CMU 17-759 - Advanced Topics in ML and Game Theory
Programming Assignment 1

October 10, 2022

1 Proximal Policy Optimization for Hanabi

Proximal Policy Optimization [5] is an on-policy reinforcement learning (RL) algorithm. PPO is popular in single-agent settings as a high-performance RL algorithm that has been the driving factor behind impressive feats such as defeating the Dota-II world champion [2].

Widely-used Multi Agent RL (MARL) algorithms such as Q-MIX [4] and Multi Agent Deep Deterministic Policy Gradient (MADDPG) [3] are largely based on their respective single agent algorithms (Deep Q learning and DDPG). Despite its successes in single-agent RL, extensions of PPO to multi-agent settings remain relatively less explored.

Hanabi is turn-based card game played by 2-5 players. The game is fully cooperative. It is an imperfect-information game where each player (agent) is not able to see their own cards. To learn successful strategies, each agent must learn to cooperate with other agents by means of implicit communication, through their own actions. You can watch this video to get a better idea of how the game is played. Hanabi is used as a popular benchmark for the development of MARL algorithms [1].

The aim of this assignment is to understand how PPO works, to implement a multi-agent extension of PPO (MA-PPO) and to run this algorithm on Hanabi.

Section 2 gives a brief introduction to MA-PPO. Section 3 explains how to set up the environment. The code for this assignment is based on the official code implementation for [6].

2 Multi-Agent PPO

This section briefly explains how PPO can be extended to MA-PPO.

Key differences:

- Since we are now in a multi-agent setting, each agent has an actor network and a critic network.
- In the single-agent case, the input to the actor is the state of the environment, if the state is fully observable. Otherwise, the input to the actor is an "observation", i.e., the observable part of the state. In Hanabi, the state of the environment is not fully observable. The input to each agent's actor is the local observation of that agent, i.e., the part of the "global" state that the given agent sees. Each agent might see different parts of the state, leading to different local observations for each agent.
- MARL algorithms typically follow a Centralized Training, Decentralized Execution (CTDE) paradigm. The idea is that even though agents should only have access to

their local observation while taking actions in the environment, they may have access to the global state while training.

In MA-PPO, the critic of each agent is given access to the global state. Sometimes, we use the concatenated observations of all agents, called the joint observation, as input to the critic if the global state is not available. We know that only the actor is required when an agent chooses an action to execute in the environment. Thus, using the global state as input to the critic does not violate the CTDE paradigm.

This idea of using a centralized critic was introduced in [3], as a means to get a better estimate of the global Q function. It could also help in reducing non-stationarity in the environment arising due to the fact that from the perspective of each agent, the policies of other agents keep changing over time as they learn.

- In some problems, all agents are of the same type (homogenous). To simplify training, one commonly used technique is to have all agents share the same critic and actor. In fact, such parameter sharing is widely used in MARL, especially when the number of agents is large.
- In short, the only major difference from single-agent PPO will be that the input to the critic will be a joint observation/global information rather than the local observation.

3 Getting Started

In order to circumvent installation and compatibility issues, we have provided a Google Colab Notebook with steps to run the code. Before running the cells in the notebook, please make sure you upload the PA1 folder to Google Drive. The code does the following:

1. Installation of the "PA1" package.
2. Installation of Hanabi.
3. Installation of a few required packages.

You will find that the compute provided for free by Colab is sufficient to run your code and get the results needed to complete the assignment. Alternatively, you may choose to run the code on your local machine. You need to have Python and Anaconda installed on your local machine.

- Create a new Conda virtual environment and install CUDA as show in Fig. 1(The version in the figure is 10.1, but you may need to install a different version depending on the configuration of other packages on your local machine.

```
# create conda environment
conda create -n pa1 python==3.6.1
conda activate pa1
pip install torch==1.5.1+cu101 torchvision==0.6.1+cu101 -f https://download.pytorch.org/whl/torch_stable.html
```

- Install the required dependencies provided in "PA1/requirements.txt". You may install additional dependencies during run-time seperately but this should cover most of them.
- Build and install Hanabi.

```
# install on-policy package
cd PA1
pip install -e .
```

```
#Hanabi installation
pip install cffi
cd envs/hanabi
mkdir build & cd build
cmake ..
make -j
```

- Run the python training code as shown under the "Running Hanabi" section of the Google Colab notebook.

The **PA1/onpolicy** folder is structured as follows:

- **algorithms** sub-folder contains the required code for the MA-PPO algorithm.
- **envs** contains the Hanabi environment code.
- **runner** contains the code for training and policy updates, calling the code in the "algorithms" sub-folder.
- **scripts/train** contains the python training script.
- **utils** contains helper code such as code for the replay buffer.
- **onpolicy/config.py** contains information about all different hyperparameters and their default values.

For this assignment, you will be only be required to work with the code in the **algorithms** subfolder. Again, we very strongly recommend working with Google Colab, since there are a large number of dependencies.

4 Submission Instructions

You are required to submit both your report in PDF format and your code. Prepare a .zip file containing the report and the code and submit by **11:59 pm on Oct 10, 2022**.

