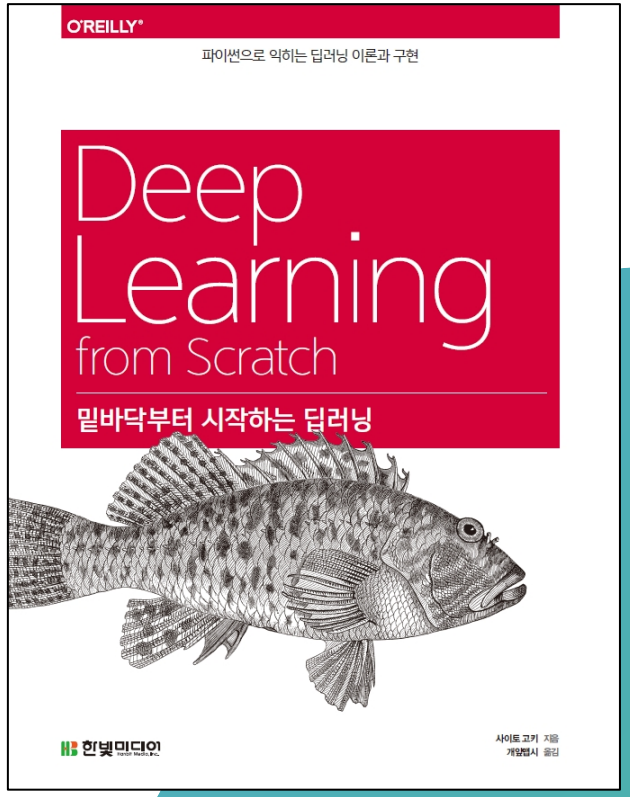
CHAPTER 5 오차역전파법

밑바닥부터 시작하는 딥러닝



홍익대학교 컴퓨터공학과
박 준

이 책의 학습 목표

- CHAPTER 1 파이썬에 대해 간략하게 살펴보고 사용법 익히기
- CHAPTER 2 퍼셉트론에 대해 알아보고 퍼셉트론을 써서 간단한 문제를 풀어보기
- CHAPTER 3 신경망의 개요, 입력 데이터가 무엇인지 신경망이 식별하는 처리 과정 알아보기
- CHAPTER 4 손실 함수의 값을 가급적 작게 만드는 경사법에 대해 알아보기
- CHAPTER 5 가중치 매개변수의 기울기를 효율적으로 계산하는 오차역전파법 배우기
- CHAPTER 6 신경망(딥러닝) 학습의 효율과 정확도를 높이기
- CHAPTER 7 CNN의 메커니즘을 자세히 설명하고 파이썬으로 구현하기
- CHAPTER 8 딥러닝의 특징과 과제, 가능성, 오늘날의 첨단 딥러닝에 대해 알아보기

Contents

Contents

CHAPTER 5 오차역전파법

가중치 매개변수의 기울기를 효율적으로 계산하는 오차역전파법 배우기

SECTION 05 오차역전파법

5.1.1 계산 그래프로 풀다

1. 100원짜리 사과 2개 구매, 10% 소비세

2. 100원짜리 사과 2개, 150원짜리 귤 3개 구매, 10% 소비세

지불 금액을 Computational Graph로 표현하기

그림 5-1 계산 그래프로 풀어본 문제 1 의 답

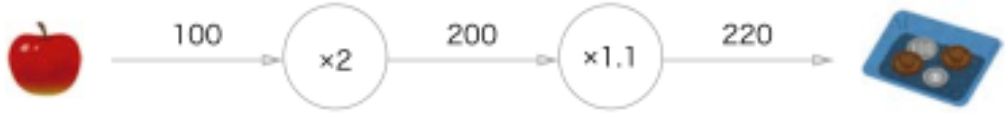


그림 5-2 계산 그래프로 풀어본 문제 1 의 답 : '사과의 개수'와 '소비세'를 변수로 취급해 원 밖에 표기

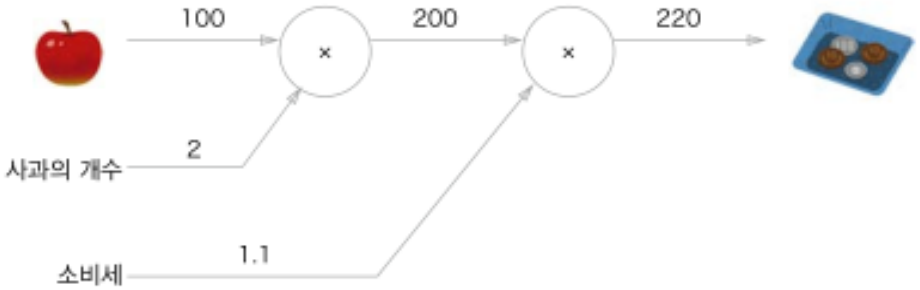
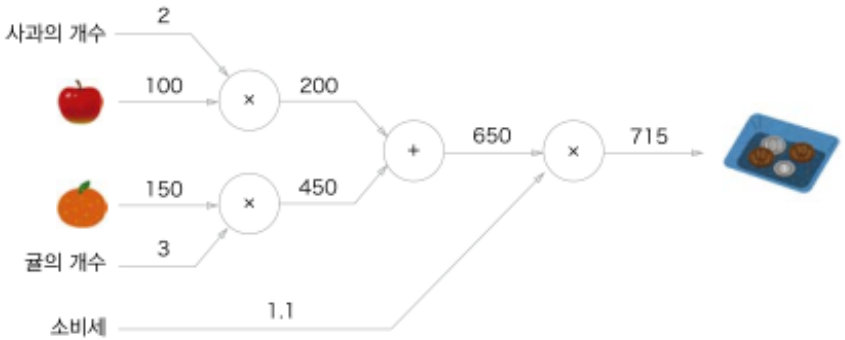


그림 5-3 계산 그래프로 풀어본 문제 2 의 답



지금까지 살펴본 것처럼 계산 그래프를 이용한 문제풀이는 다음 흐름으로 진행한다.

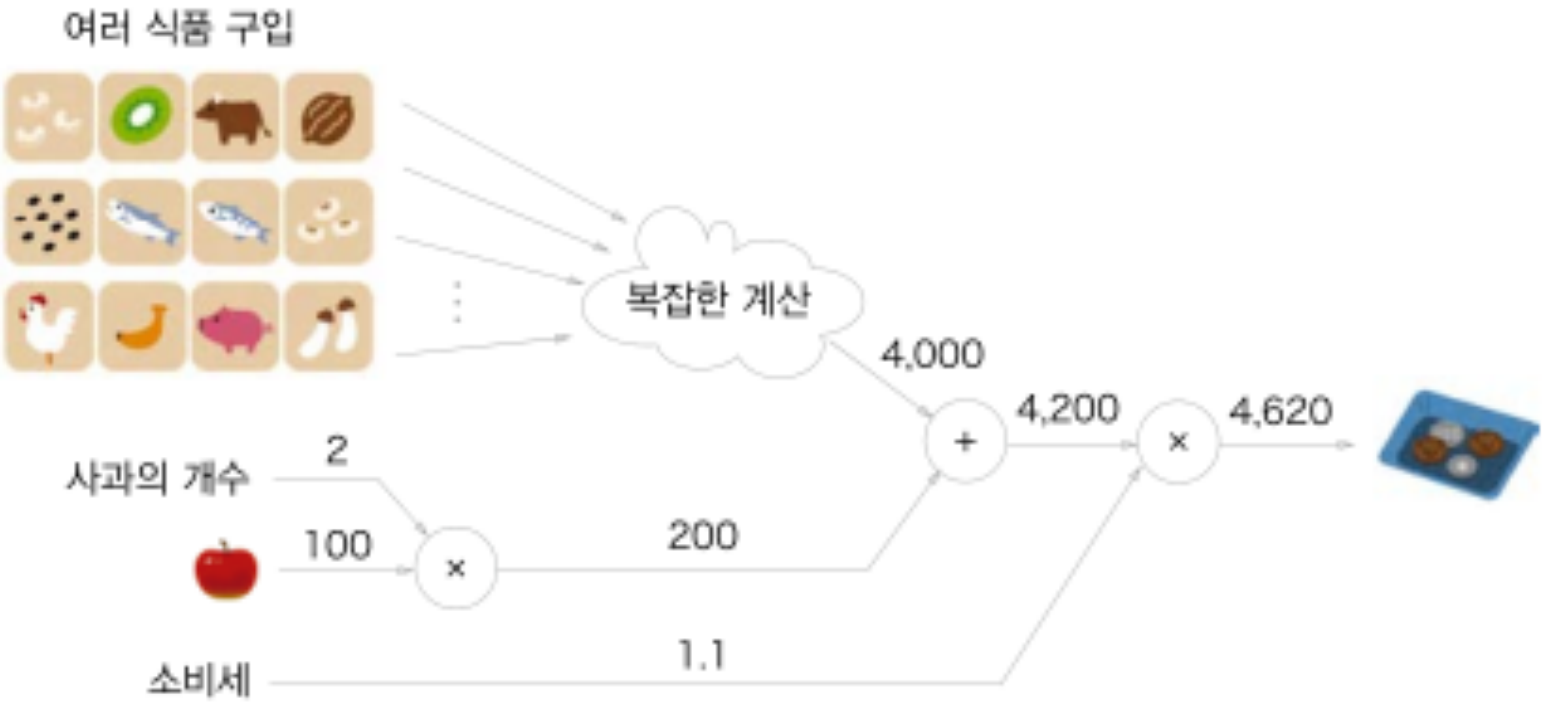
1. 계산 그래프를 구성한다.

2. 그래프에서 계산을 왼쪽에서 오른쪽으로

여기서 2 번째 '계산을 왼쪽에서 오른쪽으로 진행'하는 단계를 순전파 forward propagation 라고 한다.
순전파는 계산 그래프의 출발점부터 종착점으로의 전파이다.
순전파라는 이름이 있다면 반대 방향(그림에서 말하면 오른쪽에서 왼쪽)의 전파도 가능한데 그것을 역전파 backwardpropagation 라고 한다

5.1.2 국소적 계산

계산 그래프의 특징은 '국소적 계산'을 전파함으로써 최종 결과를 얻는다는 점에 있다다.
국소적이란 '자신과 직접 관계된 작은 범위'라는 뜻이다.



5.1.3 Why Computational Graph?

1. 전체가 복잡해도 각 노드에서는 단순한 계산에 집중

2. 중간 결과 보관

3. 역전파(back-propagation)을 통해 미분을 효율적으로 계산

5.2.1 계산 그래프의 역전파

그림 5-6 계산 그래프의 역전파 : 순방향과는 반대 방향으로 국소적 미분을 곱한다.

※ 교재의 상세 과정을 참고하여 실습을 진행합니다.

5.2.2 연쇄법칙이란?

계산해 보기

연쇄법칙을 설명하려면 우선 합성 함수 이야기부터 시작해야 한다. 합성 함수란 여러 함수로구성된 함수다

$$z = t^2$$
$$t = x + y \qquad \text{[식 5.1]}$$

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

을 예로 설명하면, $\frac{\partial z}{\partial x}$ (x에 대한 z의 미분)은 $\frac{\partial z}{\partial t}$ (t에 대한 z의 미분)과 $\frac{\partial t}{\partial x}$ (x에 대한 t의 미분)의 곱으로 나타낼 수 있다는 것이죠. 수식으로는 [식 5.2]처럼 쓸 수 있습니다.

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} \qquad \text{[식 5.2]}$$

5.2.3 연쇄법칙과 계산 그래프

2 제곱 계산을 ‘ ** 2 ’ 노드로 나타내면 [그림 5 - 7]처럼 그릴 수 있다.

역전파가 하는 일은 연쇄법칙의 원리와 동일

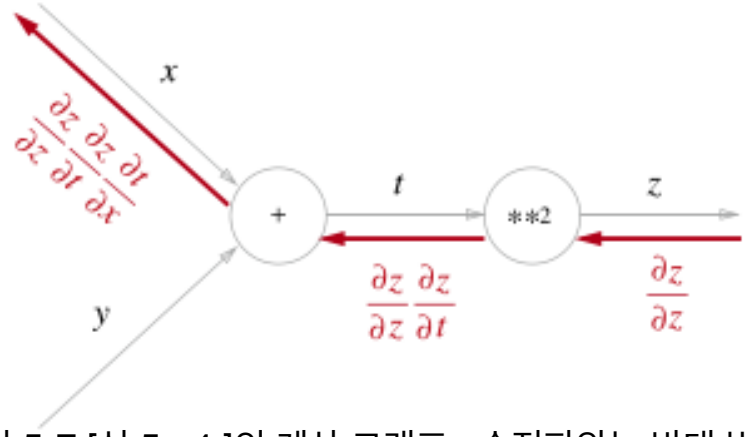


그림 5-7 [식 5 . 4]의 계산 그래프 : 순전파와는 반대 방향으로 국소적 미분을 곱하여 전달한다.

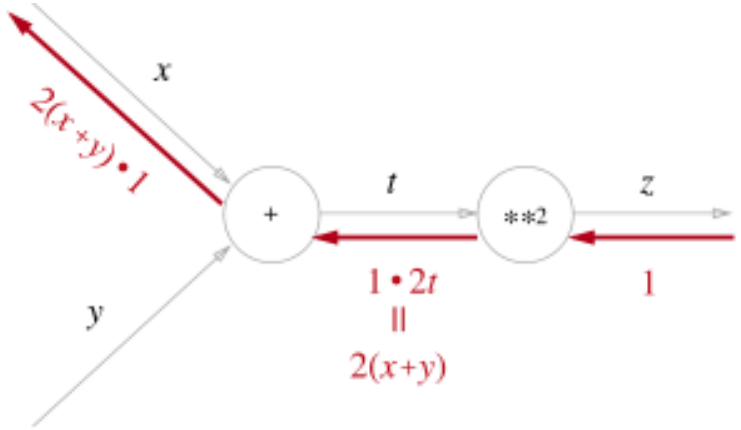


그림 5-8 계산 그래프의 역전파 결과에 따르면 는 2 (x + y)가 된다.

