ДОДАТОК А

Модуль алгоритм QR-DQN з декількома параметрами входу

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import cv2
import random
import datetime
import os
import time

from mlagents_envs.side_channel.engine_configuration_channel import EngineConfigurationChannel
from mlagents_envs.environment import UnityEnvironment
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"

my_devices = tf.config.experimental.list_physical_devices(device_type='CPU')
tf.config.experimental.set_visible_devices(devices= my_devices, device_type='CPU')

%matplotlib inline
os_ = "Windows"

env_name = "Y:/DRL_based_SelfDrivingCarControl-master/environment/" + os_ + "/Driving" # Name of the Unity environment binary to
launch

train_mode = True # Whether to run the environment in training or inference mode

channel = EngineConfigurationChannel()
env = UnityEnvironment(base_port = 8888, file_name=env_name, worker_id=1, seed=1, side_channels=[channel])

print(str(env))
default_brain = env.brain_names[0]
brain = env.brains[default_brain]

env_info = env.reset(train_mode=train_mode)[default_brain]

print("Sensor data (LIDAR): \n{}".format(env_info.vector_observations[0]))

Num_obs = len(env_info.visual_observations)

print("Image data (Front Camera): \n{}")
if Num_obs > 1:
    f, axarr = plt.subplots(1, Num_obs, figsize=(20,10))
    for i, observation in enumerate(env_info.visual_observations):
        if observation.shape[3] == 3:
            axarr[i].imshow(observation[0,:,:,:])
            axarr[i].axis('off')
        else:
            axarr[i].imshow(observation[0,:,:,0])
            axarr[i].axis('off')
else:
    f, axarr = plt.subplots(1, Num_obs)
    for i, observation in enumerate(env_info.visual_observations):
        if observation.shape[3] == 3:
            axarr.imshow(observation[0,:,:,:])
            axarr.axis('off')
        else:
            axarr.imshow(observation[0,:,:,0])
            axarr.axis('off')
algorithm = 'QR-DQN'
Num_action = brain.vector_action_space_size[0]

# QR-DQN Parameter
Num_quantile = 50

# parameter for DQN
Num_replay_memory = 100000
Num_start_training = 50000
Num_training = 1000000
Num_update = 10000
Num_batch = 32
Num_test = 100000
Num_skipFrame = 4
Num_stackFrame = 4
```

```
Num_colorChannel = 1

Epsilon = 1.0
Final_epsilon = 0.1
Gamma = 0.99
Learning_rate = 0.00005

# Parameter for LSTM
Num_dataSize = 366
Num_cellState = 512

# Parameters for network
img_size = 80
sensor_size = 360

first_conv   = [8,8,Num_colorChannel * Num_stackFrame * Num_obs,32]
second_conv  = [4,4,32,64]
third_conv   = [3,3,64,64]
first_dense  = [10*10*64 + Num_cellState, 512]
second_dense = [first_dense[1], Num_action * Num_quantile]

# Path of the network model
load_path = 'Y:\DRL_based_SelfDrivingCarControl-master\saved_networks\2018-09-13_17_13_QR-DQN_both\model.ckpt'

# Parameters for session
Num_plot_episode = 5
Num_step_save = 50000

GPU_fraction = 0.4
# Initialize weights and bias
def weight_variable(shape):
return tf.Variable(xavier_initializer(shape))

def bias_variable(shape):
return tf.Variable(xavier_initializer(shape))

# Xavier Weights initializer
def xavier_initializer(shape):
dim_sum = np.sum(shape)
if len(shape) == 1:
dim_sum += 1
bound = np.sqrt(2.0 / dim_sum)
return tf.random_uniform(shape, minval=-bound, maxval=bound)

# Convolution function
def conv2d(x,w, stride):
return tf.nn.conv2d(x,w,strides=[1, stride, stride, 1], padding='SAME')

# Assign network variables to target network
def assign_network_to_target():
# Get trainable variables
trainable_variables = tf.trainable_variables()
# network lstm variables
trainable_variables_network = [var for var in trainable_variables if var.name.startswith('network')]

