rl_traffic

mkorecki

Jan 25, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

This is the documentation for a traffic simulation code based on cityflow used to test and compare different approaches to traffic lights management.

To run the simulation, change directory to the src folder and call:

```
python3 traffic_sim.py
```

This would run the simulation with the default arguments. If you want to specify your own arguments here is what you have at your disposal with example uses:

–sim_config "../4x4/1.config"
The path to the cityflow config file for the simulation you want to run

–num_episodes 1
The number of episodes you want your learning algorithm to learn for

–num_sim_steps 1800
Number of simulation steps you want the simulation to run for in a single episode

–agents_type "learning"
The type of agents you want to run, for now available options are: learning or analytical

–update_freq 10
How often the reinforcement learning agent updates its q-network

–batch_size 64
The size of the mini-batch used to train the deep-q-network

–lr 5e-4
The learning rate

Example call would look like this:

```
python3 traffic_sim.py --agents_type 'analytical' --sim_config '../4x4/1.config' --
↪num_episodes 1 --num_sim_steps 1800
```

The docs folder contains this documentation.

CHAPTER

TWO

CODE

Here follows the documentation of the python code of the traffic simulation based on cityflow and used to test, compare and improve different algorithms for managing traffic lights operation.

2.1 environ

class `environ.`**`Environment`**(*args*, *n_actions=9*, *n_states=44*)

The class Environment represents the environment in which the agents operate in this case it is a city consisting of roads, lanes and intersections which are controled by the agents

`analytical_step`(*time*, *done*)

represents a single step of the simulation for the analytical agent :param time: the current timestep :param done: flag indicating weather this has been the last step of the episode, used for learning, here for interchangability of the two steps

`learning_step`(*time*, *done*)

represents a single step of the simulation for the learning agent :param time: the current timestep :param done: flag indicating weather this has been the last step of the episode

`reset`()

resets the movements for each agent and the simulation environment, should be called after each episode

2.2 logger

class `logger.`**`Logger`**(*args*)

The Logger class is responsible for logging data, building representations and saving them in a specified location

`__init__`(*args*)

Initialises the logger object :param args: the arguments passed by the user

`log_measures`(*environ*)

Logs measures such as reward, vehicle count, average travel time and q losses, works for learning agents and aggregates over episodes :param environ: the environment in which the model was run

`save_log_file`(*environ*)

Creates and saves a log file with information about the experiment in a .txt format :param environ: the environment in which the model was run

`save_measures_plots`()

Saves plots containing the measures such as vehicle count, travel time, rewards and q losses The data is over the episodes and for now works only with learning agents

save_models(*environ*)
Saves machine learning models (for now just neural networks) :param environ: the environment in which the model was run

2.3 intersection

class intersection.**Movement**(*ID*, *in_road*, *out_road*, *in_lanes*, *out_lanes*, *lane_length*, *phases=[]*, *clearing_time=2*)
The class defining a Movement on an intersection, a Movement of vehicles from incoming road -> outgoing road

__init__(*ID*, *in_road*, *out_road*, *in_lanes*, *out_lanes*, *lane_length*, *phases=[]*, *clearing_time=2*)
initialises the Movement, the movement has a type 1, 2 or 3 1 -> turn right, 2 -> turn left, 3 -> go straight :param ID: the unique (for a given intersection) ID associated with the movement :param in_road: the incoming road of the movement :param out_road: the outgoing road of the movement :param in_lanes: the incoming lanes of the movement :param out_lanes: the outgoing lanes of the movement :param lane_length: the length of the incoming lane (if there is more than one incoming lane we assume they have the same length) :param phases: the indices of phases for which the give movement is enabled

get_arr_veh_num(*start_time*, *end_time*)
gets the number of vehicles arrived to the movement's lanes in a given interval :param start_time: the start of the time interval :param end_time: the end of the time interval :returns: the number of vehicles arrived in the interval

get_demand(*eng*, *lanes_count*)
Gets the demand of the incoming lanes of the movement the demand is the sum of the vehicles on all incoming lanes :param eng: the cityflow simulation engine :param lanes_vehs: a dictionary with lane ids as keys and number of vehicles as values :returns: the demand of the movement

get_dep_veh_num(*start_time*, *end_time*)
gets the number of vehicles departed from the movement's lanes in a given interval :param start_time: the start of the time interval :param end_time: the end of the time interval :returns: the number of vehicles departed in the interval

get_green_time(*time*, *current_movements*)
Gets the predicted green time needed to clear the movement :param time: the current timestep :param current_movements: a list of movements that are currently enabled :returns: the predicted green time of the movement

get_pressure(*eng*, *lanes_count*)
Gets the pressure of the movement, the pressure is defined in traffic RL publications from PenState :param eng: the cityflow simulation engine :param lanes_vehs: a dictionary with lane ids as keys and number of vehicles as values :returns: the pressure of the movement

update_arr_dep_veh_num(*lanes_vehs*)
Updates the list containing the number vehicles that arrived and departed :param lanes_vehs: a dictionary with lane ids as keys and number of vehicles as values

update_wait_time(*action*, *green_time*, *waiting_vehs*)
Updates movement's waiting time - the time a given movement has waited to be enabled :param action: the phase to be chosen by the intersection :param green_time: the green time the action is going to be enabled for :param waiting_vehs: a dictionary with lane ids as keys and number of waiting cars as values

class intersection.**Phase**(*ID=''*, *movements=[]*)
The class defining a Phase on an intersection, a Phase is defined by Movements which are enabled by it (given the green light)

`__init__`(*ID=''*, *movements=[]*)
initialises the Phase :param ID: the unique (for a given intersection) ID associated with the Phase, used by the engine to set phase on :param movements: the indeces of Movements which the Phase enables

2.4 agent

class `agent.Agent`(*ID*)
The base clase of an Agent, Learning and Analytical agents derive from it, basically defines methods used by both types of agents

`__init__`(*ID*)
initialises the Agent :param ID: the unique ID of the agent corresponding to the ID of the intersection it represents

`init_movements`(*eng*)
initialises the movements of the Agent based on the lane links extracted from the simulation roadnet the eng.get_intersection_lane_links used in the method takes the intersection ID and returns a tuple containing the (in_road, out_road) pair as the first element and (in_lanes, out_lanes) as the second element :param eng: the cityflow simulation engine

`init_phases`(*eng*)
initialises the phases of the Agent based on the intersection phases extracted from the simulation data :param eng: the cityflow simulation engine

`reset_movements`()
Resets the set containing the vehicle ids for each movement and the arr/dep vehicles numbers as well as the waiting times the set represents the vehicles waiting on incoming lanes of the movement

`update_arr_dep_veh_num`(*lanes_vehs*)
Updates the list containing the number vehicles that arrived and departed :param lanes_vehs: a dictionary with lane ids as keys and number of vehicles as values

`update_wait_time`(*action*, *green_time*, *waiting_vehs*)
Updates movements' waiting time - the time a given movement has waited to be enabled :param action: the phase to be chosen by the intersection :param green_time: the green time the action is going to be enabled for :param waiting_vehs: a dictionary with lane ids as keys and number of waiting cars as values

2.5 analytical_agent

class `analytical_agent.Analytical_Agent`(*eng*, *ID=''*)
The class defining an agent which controls the traffic lights using the analytical approach from Helbing, Lammer's works

`__init__`(*eng*, *ID=''*)
initialises the Analytical Agent :param ID: the unique ID of the agent corresponding to the ID of the intersection it represents :param eng: the cityflow simulation engine

`act`(*eng*, *time*)
selects the next action - phase for the agent to select along with the time it should stay on for :param eng: the cityflow simulation engine :param time: the time in the simulation, at this moment only integer values are supported :returns: the phase and the green time

`set_phase`(*eng*, *phase*)
sets the phase of the agent to the indicated phase :param eng: the cityflow simulation engine :param phase: the phase object, its ID corresponds to the phase ID in the simulation envirionment

stabilise (*time*)
Implements the stabilisation mechanism of the algorithm, updates the action queue with phases that need to be prioritiesd :param time: the time in the simulation, at this moment only integer values are supported

update_clear_green_time (*time*)
Updates the green times of the movements of the intersection :param time: the time in the simulation, at this moment only integer values are supported

update_priority_idx (*time*)
Updates the priority of the movements of the intersection, the higher priority the more the movement needs to get a green lights :param time: the time in the simulation, at this moment only integer values are supported

2.6 learning_agent

class learning_agent.**Learning_Agent** (*eng*, *ID=''*, *in_roads=[]*, *out_roads=[]*)
The class defining an agent which controls the traffic lights using reinforcement learning approach called PressureLight

__init__ (*eng*, *ID=''*, *in_roads=[]*, *out_roads=[]*)
initialises the Learning Agent :param ID: the unique ID of the agent corresponding to the ID of the intersection it represents :param eng: the cityflow simulation engine

act (*net_local*, *state*, *eps=0*, *n_actions=8*)
generates the action to be taken by the agent :param net_local: the neural network used in the decision making process :param state: the current state of the intersection, given by observe :param eps: the epsilon value used in the epsilon greedy learing :param n_actions: number of actions to choose from

get_in_lanes_veh_num (*eng*, *lanes_count*)
gets the number of vehicles on the incoming lanes of the intersection :param eng: the cityflow simulation engine :param lanes_count: a dictionary with lane ids as keys and vehicle count as values

get_out_lanes_veh_num (*eng*, *lanes_count*)
gets the number of vehicles on the outgoing lanes of the intersection :param eng: the cityflow simulation engine :param lanes_count: a dictionary with lane ids as keys and vehicle count as values

get_reward (*eng*, *time*, *lanes_count*)
gets the reward of the agent in the form of pressure :param eng: the cityflow simulation engine :param time: the time of the simulation :param lanes_count: a dictionary with lane ids as keys and vehicle count as values

init_phases_vectors (*eng*)
initialises vector representation of the phases :param eng: the cityflow simulation engine

observe (*eng*, *time*, *lanes_count*)
generates the observations made by the agents :param eng: the cityflow simulation engine :param time: the time of the simulation :param lanes_count: a dictionary with lane ids as keys and vehicle count as values

set_phase (*eng*, *phase*)
sets the phase of the agent to the indicated phase :param eng: the cityflow simulation engine :param phase: the phase object, its ID corresponds to the phase ID in the simulation envirionment

2.7 dqn

class dqn.**DQN**(*state_size*, *action_size*, *seed=2*, *fc1_unit=128*, *fc2_unit=64*)
Actor (Policy) Model.

__init__(*state_size*, *action_size*, *seed=2*, *fc1_unit=128*, *fc2_unit=64*)
Initialize parameters and build model. Params =======

state_size (int): Dimension of each state action_size (int): Dimension of each action seed (int): Random seed fc1_unit (int): Number of nodes in first hidden layer fc2_unit (int): Number of nodes in second hidden layer

forward(*x*)
Build a network that maps state -> action values.

class dqn.**ReplayMemory**(*action_size*, *buffer_size=100000*, *batch_size=64*, *seed=2*)
Fixed -size buffe to store experience tuples.

__init__(*action_size*, *buffer_size=100000*, *batch_size=64*, *seed=2*)
Initialize a ReplayBuffer object.

action_size (int): dimension of each action buffer_size (int): maximum size of buffer batch_size (int): size of each training batch seed (int): random seed

__len__()
Return the current size of internal memory.

add(*state*, *action*, *reward*, *next_state*, *done*)
Add a new experience to memory.

sample()
Randomly sample a batch of experiences from memory

dqn.**optimize_model**(*experiences*, *net_local*, *net_target*, *optimizer*, *gamma=0.999*)
Update value parameters using given batch of experience tuples.

experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', done) tuples

gamma (float): discount factor

dqn.**soft_update**(*local_model*, *target_model*, *tau*)
Soft update model parameters. _target = *_local + (1 -)*_target

local model (PyTorch model): weights will be copied from target model (PyTorch model): weights will be copied to tau (float): interpolation parameter

PYTHON MODULE INDEX

INDEX

O

P

R

S

U