実装演習：2-4.畳み込みニューラルネットワークの概念

In [1]:

```
import pickle
import numpy as np
from collections import OrderedDict
from common import layers
from common import optimizer
from data.mnist import load_mnist
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm
plt.style.use('ggplot')
```

In [2]:

```
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    """画像データを2次元配列に変換

    Parameters
    -------------
    input_data: 入力値
    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
    """
    # N: numer, C: channel, H: height, W: width
    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]

    # (N, C, filter_h, filter_w, out_h, out_w) -> (N, filter_w, out_h, out_w, C, filter_h)
    col = col.transpose(0, 4, 5, 1, 2, 3)
    col = col.reshape(N * out_h * out_w, -1)

    return col
```

im2colの処理確認

In [3]:

```
# number, channel, height, widthを表す
input_data = np.random.rand(2, 1, 4, 4) * 100 // 1
print(f'input_data:\n{input_data}')
```

```
input_data:
[[[[92. 65. 70. 14.]
   [ 6. 80. 39. 32.]
   [89. 85. 12. 85.]
   [40. 30. 53. 56.]]]


 [[[71.  8. 85. 49.]
   [61. 43. 23. 83.]
   [77. 10. 94. 98.]
   [27. 28. 89. 57.]]]]
```

In [4]:

```
filter_h = 3
filter_w = 3
stride = 1
pad = 0
```

In [5]:

```
col = im2col(input_data, filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
print(f'col:\n{col}')
```

```
col:
[[92. 65. 70.  6. 80. 39. 89. 85. 12.]
 [65. 70. 14. 80. 39. 32. 85. 12. 85.]
 [ 6. 80. 39. 89. 85. 12. 40. 30. 53.]
 [80. 39. 32. 85. 12. 85. 30. 53. 56.]
 [71.  8. 85. 61. 43. 23. 77. 10. 94.]
 [ 8. 85. 49. 43. 23. 83. 10. 94. 98.]
 [61. 43. 23. 77. 10. 94. 27. 28. 89.]
 [43. 23. 83. 10. 94. 98. 28. 89. 57.]]
```

column to image

In [6]:

```
def col2im(col, input_shape, filter_h, filter_w, stride=1, pad=0):
    """2次元配列を画像データに変換"""
    # N: number, C: channel, H: height, W: width
    N, C, H, W = input_shape

    # 切り捨て除算
    out_h = (H + 2 * pad - filter_h) // stride + 1
    out_w = (W + 2 * pad - filter_w) // stride + 1

    # (N, filter_h, filter_w, out_h, out_w, C)
    col = col.reshape(N, out_h, out_w, C, filter_h, filter_w).transpose(0, 3, 4, 5, 1, 2)

    img = np.zeros((N, C, H + 2 * pad + stride - 1, W + 2 * pad + stride - 1))

    for y in range(filter_h):
        y_max = y + stride * out_h

        for x in range(filter_w):
            x_max = x + stride * out_w
            img[:, :, y:y_max:stride, x:x_max:stride] += col[:, :, y, x, :, :]

    return img[:, :, pad:H + pad, pad:W + pad]
```

convolution class

In [7]:

```
class Convolution:
    """畳み込みクラス

    W: フィルター
    b: バイアス
    stride: ストライド
    pad: パディング
    """
    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

        # フィルター・バイアスパラメータの勾配
        self.dW = None
        self.db = None

    def forward(self, x):
        """順伝搬"""
        # FN: filter_number, C: channel, FH: filter_height, FW: filter_width
        FN, C, FH, FW = self.W.shape
        N, C, H, W = x.shape

        # 出力値のheight, width
        out_h = 1 + int((H + 2 * self.pad - FH) / self.stride)
        out_w = 1 + int((W + 2 * self.pad - FW) / self.stride)

        # xを行列に変換
        col = im2col(x, FH, FW, self.stride, self.pad)
        # フィルターをxに合わせた行列に変換
        col_W = self.W.reshape(FN, -1).T

        out = np.dot(col, col_W) + self.b
        # 計算のために変えた形式を戻す
        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

    def backward(self, dout):
        """逆伝搬"""
        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)
        # dcolを画像データに変換
        dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.pad)

        return dx
```

pooling class

In [8]:

```
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

    def forward(self, x):
        """順伝搬"""
        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)

        # xを行列に変換
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        # プーリングのサイズに合わせてリサイズ
        col = col.reshape(-1, self.pool_h*self.pool_w)

        # 行ごとに最大値を求める
        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)
        # 整形
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)

        self.x = x
        self.arg_max = arg_max

        return out

    def backward(self, dout):
        """逆伝搬"""
        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
```

