밑바닥부터 시작하는 딥러닝 CHATER 5

오차역전파법

가중치 매개변수의 기울기를 효율적으로 계산하는 방법이다. 크게 두 가지로 이해할 수 있다. 하나는 수식을 통한 것, 하나는 계산 그래프를 통한 것이다.

계산 그래프

계산그래프는 계산 과정을 그래프로 나타낸 것이다. 여기서의 그래프는 우리가 잘 아는 그래프 자료구조로 복수의 node와 edge로 표현된다.

계산 그래프로 풀다

계산그래프를 이용한 문제풀이는 다음 흐름으로 진행된다

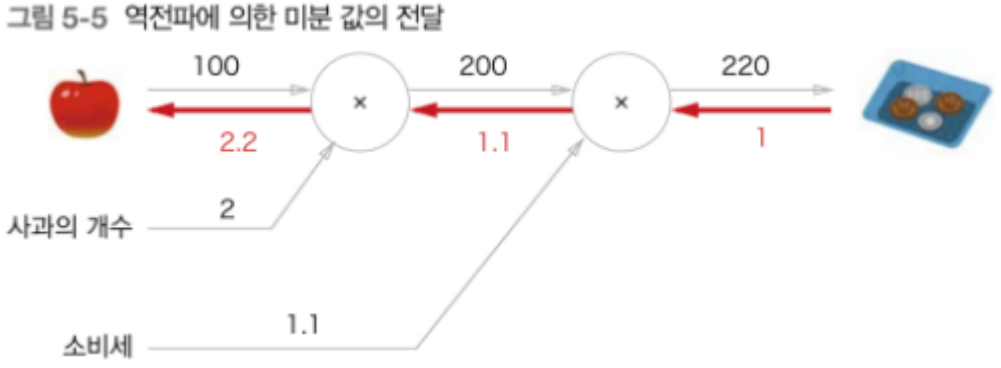
1. 계산 그래프를 구성한다

2. 그래프에서 계산을 왼쪽에서 오른쪽으로 진행한다.

2번에 해당하는 단계를 순전파라고 한다. 순전파는 계산 그래프의 출발점부터 종착점으로의 전파이다. 이것에 반대되는 계산 방법을 역전파라고 한다.

계산 그래프의 특징은 '국소적 계산'을 전파함으로써 최종 결과를 얻는다는 점에 있다. 전체 계산이 아무리 복잡해도 각 노드에서는 단순한 계산에 집중하여 문제를 단순화할 수 있다. 또 다른 이점으로 계산 그래프는 중간 계산 결과를 모두 보관할 수 있다. 마지막으로 실제 계산 그래프를 사용하는 가장 큰 이유는 역전파를 통해 미분을 효율적으로 계산할 수 있기 때문이다. 역전파에 의한 미분 값의 전달을 그림으로 살펴보면 다음과 같다.

그림 5-5 역전파에 의한 미분 값의 전달



이처럼 계산 그래프의 이점은 순전파와 역전파를 활용해서 각 변수의 미분을 효율적으로 구할 수 있다는 것이다.

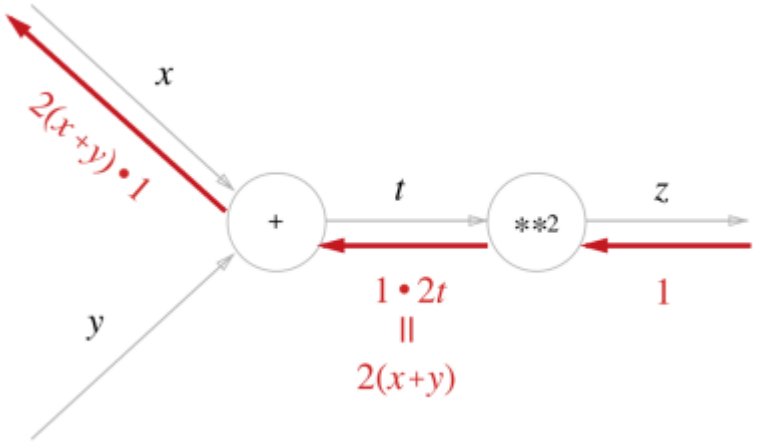
연쇄법칙

역전파는 국소적인 미분을 순방향과는 반대인 오른쪽에서 왼쪽으로 전달하는데 이 원리는 연쇄법칙에 따른 것이다. 연쇄법칙을 설명하려면 합성 함수부터 시작해야한다. 연쇄법칙은 합성 함수의 미분에 대한 성질이며 합성 함수의 미분은 합성 함수를 구성하는 각 함수의 미분의 곱으로 나타낼 수 있다.

연쇄법칙과 계산 그래프

연쇄법칙 계산을 계산 그래프로 나타내면 다음과 같이 나타낼 수 있다.

