Machine Learning

COMP5611M

Coursework 2: Reinforcement Learning

This coursework comprises 10% of the marks for this module.

Deadline: **14/12/2020 at 10:00:00** by electronic submission in Minerva.

Implement the following specification in Python. Submit a python source file with your implementation, and a pdf with the required plots.

Environment

Winter is here. You and your friends were tossing a frisbee around at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc.

We will use a modified version of the Frozen Lake environment from OpenAI Gym (https://gym.openai.com/envs/FrozenLake-v0/) to simulate this scenario. The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, denoted by “F” for frozen, and others lead to the agent falling into the water, denoted by “H” for hole. The agent is rewarded for finding a walkable path to a goal tile, denoted by “G” for goal. The starting point is denoted by “S” for start.

Template

You are given a template of the code structure which you will use for implementing this work. The “main” function initializes the World and the Agent. The main function includes the main loop for running training episodes. Also, there are plotting functions provided to help you visualize the results. You will see that there are three plots that we expect you to show. The first plot contains the q-value of the available actions in the initial state over training episodes as recorded by “world.q_Sinit_progress”. The second plot contains the evolution of the number of steps per successful episode over training episodes. A successful episode is an episode in which the agent reaches the goal (i.e., not just any terminal state) as recorded by “episodeStepsCnt_progress”. The third plot contains the evolution of the return per episode as recorded by “r_total_progress”. The last plot is made smoother using the running_mean function. It is for you to implement the code that updates these values.

The World class is where you will write your code for running a training episode, and the Agent class is where you will write your code for handling the action-value function and the policy. The given code already has the functions in place with their corresponding input parameters. Some of

the functions also have some initial code that will help you get started in writing your own.

Here are a few functions from OpenAI GYM that will help you in this process:

- `env.observation_space.n`: returns the number of states in the environment.
- `env.action_space.n`: returns the number of actions in the environment.
- `env.reset`: sets the agent at the initial state, and returns the index of the initial state (that is, 0).
- `env.render`: prints the current environment state, to help you visualize the environment and the agent's location.
- `env.step(a)`: executes action "a" and moves the agent to the next state. It returns the next state, the reward, a boolean indicating if a terminal state is reached, and some diagnostic information that you can neglect.

Implementation

Your task is to implement a domain-independent reinforcement learning agent. Your implementation must have the following functions:

- `predict_value(s)`: returns a vector with the value of each action in state s.
  [2 Mark]
- `update_value_Q(s, a, r, s_next)`: updates the current estimate of the value of the state-action pair <s,a> using Q-learning.
  [10 marks]
- `update_value_S(s, a, r, s_next)`: updates the current estimate of the value of the state-action pair <s,a> using Sarsa.
  [10 Marks]
- `choose_action(s)`: returns the action to execute in state s, implementing an ε-greedy policy.
  [6 Marks]
- `run_episode_qlearning()`: runs an episode, learning with the Q-learning algorithm.
  [8 Marks]
- `run_episode_sarsa()`: runs an episode, learning with the Sarsa algorithm.
  [12 Marks]
- `run_evaluation_episode()`: runs an episode executing the currently optimal policy.
  [2 Marks]

Note: when implementing both Q-learning and Sarsa, extra care must be taken when the update is about a terminal state. For any non-terminal state, the update rule includes the value of the next action in the next state, that is:

$$q_{k+1}(s,a)=q_k(s,a)+\alpha(r+\gamma q_k(s',a')-q_k(s,a))$$

for Sarsa, and:

$$q_{k+1}(s,a) = q_k(s,a) + \alpha(r + max_{a'} \gamma q_k(s',a') - q_k(s,a))$$

for Q-learning, where, α is the learning rate, r is the immediate reward, γ is the discount factor, s' is the next state, and a' is the next action. If a state is terminal, there is no next state s'. In this case, the update for both methods is:

$$q_{k+1}(s,a) = q_k(s,a) + \alpha(r - q_k(s,a)) \quad .$$

Experiments

Generate the three plots as detailed in the Section Code, and add them to the pdf report.

In these experiments, you will study the difference between on- and off-policy learning. Perform the following four experiments, and add the corresponding plots to the report:

- RL Algorithm: Q-learning; ε = 0.8.
- RL Algorithm: Sarsa; ε = 0.8.
- RL Algorithm: Q-learning; ε = 0.1.
- RL Algorithm: Sarsa; ε = 0.1,

where ε is the parameter of the ε-greedy policy used for exploration. Briefly describe in the report the difference between on- and off-policy learning that emerges from the experiments.

[10 Marks]

[total: 60 Marks]