# Backpropagation

Computational Linguistics @ Seoul National University

DL from Scratch

By Hyopil Shin

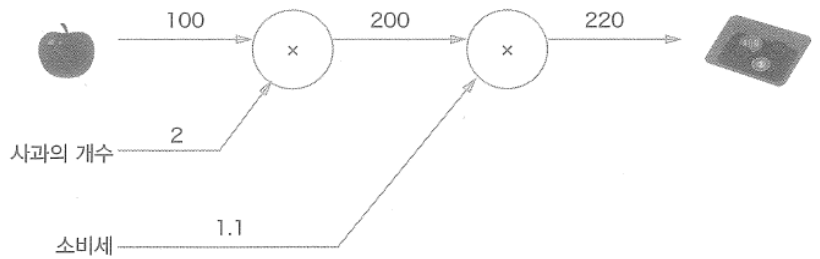
# Computational Graph

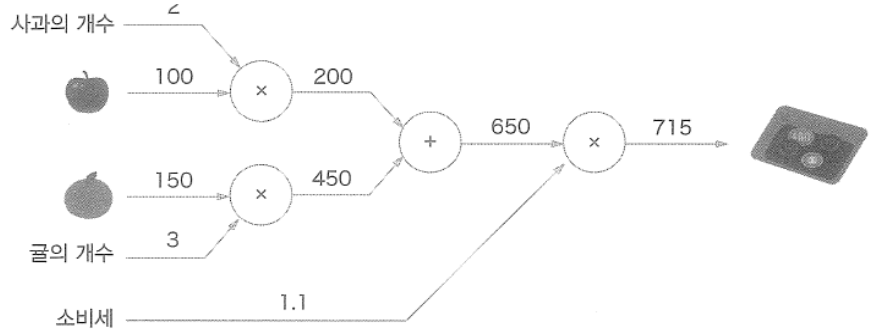
- 수치미분을 통해 신경망의 가중치 매개변수의 기울기를 구함
  - 단순하고 구현하기 쉽지만 계산 시간이 오래 걸림
  - Backpropagation을 통해 가중치 매개변수의 기울기를 효율적으로 계산
- Computational graph를 도입

- Computational graph-계산과정을 그래프로 나타낸 것
  - Node, edge

  - 순전파(forward propagation)
    - 계산 그래프에서 계산을 왼쪽에서 오른쪽으로 진행
  - 역전파(backward Propagation)

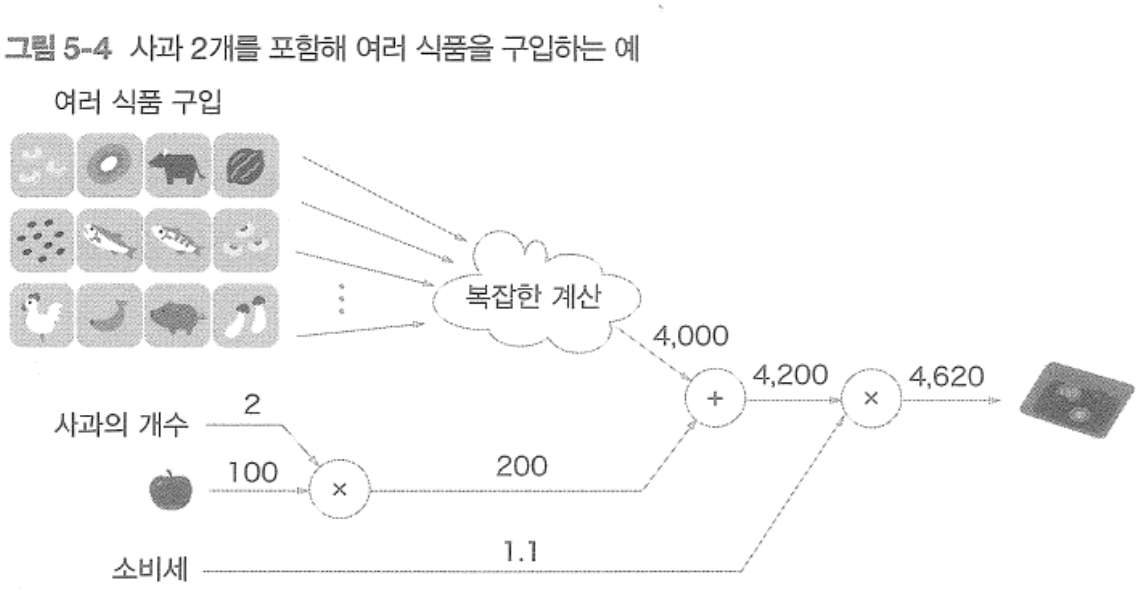
그림 5-2 계산 그래프로 풀어본 문제 1의 답 : '사과의 개수'와 '소비세'를 변수로 취급해 원 밖에 표기



# Computational Graph

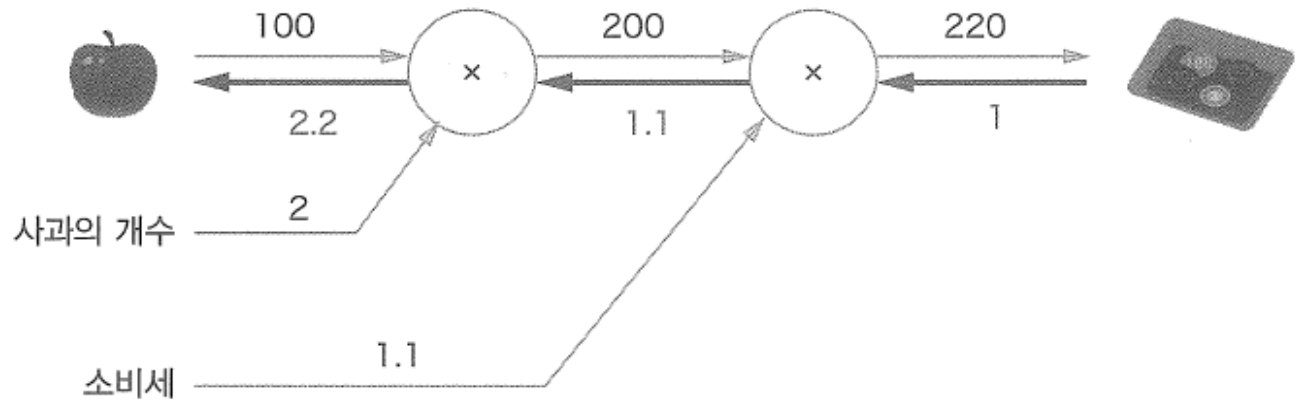
- 국소적 계산
  - 자신과 관계된 정보만으로 그 후의 결과를 출력
  - 전체 계산이 복잡해도 각 노드에서는 단순한 계산에 집중하여 문제를 단순화
  - 중간 계산 결과를 보관
  - 역전파를 통한 미분의 효율적 계산 가능



# Computational Graph

- 사과 가격(x)이 오르면 최종 금액(L)에 어떤 영향을 끼치는지?
  - '사과 가격에 대한 지불 금액의 미분 ' $\frac{\partial L}{\partial x}$

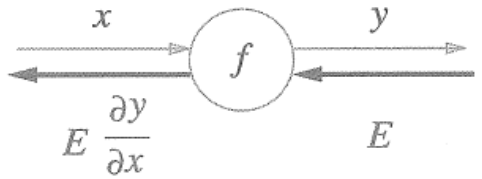
그림 5-5 역전파에 의한 미분 값의 전달



# Computational Graph

- 역전파는 국소적인 미분을 순방향과 반대인 오른쪽에서 왼쪽으로 전달
- Chain Rule
  - *y=f(x)* 의 역전파

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




  - 합성함수 미분은 합성 함수를 구성하는 각 함수의 미분의 곱으로 나타낼 수 있다.
    - $Z = (x+y)^2$

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

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

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

$$\frac{\partial z}{\partial t} = 2t$$
$$\frac{\partial t}{\partial x} = 1$$

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

# Computational Graph

- Chain rule and computational graph
  - 'x에 대한 z의 미분'

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

그림 5-7 [식 5.4]의 계산 그래프 : 순전파와는 반대 방향으로 국소적 미분을 곱하여 전달한다.

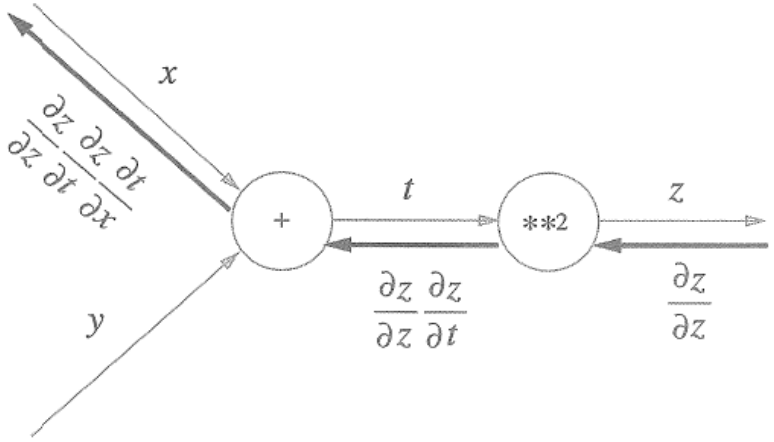
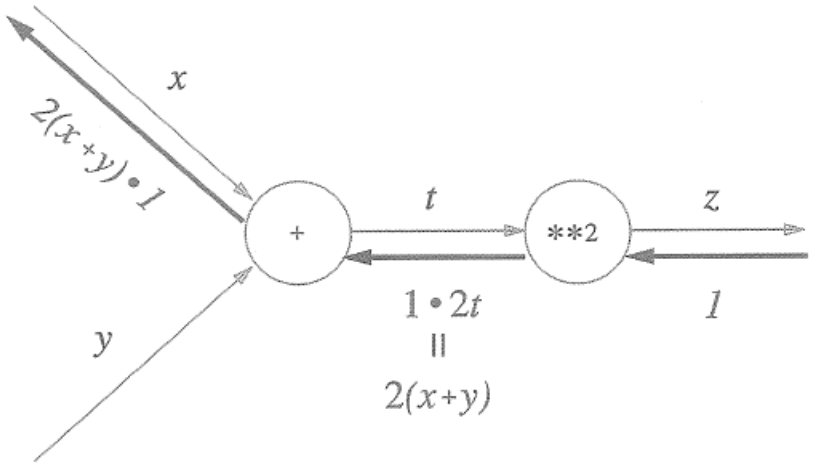



그림 5-8 계산 그래프의 역전파 결과에 따르면 $\frac{\partial z}{\partial x}$는 $2(x+y)$가 된다.



