. 또한, 신경망에서 정규화하지 않는 출력 결과([그림 5-28]에서는 Softmax 앞의 Affine 계층의 출력)를 **점수**score라 합니다. 즉, 신경망 추론에서 답을 하나만 내는 경우에는 가장 높은 점수만 알면 되니 Softmax 계층은 필요 없다는 것이죠. 반면, 신경망을 학습할 때는 Softmax 계층이 필요합니다.

5.6.3 Softmax-with-Loss 계층

Softmax-with-Loss 계층의 계산 그래프(부록 참고)

그림 5-29 Softmax-with-Loss 계층의 계산 그래프

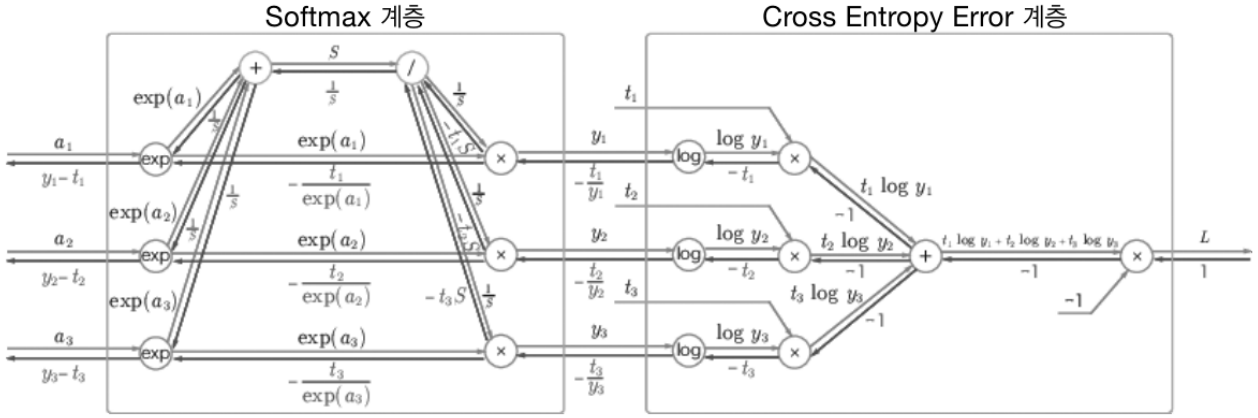
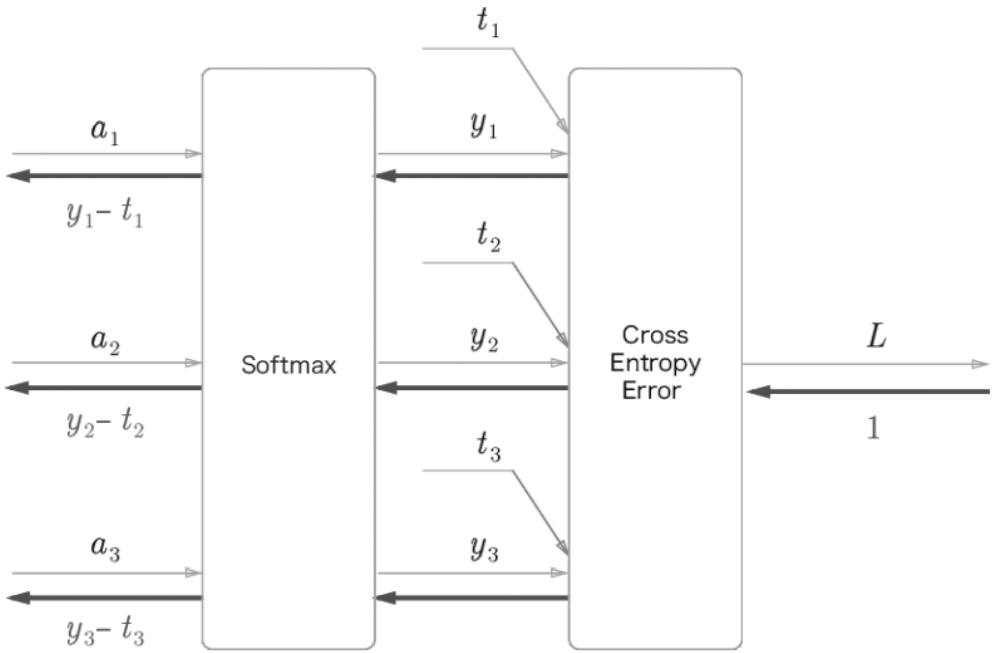


그림 5-30 '간소화한' Softmax-with-Loss 계층의 계산 그래프

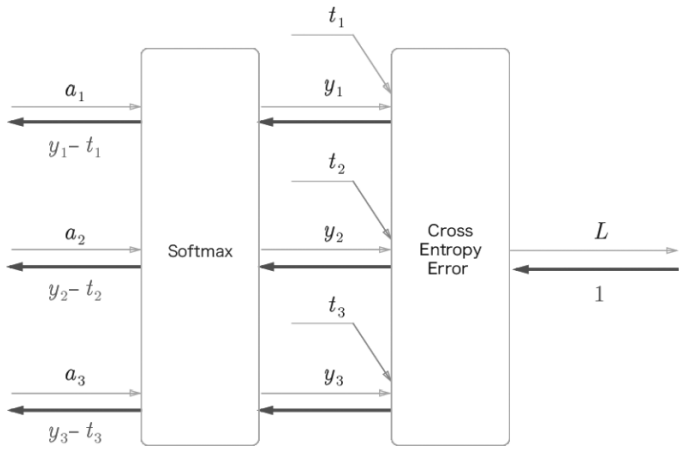


NOTE_ '소프트맥스 함수'의 손실 함수로 '교차 엔트로피 오차'를 사용하니 역전파가 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$로 말끔히 떨어집니다. 사실 이런 말끔함은 우연이 아니라 교차 엔트로피 오차라는 함수가 그렇게 설계되었기 때문입니다. 또, 회귀의 출력층에서 사용하는 '항등 함수'의 손실 함수로 '오차제곱합'을 이용("3.5 출력층 설계하기" 참고)하는 이유도 이와 같습니다. 즉, '항등 함수'의 손실 함수로 '오차제곱합'을 사용하면 역전파의 결과가 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$로 말끔히 떨어집니다.

5.6.3 Softmax-with-Loss 계층

```
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None # 손실
        self.y = None    # softmax의 출력
        self.t = None    # 정답 레이블(원-핫 벡터)

    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]
        dx = (self.y - self.t) / batch_size

        return dx
```

그림 5-30 '간소화한' Softmax-with-Loss 계층의 계산 그래프



5.7.1 신경망 학습의 전체 그림

다음 시간에 ~

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 매개변수에 대한 기울기를 구한다. (기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시한다)

3단계 – 매개변수 갱신

가중치 매개변수를 기울기 방향으로 아주 조금 갱신한다.

4단계 – 반복

1~3단계를 반복한다.