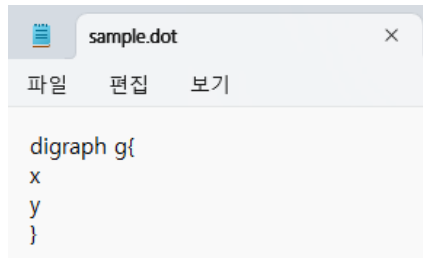


## Ch.05

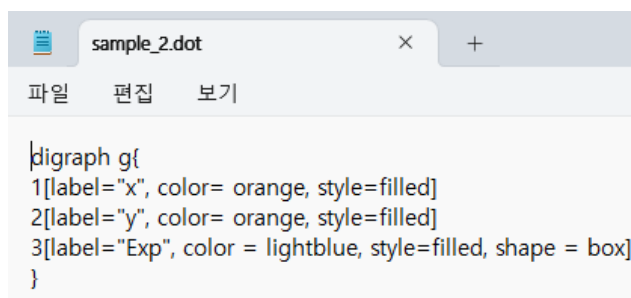
- Dot 언어로 그래프 작성

- 1. Dot 문법



```
digraph g{
x
y
}
```

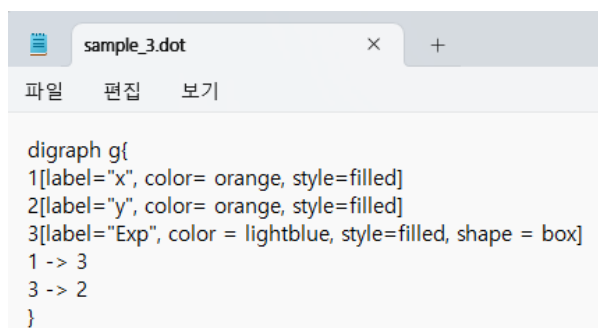
- digraph {.....} : 기본 구조
    - ..... : 그래프 정보
    - 각 노드는 줄 바꿈으로 구분
  - 2. Dot 문법



```
digraph g{
1[label="x", color= orange, style=filled]
2[label="y", color= orange, style=filled]
3[label="Exp", color = lightblue, style=filled, shape = box]
}
```

- 노드 ID: 1과 2같은 숫자
    - [ ]: 노드에 부여할 속성
    - label: 노드 안에 들어갈 문자
    - color: 노드의 색
    - style: 노드 안쪽 색칠 방법,  
ex) style=filled – 노드 안쪽을 채우라는 뜻

- 3. Dot 문법



```
digraph g{
1[label="x", color= orange, style=filled]
2[label="y", color= orange, style=filled]
3[label="Exp", color = lightblue, style=filled, shape = box]
1 -> 3
3 -> 2
}
```

- shape: 그래프의 형태를 지정, 디폴트는 원형(타원형)

- -> : 노드를 연결

ex) 1 -> 2 라고 쓰면 ID가 1인 노드에서 ID가 2인 노드로 화살표가 그려진다.

- 계산 그래프 시각화 함수

- \_dot\_var 함수

```
def _dot_var(v, verbose=False):
    dot_var = '{} [label={}, color=orange, style=filled]\n'

    name = "" if v.name is None else v.name
    if verbose and v.data is not None:
        if v.name is not None:
            name += ": "
        name += str(v.shape) + " " + str(v.dtype)

    return dot_var.format(id(v), name)
```

- Variable 인스턴스를 건네면 인스턴스 내용을 DOT 언어로 작성된 문자열로 바꿔서 변환

- 변수 노드에 고유한 ID를 부여하기 위해 파이썬 내장 함수인 id를 사용

- ✓ ID 연산자

- id 함수는 객체의 고유 값을 리턴

- 파이썬이 객체를 구별하기 위해서 부여한 일련번호, 숫자로서 의미는 없다

```
a = 5
b = 10

print(id(a))
print(id(b))

b = a

print(id(b))
```

```
140706176222120
140706176222280
140706176222120
```

```
from dezero.utils import _dot_var

x = Variable(np.random.randn(2,3))
x.name = 'x'
print(_dot_var(x))
print(_dot_var(x, verbose = True))
```

```
2831465713040 [label="x", color=orange, style=filled]
```

```
2831465713040 [label="x: (2, 3) float64", color=orange, style=filled]
```