5.3.1 덧셈 노드의 역전파 $z = x + y$

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

$$\frac{\partial z}{\partial y} = 1 \quad \text{[식 5.5]}$$

이를 계산 그래프로는 [그림 5-9]처럼 그릴 수 있다.

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

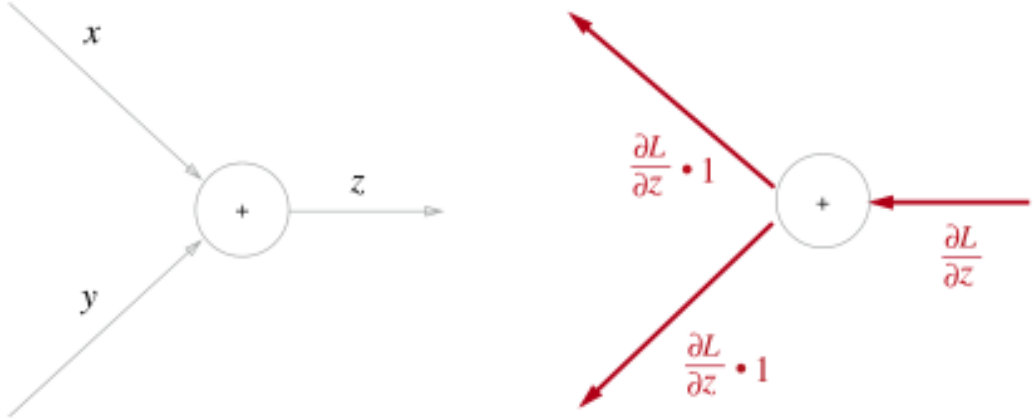
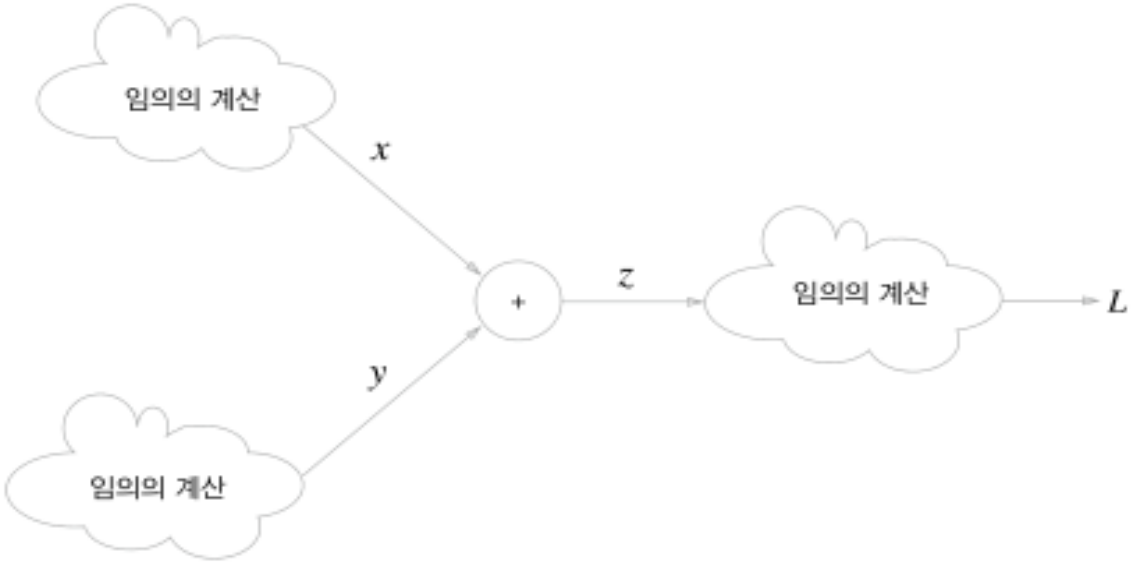


그림 5-10 최종 출력으로 가는 계산의 중간에 덧셈 노드가 존재한다. 역전파에서는 국소적 미분이 가장 오른쪽의 출력에서 시작하여 노드를 타고 역방향(왼쪽)으로 전파된다



5.3.2 곱셈 노드의 역전파

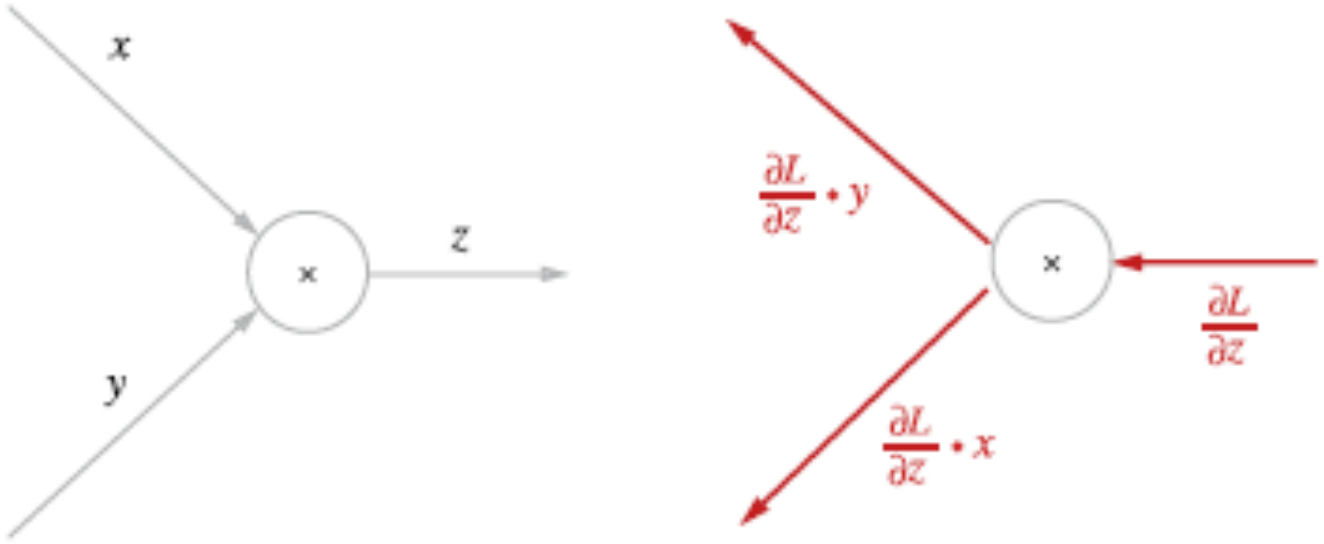
$z = x * y$

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

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

[식 5.6]