# Backpropagation

- 덧셈 노드의 역전파
  - $z = x + y$

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

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

  - 덧셈 노드의 역전파는 1을 곱하기만 할 뿐 입력된 값을 그대로 다음 노드로 보냄

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

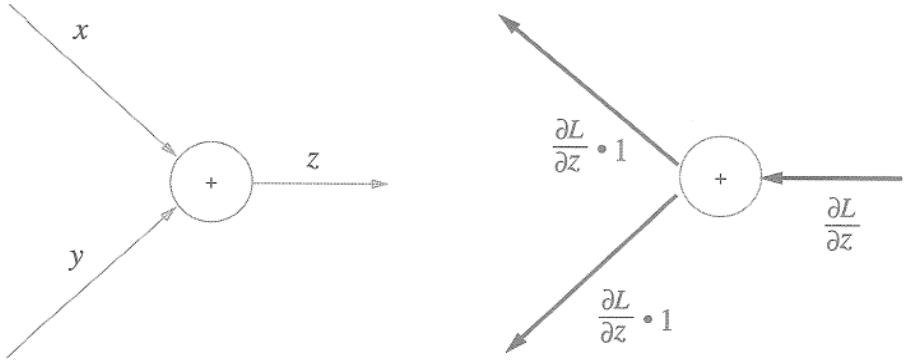
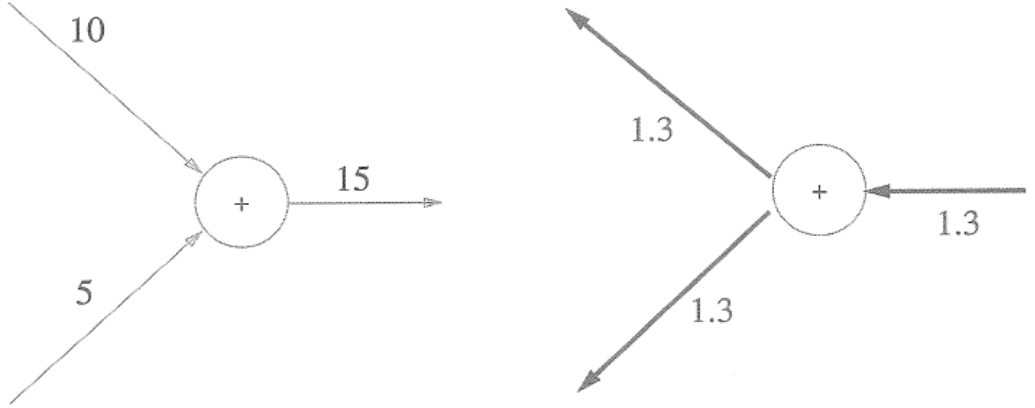



그림 5-11 덧셈 노드 역전파의 구체적인 예



# Backpropagation

- 곱셈노드의 역전파

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

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

  - $z = xy$
  - 상류의 값에 순전파 때의 입력신호들을 서로 바꾼 값을 곱해서 하류로 보냄

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

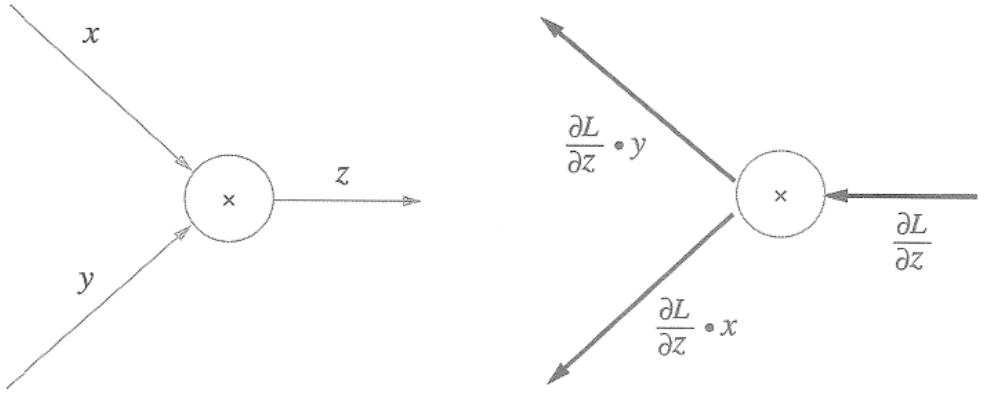
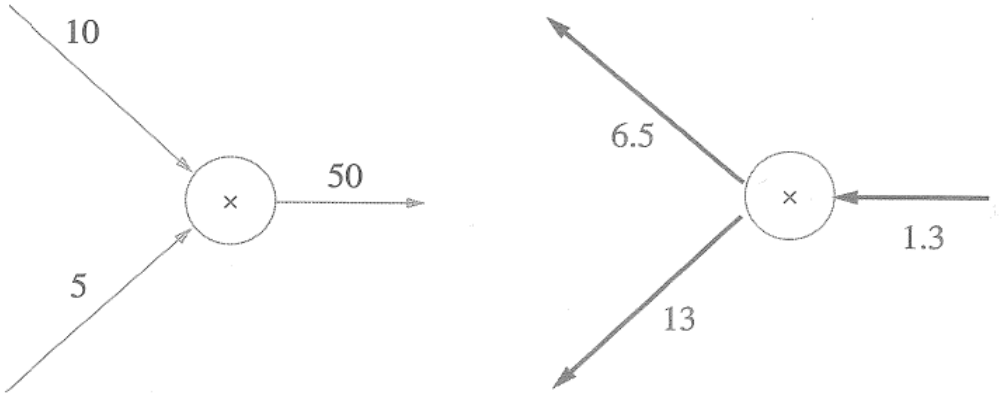
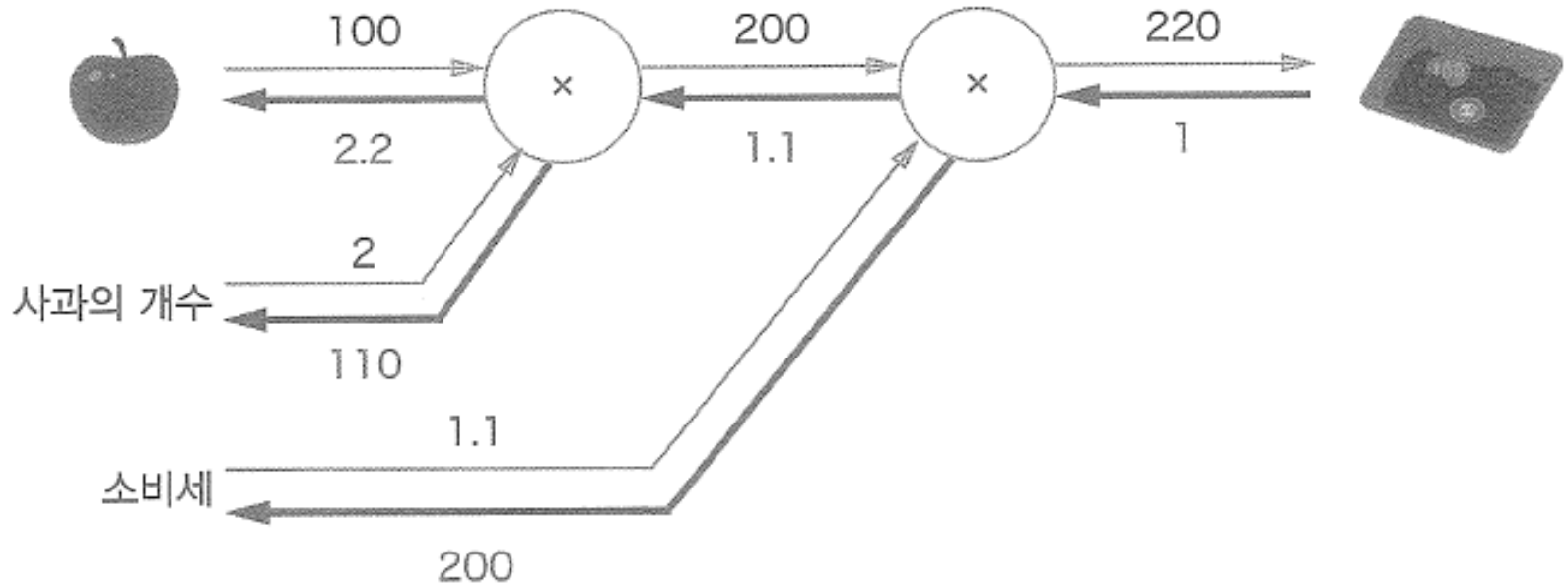



그림 5-13 곱셈 노드 역전파의 구체적인 예



# Backpropagation

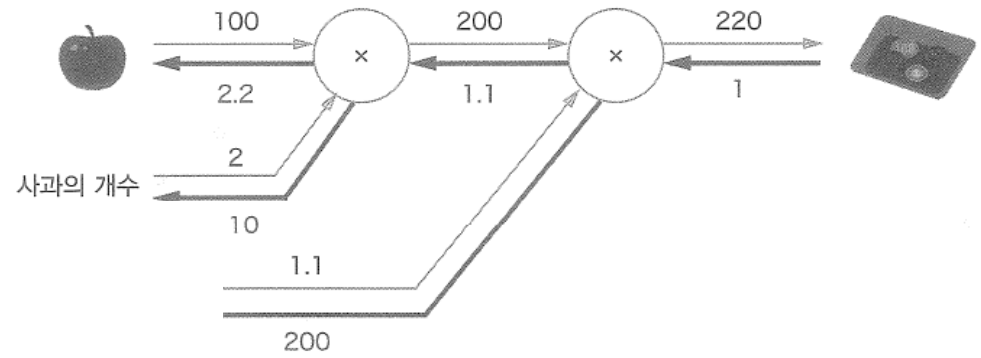
그림 5-14 사과 쇼핑의 역전파 예



# Backpropagation

- layer_native.py
- buy_apple.py

그림 5-16 사과 2개 구입




```
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
```

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

print(dapple, dapple_num, dtax) # 2.2 110 200
```

# Backpropagation

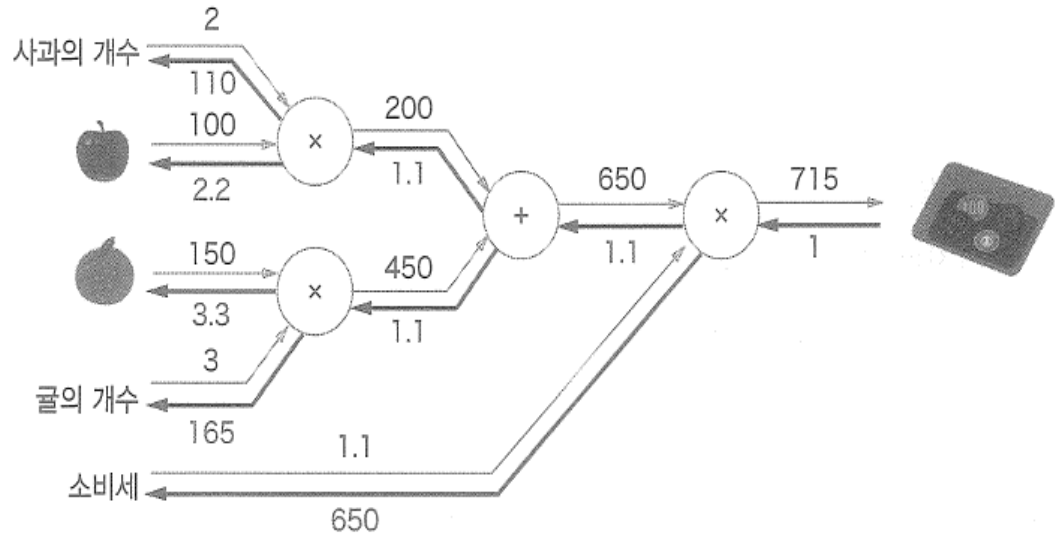
- 덧셈계층

```
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
```

# Backpropagation

- buy_apple_orange.py

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
```

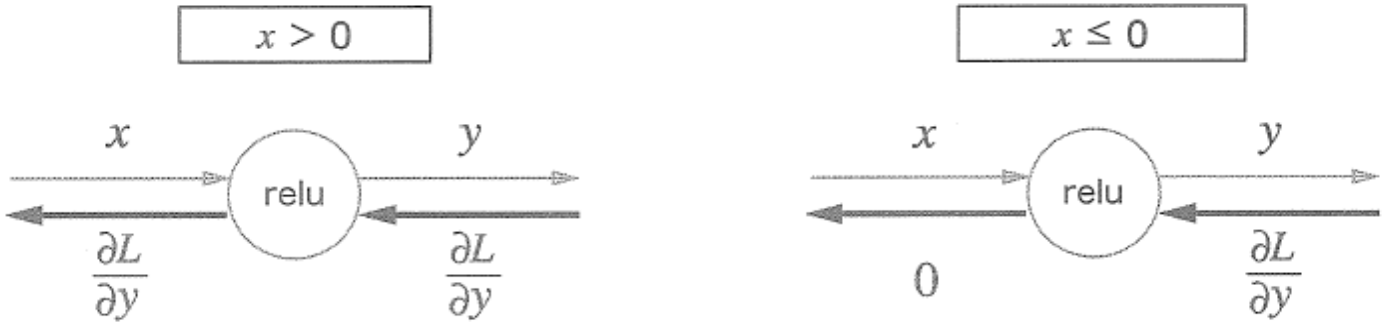
# Activation Function

- ReLU
  - layers.py

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

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

그림 5-18 ReLU 계층의 계산 그래프

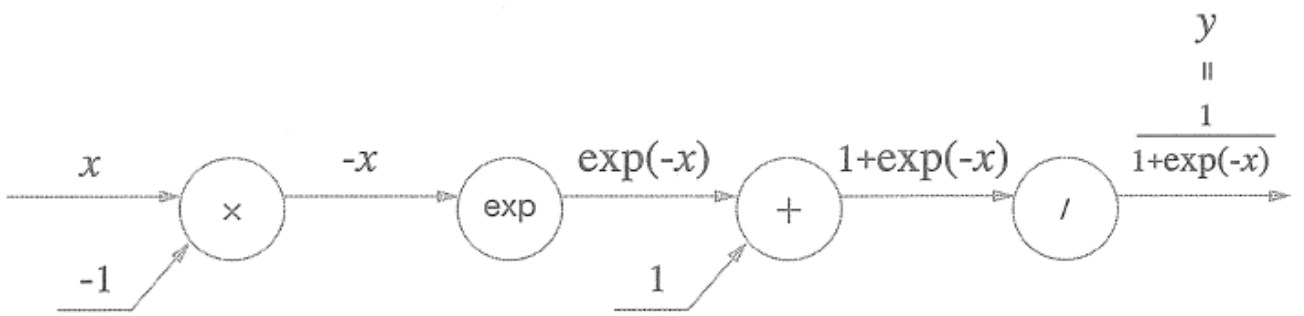


# Activation Function

- Sigmoid

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

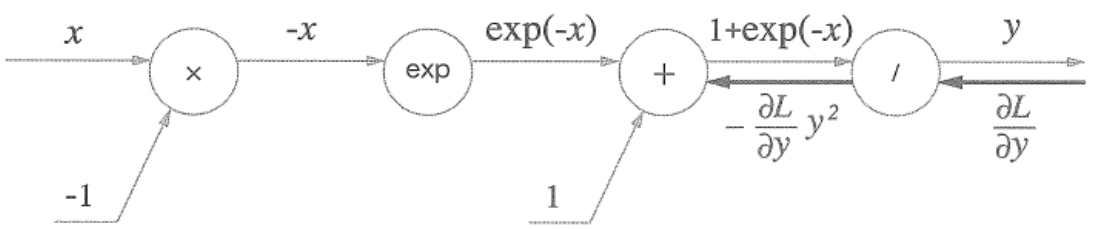
그림 5-19 Sigmoid 계층의 계산 그래프(순전파)




- 1단계

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

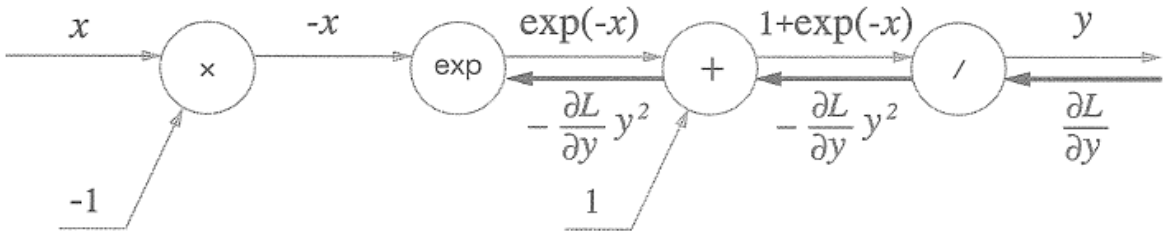
$$= -y^2$$



# Activation Function

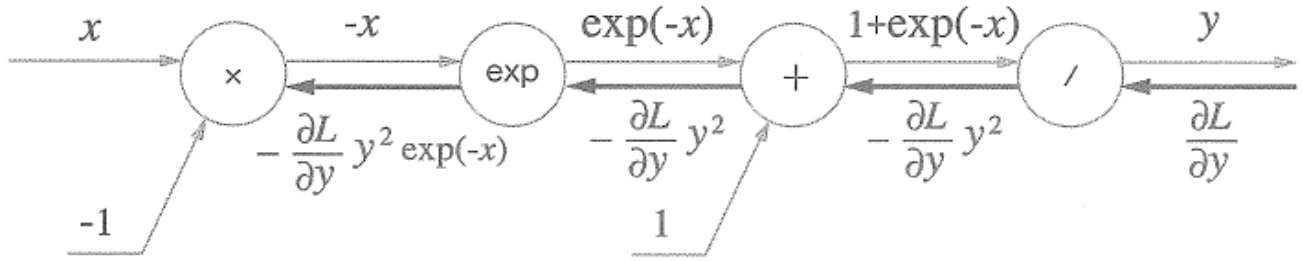
- 2단계

'+' 노드는 상류의 값을 여과 없이 하류로 내보내는 게 다입니다. 계산 그래프에서는
습니다.




