7,8장 정리노트

텐서 사용

-머신러닝 데이터로 벡터나 행렬 드으이 텐서가 주로 사용

-텐서 사용시의 주의점 파악과 DeZero 확장을 준비

-지금까지 구현한 DeZero 함수들이 텐서도 다룰 수 있음을 보여줌

DeZero에서 구현 함수들

-add, mul, div, sin 등등

-입력과 출력이 모두 스칼라라고 가정함

-x는 단일값인 스칼라(0차원의 ndarray 인스턴스)

텐서 처리

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

```
variable([[ 0.84147098  0.90929743  0.14112001]
          [-0.7568025  -0.95892427 -0.2794155 ]])
```

```
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]])
```

➔ X가 텐서일 경우 : sin 함수가 원소별로 적용됨

➔ 입력과 출력 텐서의 형상은 바뀌지 않음((2,3)텐서)

구현한 함수들이 텐서를 아용해 계산해도 역전파 코드가 문제없이 동작

-스칼라를 대상으로 역전파를 구현

-DeZero함수에 텐서를 건네면 텐서의 원소마다 스칼라로 계산

-텐서의 원소별 스칼라 계산이 이루어지면, 스칼라를 가정해 구현한 역전파는 텐서의 원소별 계산에서도 성립

원소 별 계산을 수행하는 DeZero

-텐서를 사용한 계산에도 역전파를 올바르게 해낼 것임을 유추할 수 있음

-sum함수를 사용하면 주어진 텐서에의 모든 원소의 총합을 구해 하나의 스칼라로 출력

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

```
variable(231)
```

➔ 만들어 놓은 게 텐서에도 작동하는지 보는 과정

마지막 출력이 스칼라인 계산 그래프에 대한 역전파

-y.backward(retain_grad = True)를 실행하면 각 변수의 미분값이 구해진다

-기울기의 형상과 순전파 때의 데이터의 형상이 일치

(x.shape == x.grad,shape, c.shape == c.grad.shape)(shape : 형상)

텐서를 사용한 계산에서의 역전파

-원소 별 연산을 수행하는 함수는 입출력 데이터가 스칼라라고 가정, 순전파와 역전파를 구현 가능

-텐서를 입력해도 역전파가 올바르게 성립

원소 별로 계산하지 않는 함수

-텐서의 형상을 변환하는 reshape함수

-행렬을 전치하는 transpose함수

-두 함수 모두 텐서의 형상을 바꾸는 함수

텐서의 형상을 바꾸는 함수

-넘파이의 reshape 함수 사용법

-np.reshape(x, shape)형태로 쓰며 x를 shape인수로 지정한 형상으로 변환

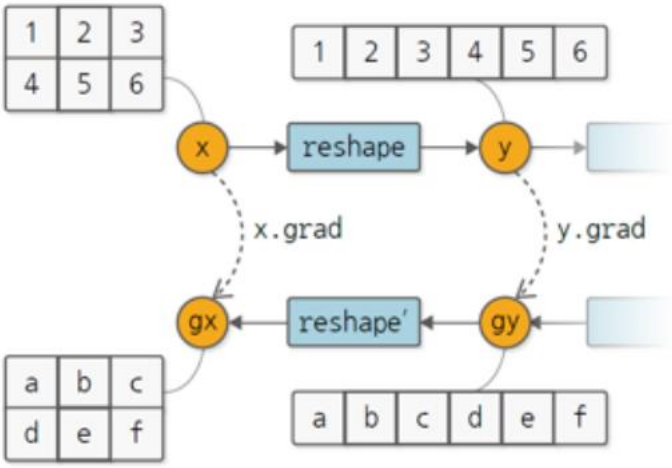
```
x = np.array([[1, 2, 3],[4, 5, 6]])
y = np.reshape(x, (6,))
print(y)
```