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

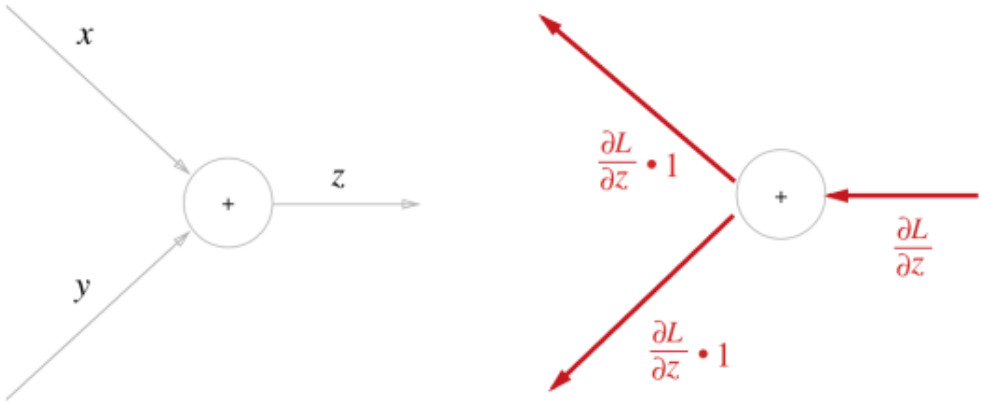


역전파의 계산 절차에서는 노드로 들어온 입력 신호에 그 노드의 편미분을 곱한 후 다음 노드로 전달한다.

역전파

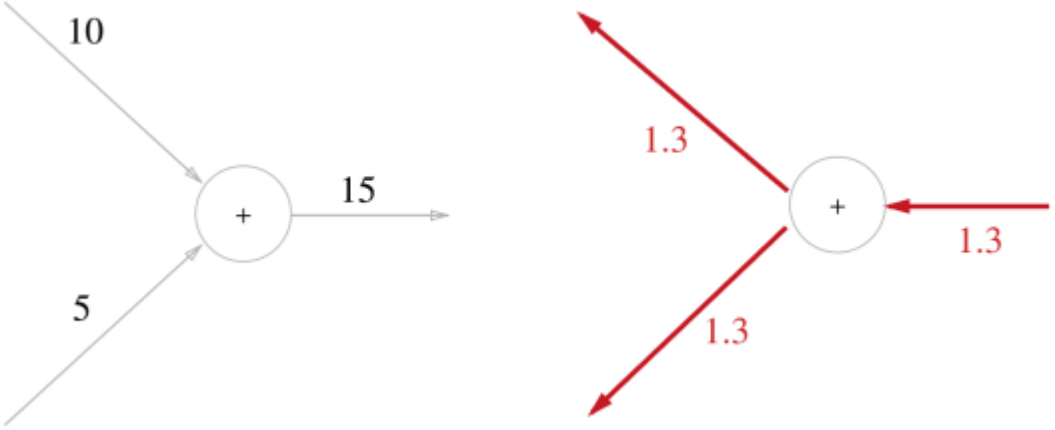
덧셈 노드의 역전파

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



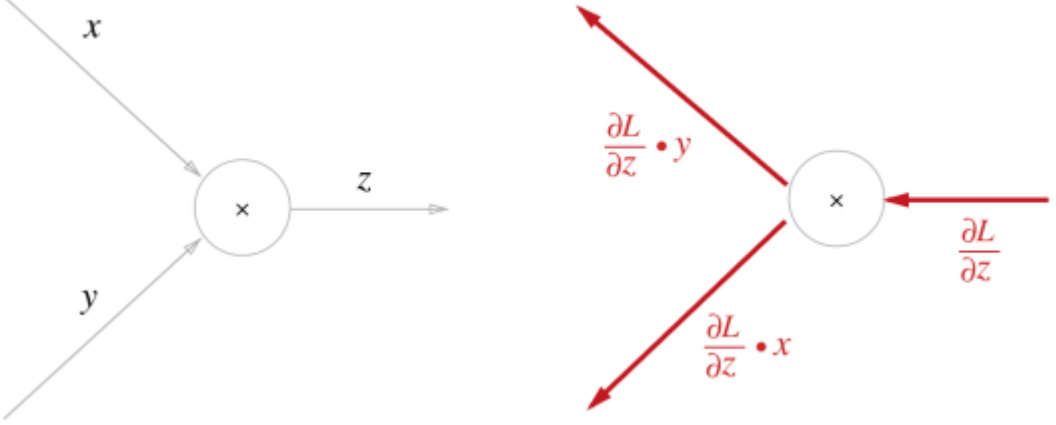
덧셈 노드의 역전파는 1을 곱하기만 할 뿐이므로 입력된 값을 그대로 다음 노드로 보내게 된다. 그 구체적인 예를 살펴보면 다음과 같다.

그림 5-11 덧셈 노드 역전파의 구체적인 예



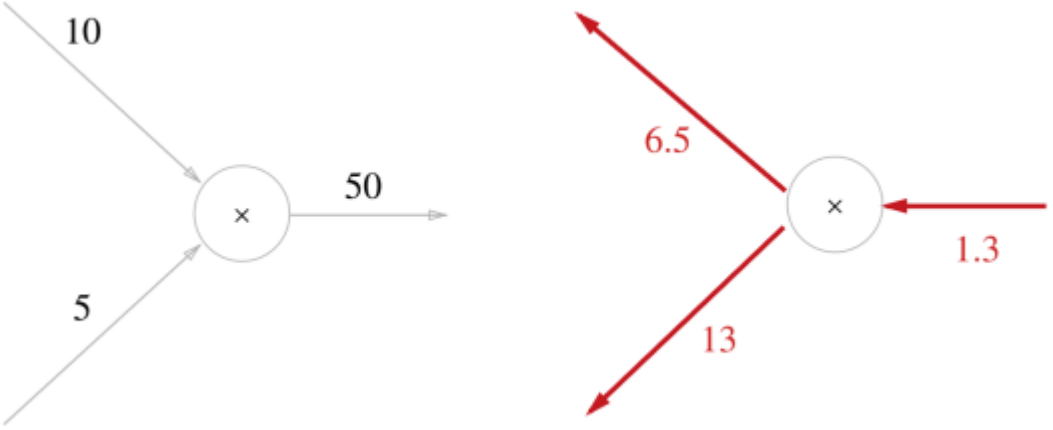
곱셈 노드의 역전파

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



곱셈 노드의 역전파는 상류의 값에 순전파 때의 입력 신호들을 서로 바꾼 값을 곱해서 하류로 흘려보낸다. 그 구체적인 예를 살펴보면 다음과 같다.

그림 5-13 곱셈 노드 역전파의 구체적인 예



덧셈의 역전파에서는 상류의 값을 그대로 흘려보내서 순방향 입력 신호의 값은 필요하지 않았었지만 곱셈의 역전파는 순방향 입력 신호의 값이 필요하다. 그렇기 때문에 곱셈 노드를 구현할 때는 순전파의 입력 신호를 변수에 저장해둔다.

단순한 계층 구현하기

곱셈 계층

모든 계층은 forward()와 backward()라는 공통의 메서드를 갖는다. forward()는 순전파, backward()는 역전파를 처리한다.

```
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
```

init()을 통해 변수 x,y를 초기화 한다. 이 변수들은 순전파 시의 입력값을 유지하기 위해 사용한다. forward()에서는 x,y를 입력받고 두 값을 곱해서 반환하는 반면 backward()에서는 상류에서 넘어온 미분(dout)에 순전파 떄의 값을 서로 바꿔 곱한 후 하류로 흘려보낸다.

이것을 이용하여 사과 문제를 풀면 다음과 같은 값을 얻을 수 있다.

```
from layer_naive import *

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
```

여기에서 주의할 점은 backward()의 호출 순서는 forward() 때와는 반대인 점이다. 또 backward()가 받는 인수는 '순전파의 출력에 대한 미분'이다.

덧셈 계층

덧셈 계층은 다음과 같이 구현할 수 있다.

```
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 필요없으므로 **init**() 메서드에서 pass를 통해 아무 것도 하지 않는다. 덧셈 계층에서의 forward()에서는 입력받은 두 인수 x,y를 더해서 반환한다. backward()에서는 상류에서 내려온 미분(dout)을 그대로 하류로 흘린다. 이 덧셈 계층을 통해 사과문제를 풀면 코드는 다음과 같다.

```
from layer_naive import *

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

```
tax = 1.1

# layer
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()

# forward
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, ora
nge_price)  # (3)
price = mul_tax_layer.forward(all_price, tax)  # (4)

# backward
dprice = 1
dall_price, dtax = mul_tax_layer.backward(dprice)  # (4)
dapple_price, dorange_price = add_apple_orange_layer.backwa
rd(dall_price)  # (3)
dorange, dorange_num = mul_orange_layer.backward(dorange_pr
ice)  # (2)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)
# (1)

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)
```

필요한 계층을 만들어 순전파 메서드인 forward()를 적절한 순서로 호출한다. 그 다음 순전파와 반대 순서로 역전파 메서드인 backward()를 호출하면 원하는 미분을 얻을 수 있다.

활성화 함수 계층 구현하기

ReLU 계층

활성화 함수로 사용되는 ReLU의 수식과 이를 미분한 식은 다음과 같다.

