

5. 오차역전파법

신경망의 가중치 매개변수의 기울기(정확히는 가중치 매개변수에 대한 손실 함수의 기울기)는 수치 미분을 사용해 구했다. 단순하지만 오래걸린다. 이 번장에서는 '오차역전파법(backpropagation)'을 배워보자.

이를 제대로 이해하는 방법은 두 가지가 있다.

하나는 수식을 통한 것이고, 다른 하나는 계산 그래프를 통한 것이다. 전자 쪽이 일반적인 방법으로, 기계 학습을 다루는 책 대부분은 수식을 중심으로 이야기를 전개한다. 확실히 수식을 사용한 설명은 정확하고 간결하므로 올바른 방법이다. 하지만 어렵다. 이에 이번 장에서는 계산 그래프를 사용해서 '시각적'으로 이해해보자. 그런 다음 실제로 코드를 실행시켜보자.

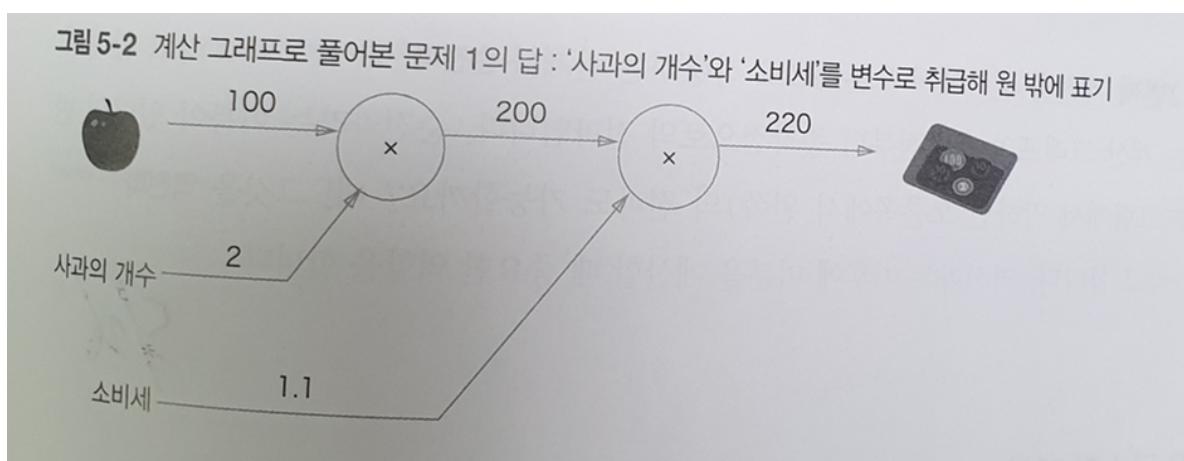
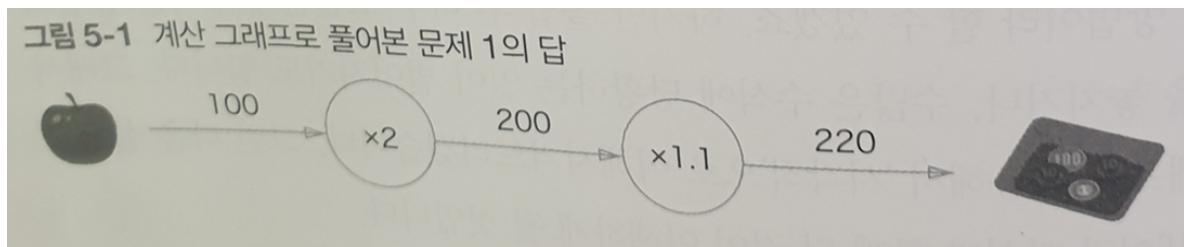
5.1 계산 그래프

계산 그래프는 계산 과정을 그래프로 나타낸 것이다. 여기에서의 그래프는 우리가 잘 아는 그래프 자료구조로, 복수의 노드와 에지로 표현된다(노드 사이의 직선을 에지라고 한다).

5.1.1 계산 그래프로 풀다.

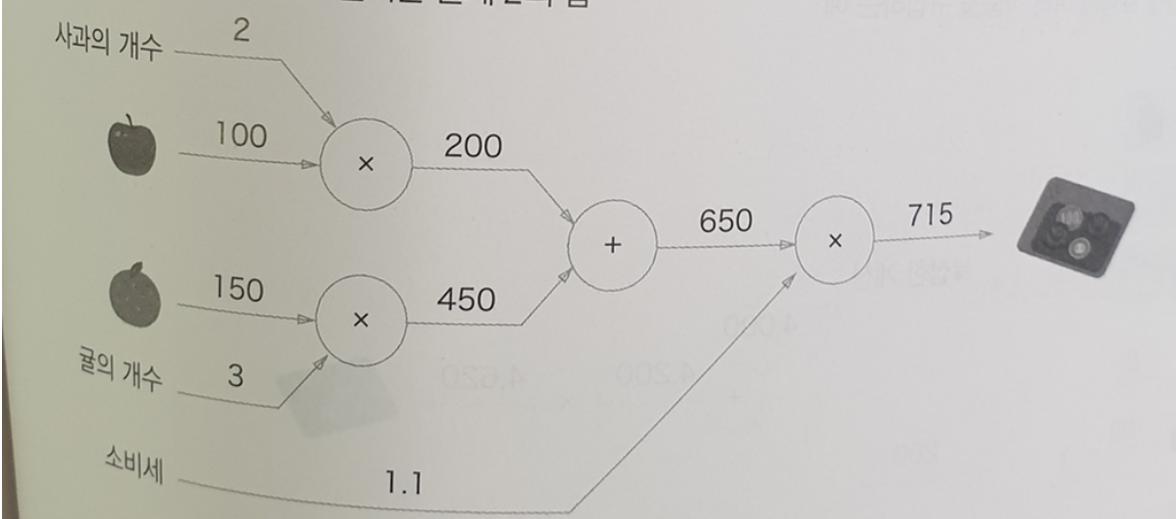
문제 1: 현빈 군은 슈퍼에서 1개에 100원인 사과를 2개 샀다. 이때 지불 금액을 구하라. 단 소비세가 10% 부과된다.

계산 그래프는 계산 과정을 노드와 화살표로 표현한다. 노드는 원(O)으로 표기하고 원 안에 연산 내용을 적는다. 다음과 같다.



문제 2: 현빈 군은 슈퍼에서 사과를 2개, 블루베리 3개 샀다. 사과는 1개에 100원, 블루베리는 1개 150원이다. 소비세가 10% 일 때 지불 금액을 구하라.

그림 5-3 계산 그래프로 풀어본 문제 2의 답



이 문제는 덧셈 노드인 '+'가 새로 등장하여 사과와 귤의 금액을 합산했다.

지금까지 살펴본 것처럼 계산 그래프를 이용한 문제풀이는 다음 흐름으로 진행한다.

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

여기서 2번째 '계산을 오른쪽으로 진행'하는 단계를 **순전파**라고 한다. 순전파는 계산 그래프의 출발점부터 종착점으로의 전파이다. 순전파도 있으면 역전파도 있다. 역전파는 이후에 미분을 계산할 때 중요한 역할을 한다.

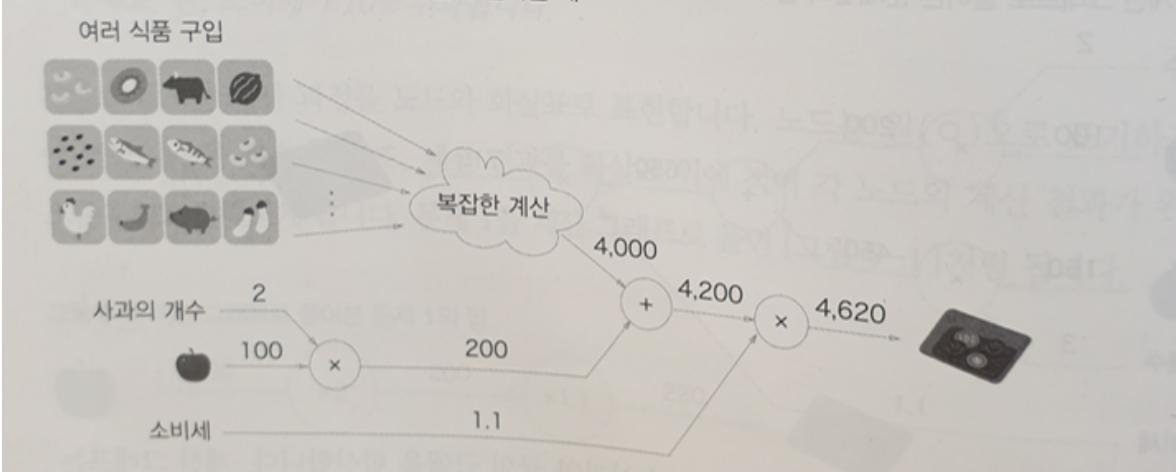
5.1.2 국소적 계산

계산 그래프의 특징은 '국소적 계산'을 전파함으로써 최종 결과를 얻는다는 점에 있다. 국소적이란 '자신과 직접 관계된 작은 범위'라는 뜻이다. 국소적 계산은 결국 전체에서 어떤 일이 벌어지는 상관없이 자신과 관계된 정보만으로 결과를 출력할 수 있다는 것이다.

국소적 계산을 구체적인 예를 들어 설명하겠다. 가령 슈퍼마켓에서 사과 2개를 포함한 여러 식품을 구입하는 경우를 생각해보자.

아래와 같은 계산 그래프로 나타낼 수 있겠다.

그림 5-4 사과 2개를 포함해 여러 식품을 구입하는 예



위 그림에서는 여러 식품을 구입하여 (복잡한 계산을 거쳐) 총금액이 4,000원이 됐다. 여기에서 핵심은 각 노드에서의 계산은 국소적 계산이라는 점이다. 가령 사과와 그 외의 물품 값을 더하는 계산($4,000 + 200 \rightarrow 4,200$)은 4,000이라는 숫자가 어떻게 계산되었느냐와는 상관없이, 단지 두 숫자를

더하면 된다는 뜻이다. 각 노드는 자신과 관련한 계산(이 예에서는 입력된 두 숫자의 덧셈) 외에는 아무것도 신경 쓸 게 없다. 이처럼 계산 그래프는 국소적 계산에 집중한다. 전체 계산이 제 아무리 복잡하더라도 각 단계에서 하는 일은 해당 노드의 '국소적 계산'이다. 국소적인 계산은 단순하지만, 그 결과를 전달함으로써 전체를 구성하는 복잡한 계산을 해낼 수 있다.

Note 비유하자면 복잡한 자동차 조립은 일반적으로 '조립 라인 작업'에 의한 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 라인 작업을 수행하며 계산 결과를 다음 노드로 전달한다. 복잡한 계산도 분해하면 단순한 계산으로 구성된다는 점은 자동차 조립과 마찬가지인 것이다.

5.1.3 왜 계산 그래프로 푸는가?

지금까지 계산 그래프를 써서 두 문제를 풀어봤다. 계산 그래프의 이점은 무엇인가? 이점 하나는 방금 설명한 '국소적 계산'이다. 전체가 아무리 복잡해도 각 노드에서는 단순한 계산에 집중하여 문제를 단순화 할 수 있다. 또 다른 이점으로, 계산 그래프는 중간 계산 결과를 모두 보관할 수 있다. 예를 들어 사과 2개 까지 계산했을 때의 금액은 200원, 소비세를 더하기 전의 금액은 650원 인식이다. 그러나 이정도만으로 계산 그래프를 사용하는 이유가 충분히 와닿지 않을지 모른다.

