<모델 저장 및 읽어오기>

모델이 가지는 매개변수를 외부 파일로 읽고 저장하고 다시 읽어오는 기능을 만든다.
학습 중인 모델의 스냅샷을 저장하거나 이미 학습된 매개변수를 읽어와서 추론만 수행한다.
* Dezero의 매개변수
- Parameter 클래스로 구현한다.
- Parameter의 데이터는 인스턴스 변수 data에 ndarray 인스턴스로 보관한다.
- ndarray 인스턴스를 외부 파일로 저장한다.

<넘파이의 save와 load함수>

* 넘파이의 저장 및 읽어오기
- np.save와 np.load 함수를 사용하면 ndarray인스턴스를 저장하고 읽어올 수 있다.
- np.save 함수, ndarray 인스턴스를 외부 파일로 저장한다.
- np.load 함수는 저장되어 있는 데이터를 읽어온다.

```
import numpy as np

x = np.array([1, 2, 3])
np.save('test.npy', x)

x = np.load('test.npy')
print(x)
```

```
[1 2 3]
```

* 여러개의 ndarray 인스턴스를 저장하고 읽어오기

- np.savez('test.npz', x1 = x1, x2 = x2)코드로 여러개의 ndarray 인스턴스를 저장한다.
- x1 = x1, x2 = x2처럼 키워드 인수를 지정할 수 있다.
- 데이터를 읽을 때 array['x1']이나 arrays['x2']처럼 원하는 키워드를 명시하여 해당 데이터를 불러올 수 있다.
- np.savez 함수로 저장하는 파일의 확장자는 .npz로 해준다.

```
x1 = np.array([1, 2, 3])
x2 = np.array([4, 5, 6])
np.savez('test.npz', x1 = x1, x2 = x2)
arrays = np.load('test.npz')

x1 = arrays['x1']
x2 = arrays['x2']
print(x1)
print(x2)
```

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

* 넘파이의 코드를 파이썬 딕셔너리를 이용하여 수정

- np.savez('test.npz', **data) 코드로 데이터를 저장한다.
- 딕셔너리 타입의 인수를 전달할 때 **data와 같이 앞에 별표 두 개를 붙여주면 딕셔너리가 자동으로 전개되어 전달된다.

```
x1 = np.array([1, 2, 3])
x2 = np.array([4, 5, 6])
data = {'x1':x1, 'x2':x2}
np.savez('test.npz', **data)

arrayts = np.load('test.npz')
x1 = arrays['x1']
x2 = arrays['x2']
print(x1)
print(x2)
```

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

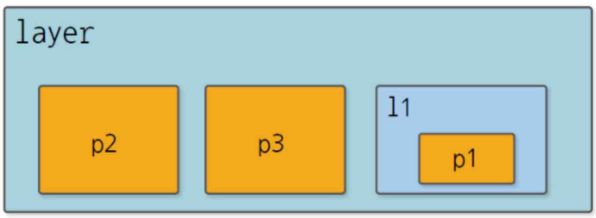
* Dezero 매개변수를 외부 파일로 저장하는 기능

- 위에서 설명한 함수들을 이용한다.
- Layer 클래스 안의 Parameter를 평탄화로 꺼낸다.

<Layer 클래스의 매개변수를 평평하게>

* Layer 클래스의 계층 구조

- 계층은 Layer 안에 다른 Layer가 들어가는 중첩 형태의 구조이다.
- layer에 또 다른 계층인 l1을 넣는다.




```
layer = Layer()
l1 = Layer()
l1.p1 = Parameter(np.array(1))
layer.l1 = l1
layer.p2 = Parameter(np.array(2))
layer.p3 = Parameter(np.array(3))
```

* 중첩되지 않는 딕셔너리로 선출 방법

- Layer 클래스에 _flatten_params 메서드를 추가한다.
- 출력 결과를 보면 l1계층 안의 매개변수 p1은 1l/p1이라는 키로 저장된다.

```
params_dict = {}
layer._flatten_params(params_dict)
print(params_dict)
```

```
{'l1/p1': variable(1), 'p2': variable(2), 'p3': variable(3)}
```

* _flatten_params 메서드 구현

```
def _flatten_params(self, params_dict, parent_key = ''):
    for name in self._params:
        obj = self.__dict__[name]
        key = parent_key + '/' + name if parent_key else name

        if isinstance(obj, Layer):
            obj._flatten_params(params_dict, key)
        else:
            params_dict[key] = obj
```

- 인수로 딕셔너리인 params_dict와 텍스트인 parent_key를 받는다.
- _params에는 'Parameter의 인스턴스 변수 이름' 또는 'Layer 인스턴스 변수 이름'이 담긴다.
- 실제 객체는 obj = self.__dict__[name]으로 꺼내야 한다.
- 꺼낸 obj가 Layer라면 obj의 _flatten_params 메서드를 호출한다.
- 메서드가 재귀적으로 호출되므로 모든 Parameter를 한 줄로 평탄화시켜 꺼낼 수 있다.

* Layer 클래스의 매개변수를 외부 파일로 저장

```
def save_weights(self, path):
    self.to_cpu()

    params_dict = {}
    slef._flatten_params(params_dict)
    array_dict = {key: param.data for key, param in params_dict.item()
                  if param is None}
    try:
        np.savez_compressed(path, **array_dict)
    except (Exception, KeyboardInterrupt) as e:
        if os.path.exists(path):
            os.remove(path)
        raise

def load_weights(self, path):
    npz = np.load(path)
    params_dict = {}
    self._flatten_params(params_dict)
    for key, param in params_dict.items():
        param.data = npz[key]
```

- save_weights 메서드는 먼저 self.to_cpu()를 호출하여 데이터가 메인 메모리에 존재함을 보장한다.
- ndarray 인스턴스를 값으로 갖는 딕셔너리 array_dict를 만든다.
- np.savez_compressed 함수를 호출하여 데이터를 외부 파일로 저장한다.
- load_weights 메서드는 np.load 한수로 데이터를 읽어 들인 후 대응하는 키 데이터를 매개변수로 설정한다.

* MNIST 학습으로 매개변수 저장과 읽기 기능 시험

- 모델의 매개변수를 무작위로 초기화한 상태에서 학습을 시작한다.
- (1) model.save_weights('my_mlp.npz') 줄에서 학습된 매개변수들을 저장한다.
- (2) my_mlp.npz 파일이 존재하므로 파일로부터 매개변수들을 읽어들인다.
- 앞서 학습한 매개변수값이 모델에 설정된다.

```
import os
import dezero
import dezero.functions as F
from dezero import optimizers
from dezero import DataLoader
from dezero.models import MLP
```

```
max_epoch = 3
batch_size = 100

train_set = dezero.datasets.MNIST(train = True)
train_loader = DataLoader(train_set, batch_size)
model = MLP((1000, 10))
optimizer = optimizers.SGD().setup(model)

if os.path.exists('my_mlp.npz'):
    model.load_weights('my_mlp.npz')

for epoch in range(max_epoch):
    sum_loss = 0

    for x, t in train_loader:
        y = model(x)
        loss = F.softmax_cross_entropy(y, t)
        model.cleargrads()
        loss.backward()
        optimizer.update()
        sum_loss += float(loss.data) * len(t)

    print('epoch: {}, loss: {:.4f}'.format(
        epoch + 1, sum_loss / len(train_set)))

model.save_weights('my_mlp.npz')
```

```
epoch: 1, loss: 0.7343
epoch: 2, loss: 0.6315
epoch: 3, loss: 0.5660
```

<드롭아웃과 테스트 모드>

* 과적합이 일어나는 주요 원인
- 훈련 데이터가 적음
- 모델의 표현력이 지나치게 높음

*과적합 해결 방법
- 훈련 데이터가 적을 때는 데이터를 더 확보하거나 데이터 수를 인위적으로 늘리는 데이터 확장을 이용하면 효과적이다.
- 모델의 표현력이 지나치게 높을 때는 가중치 감소, 드롭아웃, 배치 정규화 등이 유효하다.

*드롭아웃
- 드롭아웃을 적용하려면 학습할 때와 테스트할 때의 처리 로직을 달리해야 한다.
- 학습 단계인지 테스트 단계인지 구별하는 구조를 만든다.

<드롭아웃>

* 드롭아웃

- 뉴런을 임의로 삭제하면서 학습하는 방법
- 학습 시에는 은닉층 뉴련을 무작위로 골라 삭제한다.
- 삭제된 뉴런은 신호를 전송하지 않는다.
- 학습 데이터를 흘려보낼 때마다 삭제할 뉴런을 무작위로 선택한다.

* 드롭아웃 코드 설명

```
mask = np.random.rand(10) > dropout_ratio
y = x * mask

scale = 1 - dropout_ratio
y = x * scale
```

- mask는 원소가 True 혹은 False인 배열이다.
- mask를 만드는 방법은 0.0 ~ 1.0 사이의 값을 임의로 10개 생성한다.
- 각 원소의 값은 dropout_ratio 0.6과 비교한다.
(dropout_ratio보다 큰 원소는 True, 그렇지 않은 원소는 False로 변환, 생성한 mask는 False 비율이 평균적으로 60%가 된다.)
- y = x * mask는 mask에서 값이 False인 원소에 대응하는 x의 원소를 0으로 설정한다.(삭제)
- 결과적으로 매회 평균 4개의 뉴런만이 출력을 다음 층으로 전달한다.
- 드롭아웃 계층은 학습시 데이터를 흘려보낼 때마다 이와 같은 선별적 비활성화를 수행한다.
- 테스트할 때에는 모든 뉴런을 사용하면서도 앙상블 학습처럼 유사하게 동작해야 한다.
- 모든 뉴런을 써서 출력을 계산하고, 그 결과를 약화시킨다.

- 약화하는 비율은 학습 시에 살아남은 뉴런의 비율이다.(학습 시 평균 40%의 뉴런이 생존)
- 테스트 할 때는 모든 뉴런을 사용하여 계산한 출력에 0.4를 곱한다.

<역 드롭아웃>

* 역 드롭아웃
- 스케일 맞추기를 학습할 때 수행한다.
- 스케일을 맞추기 위해 테스트할 때 scale을 곱한다.
- 학습할 때 미리 뉴런의 값에 1/scale을 곱해두고, 테스트 때는 아무런 동작도 하지 않는다.
- 테스트 시 아무런 처리도 하지 않기 때문에 테스트 속도가 조금 향상된다.
- 추론 처리만을 이용하는 경우에 바람직한 특성이다.
- 역 드롭아웃은 학습할 때 dropout_ratio를 동적으로 변경할 수 있다.
- 많은 딥러닝 프레임워크에서 역 드롭아웃 방식을 채용하고 있다.

```
scale = 1 - dropout_ratio
mask = np.random.rand(*x.shape) > dropout_ratio
y = x * mask / scale

y = x
```

<테스트 모드 추가>

* 드롭아웃은 학습단계와 테스트단계의 구분이 필요

- 역전파 비활성 모드 (with dezero.no_grad():) 방식을 유용하게 활용할 수 있다.
- Config 클래스 주변에 코드를 추가한다.

```
class Config:
    enable_backprop = True
    train = True

@contextlib.contextmanager
def using_config(name, value):
    old_value = getattr(Config, name)
    setattr(Config, name, value)
    try:
        yield
    finally:
        setattr(Config, name, old_value)

def test_mode():
    return using_config('train', False)
```

<드롭아웃 구현>

* 드롭아웃 코드 분석

- x는 Variable 인스턴스 또는 ndarray 인스턴스이다.
- 쿠파이의 ndarray 인스턴스의 경우도 고려하여 xp = cuda.get_array_module(x)를 사용한다.

```
def dropout(x, dropout_ratio = 0.5):
    x = as_variable(x)

    if dezero.Config.train:
        xp = cuda.get_array_module(x)
        mask = xp.random.rand(*x.shape) > dropout_ratio
        scale = xp.array(1.0 - dropout_ratio).astype(x.dtype)
        y = x * mask / scale
        return y
    else:
        return x
```

- F.dropout 함수를 사용할 수 있다.
- 과적합 발생 시 드롭아웃을 사용한다.

```
x = np.ones(5)
print(x)

y = F.dropout(x)
print(y)

with test_mode():
    y = F.dropout(x)
    print(y)
```

```
[1. 1. 1. 1. 1.]
variable([0. 2. 0. 0. 0.])
variable([1. 1. 1. 1. 1.])
```

<CNN 메커니즘>

* CNN(Convolutional Neural Network)

- 합성곱 신경망은 이미지인식, 음성인식, 자연어 처리 등 다양한 분야에서 사용된다.
- 이미지 인식용 딥러닝이라고 하면 대부분 CNN 기반이다.

