



7장 합성곱 신경망(CNN)

0827 랩세미나 이상윤



목차

7.1 전체 구조

7.2 합성곱 계층

7.3 풀링 계층

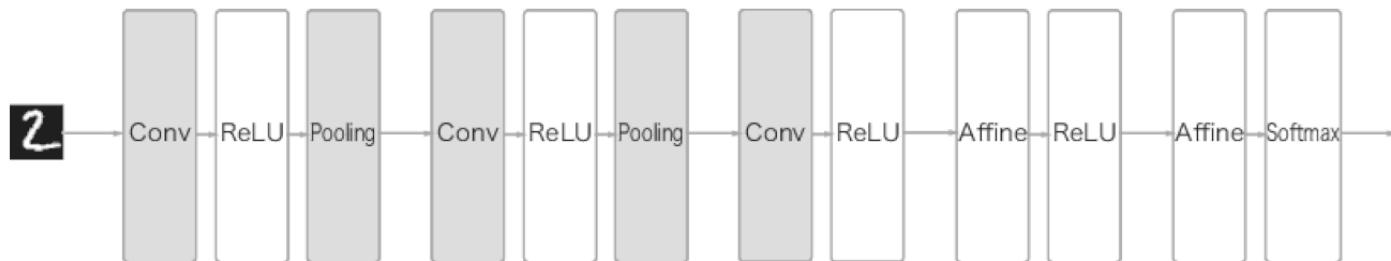
7.4 합성곱/ 풀링 계층 구현


7.5 CNN구현

전체 구조

완전연결 신경망 : 신경망이 인접하는 계층의 모든 뉴런과 결합되어있는 구조

CNN : 합성곱 계층과 풀링 계층이 추가된 구조





완전연결 계층의 문제점과 합성곱 연산

이미지 데이터 : 3차원 형상 (채널*가로*세로)

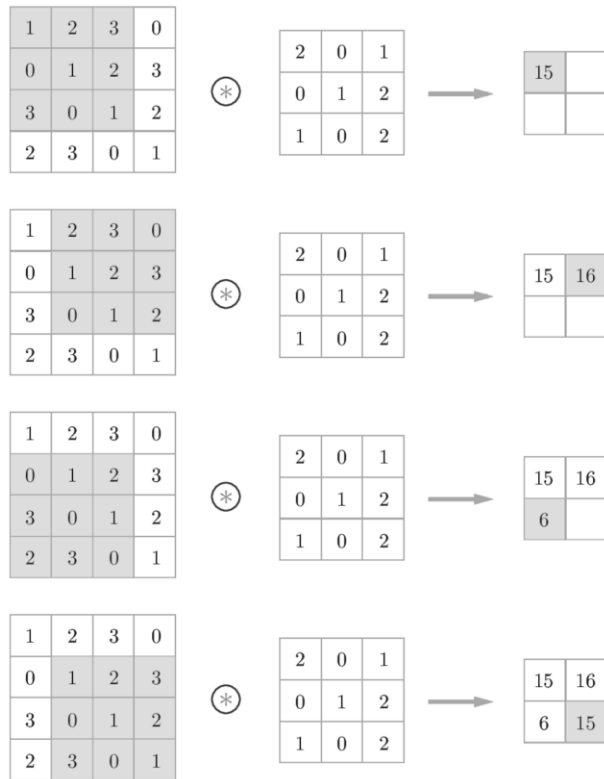
데이터 형상의 무시 :

- 입력 데이터를 무조건 1차원으로 평탄화하므로 이미지 같은 3차원 데이터를 제대로 이해할 수 없게 된다.
- 큰 이미지를 입력으로 넣을 경우 가중치의 개수가 매우 많아진다.

합성곱 계층은 형상을 유지하고 이미지도 형상 그대로 이해하게 된다.

합성곱 연산

- 합성곱 연산은 필터 연산으로 입력 데이터에 필터를 적용한다.
- 필터 윈도우를 일정 간격으로 이동하며 입력 데이터에 적용하여 계산한다.
- 합성곱 연산에서도 필터가 존재하는데 이는 각 값에 편향 값을 더해주면 된다. (편향은 항상 1x1만 존재)
- 입출력 데이터 = 특징 맵



패딩

합성곱 연산을 수행하기 전 입력 데이터 주변을 특정 값으로 채우는 것.

출력 크기를 조정할 수 있다.

원본 (4,4) 데이터에 (3,3) 필터 적용시 (2,2)의 결과가 나오지만, 패딩으로 입력 데이터가 (6,6)이 되자 출력 데이터가 (4,4)가 나오는 모습이다.

	1	2	3	0
	0	1	2	3
	3	0	1	2
	2	3	0	1

(4, 4)

입력 데이터(패딩 : 1)

⊗

2	0	1
0	1	2
1	0	2

(3, 3)

필터



7	12	10	2
4	15	16	10
10	6	15	6
8	10	4	3

(4, 4)

출력 데이터

스트라이드

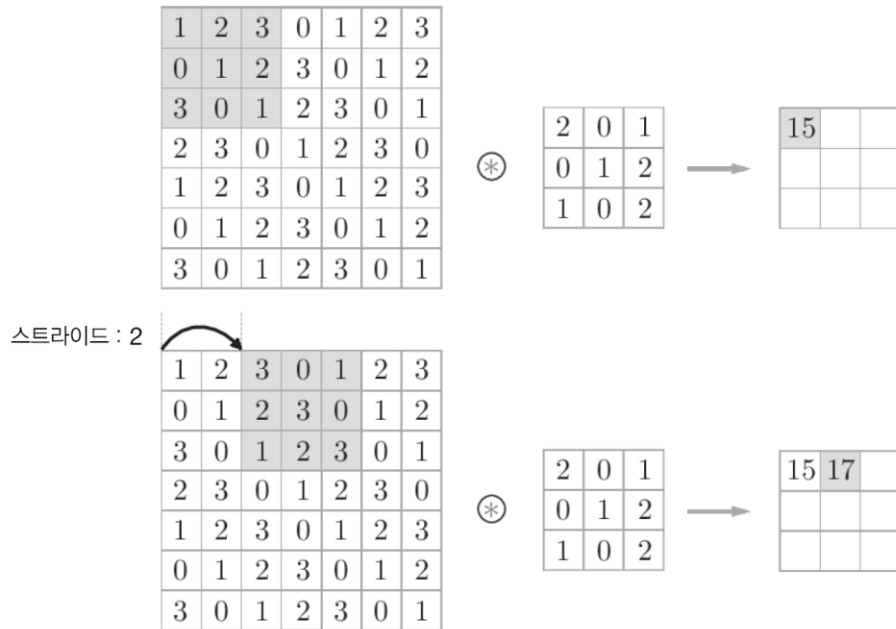
- 필터를 적용하는 위치의 간격

수식으로 살펴보면,

입력 크기를 (H, W) , 필터 크기를 (FH, FW) , 출력 크기를 (OH, OW) , 패딩을 P , 스트라이드를 S 라고 하면 출력의 크기는

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

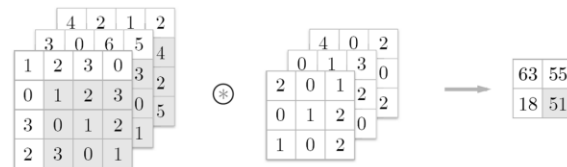
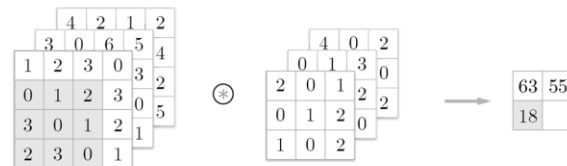
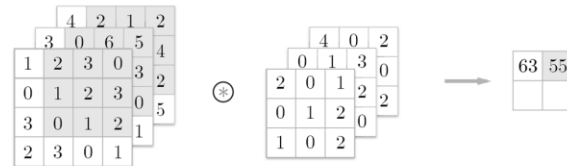
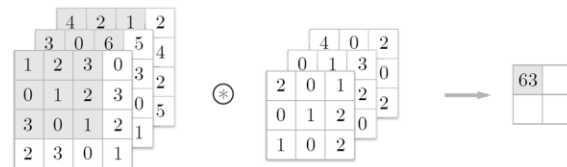
$$OW = (W + 2P - FW) / S + 1$$



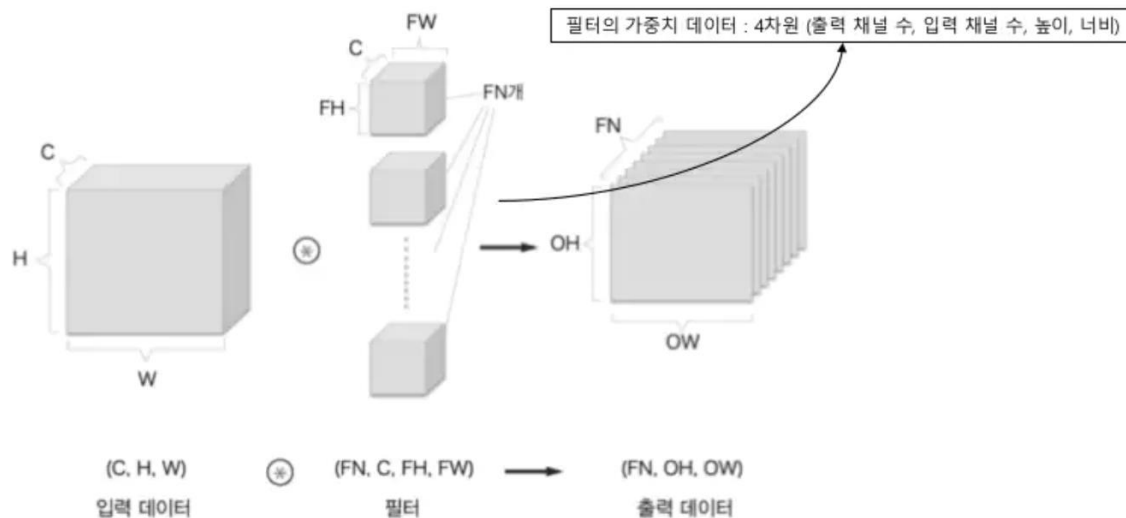
3차원 데이터의 합성곱 연산

채널 수 만큼 필터의 채널 수를 늘려 각 채널에 필터 연산을 해준다.

이때 출력 데이터는 한 장의 특징 맵이다.



출력 데이터를 다수의 채널로 내보내려면?



필터(가중치)를 다수 사용하면 앞선 특징 맵들이 모여 직육면체 형태의 3차원 데이터가 생성된다.

풀링 계층

1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



2	

1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



2	3
4	

1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



2	3

1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1



2	3
4	2

가로, 세로의 공간을 줄이는 연산

그림의 2x2 최대 풀링은 2x2의 영역에서 가장 큰 원소를 꺼내고 스트라이드 (2칸) 만큼 이동하여 다시 값을 추출하는 것



풀링 계층의 특징

1. 학습해야 할 매개변수가 없다.
 - 최대값이나 평균을 구하는 것으로 학습할 것이 없다.
1. 채널 수가 변하지 않는다.
 - 가로, 세로의 크기만 줄일 뿐 채널의 수는 변하지 않는다.
1. 입력의 변화에 영향을 적게 받는다.
 - 입력 데이터가 조금 변하더라도 결과는 크게 바뀌지 않는다.



합성곱/풀링 계층 구현하기

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

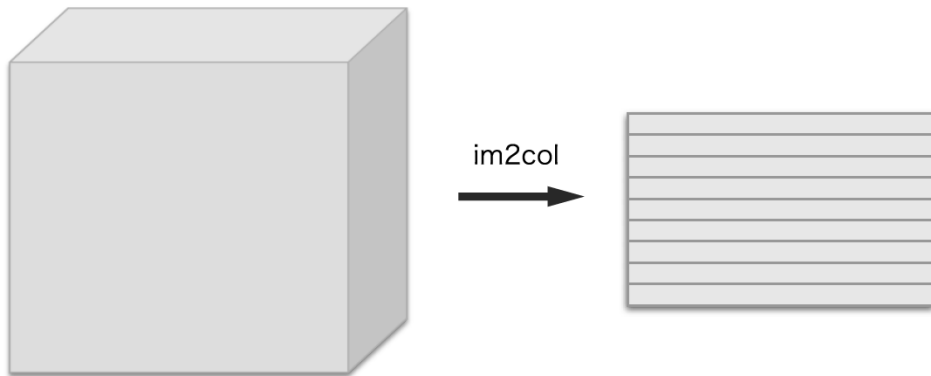
높이 28, 너비 28, 채널 1개, 데이터 10개를 넘파이로 구현하고 합성곱 연산을 적용하려면, for 문을 겹겹이 써야한다.

이를 간소화하기 위해 im2col 이라는 함수를 사용하자.

image to column (im2col)

입력 데이터를 필터링하기 좋게 전개하는 함수

원소의 수가 원래 블록의 원소 수보다 많아져서 메모리를 많이 소비한다. 하지만 연산속도가 빠르므로 시간면에서 유리하다.



입력 데이터

이 후, 합성곱 계층의 필터(가중치)를 1열로 전개하고 두 행렬의 곱을 계산한다.

im2col 함수의 구현

```
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), (pad, 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:stride, x:x_max:stride]  
  
    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1)  
    return col
```

합성곱 계층 구현

```
class Convolution:
    def __init__(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.reshape(n, out_h, out_w, -1).transpose(0,3,1,2)

    return out
```

풀링 계층 구현하기

		4	2	1	2
	3	0	6	5	
1	2	3	0	3	
0	1	2	4	0	
1	0	4	2	1	
3	2	0	1		

입력 데이터

전개
→

1	2	0	1
3	0	2	4
1	0	3	2
4	2	0	1
3	0	4	2
6	5	4	3
3	0	2	3
1	0	3	1
4	2	0	1
1	2	0	4
3	0	4	2
6	2	4	5

최댓값
→

2
4
3
4
4
6
3
3
4
4
4
6

reshape
→

		4	4
	4	6	6
2	4	3	
3	4		

출력 데이터

1. 입력 데이터를 전개한다.
2. 행 별 최댓값(혹은 평균값)을 구한다.
3. 출력 모양으로 바꿔준다.

풀링 계층 구현

```
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)

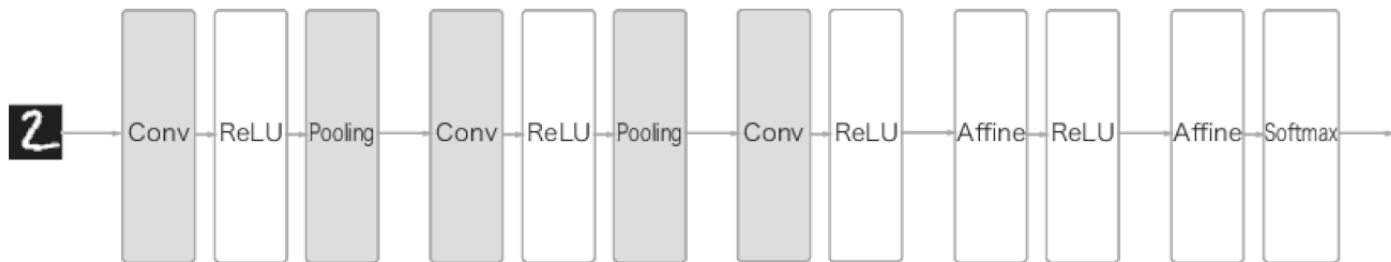
        out=np.max(col, axis=1) #최댓값

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

        return out
```

CNN 구현하기

CNN 네트워크 => "Conv-ReLU-Pooling-Affline-ReLU-Affine-Softmax"



```

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
    output_size : 출력 크기 (MNIST의 경우엔 10)
    activation : 활성화 함수 - 'relu' 혹은 'sigmoid'
    weight_init_std : 가중치의 표준편차 지정
    """
    def __init__(self, input_dim=(1, 28, 28),
                  conv_param={'filter_num':30, 'filter_size':5, 'pad':0, 'stride':1},
                  hidden_size=100, output_size=10, weight_init_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*filter_pad) / filter_stride + 1
        pool_output_size = int(filter_num * (conv_output_size/2) * (conv_output_size/2))

```

```
# 가중치 초기화
```

```
self.params = {}
```

```
self.params['W1'] = weight_init_std * \
    np.random.randn(filter_num, input_dim[0], filter_size, filter_size)
```

```
self.params['b1'] = np.zeros(filter_num)
```

```
self.params['W2'] = weight_init_std * \
    np.random.randn(pool_output_size, hidden_size)
```

```
self.params['b2'] = np.zeros(hidden_size)
```

```
self.params['W3'] = weight_init_std * \
    np.random.randn(hidden_size, output_size)
```

```
self.params['b3'] = np.zeros(output_size)
```

```
# 계층 생성
```

```
self.layers = OrderedDict()
```

```
self.layers['Conv1'] = Convolution(self.params['W1'], self.params['b1'],
    conv_param['stride'], 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()
```

```

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)
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)

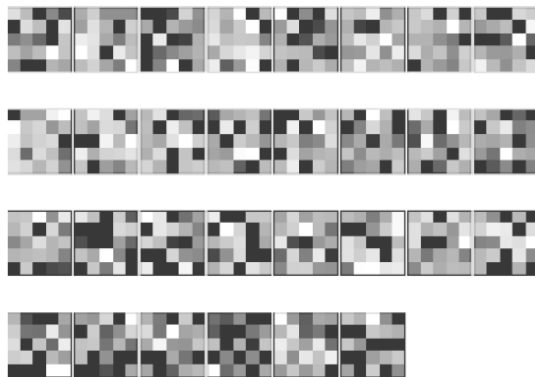
    # 결과 저장
    grads = {}
    grads['W1'], grads['b1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db
    grads['W2'], grads['b2'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
    grads['W3'], grads['b3'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

    return grads

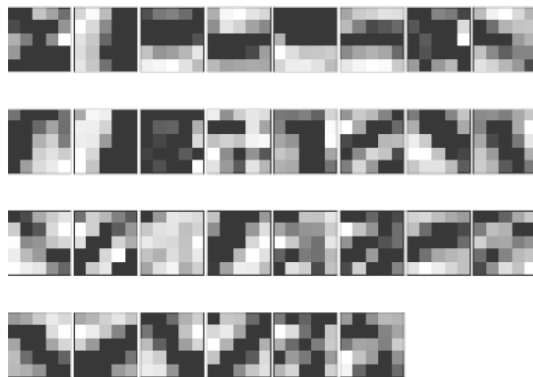
```

CNN 시각화하기

학습 전



학습 후



학습 전 필터는 무작위로 초기화되고 있어 흑백의 정도에 규칙성이 없으나, 학습을 마친 필터는 규칙성이 있는 이미지를 보여준다.



정리

CNN은 완전연결 계층 네트워크에 합성곱 계층과 풀링 계층을 추가한다.

CNN을 통해 이미지와 같은 3차원 데이터를 구조적 손실없이 학습할 수 있다.

im2col 함수를 이용하여 합성곱 계층과 풀링 계층을 쉽게 구현할 수 있다.

CNN을 시각화해보면 계층이 깊어질수록 고급 정보가 추출됨을 볼 수 있다.