Chapter 4

Backpropagation

1. Computational graph
2. Chain rule
3. Backpropagation
4. 단순한 계층 구현
5. 활성화 함수 구현
6. 출력 함수 구현
7. 최종 구현

학습 목표

✓ 학습에 대해 이해한다.
✓ 주요 학습 기술들을 이해한다.
✓ 신경망을 직접 구현한다.

주요 내용

✓ 딥러닝의 기본인 신경망 구현에 필요한 수식 이해
✓ 딥러닝에 사용되는 Loss Function 이해
✓ 신경망 학습 방법 및 알고리즘 구현

4

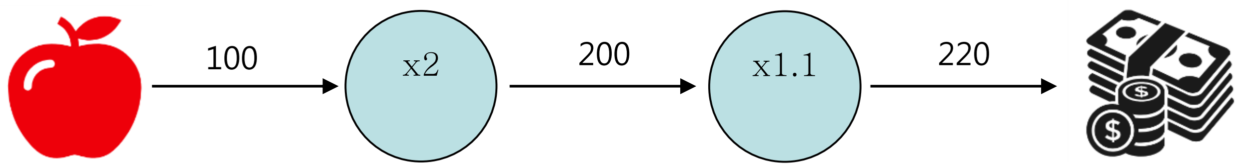
1. Computational Graph

Computational Graph

❖ Computational graph (계산 그래프)

- 계산 그래프 란?
    - 계산 과정을 그래프로 나타낸 것
    - 그래프는 자료구조를 나타내며, Node(노드) 와 Edge(엣지)로 표현

- 문제 : A 군은 슈퍼에서 1개에 100원인 사과를 2개 샀다. 이때의 지불 금액은 얼마인가?
  단, 소비세가 10% 부과된다.

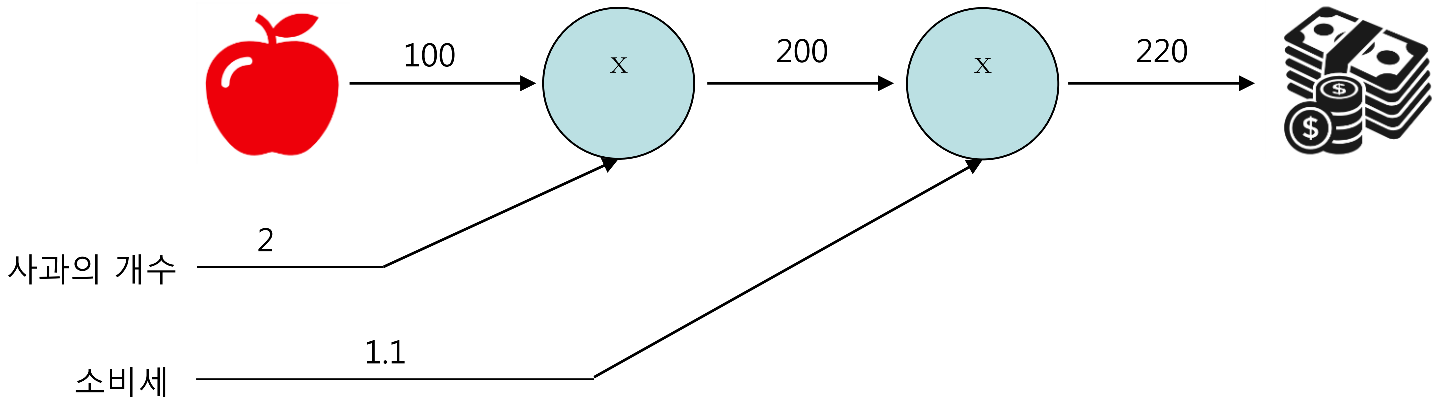


Computational Graph

❖ Computational graph (계산 그래프)

- 문제 : A 군은 슈퍼에서 1개에 100원인 사과를 2개 샀다. 이때의 지불 금액은 얼마인가?
  단, 소비세가 10% 부과된다.

- 그래프를 좀더 자세하게 그려봅시다.

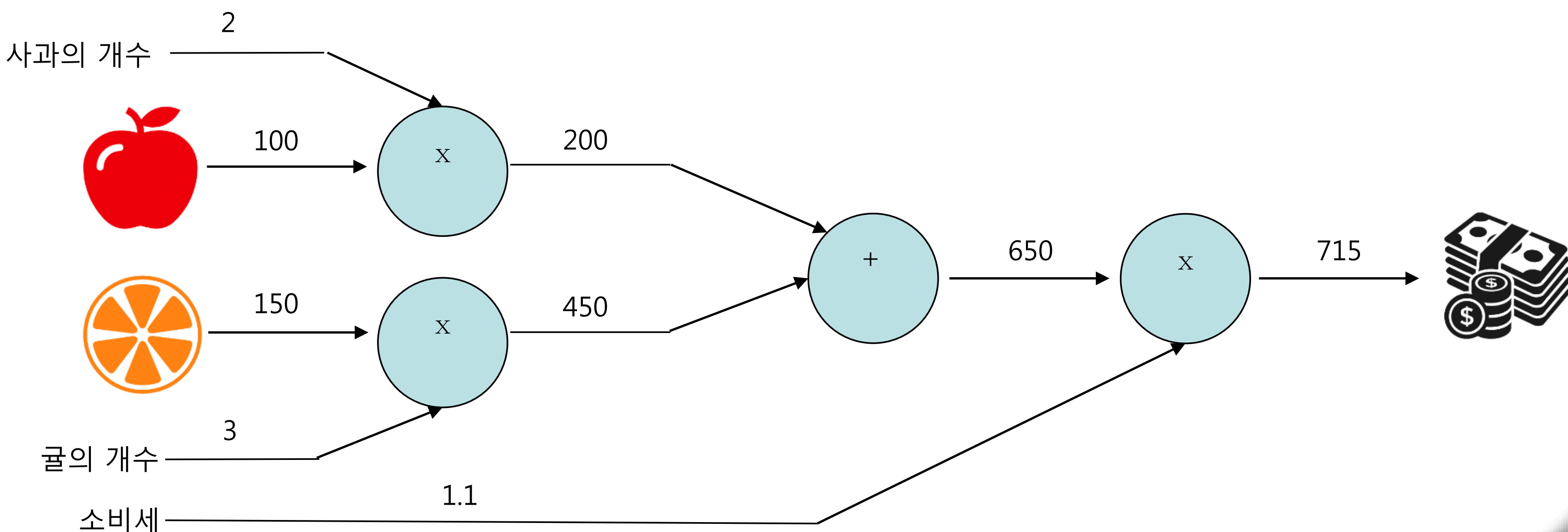


Computational Graph

❖ Computational graph (계산 그래프)

- 문제 : A 군은 슈퍼에서 사과 2개, 귤 3개를 샀다. 사과는 개당 100원, 귤은 개당 150원 이다. 이때의 지불 금액은 얼마인가?
 단, 소비세가 10% 부과된다.

- 계산 그래프를 그려봅시다.



Computational Graph

❖ Computational graph (계산 그래프)

- 계산 그래프를 통하여 복잡한 문제를 단순화 하고 국소화하여 문제풀이
- 전체와 상관없이 자신과 관계된 정보만으로 "국소적 계산 " 을 하여 다음 결과를 출력

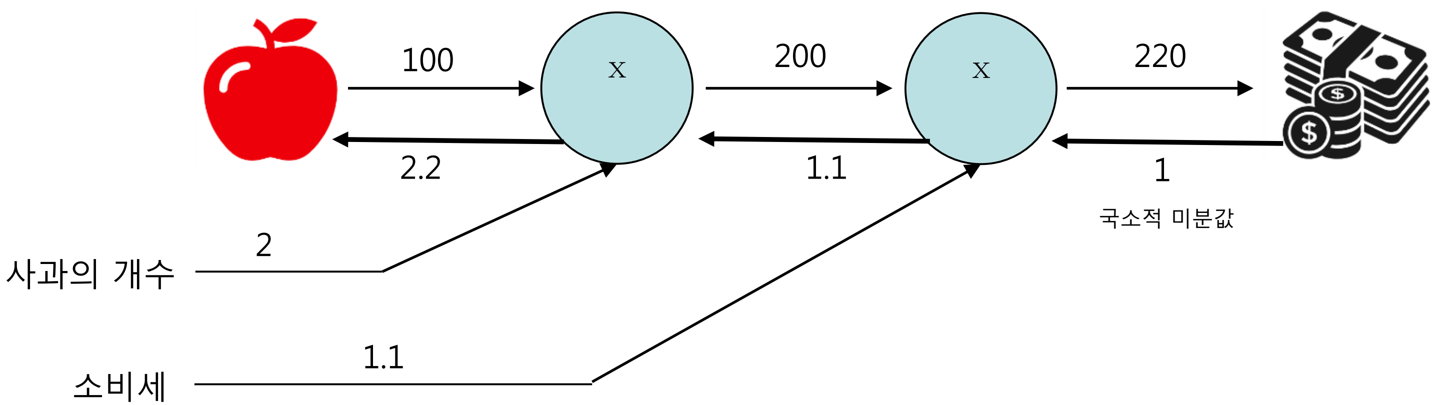
- 계산이 왼쪽에서 오른쪽으로 진행 - 순전파(forward propagation)

- 왜 계산 그래프로 문제를 풀어야 하는가?
  - 문제를 단순하게 만들어 계산 가능
  - 중간 계산 결과 보관
  - 역전파(back propagation)를 통해 미분을 효율적으로 계산할 수 있기 때문

Computational Graph

❖ Computational graph (계산 그래프)

- 문제: 사과 가격이 오르면 최종 금액에 어떤 영향을 끼칠까?
  - 사과 가격에 대한 지불 금액의 미분을 구하는 문제: $\frac{\partial L}{\partial x}$
  - x: 사과 값, L: 지불금액
  - 오른쪽에서 왼쪽으로 미분 값을 전달(연쇄 법칙)



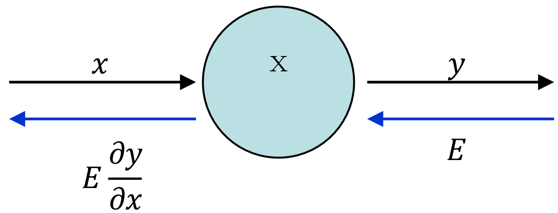
4

2. Chain Rule

Chain Rule

❖ Chain Rule (연쇄 법칙)

- $y = f(x)$의 역전파



- 순방향과 반대방향으로 국소적 미분을 곱함(파란 화살표)
- 상류에서 전달된 신호 E에 국소적 미분($\frac{\partial y}{\partial x}$)을 곱한 후 다음 노드로 전달
- 국소적 미분?
  - 순전파로 계산된 $y = f(x)$의 미분을 구하는 것
  - x에 대한 y의 미분($\frac{\partial y}{\partial x}$)을 구하는 것