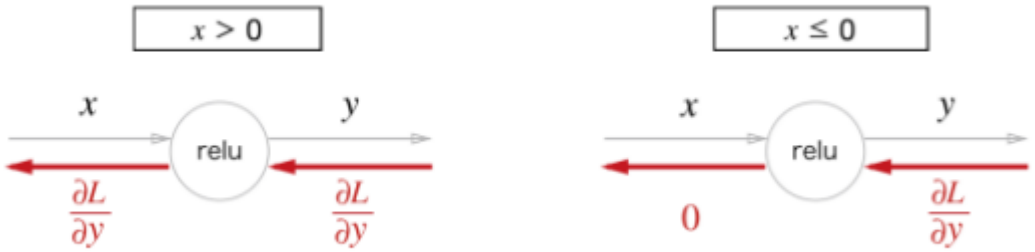
$$y = \begin{cases} x & (x > 0) \\ 0 & (x \le 0) \end{cases}$$

[식 5.7]에서 x에 대한 y의 미분은 [식 5.8]처럼 구합니다.

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

그리고 이것을 계산 그래프로 나타내면 다음과 같이 나타낼 수 있다.

그림 5-18 ReLU 계층의 계산 그래프



이러한 ReLU 함수를 코드로 구현해보면 다음과 같이 구현할 수 있다.

```
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 갖는다. mask는 True/False로 구성된 넘파이 배열로, 순전파의 입력인 x의 원소 값이 0 이하인 인덱스는 True, 그 외는 전부 False로 유지한다. 그러므로 순전파 때의 입력 값이 0 이하면 역전파 때의 값은 0이 되야 한다. 역전파 때는 순전파 때 만들어둔 mask를 써서 mask의 원소가 True인 곳에서는 상류에서 전파된 dout를 0으로 설정한다.

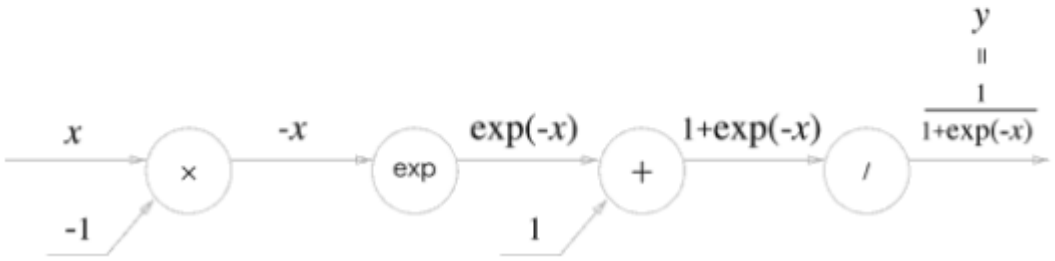
Sigmoid 계층

시그모이드 함수를 식과 계산그래프로 나타내면 다음과 같다.

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

[식 5.9]를 계산 그래프로 그리면 [그림 5-19]처럼 됩니다.

그림 5-19 Sigmoid 계층의 계산 그래프(순전파)