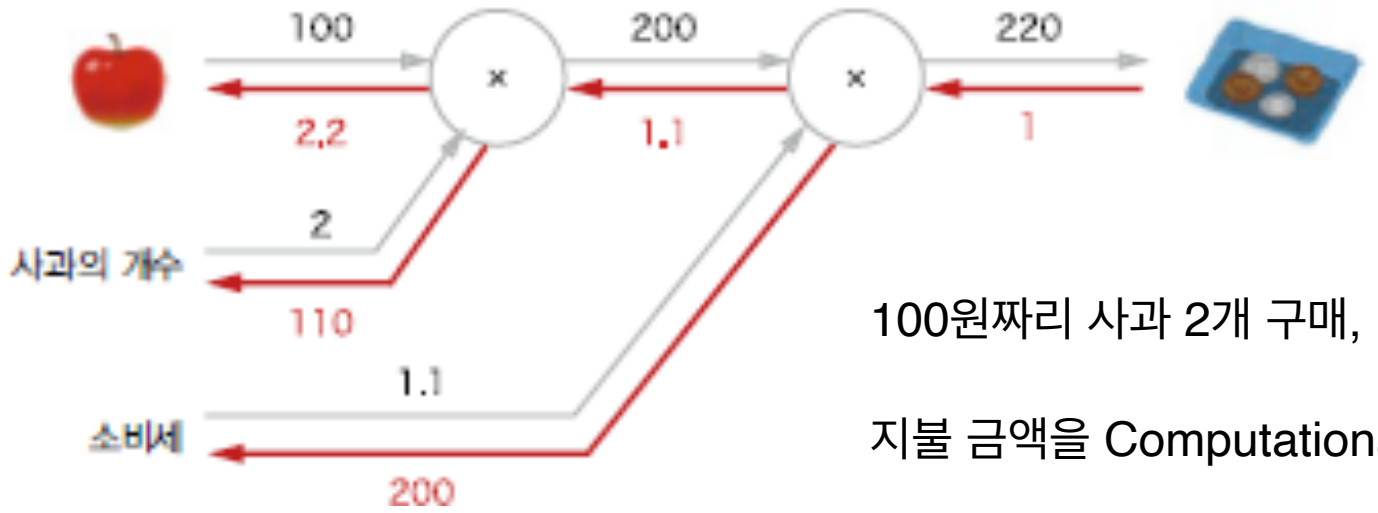
[식 5.6]에서 계산 그래프는 다음과 같이 그릴 수 있다.



곱셈 노드에서는 역전차 시에
순전파의 입력 신호가 필요
—> 저장할 필요가 있음

5.3.3 사과 쇼핑의 예

그림 5-16 사과 2개 구입



100원짜리 사과 2개 구매, 10% 소비세

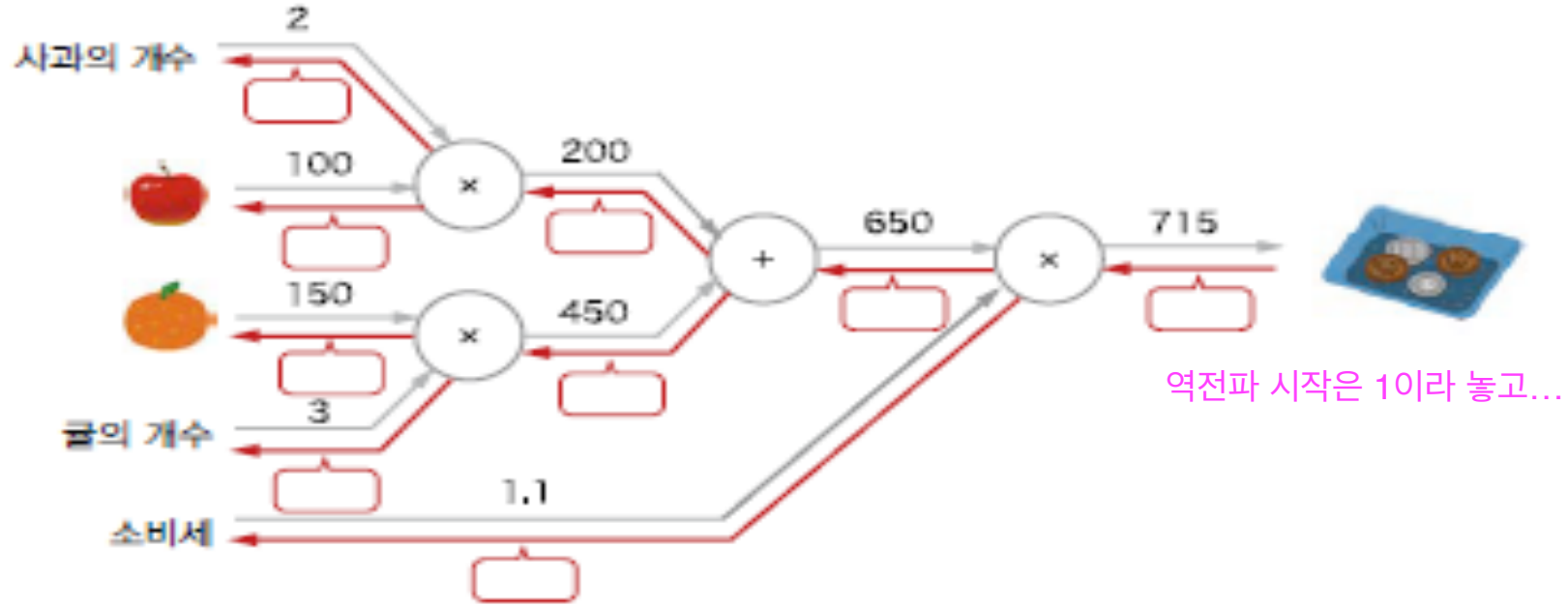
지불 금액을 Computational Graph로 표현하기

지금까지 설명한 바와 같이 곱셈 노드의 역전파에서는 입력 신호를 서로 바꿔서 하류로 흘린다.

5.3.3 사과 쇼핑의 예

손으로 계산해 보기

그림 5-15 사과와 귤 쇼핑의 역전파 예 : 빈 상자 안에 적절한 숫자를 넣어 역전파를 완성하시오.



역전파 시작은 1이라 놓고…

지금까지 설명한 바와 같이 곱셈 노드의 역전파에서는 입력 신호를 서로 바꿔서 하류로 흘린다.

