Section 4 畳み込みニューラルネットワークの概念

In [ ]:

```
```

In [20]:

```
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
import time
```

In [11]:

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

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

In [13]:

```
# 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

    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]
```

In [15]:

```
# 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('==============================')
col2im(col ,input_data.shape, filter_h=filter_h, filter_w=filter_w, stride=stride, pa
```

```
========== input_data ==========
 [[[[13. 76. 63. 39.]
   [71. 46.  3. 13.]
   [73. 62. 40. 30.]
   [98. 27.  4. 40.]]]


 [[[46. 95. 17. 22.]
   [41.  9.  3. 87.]
   [19. 53. 33.  0.]
   [69. 99. 13. 19.]]]]
==============================
[[[[[[13. 76.]
     [71. 46.]]

    [[76. 63.]
     [46.  3.]]

    [[63. 39.]
     [ 3. 13.]]]


   [[[71. 46.]
     [73. 62.]]

    [[46.  3.]
     [62. 40.]]

    [[ 3. 13.]
     [40. 30.]]]


   [[[73. 62.]
     [98. 27.]]

    [[62. 40.]
     [27.  4.]]

    [[40. 30.]
     [ 4. 40.]]]]]



 [[[[[46. 95.]
     [41.  9.]]

    [[95. 17.]
     [ 9.  3.]]

    [[17. 22.]
     [ 3. 87.]]]


   [[[41.  9.]
     [19. 53.]]

    [[ 9.  3.]
     [53. 33.]]

    [[ 3. 87.]
     [33.  0.]]]
```

```
   [[[19. 53.]
     [69. 99.]]

    [[53. 33.]
     [99. 13.]]

    [[33.  0.]
     [13. 19.]]]]]]
============= col ==============
 [[13. 76. 63. 71. 46.  3. 73. 62. 40.]
 [76. 63. 39. 46.  3. 13. 62. 40. 30.]
 [71. 46.  3. 73. 62. 40. 98. 27.  4.]
 [46.  3. 13. 62. 40. 30. 27.  4. 40.]
 [46. 95. 17. 41.  9.  3. 19. 53. 33.]
 [95. 17. 22.  9.  3. 87. 53. 33.  0.]
 [41.  9.  3. 19. 53. 33. 69. 99. 13.]
 [ 9.  3. 87. 53. 33.  0. 99. 13. 19.]]
==============================
```

Out[15]:

```
array([[[[ 13., 152., 126.,  39.],
         [142., 184.,  12.,  26.],
         [146., 248., 160.,  60.],
         [ 98.,  54.,   8.,  40.]]],


       [[[ 46., 190.,  34.,  22.],
         [ 82.,  36.,  12., 174.],
         [ 38., 212., 132.,   0.],
         [ 69., 198.,  26.,  19.]]]])
```

In [16]:

```
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

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.p

        return dx

class SimpleConvNet:
    # conv - relu - pool - affine - relu - affine - softmax
    def __init__(self, input_dim=(1, 28, 28), conv_param={'filter_num':30, 'filter_si
                 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_strid
        pool_output_size = int(filter_num * (conv_output_size / 2) * (conv_output_siz

        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(filter_num, input_dim[0
        self.params['b1'] = np.zeros(filter_num)
        self.params['W2'] = weight_init_std * np.random.randn(pool_output_size, hidde
        self.params['b2'] = np.zeros(hidden_size)
        self.params['W3'] = weight_init_std * np.random.randn(hidden_size, output_siz
        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'
        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
```

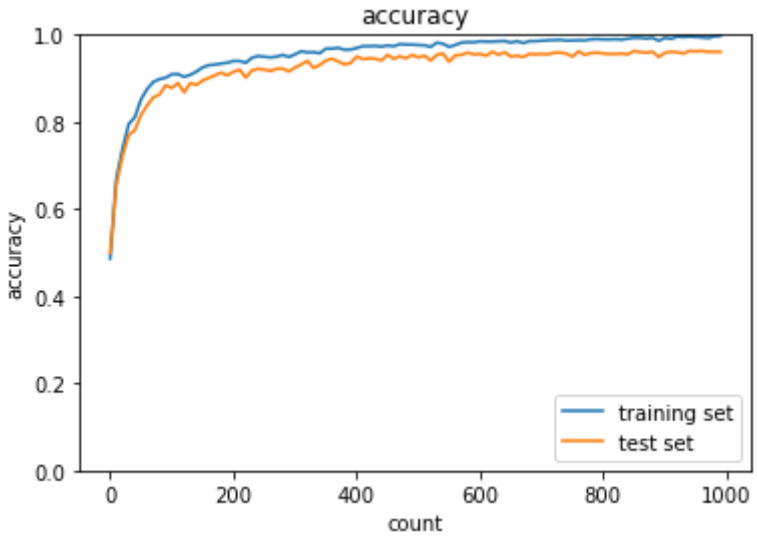
In [17]:

```
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_
                        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_trai
        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) = 0.4864
                : 10. 正答率(テスト) = 0.499
Generation: 20. 正答率(トレーニング) = 0.67
                : 20. 正答率(テスト) = 0.657
Generation: 30. 正答率(トレーニング) = 0.738
                : 30. 正答率(テスト) = 0.722
Generation: 40. 正答率(トレーニング) = 0.7948
                : 40. 正答率(テスト) = 0.77
Generation: 50. 正答率(トレーニング) = 0.8108
                : 50. 正答率(テスト) = 0.781
Generation: 60. 正答率(トレーニング) = 0.8516
                : 60. 正答率(テスト) = 0.816
Generation: 70. 正答率(トレーニング) = 0.8754
                : 70. 正答率(テスト) = 0.838
Generation: 80. 正答率(トレーニング) = 0.8918
                : 80. 正答率(テスト) = 0.856
Generation: 90. 正答率(トレーニング) = 0.8982
                : 90. 正答率(テスト) = 0.863
Generation: 100. 正答率(トレーニング) = 0.9014
                : 100. 正答率(テスト) = 0.884
Generation: 110. 正答率(トレーニング) = 0.9094
                : 110. 正答率(テスト) = 0.878
Generation: 120. 正答率(トレーニング) = 0.9098
                : 120. 正答率(テスト) = 0.889
Generation: 130. 正答率(トレーニング) = 0.9036
                : 130. 正答率(テスト) = 0.868
Generation: 140. 正答率(トレーニング) = 0.908
                : 140. 正答率(テスト) = 0.889
Generation: 150. 正答率(トレーニング) = 0.9156
                : 150. 正答率(テスト) = 0.885
Generation: 160. 正答率(トレーニング) = 0.9252
                : 160. 正答率(テスト) = 0.895
Generation: 170. 正答率(トレーニング) = 0.9302
                : 170. 正答率(テスト) = 0.901
Generation: 180. 正答率(トレーニング) = 0.9316
                : 180. 正答率(テスト) = 0.907
Generation: 190. 正答率(トレーニング) = 0.934
                : 190. 正答率(テスト) = 0.913
Generation: 200. 正答率(トレーニング) = 0.9358
                : 200. 正答率(テスト) = 0.907
Generation: 210. 正答率(トレーニング) = 0.9398
                : 210. 正答率(テスト) = 0.915
Generation: 220. 正答率(トレーニング) = 0.9398
                : 220. 正答率(テスト) = 0.92
Generation: 230. 正答率(トレーニング) = 0.9356
                : 230. 正答率(テスト) = 0.902
Generation: 240. 正答率(トレーニング) = 0.9472
                : 240. 正答率(テスト) = 0.918
Generation: 250. 正答率(トレーニング) = 0.9512
                : 250. 正答率(テスト) = 0.922
Generation: 260. 正答率(トレーニング) = 0.9496
                : 260. 正答率(テスト) = 0.92
Generation: 270. 正答率(トレーニング) = 0.948
                : 270. 正答率(テスト) = 0.917
Generation: 280. 正答率(トレーニング) = 0.9498
                : 280. 正答率(テスト) = 0.922
Generation: 290. 正答率(トレーニング) = 0.9538
                : 290. 正答率(テスト) = 0.922
Generation: 300. 正答率(トレーニング) = 0.949
                : 300. 正答率(テスト) = 0.916
Generation: 310. 正答率(トレーニング) = 0.9552
                : 310. 正答率(テスト) = 0.925
Generation: 320. 正答率(トレーニング) = 0.9622
                : 320. 正答率(テスト) = 0.932
Generation: 330. 正答率(トレーニング) = 0.9596
                : 330. 正答率(テスト) = 0.94
Generation: 340. 正答率(トレーニング) = 0.96
                : 340. 正答率(テスト) = 0.924
Generation: 350. 正答率(トレーニング) = 0.9576
                : 350. 正答率(テスト) = 0.931
Generation: 360. 正答率(トレーニング) = 0.9682
                : 360. 正答率(テスト) = 0.941
Generation: 370. 正答率(トレーニング) = 0.9682
                : 370. 正答率(テスト) = 0.945
Generation: 380. 正答率(トレーニング) = 0.97
                : 380. 正答率(テスト) = 0.939
Generation: 390. 正答率(トレーニング) = 0.9658
                : 390. 正答率(テスト) = 0.932
Generation: 400. 正答率(トレーニング) = 0.9662
```

```
                : 400. 正答率(テスト) = 0.935
Generation: 410. 正答率(トレーニング) = 0.9696
                : 410. 正答率(テスト) = 0.95
Generation: 420. 正答率(トレーニング) = 0.974
                : 420. 正答率(テスト) = 0.944
Generation: 430. 正答率(トレーニング) = 0.9738
                : 430. 正答率(テスト) = 0.946
Generation: 440. 正答率(トレーニング) = 0.9746
                : 440. 正答率(テスト) = 0.945
Generation: 450. 正答率(トレーニング) = 0.9728
                : 450. 正答率(テスト) = 0.941
Generation: 460. 正答率(トレーニング) = 0.9756
                : 460. 正答率(テスト) = 0.954
Generation: 470. 正答率(トレーニング) = 0.9738
                : 470. 正答率(テスト) = 0.945
Generation: 480. 正答率(トレーニング) = 0.9786
                : 480. 正答率(テスト) = 0.951
Generation: 490. 正答率(トレーニング) = 0.9776
                : 490. 正答率(テスト) = 0.946
Generation: 500. 正答率(トレーニング) = 0.9776
                : 500. 正答率(テスト) = 0.954
Generation: 510. 正答率(トレーニング) = 0.9764
                : 510. 正答率(テスト) = 0.948
Generation: 520. 正答率(トレーニング) = 0.9758
                : 520. 正答率(テスト) = 0.952
Generation: 530. 正答率(トレーニング) = 0.9722
                : 530. 正答率(テスト) = 0.941
Generation: 540. 正答率(トレーニング) = 0.982
                : 540. 正答率(テスト) = 0.954
Generation: 550. 正答率(トレーニング) = 0.9794
                : 550. 正答率(テスト) = 0.957
Generation: 560. 正答率(トレーニング) = 0.972
                : 560. 正答率(テスト) = 0.938
Generation: 570. 正答率(トレーニング) = 0.9768
                : 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) = 0.954
Generation: 590. 正答率(トレーニング) = 0.9832
                : 590. 正答率(テスト) = 0.959
Generation: 600. 正答率(トレーニング) = 0.9828
                : 600. 正答率(テスト) = 0.955
Generation: 610. 正答率(トレーニング) = 0.985
                : 610. 正答率(テスト) = 0.956
Generation: 620. 正答率(トレーニング) = 0.9844
                : 620. 正答率(テスト) = 0.952
Generation: 630. 正答率(トレーニング) = 0.9844
                : 630. 正答率(テスト) = 0.961
Generation: 640. 正答率(トレーニング) = 0.9852
                : 640. 正答率(テスト) = 0.954
Generation: 650. 正答率(トレーニング) = 0.9856
                : 650. 正答率(テスト) = 0.96
Generation: 660. 正答率(トレーニング) = 0.9822
                : 660. 正答率(テスト) = 0.95
Generation: 670. 正答率(トレーニング) = 0.9854
                : 670. 正答率(テスト) = 0.952
Generation: 680. 正答率(トレーニング) = 0.981
                : 680. 正答率(テスト) = 0.949
Generation: 690. 正答率(トレーニング) = 0.9854
                : 690. 正答率(テスト) = 0.956
Generation: 700. 正答率(トレーニング) = 0.9854
                : 700. 正答率(テスト) = 0.955
Generation: 710. 正答率(トレーニング) = 0.9864
                : 710. 正答率(テスト) = 0.956
Generation: 720. 正答率(トレーニング) = 0.9874
                : 720. 正答率(テスト) = 0.955
Generation: 730. 正答率(トレーニング) = 0.9878
                : 730. 正答率(テスト) = 0.959
Generation: 740. 正答率(トレーニング) = 0.9884
                : 740. 正答率(テスト) = 0.959
Generation: 750. 正答率(トレーニング) = 0.9868
                : 750. 正答率(テスト) = 0.957
Generation: 760. 正答率(トレーニング) = 0.987
                : 760. 正答率(テスト) = 0.95
Generation: 770. 正答率(トレーニング) = 0.9876
                : 770. 正答率(テスト) = 0.962
Generation: 780. 正答率(トレーニング) = 0.9872
                : 780. 正答率(テスト) = 0.954
Generation: 790. 正答率(トレーニング) = 0.9896
                : 790. 正答率(テスト) = 0.958
Generation: 800. 正答率(トレーニング) = 0.9902
```

```
                : 800. 正答率(テスト) = 0.959
Generation: 810. 正答率(トレーニング) = 0.9894
                : 810. 正答率(テスト) = 0.957
Generation: 820. 正答率(トレーニング) = 0.9892
                : 820. 正答率(テスト) = 0.956
Generation: 830. 正答率(トレーニング) = 0.9898
                : 830. 正答率(テスト) = 0.956
Generation: 840. 正答率(トレーニング) = 0.989
                : 840. 正答率(テスト) = 0.957
Generation: 850. 正答率(トレーニング) = 0.9912
                : 850. 正答率(テスト) = 0.955
Generation: 860. 正答率(トレーニング) = 0.9932
                : 860. 正答率(テスト) = 0.963
Generation: 870. 正答率(トレーニング) = 0.993
                : 870. 正答率(テスト) = 0.96
Generation: 880. 正答率(トレーニング) = 0.9914
                : 880. 正答率(テスト) = 0.959
Generation: 890. 正答率(トレーニング) = 0.993
                : 890. 正答率(テスト) = 0.961
Generation: 900. 正答率(トレーニング) = 0.9864
                : 900. 正答率(テスト) = 0.949
Generation: 910. 正答率(トレーニング) = 0.9932
                : 910. 正答率(テスト) = 0.959
Generation: 920. 正答率(トレーニング) = 0.9908
                : 920. 正答率(テスト) = 0.961
Generation: 930. 正答率(トレーニング) = 0.996
                : 930. 正答率(テスト) = 0.96
Generation: 940. 正答率(トレーニング) = 0.9948
                : 940. 正答率(テスト) = 0.957
Generation: 950. 正答率(トレーニング) = 0.9958
                : 950. 正答率(テスト) = 0.963
Generation: 960. 正答率(トレーニング) = 0.9952
                : 960. 正答率(テスト) = 0.962
Generation: 970. 正答率(トレーニング) = 0.9942
                : 970. 正答率(テスト) = 0.963
Generation: 980. 正答率(トレーニング) = 0.9926
                : 980. 正答率(テスト) = 0.961
Generation: 990. 正答率(トレーニング) = 0.9964
                : 990. 正答率(テスト) = 0.961
Generation: 1000. 正答率(トレーニング) = 0.9976
                : 1000. 正答率(テスト) = 0.961
```