# target lstm variables
trainable_variables_target = [var for var in trainable_variables if var.name.startswith('target')]

# assign network variables to target network
for i in range(len(trainable_variables_network)):
sess.run(tf.assign(trainable_variables_target[i], trainable_variables_network[i]))

# Code for tensorboard
def setup_summary():
episode_speed      = tf.Variable(0.)
episode_overtake   = tf.Variable(0.)
episode_lanechange = tf.Variable(0.)

tf.summary.scalar('Average_Speed/' + str(Num_plot_episode) + 'episodes', episode_speed)
tf.summary.scalar('Average_overtake/' + str(Num_plot_episode) + 'episodes', episode_overtake)
tf.summary.scalar('Average_lanechange/' + str(Num_plot_episode) + 'episodes', episode_lanechange)

summary_vars = [episode_speed, episode_overtake, episode_lanechange]
summary_placeholders = [tf.placeholder(tf.float32) for _ in range(len(summary_vars))]
update_ops = [summary_vars[i].assign(summary_placeholders[i]) for i in range(len(summary_vars))]
summary_op = tf.summary.merge_all()
return summary_placeholders, update_ops, summary_op
tf.reset_default_graph()
```

```
# Input
x_image = tf.placeholder(tf.float32, shape = [None, img_size, img_size, Num_colorChannel * Num_stackFrame * Num_obs])
x_normalize = (x_image - (255.0/2)) / (255.0/2)

x_sensor = tf.placeholder(tf.float32, shape = [None, Num_stackFrame, Num_dataSize])
x_unstack = tf.unstack(x_sensor, axis = 1)

with tf.variable_scope('network'):
# Convolution variables
w_conv1 = weight_variable(first_conv)
b_conv1 = bias_variable([first_conv[3]])

w_conv2 = weight_variable(second_conv)
b_conv2 = bias_variable([second_conv[3]])

w_conv3 = weight_variable(third_conv)
b_conv3 = bias_variable([third_conv[3]])

# Densely connect layer variables
w_fc1 = weight_variable(first_dense)
b_fc1 = bias_variable([first_dense[1]])

w_fc2 = weight_variable(second_dense)
b_fc2 = bias_variable([second_dense[1]])

# LSTM cell
cell = tf.contrib.rnn.BasicLSTMCell(num_units = Num_cellState)
rnn_out, rnn_state = tf.nn.static_rnn(inputs = x_unstack, cell = cell, dtype = tf.float32)

# Network
h_conv1 = tf.nn.relu(conv2d(x_normalize, w_conv1, 4) + b_conv1)
h_conv2 = tf.nn.relu(conv2d(h_conv1, w_conv2, 2) + b_conv2)
h_conv3 = tf.nn.relu(conv2d(h_conv2, w_conv3, 1) + b_conv3)

h_pool3_flat = tf.reshape(h_conv3, [-1, 10 * 10 * 64])
rnn_out = rnn_out[-1]
h_concat = tf.concat([h_pool3_flat, rnn_out], axis = 1)

h_fc1 = tf.nn.relu(tf.matmul(h_concat, w_fc1)+b_fc1)

# Get Q value for each action
logits = tf.matmul(h_fc1, w_fc2) + b_fc2
logits_reshape = tf.reshape(logits, [-1, Num_action, Num_quantile])
Q_action = tf.reduce_sum(tf.multiply(1/Num_quantile, logits_reshape), axis = 2)

with tf.variable_scope('target'):
# Convolution variables target
w_conv1_target = weight_variable(first_conv)
b_conv1_target = bias_variable([first_conv[3]])

w_conv2_target = weight_variable(second_conv)
b_conv2_target = bias_variable([second_conv[3]])

w_conv3_target = weight_variable(third_conv)
b_conv3_target = bias_variable([third_conv[3]])

# Densely connect layer variables target
w_fc1_target = weight_variable(first_dense)
b_fc1_target = bias_variable([first_dense[1]])

w_fc2_target = weight_variable(second_dense)
b_fc2_target = bias_variable([second_dense[1]])