5.3.3 사과 쇼핑의 예

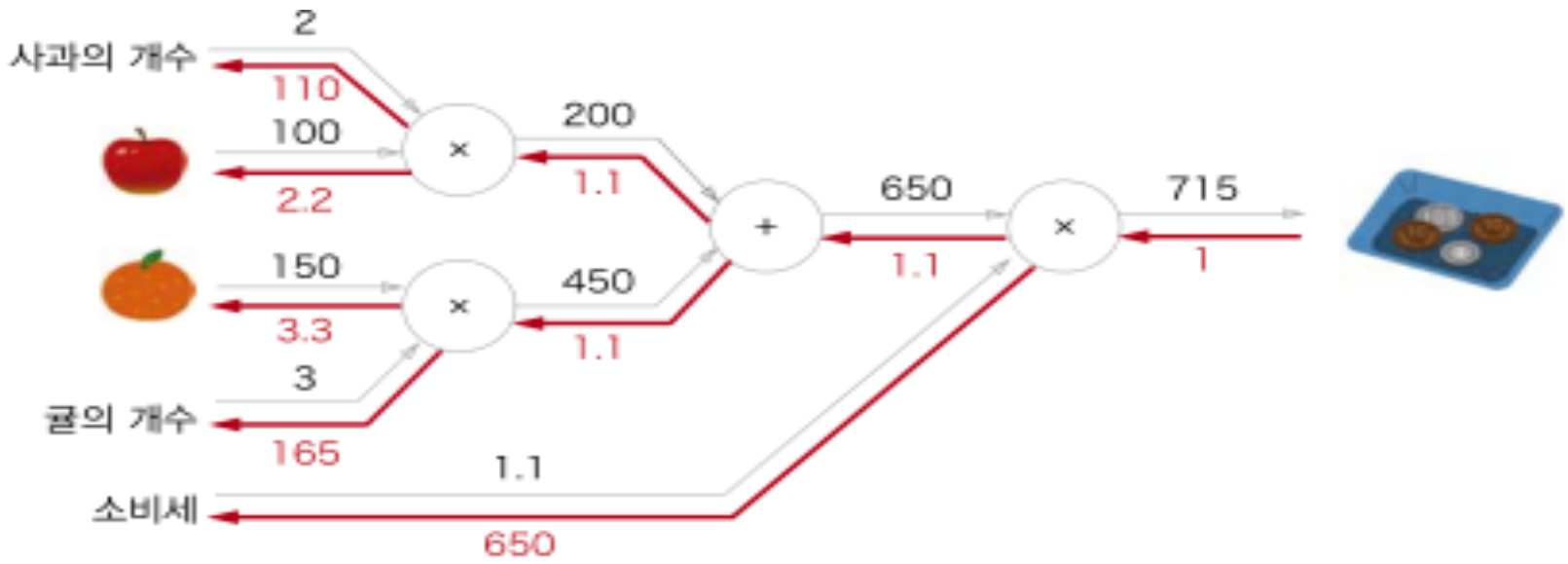


그림 5-17 사과 2 개와 귤 3 개 구입

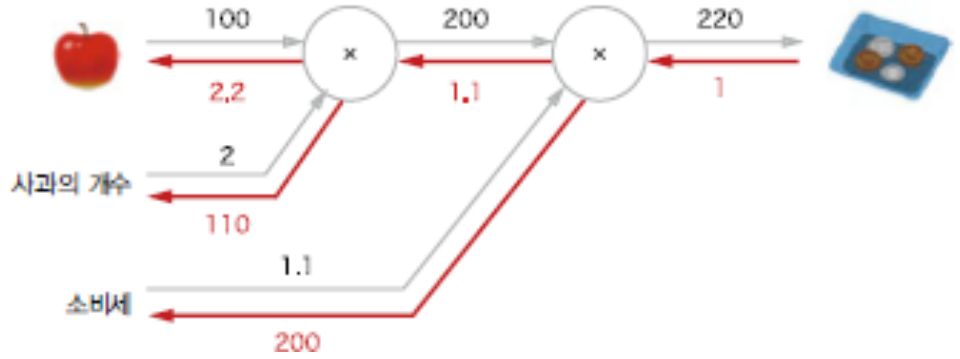
SECTION 05 오차역전파법

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
```

__init__ ( )에서는 인스턴스 변수인 x와 y를 초기화합니다. 이 두 변수는 순전파 시의 입력 값을 유지하기 위해서 사용한다.
Forward( )에서는 x와 y를 인수로 받고 두 값을 곱해서 반환한다.
반면 backward( )에서는 상류에서 넘어온 미분(dout)에 순전파 때의 값을 '서로 바꿔' 곱한 후 하류로 흘린다.

그림 5-16 사과 2개 구입



[그림 5-16]의 순전파를 다음과 같이 구현할 수 있다

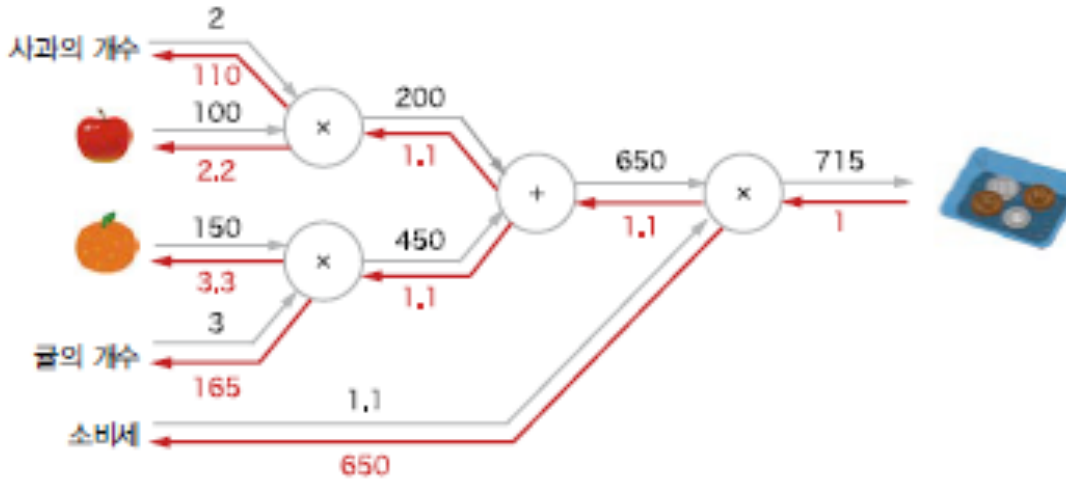
```
apple = 100
apple_num = 2
tax = 1.1

# 계층들
mul_apple_layer = MulLayer()
mul_tax_layer = MulLayer()

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

print(price) # 220
```

SECTION 05 오차역전파법

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 5-17 사과 2개와 귤 3개 구입




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

# 계층들
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()

# 순전파
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)

# 역전파
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)

print(price) # 715
print(dapple_num, dapple, dorange, dorange_num, dtax) # 110 2.2 3.3 165 650
```

SECTION 05 오차역전파법

5.5.1 ReLU 계층

활성화 함수로 사용되는 ReLU의 수식은 다음과같다.

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

[식 5.7]에서 x에 대한 y의 미분은 [식 5.8]처럼 구한다

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

그림 5-18 ReLU 계층의 계산 그래프


```
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
```

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

5.5.2 Sigmoid 계층

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

[식 5.9]를 계산 그래프로 그리면 [그림 5-19]처럼 된다

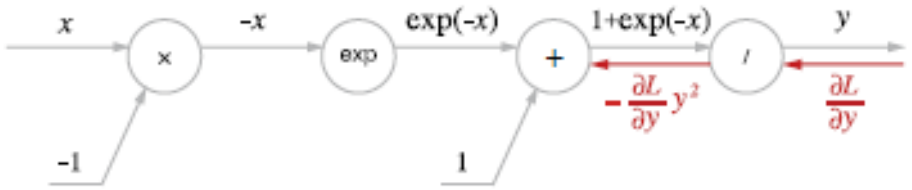
그림 5-19 Sigmoid 계층의 계산 그래프(순전파)

5.5.2 Sigmoid 계층

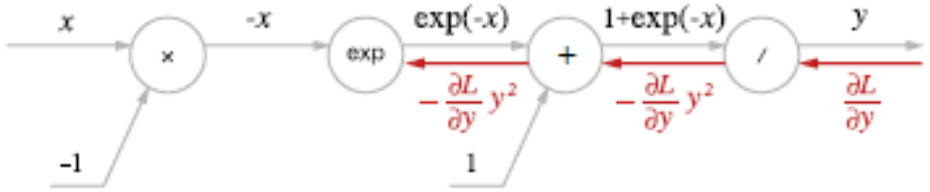
1 단계

'/' 노드, 즉 $y=\frac{1}{x}$ 을 미분하면 다음 식이 됩니다.

