

# 7章：畳み込みニューラル ネットワーク（後半）

石田研 M1 宮尾克久

# 前回の補足

# プーリングの種類と使い分け

- Maxプーリング
- Averageプーリング
- L2プーリング : 領域内の活性化出力の和の平方根を取る

## ➤ 実験評価

Scherer, Dominik, Andreas Müller, and Sven Behnke. "Evaluation of pooling operations in convolutional architectures for object recognition." *Artificial Neural Networks–ICANN 2010*. Springer, Berlin, Heidelberg, 2010. 92–101.

## ➤ 理論的比較

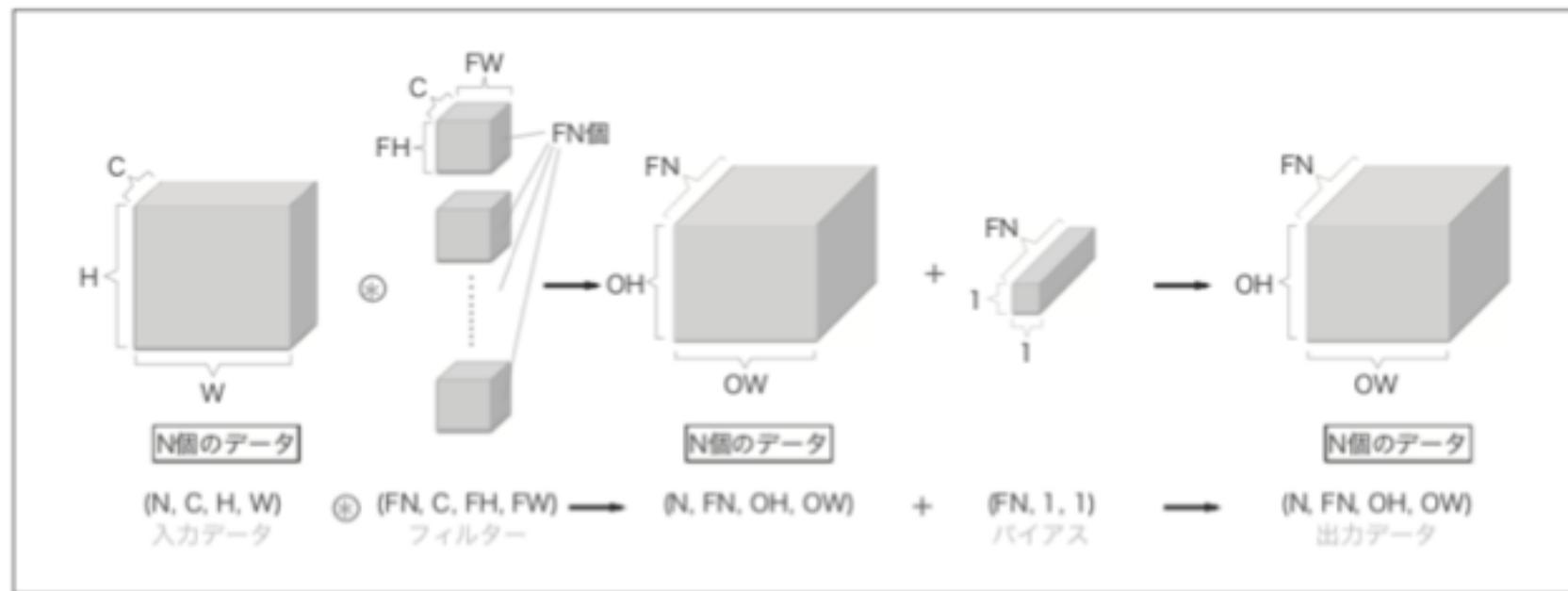
Boureau, Y-Lan, Jean Ponce, and Yann LeCun. "A theoretical analysis of feature pooling in visual recognition." *Proceedings of the 27th international conference on machine learning (ICML–10)*. 2010.

# **畳み込み層・プーリング層の実装**

# 4次元配列

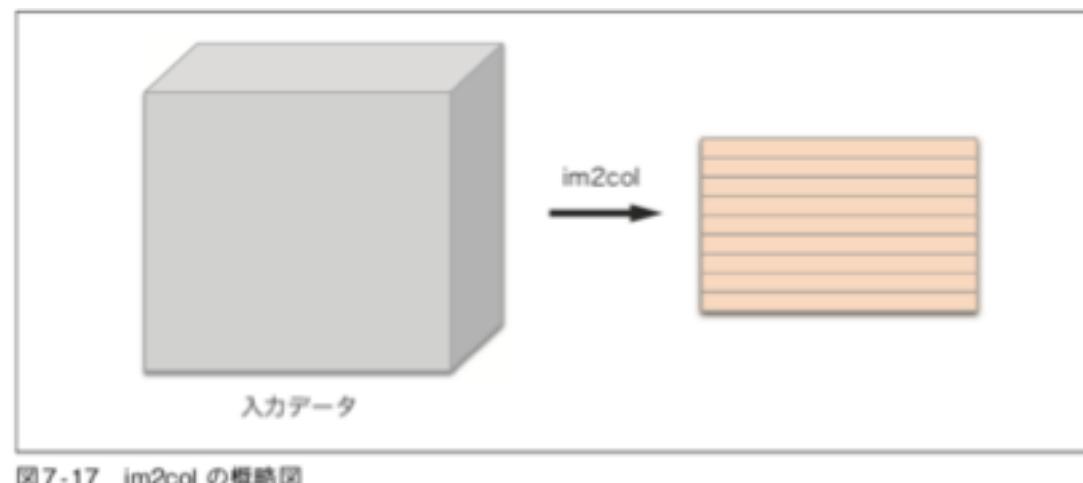
- 各層を流れるデータは4次元で表現される。

ex.  $(10, 1, 28, 28)$  : 高さ 28・横幅 28 で 1 チャンネルのデータが 10 個



# im2colによる展開 (1)

- 置み込み演算の実装は、for文を用いると面倒かつNumPyの処理が遅い  
→ im2colという関数を用いてシンプルに実装
- **im2col** (image to column):  
フィルター(重み)にとって都合の良いように入力データを展開する関数
  - 3次元の入力データに対して im2col を適用すると、2次元の行列に変換



# im2colによる展開 (2)

- 具体的には、入力データに対してフィルターを適用する3次元のブロックを横方向に1列に展開

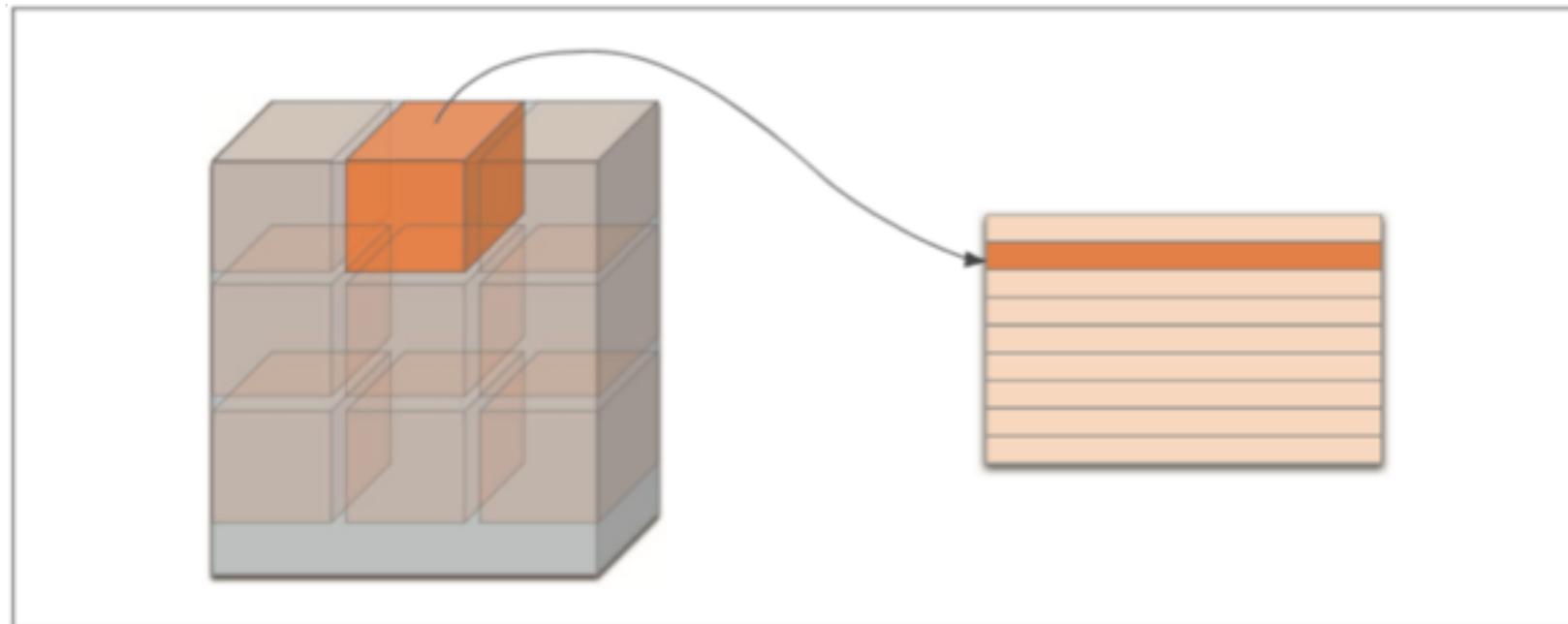


図7-18 フィルターの適用領域を、先頭から順番に 1 列に展開する

# im2colの利点

- 行列計算のライブラリ(線形代数ライブラリ)などは、行列の計算実装が高度に最適化されており、大きな行列の掛け算を高速に行うことが可能。  
→ 行列の計算に帰着させることで、線形代数ライブラリを有効に活用
- 逆に欠点として…  
展開後の要素の数が元のブロックの要素数よりも多くなるため、通常よりも多くのメモリを消費する

# im2colのフィルター処理

- 入力データの展開後、畳み込み層のフィルター(重み)も1列に展開し2つの行列の積を計算する。

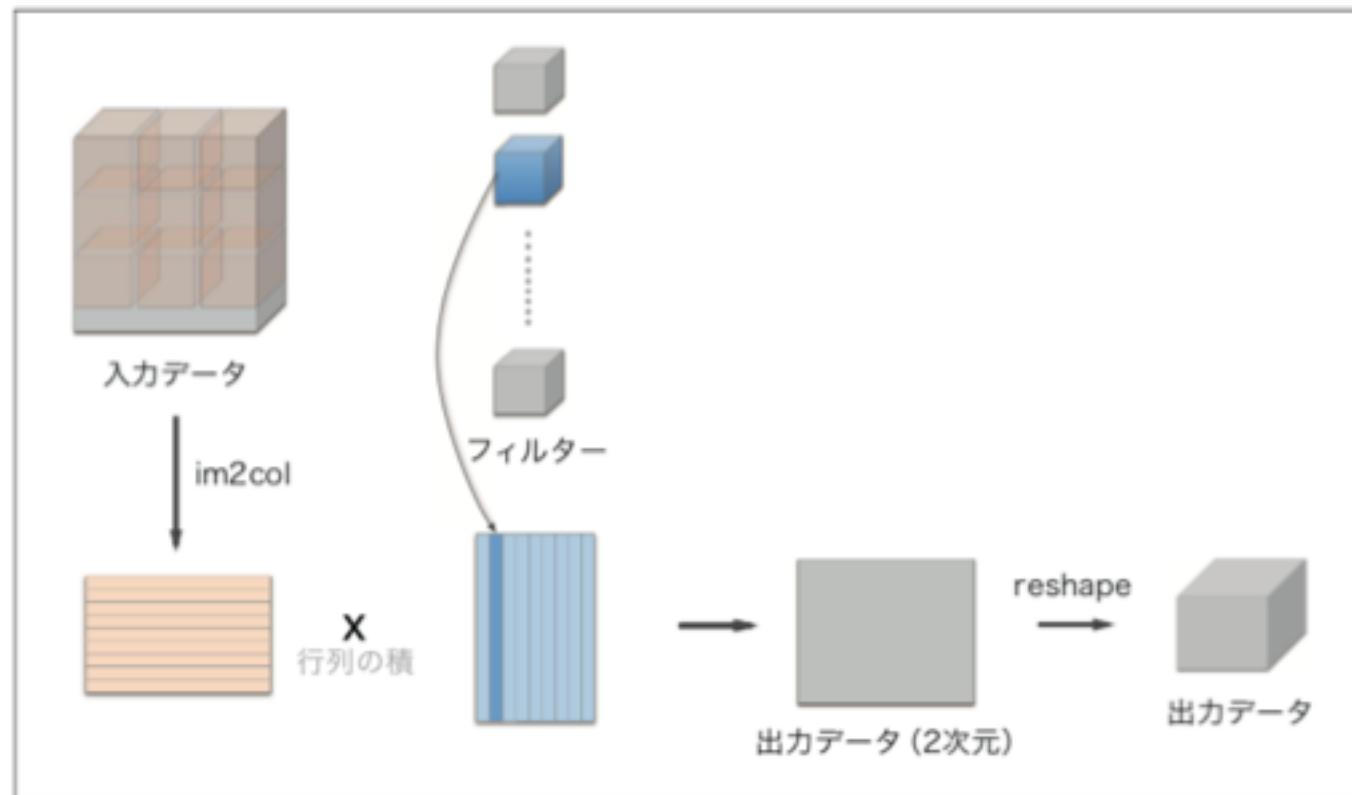


図7-19 畳み込み演算のフィルター処理の詳細：フィルターを縦方向に1列に展開して並べ、`im2col`で展開したデータと行列の積を計算する。最後に、出力データのサイズに整形（`reshape`）する

# im2colのインターフェース

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

- `input_data`: (データ数, チャンネル, 高さ, 横幅) の4次元配列からなる入力データ
- `filter_h`: フィルターの高さ
- `filter_w`: フィルターの横幅
- `stride`: ストライド
- `pad`: パディング

※ 実装の中身は `common/util.py` を参照。

# 実際に使ってみる

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

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)
```

- 両方のケースで2次元目の要素数は75になっているが、これはフィルター(チャンネル3、サイズ $5 \times 5$ )の要素数の総和。

# 畳み込み層の実装

```
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)  
        *1 col_W = self.W.reshape(FN, -1).T # フィルターの展開  
        out = np.dot(col, col_W) + self.b  
  
        *2 out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)  
  
    return out
```

- \* reshapeの際に-1を指定すると、多次元配列の要素数の辻褷が合うように要素数をまとめてくれる。
  - (10, 3, 5, 5)の形状の配列は要素数が全部で750個だが、ここでreshape(10, -1)とすると(10, 75)の形状の配列に整形される。
- \* transpose関数：多次元配列の軸の順番を入れ替える関数

# プーリング層の実装上の特徴

- プーリング層でも im2col を用いるが、プーリングの処理はチャンネル方向に独立なため適用領域はチャンネルごとに独立して展開
- 展開した行列に対して、行ごとに最大値を求め適切な形状に整形するだけ

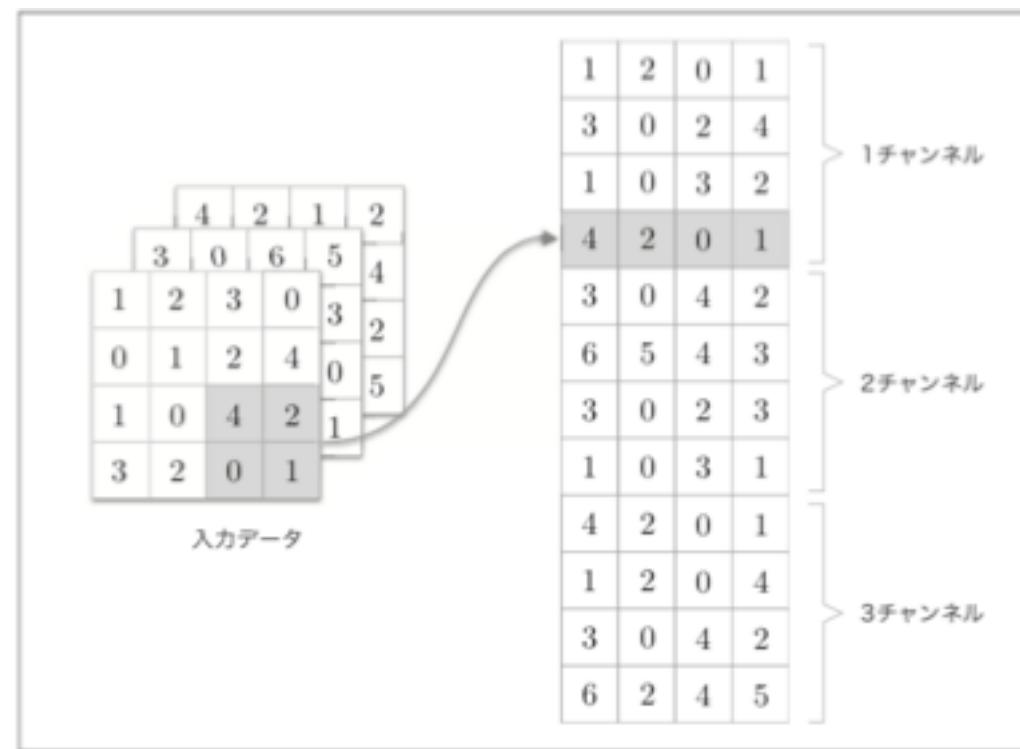


図7-21 入力データに対してプーリング適用領域を展開 (2 × 2 のプーリングの例)

# プーリング層実装の流れ

1. 入力データを展開する
2. 行ごとに最大値を求める
3. 適切な出力サイズに整形する

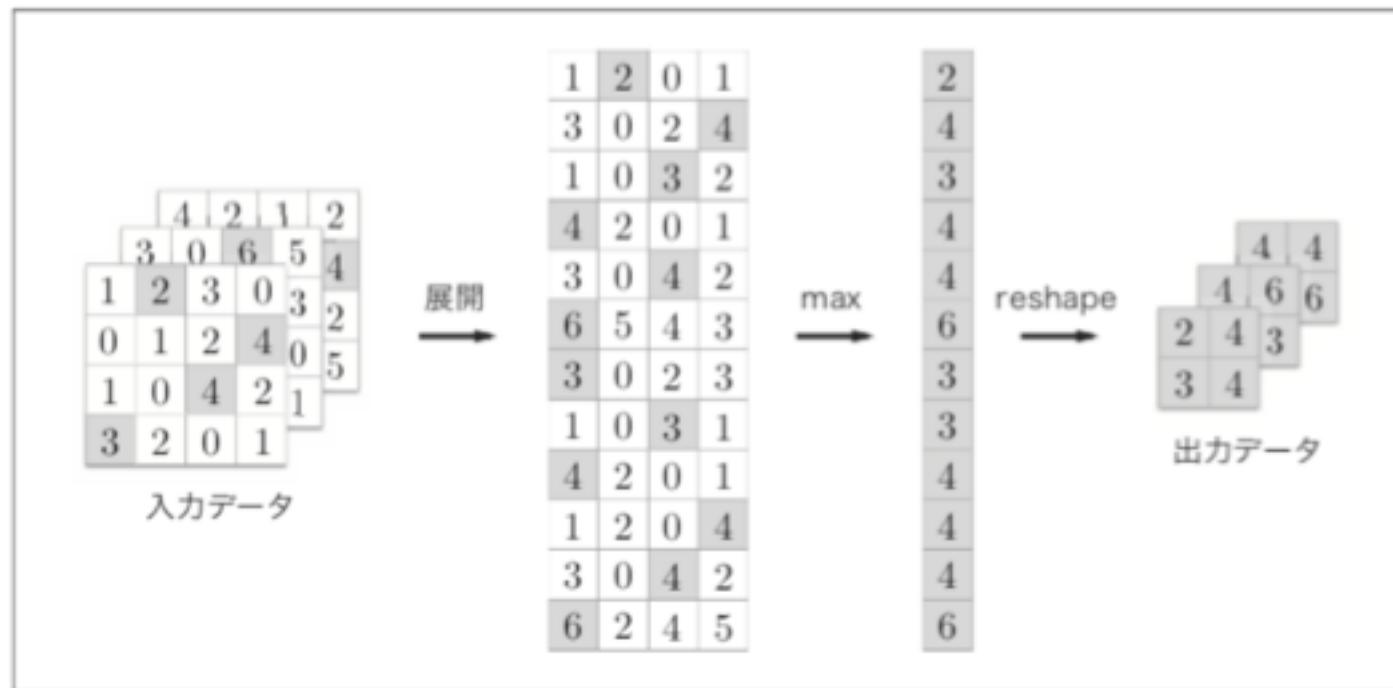


図7-22 Pooling レイヤの実装の流れ：プーリング適用領域内の最大値の要素は背景をグレーで描画

# プーリング層の実装

```
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)  
        *1 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
```

\* np.max は引数に axis を指定することができ、その引数で指定した軸ごとに最大値を求めることができる。

# 畳み込み層・プーリング層の逆伝播

- 逆伝播の実装は、Affineレイヤとほとんど同様  
(im2colの逆を行う処理 col2im を用いる点のみ異なる)。
- 詳細は common/layer.py を参照

# CNNの実装

# 全体の流れ

- 下図のようなネットワークを SimpleConvNet という名前のクラスで実装する

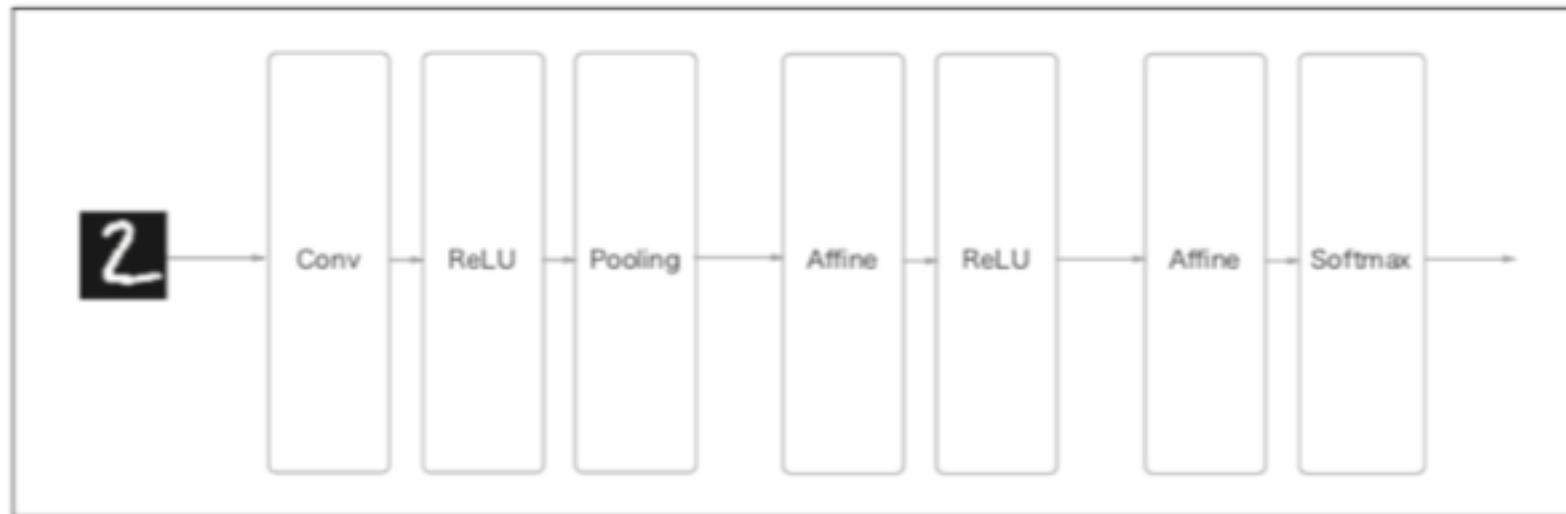


図 7-23 単純な CNN のネットワーク構成

# 初期化(`_init_`)

## 引数

- `input_dim`: 入力データの(チャンネル, 高さ, 幅)の次元
- `conv_param`: 畳み込み層のハイパーパラメータ(ディクショナリ)。ディクショナリのキーは下記のとおり
  - `filter_num`: フィルターの数
  - `filter_size`: フィルターのサイズ
  - `stride`: ストライド
  - `pad`: パディング
- `hidden_size`: 隠れ層(全結合)のニューロンの数
- `output_size`: 出力層(全結合)のニューロンの数
- `weight_init_std`: 初期化の際の重みの標準偏差

# 初期化 – パラメータの取り出し

```
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))
```

- 初期化の引数で与えられた畳み込み層のハイパーパラメータをディクショナリから取り出す
- 畳み込み層の出力サイズを計算

# 初期化 – 重みパラメータの初期化

```
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)
```

- 学習に必要なパラメータは、1層目の畳み込み層と、残り2つの全結合層の重みとバイアス
- これらのパラメータをインスタンス変数の params ディクショナリに格納

# 初期化 - レイヤ生成

```
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()
```

- 先頭から順にレイヤを順序付きディクショナリ(OrderedDict)の layers に追加していく。
- 最後の SoftmaxWithLoss レイヤだけは、lastLayer という別の変数に追加

# predictメソッド / lossメソッド

```
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)
```

x は入力データ、t は教師ラベル

- predictメソッド

追加したレイヤを先頭から順に呼び出しその結果を次のレイヤに渡す

- lossメソッド

predictメソッドで行った forward 処理に加えて、最後の層の SoftmaxWithLoss レイヤまで forward 処理を行うことで損失関数を求める。

# 誤差逆伝播法による勾配の計算

```
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
```

- 順伝播と逆伝播を続けて行う
- それぞれのレイヤで順伝播と逆伝播の機能が正しく実装されているので、ここでは単にそれらを適切な順番で呼ぶだけ
- 最後に grads というディクショナリに各重みパラメータの勾配を格納

# SimpleConvNet でMNISTデータセットを学習してみる

- 訓練データの認識率は 99.82%、テストデータの認識率は 98.96% であり比較的小さなネットワークにしてはとても高い認識率(らしい)
- 次章では、さらに層を重ねてディープにすることで、テストデータの認識率が 99% を超えるネットワークを実現する

# CNNの可視化

# 1層目の重みの可視化(1)

- MNIST データセットに対して行なった学習では、1層目の畳み込み層の重みの形状は  $(30, 1, 5, 5)$  だった。
- フィルターのサイズが  $5 \times 5$  でチャンネル数が 1 ということは、フィルターは 1 チャンネルのグレー画像として可視化できるということを意味する。
- 学習前のフィルターはランダムに初期化されているため、白黒の濃淡には規則性がないが、学習を終えたフィルターは規則性のある画像になっている

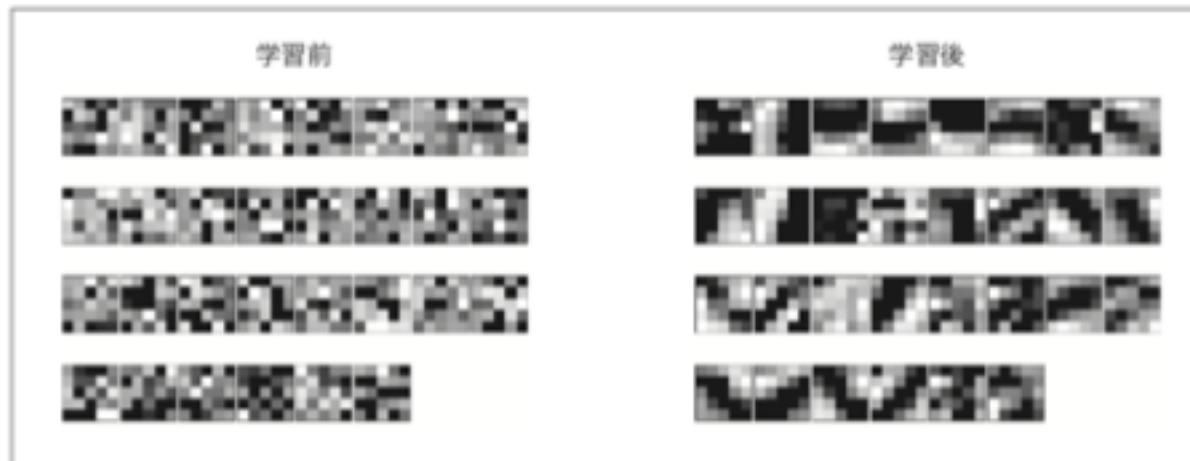


図7-24 学習前と学習後における 1 層目の畳み込み層の重み：重みの要素は実数であるが、画像の表示においては、最も小さな値は黒（0）、最も大きな値は白（255）に正規化して表示する

# 1層目の重みの可視化(2)

- 規則性のあるフィルターは、エッジ(色が変化する境目)やプロブ(局所的に塊のある領域)などを“見ている”
- 学習済みのフィルターを 2 つ選んで、入力画像に畳み込み処理を行うと…

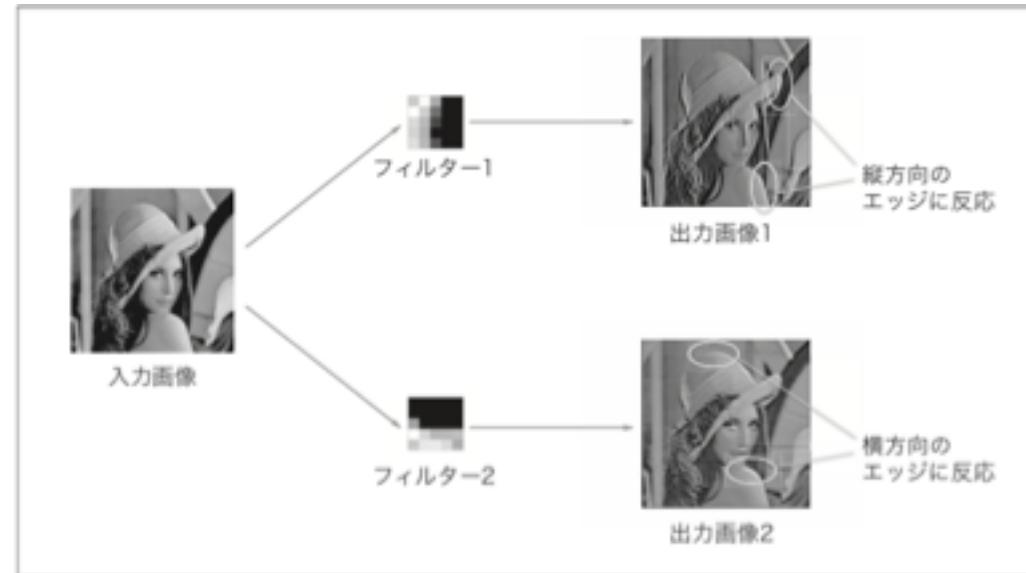


図7-25 横方向のエッジと縦方向のエッジに反応するフィルター：出力画像 1 は縦方向のエッジに白いピクセルが出現。一方、出力画像 2 は横方向のエッジに白いピクセルが多く現れる

- このような低レベルの情報が抽出され後段の層に渡されていく

# 階層構造による情報抽出

- 層が深くなるに従って、ニューロンは単純な形状から“高度”な情報へと変化していく。言い換えれば、モノの「意味」を理解するように反応する対象が変化していく

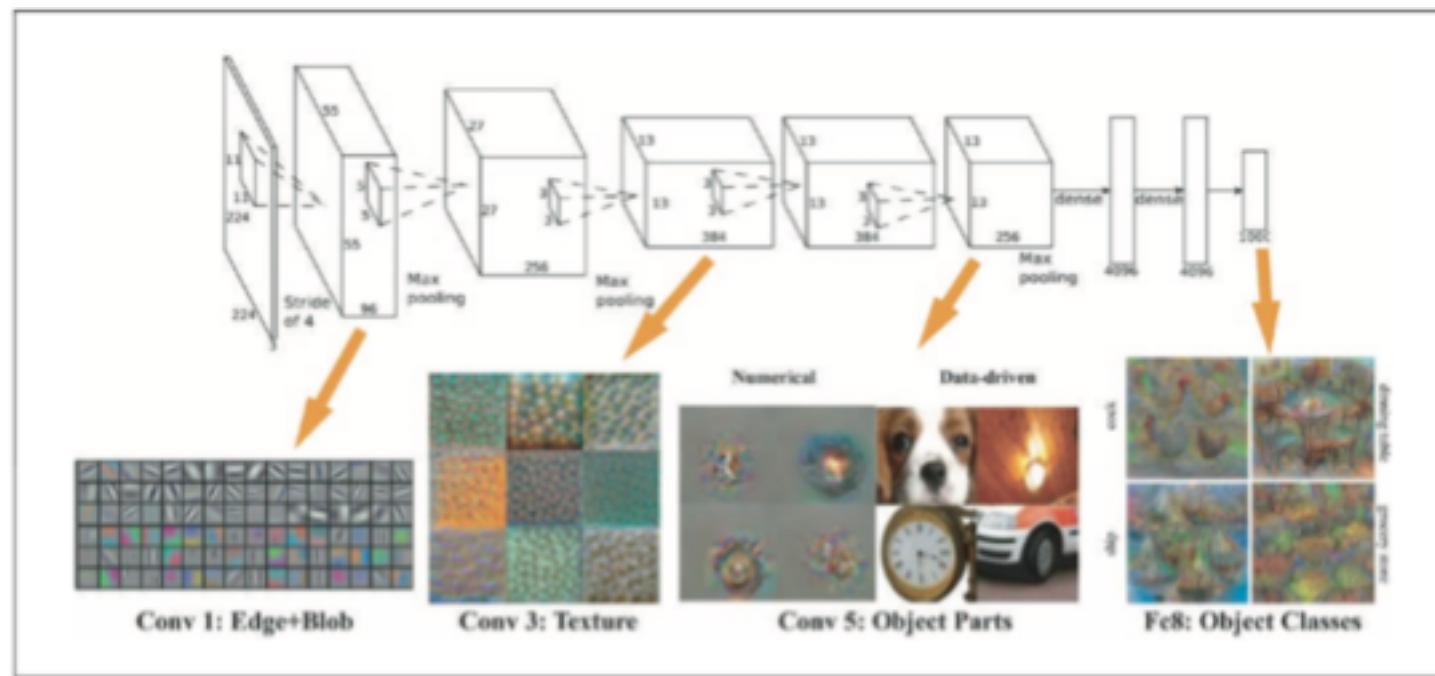


図7-26 CNN の畠み込み層で抽出される情報。1 層目はエッジやプロブ、3 層目はテクスチャ、5 層目は物体のバーツ、そして、最後の全結合層で物体のクラス（犬や車など）にニューロンは反応する（画像は文献 [19] より引用）