Chain Rule

❖ Chain Rule (연쇄 법칙)의 원리

- 합성 함수: 여러 함수로 구성된 함수
- 예: $z = (x + y)^2$
  - $z = t^2$
  - $t = x + y$

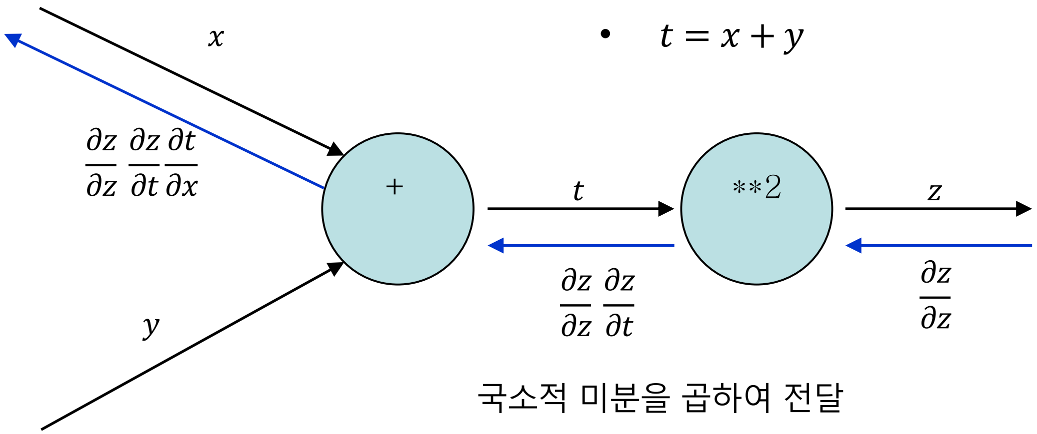
- 연쇄법칙: 합성 함수의 미분은 합성 함수를 구성하는 각 함수의 미분의 곱으로 나타낼 수 있음
  - $z = t^2$
  - $t = x + y$
  - $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x}$ ⇒ $\frac{\partial z}{\partial x} = \frac{\partial z}{\cancel{\partial t}}\frac{\cancel{\partial t}}{\partial x}$
  - 연쇄법칙을 이용하여 $\frac{\partial z}{\partial x}$를 구해봅시다
    - $\frac{\partial z}{\partial t} = 2t, \frac{\partial t}{\partial x} = 1$
  - $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} = 2t \cdot 1 = 2(x + y)$

Chain Rule

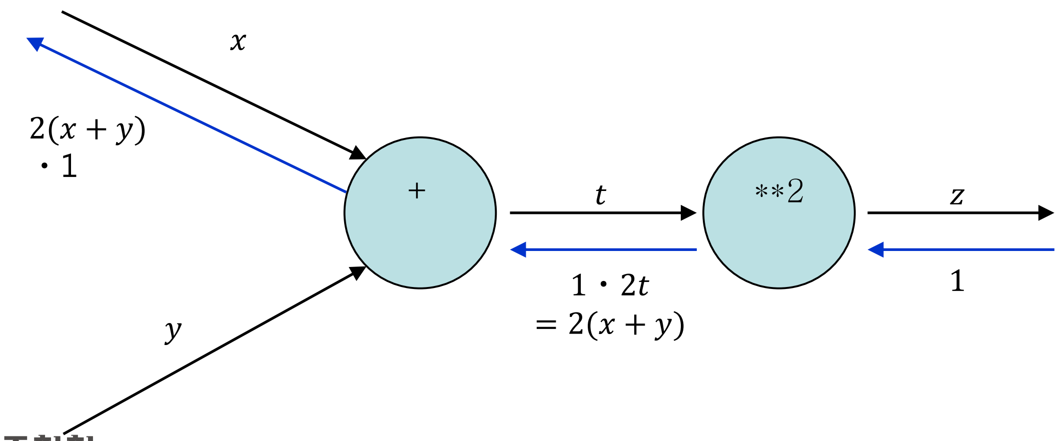
❖ Chain Rule (연쇄 법칙)과 계산 그래프

- 2제곱 계산을 '**2' 노드로 표현

- $z = t^2$
- $t = x + y$



국소적 미분을 곱하여 전달



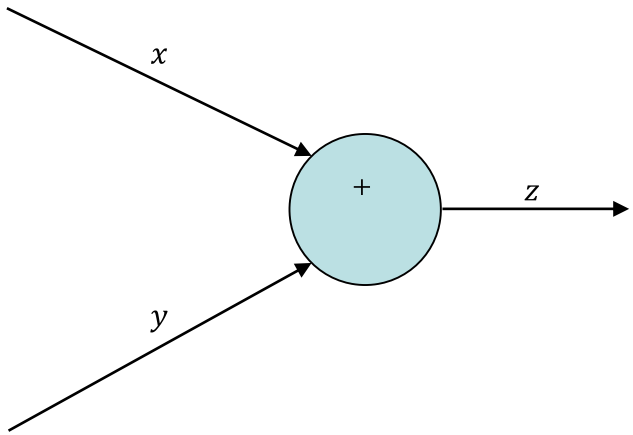
4

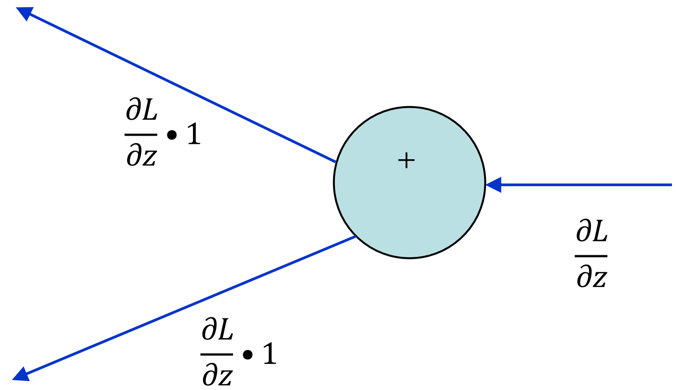
3. Backpropagation

Backpropagation

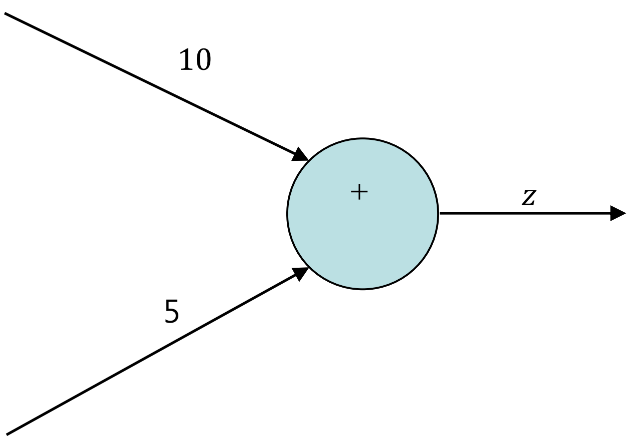
❖ 덧셈 노드의 역전파

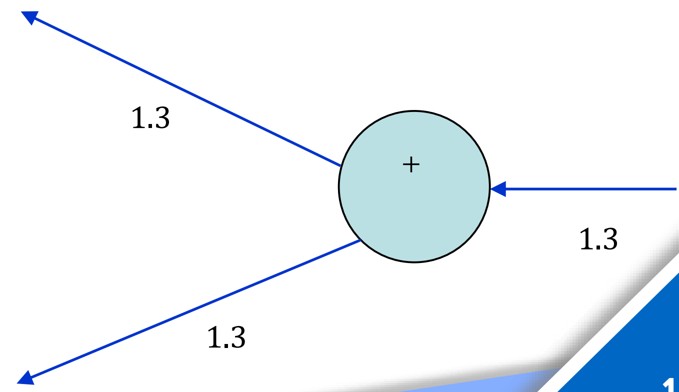
- 예: $z = x + y$
- $\frac{\partial z}{\partial x} = 1,\ \frac{\partial z}{\partial y} = 1$





- 예: $10 + 5 = 15$
- 상류에서 1.3

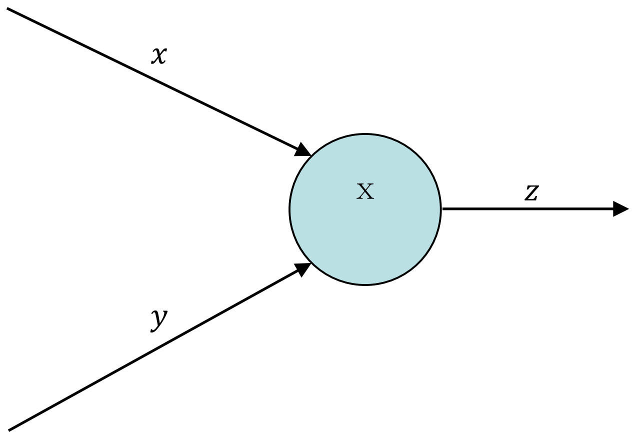


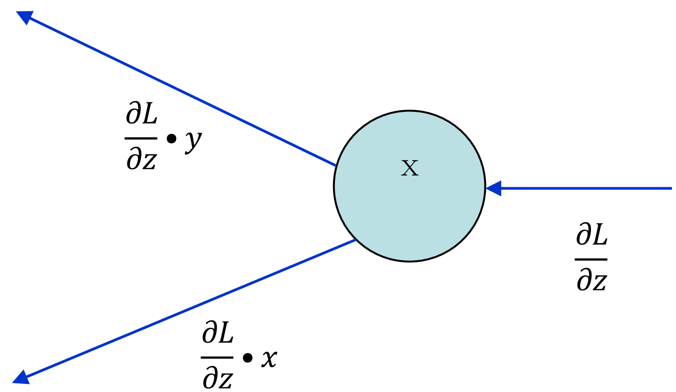


Backpropagation

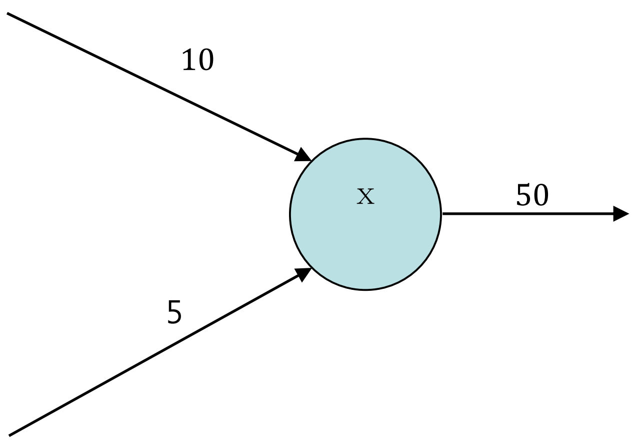
❖ 곱셈 노드의 역전파

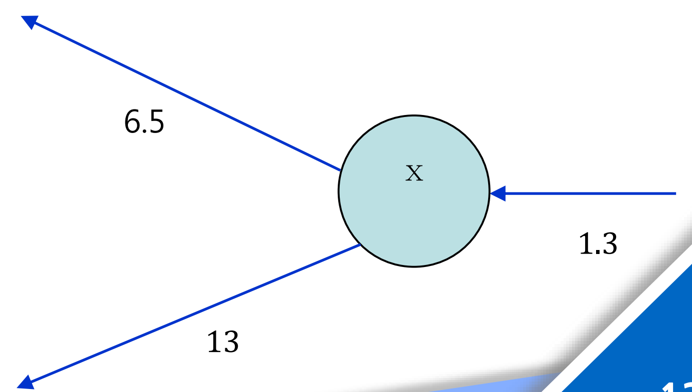
- 예: $z = xy$
- $\frac{\partial z}{\partial x} = y,\ \frac{\partial z}{\partial y} = x$





- 예: $10 \times 5 = 50$
- 상류에서 1.3

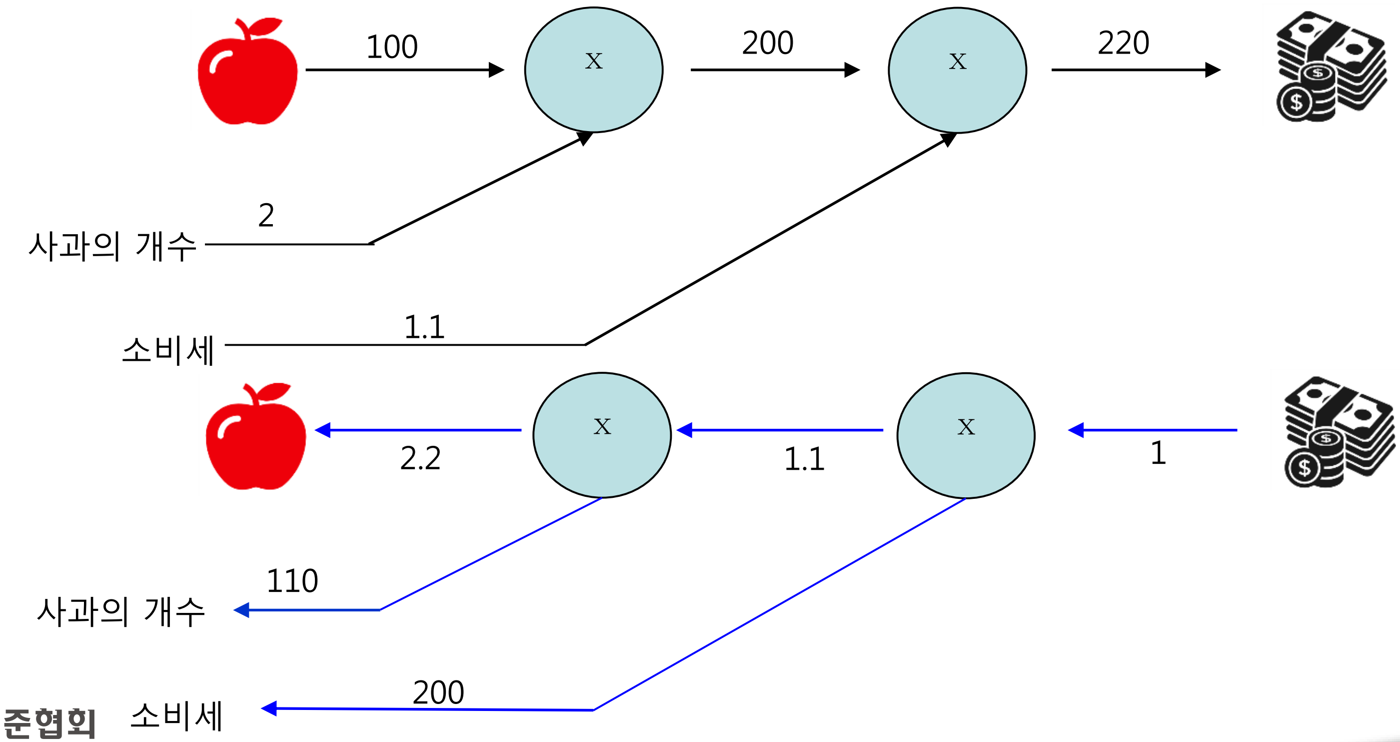




Backpropagation

❖ 사과 쇼핑의 예

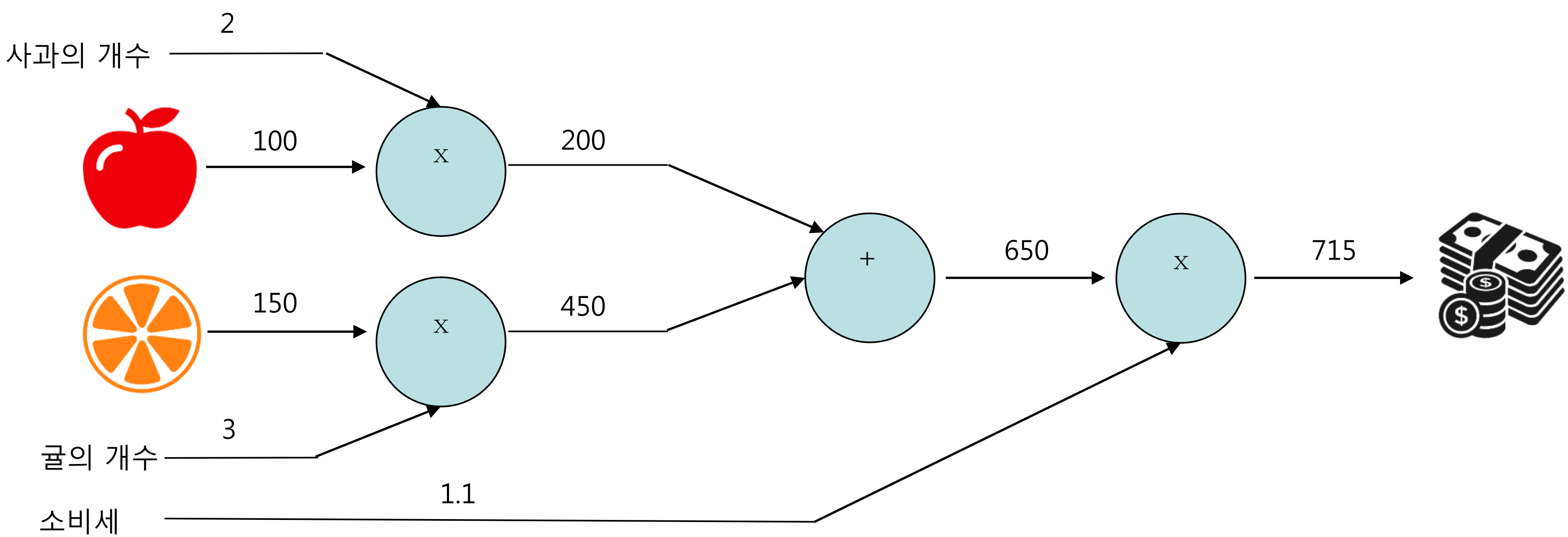
- 사과의 가격, 사과의 개수, 소비세라는 세 변수 각각이 최종 금액에 어떻게 영향을 주는가
- 사과 가격에 대한 지불 금액의 미분
- 사과 개수에 대한 지불 금액의 미분
- 소비세에 대한 지불 금액의 미분



Backpropagation

❖ 사과와 귤 쇼핑의 예

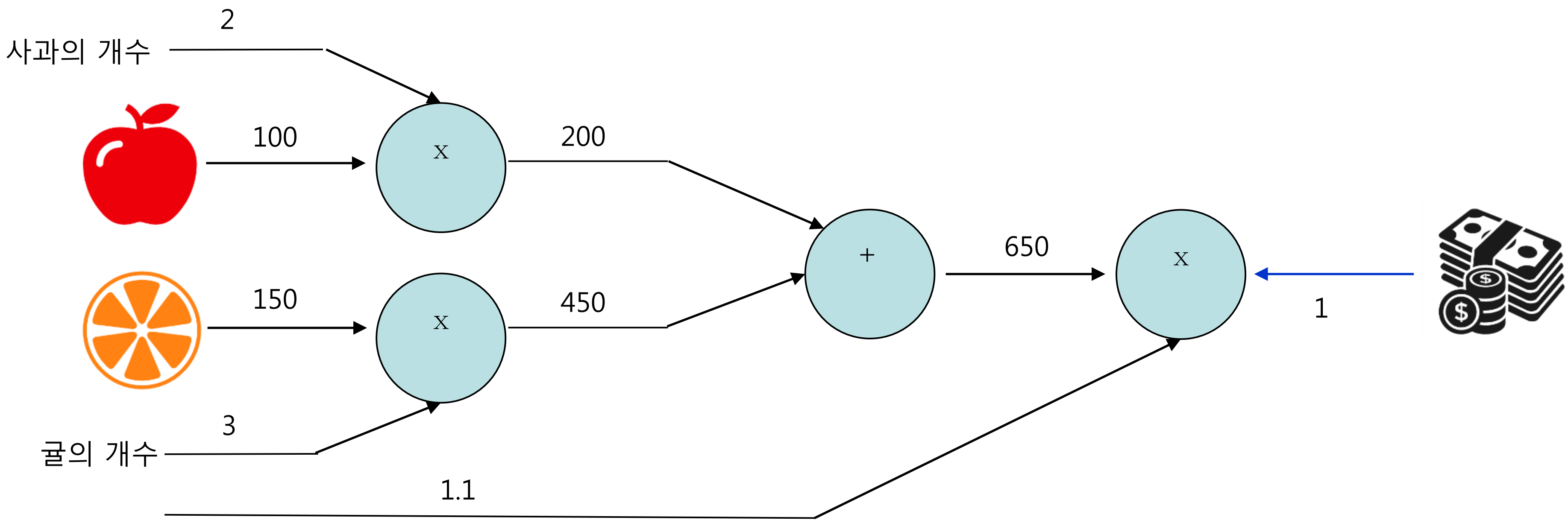
- 사과와 귤 쇼핑의 역전파



Backpropagation

❖ 사과와 귤 쇼핑의 예

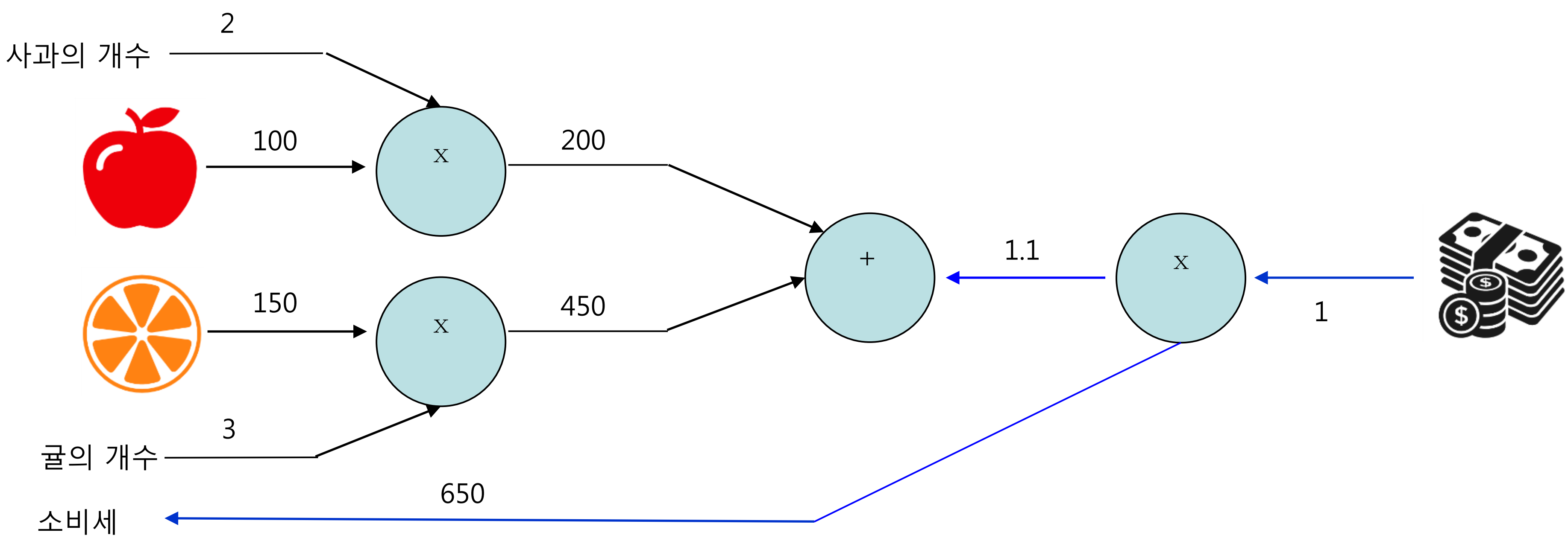
- 사과와 귤 쇼핑의 역전파



Backpropagation

❖ 사과와 귤 쇼핑의 예

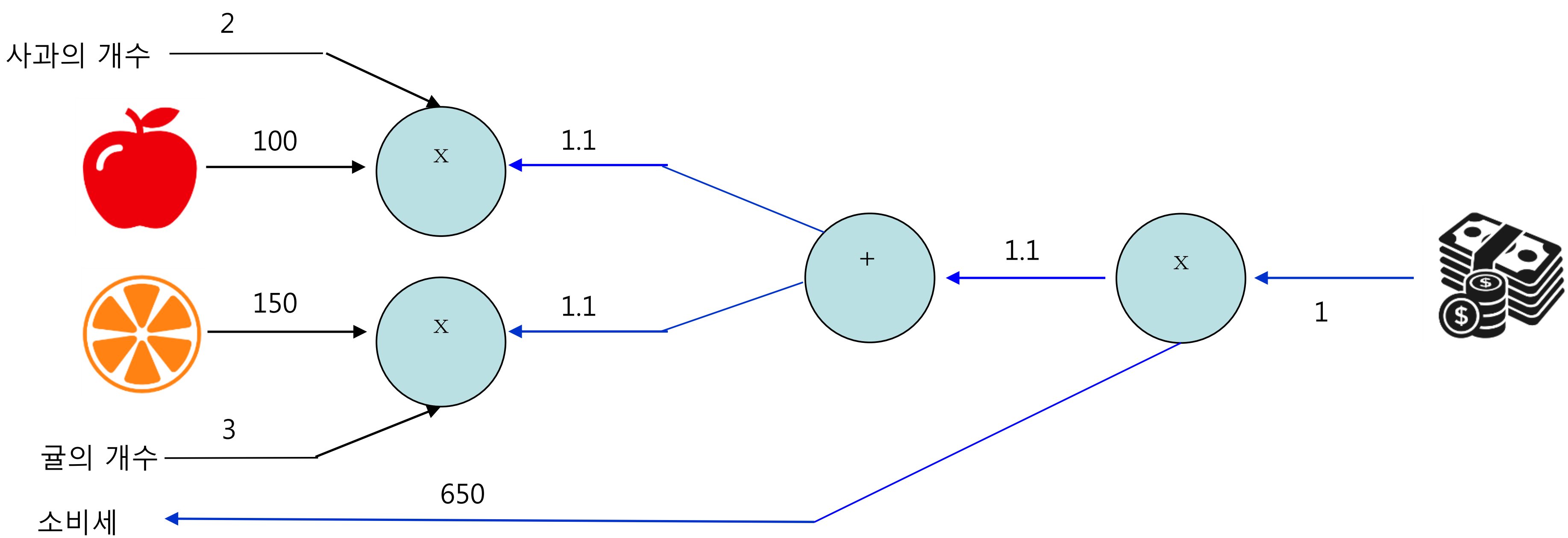
- 사과와 귤 쇼핑의 역전파



Backprogation

❖ 사과와 귤 쇼핑의 예

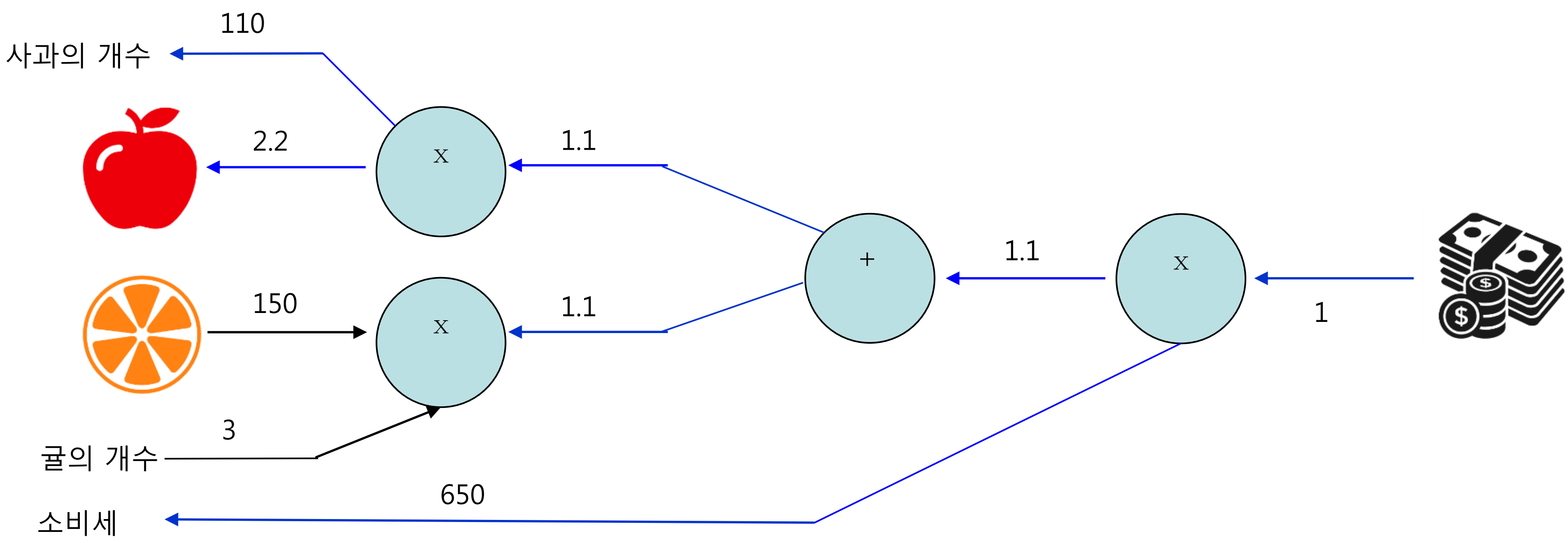
- 사과와 귤 쇼핑의 역전파



Backpropagation

❖ 사과와 귤 쇼핑의 예

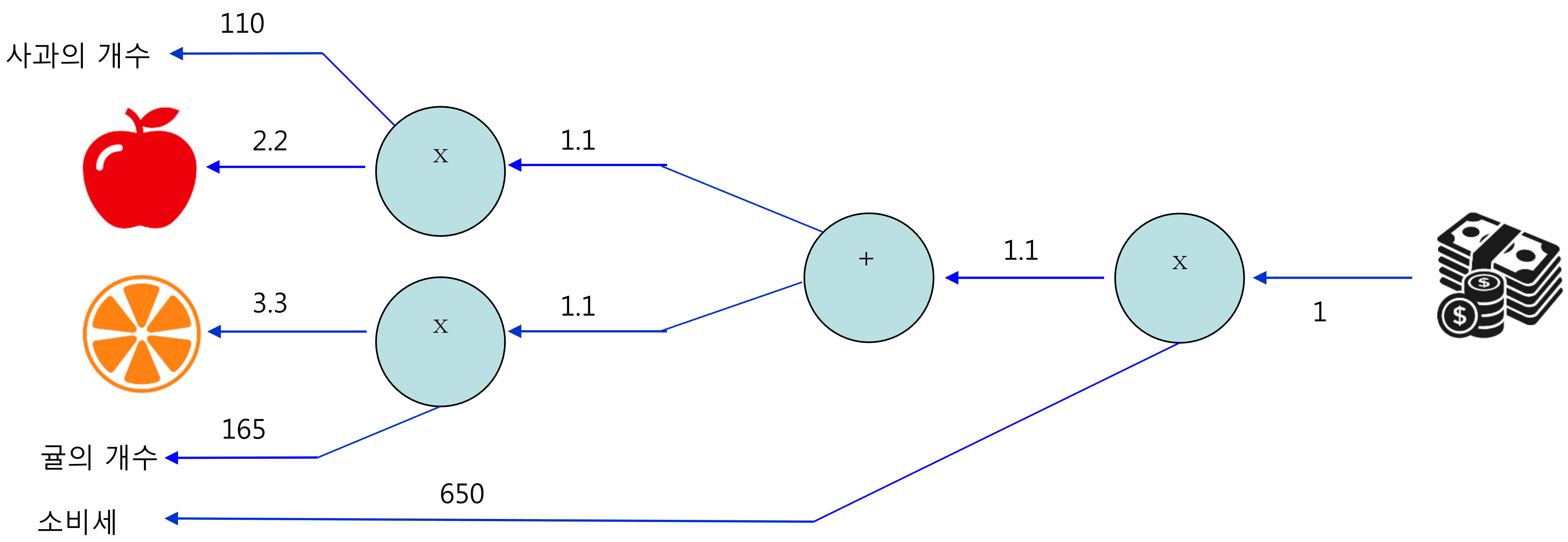
- 사과와 귤 쇼핑의 역전파



Backpropagation

❖ 사과와 귤 쇼핑의 예

- 사과와 귤 쇼핑의 역전파



4

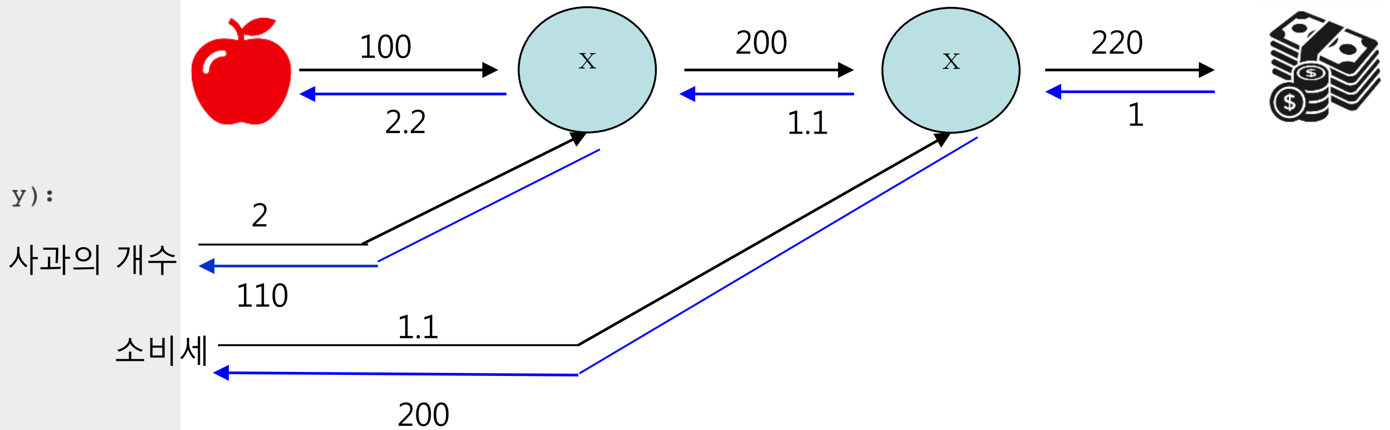
4. 단순한 계층 구현하기

단순한 계층 구현하기

❖ 곱셈 계층 구현하기

- '사과 쇼핑' 예를 구현하여 보자
- 계산 그래프의 곱셈 노드를 'MulLayer', 덧셈 노드를 'AddLayer' 로 구현
- forward()는 순전파, backward() 역전파

```
In [1]:
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
        dy = dout * self.x

        return dx, dy
```



단순한 계층 구현하기

❖ 곱셈 계층 구현하기

In [2]:
```
apple = 100
apple_num = 2
tax = 1.1
```

In [3]:
```
mul_apple_layer = MulLayer()
mul_tax_layer = MulLayer()
```

In [4]:
```
apple_price = mul_apple_layer.forward(apple, apple_num)
price = mul_tax_layer.forward(apple_price, tax)
```

In [5]:
```
dprice = 1
dapple_price, dtax = mul_tax_layer.backward(dprice)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)
```

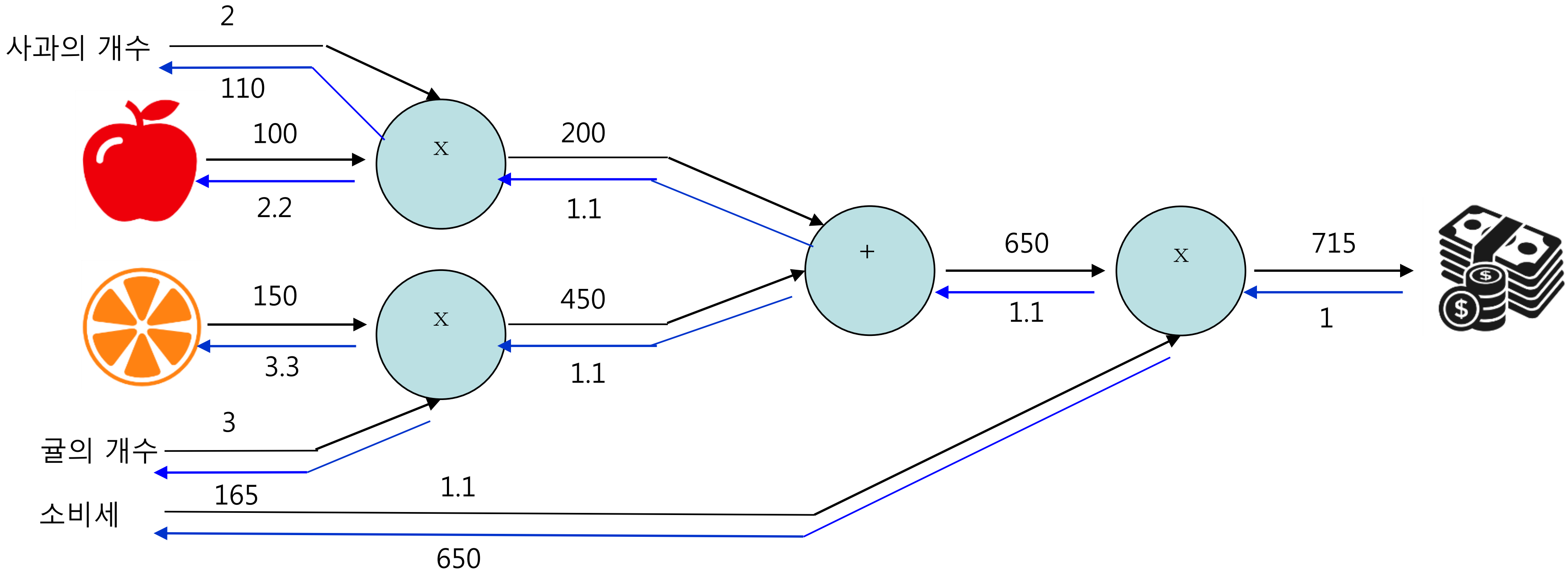
In [6]:
```
print("price:", int(price))
print("dApple:", dapple)
print("dApple_num:", int(dapple_num))
print("dTax:", dtax)
```

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

단순한 계층 구현하기

❖ 덧셈 계층 구현하기

- forward() x와 y를 인수로 받고 두 값을 더하여 반환
- backward() 상류에서 넘어온 미분을 그대로 하류로 전달



단순한 계층 구현하기

❖ 덧셈 계층 구현하기

In [7]:
```
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
