```
# coding: utf-8
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
import pickle
import numpy as np
from collections import OrderedDict
from common.layers import *
from common.gradient import numerical_gradient


class SimpleConvNet:
    """简单的 ConvNet

    conv - relu - pool - affine - relu - affine - softmax

    Parameters
    ----------
    input_size : 输入大小（对于 MNIST 为 784）
    hidden_size_list : 隐层神经元数量列表（e.g.[100,100,100]）
    output_size : 输出大小（10 表示 MNIST）
    activation : 'relu' or 'sigmoid'
    weight_init_std : 指定权重的标准差(e.g.0.01)
        'relu'或'he'时设置“He 初始值”
        'sigmoid'或“xavier”时设置“初始 Xavier”
    """
    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], filter_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)

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

        self.last_layer = SoftmaxWithLoss()

    def predict(self, x):#将除 last_layer 外的所有层全部执行一次得到结果 x
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

    def loss(self, x, t):
        """
            求损失函数
          参数 x 是输入数据，t 是教师标签
        """
        y = self.predict(x)
        return self.last_layer.forward(y, t)

    def accuracy(self, x, t, batch_size=100):
        if t.ndim != 1 : t = np.argmax(t, axis=1)#如果维数！=1 就将维数变为 1

        acc = 0.0

        for i in range(int(x.shape[0] / batch_size)):
            tx = x[i*batch_size:(i+1)*batch_size]#切片切出 batch_size 个数据，在本例中为 100
            tt = t[i*batch_size:(i+1)*batch_size]
            y = self.predict(tx)
            y = np.argmax(y, axis=1)#取出数组中的最大值的索引
            acc += np.sum(y == tt)#将回答正确的概率作为识别精度

        return acc / x.shape[0]
```

```
def numerical_gradient(self, x, t):
    """求梯度（数值微分）

    Parameters
    ----------
    x : 输入数据
    t : 教师标签
    Returns
    -------
    具有每层梯度的字典变量
    grads['W1']，grads['W2']，...是各层的权重
    grads['b1]，grads['b2]，...是每个层的偏差
    """
    loss_w = lambda w: self.loss(x, t)

    grads = {}
    for idx in (1, 2, 3):
        grads['W' + str(idx)] = numerical_gradient(loss_w, self.params['W' + str(idx)])
        grads['b' + str(idx)] = numerical_gradient(loss_w, self.params['b' + str(idx)])

    return grads

def gradient(self, x, t):
    """求梯度（误差反向传播法）

    Parameters
    ----------
    x : 输入数据
    t : 教师标签


    Returns
    -------
    具有每层梯度的字典变量
    grads['W1']，grads['W2']，...是各层的权重
    grads['b1]，grads['b2]，...是每个层的偏差
    """
    # forward
    self.loss(x, t)

    # backward
    dout = 1
    dout = self.last_layer.backward(dout)
```

```
        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

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

        return grads

    def save_params(self, file_name="params.pkl"):
        params = {}
        for key, val in self.params.items():
            params[key] = val
        with open(file_name, 'wb') as f:
            pickle.dump(params, f)

    def load_params(self, file_name="params.pkl"):
        with open(file_name, 'rb') as f:
            params = pickle.load(f)
        for key, val in params.items():
            self.params[key] = val

        for i, key in enumerate(['Conv1', 'Affine1', 'Affine2']):
            self.layers[key].W = self.params['W' + str(i+1)]
            self.layers[key].b = self.params['b' + str(i+1)]
```