Chapter 05 오차역전파법

수치 미분은 단순하고 구현하기도 쉽지만 계산 시간이 오래걸린다. 오차 역전파법이란 오차를 역(반대방향)으로 전파하는 방법이다.(backward propagation of errors)

5.1 계산 그래프

계산 그래프는 복수의 노드(node)와 에지(edge)로 표현된다.

5.1.1 계산 그래프로 풀다.

계산 그래프를 이용한 문제풀이는

1. 계산 그래프를 구성한다.
2. 그래프에서 계산을 왼쪽에서 오른쪽으로 진행한다.

여기서 계산을 왼쪽에서 오른쪽으로 진행하는 단계를 순전파(foward propagation)이라고 한다. 여기서 오른쪽에서 왼쪽으로 전파하는 경우를 역전파(backward propagation)이라고 한다. 역전파는 미분을 계산할 때 중요한 역할을 한다.

5.1.2 국소적 계산

계산 그래프의 특징은 '국소적 계산'을 전파함으로써 최종 결과를 얻는다는 점에 있다. 여기서 국소적이란 '자신과 직접 관계된 작은 범위' 라는 뜻이다.국소적 계산은 결국 전체에서 어떤 일이 벌어지든 상관없이 자신과 관계된 정보만으로 결과를 출력할 수 있다는 것이다. 결국 각 노드는 자신과 관련된 계산 외에는 아무것도 신경 쓸 것이 없다.

이렇듯 계산 그래프는 국소적 계산에 집중한다. 전체 계산이 아무리 복잡하더라도 각 단계에서 하는 일은 해당 노드의 '국소적 계산' 이다. 국소적 계산은 단순하지만 그 결과를 전달함으로써 전체를 구성하는 복잡한 계산을 해낼 수 있다.

NOTE 자동차 조립은 일반적으로 '조립 라인 작업'에 대한 분업으로 행해진다. 각 담당자(담당 기계)는 단순화된 일만 수행하며 그 일의 결과가 다음 담당자로 전달되어 최종적으로 차를 완성한다. 계산 그래프도 복잡한 계산을 '단순하고 국소적 계산'으로 분할하고 조립 작업 라인을 수행하며 계산 결과를 다음 노드로 전달한다. 복잡한 계산도 분해하면 단순한 계산으로 구성된다는 점은 자동차 조립과 마찬가지이다.

5.1.3 왜 계산 그래프로 푸는가?

전체가 아무리 복잡해도 각 노드에서는 단순한 계산에 집중하여 문제를 단순화할 수 있다. 게다가 무엇보다도 역전파를 통해 '미분'을 효율적으로 계산할 수 있는 장점이 있다.

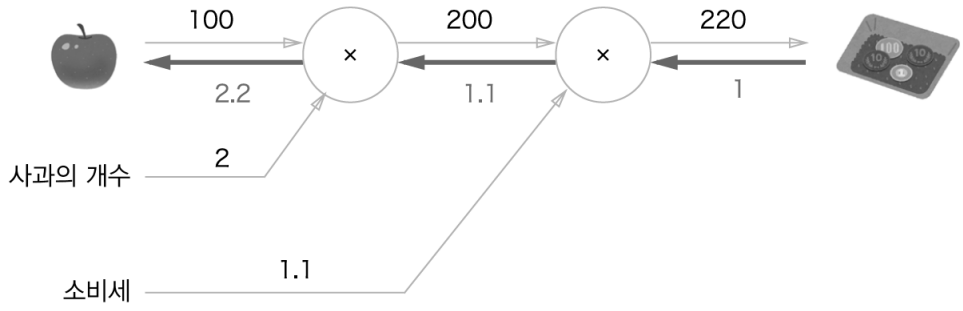


그림 **5-5** 역전파에 의한 미분 값의 전달

[그림 5-5]에서 역전파는 순전파와는 반대 방향의 화살표(굵은 선)으로 그린다. 이 전파는 '국소적 미분'을 전달하고 그 미분 값은 화살표 아래에 적는다. 이 예에서 역전파는 '1-> 1.1-> 2.2' 순으로 미분 값을 전달한다. '사과 가격에 대한 지불 금액의 미분' 값은 2.2라고 할 수 있다. 사과가 1원 오르면 최종 금액은 2.2원 오른다는 뜻이다.

5.2 연쇄법칙

역전파는 '국소적인 미분'을 순방향과는 반대인 오른쪽에서 왼쪽으로 전달한다. 또한, 이 '국소적 미분'을 전달하는 원리는 연쇄법칙(chain rule)에 따른 것이다.

5.2.1 계산 그래프의 역전파

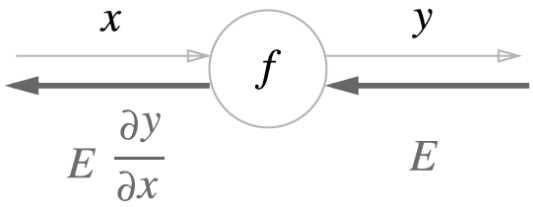


그림 **5-6** 계산 그래프의 역전파: 순방향과는 반대 반대로 국소적 미분을 곱한다.

역전파의 계산 절차는 신호 E에 노드의 국소적 미분($\frac{\partial y}{\partial x}$)을 곱한 후 다음 노드로 전달하는 것이다.

5.2.2 연쇄법칙이란?

합성 함수란 여러 함수로 구성된 함수이다.

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

이것이 연쇄법칙의 원리이다.

5.2.3 연쇄법칙과 계산 그래프

계산 그래프의 역전파는 오른쪽에서 왼쪽으로 신호를 전파한다. 역전파의 계산 절차에서는 노드로 들어온 입력 신호에 그 노드의 국소적 미분(편미분)을 곱한 후 다음 노드로 전달한다.

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

[식 5.4]

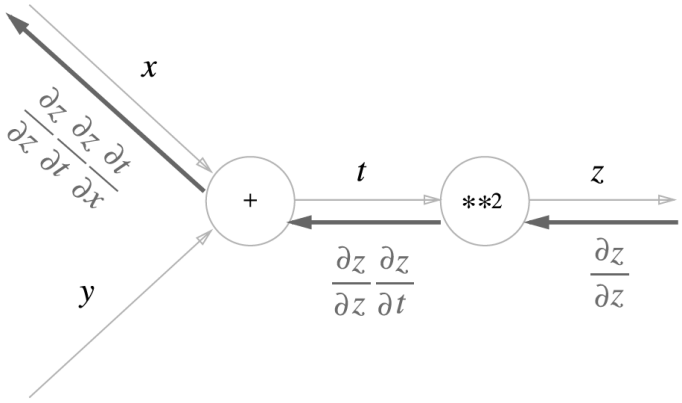


그림 **5-7** [식 5.4]의 계산 그래프: 순전파와는 반대 방향으로 국서적 미분을 곱하여 전달한다.

그림 5-7을 보면 역전파가 하는 일은 결국 연쇄법칙의 원리와 같다는 것을 알 수 있다.

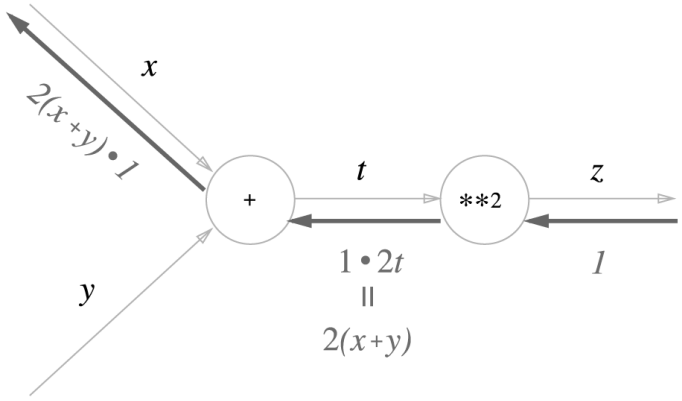


그림 **5-8** 계산 그래프의 역전파 결과에 따르면 $\frac{\partial z}{\partial x}$ 는 2(x + y)가 된다.

5.3 역전파

5.3.1 덧셈 노드의 역전파

덧셈 노드의 역전파는 1을 곱하기만 할 뿐이므로 입력된 값을 그대로 다음 노드로 보내게 된다.

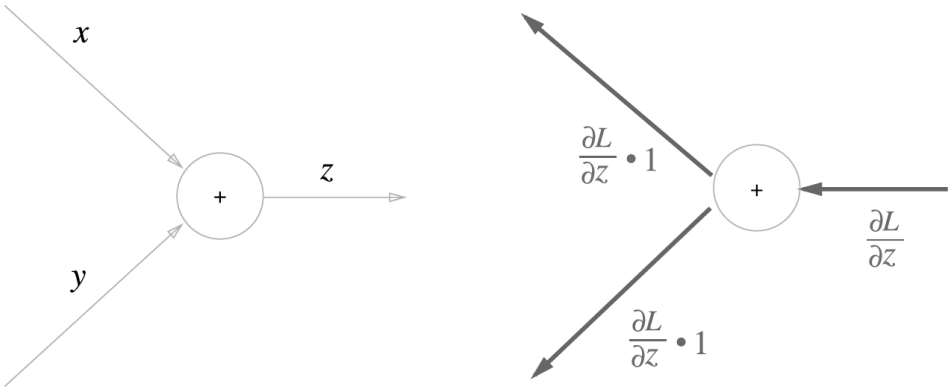


그림 **5-9** 덧셈 노드의 역전파: 왼쪽이 순전파, 오른쪽이 역전파이다. 덧셈 노드의 역전파는 입력 값을 그대로 흘려보낸다.

5.3.2 곱셈 노드의 역전파

곱셈 노드 역전파는 상류의 값에 순전파 때의 입력 신호들을 '서로 바꾼 값'을 곱해서 하류로 보낸다. 덧셈의 역전파에서는 상류의 값을 그대로 흘려보내서 순방향 입력 신호의 값은 필요하지 않았지만, 곱셈의 역전파는 순방향 입력 신호의 값이 필요하다.

