SIM2REAL

Olivier Sutter Mathieu Verm Michael Romagne
Alexandre Martin Pablo Mirales Vincent Coyette

January 22, 2020

Contents

1 Introduction

Reinforcement learing is an area of machine learning. The goal is for an agent to learn an optimal behaviour to adopt in a given environment based on experience. Reinforcement Learning has been very successful in the last few years, notably for its great performance in games (e.g. AlphaGo[1]). However, it suffers from a lack of industrial applications.

One of the reason for this is the fact that an agent needs a lot of experiences in order to learn an acceptable behaviour. Those simulations must be run in a simulator for security, speed and economic reasons. The transfer of behaviour from the simulated agent to the real world (for example an autonomous vehicle) is far from being trivial. Indeed, even for high quality simulators, there is a shift in both spaces of states and transitions between the simulator and the real world.

The goal of this project is to study different methods to robustly train an autonomous car in a simulator, in order for the real-world agent to behave as suited.

DuckieTown[2] environment will be used during all the project. It provides both a simulator (Gym-Duckietown) and real-world agent and environment.

2 Environment

An environment in which a Reinforcement Learning agent is trained is modelled as an Markov Decision Process (MDP). An MDP is a 4-tuple (S, A, P_a, R_a). Let's detail each of these component for duckietown.

2.1 State

In an MDP, the state is defined as the state of the environment as it is seen by the agent. Here, the robot contains only one captor, the camera. The camera has a resolution of 160x120, and thus the state space is $S = [0, 255]^{160x120x3}$ (3 is for the 3 RGB colors).

[1]Silver, D., Schrittwieser, J., Simonyan, K. et al. Mastering the game of Go without human knowledge. Nature 550, 354–359 (2017)

[2]See https://www.duckietown.org/

2.2 Actions

The actions the environment can take are composed by a forward velocity and a steering angle. By default, the action space is continuous, $A = [-1, 1]^2$:

- The first number correspond to the forward velocity. 1 is for going full speed forward, -1 is for full speed backward.
- The second number correspond to the steering angle. 1 is for the steering wheel being fully to the left.

The action space can be changed to be discrete. In this case, the possible actions would be move forward, turn left or turn rigth.

2.3 Transitions

The transition of a MDP is the distribution $P(s'|s, a)$, i.e. the probability of ending up in the state s' if the agent is in the state s and take the action a. The transitions would typically be different inside the simulator and in the real life.

Inside the simulator, the transitions are considered as perfect. The agent being in an initial position and the taken action being $speed, angle$, the simulator will update the position of the agent accordingly. The state s' is then the new camera signal. The state is not fully deterministic, as some objects (such as duckie pedestrians for example, may have a stocastic behaviour and appear on the camera), but the updated position of the agent is a deterministic function of the current position and the current action.

In the real world however, the transitions of positions are not perfect. Given a position and an action, the next position is not deterministic. It can slightly vary because of mechanical links not being perfect, because of unusual road surface.. This shift in the transition distribution is one of the problem we will have to address during the simulation to reality transfer.

2.4 Reward

The choice of the reward function is primordial to correctly train the agent. The reward may depend on :

- The position of the agent. The position of the agent on the driving line is important, a position in the middle of a line must be prefered. The position may also be used to compute the distance to a target position.

- The speed of the agent. The agent may be rewarded if it is going straight forward.
- Collisions. An agent must have a negative reward in case of collision, and possibly if it gets to close to another object.

The design of the reward function will condition the performance of our agent in the simulator. This reward will be use to train a policy, i.e. a mapping from a state s to an action a to take. This policy will need to be adapted before transfer to real world. However, the reward may not need to be adapted.

3 Gym-DuckieTown Simulator

From the README.md in the github repository[3], here is an introduction to Gym-Duckietown.

> Gym-Duckietown is a simulator for the Duckietown Universe, written in pure Python/OpenGL (Pyglet). It places your agent, a Duckiebot, inside of an instance of a Duckietown: a loop of roads with turns, intersections, obstacles, Duckie pedestrians, and other Duckiebots. It can be a pretty hectic place!

This project has a github dedicated repository[4], which is a fork from the Gym-Duckietown original repo.

3.1 Installation

The installation is pretty straight-forward from the repository. Use the following commands :

```
git clone https://github.com/vcoyette/gym-duckietown
cd gym-duckietown
conda env create -f environment.yaml
```

[3] https://github.com/duckietown/gym-duckietown
[4] https://github.com/vcoyette/gym-duckietown

The installation have been tested on Windows, Linux and MacOS. Some problems may be encountered for the installation of certain packages. They can be resolved with package-specific installation instructions. For example, the installation of pyglet may raise an issue. It can be resolved by installing it from source from the pyglet github repository.

