5장,6장 정리 노트

산업데이터사이언스학부
201904213
심성빈

5장

DeZero의 연산자 지원

-대표적인 연산자들(+, *, -, /, **)을 지원

최적화 문제의 테스트 함수

-다양한 최적화 기법을 평가하는데 사용되는 함수

-벤치마크용 함수

-세 함수 미분 수행(Sphere, matyas, Goldstein-Price 함수)

Sphere 함수 미분

-함수 수식 : z = x^2 + y^2

-(x,y) = (1.0, 1.0)인 경우, 미분 수행 결과는 (2.0, 2.0)이 되어야함

```
import numpy as np
from dezero import Variable

def sphere(x, y):
    z = x ** 2 + y ** 2
    return z

x = Variable(np.array(1.0))
y = Variable(np.array(1.0))
z = sphere(x, y)
z.backward()
print(x.grad, y.grad)
```

Matyas 함수 미분

-함수 수식 : z = 0.26(x^2 + y^2) – 0.48xy

-(x,y) = (1.0, 1.0)인 경우, 미분 수행 결과는 (0.04, 0.04)이 되어야함

```python
def matyas_fun(x, y):
    z = sub(mul(0.26, add(pow(x,2), pow(y,2))), mul(0.48, mul(x, y)))
    return z
```

```python
def matyas(x, y):
    z = 0.26*(x**2 + y **2) - 0.48*x*y
    return z
```

```python
x = Variable(np.array(1.0))
y = Variable(np.array(1.0))
z = matyas(x, y)
z.backward()
print(x.grad, y.grad)
```

Goldstein -Price 함수 미분

-함수 수식 : f(x,y) = [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)]

-(x,y) = (1.0, 1.0)인 경우, 미분 수행 결과는 (-5376.0, 8064.0)이 되어야함

```python
def goldstein(x, y):
    z = (1 + (x + y + 1)**2 * (19 - 14*x + 3*x**2 - 14*y + 6*x*y + 3*y**2)) * ₩
        (30 + (2*x - 3*y)**2 * (18 - 32*x + 12*x**2 + 48*y - 36*x*y + 27*y**2))
    return z
```

```python
x = Variable(np.array(1.0))
y = Variable(np.array(1.0))
z = goldstein(x, y)
z.backward()
print(x.grad, y.grad)
```

계산 그래프 시각화 필요성

-복잡한 식을 계산할 때 계산 그래프가 만들어지는 전모를 직접 확인

-문제가 발생했을 때 원인이 되는 부분을 파악하기 쉬움

-더 나은 계산 방법을 발견할 수 있음

-신경망의 구조를 3자에게 시각적으로 전달하는 용도로 활용

계산 그래프 시각화 도구

-Graphviz 활용

DOT 언어로 그래프 작성하기

-DOT 언어로 그래프 그림(편집기를 열고 텍스트 입력

-DOT 문법 설명

-> 반드시 digraph g{…} 구조여야 함

-> 각 노드는 줄바꿈으로 구분

-> sample.dot 파일 저장

-DOT 실행

-> Anaconda Prompt > dot sample.dot –T png -o sample.png

-> 실행 후 sample.png 파일이 생김

sample_1 - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
digraph g{
1 [ label="x", color=orange, style=filled ]
2 [ label="y", color=orange, style=filled ]
}
```

DOT 문법 설명

-노드 ID -> '1'과 '2' 같은 숫자로 시작

-해당 ID의 노드에 부여할 속성을 대괄호 [] 안에 기술

-'label'은 노드 안에 들어갈 문자 표시, 'color'는 노드의 색

-'style'노드 안쪽 색칠 방법 지정, 'style = filled'는 노드 안쪽을 채우라는 뜻

-'shape'은 그래프의 형태를 지정. 디폴트는 원형(타원형)

노드 연결 방법

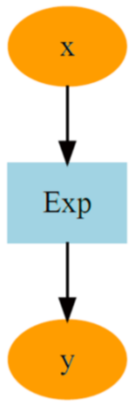
-연결할 두 노드의 ID를 ->로 연결하면 됨

sample_3 - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
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
}
```

