메모리 관리

- 파이썬 메모리 관리

필요 없어진 객체를 메모리에서 자동으로 삭제

코드 작성에 따라 메모리 누수 또는 메모리 부족 문제 발생

참조 수를 세는 방식

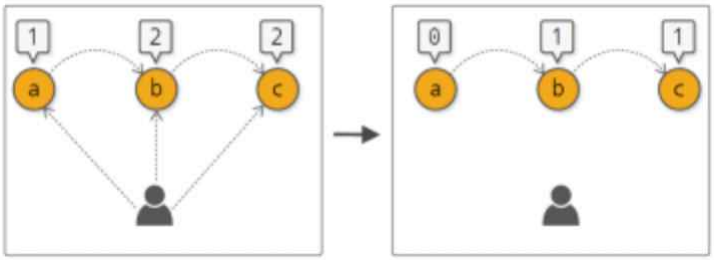
Garbage Collection(java에도 있는 방식) : 세대를 기준으로 쓸모없어진 객체를 회수하는 방식

참조 카운트

모든 객체는 참조 카운트가 0인 상태로 생성, 다른 객체가 참조할 때마다 1씩 증가

객체에 대한 참조가 끊길 때마다 1씩 감소, 0이 되면 회수

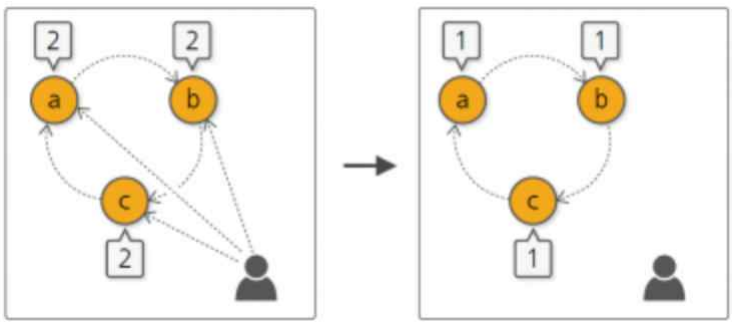
그림 17-1 객체 관계도(참조 관계는 점선, 숫자는 참조 카운트)



순환 참조

참조 카운트로는 해결할 수 없는 문제

그림 17-2 순환 참조가 발생한 객체 관계도(점선이 참조를 뜻함)



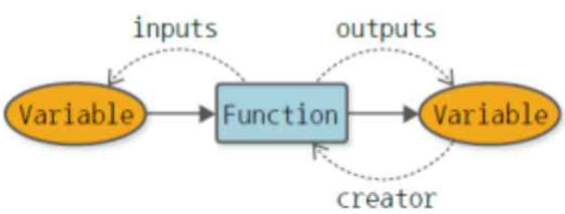
GC(Garbage Collection)

메모리가 부족해지는 시점에서 자동 호출(gc.colect()로 명시적(강제적) 호출도 가능)

메모리 해제를 GC에 미루다 보면 메모리 사용량이 커짐

DeZero에서는 순환참조를 만들지 않는 것이 좋음

그림 17-3 Variable과 Function 사이의 순환 참조



약한 참조(weak reference)

-다른 객체를 참조하되 참조 카운터는 증가시키지 않는 기능

```
>>> import weakref
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> b = weakref.ref(a)
>>> b
<weakref at 0x000001DBC37C7090; to 'numpy.ndarray' at 0x000001DBC3762DB0>
>>> b()
array([1, 2, 3])
>>> a = None
>>> b
<weakref at 0x000001DBC37C7090; dead>
>>>
```

b는 약한참조, 약한 참조된 데이터에 접근하려면 b()라고 쓰면 됨

a = None을 명시 후, b를 출력하면 dead라고 나옴 -> 인스턴스가 삭제됨

Weakref 구조를 Dezero에 도입

```
self.inputs = inputs
# 수정 전 : self.outputs = outputs
self.outputs = [weakref.ref(output) for ouput in outputs] # 수정 후
return outputs if len(outputs) > 1 else outputs[0]
```

```
funcs = [self.creator]
while funcs:
  f = funcs.pop()
  # 수정 전 : gys = [output.grad for output in f.outputs]
  gys = [output().grad for output in f.outputs] # 수정 후
  gxs = f.backward(*gys)
```

메모리 절약모드

1. 역전파 시 사용하는 메모리 양을 줄이기(불필요한 미분값 제거)

y.backward()를 실행하면 미분값을 메모리에 유지하기 때문에 backward에 메서드를 추가

2. 역전파가 필요 없는 경우용 모드 제공(불필요한 계산 생략)

- Config클래스를 활용한 모드 전환

enable_backprop이 True일때만 역전파 실행

- with문을 활용한 모드 전환

@contextlib.contextmanager 데코레이더를 달면 문맥을 판단하는 함수 생성

using_config함수 구현

위 내용 요약

##순환참조 해결

메모리절약(미분값 제거, 불필요한 역전파 생략)##

변수 사용성 개선

- Variable 클래스를 더욱 쉽게 사용하도록 개선

변수 이름 지정

```
class Variable:
  # 수정 전 : def __init__(self, data):
  def __init__(self, data, name = None): # 수정 후
    if data is not None:
      if  not isinstance(data, np.ndarray):
        raise TypeError('{} is not supported'.format(type(data)))

    self.data = data
    self.name = name # 추가
    self.gard = None
    self.creator = None
    self.generation = 0
```

- Variable 인스턴스를 ndarray 인스턴스처럼 보이게 함

@property는 shape메서드를 인스턴스 변수처럼 사용할 수 있게 함

```
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
```

ndim, size, dtype 등을 변수처럼 사용 가능

- len함수와 print함수

#특수 메서드(__len__ , __repr__)를 구현하면 Variable 인스턴스에 대해서도 len(), print()함수 사용 가능

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

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

```
2
```

```
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]])
```

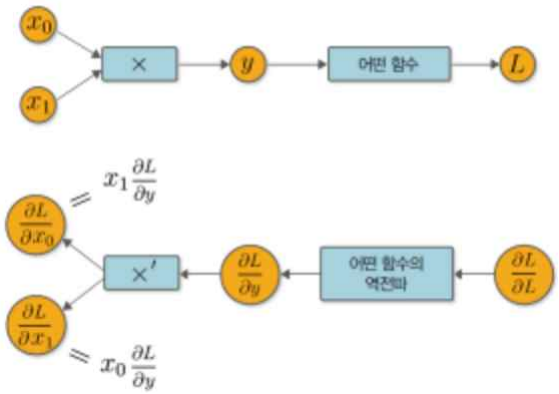
연산자 오버로드

- 연산자를 지원하도록 Variable확장(덧셈, 곱셈 연산자)

- Variable 인스턴스를 ndarray 인스턴스처럼 사용하도록 구성(y = a * b)
#DeZero를 평범한 넘파이 코드를 작성하듯 사용 가능

곱셈의 순전파와 역전파
- 역전파는 Loss의 각 변수에 대한 미분을 전파

그림 20-1 곱셈의 순전파(위)와 역전파(아래)



- Mul 클래스 구현

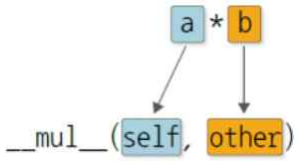
```
class Mul(Function):
    def forward(self, x0, x1):
        y = x0 * x1
        return y

    def backward(self, gy):
        x0, x1 = self.inputs[0].data, self.inputs[1].data
        return gy * x1, gy * x0


def mul(x0, x1):
    return Mul()(x0, x1)
```

곱셈 연산자 오버로드

y = add(mul(a, b), c) 에서 y = (a * b) + c 가 가능하도록 지원




```
class Variable:
    ''''''
    def __mul__(self, other):
        return mul(self, other)
```

```
class Variable:
    ''''''
    Variable.__mul__ = mul
    Variable.__add__ = add
```

Variable을 ndarray 인스턴스와 함께 사용하려면

- ndarray 인스턴스를 자동으로 Variable 인스턴스로 변환

#as_variable 함수를 이용

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

Variable을 float, int와 함께 사용하려면

- 변수가 float, int인 경우 ndarray 인스턴스로 변환

as_array 함수를 이용

```
def as_array(x):
    if np.isscalar(x):
        return np.array(x)
    return x
```

문제점 1 : 첫 번째 인수가 float, int인 경우

- y = 2.0 * x (TypeError 발생)

- 오류 발생 과정

연산자가 왼쪽에 있는 2.0의 __mul__메서드 호출 시도

2.0은 float타입이므로 __mul__메서드가 구현되어있지 않음

다음은 * 연산자 오른쪽에 있는 x의 특수 메서드 호출 시도

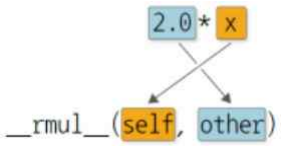
x가 오른쪽에 있기 때문에 __rmul__메서드 호출 시도

Variable 인스턴스에는 __rmul__메서드가 구현되어있지 않음

해결 방법

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

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



<- __rmul__메서드로 인수가 전달되는 방식

y = x * 2.0는 가능하지만 y = 2.0 * x가 안되는 문제를 rmul을 이용하여 해결

문제점 2 : 좌항이 ndarray 인스턴스인 경우

좌항은 ndarray 인스턴스의 __add__메서드를 호출하지만

우항은 Variable 인스턴스의 __radd__메서드가 호출되어야 함

Variable 인스턴스의 속성에 __array_priority__를 추가하고 그 값을 큰 수로 설정하여

연산자 우선순위를 지정해야 함

```
class Variable:
    __array_priority__ = 200
```

새로운 연산자들(연산자 오버로드 가능)

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

음수(부호 변환)

- 음수의 미분

역전파 상류에서 전해지는 미분에 -1을 곱하여 하류로 전달

Neg클래스를 구현 후 neg함수 구현

__neg__에 neg 대입

```
class Neg(Function):
    def forward(self, x):
        return -x

    def backward(self, gy):
        return -gy


def neg(x):
    return Neg()(x)

Variable.__neg__ = neg
```

- 뺄셈의 미분

역전파 상류에서 전해지는 미분값에 1을 곱한 값이 x0의 미분 결과가 되면 -1을 곱합 값이 x1의 미분 결과가 됨

```
class Sub(Function):
    def forward(self, x0, x1):
        y = x0 - x1
        return y

    def backward(self, gy):
        return gy, -gy


def sub(x0, x1):
    x1 = as_array(x1)
    return Sub()(x0, x1)

Variable.__sub__ = sub
```

x0와 x1이 Variable 인스턴스라면 y = x0 - x1 수행 가능

뺄셈 미분의 문제 : x0가 Variable 인스턴스가 아닌 경우 x의 __rsub__메서드가 호출, 처리 불가

해결방법 : 함수 rsub(x0, x1)을 정의 후 인수의 순서를 바꾸어 Sub()(x0, x1) 호출

```
def rsub(x0, x1):
    x1 = as_array(x1)
    return sub(x1, x0)

Variable.__rsub__ = rsub
```

- 거듭제곱의 미분

y = cx^(c-1) : c는 상수로 취급하여 미분하지 않음

순전파 메서드인 forward(x)는 x만 받게 함

```
class Pow(Function):
    def __init__(self, c):
        self.c = c

    def forward(self, x):
        y = x ** self.c
        return y

    def backward(self, gy):
        x = self.inputs[0].data
        c = self.c

        gx = c * x ** (c - 1) * gy
        return gx


def pow(x, c):
    return Pow(c)(x)

Variable.__pow__ = pow
```

위 내용 요약

##Variable클래스의 사용성을 개선

연산자를 자유롭게 사용하기 위한 방법(문제점, 해결 방법)##

패키지로 정리

- 파일로 구성

#dezero 패키지 - 딥러닝프레임워크

#steps 디렉토리 - step01.py~step.60.py

- 코어 클래스로 옮기기

~.py코드를 dezero/core_simple.py 코어 파일로 이동

지금까지의 기능들은 DeZero의 핵심이기 때문에 아래 코드가 정상 작동을 해야함

```
import numpy as np
from dezero.core_simple import Variable


x = Variable(np.array(1.0))
print(y)
```

- dezero임포트하기

dezero라는 패키지가 생성됨