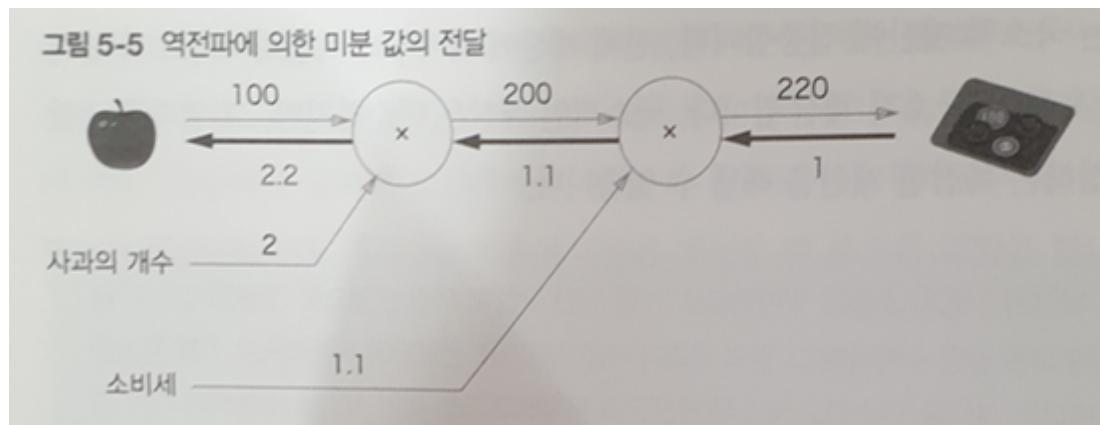
실제 계산 그래프를 사용하는 가장 큰 이유는 역전파를 통해 '미분'을 효율적으로 계산할 수 있는 점에 있다.

계산 그래프의 역전파를 설명하기 위해 문제 1을 다시 꺼내보겠다. 문제 1은 사과를 2개 사서 소비세를 포함한 최종 금액을 구하는 것이었다. 여기서 가령 사과 가격이 오르면 최종 금액에 어떤 영향을 끼치는지를 알고 싶다고 하자. 이는 '사과 가격에 대한 지불 금액의 미분'을 구하는 문제에 해당한다. 기호로 나타내면 사과 값을 x , 지불 금액을 L 이라고 했을 때 $\frac{\partial L}{\partial x}$ 를 구하는 것이다. 이 미분 값은 사과 값이 '아주 조금' 올랐을 때 지불 금액이 얼마나 증가하느냐를 표시한 것이다.

앞에서 말했듯이 '사과 가격에 대한 지불 금액의 미분' 같은 값은 계산 그래프에서 역전파를 하면 구할 수 있다. 먼저 결과만을 나타내면 다음 그림처럼 계산 그래프 상의 역전파에 의해 미분을 구할 수 있다.

위 그림과 같이 역전파는 순전파와는 반대 방향으로 화살표(굵은 선)으로 그린다. 이 전파는 '국소적 미분'을 전달하고 그 미분 값은 화살표의 아래에 적는다. 이 예에서 역전파는 오른쪽에서 인쪽으로 '1->1.1->2.2' 순으로 미분 값을 전달한다. 이 결과로부터 '사과 가격에 대한 지불 금액의 미분'값은 2.2라 할 수 있다. 시과가 1원 오르면 최종 금액은 2.2원 오른다는 뜻이다(정확히는 사과 값이 아주 조금 오르면 최종 금액은 그 아주 작은 값의 2.2배 만큼 오른다는 뜻이다).

여기에서는 사과 값에 대한 미분만 구했지만, '소비세에 대한 지불 금액의 미분'이나 '사과 개수에 대한 지불 금액의 미분'도 같은 순서로 구할 수 있다. 그리고 그때는 중간까지 구한 미분 결과를 공유할 수 있어서 다수의 미분을 효율적으로 계산할 수 있다. 이처럼 계산 그래프의 이점은 순전파와 역전파를 활용해서 각 변수의 미분을 효율적으로 구할 수 있다는 것이다.

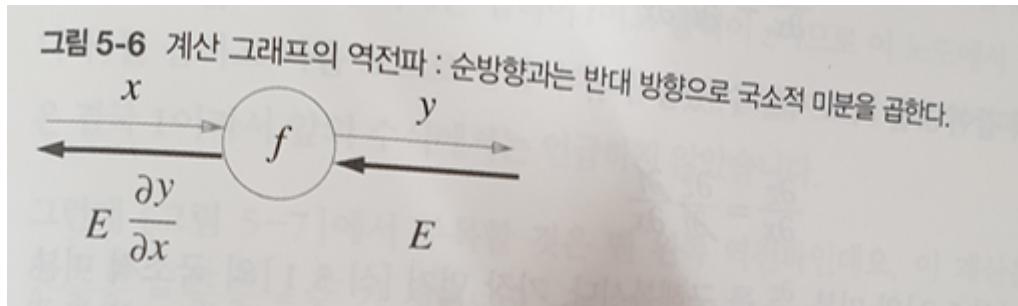


5.2 연쇄법칙

그동안 해온 계산 그래프의 순전파는 계산 결과를 왼쪽에서 오른쪽으로 전달했다. 이 순서는 평소 하는 방식이니 자연스럽게 느껴졌을 것이다. 한편 역전파는 '국소적인 미분'을 순방향과는 반대인 오른쪽에서 왼쪽으로 전달한다. 또한, 이 '국소적 미분'을 전달하는 원리는 연쇄법칙에 따른 것이다. 이 번 절에서는 연쇄법칙을 설명하고 그것이 계산 그래프 상의 역전파와 같다는 사실을 밝히겠다.

5.2.1 계산 그래프의 역전파

서둘러 계산 그래프로 사용한 역전파의 예를 하나 살펴보자. $y = f(x)$ 라는 계산의 역전파를 다음 그림으로 그려봤다.



위 그림과 같이 역전파의 계산 절차는 신호 E 에 노드의 국소적 미분($\frac{\partial y}{\partial x}$)을 곱한 후 다음 노드로 전달하는 것이다. 여기에서 말하는 국소적 미분은 순전파 때의 $y = f(x)$ 계산의 미분을 구한다는 것이며, 이는 x 에 대한 y 의 미분($\frac{\partial y}{\partial x}$)을 구한다는 뜻이다. 가령 $y = f(x) = x^2$ 이라면 $\frac{\partial y}{\partial x} = 2x$ 가 된다.

그리고 이 국소적인 미분을 상류에서 전달된 값(이 예에서는 E)에 곱해 앞쪽 노드로 전달하는 것이다.

이것이 역전파의 계산 순서인데, 이러한 방식을 따르면 목표로 하는 미분 값을 효율적으로 구할 수 있다 는 것이 이 전파의 핵심이다. 왜 그런 일이 가능한가는 연쇄법칙의 원리로 설명할 수 있다.

5.2.2 연쇄법칙이란?

연쇄법칙을 설명하려면 우선 합성 함수 이야기부터 해야한다. 연쇄법칙은 합성 함수의 미분에 대한 성질이며, 예시로 다음 함수식을 들어 설명하겠다.

$$\begin{aligned} z &= t^2 \\ t &= x + y \\ \text{식 } 5.1 \end{aligned}$$

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

이것이 연쇄법칙의 원리이다. 언뜻 어렵게 보일지도 모르지만 간단한 성질이다.

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

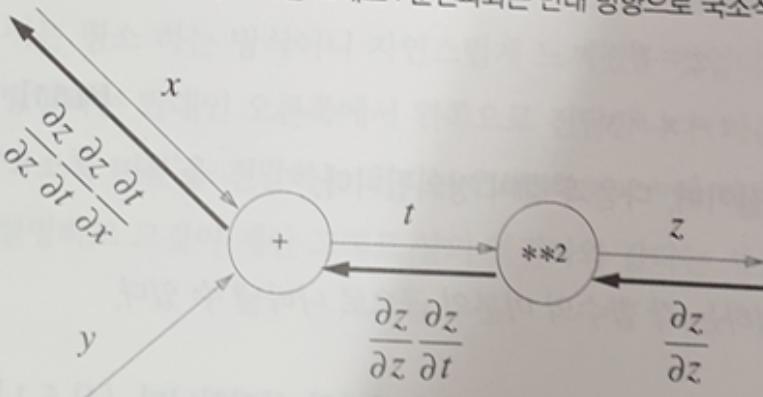
그렇다면 대입시켜 보자.

$$\begin{aligned} \frac{\partial z}{\partial t} &= 2t \\ \frac{\partial t}{\partial x} &= 1 \\ \frac{\partial z}{\partial x} &= \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} = 2t = 2(x + y) \quad \text{식 } 5.4 \end{aligned}$$

5.2.3 연쇄법칙과 계산 그래프

다음 그림은 식 5.4를 도식화 한 것이다. 2제곱 계산을 '**2' 노드로 나타내면 아래와 같다.

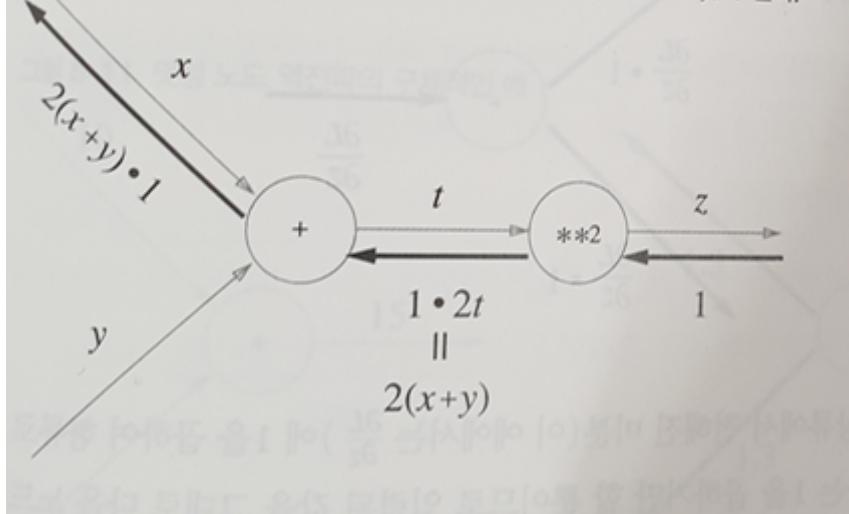
그림 5-7 [식 5.4]의 계산 그래프 : 순전파와는 반대 방향으로 국소적 미분을 곱하여 전달한다.



위와 같이 계산그래프의 역전파는 오른쪽에서 왼쪽으로 신호를 전파한다. 역전파의 계산 절차에서는 노드로 들어온 입력 신호에 그 노드의 국소적 미분(편미분)을 곱한 후 다음 노드로 전달한다. 예를 들어 '**2'노드에서의 역전파를 보자. 입력은 $\frac{\partial z}{\partial z}$ 이며, 이에 국소적 미분인 $\frac{\partial z}{\partial t}$ (순전파 시에는 입력이 t이고 출력이 z이므로 이 노드에서 (국소적)미분은 $\frac{\partial z}{\partial t}$ 이다)를 곱하고 다음 노드로 넘긴다.

위 그림에서 주목해야 할 것은 맨 왼쪽 역전파이다. 이 계산은 연쇄법칙에 따르면 $\frac{\partial z}{\partial z} \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} = \frac{\partial z}{\partial x}$ 가 성립되어 'x에 대한 z의 미분'이 된다. 즉, 역전파가 하는 일은 연쇄법칙의 원리와 같다는 것이다.

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



5.3 역전파

앞 절에서는 계산 그래프의 역전파가 연쇄법칙에 따라 진행되는 모습을 설명했다. 이번 절에서는 '+', 'x' 등의 연산을 예로 들어 역전파의 구조를 설명한다.

5.3.1 덧셈 노드의 역전파

먼저 덧셈 노드의 역전파이다. 여기에서는 $z = x+y$ 라는 식을 대상으로 그 역전파를 살펴보겠다.

