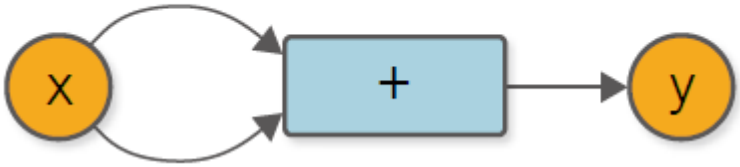
제2 고지 : 자연스러운 코드로

STEP 14 : 같은 변수 반복 사용

그림 14-1 y = add(x, x)의 계산 그래프



현재까지의 코드는 아래와 같이 **동일한 변수를 반복 사용**할 경우 의도대로 동작하지 않을 수 있다는 문제가 있다.

In [ ]:

```
import numpy as np


def as_array(x):

    """
    0차원 ndarray / ndarray가 아닌 경우
    """
    if np.isscalar(x):
        return np.array(x)
    return x


class Variable:
    def __init__(self, data: np.ndarray) -> None:
        if data is not None:
            if not isinstance(data, np.ndarray):
                raise TypeError(f"{type(data)}은(는) 지원하지 않습니다.")
        self.data = data
        self.grad = None  # gradient
        self.creator = None  # creator

    def set_creator(self, func) -> None:
        self.creator = func

    def backward(self):
        """
        자동 역전파 (반복)
        """
        if self.grad is None:
            self.grad = np.ones_like(self.data)
        funcs = [self.creator]
        while funcs:
            f = funcs.pop()
            ########################################
            gys = [output.grad for output in f.outputs] # 1. 순전파의 결과가 **
            gxs = f.backward(*gys) # 2. 역전파 기준 **여러 개의 입력(=순전파의 여러 
            if not isinstance(gxs,tuple): # 3. 역전파 **결과값이 하나인 경우(=역전파
                gxs = (gxs,)
            for x,gx in zip(f.inputs,gxs): # 4. **역전파 결과가 여러개의 출력인 경우
                x.grad = gx
```

```
                if x.creator is not None:
                    funcs.append(x.creator)  # 하나 앞의 함수를 리스트에 추가한다
            ########################################


class Function:
    """
    Function Base Class

    """

    def __call__(self, *inputs):  #  1. * 를  활용하여 임의 개수의 인수
        xs = [x.data for x in inputs]
        ####################################
        ys = self.forward(*xs) # 1. 리스트 언팩
        if not isinstance(ys,tuple): # 2. 튜플이 아닌 경우 추가 지원
            ys = (ys,)
        ####################################
        outputs = [Variable(as_array(y)) for y in ys]

        for output in outputs:
            output.set_creator(self)
        self.inputs = inputs
        self.outputs = outputs

        # 2. 리스트의 원소가  하나라면  첫번째 원소를 반환
        return outputs if len(outputs) > 1 else outputs[0]

    def forward(self, xs):
        """
        구체적인 함수 계산 담당
        """
        raise NotImplementedError()

    def backward(self, gys):
        """
        역전파
        """
        raise NotImplementedError()


class Add(Function):
    def forward(self, x0,x1):
        y = x0 + x1
        return y
    def backward(self, gy):
        # 역전파시 , 입력이 1개 , 출력이 2개
        return gy,gy

def add(x0,x1):
    return Add()(x0,x1)

class Square(Function):
    def forward(self, x):
        y= x**2
        return y
    def backward(self, gy):
        x = self.inputs[0].data  #  수정 전 : x= self.input.data
        gx = 2 * x * gy
        return gx

def square(x):
    return Square()(x)
```

$$y = x + x \Rightarrow \frac{\partial y}{\partial x} = 2$$

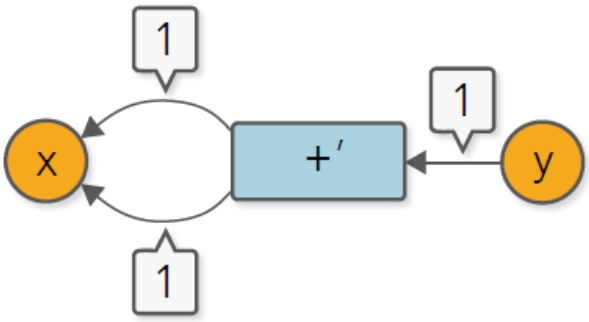
In [ ]:

```
x = Variable(np.array(3.0))
y = add(x,x)
print(f"y : {y.data}")
y.backward()
print(f"x.grad : {x.grad}") # 잘못된 결과값
```

```
y : 6.0
x.grad : 1.0
```

14.1 문제의 원인

그림 14-2 y = add(x, x)의 역전파(화살표 위아래의 숫자는 전파되는 미분값)



- 동일한 변수 반복 사용시 `backward()` 에서 전파되는 미분값이 **덮어 써진다.**
- 이를 해결 하기 위해 전파되는 미분값의 **합** 을 구해야 한다.

14.2 해결책

In [ ]:

```
class Variable:
    def __init__(self, data: np.ndarray) -> None:
        if data is not None:
            if not isinstance(data, np.ndarray):
                raise TypeError(f"{type(data)}은(는) 지원하지 않습니다.")
        self.data = data
        self.grad = None  # gradient
        self.creator = None  # creator

    def set_creator(self, func) -> None:
        self.creator = func

    def backward(self):
        """
        자동 역전파 (반복)
        """
        if self.grad is None:
            self.grad = np.ones_like(self.data)
        funcs = [self.creator]
        while funcs:
            f = funcs.pop()
            gys = [output.grad for output in f.outputs] # 1. 순전파의 결과가 **
            gxs = f.backward(*gys) # 2. 역전파 기준 **여러 개의 입력(=순전파의 여러 
            if not isinstance(gxs,tuple): # 3. 역전파 **결과값이 하나인 경우(=역전파
                gxs = (gxs,)
            for x,gx in zip(f.inputs,gxs): # 4. **역전파 결과가 여러개의 출력인 경우
                ##################
```

```
                #  첫 grad를 설정시에는 `그대로` 출력하고,
                if x.grad is None :
                    x.grad = gx
                # 다음 미분은 기존 미분 값에 `더해준다.`
                else :
                    ## NOTE :  in-place 연산 (x.grad+=gx) 을 하지 않는 이유는 **
                    x.grad = x.grad + gx

                ######################

                if x.creator is not None:
                    funcs.append(x.creator)  # 하나 앞의 함수를 리스트에 추가한다
```

여기서 주목해야 할것은

```
x.grad = x.grad + gx  # NOTE: x.grad+=gx (in-place 연산) 을 사용하지 않도록 한다
```

인데, in-place 연산 을 사용하지 않는 이유는 **메모리 참조**로 원하지 않는 값의 변동이 일어 날 수 있기 때문이다.

구체적으로 아래의 예를 살펴보면, in-place 연산 시 **메모리 위치는 동일하고, 값만 바뀌는 것**을 확인할 수 있다. 물론 메모리 효율적인 측면에서는 in-place 연산 이 문제가 없는 상황이라면 바람직하겠지만, 현재는 **메모리를 참조하는 다른 변수의 미분값이 바뀔수 있으므로, 새로운 값을 복사해서 사용**해야 한다.

In [ ]:

```
x=np.array(1)
print(id(x))

x+=x #  in-place 연산
print(id(x))

x = x+x # 복사
print(id(x))
```

```
4437014160
4437014160
4783135824
```

In [ ]:

```
class Variable:
    def __init__(self, data: np.ndarray) -> None:
        if data is not None:
            if not isinstance(data, np.ndarray):
                raise TypeError(f"{type(data)}은(는) 지원하지 않습니다.")
        self.data = data
        self.grad = None  # gradient
        self.creator = None  # creator

    def set_creator(self, func) -> None:
        self.creator = func

    def backward(self):
        """
        자동 역전파 (반복)
        """
        if self.grad is None:
            self.grad = np.ones_like(self.data)
        funcs = [self.creator]
        while funcs:
            f = funcs.pop()
            gys = [output.grad for output in f.outputs] # 1. 순전파의 결과가 **
            gxs = f.backward(*gys) # 2. 역전파 기준 **여러 개의 입력(=순전파의 여러 
            if not isinstance(gxs,tuple): # 3. 역전파 **결과값이 하나인 경우(=역전파
```

```
                gxs = (gxs,)
            for x,gx in zip(f.inputs,gxs): # 4. **역전파 결과가 여러개의 출력인 경우
                ##################
                #  첫 grad를 설정시에는 `그대로` 출력하고,
                if x.grad is None :
                    x.grad = gx
                # 다음 미분은 기존 미분 값에 `더해준다.`
                else :
                    ## NOTE :  만약 inplace연산을 사용한다면 어떻게 될까?
                    x.grad +=gx

                ######################

                if x.creator is not None:
                    funcs.append(x.creator)  # 하나 앞의 함수를 리스트에 추가한다
```

In [ ]:

```
x = Variable(np.array(3.0))
y = add(x,x)
print(f"y : {y.data}")
y.backward()
print(f"in-place 연산 x.grad : {x.grad}({id(x)})")
print(f"in-place 연산 y.grad : {y.grad}({id(x)}), 정확한 y.grad: {1.0}")  # 메
```

```
y : 6.0
in-place 연산 x.grad : 2.0(4436751408)
in-place 연산 y.grad : 2.0(4436751408), 정확한 y.grad: 1.0
```

14.3 미분값 재설정

하지만, 역전파시 미분값을 더해주도록 코드를 수정함에 따라 또 다른 문제가 발생하는데, **동일한 변수**를 사용하여 **다른 계산** 을 할 경우 계산이 꼬인다. 이를 해결하기 위해 미분값을 초기화 해주는 `cleargrad()` 를 추가한다.

In [ ]:

```
class Variable:
    def __init__(self, data: np.ndarray) -> None:
        if data is not None:
            if not isinstance(data, np.ndarray):
                raise TypeError(f"{type(data)}은(는) 지원하지 않습니다.")
        self.data = data
        self.grad = None  # gradient
        self.creator = None  # creator

    def set_creator(self, func) -> None:
        self.creator = func

    def backward(self):
        """
        자동 역전파 (반복)
        """
        if self.grad is None:
            self.grad = np.ones_like(self.data)
        funcs = [self.creator]
        while funcs:
            f = funcs.pop()
            gys = [output.grad for output in f.outputs] # 1. 순전파의 결과가 **
            gxs = f.backward(*gys) # 2. 역전파 기준 **여러 개의 입력(=순전파의 여러 개
            if not isinstance(gxs,tuple): # 3. 역전파 **결과값이 하나인 경우(=역전파
                gxs = (gxs,)
            for x,gx in zip(f.inputs,gxs): # 4. **역전파 결과가 여러개의 출력인 경우
                ##################
                #  첫 grad를 설정시에는 `그대로` 출력하고,
```

```
                if x.grad is None :
                    x.grad = gx
                # 다음 미분은 기존 미분 값에 `더해준다.`
                else :
                    ## NOTE :  in-place 연산 (x.grad+=gx) 을 하지 않는 이유는 **d
                    x.grad = x.grad + gx

                ####################

                if x.creator is not None:
                    funcs.append(x.creator)  # 하나 앞의 함수를 리스트에 추가한다
```

In [ ]:

```
## 첫 번째 계산
x = Variable(np.array(3.0))
y = add(x,x)
y.backward()
print(f"x.grad : {x.grad}")

## 두 번째 계산 (같은 x를 사용하여 다른 계산을 수행)
y = add(add(x,x),x) # y = (x + x) + x
y.backward()
print(f"같은 x를 사용하여 다른 계산 수행 x.grad : {x.grad}, 정확한 x.grad : 3.0")
```

```
x.grad : 2.0
같은 x를 사용하여 다른 계산 수행 x.grad : 5.0, 정확한 x.grad : 3.0
```

이를 해결하기 위해 미분값을 초기화 해주는 `cleargrad()` 를 추가한다.

In [ ]:

```
class Variable:
    def __init__(self, data: np.ndarray) -> None:
        if data is not None:
            if not isinstance(data, np.ndarray):
                raise TypeError(f"{type(data)}은(는) 지원하지 않습니다.")
        self.data = data
        self.grad = None  # gradient
        self.creator = None  # creator

    def cleargrad(self):
        self.grad = None

    def set_creator(self, func) -> None:
        self.creator = func

    def backward(self):
        """
        자동 역전파 (반복)
        """
        if self.grad is None:
            self.grad = np.ones_like(self.data)
        funcs = [self.creator]
        while funcs:
            f = funcs.pop()
            gys = [output.grad for output in f.outputs] # 1. 순전파의 결과가 **
            gxs = f.backward(*gys) # 2. 역전파 기준 **여러 개의 입력(=순전파의 여러 개
            if not isinstance(gxs,tuple): # 3. 역전파 **결과값이 하나인 경우(=역전파
                gxs = (gxs,)
            for x,gx in zip(f.inputs,gxs): # 4. **역전파 결과가 여러개의 출력인 경우
                ##################
                #  첫 grad를 설정시에는 `그대로` 출력하고,
                if x.grad is None :
                    x.grad = gx
                # 다음 미분은 기존 미분 값에 `더해준다.`
```

```
                else :
                    ## NOTE :  in-place 연산 (x.grad+=gx) 을 하지 않는 이유는 **D
                    x.grad = x.grad + gx

                ######################

                if x.creator is not None:
                    funcs.append(x.creator)  # 하나 앞의 함수를 리스트에 추가한다
```

In [ ]:

```
# Dezero ~ PyTorch
## Dezero
## 첫 번째 계산
x = Variable(np.array(3.0))
y = add(x,x)
y.backward()
print(f"Dezero x.grad : {x.grad}")

## 두 번째 계산 (같은 x를 사용하여 다른 계산을 수행)
x.cleargrad()
y = add(add(x,x),x) # y = (x + x) + x
y.backward()
print(f"Dezero x.grad : {x.grad}")

## PyTorch
import torch
x = torch.tensor([3.0],requires_grad=True)
y = x+x
y.backward()
print(f"PyTorch x.grad : {x.grad}")

## 두 번째 계산 (같은 x를 사용하여 다른 계산을 수행)
y = x+x+x
y.backward()
print(f"PyTorch 미분값 초기화 전 x.grad : {x.grad}, 정확한 x.grad : 3.0")

x.grad.zero_()
y = x+x+x # y = (x + x) + x
y.backward()
print(f"PyTorch 미분값 초기화 후 x.grad : {x.grad}")
```

```
Dezero x.grad : 2.0
Dezero x.grad : 3.0
PyTorch x.grad : tensor([2.])
PyTorch 미분값 초기화 전 x.grad : tensor([5.]), 정확한 x.grad : 3.0
PyTorch 미분값 초기화 후 x.grad : tensor([3.])
```