$$\frac{\partial y}{\partial x}=\frac{1}{x^2}=-y^2 \qquad \text{[식 5.10]}$$

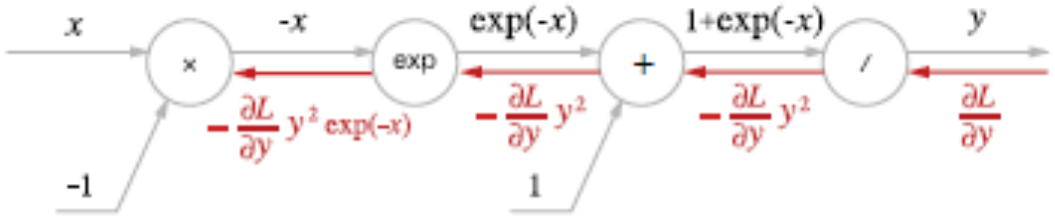


2 단계



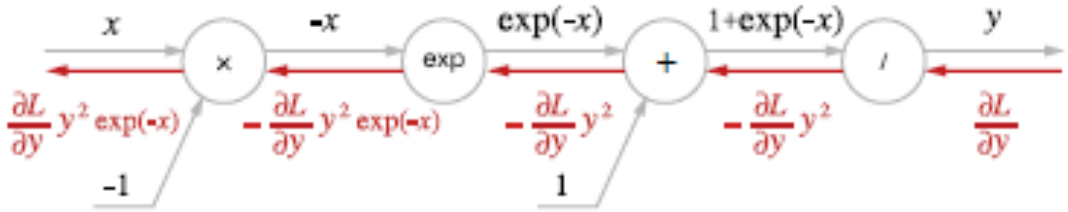
3 단계

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



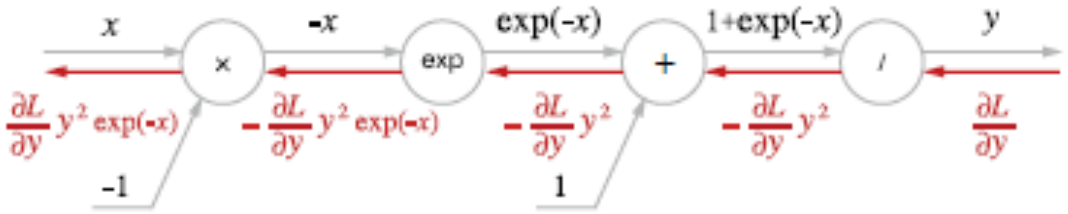
4 단계

그림 5-20 Sigmoid 계층의 계산 그래프



5.5.2 Sigmoid 계층

그림 5-20 Sigmoid 계층의 계산 그래프



식을 간단하게 정리하기

$$\frac{\delta L}{\delta y} y^2 exp(-x)$$ = y * (y * exp(-x))

= y * (exp(-x) * 1/(1+exp(-x))

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

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

        return out

    def backward(self, dout):
        dx = dout * self.out * (1-self.out)

        return dx
```

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.

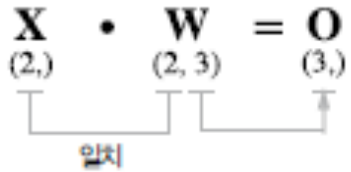
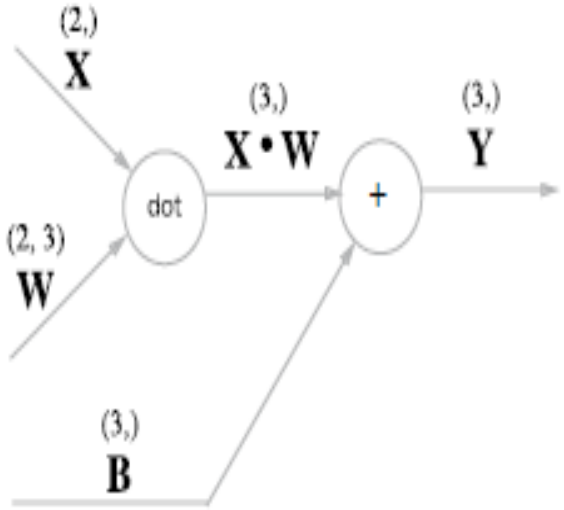


그림 5-24 Affine 계층의 계산 그래프 : 변수가 행렬임에 주의. 각 변수의 형상을 변수명 위에 표기했다.



5.6.1 Affine 계층

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

$$\underset{(2,)}{\mathbf{X}} \cdot \underset{(2,3)}{\mathbf{W}} = \underset{(3,)}{\mathbf{O}}$$

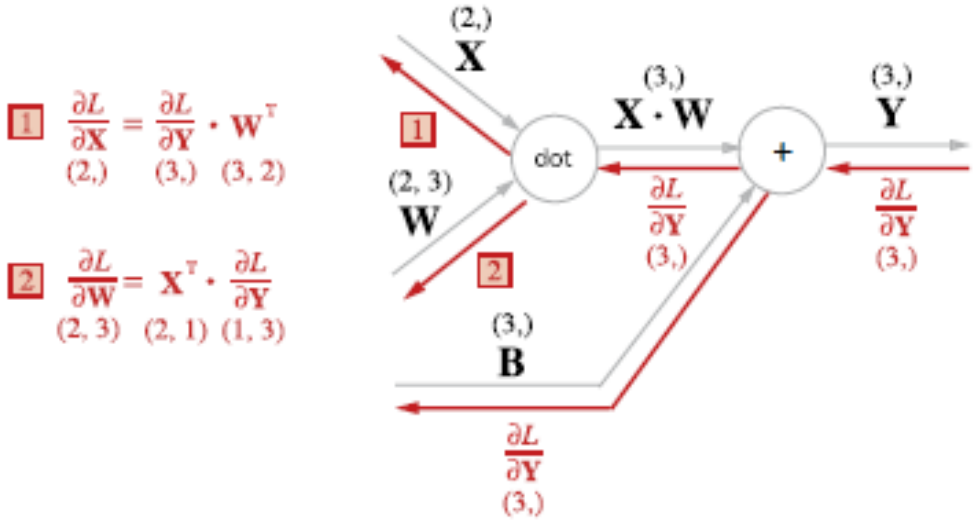
일치

[식 5.13]에서 $\mathbf{W}^T$의 T는 전치행렬을 뜻합니다. 전치행렬은 $\mathbf{W}$의 (i, j) 위치의 원소를 (j, i)위치로 바꾼 것을 말한다.
수식으로는 다음과 같이 쓸 수 있다.

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

$$\mathbf{W}^T = \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{pmatrix} \quad [식\ 5.14]$$

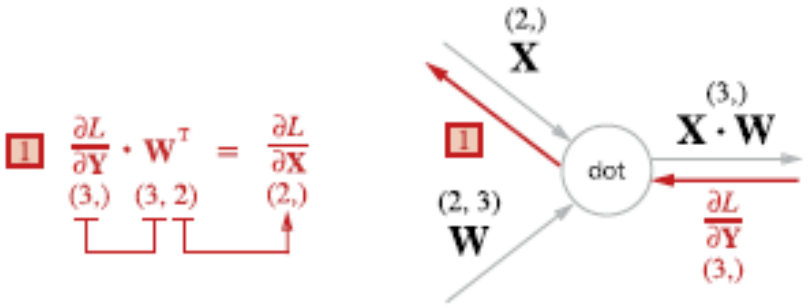
그림 5-25 Affine 계층의 역전파 : 변수가 다차원 배열임에 주의. 역전파에서의 변수 형상은 해당 변수명 아래에 표기했다.



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

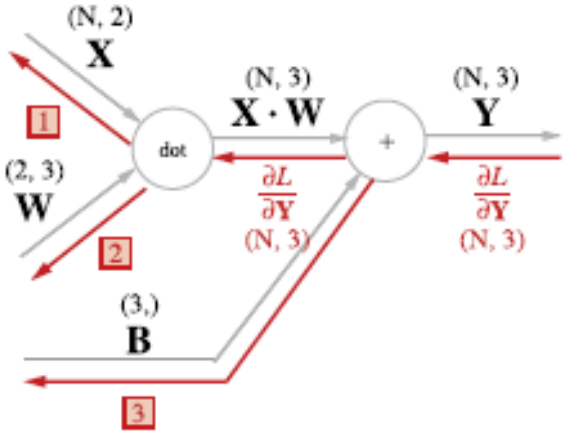
$$\frac{\partial L}{\partial \mathbf{X}} = \left(\frac{\partial L}{\partial x_0}, \frac{\partial L}{\partial x_1}, \cdots, \frac{\partial L}{\partial x_n}\right) \quad [식\ 5.15]$$

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



5.6.2 배치용 Affine 계층

그림 5-27 배치용 Affine 계층의 계산 그래프



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

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

