⑤最新のCNN

=====

目次:

1.層の深い畳み込みニューラルネットワーク(deep convolution network)

【要約】

・CNNとして、認識精度の高いモデルがいくつも公表されている

近年の例:AlexNet(2012年)、VGG(2015年)

・認識精度を高めるため、層(畳み込み層・プーリング層)を何層も重ねて深くしたモデルとなっている

・過学習を防ぐために、ドロップアウトを利用(全結合層及び出力層で使用)

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.層の深い畳み込みニューラルネットワーク(deep convolution network)

・層の深い畳み込みネットワークの実装(クラス)

In [2]:

```
import pickle
import numpy as np
from collections import OrderedDict
from common import layers
from data.mnist import load_mnist
import matplotlib.pyplot as plt
from common import optimizer

class DeepConvNet:
    '''
    認識率99%以上の高精度なConvNet

    conv - relu - conv- relu - pool -
    conv - relu - conv- relu - pool -
    conv - relu - conv- relu - pool -
    affine - relu - dropout - affine - dropout - softmax
    '''
    def __init__(self, input_dim=(1, 28, 28),
                 conv_param_1 = {'filter_num':16, 'filter_size':3, 'pad':1, 'stride':1},
                 conv_param_2 = {'filter_num':16, 'filter_size':3, 'pad':1, 'stride':1},
                 conv_param_3 = {'filter_num':32, 'filter_size':3, 'pad':1, 'stride':1},
                 conv_param_4 = {'filter_num':32, 'filter_size':3, 'pad':2, 'stride':1},
                 conv_param_5 = {'filter_num':64, 'filter_size':3, 'pad':1, 'stride':1},
                 conv_param_6 = {'filter_num':64, 'filter_size':3, 'pad':1, 'stride':1},
                 hidden_size=50, output_size=10):
        # 重みの初期化===========
        # 各層のニューロンひとつあたりが、前層のニューロンといくつのつながりがあるか
        pre_node_nums = np.array([1*3*3, 16*3*3, 16*3*3, 32*3*3, 32*3*3, 64*3*3, 64*4*4, hidden_
size])
        wight_init_scales = np.sqrt(2.0 / pre_node_nums)  # Heの初期値

        self.params = {}
        pre_channel_num = input_dim[0]
        for idx, conv_param in enumerate([conv_param_1, conv_param_2, conv_param_3, conv_param_
4, conv_param_5, conv_param_6]):
            self.params['W' + str(idx+1)] = wight_init_scales[idx] * np.random.randn(conv_param[
'filter_num'], pre_channel_num, conv_param['filter_size'], conv_param['filter_size'])
            self.params['b' + str(idx+1)] = np.zeros(conv_param['filter_num'])
            pre_channel_num = conv_param['filter_num']
        self.params['W7'] = wight_init_scales[6] * np.random.randn(pre_node_nums[6], hidden_size
)
        print(self.params['W7'].shape)
        self.params['b7'] = np.zeros(hidden_size)
        self.params['W8'] = wight_init_scales[7] * np.random.randn(pre_node_nums[7], output_size
)
        self.params['b8'] = np.zeros(output_size)

        # レイヤの生成===========
        self.layers = []
        self.layers.append(layers.Convolution(self.params['W1'], self.params['b1'],
                           conv_param_1['stride'], conv_param_1['pad']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Convolution(self.params['W2'], self.params['b2'],
                           conv_param_2['stride'], conv_param_2['pad']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(layers.Convolution(self.params['W3'], self.params['b3'],
                           conv_param_3['stride'], conv_param_3['pad']))
        self.layers.append(layers.Relu())
```

```
        self.layers.append(layers.Convolution(self.params['W4'], self.params['b4'],
                           conv_param_4['stride'], conv_param_4['pad']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(layers.Convolution(self.params['W5'], self.params['b5'],
                           conv_param_5['stride'], conv_param_5['pad']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Convolution(self.params['W6'], self.params['b6'],
                           conv_param_6['stride'], conv_param_6['pad']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(layers.Affine(self.params['W7'], self.params['b7']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Dropout(0.5))
        self.layers.append(layers.Affine(self.params['W8'], self.params['b8']))
        self.layers.append(layers.Dropout(0.5))

        self.last_layer = layers.SoftmaxWithLoss()

    def predict(self, x, train_flg=False):
        for layer in self.layers:
            if isinstance(layer, layers.Dropout):
                x = layer.forward(x, train_flg)
            else:
                x = layer.forward(x)
        return x

    def loss(self, x, d):
        y = self.predict(x, train_flg=True)
        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, train_flg=False)
            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)

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

        # 設定
        grads = {}
        for i, layer_idx in enumerate((0, 2, 5, 7, 10, 12, 15, 18)):
```

```
            grads['W' + str(i+1)] = self.layers[layer_idx].dW
            grads['b' + str(i+1)] = self.layers[layer_idx].db

        return grads
```

