

## 畳み込みニューラルネット (2)

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

(2024.12.06)

## 復習: 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
    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)

## 復習: 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, チャンネル数 3 で 7x7 の乱数データを、  
5x5 のフィルタ用に *im2col* している)

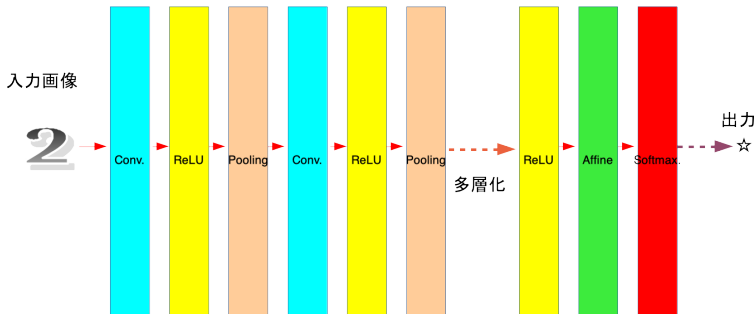
```
x2 = np.random.rand(10, 3, 7, 7)
col2 = im2col(x2, 5, 5, stride=1, pad=0)
print(col2.shape)
```

- col2 の結果は 90,75 になるはず
- 90 は 5x5 のフィルタを 7x7 のデータに適用するので適用回数 9 にバッチ数 10 を掛けたもの
- 75 は 5x5 のフィルタが 3 チャンネル分

# 復習：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

# 畳み込みニューラルネット（再掲）



# 畳み込み層の実装

-  , ,  
を引数にもつ
- フィルタの展開には reshape を用いている
- forward の実装では出力サイズを整形している
- 逆伝搬のときには im2col の逆の処理  を使う（詳細は略）

# 畳み込み層の実装

- フィルタ（重み）, ,  
,  を引数にもつ
- フィルタの展開には reshape を用いている
- forward の実装では出力サイズを整形している
- 逆伝搬のときには im2col の逆の処理 を使う（詳細は略）

# 畳み込み層の実装

- フィルタ（重み）, バイアス,   
 ,  を引数にもつ
- フィルタの展開には reshape を用いている
- forward の実装では出力サイズを整形している
- 逆伝搬のときには im2col の逆の処理    
 を使う（詳細は略）

# 畳み込み層の実装

- フィルタ（重み）, バイアス, スライド,  を引数にもつ
- フィルタの展開には reshape を用いている
- forward の実装では出力サイズを整形している
- 逆伝搬のときには im2col の逆の処理  を使う（詳細は略）

# 畳み込み層の実装

- フィルタ（重み）, バイアス, スライド, パディング を引数にもつ
- フィルタの展開には reshape を用いている
- forward の実装では出力サイズを整形している
- 逆伝搬のときには im2col の逆の処理  を使う（詳細は略）

# 畳み込み層の実装

- フィルタ（重み）, バイアス, スライド, パディング を引数にもつ
- フィルタの展開には reshape を用いている
- forward の実装では出力サイズを整形している
- 逆伝搬のときには im2col の逆の処理 col2im を使う（詳細は略）

# 畳み込み層の実装

☆以下が実装コード（後でまとめたもののリンクがあるので、ここではコードの紹介だけ）

```
class Convolution:
    def __init__(self, W, b, stride=1, pad=0):
        self.W = W
        self.b = b
        self.stride = stride
        self.pad = pad
        # 中間データ (backward 時に使用)
        self.x = None
        self.col = None
        self.col_W = None
        # 重み・バイアスパラメータの勾配
        self.dW = None
        self.db = None
```

(…次頁に続く)

# 畳み込み層の実装

## (前頁からの続き)

```
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
```

## (…次頁に続く)

# 畳み込み層の実装

## (前頁からの続き)

```
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
```

# プーリング層の実装

- `im2col` を使って入力を展開
  - チャンネル方向には  であることに注意
- `forward` では展開した行列にたいして行ごとに  を求める
- 出力を整形

# プーリング層の実装

- `im2col` を使って入力を展開
  - チャンネル方向には **独立** であることに注意
- `forward` では展開した行列にたいして行ごとに  を求める
- 出力を整形

# プーリング層の実装

- im2col を使って入力を展開
  - チャンネル方向には **独立** であることに注意
- forward では展開した行列にたいして行ごとに **最大値** を求める
- 出力を整形

# プーリング層の実装

☆以下が実装コード（後でまとめたもののリンクがあるので、ここではコードの紹介だけ）

```
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
        self.x = None
        self.arg_max = None
```

(…次頁に続く)

# プーリング層の実装

## (前頁からの続き)

```
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
```

## (…次頁に続く)

# プーリング層の実装

## (前頁からの続き)

```
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
```

# 畳み込みニューラルネットの実装

## ☆引数は以下の通り

- `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（フィルター数、フィルターサイズ、ストライド、パディング）
- `hidden_size`: 隠れ層のニューロンの数
- `output_size`: 出力層のニューロンの数
- `weight_init_std`: 初期化のときの重みの標準偏差

# 畳み込みニューラルネットの実装

☆以下が実装コード（後でまとめたもののリンクがあるので、ここではコードの紹介だけ。長い行は適宜折り込んでいる。）

```
# coding: utf-8
import sys, os, pickle
import numpy as np
from collections import OrderedDict
from layers import *
from gradient import numerical_gradient
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):
```

(…次頁に続く)

# 畳み込みニューラルネットの実装

(前頁からの続き)

```
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))
```

(…次頁に続く)

# 畳み込みニューラルネットの実装

(前頁からの続き)

```
# 重みの初期化
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)
```

(…次頁に続く)

# 畳み込みニューラルネットの実装

(前頁からの続き)

```
# レイヤの生成
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()
```

(…次頁に続く)

# 畳み込みニューラルネットの実装

(前頁からの続き)

```
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.last_layer.forward(y, t)
def accuracy(self, x, t, batch_size=100):
    if t.ndim != 1 : t = np.argmax(t, axis=1)
    acc = 0.0
    for i in range(int(x.shape[0] / batch_size)):
        tx = x[i*batch_size:(i+1)*batch_size]
        tt = t[i*batch_size:(i+1)*batch_size]
        y = self.predict(tx)
        y = np.argmax(y, axis=1)
        acc += np.sum(y == tt)
    return acc / x.shape[0]
```

(…次頁に続く)

# 畳み込みニューラルネットの実装

(前頁からの続き)

```
def numerical_gradient(self, x, t):  
    loss_w = lambda w: self.loss(x, t)  
    grads = {}  
    for idx in (1, 2, 3):  
        grads['W' + str(idx)] = numerical_gradient(loss_w, \  
                                                    self.params['W' + str(idx)])  
        grads['b' + str(idx)] = numerical_gradient(loss_w, \  
                                                    self.params['b' + str(idx)])  
    return grads
```

(…次頁に続く)

# 畳み込みニューラルネットの実装

## (前頁からの続き)

```
def gradient(self, x, t):
    # forward
    self.loss(x, t)
    # backward # init. dout = 1
    dout = self.last_layer.backward(1)
    layers = list(self.layers.values())
    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)
    # 設定
    grads = {}
    grads['W1'], grads['b1'] = self.layers['Conv1'].dW, \
                               self.layers['Conv1'].db
    grads['W2'], grads['b2'] = self.layers['Affine1'].dW, \
                               self.layers['Affine1'].db
    grads['W3'], grads['b3'] = self.layers['Affine2'].dW, \
                               self.layers['Affine2'].db
    return grads
```

## (…次頁に続く)

# 畳み込みニューラルネットの実装

## (前頁からの続き)

```
def save_params(self, file_name="params.pkl"):
    params = {}
    for key, val in self.params.items():
        params[key] = val
    with open(file_name, 'wb') as f:
        pickle.dump(params, f)
def load_params(self, file_name="params.pkl"):
    with open(file_name, 'rb') as f:
        params = pickle.load(f)
    for key, val in params.items():
        self.params[key] = val
    for i, key in enumerate(['Conv1', 'Affine1', 'Affine2']):
        self.layers[key].W = self.params['W' + str(i+1)]
        self.layers[key].b = self.params['b' + str(i+1)]
```

# 畳み込みニューラルネットを使った学習

- まず、[mym.zip](#) のリンク先を解凍し、そのファイル（フォルダの中身だけ）を Colab. の mymodules の中に保存する（一部のファイルは上書きされる）
- その上で、[con\\_nn.py](#)（以下はコード）のリンク先を Colab. にコピー＆ペーストして実行する

```
# coding: utf-8
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
bd = 'drive/My Drive/Colab Notebooks/mymodules'
import sys, os
sys.path.append(bd) # mymodules のファイルをインポートするための設定
import numpy as np
import matplotlib.pyplot as plt
from mnist import load_mnist
```

(…次頁に続く)

# 畳み込みニューラルネットを使った学習

## (前頁からの続き)

```
from simple_convnet import SimpleConvNet
from trainer import Trainer
# データの読み込み
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=False)
max_epochs = 10 # 20 だと十二分?!
network = SimpleConvNet(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)
trainer = Trainer(network, x_train, t_train, x_test, t_test, \
    epochs=max_epochs, mini_batch_size=100, \
    optimizer='Adam', optimizer_param={'lr': 0.001}, \
    evaluate_sample_num_per_epoch=1000)
trainer.train()
# パラメータの保存
network.save_params("params.pkl")
print("Saved Network Parameters!")
```

## (…次頁に続く)

# 畳み込みニューラルネットを使った学習

## (前頁からの続き)

### # グラフの描画

```
markers = {'train': 'o', 'test': 's'}
x = np.arange(max_epochs)
plt.plot(x, trainer.train_acc_list, marker='o', \
         label='train', markevery=2)
plt.plot(x, trainer.test_acc_list, marker='s', \
         label='test', markevery=2)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

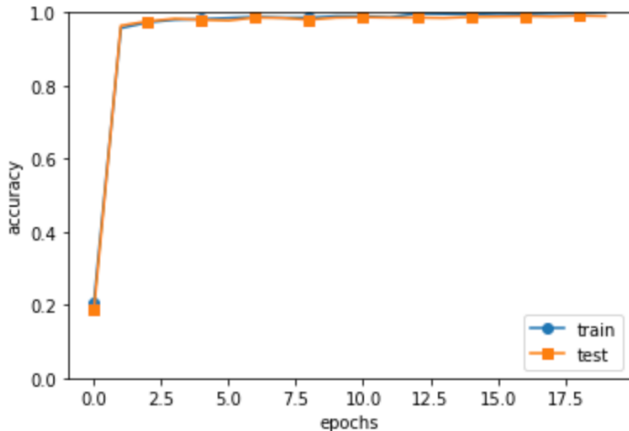
# 畳み込みニューラルネットを使った学習

◎**実行結果**…結構時間がかかるので注意！前田の予備実験では40分くらいかかりました。なので、実行しはじめたら、のんびり構えてください。約99%の認識率となっています。ちなみにバックプロパゲーションでは約97%でした。

```
Mounted at /content/drive
(略)
train loss:3.3783047300109845e-05
train loss:0.00016617674794723952
===== Final Test Accuracy =====
test acc:0.988
Saved Network Parameters!
```

# 実行結果

===== Final Test Accuracy =====  
test acc:0.988  
Saved Network Parameters!



# 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！