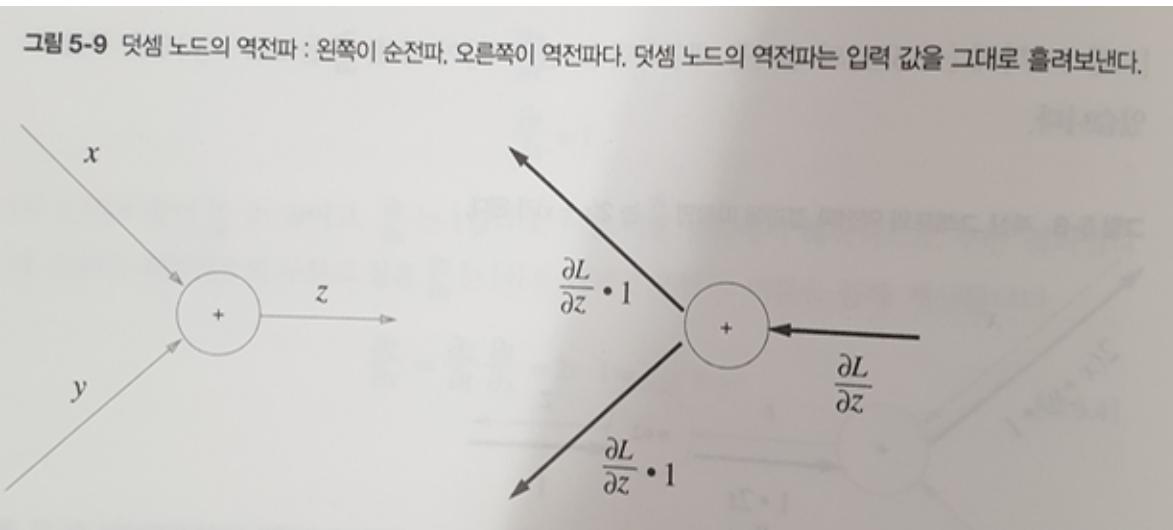
우선 $z = x+y$ 의 미분은 다음과 같이 해석적으로 계산할 수 있다.

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

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

식 5.5

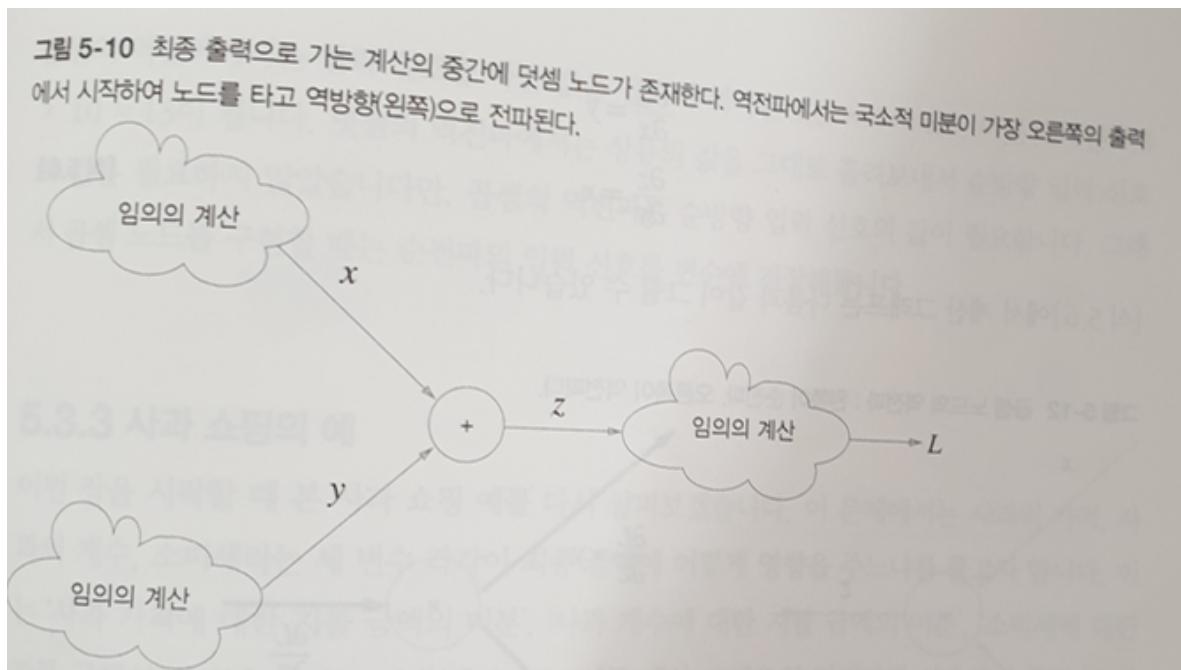
위와 같이 $\frac{\partial z}{\partial x}$ 와 $\frac{\partial z}{\partial y}$ 는 모두 1이 된다. 이를 계산 그래프로는 다음과 같이 그릴 수 있다.



위 그림과 같이 역전파 때는 상류에서 전해진 미분(이 예에서는 $\frac{\partial L}{\partial z}$)에 1을 곱하여 하류로 흘린다.

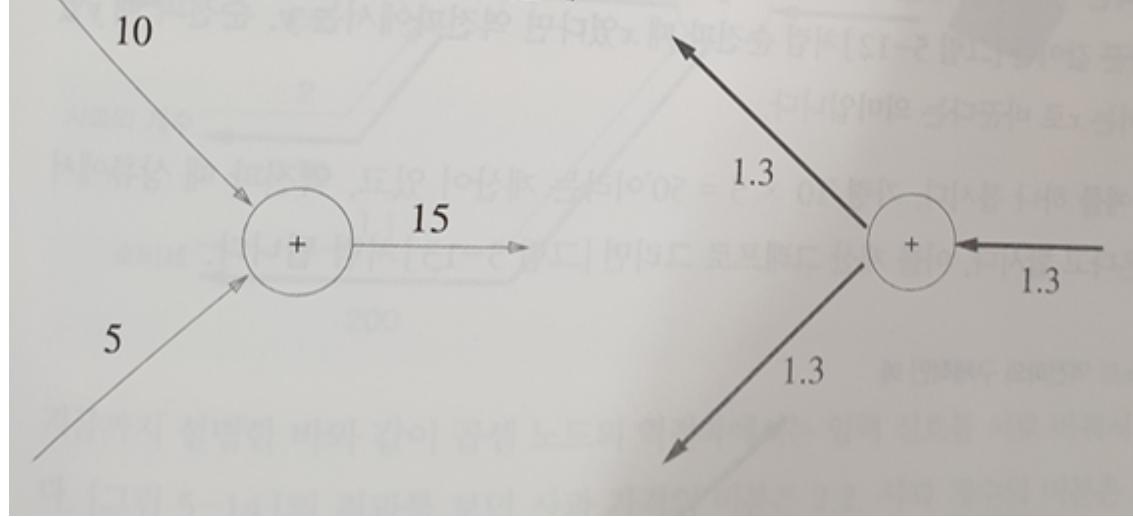
즉, 덧셈 노드의 역전파는 1을 곱하기만 할 뿐이므로 입력된 값을 그대로 다음 노드로 보낸다.

이 예에서는 상류에서 전해진 미분 값은 $\frac{\partial L}{\partial z}$ 이라 했는데, 이는 다음 그림과 같이 최종적으로 L 이라는 값 출력하는 큰 계산 그래프를 가정하기 때문이다. $z = x + y$ 계산은 그 큰 계산 그래프의 중간 어딘가에 존재하고, 상류로부터 $\frac{\partial L}{\partial z}$ 값이 전해진 것이다. 그리고 다시 하류로는 그리고 다시 하류로는 $\frac{\partial L}{\partial x}$ 와 $\frac{\partial L}{\partial y}$ 값들을 전달하는 것이다.



이제 구체적인 예를 하나 살펴보자. 가령 '10+5=15'라는 계산이 있고, 상류에서 1.3이라는 값이 흘러나온다.

그림 5-11 덧셈 노드 역전파의 구체적인 예



덧셈 노드 역전파는 입력 신호를 다음 노드로 출력할 뿐이므로 1.3을 그대로 다음 노드로 전달한다.

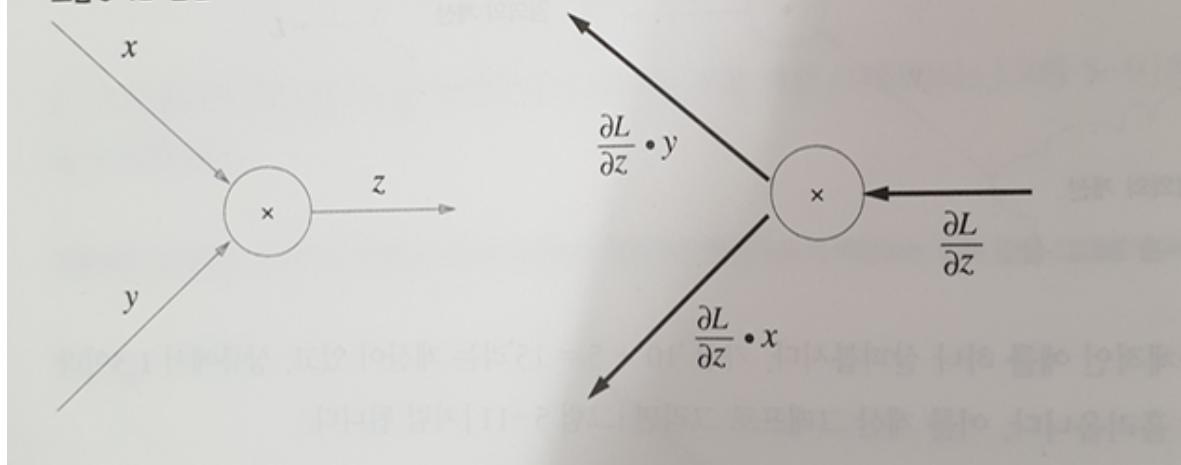
5.3.2 곱셈 노드의 역전파

$z = xy$ 라는 식을 생각해보자. 이 식의 미분은 다음과 같다.

$$\begin{aligned}\frac{\partial z}{\partial x} &= y \\ \frac{\partial z}{\partial y} &= x\end{aligned}$$

식 5.6

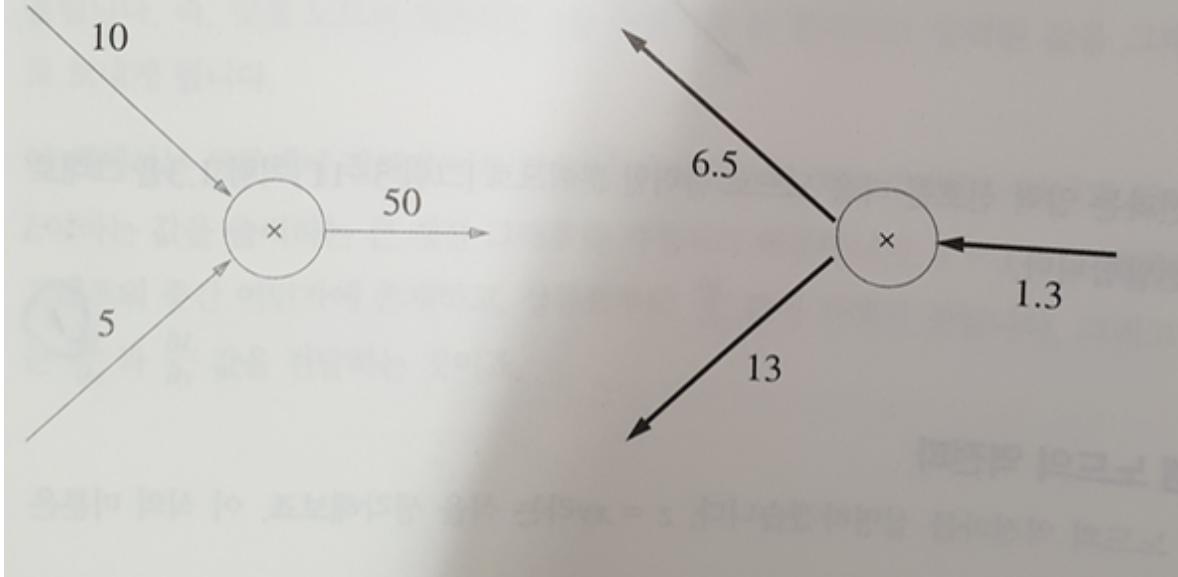
그림 5-12 곱셈 노드의 역전파 : 왼쪽이 순전파, 오른쪽이 역전파다.



곱셈 노드 역전파는 상류의 값에 순전파 때의 입력 신호들을 '서로 바꾼 값'을 곱해서 하류로 보낸다. 서로 바꾼 값이란 위 그림처럼 순전파 때 x 였다면 역전파에서는 y , 순전파 때 y 였다면 역전파에서는 x 로 바꾼다는 의미이다.

그럼 구체적인 예를 하나 보자. 가령 ' $10 \times 5 = 50$ '이라는 계산이 있다. 역전파 때 상류에서 1.3 값이 흘러나온다고 하자.

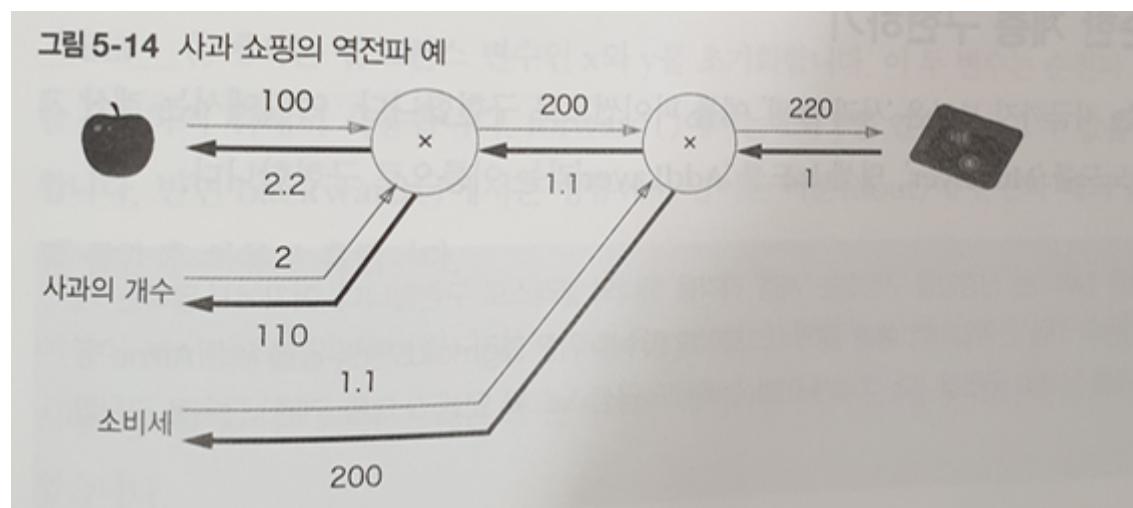
그림 5-13 곱셈 노드 역전파의 구체적인 예



곱셈의 역전파에서는 입력 신호를 바꾼 값을 곱하여 하나는 $1.3 \times 5 = 6.5$, 다른 하나는 $1.3 \times 10 = 13$ 이 된다. 덧셈의 역전파에서는 상류의 값을 그대로 흘려보내서 순방향 입력 신호의 값은 필요하지 않았지만, 곱셈의 역전파는 손방향 입력 신호의 값이 필요하다. 그래서 곱셈 노드를 구현할 때는 순전파의 입력 신호를 변수에 저장한다.

5.3.3 사과 쇼핑의 예

사과 쇼핑의 예를 다시 살펴보자. 이 문제에서는 사과의 가격, 사과의 개수, 소비세라는 세 변수 각각이 최종 금액에 어떻게 영향을 주느냐를 풀고자 한다. 이는 '사과 가격에 대한 지불 금액의 미분', '사과 개수에 대한 지불 금액의 미분', '소비세에 대한 지불 금액의 미분'을 구하는 것에 해당한다. 이를 계산 그래프의 역전파를 사용해서 풀면 다음 그림처럼 된다.



지금까지 설명한 바와 같이 곱셈 노드의 역전파에서는 입력 신호를 서로 바꿔서 하류로 흘린다. 위의 결과를 보면 사과 가격의 미분은 2.2, 사과 개수의 미분은 110, 소비세의 미분은 200이다. 이는 소비세와 사과 가격이 같은 양만큼 오르면 최종 금액에는 소비세가 200의 크기로, 사과 가격이 2.2크기로 영향을 준다고 해석할 수 있다. 이 예에서 소비세와 사과 가격은 단위가 다르니 주의해야 한다.

(소비세 1은 100%, 사과 가격 1은 1원)

정리할 겸, 마지막으로 '사과와 굴 쇼핑'의 역전파를 풀어보자.

5.4 단순한 계층 구현하기

이번 절에서는 지금까지 보아온 '사과 쇼핑'예를 파이썬으로 구현한다. 여기에서는 계산 그래프의 곱셈 노드를 'MulLayer', 덧셈 노드를 'AddLayer'라는 이름으로 구현한다.

Note 다음 절에서는 신경망을 구성하는 '계층' 각각을 하나의 클래스로 구현한다. 여기에서 말하는 '계층' 이란 신경망의 기능 단위이다. 예를 들어 시그모이드 함수를 위한 Sigmoid, 행렬 곱을 위한 Affine 등 의 기능을 계층 단위로 구현한다. 그래서 이번 절에서도 곱셈 노드와 덧셈 노드를 '계층' 단위로 구현한다.

5.4.1 곱셈 계층

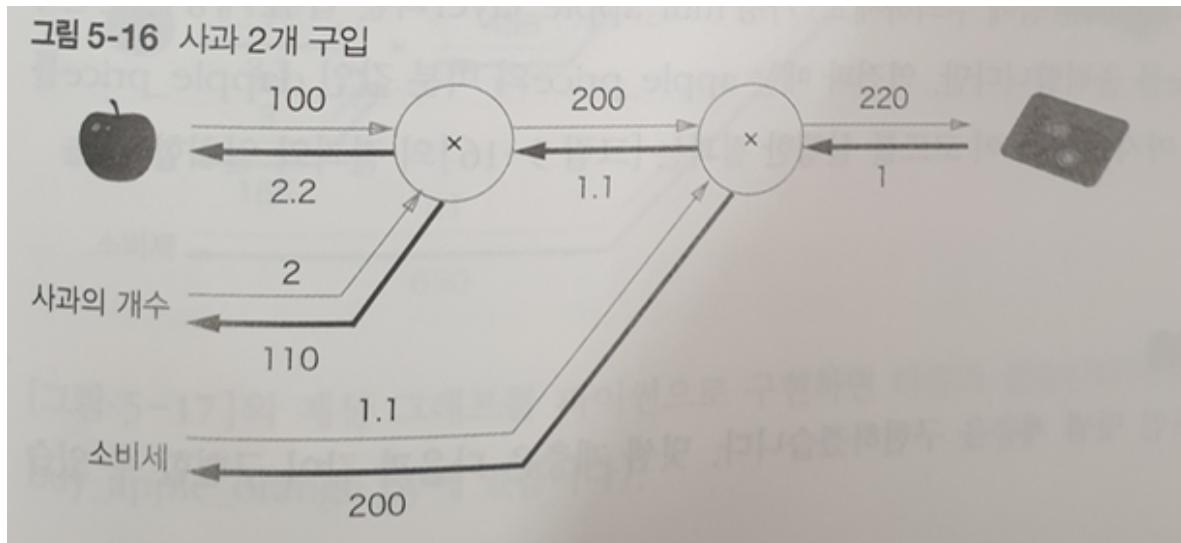
모든 계층은 forward()와 backward()라는 공통의 메서드(인터페이스)를 갖도록 구현할 것이다.

그럼 먼저 곱셈 계층을 구현해보자. 곱셈 계층은 MulLayer라는 이름의 클래스로 다음과 같이 구현할 수 있다.

```
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__()` 에서는 인스턴스 변수인 `x`와 `y`를 초기화한다. 이 두 변수는 순전파 시의 입력값을 유지하기 위해서 사용한다. `forward()`에서는 `x`와 `y`를 인수로 받고 둘 값을 곱해서 반환한다. 반면 `backward()`에서는 상류에서 넘어온 미분(`dout`)에 순전파 때의 값을 '서로 바꿔' 곱한 후 하류로 흘린다.

이 `MulLayer`를 구현하자.



`MulLayer`를 사용하여 위 그림의 순전파를 다음과 같이 구현할 수 있다.

```

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)

```

또, 각 변수에 대한 미분은 backward()에서 구할 수 있다.

```

#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)

price: 220
dApple: 2.2
dApple_num: 110
dTax: 200

```

backward() 호출 순서는 forward() 때와는 반대이다. 또, backward()가 받는 인수는 '순전파의 출력에 대한 미분' 임에 주의해라. 가령 mul_apple_layer라는 곱셈 계층은 순전파는 apple_price를 출력하지만, 역전파 때는 apple_price의 미분 값인 dapple_price를 인수로 받는다.

5.4.2 덧셈 계층

이어서 덧셈 노드인 덧셈 계층을 구현하자.

```

class AddLayer :
    def __init__(self):
        pass

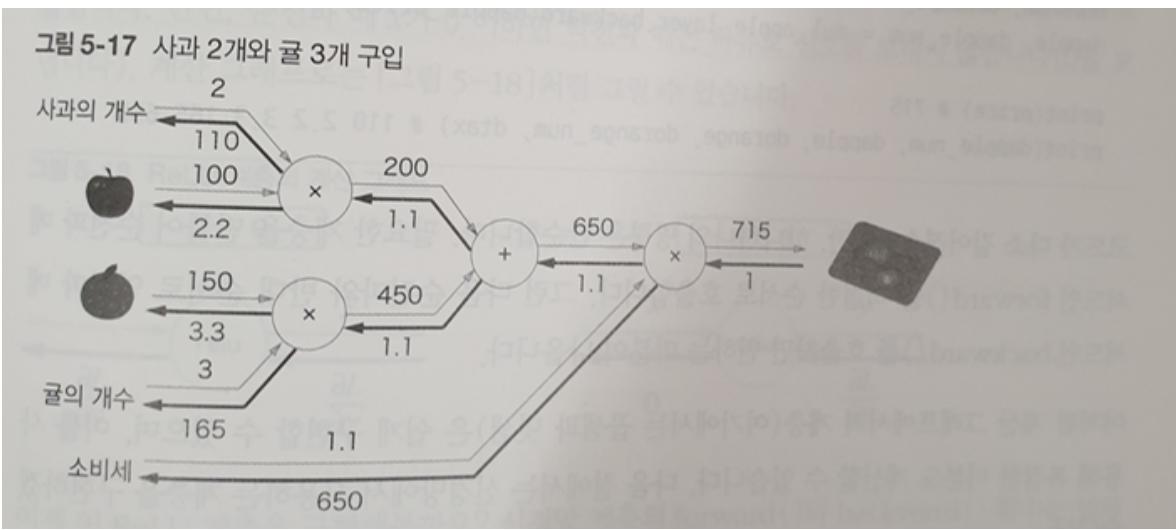
    def forward(self, x,y):
        out = x+y
        return out

    def backward(self, dout):
        dx = dout * 1
        dy = dout * 1
        return dx,dy

```

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

이상의 덧셈 계층과 곱셈 계층을 사용하여 사과 2개와 귤 3개를 사는 아래 그림을 구현해보자.



```

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(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(dapple_price) #(3)
dorange, dorange_num = mul_orange_layer.backward(dorange_price) #(2)
dapple, dapple_num = mul_apple_layer.backward(dapple_price) #(1)

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

#result
In [6]: %run buy_apple_orange.py
price: 715
dApple: 2.2
dApple_num: 110
dOrange: 3.3000000000000003
dOrange_num: 165
dTax: 650

```

코드가 다소 길어졌지만, 하나하나의 명령은 단순하다. 필요한 계층을 만들어 순전파 메서드인 forward()를 적절한 순서로 호출한다. 그런 다음 순전파와 반대 순서로 역전파 메서드인 backward()를 호출하면 원하는 미분이 나온다.

이처럼 계산 그래프에서의 계층(여기에서는 곱셈과 곱셈과 덧셈)은 쉽게 구현할 수 있으며, 이를 사용해 복잡한 미분도 계산할 수 있다.

5.5 활성화 함수 계층 구현하기

드디어 계산 그래프를 신경망에 적용할 때가 왔다. 여기에서는 신경망을 구성하는(계층) 각각을 클래스 하나로 구현한다. 우선은 활성화 함수인 ReLU와 Sigmoid 계층을 구현하겠다.

5.5.1 ReLU 계층

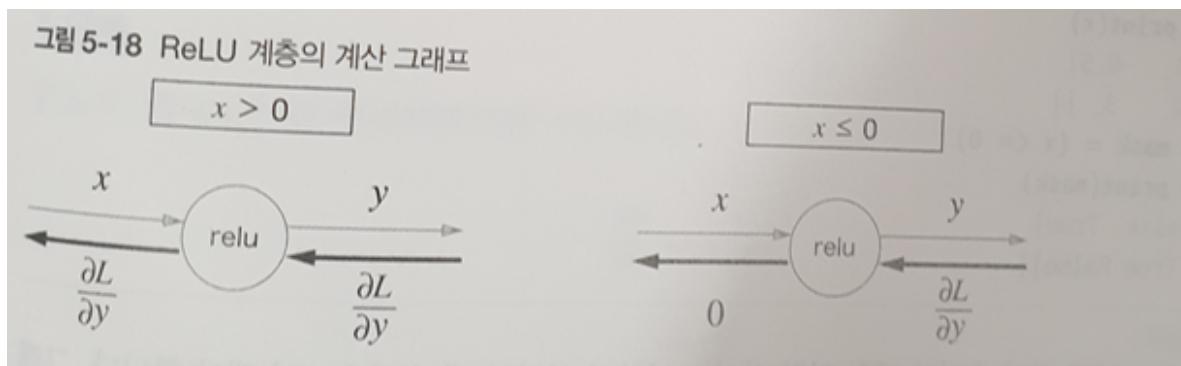
활성화 함수로 사용되는 ReLU의 수식은 다음과 같다.

$$y = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$
 식(5.7)

식 5.7에서 x 에 대한 y 의 미분은 다음 식처럼 구한다.

$$\frac{\partial y}{\partial x} = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$
 식(5.8)

식 5.8에서와 같이 순전파 때의 입력인 x 가 0보다 크면 역전파는 상류의 값을 그대로 하류로 흘린다. 반면, 순전파 때 x 가 0이하면 역전파 때는 하류로 신호로 보내지 않는다(0을 보낸다). 계산 그래프로는 다음처럼 그릴 수 있다.



이제 이 ReLU 계층을 구현해볼까? 신경망 계층의 forward()와 backward() 함수는 넘파이 배열을 인수로 받는다고 가정한다.

```
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, 그 외(0보다 큰 원소)는 False로 유지한다. 예컨대 mask 변수는 다음 예와 같이 T/F로 구성된 넘파이 배열을 유지한다.

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

In [16]: print(x)
[[ 1. -0.5]
 [-2.  3. ]]

In [17]: mask = (x<=0)

In [18]: print(mask)
[[False  True]
 [ True False]]
```

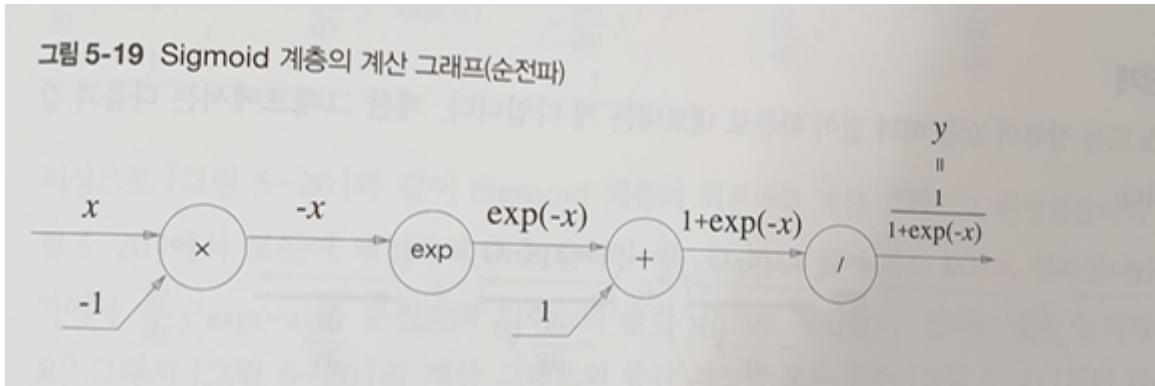
위 그림과 같이 순전파 때의 입력 값이 0 이하면 역전파 때의 값은 0이 돼야 한다. 그래서 역전파 때는 순전파 때 만들어 둔 mask를 써서 mask의 원소가 True인 곳에는 상류에서 전파된 dout을 0으로 설정한다.

5.5.2 Sigmoid 계층

다음은 시그모이드 함수 차례다. 시그모이드 함수는 다음 식을 의미하는 함수다.

$$y = \frac{1}{1 + \exp(-x)} \quad \text{식 (5.9)}$$

위 식을 그래프로 그리면 다음과 같다.



위 그림에는 'exp'와 '/' 노드가 새롭게 등장했다.

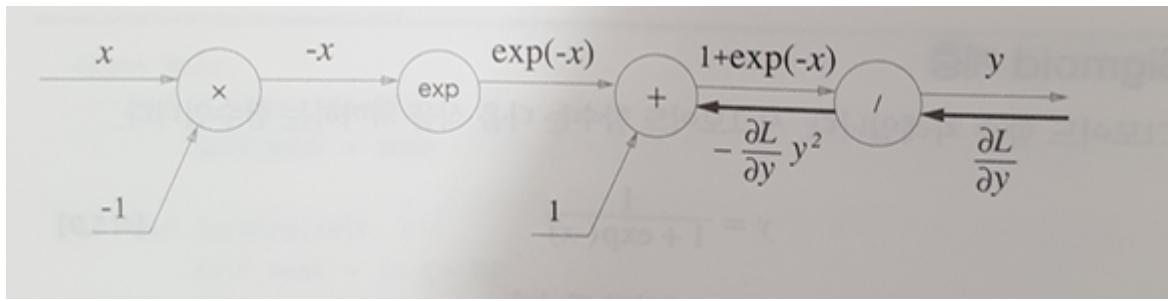
역전파의 흐름을 오른쪽에서 왼쪽으로 한 단계씩 짚어보겠다.

1단계

'/' 노드, 즉 $y = \frac{1}{x}$ 을 미분하면 다음 식이 된다.

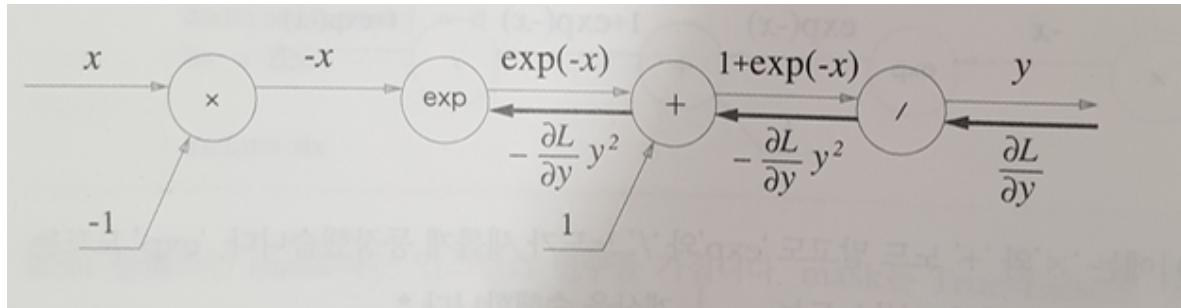
$$\begin{aligned} \frac{\partial y}{\partial x} &= -\frac{1}{x^2} \\ &= -y^2 \quad \text{식 (5.10)} \end{aligned}$$

위 식에 따르면 역전파일 때는 상류에서 훌러온 값에 $-y^2$ (순전파의 출력을 제곱한 후 마이너스를 붙인 값)을 곱해서 하류로 전달한다. 계산 그래프에서는 다음과 같다.



2단계

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

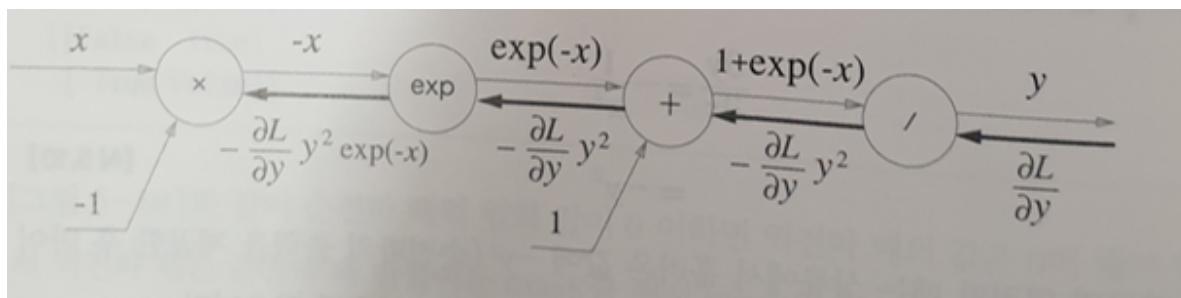


3단계

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

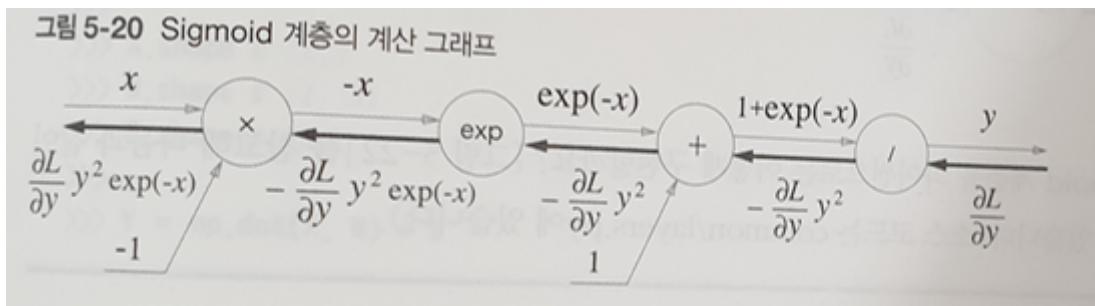
$$\frac{\partial y}{\partial x} = \exp(x)$$

계산 그래프에서는 상류의 값에 순전파 때의 출력(이 예에서는 $\exp(-x)$)을 곱해 하류로 전파한다.



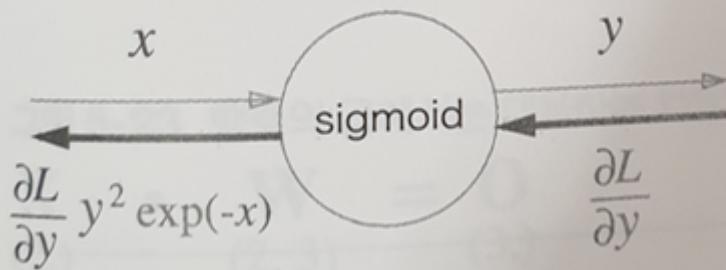
4단계

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



이상으로 Sigmoid 계층의 역전파를 계산 그래프로 완성했다. 위 그림에서 보듯이 역전파의 최종 출력인 $\frac{\partial L}{\partial y} y^2 \exp(-x)$ 값이 하류 노드로 전파된다. 여기에서 $\frac{\partial L}{\partial y} y^2 \exp(-x)$ 를 순전파의 입력 x와 출력 y 만으로 계산할 수 있다는 것을 눈치챘니? 이를 단순한 'sigmoid'노드 하나로 대체할 수 있다.

그림 5-21 Sigmoid 계층의 계산 그래프(간소화 버전)



위 그림 5-20과 그림 5-21의 간소화 버전의 결과는 똑같다. 그러나 간소화 버전은 역전파 과정의 중간 계산들을 생략할 수 있어 더 효율적인 계산이라고 말할 수 있다.

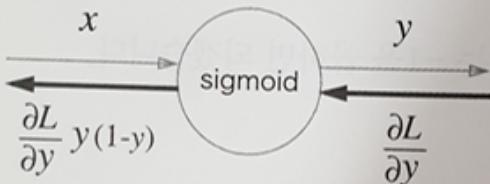
또, 노드를 그룹화하여 Sigmoid 계층의 세세한 내용을 노출하지 않고 입력과 출력에만 집중할 수 있다는 것도 중요한 포인트다.

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

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

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

그림 5-22 Sigmoid 계층의 계산 그래프 : 순전파의 출력 y 만으로 역전파를 계산할 수 있다.



그럼 Sigmoid 계층을 파이썬으로는 어찌 구현할까?

```
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
```

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

5.6 Affine/Softmax 계층 구현하기

5.6.1 Affine 계층

신경망이 순전파에서는 가중치 신호의 총합을 계산하기 때문에 행렬의 곱(넘파이에서는 np.dot()을 사용했다.)

```
In [19]: X = np.random.rand(2) #입력  
In [20]: W = np.random.rand(2,3) #가중치  
In [21]: B = np.random.rand(3) #편향  
In [22]: X.shape  
Out[22]: (2,)  
In [23]: W.shape  
Out[23]: (2, 3)  
In [24]: B.shape  
Out[24]: (3,)  
In [25]: Y = np.dot(X,W) + B  
In [26]: Y  
Out[26]: array([0.3182512 , 1.06804937, 0.25621728])
```

여기에서 X, W, B 는 각각 형상이 $(2,)$, $(2,3)$, $(3,)$ 인 다차원 배열이다. 그러면 뉴런의 가중치 합은 $Y = np.dot(X, W) + B$ 처럼 계산한다. 그리고 이 Y 를 활성화 함수로 변환해 다음 층으로 전파하는 것이 신경망 순전파의 흐름이었다. 또한, 행렬의 곱 계산은 대응하는 차원의 원소 수를 일치시키는 것이 핵심이었다. 예를 들어 X 와 W 의 곱은 이처럼 원소 수를 일치시켜야 한다.

Note 신경망의 순전파 때 수행하는 행렬의 곱은 기하학에서는 어파인 변환이라고 한다. 이에 어파인 변환을 수행하는 처리를 'Affine 계층'이라는 이름을 수행한다.

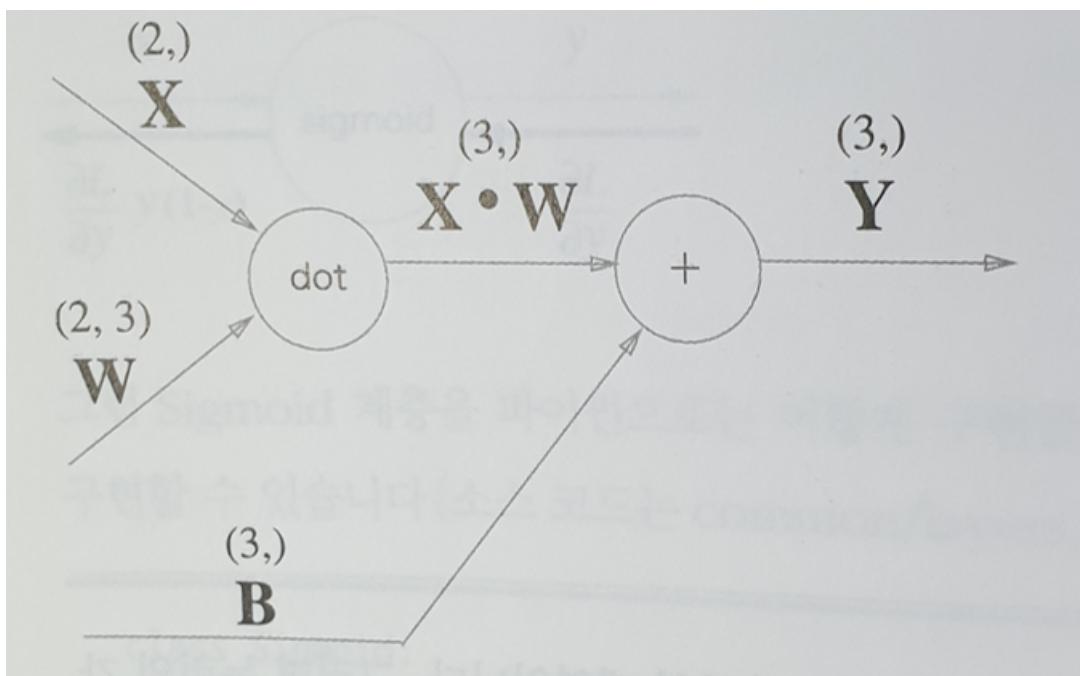


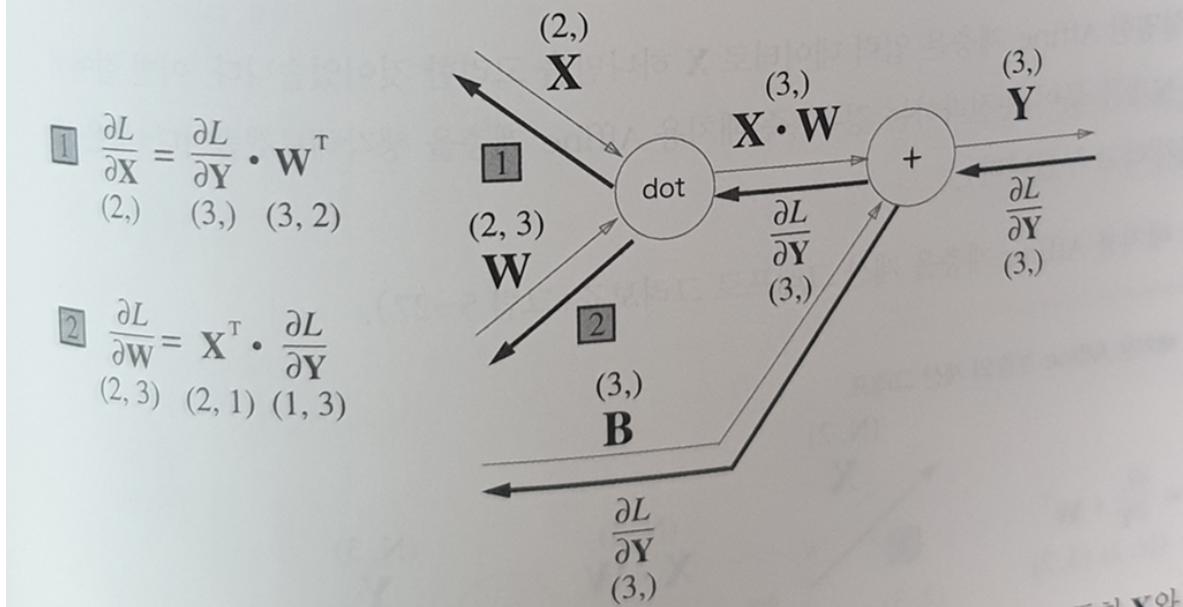
그림 5.24

위 그림은 이를 단순화한 계산 그래프이다. 단, X, W, B 가 행렬(다차원 배열)이라는 점에 유의해야 한다. 지금까지의 계산 그래프는 노드 사이에 '스칼라값'이 흘렀는데 반해, 이 예에서는 '행렬'이 흐르고 있는 것이다.

이제 그림 5.24의 역전파에 대해 생각해보자. 행렬을 사용한 역전파도 행렬의 원소마다 전개해보면 스칼라값을 사용한 지금까지의 계산 그래프와 같은 순서로 생각할 수 있다. 실제로 전개해보면 다음 식이 도출된다.

$$\begin{aligned}\frac{\partial L}{\partial X} &= \frac{\partial L}{\partial Y} \cdot W^T \\ \frac{\partial L}{\partial W} &= X^T \cdot \frac{\partial L}{\partial Y}\end{aligned}$$

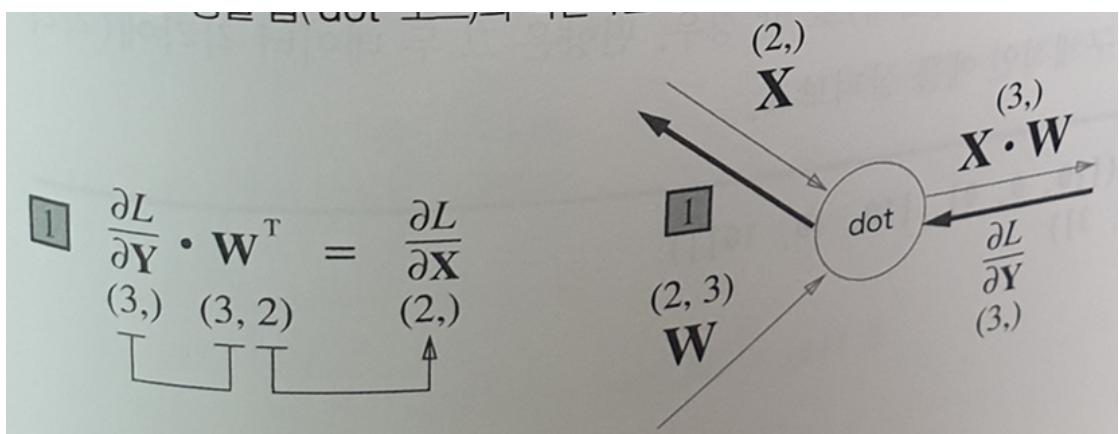
식 5.13



Affine 계층의 역전파 : 변수가 다차원 배열임에 주의, 역전파에서의 변수 형상은 해당 변수명 아래에 표기했다.

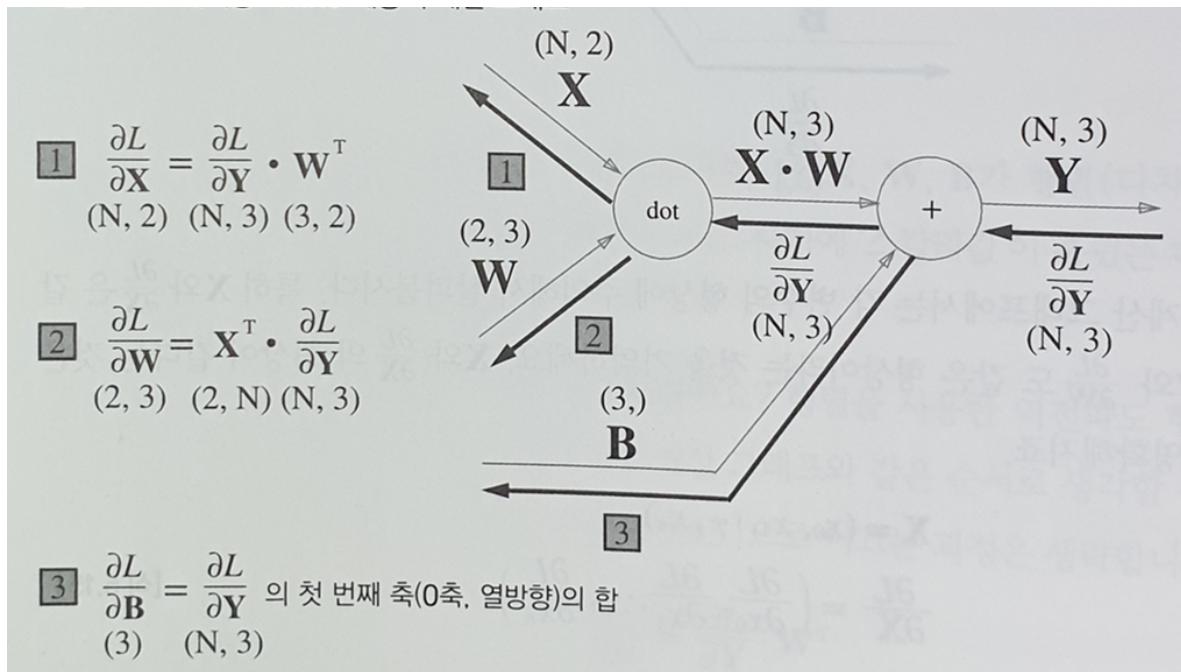
위 계산 그래프를 잘 살펴보자. 특히 X 와 $\frac{\partial L}{\partial X}$ 은 같은 형상이고, W 와 $\frac{\partial L}{\partial W}$ 도 같은 형상이다.

$$\begin{aligned}X &= (x_0, x_1, x_2, \dots, x_n) \\ \frac{\partial L}{\partial X} &= \left(\frac{\partial L}{\partial x_0}, \frac{\partial L}{\partial x_1}, \dots, \frac{\partial L}{\partial x_n} \right)\end{aligned}$$



5.6.2 배치용 Affine 계층

지금까지 설명한 Affine 계층은 입력 데이터로 X 하나만을 고려한 것이다. 이번 절에서는 데이터 N 개를 묶어 순전파하는 경우, 즉 배치용 Affine 계층을 생각해보겠다.(묶은 데이터를 '배치'라고 한다.)



기준과 다른 부분은 입력인 X 의 형상이 $(N, 2)$ 가 된 것이다. 그 뒤로는 지금까지와 같이 계산 그래프의 순서를 따라 순순히 행렬 계산을 하게 된다. 또, 역전파 때는 행렬의 형상에 주의하면 $\frac{\partial L}{\partial X}$, $\frac{\partial L}{\partial W}$ 은 이전과 같이 도출할 수 있다. 편향을 더할 때도 주의해야 한다. 순전파 때의 편향 덧셈은 $X \times W$ 에 대한 편향이 각 데이터에 더해진다. 예를 들어 $N = 2$ (데이터가 2개)로 한 경우, 편향은 그 두 데이터 각각에(각각의 계산 결과에) 더해진다.

```
In [3]: x_dot_w = np.array([[0,0,0],[10,10,10]])  
  
In [4]: b = np.array([1,2,3])  
  
In [5]: x_dot_w  
Out[5]:  
array([[ 0,  0,  0],  
       [10, 10, 10]])  
  
In [6]: b  
Out[6]: array([1, 2, 3])  
  
In [7]: x_dot_w + b  
Out[7]:  
array([[ 1,  2,  3],  
       [11, 12, 13]])
```

순전파의 편향 덧셈은 각각의 데이터(1번째 데이터, 2번째 데이터, ...)에 더해진다. 그래서 역전파 때는 각 데이터의 역전파 값이 편향의 원소에 모여야 한다. 코드는 다음과 같다.

```
In [8]: dY = np.array([[1,2,3],[4,5,6]])
```

```
In [9]: dY
```

```
Out[9]:
```

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

```
In [10]: dB = np.sum(dY, axis = 0)
```

```
In [11]: dB
```

```
Out[11]: array([5, 7, 9])
```

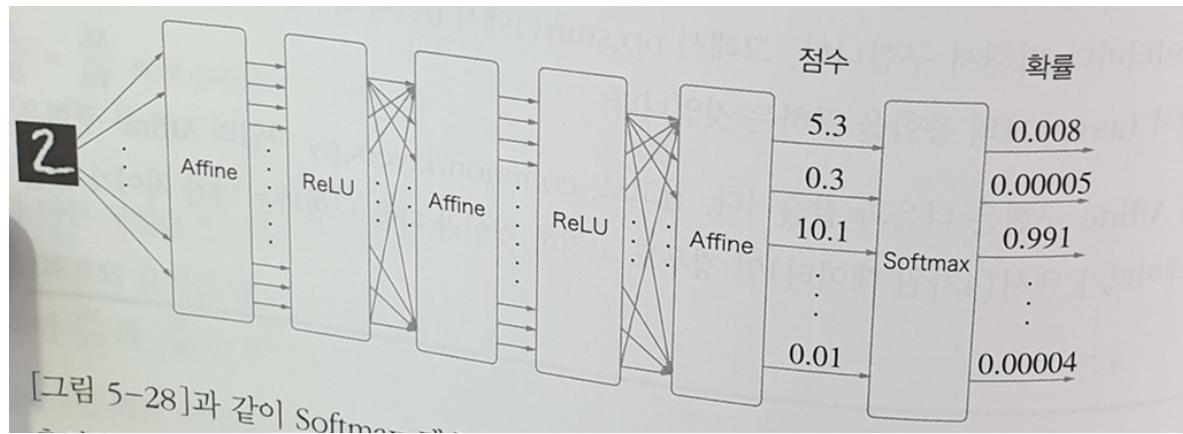
이 예에서는 데이터가 2개($N = 2$)라고 가정한다. 편향의 역전파는 그 두 데이터에 대한 미분을 데이터마다 더해서 구한다. 그래서 `np.sum()`에서 0번째 축(데이터를 단위로 한 축)에 대해 (`axis = 0`)의 총합을 구하는 것이다.

이상의 Affine 구현은 다음과 같다.

```
In [13]: 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)  
....  
....         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

마지막으로 출력층에서 사용하는 소프트맥스 함수에 관해 설명하겠다. 앞서 말했듯 소프트맥스 함수는 입력 값을 정규화하여 출력한다. 예를 들어 손글씨 숫자 인식에서의 Softmax 계층의 출력은 다음 그림과 같다.

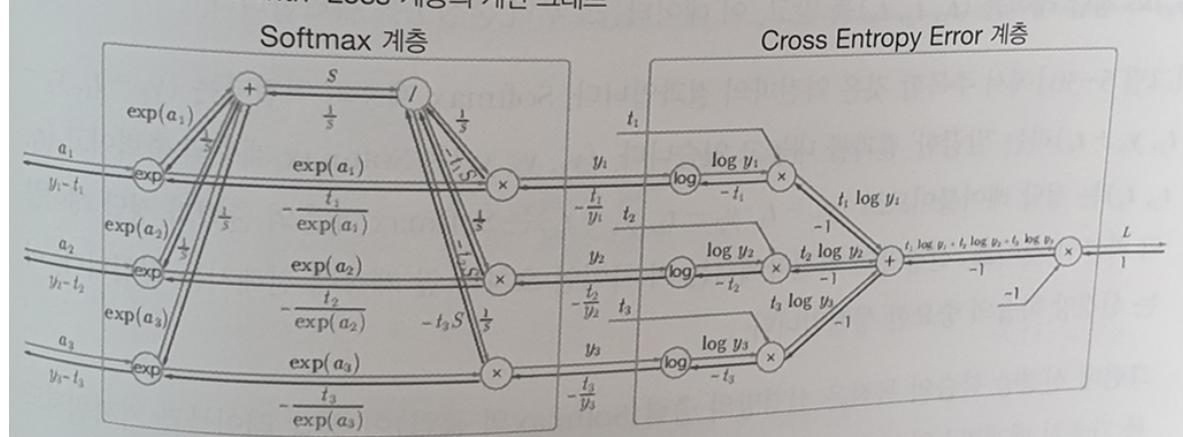


Softmax 계층은 입력 값을 정규화(출력의 합이 1이 되도록 변형) 하여 출력한다. 또한, 손글씨 숫자는 가짓수가 10개(10클래스 분류)이므로 Softmax 계층의 입력은 10개가 된다.

Note 신경망에서 수행하는 작업은 학습과 추론 두 가지이다. 추론할 때는 일반적으로 Softmax 계층을 사용하지 않는다. 예를 들어 위 그림의 신경망을 추론할 때는 마지막 Affine 계층의 출력을 인식 결과로 이용한다. 또한 신경망에서 정규화하지 않은 출력 결과(위 그림에서는 Softmax 앞의 Affine 계층의 출력) 점수(Score)라 한다. 즉, 신경망 추론에서 답을 하나만 내는 경우에는 가장 높은 점수만 알면 되니 Softmax 계층은 필요 없다는 것이다.

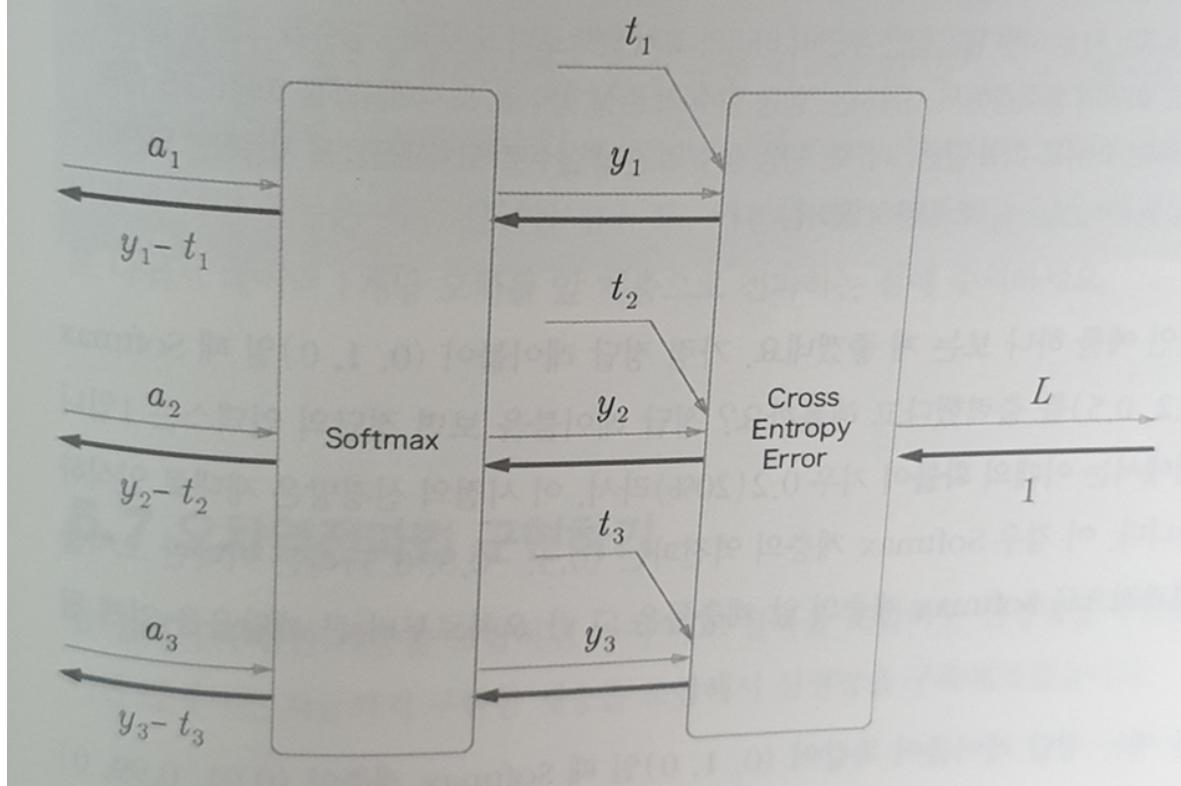
이제 소프트맥스 계층을 구현할 텐데, 순실 함수인 엔트로피 오차도 포함하여 'Softmax-with-Loss 계층'이라는 이름을 구현한다.

