Deep Reinforcement Learning
Assignment 3

2023 Spring

April 28, 2024

In this homework, you will implement the DDPG [1], TD3 [2] and SAC [3] algorithms to solve the `LunarLanderContinuous-v2` environment from Gymnasium.

1 About the Environment

The Lunar Lander environment is a classic rocket trajectory optimization problem. The state is an 8-dimensional vector: the x, y coordinates of the lander, its linear velocities in the two directions, its angle, its angular velocity, and two booleans that represent whether each leg is in contact with the ground or not. Since you are implementing RL algorithms with continuous action spaces, we use the continuous version of this environment, named `LunarLanderContinuous-v2`.

The fuel is infinite, so an agent can learn to fly and then land on its first attempt. After every step, a composite reward is granted for the posture, velocity and engine status of the rocket. The episode receives an additional reward of -100 or +100 points for crashing or landing safely. An episode is considered successful if it scores at least 200 points. For more details on the reward, observation and action spaces, please refer to the documentation.

2 About the Starting Code

The codebase of Homework 3 is an extension of Homework 2. You will find much of the code familiar. Again, we are using the `hydra` framework to manage the configuration of the experiments. Please refer to `hydra`'s documentation for more details.

The results and saved files will be stored in the `runs` folder, under the subfolder specified by the time of execution. You can find the training curves and a video of the trained agent in the subfolder. If you want to turn off this behavior and save everything in the current folder, you can change the `hydra.run.chdir` field in the `config.yaml` file to `false`.

To improve the credibility of the results, we used three random seeds and plotted the current results' average and standard deviation periodically during training. We provide two versions of the program entry: `main.py` and `main_mp.py`. The former runs the experiments for the three seeds sequentially and updates the resulting standard deviation from the second seed on. This version is more suitable for debugging your code. The latter runs all three seed experiments

in parallel and is suitable for quickly iterating and getting the results. You can choose either version to run, and after all three seeds, their results should be the same. You don't need to change anything in the program entries.

3 Requirements

As in Homework 2, we will combine automatic grading with manual grading, which requires you to not only pass the test cases but also provide a brief report on your results and findings. The total score of the test cases is 30 pts, while the report will be graded up to 70 pts. Please submit a `zip` file containing the following files or folders:

- The auto-generated `runs` folder which contains the results for each of the required experiments and their corresponding config, log, and model files. You may remove the generated videos but leave the other files untouched. Please make sure the code you provide can reproduce the contents of the `runs` folder given the corresponding config file.
- A `report.pdf` file briefly summarizing your results. It should include the auto-generated results figure. You may also include any other findings you think are relevant.

You can find the reference training curves in the `gallery` folder. The TAs are using five seeds to generate results for the gallery, so we don't require your curves to be the same as ours anyway. However, if your results and losses are drastically different from the reference curves, there may be something wrong with your implementation.

For each of the algorithms, we expect the average return of your runs to exceed 200 points at the end of the training (200_000 steps).

Please refer to the `README.md` file for dependencies of this homework.

4 Deep Deterministic Policy Gradient [30 pts]

In this section, you will implement a DDPG [1] agent which outputs continuous actions and solve the `LunarLanderContinuous-v2` environment. The DDPG algorithm is like a continuous version of the DQN algorithm, with a neural-network-based "Actor" simulating the maximum operation needed for Q value update. You'll need to read through the following files and implement the missing parts:

`models.py` This file contains the model definitions for the actor (policy) and critic (Q-value) networks. Read through the `Actor, Critic` classes to get a sense of how the network is structured and the output is normalized. You don't need to implement anything in this file for this assignment.

`buffer.py` This file contains the replay buffer classes we use to store trajectories and sample mini-batches from them. Please note that the implementation of the advanced buffers other than the basic `ReplayBuffer` is optional for this Homework. You can refer to your implementations from Homework 2 and make modifications accordingly (You don't need to change anything if you are not using the advanced buffers).

`agent/ddpg.py` This file contains the core `DDPGAgent` class that manages how to take actions and how to update the network. For the `DDPGAgent` class, you'll need to implement the `get_action`, `get_actor_loss`, and `get_Qs` methods. You can also read through the `update` method to get a sense of how the agent does a one-step update.

`utils.py` This file implements a handful of tool functions we use in the training process. You don't need to implement anything in this file for this assignment.

`core.py` This file contains the main training and evaluation loop. You don't need to implement anything in this file for this section. Read through the `train` and `eval` functions to get a sense of how the training and evaluation process is structured.

`main.py` This is the main file that you'll run to start the training process. You don't need to implement anything in this file for this section.

After implementing the necessary parts of the DDPG agent, you can run the following command to start the training process:

```
python main.py
```

We recommend you try out our parallel version of `main.py`:

```
python main_mp.py
```

where we use the `multiprocessing` library to train all seeds together in parallel. Feel free to read through the code of `main_mp.py` and `core_mp.py` as well.

You are also welcome to try the improvements on the replay buffer that you coded in Homework 2 – e.g., prioritized experience replay and n-step return.

5 Twin Delayed DDPG [20 pts]

To overcome the problem of overestimating Q-values, we can use the TD3 [2] algorithm. TD3 introduces three improvements upon DDPG:

1. **Clipped Double-Q Learning.** TD3 learns two Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions:

$$Y_t = r_t + \gamma(1 - d_t) \min_{i=1,2} Q_{w_i^-}(s_{t+1}, a_{\text{targ}}(s_{t+1}))$$

   where w_i^- are the parameters of the target Q-networks, r and d are the reward and done signals, and a_{targ} is a function to get the target action.

2. **Delayed Policy Updates.** TD3 updates the policy (and the target policy) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.

3. **Target Policy Smoothing.** TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along with changes in action.

$$a_{\text{targ}}(s') = \text{clip}(\pi_{\theta^-}(s_{t+1}) + \text{clip}(\epsilon, -c, c), a_{\text{low}}, a_{\text{high}}), \quad \epsilon \sim \mathcal{N}(0, \sigma)$$

where θ^- is the target policy parameters, σ is the standard deviation of the noise, and c is the clipping range of policy noise.

In this section, please read through the following files and implement the missing parts:

`agent/td3.py` Implement the `get_Qs` function with clipped double-Q learning and target policy smoothing. You also need to complete the `update` function for delayed policy updates.

After implementing the necessary parts, you can run the following command to start the training process:

```
python main_mp.py agent=td3
```

This will override the default config of the `agent` field to `td3` and start the training process.

6 Soft Actor-Critic [20 pts]

In this section, you will implement the SAC [3] algorithm. SAC adds an entropy term to the objective function:

$$J(\theta) = \arg\max_{\theta} \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_t,a_t,s_{t+1}) + \alpha\mathcal{H}(\pi_\theta(\cdot|s_t))\right)\right]$$

In practice, we can approximate the entropy term by using the log probability of the action. The modified target value for the Q-function is:

$$Y_t = r_t + \gamma(1-d_t)\min_{i=1,2}\left[Q_{w_i^-}\left(s_{t+1},\pi_\theta(s_{t+1})\right) - \alpha\mathcal{H}(\pi_\theta(\cdot|s_{t+1}))\right]$$

where α is a parameter that controls the relative importance of the entropy term. The modified policy loss is:

$$L_\theta = \alpha\mathcal{H}(\pi_\theta(\cdot|s_t)) - Q_\phi(s_t,\pi_\theta(s_t))$$

In the later version of SAC [4], the temperature α is also dynamically adjusted, and the corresponding loss for α is:

$$L_\alpha = -\alpha\left(\mathcal{H}(\pi_\theta(\cdot|s_t)) + \text{target\_entropy}\right)$$

Where target_entropy is a hyperparameter that controls the entropy of the policy. In the original paper, target_entropy is set to $-|\mathcal{A}|$, which is the entropy of a uniform distribution over the action space.

Please read through the following files and implement the missing parts:

`models.py` The `SoftActor` class for SAC inherits the `Actor` class and uses a stochastic policy network. Implement the `forward` and `evaluate` methods of `SoftActor`.

`agent/sac.py` Implement the `get_Qs`, `get_actor_loss`, and `get_alpha_loss` function. Take note of the difference between SAC and the other algorithms like DDPG and TD3 – SAC has an extra entropy term in objective functions.

After implementing the necessary parts, you can run the following command to start the training process:

```
python main_mp.py agent=sac
```

7 Tips for implementation

- This homework will require slightly more time for training. For your information, it took us about 40 minutes to train the DDPG and TD3 agents for 0.2M steps, and 70 minutes for the SAC agent. We suggest you first modify the `train.timesteps` to a smaller value like 50_000 or 100_000 for debugging and then increase it to 200_000 if the tendency of the training curves looks promising.

- You can basically reuse the `buffer.py` from your hw2, including the advanced buffers like `PrioritizedReplayBuffer` and `NStepReplayBuffer`. However, since we've made a few improvements and modifications in the class definition, it's better to copy the code you implemented in hw2 to this homework and modify it if needed.

- During the implementation of SAC, you may find the `torch.distributions` module helpful. For example, you can use the `Normal` class to sample from a Gaussian distribution with mean μ and standard deviation σ. You can use the `log_prob` method to compute the log probability of a given action under the distribution. However, please note that after applying the `tanh` transformation to the output of the policy network, the sampled probability is no longer the same as the original Gaussian distribution. You can refer to the Function of Random Variables wiki page for more detail.

- Numerical error could be a big problem because of the `tanh` normalization, make sure to add a small epsilon value (like `1e-6`) when you are calculating the logarithm of a potentially very small number. You can also use the `TanhTransform` class from the `torch.distributions.transforms` module to transform the distribution, it will do the mathy part for you. However, this method is not recommended, as it will introduce numerical instability. If you do want to use this method, please make sure to add the `cache_size=1` parameter to the `TanhTransform` class, otherwise, you will likely encounter a `NaN` loss soon. And we still find that doing the transform manually is more stable.

8 Bonus [10 pts]

You can get 10 bonus points by implementing any of the following extensions, the total score will be capped at 100 points.

- Due to numerical errors, the returned `log_prob` value from the `SoftActor` class can be larger than zero sometimes (as can be seen in the gallery image, the corresponding `action_prob` could be larger than one). And this value can be very large sometimes when the action is close to the boundary of the action space. This means that this term isn't a faithful representation of the real possibility of the action. Investigate this problem and its influence on the SAC algorithm, and try to find a way to fix it.

- The evaluation process in the `core.py` and `core_mp.py` is sequential, which is not efficient because the policy model is receiving only one state at each time. You can try to implement a parallel evaluation method using the `gym.vector.SyncVectorEnv` class. To instantiate a vector environment of `eval_episodes` environments and use the evaluation result from the first trajectory of each environment.

- For a minimal implementation of the algorithms, we have stripped many tricks from the original implementation, like grad clipping, special methods for parameter initialization, reward scaling, etc. You can try to implement such tricks and see if they can improve the performance by a significant margin. If so, please report your findings in the report and the relative ablation study results.

- If you think there's any bug in the code, please let us know and provide your solutions.

References

[1] 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, July 2019.

[2] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

[3] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, August 2018.

[4] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, G. Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, P. Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *ArXiv*, abs/1812.05905, 2018.