# brng_stoc_snn.py

```
'''
Created on 15.12.2014

@author: Peter U. Diehl
'''

import numpy as np
import matplotlib.cm as cmap
import time
import os.path
import scipy
import cPickle as pickle #텍스트 외의 자료형을 파일로 저장하기 위해 제공
from   struct import unpack # 지정된 형식 레이아웃에서 압축된 데이터를 읽는 데 사용
import os
import sys
import getopt

#------------------------------------------------------------------------------
# Specify the location of the input dataset
#------------------------------------------------------------------------------
MNIST_data_path = '/home/nano01/a/koom/SNN/MNIST/'

#------------------------------------------------------------------------------
# User-defined functions
#------------------------------------------------------------------------------
def get_labeled_data(picklename, bTrain = True):
    """Read input-vector (image) and target class (label, 0-9) and return
       it as list of tuples.
    """
    if os.path.isfile('%s.pickle' % picklename):# path 에 파일이 존재하면 실행
        # 한 줄씩 파일을 읽어오고 더 이상 로드할 데이터가 없으면 EOFError 발생
       data = pickle.load(open('%s.pickle' % picklename))
    else:
       # Open the images with gzip in read binary mode # 읽기 이진 모드에서 gzip으로 이미지 열기
       if bTrain:
          images = open(MNIST_data_path + 'train-images.idx3-ubyte','rb')
          labels = open(MNIST_data_path + 'train-labels.idx1-ubyte','rb')
       else:
          images = open(MNIST_data_path + 't10k-images.idx3-ubyte','rb')
          labels = open(MNIST_data_path + 't10k-labels.idx1-ubyte','rb')

       # Get metadata for images # 레이블에 대한 메타데이터 가져오기
       images.read(4)  # skip the magic_number
                       # python magic number: 파일의 가장 처음에 위치하는 특정 바이트. 파일 포맷을 구분하기 위해 사용된다.
       number_of_images = unpack('>I', images.read(4))[0]
       rows = unpack('>I', images.read(4))[0]
       cols = unpack('>I', images.read(4))[0]

       # Get metadata for labels
       labels.read(4)  # skip the magic_number
       N = unpack('>I', labels.read(4))[0]
       if number_of_images != N:
           # image와 대응하는 label의 개수 체크
          raise Exception('number of labels did not match the number of images')

       # Get the data
       # x: for input image. y: for label
       x = np.zeros((N, rows, cols), dtype=np.uint8)  # Initialize numpy array
       y = np.zeros((N, 1), dtype=np.uint8)           # Initialize numpy array
       for i in xrange(N):
           if i % 1000 == 0:
              print "i: %i" % i
           x[i] = [[unpack('>B', images.read(1))[0] for unused_col in xrange(cols)]  for unused_row in xrange(rows) ]
           y[i] = unpack('>B', labels.read(1))[0]
        # dictionary form
       data = {'x': x, 'y': y, 'rows': rows, 'cols': cols}
       pickle.dump(data, open("%s.pickle" % picklename, "wb"))# pickle.dump(Object, file). 객체를 파일에 저장
    return data

def get_matrix_from_file(fileName):
    offset = len(ending) + 4 # ending=''. offset = 4

    # Determine the number of rows of the target matrix
    # XeAe, AeAi, AiAe
    if fileName[-4-offset] == 'X':
       n_src = n_input # n_src = 784
    else:
       if fileName[-3-offset]=='e':
          n_src = n_e # n_src = 400
       else:
```

```python
        n_src = n_i # n_src = 400

    # Determine the number of columns of the target matrix
    if fileName[-1-offset]=='e':
       n_tgt = n_e # n_tgt = 400
    else:
       n_tgt = n_i # n_tgt = 400

    readout = np.load(fileName) # np.save()로 저장된 .npy 파일을 배열로 불러오기
    # print readout.shape, fileName
    value_arr = np.zeros((n_src, n_tgt))
    if not readout.shape == (0,): # readout의 요소가 0개가 아닐 경우
       value_arr[np.int32(readout[:,0]), np.int32(readout[:,1])] = readout[:,2]
    return value_arr

def save_connections(ending = ''):
    # brian.Connection(source, target, structure='sparse', .etc)
    #     : Mechanism for propagating spikes from one group to another
    # spikes will propagate source to target
    # default data structure is sparse

    for connName in save_conns:  # save_conns: XeAe, AeAi, AiAe
        connMatrix = connections[connName][:]
        # connMatrix의 내부(1차원 속)의 shape, rowj, rowdata를 순서대로 저장
        connListSparse = ([ (i,j[0],j[1]) for i in xrange(connMatrix.shape[0]) for j in zip(connMatrix.rowj[i],connMatrix.rowdata[i])
        np.save(store_weight_path + connName + ending, connListSparse)

def save_theta(ending = ''):
    for pop_name in population_names:  # population_names: A. 모집단 이름
        np.save(store_weight_path + 'theta_' + pop_name + ending, neuron_groups[pop_name + 'e'].theta)

def save_assignments(ending = ''):
    np.save(store_weight_path + 'assignments' + ending, assignments)

def save_postlabel(ending = ''):
    np.save(store_weight_path + 'assignments' + ending, post_label)

def get_2d_input_weights():
    name              = 'XeAe'
    n_in_sqrt         = int(np.sqrt(n_input)) # 28 = int(sqrt(784))
    n_input_use       = n_in_sqrt*n_in_sqrt # 784 = 28 * 28
    weight_matrix     = np.zeros((n_input_use, n_e)) # 784x400 matrix
    n_e_sqrt          = int(np.sqrt(n_e)) # 20 = int(sqrt(400))
    num_values_col    = n_e_sqrt*n_in_sqrt # 480 = 20 * 28
    num_values_row    = num_values_col # 480
    rearranged_weights = np.zeros((num_values_col, num_values_row)) # 480x480 matrix

    # Load the weight matrix
    connMatrix = connections[name][:]
    for i in xrange(n_input_use): # 784-loop
        # connMatrix 값에 따라 weight_matrix 매핑
        weight_matrix[i, connMatrix.rowj[i]] = connMatrix.rowdata[i]

    # Form the rearranged weight matrix
    for i in xrange(n_e_sqrt):# 20-loop
        for j in xrange(n_e_sqrt):# 20-loop
            rearranged_weights[i*n_in_sqrt : (i+1)*n_in_sqrt, j*n_in_sqrt : (j+1)*n_in_sqrt] = \
                weight_matrix[:, i + j*n_e_sqrt].reshape((n_in_sqrt, n_in_sqrt))
                # weight_matrix(784x400). indent 2 for-loop 에서 20 단위로 열 추출 후 reshape 28x28 matrix. 20개씩 20번

    return rearranged_weights

def plot_2d_input_weights():
    name    = 'XeAe'
    weights = get_2d_input_weights()# weights = weights 저장
    fig     = b.figure(fig_num, figsize = (18, 18))# figure(figure number, figsize)
    # 이미지 출력
    im2     = b.imshow(weights, interpolation = "nearest", vmin = 0, vmax = wmax_ee, cmap = cmap.get_cmap('hot_r'))
    b.colorbar(im2)
    b.title('Weights of the input to output synapses')
    # 그래프 출력
    fig.canvas.draw()
    return im2, fig

def update_2d_input_weights(im, fig):
    weights = get_2d_input_weights()
    # local parameter로 받은 im은 보통 plot_2d_input_weights() 메소드에서 리턴한 im2.
    im.set_array(weights)
    # 그려진 이미지를 다시 그린다(업데이트)
    fig.canvas.draw()
    return im

def get_current_performance(performance, current_example_num):
    # update_interval = 1000
    current_evaluation = int(current_example_num/update_interval) -1
    start_num  = current_example_num - update_interval
    end_num    = current_example_num
```

```
    # num_examples = 10000
    # input_numbers = [0] * num_examples. (10000x1 list)
    # outputNumbers = 10000x10 matrix 로 이루어진 tuple
    # difference 에 start부터 end까지 차(difference) 저장
    difference = outputNumbers[start_num:end_num, 0] - input_numbers[start_num:end_num]
    # 0, 즉 일치하는 경우의 수 만큼 correct에 저장
    correct    = len(np.where(difference == 0)[0])
    # performance = 100 * ( 정답(현재 체크하는 총 정답수) / 업데이트 주기(현재 체크하는 총 문제수) )
    performance[current_evaluation] = correct / float(update_interval) * 100
    return performance

def plot_performance(fig_num):
    # math.ceil(천정함수). x 이상의 최소정수 리턴
    # num_evaluations = 현재 체크해야 하는 집단 수
    num_evaluations = int(math.ceil(num_examples/float(update_interval)))
    # time_steps = 0-num_evaluations. iterable.
    time_steps  = range(0, num_evaluations)
    # performance. 체크해야 하는 개수만큼 크기. 1x10 matrix
    performance = np.zeros(num_evaluations)
    fig         = b.figure(fig_num, figsize = (5, 5))
    fig_num    += 1
    # fig.add_subplot(1,1,1). 1st subplot of 1x1 grid.
    ax          = fig.add_subplot(111)
    im2,        = ax.plot(time_steps, performance)  # my_cmap
    b.ylim(ymax = 100)
    b.title('Classification performance')
    # draw figure
    fig.canvas.draw()
    return im2, performance, fig_num, fig

def update_performance_plot(im, performance, current_example_num, fig):
    # performance 구한 후 image, figure 출력
    performance = get_current_performance(performance, current_example_num)
    im.set_ydata(performance)
    fig.canvas.draw()
    return im, performance

def get_recognized_number_ranking(assignments, spike_rates):
    summed_rates    = [0] * n_output # element 0. 10x1 matrix
    num_assignments = [0] * n_output # element 0. 10x1 matrix
    # assignments: 1x400 matrix or target_assignments
    # result_monitor= np.zeros((update_interval, n_e)). 1000x400 tuple

    for i in xrange(n_output): # 10-loop
        # num_assignments assignments와 i가 일치하는 곳의 [0]-th element 의 길이
        num_assignments[i] = len(np.where(assignments == i)[0])
        if num_assignments[i] > 0:
            # spike_rates: 1000x400 tuple
            # assignments가 일치하는 개수 / 개수
           summed_rates[i] = np.sum(spike_rates[assignments == i]) / num_assignments[i]
    # print 'summed_rates:', summed_rates
    # print 'Sorted summed rates:', np.argsort(summed_rates)[::-1]
    # 역순으로 정렬해서 리턴.(오름차순 -> 내림차순)
    return np.argsort(summed_rates)[::-1]

def get_new_assignments(result_monitor, input_numbers):
    assignments  = np.zeros(n_e) # 1x400 matrix (n_e = 400 가정)
    input_nums   = np.asarray(input_numbers) # ([0] * num_examples). 10000x1 tuple
    maximum_rate = [0] * n_e # 1x400 list.

    for j in xrange(n_output): # 10-loop
        # num_assignments input_nums 와 j가 일치하는 곳의 [0]-th element 의 길이
        num_assignments = len(np.where(input_nums == j)[0])
        if num_assignments > 0: # 신호가 있다면
            # result_monitor= np.zeros((update_interval, n_e)). 1000x400 tuple
            # input_nums를 통해 rate 측정
           rate = np.sum(result_monitor[input_nums == j], axis = 0) / num_assignments
           for i in xrange(n_e): # n_e-loop
               # 현재 maximum rate 보다 클 경우, update maximum rate
               if rate[i] > maximum_rate[i]:
                  maximum_rate[i] = rate[i]
                  # assigments i번째 요소 j로 초기화
                  assignments[i] = j
    return assignments

#-------------------------------------------------------------------------------
# Parse command line arguments
#-------------------------------------------------------------------------------
stoc_enable = 1
opts, args = getopt.getopt(sys.argv[1:],"hs",["help", "stoc_enable"])

for opt,arg in opts:
  if opt in ("-h", "--help"):
     print '---------------'
     print 'Usage Example:'
     print '---------------'
     print os.path.basename(__file__) + ' --help        -> Print script usage example'
```

```
      print os.path.basename(__file__) + ' --stoc_enable -> Enable stochasticity'
      sys.exit(1)
   elif opt in ("-s", "--stoc_enable"):
      stoc_enable = 1

if(stoc_enable):
   print '-------------------------------------------------------------------'
   print 'Synapses connecting the input and excitatory neurons are stochastic!'
   print '-------------------------------------------------------------------'
else:
   print '-----------------------------------------------------------------------'
   print 'Synapses connecting the input and excitatory neurons are NOT stochastic!'
   print '-----------------------------------------------------------------------'

#-----------------------------------------------------------------------------
# Load the input dataset
#-----------------------------------------------------------------------------
start    = time.time()
training = get_labeled_data(MNIST_data_path + 'training')
end      = time.time()

start   = time.time()
testing = get_labeled_data(MNIST_data_path + 'testing', bTrain = False)
end     = time.time()

#-----------------------------------------------------------------------------
# Set parameters and equations
#-----------------------------------------------------------------------------
import brian_no_units  #import it to deactivate unit checking --> This should NOT be done for testing/debugging
import brian as b
from   brian import *
import math

b.set_global_preferences(
                        defaultclock          = b.Clock(dt=0.5*b.ms),  # The default clock to use if none is provided or defined in an
                        useweave              = True,  # Defines whether or not functions should use inlined compiled C code where def
                        gcc_options           = ['-ffast-math -march=native'],  # Defines the compiler switches passed to the gcc comp
                        #For gcc versions 4.2+ we recommend using -march=native. By default, the -ffast-math optimizations are turned
                        usecodegen            = True,  # Whether or not to use experimental code generation support.
                        usecodegenweave       = True,  # Whether or not to use C with experimental code generation support.
                        usecodegenstateupdate = True,  # Whether or not to use experimental code generation support on state updaters.
                        usecodegenthreshold   = False, # Whether or not to use experimental code generation support on thresholds.
                        usenewpropagate       = True,  # Whether or not to use experimental new C propagation functions.
                        usecstdp              = False, # Whether or not to use experimental new C STDP.
                       )

#-------------------------------------------------------------------------
# Initialize seed for the random number generators
#-------------------------------------------------------------------------
np.random.seed(0)

#-------------------------------------------------------------------------
# SNN topology parameters
#-------------------------------------------------------------------------
data_path = './'
ending    = ''
n_input   = 784
n_output  = 10
n_label   = 1000  #Unused
n_e       = 400
n_i       = n_e

#-------------------------------------------------------------------------
# SNN simulation parameters
#-------------------------------------------------------------------------
num_examples        = 6000 * 1
single_example_time = 0.35 * b.second
resting_time        = 0.15 * b.second
dt_clock            = 0.5 * b.ms  # Need to change the default clock option in global parameters
num_timesteps       = single_example_time / dt_clock
runtime             = num_examples * (single_example_time + resting_time)

use_testing_set       = True
use_weight_dependence = True  # Unused
use_classic_STDP      = True  # Unused
test_mode             = True
tag_mode              = False

if num_examples < 10000:
    weight_update_interval    = 20
    save_connections_interval = 10000
else:
    weight_update_interval    = 1000
    save_connections_interval = 250000

if test_mode:
    if(stoc_enable == 0):
```

```
        load_weight_path  = data_path + 'weights/'
    else:
        load_weight_path  = data_path + 'weights_stoc/'

    do_plot_performance = True
    record_spikes       = False
    record_state        = False
    ee_STDP_on          = False
    if(use_testing_set):
        num_examples_total = len(testing['y'])
    else:
        num_examples_total = len(training['y'])
    num_examples        = 10000
    update_interval     = 1000

elif tag_mode:
    if(stoc_enable == 0):
        load_weight_path  = data_path + 'weights/'
        store_weight_path = data_path + 'weights/'
    else:
        load_weight_path  = data_path + 'weights_stoc/'
        store_weight_path = data_path + 'weights_stoc/'

    do_plot_performance = False
    record_spikes       = False
    record_state        = False
    ee_STDP_on          = False
    num_examples_total  = len(training['y'])
    update_interval     = num_examples

else:
    if(stoc_enable == 0):
        load_weight_path  = data_path + 'random/'
        store_weight_path = data_path + 'weights/'
    else:
        load_weight_path  = data_path + 'random_stoc/'
        store_weight_path = data_path + 'weights_stoc/'

    do_plot_performance = False
    record_spikes       = False
    record_state        = False
    ee_STDP_on          = True
    num_examples_total  = len(training['y'])
    update_interval     = num_examples

#---------------------------------------------------------------------------
# Create the target labels for the output neurons
#---------------------------------------------------------------------------
if not test_mode:
   assert(n_e%n_output == 0)
   neurons_per_op = n_e/n_output
   post_indx      = np.random.permutation(n_e)
   post_label     = np.zeros(n_e)

   for i in range(n_output):
       post_label[post_indx[i*neurons_per_op:(i+1)*neurons_per_op]] = i

#---------------------------------------------------------------------------
# BRNG switching probability characteristics
#---------------------------------------------------------------------------
p_switch = np.array([0.0909, 0.0935, 0.0961, 0.1014, 0.1041, 0.1080, 0.1133, 0.1199, \
                     0.1238, 0.1291, 0.1344, 0.1423, 0.1502, 0.1581, 0.1647, 0.1765, \
                     0.1779, 0.1950, 0.2069, 0.2187, 0.2266, 0.2411, 0.2556, 0.2819, \
                     0.2872, 0.3030, 0.3228, 0.3439, 0.3623, 0.3874, 0.4150, 0.4493, \
                     0.4888, 0.5191, 0.5455, 0.5626, 0.5863, 0.6087, 0.6337, 0.6469, \
                     0.6693, 0.6812, 0.6970, 0.7088, 0.7352, 0.7457, 0.7589, 0.7708, \
                     0.7800, 0.7866, 0.8011, 0.8103, 0.8169, 0.8274, 0.8340, 0.8393, \
                     0.8511, 0.8538, 0.8603, 0.8669, 0.8762, 0.8814, 0.8906])
p_switch = p_switch - p_switch[0]
i_switch = np.arange(1, 64)
i_scale  = 5
i_norm   = (i_switch*1.0) / i_scale

#---------------------------------------------------------------------------
# BRNG-based excitatory neuron
#---------------------------------------------------------------------------
v_rest_e   = -65. * b.mV
v_reset_e  = -65. * b.mV
v_thresh_e = -52. * b.mV
refrac_e   =  5.  * b.ms

if test_mode:
    scr_e       = 'timer = 0*ms'
else:
    theta_plus_e = 1000  * b.mV
    theta_stop   = 100e3 * b.mV
    scr_e        = 'theta = theta+theta_plus_e; timer = 0*ms'
```

```
if(test_mode or tag_mode):
   v_thresh_e = '(interp(I_post, i_norm, p_switch) > rand())'
   #v_thresh_e = 'I_post>3'
else:
   v_thresh_e = '(interp(I_post, i_norm, p_switch) > rand()) * (img_label==post_label)'
   #v_thresh_e = '(I_post>3)* (img_label==post_label)'

if not test_mode:
   neuron_eqs_e = '''
        I_post = (ge-gi) * (theta<=theta_stop) * 1. : 1
        I_synE = ge                                 : 1
        I_synI = gi                                 : 1
        dge/dt = -ge/(2.0*ms)                       : 1
        dgi/dt = -gi/(1.0*ms)                       : 1
'''
else:
   neuron_eqs_e = '''
        I_post = (ge-gi)                            : 1
        I_synE = ge                                 : 1
        I_synI = gi                                 : 1
        dge/dt = -ge/(4.0*ms)                       : 1
        dgi/dt = -gi/(2.0*ms)                       : 1
        '''

if test_mode:
    neuron_eqs_e += '\n  theta      : volt'
else:
    neuron_eqs_e += '\n  theta      : volt'
    neuron_eqs_e += '\n  img_label  : 1.0'
    neuron_eqs_e += '\n  post_label : 1.0'

neuron_eqs_e += '\n  dtimer/dt = 100.0 : ms'

#---------------------------------------------------------------------------
# BRNG-based inhibitory neuron
#---------------------------------------------------------------------------
v_rest_i   = -60. * b.mV
v_reset_i  = -45. * b.mV
v_thresh_i = -40. * b.mV
refrac_i   =  2.  * b.ms

neuron_eqs_i = '''
        I_post = (ge-gi)      : 1
        I_synE = ge           : 1
        I_synI = gi           : 1
        dge/dt = -ge/(1.0*ms) : 1
        dgi/dt = -gi/(2.0*ms) : 1
        '''

# Update the spiking mechanism of the probabilistic inhibitory neuron
v_thresh_i = '(I_post > 5)'

#---------------------------------------------------------------------------
# Implement STDP for synapses connecting the input and liquid-excitatory neurons
#---------------------------------------------------------------------------
# Stochastic STDP (potentiation) parameters
tc_pre_ee = 6*b.ms
pre_rst   = 0.10

# Stochastic STDP (depression) parameters
tc_post_1_ee = 6*b.ms
post_rst     = 0.01

# Power-law weight-dependent STDP (potentiation) parameters
nu_ee_post  = 0.001  # Unused
STDP_offset = 0.4    # Unused
exp_ee_post = 0.9    # Unused

# Synaptic weight constraints
wmax_ee = 1.0
wmin_ee = 0.0

# STDP equations
eqs_stdp_ee = '''
            dpre/dt  = -pre/(tc_pre_ee)      : 1.0
            dpost/dt = -post/(tc_post_1_ee)  : 1.0
            '''
if(stoc_enable == 0):
   eqs_stdp_pre_ee  = 'pre += 1.'
   eqs_stdp_post_ee = 'w += (nu_ee_post * (pre - STDP_offset) * ((wmax_ee - w)**exp_ee_post)); post += 1.'
else:
   # hebb_dep_count   = np.zeros((n_input, n_e))
   # eqs_stdp_pre_ee  = 'w -= ((post>=STDP_offset_dep_neg)*1.0*(prob_dep_neg>rand('+str(n_e)+'))); pre = 1.; hebb_dep_count += ((post>
   eqs_stdp_pre_ee = 'w -= ((post>rand(' + str(n_e) + '))*1.0); pre = pre_rst'

   # hebb_pot_count   = np.zeros((n_input, n_e))
```

```
   # eqs_stdp_post_ee = 'rand_updt_onebit = rand('+str(n_input)+'); w += ((pre>=STDP_offset_pot)*1.0*(prob_pot>rand_updt_onebit)) + ((
   eqs_stdp_post_ee = 'w += ((pre>rand('+str(n_input)+'))*1.0); post = post_rst'

#--------------------------------------------------------------------------
# SNN connectivity specification
#--------------------------------------------------------------------------
conn_structure         = 'sparse'
delay                  = {}
input_population_names = ['X']
population_names       = ['A']
input_connection_names = ['XA']
save_conns             = ['XeAe','AeAi', 'AiAe']
input_conn_names       = ['ee_input']
recurrent_conn_names   = ['ei', 'ie']
delay['ee_input']      = (0*b.ms,10*b.ms)
delay['ei_input']      = (0*b.ms,5*b.ms)

#------------------------------------------------------------------------------
# Create the neuron groups
#------------------------------------------------------------------------------
b.ion()
fig_num        = 1
neuron_groups  = {}
input_groups   = {}
connections    = {}
stdp_methods   = {}
rate_monitors  = {}
spike_monitors = {}
spike_counters = {}
Ipost_monitors = {}

# Excitatory and inhibitory neuron groups
neuron_groups['e'] = b.NeuronGroup(n_e*len(population_names), neuron_eqs_e, threshold = v_thresh_e, refractory = refrac_e, reset = scr
                                   compile = True, freeze = True)
neuron_groups['i'] = b.NeuronGroup(n_i*len(population_names), neuron_eqs_i, threshold = v_thresh_i, refractory = refrac_i, reset = v_r
                                   compile = True, freeze = True)

#------------------------------------------------------------------------------
# Create network population and recurrent connections
#------------------------------------------------------------------------------
for name in population_names:  # population_names: A
    neuron_groups[name+'e'] = neuron_groups['e'].subgroup(n_e)
    neuron_groups[name+'i'] = neuron_groups['i'].subgroup(n_i)

    # neuron_groups[name+'e'].v = v_rest_e - 40. * b.mV
    # neuron_groups[name+'i'].v = v_rest_i - 40. * b.mV

    if test_mode or tag_mode or load_weight_path[-8:] == 'weights/':
       # neuron_groups['e'].theta = np.load(load_weight_path + 'theta_' + name + ending + '.npy')
       # print '\n------- THETA --------'
       # print neuron_groups['e'].theta

       if(test_mode and (not tag_mode)):
          target_assignments = np.load(load_weight_path + 'assignments' + ending + '.npy')
          print '\n------- TARGET ASSIGNMENTS --------'
          print target_assignments

          print '\n------- ASSIGNMENT STATISTICS --------'
          num_target_assign = np.zeros(n_output)
          # theta_avg       = np.zeros(n_output)
          for i in range(n_output):
              num_target_assign[i] = len(np.where(target_assignments == i)[0])
          #    if(num_target_assign[i] > 0):
          #       theta_avg[i]     = np.sum(neuron_groups['e'].theta[target_assignments == i]) / num_target_assign[i]
          print num_target_assign
          # print theta_avg
    else:
       neuron_groups['e'].theta = np.ones((n_e)) * 0.0*b.mV

    # Create recurrent connections between excitatory and inhibitory layer
    for conn_type in recurrent_conn_names: # recurrent_conn_names: ei, ie
        connName = name+conn_type[0]+name+conn_type[1]
        print '########## Creating connection:' + connName + ' ##########'
        weightMatrix = get_matrix_from_file(load_weight_path + connName + ending + '.npy')
        weightMatrix = scipy.sparse.lil_matrix(weightMatrix)
        connections[connName] = b.Connection(neuron_groups[connName[0:2]], neuron_groups[connName[2:4]], structure = conn_structure,
                                             state = 'g'+conn_type[0])
        connections[connName].connect(neuron_groups[connName[0:2]], neuron_groups[connName[2:4]], weightMatrix)

    # Create rate and spike monitors
    rate_monitors[name+'e']  = b.PopulationRateMonitor(neuron_groups[name+'e'], bin = (single_example_time+resting_time)/b.second)
    rate_monitors[name+'i']  = b.PopulationRateMonitor(neuron_groups[name+'i'], bin = (single_example_time+resting_time)/b.second)
    spike_counters[name+'e'] = b.SpikeCounter(neuron_groups[name+'e'])

    if record_spikes:
       spike_monitors[name+'e'] = b.SpikeMonitor(neuron_groups[name+'e'])
       spike_monitors[name+'i'] = b.SpikeMonitor(neuron_groups[name+'i'])
```

```
    if record_state:
       Ipost_monitors['e'] = b.StateMonitor(neuron_groups['e'], 'I_post', record=True)
       Ipost_monitors['i'] = b.StateMonitor(neuron_groups['i'], 'I_post', record=True)

if record_spikes:
   b.figure(fig_num)
   fig_num += 1
   b.ion()
   b.subplot(211)
   b.raster_plot(spike_monitors['Ae'], refresh=1000*b.ms, showlast=1000*b.ms)
   b.subplot(212)
   b.raster_plot(spike_monitors['Ai'], refresh=1000*b.ms, showlast=1000*b.ms)

if record_state:
   b.figure(fig_num)
   fig_num += 1
   b.ion()
   b.subplot(211)
   Ipost_monitors['e'].plot(refresh=1000*b.ms, showlast=1000*b.ms)
   b.subplot(212)
   Ipost_monitors['i'].plot(refresh=1000*b.ms, showlast=1000*b.ms)

#------------------------------------------------------------------------------
# Create input population and connections from input populations
#------------------------------------------------------------------------------
for i,name in enumerate(input_population_names):  # input_population_names: X
    if name == 'Y':
       input_groups[name+'e'] = b.PoissonGroup(n_label, 0)
    else:
       input_groups[name+'e'] = b.PoissonGroup(n_input, 0)
    rate_monitors[name+'e'] = b.PopulationRateMonitor(input_groups[name+'e'], bin = (single_example_time+resting_time)/b.second)

for name in input_connection_names:    # input_connection_names: XA
    for connType in input_conn_names:  # input_conn_names      : ee_input
        connName = name[0] + connType[0] + name[1] + connType[1]
        print '########## Creating connection:' + connName + ' ##########'
        weightMatrix = get_matrix_from_file(load_weight_path + connName + ending + '.npy')
        weightMatrix = scipy.sparse.lil_matrix(weightMatrix)
        connections[connName] = b.Connection(input_groups[connName[0:2]], neuron_groups[connName[2:4]], structure = conn_structure,
                                             state = 'g'+connType[0], delay = True, max_delay = delay[connType][1])
        connections[connName].connect(input_groups[connName[0:2]], neuron_groups[connName[2:4]], weightMatrix, delay = delay[connType]

    if ee_STDP_on:
       stdp_methods[name[0]+'e'+name[1]+'e'] = b.STDP(connections[name[0]+'e'+name[1]+'e'], eqs = eqs_stdp_ee, pre = eqs_stdp_pre_ee,
                                                      post = eqs_stdp_post_ee, wmin = wmin_ee, wmax = wmax_ee)

#------------------------------------------------------------------------------
# Run the simulation
#------------------------------------------------------------------------------
# Print presynaptic trace during the simulation
# @network_operation
# def print_ng():
#    print neuron_groups['Ae'].ge, neuron_groups['Ae'].I_synE
# b.network_operation(print_ng)

# Initialize the spike counters of the excitatory neurons
result_monitor       = np.zeros((update_interval, n_e))
previous_spike_count = np.zeros(n_e)

# Initialize the excitatory neuronal assignments
if not test_mode:
   neuron_groups['e'].post_label = post_label
   neuron_groups['e'].img_label  = np.ones(n_e) * -1
   assignments = np.zeros(n_e)
else:
   assignments = target_assignments

# Input/Output labels
input_numbers = [0] * num_examples
outputNumbers = np.zeros((num_examples, n_output))

# Setup the weight and performance plots
if((not test_mode) and (not tag_mode) and (weight_update_interval != 0)):
    input_weight_monitor, fig_weights = plot_2d_input_weights()
    fig_num += 1

if do_plot_performance:
    performance_monitor, performance, fig_num, fig_performance = plot_performance(fig_num)

# Initialize the spiking rate of input neurons
for i,name in enumerate(input_population_names):
    input_groups[name+'e'].rate = 0

# Configure the run-time simulation parameters
b.run(0)
if(test_mode and use_testing_set):
```

```
    j = 0
    k = 0
    epoch = 0
    SPIKE_THRESH = 5
else:
    j = 0
    k = 0
    epoch = 0
    SPIKE_THRESH = 0
input_intensity = 2.
start_input_intensity = input_intensity

# Configure to train the SNN on a subset of the input patterns
train_digits = np.array([0, 5, 1, 6, 9, 2, 4, 7, 3, 8])
dig_indx     = 0
img_target   = train_digits[dig_indx]

while j < (int(num_examples)):
    if test_mode:
        if use_testing_set:
            rates     = testing['x'][k%num_examples_total,:,:].reshape((n_input)) / 8. *  input_intensity
            img_label = testing['y'][k%num_examples_total][0]
        else:
            while(training['y'][k%num_examples_total][0] != img_target):
                k += 1
            rates     = training['x'][k%num_examples_total,:,:].reshape((n_input)) / 8. *  input_intensity
            img_label = training['y'][k%num_examples_total][0]
    else:
        while(training['y'][k%num_examples_total][0] != img_target):
            k += 1
        rates     = training['x'][k%num_examples_total,:,:].reshape((n_input)) / 8. *  input_intensity
        img_label = training['y'][k%num_examples_total][0]

    input_groups['Xe'].rate = rates
    print 'run number:', str(j+1), 'of', str(num_examples), 'image label =', str(img_label)
    if not test_mode:
        neuron_groups['e'].img_label = img_label
    b.run(single_example_time)

    if j % update_interval == 0 and j > 0:
        if not test_mode:
            assignments = get_new_assignments(result_monitor[:], input_numbers[j-update_interval : j])
        else:
            assignments = target_assignments

    if(weight_update_interval != 0):
        if j % weight_update_interval == 0 and (not test_mode) and (not tag_mode):
            update_2d_input_weights(input_weight_monitor, fig_weights)

    if j % save_connections_interval == 0 and j > 0 and not test_mode:
        if not tag_mode:
            save_connections(str(j))
            save_theta(str(j))
        else:
            save_assignments(str(j))

    # Update the spike count of the excitatory neurons
    current_spike_count  = np.asarray(spike_counters['Ae'].count[:]) - previous_spike_count
    previous_spike_count = np.copy(spike_counters['Ae'].count[:])

    if np.sum(current_spike_count) < SPIKE_THRESH:
        input_intensity += 1
        for i,name in enumerate(input_population_names):
            input_groups[name+'e'].rate = 0
        b.run(resting_time)
    else:
        # Increment the number of training epochs
        if(not test_mode):
            if(np.sum(current_spike_count) > 0):
                epoch += 1

        # Print the index of active neurons
        # ae_active_idx = [ae_idx for ae_idx,spike_count in enumerate(current_spike_count) if spike_count>0]
        # print ae_active_idx
        # print '----------------------------------'

        # Update the result monitor
        result_monitor[j%update_interval,:] = current_spike_count

        # Update the input and output labels
        if test_mode and use_testing_set:
            input_numbers[j] = testing['y'][k%num_examples_total][0]
        else:
            input_numbers[j] = training['y'][k%num_examples_total][0]
        outputNumbers[j,:]  = get_recognized_number_ranking(assignments, result_monitor[j%update_interval,:])
        # if(test_mode):
        #   print 'Input  Label:', img_label
```

```
        #   print 'Output Label:', outputNumbers[j,0]
        #   print '----------------------------------'

        if j % update_interval == 0 and j > 0:
           if do_plot_performance:
              unused, performance = update_performance_plot(performance_monitor, performance, j, fig_performance)
              print 'Classification performance', performance[:(j/int(update_interval))]

        for i,name in enumerate(input_population_names):
             input_groups[name+'e'].rate = 0
        b.run(resting_time)
        input_intensity = start_input_intensity

        if(test_mode and use_testing_set):
           j += 1
           k += 1
        else:
           # Stop training if the total post-synaptic current is zero (CODE_CHANGE)
           if((not test_mode) and (np.sum(neuron_groups['e'].I_post)==0)):
               print 'Number of training examples =', j+1
               print 'Number of training iterations =', epoch
               break
           j += 1
           k += 10
           dig_indx   = (dig_indx+1) % train_digits.size
           img_target = train_digits[dig_indx]

# Update classification performance at the end of the simulation
if do_plot_performance and test_mode:
   unused, performance = update_performance_plot(performance_monitor, performance, j, fig_performance)
   print 'Classification performance', performance[:(j/int(update_interval))]
   print 'Final accuracy = ', np.mean(performance)

# Update the synaptic weight plot at the end of the simulation
if((not test_mode) and (not tag_mode) and (weight_update_interval != 0)):
    update_2d_input_weights(input_weight_monitor, fig_weights)

#------------------------------------------------------------------------------
# Save results
#------------------------------------------------------------------------------
if not test_mode:
    # if not tag_mode:
    #   print neuron_groups['e'].theta
    #   save_theta()

    # Tag the excitatory neurons based on their spiking activity
    # update_interval = j
    # assignments     = get_new_assignments(result_monitor[:], input_numbers[j-update_interval : j])
    # print '\n------- ASSIGNMENTS --------'
    # print assignments
    # save_assignments()

    print '\n------- POST_LABELS --------'
    print post_label
    save_postlabel()

if((not test_mode) and (not tag_mode)):
    save_connections()

    # Plot the STDP update statistics
    # print "Number of Hebbian depression (switching synapses to '00' state) updates = ", str(np.sum(hebb_dep_count))
    # hebb_dep_count_per_ne = np.mean(hebb_dep_count, axis=1)
    # hebb_dep_count_per_ne = hebb_dep_count_per_ne / np.max(hebb_dep_count_per_ne)
    # hebb_dep_count_per_ne = hebb_dep_count_per_ne.reshape(28,28)
    # fig_num += 1
    # fig_hebb_dep = b.figure(fig_num, figsize = (18, 18))
    # img_hebb_dep = b.imshow(hebb_dep_count_per_ne, interpolation = "nearest", vmin = 0, vmax = 1, cmap = cmap.get_cmap('hot_r'))
    # b.colorbar(img_hebb_dep)
    # b.title('Hebbian Synaptic Depression')
    # fig_hebb_dep.canvas.draw()

    # print "Number of Hebbian potentiation (switching synapses to '11' state) updates = ", str(np.sum(hebb_pot_count))
    # hebb_pot_count_per_ne = np.mean(hebb_pot_count, axis=1)
    # hebb_pot_count_per_ne = hebb_pot_count_per_ne / np.max(hebb_dep_count_per_ne)  # Normalized with respect to the #Hebbian-depress
    # hebb_pot_count_per_ne = hebb_pot_count_per_ne.reshape(28,28)
    # fig_num += 1
    # fig_hebb_pot = b.figure(fig_num, figsize = (18, 18))
    # img_hebb_pot = b.imshow(hebb_pot_count_per_ne, interpolation = "nearest", vmin = 0, vmax = 1, cmap = cmap.get_cmap('hot_r'))
    # b.colorbar(img_hebb_pot)
    # b.title('Hebbian Synaptic Potentiation')
    # fig_hebb_pot.canvas.draw()
else:
    np.save(data_path + 'activity/resultPopVecs' + str(num_examples), result_monitor)
    np.save(data_path + 'activity/inputNumbers' + str(num_examples), input_numbers)

#------------------------------------------------------------------------------
# Plot the results
```

```
#------------------------------------------------------------------------------
# if rate_monitors:
#     b.figure(fig_num)
#     fig_num += 1
#     for i, name in enumerate(rate_monitors):
#         b.subplot(len(rate_monitors), 1, i)
#         b.plot(rate_monitors[name].times/b.second, rate_monitors[name].rate, '.')
#         b.title('Rates of population ' + name)

# if spike_monitors:
#     b.figure(fig_num)
#     fig_num += 1
#     for i, name in enumerate(spike_monitors):
#         b.subplot(len(spike_monitors), 1, i)
#         b.raster_plot(spike_monitors[name])
#         b.title('Spikes of population ' + name)

# if spike_counters:
#     b.figure(fig_num)
#     fig_num += 1
#     for i, name in enumerate(spike_counters):
#         b.subplot(len(spike_counters), 1, i)
#         b.plot(spike_counters['Ae'].count[:])
#         b.title('Spike count of population ' + name)

# print 'Classification performance', performance[len(performance)-1]
# plot_2d_input_weights()
b.ioff()
b.show()
```