분석 멘토링 C조_4주차

5장. 오차역전파법

발표자: 남유지

INDEX

1. 계산 그래프

1. 계산 그래프

: 계산 과정을 그래프로 나타낸 것
복수의 노드와 엣지로 표현

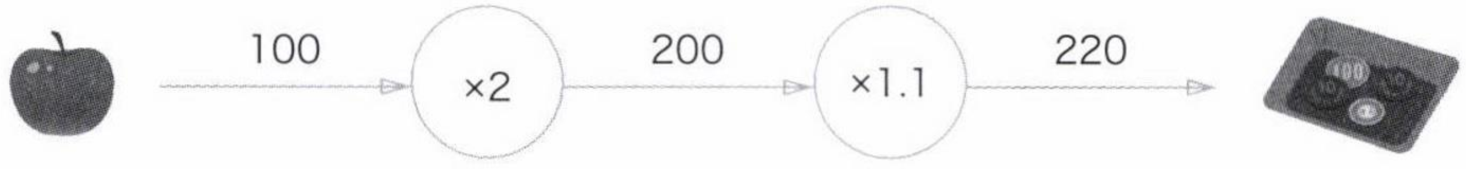
계산 그래프를 이용한 문제풀이

1) 계산 그래프 구성 (노드와 화살표)
2) 왼쪽 → 오른쪽으로 계산 진행 = 순전파

*순전파(forward propagation)
: 계산 그래프의 출발점부터 종착점으로의 전파

Q1) 슈퍼에서 1개에 100원인 사과를 2개 샀다. 이때 지불할 금액은?
(단, 소비세가 10% 부과됨)

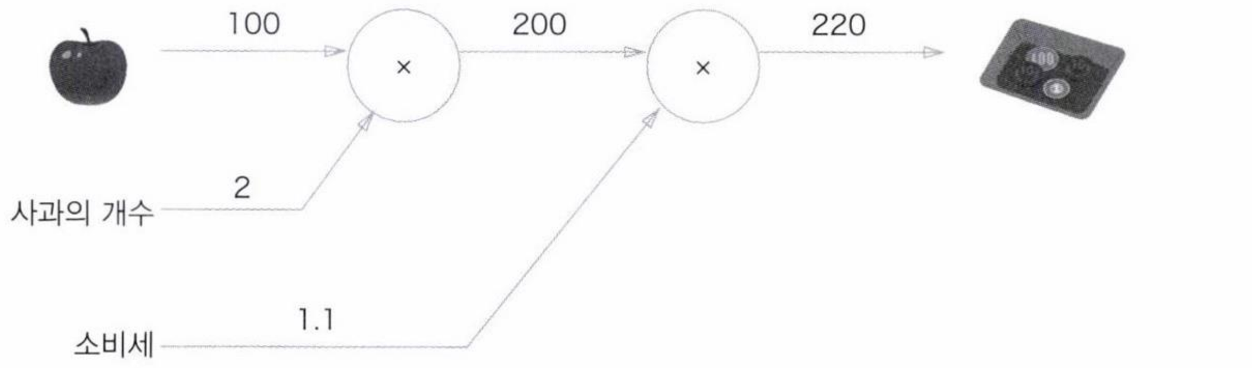
그림 5-1 계산 그래프로 풀어본 문제 1의 답



1. 계산 그래프

→ x(곱셈) 만 연산 노드로 표현
'사과 개수', '소비세'를 변수로 취급

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



계산 그래프를 사용하는 이유?

1. 복잡한 문제를 국소적 계산으로 단순화
2. 중간 계산 결과를 모두 보관할 수 있음
3. 역전파를 통해 미분을 효율적으로 계산

*국소적 계산: 자신과 관계된 정보만으로 결과를 출력

2. 연쇄법칙

2. 연쇄법칙

합성 함수: 여러 함수로 구성된 함수

$$z = (x+y)^2 \rightarrow \begin{array}{l} z = t^2 \\ t = x + y \end{array}$$

연쇄법칙: 합성함수의 미분에 대한 성질
→ 합성 함수의 미분은 합성 함수를 구성하는 각 함수의 미분의 곱으로 나타낼 수 있음

국소적 미분(편미분)

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

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

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

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

2. 연쇄법칙

계산 그래프의 역전파
: 오른쪽 → 왼쪽으로 신호를 전파

역전파 계산 절차
: (노드로 들어온 입력 신호) x (노드의 편미분) 을 다음 노드로 전파

→ 역전파가 하는 일은 연쇄법칙의 원리와 같다

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

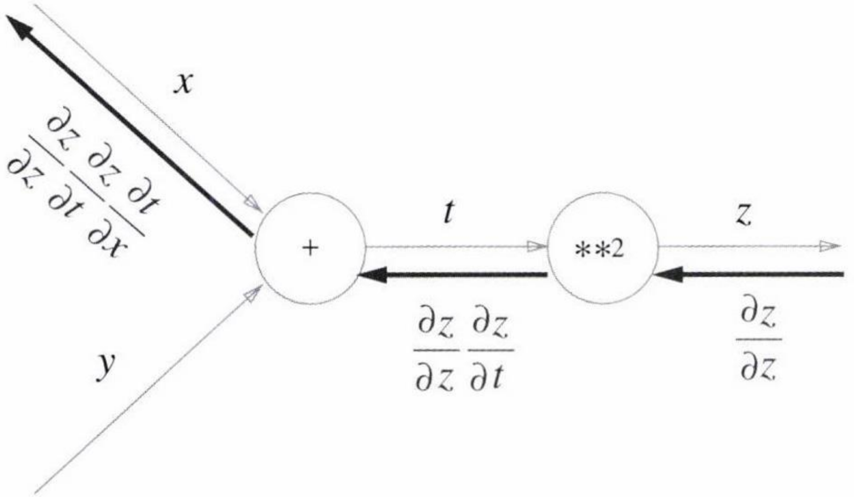
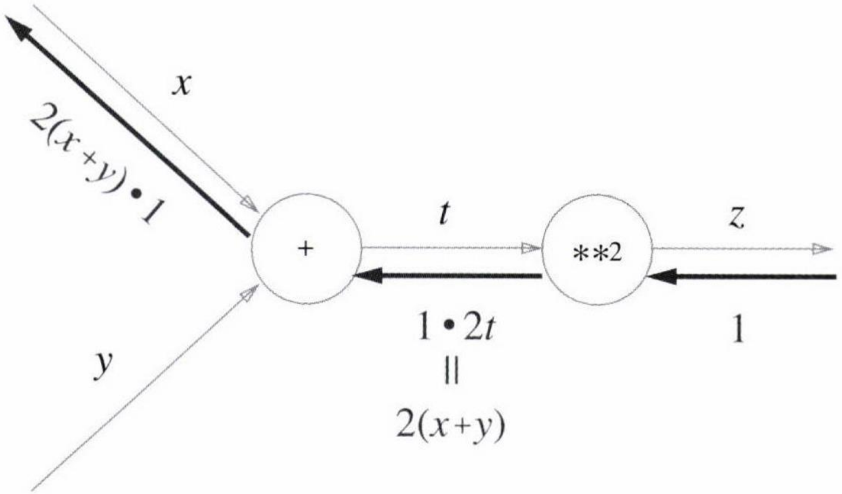


그림 5-8 계산 그래프의 역전파 결과에 따르면 $\frac{\partial z}{\partial x}$ 는 $2(x+y)$가 된다.



3. 역전파

3. 역전파

1) 덧셈 노드의 역전파
: 입력 값을 그대로 다음노드로 흘려 보냄

*최종적으로 L 값을 출력하는 큰 계산 그래프, 중간에 덧셈 노드가 존재한다고 가정

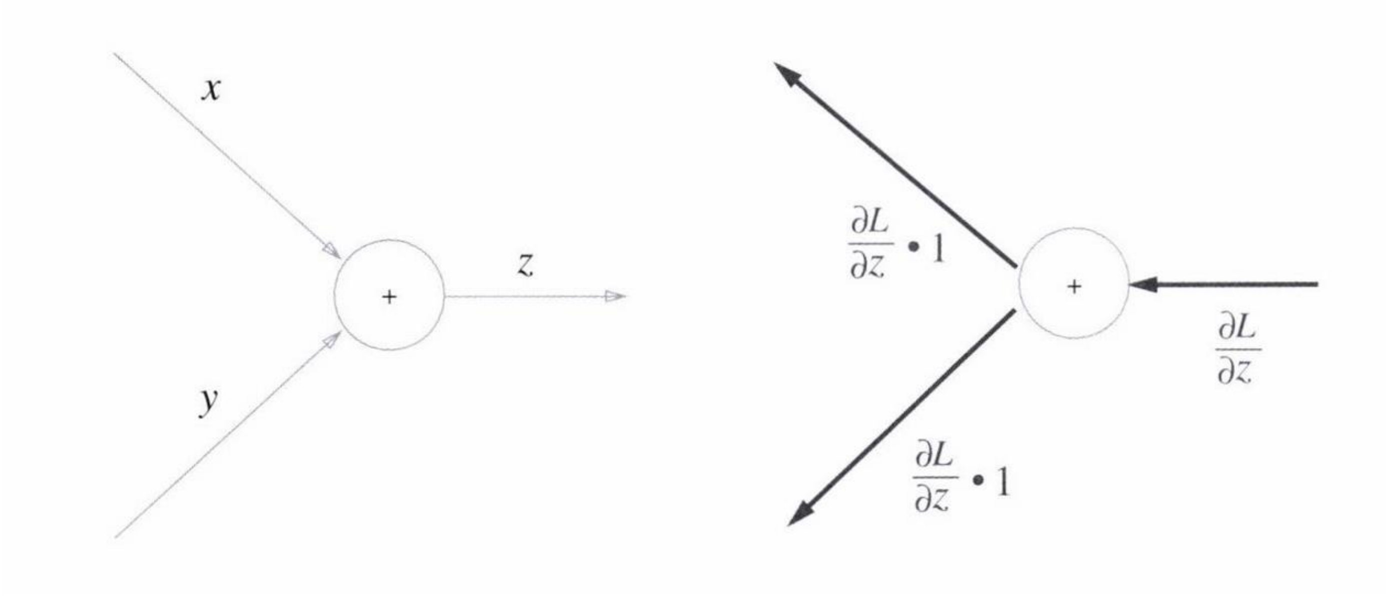
예시)

$$z = x + y$$

편미분이 모두 1

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

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



2) 곱셈 노드의 역전파

: 상류의 값에 **순전파 때의 입력 신호들**을 서로 바꾼 값을 곱해 하류로 보냄

*곱셈 노드 구현 시 순전파의 입력 신호를 변수에 저장

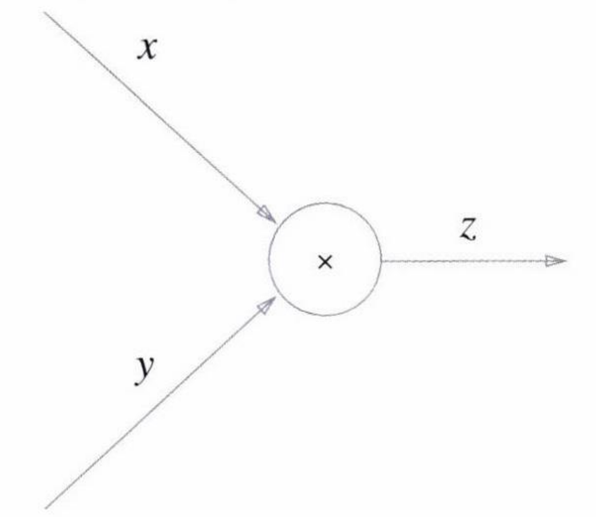
예시)

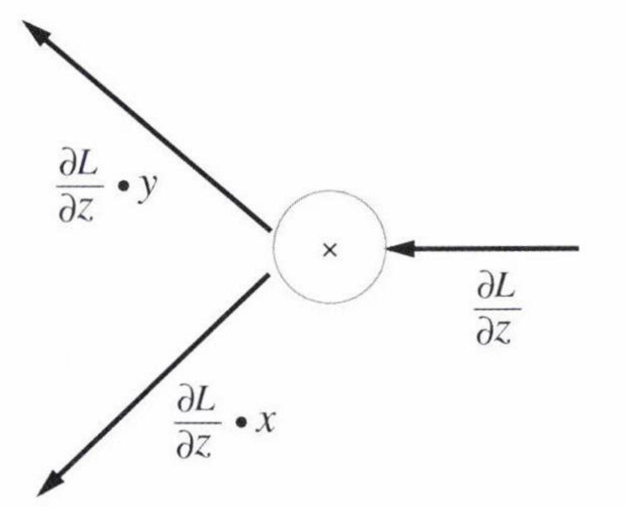
$$z = xy$$

편미분

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

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

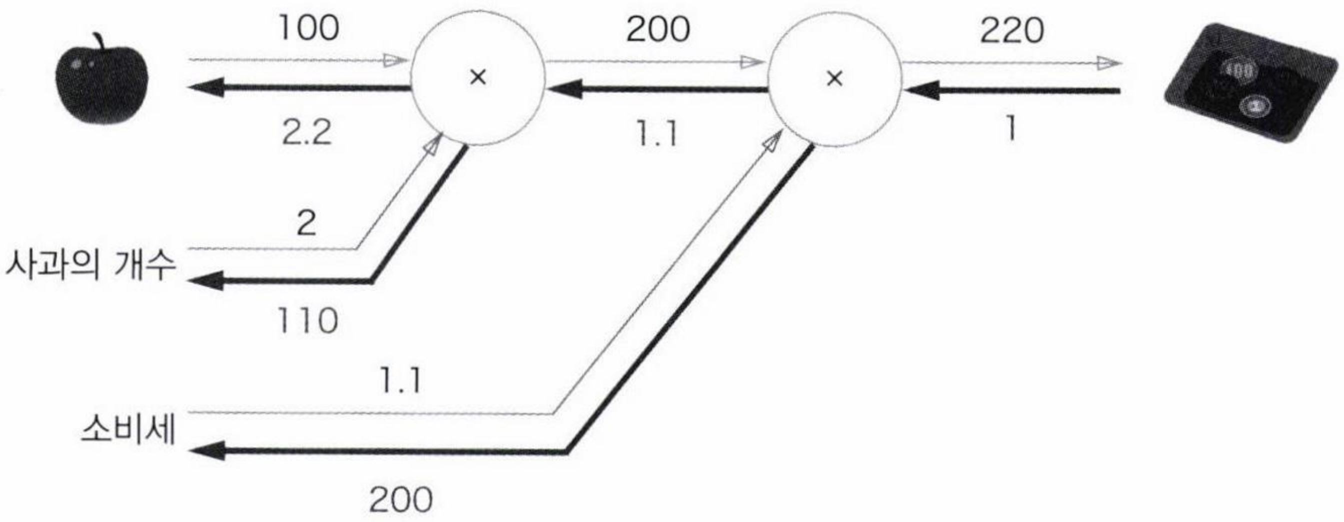




3. 역전파

Ex) 사과 쇼핑
사과 가격 / 사과 개수 / 소비세 가 최종 금액에 어떻게 영향을 주는가?
→ 각각에 대한 미분을 계산 그래프의 역전파를 사용해 계산

