深層実習
第3,4回

中田尚

再掲

成績評価

- 共通
  - 出席80%以上、課題認定80%以上
- 中間レポート60％、演習成果報告書40％
  - 必ずしも発表一発勝負ではない。準備段階や提出資料も含める。
  - レポート
    - 2週に1つ程度を予定（8点×8回程度①～⑧）
  - 任意課題
    - 5点×2回程度
  - 演習成果報告20点×2⑨⑩

再掲

レポートの書き方

- 卒業論文に向けて体裁を重視します
  - 内容6点＋体裁2点
- 原則としてWordかPDF
  - 文字サイズ等は常識の範囲で
  - TeX（で作成したPDF）でも良い
  - **それ以外のファイル（ソースコードなど）は見ないと思ってください**

体裁2点のためには、
章立て必須です
昨年の「しりとりレポート」
を参照

- 手書きの文章は不可
- スクリーンショットを挿入するのはOK
- 手書きの絵を挿入するのはOK

再掲

☆課題1

- 2つの正方行列の積を求めるpythonプログラムを作成して、行列サイズと実行時間の関係を調べる
  - 計算ライブラリを使用してはいけません
    - time や matplotlib 以外の import は禁止
    - ライブラリを使った実行は課題2で予定しています
- レポートにはグラフを含めること
  - 単に載せるのではなくグラフから読み取れる内容を説明すること
  - 実行時間の上限は「これ以上増やしても読み取れる内容が増えない」と思えるところまで
    - とはいえ1分以上の実行は不要です
- 締切：4/17 23:59

再掲

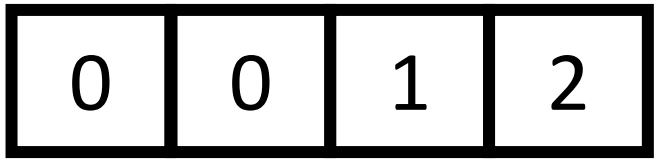
行列積の定義

- $$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} =$$

$$\begin{bmatrix} 1 \times 7 + 2 \times 9 + 3 \times 11 & 1 \times 8 + 2 \times 10 + 3 \times 12 \\ 4 \times 7 + 5 \times 9 + 6 \times 11 & 4 \times 8 + 5 \times 10 + 6 \times 12 \end{bmatrix}$$

pythonはなぜ遅いのか

- Pythonのリストの構造
- id()を使うと、アドレスのようなものを取得できる
  - 見た目もidも同じであれば実体も同じ
- 参考：C言語の配列はこんな感じ



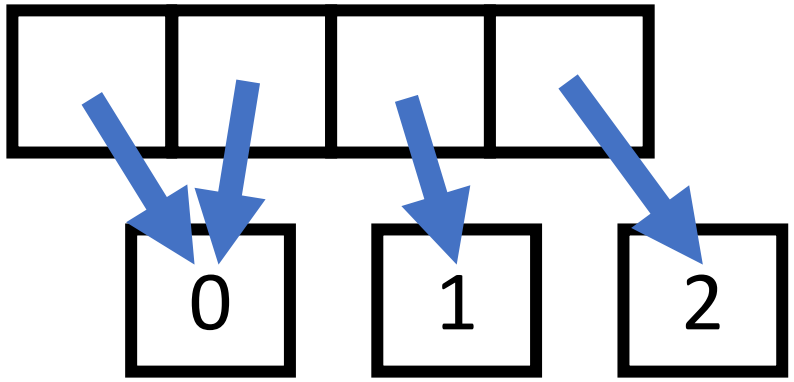
- Python
  - data=[0,0,1,2]
  - print(id(data),id(data[0]),id(data[1]),id(data[2]),id(data[3]))

  ⇒631744 343568 343568 358896 343632　（上位桁は同じなので省略）

ここのIDが同じ（0の部分）

pythonはなぜ遅いのか

- Pythonのデータ構造　（data=[0,0,1,2]）



同一になるのは256以下に限るらしい

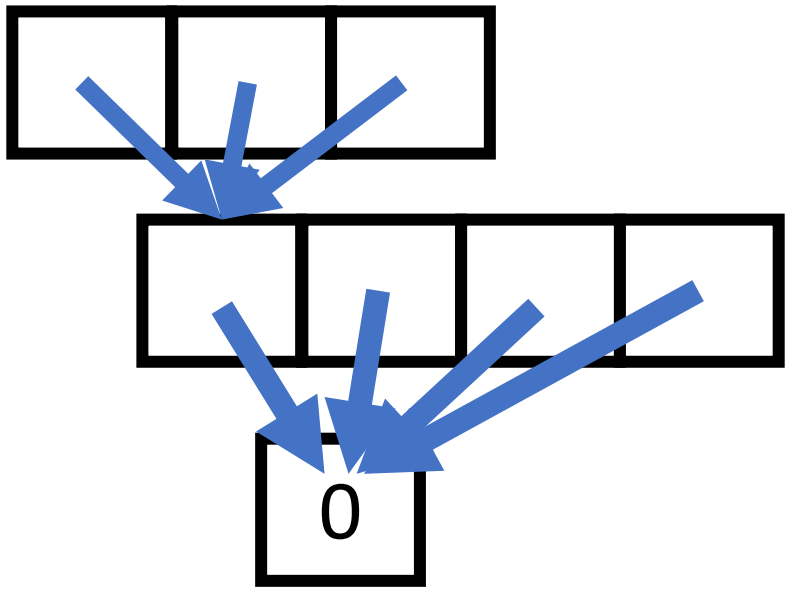
数字一つ一つが独立したオブジェクト
しかも同じ数字の部分は同一のオブジェクトへの参照
（つまり、数字を書き換える度にすでに出現したかどうかのチェックが必要）

- data=[0,0,1,2]
- print(id(data),id(data[0]),id(data[1]),id(data[2]),id(data[3]))

⇒631744 343568 343568 358896 343632　（上位桁は同じなので省略）

pythonはなぜ遅いのか

- [[0]*4] *3 のデータ構造



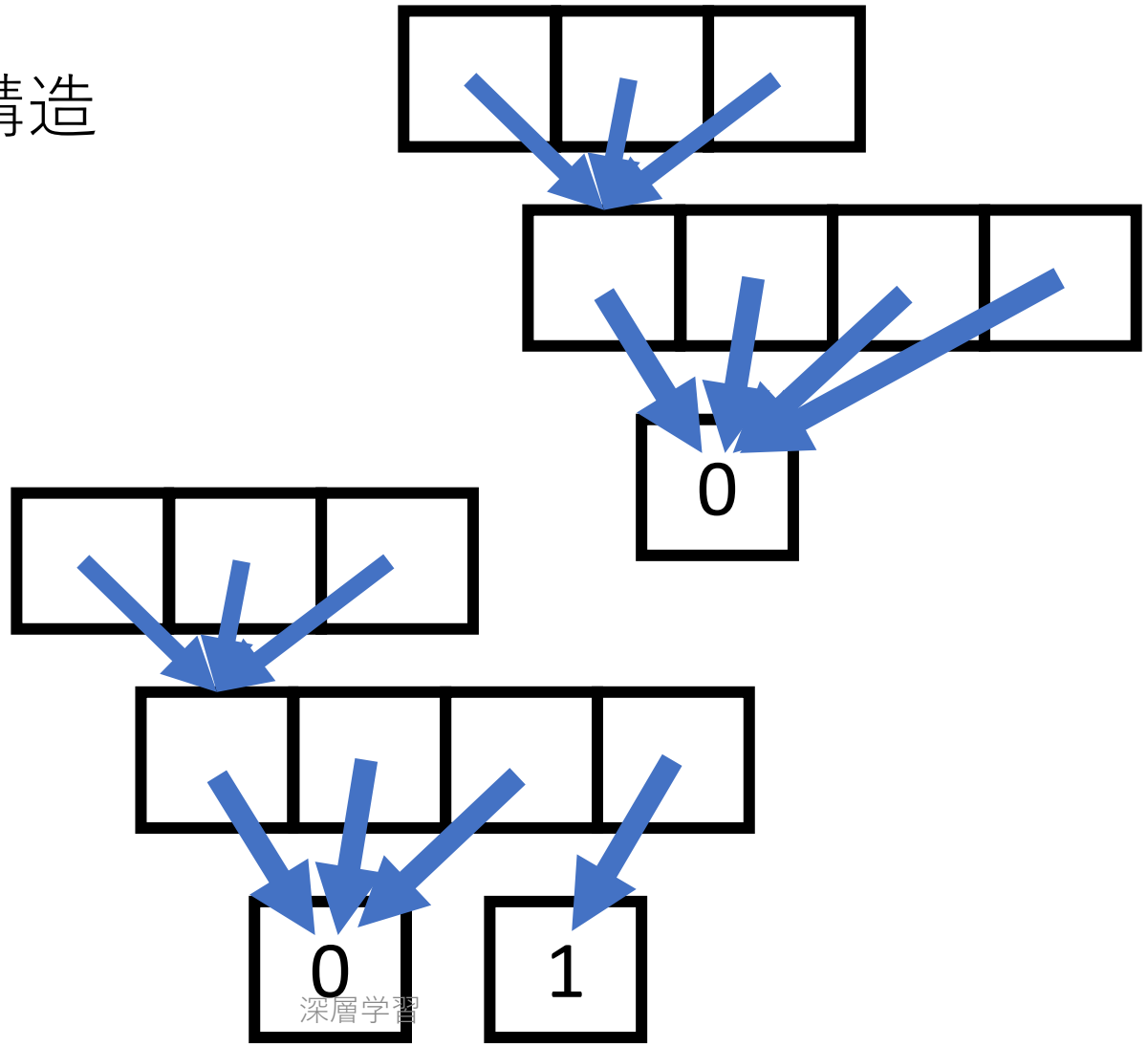
  - data= [[0]*4] *3
  - print(id(data),id(data[0]),id(data[1]),id(data[2]))