- x.name을 'x'로 지정

- \_dot\_var(x)는 verbose가 False이므로 x.name과 id(x)만 추가한 후 출력

- \_dot\_var(x, verbose=True)는 verbose가 True이므로 x.shape와 x.dtype을 추가한 후 출력

## ■ \_dot\_func 함수

```
def _dot_func(f):
    # for function
    dot_func = '{} [label={}, color=lightblue, style=filled, shape=box]\n'
    ret = dot_func.format(id(f), f.__class__.__name__)

    # for edge
    dot_edge = "{} -> {}\n"
    for x in f.inputs:
        ret += dot_edge.format(id(x), id(f))
    for y in f.outputs: # y is weakref
        ret += dot_edge.format(id(f), id(y()))
    return ret
```

## ■ 함수와 입력 변수의 관계, 함수와 출력 변수의 관계도 DOT 언어로 기술함

## ■ Function 클래스를 상속하고, inputs와 outputs라는 인스턴스 변수를 가짐

```
from dezero.utils import _dot_func

x0 = Variable(np.array(1.0))
x1 = Variable(np.array(1.0))
y = x0 + x1

txt = _dot_func(y.creator)
print(txt)
```

```
1562731801232 [label="Add", color=lightblue, style=filled, shape=box]
1562731518736 -> 1562731801232
1562731787088 -> 1562731801232
1562731801232 -> 1562731800976
```

## ■ y.creator: <dezero.core.Add object at 0x00000293402DA350> Add를 가진다.

## ■ f.\_\_class\_\_.\_\_name\_\_: Add

## ■ input: x0, x1

## ■ output: y

## ■ get\_dot\_graph 함수

```
def get_dot_graph(output, verbose=True):
    txt = ""
    funcs = []
    seen_set = set()

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

    add_func(output.creator)
    txt += _dot_var(output, verbose)

    while funcs:
        func = funcs.pop()
        txt += _dot_func(func)
        for x in func.inputs:
            txt += _dot_var(x, verbose)

        if x.creator is not None:
            add_func(x.creator)

    return "digraph g {\n" + txt + "}"
```

-Variable 클래스의 backward 메서드와 거의 같음

-backward 메서드는 미분값을 전파 -> 미분 대신 DOT 언어로 기술한 문자열 txt에 추가

-노드를 추적하는 순서는 필요없어 generation 값으로 정렬하는 코드는 주석 처리

```

from dezero.utils import get_dot_graph

x0 = Variable(np.array(1.0))
x1 = Variable(np.array(1.0))
y = x0 + x1

x0.name = 'x0'
x1.name = 'x1'
y.name = 'y'

txt = get_dot_graph(y, verbose = False)
print(txt)

with open('sample_26.dot', 'w') as o:
    o.write(txt)

```

```

digraph g {
1562731519376 [label="y", color=orange, style=filled]
1562731514128 [label="Add", color=lightblue, style=filled, shape=box]
1562731519696 -> 1562731514128
1562694262032 -> 1562731514128
1562731514128 -> 1562731519376
1562731519696 [label="x0", color=orange, style=filled]
1562694262032 [label="x1", color=orange, style=filled]
}

```

- 출력 변수 y를 기점으로 한 계산 과정을 DOT 언어로 전환한 문자열 반환
- Variable 인스턴스 속성에 name을 추가 : x0.name = "x0"
- Input: x0, x1 | output: y | func: Add | verbose: False
- \_dot\_var(y) -> \_dot\_func(Add) -> \_dot\_var(x0,x1)
- 이미지 변환

```

def plot_dot_graph(output, verbose=True, to_file="graph.png"):
    from IPython.display import Image
    dot_graph = get_dot_graph(output, verbose)
    splitted_text = os.path.splitext(to_file)
    file_name, extension = splitted_text[0], splitted_text[1][1:]

    tmp_dir = os.path.join(os.getcwd(), "graphs")
    if not os.path.exists(tmp_dir):
        os.mkdir(tmp_dir)
    graph_path = os.path.join(tmp_dir, f"{file_name}.dot")
    with open(graph_path, "w") as f:
        f.write(dot_graph)

    cmd = "dot {} -T {} -o {}".format(graph_path, extension, "graphs" + "/" + to_file)
    subprocess.run(cmd, shell=True)

    # Return the image as a Jupyter Image object, to be displayed in-Line.
    try:
        from IPython.display import Image
        to_file = "graphs" + "/" + to_file
        return Image(filename= to_file)
    except:
        pass