```

In [8]:
```
apple = 100
apple_num = 2
orange = 150
orange_num = 3
tax = 1.1
```

In [9]:
```
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()
```

단순한 계층 구현하기

❖ 덧셈 계층 구현하기

In [10]:
```
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)
```

In [11]:
```
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)
```

In [12]:
```
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
```

4

5. 활성화 함수 구현

활성화 함수 구현

❖ 활성화 함수 구현하기

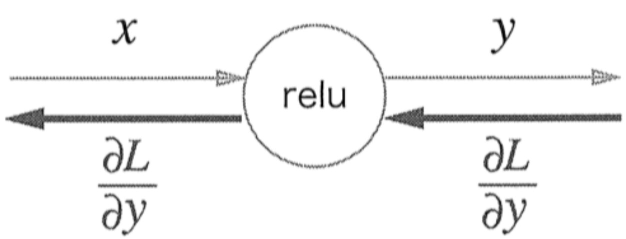
- ReLU 계층 구현하기
- 순전파 때의 입력인 x가 0보다 크면 역전파는 상류의 값을 그대로 하류로 전달
- 순전파 때의 x가 0 이하면 역전파 때는 하류로 신호 전달 하지 않음(0 전달)
- ReLU 식

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

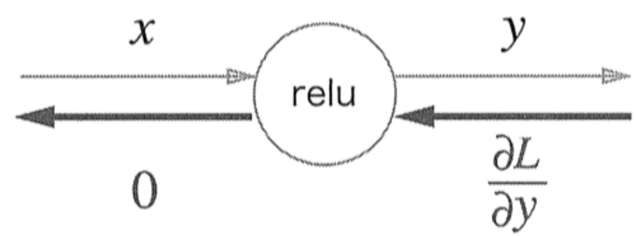
- x에 대한 y의 미분

$$\frac{\partial y}{\partial x} = \begin{cases} 1, & (x > 0) \\ 0, & (x \leqq 0) \end{cases}$$

$\boxed{x > 0}$



$\boxed{x \leq 0}$



활성화 함수 구현

❖ 활성화 함수 구현하기

- ReLU 계층 구현하기

In [13]:
```
class Relu:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout

        return dx
```

활성화 함수 구현

❖ 활성화 함수 구현하기

- ReLU 계층 구현하기

In [13]:
```python
class Relu:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout

        return dx
```

In [14]:
```python
import numpy as np
x = np.array([[1.0, -0.5], [-2.0, 3.0]])
print(x)
```

```
[[ 1.  -0.5]
 [-2.   3. ]]
```

In [15]:
```python
mask = (x <= 0)
print(mask)
```

```
[[False  True]
 [ True False]]
```

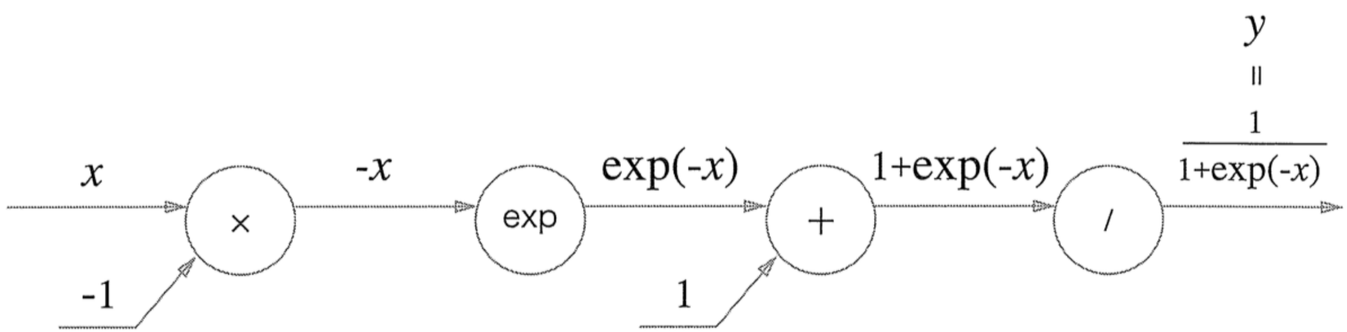
활성화 함수 구현

❖ 활성화 함수 구현하기

- Sigmoid 계층 구현하기
- Sigmoid 식

$$y = \frac{1}{1 + e^{-x}}$$

- Sigmoid 계층의 계산 그래프



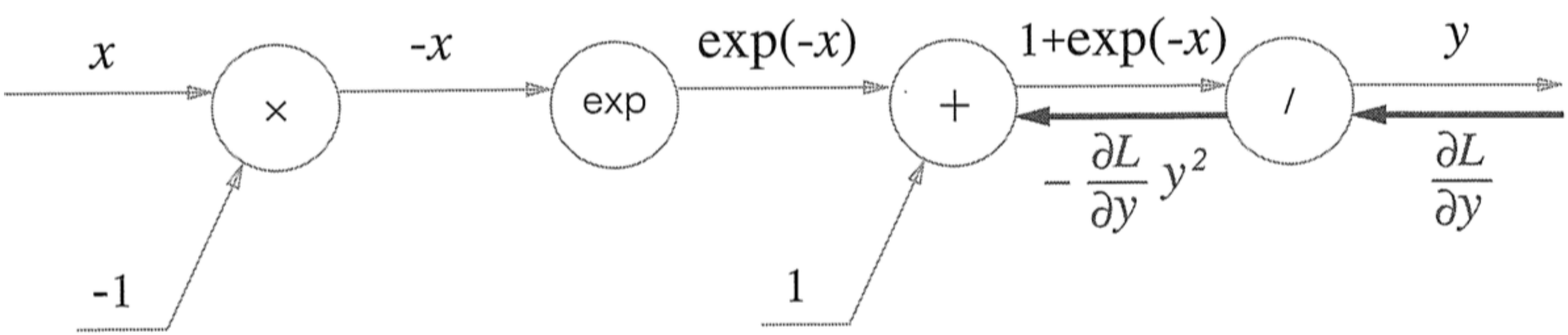
- 'exp' 노드는 $y = \exp(x)$
- '/' 노드는 $y = \frac{1}{x}$

