Section 5 最近のCNN

In [ ]:

```

```

In [5]:

```
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
import time
```

In [3]:

```
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*
        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, c
            self.params['W' + str(idx+1)] = wight_init_scales[idx] * np.random.randn(
            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],
        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],
        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']))
```

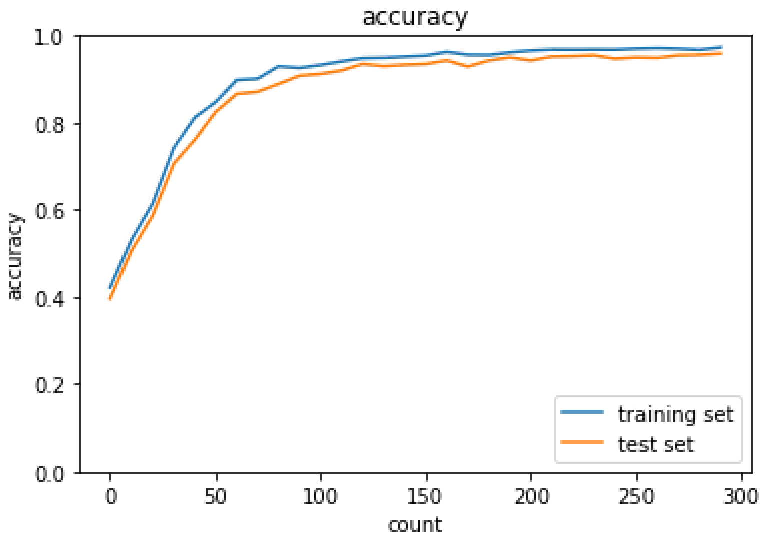
```
        self.layers.append(layers.Relu())
        self.layers.append(layers.Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(layers.Convolution(self.params['W5'], self.params['b5'],
                                              conv_param_5['stride'], conv_param_5['pad']))
        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
```

```
In [12]: from common import optimizer

(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  解析開始")
start = time.time()

network = DeepConvNet()
optimizer = optimizer.Adam()

iters_num = 300
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  解析開始
(1024, 50)
71.18267250061035
Generation: 10. 正答率(トレーニング) = 0.4216
                 : 10. 正答率(テスト) = 0.396
137.20938205718994
Generation: 20. 正答率(トレーニング) = 0.532
```

```
                : 20. 正答率(テスト) = 0.507
203.4637713432312
Generation: 30. 正答率(トレーニング) = 0.6148
                : 30. 正答率(テスト) = 0.587
273.37864661216736
Generation: 40. 正答率(トレーニング) = 0.7422
                : 40. 正答率(テスト) = 0.706
340.0337264537811
Generation: 50. 正答率(トレーニング) = 0.8116
                : 50. 正答率(テスト) = 0.761
406.65348076820374
Generation: 60. 正答率(トレーニング) = 0.8474
                : 60. 正答率(テスト) = 0.824
473.4613709449768
Generation: 70. 正答率(トレーニング) = 0.898
                : 70. 正答率(テスト) = 0.866
539.8696601390839
Generation: 80. 正答率(トレーニング) = 0.901
                : 80. 正答率(テスト) = 0.871
606.5378136634827
Generation: 90. 正答率(トレーニング) = 0.9296
                : 90. 正答率(テスト) = 0.889
672.9677352905273
Generation: 100. 正答率(トレーニング) = 0.9258
                : 100. 正答率(テスト) = 0.908
739.1366457939148
Generation: 110. 正答率(トレーニング) = 0.9326
                : 110. 正答率(テスト) = 0.912
805.5427045822144
Generation: 120. 正答率(トレーニング) = 0.9408
                : 120. 正答率(テスト) = 0.92
872.9223501682281
Generation: 130. 正答率(トレーニング) = 0.9484
                : 130. 正答率(テスト) = 0.935
939.0733180046082
Generation: 140. 正答率(トレーニング) = 0.9494
                : 140. 正答率(テスト) = 0.93
1005.5923912525177
Generation: 150. 正答率(トレーニング) = 0.9518
                : 150. 正答率(テスト) = 0.933
1071.9495239257812
Generation: 160. 正答率(トレーニング) = 0.9542
                : 160. 正答率(テスト) = 0.935
1137.995910167694
Generation: 170. 正答率(トレーニング) = 0.9628
                : 170. 正答率(テスト) = 0.943
1204.2031939029694
Generation: 180. 正答率(トレーニング) = 0.9562
                : 180. 正答率(テスト) = 0.929
1270.436176776886
Generation: 190. 正答率(トレーニング) = 0.9556
                : 190. 正答率(テスト) = 0.943
1336.7942032814026
Generation: 200. 正答率(トレーニング) = 0.9616
                : 200. 正答率(テスト) = 0.95
1402.9214317798615
Generation: 210. 正答率(トレーニング) = 0.966
                : 210. 正答率(テスト) = 0.943
1469.4618694782257
Generation: 220. 正答率(トレーニング) = 0.9684
                : 220. 正答率(テスト) = 0.952
1535.1021420955658
Generation: 230. 正答率(トレーニング) = 0.9682
                : 230. 正答率(テスト) = 0.953
1600.81880402565
Generation: 240. 正答率(トレーニング) = 0.9684
                : 240. 正答率(テスト) = 0.955
1668.5313398838043
Generation: 250. 正答率(トレーニング) = 0.9682
                : 250. 正答率(テスト) = 0.947
1739.4128260612488
Generation: 260. 正答率(トレーニング) = 0.9702
                : 260. 正答率(テスト) = 0.95
1805.298320531845
Generation: 270. 正答率(トレーニング) = 0.9714
                : 270. 正答率(テスト) = 0.949
1877.8751652240753
Generation: 280. 正答率(トレーニング) = 0.97
                : 280. 正答率(テスト) = 0.955
1944.0977573394775
```

```
Generation: 290. 正答率(トレーニング) = 0.9678
                : 290. 正答率(テスト) = 0.956
2010.1307580471039
Generation: 300. 正答率(トレーニング) = 0.973
                : 300. 正答率(テスト) = 0.959
```



深層CNNのさわりを試みた。今回のモデルでも、実際現役で使われるようなモデルに対して学習データ数もそんなに多くない上に、層もとても厚いわけではないが、それでもかなりの時間を要した。

NN系の開発に、GPUを使ったり、バッチ学習を用いたりした理由を肌感的に体験できた。