Final Project Report

519021911325 刘畅

1. Introduction

In this project, we are required to implement two kinds of model-free reinforcement learning methods, which are value-based method and policy-based method, to solve environment of Atari and MuJuCo.

Value-based methods try to get the most suitable action-value function or state-value function to better value each action or state. And thus use those functions to choose the next action each time. Among them, Deep Q-learning (DQN) is representative for introducing deep neural network to replace Q-table in traditional Q-learning and achieves great performance.

In comparison, policy-based methods try to learn the policy directly, and updating policy function using policy gradient. Among them, Actor-Critic method is representative. And to solve the drawback of sampling inefficiency, off-policy methods like DDPG are introduced.

Considering the weak computing power of my laptop, even a easy model needs to be trained for a long time. So I choose simpler environments for training to obtain relatively ideal results. In this project, DQN is implemented on PongNoFrameSkip-v4, and PPO on Ant-v2.

2. Methods

2.1 DQN

DQN is the improved version of Q-Learning, which maintains a Q-table to record the value of state-value pair. But it has a severe problem. When facing complicated environment like Atari games, which have too many actions or states, the Q-table has to be huge, which is not sensible. Thus, DQN is introduced. In comparison with Q-learning, DQN makes two main improvements.

First, DQN uses experience replay to solve the correlation and non-static distribution problem. Experience replay uses a random sample of prior actions instead of the most recent action. Each step the agent generates a sample like (state, action, next_action, reward), and it will be stored and randomly chosen to train the network. Experience replay helps to accelerate the backup of rewards and remove the correlation of samples from the environment.

Second, DQN uses a Q-network to fit the value function and a target network to decide the action. And method of fixing target network is applied. Target network with older network parameters is used when estimating the Q-value for the next state in an experience. The target network updates every N steps. Such a target network fixed the policy when Q-network updates which leads to stability.

The detail of DQN algorithm is shown below in Figure. 1.

For the game of PongNoFrameSkip-v4, as each state is in form of a picture, thus convolutional neural network(CNN) needs to be applied, which contains convolutional layers, batch normalization layers and fully connected layers. It takes a transformed picture as input, and output the value of each action.

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Figure 1. DQN Algorithm

2.2 DDQN

Although DQN achieve quite a good outcome, there is a problem. Because it always choose the highest Q-value to update neutral network, it tends to overestimate Q-Values. To solve this, Double DQN was invented. Double DQN introduces another network, and it reduces Q-Value overestimation by splinting max operator of DQN into action selection using original network and action evaluation using another network.

$$\text{DQN: } Q(s_t, a_t) \leftrightarrow r_t + \max_a Q(s_{t+1}, a)$$

$$\text{Double DQN: } Q(s_t, a_t) \leftrightarrow r_t + Q'(s_{t+1}, \text{argmax}_a Q(s_{t+1}, a))$$

2.3 PPO

Problem with original policy gradient and Actor-Critic algorithm is sampling inefficiency. More specifically, when the policy θ is updated, we have to sample training data again because the distribution has changed. Thus, importance sampling is introduced to improve this inefficiency. It works by using the sample from old policy to train the new policy. In this way, we don't need to sample again, and just need to modify surrogate objective function:

$$J^{\theta_{old}}(\theta) = E_{(s_t, a_t) \sim \pi_{\theta_{old}}} \left[\frac{p_\theta(a_t|s_t)}{p_{\theta_{old}}(a_t|s_t)} A^{\theta_{old}}(s_t, a_t) \right]$$

To avoid the difference between current policy and old policy to be too huge, we need to set some restrictions. In Trust Region Policy Optimization (TRPO), it would be like:

$$KL(\theta, \theta_{old}) < \delta$$

But this is complicated to solve optimization problem with a constraint. So the first version of PPO uses KL divergence as a penalty instead, and removing the extra constraint.

$$J_{PPO}^{\theta_{old}}(\theta) = J^{\theta_{old}}(\theta) - \beta KL(\theta, \theta_{old})$$

However, it's hard to choose a single value of the coefficient β, so further improvement is made, forming PPO2. PPO2 proposed a clipped surrogate objective shown as follows. ε is a constant normally equals 0.2. The clip function ensures that the new policy distribution is not too far away from the old one.

$$J_{\mathrm{PPO2}}^{\theta_{old}}(\theta) = \sum_{(s_t,a_t)} \min\left(\frac{p_\theta(a_t|s_t)}{p_{\theta_{old}}(a_t|s_t)} A^{\theta_{old}}(s_t,a_t), clip\left(\frac{p_\theta(a_t|s_t)}{p_{\theta_{old}}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon\right) A^{\theta_{old}}(s_t,a_t)\right)$$

When the advantage value is positive, the new policy would be more likely to take this action, but the upper bound of clip function will limit the increase. On the other hand, when the advantage value is negative, the new policy would be less likely to take this action. The lower bound of the clip function will also limit the decrease. In this way, the new policy won't be too far away from the old one.

The detail of PPO is shown below in Figure. 2.

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t,a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t,a_t))\right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Figure 2. PPO Algorithm

2.4 SAC

Although PPO algorithm is widely used, it is an on-policy algorithm, which has the problem of sample inefficiency and requires a huge amount of sampling to learn. DDPG is an off-policy algorithm, which is more sample efficient than PPO, but it is sensitive to its hyper-parameters and has poor convergence effect. The SAC algorithm is also an off-policy algorithm, it is developed for maximum entropy reinforcement learning. Compared with DDPG, SAC uses a random strategy, which has advantage over deterministic strategies.

Compared with traditional reinforcement learning which maximizes the expected sum of rewards, SAC maximizes the expected sum of rewards and entropy. There are two advantages for SAC. First, it is motivated to explore more extensively. Second, the policy can give multiple options of acting. If there are multiple actions equally good, the policy would give equal probability to those actions.

$$\pi^* = arg\max_\pi E_\pi(r(s_t,a_t) + \alpha H(\pi(\cdot|s_t)))$$

With entropy added to objective function, we could get soft Q-function:

$$Q_{soft}(s_t,a_t) = r(s_t,a_t) + \gamma E_{s_{t+1},a_{t+1}}[Q_{soft}(s_{t+1},a_{t+1}) - \alpha log(\pi(a_{t+1}|s_{t+1}))]$$

And policy update funcion, in which Z represents partition function:

$$\pi' = \arg\min_{\pi_k \in \Pi} D_{KL}(\pi_k(\cdot|s_t)||\frac{\exp(\frac{1}{\alpha}Q_{soft}^{\pi}(s_t,\cdot))}{Z_{soft}^{\pi}(s_t)})$$

According to the theorem, by soft-updating, which means the policy critic network is updated gradually, the sequence of Q would converge to the soft Q-value.

In each iteration, the SAC agent would update the parameters of Q-function, policy network, temperature and target network. The detail of SAC is shown in Figure 3.

Algorithm 1 Soft Actor-Critic

Input: θ_1, θ_2, ϕ ▷ Initial parameters
$\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$ ▷ Initialize target network weights
$\mathcal{D} \leftarrow \emptyset$ ▷ Initialize an empty replay pool
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ ▷ Sample action from the policy
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ ▷ Sample transition from the environment
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$ ▷ Store the transition in the replay pool
  end for
  for each gradient step **do**
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$ ▷ Update the Q-function parameters
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ ▷ Update policy weights
    $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ ▷ Adjust temperature
    $\bar{\theta}_i \leftarrow \tau\theta_i + (1-\tau)\bar{\theta}_i$ for $i \in \{1, 2\}$ ▷ Update target network weights
  end for
end for
Output: θ_1, θ_2, ϕ ▷ Optimized parameters

Figure 3. SAC Algorithm

3. Experiments

3.1 Atari: PongNoFrameSkip-v4

3.1.1 Environment

Atari contains many classic games. I choose the game of PongNoFrameSkip-v4 to apply DQN.

The observation of Atari environment is the screen of the game of size 210*160*3. And some preprocessing need to be done before training. First, as RGB value is in range [0, 255], I first convert it into [0, 1] by dividing 255. Then, to better match the input form of network, the input would be transformed to 3*210*160, making input dimension 3.

Some improvements are applied during training. If the environment is generated directly by **env=gym.make('PongNoFrameSkip-v4')**, I noticed that the training would be extremely slow (about 160 episodes in 8 hours) and achieve poor performance. So methods of NoopResetEnv, MaxAndSkipEnv and TimeLimit are added. NoopResetEnv skip some of the initial frames, making the initial state more random. MaxAndSkipEnv uses the same action in neighboring states, and returning once in every few frames to accelerate training. TimeLimit would limit the maximum actions made, returning done=True when achieving the limit. By applying these methods, not only the performance is much better, training time is greatly reduced (134 episodes in one hour).

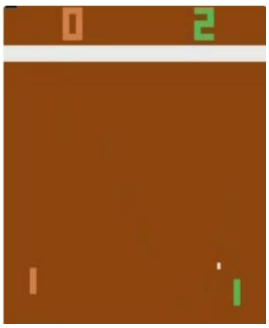

Figure 4. PongNoFrameSkip-v4

3.1.2 Implementation

The convolutional network contains eight layers, three convolutional layers, three batch normalization layers and two fully-connected layers. Detailed setting is shown below:

Layer	Input dim	Output dim	Kernel Size	Stride	Activation
Conv2d	in_channels	32	8*8	4*4	relu
BatchNorm2d	32	32	/	/	/
Conv2d	32	64	4*4	2*2	relu
BatchNorm2d	64	64	/	/	/
Conv2d	64	64	3*3	1*1	relu
BatchNorm2d	64	64	/	/	/
Linear	14*11*64	512	/	/	relu
Linear	512	n_actions	/	/	none

Table 1. QNet Structure

Other hyper-parameters are set as follows:

Hyper-parameters	Value
batch size	32
γ	0.99
ε_bound	[0.02, 1]
ε_decay	0.995
target update stride	1000
policy update stride	2
learning rate	0.001
n_episode	1000
memory size	100000

Table 2. DQN hyper parameters

3.1.3 Result

I trained DQN and DDQN on PongNoFrameSkip-v4 provided in gym for 1000 episodes.

For DQN, the result is shown in Figure.5. It could be seen that training is quite fast, after about 200 episodes, the reward significantly improved. It means the DQN algorithm quickly learns a good policy. And it also shows that the improved environment is much better. In comparison, after 200 episodes in the original environment, the reward is still around -20. And to evaluate the trained model, I use the model to play a rendered game to see what's the outcome. For the first few points, my agent would likely to lose, but after about 3 points, it seems to find a trick, and keeps winning using the same strategy. The average score is around 18.

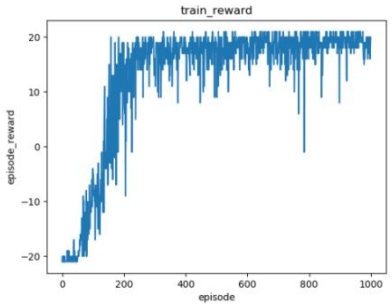

Figure 5. DQN result on Pong

For DQN, the result is shown in Figure.6. Similar to DQN, the converge comes quite quickly, after just about 200 episodes. But compared with DQN, DDQN has a more stable outcome. In the test, the action is also quite similar to DQN, with average score around 18.

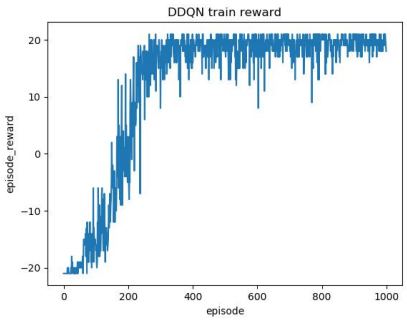

Figure 6. DDQN result on Pong

3.2 Mujoco: Ant-v2

3.2.1 Environment

MuJoco is a physics engine simulating rigid bodies with contact. And it can also be used by importing gym, with mujoco-py already installed. Compared with Atari Games, the observation and action space is one dimensional vector. So there is no need for convolutional neural networks, fully-connected layers would be fine.

3.2.2 Implementation

3.2.2.1 PPO

PPO is implemented using Actor-Critic structure, the Actor gives the policy, the Critic gives value of each state. The structure of Actor and Critic is shown in table 3. The Actor output two parameters for normal distribution (μ, σ), which stands for expectation value and standard error.

Actor			Critic		
Name	**Input**	**Output**	**Name**	**Input**	**Output**
fc1	state_size	64	fc1	state_size	64
fc2	64	64	fc2	64	64
mu	64	action_size	fc3	64	1
sigma	64	action_size			

Table 3. Actor-Critic Structure for PPO

And the hyper-parameters for PPO are set as follows:

Hyper-parameters	Value
batch size	64
γ	0.99
clip ε	0.2
λ	0.95
update stride	2048
actor learning rate	0.0003
critic learning rate	0.0003

Table 4. PPO hyper parameters

3.2.2.2 SAC

The implementation of SAC follows a similar structure. The detail is shown in table 5. But the difference is that Critic values the state-action pairs, instead of just states.

Actor			Critic		
Name	**Input**	**Output**	**Name**	**Input**	**Output**
fc1	state_size	256	fc1	state+action	256
fc2	256	256	fc2	256	256
mu	256	action_size	fc3	256	1
sigma	256	action_size			

Table 5. Actor-Critic Structure for SAC

And the hyper-parameters for S are set as follows:

Hyper-parameters	Value
batch size	256
γ	0.99
τ	0.005
α	0.05
actor learning rate	0.0003
critic learning rate	0.0003

Table 6. SAC hyper parameters

3.2.3 Results

I trained PPO and SAC on a Ant-v2 from MuJoCo environment.

For PPO, the Actor and Critic are trained for 3000 times, for about 10000 episodes. But the training is much quicker. The result is shown in Figure 7. As we can see, the result is not that steady, and generally the performance is improving quite slowly. To gain a good performance takes loads of training steps, and the best reward is around 2500.

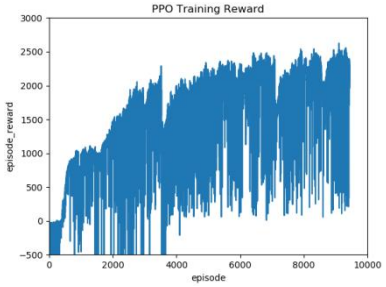

Figure 7. PPO result on Ant-v2

For SAC, the Actor and Critic are trained for 2500 episodes with the total interaction of 2000000 times. Compared with PPO, the training time is bit slower due to more training steps. The result is shown in Figure 8. The result is much more steady than PPO, and could achieve good performance quickly. The best reward is around 4000.

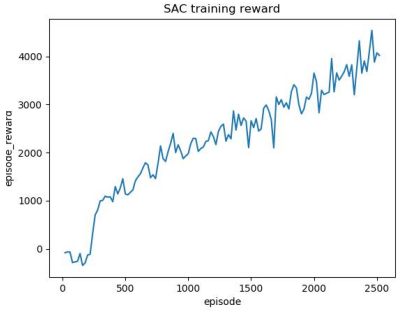

Figure 8. SAC result on Ant-v2

4. Conclusion

In this project, I implement both value-based and policy-based methods on Atari and MuJoCo environments. In specific, DQN and DDQN on Pong-v4, PPO and DDPG on Ant-v2.

For value-based RL algorithm, DQN and DDQN does perform well in Atari game, but the training time is quite costly, 1000 episodes would take hours. It also suffers from sample unpredictability. If the agent couldn't get a positive reward, which means hitting the ball and win a point in the game, it wouldn't achieve a good performance.

For policy-based RL algorithms, PPO is faster but not having good performance, while DDPG is slower but having great performance.

In general, the environment and parameters could greatly influence the performance of algorithms, it may not be fair to judge which algorithm is better merely by its performance in a specific environment. So, the selection of algorithm is a key factor.

In fact, the reason why I choose PPO and SAC is the benchmark from this site. So if I choose DDPG, although with baseline of last assignment, I would still waste long time trying to improve the performance of a method that could not gain good performance, which is not sensible.

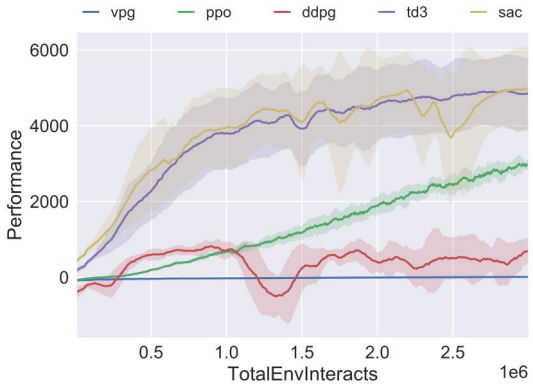

Figure 9. benchmark result on Ant-v2 using pytorch