simple convolution network class

In [9]:

```
class SimpleConvNet:
    # conv - relu - pool - affine - relu - affine - softmax
    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))

        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(filter_num, input_dim[0], filte
r_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)

        # レイヤの生成
        self.layers = OrderedDict()
        self.layers['Conv1'] = layers.Convolution(self.params['W1'], self.params['b1'], conv_pa
ram['stride'], conv_param['pad'])
        self.layers['Relu1'] = layers.Relu()
        self.layers['Pool1'] = layers.Pooling(pool_h=2, pool_w=2, stride=2)
        self.layers['Affine1'] = layers.Affine(self.params['W2'], self.params['b2'])
        self.layers['Relu2'] = layers.Relu()
        self.layers['Affine2'] = layers.Affine(self.params['W3'], self.params['b3'])

        self.last_layer = layers.SoftmaxWithLoss()

    def predict(self, x):
        for key in self.layers.keys():
            x = self.layers[key].forward(x)
        return x

    def loss(self, x, d):
        y = self.predict(x)
        return self.last_layer.forward(y, d)

    def accuracy(self, x, d, batch_size=100):
        if d.ndim != 1 : d = np.argmax(d, axis=1)

        acc = 0.0

        for i in range(int(x.shape[0] / batch_size)):
            tx = x[i*batch_size:(i+1)*batch_size]
            td = d[i*batch_size:(i+1)*batch_size]
            y = self.predict(tx)
            y = np.argmax(y, axis=1)
            acc += np.sum(y == td)

        return acc / x.shape[0]

    def gradient(self, x, d):
        # forward
        self.loss(x, d)
        # backward
        dout = 1
        dout = self.last_layer.backward(dout)
        layers = list(self.layers.values())

        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 設定
        grad = {}
        grad['W1'], grad['b1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db
        grad['W2'], grad['b2'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
        grad['W3'], grad['b3'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

        return grad
```

In [10]:

```
# MNISTデータの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(flatten=False)
```

In [11]:

```
# 処理に時間のかかる場合はデータを削減
x_train, d_train = x_train[:5000], d_train[:5000]
x_test, d_test = x_test[:1000], d_test[:1000]
```

In [12]:

```
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)
```

In [13]:

```
optimizer = optimizer.Adam()
```

In [14]:

```
iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
plot_interval=10
```

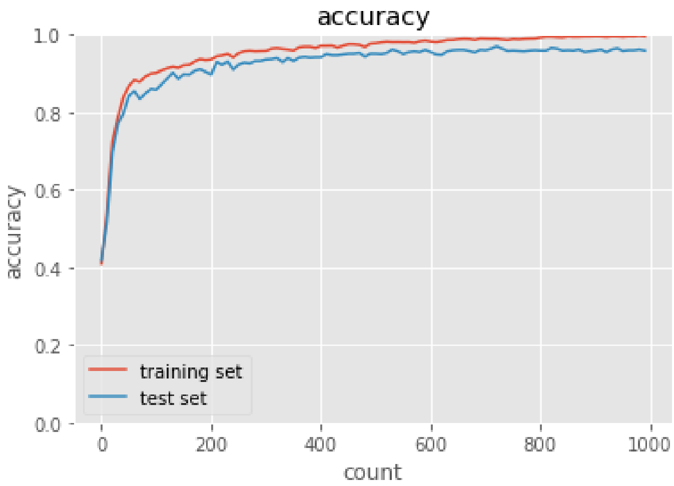
In [15]:

```
train_loss_list = []
accuracies_train = []
accuracies_test = []

for i in tqdm(range(iters_num)):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)

    loss = network.loss(x_batch, d_batch)
    train_loss_list.append(loss)

    if (i+1) % plot_interval == 0:
        accr_train = network.accuracy(x_train, d_train)
        accr_test = network.accuracy(x_test, d_test)
        accuracies_train.append(accr_train)
        accuracies_test.append(accr_test)

        if (i+1) % (plot_interval * 10) == 0:
            print(f'iter: {i+1}. accr_train={accr_train}, accr_test={accr_test}')
```

```
iter: 100. accr_train=0.8996, accr_test=0.86
iter: 200. accr_train=0.9344, accr_test=0.904
iter: 300. accr_train=0.9578, accr_test=0.932
iter: 400. accr_train=0.9658, accr_test=0.942
iter: 500. accr_train=0.9768, accr_test=0.951
iter: 600. accr_train=0.985, accr_test=0.961
iter: 700. accr_train=0.9902, accr_test=0.961
iter: 800. accr_train=0.9902, accr_test=0.96
iter: 900. accr_train=0.9954, accr_test=0.957
iter: 1000. accr_train=0.9956, accr_test=0.959
```

In [16]:

```
lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test,  label="test set")
plt.legend()
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.show()
```


accuracy
accuracy
training set
test set
count