'exp' node는 y = exp(x) 계산을 수행하고 '/' node는 y = 1/x 계산을 수행한다.

역전파의 흐름을 한 단계씩 나눠보면 다음과 같다.

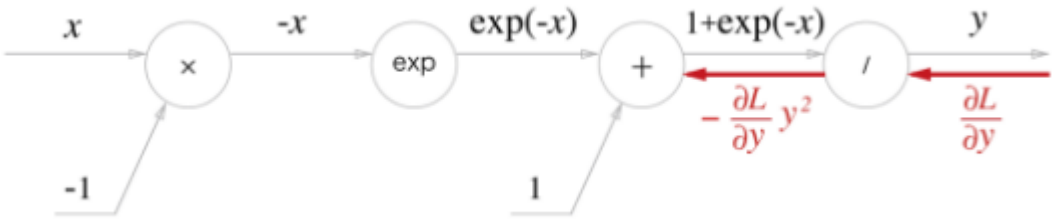
1단계

'/' 노드를 미분하면 다음 식이 된다.

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

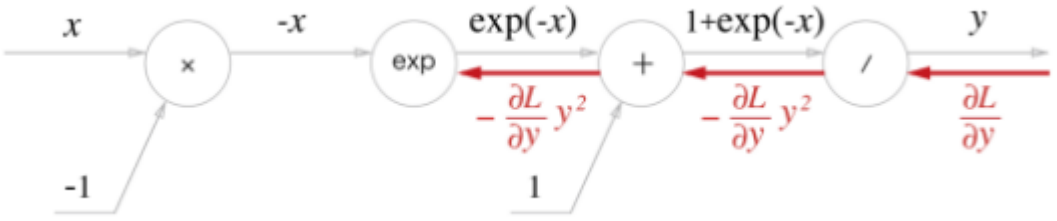
$$= -y^2$$

역전파 때는 상류에서 흘러온 값에 -y^2(순전파의 출력을 제곱한 후 -를 붙인 값)을 곱해서 하류로 전달한다.



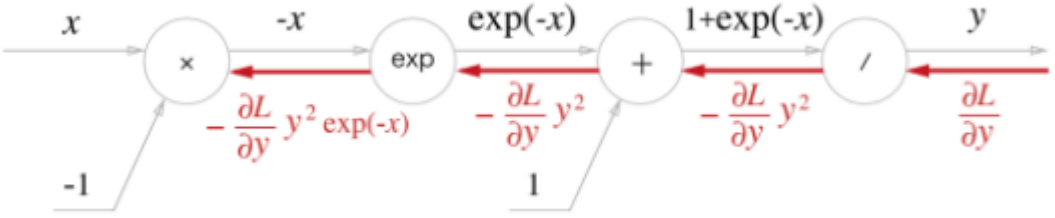
2단계

'+' 노드는 상류의 값을 여과 없이 하류로 내보내는 것이 다이다.



3단계

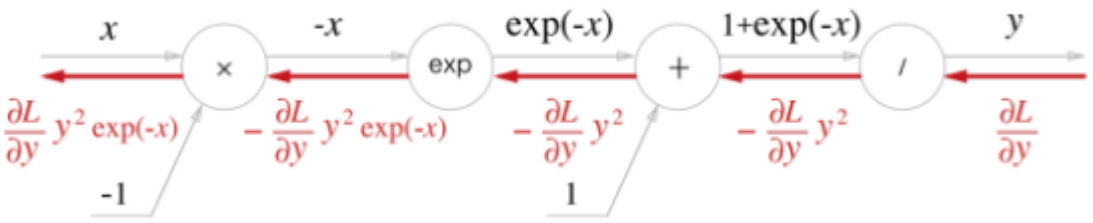
'exp' 노드는 y = exp(x) 연산을 수행한다. 계산 그래프에서는 상류의 값에 순전파 때의 출력을 곱해 하류로 전달한다.



4단계

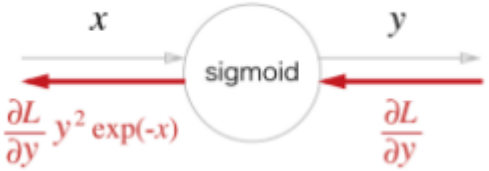
'x' 노드는 순전파 때의 값을 서로 바꿔서 곱한다.

그림 5-20 Sigmoid 계층의 계산 그래프



이 단계들을 간소화하면 다음과 같다.

그림 5-21 Sigmoid 계층의 계산 그래프(간소화 버전)



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

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

이 Sigmoid 계층을 구현하면 다음과 같다.

```
class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = sigmoid(x) # 순전파의 출력을 out에 보관한 후 역전파
계산 때 그 값을 사용
        self.out = out
        return out

    def backward(self, dout):
```

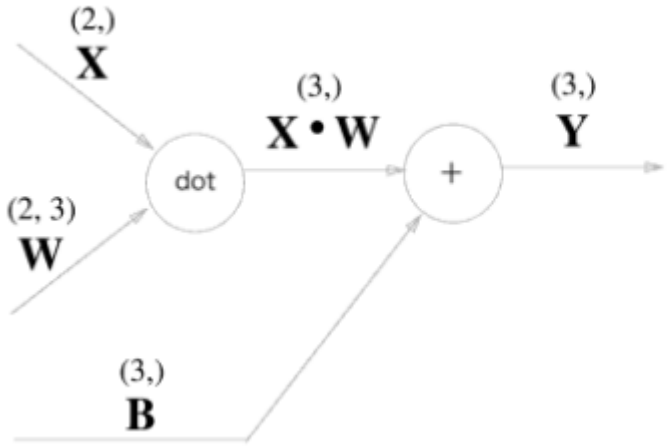
```
        dx = dout * (1.0 - self.out) * self.out

        return dx
```

Affine/Softmax 계층 구현

Affine 계층

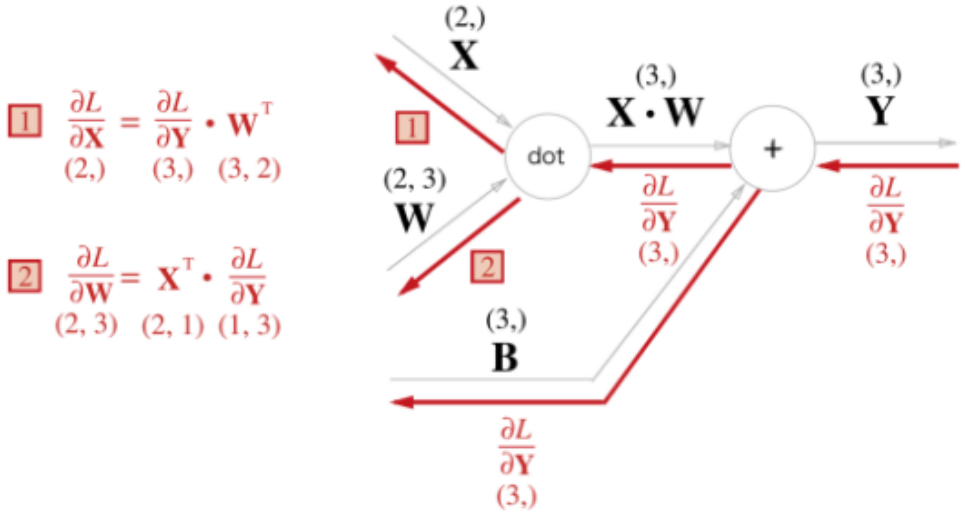
신경망의 순전파 때 수행하는 행렬의 곱은 기하학에서 affine transformation이라고 한다. 이 Affine 계층의 순전파를 계산 그래프로 나타내면 다음과 같다.

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



Affine 계층의 역전파를 계산그래프로 나타내면 다음과 같다.

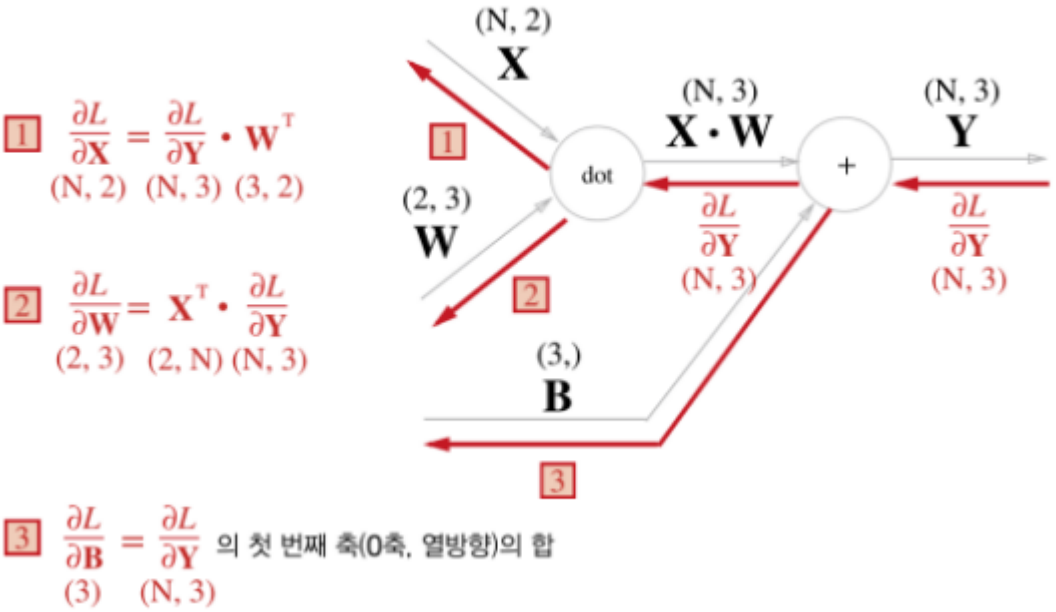
그림 5-25 Affine 계층의 역전파 : 변수가 다차원 배열임에 주의. 역전파에서의 변수 형상은 해당 변수명 아래에 표기했다.



배치용 Affine 계층

앞선 계산그래프들은 입력 데이터로 X 하나만을 고려한 것이다. 데이터 N개를 묶어 순전파하는 경우에 대해 계산 그래프를 그려보면 다음과 같다.

그림 5-27 배치용 Affine 계층의 계산 그래프



이것들을 이제 코드로 구현해보면 다음과 같이 나타내 볼 수 있다.

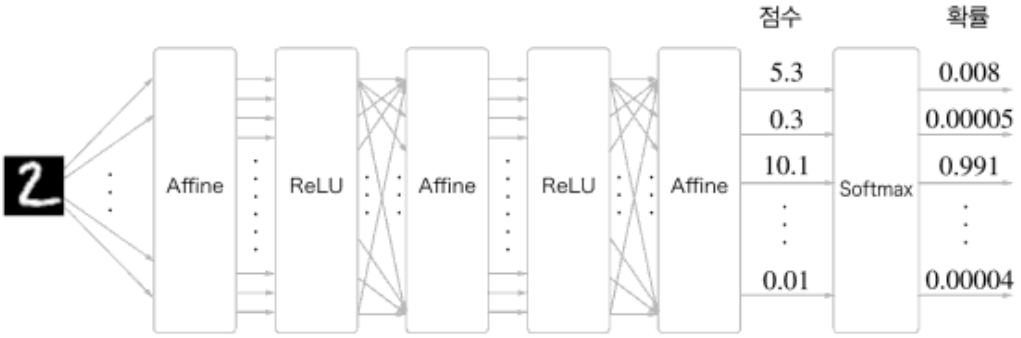
```
class Affine:
    def __init__(self, W, b):
        self.W = W
        self.b = b

        self.x = None
        self.original_x_shape = None
        # 가중치와 편향 매개변수의 미분
        self.dW = None
        self.db = None

    def forward(self, x):
        # 텐서 대응
        self.original_x_shape = x.shape
        x = x.reshape(x.shape[0], -1)
        self.x = x

        out = np.dot(self.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)

        dx = dx.reshape(*self.original_x_shape)  # 입력 데이
터 모양 변경(텐서 대응)
        return dx
```

Softmax-with-Loss 계층

softmax 함수는 입력 값을 정규화하여 출력한다. 예를 들어 손글씨 숫자 인식에서의 Softmax 계층의 출력은 다음과 같다.

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.

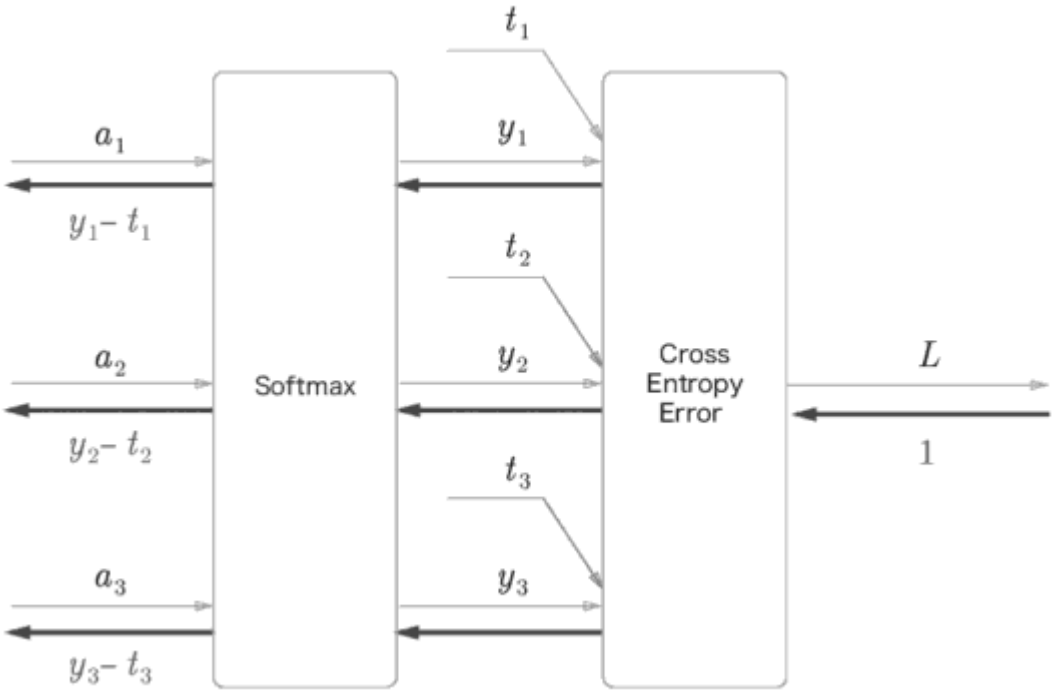


Softmax 계층은 입력값을 정규화(출력의 합이 1이 되도록 변형) 하여 출력한다.

신경망에서 수행하는 작업은 학습과 추론 두 가지이다. 추론할 때에는 일반적으로 Softmax 계층을 사용하지 않는다. [그림 5-28]의 신경망은 추론할 때는 마지막 Affine 계층의 출력을 인식 결과로 이용한다. 또한 신경망에서 정규화하지 않은 출력 결과에서는 Softmax 앞의 Affine 계를 점수라 한다. 다시말해서 신경망 추론에서 답을 하나만 내는 경우에는 가장 높은 점수만 알면 되므로 Softmax 계층은 필요 없다는 것이다. 반면 신경망을 학습할 때에는 Softmax 계층이 필요하게 된다.

Softmax-with-Loss 계층의 계산 그래프를 살표보면 다음과 같다.

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



Softmax 계층의 역전파는 y-t라는 결과를 내놓는다. y는 Softmax 계층의 출력이고 t는 정답 레이블이므로 y-t는 Softmax 계층의 출력과 정답 레이블의 차분인 것이다 . 다시말해서 신경망의 역전파에서는 이 차이인 오차가 앞 계층에 전해지는 것이다. 이는 신경망 학습의 중요한 성질이다. 그러나 신경망 학습의 목표는 신경망의 출력값이 정답 레이블과 가까워지도록 가중치 매개변수의 값을 조정하는 것이므로 오차를 효율적으로 앞 계층에 전달해야 한다. 이 Softmax-with-Loss를 구현한 코드는 다음과 같다.

```
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None # 손실함수
        self.y = None    # softmax의 출력
        self.t = None    # 정답 레이블(원-핫 인코딩 형태)

    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]
```

```
        if self.t.size == self.y.size: # 정답 레이블이 원-핫 인
코딩 형태일 때
            dx = (self.y - self.t) / batch_size
        else:
            dx = self.y.copy()
            dx[np.arange(batch_size), self.t] -= 1
            dx = dx / batch_size

        return dx
```

역전파 때는 전파하는 값을 배치의 수로 나눠서 데이터 1개당 오차를 앞 계층으로 전파한다.

오차역전파법 구현하기

신경망 학습의 전체 그림

단계별로 나누면 다음과 같다.

전체

신경망에는 적용 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 학습이라 한다.

1단계-미니배치

훈련 데이터 중 일부를 무작위로 가져오고 이를 미니배치라 한다. 미니배치의 손실 함수 값을 줄이는 것이 목표이다.

2단계-기울기 산출

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

3단계-매개변수 갱신

가중치 매개변수를 기울기방향으로 아주 조금 갱신한다

4단계-반복

1~3단계를 반복한다

이 중에서 오차역전파법은 2단계에서 등장한다. 오차역전파법을 이용하면 느린 수치 미분과 달리 기울기를 효율적이고 빠르게 구할수 있다.

오차역전파법을 적용한 신경망 구현

2층 신경망을 TwoLayerNet 클래스로 구현하고 주요 인스턴스 변수와 메서드를 정리하면 다음과 같다.

표 5-1 TwoLayerNet 클래스의 인스턴스 변수

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.

코드로 구현하면 다음과 같다.

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from common.layers import *
```

```
from common.gradient import numerical_gradient
from collections import OrderedDict


class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_siz
e, weight_init_std = 0.01):
        # 가중치 초기화
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.ran
dn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.ran
dn(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.param
s['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.param
s['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.param
s['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.param
s['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'].d
W, self.layers['Affine1'].db
        grads['W2'], grads['b2'] = self.layers['Affine2'].d
W, self.layers['Affine2'].db

        return grads
```

신경망의 계층을 OrderedDict에 보관하는 것이 중요하다. 순전파 때는 추가한 순서대로 각 계층의 forward() 메서드를 호출하기만 하면 처리가 완료된다. 마찬가지로 역전파 때는 계층을 반대 순서로 호출하기만 하면 된다. Affine 계층과 ReLU 계층이 각각 내부에서 순전파와 역전파를 처리하므로 여기서는 그냥 계층을 올바른 순서로 연결한 후 순서대로 호출해주면 된다.

오차역전파법으로 구한 기울기 검증하기

기울기는 수치 미분과 해석적으로 수식을 풀어 구하는 방법 2가지를 공부했다. 해석적인 방법은 오차역전파법을 이용하여 매개변수가 많아도 효율적으로 계산할 수 있다. 수치 미분의 이점은 구현하기 쉽고 오차가 적다. 오차역전파법은 구현하기 복잡해서 오차가 종종 나오기 때문에 둘을 비교하여 오차역전파법을 제대로 구현했는지 검증한다. 이렇게 두 방식으로 구한 기울기가 일치하는지 확인하는 작업을 기울기 확인 이라고 한다. 코드는 다음과 같다.

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 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_batc
h)
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_num
erical[key]) )
    print(key + ":" + str(diff))
```

MNIST 데이터 셋을 읽은 후 훈련 데이터 일부를 수치 미분으로 구한 기울기와 오차역전파법으로 구한 기울기의 오차를 확인한다.

오차역전파법을 사용한 학습 구현하기

```
import sys, os
sys.path.append(os.pardir)

import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 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, outpu
t_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)
```

이 코드를 실행하여 얻은 결과값은 다음과 같다.

```
0.9508 0.9494
0.9564166666666667 0.951
0.9606333333333333 0.9554
0.96455 0.9605
0.9671 0.9615
0.96915 0.9633
0.9715 0.9657
0.9713 0.9656
0.9757666666666667 0.9683
0.9767 0.9696
0.9778833333333333 0.97
0.97805 0.9705
```