```

■ OS 명령어

명령어	기능
os.getcwd	현재 자신의 디렉터리 위치를 리턴한다.
os.listdir	지정된 경로의 디렉토리에 존재하는 파일 혹은 디렉토리를 리스트로 반환
os.mkdir	디렉터를 생성한다.
os.path	폴더명이나 파일명을 확인하고 존재유무를 파악
os.path.split	path 경로명을 (head, tail) 쌍으로 분할합니다. 여기서 tail은 마지막 경로명 구성 요소이고 head는 그 앞에 오는 모든 것
os.path.splitext	확장자를 확인하고 싶을 때 사용 확장전까지의 파일 이름과 확장자를 분리해서 반환
os.path.join	경로와 파일명을 결합하고 싶을 때 사용
os.path.exists	PATH가 존재하면 True를 반환하고, 존재하지 않을 경우 False를 반환한다.

■ subprocess.run()

`subprocess.run(cmd, shell=True)`

- cmd: 쉘 명령어에 쓰일 명령어를 작성( dot 명령어를 쓸 명령어를 작성)

- shell = True: 쉘 화면에 출력을 할 것인가(윈도우 쉘 명령어를 쓰려면 반드시 True 이어야한다.)

● 테일러 급수

■ 어떤 미지의 함수를 동일한 미분계수를 갖는 어떤 다항함수로 근사시키는 것

● sin 함수의 미분을 다른 방법으로 계산(테일러 급수)

$$f(x) = f(a) + f'(a)(x-a) + \frac{1}{2!}f''(a)(x-a)^2 + \frac{1}{3!}f'''(a)(x-a)^3 + \dots$$

■ 1차미분, 2차미분 ... 식으로 항이 무한히 계속, 어느 시점에서 중단하면 f(x)의 값을 근사함

■ 항이 많이 질수록 근사의 정확도가 높음

■ 매클로린 전개

- $a = 0$ 일 때의 테일러 급수

$$f(x) = f(0) + f'(0)x + \frac{1}{2!}f''(0)x^2 + \frac{1}{3!}f'''(0)x^3 + \dots \quad [\text{식 27.2}]$$

- $f(x) = \sin(x)$  를 적용하면,  $f'(x) = \cos(x)$ ,  $f''(x) = -\sin(x)$ ,  $f'''(x) = -\cos(x)$ ...  $\sin(0)=0$ ,  $\cos(0)=1$

$$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!} \quad [\text{식 27.3}]$$

● 함수 최적화

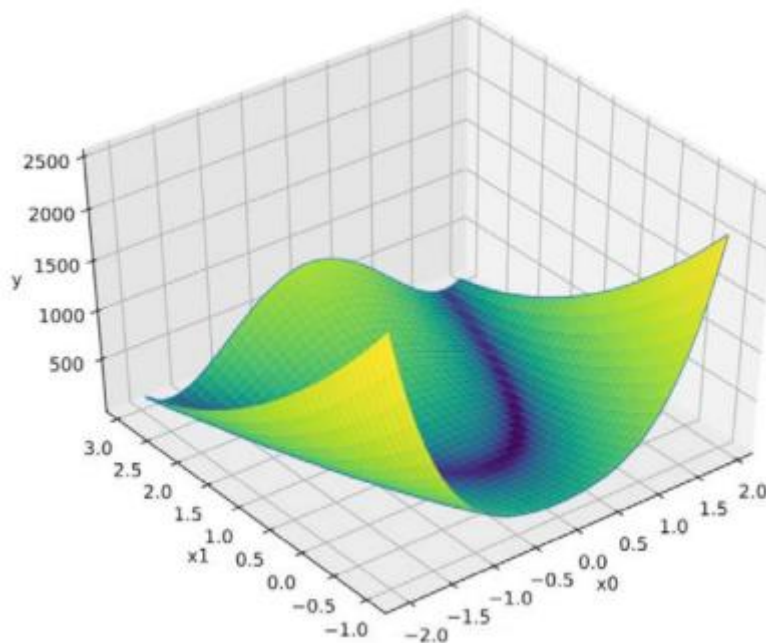
- 최적화란 어떤 함수가 주어졌을 때 그 **최솟값(또는 최댓값)**을 반환하는 **입력(함수의 인수)**을 찾는 일

- 신경망 학습 목표도 **손실 함수의** 출력을 **최소화**하는 **매개변수**를 찾는 것이니 최적화 문제에 속함

■ 로젠브록 함수 최적화

$$y = 100(x_1 - x_0^2)^2 + (1 - x_0)^2$$

$$a, b \text{ 가 정수일 때 } f(x_0, x_1) = b(x_1 - x_0^2)^2 + (a - x_0)^2$$



- 출력이 최소가 되는  $x_0$  와  $x_1$  을 찾는 것

- 경사하강법

- 기울기 방향으로 일정 거리만큼 이동하여 다시 기울기를 구하는 작업을 반복하면 점차 최솟값(혹은 최댓값)에 접근
- 로젠브록 함수의 최솟값 찾기
  - ◆ 기울기 방향에 마이너스를 곱한 방향으로 이동함
  - ◆ `iters`: 반복횟수 | `lr`: 학습률

```
x0 = Variable(np.array(0.0))
x1 = Variable(np.array(2.0))

lr = 0.001
iters = 1000

for i in range(iters):
    print(x0, x1)

    y = rosenbrock(x0, x1)

    x0.cleargrad()
    x1.cleargrad()
    y.backward()
    x0.data -=lr * x0.grad.data
    x1.data -=lr * x1.grad.data
```

```
variable(0.0) variable(2.0)
variable(0.002) variable(1.6)
variable(0.0052759968) variable(1.2800008)
variable(0.009966698110960038) variable(1.0240062072284468)
variable(0.01602875299014943) variable(0.8192248327970044)
variable(0.02324750923068761) variable(0.6554312504220874)
variable(0.031290846214210376) variable(0.5244530896747561)
variable(0.039780241951514035) variable(0.41975829515116514)
variable(0.04835473570612382) variable(0.3361231296508763)
variable(0.05671405943493354) variable(0.26936613981374286)
variable(0.06463840226323121) variable(0.2161362087585121)
variable(0.07198937826156711) variable(0.17374459161623834)
variable(0.07869927242171229) variable(0.14003216740948807)
variable(0.08475507379959696) variable(0.11326444902353183)
variable(0.0901819257435144) variable(0.0920482437257805)
variable(0.09502862514911821) variable(0.07526515092678615)
```

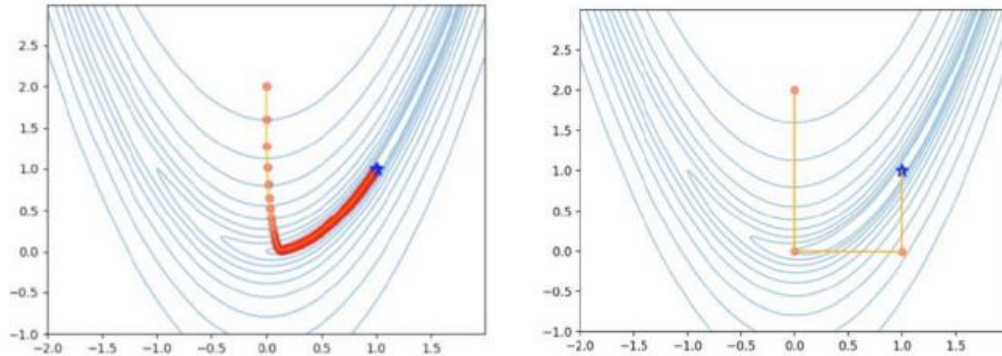
.....

```
variable(0.6819443731033584) variable(0.46352927246942793)
variable(0.6821661743620994) variable(0.46383304357700883)
variable(0.6823877273433749) variable(0.4641365727503715)
variable(0.6826090325042565) variable(0.46443986028606843)
variable(0.6828300903005774) variable(0.46474290648013417)
variable(0.683050901186937) variable(0.4650457116280863)
variable(0.6832714656167057) variable(0.4653482760249264)
variable(0.6834917840420289) variable(0.46565059996514135)
```

- 뉴턴 방법으로 최적화 찾기

- 경사하강법은 일반적으로 수렴이 느리다는 단점
- 뉴턴 방법은 계곡을 뛰어넘어 단번에 목적지에 도착
- 갱신 횟수는 초깃값이나 학습률 등의 설정에 따라 크게 좌우됨

그림 29-1 경사하강법의 갱신 경로(왼쪽)와 뉴턴 방법에 의한 최적화 기법 갱신 경로(오른쪽)



- 2차 미분에서 중단 (2차까지 테일러 급수로 근사)

$$f(x) \simeq f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2$$

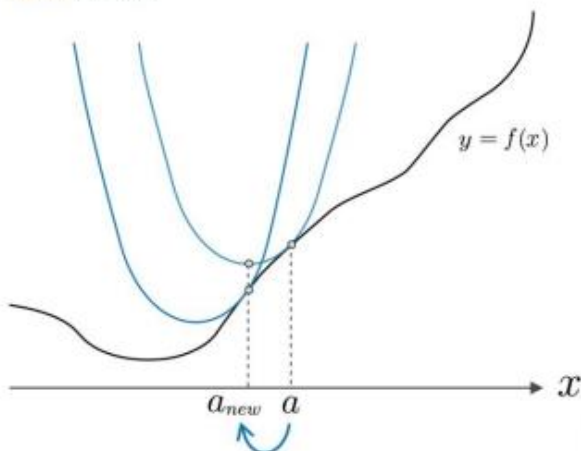
- 2차 함수의 미분 결과가 0인 위치를 확인

$$\frac{d}{dx}(f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2) = 0$$

$$f'(a) + f''(a)(x-a) = 0$$

$$x = a - \frac{f'(a)}{f''(a)}$$

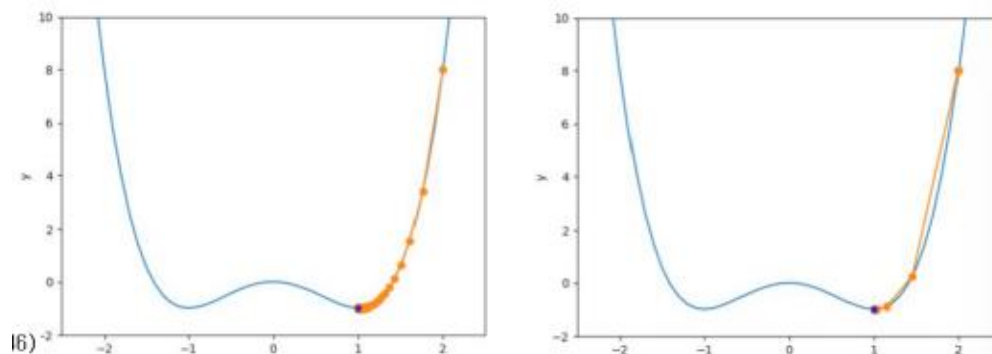
그림 29-3 a의 위치 갱신





## 경사하강법 vs 뉴턴 방법

그림 29-5 경사하강법의 갱신 경로(왼쪽)와 뉴턴 방법을 활용한 최적화 기법의 갱신 경로(오른쪽)

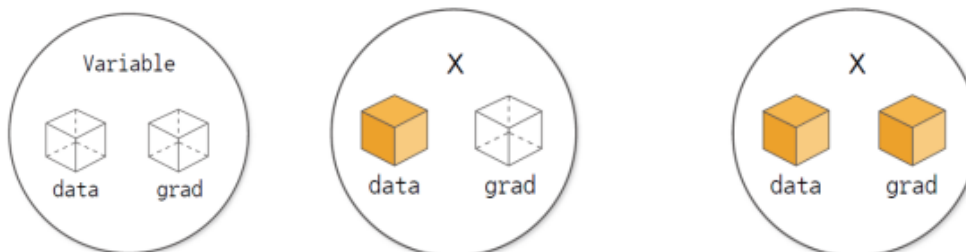


Ch.06

고차미분

- Variable class `__int__` 메서드

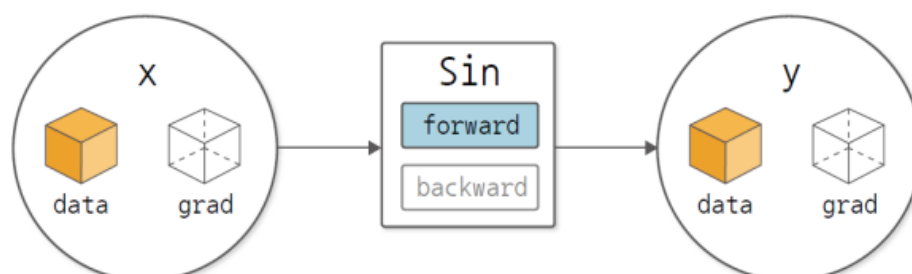
그림 30-1 Variable을 그리는 새로운 방법    그림 30-2 Variable을 그리는 새로운 방법(데이터 참조 시)



- data: 순전파 계산, grad: 역전파 계산
- data, grad: ndarray 인스턴스를 저장

- Function class `__call__` 메서드

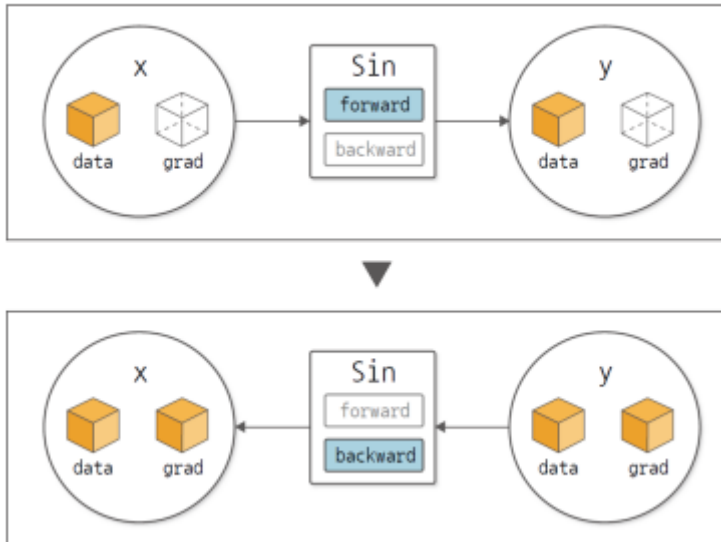
그림 30-3  $y = \sin(x)$ 의 계산 그래프(순전파만)



- 순전파 계산
  - `xs = [x.data for x in inputs]`, `forward(*xs)`
- Variable 과 Function의 '관계'가 만들어짐

- 변수에서 함수로의 연결: set\_creator
  - ➔ 미분값을 역방향으로 진행하게 하기 위함
- Sin 클래스
  - 순전파 계산: sin 클래스의 forward
  - \_\_call\_\_ 메서드: 변수와 함수의 연결
- 역전파 로직

그림 30-4  $y = \sin(x)$ 의 계산 그래프의 순전파와 역전파



- 역전파: Variable class backward
- grad(Variable 인스턴스 변수): 리스트
- backward: grad(ndarray 인스턴스)가 담긴 리스트 전달
- gxs(출력쪽에서 전파하는 미분값): 함수의 입력 변수(f.inputs)의 grad
- 고차 미분 자동 계산

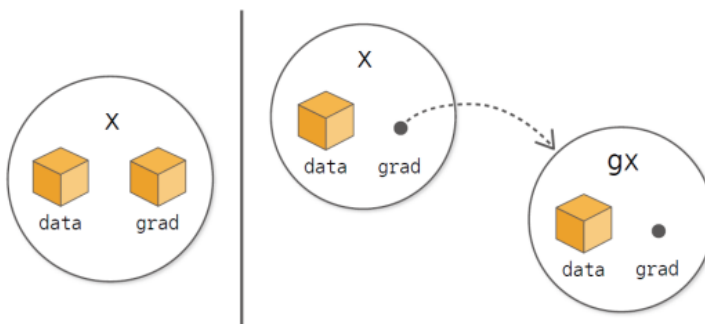
그림 31-2  $y = \sin(x)$ 의 미분을 구하는 계산 그래프( $gx$ 는  $y$ 의  $x$ 에 대한 미분 결과)

그림 31-1  $y = \sin(x)$ 의 계산 그래프



- $gx = (gy * \text{np.cos}(x))$
- 역전파로 계산 그래프 그리기

그림 31-3 지금까지의 Variable 클래스(왼쪽)와 새로운 Variable 클래스(오른쪽)