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

➔ X의 형상을 (2,3)에서 (6,)으로 변환

➔ 텐서의 원소 수는 같고 형상만 변환

Reshape 역전파 구현



➔ Reshape 함수는 형상만 변환하므로 구체적인 게산 X

➔ 역전파는 출력 쪽에서 전해지는 기울기를 그대로 입력 쪽으로 흘려 보냄

➔ 기울기의 형상이 입력의 형상과 같아지도록 변환

➔ 역전파는 출력 쪽에서부터 기울기를 전달

➔ 기울기를 x.data.shpae와 x.grad.shape가 일치하도록 반환

➔ (6,)인 형상을 (2,3)으로 반환

DeZero용 reshape 함수 구현

-Reshape 클래스를 초기화할 때 변형 목표가 되는 형상을 shape 인수로 받음

-순전파시에 넘파이의 reshape함수를 사용하여 형상을 변환

-self.x_shape = x.shape코드에서 입력 x의 형상을 기억

-역전파에서 입력 형상(self.x_shape)로 변환 가능

-> 역전파할 때 reshape에 기억해 둔거를 바로 사용

Reshape 함수 구현

```
class Reshape(Function):
    def __init__(self, shape):
        self.shape = shape

    def forward(self, x):
        self.x_shape = x.shape
        y = x.reshape(self.shape)
        return y

    def backward(self, gy):
        return reshape(gy, self.x_shape)
```

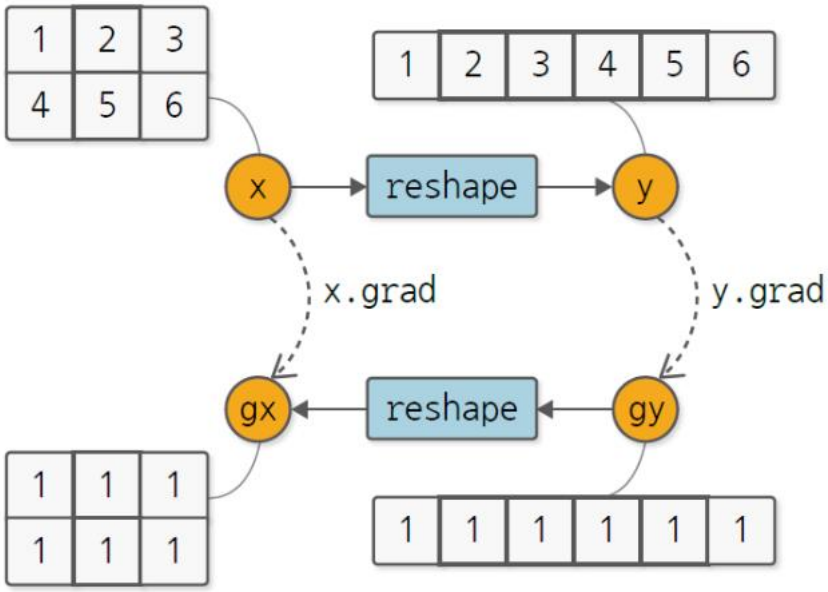
➔ x의 형상 그대로 / 기울기 그대로

```
from dezero.core import as_variable

def reshape(x, shape):
    if x.shape == shape:
        return as_variable(x)
    return Reshape(shape)(x)
```

➔ 인수 x는 ndarray인스턴스 또는 Variable 인스턴스

구현한 reshape 함수 사용



➔ Reshape 함수를 사용하여 형상 변환

➔ Y.backward(retain_grad = True)를 수행하여 x의 기울기를 구함

➔ 이과정에서 y의 기울기도 자동으로 채워짐

➔ 채워진 기울기의 형상은 y와 같음(y.grad.shape == y.shape)