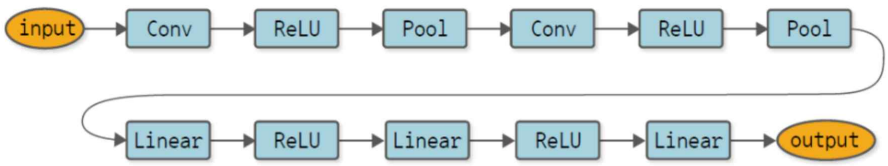
* CNN의 가장 중요한 구성 요소는 합성곱 층

- 첫 번째 합성곱 층의 뉴런은 입력 이미지의 모든 픽셀에 연결되는 것이 아니라 합성곱 층 뉴런의 수용장 안에 있는 픽셀에만 연결된다.
- 두 번째 합성곱 층에 있는 각 뉴런은 첫 번째 층의 작은 사각 영역 안에 위치한 뉴런에 연결된다.

* 사각 형태의 국부수용장을 모방한 CNN 층

<CNN 신경망의 구조>

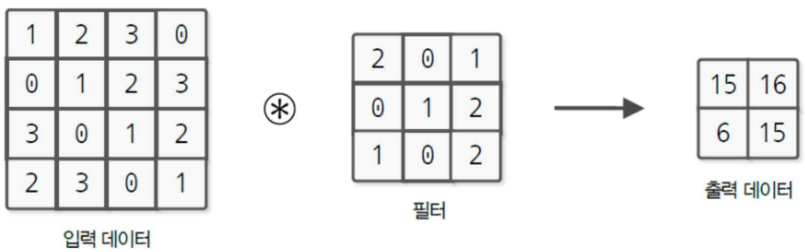
* CNN 구조



- 계층을 조합하여 만든다.
- 합성곱 층과 풀링 층(pooling layer)을 사용한다.
- Conv 계층과 Pool 계층이 새로 추가된다.
- 'Linear -> ReLU' 연결이 'Conv -> ReLU -> (Pool)'로 대체된다.
- 출력 층에서는 'Linear -> ReLU' 조합이 사용된다.

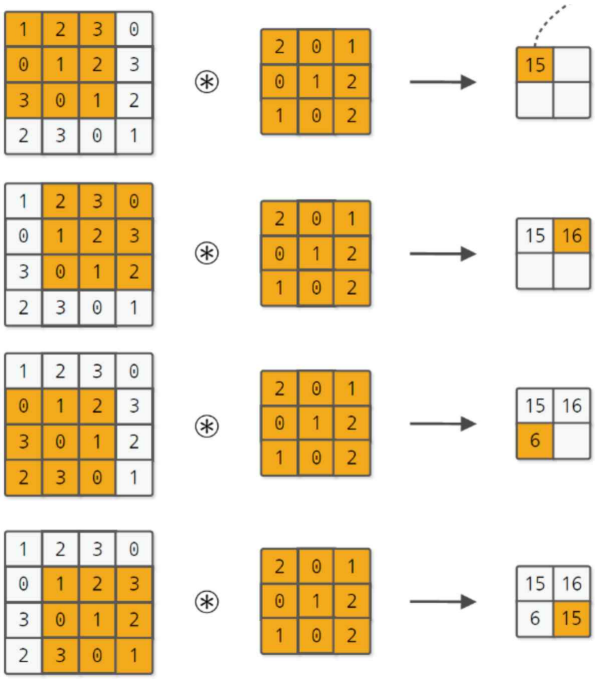
<합성곱 연산>

* 합성곱 연산



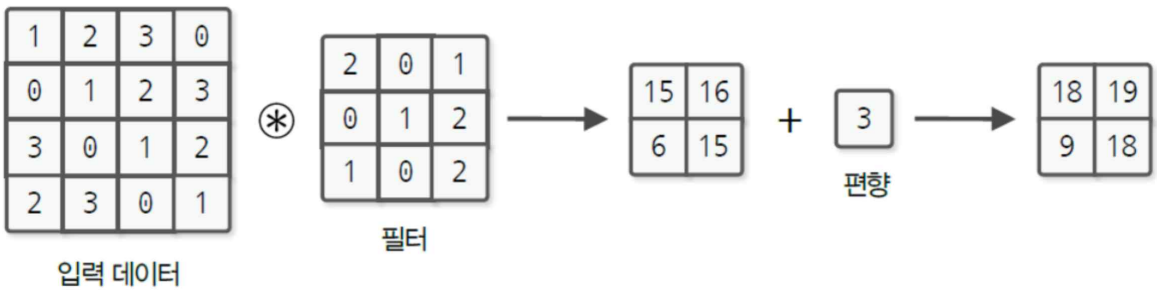
- CNN의 합성곱 층에서 수행하는 일이다.
- 이미지 처리에서 말하는 필터 연산에 해당된다.(필터를 문헌에 따라 커널이라고도 쓴다.)
- 합성곱 연산은 입력 데이터에 필터를 적용한다.
- 형상을 (높이, 너비) 순서로 표기하면, 입력 형상은 (4, 4), 필터는 (3, 3), 출력은 (2, 2)이다.

* 합성곱 연산 계산 순서



- 합성곱 연산은 입력 데이터에 대한 필터를 일정 간격으로 이동시키면서 적용한다.
- 필터와 입력의 해당 원소를 곱하여 총합을 구하고 그 결과를 해당 위치에 저장한다.
- 이 과정을 모든 장소에서 수행하면 합성곱 연산의 출력을 얻을 수 있다.

* 편향을 포함한 합성곱 연산의 처리 흐름



- 합성곱 층에도 편향이 존재한다.
- 편향은 필터링 후에 더해준다.
- 편향은 하나 뿐이다.
- 하나의 똑같은 값이 필터 적용 후의 모든 원소에 브로드캐스트 되어 더해진다.

<패딩>

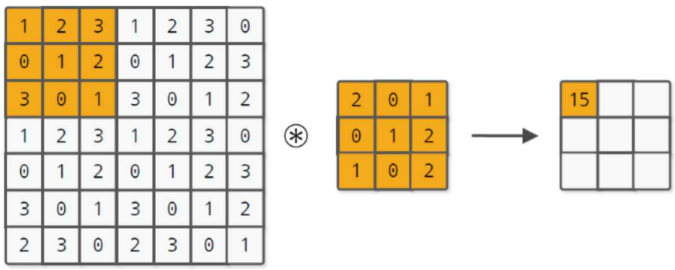
* 패딩(padding)

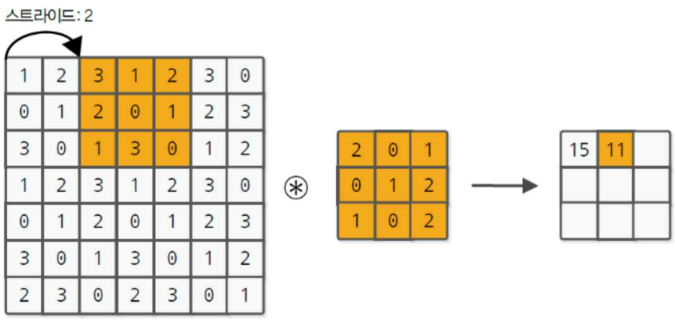
- 합성곱 층의 주요 처리 전에 입력 데이터 주위에 고정 값을 채운다.
- 패딩을 사용하는 이유는 출력 크기를 조정하기 위함이다.
- 형상이 (4, 4)인 입력 데이터에 폭이 1짜리 패딩을 적용하면 입력 데이터의 형상이 (6, 6)으로 변한다.
- (3, 3) 형상의 필터에 의해 (4, 4) 형상의 데이터가 출력된다.
- 패딩은 1이외에도 임의의 정수로 설정이 가능하다.
- 세로 방향 패딩과 가로 방향 패딩을 서로 다르게 설정할 수 있다.

<스트라이드>

* 스트라이드(stride)

- 필터를 적용하는 위치의 간격을 의미한다.





- 위 그림은 입력 크기가 (7, 7)인 데이터에 필터의 스트라이드를 2로 설정했을 때의 모습이다.
- 스트라이드를 2로 설정하면 출력 크기는 (3, 3)이 된다.
- 스트라이드는 세로 방향과 가로 방향 값을 다르게 설정할 수 있다.

<출력 크기 계산 방법>

* 출력 크기 계산

- 패딩 크기를 늘리면 출력 데이터의 크기가 커지고, 스트라이드를 크게 하면 반대로 작아진다.
- 출력 크기는 패딩과 스트라이드의 영향을 많이 받는다.
- 패딩과 스트라이드의 크기, 입력 데이터와 커널의 크기가 추어지면 출력 데이터의 크기가 결정된다.

```
def get_conv_outsize(input_size, kernel_size, stride, pad):
    return (input_size + pad * 2 - kernel_size) // stride + 1

H, W = 4, 4
KH, KW = 3, 3
SH, SW = 1, 1
PH, PW = 1, 1

OH = get_conv_outsize(H, KH, SH, PH)
OW = get_conv_outsize(W, KW, SW, PW)
print(OH, OW)
```

- input_size는 입력 데이터의 크기, kernel_size는 커널의 크기, strid는 스트라이드의 크기, pad는 패딩의 크기이다.

<CNN 메커니즘>

* 2차원 데이터에서의 합성곱 연산

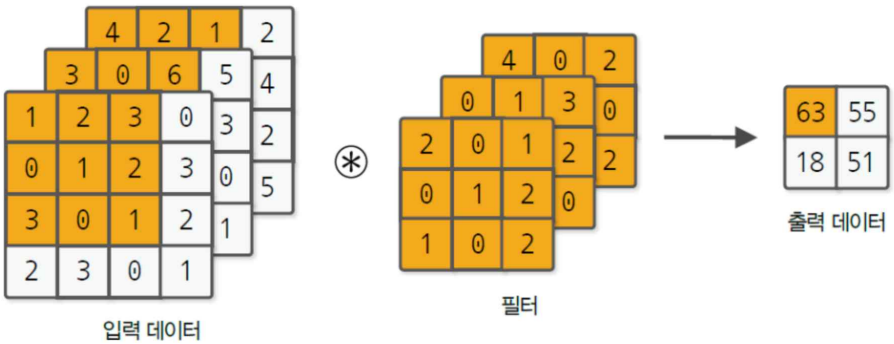
- 수직 및 수평 방향으로 늘어선 합성곱 연산
- 가로/세로 방향

* 3차원 데이터(3차원 텐서)에서의 합성곱 연산

- 사진에는 가로/세로 방향뿐 아니라 RGB처럼 채널 방향으로도 데이터가 쌓여있다.
- 이미지 처리를 위해 3차원 데이터를 다뤄야 한다.

<3차원 텐서>

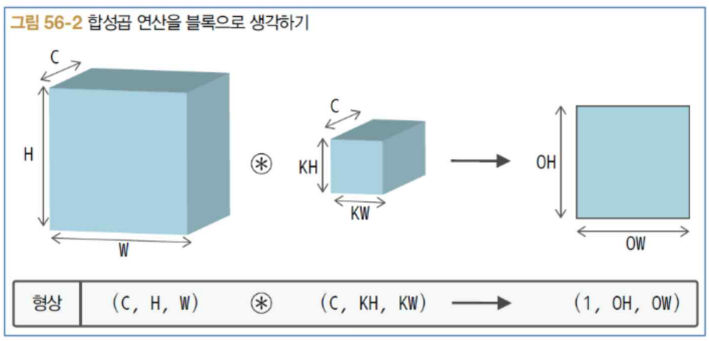
* 3차원 텐서의 합성곱 연산

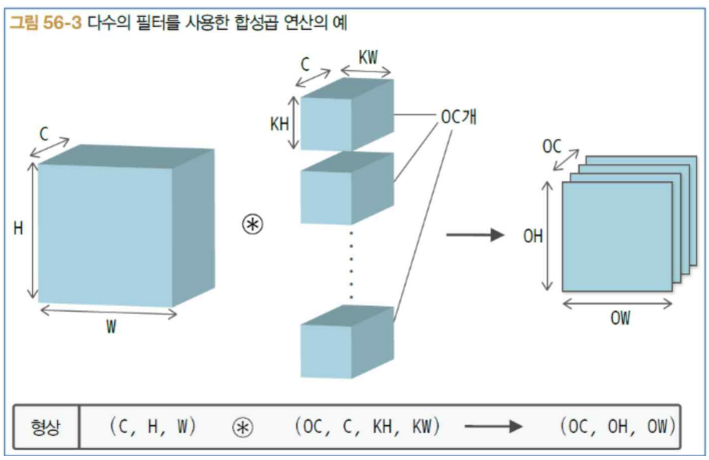


- 위 그림은 채널이 3개인 데이터로 수행하는 합성곱 연산이다.
- 깊이 방향으로 데이터가 늘어난 것을 제외하면 필터가 움직이는 방법도 계산이 같다.
- 입력 데이터와 필터의 채널수를 똑같이 맞춰주어야 한다.(모두 3개임)
- 필터의 가로, 세로 크기는 원하는 숫자로 설정할 수 있다.
- 2차원 합성곱 연산은 대부분의 딥러닝 프레임워크에서 Conv2d라는 이름으로 제공한다.

<블록으로 생각하기>

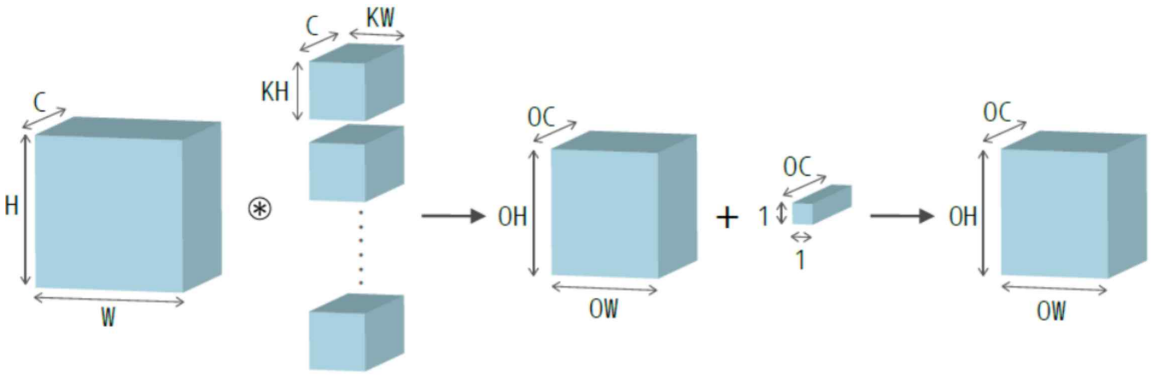
* 합성곱 연산을 블록으로 표현





- 데이터가 (채널, 높이, 너비) 순서로 정렬되었다면 데이터의 형상은 (C, H, W)로 표기하고 필터는 (C, KH, KW)로 표기한다.
- 출력은 특징 맵이라고 불린다.
- 특징 맵을 채널 방향으로 여러장을 가지려면 다수의 필터를 사용하면 된다.
- OC** 개의 필터를 개별적으로 적용한다.
- 출력의 특징 맵도 OC개가 생성된다.
- OC개의 맵을 모아(OC, OH, OW)형상의 블록을 만든다.

* 편향을 포함한 합성곱 연산 처리



(C, H, W) ⊛ (OC, C, KH, KW) ⟶ (OC, OH, OW) + (OC, 1, 1) ⟶ (OC, OH, OW)

- 편향은 채널당 하나의 값만 갖는다.
- 편향의 형상은 (OC, 1, 1)이 되고, 필터 적용 후의 출력은 (OC, OH, OW)가 된다.
- 편향은 형상이 다르기 때문에 브로드캐스트 된 다음에 더해진다.

<미니배치 처리>

* 합성곱 연산의 미니배치 처리

- 미니배치 처리를 위해서는 각 층을 흐르는 데이터를 4차원 텐서로 취급한다.
- n개의 데이터로 이루어진 미니배치 합성곱 연산을 수행한다.
- 데이터의 맨 앞에 배치를 위한 차원이 추가된다.
- 데이터를 (batch_size, channel, height, width) 형상으로 정렬된다.
- 4차원 텐서의 샘플 데이터 각각에 대해 (독립적으로) 똑같은 합성곱 연산을 수행한다.

<풀링 층>

*풀링 층

- 풀링은 가로, 세로 공간을 작게 만드는 연산이다.
- Max 풀링(최대 풀링)은 최댓값을 취하는 연산이며, 2 x 2는 대상 영역의 크기를 나타낸다.
- 일반적으로 풀링 윈도우 크기와 스트라이드 크기는 같은 값으로 설정한다.(위도우가 3 x 3이면 스트라이드는 3, 4 x 4면 스트라이드를 4로 설정)

*풀링 층의 주요 특징

- 학습하는 매개변수가 없다.
  대상의 영역에서 최댓값(or 평균값) 처리만 하면 끝이다.
- 채널 수가 변하지 않는다.
  계산이 독립적으로 이루어진다.
- 미세한 위치 변화에 영향을 덜 받는다.
  입력 데이터의 차이가 크지 않으면 풀링 결과가 크게 달라지지 않는다.
  입력 데이터의 미세한 차이에 강건하다.
  예를 들어 오른쪽으로 1원소만큼 어긋나 있더라도 출력은 달라지지 않는다.