선형회귀

y와 x가 선형 관계라고 가정하여, y = Wx + b 식으로 표현
데이터와 예측치의 차이 잔차를 최소화해야 함
예측치와 데이터의 오차를 나타내는 지표로 평균제곱오차를 사용
손실 함수의 출력을 최소화하는 W와 b를 찾는 것

데이터의 예측치를 구하는 predict함수
매개변수 W와 b를 Variable 인스턴스로 생성
matmul함수를 사용하여 행렬의 곱을 계산

평균제곱오차를 구하는 mean_squared_error 함수 구현
경사하강법으로 매개변수 갱신
손실함수의 값을 점점 줄여나감

mean_squared_error 함수 구현의 문제점
이름없는 변수들이 계산그래프가 존재하는 동안 메모리에 계속 존재함.
이 변수들의 데이터도 마찬가지로 계속 존재함
메모리 최적화를 위해 더 나은 방식 도입

mean_squared_error 함수 개선
Function 클래스를 상속받아 구현하는 방식
중간에 등장하던 변수들이 사라짐
앞서 구현한 함수와 같은 결과를 얻지만 메모리는 덜 사용함

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

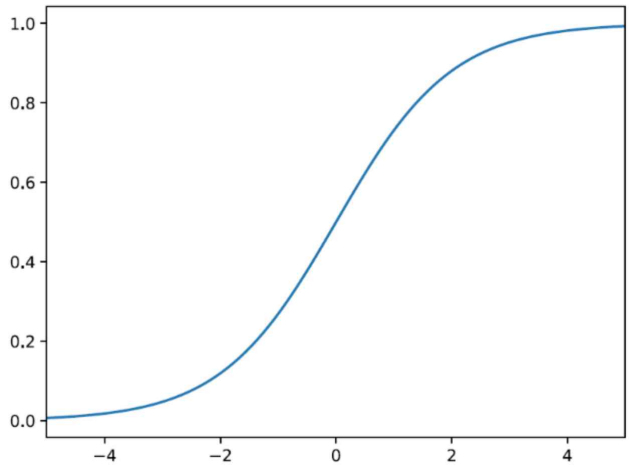
비선형 데이터를 학습하기 위해 신경망 사용

신경망은 선형 변환의 출력에 비선형 변환을 수행

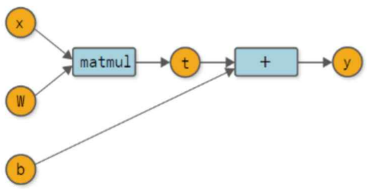
이 비선형 변환이 활성화 함수임(ReLU, sigmoid)

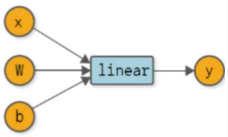
시그모이드 함수

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



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.

laye클래스 : 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

결론

편의성을 위해서 class 확장