➔ 원소는 모두 1로 이루어진 텐서

DeZero의 reshape 함수를 넘파이의 reshape와 비슷하게 만들기

-Variable 클래스에 가변 인수를 받는 reshape메서드 추가

-reshape함수를 Variable인스턴스의 메서드 형태로 호출 가능

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

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

➔ Class Reshape과 reshape 함수가 저장된 곳 : return dezero.functions

행렬을 전치해주는 함수 구현

-행렬을 전치하면 행렬의 형상이 변함

넘파이의 transpose 함수

-transpose 함수를 사용하여 전치를 할 수 있다

-x의 형상이 (2,3)에서 (3,2)로 변함

-텐서의 원소 자체는 그대로이고 형상만 바뀜

-역전파에서는 출력 쪽에서 전해지는 기울기의 형상만 변경

-순전파 때와 정확히 반대 형태

DeZero의 transpose함수 구현

-역전파에서는 순전파와는 반대의 변환이 이루어짐

```
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)
```

➔ 순전파는 np.transpose함수를 사용하여 전치

➔ 역전파는 출력 쪽에서 전해지는 기울기를 transpose함수를 사용하여 반환

Variable 클래스에 transpose함수 추가

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

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

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

➔ 두개의 메서드 추가

➔ 첫 번째인 transpose는 인스턴스 메서드로 이용하기 위한 코드

➔ 두 번째 T에는 @property 데코레이터가 붙어 인스턴스 변수로 사용

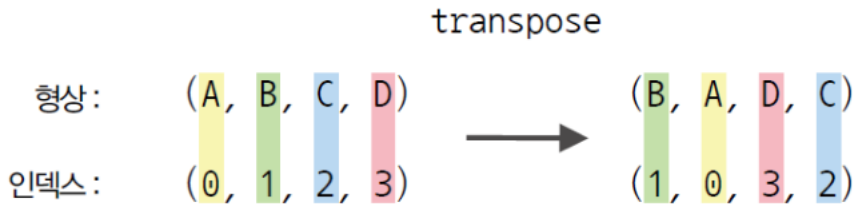
넘파이의 np.transpose함수의 범용적 사용

-축의 순서를 지정하면 그에 맞게 데이터의 축이 달라짐

-인수를 None으로 주면 축이 역순으로 정렬

-x가 행렬일 때 x.transpose()를 실행하면 행렬이 전치됨

(0번 째와 1번 째 축의 데이터가 1번 째와 0번 째 순서로 바뀜)