⇒631744 343632 343632 343632

pythonはなぜ遅いのか

- [[0]*4] *3 のデータ構造

- data[2][3]=1を実行
  - [[0,0,0,1],
    [0,0,0,1],
    [0,0,0,1]]
    になる

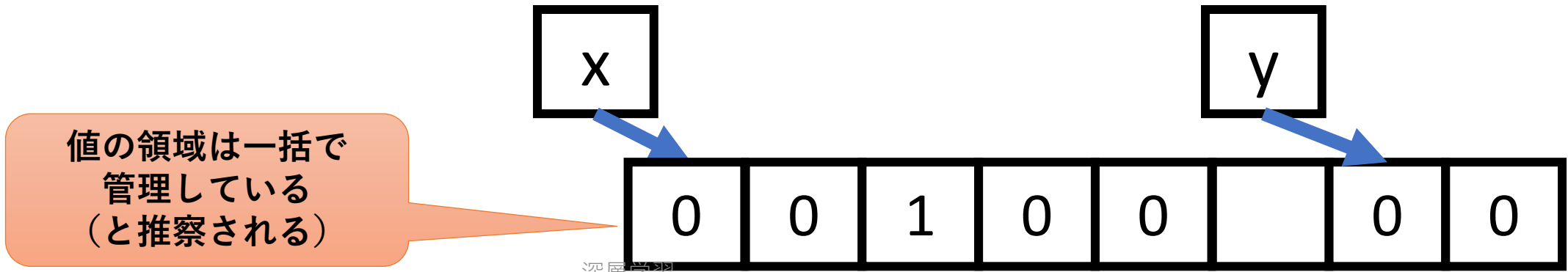


numpy

- 数値計算ライブラリ
  - GPUは使わない
  - 配列部分はc言語と同じ

- x=[0. 0. 1. 0. 0.], y=[0. 0. 1. 0. 0.]
  - print(id(x),id(x[0]),id(x[1]),id(x[2]),id(y[2]))

  ⇒611152 961072 961072 961072 961072



cupy

- numpyのGPU版
  - ほぼほぼ同じ（だけど一部違う）

import numpy as np

import cupy as cp

とすれば、npをcpにするだけで動くはず

numpyとcupyとpytorchの準備

```
import numpy as np
import cupy as cp
import torch
import time
```

numpy の行列積

```
N=10000
print("N=",N)
print("initializing")
npx=np.random.rand(N, N)
npy=np.random.rand(N, N)
```

N×N行列を
乱数で初期化

```
print("begin")
t1=time.time()
npz=np.matmul(npx, npy)
t2=time.time()
print("numpy",t2-t1,"sec",npz[0][0])
```

numpyの行列積

[0][0]のみ表示

cupy の行列積

```
# npx,npyを使うのでnumpyの後ろに貼り付けること
print("N=",N)
print("initializing")
cpx = cp.array(npx)
cpy = cp.array(npy)
device = cp.cuda.Device()
props = cp.cuda.runtime.getDeviceProperties(device.id)
print("cupy",props["name"].decode())
t1=time.time()
cpz=cp.matmul(cpx, cpy)
cp.cuda.Stream.null.synchronize()
t2=time.time()
print("cupy",t2-t1,"sec",cpz[0][0])
```

N×N行列を numpyからコピー

GPUが使われているか 確認のため表示

GPUの計算が終わるのを待つ 待たないと一瞬で終わるように見える

[0][0]のみ表示 （numpyと同じか確認すること）

```
# npx,npyを使うのでnumpyの後ろに貼り付けること
print("N=",N)
print("initializing")
tox = torch.from_numpy(npx).to("cuda")
toy = torch.from_numpy(npy).to("cuda")


print("torch",torch.cuda.get_device_name())
t1=time.time()
toz=torch.matmul(tox, toy)
torch.cuda.synchronize()
t2=time.time()
print("torch",t2-t1,"sec",toz[0][0])
```

N×N行列を
numpyからコピー

GPUが使われているか
確認のため表示

GPUの計算が終わるのを待つ
待たないと一瞬で終わるように見える

全く同じとは
限らない
計算誤差がある

[0][0]のみ表示
（numpyと同じか確認すること）

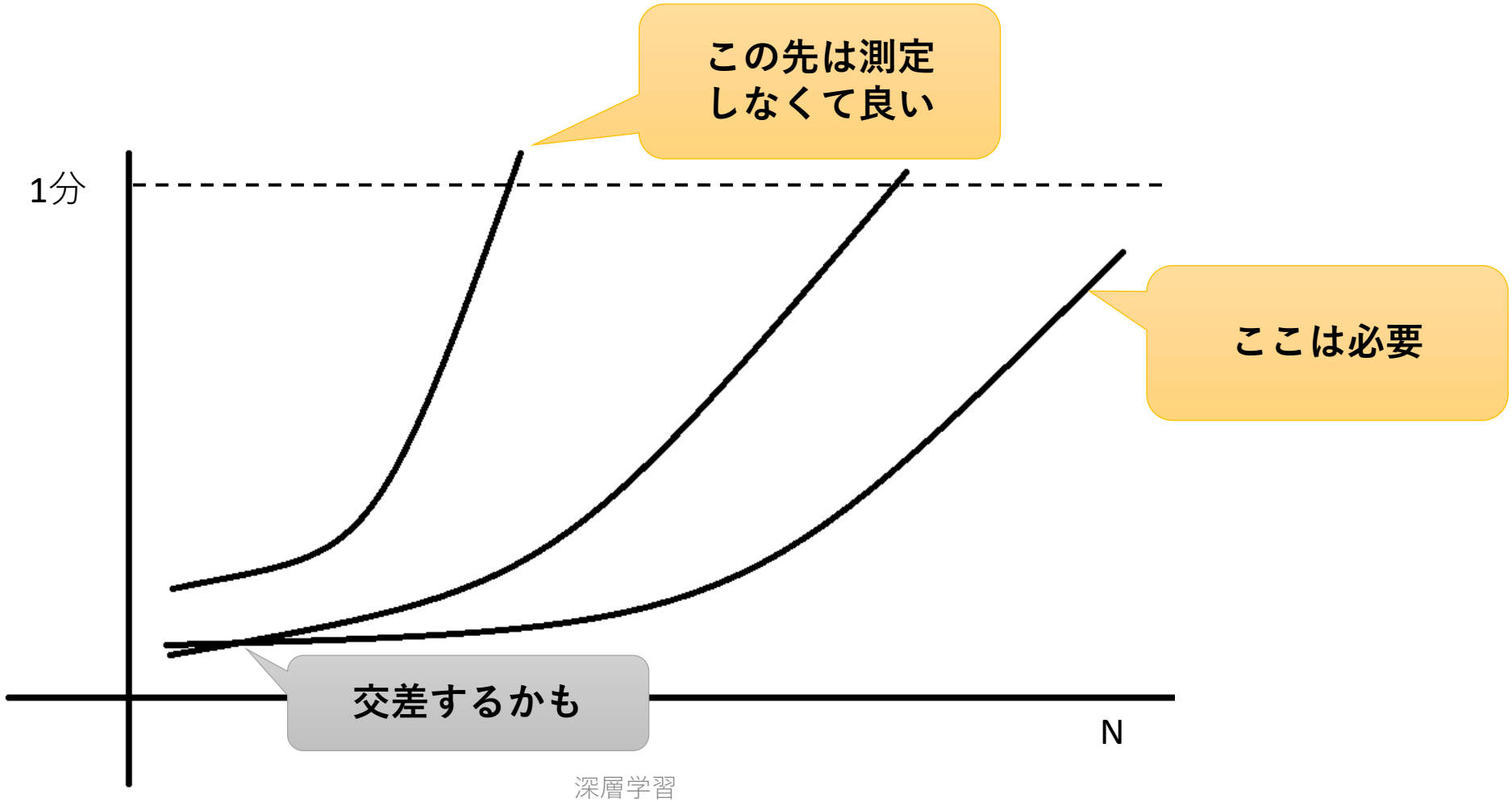
実行例

```
N= 10000
initializing
begin
numpy 2.1272497177124023 sec 2460.450931332193
N= 10000
initializing
cupy A100-SXM4-40GB
cupy 0.7341670989990234 sec 2460.4509313322037
```

これと違う結果になったら理由を考察すると良い（必須では無い）

実行時間の測定

再掲



畳み込みの計算（conv：Convolution）

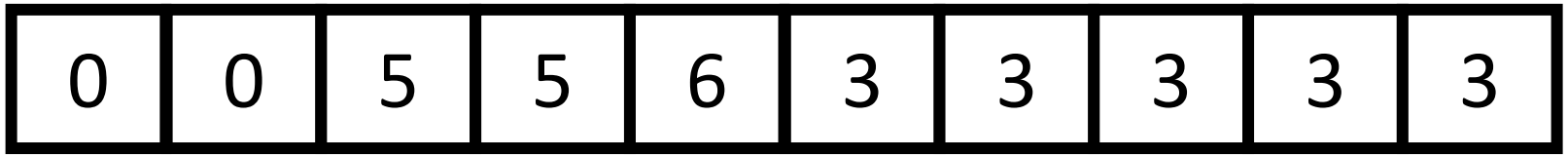
- python（の基本機能）
- numpy （+ scipyを使う）
- cupy
- pytorch

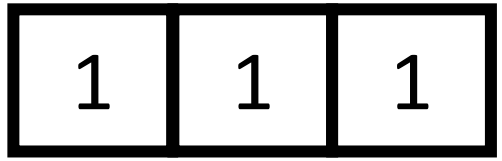
畳み込みとは

- *畳み込みとは、関数 g を平行移動しながら関数 f に重ね足し合わせる二項演算である。　Wikipedia*
  - 「重ね足し合わせる」がよくわからない
  - 「重ねた部分を要素毎に乗算し結果を足し合わせる」(離散値の場合)
    - 連続値の場合は積分として定義されるが省略

1次元の畳み込み

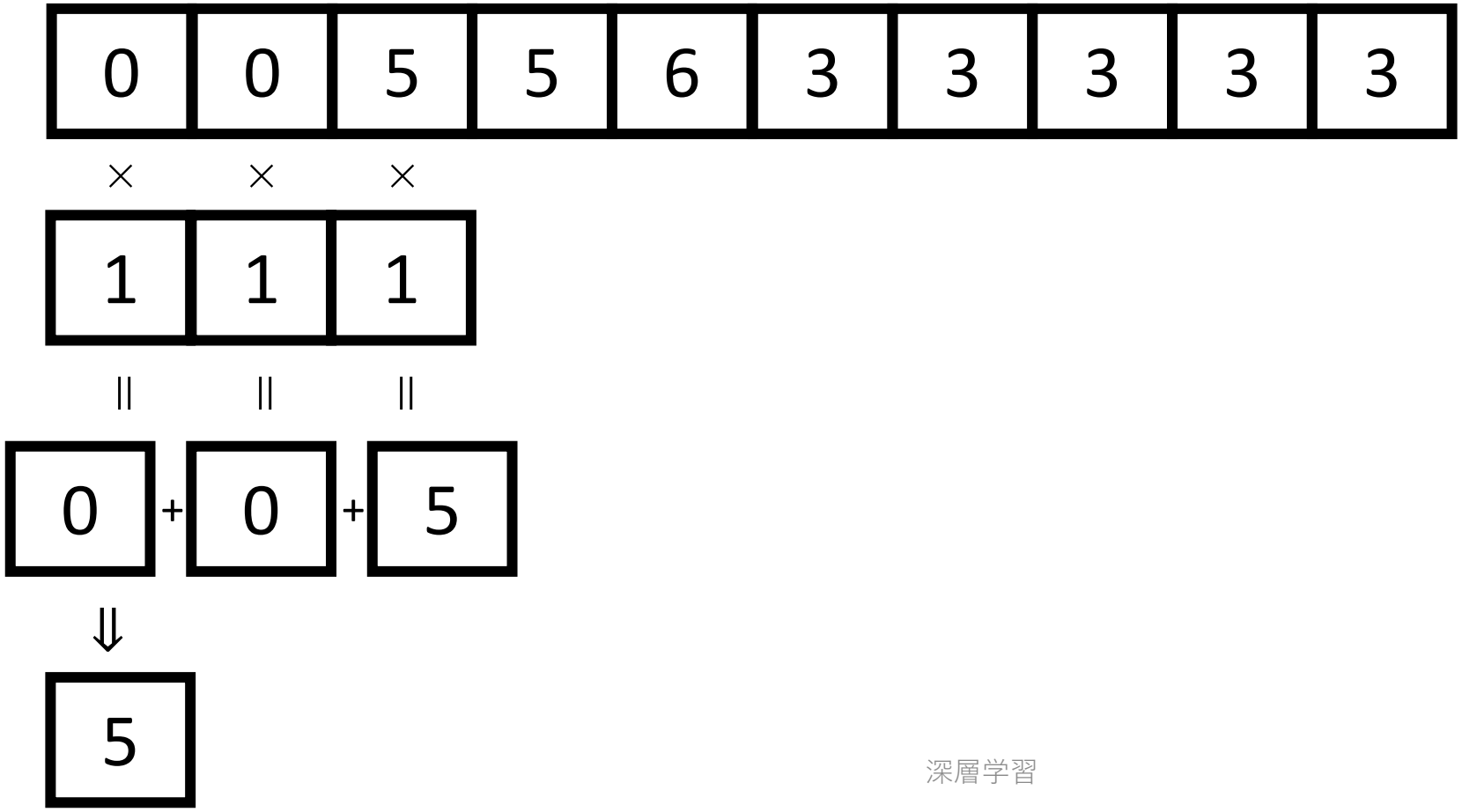
- 下記のようなデータと[1,1,1]の畳み込み





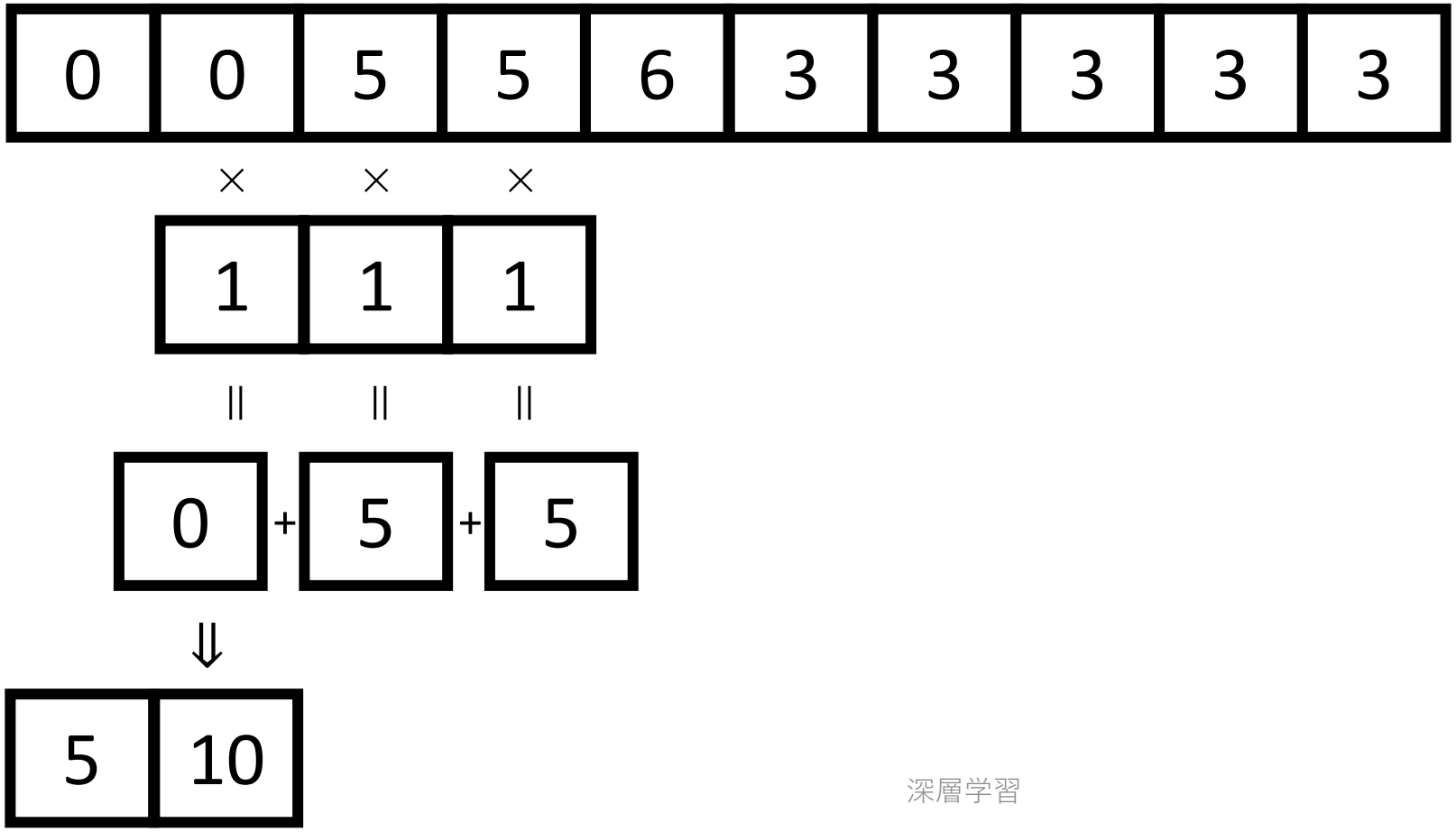
1次元の畳み込み

- 下記のようなデータと[1,1,1]の畳み込み



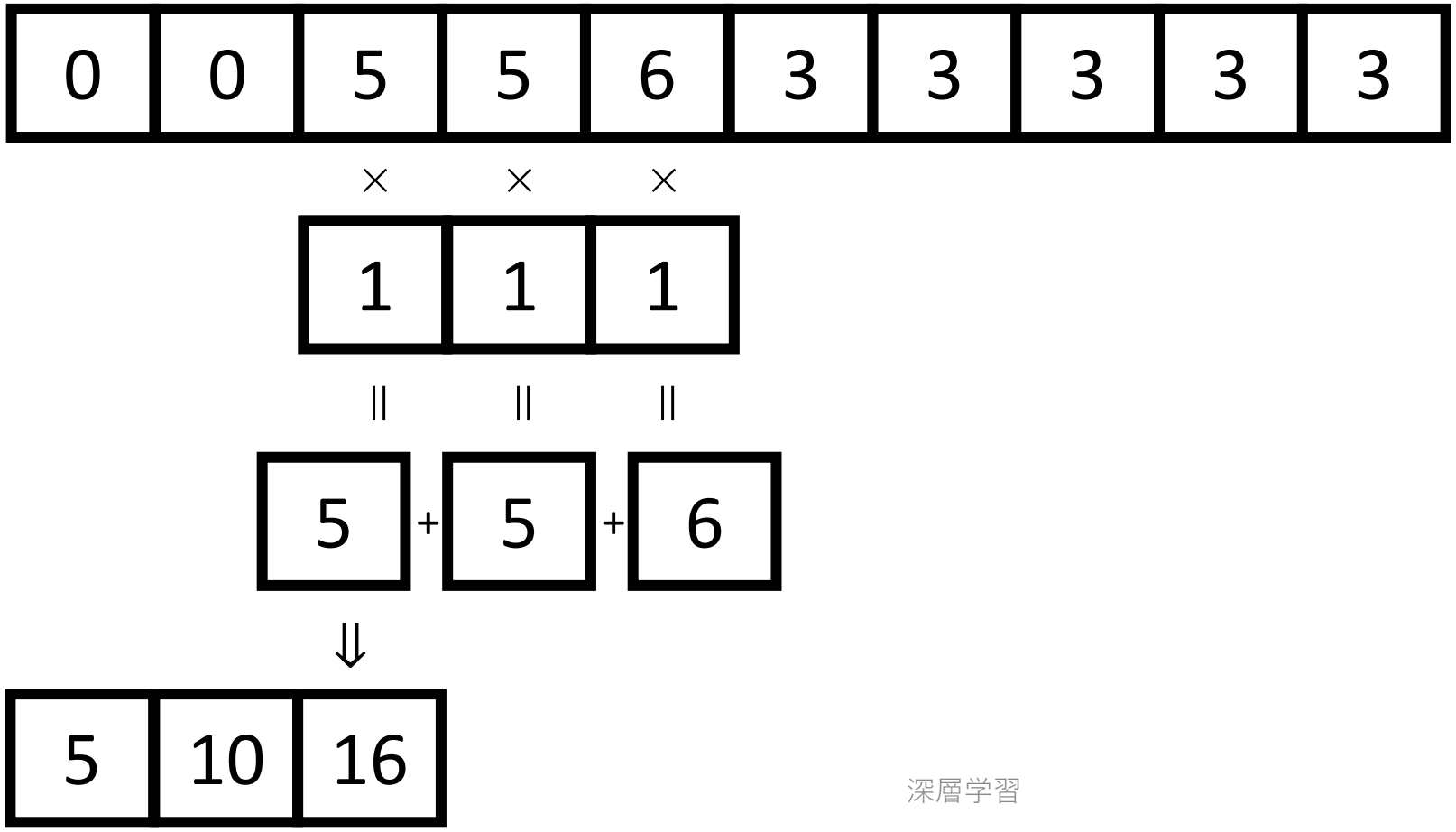
1次元の畳み込み

- 下記のようなデータと[1,1,1]の畳み込み



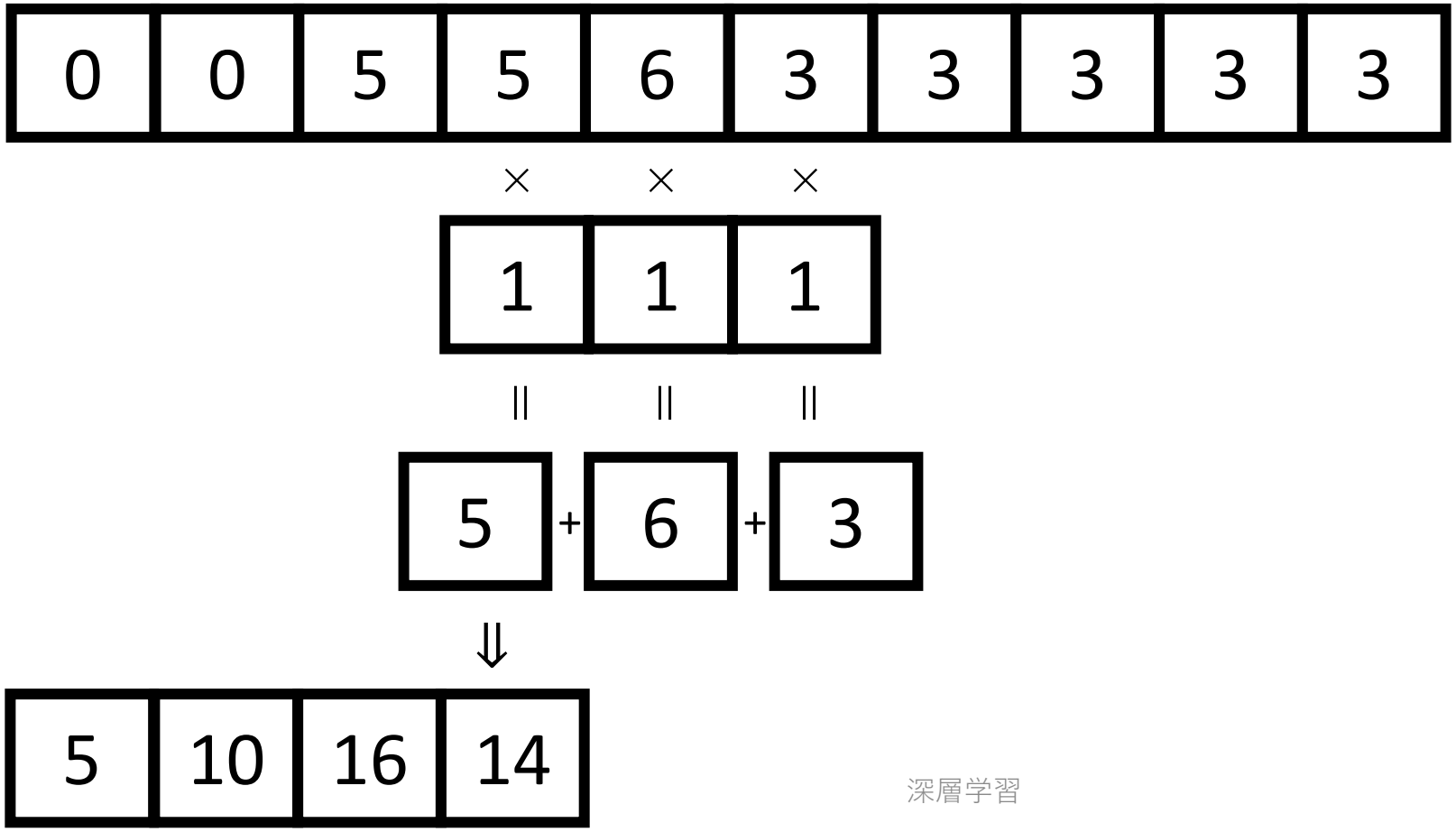
1次元の畳み込み

- 下記のようなデータと[1,1,1]の畳み込み



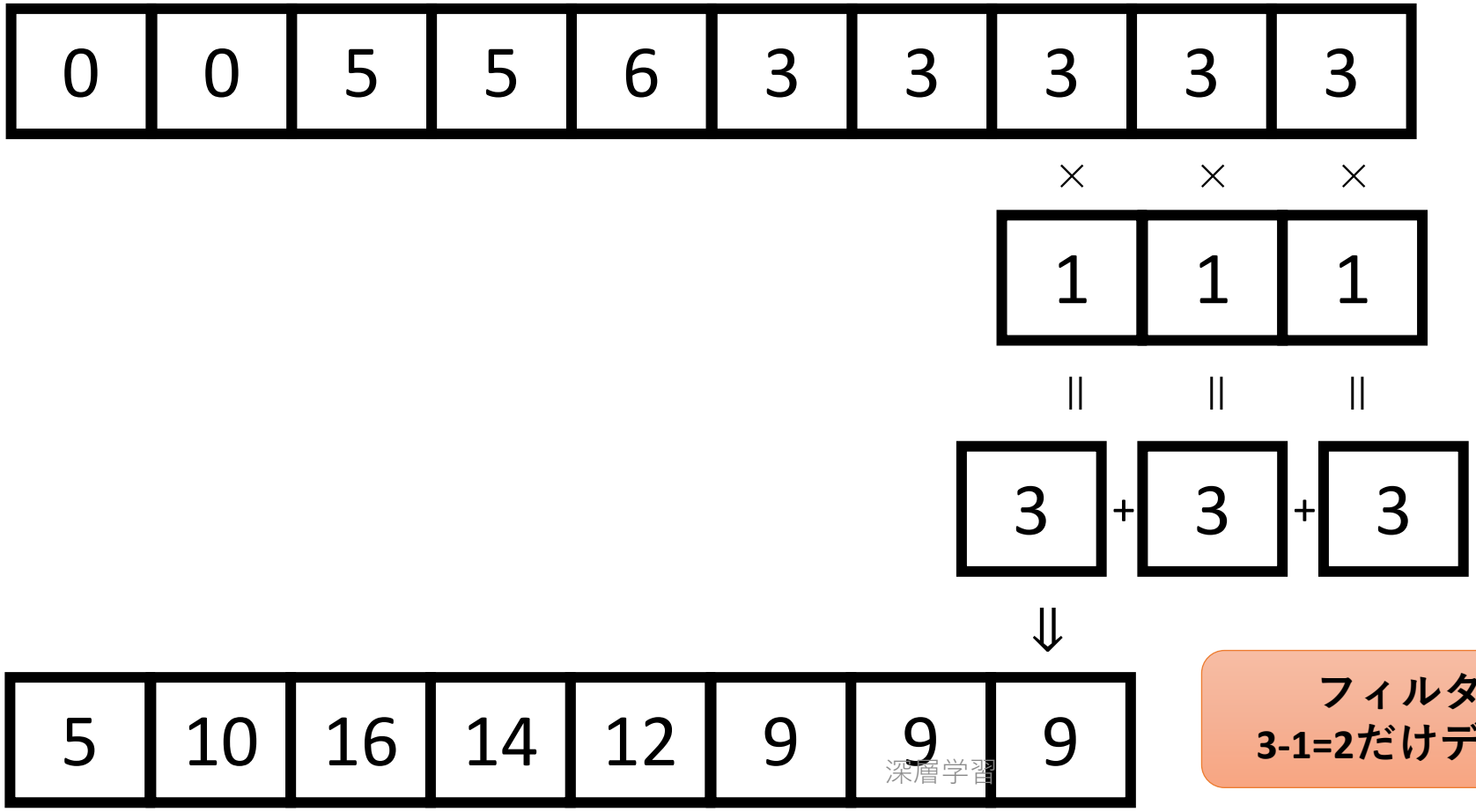
1次元の畳み込み

- 下記のようなデータと[1,1,1]の畳み込み



1次元の畳み込み

- 下記のようなデータと[1,1,1]の畳み込み

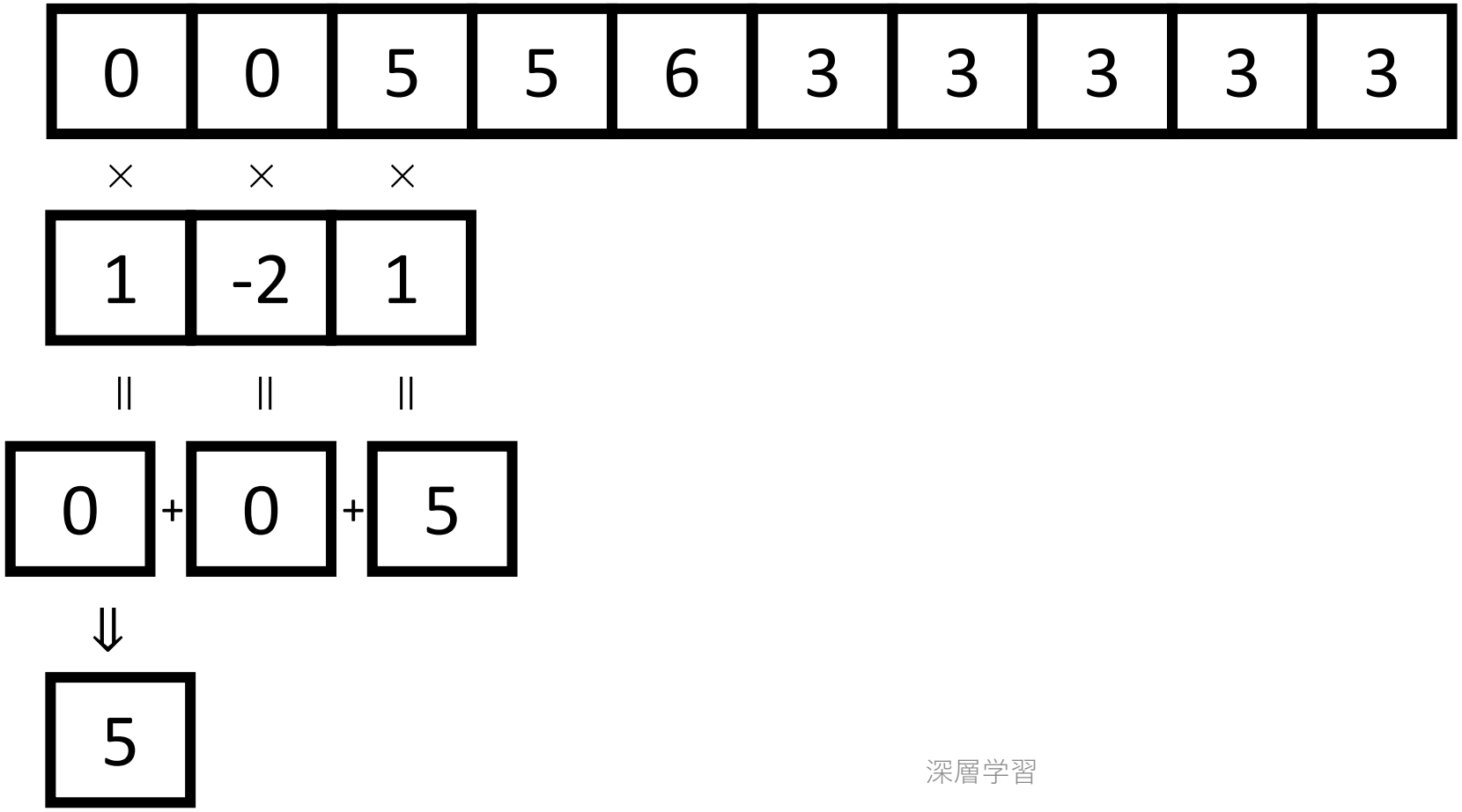


フィルタサイズが3の場合
3-1=2だけデータ数が少なくなる

1次元の畳み込み

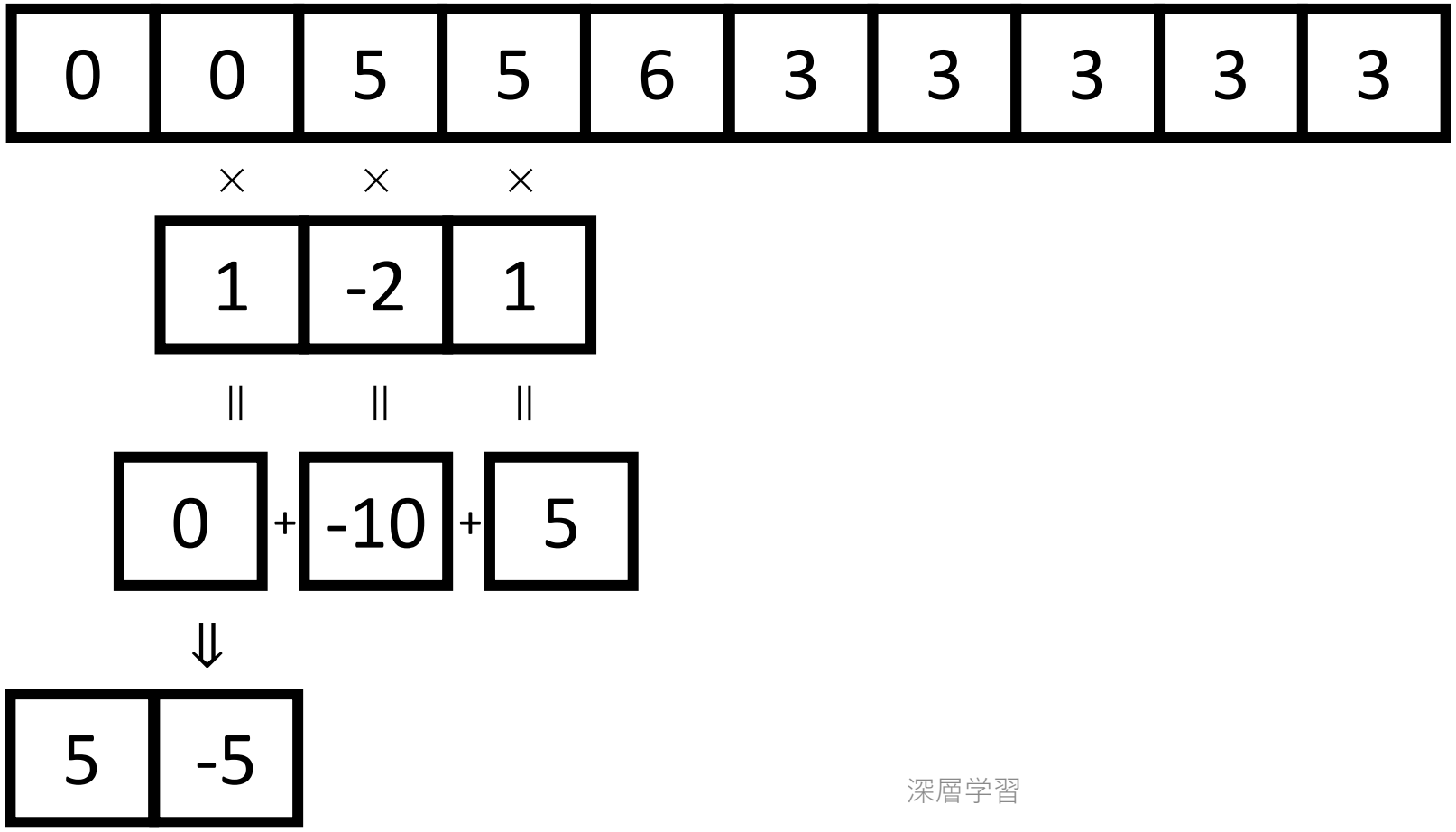
- 下記のようなデータと[1,-2,1]の畳み込み

(1次元)ラプラシアンフィルタ



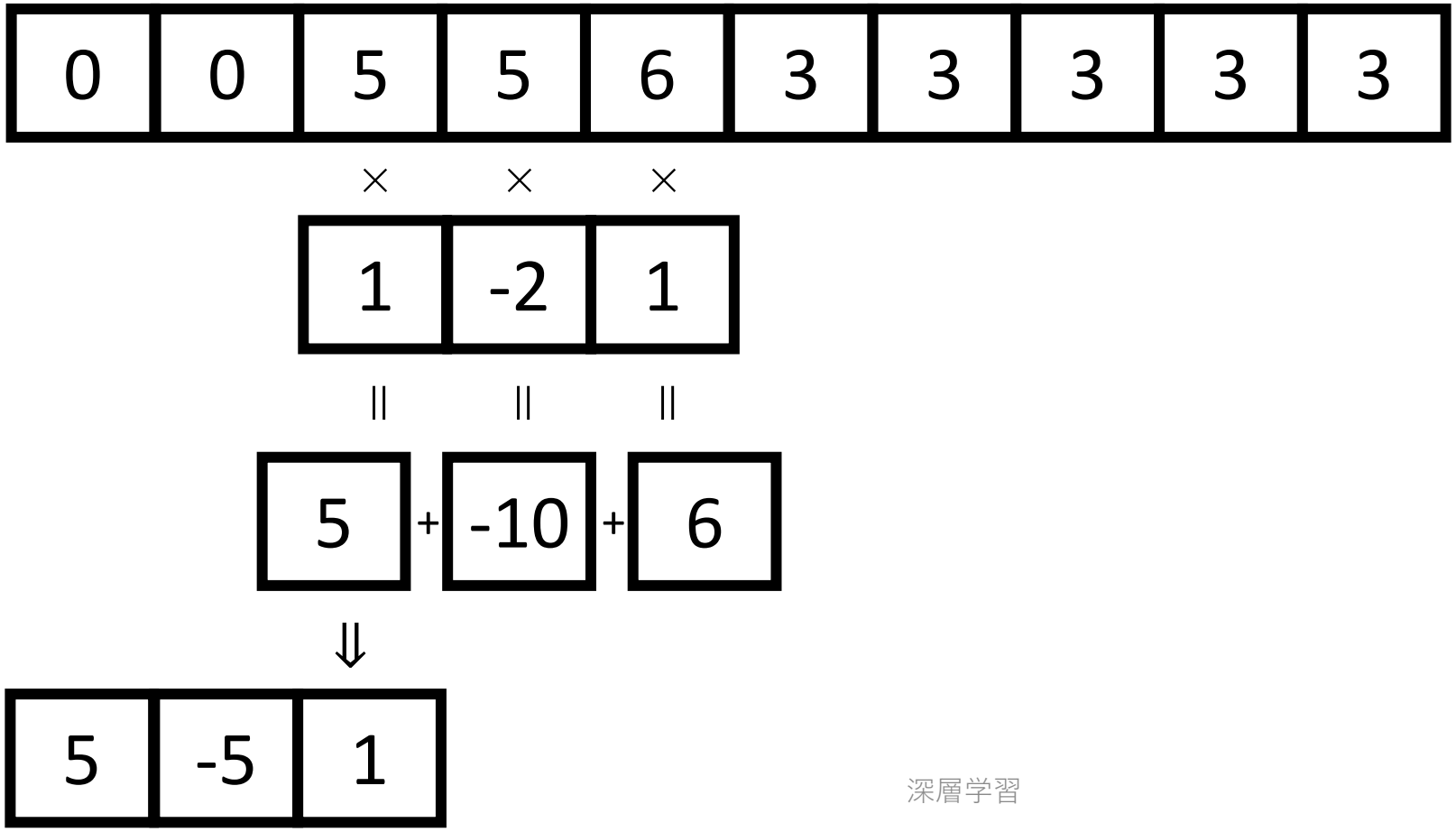
1次元の畳み込み

- 下記のようなデータと[1,-2,1]の畳み込み



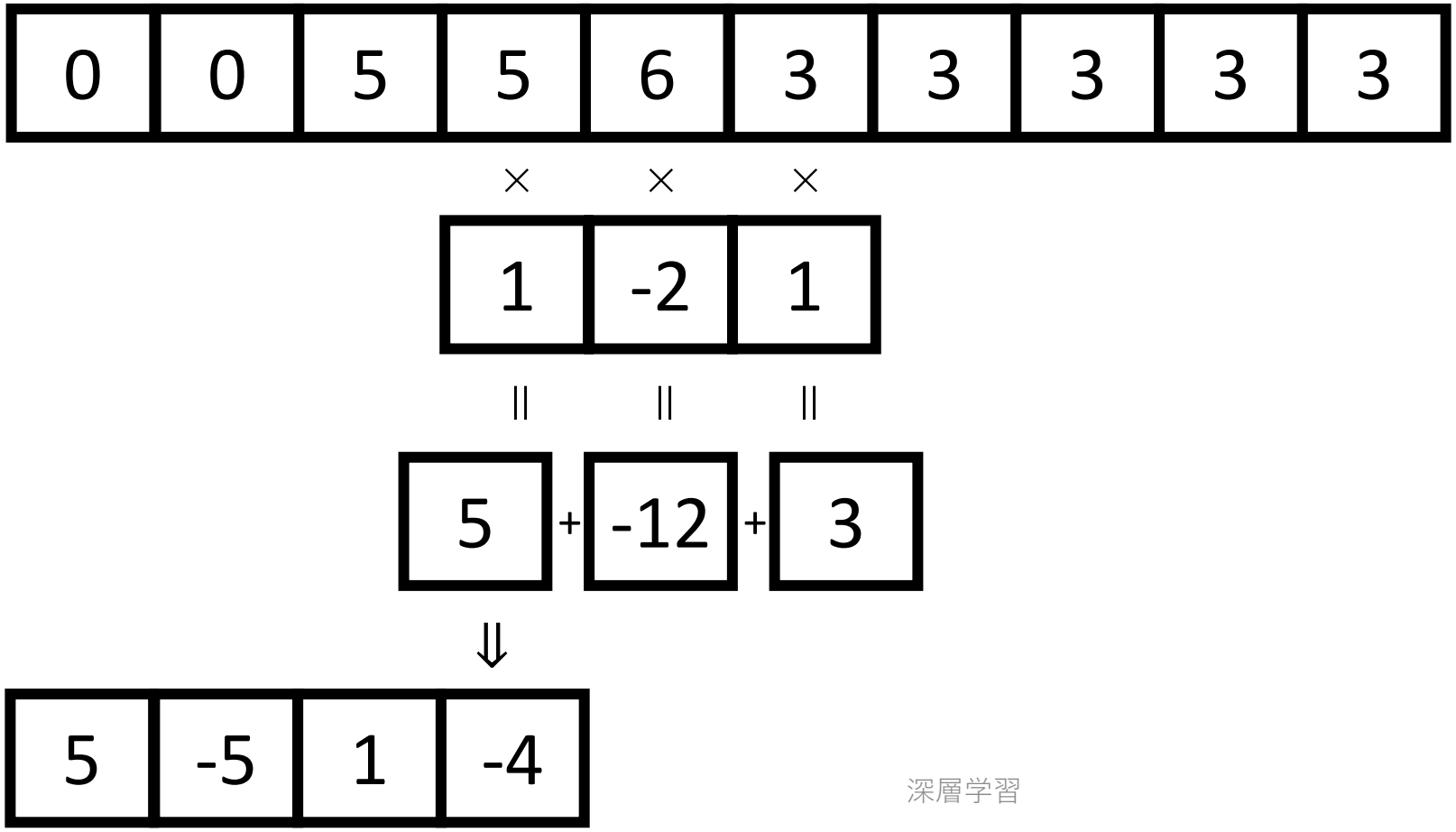
1次元の畳み込み

- 下記のようなデータと[1,-2,1]の畳み込み



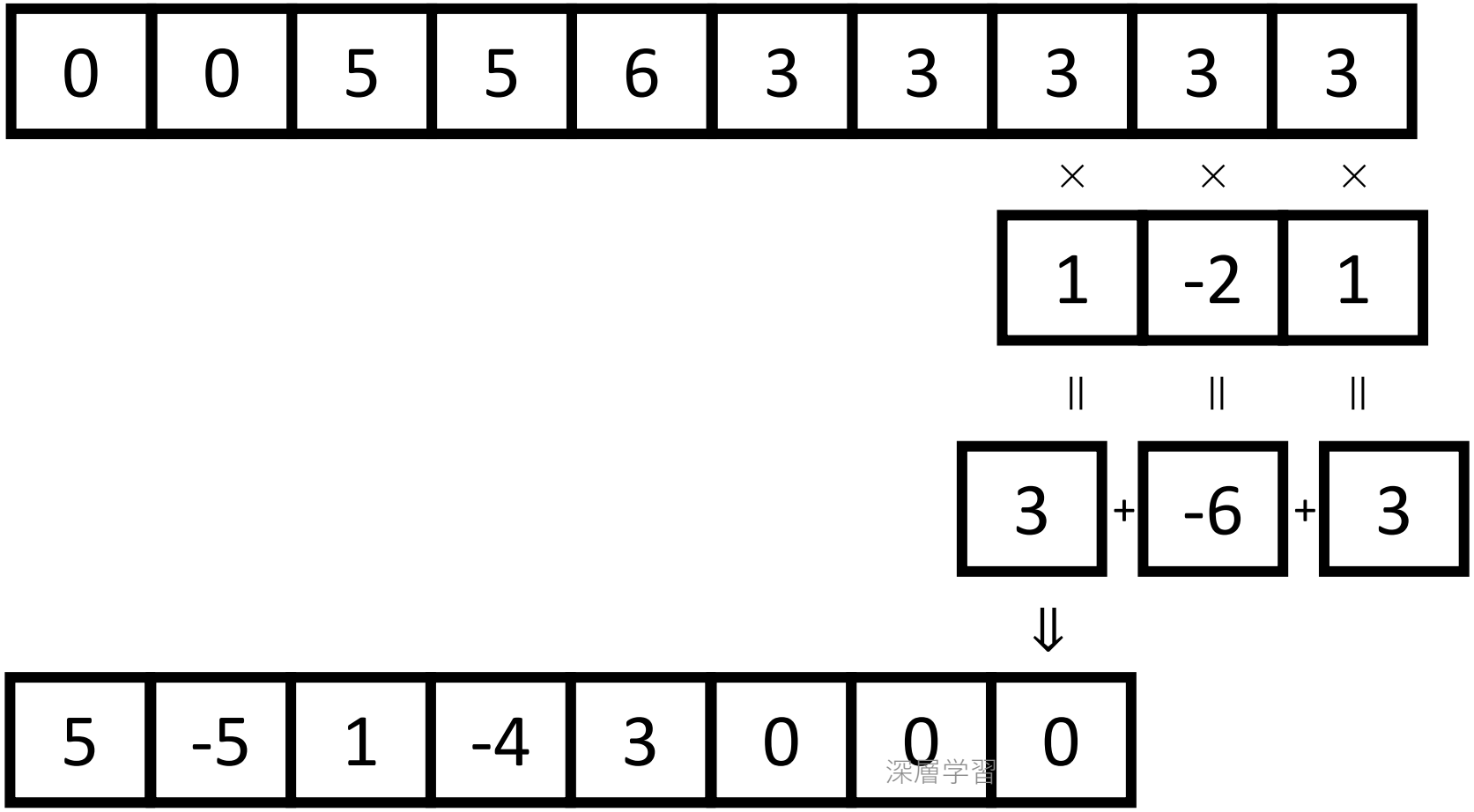
1次元の畳み込み

- 下記のようなデータと[1,-2,1]の畳み込み



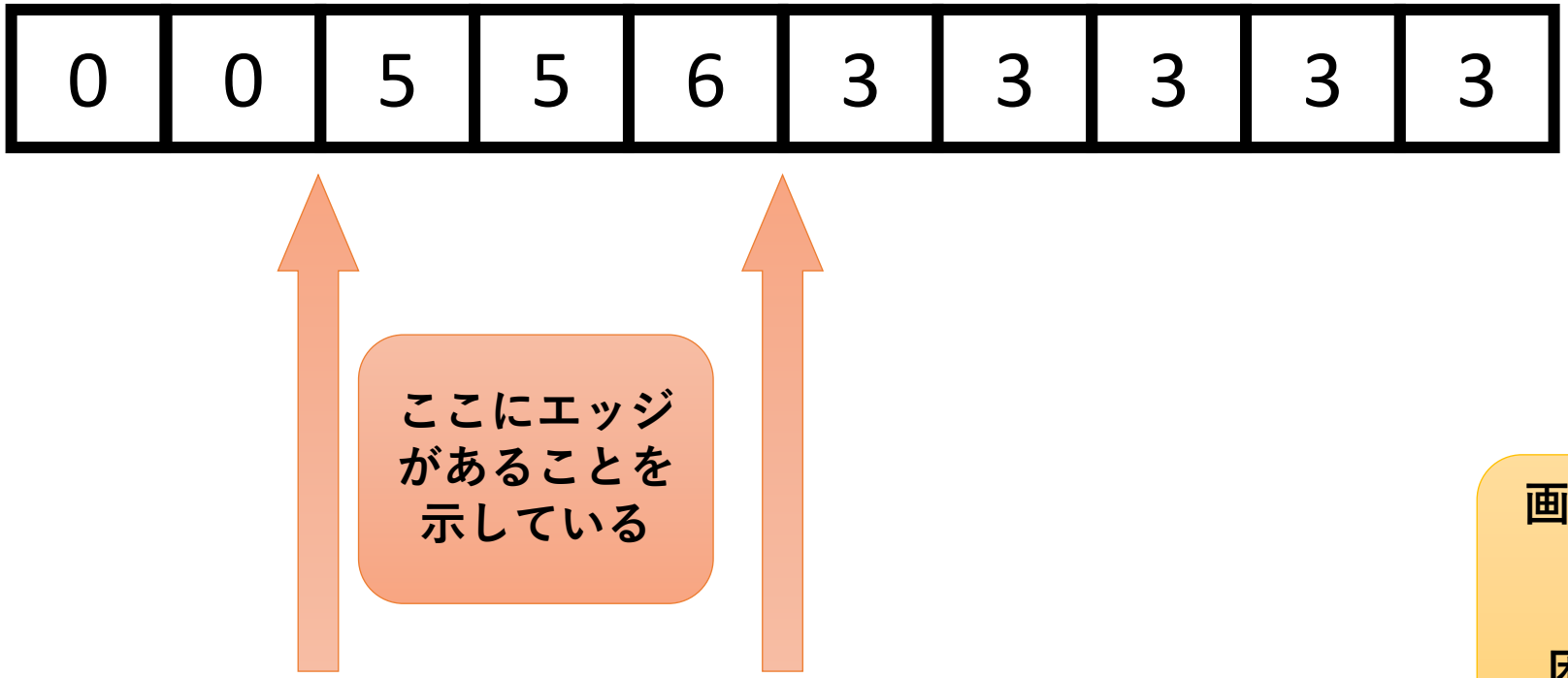
1次元の畳み込み

- 下記のようなデータと[1,-2,1]の畳み込み



1次元の畳み込み

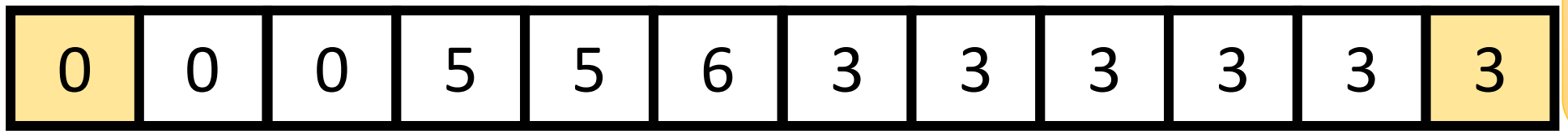
- 下記のようなデータと[1,-2,1]の畳み込み



画像処理の場合は
元画像より
小さくなると
困ることもある

1次元の畳み込み

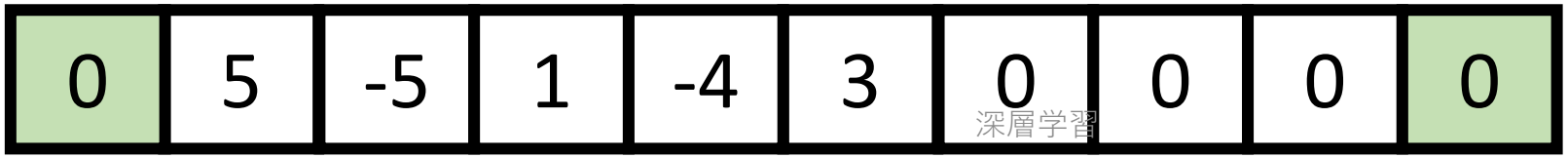
- 下記のようなデータと[1,-2,1]の畳み込み



元データを
増やしておく
(Padding)

pytorch の場合
padding='valid' で小さくなる
padding='same' で同じ大きさ

画像処理の場合は
元画像より
小さくなると
困ることもある



1次元の畳み込み

- 下記のようなデータと[1,-2,1]の畳み込み

0	0	0	5	5	6	3	3	3	3	3	3

**元データを
増やしておく
(Padding)**

pytorch の場合
padding='valid' で小さくなる
padding='same' で同じ大きさ
追加部分（黄色）の値は
padding_mode='zeros' は0で埋める
'reflect' 鏡のように反転
'replicate' 端の値をコピー
'circular' 反対の端の値をコピー

**画像処理の場合は
元画像より
小さくなると
困ることもある**

0	5	-5	1	-4	3	0	0	0	0

2次元の畳み込み

(2次元)ラプラシアンフィルタ

- 下記のようなデータと[[0,1,0],[1,-4,1],[0,1,0]]の畳み込み

0	0	0	0	0	3	3	3	0	0
0	0	5	5	6	3	3	3	3	0
0	0	5	5	6	3	3	3	3	3
0	0	5	5	6	3	3	3	3	3

× × ×
× × ×
× × ×

0	1	0
1	-4	1
0	1	0

= = = 0 0 0
= = = 0 0 5
= = = 0 0 0

⇒ 5

ラプラシアンフィルタには

1	1	1
1	-8	1
1	1	1

というパターンもある

一般的に総和が0になるようにする
そうでないと「平均値（＝明るさ）」が変わる

2次元の畳み込み

(2次元)ラプラシアンフィルタ

- 下記のようなデータと[[0,1,0],[1,-4,1],[0,1,0]]の畳み込み

0	0	0	0	0	3	3	3	0	0
0	0	5	5	6	3	3	3	3	0
0	0	5	5	6	3	3	3	3	3
0	0	5	5	6	3	3	3	3	3

× × ×
× × ×
× × ×

0	1	0
1	-4	1
0	1	0

= = =

0	0	0
0	-20	5
0	5	0

⇒

5	-10

2次元の畳み込み

(2次元)ラプラシアンフィルタ

- 下記のようなデータと[[0,1,0],[1,-4,1],[0,1,0]]の畳み込み

0	0	0	0	0	3	3	3	0	0
0	0	5	5	6	3	3	3	3	0
0	0	5	5	6	3	3	3	3	3
0	0	5	5	6	3	3	3	3	3

× × ×
× × ×
× × ×

0	1	0	= = =	0	0	0	⇒	5	-10	-4
1	-4	1	= = =	5	-20	6				
0	1	0	= = =	0	5	0				

2次元の畳み込み

(2次元)ラプラシアンフィルタ

- 下記のようなデータと[[0,1,0],[1,-4,1],[0,1,0]]の畳み込み

0	0	0	0	0	3	3	3	0	0
0	0	5	5	6	3	3	3	3	0
0	0	5	5	6	3	3	3	3	3
0	0	5	5	6	3	3	3	3	3

× × ×
× × ×
× × ×

0	1	0
1	-4	1
0	1	0

= = =

0	0	0
5	-24	3
0	6	0

⇒

5	-10	-4	-10

2次元の畳み込み

(2次元)ラプラシアンフィルタ

- 下記のようなデータと[[0,1,0],[1,-4,1],[0,1,0]]の畳み込み

0	0	0	0	0	3	3	3	0	0
0	0	5	5	6	3	3	3	3	0
0	0	5	5	6	3	3	3	3	3
0	0	5	5	6	3	3	3	3	3

× × ×
× × ×
× × ×

0	1	0
1	-4	1
0	1	0

= = =

0	3	0
6	-12	3
0	3	0

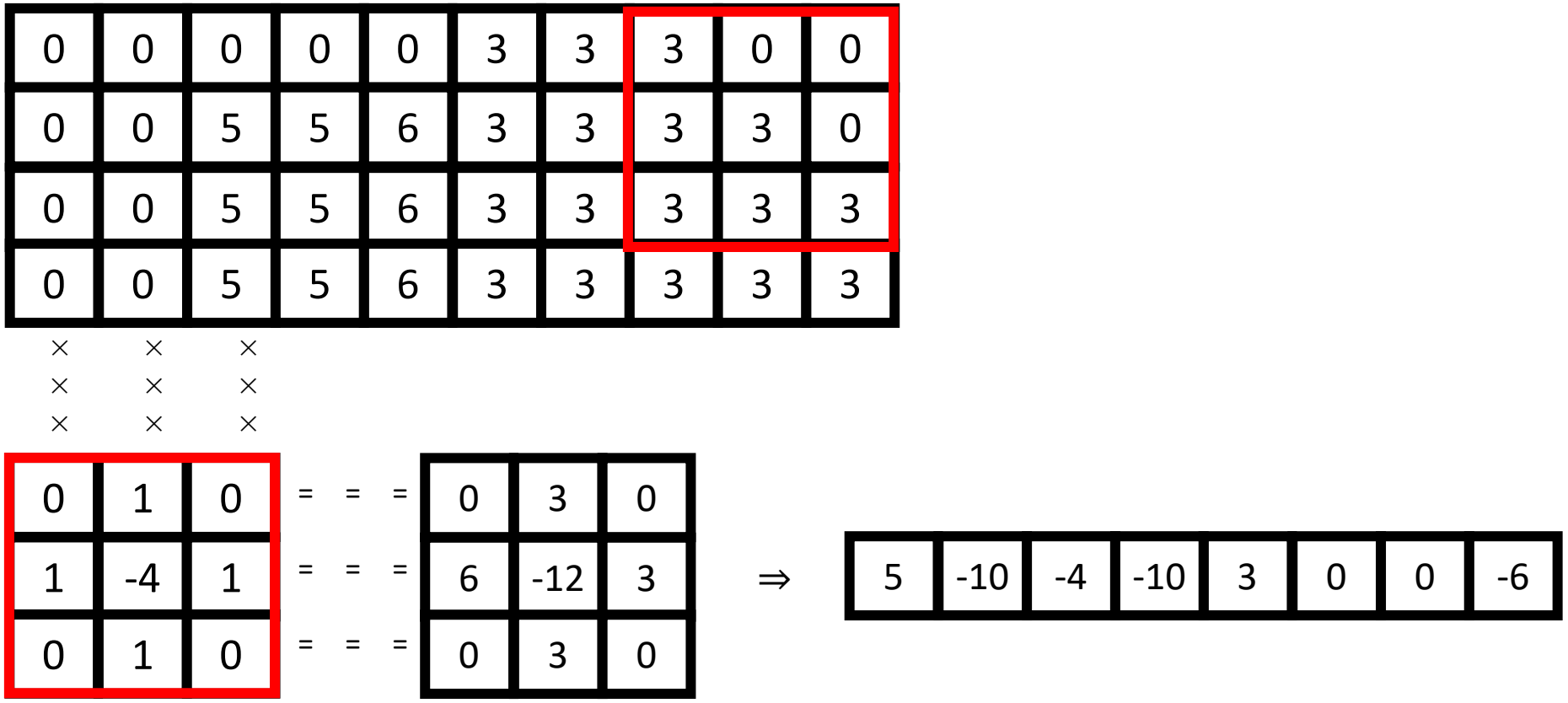
⇒

5	-10	-4	-10	3

2次元の畳み込み

(2次元)ラプラシアンフィルタ

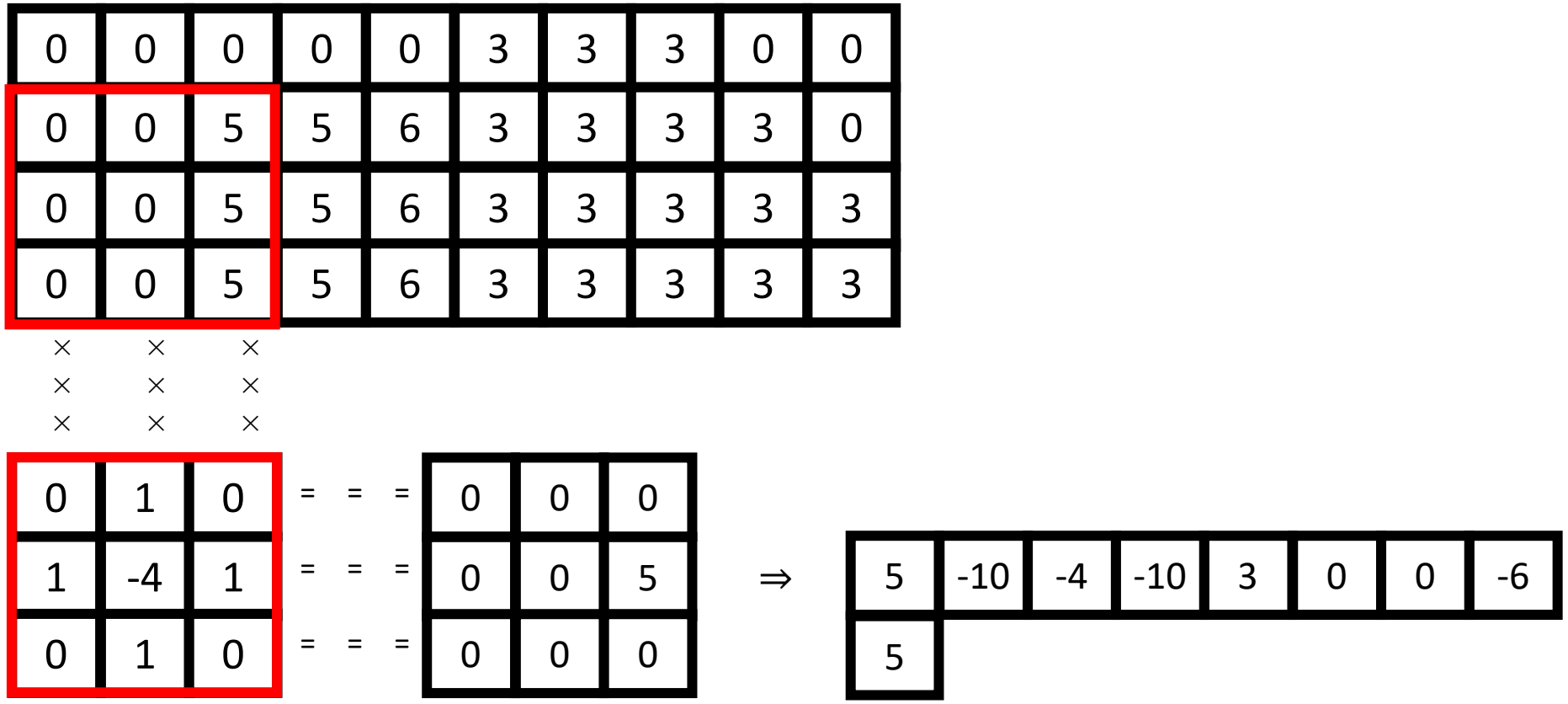
- 下記のようなデータと[[0,1,0],[1,-4,1],[0,1,0]]の畳み込み



2次元の畳み込み

(2次元)ラプラシアンフィルタ

- 下記のようなデータと[[0,1,0],[1,-4,1],[0,1,0]]の畳み込み



2次元の畳み込み

(2次元)ラプラシアンフィルタ

- 下記のようなデータと[[0,1,0],[1,-4,1],[0,1,0]]の畳み込み

0	0	0	0	0	3	3	3	0	0
0	0	5	5	6	3	3	3	3	0
0	0	5	5	6	3	3	3	3	3
0	0	5	5	6	3	3	3	3	3

× × ×
× × ×
× × ×

0	1	0
1	-4	1
0	1	0

= = =

0	0	0
0	0	0
0	0	0

⇒

5	-10	-4	-10	3	0	0	-6
5	-5	1	-4	3	0	0	0

2次元の畳み込み

(2次元)ラプラシアンフィルタ

- 下記のようなデータと[[0,1,0],[1,-4,1],[0,1,0]]の畳み込み

0	0	0	0	0	3	3	3	0	0
0	0	5	5	6	3	3	3	3	0
0	0	5	5	6	3	3	3	3	3
0	0	5	5	6	3	3	3	3	3

× × ×
× × ×
× × ×

0	1	0	= = =	0	0	0	⇒	5	-10	-4	-10	3	0	0	-6
1	-4	1	= = =	0	0	0		5	-5	1	-4	3	0	0	0
0	1	0	= = =	0	0	0									

2次元の畳み込み

(2次元)ラプラシアンフィルタ

,1],[0,1,0]]の畳み込み

0	0	0	0	0	0	3	3	3	0	0	0
0	0	0	0	0	0	3	3	3	0	0	0
0	0	0	5	5	6	3	3	3	3	0	0
0	0	0	5	5	6	3	3	3	3	3	3
0	0	0	5	5	6	3	3	3	3	3	3
0	0	0	5	5	6	3	3	3	3	3	3

元データを
増やしておく
(Padding)

画像処理の場合は
元画像より
小さくなると
困ることもある

× × ×

0	1	0
1	-4	1
0	1	0

= = =

0	0	0
0	0	0
0	0	0

⇒

0	0	0	5	9	-3	0	-3	6	0
0	5	-10	-4	-10	3	0	0	-6	0
0	5	-5	1	-4	3	0	0	0	-3
0	5	-5	1	-4	3	0	0	0	0

畳み込みのバリエーション

- padding
- stride
- dilation(省略)
- in_channels, out_channels (次回以降に説明予定)

- 昨年のスライド参照
  - 必要になってから調べれば十分

☆課題2

畳み込みは padding='valid' （小さくなる）で良い
理由を説明すれば 'same' （同じサイズ）でも良い

- **2つの正方行列の積**と**正方行列と3×3行列の畳み込み**それぞれを求めるpythonプログラムを作成して、それぞれの行列サイズと実行時間の関係を調べる
  - それぞれのpython, numpy, cupy, pytorchの実行時間について考察する
  - cupyの畳み込みは省略しても良い
- レポートにはグラフを含めること
  - 単に載せるのではなくグラフから読み取れる内容を説明すること
  - 実行時間の上限は「これ以上増やしても読み取れる内容が増えない」と思えるところまで
    - とはいえ1分以上の実行は不要です
- 締切： 4/24(木) 23:59

GPUのメモリサイズによっては
10秒程度でメモリ不足になるかも
共有メモリが使われると
一気に遅くなるので注意

それぞれについて「行列サイズと実行時間の間にはXXという関係がある」
という考察が必要

よくある計算間違い

行列積

- $\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}\begin{bmatrix}7 & 8 & 13\\ 9 & 10 & 14\\ 11 & 12 & 15\end{bmatrix}=\begin{bmatrix}1\times7+2\times9+3\times11 & .. & ..\\ & .. & \\ & .. & \end{bmatrix}$

畳み込み

- $\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} * \begin{bmatrix}7 & 8 & 13\\ 9 & 10 & 14\\ 11 & 12 & 15\end{bmatrix}=[1\times7+2\times8+3\times13+\cdots]$

課題2のヒント

- numpyはscipy.signal.correlate2dを使う
- scipy.signal.convolve2dはフィルタを反転させてから畳み込みを行うので、torchのconv2dと異なる結果になる

- **cupyx.scipy.signal.correlate2d が正しく動かない場合は省略して良い**

- pytorch はtorch.nn.functional.conv2d を使うが、入力は4次元である必要がある(次ページ参照)

課題2の追加考察（必須では無い）

tox = torch.from_numpy(npx).to("cuda")　　# numpyからコピー

toy = torch.from_numpy(npy).to("cuda")

の部分を以下のように変更する

tox = torch.rand(n,n).to("cuda")　　# pytorchで新規作成

toy = torch.rand(n,n).to("cuda")

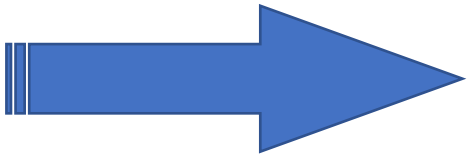
乱数の中身は変化するのは当然として、実行時間も大きく変わる可能性がある

（デスクトップPCと大学推奨ノートPCでは変わるはず）

実行時間が変わった原因を考察する

pytorchのconv2d

- 入力データは[バッチサイズ][入力チャンネル][x][y]
  フィルタは[出力チャンネル][入力チャンネル][x][y]
  のデータを要求する
- 今回はバッチサイズ数も入出力チャンネル数も1とする
  つまり、data[N][N]をdata[1][1][N][N] に変換する
  - data=torch.reshape(data,(1,1,N,N)) または
    data=data.unsqueeze(0).unsqueeze(0)
    で変換できる



[1,1,1],
[1,-8,1],
[1,1,1]
が
[[[1,1,1],
[1,-8,1],
[1,1,1]]]
となる

- バッチサイズとチャンネルについては次回以降に説明

サイズ変換（要素数は同じ）

```
>>> x=torch.zeros(3,3)                    # 3x3の配列を0で初期化
>>> x.shape                               # サイズを表示する
torch.Size([3, 3])                        # 3x3になっている
>>> x=torch.reshape(x,(1,1,3,3))          # reshape
>>> x.shape                               # サイズを表示
torch.Size([1, 1, 3, 3])                  # 1x1x3x3になっている
>>> x=x.unsqueeze(0)                      # さらに
>>> x.shape
torch.Size([1, 1, 1, 3, 3])               # 1x1x1x3x3
>>>
```

Pythonのタプルの話

- `x=torch.reshape(x, (1,1,3,3))`
  - `(1,1,3,3)`はタプル
  - `[1,1,3,3]`だとリスト

- タプルは変更（書き込み）できない
  - なので高速
  - 辞書型のキーに出来る

- `dict={}`
- `dict[[1,2]]=3` はエラーになる
- `dict[(1,2)]=5` はOK

任意課題1 満点5点

- 課題1の実装は非常に遅く、numpyを使えば100倍以上の高速化が実現できるはずである
- numpyはC言語とFortranで実装されているので、C言語で課題1相当のコードを実装したところ、それでもnumpyの方が10倍程度早い（実行環境によってこの比率は大きく変わる）

- **C言語で実装しただけでは不十分な理由を考えよ**
  - **仮説であれば検証も行うこと（検証方法も自分で考える）**
- **解消可能な理由であれば、自ら実装を行い実行速度がnumpyに近付くことを確かめよ**
  - numpyはGPUを使わないので、GPUは使用禁止
  - cupyとGPU実装で比較しても良い（さらに高難易度）
- 締切：4/24(木) 23:59

今日の実習

- （もし課題1がまだなら質問する）

- 課題2に取り組む
  - GPUを搭載したPCで実行する

- 任意課題1に取り組む

i, j, kループ

```
for i in range(n):
  for j in range(n):
    for k in range(n):
      z[i][j]+=x[i][k]*y[k][j]
```

- このプログラムのi,j,kの順序を変えると実行速度が大きく変わる

キャッシュの存在

- メモリは遅い（CPU,GPUと比較して）
- データはメモリに入っている
- 毎回メモリから読み込むのでは性能が出ない
- 同時にたくさん読み込もう
  - 1を聞いて10を知る（実際には16個くらい同時に読む）
- その16個に何が含まれるのかが問題
  - `z[0][0]`にアクセスすると`z[0][1],z[0][2],,, z[0][15]`が読み込まれる
  - つまり`z[0][1]`にアクセスすると高速

i, j, kループ

```
for i in range(n):
  for j in range(n):
    for k in range(n):
      z[i][j]+=x[i][k]*y[k][j]
      z[0][0]+=x[0][0]*y[0][0]
      z[0][0]+=x[0][1]*y[1][0]
```

最高速　　高速　　低速

- 低速にならない方法は？
  - iが低速x2、kが低速x1なので、jを変えるのが一番高速

i, k, jループ

```
for i in range(n):    # 低速x2
  for k in range(n):  # 低速x1
    for j in range(n):
      z[i][j]+=x[i][k]*y[k][j]
      z[0][0]+=x[0][0]*y[0][0]
      z[0][1]+=x[0][0]*y[0][1]
```

高速　　最高速　　高速

- 一般にはこれが一番早い