④畳み込みニューラルネットワークの概念

=====

目次:

1.単純な畳み込みネットワーク(simple convolution network)　[畳み込み層・プーリング層が1セット]

2.2層の畳み込みネットワーク(double comvolution network)　[畳み込み層・プーリング層が2セット]

【要約】

・畳み込みニューラルネットワーク(CNN)　→画像の識別、処理によく使われるニューラルネットワーク

・CNNの構造図(例)　→新たに畳み込み層、プーリング層を使用

　入力層(入力画像)→畳み込み層→畳み込み層→プーリング層→畳み込み層→畳み込み層→プーリング層→全結合層→出力層(出力画像)

In [1]:

```
import sys
sys.path.append('C:/Users/NIF/Desktop/(削除)rabitt/DNN_code_colab_lesson_1_2/DNN_code_colab_lesson_1_2')
```

=====

1.単純な畳み込みネットワーク(simple convolution network)　[畳み込み層・プーリング層が1セット]

・画像データを入力し、フィルターを掛けるために2次元配列(行列)に変換する処理(im2colを利用)

In [2]:

```
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

# 画像データを２次元配列に変換
'''
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
'''
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    # N: number, 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]

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

    col = col.reshape(N * out_h * out_w, -1)
    return col
```

[参考] im2colの動作

In [3]:

```
# im2colの処理確認
input_data = np.random.rand(2, 1, 4, 4)*100//1 # number, channel, height, widthを表す
print('========== input_data ==========¥n', input_data)
print('==============================')
filter_h = 3
filter_w = 3
stride = 1
pad = 0
col = im2col(input_data, filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
print('============= col =============¥n', col)
print('==============================')
```

```
========== input_data ==========
 [[[[20. 95. 70.  5.]
   [ 3. 91. 59. 59.]
   [81. 15. 21. 32.]
   [38. 32. 52. 89.]]]


 [[[45. 94. 38.  4.]
   [18. 27. 44. 10.]
   [22. 42.  9. 62.]
   [20. 89. 26. 23.]]]]
==============================
============= col =============
 [[20. 95. 70.  3. 91. 59. 81. 15. 21.]
 [95. 70.  5. 91. 59. 59. 15. 21. 32.]
 [ 3. 91. 59. 81. 15. 21. 38. 32. 52.]
 [91. 59. 59. 15. 21. 32. 32. 52. 89.]
 [45. 94. 38. 18. 27. 44. 22. 42.  9.]
 [94. 38.  4. 27. 44. 10. 42.  9. 62.]
 [18. 27. 44. 22. 42.  9. 20. 89. 26.]
 [27. 44. 10. 42.  9. 62. 89. 26. 23.]]
==============================
```

・再び画像データに戻す

In [4]:

```
# ２次元配列を画像データに変換
def col2im(col, input_shape, filter_h, filter_w, stride=1, pad=0):
    # 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
    col = col.reshape(N, out_h, out_w, C, filter_h, filter_w).transpose(0, 3, 4, 5, 1, 2) # (N,
 filter_h, filter_w, out_h, out_w, C)

    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]
```

・畳み込み層の実装(クラス)

In [5]:

```
class Convolution:
    # W: フィルター, b: バイアス
    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):
        # 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):
        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
```

・プーリング層の実装(クラス)

In [6]:

```
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):
        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):
        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
```

→対象領域のMax値を取得(Max Pooling)

・単純な畳み込みネットワークの実装(クラス)

In [7]:

```
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, 'pa
d':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], filter_s
ize, 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_par
am['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
```

→順伝播の順番:Conv1→Relu1→Pool1→Affine1→Relu2→Affine2→最後にsoftmax関数で処理

[内訳]

畳み込み層:Conv1→Relu1

プーリング層:Pool1

全結合層:Affine1→Relu2

出力層:Affine2→最後にsoftmax関数で処理

・単純な畳み込みネットワークでの学習及び予測

In [8]:

```
from common import optimizer

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

print("データ読み込み完了")

# 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]


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)

optimizer = optimizer.Adam()

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

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

plot_interval=10



for i in 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)

        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

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(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
```