그림 25-4 화살표로 연결된 노드



계산 그래프를 시각화하는 함수 구현

-dezero/utils.py에 get_dot_graph 함수 구현

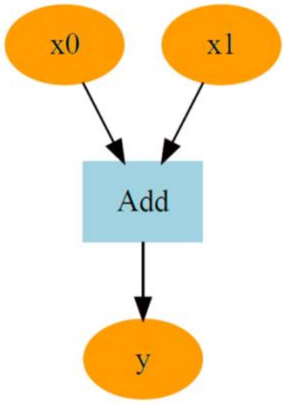
-출력 변수 y를 기점으로 한 계산 과정을 DOT 언어로 전환한 문자열 반환

-변수 노드에 레이블 달기 – Variable 인스턴스 속성에 name을 추가

계산 그래프의 노드를 DOT 언어로 변환

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

#dot 파일로 저장
with open('sample_26.dot', 'w') as o:
    o.write(txt)
```

그림 26-1 시각화된 계산 그래프의 예



➔ 역전파는 출력 변수를 기점으로 역방향으로 모든 노드(변수와 함수) 추적

_dot_var 함수

-get_dot_graph 함수 전용으로 로컬에서만 사용

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

-id 함수에서 반환하는 객체 ID는 다른 객체와 중복되지 않아서 노드의 ID로 사용하기 적합

```python
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)
```

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

➔ Format 메서드 문자열의 "{}" 부분을 인수로 건넨 객체로 차례로 바꿔줌

_dot_func 함수

-get_dot_graph 함수 전용으로 로컬에서만 사용

-DeZero 함수는 Function 클래스를 상속하고, inputs와 outputs라는 인스턴스 변수를 가짐

```python
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
```

```python
x0 = Variable(np.array(1.0))
x1 = Variable(np.array(1.0))
y = x0 + x1
txt = _dot_func(y.creator)
print(txt)
```

get_dot_graph 함수

-Variable 클래스의 backward 메서드와 거의 같음

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

-역전파 노드를 따라가는 순거가 중요하여 함수에 generation 정수값 부여

-노드를 추적하는 순서는 필요X(generation 값으로 정력하는 코드는 주석 처리

```python
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 + '}'
```

Dot 명령 실행까지 한번에 해주는 함수

-get_dot_graph 함수는 계산 그래프를 DOT 언어로 변환

-DOT 언어를 이미지로 변환하려면 dot 명령을 수동으로 실행

-dot 명령 실행까지 한 번에 해주는 함수를 제공

코드 설명

-계산 그래프를 DOT 언어(텍스트)로 변환하고 파일에 저장

-To_file에 저장할 이미지 파일의 이름을 지정

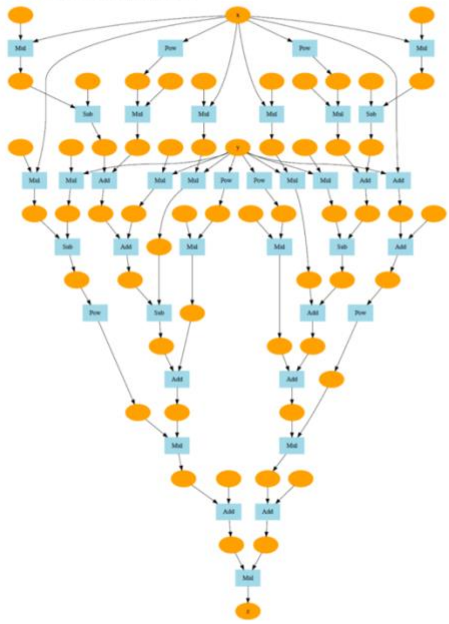
-파이썬에서 외부 프로그램을 호출하기 위해 subprocess.run 함수를 사용

-from dezero.utils import plot_dot_graph로 임포트하여 사용

Goldstein – Price 함수 시각화