In [40]:

```
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_p
#         conv_output_size_2 = (conv_output_size_1 - conv_param_2['filter_size'] + 2 *
        conv_output_size_2 = (conv_output_size_1 / 2 - conv_param_2['filter_size'] +
        pool_output_size = int(conv_param_2['filter_num'] * (conv_output_size_2 / 2)
        # 重みの初期化
        self.params = {}
```

```
        self.params['W1'] = weight_init_std * np.random.randn(conv_param_1['filter_nu
        self.params['b1'] = np.zeros(conv_param_1['filter_num'])
        self.params['W2'] = weight_init_std * np.random.randn(conv_param_2['filter_nu
        self.params['b2'] = np.zeros(conv_param_2['filter_num'])
        self.params['W3'] = weight_init_std * np.random.randn(pool_output_size, hidde
        self.params['b3'] = np.zeros(hidden_size)
        self.params['W4'] = weight_init_std * np.random.randn(hidden_size, output_siz
        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'
        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'
        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)
#             print(key)
        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
```

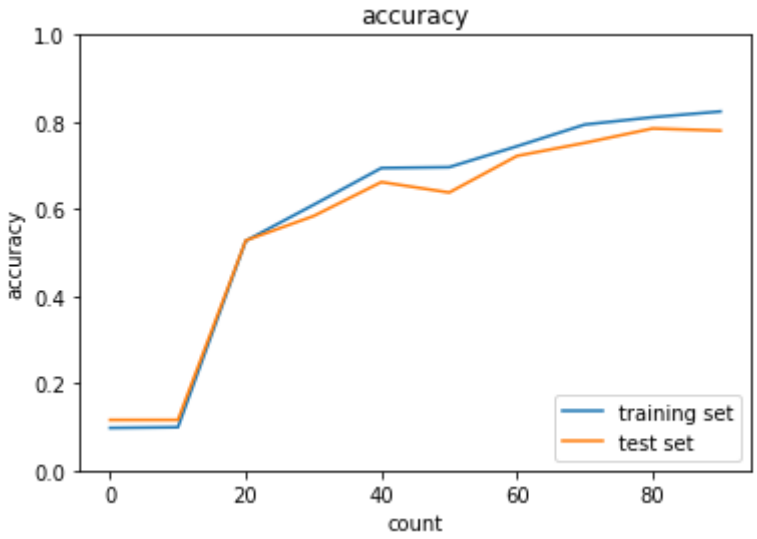
```
        grad['W4'], grad['b4'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

        return grad
```

In [41]:

```
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")
start = time.time()
# 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, 'st
                        conv_param_2={'filter_num':20, 'filter_size':3, 'pad':1, 'st
                        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)

        process_time = time.time() - start
        print(process_time)
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_trai
        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
8.877216815948486
Generation: 10. 正答率(トレーニング) = 0.0976
                : 10. 正答率(テスト) = 0.116
17.647892236709595
Generation: 20. 正答率(トレーニング) = 0.0992
                : 20. 正答率(テスト) = 0.116
27.017614364624023
Generation: 30. 正答率(トレーニング) = 0.527
                : 30. 正答率(テスト) = 0.528
34.85488724708557
Generation: 40. 正答率(トレーニング) = 0.6098
                : 40. 正答率(テスト) = 0.584
42.806095123291016
Generation: 50. 正答率(トレーニング) = 0.6942
                : 50. 正答率(テスト) = 0.662
50.64944839477539
Generation: 60. 正答率(トレーニング) = 0.6962
                : 60. 正答率(テスト) = 0.638
58.88264727592468
Generation: 70. 正答率(トレーニング) = 0.7442
                : 70. 正答率(テスト) = 0.722
67.79273748397827
Generation: 80. 正答率(トレーニング) = 0.794
                : 80. 正答率(テスト) = 0.752
76.91342759132385
Generation: 90. 正答率(トレーニング) = 0.8108
                : 90. 正答率(テスト) = 0.785
86.37917971611023
Generation: 100. 正答率(トレーニング) = 0.824
                 : 100. 正答率(テスト) = 0.78
```



CNNの実装を試みた。層の仕組みを勉強するとともに、２層畳み込み層があるモデルにおいて、プーリング層を一つ消した場合の比較を行った。
実際には重みの初期値が違うということも考慮にいれなくてはいけないかもしれないが、プーリング層抜きだと154秒で正答率（テスト）が0.85に対し、プーリング層ありだと86秒で正答率（テスト）が0.78であった。
今回は同じイテレーション回数で比較したが、同じ秒数で比較した場合、どちらに優劣がつくかわからない。少なくとも、プーリング層があるせいで決定的に精度が落ちることはなく、解析時間をかなり短縮できることを確認できた。