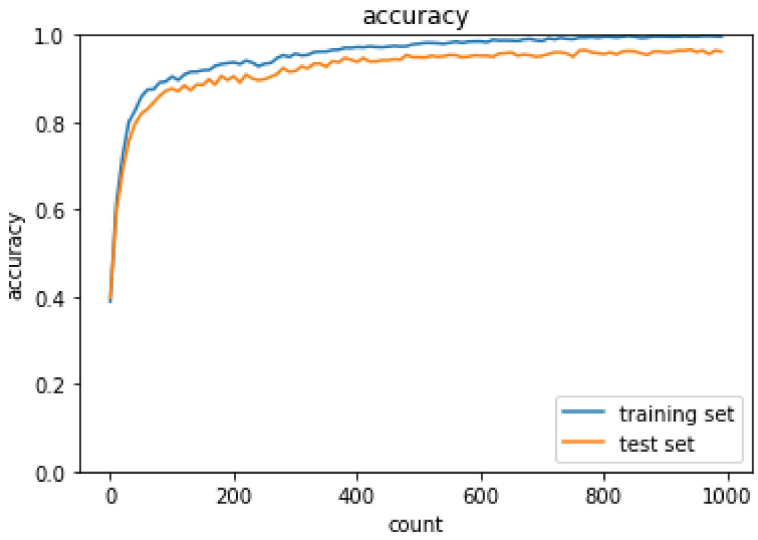
```
# グラフの表示
plt.show()
```

```
データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.389
                : 10. 正答率(テスト) = 0.398
Generation: 20. 正答率(トレーニング) = 0.616
                : 20. 正答率(テスト) = 0.595
Generation: 30. 正答率(トレーニング) = 0.7238
                : 30. 正答率(テスト) = 0.689
Generation: 40. 正答率(トレーニング) = 0.8008
                : 40. 正答率(テスト) = 0.757
Generation: 50. 正答率(トレーニング) = 0.8262
                : 50. 正答率(テスト) = 0.796
Generation: 60. 正答率(トレーニング) = 0.8576
                : 60. 正答率(テスト) = 0.819
Generation: 70. 正答率(トレーニング) = 0.8748
                : 70. 正答率(テスト) = 0.83
Generation: 80. 正答率(トレーニング) = 0.8754
                : 80. 正答率(テスト) = 0.845
Generation: 90. 正答率(トレーニング) = 0.8908
                : 90. 正答率(テスト) = 0.86
Generation: 100. 正答率(トレーニング) = 0.893
                : 100. 正答率(テスト) = 0.872
Generation: 110. 正答率(トレーニング) = 0.9046
                : 110. 正答率(テスト) = 0.877
Generation: 120. 正答率(トレーニング) = 0.8962
                : 120. 正答率(テスト) = 0.871
Generation: 130. 正答率(トレーニング) = 0.909
                : 130. 正答率(テスト) = 0.884
Generation: 140. 正答率(トレーニング) = 0.9152
                : 140. 正答率(テスト) = 0.873
Generation: 150. 正答率(トレーニング) = 0.9154
                : 150. 正答率(テスト) = 0.886
Generation: 160. 正答率(トレーニング) = 0.9188
                : 160. 正答率(テスト) = 0.885
Generation: 170. 正答率(トレーニング) = 0.9194
                : 170. 正答率(テスト) = 0.898
Generation: 180. 正答率(トレーニング) = 0.929
                : 180. 正答率(テスト) = 0.886
Generation: 190. 正答率(トレーニング) = 0.933
                : 190. 正答率(テスト) = 0.906
Generation: 200. 正答率(トレーニング) = 0.9352
                : 200. 正答率(テスト) = 0.896
Generation: 210. 正答率(トレーニング) = 0.937
                : 210. 正答率(テスト) = 0.905
Generation: 220. 正答率(トレーニング) = 0.933
                : 220. 正答率(テスト) = 0.891
Generation: 230. 正答率(トレーニング) = 0.9412
                : 230. 正答率(テスト) = 0.909
Generation: 240. 正答率(トレーニング) = 0.9364
                : 240. 正答率(テスト) = 0.9
Generation: 250. 正答率(トレーニング) = 0.928
                : 250. 正答率(テスト) = 0.896
Generation: 260. 正答率(トレーニング) = 0.9338
                : 260. 正答率(テスト) = 0.898
Generation: 270. 正答率(トレーニング) = 0.9354
                : 270. 正答率(テスト) = 0.904
Generation: 280. 正答率(トレーニング) = 0.9466
                : 280. 正答率(テスト) = 0.91
Generation: 290. 正答率(トレーニング) = 0.9534
                : 290. 正答率(テスト) = 0.924
Generation: 300. 正答率(トレーニング) = 0.9492
                : 300. 正答率(テスト) = 0.915
```

```
Generation: 310. 正答率(トレーニング) = 0.9568
                : 310. 正答率(テスト) = 0.917
Generation: 320. 正答率(トレーニング) = 0.9528
                : 320. 正答率(テスト) = 0.928
Generation: 330. 正答率(トレーニング) = 0.954