```
import numpy as np
from dezero import Variable
from dezero.utils import plot_dot_graph


def goldstein(x, y):
    z = (1 + (x + y + 1)**2 * (19 - 14*x + 3*x**2 - 14*y + 6*x*y + 3*y**2)) * \
        (30 + (2*x - 3*y)**2 * (18 - 32*x + 12*x**2 + 48*y - 36*x*y + 27*y**2))
    return z


x = Variable(np.array(1.0))
y = Variable(np.array(1.0))
z = goldstein(x, y)
z.backward()

x.name = 'x'
y.name = 'y'
z.name = 'z'
plot_dot_graph(z, verbose=False, to_file='goldstein.png')
```

그림 26-3 Goldstein-Price 함수의 계산 그래프



➔ 변수 x와 y에서 시작하여 최종적으로 변수 z가 출력

DeZero을 이용한 구체적인 문제 풀이

-sin 함수의 미분

-sin의 미분은 해석적으로 계산

-sin 함수를 DeZero로 구현하고, 미분을 테일러 급수를 이용해 계산함

테일러 급수

-테일러 급수는 어떤 미지의 함수를 동일한 미분계수를 갖는 어떤 다항함수로 근사키기는 것

-테일로 급수가 필요한 이유는 잘 모르거나 복잡한 함수를 다루기 쉽고 이해하기

쉬운 다항함수로 대체시키기 위함

Sin 함수의 미분

-sin 클래스와 sin 함수 구현은 넘파이가 제공하는 np.sin함수와 np.cos함수를 사용해 구현

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

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

    def backward(self, gy):
        x = self.inputs[0].data
        gx = gy * np.cos(x)
        return gx


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

- **테스트 결과**

```python
x = Variable(np.array(np.pi / 4))
y = sin(x)
y.backward()
print('--- original sin ---')
print(y.data)
print(x.grad)
```

```
--- original sin ---
0.7071067811865476
0.7071067811865476
```

매클로린 전개

-a = 0일 때의 테일러 급수

테일러 급수 식에 따라 sin 함수를 코드로 구현

-팩토리얼 계산은 파이썬의 math 모듈에 있는 math.factorial함수를 사용

-for 문 안에서 i번째에 추가할 항목을 t로 하여구현

-threshold로 근사치의 정밀도를 조정

->threshold를 임곗값으로 지정, threshold가 작을수록 정밀도가 높아짐

->t의 절대값이 threshold 보다 낮아지면 for문을 빠져나오게 함

```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
```

```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
```

➔ 구현한 sin 함수와 거의 같은 결과를 얻음

➔ 오차는 무시할 정도로 작고, threshold값을 줄이면 오차를 더 줄일 수 있음

계산 그래프 시각화

-threshold = 0.0001일 때 my_sin 함수의 계산 그래프

-threshold값으로 계싼 그래프의 복잡성을 제어함

-thershold = 1e – 150으로 설정하여 계산 그래프를 시각화

그림 27-1 threshold = 0.0001일 때 my_sin 함수의 계산 그래프

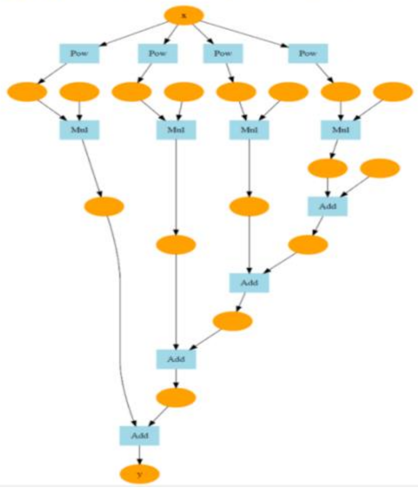
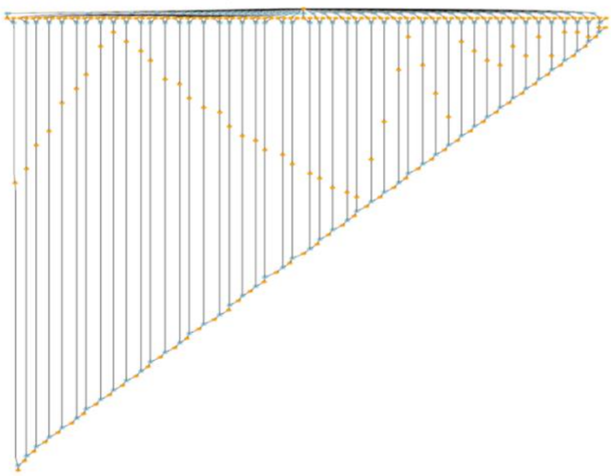


그림 27-2 threshold = 1e-150일 때 my_sin 함수의 계산 그래프



미분의 중요한 용도

-함수 최적화

-구체적인 함수를 대상으로 최적화 계산

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 매개변수를 찾는 것이니 최적화 문제에 속함

로젠브록 함수

-수식은 y = 100(x1 – x0^2)^2 + (1 - x0)^2

-a,b가 정수일 때 f(x0, x1) = b(x1 - x0^2)^2 + (a – x0)^2

-a = 1, b = 100으로 설정하여 벤치마크하는 것이 일반적

형태는 포물선 모양으로 길게 뻗은 골짜기가 보임

로젠브록의 최적화

-출력이 최소가 되는 x0와 x1을 찾는 것임

-최솟값이 되는 지점은 (x0, x1) = (1,1)

DeZero 이용하여 최솟값 찾기

-최솟값 지점을 실제로 찾아낼 수 있는지 확인

로젠브록 함수의 미분

-(x0, x1) = (0.0, 2.0)에서의 미분(x’0와 x’1) 계산

-수치 데이터를 Variable로 감싸서 건네주고 그 다음은 수식을 따라 코딩

-x0과 x1의 미분은 각각 -2.0과 400.0이 나옴

-기울기는 각 지점에서 함수의 출력을 가장 크게 하는 방향을 가르킴