그림 5-29 Softmax-with-Loss 계층의 계산 그래프



보다시피 Softmax-with-Loss 계층은 다소 복잡하다. 이는 다음 그림처럼 간소화할 수 있다.

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



위 그림의 계산 그래프에서 소프트맥스 함수는 'Softmax' 계층으로, 교차 엔트로피 오차는 'Cross Entropy Error' 계층으로 표기했다. 여기에서는 3클래스 분류를 가정하고 이전 계층에서 3개의 입력(점수)을 받는다. 그림과 같이 Softmax 계층은 입력(a_1, a_2, a_3)을 정규화하여 (y_1, y_2, y_3)를 출력한다. Cross Entropy Error 계층은 Softmax의 출력 (y_1, y_2, y_3)과 정답 레이블 (t_1, t_2, t_3)을 받고, 이 데이터들로부터 손실 L 을 출력한다.

위 그림에서 주목할 것은 역전파의 결과이다. Softmax 계층의 역전파는 ($y_1 - t_1, y_2 - t_2, y_3 - t_3$) 이라 는 '깔끔한' 결과를 내놓고 있다. (y_1, y_2, y_3)는 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. 이는 신경망 학습의 중요한 성질이다. 그런데 신경망 학습의 목적은 신경망의 출력(Softmax의 출력)이 정답 레이블과 가까워지도록 가중치 매개변수의 값을 조정하는 것이었다. 이에 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 정답레이블의 차이로, 신경망의 현재 출력과 정답 레이블의 오차를 있는 그대로 드러내는 것이다.

Note_ '소프트맥스 함수' 손실 함수로 '교차 엔트로피 오차'를 사용하니 역전파가

($y_1 - t_1, y_2 - t_2, y_3 - t_3$)로 말끔히 떨어진다. 사실 이런 말끔함은 우연이 아닌 교차 엔트로피 오차라는 함수가 그렇게 설계되었기 때문이다. 또, 회귀의 출력층에서 사용하는 '항등 함수'의 손실 함수로 '오차제곱합'을 이용하는 이유도 이와 같다. 즉 '항등 함수'의 손실 함수로 '오차제곱합'을 사용하면 역전파의 결과가 ($y_1 - t_1, y_2 - t_2, y_3 - t_3$)로 말끔히 떨어진다.

이쯤에서 구체적인 예를 하나 보자. 가령 정답 레이블이 (0,1,0)일 때 Softmax 계층이 (0.3,0.2,0.5)를 출력했다고 하자. 정답 레이블을 보면 정답의 인덱스는 1이다. 그런데 출력에서는 이때의 확률이 겨우 0.2(20%) 라서, 이 시점의 신경망은 제대로 인식하지 못하고 있다. 이 경우 Softmax 계층의 역전파는 (0.3, -0.8, 0.5)라는 커다란 오차를 전파한다. 결과적으로 Softmax 계층의 앞 계층들은 그 큰 오차로부터 큰 깨달음을 얻게 될 것이다.

이번에 살펴볼 예는 정답 레이블이 똑같이 (0,1,0)일 때, Softmax 계층이 (0.01,0.99,0)을 출력한 경우다. 이 경우 Softmax 계층의 역전파가 보내는 오차는 비교적 작은 (0.01,-0.01,0)이다. 이번에는 앞 계층으로 전달된 오차가 작으므로 학습하는 정도도 작아진다.

그럼 Softmax-with-Loss 계층을 구현한 코드를 보겠다.

```
In [17]: class SoftmaxWithLoss:  
....:     def __init__(self):  
....:         self.loss = None #손실  
....:         self.y = None #softmax의 출력  
....:         self.t = None #정답레이블 (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
```

이 구현에서는 softmax 함수와 cross_entropy_error 함수를 사용했다.

5.7 오차역전파법 구현하기

이번 절에서는 구현한 계층을 조합해서 신경망을 구축해보겠다.

5.7.1 신경망 학습의 전체 그림

전체

신경망에는 적용 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적용하도록 조정하는 과정을 '학습'이라 한다. 신경망 학습은 다음과 같은 4단계로 수행된다.

1단계 - 미니배치

훈련 데이터 중 일부를 무작위로 가져온다. 이렇게 선별한 데이터를 미니배치라고 하며, 그 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 크게 하는 방향을 제시한다.

3단계 - 매개변수 갱신

가중치 매개변수를 기울기 방향으로 아주 조금 갱신한다.

4단계 - 반복

1~3 단계를 반복한다.

지금까지 설명한 오차역전파법이 등장하는 단계는 두 번째인 '기울기 산출'이다. 앞 장에서는 이 기울기를 구하기 위해서 수치 미분을 사용했다. 근데 수치 미분은 구현하기는 쉽지만 계산이 오래 걸렸다. 오차역전파법을 이용하면 느린 수치 미분과는 달리 기울기를 효율적이고 빠르게 구할 수 있다.

5.7.2 오차역전파법을 적용한 신경망 구현하기

2층 신경망을 TwoLayerNet 클래스로 구현한다.

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']은 2번째 층의 가중치, params['b2']은 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

TwoLayerNet 클래스의 매서드

매서드	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.
gradient(self, x, t)	가중치 매개변수의 기울기를 오차역전파법으로 구한다.

```

import sys, os
    sys.path.append(os.pardir)
....: 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_size, weight_init_std = 0.01):
....:         #가중치 초기화
....:         self.params = {}
....:         self.params['w1']=weight_init_std *\n                                np.random.randn(input_size, hidden_s
....:         ize)
....:         self.params['b1']=np.zeros(hidden_size)
....:         self.params['w2']=weight_init_std *\n                                np.random.randn(hidden_size, output_s
....:         ize)
....:         self.params['b2']=np.zeros(output_size)
....:
....:
```

```

....:     #계층 형성
....:     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 layers 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)
....:     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
....:
```

이 구현에서는 굵은 글씨들을 집중해서 봐라. 특히 신경망의 계층을 OrderedDict 은 순서가 있는 딕셔너리이다. '순서가 있는'이라는 말은 딕셔너리에 추가한 순서를 기억한다는 것이다. 그래서 순전파 때는 추가한 순서대로 각 계층의 forward() 메서드를 호출하기만 하면 처리가 완료된다. 마찬가지로 역전파 때는 계층을 반대로 순서로 호출하기만 하면 된다. Affine 계층과 ReLU계층이 각자의 내부에서 순전파와 역전파를 제대로 처리하고 있으니, 여기에서는 그냥 계층을 올바른 순서로 연결한 다음 순서대로(혹은 역순으로) 호출해주면 된다.

이처럼 신경망의 구성요소를 '계층'으로 구현한 덕분에 신경망을 쉽게 구축할 수 있었다.

'계층'으로 모듈화해서 구현한 효과는 아주크다. 예컨대 5층, 10층, 15층, 20층, ... 과 같이 깊은신경망을 만들고 싶다면, 단순히 필요한 만큼 계층을 더 추가하면 된다.

5.7.3 오차역전파법으로 구한 기울기 검증하기

지금까지 기울기를 구하는 방법을 두 가지 설명했다. 하나는 수치 미분을 써서 구하는 방법, 또 하나는 해석적으로 수식을 풀어 구하는 방법이다. 후자인 해석적 방법은 오차역전파법을 이용하여 매개변수가 많아도 효율적으로 계산할 수 있었다. 그러니 이제부터는 느린 수치 미분 대신 역전파법을 사용하자.

수치 미분은 느리다. 그리고 오차역전파법을 제대로 구현하면 수치 미분은 더 이상 필요하지 않다. 하지만 정말 수치미분이 쓸모없을까? 사실 수치 미분은 오차역전파법을 정확히 구현했는지 확인하기 위해 필요하다.

수치 미분의 이점은 구현하기 쉽다는 것이다. 이에 수치 미분의 구현에는 버그가 숨어있기 어려운 반면, 오차역전파법은 구현하기 복잡해서 종종 실수를 한다. 이에 수치 미분의 결과와 오차역전파법의 결과를 비교하여 오차역전파법을 제대로 구현했는지 제대로 구현했는지 검증하곤 한다. 이처럼 두 방식으로 구한 기울기가 일치함(엄밀히 말하면 거의 같음)을 확인하는 작업을 **기울기 확인**이라고 한다.

```
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)...: (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 차이의 절댓값을 구한 후, 그 절댓값들의 평균
....: for key in grad_numerical.keys():
```

```

....      diff = np.average(np.abs(grad_backprop[key]-grad_numerical
.... [key]))
....      print(key+":"+str(diff))
....
```

w1:3.9516239983994955e-10
b1:2.074078191372994e-09
w2:6.456752534558438e-09
b2:1.3973276453971284e-07

언제나처럼 가장 먼저 MNIST 데이터셋을 읽는다. 그리고 훈련 데이터 일부를 수치 미분으로 구한 기울기와 오차역전파법으로 구한 기울기의 오차를 확인한다. 여기에서는 각 가중치 매개변수의 차이에 대한 절댓값을 구하고, 이를 평균한 값이 오차가 된다.

이 결과는 수치 미분과 오차역전파법으로 구한 기울기의 차이가 매우 작다고 말해준다.

Note_ 수치 미분과 오차역전파법의 결과 오차가 0이 되는 경우는 드물다. 이는 컴퓨터가 할 수 있는 계산의 정밀도가 유한하기 때문이다(가령 32비트 부동소수점). 이 정밀도의 한계 때문에 오차는 대부분 0이 되지는 않지만, 올바르게 구현했다면 0에 아주 가까운 작은 값이 된다. 만약 그 값이 크면 오차역전파법을 잘못 구현했다고 의심해봐야 한다.

5.7.4 오차역전파법을 사용한 학습 구현하기

마지막으로 오차역전파법을 사용한 신경망 학습을 구현해보겠다. 지금까지와 다른 부분은 기울기를 오차역전파법으로 구한다는 점 뿐이다.

```

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, 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]

    #오차역전파법으로 기울기를 구한다.
    grad = network.gradient(x_batch, t_batch)

    #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)

```

In [17]: %run train_neuralnet.py

```

0.1022 0.1043
0.9047833333333334 0.9071
0.9253333333333333 0.9273
0.93695 0.9355
0.94475 0.9411
0.95265 0.9501
0.9571333333333333 0.9537
0.9619166666666666 0.9586
0.9637666666666667 0.9607
0.9672833333333334 0.9619
0.9696833333333333 0.9646
0.9716 0.967
0.9737666666666667 0.9685
0.97545 0.97
0.9767833333333333 0.9695
0.9775333333333334 0.969
0.9786333333333334 0.9704

```

5.8 정리

이번 장에서는 계산 과정을 시각적으로 보여주는 방법인 계산 그래프를 배웠다. 계산 그래프로 이용하여 신경망의 동작과 오차역전파법을 설명하고, 그 처리 과정을 계층이라는 단위로 구현했다. 예를 들어 ReLU 계층, Softmax-with-Loss 계층, Affine 계층, Softmax 계층 등이다. 모든 계층에서 forward와 backward라는 메서드를 구현한다. 전자는 데이터를 순방향으로 전파하고, 후자는 역방향으로 전파함으로써 가중치 매개변수의 기울기를 효율적으로 구할 수 있다. 이처럼 동작을 계층으로 모듈화 한 덕분에, 신경망의 계층을 자유롭게 조합하여 원하는 신경망을 쉽게 만들 수 있다.

이번 장에서 배운 내용

- 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)

