six_layer_net.py

```
# coding: utf-8
import numpy as np
from common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict
from common.optimizer import Adam

class sixLayerNet:

    def __init__(self, input_size, hidden_size1, hidden_size2, hidden_size3,
hidden_size4, hidden_size5, output_size, weight_init_std=0.1):
        # 가중치 초기화
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size,
hidden_size1)
        self.params['b1'] = np.zeros(hidden_size1)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size1,
hidden_size2)
        self.params['b2'] = np.zeros(hidden_size2)
        self.params['W3'] = weight_init_std * np.random.randn(hidden_size2,
hidden_size3)
        self.params['b3'] = np.zeros(hidden_size3)
        self.params['W4'] = weight_init_std * np.random.randn(hidden_size3,
hidden_size4)
        self.params['b4'] = np.zeros(hidden_size4)
        self.params['W5'] = weight_init_std * np.random.randn(hidden_size4,
hidden_size5)
        self.params['b5'] = np.zeros(hidden_size5)
        self.params['W6'] = weight_init_std * np.random.randn(hidden_size5,
output_size)
        self.params['b6'] = np.zeros(output_size)

        # 정규화 파라미터
        self.gamma1 = np.ones(hidden_size1)
        self.beta1 = np.zeros(hidden_size1)
        self.gamma2 = np.ones(hidden_size2)
        self.beta2 = np.zeros(hidden_size2)
        self.gamma3 = np.ones(hidden_size3)
        self.beta3 = np.zeros(hidden_size3)
        self.gamma4 = np.ones(hidden_size4)
        self.beta4 = np.zeros(hidden_size4)
        self.gamma5 = np.ones(hidden_size5)
        self.beta5 = np.zeros(hidden_size5)

        # 계층 생성
        self.layers = OrderedDict()
        self.layers['Affine1'] = Affine(self.params['W1'], self.params['b1'])
```