                : 330. 正答率(テスト) = 0.923
Generation: 340. 正答率(トレーニング) = 0.9606
                : 340. 正答率(テスト) = 0.934
Generation: 350. 正答率(トレーニング) = 0.9616
                : 350. 正答率(テスト) = 0.934
Generation: 360. 正答率(トレーニング) = 0.9618
                : 360. 正答率(テスト) = 0.927
Generation: 370. 正答率(トレーニング) = 0.9654
                : 370. 正答率(テスト) = 0.94
Generation: 380. 正答率(トレーニング) = 0.966
                : 380. 正答率(テスト) = 0.937
Generation: 390. 正答率(トレーニング) = 0.9704
                : 390. 正答率(テスト) = 0.947
Generation: 400. 正答率(トレーニング) = 0.97
                : 400. 正答率(テスト) = 0.943
Generation: 410. 正答率(トレーニング) = 0.972
                : 410. 正答率(テスト) = 0.938
Generation: 420. 正答率(トレーニング) = 0.9712
                : 420. 正答率(テスト) = 0.947
Generation: 430. 正答率(トレーニング) = 0.9736
                : 430. 正答率(テスト) = 0.939
Generation: 440. 正答率(トレーニング) = 0.9722
                : 440. 正答率(テスト) = 0.939
Generation: 450. 正答率(トレーニング) = 0.9714
                : 450. 正答率(テスト) = 0.942
Generation: 460. 正答率(トレーニング) = 0.973
                : 460. 正答率(テスト) = 0.942
Generation: 470. 正答率(トレーニング) = 0.9746
                : 470. 正答率(テスト) = 0.944
Generation: 480. 正答率(トレーニング) = 0.9734
                : 480. 正答率(テスト) = 0.943
Generation: 490. 正答率(トレーニング) = 0.974
                : 490. 正答率(テスト) = 0.954
Generation: 500. 正答率(トレーニング) = 0.9784
                : 500. 正答率(テスト) = 0.949
Generation: 510. 正答率(トレーニング) = 0.9798
                : 510. 正答率(テスト) = 0.949
Generation: 520. 正答率(トレーニング) = 0.9818
                : 520. 正答率(テスト) = 0.948
Generation: 530. 正答率(トレーニング) = 0.9818
                : 530. 正答率(テスト) = 0.952
Generation: 540. 正答率(トレーニング) = 0.981
                : 540. 正答率(テスト) = 0.95
Generation: 550. 正答率(トレーニング) = 0.9796
                : 550. 正答率(テスト) = 0.951
Generation: 560. 正答率(トレーニング) = 0.982
                : 560. 正答率(テスト) = 0.954
Generation: 570. 正答率(トレーニング) = 0.9844
                : 570. 正答率(テスト) = 0.953
Generation: 580. 正答率(トレーニング) = 0.982
                : 580. 正答率(テスト) = 0.949
Generation: 590. 正答率(トレーニング) = 0.9844
                : 590. 正答率(テスト) = 0.95
Generation: 600. 正答率(トレーニング) = 0.9852
                : 600. 正答率(テスト) = 0.953
Generation: 610. 正答率(トレーニング) = 0.9852
```

```
                : 610. 正答率(テスト) = 0.952
Generation: 620. 正答率(トレーニング) = 0.9834
                : 620. 正答率(テスト) = 0.952
Generation: 630. 正答率(トレーニング) = 0.9886
                : 630. 正答率(テスト) = 0.95
Generation: 640. 正答率(トレーニング) = 0.9872
                : 640. 正答率(テスト) = 0.957
Generation: 650. 正答率(トレーニング) = 0.9874
                : 650. 正答率(テスト) = 0.958
Generation: 660. 正答率(トレーニング) = 0.9872
                : 660. 正答率(テスト) = 0.96
Generation: 670. 正答率(トレーニング) = 0.9862
                : 670. 正答率(テスト) = 0.952
Generation: 680. 正答率(トレーニング) = 0.9894
                : 680. 正答率(テスト) = 0.955
Generation: 690. 正答率(トレーニング) = 0.9906
                : 690. 正答率(テスト) = 0.953
Generation: 700. 正答率(トレーニング) = 0.9874
                : 700. 正答率(テスト) = 0.95
Generation: 710. 正答率(トレーニング) = 0.9862
                : 710. 正答率(テスト) = 0.952
Generation: 720. 正答率(トレーニング) = 0.992
                : 720. 正答率(テスト) = 0.956
Generation: 730. 正答率(トレーニング) = 0.9896
                : 730. 正答率(テスト) = 0.96
Generation: 740. 正答率(トレーニング) = 0.9932
                : 740. 正答率(テスト) = 0.96
Generation: 750. 正答率(トレーニング) = 0.9906
                : 750. 正答率(テスト) = 0.957
Generation: 760. 正答率(トレーニング) = 0.9902
                : 760. 正答率(テスト) = 0.95
Generation: 770. 正答率(トレーニング) = 0.9938
                : 770. 正答率(テスト) = 0.964
Generation: 780. 正答率(トレーニング) = 0.9938
                : 780. 正答率(テスト) = 0.965
Generation: 790. 正答率(トレーニング) = 0.995
                : 790. 正答率(テスト) = 0.96
Generation: 800. 正答率(トレーニング) = 0.9934
                : 800. 正答率(テスト) = 0.958
Generation: 810. 正答率(トレーニング) = 0.9956
                : 810. 正答率(テスト) = 0.956
Generation: 820. 正答率(トレーニング) = 0.9956
                : 820. 正答率(テスト) = 0.96
Generation: 830. 正答率(トレーニング) = 0.994
                : 830. 正答率(テスト) = 0.955
Generation: 840. 正答率(トレーニング) = 0.9958
                : 840. 正答率(テスト) = 0.962
Generation: 850. 正答率(トレーニング) = 0.9972
                : 850. 正答率(テスト) = 0.963
Generation: 860. 正答率(トレーニング) = 0.9956
                : 860. 正答率(テスト) = 0.962
Generation: 870. 正答率(トレーニング) = 0.9928
                : 870. 正答率(テスト) = 0.958
Generation: 880. 正答率(トレーニング) = 0.994
                : 880. 正答率(テスト) = 0.954
Generation: 890. 正答率(トレーニング) = 0.9966
                : 890. 正答率(テスト) = 0.962
Generation: 900. 正答率(トレーニング) = 0.9962
                : 900. 正答率(テスト) = 0.962
Generation: 910. 正答率(トレーニング) = 0.9962
                : 910. 正答率(テスト) = 0.96
```

```
Generation: 920. 正答率(トレーニング) = 0.9954
                 : 920. 正答率(テスト) = 0.961
Generation: 930. 正答率(トレーニング) = 0.9958
                 : 930. 正答率(テスト) = 0.964
Generation: 940. 正答率(トレーニング) = 0.997
                 : 940. 正答率(テスト) = 0.964
Generation: 950. 正答率(トレーニング) = 0.9962
                 : 950. 正答率(テスト) = 0.966
Generation: 960. 正答率(トレーニング) = 0.9962
                 : 960. 正答率(テスト) = 0.961
Generation: 970. 正答率(トレーニング) = 0.9974
                 : 970. 正答率(テスト) = 0.964
Generation: 980. 正答率(トレーニング) = 0.9974
                 : 980. 正答率(テスト) = 0.956
Generation: 990. 正答率(トレーニング) = 0.9968
                 : 990. 正答率(テスト) = 0.964
Generation: 1000. 正答率(トレーニング) = 0.9962
                 : 1000. 正答率(テスト) = 0.962
```



=====

2.2層の畳み込みネットワーク(double comvolution network)　[畳み込み層・プーリング層が2セット]

In [10]:

```
import sys
sys.path.append('C:/Users/NIF/Desktop/(削除)rabitt/DNN_code_colab_lesson_1_2/DNN_code_colab_lesson_1_2')
```

・2層の畳み込みネットワークの実装(クラス)

In [15]:

```
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

class DoubleConvNet:
    # conv - relu - pool - conv - relu - pool - affine - relu - affine - softmax
    def __init__(self, input_dim=(1, 28, 28),
                 conv_param_1={'filter_num':10, 'filter_size':7, 'pad':1, 'stride':1},
                 conv_param_2={'filter_num':20, 'filter_size':3, 'pad':1, 'stride':1},
                 hidden_size=100, output_size=10, weight_init_std=0.01):
        conv_output_size_1 = (input_dim[1] - conv_param_1['filter_size'] + 2 * conv_param_1['pa
d']) / conv_param_1['stride'] + 1
        conv_output_size_2 = (conv_output_size_1 / 2 - conv_param_2['filter_size'] + 2 * conv_pa
ram_2['pad']) / conv_param_2['stride'] + 1
        pool_output_size = int(conv_param_2['filter_num'] * (conv_output_size_2 / 2) * (conv_out
put_size_2 / 2))
        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(conv_param_1['filter_num'], input_
dim[0], conv_param_1['filter_size'], conv_param_1['filter_size'])
        self.params['b1'] = np.zeros(conv_param_1['filter_num'])
        self.params['W2'] = weight_init_std * np.random.randn(conv_param_2['filter_num'], conv_p
aram_1['filter_num'], conv_param_2['filter_size'], conv_param_2['filter_size'])
        self.params['b2'] = np.zeros(conv_param_2['filter_num'])
        self.params['W3'] = weight_init_std * np.random.randn(pool_output_size, hidden_size)
        self.params['b3'] = np.zeros(hidden_size)
        self.params['W4'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b4'] = 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_par
am_1['stride'], conv_param_1['pad'])
        self.layers['Relu1'] = layers.Relu()
        self.layers['Pool1'] = layers.Pooling(pool_h=2, pool_w=2, stride=2)
        self.layers['Conv2'] = layers.Convolution(self.params['W2'], self.params['b2'], conv_par
am_2['stride'], conv_param_2['pad'])
        self.layers['Relu2'] = layers.Relu()
        self.layers['Pool2'] = layers.Pooling(pool_h=2, pool_w=2, stride=2)
        self.layers['Affine1'] = layers.Affine(self.params['W3'], self.params['b3'])
        self.layers['Relu3'] = layers.Relu()
        self.layers['Affine2'] = layers.Affine(self.params['W4'], self.params['b4'])
        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['Conv2'].dW, self.layers['Conv2'].db
        grad['W3'], grad['b3'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
        grad['W4'], grad['b4'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

        return grad
```

→順伝播の順番:Conv1→Relu1→Pool1→Conv2→Relu2→Pool2→Affine1→Relu3→Affine2→最後にsoftmax関数で処理

[内訳]

畳み込み層:Conv1→Relu1

プーリング層:Pool1

畳み込み層:Conv2→Relu2

プーリング層:Pool2

全結合層:Affine1→Relu3

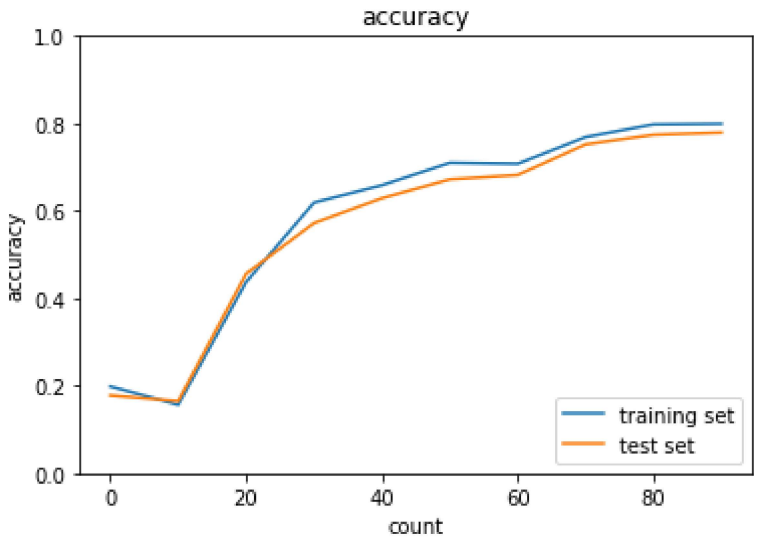
出力層:Affine2→最後にsoftmax関数で処理

・2層の畳み込みネットワークでの学習及び予測

In [16]:

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

print("データ読み込み完了")
# 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]


network = DoubleConvNet(input_dim=(1,28,28),
                        conv_param_1={'filter_num':10, 'filter_size':7, 'pad':1, 'stride':1},
                        conv_param_2={'filter_num':20, 'filter_size':3, 'pad':1, 'stride':1},
                        hidden_size=100, output_size=10, weight_init_std=0.01)

optimizer = optimizer.Adam()

# 時間がかかるため100に設定
iters_num = 100
# iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

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

plot_interval=10



for i in 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)

        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

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(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

```
データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.1984
                : 10. 正答率(テスト) = 0.178
Generation: 20. 正答率(トレーニング) = 0.1566
                : 20. 正答率(テスト) = 0.165
Generation: 30. 正答率(トレーニング) = 0.437
                : 30. 正答率(テスト) = 0.456
Generation: 40. 正答率(トレーニング) = 0.619
                : 40. 正答率(テスト) = 0.572
Generation: 50. 正答率(トレーニング) = 0.6586
                : 50. 正答率(テスト) = 0.629
Generation: 60. 正答率(トレーニング) = 0.7098
                : 60. 正答率(テスト) = 0.672
Generation: 70. 正答率(トレーニング) = 0.7076
                : 70. 正答率(テスト) = 0.682
Generation: 80. 正答率(トレーニング) = 0.7688
                : 80. 正答率(テスト) = 0.752
Generation: 90. 正答率(トレーニング) = 0.7978
                : 90. 正答率(テスト) = 0.774
Generation: 100. 正答率(トレーニング) = 0.7992
                : 100. 正答率(テスト) = 0.779
```



In [ ]: