정리노트(4주차)

산업데이터사이언스학부
201904236
전병준

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

```
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)
```

```
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

```
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)
```

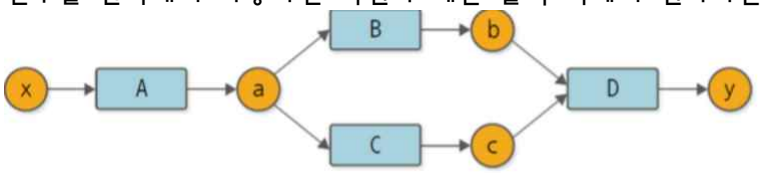
```
1. x.grad =  2.0
2. x.grad =  2.0
```

#Cleargrad 메서드 추가(Variable 클래스에 미분 값을 초기화)

가변 길이 인수(역전파)
덧셈의 역전파
상류에서 흘러나오는 미분 값을 그대로 흘려보냄(가중치 업데이트).

여러 개의 변수에 대응할 수 있도록 수정
출력 변수(outputs)에 담겨 있는 미분 값들을 리스트에 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

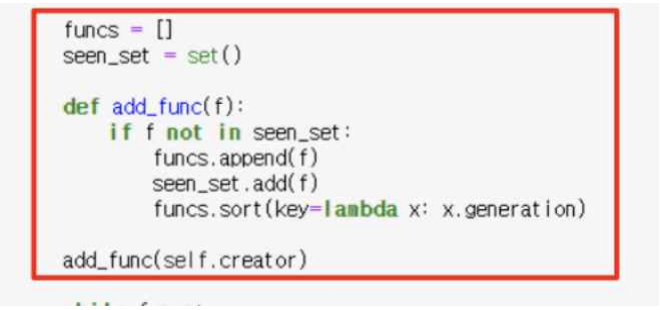
Variable 클래스 수정
Generation을 0으로 초기화. / set_creator 메서드 호출될 때 함수의 세대 보다 1만큼 큰 값을 설정

```
        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.