To use the simulator, the environment must be activated :

```
conda activate gym-duckietown
```

And the root folder of the project must be add to the PYTHONPATH environment variable. On linux :

```
export PYTHONPATH="${PYTHONPATH}:`pwd`"
```

On Windows, environment variable can be accessed in the parameters, section advanced parameters. You can then append the path to your project folder to the PYTHONPATH variable exists, or create it otherwise.

3.2 Manual Control

A UI application can be launched to manually control the robot. Actions can be sent from the keyboard, and images from the DuckieBot camera are diplayed. Here is a simple command to launch the application :

```
$ ./manual_control.py --env-name Duckietown-udem1-v0
```

3.2.1 Parameters

Here is a list of parameters which can be used.

- env-name: the name of the environment to execute (TODO anchor to env description)
- map-name: the name of the map to be used
- distortion: boolean, add distorsion
- draw-curve: boolean, draw the lane-following curve
- draw-bbox: boolean, draw the collision bounding box
- domain-rand: boolean, use domain randomization

- frame-skip: number of frames to skip (default 1)
- seed: seed

3.2.2 Keyboard

Here is a list of keys which can be used during the simulation.

- Escape : exit simulation
- Backspace : restart simulation
- Directional keys: go forward, backward or turn
- Shift : boost speed

3.2.3 Logs

The github repository contains a branch "experiment". This branch is intended to be used to do any experiment on the simulator. The difference to a classical "develop" branch is that it is not aimed to be merged into the master branch.

In this branch, we designed a logger for the manual control application. This logger will at each time step log the current position of the agent, the current speed, the current distance to the line, the action took. The objective is to manually control the agent to behave as we would expect it to behave. We would then be able to check the logs, and use them to design a reward function.

To enable this option, pass the –output option to manual_control.py. You can optionally specify another option '–filename example.csv' to specify the name of the output file, which would be in data/example.csv. If no file-name is specified, the logs will be stored in data/manual_controli.csv, where i is the first number for which this path is free.

When the episode is done, the manual control must be exit by pressing the S key to save the output.

The learning is performed using the train_reinforcement.py script. In order to execute it, change into the directory learning and run :

```
python –m reinforcement.pytorch.train_reinforcement
```

If an error is returned stating that a module doesn't exist, check that PYTHONPATH variable contains the repository base folder.

```
echo $PYTHONPATH
```

If it doesn't, get back to your to the root directory of the project and run:

```
export PYTHONPATH="${PYTHONPATH}:`pwd`"
```

The default algorithm implemented is DDPG. It can be tweaked by passing some parameters when executing train_reinforcement. A list of the parameters can be accessed by running :

```
python -m reinforcement.pytorch.train_reinforcement --h
```

3.3 Create Maps

You can very easily create a new **Duckietown** environment with a text editor. A Duckietown's map is a *.yaml* file, so you have to save your new map in the folder **maps** as "*my_new_map.yaml*" (the path should looks like this one : *./gym-duckietown-master/gym_duckietown/maps*).

If you want to see your map, you can use the following line in the terminal :

```
./manual_control.py --env-name Duckietown-udem1-v0
--map-name my\_new\_map}
```

Your robot will be controlled manually in your map.

A grassroots level of your *.yaml* should looks like this :

---

tiles:

- [floor, floor, floor , grass , grass , grass, floor, floor]
- [floor, floor , grass , grass , straight/S , grass, floor, floor]
- [floor, grass , grass , curve_left/W , curve_right/S, grass, floor, floor]
- [grass, grass , curve_left/W , curve_right/S, grass , grass, floor, floor]
- [grass, curve_left/W, curve_right/S, grass , grass , floor, floor, floor]
- [grass, curve_left/S, curve_right/E, grass , grass , floor, floor, floor]
- [grass, grass , curve_left/S , curve_right/E, grass , floor, floor, floor]

- [floor, grass , curve_left/W , curve_right/S, grass , floor, floor, floor]
- [floor, grass , straight/S , grass , grass , floor, floor, floor]
- [floor, grass , grass , grass , floor , floor, floor, floor]
- [floor, floor , floor , floor , floor , floor, floor, floor]

objects:

- kind: house
pos: [4.5, 9.1]
rotate: 90
height: 0.5

- kind: tree
pos: [1, 1]
rotate: 0
height: 0.5

- kind: tree
pos: [2, 8.5]
rotate: 90
height: 0.3

tile_size: 0.585

For each line, the number of tiles has to remain the same. The **tile_size** and the **height** of every object can change, but insofar as your goal is to transpose the simulation to real, you must not change their values. If so, you must have **tile_size**= 0.585 (for the conventional heights of the objects will be given below).