# LSTM cell
cell_target = tf.contrib.rnn.BasicLSTMCell(num_units = Num_cellState)
rnn_out_target, rnn_state_target = tf.nn.static_rnn(inputs = x_unstack, cell = cell_target, dtype = tf.float32)

# Target Network
h_conv1_target = tf.nn.relu(conv2d(x_normalize, w_conv1_target, 4) + b_conv1_target)
h_conv2_target = tf.nn.relu(conv2d(h_conv1_target, w_conv2_target, 2) + b_conv2_target)
h_conv3_target = tf.nn.relu(conv2d(h_conv2_target, w_conv3_target, 1) + b_conv3_target)

h_pool3_flat_target = tf.reshape(h_conv3_target, [-1, 10 * 10 * 64])
rnn_out_target = rnn_out_target[-1]
h_concat_target = tf.concat([h_pool3_flat_target, rnn_out_target], axis = 1)

h_fc1_target  = tf.nn.relu(tf.matmul(h_concat_target, w_fc1_target)+b_fc1_target)
```

```
# Get Q value for each action
logits_target = tf.matmul(h_fc1, w_fc2_target) + b_fc2_target
logits_reshape_target = tf.reshape(logits_target, [-1, Num_action, Num_quantile])
Q_action_target = tf.reduce_sum(tf.multiply(1/Num_quantile, logits_reshape_target), axis = 2)
# Loss function and Train
theta_loss = tf.placeholder(tf.float32, shape = [None, Num_quantile])
action_binary_loss = tf.placeholder(tf.float32, shape = [None, Num_action, Num_quantile])

# Get valid logits
logit_valid = tf.multiply(logits_reshape, action_binary_loss)
logit_valid_nonzero = tf.reduce_sum(logit_valid, axis = 1)

# Stack i and j
theta_loss_tile = tf.tile(tf.expand_dims(theta_loss, axis=2), [1, 1, Num_quantile])
logit_valid_tile = tf.tile(tf.expand_dims(logit_valid_nonzero, axis=1), [1, Num_quantile, 1])

error_loss = theta_loss_tile - logit_valid_tile

# Get Huber loss
Huber_loss = tf.losses.huber_loss(theta_loss_tile, logit_valid_tile, reduction = tf.losses.Reduction.NONE)

# Get tau
min_tau = 1/(2*Num_quantile)
max_tau = (2*(Num_quantile-1)+3)/(2*Num_quantile)
tau = tf.reshape (tf.range(min_tau, max_tau, 1/Num_quantile), [1, Num_quantile])
inv_tau = 1.0 - tau

# Get Loss
Loss = tf.where(tf.less(error_loss, 0.0), inv_tau * Huber_loss, tau * Huber_loss)
Loss = tf.reduce_mean(tf.reduce_sum(tf.reduce_mean(Loss, axis = 2), axis = 1))

train_step = tf.train.AdamOptimizer(learning_rate = Learning_rate, epsilon = 1e-02/Num_batch).minimize(Loss)
## Initialize variables
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = GPU_fraction

sess = tf.InteractiveSession(config=config)

init = tf.global_variables_initializer()
sess.run(init)
# Load the file if the saved file exists
saver = tf.train.Saver()

# check_save = 1
check_save = input('Inference? / Training?(1=Inference/2=Training): ')

if check_save == '1':
# Directly start inference
Num_start_training = 0
Num_training = 0

# Restore variables from disk.
saver.restore(sess, load_path)
print("Model restored.")

# date - hour - minute of training time
date_time = str(datetime.date.today()) + '_' + str(datetime.datetime.now().hour) + '_' + str(datetime.datetime.now().minute)

# Make folder for save data
os.makedirs('Y:\DRL_based_SelfDrivingCarControl-master\saved_networks' + date_time + '_' + algorithm + '_both')

# Summary for tensorboard
summary_placeholders, update_ops, summary_op = setup_summary()
summary_writer = tf.summary.FileWriter('Y:\DRL_based_SelfDrivingCarControl-master\saved_networks' + date_time + '_' + algorithm + '_both',
sess.graph)
# Initialize input
def input_initialization(env_info):
# Observation
observation_stack_obs = np.zeros([img_size, img_size, Num_colorChannel * Num_obs])

for i in range(Num_obs):
observation = 255 * env_info.visual_observations[i]
observation = np.uint8(observation)
observation = np.reshape(observation, (observation.shape[1], observation.shape[2], 3))
observation = cv2.resize(observation, (img_size, img_size))

if Num_colorChannel == 1:
```

```
observation = cv2.cvtColor(observation, cv2.COLOR_RGB2GRAY)
observation = np.reshape(observation, (img_size, img_size))

if Num_colorChannel == 3:
observation_stack_obs[:,:, Num_colorChannel * i: Num_colorChannel * (i+1)] = observation
else:
observation_stack_obs[:,:, i] = observation

observation_set = []

# State
state = env_info.vector_observations[0][:-7]
state_set = []

for i in range(Num_skipFrame * Num_stackFrame):
observation_set.append(observation_stack_obs)
state_set.append(state)

# Stack the frame according to the number of skipping and stacking frames using observation set
observation_stack = np.zeros((img_size, img_size, Num_colorChannel * Num_stackFrame * Num_obs))
state_stack = np.zeros((Num_stackFrame, Num_dataSize))

for stack_frame in range(Num_stackFrame):
observation_stack[:,:,Num_obs * stack_frame: Num_obs * (stack_frame+1)] = observation_set[-1 - (Num_skipFrame * stack_frame)]
state_stack[(Num_stackFrame - 1) - stack_frame, :] = state_set[-1 - (Num_skipFrame * stack_frame)]

observation_stack = np.uint8(observation_stack)
state_stack = np.uint8(state_stack)

return observation_stack, observation_set, state_stack, state_set

# Resize input information
def resize_input(env_info, observation_set, state_set):
# Stack observation according to the number of observations
observation_stack_obs = np.zeros([img_size, img_size, Num_colorChannel * Num_obs])

for i in range(Num_obs):
observation = 255 * env_info.visual_observations[i]
observation = np.uint8(observation)
observation = np.reshape(observation, (observation.shape[1], observation.shape[2], 3))
observation = cv2.resize(observation, (img_size, img_size))

if Num_colorChannel == 1:
observation = cv2.cvtColor(observation, cv2.COLOR_RGB2GRAY)
observation = np.reshape(observation, (img_size, img_size))

if Num_colorChannel == 3:
observation_stack_obs[:,:, Num_colorChannel * i: Num_colorChannel * (i+1)] = observation
else:
observation_stack_obs[:,:,i] = observation

# Add observations to the observation_set
observation_set.append(observation_stack_obs)

# State
state = env_info.vector_observations[0][:-7]

# Add state to the state_set
state_set.append(state)

# Stack the frame according to the number of skipping and stacking frames using observation set
observation_stack = np.zeros((img_size, img_size, Num_colorChannel * Num_stackFrame * Num_obs))
state_stack = np.zeros((Num_stackFrame, Num_dataSize))

for stack_frame in range(Num_stackFrame):
observation_stack[:,:,Num_obs * stack_frame: Num_obs * (stack_frame+1)] = observation_set[-1 - (Num_skipFrame * stack_frame)]
state_stack[(Num_stackFrame - 1) - stack_frame, :] = state_set[-1 - (Num_skipFrame * stack_frame)]

del observation_set[0]
del state_set[0]

observation_stack = np.uint8(observation_stack)
state_stack = np.uint8(state_stack)

return observation_stack, observation_set, state_stack, state_set

# Get progress according to the number of steps
def get_progress(step, Epsilon):
```

```
if step <= Num_start_training:
# Observation
progress = 'Observing'
train_mode = True
Epsilon = 1
elif step <= Num_start_training + Num_training:
# Training
progress = 'Training'
train_mode = True

