Workshop 10

Learning From Big Data

Reinforcement Learning (RL)

*Please do the exercises or tasks without "Optional" mark at first. After that, if you still have some time, please try the tasks with "Optional".

Part 1: OpenAI Gym

- **Gym** is a collection of environments/problems designed for testing and developing reinforcement learning algorithms—it saves the user from having to create complicated environments.

- **Gym** is written in Python, and there are multiple environments such as robot simulations or Atari games. There is also an online leaderboard for people to compare results and code.

  if you have not installed Gym, please open a terminal and type the following command:

```
$pip install gym[atari]
```

Getting start with Gym

1) Open this web-page, https://gym.openai.com/docs/, for understanding its basics. Read the text and copy the code one cell by one cell into a notebook, while run them in the order.

   (If you don't like copy/paste them, you may use my copied notebook, called Practice_with_Gym.ipynb in …/Lab_10

2) A full list of environment of Gym is presented in https://gym.openai.com/envs/#classic_control. A list of different kinds of environment under title" Algorithms", such as Atari, Box2D, classic control, … , is given on the left column. Select some to have a review. Maybe, you work with one of them one day.

3) You may also study their source codes. For example, if you have clicked MountainCar-v0, its page, https://gym.openai.com/envs/MountainCar-v0/, will be displayed with a description about it. You can click "View source on Github" to get its source code.

Part 2: Reinforcement Learning

Let's study this topic using samples in the notebook, called Reinforcement_Learning.ipynb, given in …/Lab_10., where a number of tasks are implemented.

Task 1: Introduction to OpenAI gym

- The first section in the notebook, dealing with how to use OpenAI gym for developing and comparing Reinforcement Learning algorithms. Gym toolkit provides many environments for your learning *agents* to interact with. Let's start by importing gym, and make an Gym environment, called 'MsPacman-v0'.

- We will play this game (with 3 lives), by moving in random directions for 10 steps at a time, recording each frame. Its animation (it's a bit jittery within Jupyter) will be displayed.

Actions:

- Just simply run celles one by one in sequence and see and understand what happens.

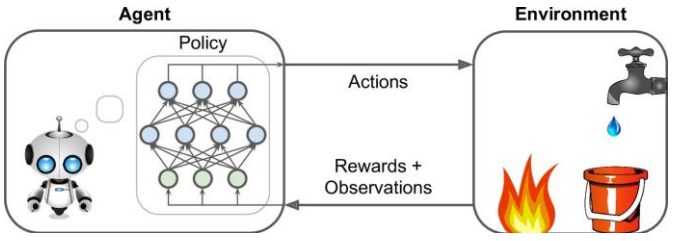
Task 2: A simple environment: the Cart-Pole

- As introduced in the lecture, the Cart-Pole is a very simple environment composed of a cart that can move left or right, and pole placed vertically on top of it. The agent must move the cart left or right to keep the pole upright.
- If you use Ubuntu (or Linux), you don't need to need for Fixing the rendering issue.
- Let's try "**hard code a simple strategy**":
    - if the pole is tilting to the left, then push the cart to the left, and *vice versa*.

Let's see if that works!

Task 3: Neural Network Learning Policy

- As lectured, the algorithm used by the software agent to determine its actions is called its **policy**. For example, the policy could be a neural network taking observations as inputs and outputting the action to take. The policy can be any algorithm you can think of, and it does not even have to be deterministic.



- Continue to work with the Card-Pole. Now can we make the pole remain upright by applying a ***policy*** for that. This is the strategy that the agent will use to select an action at each step. It can use all the past actions and observations to decide what to do.

Actions:

Continue to study the notebook. The code under the title of **"Neural Network Policies"** implements a policy, where a neural network will take observations as inputs, and output the action to take for each observation. Referring to the comments within the code, see how to achieve this policy.

Task 4: Policy Gradients

- As lectured, to train this neural network we will need to define the target probabilities y. If an action is good we should increase its probability, and conversely if it is bad we should reduce it. But how do we know whether an action is good or bad?
- The problem is that most actions have delayed effects, so when you win or lose points in a game, it is not clear which actions contributed to this result: was it just the last action? or the last 10? or just one action 50 steps earlier? This is called the ***credit assignment problem*.**
- The ***Policy Gradients*** algorithm tackles this problem by first playing multiple games, then making the actions in good games slightly more likely, while actions in bad games are made slightly less likely.

Actions:

- Referring to the code of **"Policy Gradients"**, run its code at first and then we go back and understand the code.

Part 5: (Optional) Markov Decision Processes

- It works now after I modified some codes.
- In order to make a better reinforcement learning policy, **Markov Decision Processes (MDPs),** including a number of equations, were discussed in the lecture. As a result, reinforcement Learning problems with discrete actions can often be modeled as Markov decision processes.
- Before you do next exercises, please review Section 7 "Markov Decision Processes" and Section 8 "Temporal Difference Learning and Q-Learning" of Chapter 17 in the lecture-slides.

Actions:

- Look at the two separate codes of **"Markov Process"** and **"Markov Decision Processes"**, see how to implement them in coding.
- Q-Learning will learn the optimal policy by watching the random policy play. Its implementation is given in the notebook with title **"Q-Learning"**. Try to understand it.

Task 6 (Optional): Learning to Play MsPacman using Deep Q-Learning

- It works well, but take time to run.
- This is an advanced topic and example for reinforcement learning.
- I have edited a PDF file, PacMan_Using_Deep_Q_learning.pdf, given in …/Lab_10/Lab10_tasks. You may read the text at first and then have a practice on this project, based on the last part of the notebook, called "Learning to play MsPacman using Deep Q-Learning".