7章 畳み込みニューラルネットワーク

NAL研

具志堅凌河, 大城紳之助, 松本一馬

津波大輝, 島袋寛都

7.1 畳み込みニューラルネットワークの全体の構造

CNNの全体の大まかな構造を、今までのニューラルネットワークと比較する。

今までのニューラルネットワークでは、隣接するすべてのニューロン間での結合である、全結合層(affineレイヤ)があった。

CNNではConvolutionレイヤとPoolingレイヤが新たに加わり、ニューラルネットワークを構成する。

今までのaffine-ReLUの流れから、Convolution-ReLU(Pooling)とレイヤの構成に変化している。

出力に近い層ではCNNとニューラルネットワークに変化は見られない。

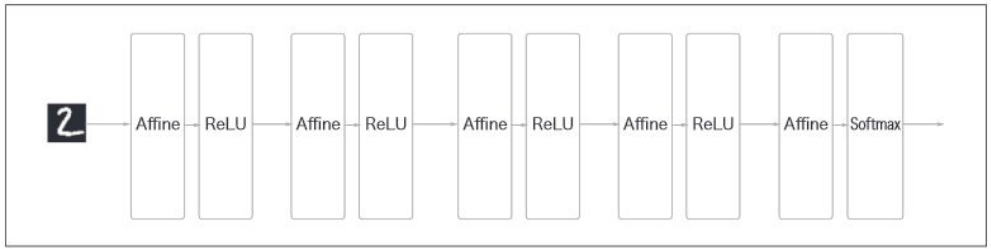


図7-1　全結合層（Affine レイヤ）によるネットワークの例

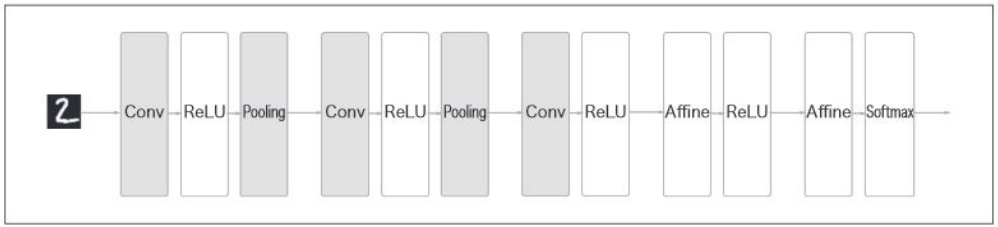


図7-2　CNN によるネットワークの例：Convolution レイヤと Pooling レイヤが新たに加わる（それぞれ背景が灰色の矩形で描画）

7.2 畳み込み層

CNNでは、パディング、ストライドなどの CNN特有の用語がある。
また各層を流れるデータは形状のあるデータになり、これまでの全結合のネットワークとは異なる。

この節ではCNN特有の用語や仕組み、今までのニューラルネットワークとの異なる点などを主に解説していく。

7.2.1 全結合層の問題点

全結合層の問題点

これまでの全結合のニューラルネットワークでは、全結合層 (Affineレイヤ)を用いてきた。

この全結合層では、入力データの構造が無視されてしまうという問題点がある。
(全結合の計算の際、1次元の行列データに直して計算するため)

