Reinforcement Learning

Introduction

The exercises below are about the two algorithms presented on the lecture: the Q-learning and the Policy Gradient (PG) or REINFORCE algorithm.

The algorithms are present on the lecture slides, they should be directly applicable.

Only one of tasks is needed for a full (one-assignment-worth) score, you can choose which one you solve.

Some remarks about Q-learning

Q-learning is an *off-policy* algorithm, meaning that the learning policy (the one used for interaction with the environment during learning) can be different from the policy that is being learnt.

In practice, the learnt policy (the result of the training) is a greedy policy with respect to the estimated Q function (see the corresponding Bellman update). The training policy is usually ϵ-greedy, meaning it is greedy with probability $1 - \epsilon$, and uniformly random (ie. chooses the action uniformly randomly from all available actions) with probability ϵ. This will add some exploration, which is useful for training. $\epsilon = 0.1$ should be a sensible value.

Tasks

1. Train an agent using Q-learning on the `GridEnvironment` task found in `gridworld.py`.

   The task of the agent is to step onto any of the goal positions marked by `X` without stepping onto any of the walls (marked by #). Both will terminate the episode with positive and negative reward, respectively. (Note that for subtasks a, b, c and d, no training is needed.)

   In the maps provided, the `S` denotes the starting position. You can reposition it anywhere you want to start the agent from different locations.

   For subtasks a and b, you solve the RL problem by hand, by looking at the dynamics in `GridEnvironment`. On the other hand, from subtask c, you "forget" the dynamics, and think of the environment as a black box you know nothing about. This latter context is where Q learning comes into the picture.

   a. By reading the code, what is the state and action space? What is the environment dynamics and reward function? Describe them using mathematical formulae or your (precise) words.

b. Looking at the `SMALL_GRID` maze, give an optimal policy (i.e. one that finds the goal state the fastest possible way) for each state(location) from where it is possible to reach the goal state. (E.g., don't bother with the wall states.)

   This is a pen and paper exercise, the goal is for you familiarise yourself with what a policy is. You can specify either a deterministic policy, in which case only the action is needed for every state, or a deterministic policy, for which you should specify a probability distribution over actions for each state.

c. Give the $Q(s, a)$ values for this policy for every location, with the parameter `controller_error` set to 0.

   This will be a $|\mathcal{S}| \times |\mathcal{A}|$ matrix containing the action values.

d. Using the Bellman equation, verify that your Q matrix is correct. Start the agent from all the different state-action pairs, and follow your policy until the end of the episode, verifying the equation along the way.

e. Using the Q-learning algorithm, train a policy to solve the environment (starting from the original starting point), using both `controller_error=0` and `controller_error=0.3` on the `SIMPLE_GRID` map.

   Plot the rewards as the training progresses. Use two different curves: one is for the training reward, i.e., the cumulative reward at the end of each training episode. The other is for the testing reward: after every 10 training episodes, run a few test episodes, i.e. with a greedy policy with respect to Q, and plot the reward. What do you think, how precise are the test episode estimates? (What is the spread of the episode return?)

f. Train an agent on the `LARGER_GRID` map as well, and plot the rewards again. How many more iterations did you need? If you could initialise your Q function with an initial vector for each state to help the agent solve this particular maze, what would you choose?

2. Train an agent using the REINFORCE algorithm on the Cartpole environment from OpenAI Gym[1]. There is a skeleton code in `cartpole.skeleton.py` written in PyTorch to get you started. You may need to install Gym (e.g., by issuing `#! pip install gym` if you use Colab) before you start.

   You may need to use the two variance reduction techniques mentioned on the lecture.

   a. Implement a loss function in `pg_loss` that Pytorch should minimise. Its gradient should correspond to the equation $\nabla_\theta G^\pi$ from the lecture, then PyTorch will take care of the differentiation.

      The arguments are the action taken at the current timestep (`taken_action`), the probability vector for the actions in the current state (`pi`) and cumulative reward from that state (`cum_reward`). (In batch form.)

   b. Implement the training on one episode (more precisely, on a batch of episodes) in `train_on_episode`, given a list of state transitions (state, action, next state tuples). Convert this list to inputs suitable for `pg_loss`.

   c. Complete the training algorithm in `train_epoch`. Sample trajectories via agent interaction, and then call `train_on_episode` to update the model.

   d. Run training, plot the episode returns and the loss for each epoch.

[1] https://gym.openai.com/envs/CartPole-v1/