Deep Reinforcement Learning
Assignment 4

2023 Spring

May 31, 2024

In this homework, you will implement the clipped variant of the Proximal Policy Optimization Algorithm [1] 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. There are two environment versions, with the action space being discrete or continuous. Since you are implementing RL algorithms with continuous action spaces, we use the continuous version, named `LunarLanderContinuous-v2`.

Fuel is infinite, so an agent can learn to fly and then land on its first attempt. After every step, a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode. The episode receives an additional reward of -100 or +100 points for crashing or landing safely respectively. An episode is considered a solution 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 4 is an extension upon Homework 3. You will find much of the code familiar. We've added headers of `PPOAgent` and `PPOReplayBuffer`, as well as support for vectorized environments to speed up the training. As you are concurrently working on your project, more of the code is already written for you, so you only need to fill in the key parts.

3 Requirements

As in Homework 2 and 3, 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. Since we are fixing all the seeds, your resulting curves could be exactly the same. We don't require this, but if your results and losses are drastically different from the reference curves, there may be something wrong with your implementation.

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

4 Clipped Variant of PPO Algorithm [70 pts]

The clipped variant of PPO uses the combined objective to train its policy and value networks:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{1}$$

where

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t, \text{clip}(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon) \hat{A}_t \right) \right], \tag{2}$$

$$L_t^{VF}(\theta) = (V_\theta(s_t) - V_t^{\text{targ}})^2, \tag{3}$$

c_1, c_2 are coeffients, and S is the entropy bonus.

For the advantage $\hat{A}_t$, PPO uses a truncated version of generalized advantage estimation (GAE) [2]:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t-1}\delta_{T-1} \tag{4}$$

$$\text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{5}$$

Please note the differences between this assignment and the previous ones:

- The `update` method for the PPO algorithm takes in a full replay buffer with rollouts collected by recent policies as it is "on-policy-ish". After the update, the buffer needs to be cleared.
- To sample more data from the environment, in this assignment we use a synchronous vectorized environment (`gym.vector.SyncVectorEnv`) containing 16 concurrent (but independent) environments. This environment will output a batch of 16 observations per step and ask the agent to give a batch of 16 actions for stepping in the environment. For the coding details of dealing with such an environment, you are welcome to read through `core.py`.
- The PPO algorithm uses a value network that outputs state-values, instead of normal critic networks that output Q-value estimates for different actions. The `ValueNet` class in `models.py` inherits the `Critic` class but simplifies it as we no longer input actions.

To complete this assignment, you should look through the following files:

`models.py` This file contains the neural network models for the actor (policy) and value networks. Read through the `PPOActor` and `ValueNet` classes to learn the differences between them and the `Actor`, `Critic` classes. 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. The `PPOReplayBuffer` contains more data fields and is designed to work with vectorized environments. Since PPO is more "on-policy", the `PPOReplayBuffer` needs to be cleared and refilled after a few gradient steps. You will need to fill in the `compute_advantages_and_returns` function of the `PPOReplayBuffer` class, which plays a key role in the learning process, also please have a look at the `get_next_values` on how to calculate the next values used for computing advantages and returns.

`agent/ppo.py` This file contains the core `PPOAgent` class that manages how to take actions and how to update the network. For the `PPOAgent` class, you'll need to implement the `get_policy_loss`, `get_value_loss`, and `get_entropy_loss` methods. After you finish the implementation, you can read through the `update` and the `update_step` methods to get a sense of how the agent does a batch 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.

After implementing the necessary parts of the PPO 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`. For this assignment, we also removed `core_mp.py` file and instead added a flag of whether we are running multiple processes into `core.py`.

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`.

An example of the training curves named `PPO.png` is shown in the `gallery` subfolder. Make sure that your agent is able to reach an average return greater than 200.

5 Bonus [10pts]

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

- There is also a variant of PPO using the KL divergence as a penalty and an adaptive KL penalty coefficient. We implemented the clipped variant in the codebase, but you are welcome to attempt the adaptive KL version and compare its performance with the clipped version.

- The clipping constant ϵ is set to 0.2 in the code. How frequently is the ratio actually clipped? Is this pattern the same throughout the training, or does clipping happen more in the beginning? You could thoroughly investigate how setting the value of ϵ – maybe even changing it during the training process – will affect the performance of the agent.
- Another implementation detail used by PPO is the clipped value loss. It is turned off by default in our code. You can try to implement the value-clipped loss and compare its performance to the original PPO agent.
- Our implementation of PPO focuses on the continuous Lunar Lander environment. However, PPO also works in discrete action spaces. You can try to implement a discrete version of the clipped PPO variant, and run it on the discrete Lunar Lander environment to evaluate performance.
- If you think there's any bug in the code, please let us know and provide your solutions.

References

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[2] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.