# Decrease the epsilon value
if Epsilon > Final_epsilon:
Epsilon -= 1.0/Num_training
elif step < Num_start_training + Num_training + Num_test:
# Testing
progress = 'Testing'
train_mode = False
Epsilon = 0
else:
# Finished
progress = 'Finished'
train_mode = False
Epsilon = 0

return progress, train_mode, Epsilon

# Select action according to the progress of training
def select_action(progress, sess, observation_stack, state_stack, Epsilon):
if progress == "Observing":
# Random action
Q_value = 0
action = np.zeros([Num_action])
action[random.randint(0, Num_action - 1)] = 1.0
elif progress == "Training":
# if random value(0-1) is smaller than Epsilon, action is random.
# Otherwise, action is the one which has the max Q value
if random.random() < Epsilon:
Q_value = 0
action = np.zeros([Num_action])
action[random.randint(0, Num_action - 1)] = 1
else:
Q_value = Q_action.eval(feed_dict={x_image: [observation_stack], x_sensor: [state_stack]})
action = np.zeros([Num_action])
action[np.argmax(Q_value)] = 1
else:
# Max Q action
Q_value = Q_action.eval(feed_dict={x_image: [observation_stack], x_sensor: [state_stack]})
action = np.zeros([Num_action])
action[np.argmax(Q_value)] = 1

return action, Q_value

def train(Replay_memory, sess, step):
# Select minibatch
minibatch =  random.sample(Replay_memory, Num_batch)

# Save the each batch data
observation_batch      = [batch[0] for batch in minibatch]
state_batch            = [batch[1] for batch in minibatch]
action_batch           = [batch[2] for batch in minibatch]
reward_batch           = [batch[3] for batch in minibatch]
observation_next_batch = [batch[4] for batch in minibatch]
state_next_batch       = [batch[5] for batch in minibatch]
terminal_batch         = [batch[6] for batch in minibatch]

# Update target network according to the Num_update value
if step % Num_update == 0:
assign_network_to_target()


# Get Target
Q_batch = Q_action.eval(feed_dict = {x_image: observation_next_batch, x_sensor: state_next_batch})
theta_batch = logits_reshape_target.eval(feed_dict = {x_image: observation_next_batch, x_sensor: state_next_batch})

theta_target = []

for i in range(len(minibatch)):
theta_target.append([])
```

```
for j in range(Num_quantile):
if terminal_batch[i] == True:
theta_target[i].append(reward_batch[i])
else:
theta_target[i].append(reward_batch[i] + Gamma * theta_batch[i, np.argmax(Q_batch[i]), j])

# Calculate action binary
action_binary = np.zeros([Num_batch, Num_action, Num_quantile])

for i in range(len(action_batch)):
action_batch_max = np.argmax(action_batch[i])
action_binary[i, action_batch_max, :] = 1

_, loss = sess.run([train_step, Loss],feed_dict = {x_image: observation_batch,
x_sensor: state_batch,
theta_loss: theta_target,
action_binary_loss: action_binary})

# Experience Replay
def Experience_Replay(progress, Replay_memory, obs_stack, s_stack, action, reward, next_obs_stack, next_s_stack, terminal):
if progress != 'Testing':
# If length of replay memeory is more than the setting value then remove the first one
if len(Replay_memory) > Num_replay_memory:
del Replay_memory[0]

# Save experience to the Replay memory
Replay_memory.append([obs_stack, s_stack, action, reward, next_obs_stack, next_s_stack, terminal])
else:
# Empty the replay memory if testing
Replay_memory = []

return Replay_memory
# Initial parameters
Replay_memory = []

step = 1
score = 0
score_board = 0

episode = 0
step_per_episode = 0

speed_list = []
overtake_list = []
lanechange_list = []

train_mode = True
env_info = env.reset(train_mode=train_mode)[default_brain]

observation_stack, observation_set, state_stack, state_set = input_initialization(env_info)
check_plot = 0

# Training & Testing
while True:

# Get Progress, train mode
progress, train_mode, Epsilon  = get_progress(step, Epsilon)

# Select Actions
action, Q_value = select_action(progress, sess, observation_stack, state_stack, Epsilon)
action_in = [np.argmax(action)]

# Get information for plotting
vehicle_speed  = 100 * env_info.vector_observations[0][-8]
num_overtake   = env_info.vector_observations[0][-7]
num_lanechange = env_info.vector_observations[0][-6]

# Get information for update
env_info = env.step(action_in)[default_brain]

next_observation_stack, observation_set, next_state_stack, state_set = resize_input(env_info, observation_set, state_set)
reward = env_info.rewards[0]
terminal = env_info.local_done[0]

if progress == 'Training':
# Train!!
train(Replay_memory, sess, step)
```

```
# Save the variables to disk.
if step == Num_start_training + Num_training:
save_path = saver.save(sess, '../saved_networks/' + date_time + '_' + algorithm + '_both' + "/model.ckpt")
print("Model saved in file: %s" % save_path)

# If progress is finished -> close!
if progress == 'Finished':
print('Finished!!')
env.close()
break

Replay_memory = Experience_Replay(progress,
Replay_memory,
observation_stack,
state_stack,
action,
reward,
next_observation_stack,
next_state_stack,
terminal)

# Update information
step += 1
score += reward
step_per_episode += 1

observation_stack = next_observation_stack
state_stack = next_state_stack

# Update tensorboard
if progress != 'Observing':
speed_list.append(vehicle_speed)

if episode % Num_plot_episode == 0 and check_plot == 1 and episode != 0:
avg_speed      = sum(speed_list) / len(speed_list)
avg_overtake   = sum(overtake_list) / len(overtake_list)
avg_lanechange = sum(lanechange_list) / len(lanechange_list)

tensorboard_info = [avg_speed, avg_overtake, avg_lanechange]
for i in range(len(tensorboard_info)):
sess.run(update_ops[i], feed_dict = {summary_placeholders[i]: float(tensorboard_info[i])})
summary_str = sess.run(summary_op)
summary_writer.add_summary(summary_str, step)
score_board = 0

speed_list = []
overtake_list = []
lanechange_list = []

check_plot = 0

# If terminal is True
if terminal == True:
# Print informations
print('step: ' + str(step) + ' / '  + 'episode: ' + str(episode) + ' / ' + 'progress: ' + progress  + ' / ' + 'epsilon: ' + str(Epsilon)  +' / ' + 'score: ' +
str(score))

check_plot = 1

if progress != 'Observing':
episode += 1

score_board += score
overtake_list.append(num_overtake)
lanechange_list.append(num_lanechange)


score = 0
step_per_episode = 0

# Initialize game state
env_info = env.reset(train_mode=train_mode)[default_brain]
observation_stack, observation_set, state_stack, state_set = input_initialization(env_info)
```

ВІДОМІСТЬ АТЕСТАЦІЙНОЇ РОБОТИ

Позначення	Найменування	Дод. відомості
	Текстові документи	
1	Пояснювальна записка	69 с.
2	Презентаційний матеріал	23 с.
	Інші документи	
3	Роздруківки програм	8 с.
4	Рецензія	2 с.
5	Відгук керівника	1 с.

<table>
<tr><td></td><td></td><td></td><td></td><td></td><td colspan="5" rowspan="3">Математичні моделі та методи навчання
з підкріпленням для задачи автономного
водіння автомобіля</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Змін</td><td>Арк.</td><td>Номер докум.</td><td>Підп.</td><td>Дата</td></tr>
<tr><td colspan="2">Розроб.</td><td>Кравченко М.О.</td><td></td><td></td><td rowspan="4">(Тема роботи)
Відомість
атестаційної
роботи</td><td colspan="3"></td><td>Аркуш</td><td>Аркушів</td></tr>
<tr><td colspan="2">Перевір.</td><td>Єсілевський В.С.</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="2">Н. контр.</td><td>Сидоров М.В.</td><td></td><td></td><td colspan="5" rowspan="2">ХНУРЕ
кафедра ПМ</td></tr>
<tr><td colspan="2">Затв.</td><td>Тевяшев А.Д.</td><td></td><td></td></tr>
</table>