

## Chap05 - 오차역전파법 Backpropagation

### 파일 설명

| 파일명                 | 파일 용도   | 관련 절                       | 페이지      |
|---------------------|---|----------------------------|----------|
| buy_apple.py        | 사과 2개를 구입하는 예제의 순전파와 역전파 구현입니다.                               | 5.4.1 곱셈 계층                | 162      |
| buy_apple_orange.py | 사과와 오렌지를 구입하는 예제의 순전파와 역전파 구현입니다.                             | 5.4.2 덧셈 계층                | 163      |
| gradient_check.py   | 수치 미분 방식과 비교하여 오차역전파법으로 구한 기울기를 검증합니다(기울기 확인).                | 5.7.3 오차역전파법으로 구한 기울기 검증하기 | 184      |
| layer_naive.py      | 곱셈 계층과 덧셈 계층의 구현입니다.  | 5.4.1 곱셈 계층 / 5.4.2 덧셈 계층  | 161, 163 |
| train_neuralnet.py  | 4장의 train_neuralnet.py와 같습니다. 단, 수치 미분 대신 오차역전파법으로 기울기를 구합니다. | 5.7.4 오차역전파법을 사용한 학습 구현하기  | 186      |
| two_layer_net.py    | 오차역전파법을 적용한 2층 신경망 클래스  | 5.7.2 오차역전파법을 적용한 신경망 구현하기 | 181      |

앞 장에서는 신경망 학습에 대해서 설명했습니다. 그때 신경망의 가중치 매개변수의 기울기(정확히는 가중치 매개변수에 대한 손실 함수의 기울기)는 수치 미분을 사용해 구했습니다. 수치 미분은 단순하고 구현하기도 쉽지만 계산 시간이 오래 걸린다는 게 단점입니다. 이번 장에서는 가중치 매개변수의 기울기를 효율적으로 계산하는 ‘오차역전파법 backpropagation’을 배워보겠습니다. 오차역전파법을 제대로 이해하는 방법은 두 가지가 있을 것입니다. 하나는 수식을 통한 것이고, 다른 하나는 계산 그래프를 통한 것입니다. 전자 쪽이 일반적인 방법으로, 특히 기계학습을 다루는 책 대부분은 수식을 중심으로 이야기를 전개합니다. 확실히 수식을 사용한 설명은 정확하고 간결하므로 올바른 방법이라 할 수 있겠죠. 하지만 졸업 후 너무 오랜만에 수식을 중심으로 생각하다 보면 본질을 놓치거나, 수많은 수식에 당황하는 일이 벌어지기도 합니다. 그래서 이번 장에서는 계산 그래프를 사용해서 ‘시각적’으로 이해시켜드리겠습니다. 그런 다음 실제로 코드를 작성해보면 ‘과연!’이란 탄성과 함께 더 깊이 이해하게 될 것입니다.

웁긴이\_ 오차역전파법을 풀어쓰면 ‘오차를 역(반대 방향)으로 전파하는 방법(backward propagation of errors)’입니다. 너무 길고 쓸데없이 어려운 느낌이라 줄여서 ‘역전파법’ 혹은 그냥 ‘역전파’라고 쓰기도 합니다.

## Copyrights

1. <https://github.com/WegraLee/deep-learning-from-scratch> (<https://github.com/WegraLee/deep-learning-from-scratch>)
2. <https://github.com/SDRLurker/deep-learning> (<https://github.com/SDRLurker/deep-learning>) (chapter4) (<https://nbviewer.jupyter.org/github/SDRLurker/deep-learning/blob/master/4%EC%9E%A5.ipynb>)
3. <https://github.com/ExcelsiorCJH/DLFromScratch> (<https://github.com/ExcelsiorCJH/DLFromScratch>) (chapter4) ([https://nbviewer.jupyter.org/github/ExcelsiorCJH/DLFromScratch/blob/master/Chap04-Neural Network Traing/Chap04-Neural Network Training.ipynb](https://nbviewer.jupyter.org/github/ExcelsiorCJH/DLFromScratch/blob/master/Chap04-Neural%20Network%20Traing/Chap04-Neural%20Network%20Training.ipynb))

Customized by Gil-Jin Jang, April 1, 2021

## 목차

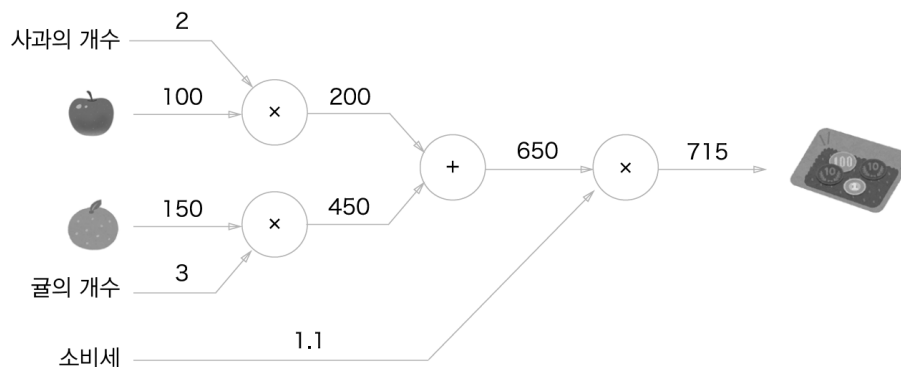
- 5.1 계산 그래프
  - \_\_5.1.1 계산 그래프로 풀다
  - \_\_5.1.2 국소적 계산
  - \_\_5.1.3 왜 계산 그래프로 푸는가?
- 5.2 연쇄법칙
  - \_\_5.2.1 계산 그래프에서의 역전파
  - \_\_5.2.2 연쇄법칙이란?
  - \_\_5.2.3 연쇄법칙과 계산 그래프
- 5.3 역전파
  - \_\_5.3.1 덧셈 노드의 역전파
  - \_\_5.3.2 곱셈 노드의 역전파
  - \_\_5.3.3 사과 쇼핑의 예
- 5.4 단순한 계층 구현하기
  - \_\_5.4.1 곱셈 계층
  - \_\_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

## 5.1 계산 그래프

**계산 그래프(computational graph)**는 계산 과정을 그래프로 나타낸 것이며, **노드(node)**와 **엣지(edge)**로 표현된다. 노드는 연산을 정의하며, 엣지는 데이터가 흘러가는 방향을 나타낸다.

### 5.1.1 계산 그래프로 풀다

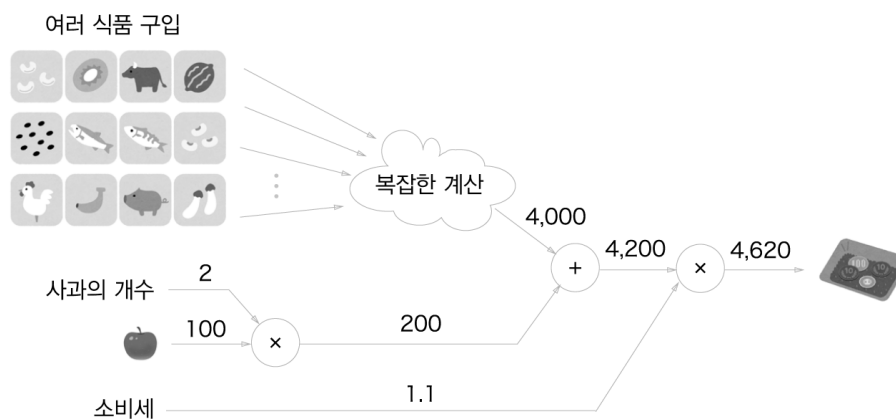
현빈 군은 슈퍼에서 사과를 2개, 귤을 3개 샀습니다. 사과는 1개에 100원, 귤은 1개 150원입니다. 소비세가 10%일 때 지불 금액을 구하라.



1. 계산 그래프를 구성한다.
2. 그래프에서 계산을 왼쪽에서 오른쪽으로 진행한다. → **순전파**(forward propagation)

### 5.1.2 국소적 계산

계산 그래프의 특징은 '국소적 계산'을 통해 최종 결과를 얻는 것이다. 즉, '자신과 직접 관계된' 범위 내에서만 계산이 이루어진다.