활성화 함수 구현

❖ 활성화 함수 구현하기

- Sigmoid 계층 구현하기
- 1 단계
- '/' 노드, 즉 $y = \frac{1}{x}$ 을 미분하면 ? $\frac{\partial y}{\partial x} = -\frac{1}{x^2} = -y^2$

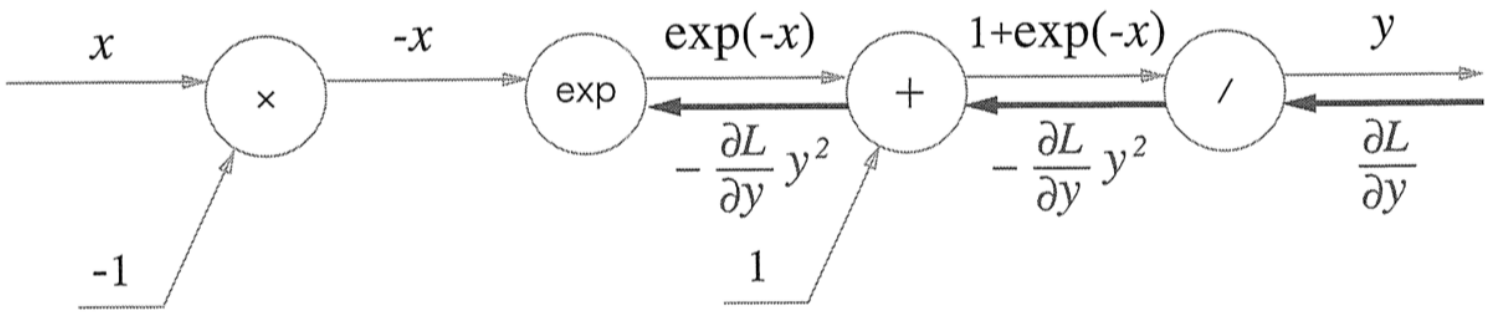
- 1단계 계산 그래프



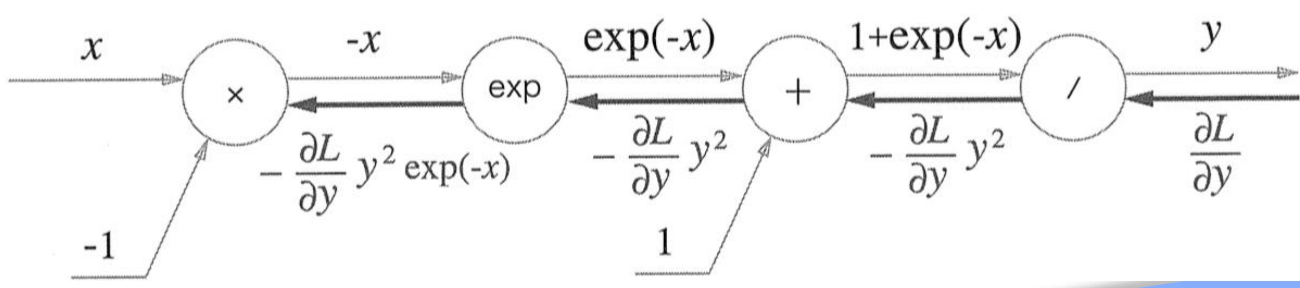
활성화 함수 구현

❖ 활성화 함수 구현하기

- 2단계 계산 그래프
  - '+' 노드는 상류의 값을 하류로 전달



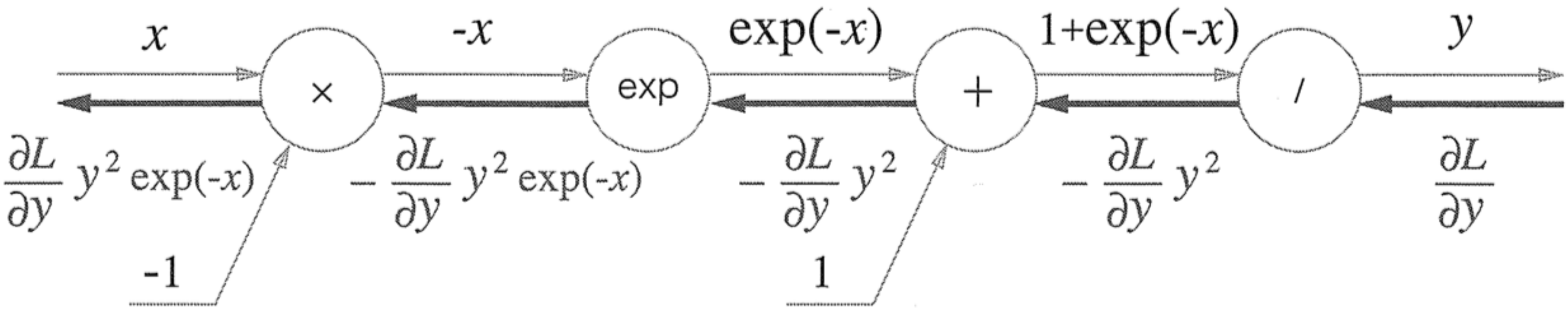
- 3단계 계산 그래프
  - 'exp' 노드는 $y = \exp(x)$ 연산 수행 $\frac{\partial y}{\partial x} = e^x$
  - 상류의 값에 순전파 때의 출력 $\exp(-x)$ 을 곱해 하류로 전달



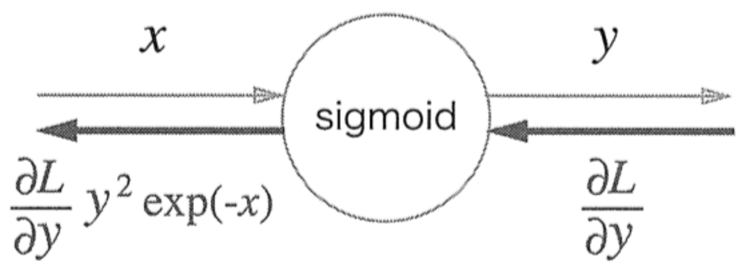
활성화 함수 구현

❖ 활성화 함수 구현하기

- 4단계 계산 그래프
  - 'x' 노드는 순전파 때의 값을 '서로 바꿔' 곱하는 것 (이 예에서는 -1 을 곱함)



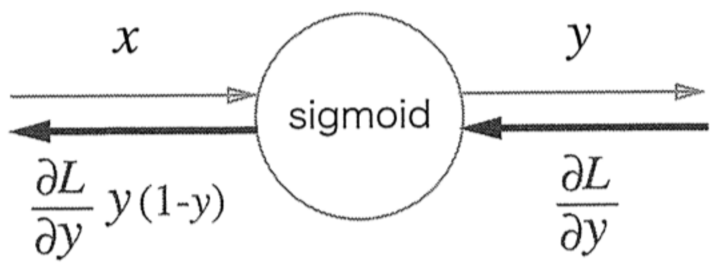
- 역전파의 최종 출력인 $\frac{\partial L}{\partial y} y^2 e^{-x}$ 가 하류 노드로 전달
  - $\frac{\partial L}{\partial y} y^2 e^{-x}$ 를 순전파의 입력 x와 출력 y만으로 계산 가능 -> Sigmoid 노드 하나로 대체 가능



활성화 함수 구현

❖ 활성화 함수 구현하기

- $\frac{\partial L}{\partial y} y^2 e^{-x}$ 의 정리
- $$\begin{aligned} \frac{\partial L}{\partial y} y^2 e^{-x} &= \frac{\partial L}{\partial y} \frac{1}{1+e^{-x^2}} \\ &= \frac{\partial L}{\partial y} \frac{1}{1+e^{-x}} \frac{e^{-x}}{1+e^{-x}} \\ &= \frac{\partial L}{\partial y} y(1-y) \end{aligned}$$
- 즉, Sigmoid 계층의 역전파는 순전파의 출력 (y)만으로 계산 가능



활성화 함수 구현

❖ 활성화 함수 구현하기

- Sigmoid 계층 구현하기

```
In [16]:
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 * (1.0 - self.out)* self.out

        return dx
```

4

6. 출력 함수 구현

출력 함수 구현

❖ 출력 함수 구현하기

- Affine 계층
  - 신경망의 순전파 때 수행하는 행렬의 내적은 기하학에서는 어파인 변환(Affine transformation)
  - X, W, B는 각각 형상이 (2,), (2, 3), (3,) 인 다차원 배열
  - 뉴런의 가중치 합: Y = np.dot(X, W) + B

X • W = O

(2,)　　(2, 3)　　(3,)

일치

```
In [16]:
X = np.random.rand(2)
W = np.random.rand(2, 3)
B = np.random.rand(3)
```

```
In [17]:
print(X.shape)
print(W.shape)
print(B.shape)
```

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

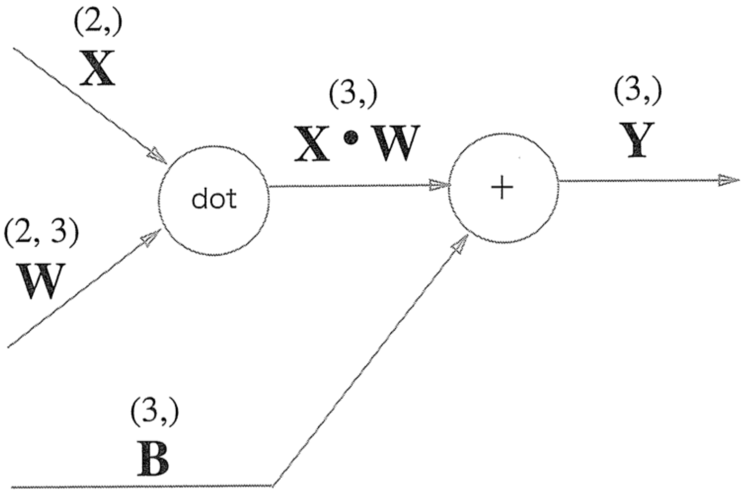
출력 함수 구현

❖ 출력 함수 구현하기

- Affine 계층
  - Y = np.dot(X, W) + B의 계산 그래프
  - **X, W, B** 는 다차원 배열

In [18]:
```
Y = np.dot(X, W) + B
print(Y)
```

```
[1.91203763 0.79620892 1.18816205]
```



