**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.

1. **Methods**
   1. **DQN**

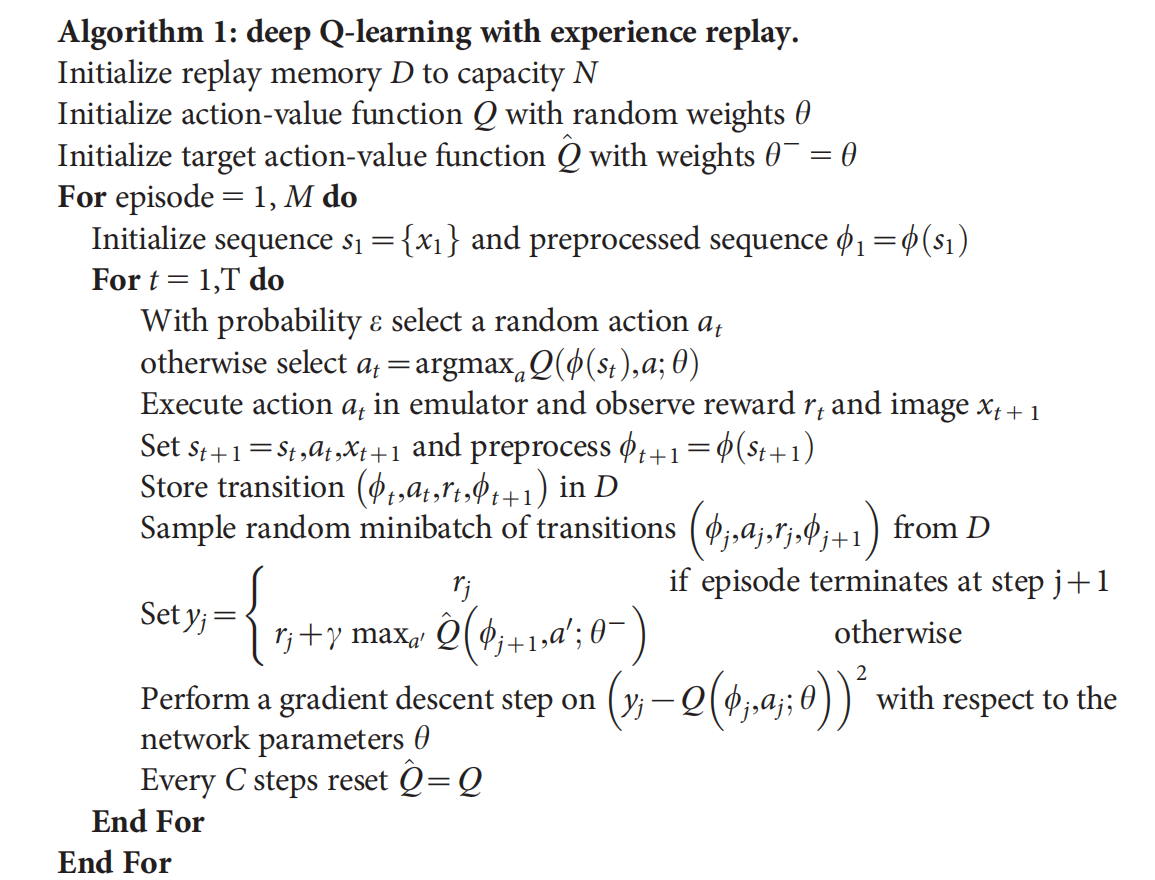
DQN is the improved version of Q-Learning, which maintains a Q-table to record the value of state-value pair. But it has a severe problem. When facing complicated environment like Atari games, which have too many actions or states, the Q-table has to be huge, which is not sensible. Thus, DQN is introduced.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



**Figure 1. DQN Algorithm**

* 1. **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.

* 1. **PPO**

Problem with original policy gradient and Actor-Critic algorithm is sampling inefficiency. More specifically, when the policy 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:

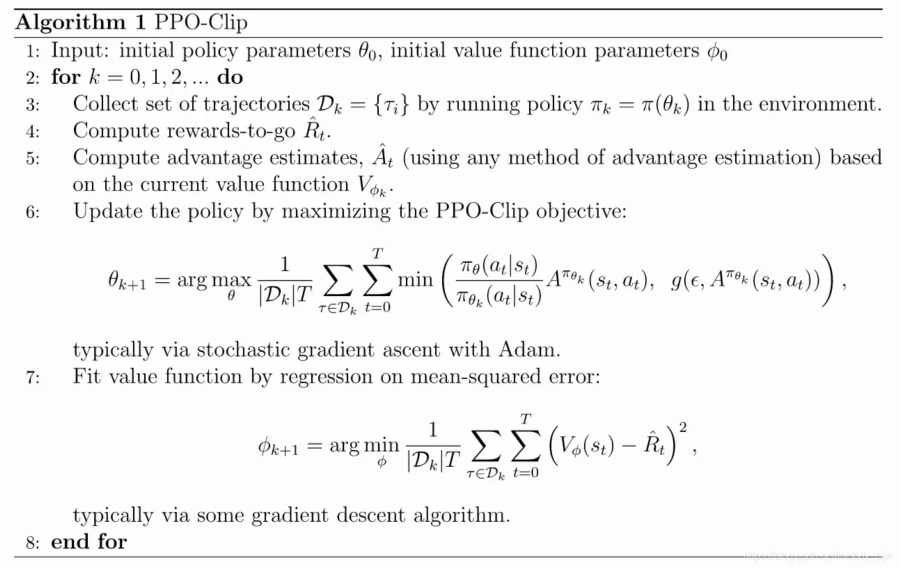
To avoid the difference between current policy and old policy to be too huge, we need to set some restrictions. In Trust Region Policy Optimization (TRPO), it would be like:

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.

However, it’s hard to choose a single value of the coefficient , so further improvement is made, forming PPO2. PPO2 proposed a clipped surrogate objective shown as follows. 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.

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.

  
**Figure 2. PPO Algorithm**

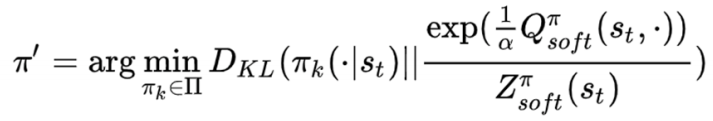
* 1. **SAC**

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.

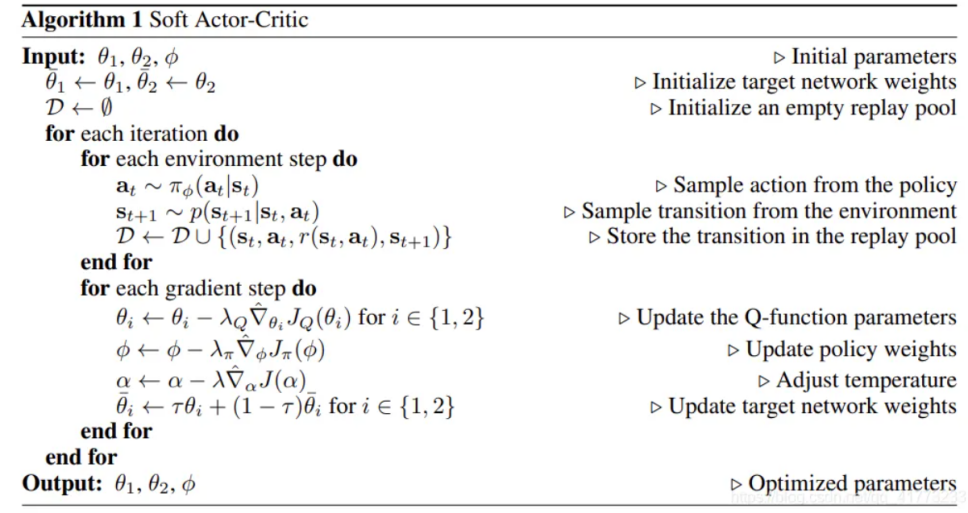
With entropy added to objective function, we could get soft Q-function:

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



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.



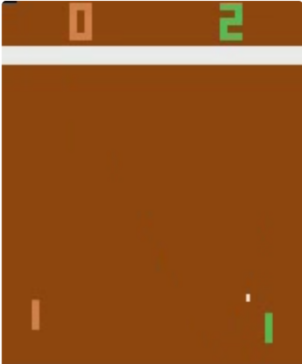
**Figure 3. SAC Algorithm**

1. **Experiments**
   1. **Atari: PongNoFrameSkip-v4**
      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**

* + 1. **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 |

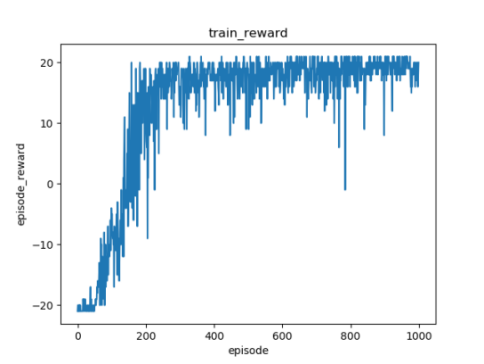
**Table 2. DQN hyper parameters**

For DDQN, the only difference is that another QNet is added to implement soft updating.

* + 1. **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 17.42.



**Figure 5. DQN result on Pong**

* 1. **Mujoco: Ant-v2**
     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.

* + 1. **Implementation**
       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 , 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 |
|  | 0.05 |
| actor learning rate | 0.0003 |
| critic learning rate | 0.0003 |

**Table 4. PPO hyper parameters**

* + - 1. **SAC**

The implementation of SAC follows a similar structure. The detail of Actor and Critic is shown in table 5. But the difference is that Critic values the pair of (state, action), instead of state in PPO.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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:

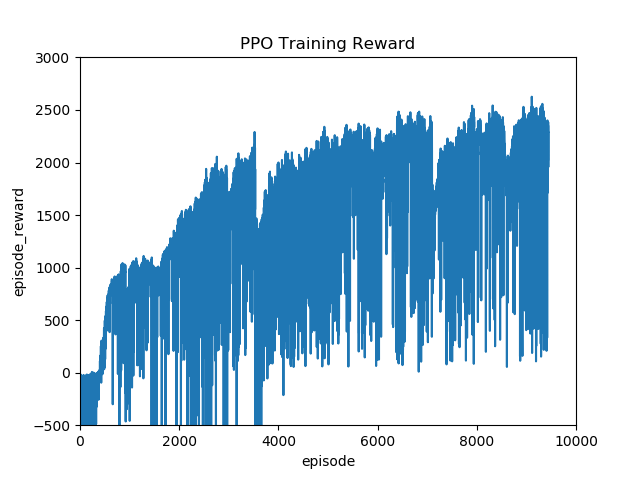
|  |  |
| --- | --- |
| **Hyper-parameters** | **Value** |
| batch size | 256 |
| γ | 0.99 |
|  | 0.05 |
| λ | 0.95 |
| update stride | 2048 |
| actor learning rate | 0.0003 |
| critic learning rate | 0.0003 |

**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 15000 times, and the gap between each training is 2048 steps. although the episode number is much larger than DQN (about 10000 episodes), but the training is much quicker. The result is shown in Figure 6. 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 6. PPO result on Ant-v2**

1. **Conclusion**

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

For value-based RL algorithm, DQN 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](https://spinningup.openai.com/en/latest/spinningup/bench.html" \l "ant-pytorch-versions). So if I choose DDPG, I would waste long time trying to improvement the performance of a method that could not gain good performance, which is not sensible.



**Figure 7. benchmark result on Ant-v2 using pytorch**