```
>>> 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]])
```

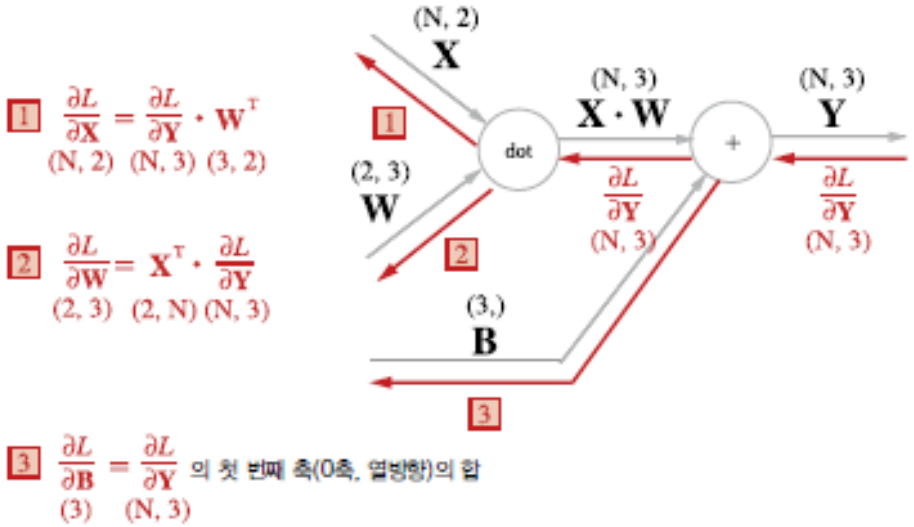
순전파의 편향 덧셈은 각각의 데이터(1번째 데이터, 2번째 데이터, ...)에 더해진다.
역전파 때는 각 데이터의 역전파 값이 편향의 원소에 모여야 한다.

코드

```
>>> 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

그림 5-27 배치용 Affine 계층의 계산 그래프

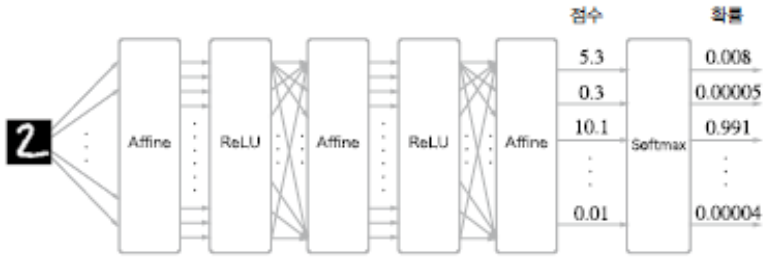


순전파의 편향 덧셈은 각각의 데이터(1번째 데이터, 2번째 데이터, ...)에 더해진다.
역전파 때는 각 데이터의 역전파 값이 편향의 원소에 모여야 한다.

```
class Affine:
    def __init__(self, W, b):
        self.W, self.b = W, b
        self.x, self.dW, self.db = \
            None, None, 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
```

