2_6_simple_convolution_network

August 16, 2020

1 simple convolution network

1.1 image to column

[1]:
```
import sys, os
sys.path.append(os.pardir)
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

#
'''
input_data:
filter_h:
filter_w:
stride:
pad:
'''
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, filter_w, out_h, out_w, C, filter_h)

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

## [try] im2col	transpose	input_data

[2]:
```
# im2col
input_data = np.random.rand(2, 1, 4, 4)*100//1 # number, channel, height,␣
↪width
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 ==========
 [[[[19. 87.  7. 27.]
   [78. 17. 43. 20.]
   [86. 12. 12. 62.]
   [81. 66. 72. 61.]]]


 [[[24. 52. 36. 10.]
   [90. 90. 39.  2.]
   [44. 77. 89. 66.]
   [71. 68. 11. 25.]]]]
================================
============= col ==============
 [[19. 87.  7. 78. 17. 43. 86. 12. 12.]
 [87.  7. 27. 17. 43. 20. 12. 12. 62.]
 [78. 17. 43. 86. 12. 12. 81. 66. 72.]
 [17. 43. 20. 12. 12. 62. 66. 72. 61.]
 [24. 52. 36. 90. 90. 39. 44. 77. 89.]
 [52. 36. 10. 90. 39.  2. 77. 89. 66.]
 [90. 90. 39. 44. 77. 89. 71. 68. 11.]
 [90. 39.  2. 77. 89. 66. 68. 11. 25.]]
================================
```

1.2 column to image

[3]:
```
#
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
    #
    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]
```

1.3 col2im

im2col　　col image

[ ]:
```

```

1.4 convolution class

[4]:
```
class Convolution:
    # W:    , b:
    def __init__(self, W, b, stride=1, pad=0):
        self.W = W
        self.b = b
        self.stride = stride
        self.pad = pad

        #    backward
        self.x = None
        self.col = None
        self.col_W = None

        #
        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
        #   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
        col = im2col(x, FH, FW, self.stride, self.pad)
        #    x
        col_W = self.W.reshape(FN, -1).T

        out = np.dot(col, col_W) + self.b
        #
        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
        dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.pad)

        return dx
```

1.5 pooling class

[5]:
```
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
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        #
        col = col.reshape(-1, self.pool_h*self.pool_w)

        #
        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)
        #
        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
```

1.6 sinple convolution network class

[6]:
```
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, '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']
```

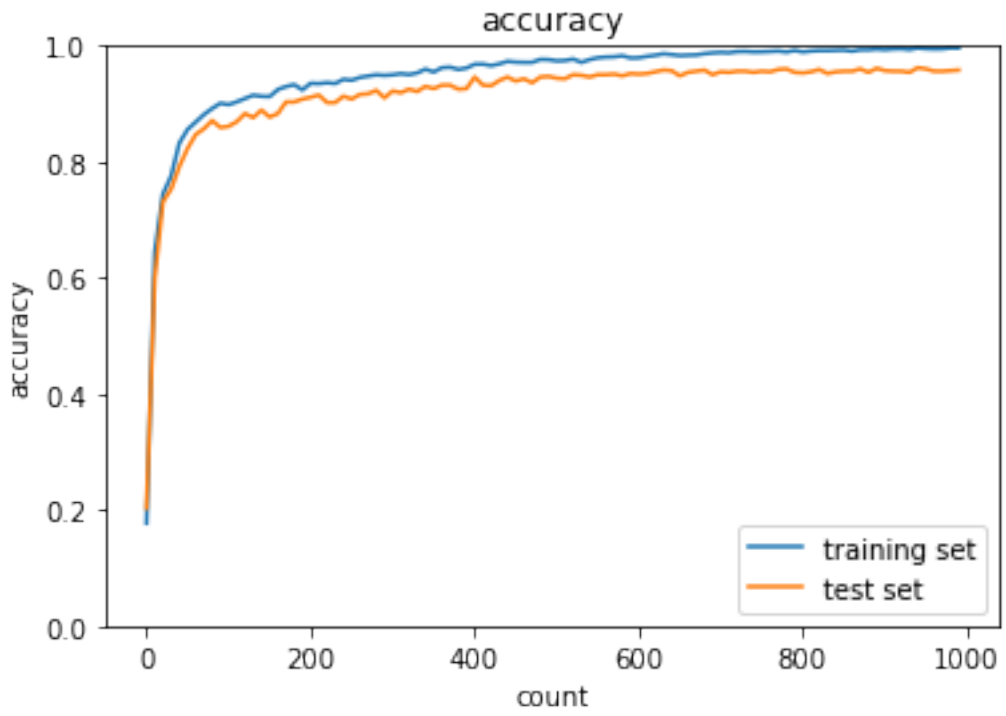
```
        filter_pad = conv_param['pad']
        filter_stride = conv_param['stride']
        input_size = input_dim[1]
        conv_output_size = (input_size - filter_size + 2 * filter_pad) /␣
↪filter_stride + 1
        pool_output_size = int(filter_num * (conv_output_size / 2) *␣
↪(conv_output_size / 2))

        #
        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)

        #
        self.layers = OrderedDict()
        self.layers['Conv1'] = layers.Convolution(self.params['W1'], self.
↪params['b1'], conv_param['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)

        #
        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
```

[9]:
```
from common import optimizer

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

print("     ")

#
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) + '.   (    ) = ' + str(accr_train))
#         print('                : ' + str(i+1) + '.   (  ) = ' +␣
↪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)
```

```
#
plt.show()
```


accuracy
1.0
0.8
0.6
0.4
0.2
0.0
accuracy
0
200
400
600
800
1000
count
training set
test set

[ ]: