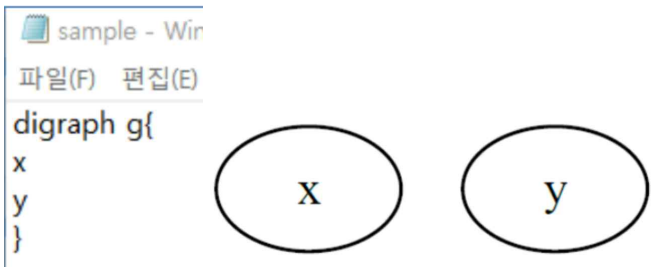
Dezero 패키지 저장

```
import os, sys

sys.path.append("/content/drive/MyDrive/Colab Notebooks/DeZero")
```

계산 그래프 시각화 : Graphviz



오류가 나서 확인해 본 결과

digraph 뒤에 g를 빼면 해결이 된다는 것을 알아냄

버전의 차이인 것 같음(윈도우 10에서는 g를 써야하지만 윈도우 11에서 g를 넣으면 작동이 안됨.)

테일러급수

a(임의의 값)에서 f(x)

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

미지의 함수를 동일한 미분계수를 갖는 어떤 다항함수로 근사시키는 것

1~n차 미분으로 항이 무한함, 어느 시점에서 중단하면 f(x)의 값에 근사함

매클로 전개

임의의 a값이 0일 때의 테일러 급수

sin(x)를 미분

f(x) = sin(x) 를 적용하면, f′(x)= cos(x), f″(x)=-sin(x), f‴(x)=-cos(x)... sin(0)=0, cos(0)=1

$$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

```python
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
```
<- 위 식을 그대로 적어 놓은 코드

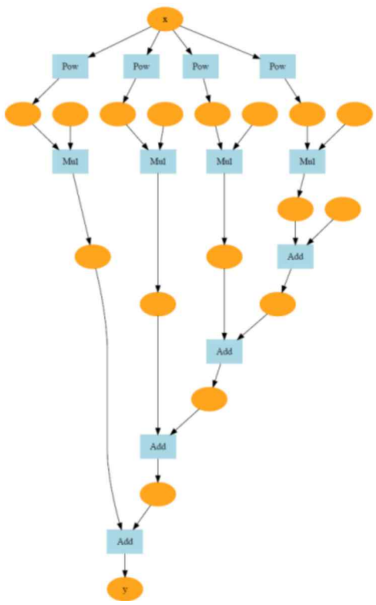
팩토리얼은 math모듈의 math.factorial 함수를 사용

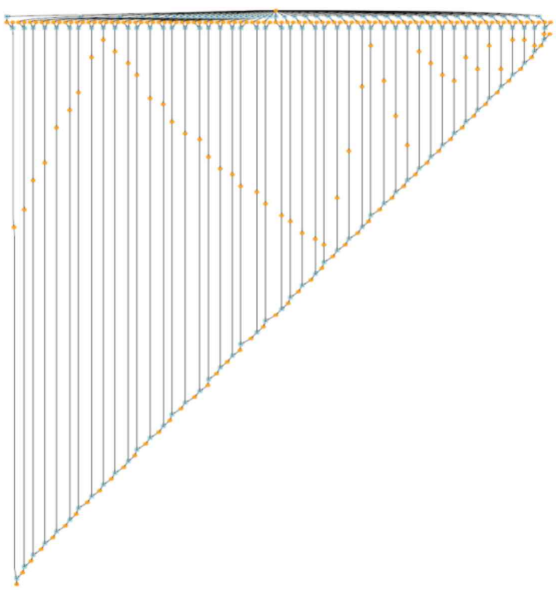
```python
x = Variable(np.array(np.pi / 4))
y = my_sin(x)  # , threshold=1e-150)
y.backward()
print('--- approximate sin ---')
print(y.data)
print(x.grad)
```

```
--- approximate sin ---
0.7071064695751781
0.7071032148228457
```
위에서 구현한 sin 함수와 거의 같은 결과