```
import numpy as np
from dezero import Variable

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

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

y = rosenbrock(x0, x1)
y.backward()
print(x0.grad, x1.grad)
```

```
-2.0 400.0
```

경사하강법

-복잡한 형상의 함수라면 기울기가 가리키는 방향에 반드시 최솟값이 존재하지X

-국소적으로 보면 기울기는 함수의 출력을 가장 크게 하는 방향을 나타냄

-좋은 초깃값은 경사하강법을 목적지까지 효율적으로 도달하게 함

로젠브록 함수의 최솟값 찾지

-기울기 방향에 마이너스를 곱한 방향으로 이동함

-iters는 반복횟수, lr은 학습률을 말함

-cleargrad메서드

-> x0.grad, x1.grad는 미분값이 누적되기 때문에 새롭게 미분할 때는 누적된 값을 초기화 해야함

- 코드를 실행해보면 (x0,x1)값이 갱신되는 과정을 볼 수 있음

경사하강법 코드

```python
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
```

로젠브록 함수의 최솟값에 접근 경로

-반복횟수 = 1000일 때 최솟값에 접근하는 도중 멈춤

-반복횟수 = 10000으로 늘려 다시 실행했을 때 최솟값이 더욱 가까워짐

-반복횟수 = 50000으로 설정시에 실제 (1.0, 1.0)위치에 도달

-경사하강법은 로젠브록 함수 같이 골짜기가 길게 뻗은 함수에서 잘 대응 못함

뉴턴 방법 적용

-경사하강법은 일반적으로 수렴이 느리다는 단점이 있음

-뉴턴 방법으로 최적화하면 더 적은 단계로 최적의 결과를 얻을 가능성이 높음

-경사하강법은 계곡에서 서서히 최솟값에 접근해감

-뉴턴 방법은 계곡을 뛰어넘어 단번에 목적지에 도착

-경사하강법은 5만법, 뉴턴은 6회 갱신만에 도달

-갱신 횟수는 초깃값이나 학습률 등의 설정에 따라 크게 좌우됨

-일반적으로 초깃값이 정답에 충분히 가까우면 뉴턴 방법이 더 빨리 수렴함

뉴턴 방법의 최적화 원리

-y = f(x)라는 함수의 최솟값을 구하는 문제

-뉴턴 방법으로 최적화하려면 테일러 급수에 따라 y = f(x)를 변환

-테일러 급수에 따라 어떤 점 a를 기점으로 f를 x의 다항식으로 나타낼 수 있음

-증가하는 걸 어느 시점에서 중단하면 f(x)를 근사적으로 나타낼 수 있음

-2차 미분에서 중단(2차까지 테일러 급수로 근사)

-근사한 2차 함수는 a에서 y = f(x)에 접하는 곡선

근사한 2차 함수의 최솟값 구하기

-2차 함수의 최솟값은 해석적으로 구할 수 있음

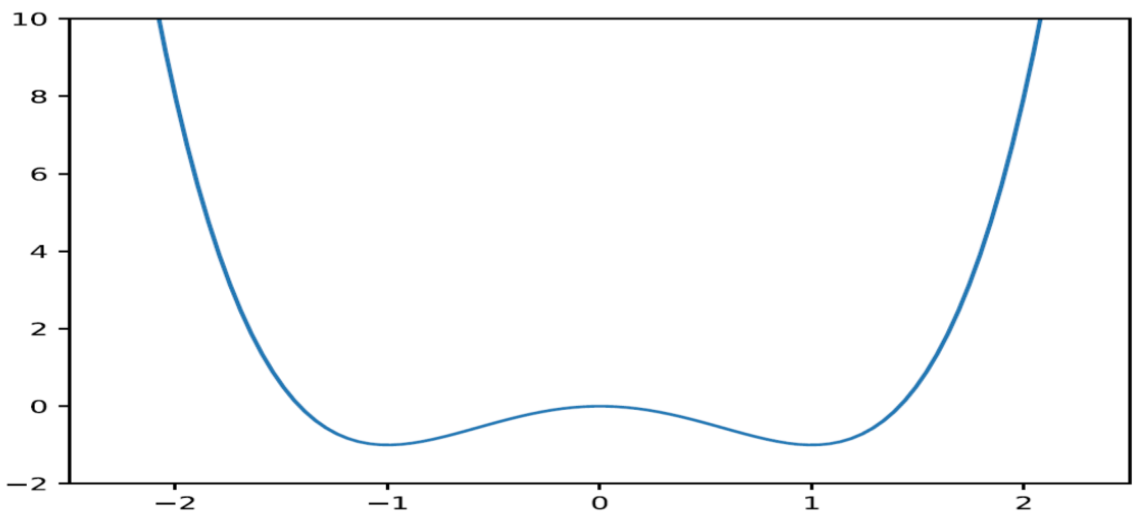
-2차 함수의 미분 결과가 0인 위치를 확인하면 됨
-갱신된 a의 위치에서 같은 작업을 반복함

뉴턴방법 vs 경사하강법
-경사하강법은 알파 계수를 사람이 수동으로 결정 알파의 값만큼 1차 미분 진행하여 x 값을 갱신

뉴턴 방법 정리
-경사하강법은 1차 미분만의 정보를 사용
-뉴턴 방법 최적화는 2차 미분의 정보도 이용
-뉴턴 방법이 추가된 2차 미분으로 효율적으로 탐색을 기대할 수 있음
-목적지에 더 빨리 도달할 확률이 커짐

뉴턴 방법을 이용한 구체적인 문제 풀이
-y = x^4 + -2x^2수식의 최적화
-오목한 부분이 두 곳이며, 최솟값은 x가 각각 -1과 1인 위치
-초깃값을 x = 2로 설정한 후 최솟값 중 하나인 x = 1에 도달하는지 검증

그림 29-4 $y = x^4 - 2x^2$ 의 그래프



뉴턴 방법을 활용한 최적화 구현 코드
-DeZero는 2차 미분을 자동으로 구하지 못하므로 수동으로 2차 미분을 구함
-1차 미분은 역전파로 구하고 2차 미분은 수동으로 코딩해 구함
-뉴턴 방법의 갱신 수식에 따라 x를 갱신
-문제의 답인 최솟값은 1임
-뉴턴 방법은 7회의 갱신 만으로 최솟값에 도달함