- 순전파 연결: backward() -> Variable 인스턴스 사용
- 미분값: Variable 인스턴스 형태로 유지

그림 31-4 Sin 클래스의 순전파와 역전파의 계산 그래프(역전파 계산 내용은 생략)

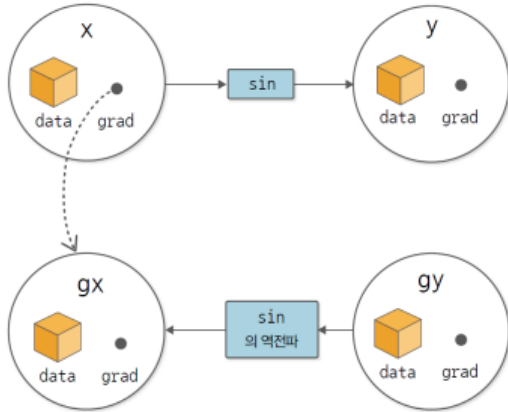
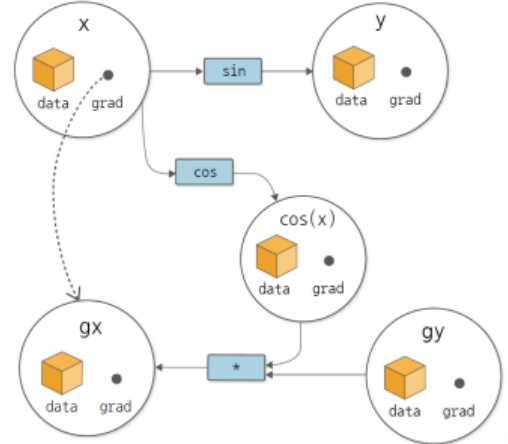


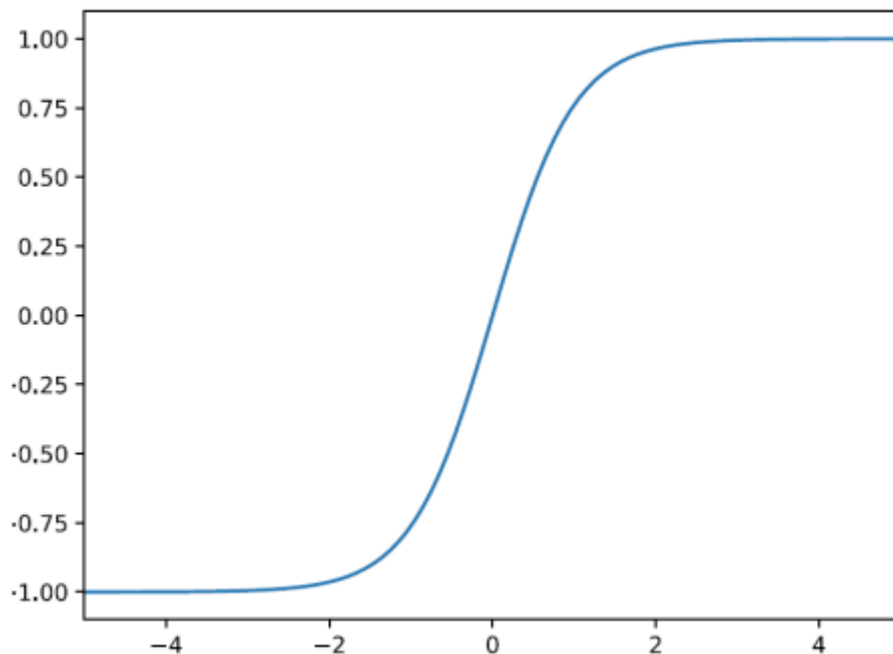
그림 31-5 실제로 만들어지는 계산 그래프



- Variable class grad가 Variable 인스턴스를 참조
- `gx = (gy * np.cos(x))` : 모든 변수가 Variable 인스턴스
- tanh 함수

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

그림 35-1 tanh 함수의 모양



- tanh 함수 미분

$$\left\{ \frac{f(x)}{g(x)} \right\}' = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$$

$$\frac{\partial \tanh(x)}{\partial x} = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$

$$= 1 - \frac{(e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$

$$= 1 - \left\{ \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \right\}^2$$

$$= 1 - \tanh(x)^2$$

$$= 1 - y^2$$

- Double Backpropagation
  - 역전파로 수행한 계산에 대해 또 다시 역전파를 수행