5.6.3 Softmax-with-Loss 계층

소프트맥스 함수는 입력 값을 정규화하여 출력



softmax-with-Loss 계층의 계산 그래프: Appendix A 참조

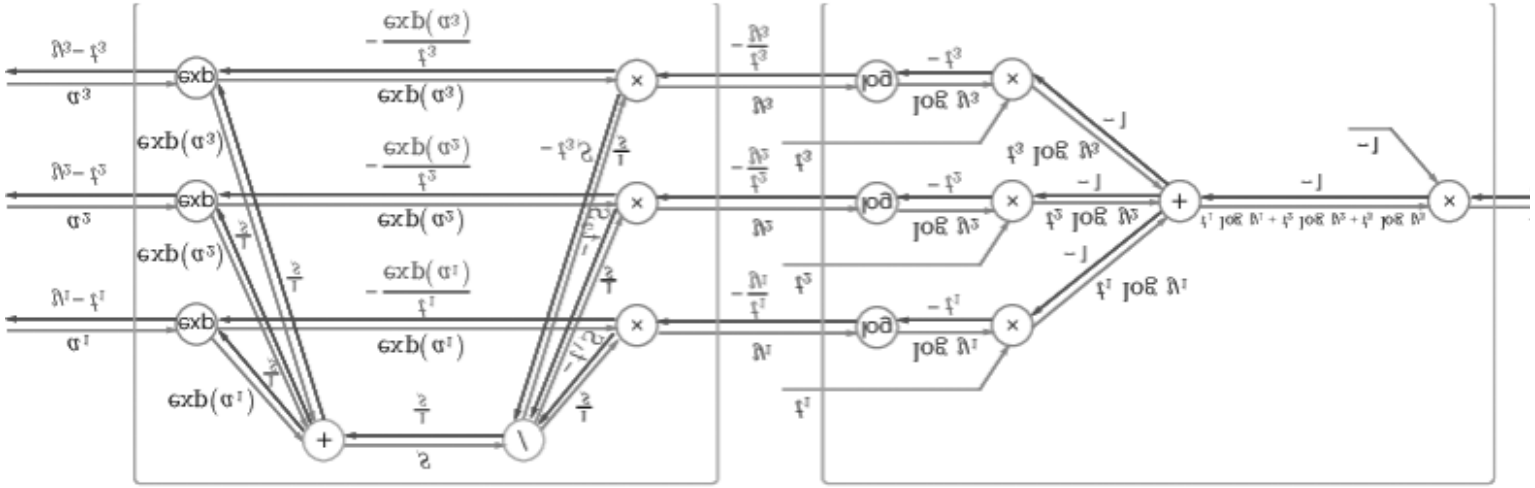
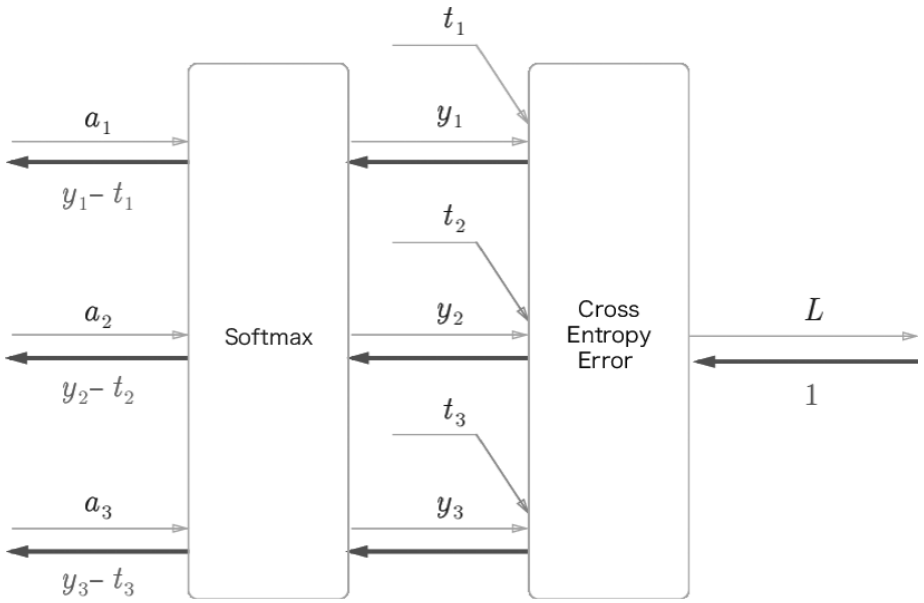


그림 5-30 '간소화한' Softmax-with-Loss 계층의 계산 그래프



학습에는 softmax 사용
추론에는 최대값을 갖는 출력 선택 (softmax 불필요)

5.6.3 Softmax-with-Loss 계층

Softmax-with-Loss
- 역전파: y - t
- 깔끔하게 나오도록 엔트로피 오차가 설계되었음

```
class SoftmaxWLoss:
    def __init__(self, W):
        self.loss = None
        self.y, self.t = None

    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
```

5.7.1 신경망 학습의 전체 그림

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.
기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시.

3단계 - 매개변수 갱신

가중치 매개변수를 기울기 방향으로 아주 조금 갱신.

4단계 - 반복

1~3단계를 반복.

5.7.2 오차역전파법을 적용한 신경망 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.

4장의 구현 (과제 #4)과 유사

5.7.2 오차역전파법을 적용한 신경망 구현하기

계층을 사용함으로써 인식 결과를 얻는 처리(predict ( ))와 기울기를 구하는 처리(gradient ( )) 계층의 전파로 동작이 이루어지는 것
- orderedDict() 사용하여 순서대로 클래스를 불러서 method 실행

```
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):
        # 가중치 초기화
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

        # 계층 생성
        self.layers = OrderedDict()
        self.layers['Affine1'] = Affine(self.params['W1'], self.params['b1'])
        self.layers['Relu1'] = Relu()
        self.layers['Affine2'] = Affine(self.params['W2'], self.params['b2'])
        self.lastLayer = SoftmaxWithLoss()
```

5.7.2 오차역전파법을 적용한 신경망 구현하기

```
def predict(self, x):
    for layer in self.layers.values():
        x = layer.forward(x)
    return x

# x : 입력 데이터, t : 정답 레이블
def loss(self, x, t):
    y = self.predict(x)
    return self.lastLayer.forward(y, t)

def accuracy(self, x, t):
    y = self.predict(x)
    y = np.argmax(y, axis=1)
    if t.ndim != 1:
        t = np.argmax(t, axis=1)
    accuracy = np.sum(y == t) / float(x.shape[0])

    return accuracy
```

5.7.2 오차역전파법을 적용한 신경망 구현하기

계층을 사용함으로써 인식 결과를 얻는 처리(predict ( ))와 기울기를 구하는 처리(gradient ( )) 계층의 전파로 동작이 이루어지는 것
- orderedDict() 사용하여 순서대로 클래스를 불러서 method 실행

```
def numerical_gradient(self, x, t):
    loss_W = lambda W: self.loss(x, t)
    grads = {}
    grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
    grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
    grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
    grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

    return grads
```

```
def gradient(self, x, t):
    # 순전파
    self.loss(x, t)
    # 역전파
    dout = 1
    dout = self.lastLayer.backward(dout)
    layers = list(self.layers.values())

    layers.reverse()

    for layer in layers:
        dout = layer.backward(dout)

    # 결과 저장
    grads = {}
    grads['W1'] = self.layers['Affine1'].dW
    grads['b1'] = self.layers['Affine1'].db
    grads['W2'] = self.layers['Affine2'].dW
    grads['b2'] = self.layers['Affine2'].db

    return grads
```

SECTION 05 오차역전파법

5.7.3 오차역전파법으로 구한 기울기 검증하기

```
(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)

x_batch = x_train[:3]
t_batch = t_train[:3]

grad_numerical = network.numerical_gradient(x_batch, t_batch)
grad_backprop = network.gradient(x_batch, t_batch)

# 각 가중치의 차이의 절댓값을 구한 후, 그 절댓값들의 평균을 낸다.
for key in grad_numerical.keys():
    diff = np.average(np.abs(grad_backprop[key] - grad_numerical[key]))
    print(key + ":" + str(diff))
```

gradient(), nuerical_gradient() 두 방식으로 구한 기울기가 일치함(엄밀히 말하면 거의 같음)을 확인하는 작업을 기울기 확인(gradient check) 필요

코드의 실행 결과는다음과 같다

```
W2:9.71260696544e-13
b2:1.20570232964e-10
W1:2.86152966578e-13
b1:1.19419626098e-12
```

5.7.4 오차역전파법을 사용한 학습 구현하기


```
import sys
import os
import numpy as np
sys.path.append(os.pardir)
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)

# 하이퍼 파라메터
iters_num = 10000  # 반복횟수
train_size = x_train.shape[0]
batch_size = 100  # 미니배치 크기
learning_rate = 0.1

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

# 1에폭당 반복 수
iter_per_epoch = max(train_size / batch_size, 1)

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

    # 오차역전파법으로 기울기 계산
    grad = network.gradient(x_batch, t_batch)

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

    # 학습 경과 기록
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    # 1에폭 당 정확도 계산
    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 | " + str(train_acc) + ", " + str(test_acc))
```