→順伝播の順番:

```
conv - relu - conv- relu - pool -
conv - relu - conv- relu - pool -
conv - relu - conv- relu - pool -
affine - relu - dropout - affine - dropout - softmax
```

[内訳]　合計11層(入力層除く)

畳み込み層(2つ):conv - relu - conv- relu

プーリング層:pool

畳み込み層(2つ):conv - relu - conv- relu

プーリング層:pool

畳み込み層(2つ):conv - relu - conv- relu

プーリング層:pool

全結合層:affine - relu - dropout

出力層:affine - dropout - softmax

・層の深い畳み込みネットワークでの学習及び予測

In [3]:

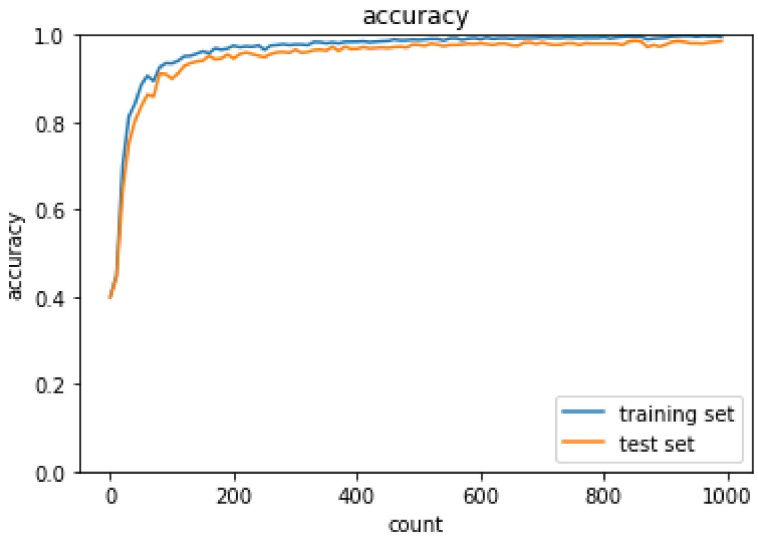
```
(x_train, d_train), (x_test, d_test) = load_mnist(flatten=False)

# 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]

print("データ読み込み完了")

network = DeepConvNet()
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
(1024, 50)
Generation: 10. 正答率(トレーニング) = 0.4004
                : 10. 正答率(テスト) = 0.398
Generation: 20. 正答率(トレーニング) = 0.4484
                : 20. 正答率(テスト) = 0.445
Generation: 30. 正答率(トレーニング) = 0.7016
                : 30. 正答率(テスト) = 0.644
Generation: 40. 正答率(トレーニング) = 0.8132
                : 40. 正答率(テスト) = 0.752
Generation: 50. 正答率(トレーニング) = 0.8424
                : 50. 正答率(テスト) = 0.803
Generation: 60. 正答率(トレーニング) = 0.885
                : 60. 正答率(テスト) = 0.836
Generation: 70. 正答率(トレーニング) = 0.9062
                : 70. 正答率(テスト) = 0.863
Generation: 80. 正答率(トレーニング) = 0.8944
                : 80. 正答率(テスト) = 0.859
Generation: 90. 正答率(トレーニング) = 0.9258
                : 90. 正答率(テスト) = 0.911
Generation: 100. 正答率(トレーニング) = 0.9352
                : 100. 正答率(テスト) = 0.91
Generation: 110. 正答率(トレーニング) = 0.9344
                : 110. 正答率(テスト) = 0.899
Generation: 120. 正答率(トレーニング) = 0.9408
                : 120. 正答率(テスト) = 0.912
Generation: 130. 正答率(トレーニング) = 0.951
                : 130. 正答率(テスト) = 0.929
Generation: 140. 正答率(トレーニング) = 0.9518
                : 140. 正答率(テスト) = 0.935
Generation: 150. 正答率(トレーニング) = 0.957
                : 150. 正答率(テスト) = 0.939
Generation: 160. 正答率(トレーニング) = 0.9624
                : 160. 正答率(テスト) = 0.941
Generation: 170. 正答率(トレーニング) = 0.9582
                : 170. 正答率(テスト) = 0.952
Generation: 180. 正答率(トレーニング) = 0.9694
                : 180. 正答率(テスト) = 0.944
Generation: 190. 正答率(トレーニング) = 0.966
                : 190. 正答率(テスト) = 0.946
Generation: 200. 正答率(トレーニング) = 0.9688
                : 200. 正答率(テスト) = 0.955
Generation: 210. 正答率(トレーニング) = 0.9752
                : 210. 正答率(テスト) = 0.946
Generation: 220. 正答率(トレーニング) = 0.9716
                : 220. 正答率(テスト) = 0.956
Generation: 230. 正答率(トレーニング) = 0.9738
                : 230. 正答率(テスト) = 0.96
Generation: 240. 正答率(トレーニング) = 0.9726
                : 240. 正答率(テスト) = 0.956
Generation: 250. 正答率(トレーニング) = 0.976
                : 250. 正答率(テスト) = 0.952
Generation: 260. 正答率(トレーニング) = 0.9658
                : 260. 正答率(テスト) = 0.949
Generation: 270. 正答率(トレーニング) = 0.9756
                : 270. 正答率(テスト) = 0.956
Generation: 280. 正答率(トレーニング) = 0.9768
                : 280. 正答率(テスト) = 0.96
