# 畳み込みニューラルネット(1)

前田利之（まえだ　としゆき）
aipr@e-chan.jp

阪南大学　経営情報学部

(2024.11.29)

# 畳み込みニューラルネット

- 畳み込みニューラルネット (Convolutional Neural Network, CNN) とは？
  -  の基礎
  - 画像認識、音声認識等、いろいろなところで使われている

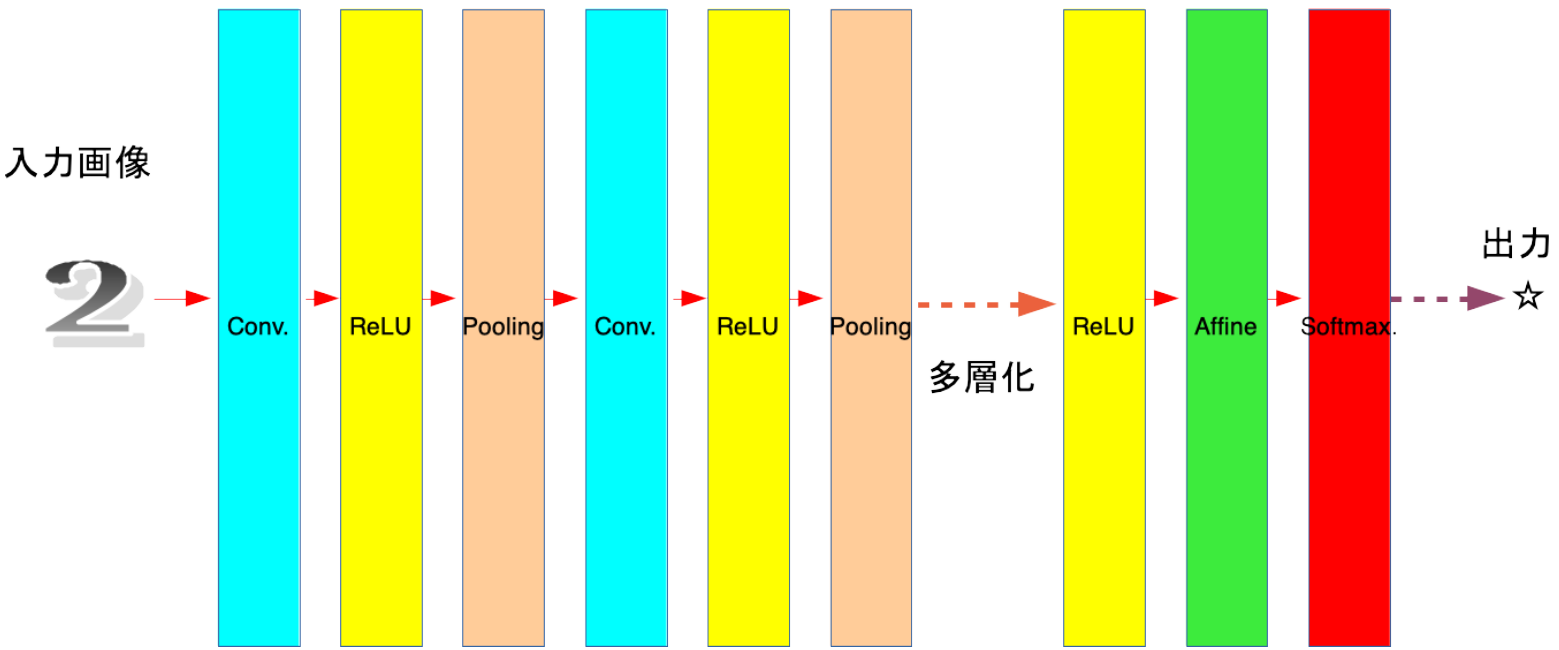
# 畳み込みニューラルネット

- 畳み込みニューラルネット (Convolutional Neural Network, CNN) とは？
  - ディープラーニング の基礎
  - 画像認識、音声認識等、いろいろなところで使われている

# 畳み込みニューラルネット

- 畳み込みニューラルネット (Convolutional Neural Network, CNN) とは？
  - ディープラーニング (Deep Learning) の基礎
  - 画像認識、音声認識等、いろいろなところで使われている

# 畳み込みニューラルネット



# 畳み込みニューラルネット

- 特徴的なレイヤは
  -  レイヤ
  -  レイヤ
- これらが何層にも（深く）重なっている=


# 畳み込みニューラルネット

- 特徴的なレイヤは
  - Convolution （畳み込み） レイヤ
  - [　　] レイヤ
- これらが何層にも（深く）重なっている=

[　　] [　　]

# 畳み込みニューラルネット

- 特徴的なレイヤは
  - Convolution （畳み込み） レイヤ
  - Pooling （プーリング） レイヤ
- これらが何層にも（深く）重なっている=

# 畳み込みニューラルネット

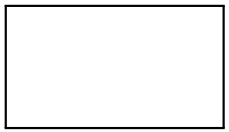
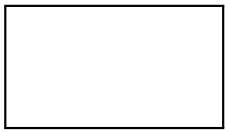
- 特徴的なレイヤは
  - Convolution （畳み込み） レイヤ
  - Pooling （プーリング） レイヤ
- これらが何層にも（深く）重なっている=

ディープラーニング

# 畳み込みニューラルネット

- 特徴的なレイヤは
  - Convolution （畳み込み） レイヤ
  - Pooling （プーリング） レイヤ
- これらが何層にも（深く）重なっている= ディープラーニング （深層学習）

# 畳み込みニューラルネット

- 先週までのニューラルネット= 
  (Affine レイヤ)
  - 隣接層のニューロンは全て連結されていた
  - 出力数は任意
- この問題点：データの  が  されている
  - MNIST のデータの場合、元は 28x28 なのを 1x784 のデータとして Affine レイヤの入力にしていた
  - ⇒ 元々ある  を捨てている (!)

# 畳み込みニューラルネット

- 先週までのニューラルネット= **全結合層** (Affine レイヤ)
  - 隣接層のニューロンは全て連結されていた
  - 出力数は任意
- この問題点：データの [　　] が [　　] されている
  - MNIST のデータの場合、元は 28x28 なのを 1x784 のデータとして Affine レイヤの入力にしていた
  - ⇒ 元々ある [　　　　] を捨てている (!)

# 畳み込みニューラルネット

- 先週までのニューラルネット= 全結合層 (Affine レイヤ)
  - 隣接層のニューロンは全て連結されていた
  - 出力数は任意
- この問題点：データの 形状 が [　　] されている
  - MNIST のデータの場合、元は 28x28 なのを 1x784 のデータとして Affine レイヤの入力にしていた
  - ⇒ 元々ある [　　] を捨てている (!)

# 畳み込みニューラルネット

- 先週までのニューラルネット= **全結合層** (Affine レイヤ)
  - 隣接層のニューロンは全て連結されていた
  - 出力数は任意
- この問題点：データの **形状** が **無視** されている
  - MNIST のデータの場合、元は 28x28 なのを 1x784 のデータとして Affine レイヤの入力にしていた
  - ⇒ 元々ある [　　] を捨てている (!)

# 畳み込みニューラルネット

- 先週までのニューラルネット= **全結合層** (Affine レイヤ)
  - 隣接層のニューロンは全て連結されていた
  - 出力数は任意
- この問題点：データの **形状** が **無視** されている
  - MNIST のデータの場合、元は 28x28 なのを 1x784 のデータとして Affine レイヤの入力にしていた
  - ⇒ 元々ある **空間情報** を捨てている (!)

# 畳み込みニューラルネット

- 畳み込みニューラルネットは
  [　　]する
- なので、画像などをより正しく
  [　　]

# 畳み込みニューラルネット

- 畳み込みニューラルネットは
  形状を維持する
- なので、画像などをより正しく

# 畳み込みニューラルネット

- 畳み込みニューラルネットは
  形状を維持する
- なので、画像などをより正しく
  理解できる（可能性がある）

# 畳み込みニューラルネット

- 畳み込み層の入出力：
  - 入力：入力特徴マップ
  - 出力：出力特徴マップ
- 入力データとフィルタとの で出力を導く（後述）

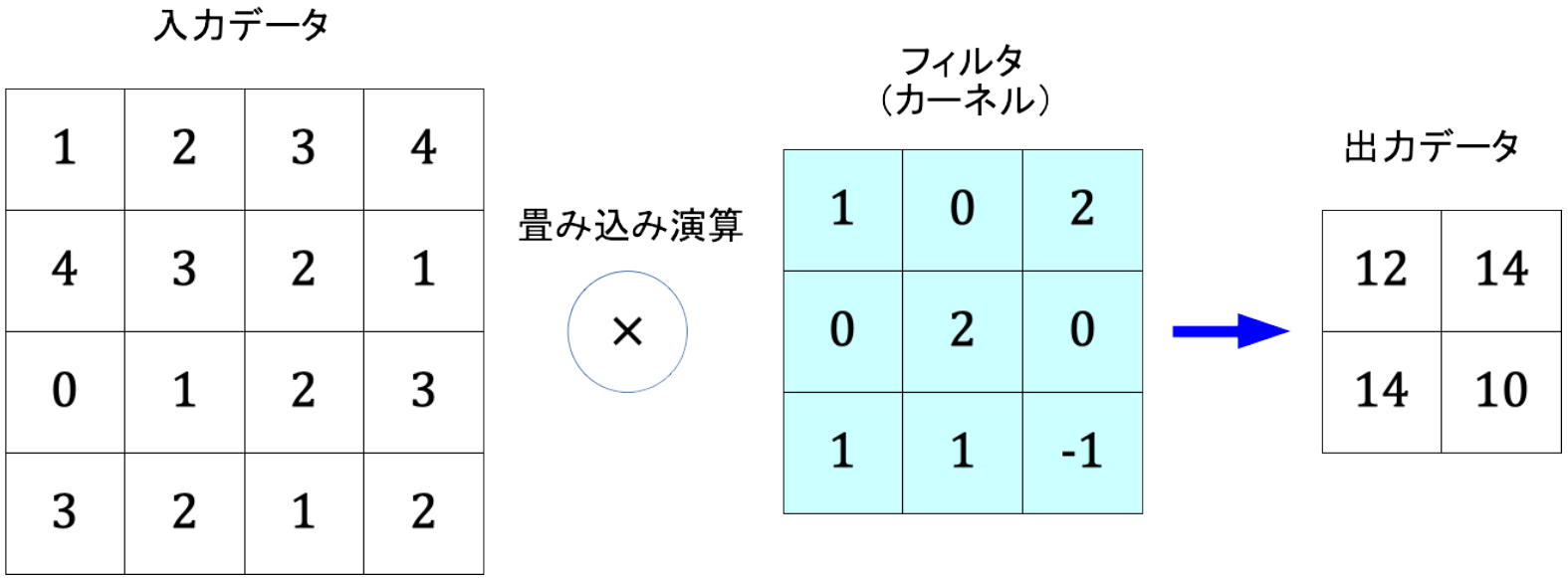
# 畳み込みニューラルネット

- 畳み込み層の入出力：「特徴マップ」
  - 入力：入力特徴マップ
  - 出力：出力特徴マップ
- 入力データとフィルタとの　　で出力を導く（後述）

# 畳み込みニューラルネット

- 畳み込み層の入出力：「特徴マップ」
  - 入力：入力特徴マップ
  - 出力：出力特徴マップ
- 入力データとフィルタとの積和計算で出力を導く（後述）

# 畳み込み演算


入力データ
1 2 3 4
4 3 2 1
0 1 2 3
3 2 1 2
畳み込み演算
×
フィルタ
（カーネル）
1 0 2
0 2 0
1 1 -1
出力データ
12 14
14 10

# 畳み込み演算＝画像処理での「フィルタ演算」

- 入力データにフィルタを適用
- 入力データ、フィルタともに［　　］がある（行列）
  1. 入力データに対してフィルタの［　　］（操作対象の枠）を一定の間隔で［　　］させる
  2. それぞれ対応する要素を掛け、その総和を求める＝［　　］（さらににバイアスを加える）
  3. 結果を対応する場所へ格納する
  4. 上の処理を全ての場所で行なう

# 畳み込み演算＝画像処理での「フィルタ演算」

- 入力データにフィルタを適用
- 入力データ、フィルタともに **縦・横** がある（行列）
  1. 入力データに対してフィルタの　　（操作対象の枠）を一定の間隔で　　させる
  2. それぞれ対応する要素を掛け、その総和を求める＝
     （さらににバイアスを加える）
  3. 結果を対応する場所へ格納する
  4. 上の処理を全ての場所で行なう

# 畳み込み演算＝画像処理での「フィルタ演算」

- 入力データにフィルタを適用
- 入力データ、フィルタともに **縦・横** がある（行列）
  1. 入力データに対してフィルタの **ウィンドウ**（操作対象の枠）を一定の間隔で [　　] させる
  2. それぞれ対応する要素を掛け、その総和を求める＝ [　　]
     （さらににバイアスを加える）
  3. 結果を対応する場所へ格納する
  4. 上の処理を全ての場所で行なう

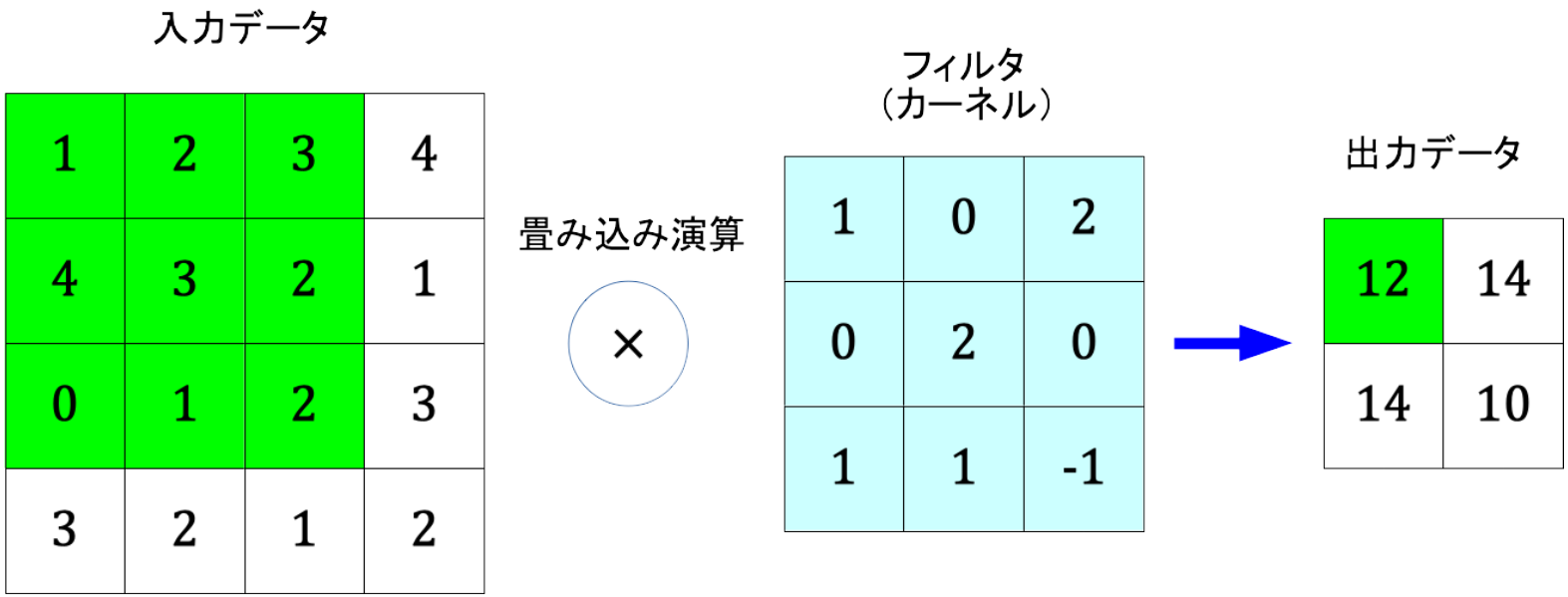
# 畳み込み演算＝画像処理での「フィルタ演算」

- 入力データにフィルタを適用
- 入力データ、フィルタともに **縦・横** がある（行列）
  1. 入力データに対してフィルタの **ウィンドウ**（操作対象の枠）を一定の間隔で **スライド** させる
  2. それぞれ対応する要素を掛け、その総和を求める＝
     
     （さらににバイアスを加える）
  3. 結果を対応する場所へ格納する
  4. 上の処理を全ての場所で行なう

# 畳み込み演算＝画像処理での「フィルタ演算」

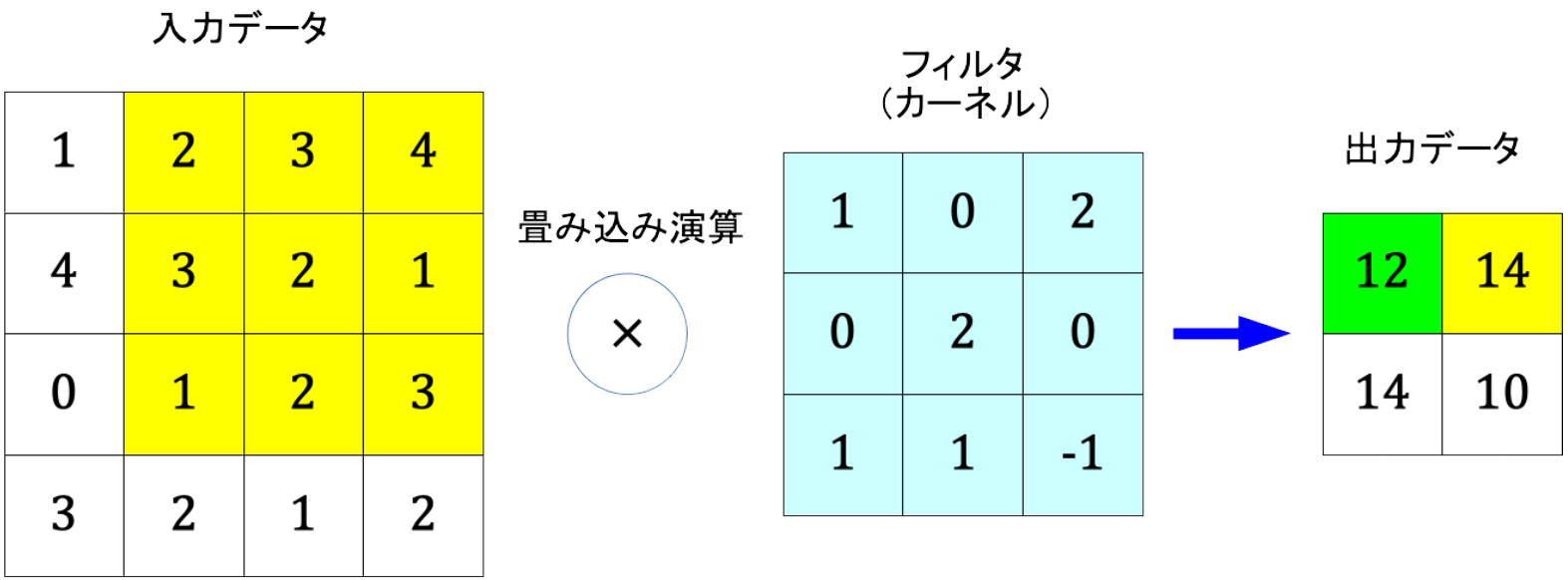
- 入力データにフィルタを適用
- 入力データ、フィルタともに **縦・横** がある（行列）
  1. 入力データに対してフィルタの **ウィンドウ**（操作対象の枠）を一定の間隔で **スライド** させる
  2. それぞれ対応する要素を掛け、その総和を求める＝ **積和計算**
     （さらににバイアスを加える）
  3. 結果を対応する場所へ格納する
  4. 上の処理を全ての場所で行なう

# 畳み込み演算過程



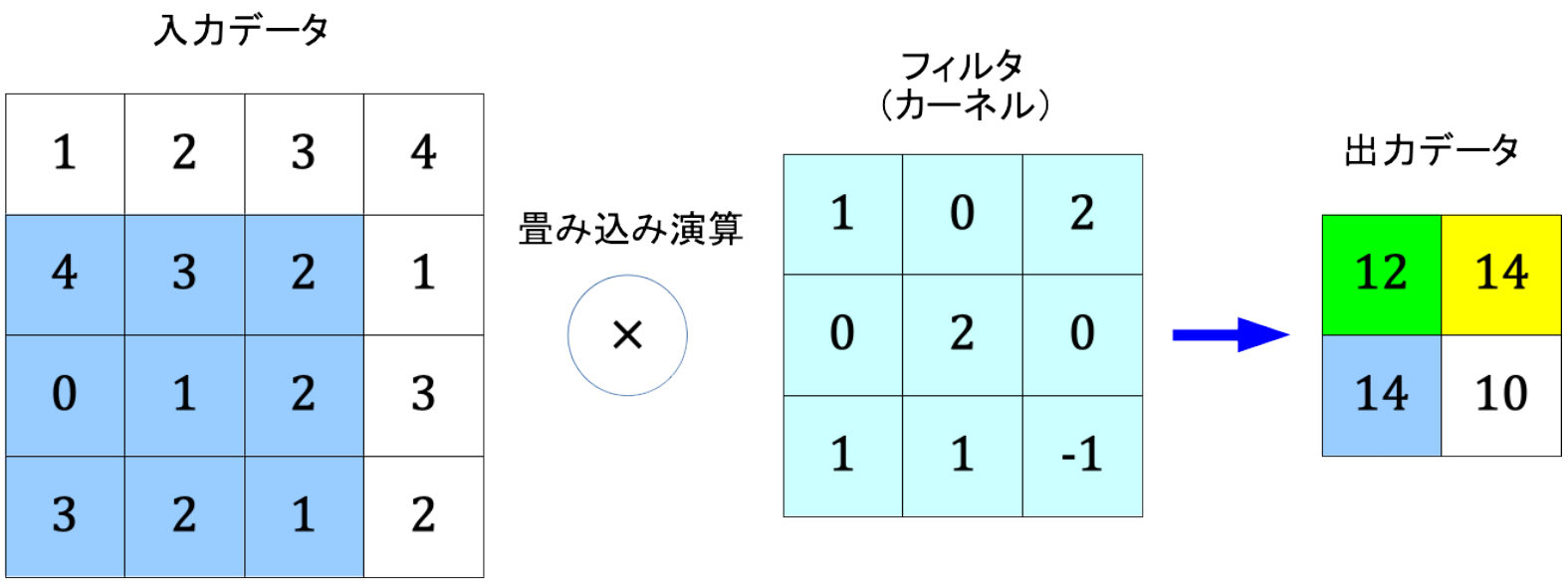

1x1+2x0+3x2+4x0+3x2+2x0+0x1+1x1+(-1)x2

# 畳み込み演算過程



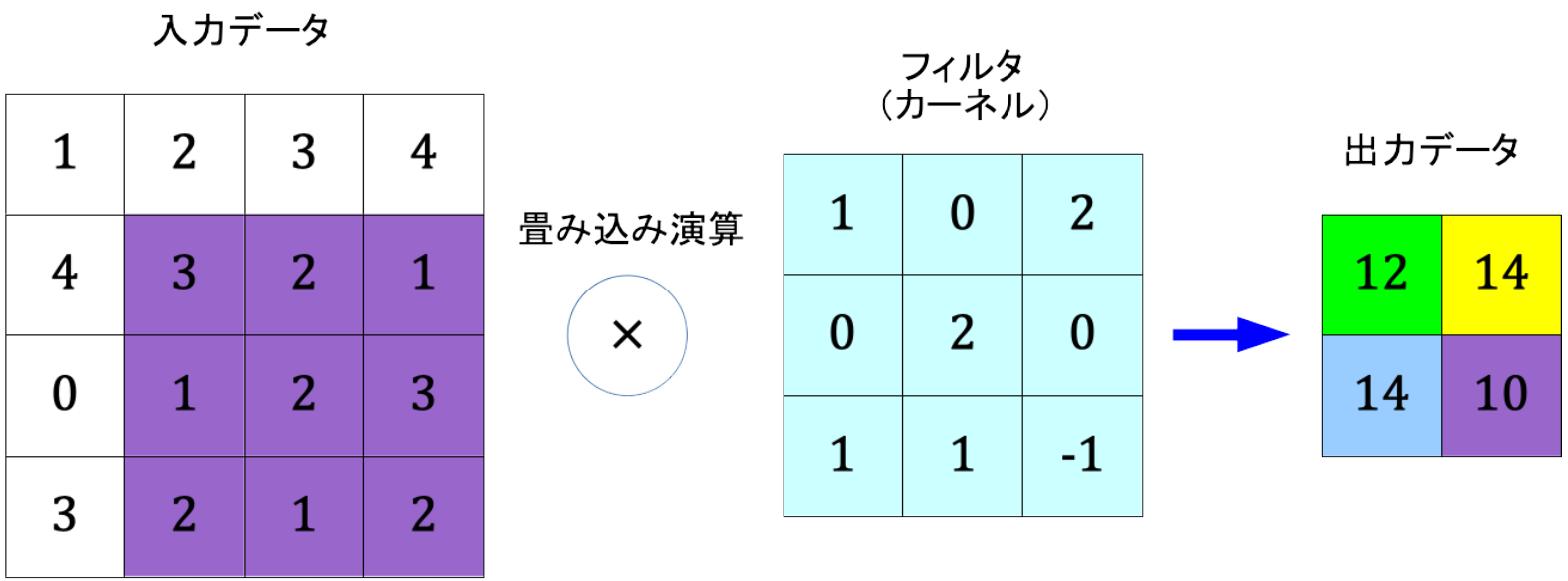

$$2x1+3x0+4x2+3x0+2x2+1x0+1x1+2x1+3x(-1)$$

# 畳み込み演算過程




$$4x1+3x0+2x2+0x0+1x2+2x0+3x1+2x1+1x(-1)$$

# 畳み込み演算過程




$$3x1+2x0+1x2+1x0+2x2+3x0+2x1+1x1+2x(-1)$$

# 畳み込み演算の演習

☆ 以下（リンクあり）を、Colab にコピー＆ペーストして実行する。

```
import numpy as np
ind  = np.array([[1,2,3,4],
                 [4,3,2,1],
                 [0,1,2,3],
                 [3,2,1,2]])
filt = np.array([[1,0,2],
                 [0,2,0],
                 [1,1,-1]])
outd = np.zeros((2,2))
for i in range(2):
  for j in range(2):
    outd[i][j] = np.sum(ind[i:i+3,j:j+3]*filt)
print(outd)
```

# パディング (Padding)

- 入力データの  を埋めた上で、畳み込みをする
- なぜかと言うと、以下を回避するため：
  - 前の例では、4x4 のデータに 3x3 のフィルタを適用し 2x2 の出力になった
  - つまり、出力データは  →繰り返すと、サイズ１になってしまう
  - すると、それ以上処理が出来なくなる
- 前の例では幅１つだけパディングすると、出力も 4x4 のサイズで保たれる

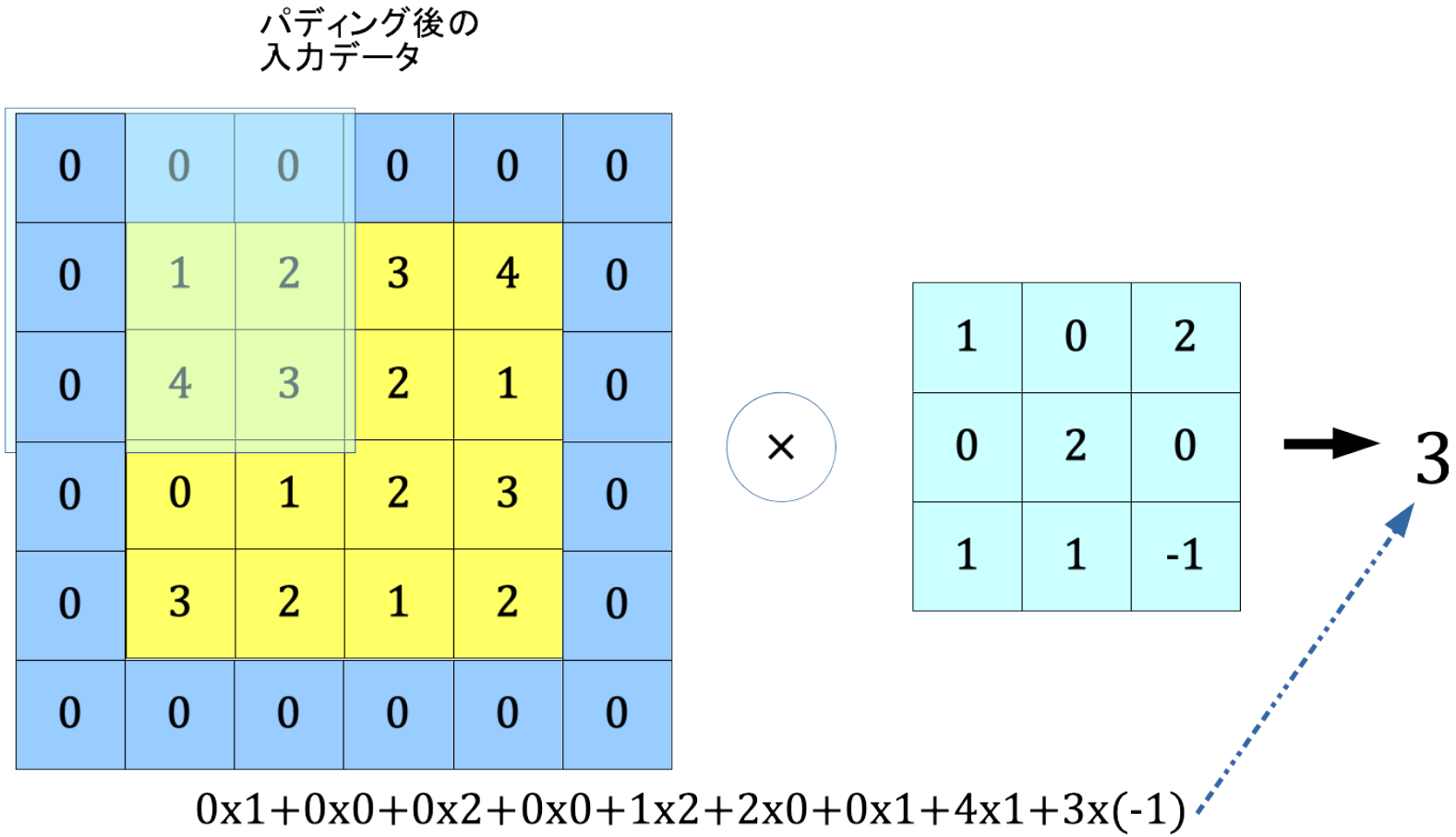
# パディング (Padding)

- 入力データの 周囲に０ を埋めた上で、畳み込みをする
- なぜかと言うと、以下を回避するため：
  - 前の例では、4x4 のデータに 3x3 のフィルタを適用し 2x2 の出力になった
  - つまり、出力データは [　　] →繰り返すと、サイズ１になってしまう
  - すると、それ以上処理が出来なくなる
- 前の例では幅１つだけパディングすると、出力も 4x4 のサイズで保たれる

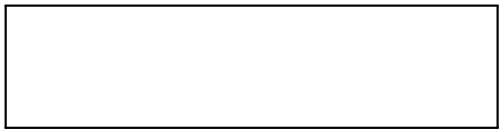
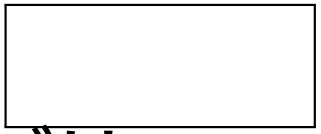
# パディング (Padding)

- 入力データの **周囲に０** を埋めた上で、畳み込みをする
- なぜかと言うと、以下を回避するため：
  - 前の例では、4x4 のデータに 3x3 のフィルタを適用し 2x2 の出力になった
  - つまり、出力データは **入力より小さい** →繰り返すと、サイズ１になってしまう
  - すると、それ以上処理が出来なくなる
- 前の例では幅１つだけパディングすると、出力も 4x4 のサイズで保たれる

# パディング


パディング後の
入力データ
0 0 0 0 0 0
0 1 2 3 4 0
0 4 3 2 1 0
0 0 1 2 3 0
0 3 2 1 2 0
0 0 0 0 0 0
×
1 0 2
0 2 0
1 1 -1
3
0x1+0x0+0x2+0x0+1x2+2x0+0x1+4x1+3x(-1)

# ストライド (stride)

- フィルタを適用する 
  - つまり、１つづつずらすとは限らない、ということ
- ストライドを増やすと出力サイズは  なり、パディングを増やすと出力サイズは  なることに注意

# ストライド (stride)

- フィルタを適用する **位置の間隔**
  - つまり、１つづつずらすとは限らない、ということ
- ストライドを増やすと出力サイズは [　　] なり、パディングを増やすと出力サイズは [　　] なることに注意

# ストライド (stride)

- フィルタを適用する **位置の間隔**
  - つまり、１つづつずらすとは限らない、ということ
- ストライドを増やすと出力サイズは **小さく** なり、パディングを増やすと出力サイズは　　なることに注意

# ストライド (stride)

- フィルタを適用する **位置の間隔**
  - つまり、１つづつずらすとは限らない、ということ
- ストライドを増やすと出力サイズは **小さく** なり、パディングを増やすと出力サイズは **大きく** なることに注意

# ３次元データの畳み込み演算

- 例えばカラー画像の場合、縦 (Height), 横 (Width) の２次元データではなく、 (RGB, CMY 等）もあわせた３次元データを扱う必要がある
- その場合、フィルターは１枚のフィルタですますことも出来るし、３枚用意して、その加算により２次元出力を得ることも出来る

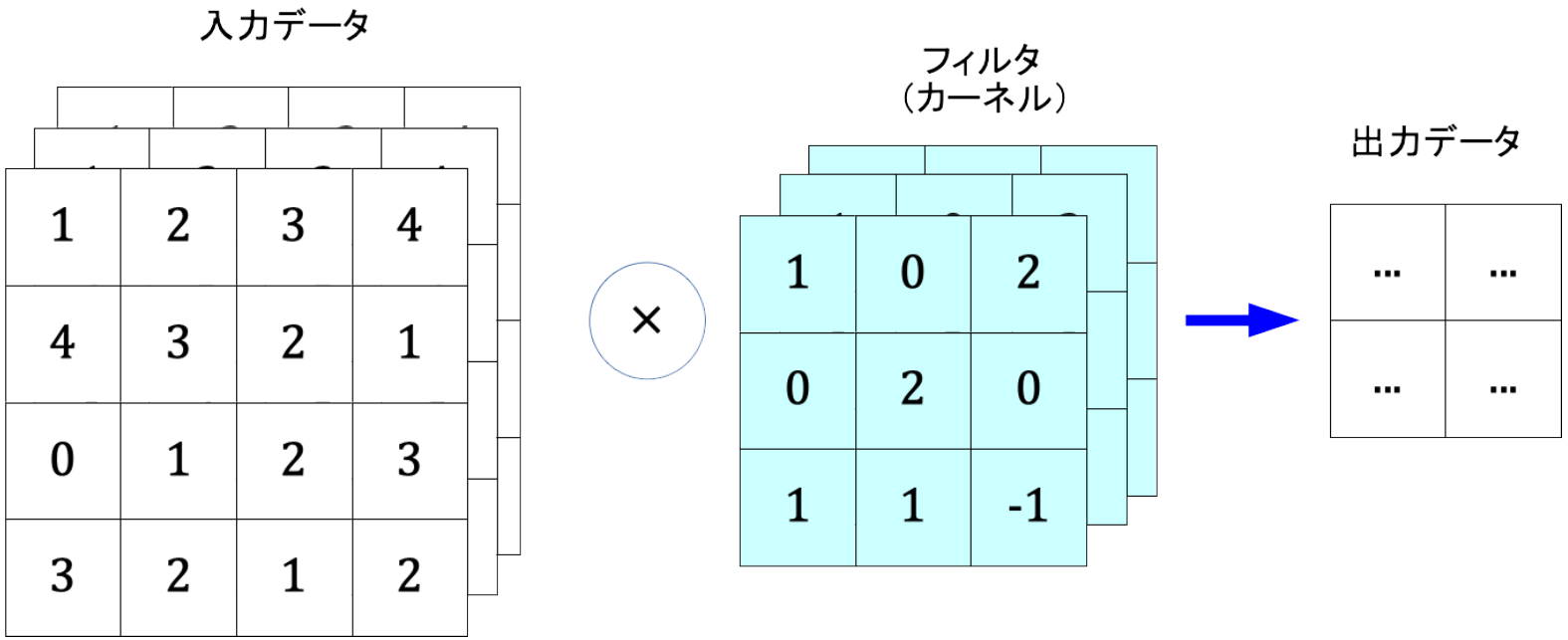
# ３次元データの畳み込み演算

- 例えばカラー画像の場合、縦 (Height), 横 (Width) の２次元データではなく、**色の３要素** (RGB, CMY 等）もあわせた３次元データを扱う必要がある
- その場合、フィルターは１枚のフィルタですますことも出来るし、３枚用意して、その加算により２次元出力を得ることも出来る

# ３次元データの畳み込み演算

- ３次元目の軸をチャネル (channel) と表すと、データは以下の次元となる
  - 入力データ: (Channel, Height, Width)
  - フイルタ: (Channel, Filter-Height, Filter-Width)
    あるいは (Filter-Number, Channel, Filter-Height, Filter-Width)
  - 出力データ: (1, Output-Height, Output-Width)

# 3 次元データの畳み込み演算



# バッチ処理＝４次元データの畳み込み演算

- これまで同様、入力データを  にまとめたバッチ処理に対応したい
- すると、さらに次元が増える
  - 入力データ: (Batch-Number, Channel, Height, Width)
  - フイルタ: (Filter-Number, Channel, Filter-Height, Filter-Width)
  - 出力データ: (Batch-Number, Filter-Number, Output-Height, Output-Width)

# バッチ処理＝４次元データの畳み込み演算

- これまで同様、入力データを **一束** にまとめたバッチ処理に対応したい
- すると、さらに次元が増える
  - 入力データ: (Batch-Number, Channel, Height, Width)
  - フイルタ: (Filter-Number, Channel, Filter-Height, Filter-Width)
  - 出力データ: (Batch-Number, Filter-Number, Output-Height, Output-Width)

# プーリング層

- 縦横方向の空間を [　　] する
- 一般的に、プーリングのウィンドウとストライドは [　　] にする
- 画像認識では [　　] Pooling、すなわちその枠内で一番大きいデータを出力とすることが多い
  - 他には [　　] Pooling (平均をとる）等がある

# プーリング層

- 縦横方向の空間を 小さく する
- 一般的に、プーリングのウィンドウとストライドは [　　] にする
- 画像認識では [　　] Pooling、すなわちその枠内で一番大きいデータを出力とすることが多い
  - 他には [　　] Pooling (平均をとる）等がある

# プーリング層

- 縦横方向の空間を 小さく する
- 一般的に、プーリングのウィンドウとストライドは 同じ にする
- 画像認識では [　　] Pooling、すなわちその枠内で一番大きいデータを出力とすることが多い
  - 他には [　　　　] Pooling (平均をとる）等がある

# プーリング層

- 縦横方向の空間を 小さく する
- 一般的に、プーリングのウィンドウとストライドは 同じ にする
- 画像認識では Max Pooling、すなわちその枠内で一番大きいデータを出力とすることが多い
  - 他には [ ] Pooling (平均をとる）等がある

# プーリング層

- 縦横方向の空間を 小さく する
- 一般的に、プーリングのウィンドウとストライドは 同じ にする
- 画像認識では Max Pooling、すなわちその枠内で一番大きいデータを出力とすることが多い
  - 他には Average Pooling (平均をとる）等がある

# プーリング層

☆ 2x2 のプーリングをストライド 2 で行なった場合

入力データ

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 4 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 |
| 3 | 2 | 1 | 2 |

出力データ

| | |
|---|---|
| 4 | 4 |
| 3 | 3 |

# プーリングの演習

☆ 以下（リンクあり）を、Colab にコピー＆ペーストして実行する。

```
import numpy as np
ind  = np.array([[1,2,3,4],
                 [4,3,2,1],
                 [0,1,2,3],
                 [3,2,1,2]])
outd = np.zeros((2,2))
for i in range(2):
  for j in range(2):
    outd[i][j] = np.max(ind[i*2:i*2+2,j*2:j*2+2])
print(outd)
```

# プーリング層の特徴

- 学習するデータが
- チャネル数は
- 微小な位置変化にたいして
  - データが多少ぶれてもプーリング処理で吸収できることがある（できない場合もある）
  - 例えば 2x2 のウィンドウの場合、１行１列目にあった最大値がずれて１行２列目に来ても、２行１列目に来ても、出力は同じ、ということ

# プーリング層の特徴

- 学習するデータが
  **ない（パラメータがない）**
- チャネル数は
- 微小な位置変化にたいして
  - データが多少ぶれてもプーリング処理で吸収できることがある（できない場合もある）
  - 例えば 2x2 のウィンドウの場合、１行１列目にあった最大値がずれて１行２列目に来ても、２行１列目に来ても、出力は同じ、ということ

# プーリング層の特徴

- 学習するデータが
  ない（パラメータがない）
- チャネル数は 変化しない
- 微小な位置変化にたいして
  - データが多少ぶれてもプーリング処理で吸収できることがある（できない場合もある）
  - 例えば 2x2 のウィンドウの場合、１行１列目にあった最大値がずれて１行２列目に来ても、２行１列目に来ても、出力は同じ、ということ

# プーリング層の特徴

- 学習するデータが
  **ない（パラメータがない）**
- チャネル数は **変化しない**
- 微小な位置変化にたいして **頑強 (robust)**
  - データが多少ぶれてもプーリング処理で吸収できることがある（できない場合もある）
  - 例えば 2x2 のウィンドウの場合、１行１列目にあった最大値がずれて１行２列目に来ても、２行１列目に来ても、出力は同じ、ということ

# 畳み込みニューラルネットの実装

- まず、ユーティリティ (im2col, etc)
- 畳み込み層
- プーリング層
- …以上を基に、畳み込みニューラルネットとその学習

# 4 次元配列への対応

- 先週での議論により、バッチ対応の畳み込み処理のデータは 4 次元配列となる
- これを For での多重ループで計算させるのは面倒であり、かつ重い処理となる
- そこで [ ] という関数を導入する
  - Caffe, Chainer で実装されているものを手本にしている

# 4 次元配列への対応

- 先週での議論により、バッチ対応の畳み込み処理のデータは 4 次元配列となる
- これを For での多重ループで計算させるのは面倒であり、かつ重い処理となる
- そこで im2col (image to column) という関数を導入する
  - Caffe, Chainer で実装されているものを手本にしている

# im2col

- フィルタにとって都合の良いように入力データを展開する
- 3 or 4 次元データを [　　] に展開する
- 畳み込み演算ではフィルタの適用範囲が重なる場合があるので、その時には展開後の要素は元のブロック数より多くなる→メモリを [　　] しがち

# im2col

- フィルタにとって都合の良いように入力データを展開する
- 3 or 4 次元データを ２次元 に展開する
- 畳み込み演算ではフィルタの適用範囲が重なる場合があるので、その時には展開後の要素は元のブロック数より多くなる→メモリを [　　] しがち

# im2col

- フィルタにとって都合の良いように入力データを展開する
- 3 or 4 次元データを **２次元** に展開する
- 畳み込み演算ではフィルタの適用範囲が重なる場合があるので、その時には展開後の要素は元のブロック数より多くなる→メモリを **消費** しがち

# im2col

- しかし行列計算ライブラリや GPGPU の恩恵を受けられるメリットのほうが一般的に大きい
- 展開後は、フィルタも１列に展開して積を計算するだけ
- →  とほぼ同じ処理

# im2col

- しかし行列計算ライブラリや GPGPU の恩恵を受けられるメリットのほうが一般的に大きい
- 展開後は、フィルタも１列に展開して積を計算するだけ
- → Affine レイヤとほぼ同じ処理

# im2col の実装

☆以下が実装コード（と処理例）なので、Colab にコピー＆ペーストして実行する。(長い行は適宜 \ を使って折り込んでいる)

```
import numpy as np
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    N, C, H, W = input_data.shape
    out_h = (H + 2*pad - filter_h)//stride + 1 #整数除算
    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
```

（…次頁に続く）

# im2col の実装

◎前頁からの続きとして、以下が処理例なので、前頁の続きとして Colab にコピー＆ペーストするか、im2col-ex.py のリンク先をコピーして実行する

（バッチ数 *10*, チャネル数３の *7x7* のデータを乱数で作り、それを *5x5* のフィルタ用に *im2col* を適用している）

```
x2 = np.random.rand(10,3,7,7)
col2 = im2col(x2,5,5,stride=1,pad=0)
print(col2.shape)
```

- col2 の結果は 90,75 になるはず
- 75 は 5x5 のフィルタが３チャネル分、ということ

# おしまい

**質問・コメント等あれば、何でもお気軽に**
**aipr@e-chan.jp に**
**メールください！**

【注意】申し訳ないですが *HInT* の「連絡」は溜めてしまうことが多いのでリプライが遅れがちです。出来るだけ普通のメールで送ってください。