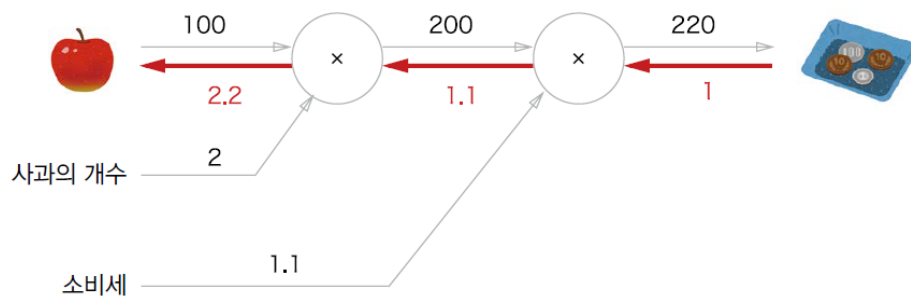


### 5.1.3 왜 계산 그래프로 푸는가?

계산그래프의 장점은 다음과 같다.

- 국소적 계산을 통해 각 노드의 계산에 집중하여 문제를 단순화할 수 있다.
- 역전파를 통해 '미분'을 효율적으로 계산할 수 있다.

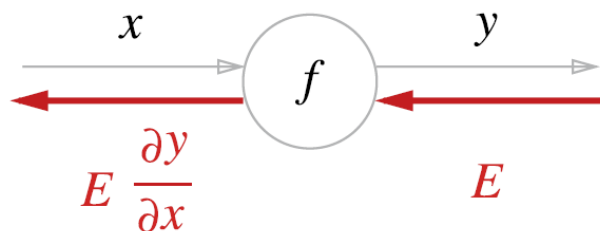
'사과 가격이 오르면 최종 금액에 어떠한 영향을 주는가'에 대해서 **사과 가격에 대한 지불 금액의 미분**을 구해 계산할 수 있다. 사과의 값을  $x$ , 지불 금액을  $L$ 라 했을 때,  $\frac{\partial L}{\partial x}$ 를 구하는 것이다. 이러한 미분 값은 사과 값( $x$ )가 '아주 조금'올랐을 때 지불 금액( $L$ )이 얼마나 증가하는지를 나타낸다.



## 5.2 연쇄 법칙 - Chain Rule

### 5.2.1 계산 그래프의 역전파

먼저, 역전파 계산 예제로  $y = f(x)$ 의 역전파를 계산해보자.



위의 그림에서 처럼 역전파 계산 순서는 신호  $E$ 에 노드( $f$ )의 국소적 미분  $\left(\frac{\partial y}{\partial x}\right)$ 을 곱한 후 엣지(edge)를 통해 다음 노드로 전달하는 것이다. 여기서 국소적 미분은 순전파 때의  $y = f(x)$ 에 대한 미분을 구하는 것이고, 이것은  $x$ 에 대한  $y$ 의 미분  $\left(\frac{\partial y}{\partial x}\right)$ 을 구한다는 의미이다.

### 5.2.2 연쇄법칙이란?

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

$$t = x + y$$

$$z = t^2$$

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

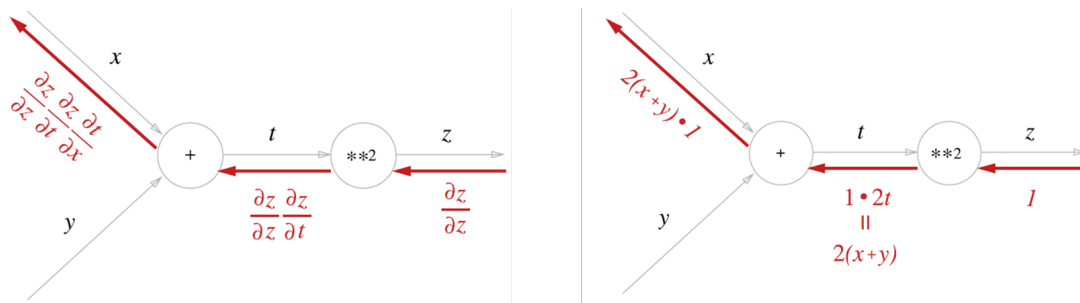
$$\frac{\partial z}{\partial t} = 2t$$

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

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

### 5.2.3 연쇄법칙과 계산 그래프

5.2.2의 예제 식을 계산 그래프로 나타내면 다음과 같다.



### 5.3 역전파

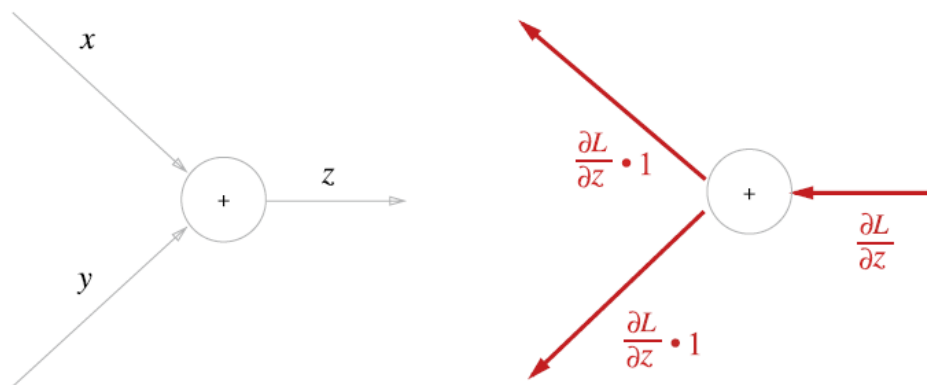
### 5.3.1 덧셈 노드의 역전파

$$z = x + y$$

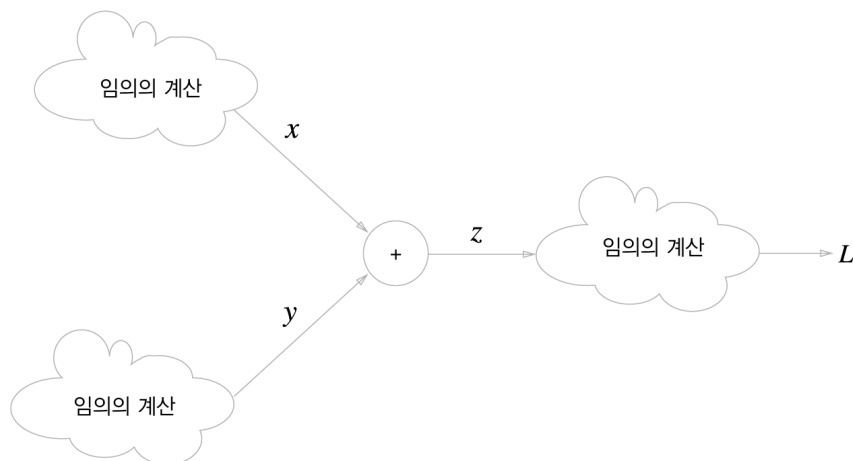
$$\frac{\partial z}{\partial x} = 1$$

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

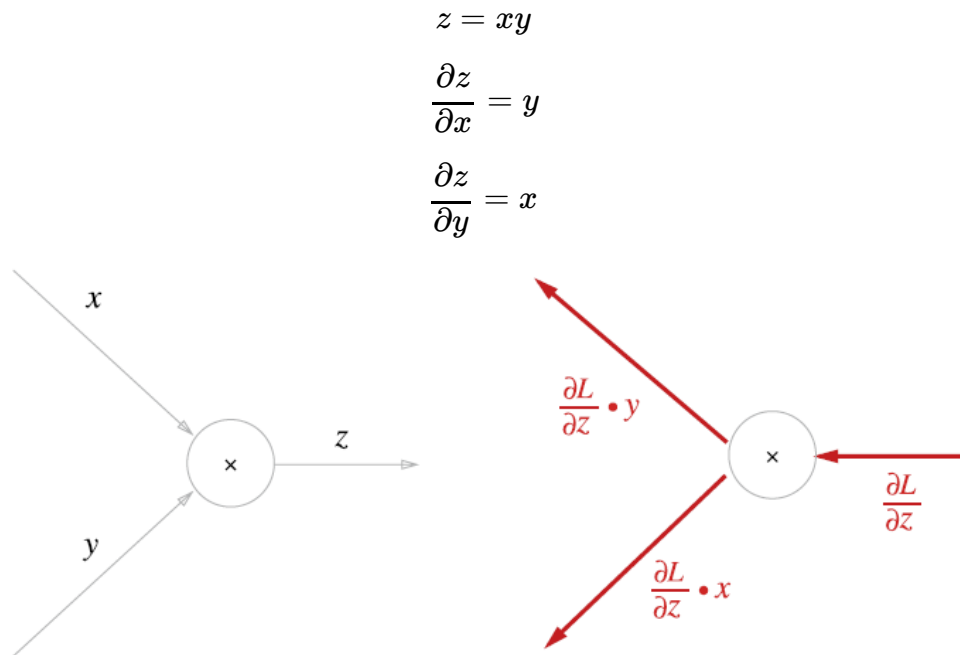
덧셈 노드의 역전파는 입력값을 그대로 흘려보낸다.



위의  $z = x + y$  계산은 큰 계산 그래프의 중간 어딘가에 존재한다고 가정했기 때문에, 이 계산 그래프의 앞부분(상류)에서부터  $\frac{\partial L}{\partial z}$ 가 전해졌다고 가정한다.



### 5.3.2 곱셈 노드의 역전파



## 5.4 단순한 계층 구현하기

### 5.4.1 곱셈 계층

forward()는 순전파, backward()는 역전파이다.

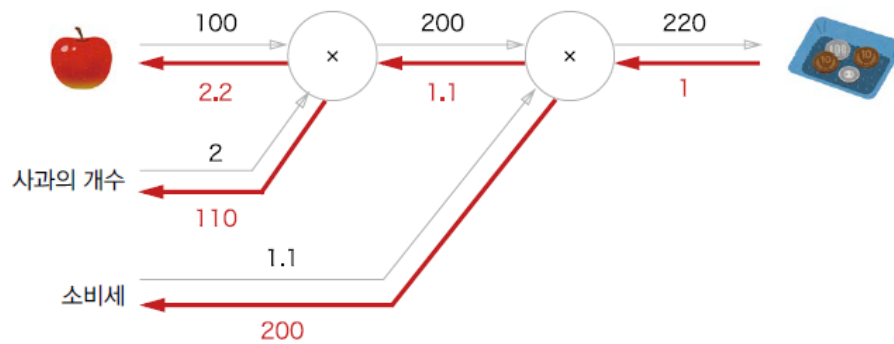
```
In [1]: class MulLayer:
    def __init__(self):
        self.x = None
        self.y = None

    def forward(self, x, y):
        self.x = x
        self.y = y
        out = x * y

        return out

    def backward(self, dout):
        dx = dout * self.y # x와 y를 바꾼다.
        dy = dout * self.x

        return dx, dy
```



```
In [2]: 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('%d' % price)

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

print("%.1f, %d, %d" % (dapple, dapple_num, dtax))

220
2.2, 110, 200
```

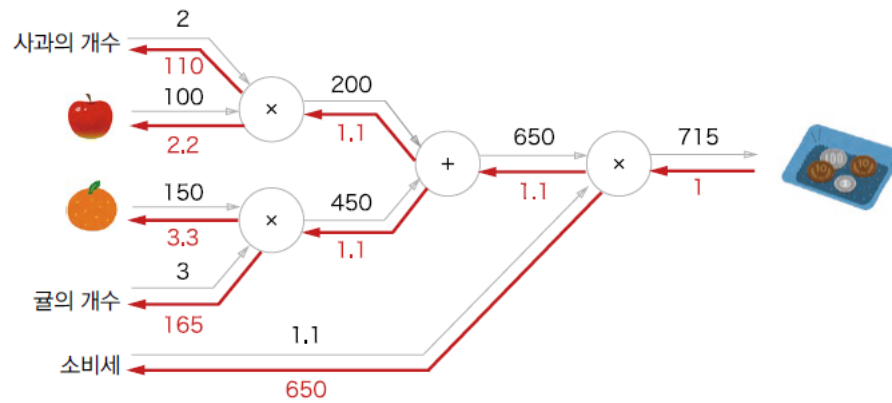
### 5.4.2 덧셈 계층

```
In [3]: 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 [4]: 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)
orange_price = mul_orange_layer.forward(orange, orange_num)
all_price = add_apple_orange_layer.forward(apple_price, orange_price)
price = mul_tax_layer.forward(all_price, tax)

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

print('%d' % price)
print("%d, %.1f, %.1f, %d, %d" % (dapple_num, dapple, dorange, dorange_num, dtax))
```

```
715
110, 2.2, 3.3, 165, 650
```

## 5.5 활성화 함수 계층 구현하기

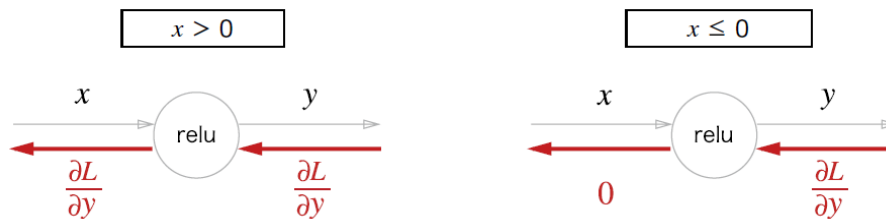
### 5.5.1 ReLU 계층

ReLU의 식은 다음과 같다.

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

위의 식에서  $x$ 에 대한  $y$ 의 미분은 아래와 같다.

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



In [6]: `import numpy as np`

```
In [5]: # common/layers.py

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
```

위의 Relu 클래스에서 mask 인스턴스 변수는 아래와 같이 True/False로 구성되는 NumPy 배열이다.

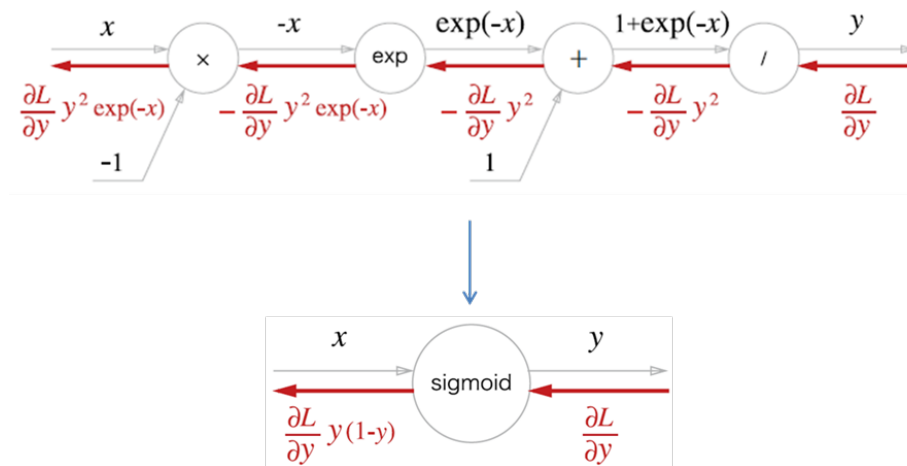
```
In [12]: x = np.array([[1., -0.5],
                        [-2., 3.]])
print('x:\n', x)

mask = (x <= 0)
print('mask:\n', mask)
```

```
x:
[[ 1. -0.5]
 [-2.  3. ]]
mask:
[[False  True]
 [ True False]]
```

### 5.5.2 Sigmoid 계층

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



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

```
In [13]: # common/layers
class Sigmoid:
    def __init__(self):
        self.out = None

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

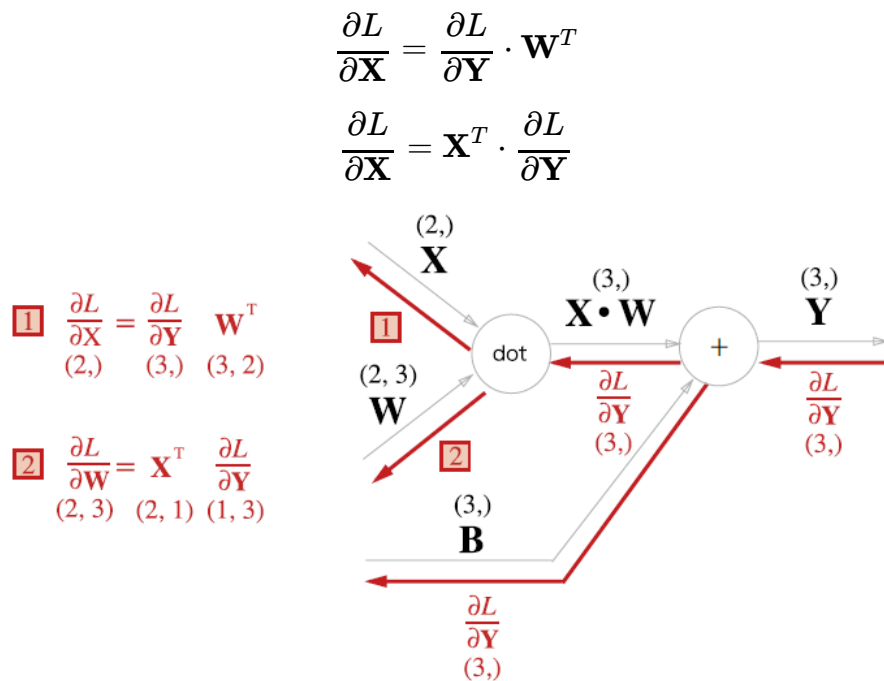
        return out

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

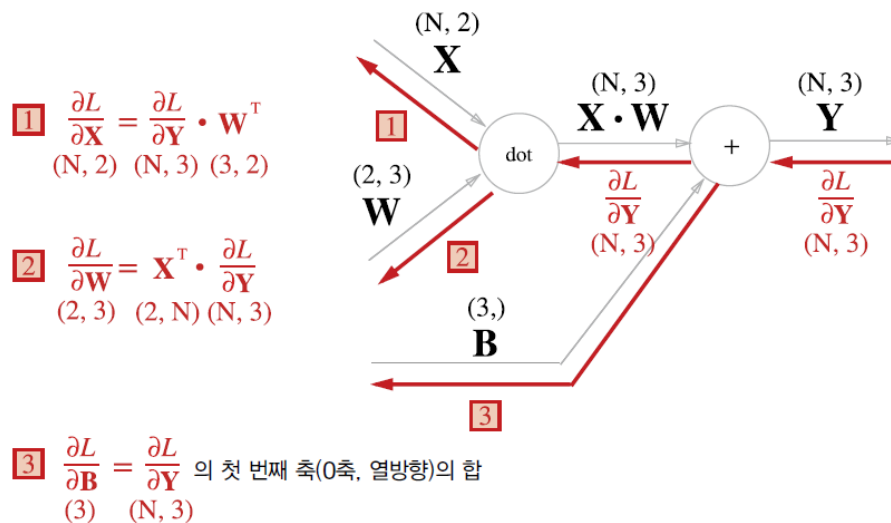
        return dx
```

## 5.6 Affine/Softmax 계층 구현하기

### 5.6.1 Affine 계층



## 5.6.2 배치용 Affine 계층



In [14]: # common/layers.py

```

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

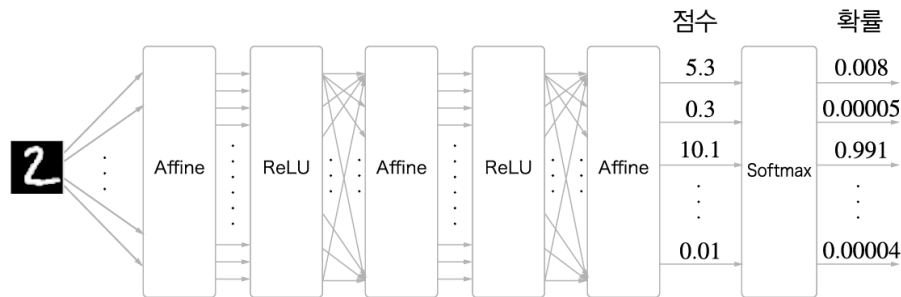
    def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)

        return dx

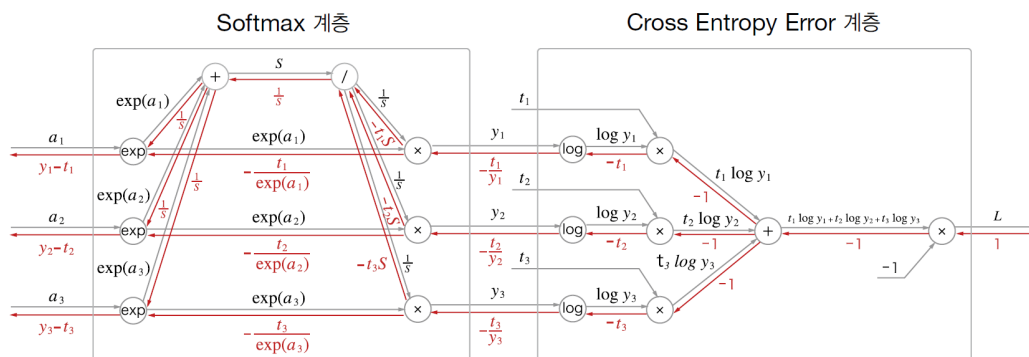
```

### 5.6.3 Softmax-with-Loss 계층

딥러닝에서는 **학습**과 **추론** 두 가지가 있다. 일반적으로 추론일 때는 **Softmax** 계층(layer)을 사용하지 않는다. Softmax 계층 앞의 Affine 계층의 출력을 **점수(score)**라고 하는데, 딥러닝의 추론에서는 답을 하나만 예측하는 경우에는 가장 높은 점수만 알면 되므로 Softmax 계층이 필요없다. 반면, 딥러닝을 **학습**할 때는 Softmax 계층이 필요하다.



소프트맥스(softmax) 계층을 구현할때, 손실함수인 교차 엔트로피 오차(cross entropy error)도 포함하여 아래의 그림과 같이 **Softmax-with-Loss** 계층을 구현한다.



```
In [17]: # common/layers.py
import os, sys
sys.path.append(os.pardir) # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from common.functions import softmax, cross_entropy_error

class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None # 손실
        self.y = None # softmax의 출력
        self.t = None # 정답 레이블(one-hot)

    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.shape[0]
        dx = (self.y - self.t) / batch_size

        return dx
```

## 5.7 오차역전파법 구현하기

### 5.7.1 신경망 학습의 전체 그림

딥러닝은 손실함수의 값이 최소로 되도록 가중치와 편향인 매개변수를 조정하는 과정을 **학습**이라고 한다. 딥러닝 학습은 다음 4단계와 같다.

1. **미니배치(mini-batch)** : Train 데이터 중 랜덤하게 샘플을 추출하는데 이것을 미니배치라고 하며, 미니배치의 손실함수 값을 줄이는 것이 학습의 목표이다.
2. **기울기 계산** : 손실함수의 값을 작게하는 방향을 가리키는 가중치(**W**) 매개변수의 기울기를 구한다.
3. **매개변수 갱신** : 가중치 매개변수를 기울기 방향으로 학습률(learning rate)만큼 갱신한다.
4. **반복** 1~3 단계를 반복한다.

### 5.7.2 오차역전파법을 적용한 신경망 구현하기

```

In [41]: # 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()

    def predict(self, x):
        '''예측(추론)
        Params:
            - 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:

```



### 5.7.3 오차역전파법으로 구한 기울기 검증하기

수치 미분을 통해 구한 기울기와 오차역전파법의 결과를 비교하여 오차역전파를 제대로 구현했는지 검증하는 작업을 **기울기 확인(gradient check)**이라고 한다.

```
In [44]: %%time
# gradient_check.py
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist

# mnist load
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

network = TwoLayerNet(input_size=28*28, 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 절대 오차의 평균을 구한다.
for key in grad_numerical.keys():
    diff = np.average(np.abs(grad_backprop[key] - grad_numerical[key]))
    print(key, ":", str(diff))

W1 : 4.479721446541244e-10
b1 : 2.5485543061868916e-09
W2 : 4.349602602871501e-09
b2 : 1.393278526204411e-07
Wall time: 6.78 s
```

### 5.7.4 오차역전파법을 사용한 학습 구현하기

```

In [46]: %%time
# train_neuralnet.py
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist

# mnist load
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

network = TwoLayerNet(input_size=28*28, hidden_size=50, output_size=10)

# 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))

print('Optimization finished!')

```

|            |                    |                   |
|------------|--------------------|-------------------|
| Step: 600  | Train acc: 0.90450 | Test acc: 0.90560 |
| Step: 1200 | Train acc: 0.92288 | Test acc: 0.92570 |
| Step: 1800 | Train acc: 0.93220 | Test acc: 0.93200 |
| Step: 2400 | Train acc: 0.94605 | Test acc: 0.94460 |
| Step: 3000 | Train acc: 0.95430 | Test acc: 0.95210 |
| Step: 3600 | Train acc: 0.95993 | Test acc: 0.95870 |
| Step: 4200 | Train acc: 0.96360 | Test acc: 0.95870 |
| Step: 4800 | Train acc: 0.96682 | Test acc: 0.96320 |
| Step: 5400 | Train acc: 0.96930 | Test acc: 0.96380 |
| Step: 6000 | Train acc: 0.97108 | Test acc: 0.96450 |
| Step: 6600 | Train acc: 0.97318 | Test acc: 0.96690 |
| Step: 7200 | Train acc: 0.97557 | Test acc: 0.96890 |
| Step: 7800 | Train acc: 0.97698 | Test acc: 0.96760 |
| Step: 8400 | Train acc: 0.97808 | Test acc: 0.96990 |
| Step: 9000 | Train acc: 0.97835 | Test acc: 0.97030 |
| Step: 9600 | Train acc: 0.98035 | Test acc: 0.97020 |

Optimization finished!  
Wall time: 26.6 s

## 5.8 정리

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