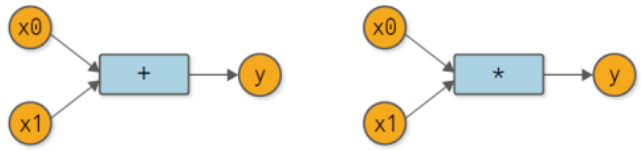
Ch03.

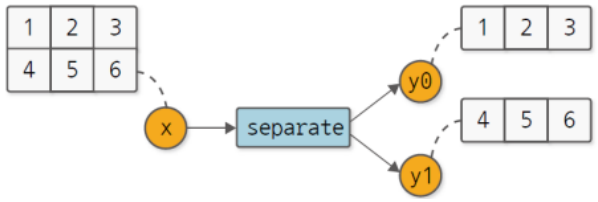
- 가변길이: 인수 또는 반환 값의 수가 달라질 수 있다는 뜻
- 가변 길이 입출력 처리
  - 여러 개의 변수를 입력 받는 함수

그림 11-1 '덧셈' 계산 그래프와 '곱셈' 계산 그래프(곱셈 연산은 *로 표시)



  - 출력이 여러 개인 함수

그림 11-2 출력이 여러 개인 계산 그래프(다차원 배열을 분할하는 함수)



- 가변 길이 입출력 표현
  - 변수들을 리스트(또는 튜플)에 넣어 처리
  - '하나의 인수' 만 받고 '하나의 값'만 반환
  - 인수와 반환 값의 타입을 리스트로 바꾸고, 필요한 변수들을 리스트에 넣음

- 가변 인수 리스트
  - 함수를 정의할 때 인수 앞에 별표(*)를 붙임

```
def function(a, *args):
  print(a, args)

function(1)
# 1 ()
function(1,2)
# 1 (2,)
function(1,2,3,4,5)
# 1 (2, 3, 4, 5)
function(1,2,3,4,5,'a','b',[6,7,8])
# 1 (2, 3, 4, 5, 'a', 'b', [6, 7, 8])
```

→인수 리스트에 반드시 넘겨야 하는 고정 인수들을 나열하고, 나머지는 튜플 형식으로 한꺼번에 받는다

- Packing, unpacking
  - Packing: 여러 개의 객체를 하나의 객체로 합쳐주는 역할
    = 가변 인수 리스트
  - Unpacking: 여러 개의 객체를 포함하고 있는 하나의 객체를 풀어주는 역할3

```
def sum(a, b, c):
    return a + b + c

numbers = [1, 2, 3]
sum(numbers) # error

print(sum(*numbers)) #6
```

- 다변수 함수: 입력 변수가 여러 개인 함수
  - 편미분: 다변수 함수에서 하나의 입력 변수에만 주목하여 다른 변수는 상수로 취급하여 미분하는 것
- 복잡하게 연결된 그래프의 올바른 순서

그림 15-3 중간에 분기했다가 다시 합류하는 계산 그래프

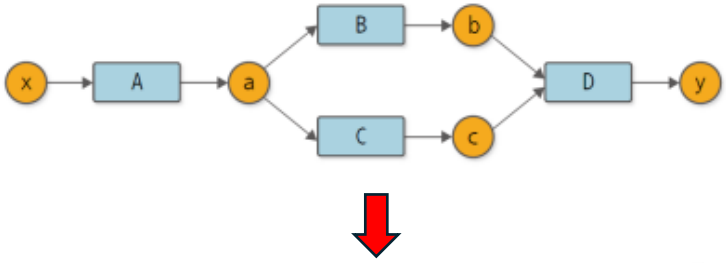
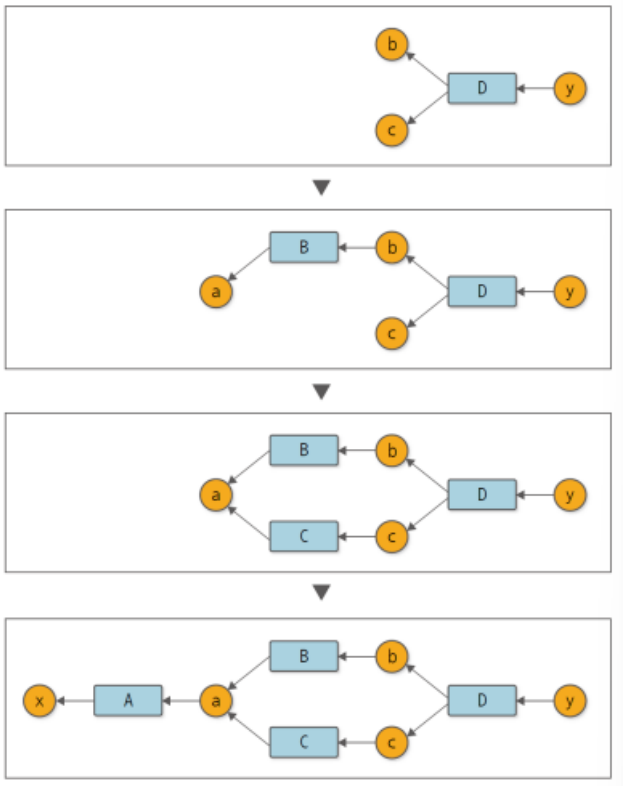
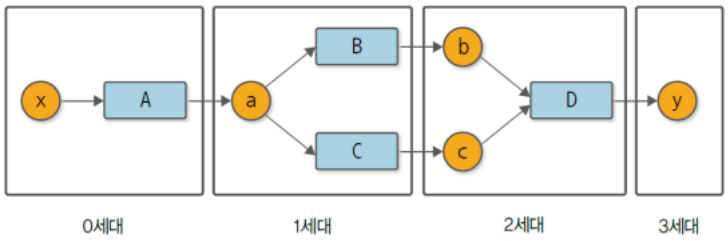


그림 15-4 올바른 역전파의 순서



- 함수 우선순위 설정
  - 순전파 때 함수가 변수를 만들어 냄. 이는 부모-자식 관계임.
  - 이 관계를 기준으로 함수와 변수의 세대(generation)을 기록함
  - 세대가 우선순위에 해당함

그림 15-8 순전파 때의 함수와 변수 '세대'



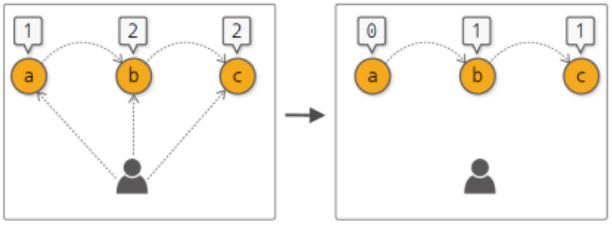
➔ 순전파 시 세대를 설정

➔ 역전파 시 최근 세대의 함수부터 꺼냄

CH04.

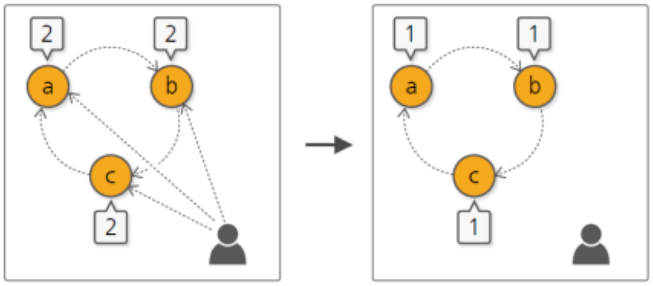
- 메모리 관리
  - 파이썬 메모리 관리
    - ➔ 파이썬은 필요 없어진 객체를 메모리에서 자동 삭제함
    - ➔ 메모리 누수 또는 메모리 부족 문제 발생
  - 신경망의 메모리 관리
    - ➔ 큰 데이터를 다룸
    - ➔ 메모리 관리가 적절하지 못하면 실행 시간이 오래 걸리는 일이 자주 발생
  - 파이썬의 메모리 관리 두가지 방식
    - ➔ 참조(reference)수를 세는 방식
    - ➔ Garbage Collection 방식 : 세대를 기준으로 쓸모 없어진 객체를 회수하는 방식
- 참조 카운터
  - 파이썬 메모리 관리의 기본
  - 수행방식
    - ➔ 모든 객체는 참조 카운트 0인 상태로 생성되고, 다른 객체가 참조할 때마다 1씩 증가
    - ➔ 객체에 대한 참조가 끊길 때마다 1만큼 감소하다가 0이 되면 회수

그림 17-1 객체 관계도(참조 관계는 점선, 숫자는 참조 카운트)



- 순환 참조
  - 참조 카운트로는 해결할 수 없는 문제

그림 17-2 순환 참조가 발생한 객체 관계도(점선이 참조를 뜻함)



  - ➔ a = b = c = None 해도 메모리에서 삭제되지 않음.
- weakref module
  - 다른 객체를 참조하되 참조 카운터는 증가시키지 않는 기능
  - 파이썬에서는 weakref.ref 함수를 사용하여 약한 참조를 만듦
  - b는 약한 참조이고, 약한 참조된 데이터에 접근하려면 b( )라고 쓰면 됨

```
import weakref
import numpy as np

a = np.array([1,2,3])
b = weakref.ref(a)
print(b)
print(b())

a = None
print(b)
```

```
<weakref at 0x00000184F1098A90; to 'numpy.ndarray' at 0x00000184F10B6E10>
[1 2 3]
<weakref at 0x00000184F1098A90; dead>
```

- 모드전환
  - 순전파만 수행할 경우를 위한 모드 전환 구조
    - ➔ 학습 시에는 역전파를 수행하지만, 추론 시에는 순전파만 수행함
    - ➔ 역전파 활성 모드와 역전파 비활성 모드를 전환하는 구조 필요
    1. Config 클래스: 역전파가 가능한지 여부를 뜻하는 enable_backprop 속성을 가짐
       A. True: 중간 계산 결과가 계속 유지되어 메모리 차지함
       B. False: 중간 계산 결과는 곧바로 삭제
    2. with문: with 블록에 들어갈 때의 처리 (전처리)와 with 블록을 빠져나올 때의 처리 (후처리)를 자동으로 수행

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False
```

```
>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

➔ ContextDecorator: 데코레이터를 구현하기 위한 로직을 제공한다.

➔ With [expression] as [변수명]에서 as를 따로 작성하지 않아도 된다.

- Variable 인스턴스에서도 할 수 있도록 확장

```
class Variable:
    '''''''
    @property
    def shape(self):
        return self.data.shape
```

```
x = Variable(np.array([[1,2,3],[4,5,6]]))
print(x.shape)  # x.shape() 대신 x.shape로 호출할 수 있음.
```

```
(2, 3)
```

```
class Variable:
    '''''''
    @property
    def ndim(self):  # 차원수
        return self.data.ndim

    @property
    def size(self):  # 원소수
        return self.data.size

    @property
    def dtype(self):  # 데이터 타입
        return self.data.dtype
```

- len함수

```
class Variable:
    '''''''
    def __len__(self):
        return len(self.data)
```

```
x = Variable(np.array([[1,2,3],[4,5,6]]))
print(len(x))
```

```
2
```

- print함수

```
class Variable:
    '''''''
    def __repr__(self):
        if self.data is None:
            return 'variable(None)'
        p = str(self.data).replace('₩n', '₩n' + ' ' * 9)
        return 'variable(' + p + ')'
```

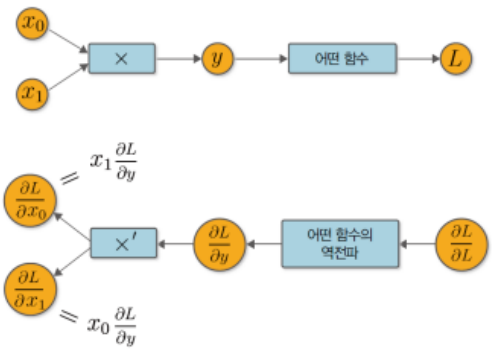
```
x = Variable(np.array([[1,2,3],[4,5,6]]))
print(x)
```

```
variable([[1 2 3]
          [4 5 6]])
```

- 연산자 오버로드
  - ■ + 와 * 연산자를 지원하도록 함
  - ■ 곱셈을 수행하는 Mul 클래스 구현

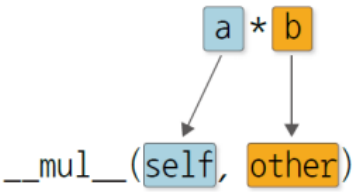
- 곱셈의 순전파와 역전파
  - 역전파는 최종 출력인 L의 미분을, 정확하게는 L의 각 변수에 대한 미분을 전파함
  - L 은 오차, 다른 말로 손실(loss)을 뜻함

그림 20-1 곱셈의 순전파(위)와 역전파(아래)



  - 곱셈 연산자 * 오버로드
    - 곱셈의 특수 메서드는 __mul__(self, other)임.
    - __mul__ 메서드를 정의하면 * 연산자를 사용할 때 __mul__ 메서드가 호출됨

그림 20-2 __mul__ 메서드로 인수가 전달되는 방식



      ➔ 먼저 인스턴스 a의 특수 메서드인 __mul__이 호출됨
      ➔ 연산자 * 왼쪽의 a가 인수 self에 전달되고, 오른쪽의 b가 other에 전달됨.

- Ndarray 와 함께 사용하기

1. as_variable 함수: 인수로 주어진 객체를 Variable 인스턴스로 변환해주는 함수

```
def as_variable(obj):
    if isinstance(obj, Variable):
        return obj
    return Variable(obj)
```

```
class Function:
    def __call__(self, *inputs):
        inputs = [as_variable(x) for x in inputs]

        xs = [x.data for x in inputs]
        ys = self.forward(*xs)
```

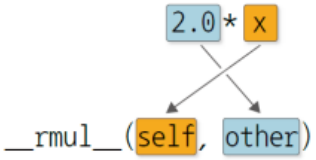
⇨

```
x = Variable(np.array(2.0))
y = x + np.array(3.0)
print(y)
```

```
variable(5.0)
```

- 이항 연산자의 경우 피연산자(항)의 위치에 따라 호출되는 특수 메서드가 다름.

그림 21-1 __rmul__ 메서드로 인수가 전달되는 방식




```
Variable.__add__ = add
Variable.__radd__ = add
Variable.__mul__ = mul
Variable.__rmul__ = mul
```

⇨

```
x = Variable(np.array(2.0))
y = 3.0 * x + 1.0
print(y)
```

```
variable(7.0)
```

표 22-1 이번 단계에서 추가할 연산자들

특수 메서드	예
__neg__(self)	-self
__sub__(self, other)	self - other
__rsub__(self, other)	other - self
__truediv__(self, other)	self / other
__rtruediv__(self, other)	other / self
__pow__(self, other)	self ** other

(추가 연산자들)

- 연산자 추가 순서
  1. Function 클래스를 상속하여 원하는 함수 클래스를 구현
  2. 파이썬 함수로 사용할 수 있도록 한다
  3. Variable 클래스의 연산자를 오버로드 한다