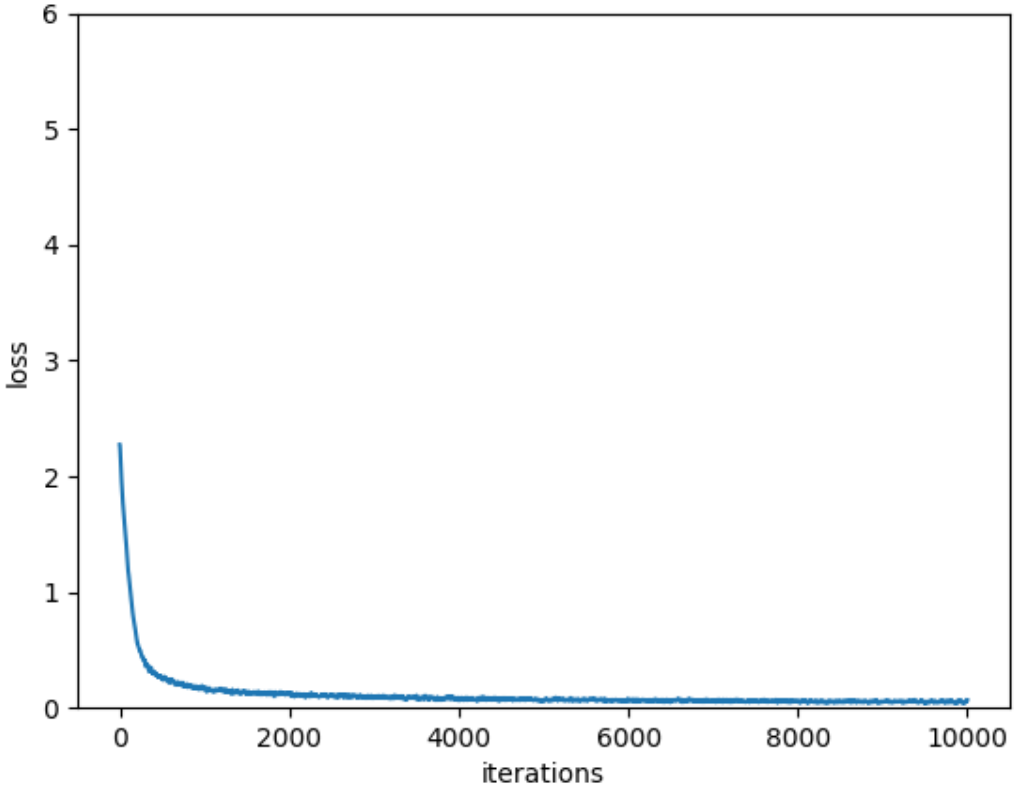
```
        self.layers['BatchNorm1'] = BatchNormalization(self.gamma1,
self.beta1)
        self.layers['Relu1'] = Relu()
        self.layers['Affine2'] = Affine(self.params['W2'], self.params['b2'])
        self.layers['BatchNorm2'] = BatchNormalization(self.gamma2,
self.beta2)
        self.layers['Relu2'] = Relu()
        self.layers['Affine3'] = Affine(self.params['W3'], self.params['b3'])
        self.layers['BatchNorm3'] = BatchNormalization(self.gamma3,
self.beta3)
        self.layers['Relu3'] = Relu()
        self.layers['Affine4'] = Affine(self.params['W4'], self.params['b4'])
        self.layers['BatchNorm4'] = BatchNormalization(self.gamma4,
self.beta4)
        self.layers['Relu4'] = Relu()
        self.layers['Affine5'] = Affine(self.params['W5'], self.params['b5'])
        self.layers['BatchNorm5'] = BatchNormalization(self.gamma5,
self.beta5)
        self.layers['Relu5'] = Relu()
        self.layers['Affine6'] = Affine(self.params['W6'], self.params['b6'])
        self.lastLayer = SoftmaxWithLoss()

        self.optimizer = Adam()

    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)
        return x

    def loss(self, x, t):
        y = self.predict(x)
        return self.lastLayer.forward(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if t.ndim != 1: t = np.argmax(t, axis=1)
        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)
        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])
        grads['W3'] = numerical_gradient(loss_W, self.params['W3'])
```

```
        grads['b3'] = numerical_gradient(loss_W, self.params['b3'])
        grads['W4'] = numerical_gradient(loss_W, self.params['W4'])
        grads['b4'] = numerical_gradient(loss_W, self.params['b4'])
        grads['W5'] = numerical_gradient(loss_W, self.params['W5'])
        grads['b5'] = numerical_gradient(loss_W, self.params['b5'])
        grads['W6'] = numerical_gradient(loss_W, self.params['W6'])
        grads['b6'] = numerical_gradient(loss_W, self.params['b6'])
        return grads

    def gradient(self, x, t):
        # forward
        self.loss(x, t)
        # backward
        dout = 1
        dout = self.lastLayer.backward(dout)
        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 결과 저장
        grads = {}
        grads['W1'], grads['b1'] = self.layers['Affine1'].dW,
self.layers['Affine1'].db
        grads['W2'], grads['b2'] = self.layers['Affine2'].dW,
self.layers['Affine2'].db
        grads['W3'], grads['b3'] = self.layers['Affine3'].dW,
self.layers['Affine3'].db
        grads['W4'], grads['b4'] = self.layers['Affine4'].dW,
self.layers['Affine4'].db
        grads['W5'], grads['b5'] = self.layers['Affine5'].dW,
self.layers['Affine5'].db
        grads['W6'], grads['b6'] = self.layers['Affine6'].dW,
self.layers['Affine6'].db
        grads['gamma1'], grads['beta1'] = self.layers['BatchNorm1'].dgamma,
self.layers['BatchNorm1'].dbeta
        grads['gamma2'], grads['beta2'] = self.layers['BatchNorm2'].dgamma,
self.layers['BatchNorm2'].dbeta
        grads['gamma3'], grads['beta3'] = self.layers['BatchNorm3'].dgamma,
self.layers['BatchNorm3'].dbeta
        grads['gamma4'], grads['beta4'] = self.layers['BatchNorm4'].dgamma,
self.layers['BatchNorm4'].dbeta
        grads['gamma5'], grads['beta5'] = self.layers['BatchNorm5'].dgamma,
self.layers['BatchNorm5'].dbeta
        self.optimizer.update(self.params, grads)
        return grads
```

train_neuralnet2.py

```
# coding: utf-8
import sys, os
sys.path.append(os.pardir)

import numpy as np
from dataset.mnist import load_mnist
#from two_layer_net import TwoLayerNet
from six_layer_net import sixLayerNet
import matplotlib.pyplot as plt
from common.util import smooth_curve

# 데이터 읽기
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True,
one_hot_label=True)

network = sixLayerNet(input_size=784, hidden_size1=15, hidden_size2=14,
hidden_size3=13, hidden_size4 = 12, hidden_size5 = 11, output_size=10)
iters_num = 10000
train_size = x_train.shape[0]
batch_size = 480
learning_rate = 0.019

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 기울기 계산
    grad = network.gradient(x_batch, t_batch)

    # 갱신
    for key in ('W1', 'b1', 'W2', 'b2', 'W3', 'b3', 'W4', 'b4', 'W5', 'b5',
'W6', 'b6'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
```

```
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc:", train_acc, "test acc:", test_acc)

max_iterations = len(train_loss_list)
x = np.arange(max_iterations)

plt.plot(x, smooth_curve(train_loss_list))
plt.xlabel("iterations")
plt.ylabel("loss")
plt.ylim(0, 6)
plt.show()
```



터미널 로그

```
PS C:₩Users₩장주훈₩Desktop₩대학교₩3학년₩인공지능개론 - 최인엽교수님₩deep-learning-from-scratch-master> & C:/Users/장주훈/AppData/Local/Programs/Python/Python38-32/python.exe "c:/Users/장주훈/Desktop/대학교/3학년/인공지능개론 - 최인엽교수님/deep-learning-from-scratch-master/ch05/train_neuralnet2.py"
train acc: 0.11435 test acc: 0.1196
train acc: 0.6131333333333333 test acc: 0.6126
train acc: 0.82955 test acc: 0.8329
```

train acc: 0.9252 test acc: 0.9243
train acc: 0.9391 test acc: 0.9346
train acc: 0.94585 test acc: 0.94
train acc: 0.9493166666666667 test acc: 0.9398
train acc: 0.9516333333333333 test acc: 0.9441
train acc: 0.9554333333333334 test acc: 0.947
train acc: 0.9578333333333333 test acc: 0.9472
train acc: 0.9593 test acc: 0.9501
train acc: 0.9626166666666667 test acc: 0.9509
train acc: 0.9635666666666667 test acc: 0.9513
train acc: 0.9641666666666666 test acc: 0.9519
train acc: 0.9662666666666667 test acc: 0.9507
train acc: 0.9666333333333333 test acc: 0.9501
train acc: 0.9670833333333333 test acc: 0.9519
train acc: 0.9688166666666667 test acc: 0.9507
train acc: 0.96875 test acc: 0.952
train acc: 0.9700666666666666 test acc: 0.9525
train acc: 0.9701 test acc: 0.9524
train acc: 0.9692166666666666 test acc: 0.9514
train acc: 0.9713833333333334 test acc: 0.9529
train acc: 0.9727 test acc: 0.9532
train acc: 0.9726333333333333 test acc: 0.9543
train acc: 0.9731333333333333 test acc: 0.9536
train acc: 0.9731666666666666 test acc: 0.9543
train acc: 0.9732833333333333 test acc: 0.9527
train acc: 0.9748833333333333 test acc: 0.9544
train acc: 0.9747833333333333 test acc: 0.9527
train acc: 0.9744333333333334 test acc: 0.9521
train acc: 0.9743 test acc: 0.9524
train acc: 0.9755666666666667 test acc: 0.9532
train acc: 0.97585 test acc: 0.9535
train acc: 0.97715 test acc: 0.9522
train acc: 0.9761833333333333 test acc: 0.953
train acc: 0.9766333333333334 test acc: 0.9547
train acc: 0.9774666666666667 test acc: 0.9548
train acc: 0.9769 test acc: 0.9527
train acc: 0.97825 test acc: 0.9531
train acc: 0.9784 test acc: 0.954
train acc: 0.9788666666666667 test acc: 0.9535
train acc: 0.97845 test acc: 0.9514

train acc: 0.9784333333333334 test acc: 0.951
train acc: 0.9790833333333333 test acc: 0.9529
train acc: 0.9801333333333333 test acc: 0.9537
train acc: 0.9792166666666666 test acc: 0.9537
train acc: 0.97955 test acc: 0.9541
train acc: 0.9795666666666667 test acc: 0.9536
train acc: 0.9812833333333333 test acc: 0.9541
train acc: 0.9790666666666666 test acc: 0.9526
train acc: 0.98155 test acc: 0.9535
train acc: 0.98045 test acc: 0.9539
train acc: 0.9801666666666666 test acc: 0.9536
train acc: 0.9813333333333333 test acc: 0.9527
train acc: 0.9813 test acc: 0.9526
train acc: 0.9809 test acc: 0.954
train acc: 0.9812333333333333 test acc: 0.9526
train acc: 0.9818166666666667 test acc: 0.9526
train acc: 0.9817666666666667 test acc: 0.9548
train acc: 0.9825666666666667 test acc: 0.9543
train acc: 0.9816833333333334 test acc: 0.9533
train acc: 0.9818833333333333 test acc: 0.9549
train acc: 0.9821166666666666 test acc: 0.9525
train acc: 0.9829 test acc: 0.9564
train acc: 0.9829666666666667 test acc: 0.9537
train acc: 0.9834833333333334 test acc: 0.954
train acc: 0.9830666666666666 test acc: 0.9523
train acc: 0.9834666666666667 test acc: 0.9529
train acc: 0.9836666666666667 test acc: 0.9531
train acc: 0.98395 test acc: 0.9531
train acc: 0.9837 test acc: 0.9552
train acc: 0.9839333333333333 test acc: 0.9529
train acc: 0.9829833333333333 test acc: 0.9531
train acc: 0.9838833333333333 test acc: 0.9533
train acc: 0.9839833333333333 test acc: 0.9528
train acc: 0.9837833333333333 test acc: 0.9528
train acc: 0.9838833333333333 test acc: 0.9532
train acc: 0.9845333333333334 test acc: 0.9534
train acc: 0.98445 test acc: 0.9528