그림 5-14 사과 쇼핑의 역전파 예



*단위 주의(소비세 1 = 100%, 사과 가격 1 = 1원)

4. 단순한 계층 구현하기

4. 단순한 계층 구현하기

1) 곱셈 계층(MulLayer)

```
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
```

순전파 출력에 대한 미분

ex) 사과 쇼핑 순전파

```
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
```

4. 단순한 계층 구현하기

2) 덧셈 계층(AddLayer)

```
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
```

ex) 사과&귤 쇼핑 순전파

```
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)  #
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)
dorange, dorange_num = mul_orange_layer.backward(dorange_price)  # (2)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)  # (1)
```

5. 활성화 함수 계층 구현하기

5. 활성화 함수 계층 구현하기

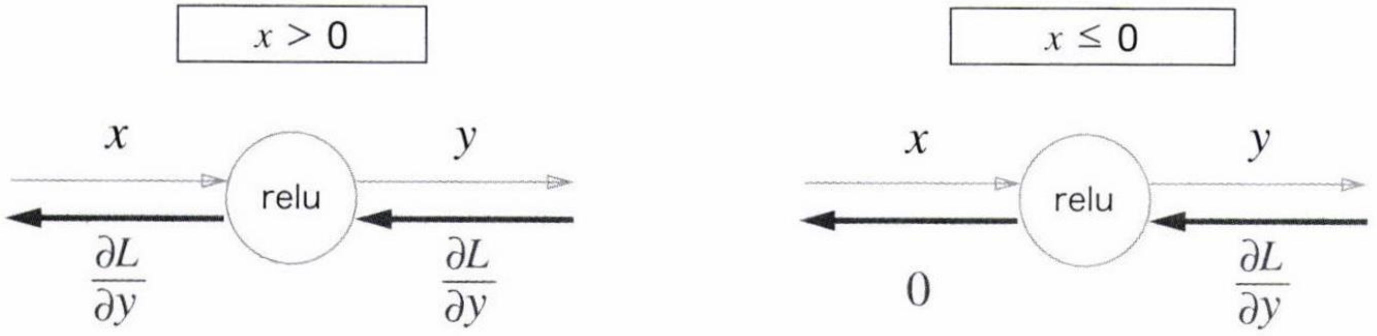
1) ReLU 계층

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

역전파

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

그림 5-18 ReLU 계층의 계산 그래프



5. 활성화 함수 계층 구현하기

1) ReLU 계층

```python
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
```

*forward(), backward()의 인수는 넘파이 배열

- mask 변수: 넘파이 배열,
순전파의 입력 x의 원소 값이 0 이하인 인덱스는 True, 그 외는 False 유지

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

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

```python
mask = (x <= 0)
print(mask)
```

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

```python
out = x.copy()
out[mask] = 0
out
```

```
array([[ 1. ,  0.5],
       [ 0. ,  3. ]])
```

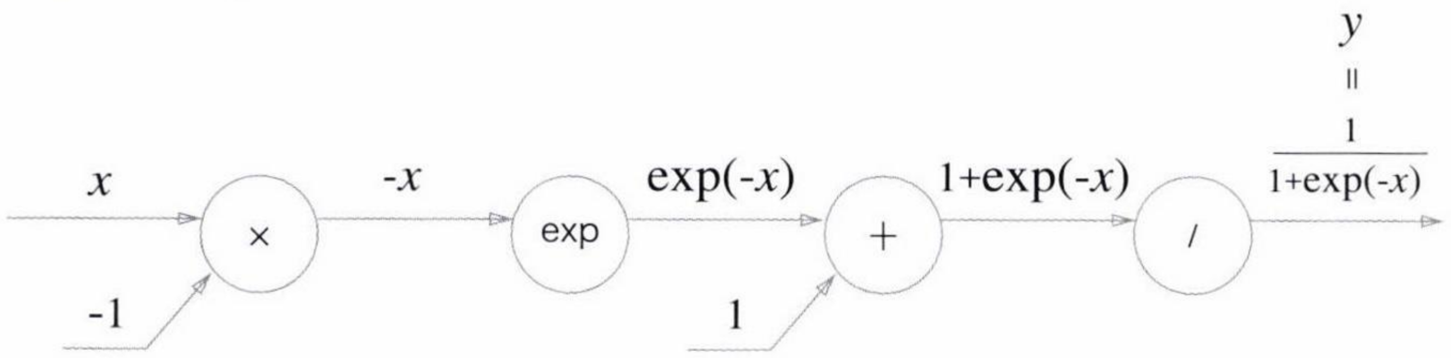
5. 활성화 함수 계층 구현하기

2) Sigmoid 계층

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

Sigmoid 계산 그래프

그림 5-19 Sigmoid 계층의 계산 그래프(순전파)



5. 활성화 함수 계층 구현하기

2) Sigmoid 계층

역전파

1단계 '/' 노드
2단계 '+' 노드
3단계 'exp' 노드
4단계 '*' 노드

- y = 1/x 미분

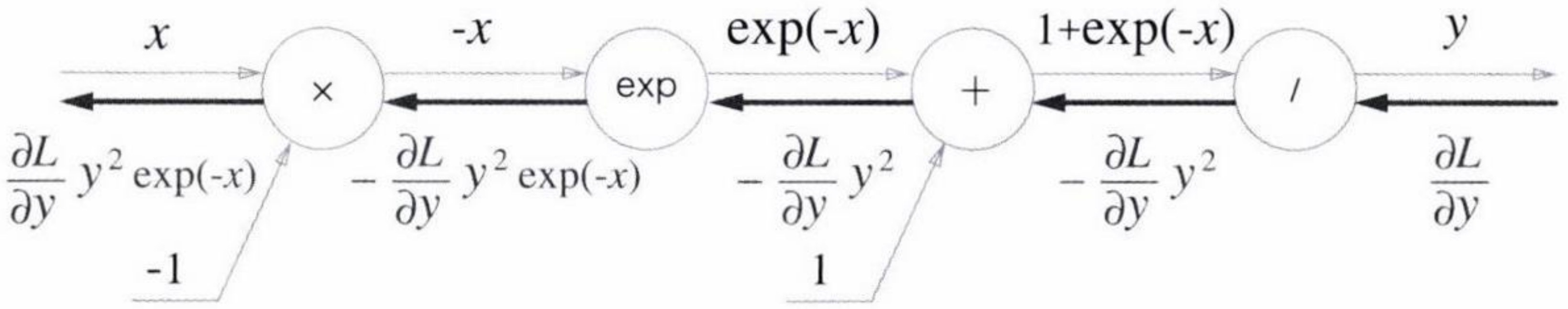
$$\frac{\partial y}{\partial x} = -\frac{1}{x^2}$$

$$= -y^2$$

- y = exp(x) 미분

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

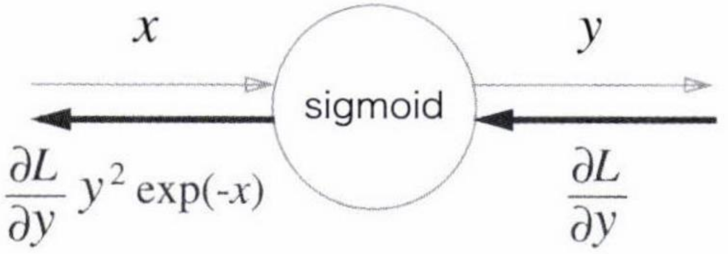
그림 5-20 Sigmoid 계층의 계산 그래프



5. 활성화 함수 계층 구현하기

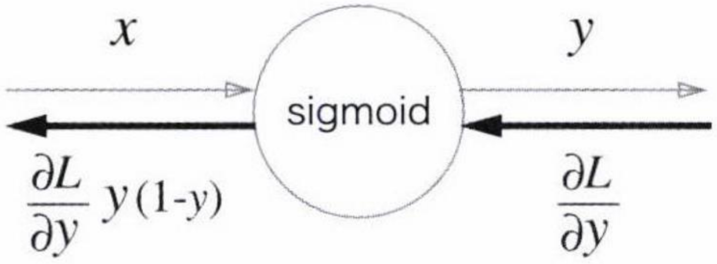
노드를 그룹화하여 ‘sigmoid’ 노드로 대체

그림 5-21 Sigmoid 계층의 계산 그래프(간소화 버전)



Sigmoid 계층의 역전파를 순전파의 출력 y만으로 계산할 수 있음

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



5. 활성화 함수 계층 구현하기

2) Sigmoid 계층

```
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
```

- out 변수

: 순전파의 출력을 보관, 역전파 계산 시 저장된 값을 사용해 효율적으로 계산 가능

6. Affine/Softmax 계층 구현하기

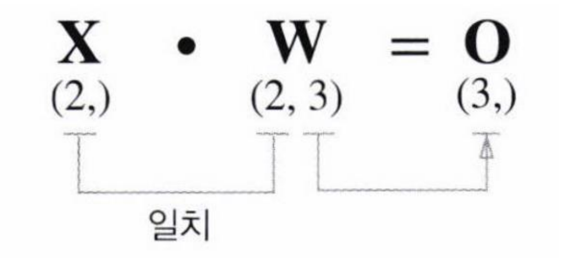
6. Affine/Softmax 계층 구현하기

신경망의 순전파에서 가중치 신호의 총합을 계산하기 위해
행렬의 곱 (np.dot())을 사용

```
X = np.random.rand(2)   # 입력
W = np.random.rand(2,3) # 가중치
B = np.random.rand(3)   # 편향

print(X.shape) # (2,)
print(W.shape) # (2, 3)
print(B.shape) # (3,)

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



해당하는 차원의 원소 수를 일치시키는게 핵심

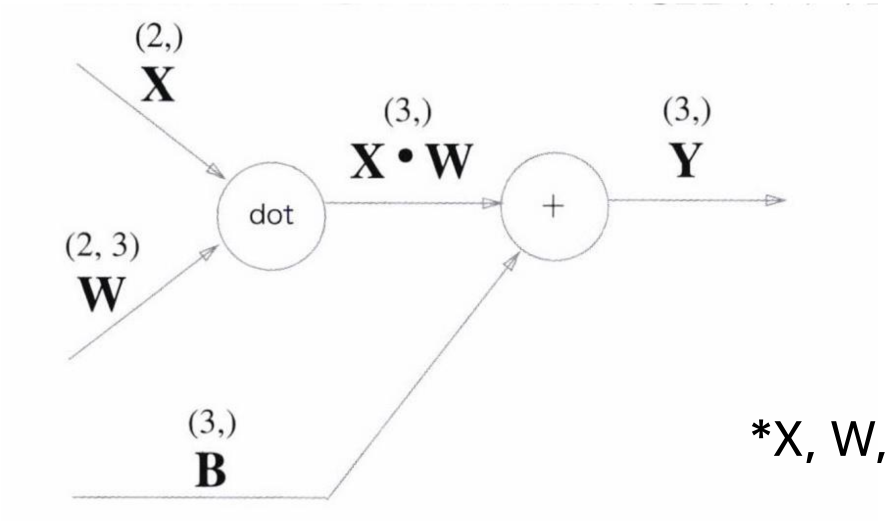
6. Affine/Softmax 계층 구현하기

1) Affine 계층

어파인 변환(affine transformation): 신경망의 순전파 때 수행하는 행렬의 곱
→ 어파인 변환을 수행하는 처리를 'Affine 계층'으로 구현

Affine 계층의 계산 그래프
(*입력 데이터로 X 하나만 고려)

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



*X, W, B : 행렬

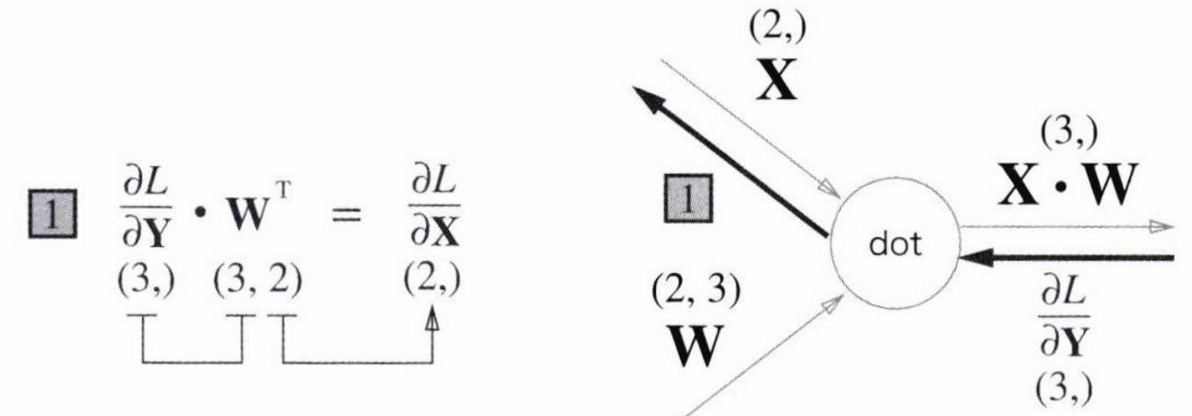
6. Affine/Softmax 계층 구현하기

- Affine 계층 역전파

Y = X·W 의 미분

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

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



*W^T: W의 전치행렬

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

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

6. Affine/Softmax 계층 구현하기

cf) 행렬에 대한 미분 증명

f가 $(1,m)$ x가 $(1\times n)$의 경우, 미분의 결과는 (m,n)

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \ddots & & \\ \vdots & & \ddots & \\ \frac{\partial f_m}{\partial x_1} & \cdots & & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

$$\frac{\partial (w_{11}x_1 + \overset{\text{상수항}}{\cancel{w_{21}x_2}})}{\partial x_1} = w_{11}$$

$$\begin{array}{l} (1,3) \\ (1,2) \end{array} \quad \frac{\partial Y}{\partial X} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \\ \frac{\partial y_3}{\partial x_1} & \frac{\partial y_3}{\partial x_2} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} = W^T$$

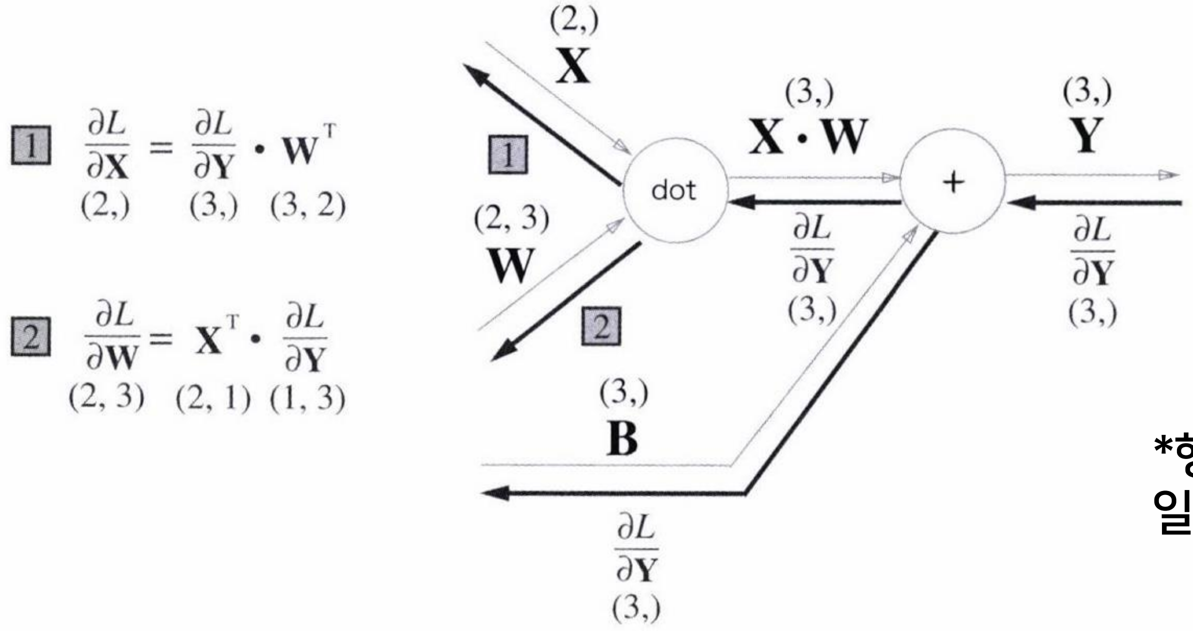
$(3,2)$

6. Affine/Softmax 계층 구현하기

- Affine 계층의 역전파

X와 $\frac{\partial L}{\partial X}$는 (2,)로 , W와 $\frac{\partial L}{\partial W}$ 는 (2,3)으로 같은 형상

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



*행렬의 곱에서는 차원의 원소 수를
일치시켜야 하기 때문에 행렬의 형상에 주의!

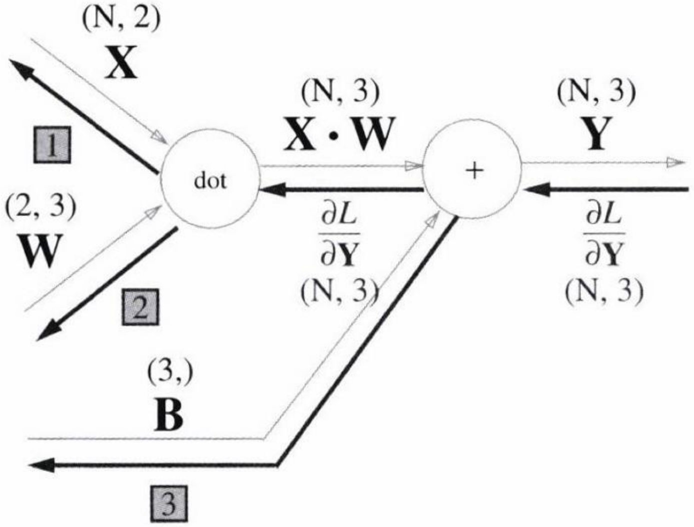
6. Affine/Softmax 계층 구현하기

- 배치용 Affine 계층의 역전파

: 데이터 N개를 묶어 순전파

- 입력 X의 형상이 (N, 2)

그림 5-27 배치용 Affine 계층의 계산 그래프



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

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

- 순전파: 편향이 각 데이터에 더해짐

```
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: 각 데이터의 역전파 값이 편향의 원소에 모여야 함

```
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])
```

6. Affine/Softmax 계층 구현하기

1) 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
```

6. Affine/Softmax 계층 구현하기

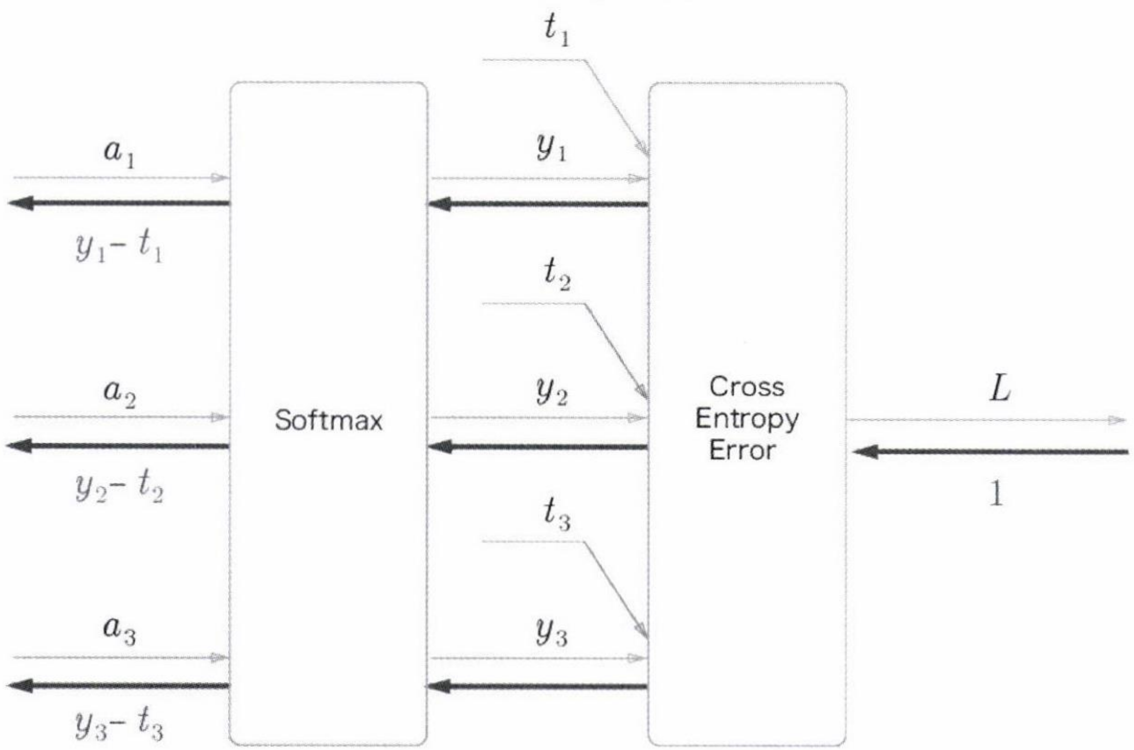
2) Softmax-with-Loss 계층

소프트맥스 함수: 입력 값을 정규화(출력의 합이 1이 되도록)하여 출력
→ 손실 함수 '교차 엔트로피 오차'도 포함하여 'Softmax-with-Loss'계층으로 구현

*3 클래스 분류를 가정

역전파의 결과 (y1-t1, y2-t2, y3-t3)
= softmax 계층의 출력과 정답 레이블의 차분

→ 신경망의 역전파에서는
오차가 앞 계층에 전해짐



6. Affine/Softmax 계층 구현하기

2) Softmax-with-Loss 계층

```python
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
```

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 4.2.2. 교차 엔트로피 오차 참고
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    # 훈련 데이터가 원-핫 벡터라면 정답 레이블의 인덱스로 반환
    if t.size == y.size:
        t = t.argmax(axis=1)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t])) / batch_size
```

*역전파 시 전파하는 값을 배치의 수로 나눠서 데이터 1개당 오차를 앞 계층으로 전파

7. 오차역전파법 구현하기

7. 오파역전파법 구현하기

신경망 학습의 순서

전제

신경망에는 적응 가능한 가중치와 편향 존재, 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정이 '학습 '

1단계 – 미니배치

훈련 데이터 중 일부를 무작위로 선택 → 미니배치

목표: 미니배치의 손실 함수 값을 줄이는 것

2단계 – 기울기 산출 **오차역전파법

각 가중치 매개변수의 기울기를 구함 (기울기는 손실함수의 값을 가장 작게하는 방향을 제시)

3단계 – 매개변수 갱신

가중치 매개변수를 기울기 방향으로 아주 조금 갱신

4단계 – 반복

1~3단계를 반복

7. 오파역전파법 구현하기

오차역전파법을 적용한 신경망 구현하기 -2층 신경망

```
def softmax(x):
    if x.ndim == 2:
        x = x.T
        x = x - np.max(x, axis=0)
        y = np.exp(x) / np.sum(np.exp(x), axis=0)
        return y.T

    x = x - np.max(x) # 오버플로 대책
    return np.exp(x) / np.sum(np.exp(x))
```

```
def numerical_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x)

    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        idx = it.multi_index
        tmp_val = x[idx]
        x[idx] = float(tmp_val) + h
        fxh1 = f(x) # f(x+h)

        x[idx] = tmp_val - h
        fxh2 = f(x) # f(x-h)
        grad[idx] = (fxh1 - fxh2) / (2*h)

        x[idx] = tmp_val # 값 복원
        it.iternext()

    return grad
```

7. 오파역전파법 구현하기

```python
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()                                        ###

    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
```

```python
# x : 입력 데이터, t : 정답 레이블
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):
    # forward
    self.loss(x, t)                              ###

    # backward
    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'], grads['b1'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
    grads['W2'], grads['b2'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

    return grads
```

7. 오파역전파법 구현하기

기울기 확인 (gradient check)

: 수치미분의 결과와 오차역전파법의 결과를 비교해 오차역전파법을 제대로 구현했는지 확인

```
# 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)

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))
```

```
b2:1.20126118774e-10
W1:2.80100167994e-13
W2:9.12804904606e-13
b1:7.24036213471e-13
```

*오차역전파법을 잘 구현했다면 오차가 0에 아주 가까운 값

7. 오파역전파법 구현하기

오차역전파법을 사용한 신경망 학습 구현하기

```
# 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)

iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

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

iter_per_epoch = max(train_size / batch_size, 1)
```

```
for i in range(iters_num):
    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(훨씬 빠르다)

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

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

    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)
```

*Note