この全結合層の問題により、入力データが画像の場合では、識別するうえで必要な情報である縦・横・チャンネルといった3次元の情報が1次元にまとめられ、消失してしまう。

よって形状に関する情報を生かすことができない。

7.2.1 全結合層の問題点

畳み込み層では、入力データの形状を維持することができる。

画像の場合、入力データを3次元データとして受け取り、3次元データとして次の層に出力することができる。

これにより、**CNNでは画像などの形状を有したデータを、正しく理解できる可能性がある。**

CNNでは畳み込み層の入出力データを特徴マップ、畳み込み層の入力データを入力特徴マップ
出力データを出力特徴マップと呼ぶ。

7.2.2 畳み込み演算

畳み込み層(Convolutionレイヤ)では、
畳み込み演算を行なっている。

具体的な処理は右の図7-4のように行う。

畳み込み演算は入力データに対し、フィルターのウィンドウ(3*3の灰色の部分)を一定の間隔でスライドさせながら適応さている。

ここでは4*4の入力データに対し、3*3のフィルターを掛けることで、情報を2*2で表現することができた。

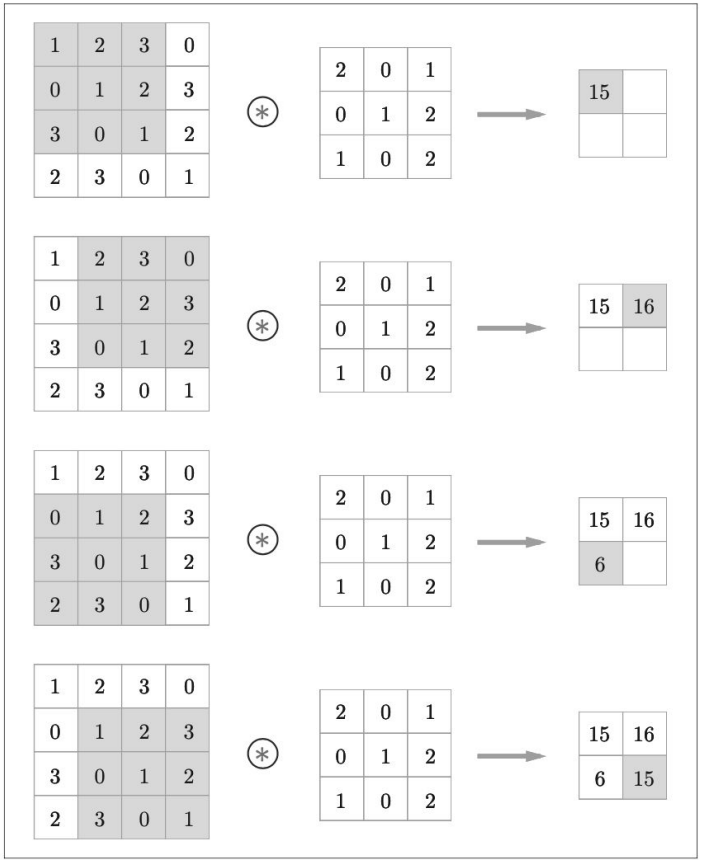


図7-4　畳み込み演算の計算手順

7.2.2 畳み込み演算

・全結合のニューラルネットワークでは、重みパラメータ、バイアスが存在した。

・CNNでは、フィルターがこれまでの重みに対応し、バイアスも存在する。

これらを踏まえ、より詳しく畳み込み演算の流れを表したものが右の図7-5のようになる。

バイアス項の加算は、フィルターの適応後に行われ、またバイアスは一つ(1*1)だけ存在する。

このバイアスはフィルター適用後の全ての要素に加算される。

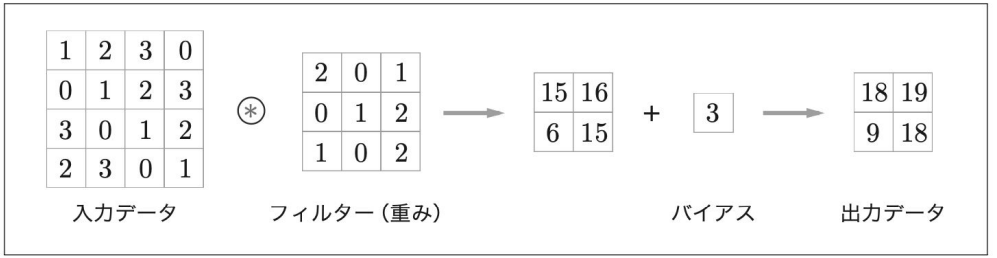


図7-5　畳み込み演算のバイアス：フィルターの適用後の要素に固定の値（バイアス）を加算する

7.2.3 パディング

畳み込み層での処理の前に、入力データの
周囲を固定のデータで埋める処理の事を、
パディングと呼ぶ。

右の図7-6では、4*4の入力に幅1のパディング
を適用している状態である。
今回はゼロパディングと呼ばれ、入力データ
の周囲を0で埋める処理が行われている。

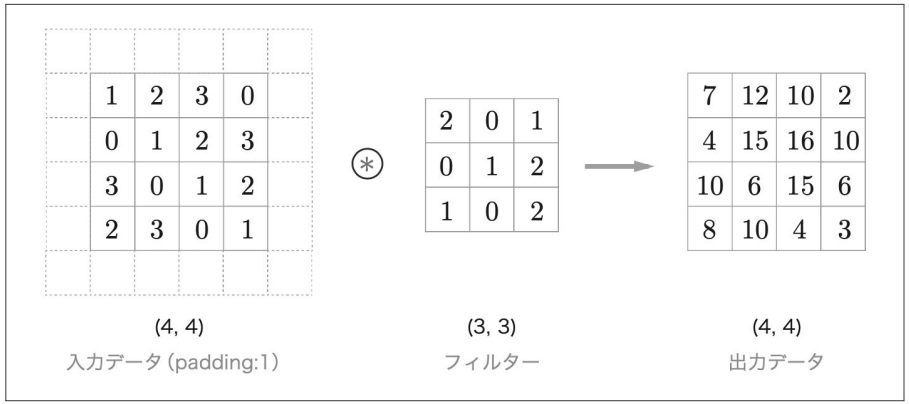


図7-6　畳み込み演算のパディング処理：入力データの周囲に 0 を埋める（図ではパディングを破線で表し、中身の「0」の記載は省略する）

7.2.3 パディング

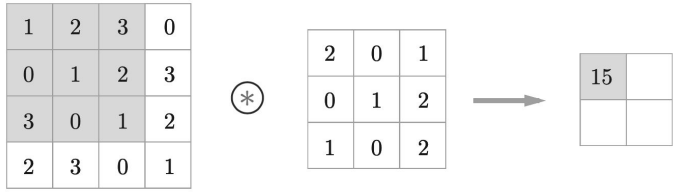
パディングを用いる主な理由は出力データのサイズ調整のためである。

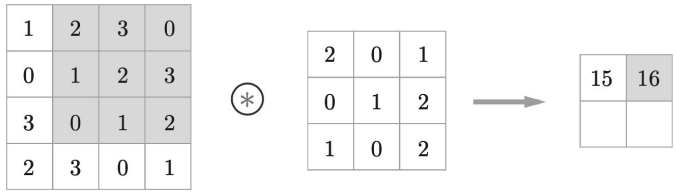
4*4のデータを3*3のフィルターを適応させると、2*2の出力となる事は図7-5で示した。

元のデータに対し、出力が2要素分小さくなる場合、何度も畳み込みを行うと、いずれ出力サイズが 1になってしまい、それ以上畳み込みが適用できなくなる。

先に示した例のように、パディングの幅を 1にする事で、入力と出力のサイズを同じにする事で、
空間的なサイズを一定にしたまま、次の層へとデータを渡すことができる。

7.2.4 ストライド





・フィルターの適応する位置の間隔の事をストライドと言う。

ストライドが1であると、右の上図ように計算を行う。

またストライドを2にした場合の例を右の下図 7-7に示す。

・ストライドの主な目的はパディングとは逆に、出力サイズを小さくすることにある。

図7-7の例では7*7の入力に対し、フィルターが 3*3でストライドが2であるため、得られる出力が 3*3となっている。

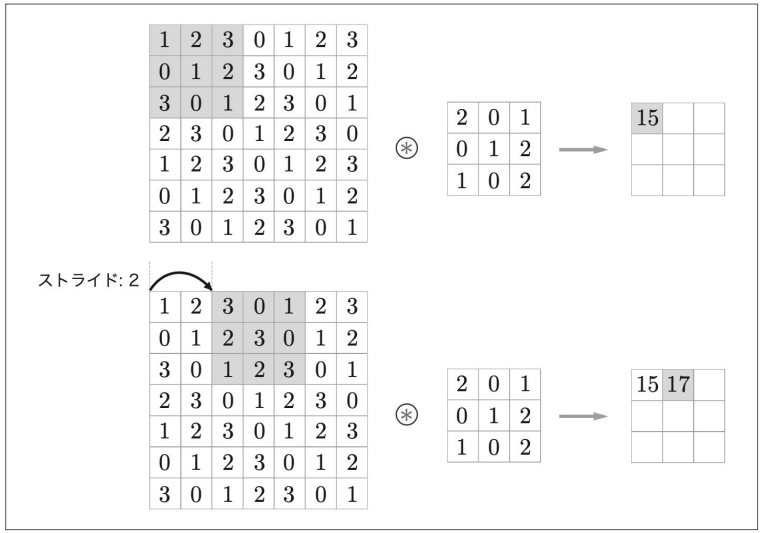


図7-7　ストライドが 2 の畳み込み演算の例

7.2.4 ストライド

入力サイズを(H, W)、フィルターサイズを(FH, FW)、出力サイズを(OH, OW)、パディングをP、
ストライドをSとする。

パディングとストライドに対し、出力サイズがどう計算されるのかを式で表すと

$$OH = \frac{H + 2P - FH}{S} + 1$$
$$OW = \frac{W + 2P - FW}{S} + 1 \qquad (7.1)$$