출력 함수 구현

❖ 출력 함수 구현하기

- Affine 계층의 역전파
  - 행렬을 사용한 역전파도 행렬의 원소마다 전개하여 보면 스칼라 값과 동일하게 계산 그래프 그릴 수 있음
  - Y = np.dot(X, W) + B의 계산 그래프
  - $\boldsymbol{W}^T$ 의 T 는 전치 행렬
    - 전치 행렬은 W의 (i, j)위치의 원소를 (j, i)위치로 바꾼 것

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

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

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

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

출력 함수 구현

❖ 출력 함수 구현하기

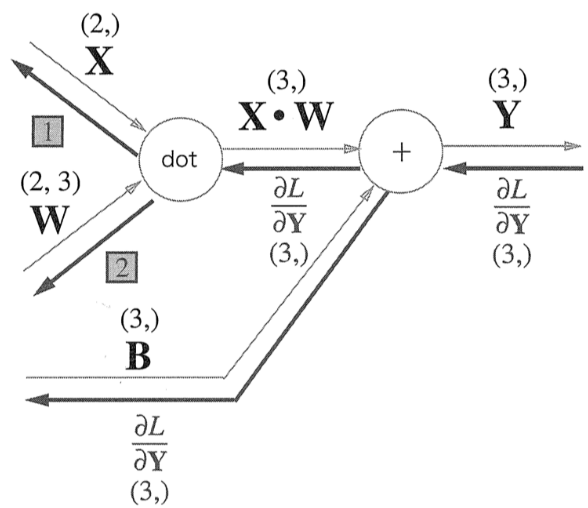
- Affine 계층의 역전파 계산 그래프
- X와 $\frac{\partial L}{\partial x}$은 같은 형상, W와 $\frac{\partial L}{\partial x}$은 같은 형상

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

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

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

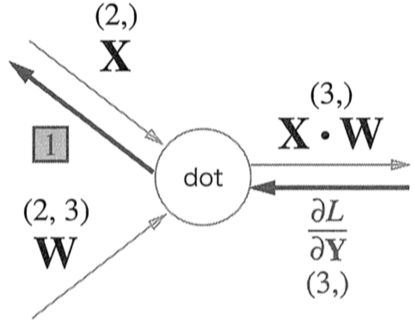
2 $\quad \underset{(2,\,3)}{\frac{\partial L}{\partial \mathbf{W}}} = \underset{(2,\,1)}{\mathbf{X}^{\mathrm{T}}} \underset{(1,\,3)}{\frac{\partial L}{\partial \mathbf{Y}}}$



- 왜 행렬의 형상을 계속 주의해야 할까?
  - 행렬의 내적에서는 대응하는 차원의 원소 수를 일치시켜야 함

$$\underset{(3,\ )}{\frac{\partial L}{\partial Y}} \bullet \underset{(3,\ 2)}{W^T} = \underset{(2,\ )}{\frac{\partial L}{\partial X}}$$

일치



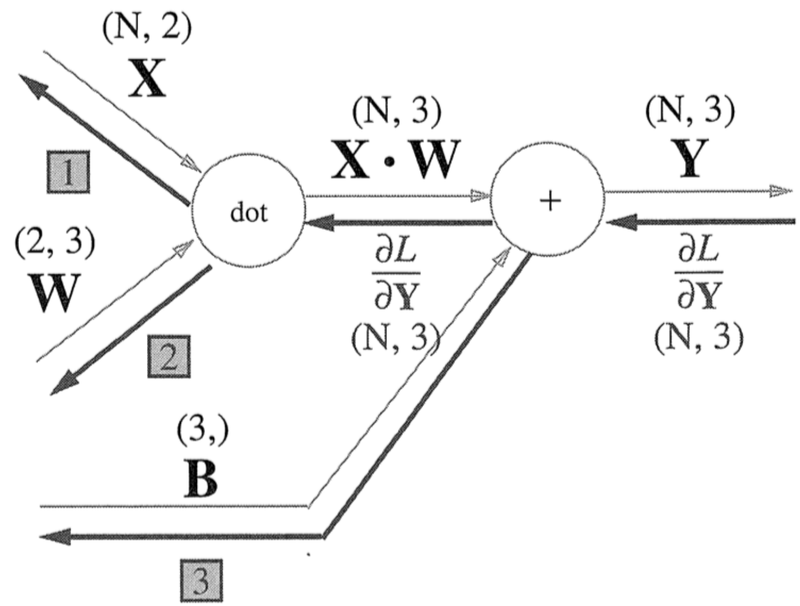
출력 함수 구현

❖ 출력 함수 구현하기

- 배치 Affine 계층
- 데이터 N개를 묶어 순전파 하는 경우의 Affine 계층을 구현
    - 기존과 다른 부분은 입력 부분의 **X**의 형상이 (N, 2)로 바뀐 것 뿐

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

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



$$\boxed{3}\quad \underset{(3)}{\frac{\partial L}{\partial \mathbf{B}}} = \underset{(N,\,3)}{\frac{\partial L}{\partial \mathbf{Y}}}$$ 의 첫 번째 축(제0축, 열방향)의 합

출력 함수 구현

❖ 출력 함수 구현하기

- 배치 Affine 계층: 편향
- 데이터 N개를 더해서 구함

```
In [19]: X_dot_W = np.array([[0, 0, 0], [10, 10, 10]])
         B = np.array([1, 2, 3])

In [20]: X_dot_W

array([[ 0,  0,  0],
       [10, 10, 10]])

In [21]: X_dot_W + B

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

```
In [22]: dY = np.array([[1, 2, 3], [4, 5, 6]])
         dY

array([[1, 2, 3],
       [4, 5, 6]])

In [25]: dB = np.sum(dY, axis = 0)
         dB

array([5, 7, 9])
```

출력 함수 구현

❖ 출력 함수 구현하기

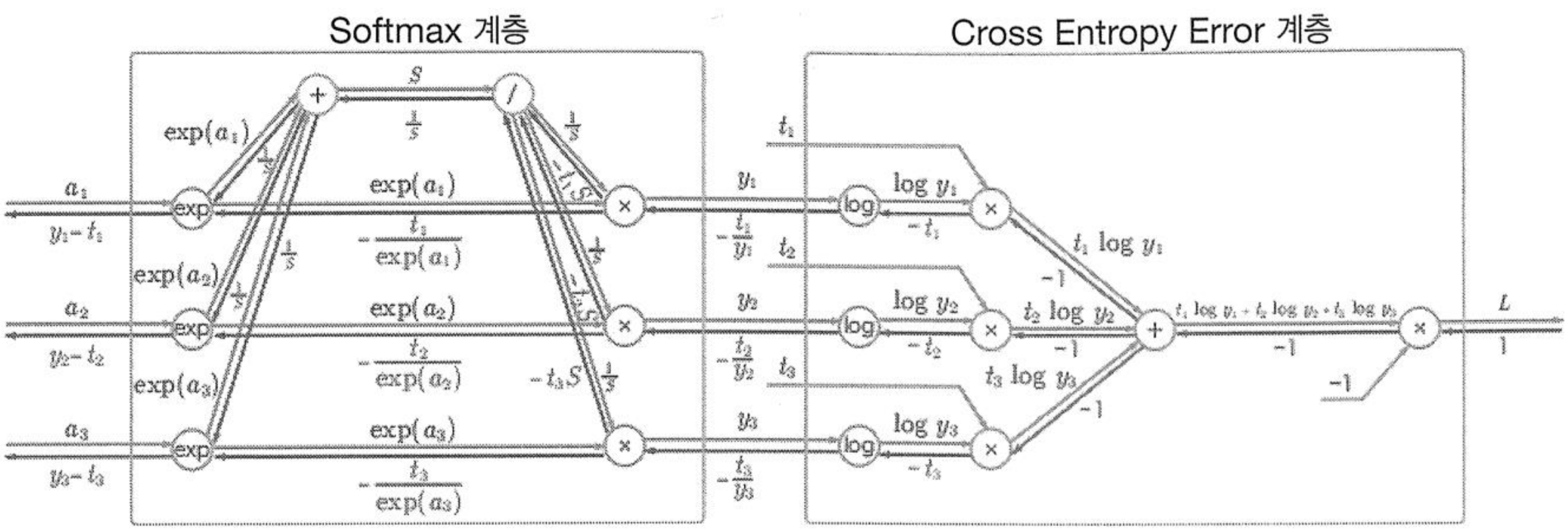
- 배치 Affine 계층 최종 구현

In [24]:
```
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 backword(self, out):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis = 0)