```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)
```

```
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)
```

6장

문제의 핵심

-계산 그래프의 연결이 만들어지는 시점으로 순전파를 계산할 때 만들어짐

-역전파를 계산할 때는 연결리 만들어 지지 않음

고차 미분을 자동으로 계산할 수 있는 아이디어

-역전파를 계산할 때도 연결이 만들어 지도록 하면됨

-sin 함수의 미분을 구하기 위한 계산 그래프

-gx.backward()를 호출하여 gx의 x에 대한 미분을 계산할 수 있음

-gx는 y = sin(x)의 미분이기 때문에 gx.backward()를 호출함으로 x의 2차미분에 해당함

순전파 계산의 연결

-Variable인스턴그를 사용하여 순전파를 하는 시점에서 연결이 만들어짐

-backward()메서드에서 ndarray인스턴스가 아닌 Variable인스턴스를 사용하면 계사의 연결이 만들어진다는 뜻

-미분값(기울기)를 Variable인스턴스 형태로 유지해야함

-Variable클래스의 grad는 ndarray인스턴스를 참조하는 대신 Variable인스턴스를 참조하도록 변경

Sin클래스의 순전파와 역전파의 계산 그래프

-Variable 클래스의 grad가 Variable인스턴스를 참조

-미분값을 나타내는 gy가 Variable인스턴스가 된 덕분에 gy를 사용한 계산에도 연

결이 만들어짐
-sin 클래스에서 backward() 메서드 구현시 미분을 계산하는 코드 추가

고차 미분 구현
-역전파 시 수행되는 계싼에 대해서 계산 그래프를 만들면 됨
-역전파 시에도 Variable인스턴스를 사용하면 해결됨

패키지 구조 변경
-지금까지는 Variable클래스를 dezero/core_simple.py에 구현함
-고차 미분을 할 수 있는 새로운 Variable 클래스를 dezero/core.py에 구현
-dezero/core_simple.py에 구현했던 사칙연산등의 함수와 연산자 오버로드를 또한 dezero/core.py에서 구현

새로운 DeZero의 가장 중요한 변화
-Variable 클래스의 인스턴스 변수인 grad임
-기존 grad는 ndarray인스턴스를 참조 했는데, 새로운 클래스에서는 Variable인스턴스를 참조함
-Variable클래스의 소스 변경
->미분값을 자동으로 저장하는 코드에서 self.grad가 Varialbe인스턴스를 담게 됨

```
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))
    ...
```

Backward 메서드 수정
-Function클래스는 수정할 것이 없음
-Add, Mul, Neg, Sub, Div, Pow클래스의 backward메서드 수정

함수 클래스의 역전파 구현
-Add클래스의 역전파가 하는 일은 출력 쪽에서 전해지는 미분값을 입력 쪽으로 전달 (역전파 때는 아무것도 계산하지 않기 때문에 수정할 것이 없음)
-Mil 클래스의 역전파 수정

-> 수정 전에는 Variable인스턴스 안에 있는 데이터를 꺼내야 했음
-> 수정 후에는 Mul클래스에서 Variable인스턴스를 그대로 사용
-> 역전파를 계산하는 gy*x1코드에서 gy와 x1이 Variable인스턴스임
-> gy*x1이 실행이되는 뒤편에서 Mul클래스의 순전파가 호출되면서 그 때 Function.__call__이 호출되고 그 안에서 계산 그래프가 만들어짐

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

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

        return gy * x1, gy * x0
```

역전파의 활성/비활성 모드 도입

-역전파가 필요없는 경우는 역전파 비활성 모드로 전환하여 역전파 처리 생략
-역전파를 1회만 한다면 역전파 계산도 역전파 비활성 모드로 실행하도록 함
-Variable클래스의 backward메서드에 다음 코드 추가

```
def backward(self, retain_grad=False, create_graph=False):
    ...
    while funcs:
            f = funcs.pop()
            gys = [output().grad for output in f.outputs]

            with using_config('enable_backprop', create_graph):
                gxs = f.backward(*gys) # 메인 backward
                if not isinstance(gxs, tuple):
                    gxs = (gxs,)

                for x, gx in zip(f.inputs, gxs):
                    if x.grad is None:
                        x.grad = gx
                    else:
                        x.grad = x.grad + gx

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

- ➔ Create_graph를 추가(기본값을 False로 설정/ 2차미분이 필요하면 True로 설정)
- ➔ 역전파 처리(with using_config(...)에서 수행)

2차 미분 자동 계산

-29 단계까지 2차 미분을 수동으로 계산함
-새로운 DeZero를 사용하여 2차 미분도 자동으로 계산 수행

2차 미분 자동 계산 수행

-간단한 수식의 2차 미분 계산 수행

-뉴턴 방법을 사용하여 최적화 수행

간단한 수식의 2차 미분 수행

-y = x^4 + -2x^2 수식 2차 미분 계산

- y = backward(create_graph = True)의해 첫 번째 역전파 진행하고, 계산 그래프 생성

-gx = x.grad로 y의 x에 대한 미분값을 꺼냄

-gx.backward꺼내 미분값이 gx에 한 번더 역전파 진행 이 두번째 미분이 바로 2차 미분임

-문제를 해결하기 위해 새로운 계산을 하기 전에 Variable의 미분값을 재설정

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

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

x = Variable(np.array(2.0))
y = f(x)
y.backward(create_graph=True)
print(x.grad)

gx = x.grad
gx.backward()
print(x.grad)
```

- 결과값

```
variable(24.0)
variable(68.0)
```

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

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

x = Variable(np.array(2.0))
y = f(x)
y.backward(create_graph=True)
print(x.grad)

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

- 결과값

```
variable(24.0)
variable(44.0)
```

뉴턴 방법을 활용한 최적화

-f(x)의 1차 미분과 2차 미분을 사용하여 x를 갱신

-backward메서드를 두 번 실행하여 자동으로 계산하게 수정

-7회만에 최솟값 1에 도달

```python
import numpy as np
from dezero 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.data -= gx.data / gx2.data
```

- 결과값

```
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)
```

고차 미분에 대응하는 새로운 sin클래스 구현

-backward메서드 안의 모든 변수가 Variable인스턴스임

```python
import numpy as np
from dezero.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)
```

- ➔ Gx = gy*cos(x)에서 cos(x)는 DeZero의 cos함수임
- ➔ Gy*cos(x)에는 곱셈 연산자를 오버로드해 놓았기 때문에 mul함수가 호출

고차 미분에 대응하는 새로운 cos클래스와 cos 함수 구현

-backward 메서드에서 구체적인 계산에서 sin 함수를 사용

```python
import numpy as np
from dezero.core import Function

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)
```

Sin 함수의 고차 미분

-2차 미분뿐만 아니라 3차 미분, 4차 미분도 계산

-for 문을 사용하여 역전파를 반복하여, n차 미분을 구함

-먼저 gx = x.grad에서 미분값을 꺼내 gx에서 역전파하는 것임

-역전파를 하기전에 x.cleargrad()를 호출하여 미분값을 재설정함

```python
import numpy as np
from dezero import Variable
import dezero.functions as F

x = Variable(np.array(1.0))
y = F.sin(x)
y.backward(create_graph=True)

for i in range(3):
    gx = x.grad
    x.cleargrad()
    gx.backward(create_graph=True)
    print(x.grad)
```

- 결과값

```
variable(-0.8414709848078965)
variable(-0.5403023058681398)
variable(0.8414709848078965)
```

