정리노트(4,5주차)

산업데이터사이언스학부
201904213
심성빈

(4주차)

테스트

-단위 테스트 :

테스트 대상 단위의 크기를 작게 설정해서 단위 테스트를 최대한 간단하게 작성.

단위 테스트는 TDD(test driven develop)와 함께 할 때 더 강력.

-통합 테스트 :

여러 모듈을 모아 의도대로 동작되는지 확인.

단위 테스트에서 발견하기 어려운 버그를 찾을 수 있는 장점.

신뢰성 하락 가능성, 어디서 에러가 발생했는지 확인하기 쉽지 X.

파이썬 단위 테스트 지원

-unittest 라이브러리 사용.

-테스트할 때 이름이 test로 시작하는 메서드를 만드는 규칙이 있음.

```
import unittest

class SquareTest(unittest.TestCase):
    def test_forward(self):
        x = Variable(np.array(2.0))
        y = square(x)
        expected = np.array(4.0)
        self.assertEqual(y.data, expected)
```

➔ 잘못 된거면 forward에서 단위 테스트를 잘못 설정.

Square 함수의 역전파 테스트

-test_backward 메서드 추가.

-메서드 안에서 y.backward()로 미분값을 구하고, 그 값이 기댓값과 일치하는지 확인.

기울기 확인을 이용한 자동 테스트

-기울기 확인 :

수치 미분으로 구한 결과와 역전파로 구한 결과를 비교.

비교 했을때 차이가 크면 역전파 구현에 문제.

-기울기 확인을 위한 테스트 메서드 구현 :

```python
def numerical_diff(f, x, eps=1e-4):
    x0 = Variable(x.data - eps)
    x1 = Variable(x.data + eps)
    y0 = f(x0)
    y1 = f(x1)
    return (y1.data - y0.data) / (2 * eps)
```

```python
def test_gradient_check(self):
    x = Variable(np.random.rand(1))
    y = square(x)
    y.backward()
    num_grad = numerical_diff(square, x)
    flg = np.allclose(x.grad, num_grad)
    self.assertTrue(flg)
```

테스트 코드 : 테스트 코드는 tests 디렉토리에 모아둠.

칼럼 : 자동 미분
-수치 미분 : 변수에 미세한 차이를 주어 두 출력의 차이로부터 근사적으로 미분 계산.
-기호 미분 : 미분 공식을 이용하여 계산.
-자동 미분 : 연쇄 법칙을 사용(역전파 방식 사용).

가변 길이 인수(순전파)
-가변 길이 : 인수 또는 반환값의 수가 달라질 수 있다.
-가변 길이 입출력 표현 :
변수들을 리스트(또는 튜플)에 넣어 처리.
하나의 인수만 받고 하나의 값만 반환.
인수와 반환값의 타입을 리스트로 바꾸고 필요한 변수들을 리스트에 넣는다.
-인수와 반환값을 리스트로 변경 방법 :

```python
class Function:

    def __call__(self, inputs):
        xs = [ x.data for x in inputs]
        ys = self.forward(xs)
        outputs = [Variable(as_array(y)) for y in ys]

        for output in outputs:
            output.set_creator(self)
        self.inputs = inputs
        self.outputs = outputs
        return outputs

    def forward(self, x):
        raise NotImplementedError()

    def backward(self, gy):
        raise NotImplementedError()
```

➔ 많은 variable을 받을 수 있게 됨.
➔ 부모 creator 필수.

Add 클래스의 forward 메서드 구현 :
인수와 반환값이 리스트 또는 튜플.
인수는 변수가 두개 담긴 리스트.
결과를 반환할 때는 튜플을 반환.

순전파에서 가변 길이 입출력 처리 :

입력을 리스트로 바꿔서 여러 개의 변수를 다룸.

```
class Function:

    def __call__(self, inputs):
        xs = [x.data for x in inputs]
        ys = self.forward(xs)
        outputs = [Variable(as_array(y)) for y in ys]

        for output in outputs:
            output.set_creator(self)
        self.inputs = inputs
        self.outputs = outputs
        return outputs

    def forward(self, x):
        raise NotImplementedError()

    def backward(self, gy):
        raise NotImplementedError()
```

```
class Add(Function):
    def forward(self, xs):
        x0, x1 = xs
        y = x0 + x1
        return (y, )
```

```
xs = [Variable(np.array(2)), Variable(np.array(3))]
f = Add()
ys = f(xs)
y = ys[0]
print(y.data)
```

➔ 리스트를 뽑아서 가변 처리.

개선점 :

입력시에 변수를 리스트로 전달하도록 요청.

반환값도 튜플로 전달.

사용시 복잡.

개선사항 :

리스트나 튜플을 거치지 않고 인수와 결과를 직접 주고 받도록 함.

소스 수정 :

함수를 정의할 때 인수 앞에 *을 붙임.

```
class Function:

    def __call__(self, inputs):
        xs = [x.data for x in inputs]
        ys = self.forward(xs)
        outputs = [Variable(as_array(y)) for y in ys]

        for output in outputs:
            output.set_creator(self)
        self.inputs = inputs
        self.outputs = outputs
        return outputs
```

Add 클래스 구현을 위한 개선 :

Forward 메서드의 코드를 입력도 변수로 받고, 결과도 변수로 반환.

```
class Function:
    def __call__(self, *inputs):
        xs = [x.data for x in inputs]
        ys = self.forward(*xs)
        if not isinstance(ys, tuple):
            ys = (ys,)
        outputs = [Variable(as_array(y)) for y in ys]

        for output in outputs:
            output.set_creator(self)
        self.inputs = inputs
        self.outputs = outputs
        return outputs if len(outputs) > 1 else outputs[0]
```

```
class Add(Function):
    def forward(self, x0, x1):
        y = x0 + x1
        return y
```

➔ 포인터 X.

➔ Add 클래스를 구현하기 쉽게함.

Function 클래스 수정 :

-리스트 언팩 사용 :

함수 호출할 때 *을 붙임(리스트를 다 풀어서 전달.)

Add 클래스를 파이썬 함수로 변환 :

Add클래스 대신 add함수를 사용.

Add 함수를 사용한 계산 코드.

```
x0 = Variable(np.array(2))
x1 = Variable(np.array(3))
y = add(x0, x1)
print(y.data)
```

```
x0 = Variable(np.array(2))
x1 = Variable(np.array(3))
y = Add()(x0, x1)
print(y.data)
```

➔ 순전파 구현.

```
class Variable:
    def __init__(self, data):
        self.data = data
        self.grad = None
        self.creator = None

    def set_creator(self, func):
        self.creator = func

    def backward(self):
        if self.grad is None:
            self.grad = np.ones_like(self.data)

        funcs = [self.creator]
        while funcs:
            f = funcs.pop()  # 1. Get a function
            x, y = f.input, f.output  # 2. Get the function's input/output
            x.grad = f.backward(y.grad)  # 3. Call the function's backward

            if x.creator is not None:
                funcs.append(x.creator)
```

```
    def __init__(self, data):
        self.data = data
        self.grad = None
        self.creator = None

    def set_creator(self, func):
        self.creator = func

    def backward(self):
        if self.grad is None:
            self.grad = np.ones_like(self.data)

        funcs = [self.creator]
        while funcs:
            f = funcs.pop()
            gys = [output.grad for output in f.outputs]
            gxs = f.backward(*gys)
            if not isinstance(gxs, tuple):
                gxs = (gxs,)

            for x, gx in zip(f.inputs, gxs):
                x.grad = gx

            if x.creator is not None:
                funcs.append(x.creator)
```

➔ 한 변수만 처리 가능한 거를 다변수 처리로 바꿈.

➔ 평 미분 값을 전부 하나로 묶음(zip함수 사용 -> 미분을 하나로 모으기 위해 사용).

Square 클래스 가변 길이 입출력 지원으로 개선 :

Square 클래스를 새로운 Variable과 Function 클래스에 맞게 수정

같은 변수를 반복 사용시 문제점 :

Backward에서 출력 쪽에서 전해지는 미분 값을 그대로 대입

같은 변수 반복하여 사용시 전파되는 미분 값이 덮어 써짐.

문제점 해결 방법 :

미분 값(grad)을 처음 설정하는 경우는 출력에서 전해지는 미분 값을 그대로 대입.

처음 이후부터는 전달된 미분 값을 더해주도록 수정.

역전파시 미분 값을 더해주는 코드 문제점 해결 :

```
x = Variable(np.array(3.0))
y = add(x, x)
y.backward()
print('1. x.grad = ', x.grad)

y = add(x, x)
y.backward()
print('2. x.grad = ', x.grad)
```

```
1. x.grad =  2.0
2. x.grad =  4.0
```

```
x = Variable(np.array(3.0))
y = add(x, x)
y.backward()
print('1. x.grad = ', x.grad)

x.cleargrad()
y = add(x, x)
y.backward()
print('2. x.grad = ', x.grad)
```

```
1. x.grad =  2.0
2. x.grad =  2.0
```

➔ Cleargrad 메서드 추가(Variable 클래스에 미분 값을 초기화).

가변 길이 인수(역전파)

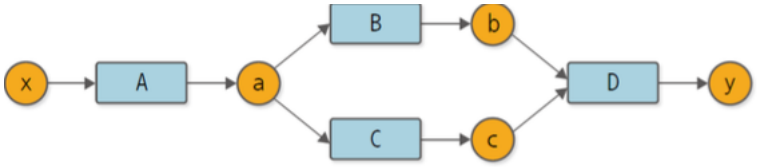
덧셈의 역전파 : 상류에서 흘러나오는 미분 값을 그대로 흘러보냄(가중치 업데이트).

여러 개의 변수에 대응할 수 있도록 수정 :

출력 변수(outputs)에 담겨 있는 미분 값들을 라스트에 포함.

복잡하게 연결된 그래프의 올바른 순서 :

변수를 반복해서 사용하면 역전파 때는 출력 쪽에서 전파하는 미분 값을 더해야 함.



➔ D -> B or C -> A 순으로 처리 (올바른 순서)

DeZero가 14단계까지 구현 사항 파악 :

Func 리스트 구현 부분 파악 필요

역전파의 흐름 파악 : 다음에 처리할 함수를 그 리스트의 끝에서 꺼냄.

```
class Variable:
    def __init__(self, data):
        self.data = data
        self.grad = None
        self.creator = None

    def set_creator(self, func):
        self.creator = func

    def backward(self):
        if self.grad is None:
            self.grad = np.ones_like(self.data)

        funcs = [self.creator]
        while funcs:
            f = funcs.pop()
            gys = [output.grad for output in f.outputs]
            gxs = f.backward(*gys)
            if not isinstance(gxs, tuple):
                gxs = (gxs,)

            for x, gx in zip(f.inputs, gxs):
                x.grad = gx

            if x.creator is not None:
                funcs.append(x.creator)
```

➔ Func.pop() : 마지막 함수 값 소환

함수 우선 순위 설정 :
함수와 변수의 세대를 기록.
세대가 우선 순위에 해당.
역전파 시 세대 수가 큰 쪽부터 처리하면 부모 보다 자식이 먼저 처리.

순전파시 세대 추가 :

Variable, Function 클래스에 인스턴스 변수 generation을 추가.

Variable 클래스 수정 :

Generation을 0으로 초기화. / set_creator 메서드 호출될 때 함수의 세대 보다 1만큼 큰 값을 설정.

```
        self.generation = 0

    def set_creator(self, func):
        self.creator = func
        self.generation = func.generation + 1
```

Variable 클래스의 backward 메서드 구현 :

Backward 메서드에서 중첩 메서드 add_func 함수(함수 리스트를 세대 순으로 정렬) 추가.
정렬이 되어서 func.pop()을 수행시 세대가 가장 큰 함수를 꺼냄.

```
    funcs = []
    seen_set = set()

    def add_func(f):
        if f not in seen_set:
            funcs.append(f)
            seen_set.add(f)
            funcs.sort(key=lambda x: x.generation)

    add_func(self.creator)
```

➔ 중첩 메서드 추가 / 세대 별로 분류.

(5주차)

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(세대를 기준으로 쓸모 없어진 객체 회수하는 방식 / 자바에도 있음)

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

순환 참조

참조 카운트로는 해결할 수 없는 문제

GC(가비지 컬렉션)

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 좋다

약한 참조

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
self.outputs = outputs
 self.outputs = [weakref.ref(output) for output in outputs]
 return outputs if len(outputs) > 1 else outputs[0]
```

```
  funcs = [self.creator]
  while funcs:
    f = funcs.pop()
    #x,y = f.input, f.output
    #x.grad = f.backward(y.grad)

    gys = [output().grad for output in f.outputs]
    gxs = f.backward(*gys)
```

메모리 절약모드

역전파 시 사용하는 메모리 양을 줄이기

➔ 불필요한 미분값 제거

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

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

순환참조 해결

메모리절약(미분값 제거, 불필요한 역전파 생략)

변수 사용성 개선

Variable 클래스를 더욱 쉽게 사용하도록 개선

변수 이름 지정

Variable 인스턴스를 ndarray 인스턴스처럼 보이게 함

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

    self.data = data
    self.name = name
    self.gard = None # grad : 기울기(미분값 계산해서 대입)
    self.creator = None
    self.generation = 0
```

@property는 shape메서드

@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, ize, dtype 등을 변수처럼 사용 가능

len함수와 print함수

특수 메서드(__len__ , __repr__)를 구현

➔ 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

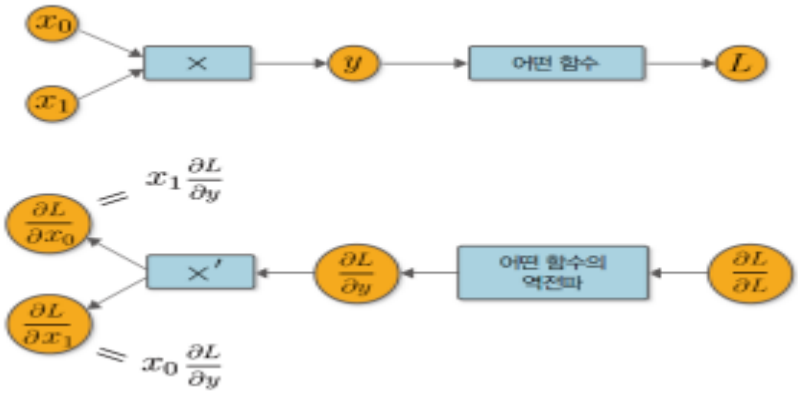
➔ y = a * b

➔ 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

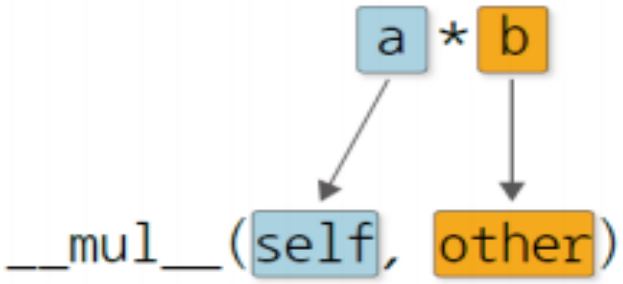
```
class Mul(Function):
    def forward(self, x0, x1):
        y = x0 * x1
        return y

    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
```

Varialbe을 ndarray 인스턴스와 함께 사용

ndarray 인스턴스를 자동으로 Variable 인스턴스로 변환

as_variable 함수를 이용

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

변수가 float, int인 경우 ndarray 인스턴스로 변환

as_array 함수를 이용

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

문제점

1)첫 번째 인수가 float, int인 경우

TypeError 발생

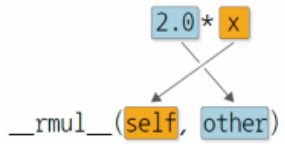
오류 발생 과정

- ➔ 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
- ➔ 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 피연산자의 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

2)좌항이 ndarray 인스턴스인 경우

좌항은 ndarray 인스턴스의 __add__메서드를 호출하지만

우항은 Variable 인스턴스의 __radd__메서드가 호출되어야 함

Variable 인스턴스의 속성에 __array_priority__를 추가

- ➔ 그 값을 큰 수로 설정하여 연산자 우선순위를 지정해야 함

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 곱한 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

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
```

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

느낀점

아직 적응은 되지 않았지만 이제 뭐를 어떻게 해야할지는 알게 되었습니다.