You can find your way in the map knowing that going upward is going North, and knowing that when you drive on a road tile, the name of the road tile tells you which direction you were facing when you arrived forward on it, and by which direction you will leave the tile. Let's give an example : the tile "*curve_left/W*" means you arrive on it while you were moving West, and the fact that you'll turn left on it says you are going to leave the tile by the direction South (it's of course reversible, you can arrive from South facing

North, and leave to the East).
You can also know your position $(x,\ y) \in \mathbb{R}^2$ on the map (this is the way you put objects on it). The point $[0,\ 0]$ matches the upper left corner of the upper left tile of the map (so the coordinates start at the very North/West). The coordinates keep going higher as you move toward South/East, increasing of 1 each time you go through one tile.

Here is the list of the tiles you can use to build your map:
- straight
- curve_left
- curve_right
- 3way_left (3-way intersection)
- 3way_right
- 4way (4-way intersection)
- asphalt
- grass
- floor (office floor)

You can (and should) orientate the roads (the six first tiles on the list) by adding "/N", "/E", "/S", "/W" (this will be oriented according to the rule given above).

The objects can be added following the example (changing the name of the object). Here is the list of them :

- barrier (height : 0.08)
- cone (height : 0.08)
- duckie (height : as you wish between 0.06 and 0.08)
- duckiebot (height : 0.12)
- tree (height : as you wish between 0.1 and 0.9)
- house (height : 0.5)
- truck (height : 0.25)
- bus (height : 0.18)
- building (height : 0.6)
- sign_stop, sign_T_intersect, sign_yield, etc... (height : 0.18 for the signs, 0.4 for a traffic light)

There are many other signs, you can have the complete list here :

https://github.com/duckietown/gym-duckietown/blob/master/gym_duckietown/meshes

It is possible to add the attributes :

- optional: True or False (makes the object optional)
- static: True or False (for the Duckiebot for example if you want to see them move)

To go any further about map creation, check out the Github of Duckietown at this link :

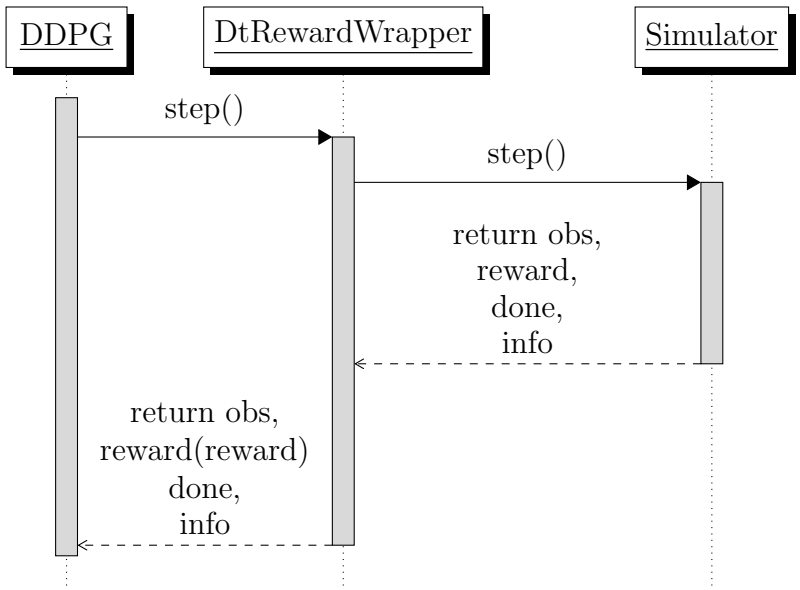
3.4 Reward Wrapper

The reward function is computed by the compute_reward function defined in the class Simulator (in the file gym_duckietown/simulator.py). The peferred way of modifying the reward function is to edit the function reward of the DtRewardWrapper class in the file learning/utils/wrappers.py.

This class is a subclass of the gym RewardWrapper class. This class implements the Wrapper/Adapter design pattern. It overrides the step function of the simulator. It is hard to explain, but you can have a look at the following sequence diagram to get an idea of how it works.

The idea is that the reward wrapper replaces the environment in the DDPG implementation. During the training, the DDPG calls the step function of the wrapper. This wrapper instance has an attribute which is the environment. The step function in the wrapper calls the function step of the environment, and captures the returned values (observation, reward, done and info). Then, the wrapper runs its own reward function with as parameters the reward returned by the previous call. The wrapper step function returns the values of observation, done and info which were returned by the environment, and returns its custom reward to replace the reward of the environment.

The wrapper containing a environment attribute, it has access to all the same information as the simulator to compute its custom reward. For example, it can use the current postion through env.cur_pos, the angle through env.angle, etc.



TODO : add legend to diagram