threshold의 값을 줄이면 오차를 더 줄일 수 있음

threshold = 0.0001일 때 threshold = 1e-150일 때





함수의 최적화

f(x)의 최솟값(또는 최댓값)을 반환하는 입력값(최적의 가중치)을 찾는 일

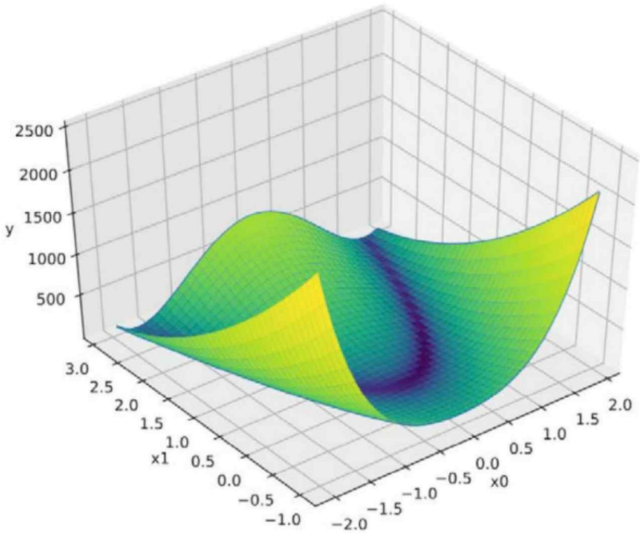
신경망의 경우에도 손실 함수를 최소화 하는 매개변수를 찾는 것

로젠브록 함수의 최적화

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

a, b가 정수일 때 $f(x_0, x_1) = b(x_1 - x_0^2)^2 + (a - x_0)^2$

로젠브록 함수의 형태



a벡터와b벡터의 내적값이 가장 큰 값 찾기

cos함수가 최소가 되는 값을 구하는 과정

출력이 최소가 되는 x0와 x1을 DeZero를 이용하여 찾기

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

위 수식을 적어놓은 코드

경사하강법

기울기 방향으로 일부 이동하여 다시 기울기를 구하는 작업을 반복하면

점차 최솟값에 접근 할 수 있다.(초기값을 잘 설정하면 효율적으로 최솟값에 도달가능)

로젠브록 함수에 적용

기울기 방향에 마이너스를 곱한 방향으로 이동함

```
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
    x1.data -= lr * x1.grad
```

학습률 - 0.001, 반복 - 1000

```
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)
```

출력값을 보면 점차 값이 변화하는 것을 볼 수 있음

반복이 1000번일 때 최솟값에 접근하는 도중 중단

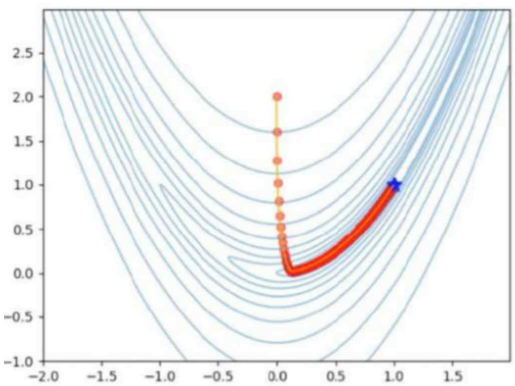
50000번을 했을 때 실제 (1.0, 1.0)에 도달

경사하강법은 로젠브록 함수같이 골짜기가 길게 뻗은 함수에서 학습이 느려짐

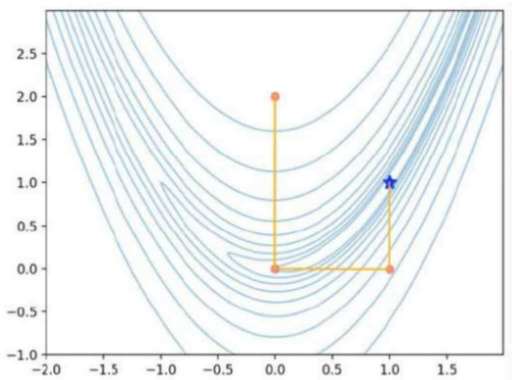
뉴턴방법(학습이 느리다는 것을 보완)

경사하강법은 50000번을 해야했지만, 뉴턴방법은 6번만에 도달.

경사하강법

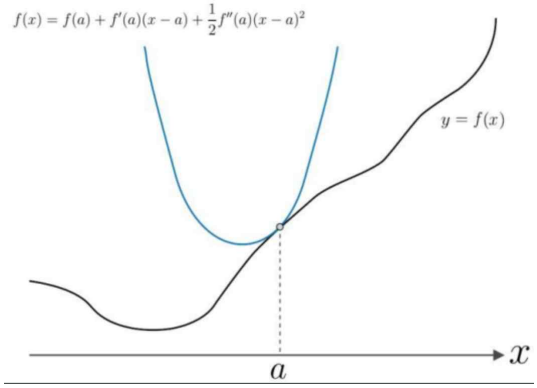


뉴턴방법



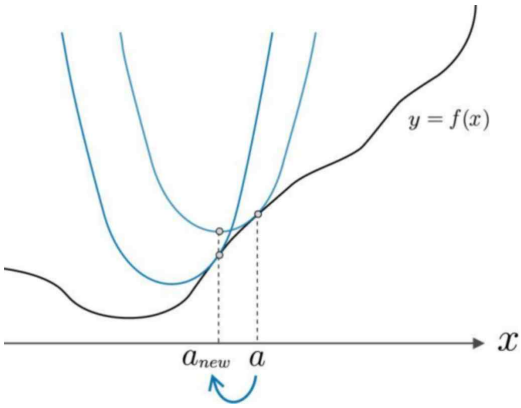
뉴턴방법의 최적화 원리

테일러급수를 y = f(x)라는 함수에 적용



a지점에서 접합함

2차 미분에서 중단(2차까지 테일러 급수로 근사)



새로운 a지점에서 접합함

경사하강법은 속도만을 고려하여 값을 구했지만,

뉴턴방법은 2차미분을 통해 가속도까지 고려하여 값을 구함

```python
import numpy as np
from dezero import Variable

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

def gx2(x):
    return 12 * x ** 2 - 4

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

for i in range(iters):
    print(i, x)

    y = f(x)
    x.cleargrad()
    y.backward()

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

뉴턴방법 코드

```
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)
```

출력값

DeZero는 1차미분만 가능하여 2차미분은 수동으로 작성.

출력값을 보면 6~7번만에 실제값에 도달했다.