10-403 Deep Reinforcement Learning and Control
Homework 5: Advanced Policy Gradients
Spring 2019

March 29th, 2019

Due by 11:59 PM on April 15th, 2019

Instructions

You may work in teams of **2** on this assignment. Only one person should submit the writeup on Gradescope. Additionally, the same person who submitted the writeup to Gradescope must upload the code to Autolab. Make sure you mark your partner as a collaborator on Gradescope (you do not need to do this in Autolab) and that both names are listed in the writeup. Writeups should be typeset in LaTeXand submitted as PDF. All code, including auxiliary scripts used for testing, should be submitted with a README. Please limit your writeup to 8 pages or less (excluding the provided instructions).

We've provided some code templates that you can use if you wish. Abiding to the function signatures defined in these templates is not mandatory; you can write your code from scratch if you wish. You can use any deep learning package of your choice (e.g., Keras, Tensorflow, PyTorch). You should not need the cluster or a GPU for this assignment. The models are small enough that you can train on a laptop CPU.

It is expected that all of the work you submit is your own. Submitting a classmate's code or code which copied from online and claiming it is your own is not allowed. Anyone who does this will be violating University policy, and risks failure of the assignment, course and possibly disciplinary action taken by the university.

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')
```

Then you can interact with the environment as you usually do.

An overview of the environment is available in the recitation slides. 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) [40 pts]

Previously, in Homework 3, you worked with Deep Q-networks, which is an off-policy method for discrete spaces that learns the actual value of actions given a certain state and then acts greedily with respect to the learned values. Then, in Homework 4, you implemented on-policy policy gradient methods that learn a good policy directly over a discrete action space. In this section, you will implement DDPG, an off-policy policy gradient method for a continuous action space. Similar to DQN, you will

1. use a replay buffer from which you will sample transitions in order to learn off-policy and minimize correlations between samples

2. train with a slowly updated target Q network to give consistent targets when performing the temporal difference backups

However, since this is a policy gradient method, you will be learning a policy directly, instead of learning Q-values and then acting greedily.

The DDPG algorithm is shown in Figure 1. There are a few things to note:

1. Similar to DQN, there are two sets of weights: $\{\theta^Q, \theta^\mu\}$ are trained and target weights $\{\theta^{Q'}, \theta^{\mu'}\}$ are slowly updated towards the trained weights.

2. 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.

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

4. 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 $\{\theta^{\mu'}, \theta^{Q'}\}$.

5. 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.

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 [2].

For this environment, a simple fully connected network with 2 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. Think of it like a fancier SGD with momentum.

Train your implementation on the `Pushing2D-v0` environment until convergence[1], and answer the following questions:

1. Describe your implementation, including the optimizer and any hyperparameters you used (learning rate, γ, etc.).

[1] `Pushing2D-v0` is considered solved if your implementation can reach the goal at least 95% of the time.

Solution

2. Plot the learning curve: 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 error-bars vs. the number of training episodes. 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

Solution

Problem 2: Hindisght Experience Replay (HER) [20 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.

In your implementation of HER, you can use the implementation of DDPG that you have from Part 1. Then, the only thing you need to do is to form hallucinated transitions that use a different goal state and add them to the experience replay buffer as well. Your set of goals for replay G will only include the goal corresponding to the ending local of the box. Use the goal to form new transitions to add to your experience buffer. Notice that the new transitions assume that the transition dynamics do not depend on the reward r_t that you you observe in each timestep.

To help you form new transitions to add to the replay, the code for the `Pushing2D-v0` environment provides a method to compute the reward given a new goal state. This is the function $r(\cdot)$ in the hindsight experience replay algorithm shown in Figure 2. You can refer to the recitation slides for more information on this as well.

1. 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.

Solution

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: The hindsight experience replay algorithm presented by [1].

2. Plot the learning curve: 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 error-bars vs. the number of training episodes. Do this on the same axes as the curve from Part 1 so that you can compare the two curves.

Solution

3. How does the learning curve for DDPG+HER compare to that for DDPG? What are the limitations of HER? Give some scenarios in which you would or would not be able to use it to speed up training.

Solution

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] 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.