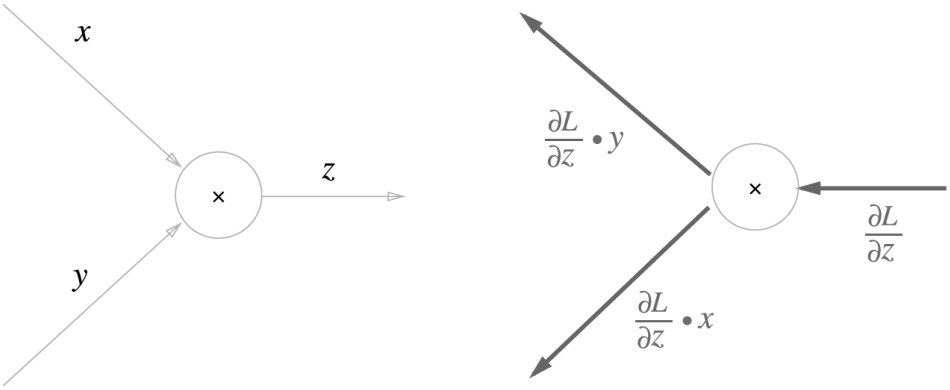
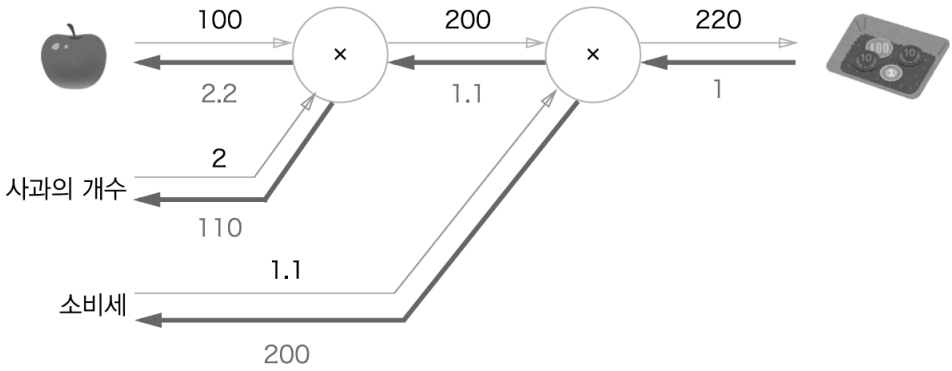


그림 **5-12** 곱셈 노드의 역전파: 왼쪽이 순전파, 오른쪽이 역전파다.

5.3.3 사과 쇼핑의 예



5.4 단순한 계층 구현하기

곱셈 노드를 'MulLayer', 덧셈 노드를 'AddLayer'라 명명한다.

5.4.1 곱셈 계층

모든 계층은 forward()와 backward()라는 공통의 메서드(인터페이스)를 갖도록 구현한다. foward()는 순전파, backward()는 역전파를 처리한다.

In [18]:

```
import numpy as np
```

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
```

__init__() 에서는 인스턴스 변수인 x와 y를 초기화한다. 이 두 변수는 순전파 시의 입력값을 유지하기 위해서 사용한다.

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(price)
```

```
220.00000000000003
```

각 변수에 대한 미분은 backward()에서 구할 수 있다.

In [3]:

```
# 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)
```

```
2.2 110.00000000000001 200
```

backward() 호출 순서는 foward()와는 반대이다. 또, bacward()가 받는 인수는 '순전파의 출력에 대한 미분'이다.

가령, mul_apple_layer라는 곱셈 계층은 순전파때는 apple_price를 출력하지만, 역전파때는 apple_price의 미분 값인 dapple_price를 인수로 받는다.

5.4.2 덧셈 계층

덧셈 계층은 다음과 같이 구현할 수 있다.

In [4]:

```
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
```

덧셈 계층에서는 초기화가 필요 없으니 __init__()에서는 아무것도 하지 않는다. 덧셈 계층의 forward()에서는 입력받은 두 인수 x, y를 더해서 반환한다. backward()에서는 상류에서 내려온 미분(dout)을 그대로 하류로 흘릴 뿐이다.

그림 5-17 구현

In [6]:

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

In [7]:

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

In [8]:

```
# 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)
```

In [10]:

```
# 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)
```

In [11]:

```
print (price)
```

```
715.0000000000001
```

In [13]:

```
print (dapple_num, dapple, dorange, dorange_num, dtax)
```

```
110.00000000000001 2.2 3.3000000000000003 165.0 650
```

필요한 계층을 만들어서 순전파 메서드인 forward()를 적절한 순서로 호출한다. 그 다음 순전파와 반대 순서로 역전파 매서드인 backward()를 호출한다.

5.5 활성화 함수 계층 구현하기

계산 그래프를 신경망에 적용한다. 여기에서는 신경망을 구성하는 층(계층) 각각을 클래스 하나로 구현한다. 우선 활성화 함수인 ReLU와 Sigmoid 계층을 구현한다.

5.5.1 ReLu 계층

활성화 함수로 사용되는 ReLU 수식은 다음과 같다.

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

[식 5.7]

위 식의 x에 대한 y 미분은 다음과 같다.

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

[식 5.8]

순전파 때의 입력인 x가 0보다 크면 역전파는 상류의 값을 그대로 하류로 흘린다. 하지만 순전파때 x가 0 이하면 역전파 때는 하류로 신호를 보내지 않는다.(0을 보낸다.)

그림 **5-18** ReLU 계층의 계산 그래프

In [17]:

```
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 0이하인 인덱스는 True, 그외(0보다 큰 원소)는 False로 유지한다. mask 변수는 다음 예와 같이 True/False로 구성된 넘파이 배열을 유지한다.

In [19]:

```
x = np.array([
    [1.0, -0.5],
    [-2.0, 3.0]
])
```

In [20]:

```
print (x)
```

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

In [21]:

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

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

순전파 때의 입력 값이 0 이하면 역전파 때의 값은 0이 되어야 한다. 그래서 역전파 때는 순전파때 만들어둔 mask를 써서 mask의 원소가 True인 곳에서는 상류에서 전파된 dout을 0으로 설정한다.

NOTE_ ReLU 계층은 전기 회로의 '스위치'에 비유할 수 있다. 순전파 때 전류가 흐르고 있으면 스위치를 ON으로 하고, 흐르지 않으면 OFF로 한다. 역전파 때는 스위치가 ON이라면 전류가 그대로 흐르고, OFF라면 더 이상 흐르지 않는다.

5.5.2 Sigmoid 계층

시그모이드 함수는 다음과 같다.

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

[식 5.9]

이를 순전파 계산그래프로 그리면 다음과 같다.

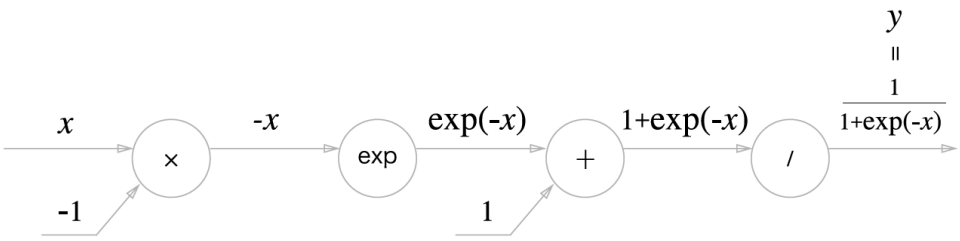


그림 **5-19** Sigmoid계층의 계산 그래프(순전파)

1단계

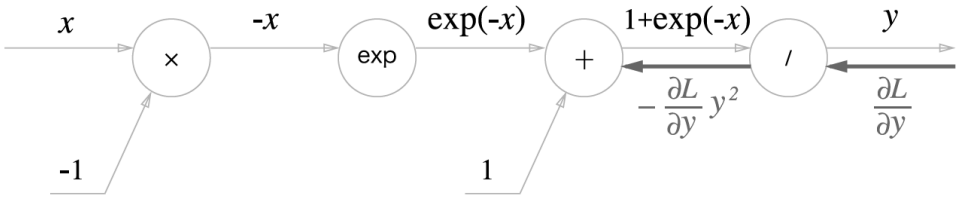
'/' 노트, 즉 y = 1/x를 미분하면 다음과 같다.

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

$$= -y^2$$

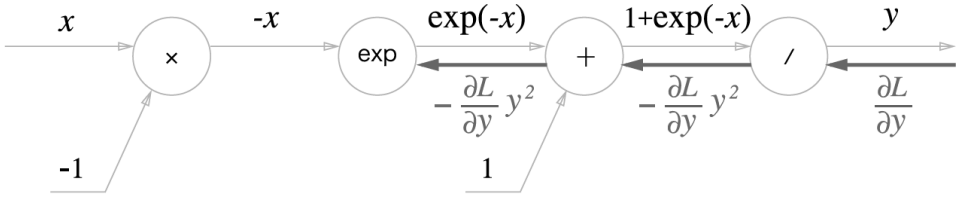
[식 5.10]

상류에서 흘러온 값에 $-y^2$을 곱해서 하류로 전달한다. 계산 그래프에서는 다음과 같다.



2단계

'+' 노드는 상류의 값을 여과 없이 하류로 내보내는 것이 다다. 계산 그래프에서는 다음과 같다.

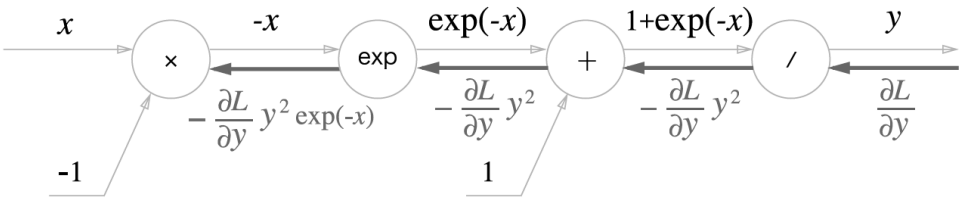


3단계

'exp'노드는 y=exp(x) 연산을 수행하며, 그 미분은 다음과 같다.

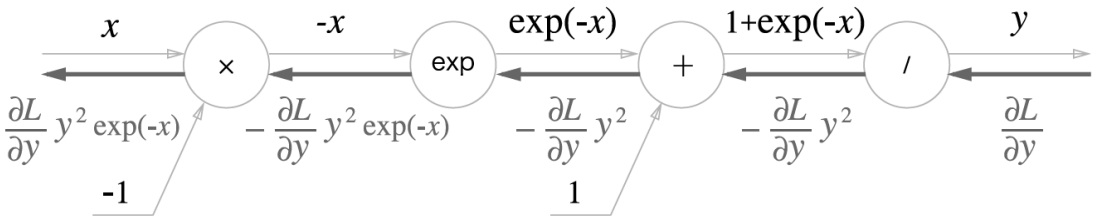
$$\frac{\partial y}{\partial x} = \exp(x)$$

계산 그래프에서 상류의 값에 순전파 때의 출력(이 예에서는 exp(-x))을 곱해 하류로 전달한다.

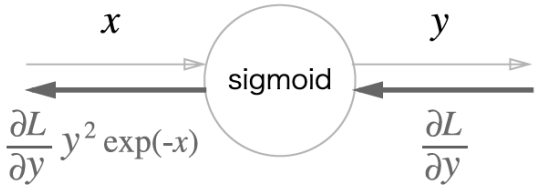


4단계

'X'노드는 순전파 때의 값을 서로 '바꿔' 곱한다. 이 예에서는 -1을 곱하면 된다.



위를 보면 순전파의 입력 x와 출력 y으로만 $\frac{\partial L}{\partial y} y^2 \exp(-x)$ 을 계산할 수 있다. 결국 이는 단순한 sigmoid 노드 하나로 대체될 수 있다.

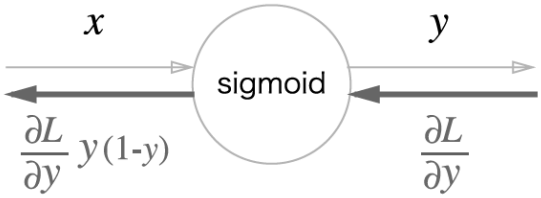


두 그래프 모두 결과는 똑같지만, 간소화 버전은 역전파 과정의 중간 계산들을 생략할 수 있어 더 효율적인 계산이라고 할 수 있다.

또한 $\frac{\partial L}{\partial y} y^2 \exp(-x)$는 다음 식 처럼 정리해서 쓸 수 있다.

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

[식 5.12]



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

이처럼 Sigmoid 계층의 역전파는 순전파의 출력(y)만으로 계산할 수 있다.

In [24]:

```
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
```

순전파의 출력을 인스턴스 변수 out에 보관했다가, 역전파 계산 때 그 값을 사용한다.

5.6 Affine / Softmax 계층 구현하기

5.6.1 Affine 계층

신경망의 순전파 때 수행하는 행렬의 내적은 기하학에서는 어파인 변환(affine transformation)이라고 한다. 이 책에서 어파인 변환을 수행하는 처리를 'Affine 계층' 이라는 이름으로 구현한다.

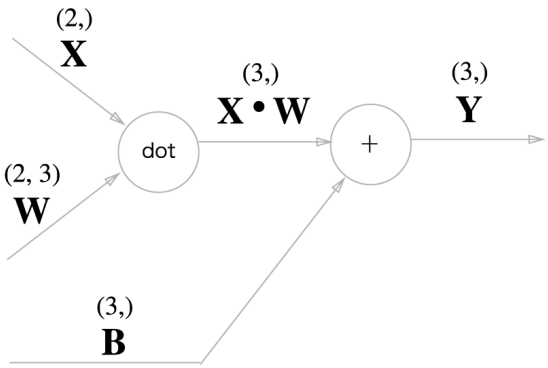


그림 **5-24** Affine 계층의 계산 그래프: 변수가 행렬임에 주의하자. 각 변수의 형상을 변수명 위에 표기했다.

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

[식 5.13]

[식 5.13]을 바탕으로 계산 그래프의 역전파를 구하면 다음 그림과 같다.

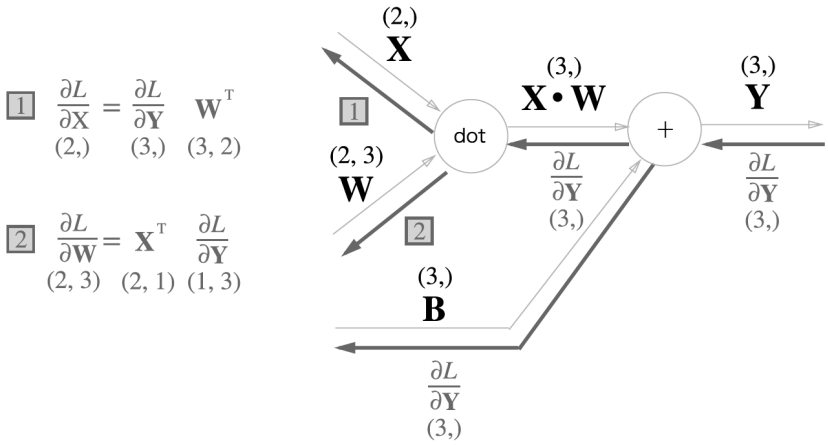


그림 **5-25** Affine 계층의 역전파: 변수가 다차원 배열임에 주의. 역전파에서의 변수 형상은 해당 변수명 아래에 표기한다.

X 와 $\frac{\partial L}{\partial X}$ 은 같은 형상이고 W 와 $\frac{\partial L}{\partial W}$ 도 같은 형상이라는 것을 기억하자. 이는 다음 식을 보면 명확해 진다.

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

[식 5.15]

또한 행렬의 형상에 주의하자. 행렬의 내적에서는 대응하는 차원의 수를 일치시켜야 한다.

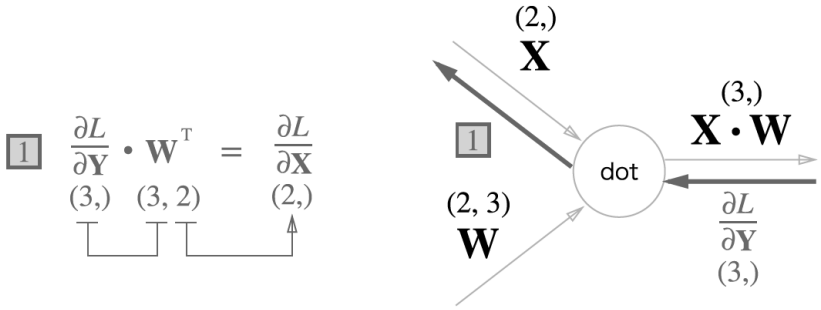


그림 **5-26** 행렬 내적('dot' 노드)의 역전파는 행렬의 대응하는 차원의 수가 일치하도록 내적을 조립하여 구할 수 있다.

In [27]:

```
X_dot_W = np.array([[0, 0, 0], [10, 10, 10]])
```

In [26]:

```
B = np.array([1,2,3])
```

In [28]:

```
X_dot_W
```

Out[28]:

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

In [29]:

```
X_dot_W + B
```

Out[29]:

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

순전파의 편향 덧셈은 각가의 데이터(1번째 데이터, 2번째 데이터, 3번째 데이터, ...)에 더해진다. 그래서 역전파 때는 각 역전파 데이터의 역전파 값이 편향의 원소에 모여야 한다.

In [30]:

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

Out[30]:

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

In [32]:

```
dB = np.sum(dY, axis=0)
dB
```

Out[32]:

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

이 예에서는 데이터가 2개(N = 2)라고 가정한다. 편향의 역전파는 그 두 데이터에 대한 미분을 데이터마다 더해서 구한다. 그래서 np.sum()에서 0번째 축(데이터를 단위로 한 축)에 대해서 (axis=0)의 총합을 구하는 것이다.

5.6.2 배치용 Affine 계층

데이터 N개를 묶어 순전파 하는 경우, 즉 배치용 Affine계층은 아래 그림과 같다.

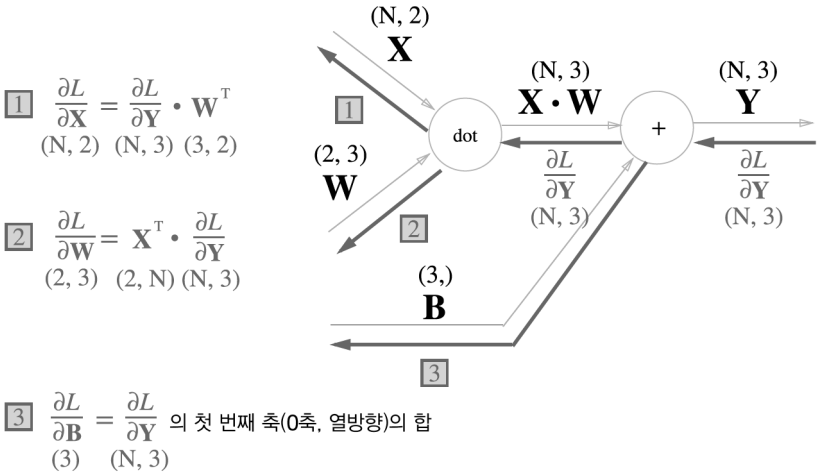


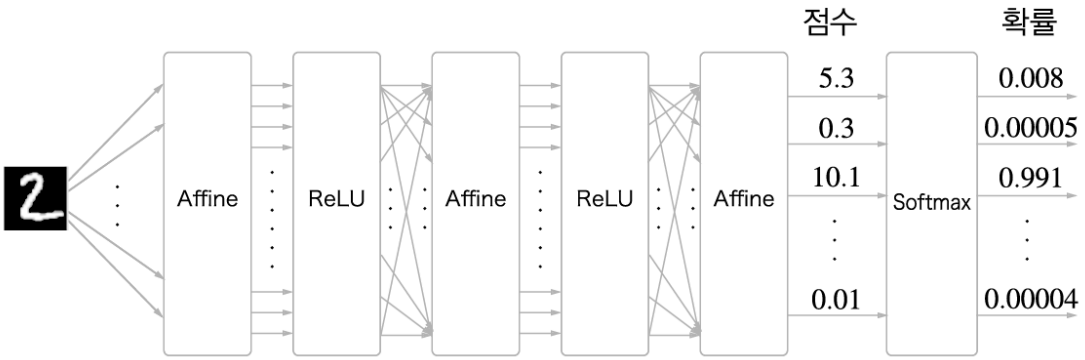
그림 **5-27** 배치용 Affine 계층의 계산 그래프

In [34]:

```
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 Softxmax-with-Loss 계층

소프트맥스함수는 입력 값을 정규화하여 출력한다.(출력의 합이 1이 되도록 변형). 또한 손글씨 숫자는 가짓수가 10개(10 클래스 분류)이므로 Softmax 계층의 입력은 10개가 된다.



Note 신경망에서 수행하는 작업은 학습과 추론 두 가지이다. 추론할 때는 일반적으로 Softmax 계층을 사용하지 않는다.

예컨대 신경망을 추론할 때에는 마지막 Affine 계층의 출력을 인식 결과로 이용한다. 또한 신경망에서 정규화하지 않은 출력 결과를 점수라 한다. 신경망 추론에서 답을 하나만 내는 경우에는 가장 높은 점수만 알면 되니 Softxmax 계층은 필요 없다는 것이다. 반면 신경망을 학습할 때에는 Softmax 계층이 필요하다.

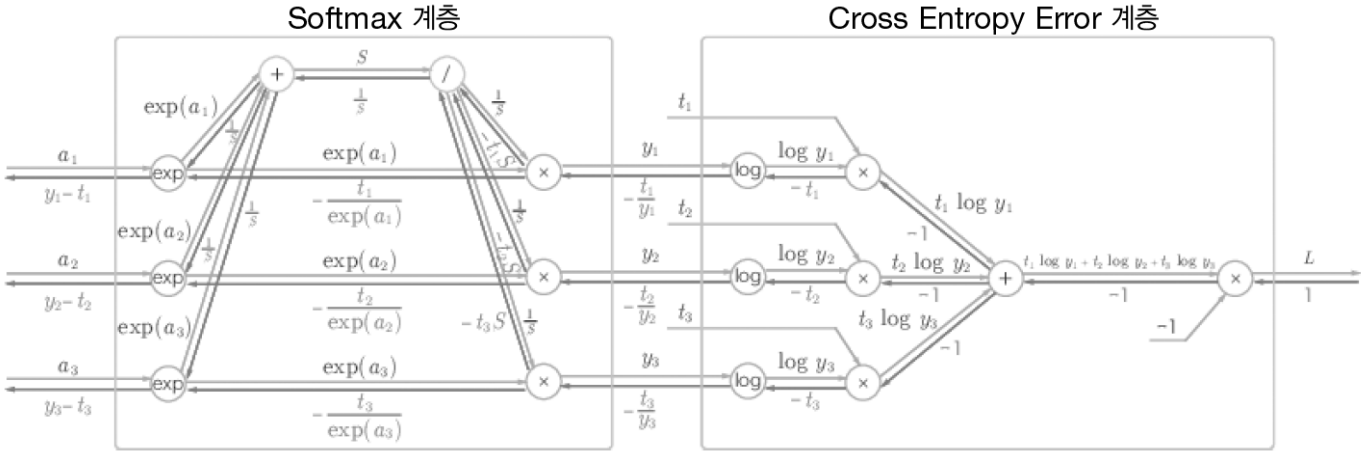


그림 **5-29** Softmax-with-Loss 계층의 계산 그래프

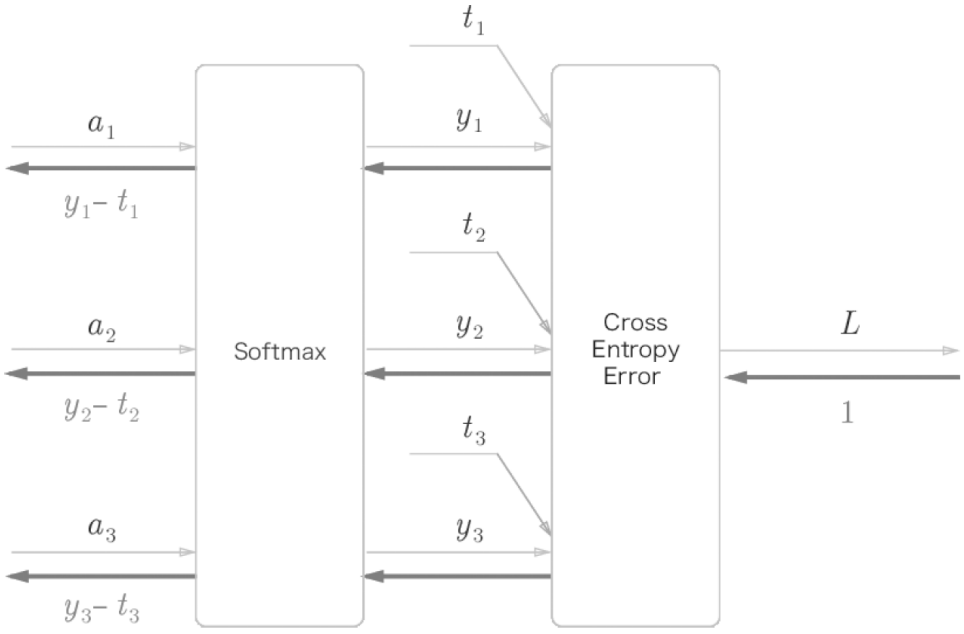


그림 **5-30**'간소화한' Softmax-with-Loss 계층의 계산 그래프

[그림 5-30]의 계산 그래프에서 소프트맥스 함수는 'Softmax' 계층으로, 교차 엔트로피 오차는 'Cross Entropy Error' 계층으로 표기했다. 여기에서는 3클래스 분류를 가정하고 이전 계층에서 3개의 입력(점수)를 받는다. 그림과 같이 Softmax 계층은 입력 (a_1, a_2, a_3)를 정규화하여 (y_1, y_2, y_3)를 출력한다. Cross Entropy Error 계층은 Softmax의 출력 (y_1, y_2, y_3)와 정답 레이블 (t_1, t_2, t_3)를 받고 이들 데이터에서 손실 L을 출력한다.

Softmax 계층의 역전파는 ($y_1 - t_1, y_2 - t_2, y_3 - t_3$) 라는 말끔한 결과를 내놓고 있다. ($y_1$, y_2, y_3)은 Softmax 계층의 출력이고 (t_1, t_2, t_3)는 정답 레이블이므로 ($y_1 - t_1, y_2 - t_2, y_3 - t_3$)는 Softmax 계층의 출력과 정답 레이블의 차분인 것이다. 신경망의 역전파에서는 이 차이인 오차가 앞 계층에 전해지는 것이다.

신경망 학습의 목적은 신경망의 출력(Softmax의 출력)이 정답 레이블에 가까워지도록 가중치 매개변수의 값을 조정하는 것이다. 그래서 신경망의 출력과 정답 레이블의 오차를 효율적으로 앞 계층에 전달해야 한다. 앞의 ($y_1 - t_1, y_2 - t_2, y_3 - t_3$)라는 결과는 바로 Softmax 계층의 출력과 정답 레이블의 차이로, 신경망의 현재 출력과 정답 레이블의 오차를 그대로 드러낸다.

NOTE '소프트맥수 함수'의 손실 함수로 '교차 엔트로피 오차'를 사용하면 역전파가 말끔히 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$로 떨어진다. 이 이유는 교차 엔트로피 오차라는 함수가 그렇게 설계되었기 때문이다. 회귀의 출력 등에서 사용하는 '항등 함수'의 손실 함수로 '평균 제곱 오차'를 이용하면 역전파의 결과가 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$로 말끔히 떨어진다.

예를 들자면, 정답 레이블이 (0, 1, 0)일때 Softmax 계층이 (0.3, 0.2, 0.5)를 출력하였다면 Softmax 계층의 역전파는 (0.3, -0.8 0.5)라는 커다란 오차를 전파한다. 결과적으로 Softmax 계층의 앞 계층들은 그 큰 오차들로부터 큰 깨달음을 얻게 된다. 가령 정답 레이블이 (0, 1, 0)일때 Softmax 계층이 (0.01, 0.99, 0)을 출력한다면 Softmax 계층의 역전파가 보내는 오차는 비교적 작은 (0.01, -0.01, 0)이다. 이번에는 앞 계층으로 전달된 오차가 작으므로 학습하는 정도도 작아진다.

Softmax-with-Loss 계층을 구현한 코드는 다음과 같다.

In [36]:

```
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.y)
        return self.loss

    def backward(self, dout=1):
        batch_size = self.t.shape[0]
        dx = (self.y - self.t) / batch_size

        return dx
```

역전파 때는 전파하는 값을 배치의 수(batch_size)로 나눠서 데이터 1개당 오차를 앞 계층으로 전파하였다.

5.7 오차역전파법 구현하기

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 손실 함수 값을 줄이는게 목표다.

2단계 **-** 기울기 산출

미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구한다. 기울기는 손실 함수의 값을 가장 적게 하는 방향을 제시한다.

3단계 **-** 매개변수 갱신

가중치 매개변수를 기울기 방향으로 아주 조금 갱신한다.

4단계 **-** 반복

1~3단계를 반복한다.

오차역전파법이 등장하는 단계는 두 번째인 '기울기 산출'이다. 앞 장에서는 기울기를 구하기 위해 수치 미분을 사용했지만, 오차역전파법을 이용하면 수치 미분과 달리 기울기를 효과적이고 빠르게 구할 수 있다.

5.7.2 오차역전파법을 적용한 신경망 구현하기

2층 신경망을 TwoLayerNet 클래스로 구현한다.

In [38]:

```
import sys, os
import numpy as np
from common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict
```

In [40]:

```
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

    # 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):
        # 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
```

신경망의 계층을 OrderedDict에 보관한다. 즉, 순서가 있는 딕셔너리이다. 그래서 순전파 때는 추가한 순서대로 각 계층의 forward() 메서드를 호출하기만 하면 처리가 완료된다. 마찬가지로 역전파 때는 계층을 반대로 호출하기만 하면 된다.

Affine 계층과 ReLU 계층이 각자의 내부에서 순전파와 역전파를 처리하고 있으니, 여기에서는 그냥 계층을 올바른 순서대로 연결한 다음 순서대로(혹은 역순으로) 호출해주면 된다.

5.7.3 오차역전파법으로 구한 기울기 검증하기

수치 미분의 이점은 구현하기 쉽다는 점이다. 그래서 수치 미분의 구현에는 버그가 숨어있기 어려운 반면, 오차역전파법은 구현하기 복잡해서 종종 실수를 하곤 한다. 여기서 수치 미분의 기울기와 오차역전파의 기울기가 일치함을 확인하는 작업을 기울기 확인(gradient check)이라고 한다.

In [45]:

```
import sys, os
import numpy as np
from dataset.mnist import load_mnist
```

In [46]:

```
# 데이터 읽기
(x_train, t_train), (x_test, t_test) = \
                                        load_mnist(normalize=True, one_hot_label=Tr

network = TwoLayerNet(input_size = 784, hidden_size=50, output_size=10)
```

In [47]:

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

In [48]:

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

In [49]:

```
# 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))
```

```
b1:6.98727888903e-13
b2:1.20126136816e-10
W2:8.04333310893e-13
W1:2.37758035282e-13
```

기울기의 차이가 매우 적다.

NOTE 수치 미분과 오차역전파법의 결과 오차가 0이 드는 되는 경우는 드물다. 컴퓨터의 정밀도가 유한하기 때문이다. 하지만 올바르게 구현하였다면 0에 아주 가까운 작은 값이 된다. 만약 그 값이 크면 오차역전파법을 잘못 구현하였다고 의심해 보아야 한다.

5.7.4 오차역전파법을 사용한 학습 구현하기

In [54]:

```
import numpy as np
from dataset.mnist import load_mnist

# 데이터 읽기
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=Tru

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(i, iter_per_epoch, train_acc, test_acc)
```

```
0 600.0 0.100683333333 0.098
600 600.0 0.905433333333 0.9085
1200 600.0 0.924583333333 0.9264
1800 600.0 0.938883333333 0.9381
2400 600.0 0.9458 0.9454
3000 600.0 0.9539 0.9515
3600 600.0 0.95825 0.955
4200 600.0 0.96115 0.9571
4800 600.0 0.965566666667 0.9593
5400 600.0 0.96805 0.9614
6000 600.0 0.971533333333 0.9634
6600 600.0 0.973083333333 0.9659
7200 600.0 0.974616666667 0.9669
7800 600.0 0.97575 0.9682
8400 600.0 0.97765 0.9678
9000 600.0 0.97815 0.9693
```

```
9600 600.0 0.980283333333 0.9706
```

5.8 정리

계산 과정을 시각적으로 보여주는 보여주는 계산 그래프를 배웠다. 그리고 계산 그래프를 이용하여 신경망의 동작과 오차역전파법을 설명하고, 그 처리 과정을 계층이라는 단위로 구현하였다. 예를 들어 ReLu 계층, Softmax-with-Loss 계층, Affine 계층, Softmax 계층 등이다.

모든 계층에서 forward와 backward라는 메서드를 구현하였다. 전자는 데이터를 순방향으로 전파하고, 후자는 역방향으로 전파함으로써 가중치 매개변수의 기울기를 효율적으로 구할 수 있다. 이처럼 동작을 계층으로 모듈화한 덕택에 신경망의 계층을 자유롭게 조합하여 원하는 신경망을 쉽게 만들 수 있다.

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

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: