CNN

ゼロから作る Deep learning 7章

藤本 16.10.20

実装

4次元配列とim2col

```
>>> x = np.random.rand(10, 1, 28, 28) # ランダムにデータを生成
>>> x.shape
(10, 1, 28, 28)

>>> x[0].shape # (1, 28, 28)
>>> x[1].shape # (1, 28, 28)
```

- Numpyでは、for文を使うと処理が遅くなる！
- Im2col (image to column)を用いる（フィルター使用に都合良いように展開する）　メモリ食うけど、行列計算が速い

(例) 3次元->2次元

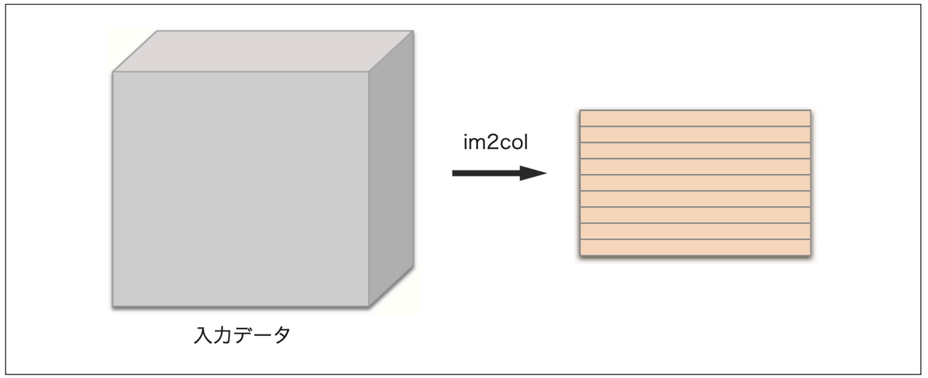


図 7-17　im2col の概略図

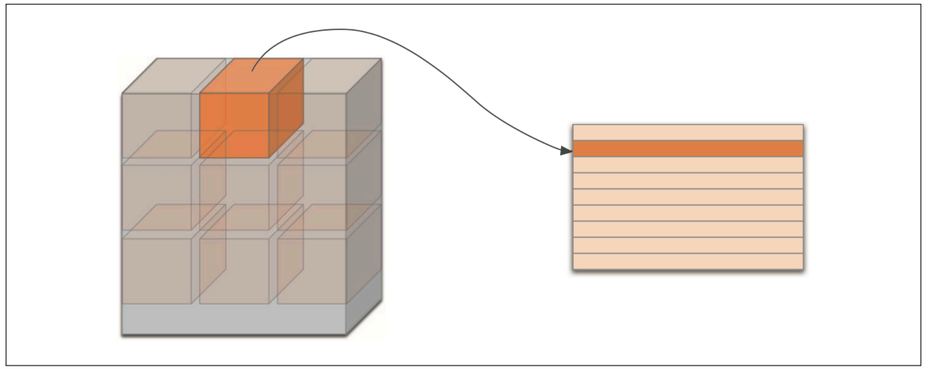


図 7-18　フィルターの適用領域を、先頭から順番に 1 列に展開する

4次元配列とim2col

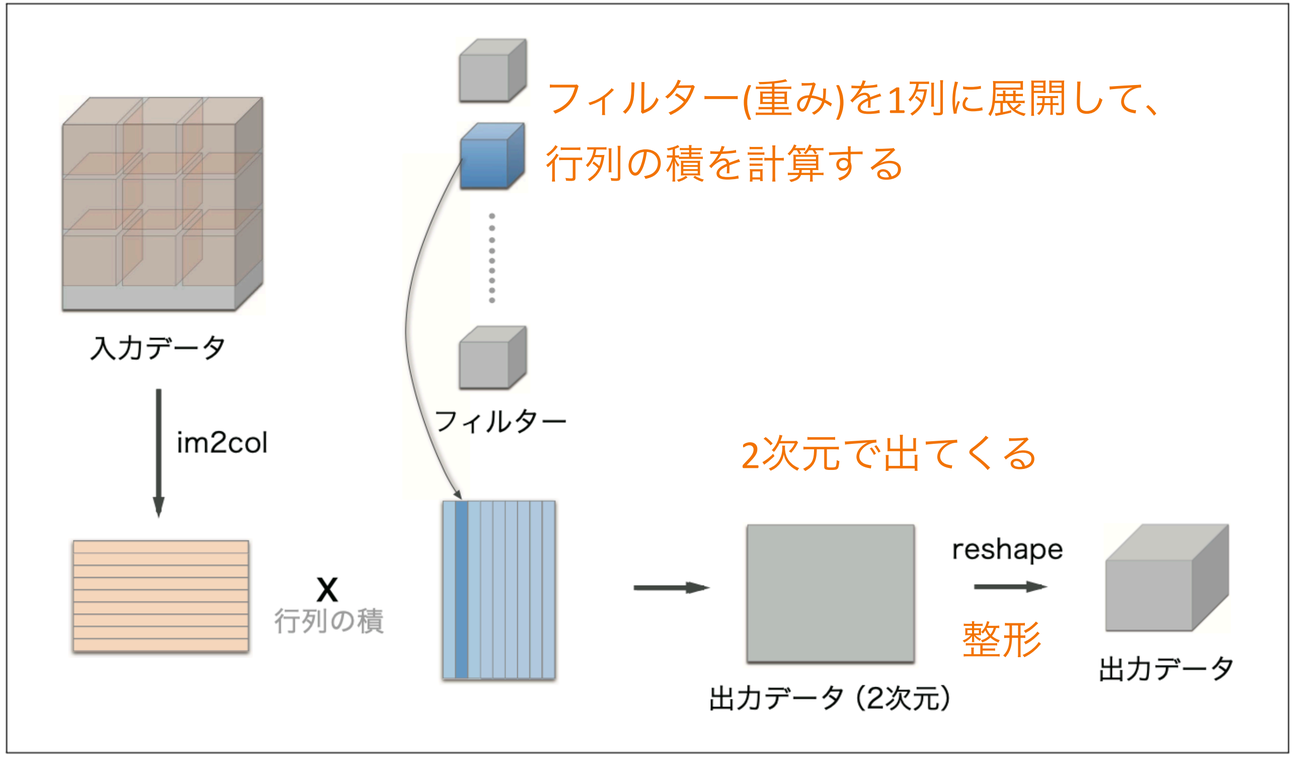


図7-19　畳み込み演算のフィルター処理の詳細：フィルターを縦方向に 1 列に展開して並べ、im2colで展開したデータと行列の積を計算する。最後に、出力データのサイズに整形（reshape）する

Convolutionレイヤ

```
im2col(input_data, filter_h, filter_w, stride=1, pad=0)
```

(データ数、チャンネル、高さ、横幅)の4次元配列　-> 2次元に展開する

```
import sys, os
sys.path.append(os.pardir)
from common.util import im2col
```

(1) バッチサイズ1

```
x1 = np.random.rand(1, 3, 7, 7)
col1 = im2col(x1, 5, 5, stride=1, pad=0)
print(col1.shape) # (9, 75)
```

(2) バッチサイズ10

```
x2 = np.random.rand(10, 3, 7, 7) # 10 個のデータ
col2 = im2col(x2, 5, 5, stride=1, pad=0)
print(col2.shape) # (90, 75)
```

Reshape (x,-1)
で要素数が辻褄があうように
してくれる

```
class Convolution:
    def __init__(self, W, b, stride=1, pad=0):
        self.W = W
        self.b = b
        self.stride = stride
        self.pad = 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 # フィルターの展開
        out = np.dot(col, col_W) + self.b

        out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)

        return out
```

(フィルター、バイアス、ストライド、パディング)

フィルター(4次元)

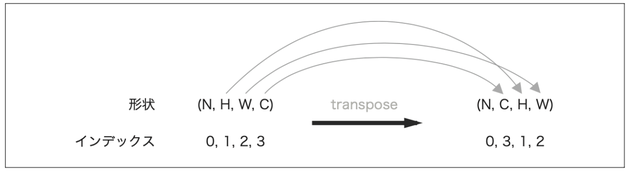


図7-20　NumPy の transpose による軸の順番の入れ替え：インデックス（番号）によって、軸の順番を変更する

ここまでforward処理, Affineレイヤとほぼ同じ実装にできる

この後逆伝播する。col2imを使う (im2colの逆)

Convolutionレイヤ

```
def forward(self, x):
    FN, C, FH, FW = self.W.shape
    N, C, H, W = x.shape
    out_h = 1 + int((H + 2*self.pad - FH) / self.stride)
    out_w = 1 + int((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)

    self.x = x
    self.col = col
    self.col_W = col_W

    return out

def backward(self, dout):
    FN, C, FH, FW = self.W.shape
    dout = dout.transpose(0,2,3,1).reshape(-1, FN)

    self.db = np.sum(dout, axis=0)
    self.dW = np.dot(self.col.T, dout)
    self.dW = self.dW.transpose(1, 0).reshape(FN, C, FH, FW)

    dcol = np.dot(dout, self.col_W.T)
    dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.pad)

    return dx
```

Poolingレイヤ

チャンネルごとに展開する

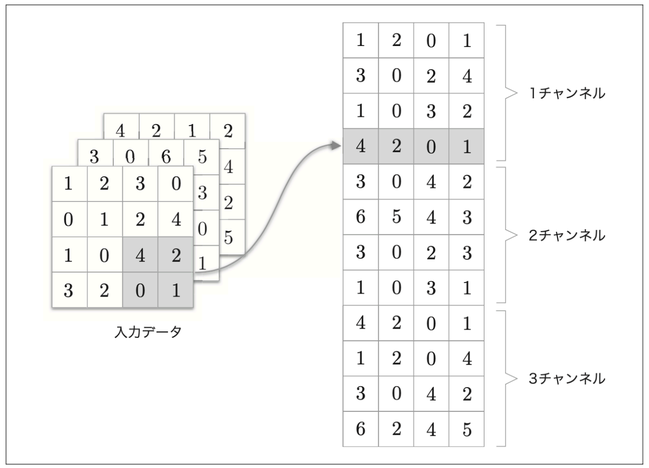


図7-21　入力データに対してプーリング適用領域を展開（2 × 2 のプーリングの例）

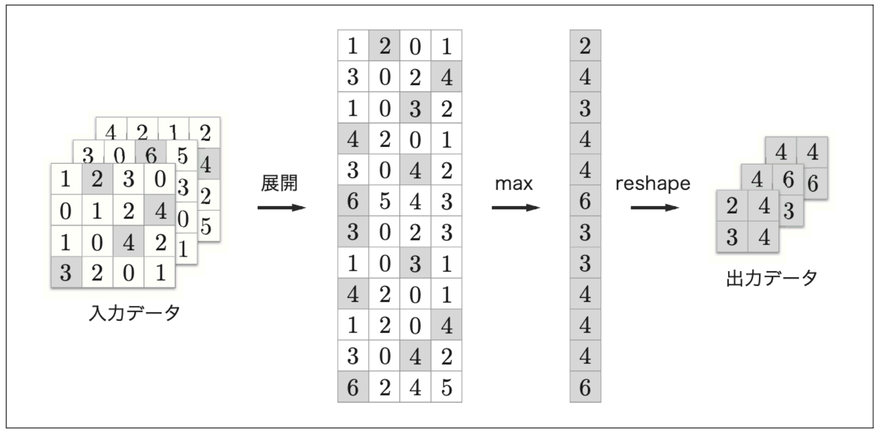


図7-22　Pooling レイヤの実装の流れ：プーリング適用領域内の最大値の要素は背景をグレーで描画

```
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)

        # 展開 (1)

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

        # 最大値 (2)
        out = np.max(col, axis=1)
        # 整形 (3)
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)

        return out
```

Poolingレイヤ

```python
class Pooling:
    def __init__(self, pool_h, pool_w, stride=2, pad=0):
        self.pool_h = pool_h
        self.pool_w = pool_w
        self.stride = stride
        self.pad = pad

        self.x = None
        self.arg_max = None

    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)
        col = col.reshape(-1, self.pool_h*self.pool_w)

        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)

        self.x = x
        self.arg_max = arg_max

        return out

    def backward(self, dout):
        dout = dout.transpose(0, 2, 3, 1)

        pool_size = self.pool_h * self.pool_w
        dmax = np.zeros((dout.size, pool_size))
        dmax[np.arange(self.arg_max.size), self.arg_max.flatten()] = dout.flatten()
        dmax = dmax.reshape(dout.shape + (pool_size,))

        dcol = dmax.reshape(dmax.shape[0] * dmax.shape[1] * dmax.shape[2], -1)
        dx = col2im(dcol, self.x.shape, self.pool_h, self.pool_w, self.stride, self.pad)

        return dx
```

CNNの実装

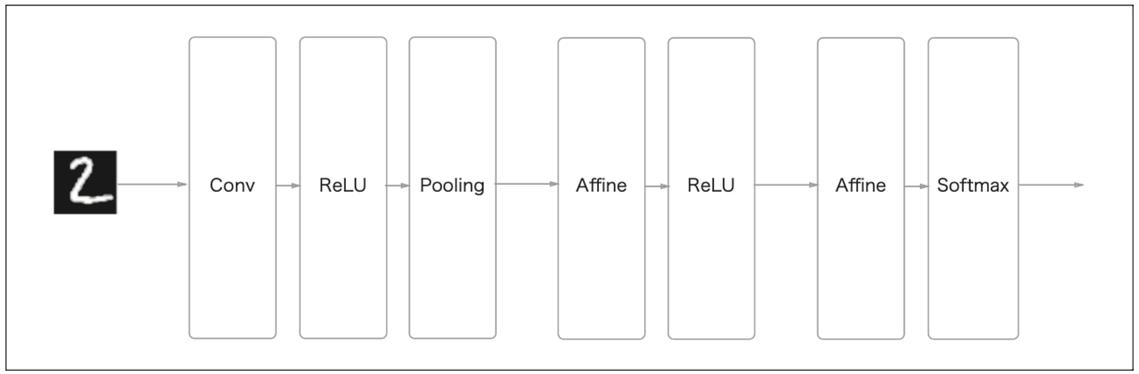


図7-23　単純な CNN のネットワーク構成

初期化

ハイパーパラメータをdictionaryから取り出す

```
class SimpleConvNet:
    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))

        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)


        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()
```

出力サイズを計算

重みパラメータの初期化
(Conv.層と、のこり2つの層
の重みとバイアスを格納

レイヤーを出力

CNNの実装

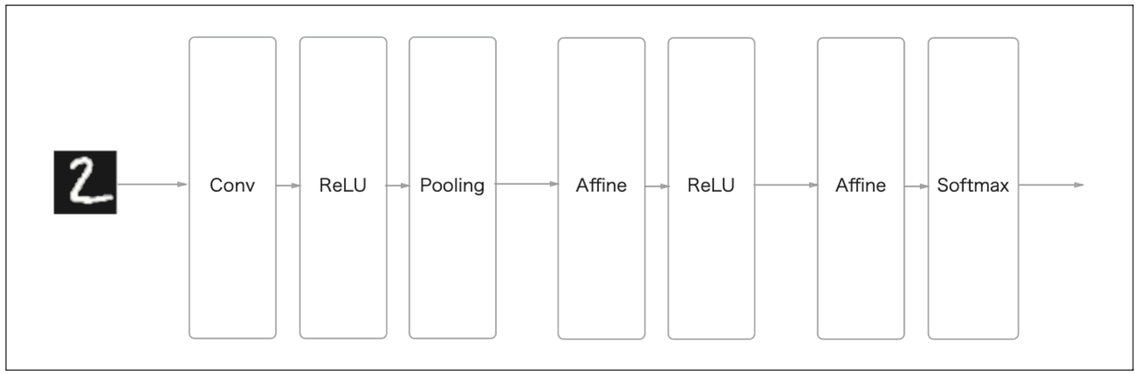
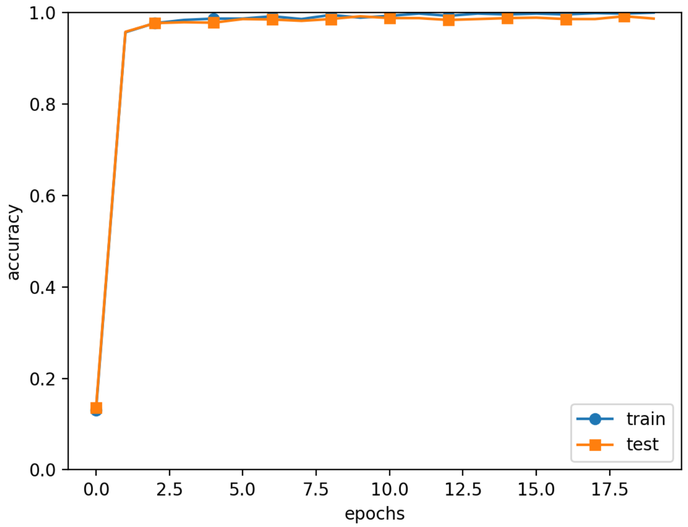


図7-23　単純な CNN のネットワーク構成




```
============== Final Test Accuracy ==============
test acc:0.9894
Saved Network Parameters!
```

x : 入力データ

t : 教師ラベル

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

def loss(self, x, t):
    y = self.predict(x)
    return self.lastLayer.forward(y, t)
```

結果を次の
レイヤーに渡すだけ

```
def gradient(self, x, t):
    # forward
    self.loss(x, t)

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

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

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

    return grads
```

で、誤差伝播法

格納

CNNの可視化

実際どういうものを学習しているのか？

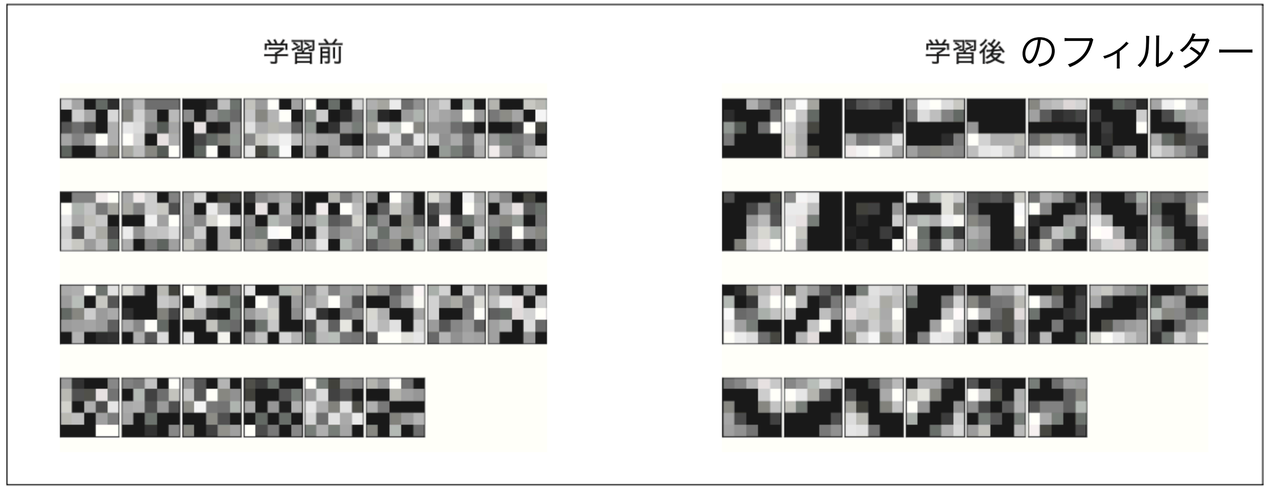
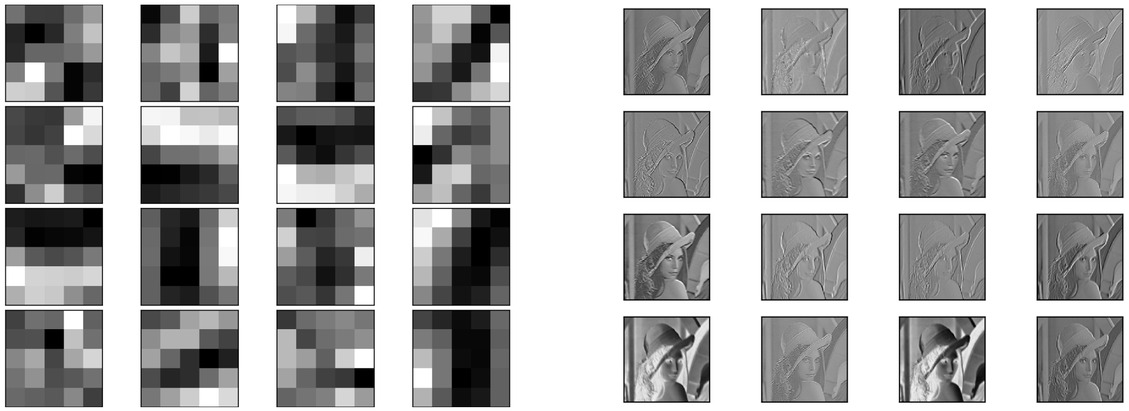


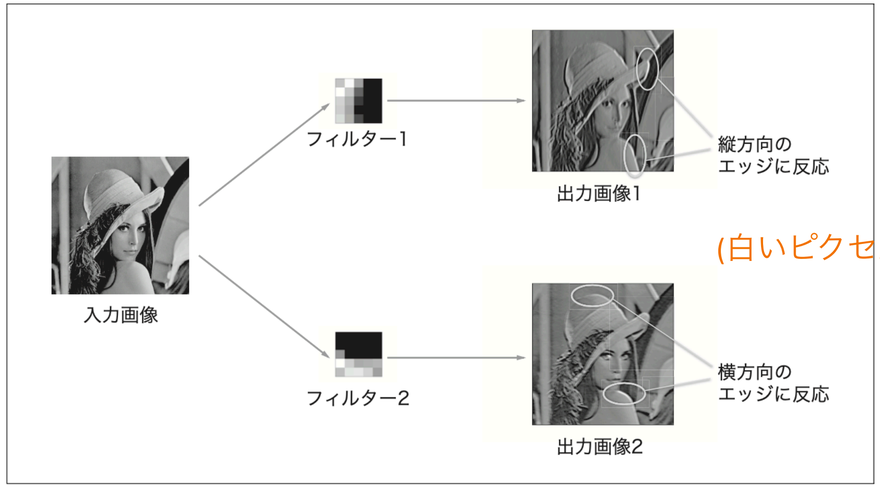
図7-24　学習前と学習後における 1 層目の畳み込み層の重み：重みの要素は実数であるが、画像の表示においては、最も小さな値は黒（0）、最も大きな値は白（255）に正規化して表示する



学習によって規則性のあるフィルターに更新された
それぞれのフィルターが

Edge : 境目

Blob: かたまりのある領域　を見ている



(白いピクセルが出る)

図7-25　横方向のエッジと縦方向のエッジに反応するフィルター：出力画像 1 は縦方向のエッジに白いピクセルが出現。一方、出力画像 2 は横方向のエッジに白いピクセルが多く現れる

CNNの可視化

層を重ねると抽象化されていく

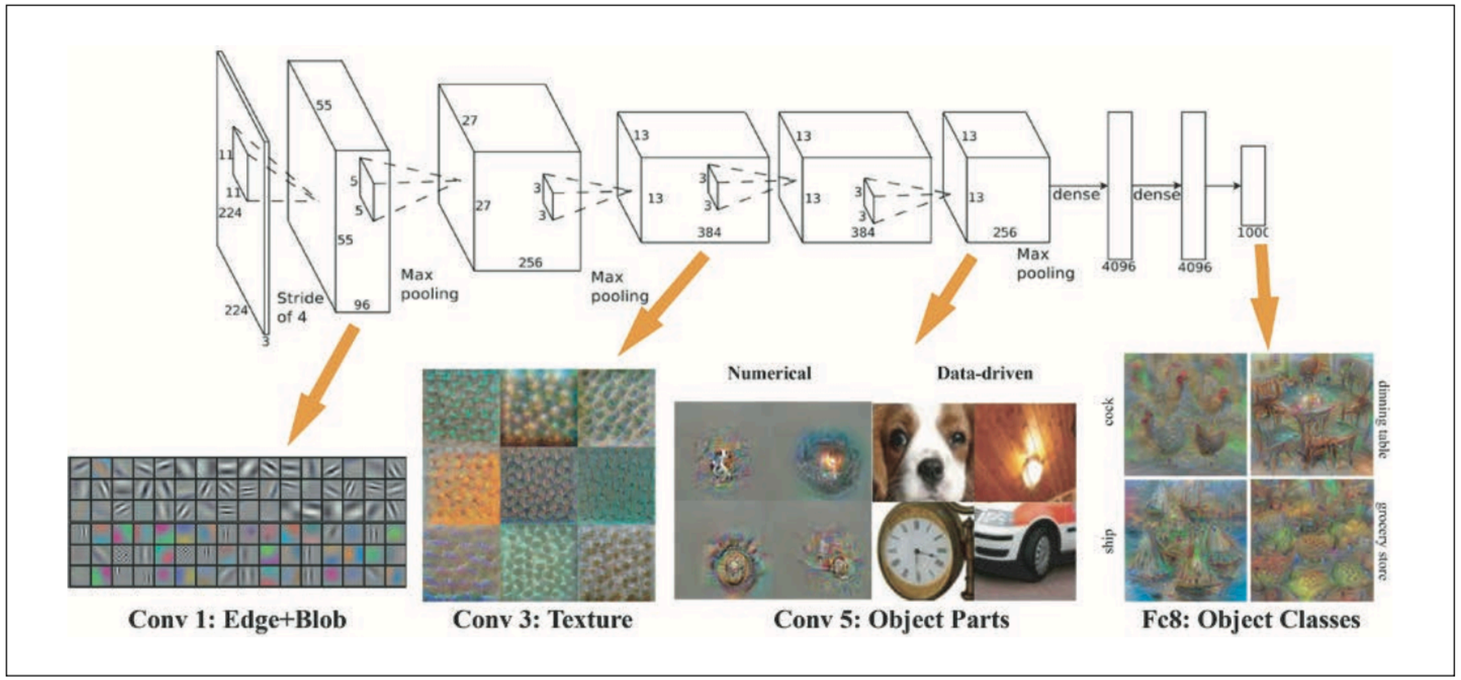


図7-26　CNN の畳み込み層で抽出される情報。1 層目はエッジやブロブ、3 層目はテクスチャ、5 層目は物体のパーツ、そして、最後の全結合層で物体のクラス（犬や車など）にニューロンは反応する（画像は文献 [19] より引用）

代表的なCNN

LeNet （初めてのCNN）

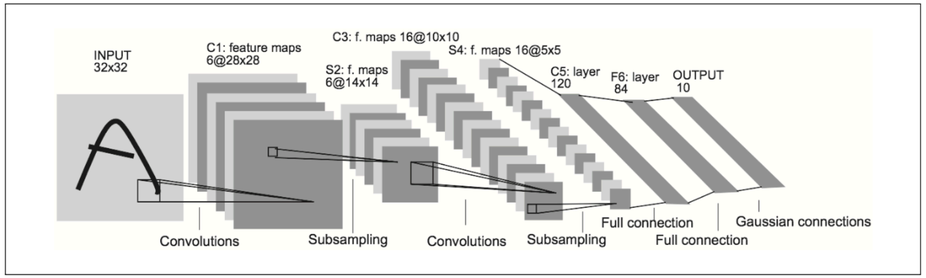


図7-27　LeNet のネットワーク構成（文献 [20] より引用）

AlexNet

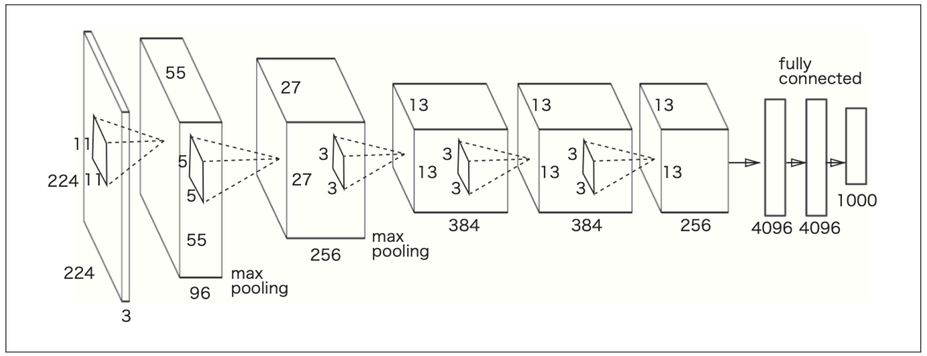


図7-28　AlexNet（文献 [21] を参考に作成）

- 活性化関数に ReLU を用いる
- LRN（Local Response Normalization）という局所的正規化を行う層を用いる
- Dropout（「6.4.3 Dropout」参照）を使用する

そしてGPUが必要に…

backup

CNNの構造

- CNN (convolutional neural network)

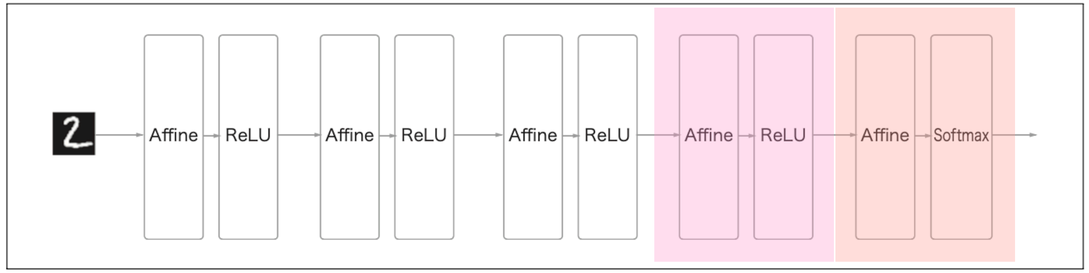


図7-1　全結合層（Affine レイヤ）によるネットワークの例

Affine - ReLU -> Convolution - ReLU - (Pooling)

出力に近い層　最後の出力層

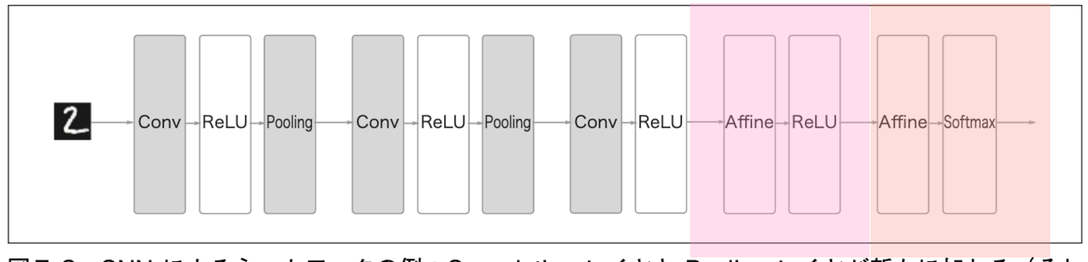


図7-2　CNN によるネットワークの例：Convolution レイヤと Pooling レイヤが新たに加わる（それぞれ背景が灰色の矩形で描画）

- 今までは、隣接する層の全てのニューロン間で結合があった（**全結合**）
  全部Affineレイヤで結合
- ConvolutionレイヤとPoolingレイヤを追加

全結合の問題点

- データの形状が無視されてしまう。
  (Affineレイヤでは、画像のような3次元データ (縦、横、チャンネル)を1列に並べて入力)

- Convolutionレイヤでは、形状を維持して入力する。

  特徴マップ (feature map) : Convolutionレイヤの入出力データ

畳み込み演算

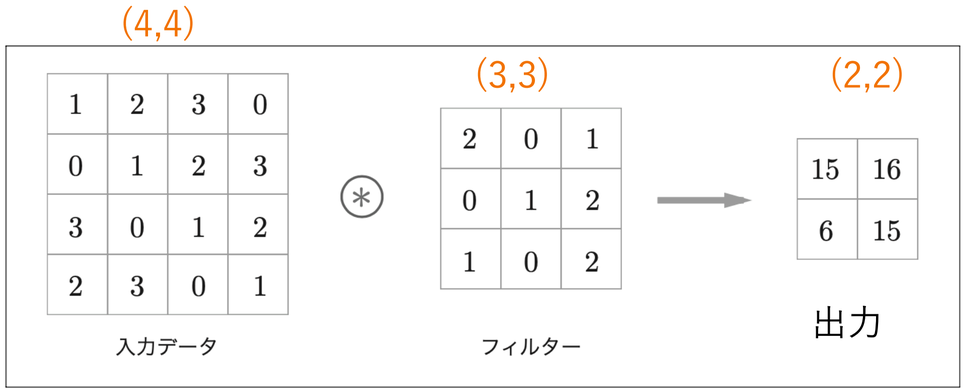


図7-3　畳み込み演算の例：畳み込み演算を「⊛」記号で表記

- フィルター(3,3)をずらしながら積和演算をして、出力の対応する場所に格納する。

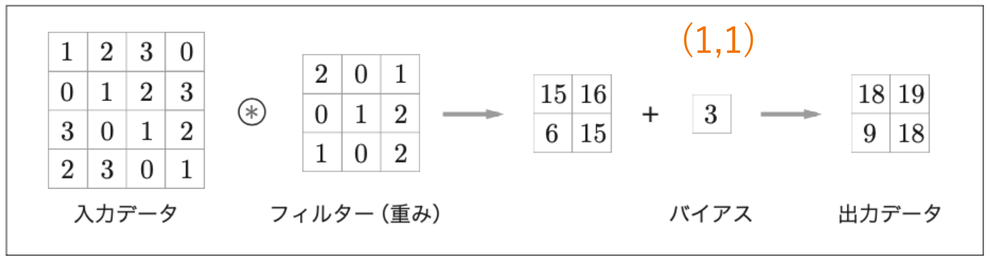


図7-5　畳み込み演算のバイアス：フィルターの適用後の要素に固定の値（バイアス）を加算する

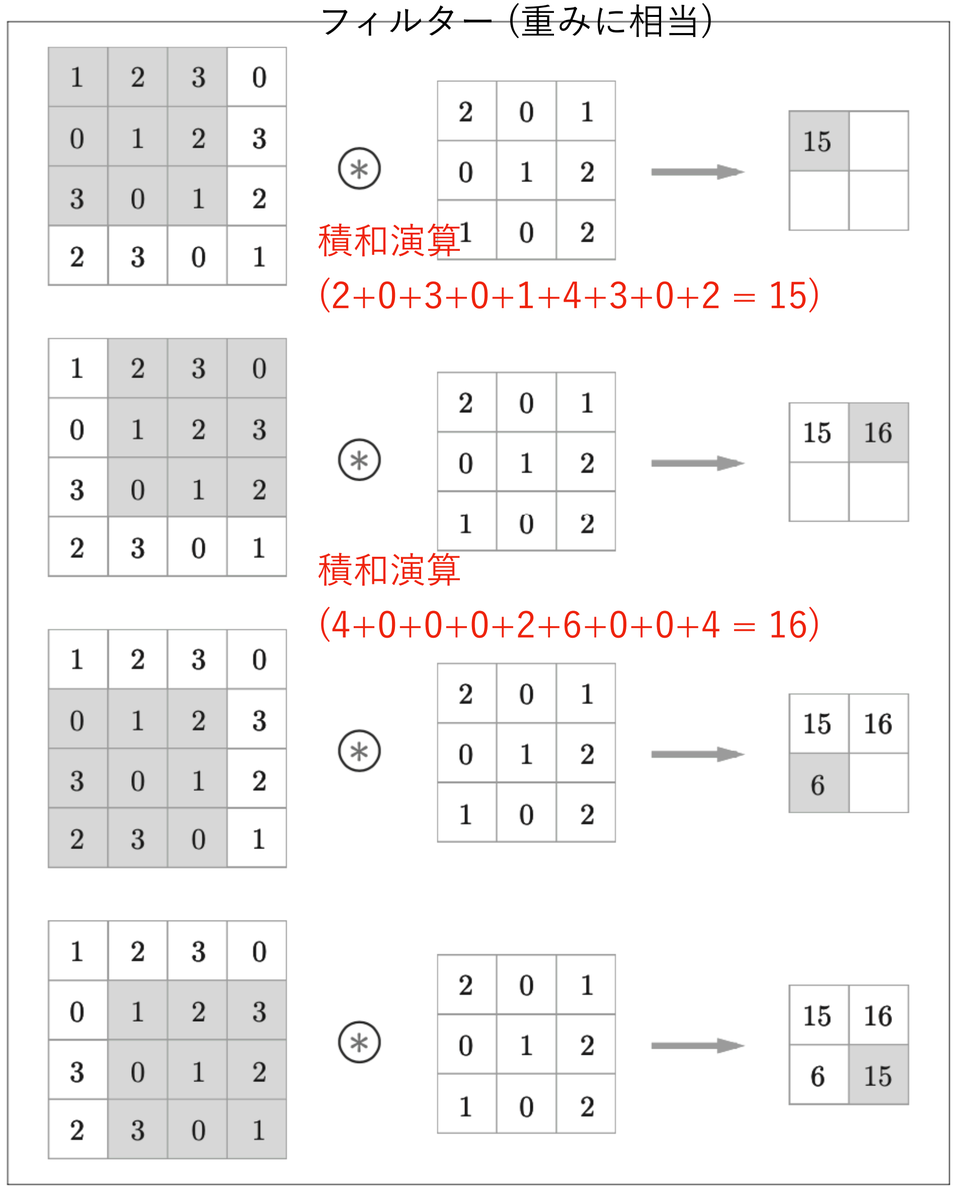


図7-4　畳み込み演算の計算手順

Padding

幅1のpaddingで入力サイズは(6,6)になる

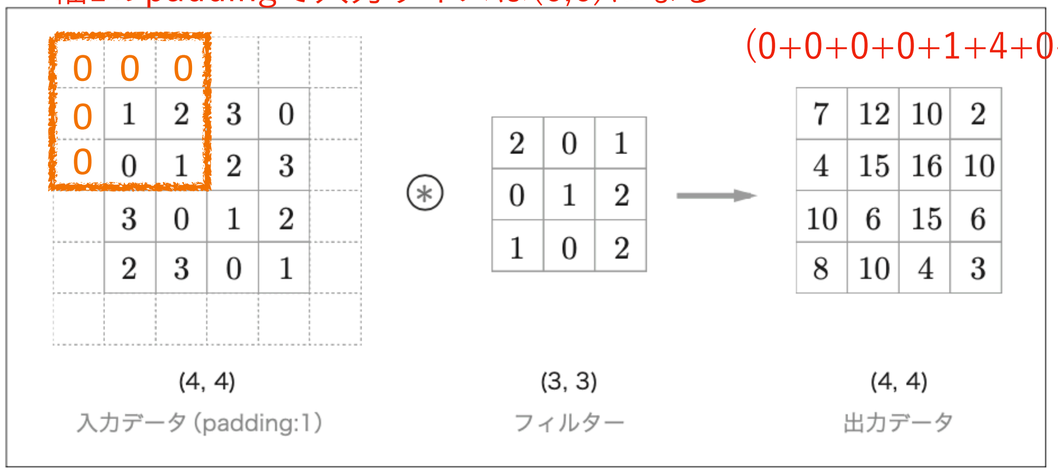


図7-6　畳み込み演算のパディング処理：入力データの周囲に 0 を埋める（図ではパディングを破線で表し、中身の「0」の記載は省略する）

- 出力サイズを調整するために、固定の値 (0とか)で周囲を埋める

- Paddingしなかったら出力が (2,2) になって、
  入力サイズから縮小されてしまう
  -> 空間的なサイズを一定にしたまま次の層にデータを渡す。
  　　繰り返すとそのうち(1,1)になるのを防ぐ

Stride

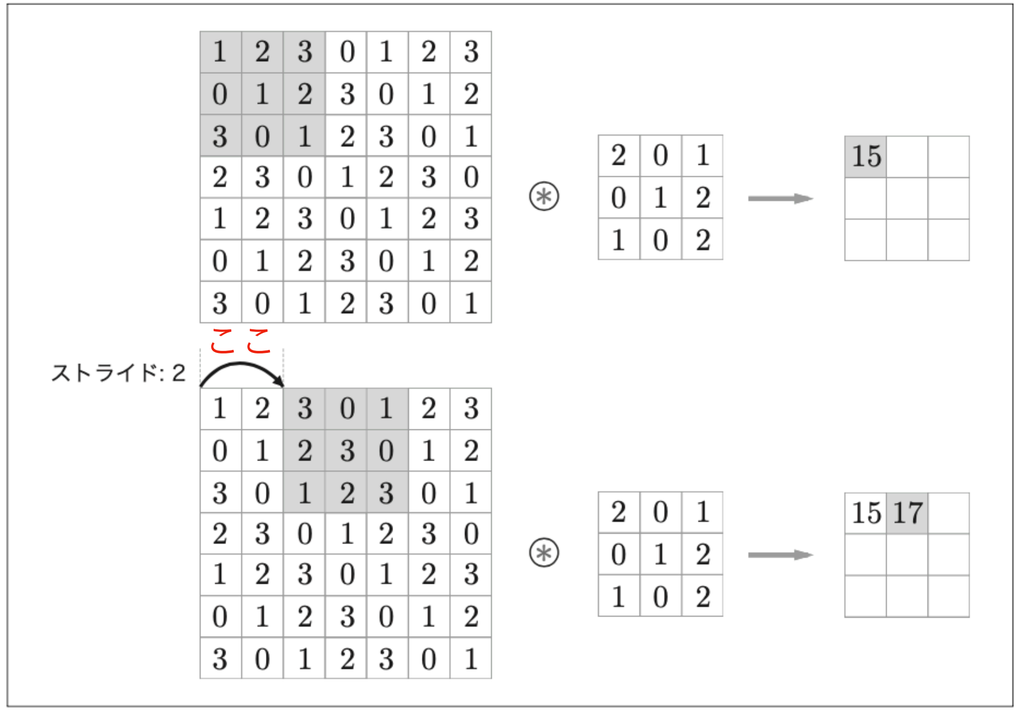


図7-7　ストライドが２の畳み込み演算の例

- フィルターをどれだけずらすか
- Strideを大きくすると、出力サイズは小さくなる。

Padding & Stride

入力 Height　Filter Height
Padding
出力 Height

$$OH = \frac{H + 2P - FH}{S} + 1$$

Stride
入力 Width　Filter Width
出力 Weight

$$OW = \frac{W + 2P - FW}{S} + 1$$

- 注意：割り切れるように値を設定する。
  割り切れない場合はエラーを出力するようにする。

3次元の場合

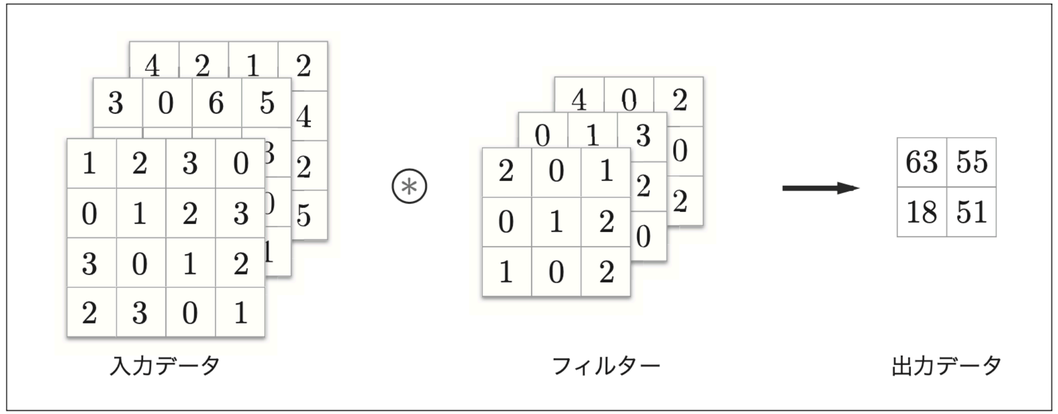


図7-8　3 次元データに対する畳み込み演算の例

- チャンネル方向も加えて、3次元
- チャンネル方向に複数の特徴マップがある場合、
  チャンネルごとに畳み込み演算をして、それらの結果を加算する。
- 入力データのチャンネル数とフィルターのチャンネル数は同じ。

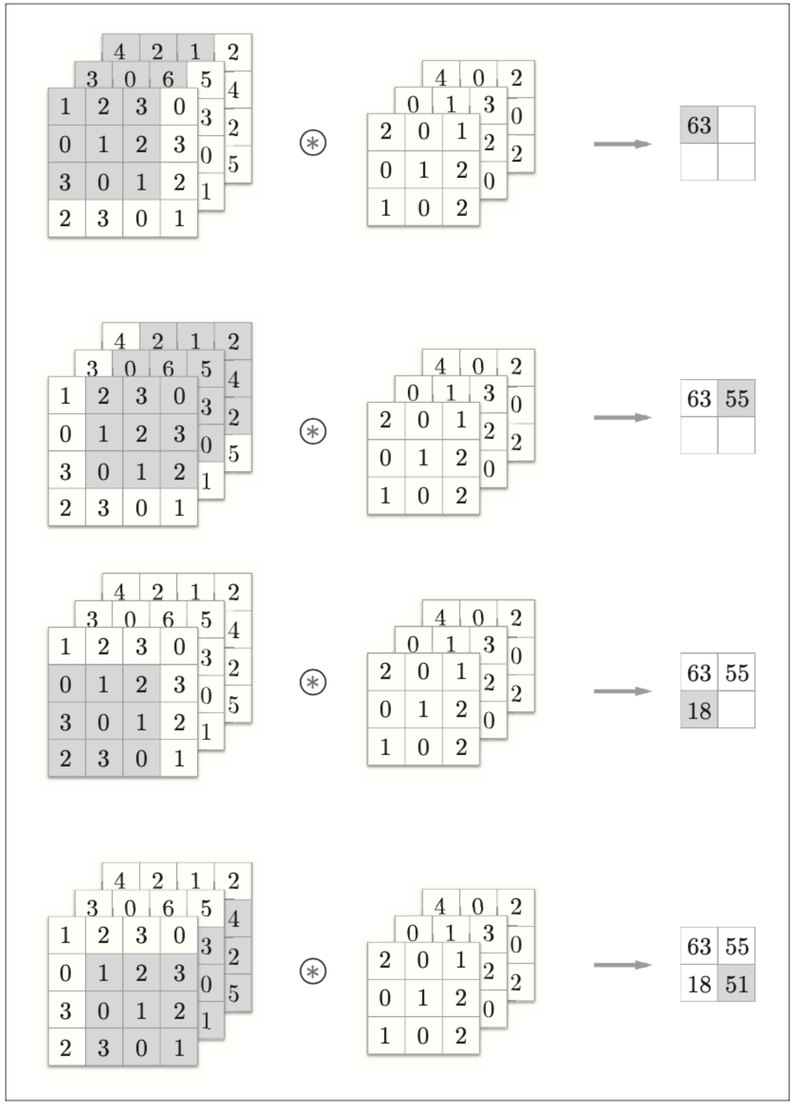


図7-9　3 次元データに対する畳み込み演算の計算手順

3次元の場合：ブロックで考える

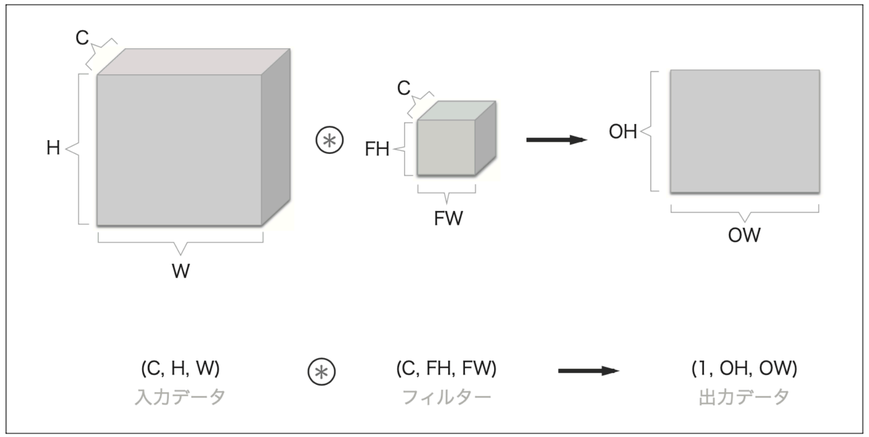


図7-10　畳み込み演算をブロックで考える。ブロックの形状に注意

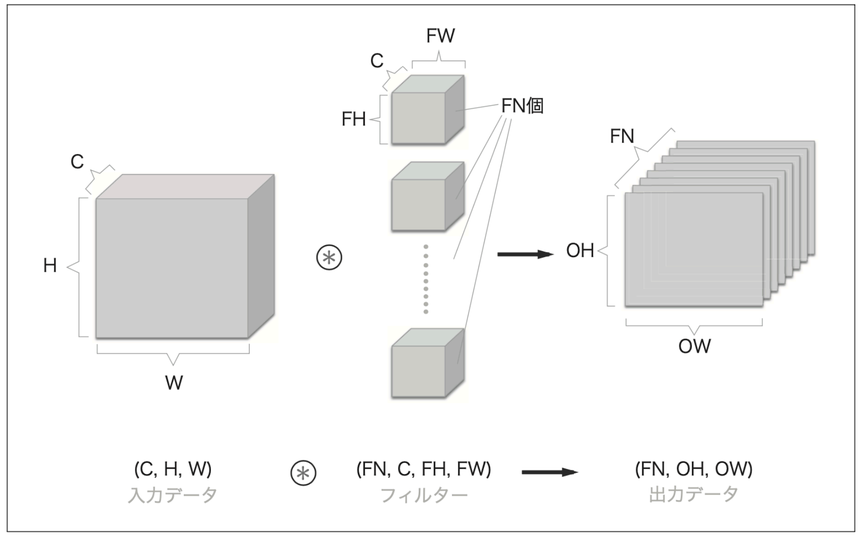


図7-11　複数のフィルターによる畳み込み演算の例

- 出力は channel方向1の特徴マップ
- 出力を channel方向も複数にしたかったら、
  フィルター(重み)を複数にする
- あとバイアスも

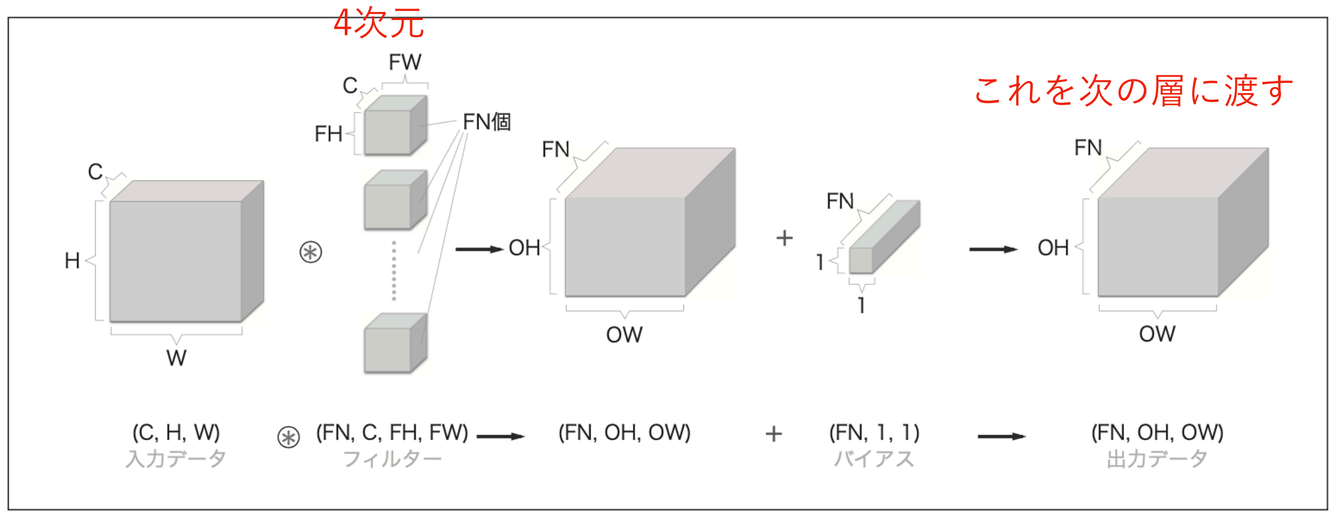


図7-12　畳み込み演算の処理フロー（バイアス項も追加）

バッチ処理

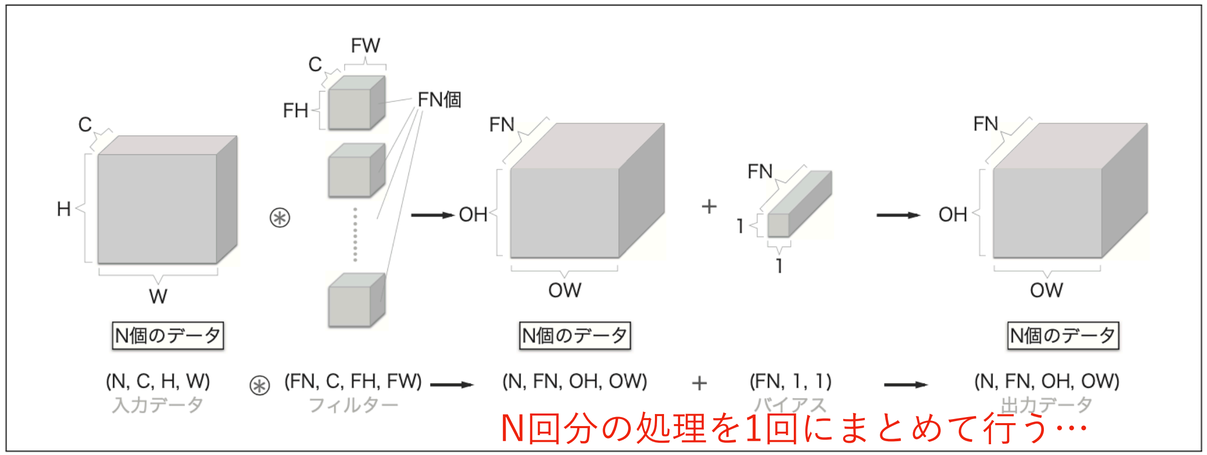


図7-13　畳み込み演算の処理フロー（バッチ処理）

- バッチ処理もやるよ
- (batch_num , channel, height, width)の4次元でデータを流す

Pooling 層

2 ✖ 2の Max pooling (stride 2 の場合)

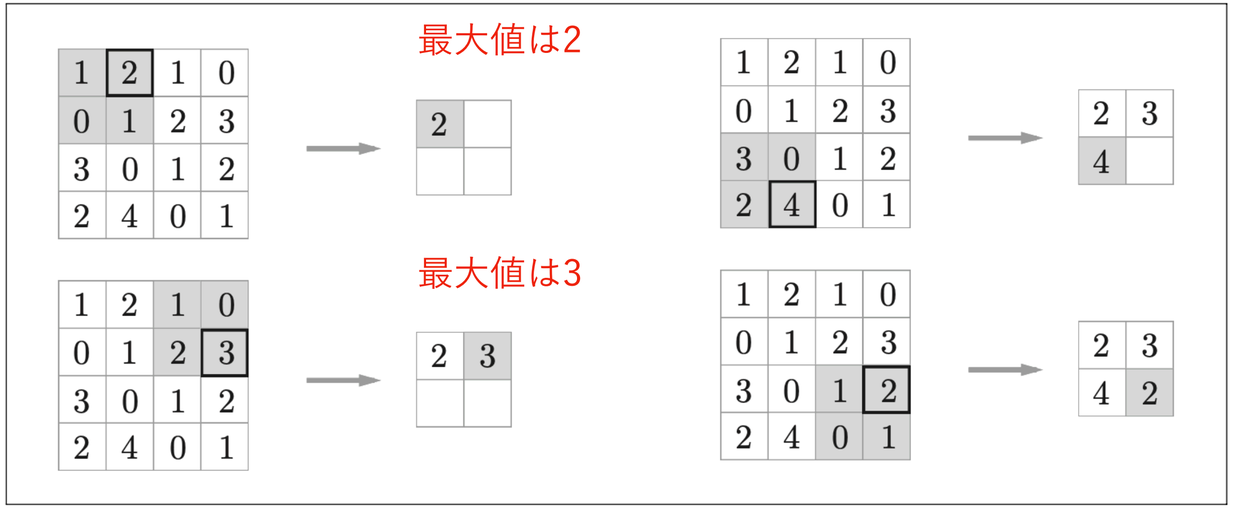


図7-14　Max プーリングの処理手順

- 縦横方向の空間を小さくする演算
  一般的にpoolingのwindowサイズとstrideは同じ値に

Maxプーリング　：最大値をとる

Pooling 層：特徴

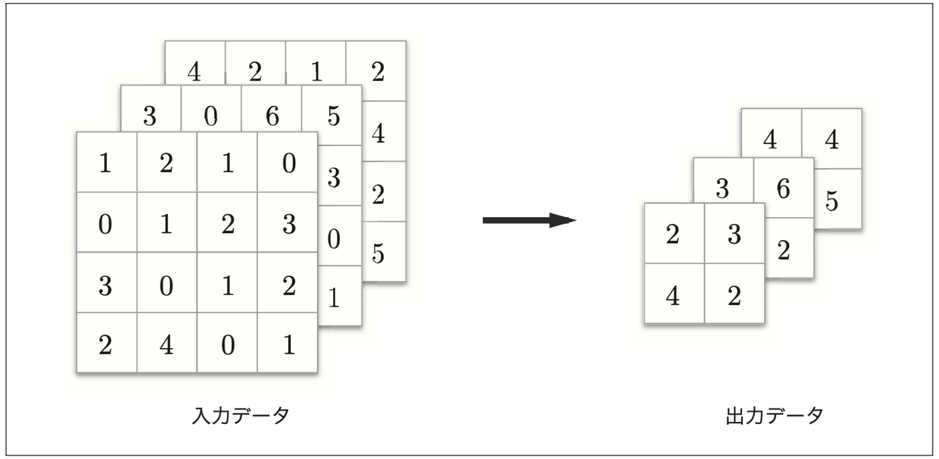


図7-15　プーリングではチャンネル数は変わらない

- 学習するパラメータはない
- チャンネル数は変化しない
- 微小な位置変化に対してロバスト
  （入力データの位置が少しずれても、影響があまりなくなる）

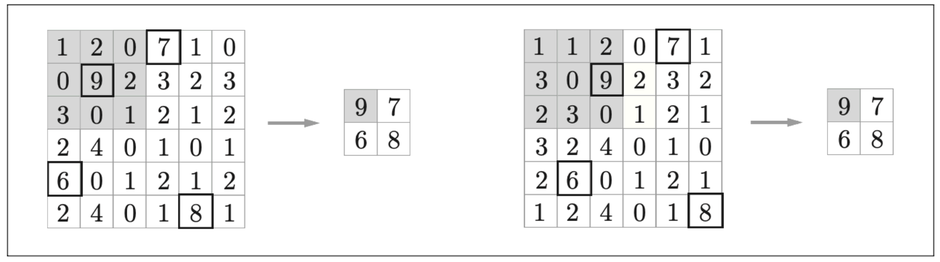


図7-16　入力データが横方向に 1 要素分だけずれた場合でも、出力は同じような結果になる（データによっては同じにならない場合もある）

実装

- 実装部分はまた次回…