제 2고지 17~19. 자연스러운 코드로

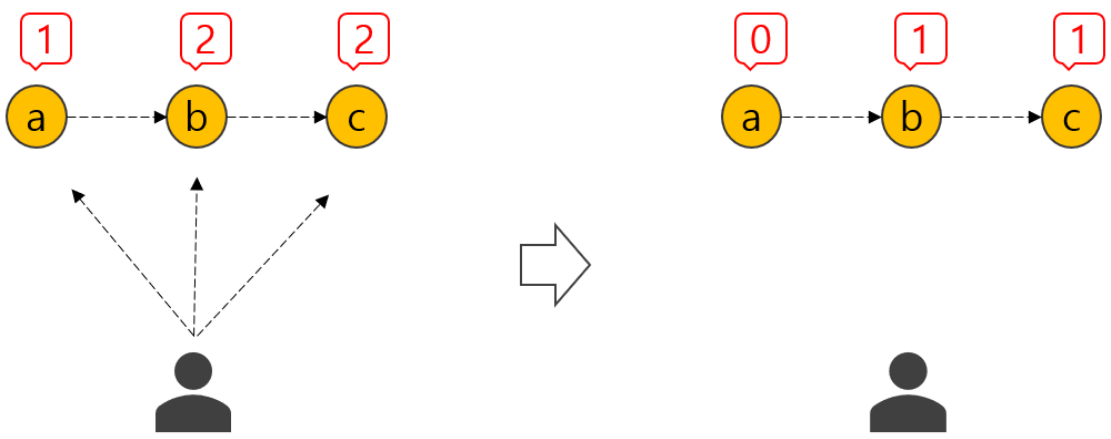
메모리 관리 방식

CPython의 메모리 관리는 두가지 방식으로 진행

1. **참조 카운트** : 참조(reference) 수를 세어 쓸모없는 객체를 회수
2. **GC(Garbage Collection)** : 세대(generation)을 기준으로 쓸모없는 객체를 회수 (순환참조)

참조카운트

- 모든 객체는 참조 카운트가 0인 상태로 생성
- 다른 객체가 참조할 때 마다 +1, 객체에 대한 참조가 끊기면 -1
- 0이 되면 해당 객체를 메모리에서 삭제



a가 b를 참조, b가 c를 참조 후 a = b= c = None을 실행하면 오른쪽처럼 참조카운트가 변화함
a의 참조 카운트가 0이 되면서 삭제되고, b의 참조 카운트도 0이 되어 삭제, c의 참조 카운트도 0이 되어 삭제

```
class obj:
  pass

def f(x):
  print(x)

a = obj() #변수에 대입: 참조 카운트 1
f(a) #함수에 전달: 참조 카운트 2
#함수 완료: 빠져나오면 참조 카운트 1
a = None #대입 해제: 참조 카운트 0
```