Generation: 290. 正答率(トレーニング) = 0.9788
                : 290. 正答率(テスト) = 0.961
Generation: 300. 正答率(トレーニング) = 0.9766
```

```
                  : 300. 正答率(テスト) = 0.959
Generation: 310. 正答率(トレーニング) = 0.9782
                  : 310. 正答率(テスト) = 0.966
Generation: 320. 正答率(トレーニング) = 0.978
                  : 320. 正答率(テスト) = 0.959
Generation: 330. 正答率(トレーニング) = 0.9766
                  : 330. 正答率(テスト) = 0.961
Generation: 340. 正答率(トレーニング) = 0.9838
                  : 340. 正答率(テスト) = 0.965
Generation: 350. 正答率(トレーニング) = 0.9832
                  : 350. 正答率(テスト) = 0.966
Generation: 360. 正答率(トレーニング) = 0.9808
                  : 360. 正答率(テスト) = 0.964
Generation: 370. 正答率(トレーニング) = 0.983
                  : 370. 正答率(テスト) = 0.973
Generation: 380. 正答率(トレーニング) = 0.9806
                  : 380. 正答率(テスト) = 0.963
Generation: 390. 正答率(トレーニング) = 0.984
                  : 390. 正答率(テスト) = 0.973
Generation: 400. 正答率(トレーニング) = 0.9842
                  : 400. 正答率(テスト) = 0.968
Generation: 410. 正答率(トレーニング) = 0.9844
                  : 410. 正答率(テスト) = 0.968
Generation: 420. 正答率(トレーニング) = 0.9854
                  : 420. 正答率(テスト) = 0.972
Generation: 430. 正答率(トレーニング) = 0.9832
                  : 430. 正答率(テスト) = 0.969
Generation: 440. 正答率(トレーニング) = 0.9844
                  : 440. 正答率(テスト) = 0.97
Generation: 450. 正答率(トレーニング) = 0.9856
                  : 450. 正答率(テスト) = 0.971
Generation: 460. 正答率(トレーニング) = 0.9862
                  : 460. 正答率(テスト) = 0.97
Generation: 470. 正答率(トレーニング) = 0.9894
                  : 470. 正答率(テスト) = 0.972
Generation: 480. 正答率(トレーニング) = 0.9874
                  : 480. 正答率(テスト) = 0.973
Generation: 490. 正答率(トレーニング) = 0.9878
                  : 490. 正答率(テスト) = 0.971
Generation: 500. 正答率(トレーニング) = 0.989
                  : 500. 正答率(テスト) = 0.978
Generation: 510. 正答率(トレーニング) = 0.9884
                  : 510. 正答率(テスト) = 0.977
Generation: 520. 正答率(トレーニング) = 0.9894
                  : 520. 正答率(テスト) = 0.975
Generation: 530. 正答率(トレーニング) = 0.9906
                  : 530. 正答率(テスト) = 0.98
Generation: 540. 正答率(トレーニング) = 0.9904
                  : 540. 正答率(テスト) = 0.978
Generation: 550. 正答率(トレーニング) = 0.987
                  : 550. 正答率(テスト) = 0.974
Generation: 560. 正答率(トレーニング) = 0.9918
                  : 560. 正答率(テスト) = 0.977
Generation: 570. 正答率(トレーニング) = 0.9924
                  : 570. 正答率(テスト) = 0.977
Generation: 580. 正答率(トレーニング) = 0.9882
                  : 580. 正答率(テスト) = 0.978
Generation: 590. 正答率(トレーニング) = 0.9914
                  : 590. 正答率(テスト) = 0.98
Generation: 600. 正答率(トレーニング) = 0.9926
                  : 600. 正答率(テスト) = 0.979
```

```
Generation: 610. 正答率(トレーニング) = 0.99
                : 610. 正答率(テスト) = 0.981
Generation: 620. 正答率(トレーニング) = 0.9936
                : 620. 正答率(テスト) = 0.979
Generation: 630. 正答率(トレーニング) = 0.9908
                : 630. 正答率(テスト) = 0.977
Generation: 640. 正答率(トレーニング) = 0.9922