- 3단계

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



# Activation Function

- 4단계

그림 5-20 Sigmoid 계층의 계산 그래프

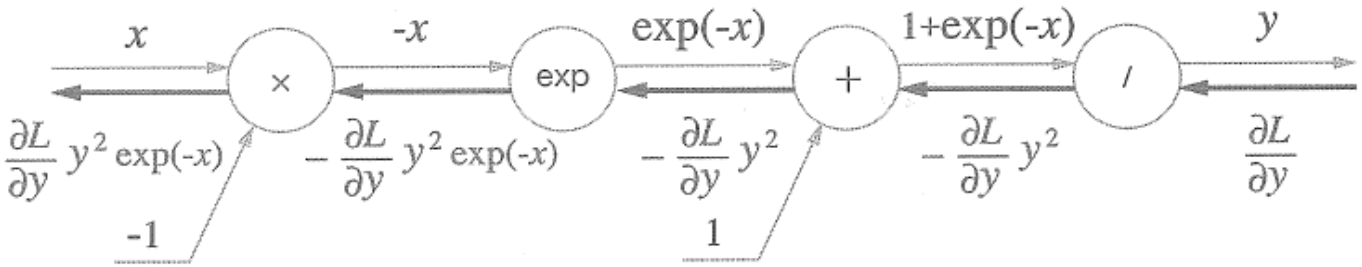
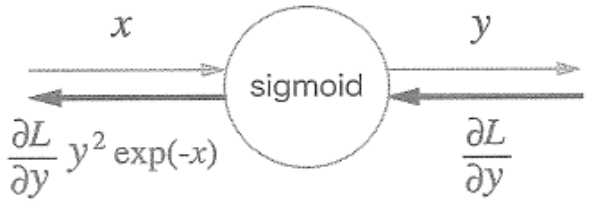
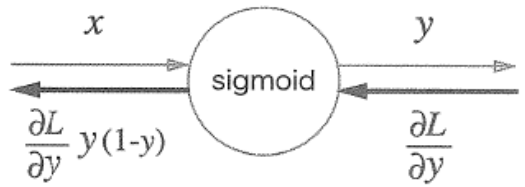



그림 5-21 Sigmoid 계층의 계산 그래프(간소화 버전)




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



# Affine Layers

- 신경망의 순전파 때 수행하는 행렬의 내적을 기하학에서는 affine transformation이라 함

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

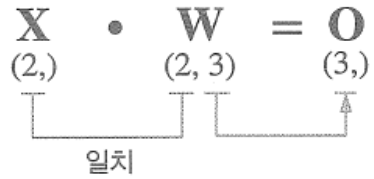
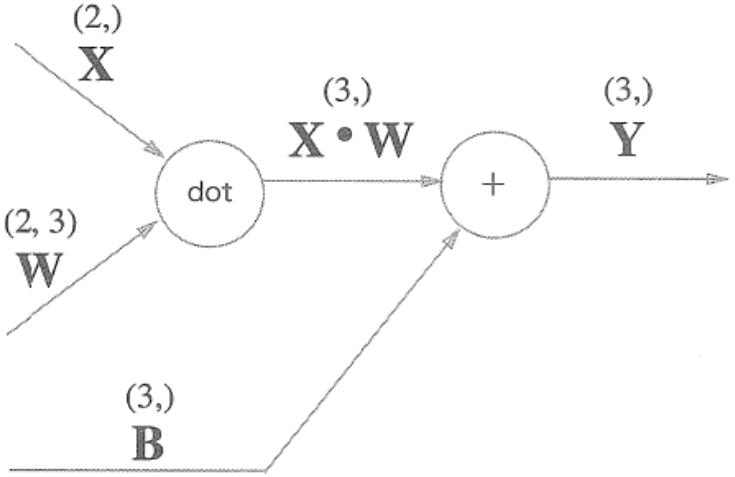



그림 5-24 Affine 계층의 계산 그래프 : 변수가 행렬임에 주의. 각 변수의 형상을 변수명 위에 표기했다.



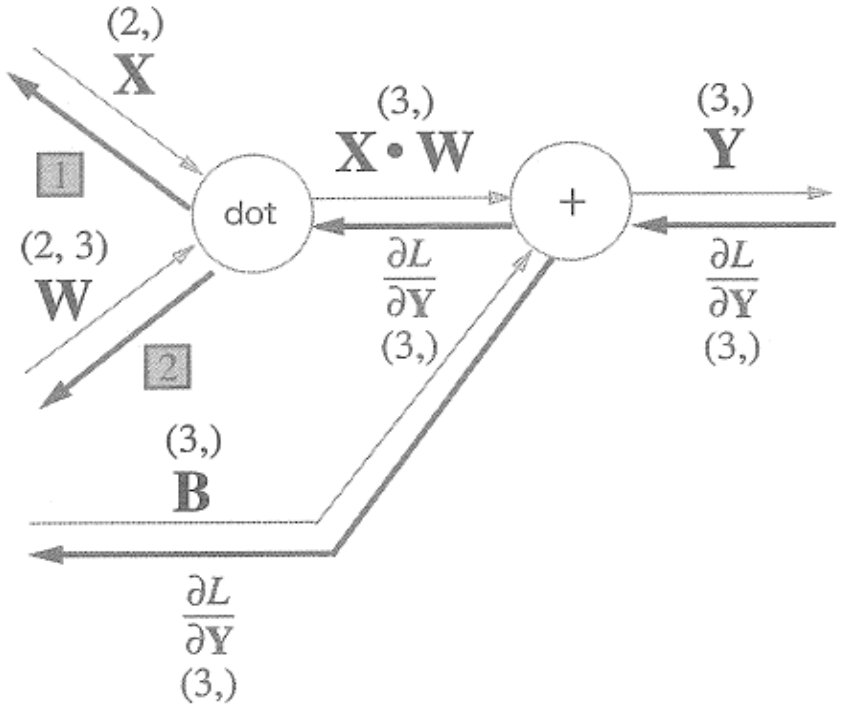
# Affine Layers

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

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

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

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




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

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

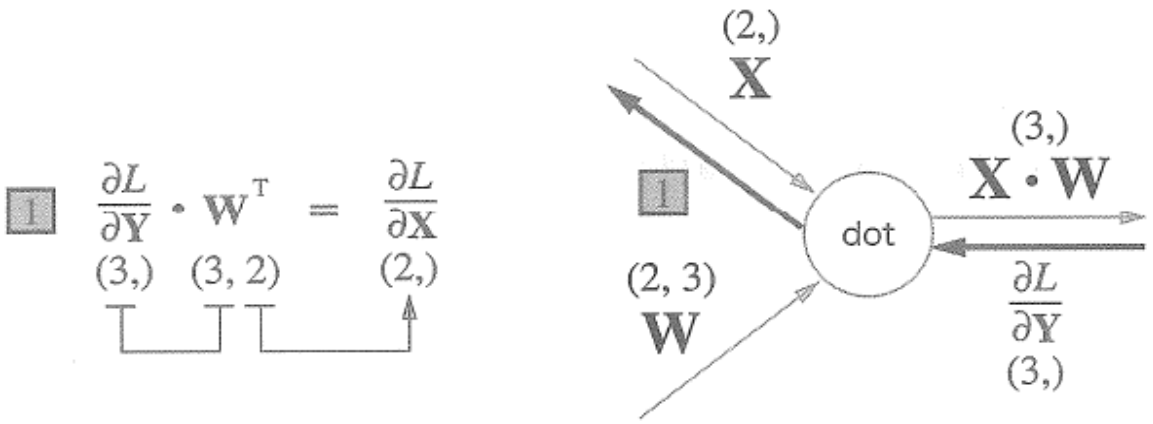
$$\mathbf{X} = (x_0, x_1, \cdots, x_n)$$

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

# Affine Layers

- 행렬의 형상에 주의- 행렬의 내적에서는 대응하는 차원의 원소 수를 일치시켜야 함

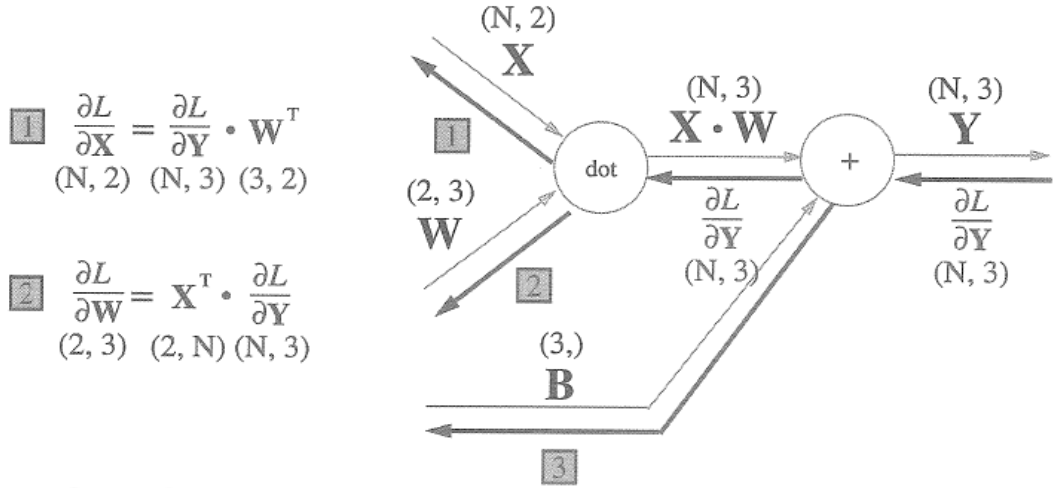
그림 5-26 행렬 내적('dot' 노드)의 역전파는 행렬의 대응하는 차원의 원소 수가 일치하도록 내적을 조립하여 구할 수 있다.



# Batch Affine Layers

N개의 배치의 경우

layers.py




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(제0축, 열방향)의 합
(3) (N, 3)

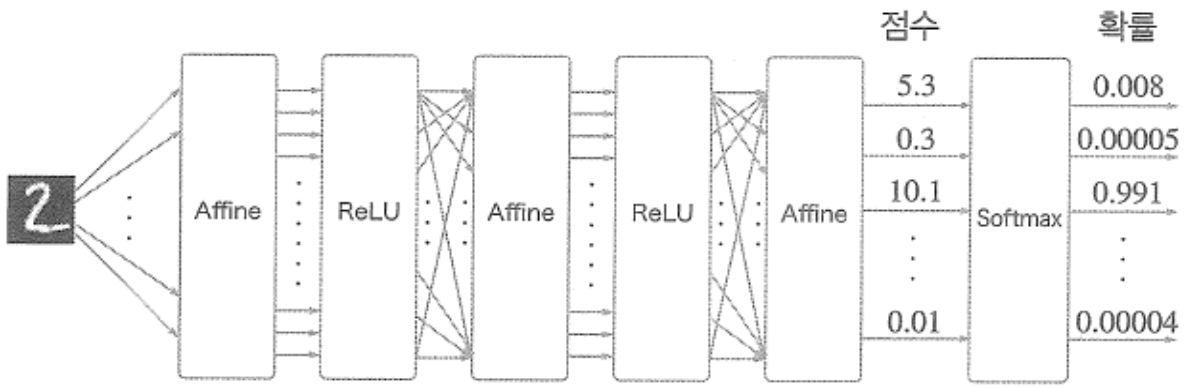
편향의 경우 주의
순전파/역전파

```
>>> 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
```

```
>>> 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])
```

# Softmax-with-Loss Layers

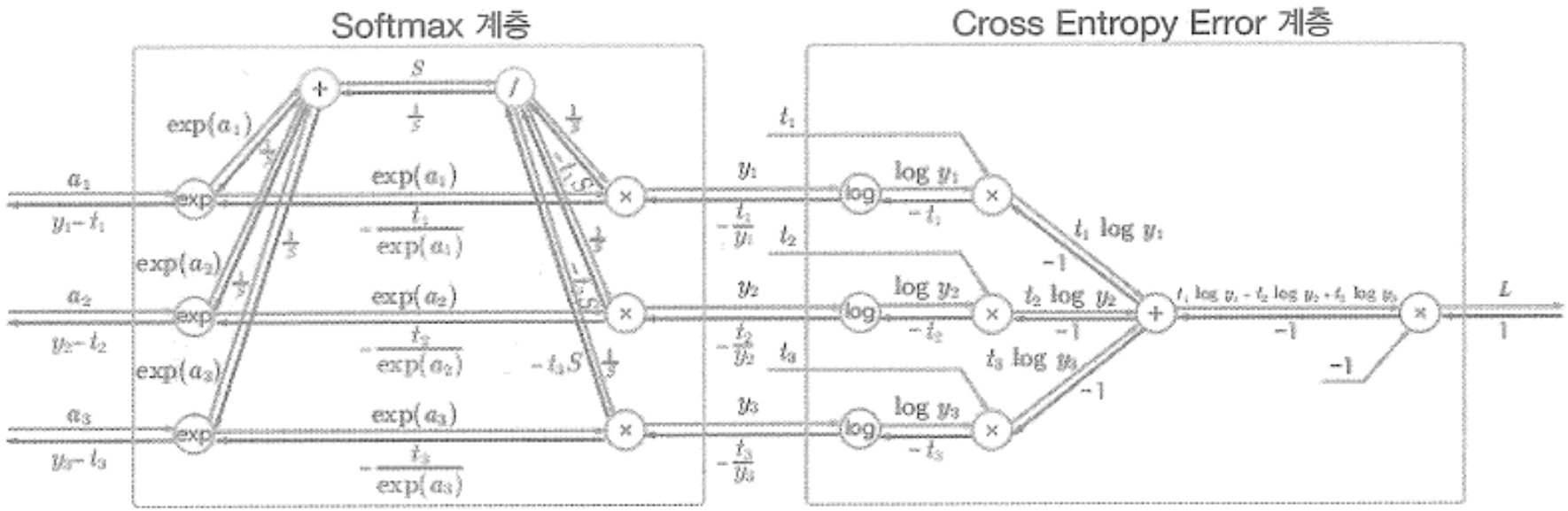
**그림 5-28** 입력 이미지가 Affine 계층과 ReLU 계층을 통과하며 변환되고, 마지막 Softmax 계층에 의해서 10개의 입력이 정규화된다. 이 그림에서는 숫자 '0'의 점수는 5.3이며, 이것이 Softmax 계층에 의해서 0.008(0.8%)로 변환된다. 또, '2'의 점수는 10.1에서 0.991(99.1%)로 변환된다.




**NOTE_** 신경망에서 수행하는 작업은 **학습**과 **추론** 두 가지가 있습니다. 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.

# Softmax-with-Loss Layers

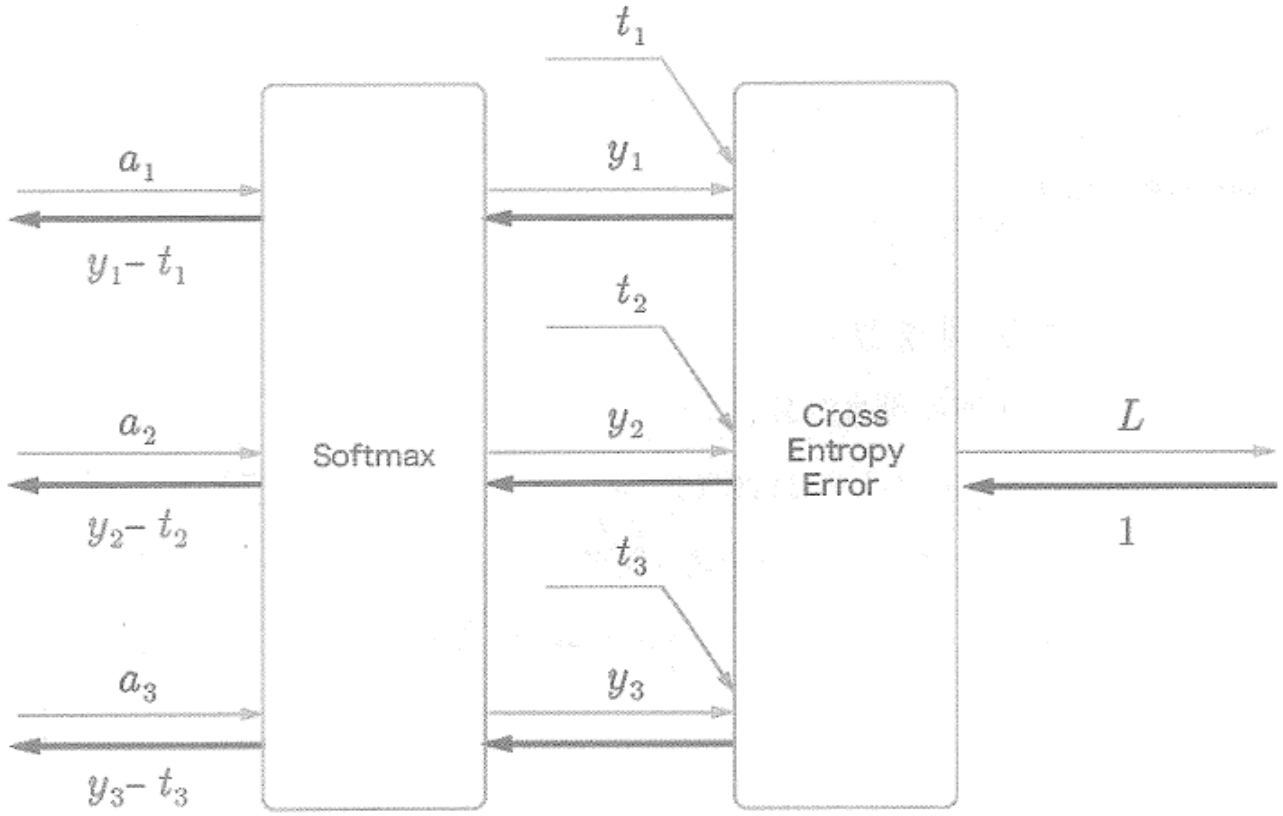
그림 5-29 Softmax-with-Loss 계층의 계산 그래프



# Softmax-with-Loss Layers

- Simplified version

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



# Softmax-with-Loss Layers

```
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
```

# Backpropagation Implementation

- 신경망 학습

**전제**

신경망에는 적응 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 '학습'이라 합니다. 신경망 학습은 다음과 같이 4단계로 수행합니다.

**1단계 – 미니배치**

훈련 데이터 중 일부를 무작위로 가져옵니다. 이렇게 선별한 데이터를 미니배치라 하며, 그 미니배치의 손실 함수 값을 줄이는 것을 목표로 한다.

**2단계 – 기울기 산출**

미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구합니다. 기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시합니다.

**3단계 – 매개변수 갱신**

가중치 매개변수를 기울기 방향으로 아주 조금 갱신합니다.

**4단계 – 반복**

1~3단계를 반복합니다.

# Backpropagation Implementation

- 05-train_neuralnet.py

| 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. |