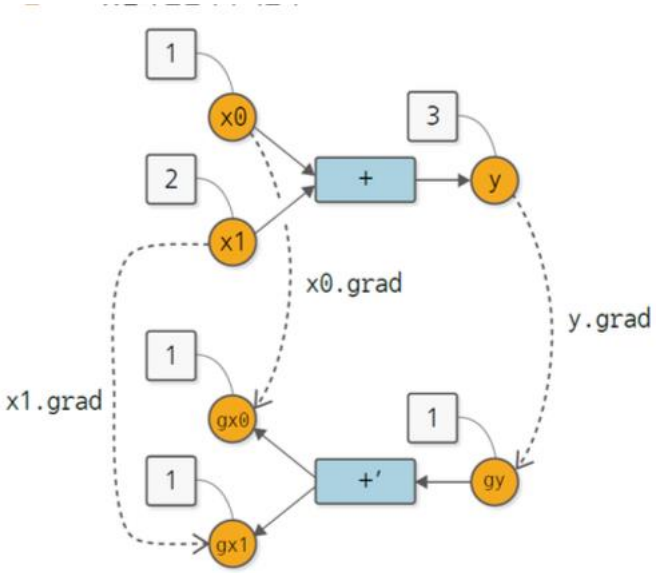
DeZero에 합계를 구하는 함수 sum 추가

-덧셈의 미분

-sum 함수의 미분을 이끌어냄

-sum 함수 구현

덧셈의 미분



➔ 역전파는 출력 쪽에서 전해지는 기울기를 그대로 입력 쪽으로 흘려보냄

➔ 덧셈을 수행한 후 변수 y로 부터 역전파함

➔ X0과 X1에는 출력 쪽에서 전해준 1이라는 기울기를 두 개로 복사하여 전달

원소가 2개로 구성된 벡터 합의 역전파

-벡터에 sum함수를 적용하면 스칼라를 출력

-역전파는 출력 쪽에서 전해준 값인 1을 [1, 1]이라는 벡터로 확장해 전파

원소가 2개 이상인 벡터의 합에 대한 역전파

-기울기 벡터의 원소 수 만큼 복사하면 됨

-기울기를 입력 변수의 형상과 같아지도록 복사

-입력 변수가 2차원 이상의 배열일 때도 동일하게 적용

DeZero의 Sum클래스와 sum함수 구현

-sum함수 역전파에서는 입력 변수의 형상과 같아지도록 기울기의 원소 복사

-지정한 형상에 맞게 원소를 복사하기 위해 broadcast_to함수 사용

-broadcast_to함수를 사용하여 입력 변수와 형상이 같아지도록 기울기 gy의 원소를 복사

행렬을 입력하여 벡터가 아닌 경우의 동작 확인

```python
class Sum(Function):
    def forward(self, x):
        self.x_shape = x.shape
        y = x.sum()
        return y

    def backward(self, gy):
        gx = broadcast_to(gy, self.x_shape)
        return gx

def sum(x):
    return Sum()(x)
```

➔ 순전파일 때는 sum, 역전파일 때는 broadcast(transpose와 다른 점)

Axis(축) 지정 인수

-Axis는 축을 뜻하며, 다차원 배열에서 화살표의 방향을 의미

Keepdims 인수

-keepdims는 입력과 출력의 차원 수(축 수)를 똑같게 유지할지 정하는 플래그

-keepdims = True로 지정하면 축의 수가 유지

-keepdims = False로 지정하면 y의 형상은 스칼라

DeZero의 sum 함수에서 axis와 keepdims 인수 지원

-sum 함수의 역전파에 적용되는 이론은 동일함

-입력 변수와 형상이 같아지도록 기울기의 원소를 복사

-Sum 클래스를 초기화할 때 axis와 keepdims를 입력 받아 속성으로 설정

-순전파에서 이 속성들을 사용해 합계를 구함

-역전파시 broadcast_to 함수를 사용하여 입력 변수의 형상과 같아지도록 기울기 원소 복사

-순전파 sum, 역전파 broadcast -> 복원

넘파이 브로드캐스트

-서로 다른 형상을 가진 배열들간에 산술 연산을 수행하기 위해 배열의 형상을 조정

-브로드캐스트를 사용하여 작은 배열을 큰 배열에 맞추어 연산을 수행

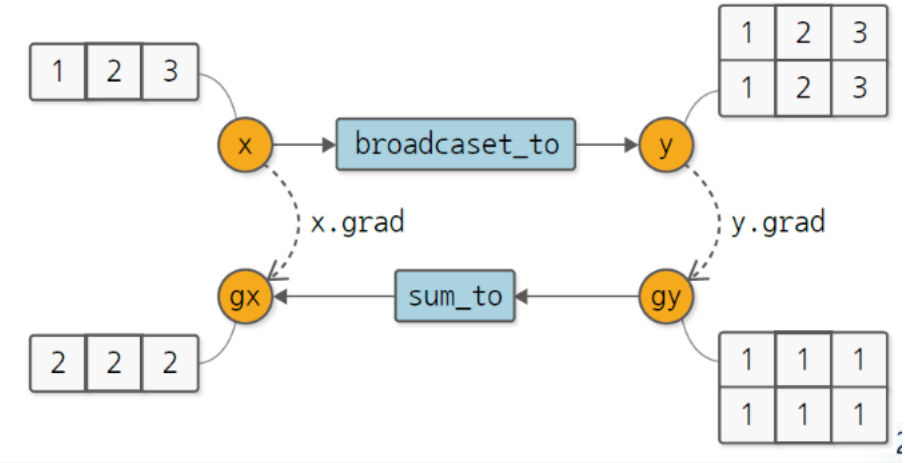
DeZero에서도 넘파이와 같은 브로드캐스트 지원

-sum 함수를 구현시 역전파에서 구현되지 않은 broadcast_to 함수를 이용
-broadcast_to 함수를 구현

넘파이의 브로드케스트 함수

-넘파이의 np.broadcast_to(x, shape)함수
-원래는 (3,) 형상이던 1차원 배열의 원소를 복사하여 (2,3)형상으로 바꿈

Np.broadcast_to 함수의 역전파



➔ 입력 x의 형상과 같아지도록 기울기의 합을 구함
➔ Sum_to(x, shape) 함수가 있으면 간단하게 해결
➔ (x의 원소의 합을 구해 shape 형상으로 만들어주는 함수)

넘파이 버전 sum_to 함수 준비

```
from dezero.utils import sum_to

x = np.array([[1, 2, 3], [4, 5, 6]])
y = sum_to(x, (1, 3))
print(y)

y = sum_to(x, (2, 1))
print(y)
```

```
[[5 7 9]]
[[ 6]
 [15]]
```

➔ 파일 위치는 dezero/utils.py
➔ Sum_to(x, shape)함수는 shape형상이 되도록 합을 계산
➔ 기능은 np.sum함수와 같지만 인수를 주는 방법이 다름

Sum_to 함수의 역전파

-역전파는 broadcast_to 함수를 그대로 이용

-broadcast_to함수를 사용하여 입력 x의 형상과 같아지도록 기울기의 원소를 복제

BroadcastTo 클래스와 broadcast_to 함수 구현

-역전파에서는 입력 x와 형상을 일치시키는 데 DeZero의 sum_to함수를 이용함

```
class BroadcastTo(Function):
    def __init__(self, shape):
        self.shape = shape

    def forward(self, x):
        self.x_shape = x.shape
        xp = dezero.cuda.get_array_module(x)
        y = xp.broadcast_to(x, self.shape)
        return y

    def backward(self, gy):
        gx = sum_to(gy, self.x_shape)
        return gx

def broadcast_to(x, shape):
    if x.shape == shape:
        return as_variable(x)
    return BroadcastTo(shape)(x)
```

SumTo클래스와 sum_to함수 구현

-역전파에서는 입력 x와 형상이 같아지도록 기울기의 원소를 복제

(이를 위해 DeZero의 broadcast_to함수를 사용)

-broadcst_to함수와 sum_to함수는 상호 의존적

```
class SumTo(Function):
    def __init__(self, shape):
        self.shape = shape

    def forward(self, x):
        self.x_shape = x.shape
        y = utils.sum_to(x, self.shape)
        return y

    def backward(self, gy):
        gx = broadcast_to(gy, self.x_shape)
        return gx

def sum_to(x, shape):
    if x.shape == shape:
        return as_variable(x)
    return SumTo(shape)(x)
```

브로드캐스트란

-형상이 다른 다차원 배열끼리의 연사을 가능하게 하는 넘파이 기능

-sum_to 함수를 구현한 이유는 넘파이 브로드 캐스트에 대응하기 위함

-x0와 x1은 형상이 다르지만, 계산 과정에서 x1의 원소가 x0형상에 맞춰 복제

```
x0 = Variable(np.array([1, 2, 3]))
x1 = Variable(np.array([10]))
y = x0 + x1
print(y)
```

```
variable([11 12 13])
```

➔ 순전파는 ndarray인슨턴스를 사용해 구현했기 때문에 브로드 캐스트가 일어남

➔ 브로드캐스트는 broadcast_to함수에서 이루어지고, broadcast_to함수의 역전파는 sum_to함수에 일어남

➔ 브로드캐스트의 역전파는 일어나지 않음

브로드캐스트 역전파 계산

-DeZero의 Add클래스 수정

-순전파 때 브로드캐스트가 일어난다면 입력되는 x0와 x1의 형상이 다르다는 것

-이 점을 이용해 두 형상이 다를 때 브로드캐스트용 역전파 계산

```
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 broadcaset
            gx0 = dezero.functions.sum_to(gx0, self.x0_shape)
            gx1 = dezero.functions.sum_to(gx1, self.x1_shape)
        return gx0, gx1
```

➔ 자기 형상 기억(shape)

벡터의 내적

-두 벡터 사이의 대응 원소의 곱을 모두 합한 값이 벡터의 내적

행렬의 곱

-왼쪽 행렬의 가로 방향 벡터와 오른쪽 행렬의 세로 방향 벡터 사이의 내적을 계산

-벡터의 내적과 행령의 곱 계산은 모두 np.dot함수로 처리 할 수 있음

$$\overbrace{1\times 5+2\times 7}$$

$$\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}\begin{pmatrix}5 & 6\\ 7 & 8\end{pmatrix}=\begin{pmatrix}19 & 22\\ 43 & 50\end{pmatrix}$$

$$\mathbf{a} \qquad \mathbf{b} \qquad \mathbf{c}$$

행렬과 벡터를 사용한 계산 시 체크할 점

-형상(shape)에 주의 해야함

-행렬 a와 b의 대응하는 차원(축)의 원소 수가 일치해야 함

-결과로 만들어진 행렬 c의 형상은 행렬 a와 같은 수의 행을 행렬 b와 같은 수의 열을 갖음

행렬 곱의 역전파

-최종적으로 스칼라를 출력하는 계산을 다룸

-L(손실 함수)의 각 변수에 대한 미분을 역전파로 구함

-y = xW계산을 예로 행렬 곱의 역전파를 설명

-y의 각 원소의 변화를 통해 궁극적으로 L이 변화하게 됨

(순전파 역전파 반복으로 손살 함수(오차 값) 축소)

행렬 곱 수행

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

형상 : $(N\times D)$ $(N\times H)$ $(H\times D)$

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

형상 : $(D\times H)$ $(D\times N)$ $(N\times H)$

➔ 미분 할 때 w transpose 곱한다

➔ 행렬의 원소를 계산하여 양변을 비교하면 유도됨

➔ 행렬 곱의 형상 체크도 충족하는지 확인

행렬의 곱 코드 구현

```
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

def matmul(x, W):
    return MatMul()(x, W)
```

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

x = Variable(np.random.randn(2, 3))
W = Variable(np.random.randn(3, 4))
y = F.matmul(x, W)
y.backward()

print(x.grad.shape)
print(W.grad.shape)
```

```
(2, 3)
(3, 4)
```

➔ 순전파는 np.dot(x,W) 대신 x.dot(W)로 구현

➔ 전치시에는 DeZero의 transpose 함수가 호출

➔ X.grad.shape와 x.shape가 동일하고, w.grad.shape와 W.shape가 동일함을 확인할 수 있음

토이 데이터셋

-실험용으로 만든 작은 데이터셋

선형 회귀 이론

예측 모델의 목표

-주어진 데이터를 잘 표현하는 함수 찾기

-y와 x가 선형 관계라고 가정하야, $y = Wx + b$ 식으로 표현할 수 있음

-데이터와 예측치의 차이 잔차를 최소화해야함

평균 제곱 오차

-예측치(모델)와 데이터의 오차를 나타내는 지표

-함수의 최적화 문제

-경사하강법을 사용하여 최소화하는 매개변수를 찾음

데이터의 예측치를 구하는 predict 함수 구현

-매개변수 W와 b를 Variable 인스턴스로 생성

-matmul 함수를 사용하여 행렬의 곱을 계산

```
W = Variable(np.zeros(1, 1)))
b = Variable(np.zeros(1))

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

평균 제곱 오차를 구하는 mean_squared_error 함수 구현

```
def mean_sqaured_error(x0, x1):
    diff = x0 -x1
    return F.sum(diff ** 2) / len(diff)
```

경사하강법으로 매개변수 갱신

-매개변수를 갱신할 때 W.data -= lr * W.grad.data처럼 인스턴스 변수의 data에 대한 계산

-매개변수 갱신은 단순히 단순히 데이터를 갱신함

-코드를 실행하면, 손실함수의 출력값이 줄어드는 것을 확인할 수 있음

최종 코드

```
import numpy as np
import matplotlib.pyplot as plt
from dezero import Variable
import dezero.functions as F

# Generate toy dataset
np.random.seed(0)
x = np.random.rand(100, 1)
y = 5 + 2 * x + np.random.rand(100, 1)
x, y = Variable(x), Variable(y)

W = Variable(np.zeros((1, 1)))
b = Variable(np.zeros(1))
```

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

```
def mean_squared_error(x0, x1):
    diff = x0 -x1
    return F.sum(diff ** 2) / len(diff)
```

```
lr = 0.1
iters = 100

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

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

    # Update .data attribute
    W.data -= lr * W.grad.data
    b.data -= lr * b.grad.data
    print(W, b, loss)
```

Mean_squared_error 함수

-메모리를 차지하고 있기 때문에 temp를 초기화

-mean_squared_error 함수를 class로 만들어서 클래스에서 순전파, 역전파를 하여서 메모리를 최적화 한다

신경망

-선형 회귀 구현을 신경망으로 확장

-아핀 변환(Affine Transformation)

-행렬 곱을 구하고 b(매개변수)를 더함

-선형 변환은 신경망에서는 완전연결계층에 해당 - fully connect

➔ 선형변환을 linear함수로 구현하는 방식

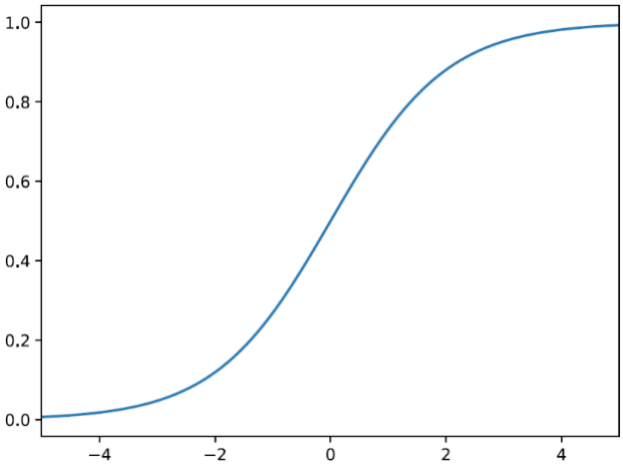
비선형 데이터를 학습하기 위해 신경망 사용

-신경망은 선형 변환의 출력에 비선형 변환을 수행

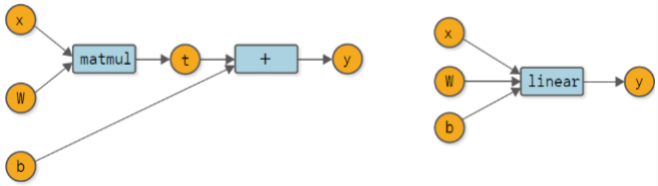
-이 비선형 변환이 활성화 함수임(ReLU, sigmoid)

시그모이드 함수

$$y = \frac{1}{1 + \exp(-x)}$$



W * x + b 를 linear라는 클래스로 정의하여 사용자가 사용하기 쉽게 구현




```
def linear_simple(x, W, b=None):
    t = matmul(x, W)
    if b is None:
        return t

    y = t + b
    t.data = None # t의 데이터 삭제
    return y
```

➔ 층이 깊어질수록 매개변수 관리가 힘들어진다.

➔ parameter와 layer 라는 클래스를 구현해서 매개변수를 쉽게 다룰 수 있다.

➔ parameter클래스는 Variable클래스를 상속한 것 뿐이기 때문에 똑같은 기능을 가짐

➔ parameter인스턴스와 Variable인스턴스는 isinstance함수로 구별이 가능하다.

Layer클래스 : 변수를 변환하는 클래스

-매개변수를 유지한다는 점에서 Function클래스와 차이점이 있음

```
from dezero.core import Parameter

class Layer:
    def __init__(self):
        self._params = set()

    def __setattr__(self, name, value):
        if isinstance(value, Parameter):
            self._params.add(name)
        super().__setattr__(name, value)
```

➔ params라는 인스턴스 변수에 매개변수를 set()으로 보관

```
layer = Layer()

layer.p1 = Parameter(np.array(1))
layer.p2 = Parameter(np.array(2))
layer.p3 = Variable(np.array(3))
layer.p4 = 'test'

print(layer._params)
print('---------------')
```

```
for name in layer._params:
    print(name, layer.__dict__[name])
```

```
{'p2', 'p1'}
---------------
p2 variable(2)
p1 variable(1)
```

➔ parameter를 보관하기 때문에 p1, p2가 출력됨

```
class Layer:
    ...

    def __call__(self, *inputs):
        outputs = self.forward(*inputs)
        if not isinstance(outputs, tuple):
            outputs = (outputs,)
        self.inputs = [weakref.ref(x) for x in inputs]
        self.outputs = [weakref.ref(y) for y in outputs]
        return outputs if len(outputs) > 1 else outputs[0]

    def forward(self, inputs):
        raise NotImplementedError()

    def params(self):
        for name in self._params:
            yield self.__dict__[name]

    def cleargrads(self):
        for param in self.params():
            param.cleargrad()
```

➔ layer클래스에 4개의 메서드 추가

➔ __call__ 메서드

➔ forward 메서드

➔ params 메서드

➔ cleargrad 메서드

yield 반환 : 처리를 일시 중지하고 값을 반환, yield를 사용하면 작업을 재개할 수 있음

선형 변환을 하는 linear클래스 구현

-layer클래스를 상속하여 계층으로서 구현

```
class Linear(Layer):
    def __init__(self, in_size, out_size, nobias=False, dtype=np.float32):
        super().__init__()

        I, O = in_size, out_size
        W_data = np.random.randn(I, O).astype(dtype) * np.sqrt(1 / I)
        self.W = Parameter(W_data, name='W')
        if nobias:
            self.b = None
        else:
            self.b = Parameter(np.zeros(O, dtype=dtype), name='b')

    def forward(self, x):
        y = F.linear(x, self.W, self.b)
        return y
```

➔ forward 메서드로 선형 변환을 구현(linear함수 호출)

linear클래스를 구현하는 더 나은 방법

-__init__메서드에서 in_size를 지정하지 않아도 됨

-forward(self, x)메서드에서 입력x의 크기에 맞게 가중치 데이터를 생성하기 때문

-순전파 시 입력값을 초기화하기 힘들기 때문에 자동으로 초기화를 시켜줌

layer 인스턴스 자체도 관리가 필요함

-현재 layer클래스는 parameter만 관리를 했지만

-layer안에 layer가 들어가는 구조를 만들어서 바깥 layer에서 그 안에 존재하는 모든 매개변수를 꺼낼 수 있도록 함.

-layer인스턴스의 이름도 params에 추가함

-layer에 parameter만 있었기 때문에 name만 집어넣으면 됐지만 layer안에 layer가 들어가게 되면 layer를 꺼낼 수 있는 함수를 설정해야함

-obj.params()함수로 layer 속에 layer에서도 매개변수를 재귀적으로 꺼낼 수 있음

- ➔ TwoLayerNet클래스를 구현하여 신경망에 필요한 모든 코드를 집약할 수 있다.
- ➔ 모델을 표현하기 위해 Model클래스 생성
- ➔ Model클래스는 마치 Layer클래스처럼 활용할 수 있음
- ➔ 편의성을 위해서 class 확장

느낀점

-DeZero부분이 많아서 끝났을 때 거의 다 했다고 생각했는데 그 이후에 하는게 더 복잡하다는것을 느꼈다.