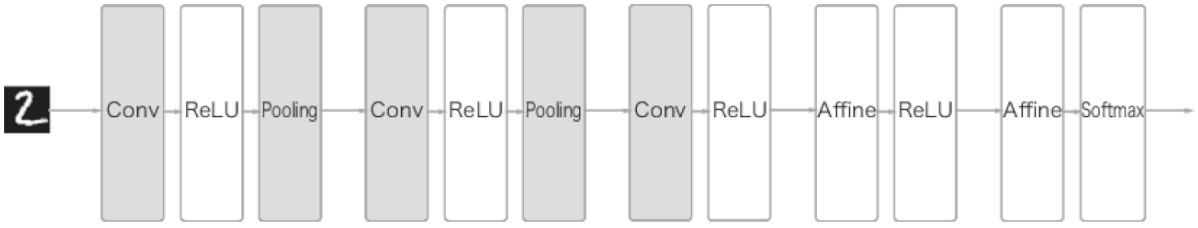
밑바닥부터 시작하는 딥러닝 CHAPTER 7

전체 구조

CNN에서는 새로운 합성곱 계층과 풀링 계층이 추가된다. CNN 계층은 conc-relu-pooling 흐름으로 연결된다. 여기서 중요한 것은 출력에 가까운 층에서는 affine-relu 구성을 사용할 수 있다.

합성곱 계층

CNN에서는 패딩, 스트라이드 등 CNN 고유의 용어가 등장한다. 그리고 각 계층 사이에 3차원 데이터같이 입체적인 데이터가 흐른다는 점에서 완전연결 신경망과 다르다.



완전연결 계층의 문제점

완전연결계층에서는 인접하는 계층의 뉴런이 모두 연결되고 출력의 수는 임의로 정할 수 있다. 하지만, 완전연결계층은 데이터의 형상이 무시된다는 문제점이 있다. 예를 들면, 이미지는 3차원 데이터(가로, 세로, 색상)이지만 완전연결계층에 입력할 때는 1차원 데이터로 바꿔줘야 한다. 그렇기 때문에 형상에 담긴 정보를 살릴 수 없다.

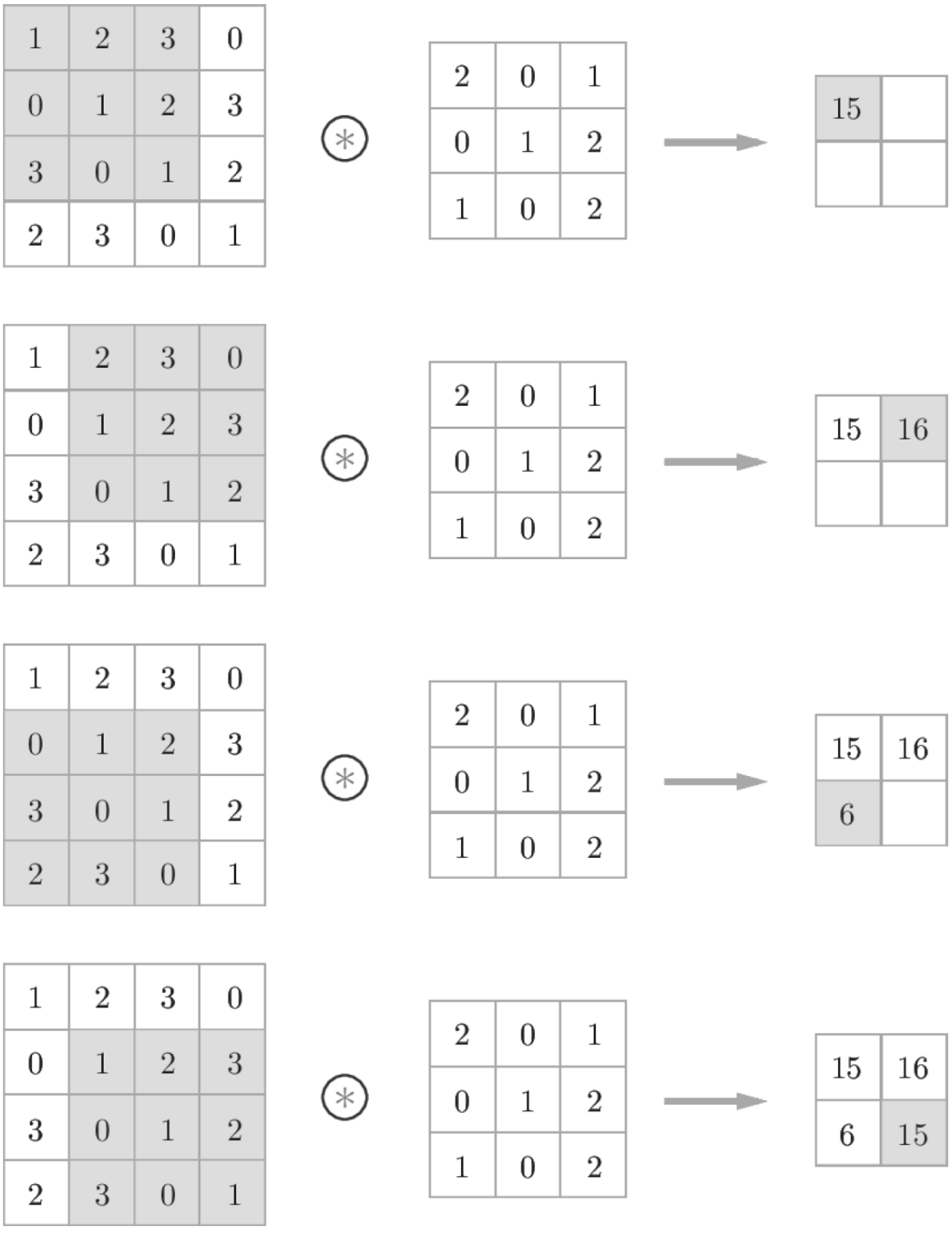
하지만, 합성곱 계층은 형상을 유지한다. 이미지도 3차원 데이터로 입력받아서 형상을 가진 데이터로 제대로 이해할 수 있다.

CNN에서는 합성곱 계층의 입출력 데이터를 특징맵이라 하고, 입력 데이터를 입력 특징 맵, 출력 데이터를 출력 특징 맵이라고 한다.

7.2.2 합성곱 연산

합성곱 연산은 이미지 처리에서 말하는 필터 연산이다. 입력 데이터에 필터를 적용한다.

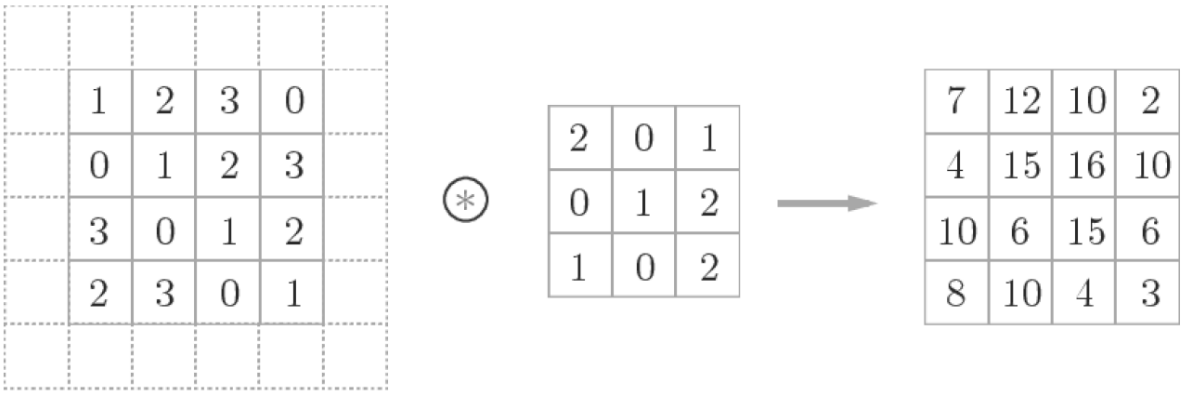
필터의 윈도우를 일정 간격으로 이동해 가면서 입력 데이터에 적용한다. 윈도우는 회색 부분을 말한다. 입력과 필터에 대응하는 원소끼리 곱한 후에 그 총합을 구한다.



CNN에서SMS 필터의 매개변수가 가중치에 해당하고 편향(고정값) 또한 존재할 수 있다. 필터를 적용한 후에 편향을 더한다. 편향은 항상 하나만 존재한다.

패딩

합성곱 연산을 수행하기 전에 입력 데이터 주변을 0과 같은 특정 값으로 채우기도 하는 것을 패딩이라고 한다. 패딩은 주로 출력 크기를 조정할 목적으로 사용한다.



(4, 4)
입력 데이터(패딩 : 1)

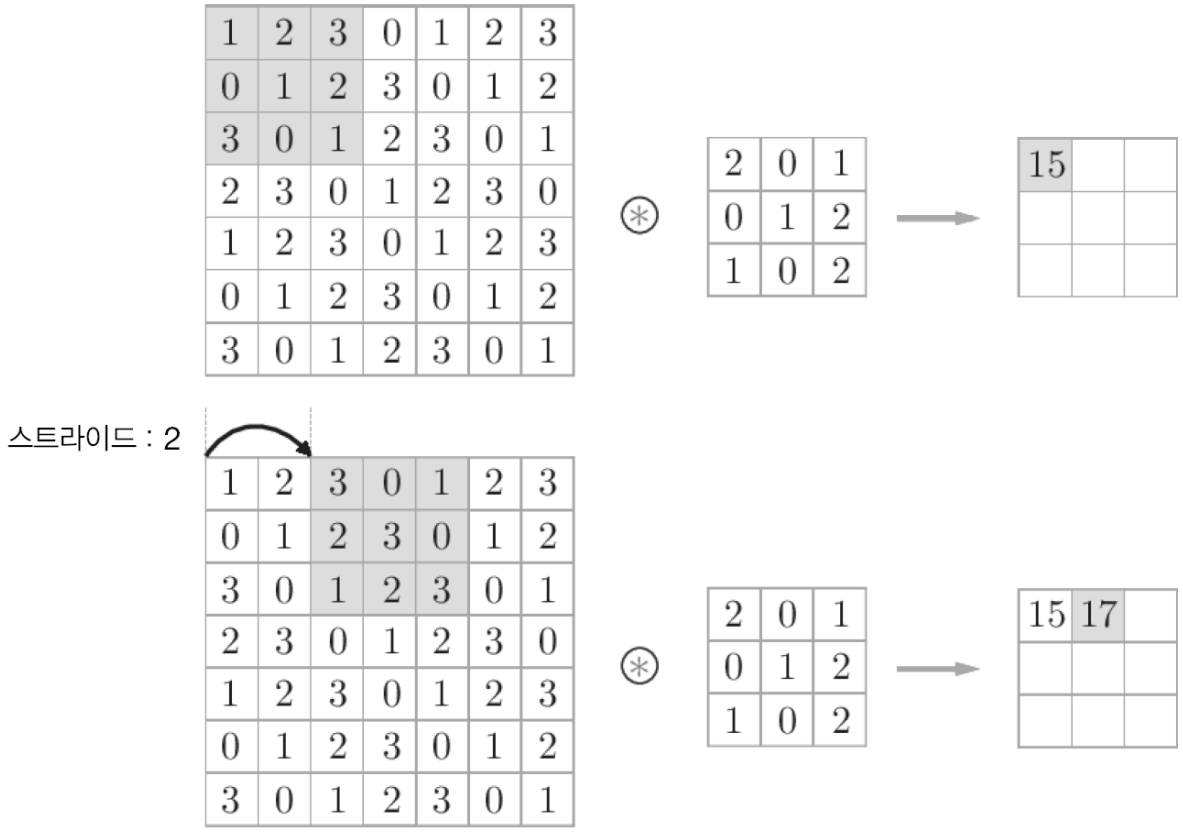
(3, 3)
필터

(4, 4)
출력 데이터

이 그림을 보면 패딩이 추가되어서 입력 데이터가 (6,6)이 된다. 이 입력에 (3,3) 필터를 걸면 (4,4) 출력 데이터가 만들어 진다.

스트라이드

필터를 적용하는 위치의 간격을 스트라이드라고 한다. 위의 예는 모두 스트라이드가 1이었지만, 스트라이드를 2로 하면 윈도우가 두 칸씩 이동한다.



스트라이드를 키우면 출력 크기는 작아지고 패딩을 크게 하면 출력 크기가 커진다. 이를 수식화 하면, 다음과 같다.

입력 크기를 (H,W), 필터 크기를 (FH, FW), 출력 크기를 (OG, OW), 패딩을 P, 스트라이드를 S라고 하면 출력의 크기는

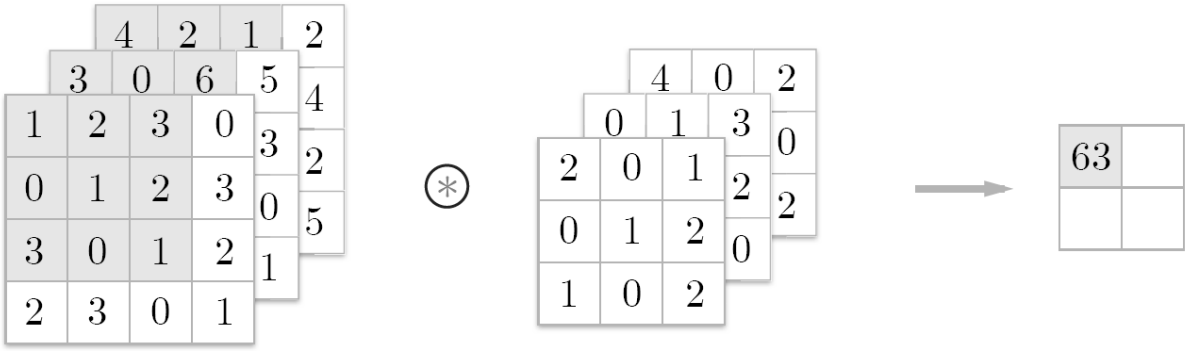
OH=(H+2P-FH)/S+1

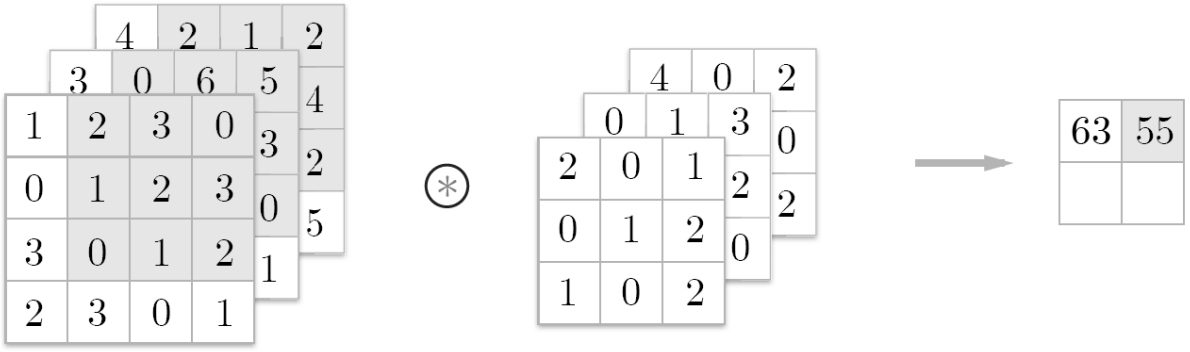
OW=(W+2P-FW)/S+1

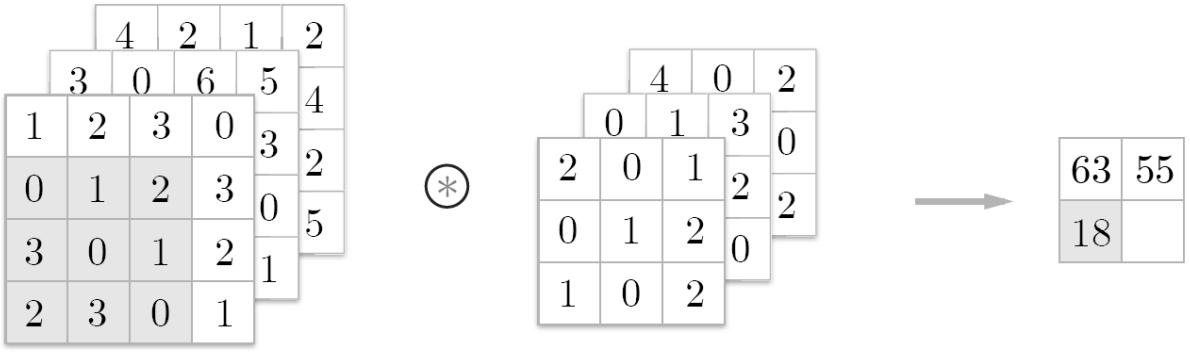
딥러닝 프레임워크 중에는 값이 딱 나눠떨어지지 않을 때는 가장 가까운 정수로 반올림한다.

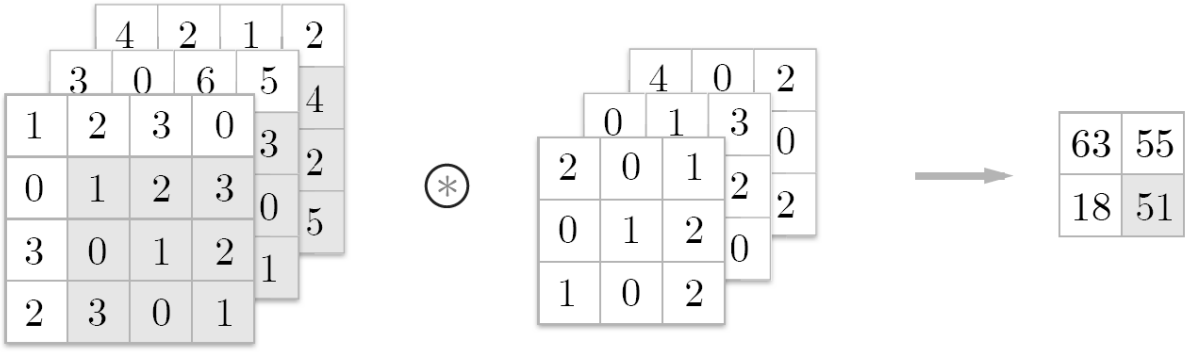
7.2.5 3차원 데이터의 합성곱 연산

입력 데이터와 필터의 합성곱 연산을 채널마다 수행하고 그 결과를 더해서 하나의 출력을 얻는다.


4 2 1 2
3 0 6 5 4
1 2 3 0 3 2
0 1 2 3 0 5
3 0 1 2 1
2 3 0 1
⊛
4 0 2
0 1 3 0
2 0 1 2 2
0 1 2 0
1 0 2
63


4 2 1 2
3 0 6 5 4
1 2 3 0 3 2
0 1 2 3 0 5
3 0 1 2 1
2 3 0 1
⊛
4 0 2
0 1 3 0
2 0 1 2 2
0 1 2 0
1 0 2
63 55


4 2 1 2
3 0 6 5 4
1 2 3 0 3 2
0 1 2 3 0 5
3 0 1 2 1
2 3 0 1
⊛
4 0 2
0 1 3 0
2 0 1 2 2
0 1 2 0
1 0 2
63 55
18


4 2 1 2
3 0 6 5 4
1 2 3 0 3 2
0 1 2 3 0 5
3 0 1 2 1
2 3 0 1
⊛
4 0 2
0 1 3 0
2 0 1 2 2
0 1 2 0
1 0 2
63 55
18 51

이 때 주의할 점은 입력 데이터의 채널수와 필터의 채널수가 같아야 한다는 것이다. 이 예에서는 3개로 같다.

블록으로 생각하기

3차원 합성곱 연산을 직육면체 블록이라고 생각하면 쉽다.

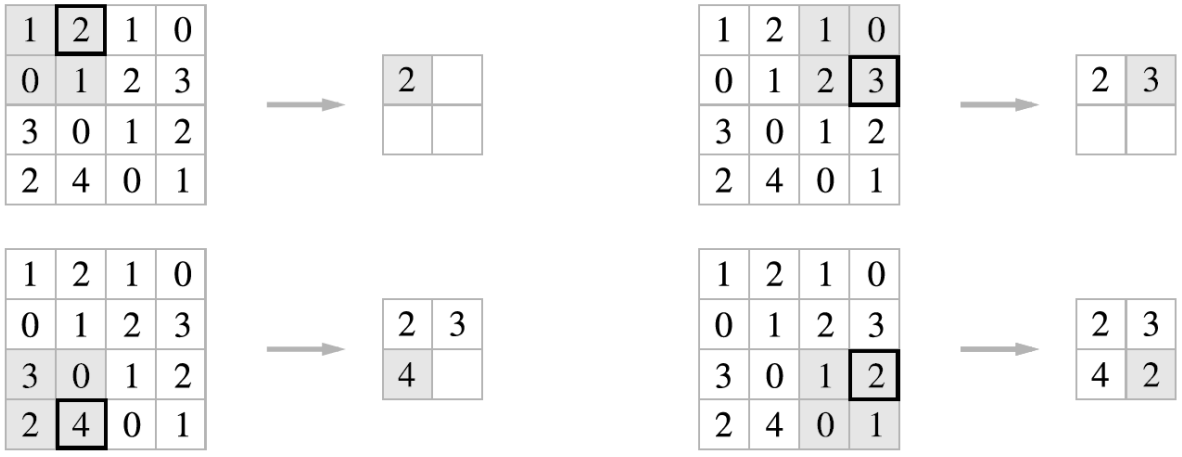
배치 처리

신경망 처리에서는 입력 데이터를 한 덩어리로 묶어 배치로 처리했다.

합성곱 연상에서는 각 계층을 흐르는 데이터의 차원을 하나 늘려 (데이터 수, 채널수, 높이, 너비) 4차원 데이터로 저장한다. 데이터는 4차원 형상을 가진 채 각 계층을 타고 흐른다. 4차원 데이터가 하나 흐를 때마다 데이터 n개에 대한 합성곱 연산이 이뤄진다. 즉, N회분의 처리를 한번에 수행한다.

풀링 계층

풀링은 세로, 가로 방향의 공간을 줄이는 연산이다. 최대 풀링은 대상 영역 중에서 최댓값을 구하는 것이다. 설정한 스트라이드의 크기 대로 윈도우를 설정하고 그 간격으로 이동한다.



이 그림을 보면 2x2 최대 풀링을 스트라이드 2로 처리하는 것이다. 2x2 크기의 영역에서 가장 큰 원소 하나를 꺼내고 스트라이드 2로 설정했으므로 2x2 윈도우가 2칸 간격으로 이동한다.

풀링 계층의 특징

학습해야 할 매개변수가 없다

1. 풀링은 대상 영역에서 최댓값이나 평균을 취하는 명확한 처리이므로 학습할 것이 없다.

2. 채널수가 변하지 않는다.

3. 입력의 변화에 영향을 적게 받는다.

7.4 합성곱/풀링 계층 구현하기

7.4.1 4차원 배열

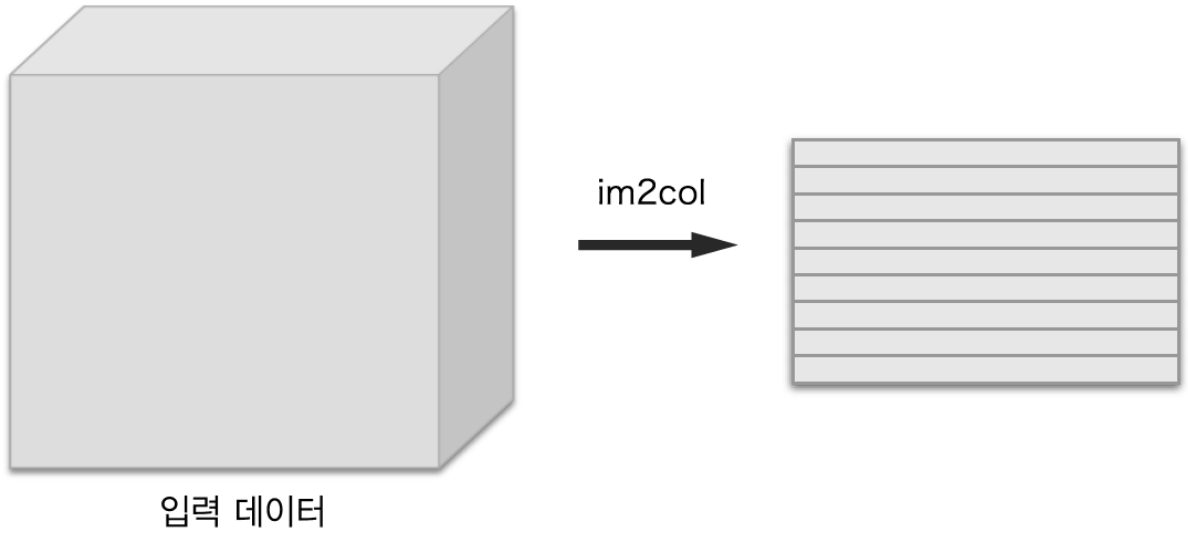
- 데이터 형상이 높이 28, 너비 28, 채널 1개, 데이터 10개를 구현한 것이다.

```
import numpy as np
x=np.random.rand(10, 1, 28, 28)
x.shape
```

```
x[0].shape
x[1].shape
x[0,0] # 첫번째 데이터의 첫 채널의 공간 데이터
```

7.4.2 im2col로 데이터 전개하기

넘파이에 for 문을 사용하면 성능이 떨어지기 때문에 im2col이라는 편의 함수를 이용해본다. im2col은 입력 데이터를 필터링하기 좋게 전개하는 함수이다.



im2col을 사용하면 원소의 수가 원래 블록의 원소 수보다 많아 져서 메모리를 많이 소비한다. 하지만 컴퓨터는 큰 행렬의 곱셈을 빠르게 계산할 수 있기 때문에 im2col을 이용한다. im2col로 데이터를 전개한 다음에는 합성곱 계층의 필터(가중치)를 1열로 전개하고 두 행렬의 내적을 계산하면 된다. 이렇게 출력한 결과는 2차원 행렬이다. CNN은 4차원 배열 데이터로 저장하기 때문에 reshape을 이용해서 4차원으로 변형한다.

합성곱 계층 구현하기

- im2col 함수를 구현해보면

```
def im2col(input_data, filter_h, filter_w, stride=1, pad=
0):
    """다수의 이미지를 입력받아 2차원 배열로 변환한다(평탄화).

    Parameters
    ----------
    input_data : 4차원 배열 형태의 입력 데이터(이미지 수, 채널 수,
높이, 너비)
    filter_h : 필터의 높이
    filter_w : 필터의 너비
    stride : 스트라이드
    pad : 패딩

    Returns
    -------
    col : 2차원 배열
    """
    N, C, H, W = input_data.shape
    out_h = (H + 2*pad - filter_h)//stride + 1
    out_w = (W + 2*pad - filter_w)//stride + 1

    img = np.pad(input_data, [(0,0), (0,0), (pad, pad), (pa
d, pad)], 'constant')
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_
w))

    for y in range(filter_h):
        y_max = y + stride*out_h
        for x in range(filter_w):
            x_max = x + stride*out_w
            col[:, :, y, x, :, :] = img[:, :, y:y_max:strid
e, x:x_max:stride]

    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*o
```

```
ut_w, -1)
    return col
```

- 합성곱 계층을 구현한 것이다.

```
class Convolution:
    def __inti__(self, w, b, stride=1, pad=0):
        self.w=w
        self.b=b
        self.stride=stride
        self.pad

    def forward(self, x):
        fn, c, fh, fw=self.w.shape
        n,c,h,w=x.shape
        out_h=int(1+(h+2*self.pad-fh)/self.stride)
        out_w=int(1+(w+2*self.pad-fw)/self.stride)

        col=im2col(x, fh, fw, self.stride, self.pad)
        col_w=self.w.reshape(fn, -1).T
        out=np.dot(col,col_w)+self.b

        out=out.reshpae(n, out_h, out_w, -1).transpose(0,3,
1,2)

        return out
```

- 필터는 (FN, C, FH, FW)의 4차원 형상이다. FN은 필터 개수, C는 채널, FH는 필터 높이, FW는 필터 너비이다.

col=im2col(x, fh, fw, self.stride, self.pad))

col_w=self.w.reshape(fn, -1).T

out=np.dot(col,col_w)+self.b

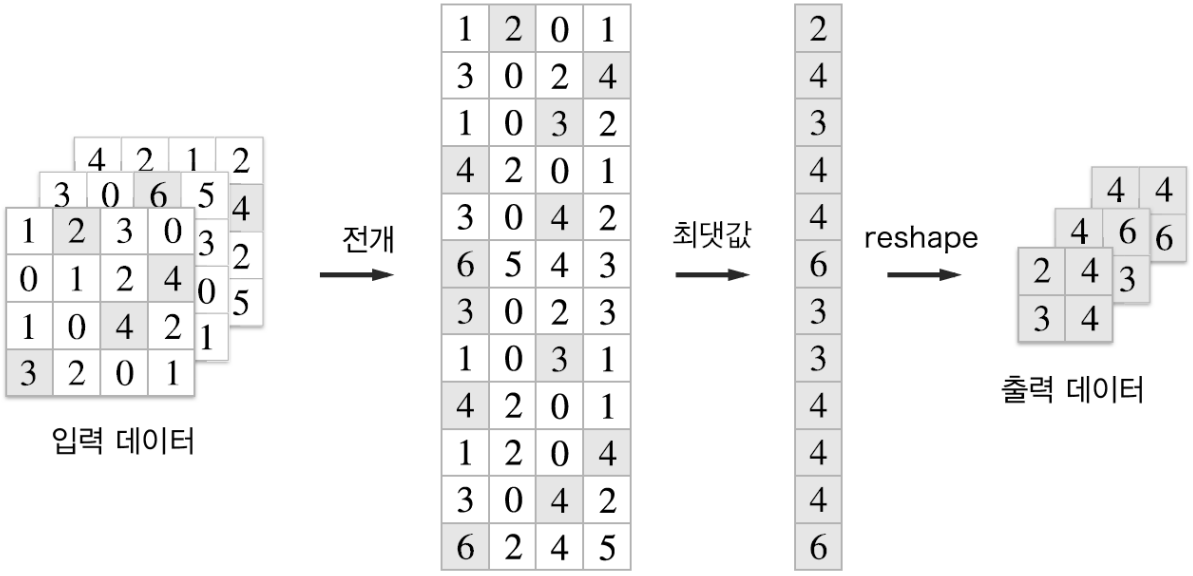
이 부분에서 입력 데이터를 im2col로 전개하고 필터도 reshape를 사용해 2차원 배열로 전개한다. 이렇게 전개한 두 행렬의 내적을 구한다. 각 필터 블록을 1줄로 펼쳐 세우
고 reshpae에 -1을 지정해서 다차원 배열의 원소 수가 변환 후에도 똑같이 유지되도록 적절히 묶어준다.

- 마지막 출력 데이터를 적절한 형상으로 바꿔주는 것이 forward 구현의 마지막이다.

풀링 계층 구현하기

채널이 독립절이가는 점이 합성곱 계층때와는 다르다.

위와 같이 전개한 후 행별 최댓값을 구한후에 적절한 형상으로 성형한다.



풀링 계층 구현은 세단계로 진행한다.

- 입력 데이터를 전개한다.
- 행별 최댓값을 구한다.
- 적절한 모양으로 성형한다.

```
class Pooling:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        self.pool_h=pool_h
        self.pool_w=pool_w
        self.stride=stride
        self.pad=pad

    def forward(self, x):
        n, c, h, w=x.shape
        out_h=int(1+(h-self.pool_h)/self.stride)
        out_w=int(1+(w-self.pool_w)/self.stride)
```

```
        col=im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)  #전개
        col=col.reshape(-1, self.pool_h*self.pool_w)

        out=np.max(col, axis=1) #최댓값

        out=out.reshape(n, out_h, out_w, c).transpose(0, 3, 1, 2)

        return out
```

7.5 CNN구현하기

CNN 네트워크는 `Convolution-ReLU-Pooling-Affine-ReLU-Affine-Softmax` 순으로 흐른다.

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import pickle
import numpy as np
from collections import OrderedDict
from common.layers import *
from common.gradient import numerical_gradient


class SimpleConvNet:
    """단순한 합성곱 신경망

    conv - relu - pool - affine - relu - affine - softmax

    Parameters
    ----------
    input_size : 입력 크기 (MNIST의 경우엔 784)
    hidden_size_list : 각 은닉층의 뉴런 수를 담은 리스트 (e.g. [100, 100, 100])
    output_size : 출력 크기 (MNIST의 경우엔 10)
    activation : 활성화 함수 - 'relu' 혹은 'sigmoid'
    weight_init_std : 가중치의 표준편차 지정 (e.g. 0.01)
```

```
        'relu'나 'he'로 지정하면 'He 초깃값'으로 설정
        'sigmoid'나 'xavier'로 지정하면 'Xavier 초깃값'으로 설정
    """
    def __init__(self, input_dim=(1, 28, 28),
                 conv_param={'filter_num':30, 'filter_siz
e':5, 'pad':0, 'stride':1},
                 hidden_size=100, output_size=10, weight_in
it_std=0.01):
        filter_num = conv_param['filter_num']
        filter_size = conv_param['filter_size']
        filter_pad = conv_param['pad']
        filter_stride = conv_param['stride']
        input_size = input_dim[1]
        conv_output_size = (input_size - filter_size + 2*fi
lter_pad) / filter_stride + 1
        pool_output_size = int(filter_num * (conv_output_si
ze/2) * (conv_output_size/2))
        # 가중치 초기화
        self.params = {}
        self.params['W1'] = weight_init_std * \
                            np.random.randn(filter_num, inp
ut_dim[0], filter_size, filter_size)
        self.params['b1'] = np.zeros(filter_num)
        self.params['W2'] = weight_init_std * \
                            np.random.randn(pool_output_siz
e, hidden_size)
        self.params['b2'] = np.zeros(hidden_size)
        self.params['W3'] = weight_init_std * \
                            np.random.randn(hidden_size, ou
tput_size)
        self.params['b3'] = np.zeros(output_size)
        # 계층 생성
        self.layers = OrderedDict()
        self.layers['Conv1'] = Convolution(self.params['W
1'], self.params['b1'],
                                           conv_param['stri
de'], conv_param['pad'])
        self.layers['Relu1'] = Relu()
```

```
        self.layers['Pool1'] = Pooling(pool_h=2, pool_w=2,
stride=2)
        self.layers['Affine1'] = Affine(self.params['W2'],
self.params['b2'])
        self.layers['Relu2'] = Relu()
        self.layers['Affine2'] = Affine(self.params['W3'],
self.params['b3'])

        self.last_layer = SoftmaxWithLoss()
```

초기화를 마친 후에는 추론을 수행하는 predict메서드와 손실함수의 값을 구하는 loss 메서드를 구현할 수 있다.

```
def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

def loss(self, x, t):
        """손실 함수를 구한다.
        Parameters
        ----------
        x : 입력 데이터
        t : 정답 레이블
        """
        y = self.predict(x)
        return self.last_layer.forward(y, t)
```

x는 입력데이터, t는 정답 레이블이다. 추론을 수행할 때는 초기화 때 layers에 추가한 계층을 맨 앞에서부터 차례로 forward 메서드를 호출해서 다음 계층에 결과를 전달한다.

손실함수는 predict 메서드의 결과를 인수로 마지막 층의 forwoard 메서드를 호출한다.

오차역전파법으로 기울기를 구하는 구현

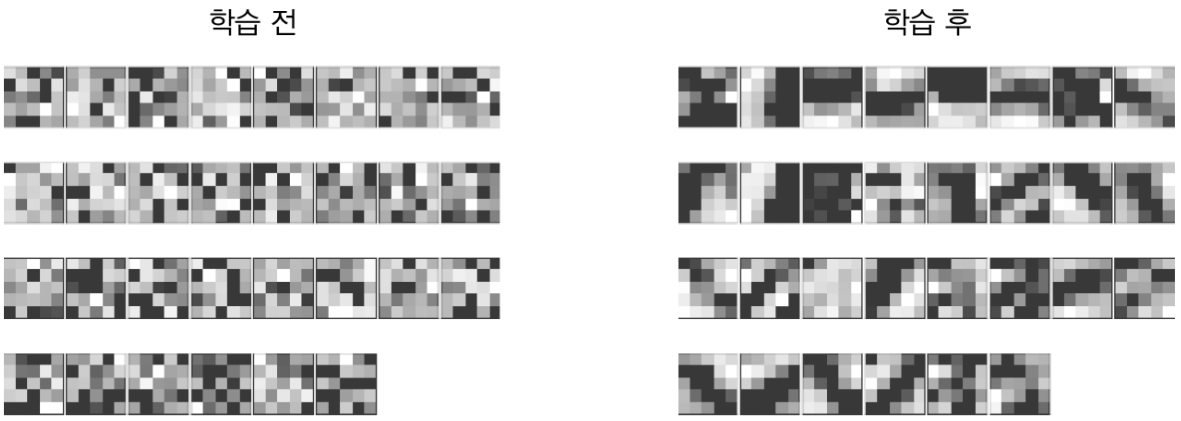
```
 def gradient(self, x, t):
        """기울기를 구한다(오차역전파법).
        Parameters
        ----------
```

```
        x : 입력 데이터
        t : 정답 레이블
        Returns
        -------
        각 층의 기울기를 담은 사전(dictionary) 변수
            grads['W1']、grads['W2']、... 각 층의 가중치
            grads['b1']、grads['b2']、... 각 층의 편향
        """
        # forward
        self.loss(x, t)

        # backward
        dout = 1
        dout = self.last_layer.backward(dout)

        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 결과 저장
        grads = {}
        grads['W1'], grads['b1'] = self.layers['Conv1'].dW,
self.layers['Conv1'].db
        grads['W2'], grads['b2'] = self.layers['Affine1'].d
W, self.layers['Affine1'].db
        grads['W3'], grads['b3'] = self.layers['Affine2'].d
W, self.layers['Affine2'].db

        return grads
```

CNN 시각화하기

1번째 층의 가중치 시각화하기

- 합성곱 계층 필터를 이미지로 나타내보자.

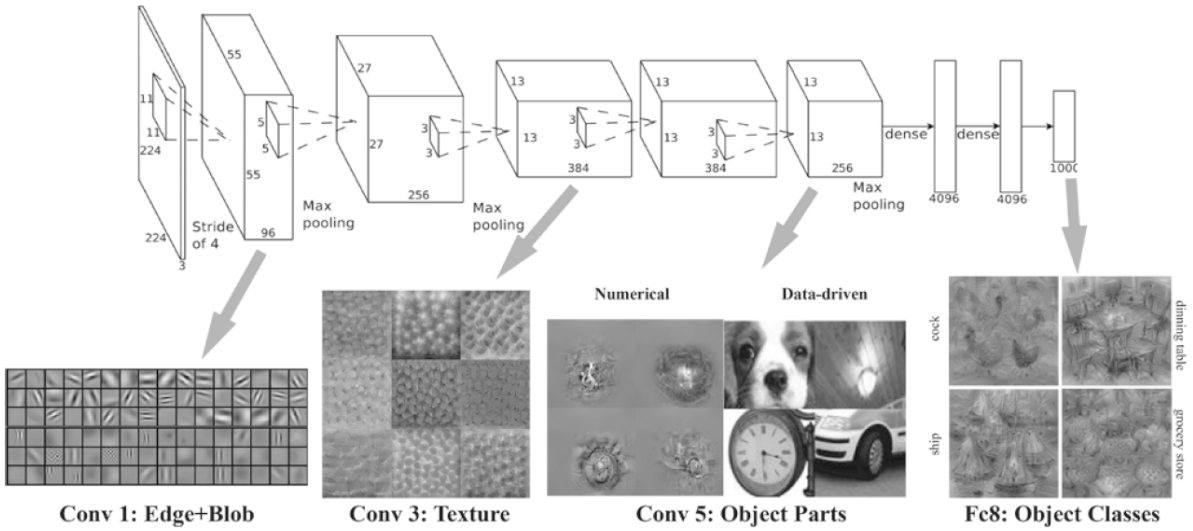


학습 전 필터는 무작위로 초기화되고 있어 흑백의 정도에 규칙성이 없다. 학습을 마친 필터는 규칙이 있는 이미지가 되었다.

- 흰색에서 검은색으로 변화하고 덩어리를 가지기도 하는 규칙을 나타낸다. 규칙성
- 이 있는 필터는 색상이 바뀐 경계선인 에지와 덩어리진 영역인 블롭을 보고 있다.
- 이렇게 합성곱 계층의 필터는 에지나 블롭 등 원시적인 정보를 추출할 수 있다.이런 원시적인 정보가 뒷단 계층에 전달되는 것이 CNN에서 일어나는 일이다.

7.6.2 층 깊이에 따른 추출 정보 변화

계층이 깊어질수록 추출되는 정보는 더 추상화된다.



위의 그림은 일반 사물 인식을 수행한 8층의 CNN이다. 이 네트워크는 AlexNet이라고 한다. 합성곱 계층과 풀링 계층을 여러 겹 쌓고, 마지막으로 완전연결 계층을 거쳐 결과를 출력한다. 1층은 에지와 블롭, 3층은 텍스쳐, 5층은 사물의 일부, 마지막은 사물의 클래스에 뉴런이 반응한다.

층이 깊어질수록 뉴런이 고급 정보에 반응한다. 즉, 사물의 의미를 이해하도록 변화한다.

7.7 대표적인 CNN

7.7.1 LeNet

손글씨 숫자를 인식하는 네트워크이다. 합성곱 계층과 풀링 계층을 반복하고 마지막으로 완전연결 계층을 거치면서 결과를 출력한다.

LeNet과 현재의 CNN과의 차이점

- LeNex는 시그모이드 함수를 사용하고 현재는 ReLU를 사용한다.
- LeNex는 서브샘플링을 하여 중간데이터 크기가 작아지지만 현재는 최대 풀링이 주류다.

7.7.2 AlexNet

합성곱 계층과 풀링 계층을 거듭하며 마지막으로 완전연결 계층을 사용하지만, 다음과 같은 변화가 있다

- 활성화 함수로 ReLU를 이용한다
- LRN이라는 국소적 정규화를 실시하는 계층을 이용한다.
- 드롭아웃을 사용한다.