Experiments of Advanced Model-Free Algorithms on Value-Based and Policy-Based Reinforcement Learning

CS489 Reinforcement Learning 2021 Spring

Chunyu Xue
Student ID {518021910698}
Dicardo@sjtu.edu.cn

Department of Computer Science,
Shanghai Jiao Tong University, Shanghai, China

Abstract. In area of model-free control Reinforcement Learning, value-based and policy-based methods have commonly classified the problems based on the continuity of action spaces. Problems with discrete action spaces such as part of video games in Atari 2600 are well solved by value-based methods like DQN and its advanced versions, while the other problems with continuous action space such as humanoid action simulation can be satisfactorily solved by policy-based methods like AC and its series algorithms. In this paper, we implement Dueling-DQN and its improved version Dueling-Double-Deep Q-Network (Dueling DDQN) algorithm as the representation of our advanced value-based methods as well as making detailed comparison and analysis on the performance on DQN series methods. As for continuous action space, we choose to implement Twin Delayed Deep Deterministic policy gradient (TD3) algorithm and compare its performance with its original version DDPG. All experiments are operated on various environments provided with Gym Atari and MuJoCo Simulator.

Keywords: Model-free control, Dueling-DDQN, TD3, Gym Atari, MuJoCo Simulator

1 Introduction

In Reinforcement Learning problems with discrete actions, value-based algorithm Deep Q-Network [1] has laid significant foundation for the performance improvement in model-free control. Series of algorithms based on DQN have been proposed, including Double DQN [2], which uses current parameters θ_t to choose optimized action for eliminating over-estimation, and Dueling DQN [3], which combines state value and action advantage together to represent the output Q value from Q network. Detailed analysis of DQN series algorithms would be given in section 3. In addition to the algorithms mentioned above, a Dueling-Double-Deep Q-Network algorithm has been proposed in 2020 IEEE Access [4] to improve the Magnetic Levitation Ball System, which integrating the core of Double DQN and Dueling DQN together, attempting to get higher performance.

In this paper, we will refer to the network structure in paper [4] to implement our Dueling-Double-Deep Q-Network algorithm, make reasonable modifications to better fit the local host environment, and do sufficient experiments on various Gym Atari environments with the performance comparison to Dueling DQN, which used to be the best partner in DQN series algorithms and obtained the best paper reward in ICML 2016 for its amazing performance.

As for continuous action space RL problems, policy-based algorithm Actor-Critic [5] would also lead to overestimated value estimates and sub-optimal policies proved in [6]. Therefore, AC method has also became the foundation for exploring more significant improvement on performance in continuous action space issues. Based on AC, recent researches have proposed Asynchronous Advantage Actor-critic (A3C) [7], Deep-Deterministic Policy Gradient (DDPG) [8] and some other efficient algorithms, each of these advanced policy-based methods has all achieved a considerable performance on continuous action situations. Detailed analysis of AC series algorithms would be given in section 4.

In this paper, we would also choose Twin Delayed Deep Deterministic policy gradient algorithm (TD3) [6], one of these advanced AC algorithms, to be our experiment subject. We first implement our TD3 algorithm based on paper [6], then operate performance tests on Gym MuJoCo environments. With the performance comparison with DDPG algorithm, which is the original base of TD3, we would analyse the advantages of TD3 method.

2 Environment Settings

In order to Operate experiments on our implemented algorithms as well as provide benchmarks in evaluating the performance of the given methods, Gym Atari and MuJoCo simulators are used to construct our experimental environment. In section 2.1, we introduce the platform and toolkit versions in our experiments, and in section 2.2 and 2.3, we would formally introduce the detailed configurations and implementations of the two environmental schemes mentioned above.

2.1 Experiment Platform and Toolkit Versions

- Value-based Experiment Platform: *MatPool: NVIDIA GeForce RTX* 2080
- Policy-based Experiment Platform: *MacOS Big Sur* 11.2.3 16*GB*
- *Python*: 3.7
- *PyTorch*: 1.2.0
- *Gym*: 0.18.3 (*Pyglet* 1.5.15, *Numpy* 1.20.3, *Scipy* 1.6.3, *Pillow* 8.2.0)
- *Opencv-python*: 4.5.2.52
- *Atari_py*: 0.2.9
- *Mujoco_py*: 2.0.3.13 (*gcc*@9)

2.2 Gym Atari Installation

Gym Atari[9] is a simulation environment of video games, which provides more than 100 original games and different versions of them in Atari-2600 platform. Considering the properties of operated actions in these games, Atari is one of the most ideal experimental environments for value-based model-free control RL, which aims to solve problems with discrete action space.

In order to operate Atari games simulation in Gym, we need to use pip tools to install **atari-py** package in our virtual environment. However, we have received an exception when we try to import and run the Atari game environment, informing us that "*ROM is missing for the chosen video game*". By following the tutorial provided in official document [10], we first download the rar package and unrar to the /.atari directory. After unzip the ROMS.zip file, we use "*python -m atari_py.import_roms ROMS*" command to import the downloaded ROMS into the Atari settings. So far, the installation of Gym Atari environment has been completed.

2.3 MuJoCo Simulator Installation

MuJoCo stands for Multi-Joint dynamics with Contact. It is being developed by Emo Todorov for Roboti LLC. Initially it was used at the Movement Control Laboratory, University of Washington, and has now been adopted by a wide community of researchers and developers [11]. In problems with continuous action space, MuJoCo has became one of the most popular simulators supported by OpenAI Gym.

Unlike Gym Atari toolkit, we need to buy the qualified license for the usage of MuJoCo simulator. In our experiments, we choose to obtain personal license for 30-days trial for free.

- First, we should get our local computer id by executing the executable file *./getid_osx* in the local host. We need to download it from the official website[1] and change the permission of it.

- Then, we should create */.mujoco* directory, downloading the MuJoCo file by executing "*wget*" command[2], and move it into */.mujoco* directory after unzip.

- After we receive the mail sent from Roboti Org, we should download the *mjkey.txt* in attachment and move it into */.mujoco* directory.

- Finally, we use the command "*git clone https://github.com/openai/mujoco-py.git*" to download the main files of MuJoCO, get into the directory and execute "*python setup.py install*" to begin MuJoCo installation.

[1] MuJoCo License: https://www.roboti.us/license.html
[2] Wget Address: https://www.roboti.us/download/mujoco200_macos.zip

Note that when we import *mujoco_py* in our code, an error occurred and informs us that "*Can't find GCC compiler*". To solve this error, we look into *./mujoco-py/mujoco_py/builder.py* file and search for the information of GCC version supported, the result can be found in Fig 1. From the information above, we know that MuJoCo of this version only supports GCC with the version before 9. Therefore, we use "*brew install gcc@9*" to reinstall GCC compiler and successfully import the package. So far, the installation of MuJoCo Simulator has been completed.

```
309     def _build_impl(self):
310         if not os.environ.get('CC'):
311             # Known-working versions of GCC on mac (prefer latest one)
312             c_compilers = [
313 
314                 '/usr/local/bin/gcc-9',
315                 '/usr/local/bin/gcc-8',
316                 '/usr/local/bin/gcc-7',
317                 '/usr/local/bin/gcc-6',
318                 '/opt/local/bin/gcc-mp-9',
319                 '/opt/local/bin/gcc-mp-8',
320                 '/opt/local/bin/gcc-mp-7',
321                 '/opt/local/bin/gcc-mp-6',
322             ]
```

Fig. 1. Information of GCC Version Supported

3 Value-based Method: Dueling-Double-Deep Q-Network

In value-based reinforcement learning methods such as deep Q-learning, function approximation errors are known to lead to overestimated value estimates and sub-optimal policies [6]. Therefore, series of advanced DQN versions have been proposed in recent years in order to overcome the shortcomings of the original DQN, including the over-estimation mentioned above. According to the tutorial introduction given by David Silver in ICML 2016, these improvements could be classified into three directions: Double DQN, Prioritised Replay and Dueling Network.

- **Double DQN**: Remove upward bias caused by $\max_a Q(s, a, \boldsymbol{w})$. That is, current Q-network $\boldsymbol{w}$ is used to select actions, while older Q-network $\boldsymbol{w}^-$ is used to evaluate actions.

$$I = (r + \gamma Q(s', \underset{a'}{argmax}\, Q(s', a', \boldsymbol{w}), \boldsymbol{w}^-) - Q(s, a, \boldsymbol{w}))^2 \tag{1}$$

- **Prioritised replay**: Weight experience according to surprise. That is, store experience in priority queue according to DQN error:

$$|r + \gamma \max_{a'} Q(s', a', \boldsymbol{w}^-) - Q(s, a, \boldsymbol{w})| \tag{2}$$

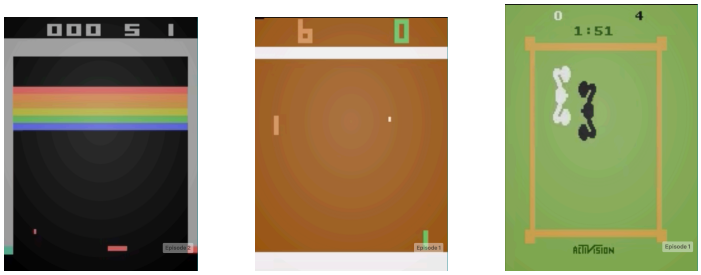
- **Dueling network**: Split Q-network into two channels: action-independent value function $V(s, v)$ and action-dependent advantage function $A(s, a, \boldsymbol{w})$

$$Q(s, a) = V(s, v) + A(s, a, \boldsymbol{w}) \tag{3}$$

Based on the three improved methods above, we would introduce the Dueling-Double-Deep Q-Network algorithm [4], which combines the core ideas of Double DQN and Dueling DQN together tog get better performance. Detailed descriptions about the Dueling DDQN would be given in section 3.2. Before that, we first introduce the experimental environment in section 3.1, which will analyse the configurations and interaction methods of Atari 2600 games in Gym. In section 3.3, we given the experimental results and performance comparison with Dueling DQN, the best representative of DQN series algorithm.

3.1 Environment: Gym Atari

In this part of the experiments, we use *BreakoutNoFrameskip-v4*, *PongNoFrameskip-v4* and *Boxing-NoFrame skip-v4* shown in Fig 3.1 to be our test environment for value-based algorithms. Therefore, We would like to introduce the basic ideas on the implementation as well as the usage of Gym Atari environments [12].



(a) Breakout-v4 (b) Pong-v4 (c) Boxing-v4

Fig. 2. Experimental Gym Atari Environments

Wrapper Class Gym provides a Wrapper class for better adaption to Atari 2600 games, which is inherited from Env base class. In this way, we can customize the configurations and interaction method when using different Atari environments by overwriting part of the methods in Wrapper class.

Reset Rules and Episode Termination The whole Atari game environment is a deterministic environment. An agent may perform well in the deterministic environment, but it may be highly sensitive to a small disturbance, so randomness is often added in the setting. In the games we choose, a player may have more than one life. Taking the termination of the game as the termination of training epic may not help the agent learn the importance of losing life. In the program, *was_ real_ Done* sets the mark of whether the game is really over, and every time you lose your life is taken as the mark of *done*.

Reward and Observation Clip According to the positive and negative nature of reward, it is divided into $\{+1, 0, -1\}$ to prevent the influence of great difference of reward in different environments on the algorithm. The original image is converted from 210x160 to 84x84, and the color image is converted to gray scale image, which would be store into replay buffer after transformation.

Frame Stacking Using only one frame of image as observation may not have enough information for the agent. Frame stack technology combines the image information of the first k frames into observation to avoid the possibility of the environment falling into some observable problems. In our experiments, we take the first $k = 4$ frames into the observation of our agent, seen in Fig 3.

```
42      for i in range(cur_n_episodes):
43          # Image
44          cur_img = env.reset()
45          # Current episode reward
46          cur_episode_reward = 0
47          # Whether the target is done
48          cur_is_done = False
49          # State buffer
50          state_buffer = []
51          for j in range(5):
52              state_buffer.append(cur_img)
53          cur_state = state_buffer[1:5]
```

Fig. 3. Implementation of Frame Stacking

3.2 Algorithm Description

According to paper [4], Since Double DQN and Dueling DQN have been focus on different ideas on the modification of DQN, the idea that we combine them together is rather reasonable. In our implementation of Dueling-Double-Deep Q-Network, we build Double DQN part Dueling DQN part separately on the basis of DQN. In order to spped the training progress in our local host, we make som modifications including simplify the network structures in our Dueling DDQN algorithm compared to the original method described in the paper.

Based on the ideas above, we would provide the two main improvements of our algorithm from DQN prototype and analysis the advantages on each of them.

Improvement 1: Dueling-Deep Q-Network In traditional DQN, the output of its network is the Q value with the size equals to the size of action space. The last layer of DQN network structure is a fully connect layer, which is exactly the point Dueling DQN would modify. In Dueling DQN, there are two streams in the last part of the network to separately estimate (scalar) state value and the advantages for each action, where the advantage function is defined as equation (4) shown.

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{4}$$

In the equation above, the value function $V^{\pi}(s)$ indicates whether the current state is good or not, while the Q function $Q^{\pi}(s,a)$ represents the deterministic action value in current state. Therefore, the advantage function $A^{\pi}(s,a)$ indicates the good or bad degree of each action in the chosen state. The network structure of Dueling DQN is shown in Fig 4.

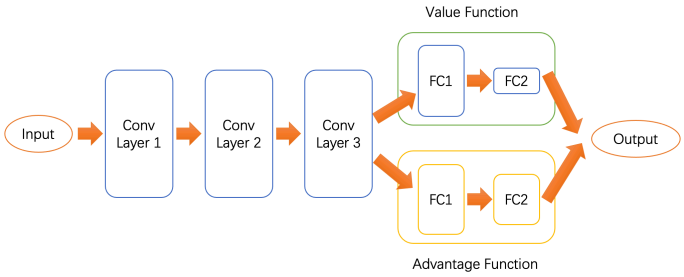


Fig. 4. Network Structure of Dueling DQN

Since the $V^{\pi}(s)$ is a scalar, this value could be left-bias or right-bias in the network, which has no direct effect on the output Q value described in [3]. In this case, a constant difference in the values of A and V is common since Dueling DQN is a end-to-end network, which is not good for our prediction and approximation if frequently fluctuate. Therefore, a solution is proposed in [3] that we could fix a bias with equation (5) shown as follows:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)) \tag{5}$$

By calculating Q value in the last layer of Dueling DQN network, we could establish a unified standard for the evaluation of all states and actions.

Improvement 2: Double-Deep Q-Network The idea of DDQN is proposed by [2] to solve the over-estimation problem in DQN. The only difference between DQN and DDQN is the different Q value causing by the choice of action. In DQN, we calculate the Q value as equation (6) shows:

$$Q(s_t,a_t) \leftarrow r_t + \max_a Q(s_{t+1},a) \tag{6}$$

However, in Double DQN, we use the action predicted by the evaluate net to determine the output Q value by target net, as equation (7) shows:

$$Q(s_t,a_t) \leftarrow r_t + Q'(s_{t+1}, \underset{a}{argmax}\, Q(s_{t+1},a)) \tag{7}$$

That is, we choose our action according to a Q table, and then measure the value of $Q(s',a')$ with another Q table or network parameters. Therefore, we could decouple the process of action choosing and Q value calculating, which could significantly eliminate the over-estimation issue.

Based on the two improvements above, we could construct the network in our Dueling DDQN algorithm by adapting the Dueling DQN into our *QNet* class, and Double DQN into our train logic in *learn()* function. Pseudo code of our algorithm could be referred to both Dueling DQN and Double DQN, omitted here for space and typesetting reasons.

3.3 Experiments Analysis

Settings We implement our Dueling-Double-Deep Q-Network and one of its basis versions Dueling-Deep Q-Network, which is once proved to be the best in DQN series, to operate tests on Gym Atari environments and make sufficient analysis based on the comparison of experimental results.

We totally trained our models $10M$ steps for each on *MatPool: NVIDIA GeForce RTX2080* platform, with the learning rate equals to 0.0000625 and batch size equals to 32. We evaluate our models every 20 rounds, thus there are about 600 evaluation episodes in total.

As for environmental settings, *BreakoutNoFrameskip-v4*, *PongNoFrameskip-v4* and *BoxingNoFrameskip-v4* are all supported for our program. We use the first *Breakout* as the prototype to analysis the results.

The two algorithms above are implemented in independent project directory, and a command "*python atari_playground.py –env_name [env name]*" could be used to run our program in any of the two directories. More information could be found in *README.md*.

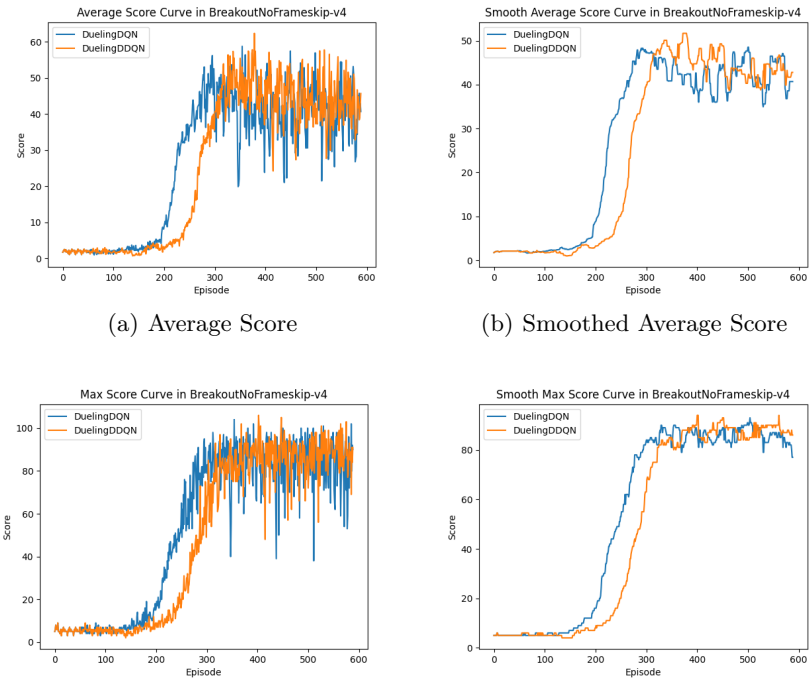
Average and Max Score We first analyse the average and max scores achieved by each method, where the results are shown in Fig 5. Note that we provided a filtered version by a median filter with the kernel size equals to 13. We would give the phenomenons we discovered in the results:

– Dueling DDQN has a **slower convergence speed** than Dueling DQN;

– Dueling DDQN has a **more stable performance** than Dueling DQN;

– The **highest score is basically similar** for the two algorithms;

The first phenomenon is easy to understand, which is caused by the decoupling on the action choosing and calculating Q value described in section 3.3. The detailed reason would be given in the later analysis of loss. And the higher variance on Duelnig DQN is also caused by the higher dependency on the action choosing and Q value calculation, which will implicitly impacts on each other. And since there is no additional tuning for Dueling DDQN, the max scores they achieved have no significant difference.



(a) Average Score (b) Smoothed Average Score

(c) Max Score (d) Smoothed Max Score

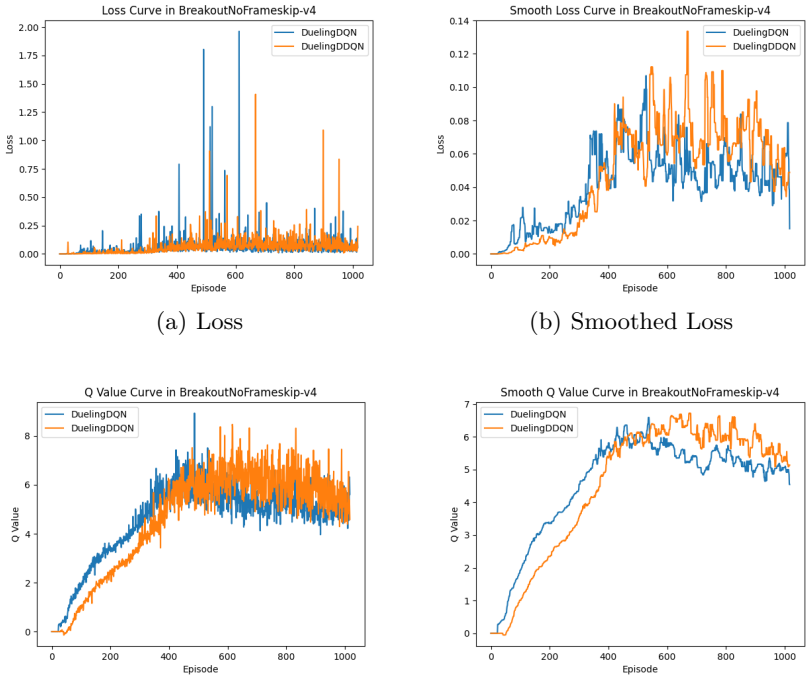
Fig. 5. Average and Max Score in BoxingNoFrameskip-v4

Loss and Q Value To further analyse the performance difference between the two methods, we give the loss and Q value comparison in Fig 6, and a smoothed version by median filter with kernel size equals to 13 is also provided. We also give the phenomenons we discovered as follows:

- Dueling DDQN has a **higher train loss** than Dueling DQN;
- Dueling DDQN has a **slower convergence speed** and a **higher variance** on Q value than Dueling DQN, but a **higher Q value performance generally**;

After detailed analysis, we have found that the higher loss is caused by the independency of action choosing and calculating Q value. Since the predicted best next action a_{next} in evaluate net could be different in the one in target net, when we calculate q_{target} in the target net with the input a_{next}, q_{target} **could be smaller than the one we obtain by using directly the optimized action of target net as input**. Notice that the evaluate net is always newer than the target net, so q_{eval} **should be larger than** q_{target}. Therefore, when we calculate mse loss between q_{eval} and q_{target} in Dueling DDQN, it could become larger than not using the Double Q structure.

As for difference of the Q value, which is similar to the analysis in average score above, would become slower updated and better performance in our Dueling DDQN. However, unlike the variance of average score, The higher variance of Q value in Dueling DDQN is caused by the different input for the target net, which will not directly influence our model performance.



(a) Loss (b) Smoothed Loss

(c) Q Value (d) Smoothed Q Value

Fig. 6. Loss and Q Value in BoxingNoFrameskip-v4

4 Policy-based Method: Twin Delayed Deep Deterministic Policy Gradient

As mentioned in section 1, over-estimation could also exist in Actor-Critic setting, which leads to the unremitting search for improved methods. Similar to DQN situations, advanced AC algorithms could also be classified into three directions: Deep Deterministic Policy Gradient (DPG) and its improved version TD3, Proximal Policy Optimization (PPO) and its improved version Distributed Proximal Policy

Optimization (DPPO), and Soft Actor-Critic (SAC) algorithm. Here we only discuss the main advantages/disadvantages of these algorithms [13] except for our star TD3 algorithm, which will be discussed in detail in section 4.2.

- **DDPG/TD3**: DDPG converges quickly on small tasks and works well. On continuous control tasks, TD3, which operates simple changes from DDPG could get much better performance.

- **PPO/DPPO**: PPO method is not necessarily the best method, but it is generally the most stable method. DPPO uses orthogonal initialization and the maximum batch size allowed by hardware conditions to train, which is very effective. However, the problem of PPO is the low sampling efficiency, which requires a large number of data to be used.

- **SAC**: SAC needs learned temperature to get the best results as well as grid search for reward scaling ratio, which is very unstable in the process of model train.

Based on the performance analysis provided above, it's reasonable for us to choose TD3 algorithm as our experimental target to get scores as better as possible. In section 4.1, we introduce the MuJoCo environment by analysing its implementation and interaction method. Then, in section 4.2, we would formally give introduction on TD3 algorithm, analyse its three main improvements from DDPG and some possible further improvements on it. Finally, in section 4.3, we show the results of experiments on TD3 and DDPG and further analyse the difference between their performances.

4.1 Environment: MuJoCo

In this part of the experiments, we use classic MuJoCo environments include *Hopper-v2*, *Humanoid-v2*, *HalfCheetah-v2* and *Ant-v2* to be our test environment for policy-based algorithms. Therefore, we first give a brief introduction on the implementation and usage of MuJoCo Simulator.

MuJoCo[3] is a physics engine for detailed, efficient rigid body simulations with contacts. mujoco-py allows using MuJoCo from Python3. XML configurations file is used in MuJoCo for users to specify or customize the details of physical simulation, including physical models, velocity change and so on. There are three main structures in XML configurations:

- **<asset>**: import STL file with < mesh > tag;
- **<world body>**: all simulator components are defined with < body > tag, including lights, floors and your robot;
- **<acutator>**: defines the joints that can perform motion. For example, the last joint near the tool coordinate is joint0, and so on.

In the python3 coding environment provided by MuJoCo, we could use the *env* defined by gym to interact with the simulator. Series of original member functions in Env Class like *reset()* and *step()* are used to obtain simulator current state or take the next action. The implementation of interaction methods is shown in Fig 7.

```
24      eval_env = gym.make(env_name)
25      eval_env.seed(seed + 100)
26
27      avg_reward = 0.
28      for _ in range(eval_episodes):
29          state, done = eval_env.reset(), False
30          while not done:
31              action = policy.select_action(np.array(state))
32              state, reward, done, _ = eval_env.step(action)
33              avg_reward += reward
34
35      avg_reward /= eval_episodes
```

Fig. 7. Interaction methods with MuJoCo in Python

[3] MuJoCo GitHub: https://github.com/openai/mujoco-py

4.2 Algorithm Description

TD3 algorithm is an improved version of current DDPG. To minimize the effects from over estimated on both actor and critic network, a novel mechanisms is proposed by [6] in ICML 2018, which builds on Double Q-learning by taking the minimum value between a pair of critic nets to limit over-estimation. In order to get better algorithm performance, we are suggested delaying policy updates to reduce per-update error according to the experiments in the paper above. Besides, we are recommended to learn for two Q nets and use the one with smaller Q value to construct the target network. The network structure of the TD3 above is shown in Fig 8.

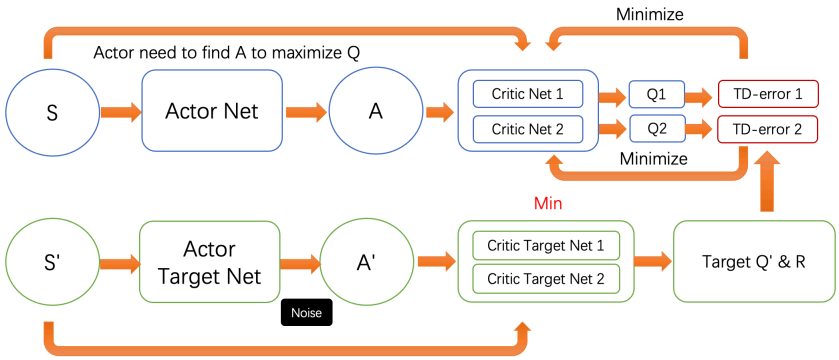


Fig. 8. Network Structure of TD3 Algorithm

Based on the suggestions and network structures above, we would give the three main improvements on TD3 algorithm provided by paper [6].

Improvement 1: Clipped Double Q-Learning for Actor-Critic In Double DQN [2], the authors propose using the target network as one of the value estimates, and obtain a policy by greedy maximization of the current value network rather than the target network. In an actor-critic setting, an analogous update uses the current policy rather than the target policy in the learning target:

$$y = r + \gamma Q_{\theta'}(s', \pi_{\varphi}(s')) \tag{8}$$

However, it's found that the evaluate and target networks are too similar to make an independent estimation and offer little improvement in target update with the slow-updated policy in actor-critic. Therefore, a pair of actors $(\pi_{\varphi_1}, \pi_{\varphi_2})$ and critics $(Q_{\theta_1}, Q_{\theta_2})$ are used to form a double Q-learning structure. To eliminate the over-estimated problem caused by not entire independence between critics, the authors in [6] proposed to simply upper-bound the chosen Q value for each actor by taking the minimum between the two estimates, to give the target update of our Clipped Double Q-learning algorithm:

$$y_1 = r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \pi_{\varphi_1}(s')) \tag{9}$$

According to the paper, the value target in Clipped Double Q-Learning can't introduce any additional over-estimation compared to using the standard Q-learning target. Although it may induce an under-estimation bias, this is far more preferable to over-estimation bias since under-estimation bias will not be explicitly propagated through the policy update.

Improvement 2: Delayed Policy Updates From the experimental results in the paper, we are informed that target networks can be used to reduce the error over multiple updates, and when the policy is trained with the current value estimate, the use of fast-updating target networks results in highly divergent behavior. Therefore, we could know that the policy network should be updated at a lower frequency than the value network, to first minimize error before introducing a policy update.

Based on the experimental facts above, we are recommended that delaying policy updates until the value error is as small as possible, which need us only update the policy and target networks after a fixed

d times of the update in critic. We slowly update the target networks with $d = 2$ in our experiments as equation (10) shows.

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{10}$$

We could limit the likelihood of repeating updates with respect to a temporary stable critic by sufficiently delaying the policy updates. Policy that update in a low frequency could have a higher bias but lower variance in the estimated value, this should help the algorithm to get better performance.

Improvement 3: Target Policy Smoothing Since the deterministic policy has the property that could overfit the peak curve in value estimation, a deterministic learning target is easy to overfit on the network input, which will cause a unacceptable high variance in the estimated results.

To reduce the high varance mentioned above, the paper introduce a regularization startegy of deep value learning to smooth the target policy, which enforces the notion that similar actions should have similar value. Therefore, we are proposed that fitting the value of a small area around the target action:

$$y = r + \mathbb{E}_\epsilon[Q_{\theta'}(s', \pi_{\Phi'}(s') + \epsilon)] \tag{11}$$

We can approximate this expectation over actions through adding a small random noise to the target policy and averaging over mini batches. Note that the added noise is clipped to let the target as close as possible to the original action.

$$\begin{aligned} y &= r + \gamma Q_{\theta'}(s', \pi_{Phi'}(s') + \epsilon) \\ \epsilon &\sim clip(N(0,\sigma), -c, c) \end{aligned} \tag{12}$$

Pseudo Code of TD3 Algorithm Based on the three improvements provided by paper [6] and the original DDPG algorithm, we would give the pseudo code of our TD3 algorithm as follows.

Algorithm 1: TD3 Algorithm

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_Φ with random parameters θ_1, θ_2 and Φ
Initialize target networks $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\Phi' \leftarrow \Phi$
Initialize replay buffer B
for $t = 1$ ***to*** T **do**
  Select action with exploration noise $a \sim \pi_\Phi(s) + \epsilon$, $\epsilon \sim N(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in B
  Sample mini-batch of N transitions (s, a, r, s') from B
  $\tilde{a} \leftarrow \pi_{\Phi'}(s') + \epsilon$, $\epsilon \sim clip(N(0,\sigma), -c, c)$
  $y = r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow argmin_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s,a))^2$
  if t mod d **then**
    Update Φ by the deterministic policy gradient:
    $\nabla_\Phi J(\Phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s,a)|_{a=\pi_\Phi(s)} \nabla_\Phi \pi_\Phi(s)$
    Update target networks: $\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$, $\Phi' \leftarrow \tau\Phi + (1-\tau)\Phi'$

4.3 Experiments Analysis

Settings In this part of our experiments, we first implement both **TD3 algorithm** according to the pseudo code provided above and its original version **DDPG** to test and compare the their performances. Then, we would analysis the advantages of TD3 based on the experimental results.

We totally trained our models for $1M$ time steps for each on *MacOS Big Sur* 11.2.3 $16GB$ platform, with the learning rate equals to $3e-4$ and the batch size equals to 256 for TD3 method. Also, we set

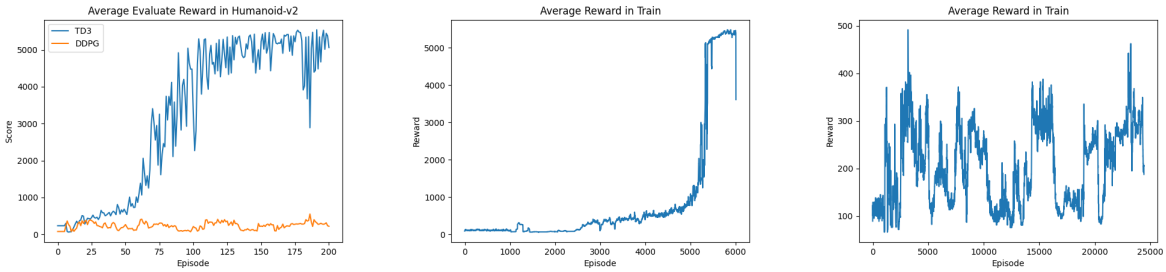
the frequency of the update in target network to be 2 and the update rate $\tau = 0.005$, and evaluate our model every $5e3$ steps, thus there are 201 evaluation episodes in total.

As for environmental settings, classic MuJoCo environments like *Hopper-v2*, *Humanoid-v2*, *HalfCheetah-v2* and *Ant-v2* are all supported, we have done multiple experiments on these environments and would provide results later. We use the *Humanoid-v2* to compare on the performance of TD3 and DDPG.

The two algorithms above are integrated in the sample *main.py* file. A command "*python main.py –env [env name] –policy [policy] –save_model*" could help you train and save model with TD3 algorithm in *Humanoid-v2* environment, and *–load_model* could help you load a trained model. More information could be found in *README.md*.

Average Evaluate Reward and Train Reward We first make comparison to the performance on TD3 and DDPG by analysing the average reward in evaluation, where the result are shown in Fig 9. Obviously, the general performance of TD3 is far more better than DDPG, which indicates that the small improvements on TD3 do have a great function. Note that the reward curve in evaluation has a high variance, the reason includes the introduced noise when we smooth the target policy, and the environmental properties in *Humanoid-v2*.

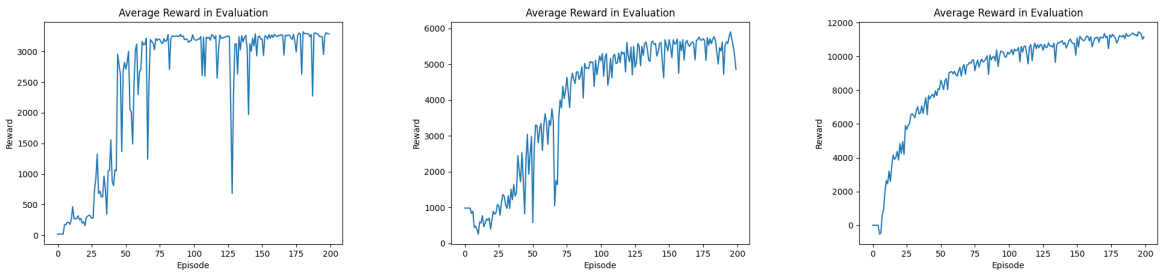
In Fig 9, we could also know that the convergence speed of TD3 in *Humanoid-v2* environment is rather slow, which may caused by several defects (with a extremely small reward) on the average operation in the statistic of train reward and the environment properties as well. Similarly, the convergence speed for DDPG is even slower, which seems to be non-convergent.



(a) Average Evaluation Reward (b) Average Train Reward in TD3 (c) Average Train Reward in DDPG

Fig. 9. Average Reward in Humanoid-v2

Other Environments We also do experiments on the other three MuJoCo environments to test the performance of our TD3 algorithm, the results are in Fig 10. We could find that TD3 algorithm converges well on all selected environments. However, different environment may induce different convergence speed and variance, and this phenomenon is caused by the differences on environment properties.



(a) Hopper-v2 (b) Ant-v2 (c) HalfCheetah-v2

Fig. 10. Average Evaluation Reward in Other Environments

5 Conclusion

In this paper, we do the model-free control experiments to evaluate two advanced RL algorithms including Dueling-Double-Deep Q-Network for value based method, which is based on Gym Atari environments with discrete action space, and TD3 for policy-based method, which is based on MuJoCo Simulator environments with continuous action space. To further test the advantages of the chosen algorithms, we compare them to the other methods in the same series, which are Dueling DDQN/Dueling DQN in Atari environments, and TD3/DDPG in MuJoCo environments.

Through the experimental results, the chosen methods all perform much better than the benchmarks, which proves that Dueling DDQN and TD3 algorithms all hold significant advantages in value-based/policy-based situations.

References

1. Mnih, V.: Artificial intelligence human-level control through deep reinforcement learning. NATURE - LONDON- (2015)
2. van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double q-learning. CoRR **abs/1509.06461** (2015)
3. Wang, Z., de Freitas, N., Lanctot, M.: Dueling network architectures for deep reinforcement learning. CoRR **abs/1511.06581** (2015)
4. Han, B.A., Yang, J.J.: Research on adaptive job shop scheduling problems based on dueling double dqn. IEEE Access **8** (2020) 186474–186495
5. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y., et al.: Policy gradient methods for reinforcement learning with function approximation. In: NIPs. Volume 99., Citeseer (1999) 1057–1063
6. Fujimoto, S., Hoof, H., Meger, D.: Addressing function approximation error in actor-critic methods. In: International Conference on Machine Learning, PMLR (2018) 1587–1596
7. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: International conference on machine learning, PMLR (2016) 1928–1937
8. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971 (2015)
9. Atari, O.G.: Openai gym atari game environments. https://gym.openai.com/envs/#atari Accessed June 6, 2021.
10. Atari, O.G.: Rom issues in openai gym atari game environments. https://github.com/openai/atari-py#roms Accessed June 6, 2021.
11. Mujuco, R.: Mujoco simulator in gym environment. http://www.mujoco.org Accessed June 6, 2021.
12. Qirenzhr, C.: Related settings in gym atari. https://blog.csdn.net/qq_27008079/article/details/100126060 Accessed June 6, 2021.
13. Liu98, C.E.: Pseudo code of ac series algorithms. https://blog.csdn.net/qq_41061258/article/details/107099860 Accessed June 6, 2021.