式(7.1)のように表せられた。上記の式は割り切れるように値を決める必要があるため、ストライドやパディング
の設定の際の指標になる。

7.2.5 3次元データの畳み込み演算

これまでの畳み込み演算の例では、縦・横方向の 2次元を対象としていた。

画像の場合、縦・横・チャンネル方向 (RGBなど)の3次元配列(テンソル)を扱う必要がある。

その場合の畳み込み演算の例を右の図 7-9に示す。

チャンネル方向(奥行き)に複数の特徴マップ(畳み込み層の入出力データ)がある場合、チャンネルごとに畳み込み演算を行い、それらの結果を加算し、出力を得る。

注意点として、入力データとフィルターのチャンネル数は同じ値にしなければならない。

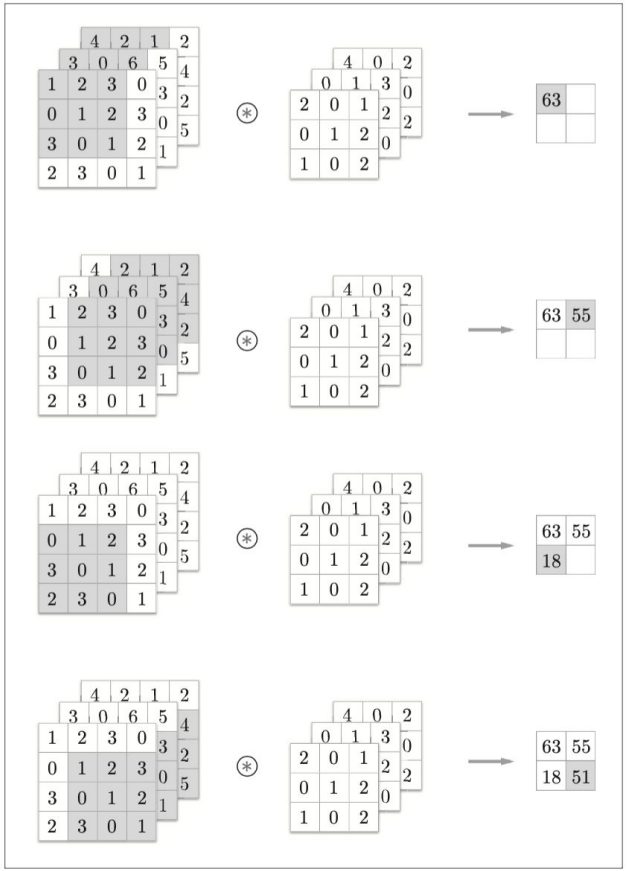


図7-9　3次元データに対する畳み込み演算の計算手順

7.2.6 ブロックで考える

3次元の畳み込み演算のデータやフィルターを
ブロックで考えてみる。

入力データはチャンネル数 C,高さH,横幅W
フィルターはチャンネル数 C,高さFH,横幅FW
とした例を右の図7-10に示す。

畳み込み演算の結果得られる出力は 1つの特徴
マップが得られる。

すなわちフィルター(重み)1つによって得られる出
力がチャンネル方向が 1の特徴マップである事を示
している。

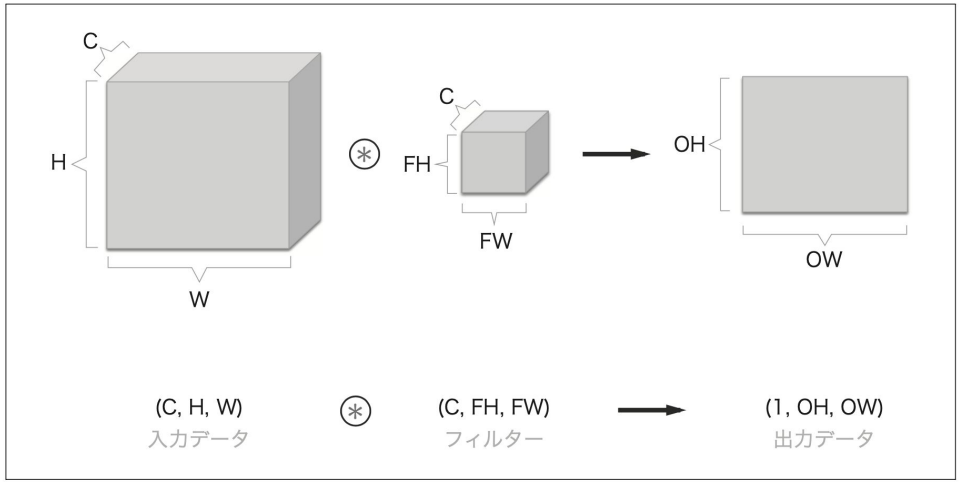


図7-10　畳み込み演算をブロックで考える。ブロックの形状に注意

7.2.6 ブロックで考える

畳み込み演算の出力をチャンネル方向にも複数もたせたのが右上の図 7-11である。

フィルターをFN個用いる事で、出力もFN個出力され、形状が(FN,OH,OW)のブロックができる。このブロックを次の層に渡すのが CNNでの処理フローである。

またバイアスの処理を追加したものが右下の図 7-12である。

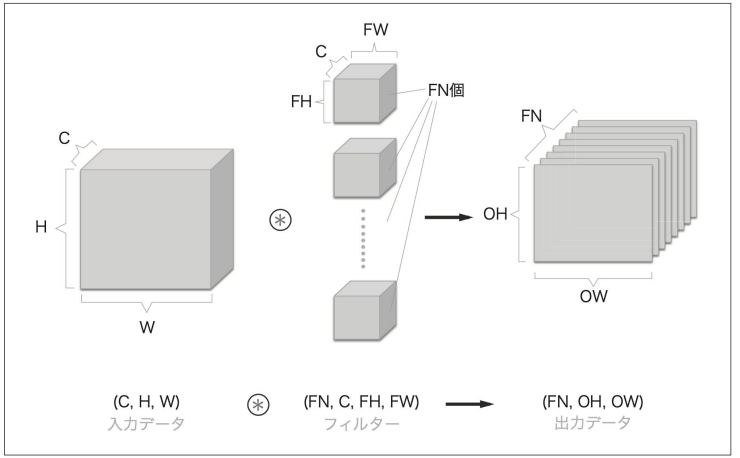


図 7-11　複数のフィルターによる畳み込み演算の例

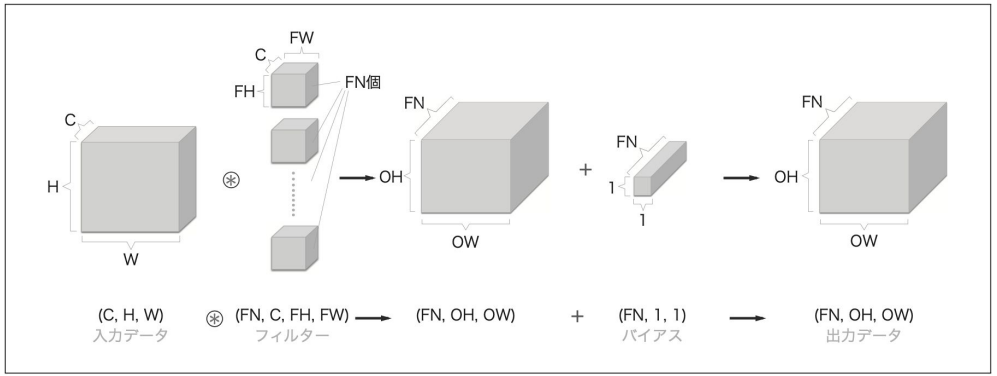


図 7-12　畳み込み演算の処理フロー（バイアス項も追加）

7.2.7 バッチ処理

CNNでもバッチ処理を行える。

データを分割し、1つのバッチに N個データがあるようにする。

各層を流れるデータを 4次元のデータとして格納する。

具体的には(バッチナンバー,チャンネル,高さ、横幅)といった順に格納する事で、バッチ処理に必要な準備ができる。

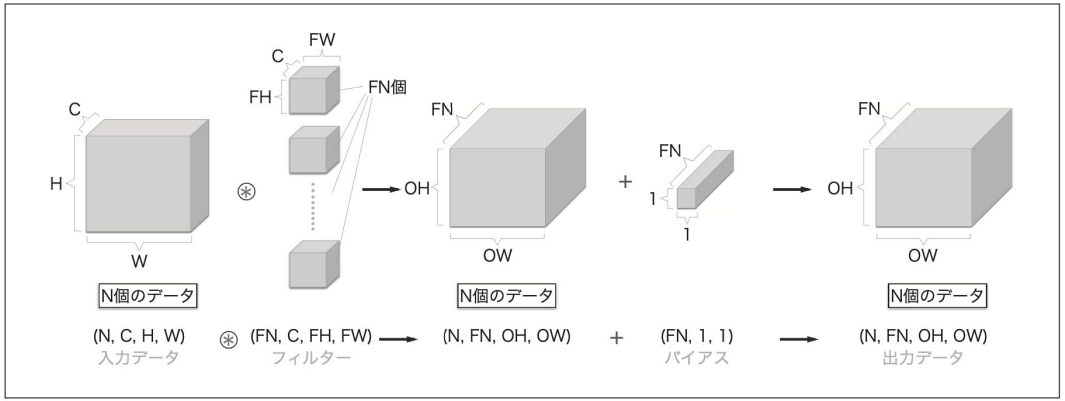
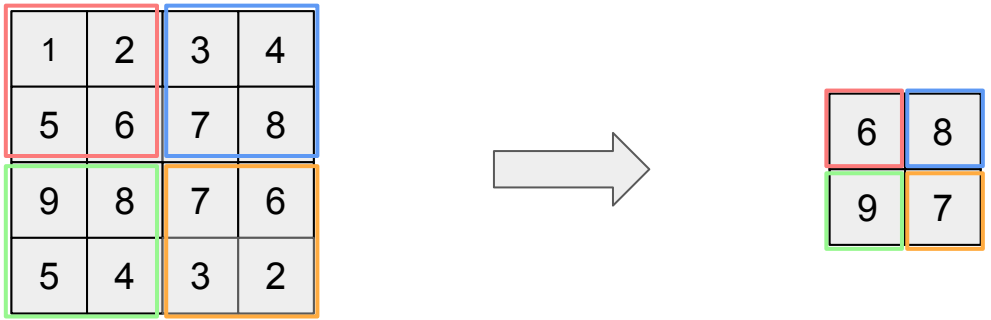


図 7-13　畳み込み演算の処理フロー（バッチ処理）

7.3 プーリング層 (概要, 特徴, 種類)

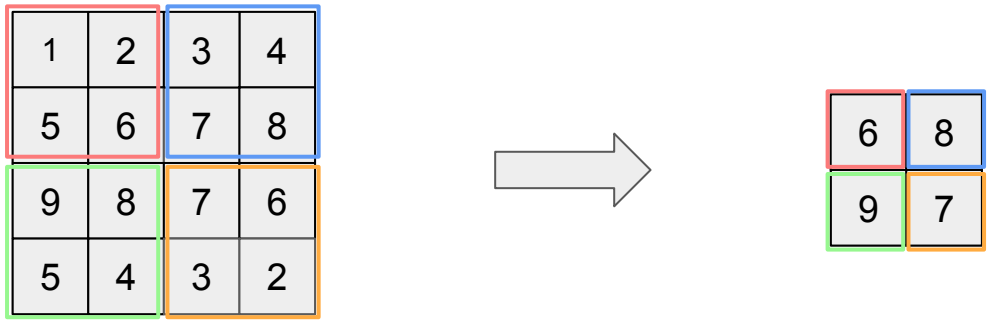
- 概要
  - プーリング (pooling)とは, 縦・横方向の空間を小さくする演算のこと
  - 下の図では, 同じ色で囲まれた領域に対して , プーリングの演算を行い , 4×4の行列を2×2の行列に小さくしている例。



4×4の行列　　2×2の行列

7.3 プーリング層 (概要, 特徴, 種類)

- プーリング
  - プーリングを行う時 , 対象の行列に対して , プーリングを行う領域のサイズを設定します。(2×2, 3×3, etc...)
  - 領域のサイズを設定したら , プーリングを行う次の領域までの移動距離を設定します。この移動距離をストライドと言います。
  - 例えば, 下の図では , 2×2のMaxプーリングをストライド 2で行ったと言います。



4×4の行列　　2×2の行列

7.3 プーリング層 (概要, 特徴, 種類)

- 特徴が3つあります。
  - ちなみにPoolは名詞でビリヤードって意味もある

7.3 プーリング層 (概要, 特徴, 種類)

- 特徴1・・・学習するパラメータがない
  - プーリング層では , 対象領域の要素を圧縮するだけの処理。

- 特徴2・・・チャンネル数が変化しない
  - 入力データはチャンネル毎に , 独立して処理が行われるためチャンネル数は変化しません

- 特徴3・・・微小な位置変化に対してロバスト
  - 入力データの小さなズレに対して , プーリングをすることによって , 同じような結果を返すことができる

7.3 プーリング層 (概要, 特徴, 種類)

- 特徴3・・・微小な位置変化に対してロバスト
  - 入力データの小さなズレ？

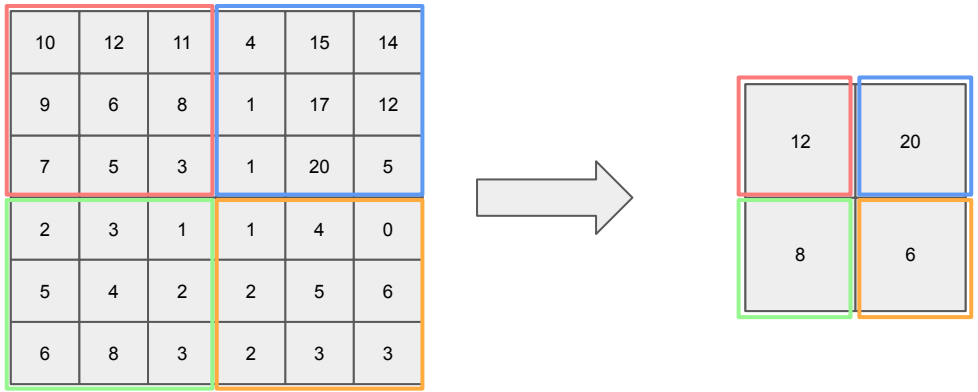
0	255	252	0
0	255	253	0
0	255	255	0
0	0	253	0

0	254	230	0
0	254	255	0
0	178	254	0
0	0	254	0

- 左の図では, MNISTデータセットの数字 1の画像に対して, Maxプーリングを行った結果を示しています。
- 結果を見ると, 同じような行列になっていることが確認　できます。
- しかし, 全てが同じような結果になるとは限りません

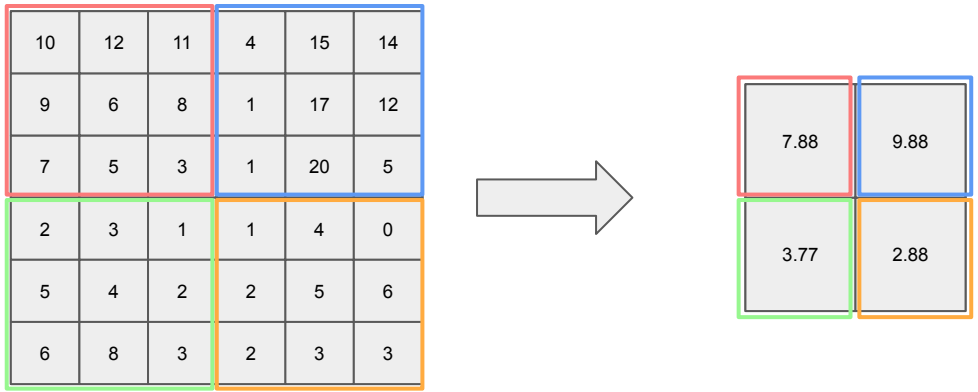
7.3 プーリング層 (概要, 特徴, 種類)

- プーリング処理の種類
  - Maxプーリング
    - 対象領域から, 最大値をとる
    - 下の図は, 3×3のMaxプーリングをストライド 3で行った時の例です。



7.3 プーリング層 (概要, 特徴, 種類)

- プーリング処理の種類
  - Averageプーリング
    - 対象領域の平均を計算した結果をとる
    - 下の図は, 3×3のAverageプーリングをストライド 3で行った時の例です。



7.4 Convolution／poolingレイヤの実装

畳み込み層とプーリング層を Pythonで実装する
5章で出てきたforward、backwardというメソッドを持たせモジュールとして利用できるようにする

7.4.1 4次元配列

CNNでは流れるデータは4次元である

例：データの形状が (10,1,28,28)とすると高さ28・横幅28で1チャンネルのデータが 10個

Pythonでの実装

```
>>> 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)
```

7.4.1 4次元配列

データにアクセスする方法、x[0]やx[1]のように書くだけ

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

一つ目のデータの 1チャンネルに目の空間データにアクセスする方法

```
>>> x[0, 0] # もしくは x[0][0]
```

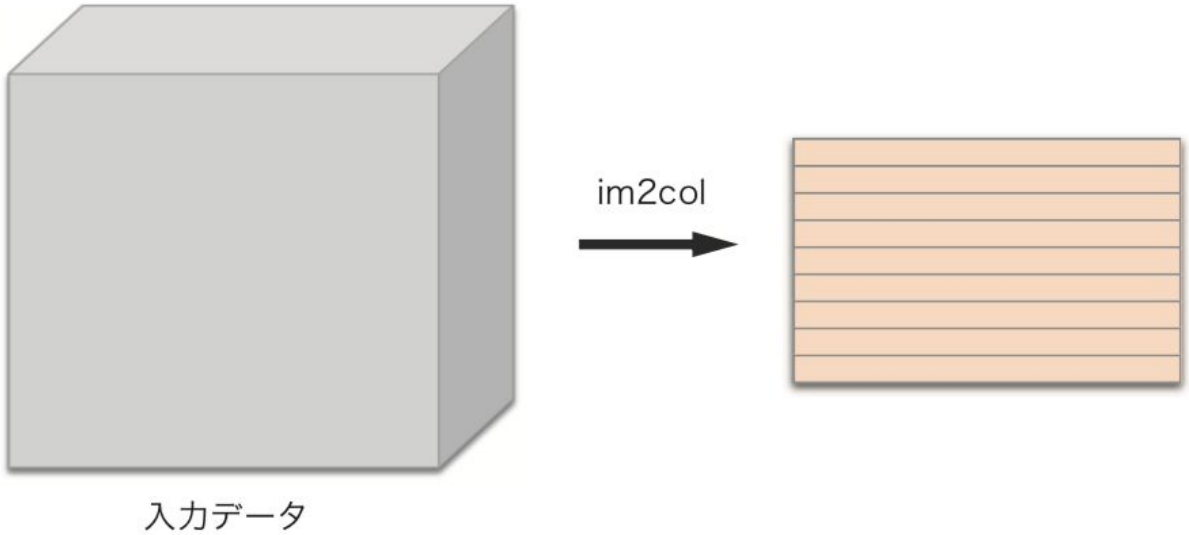
CNNでは4次元のデータを扱うことになります

畳み込みの演算は複雑になりそうですが im2colという関数によって簡単になる
(データが4次元あることで)

7.4.2 im2colによる展開

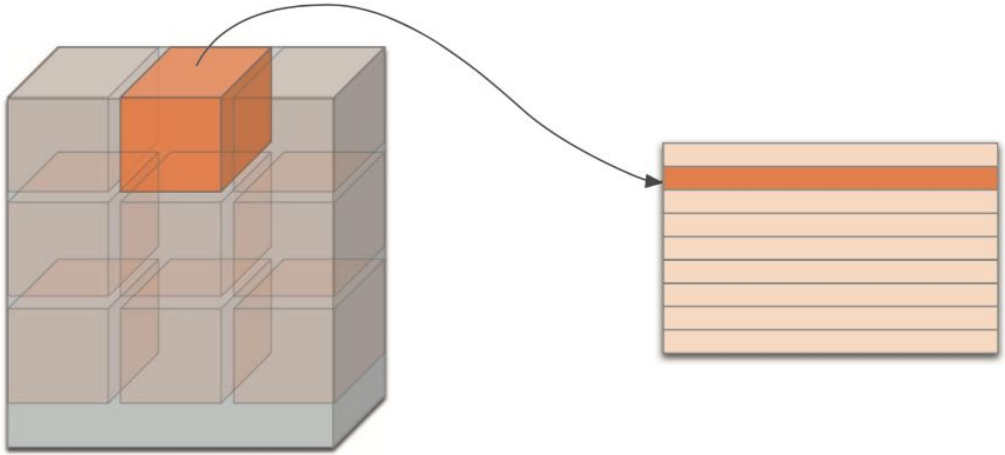
畳み込みの演算をfor文を使用して実装すると面倒な処理なってしまう。
NumPyではfor文を使うと処理が遅くなってしまうという欠点がある。

im2colを使用する
im2colはフィルターにとって都合のいいように **入力データを行列で展開する** 関数
下図は3次元の入力データに対して2次元の行列に変換したイメージ



7.4.2 im2colによる展開

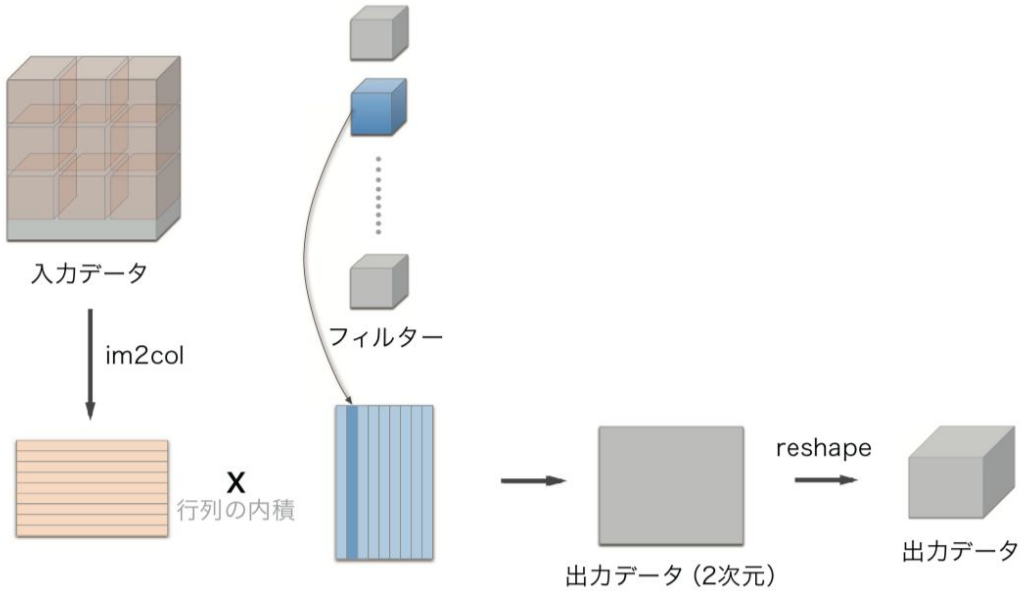
im2colは入力データに対してフィルターを適用する場所の領域 (3次元のブロック)を
横方向に1列に展開する。フィルターを適用する全ての場所で行う。



上図はフィルターの領域が重なっていないが実際は重なる場合がほとんどなため展開後の
要素の数はブロックの要素の数よりも多くなる。
そのため、**メモリを多く使用するという欠点** がある。
行列にまとめて計算することは **行列計算ライブラリを有効的に使用できるため**
高速に処理 できる

7.4.2 im2colによる展開

データを展開したあとは畳み込みフィルターを 1列に展開し、2つの行列の内積を
計算するだけ。



im2colの出力データは2次元なのでCNNのデータは4次元なので4次元データに整形する。

7.4.3 Convolutionレイヤの実装

im2colという関数はブラックボックスとして扱う
https://github.com/oreilly-japan/deep-learning-from-scratch/blob/master/common/util.py

im2colのインタフェース

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次元からなるデータ
- 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

フィルターサイズ、ストライド、パディングに考慮して入力データを 2次元配列に展開

7.4.3 Convolutionレイヤの実装

im2colの使用例

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

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)
```

例1
- バッチサイズ : 1
- チャンネル : 3
- データサイズ : 7×7

例2
- バッチサイズ : 10
- チャンネル : 3
- データサイズ : 7×7

2次元目の要素数75になる(フィルターが5×5)
フィルターの数の総和

例1ではバッチサイズが1でprintの結果が(9,75)

例2ではバッチサイズが10でprintの結果が(90,75)

7.4.3 Convolutionレイヤの実装

畳み込み層をConvolutionという名前のクラスで実装

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

初期化メソッドはフィルターとバイアス、ストライドとパディングを引数として受け取る。
フィルターは(FN, C, FH, FW)の4次元

- FN：Filter Number(フィルターの個数)
- C：Channel
- FH：Filter Height
- FW：Filter Widht

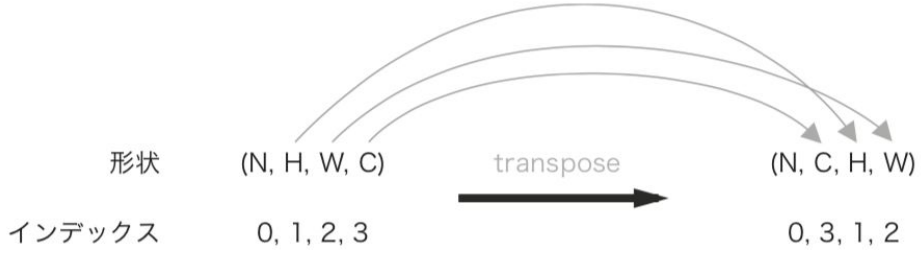
7.4.3 Convolutionレイヤの実装

```
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
```



コード中の太字の reshape(FN, -1)で−1が指定されているが辻褄が合うように要素数をまとめている
例(10,3,5,5)の形状の配列は要素が全部で 750個ある。reshape(10, -1)を行うと(10,75)の形状にまとめることができる。

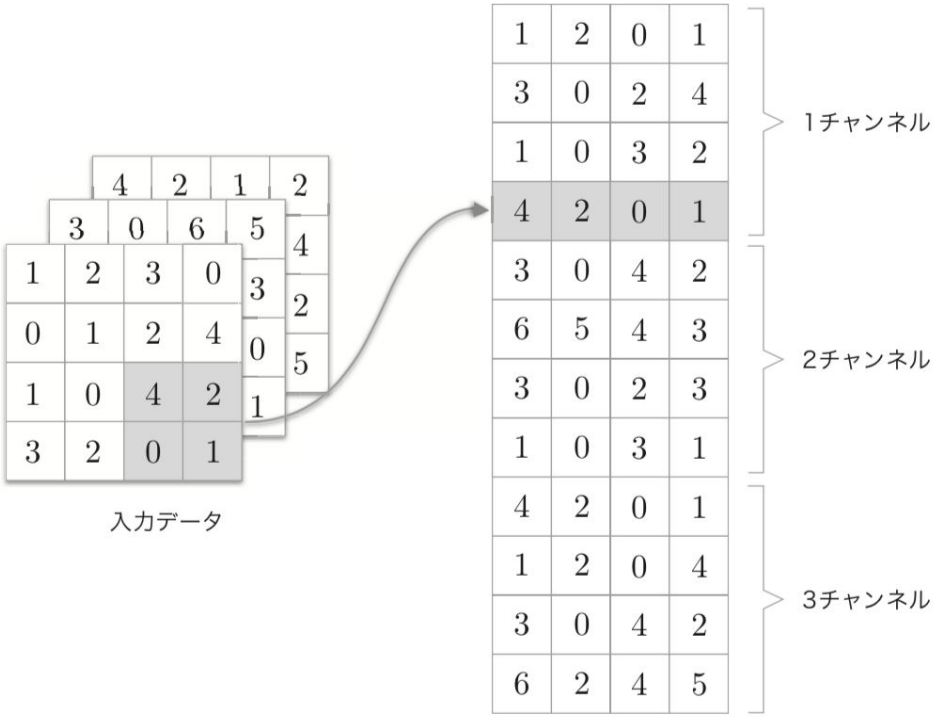
transpose(0,3,1,2)で軸の順番を入れ替えている。

逆伝播の実装は興味ある方は下記の URLへ
https://github.com/oreilly-japan/deep-learning-from-scratch/blob/master/common/layers.py

7.4.4 Poolingレイヤの実装

プーリングもim2colを使用して
データを展開するがチャンネル
方向には独立である。

行ごとに最大値を求め
適切な形状にする。



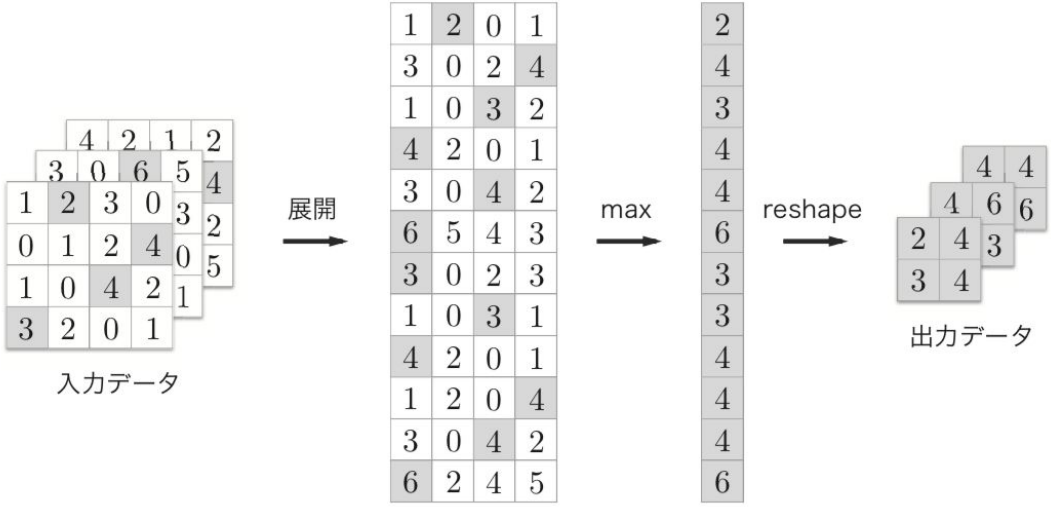
7.4.4 Poolingレイヤの実装

Poolingレイヤでは右図の
3段階の流れで行う。

step1
- 入力データを展開

step2
- 行ごとに最大値を求める

step3
- 適切な出力サイズに整形する



forward処理のソースコードは省略
backward処理は下記のURL
https://github.com/oreilly-japan/deep-learning-from-scratch/blob/master/common/layers.py

7.5 CNN実装(概要, 初期化, 推論と損失関数, 勾配を求める)

実装するCNNのネットワーク構成は図 7-5-1
クラス名をSimpleConvNet とする

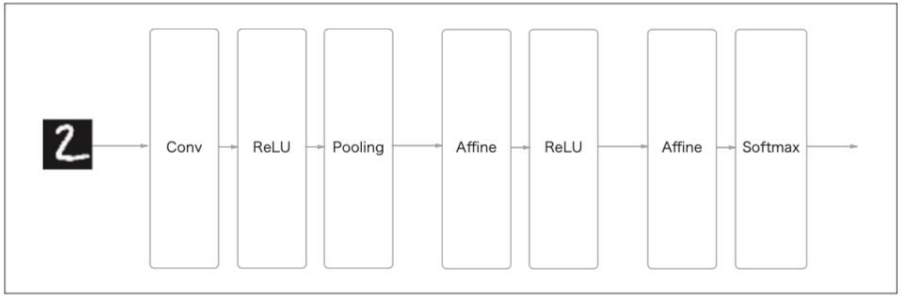


図7-5-1:CNNのネットワーク構成

初期化(_init_)から実装していく

引数は図7-5-2のようになっている。

畳み込み層のハイパーパラメータは conv_param というディクショナリ型で与えられる。

実装は次スライドから3つの画像に分けて解説

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

図7-5-2:初期化の引数

7.5 CNN実装(概要, 初期化, 推論と損失関数, 勾配を求める)

- 図7-5-3が初期化の実装の1つ目の画像であるここでは初期化の引数として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して保存している。

- 図7-5-4は初期化の実装の2つ目となっているここでは重みパラメータの初期化を行なっている。学習の必要なパラメータは畳み込み層の重みと、全結合層2つの重みとバイアスのみである。それぞれれのパラメータはディクショナリ型の変数paramsに保存される。
重みはガウス分布のスケールに基づいてランダムで初期値を設定している。

```
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))
```

図7-5-3:初期化の実装(1)

```
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)
```

図7-5-4:初期化の実装(2)

7.5 CNN実装(概要, 初期化, 推論と損失関数, 勾配を求める)

- 図7-5-5が初期化の実装3つ目である。
  これまでに作ったレイヤークラスを順番にディクショナリ型変数layersに追加していく
  ただし、最後のsoftmax層に相当するSoftmaxWithLossレイヤはlastlayerという別の変数に追加する。

- 続いて推論を行うpredictメソッドと損失関数の値を求めるlossメソッドの実装を行う。
  具体的なコードは図7-5-6のようになる。
  lossメソッドではpredictメソッドのforward処理に加えて、SoftmaxWithLossレイヤのforward処理まで行う。
  ここでの引数はxが入力データ、tが教師データである。

```
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'])
```

図7-5-5:初期化の実装(3)

```
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)
```

図7-5-6:predictとlossの実装

7.5 CNN実装(概要, 初期化, 推論と損失関数, 勾配を求める)

- 続いて誤差逆伝播法によって勾配を求めるコードを実装する。具体的には図 7-5-7のようになる。
  ここではそれぞれのレイヤで逆伝播を呼び出し、ディクショナリ型変数 gradsに各重みパラメータの値を格納する。
  ここで重要なのはlayer.reverseによってlayerの要素の順番が逆になっていることである。これによってfor文を使ってbackwardを実行しても問題なく勾配を求めることができる。

- この後の学習のコードは「 4.5 学習のアルゴリズムの実行」で説明したものとほぼ同じなのでコードの掲載は省略(ソースコードはch07/train_convnet.pyに掲載)

```
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
```

図7-5-7 : 勾配を求めるコード

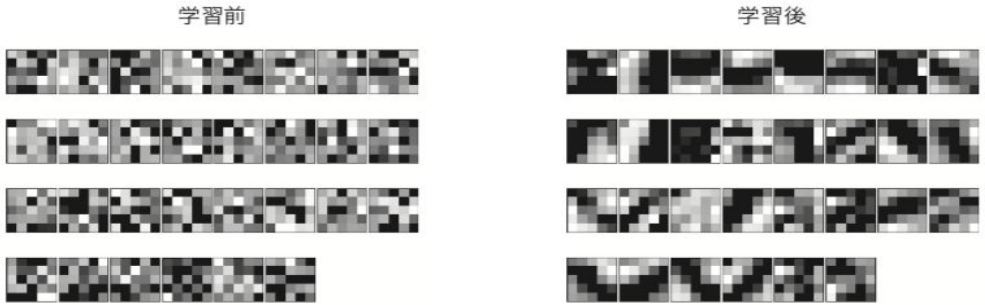
7.6 CNNの可視化

- 概要
  - CNNで用いられている畳み込み層は何を見ているのか(画像のどの部分に反応しているのか)

- 目次
  - 1層目の重みの可視化
  - 階層構造による情報抽出

7.6.1 1層目の重みの可視化

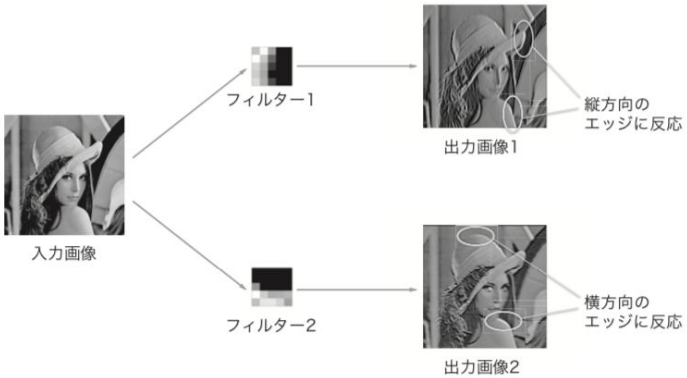
- 学習前のフィルターはランダムに初期化されているため、規則性がない
- 学習後のフィルターは規則性があり、情報を抽出しやすくなっている
  例：白から黒へグラデーションを伴っているフィルター、 blobを持つフィルター
  (blobとは局所的に塊のある領域のこと)



CNNの1層目の畳み込み層を学習前と学習後で比較した図

7.6.1 1層目の重みの可視化

- 左半分が白で右半分が黒のフィルターは縦方向のエッジに反応
- 下半分が白で上半分が黒のフィルターは横方向のエッジに反応
- ブロブを持つフィルターは出力画像のブロブに反応？

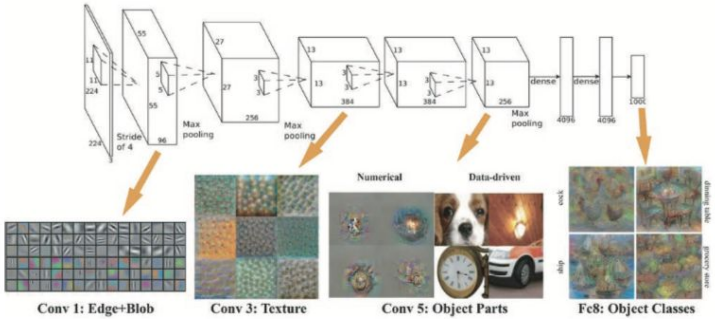


規則性のあるフィルターを適用してどの様な情報を抽出できるか表した図

7.6.2 階層構造による情報抽出

- AlexNetとは・・・畳み込み層とプーリング層が複数重なり、最後に全結合層を利用しているネットワーク構造

- 各層で主にニューロンが反応する部分 (抽出される情報)
  - 1層目の畳み込み層→エッジやブロブ
  - 3層目の畳み込み層→テクスチャ3(3次元の物体表面の模様？)
  - 5層目の畳見込み層→認識したい物体のパーツ
  - 全結合層→認識したい物体のクラス(犬や車など)



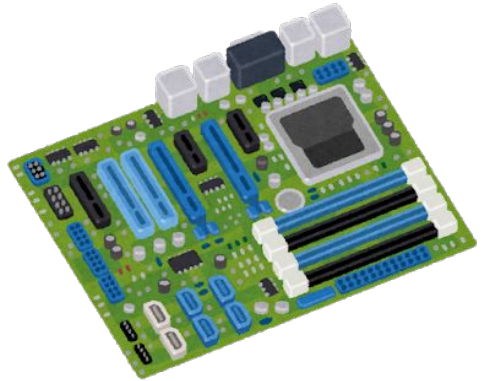
ネットワーク構造が AlexNetである一般物体認識 (犬や車など)を行う8層のCNNの図

7.6.2 階層構造による情報抽出

- CNNでは層が深くなるに従って、より複雑で抽象化された情報が抽出される
- つまり、ニューロンは層が深くなるに従って、単純な形状から高度な情報へと変化していくということ

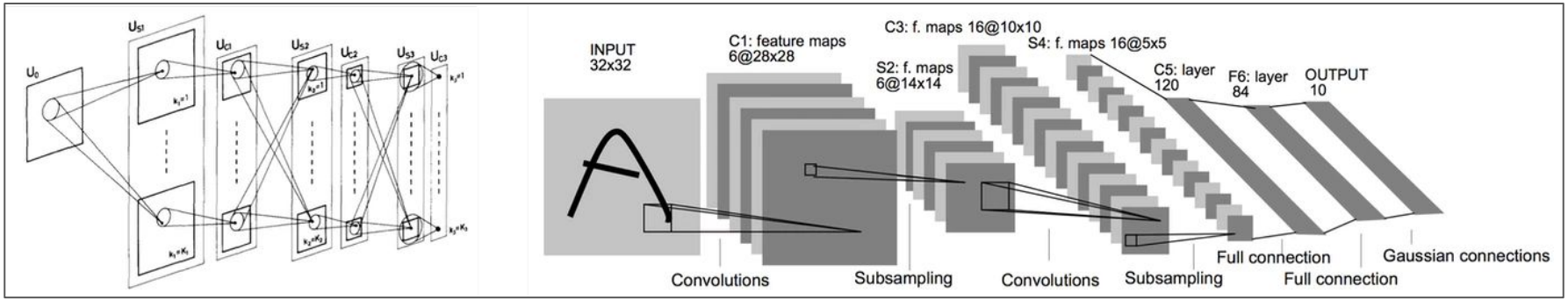
7.7 代表的なCNN(概要, LeNet, AlexNet)

- 概要
  - LeNetとAlexNetの2つのCNNについて紹介している。
  - 上記の2つのネットワークでは , 大きな違いはないことを本書では述べている。
  - 変化したことは取り巻く環境やコンピュータ技術の進歩と解説。
    - ビッグデータやGPUがニューラルネットワークの発展に大きく貢献している



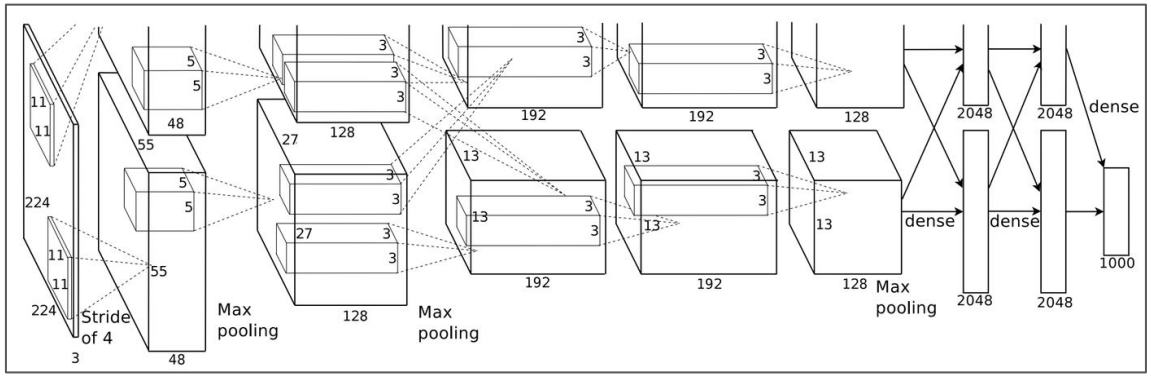
7.7 代表的なCNN(概要, LeNet, AlexNet)

- LeNet
  - CNNの原型とも言われているニューラルネットワーク
  - 畳み込み層とプーリング層を連続して行い，最後に全結合層を経て結果を出力
    - ただ，プーリング層では，単に「要素を間引く」ことだけの処理をしており，サブサンプリング層としている
  - 活性化関数は現在主流の ReLU関数ではなく，シグモイド関数を使用



7.7 代表的なCNN(概要, LeNet, AlexNet)

- AlexNet
  - 基本的なネットワークの構成は , LeNetとあまり変わらない
  - 活性化関数には , ReLU関数を使用
  - Dropoutを使用(過学習を抑えることが可能)
  - LRN(Local Response Nomalization)という局所的正規化を行う層を使用
    - 特徴マップと同一の位置にあり , 隣接する出力の結果を使って , 自身の出力の値を正規化する



7.8　まとめ

- CNNは、これまでの全結合層のネットワークに対して、畳み込み層とプーリング層が新たに加わる。
- 畳み込み層とプーリング層は、im2col（画像を行列に展開する関数）を用いるとシンプルで効率の良い実装ができる。
- CNNの可視化によって、層が深くなるにつれて高度な情報が抽出されていく様子が分かる。
- CNNの代表的なネットワークには、LeNetとAlexNetがある。
- ディープラーニングの発展に、ビッグデータと GPUが大きく貢献している。