Project 074 : Simulation to Real

Study on Gym-Duckietown environment, supervised by David Bertoin

Auteurs :
Olivier SUTTER
Mathieu VERM
Michael ROMAGNE
Alexandre MARTIN
Pablo MIRALLES
Vincent COYETTE

March 19, 2020

Contents

Introduction

Reinforcement Rearning (RL) is an area of machine learning. The goal is for an agent to learn an optimal behaviour and adapt its comportement, based on experience, in a given environment. Reinforcement Learning has been very successful in the last few years, notably for its great performance in games (e.g. AlphaGo [3]). However, it suffers from a lack of industrial applications.

One of the reason for this is that an agent needs a lot of experience in order to learn a reasonable behaviour. Those learning experiences must be run in a simulator for safety, speed and financial reasons. Moreover, the transfer of behaviour from the simulated agent to the real world (for example an autonomous vehicle) is far from being trivial. Even for high quality simulators, there is a shift in both states and transitions spaces between the simulator and the real world.

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

All the project experiments will be based on Duckietown[1] resources. The Duckietown fundation is a non-profit foundation providing tools dedicated to education and research in the fields of AI and robotics. In particular, it provides a gym-based simulator (gym-duckietown), a robot (Duckiebot) and a real world environment (Duckietown).

This document aims at providing an understanding of the basic concepts required to work on the project.

A fork of the gym-duckietown repository is dedicated to this project[2]. The latest version of this manual can be found on the branch documentation.

[1] `https://www.duckietown.org/`

[2] `https://github.com/vcoyette/gym-duckietown`

Chapter 1

Duckietown environment

A reinforcement learning agent is usually trained in an environment described as a Markov Decision Process (MDP). A MDP is basically a 4-tuple (S, A, P_a, R_a). Let's detail each of these components for the duckietown environment.

1.1 State

The state S is the state of the environment from the agent point of view. Here, the robot contains only one sensor, the camera. The state of this environment is the output of this camera. The camera has a resolution of 640x480, and thus the state space is $S = [0, 255]^{640x480x3}$ (3 is for the 3 RGB colors).

1.2 Actions

The action is composed by the velocities of each of the two wheels. By default, the action space is continuous, $A = [-1, 1]^2$, 1 being full forward velocity, -1 full backward.

Caution : Be careful here as the definition of the actions in the README of the duckietown repository is describing actions differently. It also defines a two-element tuple, but the first one represents the forward velocity while the second one represents the steering angle. This definition is more natural to manually control the robot. A wrapper (DuckietownEnv) exists to switch from the control of wheels velocity to the control of forward velocity and streering. However, we chose not to use this wrapper and to keep the default implementation. The code contains some inconsistencies without the wrapper, which will be described later.

1.3 Transitions

The transitions 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. This is typically a point where the transfer from the simulator to the robot may be harmful.

Indeed, the transitions in the simulator can be considered as "perfect", the position of the robot in state s' is computed from the position in state s using the velocities stated by action a. This is not fully deterministic, as some objects (such as duckie pedestrians) may have a stochastic behaviour, but the updated position is a deterministic function of the previous position and current action.

However, in the real world, the transitions between the 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.

1.4 Reward

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

- The position of the agent. The position may be used to compute the distance from a target position, and the distance from the middle of the lane.
- The speed of the agent. The faster the better.
- Collisions. The agent may get a negative reward in case of collision, and possibly if it gets too close to another object.

The default reward in duckietown is as follows:

$$R(t) = 40 * C_p - 10 * dist + alignment * speed \tag{1.1}$$

$$C_p = \sum_{Obj}((pos_{agent} - pos_{object}) - SR_{agent} - SR_{object}) \tag{1.2}$$

$$\tag{1.3}$$

- C_p is the collision penalty for being dangerously close to other objects. It is a proxy for area overlap. Notably, one can collide with several objects at once (with additive

effects) and one can collide with respect to the reward function (which uses the safety radius) without the episode restarting (which depends on the collision radius). Note that collision penalty is smaller than 0 whenever there is a collision, therefore this is actually a penalty despite the positive sign.

- dist is the distance between the agent's center and the closest point on the line defining the lane's center
- alignment is the dot product between direction and the normalized tangent of the road.
- speed is the agent's speed

This reward function has presented several issues:

1. The penalties (collision and distance) are so strong that the agent usually much prefers to go around in circles at max speed at the center of the lane than to actually follow it.

To face this issues, a minimalist reward function has been used :

$$R = \begin{cases} speed, & \text{if } dist \leq d \\ -1, & \text{otherwise} \end{cases} \tag{1.4}$$

Where d is a constant defining a minimal distance to the center of the line, over which we consider the robot should be penalised. We used a value $d = 0.1\ m$.

The design of the reward function will condition the performance of our agent in the simulator. This reward will be used 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 the real world. However, the reward does not need to be adapted.

Chapter 2

Gym-DuckieTown Simulator

From the README.md in the github repository [1], here is an introduction to Gym-Duckietown.

> Gym-Duckietown is a simulator for the Duckietown Universe, written in pure Python/OpenGL (Pyglet). It puts the 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 dedicated github repository[2], which is a fork from the Gym-Duckietown original repo.

2.1 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
```

The installation has 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 the pyglet github repository.

To use the simulator, the environment must be activated :

[1] *https://github.com/duckietown/gym-duckietown*
[2] *https://github.com/vcoyette/gym-duckietown*

```
source activate gym-duckietown
```

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

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

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

2.2 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 displayed. Here is a simple command to launch the application :

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

2.2.1 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

2.2.2 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

2.2.3 Logs

The github repository contains a branch "experiment". This branch is to be used to do any experiment on the simulator. The main difference from a classical "develop" branch is that it does not aim at being merged into the master branch.

In this branch, we designed a logger for the manual control application. At each time step, this logger will log the current position of the agent, the current speed, the current distance from the lane, the action took. The goal is to manually control the agent so that it behaves as we would expect. 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 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 exited by pressing the S key to save the output.

2.3 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*

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 manually controlled 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 the real world, you must not change their values. If so, you should 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 on it, and by which direction you will leave the tile. Let's give an example : the tile "*curve_left/W*" means that you arrived 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 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 as shown in the following example (changing the name of the object). Here is a 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 check the whole 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 the Github of Duckietown on this link : [3]

2.4 Reward Wrapper

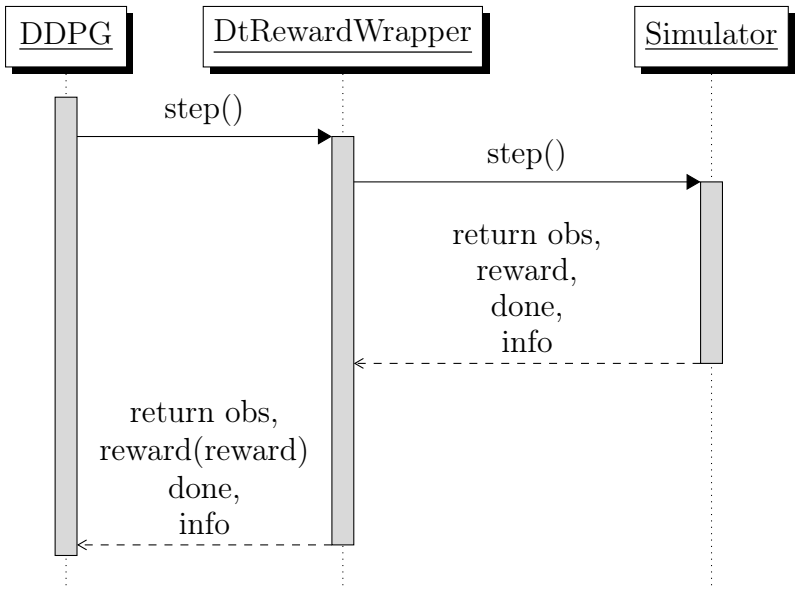
The reward function is computed by the compute_reward function defined in the class Simulator (in the file gym_duckietown/simulator.py). The best way to modify the reward function is to edit the function reward of the DtRewardWrapper class in the file learning/utils/wrappers.py.

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 the reward returned by the previous call as parameter. 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.

[3] *https://github.com/vcoyette/gym-duckietown/tree/documentation*

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



TODO : add legend to diagram

2.5 Domain randomization

When it comes to transfer knowledge from simulation to reality, a problem we may face is that the images collected from simulation diverge too much from reality. Often, people even retrain their model from scratch when moving to the physical world.
One solution to this problem is domain randomization. The idea is to perturb the dynamics or look of the simulator such as colors, textures, horizon ... Thus, we obtain a more variable dataset and we have a better generalization, which is beneficial for transfer on the real robot. This part will deal with how you can do domain randomization in gym-duckietown environments.

2.5.1 The randomisation API

The folder at `gym-duckietown/gym_duckietown/randomization` contains the domain randomization API. This API contains all of the pre-packaged methods for randomization within gym-duckietown, which I'll list here. This folder also contains a readme file detailing the API.

The domain randomization is driven by the Randomizer class, which takes as input

a configuration file and outputs (upon call to randomize) a set of settings used by the Simulator class (the core class of gymduckietown managing the environment). To activate any domain randomization at all, the simulator class must have domain_rand true passed as a parameter to its constructor

If a randomizable variable is not found in the configuration file, it will be randomized according to the default values found in `gym-duckietown/gym_duckietown/randomization/config/default.json`. 3 types of distribution are supported, int, uniform and normal. 4 variables are randomizable by default:

1. horz_mode: The task is made harder by making the horizon more similar to the road. It can take integer values from 0 to 3 where:
   (a) 0: Sets the skybox to a blue sky, the default
   (b) 1: Sets the skybox to a gray wall, intended to be similar to room testing conditions
   (c) 2: Sets the skybox to a dark gray box
   (d) 3: Sets the skybox to a light gray box
2. light_pos: Makes the simulator more or less iluminated, by changing the position of the single light source.
3. camera_noise: Adds noise to the camera position for data augmentation purposes. This noise is applied at each render, giving the screen input a "twitchy" behaviour.
4. frame_skip: No info provided. Code inspection shows this is the number of frames to skip per action. Higher frameskip makes the agent able to act less frequently.

The API is sorely lacking in the amount of variables that can be randomized. It provides some flexibility in randomization of the input, but it provides no randomization besides frame_skip of transitions and/or actions.

2.5.2 Non API Randomisation

Code inspection reveals that more randomisation occurs, contingent on domain-rand being activated.

1. Within Simulator.py: All calls to _perturb(val,scale) distort the value passed as argument if and only if domain rand is true, otherwise they return it undisturbed. The distortion is a multiplicative % error drawn uniformly between $1 - scale$ and $1 + scale$, with a default of $scale = 0.1$ (e.g. by default values are randomised between 90 and 110%). The function is called on:

- horizon_color, beyond the randomization introduced by horz_mode. 10% for blue_sky and wall_color sky (modes 1 and 2), 40% for modes 3 (dark gray) and 4 (clear gray)
- glLight function : sets the values of individual light source parameters, making the environment more or less iluminated. It modifies the light position, the ambient and diffusion of light.
- ground_color, 30%
- wheel_dist, 10%
- cam_height, 8%
- cam_angle, 20%
- cam_fov_y, 20%, camera field of view side length of the ground/noise triangles generated as distractors (which themselves are generated randomly in the first place? Seems redundant to do it twice), 10%
- tile color, 20% for each tile
- object color, 20% for each color
- CAMERA_FORWARD_DIST of gl.glTranslatef on line 1434, 10%
- The actual tile texture loaded is randomised with randint amongst all possible candidates
- Some optional objects are invisible, 33% chance

2. Within objects.py

- DuckiebotObj (Cars)
  - follow_dist is randomised from 0.3 to a uniform between 0.3 and 0.4
  - velocity is randomised from 0.1 to a uniform between 0.05 and 0.15
- DuckieObj (Pedestrians)
  - pedestrian_wait_time randomised from 8 to a randint from 3 to 20, takes on new random value on same range when finish crossing street
  - vel randomised from 0.02 to a normal with avg 0.02 and stdev 0.005, takes on new random value on same range when finish crossing street
- TrafficLightObj
  - freq is randomised from 5 with randint(4,7)
  - The lights start randomly from off as either On or Off, 50% chance each

2.5.3 Randomizing inputs

Randomizing inputs can be accomplished via the API. Other possible sources of randomization are:

- Changing hue of key elements: stop signs, road... Might be hard since they're all texture based. A possible approach: Generating variations on existing textures: Automatically generate X diff textures with some other software from existing textures passed through some filters, then use the in-built texture randomiser
- Adding noise to the camera acquisition (not position)? This is similar to randomising the color of the sky and objects themselves so it might not be interesting. However camera noise provides variations during a single episode or each timestep which may imitate real camera noise (or not?)
- Approaching photorrealism in some way can be helpful in improving the performance (Johnson-Robertson et al.) May prove unfeasible given the simulator and available time and resources
- Addding additional light sources?
- Gaussian noise foreground of images and edge blurring through gaussian noise at edges on input images is also helpful (Hinterstoisser et al.) The technique can be extended to other blending techniques (Dwibedi et al.)

Chapter 3

Reinforcement Learning Training

3.1 Training Scipt

The learning is performed using the train_reinforcement.py script in the learning/reiforcement/pytorch directory. 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 if PYTHONPATH variable contains the repository root folder.

```
echo $PYTHONPATH
```

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

```
cd ../
export PYTHONPATH="${PYTHONPATH}:`pwd`"
```

The training can be tweaked by passing some hyperparameters when executing train_reinforcement. A list of this parameters can be accessed by running :

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

3.2 Policy

The policy can be tweaked through the "–policy" argument, accepting as a string the name of the policy to use to train the agent. Currently available values are ddpg[1] and td3[2], ddpg being the default.

The policies are defined in their own files (e.g. ddpg.py and td3.py). The implementation is decoupled from the architecture of actors and critic, which are defined in actor.py and critic.py

Among these paramters, you can notice the "–per" parameter, which can be used to use a Prioritize Experience Replay buffer. It is for now functional with the ddpg algorithm, **not with the td3**.

Chapter 4

Transfer to the Duckiebot

Bibliography

[1] Gabriel Barth-Maron et al. “Distributed Distributional Deterministic Policy Gradients”. In: *CoRR* abs/1804.08617 (2018).

[2] Scott Fujimoto, Herke van Hoof, and David Meger. “Addressing Function Approximation Error in Actor-Critic Methods”. In: *CoRR* abs/1802.09477 (2018).

[3] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (2017).