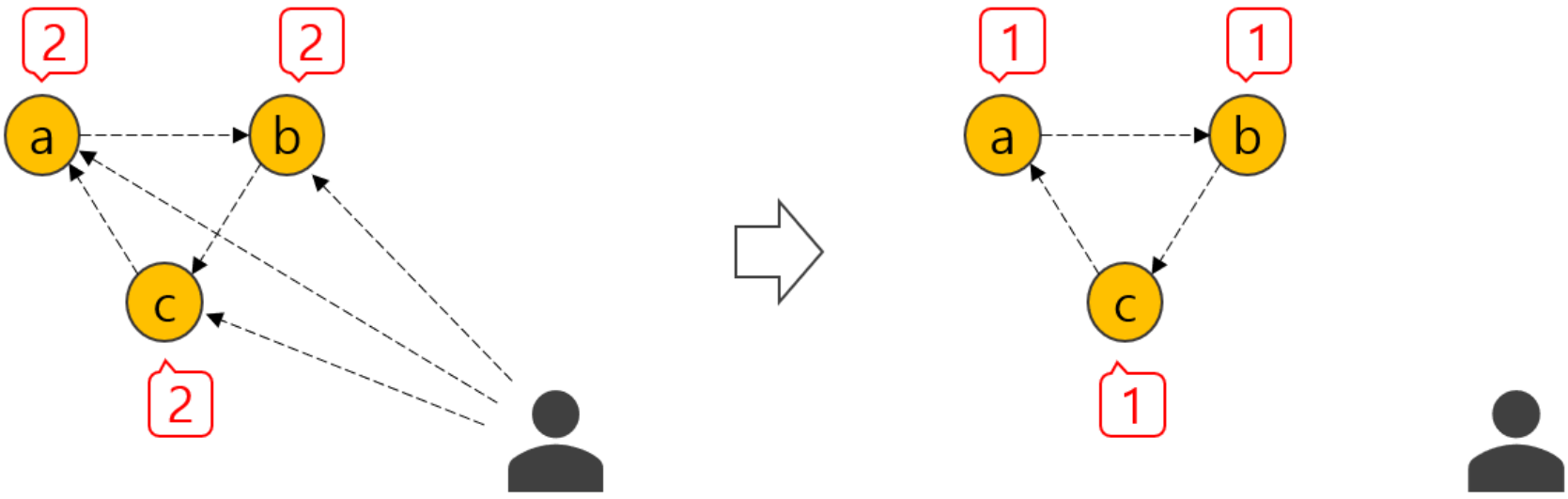
GC(순환 참조)

순환 참조의 메모리 관리를 위한 방식

```
a = obj()
b = obj()
c = obj()

a.b = b
b.c = c
c.a = a

a = b = c = None
```



- 우측 그림에서 a,b,c의 참조 카운트는 모두 1이나, 사용자는 어느 객체에도 접근이 불가능 하다. → 불필요한 객체
- 그러나 a = b = c = None을 실행하여도 순환 참조의 참조 카운트는 0이 되지 않아 메모리에서 삭제되지 않는다.

- GC는 메모리가 부족해지는 시점에 **파이썬 인터프리터에 의해 자동**으로 호출
  - import하여 gc.collect()로 명시적 호출도 가능
- 신경망 코드 개발에서 메모리가 중요한 자원이므로 순환참조가 생기지 않도록 구현해야함

weakref 모듈

- 순환 참조 해결에 weakref.ref 함수를 사용하여 약한 참조를 생성
- **약한 참조(weak reference) : 다른 객체를 참조하되, 참조 카운트는 증가시키지 않음**

```
import weakref
import numpy as np

a = np.array([1, 2, 3])
b = weakref.ref(a)

b() #[1 2 3]
```

- a는 일반적인 방식으로 참조하고 b는 약한 참조를 가지도록 함
- 참조 데이터에 접근할 때는 b() 로 접근

```
a = None
b #<weakref at 0x103b7f048; dead>
```

- a = None을 실행하면 결과는 아래와 같으며, ndarray 인스턴스는 메모리에서 삭제된다.
- b는 약한 참조이므로 참조 카운트에는 영향을 주지 못하지만, b를 출력했을 때 dead를 보고 파이썬 인터프리터에 의해 삭제된 것을 알 수 있음

weakref 구조를 신경망 구조에 도입하면 다음과 같다

```
import weakref

class function:
  def __call__(self, *inputs):
      xs = [x.data for x in inputs]
        ys = self.forward(*xs)
        if not isinstance(ys, tuple)
          ys = (ys,)
        outputs = [Variable(as_array(y)) for y in ys]

        self.generation = max([x.generation for x in inputs])
        for output in outputs:
          output.set_creator(self)
        self.inputs = inputs
        self.outputs = [weakref.ref(output) for output in outputs] #weakref적용
        return outputs if len(outputs) > 1 else outputs[0]
```

- self.outputs가 대 상을 약한 참조로 가리키도록 하였으며, 다른 클래스에서 Function class의 output을 참조하는 코드도 수정하면 된다.

```
class Variable:

  #생략

    def backward(self):

        #생략

        while funcs:
        f = funcs.pop()
        #수정 전: gys = [output.grad for output in f.outputs]
        gys = [output().grad for output in f.outputs]

        #생략
```

- 참조한 데이터에 접근하기 위해서는 괄호()를 붙여야 하므로,
  [output.grad for ~] 부분을 [output**()**.grad for ~]로 수정

메모리 절약 모드

1. 역전파 시 불필요한 미분 결과를 즉시 삭제
   - 아래 코드에서 역전파로 구하고자 하는 미분값은 말단 변수인 x0, x1 인데,
     y나 t와 같은 중간 변수의 미분값도 저장된다.

```
x0 = Variable(np.array(1.0))
x1 = Variable(np.array(1.0))
t = add(x0, x1)
y = add(x0, t)
y.backward()

print(y.grad, t.grad) #1.0 1.0
print(x0.grad, x1.grad) #2.0 1.0
```

   - 이를 아래와 같이
     메서드 인수에 retain_grad를 추가한다.
     - retain_grad가 True이면 모든 미분값이 유지되고, False이면 중간 변수의 미분값은 None으로 재설정하고 말단 변수의 미분값만 유지한다.

```
class Variable:

    #생략

    def backward(self, retain_grad = False):
      if self.grad is None:
          self.grad = np.ones_like(self.data)

        funcs = []
        seen_set = set()

        def add_func(f):
          if f not in seen_set:
              funcs.append(f)
                seen_set.add(f)
                funcs.sort(key=lambda x: x.generation)

        add_func(self.creator)

        while funcs:
          f = funcs.pop()
            gys = [output().grad for output in f.outputs]
            gxs = f.backward(*gys)
            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 += gx

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

```
            if not retain_grad:
              for y in f.outputs:
                  y().grad = None #y는 약한 참조
```

2. 역전파가 필요 없는 경우용 모드

- 미분을 수행하기 위해서는 순전파를 수행한 뒤 역전파를 수행해야 함
- 역전파 때는 순전파 시의 계산 결과 값이 필요하여 그 값을 저장해두어야 함
  - 아래의 self.inputs = inputs 를 통해 값을 저장한다.

```
class Function:
  def __call__(self, *inputs):
      xs = [x.data for x in inputs]
        ys = self.forward(*xs)
        if not isinstance(ys, tuple):
          ys = (ys,)
        outputs = [Variable(as_array(y)) for y in ys]

        self.generation = max([x.generation for x in inputs])
        for output in outputs:
          output.set_creator(self)
        self.inputs = inputs ##################이 부분에서 순전파 결과값 저장###############
        self.outputs = [weakref.ref(output) for output in outputs]
        return outputs if len(outputs) > 1 else outputs[0]
```

- 함수의 입력을 인스턴스 변수 inputs로 참조하며, 역전파 하는 경우 참조할 변수를 inputs에 보관한다.
  - 신경망에는 학습과 추론이 있으며, 학습 시에는 미분값을 구해야하지만, 추론 시에는 단순 순전파만 수행하므로 중간 계산 결과를 즉시 버려 메모리 사용량을 줄일 수 있도록 함
- 역전파가 필요 없는 경우, 메모리 절약을 위해 config 클래스를 활용하여 역전파 활성 여부를 설정할 수 있도록 한다

```
class Config:
  enable_backprop = True #불리언 타입으로 역전파 가능 여부를 의미
                         # True 이면 역전파 활성 모드
```

- **with문을 활용한 모드 전환**
  - 후처리를 자동으로 수행하고자 할 때 사용하는 with

```
f = open('sample.txt', 'w')
f.write('hello world!')
f.close()
```

  - open()으로 파일을 열어 쓴뒤 close()로 닫아야 함
  - 매번 close() 하는 불편함을 해결할 때 with 사용

```
with open('sample.txt', 'w') as f:
  f.write('hello world!')
```

    - with 문에 들어갈때 파일이 열리고(전처리) 나올때 자동으로 닫힘(후처리)

- contextlib 모듈과 with문을 사용하여 모드 전환 구현

```
import contextlib

@contextlib.contextmanager
def config_test():
  print('start') #전처리
    try:
      yield
    finally:
      print('done') #후처리

with config_test():
  print('process...')


#실행 결과
```

```
start
process...
done
```

-