        return dx
```

출력 함수 구현

❖ 출력 함수 구현하기

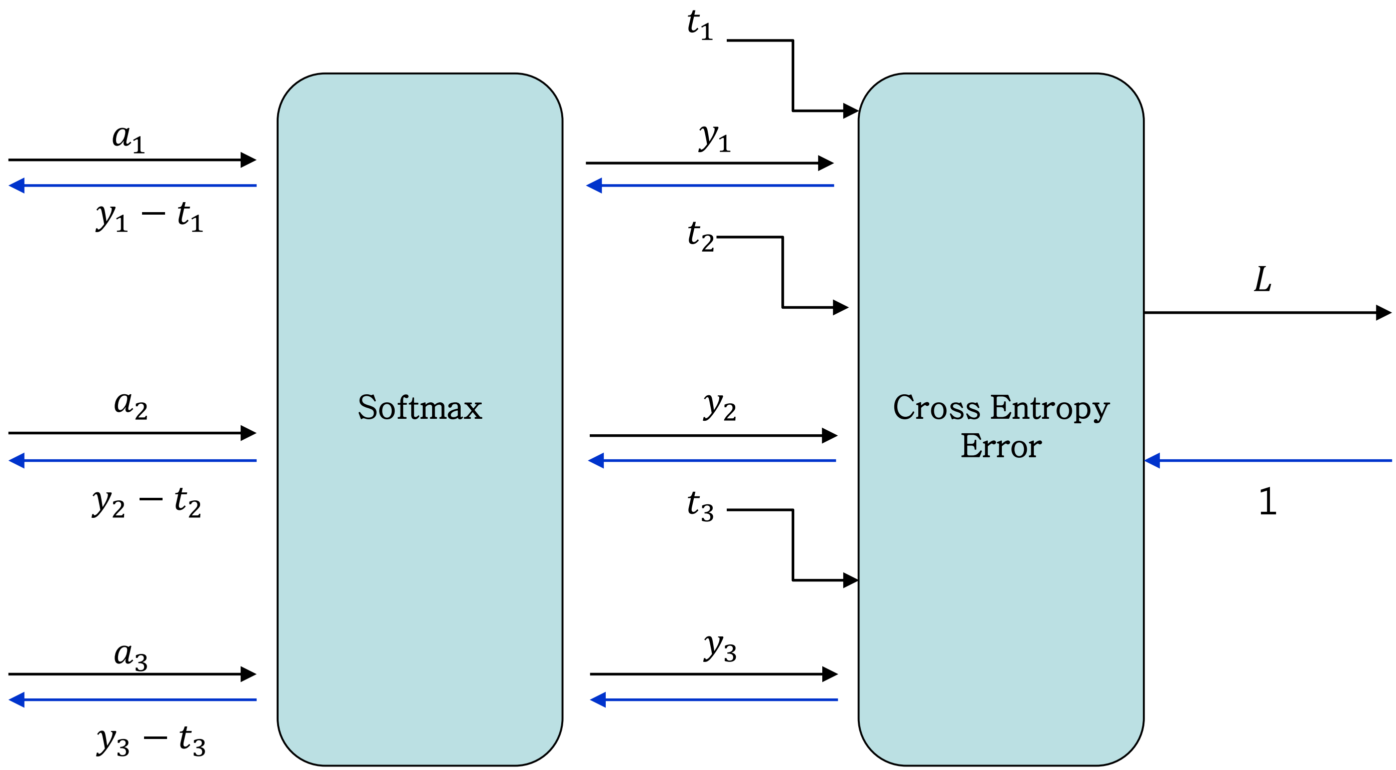
- Softmax 계층 + 손실 함수(교차 엔트로피 함수)



출력 함수 구현

❖ 출력 함수 구현하기

- Softmax 계층 + 손실 함수(교차 엔트로피 함수)



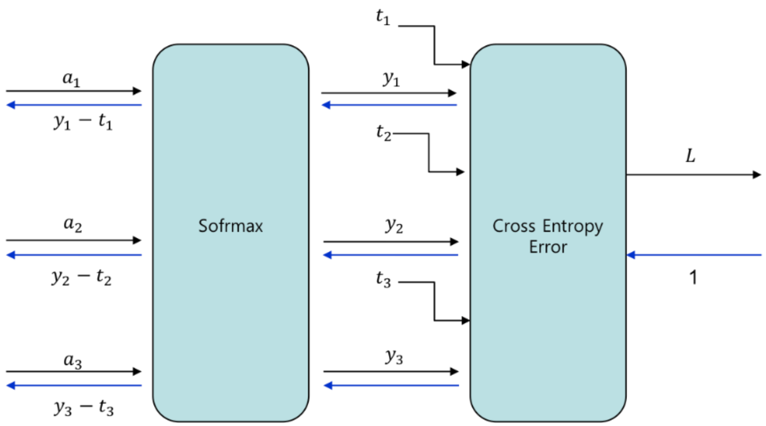
출력 함수 구현

❖ 출력 함수 구현하기

- Softmax 계층 + 손실 함수(Cross Entropy Error)

- Softmax 계층은 (a_1, a_2, a_3) 를 정규화 하여 (y_1, y_2, y_3) 출력
- Cross Entropy Error 계층은 출력 (y_1, y_2, y_3)와 정답 레이블

(t_1, t_2, t_3) 를 받아서 이들로 부터 손실 L 출력
- Softmax 계층의 역전파 결과: $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$
  - 이는, Softmax계층의 출력과 정답 레이블의 차
  - 신경망의 역전파에서는 이 차이인 오차가 앞 계층에 전달



출력 함수 구현

❖ 출력 함수 구현하기

- Softmax 계층 + 손실 함수(교차 엔트로피 함수): Code

```
def softmax(matrix):
    maximum_of_matrix = np.max(matrix)
    difference_from_maximum = matrix - maximum_of_matrix
    exponential_of_difference = np.exp(difference_from_maximum)
    sum_of_exponential = np.sum(exponential_of_difference)
    y = exponential_of_difference / sum_of_exponential
    return y
def cross_entropy_error(y, t):
    delta = 1e-7
    return -np.sum(t * np.log(y + delta))

class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None
        self.y = None
        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
```

4

7. 오차역전파법을 적용한 신경망 구현

오차역전파법을 적용한 신경망 구현

- 신경망 학습의 전체 구조
  - 1 단계: 미니배치

    훈련 데이터 중 일부를 무작위로 가져오는 것을 미니배치라고 하며, 미니배치의 손실함수를 줄이는 것을 목표로 함

  - 2 단계: 기울기 산출-> 오차역전파법 등장

    미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구함
    기울기는 손실 함수의 값을 가장 작게 하는 방향 제시

  - 3 단계: 매개변수 갱신

    가중치 매개변수를 기울기 방향으로 조금씩 갱신

  - 4 단계: 반복

    1 ~3 단계 반복

오차역전파법을 적용한 신경망 구현

```python
# two_layer_net.py
import sys,os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from collections import OrderedDict
from common.layers import *
from common.gradient import numerical_gradient


class TwoLayerNet:
    '''2층 신경망 구현'''
    def __init__(self, input_size,
                 hidden_size, output_size, weight_init_std=0.01):
        '''
        초기화 수행
        Params:
            - input_size: 입력층 뉴런 수
            - hidden_size: 은닉층 뉴런 수
            - output_size: 출력층 뉴런 수
            - weight_init_std: 가중치 초기화 시 정규분포의 스케일
        '''
        # 가중치 초기화
        self.params = {
            'W1': weight_init_std * np.random.randn(input_size, hidden_size),
            'b1': np.zeros(hidden_size),
            'W2': weight_init_std * np.random.randn(hidden_size, output_size),
            'b2': np.zeros(output_size)
        }

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

        self.last_layer = SoftmaxWithLoss()
```

오차역전파법을 적용한 신경망 구현

```
def predict(self, x):
    '''예측(추론)
        Pararms:
            - x: 이미지 데이터'''
    for layer in self.layers.values():
        x = layer.forward(x)

    return x

def loss(self, x, t):
    '''
    손실함수의 값을 계산
    Params:
        - x: 이미지데이터, t: 정답 레이블
    '''
    y = self.predict(x)
    return self.last_layer.forward(y, t)

def accuracy(self, x, t):
    '''
    정확도 계산
    Params:
        - x: 이미지 데이터
        - t: 정답 레이블
    '''
    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
```

오차역전파법을 적용한 신경망 구현

```
def numerical_gradient(self, x, t):
    '''
    미분을 통한 가중치 매개변수의 기울기 계산
    Params:
        - x: 이미지 데이터
        - t: 정답 레이블
    '''
    loss_W = lambda W: self.loss(x, t)

    grads = {
        'W1': numerical_gradient(loss_W, self.params['W1']),
        'b1': numerical_gradient(loss_W, self.params['b1']),
        'W2': numerical_gradient(loss_W, self.params['W2']),
        'b2': numerical_gradient(loss_W, self.params['b2'])
    }
    return grads

def gradient(self, x, t):
    # forward
    self.loss(x, t)

    # backward
    dout = 1
    dout = self.last_layer.backward(dout)

    layers = list(self.layers.values())
    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)

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

오차역전파법을 적용한 신경망 구현

- TowLayerNet클래스의 인스턴스 변수

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

오차역전파법을 적용한 신경망 구현

- TowLayerNet클래스의 메서드

메서드		설명
__init__(self, input_size, hidden_size,	1	초기화를 수행한다.
output_size, weight_init_std)	1	인수는 앞에서부터 입력층의 뉴런 수, 은닉층의 뉴런 수, 출력층의 뉴런 수, 가중치 초기화 시 정규분포의 스케일
predict(self, x)	1	예측(추론)을 수행한다.
	2	인수 x는 이미지 데이터
loss(self, x, t)	1	손실 함수의 값을 구한다.
	2	인수 x는 이미지 데이터, t는 정답 레이블 (아래 3가지 메서드의 인수들도 마찬가지)
accuracy(self, x, t)	1	정확도를 구한다.
numerical_gradient(self, x, t)	1	가중치 매개변수의 기울기를 구한다.
gradient(self, x, t)	1	가중치 매개변수의 기울기를 구한다.
	2	numerical_gradient()의 성능 개선 메서드

오차역전파법을 적용한 신경망 구현

- 기울기 확인(gradient check)

In [32]:
```
import sys, os
sys.path.append("./dataset")
import numpy as np
import pickle
from mnist import load_mnist
import matplotlib.pylab as plt
```

In [33]:
```
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)
```

In [34]:
```
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)
```

In [35]:
```
x_batch = x_train[:3]
t_batch = t_train[:3]
```

In [36]:
```
grad_numerical = network.numerical_gradient(x_batch, t_batch)
grad_backprop = network.gradient(x_batch, t_batch)
```

오차역전파법을 적용한 신경망 구현

- 학습 구현

```python
# Train Parameters
iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1
iter_per_epoch = max(train_size / batch_size, 1)

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

for step in range(1, iters_num+1):
    # get mini-batch
    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(압도적으로 빠르다)

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

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

    if step % 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('Step: {:4d}\tTrain acc: {:.5f}\tTest acc: {:.5f}'.format(step,
                                                                        train_acc,
                                                                        test_acc))
```

```
Step:  600	Train acc: 0.90545	Test acc: 0.90930
Step: 1200	Train acc: 0.92533	Test acc: 0.92530
Step: 1800	Train acc: 0.93527	Test acc: 0.93370
Step: 2400	Train acc: 0.94643	Test acc: 0.94560
Step: 3000	Train acc: 0.95393	Test acc: 0.95140
Step: 3600	Train acc: 0.95905	Test acc: 0.95690
Step: 4200	Train acc: 0.96318	Test acc: 0.95980
Step: 4800	Train acc: 0.96613	Test acc: 0.96190
Step: 5400	Train acc: 0.96948	Test acc: 0.96640
Step: 6000	Train acc: 0.97127	Test acc: 0.96630
Step: 6600	Train acc: 0.97338	Test acc: 0.96720
Step: 7200	Train acc: 0.97565	Test acc: 0.97010
Step: 7800	Train acc: 0.97652	Test acc: 0.96990
Step: 8400	Train acc: 0.97768	Test acc: 0.97100
Step: 9000	Train acc: 0.97888	Test acc: 0.97180
Step: 9600	Train acc: 0.97983	Test acc: 0.97220
Optimization finished!
Wall time: 18.1 s
```

오차역전파법을 적용한 신경망 구현

- 결과 그래프

```
# 그래프 그리기
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 5))
markers = {'train': 'o', 'test': 's'}

x_loss = np.arange(len(train_loss_list))
plt.plot(x_loss, train_loss_list)
plt.xlabel("iteration")
plt.ylabel("loss")
plt.show()

plt.figure(figsize=(10, 5))
x_acc = np.arange(len(train_acc_list))
plt.plot(x_acc, train_acc_list, label='train acc')
plt.plot(x_acc, test_acc_list, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()
```