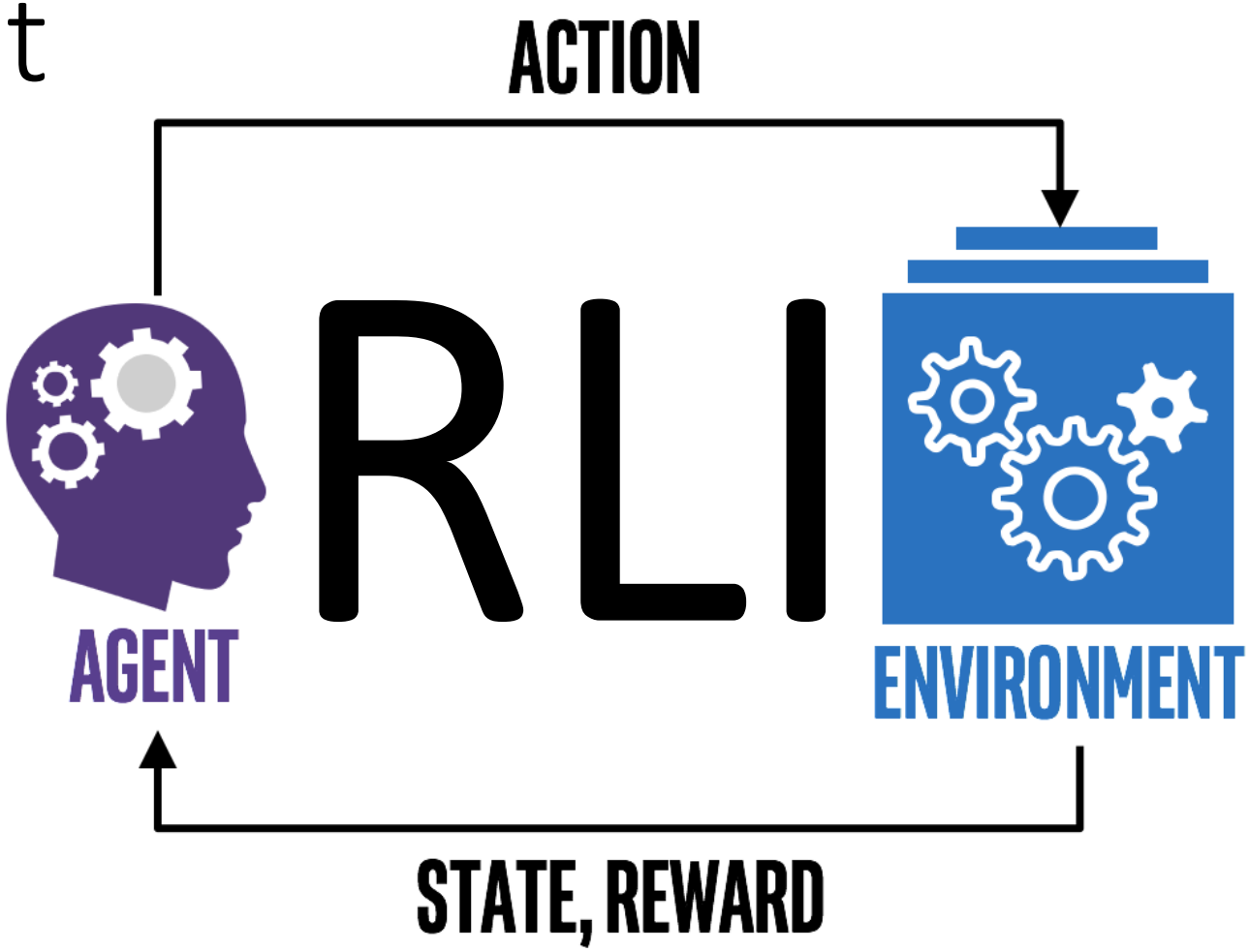
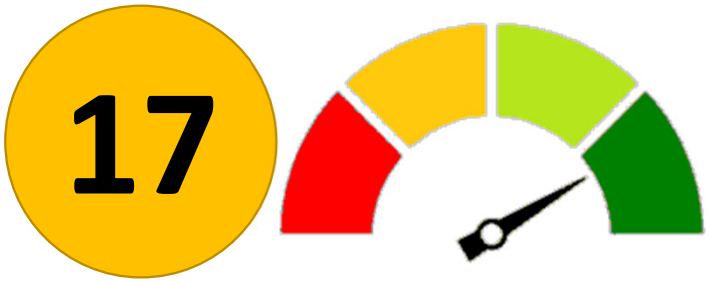
Reinforcement Learning Introduction





José Manuel Rey, IE

Pit lane repairs

Reinforcement Learning Introduction Assignment 17.00

Pit lane repairs

Date: Mar/07 2025

Due Date: Apr/4 2025 [SESSION 25]

Ongoing grading (maximum)

50 Individual Assignment [mini-groups]

Assignment Description
Overview

Using as an starting point the code presented for the “Car Driving” practice, and included in the files attached, this assignment consists of 2 parts/questions:

Assignment Part 01 → Convert the QTable based method to a “vainilla” DQN algorithm (i.e. just a NN and basic experience replay [No target network required, but optional]) *[60% points]*

- Some useful references could be the code explained in class for DQN:
  - deep-q-learning-keras.ipynb → Tensorflow.Keras
  - reinforcement_q_learning.ipynb → Pytorch
- The current code for the “Car Driving” practice is located at: RLI_17_A0.zip
- Note: all the referenced code files and the code zip file are also included in Assignment file

Assignment Description Overview

Assignment Part 02 → Discuss and explain (and eventually try to implement and test) possible ways for improving the current model *[40% points]*

- Possible continuous inputs instead of Box2D of *int*
  - The state-to-bucket is redundant in the current implementation as the RaceEnv already gives us a Boxed observation_space (discrete)
  - The Car class however could provide us with a continuous (in pixels) range of values for the radars, but in PyRace2D.observe() we are already rounding them to 20 pixels discrete intervals
- Possible range of Actions instead of discrete
- ```
  def action(self, action):
      if action == 0: self.car.speed += 2
      elif action == 1: self.car.angle += 5
      elif action == 2: self.car.angle -= 5
  ```
- Optionally a possible new Action of 'BRAKE' could be considered (Remember that this is now controlled just by 'friction' [deceleration] at every update)
- ```
  def update(self,map=None):
      #check speed
      self.speed -= 0.5
      if self.speed > 10: self.speed = 10
      if self.speed < 1:  self.speed = 1
  ```

Assignment Description Overview

THIS PART IS TOTALLY OPTIONAL. IT IS AN EXTRA ACTIVITY YOU CAN PERFORM ON YOUR MODEL IN ORDER TO EXPLORE SOME MORE ADVANCED OPTIONS

Assignment Bonus → Migrate your DQN model to some more advanced algorithms *[extra: 30% points]*

- Once you have completed the previous parts 01 (DQN implementation) and 02 (definition of new possible states [observations] and actions) and taking advantage of the fact that the implementation of the environment is fully compatible with *OpenAI Gymnasium* you can take benefit from this to test possible algorithms that include among other features the possibility of using continuous variables for the characterization of states or actions as well as the use of more advanced models (including for example Policy Gradients)
- One possible smooth line of work could be migrating your existing Q-Learning approach to a DDPG (handling continuous environments)
- This process can become very straightforward (and extremely simple) if you use some of the frameworks covered in class (My suggestion would be trying **stable-baselines-3**)

Assignment Description
Overview

For each part/question you should elaborate a self-contained directory-zip file with your ".py" and ".ipynb" files (derived from the documents delivered with the assignment), providing (when required) a clear concise explanation of the conceptual approach, the diagrams or models representing the problem, comments for the code used and the explanation of the solution

SUBMIT YOUR WORK BY ZIPPING TOGETHER THE FINAL CODE (COMPLETED WITH YOUR ANSWERS) IN A SINGLE SUBMISSION FILE NAMED:

"RLI_17_00 – mini-group number.zip" (see the attached .txt for the mini-groups)

Additional suggestions…
For Part 01

- Just try to track the use of the “q_table” variable through the code in “Pyrace_RL_QTable.py”. That will lead you quite directly to spot the code to be refactored.
- I would suggest you to name your newly generated code file as “Pyrace_RL_DQN.py” for consistency purposes.
- Just focus in the “simulate()” method. The “load_and_play()” requires you to also properly adapt the “load” and “save” methods of your model to the new architecture (an example of this can be seen in some of the examples covered in class, but for this assignment you can freely skip that part so as to alleviate you some of the tedious technical stuff) [so, no need for implementing “warm start” from previous pre-trained stages of your model]
- No changes are required at this stage to the “gym_race” parts of the code

Additional suggestions…
For Part 02

- As commented in class try to think of feasible ways for enhancing the “information” that the agent receives from its environment [observations] and the type of more fine-grained responses [actions] it could take in order to improve its performance
- For implementing this ideas probably you will have to modify some “methods” of the environment itself [**gym_race\envs\pyrace_2d.py**]. In some cases this could be also translated in changes in the “gym wrapper” [**gym_race\envs\race_env.py**] for consistence, and of course, in appropriate changes in the “input” layers [states] of your DQN architecture and eventually in the “output” layer [actions]
- Note: please, note that the “state-to-bucket” is redundant in the current implementation as the RaceEnv already gives us a Boxed observation_space (discrete) as the **Car class** already transforms its continuous radar detections into discrete ranges. I have just kept it so that if you decide to implement some other more “continuous” measurements but keeping open the possibility of comparing flexibly the behavior of discrete vs continuous models you can have a handy function for that.
- I would suggest you to name register your newly generated environments as “Pyrace-v3” [in the “__init__.py” file] for proper OpenAI Gym consistency

```
from gymnasium.envs.registration import register
register(id='Pyrace-v3’, entry_point='gym_race.envs:RaceEnv’,
         max_episode_steps=2000,)
```

Additional suggestions…
For Part 02 and 03 (Bonus)

- Have a critical look at the rewarding function being used.
- Try to analyze if its really consistent (aligned an informative) with the true performance objective of the vehicle, and experiment (or at least suggest) with possible changes or improvements in such rewarding function

Administrative and additional notes

- Find attached (in the zip file) the configuration of the mini-groups for the assignment:

  →RLI_2024-25 (A or B) - ASSIGNMENT 1 RLI_17_00 TEAMS.txt

  Only submit ONE zip file per group. Choose ONE member of each group for that submission. In case you send more than one, ensure that all the submissions by the different members are identical, as the system will choose just ONLY one of them (randomly) for the review and evaluation of the whole group.

- As usual, **the code should run without warnings or errors**. All additional required files should be included in the ZIP file, and if you were using any "novel" additional external library for your work, clearly indicate a brief description of its purpose, the version used and the "**`!pip install <package>`**" required for its installation