Homework 4

CMU 10-703: Deep Reinforcement Learning (Fall 2019)
OUT: Oct. 14, 2019
DUE: Oct. 28, 2019 by 11:59pm

Instructions: START HERE

- **Collaboration policy:** You may work in groups of up to three people for this assignment. It is also OK to get clarification (but not solutions) from books or online resources after you have thought about the problems on your own. You are expected to comply with the University Policy on Academic Integrity and Plagiarism[1].
- **Late Submission Policy:** You are allowed a total of 10 grace days for your homeworks. However, no more than 3 grace days may be applied to a single assignment. Any assignment submitted after 3 days will not receive any credit. Grace days do not need to be requested or mentioned in emails; we will automatically apply them to students who submit late. We will not give any further extensions so make sure you only use them when you are absolutely sure you need them. See the Assignments and Grading Policy here for more information about grace days and late submissions: `https://cmudeeprl.github.io/703website/logistics/`
- **Submitting your work:**
  - **Gradescope:** Please write your answers and copy your plots into the provided LaTeX template, and upload a PDF to the GradeScope assignment titled "Homework 4." Additionally, upload your code (as a zip file) to the GradeScope assignment titled "Homework 4: Code." Each team should only upload one copy of each part. Regrade requests can be made within one week of the assignment being graded.
  - **Autolab:** Autolab is not used for this assignment.

[1] `https://www.cmu.edu/policies/`

Problem 0: Collaborators

Please list your name and Andrew ID, as well as those of your collaborators.

Environment

You are provided with a custom environment in `2Dpusher_env.py`. In order to make the environment using `gym`, you can use the following code:

```
import gym
import envs

env = gym.make('Pushing2D-v0')
```

The environment is considered "solved" once the percent successes (i.e., the box reaches the goal within the episode) reaches 95%.

Problem 1: Deep Deterministic Policy Gradients (DDPG) [60+5 pts]

In this problem you will implement DDPG, an off-policy RL algorithm for continuous action spaces. In homework 2 you implemented DQN, another off-policy RL algorithm. Like DQN, DDPG will learn a Q-function. Recall DQN chose actions by computing the Q value for each action and then chose the maximum:

$$\pi(a \mid s) = \mathbb{1}(a = \max_a Q(s, a))$$

While we would like to use this same policy in continuous action spaces, finding the optimal action involves solving an optimization problem. Since solving this optimization problem is expensive, you will *amortize* the cost of optimization by learning a policy that predicts the optimum. Intuitively, you will solve the following optimization problem:

$$\max_{\theta^\mu} Q(s, a = \mu(s \mid \theta^\mu))$$

Using TensorFlow/PyTorch, you can directly take the gradient of this objective w.r.t. the policy parameters, θ^μ. If you work this out by hand by applying the chain rule, you will get the same expression as in the Algorithm 1. There are a few things to note:

1. You will learn an actor network with parameters θ^μ and a critic network with parameters θ^Q.

2. Similar to DQN, you will use a *target network* for both the actor and the critic. These target networks have parameters$\{\theta^{Q'}, \theta^{\mu'}\}$ are slowly updated towards the trained weights.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 1: DDPG algorithm presented by [3].

3. The algorithm requires a random process $\mathcal{N}$ to offset the deterministic actor policy. For this assignment, you can use an ϵ-normal noise process, where with probability ϵ, you sample an action uniformly from the action space and otherwise sample from a normal distribution with the mean as indicated by your actor network and standard deviation as a hyperparameter.

4. There is a replay buffer R which can have a burn-in period, although this is not required to solve the environment.

5. The target values values y_i used to update the critic network is a one-step TD backup where the bootstrapped Q value uses the slow moving target weights $\{\theta^{\mu'}, \theta^{Q'}\}$.

6. The update for the actor network differs from the traditional score function update used in vanilla policy gradient. Instead, DDPG uses information from the critic about how actions influence value estimates and pushes the policy in a direction to maximize increase in estimated rewards.

To implement DDPG, we recommend following the steps below:

1. Create actor and critic networks; for actor and critic network, use the `algo/criticnetwork.py` and algo/actornetwork.py respectively. For this environment, a simple fully connected network with two layers should suffice. You can choose which optimizer and hyperparameters to use, so long as you are able to solve the environment. We recommend using Adam as the optimizer. It will automatically adjust the learning rate based on the statistics of the gradients it's observing. You can check they create the network you wanted using the function `create_actor_network` and `create_critic_network` and printing the model architectures.

2. Connect the two implemented models in the DDPG method in `ddpg.py` script.

3. In the file run.py, implement the main training and evaluation loops.

The file `ReplayBuffer.py` does not need to be changed. Generally, you can find the places where we expect you to add code by "NotImplementedError". For this part you don't need to modify `add_hindsight_replay_experience` function. Train your implementation on the `Pushing2D-v0` environment until convergence[2], and answer the following questions:

1. (15 pts) The neat trick of DDPG is that you can learn a policy by taking gradients of the Q function directly w.r.t. the policy parameters. An alternative approach that seems easier would if we could directly take gradients of the *cumulative reward* w.r.t. the policy parameters, without having to learn a Q function. Why is this approach not feasible? Optional (0 pts): How could you make this approach feasible?

2. (10 pts) In 2-3 sentences, explain how you implemented the actor update.

3. (5 pts) Describe the hyperparameter settings that you used to train DDPG.

