注：带L的指是输出log相关，影响程序运行

```
from __future__ import print_function
from collections import deque

from hjk_real_facing_people_webots_env_obstacle import WebotsLidarNnEnv
from hjk_saved_neural_qlearning import NeuralQLearner
from hjk_real_facing_people_webots_env_obstacle import actionoutPath
import tensorflow as tf
import numpy as np
import sys
import rospy
import random
import copy
import xml.dom.minidom
import gc
from time import gmtime, strftime

out_wintimes_path = "my_net22/wintimes_" + strftime("%Y-%m-%d-%H-%M-%S", gmtime()) + ".txt"
out_test_wintimes_path = "my_net22/test_wintimes_" + strftime("%Y-%m-%d-%H-%M-%S", gmtime()) + ".txt"
laser_dim = 16
dst_dim = 2
facing_dim = 1
history_dim = 1
state_dim = laser_dim + dst_dim + facing_dim + history_dim

num_actions = 5

MAX_STEPS = 150
COLLISION_THRESHOLD = 0.3
episode_history = deque(maxlen=100)
# 0 means to train; 1 means to test for my case ;
py_function = 0
MAX_EPISODES = 100000
JUSTTEST = 0
# now when justtest, it will make use. And should use right limit_sta_dis_value and limit_sta_dis_pos
USELIMITSTADIS = 0
LIMITSTADISVALUE = 3 * 10
LIMITSTADISPOS = laser_dim    # 0~15is laser, 16 is dist

## four layers
num_layers = 4
num_neural = [state_dim, 18, 18, num_actions]
```

```
human_x = []
human_y = []
human_rotation_z = []
robot_x = []
robot_y = []
robot_rotation_z = []


def loadxml():
    dom = xml.dom.minidom.parse('src/env/test.xml')
    root = dom.documentElement
    bb = root.getElementsByTagName('episode')
    for i, var in enumerate(bb):
        human_x.append(float(var.getElementsByTagName('human_x')[0].firstChild.data))
        human_y.append(float(var.getElementsByTagName('human_y')[0].firstChild.data))

human_rotation_z.append(int(var.getElementsByTagName('human_rotation_z')[0].firstChild.d
ata))
        robot_x.append(float(var.getElementsByTagName('robot_x')[0].firstChild.data))
        robot_y.append(float(var.getElementsByTagName('robot_y')[0].firstChild.data))

robot_rotation_z.append(float(var.getElementsByTagName('robot_rotation_z')[0].firstChild.da
ta))


def add_layer(inputs, in_size, out_size, w_name, b_name, activation_function=None):
    Weights = tf.get_variable(w_name, [in_size, out_size],
                              initializer=tf.random_normal_initializer(mean=0.0,
stddev=0.2))

    biases = tf.get_variable(b_name, out_size,
                             initializer=tf.constant_initializer(0))

    Wx_plus_b = tf.matmul(inputs, Weights) + biases

    if activation_function is None:
        outputs = Wx_plus_b
    else:
        outputs = activation_function(Wx_plus_b)

    return outputs
```

```
def init_q_net(states):
    h1 = add_layer(states, num_neural[0], num_neural[1], 'W1', 'b1', activation_function=tf.nn.relu)

    h2 = add_layer(h1, num_neural[1], num_neural[2], 'W2', 'b2', activation_function=tf.nn.relu)

    q = add_layer(h2, num_neural[2], num_neural[3], 'W3', 'b3', activation_function=None)

    return q


def complex_init_q_net(states):
    laser = tf.slice(states, [0, 0], [-1, laser_dim])
    goal = tf.slice(states, [0, laser_dim], [-1, dst_dim + facing_dim])
    his_a = tf.slice(states, [0, laser_dim + dst_dim + facing_dim], [-1, history_dim])
    # print(laser)
    # print(goal)
    # print(his_a)
    laser_8 = add_layer(laser, laser_dim, 8, 'W1', 'b1', activation_function=tf.nn.relu)
    cat_laser_8_his_a = tf.concat([laser_8, his_a], 1)
    # print(cat_laser_8_his_a)
    laser_5 = add_layer(cat_laser_8_his_a, 9, 5, 'W2', 'b2', activation_function=tf.nn.relu)
    cat_laser_5_goal = tf.concat([laser_5, goal], 1)
    cat_r8 = add_layer(cat_laser_5_goal, 8, 8, 'W3', 'b3', activation_function=tf.nn.relu)
    cat_5 = add_layer(cat_r8, 8, 5, 'W4', 'b4', activation_function=None)
    # h_8_and_1 = tf.concat(1, h_)
    # h2 = add_layer(h1, num_neural[1], num_neural[2], 'W2', 'b2', activation_function=tf.nn.relu)

    # q = add_layer(h2, num_neural[2], num_neural[3], 'W3', 'b3', activation_function=None)

    return cat_5


def testtest(path):
    env.collision_threshold = 0.25
    xml_test_cnt = 0
    sum_steps = 0
    l = len(human_x)
    #print('---- ', l)
    for i_episode in xrange(l):
        env.my_case = -1
        state = env.reset(1, human_x[i_episode], human_y[i_episode],
```

```
human_rotation_z[i_episode], robot_x[i_episode],
                                robot_y[i_episode], robot_rotation_z[i_episode])
        last_action = -1
        wrongActionTimes = 0
        record_t = -1
        for t in xrange(MAX_STEPS):
            record_t = t
            #print("In episode    ", i_episode, ":")

            #print('step ' + str(t))

            # print("new change : ", state, "sure: ", state[-1])


            # print("333333 --- state: ", state[LIMITSTADISPOS], state)
            limit_state = copy.deepcopy(state)
            if limit_state[LIMITSTADISPOS] > LIMITSTADISVALUE:
                limit_state[LIMITSTADISPOS] = LIMITSTADISVALUE - random.random() *
10

                # print("www i don't konw: ", state)
            print('ttttttttttttttttttttttttttttttttttttttttttttttttt:    ', i_episode, 'wintimes: ', xml_test_cnt)
            action = q_learner.eGreedyAction(limit_state[np.newaxis, :], False)

            if (last_action == 3 and action == 4) or (last_action == 4 and action == 3):
                wrongActionTimes = wrongActionTimes + 1
                if wrongActionTimes == 2:
                    wrongActionTimes = 0
                    print('hjk--- ffffff :', action)
                    action = 0

            #print('hjk--action:    ', action, 'lastaction', last_action)
            actionout = open(actionoutPath, 'a+')
            #
print("??????????????????????????????????????????????????????????????????????????????????????????????????
??")
            print('action: ', action, 'lastaction: ', last_action, file = actionout)
            sys.stdout.flush()
            actionout.close()
            next_state, reward, done, _ = env.step(action)

            last_action = action


            state = next_state
```

```
                if done:
                    if reward >= 500 - 1:
                        xml_test_cnt += 1
                        sum_steps += t
                        out_test_wintimes = open(out_test_wintimes_path, 'a+')
                        print('testround:  ', i_episode, "win!!  use  steps:  ", t,
file=out_test_wintimes)
                        out_test_wintimes.close()
                    break
        if xml_test_cnt == 0:
            mean_steps = 0
        else:
            mean_steps = sum_steps * 1.0 / xml_test_cnt
        out_test_wintimes = open(out_test_wintimes_path, 'a+')
        print('test  ', path, "wintimes:  ", xml_test_cnt, "mean_steps:  ", mean_steps,
file=out_test_wintimes)
        sys.stdout.flush()
        out_test_wintimes.close()
        env.collision_threshold = COLLISION_THRESHOLD


if __name__ == '__main__':
    len_args = len(sys.argv)
    path = None

    if (len_args > 1):
        path = str(sys.argv[1])
    loadxml()
    env_name = 'facing_people_webots_env_obstacle'
    sess = tf.Session()
    #### hjk change the learning_rate to 0.001. nnn.....
    optimizer = tf.train.RMSPropOptimizer(learning_rate=0.004, decay=0.9)
    #    writer    =    tf.train.SummaryWriter("/tmp/{}-experiment-1".format(env_name),
graph=sess.graph)
    writer    =    tf.summary.FileWriter("my_net20/{}-experiment-1".format(env_name),
graph=sess.graph)
    if path is not None:
        print('resotre net path: ' + path)
    else:
        print("init")
    # restore_net(sess, path)
```

```
    q_learner = NeuralQLearner(sess,
                               optimizer,
                               complex_init_q_net,
                               path,
                               state_dim,
                               num_actions,
                               512,  # batch_size=32,
                               0.5,  # init_exp=0.3,  # 0.5,          # initial exploration
prob
                               0.1,  # final_exp=0.001,  # final exploration prob
                               #  anneal_steps=10000,  #  N  steps  for  annealing
exploration
                               200000,    #  anneal_steps=2000,    #  N  steps  for
annealing exploration
                               10000,  # replay_buffer_size=10000,
                               3,  # store_replay_every=3,  # how frequent to store
experience
                               0.9,  # discount_factor=0.9,  # discount future rewards
                               0.01,  # target_update_rate=0.01,
                               0.01,  # reg_param=0.01,  # regularization constants
                               5,  # max_gradient=5,  # max gradient norms
                               False,  # double_q_learning=False,

                               None,  # summary=None,
                               100  # summary_every=100
                               )

    # print(sess.run(tf.get_default_graph().get_tensor_by_name("q_network/b3:0")))

    # print(sess.run(tf.get_default_graph().get_tensor_by_name("target_network/b3:0")))

    env = WebotsLidarNnEnv(laser_dim, COLLISION_THRESHOLD)
    wintimes = 0
    for i_episode in xrange(MAX_EPISODES):
        # initialize
        if py_function == 1:
            env.my_case = i_episode
        else:
            env.my_case = -1
        state = env.reset()
        # print("2222222 --- state: ", state[LIMITSTADISPOS])
        total_rewards = 0
```

```
        last_action = -1
        wrongActionTimes = 0
        record_t = -1
        for t in xrange(MAX_STEPS):
            gc.collect()
            record_t = t
            print("In episode    ", i_episode, ":")

            print('step ' + str(t))

            # print("new change : ", state, "sure: ", state[-1])

            if JUSTTEST == 0:
                action = q_learner.eGreedyAction(state[np.newaxis, :])
            else:
                if USELIMITSTADIS == 0:
                    action = q_learner.eGreedyAction(state[np.newaxis, :], False)
                else:
                    # print("333333 --- state: ", state[LIMITSTADISPOS], state)
                    limit_state = copy.deepcopy(state)
                    if limit_state[LIMITSTADISPOS] > LIMITSTADISVALUE:
                        limit_state[LIMITSTADISPOS]      =      LIMITSTADISVALUE      -
random.random() * 10
                        # print("www i don't konw: ", state)
                    print('hjk--- limit_state:    ', state, limit_state)
                    action = q_learner.eGreedyAction(limit_state[np.newaxis, :], False)

            if (last_action == 3 and action == 4) or (last_action == 4 and action == 3):
                wrongActionTimes = wrongActionTimes + 1
                if wrongActionTimes == 2:
                    wrongActionTimes = 0
                    print('hjk--- ffffff :', action)
                    action = 0

            print('hjk--action:    ', action, 'lastaction', last_action)
            actionout = open(actionoutPath, 'a+')
            #
print("??????????????????????????????????????????????????????????????????????????????????????????????????
??")
            print('action: ', action, 'lastaction: ', last_action, file=actionout)
            sys.stdout.flush()
            actionout.close()
            next_state, reward, done, _ = env.step(action)
```

```
            last_action = action

            total_rewards += reward
            if JUSTTEST == 0:
                if state[-1] != -1 and next_state[-1] != -1:
                    q_learner.storeExperience(state, action, reward, next_state, done)
                q_learner.updateModel(i_episode)
            state = next_state

            if done:
                if reward >= 500 - 1:
                    wintimes += 1
                else:
                    record_t = -1
                break

        episode_history.append(wintimes)
        # mean_rewards = np.mean(episode_history)

        print("Episode {}".format(i_episode))
        #print("Finished after {} timesteps".format(t + 1))
        print("Reward for this episode: {}".format(total_rewards))
        # print("last 99 episodes wintimes:    ", episode_history[-1][0] - episode_history[0][0])
        out_wintimes = open(out_wintimes_path, "a+")
        print("Episode: ", i_episode, " ", q_learner.exploration, " ", "last 99 episodes wintimes:
",
              episode_history[-1] - episode_history[0], 'step: ', record_t,
              file=out_wintimes)
        sys.stdout.flush()
        out_wintimes.close()
        # print("Average reward for last 100 episodes: {}".format(mean_rewards))
        if JUSTTEST == 0:
            if i_episode >= 200 and i_episode % 200 == 0:
                path = 'my_net22/' + env_name + '_' + str(num_layers) \
                       + 'layers_' + str(i_episode + 1) + 'epsiode_' + \
                       strftime("%Y-%m-%d-%H-%M-%S",          gmtime())          +
'restore_network_rerandom'
                q_learner.save_net(path)
                testtest(str(i_episode + 1))、
```