                : 640. 正答率(テスト) = 0.98
Generation: 650. 正答率(トレーニング) = 0.9924
                : 650. 正答率(テスト) = 0.98
Generation: 660. 正答率(トレーニング) = 0.9912
                : 660. 正答率(テスト) = 0.976
Generation: 670. 正答率(トレーニング) = 0.9926
                : 670. 正答率(テスト) = 0.975
Generation: 680. 正答率(トレーニング) = 0.9924
                : 680. 正答率(テスト) = 0.982
Generation: 690. 正答率(トレーニング) = 0.9928
                : 690. 正答率(テスト) = 0.983
Generation: 700. 正答率(トレーニング) = 0.993
                : 700. 正答率(テスト) = 0.979
Generation: 710. 正答率(トレーニング) = 0.9944
                : 710. 正答率(テスト) = 0.983
Generation: 720. 正答率(トレーニング) = 0.9936
                : 720. 正答率(テスト) = 0.979
Generation: 730. 正答率(トレーニング) = 0.9928
                : 730. 正答率(テスト) = 0.977
Generation: 740. 正答率(トレーニング) = 0.9928
                : 740. 正答率(テスト) = 0.978
Generation: 750. 正答率(トレーニング) = 0.994
                : 750. 正答率(テスト) = 0.981
Generation: 760. 正答率(トレーニング) = 0.9934
                : 760. 正答率(テスト) = 0.981
Generation: 770. 正答率(トレーニング) = 0.9926
                : 770. 正答率(テスト) = 0.977
Generation: 780. 正答率(トレーニング) = 0.9932
                : 780. 正答率(テスト) = 0.981
Generation: 790. 正答率(トレーニング) = 0.993
                : 790. 正答率(テスト) = 0.98
Generation: 800. 正答率(トレーニング) = 0.993
                : 800. 正答率(テスト) = 0.98
Generation: 810. 正答率(トレーニング) = 0.9948
                : 810. 正答率(テスト) = 0.98
Generation: 820. 正答率(トレーニング) = 0.9918
                : 820. 正答率(テスト) = 0.98
Generation: 830. 正答率(トレーニング) = 0.994
                : 830. 正答率(テスト) = 0.98
Generation: 840. 正答率(トレーニング) = 0.9956
                : 840. 正答率(テスト) = 0.977
Generation: 850. 正答率(トレーニング) = 0.9962
                : 850. 正答率(テスト) = 0.985
Generation: 860. 正答率(トレーニング) = 0.996
                : 860. 正答率(テスト) = 0.987
Generation: 870. 正答率(トレーニング) = 0.9956
                : 870. 正答率(テスト) = 0.985
Generation: 880. 正答率(トレーニング) = 0.9896
                : 880. 正答率(テスト) = 0.972
Generation: 890. 正答率(トレーニング) = 0.9916
                : 890. 正答率(テスト) = 0.977
Generation: 900. 正答率(トレーニング) = 0.9926
                : 900. 正答率(テスト) = 0.973
Generation: 910. 正答率(トレーニング) = 0.9934
```

```
                : 910. 正答率(テスト) = 0.978
Generation: 920. 正答率(トレーニング) = 0.9964
                : 920. 正答率(テスト) = 0.984
Generation: 930. 正答率(トレーニング) = 0.9972
                : 930. 正答率(テスト) = 0.986
Generation: 940. 正答率(トレーニング) = 0.9976
                : 940. 正答率(テスト) = 0.984
Generation: 950. 正答率(トレーニング) = 0.9976
                : 950. 正答率(テスト) = 0.981
Generation: 960. 正答率(トレーニング) = 0.9958
                : 960. 正答率(テスト) = 0.981
Generation: 970. 正答率(トレーニング) = 0.9976
                : 970. 正答率(テスト) = 0.98
Generation: 980. 正答率(トレーニング) = 0.9966
                : 980. 正答率(テスト) = 0.983
Generation: 990. 正答率(トレーニング) = 0.997
                : 990. 正答率(テスト) = 0.984
Generation: 1000. 正答率(トレーニング) = 0.9958
                : 1000. 正答率(テスト) = 0.986
```



→高精度な認識率を達成

In [ ]: