

O'REILLY®

파이썬으로 직접 구현하며 배우는 딥러닝 프레임워크

w1



사이토 고키 저음  
개일염시 옮김

# Book Review & Project

2021.3.2 ~ 3.18

한국IT교육원 303호



# 슬라이드 1

---

w1

색상코드 RGB(215, 1, 57)

w, 2021-02-26

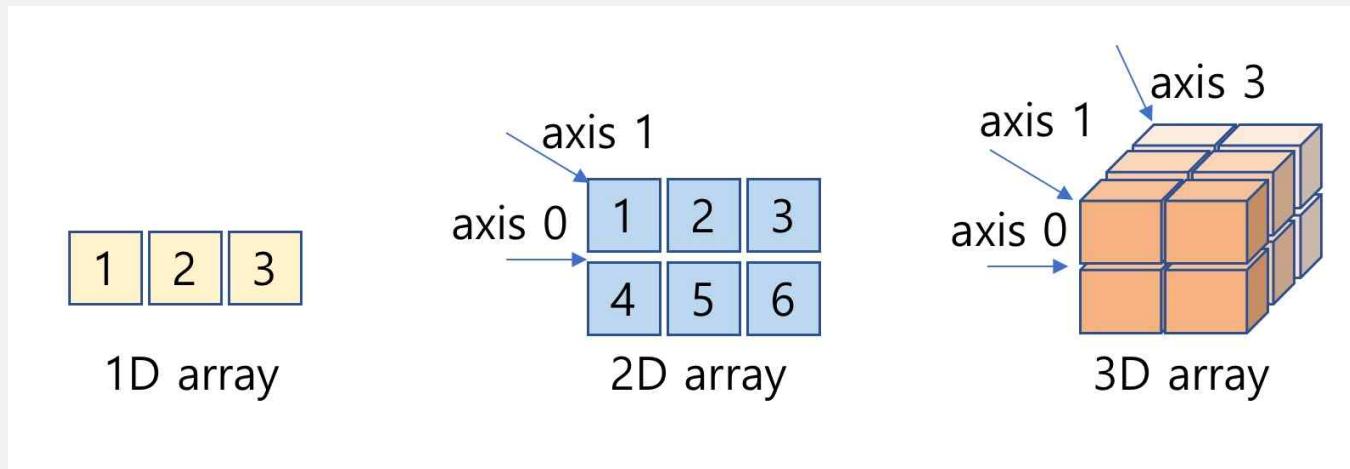
# CONTENTS

- |        |            |
|--------|------------|
| 제 1 고지 | 미분 자동 계산   |
| 제 2 고지 | 자연스러운 코드로  |
| 제 3 고지 | 고차 미분 계산   |
| 제 4 고지 | 신경망 만들기    |
| 제 5 고지 | DeZero의 도전 |
-

# 제 1고지 미분 자동 계산

## 1단계: 상자로서의 변수 & steps/steps01.py w2

- Variable 클래스 구현 (\*PEP8 규칙 참조)
- ML 시스템에서 기본 데이터 구조는 “다차원 배열“



〈출처: [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)〉

## 슬라이드 4

---

w2

색상코드: RGB(62,125,51)

w, 2021-02-26

## 2단계: 변수를 넣는 함수 & steps/steps02.py

- Function 클래스 구현
  - `__call__(self, input)` 메서드의 인수 `input`은 Variable 인스턴스라 가정\*
  - `__call__` 메서드를 정의하면 `f = Function()` 형태로 사용 가능  
: 예) `f(...)` 형태로 `__call__` 메서드 호출

\* <https://tech.ssut.me/python-functions-are-first-class/>

## 3단계: 함수 연결 & steps/steps03.py

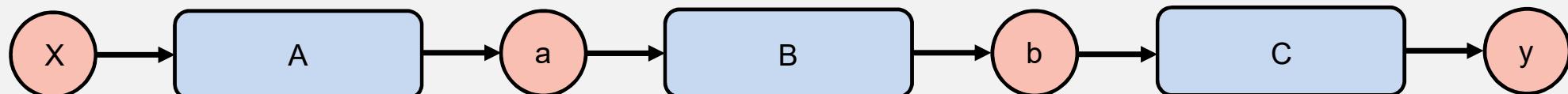
- Exp 함수 구현

- $y = e^x$  계산하는 함수 구현

- :  $e$ 는 자연로그의 밑, 2.718…

- Square() & Exp() 모두, Function 클래스 상속\*

- 합성 함수(Composite Function)

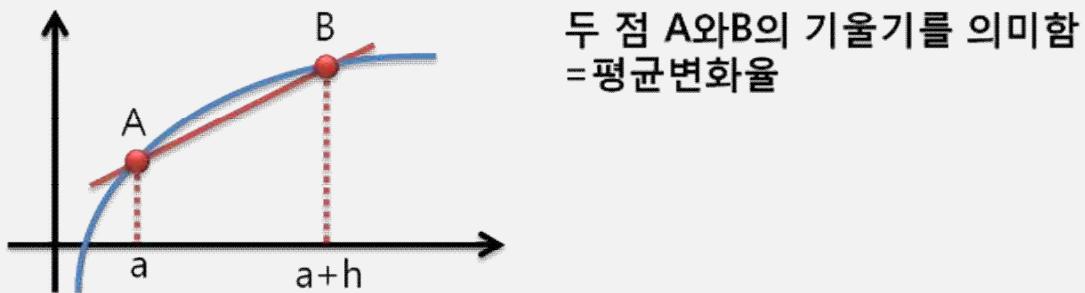


## 4단계: 수치 미분 & steps/steps04.py

- 미분 (Derivative)

- 변화율 (예, 물체의 시간에 따른 위치 변화율)\*

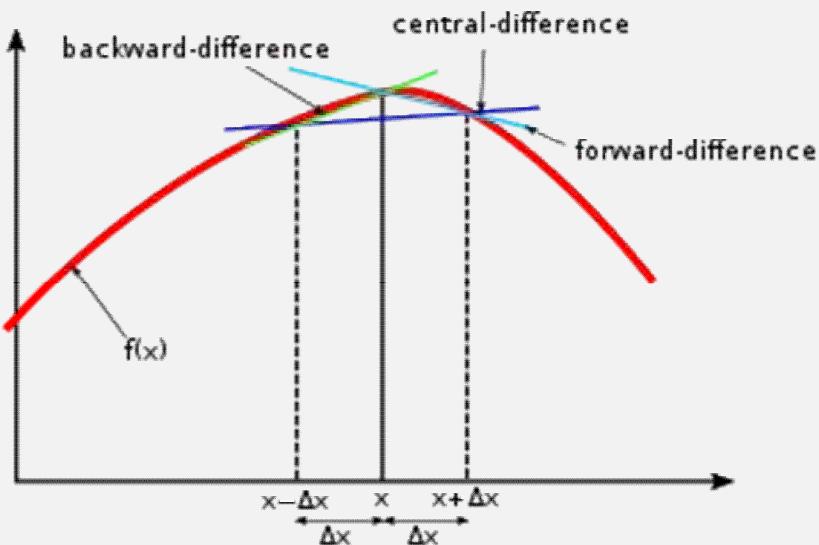
$$\text{평균변화율: } \frac{f(a+h) - f(a)}{(a+h) - a} = \frac{f(a+h) - f(a)}{h}$$



## 4단계: 수치 미분 & steps/steps04.py

- 미분 (Derivative)

- 변화율 (예, 물체의 시간에 따른 위치 변화율)\*
- 중앙차분 (Centered Difference)



\* <https://j1w2k3.tistory.com/295>

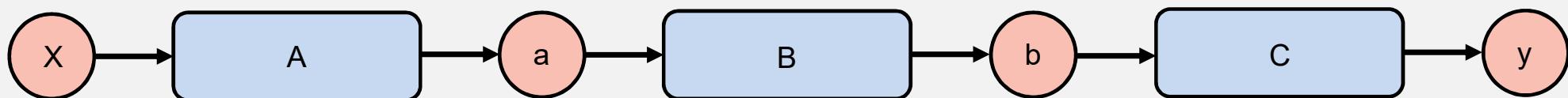
## 4단계: 수치 미분 & steps/steps04.py

- 미분 (Derivative)
  - 변화율 (예, 물체의 시간에 따른 위치 변화율)
  - 중앙차분 (Centered Difference)
  - $\text{eps} = \text{가장 작은 값} = 1e-4^*$
  - 결괏값 3.297의 의미:  $x$ 를 0.5만큼 변화하면 3.297 만큼 변한다는 의미
- 수치 미분의 문제점
  - 오차가 존재 (자릿수 누락, 예: 1.233와 1.2333은 다른 숫자 이지만, 유효숫자 처리에 따라 같아 질 수 있음)
  - 계산량이 많아짐. 특히, 신경망에서는 모두 미분으로 구할 수 없음

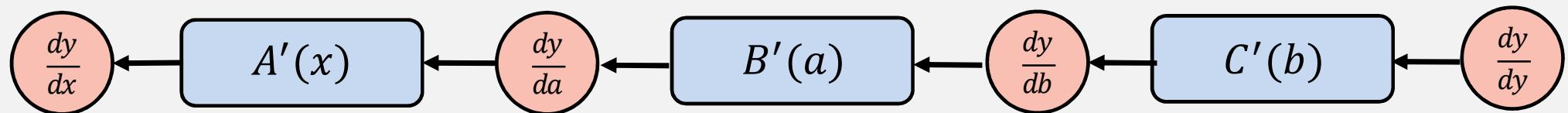
\* 엡실론은 수학에서 가장 작은 양의 값을 나타내는데 사용, 컴퓨터에서 아주 작은 양의 부동소수점 값을 담는 변수의 이름으로 사용 중

## 5단계: 역전파

- 순전파



- 역전파



제 1고지

제 2고지

제 3고지

제 4고지

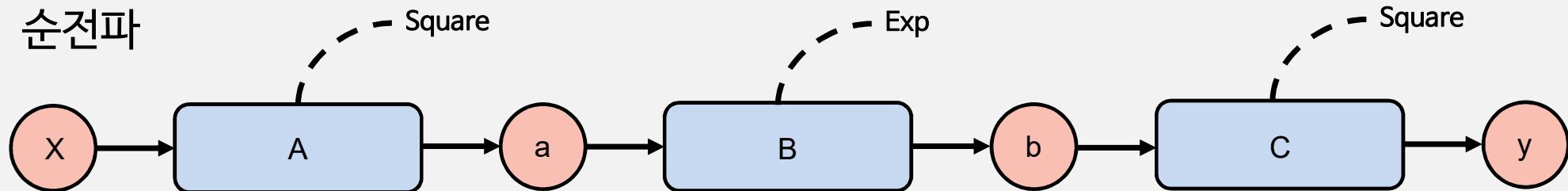
제 5고지

참고자료

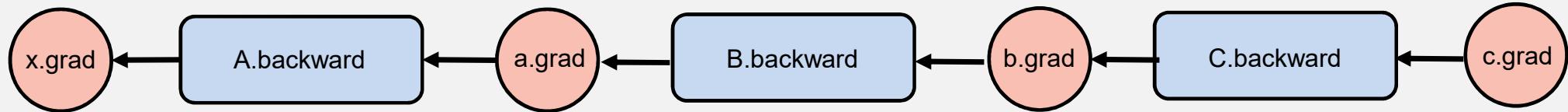
## 5단계: 역전파

## 6단계: 역전파 & steps/steps06.py

- 순전파

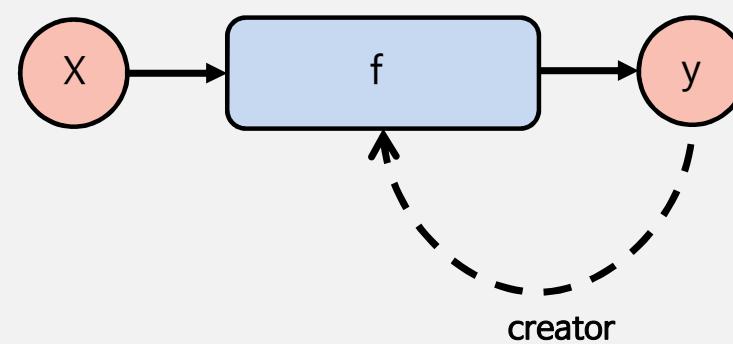
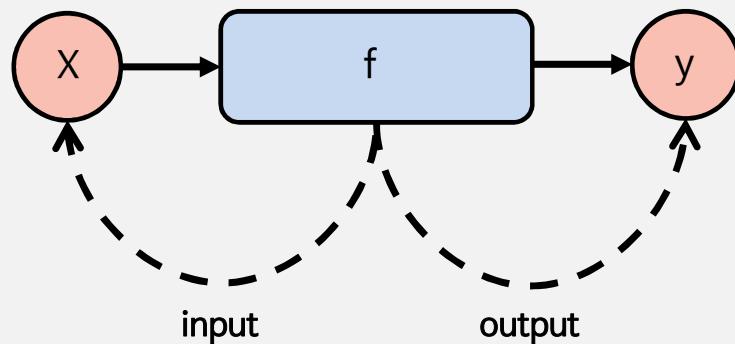


- 역전파



## 7단계: 역전파 자동화 & steps/steps07.py

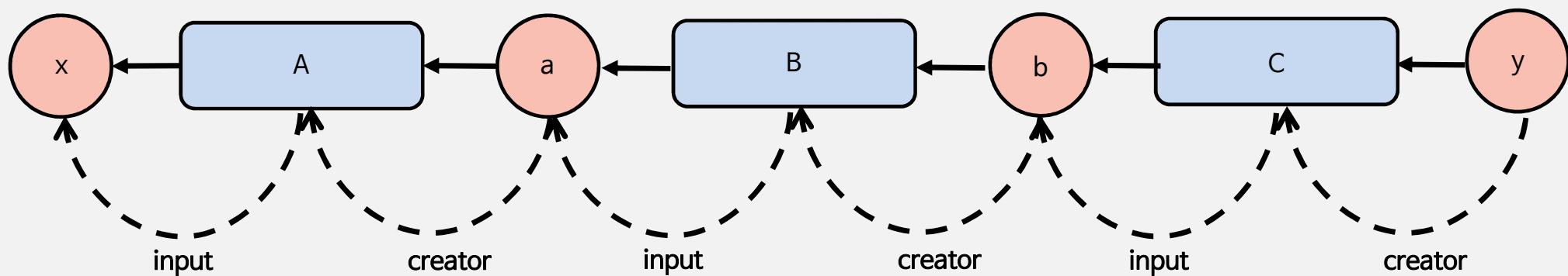
- 변수와 함수와의 관계



- assert 문: 평가 결과가 True가 아니면 예외 발생

## 7단계: 역전파 자동화 & steps/steps07.py

- 계산 그래프 역 추적



## 8단계: 재귀에서 반복문으로 & steps/steps08.py, helper/recursive\_call.py

- 반복문의 장점
  - 재귀는 함수를 재귀적으로 호출할 때마다 중간 결과를 메모리에 유지 스택에 쌓으면서 처리
  - 처리 효율도 우수
  - 이 때, 반복문에서는 For-Loop 대신, while 반복문 사용

## 9단계: 함수를 더 편리하게 & steps/steps09.py

- 파이썬 함수로 이용하기
  - square & exp 함수 구현 완료
  - 함수를 연속으로 적용 가능

```
...  
y = square(exp(square(x)))  
...
```

## 9단계: 함수를 더 편리하게 & steps/steps09.py

- Backward 메서드 간소화

```
...
def backward(self):
    if self.grad is None:
        self.grad = np.ones_like(self.data)
    funcs = [self.creator]
...
...
```

- grad가 None이면 자동으로 미분값 생성, self.data가 스칼라이면, self.grad가 스칼라가 됨
- np.ones\_like(): Variable data와 grad의 데이터 타입을 같게 만들도록 하기 위한 것  
: 예) self.data 32비트 부동소수점이면 grad도 32비트 부동 소수점

## 9단계: 함수를 더 편리하게 & steps/steps09.py

- o ndarray만 취급하기

```
...
def __init__(self, data):
    if data is not None:
        if not isinstance(data, np.ndarray):
            raise TypeError('{}은(는) 지원하지 않습니다.'format(type(data)))
...
...
```

- as\_array( ) 함수: Variable은 항상 ndarray( ) 가정하고 있어서, ndarray 인스턴스로 변환

## 10단계: 테스트 & steps/steps10.py

- Unittest의 사용
- Square 함수의 역전파 테스트
- 기울기 확인(Gradient checking)을 이용한 자동 테스트
  - $|a - b| \leq (\text{atoll} + \text{rtol} \times |b|) \Rightarrow$  결괏값이 True 반환해야 함

```
w@HKIT203-LECT MINGW64 ~/Desktop/dl_framework (main)
$ python -m unittest steps/step10.py
```

```
...
```

```
Ran 3 tests in 0.008s
```

```
OK
```

제 2고지 자연스로운 코드로

## 11, 12단계 - 가변 길이 인수(순전파) & steps/step11~12.py

- Function 클래스 수정
  - 인수와 반환값의 타입을 리스트로 변환
  - 리스트 내포(list comprehension) 사용됨
  - 인수 앞에 별표(\*) 붙임 -> 임의 개수의 인수(가변 길이 인수)를 건네 함수를 호출
  - 함수를 호출할 때 별표를 붙이면 리스트 언팩(list unpack)이 일어남\*
- : 예) xs = [x0, x1] 일 때 self.forward(\*xs)를 하면 self.forward(x0, x1)로 호출하는 것과 동일하게 동작

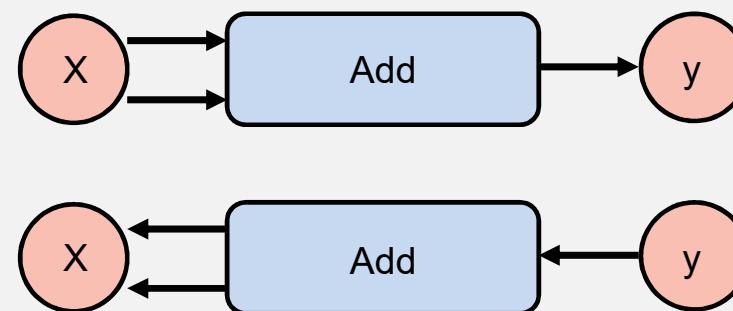
\* <https://dojang.io/mod/page/view.php?id=2345>

## 13단계 - 가변 길이 인수(역전파) & steps/step13.py

- Variable 클래스 수정
  - 여러 개의 변수에 대응할 수 있도록 수정됨
    - ① 출력 변수인 outputs에 담겨 있는 미분값들을 리스트에 담음
    - ② 함수 f의 역전파를 호출 - `f.backward(*gys)` 인수에 별표를 붙여 호출
    - ③ gxs가 튜플이 아니라면 튜플로 변환
    - ④ 역전파로 전파되는 미분값을 Variable의 인스턴스 변수 grad에 저장함

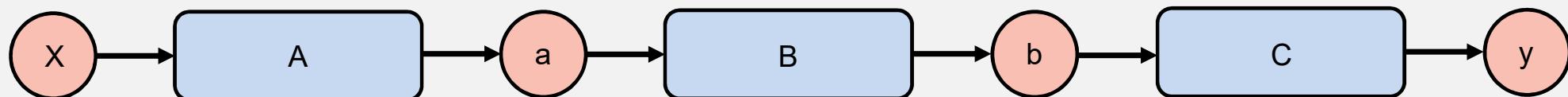
## 14단계: 같은 변수를 반복 사용 & steps/step13~14.py

- 문제 (1) : 역전파에서 미분값을 더해주는 과정에서 미분값을 그대로 대입  
=> 미분의 합
  - 처음일 경우 : 지금과 같이 미분값 대입
  - 두번째 이후 : 이전 미분값과 전달받은 미분값을 더함
- 문제 (2) : 같은 변수로 다른 계산을 할 때  
=> 미분값 초기화하는 cleargrad에서도 추가

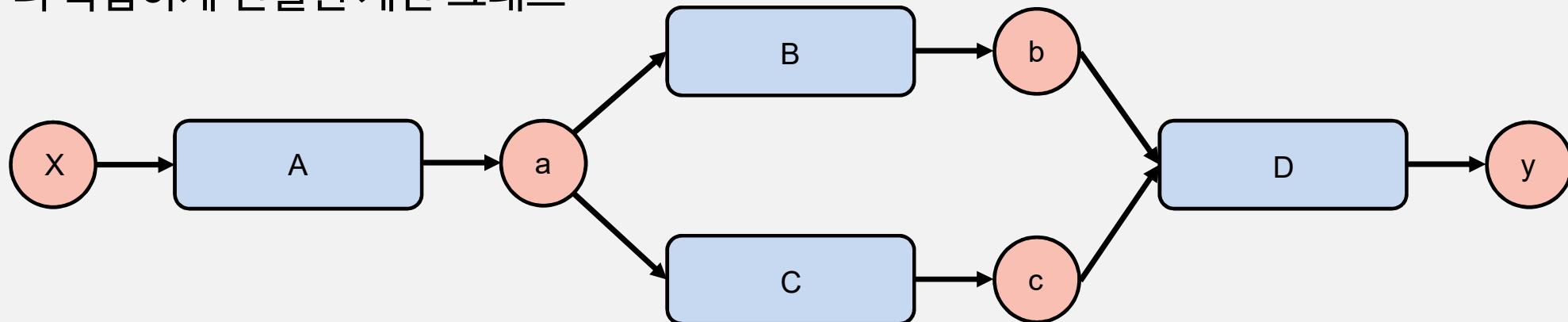


## 15~16단계: 복잡한 계산 그래프

- 일직선 계산 그래프 (앞에서 해왔던 것)



- 더 복잡하게 연결된 계산 그래프



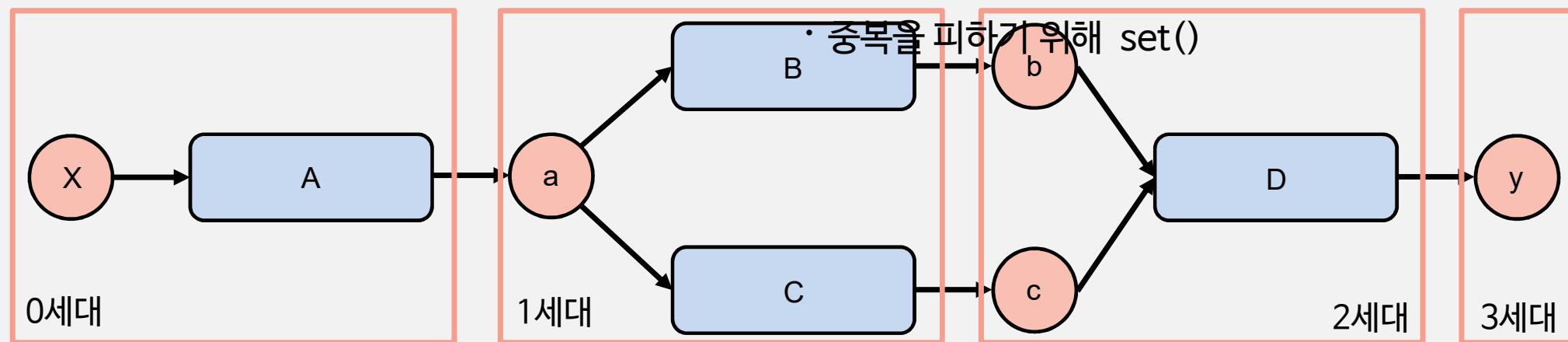
## 15~16단계: 복잡한 계산 그래프 & steps/step15~16.py

- 역전파의 올바른 순서

- 같은 변수를 사용할 때 역전파는 출력쪽에서 전파되는 미분값을 더해야함
- 함수B와 C의 역전파를 모두 끝내고 함수A를 역전파 해야함. (B와 C의 순서는 상관없음)

### 구현방법

- 세대(generation) 추가 : 순전파에서 사용하는 함수의 순서를 기억
- 세대순으로 꺼내기 : 최근 세대의 함수부터 꺼내도록 함



## 17단계: 메모리 관리 및 순환 참조.

- 메모리 관리 방식 2가지
  1. 참조(Reference) 수 Count
  2. 세대(Generation)를 기준으로 쓸모없어진 객체(garbage)를 회수(collection)

## 17단계: 메모리 관리 및 순환 참조.

- 참조(Reference) 수 Count
  1. 모든 클래스 객체는 생성 시 참조 카운트가 0인 상태로 생성
  2. 생성된 객체를 참조하는 프로그래밍 문법이 발생할 경우 카운트 1만큼 증가 (Edge 생성)  
※ 참조 카운트가 증가되는 경우(일부)
    - 대입 연산자 사용
    - 함수에 인수로 전달
    - 컨테이너 (리스트, 튜플, 클래스 등)에 추가할 경우
  3. 이와 대조적으로 객체에 대한 참조가 끊어질 경우 (Edge 소멸) 참조 카운트 1만큼 감소

## 17단계: 메모리 관리 및 순환 참조.

- 참조(Reference) 수 Count

```
# 객체 생성 (참조 카운트 0)
```

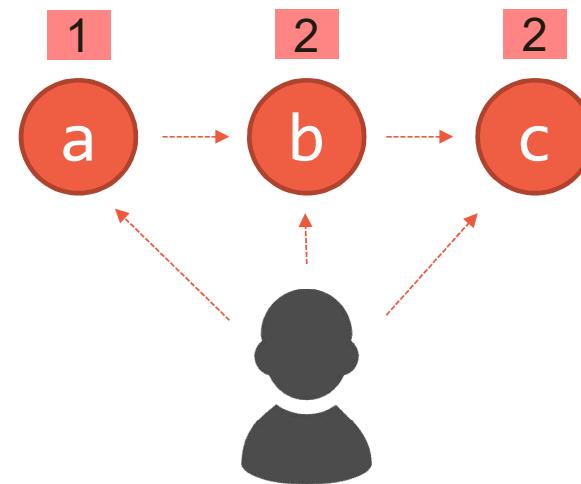
```
a = obj()  
b = obj()  
c = obj()
```

```
# 대입 연산자 사용
```

```
a.b = b  
b.c = c
```

```
# 대입 연산자를 활용하여 객체 소멸
```

```
a = b = c = None
```



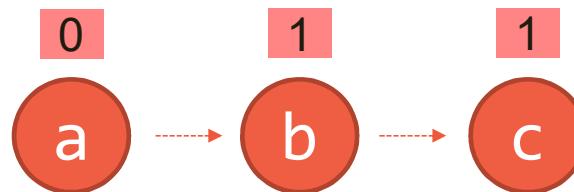
어떤 경우에 의해 참조된 경우 참조 수 Count. b와 c는 각각 대입연산자와 class 멤버변수로 호출되어 2번 참조됨

## 17단계: 메모리 관리 및 순환 참조.

- 참조(Reference) 수 Count

```
# 객체 생성 (참조 카운트 0)
```

```
a = obj()  
b = obj()  
c = obj()
```



```
# 대입 연산자 사용
```

```
a.b = b  
b.c = c
```



```
# 대입 연산자를 활용하여 객체 소멸
```

```
a = b = c = None
```

참조되어 인스턴스화 되었던 a, b, c 클래스에 대해 메모리 해제 작업 시 참조 수 Count 확인

a, b, c가 각각 0, 1, 1의 값을 처음에 나타내고 b, c는 순차적으로 앞 노드(클래스 객체)에 의해 메모리 할당이 해제됨

## 17단계: 메모리 관리 및 순환 참조.

- 순환 참조\*

```
# 객체 생성 (참조 카운트 0)
```

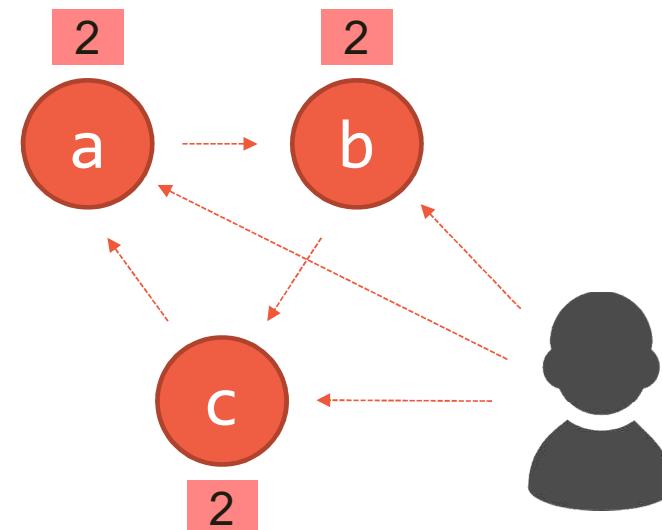
```
a = obj()  
b = obj()  
c = obj()
```

```
# 대입 연산자 사용
```

```
a.b = b  
b.c = c  
c.a = a
```

```
# 대입 연산자를 활용하여 객체 소멸
```

```
a = b = c = None
```



a, b, c 객체가 서로 순환하는 구조를 가진 경우 참조 count.

## 17단계: 메모리 관리 및 순환 참조.

- 순환 참조

```
# 객체 생성 (참조 카운트 0)
```

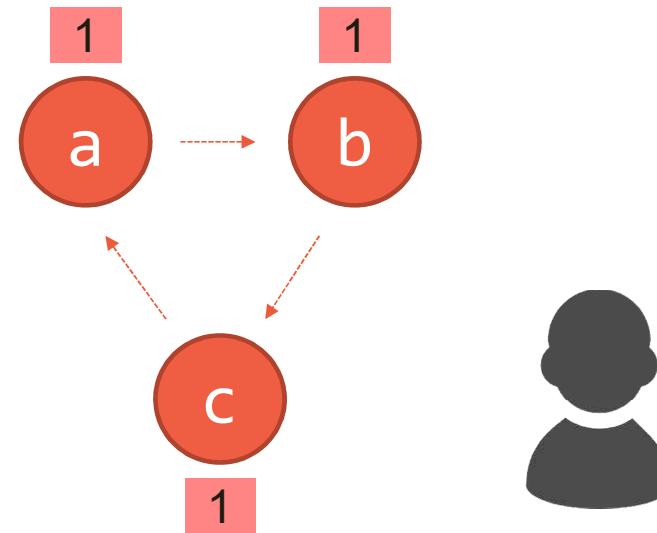
```
a = obj()  
b = obj()  
c = obj()
```

```
# 대입 연산자 사용
```

```
a.b = b  
b.c = c  
c.a = a
```

```
# 대입 연산자를 활용하여 객체 소멸
```

```
a = b = c = None
```



세 객체에 대해 메모리 해제를 시도했으나, 순환 참조에 의해 메모리 해제가 불가. (메모리 누수)

## 17단계: 메모리 관리 및 순환 참조.

- weakref

```
import weakref
import numpy as np

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

b # b 메모리 주소 확인
<weakref at 0x103b7f048; to 'numpy.ndarray' at 0x103b67e90>
b() # b 객체 내장데이터 확인
[1 2 3]
```

weakref 모듈을 사용하여 약한 연결고리를 가지는 순환 참조 객체를 생성

## 17단계: 메모리 관리 및 순환 참조.

- weakref

```
>>> import weakref
>>> import numpy as np

>>> a = np.array([1, 2, 3])
>>> b = weakref.ref(a) # weakref 객체 생성

>>> b # b 메모리 주소 확인
<weakref at 0x103b7f048; to 'numpy.ndarray' at 0x103b67e90>
>>> b() # b 객체 내장데이터 확인
[1 2 3]

>>> a = None
>>> b # a 컨테이너에 대한 참조가 사라지면서 weakref status가 dead로 변경
<weakref at 0x103b7f048; dead>
```

## 17단계: 메모리 관리 및 순환 참조.

- weakref & step/step17.py

```
import weakref

class Function:
    def __call__(self, *inputs):
        ...
        self.outputs = [weakref.ref(output) for output in outputs]
        ...

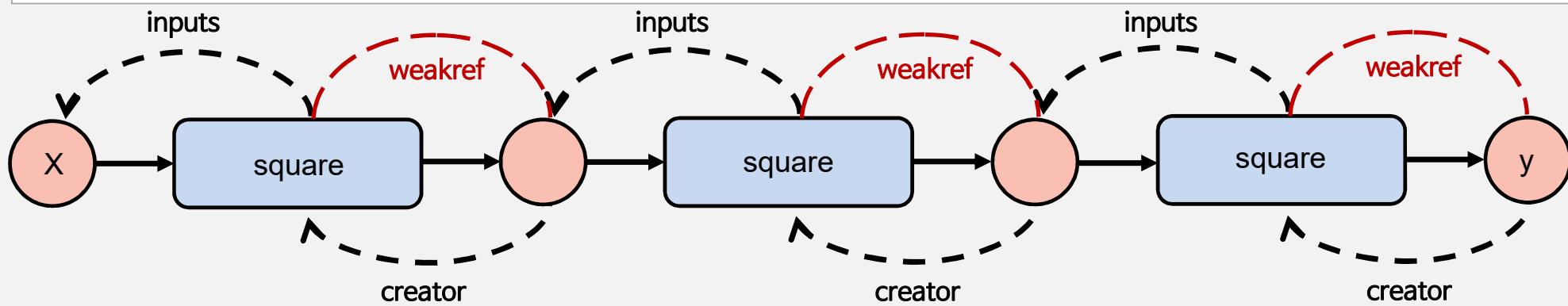
class Variable:
    ...
    def backward(self):
        ...
        gys = [output().grad for output in f.outputs]
        ...
```

## 17단계: 메모리 관리 및 순환 참조.

- weakref & step/step17.py

```
import numpy as np

for i in range(10):
    x = Variable(np.random.randn(10000))
    y = square(square(square(x)))
```



## 17단계: 메모리 관리 및 순환 참조.

- Garbage Collection \*
- 세대(Generation)를 기준으로 쓸모없어진 객체(garbage)를 회수(collection)
- 장점: (3가지 케이스에 대해서) 메모리 수집 편리
  - 1. 유효하지 않은 포인터에 대한 접근
  - 2. 이중 해제
  - 3. 메모리 누수
- 단점
  - 1. GC에 의존하므로 메모리 해제 시점을 알아도 GC를 사용하므로 오버헤드 발생
  - 2. 메모리의 해제 시점을 정확하게 판단할 수 없다.

\* Wikipedia.org → Garbage Collection

## 18단계: 메모리 절약 모드

- 할당된 메모리 해제 작업

```
class Variable:  
    ...  
    def backward(self, retain_grad = False):  
        ...  
        if not retain_grad:  
            for y in f.outputs:  
                y().grad = None # 불필요한 메모리 해제,  
                                # y의 기울기는 Framework 계산 시를 제외하고 활용되지 않음
```

Variable 클래스의 backward 메서드에 retain\_grad 인자를 추가하는 것만으로도 불필요한 메모리 사용을 억제하는 효과

## 18단계: 메모리 절약 모드

- Config 클래스 활용 - Flag 설정

```
class Config:  
    enable_backprop = True  
  
class Function:  
    def __call__(self, *inputs):  
        ...  
        if Config.enable_backprop:  
            ...  
        ...
```

설정 데이터는 단 한 군데 지정하는게 전체적인 측면에서 유리하므로 Config 클래스를 인스턴스화하지 않고 ‘클래스’ 그대로 활용

## 18단계: 메모리 절약 모드

- with 문을 활용한 모드 전환
  - Python의 들여쓰기를 활용하여 후처리를 자동으로 진행해주는 구문 생성
  - 대표적인 예: with < open, close >
  - 데코레이션(@)을 활용하여 [contextlib.contextmanager](#)에 접근

## 18단계: 메모리 절약 모드

- with 문을 활용한 모드 전환

```
import contextlib

@contextlib.contextmanager
def using_config(name, value):
    old_value = getattr(Config, name) # 기존 설정내용 저장
    setattr(Config, name, value) # 새로운 설정내용으로 작업 수행
    try:
        yield # with문으로 호출 시 body 부분 작업을 수행하는 영역
    finally:
        setattr(Config, name, old_value) # 기존 설정내용으로 복구
```

contextlib 모듈 내에서 callable 함수 객체를 받아서 decorator에 의해 처리

위 코드의 경우 기존 설정을 백업해둔 상태로 새로운 작업을 수행하고 복구하는 함수.

## 18단계: 메모리 절약 모드

- with 문을 활용한 모드 전환

```
with using_config('enable_backprop', False): # Config class's 멤버변수(str), name(bool)
    x = Variable(np.array(2.))
    y = square(x)
```



설정 간소화

```
Def no_grad():
    return using_config('enable_backprop', False)
```

```
with no_grad():
    x = Variable(np.array(2.))
    y = square(x)
```

with문을 활용하여 후처리 진행 + 반복 수행 시 번거로움을 방지하고자 no\_grad 함수 생성

## 19단계: 변수 사용성 개선 & steps/step19.py

- 변수 이름 지정
  - 변수의 구분을 위해 변수 이름을 저장
  - 계산 그래프를 시각화 할 때 변수이름을 그래프에 표시할 수 있음
  - 지정하지 않을 경우 None로 할당
- ndarray 인스턴스 변수
  - shape : 다차원 배열의 형상
  - ndim : 차원 수
  - size : 원소 수
  - dtype : 데이터 타입
  - @property라는 데코레이터를 사용해 메서드를  
인스턴스 변수처럼 사용  
ex) x.shape

## 19단계: 변수 사용성 개선 & steps/step19.py

- len 함수와 print 함수
  - len : 리스트 등의 안에 포함된 원소 수를 반환
  - print : Variable안에 데이터 내용을 출력하는 기능
  - \_\_len\_\_와 같이 특수 메서드로 정의하여 함수처럼 사용  
ex) len(x)

## 20단계: 연산자 오버로드(1) & steps/step20.py

- 연산자 오버로드(operator overlaod)
    - 연산자(+, \*등) 대응
    - 특수 메서드를 정의함으로써 사용자 지정 함수가 호출되도록 함
      - ex) `__mul__ = > *`
      - `__add__ = > +`
    - 파이썬에서는 함수도 객체이므로 함수 자체를 할당할 수 있음
      - ex) Variable 의 `__mul__` 를 호출할 때 파이썬의 mul함수 부름
- Variable.`__add__` = add
- Variable.`__mul__` = mul

## 21단계: 연산자 오버로드(2) & steps/step21.py

- 연산자 오버로드(operator overload)

- 앞에서  $a * b$  or  $a + b$  같은 연산자를 사용한 코드 작성할 수 있게 됨
- 1)  $a * np.array(2.0)$  같은 ndarray 인스터스와 사용할 수 없음
- 2)  $a + 3$  같은 수치 데이터와 사용할 수 없음



ndarray, int, float도 사용할 수 있게 하자

## 21단계: 연산자 오버로드(2) & steps/step21.py

- ndarray, float, int와 함께 사용하기
  - ndarray 인스턴스를 Variable인스턴스로 변환 -> Variable인스턴스로 통일
  - float, int를 ndarray 인스턴스로 변환 -> 이후 Funcrtion 클래스에서 Variable 인스턴스로 다시변환되기 때문에 결과적으로 전부 Variable인스턴스로 통일됨
  - 하지만 지금의 이항 연산자 \* 는 좌항의 인스턴스에 속해있는 메서드를 통해서만 호출됨

\* 이항 연산자 : 연산을 수행하는 피연산자(a, b같은)가 두개일 때의 연산자

## 21단계: 연산자 오버로드(2) & steps/steps22.py

- 좌항에 int, float가 있을 경우

- 구현되어 있지 않기 때문에 특수 메서드를 호출하지 못함

해결 : 피연산자의 위치에 따라 호출되는 특수 메서드를 지정하자

ex) 곱셈의 경우

- 피연산자가 좌항 -> '\_\_mul\_\_'
  - 피연산자가 우항 -> '\_\_rmul\_\_' (r은 right의 r)

## 21단계: 연산자 오버로드(2) & steps/steps22.py

- 좌항에 ndarray 가 있을 경우

- 좌항의 ndarray의 인스턴스를 통해 연산자가 호출됨

하지만 Variable 인스턴스로 통일 해주어야 해서 Variable 인스턴스의 특수메서드를 호출하길 원함

해결 : 연산자의 우선순위를 두자

Variable 인스턴스의 연산자 우선순위를 ndarray 인스턴스의 연산자 우선순위보다 높일 수 있음

ex) 덧셈의 경우

```
...
```

```
class Variable:
```

```
    __array_priority__ = 200
```

```
...
```

## 22단계: 연산자 오버로드(3) & steps/steps23.py

- 특수 메서드

특수 메서드	설명	예
<code>_neg_(self)</code>	부호 변환 연산자	<code>-self</code>
<code>_sub_(self, other)</code>	뺄셈 연산자	<code>self - other</code>
<code>_rsub_(self, other)</code>	역순 뺄셈 연산자	<code>other - self</code>
<code>_truediv_(self, other)</code>	나눗셈 연산자	<code>self / other</code>
<code>_rthredic_(self, other)</code>	역순 나눗셈 연산자	<code>other / self</code>
<code>_pow_(self, other)</code>	거듭제곱 연산자	<code>self ** other</code>

## 23단계 - 패키지로 정리 & steps/step13.py

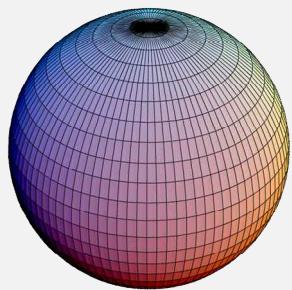
- 모듈 \*
  - 파이썬 파일, 특히 임포트(import)하여 사용하는 것을 가정하고 만들어진 파이썬 파일
- 패키지
  - 여러 모듈을 묶은 것
- 라이브러리
  - 여러 패키지를 묶은 것, 때로는 패키지를 가리켜 ‘라이브러리’라고 부르기도 함

\* <https://offbyone.tistory.com/106>

## 24단계: 복잡한 함수의 미분 (최적화 문제 - Benchmark 함수)

- Sphere 함수

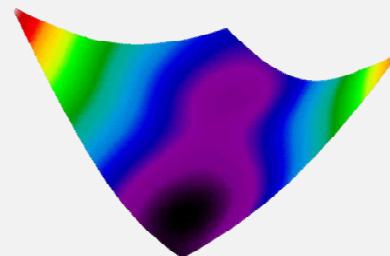
$$z = x^2 + y^2$$



Function 클래스에 정의된 모든 사칙연산에  
대해 `z.Backward()` 하나로 역전파 해결

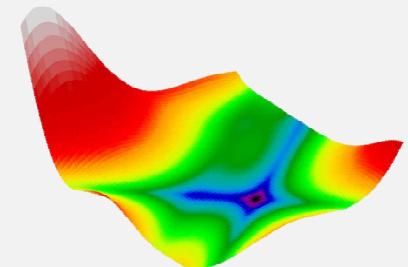
- Matyas 함수

$$z = 0.26 * (x^2 + y^2) - 0.48 * x * y$$



- Goldstein-Price 함수

$$\begin{aligned} z = & (1 + (x + y + 1)^2 * (19 - 14x + 3x^2 - 14y + 6xy + 3y^2)) \\ & * (30 + (2x - 3y)^2 * (18 - 32x + 12x^2 + 48y - 36xy + 27y^2)) \end{aligned}$$



## 제 3고지 고차 미분 계산

## 25단계 - 계산 그래프 시각화(1)

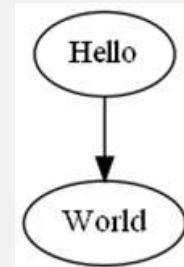
- Graphviz 라이브러리

- <http://graphviz.gitlab.io/download/>

- : 설치 시, 환경변수 설정 필요

- 설치 후, dot 명령어 실행 여부 확인 (\*Windows Git Bash)

```
$ dot -V  
dot - graphviz version 2.46.1 (20210213.1702)  
$ echo "digraph G {Hello->World}" | dot -Tpng > hello.png
```



- 그림 테스트 방법

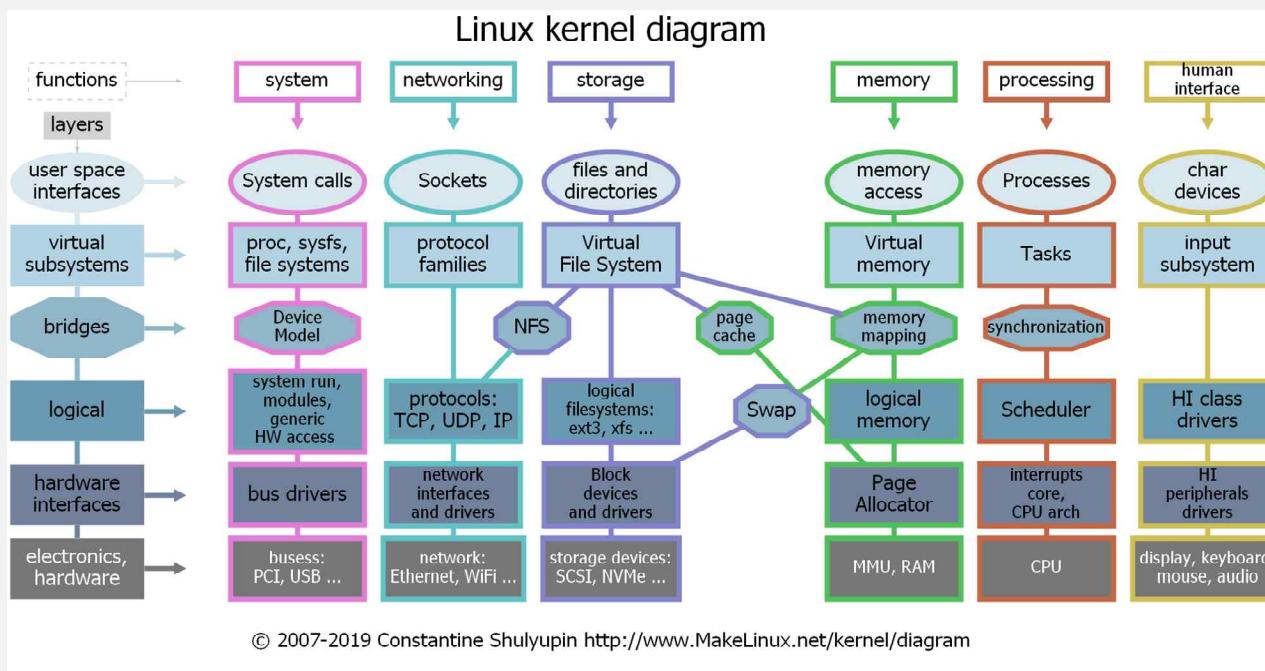
- : <http://graphviz.gitlab.io/gallery/> 방문하여 특정 그림 선택 및 노드 복사 (Ctrl + C)

- : 메모장에 붙여 넣기 (Ctrl + V) 후 위 명령어 입력 (\$ dot file.dot - T png - o file.png)

## 25단계 - 계산 그래프 시각화(1)

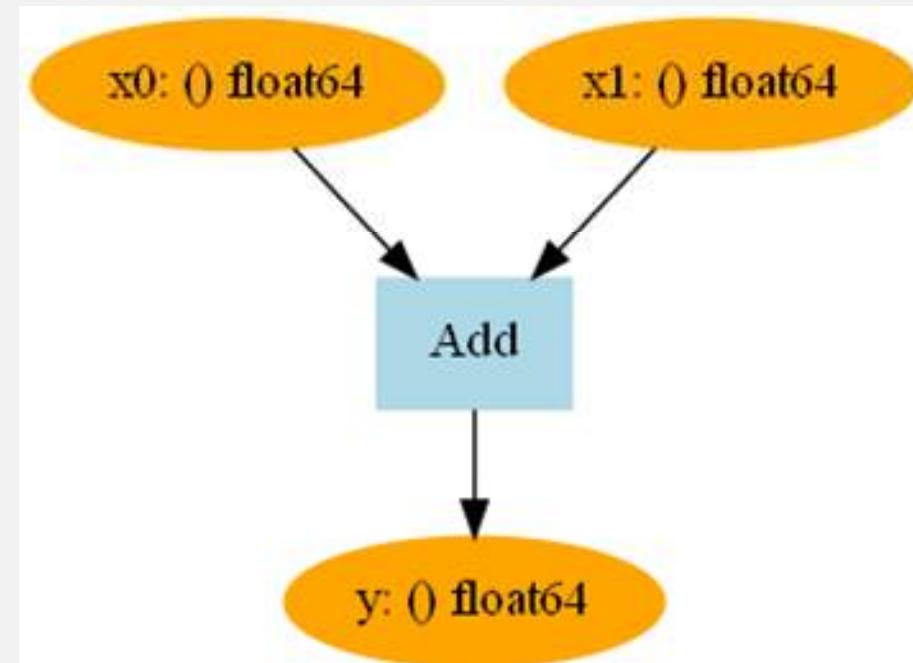
- Linux Kernel Diagram

- [http://graphviz.gitlab.io/Gallery/directed/Linux\\_kernel\\_diagram.html](http://graphviz.gitlab.io/Gallery/directed/Linux_kernel_diagram.html)



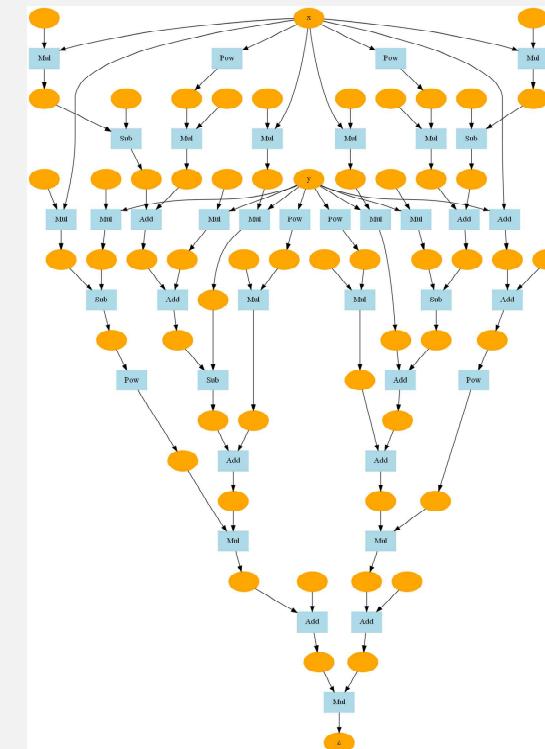
## 26단계 - 계산 그래프 시각화(2)

- 시각화 코드 예제 (p. 216)
  - get\_dot\_graph 메인 함수 구현
    - : 터미널에서 dot 명령어로 수정 변환 필요
  - \_dot\_var & \_dot\_func 보조 함수 구현
    - : 계산 그래프에서 DOT 언어로 변환
  - plot\_dot\_graph 함수 구현
    - : 이미지 변환까지 한번에 구현함
    - : dot 명령어를 함수 안에서 직접 구현



## 26단계 - 계산 그래프 시각화(2)

- 시각화 코드 예제 (p. 216)
  - `get_dot_graph` 메인 함수 구현
    - : 터미널에서 `dot` 명령어로 수정 변환 필요
  - `_dot_var` & `_dot_func` 보조 함수 구현
    - : 계산 그래프에서 DOT 언어로 변환
  - `plot_dot_graph` 함수 구현
    - : 이미지 변환까지 한번에 구현함
    - : `dot` 명령어를 함수 안에서 직접 구현



## 27단계 - 테일러 급수 Taylor Series 미분 & & steps/steps27.py

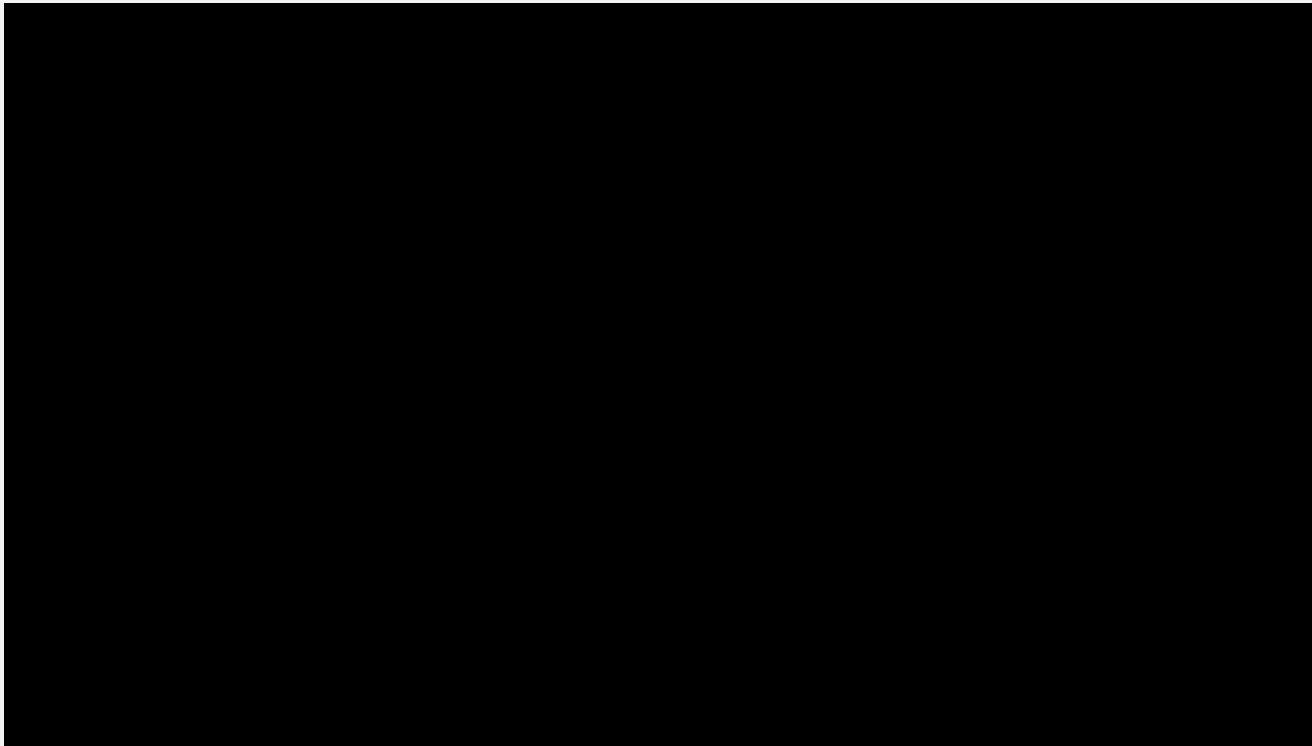
- $y = \sin(x)$  일 때, 미분은  $\frac{\partial y}{\partial x} = \cos(x)$

: Sin 클래스와 sin() 함수 구현

:  $\sin\left(\frac{\pi}{4}\right) = \cos\left(\frac{1}{\sqrt{2}}\right) = 0.7071067811865476$

## 27단계 - 테일러 급수 Taylor Series 미분 & & steps/steps27.py

- 테일러 급수에 관한 설명은 Page 227 참조



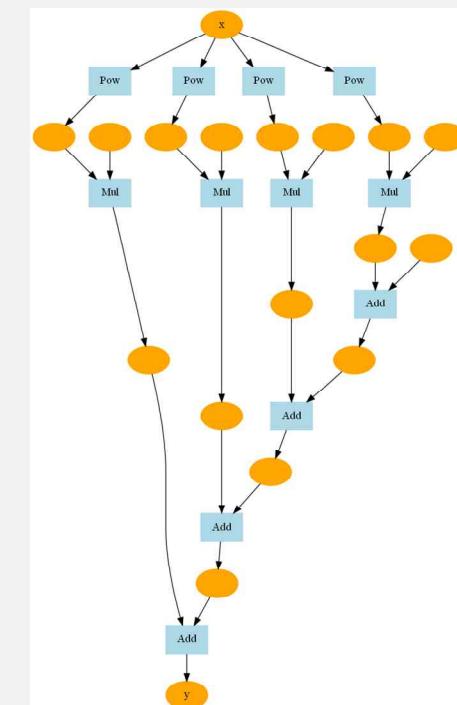
## 27단계 - 테일러 급수 Taylor Series 미분 & & steps/steps27.py

- my\_sin 함수 구현 및 풀이 & 그래프 구현 (threshold = 0.001)

```
Import math

def my_sin(x, threshold = 0.0001):
    y = 0
    for i in range(100000):
        c = (-1) ** i / math.factorial(2 * i + 1)
        t = c * x ** (2 * i + 1)
        y = y + t
        if abs(t.data) < threshold:
            break

    return y
```



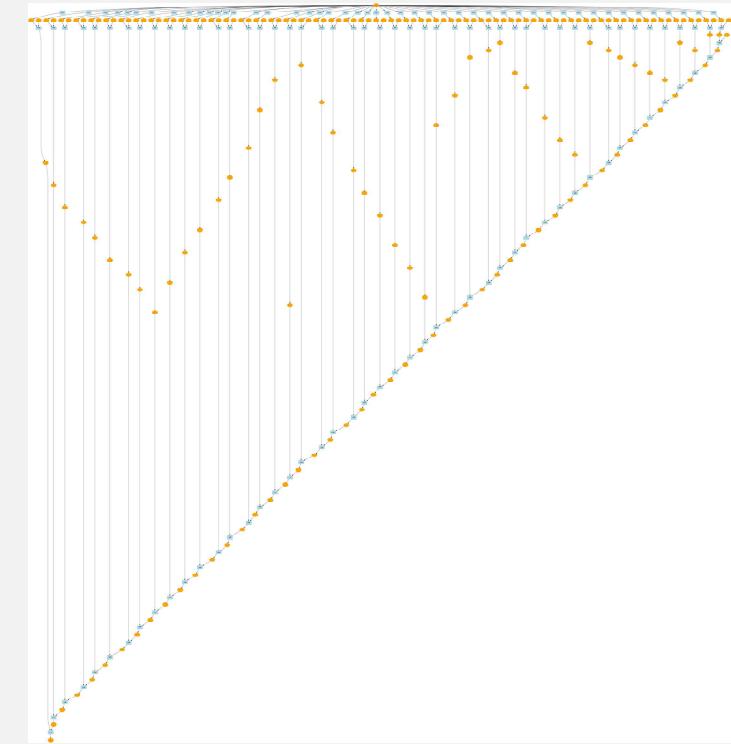
## 27단계 - 테일러 급수 Taylor Series 미분 & & steps/steps27.py

- my\_sin 함수 구현 및 풀이 & 그래프 구현 (threshold = 0.001)

```
Import math

def my_sin(x, threshold = 1e-150):
    y = 0
    for i in range(100000):
        c = (-1) ** i / math.factorial(2 * i + 1)
        t = c * x ** (2 * i + 1)
        y = y + t
        if abs(t.data) < threshold:
            break

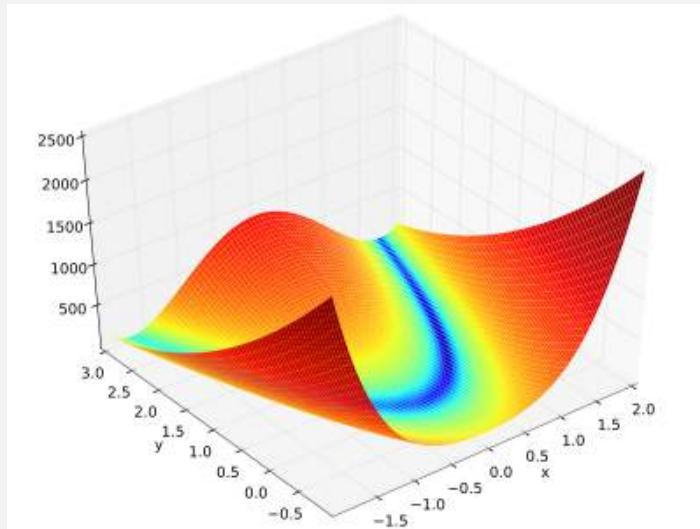
    return y
```



## 28단계 - 로젠브록 함수 Rosenbrock Function & & steps/steps28.py

- 함수의 공식은 다음과 같음

- $f(x_0, x_1) = b(x_1 - x_0^2)^2 + (a - x_0)^2$
- a = 1, b = 100일 때,  $y = 100(x_1 - x_0^2)^2 + (1 - x_0)^2$



\* 이미지 출처: [https://upload.wikimedia.org/wikipedia/commons/thumb/3/32/Rosenbrock\\_function.svg/400px-Rosenbrock\\_function.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/3/32/Rosenbrock_function.svg/400px-Rosenbrock_function.svg.png)

## 28단계 - 로젠브록 함수 Rosenbrock Function & & steps/steps28.py

- 파이썬 함수 구현

- $f(x_0, x_1) = b(x_1 - x_0^2)^2 + (a - x_0)^2$
- a = 1, b = 100일 때,  $y = 100(x_1 - x_0^2)^2 + (1 - x_0)^2$

```
import math
from dezzero import Variable

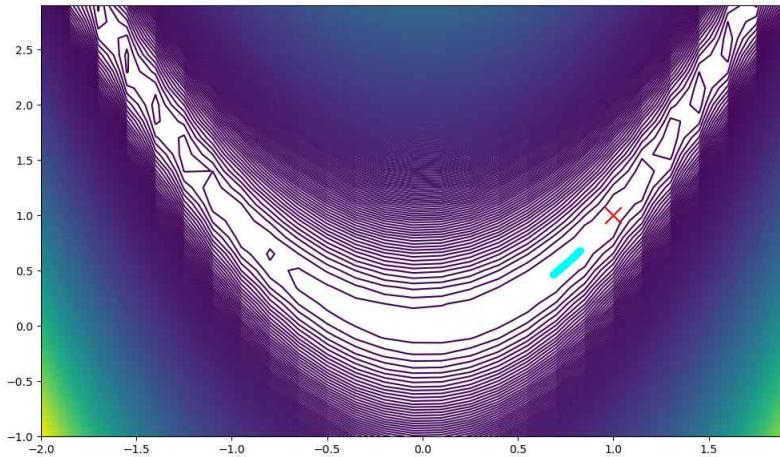
def rosenbrock(x0, x1):
    y = 100 * (x1 - x0 ** 2) ** 2 + (1 - x0) ** 2
    return y
```

## 28단계 - 로젠브록 함수 Rosenbrock Function & & steps/steps28.py

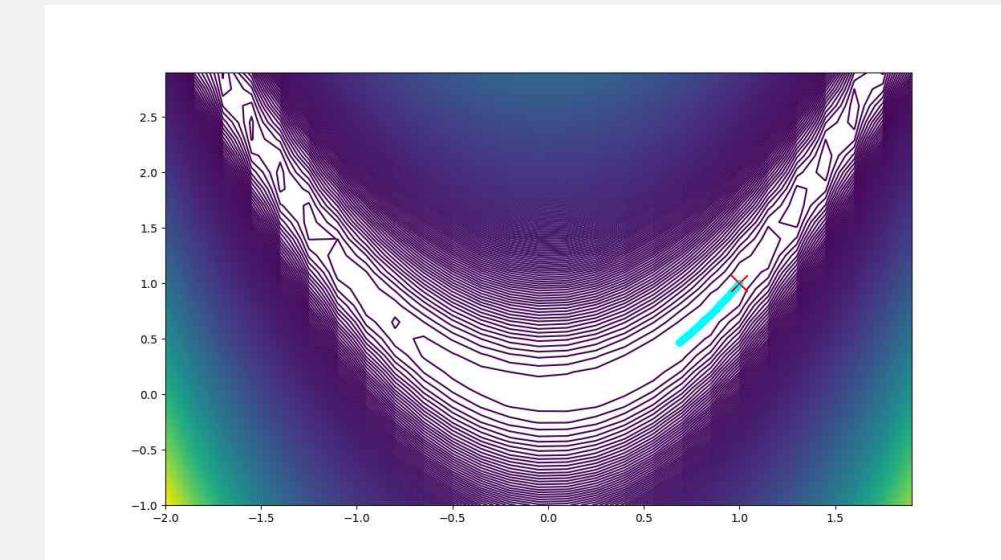
- 경사하강법 Gradient Descent 구현
  - 최솟값 찾기
  - 기울기 방향으로 거리만큼 이동하여 다시 기울기를 구하는 작업을 반복
  - 알맞은 지점(좋은 초기값)에서 시작하면 경사 하강법은 우리를 목적지까지 효율적으로 안내

## 28단계 - 로젠브록 함수 Rosenbrock Function & & steps/steps28.py

- 경사하강법 Gradient Descent 구현



```
< lr = 0.001, iters = 1000 >
```

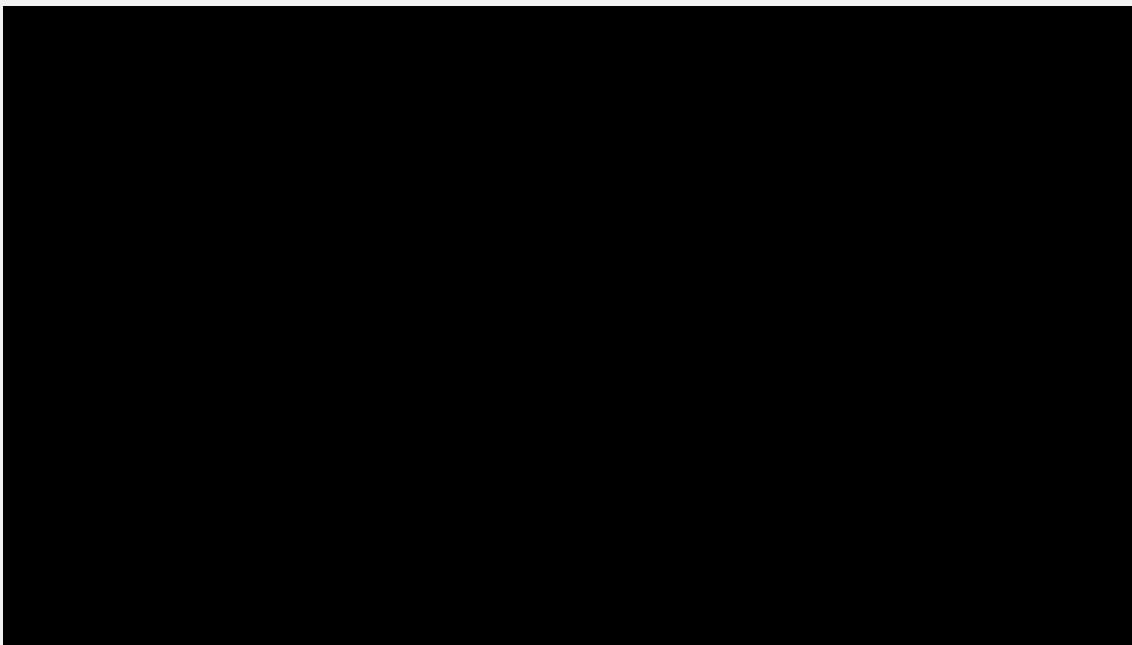


```
< lr = 0.001, iters = 50000 >
```

\* 이미지 출처: [https://upload.wikimedia.org/wikipedia/commons/thumb/3/32/Rosenbrock\\_function.svg/400px-Rosenbrock\\_function.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/3/32/Rosenbrock_function.svg/400px-Rosenbrock_function.svg.png)

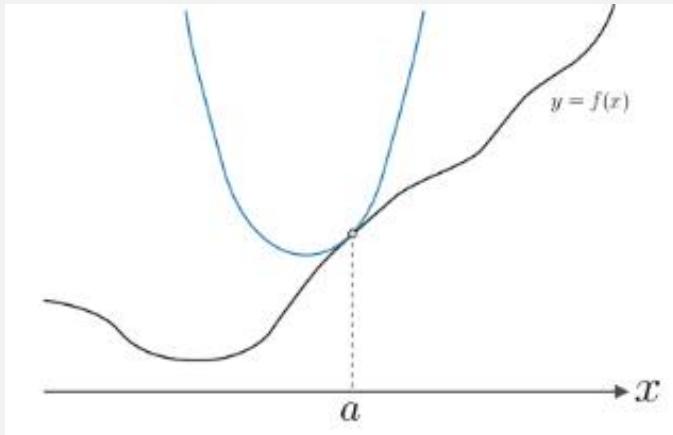
## 29단계 - 뉴턴 방법(수동 계산) Newton's method & & steps/steps29.py

- $y = f(x)$ 의 최솟값을 구하는 예제
  - 테일러 급수에 따라  $y = f(x)$ 식을 2차 테일러 급수로 근사 후 최소값 구하기



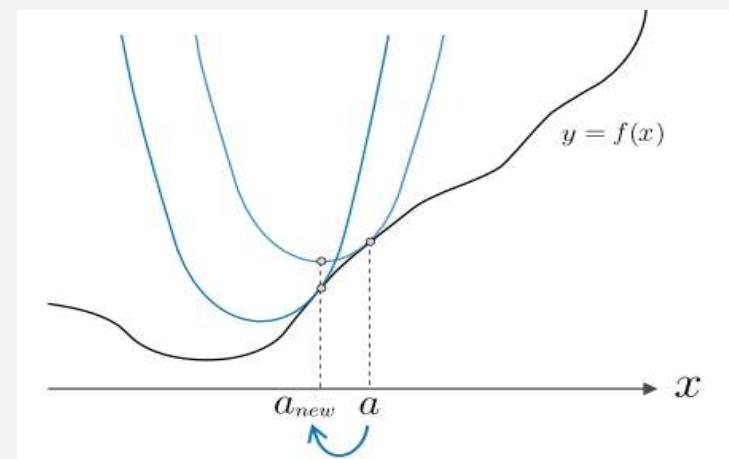
## 29단계 - 뉴턴 방법(수동 계산) Newton's method & & steps/steps29.py

- $y = f(x)$ 의 최솟값을 구하는 예제
  - 테일러 급수에 따라  $y = f(x)$ 식을 2차 테일러 급수로 근사 후 최소값 구하기



경사하강법

$$x \leftarrow x - \alpha f'(x)$$

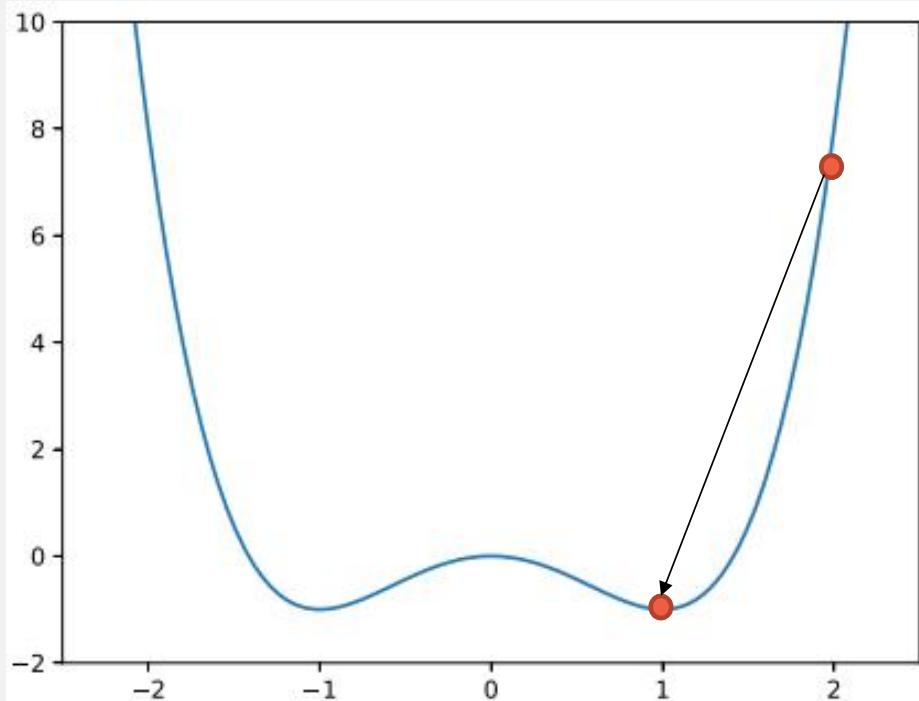


뉴턴 방법

$$x \leftarrow x - \frac{f'(x)}{f''(x)}$$

## 29단계 - 뉴턴 방법(수동 계산) Newton's method & & steps/steps29.py

- (예시) 함수  $y = x^4 - 2x^2$



```
Import numpy as np
From dezero import Variable

def f(x):
    y = x ** 4 - 2 * x ** 2
    return y

def gx2(x): # 2차 미분을 자동으로 구현하지 못하기 때문에 수동으로 입력
    return 12 * x ** 2 - 4

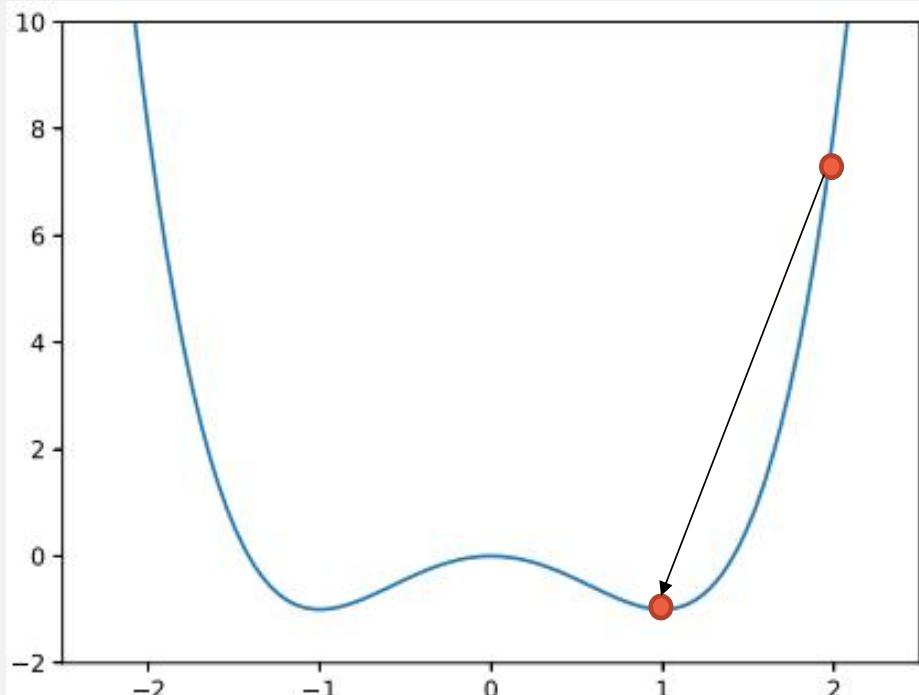
X = Variable(np.array(2.0))

for i in range(iters):
    print(i, x)
    y = f(x)
    x.cleargrad()
    y.backward()

    x.data -= x.grad / gx2(x.data)
```

## 29단계 - 뉴턴 방법(수동 계산) Newton's method & & steps/steps29.py

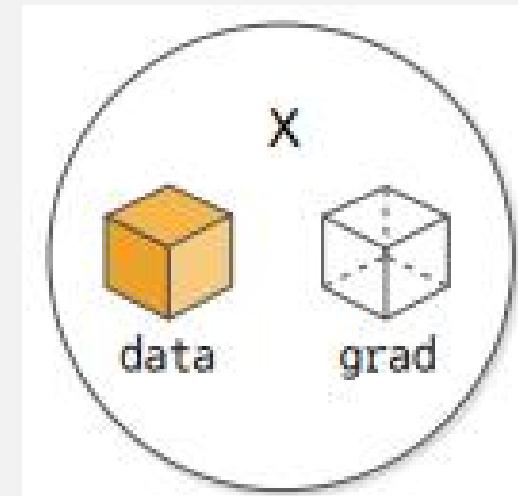
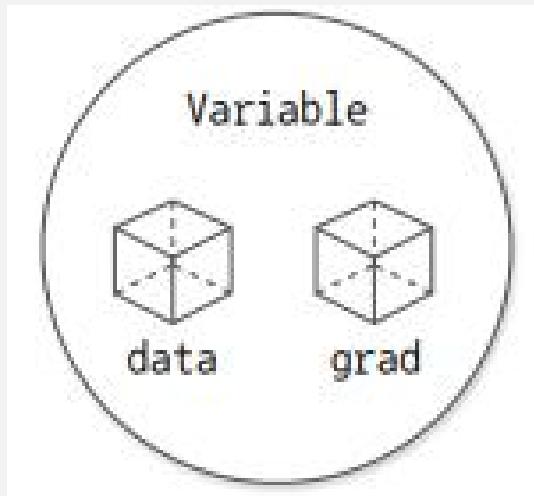
- 초깃값 2에서 1에 도달하는 횟수 (총 7회만에 가능 / 경사하강법은 124회, 245 p)



```
0 variable(2.0)
1 variable(1.4545454545454546)
2 variable(1.1510467893775467)
3 variable(1.0253259289766978)
4 variable(1.0009084519430513)
5 variable(1.0000012353089454)
6 variable(1.000000000002289)
7 variable(1.0)
8 variable(1.0)
9 variable(1.0)
```

## 30단계 - 고차 미분(준비편), page 247 ~ 249

- 현재 DeZero 1차 미분 한정 → 확장 DeZero 모든 형태의 고차 미분 자동 계산
  - Variable 인스턴스 변수 확인



## 30단계 - 고차 미분(준비편), page 249 ~ 251

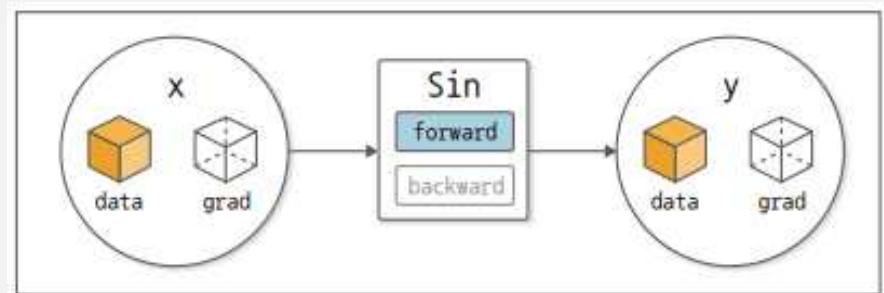
- 현재 DeZero 1차 미분 한정 → 확장 DeZero 모든 형태의 고차 미분 자동 계산

( 현재 역전파 구현 )

- Variable 인스턴스 변수 확인
- Function 클래스 확인 (\*연결)

: 미분값을 역방향으로 흘려 보내기 위함

$y = \sin(x)$ 의 계산 그래프 (순전파)



## 30단계 - 고차 미분(준비편), page 249 ~ 251

- 현재 DeZero 1차 미분 한정 → 확장 DeZero 모든 형태의 고차 미분 자동 계산

( 현재 역전파 구현 )

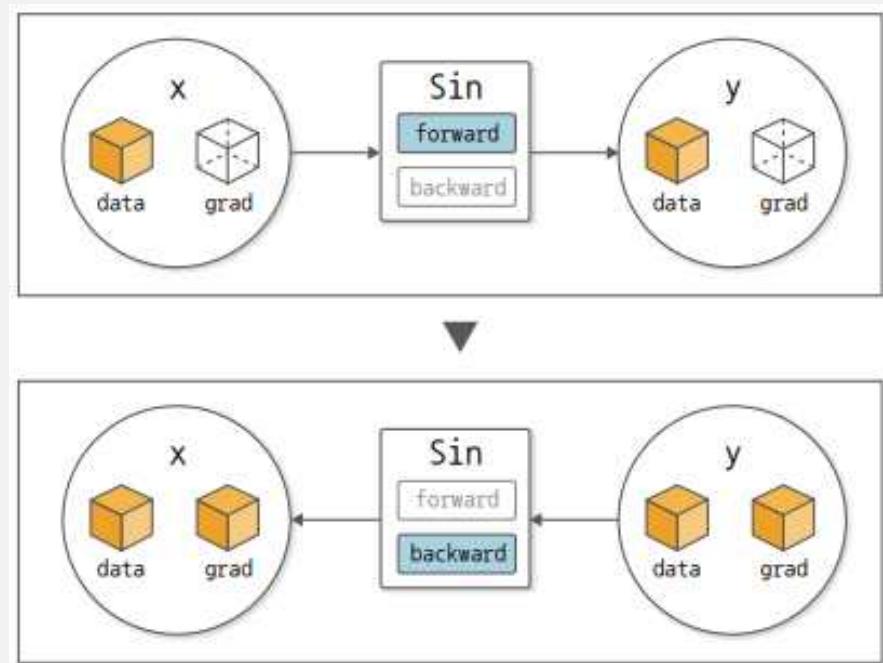
- Variable 인스턴스 변수 확인

- Function 클래스 확인 (\*연결)

: 미분값을 역방향으로 흘려 보내기 위함

- Variable 클래스의 역전파

$y = \sin(x)$ 의 계산 그래프 (순전파와 역전파)

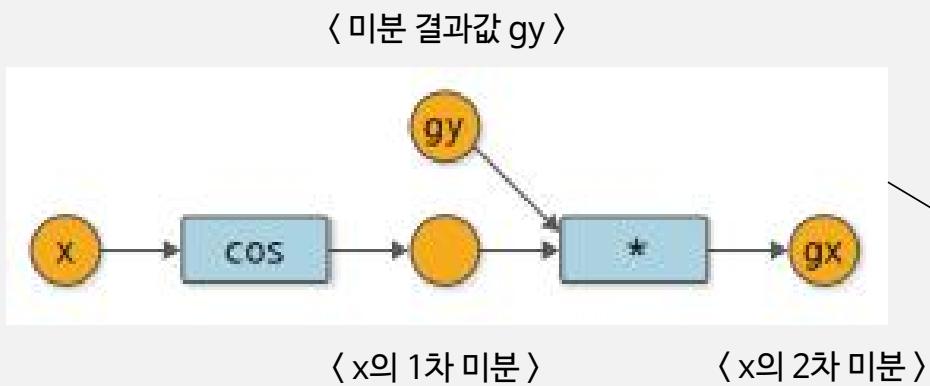


## 31단계 - 고차 미분(이론)

- 30단계 시점까지의 구현 요점 및 주요 문제
  - 계산의 연결은 Function 클래스의 `_call_` 메서드에서 만들어짐
  - 구체적인 순전파와 역전파 계산은 Function 클래스를 상속한 forward & backward
- 계산 그래프의 연결은 “순전파 ” 에서만 생성 / 역잔파에서는 미 생성이 문제의 핵심

## 31단계 - 고차 미분(이론)

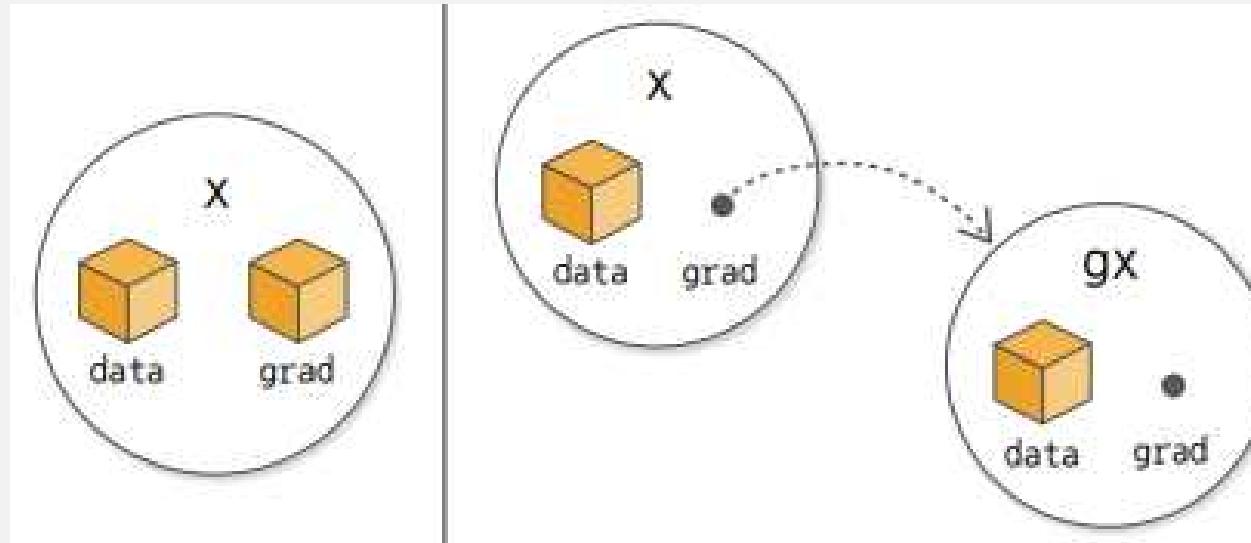
- Sin 함수의 미분을 구하기 위한 계산 그래프 (x의 2차 미분)



```
class Sin(Function):  
    ...  
  
    def backward(self, gy):  
        x, = self.inputs  
        gx = gy * cos(x)  
        return gx
```

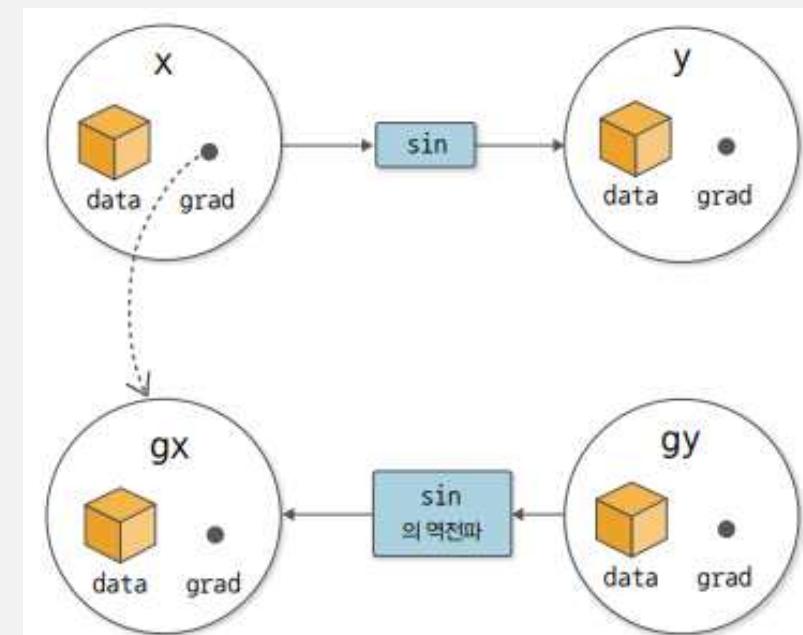
## 31단계 - 고차 미분(이론)

- 역전파로 계산 그래프 만들기 과정
  - 미분값(기울기)을 Variable 인스턴스 형태로 유지
  - (오른쪽) 새로운 Variable 클래스 변형 필요



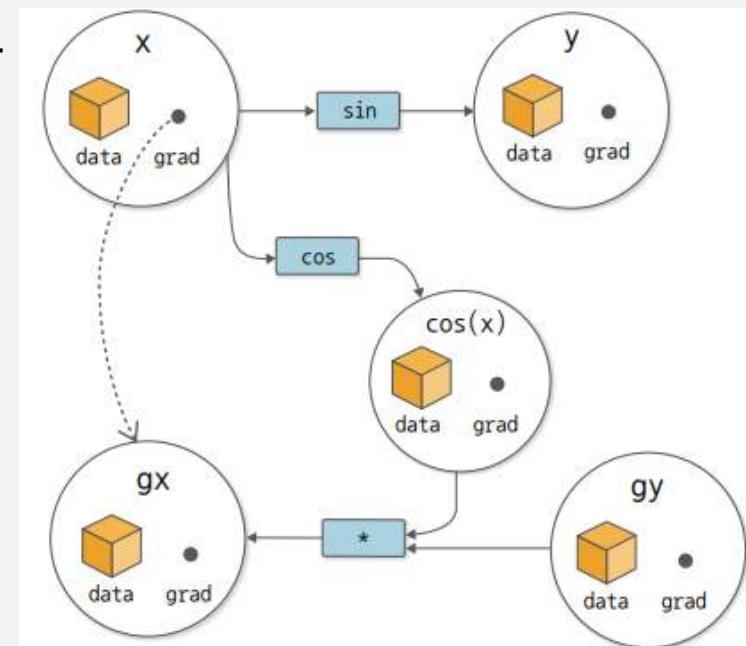
## 31단계 - 고차 미분(이론)

- 역전파로 계산 그래프 만들기 과정
  - Sin 클래스의 순전파와 역전파를 수행한 후의 계산 그래프
  - 역전파 계산에 대한 계산 그래프도 만들어짐



## 31단계 - 고차 미분(이론)

- 역전파로 계산 그래프 만들기 과정
  - `y.backward()`를 호출함으로써 “새로” 만들어지는 계산 그래프
  - `gx.backward()`를 호출함으로써  $y$ 의  $x$ 에 대한 2차 미분



## 32단계 - 고차 미분(구현) && dezero/core.py

- Variable 클래스 grad 인스턴스 변수 변환
  - ndarray → Variable

```
class Variable:  
    ...  
  
    def backward(self, retain_grad=False):  
        if self.grad is None:  
            # self.grad = np.ones_like(self.data)  
            self.grad = Variable(np.ones_like(self.data))  
  
        ...
```

## 32단계 - 고차 미분(구현) && dezero/core.py

- Variable 클래스 grad 인스턴스 변수 변환
  - ndarray → Variable
- 함수 클래스의 역전파
  - Add 클래스 수정 불 필요
  - Mul, Sub, Div, Pow 클래스 모두 수정
  - : tuple 형태로 저장

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

수정 전

```
class Mul(Function):
    ...
    def backward(self, gy):
        x0, x1 = self.inputs
        return gy * x1, gy * x0
```

수정 후

## 32단계 - 고차 미분(구현) && dezero/core.py & \_\_init\_\_.py

- (core.py) 역전파를 더 효율적으로 변경 (p.264 페이지 코드 참조)
  - create\_graph=False 설정
  - with using\_config(…)에서 역전파 처리 수행
- \_\_init\_\_.py 변경 (p.265 페이지 코드 참조)
  - dezzero/core\_simple.py에서 dezzero/core.py로 수정

## 33단계 - 뉴턴 방법으로 푸는 최적화(자동계산) & steps/steps33.py

- 2차 미분 계산하기

```
import numpy as np
from dezzero import Variable

def f(x):
    y = x ** 4 - 2 * x ** 2
    return y
```

# 첫번째 역전파 구현

# 두번째 역전파 구현

2 )&gt;&gt;&gt; variable(2.0)

첫번째 역전파

24 )&gt;&gt;&gt; variable(24.0)

두번째 역전파

68 )&gt;&gt;&gt; variable(68.0)

\* 손계산 식 결괏값과 **상이**

$$f(x) = x^4 - 2x^2$$

$$f'(2.0) = 4x^3 - 4x \rightarrow 24$$

$$f''(2.0) = 12x^2 - 4 \rightarrow 48$$

\* 새로운 계산 전, 미분값 재 설정 필요

## 33단계 - 뉴턴 방법으로 푸는 최적화(자동계산) & steps/steps33.py

- Variable 미분 값이 남아있는 상태로 새로운 역전파를 수행하는 문제

```
import numpy as np
from dezzero import Variable

def f(x):
    y = x ** 4 - 2 * x ** 2
    return y
```

# 첫번째 역전파 구현

# 두번째 역전파 구현

2 )&gt;&gt;&gt; variable(2.0)

첫번째 역전파

24 )&gt;&gt;&gt; variable(24.0)

두번째 역전파

48 )&gt;&gt;&gt; variable(48.0)

1차 미분

2차 미분

\* 손계산 식 결괏값과 일치

$$f(x) = x^4 - 2x^2$$

$$f'(2.0) = 4x^3 - 4x \rightarrow 24$$

$$f''(2.0) = 12x^2 - 4 \rightarrow 48$$

\* 두개의 값 일치 확인

## 33단계 - 뉴턴 방법으로 푸는 최적화(자동계산) & steps/steps33.py

- 뉴턴 방법을 활용한 최적화 (수식)

$$x \leftarrow x - \frac{f'(x)}{f''(x)}$$

- 1차 미분과 2차 미분을 사용하여 x의 값 갱신

```
0 variable(2.0)
1 variable(1.45454545454546)
2 variable(1.1510467893775467)
3 variable(1.0253259289766978)
4 variable(1.0009084519430513)
5 variable(1.0000012353089454)
6 variable(1.000000000002289)
7 variable(1.0000000000000002)
8 variable(1.0)
9 variable(1.0)
```

```
import numpy as np
from dezzero import Variable

def f(x):
    y = x ** 4 - 2 * x ** 2
    return y

x = Variable(np.array(2.0))
iters = 10

for i in range(iters):
    print(i, x)
    y = f(x)
    x.cleargrad()
    y.backward(create_graph=True)

    gx = x.grad
    x.cleargrad()
    gx.backward()
    gx2 = x.grad
```

$$x \leftarrow x - \frac{f'(x)}{f''(x)}$$

## 34단계 - sin 함수 고차 미분 & dezzero/functions.py

- Sin 함수 구현

- $y = \sin(x)$  일 때,  $\frac{\partial y}{\partial x} = \cos(x)$
- Sin 클래스 구현 시, Cos 클래스와 Cos 함수도 필요
- $gy * \cos(x)$ : 곱셈 연산자 Mul 함수 호출

```
import numpy as np
from dezzero.core import Function

class Sin(Function):
    def forward(self, x):
        y = np.sin(x)

        return y
    def backward(self, gy):
        x, = self.inputs
        gx = gy * cos(x)

        return gx

def sin(x):
    return Sin()(x)
```

## 34단계 - sin 함수 고차 미분 & dezzero/functions.py

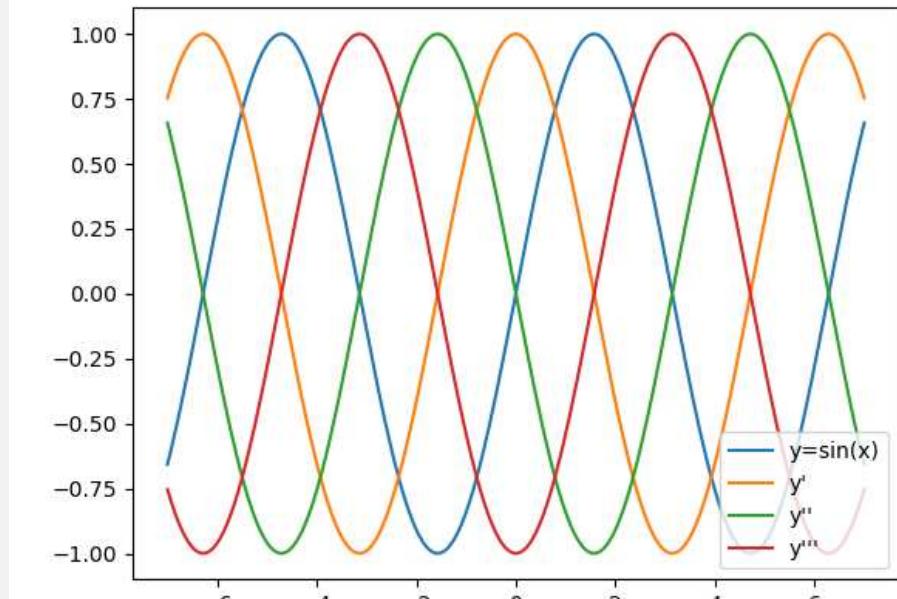
- Cos 함수 구현

- $y = \cos(x)$  일 때,  $\frac{\partial y}{\partial x} = -\sin(x)$
- 구체적인 계산은 sin() 함수 사용

```
class Cos(Function):  
    def forward(self, x):  
        y = np.cos(x)  
        return y  
  
    def backward(self, gy):  
        x, = self.inputs  
        gx = gy * -sin(x)  
  
        return gx  
  
def cos(x):  
    return Cos()(x)
```

## 34단계 - sin 함수 고차 미분 & steps/step34.py

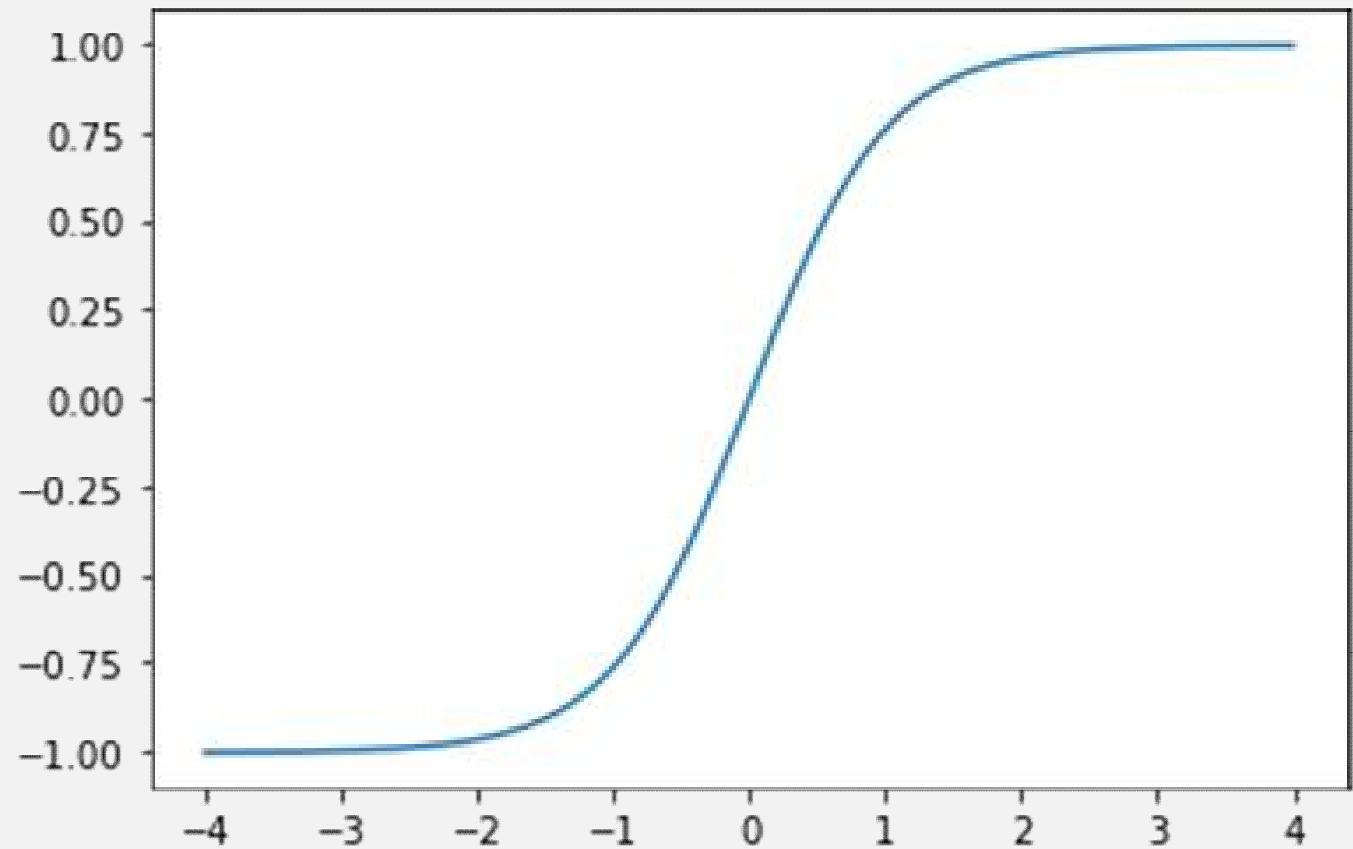
- Sin 함수 고차 미분
  - 코드는 p. 274 참조
  - np.linspace(-7, 7, 200)  
: 200 등분한 배열
  - $y'$  는 1차 미분,  $y''$ 는 2차 미분,  $y'''$ 는 3차 미분
  - 1차 미분, 2차 미분, 3차 미분, ...  
 $y = \sin(x) \rightarrow y = \cos(x) \rightarrow y = -\sin(x) \rightarrow y = -\cos(x) \dots$  형태로 진행



## 35단계 - 고차 미분 계산 그래프 & dezzero/functions.py

- tanh 함수 추가

- $y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$



## 35단계 - 고차 미분 계산 그래프 & dezzero/functions.py

- tanh 함수 추가

-  $y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

- tanh 함수 미분 (p. 278 참조)

-  $y = \tanh(x)$  일 때,  $\frac{\partial \tanh(x)}{\partial x} = 1 - y^2$

```
class Tanh(Function):
    def forward(self, x):
        y = np.tanh(x)

        return y

    def backward(self, gy):
        y = self.outputs[0]()
        gx = gx * (1 - y * y)

        return gx

    def tanh(x):
        return Tanh()(x)
```

## 35단계 - 고차 미분 계산 그래프 & dezzero/step35.py

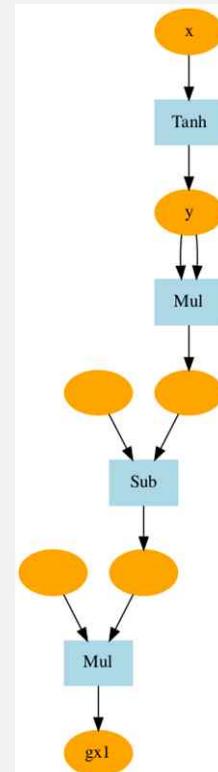
- tanh 함수 추가

-  $y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

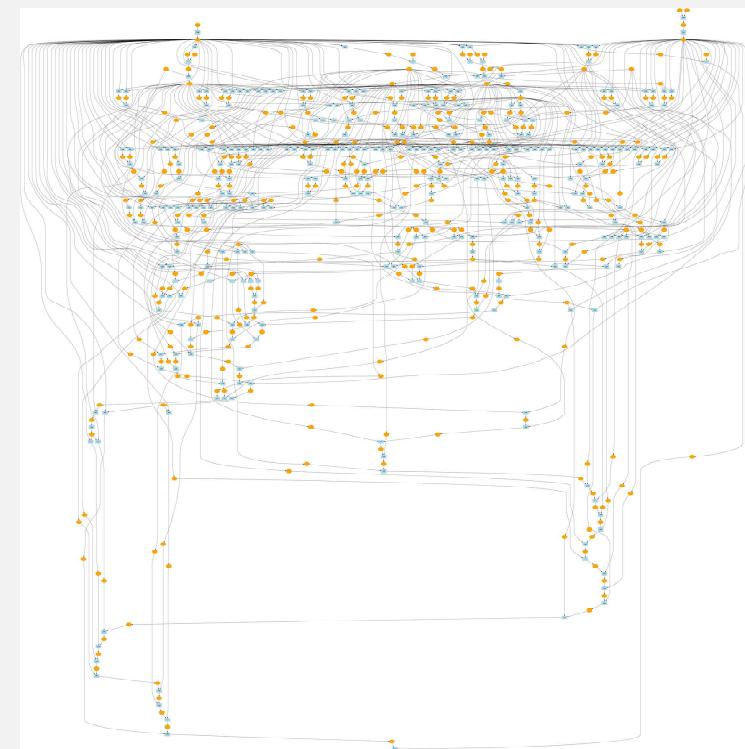
- tanh 함수 미분 (p. 278 참조)

-  $y = \tanh(x)$  일 때,  $\frac{\partial \tanh(x)}{\partial x} = 1 - y^2$

- 계산 그래프 그리기



1차 미분



5차 미분

## 36단계 - 고차 미분 이외의 용도 & dezzero/step36.py

- double backprop의 용도
  - 역전파로 수행한 계산에 대해 또 다시 역전파 하는 것 (현대적인 프레임워크는 대부분 지원함)
- 딥러닝 연구에서의 사용 예
  - WGAN-GP 최적화하는 함수

기울기

$$L = \underbrace{\mathbb{E}_{\tilde{x} \sim \mathbb{P}_g} [D(\tilde{x})] - \mathbb{E}_{x \sim \mathbb{P}_r} [D(x)]}_{\text{Original critic loss}} + \underbrace{\lambda \mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{g}}} [\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1]^2}_{\text{Our gradient penalty}}.$$

- Finn, C., Abbeel, P., & Levine, S. (2017). Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. *ICML*.
- Schulman, J., Levine, S., Abbeel, P., Jordan, M.I., & Moritz, P. (2015). Trust Region Policy Optimization. *ArXiv*, *abs/1502.05477*.

## 제 4고지 신경망 만들기

## 37단계 - 텐서를 다루다 & dezzero/step37.py

- 원소별 계산

- 머신러닝 데이터로는 벡터나 행렬 등의 “텐서”가 주로 쓰임
- 텐서 사용 시 주의 점 확인하면서 DeZero 확장 준비 및 문제 없이 텐서 다루는 것 목적

〈 Sin 함수 구현 스칼라 방식 〉

```
import numpy as np
import dezzero.functions as F
from dezzero import Variable

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

variable(0.8414709848078965)

〈 Sin 함수 구현 텐서 방식 〉

```
import numpy as np
import dezzero.functions as F
from dezzero import Variable

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

variable([[ 0.841 0.909 0.141]
 [-0.756 -0.958 -0.279 ]])

## 37단계 - 텐서를 다루다 & dezzero/step37.py

- 원소별 계산
  - 머신러닝 데이터로는 벡터나 행렬 등의 “텐서”가 주로 쓰임
  - 텐서 사용 시 주의 점 확인하면서 DeZero 확장 준비 및 문제 없이 텐서 다루는 것 목적
  - 덧셈에서도 각 원소별 더한 값이며 출력의 형상도 입력 형상과 같음

```
import numpy as np
import dezzero.functions as F
from dezzero import Variable

x = Variable(np.array([[1, 2, 3], [4, 5, 6]]))
c = Variable(np.array([[10, 20, 30], [40, 50, 60]]))
y = x + c
print(y)
```

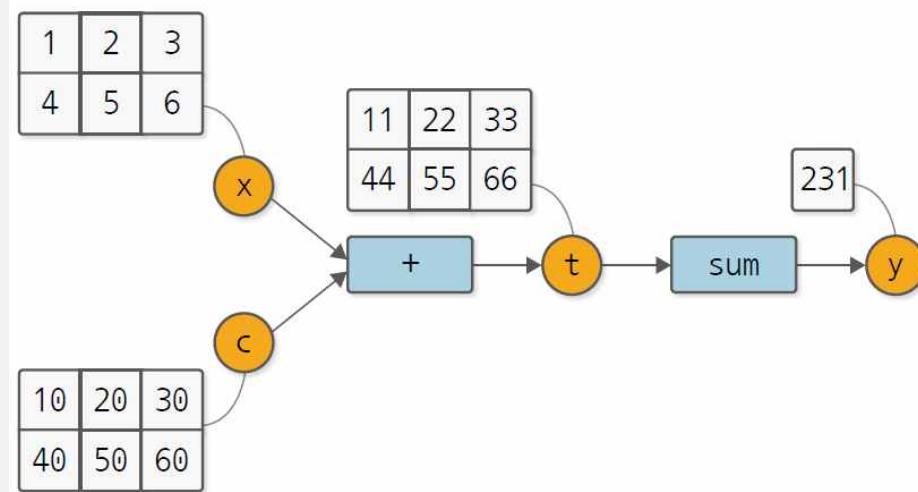
```
variable([[11 22 33]
          [44 55 66]])
```

## 37단계 - 텐서를 다루다 & dezzero/step37.py

- 원소별 계산

- 머신러닝 데이터로는 벡터나 행렬 등의 “텐서”가 주로 쓰임
- 텐서 사용 시 주의 점 확인하면서 DeZero 확장 준비 및 문제 없이 텐서 다루는 것 목적
- 덧셈에서도 각 원소별 더한 값이며 출력의 형상도 입력 형상과 같음

그림 37-1 텐서를 사용한 계산 그래프



## 37단계 - 텐서를 다룬다 & dezro/step37.py

- 원소별 계산

...

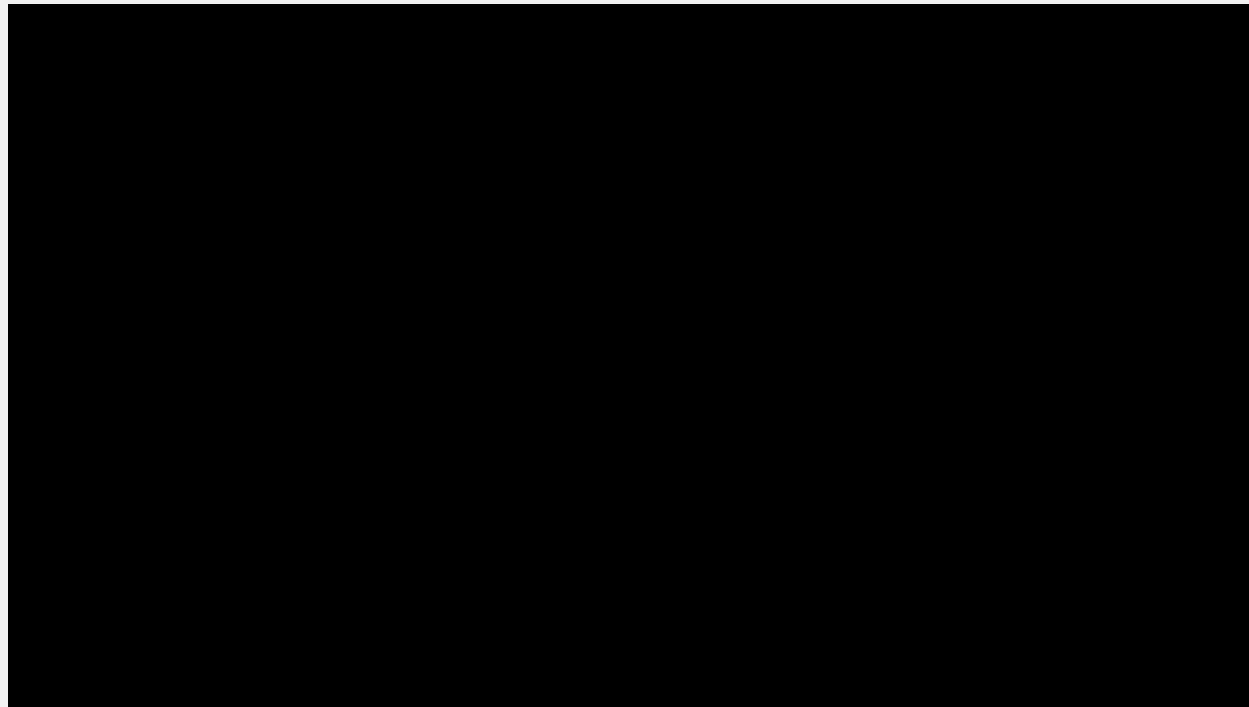
- 마지막 출력값이 스칼라인 계산 그래프에 대해 역전파를 구현한다.

```
y.backward(retain_grad=True)
print(y.grad)
print(x.grad)
print(c.grad)
print(t.grad)
```

```
variable(1)
variable([[1 1 1]
          [1 1 1]])
variable([[1 1 1]
          [1 1 1]])
variable([[1 1 1]
          [1 1 1]])
```

## 37단계 - 텐서를 다룬다 & dezzero/step37.py

- 야코비 행렬: 벡터에 대한 벡터의 미분
  - 야코비 행렬이란?



## 37단계 - 텐서를 다루다 & dezzero/step37.py

- 야코비 행렬: 벡터에 대한 벡터의 미분
  - 행렬의 곱을 계산하는 순서 방법 2가지, 입력에서 출력쪽으로 계산
    - : 자동 미분의 forward 모드 / 행렬 곱의 결과가 다시 행렬이 됨
    - : 결과:  $N \times N$  행렬

**그림 37-2** 입력 쪽에서 출력 쪽으로 괄호를 친다(forward 모드).

$$\frac{\partial y}{\partial \mathbf{x}} = \left( \frac{\partial y}{\partial \mathbf{b}} \underbrace{\left( \frac{\partial \mathbf{b}}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{x}} \right)}_{\text{forward mode}} \right)$$

$$\frac{\partial \mathbf{b}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial b_1}{\partial x_1} & \frac{\partial b_1}{\partial x_2} & \cdots & \frac{\partial b_1}{\partial x_n} \\ \frac{\partial b_2}{\partial x_1} & \frac{\partial b_2}{\partial x_2} & \cdots & \frac{\partial b_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial b_n}{\partial x_1} & \frac{\partial b_n}{\partial x_2} & \cdots & \frac{\partial b_n}{\partial x_n} \end{pmatrix}$$

## 37단계 - 텐서를 다루다 & dezzero/step37.py

- 야코비 행렬: 벡터에 대한 벡터의 미분
  - 행렬의 곱을 계산하는 순서 방법 2가지, 출력 쪽에서 입력 쪽으로 계산 (역전파)
    - : 자동 미분의 reverse 모드
    - : y가 스칼라이므로 중간의 행렬 곱의 결과는 모두 벡터 (행 벡터)

**그림 37-3** 출력 쪽에서 입력 쪽으로 괄호를 친다(reverse 모드).

$$\frac{\partial y}{\partial \mathbf{x}} = \left( \underbrace{\left( \frac{\partial y}{\partial \mathbf{b}} \frac{\partial \mathbf{b}}{\partial \mathbf{a}} \right)}_{\text{reverse mode}} \frac{\partial \mathbf{a}}{\partial \mathbf{x}} \right)$$

$$\frac{\partial y}{\partial \mathbf{a}} = \begin{pmatrix} \frac{\partial y}{\partial a_1} & \frac{\partial y}{\partial a_2} & \cdots & \frac{\partial y}{\partial a_n} \end{pmatrix}$$

## 38단계 - 형상 변환 함수 & dezzero/step38.py

- `reshape` 함수 구현
  - `np.reshape(x, shape)` 형태로 쓰며, x를 shape 인수로 지정한 형상으로 변환

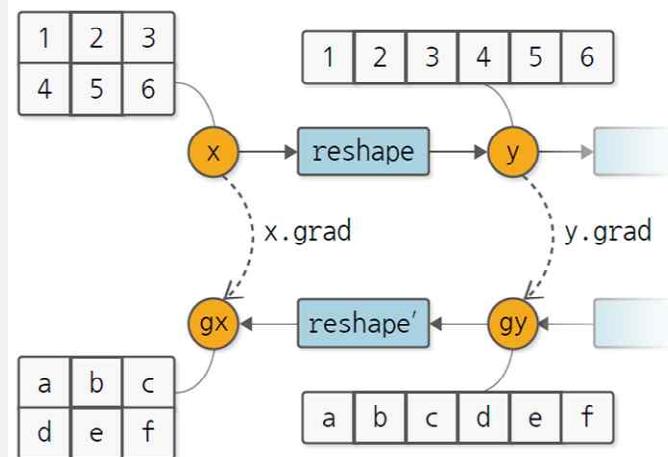
```
import numpy as np
x = np.array([[1,2,3], [4,5,6]])
y = np.reshape(x, (6, ))
print(y)
```

```
[1 2 3 4 5 6]
```

## 38단계 - 형상 변환 함수

- `reshape` 함수 구현
  - `np.reshape(x, shape)` 형태로 쓰며, `x`를 `shape` 인수로 지정한 형상으로 변환
  - `reshape` 함수는 단순히 형상만 변환함  
: 따라서, 역전파는 출력 쪽에서 전해지는 기울기에 아무런 손도 대지 않고 입력쪽으로 흘려 보내 줌

**그림 38-1** `reshape` 함수의 순전파와 역전파 계산 그래프(역전파 함수는 `reshape'`로 표시했으며 (a, b, c, d, e, f)라는 더미 기울기를 사용함)



## 38단계 - 형상 변환 함수 & dezzero/functions.py

- `reshape` 함수 구현
  - `np.reshape(x, shape)` 형태로 쓰며, `x`를 `shape` 인수로 지정한 형상으로 변환
  - `reshape` 함수는 단순히 형상만 변환함
    - : 따라서, 역전파는 출력 쪽에서 전해지는 기울기에 아무런 손도 대지 않고 입력쪽으로 흘려 보내 줌
  - `Reshape` 클래스 구현 (p. 309)
    - : `Reshape` 클래스 초기화 시 변형 목표 형상을 `shape` 인수로 받음,
    - : forward 메서드는 넘파이의 `reshape` 함수를 사용하여 형상 변환,
    - : 이때 `self.x_shape == x.shape` 코드에서 `x`의 형상을 기억 시킴으로서 backward 메서드에서 입력형상(`self.x_shape`)으로 변환 가능

## 38단계 - 형상 변환 함수 & dezzero/functions.py

- `reshape` 함수 구현

...

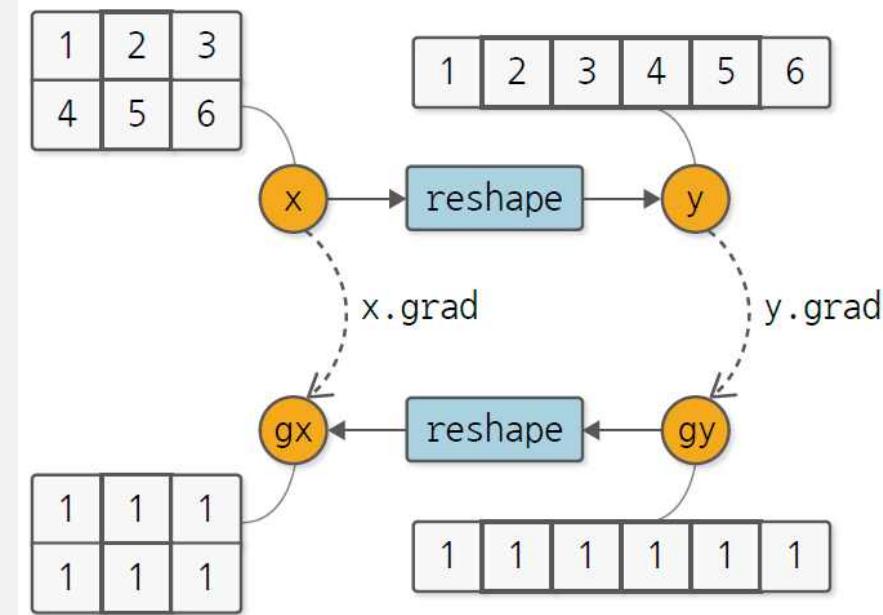
- `Reshape` 클래스 활용 (p. 310)

```
import numpy as np
from dezzero import Variable
import dezzero.functions as F

x = Variable(np.array([[1,2,3], [4,5,6]]))
y = F.reshape(x, (6,))
y.backward(retain_grad=True)
print(x.grad)
```

```
variable([[1 1 1]
          [1 1 1]])
```

그림 38-2 `reshape` 함수를 사용한 계산 예



## 38단계 - 형상 변환 함수 & dezzero/step38.py

- Variable에서 reshape 사용하기
  - reshape를 ndarray 인스턴스의 메서드로 사용함.

```
import numpy as np

x = np.random.rand(1,2,3)
y = x.reshape((2, 3)) # 튜플로 받기
y = x.reshape(([2,3])) # 리스트로 받기
y = x.reshape(2,3) # 인수를 그대로(풀어서) 받기
print(y)
```

```
[[0.07647932  0.00277102  0.98456558]
 [0.20031618  0.49031364  0.694241]]
```

## 38단계 - 형상 변환 함수 & dezzero/core.py

- Variable에서 reshape 사용하기
  - reshape를 ndarray 인스턴스의 메서드로 사용함.

```
import dezzero

class Variable:
    ...
    def reshape(self, *shape):
        if len(shape) == 1 and isinstance(shape[0], (tuple, list)):
            shape = shape[0]
        return dezzero.functions.reshape(self, shape)
```

## 38단계 - 형상 변환 함수 & dezzero/core.py

- Variable에서 reshape 사용하기
  - Dezzero reshape 함수를 호출하여 다음과 같이 사용함

```
x = Variable(np.random.randn(1, 2, 3))
y = x.reshape((2, 3))
print(y)
y = x.reshape(2,3)
print(y)
```

```
variable([[ 0.14618247 -0.83020541  0.30013048]
          [-1.70066527 -0.7853249   0.15724758]])
variable([[ 0.14618247 -0.83020541  0.30013048]
          [-1.70066527 -0.7853249   0.15724758]])
```

## 38단계 - 형상 변환 함수

- 행렬의 전치
  - 행렬을 전치해주는 클래스 및 함수 구현

**그림 38-3** 행렬 전치의 예

$$\mathbf{x} = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix} \quad \mathbf{x}^T = \begin{pmatrix} x_{11} & x_{21} \\ x_{12} & x_{22} \\ x_{13} & x_{23} \end{pmatrix}$$

## 38단계 - 형상 변환 함수 & dezzero/functions.py

- 행렬의 전치
  - 행렬을 전치해주는 클래스 및 함수 구현

```
class Transpose(Function):  
    def forward(self, x):  
        y = np.transpose(x)  
        return y  
    def backward(self, gy):  
        gx = transpose(gy)  
        return gx  
  
def transpose(x):  
    return Transpose()(x)
```

그림 38-3 행렬 전치의 예

$$\mathbf{x} = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix}$$

$$\mathbf{x}^T = \begin{pmatrix} x_{11} & x_{21} \\ x_{12} & x_{22} \\ x_{13} & x_{23} \end{pmatrix}$$

## 38단계 - 형상 변환 함수 & dezzero/core.py

- 행렬의 전치
  - Variable 인스턴스에서도 transpose 함수를 사용할 수 있도록 코드 추가

```
import dezzero
...
class Variable:
    ...
    def transpose(self):
        return dezzero.functions.transpose(self)

    @property
    def T(self):
        return dezzero.functions.transpose(self)
```

## 38단계 - 형상 변환 함수 & dezzero/core.py

- 행렬의 전치
  - @property는 인스턴스 변수로 사용할 수 있게 하는 데코레이터

```
x = Variable(np.random.rand(2,3))
print(x)
y = x.transpose()
print(y)
y = x.T
print(y)

variable([[0.91608834  0.35185748  0.49855669]
          [0.16827258  0.32771712  0.67739088]])
variable([[0.91608834  0.16827258]
          [0.35185748  0.32771712]
          [0.49855669  0.67739088]])
variable([[0.91608834  0.16827258]
          [0.35185748  0.32771712]
          [0.49855669  0.67739088]])
```

그림 38-3 행렬 전치의 예

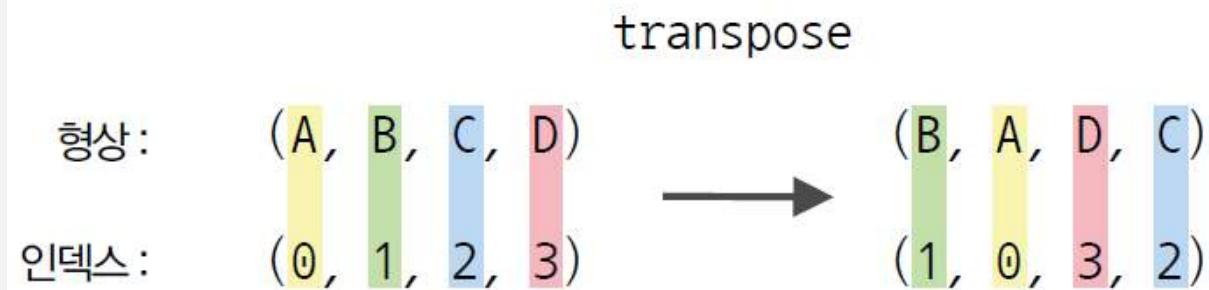
$$\mathbf{x} = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix} \quad \mathbf{x}^T = \begin{pmatrix} x_{11} & x_{21} \\ x_{12} & x_{22} \\ x_{13} & x_{23} \end{pmatrix}$$

## 38단계 - 형상 변환 함수 & dezzero/step38.py

- 행렬의 전치
  - @property는 인스턴스 변수로 사용할 수 있게 하는 데코레이터

```
A, B, C, D = 1, 2, 3, 4
x = np.random.rand(A, B, C, D)
print(x)
y = x.transpose(1, 0, 3, 2)
print(y)
```

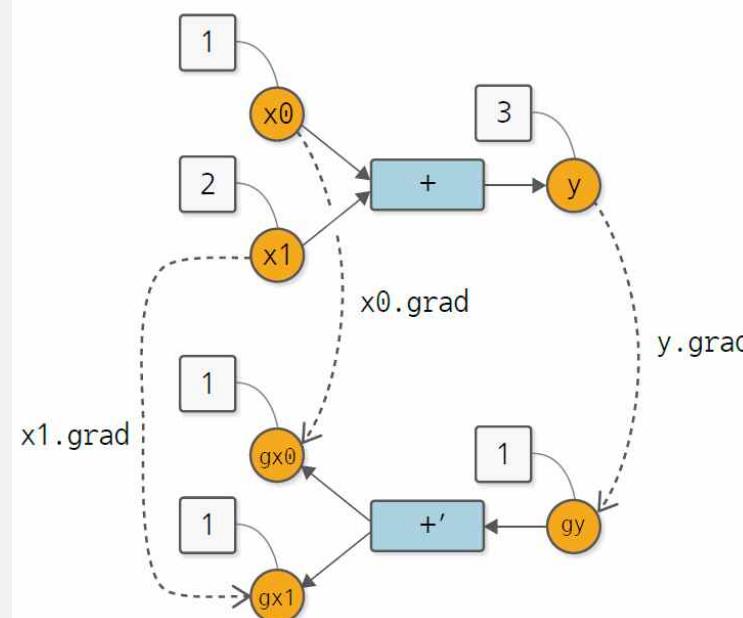
**그림 38-4** np.transpose 함수의 구체적인 예



## 39단계 - 합계 함수

- Sum 함수의 역전파
  - 덧셈을 수행한 후 변수  $y$ 로부터 역전파함
  - 변수  $x_0$ 와  $x_1$ 에는 출력 쪽에서 전해준 1이라는 기울기를 두 개로 “복사”하여 전달
  - 덧셈의 역전파는 원소가 2개인 벡터를 사용해도 동일함

그림 39-1 덧셈의 순전파와 역전파



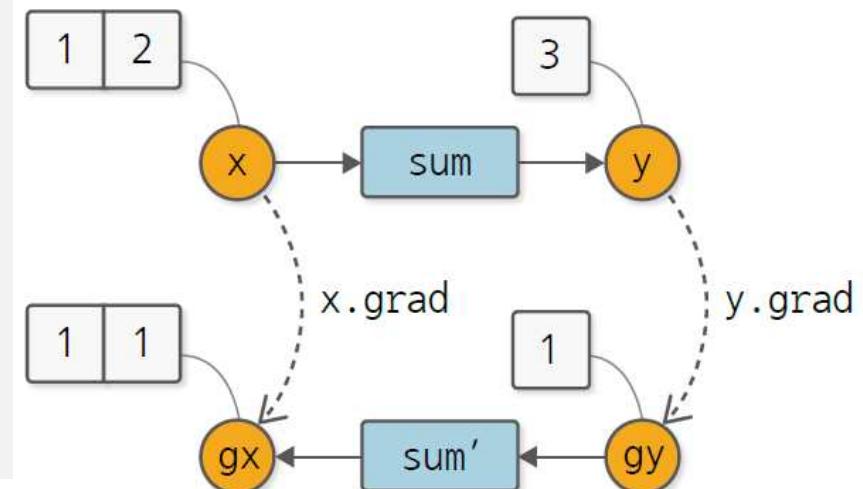
## 39단계 - 합계 함수

- Sum 함수의 역전파

...

- 2개의 원소로 구성된 벡터가 Sum 함수 적용 시 스칼라로 출력
- 역전파는 스칼라로 출력된 값을 다시 [1, 1] 형태의 벡터로 확장해 전파

그림 39-2 sum 함수의 계산 그래프 예 1(역전파 함수는  $\text{sum}'$ 로 표시)



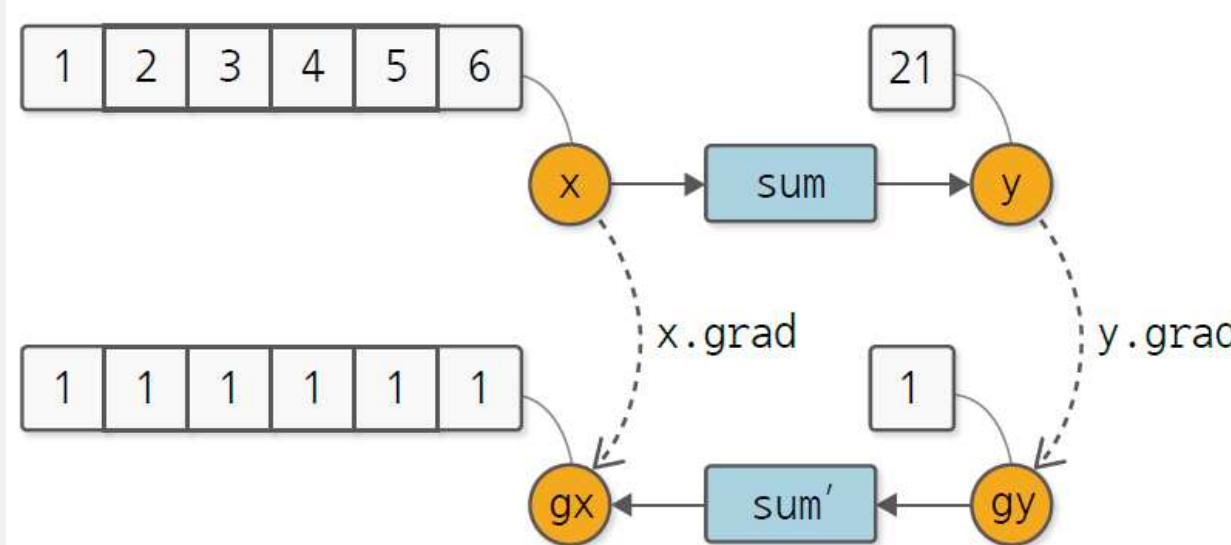
## 39단계 - 합계 함수

- Sum 함수의 역전파

...

- 원소가 2개 이상인 벡터의 합에 대한 역전파도 이끌어낼 수 있음, 벡터의 원소 수만큼 복사 함

그림 39-3 sum 함수의 계산 그래프 예 2



## 39단계 - 합계 함수 & dezzero/functions.py

- Sum 함수 구현
  - 지정한 형상에 맞게 원소를 복사하는 작업은 NumPy의 브로드캐스트와 같은 기능

```
class Sum(Function):  
    def forward(self, x):  
        self.x_shape = x.shape  
        y = x.sum()  
        return y  
  
    def backward(self, gy):  
        # step 40 단계에서 구현  
        gx = broadcast_to(gy, self.x_shape)  
        return gx  
  
def sum(x):  
    return Sum()(x)
```

## 39단계 - 합계 함수 & dezzero/step39.py

- Sum 함수 구현
  - x.grad와 x가 같은지 확인

```
import numpy as np
from dezzero import Variable
import dezzero.functions as F

x = Variable(np.array([1, 2, 3, 4, 5, 6]))
y = F.sum(x)
y.backward()
print(y)
print(x.grad)
```

```
variable(21)
variable([1 1 1 1 1 1])
```

```
import numpy as np
from dezzero import Variable
import dezzero.functions as F

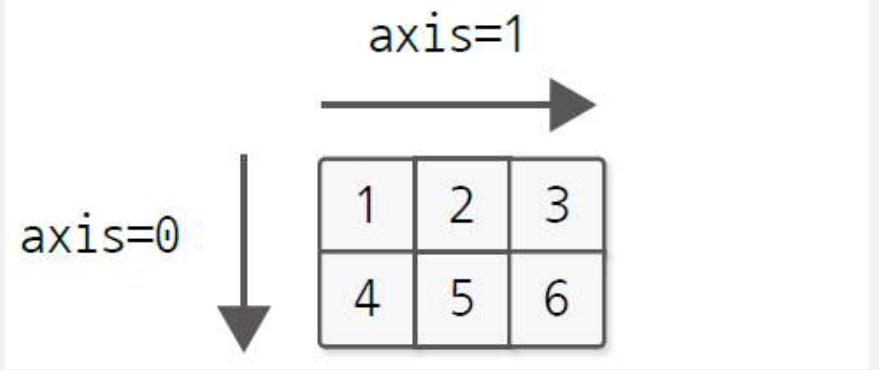
x = Variable(np.array([[1, 2, 3], [4, 5, 6]]))
y = F.sum(x)
y.backward()
print(y)
print(x.grad)
```

```
variable(21)
variable([[1 1 1]
          [1 1 1]])
```

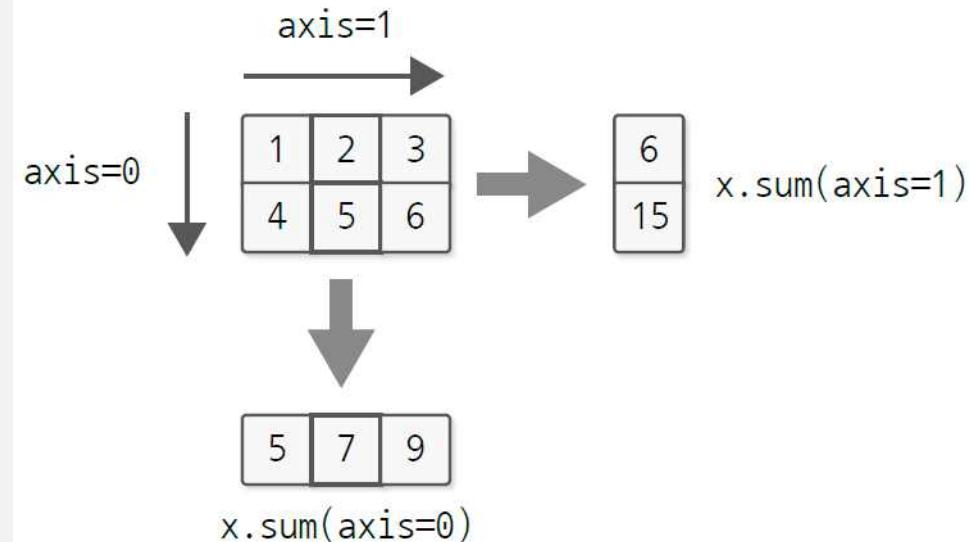
## 39단계 - 합계 함수 & dezzero/functions.py

- axis와 keepdims
  - axis = 축 (0은 행, 1은 열)
  - keepdims = 입력과 출력의 차원 수(축 수)를 똑같게 해줌 (np.sum 함수 파라미터)

**그림 39-4** ndarray 인스턴스의 axis(축의 인덱스)



**그림 39-5** axis에 따른 x.sum()의 계산 결과



## 39단계 - 합계 함수 & dezzero/functions.py

- axis와 keepdims
  - 사전에 reshape\_sum\_backward & broadcast\_to 함수 구현이 되어 있어야 함

```
from dezzero import utils

class Sum(Function):
    def __init__(self, axis, keepdims):
        self.axis = axis
        self.keepdims = keepdims

    def forward(self, x):
        self.x_shape = x.shape
        y = x.sum(axis=self.axis, keepdims=self.keepdims)
        return y

    def backward(self, gy):
        gy = utils.reshape_sum_backward(gy, self.x_shape, self.axis, self.keepdims)
        gx = broadcast_to(gy, self.x_shape)
        return gx
```

## 39단계 - 합계 함수 & dezzero/core.py & step39.py

- axis와 keepdims
  - 사전에 reshape\_sum\_backward & broadcast\_to 함수 구현이 되어 있어야 함

```
class Variable:
    ...
    def sum(self, axis=None, keepdims=False):
        return dezzero.functions.sum(self, axis, keepdims)
```

dezzero/core.py

```
x = Variable(np.array([[1, 2, 3], [4, 5, 6]]))
y = F.sum(x, axis=0)
y.backward()
print(y)
print(x.grad)

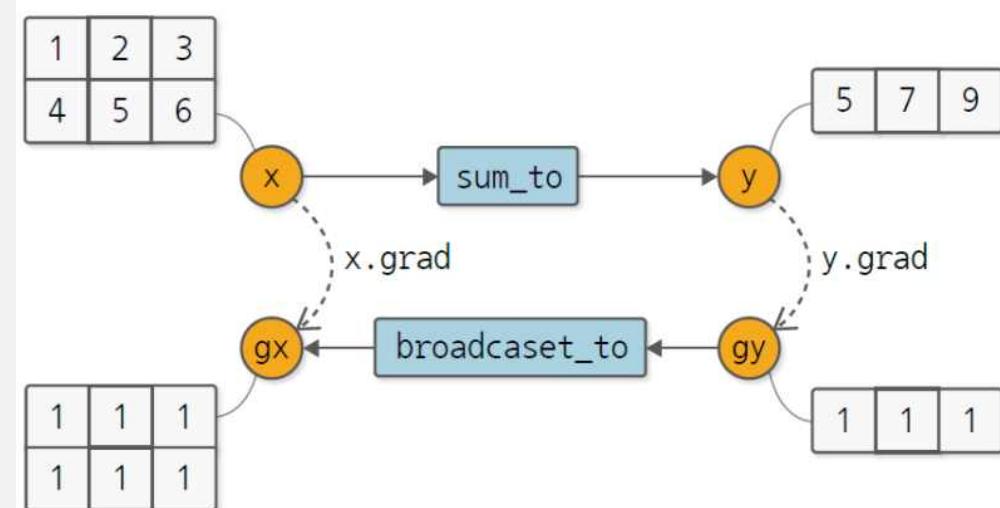
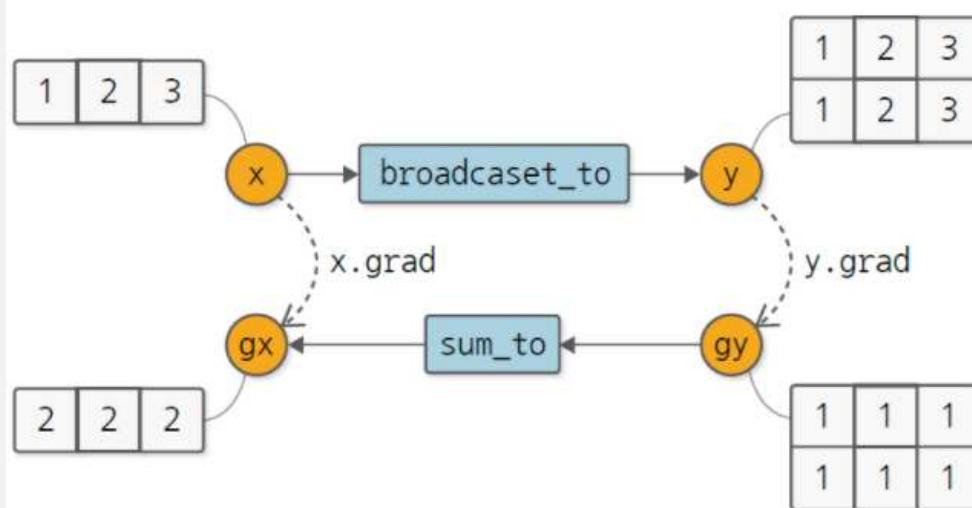
x = Variable(np.random.randn(2, 3, 4, 5))
y = x.sum(keepdims=True)
print(y.shape)
```

```
variable([5 7 9])
variable([[1 1 1]
          [1 1 1]])
(1, 1, 1, 1)
```

steps/step39.py

## 40단계 - 브로드캐스트 함수

- `broadcast_to(x, shape)` : x의 원소를 복사하여 shape 인수로 지정한 형상이 되도록 하는 함수
- `sum_to(x, shape)` : x의 원소의 합을 구해 shape 형상을 만들어주는 함수

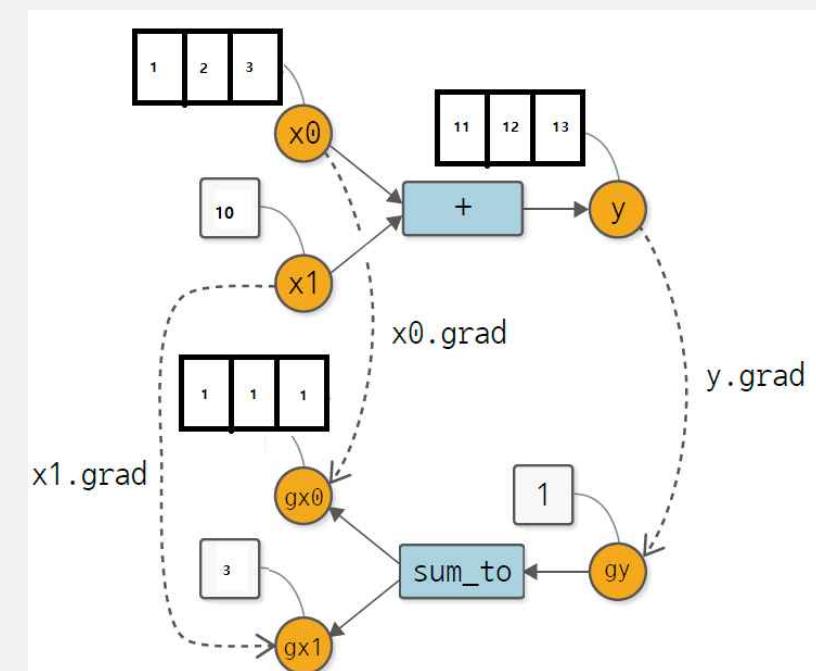


## 40단계 - 브로드캐스트 함수

- 브로드캐스트를 고려한 Add 클래스

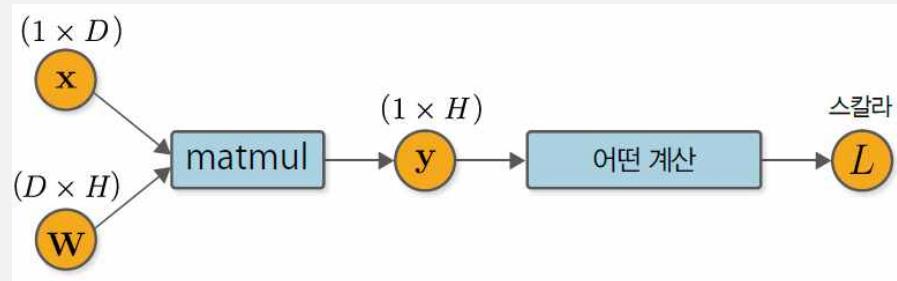
```
class Add(Function):
    def forward(self, x0, x1):
        self.x0_shape, self.x1_shape = x0.shape, x1.shape
        y = x0 + x1
        return y

    def backward(self, gy):
        gx0, gx1 = gy, gy
        if self.x0_shape != self.x1_shape: # for broadcast
            gx0 = dezero.functions.sum_to(gx0, self.x0_shape)
            gx1 = dezero.functions.sum_to(gx1, self.x1_shape)
        return gx0, gx1
```



## 41단계 - 행렬의 곱

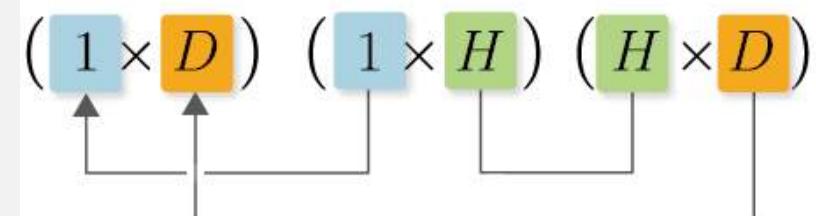
- 행렬 곱 역전파 식 도출



$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^T$$

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} W_{ij}$$

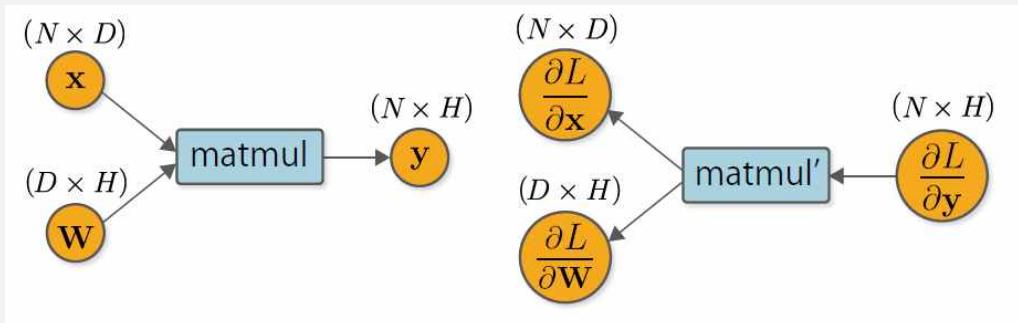
$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^T$$



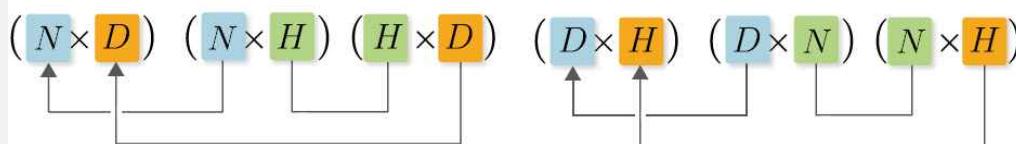
## 41단계 - 행렬의 곱

- 행렬 곱의 역전파

$$y = xW$$



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} W^T \quad \frac{\partial L}{\partial W} = x^T \frac{\partial L}{\partial y}$$



```
class MatMul(Function):
    def forward(self, x, W):
        y = x.dot(W)
        return y

    def backward(self, gy):
        x, W = self.inputs
        gx = matmul(gy, W.T)
        gw = matmul(x.T, gy)
        return gx, gw
```

## 42단계 - 선형 회귀

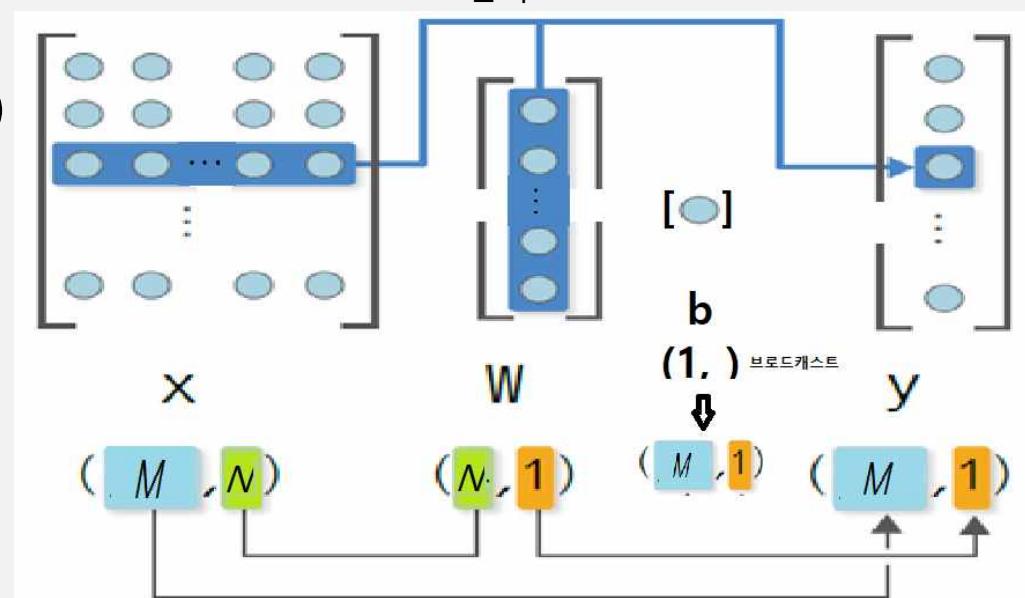
- 회귀 :  $x$ 로부터 실수값  $y$ 를 예측하는 것
- 선형 회귀 : 회귀 모델 중 예측값이 선형( $y = Wx + b$ )을 이루는 것

MSE (평균 제곱 오차)

$$L = \frac{1}{N} \sum_{i=1}^N (f(x_i) - y_i)^2$$

$x.shape = (M, N)$

\*  $W$  : 스칼라



## 42단계 - 선형 회귀

```

def predict(x):
    y = F.matmul(x, W) + b
    return y

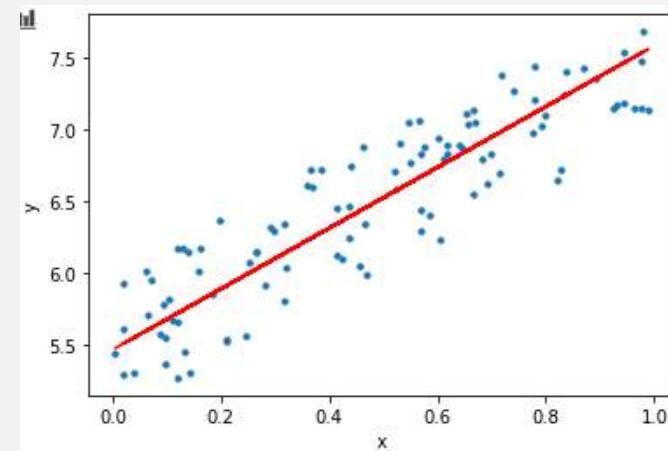
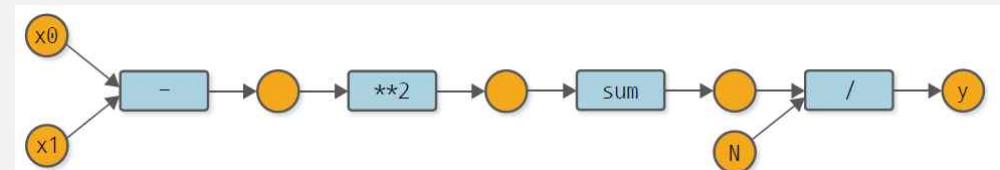
def mean_squared_error(x0, x1):
    return MeanSquaredError()(x0, x1)

for i in range(iters):
    y_pred = predict(x)
    loss = mean_squared_error(y, y_pred)

    W.cleargrad()
    b.cleargrad()
    loss.backward()

    W.data -= lr * W.grad.data
    b.data -= lr * b.grad.data
    print(f'W : {W}, b : {b}, loss : {loss}')

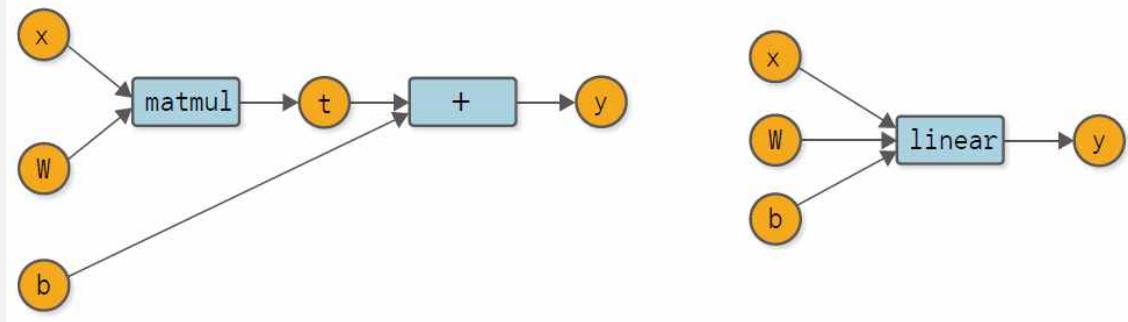
```



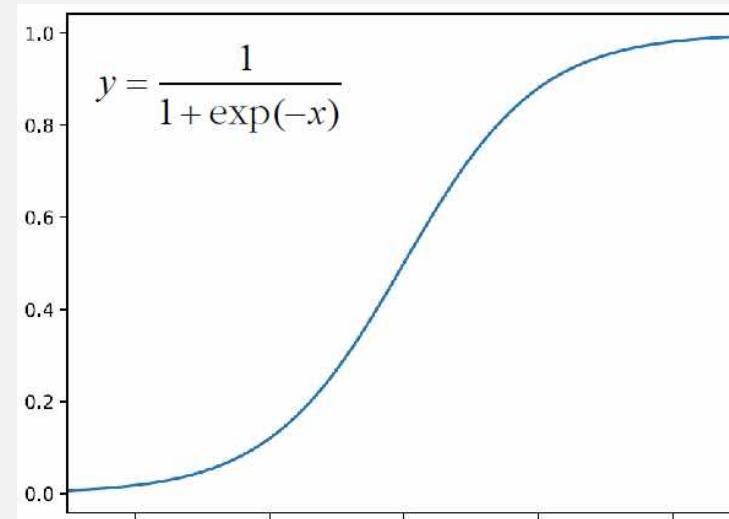
W : variable([[2.11807369]]), b : variable([5.46608905]), loss : variable(0.07908606512411756)

## 43단계 - 신경망

$$y = F.\text{matmul}(x, W) + b \rightarrow y = F.\text{linear}(x, w, b)$$



시그모이드



신경망은 선형 변환의 출력에 활성화 함수 수행.  
선형 변환 -> 활성화 함수-> 선형 변환 -> ... 형태를 가짐

## 43단계 - 신경망

```

def predict(x):
    y = F.linear(x, w1, b1)
    y = F.sigmoid(y)
    y = F.linear(y, w2, b2)
    return y

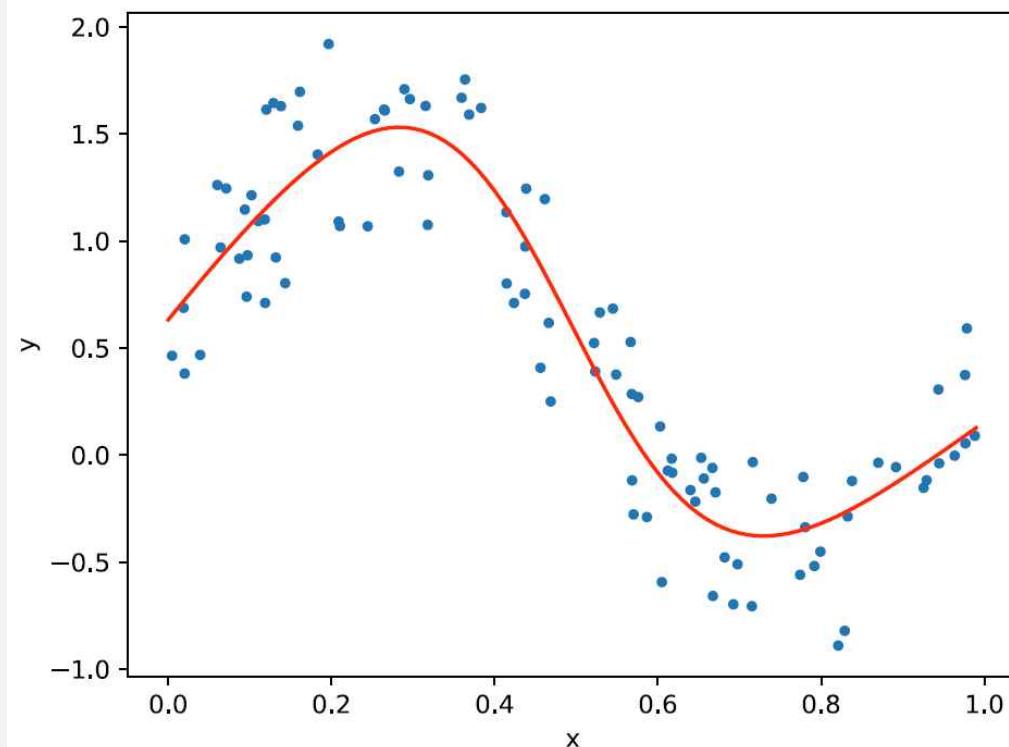
lr = 0.2
iters = 10000

for i in range(iters):
    y_pred = predict(x)
    loss = F.mean_squared_error(y, y_pred)

    w1.cleargrad()
    b1.cleargrad()
    w2.cleargrad()
    b2.cleargrad()
    loss.backward()

    w1.data -= lr * w1.grad.data
    b1.data -= lr * b1.grad.data
    w2.data -= lr * w2.grad.data
    b2.data -= lr * b2.grad.data

```



$W_1 : [-1.815772 \ -1.28109675 \ -1.81232202 \ -2.59936717 \ -1.92147264 \ -9.41186652 \ -1.40881487 \ -1.01819045 \ -1.77196159 \ -2.02364167]$ ,  $b_1 : [-0.02666195 \ 0.21627403 \ -0.02635124 \ 1.09611584 \ -0.01360137 \ 4.63231273 \ 0.12779263 \ 0.57492124 \ -0.0194288 \ 0.96386394]$ ,  $w_2 : [-1.45999069] \ [-0.68388866] \ [-1.45471095] \ [-1.6058759] \ [-1.61935087] \ [4.78915775] \ [-0.86453031] \ [-0.27376012] \ [-1.39259892] \ [-1.08907466]$ ,  $b_2 : [1.90125216]$ . loss : variable(0.07852506088311084)

## 48단계: 다중 클래스 분류

- Spiral dataset

```
import dezzero

x, t = dezzero.datasets.get_spiral(train=True)
print(x.shape)
print(t.shape)

print(x[10],t[10])
print(x[110],t[110])
```

- dezzero/datasets.py 모듈을 통해 Spiral dataset을 읽어옴
- train = True : 학습(훈련)용 데이터를 반환  
\* train = False : 테스트용 데이터

## 48단계: 다중 클래스 분류 & steps/steps48.py

- **다중 클래스 분류**

```
# 하이퍼파라미터 설정
Max_epoch = 300
Batch_size = 30
Hidden_size = 10
Lr = 1.0

# 데이터 읽기 / 모델, 옵티마이저 설정
x, t = dezero.datasets.get_spiral(train=True)
model = MLP((hidden_size, 3))
optimizer = optimizers.SGD(lr).setup(model)
```

- **하이퍼파라미터** : 사람이 설정하는 매개변수로 은닉층 수, 학습률 등을 포함
- **epoch** : 전체 데이터 셋에 대해 한 번 학습을 완료한 상태를 나타내는 단위

## 48단계: 다중 클래스 분류 & steps/steps48.py

- **다중 클래스 분류**

```
for epoch in range(max_epoch):
    # 데이터셋의 인덱스 생성
    index = np.random.permutation(data_size)
    sum_loss = 0

    # 미니배치 생성
    for i in range(max_iter):
        batch_index = index[i * batch_size:(i + 1) * batch_size]
        batch_x = x[batch_index]
        batch_t = t[batch_index]
```

- 미니배치 : 데이터가 많을 경우 한번에 처리하기 보단 조금씩 무작위로 모아서 처리를 하는데,  
이 때 모아진 데이터뭉치를 미니배치라 함

## 49단계: Dataset 클래스와 전처리 & dezzero/datasets.py

- Datasets 구현과 전처리
  - 이 전 단계(48단계)의 x, t는 ndarray 인스턴스로, 대규모 데이터 사용 시 메모리 문제가 발생할 수 있기에 별도의 전용 데이터셋 클래스가 필요
  - 머신러닝에서 모델에 데이터를 입력하기 전 데이터를 특정한 형태로 가공을 하는 경우가 많음  
이러한 경우를 위해 전처리 기능을 Datasets에 추가
  - `__getitem__` : 지정된 인덱스에 위치하는 데이터 반환
  - `__len__` : 데이터의 길이 반환

## 50단계: 미니배치를 뽑아주는 DataLoader & dezzero/dataloaders.py

- Dataloaders 구현
  - 미니배치 생성과 데이터셋 뒤섞기 등의 기능을 제공
  - 반복자 구조를 이용
  - `__iter__` : `iter` 함수와 동일한 기능을 할 수 있게 해주는 python 특수 메소드  
객체를 순서대로 액세스 할 수 있도록 해줌
  - `__next__` : `next` 함수와 동일한 기능을 할 수 있게 해주는 python 특수 메소드  
리스트 안의 원소를 차례대로 꺼내옴

## 50단계: 미니배치를 뽑아주는 DataLoader & dezzero/functions.py

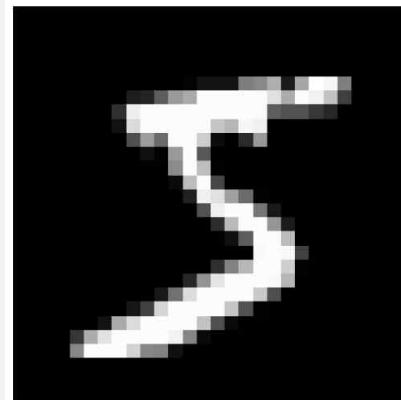
- Accuracy 함수 구현

```
def accuracy(y,t):  
    y,t = as_variable(y), as_variable(t)  
  
    pred = y.data.argmax(axis=1).reshape(t.shape)  
    result = (pred == t.data)  
    acc = result.mean()  
    return Variable(as_array(acc))
```

- 입력 데이터(y)와 정답 데이터(t) 사이의 정확도를 계산  
비교 결과로 True/False의 텐서(ndarray)가 되며, 이 중 True의 비율이 정확도가 됨

## 51단계: MNIST 학습 & steps/step51.py

- MNIST 데이터셋
  - 훈련용 데이터 - 6만개 / 테스트용 데이터 - 1만개
  - 0 ~ 9 까지의 숫자를 나타내는 손글씨 이미지 데이터
  - (1,28,28)의 데이터 형상으로 1채널(그레이스케일)의 28x28 이미지 데이터



## 51단계: MNIST 학습 & dezzero/functions.py

- ReLU 함수 구현

```
class ReLu(Function):  
    def forward(self,x):  
        y = np.maximum(x,0.0)  
        return y  
  
    def backward(self,gy):  
        x, = self.inputs  
        mask = x.data > 0  
        gx = gy*mask  
        return gx  
  
def relu(x):  
    return ReLu()(x)
```

- 0보다 큰 입력은 그대로 출력하고, 이하는 0을 출력하는 함수

# 제 5고지 DeZero의 도전

## 52단계:GPU 지원

- 딥러닝으로 하는 계산은 ‘행렬의 곱’이 대부분을 차지
- 행렬의 곱은 병렬로 계산이 가능
- 병렬 계산에는 CPU보다 GPU가 훨씬 뛰어남
- GPU

- 엔비디아 GPU



- Google Colab



<출처: [http://m.ddaily.co.kr/m/m\\_article/?no=195642](http://m.ddaily.co.kr/m/m_article/?no=195642)>

\* <https://www.python.org/dev/peps/pep-0008>

## 52단계:GPU 지원

- CUPY는 GPU를 활용하여 병렬 계산을 해주는 라이브러리
- 설치

```
...  
$ pip install cupy  
...
```

- CUPY는 NUMPY와 API가 거의 같음
- 넘파이 지식을 쿠파이에서도 그대로 활용 가능
  - 거의 같지만 완전히 똑같지 않음

## 52단계:GPU 지원

- 넘파이와 쿠파이의 다차원 배열을 서로 변환

```
import numpy as np
import cupy as cp

# 넘파이 -> 쿠파이
n = np.array([1, 2, 3])
c = cp.asarray(n)
assert type(c) == cp.ndarray

# 쿠파이 -> 넘파이
c = cp.array([1, 2, 3])
n = np.asnumpy(c)
assert type(n) == np.ndarray
```

## 52단계:GPU 지원

- `cp.get_array_module` 함수는 주어진 데이터에 적합한 모듈을 돌려 줌

```
# x가 넘파이 배열인 경우
n = np.array([1, 2, 3])
xp = cp.get_array_module(x)
assert xp == np
```

```
# x가 파이 배열인 경우
c = cp.array([1, 2, 3])
xp = cp.get_array_module(x)
assert xp == cp
```

## 52단계:GPU 지원

- dezzero/cuda.py 모듈
  - 넘파이와 쿠파이 임포트
  - 세가지 함수 추가

```
import numpy as np
gpu_enable = True
try:
    import cupy as cp
    cupy = cp
except ImportError:
    gpu_enable = False
from dezzero import Variable
```

```
# 인수 x에 대응하는 모듈을 돌려줌
def get_array_module(x):
    ...

# 넘파이의 ndarray로 변환하는 함수
def as_numpy(x):
    ...

# 쿠파이의 ndarray로 변환하는 함수
def as_cupy(x):
    ...
```

## 52단계:GPU 지원

- 다른 클래스들에 GPU 대응 기능 추가
  - dezzero/core.py 모듈 수정

```

...
try:
    import cupy
    # 두 배열 타입을 동적으로 변경
    array_types = (np.ndarray, cupy.ndarray)
except ImportError:
    array_types = (np.ndarray)

class Variable:
    def __init__(self, data, name=None):
        if data is not None:
            # data로 cp.ndarray가 넘어와도 대응할 수 있도록 수정
            if not isinstance(data, array_types):
                raise TypeError('{} is not supported'.format(type(data)))

    ...
    def backward(self, retain_grad=False, create_graph=False):
        if self.grad is None:
            # 기울기를 자동으로 보완하는 부분 수정
            xp = dezzero.cuda.get_array_module(self.data)
            self.grad = Variable(xp.ones_like(self.data))

    ...
    # 넘파이 다차원 배열을 보관하던 인스턴스 변수 data를 GPU나 CPU로 전송하는 기능
    def to_cpu(self):
        if self.data is not None:
            self.data = dezzero.cuda.as_numpy(self.data)

    def to_gpu(self):
        if self.data is not None:
            self.data = dezzero.cuda.as_cupy(self.data)

```

## 52단계:GPU 지원

- 다른 클래스들에 GPU 대응 기능 추가
  - dezzero/core.py 모듈 수정
  - dezzero/layers.py 모듈 수정

```
class Layer:  
    ...  
    # Layer 클래스의 매개변수를 GPU나 CPU로 전송하는 기능 추가  
    def to_cpu(self):  
        for param in self.params():  
            param.to_cpu()  
  
    def to_gpu(self):  
        for param in self.params():  
            param.to_gpu()
```

## 52단계:GPU 지원

- 다른 클래스들에 GPU 대응 기능 추가
  - dezzero/core.py 모듈 수정
  - dezzero/layers.py 모듈 수정
  - dezzero/dataloaders.py 모듈 수정

```

fromdezeroimportcuda
#데니터셋을미니배치로뽑는역할을수행
classDataLoader:
    def__init__(self,dataset,batch_size,shuffle=True,gpu=False):
        ...
        self.gpu=gpu

    def__next__(self):
        ...
        #미니배치
        xp=cuda.cupyifself.gpuelsenp
        x=xp.array([example[0]forexampleinbatch])
        t=xp.array([example[1]forexampleinbatch])
        ...

        #gpu플래그를확인,쿠파이와넘파이중알맞은다차원배열로만듦
        defto_cpu(self):
            self.gpu=False

        defto_gpu(self):
            self.gpu=True

```

## 52단계:GPU 지원

- 다른 클래스들에 GPU 대응 기능 추가
  - dezzero/core.py 모듈 수정
  - dezzero/layers.py 모듈 수정
  - dezzero/dataloaders.py 모듈 수정
  - dezzero/functions.py 모듈 수정

```
# dezzero/functions.py
# 넘파이와 쿠파이 어느 경우에도 작동하도록 수정
class Sin(Function):
    def forward(self, x):
        xp = cuda.get_array_module(x)
        y = xp.sin(x)
        return y

# dezzero/core.py
# 사칙연산 코드 수정
def as_array(x, array_module=np):
    if np.isscalar(x):
        return array_module.array(x)
    return x

def add(x0, x1):
    x1 = as_array(x1, dezzero.cuda.get_array_module(x0.data))
    return Add()(x0, x1)
# mul, sub, rsub, div, rdiv도 똑같이 수정
...
```

## 53단계:모델 저장 및 읽어오기

- 모델이 가지는 매개변수를 외부 파일로 저장 및 읽어오는 기능을 만듦
- 학습 중인 모델의 ‘스냅샷’ 을 저장하거나 학습된 매개변수를 읽어와서 추론만 수행 가능
- DeZero의 매개변수는 Parameter 클래스로 구현
- Parameter의 데이터는 인스턴스 변수 data에 ndarray 인스턴스로 보관
- ndarray 인스턴스를 외부 파일로 저장 및 읽어오기

## 53단계:모델 저장 및 읽어오기

- 넘파이 함수를 사용하면 ndarray 인스턴스를 저장하고 읽어올 수 있음

```
x1 = np.array([1, 2, 3])
x2 = np.array([4, 5, 6])
data = { 'x1' : x1, 'x2' : x2 }

# ndarray 인스턴스를 외부 파일로 저장
np.save('test.npy', x1)
# 여러 개를 한번에 저장
np.saves('test.npz', x1=x1, x2=x2) # 키워드 인수를 지정
np.saves('test.npz', **data) # 딕셔너리를 자동으로 전개

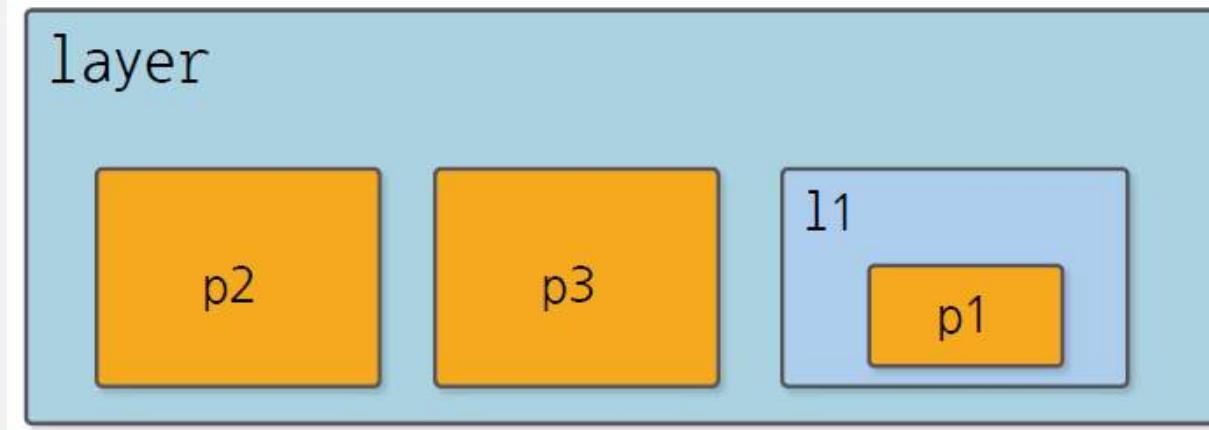
# 파일 읽어오기
arrays = np.load('test.npy')
arrays = np.load('test.npz')
x1 = arrays['x1']
x2 = arrays['x2']

print(x1)
print(x2)
```

## 53단계: 모델 저장 및 읽어오기

- Layer 클래스는 계층의 구조를 표현

그림 53-1 Layer 클래스는 계층의 구조를 표현함



## 53단계:모델 저장 및 읽어오기

- 계층 구조로부터 Parameter를 ‘하나의 평평한 딕셔너리’로 뽑아 냄
- \_flatten\_params 메서드 추가

```
class Layer:  
    ...  
    def _flatten_params(self, params_dict, parent_key=""):  
        for name in self._params:  
            obj = self.__dict__[name]  
            key = parent_key + '/' + name if parent_key else name  
  
            # 재귀적으로 호출  
            if isinstance(obj, Layer):  
                obj._flatten_params(params_dict, key)  
            else:  
                params_dict[key] = obj
```

## 53단계:모델 저장 및 읽어오기

- save\_weights와 load\_weights 메서드 추가

```
class Layer:  
    ...  
    def save_weights(self, path):  
        self.to_cpu() # ndarray 확인  
        params_dict = {}  
        self._flatten_params(params_dict) # 평탄화  
        array_dict = {key: param.data for key,  
                     param in params_dict.items() if param is not None}  
        try:  
            np.savez_compressed(path, **array_dict) # 압축저장  
        except (Exception, KeyboardInterrupt) as e:  
            if os.path.exists(path):  
                os.remove(path)  
            raise  
  
    def load_weights(self, path):  
        npz = np.load(path)  
        params_dict = {}  
        self._flatten_params(params_dict)  
        for key, param in params_dict.items():  
            param.data = npz[key]
```

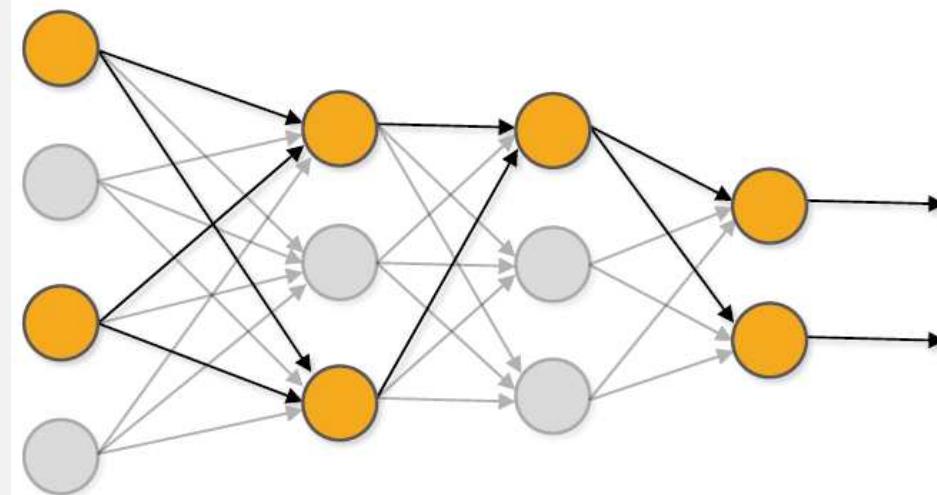
## 54단계: 드롭아웃과 테스트 모드

- 과대적합이 일어나는 주요 원인
  - 훈련 데이터 부족
    - \* 데이터 확장(data augmentation)
  - 모델의 표현력이 지나치게 높음
    - \* 가중치 감수(Weight Decay)
    - \* 드롭아웃(Dropout) # 가장 효과적이고 실무에 많이 사용됨
    - \* 배치 정규화(Batch Normalization)

## 54단계: 드롭아웃과 테스트 모드

- 드롭아웃은 뉴런을 임의로 삭제(비활성화)하면서 학습하는 방법

그림 54-1 드롭아웃을 적용해 학습할 때의 신경망 동작



- 양상을 학습은 드롭아웃과 가까운 관계

\* <https://www.python.org/dev/peps/pep-0008>

## 54단계: 드롭아웃과 테스트 모드

- 양상블 학습
  - 여러 모델을 개별적으로 학습시킨 후 추론 시 모든 모델의 출력을 평균 내는 방법
- 드롭아웃은 학습 시 뉴런을 임의로 삭제하는데, 이를 매번 다른 모델을 학습하고 있다고 해석
- 양상블 학습과 같은 효과를 신경망 하나에서 가상으로 시뮬레이션한다고 간주할 수 있음

## 54단계: 드롭아웃과 테스트 모드

- **다이렉트 드롭아웃**

- 테스트 시에는 모든 뉴런을 사용하면서도 양상을 학습처럼 동작하게끔 ‘흉내’ 내야 함
- 우선 모든 뉴런을 써서 출력을 계산하고, 그 결과를 ‘약화’ 시킴
- 약화하는 비율은 학습 시 살아남은 뉴런의 비율

```
dropout_ratio = 0.6
x = np.ones(10)

# 학습 시
mask = np.random.rand(*x.shape) > dropout_ratio
y = x * mask

# 테스트 시
scale = 1 - dropout_ratio # 학습 시에 살아남은 뉴런의 비율
y = x * scale
```

## 54단계: 드롭아웃과 테스트 모드

- 역 드롭아웃
  - 스케일 맞추기를 ‘학습할 때’ 수행
  - 미리 뉴런의 값에  $1/\text{scale}$ 을 곱해두고, 테스트 때는 아무런 동작도 하지 않음

```
dropout_ratio = 0.6
x = np.ones(10)

# 학습 시
scale = 1 - dropout_ratio
mask = np.random.rand(*x.shape) > dropout_ratio
y = x * mask / scale

# 테스트 시
y = x
```

## 54단계: 드롭아웃과 테스트 모드

- 디렉트 드롭아웃은 dropout\_ratio를 고정해두고 학습해야 하기 때문에 값을 중간에 바꾸면 테스트 시의 동작과 어긋나게 됨
- 역 드롭아웃은 학습할 때 dropout\_ratio를 동적으로 변경할 수 있음
- 이러한 이유 때문에 많은 딥러닝 프레임워크에서 역 드롭아웃 방식을 채용하고 있음

## 54단계: 드롭아웃과 테스트 모드

- 드롭아웃을 사용하려면 학습 단계인지 테스트 단계인지 구분해야 함
- dezzero/core.py 모듈 수정

```
class Config:  
    enable_backprop = True  
    train = True  
  
    @contextlib.contextmanager  
    def using_config(name, value):  
        old_value = getattr(Config, name)  
        setattr(Config, name, value)  
        yield  
        setattr(Config, name, old_value)  
  
    def test_mode(): # 모드 변경을 위한 함수  
        return using_config('train', False)
```

## 54단계: 드롭아웃과 테스트 모드

- dezero/function.py 모듈에 dropout 구현

```
def dropout(x, dropout_ratio=0.5):
    x = as_variable(x)

    if dezzero.Config.train:
        xp = cuda.get_array_module(x)
        mask = xp.random.rand(*x.shape) > dropout_ratio
        scale = xp.array(1.0 - dropout_ratio).astype(x.dtype)
        y = x * mask / scale
        return y
    else:
        return x
```

```
import numpy as np
from dezzero import test_mode
import dezzero.functions as F

x = np.ones(5)
print(x)

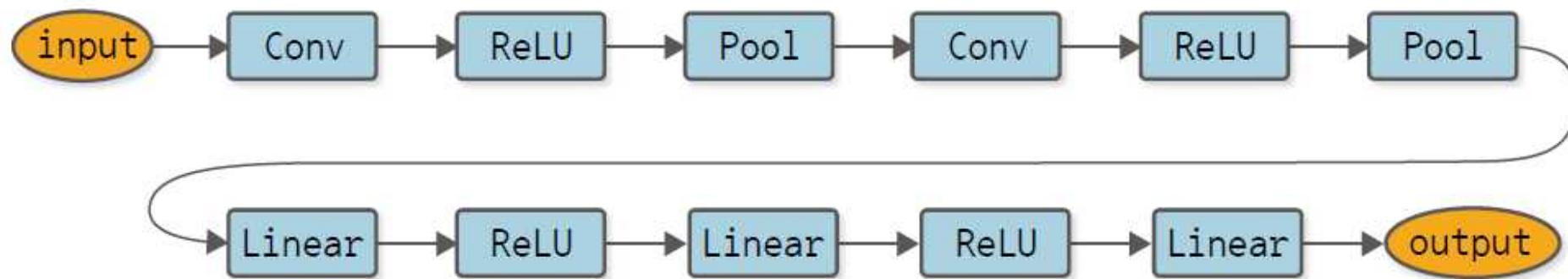
# When training
y = F.dropout(x)
print(y)

# When testing (predicting)
with test_mode():
    y = F.dropout(x)
    print(y)
```

## 55단계 : CNN 메커니즘(1)

- CNN 신경망의 구조

**그림 55-1** CNN 신경망 예(계산 그래프를 계층 단위로 그리고, 변수는 입력과 출력만 표시함. Conv=합성곱층, Pool=풀링층)



## 55단계 : CNN 메커니즘(1)

- 합성곱 연산

그림 55-2 합성곱 연산의 예(합성곱 연산은  $\circledast$ 로 표기)

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

입력 데이터

$\circledast$

2	0	1
0	1	2
1	0	2

필터



15	16
6	15

출력 데이터

## 55단계 : CNN 메커니즘(1)

- 연산 계산 순서

그림 55-3 합성곱 연산의 계산 순서

$$1*2 + 2*0 + 3*1 + 0*0 + 1*1 + 2*2 + 3*1 + 0*0 + 1*2 = \underline{\underline{15}}$$

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

⊗

2	0	1
0	1	2
1	0	2



15	

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

⊗

2	0	1
0	1	2
1	0	2



15	16

## 55단계 : CNN 메커니즘(1)

- 연산 계산 순서

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

(\*)

2	0	1
0	1	2
1	0	2



15	16
6	

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

(\*)

2	0	1
0	1	2
1	0	2

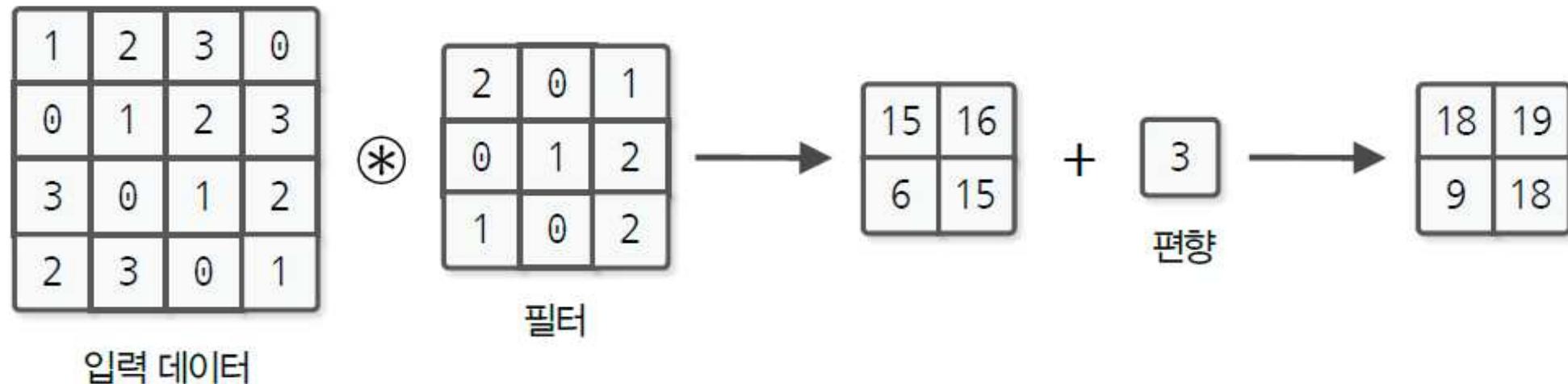


15	16
6	15

## 55단계 : CNN 메커니즘(1)

- 연산의 편향

그림 55-4 합성곱 연산의 편향



## 55단계 : CNN 메커니즘(1)

- 패딩

그림 55-5 합성곱 연산의 패딩 처리

0	0	0	0	0	0
0	1	2	3	0	0
0	0	1	2	3	0
0	3	0	1	2	0
0	2	3	0	1	0
0	0	0	0	0	0

입력 데이터(패딩:1)

⊗

2	0	1
0	1	2
1	0	2

필터

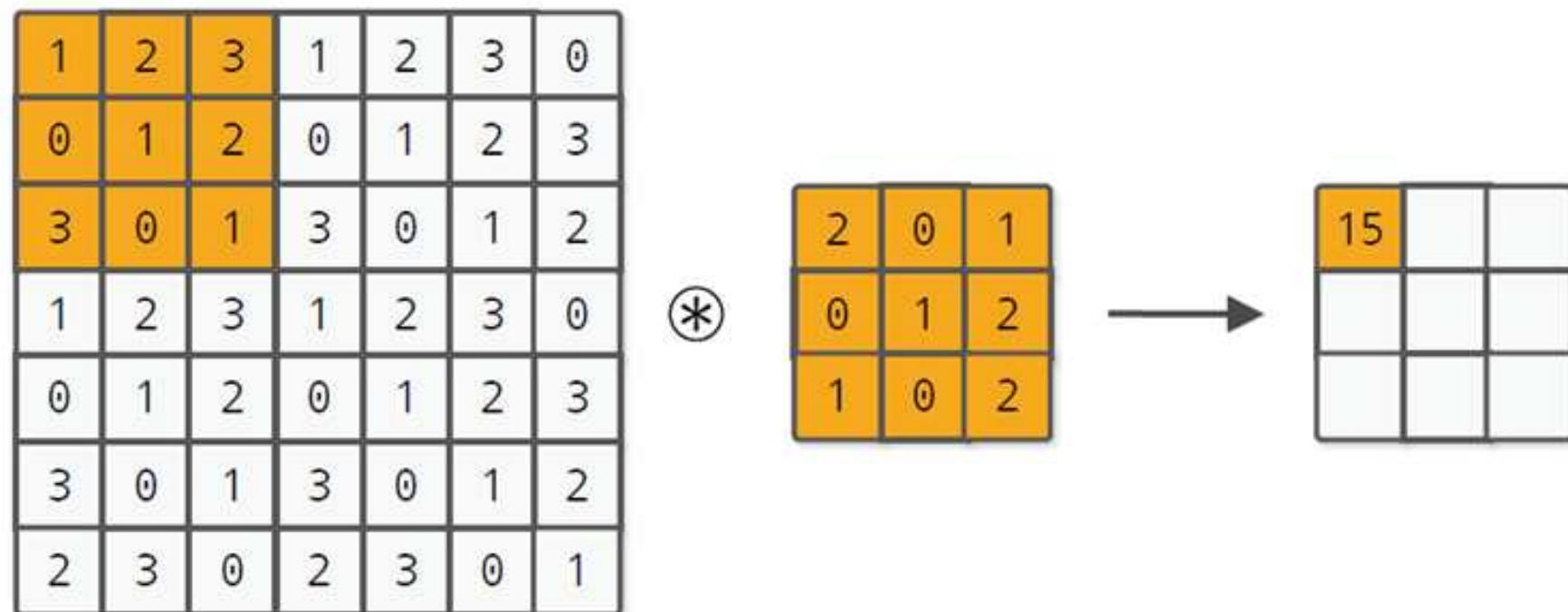


7	12	10	2
4	15	16	10
10	6	15	6
8	10	4	3

출력 데이터

## 55단계 : CNN 메커니즘(1)

- 스트라이드 [그림 55-6](#) 스트라이드가 2인 합성곱 연산의 예



## 55단계 : CNN 메커니즘(1)

- 스트라이드

스트라이드: 2

1	2	3	1	2	3	0	
0	1	2	0	1	2	3	
3	0	1	3	0	1	2	
1	2	3	1	2	3	0	
0	1	2	0	1	2	3	
3	0	1	3	0	1	2	
2	3	0	2	3	0	1	

⊗

2	0	1
0	1	2
1	0	2



15	11	

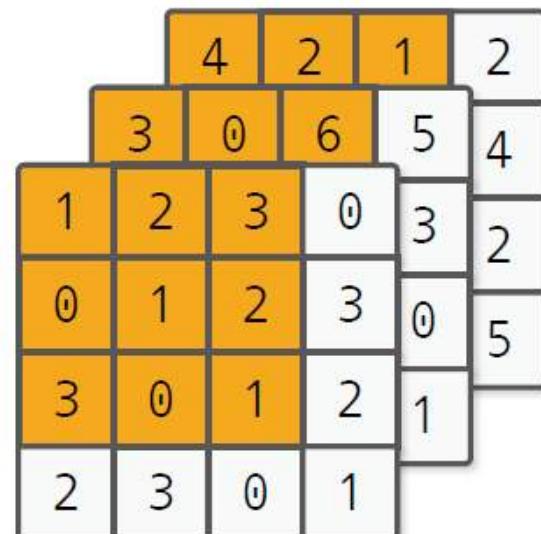
## 55단계 : CNN 메커니즘(1)

- 출력 크기 계산 방법
- 입력크기 + 패딩 \* 2 - 커널크기 // 스트라이드 + 1
- $(4,4) + (1,1) * 2 - (3,3) // (1,1) + 1$
- $4 + 2 - 3 // 1$
- $3 + 1$
- 4

## 56단계 : CNN 메커니즘(2)

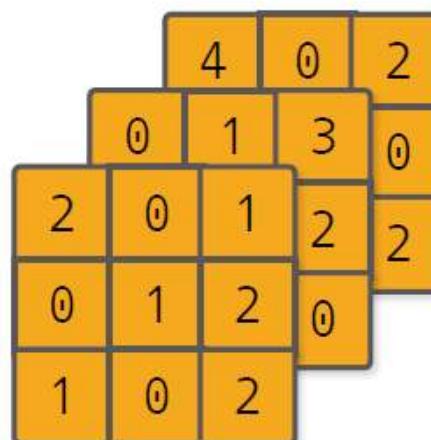
- 3차원 텐서

그림 56-1 3차원 텐서에서의 합성곱 연산 예



입력 데이터

⊗



필터

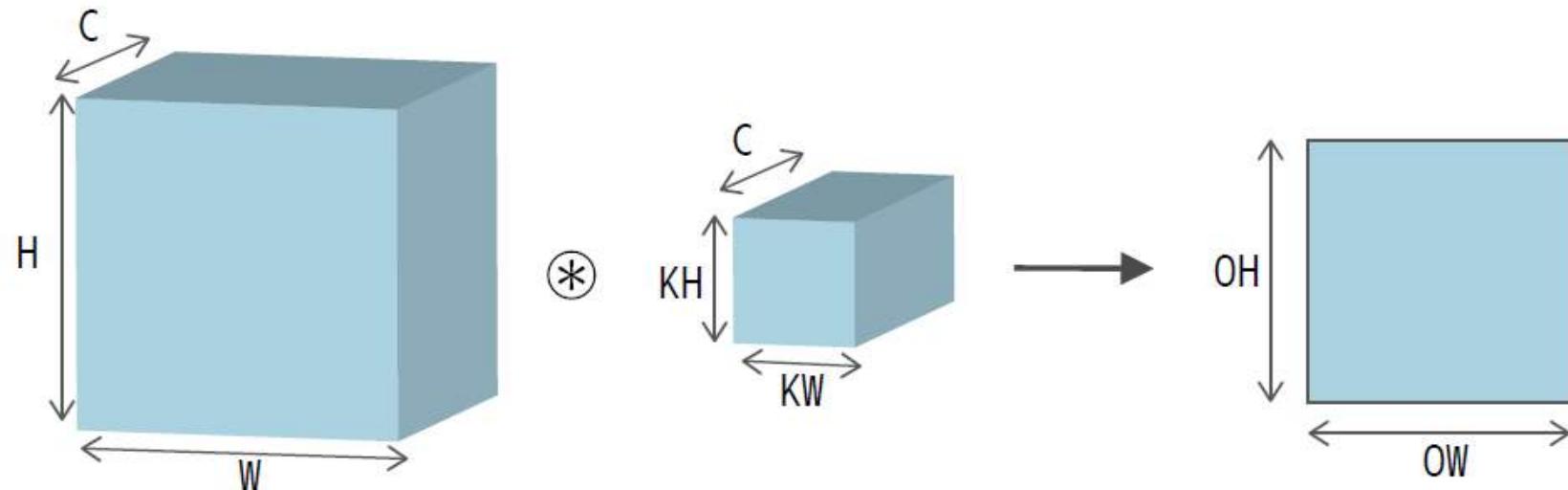


출력 데이터

## 56단계 : CNN 메커니즘(2)

- 블록으로 생각하기

그림 56-2 합성곱 연산을 블록으로 생각하기

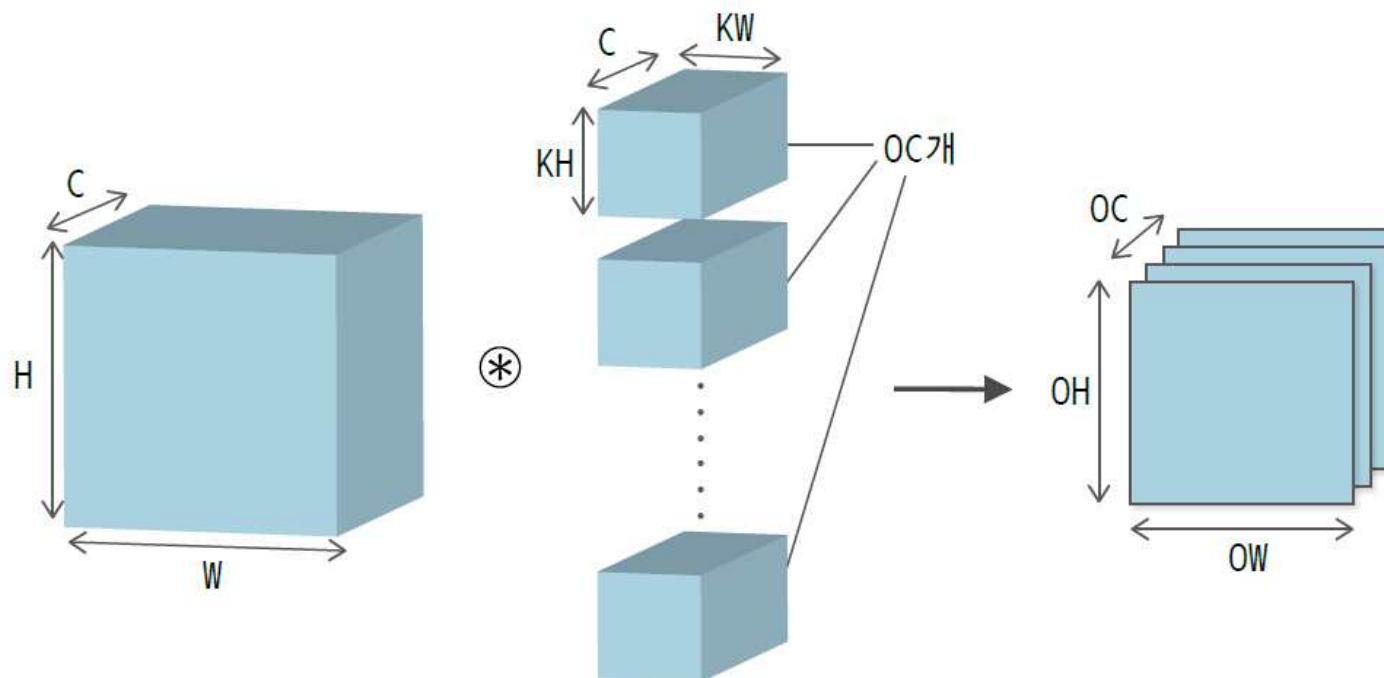


형상	$(C, H, W)$	$\otimes$	$(C, KH, KW)$	$\rightarrow$	$(1, OH, OW)$
----	-------------	-----------	---------------	---------------	---------------

## 56단계 : CNN 메커니즘(2)

- 다수의 필터 연산

그림 56-3 다수의 필터를 사용한 합성곱 연산의 예

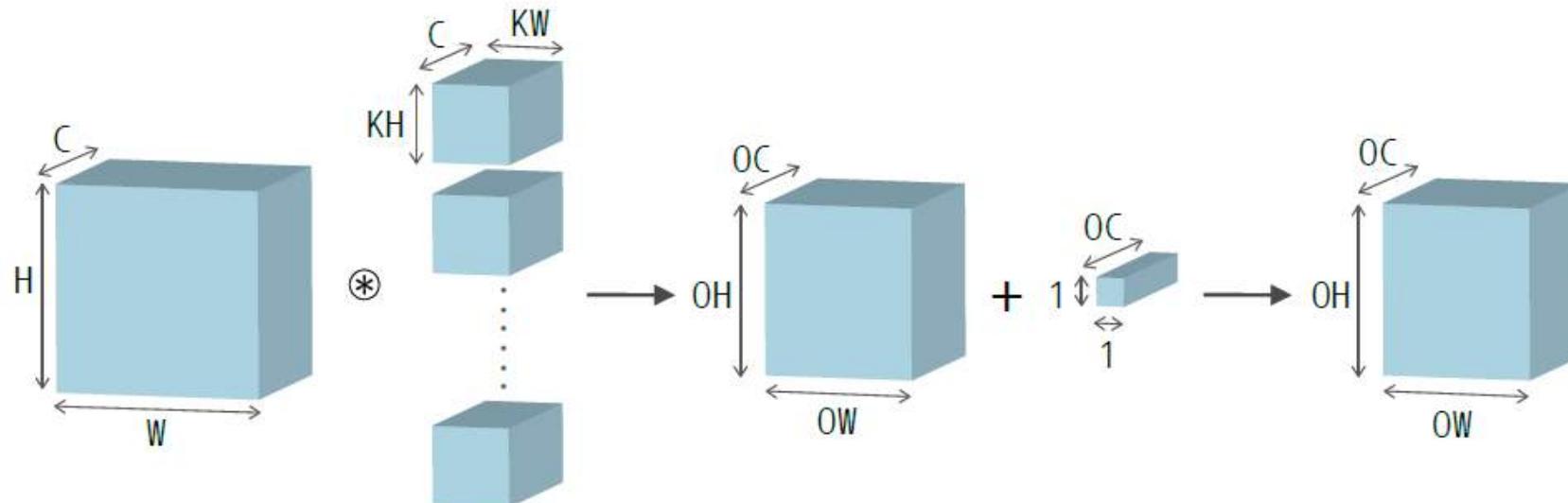


형상	$(C, H, W)$	$\otimes$	$(OC, C, KH, KW)$	$\longrightarrow$	$(OC, OH, OW)$
----	-------------	-----------	-------------------	-------------------	----------------

## 56단계 : CNN 메커니즘(2)

- 편향 추가

그림 56-4 합성곱 연산 처리 흐름(편향 추가)

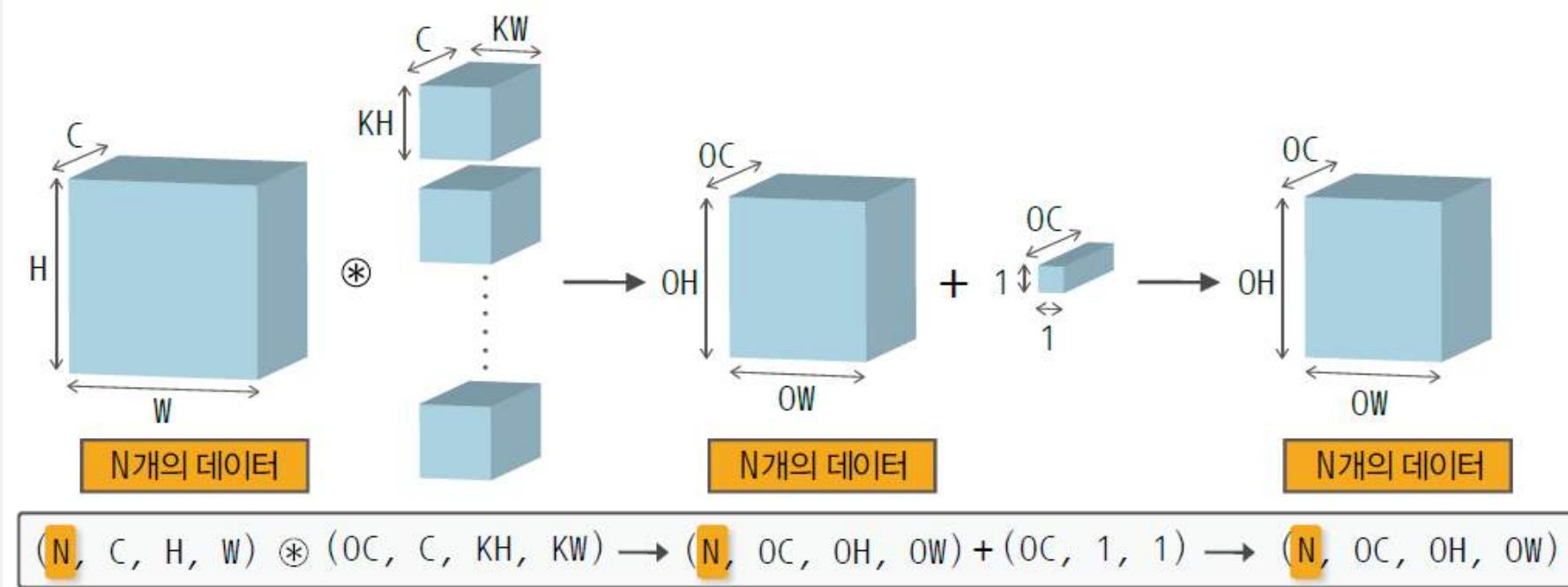


$$(C, H, W) \circledast (OC, C, KH, KW) \rightarrow (OC, OH, OW) + (OC, 1, 1) \rightarrow (OC, OH, OW)$$

## 56단계 : CNN 메커니즘(2)

- 미니배치 처리

그림 56-5 합성곱 연산의 처리 흐름(미니배치 처리)

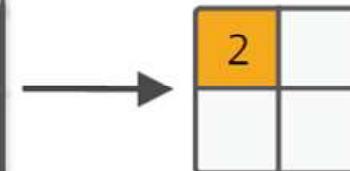


## 56단계 : CNN 메커니즘(2)

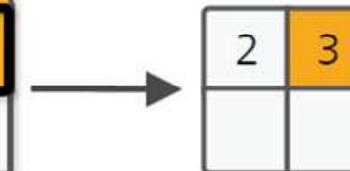
- 풀링층

그림 56-6 Max 풀링의 처리 절차

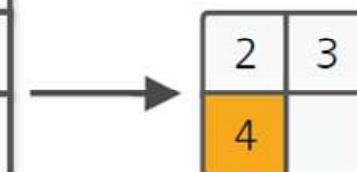
1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



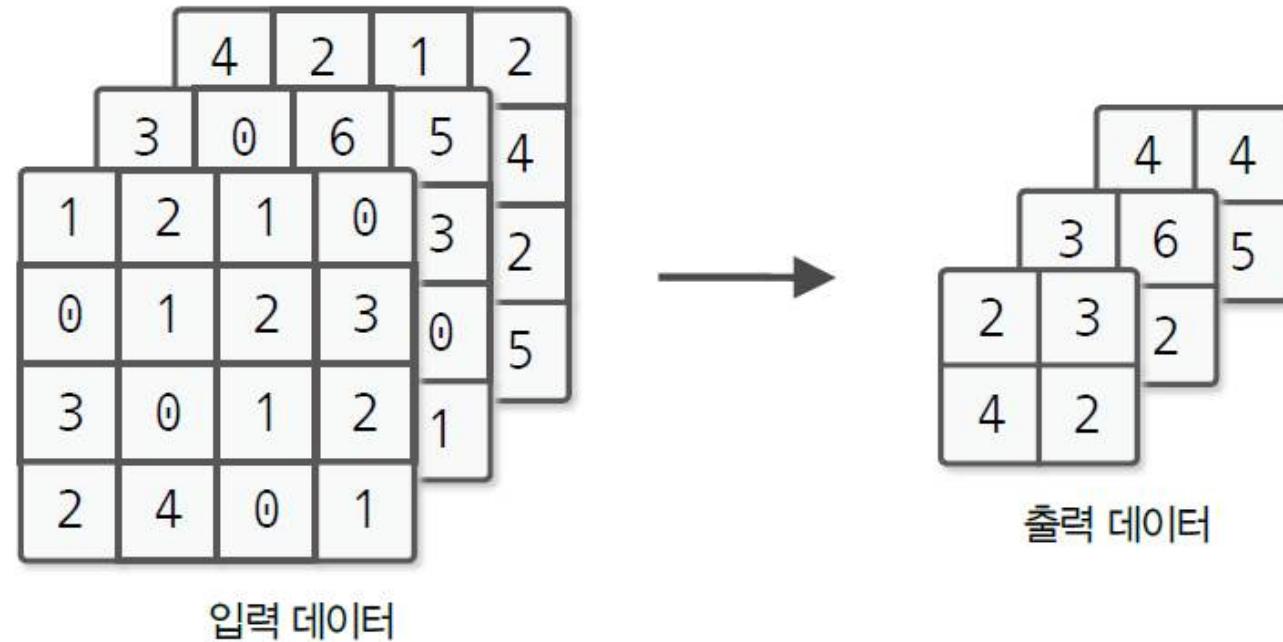
1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



## 56단계 : CNN 메커니즘(2)

- 채널 수

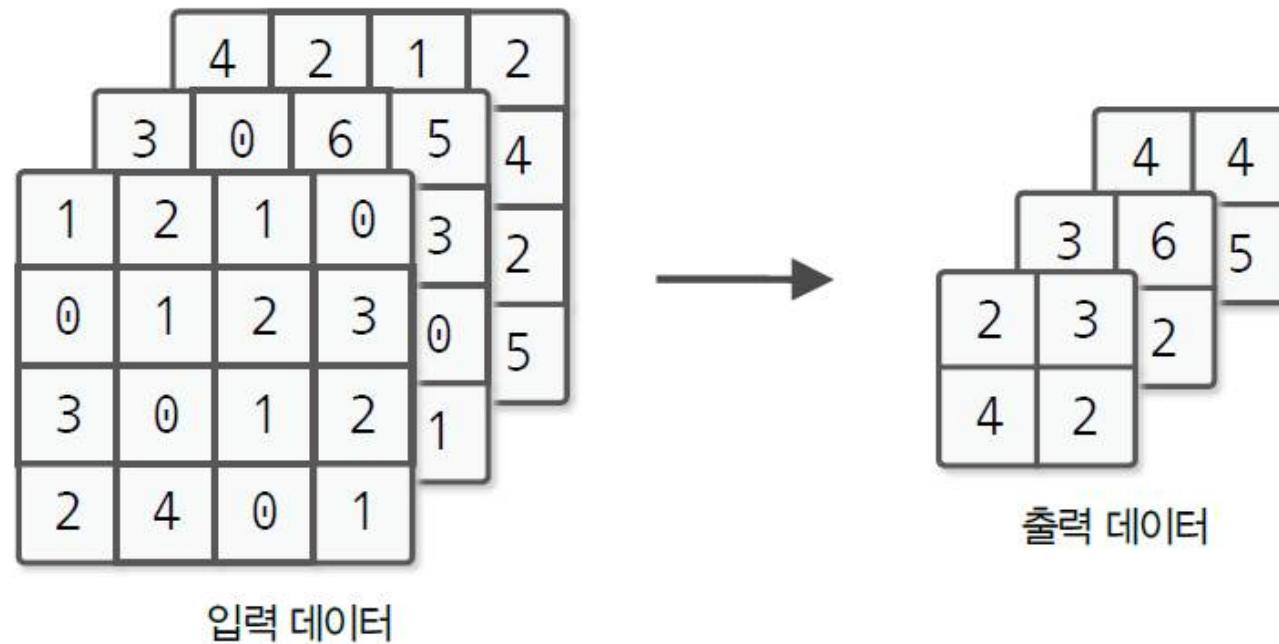
그림 56-7 풀링 시의 채널 수



## 56단계 : CNN 메커니즘(2)

- 특징
- 매개변수 없음
- 채널 수 변화 없음

그림 56-7 풀링 시의 채널 수



## 56단계 : CNN 메커니즘(2)

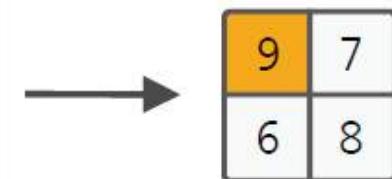
- 미세한 위치 변화 영향 덜 받음

그림 56-8 입력 데이터에 작은 차이가 있을 때의 비교

1	2	0	7	1	0
0	9	2	3	2	3
3	0	1	2	1	2
2	4	0	1	0	1
6	0	1	2	1	2
2	4	0	1	8	1



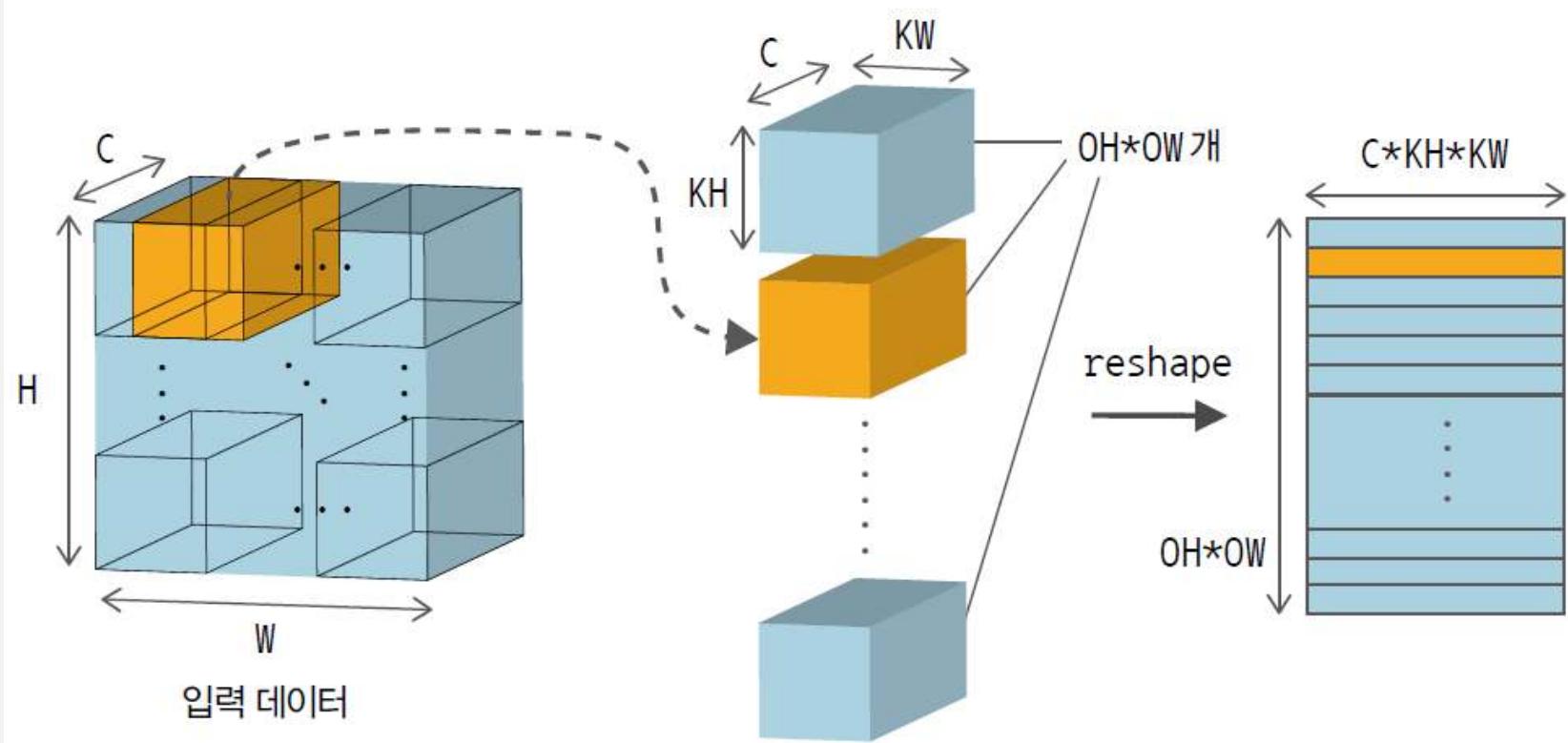
1	1	2	0	7	1
3	0	9	2	3	2
2	3	0	1	2	1
3	2	4	0	1	0
2	6	0	1	2	1
1	2	4	0	1	8



## 57단계 : conv2d 함수 pooling 함수

- Im2col로 전개

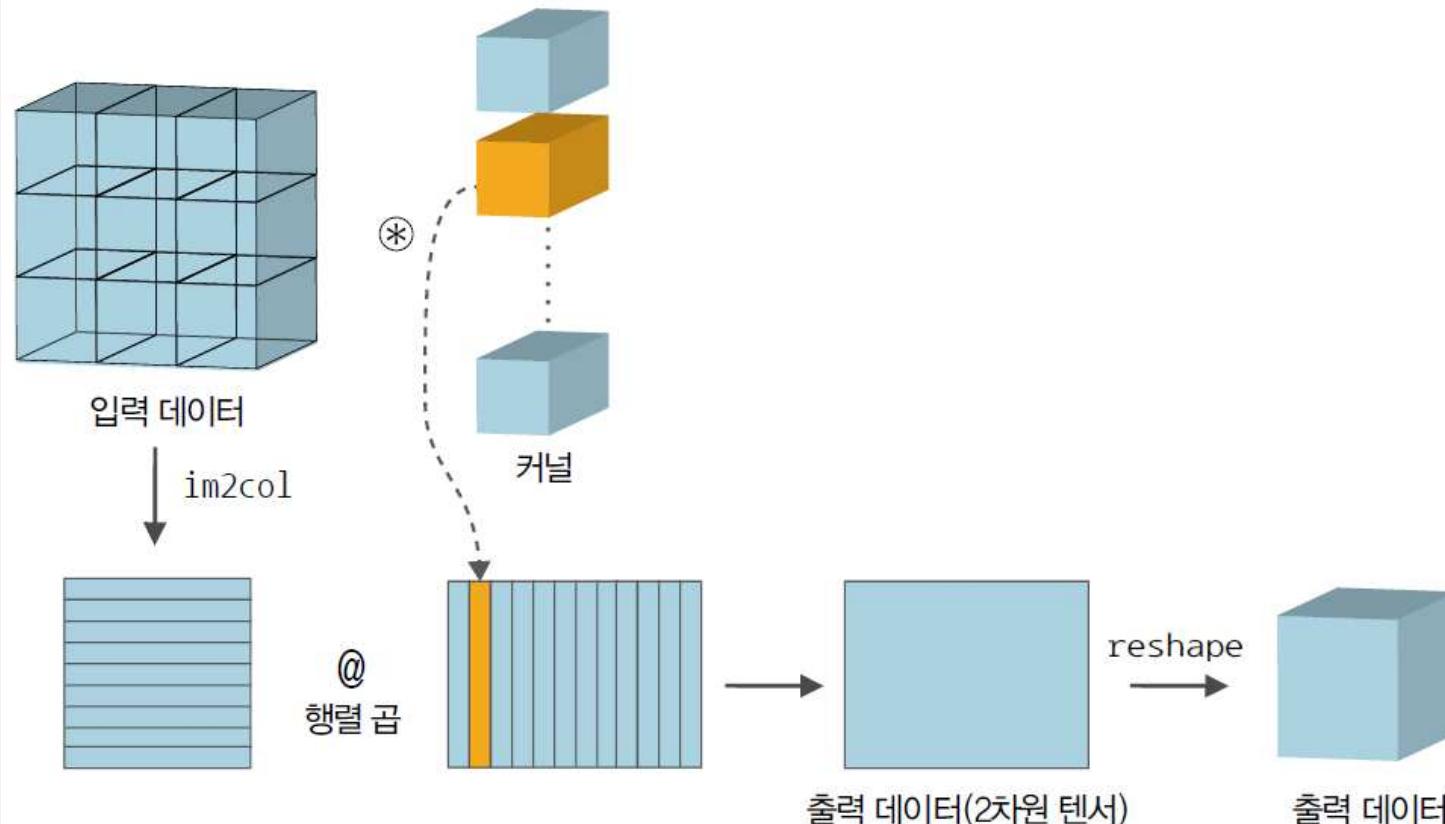
그림 57-1 커널 적용 영역의 전개



## 57단계 : conv2d 함수 pooling 함수

- 행렬 곱 계산

그림 57-2 입력 데이터와 커널의 행렬 곱 계산



## 57단계 : conv2d 함수 pooling 함수

- `Im2col (x, kernel_size, stride=1, pad=0, to_matrix=True)`

**표 57-1** im2col 함수의 인수

인수	데이터 타입	설명
x	Variable 또는 ndarray	입력 데이터
kernel_size	int 또는 (int, int)	커널 크기
stride	int 또는 (int, int)	스트라이드
pad	int 또는 (int, int)	패딩
to_matrix	bool	행렬로 형상 변환 여부

## 57단계 : conv2d 함수 pooling 함수

- Conv2d 함수 구현
- Im2col (x, kernel\_size, stride=1, pad=0, to\_matrix=True)

**표 57-1** im2col 함수의 인수

인수	데이터 타입	설명
x	Variable 또는 ndarray	입력 데이터
kernel_size	int 또는 (int, int)	커널 크기
stride	int 또는 (int, int)	스트라이드
pad	int 또는 (int, int)	패딩
to_matrix	bool	행렬로 형상 변환 여부

## 57단계 : conv2d 함수 pooling 함수

- Conv2d 함수 구현
- Im2col (x, kernel\_size, stride=1, pad=0, to\_matrix=True)

**표 57-1** im2col 함수의 인수

인수	데이터 타입	설명
x	Variable 또는 ndarray	입력 데이터
kernel_size	int 또는 (int, int)	커널 크기
stride	int 또는 (int, int)	스트라이드
pad	int 또는 (int, int)	패딩
to_matrix	bool	행렬로 형상 변환 여부

## 57단계 : conv2d 함수 pooling 함수

- Conv2d 계층 구현

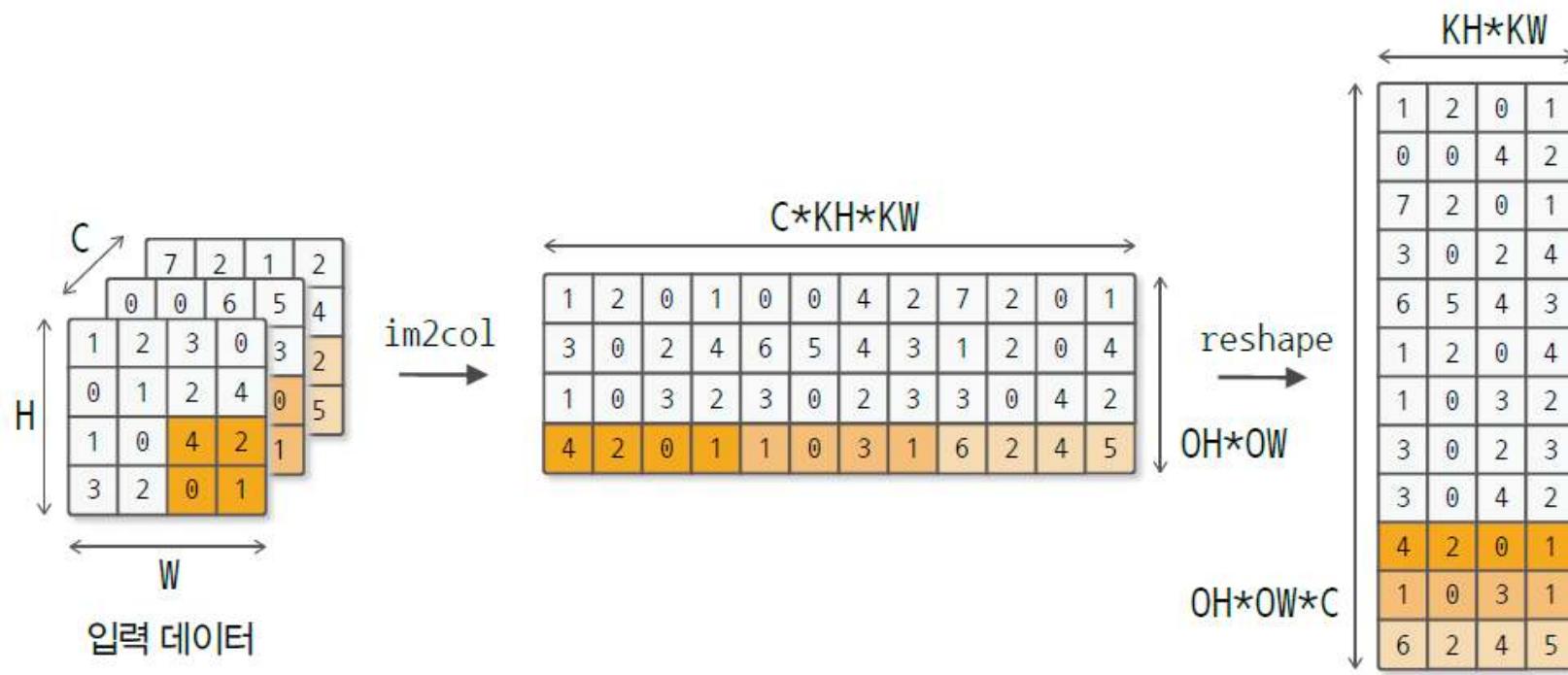
**표 57-2** Conv2d 클래스의 초기화 인수

인수	데이터 타입	설명
out_channels	int	출력 데이터의 채널 수
kernel_size	int 또는 (int, int)	커널 크기
stride	int 또는 (int, int)	스트라이드
pad	int 또는 (int, int)	패딩
nobias	bool	편향 사용 여부
dtype	numpy.dtype	초기화할 가중치의 데이터 타입
in_channels	int 또는 None	입력 데이터의 채널 수

## 57단계 : conv2d 함수 pooling 함수

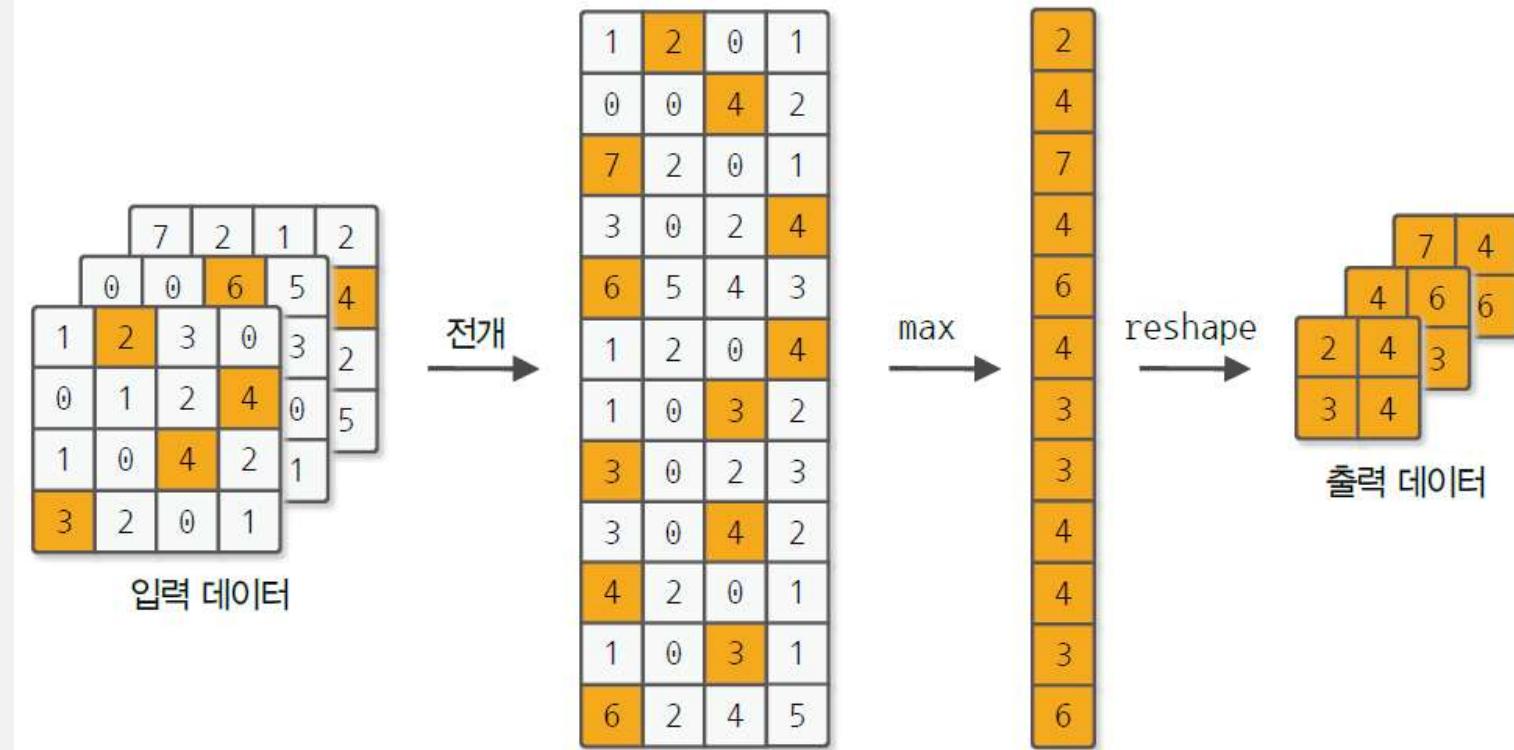
- Pooling 함수 구현

그림 57-4 입력 데이터에 대해 풀링 적용 영역을 전개(2×2 풀링의 예)



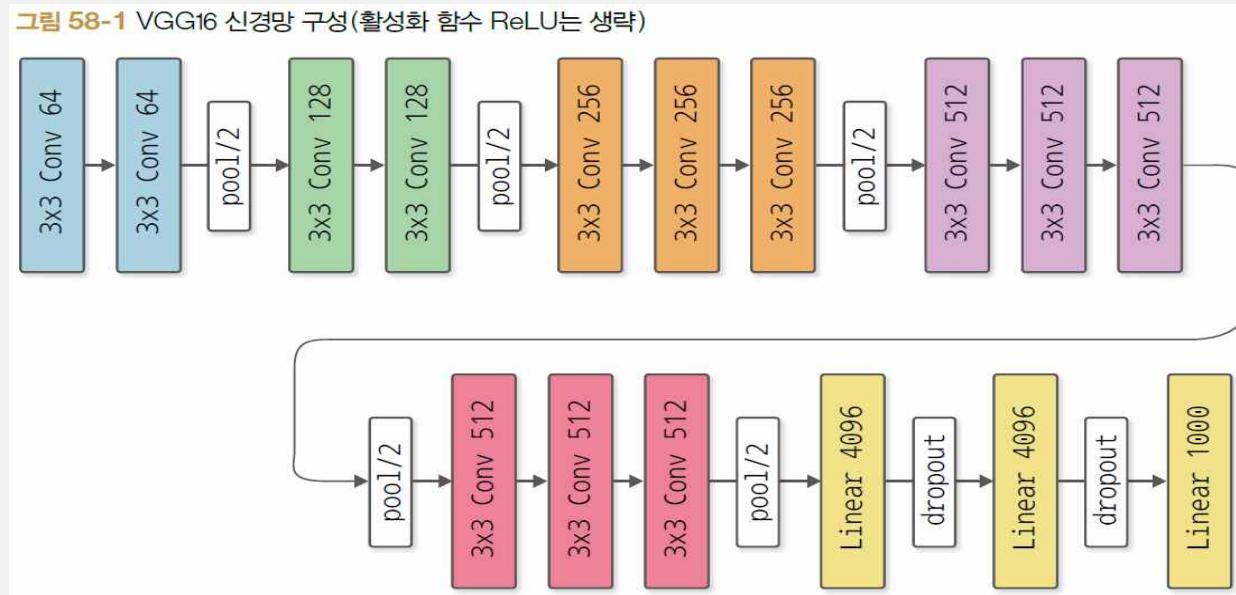
## 57단계 : conv2d 함수 pooling 함수

- Pooling 함수 구현 흐름 **그림 57-5** pooling 함수의 구현 흐름(색칠된 원소 = 풀링 적용 영역에서 값이 가장 큰 원소)



## 58단계 : 대표적인 CNN(VGG16)

- 2014년 ILSVRC 대회에서 준우승한 모델
- 모델의 구성

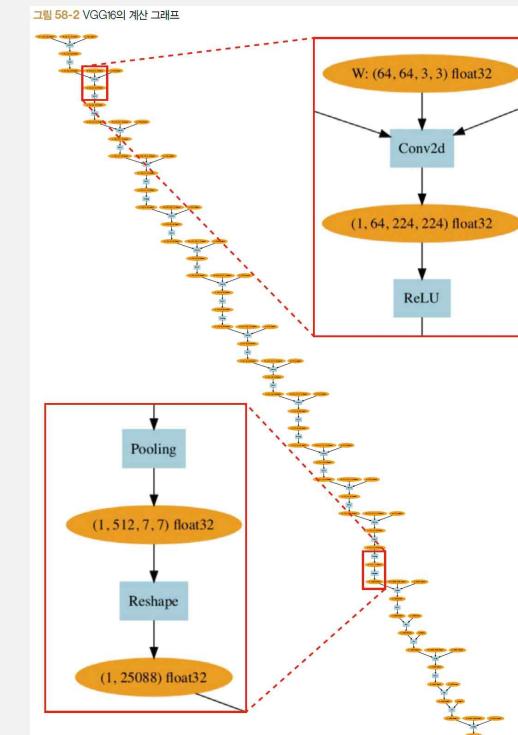


## 58단계 : 대표적인 CNN(VGG16)

- 3x3 합성곱층 사용 (패딩 1x1)
- 합성곱층 채널 수는 풀링하면 2배로 증가
- 완전연결계층에서는 드롭아웃 사용
- 활성화 함수로는 ReLU 사용
- ImageNet으로 학습된 Weights 가 존재

## 58단계 : 대표적인 CNN(VGG16)

- VGG16의 계산 그래프
- 활성화 ReLU 사용
- Conv2d시에는 4차원 텐서를 이용
- 완전연결계층 진입 시 2차원으로 Reshape



## 58단계 : 대표적인 CNN(VGG16)

- 실제 활용
- Preprocessing : (3, 224, 224)
- Predict

```
[9] import numpy as np
from PIL import Image
import dezero
from dezero.models import VGG16

img = Image.open('/content/deep-learning-from-scratch-3/zebra.jpg')
x = VGG16.preprocess(img)
x = x[np.newaxis]

model = VGG16(pretrained=True)
with dezero.test_mode():
    y = model(x)

predict_id = np.argmax(y.data)

model.plot(x, to_file='vgg.pdf')
labels = dezero.datasets.ImageNet.labels()
print(labels[predict_id])

zebra
```



## 59단계 : RNN을 활용한 시계열 데이터 처리

- RNN의 구조
- 순환 구조 (순환 신경망)
- 데이터 입력 시 상태( $h$ )가 갱신  $\rightarrow$  상태가 다음 출력에 영향을 줌
- 따라서 출력 값이 다음 출력 값에 영향을 준다.

그림 59-1 RNN의 구조



## 59단계 : RNN을 활용한 시계열 데이터 처리

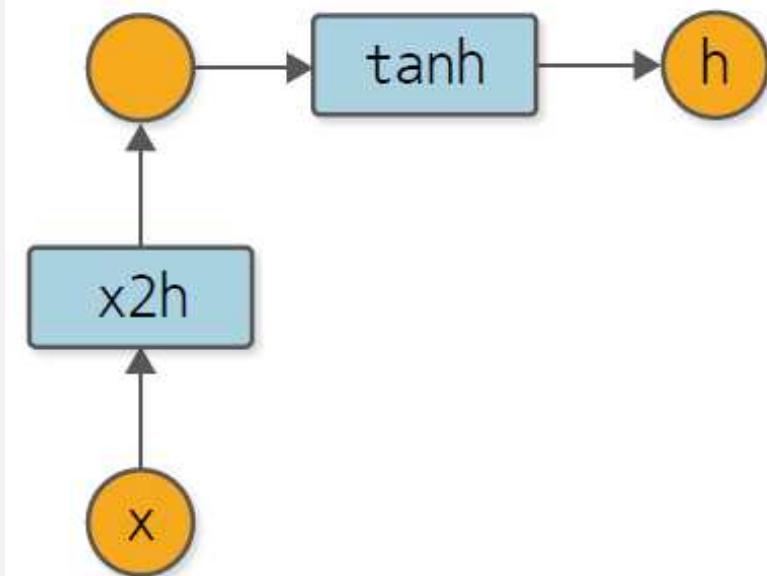
- RNN의 순전파식
- 순환 구조
- $h(t)$  와  $h(t-1)$ 의 연관성이 있다.

$$\mathbf{h}_t = \tanh(\mathbf{h}_{t-1}\mathbf{W}_h + \mathbf{x}_t\mathbf{W}_x + \mathbf{b})$$

[식 59.1]

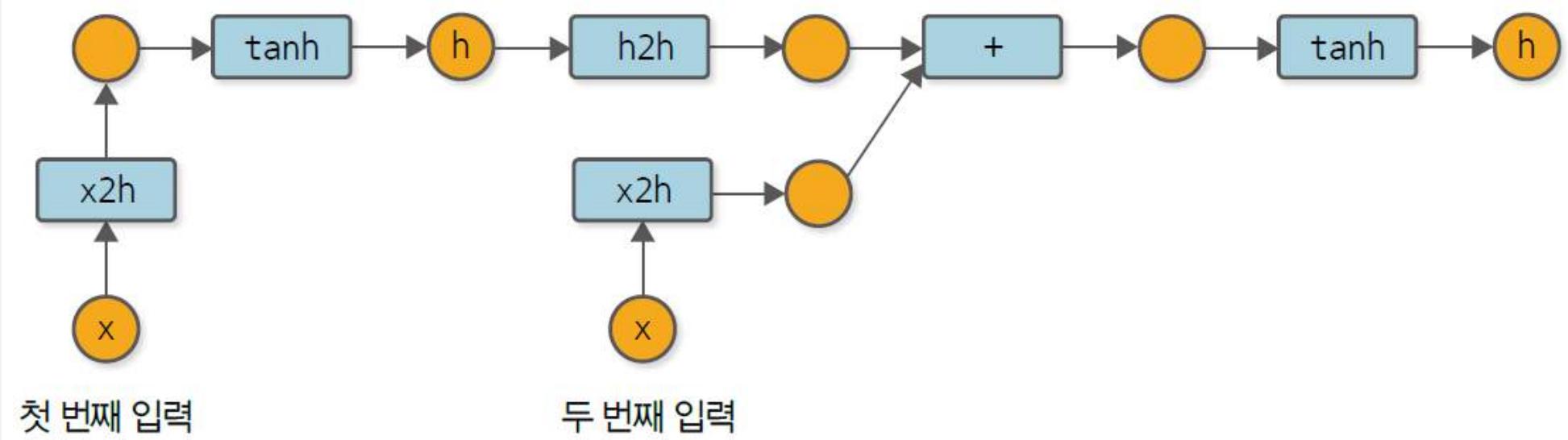
## 59단계 : RNN을 활용한 시계열 데이터 처리

그림 59-2 최초  $x$ 를 주었을 때의 계산 그래프( $x2h$ 는 Linear 계층)



## 59단계 : RNN을 활용한 시계열 데이터 처리

그림 59-3 두 번째 입력 데이터가 처리된 후의 계산 그래프



## 59단계 : RNN을 활용한 시계열 데이터 처리

- RNN의 순전파식
- 순환 구조
- $h(t)$  와  $h(t-1)$ 의 연관성이 있다.

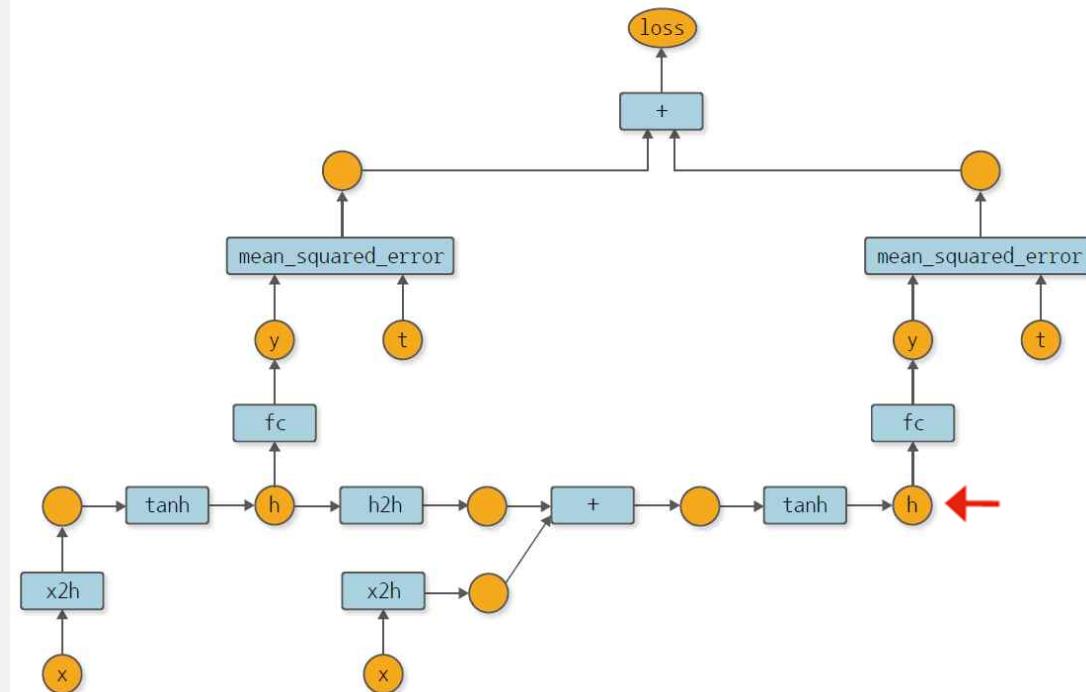
$$\mathbf{h}_t = \tanh(\mathbf{h}_{t-1}\mathbf{W}_h + \mathbf{x}_t\mathbf{W}_x + \mathbf{b})$$

[식 59.1]

## 59단계 : RNN을 활용한 시계열 데이터 처리

- 처음 손실 함수 적용 후의 계산 그래프

그림 59-4 손실 함수 적용 후의 계산 그래프



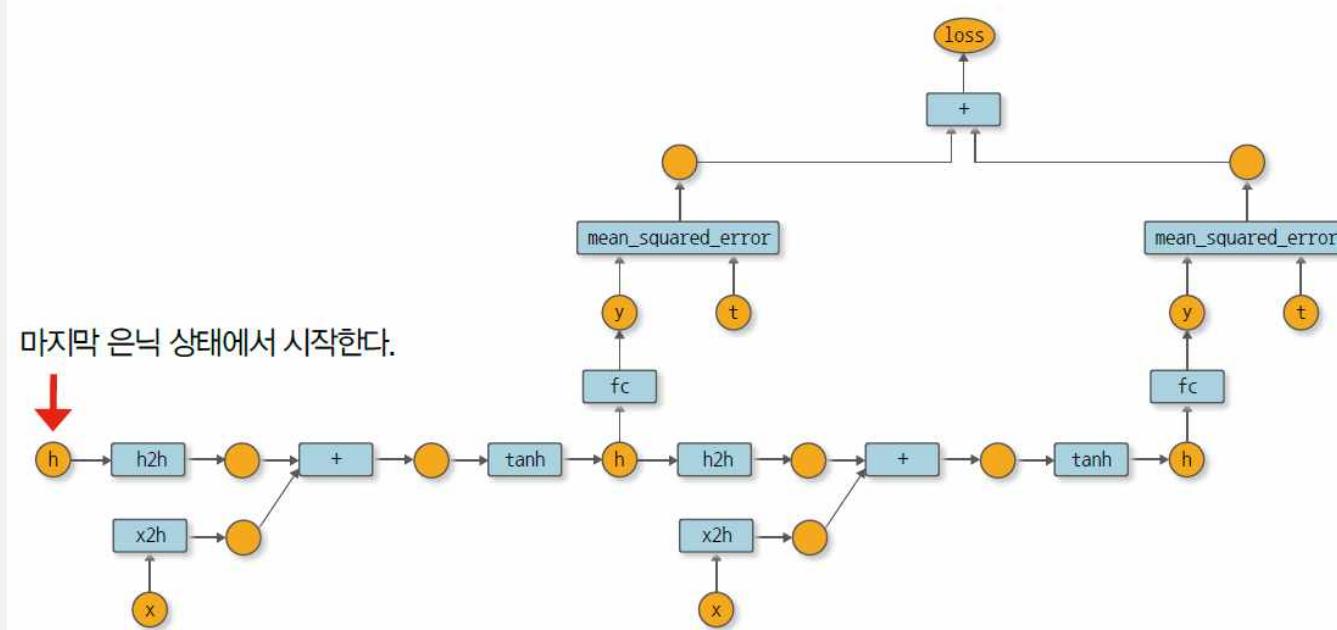
## 59단계 : RNN을 활용한 시계열 데이터 처리

- 이후 반복 시 만들어지는 계산 그래프

그림 59-5 다음 반복에서 만들어지는 계산 그래프

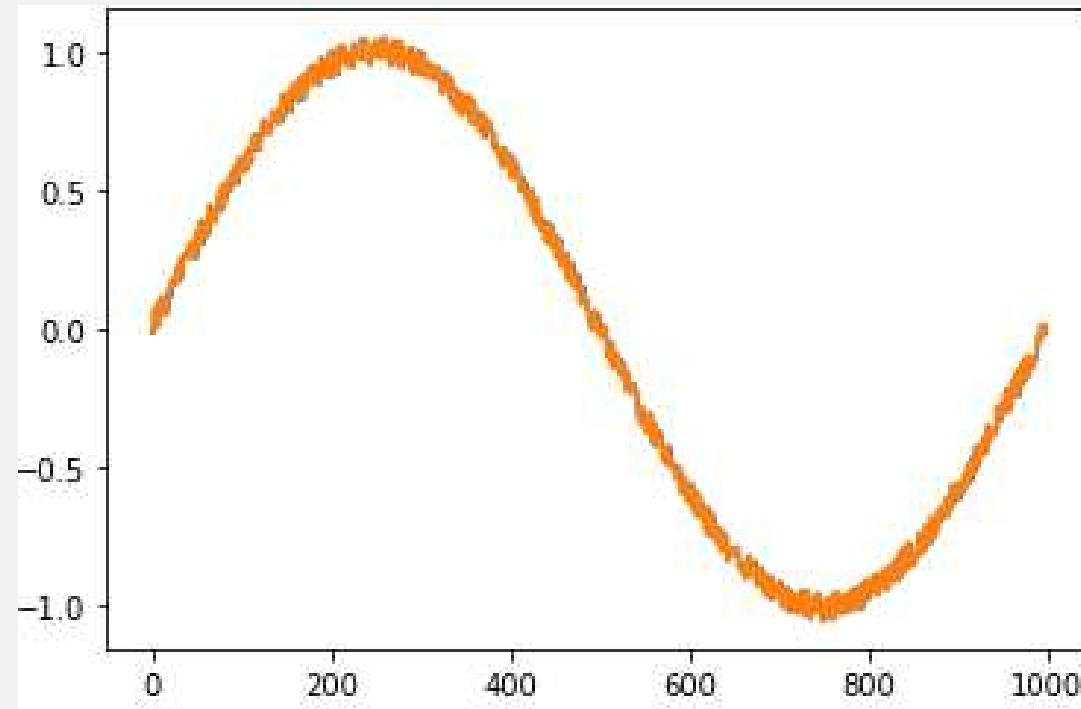
이전  $h$

마지막 은닉 상태에서 시작한다.



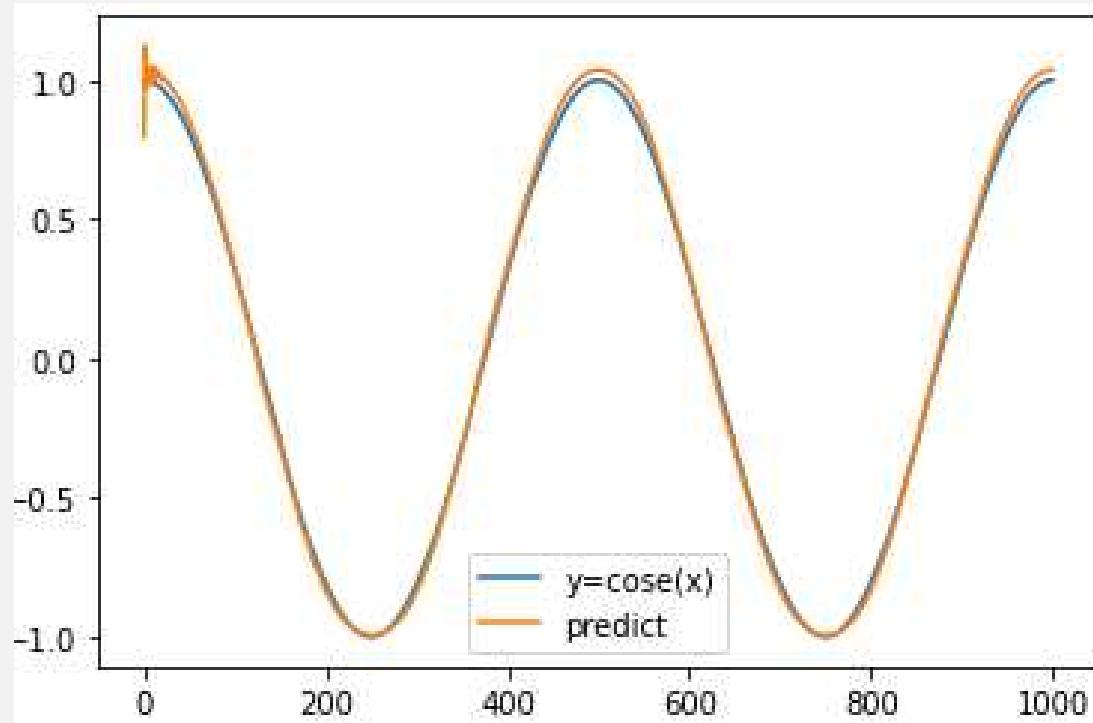
## 59단계 : RNN을 활용한 시계열 데이터 처리

- Sin 예측 실제 데이터 (train\_data)



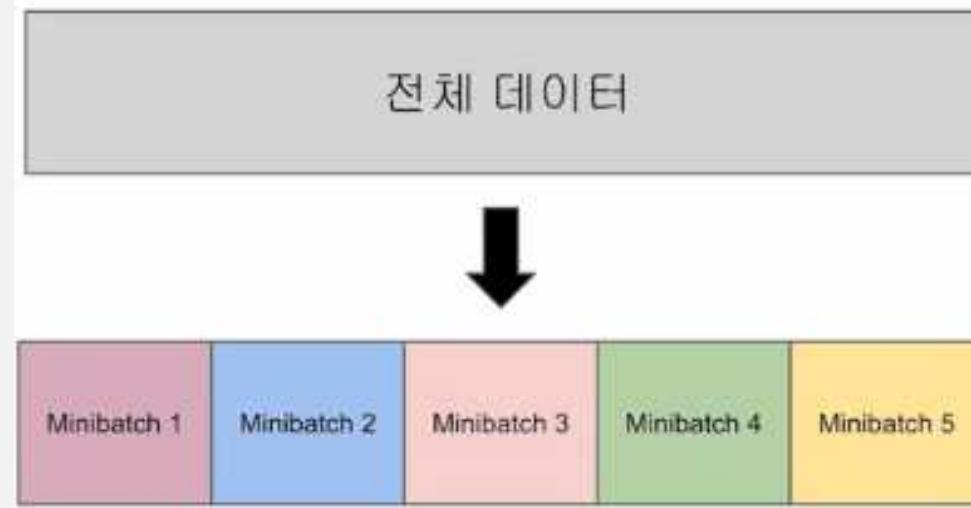
## 59단계 : RNN을 활용한 시계열 데이터 처리

- Sin 예측



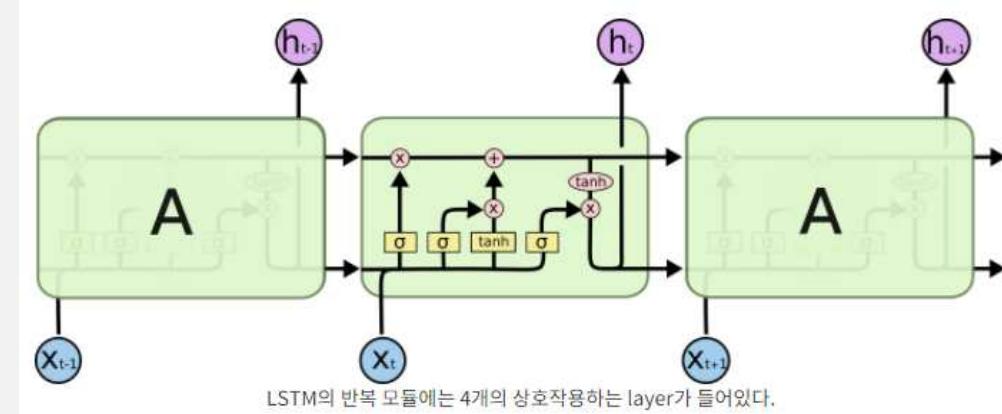
## 60단계 : LSTM과 데이터 로더

- 시계열 데이터용 데이터 로더 만들기
- 데이터를 미니배치로 처리



## 60단계 : LSTM과 데이터 로더

- LSTM
- Long Short-Term Memory
- RNN의 한 종류
- 실제 문제에서 긴 기간의 학습에 제한이 있는 RNN을 보완한 모델



## 60단계 : LSTM과 데이터 로더

- LSTM의 수식

$$\mathbf{f}_t = \sigma(\mathbf{x}_t \mathbf{W}_{\mathbf{x}}^{(f)} + \mathbf{h}_{t-1} \mathbf{W}_{\mathbf{h}}^{(f)} + \mathbf{b}^{(f)})$$

$$\mathbf{i}_t = \sigma(\mathbf{x}_t \mathbf{W}_{\mathbf{x}}^{(i)} + \mathbf{h}_{t-1} \mathbf{W}_{\mathbf{h}}^{(i)} + \mathbf{b}^{(i)})$$

[식 60.1]

$$\mathbf{o}_t = \sigma(\mathbf{x}_t \mathbf{W}_{\mathbf{x}}^{(o)} + \mathbf{h}_{t-1} \mathbf{W}_{\mathbf{h}}^{(o)} + \mathbf{b}^{(o)})$$

$$\mathbf{u}_t = \tanh(\mathbf{x}_t \mathbf{W}_{\mathbf{x}}^{(u)} + \mathbf{h}_{t-1} \mathbf{W}_{\mathbf{h}}^{(u)} + \mathbf{b}^{(u)})$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{u}_t$$

[식 60.2]

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

[식 60.3]

## 60단계 : LSTM과 데이터 로더

- LSTM 계층을 사용한 모델의 예측 결과