➔ 이 작업을 반복하여 n차 미분을 계산

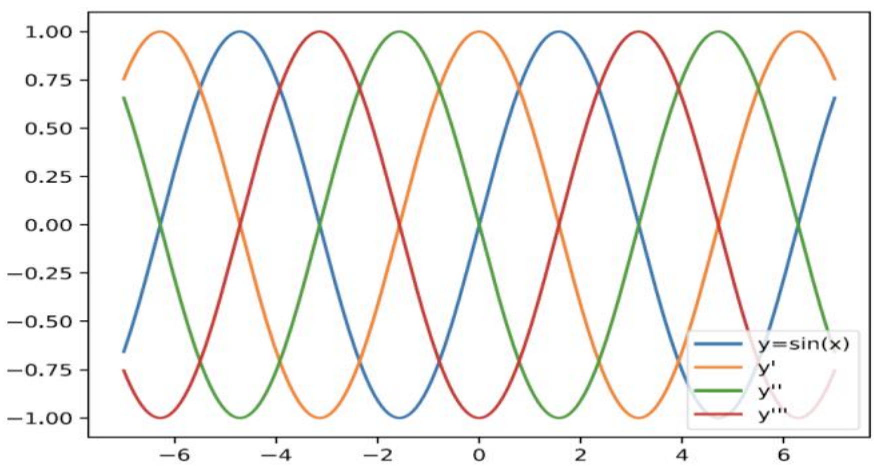
Sin 함수의 고차 미분 그래프 그리기

-다차원 배열을 입력받으면 각 원소에 대해 독립적으로 계산함

-한 번의 순전파로 원소 200개의 계산이 모두 이루어짐

-가가의 그래프는 sin(x)- > cos(x) -> -sin(x) -> -cos(x)식으로 진행이 되어 위상이 어긋남

그림 34-1 y = sin(x)와 고차 미분 그래프(y'는 1차 미분, y''는 2차 미분, y'''는 3차 미분)



Tanh 함수 추가

-tanh는 쌍곡탄젠트 혹은 하이퍼볼릭 탄젠트로 읽음

-tanh 함수는 입력을 -1 ~ 1사이의 값으로 변환

Tanh 함수 미분

-tanh 함수의 미분 공식을 이용해 계산

-분수 함수의 미분 공식을 이용하여 tanh 함수는 다음과 같이 미분할 수 있음

-y = tanh(x)일 때, y' = 1 – y^2

-순전파에서 np.tanh메서드를 이용

-역전파에서는 gy*(1 – y * y)형태로 구현

-재 사용할 수 있도록 dezero/functions.py에 추가

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

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

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

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

Tanh 함수의 고차 미분 계산 그래프 시각화

-for 문에서 반복해서 역전파함으로 고차 미분을 계산

-iters = 0이면 1차 미분, 1이면 2차 미분이 계산되는 방식

```
import numpy as np
from dezero import Variable
from dezero.utils import plot_dot_graph
import dezero.functions as F

x = Variable(np.array(1.0))
y = F.tanh(x)
x.name = 'x'
y.name = 'y'
y.backward(create_graph=True)

iters = 0

for i in range(iters):
    gx = x.grad
    x.cleargrad()
    gx.backward(create_graph=True)

gx = x.grad
gx.name = 'gx' + str(iters + 1)
plot_dot_graph(gx, verbose=False, to_file='tanh.png')
```

고차 미분 계산 정리

-고차 미분을 하기 위해 역전파 시 수행되는 계산에 대해서도 연결을 만들도록 함

-역전파의 계산 그래프를 만들 수 있음

-고차 미분 외에 어떻게 활용할 수 있는지를 살펴봄

Double Backpropagation

-역전파로 수행한 계산에 대해 또 다시 역전파를 수행

-Double backprop은 현대적인 딥러닝 프레임워크 대부분이 지원

Double backprop 활용 용도

-미분이 포함된 식에서 다시 한번 미분 수행

DeZero를 사용하여 문제 계산

-y.backward(create_graph = True)는 미분을 하기 위한 역전파 코드

-역전파가 만들어낸 계산 그래프를 사용하여 새로운 계산을 하고 다시 역전파함

-미분식을 구하고, 그 식을 사용하여 계산 후 또 다시 미분하는 문제

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

x = Variable(np.array(2.0))
y = x ** 2
y.backward(create_graph=True)
gx = x.grad
x.cleargrad()

z = gx ** 3 + y
z.backward()
print(x.grad)
```

DeZero의 역전파를 수정하여 double backprop를 가능하게 됨

느낀점 : 여태까지하면서 고차 미분은 혹시 안되는것인가? 근데 안될 수 없는데 라는 의문이 들고 있었는데 이번 6장을 통해 의문점이 해소 되었고 dezero에서도 시각화가 가능하다는것이 놀랐습니다.