5 Task 1 [50 points]

Algorithm 1 PPO-Clip

1: **Input:** Initial policy parameters θ_0 initial value function parameters ϕ_0
2: **Parameters:** m **= number of mini-batches,** l **= number of trajectories to be collected in each iteration**
3: **for** iteration $k = 1, 2, \ldots$ **do**
4: Collect l trajectories. A trajectory τ is collected by running policy π_{θ_k} in environment for T time steps, i.e., $\tau = \langle s_0, a_0, r_1, s_1, a_1, r_2, ...s_T, a_T, r_{T+1}, s_{T+1}\rangle$
5: For each trajectory, compute returns $\hat{R}_1, \ldots, \hat{R}_T$ where $\hat{R}_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}...$ and advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$ where $\hat{A}_t = \hat{R}_t - V_{\phi_k}(s_t)$
6: Split the collected trajectories into minibatches $M_1, M_2, \ldots, M_m$
7: $\theta \leftarrow \theta_k$, $\phi \leftarrow \phi_k$
8: **for** *ppo_epoch* $= 1, 2, \ldots, N$ **do**
9: **for** $i = 1, 2, \ldots, m$ **do**
10: Compute estimated value of objective $\hat{J}_{CLIP}(\theta) = \mathbb{E}[J_{CLIP}(\theta)] \approx$
11: $\frac{1}{|M_i|} \sum_{(s_t, a_t, r_t) \in M_i} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \text{clip}(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, 1-\epsilon, 1+\epsilon) A^{\pi_{\theta_k}}(s_t, a_t) \right)$
12: $L_{CLIP}(\theta) = -1 * \hat{J}_{CLIP}(\theta)$
13: Update actor: $\theta \leftarrow \theta - \nabla_\theta L_{CLIP}(\theta)$
14:
15: Compute TD error: $L_{critic} = \frac{1}{|M_i|} \sum_{(s_t, a_t, r_t) \in M_i} \left(V_{\phi_k}(s_t) - \hat{R}_t \right)^2$
16: Update critic: $\phi \leftarrow \phi - \nabla_\phi L_{critic}$
17: **end for**
18: **end for**
19: $\theta_{k+1} \leftarrow \theta$, $\phi_{k+1} \leftarrow \phi$
20: **end for**

Algorithm 1 is the pseudocode for single-agent PPO. In Line 6, we split the collected trajectories into minibatches. Each mini-batch will have T steps of data where each data point is a tuple of $(s_t, a_t, r_t, s_{t+1}, \hat{R}_t, \hat{A}_t)$. For environments where we cannot observe the state, state s_t is replaced with observation o_t, and we often use a recurrent neural network for the actor and critic to incorporate the previous observations.

For Tasks 1 and 2, you will be working with MA-PPO and use MA-PPO to train agents to play a simplified version of the Hanabi. As discussed in Section 2, MA-PPO differs from single agent PPO in that the input to the critic can include all the agents' observations, actions, and any other global information following the CTDE paradigm. Since Hanabi is a fully cooperative game and all the players in the game play a similar role, we have all the players share their actor and critic, and only use a single V_ϕ to represent the state value. So the pseudocode of MA-PPO for Hanabi is very similar to Algorithm 1 and the only difference is in the input to the actor and critic networks. Specifically, the input to the actor network will be a player's local or private observation, and the input to the critic network will be the joint observations and other global information.

We ask you to complete the code for MA-PPO. In particular, you will be implementing the code for lines 10 to 16 for MAPPO, while the rest of the code is already provided to you.

As mentioned earlier, we use a single actor and a single critic shared by all agents. The main difference from single-agent PPO is that in lines 11 and 15, the inputs used to calculate the advantage and value function are the **global state** and **joint action**. The

input to the policy is the **local observation** of each agent. All required inputs including returns and advantages are already calculated and are provided to you.

Keep in mind that Hanabi is a turn-based game. Our T is set to 100, but at each timestep, each agent has one turn of play. Please do not change this value of T. Another point to note is that T time steps does not necessarily correspond to one episode of game-play from initial states at $t = 0$ to terminal states at $t = 99$. You could, for example, have a game ending at $t = 31$ and the next game starting at $t = 32$. You will not need to worry about this either, since it is already implemented in the code.

For Task 1, you will be required to run and implement the MA-PPO algorithm with a clipped objective function by completing certain functions in **algorithms/r_mappo/r_mappo.py**. To gain a better understanding of PPO, you may refer to [5] and the Open-AI PPO blog post. You may refer to [6] for implementation details of the MA-PPO algorithm.

You will definitely have noticed by now that the code for the paper is available on Github. We will consider an honor system whereby we rely on you to **not plagiarize the code**.

Task 1.1 (10 points) Complete the code for the cal_value_loss method. This method calculates the loss for training the Critic neural network, L_{critic}, as shown in line 15 of the pseudocode in Fig. 1.

Task 1.2 (25 points) Complete the code for the ppo_update method. This method first computes the estimated MA-PPO-clipped objective, $\hat{J}_{CLIP}$, as shown in line 11 of the pseudocode. It then computes L_{CLIP}, the actor loss, and updates the actor network as shown in lines 12 and 13.
Once this is done, the function updates the critic network, as shown in line 16, after calling the cal_value_loss method in Task 1.1 to get the loss for the critic.

Task 1.3 (15 points) Once you implement parts (1) and (2), run the code, setting env_name to "Hanabi-Very-Small" and the number of agents to 2. Plot the reward curve over 30k environment steps and report it. It will not take you more than 15 minutes to do this, as long as you are using a Colab GPU.

Task 1.4 (5 points) Once you have completed Task 1.3, attempt to improve over the results you obtained in Task 1.3. For doing this, you may conduct hyperparameter tuning, run the experiment for longer (say 100k steps) or do both. You may also experiment with any other tricks and techniques that you think will improve performance. Include the new reward curve in your report and mention how you improved over Task 1.3.

Hints and Remarks (May be updated later)

1. In the implementation, l, the number of trajectories collected, can be changed by setting the argument "–n_rollout_threads" to l. (Do not change the "–n_training_threads" argument from 1.)

2. The number of mini-batches cannot be more than $l * N$, where N is the number of agents. The code is implemented in such a way that each mini-batch will have T steps of data (observations, states, actions, rewards etc.) for one agent, for any one of the l copies of data-collecting π_{θ_k}. Using this data, we estimate $\hat{J}_{CLIP}$ and L_{critic} for the given mini-batch, compute the derivative w.r.t. θ and ϕ, respectively, and update the parameters.

3. You do not need to use wandb to plot the data. We would like you to use matplotlib. You do not need to create a wandb account and when prompted, you can run the code by entering option (3), as Steven already mentioned.

For 2, refer to hanabi_runner_forward.py.

4. At every iteration, the policy collects a certain amount of data which is equal to [100 (which is the episode_length) *

n_rollout_threads] tuples. The code then computes the average score for each iteration, which is the average over episode rewards, where episode reward is the sum of rewards over 100 steps. All you need to do is to plot this data using matplotlib. So that the data you're plotting will look like: (mu1, 100), (mu2, 200), ..., (mu_n, 100*n); your x-axis being steps in multiples of 100 and y-axis being the average scores.

5. No need to compare the score with the score in the paper. The environment is the full Hanabi environment in the paper. Additionally, they have used a variety of deep learning tricks and parameter tuning, with a large amount of data, and trained for millions of steps

6 Task 2 [50 points]

The PPO paper [5] implements a variant of PPO that uses a KL-penalty coefficient as part of the objective function. The objective function is estimated as:

$$\hat{J}_{KLPEN}(\theta) = \frac{1}{|M_i|} \sum_{(s_t,a_t,r_t)\in M_i} \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t,a_t) - \beta \text{ KL } [\pi_{\theta_k}(a_t|s_t), \pi_\theta(a_t|s_t)]\right.$$

For two probability distributions $q(X)$ and $p(X)$: $KL(q(X)||p(X)) = E_{q(X)}[\log \frac{q(X)}{p(X)}]$

Task 2.1 (25 points) Modify the code for the ppo_update method to implement MA-PPO with a **Fixed** KL penalty. [5] also uses an **Adaptive** KL penalty, where β is updated after each PPO update, but we do not need you to update β (hence Fixed KL penalty).

The difference from Task 1.2 is that this method computes $\hat{J}_{KLPEN}(\theta)$ instead of $\hat{J}_{CLIP}(\theta)$ in line 11 and $L_{KLPEN}(\theta) = -1 * \hat{J}_{KLPEN}(\theta)$ in line 12. The variable "self.beta" in the code represents β.

Task 2.2 (10 points) Once you implement Task 2.1, run the code, setting env_name to "Hanabi-Very-Small" and the number of agents to 2. Plot the reward curve over 30k environment steps and report it.

Task 2.3 (5 points) Once you have completed Task 2.2, attempt to improve over the results you obtained in Task 2.2. For doing this, you may conduct hyperparameter tuning, run the experiment for longer (say 100k steps) or do both. Include the new reward curve in your report and mention how you improved over Task 2.2. Among other hyperparameters, you may attempt to change the value of β (see Fig. 2) to improve performance. You may also experiment with any other tricks and techniques that you think will improve performance.

Task 2.4 (10 points) Compare the reward curve obtained in Task 2.3 with the reward curve from Task 1.4. Which version performs better? Provide your reasoning for the difference in performance. Include this in your report.
In case you are not able to implement Tasks 1.4 and 2.3, perform the comparison between the reward curves obtained from Task 1.3 and Task 2.2.

References

[1] Nolan Bard, Jakob N Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, et al. The hanabi challenge: A new frontier for ai research. *Artificial Intelligence*, 280:103216, 2020.

[2] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[3] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.

[4] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International conference on machine learning*, pages 4295–4304. PMLR, 2018.

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[6] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative, multi-agent games. *arXiv preprint arXiv:2103.01955*, 2021.