4. (20 pts) Plot the mean cumulative reward: Every k episodes, freeze the current cloned policy and run 10 test episodes, recording the mean/std of the cumulative reward. Plot the mean cumulative reward μ on the y-axis with $\pm\sigma$ standard deviation as error-bars vs. the number of training episodes on the x-axis. You don't need to use the noise process $\mathcal{N}$ when testing. Hint: You can use matplotlib's `plt.errorbar()` function. `https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html`

5. (10 pts) You might have noticed that the TD error is *not* a good predictor of whether your DDPG agent is learning. What other metric might you use to measure performance *without collecting new transitions from the environment*? *Why* is this a reasonable metric? Implement this metric, and then determine whether this metric is actually useful for predicting the agent's performance.

6. (Extra credit: up to 5 pts) DDPG is known to overestimate the value of states and actions. A recent method, TD3 [2], proposes a correction that avoids this overestimation by learning two Q functions, $Q_1(s, a)$ and $Q_2(s, a)$, and then choosing actions according to the minimum (i.e., acting pessimistically):

$$\max_a \min(Q_1(s, a), Q_1(s, a))$$

[2] `Pushing2D-v0` is considered solved if your implementation can reach the *original* goal at least 95% of the time. Note that this is not the same as reaching the *hindsight* goal.

Extend your DDPG implementation to implement TD3, and conduct an experiment to compare the two algorithms. Provide a plot comparing the two algorithms and write a few sentences explaining your results.

Problem 2: Hindisght Experience Replay (HER) [40+2 pts]

In this section, you will combine HER with DDPG to hopefully learn faster on the `Pushing2D-v0` environment (see Figure 2 for the full algorithm). The motivation behind hindsight experience replay is that even if an episode did not successfully reach the goal, we can still use it to learn something useful about the environment. To do so, we turn a problem that usually has sparse rewards into one with less sparse rewards by hallucinating different goal states that would hopefully provide non-zero reward given the actions that we took in an episode and add those to the experience replay buffer.

To use HER in our setup, set hindsight=True in the train method. For this part, you will need to implement the `add_hindsight_replay_experience` function. To help you form new transitions to add to the replay, the code for the `Pushing2D-v0` environment provides a method, `apply_hindsight`, to compute the reward given a new goal state ($r(\cdot)$ Fig. 2) and to modify each state to set the goal to be the state actually reached.

1. (5 pts) Describe the hyperparameter settings that you used to train DDPG with HER. Ideally, these should match the hyperparameters you used in Part 1 so we can isolate the impact of the HER component.

2. (15 pts) Plot the mean cumulative reward: Every k episodes, freeze the current cloned policy and run 100 test episodes, recording the mean/std of the cumulative reward. Plot the mean cumulative reward μ on the y-axis with $\pm\sigma$ standard deviation as errorbars vs. the number of training episodes on the x-axis. Do this on the same axes as the curve from Part 1 so that you can compare the two curves.

3. (5 pts) How does the learning curve for DDPG+HER compare to that for DDPG?

4. (10 pts) In the typical multi-goal RL setup, we are given a distribution over goals, $p(g)$. HER trains on a different distribution over goals, $\hat{p}(g)$. Mathematically define $\hat{p}(g)$. When will $\hat{p}(g)$ be very different from $p(g)$? Why might a big difference between $\hat{p}(g)$ and $p(g)$ be problematic?

5. (Extra credit: up to 2 pts) How might you solve this distribution shift problem?

6. (5 pts) What are settings where you cannot apply HER, or where HER would not be able to use it to speed up training?

Algorithm 1 Hindsight Experience Replay (HER)

Given:

- an off-policy RL algorithm $\mathbb{A}$, ▷ e.g. DQN, DDPG, NAF, SDQN
- a strategy $\mathbb{S}$ for sampling goals for replay, ▷ e.g. $\mathbb{S}(s_0, \ldots, s_T) = m(s_T)$
- a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$. ▷ e.g. $r(s, a, g) = -[f_g(s) = 0]$

Initialize $\mathbb{A}$ ▷ e.g. initialize neural networks
Initialize replay buffer R
for episode = $1, M$ **do**
  Sample a goal g and an initial state s_0.
  for $t = 0, T-1$ **do**
    Sample an action a_t using the behavioral policy from $\mathbb{A}$:
      $a_t \leftarrow \pi_b(s_t||g)$ ▷ $||$ denotes concatenation
    Execute the action a_t and observe a new state s_{t+1}
  end for
  for $t = 0, T-1$ **do**
    $r_t := r(s_t, a_t, g)$
    Store the transition $(s_t||g,\ a_t,\ r_t,\ s_{t+1}||g)$ in R ▷ standard experience replay
    Sample a set of additional goals for replay $G := \mathbb{S}$(**current episode**)
    for $g' \in G$ **do**
      $r' := r(s_t, a_t, g')$
      Store the transition $(s_t||g',\ a_t,\ r',\ s_{t+1}||g')$ in R ▷ HER
    end for
  end for
  for $t = 1, N$ **do**
    Sample a minibatch B from the replay buffer R
    Perform one step of optimization using $\mathbb{A}$ and minibatch B
  end for
end for

Figure 2: Hindsight Experience Replay [1].

General Advice

Due to the more complicated objective function for DDPG, you will likely have to write some of the code in Tensorflow so you have more control over the training procedure. This tutorial could be useful for you to reference as you are implementing the training procedure for the actor network in DDPG.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[2] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[3] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.